计 算 机 科 学 丛 书

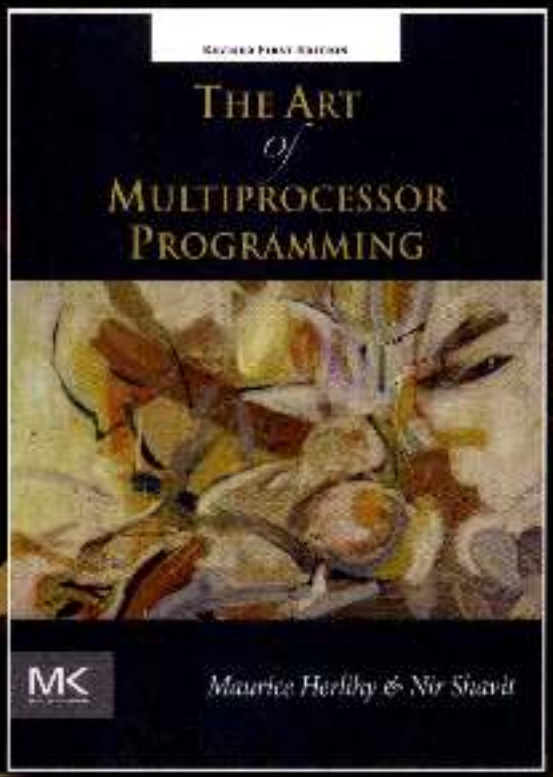
多处理器编程的艺术

（修订版）

（美） Maurice Herlihy 布朗大学 Nir Shavit 麻省理工学院 著
金海 胡侃 华中科技大学 译

The Art of Multiprocessor Programming

Revised First Edition



机械工业出版社
China Machine Press

多处理器编程的艺术（修订版）

The Art of Multiprocessor Programming Revised First Edition

工业界称为多核的多处理器机器正迅速地渗入计算的各个领域。多处理器编程要求理解新型计算原理、算法和编程工具，至今很少有人能够精通这门编程艺术。

现今，大多数工程技术人员都是通过艰辛的反复实践，永远有经验的朋友共学习多处理器编程技巧。这本最新的改进著作致力于改变这种状况，作者全面论述了多处理器编程的指导原则，介绍了编制高效的多处理器程序所必备的算法技术。了解本书所阐述的多处理器编程关键问题将使在校学生以及相关技术人员受益匪浅。

本书特色

- 本修订版结合2008年第1版出版以来课堂教学和读者反馈的错误和修改意见，对全书进行了多方面的修订和更新。
- 循序渐进地讲述共享存储器多线程编程的基础知识。
- 详细解释当今多处理器硬件对并发程序设计的支持方式。
- 全面考察主流的并发数据结构及其实施设计要点。
- 从简单的锁机制到最新的非等内存系统，独立、完整地阐述了同步技术。
- 给出大量利用Java并发工具包编写的可完全执行的Java实例。
- 对课程中采用其他程序设计语言和包（如C#、C及C++的Pthreads库）进行编程的相关背景知识以及硬件基础知识。

作者简介



Maurice Herlihy 哈佛大学的数学学士和麻省理工学院的计算机科学博士，目前为美国布朗大学计算机科学系教授，曾工作于卡内基-梅隆大学和DEC剑桥实验室。他是美国ACM会士，2003年分布式计算领域Dijkstra奖获得者。



Nir Shavit 以色列希伯来大学的计算机科学博士，目前为麻省理工学院电子工程和计算机科学系教授、以色列特拉维夫大学计算机科学系教授。1999-2011年期间，他担任Sun实验室的技术人员。

两位作者在2004年获得了理论计算机领域最高奖——哥德尔奖（Gödel Prize），2012年他们共享了分布式计算领域的Edsger W. Dijkstra奖。40多年来他们一起合作，从事并行和分布式计算教学和研发工作。



英文影印版
书号：978-7-111-41233-5
定价：79.00元

本书译自原版
The Art of Multiprocessor Programming Revised First Edition
许可Elsevier授权出版







上架指南：计算机/软件工程/多核

ISBN 978-7-111-41858-0

9 787111 418580 >

定价：69.00元

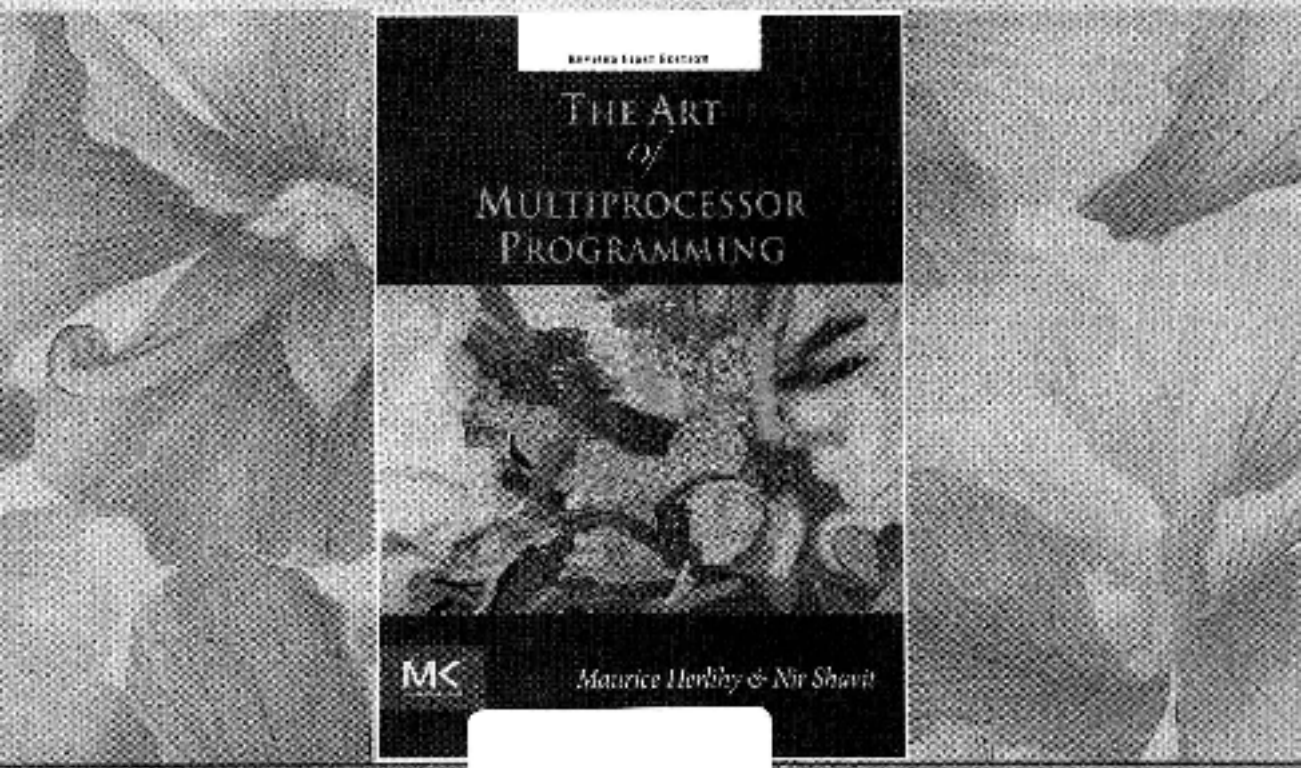
客服电话：(010) 88379201 88361066
购书电话：(010) 68326294 88379649 68995259
投稿电话：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

计 算 机 科 学 丛

多处理器编程的艺术
（修订版）

（美） Maurice Herlihy 布朗大学 Nir Shavit 麻省理工学院 著
金海 胡侃 华中科技大学 译

The Art of Multiprocessor Programming
Revised First Edition



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

多处理器编程的艺术（修订版）/（美）赫利希（Herlihy, M.），（美）夏菲特（Shavit, N.）著；金海，胡侃译．—北京：机械工业出版社，2013.4
（计算机科学丛书）
书名原文：The Art of Multiprocessor Programming, Revised First Edition

ISBN 978-7-111-41858-0

Ⅰ.多… Ⅱ.①赫… ②夏… ③金… ④胡… Ⅲ.微处理器—程序设计 Ⅳ.TP332

中国版本图书馆CIP数据核字（2013）第055426号

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2012-7897

本书从原理和实践两个方面全面系统地介绍了多处理器编程的指导原则，包含编制高效的多处理器程序所必备的算法技术。此外，附录提供了采用其他程序设计语言包（如C#、C及C++的PThreads库）进行编程的相关背景知识以及额外基础知识。

本书适合作为高等院校计算机及相关专业高年级本科生及研究生的教材，同时也可作为相关技术人员的参考书。

Maurice Herlihy and Nir Shavit: The Art of Multiprocessor Programming, Revised First Edition (ISBN 978-0-12-397337-5).

Copyright © 2012 by Elsevier Inc. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

Copyright © 2012 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书仅限中文版由Elsevier(Singapore)Pte Ltd. 授权机械工业出版社在中国大陆境内独家出版和发行。本版仅限在中国境内（不包括香港特别行政区、澳门特别行政区及台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

本书封底贴有Elsevier防伪标签，无标签者不得销售。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：迟振春
北京市荣盛彩印有限公司印刷
2013年5月第1版第1次印刷
185mm×260mm · 24.75印张
标准书号：ISBN 978-7-111-41858-0
定 价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010) 88378991 88361066 投稿热线：(010) 88379604
购书热线：(010) 68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅奠定了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减损。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨，建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作，而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

译者序

The Art of Multiprocessor Programming (Revised First Edition)

每逢我们在多处理器平台上进行编程时，往往会有这么一种感觉：即使已熟练掌握了系统提供的各种同步原语，作所编制的并行程序的实际性能似乎总有些差强人意，并不十分理想。究其原因，问题的根结在于多处理器编程应是一门科学和艺术完美结合的学科，若要在多处理器系统结构上编制出性能良好的并行程序，要求设计者不仅要精通多处理器系统结构、并行算法以及一些系统构建工具，还应能基于一种设计理念，充分发挥个人的想象空间，合理搭配这些知识和资源，从而和谐地构建完整的系统，使设计者能比底层硬件和操作系统"做得更好"，也就是说，在编写多处理器程序时，要能同时从宏观和微观两种角度分析问题，并能在这两种角度之间灵活地转换。

自20世纪中叶第一台通用电子计算机研制成功以来，程序的编制大多是基于顺序计算模型的，程序的执行过程是操作的有序序列。由于顺序计算机能够用图灵机精确地描述，因此顺序计算的编程能在一组易于理解且完备定义的抽象之上进行，而不需要了解底层的细节。近年来，尽管单处理器仍在发展，但由于指令级并行的开发空间正在减少，再加上散热等问题限制了时钟频率的继续提高，所以单处理器发展的速度正在减缓，这最终导致了起源于在单独一个晶片上设计多个内核的多处理器系统结构的出现。多处理器系统结构允许多个处理器执行同一个程序，共享同一程序的代码和地址空间，并利用并行技术来提高计算效率。在这种计算模型中，并发程序的执行可以看做是多个并发线程对一组共享对象的操作序列，为了在这种异步并发环境中获得更好的性能，底层系统结构的细节需要呈现给设计者。

作为一名优秀的程序设计员，在编写多处理器程序之前首先应弄清楚：多处理器计算机的能力和限制是什么；在异步并发计算模型中什么问题是可解决的，什么问题是不可解决的；是什么使得某些问题很难计算，而又使另一些问题容易计算。这要求设计者具备一定的多核并行计算理论基础知识，掌握多处理器系统结构上并发计算模型的可计算性理论及复杂性理论。其次，应掌握基本的多核平台上的并行程序设计技术，包括并行算法、同步原语以及各种多核系统结构。

Maurice Herlihy 教授和 Nir Shavit教授在并发程序设计领域具有很深的造诣，并拥有40年以上一起从事并发程序设计教学的合作经验，他们对多处理器并行程序设计技术做出了巨大的贡献，并因此而成为2004年ACM/EATCS哥德尔奖和2012年分布式计算领域Dijkstra奖的共同获得者。这本由他们合著的专著致力于解决如何采用更好的并行算法来克服多核并发程序并行度低的问题。

Amdahl定律早已明确地告诉我们，从程序本身可获得的并行度是有限的，加速比的提高主要取决于程序中必须增加的串行执行部分，而这部分又往往包含着具有相对较高开销的通信和协作。因此，在多处理器系统结构上，如何提高程序中必须串行部分的并行度，以及降低并行处理器中远程访问的时延是我们目前面临的两大技术挑战。这些问题的有效解决，必须依靠软件技术和硬件技术的改进和发展。本书则侧重于对前一个挑战的研究。

作者先从宏观的抽象角度出发，在一个理想化的共享存储器系统结构中研究各种并行算

法的可计算性及正确性。通过对这些经典算法的非理分析，向读者揭示了现代协作范例和并发数据结构中所隐藏的核心思想，使读者学会如何分析饥饿和死锁等微妙的活性问题，深层次地研究现代硬件同步原语所应具有的能力及其特性。随后，从微观的实际角度出发，针对当今主流的多处理器系统结构，设计了一系列完美高效的并行算法及并发数据结构，并对各种算法的效率及其机理进行了分析。所有的设计全部采用Java程序设计语言详细地描述，可以非常容易地将它们扩展到实际应用中。

本书的前6章讲述了多处理器程序设计的原理部分，着重于异步并发环境中的可计算性问题，借助于一个理想化的计算模型来阐述如何描述和证明并行程序的实际执行行为。由于其自身的特点，多处理器程序的正确性要比顺序执行程序的正确性复杂得多，书中为我们展现了一系列不同的辅助论证工具，令人有耳目一新之感。随后的11章阐述了多处理器程序设计的实践部分。由于在多处理器环境中编写程序时，底层系统结构的细节并不像编写顺序程序那样被完全隐藏在一种编程抽象中，因此，本书附录B介绍了多处理器硬件的基础知识。最后的第18章介绍了当今并发问题研究中最先进的事务方法，可以预言这种方法在今后的研究中将会越来越重要。

感谢王辰飞博士在本书第13～18章翻译中所做的工作，感谢胡雅婷、袁昊昊、耿玮、蔡硕、张亮同学在本书翻译的初始资料整理中所给予的帮助。

前　言

The Art of Multiprocessor Programming (Revised First Edition)

本书可以作为高年级本科生的教材，也可以作为相关技术人员的参考书。

读者应具备一定的离散数学基础知识，能够理解“大O”符号的含义，以及它在NP完全问题中所起的作用；熟悉计算机系统的基本组成部件，如处理器、线程、高速缓存等；为了能够理解书中的实例，还需要具备初步的Java知识。（在使用这些高级程序设计语言之前，本书例述了语言的相关特征。）书中提供两个附录以供读者参考：附录A包含程序设计语言的相关知识，附录B给出了多处理器硬件系统结构的相关内容。

本书前三分之一涵盖并发程序设计的基本原理，例述并发程序设计的编程思想。就像掌握汽车驾驶技术、烹饪食物和品尝鱼子酱一样，并发思维也需要培养，需要适当的努力才能学好。希望立刻动手编程的读者可以跳过这部分的大多数内容，但仍需阅读第2章及第3章的内容，这两章包含了理解本书其他部分所必不可少的基本知识。

在原理部分中，首先讨论了经典的互斥问题（第2章），包括诸如公平性和无锁这样的基本概念，这对于理解并发程序设计的难点尤为重要。然后，结合并发执行和并发设计中可能出现的各种情形和并发环境，给出了并发程序正确性的定义（第3章）。接着，研究了对并发计算至关重要的共享寄存器的性质（第4章）。最后，介绍了几种为实现高并发性数据结构而使用的同步原语（第5、6章）。

对于每一位渴望真正掌握多处理器编程艺术的程序设计人员来说，花上一定的时间去解决本书第一部分所提及的问题是很有必要的。虽然这些问题都是理想化的，但它们为编写高效的多处理器程序提供了非常有益的编程思想。尤为重要的是，通过在问题解决中获取的思维方式，能够避免出现那些初次编写并发程序的设计人员普遍易犯的错误。

接下来的三分之二讲述并发程序设计的具体实践。每章都有一个相应主题，证明一种特定的程序设计模式或者算法技巧。第7章从系统和语言这两个不同的抽象层面，讨论争用及自旋锁的概念，强调了底层系统结构的重要性，指出对于自旋锁性能的理解必须建立在对多处理器层次存储结构充分理解的基础上。第8章涉及等待及管程锁的概念，这是一种常用的同步用语（特别是在Java中）。第16章包括并行性及工作窃取问题，第17章则介绍了障碍技术。这种技术往往在具有并发结构的应用中得以广泛的使用。

其他章节讲述各种类型的并发数据结构。它们均以第9章的概念为基础，因此建议读者在阅读其他章节之前首先阅读第9章的内容。该章采用链表结构来阐明各种类型的同步模式，包括粗粒度锁、细粒度锁及无锁结构。第10章则借助于FIFO队列说明在使用同步原语时可能出现的ABA问题，第11章通过栈描述了一种重要的同步模式——淘汰，第13章通过哈希映射例述如何利用同有并行进行算法设计。高效率的并行查找技术则借助于跳表来阐述（第14章），而如何通过降低正确性标准来获得更高的性能则通过优先级队列进行了阐述（第15章）。

最后，第18章介绍了在并发问题的研究中新出现的事务方法，可以确信这种方法在不远的将来会变得越来越重要。

并发性的重要性还没有得到人们的广泛认可。在此，引用《纽约时报》1989年关于IBM PC中新型操作系统的一段评论：

真正的并发（当你唤醒并使用另一个程序时原来的程序仍继续运行）是非常令人振奋的。但对于普通使用者来说用处却很小。您能有几个程序在执行时需要花费数秒甚至更多的时间呢？

致谢

感谢Doug Lea、Michael Scott、Ron Rivest、Tom Corman、Radia Perlman、George Varghese和Michael Sipser在本书出版过程中所做的努力和给予的帮助。

感谢在本书的起草和修订过程中，向我们提供了大量宝贵意见的所有学生、同事和朋友：Yehuda Afek, Shai Ber, Martin Buchholz, Vladimir Budovsky, Christian Cachin, Cliff Click, Yoav Cohen, Dave Dice, Alexandra Fedorova, Pascal Felber, Christof Fetzer, Shafi Goldwasser, Rachid Guerraoui, Tim Harris, Danny Hendler, Maor Hizkiev, Eric Koskinen, Christos Kozyrakis, Edya Ladan, Doug Lea, Oren Lederman, Pierre Leone, Yossi Lev, Wei Lu, Victor Luchangco, Virendra Marathe, John Mellor-Crummey, Mark Moir, Dan Nussbaum, Kiran Pamnany, Ben Pere, Torvald Riegel, Vijay Saraswat, Bill Scherer, Warren Schudy, Michael Scott, Ori Shalev, Marc Shapiro, Yotam Soen, Ralf Suckow, Seth Syberg, Alex Weiss和Zhenyuan Zhan。同时，也向在这里没有提及的许多朋友表示谢意和感谢。

感谢Mark Moir、Steve Heller和Sun公司的Scalable Synchronization小组成员，他们为本书的撰写提供了巨大的支持和帮助。

感谢提出勘误帮助我们改进本书的所有朋友：Rajeev Alur, Matthew Allen, Karolos Antoniadis, Cristina Basescu, Liran Barsisa, Igor Berman, Konstantin Boudnik, Bjoern Brandenburg, Martin Buchholz, Kyle Cackett, Mario Calha, Michael Champigny, Neill Clift, Eran Cohen, Daniel B. Curtis, Gil Danziger, Venkat Dhinakaran, David Dice, Wan Fokkink, David Fort, Robert P. Goddard, Brian Goetz, Bart Golsteijn, K. Gopinath, Enes Goktas, Jason T. Greene, Dan Grossman, Tim Halloran, Muhammad Amber Hassaan, Matt Hayes, Francis Hools, Ben Horowitz, Baruk Itkin, Paulo Janotti, Kyungho Jeon, Ahmed Khademzadeh, Irena Karlinsky, Habib Khan, Omar Khan, Namhyung Kim, Guy Korland, Sergey Kotov, Doug Lea, Yossi Lev, Adam MacBeth, Kevin Marth, Adam Morrison, Adam Weinstock, Mike Maloney, Tim McIver, Sergejs Melderis, Bartosz Milewski, Mark Moir, Adam Morrison, Victor Luchangco, Jose Pedro Oliveira, Dale Parson, Jonathan Perry, Amir Pnueli, Pat Quillen, Binoy Ravindran, Roei Raviv, Sudarshan Raghunathan, Jean-Paul Rigault, Michael Rueppel, Mohamed M. Saad, Assaf Schuster, Marc Shapiro, Nathar Shah, Huang-Ti Shih, Joseph P. Skudlarek, James Stout, Mark Summerfield, Deqing Sun, Seth Syberg, Fuad Tabba, Binil Thomas, John A Trono, Thomas Weibel, Adam Weinstock, Jaeheon Yi, Zhenyuan Zhao, Ruiwen Zuo, Chong Xing。

对于技术人员

技术人员对直接解决手边问题的技术感兴趣，针对技术人员的短期课程应该包括以下内容：

强调Amdahl定律及其作用的第1章，第2章的2.1、2.2、2.3和2.6节；了解关于不可能性证明的作用的2.8节，第3章的3.1、3.2、3.4、3.5、3.7、3.8和3.9节。

第7章的7.1~7.6和7.9节。对于一些技术人员来说，可能熟悉讨论管程锁和可重入锁的第8章，跳过有关信号量的8.5节。

第9章，第10章（除10.7节外），第11章的11.1和11.2节。

第13和14章，第16章（除16.5节外），第17章的17.1~17.3节。

对于非计算机专业学生

非计算机专业学生对多处理器编程的基础知识以及可能对他们自己领域有用的技术感兴趣，针对非计算机专业学生的中期课程应该包括以下内容：

强调Amdahl定律及其作用的第1章。第2章的2.1、2.2、2.3、2.5和2.6节，了解关于不可能性证明的作用的2.8节。第3章的3.1~3.5、3.7~3.9节。

第4章的4.1和4.2节，第5章，了解关于一致性的通用性的第6章。

第7章的7.1~7.6和7.9节，第8章。

第9章，第10章（除10.7节外），第11章。

第13和14章，第16章，第17章的17.1~17.3节，第18章。

对于计算机专业学生

针对计算机专业学生（高年级本科生或研究生）的一学期课程应该包括以下内容：

第1章，第2章（2.7节可选），第3章（3.6节可选），第4章，第5章，第6章。在学习第7章之前，有必要复习一下多处理器系统结构的基础知识（附录B）。

第7章（7.7和7.8节可选），第8章（如果学生不熟悉Java管程知识），第9章，第10章（10.7节可选），第11章，第12章（12.7~12.9节可选），第13章，第14章。第15章可选。第16章，第17章。第18章可选。

本书的配套网站（http://store.elsevier.com/product.jsp?isbn=9780123973375）为一学期课程提供了授课PPT。

推荐阅读





大规模并行处理器程序设计（英文版·第2版）

作者：David B. Kirk Wen-mei W. Hwu

ISBN：978-7-111-41629-6 定价：79.00元

云计算与分布式系统：从并行处理到物联网

作者：Kai Hwang Geoffrey C. Fox Jack Dongarra 译者：武永卫 等 审校：黄铠

ISBN：978-7-111-41065-2 定价：65.00元

并行程序设计导论

作者：Peter S.Pacheco 译者：清华大学 ISBN：978-7-111-36734-0 定价：45.00元

并行程序设计原理

作者：[illegible] 译者：[illegible] ISBN：978-7-111-27275- 定价：45.00元

高性能CUDA应用设计与开发：方法与最佳实践

作者：Rob Farber 译者：于玉龙 [illegible] ISBN：978-7-111-47446-0 定价：59.00元

GPU高性能编程CUDA实战

作者：[illegible] 译者：[illegible] ISBN：978-7-111-[illegible] 定价：[illegible]

出版者的话
译者序
前言
教学建议

第1章 引言 ……1
1.1 共享对象和同步 ……2
1.2 生活实例 ……4
1.2.1 互斥特性 ……6
1.2.2 通信 ……7
1.3 生产者-消费者问题 ……7
1.4 读者/写者问题 ……9
1.5 并行的困境 ……9
1.6 并行程序设计 ……11
1.7 本章注释 ……11
1.8 习题 ……11

第一部分 原 理

第2章 互斥 ……14
2.1 时间 ……14
2.2 临界区 ……14
2.3 双线程解决方案 ……16
2.3.1 LockOne类 ……16
2.3.2 LockTwo类 ……17
2.3.3 Peterson锁 ……18
2.4 过滤锁 ……19
2.5 公平性 ……21
2.6 Bakery算法 ……21
2.7 有界时间戳 ……23
2.8 存储单元数量的下界 ……25
2.9 本章注释 ……27
2.10 习题 ……28
第3章 并发对象 ……31
3.1 并发性与正确性 ……31
3.2 顺序对象 ……33
3.3 静态一致性 ……34
3.4 顺序一致性 ……35
3.5 可线性化性 ……38
3.5.1 可线性化点 ……38
3.5.2 评析 ……38
3.6 形式化定义 ……38
3.6.1 可线性化性 ……39
3.6.2 可线性化性的复合性 ……40
3.6.3 非阻塞特性 ……40
3.7 满足条件 ……41
3.8 Java存储器模型 ……43
3.8.1 锁和同步块 ……44
3.8.2 volatile域 ……44
3.8.3 final域 ……44
3.9 评析 ……45
3.10 本章注释 ……46
3.11 习题 ……46
第4章 共享存储器基础 ……50
4.1 寄存器空间 ……50
4.2 寄存器构造 ……54
4.2.1 MRSW安全寄存器 ……55
4.2.2 MRSW规则布尔寄存器 ……55
4.2.3 M值MRSW规则寄存器 ……56
4.2.4 SRSW原子寄存器 ……57
4.2.5 MRSW原子寄存器 ……59
4.2.6 MRMW原子寄存器 ……60
4.3 原子快照 ……62
4.3.1 无碰时快照 ……63
4.3.2 无等待快照 ……64
4.3.3 正确性证明 ……66
4.4 本章注释 ……67
4.5 习题 ……67
第5章 同步原子操作的相关能力 ……70
5.1 一致数 ……70
5.2 原子寄存器 ……72

5.3 一致性协议 ……74
5.4 FIFO队列 ……74
5.5 多重赋值对象 ……77
5.6 读 改 写操作 ……79
5.7 Common2 RMW操作 ……80
5.8 compareAndSet()操作 ……81
5.9 本章注释 ……82
5.10 习题 ……83
第6章 一致性的通用性 ……87
6.1 引言 ……87
6.2 通用性 ……88
6.3 一种通用的无锁构造 ……88
6.4 一种通用的无等待构造 ……91
6.5 本章注释 ……95
6.6 习题 ……95

第二部分 实 践

第7章 自旋锁与争用 ……98
7.1 实际问题 ……98
7.2 测试-设置锁 ……100
7.3 再论基于TAS的自旋锁 ……102
7.4 指数后退 ……102
7.5 队列锁 ……104
7.5.1 基于数组的锁 ……104
7.5.2 CLH队列锁 ……106
7.5.3 MCS队列锁 ……107
7.6 时限队列锁 ……110
7.7 复合锁 ……112
7.8 层次锁 ……118
7.8.1 层次后退锁 ……118
7.8.2 层次CLH队列锁 ……119
7.9 由一个锁管理所有的锁 ……123
7.10 本章注释 ……123
7.11 习题 ……124
第8章 管程和阻塞同步 ……126
8.1 引言 ……126
8.2 等待集和条件 ……126
8.2.1 条件 ……127
8.2.2 唤醒丢失问题 ……130
8.3 读者-写者锁 ……131
8.3.1 简单的读者-写者锁 ……131
8.3.2 公平的读者-写者锁 ……132
8.4 我们的可重入锁 ……134
8.5 信号量 ……135
8.6 本章注释 ……136
8.7 习题 ……136
第9章 链表：锁的作用 ……139
9.1 引言 ……139
9.2 基于链表的集合 ……140
9.3 并发推理 ……141
9.4 粗粒度同步 ……142
9.5 细粒度同步 ……143
9.6 乐观同步 ……146
9.7 惰性同步 ……149
9.8 非阻塞同步 ……153
9.9 讨论 ……157
9.10 本章注释 ……157
9.11 习题 ……158
第10章 并行队列和ABA问题 ……159
10.1 引言 ……159
10.2 队列 ……160
10.3 部分有界队列 ……160
10.4 完全无界队列 ……163
10.5 无锁的无界队列 ……164
10.6 内存回收和ABA问题 ……166
10.7 双重数据结构 ……170
10.8 本章注释 ……172
10.9 习题 ……172
第11章 并发栈和消除 ……174
11.1 引言 ……174
11.2 无锁的无界栈 ……174
11.3 消除 ……176
11.4 后退消除栈 ……176
11.4.1 无锁交换机 ……177
11.4.2 消除数组 ……179
11.5 本章注释 ……181
11.6 习题 ……181
第12章 计数、排序和分布式协作 ……184
12.1 引言 ……184

12.2 共享计数 ……184
12.3 软件组合 ……185
12.3.1 概述 ……185
12.3.2 一个扩展实例 ……190
12.3.3 性能和伸缩性 ……191
12.4 静态一致性和计数器 ……192
12.5 计数网 ……192
12.5.1 可计数网 ……193
12.5.2 双调计数网 ……194
12.5.3 性能和流水线 ……201
12.6 前缀和 ……201
12.7 并行排序 ……204
12.8 排序网 ……204
12.9 样本排序 ……207
12.10 分布式操作 ……208
12.11 本章注释 ……208
12.12 习题 ……209
第13章 并发哈希和固有并行 ……212
13.1 引言 ……212
13.2 封闭地址哈希集 ……213
13.2.1 粗粒度哈希集 ……214
13.2.2 空间分带哈希集 ……215
13.2.3 细粒度哈希集 ……217
13.3 无锁哈希集 ……219
13.3.1 逆序有序划分 ……219
13.3.2 BucketList类 ……222
13.3.3 LockFreeHashSet<T>类 ……223
13.4 开放地址哈希集 ……225
13.4.1 Cuckoo哈希 ……225
13.4.2 并发Cuckoo哈希 ……226
13.4.3 空间分带的并发Cuckoo哈希 ……230
13.4.4 细粒度的并发Cuckoo哈希集 ……231
13.5 本章注释 ……233
13.6 习题 ……234
第14章 跳表和平衡查找 ……235
14.1 引言 ……235
14.2 顺序跳表 ……235
14.3 基于锁的并发跳表 ……236
14.3.1 简介 ……236
14.3.2 算法 ……238
14.4 无锁并发跳表 ……243
14.4.1 简介 ……243
14.4.2 算法细节 ……245
14.5 并发跳表 ……251
14.6 本章注释 ……251
14.7 习题 ……251
第15章 优先级队列 ……253
15.1 引言 ……253
15.2 基于数组的有界优先级队列 ……253
15.3 基于树的有界优先级队列 ……254
15.4 基于堆的无界优先级队列 ……256
15.4.1 顺序堆 ……256
15.4.2 并发堆 ……258
15.5 基于跳表的无界优先级队列 ……262
15.6 本章注释 ……264
15.7 习题 ……265
第16章 异步计算、调度和工作分配 ……266
16.1 引言 ……266
16.2 并行分析 ……271
16.3 多处理器的实际调度 ……273
16.4 工作分配 ……274
16.4.1 工作窃取 ……275
16.4.2 原则和多道程序设计 ……275
16.5 工作窃取双端队列 ……276
16.5.1 有界工作窃取双端队列 ……276
16.5.2 无界工作窃取双端队列 ……279
16.5.3 工作平衡 ……282
16.6 本章注释 ……283
16.7 习题 ……284
第17章 障碍 ……287
17.1 引言 ……287
17.2 障碍实现 ……288
17.3 语义依向障碍 ……288
17.4 组合树障碍 ……289
17.5 静态树障碍 ……291
17.6 终止检测障碍 ……293
17.7 本章注释 ……295
17.8 习题 ……296

第18章 事务内存 ……302
18.1 引言 ……302
18.1.1 关于锁的问题 ……302
18.1.2 关于compareAndSet()的问题 ……303
18.1.3 关于复合性的问题 ……304
18.1.4 我们能做什么 ……305
18.2 事务和原子性 ……305
18.3 软事务内存 ……306
18.3.1 事务和事务线程 ……309
18.3.2 僵尸事务和一致性 ……310
18.3.3 原子对象 ……311
18.3.4 如何演进 ……311
18.3.5 争用管理器 ……312
18.3.6 原子对象的实现 ……314
18.3.7 无干扰原子对象 ……315
18.3.8 基于锁的原子对象 ……318
18.4 硬事务内存 ……323
18.4.1 缓存一致性 ……324
18.4.2 事务缓存一致性 ……324
18.4.3 改进 ……325
18.5 本章注释 ……325
18.6 习题 ……326

第三部分 附 录

附录A 软件基础 ……328
附录B 硬件基础 ……340
参考文献 ……350
索引 ……359

引　言

计算机产业正在经历着一场重大的结构重组和巨变，在没有其他变革之前，这个过程无疑将会继续进行。主要的芯片制造商，至少是现在，都纷纷放弃尝试研制速度更快的处理器。虽然摩尔定律仍旧适用：每年集成在同样大小空间中的晶体管数越来越多，然而，由于散热问题难于解决，它们的时钟速度无法继续得到提高。取而代之的做法是，制造商开始转向"多核"系统结构的研制，由多个处理器（多核）共享硬件高速缓存直接进行通信。通过将多个处理器同时分配给单一任务以获得更高的并行性，从而提高计算的效率。

多处理器系统结构的普及对计算机软件的发展带来了深刻的影响。在至今日以前，技术的进步意味着时钟速度的提升，时钟本身的加速导致了软件执行效率的提高。今天，这种搭便车的现象已不复存在，技术的进步不再指时钟速度的提升而是指并行度的提升，并行问题已经成为现代计算机科学的主要挑战。

本书着重讲述共享存储器通信方式下的多处理器编程技术。通常称这样的系统为共享存储器的多处理器，现在称之为多核。在各种规模的多处理器系统中都存在着不同的编程挑战——对于小规模的系统来说，需要协调单个芯片内的多个处理器对同一个共享存储单元的访问；对于大规模系统来说，需要协调一台超级计算机中各个处理器之间的数据路由。其次，现代计算机系统所固有的异步等待也给多处理器编程带来了挑战：在没有任何警示的情形下，系统的活动可以被各种不同的事件中止或延迟，例如中断、抢占、cache缺失和系统故障等。这种延迟现象本身是无法预测的，时延的长短也是不确定的：cache缺失可以造成不到十条指令执行时间的时延，页故障可能造成几百万条指令执行时间的时延，而操作系统抢占则会导致多达上亿条指令执行时间的时延。

本书从原理和实践这两个互补的方面阐述多处理器的程序设计。原理部分着重于可计算性理论：理解异步并发环境中的可计算问题。借助于一个理想化的计算模型，对异步并发环境中什么是可解的这一问题进行了深入研究。在这个模型中，多个并发线程对一组共享对象进行操作，这些并发线程的对象操作序列被称为并发程序或并发算法。该模型实质上也正是Java[TM]、C#及C++线程库中所采用的计算模型。

令人不可思议的是，的确存在一些易于说明的共享对象，我们无法采用任何并发算法来实现。因此，在编写多处理器程序之前弄清楚什么问题不能用计算机解决是十分重要的。大多数困扰多处理器程序员的问题都源自于计算模型自身的限制，所以，对并发共享存储器模型中可计算性理论的理解是学习多处理器编程必不可少的一个环节。书中与原理相关的章节向读者展现了各种各样的可计算问题，以帮助读者尽快地了解异步可计算性理论，同时也讲述了如何通过硬件和软件机制来解决这些问题。

理解可计算性的关键在于描述和证明特定程序的实际执行行为，更准确地说，即程序正确性问题。由于其自身的特点，多处理器程序的正确性比顺序程序的正确性更为复杂。因此，需要一系列不同的辅助论证工具来证明并发程序的正确性，甚至有可能只是为了"非形式化地证明"程序正确性（实际上程序员往往这么做）。顺序程序的正确性主要关心程序的各种安

全特性，安全性说明了“不好的事件”绝不会发生。例如，即使在断电时，交通指示灯也决不给任何方向显示绿灯。同样，并发程序的正确性也需要考虑安全性，但要考虑如何确保多个并发线程在各种可能的交互情形下的安全性，这使问题的解决变得非常上加难。另外，还要考虑一个同样重要的因素，并发程序的正确性包括了各种各样的活性特性，而这种特性是顺序程序执行中所不会出现的。所谓活性，是指一个特定的“好的事件”一定会发生。例如，红色指示灯最终一定会变为绿灯。本书原理部分的最终目标就是要引入一系列分析推理并发程序的衡量标准和方法，为接下来研究现实对象和程序的正确性奠定基础。

本书的第二部分阐述多处理器程序设计的具体实践，着重于并发程序性能的分析。多处理器并发算法的性能分析与顺序算法的性能分析在风格上完全不同。顺序程序设计是基于一组易于理解且完备定义的抽象来进行的。编写顺序程序时，不需要了解底层的详细细节，例如，在硬盘和内存之间如何交换页面，在层次结构的高速缓存中如何移进/移出那些较小的内存单位。这种复杂的存储器层次结构实质上对程序员是不可见的，它被隐藏在一种完全的编程抽象之中。

然而在多处理器环境下，这种编程抽象被打破了，至少从性能角度来讲应该这样做。为了获得足够好的性能，程序设计人员有时需要比底层存储器系统“做得更好”，他们编写的程序代码可能让那些不熟悉多处理器系统结构的人感到莫名其妙，或许某一天，并发系统结构会像今天的顺序系统结构一样支持完全的抽象，然而到那时，程序设计人员早已了解这种新的系统结构了。

本书的原理部分讲述了一组共享对象和编程工具，着眼于每种对象和工具自身的能力，并借助于它们引出一些高层次的问题，用自旋锁来说明争用，用链表阐述数据结构设计中锁的作用等。这些问题对程序的性能都有着重要的影响，希望读者能充分理解它们，并能将所理解的内容应用于日后具体的多处理器系统设计中。最后，通过讨论诸如事务内存这种目前最先进的技术，作为本书的结束。

下面简要说明本书的写作风格。尽管有很多合适的语言可供选择，但本书仍采用了Java程序设计语言。当然，有大量的理由可以解释为何要做出这种选择，然而这样的话题还是更适于在闲暇时讨论！附录解释了Java所支持的一些概念在其他的常用语言或库中是如何表示的，同时也介绍了关于多处理器硬件的一些基础知识。始终牢记，我们尽量避免列出关于程序和算法性能的具体数据，而是从一般情形来考虑。这样做的理由是：多处理器系统之间差异很大，在一台机器上工作良好的并发程序并不代表在另一台机器上也有同样的表现，紧密结合一般情形能够保证本书所陈述的结论具有更加长久的有效性。

在每一章的末尾都提供了相关文献的引用，读者可以根据参考文献目录找到相应内容以便进一步阅读。此外，每章都提供了一些习题，读者可以据此检查自己对知识的理解程度。

1.1 共享对象和同步

在开始新工作的第一天，老板要求在一台能够支持10个并发线程的并行机上编写出查找$1\sim10^{20}$之间素数的程序（不要考虑为什么这样做）。机器是按照分钟租用的。程序运行时间越长，花费也就越大。如果想给老板留下一个不错的印象，应该怎样去做？

在最初的尝试中，可能会为每个线程分配一个大小相等的输入域。各个线程分别找出10^{20}个数字内的素数，如图1-1所示。这种方法可能会由于一个简单但很重要的原因而最终导致失败，那就是相同大小的输入范围并不意味着相同的工作量。素数的分布是不均匀的：在$1\sim10^{20}$之间有很多素数，但分布在$9\times10^{19}\sim10^{20}$的素数却非常少。更为糟糕的是，整个范围内不同

素数的计算时间也是不相同的：判断一个较大的数是否为素数通常要比判断较小的数所花费 [3]
的时间更长。简而言之，没有理由认为这种方式能够使得整个工作是由所有的线程平均承担完成的，甚至也不清楚哪个线程承担的工作最多。

```
1  void primePrint {
2    int i = ThreadID.get();  // thread IDs are in {0..9}
3    int block = power(10, 9);
4    for (int j = (i * block) + 1; j <= (i + 1)* block; j++) {
5      if (isPrime(j))
6        print(j);
7    }
8  }
```

图1-1 通过等分输入域达到负载平衡。对{0..9}中的每个线程分配同样大小的输入子集

在线程间划分工作的另一种可行方案就是为每个线程一次分配一个整数（图1-2）。当一个线程结束对该整数的判断后，再次请求分配另一个整数。为此，需要引入一个共享计数器对象。该对象将一个整型值封装起来，线程通过调用getAndIncrement()方法对该整型值进行自增操作，并返回未被增加前的先前值。

```
1  Counter counter = new Counter(1);  // shared by all threads
2  void primePrint {
3    long i = 0;
4    long limit = power(10, 10);
5    while (i < limit) {                // loop until all numbers taken
6      i = counter.getAndIncrement(); // take next untaken number
7      if (isPrime(i))
8        print(i);
9    }
10 }
```

图1-2 通过共享计数器达到负载平衡。每个线程对动态确定的数字进行判断

图1-3是Counter对象的Java实现。该计数器由单线程调用时效果很好，但由多线程共享使用时却会出现错误。其问题的根本就在于语句

```
return value++;
```

实质上是下面几行代码的缩写方式：

```
long temp = value;
value = temp + 1;
return temp;
```

在这段代码中，value是对象Counter的一个域，它被所有的线程所共享。但是，每个线程都有一个它自己的本地拷贝temp，该拷贝是线程的局部变量。 [4]

假设线程*A*和线程*B*同时调用Counter的getAndIncrement()方法。那么，*A*和*B*有可能同时从value中读入1，然后分别将各自的局部变量temp设置为1，再将value设置为2。最终，*A*和*B*都返回了value的先前值1。显然，这种情形并不是我们所期望的：对计数器getAndIncrement()方法的并发调用返回了同一个值。我们希望不同的线程返回不同的值。事实上，还有可能出现更坏的情形：一个线程从value中读入了1，在它将value置为2之前，另一个线程执行了多次增量循环，读入1设置为2，接着读入2设置为3。当第一个线程最终完

成共增量操作并将value设置为2时。它实际上是将value的值从3改回到2。

```
1  public class Counter {
2    private long value;    // counter starts at one
3    public Counter(int i) {    // constructor initializes counter
4      value = 1;
5    }
6    public long getAndIncrement() { // increment, returning prior value
7      return value++;
8    }
9  }
```

图1-3 共享计数器的实现

出现上述问题的根本原因在于对计数器值的增加需要在共享变量上执行两种不同类型的操作：将value的值读入temp变量，并将temp的值写入Counter对象。

类似的情形在现实生活中同样也会出现。当在走廊中行走时，需要绕过向你迎面走来的人。你可能发现那一瞬间自己一会儿向右，一会儿向左，这么来回好几次，因为另一个人也在试图这么做以避免与你碰撞。有时成功避开了，但有时还是撞上了。实际上，正如在后面的章节中将要讲述的那样，这种碰撞在很多时候是无法避免的。[⊖]直观上来看，你和向你迎面走来的人都在做两件事：观察（"读"）对方当前的位置，然后移向（"写"）另一边。然而问题是，当你在读对方的位置时，无法知道他下一秒是继续待着还是移动躲闪。你和对面的这个陌生人必须决定谁从左边走谁从右边走。同样，共享计数器的各个线程也需要决定谁先使用谁后使用。

在第5章将会看到，现代的多处理器硬件通常都提供了特殊的读-改-写指令，允许线程以原子的硬件操作来读、写或修改存储器的值。对于上述的Counter对象，可以采用这种硬件
[5] 方式原子地完成计数器值的自增。

同样，通过在软件（只使用读、写指令）中保证一个时刻只有一个线程在执行读/写操作序列，也可以获得这种原子的行为效果。这种确保一个时刻只允许一个线程执行特定代码段的问题称为互斥问题，它是多处理器程序设计中经典的协作问题之一。

在实际编程中并不需要由自己来设计互斥算法（而是调用库）。但是，深入地理解互斥算法的实现是从全局上掌握并发计算的基础。同样，对于死锁、有界公平性、不同于非阻塞方式的阻塞同步等这些背后很重要问题的具体解决方法，也需要进行深入的研究。

1.2 生活实例

并发协作问题（如互斥）应该当作实际的具体问题来处理，而不应看作是一种编程训练。本书通过一些现实生活实例阐述基本的并发问题。像大多数讲述这些实例的其他作者一样，我们也是在原有的情节上重述这些故事（见本章末尾的注释）。

Alice和Bob是一对邻居，他们共用一个院子。Alice养了一只猫而Bob养了一只小狗。这两只小宠物都喜欢在院子里跑来跑去，然而（自然地）它们总是不能融洽地相处。在发生了一些不愉快的事情后，Alice和Bob决定采取措施，让两只小宠物不同时出现在院子里。显然，应该排除不许任一只动物待在院子里的做法。

应该怎样去做呢？Alice和Bob先要约定一种相互都可以接受的过程以决定他们该做什么。

⊖ 无法使用类似"总是靠右行"这种协作的策略，因为对方有可能是英国人。

这种约定称为协作协议（简称协议）。

由于院子很大，Alice无法从窗户观察到Bob的小狗是不是在院子里。当然，她可以到Bob家敲门进去看看，但这太浪费时间，况且下雨怎么办呢？Alice也可以站在窗户边冲着Bob的屋子大声喊叫："Bob！我的猫可以出来了吗？"然而，问题是Bob有可能听不见，他或许在看电视，或许在招待他的女朋友，也可能出去买狗食了。他们两人同样也可以尝试通电话，但也会出现问题，例如Bob正在洗澡或给电话换电池等。

于是，Alice想出了一个好主意。她在Bob家的窗台上放了几个空啤酒罐（如图1-4所示），用绳子将它们一个个地绑起来，并将绳子的一端系在自己的屋子里。Bob也按照同样的方法在Alice家的窗台上安放啤酒罐。当Alice想给Bob发出信号时，则猛拉绳子打翻其中一个啤酒罐。若Bob发现一个啤酒罐被打翻，则重新摆好它。

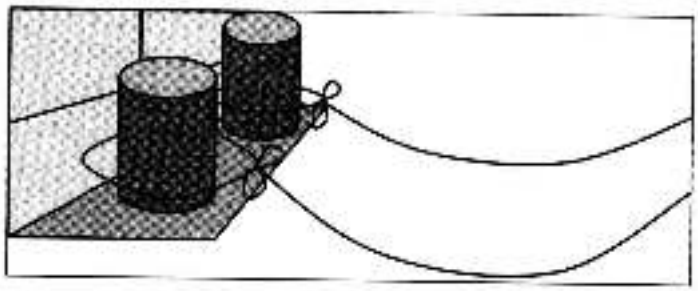


图1-4 使用啤酒罐进行通信

远程啤酒罐控制看似不错，但仍存在很大的缺陷。问题在于Alice只能在Bob的窗台上摆放有限个数的啤酒罐，迟早有一天，她会打翻所有的啤酒罐。即使Bob总是及时地扶正被打翻的啤酒罐，然而，一旦他去欧昆速假该怎么办呢？只要是指望由Bob来扶正啤酒罐，那么Alice早晚都会用完所有的啤酒罐。

Alice和Bob于是又尝试使用另外一种方法。他俩各自在对方很容易看到的地方竖起一个旗杆。如果Alice想让她的猫去院子里活动，则按以下步骤进行处理：

1. 升起她自己的旗子。
2. 若Bob的旗子是降下来的，则将她的猫放出去。
3. 当猫返回屋子时，将她自己的旗子降下。

Bob的操作则相对复杂一些。

1. 升起他自己的旗子。
2. 若Alice的旗子处于升起状态，则重复执行下列操作：
   a) 降下他自己的旗子。
   b) 等待直到Alice的旗子被降下。
   c) 重新升起他自己的旗子。
3. 只要Bob升起了自己的旗子并且发现Alice的旗子是降下来的，就可以将自己的小狗放出去。
4. 当小狗返回屋子里时，Bob则降下他自己的旗子。

下面进一步地研究这个用于解决Alice-Bob问题的协议。从直观上来看，该协议之所以能够正常地工作是由于下面的读写系列，如果Alice和Bob都

1. 升起自己的旗子，然后

7 2. 观察对方的旗子。

那么，至少有一个人会看到对方的旗子是升起的（显然，最后一个观察的人会看到对方的旗子是升起的），这样的话，Alice和Bob中的一个人不会让自己的宠物进入院子。然而，采用这种观察方法并不能证明在院子里决不会同时出现两只宠物，例如，当Bob正在观察Alice的旗子时，如果Alice让她的猫多次地进出院子，那么会出现什么样的情形呢？

为了说明两只宠物决不会同时出现在院子里，我们采用反证法来证明。假设存在一种方法能使得两只宠物同时出现在院子里。现在考虑Alice和Bob分别升起了自己的旗子，并准备让各自的宠物进入院子之前最后观察对方旗子这一时刻。当Alice观察对方的旗子时自己的旗子已经完全升起来了，而且她肯定没有看到Bob的旗子，否则她不会让自己的猫到院子里去。所以，在Alice开始观察的瞬间，Bob肯定还没有完全把旗子升起来。由此推出当Bob升起自己的旗子并最后观察Alice的旗子时，Alice必定已经查看过Bob的旗子，所以Bob肯定会看到Alice的旗子已被升起，于是他不会将自己的小狗放出去，这与假设矛盾。

这种反证的论证方法今后将会被经常使用，因此要仔细地推敲为什么上述断言成立。有一点需要注意，我们从未假定“升起自己的旗子”及“观察对方的旗子”这种行为是瞬间发生的，也没有假定这种动作会持续多久。我们只关心这些动作何时开始或者何时结束。

1.2.1 互斥特性

为了证明旗子协议能够正确地解决Alice-Bob问题，首先必须理解正确的解决方案应该满足什么样的特性，然后再证明旗子协议能保证这些特性成立。

前面已证明两个宠物不会同时待在院子里，这种特性称为互斥。

然而，互斥只是所需的特性之一。如前面所提到的，Alice和Bob都不允许自己的宠物进入院子这样的协议也满足互斥特性，但对于他们的宠物来说这种协议显然是不可接受的。下面分析另外一种重要特性。如果一个宠物想进入院子，则最终必会成功；如果两个宠物同时都想进入院子，则至少有一个能够成功。这种特性称为无死锁，它是解决Alice-Bob问题的基本特性。

可以断定上述Alice-Bob协议是无死锁的。假设两个宠物都想进入院子，于是Alice和Bob各自升起自己的旗子，Bob最终总会发现Alice的旗子是升起的，则会降下旗子暂缓自己的请求，从而让Alice的猫进入院子。

无饥饿（或无对锁）特性是一种必须关注的特性：如果一个宠物想进入院子，它最终能够成功吗？对此，Alice-Bob协议并不能完全保证。每当Alice和Bob发生冲突时，Bob都必须
8 屈从Alice而暂缓自己的请求，因此，有可能出现这样的情形：Alice的猫一次又一次地进入院子，而Bob的小狗则一直窝在房子里变得越来越烦躁。稍后，我们将会看到如何通过协议防止出现这种饥饿现象。

最后一种需要关注的特性就是冬号。假设Alice升起自己的旗子，随后突发阑尾炎，于是她（和她的猫）去了医院。手术成功之后，留院观察一周。虽然Bob为Alice的病情得以好转而松了一口气，但在Alice离开家里的这段日子里，Bob的小狗无法进入院子。问题的根源就在于协议中规定Bob必须等待Alice把旗子降下后才能把自己的小狗放出去。如果Alice被耽误了（即便理由是好的），Bob也会被延迟（没有什么理由）。

等待问题在系统中是非常重要的，通常情况下，希望在一段合理的时间内，Alice和Bob都能够及时地响应对方，但如果没有及时响应该怎么办？互斥的本质就是等待：无论一个互

序协议设计得多么巧妙，都无法避免等待。然而，大多数其他的协作问题无需等待就可以解决，有时甚至采用一些出乎预料的方法。

1.2.2 道德

上面分析了Bob—Alice协议的先缺点，现在回到计算机科学上来。

首先来看看隔着院子喊叫和打电话的方式为什么不行。通常，并发系统中存在两种类型的通信：

- 瞬时通信要求通信双方在同一时间都参与通信。喊叫、打手势或者通电话都是瞬时通信。
- 持续通信允许发送者和接收者在不同时间参与通信。写信、发邮件或在石块上留言都是持续通信。

互斥需要的是持续通信。隔着院子喊叫或打电话方式的问题就在于，此时无法确定是否让Bob将小狗放出去。如果Alice不能给予回复，Bob将永远不知道该怎么做。

啤酒罐—绳子协议似乎有些夸张，但它却完全符合并发系统中常用的一种通信协议：中断。现代操作系统中，一个线程要引起另一个线程注意的常用方法就是发送中断信号。更准确地说，线程A通过设置一个位向线程B发出一个中断信号，线程B周期地检查这个位。一旦B检测到该位被设置，则做出相应的响应。响应结束后，通常由B进行复位（A不能复位）。虽然中断不能解决互斥问题，但它仍是非常有用的。例如，Java中wait()调用和notifyAll()调用的本质就是中断。

从更积极的角度来看，这个故事揭示了这样一个事实：两个线程之间的互斥问题能够通过两个1比特变量来解决（尽管不完善），每个变量只能被一个线程写，而由另一个线程读。

1.3 生产者—消费者问题

互斥并不是唯一需要关注的问题。故事的后来，Alice和Bob相爱并且结了婚。最终，他们又离了婚。（他们在想些什么？）法官将宠物的监管权判给了Alice，而让Bob负责喂养它们。现在两只小宠物相处得十分融洽，但它们只和Alice亲近，每当看到Bob就会上前攻击他。于是，Alice和Bob需要设计另外一个协议，允许Bob在他和宠物不同时在院子里的情况下给它们喂食物。更进一步，该协议还要能保证不浪费双方的时间：在院子里没有食物的情况下，Alice不会将宠物放出去；而在食物未被吃完前，Bob也不愿再次进入院子放置食物。这种问题称为生产者—消费者问题。

有趣的是，解决互斥问题时被放弃的啤酒罐—绳子协议在解决生产者—消费者问题时却非常有用。Bob在Alice的窗台上竖直摆放了一个啤酒罐，并用绳子的一端将其绑住，而把绳子的另一端牵进自己的房间。接着，他把食物放到院子里，拉动绳子将啤酒罐打翻。此后，当Alice要把宠物放入院子时，她按照以下步骤进行：

1. 等待直到啤酒罐被打翻。
2. 将宠物放出去。
3. 当宠物返回屋子以后，检查它们是否吃完了院子里的食物，如果吃完了，则重新将翻倒的啤酒罐摆正。

Bob的步骤如下：

1. 等待直到发现啤酒罐被重新摆正。
2. 将食物放到院子里。

3. 竖起绳子，使啤酒罐再次翻倒。

啤酒罐的状态实质上反映了院子里的状态。如果啤酒罐是翻倒的，则意味着院子里有食物且宠物可以进去吃；如果啤酒罐是竖直放好的，则表示食物已经被吃完且Bob可以再放些食物。现在，考察下面三种特性：

- *互斥*：Bob和宠物决不会同时出现在院子里。
- *无饥饿*：如果Bob始终愿意放食物，而两只小宠物总是很饿，那么两只小宠物将能够永远不停地吃到食物。
- *生产者-消费者*：除非院子里有食物，否则宠物不会进入院子；除非院子里的食物已被吃完，否则Bob不会继续投放更多的食物。

10

此处的生产者-消费者协议以及上一节的互斥协议都能够保证Alice和Bob决不会同时出现在院子里。然而，不能使用这个生产者-消费者协议来保证Alice和Bob的互斥，理解这一点非常重要。互斥要求无死锁：任何一方就其自身而言，都能够无限次地进入院子，即使另一方不在场的情形下也是如此。与此相反，生产者-消费者协议的无饥饿特性却假设双方保持连续的协同操作。

下面对该协议进行分析：

- *互斥*：这里采用一种与前面的互斥证明稍有区别的证明方法：基于“状态机”而不是反证法来证明。我们将系统视为一个状态机，啤酒罐只有两个状态：竖直和翻倒，并且在这两个状态之间反复地转换。现在需要证明，因为自动机的初始状态满足互斥特性，并且从任意一个状态向另一个状态的转换也保持着互斥特性，所以该协议满足互斥特性。

  初始状态下啤酒罐只能是竖直或者翻倒的。现假设它为翻倒的，则只有宠物能够进入院子，故此时满足互斥特性。若Alice要摆正啤酒罐，宠物首先必须离开院子，因此，当啤酒罐被摆正的时候宠物不在院子里；又因为在啤酒罐再次被打翻之前宠物不会进入院子，所以自动机从翻倒状态转换为竖直状态时保持互斥特性。若要打翻啤酒罐，Bob必定已经离开了院子，并且在啤酒罐下次被摆正之前不会进入院子，由此从竖直状态转换为翻倒状态自动机也保持着互斥特性。此外，不再存在其他可能的转换过程了，至此断言成立。
- *无饥饿*：假设此断言不成立，那么必然存在以下情形：Alice的宠物由于没有食物而一直处于饥饿状态，Bob试图投放食物但不能获得成功。此时，啤酒罐不可能是竖直的，因为这时Bob想给宠物提供食物并打翻了啤酒罐。那么，啤酒罐必定是翻倒的。又因为宠物是饥饿的，Alice最终必会摆正啤酒罐，与前述矛盾。
- *生产者-消费者*：互斥特性意味着宠物和Bob不会同时在院子里出现。在Alice没有摆正啤酒罐之前Bob不会进入院子，而只有在院子里没有食物时Alice才会去摆正啤酒罐。同样，在Bob没有打翻啤酒罐之前宠物不会进入院子，而只有在Bob放置了食物以后他才会打翻啤酒罐。

与前面已经讲过的互斥协议一样，该协议存在着等待。若Bob在院子里放置食物后忘记了打翻啤酒罐并马上度假去了，那么此时院子里虽然有食物，但宠物却是饥饿的。

现在把注意力转回到计算机科学上来，几乎所有的并行分布式系统都会出现生产者-消费者问题。它是各个处理器向通信缓冲区中放置数据，由其他处理器读取或者通过互联网络或

11

共享总线进行传递的方式。

1.4 读者-写者问题

Bob和Alice都十分喜爱自己的宠物，于是决定彼此之间交流一些与宠物相关的信息。Bob在屋子前面竖起了一块公告牌。公告牌上可以粘贴一串很大的瓷片，每个瓷片上只能写下一个字母。空闲的时候，Bob采用一次贴上一块瓷片的方式，通过公告牌传递信息。Alice一有空就用望远镜看Bob在公告牌上留的信息，一次也只读一块瓷片上的内容。

这种方法听起来似乎是一种可行的方案，其实不然。我们来分析这样的场景：假设Bob传递了信息：

```
sell the cat
```

Alice通过望远镜看到了

```
sell the
```

恰恰此刻，Bob取下了所有的瓷片又全部写上新信息：

```
wash the dog
```

Alice接着继续扫描公告牌，最后看到信息

```
sell the dog
```

结局可想而知。

还有其他一些简单明了的方法可以解决读者-写者问题。

- Alice和Bob可以利用互斥协议来确保Alice只能读到完整的语句。然而，她可能漏掉某个语句。
- 他们可以使用啤酒罐-绳子协议，Bob生产语句而Alice消费语句。

如果问题这么容易解决，为什么还特意拿出来讲呢？互斥协议和生产者-消费者协议都要求等待：如果参与者一方由于某个不能预测的事情延误了，另一方也必然被延误。在多处理器共享存储器方式下，解决读者-写者问题的一种可行方法就是让每个线程能够获得多个存储单元的瞬间视图，也就是说无需等待就可以获得这样的视图，或者说当这些存储单元的内容正被读取时，无需防止其他的线程修改它们。这种方法在备份、调试以及其他一些场合是十分有用的，令人惊讶的是，的确存在着这种无等待的办法可以解决读者-写者问题。后面将会看到几个这样的例子。

12

1.5 并行的困境

现在来看看为什么多处理器编程富有趣味性。理想情形下，从单处理器升级到n路关联多处理器应该提高了n倍的计算能力，遗憾的是，实际中不可能做到这一点，其主要原因就在于现实世界中的大多数计算问题在不考虑处理器之间通信和协作开销的情况下无法有效地并行化。

假设有5个朋友要粉刷一套有5个房间的房屋。如果所有房间大小都一样，那么可以指定每人负责一间，只要这5个人以同样的速度来粉刷，就能获得相当于由一个人完成整个粉刷工作5倍的加速比。如果每个房间的大小不一，问题就复杂多了。例如，若有一个房间是其他房间的2倍，那么这5个人同时工作就不可能获得5倍的加速比，因为整个任务的完成时间取决于粉刷时间最长的那个房间。

这样的分析对并发计算非常重要，可以用Amdahl定律进行解释。该定律揭示了这样一个概念：完成复杂工作（不只是粉刷房子）可获得的加速比是有限的，受限于这个工作中必须被串行执行的部分。

工作的加速比S定义为由一个处理器来完成一项工作的时间（用串行时间来计算）与采用

n个处理器并发完成该工作的时间之比。Amdahl定律给出了用n个处理器协同完成一个应用时可获得的最大加速比S，其中p是该应用中可以并行执行的部分。为简单起见，假设由一个处理器完成整个应用需用1个单位的时间。使用n个处理器并发执行应用中的并行部分需用时p/n，应用中串行部分的执行时间为$1-p$。因此，并行化以后的计算时间为：

$$1 - p + p/n$$

按照Amdahl定律所给出的加速比定义，得到

$$S = 1 / (1 - p + p/n)$$

我们仍采用粉刷房屋的例子来解释Amdahl定律的含义。假定每个小房间是1个单位，唯一的大房间是2个单位。每个房间指定一个人（处理器）粉刷意味着6个单位中的5个可以并行粉刷，即$p=5/6$，$1-p=1/6$。由Amdahl定律可得加速比为

13

$$S = 1 / (1 - p + p/n) = 1 / (1/6 + 1/6) = 3$$

由此可知，若有一间房间的大小是其他房间的2倍，那么5个人同时粉刷房屋的加速比仅仅是一个人粉刷的3倍。

如果由10个人分别粉刷10间房间，其中一间房间是其他房间的2倍，情况将变得更糟。下面是所求出的加速比：

$$S = 1 / (1/11 + 1/11) = 5.5$$

可以看出，仅仅微小的失衡就使10个人粉刷房屋只能获得5倍多的加速比，几乎是预期值的一半。

因此，可以使用类似于前面素数打印中所采用的方案：一旦某个人完成了自己的工作，他就马上帮助其他人完成剩余的工作。然而，这种共享式粉刷所存在的问题就在于粉刷者之间需要协调合作，那么是否可以设法避免这种协调问题呢？

下面分析Amdahl定律所揭示的多处理器利用率问题。有些计算问题是可以“高度并行”的，那些易于划分为多个可并发执行部分的计算。这种问题有时会在科学计算或图像处理中出现，但在系统中却很少出现。通常情况下，对于一个给定的问题以及一台具有10个处理器的机器，由Amdahl定律可知，即使其中的90%可以并行，而仅有10%需要串行，最终也只能获得5倍的加速比，而不是10倍。也就是说，串行执行的10%使得机器的利用率降低了一半。由此可见，应该尽量使那10%的部分达到最大程度的并行。当然，这一点实现起来非常困难，其难点就在于新增的并行部分中往往涉及通信和协作。本书的重点就是：理解和掌握对程序代码中那些需要同步和协作的部分进行高效编程的技术和工具。这部分代码的改进将对系统性能产生很大的影响。

我们回顾图1-2所示的素数打印问题，分析其中三行主要代码：

```
i = counter.getAndIncrement(); // take next untaken number
if (isPrime(i))
  print(i);
```

如果线程将这三行代码作为一个原子单位来执行，也就是将它们放入一个互斥域内，那么问题的解决非常简单。然而，现在只让getAndIncrement()调用为原子的。这样的实现符合Amdahl定律：应最小化串行代码的粒度，即只采用互斥方式执行getAndIncrement()。此外，由于围绕着该可共享的互斥计数器的通信和协作本质上也会影响整个程序的性能，因此

14 互斥机制的高效率实现也是很重要的。

1.6 并行程序设计

对于大多数急欲被并行化的应用，很容易就可以发现其中的许多部分能够并行执行，其原因在于这些部分之间不需要任何通信和协作。在写这本书的时候，还没有专门讲述如何辨别应用中可并行部分的指导大全。这种辨别技术要用到设计人员实际积累的关于并行划分的知识，但幸运的是，大多数情况下应用的可并行部分是显而易见的。然而，本书阐述另外一个更为本质的问题，这就是如何处理余下的那部分不可并行的程序。前述已知，由于程序需要存取共享数据以及在处理器之间进行通信和协作，所以剩下的部分很难被并行化。

本书的目的就是向读者揭示现代协作范例和并发数据结构中所隐藏的核心思想。从基本的原理到最优的实用技术，统一、全面地介绍了高效多处理器编程的关键要素。

多处理器编程面临着许多挑战，大到智能化问题小到微妙的工程技巧。我们采用逐步求精的方法来解决这些问题，先从理想化的数学模型开始，逐层细化到考虑工程设计原则的实际模型。

例如，对于最早提出的互斥问题（一个古老但很重要的问题），我们先从数学的抽象角度出发，在一个理想化的系统结构中研究各种算法的可计算性及正确性。虽然在现代的系统结构中这些算法并不实用，但它们却是一些经典的算法。况且，对这些经典算法的推理分析过程进行深入的研究，是学习如何分析设计更加复杂的实用算法的必经之路。尤其是要学会如何来分析和处理类似于饥饿和死锁这种微妙的活性问题。

一旦大体上掌握了这些算法的一般分析方法，就可以转向更加实际的情形，我们针对不同的多处理器系统结构，设计开发了一系列不同的算法和数据结构，其目的在于对比分析那种算法的效率更高以及发生这种情形的原因。

1.7 本章注释

大多数有关Alice和Bob的故事都来自于Leslie Lamport于1984年在ACM分布式计算原理会议上的特邀演讲[92]。读者-写者问题则是在过去20多年的许多文章中都已讨论过的经典同步问题。Amdahl定律归功于Gene Amdahl，他是并行处理领域中的先驱人物[9]。 15

1.8 习题

习题1. 哲学家就餐问题是由并发处理的先驱E. W. Dijkstra所提出的，主要用于阐述死锁和无饥饿概念。假设五个哲学家一生只在思考和就餐。他们围坐在一个大圆桌旁，桌上有一大盘米饭。然而只有五根可用的筷子，如图1-5所示。所有的哲学家都在思考。若某个哲学家饿了，则拿起自己身边的两根筷子。如果他能够拿到这两根筷子，则可以就餐，当这个哲学家吃完后，又放下筷子继续思考。

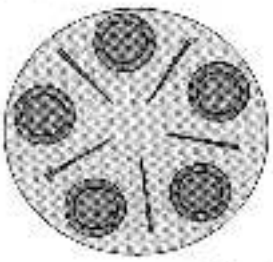


图1-5 Dijkstra提出的餐桌布局

1. 试编写模仿哲学家就餐行为的程序，其中每个哲学家为一个线程而筷子则是共享对象。注意，必须防止出现两个哲学家同时使用同一根筷子的情形。
2. 修改所编写的程序，不允许出现死锁情形。也就是说，保证不会出现这样的情形：每个哲学家都已拥有一根筷子，并且正在等待获得另一个人手中的筷子。
3. 修改所编写的程序使得不会出现饥饿现象。
4. 编写能够保证任意n个哲学家无饥饿就餐的程序。

习题2. 下面各种方法满足安全性还是活性？指出所关心的“坏事”和“好事”。

1. 按到达的顺序为顾客人服务。

16

2. 升上去的东西都必须降下来。
3. 如果有两个或多个进程正在等待进入自己的临界区，则至少有一个会成功。
4. 如果发生一个中断，则在一秒内要输出一条消息。
5. 如果发生一个中断，则要输出一条消息。
6. 生活费决不下降。
7. 有两件事是肯定的：死亡和税。
8. 你总是能够告诉一个哈佛人。

习题3. 在生产者-消费者问题中，假设Bob能够看见Alice窗台上的啤酒罐是竖直还是颠倒的。请基于啤酒罐-绳子协议来设计一种生产者-消费者协议，使得即使Bob无法看见Alice窗台上啤酒罐的状态，也能够正常工作（即实际中断位是如何工作的）。

习题4. 假设你是最近被捕的P个囚犯之一。监狱长是个疯狂的计算机科学家，他给出了以下告示：

你们今天可以在一起商定一个策略，但是从今天之后，你们将会被隔开关在不同的房间，相互间无法再进行交流。

我们已建造了一对"开关房间"，里面有一个灯开关，这个开关只能为开或关，且没有和任何东西相连。

我将不时地从你们中间随机选择一位到"开关房间"里来。这名囚犯可以拨动开关（从开到关，或相反），也可保持开关的状态不变。其他人这时都不能进入房间。

每一名囚犯都将任意多次地进入开关房间。更确切地说，对于任意的N，你们中的每个人最终都至少进入这个房间N次。

任何时刻，任意一名囚犯都可以宣布："我们所有的人都已经至少到过开关房间一次了。"如果该断言是对的，我将释放你们。如果错了，我就把你们全都送去喂鳄鱼。谨慎抉择吧！

- 在开关的初始状态为关的情况下，设计一个可以成功取胜的策略。
- 在不知道开关初始状态的情况下，设计一个可以成功取胜的策略。

提示：所有的囚犯不必做相同的动作。

习题5. 上题中的监狱长又有了另外一个想法。他命令囚犯们站成一排，每个人都戴一顶红色或蓝色的帽子。没有人知道自己所戴帽子的颜色，也不知道他后面所有人帽子的颜色，但能看见前面所有人帽子的颜色。监狱长从队伍的后面开始询问每个囚犯，让他们猜测自己帽子的颜色。囚犯们只能回答"红色"或"蓝色"。如果他答错了，就会被送去喂鳄鱼。如果他答对了，则会被释放，每个囚犯都能听到后面所有人的回答，但不知道答案是否正确。

17

囚犯们在站队之前可以商讨一个策略（监狱长是听着的）。一旦站好队之后，每个人除了能回答"红色"或"蓝色"以外，再也不能以其他任何方式进行交流。

设计一个能够保证P个囚犯中至少有$P-1$个会被释放的策略。

习题6. 使用Amdahl定律解决下面问题：

- 假定在一个程序中包含有一个无法并行化的方法M，其执行时间为该程序总运行时间的40%。若在一台n处理器的多核机器上运行此程序，总加速比的上限应为多少？
- 假定方法M占整个程序执行时间的30%。要使程序的总运行时间比原来提高2倍，M的加速比应为多少？
- 假定方法M的速度可以提高3倍。要使程序的总加速比为原来的2倍，那么在程序的总运行时间中，M应占多大的比例？

习题7. 在两个处理器上运行时，程序可获得的加速比为S_2。使用Amdahl定律推导出在n个处理器上运行时程序的加速比S_n，要求用n和S_2来表示。

习题8. 现有一台每秒可执行5亿万条指令的单处理器机器和一台有10个处理器的多处理器机器，其中每个处理器每秒可执行1亿万条指令。针对一个特定的应用，使用Amdahl定律来解释应该购买哪台机器。

18

原　理

第2章　互斥

第3章　并发对象

第4章　共享存储器基础

第5章　同步原子操作的相对能力

第6章　一致性的通用性

第2章

The Art of Multiprocessor Programming (Revised First Edition)

互　斥

互斥是多处理器程序设计中最常见的一种协作方式，本章涵盖了基于读/写共享存储器模型的各种经典互斥算法。虽然这些算法并不实用，但它们全面地概括了同步领域中各种算法设计及正确性证明问题，因此值得深入研究。此外，本章还介绍了不可能性的证明方法。通过这种证明，指出了用于读/写共享存储器模型中的各种互斥算法所存在的不足之处，为后面提出更加实用的互斥算法奠定了基础。本章是书中少数几个包含有算法证明的章节之一。为轻松起见，读者可以跳过这些证明部分，然而，理解这些证明的推导过程是非常有益的，因为这种推导方法可以用于后面实际算法的推理分析中。

2.1　时间

分析并发计算的实质就是分析时间。有时希望事件同时发生，有时希望事件在不同时间发生。需要对各种复杂情形进行分析，包括多个时间片应该怎样交叉重叠，或者相互之间不允许重叠。为此，需要一种简单而无二义性的语言来论述事件及其时延。由于日常英语的不确定性及二义性，因此我们引入一些简单的符号和词汇来描述并发线程中与时间相关的行为特征。

1689年，牛顿（Isaac Newton）曾说过："真正的，绝对的，数学意义上的时间，就其自身及其自身的自然属性而言，总是稳定平滑地流动着，而与外界任何事物无关。"除了这种晦涩难懂的定义方式外，我们完全赞同牛顿关于时间的看法。所有线程共享一个共同的时间（不必是同一个公共时钟）。一个线程是一个状态机，其状态的转换称为事件。

19
2
21

事件是瞬时的：它们在单个瞬间发生。为了便于讨论，我们认为事件绝不是同时的：不同的事件在不同的时刻发生。（实际应用中，如果不能确定在时间上非常接近的事件到底谁先谁后，那么以任意次序都行。）线程A产生一个事件序列a_0，a_1，…。当于线程中往往包含有循环，因此一条程序语句可以产生多个事件。用a^j表示事件a_j的第j次发生。如果事件a在事件b之前发生，则称a先于b，记作$a \rightarrow b$。事件集上的先于关系"$\rightarrow$"是个序的。

令a_0和a_1表示事件，且$a_0 \rightarrow a_1$。$interval(a_0, a_1)$表示a_0和a_1之间的间隔。如果$a_1 \rightarrow b_0$（也就是说，I_A的结束事件先于I_B的开始事件），则间隔I_A= $interval\ (a_0, a_1)$先于间隔I_B= $interval\ (b_0, b_1)$，记作$I_A \rightarrow I_B$。更简单地说，"$\rightarrow$"关系是间隔集合上的偏序关系。多个不存在"$\rightarrow$"关系的间隔称为并发的。类似于事件中的记法，用I_A^j表示间隔I_A的第j次执行。

2.2　临界区

前面讨论了Counter类的实现（图2-1）。我们已知这种实现在单线程系统中能够正常工作，而在多线程系统中则有可能出错。出现这种现象的原因就在于两个线程都在"start of danger zone"（危险区起点）那一行读了value域的值，随后又都在"end of danger zone"（危险区终点）那一行修改了value域的值。

```
1  class Counter {
2    private int value;
3    public Counter(int c) {        // constructor
4      value = c;
5    }
6    // increment and return prior value
7    public int getAndIncrement() {
8      int temp = value;            // start of danger zone
9      value = temp + 1;            // end of danger zone
10     return temp;
11   }
12 }
```

图2-1 Counter类

为了避免出现上述情形，我们将这两行代码置入临界区内：某个时刻仅能被一个线程执行的代码段。称这样的特性为*互斥*。实现互斥的标准方法就是采用一个具有如图2-2所述接口的Lock对象。

```
1  public interface Lock {
2    public void lock();    // before entering critical section
3    public void unlock();  // before leaving critical section
4  }
```

图2-2 Lock对象的接口

```
1  public class Counter {
2    private long value;
3    private Lock lock;              // to protect critical section
4
5    public long getAndIncrement() {
6      lock.lock();                  // enter critical section
7      try {
8        long temp = value;          // in critical section
9        value = temp + 1;           // in critical section
10       return temp;
11     } finally {
12       lock.unlock();              // leave critical section
13     }
14   }
15 }
```

图2-3 使用Lock对象的Counter类

为简短起见，一个线程若执行了lock()方法调用，则称该线程获得一个锁（或称上锁），若执行了unlock()方法调用，则称该线程释放这个锁（或称开锁）。图2-3说明了在共享计数器的实现中，如何通过使用Lock锁来保证对象的互斥特性。线程必须按照指定的方式调用lock()和unlock()。如果一个线程满足下列条件，则称它是良构的：

1. 一个临界区只和唯一的Lock对象相关联；
2. 线程准备进入临界区时调用该对象的lock()方法；
3. 当线程离开临界区时调用unlock()方法。

编程提示2.2.1 在Java中，应该采用下述结构化方式调用这些方法。

```
mutex.lock();
try {
  ...          //body
  } finally {
    mutex.unlock();
  }
```

22-23 这种用法能够确保在进入try程序块以前获得锁，在离开程序块时释放锁，即使在程序块中的某些语句抛出异常时也是如此。

下面我们形式化地描述一个好的锁算法应该满足哪些特性。令CS_A^j是A第j次执行临界区的时间段，为简单起见，假设线程可以无限次地获得锁或者释放锁，而在它们上锁/开锁的期间，允许其他的事件发生。

互斥 不同线程的临界区之间没有重叠。对于线程A、B以及整数j、k，或者$CS_A^j \to CS_B^k$或者$CS_B^j \to CS_A^k$。

无死锁 如果一个线程正在尝试获得一个锁，那么总会成功地获得这个锁。若线程A调用lock()但无法获得锁，则一定存在其他的线程正在无穷次地执行临界区。

无饥饿 每一个试图获得锁的线程最终都能成功。每一个lock()调用最终都将返回。这种特性有时称为无对锁特性。

注意无饥饿意味着无死锁。

显然互斥是基本的特性。没有这种特性，将无法保证计算结果的正确性。按照第1章的术语，互斥是一种安全特性。无死锁是重要的特性，它表明系统决不会"冻结"，个别线程可能会永久地停滞等待（称为饥饿），而总有一些线程能够继续执行。无死锁可以看作是一种活性特性（见第1章）。值得注意的是，即使一个程序中所使用的每个锁都满足无死锁特性，该程序也可能死锁。例如，线程A和线程B共享锁ℓ_0和ℓ_1。A首先获得ℓ_0而B获得了ℓ_1。然后，A试图获取ℓ_1而B试图获取ℓ_0。此时由于两个线程都需要对方释放锁，从而陷入等待发生死锁。

无饥饿特性无疑是最令人满意的一个特性，但却是三个特性中最不需要保持的。稍后，将会看到一些不满足无饥饿特性但却很实用的互斥算法。这些算法被广泛地使用在一些理论上有可能发生饥饿而实际上却不会出现的应用场景中。尽管如此，学会分析饥饿现象对理解是否存在实际的威胁依然是很重要的。

无饥饿特性不保证线程在进入临界区以前需要等待多长时间，就这一点来说，它也是比较弱的。下面将会讨论为线程设置等待时间边界的算法。

2.3 双线程解决方案

24 我们先从两个虽然存在不足但却十分有趣的锁算法讲起。

2.3.1 LockOne类

图2-4描述了LockOne算法。双线程的锁算法遵循以下两点约定：线程的标识为0或1，若当前调用者的标识为i，则另一方为$j=1-i$；每个线程通过调用ThreadID.get()获取自己的标识。

编程提示2.3.1 实际编程中，为了保证正常地工作，例2-4中的布尔型变量flag以及后面算法中的victim和label变量都必须被声明为volatile类型，我们在第1章和附录B中简述其原因，我们将在第7章将相应的变量声明为volatile类型。

用$write_A(x = v)$表示A将值v赋于域x，用$read_A(v == x)$表示A从域x中读取值v。在值不重要的情形下，可以省略v。图2-4中的$write_A(flag[i] = true)$事件是由$lock()$方法中第7行代码的执行所引起的。

```
1  class LockOne implements Lock {
2    private boolean[] flag = new boolean[2];
3    // thread-local index, 0 or 1
4    public void lock() {
5      int i = ThreadID.get();
6      int j = 1 - i;
7      flag[i] = true;
8      while (flag[j]) {}          // wait
9    }
10   public void unlock() {
11     int i = ThreadID.get();
12     flag[i] = false;
13   }
14 }
```

图2-4 LockOne算法

引理2.3.1 LockOne算法满足互斥特性。

证明 假设不成立，则存在整数j和k使得$CS_A^j \to CS_B^k$并且$CS_B^k \to CS_A^j$。考虑每个线程在第k次（第j次）进入临界区前最后一次调用lock()方法的执行情形。

通过观察代码可以看出

$$write_A(flag[A] = true) \to read_A(flag[B] == false) \to CS_A^j \tag{2.3.1}$$

$$write_B(flag[B] = true) \to read_B(flag[A] == false) \to CS_B^k \tag{2.3.2}$$

$$read_A(flag[B] == false) \to write_A(flag[B] = true) \tag{2.3.3}$$

注意一旦flag[B]被设置为$true$，则将保持不变。由此得知公式（2.3.3）成立，否则线程A就不可能读到flag[B]的值为$false$。由公式（2.3.1）～（2.3.3）和先于关系的传递性可导出公式（2.3.4）。

$$write_A(flag[A] = true) \to read_A(flag[B] == false) \to write_B(flag[B] = true) \to read_B(flag[A] == false) \tag{2.3.4}$$

由此可知，$write_A(flag[A] = true) \to read_B(flag[A] == false)$过程中没有对数组flag[]进行写操作，得出矛盾。□

LockOne算法存在缺陷，其原因在于线程交叉执行时会出现死锁。若事件$write_A(flag[A] = true)$及$write_B(flag[B] = true)$在事件$read_A(flag[B])$和$read_B(flag[A])$之前发生，那么两个线程都将陷入无穷等待。然而，LockOne算法有一个有趣的特点：如果一个线程在另一个线程之前运行，则不会发生死锁，一切都工作得很好。

2.3.2 LockTwo类

图2-5给出了另一种锁算法LockTwo类。

```
1  class LockTwo implements Lock {
2    private int victim;
3    public void lock() {
4      int i = ThreadID.get();
5      victim = i;              // let the other go first
6      while (victim == i) {};  // wait
7    }
8    public void unlock() {}
9  }
```

图2.5 LockTwo算法

引理2.3.2 LockTwo算法满足互斥特性。

证明 假设不成立，那么存在整数k和k'使得$CS_A^k \rightarrow CS_B^{k'}$且$CS_B^{k'} \rightarrow CS_A^k$。考虑每个线程在第$k$次（第$k'$次）进入临界区前最后一次调用lock()方法的执行情形。

通过观察代码可以看出

$$\text{write}_A(\text{victim} = A) \rightarrow \text{read}_A(\text{victim} == B) \rightarrow CS_A^k \tag{2.3.5}$$

$$\text{write}_B(\text{victim} = B) \rightarrow \text{read}_B(\text{victim} == A) \rightarrow CS_B^{k'} \tag{2.3.6}$$

线程B必须在事件$\text{write}_A(\text{victim} = A)$和事件$\text{read}_A(\text{victim} = B)$之间将$B$赋给victim域（见公式（2.3.5））。由于这是最后一次赋值，所以有

$$\text{write}_A(\text{victim} = A) \rightarrow \text{write}_B(\text{victim} = B) \rightarrow \text{read}_A(\text{victim} == B) \tag{2.3.7}$$

一旦victim域被设置为B，则将保持不变。所以，随后的读操作都将返回B，与公式（2.3.6）矛盾。

26

LockTwo类也存在缺陷，当一个线程完全先于另一个线程就会出现死锁。尽管如此，LockTwo也有一个有趣的特点：如果线程并发地执行，lock()方法则是成功的。LockOne类和LockTwo类彼此互补：能够保证一种解法正常工作的条件将会使另一种解法发生死锁。

2.3.3 Peterson锁

在图2-6中，我们将LockOne和LockTwo结合起来，构造出一种无饥饿的锁算法。该算法无疑是最简洁、最完美的双线程互斥算法，按照其发明者的名字被命名为“Peterson算法”。

```
1  class Peterson implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    private int victim;
5    public void lock() {
6      int i = ThreadID.get();
7      int j = 1 - i;
8      flag[i] = true;              // I'm interested
9      victim = i;                  // you go first
10     while (flag[j] && victim == i) {}; // wait
11   }
12   public void unlock() {
13     int i = ThreadID.get();
14     flag[i] = false;             // I'm not interested
15   }
16 }
```

图2-6 Peterson锁算法

引理2.3.3 Peterson锁算法满足互斥特性。

证明 假设不成立。像前面一样，考虑线程A和线程B最后一次执行lock()方法的情形。通过观察代码可以看出

$$\text{write}_A(\texttt{flag}[A] = true) \rightarrow \text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B]) \rightarrow \text{read}_A(\texttt{victim}) \rightarrow CS_A \tag{2.3.8}$$

$$\text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{write}_B(\texttt{victim} = B) \rightarrow \text{read}_B(\texttt{flag}[A]) \rightarrow \text{read}_B(\texttt{victim}) \rightarrow CS_B \tag{2.3.9}$$

不失一般性，假定A是最后一个对victim域进行写操作的线程。

$$\text{write}_B(\texttt{victim} = B) \rightarrow \text{write}_A(\texttt{victim} = A) \tag{2.3.10}$$

27

公式（2.3.10）隐含着线程A在公式（2.3.8）中读到的victim值为A。然而由于A已进入了自己的临界区，所以它读到的flag[B]肯定为*false*，因此有

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == false) \tag{2.3.11}$$

由公式（2.3.9）～（2.3.11）以及“→”关系的传递性，可得公式（2.3.12）。

$$\text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{write}_B(\texttt{victim} = B) \rightarrow \text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == false) \tag{2.3.12}$$

由此推出$\text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{read}_A(\texttt{flag}[B] == false)$。因为在进入临界区之前，没有对flag[B]执行过任何其他的写操作，于是产生矛盾。 □

引理2.3.4 Peterson锁算法是无饥饿的。

证明 假设不成立。假定（不失一般性）线程A一直在执行lock()方法，那么它必定在执行while语句，等待flag[B]被设置为*false*或者victim被赋值为B。

当A不能继续前进时，线程B在做什么呢？一种可能的情况是，B在反复地进入临界区又离开临界区。若是这样，线程B一旦重新进入临界区便会将victim设为B，一旦victim被设为B，就不再改变了，那么A最终肯定会从lock()方法返回，矛盾。

因此只可能是另一种情况，线程B也陷入lock()方法同用，等待flag[A]被设置为*false*或者victim被赋值为A。但是victim不可能同时被赋值为A和B，再次出现矛盾。 □

推论2.3.1 Peterson锁算法是无死锁的。

2.4 过滤锁

下面分析两种支持n（$n>2$）线程的互斥协议。第一种协议称为过滤锁，它是Peterson锁算法在多线程上的直接一般化。第二种协议称为Bakery锁，是一种最简单也最为人们所熟知的n线程锁算法。

图2-7所示为过滤锁，它建立了$n-1$个称为层的“等候室”，每个线程在获得锁之前必须穿过所有的层。图2-8描绘了这种层次结构。所有的层必须满足两个重要特性：

- 至少有一个正在尝试进入层ℓ的线程会成功。
- 如果有一个以上的线程要进入层ℓ，则至少有一个线程会被阻塞（即继续在那个层等待）。

28

Peterson锁用一个2元布尔数组flag来表示某个线程是否正在尝试进入临界区。过滤锁将此概念一般化，使用一个n元整型数组level[]，其中level[A]的值表示线程A正在尝试进入的最高层次。每个线程都必须通过$n-1$层的“排除”才能进入自己的临界区。每个层ℓ都有一个

victim[q]域，用来“过滤出”一个线程，使其不能进入下一层。

```
1  class Filter implements Lock {
2    int[] level;
3    int[] victim;
4    public Filter(int n) {
5      level = new int[n];
6      victim = new int[n]; // use 1..n-1
7      for (int i = 0; i < n; i++) {
8        level[i] = 0;
9      }
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     for (int i = 1; i < n; i++) { // attempt level i
14       level[me] = i;
15       victim[i] = me;
16       // spin while conflicts exist
17       while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18     }
19   }
20   public void unlock() {
21     int me = ThreadID.get();
22     level[me] = 0;
23   }
24 }
```

图2-7 过滤锁算法

初始时线程A在层0中。当它最后一次完成第17行的等待循环时，level[A]=j，称此时线程A在层j（j>0）中。（因此一个在层j中的线程也在层$j-1$中，以此类推。）

引理2.4.1 对于0到$n-1$中的整数j，层j上最多有$n-j$个线程。

证明 对j使用归纳法，当$j=0$时，显然成立。假设层$j-1$中最多有$n-j+1$个线程。现要证明至少有一个线程不能进入层j，为此，我们采用反证法：假设层j中有$n-j+1$个线程。

令A是层j中最后一个对victim[j]执行写操作的线程。由于A是最后一个线程，那么对层j中的任何其他线程B都有：

$$\text{write}_B(\text{victim}[j]) \rightarrow \text{write}_A(\text{victim}[j])$$

观察代码，可以看出线程B对level[B]的写是在它对victim[j]赋值之前完成的，所以

$$\text{write}_B(\text{level}[B]=j) \rightarrow \text{write}_B(\text{victim}[j]) \rightarrow \text{write}_A(\text{victim}[j])$$

另外还可以看出线程A对level[B]的读是在它对victim[j]写之后才进行的，所以

$$\text{write}_B(\text{level}[B]=j) \rightarrow \text{write}_B(\text{victim}[j]) \rightarrow \text{write}_A(\text{victim}[j]) \rightarrow \text{read}_A(\text{level}[B])$$

又因为B在层j中，所以A每次读level[B]时，必然读到一个大于或等于j的值，也就是说A

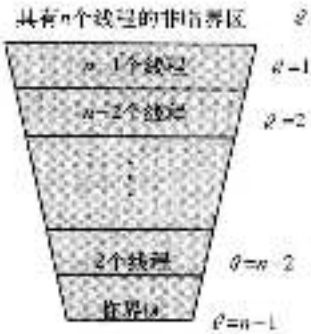


图2-8 线程需要通过$n-1$个层，最后一层是临界区。最多有n个线程可以同时进入层0，最多有$n-1$个线程可以同时进入层1（层1中的线程已在层0中），最多有$n-2$个线程可以同时进入层2，以此类推，最后只有一个线程能够进入$n-1$层中的临界区

不能完成第17行的等待循环，矛盾。

进入临界区等价于进入层$n-1$。

推论2.4.1 过滤锁算法满足互斥特性。

引理2.4.2 过滤锁算法是无饥饿的。

证明 对层数进行反向归纳。对层$n-1$，引理显然成立，因为在层$n-1$中最多只有一个线程，根据归纳假设，现假定每个到达层$j+1$或更高层的线程最终都能进入（并且离开）自己的临界区。

假设线程A被阻塞在层j中，由归纳假设可知，在比j高的层次中最终将没有线程存在。一旦A将level[A]设置为j，那么所有在层$j-1$中读level[A]的线程将都不能进入层j。于是，层$j-1$中的所有线程由于其随后对level[A]的读而都不能进入层j。最终，没有线程可以从小于j的层进到层j。于是，所有在层j中被阻塞的线程都在执行第17行的等待循环，并且victim域和level域的值不再改变。

现在对阻塞在层j中的线程个数进行归纳。若A是层j或更高层中唯一的线程，那么它无疑将会进入层$j+1$。现假设有小于k个线程不会被阻塞在层j中，如果线程A和线程B被阻塞在层j中，那么A只有读到victim[j] = A时才会阻塞，同样B只有读到victim[j] = B时才会阻塞。因为victim域是不变的，所以此时它不可能同时等于A和B，因此这两个线程中必有一个会进入层$j+1$，从而将阻塞的线程个数减少为$k-1$，与归纳假设矛盾。□

推论2.4.2 过滤锁算法是无死锁的。

2.5 公平性

无饥饿特性能够保证每一个调用lock()的线程最终都将进入临界区，但并不保证进入临界区需要多长时间，理想情况下（非形式化的），如果A在B之前调用lock()方法，那么A也应该先于B进入临界区。然而，运用现有的工具无法确定哪个线程首先调用lock()方法。取而代之的做法是，将lock()方法的代码划分为两个部分（根据相应的执行区间）：

1. 门廊区，其执行区间D_A由有限个操作步组成。
2. 等待区，其执行区间W_A可能包括无穷个操作步。

门廊区应该在有限步内完成是一种强约束条件。称这种约束为有界无等待演进特性。稍后的章节中将会讨论保障这一特性的系统实现方法。

下面是公平性定义。

定义2.5.1 满足下面条件的锁称为先来先服务的：如果线程A门廊区的结束在线程B门廊区的开始之前完成，那么线程A必定不会被线程B赶超。也就是说，对于线程A、B及常数j、k：

$$\text{若 } D_A^j \rightarrow D_B^k, \text{ 则 } CS_A \rightarrow CS_B^k。$$

2.6 Bakery算法

图2-9描述了Bakery锁算法。该算法采用面包店里发号机的一种分布式版本来保证先来先服务特性：每个线程在门廊区得到一个序号，然后一直等待，直到再没有序号比自己更早的线程尝试进入临界区为止。

在Bakery锁算法中，flag[A]是一个布尔型标志，表示线程A是否想要进入临界区；label[A]是一个整型数，说明线程进入面包店的相对次序。

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5      flag = new boolean[n];
6      label = new Label[n];
7      for (int i = 0; i < n; i++) {
8        flag[i] = false; label[i] = 0;
9      }
10   }
11   public void lock() {
12     int i = ThreadID.get();
13     flag[i] = true;
14     label[i] = max(label[0], ...,label[n-1]) + 1;
15     while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }
17   public void unlock() {
18     flag[ThreadID.get()] = false;
19   }
20 }
```

图2-9 Bakery锁算法

每当线程想获得一个锁时，它按照下面两个步骤产生新的label[]。第一步，它以任意的次序读取所有其他线程的label值。第二步，相继地读取其他线程的label值（可以按照某种次序），生成一个比它所读到的最大值大1的label值。我们把升起flag（第13行）到写新的label[]（第14行）这一段代码称为门廊。它表明线程的序号与正在试图获得锁的其他线程相关。如果有两个线程同时在门廊中，它们有可能读到相同的最大label值，从而产生同样的新label值。为了打破这种对称性，算法中采用了字典顺序"<<"来比较（label[], id）对：

$$(label[i], i) << (label[j], j) \quad \text{当且仅当}$$
$$label[i] < label[j] \text{或} label[i] = label[j] \text{且} i < j \tag{2.6.13}$$

在Bakery算法的等待部分（第15行）中，每个线程以某种任意的顺序交替地反复读取label，直到在所有已升起flag的线程中，该线程的（label[], id）变为最小为止。

由于释放锁时并不重设label[]，所以每个线程的label值是严格递增的。有趣的是，在门廊区和等待区，线程都是异步地读取label值，其次序是随机的，所以产生新label集之前的那个label集可能绝不会在同一个时刻存在于内存中。尽管如此，Bakery算法还是可以工作的。

32

引理2.6.1 Bakery锁算法是无死锁的。

证明 正在等待的线程中，必定存在某个线程A具有唯一的最小（label[A], A），那么这个线程决不会等待其他的线程。 □

引理2.6.2 Bakery锁算法是先来先服务的。

证明 如果A的门廊区先于B的门廊区，$D_A \rightarrow D_B$，那么A的label必小于B的label，因为

$$write_A(label[A]) \rightarrow read_B(label[A]) \rightarrow write_B(label[B]) \rightarrow read_B(flag[A])$$

所以，当flag[A]为*true*时B被封锁在外无法进入。 □

注意，既满足无死锁又满足先来先服务特性的算法必是无饥饿的。

引理2.6.3 Bakery算法满足互斥特性。

证明 假设不成立。令A和B是两个同时在临界区内的线程。$labeling_A$和$labeling_B$为它们各

自进入临界区前最后获得新label的事件。假设(label[A], A) << (label[B], B)。当B成功地完成在它的等待区内的检测时，它必定已读到flag[A]的值为*false*或(label[B], B) << (label[A], A)。然而，对一个给定的线程来说，其id是固定的且它的label[]值是严格递增的，所以B只能读到flag[A]的值为*false*。由此推出

$$\text{labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{labeling}_A$$

与假设(label[A], A) << (label[B], B)矛盾。 □

2.7 有界时间戳

在Bakery锁中，label值是无限增长的，因此在生命期很长的系统中不得不考虑溢出问题。如果某个线程的label域在其他线程都不知情的情况下从一个很大的值返回到零，那么先来先服务特性将被破坏。

下面将会看到一种利用计数器给线程排序，甚至可为每个线程产生一个唯一标识符的构造。溢出问题在现实中到底有多重要呢？这很难概括。有的时候它的问题很大。在20世纪最后的几年中媒体曾无数次报道著名的“Y2K”程序缺陷，虽然其引发的后果并没有想象中那么可怕，但它却是溢出问题的一个典型实例。到2038年1月18日，Unix的time_t数据结构将会溢出，因为其秒的数值是从1970年1月开始计算的，而在那一刻将会超过2^{31}。没有人知道这到底会引发什么。当然，有的时候计数器的溢出并不会产生什么大问题。大多数采用64位计数器的应用程序在其生存周期内是不可能发生这种“回零”问题的。（让我们的孙子辈来忧心吧！）

33

在Bakery锁中，label扮演着时间戳的角色：它们为所有争用的线程安排了一种次序。非形式化地来说，我们要确保若某个线程在另一个线程之后得到一个label，那么后者的label值一定要比前者的大。仔细观察Bakery锁算法的代码，可以看出一个线程需要具备两种能力：

- 读取其他线程的时间戳（扫描）。
- 为自己指定一个更晚的时间戳（标记）。

图2-10描述了一个针对这种时间戳系统的Java接口。由于有界时间戳系统主要用于实现Lock类的门廊区，所以时间戳系统必须是无等待的。构造这种无等待的并发时间戳系统（参见本章注释）是可行的，但其构造过程非常复杂且技术要求也非常高。取而代之的是，我们重点关注一种更为简单的问题，只考虑其自身的正确：构造一个串行的时间戳系统。在该系统中并发线程互不干扰地交替执行扫描-标记操作，就好像每个扫描-标记操作是通过互斥来完成的。换句话说，只考虑这样的执行，即线程能完成对其他线程label的一次扫描（或一个扫描），然后指定一个新的label，每个这样的操作序列是一个单独的原子操作步。并发时间戳系统和串行时间戳系统的原理其本质是相同的，只是在细节上有所差别。

```
1  public interface Timestamp {
2    boolean compare(Timestamp);
3  }
4  public interface TimestampSystem {
5    public Timestamp[] scan();
6    public void label(Timestamp timestamp, int i);
7  }
```

图2-10 时间戳系统的接口

所有的时间戳都可以看作一个有向图（称为覆盖图）中的结点。结点a到结点b的边则表

示a的时间戳比b的晚。时间戳的次序是反自反的：任意结点a不存在从自己出发指向自己的边。时间戳的次序也是反对称的：若存在一条从a到b的边，则必不存在从b到a的边。注意，并不要求时间戳的次序是可传递的：虽然存在一条从a到b的边和一条从b到c的边，但并不一定存在一条从a到c的边。

给一个线程指定一个时间戳可以看作是将该线程的令牌放在那个时间戳的结点上。线程通过定位其他线程的令牌来完成扫描，然后通过将自己的令牌移到结点a来为自己指定一个新的时间戳，并使得从a到其他所有的线程结点都存在一条边。

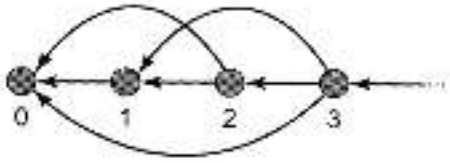


图2-11 无界时间戳系统的前趋图。结点代表自然数的集合，边表示自然数之间的全序关系

实际编程中，这种系统是作为一个单写者/多读者域所组成的数组来实现的，其中数组元素A代表线程A最近放置其令牌的有向图结点。scan()方法可以获取该数组的一个“快照”。线程A的label()方法将会修改数组的第A个元素。

34

图2-11是Bakery锁中无界时间戳系统的前趋图。显然它是无限的：每一个自然数都有一个结点，且只要$a > b$，就存在一条从a到b的有向边。

考虑图2-12中的前趋图T^2。图中有三个结点，分别标记为0、1和2，其中边定义了结点集上的次序关系，0小于1，1小于2，2又小于0。如果只有两个线程，则可以使用这个图定义一个有界（串行的）时间戳系统，该系统满足下面不变式：两个线程的令牌总是放在相邻的结点上，边的方向表示它们的相对次序。假设A的令牌在结点0上，B的令牌在结点1上（所以A具有较晚的时间戳）。对于A来说，方法label()是平凡的：因为它已经是最晚的时间戳，所以不做任何动作。对于B来说，方法label()则“跳过”A的结点从0变为2。

有向图中的环（cycle）[⊖]是指一系列结点n_0，n_1，…，n_k，其中有一条边从n_0到n_1，有一条边从n_1到n_2，最后有一条边从n_{k-1}到n_k，并有一条边从n_k返回n_0。

因为T^2中唯一判环的长度为3，且只有两个线程，所以线程之间的次序是确定的。对于两个以上的线程，需要附加的概念工具。令G是一个前趋图，A和B是G的子图（可能为单个结点）。若A的每个结点都有指向B中所有结点的边，则称图G中A支配B。图的乘法则定义为下面这种不可交换的复合运算符（记为$G \circ H$）：

用H的一个拷贝（表示为H_v）来替换G中的每一个结点v，且如果在图G中v支配w，则在$G \circ H$中H_v支配H_w。

递归地定义图T^k如下：

1. T^1是单个结点。
2. T^2是前面所定义的三结点图。
3. 对于$k>2$，$T^k = T^2 \circ T^{k-1}$。

35

例如，图2-12描述了图T^3。

前趋图T^n是n线程有界串行时间戳系统的基础。可以采用三进制概念，用$n-1$位数字对图T^n中的任意结点过行“编址”。例如，T^2中的结点被编址为0、1和2。T^3中的结点被标识为00、01、…、22，其中高位数字指三个子图中的一个子图，低位数字指相应子图中的一个结点。

n线程标记算法的关键就在于所有被令牌覆盖的结点决不会形成闭环。正如前面所指出的，在T^2中两个线程不可能形成闭环，因为T^2中最短的闭环需要三个结点。

⊖ “cycle”来自于相同的希腊词根“circle”。

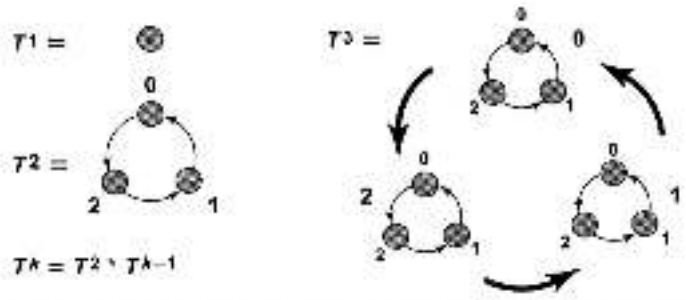


图2-12 有界时间戳系统的前驱图。初始时，令牌A在结点12上（子图1中的结点2），令牌B和令牌C分别位于结点21和22上（子图2中的结点1和结点2）。令牌B准备移到结点20以支配其他的令牌，然后令牌C准备移到21以支配其他令牌，令牌B和令牌C都可以继续在子图2的T^2中无限循环。如果A打算移动以支配B和C，那么在子图2中不可能找出一个结点，因为子图2已经满了（任意子图T^n最多可容纳n个令牌）。于是，将令牌A移到结点00。若现在B要移动，它将选择结点01，C将选择10，以此类推

关三个线程label()方法是如何工作的？当A调用label()时，如果其他两个线程在同一个子图T^2中都有令牌，那么令牌将移动到下一个最高子图T^3中的某个结点上，该子图的所有结点都支配前面的子图T^2。例如，考虑图2-12中的T^3，假设初始状态没有初始，令牌A位于结点12上（子图1中的结点2），令牌B和令牌C分别位于结点21和22上（子图2中的结点1和结点2），令牌B准备移动到20以支配其他令牌，然后令牌C准备移动到21以支配其他令牌，B和C可以继续在子图2的T^2内无限循环。若此时A要移动到支配B和C的位置，那么它无法在子图2中选择出一个结点，因为子图2已经满了（任意子图T^n最多可容纳n个令牌）。于是，令牌A移动到结点00。若现在B要移动，它将选择结点01，C将选择10，以此类推。

2.8 存储单元数量的下界

Bakery锁是简洁、优美且公平的。那么它为什么不实用呢？最主要的问题就是要读/写n个不同的存储单元，其中n（n可能非常大）是并发线程的最大个数。

是否存在更好的基于读/写存储器的Lock算法可以避免这种开销呢？下面来证明答案是否定的。也就是说，任意一种无死锁的Lock算法在最坏情况下至少需要读/写n个不同的存储单元。该结论非常重要，正是因为这样的结论，才促使我们在多处理器机器中，增加一些功能要比读/写操作更加强大的同步操作，并以这些操作作为互斥算法的基础。

在讨论实用的互斥算法之前，我们首先证明为什么这种线性下界是解决互斥问题时所固有的。下面将会看到只能通过读/写指令（实际中称为载入和存储）访问的存储单元具有如下重要限制：一个线程向某指定单元写的任何信息，在其他线程读取之前可能会被重写（覆盖）。

为了完成证明，首先介绍多线程程序使用的存储器状态的概念。对象的状态就是该对象域的状态。线程的局部状态就是该线程局部变量和程序计数器的状态。全局状态或系统状态则是所有对象的状态以及所有线程的局部状态之和。

定义2.8.1 对于任意一个全局状态，若此刻有某个线程正在临界区内，而锁的状态却与一个没有线程在临界区内或正在尝试进入临界区的全局状态相符，则称该Lock对象的状态s是不一致的。

引理2.8.1 无死锁的Lock算法不可能进入不一致状态。

证明 假设Lock对象处于不一致状态s，且此时没有线程在临界区内或正在尝试进入临界区。如果线程B想要进入临界区，由于算法是无死锁的，因此它最终必成功进入。于是得到矛盾，因为B无法确定A是否处于临界区内。 □

[27] 任何解决无死锁互斥问题的Lock算法必定需要n个不同的存储单元。这里，只以3个线程的情况为例，说明被3个线程访问的无死锁Lock算法必须使用3个不同的存储单元。

定义2.8.2 Lock对象的覆盖状态是指这样的状态：至少有一个线程欲写所有的共享存储单元，而该Lock对象的存储单元"看上去"就好像临界区是空的（也就是说，这些存储单元的状态就像是既没有线程在临界区内也没有线程正在尝试进入临界区）。

在覆盖状态中，称一个线程覆盖它将要写的存储单元。

定理2.8.1 任意采用读/写存储器方式解决3线程无死锁互斥的Lock算法必须至少使用3个不同的存储单元。

证明 采用反证法，假设存在一种只使用两个存储单元解决3线程无死锁的Lock算法。在初始状态s中，没有任何线程处于临界区内或正在试图进入临界区。若有一个线程在运行，那么在进入临界区前该线程必须至少要写一个存储单元，否则，s是一个不一致状态。

由此推出，每个线程在进入临界区前都必须至少写一个存储单元。若共享存储单元都是类似于Bakery锁的单写者存储单元，那么显然需要3个不同的存储单元。

我们考虑类似于Peterson算法（图2-6）中victim那样的多写者存储单元的情况。令s是一个覆盖的Lock状态，其中A和B分别覆盖不同的存储单元。考虑从状态s开始的下面这种可能的执行情形，如图2-13所示：

让C单独运行。由于该Lock算法满足无死锁特性，C将最终进入临界区。然后，让A和B分别修改它们覆盖的存储单元，使该Lock对象处于状态s'中。

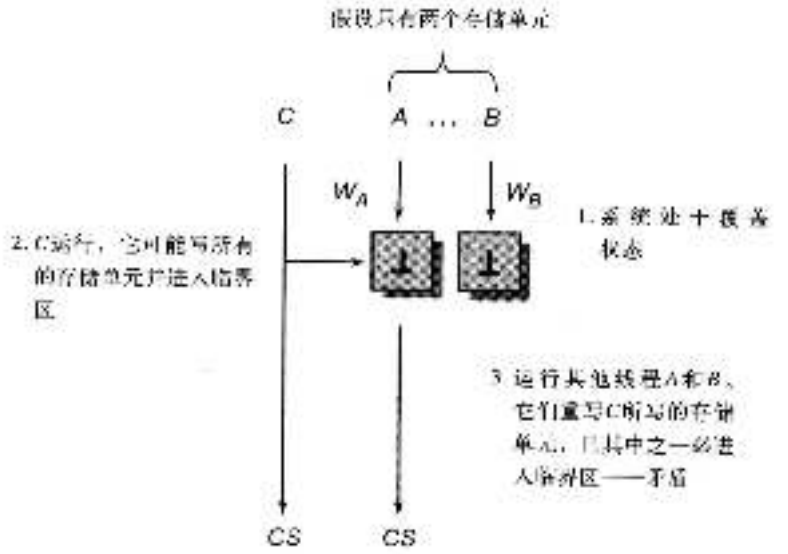


图2-13 对于两个存储单元使用覆盖状态导致矛盾。初始状态下两个存储单元均为空闲⊥

由于没有线程能够判断C是否在临界区中，所以状态s'是非一致的。因此，不可能有一个仅有两个存储单元的锁。

接下来的问题是如何设法使得线程A和B进入覆盖状态。考虑一种B三次通过临界区的执行情形。每一次通过时，B必须写其个存储单元，所以考虑它在试图进入临界区时写的第一个单元。由于只有两个存储单元，B必定对某个单元写了两次。称这个单元为L_B。

让B一直运行直到它准备第一次写单元L_B。若A现在正在运行，则由于B还没有写任何信息，因此A将进入临界区。A必须在进入临界区之前写L_A。否则，如果A只写L_B，则使得A进入临界区，B写L_B（冲掉A最后一次写的内容），结果将是一个不一致状态：B不能判断A是否在临界区内。

让A一直运行直到它第一次写L_A单元。这个状态不是一个覆盖状态，因为A可能已经对L_B写了某些信息以提示线程C它要进入临界区。让B运行，冲掉A向L_B写入的所有内容，最多三次进出临界区，且恰好在第二次写L_B前停停。注意，每次B进入和离开临界区，它向存储单元写的任何信息都不再有关系。

在这种状态下，A准备写L_A，B准备写L_B，并且存储单元是一致的（没有线程正在进入或正处于临界区内），正如覆盖状态所需的那样。图2-14描述了这个场景。 □

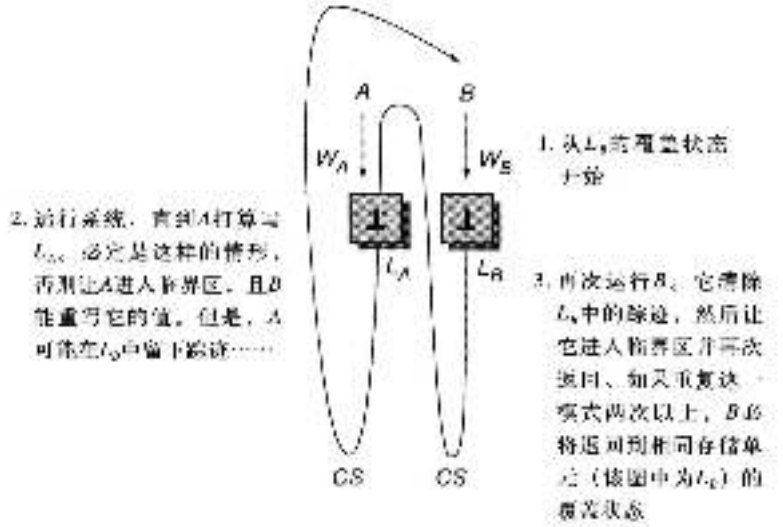


图2-14 到达了一个覆盖状态。在L_B的初始覆盖状态下，两个存储单元均为空值。

以上证明可以推广到n线程。n线程的无死锁互斥算法需要n个不同的存储单元。因此，Peterson和Bakery锁也是最优的（在不变因素下）。然而，正如我们所知道的，为每个Lock分配n个存储单元是不实际的。

该证明说明读/写操作所固有的限制：一个线程所写的信息可能在没有被其他线程读取之前又被重写了。在接下来的其他算法设计中要记住这个限制。

在后面的章节中，我们将会看到现代计算机系统结构提供了特殊的指令来克服读/写指令的"重写"限制，允许n线程锁的实现使用固定数量的存储单元。同时也将看到有效地利用这些指令来解决互斥问题并不是一件烦琐的事。

2.9 本章注释

牛顿关于时间流动性的概念出自他所撰写的著名的《原理》[121]一书。形式化的"→"归功于Leslie Lamport[89]。本章的前三个算法归功于Gary Peterson，他在1981年发表的一篇两页纸的文章[124]中提出了这些算法。本章介绍的Bakery锁是Leslie Lamport[88]所提出的

Bakery算法的一种简化版本。串行时间戳算法来自Amos Israeli和Ming Li[77]，他们提出了有界时间戳系统的概念。Danny Dolev和Nir Shavit[34]开发了第一个并发有界时间戳系统。其他的有界时间戳系统方案包括Sibsankar Haldar和Paul Vitányi[51]，Cynthia Dwork和Orli Waarts[37]。锁中域的数量的下界由Jim Burns和Nancy Lynch[23]提出，他们的验证方法——覆盖证明，被广泛地用于分布式计算中下界的证明。有兴趣的读者可以在Michel Raynal[131]的经典著作中找到更多关于互斥算法的历史资料。

38/40

2.10 习题

习题9. 对于一个给定的互斥算法，定义r-有界等待为：如果$D_A^i \to D_B^i$，则$CS_A^i \to CS_B^{i+r}$。是否存在一种定义Peterson算法门廊的方法，使得对于某个值r，该算法能够支持r-有界等待？

习题10. 为什么需要定义门廊区？为什么不能在基于lock()方法中第一条指令被执行的次序的互斥算法中定义先来先服务（FCFS）？根据lock()方法执行第一条指令的方式——对不同单元或相同单元的读和写，逐一地证明你的结论。

习题11. Flaky计算机公司的程序员设计了一个如图2-15所示的协议，以保证n线程的互斥。对于以下每个问题，或证明其成立，或给出一个反例。

- 该协议满足互斥特性吗？
- 该协议是无饥饿的吗？
- 该协议是无死锁的吗？

```
class Flaky implements Lock {
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = Thread10.get();
    do {
      do {
        turn = me;
      } while (busy);
      busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

图2-15 习题11的Flaky锁

习题12. 证明过滤锁允许某些线程任意次数地超过其他线程。

习题13. 双线程Peterson锁的一种改进方案就是在一根二叉树中排列一系列双线程Peterson锁。假设n为2的幂。为每一个线程指定一个叶子锁，该锁可以由另一个线程共享。每个锁将共享自己的两个线程视为线程0和线程1。

41

在树-锁请求中，线程依次获得从该线程对应的叶子直到树根的所有双线程Peterson锁。在树-锁释放中，从二叉树的根直到叶子释放该线程已获得的每个双线程Peterson锁。在任何时候，一个线程都可能被延迟一段有限的时间。（换句话说，线程可以打个盹，甚至可以放个假，但它们始终不会死掉。）对于下述每种特性，或证明扩展锁能保持这种特性，或给出一个执行反例（可能是无限的）说明它不具备该特性。

1. 互斥。
2. 无死锁。
3. 无饥饿。

从一个线程开始请求树-锁直到成功获得树-锁这一时间段内，树-锁被请求和释放的次数是否存在上界？

习题14. ℓ-互斥是无饥饿互斥的一种演变。它具有如下两个变化：在同一时刻，可能有ℓ个线程处于临界区内；在临界区内，可能有小于ℓ个线程会失败（中止）。

我们的实现必须满足下列条件：

ℓ-互斥：任何时候，至多有ℓ个线程同时处于临界区内。

ℓ 无饥饿：只要处于临界区内的线程个数少于ℓ，则某个线程想要进入临界区，最终必将成功（即使临界区内的某些线程已经中止）。

修改n进程Filter互斥算法使其变为ℓ-互斥算法。

习题15. 实际应用中，几乎所有的锁请求都是无争用的，因此衡量锁性能的一种实用标准就是在没有其他线程同时请求锁的情况下，一个线程获得锁所需的操作步。

Cantaloupe-Melon大学的科学家设计了针对任意锁的“包装器”，如图2-16所示。同时指出，如果基本的Lock类具有互斥和无饥饿特性，那么FastPath锁也具有这两个特性，且在无争用的情况下能在常数步内获得锁。试论述为什么他们的结论是正确的，或者给出一个反例。 42

```
1  class FastPath implements Lock {
2    private static ThreadLocal<Integer> myIndex;
3    private Lock lock;
4    private int x, y = -1;
5    public void lock() {
6      int i = myIndex.get();
7      x = i;                    // I'm here
8      while (y != -1) ;         // is the lock free?
9      y = i;                    // me again?
10     if (x != i)               // Am I still here?
11       lock.lock();            // slow path
12   }
13   public void unlock() {
14     y = -1;
15     lock.unlock();
16   }
17 }
```

图2-16 习题15的FastPath互斥算法

习题16. 假设n个线程调用图2-17所示Bouncer类的visit()方法。证明：

```
1  class Bouncer {
2    public static final int DOWN = 0;
3    public static final int RIGHT = 1;
4    public static final int STOP = 2;
5    private boolean goRight = false;
6    private ThreadLocal<Integer> myIndex; // initialize myIndex
7    private int last = -1;
8    int visit() {
9      int i = myIndex.get();
10     last = i;
11     if (goRight)
12       return RIGHT;
13     goRight = true;
14     if (last == i)
15       return STOP;
16     else
17       return DOWN;
18   }
19 }
```

图2-17 Bouncer类的实现

- 最多只有一个线程获得值STOP。
- 最多有$n-1$个线程获得值DOWN。
- 最多有$n-1$个线程获得值RIGHT。

[43] 注意，后两个证明并不是对称的。

习题17. 到目前为止，我们假设n个线程都具有唯一的小标识码。下面是一种给线程指定唯一小标识码的方法。在一个三角矩阵中排列Bouncer对象，每个Bouncer对象有一个如图2-18所示的id。每个线程都从访问Bouncer 0开始。如果它得到STOP，则停止，如果它得到RIGHT，则访问1，如果它得到DOWN，就访问2。通常情形下，如果某个线程得到STOP，则停止。如果得到RIGHT，则访问同一行中的下一个Bouncer；如果得到DOWN，就访问同一列中的下一个Bouncer。每个线程都获得它停止时的那个Bouncer对象的id。

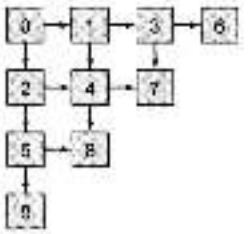


图2-18 Bouncer对象的数组布局

- 证明每个线程最终都将停在某个Bouncer对象上。
- 若事先知道总的线程数n，那么数组中需要多少个Bouncer对象？

习题18. 试举反例证明，对于串行时间戳系统7，若从一个有效的初始状态（label之间没有循环）开始，该系统并不支持3个线程并发地工作。注意，可以有两个相同的label，因为可以用线程ID来破坏这种联系。所举的反例中需要给出一种三个label之间不满足全序关系的执行状态。

习题19. 串行时间戳系统7只有3个可能的不同label值。试设计一个只需n2个label的串行时间戳系统。注意，在一个时间戳系统中，线程可以查看所有的label来选择一个新的label，然而一旦这个label被选定，那么它不用知道系统中其他的label是什么就可以与它们相比较。提示：考虑label的位表示法。

习题20. 采用无界label，给出图2-10所示Timestamp接口的Java代码。然后，说明如何使用你的
[44] Timestamp Java代码来替换图2-9中Bakery锁的伪代码[82]。

并发对象

并发对象的行为能够用它们的安全性和活性有效地进行描述，通常称为正确性和演进性。本章讲述并发对象正确性和演进性的相关概念及其定义。

虽然并发对象的正确性基于其与顺序行为等价的概念，然而，不同的概念适用于不同的系统。考虑下面三种正确性条件，静态一致性适用于以相对较弱的对象行为约束代价获得高性能的应用，顺序一致性是一种较强的约束形式，通常用于描述类似于硬件存储器接口这样的底层系统。可线性化特性是一种更强的约束，适用于描述由可线性化组件构成的高层系统。

在正确性保障的多个空间维上，方法的不同实现提供了各种不同的演进保障。有些是可阻塞的，即任一线程的延迟能够延迟其他的线程，有些则是非阻塞的，即一个线程的延迟不能延迟其他的线程。

3.1 并发性与正确性

并发对象的正确性究竟指的是什么呢？图3-1描述了一种简单的基于锁的先进先出（FIFO）并发队列。其中，enq()和deq()方法采用了第2章介绍的互斥锁来获得同步。不难看出这样的实现是一个正确的并发FIFO队列。因为每个方法在访问和修改域时都持有互斥锁，所以这种方法调用能够获得顺序的执行效果。

图3-2描述了这种队列实现的思想，图中展示了这样一种执行场景：*A*使元素*a*入队，*B*使元素*b*入队，*C*做了两次出队操作，第一次抛出空异常EmptyException，第二次返回*b*。垂直区间表示并发的方法调用。三个方法调用在时间上相互重叠，在这个图示中，时间从左向右移动，黑线代表时间间隔。单个线程的时间间隔沿着一条单水平线来描述。为方便起见，线程的名字标记在水平线的左侧。一个栅栏表示一段具有固定起始时间和停止时间的时间间隔。右侧为虚线的栅栏表示具有固定起始时间和不确定停止时间的时间间隔。符号“*q*.enq(*x*)”表示线程使元素*x*在对象*q*中入队，“*q*.deq(*x*)”则表示线程使*x*从对象*q*中出队。

时间线说明哪个线程持有锁。在图3-2中，*C*首先获得锁，发现队列为空，于是抛出一个异常并释放锁，*C*不修改队列。接着*B*获得锁，向数组中插入*b*然后释放锁。接下来*A*获得锁，向数组中插入*a*然后释放锁。*C*再次获得锁，使*b*出队，释放锁并且返回。所有调用的执行产生了一种顺序的执行效果，并且很容易验证*b*先于*a*出队，这与通常的顺序FIFO队列的行为是一致的。

图3-3给出了并发队列的另一种实现（该队列仅在单入队者和单出队者共享使用时才能正确地工作）。它的时间间隔描述与图3-1基于锁的队列几乎相同，唯一的区别是没有锁。可以认为这个单入队者/单出队者FIFO队列的实现是正确的，虽然解释其理由不再那么容易。甚至当入队者和出队者并发时，一个队列是FIFO的到底意味着什么也并不是十分清楚。

然而，Amdahl定律（第1章）已指出，持有互斥锁的并发对象（因此也是一个接一个地有效执行）不如具有细粒度锁或根本没有锁的对象令人满意。因此，我们需要一种不依赖于在方法层次上加锁的方式，来规范并发对象的行为以及分析它们的实现过程。尽管如此，上述

```
1  class LockBasedQueue<T> {
2    int head, tail;
3    T[] items;
4    Lock lock;
5    public LockBasedQueue(int capacity) {
6      head = 0; tail = 0;
7      lock = new ReentrantLock();
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) throws FullException {
11     lock.lock();
12     try {
13       if (tail - head == items.length)
14         throw new FullException();
15       items[tail % items.length] = x;
16       tail++;
17     } finally {
18       lock.unlock();
19     }
20   }
21   public T deq() throws EmptyException {
22     lock.lock();
23     try {
24       if (tail == head)
25         throw new EmptyException();
26       T x = items[head % items.length];
27       head++;
28       return x;
29     } finally {
30       lock.unlock();
31     }
32   }
33 }
```

图3-1 基于锁的先进先出队列。队列元素存储在数组items中，head是下一个出队元素的索引号，tail是第一个空数组槽（以capacity为模）的索引号。lock域是保证方法互斥执行的锁。初始状态下head和tail均为0，队列为空。若enq()发现队列已满，也就是head和tail相差一个队列长度，那么它将抛出一个异常。否则，仍有空对空间，enq()则在数组入口tail处存入元素，并使tail增加1。deq()方法按照对称的方式工作

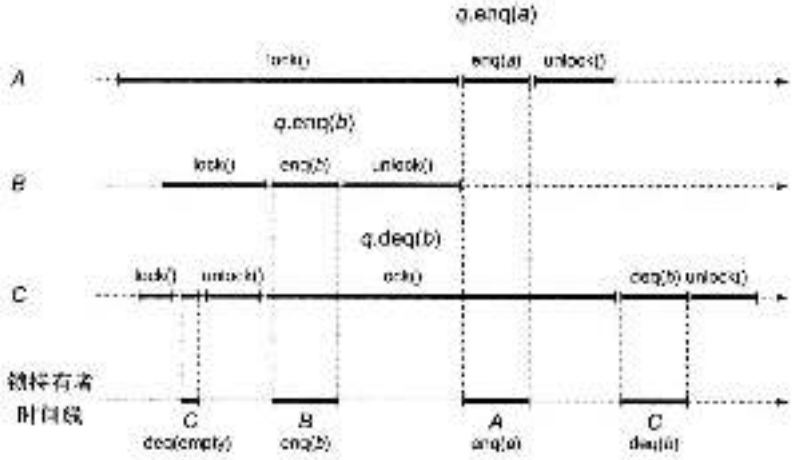


图3-2 锁队列的执行过程。C首先获得锁，发现队列为空，抛出一个异常并释放锁。然后B获得锁，向数组中插入b然后释放锁。接下来A获得锁，向数组中插入a然后释放锁。C再次获得锁，使b出队，释放锁并且返回

基于锁的例子仍然阐述了一个很有用的原则：如果能将并发执行转换为顺序执行，则只需对该顺序执行进行分析，从而简化了并发对象的分析。这条原则也正是本章关于正确性的关键准则。

46 ↓ 47

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[]) new Object[capacity];
6    }
7    public void enq(T x) throws FullException {
8      if (tail - head == items.length)
9        throw new FullException();
10     items[tail % items.length] = x;
11     tail++;
12   }
13   public T deq() throws EmptyException {
14     if (tail - head == 0)
15       throw new EmptyException();
16     T x = items[head % items.length];
17     head++;
18     return x;
19   }
20 }
```

图3-3 单入队者/单出队者FIFO队列。在这个构造中，除了不需要使用锁机制来协调访问以外，其他均与基于锁的FIFO队列相同

3.2 顺序对象

在Java和C++等语言中，对象就是一个包含有数据的容器。每个对象都提供一系列的方法，只有通过这些方法才能对该对象进行操作。每个对象都有一个类，类定义了对象的方法以及方法的行为。每个对象都具有良构的状态（例如，FIFO队列的当前元素序列）。有多种可以描述对象方法行为的方式，从直观的自然语言直到抽象的形式化定义。我们经常用到的应用程序接口（API）文档是处于它们中间的一种方式。

API文档的内容一般如下：若调用方法之前对象处于某个状态，则调用返回时将处于另外的某个状态，该调用会返回某个特定的值或抛出一个特定的异常。显然，这种描述可分为前置条件（描述对象在方法调用前的状态）和后置条件（描述调用返回时对象的状态及其返回值）。对象状态的变化有时称为副作用。例如，考虑如何来描述一个先进先出（FIFO）队列的类。该类提供了两个方法：enq()和deq()。队列的状态就是其中元素的序列，可以为空。如果队列状态为序列q（前置条件），那么enq(x)调用将使队列的状态变为$q \cdot x$，其中"$\cdot$"表示级联。如果队列对象不为空（前置条件），记作$a \cdot q$，那么deq()方法将移出并返回队列中的第一个元素a（后置条件），同时使队列的状态变为q（副作用）。反之，如果队列对象为空（前置条件），deq()方法将抛出EmptyException异常并保持队列状态不变（后置条件）。

48

这种说明文档称为顺序规范，是一种常用的方法，它具有简单明了、功能强大的特点。由于每个方法需要单独描述，因此对象文档的长度与方法的个数呈线性关系。在各个方法之间存在着大量可能的交互，所有这些交互都可以通过方法在对象状态上的副作用来刻画。对

象的说明文档描述了每次方法调用前后的对象状态，但是忽略了在方法调用执行过程中对象可能出现的中间状态。

在由单线程对一组对象进行操作的顺序计算模型中，采用前置条件和后置条件来定义对象是非常有效的。然而，对于多线程共享的对象，这种常用的有效文档并不适用。若一个对象的方法可以被并发线程调用，那么这些调用在时间上可以相互重叠，讨论它们之间的调用顺序就不再有意义了。在一个多线程程序中，若x和y在重叠的时间间隔中都要从一个FIFO队列中出队，这将意味着什么呢？哪一个会先出队？能否继续采用前置/后置条件独立地描述每个方法，或者是否必须对并发调用之间的各种可能的交互情形都提供显式描述呢？

共享对象的状态这一概念也会变得模糊不清。在单线程程序中，必须假定对象在方法调用之间存在着一个有意义的状态。[⊖]然而对于并发对象，重叠的方法调用可能时刻都在进行，因此对象有可能根本不会处于方法调用之间的某个状态。每个方法调用都可能面对着一组由其他的并发方法调用所产生的不完整效果的对象状态，这个问题在单线程程序中显然不会发生。

3.3 静态一致性

要直观地了解并发对象的执行行为，可以考察一些包含有简单对象的并发计算实例，分析它们在各种情形下的行为是否和我们直觉上所预想的并发对象的行为相一致。

方法的调用需要时间，方法调用是一段时间间隔，从调用事件开始直到响应事件结束。
49 并发线程的方法调用可以相互重叠，而单线程的方法调用总是顺序的（无重叠，一个接一个地）。如果一个方法的调用事件已发生，但其响应事件还未发生，则称这个方法调用是未决的。

出于一些历史上的原因，通常将基于读/写方式存储单元的对象版本称作寄存器（见第4章）。在图3-4中，两个线程并发地向共享寄存器r写入-3和7（如前所述，"r.read(x)"表示线程从寄存器r中读x，"r.write(x)"表示线程向寄存器r中写x），随后，一个线程读r并返回值-7。这显然是不能接受的，我们希望寄存器中不是7就是-3，而不是两者的混合。这个例子阐述了如下原则：

原则3.3.1 方法调用应呈现出以某种顺序次序执行且每个时刻只有一个调用发生。

由于这个原则本身太弱，所以在实际中并不实用。例如，它允许读操作总是返回对象的初始状态，即使在顺序执行中也是如此。

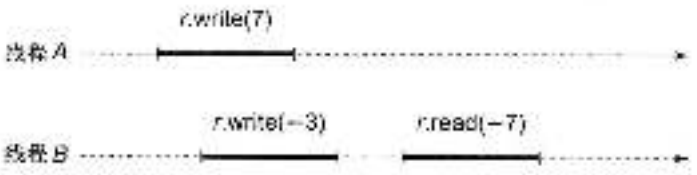


图3-4 为什么每个方法调用应该呈现出具有瞬间发生的效果？两个线程并发地对共享寄存器r写入-3和7，随后，一个线程读r并返回了-7。我们希望寄存器中不是7就是-3，而不是两者的混合

⊖ 存在一个例外，当一个方法部分改变了对象的状态，然后调用该对象的另一个方法时，必须引起注意。

下面是一个稍强的约束条件，若一个对象中不存在未决的方法调用，则该对象是静态的。

原则3.3.2 由一系列静止状态分隔开的方法调用应呈现出与按照它们的实时调用次序相同的执行效果。

例如，假设对于一个FIFO队列，A和B并发地将x和y入队，然后队列变为静态的，C再使z入队。队列中x和y的相对次序可能无法确定，但可以肯定它们都在z的前面。

总的来说，原则3.3.1和3.3.2定义了一种正确性特性，称为静态一致性。非形式化地来说，静态一致性是指在任一时刻若对象变为静态的，那么到此刻为止的执行等价于目前已完成的所有方法调用的某种顺序执行。

以第1章中的共享计数器作为一个静态一致性对象的例子。静态一致的共享计数器应该返回整数数字，虽然不必按照getAndIncrement()的调用次序，但是不允许出现任何数字的重复和遗漏。静态一致对象的执行就好像沧座位游戏一样：任何时刻，音乐都可能停止，即状态变为静态的；在音乐停止的瞬间，每一个未决的方法调用都必须返回一个索引，所有索引一起来满足顺序计数器的说明规范，保证没有重复或者遗漏的数字。换言之，静态一致的计数器是一个索引分发机制，类似于程序中的"循环计数器"，但不用考虑索引分发的次序。 50

评析

静态一致性对并发性的限制到底有多大？具体而言，也就是说在什么环境下静态一致性会使得一个方法调用阻塞等待另一个调用完成？令人不可思议的是，答案（原则二）是绝不会阻塞。如果一个方法对对象的所有状态都给出了定义，则称该方法是完全的；否则称为部分的。例如，考虑下面这种针对顺序无界FIFO队列的规范说明：总是能够使得一个元素入队，但是只能从非空队列中出队。在这个FIFO队列的顺序说明中，enq()是完全的，因为它对队列的所有状态都定义了执行效果；而deq()则是部分的，因为它只定义了非空队列的执行效果。

在并发执行中，对于完全方法的任何一个未决调用，都必定存在着一个静态一致的响应，该结论只说明正确性条件本身在这种方式中并不成立，而没有说明响应的具体值是可以（或总是能够）确定的。静态一致性是一种非阻塞的正确性条件，关于这一点将在3.6节中进一步解释。

对于正确性P，如果系统中每个对象都满足P，则整个系统也满足P，那么P是可复合的。复合性在大型系统中十分重要，任何复杂系统都是采用模块方式设计和实现的。各组件都是独立设计、实现以及证明其正确性的。每个模块都将功能实现（被隐藏的）与模块接口（准确地表述了对其他模块提供的保证）明确地区分开来。例如，若一个并发对象的接口声明它是一个顺序一致的FIFO队列，那么该队列的用户不需要知道这个队列是如何实现的。把每个在接口上相互依赖的正确模块组合起来，其结果应该是一个正确的系统。事实上，能否将一组单独实现的静态一致对象组合起来构造一个静态一致的系统呢？答案是可以的，即静态一致性是可复合的，所以能够用静态一致对象复合构造更为复杂的静态一致对象。

3.4 顺序一致性

在图3-5中，一个单线程先后向共享寄存器r写入7和-3，随后它读r并返回了7。在某些应用中并不接受这样的行为，因为线程读的值并不是它最近写入的值。一个单线程的方法调用次序称为该线程的程序次序。（多个不同线程的方法调用与程序次序无关。） 51

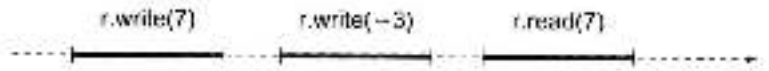


图3-5 为什么方法调用应该呈现出按照程序次序执行的效果？因为线程读的值不是最近写入的值，所以这种行为效果是不可接受的

在这个例子中，操作的调用并没有按照程序次序执行。因此，这个实例向我们提出了另外一个原则：

原则3.4.1 方法调用应该呈现出按照程序次序调用的执行效果。

这条原则保证纯粹的顺序计算具有我们所期望的行为。

原则3.3.1和原则3.4.1定义了一种正确性特性，称为顺序一致性，该特性在多处理器同步的文献中被广泛地使用。

顺序一致性要求方法调用的执行行为具有按照某种顺序次序的执行效果，并且这种顺序执行的次序应该与程序次序保持一致，也就是说，在任意的并发执行中，都存在着一种办法能够得方法调用按照某种顺序次序排序，并且这种顺序次序(1)与程序次序相一致，(2)满足对象的顺序规范说明。可以有多种调用次序满足这个条件，在图3-6中，线程A和B分别同时入队x和y，然后，A和B分别同时出队y和x，有两种可能的顺序次序说明它们的结果：(1) A入队x，B入队y，B出队x，A出队y；(2) B入队y，A入队x，A出队y，B出队x。这两种次序都与程序次序一致，其中任意一种都足以说明该执行是顺序一致的。

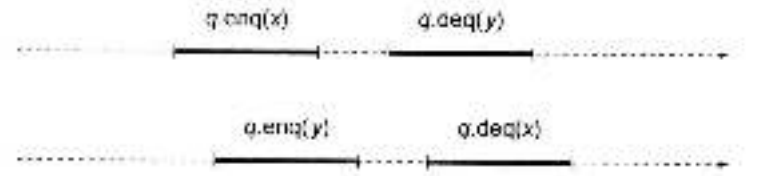


图3-6 有两种可能的顺序次序能够验证这种执行。两种次序都与方法调用的程序次序相一致，任何一种都说明执行是顺序一致的

评析

值得注意的是，顺序一致性与静态一致性之间是不可比的：存在着非静态一致但却是顺序一致的执行，反之亦然。静态一致性中不需要保持程序次序，而顺序一致性也不受静止状态周期的影响。

52

大多数现代的多处理器系统结构中，存储器的读/写都不是顺序一致的：这些读/写操作可以通过复杂的方式重新安排。大多数时候无法察觉，因为大部分读/写操作并不是作为同步操作来使用的，在那些程序员需要顺序一致的特殊情形中，必须显式地申请。这种系统结构提供了特殊的指令（通常称作内存屏障或内存栅栏），控制处理器按照需求对存储器传送修改，保证读/写操作正确地交互，从而最终实现指定的顺序一致性。3.8节将进一步讨论与顺序一致性相关的问题以及Java程序设计语言的详细内容。

在图3-7中，线程A使x入队，然后B使y入队，最后A使y出队，这个执行过程违反了直观上理解的FIFO队列行为：入队x在出队y开始前已经完成，虽然y后于x入队，但它却先出队。但是，这个执行过程却是顺序一致的。虽然x的入队调用在y的入队调用之前发生，但这些调用

按照程序次序是相互无关的，所以顺序一致性与它们的次序重排无关。

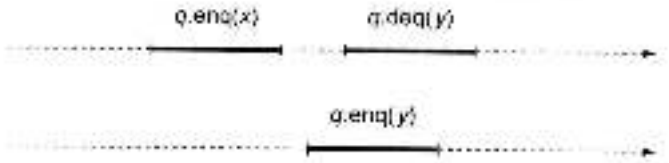


图3-7 顺序一致性与实时次序的对照。线程A先x入队，然后B使y入队，最后A使y出队。这个执行过程也许违反了直观上理解的FIFO队列行为，因为x的入队调用在y的出队调用开始前已经完成，所以虽然y后于x入队，但它还是先出队。尽管如此，这个执行过程仍是顺序一致的

那么重排不同线程的彼此不重叠的方法调用是否可被接受呢？举个例子，如果你在星期一存入工资，但是由于银行在提款后重排了存款的顺序，导致到下一个星期五才退出租金收据，这将使你感到非常恼火。

顺序一致性和静态一致性一样也是非组合的：对完全方法的任何未决调用总是能够完成。

顺序一致性是否可复合呢？也就是说，由多个顺序一致对象组合成一个整体是否也具有顺序一致的特性？很不幸，答案是否定的。图3-8中有两个线程A和B，分别对队列对象p和q调用入队方法和出队方法。不难看出p和q各自都是顺序一致的：p的方法调用序列与图3-7所示的顺序一致执行是相同的，q的行为与之类似。但是，将它们作为一个整体来看，其执行却不是顺序一致的。

53

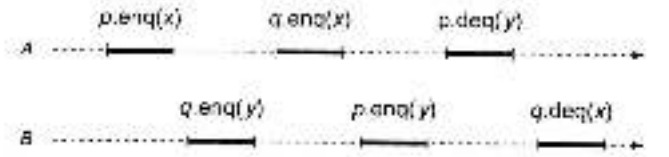


图3-8 顺序一致性是不可复合的。两个线程A和B分别对队列对象p和q调用入队方法和出队方法。不难看出p和q各自都是顺序一致的，然而作为一个整体其执行却不是顺序一致的

我们来证明不存在这种正确的顺序执行，其中的方法调用能够以一种与程序次序相一致的次序进行排序，采用反证法，假设这些方法调用能够重新排列形成一个正确的FIFO队列执行，其中方法调用的次序与程序次序相一致。我们使用标记$\langle p.enq(x)\ A\rangle \rightarrow \langle q.deq(x)\ B\rangle$表示任意的顺序执行必须使得A对p的入队x操作先于B对q的出队x操作，以此类推。由于p是FIFO的且A从p中出队y，则y肯定在x之前入队：

$$\langle p.enq(y)\ B\rangle \rightarrow \langle p.enq(x)\ A\rangle$$

同理，

$$\langle q.enq(x)\ A\rangle \rightarrow \langle q.enq(y)\ B\rangle$$

但程序次序说明

$$\langle p.enq(x)\ A\rangle \rightarrow \langle q.enq(x)\ A\rangle \text{ 且 } \langle q.enq(y)\ B\rangle \rightarrow \langle p.enq(y)\ B\rangle$$

综上所述，这种排序序列形成了一个环。

3.5 可线性化性

顺序一致性的根本缺陷在于它是不可复合的：将多个顺序一致的部分组合起来，其结果并不一定是顺序一致的。下面提出一种解决该问题的方法。顺序一致性要求方法调用必须与程序次序相一致，我们用一种更强的约束条件来替换这一要求：

原则3.5.1 每个方法调用都应该表现出一种与它的调用和响应之间的某个时刻的行为相同的瞬时效果。

这个原则说明方法调用的实时行为必须被保持，这种正确性特性称为可线性化性。可线性化的执行都是顺序一致的，反之则不成立。

54

3.5.1 可线性化点

用来说明并发对象实现是可线性化的一种常用办法就是对每个方法在它生效的那个地方指定一个可线性化点。对于基于锁的实现来说，每个方法的临界区可以当作它的可线性化点。对那些不使用锁的实现，可线性化点通常是该方法调用的结果对其他方法调用可见时的那个操作步。

以图3-3所示的单入队者/单出队者队列为例。在这个实现中没有临界区，但是仍然能够识别其可线性化点。它的可线性化点依赖于执行过程。若调用返回一个元素，则在head域被更新时（第17行），deq()方法有一个可线性化点。若队列为空，则在它抛出空异常EmptyException时（第15行），deq()方法有一个可线性化点。enq()方法的分析与之类似。

3.5.2 评析

顺序一致性是一种描述独立系统（例如硬件存储器）的有效方法。在这样的系统中不存在复合性问题。而可线性化性则非常适合描述大型系统的组件，在这种系统中各个组件必须独立地实现和验证。此外，实现并发对象所用的技术全都是可线性化的。由于我们重点对能够保持程序次序和复合性的系统感兴趣，所以本书中大多数（不是全部）数据结构都是可线性化的。

可线性化性对并发的限制有多大呢？与顺序一致性一样，可线性化性也是非阻塞的。然而，可线性化性又是可复合的，可线性化对象组合在一起仍然是一个可线性化的对象。这一点与顺序一致性不同，但与静态一致性相同。

3.6 形式化定义

现在来考虑更为精确的描述。本节着重于可线性化特性的形式化定义，其原因在于这一特性是书中最常用的性质。针对静态一致性和顺序一致性的类似定义则留作习题。

非形式化地来看，如果并发对象的每次方法调用都可以看作具有与其被调用和响应之间的某个时刻的行为相同的瞬时效果，那么这个并发对象是可线性化的。对于大多数非形式化的推理，这种断言就已足够了，但是对于一些更细致的情形（如还未返回的方法调用），则需要精确的形式化公式，进行更加严谨的论证。

55

并发系统的一次执行过程可以采用经历（history）模型来描述，经历是方法的调用事件和响应事件的一个有限序列。经历H的子经历就是H的事件序列中的一个子序列，方法的一次

调用记作<x.m(a*) A>，其中x是对象，m是方法名，a*是参数序列，A是线程。方法调用的一个响应记作<x:r(r*) A>，其中r或者是Ok或者是一个异常名，r*是结果值序列。有时我们把由线程A标记的一个事件看作A的一个操作步。

若一次调用与一个响应都具有相同的对象和线程，则称这个响应匹配这次调用。前面非形式化地使用了术语"方法调用"，这里给出一种形式化的定义：经历H中的一个方法调用是一个二元组，它由H中的一个调用和一个紧接其后且与其相匹配的响应所组成。必须把已经返回的调用与还未返回的调用区分开来：在H中，若一个调用还没有与之相匹配的响应，则该调用是未决的。H的一个扩展是这样的一个经历：对H中的零个或多个未决调用增加了相应的响应后构成的经历。有时可以忽略所有的未决调用：complete(H)是全部由匹配的调用和响应所构成的H的子序列。

有些经历中，方法调用相互间不重叠：如果H中的第一个事件是调用事件，除最后一个事件以外，H中的每个调用都紧随一个与之匹配的响应，则经历H是顺序的。

有时只需关注单一线程或单一对象的情形：H中线程A的子经历是指由H中所有线程名为A的事件组成的子序列，记作H|A（读作H在A上）。同理可以定义H中一个对象x的子经历H|x。余下的问题就是每个线程如何看待已发生的事件：对于两个经历H和H'，如果对于任意线程A都有H|A=H'|A，则称H和H'是等价的。最后，必须将没有意义的经历排除：如果H中每个线程的子经历都是顺序的，则H是良构的。本章所考虑的经历都是良构的，注意，一个良构经历的每个线程的子经历必定是顺序的，但它的对象的子经历却不一定是顺序的。

如何判断一个对象是否是一个真正的FIFO队列呢？假定存在一种有效的方法，可以判断任意一个对象的顺序经历对于这个对象的类来说是否是一个合法的经历。一个对象的顺序规范恰好就是该对象顺序经历的集合。如果每个对象的子经历对该对象都是合法的，则顺序经历H是合法的。

第2章中已指出，集合X上的偏序关系"→"是反自反和可传递的。也就是说，x→x决不会成立，且只要x→y以及y→z，就有x→z。注意，可能存在不同的x和y，使得x→y和y→x都不成立。集合X上的全序关系"<"也是一种偏序关系，但对于任意不同的x和y，必有x<y或者y<x。

任何偏序关系都能扩展为全序关系：

结论3.6.1 若"→"是集合X上的偏序关系，则必存在X上的全序关系"<"，使得如果x→y，则x<y。

在经历H中，如果方法调用m_0在方法调用m_1开始之前完成，则称m_0先于m_1；也就是说，m_0的响应事件在m_1的调用事件之前发生。基于这个概念，我们引入一些简写符号。给定一个包含方法调用m_0和m_1的经历H，若H中m_0先于m_1，则记为$m_0 \rightarrow_H m_1$。关于$\rightarrow_H$是一个偏序关系的证明留作习题。注意，如果H是顺序的，则$\rightarrow_H$是一个全序关系。给定一个经历H及一个对象x，且H|x中包含有方法调用m_0和m_1，如果H|x中m_0先于m_1，则记作$m_0 \rightarrow_x m_1$。

56

3.6.1 可线性化性

可线性化性所隐含的基本思想就是每一个并发经历都等价于某一个顺序经历（从下面的角度来看）。基本准则就是如果一个方法调用先于另一个方法调用，则较早的调用必定在较晚的调用前生效。反之，如果两个方法调用彼此重叠，则它们的次序将无法确定。可以按照任

意的次序对其进行排序。

更形式化地来讲。

定义3.6.1 *如果经历H存在有一个扩展H'及一个合法的顺序经历S，并使得*

L1 *complete(H')与S等价，且*

L2 *若在H中方法调用m_0先于m_1，那么在S中也成立。*

则称经历H是可线性化的。

S称作是H的一个线性化。(H可以有多个线性化。)

非正式地来看，把H扩展为H'意味着即使某些未决调用还没有给调用者返回响应，但它们有可能已经生效。图3-9说明了这样一个概念：必须完成未决的方法调用enq(x)以保证deq()调用返回x。第二个条件表明，如果某个方法调用在原先经历中先于另外一个调用，那么在该线性化中也必须保持它们之间的这种次序。

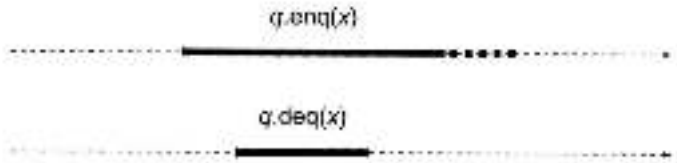


图3-9 未决的enq(x)调用必须提前生效来确保deq()调用返回x

3.6.2 可线性化性的复合性

可线性化性是可复合的：

57

定理3.6.1 *对于每个对象x，当且仅当H\|x是可线性化的，H是可线性化的。*

证明 “仅当”部分的证明留作习题。

对每个对象x，找出一个H\|x的线性化。令R_x是H\|x中构造这个线性化的所有响应的集合，令$\rightarrow_x$为对应的线性化次序。令H'是将R_x中的每个响应加入H中所构成的经历。

对H'中方法调用的个数进行归纳证明。在最基本的情况下，H'仅包含一个方法调用，结论显然成立。接下来，假设结论对所有的包含方法调用个数少于k（k>1）的H都成立。对每个对象x，考虑H'\|x中最后的方法调用。在这些调用中必定存在一个调用m，它对于关系$\rightarrow_x$是最大的调用；也就是说，不存在m'，使得$m \rightarrow_x m'$。令G'是从H'中移去m后得到的经历。由于m是最大的，所以H'等价于G'·m。由归纳假设可知，G'对于顺序经历S'是可线性化的，H'和H对于S'·m也都是可线性化的。 □

复合性是一个很重要的特性，它使得并发系统的设计和构造能够采用模块化的方式，不同的可线性化对象可以独立地实现、验证和执行。而基于不可复合正确性特性的并发系统必须依赖于一个集中式的调度器来调度所有的对象，或者要满足一些额外的附加在对象上的约束条件，以保证对象遵守一种相容的调度协议。

3.6.3 非阻塞特性

可线性化是一种非阻塞特性：一个完全方法的未决调用不需要等待另一个未决调用完成。

定理3.6.2 *令inv(m)是完全方法的一个调用。如果<x inv P>是可线性化经历H中的一个未决调用，则必定存在一个响应<x res P>使得H·<x res P>是可线性化的。*

证明 令S是H的任意一个线性化。如果S中包含有一个对应于<x inv P>的响应<x res P>，因为S也是$H \cdot$ <x res P>的一个线性化，所以结论成立。否则，因为按照定义，线性化不包含任何未决调用，所以<x inv P>也不会在S中出现。由于方法是完全的，所以存在一个响应<x res P>使得

$$S' = S \cdot \text{<x inv P>} \cdot \text{<x res P>}$$

是合法的。而S'是$H \cdot$ <x res P>的一个线性化，因此它也是H的一个线性化。 □

该定理说明可线性化性本身决不会使一个包含有对完全方法的未决调用的线程被阻塞。当然，阻塞（甚至死锁）有可能作为可线性化性特定实现中的人为因素而出现，但它并不是正确性本身所固有的。这条定理也说明可线性化性是一种适于并发性和实时性要求较高的系统的正确性条件。 58

非阻塞特性并不排除以显式方式说明的阻塞情形，例如，考虑某个线程试图对空队列进行出队操作的情形，让该线程阻塞直到其他线程向队列中写入元素应该是有意义的做法。为了能够发现这种对空队列出队的企图，队列规范中必须将deq()方法说明成部分的，该方法应用到空队列时的影响效果在说明中并不需要作出定义。对于部分顺序规范，最自然的并发解释就是使调用一直等待，直到对象到达一个已在方法说明中定义了的状态。

3.7 演进条件

可线性化性的非阻塞特性说明任何未决调用都有一个正确的响应，但并没有说明如何计算这个响应，例如，考虑图3-1中所示的基于锁的队列。假设队列的初始状态为空。在x入队的过程中间A发生了中断，然后B调用deq()。非阻塞特性要求确保B的deq()调用必须要有一个响应：抛出一个异常或者返回x。然而，在这个实现中B由于无法获得锁，从而使得只要A被延迟则B也将被延迟。

这样的实现称为阻塞，因为一个线程的意外延迟将会阻止其他线程继续推进。而线程的意外延迟在多处理器系统结构中是经常出现的。一个cache缺失可能会导致处理器延迟上百个指令周期，一个页故障可能导致几百万个指令周期的时延。而操作系统抢占则可能导致上亿个指令周期的时延。确切的数据取决于具体的机器和操作系统。

如果能够保证一个方法的每次调用执行都能在有限步内完成，则该方法是无等待的。如果对于一个方法调用存在着关于它的操作步个数的界限，则该方法是有界无等待的。这个界限有可能依赖于线程的个数。例如，第2章中Bakery算法的门廊区就是有界无等待的，其中的界限就是线程的个数。对于一个无等待的方法，若它的性能与处于活动状态的线程个数无关，则称该方法是集居数无关的。如果一个对象的所有方法都是无等待的，则称这个对象是无等待的，在面向对象的语言中，若一个类的所有实例都是无等待的，则称这个类是无等待的。无等待是一种非阻塞的演进条件，意味着一个线程的任意意外延迟（比如说一个线程持有锁）都不会阻塞其他线程的继续执行。

图3-3所示的队列是无等待的。例如，如果在A使x入队的过程中发生中断，然后B调用deq()，那么在这样的情形中，B或者将抛出空异常（A在向数组存入元素x之前发生中断）或者返回x（在A向数组存入元素之后中断）。而图3-1中基于锁的队列不是非阻塞的，因为B将执行无限的操作步请求获得锁。 59

由于无等待特性能够保证所有线程都继续向前推进，因此具有一定的吸引力。然而，无等待算法的效率较低，有时选择一个稍弱的非阻塞特性来解决问题可能是一种更加合适的方案。

如果能够保证一个方法的无限次调用都能够在有限步内完成，则称这个方法是无锁的。显然，任何无等待的方法实现必是无锁的，但反过来不成立。无锁算法允许某些线程可以出现饥饿。事实上，很多可能出现饥饿的情形往往极少发生，因此一个快速的无锁算法比一个较慢的无等待算法更具吸引力。

相关演进条件

无等待和无锁的非阻塞演进条件保证整个计算作为一个整体向前推进，而与系统中线程是如何调度的无关。

第2章已讲述了两种针对阻塞实现的演进条件：无死锁和无饥饿。这两个特性是相关的演进条件：仅当底层平台（即操作系统）提供某种保证时，程序才能推进。原则上，当操作系统能够保证所有的线程最终都可以离开各自的临界区时，无死锁和无饥饿特性是有用的。而实际上，通常在操作系统能够保证每个线程最终都能及时地离开各自的临界区的情形下，这两个特性才是有用的。

如果一个类的方法实现依赖于基于锁的同步，那么最多只能保证相关演进特性。这个结论是否表明要避免使用基于锁的算法呢？答案是未必。若在临界区内的抢占行为很少发生，那么相关阻塞演进条件与相应的非阻塞演进条件的效率相差甚微。若这种抢占经常发生以致引起关注，或者这种基于抢占的时延代价很高，那么采用非阻塞演进条件更为明智。

还有另一种相关的非阻塞演进条件：无干扰。若一个方法调用执行时没有其他的线程也在执行，则称该方法调用的执行过程是隔离的。

定义3.7.1 *若一个方法能在有限步内完成，并且从任一点开始，它的执行都是隔离的，则这个方法是无干扰的。*

与其他的非阻塞演进条件一样，无干扰条件保证并非所有的线程都能被某个或某些其他线程的突发延迟所阻塞。无锁的算法必定是无干扰的，但反过来并不一定成立。

无干扰算法不需要使用锁，但并不能保证多线程并发执行的演进条件，这一点似乎与大
60 多数操作系统调度器所采用的公平方案相违背，在这些系统中，仅当一个线程在其他线程的前面被不公平地调度时，才需要保证演进条件。

然而，在实际中这个问题并不会带来什么影响。无干扰条件不需要中止所有的线程，只是中止那些存在冲突的线程，即调用同一个共享对象的方法的线程。设计无干扰算法的一种简单办法就是引入后退机制：中止检测到冲突的线程，让较早的线程先完成。选择何时后退以及后退多长时间是一个很复杂的问题，将在第7章中进行详细讨论。

为一个并发对象的实现选择演进条件取决于应用的需求和底层平台的特性。从理论上来讲，绝对的无等待和无锁演进条件具有很好的性质，它们可以在任意平台上工作，且能够为音乐、电子游戏以及其他交互应用提供实时保证。而相关的无干扰特性、无死锁特性和无饥饿特性则依赖于底层平台所提供的保证。如果已经提供了这些保证，那么这些相关的特性通常能使实现变得更加简单有效。

3.8 Java存储器模型

Java程序设计语言在读/写共享对象的域时并不保证可线性化性，甚至不保证顺序一致性。为什么呢？其主要的原因就在于若严格地遵循顺序一致性，那么将会使已被广泛采用的编译优化技术变得无效，如寄存器分配、公用子表达式消除、冗余读消除等，因为所有这些优化技术都是通过重排存储器读写次序来实现的。在单线程计算中，这种重新排序对于要被优化的程序来说是不可见的，但在多线程计算中，一个线程可以监视到另一个线程，并观察到混乱无序的执行次序。

Java存储器模型满足松弛存储器模型的基本特性：如果程序的顺序一致执行满足某规则，那么该程序的所有执行在松弛模型中仍然是顺序一致的。本节描述保证Java程序满足顺序一致性的规则，但并不讨论所有的规则，因为那样的话将会非常庞大和复杂。这里着重讲述那些适合于大多数应用的直接规则。

图3-10描述了双重校验上锁，这是一个曾经经常使用的程序设计术语，但由于Java不具备顺序一致性，因而目前很少再被使用。Singleton类管理着一个单一的Singleton对象实例，通过getInstance()方法进行访问，在此方法首次被调用时创建这个实例。为了确保只有一个实例被创建，getInstance()方法必须是同步的，即使多个线程同时观察到instance为*null*时也应保证只有一个实例被创建。然而，一旦实例被创建，则不再需要同步。在图3-10所示的 61
代码中，作为一种优化方式，只有在instance为*null*时才能进入同步块，一旦进入同步块，则在创建实例之前再次进行双重校验，验证instance是否仍然为*null*。

```
1  public static Singleton getInstance() {
2    if (instance == null) {
3      synchronized(Singleton.class) {
4        if (instance == null)
5          instance = new Singleton();
6      }
7    }
8    return instance;
9  }
```

图3-10 双重校验上锁

然而，这种曾经流行的模式并不正确。在第5行，构造函数调用似乎应该在instance域被赋值之前进行，但Java存储器模型允许这两个步骤以任意次序进行，以有效地构造一个对其他程序可见的部分初始化的Singleton对象。

在Java存储器模型中，对象驻留在共享存储器中。每个线程具有一个私有的内存工作区，其中存放着该线程已读/写的域的cache拷贝。在没有显式同步（稍后解释）的情形下，对一个域执行了写操作的线程可能没有将它的更新立刻传送到存储器中；而对一个域执行读操作的线程，当存储器中域的值发生改变时，其相应的内存工作区可能还没有被修改。Java虚拟机不用保持这种cache一致性，但实际上这些cache拷贝往往是一致的，虽然并没有要求这样做。在这种情形下，只能保证一个线程自己的读/写对该线程来说是按照顺序发生的，由一个线程读的任意域值一定是已经被写入那个域的值（即值不是凭空出现的）。

某些语句是同步事件。通常，术语"同步"意味着某种形式的原子性或者互斥。在Java中，它还意味着保持共享存储器与线程的内存工作区之间的一致性。有一些同步事件要引起

线程把cache中的更新写回到共享存储器中，从而使这些更新对其他线程是可见的。而另外一些同步事件则要求线程将其cache中的值设为无效，再重新从存储器中读取域值，从而使其他线程的更新对该线程是可见的。同步事件是可线性化的：它们是全序的，且所有线程都要遵循这种约定次序。下面来看看几种不同的同步事件。

3.8.1 锁和同步块

线程可以通过两种方式达到互斥，或者进入某个**synchronized**块或方法，从而获得一个隐式锁；或者直接获得显式的锁（例如java.util.concurrent.locks包中的ReentrantLock）。两种方法对存储器行为来说具有相同的含义。

如果对一个特定域的所有访问通过同一个锁来保护，那么对这个域的读/写操作是可线性
62 化的。当一个线程释放锁时，内存工作区中被修改的域也被写回到共享存储器中，修改的完成是通过一直持有可由其他线程访问的锁来实现的。若一个线程获得锁，则把自己工作区置为无效，以此保证域值是从共享存储器中重新读取的。所有这些条件确保了由一个单锁保护的对象的读/写操作是可线性化的。

3.8.2 volatile域

volatile（挥发）域是可线性化的。对一个volatile域的读就如同获得一个锁：工作区被置为无效，重新从存储器中读取该挥发域的当前值。对一个volatile域的写类似于释放一个锁：挥发域的值立刻被写回存储器。

尽管volatile域的读/写操作在保持存储器一致性方面具有和申请/释放锁操作相同的效果，但多个读/写操作的执行并不是原子的。例如，假设x是一个volatile变量，如果并发线程可以修改x，则表达式x++并不一定会使x增加1。因此，还是需要某种形式的互斥。volatile变量的一种常用形式是：一个域被多个线程读，而仅被一个线程写。

java.util.concurrent.atomic包中包含有能够提供可线性化存储器的类，如AtomicReference<T>和AtomicInteger。compareAndSet()和set()方法的行为类似于挥发的写，get()方法的行为类似于挥发的读。

3.8.3 final域

被声明为final类型的域一旦初始化后就不能再被修改了。一个对象的final域在其构造函数中被初始化。如果这个构造函数遵循下面描述的一些简单规则，那么任意final域中的正确值不需要同步就对其他线程可见。例如，在图3-11所示的代码中，由于x域被声明成final类型的，所以能够保证调用reader()的线程看到的x值等于3。由于y不是final类型的，因此不保证y值等于4。

如果构造函数被错误地同步，那么看到的final域的值就可能是改变的值。规则很简单：this引用在构造函数返回前决不能被它所释放。

图3-12描述了一个事件驱动系统中错误的构造函数，其中，EventListener类用一个EventSource类来注册自己，构造了一个其他线程可访问的监听对象的引用。因为代码中的注册是构造函数的最后一步，所以看上去似乎是安全的，但实际上却是错误的，其原因在于如果另一个线程在构造函数结束前调用了事件监听器的onEvent()方法，则将无法保证

onEvent()方法看到正确的x值。

```
1  class FinalFieldExample {
2    final int x; int y;
3    static FinalFieldExample f;
4    public FinalFieldExample() {
5      x = 3;
6      y = 4;
7    }
8    static void writer() {
9      f = new FinalFieldExample();
10   }
11   static void reader() {
12     if (f != null) {
13       int i = f.x; int j = f.y;
14     }
15   }
16 }
```

图3-11 有final域的构造函数

```
1  public class EventListener {
2    final int x;
3    public EventListener(EventSource eventSource) {
4      eventSource.registerListener(this); // register with event source ...
5    }
6    public onEvent(Event e) {
7      ... // handle the event
8    }
9  }
```

图3-12 错误的EventListener类

总之，只有当一个域是volatile类型，或者这个域被一个由所有读者和写者都能获得的唯一的锁保护时，对这个域的读/写操作才是可线性化的。

3.9 评析

一个应用应选择什么样的演进条件才是正确的呢？这取决于应用的需求以及运行应用的支撑系统自身所具有的属性。在实际应用中，这是一个"技巧性的问题"，因为不同的方法，甚至那些应用到同一对象的方法，都可能具有不同的演进条件。频繁调用的实时方法应该是无等待的，例如防火墙程序中的表格查询，而像更新表格条目这种不常用的调用则可以使用互斥来实现。正如我们将看到的，在编写应用程序中，采用不同演进条件的方法是很自然的事情。

对于一个应用来说，选择什么样的正确性条件才是适合的呢？这取决于应用的需求。对于一个使用队列来处理打印任务的经负荷打印服务器，只需采用一个满足静态一致性的队列即可，因为文档以什么顺序打印并不重要。一个银行服务器则必须以程序次序（从储蓄中转入100美元到支票中，写一张50美元的支票）执行顾客的请求，所以必须使用顺序一致的队列。而一个股票交易服务器则必须是公平的，所以必须按照不同顾客到达的先后次序来提供服务，这意味着它需要一个可线性化的队列。

下面这个笑话在20世纪20年代的意大利非常流行。根据墨索里尼的观点，完美的公民是充满智慧的、诚实的且奉行法西斯主义的。不幸的是，不存在如此完美的人，这也就解释了为什么日常所见的人要么是充满智慧的、奉行法西斯主义的，但是却不诚实；要么是诚实的、奉行法西斯主义的，但却缺乏智慧；要么是充满智慧的、诚实的，但却不奉行法西斯主义。

对于程序设计人员而言，最理想的情形也许就是拥有可线性化的硬件、可线性化的数据结构及良好的性能。不幸的是，技术总是不完善的，且随着时间的推移，性能良好的硬件甚至都不是顺序一致的，正如这则笑话所说的，具有良好的性能并且是可线性化的数据结构，其实现的可能性是不得而知的。毫无疑问，让这个幻想变成现实存在着许多挑战，本书余下的部分可以当作一个路线图，展示如何实现这个目标。

3.10 本章注释

静态一致性概念最初由James Aspnes、Maurice Herlihy和Nir Shavit[16]提出，而Nir Shavit和Asaph Zemach[143]给出了这一概念的确切定义。Leslie Lamport[90]提出了顺序一致性的概念，Christos Papadimitriou[123]则定义了可顺序化性的规范形式。William Weihl[149]第一个指出了复合性（在他的论文中称之为局部性）的重要性。Maurice Herlihy和Jeannette Wing[69]于1990年提出了可线性化性的概念。Leslie Lamport[93, 94]于1986年提出了原子寄存器的概念。

根据目前所掌握的情况，无等待概念最早在Leslie Lamport的Bakery算法[88]中被非形式化地引入。曾经有过几种不同的无锁概念，最近几年才趋向于现在的定义。无干扰由Maurice Herlihy、Victor Luchangco和Mark Moir[61]提出，相关演进概念由Maurice Herlihy和Nir Shavit[63]提出。

C或C++等程序设计语言并不是针对并发编程的，因此在这些语言中没有定义存储器模型。并发的C或C++程序的实际行为是由底层硬件、编译器和并发库的复杂组合所产生的结果。关于这些问题的详细讨论可见Hans Boehm[21]。本章提到的Java存储器模型是Java语言的第二个存储器模型。Jeremy Manson、Bill Pugh和Sarita Adve[111]给出了当前Java存储器的更为详尽的说明。

双线程队列是一种习惯用法，然而据了解，这一概念以文字方式首次出现是在Leslie
65 Lamport[91]的一篇论文中。

3.11 习题

习题21. 试解释为什么静态一致性是复合的。

习题22. 考虑一个包含有两个寄存器组件的存储器对象。已知如果这两个寄存器是静态一致的，则该存储器也是静态一致的。反过来是否成立？即如果该存储器是静态一致的，每个寄存器是否是静态一致的？请简略证明，或给出一个反例。

习题23. 举出一个例子，其执行是静态一致的但不是顺序一致的。再举一个相反的例子，其执行是顺序一致的但不是静态一致的。

习题24. 对3-13和对3-14中的经历是否是静态一致的、顺序一致的、可线性化的？证明你的结论。

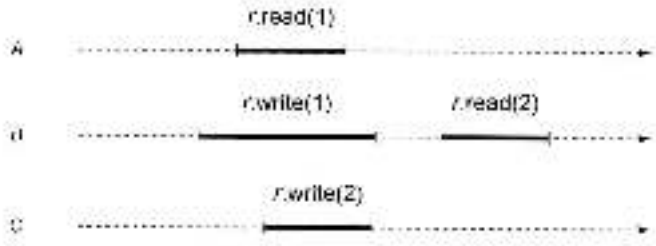


图3-13 习题24的第一个经历

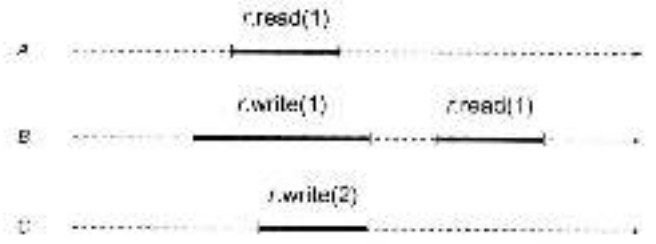


图3-14 习题24的第二个经历

习题25. 如果从可线性化性定义中去掉条件2，所得的性质是否与顺序一致性相同？解释其原因。 66

习题26. 证明定理3.6.1中的“仅当”部分。

习题27. AtomicInteger类（在java.util.concurrent.atomic包中）是一个整型值的容器。它的一个方法为

```
boolean compareAndSet(int expect, int update)
```

该方法将对象的当前值与预期值相比较。若值相等，则用update原子地替换对象的值并返回*true*。否则，不改变对象的值并返回*false*。这个类还提供了

```
int get()
```

该方法返回对象的实际值。

考虑图3-15所示的FIFO队列实现。该队列使用一个数组items来存放元素，为简单起见，假设该数组是无界的。队列具有两个AtomicInteger域：tail域是下一个将被移出元素的数组槽的索引号，head域是下一个要被存入元素的数组槽的索引号。举一个例子说明这个实现是不可线性化的。

习题28. 考虑图3-16所示的类，根据你所了解的有关Java存储器模型的知识，reader方法中会存在被0除的情形吗？

习题29. 下述性质是否等价于对象x是无锁的？

对于x的任意一个无限经历H，所有在H中执行了无限个操作步的线程都要完成无限个方法调用。

习题30. 下述性质是否等价于对象x是无锁的？

对于x的任意一个无限经历H，都有无限个方法调用被完成。

习题31. 考虑下面这种很少被使用的关于方法m的实现。在所有的经历中，一个线程第i次调用m，则该调用在第2^i步后返回。该方法是无等待的吗？是有界无等待的吗？或者都不是？

习题32. 考虑一种队列实现（图3-17），其中enq()方法没有可线性化点。

```
1  class IQueue<T> {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicInteger tail = new AtomicInteger(0);
4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
5    public void enq(T x) {
6      int slot;
7      do {
8        slot = tail.get();
9      } while (! tail.compareAndSet(slot, slot+1));
10     items[slot] = x;
11   }
12   public T deq() throws EmptyException {
13     T value;
14     int slot;
15     do {
16       slot = head.get();
17       value = items[slot];
18       if (value == null)
19         throw new EmptyException();
20     } while (! head.compareAndSet(slot, slot+1));
21     return value;
22   }
23 }
```

图3-15 IQueue实现

```
1  class VolatileExample {
2    int x = 0;
3    volatile boolean v = false;
4    public void writer() {
5      x = 42;
6      v = true;
7    }
8    public void reader() {
9      if (v == true) {
10       int y = 100/x;
11     }
12   }
13 }
```

图3-16 习题28的volatile域

该队列使用一个items数组存放元素，为了方便起见假设数组是无界的。tail域是一个AtomicInteger，初始值为0。enq()方法通过将tail增加1来保留一个槽，然后在那个地址存入元素。注意，这两个步骤不是原子的：在tail被增加1以后和在元素被存入数组中之前存在一个时间间隔。

deq()方法读tail的值，然后以升序次序从槽0到tail遍历数组。对每一个槽，用空值与当前内容交换，并返回第一个非空元素。若所有的槽都为空，重新开始这个过程。

给出一个执行实例，说明enq()的可线性化点不可能在第15行出现。

67
↓
68

提示：给出一个执行，其中两个enq()调用没有按照它们执行第15行代码的次序被线性化。

给出另一个例子，说明enq()的可线性化点不可能在第16行出现。

由于这是enq()中仅有的两个存储器访问操作，可以推知enq()方法没有单个可线性化点，这是否意味着enq()是不可线性化的呢？

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8          CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

图3-17 Herlihy/Wing队列

习题33. 证明顺序一致性是非阻塞的。

第4章

The Art of Multiprocessor Programming (Revised First Edition)

共享存储器基础

顺序计算基础是由Alan Turing和Alonzo Church在20世纪30年代所奠定的，他们各自独立地提出了丘奇-图灵论题：能被计算的所有事情，都能由图灵机（或等价的兰奇演算）计算，任何由图灵机不能解的问题（比如判定一个程序对于任意的输入是否会停机），对实际的计算设备也是不可解的。图灵论题是一个论题而不是定理，因为"什么是可计算的"这一概念不可能用精确的、数学上严谨的方式来定义。尽管如此，几乎所有人都相信图灵论题。

本章讲述共享存储器并发计算的基本概念。一个共享存储器的计算由多个线程构成，每个线程自身是一个顺序程序。它们相互之间通过调用驻留在共享存储器中的对象进行通信，线程是异步的，它们各自以不同的速度执行，且在任意时刻可以停止一个不可预测的时间间隔。这种异步概念反映了现代多处理器系统结构的实际情形，线程时延是不可预测的，从微秒级（cache缺失）到毫秒（页故障）、秒级（调度中断）。

顺序计算性理论的发展分可为多个阶段。最初是有穷自动机，随后是下推自动机，最后以图灵机到达顶峰。我们也同样分阶段地研究并发计算模型。本章从最简单的共享存储器计算模式开始：并发线程对共享存储单元执行简单的读/写操作。由于历史上的原因，共享存储单元也可称为寄存器，首先讲述简单的寄存器，再进一步说明如何利用这些简单寄存器来构造一系列更为复杂的寄存器。

传统的顺序计算性理论（大多数）不考虑效率因素：要说明一个问题是可计算的，只需说明该问题能用图灵机来解决就足够了。很少考虑如何去提高图灵机的效率，因为图灵机并
71 不是一种实际的计算工具。同样，在并发计算理论的研究中，我们无需关心如何构造高效的寄存器，而是着重理解这样的构造是否存在以及它们是如何工作的。我们没有将它们作为实际的计算模型，本章内容侧重于易于理解但效率不高的寄存器构造，而不考虑结构复杂但效率较高的构造。在所讲述的一些构造中，特别使用了时间戳（计数器值）来区分新值与旧值。

时间戳的问题在于它们是无界增长的，最终会超出任意固定大小的变量。有界的解决方案（比如第2章2.7节中的例子）更能满足实际的需求（可论证的），希望读者通过本章注释提供的参考资料深入研究。本章关注简单的无界结构，这样可以避免读者的注意力被更多技巧性的内容所分散，以便更好地掌握并发程序设计的基本原理。

4.1 寄存器空间

在硬件层，线程是通过读/写共享存储器进行通信的，理解线程间相互通信的一种办法就是对硬件原语进行抽象，把通信看作是通过读/写并发共享对象实现的。第3章给出了共享对象的详细描述。现在，回顾对象设计中的两个关键特性：安全性和活性，前者由一致性条件定义，后者由演进性条件定义。

读/写寄存器（或寄存器）是一个将值封装起来的对象，且只能通过read()调用读取这个值，通过write()调用来修改该值（在实际的系统中称这些方法调用为加载/存储）。图4-1描述了所有寄存器都具有的接口Register<T>。值的类型T可以是布尔型、整型或者是对另一个

对象的引用，具有Register<布尔型>接口的寄存器称为布尔寄存器（有时用1和0代表*true*和*false*）。具有Register<整型>接口且值在范围M内的寄存器称为M-值寄存器。本章不再讨论其他类型的寄存器，而是将引用看作整数，从而使得任何针对整型寄存器的算法都能够用于具有引用类型寄存器的实现中。

```
1  public interface Register<T> {
2    T read();
3    void write(T v);
4  }
```

图4-1　Register<T>接口

如果方法调用之间没有重叠，那么寄存器的实现应呈现出如图4-2所示的行为。然而在多处理器中，我们希望方法调用在任何时候都是重叠的，因此需要规范化地说明什么是并发的方法调用。

```
1  public class SequentialRegister<T> implements Register<T> {
2    private T value;
3    public T read() {
4      return value;
5    }
6    public void write(T v) {
7      value = v;
8    }
9  }
```

图4-2　SequentialRegister类

一种说明方式就是依靠互斥来定义并发方法调用：使用互斥锁来保护每个寄存器，要求每一次read()和write()调用都必须首先获得这个锁。然而不幸的是，在多处理器系统结构中，不能使用互斥来实现共享对象的并发调用。首先，第2章讲述了如何利用寄存器实现互斥，因此，再通过互斥来实现寄存器已几乎没有任何意义。其次，第3章已指出，若通过互斥来实现寄存器，即使这种实现是无死锁或无饥饿的，计算的演进仍然依赖于操作系统的调度器，要通过调度器来保证线程不会在临界区内阻塞。由于我们要用共享对象来检验并发计算的基础构造块，所以，让一个单独的实体来提供关键的演进特性将不会有任何意义。

下面介绍另外一种说明方式。回顾前面所讲过的内容，如果对象的每个方法调用都能在有限步内完成，并且每个方法的调用执行都与它和其他并发的方法调用之间的交互无关，则称这个对象的实现是无等待的。无等待条件看上去简单自然，但却具有很深的含义，特别是它排除了任何形式的互斥，能够保证独立地演进，也就是说，不依赖于操作系统的调度器。因此，通常要求寄存器的实现应是无等待的。[⊖]

所谓原子寄存器就是图4-2中顺序寄存器类的一种可线性化的实现。非形式化地讲，原子寄存器的行为如同我们所期望的一样：每一个读操作都返回“最后”一次写入的值。各个线程通过读/写原子寄存器进行通信的模型显然具有吸引力，该模型很久以来一直作为并发计算的标准模型。

规范说明读者和写者的个数也是非常重要的。显然，支持单读者-单写者的寄存器实现比

⊖　一个无等待的实现也是无锁的。

支持多读者-多写者的寄存器实现要容易一些。为简单起见，下面用SRSW表示“单读者-单
73 写者”，用MRSW表示“多读者-单写者”，用MRMW表示“多读者-多写者”。

本章主要研究下述基本问题：

是否我们定义的功能最强的寄存器所实现的数据结构是否也能通过较弱的寄存器来实现？

回顾第1章中所讲的，线程之间任何一种可用的通信方式必定是持续的：消息发送的持续时间比发送者的主动参与时间更长。这种持续同步的最弱形式就是（可论证）要能在共享存储器中设置一个持续位，而同步的最弱形式则是什么都没有（不可论证）：如果设置一个位的动作与读这个位的动作间没有重叠，那么读到的值与写入的值是一致的。否则，重叠的读与写可以返回任意值。

不同类型的寄存器能够作出不同的保障，从而使寄存器具有更强或更弱的能力。例如，寄存器可以在它们所封装值的范围上有所差异（布尔寄存器和M-值寄存器），它们能支持的读者和写者的个数有可能不同，它们所提供的一致性程度也可能不同。

一个单写者-多读者寄存器的实现是安全的，如果

- 与write()调用不相重叠的read()调用，其返回值是最近一次write()调用所写入的值，
- 如果read()调用与write()调用相重叠，则read()调用可以返回该寄存器所允许范围内的任意值（例如，对于M-值寄存器是从0到$M-1$中的任意值）。

注意，“安全”一词是历史的偶然：它们提供的保证其实是很弱的，“安全”的寄存器实际上非常不安全。

考虑图4-3所示的经历。如果寄存器是安全的，那么三个读调用的行为应该如下：

- R^1返回最近一次写入的值0，
- R^2和R^3与$W(1)$是并发执行的，所以可以返回寄存器允许范围内的任意值。



图4-3 一个单读者-单写者寄存器的执行过程：R^i表示第i次读，$W(v)$表示写值v。时间自左向右。无论寄存器是否是安全的、规则的或者原子的，R^1必定返回最近一次写入的值0。若寄存器是安全的，则由于R^2和R^3与$W(1)$是并发的，它们可以返回寄存器允许范围内的任意值。若寄存器是规则的，R^2和R^3可能都返回0或者1。若寄存器是
74 原子的，那么当R^2返回1时R^3也必然返回1，R^2返回0时R^3可能返回0也可能返回1

下面定义一种介于安全寄存器和原子寄存器之间的一致性标准，以便于说明问题，规则寄存器是一种多读者-单写者寄存器，其中写操作不是原子的。当write()调用正在执行的时候，若旧值还没有完全被新值所替代，那么读到的值有可能在旧值和新值之间“闪动”。更确切地说：

- 规则寄存器是安全的，因此任意一个不与write()调用相重叠的read()调用都返回最近一次被写入的值，
- 假设一个read()调用与一个或多个write()调用重叠，令v^0是最后一次write()调用所写

入的值，v^1，…，v^k为重叠的write()调用所写入的值的序列，那么，read()调用有可能返回任意的值v^i（i从0到k）。

对于图4-3所示的执行过程，一个单读者规则寄存器可能会有如下的行为：

- R^1返回旧值0。
- R^1和R^1可能都返回旧值0，或者新值1。

规则寄存器必定是静态一致的（第3章），但反过来不成立。安全和规则寄存器只允许一个单写者线程使用。注意，规则寄存器是静态一致的单写者顺序寄存器。

对于单读者/单写者原子寄存器，图4-3所示的执行过程可能产生如下的结果：

- R^1返回旧值0。
- 若R^2返回1，则R^1也返回1。
- 若R^2返回0，则R^1可能返回0，也可能返回1。

图4-4是一个各种寄存器的二维示意图：其中，一个维定义了寄存器的大小，另一个维定义了读者和写者的个数，而第三个维定义了寄存器的一致性特性。该示意图不能完全照字面来看：其中有多种联系是没有意义的，例如多写者安全寄存器。

为了便于分析规则和原子寄存器的实现算法，可以直接使用对象经历来重新进行定义。从现在开始只考虑这样的经历，各个read()调用返回的值必定是被某个write()调用写入的值（规则和原子寄存器不允许读操作虚构返回值）。假设读或写的值都是唯一的。[⊖]

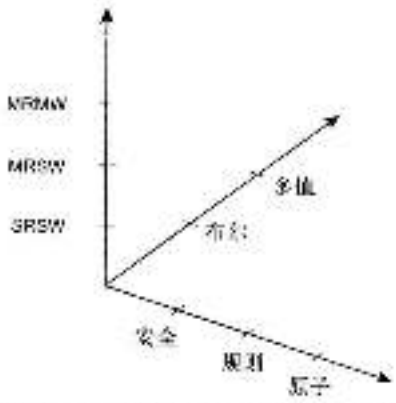


图4-4 基于读/写寄存器实现的三维空间

回顾前面所讲内容，对象经历是由调用事件和响应事件所组成的一个序列，当一个线程调用某个方法时发生调用事件，当该调用返回时发生与之匹配的响应事件。方法调用（简称调用）是指匹配的调用事件和响应事件之间的时间间隔。对任意一个经历，必定存在着一个关于方法调用的偏序关系"→"，可按如下方式来定义：对于方法调用m_0和m_1，如果m_0的响应事件先于m_1的调用事件，则$m_0 \rightarrow m_1$。（完整的定义见第3章。）

任意一种寄存器实现（无论是安全的、规则的还是原子的）都可以在write()调用上定义一个全序关系，称这种关系为写次序，用来表示寄存器中写操作"生效"的次序。对于安全和规则寄存器，写次序并不重要，因为它们一次只允许一个写者。而在原子寄存器中，所有的方法调用具有一个可线性化的次序，那么，我们可以使用这个次序来索引写调用：写调用W^0为第一个，W^1为第二个，以此类推。注意，对于SRSW或MRSW安全寄存器或规则寄存器来说，写次序与先于次序是完全一致的。

用R^i表示返回值为v^i的读调用，其中v^i是由W^i所写入的唯一值。注意，一个经历中只包含一个W^i调用，但可以包含多个R^i调用。

⊖ 如果值本身不是唯一的，我们可以利用一种标准方法，即为每个值附加一个算法可见的时间值，该值仅在辨别多个值的过程中有用。

可以证明下面的条件能够准确地描述规则寄存器。首先，读调用不会返回将来的值：

$$\text{绝不可能存在} R^i \to W^j \tag{4.1.1}$$

其次，读调用不会返回更远的过去值，即先于最近一次写的且没有重叠的值：

76 $$\text{对于某个值 } j\text{，绝不会存在 } W^{j'} \to W^{i'} \to R^i \tag{4.1.2}$$

为了证明一种寄存器的实现是规则的，必须证明该寄存器的经历满足条件（4.1.1）和（4.1.2）。

原子寄存器还要满足另一个附加条件：

$$\text{若 } R^i \to R^j\text{，则 } i \le j \tag{4.1.3}$$

这个条件表明较早的读操作的返回值决不能比较晚的读操作的返回值更晚。规则寄存器不需要满足条件（4.1.3）。为了证明一种寄存器的实现是原子的，首先要定义一个写次序，然后证明其经历满足条件（4.1.1）～（4.1.3）。

4.2 寄存器构造

下面讨论如何利用简单的单读者/单写者安全布尔寄存器来实现功能更加强大的寄存器。在这些构造中，我们都假定所有读/写寄存器的类型是等价的，至少从可计算性的角度来说是这样的。下面考虑通过较弱的寄存器所实现的具有较强功能的寄存器的构造序列。

图4-5列出了这种构造的序列。

基类	实现的类	节
SRSW安全寄存器	MRSW安全寄存器	4.2.1
MRSW安全布尔寄存器	MRMW规则布尔寄存器	4.2.2
MRSW规则布尔寄存器	MRSW规则寄存器	4.2.3
MRSW规则寄存器	SRSW原子寄存器	4.2.4
SRSW原子寄存器	MRSW原子寄存器	4.2.5
MRSW原子寄存器	MRMW原子寄存器	4.2.6
MRSW原子寄存器	原子快照	4.3

图4-5 寄存器的构造序列

我们来解释一下表中的最后一行，用原子寄存器（也是安全寄存器）实现的原子快照：这是一种由不同的线程所写的MRSW寄存器数组，能被任何线程原子地读。上述某些构造的功能要比实现派生序列所需的功能更为强大（例如，不需要为规则和安全寄存器提供多读者特性，就可以实现SRSW原子寄存器的衍生）。之所以把它们列出来，是因为这个构造序列能提供一些有价值的帮助。

书中的代码示例遵循下面一些规约。在描述实现特定类型的寄存器算法时（例如，一个安全的MRSW布尔寄存器），往往用如下的形式来表示：

```
class SafeMRSWBooleanRegister implements Register<Boolean>
{
...
}
```

77

虽然这样会使得要被实现的寄存器类的特性非常清晰，但若用它们来实现其他的类则显得非

常领域。因此，当描述类的实现时，我们使用下面的约定来说明一个具体的域是否是安全的、规则的或者原子的。以一个mumble域为例，若它是安全的，则称为s_mumble；若它是规则的，则称为r_mumble；若是原子的，则称为a_mumble。该域的其他方面，如它的类型或它是否支持多读者或多写者，都将在代码中以注释的形式给出，并且从上下文来看应该是语义清晰的。

4.2.1　MRSW安全寄存器

图4-6描述了如何使用安全的SRSW寄存器构造安全的MRSW寄存器。

```
1  public class SafeBooleanMRSWRegister implements Register<Boolean> {
2    boolean[] s_table; // array of safe SRSW registers
3    public SafeBooleanMRSWRegister(int capacity) {
4      s_table = new boolean[capacity];
5    }
6    public Boolean read() {
7      return s_table[ThreadID.get()];
8    }
9    public void write(Boolean x) {
10     for (int i = 0; i < s_table.length; i++)
11       s_table[i] = x;
12   }
13 }
```

图4-6　SafeBooleanMRSWRegister类：一种安全的布尔MRSW寄存器

引理4.2.1　图4-6中的构造是一种MRSW安全寄存器。

证明　若线程A的read()调用不与任何write()调用重叠，那么该read()调用返回最近一次写入的s_table[A]的值。对于重叠的方法调用，由于寄存器是安全的，读者可以返回任意值。　□

4.2.2　MRSW规则布尔寄存器

图4-7描述了如何使用MRSW安全布尔寄存器来构造MRSW规则布尔寄存器。对于布尔寄

```
1  public class RegBooleanMRSWRegister implements Register<Boolean> {
2    ThreadLocal<Boolean> last;
3    boolean s_value; // safe MRSW register
4    RegBooleanMRSWRegister(int capacity) {
5      last = new ThreadLocal<Boolean>() {
6        protected Boolean initialValue() { return false; };
7      };
8    }
9    public void write(Boolean x) {
10     if (x != last.get()) {
11       last.set(x);
12       s_value =x;
13     }
14   }
15   public Boolean read() {
16     return s_value;
17   }
18 }
```

图4-7　RegBooleanMRSWRegister类：用MRSW安全布尔寄存器所构造的MRSW规则布尔寄存器

存器而言，只有当要写入的新值x与旧值一样时，安全的布尔寄存器与规则的布尔寄存器之间才会有区别。规则寄存器只能返回x，而安全寄存器则可以返回任一个布尔值。因此，只要确保写入的新值与原先写入的值不同时才允许修改，就可以解决这个问题。

78 引理4.2.2 图4-7中的构造是一种MRSW规则布尔寄存器。

证明 如果一个read()调用不与任何write()调用重叠，则返回最近一次写入的值。若调用间有重叠，则要考虑以下两种情况。

- 如果要被写入的值与最后一次写入的值相同，那么写者不对该安全寄存器写，从而确保读者读到正确的值。
- 如果要被写入的值与最后一次写入的值不同，则由于寄存器是布尔的，那么值必定为*true*或*false*。并发的读者将返回寄存器取值范围内的某个值，或者是*true*或者是*false*，且都是正确的。 □

4.2.3 *M*-值MRSW规则寄存器

如果不考虑效率会变得非常低，那么从布尔寄存器转变到*M*-值寄存器是很容易的：使用一元符号表示值。在图4-8的实现中，将*M*-值寄存器看作是一个由*M*个布尔寄存器组成的数组。寄存器的初始值为0，通过将第0位设置为*true*来表示。若一个写方法要写值*x*，则在数组单元*x*处写入*true*，然后按照数组索引的降序次序将所有比*x*低的单元都设为*false*。面对于读方法，则按照索引的升序次序读取数组单元的值，直到在单元*i*第一次读到*true*值，然后返回*i*。图4-9描述了一个8-位寄存器。

```
1  public class RegMRSWRegister implements Register<Byte> {
2    private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
3    boolean[] r_bit = new boolean[RANGE]; // regular boolean MRSW
4    public RegMRSWRegister(int capacity) {
5      for (int i = 1; i < r_bit.length; i++)
6        r_bit[i] = false;
7      r_bit[0] = true;
8    }
9    public void write(Byte x) {
10     r_bit[x] = true;
11     for (int i = x - 1; i >= 0; i--)
12       r_bit[i] = false;
13   }
14   public Byte read() {
15     for (int i = 0; i < RANGE; i++)
16       if (r_bit[i]) {
17         return i;
18       }
19     return -1; // Impossible
20   }
21 }
```

图4-8 RegMRSWRegister类：一个MRSW的*M*-值规则寄存器

引理4.2.3 在图4-8所示的构造中，read()调用总能返回一个值，该值对应于0到*M*−1之间由某个write()调用所设置的一个位。

证明 下面的性质总不变的：若一个读者正在读r_bit[i]，则必定有某个索引号大于等于
79 *i*的位被一个write()调用设置为*true*。

当寄存器初始化时没有读者，构造函数（此构造函数的调用被看作是一个write(0)调用）

将r_bit[0]设为*true*。现假设有一个读者正在读r_bit[j]，并且r_bit[k]为*true*（$k>j$）。那么，

- 若读者从*j*前进到*j*+1，则r_bit[*j*]为*false*，所以$k>j$（即一个大于或等于*j*+1的位的值为*true*）。
- 仅当写者将一个更高的位r_bit[ℓ]（$\ell>k$）设为*true*时才清除r_bit[*k*]的值。

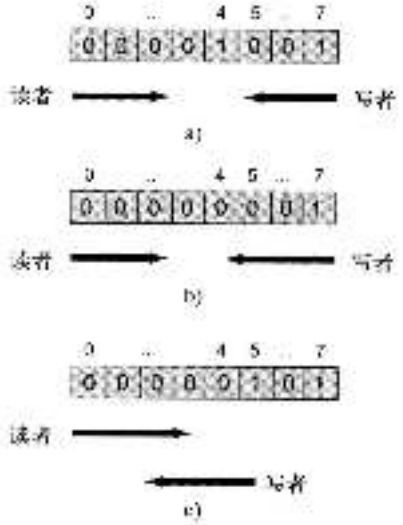


图4-9　RegMRSWRegister类：8-值的MRSW规则寄存器的执行。1和0分别代表*true*和*false*。在a中，写之前的值为4，线程*R*无法读到线程*W*写入的值7，因为在线程*W*将*false*值重新写入数组项4之前，线程*R*已到达该位置。在b中，在读数组项4之前该位置已被*W*重写，所以读操作返回7。在c中，*W*准备写入5，由于*W*在数组项5被读之前已对其写入，因此即使数组项7的值为*true*，读者依然返回5

引理4.2.4　图4-8中的构造是一个*M*-值的MRSW规则寄存器。

证明　对任意的读操作，令*x*是由最近一次与之不相重叠的write()方法所写入的值。当write()完成时，a_bit[*x*]已被设置为*true*，且对所有的$i<x$，a_bit[*i*]都为*false*。由引理4.2.3可知，如果读者返回一个不等于*x*的值，则它已观察到某个a_bit[*j*]（$j\neq x$）为true，这个位必定是由一个并发的写操作所写的，从而证明了条件(4.1.1)和(4.1.2)。　□

4.2.4　SRSW原子寄存器

首先说明如何使用SRSW规则寄存器来构造SRSW原子寄存器。（使用无界时间戳来构造。）

规则寄存器满足条件（4.1.1）和（4.1.2），原子寄存器还需要满足条件（4.1.3）。由于SRSW规则寄存器不存在并发的读，所以违背条件（4.1.3）的唯一情形就是：两个读与同一个写重叠且读到次序颠倒的值，第一个读返回*v*′，而第二个读返回*v*，其中$j<i$。

图4-10描述了一个关于值的类，其中每个值都附有一个时间戳标签。图4-11所示的AtomicSRSWRegister实现使用这些标签对写调用进行排序，从而使并发的读调用能够正确地确定次序。每个读必须记住它所读过的最后一个（最高的时间戳）时间戳/值对，为以后的读所使用。如果一个较晚的读随后读到一个较早的值（时间戳较低），那么它将忽略这个值并使用所记住的最后值。同样，写者也应该记住它所写的最后一个时间戳，并用一个更加晚的时间戳（比前一个时间戳大1）来标记每个要被写入的新值。

```
1  public class StampedValue<T> {
2    public long stamp;
3    public T    value;
4    // Initial value with zero timestamp
5    public StampedValue(T init) {
6      stamp = 0;
7      value = init;
8    }
9    // later values with timestamp provided
10   public StampedValue(long stamp, T value) {
11     stamp = stamp;
12     value = value;
13   }
14   public static StampedValue max(StampedValue x, StampedValue y) {
15     if (x.stamp > y.stamp) {
16       return x;
17     } else {
18       return y;
19     }
20   }
21   public static StampedValue MIN_VALUE =
22     new StampedValue(null);
23 }
```

图4-10 StampedValue<T>类：允许一个时间戳和一个值一起被读或写

该算法要求系统具有能将一个值和时间戳作为独立的单元进行读/写的能力。在类似C这样的语言中，可以将值和时间戳一起看作是不需解释的位，通过移位和逻辑屏蔽将这两个值打包/解包到一个或多个字中。在Java中，可以很容易地创建一个具有时间戳值对的StampedValue<T>结构，并且在寄存器中存放这个结构的引用。

80
81

引理4.2.5 图4-11中的构造是一种SRSW原子寄存器。

```
1  public class AtomicSRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    ThreadLocal<StampedValue<T>> lastRead;
4    StampedValue<T> r_value;            // regular SRSW timestamp-value pair
5    public AtomicSRSWRegister(T init) {
6      r_value = new StampedValue<T>(init);
7      lastStamp = new ThreadLocal<Long>() {
8        protected Long initialValue() { return 0; };
9      };
10     lastRead = new ThreadLocal<StampedValue<T>>() {
11       protected StampedValue<T> initialValue() { return r_value; };
12     };
13   }
14   public T read() {
15     StampedValue<T> value = r_value;
16     StampedValue<T> last = lastRead.get();
17     StampedValue<T> result = StampedValue.max(value, last);
18     lastRead.set(result);
19     return result.value;
20   }
21   public void write(T v) {
22     long stamp = lastStamp.get() + 1;
23     r_value = new StampedValue(stamp, v);
24     lastStamp.set(stamp);
25   }
26 }
```

图4-11 AtomicSRSWRegister类：使用SRSW规则寄存器构造的SRSW原子寄存器

证明　因为该寄存器是规则的，所以它满足条件（4.1.1）和（4.1.2）。又因为按照时间戳写操作是全局的，且当一个读操作要返回一个值时，由于较早写入的值具有较低的时间戳，所以较晚的读操作不可能读到较早写的值，所以，此算法也满足条件（4.1.3）。 □

4.2.5　MRSW原子寄存器

为了理解如何使用SRSW原子寄存器来构造MRSW原子寄存器，先来考虑一种直接利用4.2.1节中将SRSW寄存器改造为 MRSW安全寄存器的算法进行构造的简单办法。让组成数组a_table[0...$n-1$]的SRSW寄存器为原子的而不是安全的，并且所有其他的调用都遵循：写者以索引号递增的次序写数组单元，然后，每个读者进行读并返回相应的数组项。但这样所得的结果并不是一个多读者原子寄存器。因为每个读者都是从一个原子寄存器中读，所以条件（4.1.3）对单读者情形是成立的，但对于多个不同的读者，该条件并不成立。例如，考虑这样的一个写操作，它首先设置SRSW寄存器的第一项a_table[0]，然后在写存储单元a_table[1...$n-1$]之前被延迟。随后线程0的读将会返回正确的新值，但紧接着线程0之后的线程1将会读取并返回更早的值，因为写者还未修改a_table[1..$n-1$]。对于这个问题，我们可以通过让 82
较早的读线程将它们所读到的值告诉给较晚的线程来帮助后者解决。

具体的实现见图4-12。n个线程共享一个由具有时间戳的值所组成的$n*n$数组a_table

```
1  public class AtomicMRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    private StampedValue<T>[][] a_table; // each entry is SRSW atomic
4    public AtomicMRSWRegister(T init, int readers) {
5      lastStamp = new ThreadLocal<Long>() {
6        protected long initialValue() { return 0; }
7      };
8      a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
9      StampedValue<T> value = new StampedValue<T>(init);
10     for (int i = 0; i < readers; i++) {
11       for (int j = 0; j < readers; j++) {
12         a_table[i][j] = value;
13       }
14     }
15   }
16   public T read() {
17     int me = ThreadID.get();
18     StampedValue<T> value = a_table[me][me];
19     for (int i = 0; i < a_table.length; i++) {
20       value = StampedValue.max(value, a_table[i][me]);
21     }
22     for (int i = 0; i < a_table.length; i++) {
23       if (i == me) continue;
24       a_table[me][i] = value;
25     }
26     return value;
27   }
28   public void write(T v) {
29     long stamp = lastStamp.get() + 1;
30     lastStamp.set(stamp);
31     StampedValue<T> value = new StampedValue<T>(stamp, v);
32     for (int i = 0; i < a_table.length; i++) {
33       a_table[i][i] = value;
34     }
35   }
36 }
```

图4-12　AtomicMRSWRegister类：使用SRSW原子寄存器构造MRSW原子寄存器

[0..n-1] [0..n-1]。正如4.2.4节所讲述的，使用具有时间戳的值可以使较早的读告诉较晚的读，它所读到的值哪个是最新的。对角线上的单元，即a_table[i][i]（对所有i），对应于前面已讨论的那种简单但无效的寄存器构造。写者将一个新值及其时间戳一个接一个地写入对角线单元，同时让时间戳随着write()的每次调用而不断增加。每个读者A则像前面的算法那样首先读取a_table[A][A]。然后，使用余下的SRSW单元a_table[A][B]（$A \neq B$）来完成读者A和B之间的通信。在读取a_table[A][A]以后，读者A通过扫描其对应的列（对所有的B扫描a_table[B][A]）并查看是否包含一个更晚的值（时间戳更高），来检查是否有其他的读者已读了一个更晚的值。随后，读者A将这个值写入其对应行的单元中（a_table[A][B]，对所有的B），从而让较晚的读者能够知道它所读的最晚的值是什么。这样的话，在A的读完成后，B的每个较晚的读都可以看到A最后读的值（因为它读了a_table[A][B]）。图4-13给出了该算法的一个执行实例。

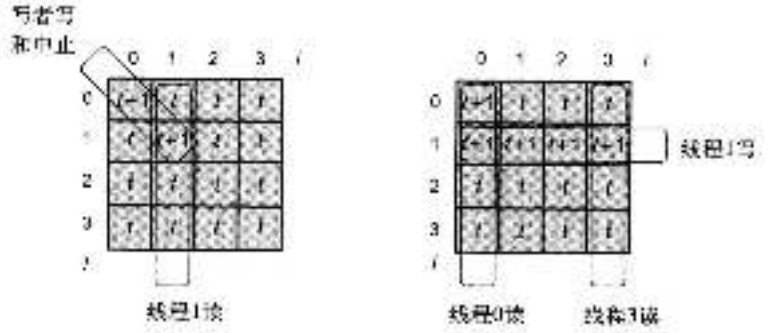


图4-13 MRSW原子寄存器的执行。每个读者具有一个0～4的索引号，并用索引号代表相应的线程。写者把一个具有时间戳t+1的新值写入单元a_table[0][0]和a_table[1][1]，然后中止。随后，线程1对所有的i读它所对应的列元素a_table[i][1]，对所有的i写它所对应的行元素a_table[1][i]，并返回时间戳为t+1的新值。线程0和3在线程1的读完成之后进行读。线程0读a_table[0][1]，其时间戳为t+1。线程3无法读时间戳为t+1的新值。因为写者此时也要写a_table[3][3]。但是，线程3读了a_table[3][1]并返回时间戳为t+1且由更早的线程1读过的正确值

引理4.2.6 图4-12的构造是一种MRSW原子寄存器。

证明 首先，读者不可能返回将来的值，因此条件（4.1.1）成立。其次，由构造可知，write()调用是按照严格递增的次序与时间戳的，该算法的关键之处在于，在所有行（列）上的最大时间戳按行（列）来看也是严格递增的。这样的话，如果A所写的值v的时间戳为t，则B的任意read()调用（这里A的调用完全先于B的调用）所读到的最大时间戳必定大于等于t，从而满足条件（4.1.2）。最后，如前面所述，如果A的读调用完全先于B的读调用，那么A将会把一个时间戳为t的值写入B的列中，这样B将选择一个时间戳大于等于t的值，所以条件（4.1.3）成立。 □

83
84

4.2.6 MRMW原子寄存器

下面讲述如何使用MRSW原子寄存器数组（每个元素对应一个线程）构造MRMW原子寄存器。

如果A要写寄存器，那么它首先读所有的数组元素，选出一个比它所读到的时间戳都要大的时间戳，再将一个附有时间戳的值写入数组元素A。如果一个线程要读寄存器，则首先读所有的数组元素，最后返回具有最高时间戳的那个元素值。这与2.6节中Bakery算法所使用的时间戳算法完全相同，也是采用有利于索引号较小的线程的解决办法；换句话说，使用时间戳和线程id对的字典次序。

引理4.2.7 图4-14中的构造是一种MRMW原子寄存器。

```
1  public class AtomicMRMWRegister<T> implements Register<T>{
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public AtomicMRMWRegister(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      StampedValue<T> value = new StampedValue<T>(init);
6      for (int j = 0; j < a_table.length; j++) {
7        a_table[j] = value;
8      }
9    }
10   public void write(T value) {
11     int me = ThreadID.get();
12     StampedValue<T> max = StampedValue.MIN_VALUE;
13     for (int i = 0; i < a_table.length; i++) {
14       max = StampedValue.max(max, a_table[i]);
15     }
16     a_table[me] = new StampedValue(max.stamp + 1, value);
17   }
18   public T read() {
19     StampedValue<T> max = StampedValue.MIN_VALUE;
20     for (int i = 0; i < a_table.length; i++) {
21       max = StampedValue.max(max, a_table[i]);
22     }
23     return max.value;
24   }
25 }
```

图4-14 MRMW原子寄存器

证明 按照（时间戳，线程id）对的字典次序对所有的write()调用进行排序，使得如果$t_A < t_B$或者$t_A = t_B$且$A < B$，则A（时间戳为t_A）的write()调用先于B（时间戳为t_B）的write()调用。这种字典次序与“ $\rightarrow$ ”是相一致的，对此断言的证明留作习题。与前面一样，我们用写次序W^0，W^1，…来引用每个write()调用。

显然，当一个read()调用完成之后，它无法读a_table[]中的写入值，并且任意一个完全在这个读之后的write()调用，其时间戳都高于该读调用完成前的任何write()调用的时间戳，即条件（4.1.1）成立。 R5

考虑条件（4.1.2），该条件不允许跳过先前最近的write()。假设A的write()调用先于B的write()调用，B的write()调用又先于C的write()调用。若$A=B$，则较晚的写者写了a_table[A]并且read()不会返回较早的写入值。若$A \neq B$，则由于A的时间戳小于B的时间戳，那么对于任意观察到这两个操作的C，都返回B的值（或具有更高时间戳的值），所以构造满足条件（4.1.2）。

最后，考虑条件（4.1.3），该条件不允许读次序违背写次序。假定A的所有read()调用都完全先于B的某一个read()调用以及C的所有write()调用，而C的write()调用在写入次序上

又先于D的write()调用。我们需要证明，若A返回D的值，则B不能返回C的值。如果$t_C<t_D$，那么若A从a_table[D]中读到时间戳t_D，则B将从a_table[D]中读到大于等于t_D的时间戳，且返回值的时间戳不等于t_C。如果$t_C=t_D$，即写操作是并发的，则按照写次序有$C<D$，所以若A从a_table[D]中读到时间戳t_D，则B从a_table[D]中也一定读到t_D，且返回具有时间戳t_D（或更高）的值，即使它从a_table[C]中读到t_C也是如此。

上述一系列构造说明可以使用SRSW安全布尔寄存器构造出MRMW无等待多值原子寄存器。显然，没有人愿意使用安全寄存器来写并发算法，但是这些构造说明任意使用原子寄存器的算法都可以在一种只支持安全寄存器的系统结构上实现。稍后，在考虑更实际的系统结构时，将重新回到这种实现算法的模式上，使我们可以在一些直接提供弱同步特性的系统结构上，作出只有强同步特性的假设。

86

4.3 原子快照

前面讲述了如何原子地读/写单个寄存器的值。如果要原子地读多寄存器的值该怎么办呢？这样的操作称为原子快照。

原子快照构造了一个原子寄存器数组的瞬间视图。如果能构造一个无等待的快照，则意味着一个线程可以在不延迟其他线程的情况下对存储器进行瞬时的拍照。原子快照对于备份和检查点非常有用。

Snapshot接口（图4-15）是一个MRSW的原子寄存器数组，每个寄存器对应于一个线程。其中的update()方法将值v写入这个数组中与调用者线程相对应的寄存器，而scan()方法则返回该数组的一个原子快照。

```
1  public interface Snapshot<T> {
2    public void update(T v);
3    public T[] scan();
4  }
```

图4-15 Snapshot接口

我们的目的是构造一种无等待的实现，使其等价于图4-16所示的顺序规范（即可线性化的）。这种顺序实现的关键性质就是它的scan()调用能够返回多个值，每个值对应于先前的最后一个update()，也就是说，该调用将返回处于同一个系统状态下的一组寄存器的值。

```
1  public class SeqSnapshot<T> implements Snapshot<T> {
2    T[] a_value;
3    public SeqSnapshot(int capacity, T init) {
4      a_value = (T[]) new Object[capacity];
5      for (int i = 0; i < a_value.length; i++) {
6        a_value[i] = init;
7      }
8    }
9    public synchronized void update(T v) {
10     a_value[ThreadID.get()] = v;
11   }
12   public synchronized T[] scan() {
13     T[] result = (T[]) new Object[a_value.length];
14     for (int i = 0; i < a_value.length; i++)
15       result[i] = a_value[i];
16     return result;
17   }
18 }
```

图4-16 顺序快照

4.3.1　无障碍快照

我们先从这样一个SimpleSnapshot类入手，该类的update()方法是无等待的，而其scan()方法则是无障碍的，然后再对这个算法进行扩展，使其scan()也是无等待的。

就像MRSW原子寄存器的构造一样，让每个值成为一个StampedValue<T>对象，其中包含stamp和value两个域。每个update()调用将时间戳加1。

收集是一个非原子的动作，它将寄存器的值一个接一个地复制到数组中，如果在一次收集以后紧接着又做了一次收集，且两次收集读到了相同的时间戳集，则必定存在一个时间间隔，且在这个间隔中没有线程更新了自己的寄存器，所以这次收集所得到的结果也就是在第一次收集结束时的那个瞬间，对系统状态拍摄的一个快照。称这一对收集为干净的双重收集。

在图4-17所示的SimpleSnapshot<T>类的构造中，每个线程反复地调用collect()（第27行），一旦发现一个干净的双重收集（两次收集的时间戳集合相等），该调用就返回。这个构造总是返回正确的值。update()调用是无等待的，而scan()调用则不是，其原因在于任何调用都可能被update()不断地中断，从而有可能永远执行而不结束。但是，scan()调用是无障碍的，因为若它自身的运行时间过长则会结束。

```
1  public class SimpleSnapshot<T> implements Snapshot<T> {
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public SimpleSnapshot(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      for (int i = 0; i < capacity; i++) {
6        a_table[i] = new StampedValue<T>(init);
7      }
8    }
9    public void update(T value) {
10     int me = ThreadID.get();
11     StampedValue<T> oldValue = a_table[me];
12     StampedValue<T> newValue =
13         new StampedValue<T>(oldValue.stamp+1, value);
14     a_table[me] = newValue;
15   }
16   private StampedValue<T>[] collect() {
17     StampedValue<T>[] copy = (StampedValue<T>[])
18         new StampedValue[a_table.length];
19     for (int j = 0; j < a_table.length; j++)
20       copy[j] = a_table[j];
21     return copy;
22   }
23   public T[] scan() {
24     StampedValue<T>[] oldCopy, newCopy;
25     oldCopy = collect();
26     collect: while (true) {
27       newCopy = collect();
28       if (! Arrays.equals(oldCopy, newCopy)) {
29         oldCopy = newCopy;
30         continue collect;
31       }
32       T[] result = (T[]) new Object[a_table.length];
33       for (int j = 0; j < a_table.length; j++)
34         result[j] = newCopy[j].value;
35       return result;
36     }
37   }
38 }
```

图4-17　简单的快照对象

4.3.2 无等待快照

要使scan()方法是无等待的，每个update()调用可以在修改寄存器之前先照一个快照，以此来帮助与它相冲突的scan()。若一个scan()在做双重收集时不断地失败，则可以从与它相冲突的update()调用中选取一个快照作为它自己的快照。关键在于，必须保证从提供帮助的update()中获得的快照在该scan()调用的执行过程中是可线性化的。

若一个线程执行了一次update()调用，则称该线程迁移了一步。若线程B的迁移使线程A无法得到一个干净的收集，那么线程A是否可以将线程B的最近一次快照作为它自己的快照呢？不幸的是，答案是不行。例如，在图4-18所示的情形中，B的快照在A开始它的update()调用之前就已取得了，而A看到B正在迁移，但B的这个快照并不是在A的扫描期间内拍摄的。



图4-18 此图说明为什么没能完成干净的双重收集的线程A不能使用在它的第二次收集过程中执行了update()的线程B的最近快照。B的快照在A开始scan()之前就已取得了，即B的快照与A的扫描没有重叠。这将导致这样的危险：线程C有可能在B的scan()和A的scan()之间调用了update()，从而使A所使用的scan()结果是不正确的

无等待的构造主要是基于下面这样的观察结果：若一个正在扫描的线程A在它进行重复收集的同时看到线程B迁移两次，则B必定在A的scan()过程中执行了一次完整的update()调用，这样A可以正确地使用B的快照。

图4-19、图4-20和图4-21描述了无等待的快照算法。每个update()调用都调用scan()，且将扫描的结果再加到值的标签上。更确切地说，每个写入寄存器的值都具有如图4-19所示的结构：一个stamp域，线程每次更新值都使stamp域增加；一个value域，包含着寄存器的当前值；一个snap域，包含着线程的最后一次扫描。快照算法由图4-21描述。一个正在扫描

```
1  public class StampedSnap<T> {
2    public long stamp;
3    public T    value;
4    public T[] snap;
5    public StampedSnap(T value) {
6      stamp = 0;
7      value = value;
8      snap = null;
9    }
10   public StampedSnap(long label, T value, T[] snap) {
11     stamp = label;
12     value = value;
13     snap = snap;
14   }
15 }
```

图4-19 具有时间戳的快照类

的线程创建一个布尔型数组moved[]（第13行），该数组记载在扫描过程中已经迁移的线程。如前所述，每个线程完成两次收集（第14和16行），并且检测线程的标记是否发生改变。如果没有，那么收集是干净的，扫描返回收集的结果。如果某线程的标记发生了改变（第18行），则正在扫描的线程检测moved[]数组来查看这次改变是否是线程的第一次迁移（第19行）。如果是，则返回那个线程的扫描（第20行），否则更新moved[]并且重新进入外层循环（第21行）。

```
1  public class WFSnapshot<T> implements Snapshot<T> {
2    private StampedSnap<T>[] a_table; // array of atomic MRSW registers
3    public WFSnapshot(int capacity, T init) {
4      a_table = (StampedSnap<T>[]) new StampedSnap[capacity];
5      for (int i = 0; i < a_table.length; i++) {
6        a_table[i] = new StampedSnap<T>(init);
7      }
8    }
9
10   private StampedSnap<T>[] collect() {
11     StampedSnap<T>[] copy = (StampedSnap<T>[])
12         new StampedSnap[a_table.length];
13     for (int j = 0; j < a_table.length; j++)
14       copy[j] = a_table[j];
15     return copy;
16   }
```

图4-20　单写者原子快照类和collect()方法

```
1    public void update(T value) {
2      int me = ThreadID.get();
3      T[] snap = scan();
4      StampedSnap<T> oldValue = a_table[me];
5      StampedSnap<T> newValue =
6        new StampedSnap<T>(oldValue.stamp+1, value, snap);
7      a_table[me] = newValue;
8    }
9
10   public T[] scan() {
11     StampedSnap<T>[] oldCopy;
12     StampedSnap<T>[] newCopy;
13     boolean[] moved = new boolean[a_table.length];
14     oldCopy = collect();
15     collect: while (true) {
16       newCopy = collect();
17       for (int j = 0; j < a_table.length; j++) {
18         if (oldCopy[j].stamp != newCopy[j].stamp) {
19           if (moved[j]) {
20             return newCopy[j].snap;
21           } else {
22             moved[j] = true;
23             oldCopy = newCopy;
24             continue collect;
25           }
26         }
27       }
28       T[] result = (T[]) new Object[a_table.length];
29       for (int j = 0; j < a_table.length; j++)
30         result[j] = newCopy[j].value;
31       return result;
32     }
33   }
34 }
```

图4-21　单写者原子快照的update()和scan()方法

87
91

4.3.3 正确性证明

本小节将略加详细地讲述无等待快照算法的正确性证明。

引理4.3.1 若一个正在扫描的线程做了一次干净的双重收集，那么它的返回值必是在该执行过程的某个状态存在于寄存器中的值。

证明 考虑在第一次收集的最后一个读操作到第二次收集的第一个读操作之间的这个时间段。如果有任意一个寄存器在这段时间内被更新，则标签将不匹配，那么双重收集就不是干净的了。□

引理4.3.2 若一个正在扫描的线程A在两个不同的双重收集期间观察到另一线程B的标签发生了变化，那么在最后一次收集中所读到的线程B的寄存器值必定是被某个update()调用写入的，且该update()调用是在这四次收集中第一次收集开始之后被调用的。

证明 若在一个scan()期间，线程A对线程B的寄存器的两个相继读操作返回了不同的标签值，那么这两个读操作之间线程B至少执行了一次写。由于线程B在update()调用的最后一步才对它的寄存器进行写，所以B的某个update()调用是在A第一次读操作之后的某个时间结束的，并且其他线程的写是在A的最后一对读之间发生的。这为只有B对它的寄存器写，所以断言成立。□

引理4.3.3 一个scan()所返回的值是在该scan()的调用和响应之间的某个状态存在于寄存器中的值。

证明 若scan()调用做了一次干净的双重收集，那么根据引理4.3.1，该断言成立。若此调用从另一个线程B的寄存器中获得扫描值，那么根据引理4.3.2，从B的寄存器中得到的扫描值已被B的一个scan()调用所获得，且该scan()调用介于A对B的寄存器的第一次读和第二次读之间。如果该scan()调用完成了一次干净的收集，则由引理4.3.1知结论成立；如果在该scan()调用的过程中存在另一个线程C的scan()调用，则对这种情形进行归纳。注意，在所有线程运行完之前最多只能有$n-1$个这样的嵌套调用，其中n为最大的线程数（见图4-22），所以最终必有某个嵌套的scan()调用完成了一次干净的双重收集。□

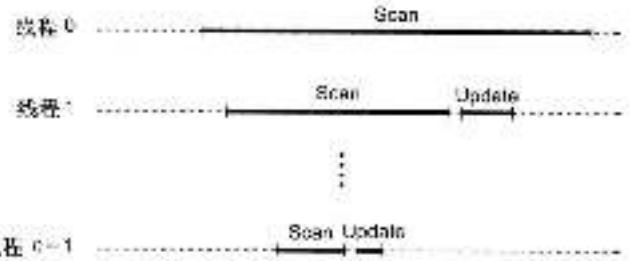


图4-22 在线程运行完之前最多有$n-1$个嵌套的scan()调用，其中n为最大的线程数。线程$n-1$的scan()包含在所有其他的scan()调用内，该调用必定有一个干净的双重收集

引理4.3.4 任何一个scan()或update()在最多执行$O(n^2)$个读或写以后将会返回。

证明 对一个给定的scan()，最多只可能有$n-1$个其他的线程。经过$n+1$次双重收集之后，要么其中的一个是干净的，要么某个线程被观察到迁移了两次。所以scan()调用是无等待的，同理知update()调用也是无等待的。□

根据引理4.3.3，由scan()返回的值形成了一个快照，因为它们都是在这个调用执行期间

的某个状态存在于寄存器中的值；该调用可以在那个时间点上被线性化。同理，可以在寄存器被写入的那个时间点上将update()调用线性化。

定理4.3.1　图4-20和图4-21给出了一种无等待的快照实现。

4.4　本章注释

Alonzo Church在1934~1935年间引入了λ演算[29]。Alan Turing在1936~1937年发表的经典论文中给出了图灵机的定义[146]。Leslie Lamport最先定义了安全、规则、原子寄存器以及寄存器层次的概念，也是最先证明可以基于安全位来实现非平凡的共享存储器的人[93,94]。Gary Peterson最先提出了原子寄存器的构造问题[125]。Jaydev Misra给出了一种关于原子寄存器的公理推导方法[116]。可线性化概念概括了Leslie Lamport和Jaydev Misra的原子寄存器概念，这一概念的提出归功于Herlihy和Wing[69]。Susmita Haldar和Krishnamurthy Vidyasankar在规则寄存器的基础上给出了一个有界的MRSW原子寄存器构造[50]。如何通过单读者原子寄存器来构造多读者原子寄存器这一问题是由Leslie Lamport[93,94]、Paul Vitányi和Baruch Awerbuch[148]作为公开问题而提出的。Paul Vitányi和Baruch Awerbuch最先提出了一种MRMW原子寄存器设计方法[148]。而这一问题的第一个解决方案则归功于Jim Anderson、Mohamed Gouda和Ambuj Singh [12,13]。其他的原子寄存器构造（只列出少数名字）分别由Jim Burns和Gary Peterson[24]、Richard Newman-Wolfe[150]、Lefteris Kirousis、Paul Spirakis和Philippas Tsigas[83]、Amos Israeli和Amnon Shaham[78]，以及Ming Li、John Tromp和Paul Vitányi[104]所提出。本章所提到的基于时间戳的MRMW原子寄存器构造是由Danny Dolev和Nir Shavit[34]提出的。

收集操作是由Mike Saks、Nir Shavit和Heather Woll[135]最早进行形式化的。原子快照的第一个构造方法则是同时由Jim Anderson[10]以及Yehuda Afek、Hagit Attiya、Danny Dolev、Eli Gafni、Michael Merritt和Nir Shavit[2]各自独立地发现的。本章所用的算法为后者所提出的。快照算法由Elizabeth Borowsky和Eli Gafni[22]以及Yehuda Afek、Gideon Stupp和Dan Touitou[4]提出。

本章所有算法的时间戳都是有界的，所以构造本身使用了有界的寄存器。有界时间戳系统由Amos Israeli和Ming Li[77]引入，而有界并发时间戳系统的引入则归功于Danny Dolev和Nir Shavit[34]。

4.5　习题

习题34. 考虑图4-6所示的MRSW安全布尔构造。判断下述情形是否为*true*：如果用一个M-值的SRSW安全寄存器数组替换SRSW安全布尔寄存器数组，那么该构造将会产生一个M-值的MRSW安全寄存器。证明你的结论。

习题35. 考虑图4-6所示的MRSW安全布尔构造。判断下述情形是否为*true*：如果用SRSW规则布尔寄存器数组替换SRSW安全布尔寄存器数组，那么该构造将产生一个MRSW规则布尔寄存器。证明你的结论。

习题36. 考虑图4-12中原子的MRSW构造。判断下述情形是否为*true*：如果用SRSW规则寄存器替换SRSW原子寄存器，那么该构造将产生一个MRSW原子寄存器。证明你的结论。

习题37. 给出一个静态一致但不规则的寄存器执行实例。

习题38. 考虑图4-6所示的MRSW安全布尔构造。判断下述情形是否为*true*：如果用M-值SRSW规则寄存器数组替换SRSW安全布尔寄存器数组，则该构造将产生一个M-值MRSW规则寄存器。证明你的结论。

习题39. 考虑图4-7所示的MRSW规则布尔构造。判断下述情形是否为*true*：如果用一个M-值MRSW安全寄存器替换MRSW安全布尔寄存器，则该构造将产生一个M-值MRSW规则寄存器。证明你的结论。

习题40. 如果用规则寄存器替换Peterson双线程互斥算法中的共享原子寄存器，该算法还能正常工作吗？

习题41. 考虑下面在分布式消息传递系统中的一种Register实现。n个处理器$P_0,\cdots,P_{n-1}$组成一个环，
94 P_i只能向$P_{(i+1) \bmod n}$发送消息。消息是以FIFO次序沿着链路来传递的。

每个处理器保存一个共享寄存器的拷贝。

- 处理器读取本地存储器中的拷贝来读取寄存器内容。
- 处理器P_i通过向$P_{(i+1) \bmod n}$发送消息“P_i：向x中写v”，开始执行向寄存器x中写入v的write()调用。
- 如果P_i收到消息“P_j：向x中写v”（$i \neq j$），那么它就将值v写入x在自己的本地拷贝中，然后将消息转发给$P_{(i+1) \bmod n}$。
- 如果P_i收到一个消息“P_i：向x中写v”，那么它就将值v写入x在自己的本地拷贝中，然后丢弃此消息。这次write()到此结束。

简要给出证明或举出反例。

若write()调用彼此间没有重叠，

- 这个寄存器实现是规则的吗？
- 该实现是原子的吗？

若有多个处理器同时调用write()，

- 这个寄存器实现是原子的吗？

习题42. 假设你的竞争对手Acme Atomic Register公司开发出了一种用原子布尔（单个位）寄存器来构造针对单读者-单写者的一次写原子寄存器的方法。通过调查，得到了如图4-23所示的代码片段，不幸的是其中丢失了read()方法的部分，为这个类设计一个能正常工作的read()方法，并且证明（非形式化地）为什么它能工作。（注意，该寄存器是一次写的，这表示读操作最多与一个写操作重叠。）

习题43. 证明图4-6所示的基于SRSW安全布尔寄存器的MRSW安全布尔寄存器构造中，如果组件寄存器为SRSW规则寄存器，则该构造是一个正确的MRSW规则寄存器实现。

习题44. 单调计数器是数据结构$c = c_1 \cdots c_n$（即c由单个数字c_i组成，$n>0$），$c^0 \lessdot c^1 \lessdot c^2 \lessdot \cdots$，其中$c^0$，$c^1$，$c^2$，…表示$c$所假定的连续值。

如果c不是一个由单个位组成的数，那么对c的读/写可以包括多个独立的操作。对c的一次读者与一个或多个对c的写并发执行，则有可能得到一个与任何c^i（$i>0$）都不相同的值。因此，读到的值有可能为多种不同的版本轨迹。若一个读得到版本轨迹为$c^{i_1},\cdots,c^{i_n}$，那么称它得到了值$c^{[k,l]}$，$k=\min(i_1,\cdots,i_n)$，$l=\max(i_1,\cdots,i_n)$，且$0 \leqslant k \leqslant l$。若$k=l$，则$c^{[k,l]}=c^l$且读到的是与$c$一致的版本。

这种计数器的正确性条件可以保证如果一个读得到值$c^{[k,l]}$，那么有$c^k \lessdot c^{[k,l]} \lessdot c^l$。假定每个单独的位$c_i$是原子的。

```
1  class AcmeRegister implements Register{
2    // N is the length of the register
3    // Atomic multi-reader single-writer registers
4    private BoolRegister[] b = new BoolMRSWRegister[3 * N];
5    public void write(int x) {
6      boolean[] v = intToBooleanArray(x);
7      // copy v[i] to b[i] in ascending order of i
8      for (int i = 0; i < N; i++)
9        b[i].write(v[i]);
10     // copy v[i] to b[N+i] in ascending order of i
11     for (int i = 0; i < N; i++)
12       b[N+i].write(v[i]);
13     // copy v[i] to b[2N+i] in ascending order of i
14     for (int i = 0; i < N; i++)
15       b[(2*N)+i].write(v[i]);
16   }
17   public int read() {
18     // missing code
19   }
20 }
```

图4-23　Acme公司的部分寄存器实现代码

假定下面的定理成立，给出一种单调计数器实现：

定理4.5.1　如果$c = c_1 \cdots c_n$总是从右至左写入，那么一个从左至右的读得到的值序列为$c_1^{k_1} c_2^{k_2} \cdots c_n^{k_n}$，且$k_1 \geqslant \ell_1 \geqslant k_2 \geqslant \cdots \geqslant k_n \geqslant \ell_n$。

定理4.5.2　令$c = c_1 \cdots c_n$且假定$c^0 \leqslant c^1 \leqslant \cdots$。

1. 若$i_1 \leqslant \cdots \leqslant i_n \leqslant i$，则$c_1^{i_1} \cdots c_n^{i_n} \leqslant c^i$。

2. 若$i_1 \geqslant \cdots \geqslant i_n \geqslant i$，则$c_1^{i_1} \cdots c_n^{i_n} \geqslant c^i$。

定理4.5.3　令$c = c \cdots c_n$，假设$c^0 \leqslant c^1 \leqslant \cdots$且每个位上的数字$c_i$是原子的。

1. 若c总是从右至左写入，那么一个从左至右的读得到的值为$c^{i'} \leqslant c^i$。

2. 若c总是从左至右写入，那么一个从右至左的读得到的值为$c^{i'} \geqslant c^i$。

注意：

如果对c的一个读得到版本c^i（$i>0$）的轨迹，那么：

- 该读操作的开始必领先于对c^{i+1}写操作的结束。
- 该读操作的结束必落后于对c^i写操作的开始。

95
96

此外，对于每一个$j>0$，如果对c_j的读（写）操作在对c_{j-1}的读（写）操作开始之前完成，则称对c的读（写）操作是从左至右的。类似地，可以定义从右至左的读（写）操作。

最后，始终要记住下标表示c中的各个位，而上标表示c所假定的连续的值。

习题45. 证明习题44中的定理4.5.1。注意，由于$k_j \leqslant \ell_j$，只需证明当$1 \leqslant j < m$时，$\ell_j \leqslant k_{j+1}$。

证明习题44中的定理4.5.3，假定引理成立。

习题46. 本章讲述了安全规则寄存器，定义环绕寄存器为这样的寄存器：存在一个值v，对v加1所产生的结果为0，而不是$v+1$。

如果将Bakery算法中的共享变量替换成(a)闪动的，(b)安全的，(c)环绕的寄存器，那么该算法是否满足(1)互斥条件，(2)先来先服务顺序？

给出6个答案（其中某些答案可能包含其他的答案），并逐个证明。

97

第5章

The Art of Multiprocessor Programming (Revised First Edition)

同步原子操作的相对能力

假设你在负责设计一种新的多处理器，应该将哪种原子指令包含进来呢？在相关文献中已介绍了一系列令人眼花缭乱的可选指令：读/写存储器、getAndDecrement()、swap()、getAndComplement()、compareAndSet()以及许多其他的指令。如果提供对所有这些指令的支持将会使设计变得非常复杂而且效率低下，但若提供了错误的指令将会使一些重要的同步问题很难解决甚至无法解决。

我们的目标就是找出一组基本的同步操作原语，用于解决实际中可能出现的各种同步问题。（当然，为了方便起见也可以提供一些非基本的同步操作。）为了实现这个目标，需要提供一种能够评测各种同步原语能力的测试方法：这些同步原语能够解决什么样的同步问题，解决问题的效率如何。

如果对一个并发对象的每一次方法调用都能在有限步内完成，则称该并发对象的实现是无等待的。如果能保证某个方法的无限次调用都能在有限步内完成，则称该方法是无锁的。在第4章中已介绍过无等待（按定义也是无锁的）寄存器的实现。评测同步指令能力的一种方法就是评价同步指令对于具体对象（如队列、栈、表等）的实现的支持程度如何。正如第4章所讲述的，我们主要评测那些无等待或无锁的解决方法，也就是说，只评测那些能够保证状态的演进不依赖外界支持的解决方法。[⊖]

所有的同步指令并不是等价的。如果把同步原子指令看作是其对外的方法就是指令本身的对象（通常书中称这些对象为同步原语），则可以证明存在着一种由同步原语组成的无限层次的层次结构，任一层的原语都不能用在更高层原语的无等待或无锁实现中。其证明思路很简单：在这种层次结构中，每个类都有一个相关的一致数，所谓一致数就是这个类的对象解决基本的同步问题（称为一致性）时所能针对的最大线程数。我们将会看到在一个有n个或更多个并发线程的系统中，不可能使用一致数小于n的对象构造一个一致数为n的对象的无等待或无锁实现。

5.1 一致数

一致性是一个看起来无关痛痒且有点抽象的问题，从算法设计到硬件系统结构的各个方面对该问题都有大量结论。一致性对象只提供单个decide()方法，如图5-1所示。每个线程以输入v最多调用decide()方法一次。一致性对象的decide()方法将返回一个满足下列条件的值：

- 一致性：所有的线程都决定同一个值。
- 有效性：这个共同的决定值是某个线程的输入。

换句话说，并发一致性对象可以被线性化为一个串行一致性对象，其中值被选中的线程首先完成它的decide()调用。有时，我们只关注所有输入为1或0的一致性问题。这种特殊的

⊖ 仅评测满足演进依赖条件的解决方案是没有意义的，因为那些基于依赖条件（如无障碍或无死锁）实现方法的实现能力被它们所依赖的操作系统的能力所屏蔽。

问题称为二进制一致性。为方便表述，本节只讨论二进制一致性，但所有的结论同样适用于一般的一致性问题。

```
public interface Consensus<T> {
  T decide(T value);
}
```

图5-1　一致性对象的接口

下面着重考虑一致性问题的无等待解决方案，即一致性对象的无等待并发实现。读者将会看到，对于一个给定的一致性对象，既然它的decide()方法只被每个线程执行一次，那么根据定义，一个无锁的实现也是无等待的，反之亦然。因此，只需考虑无等待实现，出于历史的原因，所有以无等待方式实现的一致性类被称为一致性协议。 100

本章仅考虑具有确定顺序说明的对象类（即每个串行的方法调用都有单一的返回结果）。[⊖]

我们要理解一个特定对象的类是否能解决一致性问题。但如何使这个概念更加准确呢？首先，如果将这些对象看作是由系统底层（如操作系统或者硬件）提供的，则只用考虑类的性质而不需关心对象的个数。（如果系统能够提供这种类的一个对象，它就能提供更多的对象。）其次，可以合理地假设现代系统都能提供大量的读/写存储器进行簿记。基于上述两个观点给出下面的定义。

定义5.1.1　如果存在一种使用类C的任何数量的对象和原子寄存器的一致性协议，则类C能够解决n线程一致性问题。

定义5.1.2　类C的**一致数**是指用这个类来解决n线程一致性时所能针对的最大的n值。如果最大的n值不存在，则称这个类的一致数是**无限的**。

推论5.1.1　假设类C的一个对象可以通过类D的一个或多个对象以及一定数量的原子寄存器实现，如果类C可以解决n线程一致性，那么类D也可以。

状态和价

最好的入手点就是考虑下面这种最简单的情形：双线程（称为A和B）的二进制一致性（输入为0或1）。每个线程不断地进行迁移直到它选定一个值。这里的迁移是指对一个共享对象的一次方法调用。协议状态包含线程的状态和共享对象的状态。初始状态指所有线程开始迁移之前的协议状态，结束状态指所有线程结束以后的协议状态。结束状态的决定值是指由所有处于结束状态的线程所决定的值。

无等待协议的所有可能状态组成了一棵树，其中一个结点代表一种可能的协议状态，一条边则代表某个线程的一次可能的迁移。图5-2描述了双线程的协议树，其中每个线程可以迁移两次。A的迁移用黑色表示，B的迁移则用灰色表示。A的一条从结点s到s'的边表示：如果A在协议状态s迁移，则新的协议状态为s'，s'称为s的后继状态。由于协议是无等待的，所以树一定是有限的。叶子结点代表最终的协议状态，并且被标上它们的决定值（0或1）。 101

如果一个协议状态的决定值是不确定的，则该协议状态是二价的（bivalent）：从这个状态开始执行的线程可以决定0或1。反之，如果决定值是确定的，则该协议状态是单价的（univalent）：所有从该状态开始的执行都决定同一个值。如果一个协议状态是单价的并且决

⊖　之所以避开非确定对象，是因为它们的结构要复杂得多，参见本章注释。

定值都是1，则它是1-价（1-valent）的，同样可以定义0-价的协议状态。在图5-2中，二价状态是这样的一个结点，其子孙结点中既包含标记为0的叶子结点也包含标记为1的叶子结点，而单价状态的子孙结点中只包含标记为单一决定值的叶子结点。

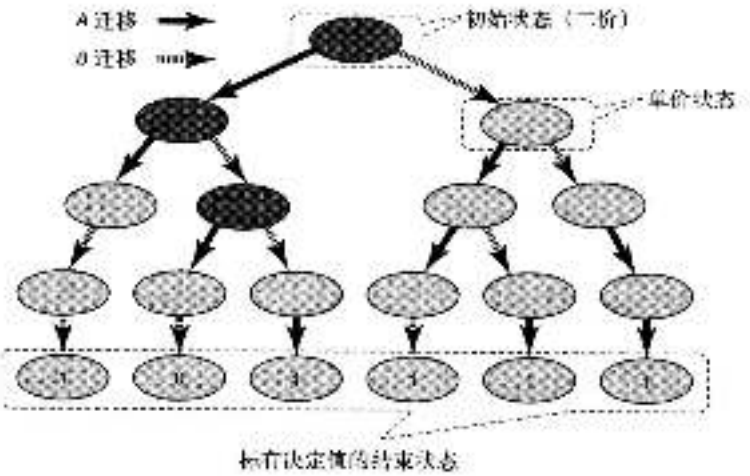


图5-2 双线程A和B的执行树。深色结点表示二价状态，浅色结点表示单价状态

下面的引理说明存在着一个二价的初始状态。该结论表明协议的结果不能事先确定，必须依赖于读和写的交叉次序。

引理5.1.1 每个双线程一致性协议都有一个二价的初始状态。

证明 考虑A的输入为0、B的输入为1的初始状态。如果A在B开始之前完成协议，则A决定0，因为A必须决定某个线程的输入，而0是它所看到的唯一输入（A不能决定1，因为它没有办法把这个状态同B输入0时的状态区分开来）。对称地，如果B在A开始之前完成协议，那么B决定1，因为它必须决定某个线程的输入，并且1是它所看到的唯一的输入。由此可见，A输入0并且B输入1时的初始状态是二价的。 □

102

引理5.1.2 每个n线程一致性协议都存在一个二价的初始状态。

证明 留作习题。 □

一个协议状态是临界的，如果它满足：

- 它是二价的。
- 如果任何一个线程迁移，该协议状态将变为单价的。

引理5.1.3 每一个无等待的一致性协议都有一个临界状态。

证明 假设引理不成立，根据引理5.1.2，该协议有一个二价的初始状态。从这个初态开始运行这个协议。只要存在某个线程不用把协议状态变为单价就能迁移，则让该线程迁移。如果这个协议的运行不终止，则它不是无等待的。因此，这个协议最终必定进入一个不可能进行上述迁移的状态，而这个状态为一个临界状态。 □

至今为止，对于任何一致性协议，无论它使用哪种共享对象类，我们所证明的一切结论都适用。下面来研究一些特殊的对象类。

5.2 原子寄存器

首先考虑这样一个问题：能否用原子寄存器解决一致性问题？令人惊讶的是，答案是否

定的。下面将证明不存在针对双线程的二进制一致性协议。我们将下述结论的证明留作习题：如果两个线程对两个值不能达成一致，那么n个线程对k个值也不能达成一致，其中$n>2$，$k>2$。

在讨论是否存在解决某特定问题的协议时，通常会构造这样的场景："如果存在一个这样的协议，则在如下的情形下，该协议将具有这样的行为……"其中一个常用的场景就是让一个线程（称为A）完全独自运行，直到它完成协议。由于这种特殊的场景经常会被用到，所以将它命名为"让A独奏"。

定理5.2.1 原子寄存器的一致数为1。

证明 假设存在一个针对线程A、B的二进制一致性协议，我们来分析这个协议的特性并由此推出矛盾。

根据引理5.1.3，可以让这个协议运行直到它到达一个临界状态s。假设A的下一个迁移将使该协议到达一个0-价状态，B的下一个迁移将使该协议到达一个1-价状态。（若不是这样，则更换线程名。）那么A和B将会调用哪些方法呢？现在来考虑所有的可能：其中的一个线程对一个寄存器读，或者两个线程分别对不同的寄存器写，或者两个线程对同一寄存器写。

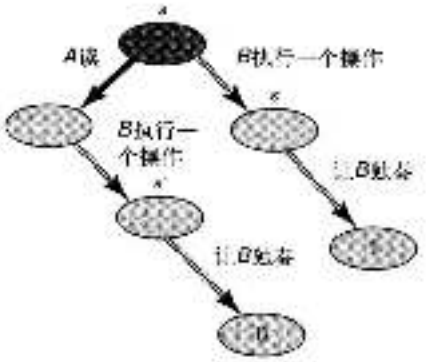


图5-3 场景：A首先读。在第一种执行情形中，B先迁移，使该协议到达一个1-价状态s'，然后让B独奏并最终决定1。在第二种执行情形中，A先迁移，使该协议到达一个0-价状态s''，然后让B从状态s''开始独奏并最终决定0

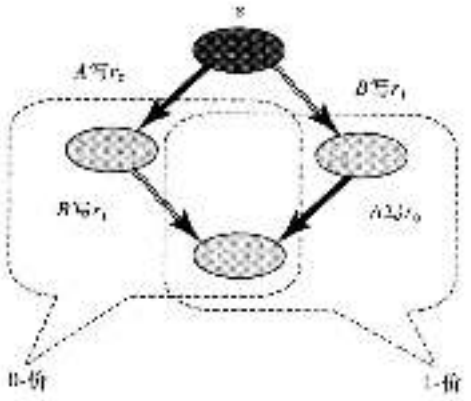
假设A准备读一个给定的寄存器（B可能读/写同一个寄存器或者读/写不同的寄存器），如图5-3所示。考虑两种可能的执行情形。在第一种情形中，B先迁移，使该协议到达一个1-价状态s'，然后让B独奏并最终决定值1。在第二种情形中，A先迁移，使该协议到达一个0-价状态s''，然后B从状态s''开始独奏并最终决定0。问题是状态s'和s''对B来说是不可区分的（A的读只能改变它自己的局部线程状态，该局部状态对B来说是不可见的），这意味着B在两种情形下都必须决定同一个值，得到矛盾。

假设两个线程准备写不同的寄存器，如图5-4所示。A准备写r_0而B准备写r_1。考虑两种可能的执行情形。在第一种情形中，A先写r_0然后B再写r_1，由于A首先执行，所以最终的协议状态是0-价的。在第二种情形中，B先写r_1然后A再写r_0，由于B先执行，所以最终的协议状态是1-价的。

问题是上述两种情形都导致了不可区分的协议状态。无论A或B都无法确定哪一个迁移先进行。结束状态既是0-价状态又是1-价状态，得到矛盾。

最后，假设两个线程准备写同一个寄存器r，如图5-5所示。考虑两种可能的执行情形。在A先写的情形下，协议状态s'是0-价的，然后让B独奏并决定0。在B先写的情形下，协议状态s''是1-价的，然后让B独奏并决定1。问题是B无法区分s'和s''（因为不管是在s'还是s''，B都重写了寄存器r并抹去了任何A的写痕迹），所以B从任意一个状态开始都必须决定同一个值，得到矛盾。□

推论5.2.1 对于任意一致数大于1的对象，不可能用原子寄存器构造该对象的无等待实现。



105 图5-4 场景：A和B写不同的寄存器

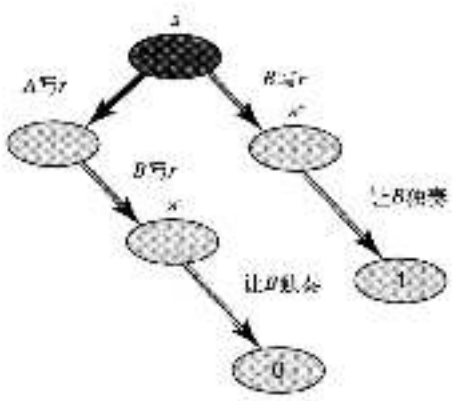


图5-5 场景：A和B写同一个寄存器

上述非论也许是计算机科学领域里最引人注目的不可能性结论之一。它说明，如果我们要在现代多处理器上实现无锁的并发数据结构，硬件必须提供原子的同步操作而不是加载/存储（读/写）操作。

5.3 一致性协议

现在考虑我们所关注的一些对象类，研究每种类能在多大程度上解决一致性问题。这些协议具有一种通用范式，如图5-6所示。该对象具有一个原子寄存器数组，每个decide()方法在数组中指定自己的输入值，然后继续执行一系列操作步，从指定值中决定一个值。我们将使用各种同步对象来构造不同的decide()方法实现。

```
1  public abstract class ConsensusProtocol<T> implements Consensus<T> {
2    protected T[] proposed = (T[]) new Object[N];
3    // announce my input value to the other threads
4    void propose(T value) {
5      proposed[ThreadID.get()] = value;
6    }
7    // figure out which thread was first
8    abstract public T decide(T value);
9  }
```

图5-6 通用一致性协议

5.4 FIFO队列

第3章给出了只使用原子寄存器且仅针对单入队者和单出队者的FIFO队列的无等待实现。能否构造一种支持多个入队者和出队者的FIFO队列的无等待实现呢？首先来研究一个比较特殊的问题：能够用原子寄存器构造出针对双出队者的FIFO队列的无等待实现吗？

定理5.4.1 双出队者FIFO队列类的一致数至少为2。

证明 图5-7描述了采用单个FIFO队列实现的双线程一致性协议。队列中存放着整数，通
106 过将值WIN和值LOSE先后入队来对队列进行初始化。和其他所有的一致性协议一样，decide()首先调用propose(v)，将值v存放在指定输入值的共享数组proposed[]中，然后让队列中的下一项出队。如果这一项的值是WIN，则首先通过调用线程，并且决定它自己的值。如果这一项

的值是LOSE，则另一个线程首先被选定。这样调用线程按照数组proposed[]中的声明返回另一个线程的输入。

```
1  public class QueueConsensus<T> extends ConsensusProtocol<T> {
2    private static final int WIN = 0; // first thread
3    private static final int LOSE = 1; // second thread
4    Queue queue;
5    // initialize queue with two items
6    public QueueConsensus() {
7      queue = new Queue();
8      queue.enq(WIN);
9      queue.enq(LOSE);
10   }
11   // figure out which thread was first
12   public T decide(T value) {
13     propose(value);
14     int status = queue.deq();
15     int i = ThreadID.get();
16     if (status == WIN)
17       return proposed[i];
18     else
19       return proposed[1-i];
20   }
21 }
```

图5.7　用一个FIFO队列实现的双线程一致性

由于不存在环路，所以该协议是无等待的。如果每一个线程都返回它自己的输入，那么它们必须都让WIN出队，这将违背FIFO队列的定义。如果每一个线程都返回另一个线程的输入，那么它们必须都让LOSE出队，同样也违背了队列的定义。

从这个分析可以看出，有效性条件遵循：让WIN出队的线程，在任何值出队之前已把它自己的输入存放在数组proposed[]中。 □

对于任何其他的以不同调用次序返回不同结果的对象，像栈、优先级队列、表、集合等，只需将这段程序稍作修改就可以产生相应的协议。

推论5.4.1　用一组原子寄存器不可能构造队列、栈、优先级队列、集合或链表的无等待实现。

虽然FIFO队列能够解决双线程一致性问题，但是不能解决3线程一致性问题。

定理5.4.2　FIFO队列的一致数为2。

证明　用反证法，假设存在一个针对线程A、B和C的一致性协议。根据引理5.1.3，该协议有一个临界状态s。不失一般性，假设A的下一个迁移将使该协议到达一个0-价状态，而B的下一个迁移将使该协议到达一个1-价状态。和前面一样，剩下的工作就是情形分析。

首先，因为A和B的未决迁移不能交换，这意味着它们都将调用s一对象的方法。其次，由于A和B不能读/写共享寄存器，所以它们将调用单个队列对象的方法。

首先，假设A、B都调用deq()，如图5-8所示。如果A

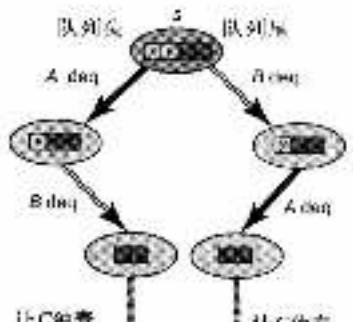


图5-8　场景：A和B都调用deq()

先出队然后B再出队，则协议状态为s'；如果出队次序相反，则协议状态为s''。由于s'是0-价的，那么，如果让C从s'开始不被中断地一直运行，则它将会决定0。由于s''是1-价的，如果C从s''开始不被中断地一直运行，则决定1。但对C来说s'和s''是不可区分的（从队列中移出了相同的两个项），所以在两种状态C都必须决定相同的值，得到矛盾。

其次，假设A调用enq(a)，而B调用deq()，如果队列非空，那么直接产生矛盾，其原因在于这两个方法可以交换（每种方法在队列的不同端进行操作）：C无法观察到它们发生的次序，假设队列为空，若B先对空队列执行出队操作然后A执行入队操作，则到达1-价状态；若A单独执行了入队操作，则到达0-价状态。然而，从这个0-价状态开始到达1-价状态对C是不可区分的。注意，我们并不关心一个deq()对一个空队列到底做了什么（中断或等待），因为这并不影响该状态对C的可见性。

108 最后，假设A调用enq(a)，而B调用enq(b)，如图5-9所示，设s'为下面操作结束时的状态：

1. A和B分别使得a和b入队。

2. 运行A直至它使a出队。（由于方法deq()是观察队列状态的唯一方法，所以A在还未观察到a或b之前不能做出决定。）

3. 在A进一步执行之前，运行B直至它使b出队。

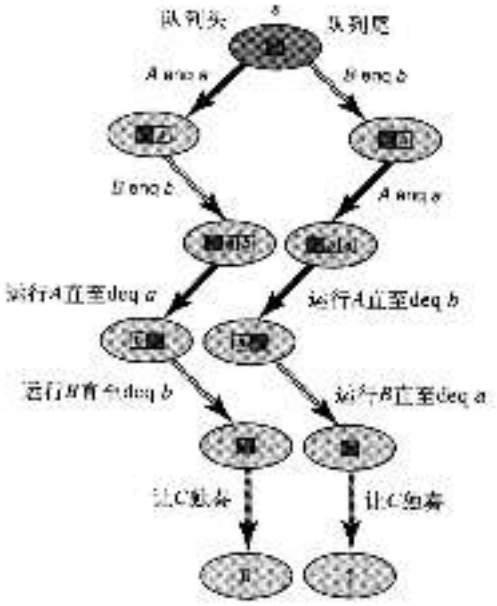


图5-9 场景：A调用enq(a)、B调用enq(b)。注意，在A和B分别将它们各自的项入队以后，一个新项又被A入队（B也可以在该项出队前入队新的项）但该项还未出队。然而，在两种执行情形中该项是相同的

设s''为下面操作交替执行后的状态：

1. B和A分别使得b和a入队。

2. 运行A直至它使b出队。

3. 在A进一步执行之前，运行B直至它使a出队。

显然，s'为0-价的而s''为1-价的。在这两种情形下，A的执行都是相同的，A一直执行直到

它使a或b出队为止。由于A在修改其他对象之前被中止，那么在两种情况下B的执行也是相同的，B一直运行直到它使a或b出队为止。按照现在已熟悉的论证方法，因为s'和s''对C是不可区分的，所以产生矛盾。 □ 109

对上述论证过程稍加修改，就可以证明许多类似的数据类型，如集合、栈、双端队列以及优先级队列，它们的一致数也正好为2。

5.5 多重赋值对象

在(m, n)-赋值问题（或称为多重赋值）中（其中$n \geq m > 1$），给定一个具有n个域的对象（有时为一个n元数组）。方法assign()以m个值v_i（$i \in 0, \cdots, m-1$）和m个索引值i_j（$j \in 0, \cdots, m-1$，$i_j \in 0, \cdots, n-1$）为输入参数，将值v_j原子地赋予数组元素i_j。方法read()以索引i为参数，返回数组中的第i个元素。该问题是原子快照对象（第4章）的对偶问题，在原子快照对象中，是对单个域赋值而对多个域进行原子地读。由于快照可以采用读/写寄存器实现，所以定理5.2.1隐含地说明快照对象的一致数为1。

图5-10描述了针对(2,3)-赋值对象的一种基于锁的实现。其中，线程能够对三个数组元素中的任意两个进行原子地赋值。

```
1  public class Assign23 {
2    int[] r = new int[3];
3    public Assign23(int init) {
4     for (int i = 0; i < r.length; i++)
5       r[i] = init;
6    }
7    public synchronized void assign(T v0, T v1, int i0, int i1) {
8      r[i0] = v0;
9      r[i1] = v1;
10   }
11   public synchronized int read(int i) {
12     return r[i];
13   }
14 }
```

图5-10 (2,3)-赋值对象的基于锁的实现

定理5.5.1 *对任意的$n \geq m > 1$，不存在通过原子寄存器构造的(m, n)-赋值对象的无等待实现。*

证明 对于一个给定的(2, 3)-赋值对象以及两个线程，只需证明能够解决双线程一致性即可。（习题75要求证明这个结论。）与通常一样，decide()方法必须决定哪一个线程首先运行。所有的数组元素被初始化为*null*。图5-11描述了该协议。线程A（原子地）写域0和域1，同时线程B（原子地）写域1和域2。然后它们尝试着决定谁先运行。从线程A的角度来看，存在着3种情形（如图5-12所示）： 110

- 如果先执行A的赋值，而B的赋值还没有发生，则域0和域1为A的值，域2的值为*null*。这样A决定它自己的输入。
- 如果先执行A的赋值，随后执行B的赋值，则域0为A的值，域1和域2为B的值。这样A决定它自己的输入。
- 如果先执行B的赋值，随后执行A的赋值，则域0和域1为A的值，域2为B的值。这样A决

定B的输入。

同样的分析也适用于B。

```
1  public class Multi2consensus<T> extends ConsensusProtocol<T> {
2    private final int NULL = -1;
3    Assign23 assign23 = new Assign23(NULL);
4    public T decide(T value) {
5      propose(value);
6      int i = ThreadID.get();
7      int j = 1-i;
8      // double assignment
9      assign23.assign(i, i, i, i+1);
10     int other = assign23.read((i+2) % 3);
11     if (other == NULL || other == assign23.read(i))
12       return proposed[i];      // I win
13     else
14       return proposed[j];      // I lose
15   }
16 }
```

图5-11 使用(2,3)-多重赋值的双线程一致性

定理5.5.2 对于$n>1$，原子的$\left(n, \frac{n(n+1)}{2}\right)$-寄存器赋值的一致数至少为$n$。

证明 我们来设计一种针对n线程0，…，$n-1$的一致性协议。该协议使用一个$\left(n, \frac{n(n+1)}{2}\right)$-赋值对象。为方便起见，以下面的方式命名对象的域。有n个域r_0，…，r_{n-1}，其中线程i写寄存器r_i；有$n(n-1)/2$个域r_{ij}，其中$i>j$，线程i和线程j都写域r_{ij}。所有的域都被初始化为$null$。每个线程i把它的输入值原子地赋给n个域：它的单写者域r_i及其$n-1$个多写者寄存器r_{ij}，该协议决定要被赋予的第一个值。

111

在给自己的域赋值之后，线程按如下方法决定任意两个线程i、j的相对赋值顺序：

- 读r_{ij}。如果该值为$null$，则不发生任何赋值。
- 否则，读r_i和r_j。如果r_i的值为$null$，则线程j在i之前赋值。按照同样的方法处理r_j。
- 如果r_i和r_j都不为$null$，则重读r_{ij}。如果其值等于r_i中的值，那么j在i之前赋值，否则，按照相反次序赋值。

不断地重复这个过程，则一个线程可以决定哪个值是由最早的赋值所写的。图5-13给出了两个决定次序的例子。□

注意，对于任意$m>n>1$且其对偶结构和原子快照的一致数最大为1的线程，多重赋值能

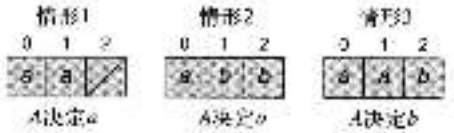


图5-12 使用多重赋值的一致性：可能的情形

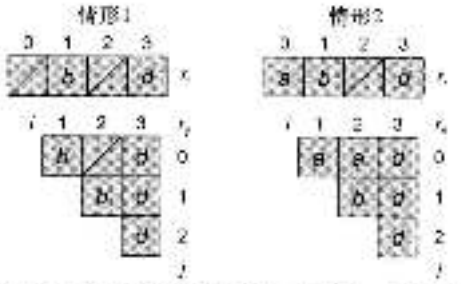


图5-13 使用(4,10)-赋值解决4线程一致性时可能出现的两种情况。在第一种情形中，只有线程B和D，B首先准备赋值并且赢得了一致性。在第二种情形中，有三个线程A、B和D，和前面一样，通过让B首先赋值，D最后赋值，使B赢得一致性。线程之间的次序可以通过查看任意两个线程之间的两两次序来确定。因为赋值是原子的，所以这些单独的次序总是一致的，并且定义了所有调用之间的全序关系

够解决它们的一致性问题。虽然这两个问题看起来相似，但我们已证明了对多存储单元的原子写要比对它们的原子读需要更多的计算能力。

5.6　读-改-写操作

由多处理器硬件所提供的大多数经典同步操作可以表示为读-改-写（RMW）操作。按照它们的对象术语，称为读-改-写寄存器。考虑一个将整数值封装起来的RMW寄存器，令$\mathcal{F}$为从整数到整数的映射函数集。[⊖] 112

对于某个$f\in\mathcal{F}$，如果一个方法能够用$f(v)$原子地替换寄存器的当前值v，并返回寄存器的先前值v，则称该方法是一个对于函数集$\mathcal{F}$的RMW操作（有时$\mathcal{F}$是一个单例集）。我们遵循Java的规定，把使用函数mumble的RMW方法称为getAndMumble()。

例如，java.util.concurrent包提供了具有各种RMW方法的AtomicInteger类：

- getAndSet(v)方法用v原子地替换寄存器的当前值并返回先前值。该方法（或称为swap()）是一个对于类型为$f_v(x)=v$的常量函数集的RMW方法。
- getAndIncrement()方法将寄存器的当前值原子地加1并返回先前值。该方法（或称为取值-自增）是一个对于函数$f(x)=x+1$的RMW方法。
- getAndAdd(k)方法将寄存器的当前值原子地加k并返回先前值。该方法（或称为取值-增加）是一个对于函数集$f_k(x)=x+k$的RMW方法。
- compareAndSet()方法使用两个参数值：期望值e和更新值u，如果寄存器值等于e，则用u原子地替换它，否则不改变。同时，该方法返回一个布尔值以说明寄存器值是否被改变。[⊖]形式化地来说，如果$x=e$则$f_{e,u}(x)=x$，否则$f_{e,u}(x)=u$，严格地讲，compareAndSet()方法并不是一个对于$f_{e,u}$的RMW方法，因为RMW方法应返回寄存器的先前值而不是一个布尔值。但这种区别只是技术上的问题。
- get()方法返回寄存器的值。该方法是一个对于恒等函数$f(v)=v$的RMW方法。

由于RMW方法有可能成为潜在的硬件原语，所以人们十分关注这些被刻在硅片上而并不是刻在石头上的RMW方法的研究工作。本书采用Java同步术语定义RMW寄存器及其方法，然而，在实际编程中，它们几乎完全对应于真正的（或被建议的）硬件同步原语。

如果在一个RMW方法的函数集中至少包含一个非恒等函数，那么该方法是非平凡的（nontrivial）。

定理5.6.1　非平凡RMW寄存器的一致数至少为2。

证明　图5-14给出了一种双线程一致性协议。由于在$\mathcal{F}$中必定存在f为非恒等函数，那么必存在着值v使得$f(v)\neq v$。在decide()方法中，propose(v)先将线程的输入v写入数组proposed[]中。然后，每个线程对一个共享寄存器调用该RMW方法。如果线程的调用返回v，那么它被线性化为第一个，并且决定自己的值；否则，它被线性化为第二个，并且决定为一个线程的值。 113 □

推论5.6.1　对于两个或两个以上的线程，使用原子寄存器不可能为它们构造出任意非平凡RMW方法的无等待实现。

⊖　为简单起见，仅考虑具有整数值的寄存器，但它们同样也可以表示对其他对象的引用。

```
1  class RMWConsensus extends ConsensusProtocol {
2    // initialize to v such that f(v) != v
3    private RMWRegister r = new RMWRegister(v);
4    public Object decide(Object value) {
5      propose(value);
6      int i = ThreadID.get();      // my index
7      int j = 1-i;                 // other's index
8      if (r.rmw() == v)            // I'm first, I win
9        return proposed[i];
10     else                         // I'm second, I lose
11       return proposed[j];
12   }
13 }
```

图5-14 使用RMW操作的双线程一致性协议

5.7 Common2 RMW操作

下面分析Common2 RMW寄存器，这种寄存器是20世纪末大多数处理器所支持的常用同步原语。虽然Common2寄存器和非平凡的RMW寄存器一样，具有比原子寄存器更为强大的能力，但仍能证明它的一致数恰好为2，这意味着Common2寄存器的同步能力也是有限的。然而幸运的是，在现代处理器系统结构中已基本上放弃了这种同步原语。

定义5.7.1 对于任意的值v以及函数集F中的函数f_i和f_j，如果它们满足下面条件之一：

- f_i和f_j可交换：$f_i(f_j(v))=f_j(f_i(v))$，或者
- 一个函数重写另一个函数：$f_i(f_j(v))=f_i(v)$或$f_j(f_i(v))=f_j(v)$。

则函数集F属于Common2。

定义5.7.2 如果一个RMW寄存器的函数集F属于Common2，则该寄存器也属于Common2。

114

文献中的很多RMW寄存器只提供一个非平凡函数。例如，getAndSet()使用常量函数来重写任意的先前值。getAndIncrement()和getAndAdd()方法使用了可交换函数。

首先非形式化地说明为什么Common2 RMW寄存器不能解决3线程一致性问题。第一个线程（获胜者）总能识别出它是第一个，第二个和第三个线程（失败者）也能够识别出它们是失败者。然而，由于用来定义Common2操作后的协议状态的函数是可交换或可重写的，因此，失败者线程无法识别出其他线程中的哪一个首先执行（成为获胜者）。又因为协议是无等待的，所以它不可能一直等待直到找出哪个是获胜者为止。下面对该结论进行更为准确的论证。

定理5.7.1 任意Common2 RMW寄存器的一致数（恰好）为2。

证明 定理5.6.1已证明所有RMW寄存器的一致数至少为2。现只需证明任意Common2寄存器都不能解决3线程一致性问题。

采用反证法，假定存在一个只使用Common2寄存器和读/写寄存器的3线程协议。不妨假设线程A、B和C通过Common2寄存器达到了一致性。根据引理5.1.3，任何这样的协议都有一个临界状态s，在此状态下，协议是二价的，同时，任意线程的任意方法调用都会使该协议进入单价状态。

现在进行情形分析，检查各种可能的方法调用。通过定理5.2.1的证明过程中所采用的推理分析可以看出，未决的方法不可能是读或写，同样线程也不可能调用不同对象的方法。由此推出线程准备调用单个寄存器r的RMW方法。

假设A准备调用一个针对函数f_A的方法，使协议进入一个0-价状态。B准备调用一个针对f_B的方法，使协议进入一个1-价状态。则存在两种可能的情形：

1. 如图5-15所示，一个函数重写了另一个函数：$f_B(f_A(v))=f_B(v)$。令s'是A先调用f_A，随后B再调用f_B所导致的状态。由于s'是0-价的，如果让C单独运行直至完成协议，那么它将决定0。令s''是B单独调用f_B所导致的状态。由于s''是1-价的，如果让C单独运行直至完成协议，则它将决定1。问题是这两个可能的寄存器状态$f_B(f_A(v))$和$f_B(v)$是相同的，所以s'和s''只能在A和B的内部状态中不同。如果现在让线程C开始执行，由于C不需要与A，B通信即可完成协议，那么这两个状态对C来说是一样的，因此，从这两个状态不可能决定出不同的值。

2. 函数相互交换：$f_A(f_B(v))=f_B(f_A(v))$。令s'是A先调用f_A，随后B再调用f_B所导致的状态，由于s'是0-价的，如果让C单独运行直至完成协议，那么它将决定0。令s''是让A和B以相反的次序进行调用所导致的状态。由于s''是1-价的，如果让C单独运行直至完成协议，则它将决定1。问题是这两个可能的寄存器状态$f_A(f_B(v))$和$f_B(f_A(v))$是相同的，所以s'和s''只能在A和B的内部状态中不同。如果现在让C开始执行，由于C不需要与A，B通信即可完成协议，那么这两个状态对C来说是一样的，因此，从这两个状态中不可能决定出不同的值。

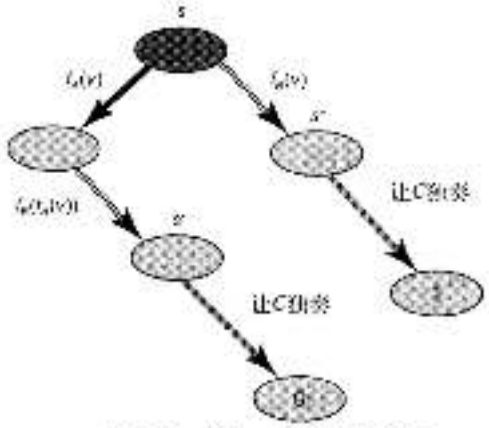


图5-15　场景：两个函数重写

5.8　compareAndSet()操作

现在考虑前面提及的compareAndSet()操作，它是多种现代系统结构（例如，在Intel Pentium™处理器上的CMPXCHG）所支持的一种同步操作。在文献中往往将这种操作称为比较-交换（compare-and-swap）。前面已指出，compareAndSet()将期望值和更新值作为参数。如果寄存器的当前值等于期望值，则用更新值替换；否则，值不变。该方法调用返回一个布尔值以说明值是否被改变。

定理5.8.1　能够支持compareAndSet()和get()方法的寄存器，其一致数是无限的。

证明　图5-16描述了一种采用AtomicInteger类中的compareAndSet()方法，针对n线程0，…，$n-1$的一致性协议，这些线程共享一个AtomicInteger对象，该对象的初始值为常量FIRST，该初始值与任何线程的索引号不相同。每个线程以FIRST作为期望值、以它自己的索引作为新值来调用compareAndSet()。如果某个线程A的调用返回*true*，则这次调用是顺序次序中的第一个，于是A决定它自己的值。否则，A读取AtomicInteger的当前值，从数组proposed[]中获得该值所对应线程的输入。

注意，为了方便起见，在定理5.8.1的compareAndSet()寄存器中提供一个get()方法。下面的推论留作习题。

推论5.8.1　仅支持compareAndSet()方法的寄存器具有无限的一致数。

在第6章将会看到，支持类似于compareAndSet()[⊖]这种原语操作的机器是顺序计算意义

⊖　在一些系统结构提供了get()/compareAndSet()操作对，也称为链接和条件存储。通常，链接可被在加载时对存储单元进行标记，若其他线程修改了该单元，则由于该单元已被标记，条件存储失败。具体参见附录B和第18章。

机的异步计算等价机器：对于任意的并发对象，如果它是可实现的，则必定能在这种机器上以无等待的方式实现。用Maurice Sendak的话来讲，compareAndSet()是“万物之首”。

```
1  class CASConsensus extends ConsensusProtocol {
2    private final int FIRST = -1;
3    private AtomicInteger r = new AtomicInteger(FIRST);
4    public Object decide(Object value) {
5      propose(value);
6      int i = ThreadID.get();
7      if (r.compareAndSet(FIRST, i)) // I won
8        return proposed[i];
9      else                           // I lost
10       return proposed[r.get()];
11   }
12 }
```

图5-16 采用compareAndSwap()的一致性协议

5.9 本章注释

Michael Fischer、Nancy Lynch和Michael Paterson[40]最先证明了在单个线程可以中止的消息传递系统中，不可能实现一致性。在他们的这篇开创性论文中，引入了分布式计算领域中广泛采用的不可能性证明的“二价”形式。M. Loui和H. Abu-Amara[108]以及Herlihy[62]首先将这个结论推广到共享存储器。

117

Clyde Kruskal、Larry Rudolph和Marc Snir[133]将读-改-写操作作为NYU Ultracomputer项目的组成部分。

Maurice Herlihy[62]提出了一致数的概念，将其作为一种衡量计算能力的指标，并第一个证明了本章和第6章中的大部分不可能性结论和通用性结论。

包含常用同步操作原语的Common2类，是由Yehuda Afek、Eytan Weiberger和Hanan Weisman[5]定义的。习题中用到的“粘性位”（sticky-bit）对象则归功于Serge Plotkin[126]。

只有任意一致数n的n-界compareAndSet()对象（习题5.10）则是基于Prasad Jayanti和Sam Toueg[81]所提出的一种构造。在这种层次结构中，如果能用X的任意个数的实例和任意数量的读/写存储器构造一个无等待的一致性协议，则称X解决了一致性问题。Prasad Jayanti[79]指出，在限定只能使用固定个数的X的实例或固定数量的存储器的情形下，可以定义资源-有界的层次结构。由于任何其他的层次结构都是无界层次结构的一种粗粒度形式，所以无界层次结构似乎应是最自然的一种。

Jayanti提出了层次结构的就绪性问题，也就是说，是否可以通过将层次m上的对象X与另一个相同层次或更低层次上的对象Y相结合，将X“提升”到更高的一致性层次上呢？Wai-Kau Lo和Vassos Hadzilacos[106]以及Eric Schenk[144]证明了一致性层次结构不是健壮的：只有一部分对象能够提升。非形式化地看，其构造按照下面的过程来实现：令X是一个具有如下奇特性质的对象，X能解决n线程一致性问题但却“拒绝”泄露结果，除非调用者可以证明他自己能够解决一种中间级别的任务。该任务比n线程一致性弱但又比可通过原子读/写寄存器来解决的任务强。如果Y是一个可用来解决该中间级别任务的对象，那么Y可以通过设法获得X的信任，让X泄露n线程一致性的结果，从而提升X。在这些证明中所使用的对象都是不确定的。

Maurice Sendak的引用来自*Where the Wild Things Are*[140]。

5.10　习题

习题47. 证明引理5.1.2。

习题48. 证明每个n线程一致性协议都有一个二价的初始状态。

习题49. 证明在一个临界状态中，一个后继状态必为0-价的，另一个后继状态为1-价的。

习题50. 证明：若使用原子寄存器的二进制一致性对于双线程是不可能的，则对于n线程也是不可能的，其中$n>2$。（提示：用归纳法证明，如果已经存在一个针对n线程的二进制一致性协议，则可以将该协议转化为一个双线程协议。） 118

习题51. 证明：若采用原子寄存器的二进制一致性对于n线程是不可能的，则对于k值也是不可能的，其中$k>2$。

习题52. 证明使用足够多的n线程二进制一致性对象和原子寄存器能够实现对于n值的n线程一致性协议。

习题53. Stack类提供了两个方法：push(x)把一个值压入栈顶，pop()返回并删除最近入栈的值。证明Stack类的一致数恰好为2。

习题54. 假设为FIFO Queue类增加一个peek()方法，该方法返回但不删除队首元素。证明这种扩展后的队列具有无限一致数。

习题55. 考虑三个线程A、B和C，它们分别有一个MRSW寄存器X_A、X_B和X_C，每个线程可以写自己的寄存器，而其他两个线程则可以读该寄存器。

此外，每一对线程共享一个RMWRegister寄存器，该寄存器只提供一个compareAndSet()方法：A、B共享R_{AB}，B、C共享R_{BC}，A、C共享R_{AC}。只有共享某个寄存器的线程可以调用该寄存器的compareAndSet()方法或读取它的值。

或者给出一个一致性协议并解释该协议为什么能够工作，或者给出一个不可能性证明。

习题56. 考虑习题5.55中所描述的情形，不同的是，A、B和C能够立刻对两个寄存器每次调用compareAndSet()。

习题57. 在5.7节所描述的一致性协议中，如果在出队后通知了该线程的值将会发生什么情况？

习题58. StickyBit类的对象有三种可能的状态：⊥、0、1，初始状态为⊥。A调用write(v)，其中v为0或1，产生如下影响：

- 如果对象状态是⊥，则变为v。
- 如果对象状态是0或1，则保持不变。

对read()的一次调用返回该对象的当前状态。

1. 证明这种对象能够解决对任意数量线程的无等待二进制一致性（即所有输入为0或1）问题。
2. 证明当有m种可能的输入时，一个由$\log_2 m$个StickyBit对象（使用原子寄存器）所组成的数组，能够解决对于任意数量线程的无等待二进制一致性问题。（提示：需要给每个线程指定一个单写者-多读者原子寄存器。）

习题59. 和Consensus类一样，SetAgree类提供了propose()方法和decide()方法，其中每个decide()调用返回一个值，该值是某个线程propose()调用的参数。但不同的是，decide()调用所返回的值并不要求是一致的。相反，这些调用可能返回不超过k个不同的值。（当k为1时，SetAgree和Consensus相同。）当$k>1$时，SetAgree的一致数是多少？ 119

习题60. 对于一个给定的ε，近似一致的双线程类按如下方式定义：给定两个线程A和B，每个线程

可以分别调用decide(x_1)和decide(x_2)，其中x_1和x_2都是实数。这两个方法分别返回y_1和y_2，并使y_1和y_2在闭区间[min(x_1,x_2)，max(x_1,x_2)]内，同时对$\varepsilon>0$，有$|y_1-y_2|<\varepsilon$。注意这个对象是不确定的。

这种近似一致对象的一致数是多少？

习题61. 考虑一个线程之间通过消息传递进行通信的分布式系统。一个A类型的广播能够保证：

1. 每个无故障线程最终能得到每条消息。
2. 如果P先广播M_1随后广播M_2，则每个线程在M_2之前接收到M_1。
3. 但是不同线程广播的消息可以被不同线程以不同的次序接收。

一个B类型的广播能保证：

1. 每个无故障线程最终能得到每条信息。
2. 如果P广播M_1，Q广播M_2，则每个线程以相同的次序收到M_1和M_2。

对每种类型的广播：

- 如果可能，则给出一致性协议。
- 否则，给出不可能性证明。

习题62. 考虑下面的双线程QuasiConsensus（准一致性）问题：

分别对线程A、B给定一个二进制输入。如果两个线程的输入都是v，则它们决定v。如果两个线程的输入是混合的，则要么它们达成一致，要么B决定0且A决定1（但是反之不是这样）。

现有三个可能的习题（只有一个是正确的）。(1) 给出一个双线程一致性协议，使用QuasiConsensus对象证明它的一致性数为2。(2) 给出一个采用临界状态的证明，说明该对象的一致数为1。(3) 给出一个QuasiConsensus对象的读/写实现，从而证明它的一致数为1。

习题63. 解释如果共享对象是一个Consensus（一致）对象，为什么不能用临界状态证明一致性的不可能性。

习题64. 本章已证明了关于双线程一致性协议存在着一个2-价初始状态。试证明对于n线程一致性协议存在着一个2-价初始状态。

120

习题65. 求一致对象提供与Consensus对象相同的propose()和decide()方法。求一致对象可以解决不超过两个不同待选值的一致性问题。（如果有两个以上的待选值，则结果是无定义的。）

说明如何使用组一致对象来解决不超过n个不同输入值的n线程一致性问题。

习题66. 一个三元寄存器具有值⊥、0、1，同时提供通常意义的compareAndSet()方法和get()方法。每个这种寄存器的初始值都为⊥。如果线程的输入为二进制数（0或1），请给出一种只通过一个三元寄存器来解决n线程一致性的协议。

能否使用多个这样的寄存器（也许和原子读/写寄存器一起）来解决n线程一致性问题？其中，线程的输入范围为$0\cdots 2^K-1$，$K>1$。（可以假设一个输入适合一个原子寄存器。）要点：记住一致性协议必须是无等待的。

- 设计一种最多使用$O(n)$个三元寄存器的解决方案。
- 设计一种使用$O(K)$个三元寄存器的解决方案。

可以随意使用原子寄存器（它们很便宜）。

习题67. 前面已定义了无锁特性。证明不存在针对两个或更多个线程且使用读/写寄存器的一致性协议的无锁实现。

习题68. 图5-17描述了一种通过read、write、getAndSet()（即swap）和getAndIncrement()方法实现的FIFO队列。只要不对空队列调用deq()，就可以假设这种队列是可线性化且无等待的。考虑下列陈述。

```
1  class Queue {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicReference items[] =
4      new AtomicReference[Integer.MAX_VALUE];
5    void enq(Object x){
6      int slot = head.getAndIncrement();
7      items[slot] = x;
8    }
9    Object deq() {
10     while (true) {
11       int limit = head.get();
12       for (int i = 0; i < limit; i++) {
13         Object y = items[i].getAndSet(); // swap
14         if (y != null)
15           return y;
16       }
17     }
18   }
19 }
```

图5.17 队列实现

121

- getAndSet()方法和getAndIncrement()方法的一致数都是2。
- 可以通过获取队列的快照（使用前面所学的方法）及返回队首元素的值来增加一个peek()方法。
- 通过使用习题54的协议，可以用结果队列来解决任意的n一致性问题。

我们已通过只使用一致数为2的对象构造了一种n线程一致性协议。指出在这个推理过程中的错误步骤，并解释为什么出错。

习题69. 通过compareAndSet()的定义可以看出，从严格的意义上来讲，compareAndSet()并不是对于$f_{a,b}$的RMW方法，因为RMW方法应该返回寄存器的先前值，而不是布尔值。请用一个支持compareAndSet()和get()方法的对象来构造一个新的对象，该对象具有可线性化的NewCompareAndSet()方法，能返回寄存器的当前值而不是布尔值。

习题70. n-界compareAndSet()对象定义如下：该对象提供一个compareAndSet()方法，它以期望值e和更新值u作为参数。在compareAndSet()的前n次调用中，其行为与传统的compareAndSet()寄存器一样：如果寄存器的值等于e，则用u原子地替换寄存器的值，否则值不变，同时返回一个指明是否发生改变的布尔值。但在compareAndSet()被调用了n次之后，该对象进入一种错误状态，所有后继的方法调用都返回⊥。

证明n-界compareAndSet()对象的一致数恰好为n。

习题71. 用三个compareAndSet()对象（即支持compareAndSet()和get()操作的对象）构造一种双线程三单元的Assign23多赋值对象的无等待实现。

习题72. 在定理5.5.1的证明中，我们曾断言，若给定两个线程和一个(2,3)-赋值对象，则足以证明可以解决2-一致性问题，证明这个断言。

习题73. 证明推论5.8.1。

习题74. 可以把调度器看作一个对手，他能够利用协议及输入值的有关信息来阻止我们达到一致性。战胜对手的一种方法就是采用随机化。假设有两个线程欲达到一致性，每个线程都能不停不停地投掷硬币，这样对于无法控制随后的硬币投掷。

122

假设调度器对于能够观测到每次硬币投掷的结果和每次读/写的值。它能在一次硬币投掷或者一次读/写共享寄存器之前或之后停止一个线程。

随机一致性协议以概率1终止来防备调度器对手。图5-18给出了一个看似合法的随机一致性协议。举例说明该协议不正确。

```
1  Object prefer[2] = {null, null};
2
3  Object decide(Object input) {
4    int i = Thread.getID();
5    int j = 1-i;
6    prefer[i] = input;
7    while (true) {
8      if (prefer[j] == null) {
9        return prefer[i];
10     } else if (prefer[i] == prefer[j]) {
11       return prefer[i];
12     } else {
13       if (flip()) {
14         prefer[i] = prefer[j];
15       }
16     }
17   }
18 }
```

图5-18 这是个随机的一致性协议吗

习题75：可以通过实现一个无死锁或无饥饿的互斥锁的方法来实现一个使用读/写寄存器的一致性对象。然而，这种实现方法只要求提供了相关演进，操作系统必须确保线程没有在临界区内阻塞，从而保证计算作为一个整体进行。

- 对于无障碍情形，非阻塞相关演进条件是否也成立？给出一个仅使用原子寄存器的一致性对象的无障碍实现。
- 在一致性问题的无障碍解决方案中，操作系统扮演什么角色？解释基于临界状态的一致性的不可能性证明方法在哪里会失效，假设让Oracle数据库管理系统不断地暂停线程，以使其他线程能够前进。

123 （提示：考虑如何限制允许的执行集。）

一致性的通用性

6.1 引言

第5章给出了证明“不存在通过Y构造X的无等待实现”这种命题的简单方法。下面考虑具有确定顺序规范的对象类。[1]我们可以构造这样一种对象的层次结构，在这种结构中，无法使用某一层的对象来实现更高层的对象（见图6-1）。这是因为，每个对象都具有一个与之相关的一致数，它是该对象能够解决一致性问题所针对的最大线程个数。而在一个有n个或更多个并发线程的系统中，不可能使用一致数小于n的对象来构造一种一致数为n的对象的无等待实现。该结论也适用于无锁实现。今后除非我们明确说明，否则一个适用于无等待实现的结论也同样适用于无锁的实现。

一致数	对象
1	原子寄存器
2	getAndSet(),getAndAdd(),Queue,Stack
⋮	⋮
m	$[m,m(m+1)/2]$-寄存器实现
⋮	⋮
∞	存储器-存储器迁移,compareAndSet(),链接加载/条件存储[2]

图6-1 同步操作的通用层次结构及其并发可计算性

第5章的不可能性结论并不是指无等待同步是不可能或不可行的。本章将证明存在着通用的对象类：若给定足够多的对象，则对于任何并发对象，可以构造它的无等待可线性化实现。

在一个n线程系统中，当且仅当一个类的一致数大于或等于n时，这个类是通用的。在图6-1中，第n层中的每个类对于一个n线程系统来说是通用的。当且仅当一种机器的系统结构或编程语言能够以通用类的对象作为操作原语时，该机器的系统结构或编程语言具有支持任意无等待同步的计算能力。例如，提供compareAndSet()操作的现代多处理器机器对任意数量的线程都是通用的：它们能以无等待方式实现任何并发对象。

本章主要讲述如何通过一致性对象来构造并发对象的通用构造，并不介绍构造无等待对象的实用技术。和经典的计算理论一样，理解通用构造及其本质含义能够避免去解决那些无法解决的问题。一旦理解了一致性为什么足以实现任何类型的对象，就能通过工程实践使这种构造变得更加有效。

---

① 非确定对象的情形要复杂得多。

② 详见附录B。

6.2 通用性

如果通过类C的一些对象和读/写寄存器能够构造任何对象的无等待实现，则称类C是通用的。在构造中可以使用类C的多个对象，这是因为我们最终感兴趣的是对机器指令可步能力的理解，而大多数机器允许其指令作用到多个存储单元上。在实现中允许使用多个读/写寄存器，这是因为它们便于簿记，而且现代系统结构通常支持大量的存储器。为了避免分散注意力，对所使用的读/写寄存器的数量和一致性对象的个数不作任何限制，而关于存储器的回收问题则留作习题。本章首先给出一种无锁实现，然后对其进行扩展使之变成一种更加复杂的无等待实现。

6.3 一种通用的无锁构造

124
|
126

基于第3章的调用-响应方式，图6-2描述了顺序对象的一般定义。每个对象以固定的初始状态被创建，apply()方法以调用作为参数，它描述了正在被调用的方法及其参数，并返回一个响应，其中包含着该方法调用的终止条件(正常或异常)以及可能的返回值。例如，一个栈调用可以是push()及一个参数，而对应的响应则是正常和空。

```
1  public interface SeqObject {
2    public abstract Response apply(Invoc invoc);
3  }
```

图6-2 通用的顺序对象：apply()方法执行调用并返回一个响应

图6-3和图6-4描述了一种通用构造，能把任何顺序对象转化为可线性化的无锁并发对象。该构造假设顺序对象是确定的：如果调用其一特定状态的对象的方法，则只有一个响应和一种可能的新对象状态。可以将任意的对象看作是处于初始状态的顺序对象和日志的结合：日志是一个由结点组成的链表，描述了对该对象的方法调用序列（即对象的状态转换序列）。线程通过在表头增加一个描述本次调用的新结点来执行一个方法调用。然后，线程从尾到头反向遍历链表，对该对象的私有拷贝执行方法调用。最终，该线程返回只执行了它自己的操作的结果。关键是要理解只有日志头是可变的：初始状态和日志头的前驱结点决不会改变。

如何使这种基于日志的构造变为并发的，即允许线程并发地调用apply()？试图调用apply()的线程创建一个结点来保存它的调用。然后，这些并发线程相互竞争以将它们各自的结点加入到日志头，它们通过运行一个n线程一致性协议，以决定哪一个结点被添加到日志中。该一致性协议的输入是对这些线程结点的引用，而输出则是唯一的获胜结点。

然后，获胜者继续计算它的响应。首先创建该顺序对象的一个局部拷贝，按照next引用从尾到头反向遍历日志，在日志中对它的局部拷贝执行操作，最后返回与它自己的调用相关的响应。该算法即使在并发地调用apply()时也能正常工作，因为日志中直到该线程结点之前的前缀决不会改变。那些没有被一致性对象选中的失败者线程，必须再次尝试把日志头部（在尝试中会变化）的当前结点设置为指向它们。

现在来详细地分析这个构造。图6-4给出了这种通用无锁构造的相关代码。图6-5是该构造的一个执行实例，对象的状态由结点的链表所定义，每个结点包含一个调用。图6-3为结点的代码。结点的decideNext域是一个一致性对象，用来决定下一个被添加到链表中的结点，

next域用于存放一致性协议的结果（对下一个结点的引用）。seq域是结点在链表中的序号。若结点还没有被线程加入链表，则该域的值为0；否则为一个正数值，链表中后继结点的序号每次增加1。初始时，日志中只有一个序号为1的哨兵结点。

```
1  public class Node {
2    public Invoc invoc;                    // method name and args
3    public Consensus<Node> decideNext; // decide next Node in list
4    public Node next;                      // the next node
5    public int seq;                        // sequence number
6    public Node(Invoc invoc) {
7      invoc = invoc;
8      decideNext = new Consensus<Node>();
9      seq = 0;
10   }
11   public static Node max(Node[] array) {
12     Node max = array[0];
13     for (int i = 1; i < array.length; i++)
14       if (max.seq < array[i].seq)
15         max = array[i];
16     return max;
17   }
18 }
```

图6-3 Node类

```
1  public class LFUniversal {
2    private Node[] head;
3    private Node tail;
4    public Universal() {
5      tail = new Node();
6      tail.seq = 1;
7      for (int i = 0; i < n; i++)
8        head[i] = tail
9    }
10   public Response apply(Invoc invoc) {
11     int i = ThreadID.get();
12     Node prefer = new Node(invoc);
13     while (prefer.seq == 0) {
14       Node before = Node.max(head);
15       Node after = before.decideNext.decide(prefer);
16       before.next = after;
17       after.seq = before.seq + 1;
18       head[i] = after;
19     }
20     SeqObject myObject = new SeqObject();
21     Node current = tail.next;
22     while (current != prefer){
23       myObject.apply(current.invoc);
24       current = current.next;
25     }
26     return myObject.apply(current.invoc);
27   }
28 }
```

图6-4 通用的无锁算法

127
|
128

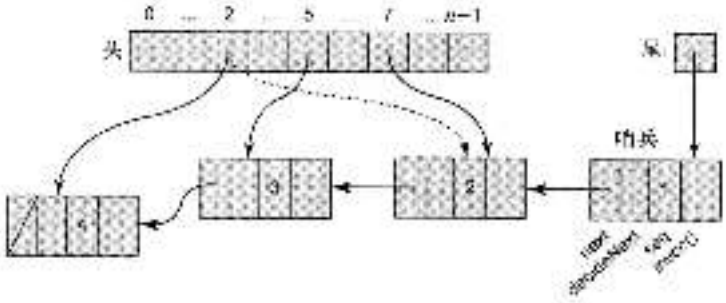


图6-3 通用无锁构造的执行过程。线程2在哨兵结点的decideNext域上赢得一致性协议，把第二个结点添加到日志中。然后将结点的序号从0增为2，并让它在head[]数组中的数据项指向它的结点。线程7在哨兵结点的decideNext域的一致性协议中失败，将next引用和决定的后继结点的序号设置为2（它们已经被线程2设置为相同的值），并让它在head[]数组中的数据项指向该结点。线程5增加第三个结点，修改它的序号为3，并让它在head[]数组中的数据项指向该结点。最后，线程2增加第四个结点，将它的序号置为4，并让它在head[]数组中的数据项指向该结点。head数组中的最大值总是指向日志的头

通用并发无锁构造构设计难点就在于一致性对象只能被使用一次。[⊖]

在图6-4所示的无锁算法中，每个线程分配一个结点来保存它的调用，然后不断地尝试把该结点添加到日志的头部。每个结点有一个decideNext域，该域是一个一致性对象。每个线程将它的结点作为关于头部的decideNext域的一致性协议的输入，来尝试将其结点加入到日志中。由于不参加一致性协议的线程需要反向遍历链表，所以将该一致性协议的结果存放在结点的next域中。多个线程可以同时修改这个域，但它们都写入相同的值。当一个线程的结点被加入到日志时，该线程则设置这个结点的序号。

一旦某个线程的结点成为日志的一部分，它就从日志尾部到最近被加入的结点反向遍历日志，并计算出与其调用相对应的响应。它在这个对象的私有拷贝上执行每个调用，并返回
129 它自己调用的响应。注意，当一个线程计算其响应时，它的所有前驱结点的next引用必定已经被设置，因为这些结点已加入到链表的头部。任何在链表中增加了结点的线程必定已用decideNext一致性协议的结果修改了它的next引用。

如何确定日志的头？由于要对日志头不断地进行修改，同时每个线程只能对一致性对象访问一次，所以不能用一致性对象来记录日志头。取而代之，我们创建一种类似于第2章Bakery算法所使用的针对每一个线程的结构。采用一个n元数组head[]，其中head[i]是线程i已观察到的链表中的最后一个结点。开始时，head[]中的每个项都指向哨兵结点tail。日志的头元素则是head[]数组所指向结点中具有最大序号的结点。图6-3中的max()方法完成了一次收集，读head[]的所有项并返回具有最大序号的结点。

该构造是顺序对象的一种可线性化实现，每个apply()调用能够在一致性协议调用中将结点增加到日志中的时间点被线性化。

⊖ 创建一个可重用的一致性对象，或者创建一个元有决定值是一流的一致性对象，并不是一项简单的任务。其实质与将要设计的通用构造是同一个问题。例如，对于第3章中基于队列的一致性协议，在已经决定之后，如何通过Queue来重复读该一致性对象的状态并不是显而易见的。

这种构造为什么是无锁的？即日志头（最近被加入的结点）是在有限步内被加入到head[]数组中的。因为该结点的前驱必定在head数组中，所以任何正在反复尝试增加新结点的结点都将不断地在head数组上执行max()函数，它检测这个前驱元素，对它的decideNext域调用一致性协议，然后修改获胜结点的域及其序号。最后，它将决定的结点存放在该结点线程的head数组项中。新的头结点最终总会出现在head[]中。这说明一个线程把它自己的结点添加到日志中而又不断失败的唯一可能就是其他线程不断地将自己的结点成功地添加到日志中。因此，只有当其他结点不停地完成它们的调用时，一个结点才会被饿死，从而说明该构造是无锁的。

6.4　一种通用的无等待构造

如何使无锁的算法变成无等待的呢？图6-6给出了完整的无等待算法。我们必须保证每个线程在有限步内完成apply()调用，即线程不会饿死。为了保证这个特性，正在演进的线程应该帮助那些不幸的线程完成它们的调用。这种帮助模式稍后将以一种专用的形式出现在其他无等待算法中。

为了允许帮助，每个线程必须要和其他线程一起共享它正在试图完成的apply()调用。为此，我们增加一个n元数组announce[]，其中announce[i]是线程i正在尝试加入到链表中的结点。开始时，所有的数组项都指向哨兵结点（其序号为1）。当线程i把一个结点存入在announce[i]时，它就通知这个结点。

130

```
1  public class Universal {
2    private Node[] announce; // array added to coordinate helping
3    private Node[] head;
4    private Node tail = new Node(); tail.seq = 1;
5    for (int j=0; j < n; j++){head[j] = tail; announce[j] = tail};
6    public Response apply(Invoc invoc) {
7      int i = ThreadID.get();
8      announce[i] = new Node(invoc);
9      head[i] = Node.max(head);
10     while (announce[i].seq == 0) {
11       Node before = head[i];
12       Node help =  announce[(before.seq + 1) % n];
13       if (help.seq == 0)
14         prefer = help;
15       else
16         prefer = announce[i];
17       Node after = before.decideNext.decide(prefer);
18       before.next = after;
19       after.seq = before.seq + 1;
20       head[i] = after;
21     }
22     SeqObject MyObject = new SeqObject();
23     Node current = tail.next;
24     while (current != announce[i]){
25       MyObject.apply(current.invoc);
26       current = current.next;
27     }
28     head[i] = announce[i];
29     return MyObject.apply(current.invoc);
30   }
31 }
```

图6-6　通用的无等待算法

为了执行apply()，线程首先要通知它的新结点。这一步能够确保如果该线程自己未能成功地把它的结点加入链表，那么其他的某个线程将会按照它自己的立场将那个结点加入链表。然后像以前一样前进，尝试着把该结点加入日志，为此，它对head[]数组只读一次（第9行）就进入算法的主循环，然后一直循环到它自己的结点被加入到链表（在第10行中，当其序号变为非零时被检测到）。下面是对无锁算法的改进。线程首先进行检查，查看数组announce[]中在它前面是否有需要帮助的结点（第12行）。由于结点不断地加入日志中，所以要被帮助的结点必定是动态决定的。线程以递增的次序尝试着去帮助announce[]数组中的结点，该次序由序号对数组announce[]的长度n进行求模来决定。我们将证明这种方法能够保证对于任何自己无法前进的结点，一旦其获胜者线程的索引与最大序号模n的结果值相匹配，最终都能得到其他结点的帮助。如果该帮助步骤被省略掉，那么一个单独的线程有可能被超过任意次。如果被选中进行帮助的结点不需要帮助（在第13行中序号非零），那么每个线程都尝试增加它自己的结点（第16行）。（所有的announce[]数组项被初始化为指向具有非零序号的哨兵结点。）算法的余下部分与无锁算法基本相同。当结点序号变为非零时被加入。在这种情形下，线程和以前一样继续前进，基于日志中从尾到其自己结点的不变部分来计算它的结果。

[131]

图6-7描述了通用无等待构造的一个执行过程。从初始状态开始，线程5通知它的新结点并把该结点加入到日志中，但在将结点加入head[]之前暂停。接下来线程7开始执行。因为before.seq的值1模n+的结果为2，所以线程7尝试帮助线程2。由于线程5已经获胜，所以线程7在对哨兵结点decideNext引用的一致性协议中将成为失败者，因此，将完成线程5的操作，

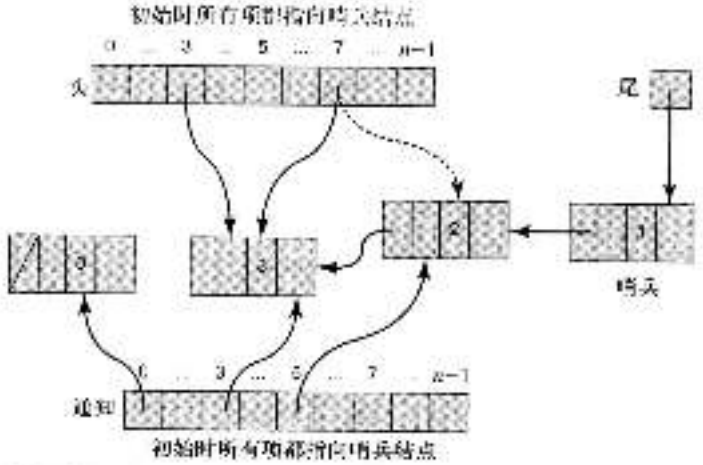


图6-7 通用无等待构造的执行过程。线程5通知它的新结点并把该结点加入到日志中，但在把结点加入数组head[]之前暂停。另一个线程7在数组head[]中不会看到线程5的结点。所以会试帮助线程(before.seq + 1 mod n)，求出该值为2。在帮助线程2时，由于线程5已经获胜，所以线程7在对哨兵结点decideNext引用的一致性协议中成为失败者。因此，线程7将完成修改线程5的结点的域，并将结点的序号设置为2，且到将该结点加入数组head[]中。注意，线程5自己在数组head[]中的项还没有被设置为它所通知的结点。接下来，线程2通知它的结点。同时线程7在将线程2的结点加入中获得成功。从而把线程2的结点序号设置为3。现在线程2醒来，由于它的结点序号不为零，所以不进入主循环，但会在第28行继续修改head[]，并使用顺序对象的一个拷贝来计算其输出值

把结点序号设为2并将线程5的结点加入数组head[]中。现在设想线程2立即通知它的结点。线 132
程7则成功地加入线程2的结点，但在把线程2的结点序号设为3并准备把它加入head[]之前再次暂停。现在线程2继续。由于它的结点序号不为零，所以不进入主循环，但会在第28行继续修改head[]，并使用顺序对象的一个拷贝来计算其输出值。

对无锁算法的这些修改中有一个精妙之处。由于不止一个线程试图把一个特定的结点加入日志中，因此必须保证结点不能被两次加入日志。一个线程可能正在添加结点，设置结点的序号，而与此同时，另一个线程可能已增加了同一结点并设置了序号。算法应避免这种由于线程读数组head[]的最大值和数组announce[]中结点序号的次序所带来的错误。设a是由线程A所创建并被线程A和B添加到日志中的结点。在第二次被添加之前，该结点已至少一次被加入到head[]中。但要注意，由B从head[A]中读出的before结点（第11行）一定是a本身或者a在日志中的后继结点。况且，在任何结点被加入到head[]之前（第20行或第28行），其序号已被设为非零（第19行）。操作的次序确保B在第9行或第20行设置它的head[B]项（基于该项来设置B的before变量，从而导致一个错误的增加）。只有这时，才能确认a的序号在第10和13行为非零（取决于是A还是另一个线程执行该操作）。由此可见，对错误的第二次增加的验证将会失败，因为结点a的序号已经为非零，它不会被第二次加入到日志中。

因为结点不会被二次加入日志中，而且结点加入日志的次序显然与对应方法调用的偏序次序相一致，所以保证了可线性化性。

为了证明算法是无等待的，需要证明这种帮助机制能够保证任何被通知的结点最终会加入到数组head[]中（意味着它在日志中），并且发出通知的线程能够完成对其结点的计算。为便于证明，首先定义一些符号。令max(head[])是数组head[]中序号最大的结点，"c∈head[]"则表示对于某个i，结点c已被赋予head[i]。

辅助变量（有时称作幻影变量）是在代码中没有显式出现的量，它不以任何方式影响程序的行为，但有助于我们推理算法的行为。下面是一些常用的辅助变量：

- *concur*(A)是自线程A最后一次通知后，被存放在数组head[]中的结点的集合。
- *start*(A)是在线程A最后一次通知时，结点max(head[])的序号。 133

图6-8给出了反映辅助变量以及它们是如何被修改的程序代码段。例如，语句

$$(\forall j)\mathit{concur}(j)=\mathit{concur}(j)\cup\mathit{after}$$

表示对于所有的线程j，把结点*after*加入到*concur*(j)中。尖括号中的代码语句被看作是原子执行的。由于辅助变量不会以任何方式影响计算，所以可以假定这种原子性。为简单起见，可以让作用于结点或结点数组的函数max()返回它们序号中的最大值。

注意，下面的性质在通用算法的整个执行过程中是不变的：

$$|\mathit{concur}(A)|+\mathit{start}(A)=\max(\text{head}[]) \tag{6.4.1}$$

引理6.4.1　对于所有的线程A，下述断言总是成立的：

$$|\mathit{concur}(A)|>n \Rightarrow \text{announce}[A]\in\text{head}[]$$

证明　如果|*concur*(A)|>n，则*concur*(A)中包含连续的结点b和c（由线程B和C加入到日志中）。它们各自的序号加上1模n分别等于A−1和A（注意，b和c是由线程B和C添加到日志中的结点，但并不要求是由B和C通知的结点）。根据第12到16行的代码，线程C将把A在announce[]中的数组项所确定的结点增加到日志中。现在需要证明当它这样做时，announce[A]已被通知

了，所以c将加入announce[A]或者announce[A]已经被加入。随后，当c被加入head[]且$|concur(A)|>n$时，announce[A]将会被引理所要求的那样在head[]中。

```
1  public class Universal {
2    private Node[] announce;
3    private Node[] head;
4    private Node tail = new Node(); tail.seq = 1;
5    for (int j=0; j < n; j++){head[j] = tail; announce[j] = tail};
6    public Response apply(Invoc invoc) {
7      int i = ThreadID.get();
8      <announce[i] = new Node(invoc); start(i) = max(head);>
9      head[i] = Node.max(head);
10     while (announce[i].seq == 0) {
11       Node before = head[i];
12       Node help =  announce[(before.seq + 1) % n];
13       if (help.seq == 0)
14         prefer = help;
15       else
16         prefer = announce[i];
17       Node after = before.decideNext.decide(prefer);
18       before.next = after;
19       after.seq = before.seq + 1;
20       <head[i] = after; (∀j) (concur[j] = concur[j]∪{after})>
21     }
22     SeqObject MyObject = new SeqObject();
23     Node current = tail.next;
24     while (current != announce[i]){
25       MyObject.apply(current.invoc);
26       current = current.next;
27     }
28    <head[i] = announce[i]; (∀j) (concur[j] = concur[j]∪{after})>
29    return MyObject.apply(current.invoc);
30   }
31 }
```

图6-8 采用辅助变量的通用无等待算法。假设尖括号中的操作是原子发生的

要弄清楚为什么当C运行到第12到16行代码时，announce[A]已经被通知，需要注意以下几点：(1) 因为C已经把它的结点c加入到b，所以它在第11行必定会把b读作*before*结点，这意味着在第11行C从head[]中读取b之前B已增加了b；(2) 由于b在$concur(A)$中，所以A在b被加入到head[]之前就已通知了。根据传递性，从(1)和(2)可得出C执行第12到16行之前A已经通知了，所以命题成立。□

引理6.4.1对方法调用时可以增加的结点个数做了限制。下面给出一系列引理来说明当A结束扫描数组head[]时，或者announce[A]被加入，或者head[A]在表尾的$n+1$个结点中。

引理6.4.2 下面的性质总是成立的：

$$\max(head[]) \gg start(A)$$

证明 head[i]的序号是非递减的。□

引理6.4.3 下面是图6-3中第13行的循环不变量（指循环的每次迭代中都保持不变）：

$$\max(head[A],head[j],\cdots,head[n]-1) \gg start(A)$$

其中，j为循环索引。

换句话说，head[A]以及从当前值j到循环结束时所有head[]项的最大序号决不会小于A通

加时数组中的最大值。

证明　若j为0，则引理6.4.2隐含说明该命题成立。当head[A]被序号为max(head[A].seq, head[j])的结点替代时，在每次迭代过程中命题都成立。□

引理6.4.4　下面的命题只在第10行以前成立：

$$\text{head}[A].seq \geq start(A)$$

证明　注意只有在第20行或28行，head[A]才会被设置为指向A的最后增加的结点。因此，在第9行执行了$Node.max()$调用之后，max(head[A].seq, head[0].seq, …, head[n−1].seq)只能为head[A].seq，由引理6.4.3可知命题成立。□

引理6.4.5　下述性质总是成立的：

$$|concur(A)| \geq \text{head}[A].seq - start(A) \geq 0$$

证明　下界可由引理6.4.4推出，上界可由等式（6.4.1）推出。□

定理6.4.1　图6-6中的算法是正确的并且是无等待的。

证明　要证明算法是无等待的，需注意A执行主循环不超过$n+1$次。在每次成功的迭代中，head[A].seq增加1，在$n+1$次迭代后，由引理6.4.5可得：

$$|concur(A)| \geq \text{head}[A].seq - start(A) \geq n$$

由引理6.4.1可知announce[A]必定已被加入到head[]中。□

6.5　本章注释

本章描述的通用构造源于Maurice Herlihy在1991年发表的论文[62]。为一种采用链接加载/条件存储的通用无锁构造则出自文献[60]。这种构造的复杂度可以通过多种方式进行改进。Yehuda Afek、Dalia Dauber和Dan Touitou[3]描述了如何提高时间复杂度使其依赖于并发线程的个数而不是可能的最大线程个数。Mark Moir[118]给出了无需拷贝整个对象的无锁且无等待的构造。James Anderson和Mark Moir[11]对这种构造进行了扩展，允许多个对象被修改。Prasad Jayanti[80]证明了任何通用构造在最坏情况下的复杂度为$\Omega(n)$，其中n是最大线程个数。Tushar Chandra、Prasad Jayanti和King Tan[26]则给出了许多对象，指出对这些对象存在更有效的通用构造。

6.6　习题

习题76. 举例说明具有不确定顺序规范的对象其通用构造可能会失败。

习题77. 给出一种解决方法，使通用构造能适于具有不确定顺序规范的对象。

习题78. 在无锁和无等待的通用构造中，若tail[]的哨兵结点的序号被初始化为1，如果哨兵结点的序号被初始化为0，这两个算法中的哪一个（如果有的话）将会出错？

习题79. 不采用通用构造而只使用一致性协议来实现一种具有read()和compareAndSet()方法的可线性化无等待寄存器。说明如何实现这个算法。

习题80. 在本章的构造中，每个线程首先查找另一个线程进行帮助，然后再尝试加入它自己的结点。

假设每个线程首先尝试加入它自己的结点，然后再去帮助其他的线程。解释这个方法是否可行。证明你的结论。

习题81. 在区6-4的构造中，我们使用“头”引用（指向试图修改其decideNext域的结点）的“分布式”实现来避免创建允许重复一致性的对象。请用一种无需head引用的实现来替换这种实现，通过从开始向下追踪日志，直到到达一个序号为0或者具有最大非0序号的结点来找出下一个“头”。

习题82. 对无锁协议进行修改，让一个线程在第28行把它最新加入的结点添加到数组*head*中，即使该结点在第20行已被加入了。这一步是必需的，因为和无锁协议不同，该线程的结点在第20行有可能已被另一个线程加入，而那个“帮助”线程恰好在第20行停止，此刻该结点的序号已被修改但数组*head*还未被修改。

1. 解释为什么删除第28行会违背引理6.4.4。

137 2. 该算法还能正常工作吗？

习题83. 给出一种能使通用构造适于有界数量的存储器的解决办法，也就是说，能适于有限个数的一致性对象和有限个数的读/写寄存器。

提示：在结点中增加一个before域，并在代码里构建一种存储器回收方案。

习题84. 实现一种一致性对象，每个线程可以通过调用read()和compareAndSet()方法多次地访问该对象，即构建一种“多路存取”的一致性对象。不允许使用通用构造。

138

实　践

第7章　自旋锁与争用
第8章　管程和阻塞同步
第9章　链表：锁的作用
第10章　并行队列和ABA问题
第11章　并发栈和消除
第12章　计数、排序和分布式协作
第13章　并发哈希和固有并行
第14章　跳表和平衡查找
第15章　优先级队列
第16章　异步执行、调度和工作分配
第17章　障碍
第18章　事务内存

第7章

The Art of Multiprocessor Programming (Revised First Edition)

自旋锁与争用

在单处理器上编写程序时，通常不用考虑系统底层系统结构的细节。然而不幸的是，对多处理器的编程目前还不能做到这点，对机器底层系统结构的理解在多核编程中仍起着至关重要的作用。本章的目的就是理解系统结构对系统性能会产生什么样的影响，以及如何利用这些知识来编写高效的并发程序。本章对已熟悉的互斥问题重新进行研究，其目的在于设计出适于多处理器的互斥协议。

任何互斥协议都会产生这样的问题：如果不能获得锁，应该怎么做？对此有两种选择。一种方案是让其继续进行尝试，这种锁称为自旋锁，对锁的反复测试过程称为旋转或忙等待。Filter和Bakery算法都属于自旋锁。在希望锁延迟较短的情形下，选择旋转的方式比较合乎情理，显然，只有在多处理器中旋转才有实际意义。另一种方案就是挂起自己，请求操作系统调度器在处理器上调度另外一个线程，这种方式称为阻塞。由于从一个线程切换到另一个线程的代价比较大，所以只有在允许锁延迟较长的情形下，阻塞才有意义。许多操作系统将这两种策略综合起来使用，先旋转一个小的时间段然后再阻塞，旋转和阻塞都是重要的技术。本章着重研究采用旋转技术的锁。

7.1 实际问题

139
↓
141

本章采用java.util.concurrent.locks包中的Lock接口来解决实际的互斥问题，我们只考虑两个最重要的方法：lock()和unlock()。这两个方法通常按照下面这种结构化的方式来使用：

```
1  Lock mutex = new LockImpl(...); // lock implementation
2  ...
3  mutex.lock();
4  try {
5    ...           // body
6    } finally {
7      mutex.unlock();
8    }
```

首先创建一个新的Lock对象mutex（第1行）。由于Lock只是一个接口而并非一个类，所以不能直接创建Lock对象，为此，需要先构建一个对象来实现Lock接口。（java.util.concurrent.locks包中已包含一些实现Lock的类，本章将提供另外一些实现Lock的类。）接下来在第3行获得锁，然后进入临界区，即第4行的try块。第6行的finally块将确保只有在控制离开临界区时才能释放锁。对lock()的调用不允许放在try块内，因为在获得锁之前lock()调用可能会抛出一个异常，这将导致在实际上没有获得锁的情形下finally块调用了unlock()。

为什么不用第2章介绍的算法（例如Filter或Bakery）来实现高效的Lock呢？其原因之一就

是第2章已证明的空间下限：采用读/写方式的互斥所需的空间与n呈线性关系，其中n指可能访问存储单元的线程数。这将使情况变得很糟糕。

例如，考虑第2章的双线程Peterson锁算法，如图7-1所示。有两个线程A和B，其ID分别为0或1。若线程A要获得锁，则将flag[A]置为*true*，将victim置为A，然后测试victim和flag[1−A]。若测试失败，则线程A旋转并重复测试。一旦测试成功，线程A则进入临界区，在它离开时将flag[A]设为*false*。由第2章可知，Peterson锁支持无饥饿互斥。

```
1  class Peterson implements Lock {
2    private boolean[] flag = new boolean[2];
3    private int victim;
4    public void lock() {
5      int i = ThreadID.get(); // either 0 or 1
6      int j = 1-i;
7      flag[i] = true;
8      victim = i;
9      while (flag[j] && victim == i) {}; // spin
10   }
11 }
```

图7-1 Peterson类（第2章）：第7、8和9行的读/写次序对于锁成互斥至关重要

[142]

假设要编写一个简单的并发程序，该程序的两个线程反复地获得Peterson锁，并将共享计数器加1，最后释放锁。在一台多处理器机器上运行这个程序，每个线程执行“获得—增加—释放”循环50万次。在大多数现代系统结构中，线程很快就结束了。然而令人不可思议的是，该计数器的最终值与我们所期望的100万次稍微有些出入。就其比例而言，这个错误或许是很小的，但是为什么会存在错误呢？不管怎样，必定存在两个线程在同一时刻都进入临界区的情形，即使已证明这种情形是不可能发生的。让我们引用Sherlock Holmes所讲的：

> 我已说过多少次？若消除所有不可能的，无论它们是多么不可能，所剩下的必定都是事实。

一定是我们的证明错了，不是在逻辑上有什么错误，而是对现实世界的假设存在错误。

在多处理器编程中，很自然会假设读/写操作是原子的，也就是说，它们可以被线性化为某种顺序的执行，或者至少应是顺序一致的。（可线性化性意味着顺序一致性。）正如第3章所讲的，顺序一致性意味着存在某个在所有操作上的全局次序，在这个次序中，每个线程的操作都按照它自己的程序所规定的次序生效。在证明Peterson锁的正确性时，我们假设存储器是顺序一致的，而没有考虑上述因素。特别要注意的是，互斥与图7-1中第7、8和9行的操作次序相关。而在证明Peterson锁具有互斥特性时，隐含地基于这样的假设：同一线程对内存的任意两次访问，即使是对不同的变量，也都是按照程序顺序生效的。（具体地说，B对flag[B]的写在B对victim的写之前已生效（公式（2.3.9）），A对victim的写在A对flag[B]的读之前已生效（公式（2.3.11））。这两点非常关键。）

然而不幸的是，现代的多处理器通常不提供顺序一致的存储器，因此，对于给定的线程，并不能保证读/写操作的程序次序。

为什么不支持这些特性呢？第一个原因在于编译器，为了提高性能，编译器要对指令进行重排序。大多数程序设计语言都能保证单个变量的程序次序，但在多个变量之间却并非如

此。因此，编译器有可能把线程*B*对flag[*B*]的写和对victim的写颠倒顺序，从而使得公式（2.3.9）无效。第二个原因则生于多处理器硬件本身。（附录B对本章的多处理器系统结构问题进行了全面的讨论。）硬件供应商公开表示对多处理器存储器的写并不一定在写操作执行时生效，因为在大多数程序中，并不要求写操作在共享存储器中立即生效。在多处理器系统结构中，对共享存储器的写往往被缓存在一个特殊的写缓冲区中（有时称为存储缓冲区），只有在需要时才写入内存。如果线程*A*对victim的写在一个写缓冲区中被延迟，那么这个值有可能
143 在*A*读了flag[*B*]之后才到达内存，从而使公式（2.3.11）无效。

在如此弱的存储器一致性保证下，如何对多处理器进行编程呢？为了防止写缓冲所带来的操作重排序，现代系统结构提供了专门的内存路障指令（有时称为内存栅栏），以迫使未完成的指令强行生效。而在哪里插入内存故障则是程序员的责任（例如，可以通过在每次读之前放置一个栅栏来固定Peterson锁）。毫无疑问，内存路障的代价是非常昂贵的，大致上与原子的compareAndSet()指令相同，因此建议尽量不要使用。而事实上在大多数系统结构中，像getAndSet()或compareAndSet()这样的一些同步指令，在对volatile域进行读/写操作时，都包含一个内存路障。

若路障和同步指令的代价是一样的，那么可以直接使用getAndSet()和compareAndSet()这种操作来设计互斥算法。这些操作要比reads和writes具有更高的一致数，可以直接使用这些操作来对谁能进入临界区的问题达成一致。

7.2 测试-设置锁

一致数为2的testAndSet()操作是大多数早期的多处理器系统结构所提供的主要同步指令，该指令对单个的存储字（或字节）进行操作，字是一个二进制值，要么为*true*要么为*false*。testAndSet()操作将*true*值原子地存入字中，并返回这个字的先前值，即用值*true*来交换字的当前值。初看起来，这条指令似乎非常适合自旋锁。当字的值为*false*时锁空闲，为*true*时锁忙。lock()方法对存储单元反复地调用testAndSet()，直到指令返回*false*为止（即锁为空闲）。unlock()方法则简单地将*false*值写入存储单元。

java.util.concurrent包具有一个用于存放布尔值的AtomicBoolean类。它提供了用值*b*替换被存储值的set(*b*)方法，以及用值*b*原子地替换当前值并返回先前值的getAndSet(*b*)方法。传统的testAndSet()指令就如同是对getAndSet(*true*)的一次调用。使用术语测试-设置是为了与习惯用法保持一致，但本书的例子使用了表达式getAndSet(*true*)，其目的是和Java相兼容。图7-2中的TASLock类描述了一个基于testAndSet()指令的锁算法。

```
1  public class TASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (state.getAndSet(true)) {}
5    }
6    public void unlock() {
7      state.set(false);
8    }
9  }
```

图7-2 TASLock类

下面考虑另外一种TASLock算法，如图7-3所示。该算法并没有直接调用testAndSet()，

而是由线程反复地读锁直到该锁看起来是空闲的（即直到get()返回*false*）。只有在锁看似为空闲时，线程才能使用testAndSet()。这种技术称为测试-测试-设置，这种锁称为TTASLock。

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {};
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13 }
```

清单7.3　TTASLock类

从正确性的角度来看，TASLock和TTASLock算法是等价的：每一个算法都保证了无死锁的互斥。在目前所使用的简单模型中，这两种算法之间应该没有什么不同。

在实际的多处理器上进行比较将会有怎样的结果呢？图7.4是n个线程固定地执行一段临界区所需时间的实测结果。每个数据点代表若干相同的工作量。在没有争用影响的情形下，整个曲线将是平直的。最上面是TASLock的曲线，中间是TTASLock的曲线，最下面的曲线表示线程在没有干扰的情况下所需的时间。显然，二者之间的差异非常显著：TASLock的性能最差；TTASLock的性能则要好一些，但与理想情形仍然相距甚远。

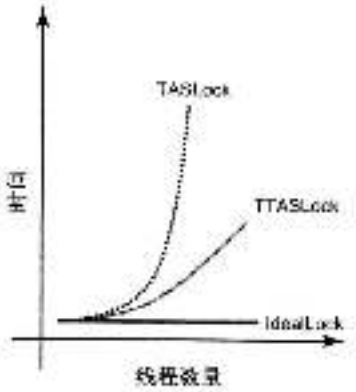


图7-4　n个线程固定地执行一段临界区所需的时间

可以用现代多处理器系统结构来解释这些差异。首先，要注意现代多处理器中包含多种形式的系统结构，因此不能过于抽象概括。但是，几乎所有的现代系统结构都存在着高速缓存和局部性的问题，虽然在细节上有所不同，但其原理却是相同的。

为简单起见，考虑一种典型的多处理器系统结构，其中处理器之间是通过一种称为总线（类似一个微型以太网）的共享广播媒介进行通信的。处理器和存储控制器都可以在总线上广播，但在一个时刻只能有一个处理器（或存储控制器）在总线上广播。所有的处理器（存储控制器）都可以监听。尽管基于总线的系统结构在处理器数量很多的情形下可扩展性很差，但这种系统结构在今天非常普遍，其原因在于它们易于构建。

每个处理器都有一个cache，它是一种高速的小容量存储器，用来存放处理器感兴趣的数据。对内存的访问通常要比对cache的访问多出几个数量级的机器周期。目前技术的发展对此问题的解决效果并不理想：内存访问时间在近期内不太可能赶上处理器的时间周期，因此cache的性能对于多处理器系统结构的整体性能具有至关重要的影响。

当处理器从内存地址中读数据时，首先检查该地址及其所存储的数据是否已在它的cache中。如果在cache中，那么处理器产生一个cache命中，并可以立即加载这个值；如果不在，则产生一个cache缺失，且必须在内存或为一个处理器的cache中查找这个数据。接着，处理器在总线上广播这个地址，其他的处理器监听总线。如果其中处理器在自己的cache中发现这个地址，则广播该地址及其值来做出响应。如果所有处理器中都没有发现此地址，则以内存中该地址所对应的值来进行响应。

7.3 再论基于TAS的自旋锁

首先分析在共享总线系统结构中TTASLock算法是怎样执行的。每个getAndSet()调用实质上是总线上的一个广播。由于所有线程都必须通过总线和内存进行通信，所以getAndSet()
141 调用将会延迟所有的线程，包括那些没有等待锁的线程。更为糟糕的是，getAndSet()调用能
146 够迫使其他的处理器丢弃它们自己cache中的锁副本，这样每一个正在自旋的线程几乎每次都会遇到一个cache缺失，并且必须通过总线来获取新的没有被修改的值。而比这更为糟糕的是，当持有锁的线程试图释放锁时，由于总线被正在自旋的线程所独占，该线程有可能会被延迟。现在可以理解为什么TASLock的性能如此之差。

下面分析当锁被线程*A*持有时TTASLock算法的执行行为。线程*B*第一次读锁时发生cache缺失，从而匪率等待值被载入它的cache中。只要*A*持有锁，*B*就不断地重读该值，且每次都命中cache，这样，*B*不产生总线流量，而且也不会降低其他线程的内存访问速度，此外，释放锁的线程也不会被正在该锁上旋转的线程所延迟。

然而，当锁被释放时情况却并不理想，锁的持有者将*false*值写入锁变量来释放锁，该操作将会使自旋线程的cache副本立刻失效。每个线程都将发生一次cache缺失并重读新值，它们都（几乎是同时）调用getAndSet()以获取锁。第一个成功的线程将使其他线程失效，这些失效线程接下来又重读那个值，从而引起一场总线流量风暴，最终，所有线程再次平静，进入本地旋转。

本地旋转指线程反复地重读被缓存的值而不是反复地使用总线，这个概念是一个重要的原则，对设计高效的自旋锁非常关键。

7.4 指数后退

现在考虑如何改进TTASLock算法。首先介绍一些专业术语：争用指多个线程试图同时获取一个锁；高争用则意味着存在大量正在争用的线程；低争用的意思与高争用相反。

在TTASLock类中，lock()方法使用了两个步骤：它不断地读锁，当锁看似空闲时，则调用getAndSet(*true*)来获取锁。下面是一个重要的结论：如果其他的某个线程在第一步和第二步之间获得了锁，那么该锁极有可能存在高争用。显然，试图获得一个存在高争用的锁是一种应该回避的情形。此时线程获得锁的机会非常小，为此这种尝试将会导致总线流量的增加（导致流量细率）。相反，若让线程后退一段时间，给正在竞争的线程以结束的机会，将会更加有效。

线程在重试之前应该后退多久呢？一种好的准则就是不成功尝试的次数越多，发生争用的可能性就越高，线程需要后退的时间就应越长。下面是一种简单的方法，每当线程发现锁
147 变为空闲但却无法获得它时，就在重试之前后退。为了确保发生冲突的并发线程不进入锁步，即在同一时刻所有线程都试图获得锁，该线程应随机地后退一段时间。每当线程试图获得一个锁但又失败以后，则将后退时间加倍，直到一个固定的最大值maxDelay为止。

对于一些锁算法来说后退是一种常用的方法，因此我们将这种逻辑封装到一个简单的Backoff类中，如图7-5所示。在构造函数中使用了下面这些参数：minDelay是最小的初始时延（线程后退一段太短的时间是没有意义的），maxDelay是最终的最大时延（为了避免不幸的线程后退太长的时间，最终的限制是必需的），limit域则控制着当前的时延限制。backoff()方法在0和当前限制之间选择一个随机的时延，在返回之前以这个时延来阻塞线程。下一次后退时把这个限制加倍，直到maxDelay为止。

```
1  public class Backoff {
2    final int minDelay, maxDelay;
3    int limit;
4    final Random random;
5    public Backoff(int min, int max) {
6      minDelay = min;
7      maxDelay = max;
8      limit = minDelay;
9      random = new Random();
10   }
11   public void backoff() throws InterruptedException {
12     int delay = random.nextInt(limit);
13     limit = Math.min(maxDelay, 2 * limit);
14     Thread.sleep(delay);
15   }
16 }
```

图7.5　Backoff类：自适应的后退逻辑。为保证争用的并发线程在同一时刻不会反复地尝试获得锁，让线程后退一个随机的时间间隔。每次线程尝试得到一个锁并失败后，就把期望的后退时间加倍，直到到达一个固定的最大值

图7-6描述了BackoffLock类。该类使用了Backoff对象，该对象的最大和最小后退时间存于常量minDelay和maxDelay中。关键要注意，只有当线程发现一个锁为空闲且不能立即获得该锁时才会后退，或者到锁被另一个线程所持有并不能够说明争用的程度。

```
1  public class BackoffLock implements Lock {
2    private AtomicBoolean state = new AtomicBoolean(false);
3    private static final int MIN_DELAY = ...;
4    private static final int MAX_DELAY = ...;
5    public void lock() {
6      Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
7      while (true) {
8        while (state.get()) {};
9        if (!state.getAndSet(true)) {
10         return;
11       } else {
12         backoff.backoff();
13       }
14     }
15   }
16   public void unlock() {
17     state.set(false);
18   }
19   ...
20 }
```

图7-6　指数后退锁。每当线程未能获得已空闲的锁时，就在重试之前后退

BackoffLock易于实现，且在许多系统结构中其性能要比TASLock好得多。然而，它的性能与常量minDelay和maxDelay的选取密切相关。为了在一个特定的系统结构中部署该锁，要对不同的值进行测试，选择性能最好的值。实验表明，最优值与处理器的个数以及它们的速度密切相关，因此，很难调整BackoffLock类以使它与各种不同的机器相互兼容。

7.5 队列锁

下面给出另一种实现可扩展自旋锁的方法，这种实现比后退锁稍复杂一些，但却具有更好的可移植性。在BackoffLock算法中有两个问题。

- cache一致性流量：所有线程都在同一个共享存储单元上旋转，每一次成功的锁访问都会产生cache一致性流量（尽管比TASLock低）。
- 临界区利用率低：线程延迟过长，导致临界区利用率低下。

可以将线程组织成一个队列来克服这些缺点。在队列中，每个线程检测其前驱线程是否已完成来判断是否轮到它自己。让每个线程在不同的存储单元上旋转，从而降低cache一致性流量。
148 – 149 队列还提高了临界区的利用率，因为没有必要去判断何时要访问它：每个线程直接由队列中的前驱线程来通知。最后，队列提供先来先服务的公平性，可获得与Bakery算法同样的高级别公平性。下面探讨关于队列锁的各种不同的实现方法，它们都是基于上述队列观点的锁算法。

7.5.1 基于数组的锁

图7-7和图7-8描述了一种基于数组的简单队列锁ALock[9]。AtomicInteger的tail域被所

```
1  public class Alock implements Lock {
2    ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> (){
3      protected Integer initialValue() {
4        return 0;
5      }
6    };
7    AtomicInteger tail;
8    volatile boolean[] flag;
9    int size;
10   public ALock(int capacity) {
11     size = capacity;
12     tail = new AtomicInteger(0);
13     flag = new boolean[capacity];
14     flag[0] = true;
15   }
16   public void lock() {
17     int slot = tail.getAndIncrement() % size;
18     mySlotIndex.set(slot);
19     while (! flag[slot]) {};
20   }
21   public void unlock() {
22     int slot = mySlotIndex.get();
23     flag[slot] = false;
24     flag[(slot + 1) % size] = true;
25   }
26 }
```

图7-7 基于数组的队列锁

[9] 大多数锁类都以其实现者名字的首字母命名，详见7.10节解释。

有的线程所共享，其初始值为0。为了获得锁，每个线程原子地增加tail域（第17行）。所得的结果值称为线程的槽。槽则被当作布尔数组flag的索引，如果flag[j]为*true*，那么槽为*j*的线程有权获得锁。在初始状态时，flag[0]为*true*。为了获取锁，线程不断地旋转直到它的槽所对应的flag变为*true*（第19行）。而在释放锁时，线程把对应于它自己槽的flag设为*false*（第23行），并将下一个槽的flag设为*true*（第24行）。所有的算术运算都对*n*进行求模，其中*n*至少应与最大的并发线程数相同。

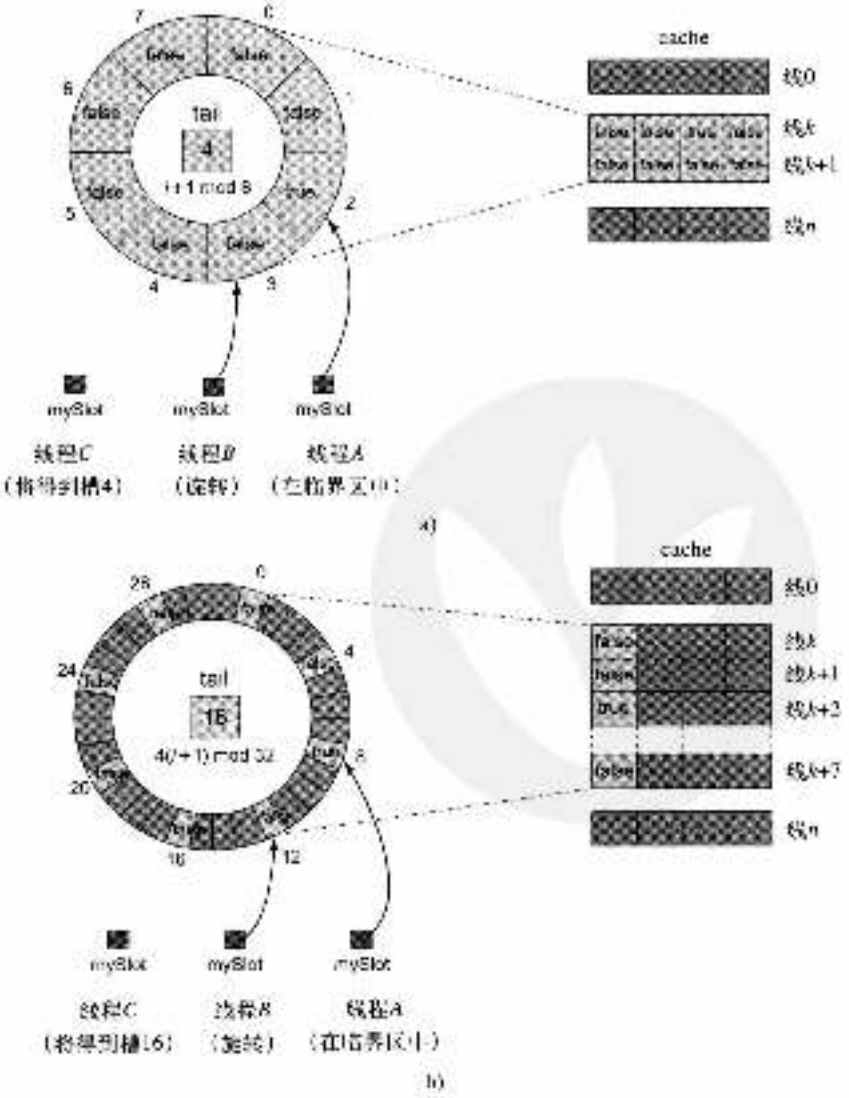


图7-8 使用填补以避免出现假共享的ALock。在a中，ALock有8个槽，通过一个模8的计数器来访问。数组项通常被连续地映射到cache线中。可以看出当线程A改变其数组项的状态时，其数组项被映射到同一个cache线k内的线程B将会产生一个假无效。在b中，每个存储单元被填补了，因此它采用一个模32的计数器与其他的4字线区分开。即使数组项被连续地映射，B的数组项也被映射到与A的数组项不同的cache线中。这样，若A使它的数组项无效，并不会导致B也无效

在ALock算法中，mySlotIndex是线程的局部变量（见附录A）。线程的局部变量与线程的常规变量不同，对于每个局部变量，线程都有它自己独立初始化的副本。局部变量不需要保存在共享存储器中，不需要同步，也不会产生任何一致性流量，因为它们只能被一个线程访问。通常使用get()和set()方法来访问局部变量的值。

数组flag[]是被多个线程所共享的。但在任意给定的时间，由于每个线程都是在一个数组存储单元的本地cache副本上旋转，大大降低了无效流量，从而使得对数组存储单元的争用达到最小。

要注意争用仍有可能发生，其原因在于存在着一种称为假共享的现象。当相邻的数据项（如数组元素）共享单一cache线时会发生这种现象。对一个数据项的写将会使该数据项的cache线无效，对于那些恰好进入同一个cache线的未改变但很接近的数据项来说，这种写将会引起正在这些数据上进行旋转的处理器的无效流量。在图7-8的例子中，访问8个ALock存储单元的线程有可能遇到不必要的无效，因为这些存储单元已被缓存到两个同样的4字线中。避免假共享的一种方法就是填补数组元素，以使不同的元素被映射到不同的cache线中。在类似于C或者C++的低级语言中填补是很容易实现的，在这些语言中程序员可以直接控制对象在存储器中的布局。在图7-8的例子中，可以通过将锁数组的大小增加为原来的4倍，并以4个字来将开存放存储单元以使得两个存储单元不会落在同一个cache线中，来填补最初的8个ALock存储单元。（通过计算4(i+1) mod 32而不是i+1 mod 8来从单元i增加到下一个单元。）

7.5.2 CLH队列锁

ALock是对BackoffLock的改进，其为它将无效性降到最低并把一个线程释放锁和另一个线程获得该锁之间的时间间隔最小化。与TASLock和BackoffLock不同，该算法能够确保无饥饿性，同时也保证了先来先服务的公平性。

然而，ALock锁并不是空间有效的。它要求并发线程的最大个数为一个已知的界限n，同时为每个锁分配一个与该界限大小相等的数组。因此，即使一个线程每次只访问一个锁，同步L个不同对象也需要$O(Ln)$大小的空间。

现在来分析另一种不同类型的队列锁。图7-9描述了CLHLock类的域、构造函数及QNode类。

```
1  public class CLHLock implements Lock {
2    AtomicReference<QNode> tail;
3    ThreadLocal<QNode> myPred;
4    ThreadLocal<QNode> myNode;
5    public CLHLock() {
6    tail = new AtomicReference<QNode>(null);
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12     myPred = new ThreadLocal<QNode>() {
13       protected QNode initialValue() {
14         return null;
15       }
16     };
17   }
18   ...
19 }
```

图7-9 CLHLock类：域和构造函数

该类在QNode对象的布尔型locked域中记录了每个线程的状态。如果该域为*true*，则相应的线程要么已经获得锁，要么正在等待锁；如果该域为*false*，则相应的线程已经释放了锁。锁本身被表示为QNode对象的虚拟链表。之所以使用术语“虚拟”是因为链表是隐式的：每个线程通过一个线程局部变量pred指向其前驱。公共的tail域对于最近加入到队列的结点来说是一个AtomicReference<QNode>。

```
20    public void lock() {
21      QNode qnode = myNode.get();
22      qnode.locked = true;
23      QNode pred = tail.getAndSet(qnode);
24      myPred.set(pred);
25      while (pred.locked) {}
26    }
27    public void unlock() {
28      QNode qnode = myNode.get();
29      qnode.locked = false;
30      myNode.set(myPred.get());
31    }
32  }
```

图7-10　CLHLock类：lock()和unlock()方法

如图7-10所示，若要获得锁，线程将其QNode的locked域设为*true*，表示该线程不准备释放锁。随后线程对tail域调用getAndSet()方法，使它自己的结点成为队列的尾部，同时获得一个指向其前驱QNode的引用。最后线程在其前驱的locked域上旋转，直到前驱释放该锁。若要释放锁，线程将其locked域设为*false*，然后重新使其前驱的QNode作为新结点以使将来的锁访问。之所以能这样做是因为该线程的前驱此刻不再使用它的QNode，而且线程的老的QNode既可以被它的后继也可以由tail所引用[1]。虽然在本例中没有这么做，但回收结点是可行的，这样的话，如果有L个锁，且每个线程每次最多访问一个锁，那么与ALock类需要$O(Ln)$的空间相比，CLHLock类只需要$O(L+n)$的空间。图7-11描述了CLHLock的一次典型的执行过程。

150
↓
153

与ALock一样，该算法让每个线程在不同的存储单元上旋转，这样当一个线程释放它的锁时，只能使其后继的cache无效。该算法比ALock类所需的空间少，且不需要知道可能使用锁的线程的数量。该算法和ALock类一样，也提供了先来先服务的公平性。

也许这种锁算法的唯一缺点就是它在无cache的NUMA系统结构下性能很差。每个线程都自旋等待其前驱结点的locked域变为*false*，如果内存位置较远，那么性能将会受到损失。然而，在cache一致性的系统结构上，该方法非常有效。

7.5.3　MCS队列锁

图7-12描述了MCSLock类的域和构造函数。该类的锁也被表示为QNode对象的链表，其中的每个QNode要么表示一个锁持有者，要么表示一个正在等待获得锁的线程。与CLHLock类不同，锁链表是显式的而不是虚拟的：整个链表通过QNode对象里的next域（全局可访问的）所体现，而不是由线程的局部变量所体现。

---

[1] 在类似于Java和C#这些具有垃圾回收功能的语言里，不必为了保证正确性而重用结点，但在C++或C等语言里则不是这样的。

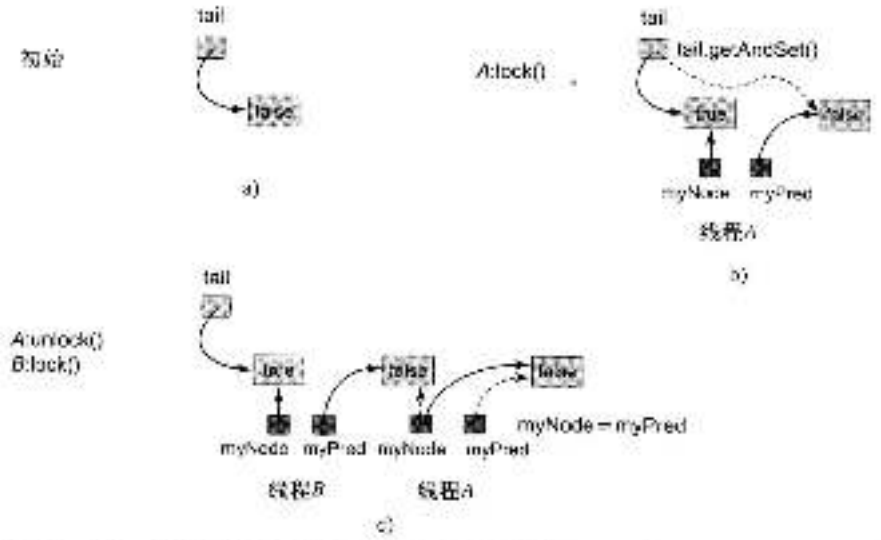


图7-11 CLHLock类：锁的获取和释放。初始时tail域指向QNode，其locked域为*false*，然后线程*A*对tail域调用getAndSet()，把它的QNode插入到队列尾部，同时获取一个指向其前驱的QNode的引用。接下来，*B*同样把它的QNode插入队列尾部。*A*接着将其结点的locked域设置为*false*来释放锁，然后回收由pred指向的QNode，以便今后的锁访问

```
1  public class MCSLock implements Lock {
2    AtomicReference<QNode> tail;
3    ThreadLocal<QNode> myNode;
4    public MCSLock() {
5      tail = new AtomicReference<QNode>(null);
6      myNode = new ThreadLocal<QNode>() {
7        protected QNode initialValue() {
8          return new QNode();
9        }
10     };
11   }
12   ...
13   class QNode {
14     boolean locked = false;
15     QNode  next = null;
16   }
17 }
```

图7-12 MCSLock类：域，构造函数和QNode类

图7-13描述了MCSLock类的lock()和unlock()方法。若要获得锁，线程把它自己的QNode添加到链表的尾部（第20行）。如果队列原先不为空，则将前驱QNode的next域设置为指向它自己的QNode。然后在它自己的QNode的（局部）locked域上自旋等待，直到其前驱将该域设为*false*为止（第21－26行）。

unlock()方法检查结点的next域是否为空（第30行）。如果是，则要么不存在其他线程正在争用这个锁，要么存在一个正在争用的线程，但该线程运行得很慢。令*q*是该线程的当前结点。为了区分这种情况，对tail域调用方法compareAndSet(*q*,*null*)。如果调用成功，则没有

其他线程正在试图获得锁，将tail域置为*null*并返回。否则，说明有另一个（较慢的）线程试图获得锁，于是该方法自旋以等待它结束（第34行）。在任一种情形下，一旦出现了后继，unlock()方法就将它的后继的locked域设置为*false*，表明目前锁是空闲的。此时，其他的线程不能访问这个QNode，所以该对象可以被重用。图7-14描述了MCSLock的执行实例。

```
18   public void lock() {
19     QNode qnode = myNode.get();
20     QNode pred = tail.getAndSet(qnode);
21     if (pred != null) {
22       qnode.locked = true;
23       pred.next = qnode;
24       // wait until predecessor gives up the lock
25       while (qnode.locked) {}
26     }
27   }
28   public void unlock() {
29     QNode qnode = myNode.get();
30     if (qnode.next == null) {
31       if (tail.compareAndSet(qnode, null))
32         return;
33       // wait until successor fills in the next field
34       while (qnode.next == null) {}
35     }
36     qnode.next.locked = false;
37     qnode.next = null;
38   }
```

清单7-13 MCSLock类：lock()和unlock()方法

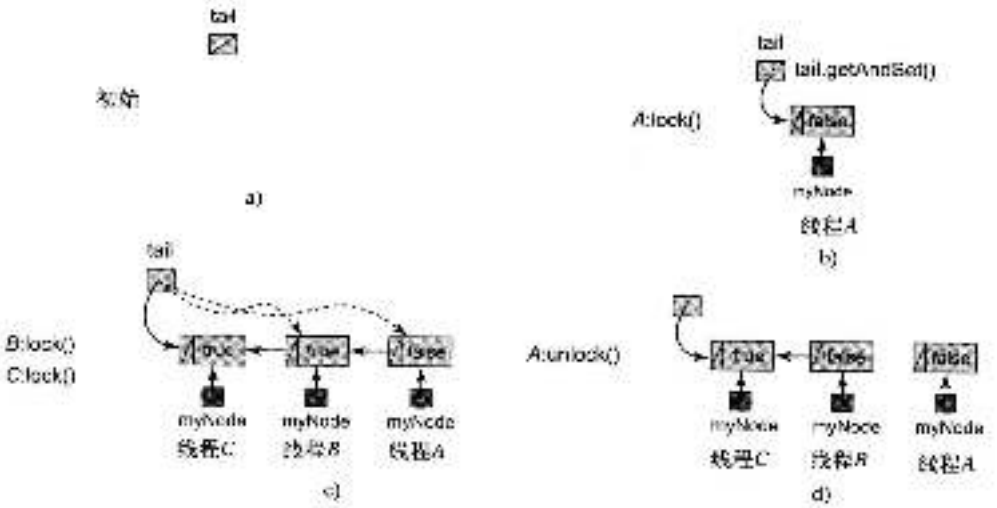


图7-14 MCSLock的锁获取和锁释放。a) tail初始化为*null*；b) 若要获得锁，线程*A*把它自己的QNode加入链表的尾部。由于该结点没有前驱，从而可以进入临界区。c) 线程*B*把它自己的QNode放在链表的尾部，修改其前驱的QNode指向它自己。随后线程*B*在它的locked域上自旋，直到它的前驱*A*把这个域从*true*改为*false*。线程*C*重复这个过程。d) 若要释放锁，*A*顺着它的next域找到它的后继*B*并把*B*的locked域设置为*false*。现在可以重用它的QNode了

该锁具有CLHLock的优点，特别是每个锁释放仅能使其后继的cache项无效，这种算法更适于无cache的NUMA系统结构，因为是由每个线程来控制它所自旋的存储单元的。如同CLHLock一样，结点能被重复使用，因此，该锁的空间复杂度为$O(L+n)$。MCSLock算法的一个缺点就是释放锁时也需要旋转，另外一个缺点就是它比CLHLock算法的读、写和compareAndSet()调用次数多。

154
-
156

7.6 时限队列锁

Java的Lock接口中包含一个tryLock()方法，该方法允许调用者指定一个时限，调用者为获得锁而准备等待的最大时间。如果在调用者获得锁之前超时，调用者则放弃获得锁的尝试。该方法通过一个布尔型的返回值来说明锁的申请是否成功。（在第8章的编程提示8.2.3中解释了为什么这种方法会抛出InterruptedException异常。）

由于线程能够非常简单地从tryLock()调用返回，所以放弃一个BackoffLock请求是很容易的。超时无需等待，只要求固定的数作步骤。与此相反，若对任意的队列锁算法都进行超时控制却并非易事：如果一个线程简单地返回，那么排在它后面的线程将会饿死。

下面是一超时退队列锁的乌瞰图。就像CLHLock一样，锁是一个结点的虚拟队列，每个线程在它的前驱结点上自旋，等待锁被释放。正如前面所提到的，若一个线程超时，则该线程不能简单地抛弃它的队列结点，因为当锁被释放时，该线程的后继无法注意到这种情形。另一方面，让一个队列结点从链表中删除而并不扰乱并发锁的释放似乎是相当困难的。因此，可以使用惰性方法：若一个线程超时，则该线程将它的结点标记为已放弃。这样该线程在队列中的后继（如果有）将会注意到它正在自旋的结点已经被放弃，于是开始在被放弃结点的前驱上自旋。这种方法有一个额外的好处：后继线程能重用被放弃的结点。

图7-15描述了TOLock类（时限锁）的域、构造函数以及QNode类。TOLock是一个基于CLHLock类的队列锁，它支持无等待超时，即使对于结点链表中正在等待锁的线程也是如此。

```
1  public class TOLock implements Lock{
2    static QNode AVAILABLE = new QNode();
3    AtomicReference<QNode> tail;
4    ThreadLocal<QNode> myNode;
5    public TOLock() {
6      tail = new AtomicReference<QNode>(null);
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12   }
13   ...
14   static class QNode {
15     public QNode pred = null;
16   }
17 }
```

图7-15 TOLock类：域、构造函数及QNode类

当一个QNode的pred域为*null*时，该结点所对应的线程或者还未获得锁或者已经释放了锁。当一个QNode的pred域指向一个可判别的静态QNode（AVAILABLE）时，其对应线程已经释放

了锁。如果pred域指向其他的某个QNode，那么相应的线程已经放弃了锁请求，这样后继结点的线程应该在被放弃结点的前驱上等待。

图7-16描述了TOLock类的tryLock()和unlock()方法。tryLock()方法创建一个pred域为空的新QNode，它像CLHLock类一样（第5－8行）把该结点加入到链表中。如果这个锁是空闲的（第9行），线程则进入临界区。否则，线程自旋等待其前驱QNode的pred域被改变（第12-19行）。如果前驱线程超时，则设置pred域指向其前驱，并在新的前驱上旋转。列7-17描述了这种操作序列的一个实例。最后，如果线程自己超时（第20行），那么它就在tail域上调用compareAndSet()来尝试从链表中删除它的QNode。如果compareAndSet()调用失败，说明这个线程有后继，线程则设置它的QNode的pred域（原来为*null*）指向其前驱的QNode，表明它已从队列中放弃。

```
1   public boolean tryLock(long time, TimeUnit unit)
2       throws InterruptedException {
3     long startTime = System.currentTimeMillis();
4     long patience = TimeUnit.MILLISECONDS.convert(time, unit);
5     QNode qnode = new QNode();
6     myNode.set(qnode);
7     qnode.pred = null;
8     QNode myPred = tail.getAndSet(qnode);
9     if (myPred == null || myPred.pred == AVAILABLE) {
10      return true;
11    }
12    while (System.currentTimeMillis() - startTime < patience) {
13      QNode predPred = myPred.pred;
14      if (predPred == AVAILABLE) {
15        return true;
16      } else if (predPred != null) {
17        myPred = predPred;
18      }
19    }
20    if (!tail.compareAndSet(qnode, myPred))
21      qnode.pred = myPred;
22    return false;
23  }
24  public void unlock() {
25    QNode qnode = myNode.get();
26    if (!tail.compareAndSet(qnode, null))
27      qnode.pred = AVAILABLE;
28  }
29 }
```

图7-16 TOLock类：tryLock()和unlock()方法

在unlock()方法中，线程通过使用compareAndSet()来检查它是否有后继（第26行）。如果有，则设置它的pred域为AVAILABLE。要注意在这个时刻重新使用线程的老结点是很危险的，因为该结点有可能被它的直接后继所引用，或被一个由这种引用所组成的链所引用。一旦线程跳过超时结点并进入临界区，那么这个链中的结点就可以被回收。

TOLock具有CLHLock的大多数优点：在缓存的存储单元上进行本地自旋以及对锁空闲的快速检测。它也具有BackoffLock的无等待超时特性。然而，该锁也存在着一些缺点，包括每次锁访问都需要分配一个新结点以及在锁上旋转的线程在它访问临界区之前有可能不得不回溯一个超时结点链。

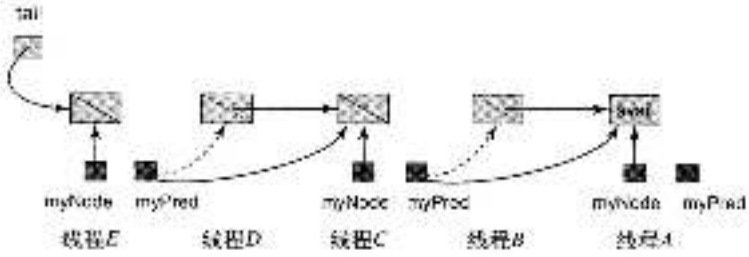


图7-17 为获得TOLock而必须跳过的超时结点。线程B和D已经超时，它们的pred域重新指向能表中它们的前驱。线程C发现B的域指向A，所以开始在A上自旋。类似地，线程E在旋转等待C。当A执行完并设置它的pred域为AVAILABLE时，C将访问临界区并在离开时将它的pred域设置为AVAILABLE，从而释放E

7.7 复合锁

自旋锁算法采用了折中的方案。队列锁则提供了先来先服务的公平性，快速锁释放以及低争用特性，但需要非平凡的协议来重用被放弃的结点。相反，后退锁能支持平凡的超时协议，但其本身是不可扩展的，况且如果没有恰当选定超时参数，锁的释放则有可能很慢。本节考虑一种具有上述两种方法优点的高级锁算法。

157
?
159

考虑下面通过观察所得的结论：在一个队列锁中，只有位于队列前面的线程需要进行锁切换。对于队列锁和后退锁的一种平衡方案就是在进入临界区的过程中只允许在队列中保持少量的等待线程。若剩下的（线程）企图进入这个短队列，则采用指数后退的方法。线程通过后退来中止是很平常的。

CompositeLock类维护一个短的固定大小的锁结点数组。每个试图获得锁的线程随机地在数组中选择一个结点。如果该结点正在使用，该线程则向后后退（自适应性地）并再次尝试。一旦线程获得一个结点，它就将该结点入队到一个类似于TOLock的队列中。线程在前驱结点上自旋。如果该结点的所有者发出已完成的信号，线程则进入临界区。当线程离开时，或者它已完成或者超时，它让出该结点的所有权，从而使另一个后退线程可以获得该结点。这个过程中难处理的部分就是当有多个线程正在试图获得结点的控制权时，如何回收数组中被释放的结点。

图7-18描述了CompositeLock的域、构造函数和unlock()方法。waiting域是一个固定大小的QNode数组，tail域是一个AtomicStampedReference<QNode>，它包含一个对队尾的引用以及一个版本号，该版本号用于避免在更新操作中存在的ABA问题（第10章的编程提示10.6.1详细解释了AtomicStampedReference<T>，第11章对ABA问题进行了完整的讨论[⊖]）。tail域要么为*null*要么指向最近被插入队列的结点。图7-19描述了QNode类。每个QNode包含一个State域和一个指向队列中前驱结点的引用。

QNode有四种可能的状态：WAITING、RELEASED、ABORTED和FREE。状态为WAITING的结点被链接在队列中，拥有该结点的线程要么在临界区内要么正在等待进入临界区。当一个结点的所有者准备离开临界区并释放锁时，该结点变为RELEASED。当线程要放弃获取锁的尝试时，则处于其他两个状态。如果准备退出的线程已获得一个结点但还未使该结点入队，则被标记

⊖ 仅在无垃圾回收的语言中使用动态存储分配时，ABA才是一个具有代表性的问题。之所以在这里提到它，是因为我们采用一个数组来实现一个动态结构。

为FREE，如果结点已入队，则标记为ABORTED。

```
1  public class CompositeLock implements Lock{
2    private static final int SIZE = ...;
3    private static final int MIN_BACKOFF = ...;
4    private static final int MAX_BACKOFF = ...;
5    AtomicStampedReference<QNode> tail;
6    QNode[] waiting;
7    Random random;
8    ThreadLocal<QNode> myNode = new ThreadLocal<QNode>() {
9      protected QNode initialValue() { return null; }};
10   };
11   public CompositeLock() {
12     tail = new AtomicStampedReference<QNode>(null,0);
13     waiting = new QNode[SIZE];
14     for (int i = 0; i < waiting.length; i++) {
15       waiting[i] = new QNode();
16     }
17     random = new Random();
18   }
19   public void unlock() {
20     QNode acqNode = myNode.get();
21     acqNode.state.set(State.RELEASED);
22     myNode.set(null);
23   }
24   ...
25 }
```

图7-18 CompositeLock类：域、构造函数和unlock()方法

```
1  enum State {FREE, WAITING, RELEASED, ABORTED};
2  class QNode {
3    AtomicReference<State> state;
4    QNode pred;
5    public QNode() {
6      state = new AtomicReference<State>(State.FREE);
7    }
8  }
```

图7-19 CompositeLock类：QNode类

图7-20描述了tryLock()方法。线程分三步获得锁。首先，它获得waiting数组中的一个结点（第7行），接着让该结点入队（第12行），最后等待直到该结点到达队首（第9行）。

```
1  public boolean tryLock(long time, TimeUnit unit)
2      throws InterruptedException {
3    long patience = TimeUnit.MILLISECONDS.convert(time, unit);
4    long startTime = System.currentTimeMillis();
5    Backoff backoff = new Backoff(MIN_BACKOFF, MAX_BACKOFF);
6    try {
7      QNode node = acquireQNode(backoff, startTime, patience);
8      QNode pred = spliceQNode(node, startTime, patience);
9      waitForPredecessor(pred, node, startTime, patience);
10     return true;
11   } catch (TimeoutException e) {
12     return false;
13   }
14 }
```

图7-20 CompositeLock类：tryLock()方法

160
/
161

图7-21描述了在waiting数组中获得一个结点的算法。线程随机地选择一个结点，并把该结点的状态从FREE变为WAITING来尝试获得该结点（第7行）。如果失败，则检查结点的状态。若结点状态为ABORTED或RELEASED（第12行），则线程可以“清除”该结点。为了避免和其他线程的同步冲突，只有当结点是队列中最后一个元素（tail的值）时才能被清除。如果队尾结点为ABORTED，那么让tail重新指向那个结点的前驱；否则，tail被置为*null*。相反，如果分配的结点状态为WAITING，那么线程后退并重试。如果线程在获得结点之前超时，则抛出TimeoutException异常（第27行）。

```
1   private QNode acquireQNode(Backoff backoff, long startTime,
2                              long patience)
3    throws TimeoutException, InterruptedException {
4     QNode node = waiting[random.nextInt(SIZE)];
5     QNode currTail;
6     while (true) {
7       if (node.state.compareAndSet(State.FREE, State.WAITING)) {
8         return node;
9       }
10      currTail = tail.get(currStamp);
11      State state = node.state.get();
12      if (state == State.ABORTED || state == State.RELEASED) {
13        if (node == currTail) {
14          QNode myPred = null;
15          if (state == State.ABORTED) {
16            myPred = node.pred;
17          }
18          if (tail.compareAndSet(currTail, myPred,
19              currStamp[0], currStamp[0]+1)) {
20            node.state.set(State.WAITING);
21            return node;
22          }
23        }
24      }
25      backoff.backoff();
26      if (timeout(patience, startTime)) {
27        throw new TimeoutException();
28      }
29    }
30  }
```

图7-21 CompositeLock类：acquireQNode()方法

一旦线程获得一个结点，图7-22所示的spliceQNode()方法则将该结点插入队列。线程反复地尝试将tail设置为被分配的结点。如果线程超时，则将分配的结点标记为FREE，然后抛出TimeoutException异常。如果成功，则返回tail的先前值，该值由队列中结点的前驱所
162 获得。

最后，一旦结点已经入队，线程必须通过调用waitForPredecessor()来等待轮到它（图7-23）。如果前驱为*null*，线程的结点则为队列中的首元素，于是线程将该结点保存在线程的局部myNode域中（为了以后的unlock()使用），然后进入临界区。如果前驱结点不是RELEASED，那么线程检查它是否为ABORTED（第11行）。如果是，线程则将结点标记为FREE并且在被放弃结点的前驱上等待。如果线程超时，则把它自己的结点标记为ABORTED并抛出

TimeoutException异常。否则，当前驱结点变为RELEASED时线程把它标记为FREE，将自己的结点记录在线程的局部myNode域中，然后进入临界区。

```
1   private QNode spliceQNode(QNode node, long startTime, long patience)
2    throws TimeoutException {
3     QNode currTail;
4     do {
5       currTail = tail.get(currStamp);
6       if (timeout(startTime, patience)) {
7         node.state.set(State.FREE);
8         throw new TimeoutException();
9       }
10    } while (!tail.compareAndSet(currTail, node,
11        currStamp[0], currStamp[0]+1));
12    return currTail;
13  }
```

图7-22　CompositeLock类：spliceQNode()方法

```
1   private void waitForPredecessor(QNode pred, QNode node, long startTime,
2                                   long patience)
3    throws TimeoutException {
4     int[] stamp = {0};
5     if (pred == null) {
6       myNode.set(node);
7       return;
8     }
9     State predState = pred.state.get();
10    while (predState != State.RELEASED) {
11      if (predState == State.ABORTED) {
12        QNode temp = pred;
13        pred = pred.pred;
14        temp.state.set(State.FREE);
15      }
16      if (timeout(patience, startTime)) {
17        node.pred = pred;
18        node.state.set(State.ABORTED);
19        throw new TimeoutException();
20      }
21      predState = pred.state.get();
22    }
23    pred.state.set(State.FREE);
24    myNode.set(node);
25    return;
26  }
```

图7-23　CompositeLock类：waitForPredecessor()方法

unlock()方法（图7-18）只是简单地从myNode中查找它的结点并标记为RELEASED。图7-24是图7-18算法的一次实际执行过程。

CompositeLock具有一些令人感兴趣的特性。当多个线程后退时，它们访问不同的存储单元，从而降低了争用。锁的切换就像CLHLock和TOLock算法一样快。放弃一个锁请求对处于后退阶段的线程来说是平常的，而且对于已经获得队列结点的线程来说要更加简单直接。假设有L个锁和n个线程，CompositeLock类在最坏情况下只需$O(L)$的存储空间，而TOLock类则需

要$O(L \cdot n)$。但该锁有一个缺点：CompositeLock类并不保证先来先服务的访问。

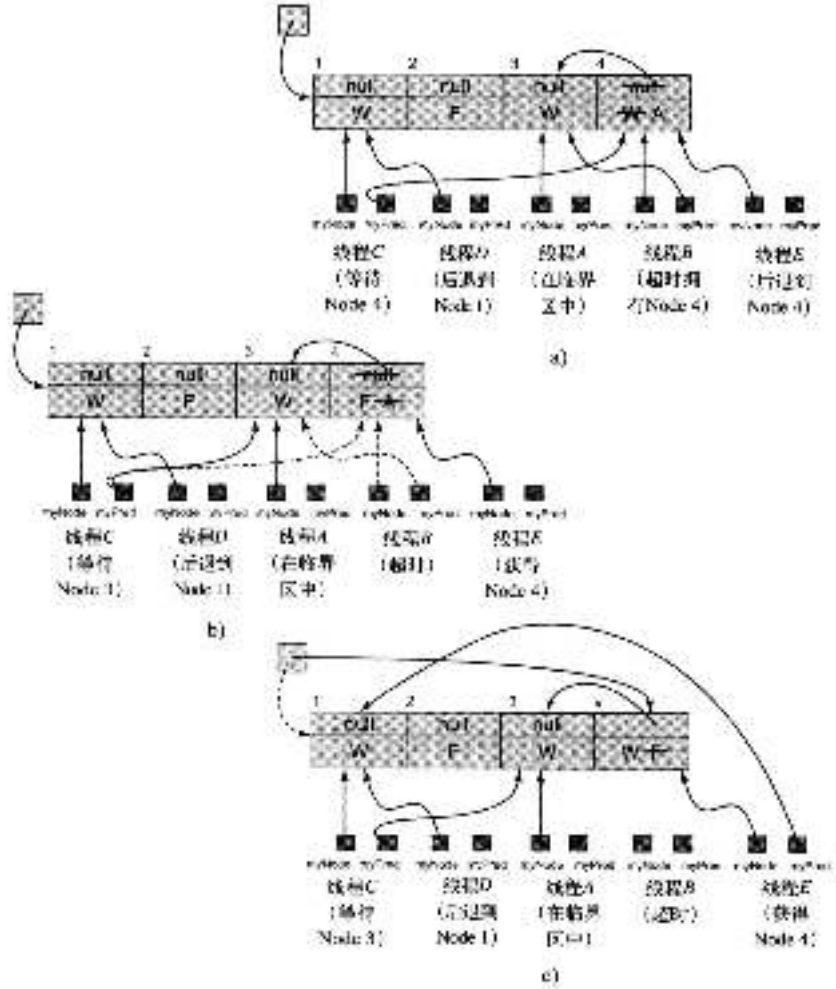


图7-21 CompositeLock类：一次执行过程。在a中，线程A（获得Node 3）处于临界区中，线程B（Node 4）正在等待A离开临界区，线程C（Node 1）则在等待线程B，线程D和E正在后退，等待获得一个结点。Node 2是空闲的，tail域指向Node 1，该结点是最后一个要被插入到队列的结点。比如B超时，从而插入一个对其前驱的显式引用，并把Node 4的状态从WAITING（用W表示）变为ABORTED（用A表示）。在b中，线程C清除状态为ABORTED的Node 4，将其状态设置为FREE，并按照显式引用从4找到3（通过重新指向其后部myPred域），然后开始等待A（Node 3）离开临界区。在c中，E获取状态为FREE的Node 4，使用compareAndSet()方法将它的状态设置为WAITING。接着线程E把Node 4插入队列，使用compareAndSet()方法把Node 4交换到tail，然后等待之前指向tail的Node 1

快速路径复合锁

尽管设计CompositeLock的初衷是为了保证在争用时有较好的性能，然而在无并发的情形下，性能也是非常重要的。理想情况下，对于一个单独运行的线程来说，获取一个锁应该和

获取一个无争用的TASLock同样简单。然而，在CompositeLock算法中，一个单独运行的线程必须重新设置tail域使之离开一个已释放的锁，声明该结点，然后把它插入队列中。

快速路径是一条让单个线程执行复杂算法时快速完成的捷径。可以扩展CompositeLock算法使之包含一条快速路径，在该快速路径中一个单独运行的线程不必获取结点并插入队列就可以获得一个空闲锁。

下面给出该算法的乌鸦图。我们增加一个额外的状态以区分被一般线程持有的锁和被快速路径线程持有的锁。如果线程发现锁是空闲的，则尝试通过快速路径来获取。若成功，那么它在一个原子步内就已经获得了锁。若失败，那么它像以前一样使自己入队。

现在详细分析这个算法。为了减少重复代码，将CompositeFastPathLock类定义为CompositeLock的子类（参见图7-25）。图7-26给出了fastPathLock()和unlock()方法。

```
1  public class CompositeFastPathLock extends CompositeLock {
2    private static final int FASTPATH = ...;
3    private boolean fastPathLock() {
4      int oldStamp, newStamp;
5      int stamp[] = {0};
6      QNode qnode;
7      qnode = tail.get(stamp);
8      oldStamp = stamp[0];
9      if (qnode != null) {
10       return false;
11     }
12     if ((oldStamp & FASTPATH) != 0) {
13       return false;
14     }
15     newStamp = (oldStamp + 1) | FASTPATH;
16     return tail.compareAndSet(qnode, null, oldStamp, newStamp);
17   }
18   public boolean tryLock(long time, TimeUnit unit)
19     throws InterruptedException {
20     if (fastPathLock()) {
21       return true;
22     }
23     if (super.tryLock(time, unit)) {
24       while ((tail.getStamp() & FASTPATH ) != 0){};
25       return true;
26     }
27     return false;
28   }
```

图7-25　CompositeFastPathLock类：如果通过快速路径成功地获得锁，那么私有的fastPathLock()方法将返回*true*

用FASTPATH标志量来标识一个线程已通过快速路径获得了锁。由于需要把该标志量和对tail域的引用作为一个整体来操作，所以要从tail域的整型戳中"窃取"一个高位（第2行）。私有的fastPathLock()方法检查tail域的整型戳中是否有一个清空的FASTPATH标志量和一个*null*引用。如果有，则该法通过调用compareAndSet()将FASTPATH标志量置为*true*来获得锁，同时确保引用仍为*null*。这样的话，对一个无争用的锁的获取只需要一个原子操作。如果fastPathLock()方法成功，则返回*true*，否则返回*false*。

163 ; 166

tryLock()方法（第18~28行）首先调用fastPathLock()来尝试快速路径。如果失败，则通过调用CompositeLock类的tryLock()方法来尝试慢速路径。然而，在从慢速路径返回之前，

它必须保证没有其他的线程持有快速路径锁，等待FASTPATH标志量被清空（第24行）。

```
1   private boolean fastPathUnlock() {
2     int oldStamp, newStamp;
3     oldStamp = tail.getStamp();
4     if ((oldStamp & FASTPATH) == 0) {
5       return false;
6     }    int[] stamp = {0};
7     QNode qnode;
8     do {
9       qnode = tail.get(stamp);
10      oldStamp = stamp[0];
11      newStamp = oldStamp & (~FASTPATH);
12    } while (!tail.compareAndSet(qnode, qnode, oldStamp, newStamp));
13    return true;
14  }
15  public void unlock() {
16    if (!fastPathUnlock()) {
17      super.unlock();
18    };
19  }
```

图7.26 CompositeFastPathLock类：fastPathLock()和unlock()方法

如果快速路径标志量没有被设置（第4行），fastPathUnlock()方法返回*false*。否则，它不断尝试清空标志量，并保持引用部分不变（第8~12行）。当它成功时则返回*true*。

CompositeFastPathLock类的unlock()方法首先调用fastPathUnlock()（第16行）。如果该调用没能释放锁，那么它接着调用CompositeLock的unlock()方法（第17行）。

7.8 层次锁

目前大多数cache一致的系统结构都以集群方式来组织处理器。在一个集群内的通信要比在集群间的通信快得多。例如，一个集群可以对应于一组通过快速互连来共享存储器的处理器，也可对应于运行在一个多核系统结构中一个单核上的所有线程。本节主要考虑这种对局部差异较敏感的锁。称这样的锁为层次锁，因为在设计中要考虑系统的层次存储结构以及访问开销。

系统结构的存储结构可以具有多个层次，为简单起见，我们假设只有两个层次。下面考虑一种由多个处理器集群所组成的系统结构。同一集群中的处理器通过共享cache进行高效通信。集群之间的通信代价要比集群内的代价大得多。

假设每个集群都有一个唯一的集群id，该id对集群内的每个线程都是已知的，并可通过ThreadID.getCluster()获得，线程不能在集群间迁移。

7.8.1 层次后退锁

"测试-测试-设置"（test-and-test-and-set）锁非常适于集群开发。假定线程A持有锁。若A所在集群中的线程具有较短的后退时间，那么当释放锁时，本地线程要比远程线程更有可能获得锁，从而降低了切换锁的拥有权所需的总时间。图7.27描述了一种基于这种原则设计的层次后退锁HBOLock。

```
1  public class HBOLock implements Lock {
2    private static final int LOCAL_MIN_DELAY = ...;
3    private static final int LOCAL_MAX_DELAY = ...;
4    private static final int REMOTE_MIN_DELAY = ...;
5    private static final int REMOTE_MAX_DELAY = ...;
6    private static final int FREE = -1;
7    AtomicInteger state;
8    public HBOLock() {
9      state = new AtomicInteger(FREE);
10   }
11   public void lock() {
12     int myCluster = ThreadID.getCluster();
13     Backoff localBackoff =
14         new Backoff(LOCAL_MIN_DELAY, LOCAL_MAX_DELAY);
15     Backoff remoteBackoff =
16         new Backoff(REMOTE_MIN_DELAY, REMOTE_MAX_DELAY);
17     while (true) {
18       if (state.compareAndSet(FREE, myCluster)) {
19         return;
20       }
21       int lockState = state.get();
22       if (lockState == myCluster) {
23         localBackoff.backoff();
24       } else {
25         remoteBackoff.backoff();
26       }
27     }
28   }
29   public void unlock() {
30     state.set(FREE);
31   }
32 }
```

图7-27　HBOLock类：层次后退锁

HBOLock的缺点之一就在于它过度利用了局部性。这样有可能存在同一集群中的线程不断地传递锁，而其他集群中的线程发生饥饿的现象。另外，获取和释放锁会使锁域的远程cache副本无效，这将在cache一致的NUMA系统结构中产生巨大开销。

7.6.2　层次CLH队列锁

为了提供一种更为平衡的集群开发方法，首先分析层次队列锁的设计。层次锁设计中，问题的关键就是要协调冲突时的公平性需求。既要保证在同一集群内进行锁的传递以避免较高的通信开销，同时也要保证某种程度的公平性，以使远程锁请求不至于比本地锁请求过于延迟。我们通过将相同集群的请求序列一起调度来平衡这些需求。

HCLHLock队列锁（图7-28）由一组本地队列和一个全局队列组成。每个集群对应一个本地队列。所有队列都是由结点组成的链表，其中的链接是隐式的，即链接被保存在线程的局部域myQNode和myPred中。

我们通常给线程拥有它自己的myQNode结点。对于队列中的任一结点（除队首以外），其拥有者的myPred结点就是它的前驱结点。图7-29给出了域和构造函数。图7-30描述了QNode类。每个结点有三个虚拟域：当前（或最近）拥有者的ClusterId，两个布尔型域successorMustWait和tailWhenSpliced。之所以称这些域是虚拟的，是因为对它们的更新要以原子的方式进行，

167
↓
168

因此，在AtomicInteger域中把它们描述为位域（bit-field），采用简单的屏蔽和移位操作来获取它们的值。tailWhenSpliced域用来说明该结点是否为当前被拼接到全局队列中的序列的最后一个结点。successorNustWait域和最初的CLH算法一样：在入队前被设为*true*，在释放锁时由结点的拥有者设为*false*。这样，对于一个正在等待获得锁的线程来说，当其前驱的successorMustWait域变为*false*时，该线程可以向前推进。由于需要原子地更新这些域，所以它们应该是私有的，并要使用同步方法间接访问。

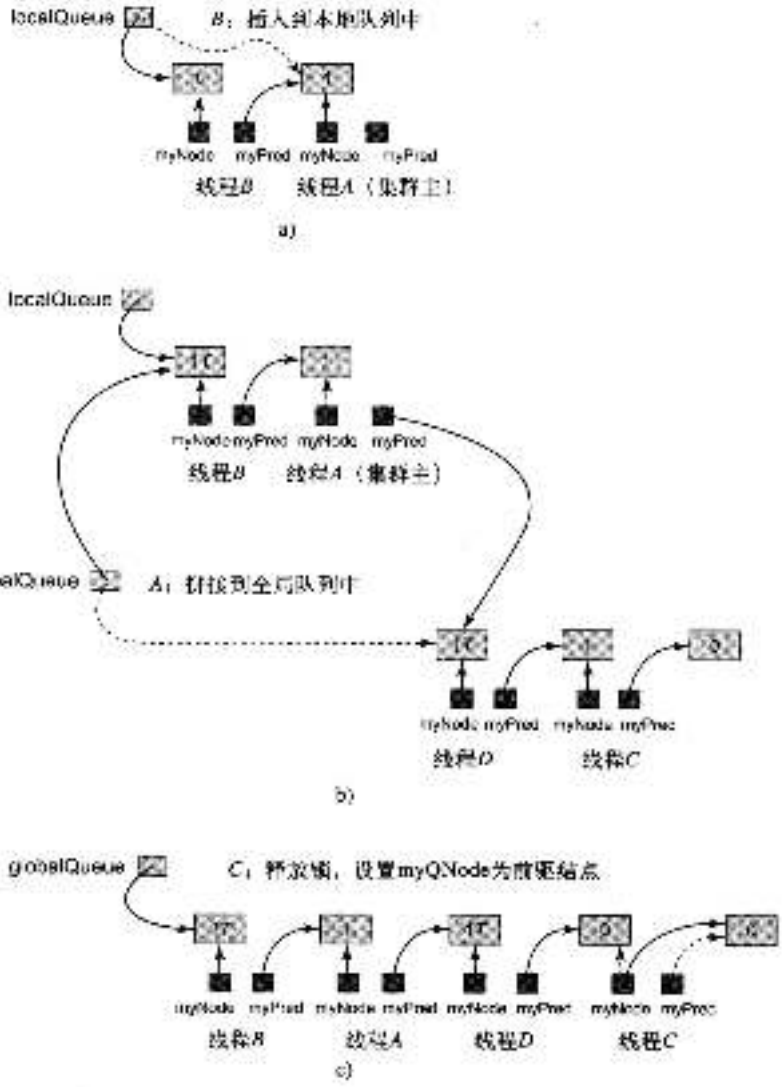


图7-28 HCLHLock中的锁请求和锁释放。在结点的successorNustWait域中，用0来标记假，用1标记真。当一个结点正通过增加符号*T*被拼接时，该结点被标记为本地队列元素。在a中，*B*将它自己的结点插入到本地队列中。在b中，*A*在将包含*A*和*B*的结点的本地队列拼接到已包含*C*和*D*的结点的全局队列中。在c中，*C*通过将它的结点的successorNustWait 标志设为假来释放锁，然后设置myQNode为前驱结点

```
1  public class HCLHLock implements Lock {
2    static final int MAX_CLUSTERS = ...;
3    List<AtomicReference<QNode>> localQueues;
4    AtomicReference<QNode> globalQueue;
5    ThreadLocal<QNode> currNode = new ThreadLocal<QNode>() {
6      protected QNode initialValue() { return new QNode(); }
7    };
8    ThreadLocal<QNode> predNode = new ThreadLocal<QNode>() {
9      protected QNode initialValue() { return null; }
10   };
11   public HCLHLock() {
12     localQueues = new ArrayList<AtomicReference<QNode>>(MAX_CLUSTERS);
13     for (int i = 0; i < MAX_CLUSTERS; i++) {
14       localQueues.add(new AtomicReference <QNode>());
15     }
16     QNode head = new QNode();
17     globalQueue = new AtomicReference<QNode>(head);
18   }
```

图7-29　HCLHLock类：域和构造函数

```
1    class QNode {
2      // private boolean tailWhenSpliced;
3      private static final int TWS_MASK = 0x80000000;
4      // private boolean successorMustWait = false;
5      private static final int SNK_MASK = 0x40000000;
6      // private int clusterID;
7      private static final int CLUSTER_MASK = 0x3FFFFFFF;
8      AtomicInteger state;
9      public QNode() {
10       state = new AtomicInteger(0);
11     }
12     public void unlock() {
13       int oldState = 0;
14       int newState = ThreadID.getCluster();
15       // successorMustWait = true;
16       newState |= SNK_MASK;
17       // tailWhenSpliced = false;
18       newState &= (~TWS_MASK);
19       do {
20         oldState = state.get();
21       } while (! state.compareAndSet(oldState, newState));
22     }
23     public int getClusterID() {
24       return state.get() & CLUSTER_MASK;
25     }
26     // other getters and setters omitted.
27   }
```

图7-30　HCLHLock类：内部QNode类

图7-28说明了HCLHLock类是如何获取和释放锁的。lock()方法首先将线程的结点加入到本地队列，然后等待直到该线程要么能够进入临界区要么它的结点成为本地队列的队首。在后一种情况下，称该线程为集群主，由它负责把本地队列拼接到全局队列中。

图7-31为lock()方法的代码。由于线程的结点已被初始化，所以successorMustWait为*true*，tailWhenSpliced为*false*，ClusterId域为调用者的集群。该线程将它的结点加入到其本地集群队列的最后（尾部），使用compareAndSet()把队尾改为它的结点（第9行）。一旦成

功，该线程则将它的myPred设置为由它替换为队尾的结点。该结点称为前驱。

```
1   public void lock() {
2     QNode myNode = currNode.get();
3     AtomicReference<QNode> localQueue = localQueues.get
4                                         (ThreadID.getCluster());
5     // splice my QNode into local queue
6     QNode myPred = null;
7     do {
8       myPred = localQueue.get();
9     } while (!localQueue.compareAndSet(myPred, myNode));
10    if (myPred != null) {
11      boolean iOwnLock = myPred.waitForGrantOrClusterMaster();
12      if (iOwnLock) {
13        predNode.set(myPred);
14        return;
15      }
16    }
17    // I am the cluster master: splice local queue into global queue.
18    QNode localTail = null;
19    do {
20      myPred = globalQueue.get();
21      localTail = localQueue.get();
22    } while(!globalQueue.compareAndSet(myPred, localTail));
23    // inform successor it is the new master
24    localTail.setTailWhenSpliced(true);
25    while (myPred.isSuccessorMustWait()) {};
26    predNode.set(myPred);
27    return;
28  }
```

图7-31 HCLHLock类：lock()方法。像CLHLock一样，lock()将前驱成近所释放的结点保存起来，以用于下一个锁请求

线程随后调用waitForGrantOrClusterMaster()（第11行），让线程自旋直到下列条件之一成立：

1. 前驱结点来自于同一个集群，且tailWhenSpliced和successorMustWait都为*false*。

2. 前驱结点来自于不同的集群或前驱的标志量tailWhenSpliced为*true*。

在第一种情形下，线程的结点为全局队列的队首，所以它进入临界区然后返回（第14行）。在第二种情形下，线程的结点位于本地队列的队首，所以该线程为集群主，从而由它来负责把本地队列拼接到全局队列中。（如果没有前驱，即本地队列尾为*null*，则该线程立刻变为集群主。）大多数由waitForGrantOrClusterMaster()所请求的旋转都是本地的，所以导致很少甚至不产生通信开销。

另外，要么该线程前驱的tailWhenSpliced标志量为*true*，要么其前驱的集群与它自己的集群不同。如果其前驱属于不同的集群，则前驱结点不可能在该线程的本地队列中。前驱结点必定已经被移到全局队列中且被不同集群中的某个线程所回收。另一方面，如果前驱的tailWhenSpliced标志量为*true*，则前驱结点是进入全局队列的最后一个结点。因此该线程的结点此时处于本地队列的队首。它不可能被移到全局队列中，因为只有其结点位于本地队列队首的集群主才能将结点移到全局队列中。

作为集群主，其任务就是将本地队列中的结点拼接到全局队列中。本地队列中的每个线程都在其前驱结点上自旋。集群主读取本地队列的队尾并调用compareAndSet()将全局队列

的队尾改为它在其本地队列队尾所看到的结点（第22行）。如果成功，则myPred就是它所替换的全局队列队尾（第20行）。随后它将最后一个被它拼接到全局队列的结点的tailWhenSpliced标志量设为*true*（第24行），使该结点的（本地）后继知道它现在已是本地队列的队首。该操作序列按照在本地队列中相同的次序将本地结点（一直到本地队尾）移到CLH形式的全局队列中。 169 ~ 172

一旦集群主进入全局队列，它就与在通常的CLHLock队列中一样，在它的（新的）前驱的successorMustWait域为*false*时进入临界区（第25行）。其他那些结点已被拼入的线程认为什么都没有发生，继续像以前一样自旋。当其前驱的successorMustWait域变为*false*时，所有线程都将进入临界区。

和原先的CLHLock算法中一样，线程通过将其结点的successorMustWait域设为*false*来释放锁（图7-32）。开锁时，线程将其前驱的结点保存起来以备下一次锁请求使用（第34行）。

```
29    public void unlock() {
30      QNode myNode = currNode.get();
31      myNode.setSuccessorMustWait(false);
32      QNode node = predNode.get();
33      node.unlock();
34      currNode.set(node);
35    }
```

图7-32　HCLHLock类：unlock()方法。该方法移动由lock()操作所保存的结点，并对QNode进行初始化以在下一个锁请求中使用

HCLHLock锁适于由本地线程所组成的序列，这些序列在全局队列的等待列表中，一个在等待另一个。和CLHLock锁一样，隐式引用的使用最小化了cache缺失，线程在它们后继结点状态的本地cache拷贝上自旋。

7.9　由一个锁管理所有的锁

本章研究了各种具有不同性能和特点的自旋锁。这种多样性非常有用，因为没有哪一种算法能够适用于所有的应用。复杂的算法适于一些应用，而简单的算法则更适于另一些应用，最佳的选择通常取决于应用和目标系统结构的具体特性。

7.10　本章注释

TTASLock归功于Clyde Kruskal、Larry Rudolph和Marc Snir[133]。指数后退是以太网路由中的著名技术，该技术是由Anant Agarwal和Mathews Cherian[6]在多处理器存储上下文中所提出的。Tom Anderson [14]发明了ALock算法，他也是最早在共享存储器多处理器上进行自旋锁性能实验研究的人员之一。由John Mellor-Crummey和Michael Scott[113]所发明的MCSLock可能是最著名的队列锁算法。目前的Java虚拟机采用了基于简化的监控算法的对象同步，如由David Bacon、Ravi Konuru、Chet Murthy和Mauricio Serrano[17]所提出的Thinlock，由Ole Agesen、Dave Detlefs、Alex Garthwaite、Ross Knippel、Y. S. Ramakrishna和Derek White[7]所提出的Metalock，以及由Dave Dice[31]提出的RelaxedLock。所有这些算法都是MCSLock锁的变种。 173

CLHLock锁归功于Travis Craig、Erik Hagersten和Anders Landin[30,110]。无阻塞时限的TOLock则归功于Bill Scherer和Michael Scott[138,139]。CompositeLock及其变种是由Virendra Marathe、Mark Moir和Nir Shavit [120]提出的。在互斥中使用快速路径的思路归功于Leslie Lamport [95]。层次锁是由Zoran Radović和Erik Hagersten提出的。HBOLock则是他们最初所提出算法[130]的一种改进；本章所描述的HCLHLock是由Victor Luchangco、Daniel Nussbaum和Nir Shavit [109]提出的。

Danny Hendler、Faith Fich和Nir Shavit[39] 扩展了Jim Burns和Nancy Lynch的工作，证明了对于任意一种无饥饿互斥算法，即使采用类似getAndSet()或compareAndSet()这样的强操作，都需要$\Omega(n)$的空间，这意味着本章所研究的所有队列锁算法都是空间最优的。

本章的性能分析图主要是基于Tom Anderson[14]的经验研究以及作者在各种现代机器上所收集的数据。之所以使用图示而没有采用实际数据，是因为机器系统结构之间的巨大差异以及它们对锁性能的巨大影响。

对Sherlock Holmes的引用来自于*The Sign of Four* [36] 。

7.11 习题

习题85. 图7-33描述了CLHLock的另一种实现技术，在这种实现中，线程重用自己的结点而不是其前驱的结点。解释这种实现为什么会出错。

```
1  public class BadCLHLock implements Lock {
2    // most recent lock holder
3    AtomicReference<Qnode> tail = new Qnode();
4    // thread-local variable
5    ThreadLocal<Qnode> myNode;
6    public void lock() {
7      Qnode qnode = myNode.get();
8      qnode.locked = true;      // I'm not done
9      // Make me the new tail, and find my predecessor
10     Qnode pred = tail.getAndSet(qnode);
11     // spin while predecessor holds lock
12     while (pred.locked) {}
13   }
14   public void unlock() {
15     // reuse my node next time
16     myNode.get().locked = false;
17   }
18   static class Qnode { // Queue node inner class
19     public boolean locked = false;
20   }
21 }
```

图7-33 一种错误的CLHLock实现

习题86. 假设有n个线程，每个线程先执行foo()方法，接着执行bar()方法。假设要确保所有线程在foo()结束之后才开始执行bar()。为了实现这种同步，在foo()和bar()之间设置一个路障。

第一种路障实现：使用一个由“测试-测试-设置”锁所保护的计数器。每个线程对计数器加锁，将计数器加1，释放锁，然后自旋，重读计数器直至它到达n。

第二种路障实现：使用一个n元布尔数组b，其值全为*false*。线程0将b[0]设为*true*。每个线程i ($0<i\leq n$) 自旋直到b[i−1]为*true*，然后将b[i]设为*true*，再继续前进。

在基于总线的cache一致性系统结构上比较这两种实现的性能。

习题87. 证明CompositeFastPathLock实现能保证互斥，但不是无饥饿的。

习题88. 在HCLHLock锁中，对于一个给定的集群主线程，在设置全局队尾引用和设置最后被拼接结点的tailWhenSpliced标志量之间，被拼接到全局队列的结点同时存在于本地队列和全局队列中。解释该算法为什么仍然是正确的。

习题89. 在HCLHLock锁中，如果在变为集群主和把本地队列成功地拼接到全局队列之间的时间间隔太短，将会出现什么情况？提出一种补救该问题的办法。

习题90. 为什么由HCLHLock锁的waitForGrantOrClusterMaster()方法所访问的State对象的域应该被原子地读和修改？给出HCLHLock锁的waitForGrantOrClusterMaster()方法的代码。在你的实现中是否需要使用compareAndSet()？如果是，那么能否不使用该方法来有效地实现呢？ 174 ↓ 175

习题91. 设计一个isLocked()方法，该方法能测试一个线程是否正在持有锁（但没有获得锁）。分别给出针对下面各种锁的实现：

- 任意testAndSet()自旋锁。
- CLH队列锁。
- MCS队列锁。

习题92.（难题）如果允许对锁使用读-修改-写操作，那么在第2章无死锁互斥的空间复杂度下界$\Omega(n)$的证明中，什么地方会出现错误？ 176

第8章

The Art of Multiprocessor Programming (Revised First Edition)

管程和阻塞同步

8.1 引言

管程是一种能将同步和数据结合在一起的结构化方法。与类将数据和方法封装为一个整体的概念相类似，管程将数据、方法和同步封装在一个模块包中。

模块的同步是非常重要的。假设应用包含两个线程，其中一个为生产者线程而另一个为消费者线程，它们通过一个共享的FIFO队列相互通信。我们可以让这两个线程共享两个对象：一个非同步队列和一个保护该队列的锁。生产者的程序结构大致如下：

```
mutex.lock();
try {
  queue.enq(x)
} finally {
  mutex.unlock();
}
```

然而，这种结构并不是一种行之有效的编程方式。假设队列是有界的，那么队列中如果不存在空闲位置，任何试图把数据项添加到满队列中的调用都不能继续执行。应该阻塞调用还是让其继续前进的决策取决于队列的内部状态，而这种内部状态对于调用者来说（应该）是不可知的。假设应用变成多个生产者或多个消费者，或者同时有多个生产者和消费者，情况将会变得更糟。每个这种线程都必须记录锁和队列的对象，仅当每个线程都遵循相同的锁约定时，应用才是正确的。

一种更合理的方法就是让每个队列来管理它自己的同步。队列自身有它自己的内部锁。当方法被调用时要获得这个锁，在方法返回时要释放这个锁。而并不要求每一个使用队列的线程都必须遵循一个繁琐的同步协议。如果一个线程试图将一个数据项添加到一个已满的队列中，那么enq()方法自身就能检测到该问题，并挂起调用者，当队列中有空间时再恢复调用者。

177

8.2 管程锁和条件

如第2章和第7章一样，Lock是保证互斥的基本机制。在同一时刻只有一个线程能够持有锁。当线程第一次持有锁时它就获得了这个锁。当线程停止持有锁时它则释放这个锁。管程将产生一系列方法，每个方法被调用时获得锁，而在方法返回时则释放锁。

如果一个线程无法立刻获得锁，那么它或者自旋，不断地测试所期望的事件是否发生，或者如果，暂时放弃处理器让另一个线程运行。[⊖]如果我们期望等待的时间较短，则采用在多处理器上自旋是一种有效的方式，其原因在于阻塞一个线程需要开销很大的操作系统调用。如果我们期望等待一个较长的时间间隔，那么阻塞是有意义的，因为一个正在旋转的线程没

⊖ 在其他地方我们把阻塞和非阻塞的同步算法区分开来，在那里意味着完全不同的东西：阻塞算法是指一个线程的延迟能够引起另一个线程延迟的算法。

有做任何事情但使处理器一直在忙。

例如，如果一个特定的锁被短暂地持有，那么正在等待该锁被释放的线程应该自旋，而正在等待着从空缓冲区中出队元素的消费者线程则应该阻塞，因为我们通常无法预测这种等待要持续多久，把自旋和阻塞结合起来往往是有意义的：对于正在等待出队元素的线程，可以先让它自旋一小段时间，如果延迟较长则切换为阻塞。阻塞在多处理器和单处理器上都可以使用，而自旋则只在多处理器上使用。

编程提示8.2.1　本书中的大部分锁都遵循图8-1所示的Lock接口。下面对Lock接口的方法加以解释：

- lock()方法将阻塞调用者直到它获得锁为止。
- lockInterruptibly()方法与lock()方法相同，但如果线程在等待时被中断则会产生一个异常。（参见编程提示8.2.2。）
- unlock()方法释放锁。
- newCondition()方法则是一个工厂，它能创建并返回一个与该锁相关的Condition对象（将在后面解释）。
- 当锁为空闲时，tryLock()方法将会获得锁，并立刻返回一个布尔值，以说明它是否已获得锁。该方法也可以使用一个超时时限来调用。

```
1  public interface Lock {
2    void lock();
3    void lockInterruptibly() throws InterruptedException;
4    boolean tryLock();
5    boolean tryLock(long time, TimeUnit unit);
6    Condition newCondition();
7    void unlock();
8  }
```

图8-1　Lock接口

178

8.2.1　条件

当线程在等待某个事件发生时，例如在等待另一个线程将一个数据项放入队列中，释放队列上的锁是一种很好的选择策略，因为否则的话，其他的线程将不能把所期望的数据项放进队列。当正在等待的线程释放了锁以后，需要一种方法来通知该线程在什么时候再次去尝试获得锁。

在Java的并发包（以及相关包，如Pthreads）中，暂时释放锁的能力是由Condition对象及其相关锁来提供的。图8-2描述了由java.util.concurrent.locks库所提供的Condition接口。每一个条件对象都与一个锁相关联，可以通过调用相应锁的newCondition()方法来创建条件。如果正在持有锁的线程调用了与该锁相对应的条件的await()方法，则该线程释放锁并把自己挂起，给其他线程以获得锁的机会。当调用线程被唤醒时，它将重新去获取锁，此时有可能与其他的线程发生竞争。

编程提示8.2.2　Java中的线程能被其他线程中断。如果一个线程在调用Condition的await()方法期间被中断，那么该调用将产生一个InterruptedException异常，对该中断的响应则依赖于应用程序（简单地忽略掉中断并不是好的编程方式）。

图8-2给出了一个例示。

```
1   Condition condition = mutex.newCondition();
2   ...
3   mutex.lock()
4   try {
5       while (!property) { // not happy
6         condition.await(); // wait for property
7       } catch (InterruptedException e) {
8           ... // application-dependent response
9           }
10      ... // happy: property must hold
11  }
```

图8-2 如何使用Condition对象

179

为了避免混乱，我们通常在实例代码中忽略了InterruptedException处理程序，尽管它们在实际代码中有可能是必需的。

和锁一样，Condition对象必须要以一种格式化的方法来使用。假设一个线程要等待其个特性满足。线程在持有锁的同时测试该特性。如果特性不满足，那么线程应调用await()来释放锁，然后休眠直到另一个线程唤醒它。要点如下：当线程被唤醒时无法保证特性是满足的。await()方法有可能出现假返回（即没有任何原因而返回），或者可能出现给条件发出信号的线程唤醒了太多的休眠线程。无论是因为哪种原因，线程都必须再次测试特性，如果发现特性仍然不满足，那么必须再次调用await()。

图8-3中所示的Condition接口是这种调用的几种演变形式。其中一些提供了能够给调用者指定最大挂起时间的能力，以及说明在线程等待过程中能否被中断的能力。当一个队列发生变化时，导致该队列发生变化的线程能够通知其他正在等待某个条件的线程。它通过调用signal()来唤醒一个正在等待条件的线程，而通过调用signalAll()来唤醒所有的等待线程。图8-4描述了管程锁的执行过程。

```
1  public interface Condition {
2    void await() throws InterruptedException;
3    boolean await(long time, TimeUnit unit)
4      throws InterruptedException;
5    boolean awaitUntil(Date deadline)
6      throws InterruptedException;
7    long awaitNanos(long nanosTimeout)
8      throws InterruptedException;
9    void awaitUninterruptibly();
10   void signal();      // wake up one waiting thread
11   void signalAll();   // wake up all waiting threads
12 }
```

图8-3 Condition接口：await()及其演变将会释放锁，放弃处理器，然后被唤醒并重新获取锁。signal()和signalAll()方法用于唤醒一个或多个等待线程

图8-5给出了采用显式锁和条件来实现一个有界FIFO队列的代码。Lock域是所有方法都必须要获得的锁。我们要对它进行初始化，从而维护一个用于实现Lock接口的类的实例。此处使用了ReentrantLock，这是一种由java.util.concurrent.locks包所提供的非常有用的锁类型。

正如8.4节所讨论的一样，这种锁是可重入的，持有这种锁的线程能够不用阻塞而重新获取它。

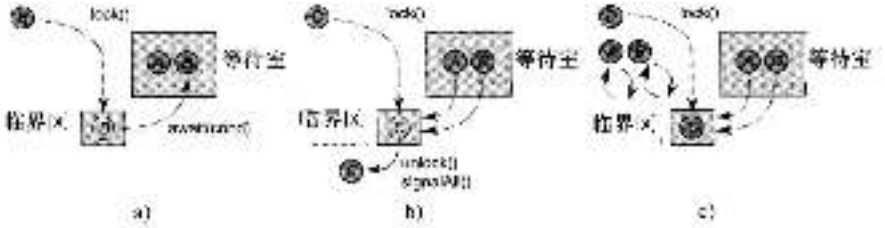


图8-4　管程的执行过程。在a中，线程A获得管程锁，调用了某个条件的await()而释放了锁，目前在等待室中。随后，线程B也执行同样的步骤，进入临界区，调用某个条件的await()，让出锁然后进入等待室。在b中，当C退出临界区并调用signalAll()后，A和B都离开等待室，然后尝试重新获得管程锁。然而，线程D设法首先获得了临界区锁，因此A和B都将自旋直到C离开临界区。注意，如果C调用signal()而不是signalAll()，那么A和B中只有一个能离开等待室，另一个则继续等待

```
1  class LockedQueue<T> {
2    final Lock lock = new ReentrantLock();
3    final Condition notFull = lock.newCondition();
4    final Condition notEmpty = lock.newCondition();
5    final T[] items;
6    int tail, head, count;
7    public LockedQueue(int capacity) {
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) {
11     lock.lock();
12     try {
13       while (count == items.length)
14         notFull.await();
15       items[tail] = x;
16       if (++tail == items.length)
17         tail = 0;
18       ++count;
19       notEmpty.signal();
20     } finally {
21       lock.unlock();
22     }
23   }
24   public T deq() {
25     lock.lock();
26     try {
27       while (count == 0)
28         notEmpty.await();
29       T x = items[head];
30       if (++head == items.length)
31         head = 0;
32       --count;
33       notFull.signal();
34       return x;
35     } finally {
36       lock.unlock();
37     }
38   }
39 }
```

图8-5　LockedQueue类：使用锁和条件的FIFO队列。有两个条件域，一个用于检测队列变为非空的情形，而另一个用于检测队列变为非满的情形

在实现中有两个条件对象：当队列从空变为非空时，通过notEmpty对象来通知正在等待的出队者，notFull对象则用于相反的过程。使用两个条件要比使用一个条件更为有效，因为这样将会减少不必要的线程唤醒，但这种方式比使用一个条件更为复杂。

这种将方法、互斥锁和条件对象组合在一起的整体称为管程。

8.2.2 唤醒丢失问题

正如锁本身容易产生死锁一样，Condition对象本身非常容易出现唤醒丢失问题。当发生唤醒丢失时，一个或多个线程一直在等待，而没有意识到它们所等待的条件已变为true。

唤醒丢失现象能够以很微妙的方式出现。图8-6给出了Queue<T>类的一种考虑不周的优化实现。在这个实现中，不是采用每当enq()从队列中入队一个数据项时都给notEmpty条件产生一个信号的方式，而是采用了仅当队列实际上从空变为非空时才给条件发出信号的方式。这样做是否更加高效呢？如果只有一个生产者和一个消费者，那么这种优化能够产生预期的效果，但如果有多个生产者或多个消费者，这样的优化产不正确。考虑下述场景：消费者A和B都试图从一个空队列中出队元素，它们检测到队列为空，于是都在notEmpty条件上阻塞。生产者C将缓冲区中的一个数据项入队，给notEmpty发出信号，唤醒了A，然而，在A获得锁之前，另一个生产者D把第二个数据项放入队列中。由于队列为非空，所以它不对notEmpty产生信号。于是A将获得锁，移走第一个数据项，而B却成为唤醒丢失的受害者。此时缓冲区中有一个等待消费的数据项，B却要永远地等待。

虽然不存在对我们的程序进行推理分析的可行办法，但有一些简单的实用编程技术却能使唤醒丢失最小化。

- 总是通知所有等待条件的进程，而不是仅仅通知一个。
- 等待时指定一个超时时限。

```
1   public void enq(T x) {
2     lock.lock();
3     try {
4       while (count == items.length)
5         notFull.await();
6       items[tail] = x;
7       if (++tail == items.length)
8         tail = 0;
9       ++count;
10      if (count == 1) {  // Wrong!
11        notEmpty.signal();
12      }
13    } finally {
14      lock.unlock();
15    }
16  }
```

图8-6 一个不正确的实例。该实例会发生唤醒丢失现象。仅当enq()方法是第一次向空缓冲区中放入数据项时，才对notEmpty产生一个信号。当多个消费者正在等待，但只有第一个被唤醒消费数据项时，唤醒丢失将会出现

这两种技术中的任一种都能限制上述的有界缓冲区错误，每种方法都需要一个小的性能开销，但与唤醒丢失的代价相比则可以忽略不计。

Java通过**synchronized**块和方法，以及内置的wait()、notify()和notifyAll()方法，

为管程提供了内置的支持。(参见附录A。)

8.3　读者-写者锁

许多共享对象都具有下述特性：大多数读者调用只返回对象的状态而不修改对象，只有少数写者调用才真正修改对象。

读者之间没有必要相互同步，它们对对象的并发访问完全是安全的。然而，写者必须锁住读者和其他的写者。读者-写者锁允许多个读者或单个写者并发地进入临界区。其接口如下：

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

该接口产生两个锁对象：读锁和写锁。它们满足下面的安全特性：

- 当任一线程持有写锁或读锁时，其他线程不能获得写锁。
- 当任一线程持有写锁时，其他线程不能获得读锁。

显然，多个线程可以同时持有读锁。

180 / 183

8.3.1　简单的读者-写者锁

下面考虑一系列由简单到复杂的读者-写者锁的实现。图8-7～图8-9描述了SimpleReadWriteLock类。该类使用一个计数器来记录已获得锁的读者的个数，同时采用一个布尔域指明是否已有写者获得锁。为定义相关的读-写锁，代码中使用了内部类（inner class），这是一种Java特性，它允许一个对象（SimpleReadWriteLock锁）创建可共享该对象私有域的其他对象（读-写锁）。readLock()和writeLock()方法都能返回实现这种Lock接口的对象。这些对象通过writeLock()对象的域进行通信。由于读-写锁方法彼此间必须同步，所以它们都在共同的SimpleReadWriteLock对象的Lock和condition域上同步。

```
1  public class SimpleReadWriteLock implements ReadWriteLock {
2    int readers;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public SimpleReadWriteLock() {
8      writer = false;
9      readers = 0;
10     lock = new ReentrantLock();
11     readLock = new ReadLock();
12     writeLock = new WriteLock();
13     condition = lock.newCondition();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
```

图8-7　SimpleReadWriteLock类：域和公共方法

```
21    class ReadLock implements Lock {
22      public void lock() {
23        lock.lock();
24        try {
25          while (writer) {
26            condition.await();
27          }
28          readers++;
29        } finally {
30          lock.unlock();
31        }
32      }
33      public void unlock() {
34        lock.lock();
35        try {
36          readers--;
37          if (readers == 0)
38            condition.signalAll();
39        } finally {
40          lock.unlock();
41        }
42      }
43    }
```

图8-8 SimpleReadWriteLock类：内部读锁

```
44    protected class WriteLock implements Lock {
45      public void lock() {
46        lock.lock();
47        try {
48          while (readers > 0 || writer) {
49            condition.await();
50          }
51          writer = true;
52        } finally {
53          lock.unlock();
54        }
55      }
56      public void unlock() {
57        lock.lock();
58        try {
59          writer = false;
60          condition.signalAll();
61        } finally {
62          lock.unlock();
63        }
64      }
65    }
66  }
```

图8-9 SimpleReadWriteLock类：内部写锁

8.3.2 公平的读者-写者锁

尽管SimpleReadWriteLock算法是正确的，但并不是令人满意的。通常情况下，读者要比写者频繁得多，这样的话，写者有可能被一系列连续的读者锁在外面很长时间。图8-10—图8-12中所示的FifoReadWriteLock类描述了一种给写者赋予优先级的方法。该类能保证一旦一个写者调用了写锁的lock()方法，则不允许有更多的读者能获得读锁，直到该写者获取并释放了该写锁为止。由于不再让读者进入，持有读锁的读者最终都将结束，写者将获得写锁。

```
1  public class FifoReadWriteLock implements ReadWriteLock {
2    int readAcquires, readReleases;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public FifoReadWriteLock() {
8      readAcquires = readReleases = 0;
9      writer   = false;
10     lock = new ReentrantLock(true);
11     condition = lock.newCondition();
12     readLock = new ReadLock();
13     writeLock = new WriteLock();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
21   ...
22 }
```

图8-10　FifoReadWriteLock类：字段和公共方法

```
23   private class ReadLock implements Lock {
24     public void lock() {
25       lock.lock();
26       try {
27         while (writer) {
28           condition.await();
29         }
30         readAcquires++;
31       } finally {
32         lock.unlock();
33       }
34     }
35     public void unlock() {
36       lock.lock();
37       try {
38         readReleases++;
39         if (readAcquires == readReleases)
40           condition.signalAll();
41       } finally {
42         lock.unlock();
43       }
44     }
45   }
```

图8-11　FifoReadWriteLock类：内部读锁类

readAcquires域记录了请求读锁的总次数，readReleases域记录了释放读锁的总次数。当这两个数量相等时，没有线程持有读锁。（当然，这里忽略了潜在的整数溢出和环绕问题。）该类有一个私有的lock域，读锁由读者持有一段较短的时间：它们获得锁，把readAcquires加1，然后释放锁。写者则从它们试图获得写锁直到释放写锁这段时间内都一直持有该锁。这种锁协议能保证一旦一个写者获得lock，则新增的读者都不能将readAcquires加1，所以其他

新增的读者不能获得读锁，最终当前持有读锁的所有读者都将释放它，从而让写者继续前进。

```
46    private class WriteLock implements Lock {
47      public void lock() {
48        lock.lock();
49        try {
50          while (writer) {
51            condition.await();
52          }
53          writer = true;
54          while (readAcquires != readReleases) {
55            condition.await();
56          }
57        } finally {
58          lock.unlock();
59        }
60      }
61      public void unlock() {
62        writer = false;
63        condition.signalAll();
64      }
65    }
```

图8-12 FifoReadWriteLock类：内部写锁类

当最后一个读者释放它的锁时如何通知正在等待的写者呢？当一个写者试图获取写锁时，它获得了FifoReadWriteLock对象的lock。一个释放读锁的读者也获得了那个lock。如果所有读者已经释放了它们的锁，释放读锁的读者则调用相关条件的signal()方法。

8.4 我们的可重入锁

若使用第2章和第7章中所描述的锁，一个试图重新获取它自己已持有的锁的线程将会使自己陷入死锁。当一个获取锁的方法嵌套调用另一个获取同一个锁的方法时，这种情形就会发生。

如果一个锁能被同一个线程多次获得，则称该锁是可重入的。现在来说明如何通过不可重入锁来构造可重入锁，该分析主要是为了说明如何使用锁和条件。实际上，java.util.concurrent.locks包已提供了可重入锁类，所以没有必要自己来写。

181
~
187

图8-13描述了SimpleReentrantLock类。Owner域保存着最后一个获得锁的线程的ID，每当获取锁时将holdCount域加1，每当释放锁时将holdCount域减1。当holdCount为零时锁为空闲。由于对这两个域的操作都是原子的，所以需要一个内部的短期锁。Lock域是由lock()和unlock()对域进行操作时所使用的锁，正在等待该锁变为空闲的线程则使用condition域。在图8-13中，内部lock域被初始化为SimpleLock类（幻影）的一个对象，该SimpleLock类被假定为不可重入的（第6行）。

lock()方法获取内部锁（第13行）。如果当前线程已经是锁的拥有者，那么它把保存的计数器加1并返回（第15行）。否则，如果保存的计数器不为零，该锁则被另一个线程所持有。调用者将释放锁并等待，直到绝条件发出信号为止（第20行）。当调用者被唤醒时，它仍必须继续检查保存的计数器是否为零。当保存的计数器为零时，调用线程则使它自己成为拥有者并将保存的计数器设置为1。

unlock()方法获取内部锁（第29行）。如果锁空闲或调用者不是拥有者，那么该方法产生一个异常（第31行）。否则，它把保存的计数器减1。如果保存的计数器为零，那么锁是空闲

的，于是调用者用信号通知条件来唤醒一个正在等待的线程（第35行）。

```
1  public class SimpleReentrantLock implements Lock{
2    Lock lock;
3    Condition condition;
4    int owner, holdCount;
5    public SimpleReentrantLock() {
6      lock = new SimpleLock();
7      condition = lock.newCondition();
8      owner = 0;
9      holdCount = 0;
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     lock.lock();
14     try {
15       if (owner == me) {
16         holdCount++;
17         return;
18       }
19       while (holdCount != 0) {
20        condition.await();
21       }
22       owner = me;
23       holdCount = 1;
24     } finally {
25       lock.unlock();
26     }
27   }
28   public void unlock() {
29     lock.lock();
30     try {
31       if (holdCount == 0 || owner != ThreadID.get())
32         throw new IllegalMonitorStateException();
33       holdCount--;
34       if (holdCount == 0) {
35         condition.signal();
36       }
37     } finally {
38       lock.unlock();
39     }
40   }
41
42   public Condition newCondition() {
43     throw new UnsupportedOperationException("Not supported yet.");
44   }
45   ...
46 }
```

图8.13　SimpleReentrantLock类：lock()和unlock()方法

8.5　信号量

如前所述，互斥锁能够保证在一个时刻只能有一个线程进入临界区。如果在临界区被占用期间任何一个线程想进入，那么该线程则阻塞，将自己挂起直到另一个线程通知它去重新尝试获得锁。Semaphore是互斥锁的一般化形式。每个Semaphore有一个容量，记为c。一个Semaphore并不是每次只让一个线程进入临界区，而是让至多c个线程进入，其中容量c是在Semaphore被初始化时所确定的。正如在本章注释中所讨论的，信号量是最早的同步形式之一。

图8-14所示的Semaphore类提供了两个方法：线程调用acquire()以请求获得进入临界区的许可，调用release()来宣布它正在离开临界区。Semahpore自身只是一个计数器，记录已被允许进入临界区的线程的个数。如果一个新的acquire()调用将要超出容量c，调用线程则被挂起直到有空间为止。当一个线程离开临界区时，它调用release()来通知一个正在等待的线程现在有空间了。

```
1  public class Semaphore {
2    final int capacity;
3    int state;
4    Lock lock;
5    Condition condition;
6    public Semaphore(int c) {
7      capacity = c;
8      state = 0;
9      lock = new ReentrantLock();
10     condition = lock.newCondition();
11   }
12   public void acquire() {
13     lock.lock();
14     try {
15       while (state == capacity) {
16         condition.await();
17       }
18       state++;
19     } finally {
20       lock.unlock();
21     }
22   }
23   public void release() {
24     lock.lock();
25     try {
26       state--;
27       condition.signalAll();
28     } finally {
29       lock.unlock();
30     }
31   }
32 }
```

图8-14 Semaphore的实现

8.6 本章注释

管程是由Per Brinch-Hansen[52]和Tony Hoare[71]发明的。信号量则由Edsger Dijkstra[33]所发明。McKenney[112]综述了不同类型的锁协议。

8.7 习题

习题93. 用Java的synchronized，wait()，notify()和notifyAll()构造代替版式的锁和条件来重新实现SimpleReadWriteLock类。

提示：必须指出内部读-写锁类的方法是如何锁住外部的SimpleReadWriteLock对象的。

习题94. 由java.util.concurrent.locks包提供的ReentrantReadWriteLock类不允许以读模式持有锁的线程再次以写模式来访问该锁（线程将会阻塞）。通过简示说明在这种设计方案中，如果允

许这种锁升级将会出现什么情况。

188 - 190

习题95.　一个储蓄账户对象具有非负结余，并且提供deposit(k)和withdraw(k)方法。deposit(k)方法使结余加k，如果结余至少为k，则withdraw(k)将从结余中减去k，否则被阻塞直至结余变为k或更多。

1. 用锁和条件实现该储蓄账户对象。
2. 现在假设有两种取款方式：普通的和优先的。设计一种实现，能保证一旦有优先的取款在等待处理则普通的取款不会进行下去。
3. 现在增加一个transfer()方法，它将总存款从一个账户转账到另一个账户：

```
void transfer(int k, Account reserve) {
  lock.lock();
  try {
    reserve.withdraw(k);
    deposit(k);
  } finally {lock.unlock();}
}
```

给定10个账户的一个集合，它们的结余是未知的。在1:00时，n个线程均设法把100美元从另一个账户转账到自己的账户。在2:00时，一个Boss线程给每个账户存1000美元。每个在1:00被调用的转账方法都一定会返回吗？

习题96.　在共享浴室问题中有两类线程，分别称为*male*（男性）和*female*（女性）。只有一个*bathroom*资源，必须以如下方式来使用：

1. 互斥：不同性别的人不能同时占用浴室。
2. 无饥饿：每个需要使用浴室的人最终会进入。

实现这个方法用到四个过程：enterMale()延迟调用者直至男性能进入浴室，leaveMale()在一个男性离开浴室时被调用。而enterFemale()和leaveFemale()则针对女性做相同的事。例如，

```
enterMale();
teeth.brush(toothpaste);
leaveMale();
```

1. 使用锁和条件变量实现这个类。
2. 使用synchronized、wait()、notify()和notifyAll()实现这个类。

对每一种实现，解释为什么满足互斥和无饥饿条件。

191

习题97.　Rooms类管理着一个编号从0至m（m是构造函数的一个参数）的房间集合。线程能进入或离开这个范围内的任一房间。每一个房间都能同时容纳任意数量的线程，但每次只有一个房间能被占用。例如，如果有两个房间，编号分别为0和1，那么可以有任意数量的线程可以进入房间0，但当房间0被占用时没有线程能够进入房间1。图8-15给出了Rooms类的概要。

```
1  public class Rooms {
2    public interface Handler {
3      void onEmpty();
4    }
5    public Rooms(int m) { ... };
6    void enter(int i) { ... };
7    boolean exit() { ... };
8    public void setExitHandler(int i, Rooms.Handler h) { ... };
9  }
```

图8-15　Rooms类

可以为每个房间分配一个出口处理程序（exit handler），调用setHandler(i,h)为房间i设置出口处理程序h。出口处理程序被最后一个离开房间的线程在任何随后的线程进入任一房间之前来调用。该方法被调用一次且当它正在运行时没有线程在任意一个房间中。

实现满足下列条件的Rooms类：

- 若某线程在房间i中，则没有线程在房间j（j≠i）中。
- 最后离开房间的线程调用房间的出口处理程序，并且当处理程序正在运行时没有线程在任何一个房间中。
- 实现必须是公平的：任何试图进入房间的线程最终都将成功。显然，可以假设每一个进入房间的线程最终都会离开。

习题98. 考虑一种具有主动和被动两类不同线程集合的应用，现要阻塞被动线程直至所有主动线程都允许被动线程继续前进。

CountDownLatch封装了一个计数器，初始化为主动线程的个数n。当一个主动的方法准备由被动线程执行时，它调用countDown()，使计数器减1。每个被动线程调用await()，阻塞该线程直至计数器为零（参见图8-16）。

```
1  class Driver {
2    void main() {
3      CountDownLatch startSignal = new CountDownLatch(1);
4      CountDownLatch doneSignal = new CountDownLatch(n);
5      for (int i = 0; i < n; ++) // start threads
6        new Thread(new Worker(startSignal, doneSignal)).start();
7      doSomethingElse();            // get ready for threads
8      startSignal.countDown();      // unleash threads
9      doSomethingElse();            // doing my time ...
10     doneSignal.await();           // wait for threads to finish
11   }
12   class Worker implements Runnable {
13     private final CountDownLatch startSignal, doneSignal;
14     Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
15       startSignal = myStartSignal;
16       doneSignal = myDoneSignal;
17     }
18     public void run() {
19       startSignal.await();      // wait for driver's OK to start
20       doWork();
21       doneSignal.countDown(); // notify driver we're done
22     }
23     ...
24   }
25 }
```

图8-16 CountDownLatch类：一个示例用法

给出一个CountDownLatch的实现。不用考虑CountDownLatch对象的重用。

习题99. 本题是习题98的后续。给出一个CountDownLatch的实现，使得CountDownLatch对象能被重用。

192 ! 195

链表：锁的作用

9.1 引言

第7章讲述了如何构建可扩展的自旋锁，这种自旋锁能保证即使在锁被频繁使用时也具有高效的互斥性。现在看来构建可扩展的并发数据结构是一件很简单的事：首先构造这个类的顺序实现，然后增加一个可扩展的锁域，并保证每个方法调用都应获取和释放这个锁。这种方式称为粗粒度同步。

通常，粗粒度同步的效果很好，但在某些重要的场合却并非如此。问题在于使用单一锁来协调控制所有方法调用的类，即使在锁本身是可扩展的情形下也并非总是可扩展的。当并发程度较低时，粗粒度同步的效果很好，但如果有很多线程试图同时存取一个对象，这个对象将变成一个顺序的瓶颈，从而使得线程必须排队等待。

本章介绍几种比粗粒度锁更优的实用技术，它们能有效地支持多个线程同时存取单一对象。

- *细粒度同步*：不再使用单一锁来解决每次对象存取的同步，而是将对象分解成一些独立的同步组件，并确保只有当多个方法调用试图同时访问同一个组件时才发生冲突。
- *乐观同步*：许多类似于树或链表这样的对象是由多个组件通过引用链接在一起所组成的。有一些方法用于查找该对象的特定组成部分（例如，一个具有特定关键字的链表或树的结点）。一种减少细粒度锁代价的办法就是在查找时无需获取任何锁。如果方法找到所需的部分，它就锁住该组件，然后确认该组件在被检测和上锁期间没有发生变化。这种技术只有在成功次数高于失败次数时才具有价值，这也是称之为乐观的原因。
- *惰性同步*：有时将较难的工作推迟完成是一种好的处理方式。例如，从一个数据结构中删除某个部分可以分为两个阶段：通过设置标志位来逻辑删除这个部分，然后再通过从数据结构中物理删除这部分来物理删除它。
- *非阻塞同步*：有时可以完全消除锁，依靠类似compareAndSet()的内置原子操作进行同步。

上述每种技术都可应用于（通过适当定制）多种通用数据结构。本章主要研究如何使用链表实现集合，这里是指不包含重复元素的集合。

为了实现此目标，如图9-1所示，一个集合应提供以下三种方法：

```
1  public interface Set<T> {
2    boolean add(T x);
3    boolean remove(T x);
4    boolean contains(T x);
5  }
```

图9-1 Set接口：add()方法将一个元素增加到集合中（如果该元素已经存在于集合中，那么该方法不产生任何影响），remove()方法删除一个元素（当该元素在集合中时），contains()方法返回一个布尔值，表示一个元素是否在集合中

- add(x)方法将元素x添加到集合中，当且仅当集合中原先不存在x时返回*true*。

- remove(x)方法将元素x从集合中删除，当且仅当集合中原先存在x时返回*true*。
- 当且仅当集合中包含元素x时contains(x)返回*true*。

对每一个方法，若对它的一次调用返回*true*，则称该调用是成功的，否则称为不成功的。在使用集合的应用中，contains()调用通常要比add()和remove()调用频繁得多。

9.2 基于链表的集合

本章给出了一系列并发集合算法，所有这些算法都基于同一个基本思想。集合是用结点组成的链表来实现的。如图9-2所示，Node<T>类有三个域，item域是实际的数据项，key域是数据元素的哈希码，结点按key值排序，以便供检测元素是否在链表中的有效方式，next域是指向链表中下一个结点的引用。（某些算法需要对这个类做一些技术上的改变，例如，增加新的域或改变已有域的类型。）为简单起见，假设每个元素的哈希码是唯一的（对这种假设的放松留作习题）。在本章的所有例子中，数据元素都是和同一个结点及key值相联系的，这样可以随意使用符号，用同一个符号来表示结点、它的key值以及它所代表的元素。例如，结点*a*的key为*a*，数据元素也是*a*，等等。

196

```
1  private class Node {
2    T item;
3    int key;
4    Node next;
5  }
```

图9-2 Node<T>类：该内部类包含元素，元素的key以及链表中的下一个结点。有些算法需要对这个类做些修改

链表具有两种类型的结点。除了包含集合元素的常规结点外，还使用了两个称为head和tail的哨兵结点作为链表的第一个和最后一个元素。哨兵结点不能被添加、删除或查找，它们的key值分别是最小的和最大的整数值。[⊖]图9-3的前一部分在暂不考虑同步的情形下描述了

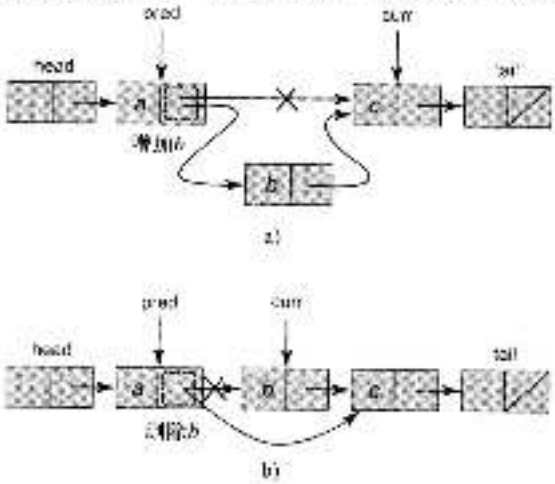


图9-3 Set的顺序实现：增加和删除结点。在图a中，线程使用两个变量来增加结点*b*，curr为当前结点，pred为前驱结点。线程顺着链表将curr的key与*b*相比较。如果找到一个匹配，说明元素已经存在于集合中，那么返回false。如果curr到达一个具有更大key值的结点，说明元素不在集合中。则让*b*的next域指向curr，pred的next域指向*b*。在图b中，要删除curr，线程将pred的next域设为curr的next

197

⊖ 这里给出的所有算法都适用于具有最小和最大且完整的关联序组成的有序集，也就是说，对于任意给定的关键字，只有有限多个关联序小于该关键字。例如，固定这些关键字为整数。

一个元素是如何被添加到集合中的。任意一个线程A具有两个用来遍历链表的变量：$curr_A$为当前结点，$pred_A$为当前结点的前驱结点。为了将一个元素添加到集合中，线程A将局部变量$pred_A$和$curr_A$设为head，然后顺着链表比较$curr_A$的key和被加入结点的key。如果匹配，说明该元素已经存在于集合中，所以调用返回false。如果$pred_A$领先$curr_A$，则$pred_A$的key比被插入结点的key小，$curr_A$的key比被插入结点的key大，所以该元素原来不在链表中。该方法创建一个新结点b来保存元素的值，设置b的next域指向$curr_A$，然后设置$pred_A$指向结点b。从集合中删除一个元素的操作采用类似的方法实现。

9.3　并发推理

并发数据结构的推理分析看似十分困难，但实际上是一种可以掌握的技巧。通常，掌握并发数据结构的关键就是理解其不变式：指一直保持的特性。可以通过证明下述性质来证明一个特性是不变的：

1. 对象被创建时该性质成立。
2. 一旦性质成立，则任何线程都不能使得该性质为*false*。

显然，大多数不变式在链表创建时都是成立的。所以，关键要关注一旦链表被创建后，不变式是如何保持的。

特别地，可以通过检查每次insert()、remove()和contains()方法调用时每个不变式都是成立的。这种方式只有在假设这些方法是唯一修改结点的途径时才是有效的，称这种性质为无干扰性。在本章的链表算法中，结点是链表的内部元素，由于链表用户无法修改内部结点，所以保证了无干扰性。

即使对于那些已从链表中删除的结点也需要无干扰性，因为有些算法允许在其他的线程遍历结点的同时，一个线程可以从链表中删除该结点。幸运的是，我们并不打算重用那些从链表中删除的结点，而是使用垃圾回收器回收这些存储器。本章的算法也适于不具备垃圾回收功能的语言，但有时需要一些重要的修改，这些修改超出了本章所讨论的范围。 198

在分析并发对象的实现时，理解对象的抽象表示（这里指元素的集合）和具体实现（这里指由结点构成的链表）之间的区别是非常重要的。

并非每种由结点构成的链表都能够很好地描述集合。一个算法的不变式说明决定了哪种说明作为抽象表示是有意义的。如果a和b为结点，当a的next域是一个指向b的引用时，则称a指向b。如果存在一个结点序列，从head开始，到b结束，其中的每个结点都指向它的后继结点，那么称结点b是可达的。

本章所描述的集合算法要求具备如下的不变式（有一些需要更多，将在后面解释）。首先，哨兵结点既不能增加也不能删除。其次，结点按关键字排序，且关键字值是唯一的。

我们将不变式表示为对象方法之间的契约。每个方法调用保持不变式，同时其他方法也保持不变式。采用这种方式，可以隔离地分析每个方法，而不用考虑它们之间所有可能的交互。

对于一个给定的满足不变式说明的链表，它描述的是哪一个集合呢？这种链表的含义由抽象映射所确定，抽象映射将满足不变式说明的链表映射为集合。此处，抽象映射非常简单：当且仅当一个元素是可达的，该元素属于该集合。

需要什么样的安全性和活性呢？我们的安全性是指可线性化性。正如第3章中所指出的，要证明一种并发数据结构是一个顺序对象的可线性化实现，只需说明一个可线性化点，即方法调用“生效”的那个原子操作步。这个操作可以是读、写或更复杂的原子操作。在基于链表的集

合的所有执行经历中，它必定是这种情形：如果抽象映射用于可线性化点的不变式说明，则状态和方法调用的结果序列将定义一个顺序的集合执行。这里，add(*a*)将*a*增加到抽象集合，remove(*a*)从抽象集合中删除元素*a*，contains(*a*)返回*true*或者*false*，取决于元素*a*是否在集合中。

不同的链表算法需要使用不同的演进保证。有些使用锁，但要注意确保无死锁和无饥饿特性。一些非阻塞算法根本不需要锁，而其他一些算法则将锁限制在特定的方法中。下面是从第3章开始所使用的非阻塞特性的简要概括[⊖]：

- 如果能保证一个方法的每次调用都在有限步内完成，那么这个方法就是无等待的。
- 如果能保证一个方法的某个调用总是能在有限步内完成，那么这个方法就是无锁的。

现在开始研究一系列基于链表的集合算法。首先从采用粗粒度同步的算法开始，然后改进这些算法以减小锁的粒度。形式化的正确性证明超出了本书范围。我们只关注解决日常问题的非形式化推理方法。

[199]

如前所述，在每一种算法中，方法使用两个局部变量扫描整个链表：curr为当前结点，pred为其前驱结点。这些变量都是线程的局部变量[⊜]，所以使用$pred_A$和$curr_A$表示线程A所用的实例。

9.4 粗粒度同步

首先分析一个使用粗粒度同步的简单算法。图9-4和图9-5描述了该粗粒度算法的add()方

```
1  public class CoarseList<T> {
2    private Node head;
3    private Lock lock = new ReentrantLock();
4    public CoarseList() {
5      head = new Node(Integer.MIN_VALUE);
6      head.next = new Node(Integer.MAX_VALUE);
7    }
8    public boolean add(T item) {
9      Node pred, curr;
10     int key = item.hashCode();
11     lock.lock();
12     try {
13       pred = head;
14       curr = pred.next;
15       while (curr.key < key) {
16         pred = curr;
17         curr = curr.next;
18       }
19       if (key == curr.key) {
20         return false;
21       } else {
22         Node node = new Node(item);
23         node.next = curr;
24         pred.next = node;
25         return true;
26       }
27     } finally {
28       lock.unlock();
29     }
30   }
```

图9-4 CoarseList类，add()方法

⊖ 第3章介绍了一种更强的无阻塞特性。

⊜ 附录A描述了Java中的局部变量的用法。

法和remove()方法。(contains方法基本上相同，留作习题。)链表本身具有一个锁，每个方法调用都必须要获取这个锁。该算法最大的优点就是其最为易见的正确性。所有的方法只有 200
在获取锁时才能对链表进行操作，所以执行实际上是串行的。为了简单起见，从此时起遵守这样一种约定，即任何试图获取锁的方法调用的可线性化点就是锁被获取的瞬间。

```
31    public boolean remove(T item) {
32      Node pred, curr;
33      int key = item.hashCode();
34      lock.lock();
35      try {
36        pred = head;
37        curr = pred.next;
38        while (curr.key < key) {
39          pred = curr;
40          curr = curr.next;
41        }
42        if (key == curr.key) {
43          pred.next = curr.next;
44          return true;
45        } else {
46          return false;
47        }
48      } finally {
49        lock.unlock();
50      }
51    }
```

图9-5　CoarseList类，remove()方法。所有方法都要获取同一个锁，该锁将在finally块退出时被释放

CoarseList类满足与它的锁相同的演进条件：如果lock是无饥饿的，则实现也是无饥饿的。如果竞争不激烈，那么该算法是实现链表的一种很好的方式。然而，如果竞争激烈，则即使锁本身非常好，线程也会延迟等待其他线程。

9.5　细粒度同步

可以通过锁定单个结点而不是整个链表来提高并发性。给每个结点增加一个Lock变量以及相关的lock()和unlock()方法，当线程遍历链表的时候，若它是第一个访问结点的线程，则锁住被访问的结点。在随后的某个时刻释放锁。这种细粒度的锁机制允许并发线程以流水线的方式遍历链表。

考虑两个结点a和b，其中a指向b。在对b上锁之前对a进行解锁是不安全的，因为在对a解锁和对b上锁期间，另一个线程有可能将b从链表中删除。然而，线程A必须以一种“交叉手”的方式来获取锁：除了初始的head哨兵结点外，只有在已获得$pred_A$的锁时，才能获得$curr_A$ 201
的锁。这种锁协议有时称为锁耦合。(注意不存在直接使用Java的synchronized方法来实现锁耦合的方式。)

图9-6描述了FineList算法的add()方法，图9-7是该算法的remove()方法。与粗粒度链表一样，remove()通过将$pred_A$的next域设为指向$curr_A$的后继，使得$curr_A$是不可达的。为了保证安全性，remove()必须锁住$pred_A$和$curr_A$。为了明白这样做的原因，考虑图9-8所示的场景。线程A准备删除链表中的第一个结点a，同时线程B准备删除b，其中a指向b。假设A锁住head，

B锁住a，然后A设置head.next指向b，而B设置a.next指向c。这样做的效果是删除a而不是删除b。问题在于两个remove()调用所获得的锁之间没有重叠。图9-9说明了“交叉手”上锁方式是如何避免这个问题的。

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

图9-6 FineList类：add()方法使用“交叉手”上锁来遍历链表。在返回之前，finally块释放锁

为了保证演进，所有的方法应以相同的次序获取锁，从head开始，顺着next引用一直到tail，如图9-10所示。如果不同的方法调用以不同的次序获得锁将导致死锁。在这个例子中，试图增加a的线程A已经锁住了b并试图去锁住head，同时试图删除b的线程B已锁住了head并试图锁住b。显然，这些方法调用将永远不会结束。避免死锁是使用锁编程的主要挑战之一。

202

FineList算法保持不变式说明：哨兵绝不会增加或删除，结点按key值排序且没有重复，抽象映射和粗粒度链表一样：当且仅当一个数据元素的结点可达，该数据元素属于集合。

add(a)调用的可线性化点依赖于该调用是否成功（即a是否已在链表中）。当具有下一个更大的key值的结点被锁定时（第7行或第13行），一个成功的调用（a不在链表中）是可线性化的。

remove(a)调用也存在同样的区别。当前驱结点被锁定时（第36行或第42行），一个成功的调用（a已存在）是可线性化的。当具有下一个更大的key值的结点被锁定时（第36行或第42行），一个成功的调用（a不存在）是可线性化的。当包含a的结点被锁定时，一个不成功的调用（a已存在）是可线性化的。

确定contains()的可线性化点留作习题。

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

清单9.7　FineList类：remove()方法在删除结点之前将欲删除结点及其前驱都锁定

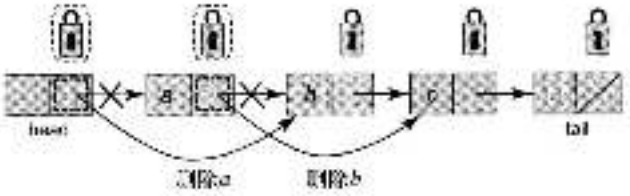
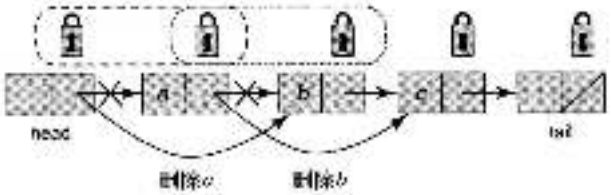


图9-8　FineList类：为什么remove()必须获得两个锁。线程A准备删除链表中的第一个结点a，同时线程B准备删除b，其中a指向b。假设A锁定head，B锁定a，然后A设置head.next指向b，而B设置a的next域指向c。这样做的效果是删除a，但没有删除b



清单9.9　FineList类：交叉于上锁能保证当并发的remove()调用试图删除相邻结点时，它们获得冲突锁。线程A准备删除链表中的第一个结点a，同时线程B准备删除b，其中a指向b。由于A必须同时锁住head和a，而B必须锁住a和b，从而保证它们在a上冲突，迫使一个调用等待另一个

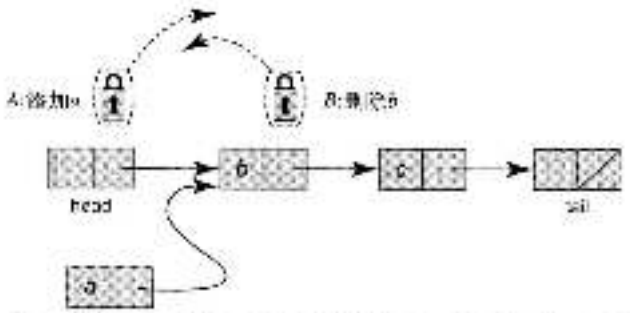


图9-10 FineList类：如果remove()和add()方法以相反的次序获取锁，则发生死锁。线程A准备插入a，它首先锁住b，然后锁住head，线程B准备删除b，它首先锁住head，然后锁住b，每个线程都持有对方等待获取的锁，结果是二者都不能继续前进

203 ~ 204

FineList算法是无饥饿的，但对这个特性的证明比在粗粒度情形要难。假设每一个锁都是无饥饿的。由于所有的方法以相同的顺着链表的次序获取锁，所以不会发生死锁。如果线程A试图锁定head并最终成功，从这个点开始，因为没有死锁发生，由A之前的其他线程获取的链表中的锁最终都会被释放，A将成功地锁住$pred_A$和$curr_A$。

9.6 乐观同步

虽然细粒度锁是对单一粗粒度锁的一种改进，但它仍可能出现很长的获取锁和释放锁的序列。而且，访问链表中不同部分的线程仍然可能相互阻塞，例如，一个正在删除链表中第二个元素的线程将会阻塞所有试图查找后继结点的线程。

减小同步代价的一种方法就是利用机遇：不需获得锁就可以进行查找，对找到的结点加锁，然后确认锁住的结点是正确的。如果一个同步冲突导致结点被错误地锁定，则释放这些锁并重新开始。在正常情况下，这样的冲突比较少，这也是称这种方法为乐观同步的原因。

在图9-11中，线程A执行了一个乐观的add(a)。在不获取任何锁的情况下遍历链表（第6行到第8行），事实上，该线程完全不管锁。当$curr_A$的key值大于或等于a的key值时，它停止查找，然后锁住$pred_A$和$curr_A$，并调用validate()以确认$pred_A$为可达的且它的next域仍指向$curr_A$。如果验证成功，则线程A和以前一样继续前进：若$curr_A$的key大于a，线程A则在$pred_A$和$curr_A$之间增加一个值为a的新结点，然后返回*true*。否则返回*false*。remove()和contains()方法（图9-12和图9-13）以同样的方式进行操作，遍历链表时无需上锁，然后锁住目标结点并验证它们仍在链表中。

validate()的代码如图9-14所示。下面这个故事带给我们一些启示：

> 一个旅行者在国外的一个城镇搭乘一辆出租车。出租车司机加速闯过一个红灯，这位旅行者惊恐地问道："为什么这样做？"司机回答道："别担心，我是个老手。"司机又加速闯过了几个红灯，这位旅行者几乎崩溃，再一次焦急地抱怨。司机回答说："放松，再放松，是一个老手在开车。"突然，绿灯亮了，司机急忙刹车，出租车打转停住，旅行者跳出出租车，大喊着问道："为什么在绿灯亮时停车？"司机回答说："太危险了，可能是另一个老手正在穿过路口。"

遍历一个动态变化的基于锁的数据结构而又不用锁需要慎重地考虑（还有其他的老手线

是也在那里），必须要使用某种形式的验证并保证无干扰性。

205

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4       Node pred = head;
5       Node curr = pred.next;
6       while (curr.key < key) {
7         pred = curr; curr = curr.next;
8       }
9       pred.lock(); curr.lock();
10      try {
11        if (validate(pred, curr)) {
12          if (curr.key == key) {
13            return false;
14          } else {
15            Node node = new Node(item);
16            node.next = curr;
17            pred.next = node;
18            return true;
19          }
20        }
21      } finally {
22        pred.unlock(); curr.unlock();
23      }
24    }
25  }
```

图9-11　OptimisticList类：add()方法在遍历链表时不需要锁，然后获得锁，并在增加结点之前进行验证

```
26  public boolean remove(T item) {
27    int key = item.hashCode();
28    while (true) {
29      Node pred = head;
30      Node curr = pred.next;
31      while (curr.key < key) {
32        pred = curr; curr = curr.next;
33      }
34      pred.lock(); curr.lock();
35      try {
36        if (validate(pred, curr)) {
37          if (curr.key == key) {
38            pred.next = curr.next;
39            return true;
40          } else {
41            return false;
42          }
43        }
44      } finally {
45        pred.unlock(); curr.unlock();
46      }
47    }
48  }
```

图9-12　OptimisticList类：remove()方法在遍历链表时不需要锁，然后获得锁，并在删除结点之前进行验证

206

```
49  public boolean contains(T item) {
50    int key = item.hashCode();
51    while (true) {
52     Node pred = this.head; // sentinel node;
53     Node curr = pred.next;
54     while (curr.key < key) {
55      pred = curr; curr = curr.next;
56      }
57     pred.lock(); curr.lock();
58     try {
59       if (validate(pred, curr)) {
60         return (curr.key == key);
61       }
62     } finally {                  // always unlock
63       pred.unlock(); curr.unlock();
64     }
65    }
66  }
```

图9-13 OptimisticList类：contains()方法在查找结点时不需要锁，然后获得锁，并验证结点是否在链表中

```
67  private boolean validate(Node pred, Node curr) {
68    Node node = head;
69    while (node.key <= pred.key) {
70      if (node == pred)
71        return pred.next == curr;
72      node = node.next;
73    }
74    return false;
75  }
```

图9-14 OptimisticList类：验证检查$pred_A$指向$curr_A$及$pred_A$是可达的

如图9-15所示，由于指向$pred_A$的引用或从$pred_A$指向$curr_A$的引用在它们最后被线程A读和线程A获得锁的这一段时间内，有可能已经发生变化，所以验证是必需的。在某些特殊情况下，一个线程有可能正在遍历已经从链表中删除的部分。例如，在线程A正在遍历$curr_A$的时候，结点$curr_A$以及$curr_A$和a（包括a）之间的所有结点有可能被删除。而线程A发现$curr_A$指向结点a，若没有验证，则即使a已经不在链表中也会“成功地”删除结点a。validate()调用将检查者结点a不在链表中，则由调用者重启该方法。

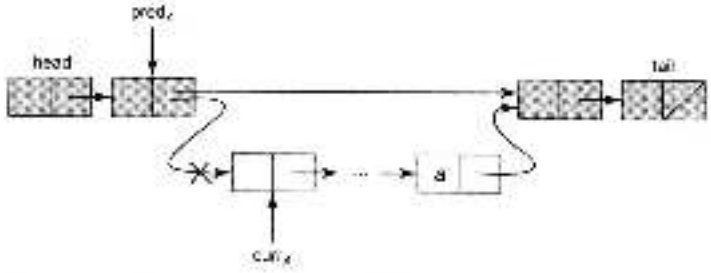


图9-15 OptimisticList类：为什么验证是必需的。线程A试图删除结点a。在遍历链表的同时，$curr_A$以及$curr_A$和a（包括a）之间的所有结点有可能被删除。在这种情形下，线程A有可能继续前进到达$curr_A$指向的a，若没有验证，即使a已经不在链表中，也会成功地删除a。需要验证来确定a是否是可达的

由于不无使用能保护并发修改的锁，所以每个方法调用都有可能遍历那些已被删除的结点。然而，由于无干扰意味着一旦一个结点从链表中删除，它的next域的值是不会改变的，所以按照这种链接的序列，最终仍可能回到链表中。而且，无干扰又依赖于垃圾回收来保证正在被遍历的结点不能重用。

即使每一个结点锁都是无饥饿的，OptimisticList算法也不是无饥饿的。如果不断地添加和删除新结点，那么一个线程就会被永远地阻塞（见习题107）。尽管如此，由于饥饿现象很少发生，所以仍期望该算法有很好的实际效果。

9.7　惰性同步

当不用锁遍历两次链表的代价比使用锁遍历一次链表的代价小许多时，OptimisticList实现的效果非常好。这种算法的缺点之一就是contains()方法在遍历时需要获得锁，这一点并不令人满意，其原因在于对contains()的调用要比对其他方法的调用频繁得多。

下面对该算法进行改进，使得contains()调用是无等待的。同时add()和remove()方法即使在被阻塞的情况下也只需遍历一次链表。对每个结点增加一个布尔类型的marked域，用于说明该结点是否在集合中。现在，遍历不再需要锁定目标结点，也没有必要通过重新遍历整个链表来验证结点是否可达，而是由算法维护一个不变式：所有未被标记的结点必是可达的。如果遍历线程没有找到结点或是发现结点已被标记，则该元素值不在集合中。总之，contains()只需要一次无等待的遍历。为了在链表中增加一个元素，add()首先遍历链表，锁住目标结点的前驱结点，最后插入该结点。remove()方法是惰性的，分两步进行：首先标记目标结点，逻辑上删除该结点；然后，重新指定其前驱结点的next域，物理上删除该结点。

207
/
208

```
1    private boolean validate(Node pred, Node curr) {
2      return !pred.marked && !curr.marked && pred.next == curr;
3    }
```

图9-16　LazyList类：验证检查pred和curr结点都没有被逻辑删除，且pred指向curr

更详细地说，所有方法不用锁就可以遍历链表（可能是逻辑上和物理上删除的结点），add()和remove()方法如以前一样锁住$pred_0$和$curr_0$结点（图9-16和图9-17），然而验证不再需要重新遍历整个链表（图9-18）来确定一个结点是否在集合中，相反，由于结点在被物理删除以前必须要作标记，所以验证只需确认$curr_0$还没有被标记，然而，如图9-19所示，对于插入和删除，由于$pred_0$结点是被修改的结点，所以必须验证$pred_0$本身没有被标记且它仍指向$curr_0$。逻辑删除需要对抽象映射做一点修改：当且仅当一个数据元素被一个未标记的可达结点指向时，该数据元素在集合中。需要注意的是，可达结点的路径中可能包含已标记的结点。读者应该验证任何未被标记的可达结点仍然是可达的，即使它的前驱已被逻辑或物理地删除。正如在OptimisticList算法中一样，add()和remove()方法不是无饥饿的，因为链表遍历有可能会被正在进行的修改任意延迟。

```
1  public boolean add(T item) {
2   int key = item.hashCode();
3   while (true) {
4     Node pred = head;
5     Node curr = head.next;
6     while (curr.key < key) {
7      pred = curr; curr = curr.next;
8     }
9     pred.lock();
10    try {
11      curr.lock();
12      try {
13        if (validate(pred, curr)) {
14          if (curr.key == key) {
15            return false;
16          } else {
17            Node node = new Node(item);
18            node.next = curr;
19            pred.next = node;
20            return true;
21          }
22        }
23      } finally {
24        curr.unlock();
25      }
26    } finally {
27      pred.unlock();
28    }
29  }
30 }
```

209

图9-17 LazyList类：add()方法

```
1  public boolean remove(T item) {
2    int key = item.hashCode();
3    while (true) {
4     Node pred = head;
5     Node curr = head.next;
6     while (curr.key < key) {
7      pred = curr; curr = curr.next;
8     }
9     pred.lock();
10    try {
11      curr.lock();
12      try {
13        if (validate(pred, curr)) {
14          if (curr.key != key) {
15            return false;
16          } else {
17            curr.marked = true;
18            pred.next = curr.next;
19            return true;
20          }
21        }
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28   }
29 }
```

图9-18 LazyList类：remove()方法分两步来删除结点：逻辑删除和物理删除

```
1    public boolean contains(T item) {
2      int key = item.hashCode();
3      Node curr = head;
4      while (curr.key < key)
5        curr = curr.next;
6      return curr.key == key && !curr.marked;
7    }
```

图9-19　LazyList类：contains()方法

contains()方法（图9-20）不用锁并遍历一次链表，如果被查找的结点已存在且未标记则返回*true*，否则返回*false*。所以该方法是无等待的。[⊖]已标记结点的值可以忽略。遍历每次都到达一个新结点，该新结点的key值总是比先前结点的key值大。即使该结点已被逻辑删除也是如此。

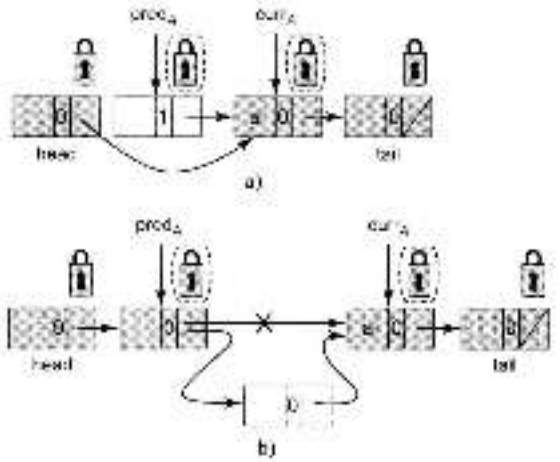


图9-20　LazyList类：为什么需要验证。在a中，线程A试图删除结点*a*。在它到达pred$_A$指向curr$_A$的地方且还未获得这两个结点的锁之前，结点pred$_A$被逻辑和物理地删除了。在线程A获得锁之后，验证将检测这个问题。在b中，A试图删除结点*a*。在它到达pred$_A$指向curr$_A$的地方且还未获得这两个结点的锁之前，有一个新结点被插入到pred$_A$和curr$_A$之间。在A获得锁之后，即使pred$_A$或是curr$_A$都没有被标记，验证也会检测到pred$_A$与curr$_A$是不同的结点，所以A的调用将被重启

逻辑删除需要对抽象映射做一点修改：当且仅当一个元素被一个未标记的可达结点指向时，该元素在集合中。注意，可达结点的路径中可能包含已标记的结点，链表的物理修改和遍历与OptimisticList类中完全一样，即使未被标记的结点的前驱被物理或逻辑地删除了，读者仍然需要验证未被标记的结点是可达的。

LazyList的add()和不成功的remove()调用的可线性化点与OptimisticList完全一样。当标记被设置时（第17行），一个成功的remove()调用是可线性化的。当找到未标记的匹配结点时，一个成功的contains()调用是可线性化的。

为了理解如何线性化一个不成功的contains()调用，考虑图9-21所描述的场景。在图a中，

⊖　对于一个给定的遍历线程，其中已被该线程遍历的部分不会因新key的插入而无限增大，其原因在于key的大小是有限的。

结点a被标记为已删除（设置其marked域），线程A试图查找与a的key值相一致的结点。当线程A正在遍历链表时，$curr_A$以及$curr_A$和a（包括a）之间的所有结点被逻辑或物理地删除，线程A仍然会继续前进到达$curr_A$指向a的地方，并验证a是被标记的且不在抽象集合中。该调用在这个点可以被线性化。

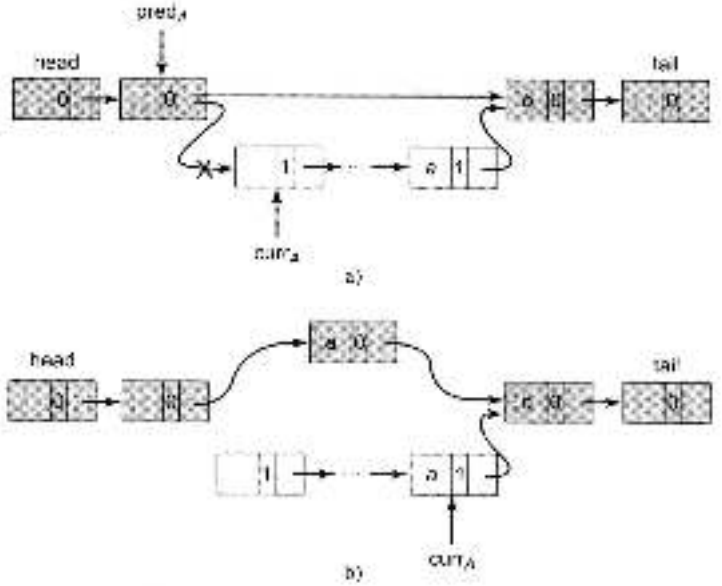


图9-21 LazyList类：线性化一个不成功的contains()调用。黑色结点代表实际在链表中的结点，而白色结点则是要被物理删除的结点。在图a中，线程A正在遍历链表，而并发执行的remove()调用则要断开由$curr_A$指向的子链表。由于数据值等于a和b的结点仍是可达的，所以一个结点是否真正在链表中只依赖于该结点是否已被标记。至此，线程A可以在发现a被标记且不再处于抽象集的时间点被线性化。再来考虑图b所示的情形，线程A正在遍历链表中已标记结点a的前面部分，另一个线程则欲增加一个key为a的新结点。若在线程A发现已标记结点a的时间点上线性化A的不成功的contains()调用，则可能出错，因为在这个时间点之前key值为a的新结点有可能已被插入到链表中

现在考虑图b描述的情景。A正在遍历链表中已删除的a前面的部分。在它到达被删除结点a之前，另一个线程将一个具有key值为a的新结点插入到链表的可达部分。如果在这个点线性化线程A的不成功的contains()方法，将发现被标记的结点a是错误的，因为这个点里现在具有key值为a的新结点被插入到链表以后。因此，对于一个不成功的contains()调用，要在它的执行过程中的以下几个时间点之前的时间段内来线性化这个调用：(1) 一个被删除的匹配结点，或者一个key值大于要查找结点的key值的结点被查找到；(2) 一个新的匹配结点被插入到链表之前的瞬间，注意，在执行过程中要保证第二个条件成立，这是因为具有相同key值的新结点的插入必须发生在contains()方法开始，或者contains()方法已经找到那个数据元素之后。正如所见的，不成功的contains()调用的可线性化点是由执行中事件的次序所决定的，并不是一个在方法代码中可以预先确定的点。

210
↕
212

惰性同步的优点之一就是能够将类似于设置一个flag这样的逻辑操作与类似于删除结点的链接这种对结构的物理改变相分开。这里给出的实例比较简单，其原因在于一个时刻只允许

解除一个结点的链接。然而，通常情况下，延迟操作可以是批处理方式进行的，且在某个方便的时候再谨慎地进行处理，从而降低了对结构进行物理修改的整体破裂性。

惰性同步的主要缺点是add()和remove()同用是阻塞的：如果一个线程延迟，那么其他线程也将延迟。

9.8　非阻塞同步

前述已知采用在物理删除链表中的某个结点之前将该结点标记为逻辑删除的思想有时是非常有益的。现在来研究如何扩展这种思想以完全消除锁，从而使add()、remove()和contains()这三个方法都变为非阻塞的。（前两个方法是无锁的，最后一个方法是无等待的。）

一种很自然的方法就是使用compareAndSet()来改变next域。不幸的是，这种方法并不适用。图9-22的后一部分描述了一个线程A试图在结点$pred_A$和$curr_A$之间插入结点a。它首先设置a的next域指向$curr_A$，然后调用compareAndSet()将$pred_A$的next域设为指向a。如果B要将$curr_B$从链表中删除，它可以调用compareAndSet()让$pred_B$的next域指向$curr_B$的后继结点。不难看出，如果这两个线程试图并发地删除这两个相邻的结点，那么结果是b没有被删除。关于并发的add()和remove()方法的类似情形则在图9-22的前一部分中进行了描述。

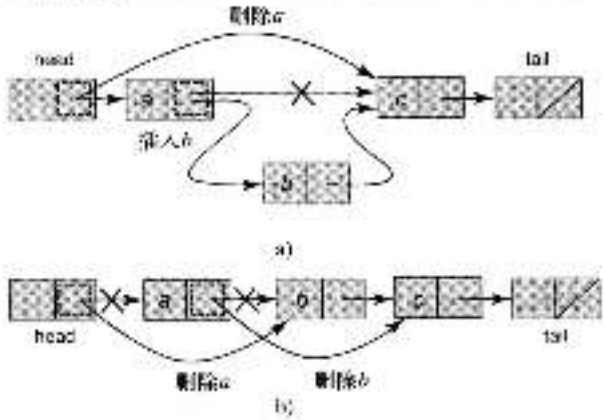


图9-22　LazyList类：为什么标记域和引用域必须原子地修改。在a中，线程A准备删除链表中的第一个结点a，同时线程B准备插入b。假设A对head.next调用compareAndSet()，同时B对a.next调用compareAndSet()。其结果是a被正确地删除，而b则没有加入到链表中。在b中，线程A准备删除链表中的第一个结点a，同时线程B准备删除a的后继结点b。假设A对head.next调用compareAndSet()，同时B对a.next调用compareAndSet()。其结果是a被删除而b未被删除

显然，需要一种方式来确保在结点被逻辑或物理删除后，该结点的域不能被修改。所采用的方法就是将结点的next域和marked域看作是单个的原子单位：当marked域为*true*时，对next域的任何修改都将失败。

编程提示9.8.1　AtomicMarkableReference<T>是java.util.concurrent.atomic包中的一个对象，它将一个对类型T的对象的引用和一个布尔型mark标志位一起。这些成员能够单个或一起被原子地更新，例如，compareAndSet()方法检测期望的引用和标记值，

如果两者都成立，则正更新后的引用和标记值来替换它们。简单来说，attemptMark()方法检测一个期望的引用值。如果测试成功，则用新的标记值来替换它。get()方法的读取与众不同：它返回对象的引用值并将标记值存入一个布尔数组的参数中。图9-23列出了这些方法的接口。

```
1  public boolean compareAndSet(T expectedReference,
2                               T newReference,
3                               boolean expectedMark,
4                               boolean newMark);
5  public boolean attemptMark(T expectedReference,
6                             boolean newMark);
7  public T get(boolean[] markHolder);
```

图9-23 AtomicMarkableReference<T>的部分方法。compareAndSet()方法测试并更新标记域和引用域，如果引用域为期望的值，attemptMark()方法则更新标记域。get()方法返回封装的引用并将标记存入参数数组的0号元素中

213
~
214

在C或C++中，可以通过使用位操作从一个字中取出标记和指针，从指针中"获取"一个位的方式，来有效地提供这种功能。在Java中，由于不能直接对指针进行操作，所以这种功能必须由库来提供。

正如编程提示9.8.1中所描述的，AtomicMarkableReference<T>对象将指向类型T的对象的引用和布尔量mark封装在一起，这两域可以一起或单个地被原子更新。

可以让每个结点的next域为一个AtomicMarkableReference<Node>，线程A通过设置结点next域中的标记位来逻辑地删除$curr_A$，和其他正在执行add()或remove()的线程共享物理删除：当每个线程遍历链表的时候，通过物理删除（使用compareAndSet()）所有被标记的结点来清理链表。换句话说，执行add()和remove()的线程不需要遍历被标记的结点，它们在继续执行以前删除了这些结点。contains()方法和在LazyList算法中一样，遍历所有被标记和未标记的结点，基于每个元素的key和mark检测元素是否在集合中。

现在来考虑一种与LazyList算法不同的LockFreeList算法的设计策略。为什么正在增加和删除结点的线程从不需要遍历被标记的结点，而是当遇到它们时物理地删除这些被标记的结点？假设线程A遍历被标记的结点而不是物理地删除它们，同样在逻辑删除$curr_A$以后，打算物理删除这个结点。这可以通过调用compareAndSet()来新设置$pred_A$的next域，同时确认$pred_A$没有被标记并且指向$curr_A$来实现。难点在于此时A并没有持有$pred_A$和$curr_A$上的锁，其他的线程可以在compareAndSet()调用之前插入新结点或删除$pred_A$。

考虑由另一个线程标记$pred_A$的情形，如图9-22所示，由于不能安全地重设被标记结点的next域，所以A将通过遍历链表来重新执行这个物理删除。然而此时，A要在删除$curr_A$之前不得不物理删除$pred_A$。更精确的是，如果存在一系列在$pred_A$前面的被逻辑删除的结点，那么A在删除$curr_A$本身之前，必须一个一个地将它们全部删除。

这个例子说明了为什么add()和remove()调用不需遍历被标记的结点：当它们到达要被修改的结点时，有可能需要重新遍历链表去删除原来已被标记的结点。所以，选择让add()和remove()在到达目标结点的路径上物理删除所有被标记的结点，相反，contains()方法不做任何修改，因此不需要参与到对逻辑删除结点的清理中，而是像在LazyList算法中一样，允

许它遍历所有被标记和未标记的结点。

为了给出LockFreeList算法，通过创建一个内部的Window类来将add()和remove()方法的公共部分分离出来。如图9.24所示，Window对象是一种具有pred和curr域的结构。Window类的find()方法以一个head结点和一个key值a为参数，查找并让pred指向具有比a小的最大key值的结点，让curr指向具有大于等于a的最小key值的结点。当线程A遍历链表时，每当它向前移动，就检查该结点是否被标记（第16行）。如果被标记，则调用compareAndSet()，通过置$pred_A$的next域指向$curr_A$的next域物理删除这个结点。这个调用既要检查域的引用又要检查布尔型的标记值，如果任意一个值发生了变化都将会失败。一个并发线程可以通过逻辑删除$pred_A$来改变标记值，或是通过物理删除$curr_A$来改变引用值。如果调用失败，A将从头结点开始重新遍历链表。否则，继续遍历。

```
1   class Window {
2     public Node pred, curr;
3     Window(Node myPred, Node myCurr) {
4       pred = myPred; curr = myCurr;
5     }
6   }
7   public Window find(Node head, int key) {
8     Node pred = null, curr = null, succ = null;
9     boolean[] marked = {false};
10    boolean snip;
11    retry: while (true) {
12      pred = head;
13      curr = pred.next.getReference();
14      while (true) {
15        succ = curr.next.get(marked);
16        while (marked[0]) {
17          snip = pred.next.compareAndSet(curr, succ, false, false);
18          if (!snip) continue retry;
19          curr = succ;
20          succ = curr.next.get(marked);
21        }
22        if (curr.key >= key)
23          return new Window(pred, curr);
24        pred = curr;
25        curr = succ;
26      }
27    }
28  }
```

图9.24　Window类：find()方法返回包含结点及其key任意一边的结构，当它遇到标记的结点时则删除它们

LockFreeList算法采用与LazyList算法相同的抽象映射：当且仅当一个元素值在一个未标记的可达结点中，该元素值在集合中。find()方法在第17行的compareAndSet()调用有这样一个很重要的作用：它改变了实际的链表但却没有改变抽象集合，因为删除一个被标记的结点并不改变抽象映射的值。

图9-25描述了LockFreeList类的add()方法。假设线程A调用add(a)，A使用find()来确定$pred_A$和$curr_A$。如果$curr_A$的key值等于a的key值，则调用返回*false*。否则，add()初始化一个新结点a来保存元素a，并让a指向$curr_A$。然后调用compareAndSet()（第11行）设置$pred_A$指向a。因为compareAndSet()同时检测标记和引用，所以仅当$pred_A$未标记且指向$curr_A$时才会

成功。如果compareAndSet()调用成功，该方法返回*true*，否则重新开始。

```
1  public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4     Window window = find(head, key);
5     Node pred = window.pred, curr = window.curr;
6     if (curr.key == key) {
7       return false;
8     } else {
9       Node node = new Node(item);
10      node.next = new AtomicMarkableReference(curr, false);
11      if (pred.next.compareAndSet(curr, node, false, false)) {
12        return true;
13      }
14    }
15   }
16  }
```

图9-25 LockFreeList类：add()方法调用find()以确定$pred_A$和$curr_A$。仅当$pred_A$为未标记且指向$curr_A$时，它增加一个新结点

图9-26描述了LockFreeList算法的remove()方法。当A调用remove()删除元素a时，它使用find()来确定$pred_A$和$curr_A$。如果$curr_A$的key值与a的key值不匹配，则调用返回*false*。否则，remove()调用compareAndSet()将$curr_A$标记为逻辑删除（第27行）。该调用仅当不存在其他线程已经先设置该标记的情形下成功。如果成功，调用返回*true*。对物理删除只是做一个简单的尝试，但没有必要再次尝试，因为下一个遍历链表中该部分的线程将会删除该结点。如果compareAndSet()调用失败，remove()方法将重新执行。

```
17   public boolean remove(T item) {
18    int key = item.hashCode();
19    boolean snip;
20    while (true) {
21     Window window = find(head, key);
22     Node pred = window.pred, curr = window.curr;
23     if (curr.key != key) {
24       return false;
25     } else {
26       Node succ = curr.next.getReference();
27       snip = curr.next.compareAndSet(succ, succ, false, true);
28       if (!snip)
29         continue;
30       pred.next.compareAndSet(curr, succ, false, false);
31       return true;
32     }
33    }
34   }
```

图9-26 LockFreeList类：remove()方法调用find()来确定$pred_A$和$curr_A$，并自动地将结点标记为已删除

LockFreeList算法的contains()方法实质上和LazyList相同（图9-27），只是有一点小的改变：为了测试curr是否已被标记，必须调用curr.next.get(marked)以确认marked[0]为*true*。

```
35    public boolean contains(T item) {
36      boolean[] marked = false;
37      int key = item.hashCode();
38      Node curr = head;
39      while (curr.key < key) {
40       curr = curr.next.getReference();
41       Node succ = curr.next.get(marked);
42      }
43      return (curr.key == key && !marked[0])
44    }
```

图9-27　LockFreeList类：无等待contains()方法实质上和LazyList类相同。只有一点区别：它调用curr.next.get(marked)以确认curr是否已被标记

9.9　讨论

本章讲述了基于链表的锁实现的发展变化，在这个演变过程中，锁的粒度和使用频率逐步地减小，最后得到了一个完全无阻塞的链表。从LazyList最终变为LockFreeList，为并发程序设计者提供了一些直接可用的设计策略。正如将要看到的，像乐观同步和惰性同步这样一些方法，在设计更复杂的数据结构时也经常被使用。

217 ~ 218

一方面，LockFreeList算法能够保证在面对任意的延迟时，线程可以继续演进。当然，这种演进保证需要一些代价：

- 对引用和布尔标记的原子修改需要额外的性能损耗。[9]
- 当add()和remove()遍历链表的时候，它们必须参与对已删除的结点的并发清理，从而导致线程之间可能发生争用，即使在每个线程试图修改的结点附近没有发生改变，有时也会使得线程重新遍历链表。

另一方面，基于锁的惰性链表在面对任意延迟时并不保证演进：它的add()和remove()方法正在执行，但是，与无锁算法不同，它并不要求每个结点具有原子的可标记引用，也不需要遍历链表来清除逻辑删除的结点，它仅顺着链表继续前进，不用考虑被标记的结点。

哪一种方法更加合适取决于应用。最后，对诸如任意线程延迟的可能、add()和remove()调用的相对频率、实现原子地可标记引用的代价等因素的综合平衡，决定了是否使用锁以及使用什么粒度的锁。

9.10　本章注释

锁耦合是由Rudolf Bayer和Mario Schkolnick[19]提出的。最早的无锁链表算法归功于John Valois[147]。本章所描述的无锁链表实现是Maged Michael[114]链表的一种变化形式，而他的工作是在早先由Tim Harris[53]提出的链表算法基础上开展的。所以，这种算法被大多数人称为Harris-Michael算法。Harris-Michael算法是Java的并发包中所使用的一种算法。OptimisticList算法是本章所提出的，而惰性算法则要归功于Steven Heller、Maurice Herlihy、Victor Luchangco、Mark Moir、Nir Shavit和Bill Scherer[55]。

[9] 例如，在Java Concurrent Package中，可以通过一个指向中间结点的引用来表明该标记位已经设置，以减少这种代价。

9.11 习题

219 习题100. 如果对象的哈希码不是唯一的，应如何修改每个链表算法。

习题101. 说明为什么细粒度锁算法不会产生死锁。

习题102. 说明为什么细粒度链表的add()方法是可线性化的。

习题103. 说明为什么乐观锁算法和惰性锁算法不会产生死锁。

习题104. 给出乐观算法中的一个场景，其中一个线程在永远试图删除一个结点。

提示：由于假设单个结点的锁都是无饥饿的，所以任意单个锁都不是活锁，一次不好的执行必定是不断地对链表增加和删除结点。

习题105. 写出细粒度算法中未给出的contains()方法的代码，并说明为什么你的实现是正确的。

习题106. 如果我们交换add()方法中锁定pred和curr的数据项的次序，乐观链表算法是否仍然正确？

习题107. 证明在乐观链表算法中，如果$pred_0$不为空，即使$pred_0$本身是不可达的，但从$pred_0$开始tail也是可达的。

习题108. 证明在乐观算法中，add()方法只需锁住pred结点。

习题109. 在乐观算法中，在决定一个key值是否存在之前，contains()方法需要锁住两个数据项。然而，现假设它不锁定任何数据项，如果找到值则返回*true*，否则返回*false*。

这种方法是否是可线性化的？若不是则给出一个反例。

习题110. 如果通过将一个结点的next域设为*null*来把一个结点标记为被删除的，那么惰性算法还是正确的吗？为什么？对无锁算法会怎样呢？

习题111. 在惰性算法中，$pred_0$是否有可能是不可达的？证明你的答案。

习题112. 你的新员工说惰性链表的验证方法（图9-16）能通过删除确验证pred.next域等于curr这一部分来简化，因为不管怎样，代码总是将pred设置为curr的旧值，在pred.next被改变之前，curr的新值必须被标记，从而导致验证失败。指出这个推理过程中的错误。

220 习题113. 你能修改惰性算法中的remove()方法，使得只需锁住一个结点吗？

习题114. 在无锁算法中，说明在清除被逻辑删除的结点时，使用contains()方法的优点和缺点。

习题115. 在无锁算法中，如只由于pred没有指向curr且pred未被标记而导致add()方法失败，那么为了完成本次调用，还需要再一次从head开始遍历链表吗？

习题116. 如只不保证逻辑删除的结点是已排好序的，那么惰性算法和无锁算法的contains()方法仍然是正确的吗？

习题117. 无锁算法的add()方法决不会找到一个具有相同key值的已标记结点。能否修改这个算法，使得如只链表中存在具有相同key值的结点，则只需简单地将要增加的新对象插入该结点中，从而不需要再插入一个新结点？

习题118. 解释为什么下述情形在LockFreeList算法中不会出现。一个具有数据值x的结点被某个线程逻辑地删除，但还未被物理地删除。这时，同一个数据值x被另一个线程增加到链表中。最后，第三个线程调用contains()方法遍历链表，发现了逻辑删除的结点，并返回*false*，即使
221 remove()方法和add()方法的可线性化次序表明x还在集合中。

并行队列和ABA问题

10.1 引言

在接下来的几章中，介绍一系列由称为池的对象所组成的类。池与第9章中讲述的Set类非常相似。但它们之间有两个不同点：不需要提供contains()方法来检测池的成员，允许同一个对象在池中多次出现。如图10-1所示，Pool只有get()方法和set()方法。在并发系统中，有许多地方都要用到池。例如，在大多数应用中，一个或多个生产者线程生产数据元素，一个或多个消费者线程使用所产生的数据。这些数据元素可以是需要执行的任务、要解释的键击输入、待处理的订单或需解码的数据包。有时生产者会突然加速，产生数据的速度远远超出消费者使用数据的速度。为使消费者能跟得上生产者，需要在生产者和消费者之间放置一个缓冲区，将那些来不及处理的数据先放在缓冲区中，以使得它们能被尽可能快地消费。池的作用往往与生产者-消费者缓冲区的作用相同。

```
1  public interface Pool<T> {
2    void put(T item);
3    T get();
4  }
```

图10-1 Pool<T>接口

池有以下几种不同的变化形式。

- 池可以是有界或无界的。有界池存放有限个数的元素，该界限称为池的容量。无界池可以存放任意数量的元素。当需要关注生产者和消费者之间的松池同步，即生产者不要过快地超过消费者时，就要用到有界池。有界池的实现要比无界池简单。当不需要设置固定的界限来限制生产者可比消费者快多少的程度时，就要用到无界池。
- 池的方法可以是完全、部分或同步的。
  - 若一个方法的调用不需要等待某个条件成立，则称该方法是完全的。例如，一个试图从空池中删除元素的get()调用将立刻返回错误码或者抛出一个异常。一个试图向满的有界池中添加一个元素的完全set()调用也会立即返回错误码或抛出异常，当生产者（或消费者）线程有比等待方法调用生效还要好的其他事情可处理时，完全接口非常有用。
  - 若一个方法的调用需要等待某个条件成立，则称该方法是部分的。例如，一个试图从空池中删除元素的部分get()调用将会阻塞，直到池中有可用元素才返回。一个试图向满的有界池中添加元素的部分set()调用将会阻塞，直到池中有一个可用的空槽插入元素为止。当生产者（或消费者）线程除了等待池变为非空（或非满）以外，没有其他更好的事情可做时，部分接口非常有用。
  - 若一个方法需要等待另一个方法与它的调用间隔相重叠，则称该方法是同步的。例如，在一个同步池中，一个向池中添加元素的方法调用将被阻塞直到该添加的元素被另一个方法调用取走。同样，一个从池中删除元素的调用将被阻塞直到另一个方法调用添加了这个可用于删除的元素。（这种方法也是部分的。）在CSP和Ada这类编程语言中，

同步池还可用于通信，线程可以通过会合（rendezvous）来交换信息。

- 池提供了各种不同的公平性保证。这些公平性包括先进先出（队列）、后进先出（栈）以及其他一些更公平性。在使用池作为缓冲区时，公平性显然是非常重要的。例如，任何一个人给银行或技术支持热线所打的电话，只是被放入一个呼叫服务等待池中。等待的时间越长，越让人恼火，但当他知道大家是按照先来先服务的方式在排队，那就会平静些了，只好无奈地等待。

10.2 队列

本章主要考虑一种能够支持先进先出公平性的池。一个顺序的Queue<T>是一个类型为T的元素所组成的有序序列。它提供enq(x)方法用于将对象x放入队列中称为tail的一端，提供deq()方法用于删除并返回队列中称为head的另一端的元素。并发队列可线性化为顺序队列，队列是一种由enq()实现put()而由deq()实现get()的池。我们使用队列实现来阐述一些重要
224 的原则，稍后的几章将考虑提供其他公平性的池。

10.3 部分有界队列

为简单起见，假定不允许向队列中增加*null*值。当然，在某些情形下，向队列中增加和删除*null*值是有意义的，对此问题的解决留作习题，可以通过改进算法使其允许*null*值。

一个有多个并发出队者和入队者的有界队列能提供多大程度的并行性呢？非形式化地来说，由于出队者和入队者分别在队列的两端进行操作，所以只要队列没有满或不为空，原则上来讲，enq()和deq()操作都可以无干扰地演进。出于同样的原因，并发的enq()有可能相互干扰，并发的deq()也可能相互干扰。这种非形式化的推理看起来很有道理，而且事实上在大多数情形下也是正确的，但是，达到这种级别的并行性并非易事。

下面采用链表来实现有界队列，（也可以使用数组。）图10-2描述了队列的域和构造函数，图10-3和图10-4描述了enq()和deq()方法，图10-5描述了队列的结点。与第9章所学的链表一样，一个队列结点包括value域和next域。

```
1  public class BoundedQueue<T> {
2    ReentrantLock enqLock, deqLock;
3    Condition notEmptyCondition, notFullCondition;
4    AtomicInteger size;
5    volatile Node head, tail;
6    int capacity;
7    public BoundedQueue(int _capacity) {
8      capacity = _capacity;
9      head = new Node(null);
10     tail = head;
11     size = new AtomicInteger(0);
12     enqLock = new ReentrantLock();
13     notFullCondition = enqLock.newCondition();
14     deqLock = new ReentrantLock();
15     notEmptyCondition = deqLock.newCondition();
16   }
```

图10-2 BoundedQueue类：域和构造函数

```
17    public void enq(T x) {
18      boolean mustWakeDequeuers = false;
19      enqLock.lock();
20      try {
21        while (size.get() == capacity)
22          notFullCondition.await();
23        Node e = new Node(x);
24        tail.next = tail; tail = e;
25        if (size.getAndIncrement() == 0)
26          mustWakeDequeuers = true;
27      } finally {
28        enqLock.unlock();
29      }
30      if (mustWakeDequeuers) {
31        deqLock.lock();
32        try {
33          notEmptyCondition.signalAll();
34        } finally {
35          deqLock.unlock();
36        }
37      }
38    }
```

图10-3 BoundedQueue类：enq()方法

```
39    public T deq() {
40      T result;
41      boolean mustWakeEnqueuers = false;
42      deqLock.lock();
43      try {
44        while (size.get() == 0)
45          notEmptyCondition.await();
46        result = head.next.value;
47        head = head.next;
48        if (size.getAndDecrement() == capacity) {
49          mustWakeEnqueuers = true;
50        }
51      } finally {
52        deqLock.unlock();
53      }
54      if (mustWakeEnqueuers) {
55        enqLock.lock();
56        try {
57          notFullCondition.signalAll();
58        } finally {
59          enqLock.unlock();
60        }
61      }
62      return result;
63    }
```

图10-4 BoundedQueue类：deq()方法

从图10-6可以看出，队列本身包含head域和tail域，分别指向队列的第一个结点和最后一个结点。队列总是包含一个作为空间占用者的哨兵结点，和第9章的哨兵结点一样，尽管它的值没有任何意义，但它标注了队列中的一个位置。然而，与第9章的链表算法所不同的是，第9章算法中的哨兵结点总是作为哨兵，而这里的队列不断地替换哨兵结点。我们分别使用两个不同的锁（enqLock和deqLock）来保证在一个时刻最多只有一个入队者和一个出队者可以

225
:
226

操作队列对象的域。这种采用两个锁而不是一个锁的方式能够保证入队者不会阻住出队者，反之亦然。每个锁都有一个与之相关的条件域。enqLock与notFullCondition条件相关联，当队列不再为满时，用来通知正在等待的入队者。deqLock与notEmptyCondition相关联，当队列不再为空时，用来通知正在等待的出队者。

由于队列是有界的，所以必须跟踪空槽的个数。size域是用来记录队列中并发对象个数的AtomicInteger，deq()调用将会减少该域的值，而enq()调用将增加该域的值。

```
64    protected class Node {
65      public T value;
66      public volatile Node next;
67      public Node(T x) {
68        value = x;
69        next = null;
70      }
71    }
72  }
```

图10-5 BoundedQueue类：链表结点

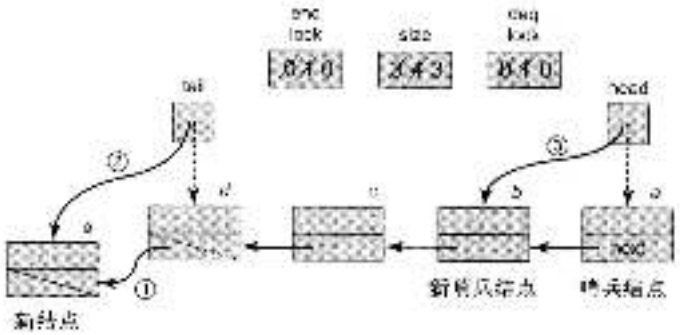


图10-6 具有4个结的BoundedQueue的enq()方法和deq()方法。首先，通过获取enqLock将一个结点插入到队列中。enq()检测队列的size为3，小于队列的界限。然后重新设置出tail域所指结点的next域（第一步），重设tail指向新结点（第二步），将size增加为4，并释放锁。由于size现在为4，任何新的enq()调用都将引起线程阻塞，直到某个deq()触发notFullCondition。接下来，某个线程将一个结点出队，deq()获得deqLock，从head所指结点（该结点此时为哨兵结点）的后继中读取新值b，重新设置head指向该后继结点（第二步），将size减为3，并释放锁。在deq()完成之前，由于在它开始时size为4，所以线程获得enqLock并对所有等待notFullCondition的入队者发出信号使它们能够继续执行

enq()方法（图10-3）按照如下方式工作。线程首先获得enqLock（第19行），然后读size域（第21行），如果该域等于队列的容量，则队列是满的，该入队者必须等待直到一个出队者产生空槽。入队者在notFullCondition域上等待（第22行），暂时释放enqLock锁，等待条件信号产生。每当入队者被唤醒，它就检查是否有空位，如果没有，则继续睡眠。

一旦空槽的个数大于0，入队者就会继续执行。注意一旦入队者发现有空槽，则当入队者还在继续执行时，其他线程不能向队列中插入元素，因为其他的入队者被锁定，只有一个并发的出队者可以增加空槽的数目。(enq()的同步相类似)。

必须仔细地检查在这种实现中不会出现第8章中所讲的"唤醒丢失"问题，这种细心检查是必需的。其原因在于入队者分两步来检测满队列：首先，发现size与队列容量相同；其次，它在notFullCondition上等待直到队列中出现空槽。当出队者将队列从满变为不满时，它获得enqLock并对notFullCondition发出信号。即使size域没有被enqLock保护，也由于出队者在触发条件之前已先获得enqLock，所以出队者不可能在入队者的两个操作步骤中间产生信号。

deq()方法按如下方式推进。首先读头元素的next域，然后检查哨兵的next域是否为*null*。如果是，则队列为空，出队者必须等待直到一个元素被插入队列。和enq()方法一样，出队者在notEmptyCondition上等待，暂时释放deqLock锁，然后阻塞直到条件被触发。每当出队者被唤醒，它就检查队列是否为空，如果是，则继续睡眠。

加象队列的头元素和尾元素并不总是与head和tail所指向的元素相同，理解这一点是非常重要的。一旦最后一个结点的next域重新指向新结点（enq()的线性化点），一个元素就被逻辑地插入到队列了，即使入队者还没有更新tail域也是如此。例如，一个线程可以持有enqLock的同时插入新结点，假设还没有更新tail域，一个并发的出队者线程可以获得deqLock，读并且返回新结点的值，重设head指向新结点，而所有这些操作都可在入队者重新设置tail指向新插入的结点之前发生。

一旦出队者锁定队列为非空，该队列将在这个deq()调用过程中一直保持为非空，因为所有其他的出队者已被锁定。考虑队列中第一个非哨兵结点（由哨兵结点的next域指向的结点）。出队者读这个结点的value域，设置队列的head域指向该结点，使该结点成为新的哨兵结点。然后出队者释放deqLock并将size减1。如果出队者发现原先的size值等于队列的容量，则可能有入队者正在等待notEmptyCondition，于是出队者获得enqLock并产生信号唤醒所有这样的线程。

这种实现的一个缺点就是并发的enq()和deq()相互干扰，但又不通过锁。所有的方法列size域调用getAndIncrement()或getAndDecrement()。这些方法比通常的读-写并销更大，且能引起顺序瓶颈。 227 / 228

减少这种干扰的一种方法就是将这个域分成两个计数器：一个由enq()增1的整型域enqSideSize和一个由deq()减1的整型域deqSideSize。调用enq()的线程检测enqSideSize，只要它小于队列的容量，就继续执行。当该域达到等于队列的容量时，线程锁住deqLock，并将deqSideSize加到enqSideSize中。当入队者的大小估值变得非常大时，这种技术能够分散地同步，而不是对每个方法调用进行同步。

10.4 完全无界队列

现在介绍另外一种队列，该队列能够存放数量不限的元素。enq()总是可以向队列中增加元素，如果队列中没有元素可以出队，deq()则抛出EmptyException。这种队列的描述与有界队列一样，但不需要保存队列中元素的个数，也不需要使用于等待的条件。如图10-7和图10-8所示，该算法要比有界的算法简单。

```
1   public void enq(T x) {
2     enqLock.lock();
3     try {
4       Node e = new Node(x);
5       tail.next = e;
6       tail = e;
7     } finally {
8       enqLock.unlock();
9     }
10  }
```

图10-7 UnboundedQueue<T>类：enq()方法

这种队列不可能死锁，因为每个方法只获得一个锁，或者enqLock或者deqLock。队列中唯一的哨兵结点永远不会被删除，所以只要每个enq()调用获得了锁就可以继续前进。当然，如果队列为空（如果head.next为*null*），则deq()有可能失败。与之前的有界队列实现一样，当enq()调用将最后一个结点的next域设置为指向新结点时，一个数据元素就被真正加入到队列中，即使在enq()重新设置tail指向新结点之前也是如此。在这个操作之后，新的结点顺着next的链是可达的，和通常情况一样，队列真正的头结点和

尾结点并不总是与head和tail所指向的结点相同。实际的头结点是head所指向结点的后继，而实际的尾结点是从头结点可达的最后一个元素。enq()和deq()都是完全的，因为它们都不需要等待队列变为空或变为满。

```
11  public T deq() throws EmptyException {
12      T result;
13      deqLock.lock();
14      try {
15        if (head.next == null) {
16          throw new EmptyException();
17        }
18        result = head.next.value;
19        head = head.next;
20      } finally {
21        deqLock.unlock();
22      }
23      return result;
24    }
```

图10-8 UnboundedQueue<T>类: deq()方法

10.5 无锁的无界队列

现在描述LockFreeQueue<T>类。这是一种无锁的无界队列实现，该类由图10-9至图10-11所描述。是10.4节中完全无界队列的自然扩展。该实现通过让较快的线程帮助较慢的线程来防止方法调用陷入饥饿。

```
1  public class Node {
2      public T value;
3      public AtomicReference<Node> next;
4      public Node(T value) {
5        this.value = value;
6        next = new AtomicReference<Node>(null);
7      }
8    }
```

图10-9 LockFreeQueue<T>类: 链表结点

和前面的做法一样，将队列表示为由结点所组成的链表。但是，如图10-9所示的那样，每个结点的next域是一个指向链表中下一个结点的AtomicReference<Node>。从图10-12可以看出，队列本身由两个AtomicReference<Node>域组成：head指向队列中的第一个结点，tail指向最后一个结点。队列中的第一个结点为哨兵结点，它的值是没有意义的。队列的构造函数将head和tail都设置为指向哨兵结点。

```
 9   public void enq(T value) {
10     Node node = new Node(value);
11     while (true) {
12       Node last = tail.get();
13       Node next = last.next.get();
14       if (last == tail.get()) {
15         if (next == null) {
16           if (last.next.compareAndSet(next, node)) {
17             tail.compareAndSet(last, node);
18             return;
19           }
20         } else {
21           tail.compareAndSet(last, next);
22         }
23       }
24     }
25   }
```

图10-10 LockFreeQueue<T>类: enq() 方法

```
25    public T deq() throws EmptyException {
27      while (true) {
28        Node first = head.get();
29        Node last = tail.get();
30        Node next = first.next.get();
31        if (first == head.get()) {
32          if (first == last) {
33            if (next == null) {
34              throw new EmptyException();
35            }
36            tail.compareAndSet(last, next);
37          } else {
38            T value = next.value;
39            if (head.compareAndSet(first, next))
40              return value;
41          }
42        }
43      }
44    }
```

图10-11 LockFreeQueue<T>类：deq()方法

enq()方法的一个有趣特点就是它是精简的：这种现象可以在两个不同的操作中出现。为了使该方法为无锁的，线程间有可能需要互相帮助。图10-12描述了这些步骤。

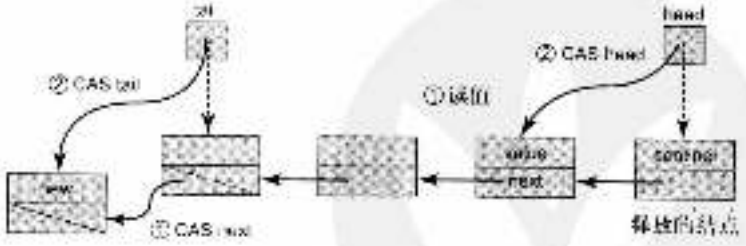


图10-12 LockFreeQueue的无锁精性enq()和deq()方法。结点分两步插入到队列中。首先，调用compareAndSet()将队列的tail所指向结点的next域从null改变为新结点。接着调用compareAndSet()让tail本身指向新结点。从队列中删除数据元素也分为两步。调用compareAndSet()从哨兵所指向的结点中读取数据项，然后将head从指向当前的哨兵结点改为指向哨兵的后继结点，使后者成为新的哨兵结点。enq()方法和deq()方法相互帮助完成未完成的tail更新

在下面的描述中，行号指图10-9到图10-11中标记的代码行。正常情况下，enq()方法创建一个新结点（第10行），定位到队列中最后一个结点（第12－13行），然后执行下面两步：

1. 调用compareAndSet()添加新结点（第16行），

2. 调用compareAndSet()将队列的tail域从原来的最后一个结点改变为当前的最后一个结点（第17行）。

由于这两个步骤都不是原子地执行的，所以所有其他的方法调用都要准备面对一个未完成的enq()调用，来完成添加结点的工作。这就是我们在第6章的通用构造中第一次看到的"帮助"技术的一个实例。

现在来详细地分析所有的步骤。一个入队者首先创建了一个包含要插入新元素的新结点（第10行），读tail，发现这个看似为最后一个结点的结点（第12～13行）。为了验证该结点的确是最后一个结点，它检查该结点是否有后继结点（第15行）。如果有，该线程则调用

compareAndSet()尝试添加新结点（第16行）。（compareAndSet()是必需的，因为其他线程也可能在做同样的事情。）若compareAndSet()成功返回，线程则第二次调用compareAndSet()使tail指向新结点（第17行）。即使第二个compareAndSet()调用失败了，线程也能成功返回，因为稍后将会看到，该调用只有在其他的某个线程已设置tail指向后继结点来"帮助"了本线程时才可能失败。如果最终结点有后继结点（第20行），则该方法通过在重新插入自己的结点之前让tail直接指向后继结点来尝试"帮助"其他线程（第21行）。该enq()是完全的，即它不用等待出队者。在正在执行的线程（或一个并发的帮助线程）调用compareAndSet()重新设置tail域指向新结点（第21行）的瞬间，一个成功的enq()可以在这个点被线性化。

deq()方法与UnboundedQueue中的完全deq()方法相类似，如果队列是非空的，出队者调用compareAndSet()将head从哨兵结点改为其后继，使后继结点变为新的哨兵结点。deq()方法采用和前面一样的办法来确认队列为非空：检查head结点的next域不为*null*。

然而，在无锁的情况下存在一个很微妙的问题，如图10-13所示：在向前移动head之前，必须确认tail没有指向将要被删除的哨兵结点。为了避免这个问题，我们增加一个测试：如果head等于tail（第32行）并且它们所指向结点（哨兵）的next域为非*null*（第33行），则认为tail落后了。与enq()方法中一样，deq()通过调整tail指向哨兵结点的后继来尝试帮助tail使其为一致的（第36行）。只有在这时才更新head以删除哨兵结点（第39行）。和部分队列一样，从哨兵结点的后继中读取值（第38行）。如果这个方法返回一个值，则它的线性化点为它完成一个成功的compareAndSet()调用的时刻（第39行），否则可以在第33行被线性化。

229
↓
232

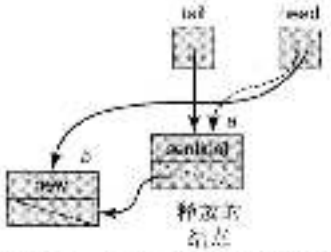


图10-13 为什么在图10-11的第36行中，出队者必须帮助推进tail。考虑这样的场景，其中一个正在向队列插入结点b的线程已经让a的next域指向b，但还没有将tail域从a变为b。如果另一个线程开始出队，它将读取b的值，并将head从a改为b，从而在tail还在指向a的时候，把a结点从队列中有效地删除了。为了避免这种情况，正在出队的线程必须在重设head之前帮助将tail从a推进到b

很容易说明队列是无锁的。每个方法调用首先找出一个未完成的enq()调用，然后尝试完成它。最坏的情形下，所有的线程都试图移动队列的tail域，且其中之一必须成功，仅当另外一个线程的方法调用在改变引用中获得成功时，一个线程才可能在入队或出队一个结点时失败，因此，总会有某个方法调用成功。这种无锁的实现从本质上改进了队列的性能，无锁算法的性能比最有效的互斥算法还要高。

10.6 内存回收和ABA问题

到现在为止，所有队列实现都依赖于Java的垃圾回收器来重复利用那些已经出队的结点。如果我们选择采用自己的内存管理，那么会出现什么样的情形呢？之所以这样做有以下几个原因，首先，像C和C++这些语言并不支持垃圾回收。其次，即使可以使用垃圾回收器，由类

本身来提供自己的内存管理也往往只有更高的效率，特别是在类创建和释放许多小的对象时。最后，若垃圾回收进程不是无锁的，则显然希望能提供自己的无锁内存回收。

以无锁方式循环结点的一种很自然的办法就是让每个线程维护它自己的由未使用队列项所组成的私有空闲链表。

233

```
ThreadLocal<Node> freeList = new ThreadLocal<Node>() {
 protected Node initialValue() { return null; }
};
```

当一个入队线程需要一个新结点时，它尝试从线程本地空闲链表中删除一个结点。如果空闲链表为空，则使用new操作分配一个结点。当一个出队线程准备释放一个结点时，它将该结点链入到线程本地空闲链表。因为链表是线程本地的，因此不需要很大的同步开销。只要每个线程的入队和出队次数大致相等，这种设计的效果就非常好。如果两种操作次数不平衡，则需要更加复杂的技术，例如定期从其他线程窃取结点。

令人惊讶的是，如果采用最直接的方式回收结点，那么这种无锁队列将会出错。考虑图10-14所描述的场景。在图a中，出队线程1发现哨兵结点为a，下一个结点是b。然后准备用旧值a和新值b调用compareAndSet()来修改head。在进入第一步之前，其他线程让b和它的后继结点相继出队，并将a和b放入空闲池。如图b所示，结点a被循环使用，并最终重新作为队列的哨兵结点。线程1在唤醒，调用compareAndSet()，由于head的旧值的确是a，所以成功返回。不幸的是，已经重设head指向了一个被回收的结点。

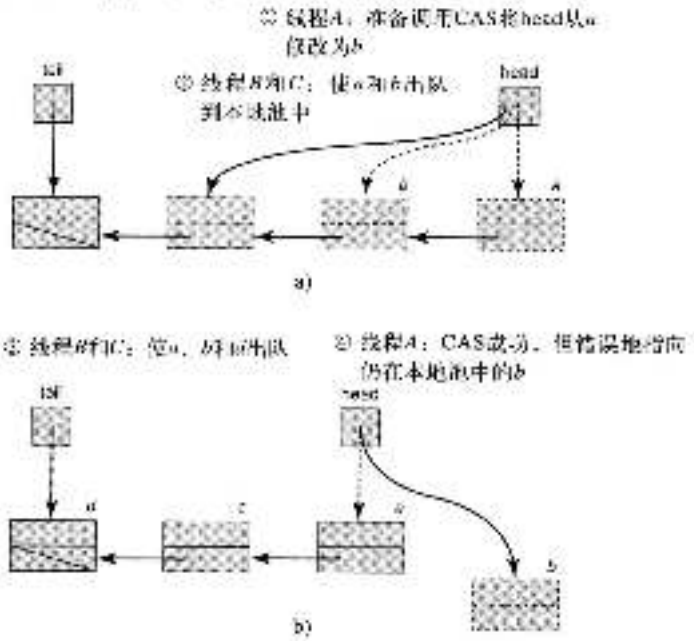


图10-14　一个ABA场景：假定在无锁队列算法中使用回收结点的本地池。在图a中，图10-11中的出队者线程A发现哨兵结点为a，下一个结点为b，（步骤1）然后准备用旧值a和新值b调用compareAndSet()来修改head。（步骤2）然而假定在执行第二步之前，其他线程让b和它的后继结点相继出队，并将a和b放入空闲池。在图b中，（步骤3）a结点被重新使用，并最终重新作为队列的哨兵结点。（步骤4）线程A现在唤醒，调用compareAndSet()，由于head的旧值的确是a，所以成功地将head置为b，现在head被错误地设置为指向一个被回收的结点

称这种现象为ABA问题。ABA现象经常出现，特别是在使用类似compareAndSet()这种条件同步操作的动态内存算法中。典型的情形是，一个将要被comapreAndSet()从a变为b的引用又被变回为a。这样一来，即使对数据结构的影响已经产生，compareAndSet()调用也将成功返回，但已不再是想要的结果。

解决这个问题的一种古怪办法就是对每个原子引用附上一个唯一的时间戳。如编程提示10.6.1中所述，AtomicStampedReference<T>对象将一个指向T类型对象的引用和一个整型stamp封装起来。这些域可以单独也可以同时被原子修改。

编程提示10.6.1 AtomicStampedReference<T>类将一个指向T类型对象的引用和一个整型stamp封装在一起。它扩展了AtomicMarkableReference<T>类（编程提示9.8.1），使其整型的时间戳来替代布尔型的mark。

通常用时间戳来避免出现ABA问题，每次修改对象时，就将时间戳的值加1，虽然有时就像第11章的LockFreeExchanger类一样，也用时间戳来存放一组有限状态集中的一个状态。

引用域和时间戳能够被原子更新，或者同时或者单独地更新。例如，compareAndSet()方法检测期望的引用值和时间戳值，如果两者都成功，则用要更新的引用值和时间戳值来替换它们。通常简述为：attemptStamp()方法测试期望的引用值，如果测试成功，则用一个新的时间戳来替换它。get()方法为一个很特别的接口：它返回对象的引用值并将时间戳值存放在一个整型参数数组中。图10-15描述了这些方法的签名。

234
-
235

```
1  public boolean compareAndSet( T expectedReference,
2                                T newReference,
3                                int expectedStamp,
4                                int newStamp);
5  public T get(int[] stampHolder);
6  public void set(T newReference, int newStamp);
```

图10-15 AtomicRefrence<T>类：compareAndSet()方法和get()方法。compareAndSet()方法检测并更新stamp和reference域，get()方法返回封装的reference并将stamp存放在参数数组的第0号位置。set()方法则更新封装的reference和stamp

在C/C++这些语言中，对于64位系统结构可以通过从指针中"窃取"位来有效地实现这种功能性，而对于32位系统结构则可能需要间接引用。

如图10-16所示，每次进入循环时，deq()读取第一个结点、下一个结点和最后一个结点的引用值和时间戳值（第6~8行），然后使用compareAndSet()同时比较引用和时间戳（第17行）。每次使用compareAndSet()更新引用时将时间戳加1（第14行和第17行）。[⊖]

在许多同步场景中，都会出现ABA问题，而不仅仅是那些包含条件同步的情形。例如，当仅使用加载/存储操作时也可能会出现ABA问题。对于那些条件同步操作，例如在有些系统结构中使用的链接加载/条件存储（见附录B），可以通过检测一个值在两个时间点之间是否被改变过，而不是检测这个值在两个时间点是否刚好相同的方式来避免ABA问题。

⊖ 我们忽略了时间戳会回绕而导致错误的可能性。

```
1   public T deq() throws EmptyException {
2     int[] lastStamp = new int[1];
3     int[] firstStamp = new int[1];
4     int[] nextStamp = new int[1];
5     while (true) {
6       Node first = head.get(firstStamp);
7       Node last = tail.get(lastStamp);
8       Node next = first.next.get(nextStamp);
9         if (first == last) {
10          if (next == null) {
11            throw new EmptyException();
12          }
13          tail.compareAndSet(last, next,
14              lastStamp[0], lastStamp[0]+1);
15        } else {
16          T value = next.value;
17          if (head.compareAndSet(first, next, firstStamp[0],
                firstStamp[0]+1)) {
18            free(first);
19            return value;
20          }
21        }
22      }
23    }
```

图10-16 LockFreeQueueRecycle<T>类：deq()方法使用时间戳来避免ABA问题

一种基本的同步队列

现在来考虑一种紧密相关的同步方式。一个或多个生产者线程生产数据元素，并由一个或多个消费者线程按照先进先出的次序取出。但是，这里的生产者和消费者之间必须相互会合：向队列中放入一个元素的生产者应阻塞直到该元素被另外一个消费者取出，反之亦然。这种会合同步在CSP和Ada语言中是内建的。

图10-17描述了SynchronousQueue<T>，这是一种基于管程的同步队列实现。它有如下几个域：item是第一个等待出队的元素，enqueuing是入队者用来在它们之间同步的布尔值，lock是用来互斥的锁，condition用于阻塞部分方法。如果enq()方法发现enqueuing为*true*（第10行），则表示另一个入队者已经提供了一个元素并正在等待与一个出队者会合，所以该入队者将会重复执行释放锁、睡眠、检查enqueuing是否为*false*（第11行）的操作。当条件满足，入队者将enqueuing设置为*true*，这将锁定其他入队者直到当前会合完成，并设置item指向新元素（第12～13行）。然后，通知所有的等待线程（第14行），并等待直到item变为*null*（第15～16行）。当等待结束时，会合已经发生，所以入队者设置enqueuing为*false*，通知所有的等待线程并返回（第17和19行）。

deq()方法简单地等待item不为空（第26～27行），记录该数据元素，将item设为*null*，然后在返回该元素之前通知所有的等待线程（第28～31行）。

由于这个队列的设计非常简单，所以它的同步代价也很高。在每个线程可能处理另一个线程的时间点，无论是入队者还是出队者都会唤醒所有的等待线程，从而唤醒的次数是等待线程数目的平方。尽管可以使用条件对象来减少唤醒次数，但由于仍需要阻塞每次调用，所以开销很大。

```
1  public class SynchronousQueue<T> {
2    T item = null;
3    boolean enqueuing;
4    Lock lock;
5    Condition condition;
6    ...
7    public void enq(T value) {
8      lock.lock();
9      try {
10       while (enqueuing)
11         condition.await();
12       enqueuing = true;
13       item = value;
14       condition.signalAll();
15       while (item != null)
16         condition.await();
17       enqueuing = false;
18       condition.signalAll();
19     } finally {
20       lock.unlock();
21     }
22   }
23   public T deq() {
24     lock.lock();
25     try {
26       while (item == null)
27         condition.await();
28       T t = item;
29       item = null;
30       condition.signalAll();
31       return t;
32     } finally {
33       lock.unlock();
34     }
35   }
36 }
```

图10-17 SynchronousQueue<T>类

10.7 双重数据结构

为了减少同步队列的同步开销，考虑另外一种同步队列的实现，它将enq()和deq()方法分成两步来完成。下面是出队者如何从一个空队列中删除元素的过程。第一步，它将一个保留对象放入队列，表示该出队者正在等待一个准备与之会合的入队者。然后，出队者在这个保留对象的flag标志上旋转。第二步，当一个入队者发现该保留时，它通过存放一个元素并设置保留对象的flag来通知出队者完成这个保留。同样，入队者能够通过创建自己的保留对象，并在保留对象的flag标志上旋转来等待会合同伴。在任意时刻，队列本身或者包含enq()的保留或deq()的保留，或者为空。

238

这种结构称为双重数据结构，其原因在于方法是通过两个步骤来生效的：保留和完成。该结构具有许多很好的性质。首先，正在等待的线程可以在一个本地缓存标志上旋转，而这是可扩展性的基础。其次，它很自然地保证了公平性。保留按照它们到达的次序来排队，从而保证请求也按照同样的顺序完成。注意这种数据结构是可线性化的，因为每个部分方法调用在它完成时是可以排序的。

该队列可以用结点组成的链表来实现，其中结点或者表示一个等待出队的元素或者表示

一个等待完成的保留（图10-18），由结点的type域指定。任何时候，所有的队列结点都应具有相同的类型：或者全部是在等待出队的元素，或者全部是等待完成的保留。

```
1   private enum NodeType {ITEM, RESERVATION};
2   private class Node {
3     volatile NodeType type;
4     volatile AtomicReference<T> item;
5     volatile AtomicReference<Node> next;
6     Node(T myItem, NodeType myType) {
7       item = new AtomicReference<T>(myItem);
8       next = new AtomicReference<Node>(null);
9       type = myType;
10    }
11  }
```

图10-18 SynchronousDualQueue<T>类：队列结点

当一个元素入队时，结点的item域存放该元素；当该元素出队时，结点的item域被重新设置为*null*。当一个保留入队时，结点的item域为*null*；当保留被一个入队者完成时，结点的item域被重新设置为一个元素。

图10-19描述了SynchronousDualQueue的构造函数及enq()方法（deq()方法和类似）。

```
1   public SynchronousDualQueue() {
2     Node sentinel = new Node(null, NodeType.ITEM);
3     head = new AtomicReference<Node>(sentinel);
4     tail = new AtomicReference<Node>(sentinel);
5   }
6   public void enq(T e) {
7     Node offer = new Node(e, NodeType.ITEM);
8     while (true) {
9       Node t = tail.get(), h = head.get();
10      if (h == t || t.type == NodeType.ITEM) {
11        Node n = t.next.get();
12        if (t == tail.get()) {
13          if (n != null) {
14            tail.compareAndSet(t, n);
15          } else if (t.next.compareAndSet(n, offer)) {
16            tail.compareAndSet(t, offer);
17            while (offer.item.get() == e);
18            h = head.get();
19            if (offer == h.next.get())
20              head.compareAndSet(h, offer);
21            return;
22          }
23        }
24      } else {
25        Node n = h.next.get();
26        if (t != tail.get() || h != head.get() || n == null) {
27          continue;
28        }
29        boolean success = n.item.compareAndSet(null, e);
30        head.compareAndSet(h, n);
31        if (success)
32          return;
33      }
34    }
35  }
```

图10-19 SynchronousDualQueue<T>类：enq()方法和构造函数

如之前讨论过的队列一样，head域总是指向一个哨兵结点，该结点作为一个空间占有者而存在，其实际值没有任何意义。当head和tail相一致时队列为空。构造函数创建一个具有任意值的哨兵结点，并让head和tail都指向该结点。

enq()方法首先检查队列是否为空或者是否包含等待出队的已入队元素（第10行）。如果条件满足，则像无锁队列一样，读队列的tail域（第11行），并确认读的值是一致的（第12行）。如果tail域没有指向队列的最后一个结点，则推进tail域并重新开始（第13--14行）。否则，enq()方法尝试通过重设尾结点的next域指向新结点，把新结点添加到队尾（第15行）。如果成功，就尝试着推进tail指向新添加的结点（第16行），然后旋转，等待一个出队者通过设置该结点的item域为*null*来通知该元素已经出队。一旦元素出队，该方法就尝试将它的结点设为哨兵结点来进行清理。最后一步仅仅用来提高性能，因为不管是否推进了head域，该实现总是正确的。

然而，如果enq()方法发现队列中有正在等待完成的出队者的保留，那么它就找出一个保留并完成。由于队列的head结点是一个其值无任何意义的哨兵结点，所以enq()读head的后继结点（第25行），确认读到的值是一致的（第26--28行），并试着将结点的item域从*null*改为要入队的元素。不管这一步是否成功，该方法都试着推进head（第30行）。如果compareAndSet()调用成功（第29行），则该方法返回，否则重试。

239
240

10.8 本章注释

部分队列综合运用了Doug Lea[98]提出的技术以及Maged Michael和Michael Scott[115]的算法中所采用的技术。无锁队列是Maged Michael和Michael Scott[115]所提出队列算法的一种简化版本。同步队列实现则来自Bill Scherer，Doug Lea和Michael Scott[136]的算法。

10.9 习题

习题119. 修改SynchronousDualQueue<T>类，使它能适用于*null*元素。

习题120. 考虑在第3章中所讲的针对单个入队者和单个出队者的简单无锁队列。图10-20描述了该队列。

这个队列是可组合的，即从一个空队列中删除一个元素或者向一个满队列中添加一个元素都会引起线程阻塞（旋转）。该队列的特殊之处在于它只需要加载/存储而不需要功能更加强大的读-改-写同步操作。是否需要使用内存路障？如果不需要，请解释原因，如果需要，请指出在代码中的哪个地方需要，为什么？

习题121. 设计一种使用数组而不是使用链表的基于锁的有界队列实现。

1. 允许为head和tail各使用一个锁的并行方式。

2. 试着将你的算法改为无锁的，在什么地方将会遇到困难？

习题122. 考虑图10-8中所描述的基于锁的无界队列的deq()方法。当检验队列为非空时，是否必须获得锁？为什么？

习题123. 在但丁的《炼狱》中，他描述了一次到地狱的旅行。在最近发现的一个章节中，他遇到了五个人，围着一张桌子坐着，桌子中央有一罐汤。尽管每个人都有一个勺子可以够到罐子，但每个勺子的手柄都长过了人的手臂，因此每个人都不能自己喝。他们都很饿并且很绝望。

但丁建议说："为什么你们不能喂其他人呢？"

余下的章节没有找到。

1. 写一个算法允许这些不幸的人互相喂食，两个或两个以上的人不能同时喂同一个人。你的算

法必须是无饥饿的。

2. 讨论你所设计算法的优缺点。它是集中式的还是分布式的？争用高还是争用低？确定的还是随机的？

```
1  class TwoThreadLockFreeQueue<T> {
2    int head = 0, tail = 0;
3    T[] items;
4    public TwoThreadLockFreeQueue(int capacity) {
5      head = 0; tail = 0;
6      items = (T[]) new Object[capacity];
7    }
8    public void enq(T x) {
9      while (tail - head == items.length) {};
10     items[tail % items.length] = x;
11     tail++;
12   }
13   public Object deq() {
14     while (tail - head == 0) {};
15     Object x = items[head % items.length];
16     head++;
17     return x;
18   }
19 }
```

图10-20 一种对于单入队者和单出队者具有阻塞语义的无锁FIFO队列。该队列在一个数组中实现。初始时head域和tail域相等且队列为空。如果head和tail的差值等于容量，则队列为满。enq()方法读head域。如果队列为满，则重复检查head，直到队列有空位置为止。接着将对象放入数组中，将tail域加1。deq()方法的工作过程与之相类似

习题124. 考虑无锁队列enq()方法和deq()方法的可线性化点。

1. 可以将返回值被从一个结点中读的时刻作为一个成功的deq()方法的可线性化点吗？
2. 可以将tail被更新（可能被其他线程）的时刻作为enq()方法的可线性化点吗（考虑若它发生在enq()执行过程中的情形）？讨论你的方案。

习题125. 考虑图10-21所示的无界队列实现。这个队列是可阻塞的，即直到deq()方法发现一个元素出队之前不会返回。

该队列有两个域：items是一个很大的数组，tail则是数组中下一个没有被使用的元素的索引。

1. enq()和deq()方法是无等待的吗？如果不是，那么是无锁的吗？请解释原因。
2. 找出enq()和deq()方法的可线性化点。（注意，它们可能与执行相关。）

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    ...
5    public void enq(T x) {
6      int i = tail.getAndIncrement();
7      items[i].set(x);
8    }
9    public T deq() {
10     while (true) {
11       int range = tail.get();
12       for (int i = 0; i < range; i++) {
13         T value = items[i].getAndSet(null);
14         if (value != null) {
15           return value;
16         }
17       }
18     }
19   }
20 }
```

图10-21 习题125中的队列

241
,
242
243

第11章

The Art of Multiprocessor Programming (Revised First Edition)

并发栈和消除

11.1 引言

Stack<T>类是一组数据项（类型为T）的集合，它提供了满足后进先出（LIFO）性质的push()方法和pop()方法：最后一个入栈的数据项最先出栈。本章讨论如何实现并发栈。初看起来，栈似乎不可能支持并发性，其原因在于push()和pop()调用似乎需要在栈顶同步。

然而令人不可思议的是，栈本身并不是顺序的，本章将阐述如何实现并发栈，它能获得高度的并行性。作为对问题研究的开始，首先来考虑如何构建一个无锁的栈，其中入栈和出栈操作需要在单一的单元上进行同步。

11.2 无锁的无界栈

例11-1描述了一个并发的LockFreeStack类，其代码分别由图11-2、列11-3和图11-4给出。该无锁栈是一个链表，其中top域指向链表的第一个结点（若栈为空则为*null*）。为简单起见，通常假定向栈中增加一个*null*是非法的。

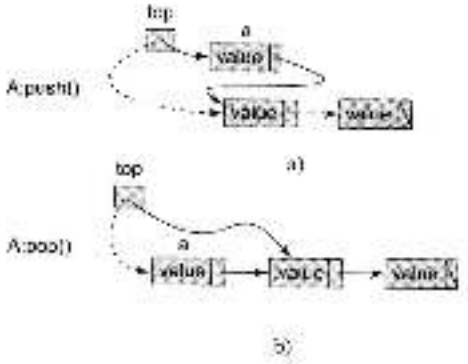


图11-1 无锁栈。在a中，一个线程通过对top域调用compareAndSet()将值a压入栈中。在b中，一个线程通过对top域调用compareAndSet()将值a从栈中弹出

试图从空栈中删除一个数据项的pop()调用将会抛出一个异常。push()方法首先创建一个新结点（第13行），然后调用tryPush()来尝试将top引用从当前的栈顶指向其后继。如果tryPush()成功，则push()调用将会返回，否则，在后退以后重新进行tryPush()尝试。pop()方法调用tryPop()，而tryPop()又使用compareAndSet()来尝试删除栈中的第一个结点。如果成功，则返回结点，否则返回*null*。（若栈为空则抛出一个异常。）tryPop()方法被不断地调用直到它成功为止，此时push()返回被删除结点的值。

如第7章中所述，使用指数后退（第7章图7-5）可以显著地减小在top域上的争用。因此，当调用tryPush()和tryPop()失败后，push()方法和pop()方法就进行后退。

```
1  public class LockFreeStack<T> {
2    AtomicReference<Node> top = new AtomicReference<Node>(null);
3    static final int MIN_DELAY = ...;
4    static final int MAX_DELAY = ...;
5    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
6
7    protected boolean tryPush(Node node){
8      Node oldTop = top.get();
9      node.next = oldTop;
10     return(top.compareAndSet(oldTop, node));
11   }
12   public void push(T value) {
13     Node node = new Node(value);
14     while (true) {
15       if (tryPush(node)) {
16         return;
17       } else {
18         backoff.backoff();
19       }
20     }
21   }
```

图11-2 LockFreeStack<T>类：在push()方法中，线程在调用tryPush()修改top引用和使用第7章图7-5的Backoff类进行后退之间交替执行

244 - 246

该实现是无锁的，因为仅当已有无限多次成功的调用修改了栈的top域时，线程才不能完成push()或pop()方法调用。push()和pop()方法的线性化点则是成功的compareAndSet()调用，或对空栈调用pop()方法抛出异常的时刻。注意pop()的compareAndSet()调用不会出现ABA问题（见第10章）。这是因为Java的垃圾回收器能够确保只要一个结点对另一个线程是可达的，则该结点就不会被同一个线程重用。如何设计一种不用垃圾回收器就能避免ABA问题的无锁栈则留作习题。

```
1  public class Node {
2      public T value;
3      public Node next;
4      public Node(T value) {
5        value = value;
6        next = null;
7      }
8  }
```

图11-3 无锁栈的链表结点

```
1    protected Node tryPop() throws EmptyException {
2      Node oldTop = top.get();
3      if (oldTop == null) {
4        throw new EmptyException();
5      }
6      Node newTop = oldTop.next;
7      if (top.compareAndSet(oldTop, newTop)) {
8        return oldTop;
9      } else {
10       return null;
11     }
12   }
13   public T pop() throws EmptyException {
14     while (true) {
15       Node returnNode = tryPop();
16       if (returnNode != null) {
17         return returnNode.value;
18       } else {
19         backoff.backoff();
20       }
21     }
22   }
```

图11-4 LockFreeStack<T>类：pop()方法在会试修改top域和后退之间交替执行

247

11.3 消除

上述LockFreeStack实现的可扩展性非常差，并不仅仅因为栈的top域是一个争用源，而主要因为这种栈是一个顺序瓶颈：方法调用只能一个接一个地前进，按照对top域的compareAndSet()成功调用次序来排序。

虽然指数后退能够有效地减少争用，但它并不能减轻顺序瓶颈的压力。为了使栈能够并行，我们利用栈的这样一种现象：如果一个push()后面紧跟着一个pop()，则这两个操作可以互相抵消，栈的状态不改变。这就好像两个操作从来没有发生过一样。如果能够采用某种办法使得并发的入队和出队操作对相互抵消，则正在调用push()的线程就可以在不改变栈本身的情况下与正在调用pop()的线程交换数据，我们称这样的两个调用相互消除。

如图11-5所示，线程通过EliminationArray来消除其他的线程，其中，线程随机地选取数组项来尝试发现互补的调用。互补的push()和pop()调用对则相互交换数值并返回。如果一个线程的调用不能消除，则或者是因为找不到一个配对调用，或者是因为所找配对的类型不对（例如，一个push()遇到一个push()），这种线程或者重新在一个新单元上再次尝试，或者访问共享的LockFreeStack。这种由数组和共享栈组成的数据结构是可线性化的，因为共享栈是可线性化的，并且可以排序被消除的调用，就好像它们发生在交换数值的时间点。

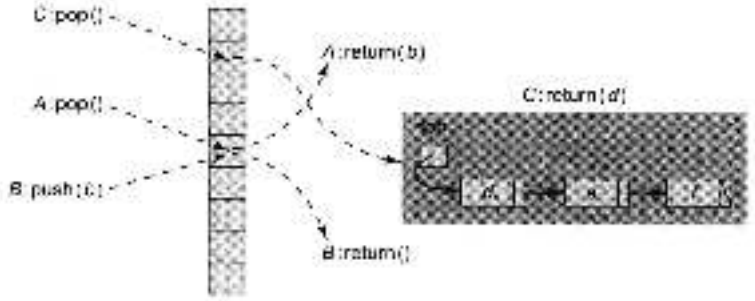


图11-5 EliminationBackoffStack类。每个线程在数组中随机地选择一个单元。如果线程A的pop()和线程B的push()几乎同时到达同一个单元，它们不必访问共享的LockFreeStack就可以交换值。线程C由于没有遇见另一个线程，最终将会对共享的LockFreeStack执行其栈操作

可以将EliminationArray作为一种在共享LockFreeStack上的后退模式。每个线程首先访问LockFreeStack，如果它的调用失败（即compareAndSet()失败），则该线程尝试使用数组而不是采用简单的后退来消除它的调用。如果没有能够消除它自己，则该线程再次访问LockFreeStack，等等。这种结构被称为EliminationBackoffStack。

248

11.4 后退消除栈

下面讲述如何构造EliminationBackoffStack，这是一种无锁可线性化的栈实现。

我们可以从一个故事中得到启发，这个故事讲述两个朋友在选举日讨论政治问题，每个人都想劝说对方改变立场，但都不能成功。

最后，其中一个人对另一个人说："瞧！既然我们在所有的政治问题上的观点都不相同，那么我们的选票自然也就互相抵消了，为什么不节省我们两个的时间，今天都不去投票呢？"

另一个人高兴地同意了，于是两个人分开了。

不久，第一个人的朋友听到这个谈话后对他说："你的这个建议很公平。"

"并不一定，"后者说，"这是我今天第三次这样做了。"

我们的构造中所采用的原理与这个故事是一样的，我们希望允许包含入栈和出栈操作的线程协商并抵消，但必须避免一个线程可以和多个线程达成约定的情形。为此，我们使用一种称为交换机的协调结构来实现EliminationArray，所谓交换机，就是只允许两个线程（不能再多）会合并交换数据的对象。

在第10章的同步队列中已学过如何通过锁来交换值，此处需要一种无锁交换，即交换时线程旋转而不是阻塞，因为希望它们只等待很短的时间。

11.4.1 无锁交换机

一个LockFreeExchanger<T>对象允许两个线程交换类型为T的值。如果线程A以参数a调用对象的exchange()方法，而线程B以参数b调用同一个对象的exchange()方法，则线程A的调用会返回值b，线程B的调用会返回值a。从一个更高的层次来看，交换机让第一个到达的线程写入自己的值，然后旋转等待直到第二个线程到来，随后，第二个线程检测到第一个线程在等待，于是读取第一个线程写入的值，并向交换机发出信号，现在两个线程都读取了对方的值，然后返回。如果第二个线程没有出现，第一个线程调用则可能会超时，从而使得若在一个合理的时间内无法完成交换，线程就能离开交换机并继续前进。 249

图11.6描述了LockFreeExchanger<T>类。它具有一个单一的AtomicStampedReference<T>类型的域slot。该交换机有三种可能的状态：EMPTY、BUSY或WAITING。该引用的时间戳记录了交换机的状态（第14行）。交换机的上循环在超时前一直执行，直到timeout抛出一个异常从而结束循环（第10行）。同时，一个线程读取slot的状态（第12行），并按如下流程进行处理：

- 如果状态为EMPTY，则该线程尝试将它的数据项放入槽中，并调用compareAndSet()（第16行）将状态设置为WAITING。如果失败，则说明某个其他线程成功，该线程重试。如果成功（第17行），则说明它的数据项在槽中且状态为WAITING，所以该线程旋转等待另一个线程完成交换。如果另一个线程出现，它将取出槽中的数据项，并用自己的数据项替换它，把状态设置为BUSY（第19行），通知等待的线程交换已经完成。等待的线程将消耗掉该数据项并把状态重新设置为0。对于empty()的重设只需要一个简单的写操作就可以了，因为等待的线程是唯一能将状态从BUSY变为EMPTY（第20行）的线程。如果没有其他线程出现，等待线程需要将槽的状态重新设置为EMPTY。这个状态改变需要调用compareAndSet()，因为其他线程有可能试图通过把状态从WAITING变为BUSY（第24行）来进行交换。如果这个调用成功，则产生一个超时异常。而如果失败，则说明某个正在交换的线程必定已出现，所以该等待的线程完成了交换（第26行）。
- 如果状态为WAITING，则说明某个线程正在等待且槽中包含它的数据项。该线程取出数据项，并试图通过调用compareAndSet将状态从WAITING变为BUSY来用它自己的数据项替换槽中的数据项（第34行）。如果有另外一个线程成功或按照一个定时重设状态为EMPTY，则该调用就会失败。如果是这样的话，该线程必须重试。如果它的确成功地将状态变为BUSY，则返回数据项。
- 如果状态为BUSY，则说明有两个其他的线程正在使用这个槽进行交换，因此该线程必须重试（第37行）。

```
1  public class LockFreeExchanger<T> {
2    static final int EMPTY = ..., WAITING = ..., BUSY = ...;
3    AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
4    public T exchange(T myItem, long timeout, TimeUnit unit)
5      throws TimeoutException {
6      long nanos = unit.toNanos(timeout);
7      long timeBound = System.nanoTime() + nanos;
8      int[] stampHolder = {EMPTY};
9      while (true) {
10       if (System.nanoTime() > timeBound)
11         throw new TimeoutException();
12       T yrItem = slot.get(stampHolder);
13       int stamp = stampHolder[0];
14       switch(stamp) {
15       case EMPTY:
16         if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17           while (System.nanoTime() < timeBound){
18             yrItem = slot.get(stampHolder);
19             if (stampHolder[0] == BUSY) {
20               slot.set(null, EMPTY);
21               return yrItem;
22             }
23           }
24           if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25             throw new TimeoutException();
26           } else {
27             yrItem = slot.get(stampHolder);
28             slot.set(null, EMPTY);
29             return yrItem;
30           }
31         }
32         break;
33       case WAITING:
34         if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
35           return yrItem;
36         break;
37       case BUSY:
38         break;
39       default: // impossible
40         ...
41       }
42     }
43   }
44 }
```

图11-6 LockFreeExchanger<T>类

注意，该算法允许插入的数据项为*null*，这将在稍后的消除数组结构中用到。该算法中不存在ABA问题，因为改变状态的compareAndSet()调用决不会检查数据项。一次成功交换的线性化点发生在第二个到达的线程将状态从WAITING改为BUSY的时刻（第34行）。在这个时间点，两个exchange()调用相重叠，所以保证交换一定成功。一次不成功交换的线性化点发生在抛出超时异常的时刻。

这个算法是无锁的，其原因在于仅当其他的交换不断地成功时，具有足够时间并相互重叠的exchange()调用才会失败。很显然，太短的交换时间可能导致其中一个线程决不会成功，因此必须十分小心地选取超时时限。

11.4.2 消除数组

EliminationArray类是作为一个最大容量为capacity的数组来实现的，数组中的数据项为Exchanger对象。准备进行一次交换的线程从数组中随机选择一个对象，并调用该对象的exchange()方法，从而保证将自己的输入作为与另一个线程交换的值。图11-7描述了EliminationArray的代码。其构造函数以数组的capacity（交换机的个数）作为参数。EliminationArray类提供了单一的方法visit()，它包含超时时限作为参数（参照java.util.concurrent包中的格式，超时时限用一个数字和一个时间单位来表示）。visit()调用以一个类型为T的值作为输入，或者返回它的交换伙伴的输入值，或者在没有和另一个线程交换值而出现超时的情况下抛出一个异常。在任意的时间点上，每个线程都会在数组的一个子集中随机地选择一个单元（第13行）。这个子范围是由数据结构的负载动态决定的，并作为参数传递给visit()方法。

```
1  public class EliminationArray<T> {
2    private static final int duration = ...;
3    LockFreeExchanger<T>[] exchanger;
4    Random random;
5    public EliminationArray(int capacity) {
6      exchanger = (LockFreeExchanger<T>[]) new LockFreeExchanger[capacity];
7      for (int i = 0; i < capacity; i++) {
8        exchanger[i] = new LockFreeExchanger<T>();
9      }
10     random = new Random();
11   }
12   public T visit(T value, int range) throws TimeoutException {
13     int slot = random.nextInt(range);
14     return (exchanger[slot].exchange(value, duration,
15             TimeUnit.MILLISECONDS));
16   }
17 }
```

图11-7 EliminationArray<T>类：每次访问时，线程可以动态地选择数组的子范围，在这个子范围中随机地选择一个槽

EliminationBackoffStack是LockFreeStack类的子类，它覆写了push()和pop()方法，并增加了一个EliminationArray域。列11-8和图11-9描述了新的push()和pop()方法。当一个tryPush()或tryPop()失败时，不再是简单的后退，而是尝试通过使用EliminationArray来交换值（第15行和第34行）。push()调用以它的输入值作为参数调用visit()，而pop()调用则以*null*作为参数调用visit()。push()和pop()都有一个线程本地的RangePolicy对象，用来决定EliminationArray的子范围。

当push()调用visit()时，它在自己的子范围内随机选择一个数组项，并尝试与其他线程进行交换。如果交换成功，则正在push的线程通过检查被交换的值是否为*null*来确认该值是否被一个pop()方法交换了（第18行）。（pop()总是把*null*交给交换机，而push()总是把一个非空值提交给交换机。）对称地，当pop()调用visit()时，它也试图交换数据，如果交换成功，则通过检查交换的值是否为非*null*来确认（第36行）该值是否被一个push()调用交换了。

交换也有可能不成功，或者是由于交换没有发生（对visit()的调用超时），或者交换发生在同类型的方法之间（一个pop()和另一个pop()交换）。为简单起见，采用一种简单的办法来处理这个问题：重新尝试tryPush()或tryPop()调用（第13行和第31行）。

250 ↗ 251
252

```
1  public class EliminationBackoffStack<T> extends LockFreeStack<T> {
2    static final int capacity = ...;
3    EliminationArray<T> eliminationArray = new EliminationArray<T>(capacity);
4    static ThreadLocal<RangePolicy> policy = new ThreadLocal<RangePolicy>() {
5      protected synchronized RangePolicy initialValue() {
6        return new RangePolicy();
7      }
8
9    public void push(T value) {
10     RangePolicy rangePolicy = policy.get();
11     Node node = new Node(value);
12     while (true) {
13       if (tryPush(node)) {
14         return;
15       } else try {
16         T otherValue = eliminationArray.visit
17                          (value, rangePolicy.getRange());
18         if (otherValue == null) {
19           rangePolicy.recordEliminationSuccess();
20           return; // exchanged with pop
21         }
22       } catch (TimeoutException ex) {
23         rangePolicy.recordEliminationTimeout();
24       }
25     }
26   }
27 }
```

图11-8 EliminationBackoffStack<T>类：该push()方法覆写了LockFreeStack类中的push()方法。它不再使用简单的Backoff类，而是使用一个EliminationArray和一个动态的RangePolicy来选择进行消除的数组子范围

```
28   public T pop() throws EmptyException {
29     RangePolicy rangePolicy = policy.get();
30     while (true) {
31       Node returnNode = tryPop();
32       if (returnNode != null) {
33         return returnNode.value;
34       } else try {
35         T otherValue = eliminationArray.visit(null, rangePolicy.getRange());
36         if (otherValue != null) {
37           rangePolicy.recordEliminationSuccess();
38           return otherValue;
39         }
40       } catch (TimeoutException ex) {
41         rangePolicy.recordEliminationTimeout();
42       }
43     }
44   }
```

图11-9 EliminationBackoffStack<T> 类：该pop()方法覆写了LockFreeStack中的push()方法

一个很重要的参数就是EliminationArray的范围选取，从这个范围中线程可选择一个Exchanger单元。一个小的范围将使得当线程个数很少时，冲突成功的机会较大，而一个大的范围则会降低在一个忙的Exchanger上线程等待的可能性（一个Exchanger一次只能处理一个交换）。这样，如果访问数组的线程很少，则应选择较小的范围；而当线程数增加时，范围也应增大。可以通过一个RangePolicy对象来动态地控制范围，该对象记录了成功交换的次数

(第37行)和超时失败的次数(第40行)。之所以忽略了由于类型不匹配所造成的交换失败(如push()和push()),是因为对于任何给定的push()和pop()调用的分布,这种情况所占的比例是固定的。一种简单的策略就是随着失败的次数增加而减小范围,反之亦然。

还有很多其他的策略。例如,可以设计一种更精巧的范围选取策略,在交换机上动态地改变延迟,在访问共享域前增加后退延迟,动态地控制是否访问共享栈或数组。这些设计都留作习题。

EliminationBackoffStack是一个可线性化的栈:任何通过访问LockFreeStack成功返回的push()和pop()调用都可以在访问LockFreeStack时被线性化。每一对被消除的push()和pop()调用可以在它们冲突时被线性化。正如前面介绍的,通过消除来完成的方法调用并不会影响这些方法在LockFreeStack中完成的可线性化性,因为它们可能已经在LockFreeStack的任意一个状态生效,且假如已经生效,LockFreeStack的状态并没有改变。

又因为EliminationArray是一种有效的后退模式,所以期望在低负载的情况下它的性能与LockFreeStack差不多。与LockFreeStack不同的是,EliminationArray具有扩展性,当负载增加时,成功消除的个数将会变大,从而允许很多操作并行地执行。此外,由于被消除的操作不会访问栈,所以在LockFreeStack上的争用也减少了。

11.5 本章注释

253
2
254

LockFreeStack应归功于Treiber[145],实际上它的出现时间要更早,可能为了促进IBM 370上的CAS运算于20世纪70年代早期就被提出来了。Danny Hendler、Nir Shavit和Lena Yerushalmi[57]则提出了EliminationBackoffStack。Doug Lea、Michael Scott和Bill Scherer[136]提出了一种高效的交换机,其令人感兴趣之处在于它采用一个消除数组。在Java的并发包中使用了这种交换机的一种变化形式。本章讲述的EliminationBackoffStack是一个采用交换机的模块,但是效率并不高。Mark Moir、Daniel Nussbaum、Ori Shalev和Nir Shavit提出了EliminationArray的一种高效实现[120]。

11.6 习题

习题126. 以链表为基础,设计一种基于锁的无界Stack <T>实现。

习题127. 采用数组设计一个基于锁的有界Stack <T>。

1. 使用单一的锁和有界数组。
2. 试着让你的算法成为无锁的。难点在哪里?

习题128. 修改11.2节中的无锁的无界栈,使其能在没有垃圾回收器的情况下正常工作。创建一个预分配结点的线程本地池,并回收它们。为了避免ABA问题,考虑使用java.util.concurrent.atomic的AtomicStanpedReference<T>类来封装引用和整型的时间戳。

习题129. 讨论我们的实现中所采用的后退策略。在LockFreeStack<T>对象中让入栈和出栈都使用同一个共享的Backoff对象是否有意义?如何从时间和空间上来在EliminationBackoffStack<T>中组织这种后退?

习题130. 实现一个栈算法,假定在执行的任何状态中,入栈数和出栈数之间的总数量差存在着一个界限。

习题131. 考虑采用一个由top计数器(初始化为0)索引的数组来实现一个有界栈时所存在的问题。在没有并发的情形下,不存在什么问题。若要压入一个数据项,将top加1来保留一个数组项,然后将数据项存入由该索引所指的位置。若要弹出一个数据项,则将top减1,并返回先前top所指位置的数据项。

显然,这种策略并不适用于并发实现,因为不能对多个内存单元进行原子地改变。一个单一的同步操作只能增加或者减少top计数器,但是不能同时进行两个操作。此外不存在原子地增

255 加计数器并且存入一个值的方法。

于是，Bob D. Hacker决定解决这个问题。他决定使用第10章的双重数据结构方法来实现一个双重栈。他的DualStack<T>类将push()方法和pop()方法分解为保留和完成两个步骤。如图11-10所示为Bob的实现。

```
1  public class DualStack<T> {
2    private class Slot {
3      boolean full = false;
4      volatile T value = null;
5    }
6    Slot[] stack;
7    int capacity;
8    private AtomicInteger top = new AtomicInteger(0); // array index
9    public DualStack(int myCapacity) {
10     capacity = myCapacity;
11     stack = (Slot[]) new Object[capacity];
12     for (int i = 0; i < capacity; i++) {
13       stack[i] = new Slot();
14     }
15   }
16   public T pop() throws EmptyException {
17     while (true) {
18       int i = top.getAndDecrement();
19       if (i <= 0) { // is stack empty?
20         throw new EmptyException();
21       } else if (i-1 < capacity)){
22         while (!stack [i-1].full ) {};
23         T value = stack[i-1].value;
24         stack[i-1].full = false;
25         return value ; //pop fulfilled
26         }
27       }
28   }
29   public T pop() throws EmptyException {
30     while (true) {
31       int i = top.getAndDecrement();
32       if (i < 0) { // is stack empty?
33         throw new EmptyException();
34       } else if (i < capacity - 1) {
35         while (!stack[i].full){};
36         T value = stack[i].value;
37         stack[i].full = false;
38         return value; //pop fulfilled
39       }
40     }
41   }
42 }
```

图11-10 Bob的有问题的双重栈

栈顶由top域所指向。这是一个只能通过getAndIncrement()和getAndDecrement()调用来
256 运行操作的AtomicInteger。Bob的push()方法的保留步骤是通过对top调用getAndIncrement()来保存一个槽。假设该调用返回索引i。如果i在范围0，capacity-1]内，则该保留完成。在完成阶段，push(x)将x存入数组的第i号单元，并设置full标识来表明准备读这个值。value域必须为volatile的，以保证一旦设置了flag标识，则读值已被写入数组的第i号单元。

如果从push()的getAndIncrement()返回的索引值小于0，那么push()方法不断地重新尝试getAndIncrement()直到它返回一个大于等于0的索引。该索引值可能小于0的原因在于对一个空栈的失败pop()调用的getAndDecrement()调用。每个这种失败的getAndDecrement()都可以对

于0界数组多次将top减1。如果返回的索引值大于capacity－1，则由于栈是满的，push()将引出一个异常。

pop()的情形与此相类似。它验证索引在界限内并通过对top调用getAndDecrement()删除数据项，返回索引i。如果i在范围[0, capacity－1]内，则该保留完成。在完成阶段，pop()在数组槽i的full标识上旋转，直到发现该标识位为*true*，表明push()调用是成功的为止。

Bob算法的错误在哪里？它是算法本身固有的问题吗？你能否想出一种办法来修改它？

习题132. 在习题97中，要求实现Rooms接口，如图11-11所示。Rooms类管理着一系列从0到*m*（其中*m*是一个已知的常数）索引的房间。线程可以进入或退出该范围内的任何一个房间。每个房间可以同时容纳任意数量的线程。但一个时刻只有一个房间能被占用，最后一个离开房间的线程触发一个onEmpty()处理程序。当所有的房间为空时该程序开始运行。

```
1  public interface Rooms {
2    public interface Handler {
3      void onEmpty();
4    }
5    void enter(int i);
6    boolean exit();
7    public void setExitHandler(int i, Rooms.Handler h) ;
8  }
```

图11-11 Rooms接口

图11-12描述了一种错误的栈实现。

```
1  public class Stack<T> {
2    private AtomicInteger top;
3    private T[] items;
4    public Stack(int capacity) {
5      top = new AtomicInteger();
6      items = (T[]) new Object[capacity];
7    }
8    public void push(T x) throws FullException {
9      int i = top.getAndIncrement();
10     if (i >= items.length) { // stack is full
11       top.getAndDecrement(); // restore state
12       throw new FullException();
13     }
14     items[i] = x;
15   }
16   public T pop() throws EmptyException {
17     int i = top.getAndDecrement() - 1;
18     if (i < 0) {             // stack is empty
19       top.getAndIncrement(); // restore state
20       throw new EmptyException();
21     }
22     return items[i];
23   }
24 }
```

图11-12 非同步的并发栈

1. 解释为什么这种栈实现不能正常工作。

2. 通过增加对一个双房间Rooms类的调用来进行修改：一个房间用于入栈，一个房间用于出栈。

习题133. 本习题是习题132的后续。这里不再让push()方法抛出一个FullException异常，而是利用下推房间的退出处理程序来调整数组的大小。注意，当一个退出处理程序正在执行时，线程不能在任何一个房间中，所以一个时刻只有一个退出处理程序可以运行。

第12章

The Art of Multiprocessor Programming (Revised First Edition)

计数、排序和分布式协作

12.1 引言

对于一些本身看似顺序的问题，如何通过将其协作任务“分散”在多个部件来使它们具有高度的并行性？这种分散又会给我们带来什么问题呢？这些是本章要讲述的主要问题。

为了回答这个问题，首先需要理解如何评测并发数据结构的性能。有两种评测指标：一种是时延，指完成一个单独的方法调用所需的时间；另一种是吞吐量，指完成所有方法调用的整体速率。例如，实时应用较关心时延，而数据库应用则更关心吞吐量。

第11章讲述了如何对EliminationBackoffStack类运用分布式协作。本章将介绍几种有用的分布式协作模式：组合、计数、衍射和筛子。有些是确定性的，而有些则采用随机的方式。另外，本章还将介绍这些协作模式下的两种基本数据结构：栈和组合树结构。有趣的是，对于某些基于分布式协作的数据结构，高吞吐量并不一定意味着高时延。

12.2 共享计数

回顾第10章中的概念，池是由元素组成的集合，它提供put()和get()方法来插入和删除元素（图10-1）。一些类似于栈和队列这种我们所熟悉的类可以被看作是提供了附加的公平性保证的池。

实现池的一种方式就是采用粗粒度锁，这也许是一种能使put()和get()同步的方法。然而，问题在于粗粒度锁过于笨拙，因为这种锁本身既会产生顺序瓶颈，从而迫使所有的方法调用同步；同时也会产生导致内存争用的热点。我们通常希望能让池的方法调用以并行方式工作，同时具有较少的同步和较低的争用。

259

下面考虑另一种方式。池中的元素均存放在循环数组中，每个数组项要么包含一个元素，要么为空。通过两个计数器来安排线程的路线。调用put()的线程对一个计数器加1以选择一个用来存放新元素的数组索引（如果该数组项不为空，则线程等待直到该数组项为空）。类似地，调用get()的线程将另一个计数器加1以选择一个删除新元素的数组索引（如果该数组项为空，则线程等待直到它不为空）。

这种方法采用两个瓶颈（计数器）来替换一个瓶颈（锁）。显然，两个瓶颈要比一个瓶颈好（想一想）。现在要寻求一种办法使得共享计数器不会成为瓶颈，而是能高效地并行工作。为此，我们面临下面两个挑战：

1. 必须防止内存争用，即许多线程试图访问同一个内存单元，从而增加底层网络通信和cache一致性协议的负担。

2. 必须获得真正的并行性。计数器加1本身是一个内在的顺序操作吗？n个线程对同一计数器各增加一次比一个线程对该计数器增加n次快吗？

下面介绍几种通过协调计数器索引分布的数据结构来构建高度并行的计数器的方式。

12.3 软件组合

首先介绍一种采用软件组合模式的可线性化共享计数器类。CombiningTree是一个由结点组成的二叉树，其中每个结点都包含簿记信息。计数器的值存放在根结点中。对每个线程分配一个叶结点，且最多只有两个线程可共享一个叶结点。因此，如果有p个物理处理器，则有$p/2$个叶结点。为增加计数器，一个线程从其叶结点开始，顺着路径向上到达树根。如果两个线程差不多同时到达一个结点，则通过将它们合在一起来组合它们的增量。其中的一个主动线程将它们的组合增量向上传递，而另一个被动线程则等待主动线程完成组合工作。一个线程可能在一个层上为主动的，而在另一个更高层上为被动的。

例如，假设线程A和B共享一个叶结点，它们同时开始，并在共享叶结点上组合。假设第一个线程B继续主动地到达下一个层，其任务是对计数器加2。而第二个线程A则被动地等待B从根结点返回确认通知，告知它的增加已经发生了。在树的下一个层次，B可能与另一线程C组合，它将继续前进并且将任务改为对计数器加3。 260

线程到达根结点时，将其组合增量的值与计数器的当前值相加形成新的计数器的值。然后，线程沿着原路返回，通知每个等待线程其增量已经完成。

组合树与锁相比有一个内在的缺点：每次增加都有一个较大的时延，即完成单个方法调用所花费的时间较长。对于锁来说，一个getAndIncrement()调用所需要的时间为$O(1)$，而对于CombiningTree来说，则需要$O(\log p)$的时间。然而，CombiningTree具有较高的吞吐量，即完成所有方法调用的总速率较高。例如，若使用队列锁，p个getAndIncrement()调用最好情况下需要$O(p)$时间，然而在使用CombiningTree时，理想情况下p个线程同时向上推进，p个getAndIncrement()调用只需要$O(\log p)$时间就能完成，这是一个指数级的改进。当然，实际情况要比理想情况差，此问题将在后面详细讨论。总之，像其他随后讨论的技术一样，CombiningTree类主要用来提高吞吐量而不是减少时延。

组合树的另一个优点是它能对被所维护的值进行任意的交换，而不仅仅是简单的递增。

12.3.1 概述

虽然CombiningTree的思路非常简单，但实现起来却并非易事。为了防止总体结构（简单的）被细节（不那么简单的）所掩盖，我们需要将数据结构分解为两个类：CombiningTree类管理着树内的导航，按照需要向上或向下移动，Node类则管理对结点的每次访问。在仔细研究该算法运用时，最好是参阅图12.3所给出的CombiningTree的一个执行实例。

该算法使用了两种类型的同步。短期同步由Node类的同步方法所提供。每个方法在其调用期间锁住结点，以确保在没有其他线程干扰的情况下读/写结点的域。算法还将从结点中排除那些延迟大于单独一个方法调用的线程。这种长期同步由一个布尔型locked域所提供。当这个域为真时，其他线程都不能访问该结点。

每个树结点都有一个组合状态，它定义了该结点是处于组合并发请求的早期、中期还是晚期阶段。

```
enum CStatus{FIRST, SECOND, RESULT, IDLE, ROOT};
```

这些值的意义如下：

- FIRST：一个主动线程已经访问了该结点，并且准备返回以检查是否有另一个被动线程

留下了一个要进行组合的值。

- SECOND：第一个线程已经访问了该结点，并在结点的value域中存放了一个值，准备与主动线程的值相组合，但是这个组合操作还未完成。
- RESULT：两个线程的操作已经组合并完成，且第二个线程的结果已经被存放在结点的result域中。
- ROOT：该值是一个特殊值，用于指明该结点为根，必须特殊对待。

图12-1描述了Node类的其他域。

```
1  public class Node {
2    enum CStatus{IDLE, FIRST, SECOND, RESULT, ROOT};
3    boolean locked;
4    CStatus cStatus;
5    int firstValue, secondValue;
6    int result;
7    Node parent;
8    public Node() {
9      cStatus = CStatus.ROOT;
10     locked = false;
11   }
12   public Node(Node myParent) {
13     parent = myParent;
14     cStatus = CStatus.IDLE;
15     locked = false;
16   }
17   ...
18 }
```

图12-1 Node类：构造函数和域

为了初始化p个线程的CombiningTree，需要创建一个宽度$w=2p$的由Node对象所组成的数组，根为node[0]，且对于任意$0<i<w$，node[i]的父结点为node[$(i-1)/2$]。叶结点为数组中最后的$(w+1)/2$个结点，其中线程i被分配给叶结点$i/2$，根的初始组合状态为ROOT，其他结点的组合状态为IDLE。图12-2描述了CombiningTree的构造函数。

```
1   public CombiningTree(int width) {
2     Node[] nodes = new Node[width - 1];
3     nodes[0] = new Node();
4     for (int i = 1; i < nodes.length; i++) {
5       nodes[i] = new Node(nodes[(i-1)/2]);
6     }
7     leaf = new Node[(width + 1)/2];
8     for (int i = 0; i < leaf.length; i++) {
9       leaf[i] = nodes[nodes.length - i - 1];
10    }
11  }
```

图12-2 CombiningTree类：构造函数

CombiningTree的getAndIncrement()方法如图12-4所示，它包括四个阶段。在预组合阶段（第16行至第19行），CombiningTree类的getAndIncrement()方法向上移动，并对所遇到的每个结点调用precombine()。precombine()方法返回一个布尔值以表明该线程是否为第一个到达该结点的线程。如果是，则getAndIncrement()方法继续向上移动。对最后一个访问的结点设置stop变量，该结点要么为该线程第一次到达的最终结点，要么为根结点。例如，

表12-3描述了一个预组合阶段的例子。线程A是最快的，在根结点上停止，而B则停在它比A晚到的中间层结点上，C停在它比B还晚到的叶结点上。

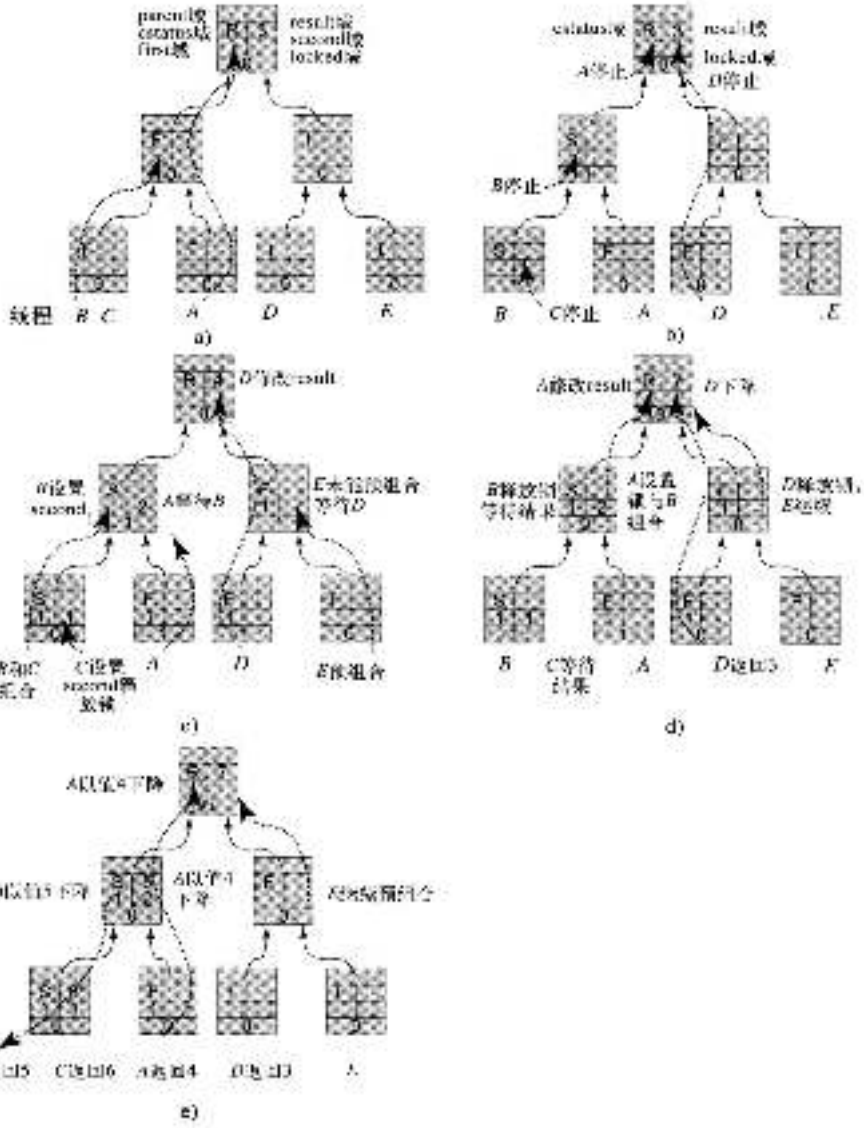


图12-3 由5个线程对一个宽度为8的组合树的并发遍历。初始时，该结构的所有结点都未上锁，根结点状态为CStatus ROOT，其他的结点为CStatus IDLE

图12-5描述了Node的precombine()方法。在第20行，线程等待直到同步状态为FREE。在第21行，线程测试组合状态。

IDLE

线程将结点的状态设置为FIRST，表示它将返回以查找一个用于组合的值。如果找到一个这样的值，它将作为主动线程继续推进。而提供这个值的线程则作为被动线程，该调用然后返回*true*，表示该线程继续向上。

```
12    public int getAndIncrement() {
13      Stack<Node> stack = new Stack<Node>();
14      Node myLeaf = leaf[ThreadID.get()/2];
15      Node node = myLeaf;
16      // precombining phase
17      while (node.precombine()) {
18        node = node.parent;
19      }
20      Node stop = node;
21      // combining phase
22      node = myLeaf;
23      int combined = 1;
24      while (node != stop) {
25        combined = node.combine(combined);
26        stack.push(node);
27        node = node.parent;
28      }
29      // operation phase
30      int prior = stop.op(combined);
31      // distribution phase
32      while (!stack.empty()) {
33        node = stack.pop();
34        node.distribute(prior);
35      }
36      return prior;
37    }
```

图12.4 CombiningTree类：getAndIncrement()方法

```
19  synchronized boolean precombine() {
20    while (locked) wait();
21    switch (cStatus) {
22      case IDLE:
23        cStatus = CStatus.FIRST;
24        return true;
25      case FIRST:
26        locked = true;
27        cStatus = CStatus.SECOND;
28        return false;
29      case ROOT:
30        return false;
31      default:
32        throw new PanicException("unexpected Node state" + cStatus);
33    }
34  }
```

图12-5 Node类：precombining()方法

FIRST

一个较早的线程最近已访问了该结点，并准备返回查找一个用于组合的值。该线程命令线程停止向上移动（通过返回*false*），然后开始下一阶段，计算要组合的值。在线程返回之前，它对该结点设置一个长期锁（通过设置locked为*true*），防止较早访问的线程在没有与该线程的值进行组合的情形下就向前推进。

ROOT

如果线程已到达根结点，它将指示线程开始下一个阶段。

第31行是一个默认情形，仅当出现一个非预期状态时才被执行。

编程提示12.3.1 编程时对于所有可能的枚举值，即使知道它不可能发生也要进行分析，这是一种很好的编程实践。如果程序编写错了，则调试起来非常方便；如果编写是正确的，那么程序也能方便地被其他人修改，即使这个人对此程序并不是特别了解。所以这样编写的程序总是具有很好的防御特点。

在组合阶段（图12-4中第21～28行），线程重新访问在预组合阶段访问过的结点，并将它自己的值与其他线程所留下的值相组合。当该线程到达预组合阶段结束所在的stop结点时，则停止。随后，以倒序方式来遍历这些结点，并将遍历时所遇的结点压入栈中。

图12-6中Node类的combine()方法将最近到达的一个被动进程所留下的值与至今已组合的值相加。和前面一样，线程首先等待，直到locked域变为*false*，然后在该结点上设置一个长期锁，以确保晚到达的线程不与该线程组合。如果状态为SECOND，则将其他线程的值与其累加值相加，否则，返回最先完成的值。在图12-3的c部分中，线程*A*在组合阶段开始沿着树向上移动，到达第二层被线程*B*锁住的结点，然后等待。在d部分中，线程*B*释放第二层结点上的锁，然后*A*看到该结点的组合状态为SECOND，它锁住该结点并以组合值3移到根结点，该组合值为*A*和*B*所写的FirstValue域和SecondValue域的总和。

```
35    synchronized int combine(int combined) {
36      while (locked) wait();
37      locked = true;
38      firstValue = combined;
39      switch (cStatus) {
40        case FIRST:
41          return firstValue;
42        case SECOND:
43          return firstValue + secondValue;
44        default:
45          throw new PanicException("Unexpected Node state " + cStatus);
46      }
47    }
```

图12-6 Node类：组合阶段。该方法将FirstValue和SecondValue相加，但是任何其他可交换方法的工作方式与此类似

在操作阶段的开始（第29行和第30行），线程已组合了所有低层结点的方法调用，现在检查它在预组合阶段结束时所停止的结点（图12-7）。如果这个结点为根结点（如图12-3的d部分所示），则该线程（此时为*A*）执行组合的getAndIncrement()操作：将它的累加值（比例中为3）加到result中并返回prior值，否则，该线程对结点开锁，通知所有被阻塞的线程，将自己的值作为SecondValue，等待另一个线程将组合操作传递到根结点后返回结果。例如，图12-3的c和d部分描述了线程*B*的一个动作序列。

当结果返回时，*A*进入分布阶段，沿着树向下传递结果。在这个阶段（第31～36行），*A*向下移动，释放锁，并通知关于这些值的被动伙伴它们应该向它们自己的被动伙伴或调用者（在最底层）报告。图12-8描述了distribute方法。如果结点的状态为FIRST，则不存在其他线程可以和正在分布的线程相组合，所以该线程能够通过释放锁并设置其状态为IDLE来将结点重新设置为初始状态。如果结点的状态为SECOND，正在分布的线程将结果更新为从上一层带来的prior值与FIRST值的总和。这反映了这样一种情形，即结点上的主动线程曾经试图在

262
↓
265

被动线程之前执行它自己的原子操作。一旦正在分布的线程将状态设置为RESULT，等待获得一个值的被动线程就读RESULT。例如，在图12-3的e部分中，主动线程A在中间层结点执行它的分布阶段，设置result为5，将状态改为RESULT，并且向下移动到叶结点，返回值4作为它的输出。被动线程B被唤醒，发现中间层结点的状态已改变了，则读结果值5。

```
48   synchronized int op(int combined) {
49     switch (cStatus) {
50       case ROOT:
51         int prior = result;
52         result += combined;
53         return prior;
54       case SECOND:
55         secondValue = combined;
56         locked = false;
57          notifyAll(); // wake up waiting threads
58         while (cStatus != CStatus.RESULT) wait();
59         locked = false;
60          notifyAll();
61         cStatus = CStatus.IDLE;
62         return result;
63       default:
64         throw new PanicException("unexpected Node state");
65     }
66   }
```

图12-7 Node类：访用操作

```
67   synchronized void distribute(int prior) {
68     switch (cStatus) {
69       case FIRST:
70         cStatus = CStatus.IDLE;
71         locked = false;
72         break;
73       case SECOND:
74         result = prior + firstValue;
75         cStatus = CStatus.RESULT;
76         break;
77       default:
78         throw new PanicException("unexpected Node state");
79     }
80     notifyAll();
81   }
```

图12-8 Node类：分布阶段

12.3.2 一个扩展实例

图12-3描述了执行过程的各个不同阶段。假设有五个线程，分别标记为A到E。每个结点有六个域，如图12-1所示。初始时，所有的结点没有被锁住，且除了根结点之外的所有结点都处于IDLE组合状态。a中计数器的初始状态值为3。这是较早时候的计算值。在a中，为了完成getAndIcrement()，线程A和B开始进入预组合阶段，线程A向上移动，将它所访问的结点状态从IDLE改变为FIRST，表示在向上组合值的过程中它将变成主动线程。线程B在它初叶结点上是主动线程，但是还没有到达与A共享的第二层结点。在b中，B到达第二层结点并停止，

将其从FIRST改变为SECOND，表示它将收集被组合的值并等待A带着组合值向根结点推进。B锁住结点（将locked域从*false*改为*true*），以防止在组合阶段A没有获得B的组合值就开始前进。然而，B还没有对这些值进行组合。在它完成组合之前，C开始预组合，到达叶结点，然后停下来，将其状态改变为SECOND。C同样也锁住结点，防止B没有输入就进入组合阶段。类似地，D开始预组合并成功地到达根结点。无论A还是D都没有改变根结点的状态。事实上它们也决不会改变。它们只是简单地将根结点标记为停止预组合的结点。在c中，A开始在组合阶段向上推进。它首先锁住叶结点，这样任何较晚到达的线程将无法在预组合阶段继续前进，必须等待A完成它的组合阶段和分布阶段。A到达第二层结点，由于该结点被B锁住，于是等待。同时，C开始组合，但由于它在叶结点停止，所以该结点执行op()方法，将SecondValue设置为1，然后释放锁。当B进入组合阶段时，叶结点已被并锁，并标识为SECOND，于是B将1写入FirstValue，并以组合值2（即FirstValue与SecondValue的和）向上进入第二层结点。

当B到达第二层结点，即它在预组合阶段终止的结点时，它对该结点调用op()方法，将SecondValue设置为2，A则必须等待它释放该锁。与此同时，在树的右手边，D执行它的组合阶段，当它向上时锁住遇到的结点，因为它没有遇见其他需要组合的线程，于是它在根结点的result域读到3并将其更新为4。然后，线程E开始预组合，但由于太晚而没有遇见D。在D锁住第二层结点的时候，E无法继续预组合。在d中，B释放第二层结点上的锁，A看到该结点状态为SECOND，于是就锁住它，并以组合值3移动到根结点，该值是分别由A和B所写的FirstValue域与SecondValue域值之和。当D完成对根结点的更新时，A被延迟。一旦D完成，A就从根结点的result域中读到4并将其更新为7。D向下访问树（通过从它的本地Stack出栈），释放锁并返回它原先从根结点的result域中读到的值3。现在，E在它的预组合阶段继续上移，最后，在e中，A执行它的分布阶段。它返回到中间层结点，将result设置为5，将状态改为RESULT，并向下直到叶结点，返回值4作为它的输出。B被唤醒并发现中间层结点的状态已改变，读值5作为result，并向下至叶结点，将叶结点的result域设置为6，状态设置为RESULT，然后，B返回5作为它的输出。最后，C被唤醒并发现叶结点的状态已改变了，读6作为result，并将该值作为它的输出值返回，线程A至D分别返回值3至6，它们与根结点result域的值7相匹配，被不同线程调用的getAndIncrement()方法的线性化次序由它们在预组合阶段中在树中的次序所决定。

268

12.3.3 性能和健壮性

和本章描述的其他算法一样，CombiningTree的吞吐量以一种复杂的方式依赖于应用程序和底层系统结构的特性，然而，采用定性的方式同顾文献中的相关实验结果仍然是很有价值的。对详细的实验结果（主要针对过时的系统结构）感兴趣的读者可查阅本章注释。

作为一个假想实验，在理想情况下，即每一个线程都能与其他线程组合它们的增量的情况下，CombiningTree应该能提供较高的吞吐量，但在最坏情况下，即有许多线程都较晚地到达一个被锁住的结点，从而失去组合的机会并被迫等待更早的请求向上及向下访问树的情形下，CombiningTree有可能提供很差的吞吐量。

在实际中，实验数据也支持这种非形式化的分析。争用越高，观察到的组合速率越快，观察到的加速比也就越大，较坏的情形变为较好的了。然而，在并行度很低时，组合树并不具优势。随着增量请求到达速率的降低，组合速率将迅速下降。吞吐量与请求的到达速率密

切相关。

由于组合可以提高吞吐量，而失败的组合并不提高吞吐量，所以让到达一个结点的请求等待一段合理的时间，直到另一个带有要组合增量的线程到达，将是一种很有意义的做法。毫无异议，当争用低时等待一小段时间，而争用高时则等待一段较长的时间。当争用足够高时，无限的等待将获得很好的效果。

若请求到达的次数波动很大时算法仍然具有很好的效果，则称该算法是健壮的。在文献中已表明具有固定等待时间的CombiningTree并不是健壮的，因为请求到达速率的高度变化有可能降低组合速率。

12.4 静态一致池和计数器

> 首先，汝要拔下神圣之顶针；然后，汝要数数直到三，不得多，不得少；第三，三应为汝数过的数，汝已数过的数应为三；……当数三作为第三个数被数到，那时，汝将此要提阿之神圣手榴弹扔向汝之仇敌，那在汝面前聚集的仇敌，就将灰飞烟灭。
>
> ——摘自《巨蟒与圣杯》

并非所有的应用都需要可线性化的计数。的确，基于计数器的Pool实现只需静态一致[⊖]的计数：所要做的事就是让计数器不出现重复和丢失。只需保证每一个元素都是由put()放入数组项，并由另一个线程调用get()访问该项并最终协调这些put()和get()调用。（环绕式处理仍可能引起多个put()和get()调用对同一个数组项的竞争。）

269

12.5 计数网

学探戈舞的人都知道舞伴之间必须密切地协调：如果动作不一致，无论他们每个人的舞技如何高超，但舞蹈却跳不好。同样，组合树也必须密切地协调：如果请求不是同时到达，则无论单个进程运行得多么快，算法却不能有效地工作。

本节讨论计数网，它看起来不像探戈，倒更像狂欢舞会：每个参与者以各自的步伐移动，但在整体上计数器却传递着一个静态一致的具有高吞吐量的索引集合。

考虑把组合树中的单个计数器替换成多个计数器，每个计数器都分布一个索引子集（见图12-9）。假设分配w个计数器（图中w=4），其中每个计数器都分发一个唯一的模w的索引集合（例如，图中第二个计数器对于递增的i分发2，6，10，…，i*w+2），难点在于如何在计数器之间分配线程而不会出现重复和丢失，以及如何以分布式和松散的方式来实现这种分配。



图12-9 由一个计数网随后接上w=4个计数器所组成的静态一致共享计数器，线程遍历整个计数网以选择访问哪个计数器

⊖ 关于静态一致性的详细定义参见第3章。

12.5.1 可计数网

平衡器是一种具有两条输入线和两条输出线的简单交换机，输入线和输出线分别称作顶线和底线（有时也叫北线和南线）。令牌可以随机地到达平衡器的输入线，并在随后的某个时刻出现在输出线上。可以将平衡器看作是一个触发电路：给定一个输入令牌流，它先将第一个令牌发送到顶端输出线，再将下一个发送到底端输出线，如此不断地执行，从而有效地平衡了两条线上的令牌数（见图12-10）。更确切地说，一个平衡器具有两个状态：上和下。如果状态为上，那么下一个令牌出现在顶线上，否则，出现在底线上。



图12-10 平衡器。令牌在任意时间到达任意输入线，然后被调整方向以确保当所有令牌离开平衡器时，出现在顶线的令牌最多比出现在底线的令牌多一个

220

用x_0和x_1分别代表到达平衡器顶端输入线和底端输入线的令牌个数，用y_0和y_1分别表示出现在顶端输出线和底端输出线的令牌数目。平衡器绝不创建令牌，即在任何时刻都有

$$x_0+x_1 \geq y_0+y_1$$

如果每一个到达输入线的令牌都已出现在输出线上，则称该平衡器是静态的，即

$$x_0+x_1=y_0+y_1$$

平衡网是通过将其平衡器的输出线连接到其他平衡器的输入线上所构成的。宽度为w的平衡网具有w个输入线x_0，x_1，…，x_{w-1}（没有连接到其他平衡器的输出线上）和w个输出线y_0，y_1，…，y_{w-1}（同样没有连接到其他平衡器的输入线上）。平衡网的深度是指从任意一个输入线开始所能遍历的最大平衡器个数。这里只考虑有限深度的平衡网（即线没有形成环）。像平衡器一样，平衡网绝不创建令牌，即

$$\sum x_i \geq \sum y_i$$

（当对一个序列中的每一个元素都求和时，往往省略总和中的索引。）如果每一个到达输入线的令牌都已出现在输出线上，则称该平衡网是静态的，即

$$\sum x_i=\sum y_i$$

至此，平衡网被描述为就像网络中的交换机一样，在一个共享存储器的多处理器中，平衡网可以作为存储器中的对象来执行。一个平衡器是一个对象，而平衡器的线则是从一个平衡器到另一个平衡器的引用。每个线程不断地遍历该对象，从一个输入线开始，而在一个输出线上出现，有效地引导着一个令牌穿越整个网络。

有些平衡网具有其些有趣的特性。图12-11所描述的网络具有4条输入线和4条输出线。初始时，所有平衡器的状态都为上。我们可以验证，任意个令牌以任意次序从任意输入线集上进入网络，它们都会按照一定的规则在输出线上出现。非形式化地说，无论在输入线上令牌的到达是如何分布的，它们在输出线上的分布都是平衡的。且首先在顶端输出线上输出。如果令牌个数n是4的倍数（网络宽度），则每条输出线上出现的令牌数是一样的。如果有1个多出的令牌，则会出现在0线上；如果多出2个，则出现在0线和1线上，以此类推。一般地，如果

221

$$n=\sum x_i$$

则

$$y_i=(x'w)+(i \bmod w)$$

这种性质称为步进特性。

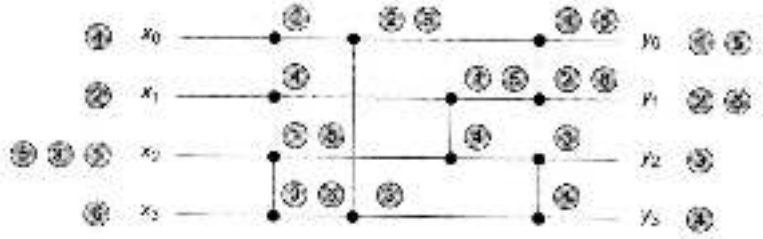


图12-11 Bitonic[4]计数网的一次顺序执行过程。每一条垂直线代表一个平衡器，水平线代表平衡器的两条输入/输出线，它们在圆点处相结合。在这个顺序执行过程中，各令牌按照令牌上的数字顺序一个接一个地穿过该网络。我们跟踪每一个穿过网络到达其输出线的令牌。例如，3号令牌从线2进入网络，中间下降到线1，最后在线3上结束。注意，每个平衡器是如何保持其步进特性的，整个网络又是如何保持其步进特性的

满足步进特性的平衡网称为计数网，因为它能够很容易地计算出穿越网络的令牌数量。如图12-9所示，它给每一条输出线i增加一个本地计数器，从而使得出现在那条线上的令牌被分派一个连续的号码i，$i+w$，…，$i+(y_i-1)w$。

步进特性有多种定义方式，可以互换地使用这些定义。

引理12.5.1 设y_0，…，y_{w-1}是一系列非负整数，则以下陈述是等价的：

1. 对任意$i<j$，有$0\le y_i-y_j\le 1$；

2. 要么对于所有的i，j有$y_i=y_j$，要么存在一个c使得对任意的$i<c$，$j\ge c$有$y_i-y_j=1$；

3. 令$n=\sum y_i$，则$y_i=\left\lceil\frac{n-i}{w}\right\rceil$。

272

12.5.2 双调计数网

本节讲述如何将图12-11中的计数网推广到宽度为2的幂次方的计数网，现给出引导性的构造。

在描述计数网的时候，通常不用关心令牌到达的时间，而只是关心网络处于静态时出现在输出线上的令牌个数满足步进特性。将宽度为w的输入或输出序列$x=x_0$，x_1，…，x_{w-1}定义为一个分为w个子集x_i的令牌集合，x_i是所有到达或离开线i的输入令牌。

宽度为$2k$的平衡网Merger[$2k$]按照下面的方式来定义。它有两个宽度为k的输入序列x和x'以及一个宽度为$2k$的输出序列。在任何静止状态下，如果x和x'都具有步进特性，则y也具有步进特性。Merger[$2k$]网可以按照归纳的方式来定义（见图12-12）。当$k=1$时，Merger[$2k$]网是一个平衡器。对于任意$k>1$，我们使用两个Merger[k]网的输入序列x和x'以及k个平衡器来构造Merger[$2k$]，利用一个Merger[k]网，将x的偶数序列x_0，x_2，…，x_{k-2}和x'的奇数子序列x'_1，x'_3，…，x'_{k-1}相归并（也就是说，x_0，x_2，…，x_{k-2}，x'_1，x'_3，…，x'_{k-1}作为Merger[k]网的输入），同时将x的奇数序列和x'的偶数序列归并作为第二个Merger[k]网的输入。称两个Merger[k]网的输出为z和z'。最后一步是通过将每一个线对z_i和z'_i送入一个输出为y_{2i}和y_{2i+1}的平衡器来组合z和z'。

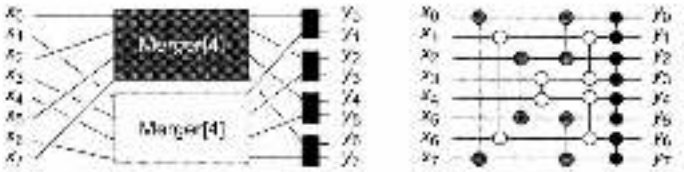


图12-12 左边为Merger[8]网的逻辑结构图，其输入由两个Bitonic[4]网的输出构成。灰色的Merger[4]网将上面一个Bitonic[4]网的奇数输出线和下面一个Bitonic[4]网的偶数输出线作为自己的输入线。另一个Merger[4]网络的情况则正好相反。当这些线离开这两个Merger[4]网络时，每一对输导相同的线将由一个平衡器组合。右图的右边我们可以看到一个Merger[8]网的实际布局图。不同的平衡器用不同的颜色标识以对应左图的逻辑结构

Merger[2k]网由log2k个层组成，每层有k个平衡器。仅当它的两个输入序列具有步进特性的时候，它的输出才具有步进特性，可以通过由较小的平衡网过滤输入来确保这一点。

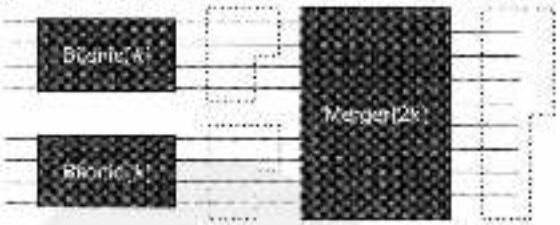


图12-13 一个Bitonic[2k]计数网的递归结构图，两个Bitonic[k]计数网的输出进入Merger[2k]平衡网

Bitonic[2k]网是通过将两个Bitonic[k]网的输出连接到Merger[2k]网的输入所构成的，这一点的最终在由单个平衡器所组成的Bitonic[2]网中接地，如图12-13所示，这种结构产生一个由（log,2k+1）层组成的网络，每一层有k个平衡器。

软件双调计数网

至此，平衡网被描述为硬件网络中的交换机一样，在一个共享存储器的多处理器中，平衡网可以作为存储器中的对象来实现。一个平衡器是一个对象，而平衡器的线则是从一个平衡器到另一个平衡器的引用。每个线程不断地遍历对象，从一条输入线开始，而在一条输出线上出现，有效地引导着一个令牌穿越整个网络。下面讲述如何将一个Bitonic[2]网作为一个共享存储的数据结构来执行。

```
1  public class Balancer {
2    boolean toggle = true;
3    public synchronized int traverse() {
4      try {
5        if (toggle) {
6          return 0;
7        } else {
8          return 1;
9        }
10     } finally {
11       toggle = !toggle;
12     }
13   }
14 }
```

图12-14 Balancer类：synchronized的实现

Balancer类（图12-14）有一个布尔域：toggle。同步的traverse()方法对toggle域求补并作为输出线返回0或1。Balancer类的traverse方法不需要参数，因为令牌离开平衡器的线并不依赖于它进入的那条线。

Merger类（图12-15）有3个域：width域值必须是2的幂，half[]是一个宽度Merger对象组成的二元数组（网络宽度为2时则为空），layer[]是一个由最终构成网络层的width/2个平衡器组成的数组。

```
1  public class Merger {
2    Merger[] half;  // two half-width merger networks
3    Balancer[] layer; // final layer
4    final int width;
5    public Merger(int myWidth) {
6      width = myWidth;
7      layer = new Balancer[width / 2];
8      for (int i = 0; i < width / 2; i++) {
9        layer[i] = new Balancer();
10     }
11     if (width > 2) {
12       half = new Merger[]{new Merger(width/2), new Merger(width/2)};
13     }
14   }
15   public int traverse(int input) {
16     int output = 0;
17     if (input < width / 2) {
18       output = half[input % 2].traverse(input / 2);
19     } else {
20       output = half[1 - (input % 2)].traverse(input / 2);
21     return (2 * output) + layer[output].traverse();
22   }
23 }
```

图12-15 Merger类

该类提供一个traverse()方法，其中i是令牌进入网络的线。（归并网与平衡器不同，一个令牌的路径依赖于其输入线。）如果令牌从较低的width/2线进入，则它将经过half[0]，否则经过half[1]。无论从哪个半宽度归并网遍历该网络，出现在线i上的平衡器都将连接到larger[i]上的第i个平衡器上。

Bitonic类（图12-16）也有3个域：width域为宽度（2的幂），half[]是一个半宽度Bitonic对象组成的二元数组，merger是全宽度的Merger网的宽度。如果网络宽度为2，则half[]未被初始化。否则，half[]的每个元素被初始化为一个半宽度的Bitonic网。

```
1  public class Bitonic {
2    Bitonic[] half; // two half-width bitonic networks
3    Merger merger; // final merger layer
4    final int width; // network width
5    public Bitonic(int myWidth) {
6      width = myWidth;
7      merger = new Merger(width);
8      if (width > 2) {
9        half = new Bitonic[]{new Bitonic(width/2), new Bitonic(width/2)};
10     }
11   }
12   public int traverse(int input) {
13     int output = 0;
14     if (width > 2) {
15       output = half[input / (width / 2)].traverse(input / 2);
16     }
17     return merger.traverse((input >= (size / 2) ? (size / 2) : 0) + output);
18   }
19 }
```

图12-16 Bitonic[]类

该类提供了traverse()方法。如果令牌从低width/2输入线进入，则它将穿过half[0]，否则穿过half[1]。若一个令牌在平归并子网的线i上出现，则会从输入线i遍历最后的归并网。

注意这个类采用了一种简单的同步Balancer实现方式，但如果这个Balancer实现是锁无关（或等待无关的），那么整个网络作为一个整体的实现也将是锁无关的（或等待无关的）。

正确性证明

下面证明Bitonic[w]是一个计数网。证明可以看作是令排序穿过该网络的论证过程。在检查网络本身之前，首先给出一些与具有步进特性的序列相关的简单引理。

引理12.5.2 如果一个序列具有步进特性，则它的所有子序列也有步进特性。

引理12.5.3 如果序列$x_0, \cdots, x_{k-1}$具有步进特性，则其奇/偶子序列满足：

$$\sum_{i=0}^{\lfloor k/2 \rfloor - 1} x_{2i} = \left\lceil \sum_{i=0}^{k-1} x_i / 2 \right\rceil, \quad \sum_{i=0}^{\lfloor k/2 \rfloor - 1} x_{2i+1} = \left\lfloor \sum_{i=0}^{k-1} x_i / 2 \right\rfloor$$

证明 要么对于$0 \le i < k/2$有$x_{2i} = x_{2i+1}$，要么根据引理12.5.1，对于所有的$i \ne j$及$0 \le i, j < k/2$，存在一个唯一的j使得$x_{2i} = x_{2j+1} + 1$且$x_{2i} = x_{2j+1}$。在第一种情形下$\sum x_{2i} = \sum x_{2i+1} = \sum x_i/2$。在第二种情形下$\sum x_{2i} = \lceil \sum x_i/2 \rceil$且$\sum x_{2i+1} = \lfloor \sum x_i/2 \rfloor$。 □

274 ~ 276

引理12.5.4 假设$x_0, \cdots, x_{k-1}$和$y_0, \cdots, y_{k-1}$是具有步进特性的任意序列，如果$\sum x_i = \sum y_i$，则对于任意$0 \le i < k$，有$x_i = y_i$。

证明 令$m = \sum x_i = \sum y_i$，根据引理12.5.1，$x_i = y_i = \left\lceil \frac{m-i}{k} \right\rceil$。 □

引理12.5.5 令$x_0, \cdots, x_{k-1}$和$y_0, \cdots, y_{k-1}$是具有步进特性的任意序列。如果$\sum x_i = \sum y_i + 1$，则存在唯一的j $(0 \le j < k)$，使得$x_j = y_j + 1$，并且对于$i \ne j$，$0 \le i < k$，有$x_i = y_i$。

证明 令$m = \sum x_i = \sum y_i + 1$。根据引理12.5.1，$x_i = \left\lceil \frac{m-i}{k} \right\rceil$且$y_i = \left\lceil \frac{m-1-i}{k} \right\rceil$。对任意的$i$ $(0 \le i < k)$，除了唯一的$i = m-1 \pmod k$以外，上述两项都成立。 □

现在来证明MERGER[w]网具有步进特性。

引理12.5.6 如果MERGER[2k]是输入的，其输入$x_0, \cdots, x_{k-1}$和$x'_0, \cdots, x'_{k-1}$都具有步进特性，则其输出$y_0, \cdots, y_{2k-1}$也具有步进特性。

证明 从$\log k$开始归纳。参考图12-17，该图描述了MERGER[8]网的一种证明结构。

如果$2k = 2$，则MERGER[2k]只是一个平衡器，根据平衡器的定义可知，其输出一定具有步进特性。

若$2k > 2$，则令$z_0, \cdots, z_{k-1}$为第一个MERGER[k]子网的输出，该子网是x的偶数子序列和x'的奇数子序列归并后的结果。令$z'_0, \cdots, z'_{k-1}$为第二个MERGER[k]子网的输出，按照假设，x和x'都具有步进特性，所以它们的奇/偶子序列同样具有步进特性（引理12.5.2），因此z和z'也具有步进特性（归纳假设）。此外，$\sum z_i = \lceil \sum x_i/2 \rceil + \lfloor \sum x'_i/2 \rfloor$并且$\sum z'_i = \lfloor \sum x_i/2 \rfloor + \lceil \sum x'_i/2 \rceil$（引理12.5.3）。简单地分析可知$\sum z_i$和$\sum z'_i$最多相差1。

可以断言，对于任意的$i < j$，$0 \le y_i - y_j \le 1$。如果$\sum z_i = \sum z'_i$，则由引理12.5.4可知，对于$0 \le i < k/2$，有$z_i = z'_i$。经过平衡器的最后一层后，

$$y_{2i} = y_{2i+1} = z_{\lfloor i/2 \rfloor} = z'_{\lfloor i/2 \rfloor}$$

这为z具有步进特性，显然该结论是成立的。

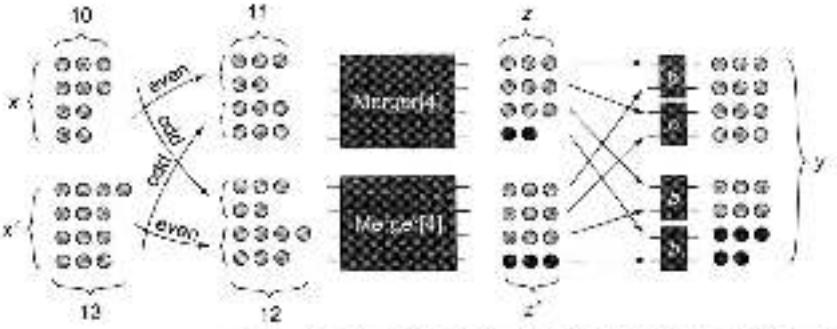


图12-17 一个MERGER [8]网能够正确地将两个宽度为4的具有步进特性的序列x和x'归并成一个宽度为8的具有步进特性的序列y的归纳证明过程。x和x'的宽度为2的奇数和偶数子序列均具有步进特性。而且，一个序列的偶数序列和另一个序列的奇数序列的令牌个数至多相差1（此例中分别有11和12个令牌）。由归纳假设可知，两个MERGER [4]网的输出z和z'也具有步进特性，且其中一个至多包含一个额外的令牌。这个额外的令牌必定在一条特殊编号的线上（本例中为线3）以得令牌引导至同一个平衡器。在本图中，这些令牌均被加黑标识，它们经由最南边的平衡器。而额外的令牌则经由北边的平衡器，从而保证最终的输出具有步进特性

类似地，如果Σz_i和$\Sigma z'_i$相差1，由引理12.5.5可知，对于任意的$0 \le i < k/2$，除了满足z_i与z'_i的差值为1的唯一值θ之外，都有$z_i=z'_i$。对于某个非负整数x，令$\max(z_\theta, z'_\theta)=x+1$，$\min(z_\theta, z'_\theta)=x$。从z和z'的步进特性可知，对于所有的$i<\theta$，$z_i=z'_i=x+1$，且对于所有的$i>\theta$，$z_i=z'_i=x$。因为$z_\theta$和$z'_\theta$被一个输出为$y_{2\theta}$和$y_{2\theta+1}$的平衡器连接，因此有$y_{2\theta}=x+1$和$y_{2\theta+1}=x$。类似地，对于$i\neq\theta$，$z_i$和$z'_i$被同一个平衡器连接。因此，对任意的$i<\theta$，$y_{2i}=y_{2i+1}=x+1$，且对于任意的$i>\theta$，$y_{2i}=y_{2i+1}=x$。所以通过选择$c=2\theta+1$及应用引理12.5.1，步进特性成立。 □

下面定理的证明过程是显然易见的。

定理12.5.1 在任何静态下，BITONIC[w]的输出具有步进特性。

周期计数网

本小节将说明Bitonic网并不是唯一的深度为$O(\log^2 w)$的计数网。下面介绍一种新的计数网，如图12-18所示，该网具有一个显著的特性，即它是周期性的，由一系列相同的子网组成。Block[k]网按照如下方式定义，当k为2时，Block[k]网由一个平衡器构成，对更大的k，Block[2k]网是按递归方式来构造的，这里我们从两个Block[k]网A和B开始。给定一个输入序列x，A的输入为x^0，B的输入为x^1。令y是两个子网的输出，其中y^0是A的输出序列，y^1是B的输出序列。构造网络的最后一步是将每一个y_i^0和y_i^1合并到单一的平衡器中，产生最终的输出z_{2i}和z_{2i+1}。

[277 ~ 278]

图12-19描述了一个Block[8]网的递归构造过程。Periodic[2k]网由$\log k$个Block[2k]网连接构成，使得一个子网的第i个输出线是下一个子网的第i个输入线。图12-18是Periodic[8]计数网。[⊖]

⊖ 尽管Block[2k]和Merger[2k]网看起来可能相同，但实际上不同：不允许来自一个网的线与另一个网的线进行置换。

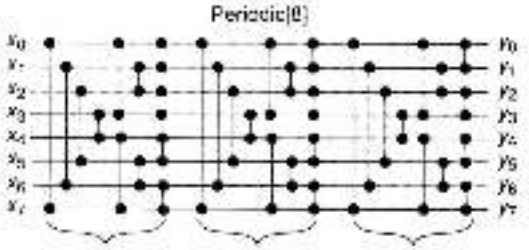


图12-18 由三个相同的Block[8]网构成一个Periodic[8]计数网

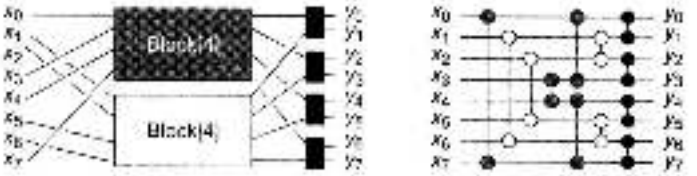


图12-19 左图描述了一个Block[8]网，该网的输入为两个Periodic[4]的输出。右图描述了Block[8]网的实际布局图。图中被着色的平衡器对应于左图的逻辑结构

周期的软件计数网

下面介绍如何采用软件方法构造Periodic网。在构造中再次使用了图12-14中的Balancer类。Block[w]网的每个层是通过Layer[w]网（图12-20）来实现的。Layer[w]网将输入线i和w－i－1连接到同一个平衡器上。

```
1  public class Layer {
2    int width;
3    Balancer[] layer;
4    public Layer(int width) {
5      this.width = width;
6      layer = new Balancer[width];
7      for (int i = 0; i < width / 2; i++) {
8        layer[i] = layer[width-i-1] = new Balancer();
9      }
10   }
11   public int traverse(int input) {
12     int toggle = layer[input].traverse();
13     int hi, lo;
14     if (input < width / 2) {
15       lo = input;
16       hi = width - input - 1;
17     } else {
18       lo = width - input - 1;
19       hi = input;
20     }
21     if (toggle == 0) {
22       return lo;
23     } else {
24       return hi;
25     }
26   }
27 }
```

图12-20 Layer网

在Block[w]类中（图12-21），当令牌自初始的Layer[w]网出现后，它将穿过两个半宽度Block[w/2]网（称为南网和北网）中的一个。

Periodic[w]网（图12-22）是一个由log w个Block[w]网组成的数组构成的。每个令牌依次遍历每个块，其中每个块的输出线作为其下一个块的输入线。（本章注释中引用了Periodic[w]网是一个计数网的证明。）

```
1  public class Block {
2    Block north;
3    Block south;
4    Layer layer;
5    int width;
6    public Block(int width) {
7      this.width = width;
8      if (width > 2) {
9        north = new Block(width / 2);
10       south = new Block(width / 2);
11     }
12     layer = new Layer(width);
13   }
14   public int traverse(int input) {
15     int wire = layer.traverse(input);
16     if (width > 2) {
17       if (wire < width / 2) {
18         return north.traverse(wire);
19       } else {
20         return (width / 2) + south.traverse(wire - (width / 2));
21       }
22     } else {
23       return wire;
24     }
25   }
26 }
```

图12-21 Block[w]网

```
1  public class Periodic {
2    Block[] block;
3    public Periodic(int width) {
4      int logSize = 0;
5      int myWidth = width;
6      while (myWidth > 1) {
7        logSize++;
8        myWidth = myWidth / 2;
9      }
10     block = new Block[logSize];
11     for (int i = 0; i < logSize; i++) {
12       block[i] = new Block(width);
13     }
14   }
15   public int traverse(int input) {
16     int wire = input;
17     for (Block b : block) {
18       wire = b.traverse(wire);
19     }
20     return wire;
21   }
22 }
```

图12-22 Periodic网

12.5.3 性能和流水线

计数网的吞吐量是怎样随着线程个数和网络宽度的变化而变化的呢？对于一个有固定宽度的网络，其吞吐量随着线程个数的增加而增加至某一个点，随后网络达到饱和，吞吐量将会保持不变或开始下降。为了更好地理解这些结论，我们可以将计数网看作是一条流水线。

- 如果并发遍历网络的令牌个数小于平衡器的个数，则流水线为部分空闲的，故吞吐量受到一定影响。
- 如果并发令牌的个数大于平衡器的个数，则流水线变为拥塞的，因为太多的令牌同时到达每个平衡器，从而导致对单个平衡器的争用明显。
- 当令牌数目与平衡器数目大致相等时，网络吞吐量达到最大。

如果一个应用需要计数网，那么最佳选择是能确保在任何时刻遍历平衡器的令牌个数大致上等于平衡器个数的网络。

279 - 281

12.6 衍射树

计数网提供了高度的流水线操作，所以吞吐量很大程度上与网络深度无关，但是，网络时延却依赖于网络深度。在我们已经了解过的计数网中，最浅的网络深度为$\Theta(\log^2 w)$。能设计一个深度为对数级的计数网吗？答案是可以，并且已经存在这样的网络，但不幸的是，对于所有已知的这样的网络构造，其中包含的常数因子都将导致这些构造难于实用。

下面是一种替换的方法。考虑一个具有单条输入线和两条输出线的平衡器集合，其中的顶线和底线分别标记为0和1。Tree[w]网（参见图12-23）是一个按如下方式构造的二叉树。令w是2的幂，采用归纳法定义Tree[2k]，当$k=1$时，Tree[2k]由一个输出线为y_0和y_1的平衡器组成。当$k>1$时，Tree[2k]由两个Tree[k]网和一个附加的平衡器构成。让单个平衡器的输入线x作为树的根，并将它的每一个输出线连接到一个宽度为k的树的输入线上。最新指定最终的Tree[2k]网的输出线，将子树Tree[k]的输出线y_0，y_1，…，y_k作为最终Tree[2k]网从“0”号输出线开始的偶数输出线y_0，y_2，…，y_{2k-2}，并将子树Tree[k]的输出线y_0，y_1，…，y_{2k-1}从平衡器的“1”号输出线开始作为最终Tree[2k]网的奇数输出线。

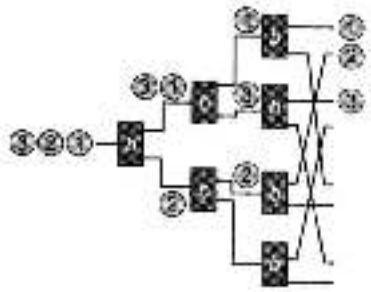


图12-23 Tree[8]类：计数树。注意观察网络是如何保持它的步进特性的

要理解为什么Tree[2k]网在静止状态下具有步进特性，我们归纳假设一个静止的Tree[2k]具有步进特性。根平衡器的子树Tree[k]的“0”号（顶）输出线传送的令牌数最多比向“1”号（底）输出线传送的令牌数多一个。在顶子树Tree[k]上存在的令牌具有一种与底子树上的令牌不同的步进特性，不同点在它们的k个输出线的某个线j上。Tree[2k]的输出线是离开两个子树的所有线的一种完美洗牌，宽度为k的两个阶型令牌序列形成宽度为$2k$的一个新的阶型序列，若有一个额外的令牌，它将出现在两个线j的高部，即有一个来自于树Tree[k]的顶。

282

Tree[w]网是一个计数网，但它是一个好的计数网吗？Tree[w]网的优点是它的深度较浅：Bitonic[w]网的深度为$\log^2 w$，而Tree[w]网的深度只有$\log w$。缺点是存在冲突：所有进入网络

的令牌都必须经过同一个根结点，从而导致该平衡器成为瓶颈。总的来说，平衡器在树中的级数越高，争用也就越严重。

可以利用类似于第11章EliminationBackOffStack的简单观察结果来降低争用：

> 如果偶数个令牌经过一个平衡器，输出将会在顶线和底线上均匀地平衡，但平衡器的状态保持不变。

衍射树的基本思想就是在每个平衡器上放置一个Prism，它是一种类似于EliminationArray的带外控制机制，允许令牌（线程）通过访问棱来交换元素。Prism允许令牌在随机的数组单元上配对并同意其分别沿着不同的方向衍射，也就是说，不用遍历平衡器的触发位或改变其状态就可以在不同的线上射出。仅当一个令牌在一个适当的时间周期内无法与另一个令牌配对时，这个令牌才会遍历平衡器的触发位，如果该令牌不准备衍射，则反复触发该位以决定沿着哪条路走。由此可知，如果该棱镜不引入太多的争用就能让足够多的令牌配成对，就可以避免平衡器的过度争用。

Prism是一个类似于EliminationArray的由Exchanger<Integer>对象组成的数组。一个Exchanger<T>对象允许两个线程交换T的值。如果线程*A*以参数*a*调用该对象的exchange()方法，线程*B*以参数*b*调用同一个对象的exchange()方法，则*A*调用的返回值是*b*，而*B*调用的返回值为*a*。第一个到达的线程被阻塞直到第二个线程到达，该调用包含一个超时参数，用来保证线程在一个适当时间段内不能与另一个线程交换值时仍能够继续推进。

Prism的实现如列12-24所示。在线程*A*访问平衡器的触发位之前，它首先访问与平衡器关联的Prism。*A*先随机地在Prism中选择一个数组项，然后调用该槽的exchange()方法，并将它自己的线程ID作为交换值。如果成功地与另一个线程交换了ID，则较低的线程ID在0号线上离开，较高的在1号线上离开。

```
1  public class Prism {
2    private static final int duration = 100;
3    Exchanger<Integer>[] exchanger;
4    Random random;
5    public Prism(int capacity) {
6      exchanger = (Exchanger<Integer>[]) new Exchanger[capacity];
7      for (int i = 0; i < capacity; i++) {
8        exchanger[i] = new Exchanger<Integer>();
9      }
10     random = new Random();
11   }
12   public boolean visit() throws TimeoutException, InterruptedException {
13     int me = ThreadID.get();
14     int slot = random.nextInt(exchanger.length);
15     int other = exchanger[slot].exchange(me, duration, TimeUnit.MILLISECONDS);
16     return (me < other);
17   }
18 }
```

图12-24 Prism类

图12-24描述了Prism的实现过程。构造函数将棱镜的容量（不同交换机的最大数）作为参数。Prism类提供了单一方法visit()，该方法随机地选择交换机。如果调用者从顶线离去，则visit()调用返回*true*，若从底线离去则返回*false*，如果没有交换值在定时已到，则抛出一个TimeoutException。调用者获得其线程ID（第13行），在数组中随机地选择一个数组项

(第14行)，并尝试将它自己的线程ID与其伙伴的ID相交换（第15行）。如果成功，则返回一个布尔值；如果超时，则重新抛出TimeoutException。

```
1  public class DiffractingBalancer {
2    Prism prism;
3    Balancer toggle;
4    public DiffractingBalancer(int capacity) {
5      prism = new Prism(capacity);
6      toggle = new Balancer();
7    }
8    public int traverse() {
9      boolean direction = false;
10     try{
11       if (prism.visit()){
12         return 0;
13       else
14         return 1;
15     } catch(TimeoutException ex) {
16       return toggle.traverse();
17     }
18   }
19 }
```

图12-25 DiffractingBalancer类：如果调用者通过棱镜与一个并发的调用者配对，则不必经过平衡器

DiffractingBalancer（图12-25）和常规平衡器一样，提供一个traversed()方法，该方法返回值0或者1。该类只有两个域：prism是一个Prism对象而toggle是一个Balancer对象。当一个线程调用traverse()时，它通过prism来尝试寻找一个伙伴，如果成功，伙伴们返回不同的值，这并不引起toggle上的争用（第11行）。否则，如果线程无法找到伙伴，则遍历（第16行）toggle（作为一个平衡器来实现）。

283
|
284

```
1  public class DiffractingTree {
2    DiffractingBalancer root;
3    DiffractingTree[] child;
4    int size;
5    public DiffractingTree(int mySize) {
6      size = mySize;
7      root = new DiffractingBalancer(size);
8      if (size > 2) {
9        child = new DiffractingTree[2
10         new DiffractingTree(size/2),
11         new DiffractingTree(size/2)];
12     }
13   }
14   public int traverse() {
15     int half = root.traverse();
16     if (size > 2) {
17       return (2 * (child[half].traverse()) + half);
18     } else {
19       return half;
20     }
21   }
22 }
```

图12-26 DiffractingTree类：域、构造函数以及traverse()方法

DiffractingTree类（图12.26）有两个域，child域是一个由子树组成的二元数组，root域是一个DiffractingBalancer类型的成员变量，在继续调用左子树或右子树时交替变换。每个DiffractingBalancer类型的变量都有一个容量，它实际上是其内部棱镜的容量。初始时，该容量为树的大小，在每一层容量递减一半。

与EliminationBackOffStack一样，DiffractingTree类的性能取决于两个参数：棱镜的容量和时限大小。如果棱镜容量太大，则线程之间彼此错过，这将导致在平衡器上的过度争用。如果数组太小，则会有过多的线程并发地访问一个棱镜中的每个交换机，从而导致在交换机上的过度争用。如果棱镜超时设置过短，线程则会彼此错过，如果设置过长，线程有可能会被不必要地延迟。对于这些值的选取没有硬性规定，因为最佳取值往往依赖于底层多处理器系统结构的负载和特征。

然而，实验结果表明，有些时候对这些值的选取能使得其性能优于CombiningTree和CountingNetwork类。下面是一些从实践中得到的较好的经验。由于树中较高层的平衡器上
285 会有较大的争用，所以在靠近树的顶部采用较大的棱镜，从而增加动态增减随机选择范围的能力。最佳的超时时限设置依赖于负荷：如果只有少许的线程在访问树，则花在等待上的时间大部分被浪费了；如果有大量线程在访问树，那么花在等待上的时间是值得的。自适应的模式应该具有较好的前景：当线程成功配对时，延长超时时限设定，否则，缩短超时时限设定。

12.7 并行排序

排序是最重要的计算任务之一，从19世纪Hollerith发明的排序机器到20世纪40年代的第一代电子计算机，直至今天的计算机，其中的大多数程序都使用了某种形式的排序。大多数计算机科学专业的大学生在学习初期就知道，排序算法的选择主要依赖于被排序的元素的个数、它们的关键字的数字特性以及这些元素存放在内存还是外存。并行排序算法可以按同样的方法来分类。

下面介绍两种排序算法：排序网，通常适用于内存中较小的数据集合；样本排序算法，通常适用于外存上的大量数据集。在下面的讲述中，为简单起见降低了算法的性能，更加复杂的技术请参阅本章注释中的引用。

12.8 排序网

正如计数网是由平衡器组成的网络一样，排序网是由比较器组成的网络。[8]比较器是一个具有两条输入线和两条输出线（称为顶线和底线）的计算单元。它从输入线上接收两个数字，并将较大的数字从顶线输出，较小的从底线输出。与平衡器不同的是，比较器是同步的：只有两个输入值都
286 到达才会产生输出（见图12-27）。



图12-27 比较器

与平衡网一样，比较网是一种由比较器组成的无环网络，输入值被放在它的w个输入线中的每一个线上。这些值同步地穿过比较器的每一层，最后一起从网络的输出线上离开。

[8] 历史上排序网要比计数网早出现几十年。

对于一个输入值为x_i和输出值为y_i的比较网，其中$i\in\{0\cdots1\}$且每个值分别在线i上。如果它的输出值为输入值的降序排序，即$y_{i-1}\geq y_i$，则该网络是一个有效的排序网。

下面的经典定理能够简化对任意一个网络进行排序的证明过程。

定理12.8.1（0-1原则） 如果一个排序网能对所有的0、1序列进行排序，则它能对任意的输入值序列进行排序。

排序网的设计

因为可以回收利用计数网的布局方案，所以没有必要去设计排序网。对于平衡网和比较网，如果通过互换平衡器和比较器，能够从一个网构造出另一个网，则称这两个网是同构的，反之亦然。

定理12.8.2 如果一个平衡网能够计数，则与之同构的比较网可以排序。

证明 首先构造一个从比较网转换到与之同构的平衡网的映射。

按照定理12.8.1，能够对所有0、1序列进行排序的比较网是一个排序网。取任意一个0、1序列作为比较网的输入，而在平衡网的每个输入线1上放置一个令牌，在输入线0上不放置令牌。如果采用锁步的方式运行两个网，则平衡网完全模拟了比较网。

采用对网络深度进行归纳的方法来证明。对于0层，按照构造该结论显然成立。假设对于第k层的线结论成立，下面证明它对第$k+1$层也成立。在比较网中当任一比较器上有两个1值相遇时，在平衡网中的某个平衡器上则会有两个令牌相遇，所以在比较网的每条线上离开的一个1值必在$k+1$层上离开，并且在平衡网的每条线上离开的令牌必在$k+1$层上离开。在比较网中当一个比较器上有两个0值相遇时，在平衡网中的某个平衡器上则没有令牌相遇，所以在比较网中$k+1$层上的每条线上0值离开，在平衡网中没有令牌离开。对于比较网中所有的0和1相遇的比较器，在$k+1$层上，1从北线（上部）上离开而0从南线（下部）上离开，同时在平衡网中令牌从北线上离开，南线上无令牌离开。

如果该平衡网是一个计数网，即在它的输出云线上具有步进特性，那么比较网必定已对0、1输入序列进行了排序。 □ 287

反之却不一定成立：并非所有的排序网都是计数网。如图12-28所示，该网是一个排序网但不是计数网，具体证明过程留作课后习题。

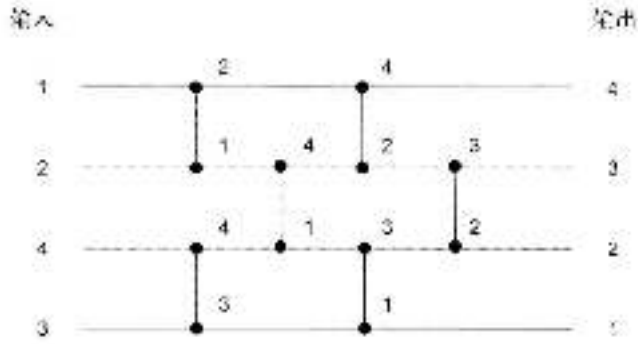


图12-28 OddEven排序网

推论12.8.1 与Bitonic[]和Periodic[]同构的比较网是排序网。

用比较网对一个大小为w的集合进行排序需要$\Omega(w\log w)$次比较。具有w个输入线的排序

网在每一层上最多有$O(w)$个比较器，所以其深度最小为$\Omega(\log w)$。

推论12.8.2 计数网的最小深度为$\Omega(\log w)$。

双调排序算法

任意宽度为w的排序网，例如Bitonic[w]，都可以看作是一个具有d层且每层有$w/2$个平衡器所组成的集合。排序网的设计布局可以表示为一张表，其中的每个表项是一个元组列，它描述了在其个层的某个平衡器上是哪两条线相遇。（例如，在图12-11所示的Bitonic[4]中，在第一层的第一个平衡器上线0和线1相遇，在第二层的第一个平衡器上线0和线3相遇。）为简单起见，假设给定了一个无界表bitonicTable[i][d][j]，其中每个表项包含了与d层的平衡器i相关的北（0）线和南（1）线的索引。

基于数组的内置排序算法以要被排序的元素所组成的数组作为输入（假定这些元素都具有唯一的整型关键字），并返回同一个按关键字排好序的元素所组成的数组。下面介绍
288 BitonicSort的实现，这是一种基于双调排序网的内置数组排序算法。假设要对一个$2*p*s$个元素组成的数组进行排序，其中p是线程个数（通常也是线程运行的最大可用处理器个数），$p*s$是2的幂。该网络的每一层都有$p*s$个比较器。

p个线程中的每一个都模仿s个比较器的工作。与计数网不同，这种网就像不合拍的舞步，而排序网则是同步的：一个比较器的所有输入必须在开始计算输出之前到达。该算法是以轮转方式推进的。在每一轮中，线程在网络的某一层中执行s个比较操作，必要时交换元素的数组项，从而使得它们排序正确。在网络的每一层，比较器将连接不同的线，所以不会有两个线程试图交换同一个数组项的元素，从而避免了在任意层进行同步操作的必要。

为了保证在某一轮（层）中的比较能在其下一轮开始之前完成，我们采用一种称为Barrier（将在第17章中详细研究）的同步构造。一个对于p个线程的路障提供了await()方法，该方法的调用直到全部p个线程都调用了await()方法时才返回。图12-29描述了BitonicSort

```
1  public class BitonicSort {
2    static final int[][][] bitonicTable = ...;
3    static final int width = ...; // counting network width
4    static final int depth = ...; // counting network depth
5    static final int p = ...;  // number of threads
6    static final int s = ...;  // a power of 2
7    Barrier barrier;
8    ...
9      public <T> void sort(Item<T>[] items) {
10       int i = ThreadID.get();
11       for (int d = 0; d < depth; d++) {
12         barrier.await();
13         for (int j = 0; j < s; j++) {
14           int north = bitonicTable[(i*s)+j][d][0];
15           int south = bitonicTable[(i*s)+j][d][1];
16           if (items[north].key < items[south].key) {
17             Item<T> temp = items[north];
18             items[north] = items[south];
19             items[south] = temp;
20           }
21         }
22       }
23     }
```

图12-29 BitonicSort类

的实现元素。每个线程一轮接一轮地通过网络的每一层。在每一轮中，它等待其他线程到达（第12行），以确保items数组中包含上一轮的结果。然后，该线程比较对应于比较器线的数组 289
元素，如果它们的key次序颠倒，则将它们互换（第14～19行），从而模仿那个层中s个平衡器的行为。

对于在p个处理器上运行的p个线程来说，BitonicSort需要$O(s\log^2 p)$的时间，如果s是常数，则时间复杂度为$O(\log^2 p)$。

12.9 样本排序

BitonicSort排序适用于内存中较小数据集的排序。对于较大的数据集（元素个数n比线程数p大得多），特别是存放在外存储设备上的数据，则需要采用不同的方式进行排序。因为访问数据的开销很大，必须尽可能地缩短本地引用，所以让单个线程顺序地对元素进行排序是比较划算的。而类似于BitonicSort的并行排序，它允许一个元素由多个线程访问，这样做显然开销太大。

下面尝试通过随机化法使访问一个给定元素的线程个数最小化。这种随机的使用不同于DiffractingTree，它是采用随机方式来分布内存访问。下面通过使用随机方式来猜测在要排序的数据集合中元素的分布情况。

由于要排序的数据集很大，可以将它分为多个桶，将key值在同一个范围的元素放入一个桶中。然后，每个线程使用顺序排序算法对每个桶中的元素进行排序，结果则是一个排好序的集合（按照适当的桶序来看）。该算法是著名的快速排序算法的一般化表达，但不是采用一个分裂器键将元素划分为两个子集，而是有$p-1$个分裂器键将输入集划分为p个子集。

这种对于n个元素和p个线程的算法包括三个阶段：

1. 线程选择$p-1$个分裂器键将数据集划分为p个桶，这些分裂器键是公开的，所有的线程都可以读取它们。

2. 每个线程顺序地处理n/p元素，将每个元素放入它自己的桶中，其中相应的桶是通过对分裂器键之间的元素键值进行二分查找来得到的。

3. 每个线程对其桶中的元素进行顺序排序。

阶段之间的路障能够确保所有的线程在开始下一个阶段之前都已完成了本阶段的工作。

在讨论第一个阶段之前，首先来讨论第二个阶段和第三个阶段。

第二个阶段的时间复杂度为$(n/p)\log p$，它包括从内存、磁盘或磁带上读取每个元素的时间、对本地缓存中的p个分裂器的二分查找时间以及将元素放入指定的桶的时间。元素所放入的桶可以是内存、磁盘或磁带，所以主要的开销是对存储数据元素访问n/p次的时间。 290

令b是桶中元素的个数。一个给定线程在第三阶段的时间复杂度是$O(b\log b)$，它是使用顺序算法（快速排序）[⊖]对元素进行排序的时间。这部分的开销最大，因为它是由访问慢速设备（磁盘或磁带）的读/写阶段所组成的。

算法的时间复杂度取决于在第三阶段桶中元素数目最多的线程。因此，尽可能选择分裂器均匀地分布元素显得特别重要，即在第二阶段中，每个桶大约应存放$n-p$个元素。

选择好的分裂器的关键就是给每个线程选取一个样本分裂器集合，用来表示它自己的$n-p$

⊖ 如果元素键值大小已知且固定，可以使用基数排序之类的算法。

大小的数据集，然后从所有线程的所有样本分裂器集中选择最终的$p-1$个分裂器。每个线程随机均匀地从它的大小为n/p的数据集中挑选s个key。（实际中，s通常为32或64。）然后，每个线程都加入到并行BitonicSort（图12-29）的运行中，对p个线程所选取的sp个样本key值进行处理。最后，每个线程在已排序的分裂器集合中的s、$2s$、…、$(p-1)s$处读取$p-1$个分裂器的key值，并将它们作为第二阶段中的分裂器。这种s个样本的选择以及随后的从已排序的所有样本集合中选择最后的分裂器的方法，将会降低在线程访问的n/p大小的数据集上key分布不均匀所带来的影响。

例如，一个样本排序算法可以选择让每个线程在它自己的n/p大小的数据集中选取第二阶段的$p-1$个分裂器，而不用与其他线程进行通信。这种方法存在的问题是，如果数据的分布不均匀，桶的大小有可能差异很大，性能将会受到影响。例如，如果在最大的桶中元素的个数为一般情形的两倍，那么排序算法的最坏时间复杂度也将加倍。

第一个阶段的时间复杂度为进行随机取样时间s（常数），而并行双调排序时间为$O(\log^2 p)$。对于具有好的分裂器集（每个桶中有n/p个元素）的样本排序算法，其总的时间复杂度为

$$O(\log^2 p)+O((n/p)\log p)+O((n/p)\log(n/p))$$

总的表达是$O((n/p)\log(n/p))$。

12.10 分布式协作

本章讲述了几种分布式协作模式，其中的一些（例如，组合树、排序网以及样本排序）具有高并行性和低开销。所有这些算法都包含同步点，即线程计算过程中必须等待与其他线程会合的点。在组合树中，线程必须同步地进行组合，而在排序中，线程则必须在路障点等待。

291

在其他模式中，例如计数网和衍射树，线程无需相互等待。（虽然使用了synchronized方法来实现平衡器效果，但也可以通过compareAndSet()方法以无锁的方式来实现。）这些分布式结构能在线程之间传递信息，虽然可以证明会合具有一些优点（如在Prism数组中），但它并不是必需的。

随机化在许多场合都是非常有用的，它能使工作分布均匀化。在衍射树中，对多个内存单元采用随机化来分布工作，从而减少了过多的线程同时访问同一单元的概率。在样本排序中，采用随机化方式能够将工作均匀地分布在多个桶中，以便线程随后并行地排序。

最后，流水线能够确保某些数据结构即使具有较长时延，却仍能具有较高的存生量。

虽然我们着重于共享存储器的多处理器，但值得一提的是，本章中介绍的分布式算法和数据结构同样适用于消息传递的系统结构。消息传递模型可以直接通过硬件来实现（例如多处理器网络），也可以在共享存储器系统结构上通过一个软件层（如MPI）来实现。

在共享存储器系统结构中，交换机（如组合树结点或平衡器）都是作为共享存储器的计数器来实现的。但在消息传递的系统结构中，交换机是作为本地处理器的数据结构来实现的，其中处理器之间的连线同样也是交换机之间的连接线。当一个处理器接收到一条信息时，它自动更新它的本地数据结构并将消息传送给管理其他交换机的处理器。

12.11 本章注释

组合树的思想归功于Allan Gottlieb、Ralph Grishman、Clyde Kruskal、Kevin McAuliffe、

Larry Rudolph和Marc Snir[47]。本章描述的软件CombiningTree则来自PenChung Yew、Nian-Feng Tzeng和Duncan Lawrie[151]，并由Beng-Hong Lim等人[65]对其进行了修改。所有这些算法都是基于James Goodman、Mary Vernon和Philip Woest[45]最早提出的设计思想。

计数网是由Jim Aspnes、Maurice Herlihy和Nir Shavit[16]所发明的。计数网与排序网相关，包括由Kenneth Batcher[18]发定的双调网，以及由Martin Dowd、Yehoshua Perl、Larry Rudolph和Mike Saks[35]提出的周期网。Miklós Ajtai、János Komlós和Endre Szemerédi发明了AKS排序网，这是一种深度为$O(\log w)$的排序网[8]。（这种渐近描述隐藏了大量常数，使基于AKS的网变为不实用的。）

292

Mike Klugerman和Greg Plaxton[84,85]最早提出了一种基于AKS的深度为$O(\log w)$的计数网构造。排序网的0-1原则是由Donald Knuth[86]提出的。一组类似的关于平衡网的规则是由Costas Busch和Marios Mavronicolas[25]提出的。衍射树是由Nir Shavit和Asaph Zemach[143]所提出的。

样本排序是由John Reif、Leslie Valiant[132]以及Huang和Chow[73]提出的。与顺序所有样本排序算法相关的顺序快速排序算法是由Tony Hoare[70]提出的。在文献中有许多并行基数排序算法，例如，由Daniel Jiménez-González、Joseph Larriba-Pey和Juan Navarro[82]提出的算法，以及由Shin-Jae Lee、Minsoo Jeon、Dongseung Kim和Andrew Sohn[101]提出的算法。

《巨蟒与圣杯》是由Graham Chapman、John Cleese、Terry Gilliam、Eric Idle、Terry Jones和Michael Palin所撰写的，并由Terry Gilliam和Terry Jones[27]共同导演。

12.12 习题

习题134. 证明引理12.5.1。

习题135. 实现一个三元CombiningTree，也就是说，最多允许来自三个子树的三个线程在一个给定的结点进行组合。与二元组合树相比，你认为它有什么样的优点和缺点？

习题136. 实现一个CombiningTree，通过使用Exchanger对象来完成正在沿树上升或下降的线程间的协作。这种构造与12.3节中的CombiningTree相比，有什么样的缺点？

习题137. 基于12.2节描述的共享池，对每个数组项使用两个简单计数器及一个ReentrantLock来实现一个循环数组。

习题138. 给出一种有效的Balancer的无锁实现。

习题139.（难题）给出一种有效的Balancer的无等待实现（不使用通用构造）。

习题140. 证明12.6节中的Tree[2k]平衡网是计数网，也就是说，在任何静止状态下，其输出线上的令牌序列具有步进特性。

293

习题141. 令N是一个处于静止状态s下深度为d、宽度为w的平衡网。令$n=2^d$。证明如果n个令牌从同一个输入线上进入网络、穿过网络并离开网络，则N在令牌离开网络以后的状态与令牌进入网络之前的状态相同。

在下面的习题中，k-光滑序列是一个满足下列要求的序列$y_0, \cdots, y_{k-1}$

$$\text{如果} i<j, \text{则} |y_i - y_j| \le k$$

习题142. 令X和Y是长度为w的k-光滑序列。平衡器关于X和Y的一个匹配是这样的一个集，即X中的每个元素通过一个平衡器与Y的每个元素按照一对一的方式相连接。

证明：如果X和Y都是k-光滑的，且X和Y匹配后的结果为Z，则Z是$(k-1)$-光滑的。

习题143. 考虑一个Block[k]网，其中每个平衡器被初始化为任意状态（上或下）。证明无论输入如何分布，输出分布总是(logk)-光滑的。

提示：可以利用习题142的结论。

习题144. 考虑网是一种平衡网，能够保证在任何静止状态下输出序列是1-光滑的。

计数网是光滑网，但反之不一定成立。

布尔排序网是一种所有输入都为布尔值的排序网。我们今天平衡网定义为一种布局与布尔排序网同构的平衡网。

令N是一个由宽度为w的光滑网S和宽度为w的伪排序平衡网P组成的平衡网，其中S的第i个输出线连接到P的第i个输入线上。

证明N是一个计数网。

习题145. 3-平衡器是一种具有三条输入线和三条输出线的平衡器。同2-平衡器一样，在任何静止状态下其输出序列都具有步进特性。使用3-平衡器和2-平衡器构建一个具有6条输入和输出线且深度为3的计数网，并说明它能正常工作的理由。

习题146. 修改BitonicSort类使得它能对宽度为w的输入数组运行排序，其中w不是2的幂。

习题147. 考虑下面的w-线程计数算法，每个线程首先使用一个宽度为w的双调计数网来获得一个计数器值v，然后穿过一个等待滤波器，在该滤波器中每个线程等待其他具有较小值的线程赶上。

294

等待滤波器是一个大小为w的布尔数组filter[]。定义相函数

$$\Phi(v)=\lfloor(v/w)\rfloor \bmod 2$$

一个以值v离开的线程在filter[$(v-1)$ mod w]上自旋，直到该位被置为$\Phi(v-1)$。该线程通过将filter[v mod w]设置为$\Phi(v)$来作出响应，然后返回值v。

1. 解释为什么这种计数器实现是可线性化的。
2. 在习题中已证明任意可线性化的计数网其深度至少为w，解释为什么filter[]的构造没有违背这个规则。
3. 在基于总线的多处理器系统中，这种filter[]构造是否比由自旋锁保护的单独变量具有更高的吞吐量？说明其理由。

习题148. 如果序列$X=x_0, \cdots, x_{w-1}$是k-光滑的，那么X穿过平衡网后的结果仍然是一个k-光滑序列。

习题149. 证明Bitonic[w]网的深度为(logw)(1+logw)/2且需要(wlogw)(1+logw)/4个平衡器。

习题150.（难题）给出一种无锁的DiffractingBalancer实现。

习题151. 给DiffractingBalancer的Prism增加一种自适应的定时机制。

习题152. 证明图12-28所示的OddEven网是排序网但不是计数网。

习题153. 计数网除了能自增外还能做其他事吗？考虑一种称为反令牌的新令牌，可以用它来自减。当令牌访问平衡器时，它执行一次getAndComplement()：自动地读取触发器值并对其取反，然后从原先触发器值所指定的输出线离开。然而，反令牌先对触发器值取反，然后从新的触发器值所指定的输出线离开。通俗地说，反令牌“消除”了最近的令牌对平衡器的触发状态的影响，反之亦然。

我们不再简单地平衡每条线上出现的令牌个数，而是给每个令牌赋一个权值+1，给每个反令牌赋一个权值−1。扩展步进特性使得在所有线上出现的令牌和反令牌权值的和也具有步进特性。我们称这种特性为加权的步进特性。

295

```
1   public synchronized int antiTraverse() {
2     try {
3       if (toggle) {
4         return 1;
5       } else {
6         return 0;
7       }
8     } finally {
9       toggle = !toggle;
10    }
11  }
```

图12-30 antiTraverse()方法

图12-30描述了antiTraverse()方法的实现，该方法能够使一个反令牌穿越平衡器。在其他的网中增加antiTraverse()方法的实现则留作习题。

令B是一个处于静止状态s下的深度为d、宽度为w的平衡网。令$n=2^d$。证明：如果n个令牌从同一条输入线上进入网络，穿过网络，直到最后退出网络，则B在令牌退出以后的状态与令牌进入网络之前的状态相同。

习题154. 令B是一个处于静止状态s的平衡网，假设一个令牌在线i上进入并穿过该网，使网络的状态为s'。证明：如果现在让一个反令牌从线i上进入并穿过这个网，则该网将返回到状态s。

习题155. 证明：如果平衡网B对于令牌来说是一个计数网，那么对于令牌和反令牌来说它也是一个平衡网。

习题156. 交换网是一个有向图，其中边代表线，结点代表交换机。每个线程引导一个令牌穿过网络。允许交换机和令牌具有内部状态。令牌通过一条输入线到达交换机。在一个原子步中，交换机吸收令牌，改变它自己的状态（也可能改变令牌的状态），然后从输出线上将令牌发射出去。为简单起见，假设交换机具有两条输入线和两条输出线。注意交换网的功能比平衡网更加强大，因为不仅令牌具有各种状态，交换机也可以有任意状态（而不是只有一个位）。

加法网络是一种允许线程增加1（或减少）任意值的交换网。

如果一个令牌在一个交换机的任意一条输入线上，则称这个令牌在该交换机处。从处于静止状态q_0的网开始，下一个要运行的令牌取值为0。假设有一个权为a的令牌t_0和$n-1$个权为b的令牌t_1，…，t_{n-1}，其中$b>a$，且每个令牌在不同的输入线上。用S表示t_0从初始状态q_0开始遍历该网络时遇到的所有交换机的集合。

证明：如果让t_1，…，t_{n-1}一次一个地穿过这个网，则每个t_i都能够在S的一个交换机前面中止。

在这个构造结束时，$n-1$个令牌在S中的交换机前。由于交换机具有两条输入线，那么t_0穿过该网的路径上至少包含$n-1$个交换机。所以任意加法网络的深度至少为$n-1$，其中n是最大的并发令牌数。这种限制并不令人满意，因为它意味着网络的规模依赖于线程个数（该结论也适用于CombiningTree，但不适用于计数网），即这种网本身是高时延的。

习题157. 扩展习题156的证明方法，论证可线性化的计数网的深度至少为n。

并发哈希和固有并行

13.1 引言

在前面的章节中，阐述了如何从队列、栈、计数器这些看似无法并行的数据结构中获取并行性。本章将采用一种与之截然不同的方法，研究并发哈希技术。该问题看起来像是“自然可并行的”，或更专业地称之为不相交的并发访问。也就是说，并发的方法调用可能访问不相交的存储单元，从而不再需要同步。

哈希是一种在顺序Set的实现中经常使用的技术，用于确保contains()，add()和remove()调用的平均时间为常量。第9章的并发Set实现要求时间与集合的大小呈线性关系。本章将研究各种能使哈希并行的方法，有时采用锁有时不用锁。尽管哈希看上去是自然并行的，但设计一种有效的并发哈希算法却并非易事。

和前面一样，Set接口提供以下具有布尔返回值的方法：

- add(x)向集合中添加x，如果x原先不在集合中，则返回*true*，否则返回*false*，
- remove(x)从集合中删除x，如果x原先在集合中，则返回*true*，否则返回*false*，
- 如果x在集合中，则方法contains(x)返回*true*，否则返回*false*，

在集合的实现中，应遵循以下原则：可以分配更多的内存，但不能占用更多的时间。如果要在一个运行得更快但耗费更多内存的算法和一个运行较慢但消耗较少内存的算法之间做出选择的话，应该更倾向于前者（这是情理之中的事情）。

298
↓
299

哈希集（有时称为哈希表）是一种实现集合的有效方法。哈希集通常用一个称为表的数组来实现。每个表项是对一个或多个元素的引用。哈希函数将元素映射为整数，不同的元素通常被映射为不同的值。（为此，Java为每个对象提供了一个hashCode()方法。）若要添加、删除或者检测一个元素是否为集合的成员，则对该元素使用哈希函数（以表的大小求模），从而确定与该元素相关的表项（称为对元素进行哈希）。

在基于哈希的集合算法中，如果每个表项都与单个元素相关联，则称为开放地址法，如果每个表项指向一个元素集（称为桶），则称为封闭地址法。

任何一个哈希集算法都要解决冲突问题：当两个不同的元素哈希到同一个表项时该如何处理。开放地址算法通常使用另外一个哈希函数来检测可替换的表项。封闭地址算法则将冲突元素放在同一个桶中，直到这个桶变得太满为止。这两种算法有时都需要重新调整表的大小。在开放地址算法中，表有可能变得太满以致无法找到可替代的表项；而在封闭地址算法中，桶有可能变得过大以致无法进行有效的查找。

一些有趣的实践结果表明，在大多数应用中，集合的方法调用遵循如下的分布：contains()占90%，add()占9%，remove()占1%。实际情形中，集合往往可能增大而不是变小，所以，本章集中讨论可扩展的哈希法，即只允许哈希集增大（缩小哈希集将留作习题）。

因为并行化封闭地址哈希集算法相对较为简单，下面先来研究这种算法。

13.2 封闭地址哈希集

编程提示13.2.1 本书中，采用标准类Java List<T>接口（在java.util.List包中）。List<T>元对象T的有序集合，其中T元对象的类型。此处采用了List方法：add(x)将x添加到链表的末尾，get(i)返回（但不删除）位置i的元素，contains(x)则在链表中包含x时返回*true*，还有其他一些方法。

可以用多种类实现List接口。为方便起见，此处采用ArrayList类。

首先通过定义一个对所有的并发封闭地址哈希集公共的基础哈希集来展开我们的讨论。BaseHashSet<T>是一个抽象类，也就是说，不必实现它的所有方法。然后再考虑二种可替换使用的同步技术：一种采用单一的粗粒度锁；一种使用固定大小的锁数组；另一种使用大小可变的锁数组。 300

图13-1描述了基础哈希集的域和构造函数。table[]域是一个桶数组，每个桶是一个用链表实现的集合（第2行）。为了方便起见，此处采用了ArrayList<T>链表，它支持标准的顺序add()，remove()和contains()方法。setSize域是表中元素的个数（第3行）。有时将table[]数组的长度（数组中桶的个数）称为表的容量。

```
1  public abstract class BaseHashSet<T> {
2    protected List<T>[] table;
3    protected int setSize;
4    public BaseHashSet(int capacity) {
5      setSize = 0;
6      table = (List<T>[]) new List[capacity];
7      for (int i = 0; i < capacity; i++) {
8        table[i] = new ArrayList<T>();
9      }
10   }
11   ...
12 }
```

图13-1 BaseHashSet<T>类：域和构造函数

BaseHashSet<T>类中没有实现下列抽象方法：acquire(x)能获得对元素x进行操作时所必需的锁，release(x)则释放这些锁，policy()用于决定是否改变集合的大小，resize()将数组table[]的容量加倍，acquire(x)方法必须是可重入的（第8章8.4节），也就是说，如果一个已调用了acquire(x)的线程再次调用该方法，那么该线程能继续推进而不会与自己发生死锁。

图13-2描述了BaseHashSet<T>类中的contains(x)和add(x)方法。每个方法首先调用acquire(x)进行必要的同步，然后进入try语句块，并由它的finally块调用release(x)。contains(x)方法只是简单地测试x是否在对应的桶中（第17行），若x不在链表中，则由add(x)将其加入（第27行）。

应如何确定桶数组的容量，以确保方法调用所耗费的时间是预期的常数？以add(x)的调用为例。第一步，对x做哈希处理，耗费固定时间。第二步，将该元素添加到桶中，这需要遍历链表。只有当链表的长度为预期的常数时，遍历链表所耗费的时间才会是预期的常数，因此，表的容量必须与表中元素的个数成正比。由于该数目可能随时间发生不可预知的变化，所以若要确保方法调用的时间（或多或少）为常数，必须不断地调整表的大小，以确保链表的长度（或多或少）保持为常数。

还需确定何时调整哈希集以及如何让resize()方法与其他方法同步。对此有多种可行的选择。对封闭地址算法，一种简单的策略就是，当桶的平均大小超出一个固定阈值时，调整

集合的大小，另一种可选的策略是，使用两个固定的整数值，分别作为桶的阈值和全局阈值。

- 如果有一定量的桶（比如说1/4）超出桶的阈值，则将表的容量加倍。
- 如果任意一个桶超过了全局阈值，则将表的容量加倍。

```
13   public boolean contains(T x) {
14     acquire(x);
15     try {
16       int myBucket = x.hashCode() % table.length;
17       return table[myBucket].contains(x);
18     } finally {
19       release(x);
20     }
21   }
22   public boolean add(T x) {
23     boolean result = false;
24     acquire(x);
25     try {
26       int myBucket = Math.abs(x.hashCode() % table.length);
27       if (! table[myBucket].contains(x)) {
28         table[myBucket].add(x);
29         result = true;
30         size++;
31       }
32     } finally {
33       release(x);
34     }
35     if (policy())
36       resize();
37     return result;
38   }
```

图13-2 BaseHashSet<T>类：contains()和add()方法对元素进行哈希及选择桶

在实际应用中，这两种策略和其他策略一样，效果都很好。开放地址算法则稍微复杂一些，稍后讨论。

13.2.1 粗粒度哈希集

图13-3描述了CoarseHashSet<T>类的域、构造函数、acquire(x)方法和release(x)方法。构造函数首先初始化它的超类（第4行）。同步则是由单一的可重入锁（第2行）保证的，该锁可通过acquired(x)获得（第8行），由release(x)释放（第11行）。

```
1  public class CoarseHashSet<T> extends BaseHashSet<T>{
2    final Lock lock;
3    CoarseHashSet(int capacity) {
4      super(capacity);
5      lock = new ReentrantLock();
6    }
7    public final void acquire(T x) {
8      lock.lock();
9    }
10   public void release(T x) {
11     lock.unlock();
12   }
13   ...
14 }
```

图13-3 CoarseHashSet<T>类：域、构造函数、acquire()方法和release()方法

图13-4描述了CoarseHashSet<T>类的policy()和resize()方法。此处采用了一种简单的策略：当桶的平均长度超过4时，则调整表的大小（第16行）。resize方法首先锁定集合（第20行），检查没有其他线程正在对表进行调整（第23行）。然后，它分配并初始化一个容量为原来两倍的新表（第25～29行），并将原表中的元素移到新的桶中（第30～34行）。最后，对集合解锁（第36行）。

```
15    public boolean policy() {
16      return setSize / table.length > 4;
17    }
18    public void resize() {
19      int oldCapacity = table.length;
20      lock.lock();
21      try {
22        if (oldCapacity != table.length) {
23          return; // someone beat us to it
24        }
25        int newCapacity = 2 * oldCapacity;
26        List<T>[] oldTable = table;
27        table = (List<T>[]) new List[newCapacity];
28        for (int i = 0; i < newCapacity; i++)
29          table[i] = new ArrayList<T>();
30        for (List<T> bucket : oldTable) {
31          for (T x : bucket) {
32            table[x.hashCode() % table.length].add(x);
33          }
34        }
35      } finally {
36        lock.unlock();
37      }
38    }
```

图13-4 CoarseHashSet<T>类：policy()方法和resize()方法

13.2.2 空间分带哈希集

和第9章的粗粒度链表一样，上节中的粗粒度哈希集易于实现且很好理解。然而，这种哈希集却是一个顺序瓶颈。方法的调用形成了一次一个的顺序效果，虽然逻辑上没有理由要求这样做。

301
↓
303

下面给出一种具有更高并行性且对锁的争用较低的封闭地址哈希表。我们不再用单一的锁来同步整个集合，而是将集合划分为独立同步的片。为此，需要引入锁分片技术，该技术也适用于其他数据结构。图13-5是StripedHashSet<T>类的域和构造函数。该集合通过一个由L个锁组成的数组locks[]和一个由$N = L$个桶组成的数组table[]进行初始化，其中每个桶都是一个不同步的List<T>。虽然这些数组初始时具有相同的容量，但当重新调整集合时，table[]将会增大，而locks[]却不变。由于要不断地加倍表的容量N，而锁数组的大小L保持不变，因此，锁最终将会保护每个表项j，这里$j = i \pmod L$。acquire(x)方法和release(x)方法使用x的哈希码来决定应获取或释放哪个锁。图13-6表明了如何调整StripedHashSet<T>的大小。

当表增大时，不改变锁数组大小的原因有以下两点：

- 将一个锁和一个表项相关联将耗费大量的空间，特别在表很大且争用低的时候。
- 虽然调整表的大小很简单，但调整锁数组（正在使用）的大小却非常复杂，此问题将在

13.2.3节中讨论。

```
1  public class StripedHashSet<T> extends BaseHashSet<T>{
2    final ReentrantLock[] locks;
3    public StripedHashSet(int capacity) {
4      super(capacity);
5      locks = new Lock[capacity];
6      for (int j = 0; j < locks.length; j++) {
7        locks[j] = new ReentrantLock();
8      }
9    }
10   public final void acquire(T x) {
11     locks[x.hashCode() % locks.length].lock();
12   }
13   public void release(T x) {
14     locks[x.hashCode() % locks.length].unlock();
15   }
```

图13-5 StripedHashSet<T>类：域，构造器，acquire()方法和release()方法

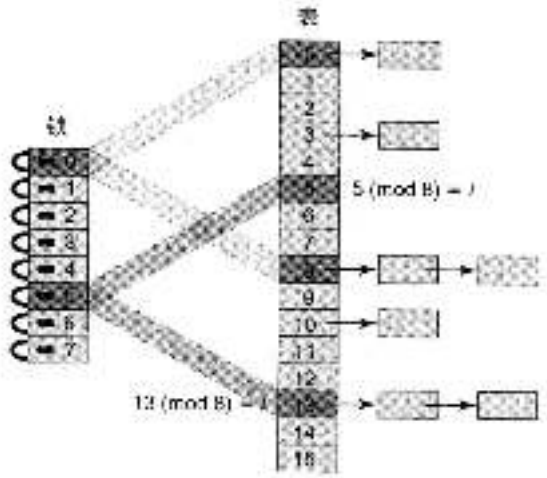


图13-6 调整基于锁的StripedHashSet哈希表。当该表增大时对分片进行调整，以确保每个锁覆盖2^{m}个表项，图中，$N = 16$，$L = 8$。当N从8变为16时，内存将按空间划分，从而使图中的锁$i = 5$可以覆盖两个都为5 mod 8的单元

调整StripedHashSet（列13-7）与调整CoarseHashSet大体上相同，其差别之一就在于，前者的resize()方法是按照升序获取lock[]中的锁（第18~20行）。它与contains()、add()或remove()调用之间不会发生死锁，因为这些方法仅获取单一的锁。它与另外一个resize()调用之间也不会产生死锁，因为它们不需持有任何锁就可以开始，且按照相同的顺序来获取锁。如果两个或更多的线程试图在同一时刻调整集合的大小将会怎样呢？当一个线程准备调整表时，将会和CoarseHashSet<T>一样记录当前的表容量。在它获取了所有锁以后，如果发现其他的某个线程改变了表的容量（第23行），则释放这些锁并放弃操作（由于已经持有所有的锁，有可能只是加倍了表的大小）。

否则，创建一个容量为原表两倍的新数组table[]（第25行），并将原表中的元素移到新表中（第30行）。最后，释放锁（第36行）。由于initializeFrom()方法调用了add()，它可能会触发嵌套的resize()调用，我们把验证嵌套的resize()在此处和后面的哈希集实现中能

正确工作这个问题留作习题。

```
16   public void resize() {
17     int oldCapacity = table.length;
18     for (Lock lock : locks) {
19       lock.lock();
20     }
21     try {
22       if (oldCapacity != table.length) {
23         return; // someone beat us to it
24       }
25       int newCapacity = 2 * oldCapacity;
26       List<T>[] oldTable = table;
27       table = (List<T>[]) new List[newCapacity];
28       for (int i = 0; i < newCapacity; i++)
29         table[i] = new ArrayList<T>();
30       for (List<T> bucket : oldTable) {
31         for (T x : bucket) {
32           table[x.hashCode() % table.length].add(x);
33         }
34       }
35     } finally {
36       for (Lock lock : locks) {
37         lock.unlock();
38       }
39     }
40   }
```

图13.7 StripedHashSet<T>类。为了调整集合的大小，首先按序对每个锁进行加锁，然后检查此时没有其他的线程在调整表

总而言之，空间分析的上领比单一粗粒度锁具有更高的并发性，其原因在于将元素哈希到不同的锁能使方法调用以并行的方式执行。add()、contain()和remove()方法的执行需要预期的固定时间，而resize()需要的是线性时间且是一种"停止一切"的操作：增加表的容量时中止所有的并发方法调用。

13.2.3 细粒度哈希集

当表的大小增长时，如只要细化锁的粒度，使得在一个分片里的单元个数不会连续增长，应该怎么做？显然，如果要调整锁数组的大小，需要另一种形式的同步。由于很少重新调整，所以我们的主要目标就是设计一种方法以允许锁数组大小被调整，同时又不会增加正常方法调用的代价。

图13-8描述了RefinableHashSet<T>类的域和构造函数。为了加入更高级别的同步，引入了一个全局共享域owner，将一个布尔值和一个线程的引用组合在一起。通常情况下，该布尔值为*false*，表示集合没有处于调整状态。当集合被调整时，布尔值为*true*，而与其相关联的引用则指向正在执行调整的线程。这两个值被封装在AtomicMarkableReference <Thread>中，以使它们能被原子地修改（第9章编程提示9.8.1）。采用owner作为resize()方法和其他add()方法之间的互斥标志，能使得在调整大小的过程中不会发生更新，而在更新的过程中不会出现调整。add()的每次调用都必须读owner域，由于很少调整大小，所以通常将owner的值缓存起来。

304
2
305

每个方法通过调用acquire(x)对x的桶加锁，如图13-9所示。该方法一直在自旋，直到不

再有其他线程调整集合的大小为止（第19~21行）。然后读锁数组（第22行）。之后，获取该元素的锁（第24行），并再次进行确认。此时，由于对锁的持有（第26行）能确保没有其他线程在调整集合，所以在第21~26行之间不会发生重新调整的行为。

```
1  public class RefinableHashSet<T> extends BaseHashSet<T>{
2    AtomicMarkableReference<Thread> owner;
3    volatile ReentrantLock[] locks;
4    public RefinableHashSet(int capacity) {
5      super(capacity);
6      locks = new ReentrantLock[capacity];
7      for (int i = 0; i < capacity; i++) {
8        locks[i] = new ReentrantLock();
9      }
10     owner = new AtomicMarkableReference<Thread>(null, false);
11   }
12   ...
13 }
```

图13-8 RefinableHashSet<T>类：域和构造函数

```
14   public void acquire(T x) {
15     boolean[] mark = {true};
16     Thread me = Thread.currentThread();
17     Thread who;
18     while (true) {
19       do {
20         who = owner.get(mark);
21       } while (mark[0] && who != me);
22       ReentrantLock[] oldLocks = locks;
23       ReentrantLock oldLock = oldLocks[x.hashCode() % oldLocks.length];
24       oldLock.lock();
25       who = owner.get(mark);
26       if ((!mark[0] || who == me) && locks == oldLocks) {
27         return;
28       } else {
29         oldLock.unlock();
30       }
31     }
32   }
33   public void release(T x) {
34     locks[x.hashCode() % locks.length].unlock();
35   }
```

图13-9 RefinableHashSet<T>类：acquire()和release()方法

如果通过了这次检测，线程则能继续推进。否则，由于正在执行的更新可能会使线程已获取的锁过时，所以线程将释放这些锁，并重新开始。在重新执行时，若要再次获得锁，先要自旋到当前的协调结束（第19~21行）。release(x)方法释放由acquire(x)方法获取的锁。

这里的resize()方法与StripedHashSet类中的resize()方法几乎相同。它们之间的唯一不同在第46行：方法不再是获得lock[]中的所有锁，而是通过调用quiesce()（图13-10）来确保没有其他的线程正在处于add()、remove()或者contains()的调用中。图13-11给出了quiesce()方法。quiesce()方法访问每个锁，并等待直到它们被开锁为止。

306
|
307

acquire()和resize()方法采用owner标志的mark()域和表的锁数组来保证互斥访问。acquire()首先获取mark()域的锁，然后读mark()域，而resize()则首先设置mark，然后在

quiesce()调用中读该锁。这种次序能确保每个在quiesce()完成之后获取锁的线程将看到该集合正处于调整中，从而回退直到此调整结束。同样，resize()首先设置mark域，然后读这些锁，当add()、remove()或者contains()调用的锁被设置时不再推进。

```
36    public void resize() {
37      int oldCapacity = table.length;
38      boolean[] mark = {false};
39      int newCapacity = 2 * oldCapacity;
40      Thread me = Thread.currentThread();
41      if (owner.compareAndSet(null, me, false, true)) {
42        try {
43          if (table.length != oldCapacity) { // someone else resized first
44            return;
45          }
46          quiesce();
47          List<T>[] oldTable = table;
48          table = (List<T>[]) new List[newCapacity];
49          for (int i = 0; i < newCapacity; i++)
50            table[i] = new ArrayList<T>();
51          locks = new ReentrantLock[newCapacity];
52          for (int j = 0; j < locks.length; j++) {
53            locks[j] = new ReentrantLock();
54          }
55          initializeFrom(oldTable);
56        } finally {
57          owner.set(null, false);
58        }
59      }
60    }
```

图13-10 RefinableHashSet<T>类：resize ()方法

总之，可以设计一种桶的数目和锁的数目能连续调整的哈希表。该算法的限制之一就是，在调整过程中不允许多个线程访问表中的元素。

```
61    protected void quiesce() {
62      for (ReentrantLock lock : locks) {
63        while (lock.isLocked()) {}
64      }
65    }
```

图13-11 RefinableHashSet<T>类：quiesce ()方法

308

13.3 无锁哈希集

下一步工作就是让哈希集的实现是无锁的，并使重调大小是可增量的，也就是说，每个add()方法调用仅仅执行重新调整工作中的一小部分。这样一来，就不需要“停止一切”地调整表的大小了。每个contains()、add()和remove()方法的执行都是预期的固定时间。

为了使可调大小的哈希集是无锁的，仅仅使单个桶无锁是不够的，因为重调表的大小要求原子地将旧桶中的元素移到新桶中。如果表的容量加倍了，就必须在两个新桶间划分旧桶中的元素。如果不是原子地完成这种迁移，那么元素有可能暂时丢失或者重复出现。

在没有锁的情况下，必须使用类似于compareAndSet()的原子方法进行同步。然而，这些方法仅仅对单一的内存单元进行操作，这使得很难原子地将结点从一个链表移到另一个链表，

13.3.1 递归有序划分

下面给出一种采用倒置常规哈希数据结构头的方法来实现的哈希集：

不是在桶间移动元素，而是在元素间移动桶。

更准确地说，类似于第9章的LockFreeList类，将所有的元素保存在一个无锁链表中。桶只是指向链表中的引用。当链表增长时，引入额外的桶引用，使得没有对象离桶的起始点太远。这种算法能确保一旦一个元素被放入链表中，则决不会被删除。但是，元素的插入一定要采用递归有序划分（recursive split-order）算法，下面会简要介绍这个算法。

图13-12b描述了一种无锁哈希集实现。它有两个组成部分：一个无锁链表和一个指向链表的引用组成的扩展数组。这些引用实际上是逻辑桶。哈希集中的任一元素都可以通过从链表的头开始遍历链表而得到，而指向链表的桶引用则提供了访问链表的快捷方式，以使最小化在搜索时需要遍访的表结点的个数。问题的关键在于，当集合中的元素个数增加时，如何确保对链表的桶引用仍然均匀分布，桶引用应该均匀地分布，以使每个结点的访问都花费固定时间。也就是说，必须创建新桶，并将其分配到链表中稀疏覆盖的区域中。

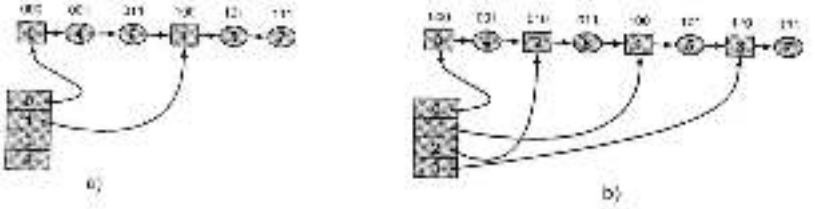


图13-12 该图说明了有序划分的递归特性。a描述了一个由两个桶组成的有序划分链表。由桶组成的数组指向一个链表。有序划分的key值（标于每个结点之上）是通过将元素key值的比特表示反过来得到的。虚线的桶数组项0和1在链表中都有特定的哨兵结点（方形结点），其他结点（常规结点）则是圆形的。元素4（比特值的反序为"001"）和6（比特值的反序为"011"）在桶0中，因为其前采key值的LSB为"0"。元素5和7（比特值的反序分别为"101"和"111"）在桶1中，因为其前采key值的LSB为1。b描述了一旦表的容量由2增加到4，每个桶是如何被划分的。两个新增的桶2和3的比特值反序恰好完全分割了桶0和1

和前面一样，哈希集的容量N永远是2的幂，初始时桶数组容量为2，且除了索引0处的那个桶以外，其他所有的桶引用都为*null*，该引用指向一个空链表。变量bucketSize用于表示这种桶结构的可变容量，桶数组中的每个项在初次访问的时候被初始化，然后指向链表中的结点。

当插入、删除或搜索哈希码为k的元素时，哈希集使用桶索引$k \pmod N$。和前面的哈希集实现一样，也是通过policy()方法来决定何时将表的容量加倍的。但这里是由修改表的方法来增量地调整大小，所以不需要显式的resize()方法。如果表容量为2^i，桶索引则是key的i最低有效位（LSB）；也就是说，每个桶b中元素的哈希码k满足$k = b \pmod{2^i}$。

由于哈希函数依赖于表的容量，所以表容量改变时必须慎重处理。对于在表被改变之前插入的元素，应保证从先前的桶和现在的桶都可以访问。当容量增长为2^{i+1}时，桶b中的元素在两个桶之间被划分：将那些$k = b \pmod{2^{i+1}}$的元素放在b桶中，而$k = b + 2^i \pmod{2^{i+1}}$的元素则被移到桶$b + 2^i$中，该算法的核心思想是：保证这两组元素在链表中是一个接一个地排列的，这样，只需简单地将桶$b + 2^i$设置在第一组元素之后和第二组元素之前，就可实现对桶b的划分。这种方式能保证第二组中的每个元素都可以通过桶b访问。

如图13-12所描述的那样，两个组中的元素由它们的第i个二进制数字位区分开来（从最低有效位向最高有效位数）。数字位为0的元素属于第一组，数字位为1的则属于第二组。下一次哈希表加倍时，将每个组再划分成两个由第$i+1$位区分开来的组，以此类推。例如，元素4

（二进制表示为“100”）和6（“110”）有着相同的最低有效位。当表容量为2^1时，它们处于同一个桶中，但是当表容量变为2^2时，则处于不同的桶中，因为它们的第二位不同。 310

从图13-12中可以看出，这种处理过程能使元素之间保持全序，所以称为递归有序划分。对于给定的key值哈希码，其次序是由它的反向比特值来决定的。

概括地说，有序划分的哈希表是一个由桶组成的数组，每个桶都是一个无锁链表的引用，而链表的结点是按照其哈希码的反序位来排序的。桶的数目是动态增长的，每个新桶在初次访问时被初始化。

为了避免在删除由桶引用指向的结点时出现“角落情形”（corner case），在每个桶的起始位置增加一个哨兵结点，该结点永远不会被删除。假设桶的容量是2^{i-1}。当桶$b+2^i$第一次被访问时，则创建一个键值为$b+2^i$的哨兵结点。该结点通过桶$b+2^i$的父桶b插入到链表中。按照有序划分，$b-2^i$在桶$b+2^i$中的所有元素之前，因为这些元素必须以（$i+1$）个位结束以形成值$b+2^i$。同时，该值也在桶b的所有不属于$b+2^i$的元素之后；它们有着相同的LSB，但它们的第i位为0。因此，这个新的哨兵结点被放置在链表中能精确地将新桶元素和桶b中剩余元素分隔开的位置。为了区分哨兵元素和普通元素，将普通元素的最高有效位（MSB）设置为1，而让哨兵元素的MSB为0。图13-14描述了两个方法：为对象生成一个有序划分key的makeOrdinaryKey()方法以及为桶引用生成一个有序划分key的makeSentinelKey()方法。

图13-13描述了将一个新的key插入集合将会如何初始化一个桶。有序划分的key值用8位字标于结点之上。例如，3的有序划分值是其二进制表示的按位反序，为11000000。方形结点是哨兵结点，对应于那些原始key值为0，1，3（mod 4）且MSB为0的桶。在开启它们的MSB之后，普通（圆形的）结点的有序划分key是原先key值的按位反序。例如，元素9和13在桶“1（mod 4）”中，该桶可以通过插入一个新的结点而被递归地划分为两个桶。图中的次序描述了在表的容量为4，桶0、1和3已被初始化的情况下，加入一个哈希码为10的对象的情形。

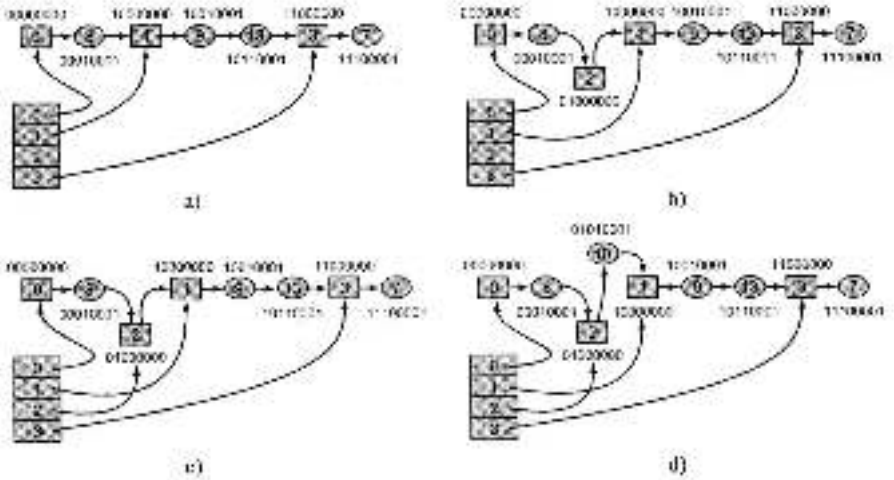


图13-13 add()方法是如何将key值10放入无锁表的。和前面的图一样，有序划分的key值用8位二进制字表示，并标于结点之上。例如，1的有序划分值是它的二进制表示的按位反序。在步骤a中，桶0、1和3已初始化，但桶2还未初始化。在步骤b中，哈希值为10的元素被插入，导致桶2被初始化。插入一个新的有序划分key值为2的哨兵，在步骤c中，将一个新的哨兵赋予桶2。最后，在步骤d中，将普通的有序划分key值10加入到桶2中

表是增量地变大的，也就是说，没有显式的resize操作。每个桶都是一个链表，结点是根据有序划分的哈希值来排序的，前面已经提到，表的调整机制不依赖于决定何时调整大小的策略，为了使这个例子更具体，我们来实现下面的策略：采用一个共享计数器，让add()调用来跟踪桶的平均负载，当平均负载超过阈值时，就将表的容量加倍。

为了避免技术上的差错，将桶数组放在一个固定大小且容量很大的数组中，开始的时候仅使用第一个数组项，随着集合的增长，逐渐使用更多的数组项。当add()方法访问一个在当前表容量下应被初始化但尚未初始化的桶时，则初始化这个桶，虽然思想很简单，但这种设计并不理想，因为固定的数组大小限制了桶的最终数目。实际中，最好是用多级树结构来表示桶，这将能覆盖机器的全部存储空间。我们将该问题留作习题。

311

13.3.2 BucketList类

例13-14描述了BucketList类的域、构造函数和一些有用的方法，该类实现了由有序划分哈希集所使用的无锁链表。尽管该类本质上与LockFreeList类一样，但仍存在两个关键的不同点。第一点不同是BucketList类中的元素是按照递归划分次序排序的，而不是按照哈希值简单地排序，makeOrdinaryKey()和makeSentinelKey()方法（第10行和第14行）说明如何计算这些有序划分的key值。（为了确保反序key值为正数，只使用哈希值的低3位。）图13-15描述了如何使用有序划分key来修改contains()方法。（与LockFreeList类一样，如果x存在，find(x)方法则返回含有结点x的记录以及它的直接前驱结点和后继结点。）

```
1  public class BucketList<T> implements Set<T> {
2    static final int HI_MASK = 0x80000000;
3    static final int MASK = 0x00FFFFFF;
4    Node head;
5    public BucketList() {
6      head = new Node(0);
7      head.next =
8       new AtomicMarkableReference<Node>(new Node(Integer.MAX_VALUE), false);
9    }
10   public int makeOrdinaryKey(T x) {
11     int code = x.hashCode() & MASK; // take 3 lowest bytes
12     return reverse(code | HI_MASK);
13   }
14   private static int makeSentinelKey(int key) {
15     return reverse(key & MASK);
16   }
17   ...
18 }
```

例13-14 BucketList<T>类：域、构造函数和方法

第二点不同就是，LockFreeList类只使用两个哨兵，分别处于链表的两端，而在BucketList<T>类中，每当表的大小被调整时，就将一个哨兵放在新桶的开始位置，这要求能在链表的中间插入哨兵，并能从这些哨兵开始遍历链表。BucketList<T>类提供了一个getSentinel(x)方法（图13-16），该方法以桶引用作为参数，查找

```
19   public boolean contains(T x) {
20     int key = makeOrdinaryKey(x);
21     Window window = find(head, key);
22     Node curr = window.curr;
23     return (curr.key == key);
24   }
```

图13-15 BucketList<T>类：contains()方法

相关的哨兵（如果不存在则插入），并返回从这个哨兵开始的BucketList<T>类的尾。

```
25   public BucketList<T> getSentinel(int index) {
26     int key = makeSentinelKey(index);
27     boolean splice;
28     while (true) {
29       Window window = find(head, key);
30       Node pred = window.pred;
31       Node curr = window.curr;
32       if (curr.key == key) {
33         return new BucketList<T>(curr);
34       } else {
35         Node node = new Node(key);
36         node.next.set(pred.next.getReference(), false);
37         splice = pred.next.compareAndSet(curr, node, false, false);
38         if (splice)
39           return new BucketList<T>(node);
40         else
41           continue;
42       }
43     }
44   }
```

列13-16 BucketList<T>类：getSentinel()方法

13.3.3 LockFreeHashSet<T>类

图13-17描述了LockFreeHashSet<T>类的域和构造函数。该集合具有以下可变域：bucket是一个由指向元素链表的LockFreeHashSet<T>所组成的数组，bucketSize是一个原子的整型数，用来记录当前有多少个bucket数组正在被使用，setSize是一个原子的整型数，记录集合中有多少个对象，以便决定何时调整集合的大小。

```
1  public class LockFreeHashSet<T> {
2    protected BucketList<T>[] bucket;
3    protected AtomicInteger bucketSize;
4    protected AtomicInteger setSize;
5    public LockFreeHashSet(int capacity) {
6      bucket = (BucketList<T>[]) new BucketList[capacity];
7      bucket[0] = new BucketList<T>();
8      bucketSize = new AtomicInteger(2);
9      setSize = new AtomicInteger(0);
10   }
11   ...
12 }
```

图13-17 LockFreeHashSet<T>类：域和构造函数

图13-18描述了LockFreeHashSet<T>类的add()方法。如果x的哈希码为k，add(x)则存取桶k (mod N)，其中，N是当前表的大小，并在必要时初始化该桶（第15行）。然后，调用BucketList<T>的add(x)方法。如果x不存在（第18行），则增加setSize，并检查是否要增加bucketSize——适应桶的数目。contains(x)和remove(x)方法的工作方式基本相同。

列13-19描述了initialBucket()方法，其任务就是在一个特定的索引处初始化bucket数组项，使该数组项指向一个新的哨兵结点。首先创建哨兵结点并将其加入到现有的父桶中，

然后，将一个指向哨兵的引用赋予该数组项。如果父桶未被初始化（第31行），则对该父桶递归地调用initialBucket()。为了控制递归，我们保持父索引小于新桶的索引这一点不变。另外，要谨慎地选取父索引尽可能地最接近新桶的索引，但要小于新桶的索引。我们通过清除桶索引的最高非零有效位来计算该索引（第39行）。

```
13  public boolean add(T x) {
14    int myBucket = BucketList.hashCode(x) % bucketSize.get();
15    BucketList<T> b = getBucketList(myBucket);
16    if (!b.add(x))
17      return false;
18    int setSizeNow = setSize.getAndIncrement();
19    int bucketSizeNow = bucketSize.get();
20    if (setSizeNow / bucketSizeNow > THRESHOLD)
21      bucketSize.compareAndSet(bucketSizeNow, 2 * bucketSizeNow);
22    return true;
23  }
```

图13-18 LockFreeHashSet<T>类：add()方法

```
24  private BucketList<T> getBucketList(int myBucket) {
25    if (bucket[myBucket] == null)
26      initializeBucket(myBucket);
27    return bucket[myBucket];
28  }
29  private void initializeBucket(int myBucket) {
30    int parent = getParent(myBucket);
31    if (bucket[parent] == null)
32      initializeBucket(parent);
33    BucketList<T> b = bucket[parent].getSentinel(myBucket);
34    if (b != null)
35      bucket[myBucket] = b;
36  }
37  private int getParent(int myBucket) {
38    int parent = bucketSize.get();
39    do {
40      parent = parent >> 1;
41    } while (parent > myBucket);
42    parent = myBucket - parent;
43    return parent;
44  }
```

图13-19 LockFreeHashSet<T>类：如果一个桶未被初始化，则通过加入一个新哨兵的办法进行初始化。对一个桶的初始化可能要初始化它的父桶

add()、remove()和contains()方法需要预期的常数操作步来找到一个key（或者决定该key不存在）。为了在一个bucketSize为N的表中初始化一个桶，initialBucket()方法可能要递归地初始化（即划分）$O(\log N)$个父桶，从而允许插入一个新桶。图13-20是一个采用这种递归初始化方式的示例。在a中，表有4个桶，只有桶0被初始化。在b中，插入key值为7的元素。现在要求初始化桶3，进而导致递归地初始化桶1，在c中，桶1被初始化。最后，在d中，桶3被初始化。尽管在这种情形下总的复杂度是对数级而不是常数，但可以看出，任意这种划分的递归序列的期望长度为常数，从而对所有哈希集操作的总预期复杂度为常数。

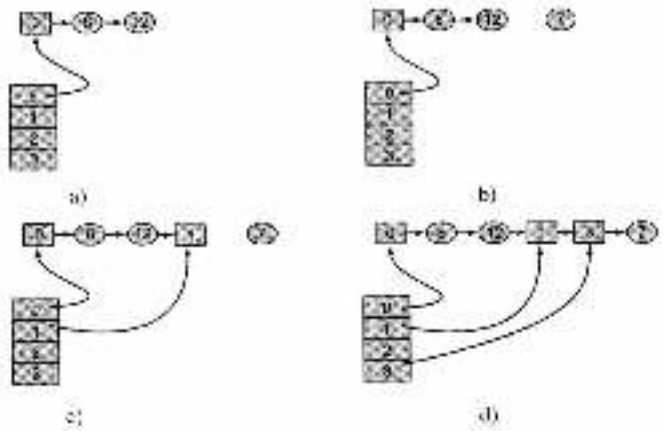


图13-20　无锁哈希表桶的递归初始化。在a中，表中有4个桶，只有桶0被初始化。在b中，准备插入key值为7的元素。现在要求初始化桶3，进而导致递归地初始化桶1。在c中，对桶1的初始化通过先向链表中加入哨兵1，然后设置桶指向该哨兵来完成。然后，在d中，以同样的方式初始化桶3，最后7被加入到链表中。在最坏情况下，插入一个元素可能要递归地初始化表容量对数的桶，但可以看出，这种递归序列的长度为常数

13.4　开放地址哈希集

下面转而研究并发开放哈希算法。在开放哈希结构中，每个表项为一个元素而不是一个集合，与封闭哈希相比，该结构似乎更难并发。我们的并发算法是基于一种称为Cuckoo（布谷）哈希的顺序算法实现的。

13.4.1　Cuckoo哈希

Cuckoo哈希是一种顺序哈希算法，其中，最近加入的元素将取代先前在同一位置的元素[⊖]。简单地说，表是一个由元素组成的有k个表项的数组。对于一个大小为$N = 2k$的哈希集，我们使用一个2表项的数组table[][⊜]和两个独立的哈希函数

$$h_0, h_1 : KeyRange \to 0, \cdots, k-1$$

316

（在代码中表示为hash0()和hash1()）将可能的key值集合映射到数组项中。为了测试x是否在集合中，contains(x)将检查table[0][$h_0(x)$]或table[1][$h_1(x)$]是否等于x。同样，remove(x)将检查x是否在table[0][$h_0(x)$]或者table[1][$h_1(x)$]中，如果查找到x，则删除它。

add(x)方法（图13-21）十分有趣。它能成功地"踢出"冲突元素，直到每个key都有一个槽为止。为了加入x，该方法将x和table[0][$h_0(x)$]的当前值y相交换（第6行）。如果原来的y值为*null*，则该过程结束（第7行）。否则，就用同样的方法将table[1][$h_1(y)$]的当前值作为最近"无巢"的y（第8行）。和前面一样，如果原先的值为*null*，那么该过程结束。否则，该方法将继续交换表项（交互的表）直到找到一个空闲位置为止。图13-22给出了这种置换序列的一个示例。

⊖　布谷是一种在北美和欧洲可见的鸟（不是时钟）。它们大多是巢穴入侵者：将自己的蛋产在其他鸟的巢穴中。布谷的幼鸟孵化得很早，它们会很快将其他的蛋推到巢穴之外。

⊜　将表划分为两个数组有助于体现并发算法。对于相同数目的哈希项，所使用的有序的Cuckoo哈希算法仅有一个大小为2k的数组。

```
1  public boolean add(T x) {
2    if (contains(x)) {
3      return false;
4    }
5    for (int i = 0; i < LIMIT; i++) {
6      if ((x = swap(0, hash0(x), x)) == null) {
7        return true;
8      } else if ((x = swap(1, hash1(x), x)) == null) {
9        return true;
10     }
11   }
12   resize();
13   add(x);
14 }
```

图13-21 顺序Cuckoo哈希：add()方法

我们有可能找不到空闲的位置，因为表是满的，或者由于置换序列构成了一个循环。因此，需要制定一个成功置换的上限（第5行），当超过这个限度时，重新调整哈希表的大小，选用新的哈希函数（第12行），重新开始（第13行）。

顺序Cuckoo哈希的吸引人之处就在于其简单性。该实现提供了常数时间的contains()和remove()方法，可以证明，随着时间的推移，由每个add()调用所导致的置换的平均数目将是一个常数。实验结果表明，顺序Cuckoo哈希在实际应用中具有良好的效果。

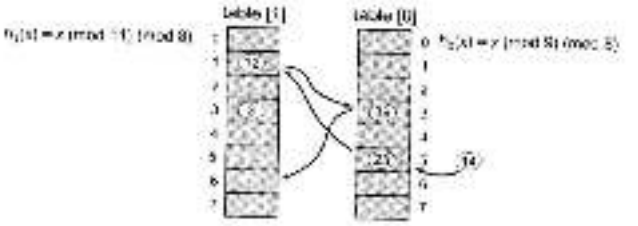


图13-22 当一个key值为14的元素发现Table[0][$h_0(14)$]和Table[1][$h_1(14)$]这两个单元已被23和25占用时，开始一个置换序列。当键值为39的元素成功地放入Table[1][$h_1(39)$]时，该序列结束

13.4.2 并发Cuckoo哈希

让顺序Cuckoo哈希算法并发执行的主要问题在于，add()方法需要执行一个较长的交换序列。为了解决这一问题，下面定义另一种Cuckoo哈希算法：PhasedCuckooHashSet<T>类。每个方法调用被分解为一系列阶段，每个阶段添加、移除或者替换一个元素x。

我们不再将集合组织成由元素组成的二维表，而是使用由测试集组成的一维表。其中，测试集是一个由具有相同哈希码的元素所组成的常数大小的集合。每个测试集最多有PROBE_SIZE个元素，算法则试图保证当集合处于静态（即没有方法调用在执行）时，每个测试集的元素不超过THRESHOLD < PROBE_SIZE个。图13-24为PhasedCuckooHashSet数据结构的一个示例，其中，PROBE_SIZE为4，THRESHOLD为2。当方法调用在进行时，一个测试集有可能暂时具有多于THRESHOLD但决不会大于PROBE_SIZE个元素。（本例中，将每个测试集作为一个固定大小的List<T>来实现。）列13-23为PhasedCuckooHashSet<T>的域和构造函数。

为了推迟关于同步的讨论，PhasedCuckooHashSet<T>类被定义为抽象类，也就是说，它没有实现它所定义的方法。PhasedCuckooHashSet<T>类具有与BaseHashSet<T>类相同的抽象方法：acquire(x)方法获取操作元素x所需的全部锁，release(x)释放这些锁，resize()则重新调整集合的大小。（和前面一样，要求acquire(x)是可重入的。）

概括地说，PhasedCuckooHashSet<T>的工作过程如下：首先对两个表中相关联的测试集上锁，然后添加和删除元素。

```
1  public abstract class PhasedCuckooHashSet<T> {
2    volatile int capacity;
3    volatile List<T>[][] table;
4    public PhasedCuckooHashSet(int size) {
5      capacity = size;
6      table = (List<T>[][]) new java.util.ArrayList[2][capacity];
7      for (int i = 0; i < 2; i++) {
8        for (int j = 0; j < capacity; j++) {
9          table[i][j] = new ArrayList<T>(PROBE_SIZE);
10       }
11     }
12   }
13   ...
14 }
```

图13-23 PhasedCuckooHashSet<T>类：域和构造函数

318

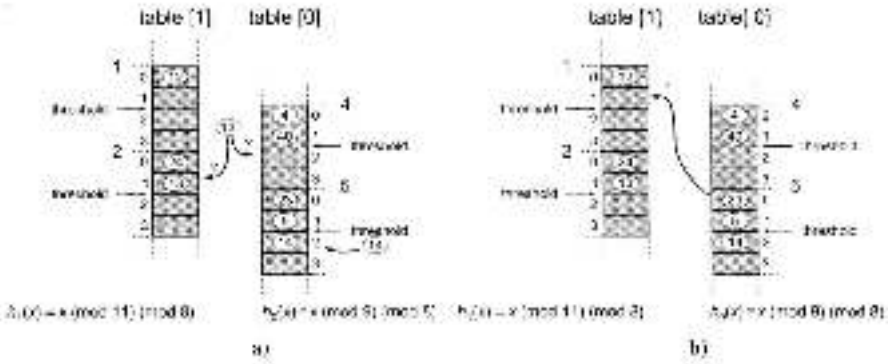


图13-24 PhasedCuckooHashSet<T>类：add()和relocate()方法。该图描述了由8个大小为4的测试集组成的数组表，其中阈值为2，图中显示的是Table[0][]的测试集4和5，以及Table[1][]的测试集1和2。在a中，key值为13的元素发现Table[0][4]和Table[1][2]超出了阈值，于是将该元素加入到测试集Table[1][2]。另一方面，key值为14的元素发现它的两个测试集都超出了阈值，于是将它的元素加入测试集Table[0][5]，并且产生该元素需要重新分配的信号。在b中，方法尝试重新分配key值为23的元素，也就是Table[0][5]中最老的元素。由于Table[1][1]未超出阈值，所以该元素被成功地重新分配。如果Table[1][1]超出阈值，算法将试图重新分配Table[1][1]中的元素12。如果Table[1][1]的测试集大小被限为4个元素，那么它将试图重新分配元素5，也就是Table[0][5]中下一个最老的元素

和在顺序算法中一样，若要删除一个元素，检查该元素是否在一个测试集中，如果在，则删除。若要增加一个元素，则将该元素增加到一个测试集中。一个元素的测试集就如同临时的溢出缓冲区，为向表中增加元素时可能出现的连续置换的长序列服务。在顺序算法中，THRESHOLD值本质上是测试集的大小。如果测试集中已经有这么多个元素，该元素还是会被添加到PROBE_SIZE-THRESHOLD的一个溢出槽中。随后，算法尝试从测试集中重新分配另一个元素。可以采用不同的策略来选择那一个元素将被重新分配。此处，首先是移出最老的元素，直到测试集在阈值限度之内为止。和顺序Cuckoo哈希算法一样，一次重分配可能触发另一个，以此类推。图13-24为PhasedCuckooHashSet <T>类的一次执行示例。

图13-25为PhasedCuckooHashSet<T>类的remove(x)方法。它调用抽象的acquire(x)方法以获取必要的锁，然后进入try块。其finally语句块则调用release(x)。在try语句块中，方

法简单地检查x是否在table[0][$h_0(x)$]或者table[1][$h_1(x)$]中。如果是，它删除x并返回*true*，否则返回*false*。contains(x)方法的执行采用类似的方式。

```
15    public boolean remove(T x) {
16      acquire(x);
17      try {
18        List<T> set0 = table[0][hash0(x) % capacity];
19        if (set0.contains(x)) {
20          set0.remove(x);
21          return true;
22        } else {
23          List<T> set1 = table[1][hash1(x) % capacity];
24          if (set1.contains(x)) {
25            set1.remove(x);
26            return true;
27          }
28        }
29        return false;
30      } finally {
31        release(x);
32      }
33    }
```

图13-25 PhasedCuckooHashSet<T>类：remove()方法

图13-26描述了add(x)方法。和remove()方法一样，它调用acquire(x)以获取必要的锁。

```
34    public boolean add(T x) {
35      T y = null;
36      acquire(x);
37      int h0 = hash0(x) % capacity, h1 = hash1(x) % capacity;
38      int i = -1, h = -1;
39      boolean mustResize = false;
40      try {
41        if (present(x)) return false;
42        List<T> set0 = table[0][h0];
43        List<T> set1 = table[1][h1];
44        if (set0.size() < THRESHOLD) {
45          set0.add(x); return true;
46        } else if (set1.size() < THRESHOLD) {
47          set1.add(x); return true;
48        } else if (set0.size() < PROBE_SIZE) {
49          set0.add(x); i = 0; h = h0;
50        } else if (set1.size() < PROBE_SIZE) {
51          set1.add(x); i = 1; h = h1;
52        } else {
53          mustResize = true;
54        }
55      } finally {
56        release(x);
57      }
58      if (mustResize) {
59        resize(); add(x);
60      } else if (!relocate(i, h)) {
61        resize();
62      }
63      return true; // x must have been present
64    }
```

图13-26 PhasedCuckooHashSet<T>类：add()方法

然后进入一个try语句块，其finally语句块调用release(x)。如果该元素已经存在（第41行），则返回*false*。如果元素的任一测试集都在阈值限度之内（第44行和第46行），则增加元素并返回。否则，如果有一个元素的测试集超出了阈值但还没有满（第48行和第50行），则增加该元素并做上记号以便稍后重新平衡测试集。如果两个集合都满了，则做上记号以重新调整整个集合的大小（第53行）。然后，释放x上的锁（第56行）。

319 ↓ 320

如果元素x的两个测试集都是满的从而使方法不能增加x，则重新调整哈希集的大小，然后再次进行尝试（第58行）。如果测试集的第r行第c列超出了阈值，则调用relocate(r, c)（稍后将详细介绍），以重新平衡测试集的大小。如果这个调用返回*false*，说明无法再度平衡测试集，则使用add()来调整表的大小。

relocate()方法如图13-27所示。它使得测试集的行、列坐标看起来具有多于THRESHOLD个元素，并尝试通过将该测试集中的元素移到可选择的测试集中来将其大小减小到阈值之下。

```
65    protected boolean relocate(int i, int hi) {
66      int hj = 0;
67      int j = 1 - i;
68      for (int round = 0; round < LIMIT; round++) {
69        List<T> iSet = table[i][hi];
70        T y = iSet.get(0);
71        switch (i) {
72        case 0: hj = hash1(y) % capacity; break;
73        case 1: hj = hash0(y) % capacity; break;
74        }
75        acquire(y);
76        List<T> jSet = table[j][hj];
77        try {
78          if (iSet.remove(y)) {
79            if (jSet.size() < THRESHOLD) {
80              jSet.add(y);
81              return true;
82            } else if (jSet.size() < PROBE_SIZE) {
83              jSet.add(y);
84              i = 1 - i;
85              hi = hj;
86              j = 1 - j;
87            } else {
88              iSet.add(y);
89              return false;
90            }
91          } else if (iSet.size() >= THRESHOLD) {
92            continue;
93          } else {
94            return true;
95          }
96        } finally {
97          release(y);
98        }
99      }
100     return false;
101   }
```

图13-27 PhasedCuckooHashSet<T>类：relocate()方法

该方法在放弃之前要进行固定次数（LIMIT）的尝试。在每一次循环中，都要持以下不变

大；iSet是要尝试着减小的测试集，y是iSet中最老的元素，jSet是可能包含y的另一个测试集。该循环识别y（第70行），它对y可能进入的两个测试集加锁（第75行），尝试着从这些测试集中删除y（第78行）。如果成功（在第70行和第78行之间，另一个线程有可能已经删除y），则准备将y加入到jSet中。如果jSet在阈值限度之内（第79行），则将y增加到jSet中并返回*true*（不必调整大小）。如果jSet超出阈值但是没有满（第82行），则尝试着通过交换iSet和jSet来减小jSet（第82~86行），并继续这个循环。如果jSet是满的（第87行），就将y放回至iSet中，并返回*false*（触发一次重新调整大小的过程）。否则，则尝试通过交换iSet和jSet来减小jSet（第82~86行）。如果在第78行无法成功地删除y，则重新检查iSet的大小。如果仍然超出阈值（第91行），则继续这个循环，并尝试再次删除一个元素。否则，iSet是小于阈值的，将会返回*true*（不必调整大小）。列13-24为PhasedCuckooHashSet<T>的一次执行示例，其中，key值14引起测试集table[0][5]中最老的元素23的一次重新分配。

13.4.3 空间分带的并发Cuckoo哈希

首先来考虑一种采用锁分片技术（第13章13.2.2节）的并发Cuckoo哈希集实现。StripedCuckooHashSet类扩展了PhasedCuckooHashSet类，提供一个固定的$2 \times L$的可重入锁数组。一般而言，lock[i][j]保护table[i][j]，其中$k \pmod L = j$。列13-28为StripedCuckooHashSet类的域和构造函数。该构造函数调用PhasedCuckooHashSet<T>构造函数（第4行），然后初始化锁数组。

```
1  public class StripedCuckooHashSet<T> extends PhasedCuckooHashSet<T> {
2    final ReentrantLock[][] lock;
3    public StripedCuckooHashSet(int capacity) {
4      super(capacity);
5      lock = new ReentrantLock[2][capacity];
6      for (int i = 0; i < 2; i++) {
7        for (int j = 0; j < capacity; j++) {
8          lock[i][j] = new ReentrantLock();
9        }
10     }
11   }
12   ...
13 }
```

321
322

列13.28 StripedCuckooHashSet类：域和构造函数

StripedCuckooHashSet类的acquire(x)方法（列13-29）按照这种次序对lock[0][$h_0(x)$]和lock[1][$h_1(x)$]上锁，从而避免死锁，release(x)方法释放这些锁。

```
14   public final void acquire(T x) {
15     lock[0][hash0(x) % lock[0].length].lock();
16     lock[1][hash1(x) % lock[1].length].lock();
17   }
18   public final void release(T x) {
19     lock[0][hash0(x) % lock[0].length].unlock();
20     lock[1][hash1(x) % lock[1].length].unlock();
21   }
```

图13.29 StripedCuckooHashSet类：acquire()和release()方法

StripedCuckooHashSet中的resize()方法（图13-30）和PhasedCuckooHashSet中的resize()方法的唯一区别就是，后者要求按升序对lock[0]上锁（第24行）。以这种次序上锁能确保在add()、remove()或者contains()调用中间没有别的线程，从而避免与其他并发的resize()调用产生死锁。

```
22   public void resize() {
23     int oldCapacity = capacity;
24     for (Lock aLock : lock[0]) {
25       aLock.lock();
26     }
27     try {
28       if (capacity != oldCapacity) {
29         return;
30       }
31       List<T>[][] oldTable = table;
32       capacity = 2 * capacity;
33       table = (List<T>[][]) new List[2][capacity];
34       for (List<T>[] row : table) {
35         for (int i = 0; i < row.length; i++) {
36           row[i] = new ArrayList<T>(PROBE_SIZE);
37         }
38       }
39       for (List<T>[] row : oldTable) {
40         for (List<T> set : row) {
41           for (T z : set) {
42             add(z);
43           }
44         }
45       }
46     } finally {
47       for (Lock aLock : lock[0]) {
48         aLock.unlock();
49       }
50     }
51   }
```

例13-30 StripedCuckooHashSet类：resize()方法

13.4.4 细粒度的并发Cuckoo哈希集

同样可以使用第13章13.2.3节中的方法来调整锁数组的大小。本小节介绍RefinableCuckooHashSet类（图13-31），如同RefinableHashSet类一样，需引入一个AtomicMarkableReference<Thread>类型的owner域，它将一个布尔值和一个线程的引用组合在一起。如果布尔值为*true*，则集合正在调整中，而引用则指向正在负责调整大小的线程。

每个阶段通过调用acquire(*x*)来锁定*x*的桶，如图13-32所示。首先读锁数组（第24行），然后自旋直到没有其他线程在调整集合大小（第21－23行）。接着获取元素的两个锁（第27行和第28行）。并检查该锁数组是否未被改变（第30行）。如果锁数组在第24-30行之间未被改变，那么该线程已获得了它继续执行所需的锁。否则，它所获得的锁就是过时的，必须释放它们，并重新开始。release(*x*)方法释放由acquire(*x*)方法获取的锁。

```
1  public class RefinableCuckooHashSet<T> extends PhasedCuckooHashSet<T>{
2    AtomicMarkableReference<Thread> owner;
3    volatile ReentrantLock[][] locks;
4    public RefinableCuckooHashSet(int capacity) {
5      super(capacity);
6      locks = new ReentrantLock[2][capacity];
7      for (int i = 0; i < 2; i++) {
8        for (int j = 0; j < capacity; j++) {
9          locks[i][j] = new ReentrantLock();
10       }
11     }
12     owner = new AtomicMarkableReference<Thread>(null, false);
13   }
14   ...
15 }
```

图13.31 RefinableCuckooHashSet<T>：域和构造函数

```
16   public void acquire(T x) {
17     boolean[] mark = {true};
18     Thread me = Thread.currentThread();
19     Thread who;
20     while (true) {
21       do { // wait until not resizing
22         who = owner.get(mark);
23       } while (mark[0] && who != me);
24       ReentrantLock[][] oldLocks = locks;
25       ReentrantLock oldLock0 = oldLocks[0][hash0(x) % oldLocks[0].length];
26       ReentrantLock oldLock1 = oldLocks[1][hash1(x) % oldLocks[1].length];
27       oldLock0.lock();
28       oldLock1.lock();
29       who = owner.get(mark);
30       if ((!mark[0] || who == me) && locks == oldLocks) {
31         return;
32       } else {
33         oldLock0.unlock();
34         oldLock1.unlock();
35       }
36     }
37   }
38   public void release(T x) {
39     locks[0][hash0(x)].unlock();
40     locks[1][hash1(x)].unlock();
41   }
```

图13.32 RefinableCuckooHashSet<T>：acquire()和release()方法

resize()方法（图13-33）与StripedCuckooHashSet类中的resize()方法几乎相同。唯一的区别就是，lock[]数组是二维的。

323 ~ 324 和RefinableHashSet类中的quiesce()方法一样，quiesce()方法（见图13-34）访问每个锁并等待直到它们被释放。唯一的不同在于，它只访问lock[0]中的锁。

```
42   public void resize() {
43     int oldCapacity = capacity;
44     Thread me = Thread.currentThread();
45     if (owner.compareAndSet(null, me, false, true)) {
46       try {
47         if (capacity != oldCapacity) { // someone else resized first
48           return;
49         }
50         quiesce();
51         capacity = 2 * capacity;
52         List<T>[][] oldTable = table;
53         table = (List<T>[][]) new List[2][capacity];
54         locks = new ReentrantLock[2][capacity];
55         for (int i = 0; i < 2; i++) {
56           for (int j = 0; j < capacity; j++) {
57             locks[i][j] = new ReentrantLock();
58           }
59         }
60         for (List<T>[] row : table) {
61           for (int i = 0; i < row.length; i++) {
62             row[i] = new ArrayList<T>(PROBE_SIZE);
63           }
64         }
65         for (List<T>[] row : oldTable) {
66           for (List<T> set : row) {
67             for (T z : set) {
68               add(z);
69             }
70           }
71         }
72       } finally {
73         owner.set(null, false);
74       }
75     }
76   }
```

图13-33 RefinableCuckooHashSet<T>: resize()方法

```
78   protected void quiesce() {
79     for (ReentrantLock lock : locks[0]) {
80       while (lock.isLocked()) {}
81     }
82   }
```

图13.34 RefinableCuckooHashSet<T>: quiesce()方法

13.5 本章注释

术语不相交的并行访问（disjoint-access parallelism）是由Amos Israeli和Lihu Rappoport[76]创造的。Maged Michael[114]已经证明，为每个桶使用一个读者-写者锁[113]的简单算法有着合理的性能，不需要重新调整大小。基于有序划分（见13.3.1节）的无锁哈希集是由Ori Shalev和Nir Shavit[141]提出的。乐观的细粒度哈希集则来自于Doug Lea[99]提出的哈希集实现，并用在java.util.concurrent中。

其他的并发封闭地址设计包括Meichun Hsu和Wei-Pang Yang[72]、Vijay Kumar[87]、Carla Schlatter Ellis[38]以及Michael Greenwald[48]。Hui Gao、Jan Friso Groote和Wim

Hesselink[44]提出了一个几乎无等待的可扩展开放地址哈希算法，Chris Purcell和Tim Harris[129]提出了一种具有开放地址的并发无阻塞哈希表，Cuckoo哈希由Rasmus Pagh和Flemming Rodler[122]提出，目前的版本则是由Maurice Herlihy，Nir Shavit和Moran Tzafrir[68]完成的。

13.6 习题

325 ↓ 326

习题158. 修改StripedHashSet，使其允许通过读/写锁同整锁数组的大小。

习题159. 对于LockFreeHashSet，试给出一个实例来说明，如果我们在每个桶的开始处不增加一个不被删除的哨兵项的话，那么在删除由项引用指向的表项时，将会出现什么样的问题。

习题160. 对于LockFreeHashSet，当访问大小为N的表中的一个未被初始化的桶时，有可能要递归地初始化（即划分）$O(\log N)$个父桶，以允许插入一个新桶。给出一个这种情形的实例。解释为什么任意的这种划分的递归序列，其预期长度是常数。

习题161. 对于LockFreeHashSet，试设计一个无锁的数据结构来替换固定大小的桶数组。你的数据结构必须允许任意数目的桶。

习题162. 概述LockFreeHashSet的add()、remove()和contains()方法的正确性证明。

327

提示：可以假定LockFreeList算法中的方法是正确的。

跳表和平衡查找

14.1 引言

前面学习了几种基于链表和哈希表的集合的并发实现，本章将介绍具有对数级深度的并发查找结构。在文献中已有很多并发的对数级查找结构，本章着重于内存数据（而不是存放在类似磁盘这样的外部存储器上的数据）的查找结构。

很多流行的顺序查找结构，比如红黑树或AVL树，需要定期的重新平衡以保持结构的对数级深度，重新平衡对于基于树的顺序查询结构效果显著，但对于并发结构则可能引发瓶颈和竞争。本章主要针对一种已证明不需要重新平衡就能提供期望的对数级查找时间的数据结构：跳表（SkipList）。下面将介绍两种SkipList实现：惰性跳表（LazySkipList）类是一种基于锁的实现，而无锁跳表（LockFreeSkipList）类则不是。在这两种算法中，最常用的典型方法是contains()，它是无等待的，用于查找一个元素。这两种结构都采用了前面第9章的设计模式。

14.2 顺序跳表

为简单起见，我们把链表看作是一个集合，这意味着键值是唯一的。SkipList是一个由已排序链表所组成的集合，它巧妙地模仿了平衡查找树。SkipList中的结点按照键值进行排序，每个结点都被链接到链表的一个子集中。每个链表都有一个级别，从0到最大值。最底层的链表包含所有的结点，每个高层链表都是低层链表的子链表。图14-1表示一个具有整型键值的SkipList。高层链表是进入低层链表的快速方式。这是因为大致来说，每个处于层i的链表都大约跳过2^i个低一层链表中的结点（例如，图14-1显示的SkipList中，层3中的每个引用都跳过2^3个结点）。在给定层的任意两个结点之间，其下一层的结点数目是固定的，因此，SkipList的总高度大约为结点数的对数。我们可以按照以下方式找到给定的键值：首先查找较高层的链表，跳过大量的低层结点，再逐渐下降，直到在最底层找到（或没有）具有目标键值的结点。

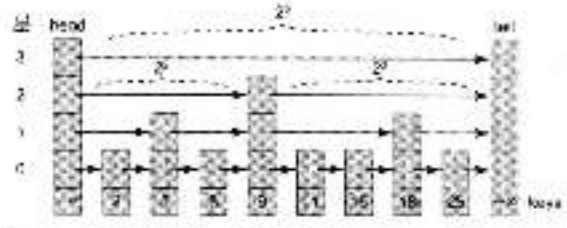


图14-1 SkipList类：本例有4个链表层。每个结点有一个键值，head和tail哨兵的键值为$-\infty$，高层的链表是一种快捷方式，每个引用跳过2^i个下一层链表的结点。例如，在层3，引用跳过了2^3个结点，在层2，跳过了2^2个，以此类推

SkipList是一种概率数据结构（没有人知道不用随机性如何来提供这种性能），每个结点都用一个随机的顶层（topLevel）来创建，并属于直到该层的所有链表。确定了顶层之后，每一层链表中结点的期望个数呈指数递减。令$0<p<1$是层i中的一个结点出现在i+1

层的条件概率。因为所有的结点都出现在0层，那么一个0层中的结点出现在i（$i>0$）层的概率则为p^i。例如，对于$p=1/2$，期望所有结点的1/2出现在1层，结点的1/4出现在2层，以此类推，从而除了不需复杂的全局重构之外，提供了类似基于树的经典顺序查找结构所具有的平衡性质。

我们将head和tail哨兵结点以允许的最大高度放在链表的起始和结束位置。初始时，SkipList为空时，在每一个层，head（左边的哨兵）是tail（右边的哨兵）的前驱结点。head的键值小于任何可能被添加到集合中的结点的键值，tail的键值则为最大值。

每个SkipList结点的next域是一个由引用组成的数组，每个数组对应一个该结点所属的链表。这样，查找一个结点就意味着查找它的前驱和后继。对SkipList的搜索总是从head开始，find()方法一个接一个地顺着层次向下推进，每个层的遍历则采用类似LazyList的方式，使用指向前驱结点的引用pred和指向当前结点的引用curr。一旦找到一个具有更大或匹配键值的结点，则将pred和curr作为结点的前驱和后继记录在数组preds[]和succs[]中，然后继续进入下一层。该遍历在最底层终止。图14-2a表示一次顺序的find()调用。

330

为了将一个结点加入到SkipList中，find()调用将填写preds[]和succs[]数组。创建这个新结点，并链接在它的前驱和后继之间。图14-2b描述了add(12)调用。

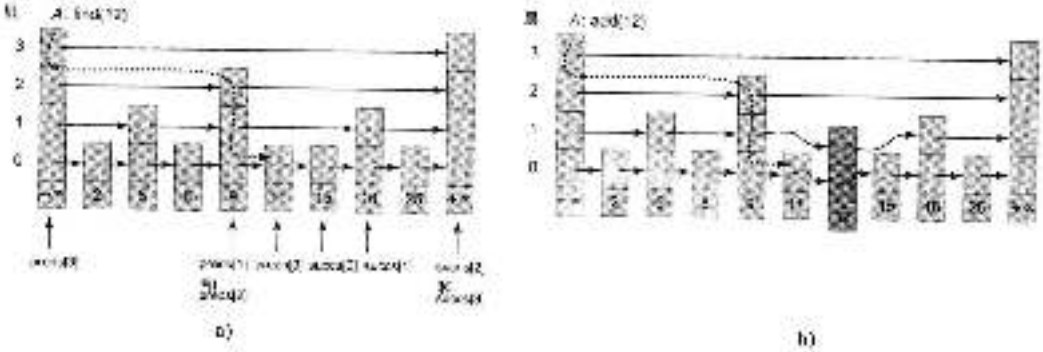


图14-2 SkipList类：find()和add()方法。在a中，find()从最高层开始，只要curr小于或等于目标键值12，则对每个层进行遍历。否则，它将pred和curr保存在每一层的preds[]和succs[]数组中，然后向下进入到低一层，例如，键值为9的结点是preds[2]和preds[1]，其tail为succs[2]，键值为18的结点是succs[1]。这里，由于在最底层的链表中找不到键值为12的结点，所以find()返回*false*，同样，b中的add(12)调用可以继续前进。在b中，一个新的结点以随机的topLevel = 2被创建。该新结点的next引用重新指向对应的succs[]结点，每个前驱结点的next引用重新指向该新结点

为了从跳表中删除一个牺牲结点，find()方法将对该牺牲结点的preds[]和succs[]数组进行初始化，然后通过让每个前驱的next引用重新指向该牺牲结点的后继，将该牺牲结点从所有层的链表中删除。

14.3 基于锁的并发跳表

下面介绍第一种并发跳表实现，即LazySkipList类。该类建立在第9章LazyList算法的基础之上：LazyList结构的每个层都是一个LazyList，和LazyList算法一样，LazySkipList类的add()方法和remove()方法也采用了乐观的细粒度方式来上锁，其contains()方法是无等待的。

14.3.1 简介

下面是LazySkipList类的概述。我们从图14-3开始。和LazyList类一样，每个结点都有

其自己的锁和一个marked域。这个域用来标记结点是否在抽象集中，或者已经被逻辑删除了。 331
任何时候，该算法都保持着跳表特性：高层链表总是包含在低层链表中。

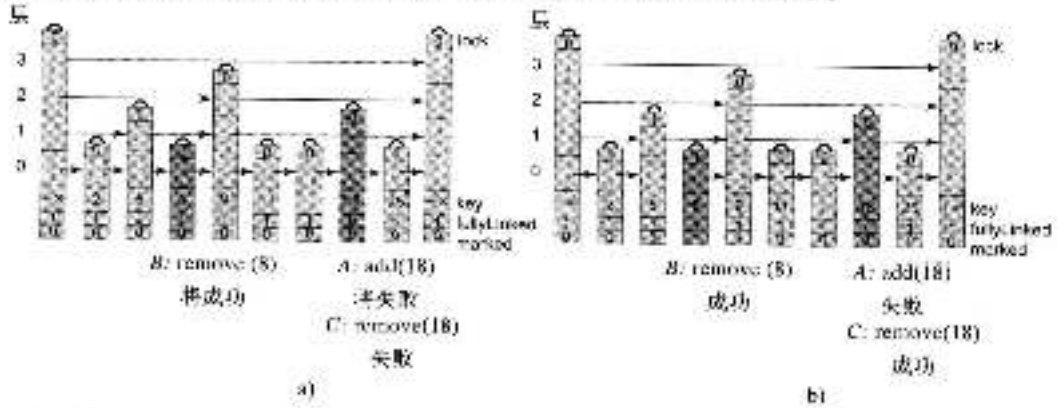


图14-3 LazySkipList类：失败和成功的add()和remove()调用。在a中，add(18)发现键值为18的结点未被标记且不是fullyLinked（完全链接），它将自旋等待直到该结点在b中变为fullyLinked为止，此时返回*false*。在a中，remove(8)发现键值为8的结点未被标记且是完全链接的，这意味着在b中它可以获得该结点的锁。于是，设置标志位，并继续对结点的前驱上锁，此时，结点的键值为5。一旦前驱结点被锁定，则通过重新调整最高层键值为5的结点的引用，将该结点从链表中物理地删除，从而完成成功的remove()。a中的remove(18)是失败的，因为它发现结点不是完全链接的，同样的remove(18)在b中则是成功的，这为它发现该结点是完全链接的

跳表特性是通过采用锁来防止在结点被增加或删除时对它附近的结构改变来保持的，将对结点的访问推迟到该结点被插入到链表的所有层为止。

要增加一个结点，必须在多个层将该结点链接到链表中。每个add()都要调用find()，对跳表进行遍历并返回结点在所有层的前驱和后继。增加结点时，为了防止对其前驱的改变，add()要锁定其前驱结点，以确保该锁定的前驱仍然指向它们的后继，然后按照类似于图14-2中顺序add()的方式，添加该结点。为了保持跳表特性，除非指向一个结点的每个引用在所有层都已被设置，否则不能认为该结点逻辑上存在于集合中。每个结点都有一个额外的标记fullyLinked（完全链接），一旦结点被链入它的所有层，就将该标记设置为*true*，除非结点是完全链接的，我们才能访问它。因此，比如，当add()试图确定它要添加的结点是否已在链表中时，它必须自旋等待，直到该结点变成fullyLinked为止。图14-3表示一次add(18)调用，它一直等待，直到键值为18的结点变成完全链接的。

若要从链表中删除一个结点，remove()首先使用find()来检查具有目标键值的牺牲结点是否已在链表中，如果是，则检查该牺牲结点是否已准备好被删除，也就是说，应是完全链接且未标记的。在图14-3a中，remove(8)发现键值为8的结点没有标记且是完全链接的，说明可以删除它。remove(18)调用是失败的，其原因在于它发现牺牲结点不是完全链接的。同样 332
的remove(18)调用在图14-3b中却成功了，因为它发现牺牲结点是完全链接的。

如果牺牲结点可以被删除，remove()则通过设置它的标志位来逻辑地删除。对牺牲结点的物理删除按照下面步骤完成：先锁定牺牲者在所有层的前驱，然后锁定牺牲者自身，再确认其前驱未标记且仍指向牺牲者，最后，一次一层地剪接牺牲结点。为了保持跳表特性，应从顶到底地剪接牺牲结点。

例如，在图14-3b中，remove(8)锁定键值为5的前驱结点。一旦该前驱被锁定，remove()就将键值为5的结点在最底层的引用改为指向键值为9的结点，从而从链表中物理地删除该结点。

在add()和remove()方法中，如果确认失败，则再次调用find()以发现最近被改变的前驱集合，并试图再次完成该方法。

无等待的contains()方法调用find()来确定包含目标键值的结点。如果发现一个结点，则检查该结点是否未标记且完全链接，以确定该结点是否在集合中。这种方法和LazyList类的contains()方法一样，是无等待的，因为它忽略了SkipList结构中的所有锁或并发改变。

概括来说，LazySkipList类采用了一种与早期算法相近的技术：持有所有将被修改单元的锁，确认没有重大改变发生，然后完成修改，再释放锁（本章中，fullyLinked标志相当于锁）。

14.3.2 算法

列14-4描述了LazySkipList的Node类。当且仅当链表中包含一个未标记的、完全链接的、具有某个键值的结点时，该键值才在集合中。列14-3a中的键值8就是一个这样的例子。

```
1  public final class LazySkipList<T> {
2    static final int MAX_LEVEL = ...;
3    final Node<T> head = new Node<T>(Integer.MIN_VALUE);
4    final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
5    public LazySkipList() {
6      for (int i = 0; i < head.next.length; i++) {
7        head.next[i] = tail;
8      }
9    }
10   ...
11   private static final class Node<T> {
12     final Lock lock = new ReentrantLock();
13     final T item;
14     final int key;
15     final Node<T>[] next;
16     volatile boolean marked = false;
17     volatile boolean fullyLinked = false;
18     private int topLevel;
19     public Node(int key) { // sentinel node constructor
20       this.item = null;
21       this.key = key;
22       next = new Node[MAX_LEVEL + 1];
23       topLevel = MAX_LEVEL;
24     }
25     public Node(T x, int height) {
26       item = x;
27       key = x.hashCode();
28       next = new Node[height + 1];
29       topLevel = height;
30     }
31     public void lock() {
32       lock.lock();
33     }
34     public void unlock() {
35       lock.unlock();
36     }
37   }
38 }
```

图14-4 LazySkipList类：构造函数、域和Node类

图14-5表示跳表的find()方法。（同样的方法在顺序和并发算法中也能使用。）如果没有找到元素，find()方法返回-1。该方法使用pred和curr引用在最高层从head开始遍历SkipList。可以动态地保持最高层，以反映SkipList的实际最高层。为简单起见，这里没有这样做。find()方法一层接一层地向下进行，在每个层，将curr设为pred结点的后继。如果找到一个具有匹配键值的结点，则记录这个层数（第48行）。否则，就将pred和curr作为该层中的前驱和后继，记录在preds[]和succs[]数组中（第51～52行），然后继续在下一层从当前的pred结点开始执行。图14-2a表示find()是如何遍历SkipList的，b表示如何利用find()的结果向SkipList中添加一个新元素。

```
39    int find(T x, Node<T>[] preds, Node<T>[] succs) {
40      int key = x.hashCode();
41      int lFound = -1;
42      Node<T> pred = head;
43      for (int level = MAX_LEVEL; level >= 0; level--) {
44        volatile Node<T> curr = pred.next[level];
45        while (key > curr.key) {
46          pred = curr; curr = pred.next[level];
47        }
48        if (lFound == -1 && key == curr.key) {
49          lFound = level;
50        }
51        preds[level] = pred;
52        succs[level] = curr;
53      }
54      return lFound;
55    }
```

图14-5 LazySkipList类：无等待的find()方法。这个算法与顺序SkipList相同，preds[]和succs[]数组中存放着关于给定键值从最高层到0层的前驱和后继

由于我们在head哨兵结点用pred来开始，并且总是在curr小于目标键值的情况下将窗口向前推进，所以pred一直都是目标键值的前驱，且永远不会指向具有该键值本身的结点。find()方法返回数组preds[]和succs[]，也返回具有匹配键值的结点所在的层。

图14-6中的add(k)方法使用find()（图14-5）来决定具有目标键值k的结点是否已在链表中（第42行）。如果发现一个具有该键值的未标记结点（第62～67行），add(k)便返回*false*，表明键值k已在集合中。然而，如果那个结点还不是完全链接的（由fullyLinked域指出），那么线程将等待直到它被链接为止（因为只有当结点是完全链接的，键值k才在抽象集中）。如果发现结点被标记，则说明其他的线程正在删除该结点，因此add()调用只是简单地重试。否则，它检查该结点是否是未标记且完全链接的，这表明add()应该返回*false*。因为remove()方法只标记完全链接的结点，所以将检查结点是否是未标记的放在检查结点是否是完全链接的之前进行，是一种安全的办法。如果一个结点是未标记的但还不是完全链接的，那么在该结点变为已标记的之前必须是未标记且完全链接的（图14-6）。第66行是一个不成功add()调用的可线性化点。

add()方法调用find()来初始化preds[]和succs[]数组，以存放要加入结点的假前驱和假后继。因为当结点被访问时这些引用有可能不再准确，所以它们是不真实的。如果没有找到具有键值k的未标记且完全链接的结点，那么线程将从新结点的0层直到topLevel锁定并验证每个由find()返回的前驱（第74～80行）。为了避免死锁，add()和remove()应以升序来获

333
334

取锁。在add()开始的最初时刻，用randomLevel()方法[⊖]来决定toplevel的值。在每个层的验证中（第79行），检查前驱是否仍然邻近后继且都没有被标记。如果验证失败，说明线程肯定受到contains方法的影响，因此，释放其所获得的锁（第87行的finally块），并重新尝试。

```
56  boolean add(T x) {
57    int topLevel = randomLevel();
58    Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
59    Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
60    while (true) {
61      int lFound = find(x, preds, succs);
62      if (lFound != -1) {
63        Node<T> nodeFound = succs[lFound];
64        if (!nodeFound.marked) {
65          while (!nodeFound.fullyLinked) {}
66          return false;
67        }
68        continue;
69      }
70      int highestLocked = -1;
71      try {
72        Node<T> pred, succ;
73        boolean valid = true;
74        for (int level = 0; valid && (level <= topLevel); level++) {
75          pred = preds[level];
76          succ = succs[level];
77          pred.lock.lock();
78          highestLocked = level;
79          valid = !pred.marked && !succ.marked && pred.next[level]==succ;
80        }
81        if (!valid) continue;
82        Node<T> newNode = new Node(x, topLevel);
83        for (int level = 0; level <= topLevel; level++)
84          newNode.next[level] = succs[level];
85        for (int level = 0; level <= topLevel; level++)
86          preds[level].next[level] = newNode;
87        newNode.fullyLinked = true; // successful add linearization point
88        return true;
89      } finally {
90        for (int level = 0; level <= highestLocked; level++)
91          preds[level].unlock();
92      }
93    }
94  }
```

图14-6 LazySkipList类：add()方法

如果线程成功地锁定并验证了直到新结点的topLevel的find()返回值，那么add()调用成功，因为线程持有了它所需的所有锁。然后，线程分配一个具有适当highest的新结点，随机地选择topLevel，将该结点链入并设置新结点的fullyLinked标志。对标志的设置则是一次成功的add()调用的可线性化点（第87行）。之后，线程释放所有的锁并返回*true*（第89行）。线程能够修改未上锁结点的next域的唯一时刻就是在它初始化新结点的next引用时（第83行）。因为初始化发生在新结点可以访问之前，所以它是安全的。

335
|
336

⊖ 基于经验评测设计的randomLevel()方法用来维持跳表特性。例如，在Java并发程序包中，对于最大级别数为31的跳表，概率是$\frac{3}{4}$时randomLevel()方法返回0，对于i∈[1,30]的概率是$2^{-(i+1)}$，且31的概率是2^{-31}。

remove()方法如图14-7所示。它调用find()来确定具有适当键值的结点是否在链表中。如果是，那么线程检查这个结点是否已准备好要被删除（第104行），即结点是完全链接的、未被标识的且在其最高层中，最高层之下的结点要么还未完全链接（图14-3a中键值为18的结点），要么已被标记并已被一个并发remove()方法部分地断开了链接。（这个remove()方法可以继续执行，但在接下来的验证中会失败。）

```
95    boolean remove(T x) {
96      Node<T> victim = null; boolean isMarked = false; int topLevel = -1;
97      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
98      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
99      while (true) {
100       int lFound = find(x, preds, succs);
101       if (lFound != -1) victim = succs[lFound];
102       if (isMarked ||
103            (lFound != -1 &&
104            (victim.fullyLinked
105            && victim.topLevel == lFound
106            && !victim.marked))) {
107         if (!isMarked) {
108           topLevel = victim.topLevel;
109           victim.lock.lock();
110           if (victim.marked) {
111             victim.lock.unlock();
112             return false;
113           }
114           victim.marked = true;
115           isMarked = true;
116         }
117         int highestLocked = -1;
118         try {
119           Node<T> pred, succ; boolean valid = true;
120           for (int level = 0; valid && (level <= topLevel); level++) {
121             pred = preds[level];
122             pred.lock.lock();
123             highestLocked = level;
124             valid = !pred.marked && pred.next[level]==victim;
125           }
126           if (!valid) continue;
127           for (int level = topLevel; level >= 0; level--) {
128             preds[level].next[level] = victim.next[level];
129           }
130           victim.lock.unlock();
131           return true;
132         } finally {
133           for (int i = 0; i <= highestLocked; i++) {
134             preds[i].unlock();
135           }
136         }
137       } else return false;
138     }
139   }
```

图14-7 LazySkipList类：remove()方法

如果结点已做好被删除的准备，线程则对其上锁（第109行）并验证它是否仍未标记。如果还没有被标记，则标记该结点，逻辑地删除这个元素。第114行的操作步是一次成功的

remove()调用的可线性化点。如果结点已被标记，那么线程返回*false*，因为这个结点已经被删除了。该操作步是一次不成功的remove()调用的可线性化点。当find()没有找到一个具有匹配键值的结点，或者找到的匹配结点已经被标记，或者不是完全链接的，或者在它的最高层中也没有找到时（第104行），则会出现另一种情形。

该方法余下的部分是对victim结点进行物理删除。为了从链表中删除牺牲者，remove()方法首先在直到牺牲者toplevel的所有层中，锁定（用升序，以防止死锁）牺牲者的前驱结点（第120--124行）。每锁定一个前驱，则验证该前驱仍未标识且仍指向牺牲者，然后，一次一层地将牺牲者剪接掉（第128行）。为了保持跳表特性，即在一个给定层可达的结点在较低层也是可达的，应该从顶向下地剪接牺牲者。如果任一层的验证失败，线程将释放所有前驱的锁（不包括牺牲者），然后调用find()获取一组新的前驱结点。因为牺牲者的isMarked域已被设置，所以线程不再尝试标记这个结点。在成功删除牺牲者结点以后，线程将释放它的所有锁并返回*true*。

最后，如果没有找到任何结点，或者找到的结点已经被标记，或者不是完全链接的，或者在其最高层没有找到，那么只是简单地返回*false*。显然，如果结点没有被标记，返回*false*是正确的，因为对任意的键值，在任何时候SkipList中最多只能有一个具有该键值的结点（也就是从head是可达的）。况且，一旦一个结点被链入链表（必定是在被find()方法找到之前链入的），那么在被标记之前就不能删除该结点。由此可知，如果结点未标记且不是完全链接的，则必定处于正在加入SkipList的过程中，但这个正在添加的方法还没有到达可线性化点（图14-3a中键值为18的结点）。

如果发现该结点时已被标记，那么它有可能不在链表中，而可能是其他某个具有相同键值的未标记结点。然而，在这种情况下，正如LazyList的remove()方法一样，在remove()调用过程中必然存在着键值不在抽象集中的时间点。

无等待的contains()方法（图14-8）调用find()来确定具有目标键值的结点。如果找到一个结点，则检查该结点是否未标记且完全链接。和第9章LazyList类的contains()方法一样，这个方法也是无等待的，它不考虑SkipList结构中的任何锁和并发改变。一次成功的contains()调用的可线性化点是遍历前驱结点的next引用时，发现它未标记且完全链接的时刻，和remove()调用一样，如果contains()方法找到一个已标记的结点，则是一次不成功的调用。但是必须谨慎行事，因为当该结点被找到时，它并不一定在链表中，而可能是某个具有相同键值的未标记结点。然而在contains()方法调用期间，必定存在一个键值不在抽象集中的时刻。

357
~
358

```
140   boolean contains(T x) {
141     Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
142     Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
143     int lFound = find(x, preds, succs);
144     return (lFound != -1
145         && succs[lFound].fullyLinked
146         && !succs[lFound].marked);
147   }
```

图14-8 LazySkipList类：无等待的contains()方法

14.4 无锁并发跳表

LockFreeSkipList实现的基础是第9章的LockFreeList算法：SkipList结构的每一层是一个LockFreeList，每个结点的next引用是一个AtomicMarkableReference<Node>，对链表的操作通过compareAndSet()完成。

14.4.1 简介

下面是LockFreeSkipList类的概述。

因为无法在所有层同时使用锁对引用进行操作，所以LockFreeSkipList不能保持跳表特性——每个链表都是较低层链表的子链表。

既然不能保持跳表特性，我们采用由最底层链表定义抽象集的方法：如果一个结点的next引用在最底层链表未被标记，那么这个结点的键值在抽象集中，在跳表中，高层链表中的结点仅仅是最底层结点的快捷符号。因此，没有必要像LazySkipList那样采用一个fullyLinked标志。

如何添加或者删除一个结点呢？将链表的每一层看成是一个LockFreeList。对一个给定的层，使用compareAndSet()方法插入一个结点，通过标记结点的next引用删除这个结点。

和LockFreeList一样，find()方法清除被标记的结点。它遍历跳表，向下访问每一层的每个链表。和LockFreeList类中的find()方法一样，当遇到被标记的结点时，不断地清除这些结点，决不查看被标记结点的键值。然而，这也意味着一个正在被链接到更高层的结点可能会被物理地删除。穿过结点中同层引用的find()调用可能删除这些引用，和前面一样，跳表特性不再成立。 339

add()方法调用find()来确定一个结点是否已在链表中，并找到该结点的前驱和后继集。一个新结点与一个随机选取的topLevel一起准备，且它的next引用指向由find()返回的那些可能的后继。下一步采用与LockFreeList相同的方法，通过将新结点链接到最底层链表，从而在逻辑上将这个新结点加入到抽象集中。如果添加成功，那么这个元素逻辑上存在于集合中。然后，add()调用将这个结点链接到更高的层次（直到它的最高层）。

图14-9表示LockFreeSkipList类。在a中，add(12)调用find(12)，此时，有3个remove()调用正在进行中。b表示重新指向虚线链接后的结果。c表示随后对键值为12的新结点进行添加的过程。d则表示如果键值为11的结点在键值为12的结点被添加前就已删除，那么将会发生的另一种添加情形。

remove()方法调用find()来确定具有目标键值的未标记结点是否在最底层链表中。如果找到一个未标记结点，则从topLevel开始标记该结点。除了最底层的next引用，所有的next引用都通过做标记来从其所在层的链表中逻辑地删除。在除最底层之外的所有层都被标记之后，再来标记最底层的next引用。如果这次标记成功，则元素从抽象集中删除。结点的物理删除是通过remove()方法本身和遍历跳表时访问它的其他线程的find()方法，在所有层的链表中都物理地删除该结点来完成的。在add()和remove()方法中，因为在compareAndSet()失败时，前驱集和后继集有可能已被改变，所以必须再次调用find()方法。

add()、remove()和find()方法之间交互的关键在于链表操作的发生次序。add()方法在将结点链接到最底层链表之前，将它的next引用设置为它的后继，这意味着一个结点在被逻

辑地加入到链表中的时刻，就已做好被删除的准备了。同样，remove()方法从上到下标记next引用，这样，一旦结点被逻辑地删除，就不会被find()调用遍历到了。

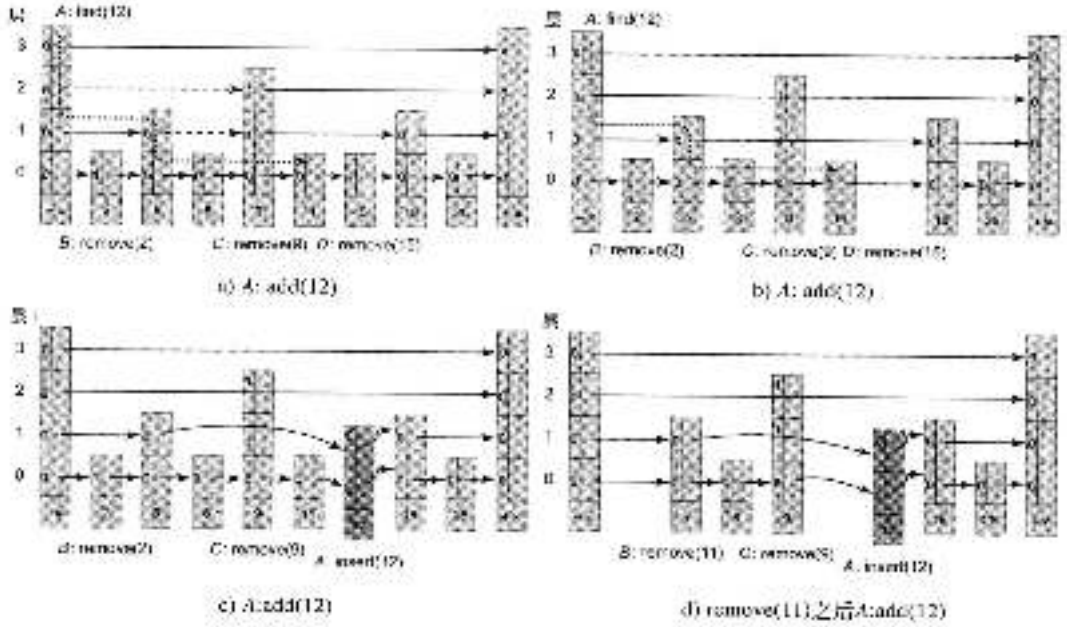


图14-9 LockFreeSkipList类：一次add()调用。标个结点由未标识的(a, 0)或已标识的(a, 1)链接组成。在a中，add(12)调用find(12)，此时，有3个正在运行中的remove()调用。find()方法在遍历跳表的过程中"清除"已标记的链接（用1表示）。这个遍历过程与顺序的find(12)不同，因为一旦遇到被标记的结点，要把它们的链接断开。图中标出的路径显示了由pred引用所遍历的结点，这个引用总是指向未标记且键值小于目标键值的结点。b显示了重新调整指向虚线链接后的结果，我们通过把链接放在结点前面来绕过该结点。结点15在最底层的next引用是被标识的，要把它从跳表中删除。c显示了随后对键值为12的新结点的添加过程。d显示了另一种可能的添加场景——键值为11的结点在添加键值为12的结点之前被删除。因为键值为9的结点在最底层的next引用还未标记，所以最底层的前驱结点的next引用为已标记的，要通过add()方法重新指向新结点。一旦线程C完成了对这个引用的标记，键值为9的结点就要删除，且键值为5的结点变为新添加结点的直接前驱

如我们所知，在大多数应用中，对contains()的调用要比对其他方法的调用次数多。所以，contains()不应该调用find()方法。虽然让单独的find()方法去物理地删除那些被逻辑删除的结点是一种行之有效的办法，但当太多的find()试图同时清除同一个结点时，则会导致争用。这种争用在频繁的contains()调用中要比在其他方法调用中更容易出现。

但是，contains()方法不能使用LockFreeList类中的contains()方法所采用的做法是：查看键值并简单地忽略已标记结点。原因在于add()和remove()方法有可能破坏跳表特性。一个被标记的结点从最底层链表中物理删除后，在高层上仍有可能是可达的。忽略标记则有可能导致跳过在较低层可达的结点。

但要注意，LockFreeSkipList的find()方法不受这个问题的影响，因为它从来不管已标

记结点的键值，而是直接删除它们。我们让contains()方法来模拟这种行为，但却不删除已标记结点。相反，contains()方法遍历跳表，忽略已标记结点的键值，跳过这些结点而不物理删除。由于不需要物理删除，所以这种方法是无等待的。

14.4.2 算法细节

当我们展现算法细节的时候，读者应该牢记抽象集只是通过最底层链表来定义的，高层链表中的结点只是进入最底层结点的一种快速方式。图14-10表示链表结点的结构。

```
1  public final class LockFreeSkipList<T> {
2    static final int MAX_LEVEL = ...;
3    final Node<T> head = new Node<T>(Integer.MIN_VALUE);
4    final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
5    public LockFreeSkipList() {
6      for (int i = 0; i < head.next.length; i++) {
7        head.next[i]
8        = new AtomicMarkableReference<LockFreeSkipList.Node<T>>(tail, false);
9      }
10   }
11   public static final class Node<T> {
12     final T value; final int key;
13     final AtomicMarkableReference<Node<T>>[] next;
14     private int topLevel;
15     // constructor for sentinel nodes
16     public Node(int key) {
17       value = null; key = key;
18       next = (AtomicMarkableReference<Node<T>>[])
19        new AtomicMarkableReference[MAX_LEVEL + 1];
20       for (int i = 0; i < next.length; i++) {
21         next[i] = new AtomicMarkableReference<Node<T>>(null,false);
22       }
23       topLevel = MAX_LEVEL;
24     }
25     // constructor for ordinary nodes
26     public Node(T x, int height) {
27       value = x;
28       key = x.hashCode();
29       next = (AtomicMarkableReference<Node<T>>[])
          new AtomicMarkableReference[height + 1];
30       for (int i = 0; i < next.length; i++) {
31         next[i] = new AtomicMarkableReference<Node<T>>(null,false);
32       }
33       topLevel = height;
34     }
35   }
```

图14-10 LockFreeSkipList类：域和构造函数

图14-11中的add()方法使用图14-13中的find()方法来确定一个键值为k的结点是否在链表中（第61行），和LazySkipList一样，add()调用find()来初始化数组preds[]和succs[]，以存放新结点的假前驱和假后继。

如果在最底层链表中找到一个具有目标键值的未标记结点，那么find()返回*true*，add()返回*false*，表示这个键值已经在抽象集中，不成功add()的可线性化点和成功find()的可线性化点相同（第42行）。如果没有找到结点，那么下一步就是尝试向结构中添加一个键值为k的结点。

```
36    boolean add(T x) {
37      int topLevel = randomLevel();
38      int bottomLevel = 0;
39      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
40      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
41      while (true) {
42        boolean found = find(x, preds, succs);
43        if (found) {
44          return false;
45        } else {
46          Node<T> newNode = new Node(x, topLevel);
47          for (int level = bottomLevel; level <= topLevel; level++) {
48            Node<T> succ = succs[level];
49            newNode.next[level].set(succ, false);
50          }
51          Node<T> pred = preds[bottomLevel];
52          Node<T> succ = succs[bottomLevel];
53          if (!pred.next[bottomLevel].compareAndSet(succ, newNode,
54                                                    false, false)) {
55            continue;
56          }
57          for (int level = bottomLevel+1; level <= topLevel; level++) {
58            while (true) {
59              pred = preds[level];
60              succ = succs[level];
61              if (pred.next[level].compareAndSet(succ, newNode, false, false))
62                break;
63              find(x, preds, succs);
64            }
65          }
66          return true;
67        }
68      }
69    }
```

图14-11 LockFreeSkipList类：add()方法

一个新结点以一个随机选择的toplevel来创建。结点的next引用是未标记的且设置为指向由find()返回的后继（第46~49行）。

下一步就是尝试添加新结点，将该结点链接到最底层链表中由find()返回的preds[0]和succs[0]之间。和LockFreeList一样，使用compareAndSet()设置引用，确认这些结点之间仍然相互关联，且没有从链表中删除（第54行）。如果compareAndSet()失败，则说明发生了改变，需要重新调用该方法；如果成功，那么该元素被添加，第54行是这个调用的可线性化点。

[343] 然后，add()方法在更高的层中链接这个结点（第57行）。对于每一层，如果前驱指向有效的后继（第61行），则通过设置前驱指向新的结点而将该结点拼接进来。如果成功，则退出并进入下一层；如果不成功，则说明前驱指向的结点已经被改变，因此重新调用find()以找到一个新的有效的前驱和后继集合。我们不使用find()调用的结果（第63行），因为我们仅仅关心在剩下的未链接层中重新计算假前驱和假后继。一旦所有的层都被链接，该方法返回*true*（第66行）。

图14-12中的remove()调用find()来确认一个具有匹配键值的未标记结点是否在最底层的链表中。如果在最底层链表中没有结点，或存在匹配结点但它已被标记，该方法则返回*false*。不成功的remove()方法的可线性化点是第76行find()方法被调用的时刻。如果一个未标记结

点被找到，那么这个方法从抽象集中将相关键值逻辑地删除，并为物理删除做准备。这一步使用了前驱集（由find()存放在preds[]中）和victim（在succs[]中通过find()返回）。首先，从toplevel开始，通过不断读取next及其标记，并调用attempMark()，对直到最底层的所有链接（但不包括最底层的链接）做上标记（第82-88行）。如只发现链接是标记的（或者它已被标记，或者是本次尝试成功），将转向下一层进行处理。否则，重新读取当前层上的链接，因为它已被其他并行的线程修改，所以需要重新进行这次标记尝试。一旦除了最底层以外的所有层都已被标记，则对最底层的next引用进行标记。如果这个标记操作（第95行）成功，则是一次成功的remove()的可线性化点。remove()方法尝试使用compareAndSet()来标记next域。如果成功，则可以确定是该线程将标记从*false*改为*true*。在返回*true*之前，find()方法被再次调用。这个调用是一种优化行为。作为一种副作用，如果一个结点已经被逻辑删除，那么find()将物理删除所有指向该结点的链接。

```
70    boolean remove(T x) {
71      int bottomLevel = 0;
72      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
73      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
74      Node<T> succ;
75      while (true) {
76        boolean found = find(x, preds, succs);
77        if (!found) {
78          return false;
79        } else {
80          Node<T> nodeToRemove = succs[bottomLevel];
81          for (int level = nodeToRemove.topLevel;
82               level >= bottomLevel+1; level--) {
83            boolean[] marked = {false};
84            succ = nodeToRemove.next[level].get(marked);
85            while (!marked[0]) {
86              nodeToRemove.next[level].compareAndSet(succ, succ, false, true);
87              succ = nodeToRemove.next[level].get(marked);
88            }
89          }
90          boolean[] marked = {false};
91          succ = nodeToRemove.next[bottomLevel].get(marked);
92          while (true) {
93            boolean iMarkedIt =
94              nodeToRemove.next[bottomLevel].compareAndSet(succ, succ,
95                                                           false, true);
96            succ = succs[bottomLevel].next[bottomLevel].get(marked);
97            if (iMarkedIt) {
98              find(x, preds, succs);
99              return true;
100           }
101           else if (marked[0]) return false;
102         }
103       }
104     }
105   }
```

图14-12 LockFreeSkipList类：remove()方法

另一方面，如果compareAndSet()调用失败，则next引用已被标记，那么就意味着另一个并发线程删除了该结点，所以remove()返回*false*。这个不成功的remove()调用的可线性化点就是由成功标记next引用的那个线程所调用的remove()的可线性化点。注意，这个可线性

化点必须出现在remove()调用的过程中，因为该find()调用在发现结点被标记之前，首先发现它未被标记。

最后，如果compareAndSet()失败且结点未被标记，那么next引用必定已被并发地改变了。既然victim是已知的，就没有必要再次调用find()，remove()则只是简单地使用从next中读取的新值来重新尝试这次标记。

如上所述，add()方法和remove()方法都依赖于find()方法，该方法搜索LockFreeSkipList，当且仅当具有目标键值的结点在集合中返回*true*。在每一层，则用目标结点的假前驱集和假后继集来填入数组preds[]和succs[]。该方法具有以下两个属性：

- 它从不遍历一个已标记的链接，相反，它从该层的链表中删除由一个标记的链接所指向的结点。
- 每个preds[]引用都指向一个键值小于目标键值的结点。

图14-13中的find()方法按照如下方式进行：它从head哨兵（具有允许的最大结点层）的toplevel开始遍历SkipList，然后，从上到下在每一层中进行操作，填入不断增多的preds结点和succs结点，直到pred指向该层中具有最大值的结点，而这个值是小于目标键值的（第117~131行）。和LockFreeList中一样，当它使用compareAndSet()遇到被标记的结点时，不断地在给定的层中删除这些结点（第119~125行）。注意，compareAndSet()确认前驱的next

```
106   boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
107     int bottomLevel = 0;
108     int key = x.hashCode();
109     boolean[] marked = {false};
110     boolean snip;
111     Node<T> pred = null, curr = null, succ = null;
112     retry:
113       while (true) {
114         pred = head;
115         for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
116           curr = pred.next[level].getReference();
117           while (true) {
118             succ = curr.next[level].get(marked);
119             while (marked[0]) {
120               snip = pred.next[level].compareAndSet(curr, succ,
121                                                     false, false);
122               if (!snip) continue retry;
123               curr = pred.next[level].getReference();
124               succ = curr.next[level].get(marked);
125             }
126             if (curr.key < key) {
127               pred = curr; curr = succ;
128             } else {
129               break;
130             }
131           }
132           preds[level] = pred;
133           succs[level] = curr;
134         }
135         return (curr.key == key);
136       }
137   }
```

图14-13 LockFreeSkipList类：比LazySkipList中更复杂的find()方法

域指向当前结点。一旦找到一个未标记的curr（第126行），就要检查其键值是否小于目标键值。如果是，则pred先于curr，否则，curr的键值就大于或等于目标键值，所以当前pred值就是目标结点的直接前驱。find()方法跳出当前层的查找循环，保存当前的pred值和curr值（第132行）。

find()方法在到达最底层时才会停止这些操作。有一点很重要：每层的遍历都具有前面提到的两种特性，特别是，如果一个具有目标键值的结点在链表中，即使它在高层链表中已经被删除，也能在最底层的链表中查找到。当遍历停止时，pred指向目标结点的前驱。该方法由上至下遍历每一层而不会跳过目标结点。如果该结点在链表中，那么会在最底层的链表中找到。另外，如果这个结点被找到，那么它不能被标记，因为它是已标记的，可能会在第119--125行中被去除。因此，第135行的测试只需检查curr的键值是否等于目标键值，以便确定目标键值是否在集合中。

344
|
346

成功或不成功的find()调用的可线性化点都出现在最底层链表的curr引用被设置的时刻，也就是在第116行或第123行，这取决于在第136行决定find()调用成功与否之前的最后时刻。图14-9显示了一个结点是如何被成功地添加到LockFreeSkipList中去的。

图14-14是无等待的contains()方法，它采用了与find()方法一样的方式遍历SkipList，从head开始逐层下降。和find()方法一样，contains()忽略已标记结点的键值。与find()不同之处是，它并不尝试删除已标记的结点，而只是简单地跳过它们（第147~150行）。图14-15中给出了一次执行实例。

```
138   boolean contains(T x) {
139     int bottomLevel = 0;
140     int v = x.hashCode();
141     boolean[] marked = {false};
142     Node<T> pred = head, curr = null, succ = null;
143     for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
144       curr = curr.next[level].getReference();
145       while (true) {
146         succ = curr.next[level].get(marked);
147         while (marked[0]) {
148           curr = pred.next[level].getReference();
149           succ = curr.next[level].get(marked);
150         }
151         if (curr.key < v){
152           pred = curr;
153           curr = succ;
154         } else {
155           break;
156         }
157       }
158     }
159     return (curr.key == v);
160   }
```

图14-14 LockFreeSkipList类：无等待的contains()方法

这个方法是正确的，因为contains()保持了和find()一样的特性，在它们之中，任何层的pred决不会指向一个未标记的、键值大于或等于目标键值的结点，pred变量总是在最底层链表上到达目标结点前面的一个结点。如果该结点在contains()调用开始之前就添加到链表中，那么它将被找到。另外，回顾add()调用find()的情形，在添加新结点之前，它将已标记

结点从最底层链表中断开，由此可知，如果contains()没有找到指定的结点，或者在最底层找到该结点但它已被标记，那么所有并发添加的、未被发现的结点必定在contains()调用开始之后添加到最底层，所以在第159行返回*false*是正确的。

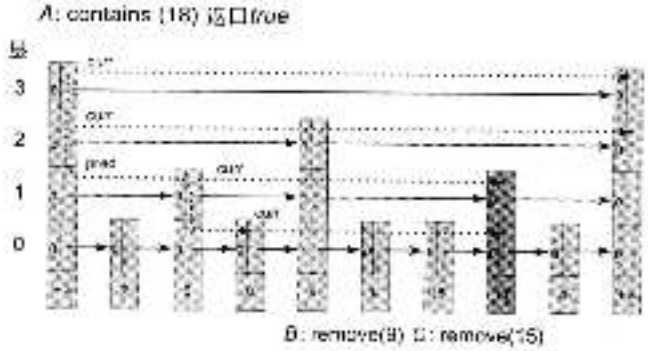


图14-15 线程A调用contains(18)，它从head结点的顶层开始遍历链表。虚点线标记了通过pred域进行的遍历，细点线标记了curr域的路径。curr域在第3层上被推进到tail。由于它的键值比18大，pred下降到第2层。curr域推进，在键值为9的结点中经过被标记的引用，再次到达tail（它大于18），所以pred下降到第1层。在这里，pred被推进到键值为5的未标记结点上，curr经过键值为9的已标记结点，到达键值为18的未标记结点。在这个点上，curr不再继续推进了。虽然18是目标键值，该方法仍然继续将pred向到最底层，将pred推进到键值为8的结点上。从这个点开始，curr遍历经过了已标记结点9、15和11，它们的键值都小于18，最终，curr到达键值为18的未标记结点，返回*true*

图14-16描述了contains()方法的一次执行过程。在a中，contains(18)调用从head结点的顶层开始遍历链表。在b中，contains(18)调用在键值为18的结点被逻辑删除后遍历链表。

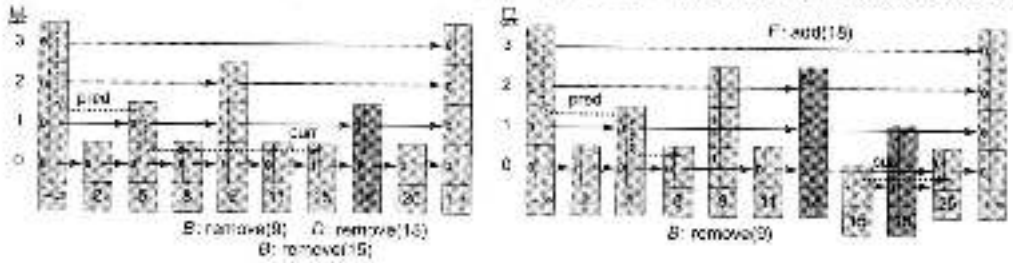


图14-16 LockFreeSkipList类：一次contains()调用。在a中，contains(18)从head的顶层开始遍历链表。点线标记了通过pred域进行的遍历。pred域最终在最底层到达了结点8。从这个点开始，我们也用点线描述了curr的路径。curr遍历经过结点9到达已标记结点15。在b中，一个键值为18的新结点被线程F添加到链表中。线程B，作为其find(18)调用的一部分，物理地删除键值为9、15和18的老结点。现在，线程A从被删除的键值为15的结点开始，继续以curr域进行遍历（由于结点15和18对于线程A是可达的，所以它们不是正确循环的）。线程A到达键值为25的结点（比18大），返回*false*。即使在这个点上，在LockFreeSkipList中存在一个键值为18的未标记结点，该结点也是被B在A遍历的同时插入的，并且在A的add(18)之后是可线性化的

14.5 并发跳表

我们已经介绍了两种高度并发的SkipList实现，每一种都支持无数级的查找，而无需再次平衡。在LazySkipList类中，add()和remove()方法采用优化的细粒度上锁方式，这意味着该方法无需上锁就能查找其目标结点，仅当发现目标结点时才获取锁并验证。最常用的contains()方法是无等待的。在LockFreeSkipList类中，add()和remove()方法是无锁的，并建立在第9章的LockFreeList类基础之上。contains()方法也在这个类中，是无等待的。

在第15章中，将会看到如何在本章中描述的并发SkipList基础之上，构建高度并发的优先级队列。

14.6 本章注释

Bill Pugh发明了顺序[128]和并发[127]的跳表。LazySkipList是由Yossi Lev、Maurice Herlihy、Victor Luchangco和Nir Shavit[103]提出的。本章的LockFreeSkipList是由Maurice Herlihy、Yossi Lev和Nir Shavit[64]发明的。它部分基于早先由Kier Fraser[42]发明的无锁SkipList算法，其一种变化方式被Doug Lea[100]封装到Java并发包中。

347
348

14.7 习题

习题163. 回顾跳表是一个概率数据结构，尽管contains()调用的期望性能为$O(\log n)$，其中n是链表中元素的个数，但最坏情形的性能可能为$O(n)$。试给出具有8个元素的跳表在最坏情形下的图示，并说明为什么是这样的。

习题164. 已知一个跳表的概率为p，MAX_LEVEL为M，如果该链表包含N个结点，那么从0到$M-1$的每个层上，期望的结点数分别是多少？

习题165. 修改LazySkipList类，使得在该结构中，find()从最高的结点开始，而不是可能的最高层（MAX_LEVEL）。

习题166. 修改LazySkipList以支持多个元素具有相同的键值。

349

习题167. 假设我们修改了LockFreeSkipList类，使得图14-12的第101行中，remove()不是返回*false*，而是重新开始主循环。

该算法还正确吗？阐述安全性和活性问题。也就是说，不成功的remove()调用的新的可线性化点是什么？这个类还是无锁的吗？

习题168. 试说明在LockFreeSkipList类中，一个结点将怎样在链表中层0和2上结束，但不会在层1上结束。画出示意图。

习题169. 修改LockFreeSkipList类，使得find()方法使用单个compareAndSet()来删除已标记结点的序列，试说明你的实现为何不能删除一个并发插入的未标记结点。

习题170. 如果最底层已被链接，然后将其他所有层以任意次序链接，那么LockFreeSkipList的add()方法还能正常工作吗？如果最底层的next引用最后标记，但其他所有层的引用都以任意次序标记，那么remove()方法中对next引用的标记是否还是正确的？

习题171.（难度较大）试修改LazySkipList，使得每个层上的链表都是双向的，并允许线程从head或者tail遍历都能并行地添加和删除元素。

习题172. 图14-17描述了LockFreeSkipList类的一个错误的contains()方法，试给出该方法返回

错误值的一种场景。提示：该方法出错的原因是它考虑了已删除结点的键值。

```
1   boolean contains(T x) {
2     int bottomLevel = 0;
3     int key = x.hashCode();
4     Node<T> pred = head;
5     Node<T> curr = null;
6     for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
7       curr = pred.next[level].getReference();
8       while (curr.key < key ) {
9         pred = curr;
10        curr = pred.next[level].getReference();
11      }
12    }
13    return curr.key == key;
14  }
```

350

图14-17 LockFree SkipList类：一个错误的contains()

优先级队列

15.1 引言

优先级队列是元素的多重集，其中，每个元素都有一个相关的优先级，它是表示该元素重要性的一个数值（按惯例，越小的数字说明越重要，表示更高的优先级）。优先级队列通常提供向集合中加入元素的add()方法，删除并返回最小值（即最高优先级）元素的removeMin()方法。从高层应用程序直到底层操作系统内核，都要使用优先级队列。

有界范围的优先级队列是一种每个元素的优先数都取自一个元素离散集的优先级队列，而在无界范围的优先级队列中，优先数则来自一个很大的集合，比如说32位整数或者浮点值。毫无疑问，有界范围的优先级队列往往更加有效，但很多应用却要求无界范围的优先级队列。图15-1给出了优先级队列的接口。

```
public interface PQueue<T> {
  void add(T item, int score);
  T removeMin();
}
```

图15-1 优先级队列的接口

并发优先级队列

在集合的并发操作中，add()方法和removeMin()方法的调用可以相互重叠，那么，一个元素在集合中到底意味着什么？

这里，我们考虑两种在第3章已介绍的一致性条件：第一种是可线性化性，它要求每个方法调用都看似是在它的调用和响应之间的某个瞬间生效的；第二种是静态一致性，这是一个相对较弱的条件，要求在每次执行过程中，在任一时刻，如果没有额外的方法调用，那么当所有待处理的方法调用完成之后，它们返回的值要与该对象的某次正确的顺序执行相一致。如果应用不要求它的优先级队列是可线性化的，那么让它们具有静态一致性往往会更加有效。对于特定的应用需要认真考虑，以选择正确的途径。

15.2 基于数组的有界优先级队列

如果一个有界范围优先级队列的优先数取自0，…，$m-1$，那么它的范围是m。现在，我们考虑采用两个成员数据结构的有界优先级队列算法：Counter和Bin。Counter（见第12章）只有一个整数值，提供getAndIncrement()和getAndDecrement()方法原子地增加和减少计数器的值，并返回该计数器的先前值。这些方法可以随意地限界，也就是说，它们不能使计数器的值超出某个特定界限。

Bin是一个具有任意元素的池，提供put(*x*)方法插入一个元素*x*，用get()方法删除并返回一个元素，若该池为空，则返回*null*。可以使用锁或以无锁方式利用第11章的栈算法来实现池。

图15-2为SimpleLinear类，它维护着一个由池组成的数组。若要增加一个优先数为*i*的元素，线程只需简单地将这个元素放入第*i*个池中，removeMin()方法按照优先级递减的顺序扫描这些池，并返回第一个成功删除的元素。如果没有找到元素，则返回*null*。如果这些池是可线性化的，那么SimpleLinear也是可线性化的。如果这些Bin的方法是无锁的，那么add()和removeMin()方法也是无锁的。

```
1  public class SimpleLinear<T> implements PQueue<T> {
2    int range;
3    Bin<T>[] pqueue;
4    public SimpleLinear(int myRange) {
5      range = myRange;
6      pqueue = (Bin<T>[])new Bin[range];
7      for (int i = 0; i < pqueue.length; i++){
8        pqueue[i] = new Bin();
9      }
10   }
11   public void add(T item, int key) {
12     pqueue[key].put(item);
13   }
14   public T removeMin() {
15     for (int i = 0; i < range; i++) {
16       T item = pqueue[i].get();
17       if (item != null) {
18         return item;
19       }
20     }
21     return null;
22   }
23 }
```

图15-2 SimpleLinear类：add()和removeMin()方法

15.3 基于树的有界优先级队列

SimpleTree（图15-3）是一种静态一致的无锁有界范围优先级队列。它是一个二叉树

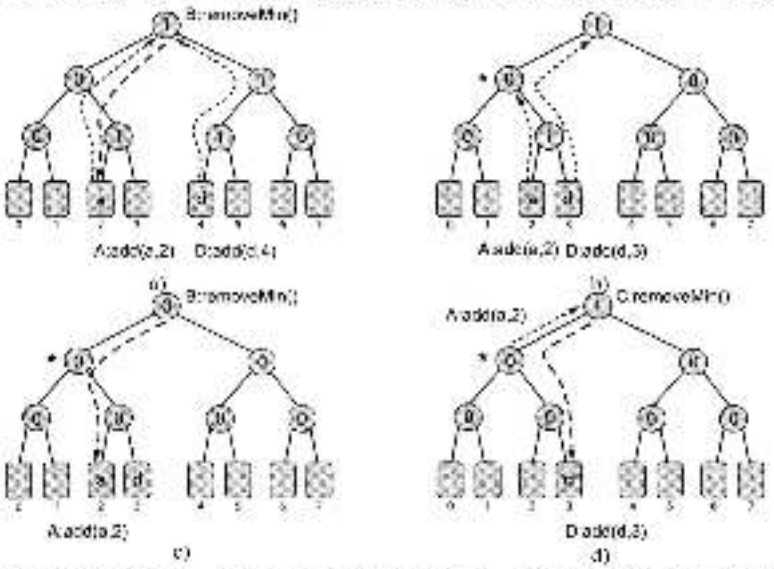


图15-3 SimpleTree优先级队列是一个由有界计数器组成的树。所有元素都在叶结点的池中，内部结点都有该结点左子树中所包含元素的个数。在a中，线程*A*和*D*通过向上遍历树来增加元素，当它们从左向上遍历时，增加结点中的计数器值。线程*B*根据计数器向下遍历树，如果计数器是非零值，则从左向下移动（没有给出*B*的递减效果）。在b、c和d中，描述了并发线程*A*和*B*在标有"*"的结点上相遇的执行序列。在b中，线程*D*添加*d*，然后*A*添加*a*并向上到达带有星号的结点处，沿着路径增加了一个计数器，在c中，*B*向下遍历树，将计数器减为0，并弹出*a*。在d中，*A*继续上升，即使*B*已经从带"*"的结点向下删除了*a*的所有痕迹。*A*也增加根结点处的计数器，然而，一切都很正常，因为根结点的非0计数器能正确地将*C*引导至具有最高优先级的元素*d*处

(图15-4)，由treeNode对象（图15-5）组成。如图15-3所示，该树只有m个叶结点，其中第i个叶结点有一个包含优先数为i的元素的池。在树的内部结点中，有$m-1$个布尔共享计数器，用于记录以每个结点的左（低优先数/高优先级）孩子为根的子树中所包含的元素的个数。

```
1  public class SimpleTree<T> implements PQueue<T> {
2    int range;
3    List<TreeNode> leaves;
4    TreeNode root;
5    public SimpleTree(int logRange) {
6      range = (1 << logRange);
7      leaves = new ArrayList<TreeNode>(range);
8      root = buildTree(logRange, 0);
9    }
10   public void add(T item, int score) {
11     TreeNode node = leaves.get(score);
12     node.bin.put(item);
13     while(node != root) {
14       TreeNode parent = node.parent;
15       if (node == parent.left) {
16         parent.counter.getAndIncrement();
17       }
18       node = parent;
19     }
20   }
21   public T removeMin() {
22     TreeNode node = root;
23     while(!node.isLeaf()) {
24       if (node.counter.boundedGetAndDecrement() > 0 ) {
25         node = node.left;
26       } else {
27         node = node.right;
28       }
29     }
30     return node.bin.get();
31   }
32 }
```

图15-4 SimpleTree有界范围优先级队列

add(x, k)调用将x添加到第k个叶结点的池中，并按照由叶子到根的顺序增加结点的计数器，removeMin()方法按照由根到叶子的顺序遍历树，从根开始，查找具有最高优先级且其池不为空的树叶，它沿着每个结点的计数器，如果为0则向右移动，否则，减少计数器并向左移动（第24行）。

```
33   public class TreeNode {
34     Counter counter;
35     TreeNode parent, right, left;
36     Bin<T> bin;
37     public boolean isLeaf() {
38       return right == null;
39     }
40   }
```

图15-5 SimpleTree类：内部的treeNode类

线程A向上进行的add()调用可能会遇到线程B向下执行的removeMin()调用。与Hansel和Gretel的算法一样，向下的线程B根据上升的add()所留下的非零计数器的痕迹，来确定并从其池中删除A的元素。图15-3中的a描述了SimpleTree的一个执行实例。

有可能会担心出现下面的“格林童话”场景。如图15-3所示，线程A向上移动，在标有星号的结点处遇到向下移动的线程B，线程B从该结点向下移动，收集A在树叶处的元素，同时，A继续向上，增加计数器直到到达根结点为止。如果另一个线程C开始跟随A的非零计数器路

径，从根开始向下移动到A和B相遇的显号结点，将会出现什么情况？当C到达该显号结点时，有可能被困在树的中间的这个地方，它发现从右孩子到达一个空的Bin没有能够追寻的标记，即使在队列中可能存在其他的元素。

幸运的是，这种场景不会发生。如图15-3b至d所示，向下的线程B在显号结点处与向上的线程A相遇的唯一途径就是，由一个更早的线程D调用的另外一个add()从显号结点到根增加了至一个计数器集合，才允许向下的线程B首先到达显号结点。向上的线程A从显号结点到根结点增加计数器时，只是简单地完成了通向由某个其他的线程D所插入元素的递增序列。总之，如果在第24行某些线程返回的元素为null，那么优先级队列的确是空的。

SimpleTree算法是不可线性化的，因为线程之间有可能相互追赶。但它是静态一致的。如果所有的池和计数器是无锁的，那么add()和removeMin()方法也是无锁的（add()所需的操作步数目受限于树的深度，只有当不断地从树中添加和删除元素时，removeMin()才可能无法完成）。一次典型的插入或删除操作需要最低优先级（最大的分数）的对数个操作步。

15.4 基于堆的无界优先级队列

本节介绍一种可线性化的优先级队列，其优先数取自一个无界的范围，该队列采用细粒度上锁进行同步。

堆是一种每个结点都包含一个元素和一个优先数的树，如果b是a的孩子结点，那么b的优先级不大于a的优先级（也就是说，树中越高的元素具有越低的优先数值和越高的优先级）。removeMin()方法删除并返回树的根，然后重新平衡根的子树。这里，我们只考虑二叉树，它仅有两个子树需要重新平衡。

355

15.4.1 顺序堆

图15-6和图15-7是顺序堆的一种实现。描述二叉堆的一种有效方式就是将其看作是由结点组成的数组，其中，树的根是数组项1，而数组项i的右孩子和左孩子分别为2·i和(2·i) + 1。next域则为第一个未使用结点的索引。

每个结点有一个item域和一个score域。为了增加一个元素，add()方法将child设为第一个空数组槽的索引（第13行）。（为简单起见，我们省略了重新调整满数组大小的那部分代码。）然后，初始化这个结点，使其具有新元素和优先数（第14行）。此时，堆的性质有可能被坏，因为这个新结点为树的一个叶结点，却可能具有比祖先结点更高的优先级（较小的优先数）。为了恢复堆的性质，这个新结点"向上过滤"树。不断地比较新结点和其父结点的优先级，如果父结点的优先级低（较大的优先数），则相互交换。如果遇到一个更高优先级的父结点，或已到达根结点，那么该新结点就找到正确的位置，方法返回。

为了删除并返回最高优先级的元素，removeMin()方法记录根的元素，也就是树中具有最高优先级的元素。（为简单起见，我们省略了处理空堆的那部分代码。）然后，它将一个叶子项移到上面，替换掉根（第27～29行）。如果树为空，则方法返回记录的元素（第30行）。否则，堆的特性有可能被坏，因为最近被推到根上的叶结点可能具有比它的某些子孙结点更低的优先级。为了恢复堆的性质，新的根"向下过滤"树。如果两个孩子都为空，则结束（第37行）。如果右孩子为空，或者右孩子的优先级比左孩子低，则检查左孩子（第39行）。否则，就检查右孩子（第41行）。如果这个孩子的优先级比父结点高，则交换孩子结点和父结点，并继续向下移动（第44行）。当两个孩子都具有更低的优先级，或者到达了一个叶结点时，该置

换结点就找到了正确的位置，方法返回。

```
1  public class SequentialHeap<T> implements PQueue<T> {
2    private static final int ROOT = 1;
3    int next;
4    HeapNode<T>[] heap;
5    public SequentialHeap(int capacity) {
6      next = ROOT;
7      heap = (HeapNode<T>[]) new HeapNode[capacity + 1];
8      for (int i = 0; i < capacity + 1; i++) {
9        heap[i] = new HeapNode<T>();
10     }
11   }
12   public void add(T item, int score) {
13     int child = next++;
14     heap[child].init(item, score);
15     while (child > ROOT) {
16       int parent = child / 2;
17       int oldChild = child;
18       if (heap[child].score < heap[parent].score) {
19         swap(child, parent);
20         child = parent;
21       } else {
22         return;
23       }
24     }
25   }
```

图15-6 SequentialHeap类：内部结点类和add()方法

```
26   public T removeMin() {
27     int bottom = --next;
28     T item = heap[ROOT].item;
29     heap[ROOT] = heap[bottom];
30     if (bottom == ROOT) {
31       return item;
32     }
33     int child = 0;
34     int parent = ROOT;
35     while (parent < heap.length / 2) {
36       int left = parent * 2; int right = (parent * 2) + 1;
37       if (left >= next) {
38         return item;
39       } else if (right >= next || heap[left].score < heap[right].score) {
40         child = left;
41       } else {
42         child = right;
43       }
44       if (heap[child].score < heap[parent].score) {
45         swap(parent, child);
46         parent = child;
47       } else {
48         return item;
49       }
50     }
51     return item;
52   }
53   ...
54 }
```

图15-7 SequentialHeap类：removeMin()方法

15.4.2 并发堆

简介

FineGrainedHeap类基本上是SequentialHeap类的一种并发版本。和顺序版一样，add()创建一个新的叶结点，并在树中向上过滤这个结点，直到堆的性质被恢复为止。为了允许并发调用以实现并行推进，FineGrainedHeap类将元素的向上过滤看作是一个离散的原子操作步的序列，它可以与其他的操作步相互交叉执行。同样，removeMin()方法删除根结点，将一个叶结点移到根部，并向下过滤这个结点，直到堆的性质被恢复为止。FineGrainedHeap类将元素的向下过滤看作是一个离散的、可以与其他同类操作步相互交叉的原子操作步的序列。

356
1
357

详细介绍

警告：下面的代码不考虑堆的上溢处理（在堆满时增加一个元素）和下溢处理（堆空时删除一个元素）。这些情况的处理会使代码变长，却不增加任何乐趣。

该类使用heapLock域对两个或更多的域做简短的原子修改（图15-8）。

```
1  public class FineGrainedHeap<T> implements PQueue<T> {
2    private static int ROOT = 1;
3    private static int NO_ONE = -1;
4    private Lock heapLock;
5    int next;
6    HeapNode<T>[] heap;
7    public FineGrainedHeap(int capacity) {
8      heapLock = new ReentrantLock();
9      next = ROOT;
10     heap = (HeapNode<T>[]) new HeapNode[capacity + 1];
11     for (int i = 0; i < capacity + 1; i++) {
12       heap[i] = new HeapNode<T>();
13     }
14   }
```

图15-8 FineGrainedHeap类：域

HeapNode类（图15-9）提供下面这些域。lock域是进行短暂修改时需要获得的锁（第21行）。向下过滤结点时也要使用。为简单起见，该类提供lock()和unlock()方法直接对结点进行加锁和释放锁。tag域可以是下面状态中的一个：EMPTY意味着结点未使用；AVAILABLE意味着结点有一个元素和一个优先数；BUSY意味着正在向上过滤结点，还未到达正确的位置。当结点处于BUSY状态时，owner域存放负责移动该结点的线程的ID。为简单起见，该类提供一个amOwner方法，当且仅当结点的tag为BUSY且owner是当前线程的时候，才返回*true*。

358

持有锁而向下过滤的removeMin()方法和tag域被设为BUSY而向上过滤的add()方法（图15-10）之间在同步上的不对称性，能够确保如果一个removeMin()调用遇到一个正处于被add()调用引导着向上移动的过程中的结点，则该removeMin()调用不会被延迟。结果是，add()调用必须准备好将它的结点从下面换出。如果该结点消失，add()只需简单地在树中上移。可以肯定，会在当前位置和根之间的某个位置上遇到这个结点。

removeMin()方法（图15-11）获取全局的heapLock，递减next域，返回叶结点的索引，锁定数组中第一个未使用的槽，再释放heapLock（第76－80行）。然后，它将根的元素保存在一个局部变量中，以便稍后将其作为这次调用的结果返回（第81行）。将结点标记为EMPTY和unowned，并与叶结点交换，再对（现在为空的）叶子解锁（第82－84行）。

```
15    private static enum Status {EMPTY, AVAILABLE, BUSY};
16    private static class HeapNode<S> {
17      Status tag;
18      int score;
19      S item;
20      int owner;
21      Lock lock;
22      public void init(S myItem, int myScore) {
23        item = myItem;
24        score = myScore;
25        tag = Status.BUSY;
26        owner = ThreadID.get();
27      }
28      public HeapNode() {
29        tag = Status.EMPTY;
30        lock = new ReentrantLock();
31      }
32      public void lock() {lock.lock();}
33      ... // other methods omitted
34    }
```

图15.9 FineGrainedHeap类：内部的HeapNode类

```
35  public void add(T item, int score) {
36    heapLock.lock();
37    int child = next++;
38    heap[child].lock();
39    heap[child].init(item, score);
40    heapLock.unlock();
41    heap[child].unlock();
42
43    while (child > ROOT) {
44      int parent = child / 2;
45      heap[parent].lock();
46      heap[child].lock();
47      int oldChild = child;
48      try {
49        if (heap[parent].tag == Status.AVAILABLE && heap[child].amOwner()) {
50          if (heap[child].score < heap[parent].score) {
51            swap(child, parent);
52            child = parent;
53          } else {
54            heap[child].tag = Status.AVAILABLE;
55            heap[child].owner = NO_ONE;
56            return;
57          }
58        } else if (!heap[child].amOwner()) {
59          child = parent;
60        }
61      } finally {
62        heap[oldChild].unlock();
63        heap[parent].unlock();
64      }
65    }
66    if (child == ROOT) {
67      heap[ROOT].lock();
68      if (heap[ROOT].amOwner()) {
69        heap[ROOT].tag = Status.AVAILABLE;
70        heap[child].owner = NO_ONE;
71      }
72      heap[ROOT].unlock();
73    }
74  }
```

图15-10 FineGrainedHeap类：add()方法

```
75    public T removeMin() {
76      heapLock.lock();
77      int bottom = --next;
78      heap[ROOT].lock();
79      heap[bottom].lock();
80      heapLock.unlock();
81      T item = heap[ROOT].item;
82      heap[ROOT].tag = Status.EMPTY;
83      heap[ROOT].owner = NO_ONE;
84      swap(bottom, ROOT);
85      heap[bottom].unlock();
86      if (heap[ROOT].tag == Status.EMPTY) {
87        heap[ROOT].unlock();
88        return item;
89      }
90      heap[ROOT].tag = Status.AVAILABLE;
91      int child = 0;
92      int parent = ROOT;
93      while (parent < heap.length / 2) {
94        int left = parent * 2;
95        int right = (parent * 2) + 1;
96        heap[left].lock();
97        heap[right].lock();
98        if (heap[left].tag == Status.EMPTY) {
99          heap[right].unlock();
100         heap[left].unlock();
101         break;
102       } else if (heap[right].tag == Status.EMPTY || heap[left].score
103                                    < heap[right].score) {
104         heap[right].unlock();
105         child = left;
106       } else {
107         heap[left].unlock();
108         child = right;
109       }
110       if (heap[child].score < heap[parent].score
111        && heap[child].tag != Status.EMPTY) {
112         swap(parent, child);
113         heap[parent].unlock();
114         parent = child;
115       } else {
116         heap[child].unlock();
117         break;
118       }
119     }
120     heap[parent].unlock();
121     return item;
122     }
123     ...
124   }
```

图15-11 FineGrainedHeap类：removeMin()方法

此时，这个方法已将它的最终结果记录在一个局部变量中，并将叶子移到根部，将叶子原先的位置标为EMPTY。它保持着根的锁，如果堆仅有一个元素，那么叶子和根是一样的，所以方法检查根是否只被标记为EMPTY。如果是，则释放根上的锁，返回这个元素（第85～89行）。

现在，按照与顺序实现几乎相同的逻辑向下过滤新的根结点，直到到达正确的位置为止。向下过滤的结点被锁定，直到它到达正确位置。当交换两个结点时，将它们两个都锁定，并

交换它们的域。在每一步，方法都锁定结点的左右孩子（第96行）。如果左孩子为空，则将两个孩子解锁并返回（第98行）。如果右孩子为空，而左孩子具有更高的优先级，则对右孩子解锁并检查左孩子（第103行）。否则，对左孩子解锁并检查右孩子（第106行）。

如果孩子结点具有较高的优先级，则交换父结点和孩子结点，并为父结点解锁（第111行）。否则，将孩子结点和父结点解锁，并返回。

并发的add()方法获取heapLock，分配、上锁、初始化一个空的叶结点，并为其解锁（第36～41行）。这个叶结点的tag为BUSY，owner是正在调用的线程。然后，再对叶结点解锁。

随后，继续向上过滤这个结点，用child变量保存结点的轨迹。它先锁定父结点，然后是孩子结点（所有的锁都以升序获取）。如果父结点为AVAILABLE，且孩子结点被调用者拥有，那么，就比较它们的优先级。如果孩子结点具有更高的优先级，则交换它们的域，并向上移动（第50行）。否则，该结点的位置不变，并标记为AVAILABLE和unowned（第53行）。如果孩子结点不是被调用者拥有，则该结点必定已被一个并发的removeMin()方法向上移动，所以，方法只是简单地向上移动，查找它的结点（第58行）。

359～361

图15-12描述了FineGrainedHeap类的一次执行过程。a中给出了堆的树形结构，优先级标在结点中，数组项标在结点之上。next域设为10，表示可以加入一个新元素的下一个数组项。可以看出，线程A开始一个removeMin()调用，从根收集到值1作为要被返回的值，将优先级为10的叶结点移到根部，将next设为9，removeMin()方法检查10是否需要在堆中向下过滤。在b中，线程A在堆中将10向下过滤。同时，线程B将优先级为2的新元素加入堆中，放入最近空出来的数组项9中。新结点的owner为B，B开始向上过滤2，将它与优先级为7的父结点交换。交换之后，释放结点上的锁。同时，A将优先级为10和3的结点相互交换，在c中，A忽略2的busy状态，采用交叉上锁方式交换10和2，然后交换10和7。这样，它从线程B的下面交换了未上锁的2。在d中，当B移到在数组项4中的父结点时，它发现那个它之前向上过滤的优先级为2的busy结点消失了。不管怎样，它继续向上，并在上升中找到优先级为2的结点，将它移到堆中的正确位置。

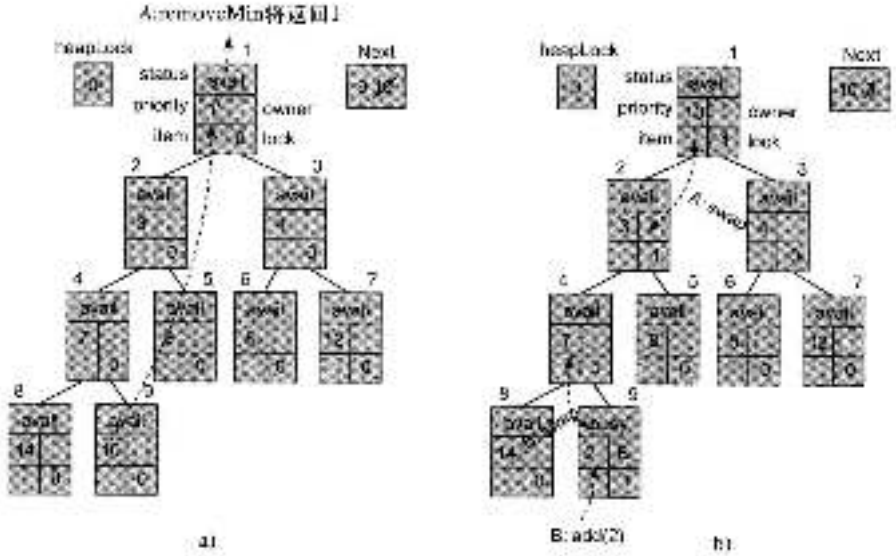


图15-12 FineGrainedHeap类：基于堆的优先级队列

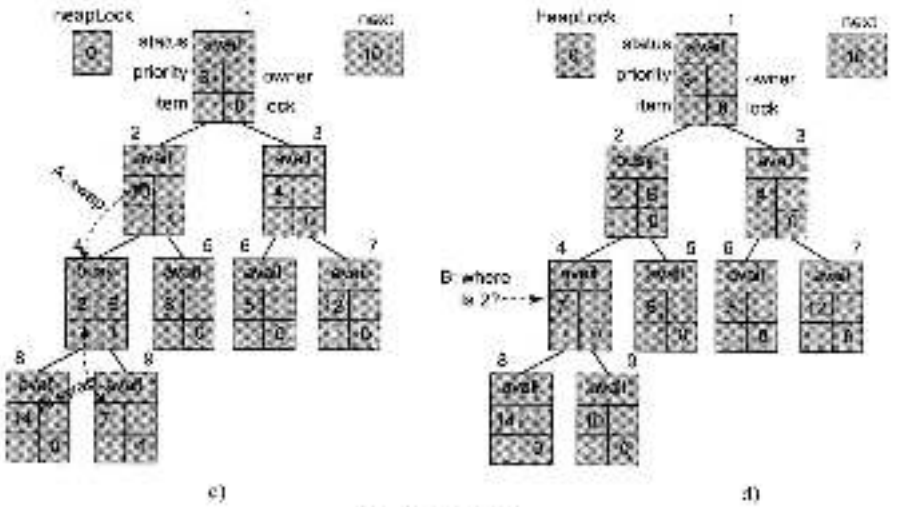


图15-12 （续）

15.5 基于跳表的无界优先级队列

FineGrainedHeap优先级队列算法的缺点之一就是下层的堆结构需要复杂操作的重新平衡。本节将给出一种无需重新平衡的算法。

回顾第14章的内容，跳表是一个由有序链表组成的集合，每个链表是结点的序列，每个结点包含一个元素。每个结点都属于这些链表中的一个子集，每个链表中的结点都按照它们的哈希值排序。每个链表有一个级别（层），从0到最大值。最高层的链表包含所有的结点，每个较高层的链表都是较低层链表的一个子链表。每个链表大约包含其下一层链表中一半的结点。因此，在一个具有k个元素的跳表中插入或者删除一个结点，所花费的期望时间为$O(\log k)$。

第14章采用跳表实现了元素的集合。在此，将采用跳表来实现附有优先级的优先级队列。下面描述一个PrioritySkipList类，它提供实现有效的优先级队列所必须的基本功能。尽管PrioritySkipList类（图15-13和图15-14）可以简单地建立在LazySkipList类之上，但我们还是以第14章的LockFreeSkipList类为基础来构建PrioritySkipList类。之后，为解决PrioritySkipList<T>类的粗糙问题，将介绍一个SkipQueue包装程序。

以下是该算法的概要。PrioritySkipList类按照优先级而不是哈希值对元素进行排序，从而确保高优先级的元素（希望首先删除的元素）出现在链表的前端。图15-15描述了一个这样的PrioritySkipList结构。最高优先级元素的删除是惰性地完成的（见第9章）。先作标记来逻辑地删除一个结点，然后再将该结点从链表中断开以完成物理删除。removeMin()方法按照下面两个步骤进行工作：首先，扫描底层链表查找第一个未标记的结点。如果找到一个结点，则尝试标记该结点。如果尝试失败，则继续向下扫描链表；如果成功，那么removeMin()调用PrioritySkipList类的对数时间的remove()方法，物理地删除这个被标记的结点。

362
|
363

下面转向算法的细节。图15-13描述了PrioritySkipList类的概要，它是第14章中LockFreeSkipList类的一种修改版本。让add()和remove()调用以跳表结点而不是元素作为参数和返回值有利于实现。这些方法是相应的LockFreeSkipList方法的直接变通，将其留作

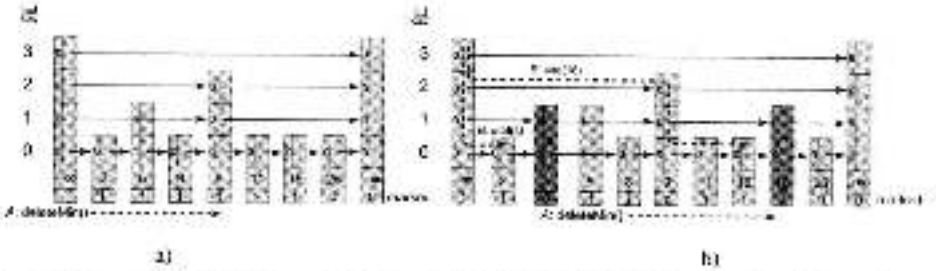
习题。这个类的结点与LockFreeSkipList类的结点有两个域不相同：整型的score域（第4行）和用来在优先级队列中（不在跳表中）进行逻辑删除（第5行）的AtomicBoolean marked域。findAndMarkMin()方法扫描最底层链表，直到找到一个marked域为*false*的结点。然后原子地尝试将这个域设为*true*（第19行）。如果失败，则再次尝试。当它成功时，给调用者返回最近标记的结点（第20行）。

```
1  public final class PriorityQueue<T> {
2    public static final class Node<T> {
3      final T item;
4      final int score;
5      AtomicBoolean marked;
6      final AtomicMarkableReference<Node<T>>[] next;
7      // sentinel node constructor
8      public Node(int myPriority) { ... }
9      // ordinary node constructor
10     public Node(T x, int myPriority) { ... }
11   }
12   boolean add(Node node) { ... }
13   boolean remove(Node<T> node) { ... }
14   public Node<T> findAndMarkMin() {
15     Node<T> curr = null;
16     curr = head.next[0].getReference();
17     while (curr != tail) {
18       if (!curr.marked.get()) {
19         if (curr.marked.compareAndSet(false, true))
20           return curr;
21         } else {
22           curr = curr.next[0].getReference();
23         }
24       }
25     }
26     return null; // no unmarked nodes
27   }
```

图15-13 PrioritySkipList<T>类：内部的Node<T>类

```
1  public class SkipQueue<T> {
2    PrioritySkipList<T> skiplist;
3    public SkipQueue() {
4      skiplist = new PrioritySkipList<>();
5    }
6    public boolean add(T item, int score) {
7      Node<T> node = (Node<T>)new Node(item, score);
8      return skiplist.add(node);
9    }
10   public T removeMin() {
11     Node<T> node = skiplist.findAndMarkMin();
12     if (node != null) {
13       skiplist.remove(node);
14       return node.item;
15     } else{
16       return null;
17     }
18   }
19 }
```

图15-14 SkipQueue<T>类



a) b)

图15-15 SkipQueue优先级队列：一次静态一致但不可线性化的执行。在a中，线程A开始一个removeMin()方法调用。它遍历PrioritySkipList中的最底层链表，以查找并逻辑删除第一个未标记的结点。它遍历了所有的标记结点，其中包括像优先数为5的那种正处于从SkipList中被物理删除的结点。在b中，当A正在访问优先数为9的结点时，线程B增加一个优先数为3的结点，然后增加一个优先数为18的结点，线程A标记并返回优先数为18的结点。一次可线性化的执行在优先数为3的元素被返回之前不能返回优先数为18的元素

图15-14为SkipQueue<T>类。这个类只是PrioritySkipList<T>的一个包装程序。add(x, p)方法增加优先数为p的元素x，先创建一个结点来保存这两个值，再将该结点传递给PrioritySkipList类的add()方法。removMin()方法调用PrioritySkipList类的findAndMarkMin()方法来将一个结点标记为逻辑删除，然后调用remove()物理地删除该结点。

SkipQueue类是静态一致的：如果元素x在removMin()调用开始之前就已存在的话，则返回元素的优先数将小于等于x的优先数。该类是不可线性化的：一个线程可能会增加一个较高优先级（较低优先数）的元素，然后再增加一个较低优先级的元素。正在遍历的线程则可能发现并返回后面插入的较低优先级的元素，从而违背了可线性化性。然而，这种行为是静态一致的，因为可以用任何removeMin()来并发地重新排序add()调用，使其与一个顺序优先级队列保持一致。

364, 365

SkipQueue类是无锁的。一个正在遍历SkipList最底层的线程，有可能总是被另一个调用排除到下一个逻辑上未删除的结点，但仅仅在其他线程不断成功的情况下才会不断地失败。

总之，静态一致的SkipQueue试图超越基于锁的可线性化队列。如果存在有n个线程，那么第一个没有被逻辑删除的结点总是在最底层链表的前n个结点之中。一旦一个结点被逻辑删除，那么在最坏情况下，它将在O(logk)个步骤内被物理删除，其中，k为链表的大小。在实际中，一个结点的删除可能要比这快得多，因为该结点很可能靠近链表的起始位置。

然而，算法中一些会引起争用的因素可能影响着算法的性能，因此需要使用回退和调谐。如果有几个线程试图并发地标记一个结点，则会出现争用。失败者将一起尝试标记下一个结点，如此反复。当从跳表中物理地删除一个元素时，也会出现争用。所有要被删除的结点可能是处于跳表起始位置的相邻结点，所以它们具有前驱的概率很高，这可能导致在试图断开指向结点的链接时，compareAndSet()调用不断地失败。

15.6 本章注释

FineGrainedHeap优先级队列是由Galen Hunt、Maged Michael、Srinivasan Parthasarathy和Michael Scott[74]提出的。SimpleLinear和SimpleTree优先级队列则是由Nir Shavit和Asaph Zemach[143]提出的。SkipQueue是由Itai Lotan和Nir Shavit[107]提出的。他们也提出了该算法的一个可线性化版本。

15.7 习题

习题173. 试给出一个静态一致但不可线性化的优先级队列执行实例。

习题174. 采用计数网或衍射树，实现无锁的boundedGetAndIncrement()和boundedGetAndDecrement()方法，从而实现一个静态一致的Counter。

习题175. 在SimpleTree算法中，如果用规则的GetAndDecrement()方法替代boundedGetAndDecrement()方法，会出现什么情况？

习题176. 在treeNode计数器中，使用boundedGetAndIncrement()方法，设计一种具有有界容量的SimpleTree算法。

习题177. 在SimpleTree类中，如果add()将一个元素放入适当的Bin之后，采用与removeMin()方法一样从上到下的方式增加了计数器，将会出现什么情况？试给出一个详细的示例。

习题178. 证明SimpleTree是静态一致的优先级队列实现。

习题179. 修改FineGrainedHeap，使得能动态地分配新的堆结点。这种方法的性能局限性体现在哪里？

习题180. 图15-16给出了一个按位倒置的计数器。可以使用这种计数器来管理FineGrainedHeap类的next域。试证明：对于任意两个连续的插入，从叶子到根的两条路径除了根之外没有共同的结点。为什么这是FineGrainedHeap的一个有用的性质？

习题181. 给出PrioritySkipList类中add()和remove()方法的代码。

习题182. 本章的PrioritySkipList类是基于LockFreeSkipList类的。写出另一种基于LazySkipList类的PrioritySkipList类。

习题183. 给出SkipQueue实现的一个场景，其中，争用是由多个并发的removeMin()方法调用所引起的。

习题184. SkipQueue类是静态一致的但不是可线性化的。下面是一种通过加入一个简单的时间戳机制，可以使得该类变为可线性化的方法。在一个结点被完全插入到SkipQueue之后，它获取一个时间戳。一个正在执行removeMin()的线程注意到它开始遍历SkipQueue中较低层的时刻，只考虑时间戳早于它开始遍历的时间，有效地忽略在它遍历过程中被插入的结点。试实现这个类并证明它为什么可行。

```
1  public class BitReversedCounter {
2    int counter, reverse, highBit;
3    BitReversedCounter(int initialValue) {
4      counter = initialValue;
5      reverse = 0;
6      highBit = -1;
7    }
8    public int reverseIncrement() {
9      if (counter++ == 0) {
10       reverse = highBit = 1;
11       return reverse;
12     }
13     int bit = highBit >> 1;
14     while (bit != 0) {
15       reverse ^= bit;
16       if ((reverse & bit) != 0) break;
17       bit >>= 1;
18     }
19     if (bit == 0)
20       reverse = highBit <<= 1;
21     return reverse;
22   }
23   public int reverseDecrement() {
24     counter--;
25     int bit = highBit >> 1;
26     while (bit != 0) {
27       reverse ^= bit;
28       if ((reverse & bit) == 0) {
29         break;
30       }
31       bit >>= 1;
32     }
33     if (bit == 0) {
34       reverse = counter;
35       highBit >>= 1;
36     }
37     return reverse;
38   }
39 }
```

图15-16 按位倒置计数器

第16章

The Art of Multiprocessor Programming (Revised First Edition)

异步执行、调度和工作分配

16.1 引言

本章将阐述如何将某种类型的问题分解成多个可并行执行的部分。有些应用可以很自然地分解为多个可并行的线程。例如，若Web服务器收到一个请求，它就创建一个线程（或者分配给一个已经存在的线程）来处理这个请求。能被组织成生产者和消费者的应用往往也是可并行化的。然而，在本章中，我们将探讨那些表面上看似乎无法直接并行但其内在本质又具有并行性的应用。

首先来考虑如何并行地求解两个矩阵的乘积。回顾一下：如果a_{ij}是矩阵A中处于第(i, j)个位置的值，那么两个$n \times n$矩阵A和B的乘积C可由下式给出：

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

第一步，可以让一个线程负责计算一个c_{ij}。图16-1描述了一个矩阵乘法程序，它创建了一个由Worker线程组成的$n \times n$数组（图16-2），其中，处于位置(i, j)的Worker线程计算c_{ij}。程序将启动每个任务，并等待这些任务全部完成。[⊖]

```
1  class MMThread {
2    double[][] a, b, c;
3    int n;
4    public MMThread(double[][] myA, double[][] myB) {
5      n = myA.length;
6      a = myA;
7      b = myB;
8      c = new double[n][n];
9    }
10   void multiply() {
11     Worker[][] worker = new Worker[n][n];
12     for (int row = 0; row < n; row++)
13       for (int col = 0; col < n; col++)
14         worker[row][col] = new Worker(row,col);
15     for (int row = 0; row < n; row++)
16       for (int col = 0; col < n; col++)
17         worker[row][col].start();
18     for (int row = 0; row < n; row++)
19       for (int col = 0; col < n; col++)
20         worker[row][col].join();
21   }
```

图16-1 MMThread任务：采用多线程实现矩阵乘法

理论上讲，这可能是一种完美的设计。程序是高度并行化的，线程甚至都不需要同步。然而在实际中，这种设计对于小矩阵的效果很好，但对于非常大的矩阵则并不理想。其原因

⊖ 在实际的代码中，必须检查各种异常，这里为了简明起见，省略了大多数安全性检查。

在于：线程需要内存来存放栈和其他信息。创建、调度和清除线程都需要大量的计算。创建多个短生命期的线程来进行多线程计算并不是一种有效的方式。

```
22    class Worker extends Thread {
23      int row, col;
24      Worker(int myRow, int myCol) {
25        row = myRow; col = myCol;
26      }
27      public void run() {
28        double dotProduct = 0.0;
29        for (int i = 0; i < n; i++)
30          dotProduct += a[row][i] * b[i][col];
31        c[row][col] = dotProduct;
32      }
33    }
34  }
```

图16-2 MMThread任务：内部的Worker线程类

组织这个程序的一种更为有效的方式就是创建一个由长生命期的线程组成的池。池中的每个线程一直在等待，直到被分派一个任务（短期的计算单元）为止。若给一个线程分派了一个任务，那么该线程将执行这个任务，然后重新进入池中等待下一次分派。线程池是平台相关的：对于大规模多处理器系统提供多个较大的池，反之亦然。线程池避免了由于短生命期的频繁变化，带来的创建和清除线程的代价。

除了性能方面的优势之外，线程池还具有另外一个不太明显但同样重要的优点：它们能隔离应用程序与特定平台的细节（如可以有效调度的并发线程数）。采用线程池能使我们编写出在单处理器、小规模多处理器和大规模多处理器上都可以高效运行的程序。线程池提供了简单的接口，掩盖了复杂的、平台相关的工程性折中细节。

369
370

在Java中，线程池被称作执行者服务（ExecutorService，java.util.ExecutorService的接口）。它为我们提供了提交任务、等待已提交任务集完成以及撤销未完成任务的能力。没有返回值的任务被表示为Runnable对象，其工作由一个不带参数、无返回值的run()方法来完成。返回值类型为T的任务则被表示为Callable<T>对象，其结果通过一个类型为T的不带参数的call()方法返回。

当一个Callable<T>对象提交给执行者服务时，该服务将返回一个实现了Future<T>接口的对象。一旦Future<T>准备就绪，它就是实际异步计算结果的一种保证。该对象提供了能返回异步计算结果的get()方法，在需要时能够阻塞直到结果准备就绪。（它也提供了撤销未完成计算和检查计算是否完成的方法。）提交一个Runnable任务也会返回一个future。与Callable<T>对象返回的future不同，该future并不返回值，但调用者可以使用这个future的get()方法进行阻塞，直到计算完成为止。没有返回值的future被声明为具有类Future<?>。

理解下面这一点非常重要：创建一个future并不能保证所有的计算在实际中都是并行执行的。相反，这些方法都是劝告式的：它们告诉底层的一个执行者服务，它可以并行地执行这些方法。

现在考虑如何使用执行者服务来实现并行矩阵操作。图16-3描述了一个Matrix类，它提供put()和get()方法来访问矩阵元素，并提供一个常数时间的split()方法来将一个$n \times n$的矩阵划分为4个$(n/2) \times (n/2)$的子矩阵。在Java术语中，4个子矩阵能返回到原始矩阵，也就是说，对子矩阵的改变能反映到原始矩阵中，反之亦然。

我们要做的就是设计一个MatrixTask类，它提供矩阵加法和矩阵乘法的并行方法。该类有一个静态域、一个称为exec的执行者服务，以及两个分别用于求和及求积的静态方法。

为简单起见，我们考虑n是2的幂的矩阵。每个这样的矩阵都可以分解为4个子矩阵：

```
1  public class Matrix {
2    int dim;
3    double[][] data;
4    int rowDisplace, colDisplace;
5    public Matrix(int d) {
6      dim = d;
7      rowDisplace = colDisplace = 0;
8      data = new double[d][d];
9    }
10   private Matrix(double[][] matrix, int x, int y, int d) {
11     data = matrix;
12     rowDisplace = x;
13     colDisplace = y;
14     dim = d;
15   }
16   public double get(int row, int col) {
17     return data[row+rowDisplace][col+colDisplace];
18   }
19   public void set(int row, int col, double value) {
20     data[row+rowDisplace][col+colDisplace] = value;
21   }
22   public int getDim() {
23     return dim;
24   }
25   Matrix[][] split() {
26     Matrix[][] result = new Matrix[2][2];
27     int newDim = dim / 2;
28     result[0][0] =
29       new Matrix(data, rowDisplace, colDisplace, newDim);
30     result[0][1] =
31       new Matrix(data, rowDisplace, colDisplace + newDim, newDim);
32     result[1][0] =
33       new Matrix(data, rowDisplace + newDim, colDisplace, newDim);
34     result[1][1] =
35       new Matrix(data, rowDisplace + newDim, colDisplace + newDim, newDim);
36     return result;
37   }
38 }
```

图16-3 Matrix类

$$A=\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

矩阵加法$C = A + B$可以分解为下式：

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}=\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}+\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$=\begin{pmatrix} A_{00}+B_{00} & A_{01}+B_{01} \\ A_{10}+B_{10} & A_{11}+B_{11} \end{pmatrix}$$

四个求和过程可以并行完成。

图16-4给出了多线程矩阵加法的代码。AddTask类有3个域，通过构造函数来初始化：a和b是要相加的两个矩阵，c是结果，其内容要被修改。每个任务的执行过程如下：在递归的最底层，它只是简单地将两个标量值相加（第20行）。[⊖]否则，它将每个参数分解为4个子矩阵

⊖ 实际中，当矩阵大小达到1之前停止递归往往更有效，最佳的大小是依赖于平台的。

(第22行)，并为每个子矩阵发起一个新任务（第24～27行）。然后，等待直到所有的future都能被求值。也就是说子计算都已完成（第28～30行）。此时，任务简单地返回，计算结果已保存在结果矩阵中。矩阵乘法$C = A \cdot B$可以如下分解：

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{pmatrix}$$

```
1  public class MatrixTask {
2    static ExecutorService exec = Executors.newCachedThreadPool();
3    ...
4    static Matrix add(Matrix a, Matrix b) throws ExecutionException {
5      int n = a.getDim();
6      Matrix c = new Matrix(n);
7      Future<?> future = exec.submit(new AddTask(a, b, c));
8      future.get();
9      return c;
10   }
11   static class AddTask implements Runnable {
12     Matrix a, b, c;
13     public AddTask(Matrix myA, Matrix myB, Matrix myC) {
14       a = myA; b = myB; c = myC;
15     }
16     public void run() {
17       try {
18         int n = a.getDim();
19         if (n == 1) {
20           c.set(0, 0, a.get(0,0) + b.get(0,0));
21         } else {
22           Matrix[][] aa = a.split(), bb = b.split(), cc = c.split();
23           Future<?>[][] future = (Future<?>[][]) new Future[2][2];
24           for (int i = 0; i < 2; i++)
25             for (int j = 0; j < 2; j++)
26               future[i][j] =
27                 exec.submit(new AddTask(aa[i][j], bb[i][j], cc[i][j]));
28           for (int i = 0; i < 2; i++)
29             for (int j = 0; j < 2; j++)
30               future[i][j].get();
31         }
32       } catch (Exception ex) {
33         ex.printStackTrace();
34       }
35     }
36   }
37 }
```

图16-4 MatrixTask类：并行的矩阵加法

373

8个乘积项可以并行地计算。当这些计算都已完成时，再并行地计算4个求和。

图16-5给出了并行矩阵乘法任务的代码。矩阵乘法类似于矩阵加法。MulTask类创建了两个临时数组来保存矩阵的乘积项（第42行）。它将所有的5个矩阵进行分割（第50行），并将这些任务提交以并行地计算8个乘积项（第56行），然后等待它们完成（第60行）。一旦这些任务完成，线程就提交任务，并行地计算4个求和（第64行），并等待它们完成（第65行）。

```
38    static class MulTask implements Runnable {
39      Matrix a, b, c, lhs, rhs;
40      public MulTask(Matrix myA, Matrix myB, Matrix myC) {
41        a   = myA; b = myB; c = myC;
42        lhs = new Matrix(a.getDim());
43        rhs = new Matrix(a.getDim());
44      }
45      public void run() {
46        try {
47          if (a.getDim() == 1) {
48            c.set(0, 0, a.get(0,0) * b.get(0,0));
49          } else {
50            Matrix[][] aa = a.split(), bb = b.split(), cc = c.split();
51            Matrix[][] ll = lhs.split(), rr = rhs.split();
52            Future<?>[][][] future = (Future<?>[][][]) new Future[2][2][2];
53            for (int i = 0; i < 2; i++)
54              for (int j = 0; j < 2; j++) {
55                future[i][j][0] =
56                  exec.submit(new MulTask(aa[i][0], bb[0][j], ll[i][j]));
57                future[i][j][1] =
58                  exec.submit(new MulTask(aa[i][1], bb[1][j], rr[i][j]));
59              }
60            for (int i = 0; i < 2; i++)
61              for (int j = 0; j < 2; j++)
62                for (int k = 0; k < 2; k++)
63                  future[i][j][k].get();
64            Future<?> done = exec.submit(new AddTask(lhs, rhs, c));
65            done.get();
66          }
67        } catch (Exception ex) {
68          ex.printStackTrace();
69        }
70      }
71    }
72    ...
73  }
```

图16-5 MatrixTask类：并行的矩阵乘法

该矩阵实例中仅仅使用future来发出任务完成的信号。future也可以用来从已完成的任务
374 中传递值。为了说明future的这种用法，我们来考虑如何将众所周知的斐波那契数列函数分解成一个多线程程序。回顾一下斐波那契数列的定义如下：

$$F(n)=\begin{cases}1 & \text{如果}n=0\\ 1 & \text{如果}n=1\\ F(n-1)+F(n-2) & \text{如果}n>1\end{cases}$$

图16-6描述了并行地计算斐波那契数列的一种方法。这个实现的效率非常低，这里只是用它来说明与多线程之间的关系。call()方法创建了两个future，一个计算$F(n-2)$，另一个计算$F(n-1)$，然后将它们相加。在多处理器系统中，花费在$F(n-1)$的future上的阻塞时间可以用来计算$F(n-2)$。

```
1  class FibTask implements Callable<Integer> {
2    static ExecutorService exec = Executors.newCachedThreadPool();
3    int arg;
4    public FibTask(int n) {
5      arg = n;
6    }
7    public Integer call() {
8      if (arg >= 2) {
9        Future<Integer> left = exec.submit(new FibTask(arg-1));
10       Future<Integer> right = exec.submit(new FibTask(arg-2));
11       return left.get() + right.get();
12     } else {
13       return 1;
14     }
15   }
16 }
```

图16-6 FibTask类：一个有future的斐波那契任务

16.2 并行分析

多线程计算可以看成是一个无环有向图（Directed Acyclic Graph，DAG），其中，每个结点代表一个任务，每条有向边连接着一个前驱任务和一个后继任务，后继任务依赖于前驱任务的计算结果。例如，一个常规的线程就是一个结点链，其中每个结点依赖于它的前驱结点。相比较而言，一个创建了future的结点有两个后继：一个是它在同一线程中的后继，另一个则是在future计算中的第一个结点。在从孩子指向父亲的方向也存在边，当一个已经创建了future的线程调用这个future的get()方法并等待这个孩子计算完成时，就会出现这种情况。图16-7描述了对应于一次给定的斐波那契数列执行的DAG。

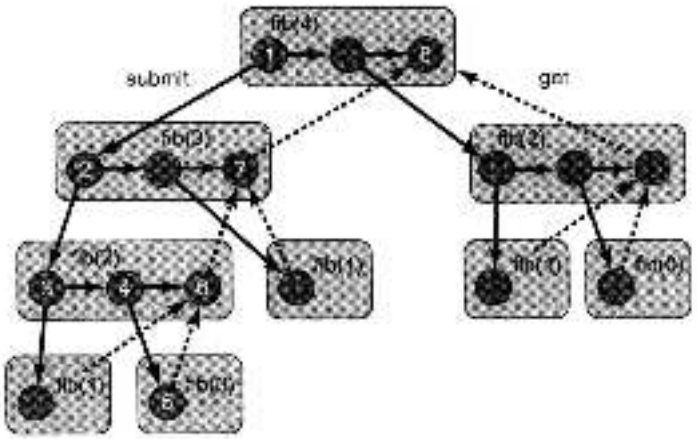


图16-7 一次多线程斐波那契数列执行的DAG。调用者创建一个FibTask(4)任务，该任务又创建了任务FibTask(3)和FibTask(2)。圆形的结点表示计算步骤，结点间的箭头表示依赖关系。例如，存在着从FibTask(4)中的前两个结点分别指向FibTask(3)和FibTask(2)中的第一个结点的两个箭头表示submit()调用，以及从FibTask(3)和FibTask(2)的最后一个结点指向FibTask(4)的最后结点的箭头表示get()调用。计算的关键路径长度为8，由编号的结点标出

某些计算本身的并行度要高于另一些计算。现在我们来明确这个概念。假定所有的单个计算步所需的时间相同，将它作为基本的测量单位。令T_P为在一个有P个专门处理器的系统上执行一个多线程程序所需的最小时间（按照计算步测量），那么T_P就是该程序的时延，即从外部观测者的角度来看，是程序从开始到完成所需的时间。要强调一点，即T_P只是一个理想化的测量值；有可能无法找出每一个处理器上执行的操作步。实际的执行时间可能受限于其他条件，例如存储器的使用情况。但是，毫无疑问，T_P是从一个多线程计算中可提取的并行度的下限。

与T相关的某些值比较重要，它们有特殊的名字。T_1是在单个处理器上执行程序所需的操作步数，称为计算的工作。工作也是整个计算过程中的总操作步数。在一个时间操作步（从外部观测者的角度）中，P个处理器最多可以执行P个计算步骤，因此，

$$T_P \geq T_1/P$$

另一个极端也很重要：T_∞。它是在无限数量的处理器上执行程序所需的操作步数，称为关键路径长度。因为在有限资源上不可能比无限资源的效果更好，所以

$$T_P \geq T_\infty$$

P个处理器上的加速比为比值

$$T_1/T_P$$

如果$T_1/T_P = \Theta(P)$，则称计算具有线性加速比。最后，计算的并行度是可能的最大加速比：T_1/T_∞。计算的并行度也是关键路径上每一个步骤中可用工作的平均数量。因此，它是计算中需要投入处理器数量的一个很好的参考值。特别是，使用多于这个数目的处理器没有多大意义。

为了说明这些概念，我们再次讨论16.1节中矩阵加法和矩阵乘法的并发实现。

设$A_P(n)$为P个处理器上进行两个$n \times n$的矩阵加法所需要的步骤数。回顾一下，矩阵加法需要4个一半大小的矩阵相加，以及用于分解矩阵的常数数量的工作。工作$A_1(n)$可递归地给出：

$$\begin{aligned} A_1(n) &= 4A_1(n/2) + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

这个程序具有和通常采用双重嵌套循环一样的工作。

因为对一半大小的矩阵进行相加可以并行地执行，所以关键路径长度由下式给出：

$$\begin{aligned} A_\infty(n) &= A_\infty(n/2) + \Theta(1) \\ &= \Theta(\log n) \end{aligned}$$

设$M_P(n)$为P个处理器上进行两个$n \times n$的矩阵乘法所需要的步骤数。回顾一下，矩阵乘法需要8个一半大小的矩阵的乘积和4个矩阵的求和。工作$M_1(n)$可以递归地给出：

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + 4A_1(n) \\ M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

这个程序也具有和通常三重嵌套循环实现一样的工作，一半大小的乘法可以并行完成，加法同样如此，但是加法必须等待乘法完成。关键路径长度由下式给出：

$$\begin{aligned} M_\infty(n) &= M_\infty(n/2) + A_\infty(n) \\ &= M_\infty(n/2) + \Theta(\log n) \\ &= \Theta(\log^2 n) \end{aligned}$$

矩阵乘法的并行度由下式给出：

$$M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n)$$

这个并行度是很高的。比如，假设想要对两个1000 × 1000的矩阵求积，这里，$n^3 = 10^9$，$\log n = \log 1000 \approx 10$（log的底为2），所以并行度大约为$10^9/10^2 = 10^7$，粗略来说，这个矩阵乘法实例大约占用百万个处理器，远远超过稍后我们将会看到的任何一种多处理器的能力。

必须理解这一点，即这里所给的计算并行度是任何多线程矩阵乘法程序高度理想化的性能上限。例如，当线程空闲时，将这些线程分派给空闲的处理器可能并非易事。另外，一个使用较低并行度但却消耗更少存储空间的程序有可能具有更好的性能，因为它遇到的页故障更少。多线程计算的实际性能是一个复杂的工程问题，但本章中所分析的内容是理解用并行方法解决问题不可缺少的第一步。

16.3 多处理器的实际调度

迄今为止，我们的分析都是建立在每个多线程程序有P个专门的处理器的假设之上的。不幸的是，这种假设并不等于实际的情形。在多处理器上往往会运行着多个作业，这些作业动态地开始和结束。例如，某个用户在P个处理器上启动了一个矩阵乘法程序。而某一时刻，操作系统有可能决定下载一个新的软件升级包，从而抢占了一个处理器，此时应用程序运行在$P-1$个处理器上。若升级程序又要暂停等待磁盘读写的完成，那么矩阵计算程序又拥有了P个处理器。

现代操作系统提供了用户级别的线程，它包含一个程序计数器和一个栈。（具有自己的地址空间的线程通常被称为进程。）操作系统内核中有一个让线程在物理处理器上运行的调度器。然而，应用程序通常不控制线程和处理器之间的映射，因此，无法控制何时调度线程。

如我们所知，在用户级线程和操作系统级的处理器之间建立联系的一种办法就是，给软件开发者提供一种三级模式。在最高层，多线程程序（如矩阵乘法）将应用分解为多个短期任务，它们的数量是动态变化的。在中间层，由用户级的调度器将这些任务映射为固定个数的线程，在最底层，内核将这些线程映射到硬件处理器上，其可利用率是动态变化的。下层的映射不受应用程序的控制；应用无法告诉内核该如何调度线程（特别是，商用操作系统的内核对于用户来说是隐藏的）。

为简单起见，假设内核以离散的操作步进行工作：在第i步，内核在$0 \le p_i \le P$个用户级线程中选择任意一个子集作为一个操作步来运行，T个操作步中的处理器平均数P_A则定义为：

$$P_A = \frac{1}{T}\sum_{i=0}^{T-1} p_i \tag{16.3.1}$$

我们不是设计用户级的调度来获得P倍的加速比，而是要获得P_A倍的加速比。若对于一种调度，在每个时间步上所执行的程序的操作步个数是程序DAG中的p_i、可用处理器的个数、就绪结点（其相关操作步已经做好执行的准备）数中的最小值，则称这种调度是贪心的。换句话说，在给定可用处理器个数的情形下，执行尽可能多的就绪结点。

定理16.3.1 对于一个工作为T_1、关键路径长度为T_∞且有P个用户级线程的多线程程序，可以断定任何贪心执行的长度T的最大值为

$$\frac{T_1}{P_A} + \frac{T_\infty(P-1)}{P_A}$$

证明 方程式（16.3.1）隐含着：

$$T = \frac{1}{P_s}\sum_{i=0}^{T-1} p_i$$

我们通过限制p_i的总和来限定T的范围。在每个内核级的操作步i，可以为每个已分配给一个处理器的线程指定一个令牌，将这些令牌放入两个桶中的一个。对于步骤i中每个执行结点的用户级线程，我们将一个令牌放入一个工作桶中，对于步骤i中每个空闲的线程（即已分配给一个处理器，但因为其下一步相关的结点有依赖，必须等待其他的线程，所以还没有准备就绪），我们将一个令牌放入一个空闲桶中。在最后一步完成之后，工作桶中有T_1个令牌，每个令牌为该计算DAG的一个结点，那么空闲桶中有多少个令牌？

我们将一个空闲操作步定义为：在这个步骤中，其线程将一个令牌放入空闲桶中。因为应用仍在执行，所以在每个步骤中，至少有一个结点准备就绪。又因为调度是贪心的，那么至少有一个结点将被调度，所以，至少有一个处理器不是空闲的。这样，在步骤i中被调度的p_i个线程中，最多有$p_i-1 \le P-1$个线程是空闲的。

那么可能有多少个空闲操作步呢？令G_i为在步骤i结束时还没有执行的结点所组成的计算的子DAG。在G_{i-1}（例如，在步骤6结束时，FibTask(2)中的最后一个结点）中，每个没有输入边的结点（除了在程序次序中它的前驱以外）在步骤i开始时就已准备就绪。这种结点的个数必定小于p_i个，因为否则的话，贪心调度能够执行p_i个这种结点，那么步骤i就不是空闲的。因此，调度器必定已执行了这一步。由此推出G_i的最长有向路径要比G_{i-1}的最长有向路径短。在步骤0之前的最长有向路径是T_∞，所以贪心调度最多有T_∞个空闲操作步。从这些结论中，可以推导出最多有T_∞个空闲操作步被执行，且在每个操作步中最多加入$(P-1)$个令牌，所以空闲桶中最多有$T_\infty(P-1)$个令牌。

因此，两个桶中的令牌总数为

$$\sum_{i=0}^{T-1} p_i \le T_1 + T_\infty(P-1)$$

从而导出给定的界限。□

由此可知，这个界限是两个优化因素中的一个。事实上，获得一个最优的调度是NP完全（NP-complete）问题，所以，贪心调度是获得接近于最优性能的一种简单而实用的方法。

16.4 工作分配

我们现在理解了获得好的加速比的关键就是要保持由任务所提供的用户级线程，从而使调度尽可能贪心。然而，多线程计算是动态地创建和消除任务的，有时是无法预测的。因此，需要一种工作分配算法来尽可能有效地将就绪的任务分配给空闲的线程。

工作分配的一种简单办法就是工作交易（work dealing）：一个超负荷的任务尝试着将任务分给其他的轻负荷线程来减轻负担。这种方法看起来很明智，但却存在一个致命的缺点：如果大多数线程都是超负荷的，那么它们将会把时间浪费在交换任务的无用的尝试中。相反，我们首先考虑工作窃取（work stealing）：一个无法工作的线程尝试着从其他线程那里“偷窃”工作。工作窃取的优点之一就是：如果所有的线程都处于忙状态，那么它们不必浪费时间去尝试将任务分给其他线程。

16.4.1 工作窃取

每个线程以双端队列（double-ended queue，DEQueue）的形式保持一个等待执行的任务池。这个队列提供了pushBottom()、popBottom()和popTop()方法（不需要pushTop()方法）。当线程创建一个新任务时，它就调用pushBottom()方法将这个任务加入它的DEQueue中。当线程需要一个任务继续工作时，它就调用popBottom()方法从它自己的DEQueue中删除一个任务。如果线程发现它的队列为空，它就变成一个*窃贼者*：随机地选择一个牺牲者（victim）线程，并调用该线程的DEQueue的popTop()方法，为自己"偷窃"一个任务。 380

在16.5节中我们设计了一种高效的可线性化的DEQueue实现。图16-8描述了一种实现工作窃取执行者服务所使用的线程的方法。这些线程共享一个DEQueue数组（第2行），每个DEQueue对应一个线程。每个线程不断地从它自己的DEQueue中删除一个任务，并执行它（第12～15行）。如果它无法工作，则不断地随机选择一个牺牲者线程，并尝试从这个牺牲者的DEQueue头窃取一个任务（第16～22行）。为了避免代码混乱，我们忽略了窃取时触发异常的可能性。

```
1  public class WorkStealingThread {
2    DEQueue[] queue;
3    Random random;
4    public WorkStealingThread(DEQueue[] myQueue) {
5      queue = myQueue;
6      random = new Random();
7    }
8    public void run() {
9      int me = ThreadID.get();
10     Runnable task = queue[me].popBottom();
11     while (true) {
12       while (task != null) {
13         task.run();
14         task = queue[me].popBottom();
15       }
16       while (task == null) {
17         Thread.yield();
18         int victim = random.nextInt(queue.length);
19         if (!queue[victim].isEmpty()) {
20           task = queue[victim].popTop();
21         }
22       }
23     }
24   }
25 }
```

图16-8 WorkStealingThread类：一种简化的工作窃取执行者池

若所有队列中的所有工作都已完成了很长时间，这个简化的执行者池可能一直在尝试窃取。为了避免线程陷入对不存在的任务无休止的搜索中，我们可以使用一种将在第17章17.6节中详细描述的*终止检测器*。

16.4.2 同从和多道程序设计

如前所述，多处理器提供了三级计算模式：短期的任务由系统级的线程执行，而线程又由操作系统在固定数量的处理器上调度执行。所谓*多道程序设计环境*，是指线程个数多于处理器个数的环境，这意味着在同一时刻，所有的线程不能同时运行，任何线程在任意时刻都可以被抢占所挂起。为了保证向前推进，必须保证那些仍有工作要做的线程不能无故地被除了

工作窃取以外无事可做的偷窃者线程所延迟。为了避免这种情况的发生，我们让每个偷窃者在试图窃取一个工作之前，立即调用Thread.yield()（图16-8中第17行）。这个调用将偷窃者的处理器让给另一个线程，从而允许未被调度的线程重新获得一个处理器以继续推进。（要注意，如果没有能够运行的未调度线程，那么调用yield()就没有意义。）

16.5 工作窃取双端队列

下面阐述如何实现工作窃取DEQueue。理想情形下，如果有可用的任务，那么工作窃取算法应该提供可线性化的实现，其出队方法应总是返回一个任务，然而在实际中，我们可以采用更弱的条件，允许popTop()调用在它与一个并发的popTop()调用相冲突时返回*null*。虽然可以简单地让失败的偷窃者重新尝试，但在这种情形下，让线程每次在一个不同的、随机选择的DEQueue上重试popTop()操作则更有意义。为了支持这种重试，popTop()调用在它与并发的popTop()调用相冲突时可以返回*null*。

下面描述工作窃取DEQueue的两种实现：第一种比较简单，因为它具有有界的容量，第二种稍微复杂一些，但实质上它的容量不受限，也就是说，它没有溢出的可能。

16.5.1 有界工作窃取双端队列

对于执行者池DEQueue来说，通常出现的情形是，线程调用pushBottom()和popBottom()从它自己的队列中入队或者出队任务。不常发生的情形是，线程调用popTop()方法从另一个线程的DEQueue中窃取任务。显然，对通常的情形进行优化是有意义的。图16-9和图16-10中BoundedDEQueue（有界双端队列）的基本思想就是让pushBottom()和popBottom()方法在通常情形下仅使用读/写操作。如图16-11所示，BoundedDEQueue是一个由任务所组成的数组，并由指向该双端队列底部和顶部的bottom和top域来索引。pushBottom()和popBottom()方法采用读/写方式对bottom引用进行操作。然而，一旦bottom和top域很接近（数组中可能只有一个元素），popBottom()则转而调用compareAndSet()，以便与潜在的popTop()调用进行协调。

```
1  public class BDEQueue {
2    Runnable[] tasks;
3    volatile int bottom;
4    AtomicStampedReference<Integer> top;
5    public BDEQueue(int capacity) {
6      tasks = new Runnable[capacity];
7      top = new AtomicStampedReference<Integer>(0, 0);
8      bottom = 0;
9    }
10   public void pushBottom(Runnable r){
11     tasks[bottom] = r;
12     bottom++;
13   }
14   // called by thieves to determine whether to try to steal
15   boolean isEmpty() {
16     int localTop = top.getReference();
17     int localBottom = bottom;
18     return localBottom <= localTop;
19   }
20 }
21 }
```

图16-9 BoundedDEQueue类：域、构造函数、pushBottom()和isEmpty()方法

```
1   public Runnable popTop() {
2     int[] stamp = new int[1];
3     int oldTop = top.get(stamp), newTop = oldTop + 1;
4     int oldStamp = stamp[0], newStamp = oldStamp + 1;
5     if (bottom <= oldTop)
6       return null;
7     Runnable r = tasks[oldTop];
8     if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
9       return r;
10    return null;
11  }
12  public Runnable popBottom() {
13    if (bottom == 0)
14      return null;
15    bottom--;
16    Runnable r = tasks[bottom];
17    int[] stamp = new int[1];
18    int oldTop = top.get(stamp), newTop = 0;
19    int oldStamp = stamp[0], newStamp = oldStamp + 1;
20    if (bottom > oldTop)
21      return r;
22    if (bottom == oldTop) {
23      bottom = 0;
24      if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
25        return r;
26      }
27    top.set(newTop,newStamp);
28    bottom = 0;
29    return null;
30  }
```

图16-10 BoundedDEQueue类：popTop()和popBottom()方法

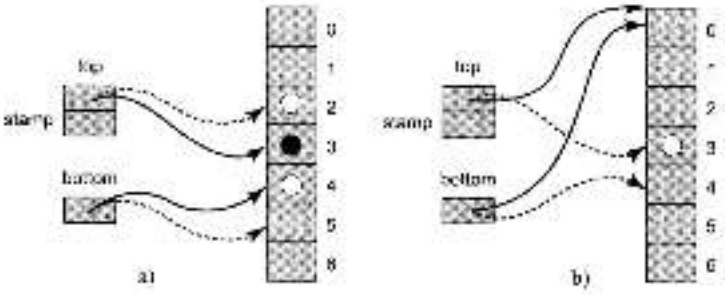


图16-11 BoundedDEQueue的实现。在a中，popTop()和popBottom()被并发地调用，此时，BoundedDEQueue中至少有一个任务。popTop()方法读数组项2中的元素，并试用compareAndSet()重设top引用指向数组项3。popBottom()方法采用一个简单的写操作将bottom引用由5改为4，然后，在确认bottom大于top之后，取除数组项4中的任务。在b中，只有一个单一的任务。当popBottom()检测到将4改为3之后，top和bottom相等，它就尝试用compareAndSet()重新设置top，在进行这个操作之前，它让bottom重新指向0，因为最后一个任务将被两个pop方法中的一个取除。如果popTop()检测到top和bottom相等，则放弃，否则，它将尝试用compareAndSet()来增加top。如果两个方法都对top使用compareAndSet()，那么其中一个将会成功并删除该任务。无论成功或失败，popBottom()都重设top为0，因为该BoundedDEQueue为空

现在来解释算法的细节。BoundedDEQueue算法的精妙之处就在于它避免了使用成本较高的compareAndSet()调用，这种方式需要一点代价：它很微妙，指令之间的次序至关重要。建议读者花点时间去理解方法之间的相互作用是如何由读/写和compareAndSet()调用的次序来决定的。

BoundedDEQueue类有三个域：tasks、bottom和top（图16-9第2～4行）。tasks域是一个由Runnable任务组成的数组，用来存放队列中的任务。bottom域是tasks中第一个空槽的索引。top域是一个AtomicStampedReference<Integer>[⊖]。top域包含有两个逻辑域：引用（reference）是队列中第一个任务的索引，时间戳（stamp）是一个计数器，每次引用改变时，它就被递增。使用时间戳是为了避免出现使用compareAndSet()时经常出现的ABA问题。假设线程A试图从索引3窃取一个任务。A读取该位置任务的引用，调用compareAndSet()将索引置为4以尝试窃取这个任务。在进行调用之前A被延迟，与此同时，线程B删除所有的任务，并插入三个新任务。当A被唤醒时，它的compareAndSet()调用成功地将索引从3改为4，但是它将窃取一个已经完成的任务。时间戳能保证A的compareAndSet()调用失败，因为时间戳不再匹配。

popTop()方法（图16-10）检查BoundedDEQueue是否为空，如果不为空，则调用compareAndSet()递增top，以尝试窃取头元素。如果compareAndSet()成功，则这次窃取成功，否则，该方法简单地返回*null*。这个方法是不确定的：返回*null*并不能说明队列为空。

如前所述，我们对通常的情形进行优化，此时每个线程都从它自己的本地BoundedDEQueue中入队和出队。大部分时间里，线程能够出队或入队自己的BoundedDEQueue对象，只需简单地装载或存储bottom索引。如果队列中只有一个任务，那么调用者有可能与试图窃取这个任务的偷窃者相冲突。所以，如果bottom接近top，那么调用者线程要转而使用compareAndSet()来出队任务。

pushBottom()方法（图16-9第10行）简单地将新任务存储在队列的bottom位置，并递增bottom。

popBottom()方法（图16-10）比较复杂。如果队列为空，方法立刻返回（第13行）。否则，它递减bottom，并声明一个任务（第15行）。下面这点很微妙但也很重要，如果被声明的任务是队列中的最后一个，那么偷窃者能注意到BoundedDEQueue为空则是非常重要的（第5行）。但是，由于popBottom()的递减既不是原子的也不是同步的，所以Java的存储模型并不能保证这次递减恰好被并发的偷窃者观测到。为了保证偷窃者可以识别空的BoundedDEQueue，bottom域必须被声明为**volatile**类型的[⊜]。

递减之后，调用者读取新的bottom索引处的任务（第16行），并测试当前的top域是否指向更高的索引。如果是，调用者不会与偷窃者相冲突，方法返回（第20行）。否则，如果top和bottom域相等，那么在BoundedDEQueue中只有一个任务，也存在着调用者与偷窃者相冲突的危险。调用者重设bottom为0（第23行）。（要么调用者成功地声明这个任务，要么偷窃者先窃取了它。）调用者通过调用compareAndSet()将top重设为0，使其与bottom相匹配。从而解决可能存在的冲突（第22行）。如果这个compareAndSet()成功，那么top已被重设为0且任务

⊖ 见第10章编程提示10.6.1。

⊜ 在C或者C++实现中，需要引入一个写障碍，如附录B所示。

已被声明，所以方法返回。否则，队列肯定为空，因为偷窃者已成功，但这就意味着top指向某个比bottom（早些时候已经被设置为0）更高的数组项。因此，在调用者返回null之前，它将top重设为0（第27行）。

这个设计吸引人的地方就在于，开销较大的compareAndSet()很少被调用，仅仅当BoundedDEQueue近乎为空的时候才会调用。 381 ↓ 385

我们可以在popTop()检测到BoundedDEQueue为空，或者compareAndSet()失败的时刻线性化每个不成功的popTop()调用。成功的popTop()调用可以在成功的compareAndSet()发生的时刻被线性化。在bottom递增的时刻可以线性化pushBottom()调用，而在bottom递减或被设置为0的时刻，可以线性化popBottom()调用，尽管在后一种情况下，popBottom()的结果是由接下来compareAndSet()的成功与否决定的。

isEmpty()方法（图16-13）首先读top，然后读bottom，再检查bottom是否小于等于top（第4行）。操作次序对于可线性化来说很重要，因为top不会减少，而bottom首先被重设为0，所以，如果一个线程在读了top之后再读bottom，并发现bottom不再大于top，那么队列确实为空，因为对top的并发修改只能增加top。另一方面，如果bottom比top大，那么即使在读top之后和读bottom之前top被增加（并且队列变为空），当读top的时候，BoundedDEQueue也不为空。唯一的选择是将bottom重设为0，然后将top重设为0。所以，读top再读bottom将正确地返回空。从而可知，isEmpty()方法是可线性化的。

16.5.2 无界工作窃取双端队列

BoundedDEQueue类的局限性就在于它要求队列具有固定的大小。而对某些应用来说，有可能很难预测这种大小，特别是在某些线程创建的任务要比其他线程多得多的情形下，为每个线程都分配具有最大容量的BoundedDEQueue将会非常浪费空间。

为了解决这种局限性，下面考虑一种无界双端队列（UnboundedDEQueue）类，它能按需动态地调整自己的大小。

我们用一个循环数组来实现UnboundedDEQueue，其top和bottom域与BoundedDEQueue中一样（除了求索引时要对数组容量求模以外）。和前面一样，如果bottom小于或等于top，则UnboundedDEQueue为空。使用循环数组不再需要重设bottom和top为0。另外，它只允许top递增而不能递减，从而不必要求top为AtomicStampedReference。再者，在UnboundedDEQueue算法中，如果pushBottom()发现当前的循环数组已满，它可以重新调整大小（扩大），将任务拷贝到一个更大的数组中去，并将新任务入队到新的（更大的）数组中。因为数组的索引是对它的容量求模所得，所以将元素移到更大的数组中时，不需要修改top和bottom域（尽管存放元素的实际数组索引可能会改变）。

CircularTaskArray()类如图16-12所示。它提供了添加和删除任务的get()和put()方法，以及分配一个新的循环数组并将老数组中的内容拷贝到新数组中的resize()方法。使用模算术能够保证即使数组的大小已改变，任务的位置有可能改变，偷窃者也可以使用top域找到下一个要窃取的任务。 386

UnboundedDEQueue类有三个域：tasks、bottom和top（图16-13第3--5行）。popBottom()和popTop()方法（图16-14）与BoundedDEQueue中相应的方法基本上相同，只有一个关键的不同之处：使用模算术计算索引意味着top索引决不会减小。如我们所知，没有必要

```
1  class CircularArray {
2    private int logCapacity;
3    private Runnable[] currentTasks;
4    CircularArray(int myLogCapacity) {
5      logCapacity = myLogCapacity;
6      currentTasks = new Runnable[1 << logCapacity];
7    }
8    int capacity() {
9      return 1 << logCapacity;
10   }
11   Runnable get(int i) {
12     return currentTasks[i % capacity()];
13   }
14   void put(int i, Runnable task) {
15     currentTasks[i % capacity()] = task;
16   }
17   CircularArray resize(int bottom, int top) {
18     CircularArray newTasks =
19         new CircularArray(logCapacity+1);
20     for (int i = top; i < bottom; i++) {
21       newTasks.put(i, get(i));
22     }
23     return newTasks;
24   }
25 }
```

图16-12 UnboundedDEQueue类：循环的任务数组

```
1  public class UnboundedDEQueue {
2    private final static int LOG_CAPACITY = 4;
3    private volatile CircularArray tasks;
4    volatile int bottom;
5    AtomicInteger top;
6    public UnboundedDEQueue(int LOG_CAPACITY) {
7      tasks = new CircularArray(LOG_CAPACITY);
8      top = new AtomicInteger(0);
9      bottom = 0;
10   }
11   boolean isEmpty() {
12     int localTop = top.get();
13     int localBottom = bottom;
14     return (localBottom <= localTop);
15   }
16
17   public void pushBottom(Runnable r) {
18     int oldBottom = bottom;
19     int oldTop = top.get();
20     int size = oldBottom - oldTop;
21     if (size >= currentTasks.capacity()-1) {
22       currentTasks = currentTasks.resize(oldBottom, oldTop);
23       tasks = currentTasks;
24     }
25     tasks.put(oldBottom, r);
26     bottom = oldBottom + 1;
27   }
```

图16-13 UnboundedDEQueue类：域、构造函数、pushBottom()和isEmpty()方法

使用时间戳来避免ABA问题。当两个方法竞争最后一个任务时，都是通过增加top来窃取该任务。为了将UnboundedDEQueue重新设置为空，只需简单地将bottom域增加为与top相同即可。在代码中，紧接在第25行的compareAndSet()之后的popBottom()，无论该compareAndSet()成功与否，都将bottom设置为top+1，这为即使它失败了，一个并发的偷窃者也必定已经窃取了最后一个任务，将top+1存入bottom能使top和bottom相等，从而将UnboundedDEQueue对象重新设置为空。

```
1   public Runnable popTop() {
2     int oldTop = top.get();
3     int newTop = oldTop + 1;
4     int oldBottom = bottom;
5     int size = oldBottom - oldTop;
6     if (size <= 0) return null;
7     Runnable r = tasks.get(oldTop);
8     if (top.compareAndSet(oldTop, newTop))
9       return r;
10    return null;
11  }
12
13  public Runnable popBottom() {
14    bottom--;
15    int oldTop = top.get();
16    int newTop = oldTop + 1;
17    int size = bottom - oldTop;
18    if (size < 0) {
19      bottom = oldTop;
20      return null;
21    }
22    Runnable r = tasks.get(bottom);
23    if (size > 0)
24      return r;
25    if (!top.compareAndSet(oldTop, newTop))
26      r = null;
27    bottom = oldTop + 1;
28    return r;
29  }
```

图16-14 UnboundedDEQueue类：popTop()和popBottom()方法

isEmpty()方法（图16-13）首先读top，然后读bottom，再检查bottom是否小于或等于top（第4行）。操作的次序非常重要，因为top决不会减小，所以，如果一个线程在读top之后再读bottom，且发现bottom不大于top，那么队列确实为空，因为对top的一个并发修改只能增加top。同样的原理也适用于popTop()方法调用。图16-15给出了一个执行实例。

pushBottom()方法（图16-13）和BoundedDEQueue中基本上相同。一个不同之处在于，如果当前的push将会导致超出容量，那么这个方法必须扩大循环数组的容量。另一个不同之处是，popTop()不需要时间戳操作。调整大小的能力是有代价的：每次调用必须读top（第20行），以决定是否有必要调整大小，这有可能导致更多的cache缺失，因为top要被所有的进程修改。我们可以让线程保存top的本地值并用它来计算UnboundedDEQueue对象的大小，从而降低这种开销。一个线程仅仅在超过这个界限的时候读top域，以确定resize()是否必要。即

至共享top的改变使得本地拷贝变为过时，top也不会减小，所以UnboundedDEQueue对象的实际大小只可能比使用局部变量计算出的值小。

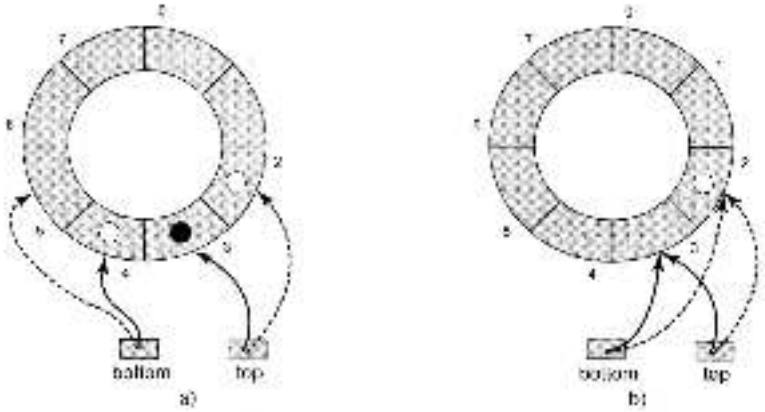


图16-15 UnboundedDEQueue类的实现。在a中，popTop()和popBottom()并发地执行，同时在UnboundedDEQueue对象中至少有一个任务。在b中，只有一个单一的任务。初始时，bottom指向数组项3，top指向3。popBottom()方法首先将bottom从3减为2（用指向数组项2的虚线表示这个改变，因为它马上就要再次改变），然后，当popBottom()检测到最新设置的bottom和top之间的差距为0时，就试图将top增加1（而不是像在BoundedDEQueue中那样将它重设为0）。popTop()方法进行同样的尝试。top或将被它们中的一个所增加，获胜者将得到最后一个任务。最后，popBottom()方法将bottom置回数组项3，它与top相同

总而言之，我们已研究了两种设计可线性化无阻塞DEQueue类的方法。我们可以摆脱在DEQueue的通常操作中只使用加载/存储的方式，但这是以更复杂的算法为代价的。对于某些应用来说是有理由采用这种算法的，例如，执行者池，其性能可能对并发多线程系统起着决定性的作用。

387、388

16.5.3 工作平衡

我们已知在工作窃取算法中，空闲线程从其他线程那里窃取任务。一个可选的方法是，让每个线程随机地选择一个伙伴，并周期性地对其工作负载进行平衡。为了确保较重负载的线程不把精力浪费在负载平衡的尝试中，我们尽量让轻负载的线程来初始化负载平衡。更确切地说，每个线程周期性地投掷硬币，决定是否与其他线程进行平衡。线程进行平衡的概率与该线程队列中的任务数成反比。也就是说，具有较少任务的线程更有可能去重新平衡，而无事可做的线程则必定会进行平衡。线程一律随机地选择一个牺牲者来平衡负载，如只它和牺牲者的负载之间的差异超过一个预先设定的阈值，则迁移任务，直到它们的队列包含同样数目的任务为止。可以证明，该算法提供了很强的公平性保证：每个线程任务队列的预期长度非常接近于平均值。该算法的一个优点是，在每次交换中，平衡操作移动多个任务。第二个优点体现在一个线程的任务比其他线程多很多的时候，特别是当各个任务要求大致相同的计算的时候。在这里给出的工作窃取算法中，当多个线程试图从超负载的线程中窃取不同的任务时，则可能发生争用。

在这种情况下，在工作窃取执行者池中，如果某个线程有很多任务，很可能发生以下情

况：其他的线程将会不断地在同一个本地任务队列上竞争，试图每次最多窃取一个任务。另一方面，在工作共享的执行者池中，一次平衡多个任务则意味着工作将很快在任务中传播开，对每个单独的任务不会产生同步开销。

图16-16描述了一个工作共享的执行者。每个线程有它自己的任务队列，被保存于一个由所有线程共享的数组中（第2行）。每个线程不断地从它的队列中取出下一个任务（第12行）。如果队列为空，那么deq()调用返回*null*，否则，线程运行这个任务（第13行）。此时，线程决定是否进行负载平衡。如果线程的任务队列大小为s，那么线程决定进行负载平衡的概率为$1/(s+1)$（第15行）。为了重新进行平衡，线程一旦随机地选择一个牺牲者线程。该线程以线程ID的次序（为了避免死锁）来两个队列都锁定（第17～20行）。如果队列大小之间的差异超过了阈值，则平衡两个队列的大小（图16-16第27～35行）。 389 ~ 390

```
1  public class WorkSharingThread {
2    Queue[] queue;
3    Random random;
4    private static final int THRESHOLD = ...;
5    public WorkSharingThread(Queue[] myQueue) {
6      queue = myQueue;
7      random = new Random();
8    }
9    public void run() {
10     int me = ThreadID.get();
11     while (true) {
12       Runnable task = queue[me].deq();
13       if (task != null) task.run();
14       int size = queue[me].size();
15       if (random.nextInt(size+1) == size) {
16         int victim = random.nextInt(queue.length);
17         int min = (victim <= me) ? victim : me;
18         int max = (victim <= me) ? me : victim;
19         synchronized (queue[min]) {
20           synchronized (queue[max]) {
21             balance(queue[min], queue[max]);
22           }
23         }
24       }
25     }
26   }
27   private void balance(Queue q0, Queue q1) {
28     Queue qMin = (q0.size() < q1.size()) ? q0 : q1;
29     Queue qMax = (q0.size() < q1.size()) ? q1 : q0;
30     int diff = qMax.size() - qMin.size();
31     if (diff > THRESHOLD)
32       while (qMax.size() > qMin.size())
33         qMin.enq(qMax.deq());
34   }
35 }
```

图16-16 WorkSharingThread类：一个简化的工作共享执行者池

16.6 本章注释

用于多线程计算分析中的基于DAG的模式是由Robert Blumofe和Charles Leiserson[20]提

出的。他们也给出了第一种基于双端队列的工作窃取实现。本章中的一些例子来自于Charles Leiserson和Harald Prokop[102]，无锁的有界双端队列算法是由Anish Arora、Robert Blumofe和Greg Plaxton[15]提出的，该算法中使用的无界时间戳可以用Mark Moir[117]提出的技术变为有界的。无界的双端队列算法是由David Chase和Yossi Lev[28]提出的。定理16.3.1及其证明
391 是由Anish Arora、Robert Blumofe和Greg Plaxton[15]提出并证明的。工作共享的算法是由Larry Rudolph、Tuli Slivkin-Allalouf和Eli Upfal[134]提出的。Anish Arora、Robert Blumofe和Greg Plaxton[15]所提出的算法后来被Danny Hendler和Nir Shavit[56]所改进，增加了在双端队列中窃取一半元素的能力。

16.7 习题

习题185. 考虑下面两部归并排序的代码：

```
void mergeSort(int[] A, int lo, int hi) {
  if (hi > lo) {
    int mid = (hi - lo)/2;
    executor.submit(new mergeSort(A, lo, mid));
    executor.submit(new mergeSort(A, mid+1, hi));
    awaitTermination();
    merge(A, lo, mid, hi);
}
```

假定该排序方法不存在内在的并行，试给出算法的工作、关键路径长度和并行度。请用某个函数f的渐近和$\Theta(f(n))$表示你的答案。

习题186. 假设在一个专用的P处理器机器上，一个并行程序的实际运行时间为：

$$T_P = T_1/P + T_\infty$$

你的研究小组已编制了两个国际象棋程序：一个比较简单而另一个是经过优化的。简单的那个程序的$T_1 = 2048$秒，$T_\infty = 1$秒。当你在一台32个处理器的机器上运行它的时候（肯定是足够的），运行时间是65秒。随后，你的学生们开发优化的版本，其$T_1 = 1024$秒，$T_\infty = 8$秒。为什么它是优化的？当你在32个处理器的机器上运行它的时候，运行时间为40秒，和按照我们的公式预测的一样。

在一台512个处理器的机器上，哪个程序将表现得更好？

习题187. 编写一个ArraySum类，它提供方法：

```
static public int sum(int[] a)
```

该方法采用分治法并行地对数组参数的元素进行求和。

习题188. Jones教授对确定的（确定的）多线程程序进行测试，它是由贪心调度器进行调度的，测试发现$T_2 = 80$秒，$T_{10} = 10$秒。在10个处理器上运行表现的计算时，最快可能是多少？使用下面的
392 不等式及其蕴含的界限来推导出你的结论。注意，P是处理器的数目。

$$T_P \geq \frac{T_1}{P}$$

$$T_P \geq T_\infty$$

$$T_P \leq \frac{(T_1 - T_\infty)}{P} + T_\infty$$

（在贪心调度中最后一个不等式成立。）

习题189. 试给出本章中Matrix类的一种实现。保证你的split()方法需要常数时间。

习题190. 设$P(x)=\sum_{i=0}^{d} p_i x^i$，$Q(x)=\sum_{i=0}^{d} q_i x^i$是$d$次多项式，其中$d$是2的幂。我们可以写成

$$P(x) = P_0(x) + (P_1(x) \cdot x^{d/2})$$

$$Q(x) = Q_0(x) + (Q_1(x) \cdot x^{d/2})$$

其中，$P_0(x)$，$P_1(x)$，$Q_0(x)$和$Q_1(x)$是$d/2$次多项式。

Polynomial类如图16-17所示，它提供了put()和get()方法以访问系数，并且提供了一个常数时间的split()方法，能将一个d次多项式$P(x)$分解为两个如上式所示的$d/2$次多项式$P_0(x)$和$P_1(x)$，其中，分解后的多项式可从原多项式得出，反之亦然。

```
1  public class Polynomial {
2    int[] coefficients; // possibly shared by several polynomials
3    int first;  // index of my constant coefficient
4    int degree; // number of coefficients that are mine
5    public Polynomial(int d) {
6      coefficients = new int[d];
7      degree = d;
8      first = 0;
9    }
10   private Polynomial(int[] myCoefficients, int myFirst, int myDegree) {
11     coefficients = myCoefficients;
12     first = myFirst;
13     degree = myDegree;
14   }
15   public int get(int index) {
16     return coefficients[first + index];
17   }
18   public void set(int index, int value) {
19     coefficients[first + index] = value;
20   }
21   public int getDegree() {
22     return degree;
23   }
24   public Polynomial[] split() {
25     Polynomial[] result = new Polynomial[2];
26     int newDegree = degree / 2;
27     result[0] = new Polynomial(coefficients, first, newDegree);
28     result[1] = new Polynomial(coefficients, first + newDegree, newDegree);
29     return result;
30   }
31 }
```

图16-17 Polynomial类

你的任务是为这个Polynomial类设计并行求和和求积算法。

1. $P(x)$和$Q(x)$的和可以分解为下式：

$$P(x) + Q(x) = (P_0(x) + Q_0(x)) + (P_1(x) + Q_1(x)) \cdot x^{d/2}$$

a) 使用这种分解以图16-13的方式构造一个基于任务的并发多项式求和算法。

b) 计算该算法的工作和关键路径长度。

2. $P(x)$和$Q(x)$的乘积可以分解为下式：

$$P(x) \cdot Q(x) = (P_0(x) \cdot Q_0(x)) + (P_0(x) \cdot Q_1(x) + P_1(x) \cdot Q_0(x)) \cdot x^{d/2} + (P_1(x) \cdot Q_1(x))$$

a) 使用这种分解以图16-4的方式构造一个基于任务的并发多项式求和算法。

b) 计算该算法的工作和关键路径长度。

习题191. 试给出一个有效且高并发度的多线程算法，它通过一个长度为n的向量x来进行$n \times n$矩阵A的乘法，并满足工作为$\Theta(n^2)$，关键路径长度为$\Theta(\log n)$。分析你的实现的工作和关键路径长度，并给出并行度。

393 ~ 394

习题192. 图16-18给出了平衡两个工作队列负载的另一种方式：首先，锁定较小的队列，然后锁定较大的，如果它们的差超过了阈值，则重新平衡。这个代码哪里出错了？

```
1     Queue cMin = (q0.size() < q1.size()) ? q0 : q1;
2     Queue cMax = (q0.size() < q1.size()) ? q1 : q0;
3     synchronized (cMin) {
4       synchronized (qMax) {
5         int diff = qMax.size() - qMin.size();
6         if (diff > THRESHOLD)
7           while(qMax.size()>qMin.size)
8             qMin.enq(cMax.deq());
9       }
10    }
```

图16-18 另一种平衡代码

习题193.

1. 在图16-10的popBotton()方法中，bottom域是volatile的，以确保在popBottom()中，第15行的减小是立即可见的。试描述一个场景，解释如果bottom没有声明为volatile的，会出现什么错误。
2. 为什么在popBottom()方法中，我们应该尽早地尝试将bottom域重新设为0？哪一行是可以安全地进行重新设置的最早的行？我们的BoundedDEQueue会溢出吗？描述如何溢出的。

习题194. 在popTop()中，如果第8行的compareAndSet()成功，那么它将返回恰好在成功的compareAndSet()操作之前它所读取的元素。为什么要在执行compareAndSet()之前从数组中读取元素？

我们可以在popTop()的第6行使用isEmpty()吗？

习题195. UnboundedDEQueue方法的可线性化点是哪里？试证明你的结论。

习题196. 修改可线性化的BoundedDEQueue实现中的popTop()方法，使得仅当队列中没有任务时返回null。注意，可能要用阻塞来实现。

习题197. 你认为在执行者池代码中，BoundedDEQueue的isEmpty()方法调用实际上会提高它的性能吗？

395

障　碍

17.1　引言

假设你正在为电脑游戏编写图形显示功能。在你的程序中准备了一系列要被图形包（可能是硬件协处理器）显示的帧。有时称这种程序为软实时应用：之所以称为实时的，是因为每秒必须至少显示35帧才是有效的，而软的则是因为偶尔发生的较晚并不会带来灾难性后果。在一台单线程机器上，可以编写如下形式的循环：

```
while (true) {
  frame.prepare();
  frame.display();
}
```

然而，如果有n个可用的并行线程，那么可以将帧划分成n个不相交的部分，而让每个线程以和其他线程并行的方式各自准备自己的部分。

```
int me = ThreadID.get();
while (true) {
  frame[me].prepare();
  frame[me].display();
}
```

采用这种方式所存在的问题就是，不同的线程需要不同的时间来准备和显示自己的部分。某些线程可能开始显示第i帧，而其他线程却还未显示完第$(i-1)$帧。

为了避免这种同步问题，我们可以将计算组织成一系列的阶段，在其他线程未完成第$(i-1)$阶段之前，任何线程不能开始第i个阶段。我们以前已经见过这种分阶段式的计算模式。在第12章，排序网算法要求每个比较阶段要和其他的阶段相隔开，类似地，在采样排序算法中，每个阶段在继续推进之前必须确定前驱阶段已经完成。

396
|
397

```
1  public interface Barrier {
2    public void await();
3  }
```

图17-1　障碍接口

实施这种同步机制称为障碍或路障（如图17-1所示）。障碍是一种强制异步线程就好像是同步的一样进行执行的方法。如果一个完成了阶段的线程调用障碍的await()方法，它将被阻塞直到所有n个线程都已完成这个阶段为止。图17-2表示如何采用障碍来使并行绘制程序正确地工作。在准备好帧i之后，所有的线程在开始显示该帧之前在障碍处同步。这种结构确保所有并发显示一个帧的线程能显示出同一个帧。

```
1  private Barrier b;
2  ...
3  while (true) {
4    frame[my].prepare();
5    b.await();
6    frame[my].display();
7  }
```

图17-2　使用障碍同步并发显示

障碍的实现引发了许多与第7章中自旋锁同样的性能问题，同时还有一些新问题。显然，障碍应该很快，也就是说要最小化最后一个到达障碍和最后一个离开障碍的线程之间的时间

间隔，线程应该在差不多相同的时刻离开障碍也是很重要的。线程的通知时间则是指某个线程探测到所有线程都已到达障碍和这个特定线程离开障碍之间的时间间隔。对于大多数软实时应用来说，只有统一的通知时间是很重要的。例如，如果帧的所有部分能在差不多相同的时间更新，那么图片的质量将会提高。

17.2 障碍实现

398 图17-3描述了SimpleBarrier类，它创建了一个AtomicInteger计数器，初始化为n，说明了障碍的大小。每个线程调用getAndDecrement()来减小计数器值，如果调用返回1（第10行），该线程则是最后一个到达障碍的，所以它重置计数器以便下次使用（第11行）。否则，线程在计数器上自旋，等待这个值降为零（第13行）。该策略表看起来好像可以工作，但它无法正确执行。

```
1  public class SimpleBarrier implements Barrier {
2    AtomicInteger count;
3    int size;
4    public SimpleBarrier(int n){
5      count = new AtomicInteger(n);
6      size = n;
7    }
8    public void await() {
9      int position = count.getAndDecrement();
10     if (position == 1) {
11       count.set(size);
12     } else {
13       while (count.get() != 0){};
14     }
15   }
16 }
```

图17-3 SimpleBarrier类

不幸的是，如果这个障碍被使用超过了一次，那么这种简单的设计就不能正常工作（如图17-2所示）。假设有两个线程，线程*A*对计数器调用getAndDecrement()，发现它不是最后一个到达障碍的线程，于是自旋等待计数器值降到零。当*B*到达时，发现自己是最后一个到达的线程，于是重设计数器值*n*为2，它完成下一个阶段并调用await()。与此同时，*A*继续自旋，且计数器值从未到达零。最后，*A*一直在等待阶段0完成，而*B*在等待阶段1完成，两个线程都出现饥饿现象。

解决这个问题的最简单办法就是交替使用两个障碍，一个用于奇数阶段，一个用于偶数阶段。然而这种方法会浪费空间，并且要从应用中获得太多的记录。

17.3 语义换向障碍

语义换向障碍是解决障碍重用问题的一种比较实用而巧妙的方案。如图17-4所示，阶段的语义是一个布尔值：对于偶数的阶段其值为*true*，奇数的阶段其值为*false*。每个SenseBarrier对象都有一个布尔型的sense域，用来表明当前正在执行阶段的语义。每个线程都将它当前的语义作为线程本地对象保存起来（见编程提示17.3.1）。初始时，障碍的sense是所有线程局部语义的补码。
399 当一个线程调用await()时，则检查它是否是递减计数器值的最后一个线程。如果是，则将障碍的语义反向并继续执行。否则，它自旋等待障碍的sense域改变为与它自己的局部语义相匹配。

```
1  public SenseBarrier(int n) {
2    count = new AtomicInteger(n);
3    size = n;
4    sense = false;
5    threadSense = new ThreadLocal<Boolean>() {
6      protected Boolean initialValue() { return !sense; };
7    };
8  }
9  public void await() {
10   boolean mySense = threadSense.get();
11   int position = count.getAndDecrement();
12   if (position == 1) {
13     count.set(size);
14     sense = mySense;
15   } else {
16     while (sense != mySense) {}
17   }
18   threadSense.set(!mySense);
19 }
```

图17-4 SenseBarrier类：语义换向障碍

对共享计数器值的递减有可能导致内存争用，因为所有的线程可能在同一时刻访问计数器。一旦计数器值已被减小，每个线程则在sense域上自旋。这种实现非常适合于缓存一致的系统结构，因为线程在域的本地缓存拷贝上自旋，且该域只在线程准备离开障碍时才被修改。sense域是一种在缓存一致的对称多处理器上保持统一的通知时间的很好的方法。

编程提示17.3.1 在图17-4中，语义换向障碍的构造函数代码是非常直观的。第5行和第6行有点复杂，这里要初始化线程本地的threadSense域。这个精显复杂的语法定义了一个线程本地的布尔值，其初始值是sense域初始值的补码。附录A.2.4给出了Java中线程本地对象的完整解释。

17.4 组合树障碍

减少内存争用（以增加时延为代价）的一种办法就是使用第12章的组合范例。将一个大的障碍分解为由较小的障碍组成的树，让线程沿着树向上组合请求，并沿着树向下分布通知。如图17-5所示，树障碍有大小n和基数r，其中n为线程的总数，r为每个结点的孩子数。为方便起见，我们假设有$n = r^{d+1}$个线程，其中d是树的深度。 400

```
1  public class TreeBarrier implements Barrier {
2    int radix;
3    Node[] leaf;
4    ThreadLocal<Boolean> threadSense;
5    ...
6    public void await() {
7      int me = ThreadID.get();
8      Node myLeaf = leaf[me / radix];
9      myLeaf.await();
10   }
11   ...
12 }
```

图17-5 TreeBarrier类：每个线程找到叶结点数组的索引，并调用那个叶子的await()方法

组合树障碍可以看做是由结点组成的树，像语义换向障碍一样，每个结点具有一个计数器和一个语义。图17-6表示一个结点的实现。线程从叶结点[//r]开始，该结点的await()方法和语义换向障碍的await()方法相类似。主要的区别在于最后一个到达的线程（完成障碍的线程）在唤醒其他线程之前访问父障碍。当r个线程都到达根时，障碍结束且反向语义。就像以前一样，线程本地的布尔型语义位允许重用障碍而不必再初始化。

```
1   private class Node {
2     AtomicInteger count;
3     Node parent;
4     volatile boolean sense;
5     public Node() {
6       sense = false;
7       parent = null;
8       count = new AtomicInteger(radix);
9     }
10    public Node(Node myParent) {
11      this();
12      parent = myParent;
13    }
14    public void await() {
15      boolean mySense = threadSense.get();
16      int position = count.getAndDecrement();
17      if (position == 1) { // I'm last
18        if (parent != null) { // Am I root?
19          parent.await();
20        }
21        count.set(radix);
22        sense = mySense;
23      } else {
24        while (sense != mySense) {};
25      }
26      threadSense.set(!mySense);
27    }
28  }
29 }
```

图17-6 TreeBarrier类：内部的树结点

对结构的障碍通过将内存的访问分散在多个障碍上来减少内存争用。它是否能减少延迟则取决于它减小单个单元快还是访问对数个障碍快。

一旦障碍结束，根结点让通知沿着树向下过滤。这种方法适于NUMA系统结构，但有可能导致不统一的通知时间，因为当线程向上移动时，将访问一系列不可预测的单元。所以这种方法在无缓存的NUMA系统结构中有可能效果并不理想。

编程提示17.4.1 对结点被声明为树障碍类的内部类，所以结点在类的外部是不能访问的。如图17-7所示，树通过调用的build()方法来初始化。该方法以父结点和深度为参数。如果深度不为零，则构建基数个儿子，并递归地构建儿子的儿子。如果深度为零，则将每个结点放入leaf[]数组中。当一个线程进入障碍时，它使用这个数组来选择一个开始的叶结点。附录A.2.1给出了Java中为何类的完整讨论。

```
1  public class TreeBarrier implements Barrier {
2    int radix;
3    Node[] leaf;
4    int leaves;
5    ThreadLocal<Boolean> threadSense;
6    public TreeBarrier(int n, int r) {
7      radix = r;
8      leaves = 0;
9      leaf = new Node[n / r];
10     int depth = 0;
11     threadSense = new ThreadLocal<Boolean>() {
12       protected Boolean initialValue() { return true; };
13     };
14     // compute tree depth
15     while (n > 1) {
16       depth++;
17       n = n / r;
18     }
19     Node root = new Node();
20     build(root, depth - 1);
21   }
22   // recursive tree constructor
23   void build(Node parent, int depth) {
24     if (depth == 0) {
25       leaf[leaves++] = parent;
26     } else {
27       for (int i = 0; i < radix; i++) {
28         Node child = new Node(parent);
29         build(child, depth - 1);
30       }
31     }
32   }
33   ...
34 }
```

图17-7 TreeBarrier类：初始化组合树障碍。build()方法为每个结点创建r个儿子，然后递归地创建儿子的儿子，在最底层，将叶子放入数组中

17.5 静态树障碍

到现在为止所介绍的障碍或者存在着内存争用（简单障碍和语义换向障碍），或者具有大量通信（组合障碍）。在后两种障碍中，线程追访了一系列不可预测的结点，这使得在无缓存的NUMA系统结构上设计障碍非常困难。令人惊奇的是，存在另一种简单的障碍，它既允许静态设计又具有低争用特性。

例17-8中的静态树障碍按如下方式工作：每个线程被指定到树中的一个结点（图17-9）。每个结点等待，直到树中比它低的所有结点都完成为止，然后通知它的父结点。接着自旋等待全局的语义位被改变。一旦根结点获知它的子结点都已完成，就触发全局语义位，通知等待的线程所有的线程都已经完成。在缓存一致的多处理器上，完成这种障碍需要在树中向上移动log(n)步，而通知只需简单地改变全局语义，这由缓存一致的机制来传播。在不具有缓存一致性的机器上，线程将采用前面所学的组合障碍的方式，沿着树向下传播通知。

401
|
403

```
1  public class StaticTreeBarrier implements Barrier {
2    int radix;
3    boolean sense;
4    Node[] node;
5    ThreadLocal<Boolean> threadSense;
6    int nodes;
7    public StaticTreeBarrier(int size, int myRadix) {
8      radix = myRadix;
9      nodes = 0;
10     node = new Node[size];
11     int depth = 0;
12     while (size > 1) {
13       depth++;
14       size = size / radix;
15     }
16     build(null, depth);
17     sense = false;
18     threadSense = new ThreadLocal<Boolean>() {
19       protected Boolean initialValue() { return !sense; };
20     };
21   }
22   // recursive tree constructor
23   void build(Node parent, int depth) {
24     if (depth == 0) {
25       node[nodes++] = new Node(parent, 0);
26     } else {
27       Node myNode = new Node(parent, radix);
28       node[nodes++] = myNode;
29       for (int i = 0; i < radix; i++) {
30         build(myNode, depth - 1);
31       }
32     }
33   }
34   public void await() {
35     node[ThreadID.get()].await();
36   }
37 }
```

图17-8 StaticTreeBarrier类，每个线程索引到一个静态指定的树结点，并访问该结点的await()方法

```
1      public Node(Node myParent, int count) {
2        children = count;
3        childCount = new AtomicInteger(count);
4        parent = myParent;
5      }
6      public void await() {
7        boolean mySense = threadSense.get();
8        while (childCount.get() > 0) {};
9        childCount.set(children);
10       if (parent != null) {
11         parent.childDone();
12         while (sense != mySense) {};
13       } else {
14         sense = !sense;
15       }
16       threadSense.set(!mySense);
17     }
18     public void childDone() {
19       childCount.getAndDecrement();
20     }
```

图17-9 StaticTreeBarrier类，内部结点类

17.6 终止检测障碍

迄今为止所讨论的所有障碍都是按阶段组织计算的，线程完成一个阶段的工作，到达障碍，然后开始一个新的阶段。

然而，存在着另外一类有趣的程序，其中每个线程完成它自己的计算部分，仅当其他线程产生新的任务时才继续工作。第16章中简化的工作窃取执行者池就是一个这样的例子（图17-10）。在这里，一旦线程用完它的本地队列中的任务，就尝试从其他线程的队列中窃取工作。execute()方法本身可以将新任务放到调用线程的本地队列中。一旦所有线程用完了它们队列中的任务，这些线程就一直运行，不断地尝试窃取元素。但是，我们更愿意设计一种终止检测障碍，从而使得一旦这些线程完成了所有的任务，它们都能终止。

```
1  public class WorkStealingThread {
2    DEQueue[] queue;
3    int size;
4    Random random;
5    public WorkStealingThread(int n) {
6      queue = new DEQueue[n];
7      size = n;
8      random = new Random();
9      for (int i = 0; i < n; i++) {
10       queue[i] = new DEQueue();
11     }
12   }
13   public void run() {
14     int me = ThreadID.get();
15     Runnable task = queue[me].popBottom();
16     while (true) {
17       while (task != null) {
18         task.run();
19         task = queue[me].popBottom();
20       }
21       while (task == null) {
22         int victim = random.nextInt() % size;
23         if (!queue[victim].isEmpty()) {
24           task = queue[victim].popTop();
25         }
26       }
27     }
28   }
29 }
```

图17-10 再探工作窃取执行者池

每个线程或者是活动的（有一个任务要执行）或者是非活动的（没有任务可执行）。注意，只要有某个线程是活动的，那么任何非活动的线程都可能变成活动的，因为非活动线程可以从活动线程那里窃取任务。一旦所有的线程都变成非活动的，那么任何线程就不能再回到活动状态。所以，检测一个计算作为一个整体是否已终止就是要及时判断在某个瞬间是否不再存在任何活动的线程。

至今所学的任何一个障碍算法都不能解决这个问题。因为线程可能不断地在活动和非活动状态之间转换，所以不能通过让每个线程声明它已变成非活动的并简单地计算这样的线程的数量来检测终止。例如，考虑如图17-10所示的线程A、B和C，假设每个线程都有一个布尔

值来说明它处于活动状态还是非活动状态。当A变成非活动状态时，它可能接着观察到B是非活动的，又观察到C也是非活动的。然而，A却不能得出整个计算都已经完成的结论，因为在A检查完B但还未检查完C之前，B有可能从C窃取了任务。

404 - 405

终止检测障碍（图17-11）提供了setActive(*v*)和isTerminated()方法。每个线程在变为活动时，调用setActive(*true*)通知障碍；当变为非活动时，调用setActive(*false*)通知障碍。当且仅当所有线程在先前的某个时刻变为非活动状态时，isTerminated()方法返回*true*。图17-12描述了终止检测障碍的一种简单实现。

```
1  public interface TDBarrier {
2    void setActive(boolean state);
3    boolean isTerminated();
4  }
```

图17-11 终止检测障碍接口

```
1  public class SimpleTDBarrier implements TDBarrier {
2    AtomicInteger count;
3    public SimpleTDBarrier(int n){
4      count = new AtomicInteger(n);
5    }
6    public void setActive(boolean active) {
7      if (active) {
8        count.getAndIncrement();
9      } else {
10       count.getAndDecrement();
11     }
12   }
13   public boolean isTerminated() {
14     return count.get() == 0;
15   }
16 }
```

图17-12 简单终止检测障碍

该障碍包含一个AtomicInteger，初始化为0。每个变为活动状态的线程递减计数器值（第8行），每个变为非活动状态的线程递增计数器值（第10行）。当计数器值变为零时，则认为计算终止了（第14行）。

406

终止检测障碍只有在正确地使用时才能正常工作。图17-13描述了当计算已终止时，如何修改工作窃取线程的run()方法来返回。初始时，每个线程设为活动的（第3行）。一旦一个线程用完它的本地队列中的任务，就改为非活动的（第10行）。然而，在它试图窃取一个新任务之前，必须改为活动的（第14行）。如果窃取失败，则再改为非活动的（第17行）。

注意，线程在窃取任务前要设置为活动的。否则的话，如果一个线程打算要窃取一个任务而它是非活动的，那么被窃取了任务的线程也可能声明自己是非活动的，从而导致所有线程都声明自己是非活动的，而计算又在连续执行。

下面是设计中很巧妙的一个地方，线程在试图窃取任务之前要测试队列是否为空（第13行）。这样，如果窃取不成功，则可以避免声明自己是活动的。若没有这个措施，线程有可能无法检测到终止，因为在窃取者尝试注定不会成功的偷窃之前，每个线程会不断地切换为活动状态。

```
1   public void run() {
2     int me = ThreadID.get();
3     tdBarrier.setActive(true);
4     Runnable task = queue[me].popBottom();
5     while (true) {
6       while (task != null) {
7         task.run();
8         task = queue[me].popBottom();
9       }
10      tdBarrier.setActive(false);
11      while (task == null) {
12        int victim = random.nextInt() % queue.length;
13        if (!queue[victim].isEmpty()) {
14          tdBarrier.setActive(true);
15          task = queue[victim].popTop();
16          if (task == null) {
17            tdBarrier.setActive(false);
18          }
19        }
20        if (tdBarrier.isTerminated()) {
21          return;
22        }
23      }
24    }
25  }
26 }
```

图17-13 工作窃取执行者池：具有终止的run()方法

终止检测障碍的正确使用必须满足安全性和活性。安全性是指如果isTerminated()返回*true*，那么计算的确已经终止。安全性要求活动的线程不能声明自己是非活动的，因为它可以触发一个不正确的终止检测。例如，对于图17-13中的工作窃取线程，如果该线程只在成功地窃取一个任务之后才声明自己是活动的，那么该线程将是不正确的。相反，一个不活动的线程声称自己是活动的则是安全的，如果一个线程在第15行窃取工作时不成功，则会出现这样的情形。

活性特性是指如果计算终止，那么isTerminated()最终会返回*true*。（不要求终止应被释时检测。）如果一个非活动状态的线程声明自己是活动的，则安全性并没受到损害，而如果一个窃取失败的线程没有再次声明自己是非活动的（第15行），则会违背活性特性，因为当这种情况发生时终止将无法被检测到。

17.7 本章注释

John Mellor-Crummey和Michael Scott[113]给出了障碍算法的综述，当然，应从历史的角度去看待他们的结论。组合树障碍是在John Mellor-Crummey和Michael Scott[113]的代码的基础上设计的，而它们又是建立在Pen-Chung Yew、Nian-Feng Tzeng和Duncan Lawrie[151]的组合树算法之上的。分散式障碍归功于Debra Hensgen、Raphael Finkel和Udi Manber [59]。习题中用到的竞赛树障碍归功于John Mellor-Crummey和Michael Scott [113]。简单障碍和静态树障碍最为广泛流传。我们是从Beng-Hong Lim那里了解到静态树障碍的。终止检测障碍及其在执行者池中的应用是以Peter Kessler对Dave Detlefs、Christine Flood、Nir Shavit和Xiolan Zhang[41]算法进行改进后的算法为基础的。

17.8 习题

习题198. 图17-14表示如何在异步系统结构上使用障碍进行并行前缀计算。

```
1  class Prefix extends java.lang.Thread {
2    private int[] a;
3    private int i;
4    public Prefix(int[] myA, int myI) {
5      a = myA;
6      i = myI;
7    }
8    public void run() {
9      int d = 1, sum = 0;
10     while (d < n) {
11       if (i >= d)
12         sum = a[i-d];
13       if (i >= d)
14         a[i] += sum;
15       d = d * 2;
16     }
17   }
18 }
```

图17-14 并行前缀计算

给定一个数字序列$a_0,\cdots,a_{n-1}$，并行前缀计算则并行地计算部分和：

$$b_i=\sum_{j=0}^{i} a_j$$

在一个异步系统中，所有的线程同时执行操作，有一些著名的简单算法，能让n个线程在$\log n$步内计算部分和。该计算从循环0开始，以一序列循环向前推进。在循环r中，如果$i\geq 2^r$，线程i则将$a[i-2^r]$中的值读到本地变量中。然后，将该值加到$a[i]$中。循环继续直到$2^r\geq n$为止。不难看出，经过$\log_2(n)$次循环后，数组a中包含了所有的部分和。

1. 如果在$n>m$个线程上执行并行前缀，会出现什么错误？
2. 修改这个程序，增加一个或多个障碍，使得它在并发地设置n个线程时可以正常工作。所需的最小障碍个数为多少？

409 习题199. 修改语义转向障碍的实现，使得等待的线程调用wait()而不是自旋。

- 给出一种挂起线程要比让线程自旋好的情形。
- 给出一种其他选择要更好的情形。

习题200. 修改树障碍的实现，使得它可以接受一个Runnable对象，该对象的run()方法在最后一个线程到达障碍后且所有的线程还未离开障碍之前被调用一次。

习题201. 修改组合树障碍，使得结点可以使用任何障碍实现，而不是只能使用语义转向障碍。

习题202. 竞赛树障碍（图17-15中的TourBarrier类）是树结构障碍的一种变化形式。假设有n个线程，其中n是2的整数次幂。该树是一个由$2n-1$个结点组成的二叉树。每个叶子由一个静态决定的单个线程所拥有。每个结点的两个儿子被链接成失斧，其中一个被静态地设计为主动的，另一个则为被动的。图17-16描述了这种树结构。

```
1   private class Node {
2    volatile boolean flag;   // signal when done
3    boolean active;          // active or passive?
4    Node parent;             // parent node
5    Node partner;            // partner node
6    // create passive node
7    Node() {
8      flag   = false;
9      active = false;
10     partner = null;
11     parent = null;
12   }
13   // create active node
14   Node(Node myParent) {
15     this();
16     parent = myParent;
17     active = true;
18   }
19   void await(boolean sense) {
20     if (active) { // I'm active
21       while (flag != sense) {};                    // wait for partner
22       if (parent != null) {parent.await(sense);}   // wait for parent
23       partner.flag = sense;                        // tell partner
24       } else {                      // I'm passive
25         partner.flag = sense;       // tell partner
26         while (flag != sense) {};   // wait for partner
27     }
28   }
29 }
```

图17-15 TourBarrier类

410

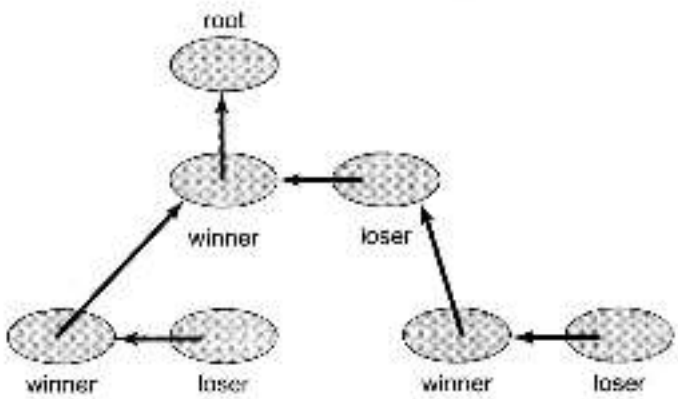


图17-16 TourBarrier类：信息流。结点被静态地结对为主动/被动。线程从叶结点开始。每个主动结点中的线程等待其被动的伙伴出现，然后继续向树的上方走进。每个被动线程等待其主动伙伴完成的通知。一旦主动线程到达根，那么所有的线程已到达。通知则以相反的次序沿着树向下流

每个线程在一个线程本地变量中保存当前的语义。当一线程到达一个被动结点时，它将其主动伙伴的sense域设置为当前语义，然后在它自己的sense域上自旋直到它的伙伴将这个域的值变为当前语义为止。当一线程到达一个主动结点时，就在它自己的sense域上自旋，直到它的被动伙伴将其设置为当前语义为止。当这个域改变时，那个特定的障碍被完成，主动的线程沿

着父结点的引用到达它的父结点。注意在一个层上的主动线程有可能在下一个层变成被动的。当根结点障碍完成时，则沿着树向下过滤通知，每个线程向下返回，将它的伙伴的sense域设置为当前语义。

这种障碍对图17-5的组合树障碍进行了一点改进。下面解释其原因。

竞赛障碍代码使用parent和partner引用来引导整个树，我们可以通过去掉这些域并将所有的结点保存在一个单独的数组中来节省空间，树的根索引为0，根的儿子索引为1和2，孙子的索引为3－6，如此类推。请采用索引算法而不用引用来引导树，重新实现竞赛障碍。

习题203. 组合树障碍对整个障碍使用了一个单独的线程本地的语义域。假设我们采用图17-17的设计给每个结点关联一个线程本地的语义域。那么，请完成下面问题：

- 或者解释为什么这种实现除了需要更多内存以外，等价于原来的实现。
- 或者给出一个反例说明这种实现是不正确的。

```
private class Node {
  AtomicInteger count;
  Node parent;
  volatile boolean sense;
  int d;
  // construct root node
  public Node() {
    sense = false;
    parent = null;
    count = new AtomicInteger(radix);
    ThreadLocal<Boolean> threadSense;
    threadSense = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return true; };
  };
  }
  public Node(Node myParent) {
    this();
    parent = myParent;
  }
  public void await() {
    boolean mySense = threadSense.get();
    int position = count.getAndDecrement();
    if (position == 1) {     // I'm last
      if (parent != null) { // root?
        parent.await();
      }
      count.set(radix);   // reset counter
      sense = mySense;
    } else {
      while (sense != mySense) {};
    }
    threadSense.set(!mySense);
  }
}
```

图17-17 线程本地的树障碍

习题204. 树障碍“自底向上”工作，也就是说障碍的完成是从树叶到树根，而唤醒信息则是从树根返回到树叶。图17-18和图17-19描述了另一种设计，称为**翻转树障碍**，除了障碍的完成是从树根开始移到树叶之外，其工作方式与树障碍很相似。

- 或者简要说明为什么这种实现是正确的，可能要对标准的树障碍进行归约。

- 或者给出反例说明为什么不对。

```
1  public class RevBarrier implements Barrier {
2    int radix;
3    ThreadLocal<Boolean> threadSense;
4    int leaves;
5    Node[] leaf;
6    public RevBarrier(int mySize, int myRadix) {
7      radix = myRadix;
8      leaves = 0;
9      leaf = new Node[mySize / myRadix];
10     int depth = 0;
11     threadSense = new ThreadLocal<Boolean>() {
12       protected Boolean initialValue() { return true; }
13     };
14     // compute tree depth
15     while (mySize > 1) {
16       depth++;
17       mySize = mySize / myRadix;
18     }
19     Node root = new Node();
20     root.d = depth;
21     build(root, depth - 1);
22   }
23   // recursive tree constructor
24   void build(Node parent, int depth) {
25     // are we at a leaf node?
26     if (depth == 0) {
27       leaf[leaves++] = parent;
28     } else {
29       for (int i = 0; i < radix; i++) {
30         Node child = new Node(parent);
31         child.d = depth;
32         build(child, depth - 1);
33       }
34     }
35   }
```

图17-18 翻转树障碍部分1

习题205. 使用一个n线程计数器和单个的布尔变量，实现一种可重用的n线程障碍。给出这个障碍能正常工作的证明。

习题206. 你能为执行者池设计一个"分布式"的终止检测算法吗？其中，线程不是反复地更新或测试一个集中的单元来检测终止，而是只使用非平凡的本地变量。变量可能是无界的，但状态的改变应是常数时间（所以你不能并行化这个共享计数器）。

提示：采用第4章的原子快照算法。

习题207. 分支算术是一种对称障碍实现，其中线程只使用加载/存储操作在静态指派的本地级存单元上自旋。如图17-20所示，该算法以一系列循环来运行。在循环r，线程i通知线程$i+2^r \pmod n$（其中n是线程个数）并等待线程$i-2^r \pmod n$的通知。

412
413

采用这种协议运行来实现一个障碍需要多少次循环？如果n不是2的整数次幂将会怎样？证明你的答案。

```
36    public void await() {
37      int me = ThreadInfo.getIndex();
38      Node myLeaf = leaf[me / radix];
39      myLeaf.await(me);
40    }
41    private class Node {
42      AtomicInteger count;
43      Node parent;
44      volatile boolean sense;
45      int d;
46      // construct root node
47      public Node() {
48        sense = false;
49        parent = null;
50        count = new AtomicInteger(radix);
51      }
52      public Node(Node myParent) {
53        this();
54        parent = myParent;
55      }
56      public void await(int me) {
57        boolean mySense = threadSense.get();
58        // visit parent first
59        if ((me % radix) == 0) {
60          if (parent != null) { // root?
61            parent.await(me / radix);
62          }
63        }
64        int position = count.getAndDecrement();
65        if (position == 1) {  // I'm last
66          count.set(radix);   // reset counter
67          sense = mySense;
68        } else {
69          while (sense != mySense) {};
70        }
71        threadSense.set(!mySense);
72      }
73    }
74  }
```

图17-19 弹转树障碍部分2：是否正确

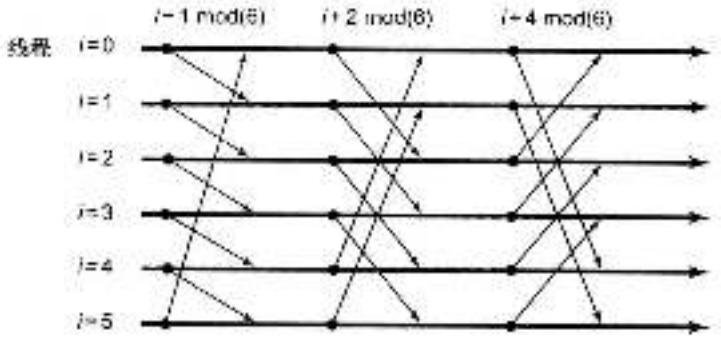


图17-20 分发障碍中的通信。在每个循环r中，线程i与线程(i+2^r) (mod n) 进行通信

习题208. 实现Java语言给出一种可重用分发障碍的实现。

提示：可能要保存当前阶段的奇偶性和语义域。

习题209. 创建一张表，能够汇总静态树、组合树和分发障碍的操作总数。

习题210. 终止检测障碍中，在窃取任务之前状态被设置为活动的；否则，偷窃者线程不能声明为非活动的，然后，它窃取一个任务，在将它的状态设置为活动状态之前，被窃取任务的线程可能变为非活动的。这将导致我们不希望发生的情形，所有的线程都声明为非活动的，然而计算仍在继续进行。你能设计一种可终止的执行者池，其中状态只有在成功地窃取了一个任务之后才被设置为活动的吗？

414
415

第18章

The Art of Multiprocessor Programming (Revised First Edition)

事务内存

18.1 引言

讨论完数据结构和算法的设计，本章评论解决这些问题所使用的工具。它们都是当今系统结构所提供的同步原语（包括各种类型的上锁、自旋和阻塞），诸如compareAndSet()操作以及一些相关的原子操作。这些操作大多都能提供很好的服务，多处理器程序设计员已经能构造出多种实用精妙的数据结构。然而，这些工具也是存在缺陷的，本章将回顾并分析标准同步原语的优缺点，同时阐述一些新出现的解决方案，它们能扩展甚至取代现今的许多标准原语。

18.1.1 关于锁的问题

作为同步规范的锁对于缺乏经验的程序员来说存在着很多缺陷。当低优先级的线程被抢占，而它又持有高优先级线程所需要的锁时，会出现优先级倒置现象。若由于页故障或其他中断而耗尽了一个持有锁的线程的调度量，使得该线程不再被调度时，将会发生转让现象。当持有锁的线程处于非活动状态时，其他请求这个锁的线程将要排队等待而不能继续前进。甚至当锁被释放后，要将队列清空也需要花费一段时间，这与年残骸已被清除的情形下，事故仍可能会造成流量降低是一样的。如果线程试图以不同的次序锁定同一个对象，则会出现死锁。如果线程必须锁定很多对象，尤其是在对象集预先不可知的情况下，避免死锁是非常困难的。过去，高扩展性的应用不仅很少而且很珍贵，对于这样的一些问题我们可以通过组织一组专业的程序员来避免。而如今，高扩展性的应用变得非常普遍，采用传统的方法则需要太高的成本。

416
↓
417

产生这种问题的关键就是没有人真正知道如何组织和维护依赖于锁的大型系统。数据和锁之间的关联通常是按照约定来建立的，它最终只存在于程序员的脑海中，或者以注释的形式作为文档被保存。图18-1是一个Linux头文件[1]中的典型注释，它描述了使用某个特定缓冲区的规范。随着时间的推移，对这种形式所书写的规范进行观察和解释则可能使代码的维护变得非常复杂。

```
/*
 * When a locked buffer is visible to the I/O layer BH_Launder
 * is set. This means before unlocking we must clear BH_Launder,
 * mb() on alpha and then clear BH_Lock, so no reader can see
 * BH_Launder set on an unlocked buffer and then risk to deadlock.
 */
```

图18-1 传统的同步：Linux内核中的一个典型注释

[1] 内核v2.4.19 /fs/buffer.c。

18.1.2 关于compareAndSet()的问题

避开上锁的方法之一就是使用类似于compareAndSet()的操作原语，但在通常情况下，采用compareAndSet()及其相关操作进行算法设计非常难，有时还会带来较高的开销。其主要的难点就在于几乎所有的同步原语，包括读、写或调用原子的compareAndSet()，都只是对单个字进行操作。这种限制往往导致在算法设计中要引入复杂而不寻常的数据结构。

我们来分析第10章中的无锁队列（图18-2中重新给出），这里着重研究底层的同步原语。

```
1  public class LockFreeQueue<T> {
2    private AtomicReference<Node> head;
3    private AtomicReference<Node> tail;
4    ...
5    public void enq(T item) {
6      Node node = new Node(item);
7      while (true) {
8        Node last = tail.get();
9        Node next = last.next.get();
10       if (last == tail.get()) {
11         if (next == null) {
12           if (last.next.compareAndSet(next, node)) {
13             tail.compareAndSet(last, node);
14             return;
15           }
16         } else {
17           tail.compareAndSet(last, next);
18         }
19       }
20     }
21   }
22 }
```

图18-2 LockFreeQueue类：enq()方法

在第12行和第13行之间会出现复杂的情形。enq()方法调用compareAndSet()，改变tail结点的next域并让tail域本身指向新结点。理想情况下，我们希望能原子地组合两个compareAndSet()调用，但因为这些调用是一次一个地出现，所以无论enq()还是deq()都有可能会遇到enq()完成了一半的情形（第12行）。解决这个问题的一种办法就是引入multiCompareAndSet()原语，如图18-3所示。该方法的参数分别是：由对象所组成的数组AtomicReference<T>、由期望值组成的类型T的数组以及用于更新的由类型T的值所组成的数组。该方法对所有的数组元素同时执行compareAndSet()。如果其中的任何一个失败，则全部失败。更具体地说，对于所有的*i*，如果target[*i*]与expected[*i*]的值相同，则让target[*i*]的值等于update[*i*]的值，并返回*true*；否则，不改变target[*i*]，返回*false*。

注意，通常的系统结构中并没有直接实现multiCompareAndSet()的办法，我们现在假设存在这样的方法，那么比较图18-2和图18-4中LockFreeQueue()的实现，可以发现multiCompareAndSet()方法简化了并发数据结构。图18-2中第11－12行的复杂逻辑被一个multiCompareAndSet()方法所替换。

尽管像multiCompareAndSet()这种多字扩展的方式非常有用，但它并不能解决另外一个严重问题，这将在18.1.3节中讨论。

```
1  <T> boolean multiCompareAndSet(
2        AtomicReference<T>[] target,
3        T[] expected,
4        T[] update) {
5    atomic {
6      for (int i = 0; i < target.length)
7        if (!target[i].get().equals(expected[i].get()))
8          return false;
9      for (int i = 0; i < target.length)
10       target[i].set(update[i].get);
11     return true;
12   }
13 }
```

图18-3 multiCompareAndSet()的伪代码。该代码原子地执行

```
1  public void enq(T item) {
2    Node node = new Node(item);
3    while (true) {
4      Node last = tail.get();
5      Node next = last.next.get();
6      if (last == tail.get()) {
7        AtomicReference[] target = {last.next, tail};
8        T[] expect = {next, last};
9        T[] update = {node, node};
10       if (multiCompareAndSet(target, expect, update)) return;
11     }
12   }
13 }
```

图18-4 LockFreeQueue类：采用multiCompareAndSet()简化的enq()方法

18.1.3 关于复合性的问题

至今为止所讨论的同步机制，无论是有锁的还是无锁的，都存在一个关键的缺点：不能简单地进行合成（复合）。假设要从队列q0中出队一个元素x，并将它入队到另一个队列q1中。这种转移必须是原子的：任何并发线程都不能观察到x在两个队列中同时消失或出现。在基于管程的Queue实现中，因为每个方法都要在内部获得锁，所以按照这种方式合成两个方法调用是根本不可能的。

无法合成不仅仅局限于互斥。我们考虑一个有界队列类，每当队列为空时这个类的deq()方法就会被阻塞（使用wait/notify或显式的条件对象）。假设有两个队列，我们要从两个队列中的任一个出队一个元素，如果这两个队列都为空，则希望阻塞直到任意一个队列中出现一个元素为止。在基于管程的Queue实现中，因为每个方法都提供了它自己的条件等待，所以按照这种方式在两个条件上等待是根本不可能的。

显然，总会存在一些特殊的解决方案。对于这种原子的转移，我们可以引入一个锁，任何试图对q0和q1进行原子修改的线程都要获得这个锁。但是，这种锁将会变成一个并发瓶颈（没有并发转移），而且要求事先知道两个队列的标识。也可以让队列本身输出它们的同步状态（比如说，通过lock()和unlock()方法），并依赖调用者来管理多对象的同步。然而通过这种方式展现同步状态将会破坏程序的模块性，复杂化接口，且要求调用者遵守复杂的约定。况且，这种简单的方法不能用于非阻塞队列的实现。

418
~
420

18.1.4 我们能做什么

下面总结常规的同步原语所存在的问题：

- 难以有效地管理锁，尤其是在大型系统中。
- 类似于compareAndSet()这样的原语一次只能对一个字进行操作，导致算法复杂。
- 很难将多个对象的多个调用组合成一个原子单位。

18.2节将会引入事务内存的概念，这是一种为解决上述问题而提出的新型程序设计模型。

18.2 事务和原子性

事务是单个线程所执行的一系列操作步骤。事务必须是可串行化的，这意味着它们看起来应像是按照一次一个的次序顺序地执行。可串行化是可线性化的一种粗粒度版本。可线性化定义了单个对象的原子性，它要求一个给定对象的每次方法调用看起来就像是在调用和响应之间的某个瞬时起作用的，而可串行化则定义了所有事务的原子性，也就是说，代码块中可能包含多个对象的调用。这样，能够确保一个事务看起来就像是在它的第一个调用和最后一个调用的响应之间生效的[1]。在正确的实现中，事务不会出现死锁或者活锁。

我们现在描述一种在Java上进行扩展后的简单程序设计语言，它能支持同步的事务模型。这些扩展目前并不是Java的组成部分，但可以用它们来说明模型。这里所描述的特性是当前事务内存系统所提供的常规特性。并不是所有的系统都提供全部的特性；有一些提供较弱的保证，而有一些则提供较强的保证。然而，理解这些特性对于理解现代事务内存模型大有帮助。

关键字atomic能对事务进行定界，这和用关键字synchronized对临界区进行定界几乎是一样的。当synchronized块获得一个特定的锁时，它只是对其他获得同一个锁的synchronized块是原子的，而一个atomic块则对所有的atomic块是原子的。嵌套的synchronized块如果按照相反的次序来获得锁则会发生死锁，而嵌套的atomic块却不会。 421

因为事务允许原子地修改多个单元，所以不再需要multiCompareAndSet()。图18-5为事务队列的enq()方法。我们把这段代码与图18-2的无锁代码进行比较：这里不需要AtomicReference域、CompareAndSet()调用和重试循环。在这里，代码实质上是由atomic块括起来的顺序代码。

```
1  public class TransactionalQueue<T> {
2    private Node head;
3    private Node tail;
4    public TransactionalQueue() {
5      Node sentinel = new Node(null);
6      head = sentinel;
7      tail = sentinel;
8    }
9    public void enq(T item) {
10     atomic {
11       Node node = new Node(item);
12       tail.next = node;
13       tail = node;
14     }
15   }
```

图18-5 无界的事务队列：enq()方法

要说明如何用事务来写并发程序，首先要说明它们是如何实现的。事务总是试探性地(speculatively)进行执行：当一事务执行时，它对对象进行暂时的(tentative)改变。如果它没有遇到同步冲突就能执行完，则事务提交(暂时的改变变成永久性的)，否则该事务终止(暂时的改变被放弃)。

事务可以嵌套。事务的嵌套必须按照简单的模块方式：一个方法可以启动一个事务，然后再调用另一个方法，但不用关心这个嵌套的调用是否启动了一个事务。如果要求一个嵌套

[1] 在一些文献中，可串行化定义不要求事务按照与实时优先次序中兼容的次序来串行化。

事务能够终止但却不终止它的父事务，这时嵌套事务就特别有用。在后面讨论条件同步时，这个性质特别重要。

回忆一下，将一个元素从一个队列原子地移到另一个队列，这对于使用内部管理锁的对象而言实质上是不可能的。而采用事务合成这种原子的方法调用却是非常容易的。图18-6描述了如何合成从队列q0中出队元素x的deq()调用和入队x到队列q1的enq(x)调用。

```
1    atomic {
2      x = q0.deq();
3      q1.enq(x);
4    }
```

图18-6 合成原子方法调用

那么条件同步将会怎样呢？列18-7为用于有界缓冲区的enq()方法。该方法进入atomic块（第2行），测试缓冲区是否满（第3行）。如果是满的，则调用retry（第4行）回滚封装的事务。首先终止事务，当该对象状态已经改变时则重启事务。条件同步是要求只回滚嵌套事务而不回滚父事务的原因之一，因为这种方式对条件同步非常方便。与wait()方法或显式的条件变量不同，retry并不是简单地让出它自己，从而丢失唤醒故障。

422

```
1    public void enq(T x) {
2      atomic {
3        if (count == items.length)
4          retry;
5        items[tail] = x;
6        if (++tail == items.length)
7          tail = 0;
8        ++count;
9      }
10   }
```

图18-7 有界事务队列：具有retry的enq()方法

回忆一下，在使用具有内部管理条件变量的对象时，等待若干个条件中的一个变成*true*是不可能的。retry的一个新颖之处就是使这种合成变得非常容易。图18-8是说明orElse语句的代码段，它可以连接两个或多个代码块。在这里，线程先执行第一个块（第2行）。如果这个块调用了retry，则该子事务回滚，线程执行第二个块（第4行）。如果第二个块也调用了retry，那么orElse作为一个整体被终止，然后返回执行每个块（当发生某些改变时），直到有一个块完成为止。

```
1    atomic {
2      x = q0.deq();
3    } orElse {
4      x = q1.deq();
5    }
```

图18-8 orElse语句：等待多重条件

在本章余下的部分里，我们研究事务内存的实现技术。事务同步可以用硬件实现（HTM），也可以用软件实现（STM），或综合使用两者来实现。下面章节将介绍STM的实现。

423

18.3 软事务内存

遗憾的是，目前并不提供对18.2节所描述语言的支持。因此，本节将介绍如何使用软件库来支持事务同步。首先介绍TinySTM，这是一种简单的软事务内存包，它也是18.2节所描述语言的扩展目标。为简单起见，我们不考虑类似于嵌套事务、retry和orElse等重要的问题。软事务内存的构造应包含两个方面的因素：运行事务的线程以及它们所访问的对象。

我们通过对并发SkipList（类似于第14章的跳表）的部分实现进行分析来阐述这些概念。该类采用跳表实现能提供常用方法的集合：add(*x*)将*x*添加到集合，remove(*x*)从集合中删除*x*，当且仅当*x*属于该集合时contains(*x*)返回*true*。

回忆一下，跳表是一个链表的集合，链表中的每个结点都包含一个item域（集合中的一个元素）、一个key域（元素的哈希码）和一个next域（next域是由指向链表中后继结点的引用所组成的数组）。数组槽0指向链表中最近的下一个结点，编号越高的数组槽则按序指向越

常后的后继结点。要找到给定的关键字，首先从较高的层进行查找，若查找不命中，则转向较低的层。这种方式能够在链表长度的对数时间内找到一个元素。（参阅第14章，了解跳表的详细描述。）

在TinyTM中，线程通过共享的原子对象进行通信，这种对象提供了同步，能确保事务不会看到其他未提交事务的效应，并能从被已终止事务的影响来进行恢复。原子对象的域是不能直接访问的，而只能通过getter和setter方法间接地访问。例如，key域的getter方法的形式为：

```
int getKey();
```

而相应的setter方法的形式则为：

```
void setKey(int value);
```

通过getter和setter对域进行访问，可以提供在每次域访问中插入事务同步和恢复的能力。图18-9描述了SkipNode的完整接口。

```
1  public interface SkipNode<T> {
2    public int getKey();
3    public void setKey(int value);
4    public T getItem();
5    public void setItem(T value);
6    public AtomicArray<SkipNode<T>> getNext();
7    public void setNext(AtomicArray<SkipNode<T>> value);
8  }
```

图18-9 SkipNode 接口

SkipList的事务实现同样也不能使用标准数组，因为TinyTM无法截获对数组的访问。相反，TinyTM提供了AtomicArray<T>类，它支持与规则数组相同的功能。

图18-10给出SkipNodeSet类的域和构造函数，图18-11给出add()方法的代码。除了因getter和setter所引起的语法复杂化以外，这个代码与顺序实现的代码几乎完全一样。（getter和setter调用可以由编译器或预处理器生成，但在这里，我们让调用是显式的。）第21行创建了一个新的实现了SkipNode接口的TSkipNode（事务的跳表结点），我们将在后面介绍这个类。

424

```
1
2  public final class SkipListSet<T> {
3    final SkipNode<T> head;
4    final SkipNode<T> tail;
5    public SkipListSet() {
6      head = new TSkipNode<T>(MAX_HEIGHT, Integer.MIN_VALUE, null);
7      tail = new TSkipNode<T>(0, Integer.MAX_VALUE, null);
8      AtomicArray<SkipNode<T>> next = head.getNext();
9      for (int i = 0; i < next.length; i++) {
10       next.set(i, tail);
11     }
12   }
13   ...
```

图18-10 SkipListSet类：域和构造函数

返回一个T类型值的事务是通过Callable<T>对象实现的（见第16章），要被执行的代码被封装到该对象的call()方法中。

```
14    public boolean add(T v) {
15      int topLevel = randomLevel();
16      SkipNode<T>[] preds = (SkipNode<T>[]) new SkipNode[MAX_HEIGHT];
17      SkipNode<T>[] succs = (SkipNode<T>[]) new SkipNode[MAX_HEIGHT];
18      if (find(v, preds, succs) != -1) {
19        return false;
20      }
21      SkipNode<T> newNode = new TSkipNode<T>(topLevel+1, v);
22      for (int level = 0; level <= topLevel; level++) {
23        newNode.getNext().set(level, succs[level]);
24        preds[level].getNext().set(level, newNode);
25      }
26      return true;
27    }
```

425

图18-11 SkipListSet类：add()方法

事务线程（TThread 类）是一种能运行事务的线程。TThread（图18-12）通过调用以Callable<T>对象作为参数的doIt()方法来运行事务。图18-13是一个事务线程将一系列值插入到一个跳表树的代码段。变量list是由多个事务线程所共享的跳表。这里，调用doIt()的参数是一个匿名内部类，这是一种Java构造，它允许以内联方式声明短期类，result变量是一个布尔值，用于说明这个值是否已经存在于链表中。

```
1   public class TThread extends java.lang.Thread {
2     static Runnable onAbort = ...;
3     static Runnable onCommit = ...;
4     static Callable<Boolean> onValidate = ...;
5     public static <T> T doIt(Callable<T> xaction) throws Exception {
6       T result = null;
7       while (true) {
8         Transaction me = new Transaction();
9         Transaction.setLocal(me);
10        try {
11          result = xaction.call();
12        } catch (AbortedException e) {
13        } catch (Exception e) {
14          throw new PanicException(e);
15        }
16        if (onValidate.call()) {
17          if (me.commit()) {
18            onCommit.run(); return result;
19          }
20        }
21        me.abort();
22        onAbort.run();
23      }
24    }
25  }
```

图18-12 TThread类

下面详细地说明TinyTM的实现。在18.3.1节中，描述事务线程的实现，然后再描述如何实现原子的事务对象。

```
1  SkipListSet<Integer> list = new SkipListSet<Integer>();
2  for (int i = 0; i < 100; i++) {
3      result = TThread.doIt( new Callable<Boolean>() {
4                                  public Boolean call() {
5                                      return list.add(i);
6                                  }
7                              });
8  }
```

426

图18-13 添加元素到事数链表

18.3.1 事务和事务线程

事务的状态封装在一个线程本地的Transaction对象中（见图18-14），该对象有三种状态：ACTIVE，ABORTED和COMMITTED（第2行）。当创建一个事务时，它的默认状态为ACTIVE（第11行）。为方便起见，对那些当前不在一个事务中执行的线程定义一个固定的Transaction，COMMITTED事务对象（第3行）。Transaction类还要使用一个线程本地域local来记录每个线程的当前事务（第5~8行）。

```
1  public class Transaction {
2    public enum Status {ABORTED, ACTIVE, COMMITTED};
3    public static Transaction COMMITTED = new Transaction(Status.COMMITTED);
4    private final AtomicReference<Status> status;
5    static ThreadLocal<Transaction> local = new ThreadLocal<Transaction>() {
6      protected Transaction initialValue() {
7        return new Transaction(Status.COMMITTED);
8      }
9    };
10   public Transaction() {
11     status = new AtomicReference<Status>(Status.ACTIVE);
12   }
13   private Transaction(Transaction.Status myStatus) {
14     status = new AtomicReference<Status>(myStatus);
15   }
16   public Status getStatus() {
17     return status.get();
18   }
19   public boolean commit() {
20     return status.compareAndSet(Status.ACTIVE, Status.COMMITTED);
21   }
22   public boolean abort() {
23     return status.compareAndSet(Status.ACTIVE, Status.ABORTED);
24   }
25   public static Transaction getLocal() {
26     return local.get();
27   }
28   public static void setLocal(Transaction transaction) {
29     local.set(transaction);
30   }
31 }
```

图18-14 Transaction类

commit()方法尝试着将事务状态从ACTIVE改为COMMITTED（第19行），abort()方法将事务状态从ACTIVE改为ABORTED（第22行）。线程可以通过调用getStatus()来测试它的当前事 427

务状态（第16行）。如果线程发现它当前的事务已终止，则抛出AbortedException异常。线程可以通过调用静态的getLocal()和setLocal()方法来获得和设置它的当前事务。

TThread类（事务线程）是标准Java的Thread类的一个子类（图18-12）。每个事务线程都有几个相关的处理程序。当事务提交或终止时，会调用onCommit和onAbort处理程序。当事务准备提交时，会调用确认处理程序，它返回一个布尔值，以说明线程的当前事务是否应该尝试提交。这些处理程序可以在运行时定义，后面我们将会看到如何使用这些处理程序来实现不同的事务同步和恢复技术。

doIt()方法（第5行）以Callable<T>对象作为输入，并将它的call()方法作为一个事务来执行。它创建一个新的ACTIVE事务（第8行），然后调用这个事务的call()方法。如果该方法抛出AbortedException异常（第12行），那么doIt()方法简单地重试这个循环。任何其他的异常都意味着应用已出错（第13行），（为简单起见）方法将抛出PanicException，打印出错信息并停止所有程序。如果事务返回，那么doIt()调用确认处理程序来测试是否准备提交（第16行）。如果确认成功，则尝试提交事务（第17行）。如果提交成功，则运行提交处理程序并返回（第18行）。否则，如果确认失败，则显式地终止这个事务。不论任何原因导致提交失败，则在重试之前运行终止处理程序（第22行）。

18.3.2 僵尸事务和一致性

同步冲突会导致事务终止，但在冲突发生之后并不一定能立刻终止事务的线程。相反，即使这种僵尸[⊖]事务已不能提交，但它们仍可能继续运行。这种情况导致了另外一个重要的设计问题：如何防止僵尸事务看到不一致的状态。

下面解释为什么会出现不一致状态。一个对象有两个域x和y，初始时分别为1和2。每个事务都保持着不变式y总是等于$2x$。事务Z读y，看到其值为2。事务A将x和y的值分别改为2和4，并提交。Z现在变为僵尸，尽管它仍在运行，但绝不可能提交。Z稍后读y，值为2，这与它对x读的值不一致。

一种解决办法就是认为这种不一致状态无关紧要。因为僵尸事务最终一定会终止，它们的修改会被丢弃，所以为什么要去关心它们看到什么呢？不幸的是，即使僵尸事务的更新不会产生影响，但仍然会引起一些问题。在前面的场景中，每个一致的状态都有$y=2x$，但是Z已读到不一致的x和y（值都为2），那么如果Z要计算表达式

$$1/(x-y)$$

它就会抛出一个被零除的“不可能”异常，从而导致线程终止，甚至可能使应用崩溃。同样的原因，如果Z现在要执行循环

```
int i = x + 1;  // i is 3
while (i++ != y) {  // y is actually 2, should be 4
  ...
}
```

那么它将永远都不会停止。

在无法依赖不变式的程序设计模型中，不存在避免“不可能”异常和无限的循环的可行方法。所以，TinyTM保证所有的事务，甚至是僵尸事务，都会看到一致的状态。

⊖ 僵尸是一个翻动的尸体，僵尸的故事起源于加勒比黑人的伏都教精神信仰。

18.3.3 原子对象

正如前面所讲的，并发事务通过共享的原子对象进行通信。我们已知（图18-9），对原子对象的访问是通过一个典型的接口来实现的，该接口提供一组匹配的getter和setter方法。图18-15描述了AtomicObject接口。

```
1  public abstract class AtomicObject <T extends Copyable<T>> {
2    protected Class<T> internalClass;
3    protected T internalInit;
4    public AtomicObject(T init) {
5      internalInit = init;
6      internalClass = (Class<T>) init.getClass();
7    }
8    public abstract T openRead();
9    public abstract T openWrite();
10   public abstract boolean validate();
11 }
```

图18-15 抽象类AtomicObject<T>

下面将构造两种类来实现这个接口：一种是顺序实现，不提供同步或恢复；另一种是事务实现，提供同步和恢复。在这里，这两种类同样也可以通过编译器来简单地生成，但是，我们将采用手动方式来实现它们的构造。

顺序实现是很简单的。对于每个匹配对getter-setter，如：

429

```
T getItem();
void setItem(T value);
```

顺序实现都定义了一个类型为T的私有域item，另外，我们还要求顺序实现满足简单的Copyable<T>接口，它提供copyTo()方法来将一个对象的域复制到另外一个对象（图18-16）。由于技术原因，这个类型还应提供一个不带参数的构造函数。为简单起见，我们使用术语版本（version）来表示原子对象接口的顺序的、Copyable<T>实现的一个实例。

```
1  public interface Copyable<T> {
2    void copyTo(T target);
3  }
```

图18-16 Copyable<T>接口

图18-17为SSkipNode类，它是SkipNode接口的一种顺序实现。该类有三个部分。首先，必须提供一个为原子对象实现所使用的无参构造函数（稍后描述），也可以提供其他基于该类实现的构造函数。其次，应提供由接口定义的getter和setter，其中每个getter和setter只是简单地读/写它的相关域。最后，还要实现Copyable接口，提供能用另一个对象的域来初始化一个对象域的copyTo()方法。需要这样一个方法是为了备份顺序对象的副本。

18.3.4 如何演进

事务内存的目标之一就是让程序员不必担心饥饿、死锁以及上锁所具有的"肉体之苦"。但是，实现STM的事务内存必须决定应该满足什么样的演进条件。

回顾第3章可知，满足强不相关演进条件的实现（如无等待或无锁），能保证线程总会向前推进。然而，尽管可以设计出无等待或无锁的STM系统，但没有人知道如何使它们变得高效实用。

对非阻塞STM的研究主要是针对弱相关演进条件来开展的。有两种途径能够保证较好的性能：无干扰的STM（也是无阻塞的）和基于锁的阻塞式STM（也是无死锁的）。就像其他非阻塞条件

一样，无干扰性能保证不是所有的线程都能被其他线程的延迟或故障所阻塞。这种特性比无锁同步要弱一些，因为当两个或两个以上的冲突线程并发执行时，它不能保证程序继续向前推进。

```
1  public class SSkipNode<I>
2   implements SkipNode<T>, Copyable<SSkipNode<T>> {
3    AtomicArray<SkipNode<T>> next;
4    int key;
5    T item;
6    public SSkipNode() {}
7    public SSkipNode(int level) {
8      next = new AtomicArray<SkipNode<I>>(SkipNode.class, level);
9    }
10   public SSkipNode(int level, int myKey, T myItem) {
11     this(level); key = myKey; item = myItem;
12   }
13   public AtomicArray<SkipNode<T>> getNext() {return next;}
14   public void setNext(AtomicArray<SkipNode<I>> value) {next = value;}
15   public int getKey() {return key;}
16   public void setKey(int value) {key = value;}
17   public T getItem() {return item;}
18   public void setItem(T value) {item = value;}
19
20   public void copyTo(SSkipNode<T> target) {
21     target.next = next;
22     target.key  = key;
23     target.item = item;
24   }
25 }
```

图18-17 SSkipNode类：顺序的SkipNode实现

如果线程在临界区内发生中断，无死锁特性并不能保证继续演进。幸运的是，和我们早先学过的大多数基于锁的数据结构一样，现代操作系统的调度程序能够最小化线程在事务中间被调出的概率。就像无干扰一样，在两个或多个冲突线程并发执行时，无死锁特性不能保证程序继续向前推进。

在无阻塞的无干扰和阻塞的无死锁STM中，冲突事务的演进是由争用管理器来保证的，这是一种决定何时延迟争用线程的机制，它通过自旋或屈从使某个线程总是能够前进。

18.3.5 争用管理器

和大多数其他的STM一样，在TinySTM中，一个事务能够检测出它将在何时引起一个同步冲突。然后，请求者事务向争用管理器进行询问。争用管理器的作用与古希腊神谕（oracle）[⊖]的作用一样，通知这个事务是否立即终止另一个事务，或者让自己停下来给另一个线程完成的机会。显然，没有事务会永远停止来等待另一个事务。

430
|
431

图18-18描述了一种简化的争用管理器基类。它提供了单一的resolve()（第12行）方法，该方法以两个事务（请求者事务和另一个事务）作为输入，或者暂时中止请求者事务或者终止另一个事务。该方法同时也通过getLocal()和setLocal()（第16行和第13行）跟踪每个线程的本地争用管理器（第2行）。

ContentionManager类是抽象的，因为它并没有实现任何冲突解决策略。下面给出一些可

⊖ 追溯到公元前1400年，德尔菲预言家——希腊神谕提要对农业和战争进行建议和预言。

能的争用管理器策略。假设事务A将与事务B发生冲突。

```
1  public abstract class ContentionManager {
2    static ThreadLocal<ContentionManager> local
3     = new ThreadLocal<ContentionManager>() {
4      protected ContentionManager initialValue() {
5        try {
6          return (ContentionManager) Defaults.MANAGER.newInstance();
7        } catch (Exception ex) {
8          throw new PanicException(ex);
9        }
10     }
11   };
12   public abstract void resolve(Transaction me, Transaction other);
13   public static ContentionManager getLocal() {
14     return local.get();
15   }
16   public static void setLocal(ContentionManager m) {
17     local.set(m);
18   }
19 }
```

图18-18 争用管理器基类

- 后退策略：A不断地后退一个随机的时间间隔，并加倍期望时间直到某个界限。当达到该界限时，A则终止B。
- 优先级策略：每个事务在启动时获得一个时间戳。如果A的时间戳比B的早，那么A终止B，否则A等待。一个终止后重启的事务仍保持它的老时间戳，以保证每个事务最终都能完成。
- 耐心策略：每个事务在启动时获得一个时间戳。如果A的时间戳比B的早，或者B在等待另一个事务，那么A终止B。这种策略消除了等待事务链。和优先级策略一样，每个事务最终都能完成。
- 回报策略：每个事务记录它已完成的工作，完成工作越多的事务优先级越高。

图18-19描述了一种采用后退策略的争用管理器实现。该管理器指定最小和最大的延迟

```
1  public class BackoffManager extends ContentionManager {
2    private static final int MIN_DELAY = ...;
3    private static final int MAX_DELAY = ...;
4    Random random = new Random();
5    Transaction previous = null;
6    int delay = MIN_DELAY;
7    public void resolve(Transaction me, Transaction other) {
8      if (other != previous) {
9        previous = other;
10       delay = MIN_DELAY;
11     }
12     if (delay < MAX_DELAY) {
13       Thread.sleep(random.nextInt(delay));
14       delay = 2 * delay;
15     } else {
16       other.abort();
17       delay = MIN_DELAY;
18     }
19   }
20 }
```

图18-19 简化的争用管理器实现

(第2~3行)。resolve()方法检查是否是第一次遇到另一个线程(第8行)。如果是,则重置它的延迟为最小值。否则使用当前的延迟。如果当前的延迟比最大值小,线程则休眠由延迟值所限定的一个随机时延(第13行),并加倍下一个延迟。如果当前的延迟值超过最大值,则由调用者终止另一个事务(第16行)。

18.3.6 原子对象的实现

可线性化性要求单个方法调用看起来就像是原子地发生。现在考虑如何保证可串行化性:多个原子地调用具有相同的性质。

原子对象的事务实现必须提供getter和setter方法,以调用事务的同步和恢复。回顾同步和恢复的两种办法:FreeObject类是无干扰的,而LockObject类则使用锁来同步。这两个类都是抽象AtomicObject类的实现,如图18-15所示。init()方法以该原子对象的类作为参数,并将其记录下来以便将来使用。openRread()方法返回一个适合读的版本(即只能调用它自己的getter方法),而openWrite()方法则返回一个可以写的版本(即可以调用getter和setter方法)。

当且仅当能够保证返回值是一致的值时,validate()方法才返回*true*。在返回任何从原子对象提取的信息之前,必须调用validate()。openRead()、openWrite()和validate()方法都是抽象的。

432
/
433

图18-20描述了TSkipNode类,这是SkipNode的一种事务实现。这个类使用LockObject原子对象实现来进行同步和恢复(第8行)。

```
1  public class TSkipNode<T> implements SkipNode<T> {
2    AtomicObject<SSkipNode<T>> atomic;
3    public TSkipNode(int level) {
4      atomic = new LockObject<SSkipNode<T>>(new SSkipNode<T>(level));
5    }
6    public TSkipNode(int level, int key, T item){
7      atomic =
8        new LockObject<SSkipNode<T>>(new SSkipNode<T>(level, key, item));
9    }
10   public TSkipNode(int level, T item){
11     atomic = new LockObject<SSkipNode<T>>(new SSkipNode<T>(level,
12         item.hashCode(), item));
13   }
14   public AtomicArray<SkipNode<T>> getNext() {
15     AtomicArray<SkipNode<T>> forward = atomic.openRead().getNext();
16     if (!atomic.validate())
17       throw new AbortedException();
18     return forward;
19   }
20   public void setNext(AtomicArray<SkipNode<T>> value) {
21     atomic.openWrite().setNext(value);
22   }
23   // getKey, setKey, getItem, and setItem omitted ...
24 }
```

图18-20 TSkipNode类:SkipNode的事务实现

该类具有一个AtomicObject<SSkipNode>域。构造函数以SSkipNode对象作为参数来初始化AtomicObject<SSkipNode>域。每个getter执行下面的操作序列。

1. 调用openRead()来提取一个版本。

2. 调用那个版本的getter来提取存放在局部变量中的域值。

3. 调用validate()来确保读到的值是一致的。

最后一个步骤用来确保在第一步和第二步之间对象没有发生改变，且在第二步中所记录的值与事务观察到的实际值是一致的。

setter以同样的方式来实现，但在第二步中调用getter。

现在已介绍了两种原子对象的实现。为简化表述，没有对实现进行相应的优化。

18.3.7 无干扰原子对象

回想一下，如果对于任意的线程，如果它自己运行足够长的时间仍能继续推进，那么这个算法称为无干扰的。在实际中，这种条件表示线程能运行足够长的时间而不与其他并发线程发生冲突，一直在向前推进。下面我们描述AtomicObject的一种无干扰实现。 434

概述

每个对象有三个逻辑域：owner域、old版本和new版本。（之所以称为逻辑域，是因为它们可能不是作为域来实现的。）owner是访问对象的最后一个事务。old版本是owner事务到达之前对象的状态，如果有更新，new版本反映事务的更新。如果owner为COMMITTED，那么new版本是对象的当前状态。若它是ABORTED，那么old版本是对象的当前状态。如果owner为ACITVE，则不存在当前版本，且未来的当前版本取决于owner是提交还是终止。

当一个事务开始时，它创建一个Transaction对象来保存这个事务的状态，初始时为ACTIVE。如果该事务提交了，则由这个事务将状态设置为COMMITTED，如果这个事务被另一个事务终止，那么由另一个事务将其状态设置为ABORTED。

每当事务A访问一个对象时，首先打开那个对象，有可能需要重新设置owner、old版本和new版本值，假设B是这个对象先前的拥有者。

1. 如果B已经COMMITTED，那么new版本就是当前的版本。A将它自己作为对象的当前拥有者，将old版本设置为先前的new版本，将new版本设置为先前new版本的一个拷贝（如果调用是一个setter），或者设置为new版本本身（如果调用是一个getter）。

2. 对称地，如果B已经ABORTED，那么old版本就是当前的版本。A将它自己作为该对象的当前拥有者，将old版本设为先前的old版本，将new版本设置为先前old版本的一个拷贝（如果调用是一个setter），或者设为old版本本身（如果调用是一个getter）。

3. 如果B是ACTIVE的，那么A和B发生冲突，A向争用管理器询问是终止B还是自己暂停以给B完成的机会。若一个事务要终止另一个事务，它可通过成功地调用compareAndSet()来把要被终止事务的状态改为ABORTED。

我们把对这个算法进行扩展以允许多个并发读者的问题留给读者。

打开对象之后，getter将版本的域读入一个局部变量，在返回该值之前，调用validate()检查调用者事务是否没有被中止。如果一切正常，则将域值返回给调用者（setter的工作方式与此相似）。

当A提交时，它调用compareAndSet()将它的状态改为COMMITTED。如果成功，则提交完成。下一个要访问A所拥有对象的事务将会观察到A已经提交，并将对象的new版本（由A设置的）看作是当前的版本。如果失败，则已被另一个事务终止。下一个要访问被A所更新对象的

事务将观察到A已终止，并将对象的old版本（先于A的版本）作为当前的。图18-21给出了一
435 个执行实例。

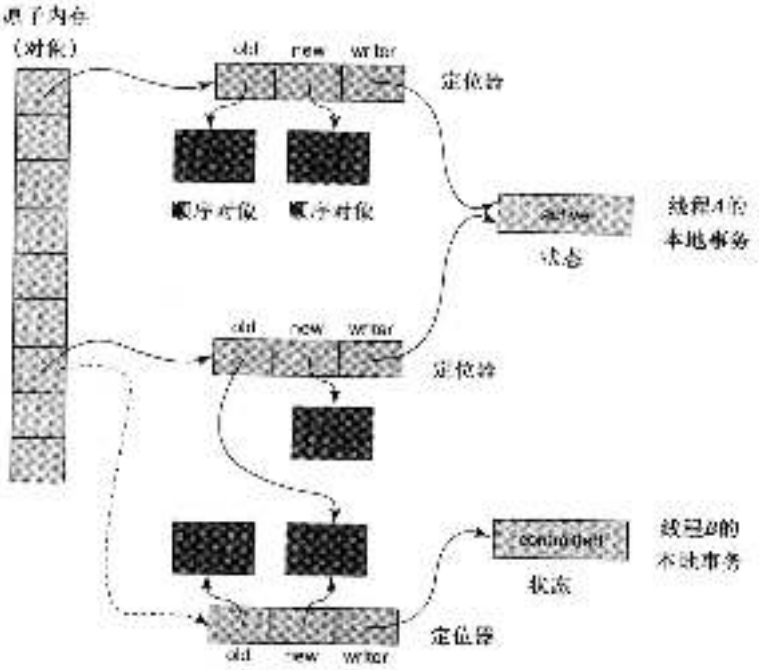


图18-21 FreeObject类：无干扰的原子对象实现。线程A已经完成了对一个对象的写，正在选择为一个最后一次是被线程B所写对象的拷贝。它创建一个新的定位器，该定位器有一个对象的新近拷贝和一个老的对象域，指向线程B的定位器的new域。然后，它使用compareAndSet()来让这个对象指向新创建的定位器

为什么能达到效果

下面说明为什么每个事务都观察到一个一致的状态。当事务A调用一个getter方法读一个对象域时，它打开这个对象，将它自己设置为这个对象的拥有者。如果这个对象已经有一个处于活动状态的拥有者B，那么A终止B。然后，A将域值读入一个局部变量中。然而，在getter将这个值返回给应用之前，它调用validate()来验证这个值是一致的。如果另一个事务C替代A成为任意对象的拥有者，那么C终止A，且A的验证失败。由此可知，如果setter返回一个值，则这个值必是一致的。

下面说明为什么事务是可串行化的。如果一个事务A成功地将它的状态从ACTIVE改为COMMITTED，那么它必定仍是它访问的所有对象的拥有者，因为任何夺走A的拥有权的事务必
436 定已先终止了A。由此可知，自A访问了对象后，它所读或写的对象都没有发生变化，所以A在有效地更新它所访问对象的快照。

详细说明

打开一个对象要求原子地改变多个域，包括修改owner、old版本域和new版本域。在不用锁的情形下，实现这种对多个域进行原子修改的唯一办法就是引入一个间接的中间层。如图18-22所示，FreeObject类具有一个start域，它是一个指向Locator对象的Atomic-

Reference（第3行），保存着该对象的当前事务、老版本和新版本（第5～7行）。

```
1  public class FreeObject<T extends Copyable<T>>
2                              extends TinyTM.AtomicObject<T> {
3    AtomicReference<Locator> start;
4    private class Locator {
5      Transaction owner;
6      T oldVersion;
7      T newVersion;
8      ...
9    }
10   ...
11 }
```

图18-22 FreeObject类：内部Locator类

回忆一下，一个对象的域是按照下面的步骤进行修改的：(1) 调用openWrite()以获得一个对象的版本，(2) 尝试修改这个版本，(3) 调用validate()以确保该版本仍是正确的。图18-23描述了FreeObject类的openWrite()方法。首先，线程检查它自己的事务状态（第14行）。如果状态是已提交的，则该线程没有在事务中运行，直接修改对象（第15行）。如果状

```
12   public T openWrite() {
13     Transaction me = Transaction.getLocal();
14     switch (me.getStatus()) {
15       case COMMITTED: return openSequential();
16       case ABORTED: throw new AbortedException();
17       case ACTIVE:
18         Locator locator = start.get();
19         if (locator.owner == me)
20           return locator.newVersion;
21         Locator newLocator = new Locator();
22         while (!Thread.currentThread().isInterrupted()) {
23           Locator oldLocator = start.get();
24           Transaction owner = oldLocator.owner;
25           switch (owner.getStatus()) {
26             case COMMITTED:
27               newLocator.oldVersion = oldLocator.newVersion;
28               break;
29             case ABORTED:
30               newLocator.oldVersion = oldLocator.oldVersion;
31               break;
32             case ACTIVE:
33               ContentionManager.getLocal().resolve(me, owner);
34               continue;
35           }
36           try {
37             newLocator.newVersion = (T) _class.newInstance();
38           } catch (Exception ex) {throw new PanicException(ex);}
39           newLocator.oldVersion.copyTo(newLocator.newVersion);
40           if (start.compareAndSet(oldLocator, newLocator))
41             return newLocator.newVersion;
42         }
43         me.abort();
44         throw new AbortedException();
45       default: throw new PanicException("Unexpected transaction state");
46     }
47   }
```

图18-23 FreeObject类：openWrite()方法

态是终止的，那么该线程立刻抛出一个AbortedException异常（第16行）。最后，如果事务是活动的，那么线程读取当前的定位器并检查它是否已为写打开了这个对象，如果是，则立即返回（第19行）。否则，它进入循环（第22行），不断地初始化并尝试设置一个新的定位器。为了确定对象的当前值，线程检查最后一个写该对象的事务的状态（第25行）。如果拥有者已提交，就使用新版本（第27行）。如果拥有者是ABORTED（第30行）则使用老版本。如果拥有者仍然是活动的（第30行），则产生了同步冲突，线程调用争用管理器模块来解决这个冲突。在没有冲突的情形下，线程创建并初始化一个新的版本（第37~39行）。最后，线程调用compareAndSet()来用新版本取代老版本，若成功则返回，若失败则重新尝试。

除了不需要产生一个老版本的拷贝以外，openRead()方法（没有给出）的工作方式与此相类似。

FreeObject类的validate()方法（没有给出）简单地确认当前线程的事务状态是否是ACTIVE。

18.3.8 基于锁的原子对象

这种无干扰的实现效率并不是很高，因为写操作会持续地分配定位器和版本，而读操作必须穿过两个间接层（两个引用）才能到达实际要读的数据。在本小节中，我们给出一种更高效的原子对象实现，它使用短临界区来消除定位器并去掉一个间接层。

一个基于锁的STM在读或写时可能会锁定所有对象。然而，大部分应用都遵循80/20规则：
437 约80%的访问是读操作，20%的访问是写操作。对一个对象进行上锁的代价是很高的，因为它
438 要调用compareAndSet()，这在读/写冲突不频繁时显得过于浪费。读操作时锁定对象真的有必要吗？答案是否定的。

概述

基于锁的原子对象实现采用乐观的方式读对象，随后检查冲突。它使用一个全局的版本时钟（version clock）和一个由所有事务共享的计数器进行冲突检查，每当一个事务提交时都要递增这个计数器。当一个事务启动时，它将当前的版本时钟值记录到一个线程本地的读时间戳中。

每个对象都有下面一些域：时间戳（stamp）域，是最后一个对该对象写的事务的读时间戳；版本（version）域，是顺序对象的一个实例；锁（lock）域，是一个锁。正如前面所说的，该顺序类型必须实现Copyable接口，并提供一个无参的构造函数。

事务虚拟地执行一系列访问对象的读写操作。所谓“虚拟地”，是指没有对象被真正地修改。相反，事务使用一个线程本地的读集来记录它所读的对象，使用一个线程本地的写集来记录它要修改的对象以及它们暂定的新版本。

当事务调用getter返回一个域值时，LockObject的openRead()方法首先检查该对象是否已经出现在写集中。如果是，则返回暂定的新版本。否则，它检查对象是否被上锁，如果是，则存在一个同步冲突，该事务终止。如果没有锁定，openRead()方法将该对象添加到读集中并返回它的版本。

openWrite()方法与上述方法相类似。如果对象不在写集中，则创建一个新的暂定的版本，将这个暂定的版本添加到写集中，并返回这个版本。

validate()方法检查对象的时间戳是否不大于事务的读时间戳。如果是，则存在冲突，

该事务终止。否则，这个getter返回在前一步读到的值。

要注意LockObject的validate()方法只能保证值是一致的，并不能保证调用者不是僵尸事务。相反，事务必须按照下面的步骤来提交。

1. 按照任意的次序，锁定它的写集中的每个对象，并采用定时器来避免死锁。

2. 使用compareAndSet()来递增全局的版本时钟，将结果存放在线程本地的写时间戳中。如果该事务提交，则这个时间点就是它被串行化的地方。

3. 事务检查它的读集中的每个对象没有被其他线程锁定，且每个对象的时间戳不大于事务的读时间戳。如果确认成功，事务则提交。（在事务的写时间戳比它的读时间戳大1的情形下，不需要确认读集，因为没有发生并发的修改。）

4. 事务修改它的写集中每个对象的时间戳域。一旦时间戳被修改，事务就释放它的锁。

如果这些测试中的任何一个失败，事务则终止，放弃它的读集和写集并释放它所持有的所有锁。

图18-24描述了一个执行实例。

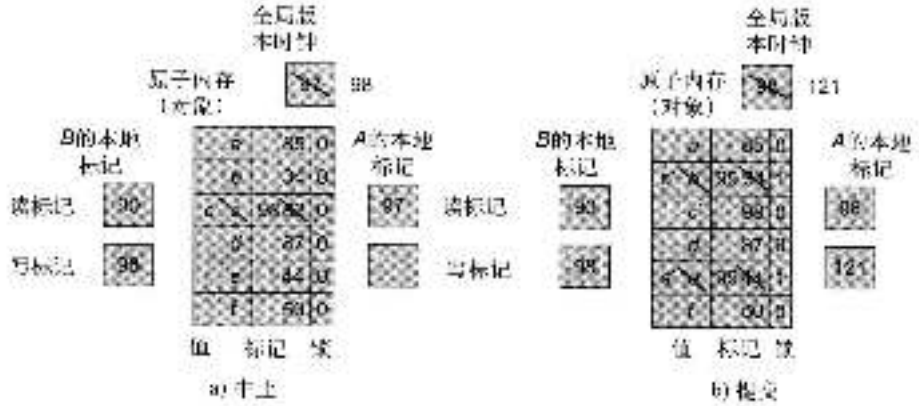


图18-24 LockObject类：基于锁的事务内存实现。在a中，线程A开始它的事务，设置它的读时间戳rs为97，即全局版本时钟值。在A开始读或写对象之前，线程B提交；它增加全局版本时钟为98，将98记录到它的本地写时间戳域ws中，并在成功地确认后用时间戳98写新值c'。（没有给出B对对象锁的请求和释放。）当A读时间戳为98的对象时，它会侦测到线程B的修改，因为它的读时间戳小于98，所以A终止。在b中，A在B完成之后开始它的事务，读到读时间戳值为98，且在读c'时并不终止。A创建读-写集合，递增全局版本时钟。（注意，其他线程已将时钟增加为120。）它锁定它要修改的对象，并成功地确认。然后，基于写时间戳修改这些对象的值和时间戳。在该图中，我们没有给出写对象锁的最后的释放

为什么能达到效果

事务按照它们递增全局版本时钟的次序是可串行化的。下面是每个事务都能观察到一致状态的原因。如果一个读时间戳为r的事务A观察到对象没有被锁定，那么这个版本将具有不超过r的最近的时间戳。任何以后修改对象的事务将锁定这个对象，增加全局版本时钟值，并将这个对象的时间戳设置为新版本的时钟，该值是超过r的。如果A观察到这个对象没有被锁定，那么A不会丢失时间戳小于等于r的修改。A还要通过在读取域之后检查时间戳来确认这个对象的时间戳不超过r。

下面是事务可串行化的原因。我们认为A先读x然后再提交，这时，x可能在A第一次读x和增加全局版本时钟这段时间内没有被改变。正如前面提到的，如果具有读时间戳r的A在时刻t观察到x没有被锁定，那么任何随后对x的修改将会给x一个比r大的时间戳。如果事务B在A之前提交，并修改了一个被A读的对象，那么，A的确认程序或者观察到x被B锁定，或者观察到x的时间戳大于r，在任何一种情形下，都将会终止。

详细说明

在描述算法之前，首先描述基本的数据结构。图18-25给出锁实现中所采用的WriteSet类，该类实质上是从对象到版本的映射，把事务所写的每个对象发送给它的暂定版本。除了get()和set()方法以外，该类还包括对表中每个对象上锁和开锁的方法。ReadSet类（没有给出）只是对象的一个集合。

```
1  public class WriteSet {
2    static ThreadLocal<Map<LockObject<?>, Object>> map
3        = new ThreadLocal<Map<LockObject<?>,Object>>() {
4      protected synchronized Map<LockObject<?>, Object> initialValue() {
5        return new HashMap();
6      }
7    };
8    public static Object get(LockObject<?> x) {
9      return map.get().get(x);
10   }
11   public static void put(LockObject<?> x, Object y) {
12     map.get().put(x, y);
13   }
14   public static boolean tryLock(long timeout, TimeUnit timeUnit) {
15     Stack<LockObject<?>> stack = new Stack<LockObject<?>>();
16     for (LockObject<?> x : map.get().keySet()) {
17       if (!x.tryLock(timeout, timeUnit)) {
18         for (LockObject<?> y : stack) {
19           y.unlock();
20         }
21         throw new AbortedException();
22       }
23     }
24     return true;
25   }
26   public static void unlock() {
27     for (LockObject<?> x : map.get().keySet()) {
28       x.unlock();
29     }
30   }
31   ...
32 }
```

图18-25 LockObject类：内部WriteSet类

图18-26描述了版本时钟。所有域和方法都是静态的。该类管理着一个全局版本计数器和一个线程本地的读时间戳集。getWriteStamp()方法返回当前的全局版本，而setWriteStamp()则将它增加1。getReadStamp()方法返回调用者的线程本地的读时间戳，而setReadStamp()将线程本地的读时间戳设置为当前的全局时钟值。

LockObject类（图18-27）有三个域：对象的锁，它的读时间戳以及对象的实际数据。图18-28显示如何为读操作来打开对象。如果对象的拷贝不在事务的写集中（第13行），那么它将

该对象放入事务的读集。然而，如果该对象被锁定，说明它正处于并发事务进行更新的过程中，那么，终止读者（第15行）。如果该对象在写集中有一个暂定版本，则返回这个版本（第19行）。

```
1  public class VersionClock {
2    // global clock read and advanced by all
3    static AtomicLong global = new AtomicLong();
4    // thread-local cached copy of global clock
5    static ThreadLocal<Long> local = new ThreadLocal<Long>() {
6      protected Long initialValue() {
7        return 0L;
8      }
9    };
10   public static void setReadStamp() {
11     local.set(global.get());
12   }
13   public static long getReadStamp() {
14     return local.get();
15   }
16   public static void setWriteStamp() {
17     local.set(global.incrementAndGet());
18   }
19   public static long getWriteStamp() {
20     return local.get();
21   }
22 }
```

图18-26 VersionClock类

```
1  public class LockObject<T extends Copyable<T>> extends AtomicObject<T> {
2    ReentrantLock lock;
3    volatile long stamp;
4    T version;
5    ...
```

图18-27 LockObject类：域

```
6    public T openRead() {
7      ReadSet readSet = ReadSet.getLocal();
8      switch (Transaction.getLocal().getStatus()) {
9        case COMMITTED:
10         return version;
11       case ACTIVE:
12         WriteSet writeSet = WriteSet.getLocal();
13         if (writeSet.get(this) == null) {
14           if (lock.isLocked()) {
15             throw new AbortedException();
16           }
17           readSet.add(this);
18           return version;
19         } else {
20           T scratch = (T)writeSet.get(this);
21           return scratch;
22         }
23       case ABORTED:
24         throw new AbortedException();
25       default:
26         throw new PanicException("unexpected transaction state");
27     }
28   }
```

图18-28 LockObject类：openRead()方法

图18-29给出LockObject类的openWrite()方法。如果调用发生在事务的外面（第31行），则简单地返回对象的当前版本。如果事务是活动的（第33行），则测试对象是否在它的写集中（第35行）。如果是，则返回这个版本。如果不是，那么在对象被锁定时终止调用者（第37行）。否则，它使用该类型的无参构造函数创建一个新的暂定版本（第39行），通过复制老版本进行初始化（第40行），并将它放入写集中（第41行），然后返回这个暂定版本。

```
29    public T openWrite() {
30      switch (Transaction.getLocal().getStatus()) {
31        case COMMITTED:
32          return version;
33        case ACTIVE:
34          WriteSet writeSet = WriteSet.getLocal();
35          T scratch = (T) writeSet.get(this);
36          if (scratch == null) {
37            if (lock.isLocked())
38              throw new AbortedException();
39            scratch = myClass.newInstance();
40            version.copyTo(scratch);
41            writeSet.put(this, scratch);
42          }
43          return scratch;
44        case ABORTED:
45          throw new AbortedException();
46        default:
47          throw new PanicException("unexpected transaction state");
48      }
49    }
```

图18-29 LockObject类：openWrite()方法

validate()方法（例18-30）仅仅检查对象的读时间戳是否小于等于事务的读时间戳（第56行）。

```
50    public boolean validate() {
51      Transaction.Status status = Transaction.getLocal().getStatus();
52      switch (status) {
53        case COMMITTED:
54          return true;
55        case ACTIVE:
56          return stamp <= VersionClock.getReadStamp();
57        case ABORTED:
58          return false;
59      }
60    }
61  }
```

图18-30 LockObject类：validate()方法

我们现在看一看事务是如何提交的。TinyTM允许用户注册在确认、提交和终止时执行的处理程序，例18-31描述了锁定TM是如何确认事务的。它先锁定写集中的每个对象（第66行）。如果这个锁请求超时，则可能存在死锁，所以该方法返回*false*，意味着事务不能提交。然后，再确认读集。对每个对象，都要检查它没有被其他事务锁定（第70行）且该对象的时间戳没有超过事务的读时间戳（第72行）。

如果确认成功，该事务现在可以提交，例18-32描述了onCommit()处理程序。它增加了版

本时钟值（第83行），将暂定版本从写集复制到原来的对象（第86--89行），并将每个对象的时间戳设置为最新增加的版本时钟值（第90行）。最后，它释放这些锁，并清除线程本地的读/写集，为下一个事务做好准备。

```
62  public class OnValidate implements Callable<Boolean>{
63    public Boolean call() throws Exception {
64      WriteSet writeSet = WriteSet.getLocal();
65      ReadSet readSet = ReadSet.getLocal();
66      if (!writeSet.tryLock(TIMEOUT, TimeUnit.MILLISECONDS)) {
67        return false;
68      }
69      for (LockObject x : readSet) {
70        if (x.lock.isLocked() && !x.lock.isHeldByCurrentThread())
71          return false;
72        if (stamp > VersionClock.getReadStamp()) {
73          return false;
74        }
75      }
76      return true;
77    }
78  }
```

图18-31 LockObject类：onValidate()处理程序

```
79  public class OnCommit implements Runnable {
80    public void run() {
81      WriteSet writeSet = WriteSet.getLocal();
82      ReadSet readSet = ReadSet.getLocal();
83      VersionClock.setWriteStamp();
84      long writeVersion = VersionClock.getWriteStamp();
85      for (Map.Entry<LockObject<?>, Object> entry : writeSet) {
86        LockObject<?> key = (LockObject<?>) entry.getKey();
87        Copyable destin = (Copyable) key.openRead();
88        Copyable source = (Copyable) entry.getValue();
89        source.copyTo(destin);
90        key.stamp = writeVersion;
91      }
92      writeSet.unlock();
93      writeSet.clear();
94      readSet.clear();
95    }
96  }
```

图18-32 LockObject类：onCommit处理程序

到现在为止我们学到了什么？我们已经看到单一的事务内存框架是如何支持两种本质上不同的同步机制的：一种是无干扰的，一种是使用短期的上锁。每种实现本身只提供了较弱的演进保证，所以我们要靠一个独立的争用管理器来确保演进。

18.4 硬事务内存

现在介绍如何通过标准的硬件系统结构来直接在硬件中支持小的短期事务。这里给出的HTM（Hardware Transaction Memory）设计是一种高层的简化形式，但它涵盖了HTM设计的最主要方面。不熟悉缓存一致性协议的读者可以参看附录B。

443
|
445

HTM中的基本思想就是现代缓存一致性协议已经做了要用来实现事务的大部分工作。它

们已能够检测和解决写者之间以及读者和写者之间的同步冲突，并能缓存暂定的改变而不是直接更新内存。我们仅需在此基础上改变一部分细节就可以使用。

18.4.1 缓存一致性

在大多数现代多处理器中，每个处理器都有一个附带的高速缓存（cache），这是一种小容量的高速存储器，用于避免与大容量慢速主存的通信。每个缓存项都包含一组称为行的相邻字，并具有一种从地址到行的映射机制。考虑一种简单的系统结构，其中处理器和存储器通过一个共享的被称为总线的广播媒介进行通信。每个缓存行有一个标记，用于标记状态信息。我们从标准MESI协议开始，协议中每个缓存行用下列状态之一进行标记：

- Modified：缓存中的行已经被修改，且最终必须被写回内存。没有别的处理器缓存了这个行。
- Exclusive：此行还没有被修改，但是没有其他处理器缓存了这个行。（一个行在被修改之前通常以互斥的方式加载。）
- Shared：此行还没有被修改，且其他处理器可能已缓存了此行。
- Invalid：此行没有包含有意义的数据。

缓存一致性协议在单个的加载和存储操作之间检测同步冲突，确保不同的处理器对共享存储器的状态达成一致。当一个处理器加载或存储内存地址a时，它在总线上广播请求，其他处理器和存储器则进行监听（有时称为窥探）：

对缓存一致性协议的完整描述非常复杂，下面是我们所感兴趣的一些主要方面。

- 当处理器请求以互斥方式加载一个行时，任何其他处理器应使这个行的副本失效，任何具有这个行修改副本的处理器必须在加载完成之前将这个行写回存储器。
- 当处理器请求以共享方式加载一个行到自己的缓存中时，任何具有互斥副本的处理器必须将其状态改为共享，且任何具有修改副本的处理器必须在加载完成之前将这个行写回存储器。

445

- 如果缓存满了，则有必要改写一行。如果这个行是共享的或互斥的，则可以简单地抛弃，但如果是修改的，则必须写回存储器。

现在我们描述如何修改这个协议以支持事务。

18.4.2 事务缓存一致性

我们除了给每个缓存行的标记增加一个事务位以外，仍保持和以前的MESI协议一样。通常，这个事务位是未设置的。当一个代表事务的值被放入缓存中时，设置这个位，我们称这个项是*事务的*。只需要确保修改的事务行不能被写回到存储器中，且类事务行无效就可以终止这个事务。

下面是更详细的规则。

- 如果MESI协议使一个事务项无效，那么该事务被终止。这样的无效表示一个同步冲突，或者在两个存储之间或者在加载和存储之间。
- 如果一个修改的事务行失效或被收回，那么它的值将被抛弃而不是写回内存。因为任何事务写的值都是暂定的，当事务为活动时，我们不能让它"逃跑"。相反，必须终止这个事务。

- 如果缓存收回一个事务行，则必须终止这个事务，因为一旦这个行不再在该缓存中，则缓存一致性协议就无法检测到同步冲突。

如果一个事务完成，它的所有事务行都没有失效或收回，那么它就能提交，并清除它的缓存行中的事务位。如果一个失效或收回使得该事务终止，则它的事务缓存行也失效。这些规则能确保提交和终止都是处理器局部的操作步。

18.4.3 改进

尽管这种模式在硬件中正确地实现了事务内存，但它还是存在着一些限制和缺陷。一种几乎对所有HTM方案都存在的制约，就是事务的大小受缓存大小的限制。当一个线程不再调度时，大多数操作系统都会清除缓存，所以事务的持续时间受平台调度量的长度限制。由此可知，HTM最适合于短小的事务。需要长事务的应用则应使用STM，或者结合使用HTM和STM。然而，当事务终止时，由硬件返回一个条件码来说明这个终止是由于同步冲突（事务应该重做），还是由于资源耗尽（事务重做中不存在可做点）则是非常重要的。 447

然而，这种特殊的设计有一些其他的缺点。大多数缓存都是直接映射的，这意味着一个地址a只映射到一个缓存行。任何访问两个被映射到同一个缓存行的内存地址的事务注定会失败，因为第二次的访问将会收回第一次访问，终止事务。有些缓存是组关联的，能将每个地址映射到k个缓存行组成的一个组中。任何访问$k+1$个地址（映射到相同组）的事务也注定要失败。几乎没有缓存是全关联的，能将每个地址映射到缓存中的任一个行。

存在一些通过划分缓存来缓解上述问题的办法。一种是将缓存分成一个较大的、直接映射的主缓存和一个小的、全关联的用来保存从主缓存中溢出项的称为牺牲缓存。另一种方法是将缓存分成一个大的、组关联的非事务缓存和一个小的、全关联的用于事务行的事务缓存。无论哪种方法，都必须修改缓存一致性协议来处理两个缓存间的一致性问题。

另一个缺陷就是没有争用管理器，这意味着事务可能会相互陷死。事务A以互斥方式加载地址a，然后事务B也以互斥方式加载地址a，从而终止A。A立即重新启动，终止B，如此循环。这个问题可以在一致性协议层解决（允许处理器拒绝或推迟一个无效的请求），也可以在软件层解决（通过在软件中让终止的事务指数后退地执行）。

对于深入解决这些问题感兴趣的读者可以参考本章注释。

18.5 本章注释

Maurice Herlihy和Eliot Moss[67]第一个提出了将硬事务内存作为一种通用的多处理器编程模型。Nir Shavit和Dan Touitou[142]提出了第一个软事务内存。retry和orElse构造则归功于Tim Harris、Simon Marlowe、Simon Peyton-Jones和Maurice Herlihy[54]。先前和现在的许多文献都对这个领域作出了贡献，Larus和Rajwar[97]给出了关于技术问题和文献的权威综述。

因果策略的争用管理器源于William Scherer和Michael Scott[137]，贪心策略的争用管理器源于Rachid Guerraoui、Maurice Herlihy和Bastian Pochon[49]。无干扰的STM基于Maurice Herlihy、Victor Luchangco、Mark Moir和Bill Scherer[66]的动态软事务内存算法。基于锁的STM则是在Dave Dice、Ori Shalev和Nir Shavit[32]的事务性上锁2算法的基础上实现的。 448

18.6 习题

习题211. 实现优先级策略、乐心策略和因果策略的争用管理器。

习题212. 不用事务回滚，描述orElse的意义。

习题213. 在TinyTM中，实现FreeObject类的openRead()方法。注意读取Locator域的次序非常重要。讨论为什么你的实现提供了对象可串行化的读。

习题214. 在TinyTM中，设计一种减少对全局版本时钟争用的方法。

习题215. 扩展LockObject类以支持并发读者。

习题216. 在TinyTM中，LockObject类的onCommit()处理程序首先检查对象是否被其他事务锁定，然后检查它的时间戳是否小于等于事务的读时间戳。

- 举例说明为什么必须检查对象是否被锁定。
- 对象有可能被正在提交的事务锁定吗？
- 举例说明为什么在检查版本数之前必须检查对象是否被锁定。

习题217. 设计一种对小数组（如跳表中使用的数组）是最优的AtomicArray<T>实现。

习题218. 设计一种对大数组是最优的AtomicArray<T>实现，在这样的数组中事务可以访问不相交的区域。

附　录

附录A　软件基础

附录B　硬件基础

附录A

The Art of Multiprocessor Programming (Revised First Edition)

软件基础

A.1 引言

本附录描述了理解本书实例以及编写并发程序所需的基本程序设计语言结构。在大多数情况下，我们采用Java语言，但也可以用其他高级语言或库来表达同样的思想。在此，我们回顾理解本书所需要的基本软件概念，首先是关于Java的概念，然后是关于C#或C#/C++的Pthreads库的一些其他的重要模型。遗憾的是，这里的讨论不可能面面俱到，如果有疑问，可以查阅相关语言或库的最新文档。

A.2 Java

Java程序设计语言使用并发模型，在该模型中线程和对象是独立的实体[1]。线程通过调用对象的方法对这些对象进行操作，并通过各种语言和库的结构来协调并发调用。我们首先阐述本书所使用的各种Java基本结构。

A.2.1 线程

450 – 453

一个线程执行一个顺序程序。在Java中，线程通常是java.lang.Thread的子类，它提供了一些方法来创建线程、启动线程、挂起线程、等待线程完成。

首先，创建一个实现Runnable接口的类，该类的run()方法完成所有的工作。例如，下面是一个打印字符串的简单线程。

```
public class HelloWorld implements Runnable {
  String message;
  public HelloWorld(String m) {
    message = m;
  }
  public void run() {
    System.out.println(message);
  }
}
```

我们可以以一个Runnable对象作为参数来调用Thread类的构造函数，将Runnable对象转变为线程，如下所示：

```
String m = "Hello world from thread" + i;
Thread thread = new Thread(new HelloWorld(m));
```

---

[1] 从技术上讲，线程也是对象。

Java提供了一种语法上的快捷方式，称为匿名内部类，它能让你无需显式地定义HelloWorld类：

```
final String m = "Hello world from thread" + i;
thread = new Thread(new Runnable() {
  public void run() {
    System.out.println(m);
  }
});
```

上面的程序段创建一个实现Runnable接口的匿名类，其run()方法的行为已描述。

当线程创建之后，它必须被启动：

```
thread.start();
```

这个方法能使线程运行，调用该方法的线程将立即返回。如果调用者打算等待线程结束，则必须连接线程：

```
thread.join();
```

调用者会被阻塞直到线程的run()方法返回。

图A-1给出了能够初始化多线程、启动多线程，等待多线程完成，然后打印一条消息的方法。该方法创建一个线程数组，并在第2～10行使用匿名内部类语法进行初始化。在循环结束时，则创建了一个休眠线程组成的数组。在第11～13行，该方法启动线程，每个线程执行其run()方法，显示各自的消息。最后，在第14～16行，该方法等待每个线程结束，并在线程完成时显示一条消息。

454

```
1   public static void main(String[] args) {
2     Thread[] thread = new Thread[8];
3     for (int i = 0; i < thread.length; i++) {
4       final String message = "Hello world from thread" + i;
5       thread[i] = new Thread(new Runnable() {
6         public void run() {
7           System.out.println(message);
8         }
9       });
10    }
11    for (int i = 0; i < thread.length; i++) {
12      thread[i].start();
13    }
14    for (int i = 0; i < thread.length; i++) {
15      thread[i].join();
16    }
17  }
```

图A-1 初始化一系列Java线程，启动这些线程并等待它们完成，然后打印一条信息

A.2.2 管程

Java提供了一系列同步访问共享数据的方法，且都是内置的并被打包在一起。在此，我们描述称为管程的内置模型，这是一种最简单、最常用的方法。在第8章已对管程进行了研究。

假设由你来负责电话中心的软件，在高峰时段，拨入电话要比应答到达得快。当一个拨入电话到达时，交换台软件应将它放在一个队列中，同时发出一个已被记录的声明以使拨号者相信你已意识到这次拨入电话是非常重要的，拨入电话将按照它们到达的次序来响应。负

负接听拨入电话的雇员称为接线员。每个接线员可以指派一个接线员线程出队，接听下一个拨入电话。当接线员完成了一个拨入电话操作以后，则可以让下一个拨入电话出队，然后接听它。

图A-2是一个简单但是错误的队列类。拨入电话保存在数组calls中，head是下一个要被处理拨入电话的索引，tail则是数组中下一个空闲槽的索引。

```
1   class CallQueue {
2     final static int QSIZE = 100; // arbitrary size
3     int head = 0;                 // next item to dequeue
4     int tail = 0;                 // next empty slot
5     Call[] calls = new Call[QSIZE];
6     public enq(Call x) {          // called by switchboard
7       calls[(tail++) % QSIZE] = x;
8     }
9     public Call deq() {           // called by operators
10      return calls[(head++) % QSIZE]
11    }
12  }
```

图A-2 错误的队列类

很容易看出，如果两个接线员试图同时出队一个拨入电话，该类则不能正确工作。表达式

```
return calls[(head++) % QSIZE]
```

不能作为一个不可分割的原子步骤。相反，编译器所生成的代码可能会类似于如下形式：

```
int temp0 = head;
head = temp0 + 1;
int temp1 = (temp0 % QSIZE);
return calls[temp1];
```

两个接线员有可能同时执行这些语句：它们同时执行第1行、第2行，等等。最后，两个接线员出队和接听同一个拨入电话，这将使客户感到厌烦。

要让这个队列能够正确地工作，必须保证一次只有一个接线员能够出队下一个拨入电话。这种特性称为互斥。Java提供了一种有用的内置机制来支持互斥，每个对象具有一个锁（隐含的）。如果线程A获得了对象的锁（或者等价地说，就是对象），那么直到A释放这个锁（或者等价地说，直到该对象被解锁）之前，其他线程不能获得这个锁。如果一个类声明一个方法是**synchronized**，则该方法被调用时隐含地获得锁，在返回时释放锁。

下面是一种能够确保enq()和deq()方法满足互斥的方法：

```
public synchronized T deq() {
  return call[(head++) % QSIZE]
}
public synchronized enq(T x) {
  call[(tail++) % QSIZE] = x;
}
```

一旦对同步方法的调用获得了对象的锁，对该对象其他同步方法的调用都将被阻塞，直到该锁被释放为止。（对其他对象的调用，由于受制于其他的锁，所以不会被阻塞。）被同步的方法共用序体通常称为临界区。

同步要比互斥复杂得多。如果接线员试图让一个拨入电话从队列中出队，但是此时队列中没有等待的拨入电话，他应该怎么办？这次拨入电话可能会产生一个异常或返回*null*，但是接下来接线员除了再次尝试还应该做什么？接线员等待一个拨入电话出现是合乎情理的。下

面是对此问题的第一种解决方案：

```
public synchronized T deq() {
  while (head == tail) ; // spin while empty
  call[(head++) % QSIZE];
}
```

455
↓
456

这种解决方式不仅是错误的，而且是一种灾难性的错误。正在出队的线程会在同步方法中等待，从而锁住其他所有的线程，包括试图将拨入电话插入队列的交换台线程。这将产生死锁：持有锁的出队线程在等待入队线程，而入队线程在等待出队线程释放锁。任何一种事件永远都不会发生。

从这个例子中可以知道，如果一个正在执行同步方法的线程需要等待其他事件发生，那么在它等待时必须释放该对象的锁。等待的线程应该周期地重新请求锁以检测它是否可以继续执行。如果能，则继续执行，否则，释放锁并返回继续等待。

在Java中，每个对象都提供了wait()方法，该方法能开锁对象，并挂起调用者。当此线程等待时，其他线程都能锁定和改变对象。随后，当该挂起的线程重新执行时，在从wait()返回前再一次锁定对象。下面是修改过但仍然不正确的出队方法[⊖]：

```
public synchronized T deq() {
  while (head == tail) {wait();}
  return call[(head++) % QSIZE];
}
```

此处，每个接线员线程寻找一个要接听的拨入电话，反复测试队列是否为空。如果为空，则释放锁并等待；如果不为空，则移出并返回一个拨入电话。类似地，入队线程则测试缓冲区是否已满。

等待的线程什么时候会被唤醒？当某些重要的事件发生时，通知等待线程是程序员的责任。notify()方法唤醒一个等待的线程，最终从等待的线程集合中任意选择一个。当线程被唤醒后，它就像其他线程一样竞争锁。当该线程重新获得锁时，从它的wait()调用返回。具体哪个线程被选中则是无法控制的。相比之下，notifyAll()方法将唤醒所有的等待线程。每当对象被开锁后，这些刚被唤醒的线程之一将会重新获得该锁并从wait()调用返回。线程重新获得该锁的次序则是无法控制的。

在电话中心的例子中，有多个接线员和一个交换台。假设交换台软件决定按下面的方法来优化notify()。如果它将一个拨入电话添加到空的队列中，则只通知一个被阻塞的出队线程，这为只有一个拨入电话可以被接听。虽然这种优化看起来是合理的，但它仍有缺陷。设想接线员线程A和B发现队列为空，它们被阻塞等待接听拨入电话。交换台线程S将一个拨入电话放入队列，并调用notify()唤醒一个接线员线程。由于通知是异步的，所以存在延迟。S然后返回并将另一个拨入电话放入队列，因为队列已经有一个等待的拨入电话，所以它不会通知其他线程。交换台线程的notify()最终生效，唤醒A，但没有唤醒B，尽管存在一个拨入电话可以让B接听。这种问题称为丢失唤醒：一个或多个等待线程在它们所等待的条件已经变为真时，并没有被通知。更详细的讨论参见8.2.2节。

457

A.2.3 层从和睡眠

除了允许持有锁的线程释放锁和中止wait()方法外，对于那些没有持有锁的线程，Java

⊖ 这个程序不会被编译，因为调用wait()会抛出InterruptedException异常，该异常必须被捕获或再次抛出，正如8.2.3节中讨论的。我们需要忽略这些异常以便使例子更易于阅读。

还提供了其他的中止方法，yield()调用可以用来中止线程，请求调度运行其他的线程。调度器决定是否暂停该线程以及何时重新启动它。如果没有其他线程可以执行，调度将会忽略yield()调用。16.4.1节讲述了屋从为何是一种高效的能够避免活锁的方法。sleep(t)（其中t是一个时间值）调用控制调度器在该时间段内停止该线程的执行。调度器可以在任何时候自由地重新启动线程。

A.2.4 本地线程对象

让每个线程拥有自己的私有变量实例往往是非常有用的。Java通过ThreadLocal<T>类支持这种本地线程对象。该类管理着一个类型为T的对象集合，每个对象对应于一个线程。由于Java中没有独立本地线程变量，所以其接口复杂且不易于使用。但是，这些对象却是非常有用的且经常使用，下面我们回顾一下如何使用它们。

ThreadLocal<T>类提供了get()和set()方法，用于读取和修改线程的本地值。线程第一次要获得本地对象的值时，则调用initialValue()方法。我们不能直接使用ThreadLocal<T>类，必须将本地线程变量定义为ThreadLocal<T>的子类，它重写其父类的initialValue()方法以初始化每个线程的对象。

这种机制可以用一个实例加以说明。在我们的大多数算法中，都假设n个并发线程中的每一个都有一个唯一的从0到$n-1$之间的本地线程标识。为了提供这种标识，我们来说明如何用一个静态方法定义一个ThreadID类：get()能返回调用线程的标识。当一个线程首次调用get()方法时，被指定下一个未使用的标识，该线程随后的每次调用都将返回其标识。

图A-3描述了使用本地线程对象来实现这种类的最简单方式。第2行声明了一个整型域 458
nextID，用于保存将要产生的下一个标识。第3行到第7行定义了一个只能在ThreadID类的体内访问的内部类。该内部类管理着线程的标识。它是ThreadLocal<Integer>的子类，重写了initialValue()方法以对当前线程指定下一个未使用的标识。

```
1  public class ThreadID {
2    private static volatile int nextID = 0;
3    private static class ThreadLocalID extends ThreadLocal<Integer> {
4      protected synchronized Integer initialValue() {
5        return nextID++;
6      }
7    }
8    private static ThreadLocalID threadID = new ThreadLocalID();
9    public static int get() {
10     return threadID.get();
11   }
12   public static void set(int index) {
13     threadID.set(index);
14 }
```

图A-3 ThreadID类：给每个线程一个唯一的标识

由于内部的ThreadLocalID类只被使用一次，因此对它取名没有什么实际意义（就像给你的感恩火鸡起名一样没有意义）。相反，如前面讲述，我们往往是使用匿名类。

下面是一个如何使用ThreadID类的例子：

```
thread = new Thread(new Runnable() {
  public void run() {
```

```
    System.out.println("Hello world from thread" + ThreadID.get());
  }
});
```

编程提示A.2.1 在类型表达式ThreadLocal<Integer>中，必须使用Integer而不是int。因为int是原子类型，而Integer则是引用类型，只有引用类型允许在尖括号内。在Java 1.5之后，加入了一种称为auto-boxing的特性，允许交替地使用int和Integer，例如：

```
Integer x = 5;
int y = 6;
Integer z = x + y;
```

更多细节请查阅Java文档。

459

A.3 C#

C#是和Java类似的一门语言，运行在Microsoft的.Net平台上。

A.3.1 线程

C#提供了与Java相类似的线程模型。C#线程是由System.Threading.Thread类实现的。当创建一个线程时，通过给它传递一个ThreadStart委托（一种指向所要调用方法的指针），告诉所要做的事。例如，下面是一个打印消息的例子：

```
void HelloWorld()
  {
    Console.WriteLine("Hello World");
  }
```

接着，将这个方法转变成ThreadStart代理，并将该委托传递给该线程的构造函数。

```
ThreadStart hello = new ThreadStart(HelloWorld);
Thread thread = new Thread(hello);
```

C#提供了简短的语法，称为匿名方法，该方法允许直接定义一个委托。例如，可以将前面的步骤组合到单一的表达式中：

```
Thread thread = new Thread(delegate()
  {
    Console.WriteLine("Hello world");
  });
```

和Java一样，当线程被创建以后，它必须启动：

```
thread.Start();
```

这个调用将使得线程开始运行，而调用者也立即返回。如果调用者要等待线程完成，它必须连接线程：

```
thread.Join();
```

调用者将被阻塞直到线程的方法返回。

图A-4描述了一个能够初始化多个线程、启动它们、等待其完成并打印出信息的方法。该方法创建一个线程数组，并用它自己的ThreadStart委托初始化每个线程。然后启动这些线程，每个线程执行它的委托，显示其信息。最后，等待每个线程完成，并在它们全部完成时显示一条信息。除了有少部分语法不同之外，该代码与用Java写的代码非常相似。

460

```
1         static void Main(string[] args)
2         {
3             Thread[] thread = new Thread[8];
4             // create threads
5             for (int i = 0; i < thread.Length; i++)
6             {
7                 String message = "Hello world from thread" + i;
8                 ThreadStart hello = delegate()
9                 {
10                    Console.WriteLine(message);
11                };
12                thread[i] = new Thread(hello);
13            }
14            // start threads
15            for (int i = 0; i < thread.Length; i++)
16            {
17                thread[i].Start();
18            }
19            // wait for them to finish
20            for (int i = 0; i < thread.Length; i++)
21            {
22                thread[i].Join();
23            }
24            Console.WriteLine("done!");
25        }
```

图A-4 该方法初始化一系列C#线程，启动线程，等待线程完成，然后打印出信息

A.3.2 管程

对于简单的互斥，C#提供了与Java中的**synchronized**修饰符相类似的锁定对象的能力：

```
int GetAndIncrement()
{
    lock (this)
    {
        return value++;
    }
}
```

与Java不同的是，C#不允许直接使用lock语句修改方法。相反，lock语句被用来封装方法体。

并发数据结构比互斥要求的更多：它们还要求具有等待条件并给条件发出信号的能力。
461 与Java中每个对象都是一个隐含的管程不同，在C#中必须明确地创建与对象相关的管程。要获得一个管程锁，应调用Monitor.Enter(this)，而要释放一个锁，则调用Monitor.Exit(this)。每个管程具有一个隐含条件，该条件通过调用Monitor.Wait(this)进行等待，通过调用Monitor.Pulse(this)或Monitor.PulseAll(this)给这个条件发出信号，分别唤醒一个或所有的睡眠线程。图A-5和图A-6描述了如何使用C#管程来实现一个有界队列。

```
1   class Queue<T>
2   {
3       int head, tail;
4       T[] call;
5       public Queue(int capacity)
6       {
7           call = new T[capacity];
8           head = tail = 0;
9       }
10      public void Enq(T x)
11      {
12          Monitor.Enter(this);
13          try
14          {
15              while (tail - head == call.Length)
16              {
17                  Monitor.Wait(this); // queue is full
18              }
19              calls[(tail++) % call.Length] = x;
20              Monitor.Pulse(this);  // notify waiting dequeuers
21          }
22          finally
23          {
24              Monitor.Exit(this);
25          }
26      }
27  }
28  }
```

图A-5 有界队列类：域和enq()方法

```
29  public T Deq()
30      {
31          Monitor.Enter(this);
32          try
33          {
34              while (tail == head)
35              {
36                  Monitor.Wait(this); // queue is empty
37              }
38              T y = calls[(head++) % call.Length];
39              Monitor.Pulse(this);  // notify waiting enqueuers
40              return y;
41          }
42          finally
43          {
44              Monitor.Exit(this);
45          }
46      }
47  }
```

图A-6 有界队列类：deq()方法

A.3.3 本地线程对象

C#提供了一种非常简单的能够使静态域变为本地线程的方法：在域声明前加上属性[ThreadStatic]。

```
[ThreadStatic]
static int value;
```

由于初始化只需进行一次，而不是对每个线程一次，所以不需对[ThreadStatic]域提供初始值。相反，每个线程将会发现该域值初始时为相应类型的默认值：整型为0，引用为*null*等。

图A-7描述了ThreadID类的实现（图A-3是Java版的）。关于这个程序有一点需要讨论。线程第一次检查它的[ThreadStatic]标识时，该域为整型的默认值0。为了区别没有初始化的0和线程ID 0，该域保存的线程ID用1替换；对于线程0该域值为1，后面相应增加。

```
1   class ThreadID
2   {
3       [ThreadStatic] static int myID;
4       static int counter;
5       public static int get()
6       {
7           if (myID == 0)
8           {
9               myID = Interlocked.Increment(ref counter);
10          }
11          return myID - 1;
12      }
13  }
```

图A-7 ThreadID类使用[ThreadStatic]为每个线程提供一个唯一的标识

A.4 Pthreads

Pthreads为C和C++提供了许多同样的功能。使用Pthreads编程时必须导入头文件：

```
#include <pthread.h>
```

下面的函数创建并启动一个线程：

```
int pthread_create (
  pthread_t* thread_id,
  const pthread_attr_t* attributes,
  void* (*thread_function)(void*),
  void* argument);
```

第一个参数是一个指向线程自身的指针。第二个参数允许指定线程的各个属性。第三个参数是一个指向线程要运行的代码的指针（在C#中则是一个委托，在Java中是一个Runnable对象）。第四个参数是线程函数的参数。与Java和C#不同，一个调用既能创建线程又能启动线程。

当函数返回或调用pthread_exit()时线程结束。线程也能通过下面的调用连接：

```
int pthread_join (pthread_t thread, void** status_ptr);
```

退出状态则存放在最后一个参数中。例如，下面的程序打印出每个线程的简单消息。

```
#include <pthread.h>
#define NUM_THREADS 8
void* hello(void* arg) {
  printf("Hello from thread %i\n", (int)arg);
}
int main() {
  pthread_t thread[NUM_THREADS];
  int status;
```

```
  int i;
  for (i = 0; i < NUM_THREADS; i++) {
    if ( pthread_create(&thread[i], NULL, hello, (void*)i) != 0 ) {
      printf("pthread_create() error");
      exit();
    }
  }
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread[i], NULL);
  }
}
```

对Pthreads库的调用会锁定互斥量mutexes。互斥通过以下代码创建：

```
int pthread_mutex_init (pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* attr);
```

464

互斥可以被锁定：

```
int pthread_mutex_lock (pthread_mutex_t* mutex);
```

也可以解锁：

```
int pthread_mutex_unlock (pthread_mutex_t* mutex);
```

与Java中的锁一样，如果一个互斥忙，它可以立即返回：

```
int pthread_mutex_trylock (pthread_mutex_t* mutex);
```

Pthreads库提供条件变量，它能通过下面的语句创建：

```
int pthread_cond_init (pthread_cond_t* cond, pthread_condattr_t* attr);
```

通常，第二个参数将属性设置为非默认值。与Java和C#不同，锁和条件变量之间的关联是显式的而非隐含的。下面的调用在条件变量上释放锁并等待：

```
int pthread_cond_wait (pthread_cond_t* cond, pthread_mutex_t* mutex);
```

（就像在其他语言中一样，当一个线程被唤醒时，并不能保证等待的条件成立，所以必须要显式地检查。）也可能出现超时等待。

下面的调用与Java中的notify()相似，至少唤醒一个被挂起的线程：

```
int pthread_cond_signal (pthread_cond_t *cond);
```

下面的调用与Java中的notifyAll()相似，唤醒全部被挂起的线程：

```
int pthread_cond_broadcast (pthread_cond_t* cond);
```

因为C没有垃圾回收，所以对于线程、锁和条件变量都提供完整的destroy()函数来回收这些资源。

图A-8和图A-9描述了一个简单的并发FIFO队列，调用被保存在一个数组中，head和tail域统计入队和出队的调用次数。与Java中的实现一样，采用单一的条件变量来等待，以使缓冲区变为非满或非空。

本地线程存储器

列A-10说明了Pthreads是如何管理本地线程存储器的。Pthreads库将一个线程特定值与一个key联系起来，在第1行中声明并在第6行进行初始化。该值是一个指针，初始化为*null*。线程通过调用threadID_get()获得一个ID。该方法查找受限于key的本地线程值（第10行）。在第一次调用时，该值为*null*（第11行），所以线程必须通过增加counter变量得到一个新的唯一的ID。此处，我们使用互斥来同步对计数器的访问（第12～16行）。

465

```
1  #include <pthread.h>
2  #define QSIZE 16
3  typedef struct {
4    int buf[QSIZE];
5    long head, tail;
6    pthread_mutex_t *mutex;
7    pthread_cond_t *notFull, *notEmpty;
8  } queue;
9  void queue_enq(queue* q, int item) {
10   // lock object
11   pthread_mutex_lock (q->mutex);
12   // wait while full
13   while (q->tail - q->head == QSIZE) {
14     pthread_cond_wait (q->notFull, q->mutex);
15   }
16   q->buf[q->tail % QSIZE] = item;
17   q->tail++;
18   // release lock
19   pthread_mutex_unlock (q->mutex);
20   // inform waiting dequeuers
21   pthread_cond_signal (q->notEmpty);
22 }
23 queue *queue_init (void) {
24   queue *q;
25   q = (queue*)malloc (sizeof (queue));
26   if (q == NULL) return (NULL);
27   q->head = 0;
28   q->tail = 0;
29   q->mutex = (pthread_mutex_t*) malloc (sizeof (pthread_mutex_t));
30   pthread_mutex_init (q->mutex, NULL);
31   q->notFull = (pthread_cond_t*) malloc (sizeof (pthread_cond_t));
32   pthread_cond_init (q->notFull, NULL);
33   q->notEmpty = (pthread_cond_t*) malloc (sizeof (pthread_cond_t));
34   pthread_cond_init (q->notEmpty, NULL);
35   return (q);
36 }
```

图A-8 使用Pthreads的并发FIFO队列的初始化和入队方法

```
37 int queue_deq(queue* q) {
38   int result;
39   // lock object
40   pthread_mutex_lock (q->mutex);
41   // wait while full
42   while (q->tail == q->head) {
43     pthread_cond_wait (q->notEmpty, q->mutex);
44   }
45   result = q->buf[q->head % QSIZE];
46   q->head++;
47   // release lock
48   pthread_mutex_unlock (q->mutex);
49   // inform waiting dequeuers
50   pthread_cond_signal (q->notFull);
51   return result;
52 }
53 void queue_delete (queue* q) {
54   pthread_mutex_destroy (q->mutex);
55   free (q->mutex);
56   pthread_cond_destroy (q->notFull);
57   free (q->notFull);
58   pthread_cond_destroy (q->notEmpty);
59   free (q->notEmpty);
60   free (q);
61 }
```

图A-9 Pthreads：并发FIFO队列的出队和删除方法

```
1  pthread_key_t key;    /* key */
2  int counter;          /* generates unique value */
3  pthread_mutex_t mutex;   /* synchronizes counter */
4  threadID_init() {
5    pthread_mutex_init(&mutex, NULL);
6    pthread_key_create(&key, NULL);
7    counter = 0;
8  }
9  int threadID_get() {
10   int* id = (int*)pthread_getspecific(key);
11   if (id == NULL) {   /* first time? */
12     id = (int*)malloc(sizeof(int));
13     pthread_mutex_lock(&mutex);
14     *id = counter++;
15     pthread_setspecific(key, id);
16     pthread_mutex_unlock(&mutex);
17   }
18   return *id;
19 }
```

图A.10 该程序使用Pthreads本地线程存储管理调用为每个线程提供一个唯一的标识符

A.5 本章注释

Java程序设计语言是由James Gosling[46]所创立的。Dennis Ritchie被认为是C语言的创立者。Pthreads是IEEE Posix包的一部分。尽管使用了不同的等待和通知机制，但基本的管程模型则被认为是由Tony Hoare[71]和Per Brinch Hansen[52]创建的。Java（及随后的C#）所使用的机制最初是由Butler Lampson和David Redell[96]提出的。

466
|
467

硬件基础

一个初学者正在试图通过重新关闭/开启电源来修理一台被HP的Lisp机器。Knight看到学生所做的事，严厉地说："当你不知道哪里出错的时候，总是重启是解决不了问题的。"说完之后，Knight关掉电源按键又打开，机器就正常工作了。

——来自"AI Koans"，20世纪80年代流行于MIT的笑话

B.1 引言（和一个难题）

除非已了解什么是多处理器，否则无法在多处理器上进行有效地编程。我们无需等待大量关于计算机系统结构的知识，就能为单处理器编写出相当好的程序，然而对于多处理器来说，情况就大不一样了。下面通过一个难题来说明这一点。考虑两个程序，除了一个比另一个效率要低一些外，它们在逻辑上是等价的，效率低一些的程序较为简单。如果对现代多处理器系统结构没有一个基本的理解，则无法解释这种差异而且也无法避免这种危险。

下面是该问题的相关背景。假设两个线程共享一个资源，在同一时刻该资源只能被一个线程使用。为了防止同时使用，每个线程在使用资源前必须要对资源上锁，使用完资源后要解锁。在第7章中已学习了多种实现锁的方法。针对这个问题，我们考虑两种简单的实现。其中都将锁看做是一个布尔域。如果该域值为*false*，则锁是空闲的，否则锁正在被使用。可以使用getAndSet(*v*)方法来控制锁，该方法能自动将参数*v*与布尔域的值交换。若要获得锁，线程则调用getAndSet(*true*)。如果调用返回*false*，则锁是空闲的，调用者成功锁定对象。否则对象已经被锁定，线程必须以后再次尝试。线程通过简单地将*false*存入布尔域中来释放锁。

```
1  public class TASLock implements Lock {
2    ...
3    public void lock() {
4      while (state.getAndSet(true)) {} // spin
5    }
6    ...
7  }
```

图B.1 TASLock类

```
1  public class TTASLock implements Lock {
2    ...
3    public void lock() {
4      while (true) {
5        while (state.get()) {}; // spin
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   ...
11 }
```

图B.2 TTASLock类

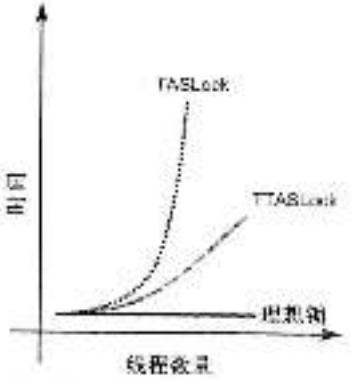
在图B-1中，测试-设置锁（TASLock）不断地调用getAndSet(*true*)（第4行），直到它返回*false*为止，然而，在图B-2中，测试-测试-设置锁（TTASLock）则不断地读锁的布尔域（在第5行调用state.get()），直到返回*false*时才调用getAndSet()（第6行）。对锁值的读操作是原子的，对锁值的getAndSet()调用也是原子的，但它们的组合却不是原子的：在线程读锁的值和调用getAndSet()之间，锁的值有可能已经发生了改变。

468
|
469

在继续讨论之前，首先应该理解TASLock和TTASLock这两个算法在逻辑上是一样的。原因

很简单：在TTASLock算法中，当读到锁为空闲时并不能保证接下来的getAndSet()调用能够成功，其原因在于其他的线程有可能在读锁和尝试获得锁的这段时间内获得了锁。那么，为什么还要在尝试获得锁之前去读锁呢？

这是一个令人费解的问题，虽然这两个锁的实现在逻辑上是等价的，但它们的表现却非常不同。在1989年的经典实验中，Anderson在当时的一些多处理器上测试了执行一个简单程序所需的时间。他测量了n个线程去执行一个较小的临界区执行一百万次所花费的时间。图B-3描述了每种锁所花费的时间，它是作为线程数量的函数来绘制的。在理想情况下，TASLock和TTASLock的曲线和底部理想曲线一样平坦，这是因为每个运行都进行了相同数量的增加。然而，可以看到两条曲线的斜率都在增加，这说明由锁导致的延迟随着线程数量的增加而增加。但有趣的是，TASLock锁要比TTASLock锁慢得多，尤其是当线程数量增加时情况更明显，这是什么原因呢？



图B-3 TASLock、TTASLock和理想锁的执行时间比较

本章描述了要写出高效的并发算法和数据结构所需的关于多处理器系统结构的大多数知识。（沿着这条思路，我们将解释图B-3中曲线分叉的原因。）

我们主要考虑下面的组成部件：

- 处理器是执行软件线程的硬件设备。通常，线程的数量要比处理器的数量多，每个处理器运行一个线程一段时间，然后将它放置在一边，转去执行另一个线程。
- 互连线是连接处理器与处理器以及处理器与内存之间的通信媒介。
- 存储器实际上是一种存放数据的层次组织部件，包括一个或多个层的小容量高速缓存直到相对较慢的大容量主存。理解这些层次之间的相互关系是理解许多并发算法实际性能的基石。

从我们的观点来看，一种系统结构的原理决定了其他的所有事情：处理器和主存相距很远。处理器从内存读一个数据要花费很长时间。处理器将数据写到内存也要花费很长时间，而处理器要确认数据是否已经写入内存则需要花费更长的时间。访问内存不像是打电话而更像是寄信。我们在本章所处理的每件事都是在试图减少处理器访问内存所花费的时间（"高延迟"）。

处理器和内存的速度都在快速地变化，但它们的相对性能却几乎没有改变。我们来考虑类似的情形：设想现在是1980年，你负责中型城市曼哈顿的送信服务。虽然在平坦的马路上汽车的效率要超过自行车，但在交通拥堵的道路上汽车却赶不上自行车的效率，所以你选择了自行车。尽管自行车和汽车的技术都已经发展和进步了，然而，上面系统结构中的类比在此仍然适用。就像现在，如果你正在设计城市的投信服务，应该使用自行车而不是汽车。

B.2 处理器和线程

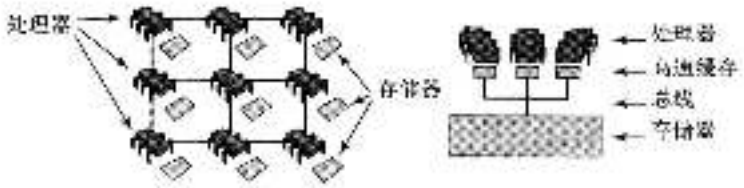
一个多处理器由多个硬件处理器组成，其中每一个处理器都能执行一个顺序程序。当讨论多处理器系统结构时，基本的时间单位是指令周期，即处理器提取和执行一条指令所花费的时间。从绝对速度上来看，时钟周期随着技术的进步发生了转变（从1980年的约每秒一千万次到2005年的约每秒30亿次），在不同的平台上也不尽相同（控制烤面包机的处理器的时钟

周期比控制网络服务器的要长）。然而，若采用指令周期来表示指令执行的相对代价，如访问内存，其变化却很慢。

线程是一个顺序程序。处理器是一个硬件设备，而线程则是一种软件构造。处理器可以执行一个线程一段时间，然后不管该线程是否执行为一个线程，即我们熟悉的上下文切换，处理器可以因为各种原因撤销一个线程或从调度中删除该线程。线程有可能已经发出一个内存请求，而该请求要花费一段时间才能得到满足，或者线程已经运行了足够长的时间，该让别的线程运行了。当线程被从调度中删除时，它可能重新在另一个处理器上执行。

B.3 互连线

互连线是处理器与内存以及处理器与处理器之间进行通信的媒介。有两种基本的互连结构：SMP（symmetric multiprocessing，对称多处理）和NUMA（nonuniform memory access，非一致内存访问），如图B.4所示。



图B.4 右边是带高速缓存的SMP系统结构，左边是无高速缓存的NUMA系统结构

在SMP系统结构中，处理器和内存之间采用总线互连结构，类似于较早以太网上的广播媒介。处理器和主存都有用来负责发送和监听总线上广播的信息的总线控制单元（监听有时称为探听）。如今SMP系统结构非常普遍，因为它们最容易构建。但是对于数量较多的处理器来说，这种系统结构不具有扩展性，因为总线最终将变为过载。

在NUMA系统结构中，一系列节点通过点对点网络相互连接，就像一个小型的局域网。每个节点包含一个或多个处理器和一个本地存储器。一个节点的本地存储对于其他节点是可访问的，所有节点的本地存储一起形成一个可以被所有处理器共享的全局存储器。NUMA的
472 名字反映了一个事实，即处理器访问自己节点存储器的速度要比访问其他节点存储器的速度快。网络要比总线复杂，需要更加复杂的协议，但是对于数量较多的处理器来说网络比总线的可扩展性更好。

可以在SMP和NUMA系统结构之间设计一种折中方案：设计一种混合系统结构，同一集群中的处理器通过总线通信，而不同集群中的处理器则通过网络通信。

从程序员的角度看，底层平台无论是基于总线，网络还是混合结构似乎并不重要。然而，理解互连线是由处理器所共享的有限资源是很重要的。如果一个处理器使用较多的互连线带宽，那么其他的处理器就会被延迟。

B.4 主存

主存由所有处理器共享使用，它是一个很大的由字所组成的数组，通过地址进行索引。依赖于这种平台，一般情况下，一个字的长度是32位或64位，地址也是一样。稍许简化地来看，处理器给主存发送一个包含有目标地址的信息，读取主存的值。处理器发送一个地址和

新的数据，向主存中写入一个值，当新数据被写入后，主存会发回一个确认信息。

B.5 高速缓存

不幸的是，在现代系统结构中一次主存访问可能会花费数百个时钟周期，因此，存在这样一种危险，即处理器将会花费许多时间等待主存响应请求。解决这一问题的方法就是引入一个或多个高速缓存：一种与处理器非常接近且此速度比主存要快的小容量存储器。这些高速缓存逻辑上位于处理器和主存之间：当处理器试图从给定的主存地址读取一个值时，首先查看该值是否已经在高速缓存中，如果在，则不需要进行较慢的主存访问。如果找到目标地址的值，则称处理器在高速缓存中命中；否则称为缺失。同样，如果处理器试图写的地址在高速缓存中，那么它就不需要执行较慢的主存访问。在高速缓存中符合请求的比例称为高速缓存的命中率。 473

高速缓存是非常有效的，因为大多数程序都表现出较高的局部性：如果处理器读或写一个内存地址（或者内存单元），那么它很快将读或写同一个地址。况且，如果处理器读或写一个内存单元，那么它很可能会立刻读或写该单元附近的单元。为了利用第二个结论，高速缓存通常在一个比字更大的粒度上进行操作：高速缓存维护一组邻近的字，称为缓存行（或缓存块）。

实际上，大多数处理器都具有二级高速缓存，称为L1 Cache和L2 Cache。L1 Cache通常和处理器在同一个芯片中，对它的访问通常需要一到两个时钟周期；L2 Cache则可放置在芯片中也可以不放置在芯片中，对它的访问需要数十个时钟周期。两者都比要花费数百个时钟周期的内存快得多。当然，对于不同的平台，访问次数会随之而变化，许多多处理器都具有更为精细的高速缓存结构。

NUMA系统结构的最初提议中并不包含高速缓存，因为当初认为有本地内存就已足够了。然而后来的商用NUMA系统结构却包含有高速缓存。术语缓存一致的NUMA（cc-NUMA）有时用来指带有高速缓存的NUMA系统结构。为了避免歧义，今后除非明确指出，我们所说的NUMA都是缓存一致的。

由于高速缓存的生产价格高，因此其大小要比内存小得多：在同一时刻只有一部分内存单元被放置在高速缓存中。因此，我们希望在高速缓存中保存那些最常使用的单元。这意味着当内存单元要被装入到高速缓存中而缓存已满时，有必要收回一个缓存块。如果该缓存块没有被修改则直接丢弃，如果已被修改则写回主存。替换策略则决定将替换掉哪一个缓存块，以便为新的内存单元腾出空间。如果替换策略是自由地替换任何缓存块，则称该高速缓存是全相联的。另一方面，如果只可以替换唯一的缓存块，则称该缓存是直接映射的。如果我们折中这种差别，允许使用一组大小为k的块的集合中的任何一个块来替换一个给定的块，则称这样的缓存为k组组相联的。

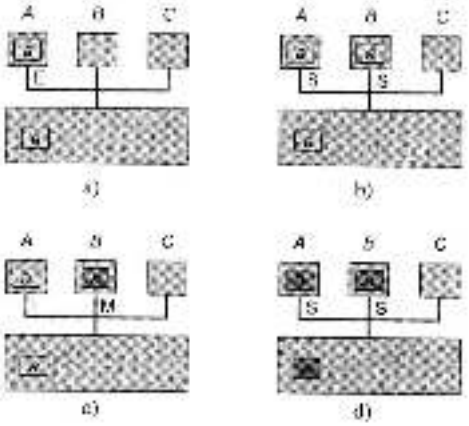
B.5.1 一致性

当一个处理器读或写被另一处理器装入高速缓存的主存地址时，将发生共享（或称为争用）现象。如果两个处理器都只读数据而不修改，那么数据可以装入到两个处理器的高速缓存中。然而，如果一个处理器要更新共享的缓存块，那么另一个处理器的副本必须作废以确保它不会读到过期的值。通常称这个问题为缓存一致性。文献中包含有各种复杂和巧妙的 474

缓存一致性协议。我们首先对缓存块的各种状态进行命名，然后讨论一种最常用的称为MESI的协议。该协议已经用在Pentium和PowerPC处理器中。下面是缓存块的状态。

- modified（修改）：缓存中的块已被修改，它最终必须写回主存。其他的处理器不能再缓存这个块。
- exclusive（互斥）：缓存块还未被修改，且其他的处理器不能将这个块装入缓存。
- shared（共享）：缓存块未被修改，且其他处理器可以缓存这个块。
- invalid（无效）：块中不包含任何有意义的数据。

下面用一个简短的例子来说明MESI协议，如图B-5所示。为了方便起见，假设处理器和内存之间是通过总线连接的。



图B-5 MESI高速缓存一致性协议的状态转换实例。在a中，处理器A从地址a读数据，将数据存入它的缓存并置为exclusive状态。在b中，当处理器B试图从相同的地址读数据时，A检测到地址冲突，以相关数据做出响应。此时，a同时被处理器A和B以shared状态装入缓存。在c中，如果B要对共享地址a进行写操作，则将其状态改变为modified，并广播此信息以提醒A（以及其他任何可能已将该数据装入缓存的处理器）将它的缓存块状态设置为invalid，在d中，如果A随后从a读数据，它会广播它的请求，B则通过将修改过的数据发送到A和主存，并置两个副本的状态为shared来做出响应

处理器A从地址a读数据，将数据存入它的高速缓存并置为exclusive状态。当处理器B试图
475 从同一个地址读数据时，A检测到地址冲突，并以相关数据做出响应。此时，a同时被A和B以shared状态装入缓存。如果B要对地址a进行写操作，则将其状态改变为modified，并广播此信息以提醒A（以及其他任何可能已将该数据装入缓存的处理器）将它的缓存块状态设置为invalid。如果A随后要从a读数据，它会广播它的请求，B则通过将修改过的数据发送到A和主存，并置两个副本的状态为shared来做出响应。

当处理器访问逻辑上不同的数据时，由于它们要访问的内存单元对应于同一个缓存块而导致发生冲突的现象称为错误共享。这种情形反映了一种难于处理的权衡问题：较大的缓存块对局部性有利，但却增加了错误共享的可能性。出现错误共享的可能性可以通过确保独立线程并发访问的数据对象距离内存足够远来降低。例如，让多个线程共享一个字节数组则可以导致错误共享，但是若让它们共享双精度整型数组则出现错误共享的危险性就变得很小了。

B.5.2 自旋

如果处理器不断地测试内存中的某个字，等待另一个处理器改变它，则称该处理器正在自旋。自旋依赖于体系结构，能对整个系统的性能产生显著的影响。

对于无高速缓存的SMP系统结构来说，自旋是一种非常糟糕的想法。每当处理器读内存时，都会消耗总线带宽却没有做任何有用的工作。由于总线是广播媒介，这些直接对内存的请求可能会阻止其他处理器的推进。

对于无高速缓存的NUMA系统结构，如果地址位于处理器的本地存储器中，那么自旋是可以接受的。尽管无高速缓存的多处理器系统结构很少见，我们仍然要研究当考虑具有自旋的同步协议时，是否允许每个处理器在它自己的本地存储器上自旋。

对于具有高速缓存的SMP或NUMA系统结构，自旋仅消耗非常少的资源。处理器第一次读地址时，会产生一个高速缓存缺失，将该地址的内容加载到缓存块中。此后，只要数据没有改变，处理器只需从它自己的高速缓存中重读数据，不需占用互连带宽。这种过程称为本地自旋。当高速缓存状态发生改变时，处理器产生一个高速缓存缺失，观察到数据已发生改变，并停止自旋。

B.6 考虑高速缓存的程序设计问题

现在可以解释为什么B.1节中的TTASLock要优于TASLock。TASLock每次对锁调用getAndSet(*true*)时，都在互连线上发送一个消息，引发大量的通信流量。在SMP系统结构上，引发的流量有可能完全占有互连线，从而延迟了所有的线程，包括正在试图释放锁的线程以及那些没有争用锁的线程。与此相反，当锁繁忙时，TTASLock则自旋，读取本地缓存的锁副本，并不在互连线上产生流量，从而说明它具有更好的性能。 476

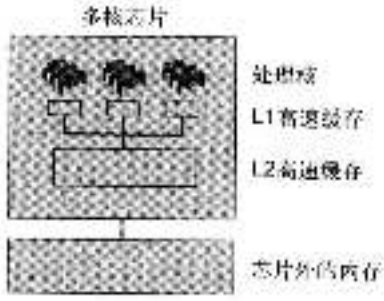
然而TTASLock本身与理想情况仍相距很远。当锁被释放时，其所有的缓存副本变为无效，而所有的等待线程都在调用getAndSet(*true*)，从而导致了流量的激增，虽然比TASLock的小，但仍然是很可观的。

我们在第7章对带有锁的高速缓存的交互问题进行了讨论。同时，下面给出了几种关于如何组织数据以避免错误共享的简单方法。其中的一些技术在类似于C或C++这种支持细粒度存储控制的语言中实现要比在Java中容易得多。

- 独立访问的对象或域应该被补充调整使得它们能在不同的缓存块上结束使用。
- 将只读数据与那些频繁修改的数据相分离。例如，考虑一个链表，其结构是固定不变的，但其元素的值频繁地变化。为了确保修改不会减慢对链表的遍历，应该补充调整值域以使得每个值占满一个缓存块。
- 在允许的情形下，将一个对象分解成一些本地线程片段。例如，用于统计的计数器可以分解为一个由计数器组成的数组，每个线程一个，每个都位于不同的缓存块中。当一个共享的计数器导致无效的流量时，分解的计数器则允许每个线程更新自己的副本而不会引起相关的流量。
- 如果用一个锁来保护频繁修改的数据，那么要将锁和数据保存在不同的缓存块，这样，正在尝试获得锁的线程不会干扰锁的持有者对数据的访问。
- 如果用一个锁来保护不常争用的数据，那么要尽量将锁和数据保存在同一个缓存块中，这样，获取锁的同时也会将一部分数据装入到缓存块中。

B.7 多核与多线程体系结构

如图B-6所示，在多核体系结构中，多个处理器被放置在同一个芯片中。芯片上的每个处理器通常都有自己的L1高速缓存，但它们共享一个公共的L2高速缓存。处理器之间可以通过共享L2高速缓存进行高效的通信，从而避免了进入内存并调用那些令人讨厌的一致性协议。



图B-6 多核SMP系统结构。L2缓存在处理器芯片上并且被所有处理器共享，而主存在芯片外

在多线程体系结构中，一个处理器可以一次执行两个或更多个线程。许多现代处理器都具有重要的内在并行性。它们能够不按次序来执行指令，或以并行的方式执行（如保持定长和浮点单元同时繁忙），甚至可以在分支或数据计算之前预测地执行指令。为了保持硬件单元繁忙，多线程的处理器能将多个流的指令混合执行。

477

现代处理器系统结构将多核系统结构与多线程系统结构相结合，多个独立地支持多线程的核可以放置到同一个芯片中，在一些多核芯片上，上下文切换所花费的代价非常低，并可以在很细的粒度上进行，特别对于那些每条指令都要切换的上下文更是如此。因此，多线程方式避免了较大的内存访问时延：当一个线程访问内存时，处理器会让另一个线程执行。

松弛的内存一致性

当处理器要将一个值写入内存时，该值被保存在高速缓存中并被标记为延值，以表明该值最终必须要写回主存。对于大多数现代处理器来说，当写请求发生时并没有直接作用到主存中，而是将它们收集到一个称为写缓冲区（或称存储缓冲区）的硬件队列中，在以后的某个时刻再一起作用到主存上。写缓冲具有两个优点。首先，它能更加高效地发布一批请求，称为批处理。其次，如果一个线程对一个地址多次写，早先的请求会被弃掉，从而省下内存访问代价，这种现象称为写吸收。

写缓冲区的应用会产生一个非常重要的结果：对主存发出的读写访问顺序并不一定与主存中实际发生的顺序一样。例如，回想第1章中对互斥的正确性起着重要作用的标志原则：如果两个处理器各自都先写自己的标志，然后再读对方的标志位，那么其中一个将会看到对方最新写的标志值。若采用写缓冲方式，则该结论不再成立，因为有可能两个处理器都在写，每个写都在它自己的写缓冲区中，此这两个缓冲区有可能在每个处理器都读了对方在内存中的标志位后才被写入。这样两者都没有读到对方的标志。

在编译中则可能出现更为严重的问题。通常，编译器适于在单处理器系统结构上进行性能优化。这种优化往往要求重排单个线程对内存的读写次序。这种重排序对于单线程程序是不可见的，但对多线程程序来说，由于线程可以观察到写发生的顺序，则会产生我们并不希望的结果。例如，如果一个线程将数据装入缓冲区后设置一个指示器以标记缓冲区是否为满，那么并发线程可能在看到新数据之前看到了指示器设置，从而导致它们读到的数据为旧值。在第3章中描述的错误的双重校验上锁模式则是一个由于Java存储器模型的不直观因素所产生错误的例子。

478

不同的系统结构对于内存读写的重排序程度提供了不同的保证。总之，最好是不依赖于这种保证，而是使用下面所描述的代价更高的技术来防止这种重排序。

所有的系统结构都提供强制写操作按照它们产生的次序来执行的能力。但这种方式的代价很高。内存路障指令（有时称为内存栅栏）将刷新写缓冲区，以确保在路障之前产生的所有写操作对于产生路障的处理器是可见的。内存路障往往是通过像getAndSet()这样的原子读-改-写操作或者标准的并发库来透明地插入。因此，只有当处理器对临界区外的共享变量执行读/写指令时，才需要显式地使用内存路障。

一方面，内存路障的代价较高（100个时钟周期或者更多），因此只有在必要时才能使用。另一方面，由于同步问题很难追踪，所以应该宽松地使用内存路障，而不是依靠复杂的特定平台来保障对内存指令重排序的限制。

Java语言本身允许那些发生在synchronized方法或代码块之外的对对象域的读/写操作重排序。Java提供了关键字volatile来保证对synchronized代码块或方法之外的volatile对象域上进行的读/写操作不被重排序。使用这个关键字的代价很高，所以只有在必要时才使用。从原理上来讲，可以使用volatile域来保证双重校验锁算法正常工作，但是可能不存在很多个点，因为无论如何访问volatile变量都需要同步。

到此为止，我们简单地介绍了多处理器硬件的基本知识。在专门的数据结构和算法中将会继续讨论这些系统结构的相关概念。一种新的模式将出现：多处理器程序的性能在很大程度上依赖于和底层硬件的协同配合。

B.8 硬件同步指令

正如第5章中所讨论的那样，任何现代多处理器系统结构都必须能够使功能强大的同步原语成为通用的，也就是说，能提供通用图灵机的并发计算等价形式。因此，在Java语言中，其同步的实现是依赖于这些专门的硬件指令（或称为硬件原语）的，从自旋锁、管程直到最复杂的无锁结构。 479

现代的典型系统结构通常支持两种通用同步原语中的一种。AMD、Intel和Sun的系统结构支持比较和交换（compare-and-swap，CAS）指令。该指令具有三个参数：内存地址a、期望值e和更新值v，返回一个布尔值。它原子地执行下列步骤：

- 如果内存地址a中包含有期望值e，
- 将更新值v写入该地址并返回*true*，
- 否则，保持该内存值不变，并返回*false*。

在Intel和AMD的系统结构中，CAS被称为CMPXCHG，而在SPARC™中被称为CAS。[1] Java的java.util.concurrent.atomic库提供了用compareAndSet()方法实现CAS的原子布尔、整型和引用类。（由于我们的例子中基本上都采用Java语言，所以我们使用cmpareAndSet()而不是CAS。）C#提供了具有相同功能的Interlocked.CompareExchange方法。

CAS指令有一个缺陷，下面是最常使用CAS的情形。一个应用从给定的内存地址读值a，并且为该地址计算出一个新值c。仅当该地址的值a在被应用读后一直未改变，才能将新值c存

[1] SPARC上的CAS返回对应地址的先前值，而不是布尔值，该值用于重试失败的CAS。Intel Pentium上的CMPXCHG能同时有效地返回一个布尔值和先前值。

入。有人可能认为用期望值a和更新值c调用CAS能实现这个目标。然而有一个问题：一个线程有可能用另一个值b覆写了a，随后又将a写入到那个地址中。CAS指令将用c替换掉a，但是这也许并不是应用所预期的结果（例如，如果地址中存放的是指针，而新值a可能是一个回收对象的地址）。CAS调用将用c替换c，但是应用并没有完成它所预期的工作。这种问题称为ABA问题，在第16章中已做了详细讨论。

另一个硬件同步原语是一对指令：加载/链接和存储/条件（load-linked和store-conditional，LL/SC）。LL指令从地址a读数据，随后的SC指令尝试将一个新值存入该地址。若线程对a产生LL指令以来，地址a的内容没有变化则该SC指令成功；若在这段期间a的内容发生了变化，则该SC指令失败。

有一些系统结构支持LL和SC指令：Alpha AXP（ldl_l/stl_c）、IBM PowerPC（lwarx/stwcx）、MIPS（ll/sc）和ARM（ldrex/strex）。LL/SC指令并不受ABA问题所影响，但在
480 实践中，往往对一个线程在LL与对应的SC之间所能做的工作加以限制。上下文切换是另一种LL指令（或另一种加载/存储指令），该指令有可能导致SC指令失败。

保守地使用原子域及其相关方法是一种比较好的办法，因为它们通常是基于CAS或LL/SC的。执行一条CAS或LL/SC指令往往要花费比执行加载或存储指令多得多的时钟周期，它包含内存路障、防止乱序执行以及各种编译器优化。准确的代价取决于许多因素，不仅包含从一种系统结构到另一种系统结构的变化，而且还包含在同一种系统结构中从一种应用到另一种应用的变化。这足以说明CSA或LL/SC要比简单的加载/存储慢得多。

B.9 本章注释

John Hennessy和David Patterson[58]给出了关于计算机系统结构的全面论述。Intel Pentium处理器采用了MESI协议[75]。考虑缓存的程序设计的要点是根据Benjamin Gamsa、Orran Krieger、Eric Parsons和Michael Stumm[43]编写的。Sarita Adve和Karosh Gharachorloo[1]给出了关于内存一致性模型的非常好的综述。

B.10 习题

习题219. 线程A必须等待另一个处理器上的一个线程改变内存中的标志位。调度器可以让A自旋，反复地测试标志位，也可以结束A的调度，允许其他线程运行。假设操作系统将处理器从一个线程切换到另一个线程总共要花10毫秒，如果操作系统放弃调度线程A并立刻重新调度它，则要花费20毫秒。然而，如果A在时刻t_0自旋，标志位在时刻t_1改变，那么操作系统将花费t_1-t_0时间做无用功。

*自测调度器*是一种可以预测将来的调度器。如果它预见到标志将在小于20毫秒的时间内改变，那么它浪费少于20毫秒的时间让A自旋是有意义的，因为放弃调度并重新调度A要花费20毫秒。如果标志位的改变要花费超过20毫秒的时间，那么让另一个线程代替A是有意义的，花费的时间不会超过20毫秒。

你的任务是实现一个调度器，在同样的环境下，它所花费的时间不会超过预测调度器所花费时间的两倍。

习题220. 假设你是一个律师，要为一个特别的观点举出最好的案例。你将如何辩论下面的观点：如果上下文切换的代价可以忽略，那么处理器不需要高速缓存，至少对于那些包含大量线程的

应用来说应该如此。

额外工作：评论你的论证。 481

习题221. 考虑一个具有16个缓存块的直接映射高速缓存，索引值从0～15。每个缓存块包含32个字。

- 用移位和掩码操作来解释如何将一个地址a映射到一个缓存块。假设地址是针对字而不是字节的；地址7指的是内存中的第7个字。
- 对于一个包含64个字的数组上循环4次的程序，计算出该程序的最好和最坏命中率。
- 对于一个包含512个字的数组上循环4次的程序，计算出该程序的最好和最坏命中率。

习题222. 考虑一个具有16个缓存块的直接映射高速缓存，索引为0～15。每个缓存块包含32个字。

考虑一个32×32的二维字数组a。该数组在内存中被作列为a[0, 0]的下一个元素是a[0, 1]，以此类推。假设该高速缓存初始为空，但a[0, 0]被映射到0号缓存块的第一个字。

考虑下面的列优先遍历：

```
int sum = 0;
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    sum += a[i,j];  // 2nd dim changes fastest
  }
}
```

以及下面的行优先遍历：

```
int sum = 0;
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    sum += a[j,i];  // 1st dim changes fastest
  }
}
```

比较两次遍历的缓存缺失个数，假设最早的缓存块被最先替换。

习题223. 在缓存一致性协议MESI中，区分独占和修改模式的优点是什么？

区分独占和共享模式的优点是什么？

习题224. 实现图B-1和图B-2中展示的测试-设置和测试-测试-设置锁，在多处理器上测试它们的相对性能，并分析结果。 482

参考文献

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *STOC '95: Proc. of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, pp. 538–547, NY, USA, 1995, ACM Press.

[4] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). In *PODC '00: Proc. of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, Portland, Oregon, USA, pp. 71–80, NY, USA, 2000, ACM Press.

[5] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *PODC '93: Proc. of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pp. 159–170, NY, USA, 1993, ACM Press.

[6] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. of the Sixteenth International Symposium on Computer Architecture*, pp. 396–406, May 1989.

[7] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[8] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3(1):1–19, 1983.

[9] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pp. 483–485, Atlantic City, NJ, April 1967, Reston, VA, USA, AFIPS Press.

[10] J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[11] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *PODC '95: Proc. of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 184–193, NY, USA, 1995, ACM Press.

[12] J. H. Anderson, M. G. Gouda, and A. K. Singh. The elusive atomic register. Technical Report TR 86.29, University of Texas at Austin, 1986.

[13] J. H. Anderson, M. G. Gouda, and A. K. Singh. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.

[14] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[15] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 119–129, NY,

USA, 1998. ACM Press.

[16] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[17] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI '98: Proc. of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, pp. 258–268, NY, USA, 1998. ACM Press.

[18] K. Batcher. Sorting Networks and Their Applications. In *Proc. of the AFIPS Spring Joint Computer Conference*, 32:307–314, Reston, VA, USA, 1968.

[19] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[20] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[21] H. J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 261–268, NY, USA, 2005. ACM Press.

[22] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93: Proc. of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pp. 41–51, NY, USA, 1993. ACM Press.

[23] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[24] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *PODC '87: Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 222–231, NY, USA, 1987. ACM Press.

[25] C. Busch and M. Mavronicolas. A combinatorial treatment of balancing networks. *Journal of the ACM*, 43(5):794–839, 1996.

[26] T. D. Chandra, P. Jayanti, and K. Tan. A polylog time wait-free construction for closed objects. In *PODC '98: Proc. of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 287–296, NY, USA, 1998. ACM Press.

[27] G. Chapman, J. Cleese, T. Gilliam, E. Idle, T. Jones, and M. Palin. Monty phyton and the holy grail, Motion Picture, Michael White Productions, Released 10 May 1975, USA.

[28] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proc. of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 21–28, NY, USA, 2005. ACM Press.

[29] A. Church. A note on the entscheidungs problem. *Journal of Symbolic Logic*, 1936.

[30] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.

[31] D. Dice. Implementing fast Java monitors with relaxed-locks. *Proc. of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, Monterey, California, p. 13, April 23–24, 2001.

[32] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Proc. of the Twentieth International Symposium on Distributed Computing (DISC*

2006), Stockholm, Sweden, pp. 194–208, 2006.

[33] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, NY, USA, 1968. ACM Press.

[34] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal of Computing*, 26(2):418–455, 1997.

[35] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, 36(4):738–757, 1989.

[36] A. C. Doyle. *A Study in Scarlet and the Sign of Four*. Berkley Publishing Group, NY, 1994. ISBN: 0425102408.

[37] C. Dwork and O. Waarts. Simple and efficient bounded concurrent time-stamping and the traceable use abstraction. *Journal of the ACM (JACM)*, 46(5):633–666, 1999.

[38] C. Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems (TODS)*, 12(2):195–217, 1987.

[39] F. E. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional primitives. *Distributed Computing*, 18(4):267–277, 2006.

[40] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[41] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01 Proc. of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, Monterey, California, 2001. Berkeley, CA, USA, USENIX Association.

[42] K. Fraser. *Practical Lock-Freedom*. Ph.D. dissertation, Kings College, University of Cambridge, Cambridge, England, September 2003.

[43] B. Gamsa, O. Kreiger, E. W. Parsons, and M. Stumm. Performance issues for multiprocessor operating systems. Technical report, Computer Systems Research Institute, University of Toronto, 1995.

[44] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

[45] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large scale cache-coherent multiprocessors. In *Proc. of the Third International Conference on Architectural support for Programming Languages and Operating Systems*, pp. 64–75, 1989, ACM Press.

[46] J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, third edition, Upper Saddle River, New Jersey, USA, 2005. ISBN: 0321246780.

[47] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer-designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[48] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *PODC '02: Proc. of the Twenty-first Annual Symposium on Principles of Distributed Computing*, Monterey, California, pp. 260–269, NY, USA, July 2002, ACM Press.

[49] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proc. of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, pp. 258–264, Las Vegas, NY, USA, 2005, ACM Press.

[50] S. Haldar and K. Vidyasankar. Constructing 1 writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, 1995.

[51] S. Haldar and P. Vitányi. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM (JACM)*, 49(1):101–126, 2002.

[52] P. B. Hansen. Structured multi programming. *Communications of the ACM*, 15(7):574–578, 1972.

[53] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. of Fifteenth International Symposium on Distributed Computing (DISC 2001), Lisbon, Portugal*, volume 2180 of *Lecture Notes in Computer Science*, pp. 300–314, October 2001. Springer-Verlag.

[54] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, USA, pp. 48–60, NY, USA, 2005. ACM Press.

[55] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. *Proc. of the Ninth International Conference on Principles of Distributed Systems (OPODIS 2005)*, Pisa, Italy, pp. 3–16, 2005.

[56] D. Hendler and N. Shavit. Non-blocking Steal-half Work Queues. In *Proc. of the Twenty-first Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Monterey, California, pp. 280–289, 2002. ACM Press.

[57] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proc. of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 206–215, NY, USA, 2004. ACM Press.

[58] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1995.

[59] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1): 1–17, 0885-7458 1988.

[60] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[61] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *ICDCS '03: Proc. of the Twenty-third International Conference on Distributed Computing Systems*, p. 522, Washington, DC, USA, 2003. IEEE Computer Society.

[62] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[63] M. Herlihy and N. Shavit. On the nature of progress. In *Principles of Distributed Systems—15th International Conference, OPODIS 2011*, Toulouse, France, pp. 313–328, December 13–16, 2011.

[64] M. Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[65] M. Herlihy, B.-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[66] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *POUC '03, Proc. of*

the Twenty-second Annual Symposium on Principles of Distributed Computing, Boston, Massachusetts, pp. 92–101, NY, USA, 2003, ACM Press.

[67] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the Twentieth Annual International Symposium on Computer Architecture*, pp. 289–300, San Diego, California, 1993, ACM Press.

[68] M. Herlihy, N. Shavit, and M. Tzafrir. Concurrent cuckoo hashing. Technical report, Providence RI, Brown University, 2007.

[69] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[70] C. A. R. Hoare. "partition: Algorithm 63," "quicksort: Algorithm 64," and "find: Algorithm 65." *Communications of the ACM*, 4(7):321–322, 1961.

[71] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[72] M. Hsu and W. P. Yang. Concurrent operations in extendible hashing. In *Symposium on Very Large Data Bases*, pp. 241–247, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[73] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proc. of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pp. 627–631, 1983.

[74] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60(3):151–157, 1996.

[75] Intel Corporation. *Pentium Processor User's Manual*. Intel Books, 1993. ISBN: 1555121934.

[76] A. Israeli and L. Rappaport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94: Proc. of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, California, United States, pp. 151–160, NY, USA, August 14–17 1994, ACM Press.

[77] A. Israeli and M. Li. Bounded time stamps. *Distributed Computing*, 6(5): 205–209, 1993.

[78] A. Israeli and A. Shaham. Optimal multi-writer multi-reader atomic register. In *PODC '92: Proc. of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, pp. 71–82, NY, USA, 1992, ACM Press.

[79] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[80] P. Jayanti. A lower bound on the local time complexity of universal constructions. In *PODC '98: Proc. of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 183–192, NY, USA, 1998, ACM Press.

[81] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *WDAG '92: Proc. of the Sixth International Workshop on Distributed Algorithms*, pp. 69–84, London, UK, 1992, Springer-Verlag.

[82] D. Jiménez-González, J. Larriba-Pey, and J. Navarro. CC-Radix: A cache conscious sorting based on Radix sort. In *Proc. Eleventh Euromicro Confer-*

ence on Parallel, Distributed, and Network-Based Processing, pp. 101–108, 2003. ISBN: 0769518753.

[83] L. M. Kirousis, P. G. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *IEEE Trans. Parallel Distributed Systems*, 5(7): 688–696, Piscataway, NJ, USA, 1994. IEEE Press.

[84] M. R. Klugerman. Small-depth counting networks and related topics. Technical Report MIT/LCS/TR-643, MIT Laboratory for Computer Science, 1994.

[85] M. Klugerman and C. Greg Plaxton. Small-depth counting networks. In *STOC '92: Proc. of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, pp. 417–428, NY, USA, 1992. ACM Press.

[86] D. E. Knuth. *The Art of Computer Programming: Second Ed. (Addison-Wesley Series in Computer Science and Information)*. Boston, MA, USA, 1978 Addison-Wesley Longman Publishing Co., Inc.

[87] V. Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6):681–694, 1990.

[88] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(5):543–545, 1974.

[89] L. Lamport. Time, clocks, and the ordering of events. *Communications of the ACM*, 21(7):558–565, July 1978.

[90] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[91] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[92] L. Lamport. Invited address: Solved problems, unsolved problems and non-problems in concurrency. In *Proc. of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 1–11, 1984. ACM Press.

[93] L. Lamport. The mutual exclusion problem—Part I: A theory of interprocess communication. *Journal of the ACM (JACM)*, 33(2):313–326, 1986, ACM Press.

[94] L. Lamport. The mutual exclusion problem—Part II: Statement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

[95] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[96] B. Lampson and D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 2(23):105–117, 1980.

[97] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, San Francisco, 2006.

[98] D. Lea. Java community process, JSR 166, concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, 2003.

[99] D. Lea. Concurrent hash map in JSR 166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`. Dec 2007.

[100] D. Lea, Personal Communication, 2007.

[101] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. *J. Parallel Distributed Computing*, 62(4):656–668, 2002.

[102] C. E. Leiserson and H. Prokop. A minicourse on multithreaded programming, Massachusetts Institute of Technology, Available on the Internet

from http://theory.lcs.mit.edu/~cilck, 1998. citeseer.ist.psu.edu/leiserson98minicourse.html.

[103] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. Fourteenth Colloquium on structural information and communication complexity (SIROCCO) 2007 pp. 124–138, June 5–8, 2007, Castiglioncello (LI), Italy.

[104] M. Li, J. Tromp, and P. M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[105] B.-H. Lim. Personal Communication, Cambridge, Massachusetts, 1995.

[106] W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: wait-free hierarchies are not robust. In *STOC '97: Proc. of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, pp. 579–588, NY, USA, 1997, ACM Press.

[107] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 263–268, Cancun, Mexico, 2000.

[108] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, CT, 1987.

[109] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proc. of the European Conference on Parallel Computing (EuroPar 2006)*, pp. 801–810, Dresdan, Germany, 2006.

[110] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the Eighth International Symposium on Parallel Processing (IPPS)*, pp. 165–171, April 1994. IEEE Computer Society, April 1994. Vancouver, British Columbia, Canada, NY, USA, 1987, ACM Press.

[111] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proc. of the Thirty-second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 378–391, NY, USA, 2005, ACM Press.

[112] P. E. McKenney. Selecting locking primitives for parallel programming. *Communications of the ACM*, 39(10):75–82, 1996.

[113] J. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[114] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proc. of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 73–82. Winnipeg, Manitoba, Canada, NY, USA, 2002, ACM Press.

[115] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, 1996, ACM Press.

[116] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):142–153, 1986.

[117] M. Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proc. of the Sixteenth Annual ACM Symposium*

on Principles of Distributed Computing, pp. 219–228, NY, USA, 1997, ACM Press.

[118] M. Moir. Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. In *PODC '00: Proc. of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 61–70, NY, USA, 2000, ACM Press.

[119] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA '05: Proc. of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 253–262, NY, USA, 2005, ACM Press.

[120] M. Moir V. Marathe and N. Shavit. Composite abortable locks. In *Proc. of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–10, 2006.

[121] I. Newton, I. B. Cohen (Translator), and A. Whitman (Translator). *The Principia: Mathematical Principles of Natural Philosophy*. University of California Press, CA, USA, 1999.

[122] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[123] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[124] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[125] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[126] S. A. Plotkin. Sticky bits and universality of consensus. In *PODC '89: Proc. of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pp. 159–175, NY, USA, 1989, ACM Press.

[127] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989.

[128] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[129] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. Lecture Notes in Computer Science. Distributed Computing. In *DISC*. Springer Berlin/Heidelberg, pp. 108–121, 2005.

[130] Z. Radović and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Ninth International Symposium on High Performance Computer Architecture*, pp. 241–252, Anaheim, California, USA, February 2003.

[131] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, Cambridge, MA, 1986.

[132] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[133] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture, ISCA '84*, pp. 340–347. ACM, New York, NY, USA, 1984.

[134] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proc. of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 237–245, July 1991. ACM Press.

[135] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus—making resilient algorithms fast in practice. In *SODA '91: Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 351–362, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[136] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP '06: Proc. of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 147–156, NY, USA, 2006. ACM Press.

[137] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proc. of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, pp. 240–248, NY, USA, 2005. ACM Press.

[138] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *PODC '02: Proc. of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pp. 31–40, NY, USA, 2002. ACM Press.

[139] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. *ACM SIGPLAN Notices*, 36(7):44–52, 2001.

[140] M. Sendak. *Where the Wild Things Are*. Publisher: HarperCollins, NY, USA, 1988. ISBN: 0060254920.

[141] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *Journal of the ACM*, 53(3):379–405, NY, USA, 2006. ACM Press.

[142] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing*, Special Issue 10(2):99–116, 1997.

[143] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[144] E. Shenk. The consensus hierarchy is not robust. In *PODC '97: Proc. of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, p. 279, NY, USA, 1997. ACM Press.

[145] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986. San Jose, CA.

[146] A. Turing. On computable numbers, with an application to the entscheidungs problem. *Proc. Lond. Math. Soc*, Historical document, 1937.

[147] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proc. of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 214–222. Ottowa, Ontario, Canada, NY, USA, 1995. ACM Press.

[148] P. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Twenty-seventh Annual Symposium on Foundations of Computer Science*, pp. 233–243, Los Angeles, CA, USA, October 1986. IEEE Computer Society Press.

[149] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989.

[150] R. N. Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *PODC '87: Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 232–248, NY, USA, 1987. ACM Press.

[151] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

ABA problem（ABA问题）
  basic scenario（基本场景），235
  load-linked-store-conditional（链接加载/条件存储），237
  and memory reclamation（内存回收），233-237
  Abort, memory transactions（中止，内存事务），422
Abstract（抽象的）
  BaseHashSet class（BaseHashSet类），301
  concurrent Cuckoo hashing（并发Cuckoo哈希），318
  contention manager（争用管理器），432
Abstraction map（抽象映射）
  concurrent reasoning（并发推理），199
  LockFreeList class（LockFreeList类），216
Abstract value, concurrent reasoning（抽象值，并发推理），198
Acquires（获得）
  CLHLock class（CLHLock类），154
  CompositeLock class（CompositeLock类），161
  definition（定义），23
  FineList class（FineList类），203
  HCLHLock lock（HCLHLock锁），173
  Java concepts（Java概念），456
  locks（锁），178
  MCSLock class（MCSLock类），156
Active thread（主动线程）
  in software combining（软件组合中），260
  termination detection barrier（终止检测障碍），405
Addressing（寻址）
  closed-address hash sets（封闭地址哈希集），300-302
  concurrent closed-addressing（基于封闭地址的并发），325
  definitions（定义），300
  hardware concepts（硬件概念），473
  open-addressed hash set（开放地址哈希集），316-318
Algorithms（算法）
  Bakery lock（Bakery锁），31-33
  bitonic sorting（双调排序），288-289
  concurrent（并发），2，15
  Dynamic Software Transactional Memory（动态软件事务内存），448
  fast path（快速路径），43
  Lock algorithm（Lock算法），24，37-40，38
  lock-based concurrent skiplist（基于锁的并发跳表），333-339
  lock-free concurrent skiplist（无锁并发跳表），341-348
  lock-free universal（无锁通用），127
  lock-free universal construction（无锁通用构造），128
  quicksort（快速排序），290
  SkipList class（SkipList类），349
  Transactional Locking（事务锁），2，448
  TTASLock，147-149
  wait-free universal construction（通用无等待构造），131
Amdahl's Law（Amdahl定律）
  definition（定义），13
  in parallelization（并行），13-14
Announce event, wait-free universal construction（声明事件，通用无等待构造），130-132
Anonymous inner class（匿名内部类）
  Java thread concepts（Java线程概念），454
  software transactional memory（软事务内存），426

Anonymous method, C# concepts (匿名方法, C#概念), 460
Antisymmetric, timestamps (反对称的, 时间戳), 34
Array-based bounded priority queues, implementation (基于数组的有界优先级队列, 实现), 352-353
Array-based locks (基于数组的锁)
  implementation (实现), 150-151
  without false sharing (无假共享), 152
Asynchronous (异步)
  definition (定义), 1
  threads (线程), 71
Atomic hardware step (原子硬件步)
  compare-and-swap (比较和交换), 480
  shared counter implementation (共享计数器实现), 5-6
Atomicity and transactions (原子性和事务), 421-423
AtomicMarkableReference class function (AtomicMarkableReference类函数), 234
  function and methods (函数和方法), 213-215
AtomicMRSWRegister class, implementation (AtomicMRSWRegister类, 实现), 83
Atomic objects (原子对象)
  implementation (实现), 433-434
  lock-based (基于锁的), 438-445
  obstruction-free (无干扰), 434-438
  software transactional memory (软事务内存), 429-431
Atomic primitives, problems (原子原语, 问题), 418-420
AtomicReference class function (AtomicReference类函数), 236
  unbounded lock-free queue (无锁无界队列), 231
Atomic register (原子寄存器)
  for consensus problem (一致性问题), 103-105
  definition (定义), 73
  first definition (首次定义), 93
  history (经历), 65
  implementation considerations (实现问题), 75-76
  MRMW (多读者-多写者), 85-87
  MRSW (多读者-单写者), 82-85
  SRSW (单读者-单写者), 77-78, 81-82, 83
Atomic snapshots (原子快照)
  construction (构造), 93
  correctness arguments (正确论点), 90-93
  definition (定义), 87
  and multiple assignment (多重赋值), 110
  obstruction-free (无干扰), 87-88
  wait-free snapshot (无等待快照), 88-90
AtomicSRSWRegister class, implementation (AtomicSRSWRegister类, 实现), 83
AtomicStampedReference class (AtomicStampedReference类)
  bounded work-stealing dequeue (有界工作窃取队列), 384
  function (函数), 234
Atomic step (原子步骤), 456
Auxiliary variables, wait-free universal construction (辅助变量, 通用无等待构造), 133-134

B

Backoff lock (后退锁)
  contention manager (争用管理), 432
  hierarchical (层次), 167-168
BackoffLock class (BackoffLock类)
  implementation (实现), 148
  problems (问题), 149
Bakery lock (Bakery锁)
  bounds (界限), 39
  in mutual exclusion (互斥), 31-33
  as spin locks (自旋锁), 141
Balancer (平衡器)
  counting networks (计数网), 270
  operation (操作), 271
  sorting networks (排序网), 286-287
Balancer class, software bitonic counting network (Balancer类, 软件双调计数网), 275
Balancing, work (平衡, 工作), 389-391
Balancing network, construction (平衡网, 构造), 271-272
Barriers (障碍)

combining tree（组合树），400-402
definition（定义），398
implementation（实现），398-399
sense-serving（语义服务），399-400
static tree（静态树），402-404
supporting concepts（支持概念），397-398
survey（调查），408
BaseHashSet class（BaseHashSet类）
implementation（实现），300-301
methods（方法），302
thresholds（阈值），301
Batching, definition（批处理，定义），478
Benevolent side effect, lock-free lists（慈善副作用，无锁链表），217
Bitonic class, implementation（Bitonic类，实现），276
Bitonic counting network（双调计数网）
basic concepts（基本概念），273-274
proof of correctness（正确性证明），276-278
recursive structure（递归结构），274
sequential execution（顺序执行），272
software implementation（软件实现），274-276
Bitonic sorting algorithm, design and function（双调排序算法，设计和函数），288-289
Bivalence, protocol state（二价，协议状态），102
Blocking（阻塞）
concurrent objects（并发对象），59
definition（定义），45，141
in Java memory model（Java存储器模型），62-63
locks（锁），178
Block network, implementation（Block网，实现），281
Boolean register（布尔寄存器）
atomic SRSW（原子SRSW），81-82
definition（定义），72
MRSW，77-78
safe MRSW（安全MRSW），78-79
safe SRSW（安全SRSW），86
Bottom wires（底线）
counting networks（计数网），270
sorting networks（排序网），286
Bouncer class（Bouncer类）
array layout（数组布局），44
implementation（实现），43
Bounded, pool（有界的，池），223
BoundedDEQueue class, implementation（BoundedDEQueue类，实现），382-384
Bounded partial queue considerations and drawbacks（部分有界队列问题和缺陷），228-229
implementation（实现），225-227
BoundedQueue class, C# constructs（BoundedQueue类，C#结构），462-463
fields and constructor（域和构造函数），225
list node（链表结点），227
methods（方法），226-227
Bounded-range priority queue（有界优先级队列）
array-based（基于数组），352-353
definition（定义），351
tree-based（基于树），353-355
Bounded transactional queue, implementation（有界事务队列，实现），423
Bounded wait-free method, concurrent objects（有界的无等待方法，并发对象），59
Bounded work-stealing double-ended queues（有界工作窃取双端队列）
creator（生成器），392
implementation（实现），382-386
Bucket（桶）
BaseHashSet class（BaseHashSet类），301
definition（定义），300
split-ordered hash set（有序划分哈希集），311
BucketList class, implementation（BucketList类，实现），312-313
Bucket threshold, BaseHashSet class（桶阈值，BaseHashSet类），301
Buffer, between producers and consumers（缓冲区，生产者和消费者之间），223
Bursty, producers（突发，生产者），223
Bus（总线）
cache coherence（缓存一致性），446
definition（定义），145
Bus controller（总线控制器），472
Busy-waiting, definition（忙等待，定义），141

C

Cache（高速缓存）
  definition（定义），146,446
  hardware concepts（硬件概念），471,473-474
Cache blocks, definition（缓存块，定义），474
Cache coherence（缓存一致性）
  as BackoffLock problem（BackoffLock问题），149
  hardware concepts（硬件概念），474-476
  hardware transactional memory（硬事务内存），446-447
Cache-coherent NUMA（缓存一致的NUMA），474
Cache-conscious programming（高速缓存的程序设计问题），476-477,481
Cache hit, definition（高速缓存命中，定义），146
Cache lines, definition（缓存行，定义），474
Cache miss, definition（高速缓存缺失，定义），146
Callable object, definition（Callable对象，定义），371
Calls（调用）
  atomic methods（原子方法），423
  successful or unsuccessful（成功的或不成功的），196
Capacity（容量）
  BaseHashSet class（BaseHashSet类），301
  pool（池），223
  semaphore（信号量），189
CAS, 参见Compare-and-swap (CAS) instruction
Church-Turing Thesis（丘奇-图灵论题），71
Circular array, modulo, unboundedDEQueue class（循环数组，取模，unboundedDEQueue类），46,150,387
Clean double collect, obstruction-free snapshot（干净的双重收集，无障碍快照），88
CLHLock lock, creators（CLHLock锁，生成器），174
CLH queue lock（CLH队列锁）
  fields and constructor（域和构造函数），151,153
  hierarchical（分层），168-172
  lock acquisition and release（获取锁和释放锁），154
  methods（方法），153
Closed-address hash sets, basic concepts（封闭地址哈希集，基本概念），300-302
Closed addressing, definition（封闭地址，定义），300
Cluster id, hierarchical locks（集群id，层次锁），167
Coarse-grained hash set, implementation（粗粒度哈希集，实现），302-303
Course-grained synchronization（粗粒度同步）
  definition（定义），195
  implementations（实现），200-201
CoarseHashSet class, implementation（CoarseHashSet类，实现），302-303
CoarseList class, methods（CoarseList类，方法），200-201
Collects（采集）
  obstruction-free snapshot（无障碍快照），87-88
  single writer atomic snapshot class（单写者原子快照类），91
Collisions, hash sets（冲突，哈希集），300
Combining, software（组合，软件），260-261
Combining phase, CombiningTree（组合阶段，组合树），265
Combining status, definition（组合状态，定义），261
Combining tree barrier（组合树障碍）
  creators（生成器），408
  implementation（实现），400-402
CombiningTree class（CombiningTree类）
  combining status（组合状态），261-262
  constructor（构造函数），262
  disadvantages（劣势），261
  distribution phase（分布阶段），267
  execution phases（执行阶段），263,267
  methods（方法），264
  Node class（Node类），265-267
  performance（性能），269
  precombining phase（预组合阶段），262
  robustness（健壮性），269
  status（状态），268

Combining trees, original idea (组合树，原始方案), 292
Common2 register (Common2寄存器)
  definition (定义), 117
  RMW, 114-116
Communication, and mutual exclusion (通信，互斥), 9-10
Comparator, comparison network (比较器，比较网络), 286
compareAndSet() operation (compareAndSet()操作)
  hardware synchronization (硬件同步), 480
  synchronization primitives (同步原语), 116-117
Compare-and-swap (CAS) instruction, hardware considerations (CAS指令，考虑硬件), 480
Comparison network, definition (比较网络，定义), 286
CompositeFastPathLock class, methods (CompositeFastPathLock类，方法), 166
Composite lock (复合锁)
  advantages (优势), 159-160
  fast path (快速路径), 165-167
CompositeLock class (CompositeLock类)
  execution (执行), 164
  fields, constructor, and methods (域，构造函数，方法), 160
  methods (方法), 161-163
  properties (属性), 165
  QNode class (QNode类), 161
Compositional, correctness property (复合，正确性), 51
Compositionality, problems (复合性，问题), 420-421
Compositional linearizability,
concurrent objects (复合的可线性化性，并发对象), 57-58
Concrete representation, concurrent reasoning (具体表示，并发推理), 198
Concurrency (并发性)
  concurrent objects (并发对象), 45-48
  reasoning (推理), 198-200
Concurrent algorithm (并发算法)
  challenges (挑战), 15
  definition (定义), 2
Concurrent closed-addressing schemes creators (并发封闭地址策略，生成器), 325
Concurrent Cuckoo hashing, definition and implementation (并发Cuckoo哈希，定义和实现), 318-322
Concurrent heap (并发堆)
  overview (综述), 357-358
  structure and implementation (结构和实现), 358-363
Concurrent objects (并发对象)
  compositional linearizability (复合的可线性化性), 57-58
  concurrency (并发), 45-48
  correctness (正确性), 45-48
  dependent progress conditions (相关演进条件), 60-61
  formal definitions (形式化定义), 55-57
  linearizability (可线性化性), 54-55, 57
  nonblocking property (非阻塞特性), 58-59
  progress conditions (演进条件), 59-60
  quiescent consistency (静态一致性), 49-51
  sequential consistency (顺序一致性), 51-54
  sequential objects (顺序对象), 48-49
Concurrent priority queues, implementation (并发优先级队列，实现), 351-352
Concurrent program (并发程序)
  definition (定义), 2
  synchronization universality hierarchy (同步操作的通用层次结构), 126
Concurrent shared memory computing, definition (并发共享存储器计算，定义), 71
Concurrent skiplists, overview (并发跳表，概述), 348
Concurrent timestamping, Lock class (并发时间戳，Lock类), 34
Condition field, bounded partial queues (条件域，部分有界队列), 225
Conditions objects (Conditions对象)
  interface (接口), 180
  interrupts (中断), 179

LockedQueue class (LockedQueue类), 182
lost wakeups (唤醒丢失), 181-183
usage example (范例用法), 179-180
Consensus numbers (一致数)
definition (定义), 100
interface and definitions (接口和定义), 100-101
states and valence (状态和价), 101-103
Consensus object (一致性对象)
interface (接口), 100
lock-free universal construction (无锁通用构造), 126-130
universality definition (通用性定义), 126
wait-free universal construction (无等待通用构造), 130-136
Consensus synchronization problem (一致同步问题)
atomic register solution (原子寄存器方案), 103-105
Common2 registers (Common2寄存器), 114-116
consensus protocols (一致性协议), 106
definition (定义), 100
FIFO queues (先进先出队列), 106-109
Consensus protocols, generic (一致性协议, 通用), 106
Consistency, software transactional memory (一致性, 软事务内存), 428-429
Consumers (消费者)
naive synchronous queue (基本同步队列), 237
pools (池), 223
Contention (争用)
definition (定义), 147
high and low (高和低), 147
LockFreeStack, 248
Contention manager (争用管理器)
greedy and karma (贪婪和因果), 448
implementation (实现), 433
software transactional memory (软事务内存), 431-433
Context switch, definition (上下文切换, 定义), 472
Convoying, in locking (转让, 加锁), 417
Coordination protocol (OR Protocol),definition (协作协议, 定义), 6
Copyable class, interface (Copyable类, 接口), 430
Correctness (正确性)
bitonic counting network (双调计数网), 276-278
compositional (复合), 51
concurrent objects (并发对象), 45-48
Merger class (Merger类), 277
Correctness arguments, atomic snapshots (正确性论据, 原子快照), 90-93
Counters (计数器)
definition (定义), 22
quiescent consistency (静态一致性), 269-270
as shared counter implementation (作为共享计数器实现), 4-5
Counting, shared (计数, 共享), 259-260
Counting networks (计数网)
basic concepts (基本概念), 270
bitonic (双调), 272-278
components (组件), 270-273
invention (发明), 292
performance (性能), 280-281
periodic (周期), 278-280
pipelining (流水线), 280-281
Covering state, Lock algorithm (覆盖状态, Lock算法), 38, 40
Critical-path length, parallelism (关键路径长度, 并行), 376-377
Critical sections (临界区)
as BackoffLock problem (BackoffLock问题), 149
Java concepts (Java概念), 456
in mutual exclusion (互斥), 22-24
C# construct concepts (C#构造概念)
creators (生成器), 466
monitors (管程), 461-462
thread-local objects (线程本地对象), 462-463
threads (线程), 460-461
Cuckoo hashing (Cuckoo哈希)
creators (生成器), 325
definition and implementation (定义和实现), 316-318

D

DAG（无环有向图），参见Directed acyclic graph
Data, and memory（数据和内存），473
Data structure design（数据结构设计）
  dual data structures（双重数据结构），239-241
  SkipList class（SkipList类），230
Deadlock（死锁）
  avoidance（避免），417-418
  FineList class（FineList类），204
  freedom from deadlock（无死锁），24
  software transactional memory（软事务内存），431
  unbounded total queue（无界全局队列），230
Deadlock-freedom property（无死锁特性）
  definition（定义），8
  as dependent progress condition（相关演进条件），60
  Filter lock（Filter锁），31
Decision value, consensus numbers（决策值，一致数），101
Delegate, C# concepts（委托，C#概念），460
DEQueue（双端队列），参见Double-ended queue
Deschedule, threads（退出调度，线程），472
DiffractingBalancer class（DiffractingBalancer类）
  implementation（实现），284
Diffracting trees（衍射树）
  basic concept（基本概念），282
  DiffractingBalancer, 284
  implementation（实现），285
  performance（性能），285
  Prism, 283-284
Directed acyclic graph (DAG, 无环有向图)
  creators（生成器），391
  definition（定义），375
  Fibonacci sequence（斐波那契序列），375-376
  steps（步骤），380
Direct mapped cache, definition（直接映射高速缓存，定义），474
Dirty, definition（脏，定义），478
Disjoint-access-parallelism（不相交的并行访问）
  definition（定义），299
Dissemination barrier, creators（分散式障碍，生成器），408
Distributed coordination, overview（分布式协作，概述），291-292
Distribution phase, CombiningTree（分布阶段，组合树），267
Doorway section, in Lock class（门廊区，Lock类），31
Double checked locking（双重检验上锁）
  in Java memory model（在Java存储器模型中），61-62
  and memory consistency（内存一致性），479
Double-ended queue（DEQueue, 双端队列）
  bounded work-stealing（有界工作窃取），382-386, 391
  unbounded work-stealing（无界工作窃取），386-388, 391
  work stealing（工作窃取），380-389
Down state, balancers（向下状态，平衡器），271
Dual data structures（双重数据结构）
  definition（定义），239
  implementation（实现），239-241
  reservation（保留），238-239
Dynamic Software Transactional Memory algorithm, creators（动态软事务内存，算法，生成器），448

E

Elimination, LockFreeStack（消除，无锁栈），248-249
Elimination array, implementation（消除数组，实现），251-252, 255
EliminationBackoffStack class（EliminationBackoffStack类）
  basic concept（基本概念），249
  efficiency（效率），255
  elimination array（消除数组），251-254
  lock-free exchanger（无锁交换机），249-251
  methods（方法），253
  structure（结构），248
EventListener class, incorrect example（EventListener类，错误例子），64
Events, in state machine（事件，状态机），21

Exclusive state, cache coherence (互斥状态，缓存一致性), 446, 475-476
Executor service, definition (执行者服务，定义), 371
Expected value, compareAndSet()Operation (期望值，compareAndSet()操作), 116, 480
Exponential backoff (指数后退)
  implementation (实现), 149
  TTASLock algorithm (TTASLock算法), 147-149
Extensible hashing, definition (可扩展哈希，定义), 300

F

Factory, Lock interface (工厂，锁接口), 178
Fairness (公平性)
  in mutual exclusion (互斥), 31
  pools (池), 224
Fair readers-writers lock (公平的读者-写者锁)
  fields (域), 185
  fields and public methods (域和公共方法), 186
  inner read lock (内部读锁), 186
  inner write lock (内部写锁), 187
False sharing (假共享) in ALock (在ALock中), 152
  array-based locks (基于数组的锁), 151
  occurrence (发生), 476
Fault-tolerance, in mutual exclusion (容错，互斥), 9
Fence, definition (栅栏，定义), 479
Fibonacci sequence (斐波那契序列)
  definition (定义), 375
  directed acyclic graph (无环有向图), 375-376
  FibTask class (FibTask类)
  implementation (实现), 375
FIFO queue, lock-based (先进先出队列，基于锁的), 45-48
Filter lock (Filter锁)
  in mutual exclusion (互斥), 28-31
  as spin locks (自旋锁), 141
final fields, in Java memory model (final域，Java存储器模型), 63-64
Final state, consensus numbers (结束状态，一致数), 101
FineGrainedHeap class (FineGrainedHeap类)
  creators (生成器), 366
  implementation (实现), 358-361
  overview (概述), 357-358
  structure (结构), 362
Fine-grained synchronization (细粒度同步)
  basic concepts (基本概念), 201-202
  definition (定义), 195
FineList class (FineList类)
  deadlock (死锁), 204
  hand-over-hand locking (交叉手上锁), 204
  lock acquisitions (获取锁), 203
  methods (方法), 202-203
FIRST, CombiningTree, 264-265
First-come-first-served (先到先服务)
  Bakery lock algorithm (Bakery锁算法), 31, 33
  definition (定义), 31
First-in-first-out (FIFO, 先进先出)
  for consensus problem (一致性问题), 106-108
  pool fairness (公平性的池), 224
  via Pthreads (通过Pthreads), 466
  quiescent consistency (静态一致性), 51
  sequential objects (顺序对象), 48-49
FIRST status (FIRST状态)
  CombiningTree, 268
  definition (定义), 261
Frames, definition (框架，定义), 397
Freedom from deadlock property, in Lock algorithm (无死锁特性，锁算法), 24
Freedom from starvation, in Lock algorithm (无饥饿，锁算法), 24
Free list, for node recycling (空闲链表，循环结点), 233-234
FreeObject class, structure (FreeObject类，结构), 436-438
Fully-associative caches, definition (全关联缓存，定义), 448, 474
Future interface, definition (Future接口，定义), 371

G

Ghost variables, 参见Auxiliary variables

Global queue, in HCLHLock queue (全局队列, HCLHLock队列), 168
Global state, definition (全局状态, 定义), 37
Global threshold, BaseHashSet class (全局阈值, BaseHashSet类), 301
Granularity, and cache (粒度, 缓存), 474
Greedy (贪婪)
  contention manager (争用管理器), 432
  schedule (调度), 379
Greedy contention manager, creators (贪婪争用管理器, 生成器), 448

H

Hand-and-over-hand locking, FineList class (交叉手上锁, FineList类), 204
Handlers, software transactional memory (处理程序, 软事务内存), 428
Hardware concepts (硬件概念)
  architectures (系统结构), 477-479
  basic considerations (基本考虑), 469-472
  cache-conscious programming (高速缓存的程序设计问题), 476-477
  caches (缓存), 473-474
  coherence (一致), 474-476
  interconnect (互连), 472-473
  memory (主存), 473
  processors and threads (处理器和线程), 472
  spinning (自旋), 476
  synchronization instructions (同步指令), 480-481
Hardware transactional memory (HTM, 硬事务内存)
  cache coherence (缓存一致), 446-447
  enhancements (改进), 447-448
  first proposal (第一次提出), 448
  overview (概述), 445-446
  transactional cache coherence (事务缓存一致性), 447
Hash function, definition (哈希函数, 定义), 300
Hashing, definition (哈希, 定义), 299
Hash sets (哈希集)
  closed-address (封闭地址), 300-302
  definition (定义), 299-300
  resizing (调整), 305
HBOLock class, implementation (HBOLock类, 实现), 168
HCLHLock queue (HCLHLock 队列)
  acquisition and release (获得和释放), 173
  components (组件), 168
  fields and constructor (域和构造函数), 169
  global queue (全局队列), 171
  inner QNode class (内部QNode类), 169
  methods (方法), 170, 172
Head, definition (头部, 定义), 224
Hierarchical backoff lock, implementation (层次后退锁, 实现), 167-168
Hierarchical CLH queue lock, design (层次CLH队列锁, 设计), 168-172
Hierarchical locks, definition (层次锁, 定义), 167
High contention, definition (高争用, 定义), 147
Hit rate, and cache (命中率, 缓存), 474
Hit ratio, and cache (命中率比, 缓存), 474
Hits, in cache (命中, 缓存), 474
Hold, locks (持有, 锁), 178
Hot spot, definition (热点, 定义), 259-260
HTM, 参见Hardware transactional memory

I

IDLE, CombiningTree, 264
Idle step, scheduler (空闲步, 调度程序), 379
Inactive thread, termination detection barrier (非活动线程, 终止检测障碍), 405
Initial state, consensus numbers (初始状态, 一致数), 101
Inner classes (内部类)
  anonymous (匿名), 426
  definition (定义), 183
  tree nodes (树结点), 402
Inner read lock (内部读锁)
  fair readers-writers lock (公平的读者-写者锁, 187
  simple readers-writers lock (简单的读者-写者锁), 184
Inner write lock (内部写锁)
  fair readers-writers lock (公平的读者-写者锁),

187
simple readers-writers lock（简单的读者—写者锁），185
In-place array sorting, function（内置数组排序，函数），288
Instantaneous events, in mutual exclusion（瞬时事件，互斥），22
Interconnect, hardware concepts（互连，硬件概念），471, 472-473
Interference（干扰）
concurrent reasoning（并发推理），198
optimistic synchronization（乐观同步），207
Interrupts（中断）
Conditions objects（Conditions对象），179
definition（定义），9
Invalid state, cache coherence(无效状态，缓存一致)，446, 475-476
Invariants, concurrent reasoning（不变量，并发推理），198
Invocation events（调用事件）
concurrent objects（并发对象），56
lock-free universal algorithm（通用无锁算法），127
and register space（寄存器空间），76
Irreflexive, timestamps（反自反的，时间戳），34
Items（数据项）
hash sets（哈希集），300
priority queue（优先级队列），351
PrioritySkipList class（PrioritySkipList类），363
set definition（集合定义），196
software transactional memory（软事务内存），426

J

Java construct concepts（Java构造概念）
creator（生成器），466
monitors（管程），454-458
thread-local objects（本地线程对象），458-459
threads（线程），453-454
yielding and sleeping（屈从和睡眠），458
Java memory model（Java存储器模型）
basic principles（基本原则），61-62
final fields（final域），63-64
locks and synchronized blocks（锁和同步块），62-63
volatile fields（volatile域），63
Join（连接）
C# constructs（C#构造），460
Join, Java threads（连接，Java线程），454

K

Karma contention manager（Karma争用管理器）
characteristics（特性），432
creators（生成器），448
Kernel maps, function（核映射，函数），378
k-way set associative, caches（k路组相联，缓存），474

L

Labels（标志）
Bakery lock algorithm（Bakery锁算法），32
Timestamps（时间戳）as, 34
Last-in-first-out (LIFO), StackT class（后进先出，StackT类），245
Latency, definition（延迟，定义），259
Layer network, implementation（Layer网实现），279-280
Layout, distributed coordination（布局，分布式协作），292
Lazy（惰性）
PrioritySkipList class（PrioritySkipList类），363
unbounded lock-free queue（无锁的无界队列），231-232
LazyList class（LazyList类）
fields（域），214
linearizability（可线性化性），212
methods（方法），209-210
validation（确认），209, 211
LazySkipList class（LazySkipList类）
creators（生成器），348
implementation（实现），333-337, 339
overview（概述），331-333
Lazy synchronization（惰性同步）
advantages（优点），213

basic concepts（基本概念），208-209
creators（生成器），219
definition（定义），196
linearization（线性化），211-212
methods（方法），209-210
Levels, in Filter lock（级别，Filter锁），28
Line, cache coherence（线，缓存一致），446
Linearizability（可线性化性）
applications（应用），45
concurrent objects（并发对象），54-55, 57
concurrent priority queues（并发优先级队列），351
concurrent reasoning（并发推理），199
first definition（首次定义），93
LazyList class（LazyList类），212
wait-free universal construction（通用的无等待构造），133
Linearization points（线性化点）
concurrent objects（并发对象），55
fine-grained synchronization（细粒度同步），204
LazyList class（LazyList类），211
Linear speedup, parallelism（线性加速比，并行度），377
Linked lists（链表）
coarse-grained synchronization（粗粒度同步），200-201
concurrent reasoning（并发推理），198-200
early work（早期工作），219
fine-grained synchronization（细粒度同步），201-205
lazy synchronization（惰性同步），208-213
list-based sets（基于链表的集合），196-198
lock-free list（无锁链表），213-218
optimistic synchronization（乐观同步），205-208
overview（概述），195-196
List-based sets, basic concepts（基于链表的集合，基本概念），196-198
List nodes（链表结点）
bounded partial queue（有界部分队列），227
lock-free stack（无锁栈），247
unbounded lock-free queue（无锁的无界队列），230
Liveness property, definition（活性，定义），2
Load-linked store-conditional（LL/SC，链接加载/条件存储）
and ABA problem（ABA问题），237
hardware synchronization（硬件同步），480
origins（起源），136
Loads（加载）
definition（定义），37
Lock algorithm bounds（右界Lock算法），37
in registers（在寄存器中），72
Locality, and cache（局部性，缓存），474
Local spinning, definition（本地自旋，定义），147
Local state, definition（局部状态，定义），37
Lock-based atomic object（基于锁的原子对象）
consistency（一致性），440-441
overview（概述），439-440
structure details（结构细节），441-445
Lock-based concurrent skiplist（基于锁的并发跳表）
algorithm（算法），333-339
overview（概述），331-333
Lock-based hash table, resizing（基于锁的哈希表，调整），305
Lock class（Lock类）
interface（接口），178-179
lower bounds（下界），37-40
for mutual exclusion（互斥），141-144
timeout（超时），157
timestamping（时间戳），34
Lock coupling（锁耦合）
definition（定义），202
invention（发明），219
LockedQueue class, with locks and conditions（LockedQueue类，锁和条件），182
Lock free concurrent skiplist（无锁并发跳表）
algorithm（算法），341-348
overview（概述），339-341
Lock-free exchanger（无锁交换机）
basic function（基本功能），249
implementation（实现），250
Lock-free hash set（无锁哈希集）

basic concept（基本概念），309
BucketList class（BucketList类），312-313
implementation（实现），313-315
recursive initialization（递归初始化），316
recursive split-ordering（递归有序划分），309-312
LockFreeHashSet class, implementation（LockFreeHashSet类，实现），313-315
LockFreeList class, methods（LockFreeList类，方法），217-219
Lock free lists（无锁链表）
abstraction maps（抽象映射），216
AtomicMarkableReference，213-215
basic concepts（基本概念），213-214
methods（方法），217-219
Window class（Window类），216
Lock-free method（无锁方法）
concurrent objects（并发对象），60
concurrent reasoning（并发推理），199
definition（定义），99
LockFreeQueue class（LockFreeQueue类）
creators（生成器），241
list node（链表结点），230
methods（方法），230-231，419-420
LockFreeQueueRecycle class,implementation（LockFreeQueueRecycle类，实现），236
LockFreeSkipList class（LockFreeSkipList类）
call structure（调用结构），340
creators（生成器），348-349
implementation（实现），342-344，346-347，349
overview（概述），339
LockFreeStack class（LockFreeStack类）
creators（生成器），255
elimination（消除），248-249
implementation（实现），246-247
structure（结构），246
Lock-free universal construction（通用的无锁构造）
algorithm（算法），128
execution(执行)，129
generic sequential object（一般顺序对象），126-127
Locking（上锁）
double-checked, in Java memory model（双重检验，Java存储器模型），61-62
execution（执行），47
hand-and-over-hand locking, FineList class（交叉手上锁，FineList类），204
problems（问题），417-418
LockObject class, implementation(LockObject类，实现），23，440-445
LockOne class, for mutual exclusion（LockOne类，互斥），25-26
Lockout-freedom property（无锁特性）
definition（定义），8-9
as dependent progress condition（相关演进条件），60
in Lock algorithm（Lock 算法），24
in producer-consumer problem（生产者-消费者问题），10-11
Locks（锁）
acquires（获取），178
array-based locks（基于数组的锁），150-152
backoff lock（后退锁），167-168
Bakery lock（Bakery 锁），31-33，39，141
block（块），178
C# constructs（C#构造），461
CLH queue lock（CLH队列锁），151，153-154，168-172
composite lock（复合锁），159-160，165-167
definition（定义），22
fair readers-writers lock（公平的读者-写者锁），186-187
Filter lock（Filter 锁），28-31，141
hardware concepts（硬件概念），469
hierarchical backoff lock（层次后退锁），167-168
hierarchical CLH queue lock（层次CLH队列锁），168-172
hierarchical locks（层次锁），167
hold（持有），178
inner read lock（内部读锁），184，186
inner write lock（内部写锁），185，187
interface（接口），23
Java concepts（Java概念），456
in Java memory model（Java存储器模型），62-63

lock-based atomic object（基于锁的原子对象），439
MCS queue lock（MCS队列锁），154-156
monitor locks（管程锁），178-179，181，189
Peterson lock（Peterson锁），27-28，39，142-143
queue locks（队列锁），149-159
readers-writers lock（读者—写者锁），183-187
read lock（读锁），183
reentrant lock（可重入锁），187-189
releases（释放），178
simple readers-writers lock（简单的读者-写者锁），183-185
spin（自旋），178
test-and-set locks（测试—设置锁），144-146
write lock（写锁），184
Lock striping，hash sets（锁分片，哈希集），304
LockTwo class，for mutual exclusion（LockTwo类，互斥），26-27
Logical buckets，lock-free hash set（逻辑桶，无锁哈希集），309
Logical fields，obstruction-free atomic object（逻辑域，无干扰原子对象），435
Logical removal（逻辑删除）
lazy synchronization（惰性同步），196，208
PrioritySkipList class（PrioritySkipList类），363
Loser thread，Common2 register（Loser线程，Common2寄存器），114
Lost-wakeup problem（唤醒丢失问题）
Conditions objects（Conditions对象），182-185
example（实例），183
Java concepts（Java概念），458
Low contention，definition（低争用，定义），147

M

Main cache，hardware transactional memory（主缓存，硬事务内存），448
Main memory，hardware concepts（主存，硬件概念），471，473
Matrix class，implementation（Matrix类，实现），372
Matrix multiplication，parallel，参见Parallel matrix multiplication
MatrixTask class，implementation（MatrixTask类，实现），373-374
MCSLock class，creators（MCSLock类，生成器），174
MCS queue lock（MCS队列锁）
fields and constructor（域和构造函数），154-155
lock acquisition and release（获取和释放锁），156
methods（方法），155
QNode，155-156
Memory，hardware concepts（内存，硬件概念），471，473
Memory barriers，definition（内存路障，定义），53，144，479
Memory consistency models（内存一致性模型）
relaxed（松弛的），478-479
survey（综述），481
Memory contention（内存争用）
definition（定义），474
and shared counting（共享计数），260
Memory fence（内存围栏），参见Memory barriers
Memory locations，lower bounds（内存地址，下界），37-40
Memory reclamation，and ABA problem（内存回收和ABA问题），233-237
Merger class（Merger类）
Correctness（正确性），277
software bitonic counting network（软件双调计数网），275
MERGER network，logical structure（MERGER网，逻辑结构），273
MESI protocol（MESI协议）
Intel processors（Intel处理器），481
state transition examples（状态转换实例），475
Metalock，174
Method calls（方法调用）
concurrent objects（并发对象），56
definition（定义），48
quiescent consistency（静态一致性），49-50
and register space（寄存器空间），76

RMW registers (RMW寄存器), 112-113
Misses, in cache (缺失, 高速缓存), 474
MMThread class, implementation (MMThread类, 实现), 370
Modified state, cache coherence (Modified 状态, 缓存一致性), 446, 475-476
Monitor locks (管程锁)
definitions (定义), 178-179
execution (执行), 180-181
invention (发明), 190
Monitors (管程)
creators (生成器), 466
as C# constructs (C#构造), 461-462
definition (定义), 177, 182
as Java constructs (Java构造), 454-458
MRMW, 参见Multi reader multiple-writer (MRMW)
MRSW, 参见Multi-reader single-writer (MRSW)
multiCompareAndSet, pseudocode (multiCompareAndSet, 伪代码), 419
Multi-core architecture, definition (多核结构, 定义), 477-479
Multicores, programming challenges (多核, 编程挑战), 1
Multiple assignment objects, basic Concept (多重赋值对象, 基本概念), 110
Multiprogrammed environment, work distribution (多道程序环境, 工作分配), 381-382
Multi-reader multiple writer (MRMW, 多读者多写者)
atomic register (原子寄存器), 85-87
construction (构造), 93
Multi-reader single-writer (MRSW, 多读者一单写者)
atomic register (原子寄存器), 82-85
Boolean register (布尔寄存器), 77-78
construction (构造), 93
and register space (寄存器空间), 73
regular Boolean register (规则布尔寄存器), 78-79
regular M-valued register (M-值规则寄存器), 79-81
safe registers (安全寄存器), 78
write order (写顺序), 76
Multi threaded architecture, definition (多线程结构, 定义), 477-479
Mutexes, in Pthreads (Pthreads中的互斥), 464-465
Mutual exclusion (互斥)
Bakery lock algorithm (Bakery锁算法), 31-33
bounded timestamps (有界时间戳), 33-36
and communication (通信), 9-10
in concurrent programming (并发编程), 15
critical sections (临界区), 22-24
definition (定义), 6
fairness (公平性), 31
fast path algorithm (快速路径算法), 43
Filter lock (Filter锁), 28-31
Java concepts (Java概念), 456
LockOne class (LockOne类), 25-26
LockTwo class (LockTwo类), 26-27
number of locations (存储单元数), 37-40
Peterson Lock (Peterson锁), 27-28
in producer-consumer problem (生产者一消费者问题), 10-11
properties (特性), 8-9
real-world approach (现实方法), 141-144
and register space (寄存器空间), 73
time (时间), 21-22
M-valued register (M-值寄存器)
definition (定义), 72
regular MRSW (规则MRSW), 79-81

N

Naive synchronous queue (基本的同步队列)
basic concepts (基本概念), 227
implementation (实现), 238
New version field, obstruction-free atomic object (新版本域, 无干扰原子对象), 435
Node class (Node类)
CombiningTree, 262, 265-267
implementation (实现), 127
LazySkipList class (LazySkipList类), 333, 338
StaticTreeBarrier class (StaticTreeBarrier类), 405

tree barriers（树障碍），401-402
Nodes（结点）
and free lists（空闲链表），233-234
list-based（基于链表的），197-198，227，230，247
NUMA architecture（NUMA系统结构），472
predecessor nodes（前驱结点），171
regular nodes（常规结点），197
sentinel nodes（哨兵结点），197，225，311
Nonblocking methods，definition（无阻塞方法，定义），45
Nonblocking progress，concurrent objects（无阻塞演进，并发对象），59
Nonblocking property，concurrent objects（无阻塞特性，并发对象），58-59
Nonblocking synchronization，definition（无阻塞同步，定义），196
Non-transactional cache，hardware transactional memory（非事务缓存，硬件事务内存），448
Nonuniform memory access（NUMA）architecture（非一致内存访问系统结构）
basic concepts（基本概念），472-473
spinning（自旋），476
North wires（北线）
counting networks（计数网），270
periodic network（周期网），279
Notify，Java concepts（通知，Java概念），457
NUMA，参见Nonuniform memory access（NUMA）architecture

O

Object，definition（对象，定义），48
Obstruction-free atomic object（无干扰原子对象）
basis（基本概念），448
consistency（一致性），436-437
operation details（操作细节），437
overview（概述），435
Obstruction-free property，as dependent progress condition（无干扰特性，相关演进条件），60
Obstruction-free snapshot，collects（无障碍类型，聚集），87-88
OddEven sorting network，design（OddEven排序网，设计），288
Old version field，obstruction-free atomic object（旧的版本域，无干扰原子对象），435
Open-addressed hash set，Cuckoo hashing（开放地址哈希集，Cuckoo哈希），316-318
Open addressing，definition（开放寻址，定义），300
Operator，definition（操纵员，定义），455
Operator thread，definition（操纵员线程，定义），455
OptimisticList class（OptimisticList类）
implementation（实现），205
methods（方法），206-207
validation（验证），208
Optimistic synchronization（乐观同步）
basic concepts（基本概念），205
class implementations（类实现），205-207
definition（定义），195
validation（验证），207
Owner field，obstruction-free atomic Object（Owner域，无干扰原子对象），435

P

Parallelism（并行性）
analysis（分析），375-378
definition（定义），1
and shared counting（共享计数），260
Parallelization，realities（并行，现实），13-14
Parallel matrix multiplication（并行矩阵乘法）
MatrixTask class（MatrixTask类），373-374
overview（概述），369-370
Parallel programming，challenges（并行程序设计，挑战），15
Parallel sorting，basic concepts（并行排序，基本概念），286
Partial method（部分方法）
creators（生成器），241
pool(池)，224
Partial order，concurrent objects（偏序，并发对象），56
Passive thread，in software combining（被动线程，软件组合），260
Pending（未决的）

invocation, concurrent objects (调用，并发对象)，56
method calls (方法调用)，50
Performance (性能)
combiningTree, 269
counting networks (计数网)，280-281
diffracting trees (衍射树)，285
hardware concepts (硬件概念)，471
Periodic counting networks (周期计数网)
implementation (实现)，281
software implementation (软件实现)，279-280
structure (结构)，278-279
Persistent communication, and mutual exclusion (持续通信，互斥)，9
Peterson lock (Peterson锁)
bounds (界限)，39
implementation (实现)，142-143
in mutual exclusion (互斥)，27-28
PhasedCuckooHashSet class, implementation (PhasedCuckooHashSet类，实现)，318-321
Phases (阶段)
computation organization (计算组织)，397
concurrent Cuckoo hashing (并发Cuckoo哈希)，318
Physical removal (物理删除)
lazy synchronization (惰性同步)，196, 208
PrioritySkipList class (PrioritySkipList类)，363
Pipelining, counting networks (流水线，计数网)，280-281
Pools (池)
definition (定义)，223
parallel matrix multiplication (并行矩阵乘法)，370
queues (队列)，224
quiescently consistent (静态一致)，269-270
and shared counting (共享计数)，259
termination detection barrier (终止检测障碍)，408
varieties (变化)，223-224
work stealing (工作窃取)，406-407
Population-oblivious method, concurrent objects (集居数无关方法，并发对象)，59
Postcondition, sequential objects (后置条件，顺序对象)，48
Precedence graph, timestamps (前趋图，时间戳)，34
Precedence relation, in mutual exclusion (前趋关系，互斥)，22
Precombining phase, CombiningTree (预组合阶段，CombiningTree)，262
Precondition, sequential objects (前置条件，顺序对象)，48
Predecessor nodes, HCLHLock queue (前驱结点，HCLHLock队列)，172
Predecessor task, directed acyclic graph (前驱任务，无环有向图)，375
Priority (优先级)
contention manager (争用管理器)，432
in priority queue (优先级队列)，351
Priority inversion, in locking (优先级倒置，上锁)，417
Priority queues, definition (优先级队列，定义)，351
PrioritySkipList class (PrioritySkipList类)
implementation (实现)，364
overview (概述)，363
structure (结构)，365
Prism
diffracting trees (衍射树)，283
distributed coordination (分布式协作)，291
implementation (实现)，284
Probabilistic data structure, SkipList class (概率数据结构，SkipList类)，330
Probe sets, concurrent Cuckoo hashing (测试集，并发Cuckoo哈希)，318
Processors, hardware concepts (处理器，硬件概念)，471, 472
Producer-consumer problem, example (生产者-消费者问题，例子)，10-11
Producer-consumer property, in producer-consumer problem (生产者-消费者特性，生产者-消费者问题)，10-11
Producers (生产者)

naive synchronous queue (基本的同步队列), 237
pools (池), 223
Program correctness (or Correctness), definition (程序正确性, 定义), 2
Program order, definition (程序顺序, 定义), 52
Program performance, definition (程序性能, 定义), 2
Progress conditions (演进条件)
concurrent objects (并发对象), 59-60
dependent, concurrent objects (相关, 并发对象), 60-61
Progress property, concurrent objects (演进特性, 并发对象), 45
Protocol state (协议状态)
bivalence and univalence (单价和二价), 102
consensus numbers (一致数), 101
Pthreads
basic functionality (基本功能), 464
implementation (实现), 467
invention (发明), 466
thread-local storage (本地线程存储器), 465-466

Q

QNode class (QNode类)
CLH queue locks (CLH队列锁), 153
CompositeLock class (CompositeLock类), 161
HCLHLock queue (HCLHLock队列), 169
MCS queue lock (MCS队列锁), 155-156
SynchronousDualQueue class (SynchronousDualQueue类), 239
timeout lock (时限锁), 157-158
Queue locks (队列锁)
array-based (基于数组的), 150-151
CLH, 151-154
MCS, 154-157
overview (概述), 149-150
with timeouts (超时), 157-159
Queues (队列)
array-based bounded priority queues (基于数组的有界优先级队列), 352-353
BoundedDEQueue class (BoundedDEQueue类), 383-385
bounded partial queue (部分有界队列), 225-229
BoundedQueue class (BoundedQueue类), 225-227
bounded-range priority queue (有界范围的优先级队列), 351-353
bounded transactional queue (有界事务队列), 423
CLH queue lock (CLH队列锁), 151, 153-154, 168-172
concurrent priority queues (并发优先级队列), 351-352
FIFO queue (FIFO队列), 45-48
global queue (全局队列), 168
HCLHLock queue (HCLHLock队列), 168-173
hierarchical CLH queue lock (层次CLH队列锁), 168-172
LockedQueue class (LockedQueue类), 182
lock-free queue (无锁队列), 241
LockFreeQueue class (LockFreeQueue类), 230-231, 419-420
LockFreeQueueRecycle class (LockFreeQueueRecycle类), 236
locking queue (锁队列), 47
MCS queue lock (MCS队列锁), 154-156
naive synchronous queue (基本的同步队列), 237-238
pool fairness (公平性的池), 224
priority queues (优先级队列), 351
SimpleTree priority queues (SimpleTree优先级队列), 353-354
skiplist-based unbounded priority queues (基于跳表的无界优先级队列), 363-366
SkipQueue class (SkipQueue类), 365-366
SynchronousDualQueue class (SynchronousDualQueue类), 239-240
synchronous queue (同步队列), 237-238, 241
SynchronousQueue class (SynchronousQueue类), 238
tree-based bounded priority queues (基于树的有界优先级队列), 353-355

unboundedDEQueue class (unboundedDEQueue类), 386-390
unbounded heap-based priority queues (无界基于堆的优先级队列), 357-363
unbounded lock-free queue (无锁的无界队列), 230-233
UnboundedQueue class (UnboundedQueue类), 229
unbounded-range priority queue (无界范围的优先级队列), 351
unbounded total queue (完全无界队列), 229-230
unbounded transactional queue (无界事务队列), 422
Quicksort algorithm, for sample sorting (快速排序算法, 样本排序), 290
Quiescent consistency (静态一致性)
applications (应用), 45
concurrent objects (并发对象), 49-51
concurrent priority queues (并发优先级队列), 351-352
pools and counters (池和计数器), 269-270
vs. sequential consistency (顺序一致性), 52-53
shared counters (共享计数器), 270

R

Radix children, tree nodes (基数个儿子, 树结点), 401
Reachable, concurrent reasoning (可达的, 并发推理), 199
Readers-writers lock (读者-写者锁)
basic concepts (基本概念), 183
fair lock (公平锁), 185-187
simple lock (简单锁), 184-185
Readers-writers problem, example (读者-写者问题, 实例), 11-12
Read lock, definition (读锁, 定义), 183
Read-modify-write (RMW) registers (读-改-写寄存器)
Common2 registers (Common2 寄存器), 114-116
methods (方法), 112-113
shared counter implementation (共享计数器实现), 5-6
source (源), 117
Read set, lock-based atomic object (读集合, 基于锁的原子对象), 439
Realistic multiprocessor scheduling, definitions and operation (现实中的多处理器调度, 定义和操作), 378-380
Real-time order (实时次序), vs. sequential consistency (顺序一致性), 53
Rebalancing, definition (重新平衡, 定义), 329
Recursive initialization, lock-free hash set (递归初始化, 无锁哈希集), 316
Recursive split-ordering, lock-free hash set (有序划分递归, 无锁哈希集), 309-312
Reentrant (可重入)
BaseHashSet class (BaseHashSet类), 301
Conditions objects (Conditions对象), 180
Reentrant lock (可重入锁)
definition (定义), 187
methods (方法), 188
Reference, BoundedDEQueue class (引用BoundedDEQueue类), 384
Refinable concurrent Cuckoo hash set (细化并发Cuckoo哈希集), 324-325
RefinableCuckooHashSet class (RefinableCuckooHashSet类), 324-325
RefinableHashSet class implementation (RefinableHashSet类, 实现), 306-308
resizing (可调整), 305
RegBoolMRSWRegister class, implementation (RegBoolMRSWRegister类, 实现), 79
Registers (寄存器)
construction overview (构造概述), 77-78
definition (定义), 50, 71, 72
and mutual exclusion (互斥), 73
safe (安全), 74-75
3D implementation space (3D实现空间), 76
write order (写顺序), 76
RegMRSWRegister class, implementation (RegMRSWRegister类, 实现), 80
Regular nodes, list-based sets (规则结点, 基于链表的集合), 197
Regular registers (规则寄存器)

Boolean MRSW（布尔MRSW），78-79
conditions（条件），77
definition（定义），75
first definition（首次定义），93
implementation considerations（实现考虑），75-76
M-valued MRSW register（*M*-值MRSW寄存器），79-81
safe（安全），75
RelaxedLock，174
Releases（释放）
CLHLock class（CLHLock类），154
definition（定义），23
HCLHLock lock（HCLHLock锁），173
Java concepts（Java概念），456
locks（锁），178
MCSLock class（MCSLock类），156
Reordering, and memory consistency（重排序，内存一致性），479
Replacement policy, and cache（替换策略，缓存），474
Representation invariant, concurrent reasoning（不变量表示，并发推理），198-199
Reservation object, dual data structures（保留对象，双重数据结构），239
Response events（响应事件）
concurrent objects（并发对象），56
lock-free universal algorithm（通用无锁算法），127
and register space（寄存器空间），76
RESULT status（RESULT状态）
CombiningTree，267
definition（定义），262
Reverse tree barrier, implementation（翻转树障碍，实现），412-414
RMW，参见Read-modify-write (RMW) registers
Robustness, CombiningTree（健壮性，CombiningTree），269
ROOT status（ROOT状态）
CombiningTree，265-266
definition（定义），262
Runnable object, Java thread concepts（Runnable对象，Java线程概念），453-454

S

Safe（安全的）
registers（寄存器），74-75
regular register（规则寄存器），75
SafeBoolMRSWRegister class, implementation（SafeBoolMRSWRegister类，实现），78
Safe registers（安全寄存器）
first definition（首次定义），93
MRSW，78
SRSW Boolean（SRSW布尔），86
Safety property, definition（安全特性，定义），2
Sample sorting（样本排序）
original ideas（最初起法），293
phases（阶段），290-291
Saturation, counting networks（饱和，计数网），280
Scan-and-label operations, timestamps（扫描-标记操作，时间戳），34
Scheduler（调度程序）
function（函数），378
greedy（贪心），379
idle step（空闲步），379
SECOND status（SECOND状态）
CombiningTree，268
definition（定义），261
Semaphores, definition and implementation（信号量，定义和实现），189
SenseBarrier class, constructor（感知器障类，构造函数），400
Sense-Reversing barrier, implementation（语义换向障碍，实现），399-400
Sentinel nodes（哨兵结点）
bounded partial queues（部分有界队列），225
list-based sets（基于链表的集合），197
split-ordered hash set（有序划分哈希集），311
Sequential bottleneck（顺序瓶颈）
LockFreeStack，248
and shared counting（共享计数），259
Sequential consistency（顺序一致性）
Applications（应用），45
concurrent objects（并发对象），51-54
vs. quiescent consistency（静态一致性），52-53

vs. real-time order（实时次序），53
SequentialHeap class, implementation（SequentialHeap类，实现），356-357
Sequential objects（顺序对象）
  generic definition（一般定义），127
  specifications（说明），48-49
SequentialRegister class, implementation（SequentialRegister类，实现），73
Sequential skiplists, definition and design（顺序跳表，定义和设计），329-331
Sequential specification（顺序说明）
  concurrent objects（并发对象），56
  definition（定义），49
Sequential timestamping, Lock class（顺序时间戳，Lock类），34
Serializable, transactional memory（可串行化，事务内存），421
Sets（集）
  definition（定义），196
  list-based（基于链表的），196-198
Shared concurrent objects, and register space（共享并发对象，寄存器空间），72
Shared counter（共享计数器）
  approach（方法），259-260
  implementation（实现），4-5
  quiescently consistent（静态一致），270
Shared-memory multiprocessors, programming challenges（共享存储器的多处理器，程序设计的挑战），1
Shared objects, and synchronization（共享对象，同步），3-6
Shared state, cache coherence（共享状态，缓存一致性），446, 475-476
Sharing, definition（共享，定义），474
SimpleBarrier class, implementation（SimpleBarrier类，实现），399
SimpleLinear class（SimpleLinear类）
  creators（生成器），366
  implementation（实现），352
Simple readers-writers lock（简单的读者-写者锁）
  basic concepts（基本概念），184-185
  fields and public methods（域和公共方法），184
  inner read lock（内部读锁），184
  inner write lock（内部写锁），185
SimpleTree priority queues（SimpleTree优先级队列）
  implementation（实现），354
  structure（结构），353
Single-reader, single-writer（SRSW, 单读者-单写者）
  atomic register（原子寄存器），77-78, 81-83
  register space（寄存器空间），73
  safe（安全），74
  write order（写次序），76
Single-writer atomic snapshot class（单写者原子快照类）
  collect method（采集方法），91
  update and scan methods（更新和扫描方法），91
SkipList class（SkipList类）
  algorithm（算法），349
  levels（级别），330
Skiplists（跳表）
  invention（发明），348
  LazySkipList class（LazySkipList类），331-332
  software transactional memory（软件事务内存），424-425
  unbounded priority queues（无界优先级队列），363-366
SkipListSet class, implementation（SkipListSet类，实现），425
SkipNode class, interface（SkipNode类，接口），425
SkipQueue class（SkipQueue类）
  characteristics（特性），365-366
  creators（生成器），366
Sleeping, as Java construct（睡眠，Java构造），458
Slot, array-based locks（槽，基于数组的锁），150
SMP, 参见Symmetric multiprocessing（SMP）architecture
Snapshots（快照）
  construction（构造），93
  correctness arguments（正确性证明），90-93

definition（定义），87
and multiple assignment（多重赋值），110
obstruction-free（无障碍），87-88
wait-free snapshot（无等待快照），88-90
Snooping（侦探）
cache coherence（缓存一致），446
interconnects（互相连接），472
Soft real-time application, definition（软实时应用，定义），397
Software combining, basic concepts（软件组合，基本概念），260-261
Software implementation（软件实现）
bitonic network classes（双调网络类），274-276
bitonic network proof of correctness（双调网络正确性证明），276-278
periodic counting network（周期计数网），279-280
Software transactional memory (STM) atomic object implementation（软事务内存原子对象实现），433-434
atomic objects（原子对象），429-431
contention manager（争用管理器），431-433
dependent or independent（相关或无关），431
first proposal（首先建议），448
lock-based atomic objects（基于锁的原子对象），438-445
obstruction-free atomic object（无干扰原子对象），434-438
overview（概述），424-427
skip lists（跳表），424-425
transactions and transactional threads（事务和事务线程），427-428
TThread class（TThread 类），426
zombies and consistency（僵尸和一致性），428-429
Sorting（排序）
bitonic algorithm（双调算法），288-289
parallel（并行），286
Sorting networks（排序网络）
designing（设计），287-288
OddEven design（OddEven设计），288
structure（结构），286-287
South wires（南线）
counting networks（计数网），270
periodic network（周期网），279
Speculativeness, memory transactions（试探性地，内存事务），422
Spin-locks（自旋锁）
definition（定义），141，178
TAS-based（基于TAS），146-147
Spinning（自旋）
definition（定义），141
hardware concepts（硬件概念），476
Splitters, sample sorting（分裂器，样本排序），290
SRSW，参见Single-reader, single-writer (SRSW)
SSkipNode class, implementation（SSkipNode类，实现），430
Stack class, definition（Stack类，定义），245
Stamp（标记）
ABA problem（ABA问题），234
BoundedDEQueue class（BoundedDEQueue类），384
lock-based atomic object（基于锁的原子对象），439
Stamped snapshot class, implementation（具有时间戳的快照类，实现），90
Start（启动）
C# constructs（C#构造），460
Java constructs（Java构造），454
Starvation, in Lock object（饥饿，Lock对象），24
Starvation-freedom property（无饥饿特性）
definition（定义），8-9
as dependent progress condition（相关演进条件），60
in Lock algorithm（Lock算法），24
in producer-consumer problem（生产者—消费者问题），10-11
State（状态）
consensus numbers（一致数），101-103
objects（对象），48
State machine, definition（状态机，定义），21
Static tree barrier, implementation（静态树障碍，实现），402-404
StaticTreeBarrier class, Node class（StaticTreeBarrier类，结点类），405

Step property（步进特性）
  balancing networks（平衡网络），272
  bitonic counting network（双调计数网），276
  scheduler（调度程序），379
  Tree class（Tree类），282
Steps, directed acyclic graph（步骤，无环有向图），380
STM，参见Software transactional memory (STM)
Store buffer（存储缓冲区），参见Write buffer
  definition，478
Stores（存储）
  definition（定义），37
  Lock algorithm bounds（Lock算法限制），37
  in registers（寄存器），72
Striped concurrent Cuckoo hashing（空间分带的并发Cuckoo哈希），322-324
StripedCuckooHashSet class（StripedCuckooHashSet类），322-323
Striped hash set（空间分带的哈希集）
  basic concepts（基本概念），303-304
  implementation（实现），304
  lock striping（空间分带的上锁），304
StripedHashSet class（StripedHashSet类）
  implementation（实现），304
  lock-based hash table（基于锁的哈希表），305
  resizing（调整），306
Subhistory, concurrent objects（子经历，并发对象），56
Successful call, definition（成功调用，定义），196
Successor state, consensus numbers（后继状态，一致数），101
Symmetric multiprocessing（SMP，对称多处理）
  spinning（自旋），476
Symmetric multiprocessing (SMP) architecture（对称多处理器系统结构）
  basic concepts（基本概念），472-473
Synchronization（同步）
  coarse-grained（粗粒度），195
  coarse grained hash sets（粗粒度哈希集），302
  fine-grained（细粒度），195
  hardware instructions（硬件指令），480-481
  instructions, memory barriers（指令，内存障碍），144
  in Java memory model（Java存储器模型），62
  lazy（惰性），196
  nonblocking（无阻塞），196
  optimistic（乐观），195
  and shared objects（共享对象），3-6
Synchronization primitives（同步原语）
  atomic registers（原子寄存器），103-105
  Common2 RMW registers（Common2 RMW寄存器），114-116
  compareAndSet() operation（compareAndSet()操作），116-117
  consensus numbers（一致数），100-103
  consensus protocols（一致性协议），106
  definition（定义），99-100
  FIFO queues（先进先出队列），106-109
  multiple assignment objects（多重赋值对象），110-112
  read-modify-write operations（读-改-写操作），112-114
Synchronized blocks, in Java memory model（同步块，Java存储器模型），62-63
SynchronousDualQueue class（SynchronousDualQueue类）
  method and constructor（方法和构造函数），240
  queue node（队列结点），239
Synchronous method, pool（同步方法，池），224
Synchronous queue（同步队列）
  basic concepts（基本概念），237
  creators（生成器），241
  implementation（实现），238
SynchronousQueue class, implementation（SynchronousQueue类，实现），238

T

Table（表）
  Cuckoo hashing（Cuckoo哈希），316
  definition（定义），299-300
Tail, definition（队尾，定义），224
Tasks, work distribution（任务，工作分配），381-382
TASLock class（TASLock类）

implementation (实现), 144-145
performance (性能), 146
Tentativeness, memory transactions (暂时态, 内存事务), 422
Termination detection barrier (终止检测障碍)
creators (生成器), 408
implementation (实现), 404-408
Test-and-set locks (测试-设置锁)
basic principles (基本原则), 144
hardware concepts (硬件概念), 469
real-world processors (现实处理), 145-146
Test-and-test-and-set (测试 测试 设置)
definition (定义), 144-145
hardware concepts (硬件概念), 469
Thief, work stealing (偷窃者, 工作窃取), 381-382
Thinlock, 174
Thread-local objects (局部级线程对象)
as C# constructs (C#构造), 462-463
as Java constructs (Java构造), 458-459
Thread-local storage, Pthreads (本地线程存储器, Pthreads), 465-466
Thread-local variables, array-based locks (局部线程变量, 基于数组的锁), 150
Threads (线程)
as C# constructs (C#构造), 460-461
definition (定义), 71
hardware concepts (硬件概念), 472
as Java constructs (Java构造), 453-454
Throughput, definition (吞吐量, 定义), 259
Time, in mutual exclusion (时间, 互斥), 21-22
Timeouts, in queue lock (超时, 队列锁), 157-159
Timestamps (时间戳)
Bakery lock labels (Bakery锁标记), 34
definition (定义), 72
overview (概述), 33
TOLock class (TOLock类)
fields and constructor (域和构造函数), 158
methods (方法), 158
and timed-out nodes (超时结点), 159
Top wires (顶线)
counting networks (计数网), 270
sorting networks (排序网), 286
Total method, pool (全局方法, 池), 223
Total order, concurrent objects (全序, 并发对象), 56
TourBarrier class (TourBarrier类)
implementation (实现), 410
information flow (信息流), 411
Tournament tree barrier, creators (竞赛树障碍, 生成器), 408
Transactional cache, hardware transactional memory (事务缓存, 硬事务内存), 448
Transactional cache coherence, hardware transactional memory (事务缓存一致性, 硬事务内存), 447
Transactional Locking 2 algorithm, creators (事务上锁2算法, 生成器), 448
Transactional memory (事务内存)
atomic primitive problems (原子原语问题), 418-420
compositionality problems (复合问题), 420-421
definition (定义), 3
locking problems (上锁问题), 417-418
overview (概述), 417
transactions and atomicity (事务和原子性), 421-423
Transactional threads, and transactions (事务线程, 事务), 427-428
Transactions (事务)
and atomicity (原子性), 421-423
definition (定义), 421
and transactional threads (事务线程), 427-428
Transient communication, and mutual exclusion (瞬时通信, 互斥), 9
TreeBarrier class (TreeBarrier类)
combining tree (组合树), 403
implementation (实现), 401-402
Tree-based bounded priority queues, structure and implementation (基于树的有界优先级队列, 结构和实现), 353-355
Tree class, structure (Tree类, 结构), 282
TSkipNode class, implementation (TSkipNode类, 实现), 434

TTASLock class (TTASLock类)
creators (生成器), 172
exponential backoff (指数后退), 147-149
implementation (实现), 145, 470
on shared-bus (共享总线), 146-147
performance (性能), 146
TThread class, software transactional memory (TThread类, 软事务内存), 426

U

unboundedDEQueue class, implementation (unboundedDEQueue类, 实现), 386-390
Unbounded heap-based priority queues (基于堆的无界优先级队列)
concurrent heap (并发堆), 357-363
definitions (定义), 355
sequential heap (顺序堆), 356-357
Unbounded lock-free queue (无锁的无界队列)
implementation (实现), 230-231
lazy (惰性), 231-232
step-wise operation (逐步求精操作), 232-233
Unbounded lock-free stack (无锁的无界栈)
definition (定义), 245
implementation (实现), 246-247
structure (结构), 246
Unbounded pool, characteristics (无界池, 特性), 223
UnboundedQueue class, methods (UnboundedQueue类, 方法), 229
Unbounded range priority queue, definition (无界范围的优先级队列, 定义), 351
Unbounded total queue (完全无界队列)
deadlock (死锁), 230
implementation (实现), 229-230
Unbounded transactional queue, implementation (无界事务队列, 实现), 422
Unbounded work-stealing double-ended queues (无界工作窃取双端队列)
creator (生成器), 392
implementation (实现), 386-388
Univalence, protocol state (单价, 协议状态), 102
Universality, definition (通用性, 定义), 126
Unlocks (解锁)
CLHLock class (CLHLock类), 154
definition (定义), 22, 23
hardware concepts (硬件概念), 469
HCLHLock lock (HCLHLock锁), 173
interface (接口), 23
Java concepts (Java概念), 456-457
locks (锁), 178
MCSLock class (MCSLock类), 156
Unsuccessful call, definition (不成功调用, 定义), 196
Update value, compareAndSet() operation (更新值, compareAndSet()操作), 116, 480
Up state, balancers (上状态, 平衡器), 271

V

Valence, consensus numbers (价, 一致数), 101-103
Validation (验证)
LazyList class (LazyList类), 209, 211
optimistic synchronization (乐观同步), 207
transaction handlers (事务处理程序), 428
Version, lock-based atomic object (版本, 基于锁的原子对象), 439
Version clock, lock based atomic object (版本锁, 基于锁的原子对象), 439, 441
Victim (牺牲者)
unbounded work-stealing DEQueue (无界工作窃取双重队列), 391
work stealing (工作窃取), 381
Victim cache, hardware transactional memory (牺牲者缓存, 硬事务内存), 448
Volatile fields, in Java memory model (Volatile域, Java存储器模型), 63

W

Wait-free method (无等待方法)
concurrent objects (并发对象), 59
concurrent reasoning (并发推理), 199
definition (定义), 99
history (经历), 65
LazySkipList class (LazySkipList类), 335
LockFreeList class (LockFreeList类), 219

MRMW atomic register (MRMW原子寄存器), 86
and register space (寄存器空间), 73
Wait-free snapshot, construction (无等待快照, 构造), 88-90
Wait-free universal construction (通用的无等待构造)
algorithm (算法), 131
auxiliary variables(辅助变量), 133-134
execution (执行), 132
helping pattern (帮助模式), 130-131
linearizability (可线性化性), 133
modification considerations (修改考虑), 133
node announcement (结点通知), 130-132
proof (证明), 135-136
Waiting (等待)
Java concepts (Java概念), 456
in Lock class (Lock类), 31
in mutual exclusion (互斥), 15
in readers-writers problem (读者—写者问题), 12
Well-formed thread, requirements (良构线程, 需求), 23
Window class, lock-free lists (Window类, 无锁链表), 216
Winner thread, Common2 register (胜利者线程, Common2寄存器), 114
Work, parallelism (工作, 并行), 376
Work distribution (工作分配)
balancing (平衡), 389-391
bounded work stealing dequeue (有界工作窃取队列), 382-386
overview (概述), 380
unbounded work-stealing DEQueue (无界工作窃取双端队列), 386-388
work stealing (工作窃取), 380
yielding and multiprogramming (退从和多道程序设计), 380-382
Work sharing, creators (工作共享, 生成器), 391
WorkSharingThread class, implementation (WorkSharingThread类, 实现), 391
Work stealing (工作窃取)
characteristics (特性), 381
creators (生成器), 391
executor pool (执行者池), 406-407
Work-stealing double-ended queues (工作窃取双端队列)
bounded (有界的), 382-386
overview (概述), 382
unbounded (无界的), 386-388
WorkStealingThread class, implementation (WorkStealingThread类, 实现), 382
Write absorption, definition (写吸收, 定义), 478
Write buffer (写缓存)
definition (定义), 478
shared memory writes (对共享存储器的写), 143
Write lock, definition (写锁, 定义), 183
Write order (写次序)
and register space (寄存器空间), 76
SRSW and MRSW (单读者—单写者和多读者—单写者), 76
Write set, lock-based atomic object (写集合, 基于锁的原子对象), 439

Y

Yielding (退从)
as Java construct concepts (Java构造概念), 458
work distribution (工作分配), 380-382

Z

Zombies, software transactional memory (僵尸, 软件事务内存), 428-429