



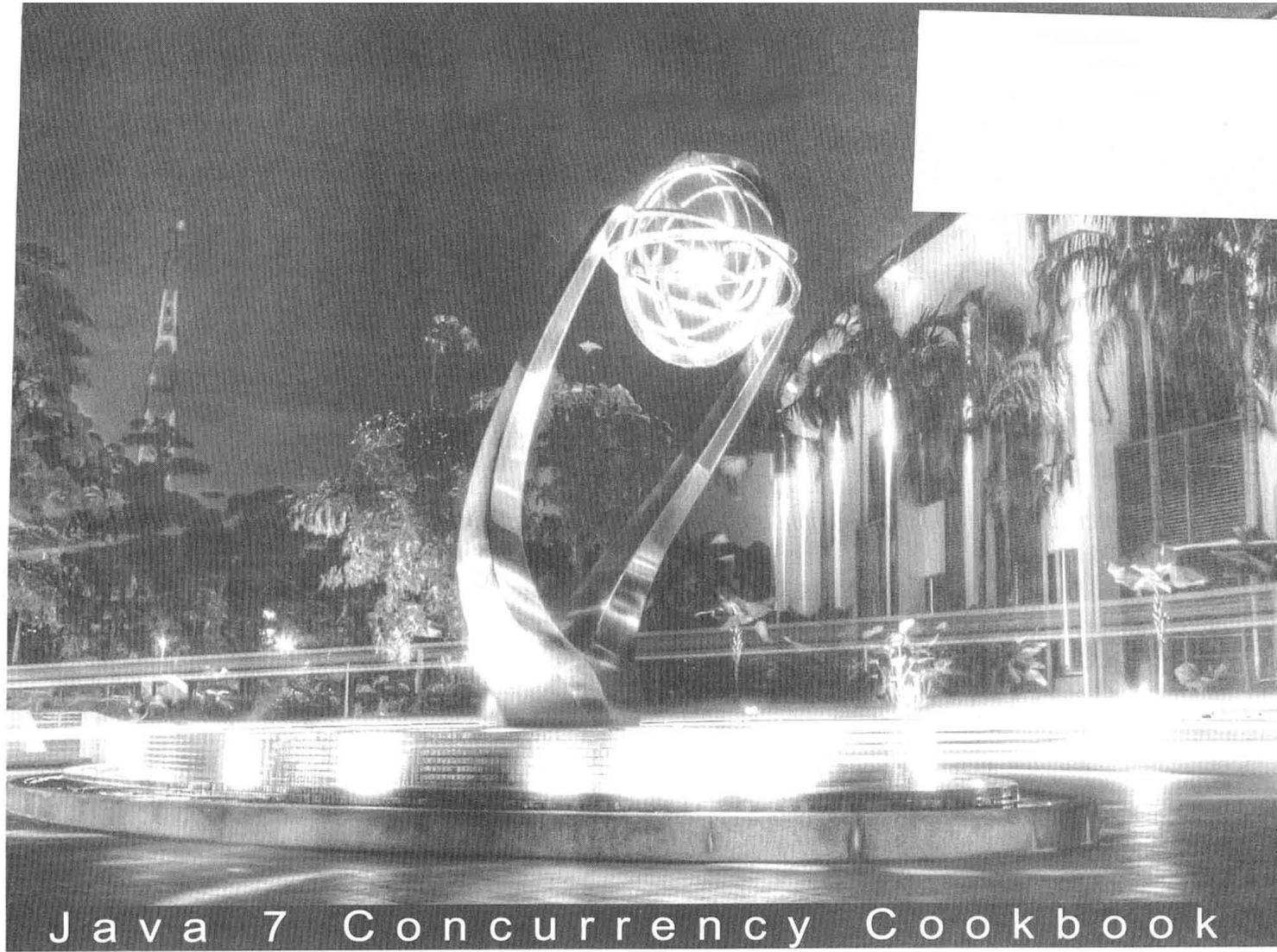
Java 7 Concurrency Cookbook

Java 7 并发编程 实战手册

超过60个简单而富有成效的技巧
彻底掌握Java 7多线程应用程序的开发

[西] Javier Fernández González 著
申绍勇 俞黎敏 译

 人民邮电出版社
POSTS & TELECOM PRESS



Java 7 Concurrency Cookbook

Java 7 并发编程 实战手册

[西] Javier Fernández González 著
申绍勇 俞黎敏 译

人民邮电出版社
北京

图书在版编目 (CIP) 数据

Java 7 并发编程实战手册 / (西) 冈萨雷斯著 ; 申绍勇, 俞黎敏译. — 北京 : 人民邮电出版社, 2014.2
ISBN 978-7-115-33529-6

I. ①J… II. ①冈… ②申… ③俞… III. ①JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2013)第282191号

版权声明

Copyright ©2012 Packt Publishing. First published in the English language under the title Java 7 Concurrency Cookbook All Rights Reserved.

本书由英国 **Packt Publishing** 公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

内 容 提 要

Java 7 在并发编程方面，带来了很多令人激动的新功能，这将使你的应用程序具备更好的并行任务性能。

本书是 Java 7 并发编程的实战指南，介绍了 Java 7 并发 API 中大部分重要而有用的机制。全书分为 9 章，涵盖了线程管理、线程同步、线程执行器、Fork / Join 框架、并发集合、定制并发类、测试并发应用等内容。全书通过 60 多个简单而非常有效的实例，帮助读者快速掌握 Java 7 多线程应用程序的开发技术。学习完本书，你可以将这些开发技术直接应用到自己的应用程序中。

本书适合具有一定 Java 编程基础的读者阅读和学习。如果你是一名 Java 开发人员，并且想进一步掌握并发编程和多线程技术，并挖掘 Java 7 并发的新特性，那么本书是你的合适之选。

◆ 著 [西] Javier Fernández González
译 申绍勇 俞黎敏
责任编辑 陈冀康
责任印制 程彦红 杨林杰
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000 1/16
印张：22.25
字数：441 千字 2014 年 2 月第 1 版
印数：1~3 500 册 2014 年 2 月北京第 1 次印刷
著作权合同登记号 图字：01-2013-3670 号

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

作者简介

Javier Fernández González 是一名有着超过 10 年 Java 技术经验的软件架构师。他曾担任过教师、研究员、程序员和分析员，现在是 Java 项目（特别是 J2EE 相关项目）的架构师。在担任教师期间，他在 Java、J2EE 和 Struts 框架上有超过 1000 小时的教学时间。当研究员时，他曾在信息检索领域，用 Java 开发应用程序来处理大量的数据，此外他还是一些期刊文章和会议演示的合作者。近些年来，Javier 在不同的领域（比如公共行政、保险、医疗保健、交通等）为不同的客户开发 J2EE Web 应用程序。目前，他在欧洲最大的咨询公司（Capgemini，凯捷）担任软件架构师，为保险公司开发和维护应用程序。

审阅者简介

Edward E. Griebel Jr 在读小学的时候，通过 Apple 电脑上的 LOGO¹ 语言和在 VAX 上的 Oregon Trail 首次接触计算机。

VAX（Virtual Address eXtension）是一种可以支持机器语言和虚拟地址的 32 位小型计算机。VAX 最初由迪吉多电脑公司（DEC）在 20 世纪 70 年代初发明。

之后，Edward 一直保持着对计算机的兴趣与追求，他毕业于 Bucknell 大学，获得了计算机工程学士学位。在 Edward 的第一份工作中，他很快意识到自己除了计算机编程技术以外其他的都不懂。在过去的 20 年间，Edward 已经在证券交易、电信、支付处理、物联网等行业中磨砺过，担任过开发人员、团队的领导者、技术顾问和导师。目前在从事 Java EE 企业应用开发，他认为编写代码的日子是很快乐的，而不是苦闷的。

他说：

我要感谢我的妻子和三个孩子，他们允许我即使深夜也仍能在电脑前工作，直到很晚才休息而无法陪伴他们。

¹ 译注：LOGO 是一款计算机程序设计语言，在 1966 年由西摩尔·派普特 和 Wally Feurzeig 在 BNN 设计，设计 LOGO 的初衷是为了向儿童教授计算机编程技术。

Jacek Laskowski 是一位非常专业的软件专家，擅长采用大量的商业和开源的解决方案来满足客户不同的业务需求。他开发应用程序、编写文章、辅导经验不足的工程师、录制视频、发布课程，同时，Jacek 也是许多 IT 书籍的技术审校者。

Jacek 专注于 Java EE、SOA (Service-Oriented Architecture)、BPM (Business Process Management) 解决方案、OSGi 技术以及函数语言 (Functional Programming)，比如 Clojure 和 F# 语言。他也涉猎 Scala、Dart，并用 Java 和 HTML 5 进行原生的 Android 开发。

Jacek 也是波兰首都、最大城市华沙的 Java 用户组 (Warszawa Java User Group, Warszawa JUG) 的创始人与领导者。他还是 Apache 软件基金会 (Apache Software Foundation) 成员，Apache OpenEJB 和 Apache Geronimo 项目管理委员会 (Project Management Committee, PMC) 会员和提交者 (Committer)。Jacek 经常在开发者大会上演讲。他的博客地址是 <http://blog.japila.pl> 和 <http://blog.jaceklaskowski.pl>，他的 Twitter 是 @jaceklaskowski。

Jacek 已经在 IBM 工作 6 年多了，现在是 World-wide WebSphere 竞争对手迁移团队成员，并且也是通过 Level 2 认证的 IT 专家(译注：Level 2 即 IBM 公司内部的 Band 9 大师)。他协助客户从竞争对手的产品 (通常是 Oracle WebLogic Server) 迁移到 IBM WebSphere Application Server 上。

最近，Jacek 加入了 IBM 技术研究院 (IBM Academy of Technology)。

他说：

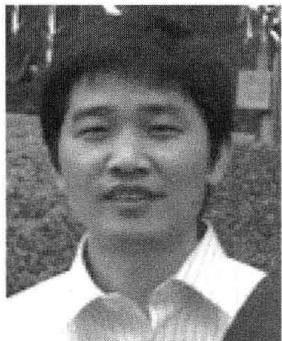
我要感谢我的家人——我的妻子 Agata 以及 3 个小孩 Iweta、Patryk 和 Maksym，有了他们坚定的支持、鼓励和耐心，我才取得这么多的成就！我爱你们！

Abraham Tehrani 也是一位具有超过十年软件开始经验的开发人员和 QA 工程师。同时，他对质量和技术充满了热情！

他说：

我要感谢我的未婚妻，她给予我支持和爱的力量。我也要感谢我的朋友和家人，谢谢他们的全力支持。

译者简介



申绍勇

2003 年研究生毕业于中山大学，2007 年加入 IBM 软件部，现任职 WebSphere 中间件资深售前工程师。擅长跨平台移动应用开发、移动应用整体解决方案设计，SOA 及企业应用集成，对 IBM 的 WebSphere MQ、Message Broker、Worklight 等产品比较熟悉。在电信、银行、政府、证券、保险、物流等各个行业都有丰富的项目经验。



俞黎敏（ID:YuLimin，网名：阿敏总司令）

2008 年 7 月 1 日加入国际商业机器（中国）有限公司广州分公司（IBM 广州），担任软件部高级信息工程师、资深技术顾问，主要负责 IBM WebSphere 系列产品的技术支持工作，专注于产品新特性、系统性能调优、疑难问题诊断与解决。

开源爱好者，曾经参与 Spring 中文论坛组织“Spring 2.0 Reference”中文翻译的一审与二审工作，“满江红开放技术研究组织”的“Seam 1.2.1 Reference”中文翻译工作，并组织和完成“Seam 2.0 Reference”中文翻译工作。利用业余时间担任 CSDN、CJSDN、Dev2Dev、Matrix、JavaWorldTW、Spring 中文、WebSphereChina.net 等 Java 论坛版主，在各大技术社区推动开源和敏捷开发做出了积极的贡献。参与审校与翻译的书籍有《Ajax 设计模式》、《CSS 实战手册》、《Hibernate 实战》（第 2 版）、《Java 脚本编程》、《Effective Java 中文版第 2 版》、《Spring 攻略》、《CSS 实战手册第 2 版》、《Seam 实战》、《REST 实战中文版》、《Java 7 程序设计》、《Servlet 和 JSP 学习指南》等。

博客：<http://blog.csdn.net/YuLimin> 或者 <http://YuLimin.ItEye.com>

微博：<http://weibo.com/iAMin83567>

译者序

Java 是一种计算机编程语言，拥有跨平台、面向对象、泛型编程的特性，广泛应用于企业级 Web 应用开发和移动应用开发。由 Sun 公司的 James Gosling 等人于 1990 年代初开发 Java 语言的雏形，最初被命名为 Oak，于 1995 年 5 月以 Java 的名称正式发布。伴随着互联网的迅猛发展而发展，Java 逐渐成为重要的网络编程语言。

自从 Java 5.0 增加了最初由 Doug Lea 编写的高质量的、广泛使用的、并发实用程序 `util.concurrent` 并变成了 JSR-166 的新包之后，在 Java 内置所提供的类库中，就提供了越来越多的并发编程的实用工具类。学习并掌握这些技术对于专注于 Java 并发编程的开发人员来讲是基本的功力，随着 Java 版本的不断更新与改进，开发人员可以通过 Java 新版本所带来的新特性，无需从头重新编写并发程序工具类。

本书作者 Jacek Laskowski 是一位非常专业的软件专家，擅长采用大量的商业和开源的解决方案来满足客户不同的业务需求。他开发应用程序，编写文章，辅导经验不足的工程师，录制视频，发布课程，同时，他也是许多 IT 书籍的技术审校者。

适合人群

本书是针对有 Java 编程语言基础的开发者的，需要已经熟悉普通的 Java 开发实践，如果掌握了的线程基本知识，那么阅读本书将更加得心应手。如果想进一步掌握并发编程和多线程技术，以及挖掘 Java 7 并发的新特性，那么，本书正适合你，一边阅读一边动手实验掌握之。

章节简介

Java 是一个并发平台，它提供了大量的类来执行 Java 程序中的并发任务。随着版本的不断更新发展，Java 不断地为程序员增加了并发编程的开发功能。本书覆盖了 Java 7 并发 API 中大部分重要而有用的机制，因此，你将能够直接在应用程序中使用它们，包括如下主题：

- 线程管理，通过基础的范例来讲解线程的创建、线程的执行以及线程的状态管理；
- 线程同步基础，为读者讲解如何使用低级的 Java 机制，比如采用 Lock 锁接口和 `synchronized` 关键字来同步代码；
- 线程同步辅助类，讲解如何使用 Java 的高级工具类来管理 Java 中的线程同步。比如介绍 Java 7 当中的 Phaser 类，用来同步被拆分成多个阶段的任务；

- 线程执行器（Thread Executor），讲解如何将线程管理委托给执行器（Executor）。执行器将为并发任务负责线程的创建、运行、管理并返回任务的结果；
- Fork / Join 框架（Fork/Join Framework），讲解如何使用 Java 7 新的一种特殊的执行器 Fork / Join 框架。用来解决通过分治技术（Divide and Conquer Technique）将任务拆分成多个子任务的问题；
- 并发集合，讲解如何使用一些由 Java 语言提供的并发数据结构，从而避免在程序的实现中采用 `synchronized` 代码块；
- 定制并发类（Customizing Concurrency Classes），讲解如何根据需要来改编 Java 并发 API 中一些非常有用的类；
- 测试并发应用（Testing Concurrent Application），讲解如何获取 Java 7 并发 API 中最有用的结构的状态信息。讲解如何使用一些免费的工具来调试并发应用程序，比如：Eclipse、NetBeans IDE。或者用来检测应用程序中是否存在 Bug 的 FindBugs 开源框架。
- 附加信息没有包含在本书中，但是它可以通过如下链接免费下载：
<http://www.Java2Class.net/Java7ConcurrentCookbook/>，讲解同步的概念，执行器框架（Executor Framework）和 Fork/Join 框架（Fork/Join Framework），并发数据结构，以及没有包含在相应章节里的并发对象的监控。
- 附录，并发编程设计（Concurrent Programming Design）也没有包含在本书当中，但是它可以通过如下链接免费下载：
<http://www.Java2Class.net/Java7ConcurrentCookbook/>，讲解每一位程序员在开发并发应用程序时应当考虑的一些技巧。

技术范围

本书旨在使你更全面、更专业地掌握 Java 7 并发 API 编程开发技术，但是它只是一本比较基础的书籍，如果你已经有多年的 Java 并发应用程序开发经验，或许可以通过快速的阅读或者直接找到自己所需要的新技术点。当阅读本书时，你会遇到许多需要动手进行验证的实例，可以利用本书附带的示例程序进行练习与实践。示例与答案下载地址为：
http://www.packtpub.com/code_download/10250。

正如在翻译过程中发现原著的错误一样，虽然我们在翻译过程中竭力求信、达、雅，但限于自身水平，必定仍会有诸多不足，还望各位读者不吝指正。大家可以通过访问我的博客 <http://YuLimin.ItEye.com> 或者发送电子邮件到 YuLimin@163.com 进行互动。

关于术语的翻译，仍然沿用翻译 Effective Java 中文第 2 版时采用的术语表以及满江红开放技术研究组织翻译术语，请见 <http://yulimin.iteye.com/blog/272088>。

感谢崔毅（<http://cuiyi.javaeye.com/>）对我在翻译中碰到的问题进行的深入讨论，并对本书翻译时所采用的术语进行了认真的磋商；感谢“满江红开放技术研究组织”的翻译同仁们在术语表讨论中提出许多中肯的建议；感谢满江红开源组织的曹晓钢提供的一些翻译注意事项和热情的帮助；感谢人民邮电出版社的编辑陈冀康认真仔细以及反复的校对与检查，辛苦了，谢谢！

本书由我组织翻译，申绍勇负责翻译第 6 至第 8 章，我负责翻译前言、第 1 章至第 5 章、附录并对全书所有章节进行全面审校，还负责对原文中的错误与作者进行沟通并加以修正，这里不得不提的一点就是，当你提交的勘误被确认后，英文原书出版社会送出一个免费的电子图书的购买优惠号，真是太感谢了！参与翻译与审校的还有：杨春花、崔毅、俞哲皆、张琬滢、蒋凌峰、魏伟、万国辉等，在此再次深表感谢。

本书章节安排合理，内容承上启下，但是需要边看书边动手做实验，才能充分理解并掌握 Java 7 并发 API 带来的开发技术及新特性。快乐分享，实践出真知，最后，祝大家能够像我一样在阅读中享受本书带来的乐趣！

Read a bit and take it out, then come back read some more.

俞黎敏

2013 年 7 月 1 日于广州

前言

使用计算机时，可以同时做几件事情：可以一边听音乐，一边使用文字处理软件编辑文档，还可以阅读电子邮件。因为操作系统支持并发任务，从而使得这些工作得以同时进行。并发编程是一种平台和机制供多个任务或程序同时运行，并且互相通讯来交换数据（或者与其他任务进行同步等待）。

Java 是一个并发平台，它提供了大量的类来执行 Java 程序中的并发任务。随着版本的不断更新发展，Java 不断地为程序员增加并发编程的开发功能。本书覆盖了 Java 7 并发 API 中大部分重要而有用的机制，因此，能够直接在应用程序中使用它们，包括下列基本的线程管理：

- ◆ 线程同步机制
- ◆ 通过执行器创建和管理线程
- ◆ 通过 Fork / Join 框架提高应用程序的性能
- ◆ 并发编程的数据结构
- ◆ 根据需要调整一些并发类的默认行为
- ◆ 测试 Java 并发应用程序

本书主要内容

第 1 章，线程管理（Thread Management）将为读者讲解如何通过线程来完成基本的操作。本章将通过基础的范例来讲解线程的创建、执行以及线程的状态管理。

第 2 章，线程同步基础（Basic Thread Synchronization）将为读者讲解如何使用基本的 Java 机制来同步代码。本章将详细阐述 Lock 锁接口和 synchronized 关键字的应用。

第 3 章，线程同步辅助类（Thread Synchronization Utilities）将为读者讲解如何使用 Java 的高级工具类来管理 Java 中的线程同步。本章使用 Java 7 当中的 Phaser 类，来同步被拆分成多个阶段的任务。

第 4 章，线程执行器（Thread Executor）将为读者讲解如何将线程管理委托给执行器（Executor）。执行器将为并发任务负责线程的创建、运行、管理并返回任务的结果。

第 5 章，Fork / Join 框架（Fork/Join Framework）将为读者讲解如何使用 Java 7 新引入的 Fork / Join 框架。它是一种特殊的执行器，用来解决通过分治技术（Divide and Conquer Technique）将任务拆分成多个子任务的问题。

第 6 章，并发集合将为读者讲解如何使用一些由 Java 语言提供的并发数据结构。这些数据结构只能使用在并发编程中，从而避免在程序的实现中采用 synchronized 代码块。

第 7 章，定制并发类（Customizing Concurrency Classes）将为读者讲解如何根据需要来对 Java 并发 API 中一些非常有用的类进行定制。

第 8 章，测试并发应用（Testing Concurrent Application）将为读者讲解如何获取 Java 7 并发 API 中最有用的结构的状态信息。读者也将学习如何使用一些免费的工具来调试并发应用程序，比如：Eclipse、NetBeans IDE。同时也将学习用来检测应用程序中是否存在 Bug 的 FindBugs 开源框架。

第 9 章，附加信息（Additional Information）没有包含在本书中，但是可以通过如下链接免费下载：<http://www.packtpub.com/sites/default/files/downloads/Additional%20%20Information.pdf>。这一章将为读者讲解同步的概念、执行器框架（Executor Framework）和 Fork/Join 框架（Fork/Join Framework）、并发数据结构，以及没有包含在相应章节里的并发对象的监控。

附录，并发编程设计（Concurrent Programming Design）也没有包含在本书当中，但是它可以通过链接免费下载：<http://www.packtpub.com/sites/default/files/downloads/Concurrent%20%20Programming%20Design.pdf>

附录将为读者讲解每一位程序员在开发并发应用程序时应当考虑使用的一些技巧。

阅读本书的前提

为了阅读本书，首先需要读者有 Java 编程语言的基础知识，需要知道如何使用一种集成开发环境（Integrated Development Environment, IDE），比如 Eclipse 或者 NetBeans，但是，这个不是必要的先决条件。

本书适合人群

如果你是一名 Java 开发人员，想进一步掌握并发编程和多线程技术，以及挖掘 Java 7 并发编程的新特性，那么，本书正适合你。你需要已经熟悉普通的 Java 开发实践，如果掌握了线程的基本知识，那么阅读本书将更加得心应手。

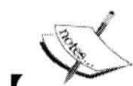
约定

本书有大量不同的文本风格，以此区别不同的信息。这里有一些范例，以及它们的解释。

代码的文字说明将以如下形式说明“继承了 Thread 类，并覆盖了 run()方法”。代码块的组织形式如下：

```
public Calculator(int number) {  
    this.number=number;  
}
```

新术语（New Term）和重要的词（Important Word）将用粗体显示。在屏幕上、菜单里或者对话框上显示的词将用如下形式说明“在菜单条的File菜单栏下通过New Project来创建一个新项目”。



【 警示或重要的备注将像这样在框中显示。】



【 技巧和窍门将像这样显示。】

读者反馈

我们一直欢迎读者们的反馈，让我们知道你对本书的想法，比如你喜欢或者不喜欢。读者反馈对我们来讲是非常重要的。发送反馈给我们相当简单，只需要发送电子邮件到

feedback@packtpub.com 信箱即可，在邮件的主题里提及本书的标题。

如果你是某一方面的技术专家，并且有兴趣编写和出版图书，可以通过 <http://www.packtpub.com/authors> 来获得作者指南。

客户支持

现在你已是 Packt 图书的读者，我们有大量的方式可以让你的购买利益最大化。

下载范例代码

在 <http://www.PacktPub.com> 网站上，通过已注册的账号可以下载到所有已经购买的 Packt 图书的范例代码。如果你已经在别的地方购买了本书，你可以访问 <http://www.PacktPub.com/support> 并注册账号，我们将直接通过邮件把代码发送给你。

勘误

虽然我们已尽力确保内容的准确性，但错误仍有可能发生。如果你在我们的图书中发现错误，哪怕只是一个错误的文字或代码，如果你将此情况告知我们，我们将不胜感激。这样做，可以帮助其他读者为了这个错误而浪费时间，同时也帮助我们提高这本书后续版本的质量。如果你发现任何错误，请访问链接 <http://www.packtpub.com/support>，选择书的标题，通过链接点击勘误提交表单，然后输入勘误表单的详细内容并提交。一旦核实，你提交的勘误将被接受，并将上传到网站的勘误列表中，或添加到标题下的勘误 Errata 一节中的现有勘误列表中。通过访问 <http://www.packtpub.com/support> 链接，选择书的标题，可以查看现有的勘误表。

盗版

所有媒体互联网上的版权材料通过各种媒体进行盗版是一个持续的问题。在 Packt，我们非常重视保护我们的版权和许可。如果你在互联网上遇到以任何形式非法复制和传播我们的作品，请立即向我们提供链接地址或网站名称，这样我们可以立即寻求解决办法。请通过 <mailto:copyright@packtpub.com> 邮箱与我们联系，将怀疑盗版材料的链接告知我们。我们非常感谢能借助你的帮助来保护我们的作者，我们有能力为你带来有价值的内容。

问题

对于本书，如果你有任何问题，可以通过 <mailto:questions@packtpub.com> 邮箱联系我们，我们将尽最大的努力来解决你的问题。

目录

第 1 章 线程管理.....	1
1.1 简介	1
1.2 线程的创建和运行	2
1.3 线程信息的获取和设置	5
1.4 线程的中断	9
1.5 线程中断的控制	11
1.6 线程的休眠和恢复	15
1.7 等待线程的终止	17
1.8 守护线程的创建和运行	20
1.9 线程中不可控异常的处理	24
1.10 线程局部变量的使用	26
1.11 线程的分组	30
1.12 线程组中不可控异常的处理	34
1.13 使用工厂类创建线程	37
第 2 章 线程同步基础.....	41
2.1 简介	41
2.2 使用 synchronized 实现同步方法	42
2.3 使用非依赖属性实现同步	47
2.4 在同步代码中使用条件	53
2.5 使用锁实现同步	57
2.6 使用读写锁实现同步数据访问	61
2.7 修改锁的公平性	65

2.8 在锁中使用多条件 (Multiple Condition)	69
第 3 章 线程同步辅助类	77
3.1 简介	77
3.2 资源的并发访问控制	78
3.3 资源的多副本的并发访问控制	83
3.4 等待多个并发事件的完成	87
3.5 在集合点的同步	91
3.6 并发阶段任务的运行	100
3.7 并发阶段任务中的阶段切换	109
3.8 并发任务间的数据交换	115
第 4 章 线程执行器	120
4.1 简介	120
4.2 创建线程执行器	121
4.3 创建固定大小的线程执行器	126
4.4 在执行器中执行任务并返回结果	129
4.5 运行多个任务并处理第一个结果	134
4.6 运行多个任务并处理所有结果	139
4.7 在执行器中延时执行任务	144
4.8 在执行器中周期性执行任务	147
4.9 在执行器中取消任务	151
4.10 在执行器中控制任务的完成	154
4.11 在执行器中分离任务的启动与结果的处理	158
4.12 处理在执行器中被拒绝的任务	164
第 5 章 Fork / Join 框架	168
5.1 简介	168
5.2 创建 Fork / Join 线程池	170
5.3 合并任务的结果	178
5.4 异步运行任务	187
5.5 在任务中抛出异常	194
5.6 取消任务	199
第 6 章 并发集合	206
6.1 简介	206

III 目录

6.2 使用非阻塞式线程安全列表	207
6.3 使用阻塞式线程安全列表	212
6.4 使用按优先级排序的阻塞式线程安全列表	215
6.5 使用带有延迟元素的线程安全列表	221
6.6 使用线程安全可遍历映射	226
6.7 生成并发随机数	231
6.8 使用原子变量	233
6.9 使用原子数组	237
第 7 章 定制并发类	242
7.1 简介	242
7.2 定制 ThreadPoolExecutor 类	243
7.3 实现基于优先级的 Executor 类	248
7.4 实现 ThreadFactory 接口生成定制线程	252
7.5 在 Executor 对象中使用 ThreadFactory	257
7.6 定制运行在定时线程池中的任务	259
7.7 通过实现 ThreadFactory 接口为 Fork/Join 框架生成定制线程	267
7.8 定制运行在 Fork / Join 框架中的任务	273
7.9 实现定制 Lock 类	278
7.10 实现基于优先级的传输队列	284
7.11 实现自己的原子对象	294
第 8 章 测试并发应用程序	300
8.1 简介	300
8.2 监控 Lock 接口	301
8.3 监控 Phaser 类	305
8.4 监控执行器框架	309
8.5 监控 Fork/Join 池	312
8.6 输出高效的日志信息	317
8.7 使用 FindBugs 分析并发代码	323
8.8 配置 Eclipse 调试并发代码	327
8.9 配置 NetBeans 调试并发代码	330
8.10 使用 MultithreadedTC 测试并发代码	335

第1章

线程管理

本章内容包括：

- ◆ 线程的创建和运行
- ◆ 线程信息的获取和设置
- ◆ 线程的中断
- ◆ 线程中断的控制
- ◆ 线程的休眠和恢复
- ◆ 等待线程的终止
- ◆ 守护线程的创建和运行
- ◆ 线程中不可控异常的处理
- ◆ 线程局部变量的使用
- ◆ 线程的分组
- ◆ 线程组中不可控异常的处理
- ◆ 使用工厂类创建线程

1.1 简介

在计算机领域中，我们说的并发（Concurrency）是指一系列任务的同时运行。如果一台电脑有多个处理器或者有一个多核处理器，这个同时性（Simultaneity）是真正意义的并发；但是一台电脑只有一个单核处理器，这个同时性并不是真正的并发。

现代操作系统都允许多任务的并发执行：在听歌的时候，你可以同时阅读电子邮件，

也可以同时阅读网页上的信息。这种并发是进程级（Process-Level）并发。但在一个进程内也可以有多个同时进行的任务。这种进程内并发的任务成为线程（Thread）。

与并发相关的另一个概念是并行（Parallelism）。与并发有不同的定义一样，并行也有不同的定义。一些学者认为并发是在单核处理器中使用多线程执行应用，与此同时你看到的程序执行只是表面的；相应的，他们认为并行是在多核处理器中使用多线程执行应用，这里的多核处理器可以是一个多核处理器，也可以是同一台电脑上的多个处理器。另一些学者认为并发执行应用的线程是非顺序执行的，相应的，他们认为并行是使用很多线程去简化问题，这些线程是按预定顺序执行的。

本章提供了很多例子来演示运用Java 7 API进行线程的基本操作。你将看到如何在Java程序里创建和运行线程，如何去控制线程的执行，如何把多个线程进行分组，以及如何去操作分组后的线程单元。

1.2 线程的创建和运行

在本章中，我们将学习如何在Java程序中创建和运行线程。在Java语言中，线程跟其他所有元素一样，都是对象（Object）。Java提供了两种方式来创建线程：

- ◆ 继承 Thread 类，并且覆盖 run()方法。
- ◆ 创建一个实现 Runnable 接口的类。使用带参数的 Thread 构造器来创建 Thread 对象。这个参数就是实现 Runnable 接口的类的一个对象。

在本章中，我们将使用第二种方法创建一个简单的程序，这个程序将创建并运行10个线程。每个线程用以计算和打印乘以1~10后的结果，即计算和打印乘法表。

准备工作

本节的范例是在Eclipse IDE里完成的。无论你使用Eclipse还是其他的IDE（比如NetBeans），都可以打开这个IDE并且创建一个新的Java工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 Calculator 的类，它实现了 Runnable 接口。

```
public class Calculator implements Runnable {
```

2. 声明一个名为 **number** 的私有 (**private**) **int** 属性。编写这个类的一个构造器，用来为属性 **number** 设置值。

```
private int number;
public Calculator(int number) {
    this.number=number;
}
```

3. 编写 **run()**方法。这个方法用来执行我们创建的线程的指令，本范例中它将对指定的数字进行乘法表运算。

```
@Override
public void run() {
    for (int i=1; i<=10; i++) {
        System.out.printf("%s: %d * %d = %d\n", Thread.
currentThread().getName(), number, i, i*number);
    }
}
```

4. 现在编写范例的主类。创建一个名为 **Main** 的类，创建的时候同时生成 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

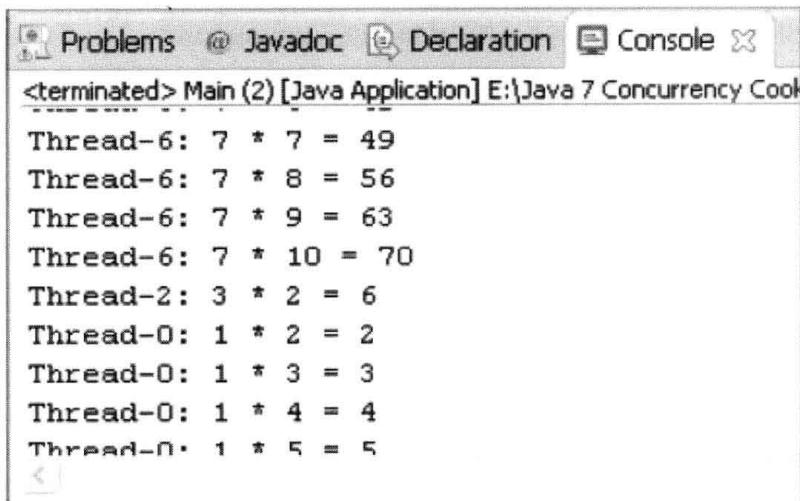
5. 在 **main()**方法中，创建一个执行 10 次的循环。在每次循环中创建一个 **Calculator** 对象，一个 **Thread** 对象，这个 **Thread** 对象使用刚创建的 **Calculator** 对象作为构造器的参数，然后调用刚创建的 **Thread** 对象的 **start()**方法。

```
for (int i=1; i<=10; i++) {
    Calculator calculator=new Calculator(i);
    Thread thread=new Thread(calculator);
    thread.start();
}
```

6. 运行程序，观察不同的线程是如何并行工作的。

工作原理

下面的截图显示了程序的部分运行结果。可以看到我们创建的 10 个线程的运行情况，它们并行的执行既定任务，并将结果显示出来。



The screenshot shows the Eclipse IDE's Console view with the following text output:

```

<terminated> Main (2) [Java Application] E:\Java 7 Concurrency Cook
Thread-6: 7 * 7 = 49
Thread-6: 7 * 8 = 56
Thread-6: 7 * 9 = 63
Thread-6: 7 * 10 = 70
Thread-2: 3 * 2 = 6
Thread-0: 1 * 2 = 2
Thread-0: 1 * 3 = 3
Thread-0: 1 * 4 = 4
Thread-0: 1 * 5 = 5

```

每个 Java 程序都至少有一个执行线程。当运行程序的时候，JVM 将启动这个执行线程来调用程序的 main()方法。

当调用 Thread 对象的 start()方法时，另一个执行线程将被创建。因而在我们的程序中，每次调用 start()方法时，都会创建一个执行线程。

当一个程序的所有线程都运行完成时，更明确的说，当所有非守护(non-daemon)线程都运行完成的时候，这个 Java 程序将宣告结束。如果初始线程（执行 main()方法的线程）结束了，其余的线程仍将继续执行直到它们运行结束。如果某一个线程调用了 System.exit()指令来结束程序的执行，所有的线程都将结束。

对一个实现了 Runnable 接口的类来说，创建 Thread 对象并不会创建一个新的执行线程；同样的，调用它的 run()方法，也不会创建一个新的执行线程。只有调用它的 start()方法时，才会创建一个新的执行线程。

更多信息

在本章简介中提到过还有另一种方法能够创建新的执行线程。编写一个类并继承 Thread 类，在这个类里覆盖 run()方法，然后创建这个类的对象，并且调用 start()方法，也会创建一个执行线程。

参见

- ◆ 参见本书 1.13 节。

1.3 线程信息的获取和设置

Thread 类有一些保存信息的属性，这些属性可以用来标识线程，显示线程的状态或者控制线程的优先级。

ID: 保存了线程的唯一标示符。

Name: 保存了线程名称

Priority: 保存了线程对象的优先级。线程的优先级是从 1 到 10，其中 1 是最低优先级；10 是最高优先级。我们并不推荐去改变线程的优先级，然而，在需要的时候，也可以这么做。

Status: 保存了线程的状态。在 Java 中，线程的状态有 6 种：new、runnable、blocked、waiting、time waiting 或者 terminated。

在本节，我们将编写程序为 10 个线程指定名称和优先级，并且输出它们的状态信息直到线程结束。每个线程都将计算一个数字的乘法表。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 Calculator 的类，它实现了 Runnable 接口。

```
public class Calculator implements Runnable {
```

2. 声明一个名为 number 的私有 int 属性。编写这个类的一个构造器，用来为属性 number 设置值。

```
private int number;
public Calculator(int number) {
    this.number=number;
}
```

3. 编写 run()方法。这个方法用来执行我们创建的线程的指令，本范例中它将对指定的数字进行乘法表运算。

```
@Override  
public void run() {  
    for (int i=1; i<=10; i++) {  
        System.out.printf("%s: %d * %d = %d\n", Thread.  
currentThread().getName(), number, i, i*number);  
    }  
}
```

4. 编写范例的主类。创建一个名为 Main 的类，创建的时候同时生成 main()方法。

```
public class Main {  
    public static void main(String[] args) {
```

5. 创建一个容量为 10 的线程数组，以用来存储线程；创建一个容量为 10 的 Thread.State 数组，以用来存放这 10 个线程运行时的状态。

```
    Thread threads[] = new Thread[10];  
    Thread.State status[] = new Thread.State[10];
```

6. 创建一个容量为 10 的 Calculator 对象数组，为每个对象都设置不同的数字，然后使用它们作为 Thread 构造器的参数来创建 10 个线程对象。并且将其中 5 个线程的优先级设置为最高，另外 5 个线程的优先级设置为最低。

```
    for (int i=0; i<10; i++) {  
        threads[i] = new Thread(new Calculator(i));  
        if ((i%2)==0){  
            threads[i].setPriority(Thread.MAX_PRIORITY);  
        } else {  
            threads[i].setPriority(Thread.MIN_PRIORITY);  
        }  
        threads[i].setName("Thread "+i);  
    }
```

7. 创建一个 PrintWriter 对象，用来把线程的状态演变写入到文件中。

```
    try (FileWriter file = new FileWriter("./\\data\\log.txt");  
PrintWriter pw = new PrintWriter(file)){
```

8. 把这 10 个线程的状态写入文件中。现在线程的状态是 NEW。

```
for (int i=0; i<10; i++) {
    pw.println("Main : Status of Thread "+i+" : " +
    threads[i].getState());
    status[i]=threads[i].getState();
}
```

9. 开始执行 10 个线程。

```
for (int i=0; i<10; i++) {
    threads[i].start();
}
```

10. 直到 10 个线程都运行完成，我们就可以查看他们的状态。所有任何一个线程的状态发生了变化，我们就会将它写入到文件中。

```
boolean finish=false;
while (!finish) {
    for (int i=0; i<10; i++) {
        if (threads[i].getState()!=status[i]) {
            writeThreadInfo(pw, threads[i], status[i]);
            status[i]=threads[i].getState();
        }
    }
    finish=true;
    for (int i=0; i<10; i++) {
        finish=finish && (threads[i].getState()==State.TERMINATED);
    }
}
```

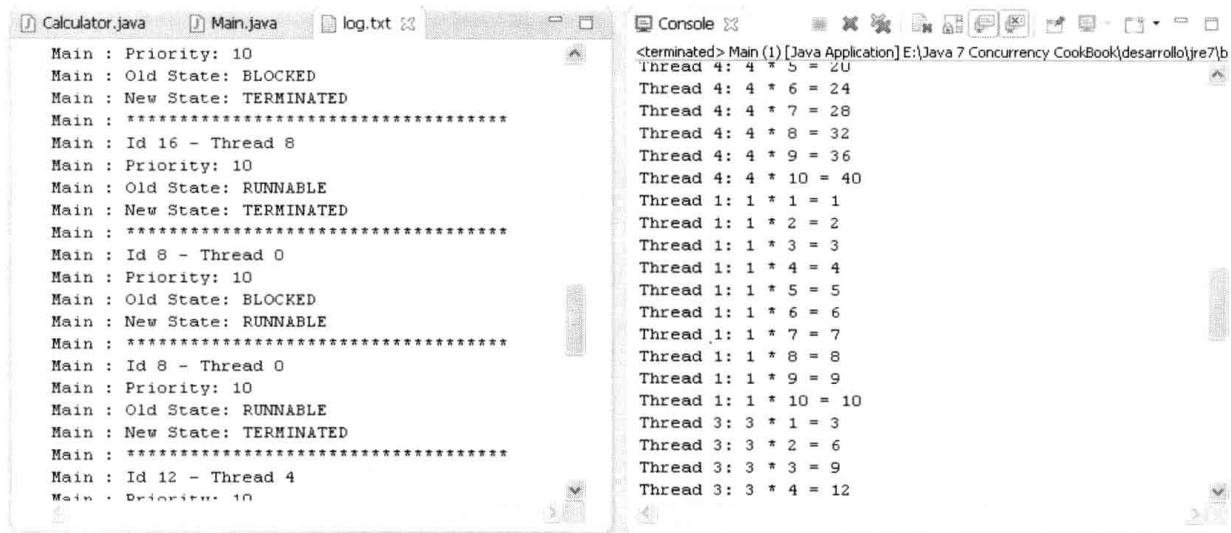
11. 编写 writeThreadInfo()方法，用来写下线程的 ID、名称、优先级、旧的状态和新的状态。

```
private static void writeThreadInfo(PrintWriter pw, Thread
thread, State state) {
    pw.printf("Main : Id %d - %s\n", thread.getId(), thread.getName());
    pw.printf("Main : Priority: %d\n", thread.getPriority());
    pw.printf("Main : Old State: %s\n", state);
    pw.printf("Main : New State: %s\n", thread.getState());
    pw.printf("Main : *****\n");
}
```

12. 运行这个范例，然后打开 log.txt 文件来查看 10 个线程的状态演变。

工作原理

下面的截屏是 log.txt 文件的一部分。在这个文件里，我们可以看到最高优先级的线程比最低优先级的线程结束得早。我们也可以看到每个线程的状态演变。



```
Main : Priority: 10
Main : Old State: BLOCKED
Main : New State: TERMINATED
Main : *****
Main : Id 16 - Thread 8
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : *****
Main : Id 8 - Thread 0
Main : Priority: 10
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : *****
Main : Id 8 - Thread 0
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : *****
Main : Id 12 - Thread 4
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : *****
<terminated> Main(1) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\b
Thread 4: 4 * 5 = 20
Thread 4: 4 * 6 = 24
Thread 4: 4 * 7 = 28
Thread 4: 4 * 8 = 32
Thread 4: 4 * 9 = 36
Thread 4: 4 * 10 = 40
Thread 1: 1 * 1 = 1
Thread 1: 1 * 2 = 2
Thread 1: 1 * 3 = 3
Thread 1: 1 * 4 = 4
Thread 1: 1 * 5 = 5
Thread 1: 1 * 6 = 6
Thread 1: 1 * 7 = 7
Thread 1: 1 * 8 = 8
Thread 1: 1 * 9 = 9
Thread 1: 1 * 10 = 10
Thread 3: 3 * 1 = 3
Thread 3: 3 * 2 = 6
Thread 3: 3 * 3 = 9
Thread 3: 3 * 4 = 12
```

这个程序的乘法表运算显示在控制台上，每个线程的状态演变记录在 log.txt 里。这样你可以更清楚地看到线程的演变过程。

Thread 类的属性存储了线程的所有信息。JVM 使用线程的 priority 属性来决定某一刻由哪个线程来使用 CPU，并且根据线程的情景为它们设置实际状态。

如果没有为线程指定一个名字，JVM 将自动给它分配一个名字，格式是 Thread-XX，其中 XX 是一组数字。线程的 ID 和状态是不允许被修改的，线程类没有提供 setId()和 setStatus()方法来修改它们。

更多信息

通过本节，你已经学会了如何通过 Thread 对象访问属性信息。但是，也可以通过实现 Runnable 接口的对象来访问这些属性信息。如果一个线程是以 Runnable 对象为参数构建的，那么也可以使用 Thread 类的静态方法 currentThread() 来访问这个线程对象。

要注意的是，如果使用 setPriority() 方法设置的优先级不是从 1 到 10 这个范围内的值，运行时就会抛出 IllegalArgumentException 异常。

参见

- ◆ 参见本书 1.4 节。

1.4 线程的中断

如果一个 Java 程序有不止一个执行线程，当所有线程都运行结束的时候，这个 Java 程序才能运行结束；更确切地说应该是所有的非守护线程运行结束时，或者其中一个线程调用了 `System.exit()` 方法时，这个 Java 程序才运行结束。如果你想终止一个程序，或者程序的某个用户试图取消线程对象正在运行的任务，就需要结束这个线程。

Java 提供了中断机制，我们可以使用它来结束一个线程。这种机制要求线程检查它是否被中断了，然后决定是不是响应这个中断请求。线程允许忽略中断请求并且继续执行。

在本节中，我们将开发程序来创建一个线程，使其运行 5 秒钟后再通过中断机制强制使其终止。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 `PrimeGenerator` 的类，并继承 `Thread` 类。

```
public class PrimeGenerator extends Thread{
```

2. 覆盖 `run()` 方法，并在方法体内包含一个无限循环。在每次循环中，我们将处理从 1 开始的连续数。对每个数字，我们将计算它是不是一个质数，如果是的话就打印到控制台。

```
@Override
public void run() {
    long number=1L;
    while (true) {
        if (isPrime(number)) {
            System.out.printf("Number %d is Prime",number);
```

```
}
```

3. 一个数字处理完后，调用 `isInterrupted()` 方法来检查线程是否被中断。如果 `isInterrupted()` 返回值是 `true`，就写一个信息并且结束线程的执行。

```
if (isInterrupted()) {
    System.out.printf("The Prime Generator has been
Interrupted");
    return;
}
number++;
}
}
```

4. 实现 `isPrime()` 方法。`isPrime()` 方法返回的是一个布尔值，如果接收到的参数是一个质数就返回 `true`，否则就返回 `false`。

```
private boolean isPrime(long number) {
    if (number <=2) {
        return true;
    }
    for (long i=2; i<number; i++) {
        if ((number % i)==0) {
            return false;
        }
    }
    return true;
}
```

5. 现在我们来实现这个范例的主类 `Main`，并且实现 `main()` 方法。

```
public class Main {
    public static void main(String[] args) {
```

6. 创建 `PrimeGenerator` 类的一个对象，并且运行这个线程对象。

```
    Thread task=new PrimeGenerator();
    task.start();
```

7. 等待 5 秒钟后，中断 `PrimeGenerator` 线程。

```
    try {
        Thread.sleep(5000);
```

```

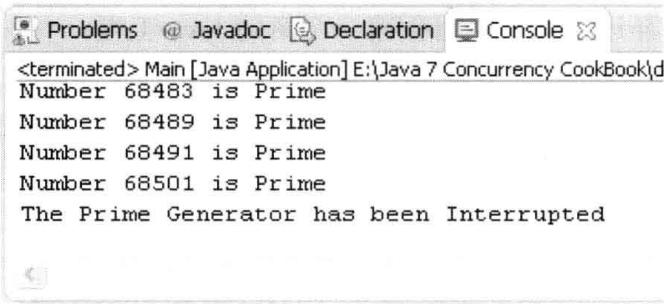
} catch (InterruptedException e) {
    e.printStackTrace();
}
task.interrupt();

```

8. 运行范例并查看结果。

工作原理

下面的截屏记录了上述范例的运行结果。通过这个图我们可以看到 PrimeGenerator 线程打印出的信息，并且看到当它被中断后就运行终止了。



Thread 类有一个表明线程被中断与否的属性，它存放的是布尔值。线程的 interrupt() 方法被调用时，这个属性就会被设置为 true。isInterrupted()方法只是返回这个属性的值。

更多信息

还有一个方法可以检查线程是否已被中断，即 Thread 类的静态方法 interrupted()，用来检查当前执行的线程是否被中断。

isInterrupted()和 interrupted()方法有一个很大的区别。isInterrupted()不能改变 interrupted 属性的值，但是后者能设置 interrupted 属性为 false。因为 interrupted()是一个静态方法，更推荐使用 isInterrupted()方法。

像之前提到的，线程可以忽略中断，但并不是预期的行为。

1.5 线程中断的控制

通过上一节，你已经学会了如何去中断执行中的线程，也学会了如何在线程对象中去控制这个中断。上一个例子中使用的机制，可以使用在线程很容易被中断的情况下。但是，

如果线程实现了复杂的算法并且分布在几个方法中，或者线程里有递归调用的方法，我们就得使用一个更好的机制来控制线程的中断。为了达到这个目的，Java 提供了 `InterruptedException` 异常。当检查到线程中断的时候，就抛出这个异常，然后在 `run()` 中捕获并处理这个异常。

在本节中，我们将实现线程类来完成下面的内容，它在一个文件夹及其子文件夹中寻找一个指定的文件。这个范例将示范如何用 `InterruptedException` 异常来控制线程的中断。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 `FileSearch` 的类，并且实现 `Runnable` 接口。

```
public class FileSearch implements Runnable {
```

2. 声明两个私有属性，一个是我们将要查找的文件名称，另一个是初始文件夹。实现这个类的构造器，用来初始化这两个属性。

```
private String initPath;
private String fileName;
public FileSearch(String initPath, String fileName) {
    this.initPath = initPath;
    this.fileName = fileName;
}
```

3. 在 `FileSearch` 中实现 `run()` 方法。它将检查 `fileName` 属性是不是一个目录，如果是，就调用 `processDirectory()` 方法。`processDirectory()` 方法会抛出 `InterruptedException` 异常，因此必须捕获并处理这个异常。

```
@Override
public void run() {
    File file = new File(initPath);
    if (file.isDirectory()) {
        try {
```

```
        directoryProcess(file);
    } catch (InterruptedException e) {
        System.out.printf("%s: The search has been
interrupted", Thread.currentThread().getName());
    }
}
```

4. 实现 `directoryProcess()`方法，这个方法会获取一个文件夹里的所有文件和子文件夹，并进行处理。对于每一个目录，这个方法将递归调用，并且用相应目录名作为传入参数。对于每个文件，这个方法将调用 `fileProcess()`方法。处理完所有的文件和文件夹后，这个方法将检查线程是不是被中断了，如果是，就抛出 `InterruptedException` 异常。

```
private void directoryProcess(File file) throws  
InterruptedException {  
    File list[] = file.listFiles();  
    if (list != null) {  
        for (int i = 0; i < list.length; i++) {  
            if (list[i].isDirectory()) {  
                directoryProcess(list[i]);  
            } else {  
                fileProcess(list[i]);  
            }  
        }  
    }  
    if (Thread.interrupted()) {  
        throw new InterruptedException();  
    }  
}
```

5. 实现 `processFile()`方法。这个方法将比较当前文件的文件名和要查找的文件名，如果文件名匹配，就将信息打印到控制台。做完比较后，线程将检查是不是被中断了，如果是，它将抛出 `InterruptedException` 异常。

```
private void fileProcess(File file) throws InterruptedException
{
    if (file.getName().equals(fileName)) {
        System.out.printf("%s : %s\n", Thread.currentThread().
getName() ,file.getAbsolutePath());
    }
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
}
```

```

    }
}

```

6. 现在，我们实现这个范例的主类。实现一个包含 main()方法的 Main 类。

```

public class Main {
    public static void main(String[] args) {

```

7. 创建 FileSearch 类的一个对象，并用它作为传入参数来创建一个线程对象，然后启动线程执行任务。

```

FileSearch searcher=new FileSearch("C:\\\\","autoexec.bat");
Thread thread=new Thread(searcher);
thread.start();

```

8. 等待 10 秒钟，然后中断线程。

```

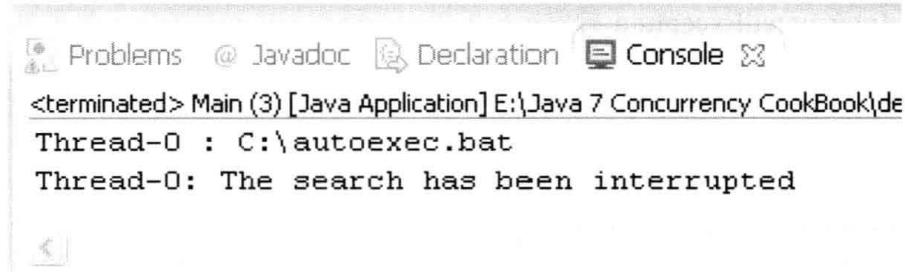
try {
    TimeUnit.SECONDS.sleep(10);
} catch (InterruptedException e) {
    e.printStackTrace();
}
thread.interrupt();
}

```

9. 运行这个范例并查看结果。

工作原理

下面的截屏记录了范例的运行结果。你可以看到当 FileSearch 对象检测到它被中断之后就结束了运行。



```

Problems @ Javadoc Declaration Console
<terminated> Main (3) [Java Application] E:\Java 7 Concurrency CookBook\de
Thread-0 : C:\autoexec.bat
Thread-0: The search has been interrupted

```

在本范例中，我们使用了 Java 异常来控制线程的中断。当运行这个范例时，程序将进

入文件夹查找是否包含指定的文件。例如，如果要查找的文件夹目录结构是**\b\c\d**，这个程序将递归调用 `processDirectory()`方法 3 次。不管递归调用了多少次，只要线程检测到它已经被中断了，就会立即抛出 `InterruptedException` 异常，然后继续执行 `run()`方法。

更多信息

与并发 API 相关的 Java 方法将会抛出 `InterruptedException` 异常，如 `sleep()`方法。

参见

- ◆ 参见本书 1.4 节。

1.6 线程的休眠和恢复

有些时候，你需要在某一个预期的时间中断线程的执行。例如，程序的一个线程每隔一分钟检查一次传感器状态，其余时间什么都不做。在这段空闲时间，线程不占用计算机的任何资源。当它继续执行的 CPU 时钟来临时，JVM 会选中它继续执行。可以通过线程的 `sleep()`方法来达到这个目标。`sleep()`方法接受整型数值作为参数，以表明线程挂起执行的毫秒数。当线程休眠的时间结束了，JVM 会分给它 CPU 时钟，线程将继续执行它的指令。

`sleep()`方法的另一种使用方式是通过 `TimeUnit` 枚举类元素进行调用。这个方法也使用 `Thread` 类的 `sleep()`方法来使当前线程休眠，但是它接收的参数单位是秒，最后会被转化成毫秒。

在本节中，我们将开发程序来完成这样的内容：使用 `sleep()`方法，每间隔一秒就输出实际时间。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 FileClock 的类，并且实现 Runnable 接口。

```
public class FileClock implements Runnable {
```

2. 实现 run()方法。

```
@Override
public void run() {
```

3. 编写一个执行 10 次的循环。在每个循环中，创建一个 Date 对象，并把它写入到文件中，然后调用 TimeUnit 类的 SECONDS 属性的 sleep()方法来挂起线程一秒钟。这个值将让线程休眠大概 1 秒钟。sleep()方法会抛出 InterruptedException 异常，我们必须捕获并处理这个异常。最佳实践是，当线程被中断时，释放或者关闭线程正在使用的资源。

```
for (int i = 0; i < 10; i++) {
    System.out.printf("%s\n", new Date());
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        System.out.printf("The FileClock has been interrupted");
    }
}
```

4. 实现范例的主类。创建一个名为 FileMain 的类并包含 main()方法。

```
public class FileMain {
    public static void main(String[] args) {
```

5. 创建 FileClock 类的一个对象，并用它作为传入参数来创建一个 Thread 对象，然后运行这个线程。

```
FileClock clock=new FileClock();
Thread thread=new Thread(clock);
thread.start();
```

6. 调用 TimeUnit 类的 SECONDS 属性的 sleep()方法，休眠 5 秒钟。

```
try {
    TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) {
```

```
    e.printStackTrace();  
};
```

7. 中断 FileClock 线程。

```
thread.interrupt();
```

8. 运行这个范例并且观察结果。

工作原理

当运行这个范例时，你可以看到程序每间隔一秒钟就会输出实际的时间，接下来是 FileClock 线程已经被中断的信息。

当调用 sleep()方法之后，线程会释放 CPU 并且不再继续执行任务。在这段时间内，线程不占用 CPU 时钟，所以 CPU 可以执行其他的任务。

如果休眠中线程被中断，该方法就会立即抛出 InterruptedException 异常，而不需要等待到线程休眠时间结束。

更多信息

Java 并发 API 还提供了另外一个方法来使线程对象释放 CPU，即 yield()方法，它将通知 JVM 这个线程对象可以释放 CPU 了。JVM 并不保证遵循这个要求。通常来说，yield()方法只做调试使用。

1.7 等待线程的终止

在一些情形下，我们必须等待线程的终止。例如，我们的程序在执行其他的任务时，必须先初始化一些必须的资源。可以使用线程来完成这些初始化任务，等待线程终止，再执行程序的其他任务。

为了达到这个目的，我们使用 Thread 类的 join()方法。当一个线程对象的 join()方法被调用时，调用它的线程将被挂起，直到这个线程对象完成它的任务。

在本节中，我们将通过初始化资源的范例来学习 join()方法。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 DataSourcesLoader 的类，并且实现 Runnable 接口。

```
public class DataSourcesLoader implements Runnable {
```

2. 实现 run()方法。这个方法先显示一个表明它开始执行的信息，然后休眠 4 秒钟，再显示另一个信息表明已完成当前执行。

```
@Override
public void run() {
    System.out.printf("Beginning data sources loading: %s\n", new
Date());
    try {
        TimeUnit.SECONDS.sleep(4);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Data sources loading has finished:
%s\n", new Date());
}
```

3. 创建一个 NetworkConnectionsLoader 类，用以实现 Runnable 接口。实现 run()方法的方式与 DataSourcesLoader 的 run()方法类似，但是它休眠 6 秒钟。

4. 创建一个包含 main()方法的 Main 类。

```
public class Main {
    public static void main(String[] args) {
```

5. 创建一个 DataSourcesLoader 对象，并用它作为传入参数来创建一个线程。

```
DataSourcesLoader dsLoader = new DataSourcesLoader();
Thread thread1 = new Thread(dsLoader, "DataSourceThread");
```

6. 创建一个 NetworkConnectionsLoader 对象，并用它作为传入参数来创建一个线程。

```
NetworkConnectionsLoader ncLoader = new
NetworkConnectionsLoader();
Thread thread2 = new Thread(ncLoader, "NetworkConnectionLoad
er");
```

7. 调用 start()方法启动这两个线程对象。

```
thread1.start();
thread2.start();
```

8. 使用 join()方法等待两个线程的终止。join()方法会抛出 InterruptedException 异常，我们必须捕获并处理这个异常。

```
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

9. 程序运行结束时，打印出信息。

```
System.out.printf("Main: Configuration has been loaded:
%s\n", new Date());
```

10. 运行程序并观察运行结果。

工作原理

运行这个程序时，你会看到两个线程对象是如何运行的。DataSourcesLoader 线程运行结束，NetworkConnectionsLoader 线程也运行结束的时候，主线程对象才会继续运行并且打印出最终的信息。

更多信息

Java 提供了另外两种形式的 join()方法：

`join (long milliseconds)`

join (long milliseconds, long nanos)

当一个线程调用其他某个线程的 join()方法时，如果使用的是第一种 join()方式，那么它不必等到被调用线程运行终止，如果参数指定的毫秒时钟已经到达，它将继续运行。例如，thread1 中有这样的代码 thread2.join(1000)，thread1 将挂起运行，直到满足下面两个条件之一：

- ◆ thread2 运行已经完成；
- ◆ 时钟已经过去 1000 毫秒。

当两个条件中的任何一条成立时，join()方法将返回。

第二种 join()方法跟第一种相似，只是需要接受毫秒和纳秒两个参数。

1.8 守护线程的创建和运行

Java 里有一种特殊的线程叫做守护（Daemon）线程。这种线程的优先级很低，通常来说，当同一个应用程序里没有其他的线程运行的时候，守护线程才运行。当守护线程是程序中唯一运行的线程时，守护线程执行结束后，JVM 也就结束了这个程序。

因为这种特性，守护线程通常被用来做为同一程序中普通线程（也称为用户线程）的服务提供者。它们通常是无限循环的，以等待服务请求或者执行线程的任务。它们不能做重要的工作，因为我们不可能知道守护线程什么时候能够获取 CPU 时钟，并且，在没有其他线程运行的时候，守护线程随时可能结束。一个典型的守护线程是 Java 的垃圾回收器（Garbage Collector）。

在本节中，我们将通过范例学到如何创建守护线程，范例程序包含两个线程；一个用户线程，它将事件写入到一个队列中；另一个是守护线程，它将管理这个队列，如果生成的事件超过 10 秒钟，就会被移除。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建 Event 类。这个类只存放满足本范例需要的信息。声明两个私有属性，一个日期类型的属性 date；另一个字符串型的属性 event。并生成这两个属性的读写方法。
2. 创建 WriterTask 类，用以实现 Runnable 接口。

```
public class WriterTask implements Runnable {
```

3. 声明一个存放 Event 对象的队列，并实现一个带参数的构造器，来初始化这个队列对象。

```
private Deque<Event> deque;
public WriterTask (Deque<Event> deque) {
    this.deque=deque;
}
```

4. 实现线程的 run() 方法。它将执行 100 次循环。在每次循环中，都会创建一个新的 Event 对象，并放入到队列中，然后休眠一秒钟。

```
@Override
public void run() {
    for (int i=1; i<100; i++) {
        Event event=new Event();
        event.setDate(new Date());
        event.setEvent(String.format("The thread %s has generated an
event",Thread.currentThread().getId()));
        deque.addFirst(event);
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5. 创建 CleanerTask 类并继承 Thread 类。

```
public class CleanerTask extends Thread {
```

6. 声明存放 Event 对象的队列，并实现一个带参数的构造器，来初始化这个队列对象。同时，在这个构造器中，通过 setDaemon() 方法把这个线程设置为守护线程。

```
private Deque<Event> deque;
```

```
public CleanerTask(Deque<Event> deque) {
    this.deque = deque;
    setDaemon(true);
}
```

7. 实现 run()方法。它将无限制的重复运行，在每次运行中，将获取当前时间，并调用 clean()方法。

```
@Override
public void run() {
    while (true) {
        Date date = new Date();
        clean(date);
    }
}
```

8. 实现 clean()方法。clean()将读取队列的最后一个事件对象，如果这个事件是 10 秒钟前创建的，就将它删除并且检查下一个。如果有事件被删除，clean()将打印出这个被删除事件的信息，也打印出队列的长度，这样，我们就可以看到程序的演化过程。

```
private void clean(Date date) {
    long difference;
    boolean delete;
    if (deque.size()==0) {
        return;
    }
    delete=false;
    do {
        Event e = deque.getLast();
        difference = date.getTime() - e.getDate().getTime();
        if (difference > 10000) {
            System.out.printf("Cleaner: %s\n", e.getEvent());
            deque.removeLast();
            delete=true;
        }
    } while (difference > 10000);
    if (delete){
        System.out.printf("Cleaner: Size of the queue: %d\n", deque.
size());
    }
}
```

9. 现在实现主类。创建一个包含 main()方法的 Main 类。

```
public class Main {
    public static void main(String[] args) {
```

10. 创建一个队列对象 Deque，用来存放事件。

```
Deque<Event> deque=new ArrayDeque<Event>();
```

11. 创建三个 WriterTask 线程和一个 CleanerTask 线程，并启动它们。

```
WriterTask writer=new WriterTask(deque);
for (int i=0; i<3; i++){
    Thread thread=new Thread(writer);
    thread.start();
}
CleanerTask cleaner=new CleanerTask(deque);
cleaner.start();
```

12. 运行程序并且查看结果。

工作原理

对程序的运行输出进行分析之后，我们会发现，队列中的对象会不断增长直到 30 个，然后到程序结束，队列的长度维持在 27~30 之间。

这个程序有 3 个 WriterTask 线程，每个线程向队列写入一个事件，然后休眠 1 秒钟。在第一个 10 秒钟内，队列中有 30 个事件，直到 3 个 WriterTask 都休眠后，CleanerTask 才开始执行，但是它没有删除任何事件。因为所有的事件都小于 10 秒钟。在接下来的运行中，CleanerTask 每秒删除 3 个对象，同时 WriterTask 会写入 3 个对象，所以队列的长度一直介于 27~30 之间。

你可以不断调试 WriterTask 休眠的时间。如果使用一个更小的值，会发现 CleanerTask 将有更少的 CPU 时间，并且队列的长度将增加，因为 CleanerTask 没有删除对象。

更多信息

setDaemon()方法只能在 start()方法被调用之前设置。一旦线程开始运行，将不能再修改守护状态。

`isDaemon()`方法被用来检查一个线程是不是守护线程，返回值 `true` 表示这个线程是守护线程，`false` 表示这个线程是用户线程。

1.9 线程中不可控异常的处理

在 Java 中有两种异常。

- ◆ 非运行时异常(Checked Exception): 这种异常必须在方法声明的 `throws` 语句指定，或者在方法体内捕获。例如：`IOException` 和 `ClassNotFoundException`。
- ◆ 运行时异常(Unchecked Exception): 这种异常不必在方法声明中指定，也不需要在方法体中捕获。例如：`NumberFormatException`。

因为 `run()`方法不支持 `throws` 语句，所以当线程对象的 `run()`方法抛出非运行异常时，我们必须捕获并且处理它们。当运行时异常从 `run()`方法中抛出时，默认行为是在控制台输出堆栈记录并且退出程序。

好在，Java 提供给我们一种在线程对象里捕获和处理运行时异常的一种机制。

在本节中，我们将通过范例学习这种机制。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现用来处理运行时异常的类。这个类实现 `UncaughtExceptionHandler` 接口并且实现这个接口的 `uncaughtException()`方法。我们的范例将使用 `ExceptionHandler` 类的 `uncaughtException()`方法打印出异常信息和抛出异常的线程代码。代码如下：

```
public class ExceptionHandler implements UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.printf("An exception has been captured\n");
        System.out.printf("Thread: %s\n", t.getId());
        System.out.printf("Exception: %s: %s\n", e.getClass().
            getName(), e.getMessage());
```

```

        System.out.printf("Stack Trace: \n");
        e.printStackTrace(System.out);
        System.out.printf("Thread status: %s\n", t.getState());
    }
}

```

2. 实现一个抛出运行时异常的线程类，命名为 Task，它实现了 Runnable 接口，在实现 run()方法时强制抛出运行时异常。例如，把一个 String 值转换成 int 值。

```

public class Task implements Runnable {
    @Override
    public void run() {
        int numero=Integer.parseInt("TTT");
    }
}

```

3. 实现范例的主程序。实现一个包含 main()方法的 Main 类。

```

public class Main {
    public static void main(String[] args) {

```

4. 创建一个 Task 对象，并用它作为传入参数来创建一个 Thread 对象。接着调用 setUncaughtExceptionHandler()方法设置线程的运行时异常处理器，然后启动这个线程。

```

        Task task=new Task();
        Thread thread=new Thread(task);
        thread.setUncaughtExceptionHandler(new ExceptionHandler());
        thread.start();
    }
}

```

5. 运行范例并查看运行结果。

工作原理

下面的截屏记录了范例的运行结果。当异常抛出并被异常处理器捕获后，将在控制台打印出异常信息和抛出异常的线程代码。

当一个线程抛出了异常并且没有被捕获时（这种情况只可能是运行时异常），JVM 检查这个线程是否被预置了未捕获异常处理器。如果找到，JVM 将调用线程对象的这个方法，并将线程对象和异常作为传入参数。

```

<terminated> Main (6) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (11/08/2012 14:27:32)
An exception has been captured
Thread: 8
Exception: java.lang.NumberFormatException: For input string: "TTT"
Stack Trace:
java.lang.NumberFormatException: For input string: "TTT"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at com.packtpub.java7.concurrency.chapter1.recipe8.task.Task.run(Task.java:16)
    at java.lang.Thread.run(Unknown Source)
Thread status: RUNNABLE
Thread has finished

```

如果线程没有被预置未捕获异常处理器，JVM 将打印堆栈记录到控制台，并退出程序。

更多信息

Thread 类还有另一个方法可以处理未捕获到的异常，即静态方法 `setDefaultUncaughtExceptionHandler()`。这个方法在应用程序中为所有的线程对象创建了一个异常处理器。

当线程抛出一个未捕获到的异常时，JVM 将为异常寻找以下三种可能的处理器。

首先，它查找线程对象的未捕获异常处理器。如果找不到，JVM 继续查找线程对象所在的线程组（`ThreadGroup`）的未捕获异常处理器，这将在“线程组中不可控异常的处理”一节中讲解。如果还是找不到，如同本节所讲的，JVM 将继续查找默认的未捕获异常处理器。

如果没有一个处理器存在，JVM 则将堆栈异常记录打印到控制台，并退出程序。

参见

- ◆ 参见 1.12 节。

1.10 线程局部变量的使用

共享数据是并发程序最核心的问题之一，对于继承了 `Thread` 类或者实现了 `Runnable` 接口的对象来说尤其重要。

如果创建的对象是实现了 `Runnable` 接口的类的实例，用它作为传入参数创建多个线程

对象并启动这些线程，那么所有的线程将共享相同的属性。也就是说，如果你在一个线程中改变了一个属性，所有线程都会被这个改变影响。

在某种情况下，这个对象的属性不需要被所有线程共享。Java 并发 API 提供了一个干净的机制，即线程局部变量（Thread-Local Variable），其具有很好的性能。

本节中，我们将创建两个程序：第一个具有刚才提到的问题，另一个使用线程局部变量机制解决了这个问题。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例

- 使要实现的范例具有之前提到的共享问题。创建一个名为 UnsafeTask 的类，它实现了 Runnable 接口。声明一个私有的 java.util.Date 属性。

```
public class UnsafeTask implements Runnable{
    private Date startDate;
```

- 实现 run()方法，这个方法将初始化 startDate 属性，并且将值打印到控制台，让线程休眠一个随机时间，然后再次将 startDate 的值打印到控制台。

```
@Override
public void run() {
    startDate=new Date();
    System.out.printf("Starting Thread: %s : %s\n",Thread.
currentThread().getId(),startDate);
    try {
        TimeUnit.SECONDS.sleep( (int)Math.sqrt(Math.random()*10));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Thread Finished: %s : %s\n",Thread.
currentThread().getId(),startDate);
}
```

3. 实现这个有问题的应用程序的主程序。创建一个包含 main()方法的 Main 类。这个方法将创建一个 UnsafeTask 类对象，用它作为传入参数创建 10 个线程对象并启动这 10 个线程，每个线程的启动间隔 2 秒。

```
public class Core {
    public static void main(String[] args) {
        UnsafeTask task=new UnsafeTask();
        for (int i=0; i<10; i++){
            Thread thread=new Thread(task);
            thread.start();
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

4. 在下面的截屏中，你将看到这个程序执行的结果。每个线程有一个不同的开始时间，但是当它们结束时，三个线程都有相同的 startDate 属性值。

```
Problems @ Javadoc Declaration Console X
<terminated> Main (7) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre
Starting Thread: 8 : Sat Aug 11 22:11:01 CEST 2012
Starting Thread: 9 : Sat Aug 11 22:11:03 CEST 2012
Starting Thread: 10 : Sat Aug 11 22:11:05 CEST 2012
Thread Finished: 8 : Sat Aug 11 22:11:05 CEST 2012
Thread Finished: 10 : Sat Aug 11 22:11:05 CEST 2012
Thread Finished: 9 : Sat Aug 11 22:11:05 CEST 2012
```

5. 如之前提到的，我们将使用线程局部变量机制来解决这个问题。

6. 创建一个 SafeTask 类，用以实现 Runnable 接口

```
public class SafeTask implements Runnable {
```

7. 声明一个 ThreadLocal<Date> 对象。这个对象是在 initialValue() 方法中隐式实现的。这个方法将返回当前日期。

```

private static ThreadLocal<Date> startDate= new
    ThreadLocal<Date>() {
        protected Date initialValue(){
            return new Date();
        }
    };

```

8. 实现 run()方法。它跟 UnsafeTask 类的 run()方法实现了一样的功能，但是访问 startDate 属性的方式改变了。

```

@Override
public void run() {
    System.out.printf("Starting Thread: %s : %s\n",Thread.
currentThread().getId(),startDate.get());
    try {
        TimeUnit.SECONDS.sleep((int)Math.rint(Math.random()*10));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Thread Finished: %s : %s\n",Thread.
currentThread().getId(),startDate.get());
}

```

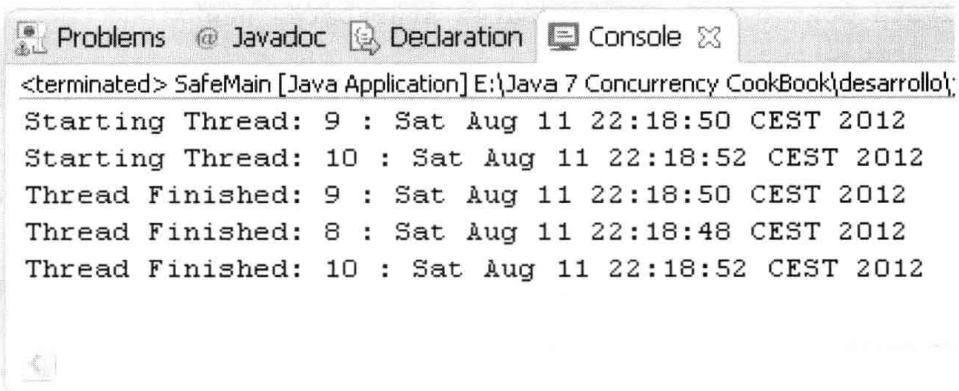
9. 这个范例的入口类与第一个范例一样，只是创建并作为参数参入的 Runnable 类对象不同而已。

10. 运行范例，并分析两个范例之间的不同。

工作原理

在下面的截屏中，你将看到安全线程类的执行结果。现在，这 3 个线程对象都有它们自己的 startDate 属性值。

线程局部变量分别为每个线程存储了各自的属性值，并提供给每个线程使用。你可以使用 get()方法读取这个值，并用 set()方法设置这个值。如果线程是第一次访问线程局部变量，线程局部变量可能还没有为它存储值，这个时候 initialValue()方法就会被调用，并且返回当前的时间值。



The screenshot shows the Eclipse IDE's Console view with the following text output:

```
<terminated> SafeMain [Java Application] E:\Java 7 Concurrency CookBook\desarrollo;
Starting Thread: 9 : Sat Aug 11 22:18:50 CEST 2012
Starting Thread: 10 : Sat Aug 11 22:18:52 CEST 2012
Thread Finished: 9 : Sat Aug 11 22:18:50 CEST 2012
Thread Finished: 8 : Sat Aug 11 22:18:48 CEST 2012
Thread Finished: 10 : Sat Aug 11 22:18:52 CEST 2012
```

更多信息

线程局部变量也提供了 `remove()`方法，用来为访问这个变量的线程删除已经存储的值。Java 并发 API 包含了 `InheritableThreadLocal` 类，如果一个线程是从其他某个线程中创建的，这个类将提供继承的值。如果一个线程 A 在线程局部变量已有值，当它创建其他某个线程 B 时，线程 B 的线程局部变量将跟线程 A 是一样的。你可以覆盖 `childValue()`方法，这个方法用来初始化子线程在线程局部变量中的值。它使用父线程在线程局部变量中的值作为传入参数。

1.11 线程的分组

方便管理

Java 并发 API 提供了一个有趣的功能，它能够把线程分组。这允许我们把一个组的线程当成一个单一的单元，对组内线程对象进行访问并操作它们。例如，对于一些执行同样任务的线程，你想控制它们，不管多少线程在运行，只需要一个单一的调用，所有这些线程的运行都会被中断。

Java 提供 `ThreadGroup` 类表示一组线程。线程组可以包含线程对象，也可以包含其他的线程组对象，它是一个树形结构。

在本节中，我们学习并使用 `ThreadGroup` 对象类开发一个简单的范例：创建 10 个线程并让它们休眠一个随机时间（例如模拟一个查询），当其中一个线程查找成功的时候，我们将中断其他的 9 个线程。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如

NetBeans), 都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 Result 的类。它存储先执行完的线程。声明一个私有字符串变量 name，并生成读写这个值的方法。
2. 创建一个名为 SearchTask 的类，它实现了 Runnable 接口。

```
public class SearchTask implements Runnable {
```

3. 声明一个 Result 类的私有属性，并实现带参数的构造器（Constructor），来为这个属性设置值。

```
private Result result;
public SearchTask(Result result) {
    this.result=result;
}
```

4. 实现 run() 方法。它将调用 doTask() 方法，并等待它完成或者抛出一个 InterruptedException 异常。run()方法也将打印出线程的开始、结束或者中断等信息。

```
@Override
public void run() {
    String name=Thread.currentThread().getName();
    System.out.printf("Thread %s: Start\n",name);
    try {
        doTask();
        result.setName(name);
    } catch (InterruptedException e) {
        System.out.printf("Thread %s: Interrupted\n",name);
        return;
    }
    System.out.printf("Thread %s: End\n",name);
}
```

5. 实现 doTask()方法。它创建 Random 对象来生成一个随机数，并用它做为传入参数调用 sleep()方法。

```
private void doTask() throws InterruptedException {
```

```
Random random=new Random((new Date()).getTime());
int value=(int)(random.nextDouble()*100);
System.out.printf("Thread %s: %d\n", Thread.currentThread().
getName(), value);
TimeUnit.SECONDS.sleep(value);
}
```

6. 创建一个包含 main()方法的主类 Main。

```
public class Main {
    public static void main(String[] args) {
```

7. 创建一个标识为 Searcher 的线程组对象。

```
ThreadGroup threadGroup = new ThreadGroup("Searcher");
```

8. 创建一个 Result 对象，并用它作为传入参数创建一个 SearchTask 对象。

```
Result result=new Result();
SearchTask searchTask=new SearchTask(result);
```

9. 使用创建的 SearchTask 对象作为传入参数创建 10 个线程对象。当调用线程的构造器时，第一个参数是 ThreadGroup 对象，第二个参数是 SearchTask 对象。

```
for (int i=0; i<5; i++) {
    Thread thread=new Thread(threadGroup, searchTask);
    thread.start();
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

10. 通过 list()方法打印线程组对象的信息。

```
System.out.printf("Number of Threads: %d\n", threadGroup.
activeCount());
System.out.printf("Information about the Thread Group\n");
threadGroup.list();
```

11. 通过 activeCount()方法获取线程组包含的线程数目，通过 enumerate()方法获取线

程组包含的线程列表。这两个方法可以帮助我们获取每个线程的信息，如线程的状态。

```
Thread[] threads=new Thread[threadGroup.activeCount()];
threadGroup.enumerate(threads);
for (int i=0; i<threadGroup.activeCount(); i++) {
    System.out.printf("Thread %s: %s\n",threads[i].
    getName(),threads[i].getState());
}
```

12. 调用 `waitFinish()`方法，我们将在下面实现这个方法。它将等到线程组的第一个线程运行结束。

```
waitFinish(threadGroup);
```

13. 使用 `interrupt()`方法中断这个组中的其余线程。

```
threadGroup.interrupt();
```

14. 实现 `waitFinish()`方法。`activeCount()`方法被用来检测是否有线程运行结束。

```
private static void waitFinish(ThreadGroup threadGroup) {
    while (threadGroup.activeCount()>0) {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

15. 运行范例并查看运行结果。

工作原理

在下面的截屏中，你会看到 `list()`方法的输出及每个线程对象的状态。

线程组类存储了线程对象和关联的线程组对象，并可以访问它们的信息（例如状态），将执行的操作应用到所有成员上（例如中断）。

```

<terminated> Main (8) [Java Application] E:\Java 7 Concurrency CookBook\des.

Information about the Thread Group
java.lang.ThreadGroup[name=Searcher,maxpri=10]
    Thread[Thread-0,5,Searcher]
    Thread[Thread-1,5,Searcher]
    Thread[Thread-2,5,Searcher]
    Thread[Thread-3,5,Searcher]
    Thread[Thread-4,5,Searcher]
    Thread Thread-0: TIMED_WAITING
    Thread Thread-1: TIMED_WAITING
    Thread Thread-2: TIMED_WAITING
    Thread Thread-3: TIMED_WAITING
    Thread Thread-4: TIMED_WAITING
    Thread Thread-2: Interrupted

```

更多信息

ThreadGroup 类有很多方法，它的 API 文档提供了所有方法的完整解释。

1.12 线程组中不可控异常的处理

提供应用程序中对错误情景的管理，是编程语言很重要的一面。和几乎所有的现代编程语言一样，Java 语言也实现了通过异常管理机制来处理错误情景，它提供了很多类来表示不同的错误。当错误情景发生时，Java 类将抛出这些异常。你可以使用这些异常，或者实现自己的异常，来管理类中的错误。

Java 也提供了捕获和处理这些异常的机制。有的异常必须被捕获，或者必须使用方法的 throws 声明再次抛出，这类异常叫做非运行时异常。还有一类异常叫做运行时异常，它们不需要被捕获或者声明抛出。

[RuntimeException](#)

在本章 1.5 节中，我们学习了如何在线程对象中处理非捕获异常。

另一种可行的做法是，建立一个方法来捕获线程组中的任何线程对象抛出的非捕获异常。

本节中，我们将通过范例学习这种异常处理的方法。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个 MyThreadGroup 类，并继承 ThreadGroup。必须声明带参数的构造器，因为 ThreadGroup 没有默认的不带参数的构造器。

```
public class MyThreadGroup extends ThreadGroup {
    public MyThreadGroup(String name) {
        super(name);
    }
}
```

非捕获异常

2. 覆盖 uncaughtException()方法。当线程组中的任何线程对象抛出异常的时候，这个方法就会被调用。在这里，这个方法将打印异常信息和抛出异常的线程代码到控制台，还将中断线程组中的其他线程。

```
@Override
public void uncaughtException(Thread t, Throwable e) {
    System.out.printf("The thread %s has thrown an Exception\n",t.
getId());
    e.printStackTrace(System.out);
    System.out.printf("Terminating the rest of the Threads\n");
    interrupt();
}
```

3. 创建一个 Task 类，它实现了 Runnable 接口。

```
public class Task implements Runnable {
```

4. 实现 run()方法。在这个方法里，我们要触发 AritmethicException 异常。为了达到目标，我们用 1000 除以一个随机数，当随机数生成器生成 0，异常将被抛出。

```
@Override
public void run() {
    int result;
    Random random=new Random(Thread.currentThread().getId());
```

```

        while (true) {
            result=1000/((int)(random.nextDouble()*1000));
            System.out.printf("%s : %d\n",Thread.currentThread().
                getId(),result);
            if (Thread.currentThread().isInterrupted()) {
                System.out.printf("%d : Interrupted\n",Thread.
                currentThread().getId());
                return;
            }
        }
    }
}

```

5. 实现这个范例的主程序，并且实现 main()方法。

```

public class Main {
    public static void main(String[] args) {

```

6. 创建一个 MyThreadGroup 线程组类对象。

```
MyThreadGroup threadGroup=new MyThreadGroup("MyThreadGroup");
```

7. 创建一个 Task 类对象。

```
Task task=new Task();
```

8. 用刚创建的两个对象作为传入参数，创建两个线程对象。

```

for (int i=0; i<2; i++) {
    Thread t=new Thread(threadGroup,task);
    t.start();
}

```

9. 运行范例并查看运行结果。

工作原理

当运行范例的时候，你会看到当一个线程对象抛出了异常，其余的线程对象都被中断。

当线程抛出非捕获异常时，JVM 将为这个异常寻找 3 种可能的处理器。

首先，寻找抛出这个异常的线程的非捕获异常处理器，参见 1.9 节。如果这个处理器不存在，JVM 继续查找这个线程所在的线程组的非捕获异常处理器，这也是本节学习的知

1. 抛出这个异常的线程的UncaughtExceptionHandler
2. 寻找默认的UncaughtExceptionHandler
3. Output to System.out and exit.

识。如果也不存在，JVM 将寻找默认的非捕获异常处理器，参见 1.9 节。

如果这些处理器都不存在，JVM 将把堆栈中异常信息打印到控制台，并且退出这个程序。

参见

- ◆ 参见本书 1.12 节。

1.13 使用工厂类创建线程

工厂模式是面向对象编程中最常使用的模式之一。它是一个创建者模式，使用一个类为其他的一个或者多个类创建对象。当我们要为这些类创建对象时，不需再使用 new 构造器，而使用工厂类。

使用工厂类，可以将对象的创建集中化，这样做有以下的好处：

- ◆ 更容易修改类，或者改变创建对象的方式；
- ◆ 更容易为有限资源限制创建对象的数目。例如，我们可以限制一个类型的对象不多于 n 个。
- ◆ 更容易为创建的对象生成统计数据。

Java 提供了 ThreadFactory 接口，这个接口实现了线程对象工厂。Java 并发 API 的高级工具类也使用了线程工厂创建线程。

在本节中，我们将学习如何通过实现 ThreadFactory 接口来创建线程对象，用以生成个性化名称的线程并且保存这些线程对象的统计信息。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 MyThreadFactory 的类，并且实现 ThreadFactory 接口。

```
public class MyThreadFactory implements ThreadFactory {
```

2. 声明3个属性：整型数字counter，用来存储线程对象的数量；字符串name，用来存放每个线程的名称；字符串列表stats，用来存放线程对象的统计数据。同时实现带参数的构造器来初始化这3个属性。

```
private int counter;
private String name;
private List<String> stats;
public MyThreadFactory(String name) {
    counter=0;
    this.name=name;
    stats=new ArrayList<String>();
}
```

3. 实现newThread()方法。这个方法以Runnable接口对象为参数，并且返回参数对应的线程对象。这里我们创建一个线程对象并生成线程名称，保存统计数据。

```
@Override
public Thread newThread(Runnable r) {
    Thread t=new Thread(r,name+"-Thread_"+counter);
    counter++;
    stats.add(String.format("Created thread %d with name %s on
%s\n",t.getId(),t.getName(),new Date()));
    return t;
}
```

4. 实现getStatistics()方法，返回一个字符串对象，用来表示所有线程对象的统计数据。

```
public String getStats(){
    StringBuffer buffer=new StringBuffer();
    Iterator<String> it=stats.iterator();
    while (it.hasNext()) {
        buffer.append(it.next());
        buffer.append("\n");
    }
    return buffer.toString();
}
```

5. 创建名为Task的类并且实现Runnable接口。在这个范例中，线程除了只休眠1秒

钟之外，不做任何事情。

```
public class Task implements Runnable {
    @Override
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

6. 创建范例的主程序。创建一个包含 main()方法的主类 Main。

```
public class Main {
    public static void main(String[] args) {
```

7. 创建一个 MyThreadFactory 对象，创建一个 Task 对象。

```
MyThreadFactory factory=new MyThreadFactory("MyThreadFactory");
Task task=new Task();
```

8. 使用工厂类 MyThreadFactory 创建 10 个线程对象，并且启动它们。

```
Thread thread;
System.out.printf("Starting the Threads\n");
for (int i=0; i<10; i++){
    thread=factory.newThread(task);
    thread.start();
}
```

9. 将线程工厂的统计打印到控制台。

```
System.out.printf("Factory stats:\n");
System.out.printf("%s\n", factory.getStats());
```

10. 运行范例并且查看运行结果。

工作原理

ThreadFactory 接口只有一个方法，即 newThread，它以 Runnable 接口对象作为传入

参数并且返回一个线程对象。当实现 ThreadFactory 接口时，必须实现覆盖这个方法。大多数基本的线程工厂类只有一行，即：

```
return new Thread(r);
```

可以通过增加一些变化来强化实现方法覆盖。

- ◆ 创建一个个性化线程，如本范例使用一个特殊的格式作为线程名，或者通过继承 Thread 类来创建自己的线程类；
- ◆ 保存新创建的线程的统计数据，如本节的范例那样；
- ◆ 限制创建的线程的数量；
- ◆ 对生成的线程进行验证；
- ◆ 更多你可以想到的。

使用工厂设计模式是一个很好的编程实践，但是，如果是通过实现 ThreadFactory 接口来创建线程，你必须检查代码，以保证所有的线程都是使用这个工厂创建的。

参见

- ◆ 参见 7.4 节。
- ◆ 参见 7.5 节。

第 2 章

线程同步基础

本章内容包含：

- ◆ 使用 synchronized 实现同步方法
- ◆ 使用非依赖属性实现同步
- ◆ 在同步代码块中使用条件
- ◆ 使用锁实现同步
- ◆ 使用读写锁同步数据访问
- ◆ 修改锁的公平性
- ◆ 在锁中使用多条件

2.1 简介

多个执行线程共享一个资源的情景，是最常见的并发编程情景之一。在并发应用中常常遇到这样的情景：多个线程读或者写相同的数据，或者访问相同的文件或数据库连接。为了防止这些共享资源可能出现的错误或数据不一致，我们必须实现一些机制来防止这些错误的发生。

临界区，构成一个广义的原子

为了解决这些问题，人们引入了临界区（CriticalSection）概念。临界区是一个用以访问共享资源的代码块，这个代码块在同一时间内只允许一个线程执行。

为了帮助编程人员实现这个临界区，Java（以及大多数编程语言）提供了同步机制。当一个线程试图访问一个临界区时，它将使用一种同步机制来查看是不是已经有其他线程

进入临界区。如果没有其他线程进入临界区，它就可以进入临界区；如果已经有线程进入了临界区，它就被同步机制挂起，直到进入的线程离开这个临界区。如果在等待进入临界区的线程不止一个，JVM 会选择其中的一个，其余的将继续等待。[阻塞队列](#)

本章将逐步讲解如何使用 Java 语言提供的两种基本同步机制：

- ◆ `synchronized` 关键字机制；
- ◆ `Lock` 接口及其实现机制。

2.2 使用 `synchronized` 实现同步方法

在本节中，我们将学习如何使用 Java 的最基本的同步方式，即使用 **synchronized** 关键字来控制一个方法的并发访问。如果一个对象已用 **synchronized** 关键字声明，那么只有一个执行线程被允许访问它。如果其他某个线程试图访问这个对象的其他方法，它将被挂起，直到第一个线程执行完正在运行的方法。

换句话说，每一个用 **synchronized** 关键字声明的方法都是临界区。在 Java 中，同一个对象的临界区，在同一时间只有一个允许被访问。

静态方法则有不同的行为。用 **synchronized** 关键字声明的静态方法，同时只能被一个执行线程访问，但是其他线程可以访问这个对象的非静态方法。必须非常谨慎这一点，因为两个线程可以同时访问一个对象的两个不同的 **synchronized** 方法，即其中一个是静态方法，另一个是非静态方法。如果两个方法都改变了相同的数据，将会出现数据不一致的错误。

我们将通过范例来学习这个概念。范例将使两个线程访问同一个对象。我们有一个银行账号和两个线程，一个线程将转钱到账户中，另一线程将从账户中取钱。如果方法不同步，账户钱数可能不正确。而同步机制则能确保账户的最终余额是正确的。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

- 创建名为 **Account** 的账号类，它是银行账户的模型，只有一个双精度浮点型属性 **balance**。

```
public class Account {
    private double balance;
```

- 实现 **setBalance()** 和 **getBalance()** 方法来写入和读取余额 **balance** 的值。

```
public double getBalance() {
    return balance;
}
public void setBalance(double balance) {
    this.balance = balance;
}
```

- 实现 **addAmount()** 方法。它会将传入的数量加入到余额 **balance** 中，并且在同一时间只允许一个线程改变这个值，所以我们使用 **synchronized** 关键字将这个方法标记成临界区。

```
public synchronized void addAmount(double amount) {
    double tmp=balance;
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    tmp+=amount;
    balance=tmp;
}
```

- 实现 **subtractAmount()** 方法。它将传入的数量从余额中扣除，并且在同一时间只允许一个线程改变这个值，所以我们使用 **synchronized** 关键字将这个方法标记成临界区。

```
public synchronized void subtractAmount(double amount) {
    double tmp=balance;
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    tmp-=amount;
    balance=tmp;
}
```

5. 实现一个 ATM 模拟类 **Bank**。它使用 **subtractAmount()**方法对账户余额进行扣除。这个类实现 **Runnable** 接口以作为线程执行。

```
public class Bank implements Runnable {
```

6. 为这个类增加账户类 **Account** 对象，用构造器初始化这个对象。

```
private Account account;
public Bank(Account account) {
    this.account=account;
}
```

7. 实现 **run()**方法。它将调用 **subtractAmount()**方法对账户余额进行扣除，并循环执行 100 次。

```
@Override
public void run() {
    for (int i=0; i<100; i++) {
        account.subtractAmount(1000);
    }
}
```

8. 实现公司模拟类 **Company**。它使用 **addAmount()**对账户的余额进行充值。这个类实现 **Runnable** 接口以作为线程运行。

```
public class Company implements Runnable {
```

9. 为 **Company** 类增加账户类 **Account** 对象，用构造器初始化这个对象。

```
private Account account;
public Company(Account account) {
    this.account=account;
}
```

10. 实现 **run()**方法。它将调用 **addAmount()**方法对账户余额进行充值，并循环执行 100 次。

```
@Override
public void run() {
    for (int i=0; i<100; i++) {
        account.addAmount(1000);
    }
}
```

```

    }
}
}
```

11. 实现范例的主程序，创建一个带有 **main()** 方法的主类 **Main**。

```
public class Main {
    public static void main(String[] args) {
```

12. 创建一个账户类 **Account** 对象并设置初始值为 1000。

```
Account account=new Account();
account.setBalance(1000);
```

13. 创建一个公司类 **Company** 对象，并用它作为传入参数创建线程。

```
Company company=new Company(account);
Thread companyThread=new Thread(company);
```

14. 创建一个**ATM**模拟类**Bank**对象，并用它作为传入参数创建线程。

```
Bank bank=new Bank(account);
Thread bankThread=new Thread(bank);
```

15. 将初始余额打印到控制台，并启动这两个线程。

```
System.out.printf("Account : Initial Balance: %f\n",account.
getBalance());
companyThread.start();
bankThread.start();
```

16. 使用 **join()**方法等待这两个线程运行完成，然后打印账户的最终余额到控制台。

```
try {
    companyThread.join();
    bankThread.join();
    System.out.printf("Account : Final Balance: %f\n",account.
getBalance());
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

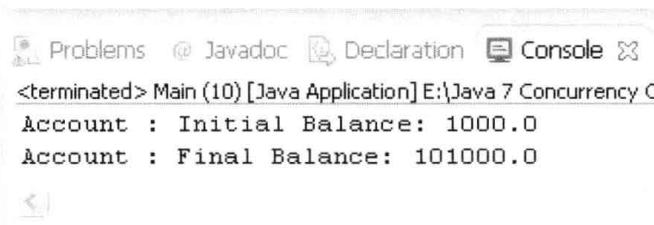
工作原理

在本节中，你已经开发了一个银行账户的模拟应用，它能够对余额进行充值和扣除。这个程序通过调用 100 次 **addAmount()** 方法对账户进行充值，每次存入 1000；然后通过调用 100 次 **subtractAmount()** 方法对账户余额进行扣除，每次扣除 1000；我们期望账户的最终余额与起初余额相等。

范例中通过使用临时变量 **tmp** 来存储账户余额，已经制造了一个错误情景：这个临时变量先获取账户余额，然后进行数额累加，之后把最终结果更新为账户余额。此外，范例还通过 **sleep()** 方法增加了延时，使得正在执行这个方法的线程休眠 10ms，而此时其他某个线程也可能会执行这个方法，因此可能会改变账户余额，引发错误。而 **synchronized** 关键字机制避免了这类错误的发生。

如果想查看共享数据的并发访问问题，只需要将 **addAmount()** 和 **subtractAmount()** 方法声明中的 **synchronized** 关键字删除即可。在没有 **synchronized** 关键字的情况下，一个线程读取了账户余额然后进入休眠，这个时候，其他某个线程读取这个账户余额，最终这两个方法都将修改同一个余额，但是其中一个操作不会影响到最终结果。

可以从下面的截屏中看到余额值并不一致。



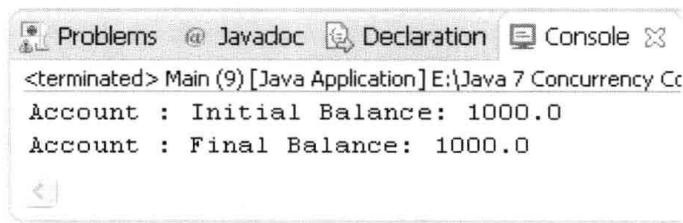
```

Problems @ Javadoc Declaration Console 
<terminated> Main (10) [Java Application] E:\Java 7 Concurrency C
Account : Initial Balance: 1000.0
Account : Final Balance: 101000.0

```

如果多次运行这个程序，你将获取不同的结果。因为 JVM 并不保证线程的执行顺序，所以每次运行的时候，线程将以不同的顺序读取并且修改账户的余额，造成最终结果也是不同的。

现在，将 **synchronized** 关键字恢复到两个方法的声明中，然后运行程序。从下面的截屏中你可以发现，运行结果跟期望是一致的。即便多次运行这个程序，仍将获取相同的结果。



```

Problems @ Javadoc Declaration Console 
<terminated> Main (9) [Java Application] E:\Java 7 Concurrency C
Account : Initial Balance: 1000.0
Account : Final Balance: 1000.0

```

synchronized 关键字的使用，保证了在并发程序中对共享数据的正确访问。

一个类，一个锁

在本节的简介中提到，一个对象的方法采用 **synchronized** 关键字进行声明，只能被一个线程访问。如果线程 A 正在执行一个同步方法 syncMethodA()，线程 B 要执行这个对象的其他同步方法 syncMethodB()，线程 B 将被阻塞直到线程 A 访问完。但如果线程 B 访问的是同一个类的不同对象，那么两个线程都不会被阻塞。

更多信息

synchronized 关键字会降低应用程序的性能，因此只能在并发情景中需要修改共享数据的方法上使用它。如果多个线程访问同一个 **synchronized** 方法，则只有一个线程可以访问，其他线程将等待。如果方法声明没有使用 **synchronized** 关键字，所有的线程都能在同一时间执行这个方法，因而总运行时间将降低。如果已知一个方法不会被一个以上线程调用，则无需使用 **synchronized** 关键字声明之。

可以递归调用被 **synchronized** 声明的方法。当线程访问一个对象的同步方法时，它还可以调用这个对象的其他的同步方法，也包含正在执行的方法，而不必再次去获取这个方法的访问权。

我们可以通过 **synchronized** 关键字来保护代码块（而不是整个方法）的访问。应该这样利用 **synchronized** 关键字：方法的其余部分保持在 **synchronized** 代码块之外，以获取更好的性能。临界区（即同一时间只能被一个线程访问的代码块）的访问应该尽可能的短。例如在获取一幢楼人数的操作中，我们只使用 **synchronized** 关键字来保护对人数更新的指令，并让其他操作不使用共享数据。当这样使用 **synchronized** 关键字时，必须把对象引用作为传入参数。同一时间只有一个线程被允许访问这个 **synchronized** 代码。通常来说，我们使用 **this** 关键字来引用正在执行的方法所属的对象。

```
synchronized (this) {
    // Java code
}
```

2.3 使用非依赖属性实现同步

当使用 **synchronized** 关键字来保护代码块时，必须把对象引用作为传入参数。通常情况下，使用 **this** 关键字来引用执行方法所属的对象，也可以使用其他的对象对其进行引用。一般来说，这些对象就是为这个目的而创建的。例如，在类中有两个非依赖属性，它们被多个线程共享，你必须同步每一个变量的访问，但是同一时刻只允许一个线程访问一个属

性变量，其他某个线程访问另一个属性变量。

在本节中，我们将通过范例学习实现电影院售票场景的编程。这个范例模拟了有两个屏幕和两个售票处的电影院。一个售票处卖出的一张票，只能用于其中一个电影院，不能同时用于两个电影院，因此每个电影院的剩余票数是独立的属性。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个电影院类 **Cinema**，增加两个 **long** 属性 **vacanciesCinema1** 和 **vacanciesCinema2**。

```
public class Cinema {
    private long vacanciesCinema1;
    private long vacanciesCinema2;
```

2. 再为电影院类 **Cinema** 增加两个对象属性 **controlCinema1** 和 **controlCinema2**。

```
private final Object controlCinema1, controlCinema2;
```

3. 实现电影院类 **Cinema** 的构造器，并初始化这4个属性。

```
public Cinema() {
    controlCinema1=new Object();
    controlCinema2=new Object();
    vacanciesCinema1=20;
    vacanciesCinema2=20;
}
```

4. 实现 **sellTickets1()** 方法，当第一个电影院有票卖出的时候将调用这个方法。它使用 **controlCinema1** 对象控制同步代码块的访问。

```
public boolean sellTickets1 (int number) {
    synchronized (controlCinema1) {
        if (number<vacanciesCinema1) {
            vacanciesCinema1-=number;
```

```
        return true;
    } else {
        return false;
    }
}
```

5. 实现 **sellTickets2()**方法，当第二个电影院有票卖出的时候将调用这个方法。它使用 **controlCinema2** 对象控制同步代码块的访问。

```
public boolean sellTickets2 (int number) {  
    synchronized (controlCinema2) {  
        if (number < vacanciesCinema2) {  
            vacanciesCinema2 -= number;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

6. 实现 **returnTickets10** 方法，当第一个电影院有票退回的时候将调用这个方法。它使用 **controlCinema1** 对象控制同步代码块的访问。

```
public boolean returnTickets1 (int number) {  
    synchronized (controlCinematic) {  
        vacanciesCinematic+=number;  
        return true;  
    }  
}
```

7. 实现 **returnTickets2()**方法，当第二个电影院有票退回的时候将调用这个方法。它使用 **controlCinema2** 对象控制同步代码块的访问。

```
public boolean returnTickets2 (int number) {  
    synchronized (controlCinema2) {  
        vacanciesCinema2+=number;  
        return true;  
    }  
}
```

8. 实现另外两个方法，它们返回各自电影院的票数。

```
public long getVacanciesCinema1() {  
    return vacanciesCinema1;  
}  
public long getVacanciesCinema2() {  
    return vacanciesCinema2;  
}
```

9. 实现售票处类 **TicketOffice1** 它实现了 **Runnable** 接口。

```
public class TicketOffice1 implements Runnable {
```

10. 声明电影院 **Cinema** 对象，并且通过构造器对其进行初始化。

```
private Cinema cinema;  
public TicketOffice1 (Cinema cinema) {  
    this.cinema=cinema;  
}
```

11. 实现 **run()**方法，它模拟了对两个电影院的操作。

```
@Override  
public void run() {  
    cinema.sellTickets1(3);  
    cinema.sellTickets1(2);  
    cinema.sellTickets2(2);  
    cinema.returnTickets1(3);  
    cinema.sellTickets1(5);  
    cinema.sellTickets2(2);  
    cinema.sellTickets2(2);  
    cinema.sellTickets2(2);  
}
```

12. 实现售票处类 **TicketOffice2**，它实现了 **Runnable** 接口。

```
public class TicketOffice2 implements Runnable {
```

13. 声明电影院 **Cinema** 对象，并且通过构造器对其进行初始化。

```
private Cinema cinema;  
public TicketOffice2(Cinema cinema) {  
    this.cinema=cinema;  
}
```

14. 实现 **run()** 方法，它模拟了对两个电影院的操作。

```
@Override
public void run() {
    cinema.sellTickets2(2);
    cinema.sellTickets2(4);
    cinema.sellTickets1(2);
    cinema.sellTickets1(1);
    cinema.returnTickets2(2);
    cinema.sellTickets1(3);
    cinema.sellTickets2(2);
    cinema.sellTickets1(2);
}
```

15. 实现范例的主程序，创建一个带有 **main()** 方法的主类 **Main**。

```
public class Main {
    public static void main(String[] args) {
```

16. 声明并创建电影院 **Cinema** 对象。

```
Cinema cinema=new Cinema();
```

17. 创建售票处类 **TicketOffice1** 对象，并用它作为传入参数创建线程。

```
TicketOffice1 ticketOffice1=new TicketOffice1(cinema);
Thread thread1=new Thread(ticketOffice1,"TicketOffice1");
```

18. 创建售票处类 **TicketOffice2** 对象，并用它作为传入参数创建线程。

```
TicketOffice2 ticketOffice2=new TicketOffice2(cinema);
Thread thread2=new Thread(ticketOffice2,"TicketOffice2");
```

19. 启动这两个线程。

```
thread1.start();
thread2.start();
```

20. 等待这两个线程运行结束。

```
try {
    thread1.join();
    thread2.join();
```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

21. 将两个电影院的余票信息打印到控制台。

```

System.out.printf("Room 1 Vacancies: %d\n", cinema.
    getVacanciesCinema1());
System.out.printf("Room 2 Vacancies: %d\n", cinema.
    getVacanciesCinema2());

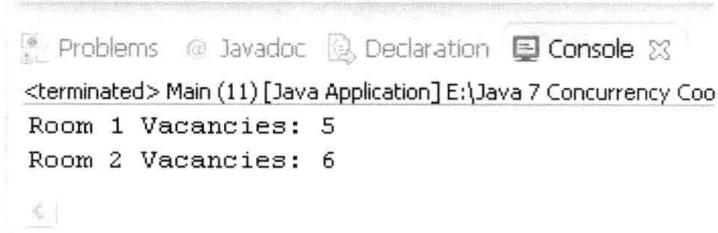
```

工作原理

用 **synchronized** 关键字保护代码块时，我们使用对象作为它的传入参数。JVM 保证同一时间只有一个线程能够访问这个对象的代码保护块（注意我们一直谈论的是对象，不是类）。相对 **synchronized** 方法，性能更好，**synchronized** 方法相当于使用 **this** 作为锁。

备注：这个例子使用了一个对象来控制对 **vacanciesCinema1** 属性的访问，所以同一时间只有一个线程能够修改这个属性；使用了另一个对象来控制 **vacanciesCinema2** 属性的访问，所以同一时间只有一个线程能够修改这个属性。但是，这个例子允许同时运行两个线程：一个修改 **vacancesCinema1** 属性，另一个修改 **vacancesCinema2** 属性。

运行这个范例，可以看到最终结果总是与每个电影院的剩余票数一致。在下面的截屏中，你可以看到这个应用的运行结果。



更多信息

synchronized 关键字还有其他的重要作用，参见其他章节对这个关键字的解释。

参见

参见 2.4 节。

2.4 在同步代码中使用条件

在并发编程中一个典型的问题是生产者-消费者（Producer-Consumer）问题。我们有一个数据缓冲区，一个或者多个数据生产者将把数据存入这个缓冲区，一个或者多个数据消费者将数据从缓冲区中取走。

这个缓冲区是一个共享数据结构，必须使用同步机制控制对它的访问，例如使用 **synchronized** 关键字，但是会受到更多的限制。如果缓冲区是满的，生产者就不能再放入数据，如果缓冲区是空的，消费者就不能读取数据。

对于这些场景，Java 在 **Object** 类中提供了 **wait()**、**notify()** 和 **notifyAll()** 方法。线程可以在同步代码块中调用 **wait()** 方法。如果在同步代码块之外调用 **wait()** 方法，JVM 将抛出 **IllegalMonitorStateException** 异常。当一个线程调用 **wait()** 方法时，JVM 将这个线程置入休眠，并且释放控制这个同步代码块的对象，同时允许其他线程执行这个对象控制的其他同步代码块。为了唤醒这个线程，必须在这个对象控制的某个同步代码块中调用 **notify()** 或者 **notifyAll()** 方法。

在本节中，我们将通过范例学习实现生产者-消费者问题，这个范例中将使用 **synchronized** 关键字和 **wait()**、**notify()** 及 **notifyAll()** 方法。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建数据存储类 **EventStorage**，它有 2 个属性：一个是 **int** 型属性 **maxSize**，另一个是 **LinkedList<Date>** 型属性 **storage**。

```
public class EventStorage {
    private int maxSize;
    private List<Date> storage;
```

2. 实现构造器，并初始化这两个属性。

```
public EventStorage() {
    maxSize=10;
    storage=new LinkedList<>();
}
```

3. 实现同步方法 **set()**，它保存数据到存储列表 **storage** 中。首先，它将检查列表是不是满的，如果已满，就调用 **wait()** 方法挂起线程并等待空余空间的出现。在这个方法的最后，我们调用 **notifyAll()** 方法唤醒所有因调用 **wait()** 方法进入休眠的线程。

```
public synchronized void set() {
    while (storage.size()==maxSize) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    storage.add(new Date());
    System.out.printf("Set: %d",storage.size());
    notifyAll();
}
```

4. 实现同步方法 **get()**，它从存储列表 **storage** 中获取数据。首先，它将检查列表中是不是有数据，如果没有，就调用 **wait()** 方法挂起线程并等待列表中数据的出现。在这个方法的最后，我们调用 **notifyAll()** 方法唤醒所有因调用 **wait()** 方法进入休眠的线程。

```
public synchronized void get() {
    while (storage.size()==0) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.printf("Get: %d: %s",storage.
        size(),((LinkedList<?>)storage).poll());
    notifyAll();
}
```

5. 创建生产者类 **Producer** 并且实现 **Runnable** 接口，用以表示范例中的生产者。

```
public class Producer implements Runnable {
```

6. 声明数据存储 **EventStorage** 对象，并且通过构造器对其进行初始化。

```
private EventStorage storage;
public Producer(EventStorage storage) {
    this.storage=storage;
}
```

7. 实现 **run()**方法，用来循环调用 100 次 **set()**方法。

```
@Override
public void run() {
    for (int i=0; i<100; i++) {
        storage.set();
    }
}
```

8. 创建消费者类 **Consumer** 并且实现 **Runnable** 接口，用以表示范例中的消费者。

```
public class Consumer implements Runnable {
```

9. 声明数据存储 **EventStorage** 对象，并且通过构造器对其进行初始化。

```
private EventStorage storage;
public Consumer(EventStorage storage) {
    this.storage=storage;
}
```

10. 实现 **run()**方法，用以循环调用 100 次 **get()**方法。

```
@Override
public void run() {
    for (int i=0; i<100; i++) {
        storage.get();
    }
}
```

11. 实现范例的主程序，创建一个带有 **main()**方法的主类 **Main**。

```
public class Main {
    public static void main(String[] args) {
```

12. 创建一个数据存储 **EventStorage** 对象。

```
EventStorage storage=new EventStorage();
```

13. 创建一个生产者 **Producer** 对象，并用它作为传入参数创建一个线程。

```
Producer producer=new Producer(storage);
Thread thread1=new Thread(producer);
```

14. 创建一个消费者 **Consumer** 对象，并用它作为传入参数创建一个线程。

```
Consumer consumer=new Consumer(storage);
Thread thread2=new Thread(consumer);
```

15. 启动这两个线程。

```
thread2.start();
thread1.start();
```

工作原理

这个范例的主要部分是数据存储 **EventStorage** 类的 **set()** 和 **get()** 方法。首先，**set()** 方法检查存储列表 **storage** 是否还有空间，如果已经满了，就调用 **wait()** 方法挂起线程并等待空余空间出现。其次，当其他线程调用 **notifyAll()** 方法时，挂起的线程将被唤醒并且再次检查这个条件。**notifyAll()** 并不保证哪个线程会被唤醒。这个过程持续进行直到存储列表有空余空间出现，然后生产者将生成一个新的数据并且存入存储列表 **storage**。

get() 方法的行为与之类似。首先，**get()** 方法检查存储列表 **storage** 是否还有数据，如果没有，就调用 **wait()** 方法挂起线程并等待数据的出现。其次，当其他线程调用 **notifyAll()** 方法时，挂起的线程将被唤醒并且再次检查这个条件。这个过程持续进行直到存储列表有数据出现。

备注：必须在 **while** 循环中调用 **wait()**，并且不断查询 **while** 的条件，直到条件为真的时候才能继续。

运行这个范例，将看到生产者和消费者是怎么样存入和取出数据的，这个存储列表最多有 10 个数据。

更多信息

`synchronized` 关键字还有其他的重要作用，参见其他章节对这个关键字的解释。

参见

- ◆ 参见 2.3 节。

2.5 使用锁实现同步

Java 提供了同步代码块的另一种机制，它是一种比 `synchronized` 关键字更强大也更灵活的机制。这种机制基于 **Lock** 接口及其实现类（例如 `ReentrantLock`），提供了更多的好处。

◆ 支持更灵活的同步代码块结构。使用 `synchronized` 关键字时，只能在同一个 `synchronized` 块结构中获取和释放控制。**Lock** 接口允许实现更复杂的临界区结构（译者注：即控制的获取和释放不出现在同一个块结构中）。

◆ 相比 `synchronized` 关键字， **Lock** 接口提供了更多的功能。其中一个新功能是 `tryLock()` 方法的实现。这个方法试图获取锁，如果锁已被其他线程获取，它将返回 `false` 并继续往下执行代码。使用 `synchronized` 关键字时，如果线程 A 试图执行一个同步代码块，而线程 B 已在执行这个同步代码块，则线程 A 就会被挂起直到线程 B 运行完这个同步代码块。使用锁的 `tryLock()` 方法，通过返回值将得知是否有其他线程正在使用这个锁保护的代码块。

- ◆ **Lock** 接口允许分离读和写操作，允许多个读线程和只有一个写线程。
- ◆ 相比 `synchronized` 关键字， **Lock** 接口具有更好的性能。

在本节中，我们将学习如何使用锁来同步代码，并且使用 **Lock** 接口和它的实现类——**ReentrantLock** 类来创建一个临界区。这个范例将模拟打印队列。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个打印队列类 **PrintQueue**。

```
public class PrintQueue {
```

2. 声明一个锁对象，并且用 **ReentrantLock** 类初始化。

```
private final Lock queueLock=new ReentrantLock();
```

3. 实现打印方法 **printJob()**，它的参数是对象型，并且不返回值。

```
public void printJob(Object document) {
```

4. 在打印方法 **printJob()** 内部，通过调用 **lock()** 方法获取对锁对象的控制。

```
queueLock.lock();
```

5. 使用以下的代码模拟文档的打印。

```
try {
    Long duration=(long)(Math.random()*10000);
    System.out.println(Thread.currentThread().getName()+" :
        PrintQueue: Printing a Job during "+(duration/1000)+"
        seconds");
    Thread.sleep(duration);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

6. 通过 **unlock()** 释放对锁对象的控制。

```
finally {
    queueLock.unlock();
}
```

7. 创建打印工作类 **Job** 并且实现 **Runnable** 接口。

```
public class Job implements Runnable {
```

8. 声明打印队列 **PrintQueue** 对象，并且通过构造器对其进行初始化。

```
private PrintQueue printQueue;
public Job(PrintQueue printQueue) {
    this.printQueue=printQueue;
}
```

9. 实现 **run()** 方法。它使用打印队列 **PrintQueue** 对象发送一个打印工作。

```
@Override
public void run() {
    System.out.printf("%s: Going to print a document\n", Thread.
        currentThread().getName());
    printQueue.printJob(new Object());
    System.out.printf("%s: The document has been printed\n",
        Thread.currentThread().getName());
}
```

10. 创建范例的主类 **Main**，它包含 **main()** 方法。

```
public class Main {
    public static void main (String args[]){
```

11. 创建一个共享的打印队列对象。

```
PrintQueue printQueue=new PrintQueue();
```

12. 创建 10 个打印工作 **Job** 对象，并把它作为传入参数创建线程。

```
Thread thread[]=new Thread[10];
for (int i=0; i<10; i++){
    thread[i]=new Thread(new Job(printQueue), "Thread "+ i);
}
```

13. 启动这 10 个线程。

```
for (int i=0; i<10; i++){
    thread[i].start();
}
```

工作原理

通过下面的截屏，你会看到部分运行结果。

这个范例的主要部分是打印队列类 **PrintQueue** 中的 **printJob()** 方法。我们使用锁实现一个临界区，并且保证同一时间只有一个执行线程访问这个临界区时，必须创建 **ReentrantLock** 对象。在这个临界区的开始，必须通过 **lock()** 方法获取对锁的控制。当线程 A 访问这个方法时，如果没有其他线程获取对这个锁的控制，**lock()** 方法将让线程 A 获

得锁并且允许它立刻执行临界区代码。否则，如果其他线程 B 正在执行这个锁保护的临界区代码，**lock()**方法将让线程 A 休眠直到线程 B 执行完临界区的代码。

```

Problems @ Javadoc Declaration Console <terminated> Main (12) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin
Thread 1: PrintQueue: Printing a Job during 2 seconds
Thread 1: The document has been printed
Thread 8: PrintQueue: Printing a Job during 9 seconds
Thread 8: The document has been printed
Thread 6: PrintQueue: Printing a Job during 5 seconds
Thread 6: The document has been printed
Thread 4: PrintQueue: Printing a Job during 1 seconds
Thread 4: The document has been printed
Thread 2: PrintQueue: Printing a Job during 0 seconds
Thread 2: The document has been printed

```

在线程离开临界区的时候，我们必须使用 **unlock()**方法来释放它持有的锁，以让其他线程来访问临界区。如果在离开临界区的时候没有调用 **unlock()**方法，其他线程将永久地等待，从而导致了死锁（**Deadlock**）情景。如果在临界区使用了 **try-catch** 块，不要忘记将 **unlock()**方法放入 **finally** 部分。

更多信息

Lock 接口（和它的实现类 **ReentrantLock**）还提供了另一个方法来获取锁，即 **tryLock()** 方法。跟 **lock()**方法最大的不同是：线程使用 **tryLock()**不能够获取锁，**tryLock()**会立即返回，它不会将线程置入休眠。**tryLock()**方法返回一个布尔值，**true** 表示线程获取了锁，**false** 表示没有获取锁。

备注：编程人员应该重视 **tryLock()**方法的返回值及其对应的行为。如果这个方法返回 **false**，则程序不会执行临界区代码。如果执行了，这个应用很可能会出现错误的结果。

ReentrantLock 类也允许使用递归调用。如果一个线程获取了锁并且进行了递归调用，它将继续持有这个锁，因此调用 **lock()**方法后也将立即返回，并且线程将继续执行递归调用。再者，我们还可以调用其他的方法。

必须很小心使用锁，以避免死锁。当两个或者多个线程被阻塞并且它们在等待的锁永远不会被释放时，就会发生死锁。例如，线程 A 获取了锁 X，线程 B 获取了锁 Y，现在，

线程 A 试图获取锁 Y，同时线程 B 也试图获取锁 X，则两个线程都将被阻塞，而且它们等待的锁永远会被释放。这个问题就在于两个线程都试图获取对方拥有的锁。附录中的“并发编程设计”，提供了并发应用设计的很好建议，它们能够避免死锁的发生。

参见

- ◆ 参见 2.2 节。
- ◆ 参见 2.8。
- ◆ 参见 8.2 节。

2.6 使用读写锁实现同步数据访问

锁机制最大的改进之一就是 **ReadWriteLock** 接口和它的唯一实现类 **ReentrantReadWriteLock**。这个类有两个锁，一个是读操作锁，另一个是写操作锁。使用读操作锁时可以允许多个线程同时访问，但是使用写操作锁时只允许一个线程进行。在一个线程执行写操作时，其他线程不能够执行读操作。

在本节中，我们将通过范例学习如何使用 **ReadWriteLock** 接口编写程序。这个范例将使用 **ReadWriteLock** 接口控制对价格对象的访问，价格对象存储了两个产品的价格。

准备工作

为了更好地理解本节内容，请阅读 2.5 节。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个价格信息类 **PricesInfo**，并且存放两个产品的价格。

```
public class PricesInfo {
```

2. 声明两个 **double** 属性 **price1** 和 **price2**。

```
private double price1;  
private double price2;
```

3. 声明读写锁 **ReadWriteLock** 对象 **lock**。

```
private ReadWriteLock lock;
```

4. 通过构造器初始化这 3 个属性。对于 **lock** 属性，我们为它创建 **ReentrantReadWriteLock** 实例。

```
public PricesInfo() {
    price1=1.0;
    price2=2.0;
    lock=new ReentrantReadWriteLock();
}
```

5. 实现 **getPrice1()** 方法来返回属性 **price1** 的值。它使用读锁来获取对这个属性的访问。

```
public double getPrice1() {
    lock.readLock().lock();
    double value=price1;
    lock.readLock().unlock();
    return value;
}
```

6. 实现 **getPrice2()** 方法来返回属性 **price2** 的值。它使用读锁来获取对这个属性的访问。

```
public double getPrice2() {
    lock.readLock().lock();
    double value=price2;
    lock.readLock().unlock();
    return value;
}
```

7. 实现 **setPrices()** 方法为这两个 **price** 属性赋值，它使用了写锁来控制对这两个属性的访问。

```
public void setPrices(double price1, double price2) {
    lock.writeLock().lock();
    this.price1=price1;
    this.price2=price2;
    lock.writeLock().unlock();
}
```

8. 创建读取类 **Reader**，它实现了 **Runnable** 接口。这个类将读取价格信息 **PricesInfo**

类的属性值。

```
public class Reader implements Runnable {
```

9. 声明价格信息 **PricesInfo** 对象，并且通过构造器为它赋值。

```
private PricesInfo pricesInfo;
public Reader (PricesInfo pricesInfo){
    this.pricesInfo=pricesInfo;
}
```

10. 实现 **run()** 方法，它将循环读取两个价格值 10 次。

```
@Override
public void run() {
    for (int i=0; i<10; i++){
        System.out.printf("%s: Price 1: %f\n", Thread.
currentThread().getName(),pricesInfo.getPrice1());
        System.out.printf("%s: Price 2: %f\n", Thread.
currentThread().getName(),pricesInfo.getPrice2());
    }
}
```

11. 创建写入类 **Writer**，它实现了 **Runnable** 接口。这个类将修改价格信息 **PricesInfo** 类的属性值。

```
public class Writer implements Runnable {
```

12. 声明价格信息 **PricesInfo** 对象，并且通过构造器为它赋值。

```
private PricesInfo pricesInfo;
public Writer(PricesInfo pricesInfo){
    this.pricesInfo=pricesInfo;
}
```

13. 实现 **run()** 方法，它将循环修改两个价格 3 次，每次修改后线程将休眠 2 秒钟。

```
@Override
public void run() {
    for (int i=0; i<3; i++) {
        System.out.printf("Writer: Attempt to modify the
prices.\n");
```

```
    pricesInfo.setPrices(Math.random()*10, Math.random()*8);
    System.out.printf("Writer: Prices have been modified.\n");
    try {
        Thread.sleep(2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

14. 创建范例的主类 **Main**, 它包含 **main()**方法。

```
public class Main {  
    public static void main(String[] args) {
```

15. 创建一个价格信息 **PricesInfo** 对象。

```
PricesInfo pricesInfo=new PricesInfo();
```

16. 创建 5 个读取类 **Reader** 对象，并把每一个对象作为传入参数创建线程。

```
Reader readers[] = new Reader[5];
Thread threadsReader[] = new Thread[5];
for (int i=0; i<5; i++) {
    readers[i] = new Reader(pricesInfo);
    threadsReader[i] = new Thread(readers[i]);
}
```

17. 创建一个写入类 **Writer** 对象，并把它作为传入参数创建线程。

```
Writer writer=new Writer(pricesInfo);
Thread threadWriter=new Thread(writer);
```

18. 启动这 6 个线程。

```
for (int i=0; i<5; i++){
    threadsReader[i].start();
}
threadWriter.start();
```

工作原理

通过下面的截屏，你会看到范例的部分运行结果。

```

Problems @ Javadoc Declaration Console X
<terminated> Main (13) [Java Application] E:\Java 7 Concurrency Co
Thread-1: Price 2: 1.798705
Thread-1: Price 1: 7.692004
Thread-1: Price 2: 1.798705
Thread-1: Price 1: 7.692004
Writer: Attempt to modify the prices.
Writer: Prices have been modified.
Thread-1: Price 2: 1.798705
Thread-1: Price 1: 9.901118
Thread-1: Price 2: 1.121504
Thread-1: Price 1: 9.901118
Thread-1: Price 2: 1.121504

```

像我们之前提到的，**ReentrantReadWriteLock** 类有两种锁：一种是读操作锁，另一种是写操作锁。读操作锁是通过 **ReadWriteLock** 接口的 **readLock()** 方法获取的，这个锁实现了 **Lock** 接口，所以我们可以使用 **lock()**、**unlock()** 和 **tryLock()** 方法。写操作锁是通过 **ReadWriteLock** 接口的 **writeLock()** 方法获取的，这个锁同样也实现了 **Lock** 接口，所以我们也可以使用 **lock()**、**unlock()** 和 **tryLock()** 方法。编程人员应该确保正确地使用这些锁，使用它们的时候应该符合这些锁的设计初衷。当你获取 **Lock** 接口的读锁时，不可以进行修改操作，否则将引起数据不一致的错误。

参见

- ◆ 参见 2.5 节。
- ◆ 参见 8.2 节。

2.7 修改锁的公平性

ReentrantLock 和 **ReentrantReadWriteLock** 类的构造器都含有一个布尔参数 **fair**，它允许你控制这两个类的行为。默认 **fair** 值是 **false**，它称为非公平模式（Non-Fair Mode）。在非公平模式下，当有很多线程在等待锁（**ReentrantLock** 和 **ReentrantReadWriteLock**）时，锁将选择它们中的一个来访问临界区，这个选择是没有任何约束的。如果 **fair** 值是 **true**，则称为公平模式（Fair Mode）。在公平模式下，当有很多线程在等待锁（**ReentrantLock** 和

ReentrantReadWriteLock)时，锁将选择它们中的一个来访问临界区，而且选择的是等待时间最长的。这两种模式只适用于**lock()**和**unlock()**方法。而**Lock**接口的**tryLock()**方法没有将线程置于休眠，**fair**属性并不影响这个方法。

在本节中，我们将修改“使用锁实现同步”一节的范例来使用这个属性，并观察公平模式和非公平模式之间的区别。

准备工作

我们将修改2.5节中的范例，请先行阅读。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现2.5节的范例。
2. 在打印队列类**PrintQueue**中，修改锁对象的构造方法，新的指令如下：

```
private Lock queueLock=new ReentrantLock(true);
```

3. 修改**printJob()**方法。将打印模拟分成两个代码块，在它们之间释放锁。

```
public void printJob(Object document) {
    queueLock.lock();
    try {
        Long duration=(long)(Math.random()*10000);
        System.out.println(Thread.currentThread().getName()+" :
            PrintQueue: Printing a Job during "+(duration/1000)+" seconds");
        Thread.sleep(duration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        queueLock.unlock();
    }
    queueLock.lock();
    try {
        Long duration=(long)(Math.random()*10000);
        System.out.println(Thread.currentThread().getName()+" :
            PrintQueue: Printing a Job during "+(duration/1000)+" seconds");
    }
```

```

        Thread.sleep(duration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        queueLock.unlock();
    }
}

```

4. 修改 **Main** 类的启动线程的代码块。新的代码块如下：

```

for (int i=0; i<10; i++) {
    thread[i].start();
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

工作原理

通过下面的截屏，可以看到范例运行的部分输出。

```

Problems @ Javadoc Declaration Console
<terminated> Main (14) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7
Thread 6: PrintQueue: Printing a Job during 8 seconds
Thread 7: PrintQueue: Printing a Job during 3 seconds
Thread 8: PrintQueue: Printing a Job during 2 seconds
Thread 9: PrintQueue: Printing a Job during 3 seconds
Thread 0: PrintQueue: Printing a Job during 0 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 7 seconds
Thread 1: The document has been printed
Thread 2: PrintQueue: Printing a Job during 8 seconds
Thread 2: The document has been printed
Thread 3: PrintQueue: Printing a Job during 0 seconds

```

所有线程创建的间隔是 0.1 秒。第一个线程请求锁是线程 0，然后是线程 1，以此类推。当线程 0 执行第一个加锁的代码块，其余 9 个线程将等待获取这个锁。当线程 0 释放了锁，

它立即又请求锁，这个时候就有 10 个线程试图获取锁。在公平模式下，**Lock** 接口将选择线程 1，因为这个线程等待的时间最久，然后，它选择线程 2，然后线程 3，以此类推。在所有线程都执行完第一个被锁保护的代码块之前，它们都没有执行第二个被锁保护的代码块。

当所有线程执行完第一个加锁代码块之后，又轮到了线程 0，然后是线程 1，以此类推。

现在来看看非公平模式，将传入锁构造器的参数设置为 **false**。在下面的截屏中，你将看到修改后范例的执行结果。

```
<terminated> Main (14) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7
Thread 8: Going to print a job
Thread 9: Going to print a job
Thread 0: PrintQueue: Printing a Job during 0 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 7 seconds
Thread 1: PrintQueue: Printing a Job during 1 seconds
Thread 1: The document has been printed
Thread 2: PrintQueue: Printing a Job during 9 seconds
Thread 2: PrintQueue: Printing a Job during 5 seconds
Thread 2: The document has been printed
Thread 3: PrintQueue: Printing a Job during 6 seconds
```

这里，所有线程是按顺序创建的，每个线程都执行两个被锁保护的代码块。然而，访问时线程并没有按照创建的先后顺序。如同前面解释的，锁将选择任一个线程并让它访问锁保护的代码。JVM 没有对线程的执行顺序提供保障。

更多信息

读/写锁的构造器也有一个公平策略的参数，这个参数的行为跟本节中讲解的一样。

参见

- ◆ 参见 2.5 节。
- ◆ 参见 2.6 节。
- ◆ 参见 7.9 节。

2.8 在锁中使用多条件 (Multiple Condition)

一个锁可能关联一个或者多个条件，这些条件通过 **Condition** 接口声明。目的是允许线程获取锁并且查看等待的某一个条件是否满足，如果不满足就挂起直到某个线程唤醒它们。**Condition** 接口提供了挂起线程和唤起线程的机制。

并发编程中的一个典型问题是**生产者-消费者** (**Producer-Consumer**) 问题。如本章前面提到的，我们使用一个数据缓冲区，一个或者多个**数据生产者** (**Producer**) 将数据保存到缓冲区，一个或者多个**数据消费者** (**Consumer**) 将数据从缓冲区中取走。

在本节中，我们将通过范例学习并使用锁和条件，来解决生产者-消费者问题。

准备工作

请先行阅读 2.5 节，以更好地理解本部分。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现一个文本文件模拟类 **FileMock**。它有两个属性：字符串数组 **content** 和 **int** 属性 **index**。属性 **content** 用来存储文件的内容，属性 **index** 用来表示要从这个文件读取的内容的行号。

```
public class FileMock {
    private String content[];
    private int index;
```

2. 通过 **FileMock** 类的构造器初始化文件内容 **content**，这里使用了随机字符。

```
public FileMock(int size, int length) {
    content=new String[size];
    for (int i=0; i<size; i++){
        StringBuilder buffer=new StringBuilder(length);
        for (int j=0; j<length; j++){
            int indice=(int)Math.random()*255;
            buffer.append((char)indice);
        }
    }
}
```

```

        content[i]=buffer.toString();
    }
    index=0;
}

```

3. 实现 **hasMoreLines()** 方法，如果文件有可以处理的数据行则返回 **true**，如果已经到达模拟文件的结尾则返回 **false**。

```

public boolean hasMoreLines() {
    return index<content.length;
}

```

4. 实现 **getLine()** 方法，它返回属性 **index** 指定的行内容，并且将 **index** 自动增加 1。

```

public String getLine() {
    if (this.hasMoreLines()) {
        System.out.println("Mock: "+(content.length-index));
        return content[index++];
    }
    return null;
}

```

5. 实现数据缓冲类 **Buffer**，它将被生产者和消费者共享。

```
public class Buffer {
```

6. 设置 **Buffer** 类的 6 个属性：

- **LinkedList<String>** 属性 **buffer**，用来存放共享数据；
- **int** 属性 **maxSize**，用来存放 **buffer** 的长度；
- **ReentrantLock** 对象 **lock**，用来对修改 **buffer** 的代码块进行控制；
- 两个 **Condition** 属性 **lines** 和 **space**；
- **boolean** 类型 **pendingLines**，用来表明缓冲区中是否还有数据。

```

private LinkedList<String> buffer;
private int maxSize;
private ReentrantLock lock;
private Condition lines;
private Condition space;
private boolean pendingLines;

```

7. 通过 **Buffer** 类的构造器，对这些属性进行初始化赋值。

```
public Buffer(int maxSize) {
    this.maxSize=maxSize;
    buffer=new LinkedList<>();
    lock=new ReentrantLock();
    lines=lock.newCondition();
    space=lock.newCondition();
    pendingLines=true;
}
```

8. 实现 **insert()**方法。这个方法的传入参数是字符串，它将把这个字符串写入到缓冲区中。首先，**insert()**方法要获取锁，当获取锁之后，它将检查这个缓冲区是否还有空位。如果缓冲区满了，它将调用条件 **space** 的 **await()**方法等待空位出现。当其他线程调用条件 **space** 的 **signal()** 或者 **signalAll()**方法时，这个线程将被唤醒。在有空位后，线程会将数据行保存到缓冲区中，并且调用条件 **lines** 的 **signalAll()**方法。一会儿我们会看到，条件 **lines** 将唤醒所有等待缓冲区中有数据的线程。

```
public void insert(String line) {
    lock.lock();
    try {
        while (buffer.size() == maxSize) {
            space.await();
        }
        buffer.offer(line);
        System.out.printf("%s: Inserted Line: %d\n", Thread.
            currentThread().getName(), buffer.size());
        lines.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

9. 实现 **get()**方法。它返回缓冲区中的第一个字符串。首先，**get()**方法要获取锁。当获取锁之后，它将检查缓冲区是不是有数据行。如果缓冲区是空的，就调用条件 **lines** 的 **await()**方法来等待缓冲区数据的出现。当其他线程调用条件 **lines** 的 **signal()**或者 **signalAll()**方法时，这个线程将被唤醒。在有数据后，**get()**方法获取缓冲区中的第一行，并且调用条

件 **space** 的 **signalAll()**方法，并且返回这个数据行字符串。

```
public String get() {
    String line=null;
    lock.lock();
    try {
        while ((buffer.size() == 0) && (hasPendingLines())) {
            lines.await();
        }
        if (hasPendingLines()) {
            line = buffer.poll();
            System.out.printf("%s: Line Readed: %d\n", Thread.
                currentThread().getName(), buffer.size());
            space.signalAll();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
    return line;
}
```

10. 实现 **setPendingLines()**方法，它将为属性 **pendingLines** 设置值。当生产者不再生产新的数据行时，线程将调用它。

```
public void setPendingLines(boolean pendingLines) {
    this.pendingLines=pendingLines;
}
```

11. 实现 **hasPendingLines()**方法。如果有数据行可以处理的时候返回 **true**，否则返回 **false**。

```
public boolean hasPendingLines() {
    return pendingLines || buffer.size()>0;
}
```

12. 实现生产者类 **Producer**，它实现了 **Runnable** 接口。

```
public class Producer implements Runnable {
```

13. 声明两个属性：一个是文件类 **FileMock** 对象，另一个是缓冲区类 **Buffer** 对象。

```
private FileMock mock;
private Buffer buffer;
```

14. 通过构造器为这两个属性设置值。

```
public Producer (FileMock mock, Buffer buffer){
    this.mock=mock;
    this.buffer=buffer;
}
```

15. 实现 **run()** 方法。这个方法用来读文件 **FileMock** 中所有的数据行，并且使用 **insert()** 方法将读到的数据行插入到缓冲区。一旦读完文件，将调用 **setPendingLines()** 方法来通知缓冲区停止生成更多的行。

```
@Override
public void run() {
    buffer.setPendingLines(true);
    while (mock.hasMoreLines()) {
        String line=mock.getLine();
        buffer.insert(line);
    }
    buffer.setPendingLines(false);
}
```

16. 实现消费者类 **Consumer**，它实现了 **Runnable** 接口。

```
public class Consumer implements Runnable {
```

17. 声明缓冲区类 **Buffer** 对象，并通过构造器对它进行初始化。

```
private Buffer buffer;
public Consumer (Buffer buffer) {
    this.buffer=buffer;
}
```

18. 实现 **run()** 方法，如果缓冲区有数据行，它将获取一行并且进行处理。

```
@Override
public void run() {
```

```
        while (buffer.hasPendingLines()) {
            String line=buffer.get();
            processLine(line);
        }
    }
```

19. 实现辅助方法 **processLine()**。该方法仅用于休眠 10 毫秒以模拟对数据行的处理。

```
private void processLine(String line) {
    try {
        Random random=new Random();
        Thread.sleep(random.nextInt(100));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

20. 实现范例的主程序 **Main**，并为它添加 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

21. 创建一个文件类 **FileMock** 对象。

```
    FileMock mock=new FileMock(100, 10);
```

22. 创建一个缓冲区 **Buffer** 对象。

```
    Buffer buffer=new Buffer(20);
```

23. 创建一个生产者对象 **Producer**，并将它作为传入参数创建线程。

```
    Producer producer=new Producer(mock, buffer);
    Thread threadProducer=new Thread(producer,"Producer");
```

24. 创建一个消费者对象 **Consumer**，并将它作为传入参数创建线程。

```
    Consumer consumers[]=new Consumer[3];
    Thread threadConsumers[]=new Thread[3];
    for (int i=0; i<3; i++){
        consumers[i]=new Consumer(buffer);
        threadConsumers[i]=new Thread(consumers[i],"Consumer "+i);
```

```
}
```

25. 启动生产者和消费者对象线程。

```

threadProducer.start();
for (int i=0; i<3; i++) {
    threadConsumers[i].start();
}

```

工作原理

与锁绑定的所有条件对象都是通过 **Lock** 接口声明的 **newCondition()** 方法创建的。在使用条件的时候，必须获取这个条件绑定的锁，所以带条件的代码必须在调用 **Lock** 对象的 **lock()** 方法和 **unlock()** 方法之间。

当线程调用条件的 **await()** 方法时，它将自动释放这个条件绑定的锁，其他某个线程才可以获取这个锁并且执行相同的操作，或者执行这个锁保护的另一个临界区代码。

备注：当一个线程调用了条件对象的 **signal()** 或者 **signalAll()** 方法后，一个或者多个在该条件下挂起的线程将被唤醒，但这并不能保证让它们挂起的条件已经满足，所以必须在 **while** 循环中调用 **await()**，在条件成立之前不能离开这个循环。如果条件不成立，将再次调用 **await()**。

必须小心使用 **await()** 和 **signal()** 方法。如果调用了一个条件的 **await()** 方法，却从不调用它的 **signal()** 方法，这个线程将永久休眠。

因调用 **await()** 方法进入休眠的线程可能会被中断，所以必须处理 **InterruptedException** 异常。

更多信息

Condition 接口还提供了 **await()** 方法的其他形式。

await(long time, TimeUnit unit)，直到发生以下情况之一之前，线程将一直处于休眠状态。

- ◆ 其他某个线程中断当前线程。
- ◆ 其他某个线程调用了将当前线程挂起的条件的 **singal()** 或 **signalAll()** 方法。
- ◆ 指定的等待时间已经过去。

- ◆ 通过 TimeUnit 类的常量 DAYS、HOURS、MICROSECONDS、MILLISECONDS、MINUTES、ANOSECONDS 和 SECONDS 指定的等待时间已经过去。

awaitUninterruptibly(): 它是不可中断的。这个线程将休眠直到其他某个线程调用了将它挂起的条件的 **singal()** 或 **signalAll()** 方法。

awaitUntil(Date date) : 直到发生以下情况之一之前，线程将一直处于休眠状态。

- ◆ 其他某个线程中断当前线程。
- ◆ 其他某个线程调用了将它挂起的条件的 **singal()** 或 **signalAll()** 方法。
- ◆ 指定的最后期限到了。

也可以将条件与读写锁 **ReadLock** 和 **WriteLock** 一起使用。

参见

- ◆ 参见 2.5 节。
- ◆ 参见 2.6 节。

第 3 章

线程同步辅助类

在本章中，我们将学习：

- ◆ 资源的并发访问控制
- ◆ 资源的多副本的并发访问控制
- ◆ 等待多个并发事件的完成
- ◆ 在集合点的同步
- ◆ 并发阶段任务的运行
- ◆ 并发阶段任务中的阶段切换
- ◆ 并发任务间的数据交换

3.1 简介

在第 2 章中，我们学习了同步和临界区的概念，并且主要讨论了多个并发任务共享一个资源时的同步情况，这个共享资源可以是一个对象也可以是对象的一个属性。访问共享资源的代码块叫做临界区。

如果不采取合适的机制，就可能得到错误的结果、不一致的数据或者错误的条件，所以必须使用 Java 语言提供的某种同步机制来避免这类问题。

在第 2 章中，我们学习了下面的基本同步机制：

- ◆ synchronized 关键字；
- ◆ Lock 接口及其实现类，如 ReentrantLock、ReentrantReadWriteLock.ReadLock 和

ReentrantReadWriteLock.WriteLock。

在本章中，我们将学习如何使用更高级的同步机制来实现多线程间的同步。

◆ **信号量 (Semaphore)**: 是一种计数器，用来保护一个或者多个共享资源的访问。它是并发编程的一种基础工具，大多数编程语言都提供了这个机制。

◆ **CountDownLatch**: 是 Java 语言提供的同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许线程一直等待。

◆ **CyclicBarrier**: 也是 Java 语言提供的同步辅助类，它允许多个线程在某个集合点 (common point) 处进行相互等待。

◆ **Phaser**: 也是 Java 语言提供的同步辅助类。它把并发任务分成多个阶段运行，在开始下一阶段之前，当前阶段中的所有线程都必须执行完成，这是 Java7 API 中的新特性。

◆ **Exchanger**: 也是 Java 语言提供的同步辅助类。它提供了两个线程之间的数据交换点。

在应用程序中，任何时候都可以使用 **Semaphore** 来保护临界区，因为它是一个基础的同步机制。而其他的同步机制，则需要根据各自的上述特性来对其选择使用。所以我们需要根据应用程序的特性来选择合适的同步机制。

本章分为七个节来讲解如何使用上述的各种同步机制。

3.2 资源的并发访问控制

在本节中，我们将学习如何使用 Java 语言提供的信号量 (Semaphore)¹ 机制。信号量是一种计数器，用来保护一个或者多个共享资源的访问。

如果线程要访问一个共享资源，它必须先获得信号量。如果信号量的内部计数器大于 0，信号量将减 1，然后允许访问这个共享资源。计数器大于 0 意味着有可以使用的资源，因此线程将被允许使用其中一个资源。

否则，如果信号量的计数器等于 0，信号量将会把线程置入休眠直至计数器大于 0。计数器等于 0 的时候意味着所有的共享资源已经被其他线程使用了，所以需要访问这个共享资源的线程必须等待。

¹ 译者注：信号量概念是生于荷兰鹿特丹的计算机科学家 Edsger Dijkstra（艾兹赫尔·戴克斯特拉）在 1965 年提出的，他曾在 1972 年获得过素有计算机科学界的诺贝尔奖之称的图灵奖，并且信号量第一次被使用在 THEOS 操作系统中，THEOS 是从 1977 年开始由 TheOS Software 公司开发的，是个人电脑上最早的多用户多任务操作系统之一。

当线程使用完某个共享资源时，信号量必须被释放，以便其他线程能够访问共享资源。释放操作将使信号量的内部计数器增加 1。

本节中，我们将学习如何使用信号量类 **Semaphore** 来实现二进制信号量（Binary Semaphore）。二进制信号量是一种比较特殊的信号量，用来保护对唯一共享资源的访问，因而它的内部计数器只有 0 和 1 两个值。为了演示它的使用方式，我们将实现一个打印队列，并发任务将使用它来完成打印。这个打印队列受二进制信号量保护，因而同时只有一个线程可以执行打印。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个打印队列类 **PrintQueue**。

```
public class PrintQueue {
```

2. 声明一个信号量对象 **semaphore**。

```
    private final Semaphore semaphore;
```

3. 实现这个类的构造器。它初始化了信号量对象，以保护对打印队列的访问。

```
    public PrintQueue() {
        semaphore=new Semaphore(1);
    }
```

4. 实现 **printJob()** 方法，它模拟了文档的打印。它的传入参数是文档对象 **document**。

```
    public void printJob (Object document) {
```

5. 通过调用 **acquire()** 方法获得信号量，它会抛出 **InterruptedException** 异常，然后必须捕获并处理这个异常。

```
        try {
            semaphore.acquire();
```

6. 实现模拟文档的打印，然后等待一段随机时间。

```
long duration=(long)(Math.random()*10);
System.out.printf("%s: PrintQueue: Printing a Job during %d
seconds\n",Thread.currentThread().getName(),duration);
Thread.sleep(duration);
```

7. 通过调用信号量的 **release()** 方法释放信号量。

```
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    semaphore.release();
}
```

8. 创建一个工作类 **Job**，并且实现 **Runnable** 接口。这个类将文档发送到打印机。

```
public class Job implements Runnable {
```

9. 声明一个打印队列对象 **printQueue**。

```
private PrintQueue printQueue;
```

10. 实现构造器，用来初始化打印队列对象 **printQueue**。

```
public Job(PrintQueue printQueue) {
    this.printQueue=printQueue;
}
```

11. 实现 **run()** 方法。

```
@Override
public void run() {
```

12. 首先，这个方法将已经开始运行的 **Job** 信息打印到控制台。

```
System.out.printf("%s: Going to print a job\n",Thread.
currentThread().getName());
```

13. 调用打印队列对象的 **printJob()** 方法。

```
printQueue.printJob(new Object());
```

14. 将工作完成的信息打印到控制台。

```
System.out.printf("%s: The document has been printed\n",
                  Thread.currentThread().getName());
}
```

15. 实现范例的主类 **Main**, 并实现 **main()**方法。

```
public class Main {
    public static void main (String args[]){
```

16. 创建打印队列对象 **printQueue**。

```
PrintQueue printQueue=new PrintQueue();
```

17. 将工作类 **Job** 对象作为传入参数创建 10 个线程, 因而每个线程都将发送文档到打印队列。

```
Thread thread[]=new Thread[10];
for (int i=0; i<10; i++){
    thread[i]=new Thread(new Job(printQueue), "Thread"+i);
}
```

18. 最后, 启动这 10 个线程。

```
for (int i=0; i<10; i++){
    thread[i].start();
}
```

工作原理

这个范例的核心部分是打印队列类 **PrintQueue** 的 **printJob()** 方法。它指出了使用信号量实现临界区必须遵循的三个步骤, 从而保护对共享资源的访问:

首先, 必须通过 **acquire()**方法获得信号量;

其次, 使用共享资源执行必要的操作;

最后, 必须通过**release()**方法释放信号量。

这个范例的另一个要点是打印队列类 **PrintQueue** 的构造器, 它初始化了信号量对象。

范例中将 1 作为传入参数，所以创建的就是二进制信号量。信号量的内部计数器初始值是 1，所以它只能保护一个共享资源的访问，如本例中的打印队列。

当启动 10 个线程时，第一个获得信号量的线程将能够访问临界区，其余的线程将被信号量阻塞，直到信号量被释放。一旦信号量被释放，信号量将选择一个正在等待的线程并且允许它访问临界区，从而所有的工作都将一个接一个地打印它们的文档。

更多信息

Semaphore 类还有其他两种 acquire()方法。

◆ **acquireUninterruptibly()**: 它其实就是 **acquire()** 方法。当信号量的内部计数器变成 0 的时候，信号量将阻塞线程直到其被释放。线程在被阻塞的这段时间中，可能会被中断，从而导致 **acquire()** 方法抛出 **InterruptedException** 异常。而 **acquireUninterruptibly()** 方法会忽略线程的中断并且不会抛出任何异常。

◆ **tryAcquire()**: 这个方法试图获得信号量。如果能获得就返回 **true**；如果不能，就返回 **false**，从而避开线程的阻塞和等待信号量的释放。我们可以根据返回值是 **true** 还是 **false** 来做出恰当的处理。

信号量的公平性

在 Java 语言中，只要一个类可能出现多个线程被阻塞并且等待同步资源的释放（例如信号量），就会涉及公平性概念。默认的模式是非公平模式。在这种模式中，被同步的资源被释放后，所有等待的线程中会有一个被选中来使用共享资源，而这个选择是没有任何条件的。公平模式则不然，它选择的是等待共享资源时间最长的那个线程。

跟其他的类一样，**Semaphore** 类的构造器也提供了第二个传入参数。这个参数是 **boolean** 型的。如果传入 **false** 值，那么创建的信号量就是非公平模式的，与不使用这个参数的效果一样。如果传入 **true** 值，那么创建的信号量是公平模式的。

参见

参见 8.2 节。

参见 2.7 节。

3.3 资源的多副本的并发访问控制

在 3.2 节中，我们学习了基础的信号量。

范例使用了二进制信号量。这种信号量可以保护对单一共享资源，或者单一临界区的访问，从而使得保护的资源在同一个时间内只能被一个线程访问。然而，信号量也可以用来保护一个资源的多个副本，或者被多个线程同时执行的临界区。

在本节中，我们将学习如何使用信号量来保护一个资源的多个副本。我们将实现这样的范例：一个打印队列，它将被三个不同的打印机使用。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例的实现参见 3.2 节。

范例实现

按照接下来的步骤实现本节的范例。

1. 修改上一节中的打印队列范例，它是使用信号量实现的。打开打印队列类 **PrintQueue**，然后声明一个 **boolean** 型数组 **freePrinters**，用来存放打印机的状态，即空闲或正在打印。

```
private boolean freePrinters[];
```

2. 然后声明一个锁对象 **lockPrinters**，用来保护对 **freePrinters** 数组的访问。

```
private Lock lockPrinters;
```

3. 修改构造器，并初始化这两个新声明的对象。将 **freePrinters** 数组的三个元素都初始化为 **true**。信号量初始化为 **3**。

```
public PrintQueue() {
    semaphore=new Semaphore(3);
    freePrinters=new boolean[3];
```

```
    for (int i=0; i<3; i++) {
        freePrinters[i]=true;
    }
    lockPrinters=new ReentrantLock();
}
```

4. 修改 **printJob()**方法，它的传入参数是对象 **document**。

```
public void printJob (Object document) {
```

5. 调用 **acquire()**方法获得信号量。由于这个方法会抛出 **InterruptedException** 异常，所以必须捕获并处理这个异常。

```
try {
    semaphore.acquire();
```

6. 使用私有函数 **getPrinter()**获得分配打印工作的打印机编号。

```
int assignedPrinter=getPrinter();
```

7. 实现模拟文档的打印，然后等待一段随机时间。

```
long duration=(long)(Math.random()*10);
System.out.printf("%s: PrintQueue: Printing a Job in Printer
                  %d during %d seconds\n",Thread.currentThread().getName(),
                  assignedPrinter,duration);
TimeUnit.SECONDS.sleep(duration);
```

8. 调用 **release()**方法释放信号量，并将打印机标记为空闲，即将这个打印机对应的 **freePrinters** 数组中的值设置成 **true**。

```
    freePrinters[assignedPrinter]=true;
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    semaphore.release();
}
```

9. 实现 **getPrinter()**方法。它是一个私有函数并且返回 **int** 值。

```
private int getPrinter() {
```

10. 声明一个 **int** 变量来存储打印机的编号。

```
int ret=-1;
```

11. 获得锁。

```
try {
    lockPrinters.lock();
```

12. 在 **freePrinters** 数组中找到第一个 **true** 值并把索引保存到 **ref** 变量中，将 **true** 值重置为 **false**，意味着被定位的打印机将要执行打印工作。

```
for (int i=0; i<freePrinters.length; i++) {
    if (freePrinters[i]){
        ret=i;
        freePrinters[i]=false;
        break;
    }
}
```

13. 释放锁对象，并且返回刚刚获得的打印机编号。

```
} catch (Exception e) {
    e.printStackTrace();
} finally {
    lockPrinters.unlock();
}
return ret;
```

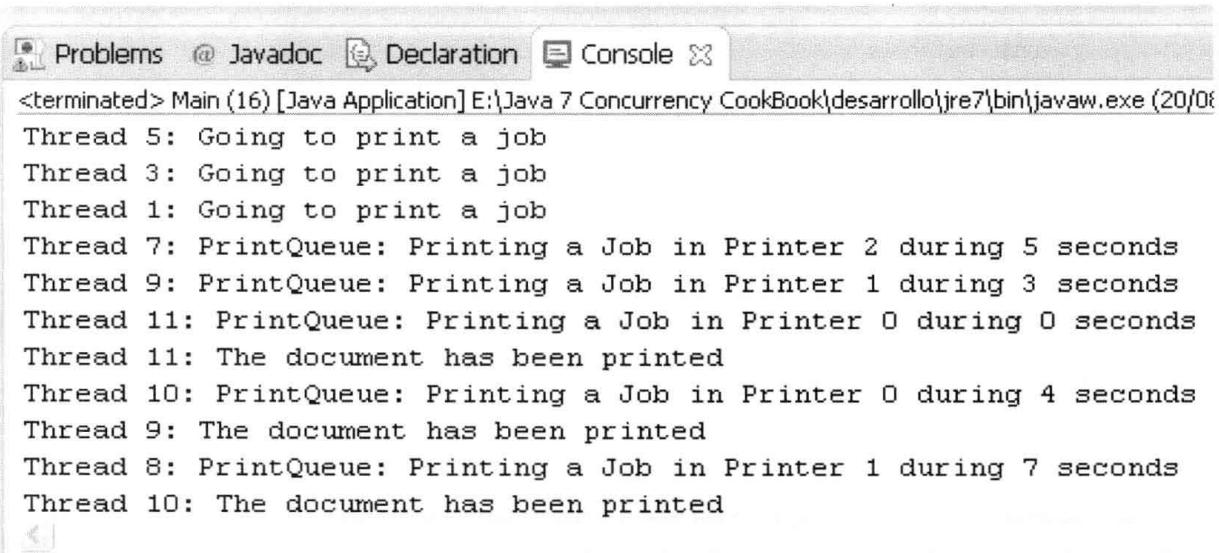
14. 打印类 **Job** 和主类 **Main** 不需要改动。

工作原理

这个范例的核心是打印队列类 **PrintQueue**，在它的构造器中使用 **3** 作为传入参数来创建信号量对象。在本例中，最开始调用 **acquire()** 方法的 3 个线程将获得对临界区的访问，其余的线程将被阻塞。当一个线程完成了对临界区的访问并且释放了信号量，另一个线程将获得这个信号量。

在临界区代码中，线程获得可以分配打印工作的打印机编号。范例中的这部分代码只是为了给出更完整的实现，并没有使用信号量相关的代码。

下面的截屏显示了范例的运行结果。



The screenshot shows a Java application running in an IDE's console tab. The output displays several threads (Thread 1 through Thread 11) performing printing tasks. Thread 5, Thread 3, and Thread 1 are each printing a job. Thread 7 is printing a job in Printer 2 over 5 seconds. Thread 9 is printing a job in Printer 1 over 3 seconds. Thread 11 is printing a job in Printer 0 over 0 seconds. All threads eventually report that the document has been printed.

```
<terminated> Main (16) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (20/0)
Thread 5: Going to print a job
Thread 3: Going to print a job
Thread 1: Going to print a job
Thread 7: PrintQueue: Printing a Job in Printer 2 during 5 seconds
Thread 9: PrintQueue: Printing a Job in Printer 1 during 3 seconds
Thread 11: PrintQueue: Printing a Job in Printer 0 during 0 seconds
Thread 11: The document has been printed
Thread 10: PrintQueue: Printing a Job in Printer 0 during 4 seconds
Thread 9: The document has been printed
Thread 8: PrintQueue: Printing a Job in Printer 1 during 7 seconds
Thread 10: The document has been printed
```

我们可以看到，每个文档的打印都使用了第一个空闲的打印机。

更多信息

acquire()、**acquireUninterruptibly()**、**tryAcquire()**和**release()**方法都有另一种实现方式，即提供了一个 int 型的传入参数。这个参数声明了线程试图获取或者释放的共享资源数目，也就是这个线程想要在信号量内部计数器上删除或增加的数目。对于 **acquire()**、**acquireUninterruptibly()**、**tryAcquire()**方法来讲，如果计数器的值少于参数对应的值，那么线程将被阻塞直到计数器重新累加到或者超过这个值。

参见

- ◆ 参见 3.2 节。
- ◆ 参见 8.2 节。
- ◆ 参见 2.7 节。

3.4 等待多个并发事件的完成

Java 并发 API 提供了 **CountDownLatch** 类，它是一个同步辅助类。在完成一组正在其他线程中执行的操作之前，它允许线程一直等待。这个类使用一个整数进行初始化，这个整数就是线程要等待完成的操作的数目。当一个线程要等待某些操作先执行完时，需要调用 **await()** 方法，这个方法让线程进入休眠直到等待的所有操作都完成。当某一个操作完成后，它将调用 **countDown()** 方法将 **CountDownLatch** 类的内部计数器减 1。当计数器变成 0 的时候，**CountDownLatch** 类将唤醒所有调用 **await()** 方法而进入休眠的线程。

在本节中，我们将学习如何使用 **CountDownLatch** 类实现视频会议系统。这个视频会议系统将等待所有的参会者都到齐才开始。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建视频会议类 **Videoconference**，并且实现 **Runnable** 接口。这个类实现的是一个视频会议系统。

```
public class Videoconference implements Runnable{
```

2. 声明 **CountDownLatch** 对象 **controller**。

```
private final CountDownLatch controller;
```

3. 实现构造器，并且初始化 **controller** 属性。

```
public Videoconference(int number) {
    controller=new CountDownLatch(number);
}
```

4. 实现 **arrive()** 方法。每一个与会者进入视频会议的时候，这个方法将被调用。它的

传入参数是 **String** 型变量 **name**。

```
public void arrive(String name) {
```

5. 打印出与会者到达的信息。

```
    System.out.printf("%s has arrived.", name);
```

6. 调用 **controller** 的 **countDown()**方法。

```
    controller.countDown();
```

7. 打印出还没有到达的与会者的数目，通过 **CountDownLatch** 对象的 **getCount()**方法实现。

```
System.out.printf("VideoConference: Waiting for %d participants.\n", controller.getCount());
```

8. 实现视频会议系统的 **run()**方法，这是 **Runnable** 对象必须实现的。

```
@Override  
public void run() {
```

9. 使用 **getCount()**方法打印出这次视频会议的人数。

```
System.out.printf("VideoConference: Initialization: %d participants.\n", controller.getCount());
```

10. 使用 **await()**方法等待所有的与会者到达。由于这个方法会抛出 **InterruptedException** 异常，所以必须捕获并处理这个异常。

```
try {  
    controller.await();
```

11. 当所有的与会者都到齐后，打印出与会者到齐会议开始的信息。

```
System.out.printf("VideoConference: All the participants have come\n");  
System.out.printf("VideoConference: Let's start...\n");  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

12. 创建与会者类 **Participant**, 并且实现 **Runnable** 接口。这个类表示的是视频会议的与会者。

```
public class Participant implements Runnable {
```

13. 声明 **Videoconference** 型私有属性 **conference**。

```
private Videoconference conference;
```

14. 声明 **String** 型私有属性 **name**。

```
private String name;
```

15. 实现构造器并初始化这两个属性。

```
public Participant(Videoconference conference, String name) {
    this.conference=conference;
    this.name=name;
}
```

16. 实现 **run()**方法。

```
@Override
public void run() {
```

17. 将线程休眠一段随机时间。

```
long duration=(long)(Math.random()*10);
try {
    TimeUnit.SECONDS.sleep(duration);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

18. 使用视频会议对象 **conference** 的 **arrive()**方法来表明一个与会者的到来。

```
conference.arrive(name);
```

19. 实现范例的主类 **Main**, 并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

20. 创建视频会议对象 **conference**, 它要等待 10 个与会者到齐。

```
Videoconference conference=new Videoconference(10);
```

21. 将视频会议对象作为传入参数创建线程，并且启动。

```
Thread threadConference=new Thread(conference);
threadConference.start();
```

22. 创建 10 个与会者对象，分别将每一个作为传入参数创建线程，并且启动。

```
for (int i=0; i<10; i++){
    Participant p=new Participant(conference, "Participant "+i);
    Thread t=new Thread(p);
    t.start();
}
```

工作原理

CountDownLatch 类有三个基本元素：

- ◆ 一个初始值，即定义必须等待的先行完成的操作的数目；
- ◆ **await()**方法，需要等待其他事件先完成的线程调用；
- ◆ **countDown()**方法，每个被等待的事件在完成的时候调用。

当创建 **CountDownLatch** 对象时，使用构造器来初始化内部计数器。当 **countDown()** 方法被调用后，计数器将减 1。当计数器到达 0 的时候，**CountDownLatch** 对象将唤起所有在 **await()** 方法上等待的线程。

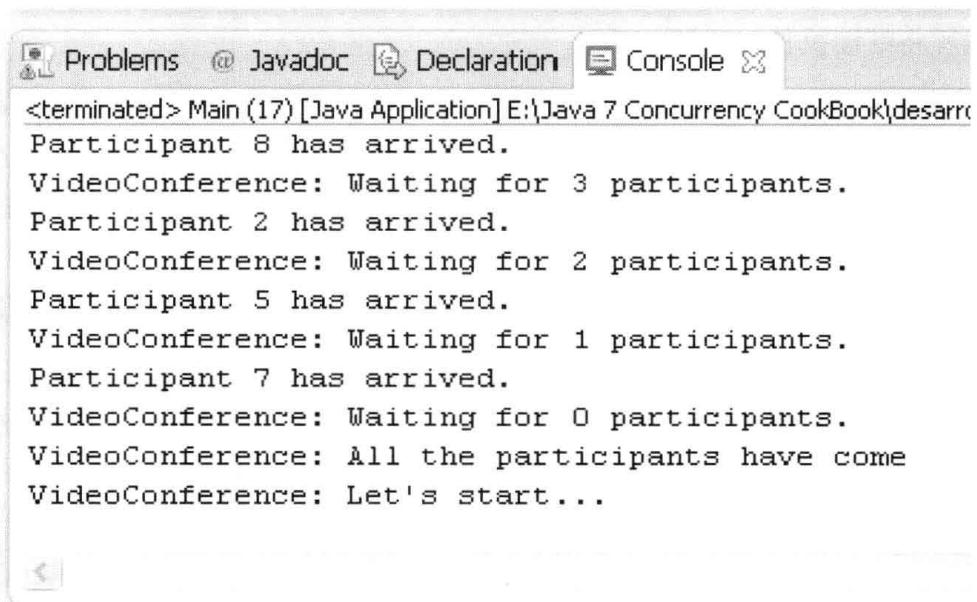
CountDownLatch 对象的内部计数器被初始化之后就不能被再次初始化或者修改。一旦计数器被初始化后，唯一能改变参数值的方法是 **countDown()** 方法。当计数器到达 0 时，所有因调用 **await()** 方法而等待的线程立刻被唤醒，再执行 **countDown()** 将不起任何作用。

和其他同步方法相比，**CountDownLatch** 机制有下述不同。

- ◆ **CountDownLatch** 机制不是用来保护共享资源或者临界区的，它是用来同步执行多个任务的一个或者多个线程；
- ◆ **CountDownLatch** 只准许进入一次。如同刚刚解释的，一旦 **CountDownLatch** 的内部计数器到达 0，再调用这个方法将不起作用。如果要做类似的同步，就必须创建一个

新的 **CountDownLatch** 对象。

下面的截屏显示了范例运行的结果。



```

Problems Javadoc Declaration Console
<terminated> Main (17) [Java Application] E:\Java 7 Concurrency CookBook\desarroll
Participant 8 has arrived.
VideoConference: Waiting for 3 participants.
Participant 2 has arrived.
VideoConference: Waiting for 2 participants.
Participant 5 has arrived.
VideoConference: Waiting for 1 participants.
Participant 7 has arrived.
VideoConference: Waiting for 0 participants.
VideoConference: All the participants have come
VideoConference: Let's start...

```

可以看到，最后一个与会者到达时，内部计数器变成 0，这时 **CountDownLatch** 对象唤醒了 **Videoconference** 对象，后者打印出的信息表明会议可以开始了。

更多信息

CountDownLatch 类提供了另外一种 **await()** 方法，即 **await(long time, TimeUnit unit)**。这个方法被调用后，线程将休眠直到被中断，或者 **CountDownLatch** 的内部计数器达到 0，或者指定的时间已经过期。第二个参数是 **TimeUnit** 类型，**TimeUnit** 类是以下常量的枚举：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

3.5 在集合点的同步

Java 并发 API 提供了 **CyclicBarrier** 类，它也是一个同步辅助类。它允许两个或者多个线程在某个点上进行同步。这个类与上一节所讲述的 **CountDownLatch** 类类似，但也有不同之处，使之成为更强大的类。

CyclicBarrier 类使用一个整型数进行初始化，这个数是需要在某个点上同步的线程数。

当一个线程到达指定的点后，它将调用 **await()** 方法等待其他的线程。当线程调用 **await()** 方法后，**CyclicBarrier** 类将阻塞这个线程并使之休眠直到所有其他线程到达。当最后一个线程调用 **CyclicBarrier** 类的 **await()** 方法时，**CyclicBarrier** 对象将唤醒所有在等待的线程，然后这些线程将继续执行。

CyclicBarrier 类有一个很有意义的改进，即它可以传入另一个 **Runnable** 对象作为初始化参数。当所有的线程都到达集合点后，**CyclicBarrier** 类将这个 **Runnable** 对象作为线程执行。这个特性使得这个类在并行任务上可以媲美分治编程技术（Divide and Conquer Programming Technique）。

在本节中，我们将学习如何使用 **CyclicBarrier** 类使一组线程在集合点上同步。在所有线程都到达集合点后，我们将使用 **Runnable** 对象并且运行它。在这个范例中，我们将在数字矩阵中寻找一个数字（使用分治编程技术）。这个矩阵会被分成几个子集，然后每个线程在一个子集中查找。一旦所有线程都完成查找，最终的任务将统一这些结果。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 通过实现两个辅助类来开始这个范例。创建一个矩阵类 **MatrixMock**，用来生成一个 1~10 组成的随机矩阵，线程将从这个矩阵中查找指定的数字。

```
public class MatrixMock {
```

2. 声明私有二维数组 **data**。

```
    private int data[][];
```

3. 实现构造器。构造器的传入参数是矩阵的行数，每行的长度，要寻找的数字。这三个参数都是 **int** 型的。

```
    public MatrixMock(int size, int length, int number) {
```

4. 初始化构造器使用的变量和对象。

```
int counter=0;
data=new int[size][length];
Random random=new Random();
```

5. 用随机数字为矩阵赋值。每生成一个数字，就用它跟要查找的数字进行比较。如果一致，就将计数器 **counter** 加 1。

```
for (int i=0; i<size; i++) {
    for (int j=0; j<length; j++) {
        data[i][j]=random.nextInt(10);
        if (data[i][j]==number) {
            counter++;
        }
    }
}
```

6. 将在矩阵中查找到的次数打印到控制台。这个信息将用来检查线程是否得到了正确的结果。

```
System.out.printf("Mock: There are %d occurrences of number in
generated data.\n",counter,number);
```

7. 实现 **getRow()** 方法。这个方法的传入参数是 **int** 型的矩阵行序号，如果矩阵中存在这个行，就返回行数据，如果不存在就返回 **null**。

```
public int[] getRow(int row) {
    if ((row>=0)&&(row<data.length)){
        return data[row];
    }
    return null;
}
```

8. 实现结果类 **Results**。这个类将保存矩阵中每行找到指定数字的次数。

```
public class Results {
```

9. 声明私有 **int** 数组 **data**。

```
private int data[];
```

10. 实现构造器。它的传入参数将指定这个 **data** 数组的长度。

```
public Results(int size){  
    data=new int[size];  
}
```

11. 实现 **setData()** 方法。它的传入参数指定了数组的索引 **position** 及其对应的值 **value**。

```
public void setData(int position, int value){  
    data[position]=value;  
}
```

12. 实现 **getData()** 方法。这个方法返回结果数组。

```
public int[] getData(){  
    return data;  
}
```

13. 到此为止我们实现了辅助类，现在来实现线程。首先，实现查找类 **Searcher**。它在随机数矩阵指定的行中查找某个数。创建 **Searcher** 类并实现 **Runnable** 接口。

```
public class Searcher implements Runnable {
```

14. 声明两个名为 **firstRow** 和 **lastRow** 的私有 **int** 属性。这两个属性将决定查找的子集范围。

```
private int firstRow;  
private int lastRow;
```

15. 声明私有 **MatrixMock** 属性 **mock**。

```
private MatrixMock mock;
```

16. 声明私有 **Results** 属性 **results**。

```
private Results results;
```

17. 声明私有 **int** 属性 **number**，用于存放要查找的数字。

```
private int number;
```

18. 声明 **CyclicBarrier** 类对象 **barrier**。

```
private final CyclicBarrier barrier;
```

19. 实现构造器，初始化刚刚声明的属性。

```
public Searcher(int firstRow, int lastRow, NumberMock mock,
    Results results, int number, CyclicBarrier barrier) {
    this.firstRow=firstRow;
    this.lastRow=lastRow;
    this.mock=mock;
    this.results=results;
    this.number=number;
    this.barrier=barrier;
}
```

20. 实现 **run()**方法，它将查找数字。它使用内部变量 **counter** 来存放每行查找到的次数。

```
@Override
public void run() {
    int counter;
```

21. 将查找的范围打印到控制台。

```
System.out.printf("%s: Processing lines from %d to %d.\n",
    Thread.currentThread().getName(), firstRow, lastRow);
```

22. 在要查找的所有行中进行查找。对每一行查找指定的数字，并将查找到的次数保存到对应的 **results** 对象的相应位置。

```
for (int i=firstRow; i<lastRow; i++) {
    int row[]=mock.getRow(i);
    counter=0;
    for (int j=0; j<row.length; j++) {
        if (row[j]==number) {
            counter++;
        }
    }
    results.setData(i, counter);
}
```

23. 将线程查找完成的信息打印到控制台。

```
System.out.printf("%s: Lines processed.\n",
    Thread.currentThread().getName());
```

24. 调用 **CyclicBarrier** 对象的 **await()**方法，并捕获及处理方法抛出的异常 **InterruptedException** 和 **BrokenBarrierException**。

```
try {
    barrier.await();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (BrokenBarrierException e) {
    e.printStackTrace();
}
```

25. 实现一个类来计算在矩阵中查找到的总次数。计算是基于 **Results** 对象的，后者存放了矩阵中每行查找到的次数。创建 **Grouper** 类并指定它实现 **Runnable** 接口。

```
public class Grouper implements Runnable {
```

26. 声明私有结果类 **Results** 的 **results** 对象。

```
private Results results;
```

27. 实现构造器并初始化结果类对象。

```
public Grouper(Results results) {
    this.results=results;
}
```

28. 实现 **run()**方法，用来计算在结果类数组中查找的次数。

```
@Override
public void run() {
```

29. 声明 **int** 变量 **finalResult**，并将开始统计的信息打印到控制台。

```
int finalResult=0;
System.out.printf("Grouper: Processing results...\n");
```

30. 使用 **results** 对象的 **getData()**方法获得存放每行发生次数的数组。然后，处理这个数组并将结果累加到 **finalResult** 变量。

```
int data[]={results.getData();
for (int number:data){
```

```

    finalResult+=number;
}

```

31. 将结果打印到控制台。

```
System.out.printf("Grouper: Total result: %d.\n", finalResult);
```

32. 实现范例的主类 **Main** 并实现 **main()** 方法。

```

public class Main {
    public static void main(String[] args) {

```

33. 声明和初始化 5 个常量。它们将作为这个应用程序的参数。

```

final int ROWS=10000;
final int NUMBERS=1000;
final int SEARCH=5;
final int PARTICIPANTS=5;
final int LINES_PARTICIPANT=2000;

```

34. 创建矩阵类 **MatrixMock** 对象 **mock**。它有 1000 行，每行有 1000 个数字，而这里要查找的是数字 5。

```
MatrixMock mock=new MatrixMock(ROWS, NUMBERS, SEARCH);
```

35. 创建结果类 **Results** 对象 **results**。它有 1000 个元素。

```
Results results=new Results(ROWS);
```

36. 创建 **Grouper** 对象 **grouper**。

```
Grouper grouper=new Grouper(results);
```

37. 创建 **CyclicBarrier** 类对象 **barrier**。这个对象将等待 5 个线程运行结束然后，它将执行创建的 **Grouper** 线程对象。

```
CyclicBarrier barrier=new CyclicBarrier(PARTICIPANTS,grouper);
```

38. 创建 5 个查找类 **Searcher** 对象，将它们分别作为传入参数创建线程并启动。

```

Searcher searchers[]=new Searcher[PARTICIPANTS];
for (int i=0; i<PARTICIPANTS; i++){

```

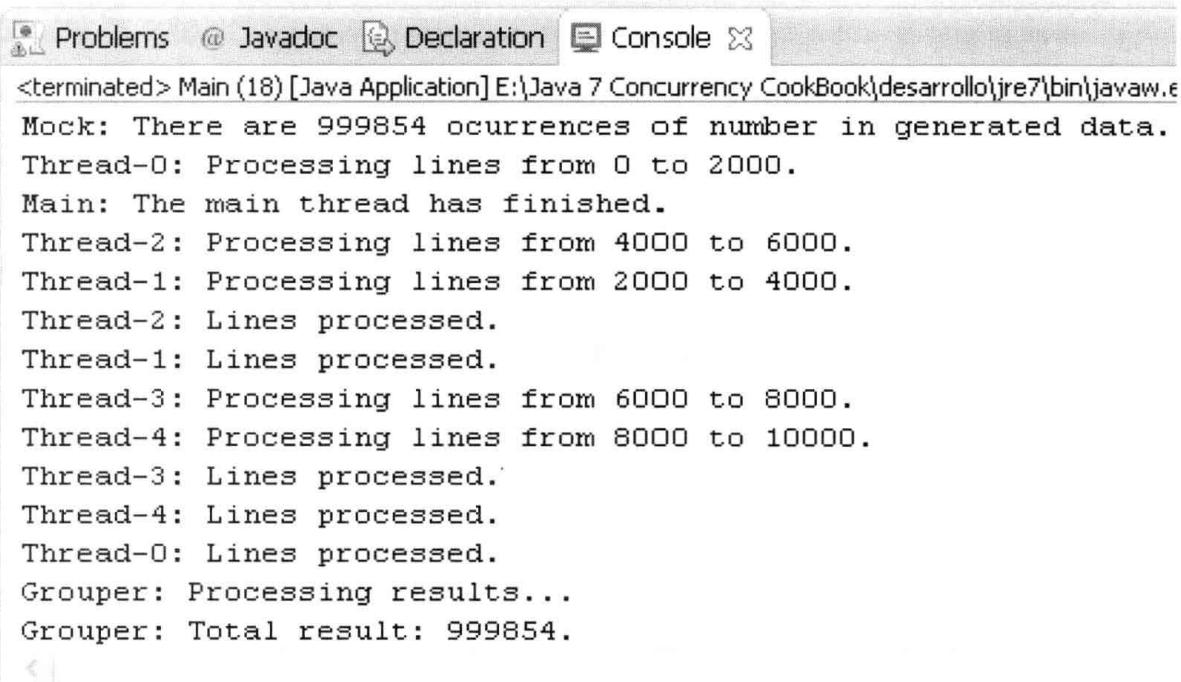
```

        searchers[i]=new Searcher(i*LINES_PARTICIPANT, (i*LINES_
            PARTICIPANT)+LINES_PARTICIPANT, mock, results, 5,barrier);
        Thread thread=new Thread(searchers[i]);
        thread.start();
    }
    System.out.printf("Main: The main thread has finished.\n");
}

```

工作原理

下面的截屏显示了范例的运行结果。



```

Problems @ Javadoc Declaration Console ✘
<terminated> Main (18) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe
Mock: There are 999854 occurrences of number in generated data.
Thread-0: Processing lines from 0 to 2000.
Main: The main thread has finished.
Thread-2: Processing lines from 4000 to 6000.
Thread-1: Processing lines from 2000 to 4000.
Thread-2: Lines processed.
Thread-1: Lines processed.
Thread-3: Processing lines from 6000 to 8000.
Thread-4: Processing lines from 8000 to 10000.
Thread-3: Lines processed.
Thread-4: Lines processed.
Thread-0: Lines processed.
Grouper: Processing results...
Grouper: Total result: 999854.

```

这个范例解决的问题很简单。我们有一个很大的随机数矩阵，想要知道这个矩阵中包含了指定数据的个数。为了获得更好的性能，我们使用了分治编程技术——将矩阵分离成5个子集，并且在每个子集中使用线程进行查找，这些线程是查找类 **Searcher** 对象。

我们使用 **CyclicBarrier** 对象同步5个线程，执行 **Grouper** 查找任务处理结果，并且计算最终的结果。

如之前提到的，**CyclicBarrier** 类有一个内部计数器，可以控制指定数目的几个线程必须都到达集合点。每一个线程到达集合点后就会调用 **await()** 方法通知 **CyclicBarrier** 对象，**CyclicBarrier** 对象会让这个线程休眠直到其他所有的线程都到达集合点。

当所有线程都到达集合点之后，**CyclicBarrier** 对象就唤醒所有在 **await()** 方法里等待的

线程，同时，还可以以构造器传入的 **Runnable** 对象（范例中的 **Grouper** 对象）创建一个新的线程，以执行其他任务。

更多信息

CyclicBarrier 类还提供了另一种 **await()** 方法：

await(long time, TimeUnit unit)。这个方法被调用后，线程将一直休眠到被中断，或者 **CyclicBarrier** 的内部计数器到达 0，或者指定的时间已经过期。第二个参数是 **TimeUnit** 类型，它是一个常量枚举类型，它的值包含：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

CyclicBarrier 类还提供了 **getNumberWaiting()** 方法和 **getParties()** 方法，前者将返回在 **await()** 上阻塞的线程的数目，后者返回被 **CyclicBarrier** 对象同步的任务数。

重置 CyclicBarrier 对象

虽然 **CyclicBarrier** 类和 **CountDownLatch** 类有很多共性，但是它们也有一定的差异。其中最重要的不同是，**CyclicBarrier** 对象可以被重置回初始状态，并把它的内部计数器重置成初始化时的值。

CyclicBarrier 对象的重置，是通过 **CyclicBarrier** 类提供的 **reset()** 方法完成的。当重置发生后，在 **await()** 方法中等待的线程将收到一个 **BrokenBarrierException** 异常。本例是将这个异常打印出来，但是在更复杂的应用程序中，它可以用来执行其他的操作，比如重新执行或者将操作复原回被中断时的状态。

损坏的 CyclicBarrier 对象

CyclicBarrier 对象有一种特殊的状态即损坏状态（**Broken**）。当很多线程在 **await()** 方法上等待的时候，如果其中一个线程被中断，这个线程将抛出 **InterruptedException** 异常，其他的等待线程将抛出 **BrokenBarrierException** 异常，于是 **CyclicBarrier** 对象就处于损坏状态了。

CyclicBarrier 类提供了 **isBroken()** 方法，如果处于损坏状态就返回 **true**，否则返回 **false**。

参见

- ◆ 参见 3.4 节。

3.6 并发阶段任务的运行

Java 并发 API 还提供了一个更复杂、更强大的同步辅助类，即 **Phaser**，它允许执行并发多阶段任务。当我们有并发任务并且需要分解成几步执行时，这种机制就非常适用。**Phaser** 类机制是在每一步结束的位置对线程进行同步，当所有的线程都完成了这一步，才允许执行下一步。

跟其他同步工具一样，必须对 **Phaser** 类中参与同步操作的任务数进行初始化，不同的是，我们可以动态地增加或者减少任务数。

在本节中，我们将学习如何使用 **Phaser** 类同步三个并发任务。这三个任务将在三个不同的文件夹及其子文件夹中查找过去 24 小时内修改过扩展名为.log 的文件。这个任务分成以下三个步骤：

1. 在指定的文件夹及其子文件夹中获得扩展名为.log 的文件；
2. 对第一步的结果进行过滤，删除修改时间超过 24 小时的文件；
3. 将结果打印到控制台。

在第一步和第二步结束的时候，都会检查所查找到的结果列表是不是有元素存在。如果结果列表是空的，对应的线程将结束执行，并且从 **phaser** 中删除。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建文件查找类 **FileSearch**，并且实现 **Runnable** 接口。它将在一个文件夹及其子文件夹中查找过去 24 小时内修改过的指定扩展名的文件。

```
public class FileSearch implements Runnable {
```

2. 声明私有 **String** 属性，用来存储查找的文件夹。

```
private String initPath;
```

3. 声明私有 **String** 属性，用来存储要查找的文件的扩展名。

```
private String end;
```

4. 声明私有列表 **List** 属性，用来存储查找到的文件的完整路径。

```
private List<String> results;
```

5. 声明一个私有 **Phaser** 属性，用来控制任务不同阶段的同步。

```
private Phaser phaser;
```

6. 实现构造器，用来初始化之前声明的属性。它接收查找文件夹的完整路径，文件的扩展名和 **Phaser** 同步类。

```
public FileSearch(String initPath, String end, Phaser phaser) {
    this.initPath = initPath;
    this.end = end;
    this.phaser=phaser;
    results=new ArrayList<>();
}
```

7. 实现辅助的方法，它们将用于 **run()** 方法中。第一个方法是 **directoryProcess()** 方法，它的传入参数是文件类 **File** 对象，并且它将处理文件夹的所有文件和子文件夹。对于每一个文件夹，这个方法将递归调用。对于每个文件，这个方法将调用 **fileProcess()** 方法。

```
private void directoryProcess(File file) {
    File list[] = file.listFiles();
    if (list != null) {
        for (int i = 0; i < list.length; i++) {
            if (list[i].isDirectory()) {
                directoryProcess(list[i]);
            } else {
                fileProcess(list[i]);
            }
        }
    }
}
```

8. 实现 **fileProcess()**方法。它的传入参数是文件类 **File** 对象，用于检查这个传入的文件的扩展名是不是我们指定的。如果是，文件的绝对路径将被加入到结果集中。

```
private void fileProcess(File file) {
    if (file.getName().endsWith(end)) {
        results.add(file.getAbsolutePath());
    }
}
```

9. 实现 **filterResults()**方法。它没有传入参数，而是对第一个阶段查找到的文件列表进行过滤，将不是过去 24 小时修改过的文件删除。这里创建一个空的列表，接着获得当前日期。

```
private void filterResults() {
    List<String> newResults=new ArrayList<>();
    long actualDate=new Date().getTime();
```

10. 遍历结果列表中的每个元素，对每个元素用文件路径创建文件类 **File** 对象，并获得最后修改的时间。

```
for (int i=0; i<results.size(); i++){
    File file=new File(results.get(i));
    long fileDate=file.lastModified();
```

11. 比较修改时间和当前时间。如果间隔小于一天，那么文件的完整路径将被添加到新创建的列表中。

```
if (actualDate-fileDate< TimeUnit.MILLISECONDS.
    convert(1,TimeUnit.DAYS)){
    newResults.add(results.get(i));
}
}
```

12. 将新列表引用赋给老列表。

```
results=newResults;
}
```

13. 实现 **checkResults()**方法。它将在第一个阶段和第二个阶段结束的时候被调用，用来检查结果集是不是空的。这个方法没有传入任何参数。

```
private boolean checkResults() {
```

14. 检查结果列表的长度。如果是 0，将没有找到任何文件的信息打印到控制台，并且调用 **Phaser** 对象的 **arriveAndDeregister()**方法，来通知 **Phaser** 对象当前线程已经结束这个阶段，并且将不再参与接下来的阶段操作。

```
    if (results.isEmpty()) {
        System.out.printf("%s: Phase %d: 0 results.\n", Thread.
            currentThread().getName(), phaser.getPhase());
        System.out.printf("%s: Phase %d: End.\n", Thread.
            currentThread().getName(), phaser.getPhase());
        phaser.arriveAndDeregister();
        return false;
    }
```

15. 如果结果集不是空的，则将查找的文件数打印到控制台，并且调用 **Phaser** 对象的 **arriveAndAwaitAdvance()**方法，来通知 **Phaser** 对象当前线程已经完成了当前阶段，需要被阻塞直到其他线程也都完成当前阶段。

```
} else {
    System.out.printf("%s: Phase %d: %d results.\n", Thread.
currentThread().getName(), phaser.getPhase(), results.size());
    phaser.arriveAndAwaitAdvance();
    return true;
}
```

16. 另一个辅助方法是 **showInfo()**方法，它将结果集元素打印到控制台。

```
private void showInfo() {
    for (int i=0; i<results.size(); i++){
        File file=new File(results.get(i));
        System.out.printf("%s: %s\n", Thread.currentThread().
            getName(), file.getAbsolutePath());
    }
    phaser.arriveAndAwaitAdvance();
}
```

17. 实现 **run()**方法。它将使用上述实现的辅助方法操作，并且使用 **Phaser** 对象控制阶段的改变。首先，调用 **Phaser** 对象的 **arriveAndAwaitAdvance()**方法，使查找工作在所有线程都被创建之后再开始。

```
@Override  
public void run() {  
    phaser.arriveAndAwaitAdvance();
```

18. 将查找任务开始执行的信息打印到控制台。

```
System.out.printf("%s: Starting.\n", Thread.currentThread().  
    getName());
```

19. 在 **initPath** 属性存储的文件夹中，使用 **directoryProcess()** 方法查找这个文件夹及其子文件夹中指定扩展名的文件。

```
File file = new File(initPath);  
if (file.isDirectory()) {  
    directoryProcess(file);  
}
```

20. 使用 **checkResults()** 方法检查结果集是不是空的，如果是，结束对应线程，并使用关键字 **return** 返回。

```
if (!checkResults()) {  
    return;  
}
```

21. 使用 **filterResults()** 方法过滤结果集。

```
filterResults();
```

22. 使用 **checkResults()** 方法再次检查新的结果集是不是空的，如果是，对应线程将结束，并使用关键字 **return** 返回。

```
if (!checkResults()) {  
    return;  
}
```

23. 使用 **showInfo()** 方法将最终的结果集打印到控制台，并且撤销线程的注册，然后将线程完成的信息打印到控制台。

```
showInfo();  
phaser.arriveAndDeregister();  
System.out.printf("%s: Work completed.\n", Thread.
```

```
currentThread().getName());
```

24. 实现范例的主类 **Main** 并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

25. 创建一个 **Phaser** 对象，并指定参与阶段同步的线程是 3 个。

```
Phaser phaser=new Phaser(3);
```

26. 创建 3 个文件查找类 **FileSearch** 对象，为每一个对象指定不同的查找目录，并且指定查找的是扩展名为**.log** 的文件。

```
FileSearch system=new FileSearch("C:\\Windows", "log",
    phaser);
FileSearch apps=
    new FileSearch("C:\\Program Files","log",phaser);
FileSearch documents=
    new FileSearch("C:\\Documents And Settings","log",phaser);
```

27. 将第一个 **FileSearch** 文件查找类对象作为传入参数创建线程，并且启动它。

```
Thread systemThread=new Thread(system,"System");
systemThread.start();
```

28. 将第二个 **FileSearch** 文件查找类对象作为传入参数创建线程，并且启动它。

```
Thread appsThread=new Thread(apps,"Apps");
appsThread.start();
```

29. 将第三个 **FileSearch** 文件查找类对象作为传入参数创建线程，并且启动它。

```
Thread documentsThread=new Thread(documents, "Documents");
documentsThread.start();
```

30. 等待三个线程执行结束。

```
try {
    systemThread.join();
    appsThread.join();
    documentsThread.join();
```

```

} catch (InterruptedException e) {
    e.printStackTrace();
}

```

31. 使用 Phaser 对象的 **isFinalized()**方法，打印出 Phaser 对象是否已经终止。

```
System.out.println("Terminated: " + phaser.isTerminated());
```

工作原理

本范例开始的时候创建了 **Phaser** 对象，用于在每个阶段结束时对线程同步进行控制。**Phaser** 构造器传入了参与阶段同步的线程的个数。在这个例子中，**Phaser** 有三个参与线程。这个数字通知 **Phaser** 在唤醒所有休眠线程以进行下一个阶段之前，必须执行 **arriveAndAwaitAdvance()**方法的线程数。

在 **Phaser** 创建后，我们使用三个不同的文件查找对象创建了三个线程并且启动了它们。

备注：这个例子使用了 Windows 操作系统的路径。如果是其他操作系统，需要根据相应环境修改文件路径。

在文件查找类 **FileSearch** 的 **run()** 方法中，第一个指令是调用 **Phaser** 对象的 **arriveAndAwaitAdvance()**方法。如前所述，**Phaser** 知道我们要同步的线程数，当一个线程调用这个方法时，**Phaser** 对象将减 1，并且把这个线程置于休眠状态，直到所有其他线程完成这个阶段。在 **run()**方法的开头调用这个方法可以保障在所有线程创建好之前没有线程开始执行任务。（译注：即所有线程都在同一个起跑线上。）

在第一阶段和第二阶段结束的时候，检查在这个阶段中是不是生成了结果集以及结果集中是不是有元素。在第一个阶段，**checkResults()**方法调用了 **arriveAndAwaitAdvance()**方法。在第二个阶段，如果结果集是空的，对应的线程没有理由继续执行，所以返回；但是必须通知 **phaser** 对象参与同步的线程少了一个。为了达到这个目地，我们使用了 **arriveAndDeregister()**方法。这就实现了对 **phaser** 对象的通知，即这个线程已经完成了当前语句，并且不会在下一个阶段中参与，因而 **phaser** 对象在开始下一个阶段时不会等待这个线程了。

在第三阶段结束的时候，在 **showInfo()** 方法中调用了 **phaser** 对象的 **arriveAndAwaitAdvance()**方法。通过这个调用，确保三个线程都已完成。当 **showInfo()**方法执行完成之后，还调用了 **phaser** 对象的 **arriveAndDeregister()**方法。通过这个调用，撤销了 **phaser** 中线程的注册，所以当所有线程运行结束的时候，**phaser** 对象就没有参与同步

的线程了。

最后，**main()**方法等待所有三个线程完成后，调用了**phaser**对象的**isTerminated()**方法。当**phaser**对象不存在参与同步的线程时，**phaser**是终止状态的，**isTerminated()**方法将返回**true**。当取消所有线程的注册时，**phaser**对象会变成终止状态，所以，这个调用将打印到控制台的是**true**信息。

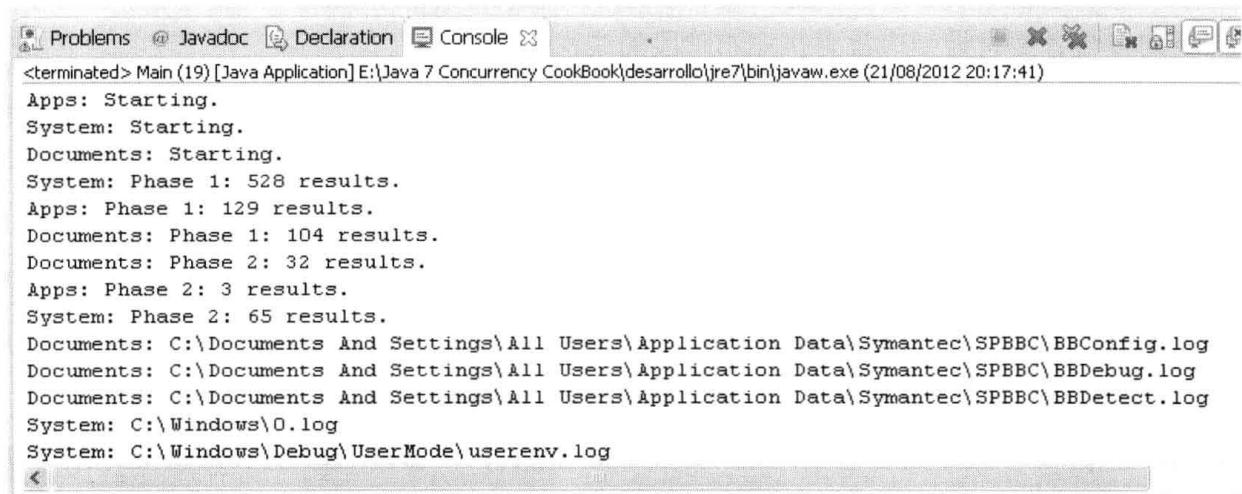
一个**Phaser**对象有两种状态。

- ◆ **活跃态（Active）**：当存在参与同步的线程的时候，**Phaser**就是活跃的，并且在每个阶段结束的时候进行同步。在这种状态中，**Phaser**的执行如前文所述。但在Java并发API中并没有提到这种状态。

- ◆ **终止态（Termination）**：当所有参与同步的线程都取消注册的时候，**Phaser**就处于终止状态，在这种状态下，**Phaser**没有任何参与者。更具体地说，当**Phaser**对象的**onAdvance()**方法返回**true**的时候，**Phaser**对象就处于了终止态。通过覆盖这个方法可以改变默认的行为。当**Phaser**是终止态的时候，同步方法**arriveAndAwaitAdvance()**会立即返回，而且不会做任何同步的操作。

Phaser类的一个重大特性就是不必对它的方法进行异常处理。不像其他的同步辅助类，被**Phaser**类置于休眠的线程不会响应中断事件，也不会抛出**InterruptedException**异常（只有一种特例会抛出异常，参见之后的“更多信息”部分）。

下面的截屏显示了范例运行的结果。



```

Problems @ Javadoc Declaration Console <terminated> Main (19) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (21/08/2012 20:17:41)
Apps: Starting.
System: Starting.
Documents: Starting.
System: Phase 1: 528 results.
Apps: Phase 1: 129 results.
Documents: Phase 1: 104 results.
Documents: Phase 2: 32 results.
Apps: Phase 2: 3 results.
System: Phase 2: 65 results.
Documents: C:\Documents And Settings\All Users\Application Data\Symantec\SPBBC\BBConfig.log
Documents: C:\Documents And Settings\All Users\Application Data\Symantec\SPBBC\BBDebug.log
Documents: C:\Documents And Settings\All Users\Application Data\Symantec\SPBBC\BBDetect.log
System: C:\Windows\0.log
System: C:\Windows\Debug\UserMode\userenv.log

```

它显示了前两个阶段的执行。我们可以看到**Apps**线程在第二个阶段结束了运行，因为查找的结果集是空的。运行本范例的时候，你将看到一部分完成当前阶段的线程，会等

待到所有线程都完成了这个阶段。

更多信息

Phaser 类提供了一些其他改变 **Phaser** 对象的方法，这些方法如下。

◆ **arrive()**: 这个方法通知 **phaser** 对象一个参与者已经完成了当前阶段，但是它不应该等待其他参与者都完成当前阶段。必须小心使用这个方法，因为它不会与其他线程同步。

◆ **awaitAdvance(int phase)**: 如果传入的阶段参数与当前阶段一致，这个方法会将当前线程置于休眠，直到这个阶段的所有参与者都运行完成。如果传入的阶段参数与当前阶段不一致，这个方法将立即返回。

◆ **awaitAdvanceInterruptibly(int phaser)**: 这个方法跟 **awaitAdvance(int phase)** 一样，不同之处是，如果在这个方法中休眠的线程被中断，它将抛出 **InterruptedException** 异常。

将参与者注册到 Phaser 中

创建一个 **Phaser** 对象时，需要指出有多少个参与者。**Phaser** 类提供了两种方法增加注册者的数量，这些方法如下。

◆ **register()**: 这个方法将一个新的参与者注册到 **Phaser** 中，这个新的参与者将被当成没有执完本阶段的线程。

◆ **bulkRegister(int Parties)**: 这个方法将指定数目的参与者注册到 **Phaser** 中，所有这些新的参与者都将被当成没有执完本阶段的线程。

Phaser 类只提供了一种方法减少注册者的数目，即 **arriveAndDeregister()** 方法。它通知 **phaser** 对象对应的线程已经完成了当前阶段，并且它不会参与到下一个阶段的操作中。

强制终止 Phaser

当一个 **phaser** 对象没有参与线程的时候，它就处于终止状态。**Phaser** 类提供了 **forceTermination()** 方法来强制 **phaser** 进入终止态，这个方法不管 **phaser** 中是否存在注册的参与线程。当一个参与线程产生错误的时候，强制 **phaser** 终止是很有意义的。

当一个 **phaser** 处于终止态的时候，**awaitAdvance()** 和 **arriveAndAwaitAdvance()** 方法立即返回一个负数，而不再是一个正值了。如果知道 **phaser** 可能会被终止，就需要验证这些方法的返回值，以确定 **phaser** 是不是被终止了。

参见

- ◆ 参见 8.3 节。

3.7 并发阶段任务中的阶段切换

Phaser 类提供了 **onAdvance()**方法，它在 **phaser** 阶段改变的时候会被自动执行。**onAdvance()**方法需要两个 int 型的传入参数：当前的阶段数以及注册的参与者数量。它返回的是 boolean 值，如果返回 **false** 表示 **phaser** 在继续执行，返回 **true** 表示 **phaser** 已经完成执行并且进入了终止态。

这个方法默认实现如下：如果注册的参与者数量是 0 就返回 **true**，否则就返回 **false**。但是我们可以通过继承 **Phaser** 类覆盖这个方法。一般来说，当必须在从一个阶段到另一个阶段过渡的时候执行一些操作，那么我们就得这么做。

在本节中，我们将通过范例学习如何控制 **phaser** 中的阶段改变。这个范例将实现自己的 **Phaser** 类，并且覆盖 **onAdvance()**方法在每个阶段改变的时候执行一些操作。这个范例将模拟考试，考生必须做三道试题，只有当所有学生都完成一道试题的时候，才能继续下一个。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建 **MyPhaser** 类并继承 **Phaser** 类。

```
public class MyPhaser extends Phaser {
```

2. 覆盖 **onAdvance()**方法。根据传入参数 **phase** 的值，调用不同的辅助方法。如果 **phase** 是 0 将调用 **studentsArrived()**方法。如果 **phase** 是 1，将调用 **finishFirstExercise()**方法。如果 **phase** 是 2，将调用 **finishSecondExercise()**方法。如果 **phase** 是 3，将调用 **finishExam()**方法。如果不是上述值，将返回 **true** 表明这个 **phaser** 已经终止了。

```
@Override  
protected boolean onAdvance(int phase, int registeredParties) {  
    switch (phase) {  
        case 0:  
            return studentsArrived();  
        case 1:  
            return finishFirstExercise();  
        case 2:  
            return finishSecondExercise();  
        case 3:  
            return finishExam();  
        default:  
            return true;  
    }  
}
```

3. 实现辅助方法 **studentsArrived()**。它打印两条信息到控制台，并且返回 **false** 表明 **phaser** 已经开始执行了。

```
private boolean studentsArrived() {  
    System.out.printf("Phaser: The exam are going to start. The  
                      students are ready.\n");  
    System.out.printf("Phaser: We have %d students.\n",  
                      getRegisteredParties());  
    return false;  
}
```

4. 实现辅助方法 **finishFirstExercise()**。它打印两条信息到控制台，并且返回 **false** 表明 **phaser** 在继续执行中。

```
private boolean finishFirstExercise() {  
    System.out.printf("Phaser: All the students have finished the  
                      first exercise.\n");  
    System.out.printf("Phaser: It's time for the second one.\n");  
    return false;  
}
```

5. 实现辅助方法 **finishSecondExercise()**。它打印两条信息到控制台，并且返回 **false** 表明 **phaser** 在继续执行中。

```
private boolean finishSecondExercise() {
    System.out.printf("Phaser: All the students have finished the
                      second exercise.\n");
    System.out.printf("Phaser: It's time for the third one.\n");
    return false;
}
```

6. 实现辅助方法 **finishExam()**。它打印两条信息到控制台，并且返回 **true** 表明 **phaser** 已经完成了。

```
private boolean finishExam() {
    System.out.printf("Phaser: All the students have finished the
                      exam.\n");
    System.out.printf("Phaser: Thank you for your time.\n");
    return true;
}
```

7. 创建学生类 **Student** 并且实现 **Runnable** 接口。这个类将模拟学生考试。

```
public class Student implements Runnable {
```

8. 声明 **Phaser** 对象 **phaser**。

```
private Phaser phaser;
```

9. 实现构造器，并初始化这个 **Phaser** 对象。

```
public Student(Phaser phaser) {
    this.phaser=phaser;
}
```

10. 实现 **run()** 方法，它将模拟考试的过程。

```
@Override
public void run() {
```

11. 将一个学生到达考场的信息打印到控制台，并且调用 **phaser** 对象的 **arriveAndAwaitAdvance()** 方法等待其他的线程。

```
System.out.printf("%s: Has arrived to do the exam.
                  %s\n", Thread.currentThread().getName(), new Date());
phaser.arriveAndAwaitAdvance();
```

12. 打印当前学生正在做考题的信息到控制台。并且调用 **doExercise1()** 方法模拟做第一道考题的过程，接着打印当前学生做完第一道题的信息到控制台，然后调用 **phaser** 的 **arriveAndAwaitAdvance()** 方法等待其他学生完成第一道考题。

```
System.out.printf("%s: Is going to do the first exercise.
    %s\n", Thread.currentThread().getName(), new Date());
doExercise1();
System.out.printf("%s: Has done the first exercise.
    %s\n", Thread.currentThread().getName(), new Date());
phaser.arriveAndAwaitAdvance();
```

13. 为第二道题和第三道题实现同样的代码。

```
System.out.printf("%s: Is going to do the second exercise.
    %s\n", Thread.currentThread().getName(), new Date());
doExercise2();
System.out.printf("%s: Has done the second exercise.
    %s\n", Thread.currentThread().getName(), new Date());
phaser.arriveAndAwaitAdvance();
System.out.printf("%s: Is going to do the third exercise.
    %s\n", Thread.currentThread().getName(), new Date());
doExercise3();
System.out.printf("%s: Has finished the exam. %s\n", Thread.
    currentThread().getName(), new Date());
phaser.arriveAndAwaitAdvance();
```

14. 实现辅助方法 **doExercise1()**，它将线程休眠一段时间。

```
private void doExercise1() {
    try {
        long duration=(long) (Math.random()*10);
        TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

15. 实现辅助方法 **doExercise2()**，它将线程休眠一段时间。

```
private void doExercise2() {
    try {
        long duration=(long) (Math.random()*10);
```

```
        TimeUnit.SECONDS.sleep(duration);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

16. 实现辅助方法 `doExercise3()`, 它将线程置于休眠一段时间。

```
private void doExercise3() {  
    try {  
        long duration=(long)(Math.random()*10);  
        TimeUnit.SECONDS.sleep(duration);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

17. 实现范例的主类 **Main**，并实现 **main()**方法。

```
public class Main {  
    public static void main(String[] args) {
```

- ## 18. 创建 MyPhaser 对象。

```
MyPhaser phaser=new MyPhaser();
```

19. 创建 5 个学生类 **Student** 对象，并且通过 **register()**方法将他们注册到 **phaser** 对象中。

```
Student students[]=new Student[5];
for (int i=0; i<students.length; i++) {
    students[i]=new Student(phaser);
    phaser.register();
}
```

20. 将创建的 **Student** 对象逐个作为传入参数创建线程，并且启动它们。

```
Thread threads[]=new Thread[students.length];
for (int i=0; i<students.length; i++){
    threads[i]=new Thread(students[i],"Student "+i);
    threads[i].start();
}
```

21. 等待 5 个线程的完成。

```
for (int i=0; i<threads.length; i++) {
    try {
        threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

22. 通过调用 **isTerminated()** 方法，将 **phaser** 是否处于终止态的信息打印到控制台。

```
System.out.printf("Main: The phaser has finished: %s.\n",
    phaser.isTerminated());
```

工作原理

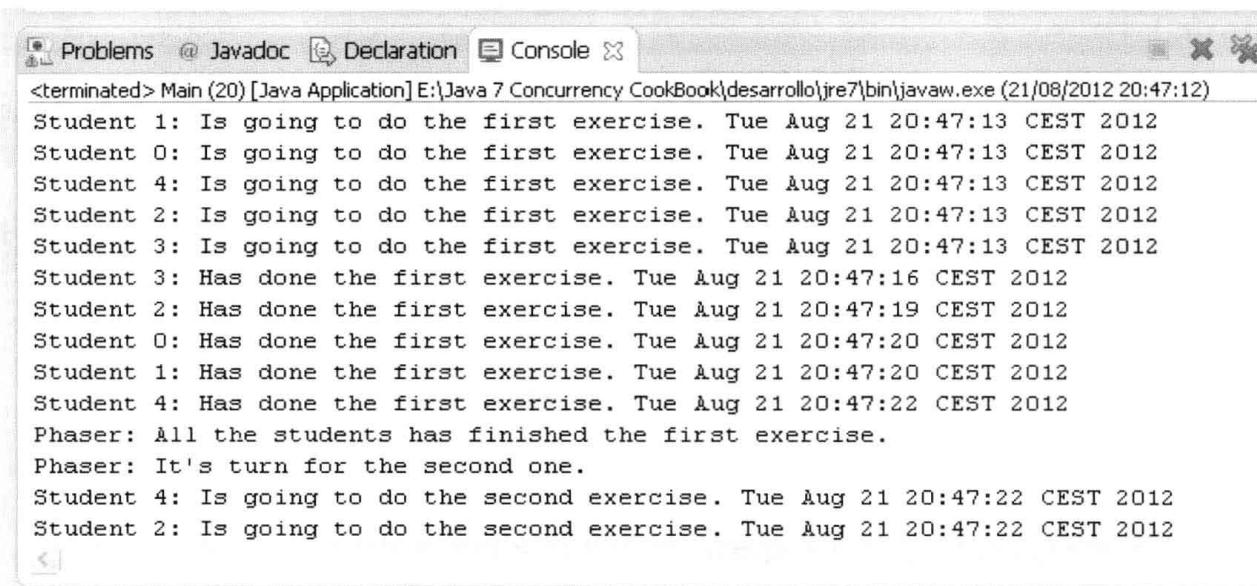
这个范例模拟了有三道试题的考试过程。所有的学生必须做完第一道题才可以开始做第二道。为了实现同步，我们使用了 **Phaser** 类，并且通过继承 **Phaser** 类和覆盖 **onAdvance()** 方法，实现了自己的 **phaser**。

phaser 对象进行阶段切换的时候，在所有在 **arriveAndAwaitAdvance()** 方法里休眠的线程被唤醒之前，**onAdvance()** 方法将被自动调用。这个方法的传入参数是当前阶段序号，其中 0 表示第一个阶段，另一个传入参数是注册的参与者。其中当前阶段序号最有用。如果要根据阶段序号执行不同的操作，那么就必须使用一个可选择的结构（**if/else** 或者 **switch**）来选择要执行的操作。在本范例中，我们使用 **switch** 结构为每个阶段来选择不同的方法。

onAdvance() 方法返回布尔值以表明 **phaser** 终止与否，**false** 表示没有终止，因而线程可以继续执行其他的阶段。如果返回值是 **true**，则 **phaser** 仍然唤醒等待的线程，但是状态已经改变成终止状态，所以继续调用 **phaser** 的方法将立即返回，并且 **isTerminated()** 方法也将返回 **true**。

在主类中，创建 **MyPhaser** 对象时，并没有指定 **phaser** 的参与者数目，但是每个学生对象都调用了 **phaser** 的 **register()** 方法，这将在 **phaser** 中注册。这个调用并没有建立学生对象或者它对应的执行线程与 **phaser** 之间的关联。实际上，**phaser** 中的参与者数目只是一个数字，**phaser** 与参与者不存在任何关联。

下面截屏是范例的运行结果。



```

Problems @ Javadoc Declaration Console 
<terminated> Main (20) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (21/08/2012 20:47:12)
Student 1: Is going to do the first exercise. Tue Aug 21 20:47:13 CEST 2012
Student 0: Is going to do the first exercise. Tue Aug 21 20:47:13 CEST 2012
Student 4: Is going to do the first exercise. Tue Aug 21 20:47:13 CEST 2012
Student 2: Is going to do the first exercise. Tue Aug 21 20:47:13 CEST 2012
Student 3: Is going to do the first exercise. Tue Aug 21 20:47:13 CEST 2012
Student 3: Has done the first exercise. Tue Aug 21 20:47:16 CEST 2012
Student 2: Has done the first exercise. Tue Aug 21 20:47:19 CEST 2012
Student 0: Has done the first exercise. Tue Aug 21 20:47:20 CEST 2012
Student 1: Has done the first exercise. Tue Aug 21 20:47:20 CEST 2012
Student 4: Has done the first exercise. Tue Aug 21 20:47:22 CEST 2012
Phaser: All the students has finished the first exercise.
Phaser: It's turn for the second one.
Student 4: Is going to do the second exercise. Tue Aug 21 20:47:22 CEST 2012
Student 2: Is going to do the second exercise. Tue Aug 21 20:47:22 CEST 2012

```

我们看到，学生在不同时间完成了第一道试题，当所有学生都完成第一道试题的时候，**phaser** 对象调用了 **onAdvance()** 方法将信息打印到控制台，然后所有学生同时开始做第二道题。

参见

- ◆ 参见 3.6 节。
- ◆ 参见 8.3 节。

3.8 并发任务间的数据交换

Java 并发 API 还提供了一个同步辅助类，它就是 **Exchanger**，它允许在并发任务之间交换数据。具体来说，**Exchanger** 类允许在两个线程之间定义同步点（Synchronization Point）。当两个线程都到达同步点时，它们交换数据结构，因此第一个线程的数据结构进入到第二个线程中，同时第二个线程的数据结构进入到第一个线程中。

Exchanger 类在生产者-消费者问题情境中很有用。这是一个经典的并发场景，包含一个数据缓冲区，一个或者多个数据生产者，一个或者多个数据消费者。**Exchanger** 类只能同步两个线程，如果有类似的问题，就可以使用 **Exchanger** 类。

在本节中，我们将学习如何使用 **Exchanger** 类来解决一对一的生产者-消费者问题。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现生产者类 **Producer**，并且实现 **Runnable** 接口。

```
public class Producer implements Runnable {
```

2. 声明 **List<String>**型对象 **buffer**，它是生产者和消费者进行交换的数据结构。

```
private List<String> buffer;
```

3. 声明 **Exchanger<List<String>>**型对象 **exchanger**，用于同步生产者和消费者的交换对象。

```
private final Exchanger<List<String>> exchanger;
```

4. 实现构造器并初始化这两个属性。

```
public Producer (List<String> buffer, Exchanger<List<String>>
    exchanger) {
    this.buffer=buffer;
    this.exchanger=exchanger;
}
```

5. 实现 **run()**方法。它循环执行 10 次数据交换。

```
@Override
public void run() {
    int cycle=1;
    for (int i=0; i<10; i++) {
        System.out.printf("Producer: Cycle %d\n",cycle);
```

6. 在每个循环中，添加 10 个字符串到 **buffer** 列表中。

```
        for (int j=0; j<10; j++) {
            String message="Event "+((i*10)+j);
```

```

        System.out.printf("Producer: %s\n", message);
        buffer.add(message);
    }
}

```

7. 调用 **exchange()**方法与消费者进行数据交换。由于这个方法会抛出 **InterruptedException** 异常，必须捕获并处理这个异常。

```

try {
    buffer=exchanger.exchange(buffer);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Producer: "+buffer.size());
cycle++;
}

```

8. 实现消费者类 **Consumer**，并实现 **Runnable** 接口。

```
public class Consumer implements Runnable {
```

9. 声明 **List<String>**型对象 **buffer**，它是生产者和消费者进行交换的数据结构。

```
private List<String> buffer;
```

10. 声明 **Exchanger<List<String>>**型对象 **exchanger**，用于同步生产者和消费者的交换对象。

```
private final Exchanger<List<String>> exchanger;
```

11. 实现构造器并初始化这两个属性。

```

public Consumer(List<String> buffer, Exchanger<List<String>>
    exchanger) {
    this.buffer=buffer;
    this.exchanger=exchanger;
}

```

12. 实现 **run()**方法，它循环执行 10 次数据交换。

```

@Override
public void run() {
    int cycle=1;
}

```

```
for (int i=0; i<10; i++) {
    System.out.printf("Consumer: Cycle %d\n", cycle);
```

13. 在每个循环中，消费者要消费数据，所以先调用 **exchange()** 方法与生产者同步。由于这个方法会抛出 **InterruptedException** 异常，必须捕获并处理这个异常。

```
try {
    buffer=exchanger.exchange(buffer);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

14. 将生产者放入消费者 **buffer** 列表中的 10 个字符串打印到控制台，并且从 **buffer** 列表中删除，保持为一个空的列表。

```
System.out.println("Consumer: "+buffer.size());
for (int j=0; j<10; j++) {
    String message=buffer.get(0);
    System.out.println("Consumer: "+message);
    buffer.remove(0);
}
cycle++;
```

15. 实现范例的主程序 **Main** 并实现 **main()** 方法。

```
public class core {
    public static void main(String[] args) {
```

16. 创建两个 **buffer** 列表，一个用于生产者，另一个用于消费者。

```
List<String> buffer1=new ArrayList<>();
List<String> buffer2=new ArrayList<>();
```

17. 创建 **Exchanger** 对象，它用来同步生产者和消费者。

```
Exchanger<List<String>> exchanger=new Exchanger<>();
```

18. 创建生产者 **Producer** 对象和消费者 **Consumer** 对象。

```
Producer producer=new Producer(buffer1, exchanger);
Consumer consumer=new Consumer(buffer2, exchanger);
```

19. 分别将生产者和消费者作为传入参数创建线程，并且启动线程。

```
Thread threadProducer=new Thread(producer);
Thread threadConsumer=new Thread(consumer);
threadProducer.start();
threadConsumer.start();
```

工作原理

消费者先创建一个空的缓存列表，然后通过调用 **Exchanger** 与生产者同步来获得可以消费的数据。生产者从一个空的缓存列表开始执行，它创建了 10 个字符串，然后存储在这个缓存中，并且使用 **exchanger** 对象与消费者同步。

在这个同步点上，两个线程（生产者和消费者）都在 **Exchanger** 里，它们交换数据结构，当消费者从 **exchange()** 方法返回的时候，它的缓存列表有 10 个字符串。当生产者从 **exchange()** 返回的时候，它的缓存列表是空的。这个操作将循环执行 10 次。

运行这个范例，我们可以看到生产者和消费者是怎样并发工作的以及两个对象是怎样在每一步中交换数据的。如果使用其他的同步辅助类，第一个线程调用 **exchange()** 后会被置于休眠，直到其他的线程到达。

更多信息

Exchanger 类还提供了另外的 **exchange** 方法，即 **exchange (V data, long time, TimeUnit unit)** 方法。其中第一个传入参数的类型是 V，即要交换的数据结构（本例中是 `List<String>`）。这个方法被调用后，线程将休眠直到被中断，或者其他线程到达，或者已耗费了指定的 time 值。第三个传入参数的类型是 `TimeUnit`，它是枚举类型，其值包含以下常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

第 4 章

线程执行器

本章内容包含：

- ◆ 创建线程执行器
- ◆ 创建固定大小的线程执行器
- ◆ 在执行器中执行任务并返回结果
- ◆ 运行多个任务并处理第一个结果
- ◆ 运行多个任务并处理所有结果
- ◆ 在执行器中延时执行任务
- ◆ 在执行器中周期性执行任务
- ◆ 在执行器中取消任务
- ◆ 在执行器中控制任务的完成
- ◆ 在执行器中分离任务的启动与结果的处理
- ◆ 处理在执行器中被拒绝的任务

4.1 简介

通常，使用 Java 来开发一个简单的并发应用程序时，会创建一些 **Runnable** 对象，然后创建对应的 **Thread** 对象来执行它们。但是，如果需要开发一个程序来运行大量的并发任务，这个方法将突显以下劣势：

- ◆ 必须实现所有与 **Thread** 对象管理相关的代码，比如线程的创建、结束以及结果获取；
- ◆ 需要为每一个任务创建一个 **Thread** 对象。如果需要执行大量的任务，这将大大地影响应用程序的处理能力；
- ◆ 计算机的资源需要高效地进行控制和管理，如果创建过多的线程，将会导致系统负荷过重。

自从 Java 5 开始，Java 并发 API 提供了一套意在解决这些问题的机制。这套机制称之为 **执行器框架（Executor Framework）**，围绕着 **Executor** 接口和它的子接口 **ExecutorService**，以及实现这两个接口的 **ThreadPoolExecutor** 类展开。

这套机制分离了任务的创建和执行。通过使用执行器，仅需要实现 **Runnable** 接口的对象，然后将这些对象发送给执行器即可。执行器通过创建所需的线程，来负责这些 **Runnable** 对象的创建、实例化以及运行。但是执行器功能不限于此，它使用了线程池来提高应用程序的性能。当发送一个任务给执行器时，执行器会尝试使用线程池中的线程来执行这个任务，避免了不断地创建和销毁线程而导致系统性能下降。

执行器框架另一个重要的优势是 **Callable** 接口。它类似于 **Runnable** 接口，但是却提供了两方面的增强。

- ◆ 这个接口的主方法名称为 **call()**，可以返回结果。
- ◆ 当发送一个 **Callable** 对象给执行器时，将获得一个实现了 **Future** 接口的对象。可以使用这个对象来控制 **Callable** 对象的状态和结果。

本章接下来将使用上述由 Java 并发 API 提供的类及其变体来展示如何使用执行器框架。

4.2 创建线程执行器

使用 **执行器框架（Executor Framework）** 的第一步是创建 **ThreadPoolExecutor** 对象。可以 **ThreadPoolExecutor** 类提供的四个构造器或者使用 **Executors** 工厂类来创建 **ThreadPoolExecutor** 对象。一旦有了执行器，就可以将 **Runnable** 或 **Callable** 对象发送给它去执行了。

在本节，我们将学习如何使用两种操作来实现一个范例，这个范例将模拟一个 Web 服务器来应对来自不同客户端的请求。

准备工作

请先行阅读 1.2 节来学习用 Java 创建线程的基本机制。然后比较这两种机制，并根据不同的问题来选择最佳的一种。

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现将被 Web 服务器执行的任务。创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

2. 声明一个名为 **initDate** 的私有 **Date** 属性，用来存储任务的创建时间，然后创建一个名为 **name** 的私有 **String** 属性，用来存储任务的名称。

```
private Date initDate;
private String name;
```

3. 实现类的构造器，用来初始化这两个属性。

```
public Task(String name) {
    initDate=new Date();
    this.name=name;
}
```

4. 实现 **run()** 方法。

```
@Override
public void run() {
```

5. 在控制台上输出 **initDate** 属性和实际时间，即任务的开始时间。

```
System.out.printf("%s: Task %s: Created on: %s\n", Thread.
    currentThread().getName(), name, initDate);
System.out.printf("%s: Task %s: Started on: %s\n", Thread.
    currentThread().getName(), name, new Date());
```

6. 将任务休眠一段随机时间。

```
try {
    Long duration=(long)(Math.random()*10);
    System.out.printf("%s: Task %s: Doing a task during %d
seconds\n",Thread.currentThread().getName(),name,duration);
    TimeUnit.SECONDS.sleep(duration);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

7. 在控制台输入任务的完成时间。

```
System.out.printf("%s: Task %s: Finished on: %s\n",Thread.
currentThread().getName(),name,new Date());
```

8. 创建一个名为 **Server** 的类，它将执行通过执行器接收到的每一个任务。

```
public class Server {
```

9. 声明一个名为 **executor** 的 **ThreadPoolExecutor** 属性。

```
private ThreadPoolExecutor executor;
```

10. 实现类的构造器，通过 **Executors** 类来初始化 **ThreadPoolExecutor** 对象。

```
public Server(){
    executor=(ThreadPoolExecutor)Executors.newCachedThreadPool();
}
```

11. 实现 **executeTask()** 方法。它接收一个 **Task** 对象作为参数，并将 **Task** 对象发
送给执行器。在控制台输出一条信息表示新的任务已经到达。

```
public void executeTask(Task task){
    System.out.printf("Server: A new task has arrived\n");
```

12. 调用执行器的 **execute()** 方法将任务发送给 **Task**。

```
executor.execute(task);
```

13. 在控制台输出一些执行器相关的数据来观察执行器的状态。

```
System.out.printf("Server: Pool Size: %d\n",executor.
```

```

        getPoolSize());
System.out.printf("Server: Active Count: %d\n", executor.
        getActiveCount());
System.out.printf("Server: Completed Tasks: %d\n", executor.
        getCompletedTaskCount());

```

14. 实现 **endServer()** 方法。在这个方法里，调用执行器的 **shutdown()** 方法来结束它的执行。

```

public void endServer() {
    executor.shutdown();
}

```

15. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```

public class Main {

    public static void main(String[] args) {
        Server server=new Server();
        for (int i=0; i<100; i++){
            Task task=new Task("Task "+i);
            server.executeTask(task);
        }
        server.endServer();
    }
}

```

工作原理

这个范例的核心在于 **Server** 类，这个类创建和使用 **ThreadPoolExecutor** 执行器来执行任务。

第一个关键点是在 **Server** 类的构造器中创建 **ThreadPoolExecutor** 对象。**ThreadPoolExecutor** 类有 4 个不同的构造器，但是，由于这些构造器在使用上的复杂性，Java 并发 API 提供 **Executors** 工厂类来构造执行器和其他相关的对象。虽然可以直接通过 **ThreadPoolExecutor** 其中之一的构造器来创建 **ThreadPoolExecutor** 对象，但是推荐使用 **Executors** 工厂类来创建它。

在这个示例中，通过使用 **Executors** 工厂类的 **newCachedThreadPool()** 方法创建了一个缓存线程池。这个方法返回一个 **ExecutorService** 对象，因此它将被强制转换为 **ThreadPoolExecutor** 类型，并拥有所有的方法。如果需要执行新任务，缓存线程池就会创

建新线程；如果线程所运行的任务执行完成后并且这个线程可用，那么缓存线程池将会重用这些线程。线程重用的优点是减少了创建新线程所花费的时间。然而，新任务固定会依赖线程来执行，因此缓存线程池也有缺点，如果发送过多的任务给执行器，系统的负荷将会过载。

备注：仅当线程的数量是合理的或者线程只会运行很短的时间时，适合采用 **Executors** 工厂类的 **newCachedThreadPool()** 方法来创建执行器。

一旦创建了执行器，就可以使用执行器的 **execute()** 方法来发送 **Runnable** 或 **Callable** 类型的任务。这个范例发送实现了 **Runnable** 接口的 **Task** 类型的对象给执行器。

范例中也打印了一些执行器相关的日志信息，专门使用了如下方法。

- ◆ **getPoolSize()**：返回执行器线程池中实际的线程数。
- ◆ **getActiveCount()**：返回执行器中正在执行任务的线程数。
- ◆ **getCompletedTaskCount()**：返回执行器已经完成的任务数。

执行器以及 **ThreadPoolExecutor** 类一个重要的特性是，通常需要显示地去结束它。如果不这样做，那么执行器将继续执行，程序也不会结束。如果执行器没有任务可执行了，它将继续等待新任务的到来，而不会结束执行。Java 应用程序不会结束直到所有非守护线程结束它们的运行，因此，如果有终止执行器，应用程序将永远不会结束。

不走心的翻译

为了完成执行器的执行，可以使用 **ThreadPoolExecutor** 类的 **shutdown()** 方法。当执行器执行完成所有待运行的任务后，它将结束执行。调用 **shutdown()** 方法之后，如果尝试再发送另一个任务给执行器，任务将被拒绝，并且执行器也将抛出 **RejectedExecutionException** 异常。

下面的截图展示了范例执行的部分结果。

```

Problems @ Javadoc Console Declaration
<terminated> Main (22) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (24/08/2012 01:43:42)
Server: Pool Size: 92
Server: Active Count: 90
Server: Completed Tasks: 8
Server: A new task has arrived
Server: Pool Size: 92
Server: Active Count: 90
Server: Completed Tasks: 8
pool-1-thread-91: Task Task 99: Created on: Fri Aug 24 01:43:43 CEST 2012
pool-1-thread-91: Task Task 99: Started on: Fri Aug 24 01:43:43 CEST 2012
pool-1-thread-92: Task Task 98: Created on: Fri Aug 24 01:43:43 CEST 2012

```

当最后一个任务到达服务器时，执行器拥有由 100 项任务和 90 个活动线程组成的池。

更多信息

ThreadPoolExecutor 类提供了许多方法来获取自身状态的信息。在范例中，已经使用了 `getPoolSize()` 方法来获取线程池的大小，用 `getActiveCount()` 方法来获取线程池中活动线程的数量，用 `getCompletedTaskCount()` 方法来获取执行器完成的任务数量。也可以使用 `getLargestPoolSize()` 方法来返回曾经同时位于线程池中的最大线程数。

ThreadPoolExecutor 类也提供了结束执行器的相关方法。

◆ `shutdownNow()`：这个方法会立即关闭执行器。执行器将不再执行那些正在等待执行的任务。这个方法将返回等待执行的任务列表。调用时，正在运行的任务将继续运行，但是这个方法并不等待这些任务完成。

◆ `isTerminated()`：如果调用了 `shutdown()` 或 `shutdownNow()` 方法，并且执行器完成了关闭的过程，那么这个方法将返回 `true`。

◆ `isShutdown()`：如果调用了 `shutdown()` 方法，那么这个方法将返回 `true`。

◆ `awaitTermination(long timeout, TimeUnit unit)`：这个方法将阻塞所调用的线程，直到执行器完成任务或者达到所指定的 `timeout` 值。

TimeUnit 是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

备注：如果想等待任务的结束，而不管任务的持续时间，可以使用一个大的超时时间，比如 **DAYS**。

参见

- ◆ 参见 4.12 节。
- ◆ 参见 8.4 节。

4.3 创建固定大小的线程执行器

当使用 **Executors** 类的 `newCachedThreadPool()` 方法创建基本的 **ThreadPoolExecutor** 时，执行器运行过程中将碰到线程数量的问题。如果线程池里没有空闲的线程可用，那么

执行器将为接收到的每一个任务创建一个新线程，当发送大量的任务给执行器并且任务需要持续较长的时间时，系统将会超负荷，应用程序也将随之性能不佳。

为了避免这个问题，**Executors** 工厂类提供了一个方法来创建一个固定大小的线程执行器。这个执行器有一个线程数的最大值，如果发送超过这个最大值的任务给执行器，执行器将不再创建额外的线程，剩下的任务将被阻塞直到执行器有空闲的线程可用。这个特性可以保证执行器不会给应用程序带来性能不佳的问题。

在本节，我们将通过修改本章 4.2 节的范例来学习如何创建固定大小的线程执行器。

准备工作

请先行阅读本章的 4.2 节，并实现其中所阐述的范例，因为本节将对其继续修改。

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 实现本章 4.2 节所描述的范例。打开 **Server** 类并修改它的构造器，使用 **newFixedThreadPool()** 方法来创建执行器，并传递数字 5 作为它的参数。

```
public Server(){  
    executor=(ThreadPoolExecutor) Executors.newFixedThreadPool(5);  
}
```

2. 修改 **executeTask()** 方法，增加一行打印日志信息。调用 **getTaskCount()** 方法来获取已发送到执行器上的任务数。

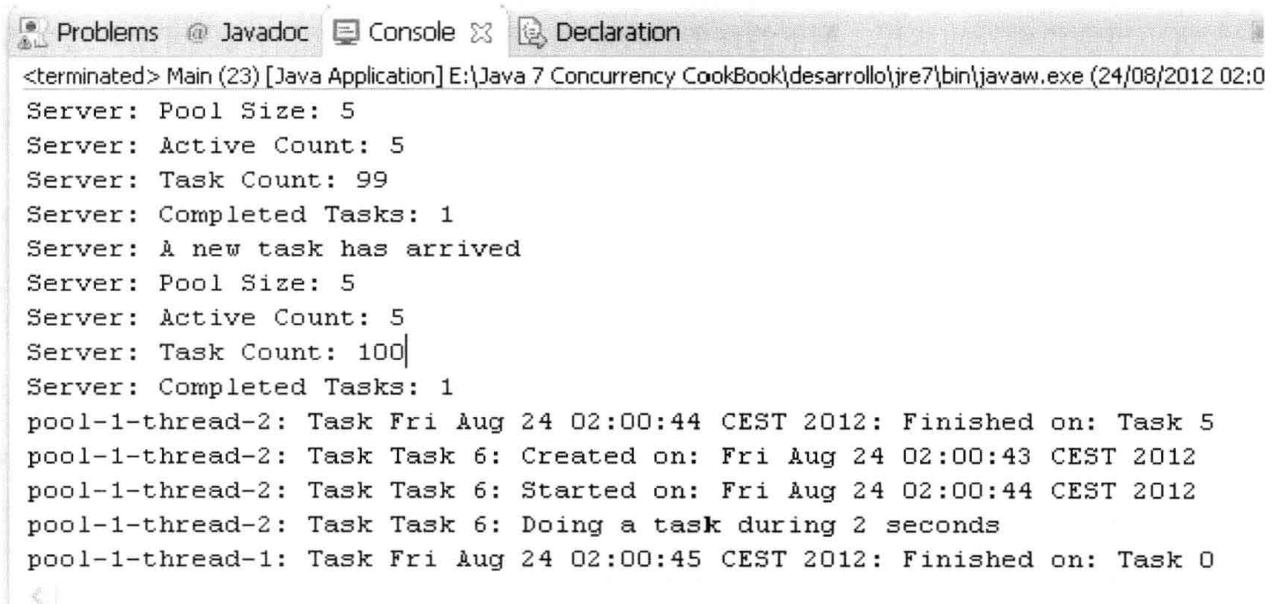
```
System.out.printf("Server: Task Count: %d\n", executor.  
    getTaskCount());
```

工作原理

在这个示例中，使用 **Executors** 工厂类的 **newFixedThreadPool()** 方法来创建执行器。这个方法创建了具有线程最大数量值的执行器。如果发送超过线程数的任务给执行器，剩余的任务将被阻塞直到线程池里有空闲的线程来处理它们。**newFixedThreadPool()** 方法接

线程执行器将拥有的线程数量的最大值作为参数。这个例子创建了一个线程数量的最大值为 5 的执行器。

下面的截图展示了范例执行的部分结果。



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```

<terminated> Main (23) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (24/08/2012 02:00)
Server: Pool Size: 5
Server: Active Count: 5
Server: Task Count: 99
Server: Completed Tasks: 1
Server: A new task has arrived
Server: Pool Size: 5
Server: Active Count: 5
Server: Task Count: 100
Server: Completed Tasks: 1
pool-1-thread-2: Task Fri Aug 24 02:00:44 CEST 2012: Finished on: Task 5
pool-1-thread-2: Task Task 6: Created on: Fri Aug 24 02:00:43 CEST 2012
pool-1-thread-2: Task Task 6: Started on: Fri Aug 24 02:00:44 CEST 2012
pool-1-thread-2: Task Task 6: Doing a task during 2 seconds
pool-1-thread-1: Task Fri Aug 24 02:00:45 CEST 2012: Finished on: Task 0

```

为了在程序中输出相关信息，已经使用的 **ThreadPoolExecutor** 类的一些方法如下。

- ◆ **getPoolSize()**: 返回执行器中线程的实际数量。
- ◆ **getActiveCount()**: 返回执行器正在执行任务的线程数量。

将看到，控制台输出的信息是 **5**，表示执行器拥有 5 个线程，并且执行器不会超过这个最大的线程连接数。

当发送最后一个任务给执行器时，由于执行器只有 5 个活动的线程，所以剩余的 95 个任务只能等待空闲线程。**getTaskCount()** 方法可以用来显示有多少个任务已经发送给执行器。

更多信息

Executors 工厂类也提供 **newSingleThreadExecutor()** 方法。这是一个创建固定大小线程执行器的极端场景，它将创建一个只有单个线程的执行器。因此，这个执行器只能在同一时间执行一个任务。

参见

- ◆ 参见 4.12 节。
- ◆ 参见 8.4 节。

4.4 在执行器中执行任务并返回结果

执行器框架（Executor Framework） 的优势之一是，可以运行并发任务并返回结果。Java 并发 API 通过以下两个接口来实现这个功能。

Callable: 这个接口声明了 `call()` 方法。可以在这个方法里实现任务的具体逻辑操作。**Callable** 接口是一个泛型接口，这就意味着必须声明 `call()` 方法返回的数据类型。

Future: 这个接口声明了一些方法来获取由 **Callable** 对象产生的结果，并管理它们的状态。

在本节，我们将学习如何实现任务的返回结果，并在执行器中运行任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **FactorialCalculator** 的类，并实现 **Callable** 接口，接口的泛型参数为 **Integer** 类型。

```
public class FactorialCalculator implements Callable<Integer> {
```

2. 声明一个名为 **number** 的私有 **Integer** 属性，存储任务即将用来计算的数字。

```
private Integer number;
```

3. 实现类的构造器，用来初始化类的属性。

```
public FactorialCalculator(Integer number) {
    this.number=number;
}
```

4. 实现 **call()**方法。这个方法返回 **FactorialCalculator** 类的 **number** 属性的阶乘 (Factorial)。

```
@Override
public Integer call() throws Exception {
```

5. 创建并初始化在 **call()**方法内使用的内部变量。

```
int result = 1;
```

6. 如果 **number** 值是 0 或 1，则返回 1；否则计算 **number** 的阶乘。为了演示效果，在两个乘法之间，将任务休眠 20 毫秒。

```
if ((num==0) || (num==1)) {
    result=1;
} else {
    for (int i=2; i<=number; i++) {
        result*=i;
        TimeUnit.MILLISECONDS.sleep(20);
    }
}
```

7. 在控制台输出操作的结果。

```
System.out.printf("%s: %d\n", Thread.currentThread().
    getName(), result);
```

8. 返回操作的结果。

```
return result;
```

9. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

10. 通过 **Executors** 工厂类的 **newFixedThreadPool()**方法创建 **ThreadPoolExecutor** 执行器来运行任务。传递参数 2 给 **newFixedThreadPool()**方法表示执行器将最多创建两个

线程。

```
ThreadPoolExecutor executor=(ThreadPoolExecutor) Executors.  
    newFixedThreadPool(2);
```

11. 创建一个 **Future<Integer>** 类型的列表对象 **resultList**。

```
List<Future<Integer>> resultList=new ArrayList<>();
```

12. 通过 **Random** 类创建一个名 **random** 的随机数字生成器。

```
Random random=new Random();
```

13. 生成 10 个介于 0~10 之间的随机整数。

```
for (int i=0; i<10; i++) {  
    Integer number= random.nextInt(10);
```

14. 创建 **FactorialCalculator** 对象，并将随机数 **number** 传递给它作为参数。

```
FactorialCalculator calculator=new  
    FactorialCalculator(number);
```

15. 调用执行器的 **submit()** 方法发送 **FactorialCalculator** 任务给执行器。这个方法返回一个 **Future<Integer>** 对象来管理任务和得到的最终结果。

```
Future<Integer> result=executor.submit(calculator);
```

16. 将 **Future** 对象添加到前面创建的 **resultList** 列表中。

```
resultList.add(result);  
}
```

17. 创建一个 **do** 循环来监控执行器的状态。

```
do {
```

18. 通过执行器的 **getCompletedTaskNumber()** 方法，在控制台输出信息表示任务完成的数量。

```
System.out.printf("Main: Number of Completed Tasks:  
%d\n", executor.getCompletedTaskCount());
```

19. 遍历 **resultList** 列表中的 10 个 **Future** 对象，通过调用 **isDone()** 方法来输出表示任务是否完成的信息。

```
for (int i=0; i<resultList.size(); i++) {
    Future<Integer> result=resultList.get(i);
    System.out.printf("Main: Task %d: %s\n", i, result.
        isDone());
}
```

20. 将线程休眠 50 毫秒。

```
try {
    TimeUnit.MILLISECONDS.sleep(50);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

21. 若执行器中完成的任务数量小于 10，则一直重复执行这个循环。

```
} while (executor.getCompletedTaskCount()<resultList.size());
```

22. 在控制台上输出每一个任务得到的结果。对于每一个 **Future** 对象来讲，通过调用 **get()** 方法将得到由任务返回的 **Integer** 对象。

```
System.out.printf("Main: Results\n");
for (int i=0; i<resultList.size(); i++) {
    Future<Integer> result=resultList.get(i);
    Integer number=null;
    try {
        number=result.get();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
```

23. 在控制台上打印出数字 **number**。

```
System.out.printf("Main: Task %d: %d\n", i, number);
}
```

24. 调用执行器的 **shutdown()** 方法结束执行。

```
executor.shutdown();
```

工作原理

在本节，我们学习了如何使用 **Callable** 接口来启动并发任务并返回结果。我们编写了 **FactorialCalculator** 类，它实现了带有泛型参数 **Integer** 类型的 **Callable** 接口。因此，这个 **Integer** 类型将作为在调用 **call()** 方法时返回的类型。

范例的另一个关键点在 **Main** 主类中。我们通过 **submit()** 方法发送一个 **Callable** 对象给执行器去执行，这个 **submit()** 方法接收 **Callable** 对象作为参数，并返回 **Future** 对象。**Future** 对象可以用于以下两个主要目的。

- ◆ 控制任务的状态：可以取消任务和检查任务是否已经完成。为了达到这个目的，可使用 **isDone()** 方法来检查任务是否已经完成。
- ◆ 通过 **call()** 方法获取返回的结果。为了达到这个目的，可使用 **get()** 方法。这个方法一直等待直到 **Callable** 对象的 **call()** 方法执行完成并返回结果。如果 **get()** 方法在等待结果时线程中断了，则将抛出一个 **InterruptedException** 异常。如果 **call()** 方法抛出异常那么 **get()** 方法将随之抛出 **ExecutionException** 异常。

更多信息

在调用 **Future** 对象的 **get()** 方法时，如果 **Future** 对象所控制的任务并未完成，那么这个方法将一直阻塞到任务完成。**Future** 接口也提供了 **get()** 方法的其他调用方式。

- ◆ **get(long timeout, TimeUnit unit)**: 如果调用这个方法时，任务的结果并未准备好，则方法等待所指定的 **timeout** 时间。如果等待超过了指定的时间而任务的结果还没有准备好，那么这个方法将返回 **null**。

TimeUnit 是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

参见

- ◆ 参见 4.2 节。
- ◆ 参见 4.5 节。
- ◆ 参见 4.6 节。

4.5 运行多个任务并处理第一个结果

并发编程比较常见的一个问题是，当采用多个并发任务来解决一个问题时，往往只关心这些任务中的第一个结果。比如，对一个数组进行排序有很多种算法，可以并发启动所有算法，但是对于一个给定的数组，第一个得到排序结果的算法就是最快的排序算法。

在本节，我们将学习如何使用 **ThreadPoolExecutor** 类来实现这个场景。范例允许用户可以通过两种验证机制进行验证，但是，只要有一种机制验证成功，那么这个用户就被验证通过了。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **UserValidator** 的类，它将实现用户验证的过程。

```
public class UserValidator {
```

2. 声明一个名为 **name** 的私有 **String** 属性，用来存储用户验证系统的名称。

```
    private String name;
```

3. 实现类的构造器，用来初始化类的属性。

```
    public UserValidator(String name) {  
        this.name=name;  
    }
```

4. 实现 **validate()** 方法。它接收两个 **String** 参数，分别取名为用户名 **name** 和密码 **password**，这两个参数也将被用来进行用户验证。

```
    public boolean validate(String name, String password) {
```

5. 创建一个名为 **random** 的 **Random** 类型的随机对象。

```
        Random random=new Random();
```

6. 等待一段随机时间来模拟用户验证的过程。

```
try {
    long duration=(long)(Math.random()*10);
    System.out.printf("Validator %s: Validating a user during %d
seconds\n",this.name,duration);
    TimeUnit.SECONDS.sleep(duration);
} catch (InterruptedException e) {
    return false;
}
```

7. 返回随机的 **boolean** 值。当用户通过验证时，这个方法返回 **true** 值，如果用户没有通过验证则返回 **false** 值。

```
return random.nextBoolean();
}
```

8. 实现 **getName()** 方法。这个方法返回 **name** 属性值。

```
public String getName(){
    return name;
}
```

9. 创建一个名为 **TaskValidator** 的类，它将通过 **UserValidation** 对象作为并发任务来执行用户验证的过程。这个类实现了带有 **String** 泛型参数的 **Callable** 接口。

```
public class TaskValidator implements Callable<String> {
```

10. 声明一个名为 **validator** 的私有 **UserValidator** 属性。

```
private UserValidator validator;
```

11. 声明两个私有的 **String** 属性，分别为用户名 **user** 和密码 **password**。

```
private String user;
private String password;
```

12. 实现类的构造器，用来初始化类的属性。

```
public TaskValidator(UserValidator validator, String user,
    String password){
    this.validator=validator;
    this.user=user;
    this.password=password;
```

```
}
```

13. 实现 **call()**方法，并返回 **String** 对象。

```
@Override
public String call() throws Exception {
```

14. 如果用户没有通过 **UserValidator** 对象的验证，就在控制台输出没有找到这个用户，表明该用户未通过验证，并抛出 **Exception** 类型的异常。

```
if (!validator.validate(user, password)) {
    System.out.printf("%s: The user has not been found\n",
                      validator.getName());
    throw new Exception("Error validating user");
}
```

15. 否则，就在控制台输出用户已经找到，表明该用户已经通过验证，然后返回 **UserValidator** 对象的名称。

```
System.out.printf("%s: The user has been found\n", validator.
    getName());
return validator.getName();
```

16. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

17. 创建两个 **String** 对象，分别取名为 **username** 和 **password**，并初始化这两个属性值为 **test**。

```
String username="test";
String password="test";
```

18. 创建两个 **UserValidator** 对象，分别取名为 **ldapValidator** 和 **dbValidator**。

```
UserValidator ldapValidator=new UserValidator("LDAP");
UserValidator dbValidator=new UserValidator("DataBase");
```

19. 创建两个 **TaskValidator** 对象，分别取名为 **ldapTask** 和 **dbTask**，并分别用 **ldapValidator** 和 **dbValidator** 来初始化他们。

```
TaskValidator ldapTask=new TaskValidator(ldapValidator,
    username, password);
TaskValidator dbTask=new TaskValidator(dbValidator,
    username, password);
```

20. 创建一个名为 **taskList** 的 **TaskValidator** 类型列表，并将 **ldapTask** 和 **dbTask** 添加到列表中。

```
List<TaskValidator> taskList=new ArrayList<>();
taskList.add(ldapTask);
taskList.add(dbTask);
```

21. 通过 **Executors** 工厂类的 **newCachedThreadPool()** 方法创建一个新的 **ThreadPoolExecutor** 执行器对象，并创建一个名为 **result** 的 **String** 对象。

```
ExecutorService executor=(ExecutorService)Executors.
    newCachedThreadPool();
String result;
```

22. 调用执行器的 **invokeAny()** 方法。这个方法接收 **taskList** 作为参数，并返回 **String** 对象。然后，在控制台上输出这个方法返回的 **String** 对象。

```
try {
    result = executor.invokeAny(taskList);
    System.out.printf("Main: Result: %s\n", result);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

23. 通过 **shutdown()** 方法来终止执行器，并在控制台输出信息表示程序已经执行结束。

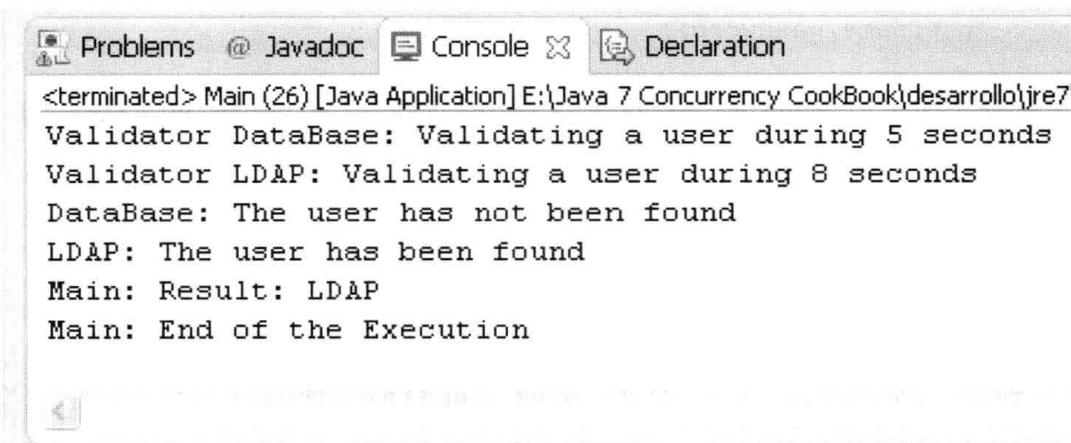
```
executor.shutdown();
System.out.printf("Main: End of the Execution\n");
```

工作原理

这个范例的关键点在 **Main** 主类中。**ThreadPoolExecutor** 类的 **invokeAny()** 方法接

收到一个任务列表，然后运行任务，并返回第一个完成任务并且没有抛出异常的任务的执行结果。这个方法返回的类型与任务里的 `call()` 方法返回的类型相同，在这个范例中，它将返回 `String` 类型值。

下面的截图展示了当范例运行后，有一个任务成功地验证了用户后的运行结果。



```

Problems @ Javadoc Console Declaration
<terminated> Main (26) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\lib\src.zip
Validator DataBase: Validating a user during 5 seconds
Validator LDAP: Validating a user during 8 seconds
DataBase: The user has not been found
LDAP: The user has been found
Main: Result: LDAP
Main: End of the Execution

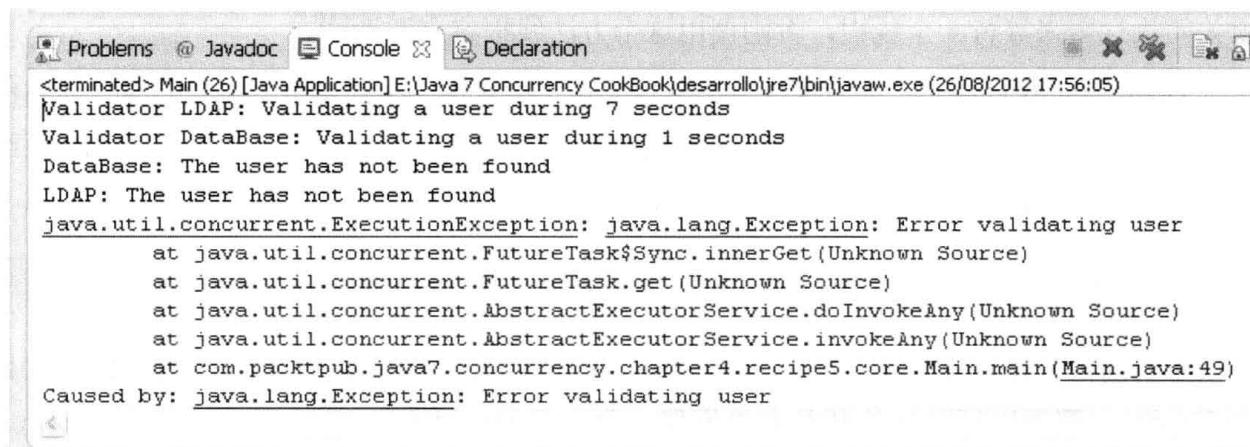
```

范例中有两个 `UserValidator` 对象，它们返回随机的 `boolean` 值。每一个 `UserValidator` 对象被 `TaskValidator` 对象使用，`TaskValidator` 对象实现了 `Callable` 接口。如果 `UserValidator` 类的 `validate()` 方法返回 `false` 值，那么 `TaskValidator` 类将抛出 `Exception` 异常。否则，返回 `true` 值。

因此，我们有两个任务可以返回 `true` 值或抛出 `Exception` 异常。从而，可以有如下 4 种可能性。

- ◆ 如果两个任务都返回 `true` 值，那么 `invokeAny()` 方法的结果就是首先完成任务的名称。
- ◆ 如果第一个任务返回 `true` 值，第二个任务抛出 `Exception` 异常，那么 `invokeAny()` 方法的结果就是第一个任务的名称。
- ◆ 如果第一个任务抛出 `Exception` 异常，第二个任务返回 `true` 值，那么 `invokeAny()` 方法的结果就是第二个任务的名称。
- ◆ 如果两个任务都抛出 `Exception` 异常，那么 `invokeAny()` 方法将抛出 `ExecutionException` 异常。

将这个范例多运行几次，那么将得到如上所述的四种可能的结果。以下截图则显示当两个任务同时抛出异常时，应用程序得到的结果。



The screenshot shows a Java application running in an IDE's terminal window. The output shows several validation attempts:

```

<terminated> Main (26) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (26/08/2012 17:56:05)
Validator LDAP: Validating a user during 7 seconds
Validator DataBase: Validating a user during 1 seconds
DataBase: The user has not been found
LDAP: The user has not been found
java.util.concurrent.ExecutionException: java.lang.Exception: Error validating user
    at java.util.concurrent.FutureTask$Sync.innerGet(Unknown Source)
    at java.util.concurrent.FutureTask.get(Unknown Source)
    at java.util.concurrent.AbstractExecutorService.invokeAny(Unknown Source)
    at java.util.concurrent.AbstractExecutorService.invokeAny(Unknown Source)
    at com.packtpub.java7.concurrency.chapter4.recipe5.core.Main.main(Main.java:49)
Caused by: java.lang.Exception: Error validating user

```

更多信息

ThreadPoolExecutor 类还提供了 **invokeAny()** 方法的其他版本：

invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit): 这个方法执行所有的任务，如果在给定的超时期满之前某个任务已经成功完成（也就是未抛出异常），则返回其结果。

TimeUnit 是一个枚举类，有如下的常量：DAYS、HOURS、MICROSECONDS、MILLISECONDS、MINUTES、NANOSECONDS 和 SECONDS。

参见

- ◆ 参见 4.6 节。

4.6 运行多个任务并处理所有结果

执行器框架（Executor Framework） 允许执行并发任务而不需要去考虑线程创建和执行。它还提供了可以用来控制在执行器中执行任务的状态和获取任务运行结果的 **Future** 类。

如果想要等待任务结束，可以使用如下两种方法。

- ◆ 如果任务执行结束，那么 **Future** 接口的 **isDone()** 方法将返回 **true**。
- ◆ 在调用 **shutdown()** 方法后，**ThreadPoolExecutor** 类的 **awaitTermination()** 方法会将线程休眠，直到所有的任务执行结束。

这两个方法有一些缺点：第一个方法，仅可以控制任务的完成与否；第二个方法，必

须关闭执行器来等待一个线程，否则调用这个方法线程将立即返回。

ThreadPoolExecutor 类还提供一个方法，它允许发送一个任务列表给执行器，并等待列表中所有任务执行完成。在本节，我们将编写范例，执行三个任务，当它们全部执行结束后打印出结果信息，用来学习如何使用这个特性。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Result** 的类，用来存储范例中并发任务产生的结果。

```
public class Result {
```

2. 声明两个私有属性。一个名为 **name** 的 **String** 属性，一个名为 **value** 的 **int** 属性。

```
    private String name;  
    private int value;
```

3. 实现对应的 **get()** 和 **set()** 方法来设置和返回 **name** 和 **value** 属性。

```
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getValue() {  
        return value;  
    }  
    public void setValue(int value) {  
        this.value = value;  
    }
```

4. 创建一个名为 **Task** 的类，并实现 **Callable** 接口，接口的泛型参数为 **Result** 类型。

```
public class Task implements Callable<Result> {
```

5. 声明一个名为 **name** 的私有 **String** 属性。

```
private String name;
```

6. 实现类的构造器，用来初始化类的属性。

```
public Task(String name) {
    this.name=name;
}
```

7. 实现 **call()** 方法。在这个范例中，这个方法将返回一个 **Result** 类型的对象。

```
@Override
public Result call() throws Exception {
```

8. 在控制台输出表示任务开始的信息。

```
System.out.printf("%s: Starting\n",this.name);
```

9. 等待一段随机时间。

```
try {
    long duration=(long)(Math.random()*10);
    System.out.printf("%s: Waiting %d seconds for results.\n",
        this.name,duration);
    TimeUnit.SECONDS.sleep(duration);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

10. 生成一个 **int** 值，准备作为返回 **Result** 对象中的 **int** 属性，这个 **int** 值为 5 个随机数的总和。

```
int value=0;
for (int i=0; i<5; i++) {
    value+=(int)(Math.random()*100);
}
```

11. 创建一个 **Result** 对象，并用任务的名称和上一步计算的 **int** 值来对其进行初始化。

```

    Result result=new Result();
    result.setName(this.name);
    result.setValue(value);
}

```

12. 在控制台输出信息表示任务执行结束。

```
System.out.println(this.name+": Ends");
```

13. 返回 **Result** 对象。

```

    return result;
}

```

14. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```

public class Main {
    public static void main(String[] args) {
}

```

15. 通过 **Executors** 工厂类的 **newCachedThreadPool()**方法创建一个 **ThreadPoolExecutor** 执行器对象。

```
ExecutorService executor=(ExecutorService) Executors.
    newCachedThreadPool();
```

16. 创建一个 **Task** 类型的任务列表 **taskList**。创建 3 个 **Task** 任务并将它们添加到任务列表 **taskList** 中。

```

List<Task> taskList=new ArrayList<>();
for (int i=0; i<3; i++){
    Task task=new Task(i);
    taskList.add(task);
}

```

17. 创建一个 **Future** 类型的结果列表 **resultList**。这些对象泛型参数为 **Result** 类型。

```
List<Future<Result>> resultList=null;
```

18. 调用 **ThreadPoolExecutor** 类的 **invokeAll()** 方法。这个方法将返回上一步所创建的 **Future** 类型的列表。

```

try {
    resultList=executor.invokeAll(taskList);
}

```

```

} catch (InterruptedException e) {
    e.printStackTrace();
}

```

19. 调用 **shutdown()**方法结束执行器。

```
executor.shutdown();
```

20. 在控制台输出任务处理的结果，即 **Future** 类型列表中的 **Result** 结果。

```

System.out.println("Main: Printing the results");
for (int i=0; i<resultList.size(); i++) {
    Future<Result> future=resultList.get(i);
    try {
        Result result=future.get();
        System.out.println(result.getName()+"："+result.
            getValue());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

```

工作原理

在本节，我们学习了如何发送任务列表给执行器，并且通过 **invokeAll()**方法等待所有任务的完成。这个方法接收一个 **Callable** 对象列表，并返回一个 **Future** 对象列表。在这个列表中，每一个任务对应一个 **Future** 对象。**Future** 对象列表中的第一个对象控制 **Callable** 列表中第一个任务，以此类推。

需要注意的一点是，在存储结果的列表声明中，用在 **Future** 接口中的泛型参数的数据类型必须与 **Callable** 接口的泛型数据类型相兼容。在这个例子中，我们使用的是相同的数据类型：**Result** 类。

另一个关于 **invokeAll()**方法重要的地方是，使用 **Future** 对象仅用来获取任务的结果。当所有的任务执行结束时这个方法也执行结束了，如果在返回的 **Future** 对象上调用 **isDone()** 方法，那么所有的调用将返回 **true** 值。

更多信息

ExecutorService 接口还提供了 **invokeAll()** 方法的另一个版本：

◆ **invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit):**

当所有任务执行完成，或者超时的时候（无论哪个首先发生），这个方法将返回保持任务状态和结果的 **Future** 列表。

TimeUnit 是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

参见

- ◆ 参见 4.4 节。
- ◆ 参见 4.5 节。

4.7 在执行器中延时执行任务

执行器框架（Executor Framework）提供了 **ThreadPoolExecutor** 类并采用线程池来执行 **Callable** 和 **Runnable** 类型的任务，采用线程池可以避免所有线程的创建操作而提高应用程序的性能。当发送一个任务给执行器时，根据执行器的相应配置，任务将尽可能快地被执行。但是，如果并不想让任务马上被执行，而是想让任务在过一段时间后才被执行，或者任务能够被周期性地执行。为了达到这个目的，执行器框架提供了 **ScheduledThreadPoolExecutor** 类。

在本节，我们将学习如何创建 **ScheduledThreadPoolExecutor** 执行器，以及如何使用它在经过一个给定的时间后开始执行任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Callable** 接口，接口的泛型参数为 **String** 类型。

```
public class Task implements Callable<String> {
```

2. 声明一个名为 **name** 的私有 **String** 属性，用来存储任务的名称。

```
private String name;
```

3. 实现类的构造器，并初始化 **name** 属性。

```
public Task(String name) {
    this.name=name;
}
```

4. 实现 **call()** 方法。在控制台输出实际的时间，并返回一个文本信息，比如“Hello, world”。

```
public String call() throws Exception {
    System.out.printf("%s: Starting at : %s\n",name,new Date());
    return "Hello, world";
}
```

5. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

6. 通过 **Executors** 工厂类的 **newScheduledThreadPool()** 方法创建一个 **ScheduledThreadPoolExecutor** 执行器，并传递 1 作为参数。

```
ScheduledThreadPoolExecutor executor=(ScheduledThreadPoolExecutor)Executors.newScheduledThreadPool(1);
```

7. 初始化一些任务（在我们的示例中是 5 个），然后通过 **ScheduledThreadPoolExecutor** 实例的 **schedule()** 方法来启动这些任务。

```
System.out.printf("Main: Starting at: %s\n",new Date());
for (int i=0; i<5; i++) {
    Task task=new Task("Task "+i);
    executor.schedule(task,i+1 , TimeUnit.SECONDS);
}
```

8. 调用执行器的 **shutdown()** 方法来结束执行器。

```
executor.shutdown();
```

9. 调用执行器的 **awaitTermination()** 方法等待所有任务结束。

```
try {
```

```

        executor.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

10. 在控制台输出信息表示程序执行结束的时间。

```
System.out.printf("Main: Ends at: %s\n", new Date());
```

工作原理

这个范例的关键点在于 **Main** 主类和 **ScheduledThreadPoolExecutor** 执行器的管理。虽然可以通过 **ThreadPoolExecutor** 类来创建定时执行器，但是在 Java 并发 API 中则推荐利用 **Executors** 工厂类来创建。在这个范例中，必须使用 **newScheduledThreadPool()** 方法，并且传递数字 1 作为方法的参数，这个参数就是线程池里拥有的线程数。

为了在定时执行器中等待一段给定的时间后执行一个任务，需要使用 **schedule()** 方法。这个方法接收如下的参数：

- ◆ 即将执行的任务；
- ◆ 任务执行前所要等待的时间；
- ◆ 等待时间的单位，由 **TimeUnit** 类的一个常量来指定。

在这个示例中，每个任务将等待 N 秒 (**TimeUnit.SECONDS**)，这个 N 值则等于任务在数组中的位置加 1。

备注：如果想在一个给定的时间点来定时执行任务，那就需要计算这个给定时间点和当前时间的差异值，然后用这个差异值作为任务的延迟值。

通过下面的截图，可以看到范例运行的部分结果。

```

Problems @ Javadoc Console Declaration
<terminated> Main (28) [Java Application] E:\Java 7 Concurrency_CookBook\desarrollc
Main: Starting at: Sun Aug 26 20:14:55 CEST 2012
Task 0: Starting at : Sun Aug 26 20:14:56 CEST 2012
Task 1: Starting at : Sun Aug 26 20:14:57 CEST 2012
Task 2: Starting at : Sun Aug 26 20:14:58 CEST 2012
Task 3: Starting at : Sun Aug 26 20:14:59 CEST 2012
Task 4: Starting at : Sun Aug 26 20:15:00 CEST 2012
Core: Ends at: Sun Aug 26 20:15:00 CEST 2012

```

从结果可知，每隔 1 秒钟就有一个任务开始执行；这是因为所有的任务被同时发送到执行器，但每个任务都比前一个任务延迟了 1 秒钟。

更多信息

也可以使用 **Runnable** 接口来实现任务，因为 **ScheduledThreadPoolExecutor** 类的 **schedule()** 方法可以同时接受这两种类型的任务。

虽然 **ScheduledThreadPoolExecutor** 类是 **ThreadPoolExecutor** 类的子类，因而继承了 **ThreadPoolExecutor** 类所有的特性。但是，Java 推荐仅在开发定时任务程序时采用 **ScheduledThreadPoolExecutor** 类。

最后，在调用 **shutdown()** 方法而仍有待处理的任务需要执行时，可以配置 **ScheduledThreadPoolExecutor** 的行为。默认的行为是不论执行器是否结束，待处理的任务仍将被执行。但是，通过调用 **ScheduledThreadPoolExecutor** 类的 **setExecuteExistingDelayedTasksAfterShutdownPolicy()** 方法则可以改变这个行为。传递 **false** 参数给这个方法，执行 **shutdown()** 方法后，待处理的任务将不会被执行。

参见

- ◆ 参见 4.4 节。

4.8 在执行器中周期性执行任务

执行器框架（Executor Framework） 提供了 **ThreadPoolExecutor** 类，通过线程池来执行并发任务从而避免了所有线程的创建操作。当发送一个任务给执行器后，根据执行器的配置，它将尽快地执行这个任务。当任务执行结束后，这个任务就会从执行器中删除；如果想再次执行这个任务，则需要再次发送这个任务到执行器。

但是，执行器框架提供了 **ScheduledThreadPoolExecutor** 类来执行周期性的任务。在本节，我们将学习如何使用这个类的功能来计划执行周期性的任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

2. 声明一个名为 **name** 的私有 **String** 属性，用来存储任务的名称。

```
private String name;
```

3. 实现类的构造器，用来初始化类的属性。

```
public Task(String name) {
    this.name=name;
}
```

4. 实现 **run()** 方法。在控制台输出实际的时间，用来检验任务将在指定的一段时间内执行。

```
@Override
public String call() throws Exception {
    System.out.printf("%s: Starting at : %s\n",name,new Date());
    return "Hello, world";
}
```

5. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

6. 通过调用 **Executors** 工厂类的 **newScheduledThreadPool()** 方法创建 **ScheduledThreadPoolExecutor** 执行器对象，传递 1 作为这个方法的参数。

```
ScheduledExecutorService executor=Executors.
    newScheduledThreadPool(1);
```

7. 在控制台输出实际时间。

```
System.out.printf("Main: Starting at: %s\n",new Date());
```

8. 创建一个新的 **Task** 对象。

```
Task task=new Task("Task");
```

9. 调用 **scheduledAtFixRate()** 方法将这个任务发送给执行器。传递给这个方法的参数分别为上一步创建的 **task** 对象、数字 **1**、数字 **2**，以及 **TimeUnit.SECONDS** 常量。这个方法返回一个用来控制任务状态的 **ScheduledFuture** 对象。

```
ScheduledFuture<?> result=executor.scheduleAtFixedRate(task,
    1, 2, TimeUnit.SECONDS);
```

10. 创建一个 10 步的循环，在控制台输出任务下一次将要执行的剩余时间。在循环体内，用 **ScheduledFuture** 类的 **getDelay()** 方法来获取任务下一次将要执行的毫秒数，然后将线程休眠 500 毫秒。

```
for (int i=0; i<10; i++) {
    System.out.printf("Main: Delay: %d\n", result.
        getDelay(TimeUnit.MILLISECONDS));
    Sleep the thread during 500 milliseconds.

    try {
        TimeUnit.MILLISECONDS.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

11. 调用 **shutdown()** 方法结束执行器。

```
executor.shutdown();
```

12. 将线程休眠 5 秒，等待周期性的任务全部执行完成。

```
try {
    TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

13. 在控制台输出信息表示程序结束。

```
System.out.printf("Main: Finished at: %s\n", new Date());
```

工作原理

想要通过执行器框架来执行一个周期性任务时，需要一个 **ScheduledExecutorService** 对象。同创建执行器一样，在 Java 中推荐使用 **Executors** 工厂类来创建 **ScheduledExecutorService** 对象。**Executors** 类就是执行器对象的工厂。在这个例子中，可以使用 **newScheduledThreadPool()** 方法来创建一个 **ScheduledExecutorService** 对象。这个方法接收一个表示线程池中的线程数来作为参数。在这个范例中，因为仅有一个任务，所以只需要传递数字 1 作为参数即可。

一旦有了可以执行周期性任务的执行器，就可以发送任务给这个执行器。在范例中，我们使用 **scheduledAtFixedRate()** 方法发送任务。这个方法接收 4 个参数，分别为将被周期性执行的任务，任务第一次执行后的延时时间，两次执行的时间周期，以及第 2 个和第 3 个参数的时间单位。这个单位是 **TimeUnit** 枚举的常量。**TimeUnit** 是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

另一个需要注意的是，两次执行之间的周期是指任务在两次执行开始时的时间间隔。如果有一个周期性的任务需要执行 5 秒钟，但是却让它每 3 秒钟执行一次，那么，在任务执行的过程中将会有两个任务实例同时存在。

scheduleAtFixedRate() 方法返回一个 **ScheduledFuture** 对象，**ScheduledFuture** 接口则扩展了 **Future** 接口，于是它带有了定时任务的相关操作方法。**ScheduledFuture** 是一个泛型参数化的接口。在这个示例中，任务是 **Runnable** 对象，并没有泛型参数化，必须通过 ? 符号作为参数来泛型化它们。

我们已经使用过 **ScheduledFuture** 接口中的一个方法。**getDelay()** 方法返回任务到下一次执行时所要等待的剩余时间。这个方法接收一个 **TimeUnit** 常量作为时间单位。

下面的截图显示了范例的部分运行结果。



```

Problems @ Javadoc Console Declaration
<terminated> Main [29] [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7
Main: Delay: 990
Main: Delay: 497
Task: Executed at: Sun Aug 26 23:38:05 CEST 2012
Main: Delay: 1997
Main: Delay: 1497
Main: Delay: 997
Main: Delay: 497
Task: Executed at: Sun Aug 26 23:38:07 CEST 2012
Main: Delay: 1997
Main: Delay: 1493
Main: Delay: 993
Main: Delay: 493
Task: Executed at: Sun Aug 26 23:38:09 CEST 2012
Main: No more tasks at: Sun Aug 26 23:38:09 CEST 2012
Main: Finished at: Sun Aug 26 23:38:14 CEST 2012

```

通过控制上面的信息，可以看到任务是每 **2** 秒执行一次；剩余的延迟时间会每隔 **500** 毫秒在控制台上输出，这个 **500** 毫秒则是主线程将被休眠的时间。当关闭执行器时，定时任务将结束执行，然后在控制台上也看不到更多的信息了。

更多信息

ScheduledThreadPoolExecutor 类还提供了其他方法来安排周期性任务的运行，比如，**scheduleWithFixedRate()** 方法。这个方法与 **scheduledAtFixedRate()** 方法具有相同的参数，但是略有一些不同需要引起注意。在 **scheduledAtFixedRate()** 方法中，第 3 个参数表示任务两次执行开始时间的间隔，而在 **scheduleWithFixedDelay()** 方法中，第 3 个参数则是表示任务上一次执行结束的时间与任务下一次开始执行的时间的间隔。

也可以配置 **ScheduledThreadPoolExecutor** 实现 **shutdown()** 方法的行为，默认行为是当调用 **shutdown()** 方法后，定时任务就结束了。可以通过 **ScheduledThreadPoolExecutor** 类的 **setContinueExistingPeriodicTasksAfterShutdownPolicy()** 方法来改变这个行为，传递参数 **true** 给这个方法，这样调用 **shutdown()** 方法后，周期性任务仍将继续执行。

参见

- ◆ 参见 4.2 节。
- ◆ 参见 4.7 节。

4.9 在执行器中取消任务

使用执行器时，不需要管理线程，只需要实现 **Runnable** 或 **Callable** 任务并发送任务给执行器即可。执行器负责创建线程，管理线程池中的线程，当线程不再需要时就销毁它们。有时候，我们可能需要取消已经发送给执行器的任务。在这种情况下，可以使用 **Future** 接口的 **cancel()** 方法来执行取消操作。在本节，我们将学习如何使用这个方法来取消已经发送给执行器的任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Callable** 接口，接口的泛型参数为 **String** 类型。接着实现 **call()**方法，构造一个无限循环，先在控制台输出信息，然后休眠 100 毫秒。

```
public class Task implements Callable<String> {
    @Override
    public String call() throws Exception {
        while (true) {
            System.out.printf("Task: Test\n");
            Thread.sleep(100);
        }
    }
}
```

2. 实现范例主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

3. 通过 **Executors** 工厂类的 **newCachedThreadPool()**方法创建一个 **ThreadPoolExecutor** 执行器对象。

```
ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
    newCachedThreadPool();
```

4. 创建一个新的 **Task** 对象。

```
Task task=new Task();
```

5. 调用 **submit()**方法将任务发送给执行器。

```
System.out.printf("Main: Executing the Task\n");
Future<String> result=executor.submit(task);
```

6. 让主线程休眠 2 秒。

```
try {
    TimeUnit.SECONDS.sleep(2);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

7. 执行器的 **submit()** 方法返回名为 **result** 的 **Future** 对象，调用 **Future** 对象的 **cancel()** 方法来取消任务的执行。传递参数 **true** 给这个 **cancel()** 方法。

```
System.out.printf("Main: Canceling the Task\n");
result.cancel(true);
```

8. 在控制台输出调用 **isCancelled()** 方法和 **isDone()** 方法的结果，来验证任务已经被取消和已完成。

```
System.out.printf("Main: Cancelled: %s\n", result.isCancelled());
System.out.printf("Main: Done: %s\n", result.isDone());
```

9. 调用 **shutdown()** 方法结束执行器，然后在控制台输出信息表示程序执行结束。

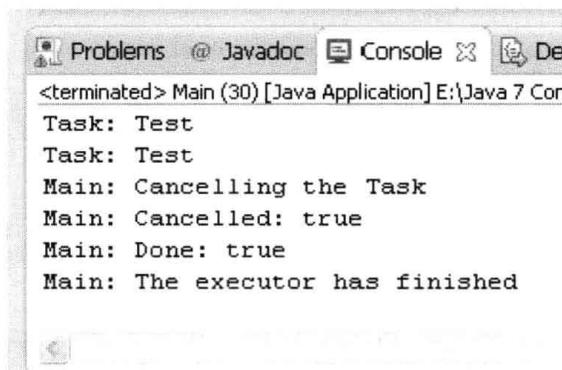
```
executor.shutdown();
System.out.printf("Main: The executor has finished\n");
```

工作原理

如果想取消一个已经发送给执行器的任务，可以使用 **Future** 接口的 **cancel()** 方法。根据调用 **cancel()** 方法时所传递的参数以及任务的状态，这个方法的行为有些不同。

- ◆ 如果任务已经完成，或者之前已经被取消，或者由于某种原因而不能被取消，那么方法将返回 **false** 并且任务也不能取消。
- ◆ 如果任务在执行器中等待分配 **Thread** 对象来执行它，那么任务被取消，并且不会开始执行。如果任务已经在运行，那么它依赖于调用 **cancel()** 方法时所传递的参数。如果传递的参数为 **true** 并且任务正在运行，那么任务将被取消。如果传递的参数为 **false** 并且任务正在运行，那么任务不会被取消。

下面的截图展示了范例执行的结果。



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```
<terminated> Main (30) [Java Application] E:\Java 7 Con
Task: Test
Task: Test
Main: Cancelling the Task
Main: Cancelled: true
Main: Done: true
Main: The executor has finished
```

更多信息

如果 **Future** 对象所控制任务已经被取消，那么使用 **Future** 对象的 **get()**方法时将抛出 **CancellationException** 异常。

参见

- ◆ 参见 4.4 节。

4.10 在执行器中控制任务的完成

FutureTask 类提供了一个名为 **done()** 的方法，允许在执行器中的任务执行结束之后，还可以执行一些代码。这个方法可以被用来执行一些后期处理操作，比如：产生报表，通过邮件发送结果或释放一些系统资源。当任务执行完成是受 **FutureTask** 类控制时，这个方法在内部被 **FutureTask** 类调用。在任务结果设置后以及任务的状态已改变为 **isDone** 之后，无论任务是否被取消或者正常结束，**done()**方法才被调用。

默认情况下，**done()**方法的实现为空，即没有任何具体的代码实现。我们可以覆盖 **FutureTask** 类并实现 **done()**方法来改变这种行为。在本节，我们将学习如何覆盖这个方法，并在任务结束后执行这些代码。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **ExecutableTask** 的类，并实现 **Callable** 接口，接口的泛型参数为 **String** 类型。

```
public class ExecutableTask implements Callable<String> {
```

2. 声明一个名为 **name** 的私有 **String** 属性，用来存储任务的名称，用 **getName()** 方

法来返回这个属性值。

```
private String name;
public String getName() {
    return name;
}
```

3. 实现类的构造器，并初始化任务的名称。

```
public ExecutableTask(String name) {
    this.name=name;
}
```

4. 实现 `call()` 方法。将任务休眠一段随机时间，并返回带有任务名称的消息。

```
@Override
public String call() throws Exception {
    try {
        long duration=(long)(Math.random()*10);
        System.out.printf("%s: Waiting %d seconds for results.\n",
        this.name,duration);
        TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
    }
    return "Hello, world. I'm "+name;
}
```

5. 实现一个名为 **ResultTask** 的类，并继承 **FutureTask** 类。**FutureTask** 类的泛型参数为 **String** 类型。

```
public class ResultTask extends FutureTask<String> {
```

6. 声明一个名为 **name** 的私有 **String** 属性，用来存储任务的名称。

```
private String name;
```

7. 实现类构造器。它接收一个 **Callable** 对象作为参数，调用父类构造器，并用接收到的任务属性来初始化 **name** 属性。

```
public ResultTask(Callable<String> callable) {
    super(callable);
    this.name=((ExecutableTask)callable).getName();
}
```

8. 覆盖 **done()**方法。检查 **isCancelled()**方法的返回值，然后根据这个返回值在控制台输出不同的信息。

```
Override
protected void done() {
    if (isCancelled()) {
        System.out.printf("%s: Has been canceled\n", name);
    } else {
        System.out.printf("%s: Has finished\n", name);
    }
}
```

9. 实现范例的主类，创建 **Main** 主类，然后实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

10. 调用 **Executors** 工厂类的 **newCachedThreadPool()**方法创建一个 **ExecutorService** 执行器对象。

```
ExecutorService executor=(ExecutorService) Executors.
    newCachedThreadPool();
```

11. 创建一个数组用来存储 5 个 **ResultTask** 对象。

```
ResultTask resultTasks[]=new ResultTask[5];
```

12. 初始化 **ResultTask** 对象。在数组的每一个位置上，必须创建 **ExecutorTask** 对象，然后创建 **ResultTask** 对象来使用 **ExecutorTask** 对象，最后调用 **submit()**方法将 **ResultTask** 任务发送给执行器。

```
for (int i=0; i<5; i++) {
    ExecutableTask executableTask=new ExecutableTask("Task "+i);
    resultTasks[i]=new ResultTask(executableTask);
    executor.submit(resultTasks[i]);
}
```

13. 将主线程休眠 5 秒钟。

```
try {
    TimeUnit.SECONDS.sleep(5);
```

```

} catch (InterruptedException e1) {
    e1.printStackTrace();
}

```

14. 取消已经发送给执行器的所有任务。

```

for (int i=0; i<resultTasks.length; i++) {
    resultTasks[i].cancel(true);
}

```

15. 通过调用 **ResultTask** 对象的 **get()**方法，在控制台上输出还没有被取消的任务结果。

```

for (int i=0; i<resultTasks.length; i++) {
    try {
        if (!resultTasks[i].isCancelled()){
            System.out.printf("%s\n",resultTasks[i].get());
        }
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

```

16. 调用 **shutdown()**方法结束执行器。

```

    executor.shutdown();
}
}

```

工作原理

当任务执行结束时， **FutureTask** 类就会调用 **done()**方法。在这个范例中，我们实现了一个 **Callable** 类、一个 **ExecutableTask** 类以及一个 **FutureTask** 类的子类 **ResultTask**，这个子类用来控制 **ExecutableTask** 对象的执行。

在创建好返回值以及改变任务状态为 **isDone** 之后，**FutureTask** 类就会在内部调用 **done()**方法。虽然我们无法改变任务的结果值，也无法改变任务的状态，但是可以通过任务来关闭系统资源、输出日志信息、发送通知等。

参见

- ◆ 参见 4.4 节。

4.11 在执行器中分离任务的启动与结果的处理

通常情况下，使用执行器来执行并发任务时，将 **Runnable** 或 **Callable** 任务发送给执行器，并获得 **Future** 对象来控制任务。此外，还会碰到如下情形，需要在一个对象里发送任务给执行器，然后在另一个对象里处理结果。对于这种情况，Java 并发 API 提供了 **CompletionService** 类。

CompletionService 类有一个方法用来发送任务给执行器，还有一个方法为下一个已经执行结束的任务获取 **Future** 对象。从内部实现机制来看，**CompletionService** 类使用 **Executor** 对象来执行任务。这个行为的优势是可以共享 **CompletionService** 对象，并发送任务到执行器，然后其他的对象可以处理任务的结果。第二个方法有一个不足之处，它只能为已经执行结束的任务获取 **Future** 对象，因此，这些 **Future** 对象只能被用来获取任务的结果。

在本节，我们将学习如何使用 **CompletionService** 类，在执行器中来分离任务的启动与结果的处理。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **ReportGenerator** 的类，并实现 **Callable** 接口，接口的泛型参数为 **String** 类型。

```
public class ReportGenerator implements Callable<String> {
```

2. 声明两个私有的 **String** 属性，分别命名为 **sender** 和 **title**，将用来表示报告的数据。

```
private String sender;  
private String title;
```

3. 实现类的构造器，用来初始化这两个属性。

```
public ReportGenerator(String sender, String title) {
```

```

    this.sender=sender;
    this.title=title;
}

```

4. 实现 **call()** 方法。让线程休眠一段随机时间。

```

@Override
public String call() throws Exception {
    try {
        long duration=(long)(Math.random()*10);
        System.out.printf("%s_%s: ReportGenerator: Generating a
report during %d seconds\n",this.sender,this.title,duration);
        TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

5. 生成包含了 **sender** 和 **title** 属性的字符串并返回该字符串。

```

String ret=sender+": "+title;
return ret;
}

```

6. 创建一个名为 **ReportRequest** 的类，实现 **Runnable** 接口。这个类将模拟请求获取报告。

```
public class ReportRequest implements Runnable {
```

7. 声明一个名为 **name** 的私有 **String** 属性，用来存储 **ReportRequest** 的名称。

```
private String name;
```

8. 声明一个名为 **service** 的私有 **CompletionService** 属性，这个 **CompletionService** 接口是泛型接口。在这个示例中，我们采用 **String** 类作为泛型参数。

```
private CompletionService<String> service;
```

9. 实现类的构造器，并初始化这两个属性。

```

public ReportRequest(String name, CompletionService<String>
service){
    this.name=name;
}

```

```

        this.service=service;
    }
}

```

10. 实现 **run()**方法。创建 **ReportGenerator** 对象，然后调用 **CompletionService** 对象的 **submit()**方法将 **ReportGenerator** 对象发送给 **CompletionService** 对象。

```

@Override
public void run() {
    ReportGenerator reportGenerator=new ReportGenerator(name,
        "Report");
    service.submit(reportGenerator);
}

```

11. 创建名为 **ReportProcessor** 的类，并实现 **Runnable** 接口。这个类将获取到 **ReportGenerator** 任务的结果。

```
public class ReportProcessor implements Runnable {
```

12. 声明一个名为 **service** 的私有 **CompletionService** 属性。由于 **CompletionService** 接口是一个泛型接口，在这个示例中，我们采用 **String** 类作为泛型参数。

```
private CompletionService<String> service;
```

13. 声明一个名为 **end** 的私有 **boolean** 属性。

```
private boolean end;
```

14. 实现类的构造器，并初始化这两个属性。

```

public ReportProcessor (CompletionService<String> service) {
    this.service=service;
    end=false;
}

```

15. 实现 **run()**方法。如果 **end** 属性值为 **false**，则调用 **CompletionService** 接口的 **poll()** 方法，来获取下一个已经完成任务的 **Future** 对象；当然，这个任务是采用 **CompletionService** 来完成的。

```

@Override
public void run() {
    while (!end) {
}

```

```
try {
    Future<String> result=service.poll(20, TimeUnit.SECONDS);
```

16. 通过调用 **Future** 对象的 **get()**方法来获取任务的结果，并在控制台输出这些结果。

```
if (result!=null) {
    String report=result.get();
    System.out.printf("ReportReceiver: Report Received:
%s\n",report);
}
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
System.out.printf("ReportSender: End\n");
}
```

17. 实现 **setEnd()**设置方法，修改 **end** 的属性值。

```
public void setEnd(boolean end) {
    this.end = end;
}
```

18. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

19. 调用 **Executors** 工厂类的 **newCachedThreadPool()**方法创建 **ThreadPoolExecutor** 执行器对象。

```
ExecutorService executor=(ExecutorService)Executors.
    newCachedThreadPool();
```

20. 创建 **CompletionService** 对象，并将上一步创建的 **executor** 对象作为构造器的参数。

```
CompletionService<String> service=new ExecutorCompletionService
    e<>(executor);
```

21. 创建两个 **ReportRequest** 对象，然后创建两个线程 **Thread** 对象分别来执行它们。

```
ReportRequest faceRequest=new ReportRequest("Face", service);
```

```
ReportRequest onlineRequest=new ReportRequest("Online",
    service);
Thread faceThread=new Thread(faceRequest);
Thread onlineThread=new Thread(onlineRequest);
```

22. 创建1个**ReportProcessor**对象，然后创建1个线程**Thread**对象来执行它。

```
ReportProcessor processor=new ReportProcessor(service);
Thread senderThread=new Thread(processor);
```

23. 启动这3个线程。

```
System.out.printf("Main: Starting the Threads\n");
faceThread.start();
onlineThread.start();
senderThread.start();
```

24. 等待**ReportRequest**线程的结束。

```
try {
    System.out.printf("Main: Waiting for the report
        generators.\n");
    faceThread.join();
    onlineThread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

25. 调用**shutdown()**方法结束执行器，然后调用**awaitTermination()**方法等待所有的任务执行结束。

```
System.out.printf("Main: Shutting down the executor.\n");
executor.shutdown();
try {
    executor.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

26. 调用**setEnd()**方法，设置**end**属性为**true**，来结束**ReportSender**的执行。

```
processor.setEnd(true);
System.out.println("Main: Ends");
```

工作原理

在范例的主类中，我们调用 **Executors** 工厂类的 **newCachedThreadPool()** 方法创建了 **ThreadPoolExecutor** 执行器对象。然后，使用这个对象初始化了 **CompletionService** 对象，因为完成服务（**Completion Service**）使用执行器来执行任务。然后，调用 **ReportRequest** 类中的 **submit()** 方法，利用“完成服务”来执行任务。

当“完成服务”任务结束，这些任务中一个任务就执行结束了，“完成服务”中存储着 **Future** 对象，用来控制它在队列中的执行。

调用 **poll()** 方法访问这个队列，查看是否有任务已经完成，如果有，则返回队列中的第一个元素（即一个任务执行完成后的 **Future** 对象）。当 **poll()** 方法返回 **Future** 对象后，它将从队列中删除这个 **Future** 对象。在这个示例中，我们在调用 **poll()** 方法时传递了两个参数，表示当队列里完成任务的结果为空时，想要等待任务执行结束的时间。

一旦创建了 **CompletionService** 对象，还要创建两个 **ReportRequest** 对象，用来执行在 **CompletionService** 中的两个 **ReportGenerator** 任务。**ReportProcessor** 任务则将处理两个被发送到执行器里的 **ReportRequest** 任务所产生的结果。

更多信息

CompletionService 类可以执行 **Callable** 或 **Runnable** 类型的任务。在这个范例中，使用的是 **Callable** 类型的任务，但是，也可以发送 **Runnable** 对象给它。由于 **Runnable** 对象不能产生结果，因此 **CompletionService** 的基本原则不适用于此。

CompletionService 类提供了其他两种方法来获取任务已经完成的 **Future** 对象。这些方法如下。

- ◆ **poll()**: 无参数的 **poll()** 方法用于检查队列中是否有 **Future** 对象。如果队列为空，则立即返回 **null**。否则，它将返回队列中的第一个元素，并移除这个元素。

- ◆ **take()**: 这个方法也没有参数，它检查队列中是否有 **Future** 对象。如果队列为空，它将阻塞线程直到队列中有可用的元素。如果队列中有元素，它将返回队列中的第一个元素，并移除这个元素。

参见

- ◆ 参见 4.4 节。

4.12 处理在执行器中被拒绝的任务

当我们想结束执行器的执行时，调用 **shutdown()** 方法来表示执行器应当结束。但是，执行器只有等待正在运行的任务或者等待执行的任务结束后，才能真正结束。

如果在 **shutdown()** 方法与执行器结束之间发送一个任务给执行器，这个任务会被拒绝，因为这个时间段执行器已不再接受任务了。**ThreadPoolExecutor** 类提供了一套机制，当任务被拒绝时调用这套机制来处理它们。

在本节，我们将学习如何处理执行器中被拒绝的任务，这些任务实现了 **RejectedExecutionHandler** 接口。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **RejectedTaskController** 的类，并实现 **RejectedExecutionHandler** 接口，然后实现接口的 **rejectedExecution()** 方法，在控制台输出已被拒绝的任务的名称和执行器的状态。

```
public class RejectedTaskController implements  
RejectedExecutionHandler {  
    @Override  
    public void rejectedExecution(Runnable r, ThreadPoolExecutor  
        executor) {  
        System.out.printf("RejectedTaskController: The task %s has  
            been rejected\n", r.toString());  
        System.out.printf("RejectedTaskController: %s\n", executor.  
            toString());  
        System.out.printf("RejectedTaskController: Terminating:  
            %s\n", executor.isTerminating());  
        System.out.printf("RejectedTaksController: Terminated:  
            %s\n", executor.isTerminated());  
    }  
}
```

```
}
```

2. 创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable{
```

3. 声明一个名为 **name** 的私有 **String** 属性，用来存储任务的名称。

```
private String name;
```

4. 实现类的构造器，用来初始化类的属性。

```
public Task(String name){  
    this.name=name;  
}
```

5. 实现 **run()** 方法。在控制台输出信息表示方法开始执行。

```
@Override  
public void run() {  
    System.out.println("Task "+name+": Starting");
```

6. 让线程休眠一段随机时间。

```
try {  
    long duration=(long)(Math.random()*10);  
    System.out.printf("Task %s: ReportGenerator: Generating a  
        report during %d seconds\n",name,duration);  
    TimeUnit.SECONDS.sleep(duration);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

7. 在控制台输出信息表示方法执行结束。

```
System.out.printf("Task %s: Ending\n",name);  
}
```

8. 覆盖 **toString()** 方法，返回任务的名称。

```
public String toString() {  
    return name;  
}
```

9. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {  
    public static void main(String[] args) {
```

10. 创建 **RejectedTaskController** 对象来管理被拒绝的任务。

```
RejectedTaskController controller=new  
RejectedTaskController();
```

11. 调用 **Executors** 工厂类的 **newCachedThreadPool()** 方法创建 **ThreadPoolExecutor** 执行器对象。

```
ThreadPoolExecutor executor=(ThreadPoolExecutor) Executors.  
newCachedThreadPool();
```

12. 设置用于被拒绝的任务的处理程序。

```
executor.setRejectedExecutionHandler(controller);
```

13. 创建 3 个任务并发送给执行器。

```
System.out.printf("Main: Starting.\n");  
for (int i=0; i<3; i++) {  
    Task task=new Task("Task"+i);  
    executor.submit(task);  
}
```

14. 调用 **shutdown()** 方法关闭执行器。

```
System.out.printf("Main: Shutting down the Executor.\n");  
executor.shutdown();
```

15. 创建另一个任务并发送给执行器。

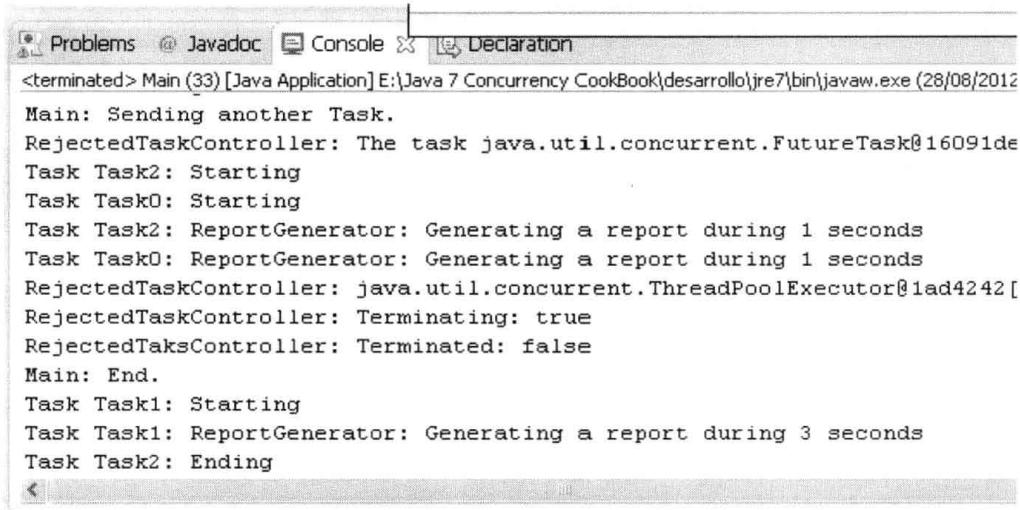
```
System.out.printf("Main: Sending another Task.\n");  
Task task=new Task("RejectedTask");  
executor.submit(task);
```

16. 在控制台输出信息表示程序结束。

```
System.out.println("Main: End");  
System.out.printf("Main: End.\n");
```

工作原理

通过下面的截图，可以看到范例运行的结果。



```

Problems @ Javadoc Console Declaration
<terminated> Main (33) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (28/08/2012)
Main: Sending another Task.
RejectedTaskController: The task java.util.concurrent.FutureTask@16091de
Task Task2: Starting
Task Task0: Starting
Task Task2: ReportGenerator: Generating a report during 1 seconds
Task Task0: ReportGenerator: Generating a report during 1 seconds
RejectedTaskController: java.util.concurrent.ThreadPoolExecutor@1ad4242[...
RejectedTaskController: Terminating: true
RejectedTaskController: Terminated: false
Main: End.
Task Task1: Starting
Task Task1: ReportGenerator: Generating a report during 3 seconds
Task Task2: Ending

```

我们可以看到被拒绝的任务，当执行已经关闭，**RejectedTaskController** 在控制台输出任务和执行器的信息。

为了处理在执行器中被拒绝的任务，需要创建一个实现 **RejectedExecutionHandler** 接口的处理类。这个接口有一个 **rejectedExecution()** 方法，其中有两个参数：

- ◆ 一个 **Runnable** 对象，用来存储被拒绝的任务；
- ◆ 一个 **Executor** 对象，用来存储任务被拒绝的执行器。

被执行器拒绝的每一个任务都将调用这个方法。需要先调用 **Executor** 类的 **setRejectedExecutionHandler()** 方法来设置用于被拒绝的任务的处理程序。

更多信息

当执行器接收一个任务并开始执行时，它先检查 **shutdown()** 方法是否已经被调用了。如果是，那么执行器就拒绝这个任务。首先，执行器会寻找通过 **setRejectedExecutionHandler()** 方法设置的用于被拒绝的任务的处理程序，如果找到一个处理程序，执行器就调用其 **rejectedExecution()** 方法；否则就抛出 **RejectedExecutionException** 异常。这是一个运行时异常，因此并不需要 **catch** 语句来对其进行处理。

参见

- ◆ 参见 4.2 节。

第 5 章

Fork/Join 框架

本章内容包含：

- ◆ 创建 Fork / Join 线程池
- ◆ 合并任务的结果
- ◆ 异步运行任务
- ◆ 在任务中抛出异常
- ◆ 取消任务

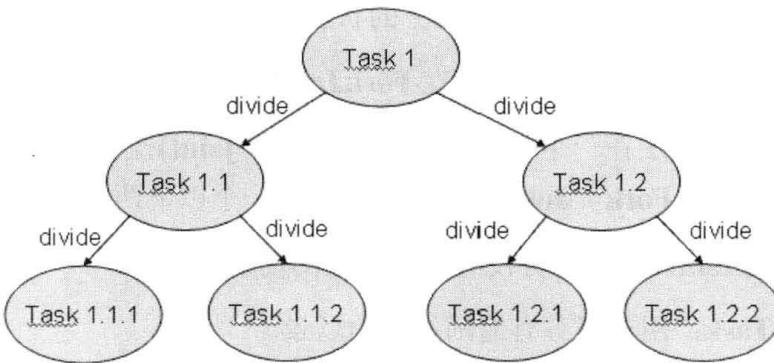
5.1 简介

通常，使用 Java 来开发一个简单的并发应用程序时，会创建一些 **Runnable** 对象，然后创建对应的 **Thread** 对象来控制程序中这些线程的创建、执行以及线程的状态。自从 Java 5 开始引入了 **Executor** 和 **ExecutorService** 接口以及实现这两个接口的类（比如 **ThreadPoolExecutor**）之后，使得 Java 在并发支持上得到了进一步的提升。

执行器框架（**Executor Framework**）将任务的创建和执行进行了分离，通过这个框架，只需要实现 **Runnable** 接口的对象和使用 **Executor** 对象，然后将 **Runnable** 对象发送给执行器。执行器再负责运行这些任务所需要的线程，包括线程的创建，线程的管理以及线程的结束。

Java 7 则又更进了一步，它包括了 **ExecutorService** 接口的另一种实现，用来解决特殊类型的问题，它就是 **Fork / Join 框架**，有时也称分解 / 合并框架。

Fork / Join 框架是用来解决能够通过分治技术（**Divide and Conquer Technique**）将问题拆分成小任务的问题。在一个任务中，先检查将要解决的问题的大小，如果大于一个设定的大小，那就将问题拆分成可以通过框架来执行的小任务。如果问题的大小比设定的大小要小，就可以直接在任务里解决这个问题，然后，根据需要返回任务的结果。下面的图形总结了这个原理。



没有固定的公式来决定问题的参考大小（**Reference Size**），从而决定一个任务是需要进行拆分或不需要拆分，拆分与否仍是依赖于任务本身的特性。可以使用在任务中将要处理的元素的数目和任务执行所需要的时间来决定参考大小。测试不同的参考大小来决定解决问题最好的一个方案，将 **ForkJoinPool** 类看作一个特殊的 **Executor** 执行器类型。这个框架基于以下两种操作。

- ◆ 分解（**Fork**）操作：当需要将一个任务拆分成更小的多个任务时，在框架中执行这些任务；
- ◆ 合并（**Join**）操作：当一个主任务等待其创建的多个子任务的完成执行。

Fork / Join 框架和执行器框架（**Executor Framework**）主要的区别在于工作窃取算法（**Work-Stealing Algorithm**）。与执行器框架不同，使用 **Join** 操作让一个主任务等待它所创建的子任务的完成，执行这个任务的线程称之为工作者线程（**Worker Thread**）。工作者线程寻找其他仍未被执行的任务，然后开始执行。通过这种方式，这些线程在运行时拥有所有的优点，进而提升应用程序的性能。

为了达到这个目标，通过 **Fork / Join 框架**执行的任务有以下限制。

- ◆ 任务只能使用 **fork()** 和 **join()** 操作当作同步机制。如果使用其他的同步机制，工

作者线程就不能执行其他任务，当然这些任务是在同步操作里时。比如，如果在 **Fork / Join** 框架中将一个任务休眠，正在执行这个任务的工作者线程在休眠期内不能执行另一个任务。

- ◆ 任务不能执行 I/O 操作，比如文件数据的读取与写入。
- ◆ 任务不能抛出非运行时异常 (Checked Exception)，必须在代码中处理掉这些异常。

Fork / Join 框架的核心是由下列两个类组成的。

- ◆ **ForkJoinPool**: 这个类实现了 ExecutorService 接口和工作窃取算法 (Work-Stealing Algorithm)。它管理工作者线程，并提供任务的状态信息，以及任务的执行信息。
- ◆ **ForkJoinTask**: 这个类是一个将在 **ForkJoinPool** 中执行的任务的基类。

Fork / Join 框架提供了在一个任务里执行 **fork()** 和 **join()** 操作的机制和控制任务状态的方法。通常，为了实现 **Fork / Join** 任务，需要实现一个以下两个类之一的子类。

- ◆ **RecursiveAction**: 用于任务没有返回结果的场景。
- ◆ **RecursiveTask**: 用于任务有返回结果的场景。

本章接下来将展示如何利用 **Fork / Join** 框架高效地工作。

5.2 创建 Fork / Join 线程池

在本节，我们将学习如何使用 **Fork / Join** 框架的基本元素。它包括：

- ◆ 创建用来执行任务的 **ForkJoinPool** 对象；
- ◆ 创建即将在线程池中被执行的任务 **ForkJoinTask** 子类。

本范例中即将使用的 **Fork / Join** 框架的主要特性如下：

- ◆ 采用默认的构造器创建 **ForkJoinPool** 对象；
- ◆ 在任务中将使用 JavaAPI 文档推荐的结构。

```
if (problem size > default size){  
    tasks=divide(task);  
    execute(tasks);  
} else {  
    resolve problem using another algorithm;  
}
```

◆ 我们将以同步的方式执行任务。当一个主任务执行两个或更多的子任务时，这个主任务将等待子任务的完成。用这种方法，执行主任务的线程，称之为**工作者线程（Worker Thread）**，它将寻找其他的子任务来执行，并在子任务执行的时间里利用所有的线程优势。

◆ 如果将要实现的任务没有返回任何结果，那么，采用 **RecursiveAction** 类作为实现任务的基类。

准备工作

本节的范例是在 EclipseIDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

在本节，我们将实现一项更新产品价格的任务。最初的任务将负责更新列表中的所有元素。我们使用 **10** 来作为**参考大小（ReferenceSize）**，如果一个任务需要更新大于 **10** 个元素，它会将这个列表分解成为两部分，然后分别创建两个任务用来更新各自部分的产品价格。

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Product** 的类，用来存储产品的名称和价格。

```
public class Product {
```

2. 声明一个名为 **name** 的私有 **String** 属性，一个名为 **price** 的私有 **double** 属性。

```
private String name;
private double price;
```

3. 实现两个属性各自的设值与取值方法。

```
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
```

```
        this.price = price;
    }
```

4. 创建一个名为 **ProductListGenerator** 的类，用来生成一个随机产品列表。

```
public class ProductListGenerator {
```

5. 实现 **generate()**方法。接收一个表示列表大小的 **int** 参数，并返回一个生成产品的 **List<Product>**列表。

```
public List<Product> generate (int size) {
```

6. 创建返回产品列表的对象 **ret**。

```
List<Product> ret=new ArrayList<Product>();
```

7. 生成产品列表，给所有的产品分配相同的价格，比如可以检查程序是否运行良好的数字 **10**。

```
for (int i=0; i<size; i++) {
    Product product=new Product();
    product.setName("Product "+i);
    product.setPrice(10);
    ret.add(product);
}
return ret;
}
```

8. 创建一个名为 **Task** 的类，并继承 **RecursiveAction** 类。

```
public class Task extends RecursiveAction {
```

9. 声明这个类的 **serialVersionUID** 属性。这个元素是必需的，因为 **RecursiveAction** 的父类 **ForkJoinTask** 实现了 **Serializable** 接口。

```
private static final long serialVersionUID = 1L;
```

10. 声明一个名为 **products** 私有的 **List<Product>**属性。

```
private List<Product> products;
```

11. 声明两个私有的 **int** 属性，分别命名为 **first** 和 **last**。这两个属性将决定任务执行时对产品的分块。

```
private int first;
private int last;
```

12. 声明一个名为 **increment** 的私有 **double** 属性，用来存储产品价格的增加额。

```
private double increment;
```

13. 实现类的构造器，用来初始化类的这些属性。

```
public Task (List<Product> products, int first, int last, double
increment) {
    this.products=products;
    this.first=first;
    this.last=last;
    this.increment=increment;
}
```

14. 实现 **compute()** 方法，实现任务的执行逻辑。

```
@Override
protected void compute() {
```

15. 如果 **last** 和 **first** 属性值的差异小于 **10**(一个任务只能更新少于 **10** 件产品的价格)，则调用 **updatePrices()** 方法增加这些产品的价格。

```
if (last-first<10) {
    updatePrices();
```

16. 如果 **last** 和 **first** 属性值的差异大于或等于 **10**，就创建两个新的 **Task** 对象，一个处理前一半的产品，另一个处理后一半的产品，然后调用 **ForkJoinPool** 的 **invokeAll()** 方法来执行这两个新的任务。

```
} else {
    int middle=(last+first)/2;
    System.out.printf("Task: Pending tasks:
    %s\n",getQueuedTaskCount());
    Task t1=new Task(products, first,middle+1, increment);
    Task t2=new Task(products, middle+1,last, increment);
```

```
    invokeAll(t1, t2);
}
```

17. 实现 **updatePrices()**方法。这个方法用来更新在产品列表中处于 **first** 和 **last** 属性之间的产品。

```
private void updatePrices() {
    for (int i=first; i<last; i++) {
        Product product=products.get(i);
        product.setPrice(product.getPrice()*(1+increment));
    }
}
```

18. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

19. 使用 **ProductListGenerator** 类创建一个有 10,000 个产品的列表

```
ProductListGenerator generator=new ProductListGenerator();
List<Product> products=generator.generate(10000);
```

20. 创建一个新的 **Task** 对象用来更新列表中的所有产品。参数 **first** 为 **0**，参数 **last** 为产品列表的大小，即 **10,000**。

```
Task task=new Task(products, 0, products.size(), 0.20);
```

21. 通过无参的类构造器创建一个 **ForkJoinPool** 对象。

```
ForkJoinPool pool=new ForkJoinPool();
```

22. 调用 **execute()**方法执行任务。

```
pool.execute(task);
```

23. 实现代码块，显示关于线程池演变的信息，每 **5** 毫秒在控制台上输出线程池的一些参数值，直到任务执行结束。

```
do {
    System.out.printf("Main: Thread Count: %d\n", pool.
```

```

        getActiveThreadCount());
System.out.printf("Main: Thread Steal: %d\n",pool.
        getStealCount());
System.out.printf("Main: Parallelism: %d\n",pool.
        getParallelism());
try {
    TimeUnit.MILLISECONDS.sleep(5);
} catch (InterruptedException e) {
    e.printStackTrace();
}
} while (!task.isDone());

```

24. 调用 **shutdown()**方法关闭线程池。

```
pool.shutdown();
```

25. 调用 **isCompletedNormally()**方法，检查任务是否已经完成并且没有错误，在这个示例中，在控制台输出信息表示任务已经处理结束。

```

if (task.isCompletedNormally()){
    System.out.printf("Main: The process has completed
        normally.\n");
}

```

26. 在增加之后，所有产品的期望价格是 **12** 元。在控制台输出所有产品的名称和价格，如果产品的价格不是 **12** 元，就将产品信息打印出来，以便确认所有的产品价格都正确地增加了。

```

for (int i=0; i<products.size(); i++) {
    Product product=products.get(i);
    if (product.getPrice() !=12) {
        System.out.printf("Product %s: %f\n",product.
            getName(),product.getPrice());
    }
}

```

27. 在控制台输出信息表示程序执行结束。

```
System.out.println("Main: End of the program.\n");
```

工作原理

在这个范例中，我们创建了 **ForkJoinPool** 对象，和一个将在线程池中执行的 **ForkJoinTask** 的子类。使用了无参的类构造器创建了 **ForkJoinPool** 对象，因此它将执行默认的配置。创建一个线程数等于计算机 **CPU** 数目的线程池，创建好 **ForkJoinPool** 对象之后，那些线程也创建就绪了，在线程池中等待任务的到达，然后开始执行。

由于 **Task** 类继承了 **RecursiveAction** 类，因此不返回结果。在本节，我们使用了推荐的结构来实现任务。如果任务需要更新大于 **10** 个产品，它将拆分这些元素为两部分，创建两个任务，并将拆分的部分相应地分配给新创建的任务。通过使用 **Task** 类的 **first** 和 **last** 属性，来获知任务将要更新的产品列表所在的位置范围。我们已经使用 **first** 和 **last** 属性，来操作产品列表中仅有的一份副本，而没有为每一个任务去创建不同的产品列表。

调用 **invokeAll()** 方法来执行一个主任务所创建的多个子任务。这是一个同步调用，这个任务将等待子任务完成，然后继续执行（也可能是结束）。当一个主任务等待它的子任务时，执行这个主任务的工作者线程接收另一个等待执行的任务并开始执行。正因为有了这个行为，所以说 **Fork / Join** 框架提供了一种比 **Runnable** 和 **Callable** 对象更加高效的任务管理机制。

ForkJoinTask 类的 **invokeAll()** 方法是执行器框架（**ExecutorFramework**）和 **Fork / Join** 框架之间的主要差异之一。在执行器框架中，所有的任务必须发送给执行器，然而，在这个示例中，线程池中包含了待执行方法的任务，任务的控制也是在线程池中进行的。我们在 **Task** 类中使用了 **invokeAll()** 方法，**Task** 类继承了 **RecursiveAction** 类，而 **RecursiveAction** 类则继承了 **ForkJoinTask** 类。

我们已经发送一个唯一的任务到线程池中，通过使用 **execute()** 方法来更新所有产品的列表。在这个示例中，它是一个同步调用，主线程一直等待调用的执行。

我们已经使用了 **ForkJoinPool** 类的一些方法，来检查正在运行的任务的状态和演变情况。这个类包含更多的方法，可以用于任务状态的检测。参见 8.5 节介绍的这些方法的完整列表。

最后，像执行器框架一样，必须调用 **shutdown()** 方法来结束 **ForkJoinPool** 的执行。

下面的截图展示了这个范例执行的部分结果。

可以看到，任务执行结束，并且产品的价格已经更新了。

```

<terminated> Main (34) [Java Application] E:\Java 7 Concurrency CookBook\des
Task: Pending tasks: 0
Task: Pending tasks: 1
Task: Pending tasks: 2
Task: Pending tasks: 0
Task: Pending tasks: 1
Task: Pending tasks: 0
Task: Pending tasks: 1
Main: Thread Count: 1
Main: Thread Steal: 2
Main: Parallelism: 2
Main: The process has completed normally.
Main: End of the program.

```

更多信息

ForkJoinPool 类还提供了以下方法用于执行任务。

- ◆ **execute(Runnable task)**: 这是本范例中使用的 **execute()** 方法的另一种版本。这个方法发送一个 **Runnable** 任务给 **ForkJoinPool** 类。需要注意的是，使用 **Runnable** 对象时 **ForkJoinPool** 类就不采用工作窃取算法（**Work-StealingAlgorithm**），**ForkJoinPool** 类仅在使用 **ForkJoinTask** 类时才采用工作窃取算法。

- ◆ **invoke(ForkJoinTask<T>task)**: 正如范例所示，**ForkJoinPool** 类的 **execute()** 方法是异步调用的，而 **ForkJoinPool** 类的 **invoke()** 方法则是同步调用的。这个方法直到传递进来的任务执行结束后才会返回。

- ◆ 也可以使用在 **ExecutorService** 类中声明的 **invokeAll()** 和 **invokeAny()** 方法，这些方法接收 **Callable** 对象作为参数。使用 **Callable** 对象时 **ForkJoinPool** 类就不采用工作窃取算法（**Work-StealingAlgorithm**），因此，最好使用执行器来执行 **Callable** 对象。

ForkJoinTask 类也包含了在范例中所使用的 **invokeAll()** 方法的其他版本，这些版本如下。

- ◆ **invokeAll(ForkJoinTask<?>... tasks)**: 这个版本的方法接收一个可变的参数列表，可以传递尽可能多的 **ForkJoinTask** 对象给这个方法作为参数。

- ◆ **invokeAll(Collection<T>tasks)**: 这个版本的方法接受一个泛型类型 T 的对象集合

(比如，**ArrayList** 对象、**LinkedList** 对象或者 **TreeSet** 对象)。这个泛型类型 T 必须是 **ForkJoinTask** 类或者它的子类。

虽然 **ForkJoinPool** 类是设计用来执行 **ForkJoinTask** 对象的，但也可以直接用来执行 **Runnable** 和 **Callable** 对象。当然，也可以使用 **ForkJoinTask** 类的 **adapt()** 方法来接收一个 **Callable** 对象或者一个 **Runnable** 对象，然后将之转化为一个 **ForkJoinTask** 对象，然后再去执行。

参见

- ◆ 参见 8.5 节。

5.3 合并任务的结果

Fork / Join 框架提供了执行任务并返回结果的能力。这些类型的任务都是通过 **RecursiveTask** 类来实现的。**RecursiveTask** 类继承了 **ForkJoinTask** 类，并且实现了由执行器框架（**Executor Framework**）提供的 **Future** 接口。

在任务中，必须使用 Java API 文档推荐的如下结构：

```
if (problem size > size){  
    tasks=Divide(task);  
    execute(tasks);  
    groupResults()  
    return result;  
} else {  
    resolve problem;  
    return result;  
}
```

如果任务需要解决的问题大于预先定义的大小，那么就要将这个问题拆分成多个子任务，并使用 **Fork / Join** 框架来执行这些子任务。执行完成后，原始任务获取到由所有这些子任务产生的结果，合并这些结果，返回最终的结果。当原始任务在线程池中执行结束后，将高效地获取到整个问题的最终结果。

在本节，我们将学习如何使用 **Fork / Join** 框架来解决这种问题，开发一个应用程序，在文档中查找一个词。我们将实现以下两种任务：

- ◆ 一个文档任务，它将遍历文档中的每一行来查找这个词；
- ◆ 一个行任务，它将在文档的一部分当中查找这个词。

所有这些任务将返回文档或行中所出现这个词的次数。

准备工作

本节的范例是在 EclipseIDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **DocumentMock** 的类。它将生成一个字符串矩阵来模拟一个文档。

```
public class Document {
```

2. 用一些词来创建一个字符串数组。这个数组将被用来生成字符串矩阵。

```
private String words[]={"the","hello","goodbye","packt", "java","t
hread","pool","random","class","main"};
```

3. 实现 **generateDocument()** 方法。它接收 3 个参数，分别是行数 **numLines**，每一行词的个数 **numWords**，和准备查找的词 **word**。然后返回一个字符串矩阵。

```
public String[][] generateDocument(int numLines, int numWords,
String word) {
```

4. 创建用来生成文档所需要的对象： **String** 矩阵，和用来生成随机数的 **Random** 对象。

```
int counter=0;
String document[][]=new String[numLines][numWords];
Random random=new Random();
```

5. 为字符串矩阵填上字符串。通过随机数取得数组 **words** 中的某一字符串，然后存入到字符串矩阵 **document** 对应的位置上，同时计算生成的字符串矩阵中将要查找的词出现的次数。这个值可以用来与后续程序运行查找任务时统计的次数相比较，检查两个值是否相同。

```

for (int i=0; i<numLines; i++) {
    for (int j=0; j<numWords; j++) {
        int index=random.nextInt(words.length);
        document[i][j]=words[index];
        if (document[i][j].equals(word)) {
            counter++;
        }
    }
}

```

6. 在控制台输出这个词出现的次数，并返回生成的矩阵 **document**。

```

System.out.println("DocumentMock: The word appears "+
    counter+" times in the document");
return document;

```

7. 创建名为 **DocumentTask** 的类，并继承 **RecursiveTask** 类，**RecursiveTask** 类的泛型参数为 **Integer** 类型。这个 **DocumentTask** 类将实现一个任务，用来计算所要查找的词在行中出现的次数。

```
public class DocumentTask extends RecursiveTask<Integer> {
```

8. 声明一个名为 **document** 的私有 **String** 矩阵，以及两个名为 **start** 和 **end** 的私有 **int** 属性，并声明一个名为 **word** 的私有 **String** 属性。

```

private String document[][];
private int start, end;
private String word;

```

9. 实现类的构造器，用来初始化类的所有属性。

```

public DocumentTask (String document[][], int start, int end,
    String word){
    this.document=document;
    this.start=start;
    this.end=end;
    this.word=word;
}

```

10. 实现 **compute()**方法。如果 **end** 和 **start** 的差异小于 **10**，则调用 **processLines()**方法，来计算这两个位置之间要查找的词出现的次数。

```

@Override
protected Integer compute() {
    int result;
    if (end-start<10){
        result=processLines(document, start, end, word);
    }
}

```

11. 否则，拆分这些行成为两个对象，并创建两个新的 **DocumentTask** 对象来处理这两个对象，然后调用 **invokeAll()** 方法在线程池里执行它们。

```

} else {
    int mid=(start+end)/2;
    DocumentTask task1=new DocumentTask(document,start,mid,word);
    DocumentTask task2=new DocumentTask(document,mid,end,word);
    invokeAll(task1,task2);
}

```

12. 采用 **groupResults()** 方法将这两个任务返回的值相加。最后，返回任务计算的结果。

```

try {
    result=groupResults(task1.get(),task2.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
return result;
}

```

13. 实现 **processLines()** 方法。接收 4 个参数，一个字符串 **document** 矩阵，**start** 属性，**end** 属性和任务将要查找的词 **word** 的属性。

```

private Integer processLines(String[][] document, int start, int
    end, String word) {
}

```

14. 为任务所要处理的每一行，创建一个 **LineTask** 对象，然后存储在任务列表里。

```

List<LineTask> tasks=new ArrayList<LineTask>();
for (int i=start; i<end; i++){
    LineTask task=new LineTask(document[i], 0, document[i].
        length, word);
    tasks.add(task);
}

```

15. 调用 **invokeAll()**方法执行列表中所有的任务。

```
invokeAll(tasks);
```

16. 合计这些任务返回的值，并返回结果。

```
int result=0;
for (int i=0; i<tasks.size(); i++) {
    LineTask task=tasks.get(i);
    try {
        result=result+task.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
return new Integer(result);
```

17. 实现 **groupResults()**方法。它将两个数字相加并返回结果。

```
private Integer groupResults(Integer number1, Integer number2) {
    Integer result;
    result=number1+number2;
    return result;
}
```

18. 创建名为 **LineTask** 的类，并继承 **RecursiveTask** 类，**RecursiveTask** 类的泛型参数为 **Integer** 类型。这个 **RecursiveTask** 类实现了一个任务，用来计算所要查找的词在一行中出现的次数。

```
public class LineTask extends RecursiveTask<Integer>{
```

19. 声明类的 **serialVersionUID** 属性。这个元素是必需的，因为 **RecursiveTask** 的父类 **ForkJoinTask** 实现了 **Serializable** 接口。声明一个名为 **line** 的私有 **String** 数组属性和两个名为 **start** 和 **end** 的私有 **int** 属性。最后，声明一个名为 **word** 的私有 **String** 属性。

```
private static final long serialVersionUID = 1L;
private String line[];
private int start, end;
private String word;
```

20. 实现类的构造器，用来初始化它的属性。

```
public LineTask(String line[], int start, int end, String word)
{
    this.line=line;
    this.start=start;
    this.end=end;
    this.word=word;
}
```

21. 实现 **compute()** 方法。如果 **end** 和 **start** 属性的差异小于 **100**, 那么任务将采用 **count()** 方法, 在由 **start** 与 **end** 属性所决定的行的片断中查找词。

```
@Override
protected Integer compute() {
    Integer result=null;
    if (end-start<100) {
        result=count(line, start, end, word);
```

22. 如果 **end** 和 **start** 属性的差异不小于 100, 将这一组词拆分成两组, 然后创建两个新的 **LineTask** 对象来处理这两个组, 调用 **invokeAll()** 方法在线程池中执行它们。

```
} else {
    int mid=(start+end)/2;
    LineTask task1=new LineTask(line, start, mid, word);
    LineTask task2=new LineTask(line, mid, end, word);
    invokeAll(task1, task2);
```

23. 调用 **groupResults()** 方法将两个任务返回的值相加。最后返回任务计算的结果。

```
try {
    result=groupResults(task1.get(),task2.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
return result;
```

24. 实现 **count()** 方法。它接收 4 个参数, 一个完整行字符串 **line** 数组, **start** 属性, **end** 属性和任务将要查找的词 **word** 的属性。

```
private Integer count(String[] line, int start, int end, String
```

```
word) {
```

25. 将存储在 **start** 和 **end** 属性值之间的词与任务正在查找的 **word** 属性相比较。如果相同，那么将计数器 **counter** 变量加 **1**。

```
int counter;
counter=0;
for (int i=start; i<end; i++) {
    if (line[i].equals(word)) {
        counter++;
    }
}
```

26. 为了延缓范例的执行，将任务休眠 **10** 毫秒。

```
try {
    Thread.sleep(10);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

27. 返回计数器 **counter** 变量的值。

```
return counter;
```

28. 实现 **groupResults()**方法。计算两个数字之和并返回结果。.

```
private Integer groupResults(Integer number1, Integer number2) {
    Integer result;
    result=number1+number2;
    return result;
}
```

29. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```
public class Main{
    public static void main(String[] args) {
```

30. 创建 **Document** 对象，包含 **100** 行，每行 **1,000** 个词。

```
DocumentMock mock=new DocumentMock();
```

```
String[][] document=mock.generateDocument(100, 1000, "the");
```

31. 创建一个 **DocumentTask** 对象，用来更新整个文档。传递数字 **0** 给参数 **start**，以及数字 **100** 给参数 **end**。

```
DocumentTask task=new DocumentTask(document, 0, 100, "the");
```

32. 采用无参的构造器创建一个 **ForkJoinPool** 对象，然后调用 **execute()**方法在线程池里执行这个任务。

```
ForkJoinPool pool=new ForkJoinPool();
pool.execute(task);
```

33. 实现代码块，显示线程池的进展信息，每秒钟在控制台输出线程池的一些参数，直到任务执行结束。

```
do {
    System.out.printf("*****\n");
    System.out.printf("Main: Parallelism: %d\n",pool.
        getParallelism());
    System.out.printf("Main: Active Threads: %d\n",pool.
        getActiveThreadCount());
    System.out.printf("Main: Task Count: %d\n",pool.
        getQueuedTaskCount());
    System.out.printf("Main: Steal Count: %d\n",pool.
        getStealCount());
    System.out.printf("*****\n");
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
} while (!task.isDone());
```

34. 调用 **shutdown()**方法关闭线程池。

```
pool.shutdown();
```

35. 调用 **awaitTermination()**等待任务执行结束。

```

try {
    pool.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

36. 在控制台输出文档中出现要查找的词的次数。检验这个数字与 **DocumentMock** 类输出的数字是否一致。

```

try {
    System.out.printf("Main: The word appears %d in the
                      document",task.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}

```

工作原理

在这个范例中，我们实现了两个不同的任务。

◆ **DocumentTask** 类：这个类的任务需要处理由 **start** 和 **end** 属性决定的文档行。如果这些行数小于 **10**，那么，就每行创建一个 **LineTask** 对象，然后在任务执行结束后，合计返回的结果，并返回总数。如果任务要处理的行数大于 **10**，那么，将任务拆分成两组，并创建两个 **DocumentTask** 对象来处理这两组对象。当这些任务执行结束后，同样合计返回的结果，并返回总数。

◆ **LineTask** 类：这个类的任务需要处理文档中一行的某一组词。如果一组词的个数小 **100**，那么任务将直接在这一组词里搜索特定词，然后返回查找词在这一组词中出现的次数。否则，任务将拆分这些词为两组，并创建两个 **LineTask** 对象来处理这两组词。当这些任务执行完成后，合计返回的结果，并返回总数。

在 **Main** 主类中，我们通过默认的构造器创建了 **ForkJoinPool** 对象，然后执行 **DocumentTask** 类，来处理一个共有 **100** 行，每行 **1,000** 字的文档。这个任务将问题拆分成 **DocumentTask** 对象和 **LineTask** 对象，然后当所有的任务执行完成后，使用原始的任务来获取整个文档中所要查找的词出现的次数。由于任务继承了 **RecursiveTask** 类，因此能够返回结果。

调用 **get()** 方法来获得 **Task** 返回的结果。这个方法声明在 **Future** 接口里，并由 **RecursiveTask** 类实现。

执行程序时，在控制台上，我们可以比较第一行与最后一行的输出信息。第一行是文档生成时被查找的词出现的次数，最后一行则是通过 **Fork / Join** 任务计算而来的被查找的词出现的次数，而且这两个数字相同。

更多信息

ForkJoinTask 类提供了另一个 **complete()** 方法来结束任务的执行并返回结果。这个方法接收一个对象，对象的类型就是 **RecursiveTask** 类的泛型参数，然后在任务调用 **join()** 方法后返回这个对象作为结果。这一过程采用了推荐的异步任务来返回任务的结果。

由于 **RecursiveTask** 类实现了 **Future** 接口，因此还有 **get()** 方法调用的其他版本：

◆ **get(long timeout, TimeUnit unit)**: 这个版本中，如果任务的结果未准备好，将等待指定的时间。如果等待时间超出，而结果仍未准备好，那方法就会返回 **null** 值。

TimeUnit 是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

参见

- ◆ 参见 5.2 节。
- ◆ 参见 8.5 节。

5.4 异步运行任务

在 **ForkJoinPool** 中执行 **ForkJoinTask** 时，可以采用同步或异步方式。当采用同步方式执行时，发送任务给 **Fork/Join** 线程池的方法直到任务执行完成后才会返回结果。而采用异步方式执行时，发送任务给执行器的方法将立即返回结果，但是任务仍能够继续执行。

需要明白这两种方式在执行任务时的一个很大的区别。当采用同步方式，调用这些方法（比如，**invokeAll()** 方法）时，任务被挂起，直到任务被发送到 **Fork/Join** 线程池中执行完成。这种方式允许 **ForkJoinPool** 类采用工作窃取算法（**Work-StealingAlgorithm**）来分配一个新任务给在执行休眠任务的工作者线程（**WorkerThread**）。相反，当采用异步方法（比如，**fork()** 方法）时，任务将继续执行，因此 **ForkJoinPool** 类无法使用工作窃取算法来提升应用程序的性能。在这个示例中，只有调用 **join()** 或 **get()** 方法来等待任务的结束时，**ForkJoinPool** 类才可以使用工作窃取算法。

本节将学习如何使用 **ForkJoinPool** 和 **ForkJoinTask** 类所提供的异步方法来管理任务。我们将实现一个程序：在一个文件夹及其子文件夹中来搜索带有指定扩展名的文件。**ForkJoinTask** 类将实现处理这个文件夹的内容。而对于这个文件夹中的每一个子文件，任务将以异步的方式发送一个新的任务给 **ForkJoinPool** 类。对于每个文件夹中的文件，任务将检查任务文件的扩展名，如果符合条件就将其增加到结果列表中。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **FolderProcessor** 的类，并继承 **RecursiveTask** 类，**RecursiveTask** 类的泛型参数为 **List<String>**类型。

```
public class FolderProcessor extends RecursiveTask<List<String>> {
```

2. 声明类的 **serialVersionUID** 属性。这个元素是必需的，因为 **RecursiveTask** 类的父类 **ForkJoinTask** 实现了 **Serializable** 接口。

```
private static final long serialVersionUID = 1L;
```

3. 声明一个名为 **path** 的私有 **String** 属性，用来存储任务将要处理的文件夹的完整路径。

```
private String path;
```

4. 声明一个名为 **extension** 的私有 **String** 属性，用来存储任务将要查找的文件的扩展名。

```
private String extension;
```

5. 实现类的构造器，用来初始化这些属性。

```
public FolderProcessor (String path, String extension) {
    this.path=path;
    this.extension=extension;
}
```

6. 实现 **compute()** 方法。既然指定了 **RecursiveTask** 类泛型参数为 **List<String>** 类型，那么，这个方法必须返回一个同样类型的对象。

```
@Override
protected List<String> compute() {
```

7. 声明一个名为 **list** 的 **String** 对象列表，用来存储文件夹中文件的名称。

```
List<String> list=new ArrayList<>();
```

8. 声明一个名为 **tasks** 的 **FolderProcessor** 任务列表，用来存储子任务，这些子任务将处理文件夹中的子文件夹。

```
List<FolderProcessor> tasks=new ArrayList<>();
```

9. 获取文件夹的内容。

```
File file=new File(path);
File content[] = file.listFiles();
```

10. 对于文件夹中的每一个元素，如果它是子文件夹，就创建一个新的 **FolderProcessor** 对象，然后调用 **fork()** 方法采用异步方式来执行它。

```
if (content != null) {
    for (int i = 0; i < content.length; i++) {
        if (content[i].isDirectory()) {
            FolderProcessor task=new FolderProcessor(content[i].
                getAbsolutePath(), extension);
            task.fork();
            tasks.add(task);
```

11. 否则，调用 **checkFile()** 方法来比较文件的扩展名。如果文件的扩展名与将要搜索的扩展名相同，就将文件的完整路径存储到第 7 步声明的列表中。

```
} else {
    if (checkFile(content[i].getName())){
        list.add(content[i].getAbsolutePath());
    }
}
```

12. 如果 **FolderProcessor** 子任务列表超过 **50** 个元素，那么就在控制台输出一条信息表示这种情景。

```
if (tasks.size()>50) {
    System.out.printf("%s: %d tasks ran.\n", file.
        getAbsolutePath(), tasks.size());
}
```

13. 调用 **addResultsFromTask()** 辅助方法。它把通过这个任务而启动的子任务返回的结果增加到文件列表中。传递两个参数给这个方法，一个是字符串列表 **list**，一个是 **FolderProcessor** 子任务列表 **tasks**。

```
addResultsFromTasks(list, tasks);
```

14. 返回字符串列表。

```
return list;
```

15. 实现 **addResultsFromTasks()** 方法。遍历任务列表中存储的每一个任务，调用 **join()** 方法等待任务执行结束，并且返回任务的结果。然后，调用 **addAll()** 方法将任务的结果增加到字符串列表中。

```
private void addResultsFromTasks(List<String> list,
    List<FolderProcessor> tasks) {
    for (FolderProcessor item: tasks) {
        list.addAll(item.join());
    }
}
```

16. 实现 **checkFile()** 方法。这个方法检查作为参数而传递进来的文件名，如果是以正在搜索的文件扩展名为结尾，那么方法就返回 **true**，否则就返回 **false**。

```
private boolean checkFile(String name) {
    return name.endsWith(extension);
}
```

17. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

18. 通过默认的构造器创建 **ForkJoinPool** 线程池。

```
ForkJoinPool pool=new ForkJoinPool();
```

19. 创建 3 个 **FolderProcessor** 任务，并使用不同的文件夹路径来初始化这些任务。

```
FolderProcessor system=new FolderProcessor("C:\\Windows",
    "log");
FolderProcessor apps=new
FolderProcessor("C:\\Program Files","log");
FolderProcessor documents=new FolderProcessor("C:\\Documents
    And Settings","log");
```

20. 调用 **execute()** 方法执行线程池里的 3 个任务。

```
pool.execute(system);
pool.execute(apps);
pool.execute(documents);
```

21. 在控制台上每隔 1 秒钟输出线程池的状态信息，直到这 3 个任务执行结束。

```
do {
    System.out.printf("*****\n");
    System.out.printf("Main: Parallelism: %d\n",pool.
        getParallelism());
    System.out.printf("Main: Active Threads: %d\n",pool.
        getActiveThreadCount());
    System.out.printf("Main: Task Count: %d\n",pool.
        getQueuedTaskCount());
    System.out.printf("Main: Steal Count: %d\n",pool.
        getStealCount());
    System.out.printf("*****\n");
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
} while ((!system.isDone()) || (!apps.isDone()) || (!documents.
    isDone()));
```

22. 调用 shutdown()方法关闭 ForkJoinPool 线程池。

```
pool.shutdown();
```

23. 在控制台输出每一个任务产生的结果的大小。

```
List<String> results;
results=system.join();
System.out.printf("System: %d files found.\n",results.size());
results=apps.join();
System.out.printf("Apps: %d files found.\n",results.size());
results=documents.join();
System.out.printf("Documents: %d files found.\n",results.
size());
```

工作原理

下面的截图显示了范例的部分运行结果。

```
<terminated> Main (36) [Java Application] E:\Java 7 Concurrency CookB*****
Main: Parallelism: 2
Main: Active Threads: 11
Main: Task Count: 223
Main: Steal Count: 10776
*****
*****
Main: Parallelism: 2
Main: Active Threads: 9
Main: Task Count: 34
Main: Steal Count: 10872
*****
*****
System: 528 files found.
Apps: 129 files found.
Documents: 110 files found.
```

这个范例的重点在于 **FolderProcessor** 类。每一个任务处理一个文件夹中的内容。文件夹中的内容有以下两种类型的元素：

- ◆ 文件；

◆ 其他文件夹。

如果主任务发现一个文件夹，它将创建另一个**Task**对象来处理这个文件夹，调用**fork()**方法把这个新对象发送到线程池中。**fork()**方法发送任务到线程池时，如果线程池中有空闲的工作者线程(**WorkerThread**)或者将创建一个新的线程，那么开始执行这个任务，**fork()**方法会立即返回，因此，主任务可以继续处理文件夹里的其他内容。对于每一个文件，任务开始比较它的文件扩展名，如果与要搜索的扩展名相同，那么将文件的完整路径增加到结果列表中。

一旦主任务处理完指定文件夹里的所有内容，它将调用**join()**方法等待发送到线程池中的所有子任务执行完成。**join()**方法在主任务中被调用，然后等待任务执行结束，并通过**compute()**方法返回值。主任务将所有的子任务结果进行合并，这些子任务发送到线程池中时带有自己的结果列表，然后通过调用**compute()**方法返回这个列表并作为主任务的返回值。

ForkJoinPool类也允许以异步的方式执行任务。调用**execute()**方法发送3个初始任务到线程池中。在**Main**主类中，调用**shutdown()**方法结束线程池，并在控制台输出线程池中任务的状态及其变化的过程。**ForkJoinPool**类包含了多个方法可以实现这个目的。参考8.5节来查阅这些方法的详细列表。

更多信息

本范例使用**join()**方法来等待任务的结束，然后获取它们的结果。也可以使用**get()**方法以下的两个版本来完成这个目的。

◆ **get()**: 如果**ForkJoinTask**类执行结束，或者一直等到结束，那么**get()**方法的这个版本则返回由**compute()**方法返回的结果。

◆ **get(long timeout, TimeUnit unit)**: 如果任务的结果未准备好，那么**get()**方法的这个版本将等待指定的时间。如果超过指定的时间了，任务的结果仍未准备好，那么这个方法将返回**null**值。**TimeUnit**是一个枚举类，有如下的常量：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS**和**SECONDS**。

get()方法和**join()**方法还存在两个主要的区别：

◆ **join()**方法不能被中断，如果中断调用**join()**方法的线程，方法将抛出**InterruptedException**异常；

◆ 如果任务抛出任何运行时异常，那么**get()**方法将返回**ExecutionException**异常，

但是 **join()**方法将返回 **RuntimeException** 异常。

参见

- ◆ 参考 5.2 节。
- ◆ 参考 8.5 节。

5.5 在任务中抛出异常

Java 有两种类型的异常。

- ◆ 非运行时异常 (**Checked Exception**)：这些异常必须在方法上通过 **throws** 子句抛出，或者在方法体内通过 **try{...}catch{...}** 方式进行捕捉处理。比如 **IOException** 或 **ClassNotFoundException** 异常。
- ◆ 运行时异常 (**Unchecked Exception**)：这些异常不需要在方法上通过 **throws** 子句抛出，也不需要在方法体内通过 **try{...}catch{...}** 方式进行捕捉处理。比如 **NumberFormatException** 异常。

不能在 **ForkJoinTask** 类的 **compute()** 方法中抛出任务非运行时异常，因为这个方法的实现没有包含任何 **throws** 声明。因此，需要包含必需的代码来处理相关的异常。另一方面，**compute()** 方法可以抛出运行时异常（它可以是任何方法或者方法内的对象抛出的异常）。**ForkJoinTask** 类和 **ForkJoinPool** 类的行为与我们期待的可能不同。在控制台上，程序没有结束执行，不能看到任务异常信息。如果异常不被抛出，那么它只是简单地将异常吞噬掉。然而，我们能够利用 **ForkJoinTask** 类的一些方法来获知任务是否有异常抛出，以及抛出哪一种类型的异常。在本节，我们将学习如何获取这些异常信息。

准备工作

本节的范例是在 **Eclipse IDE** 里完成的。无论你使用 **Eclipse** 还是其他的 **IDE**（比如 **NetBeans**），都可以打开这个 **IDE** 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Task** 的类，并继承 **RecursiveTask** 类，**RecursiveTask** 类的泛型参数为 **Integer** 类型。

```
public class Task extends RecursiveTask<Integer> {
```

2. 声明一个名为 **array** 的私有 **int** 数组。用来模拟在这个范例中即将处理的数据数组。

```
private int array[];
```

3. 声明两个分别名为 **start** 和 **end** 的私有 **int** 属性。这些属性将决定任务要处理的数据组元素。

```
private int start, end;
```

4. 实现类的构造器，用来初始化类的属性。

```
public Task(int array[], int start, int end) {
    this.array=array;
    this.start=start;
    this.end=end;
}
```

5. 实现任务的 **compute()** 方法。由于指定了 **Integer** 类型作为 **RecursiveTask** 的泛型类型，因此这个方法必须返回一个 **Integer** 对象。在控制台输出 **start** 和 **end** 属性。

```
@Override
protected Integer compute() {
    System.out.printf("Task: Start from %d to %d\n", start, end);
```

6. 如果由 **start** 和 **end** 属性所决定的元素块规模小于 **10**，那么直接检查元素，当碰到元素块的第 **4** 个元素（索引位为 **3**）时，就抛出 **RuntimeException** 异常。然后将任务休眠 **1** 秒钟。

```
if (end-start<10) {
    if ((3>start)&&(3<end)){
        throw new RuntimeException("This task throws an"+
        "Exception: Task from "+start+" to "+end);
    }
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
```

```
    e.printStackTrace();
}
```

7. 如果要处理的元素块规模大于或等于 **10**, 就拆分这个元素块为两部分, 并创建两个 **Task** 对象来处理这两部分, 然后调用 **invokeAll()**方法在线程池中执行这两个 **Task** 对象。

```
} else {
    int mid=(end+start)/2;
    Task task1=new Task(array,start,mid);
    Task task2=new Task(array,mid,end);
    invokeAll(task1, task2);
}
```

8. 在控制台输出信息, 表示任务结束, 并输出 **start** 和 **end** 属性值。

```
System.out.printf("Task: End form %d to %d\n",start,end);
```

9. 返回数字 **0** 作为任务的结果。

```
return 0;
```

10. 实现范例的主类, 创建 **Main** 主类, 并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

11. 创建一个名为 **array** 并能容纳 **100** 个整数的 **int** 数组。

```
int array[]=new int[100];
```

12. 创建一个 **Task** 对象来处理这个数组。

```
Task task=new Task(array,0,100);
```

13. 通过默认的构造器创建 **ForkJoinPool** 对象。

```
ForkJoinPool pool=new ForkJoinPool();
```

14. 调用 **execute()**方法在线程池中执行任务。

```
pool.execute(task);
```

15. 调用 **shutdown()**方法关闭线程池。

```
pool.shutdown();
```

16. 调用 **awaitTermination()**方法等待任务执行结束。如果想一直等待到任务执行完成，那就传递值 **1** 和 **TimeUnit.DAYS** 作为参数给这个方法。

```
try {
    pool.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

17. 调用 **isCompletedAbnormally()**方法来检查主任务或者它的子任务之一是否抛出了异常。在这个示例中，在控制台输出信息就表示有异常抛出。调用 **ForkJoinTask** 类的 **getException()**方法来获取异常信息。

```
if (task.isCompletedAbnormally()) {
    System.out.printf("Main: An exception has occurred\n");
    System.out.printf("Main: %s\n", task.getException());
}
System.out.printf("Main: Result: %d", task.join());
```

工作原理

在本节，我们实现的 **Task** 类用来处理一个数字数组。它检查要处理的数字块规模是否包含有 **10** 个或更多个元素。在这个情况下，**Task** 类拆分这个数字块为两部分，然后创建两个新的 **Task** 对象用来处理这两部分。否则，它将寻找位于数组中第 **4** 个位置（索引位为 **3**）的元素。如果这个元素位于任务处理块中，它将抛出一个 **RuntimeException** 异常。

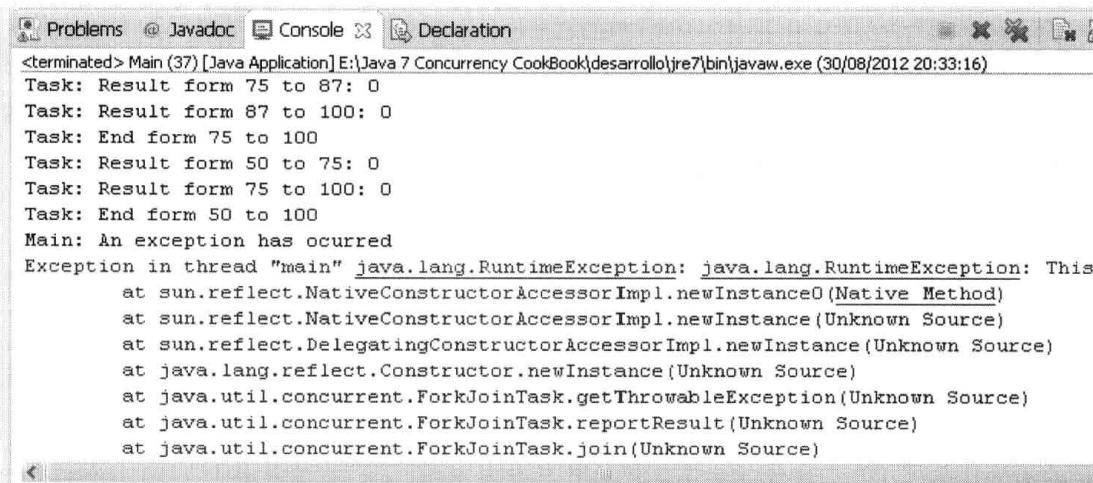
虽然运行这个程序时将抛出异常，但是程序不会停止。在 **Main** 主类中，调用原始任务 **ForkJoinTask** 类的 **isCompletedAbnormally()**方法，如果主任务或者它的子任务之一抛出了异常，这个方法将返回 **true**。也可以使用 **getException()**方法来获得抛出的 **Exception** 对象。

当任务抛出运行时异常时，会影响它的父任务（发送到 **ForkJoinPool** 类的任务），以及父任务的父任务，以此类推。查阅程序的输出结果，将会发现有一些任务没有结束的信息。那些任务的开始信息如下：

```
Task: Starting form 0 to 100
Task: Starting form 0 to 50
Task: Starting form 0 to 25
Task: Starting form 0 to 12
Task: Starting form 0 to 6
```

这些任务是那些抛出异常的任务和它的父任务。所有这些任务都是异常结束的。记住一点：在用 **ForkJoinPool** 对象和 **ForkJoinTask** 对象开发一个程序时，它们是会抛出异常的，如果不想要这种行为，就得采用其他方式。

下面的截屏展示了这个范例运行的部分结果。



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```
<terminated> Main (37) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\jre7\bin\javaw.exe (30/08/2012 20:33:16)
Task: Result form 75 to 87: 0
Task: Result form 87 to 100: 0
Task: End form 75 to 100
Task: Result form 50 to 75: 0
Task: Result form 75 to 100: 0
Task: End form 50 to 100
Main: An exception has occurred
Exception in thread "main" java.lang.RuntimeException: java.lang.RuntimeException: This
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at java.util.concurrent.ForkJoinTask.getThrowableException(Unknown Source)
    at java.util.concurrent.ForkJoinTask.reportResult(Unknown Source)
    at java.util.concurrent.ForkJoinTask.join(Unknown Source)
```

更多信息

在范例中，不采用抛出异常，而调用 **ForkJoinTask** 类的 **completeExceptionally()**方法也可以获得同样的结果。代码如下所示：

```
Exception e=new Exception("This task throws an Exception: "+ "Task
from "+start+" to "+end);
completeExceptionally(e);
```

参见

参见 5.2 节。

5.6 取消任务

在 **ForkJoinPool** 类中执行 **ForkJoinTask** 对象时，在任务开始执行前可以取消它。**ForkJoinTask** 类提供了 **cancel()** 方法来达到取消任务的目的。在取消一个任务时必须要注意以下两点：

- ◆ **ForkJoinPool** 类不提供任何方法来取消线程池中正在运行或者等待运行的所有任务；
- ◆ 取消任务时，不能取消已经被执行的任务。

在本节，我们将实现一个取消 **ForkJoinTask** 对象的范例。该范例将寻找数组中某个数字所处的位置。第一个任务是寻找可以被取消的剩余任务数。由于 **Fork / Join** 框架没有提供取消功能，我们将创建一个辅助类来实现取消任务的操作。

准备工作

本节的范例是在 EclipseIDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **ArrayGenerator** 的类。这个类将生成一个指定大小的随机整数数组。实现 **generateArray()** 方法，它将生成数字数组，接收一个 **int** 参数表示数组的大小。

```
public class ArrayGenerator {
    public int[] generateArray(int size) {
        int array[] = new int[size];
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10);
        }
        return array;
    }
}
```

2. 创建一个名为 **TaskManager** 的类。本示例将利用这个类来存储在 **ForkJoinPool** 中执行的任务。由于 **ForkJoinPool** 和 **ForkJoinTask** 类的局限性，将利用 **TaskManager**

类来取消 **ForkJoinPool** 类中所有的任务。

```
public class TaskManager {
```

3. 声明一个名为 **tasks** 的对象列表，带有 **ForkJoinTask** 泛型参数，并且 **ForkJoinTask** 带有 **Integer** 泛型参数。

```
private List<ForkJoinTask<Integer>> tasks;
```

4. 实现类的构造器，用来初始化任务列表。

```
public TaskManager() {
    tasks=new ArrayList<>();
}
```

5. 实现 **addTask()**方法。增加一个 **ForkJoinTask** 对象到任务列表中。

```
public void addTask(ForkJoinTask<Integer> task) {
    tasks.add(task);
}
```

6. 实现 **cancelTasks()**方法。遍历存储在列表中的所有 **ForkJoinTask** 对象，然后调用 **cancel()**方法取消之。**cancelTasks()**方法接收一个要取消剩余任务的 **ForkJoinTask** 对象作为参数，然后取消所有的任务。

```
public void cancelTasks(ForkJoinTask<Integer> cancelTask) {
    for (ForkJoinTask<Integer> task : tasks) {
        if (task!=cancelTask) {
            task.cancel(true);
            ((SearchNumberTask)task).writeCancelMessage();
        }
    }
}
```

7. 实现 **SearchNumberTask** 类，并继承 **RecursiveTask** 类，**RecursiveTask** 类的泛型参数为 **Integer** 类型。这个类将寻找在整数数组元素块中的一个数字。

```
public class SearchNumberTask extends RecursiveTask<Integer> {
```

8. 声明一个名为 **array** 的私有 **int** 数组。

```
private int numbers[];
```

9. 声明两个分别名为 **start** 和 **end** 的私有 **int** 属性。这两个属性将决定任务所要处理的数组的元素。

```
private int start, end;
```

10. 声明一个名为 **number** 的私有 **int** 属性，用来存储将要寻找的数字。

```
private int number;
```

11. 声明一个名为 **manager** 的私有 **TaskManager** 属性。利用这个对象来取消所有的任务。

```
private TaskManager manager;
```

12. 声明一个 **int** 常量，并初始化其值为**-1**。当任务找不到数字时将返回这个常量。

```
private final static int NOT_FOUND=-1;
```

13. 实现类的构造器，用来初始化它的属性。

```
public Task (int numbers[], int start, int end, int number,
TaskManager manager){
this.numbers=numbers;
this.start=start;
this.end=end;
this.number=number;
this.manager=manager;
}
```

14. 实现 **compute()**方法。在控制台输出信息表示任务开始，并输出 **start** 和 **end** 的属性值。

```
@Override
protected Integer compute() {
    System.out.println("Task: "+start+":"+end);
```

15. 如果 **start** 和 **end** 属性值的差异大于 **10**（任务必须处理大于 **10** 个元素的数组），那么，就调用 **launchTasks()**方法将这个任务拆分为两个子任务。

```
int ret;
if (end-start>10) {
    ret=launchTasks();
```

16. 否则，寻找在数组块中的数字，调用 **lookForNumber()**方法处理这个任务。

```
    } else {
        ret=lookForNumber();
    }
}
```

17. 返回任务的结果。

```
return ret;
```

18. 实现 **lookForNumber()**方法。

```
private int lookForNumber() {
```

19. 遍历任务所要处理的数组块中的所有元素，将元素中存储的数字和将要寻找的数字进行比较。如果它们相等，就在控制台输出信息表示找到了，并用 **TaskManager** 对象的 **cancelTasks()**方法取消所有的任务，然后返回已找到的这个元素所在的位置。

```
for (int i=start; i<end; i++){
    if (numbers[i]==number) {
        System.out.printf("Task: Number %d found in position
                           %d\n", number, i);
        manager.cancelTasks(this);
        return i;
    }
}
```

20. 在循环体中，将任务休眠 1 秒钟。

```
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
```

21. 返回-1 表示没有找到元素。

```
return NOT_FOUND;
}
```

22. 实现 **launchTasks()**方法。将这个任务要处理的元素块拆分成两部分，然后创建两个 **Task** 对象来处理它们。

```
private int launchTasks() {
    int mid=(start+end)/2;
    Task task1=new Task(array,start,mid,number,manager);
    Task task2=new Task(array,mid,end,number,manager);
```

23. 增加任务到 **TaskManager** 对象中。

```
manager.addTask(task1);
manager.addTask(task2);
```

24. 调用 **fork()**方法采用异步方式执行这两个任务。

```
task1.fork();
task2.fork();
```

25. 等待任务结束，如果第一个任务返回的结果不为**-1**，则返回第一个任务的结果；否则返回第二个任务的结果。

```
int returnValue;
returnValue=task1.join();
if (returnValue!=-1) {
    return returnValue;
}
returnValue=task2.join();
return returnValue;
```

26. 实现 **writeCancelMessage()**方法，在控制台输入信息表示任务已经取消了。

```
public void writeCancelMessage(){
    System.out.printf("Task: Cancelled task from %d to
                      %d",start,end);
}
```

27. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

28. 用 **ArrayGenerator** 类创建一个容量为 **1,000** 的数字数组。

```
ArrayGenerator generator=new ArrayGenerator();
int array[]=generator.generateArray(1000);
```

29. 创建一个 **TaskManager** 对象。

```
TaskManager manager=new TaskManager();
```

30. 通过默认的构造器创建一个 **ForkJoinPool** 对象。

```
ForkJoinPool pool=new ForkJoinPool();
```

31. 创建一个 **Task** 对象用来处理第 28 步生成的数组。

```
Task task=new Task (array,0,1000,5,manager);
```

32. 调用 **execute()**方法采用异步方式执行线程池中的任务。

```
pool.execute(task);
```

33. 调用 **shutdown()**方法关闭线程池。

```
pool.shutdown();
```

34. 调用 **ForkJoinPool** 类的 **awaitTermination()**方法等待任务执行结束。

```
try {
    pool.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

35. 在控制台输出信息表示程序结束。

```
System.out.printf("Main: The program has finished\n");
```

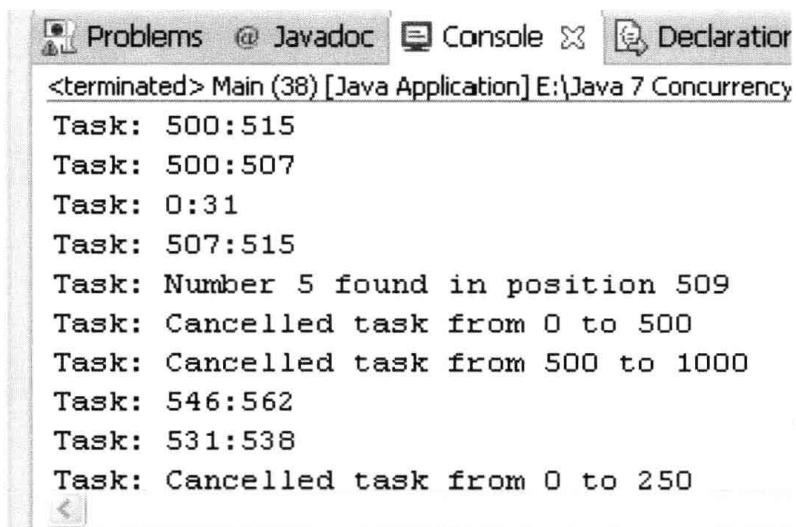
工作原理

ForkJoinTask 类提供的 **cancel()**方法允许取消一个仍没有被执行的任务，这是非常重要的一点。如果任务已经开始执行，那么调用 **cancel()**方法也无法取消。这个方法接收一个名为 **mayInterruptIfRunning** 的 **boolean** 值参数。顾名思义，如果传递 **true** 值给这个方法，即使任务正在运行也将被取消。JavaAPI 文档指出，**ForkJoinTask** 类的默认实现，这个属性没有起作用。如果任务还没有开始执行，那么这些任务将被取消。任务的取消对于已发送到线程池中的任务没有影响，它们将继续执行。

Fork / Join 框架的局限性在于, **ForkJoinPool** 线程池中的任务不允许被取消。为了克服这种局限性, 我们实现了 **TaskManager** 类, 它存储发送到线程池中的所有任务, 可以用一个方法来取消存储的所有任务。如果任务正在运行或者已经执行结束, 那么任务就不能被取消, **cancel()**方法返回 **false** 值, 因此可以尝试去取消所有的任务而不用担心可能带来的间接影响。

这个范例实现在数字数组中寻找一个数字。根据 **Fork / Join** 框架的推荐, 我们将问题拆分为更小的子问题。由于我们仅关心数字的一次出现, 因此, 当找到它时, 就会取消其他的所有任务。

下面的截图展示了范例执行的部分结果。



The screenshot shows a Java application running in an IDE's console window. The title bar indicates it's a Java Application named Main (38). The console output shows several tasks being processed:

```
<terminated> Main (38) [Java Application] E:\Java 7 Concurrency
Task: 500:515
Task: 500:507
Task: 0:31
Task: 507:515
Task: Number 5 found in position 509
Task: Cancelled task from 0 to 500
Task: Cancelled task from 500 to 1000
Task: 546:562
Task: 531:538
Task: Cancelled task from 0 to 250
```

参见

- ◆ 参见 5.2 节。

第 6 章

并发集合

本章将介绍下列内容：

- ◆ 使用非阻塞式线程安全列表
- ◆ 使用阻塞式线程安全列表
- ◆ 使用按优先级排序的阻塞式线程安全列表
- ◆ 使用带有延迟元素的线程安全列表
- ◆ 使用线程安全可遍历映射
- ◆ 生成并发随机数
- ◆ 使用原子变量
- ◆ 使用原子数组

6.1 简介

数据结构（**Data Structure**）是编程中的基本元素，几乎每个程序都使用一种或多种数据结构来存储和管理数据。Java API 提供了包含接口、类和算法的 **Java 集合框架**（**Java Collection Framework**），它实现了可用在程序中的大量数据结构。

当需要在并发程序中使用数据集合时，必须要谨慎地选择相应的实现方式。大多数集合类不能直接用于并发应用，因为它们没有对本身数据的并发访问进行控制。如果一些并发任务共享了一个不适用于并发任务的数据结构，将会遇到数据不一致的错误，并将影响

程序的准确运行。这类数据结构的一个例子是 **ArrayList** 类。

Java 提供了一些可以用于并发程序中的数据集合，它们不会引起任何问题。一般来说，Java 提供了两类适用于并发应用的集合。

◆ **阻塞式集合 (Blocking Collection)**: 这类集合包括添加和移除数据的方法。当集合已满或为空时，被调用的添加或者移除方法就不能立即被执行，那么调用这个方法的线程将被阻塞，一直到该方法可以被成功执行。

◆ **非阻塞式集合 (Non-Blocking Collection)**: 这类集合也包括添加和移除数据的方法。如果方法不能立即被执行，则返回 `null` 或抛出异常，但是调用这个方法的线程不会被阻塞。

通过本章的各个小节，你将学会如何在并发应用中使用一些 Java 集合。

- ◆ 非阻塞式列表对应的实现类：**ConcurrentLinkedDeque** 类；
- ◆ 阻塞式列表对应的实现类：**LinkedBlockingDeque** 类；
- ◆ 用于数据生成或消费的阻塞式列表对应的实现类：**LinkedTransferQueue** 类；
- ◆ 按优先级排序列表元素的阻塞式列表对应的实现类：**PriorityBlockingQueue** 类；
- ◆ 带有延迟列表元素的阻塞式列表对应的实现类：**DelayQueue** 类；
- ◆ 非阻塞式可遍历映射对应的实现类：**ConcurrentSkipListMap** 类；
- ◆ 随机数字对应的实现类：**ThreadLocalRandom** 类；
- ◆ 原子变量对应的实现类：**AtomicLong** 和 **AtomicIntegerArray** 类。

6.2 使用非阻塞式线程安全列表

最基本的集合类型是列表 (**List**)。一个列表包含的元素数量不定，可以在任何位置添加、读取或移除元素。并发列表允许不同的线程在同一时间添加或移除列表中的元素，而不会造成数据不一致。

在本节，将会学到如何在并发程序中使用非阻塞式列表。非阻塞式列表提供了一些操作，如果被执行的操作不能够立即运行（例如，在列表为空时，从列表取出一个元素），方法会抛出异常或返回 `null`。Java 7 引入了 **ConcurrentLinkedDeque** 类来实现非阻塞式并发列表。

将要实现的范例包括以下两个不同的任务：

- ◆ 添加大量的数据到一个列表中；
- ◆ 从同一个列表中移除大量的数据。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **AddTask** 的类，实现 **Runnable** 接口。

```
public class AddTask implements Runnable {
```

2. 声明一个私有的 **ConcurrentLinkedDeque** 属性 **list**，并指定它的泛型参数是 **String** 型的。

```
private ConcurrentLinkedDeque<String> list;
```

3. 实现类的构造器来初始化属性。

```
public AddTask(ConcurrentLinkedDeque<String> list) {  
    this.list=list;  
}
```

4. 实现 **run()** 方法。这个方法将 10,000 个字符串存放到列表中，这些字符串由当前执行任务的线程的名称和数字组成。

```
@Override  
public void run() {  
    String name=Thread.currentThread().getName();  
    for (int i=0; i<10000; i++){  
        list.add(name+": Element "+i);  
    }  
}
```

5. 创建名为 **PollTask** 的类，并实现 **Runnable** 接口。

```
public class PollTask implements Runnable {
```

6. 声明一个私有的 **ConcurrentLinkedDeque** 属性 **list**，并指定它的泛型参数是 **String** 型的。

```
private ConcurrentLinkedDeque<String> list;
```

7. 实现类的构造器来初始化属性。

```
public PollTask(ConcurrentLinkedDeque<String> list) {
    this.list=list;
}
```

8. 实现 **run()** 方法。这个方法将列表中的 10,000 个字符串取出，总共取 5,000 次，每次取两个元素。

```
@Override
public void run() {
    for (int i=0; i<5000; i++) {
        list.pollFirst();
        list.pollLast();
    }
}
```

9. 创建范例的主类 **Main**，并添加 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

10. 创建 **ConcurrentLinkedDeque** 对象，并指定它的泛型参数是 **String** 型的。

```
ConcurrentLinkedDeque<String> list=new
ConcurrentLinkedDeque<>();
```

11. 创建线程数组 **threads**，它包含 100 个线程。

```
Thread threads[]=new Thread[100];
```

12. 创建 100 个 **AddTask** 对象及其对应的运行线程。将每个线程存放到上一步创建的数组中，然后启动线程。

```
for (int i=0; i<threads.length ; i++) {
    AddTask task=new AddTask(list);
    threads[i]=new Thread(task);
    threads[i].start();
}
System.out.printf("Main: %d AddTask threads have been
launched\n",threads.length);
```

13. 使用 **join()**方法等待线程完成。

```
for (int i=0; i<threads.length; i++) {
    try {
        threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

14. 将列表的元素数量打印到控制台。

```
System.out.printf("Main: Size of the List: %d\n",list.size());
```

15. 创建 100 个 **PollTask** 对象及其对应的运行线程。将每个线程存放到上一步创建的数组中，然后启动线程。

```
for (int i=0; i< threads.length; i++) {
    PollTask task=new PollTask(list);
    threads[i]=new Thread(task);
    threads[i].start();
}
System.out.printf("Main: %d PollTask threads have been
launched\n",threads.length);
```

16. 使用 **join()**方法等待线程完成。

```
for (int i=0; i<threads.length; i++) {
    try {
        threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

17. 将列表的元素数量打印到控制台。

```
System.out.printf ("Main: Size of the List: %d\n", list.size());
```

工作原理

本节使用的泛型参数是 **String** 类的 **ConcurrentLinkedDeque** 对象，用来实现一个非阻塞式并发数据列表。下面的截屏显示了程序的运行结果。

```

Problems @ Javadoc Console Declaration Search
<terminated> Main (63) [Java Application] E:\Java 7 Concurrency CookBook\desa
Main: 100 AddTask threads have been launched
Main: Size of the List: 1000000
Main: 100 PollTask threads have been launched
Main: Size of the List: 0

```

首先，执行 100 个 **AddTask** 任务将元素添加到 **ConcurrentLinkedDeque** 对象 **list** 中。每个任务使用 **add()** 方法向这个列表中插入 10,000 个元素。**add()** 方法将新元素添加到列表尾部。当所有任务运行完毕，列表中的元素数量将被打印到控制台。在这一刻，列表中有 1,000,000 个元素。

接下来，执行 100 个 **PollTask** 任务将元素从列表中移除。每个任务使用 **pollFirst()** 和 **pollLast()** 方法从列表中移除 10,000 个元素。**pollFirst()** 方法返回并移除列表中的第一个元素，**pollLast()** 方法返回并移除列表中的最后一个元素。如果列表为空，这些方法返回 **null**。当所有任务运行完毕，列表中的元素数量将被打印到控制台。在这一刻，列表中有 0 个元素。

使用 **size()** 方法输出列表中的元素数量。需要注意的是，这个方法返回的值可能不是真实的，尤其当有线程在添加数据或移除数据时，这个方法需要遍历整个列表来计算元素数量，而遍历过的数据可能已经改变。仅当没有任何线程修改列表时，才能保证返回的结果是准确的。

更多信息

ConcurrentLinkedDeque 类提供了其他从列表中读取数据的方法。

- ◆ **getFirst()**和**getLast()**: 分别返回列表中第一个和最后一个元素，返回的元素不会从列表中移除。如果列表为空，这两个方法抛出 **NoSuchElementException** 异常。
- ◆ **peek()**、**peekFirst()**和**peekLast()**: 分别返回列表中第一个和最后一个元素，返回的元素不会从列表中移除。如果列表为空，这些方法返回 **null**。
- ◆ **remove()**、**removeFirst()**和**removeLast()**: 分别返回列表中第一个和最后一个元素，返回的元素将会从列表中移除。如果列表为空，这些方法抛出 **NoSuchElementException** 异常。

6.3 使用阻塞式线程安全列表

最基本的集合类型是列表。一个列表包含的元素数量不定，可以在任何位置添加、读取或移除元素。并发列表允许不同的线程在同一时间添加或移除列表中的元素，而不会造成数据不一致。

在本节，你会学到如何在并发程序中使用阻塞式列表。阻塞式列表与非阻塞式列表的主要差别是：阻塞式列表在插入和删除操作时，如果列表已满或为空，操作不会被立即执行，而是将调用这个操作的线程阻塞队列直到操作可以执行成功。Java 引入了 **LinkedBlockingDeque** 类来实现阻塞式列表。

将要实现的范例包括以下两个不同的任务：

- ◆ 添加大量的数据到一个列表中；
- ◆ 从同一个列表中移除大量的数据。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Client** 的类，并实现 **Runnable** 接口。

```
public class Client implements Runnable{
```

2. 声明一个私有的 **LinkedBlockingDeque** 属性 **requestList**，并指定它的泛型参数是 **String** 型的。

```
private LinkedBlockingDeque<String> requestList;
```

3. 实现类的构造器来初始化属性。

```
public Client (LinkedBlockingDeque<String> requestList) {
    this.requestList=requestList;
}
```

4. 实现 **run()** 方法。使用 **requestList** 对象的 **put()** 方法，每两秒向列表 **requestList** 中插入 5 个字符串。重复 3 次。

```
@Override
public void run() {
    for (int i=0; i<3; i++) {
        for (int j=0; j<5; j++) {
            StringBuilder request=new StringBuilder();
            request.append(i);
            request.append(":");
            request.append(j);
            try {
                requestList.put(request.toString());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.printf("Client: %s at %s.\n",request,new
Date());
        }
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.printf("Client: End.\n");
}
```

5. 创建范例的主类 **Main**，并添加 **main()** 方法。

```
public class Main {
```

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

6. 声明并创建 **LinkedBlockingDeque** 属性 **list**, 并指定它的泛型参数是 **String** 型的。

```
LinkedBlockingDeque<String> list=new LinkedBlockingDeque<>(3);
```

7. 将 **client** 作为传入参数创建线程 **Thread** 并启动。

```
Client client=new Client(list);
Thread thread=new Thread(client);
thread.start();
```

8. 使用 **list** 对象的 **take()**方法, 每 300 毫秒从列表中取出 3 个字符串对象, 重复 5 次。在控制台输出字符串。

```
for (int i=0; i<5 ; i++) {
    for (int j=0; j<3; j++) {
        String request=list.take();
        System.out.printf("Main: Request: %s at %s. Size:
        %d\n",request,new Date(),list.size());
    }
    TimeUnit.MILLISECONDS.sleep(300);
}
```

9. 输出一条表示程序结束的消息。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

本节使用的泛型参数是 **String** 的 **LinkedBlockingDeque** 对象, 用来实现一个阻塞式并发数据列表。

Client 类使用 **put()**方法将字符串插入到列表中。如果列表已满 (列表生成时指定了固定的容量), 调用这个方法的线程将被阻塞直到列表中有了可用的空间。

Main 类使用 **take()**方法从列表中取字符串。如果列表为空, 调用这个方法的线程将被阻塞直到列表不为空 (即有可用的元素)。

这个例子中使用了 **LinkedBlockingDeque** 对象的两个方法, 调用它们的线程可能会被阻塞, 在阻塞时如果线程被中断, 方法会抛出 **InterruptedException** 异常, 所以必须捕获

和处理这个异常。

更多信息

LinkedBlockingDeque 类也提供了其他存取元素的方法，这些方法不会引起阻塞，而是抛出异常或返回 **null**。

- ◆ **takeFirst()** 和 **takeLast()**: 分别返回列表中第一个和最后一个元素，返回的元素会从列表中移除。如果列表为空，调用方法的线程将被阻塞直到列表中有可用的元素出现。
- ◆ **getFirst()** 和 **getLast()**: 分别返回列表中第一个和最后一个元素，返回的元素不会从列表中移除。如果列表为空，则抛出 **NoSuchElementException** 异常。
- ◆ **peek()**、**peekFirst()** 和 **peekLast()**: 分别返回列表中第一个和最后一个元素，返回的元素不会从列表中移除。如果列表为空，返回 **null**。
- ◆ **poll()**、**pollFirst()** 和 **pollLast()**: 分别返回列表中第一个和最后一个元素，返回的元素将会从列表中移除。如果列表为空，返回 **null**。
- ◆ **add()**、**addFirst()** 和 **addLast()**: 分别将元素添加到列表中第一位和最后一位。如果列表已满(列表生成时指定了固定的容量)，这些方法将抛出 **IllegalStateException** 异常。

参见

- ◆ 参见 6.3 节。

6.4 使用按优先级排序的阻塞式线程安全列表

数据结构应用中的一个经典需求是实现一个有序列表。Java 引入了 **PriorityBlockingQueue** 类来满足这类需求。

所有添加进 **PriorityBlockingQueue** 的元素必须实现 **Comparable** 接口。这个接口提供了 **compareTo()** 方法，它的传入参数是一个同类型的对象。这样就有了两个同类型的对象并且相互比较：其中一个是执行这个方法的对象，另一个是参数传入的对象。这个方法必须返回一个数字值，如果当前对象小于参数传入的对象，那么返回一个小于 0 的值；如果当前对象大于参数传入的对象，那么返回一个大于 0 的值；如果两个对象相等就返回 0。

当插入元素时，**PriorityBlockingQueue** 使用 **compareTo()** 方法来决定插入元素的位置。

元素越大越靠后。

PriorityBlockingQueue 的另一个重要的特性是：它是阻塞式数据结构（**Blocking Data Structure**）。当它的方法被调用并且不能立即执行时，调用这个方法的线程将被阻塞直到方法执行成功。

在本节，你将学习如何使用 **PriorityBlockingQueue** 类。在范例中我们将大量不同优先级的事件存放到同一个列表中，并且检查队列是否按预期排序。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Event** 的类并实现 **Comparable** 接口，指定 **Comparable** 接口的泛型参数是 **Event** 类。

```
public class Event implements Comparable<Event> {
```

2. 声明一个私有 **int** 属性，名为 **thread**，来存放创建了 **event** 的线程数。

```
private int thread;
```

3. 声明一个私有 **int** 属性，名为 **priority**，来存放 **event** 的优先级。

```
private int priority;
```

4. 实现类的构造器来初始化属性。

```
public Event(int thread, int priority){  
    this.thread=thread;  
    this.priority=priority;  
}
```

5. 实现 **getThread()** 方法，返回 **thread** 属性。

```
public int getThread() {
```

```

        return thread;
    }
}

```

6. 实现 **getPriority()**方法，返回 **priority** 属性。

```

public int getPriority() {
    return priority;
}
}

```

7. 实现 **compareTo()**方法。它接收 **Event** 作为参数，然后比较当前 **event** 与作为参数接收的 **event** 的优先级。如果当前 **event** 优先级更大，它将返回**-1**；如果优先级相等，它将返回 **0**；如果当前 **event** 优先级更小，它将返回 **1**。注意：这里的实现与大多数 **Comparator.compare()** 实现相反。

```

@Override
public int compareTo(Event e) {
    if (this.priority>e.getPriority()) {
        return -1;
    } else if (this.priority<e.getPriority()) {
        return 1;
    } else {
        return 0;
    }
}
}

```

8. 创建一个名为 **Task** 的类，实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

9. 声明一个私有 **int** 属性 **id**，用来存放 **task** 的编号。

```
private int id;
```

10. 声明一个私有的 **PriorityBlockingQueue** 属性 **queue**，并指定它的泛型参数是 **Event** 类，来存放 **task** 生成的 **event**。

```
private PriorityBlockingQueue<Event> queue;
```

11. 实现类的构造器来初始化属性。

```
public Task(int id, PriorityBlockingQueue<Event> queue) {
```

```

        this.id=id;
        this.queue=queue;
    }
}

```

12. 实现 **run()**方法。它向队列中添加 1000 个 **event** 对象。使用 **task** 对象的 **id** 属性和自增数字作为传入参数创建每一个 **event** 对象，其中自增数字是用来设定优先级的。调用 **add()**方法将每个 **event** 加入到队列中。

```

@Override
public void run() {
    for (int i=0; i<1000; i++){
        Event event=new Event(id,i);
        queue.add(event);
    }
}

```

13. 创建范例主类 **Main**，并添加 **main()**方法。

```

public class Main{
    public static void main(String[] args) {
}
}

```

14. 创建 **PriorityBlockingQueue** 对象，名为 **queue**，并指定泛型参数是 **Event** 类。

```

PriorityBlockingQueue<Event> queue=new
PriorityBlockingQueue();

```

15. 创建一个含有 5 个 **Thread** 对象的线程数组，用来存放将执行的 5 个任务的线程。

```

Thread taskThreads []=new Thread[5];

```

16. 创建 5 个 **Task** 对象并作为传入参数创建线程，将创建的线程存入到上一步中创建的线程数组中。

```

for (int i=0; i<taskThreads.length; i++){
    Task task=new Task(i,queue);
    taskThreads[i]=new Thread(task);
}

```

17. 启动上一步创建的 5 个线程。

```

for (int i=0; i<taskThreads.length ; i++) {
    taskThreads[i].start();
}

```

18. 使用 **join()** 等待 5 个线程结束。

```
for (int i=0; i<taskThreads.length ; i++) {
    try {
        taskThreads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

19. 在控制台输出队列的实际大小和里面存放的 **events**。使用 **poll()** 方法从队列中取出 **events**。

```
System.out.printf("Main: Queue Size: %d\n",queue.size());
for (int i=0; i<taskThreads.length*1000; i++) {
    Event event=queue.poll();
    System.out.printf("Thread %s: Priority %d\n",event.
        getThread(),event.getPriority());
}
```

20. 在控制台输出一条表示程序运行结束的消息。

```
System.out.printf("Main: Queue Size: %d\n",queue.size());
System.out.printf("Main: End of the program\n");
```

工作原理

本节使用了 **PriorityBlockingQueue** 类实现了一个含有 **Event** 对象的优先级队列。在本节开篇时我们提到过，**PriorityBlockingQueue** 中存放的所有元素都必须实现 **Comparable** 接口，所以 **Event** 类也实现了 **compareTo()** 方法。

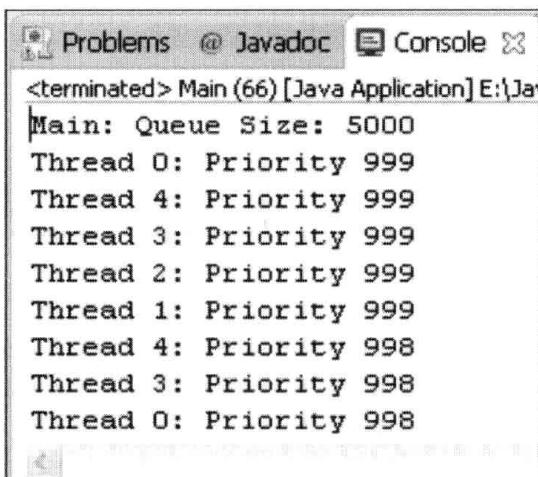
所有 **event** 都有优先级属性。带有最高优先级值的元素是队列中第一个元素。当实现 **compareTo()** 方法时，如果 **event** 本身的优先级值高于作为参数的 **event** 的优先级值，结果返回 **-1**。另一方面，如果 **event** 本身的优先级值低于作为参数的 **event** 的优先级值，结果返回 **1**。如果两个对象的优先级值相等，结果返回 **0**。在返回值为 **0** 的情况下，**PriorityBlockingQueue** 类不保证元素的次序。

Task 类添加 **Event** 对象到优先级队列中。每个 **task** 对象使用 **add()** 方法添加 1,000 个 **event** 到队列中，优先级值在 0 到 999 之间。

Main 类的 **main()** 方法创建了 5 个 **Task** 对象并在对应线程中执行。当所有线程执行完

时，将所有的元素输出到了控制台。为了从队列中取元素，线程使用了 **poll()**方法。它返回队列中的第一个元素并将其移除。

下面的截图是程序执行的部分结果。



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```
<terminated> Main (66) [Java Application] E:\Java\src\main\java\com\example\PriorityBlockingQueueTest.java
Main: Queue Size: 5000
Thread 0: Priority 999
Thread 4: Priority 999
Thread 3: Priority 999
Thread 2: Priority 999
Thread 1: Priority 999
Thread 4: Priority 998
Thread 3: Priority 998
Thread 0: Priority 998
```

我们可以看到，队列有 5,000 个元素，第一个元素有最大的优先级值。

更多信息

PriorityBlockingQueue 类还提供了其他方法。

- ◆ **clear()**: 移除队列中的所有元素。
- ◆ **take()**: 返回队列中的第一个元素并将其移除。如果队列为空，线程阻塞直到队列中有可用的元素。
- ◆ **put(E e)**: E 是 **PriorityBlockingQueue** 的泛型参数，表示传入参数的类型。这个方法把参数对应的元素插入到队列中。
- ◆ **peek()**: 返回队列中的第一个元素，但不将其移除。

参见

- ◆ 参见 6.3 节。

6.5 使用带有延迟元素的线程安全列表

Java API 提供了一种用于并发应用的有趣的数据结构，即 **DelayQueue** 类。这个类可以存放带有激活日期的元素。当调用方法从队列中返回或提取元素时，未来的元素日期将被忽略。这些元素对于这些方法是不可见的。

为了具有调用行为，存放到 **DelayQueue** 类中的元素必须继承 **Delayed** 接口。**Delayed** 接口使对象成为延迟对象，它使存放在 **DelayQueue** 类中的对象具有了激活日期，即到激活日期的时间。该接口强制执行下列两个方法。

- ◆ **compareTo(Delayed o):** **Delayed** 接口继承了 **Comparable** 接口，因此有了这个方法。如果当前对象的延迟值小于参数对象的值，将返回一个小于 0 的值；如果当前对象的延迟值大于参数对象的值，将返回一个大于 0 的值；如果两者的延迟值相等则返回 0。

- ◆ **getDelay(TimeUnit unit):** 这个方法返回到激活日期的剩余时间，单位由单位参数指定。**TimeUnit** 类是一个由下列常量组成的枚举类型：**DAYS**、**HOURS**、**MICROSECONDS**、**MILLISECONDS**、**MINUTES**、**NANOSECONDS** 和 **SECONDS**。

本例中，将会学习如何使用 **DelayQueue** 类来存放具有不同激活日期的 **event**。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Event** 的类并实现 **Delayed** 接口。

```
public class Event implements Delayed {
```

2. 声明私有的名为 **startDate** 的 **Date** 属性。

```
private Date startDate;
```

3. 实现类构造器，初始化属性。

```
public Event (Date startDate) {
    this.startDate=startDate;
}
```

4. 实现 **compareTo()** 方法。它接收一个 **Delayed** 对象为参数，并比较两者延迟值的大小。

```
@Override
public int compareTo(Delayed o) {
    long result=this.getDelay(TimeUnit.NANOSECONDS)-o.
        getDelay(TimeUnit.NANOSECONDS);
    if (result<0) {
        return -1;
    } else if (result>0) {
        return 1;
    }
    return 0;
}
```

5. 实现 **getDelay()** 方法。返回当前对象的 **startDate** 和当前实际日期的间隔，这个间隔的单位是由传入参数 **TimeUnit** 指定的。

```
public long getDelay(TimeUnit unit) {
    Date now=new Date();
    long diff=startDate.getTime()-now.getTime();
    return unit.convert(diff,TimeUnit.MILLISECONDS);
}
```

6. 创建名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

7. 声明私有的 **int** 属性，命名为 **id**，用来存放 **task** 的编号。

```
private int id;
```

8. 声明私有的 **DelayQueue** 属性，命名为 **queue**，并指定泛型参数是 **Event** 类型。

```
private DelayQueue<Event> queue;
```

9. 实现类的构造器来初始化属性。

```
public Task(int id, DelayQueue<Event> queue) {
    this.id=id;
    this.queue=queue;
}
```

10. 实现 **run()** 方法。计算要创建的 **event** 对象的激活日期，用实际日期加上当前 **Task** 编号 **id** 对应的秒数。

```
@Override
public void run() {
    Date now=new Date();
    Date delay=new Date();
    delay.setTime(now.getTime()+(id*1000));
    System.out.printf("Thread %s: %s\n",id,delay);
```

11. 调用 **queue** 对象的 **add()** 方法将 100 个 **event** 对象添加到 **queue** 中。

```
for (int i=0; i<100; i++) {
    Event event=new Event(delay);
    queue.add(event);
}
```

12. 创建范例的主类 **Main**，并添加 **main()** 方法

```
public class Main {
    public static void main(String[] args) throws Exception {
```

13. 创建 **DelayQueue** 对象，指定其泛型参数为 **Event** 类。

```
DelayQueue<Event> queue=new DelayQueue<>();
```

14. 创建一个可以存放 5 个 **Thread** 对象的线程数组。

```
Thread threads[]=new Thread[5];
```

15. 创建 5 个 **Task** 对象，它们有各不相同的编号，并分别作为传入参数创建线程。

```
for (int i=0; i<threads.length; i++) {
    Task task=new Task(i+1, queue);
    threads[i]=new Thread(task);
}
```

16. 执行刚创建的 5 个线程对象。

```
for (int i=0; i<threads.length; i++) {
    threads[i].start();
}
```

17. 使用 **join()** 方法等待所有线程结束。

```
for (int i=0; i<threads.length; i++) {
    threads[i].join();
}
```

18. 把存放在队列中的 **event** 对象输出到控制台。当队列长度大于 0 时，使用 **poll()** 方法获得一个 **Event** 类。如果返回 **null**，则使当前线程休眠 500 毫秒以等待更多 **event** 对象被激活。

```
do {
    int counter=0;
    Event event;
    do {
        event=queue.poll();
        if (event!=null) counter++;
        } while (event!=null);
    System.out.printf("At %s you have read %d events\n",new
        Date(),counter);
    TimeUnit.MILLISECONDS.sleep(500);
} while (queue.size()>0);
}
}
```

工作原理

在本节中我们实现了 **Event** 类。它只有一个属性，即对象的激活日期，因为继承了 **Delayed** 接口，所以 **Event** 对象可以存放到 **DelayQueue** 队列中。

getDelay() 方法用来计算激活日期和实际日期之间的纳秒数。这两个日期都是 **Date** 类的对象，并使用日期对象的 **getTime()** 方法将日期转化为毫秒数后进行比较，然后通过 **getDelay()** 方法的传入参数 **TimeUnit** 的 **convert()** 方法，将时间间隔转化为 **event** 激活时间的剩余纳秒数。**DelayQueue** 类本身是使用纳秒工作的，但是对于使用者来讲，是透明的。

如果当前对象的延迟值小于参数对象的值，**compareTo()** 方法将返回一个小于 0 的值；

如果当前对象的延迟值大于参数对象的值，则返回一个大于 0 的值；如果两者的延迟值相等则返回 0。

Task 类已被实现，这个类有一个名为 **id** 的整型属性。当 **Task** 对象执行时，它添加与 **task id** 相同数量的秒数到实际日期，作为 **DelayQueue** 类中当前 **task** 存放的 **event** 的激活日期。每个 **Task** 对象使用 **add()** 方法在队列中存放 100 个 **event**。

最后，在 **Main** 类的 **main()** 方法中，创建 5 个 **Task** 对象，并在对应线程中执行。当执行完时，使用 **poll()** 方法输出所有 **event** 到控制台。**poll()** 方法提取并移除队列中的第一个元素，如果队列中没有活动的元素，此方法返回 **null** 值。每调用一次 **poll()** 方法，如果返回一个 **Event** 对象，计数器加 1。当调用 **poll()** 返回 **null** 值时，输出计数器中的值到控制台，使线程休眠半秒钟以等待更多活动的 **event**。当队列中存放了 500 个 **event** 时，程序执行完毕。

下面的截图是程序执行的部分结果。

```

Problems @ Javadoc Console Declaration Search
<terminated> Main (67) [Java Application] E:\Java 7 Concurrency Cookbook\desarrollo\jre7\bin\java
Thread 1: Mon Sep 17 16:31:11 CEST 2012
Thread 5: Mon Sep 17 16:31:15 CEST 2012
Thread 3: Mon Sep 17 16:31:13 CEST 2012
Thread 4: Mon Sep 17 16:31:14 CEST 2012
Thread 2: Mon Sep 17 16:31:12 CEST 2012
At Mon Sep 17 16:31:10 CEST 2012 you have read 0 events
At Mon Sep 17 16:31:11 CEST 2012 you have read 0 events
At Mon Sep 17 16:31:11 CEST 2012 you have read 100 events
At Mon Sep 17 16:31:12 CEST 2012 you have read 0 events
At Mon Sep 17 16:31:12 CEST 2012 you have read 100 events

```

能够看到，程序在激活时仅得到 100 个 **event**。

备注：使用 **size()** 方法必须小心。它返回列表中元素的总个数，包括活动和非活动元素。

更多信息

DelayQueue 类还提供了其他一些方法。

- ◆ **clear()**: 移除队列中的所有元素。
- ◆ **offer(E e)**: E 是 **DelayQueue** 的泛型参数，表示传入参数的类型。这个方法把参数对应的元素插入到队列中。
- ◆ **peek()**: 返回队列中的第一个元素，但不将其移除。
- ◆ **take()**: 返回队列中的第一个元素，并将其移除。如果队列为空，线程将被阻塞直

到队列中有可用的元素。

参见

- ◆ 参见 6.3 节。

6.6 使用线程安全可遍历映射

Java API 还提供了一种用于并发应用程序中的有趣数据结构，即 **ConcurrentNavigableMap** 接口及其实现类。实现这个接口的类以如下两部分存放元素：

- ◆ 一个键值（**Key**），它是元素的标识并且是唯一的；
- ◆ 元素其他部分数据。

每一个组成部分都必须在不同的类中实现。

Java API 也提供了一个实现 **ConcurrentSkipListMap** 接口的类，**ConcurrentSkipListMap** 接口实现了与 **ConcurrentNavigableMap** 接口有相同行为的一个非阻塞式列表。从内部实现机制来讲，它使用了一个 **Skip List** 来存放数据。Skip List 是基于并发列表的数据结构，效率与二叉树相近。有了它，就有了一个数据结构，比有序列表在添加、搜索或删除元素时耗费更少的访问时间。

备注：Skip List 由 William Pugh 在 1990 年引入，详见 <http://www.cs.umd.edu/~pugh/>。

当你插入元素到映射中时，**ConcurrentSkipListMap** 接口类使用键值来排序所有元素。除了提供返回一个具体元素的方法之外，这个类也提供获取子映射的方法。

本节将要学习如何使用 **ConcurrentSkipListMap** 类实现对联系人对象的映射。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Contact** 的类。

```
public class Contact {
```

2. 定义两个私有字符串属性 **name** 和 **phone**。

```
private String name;
private String phone;
```

3. 实现类构建器来初始化属性。

```
public Contact(String name, String phone) {
    this.name=name;
    this.phone=phone;
}
```

4. 实现返回 **name** 和 **phone** 属性值的方法。

```
public String getName() {
    return name;
}
public String getPhone() {
    return phone;
}
```

5. 创建名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

6. 声明一个私有的 **ConcurrentSkipListMap** 类型 **map**，并指定其泛型参数是 **String** 和 **Contact** 类。

```
private ConcurrentSkipListMap<String, Contact> map;
```

7. 声明一个私有的名为 **id** 的字符串属性，用来存放当前 **task** 的 **ID**。

```
private String id;
```

8. 实现类构建器来初始化属性。

```
public Task (ConcurrentSkipListMap<String, Contact> map, String id) {
    this.id=id;
    this.map=map;
```

```
}
```

9. 实现 **run()**方法。使用 **task** 的 **ID** 和一个递增的数来创建 1000 个 **Contact** 对象，并调用 **map** 对象的 **put()**方法将这些 **contact** 对象添加到 **map** 中。

```
@Override
public void run() {
    for (int i=0; i<1000; i++) {
        Contact contact=new Contact(id, String.valueOf(i+1000));
        map.put(id+contact.getPhone(), contact);
    }
}
```

10. 实现范例主类 **Main**，并添加 **main()**方法。

```
public class Main {
    public static void main(String[] args) {
```

11. 创建一个 **ConcurrentSkipListMap** 对象，命名为 **map**，并指定泛型参数为 **String** 和 **Contact** 类。

```
ConcurrentSkipListMap<String, Contact> map;
map=new ConcurrentSkipListMap<>();
```

12. 创建一个长度为 25 的线程数组，用来存放 **Task** 对象。

```
Thread threads[]=new Thread[25];
int counter=0;
```

13. 创建 25 个 **task** 对象，每个 **task** 对象的标识符是一个大写字符。分别以这些 **task** 对象创建运行线程并启动。

```
for (char i='A'; i<'Z'; i++) {
    Task task=new Task(map, String.valueOf(i));
    threads[counter]=new Thread(task);
    threads[counter].start();
    counter++;
}
```

14. 使用 **join()**方法等待所有线程执行完成。

```
for (int i=0; i<25; i++) {
```

```

    try {
        threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

15. 使用 **firstEntry()**方法取得 **map** 中的第一个元素。将这个数据输出到控制台。

```

System.out.printf("Main: Size of the map: %d\n",map.size());

Map.Entry<String, Contact> element;
Contact contact;

element=map.firstEntry();
contact=element.getValue();
System.out.printf("Main: First Entry: %s: %s\n",contact.
    getName(),contact.getPhone());

```

16. 使用 **lastEntry()**方法取得 **map** 中的最后一个元素。将这个数据输出到控制台。

```

element=map.lastEntry();
contact=element.getValue();
System.out.printf("Main: Last Entry: %s: %s\n",contact.
    getName(),contact.getPhone());

```

17. 使用 **subMap()**取得 **map** 的一个子映射。将这些数据输出到控制台。

```

System.out.printf("Main: Submap from A1996 to B1002: \n");
ConcurrentNavigableMap<String, Contact> submap=map.
subMap("A1996", "B1002");
do {
    element=submap.pollFirstEntry();
    if (element!=null) {
        contact=element.getValue();
        System.out.printf("%s: %s\n",contact.getName(),contact.
            getPhone());
    }
} while (element!=null);
}

```

工作原理

本节中，我们实现了用 **Task** 类在一个可遍历的映射中存放 **Contact** 对象。每一个

contact 对象都有一个名称，即创建的 **task** 对象的标识 ID，和一个介于 1,000 到 2,000 的电话号码。拼接这两个值作为 **contact** 对象的键。每个 **Task** 对象将生成 1,000 个 **contact** 对象，并使用 **put()** 方法将它们存放在可遍历的映射中。

备注：如果插入的元素的键值已经存在，就用新插入的值覆盖已有的值。

Main 类的 **main()** 方法创建了 25 个 **Task** 对象，使用介于 A 到 Z 之间的字符作为 **ID**，接下来，使用其提供的方法从映射中获取数据。**firstEntry()** 方法返回一个 **Map.Entry** 对象，含有映射中的第一个元素。这个方法不会从映射中移除元素。**Map.Entry** 对象包含键值和元素。使用 **getValue()** 方法就能够获取元素。使用 **getKey()** 就能够获取元素的键值。

lastEntry() 方法返回一个 **Map.Entry** 对象，含有映射中的最后一个元素。**subMap()** 方法返回含有映射中部分元素的 **ConcurrentNavigableMap** 对象。在本例中，元素的键值介于 A1996 到 B1002 之间。可以使用 **pollFirst()** 方法来处理 **subMap()** 方法获取的元素。这个方法会返回并移除子映射中的第一个 **Map.Entry** 对象。

下面的截屏显示了程序执行的结果。

```

Problems @ Javadoc Console Declaration
<terminated> Main (68) [Java Application] E:\Java 7 Concurrency C
Main: Size of the map: 25000
Main: First Entry: A: 1000
Main: Last Entry: Y: 1999
Main: Submap from A1996 to B1002:
A: 1996
A: 1997
A: 1998
A: 1999
B: 1000
B: 1001

```

更多信息

ConcurrentSkipListMap 类还提供了其他的方法。

- ◆ **headMap(K toKey)**: K 是在 **ConcurrentSkipListMap** 对象的泛型参数里用到的键。这个方法返回映射中所有键值小于参数值 **toKey** 的子映射。
- ◆ **tailMap(K fromKey)**: K 是在 **ConcurrentSkipListMap** 对象的泛型参数里用到的键。这个方法返回映射中所有键值大于参数值 **fromKey** 的子映射。
- ◆ **putIfAbsent(K key, V value)**: 如果映射中不存在键 **key**，那么就将 **key** 和 **value** 保存到映射中。

- ◆ **pollLastEntry()**: 返回并移除映射中的最后一个 **Map.Entry** 对象。
- ◆ **replace(K key, V value)**: 如果映射中已经存在键 **key**, 则用参数中的 **value** 替换现有的值。

参见

- ◆ 参见 6.2 节。

6.7 生成并发随机数

Java 并发 API 提供了一个特殊类用以在并发程序中生成伪随机数 (Pseudo-Random Number), 即 Java 7 新引入的 **ThreadLocalRandom** 类。它是线程本地变量。每个生成随机数的线程都有一个不同的生成器, 但是都在同一个类中被管理, 对程序员来讲是透明的。相比于使用共享的 **Random** 对象为所有线程生成随机数, 这种机制具有更好的性能。

在本节, 你将学习如何使用 **ThreadLocalRandom** 类在并发应用中生成随机数。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE (比如 NetBeans), 都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **TaskLocalRandom** 的类并实现 **Runnable** 接口。

```
public class TaskLocalRandom implements Runnable {
```

2. 实现类构造器。使用 **current()**方法为当前线程初始化随机数生成器。

```
public TaskLocalRandom() {
    ThreadLocalRandom.current();
}
```

3. 实现 **run()**方法。获取执行本任务的线程名称, 使用 **nextInt()**方法生成 10 个随机数并打印到控制台。

```

@Override
public void run() {
    String name=Thread.currentThread().getName();
    for (int i=0; i<10; i++) {
        System.out.printf("%s: %d\n", name, ThreadLocalRandom.
            current().nextInt(10));
    }
}

```

4. 创建本范例的主类 **Main**, 并实现 **main()**方法。

```

public class Main {
    public static void main(String[] args) {

```

5. 创建一个长度为 3 的线程数组。

```
    Thread threads[]=new Thread[3];
```

6. 创建 3 个 **TaskLocalRandom** 对象 **task**, 并作为传入参数生成线程。将生成的线程存放到上一步创建的线程数组中。

```

for (int i=0; i<3; i++) {
    TaskLocalRandom task=new TaskLocalRandom();
    threads[i]=new Thread(task);
    threads[i].start();
}

```

工作原理

本例的核心是 **TaskLocalRandom** 类。在类的构造器中, 调用了 **TaskLocalRandom** 类的 **current()** 方法。**current()** 方法是一个静态方法, 返回与当前线程关联的 **TaskLocalRandom** 对象, 所以可以使用这个对象生成随机数。如果调用这个方法的线程还没有关联随机数对象, 就会生成一个新的。在本例中, 使用这个方法初始化与本任务关联的随机数生成器, 所以它将在下一次调用这个方法时被创建。

在 **TaskLocalRandom** 类的 **run()** 方法中, 调用 **current()** 获取与本线程关联的随机数生成器, 同时也调用了 **nextInt()** 方法并以数字 10 作为传入参数。这个方法返回一个介于 0 到 10 之间的伪随机数。每个任务生成 10 个随机数。

更多信息

TaskLocalRandom 类也提供了方法来生成 long、float 和 double 数字和 Boolean 值; 还

可以为方法指定一个数字作为输入参数，来生成介于 0 与该数字之间的随机数；还可以为方法指定两个数字作为输入参数，来生成介于两个参数之间的随机数。

参见

- ◆ 参见 1.10 节。

6.8 使用原子变量

原子变量（**Atomic Variable**）是从 Java 5 开始引入的，它提供了单个变量上的原子操作。在编译程序时，Java 代码中的每个变量、每个操作都将被转换成机器可以理解的指令。例如，当给一个变量赋值时，在 Java 代码中只使用一个指令，但是编译这个程序时，指令被转换成 JVM 语言中的不同指令。当多个线程共享同一个变量时，就会发生数据不一致的错误。

为了避免这类错误，Java 引入了原子变量。当一个线程在对原子变量操作时，如果其他线程也试图对同一原子变量执行操作，原子变量的实现类提供了一套机制来检查操作是否在一步内完成。一般来说，这个操作先获取变量值，然后在本地改变变量的值，然后试图用这个改变的值去替换之前的值。如果之前的值没有被其他线程改变，就可以执行这个替换操作。否则，方法将再执行这个操作。这种操作称为 CAS 原子操作（Compare and Set）。

原子变量不使用锁或其他同步机制来保护对其值的并发访问。所有操作都是基于 CAS 原子操作的。它保证了多线程在同一时间操作一个原子变量而不会产生数据不一致的错误，并且它的性能优于使用同步机制保护的普通变量。

本节将要学习如何使用原子变量实现一个银行帐号和两个不同的任务：一个加钱到帐号上，另一个从帐号上取钱。在例子的实现中使用了 **AtomicLong** 类。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Account** 的类来模拟银行帐户。

```
public class Account {
```

2. 声明一个私有 **AtomicLong** 属性 **balance** 来存放帐户余额。

```
    private AtomicLong balance;
```

3. 实现类构建器来初始化属性。

```
    public Account() {
        balance=new AtomicLong();
    }
```

4. 实现一个名为 **getBalance()** 的方法，返回 **balance** 属性的值。

```
    public long getBalance() {
        return balance.get();
    }
```

5. 实现一个名为 **setBalance()** 的方法来设置 **balance** 属性的值。

```
    public void setBalance(long balance) {
        this.balance.set(balance);
    }
```

6. 实现一个名为 **addAmount()** 的方法来增加 **balance** 属性的值。

```
    public void addAmount(long amount) {
        this.balance.getAndAdd(amount);
    }
```

7. 实现一个名为 **subtractAmount()** 的方法来减少 **balance** 属性的值。

```
    public void subtractAmount(long amount) {
        this.balance.getAndAdd(-amount);
    }
```

8. 创建一个名为 **Company** 的类并实现 **Runnable** 接口。这个类模拟公司的付款。

```
    public class Company implements Runnable {
```

9. 声明一个私有 **Account** 属性 **account**。

```
    private Account account;
```

10. 实现构造器初始化属性。

```
public Company(Account account) {
    this.account=account;
}
```

11. 实现任务的 **run()**方法。使用 **account** 的 **addAmount()**方法增加 10 次 1,000 到帐户余额上。

```
@Override
public void run() {
    for (int i=0; i<10; i++) {
        account.addAmount(1000);
    }
}
```

12. 创建名为 **Bank** 的类并实现 **Runnable** 接口。这个类模拟从帐户中取钱。

```
public class Bank implements Runnable {
```

13. 声明一个私有 **Account** 属性 **account**。

```
private Account account;
```

14. 实现构造器，初始化属性。

```
public Bank(Account account) {
    this.account=account;
}
```

15. 执行任务的 **run()**方法。使用 **account** 的 **subtractAmount()**方法执行 10 次从账户上减少 1000。

```
@Override
public void run() {
    for (int i=0; i<10; i++) {
        account.subtractAmount(1000);
    }
}
```

16. 创建名为 **Main** 的类，并添加 **main()**方法来实现主类。

```
public class Main {
```

```
public static void main(String[] args) {
```

17. 创建一个 **Account** 对象，设置它的账户余额为 1000。

```
Account account=new Account();
account.setBalance(1000);
```

18. 创建一个新的 **Company** 任务和一个执行它的线程。再创建一个新的 **Bank** 任务和一个执行它的线程。

```
Company company=new Company(account);
Thread companyThread=new Thread(company);

Bank bank=new Bank(account);
Thread bankThread=new Thread(bank);
```

19. 在控制台上输出账户的初始化余额。

```
System.out.printf("Account : Initial Balance: %d\n",account.
getBalance());
```

20. 开始执行线程。

```
companyThread.start();
bankThread.start();
```

21. 使用 **join()**方法等待所有线程执行完成，将帐号的最后余额打印到控制台。

```
try {
    companyThread.join();
    bankThread.join();
    System.out.printf("Account : Final Balance: %d\n",account.
        getBalance());
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

工作原理

本例的核心在 **Account** 类中。该类声明了名为 **balance** 的 **AtomicLong** 变量来存放帐户余额，然后实现对余额的存和取操作。为了获得余额，我们实现了 **getBalance()**方法，它使用了 **AtomicLong** 类的 **get()**方法。为了设置余额，我们实现了 **setBalance()**方法，它

使用了 **AtomicLong** 类的 **set()** 方法。为了增加余额，我们实现了 **addAmount()** 方法，它使用了 **AtomicLong** 类的 **getAndAdd()** 方法，这个方法返回增加指定参数值后的余额值。最后，为了减少余额，我们实现了 **subtractAmount()** 方法，它也使用了 **AtomicLong** 类的 **getAndAdd()** 方法。

接下来，实现两个不同的任务类。

- ◆ **Company** 类模拟一家公司，它将增加账户余额。这个类将执行 10 次，每次将余额增加 1,000。

- ◆ **Bank** 类模拟一家银行，它将从账户中取钱。这个类执行 10 次，每次将余额减少 1,000。

在 **Main** 类中，创建 **Account** 对象余额为 1,000。然后执行一个 **bank** 线程和一个 **company** 线程，执行完成后，最终账户余额与初始余额应该相同。

执行完程序，我们会发现最终账户余额与初始值相同。下面的截图显示了本例执行结果的控制台输出。

```

Problems @ Javadoc Console Declaration
<terminated> Main (70) [Java Application] E:\Java 7 Concurrency Co
Account : Initial Balance: 1000
Account : Final Balance: 1000

```

更多信息

在简介中提到过，Java 还提供了其他原子类，**AtomicBoolean**、**AtomicInteger** 和 **AtomicReference** 是原子类的其他实现类。

参见

- ◆ 参见 2.2 节。

6.9 使用原子数组

当实现一个并发应用时，将不可避免地会有多线程共享一个或多个对象的现象，为了避免数据不一致错误，需要使用同步机制（如锁或 **synchronized** 关键字）来保护对这些共

享属性的访问。但是，这些同步机制存在下列问题。

- ◆ 死锁：一个线程被阻塞，并且试图获得的锁正被其他线程使用，但其他线程永远不会释放这个锁。这种情况使得应用不会继续执行，并且永远不会有结束。
- ◆ 即使只有一个线程访问共享对象，它仍然需要执行必须的代码来获取和释放锁。

针对这种情况，为了提供更优的性能，Java 于是引入了比较和交换操作（Compare-and-Swap Operation）。这个操作使用以下三步修改变量的值。

1. 取得变量值，即变量的旧值。
2. 在一个临时变量中修改变量值，即变量的新值。
3. 如果上面获得的变量旧值与当前变量值相等，就用新值替换旧值。如果已有其他线程修改了这个变量的值，上面获得的变量的旧值就可能与当前变量值不同。

采用比较和交换机制不需要使用同步机制，不仅可以避免死锁并且性能更好。

Java 在原子变量（Atomic Variable）中实现了这种机制。这些变量提供了实现比较和交换操作的 **compareAndSet()** 方法，其他方法也基于它展开。

Java 也引入了原子数组（Atomic Array）提供对 **integer** 或 **long** 数字数组的原子操作。本节将学习如何使用 **AtomicIntegerArray** 类的原子数组。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **Incrementer** 的类实现 **Runnable** 接口。

```
public class Incrementer implements Runnable {
```

2. 声明一个私有 **AtomicIntegerArray** 属性 **vector**，用来存放一个整型数字数组。

```
private AtomicIntegerArray vector;
```

3. 实现类构造器初始化属性。

```
public Incrementer(AtomicIntegerArray vector) {
    this.vector=vector;
```

```
}
```

4. 实现 **run()** 方法。使用 **getAndIncrement()** 方法增加数组中的所有元素。

```
@Override
public void run() {
    for (int i=0; i<vector.length(); i++) {
        vector.getAndIncrement(i);
    }
}
```

5. 创建名为 **Decrementer** 的类实现 **Runnable** 接口。

```
public class Decrementer implements Runnable {
```

6. 声明一个私有 **AtomicIntegerArray** 属性，命名为 **vector**，用来存放一个整型数字数组。

```
private AtomicIntegerArray vector;
```

7. 实现类构造器，初始化属性。

```
public Decrementer(AtomicIntegerArray vector) {
    this.vector=vector;
}
```

8. 实现 **run()** 方法。使用 **getAndIncrement()** 方法增加数组中的所有元素。

```
@Override
public void run() {
    for (int i=0; i<vector.length(); i++) {
        vector.getAndDecrement(i);
    }
}
```

9. 创建范例主类 **Main**，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

10. 声明名为 **THREADS** 的常量，赋值 100。创建一个有 1,000 元素的 **AtomicIntegerArray** 对象。

```
final int THREADS=100;
```

```
AtomicIntegerArray vector=new AtomicIntegerArray(1000);
```

11. 创建 **Incrementer** 任务。

```
Incrementer incrementer=new Incrementer(vector);
```

12. 创建 **Decrementer** 任务。

```
Decrementer decrementer=new Decrementer(vector);
```

13. 创建两个线程数组，每个数组的长度是 100，用来存放 100 个 **Thread** 对象。

```
Thread threadIncrementer[]=new Thread[THREADS];
Thread threadDecrementer[]=new Thread[THREADS];
```

14. 创建并执行 100 个线程来执行 **Incrementer** 任务和另外 100 个线程来执行 **Decrementer** 任务。并存放线程到之前创建的数组中。

```
for (int i=0; i<THREADS; i++) {
    threadIncrementer[i]=new Thread(incrementer);
    threadDecrementer[i]=new Thread(decrementer);

    threadIncrementer[i].start();
    threadDecrementer[i].start();
}
```

15. 使用 **join()**方法等待线程执行结束。

```
for (int i=0; i<100; i++) {
    try {
        threadIncrementer[i].join();
        threadDecrementer[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

16. 通过 **get()**方法获取原子数组的各个元素，然后在控制台上输出原子数组中不为 0 的元素。

```
for (int i=0; i<vector.length(); i++) {
    if (vector.get(i)!=0) {
        System.out.println("Vector["+i+"] : "+vector.get(i));
```

```
    }  
}
```

17. 在控制台输出表示程序执行结束的字符串。

```
System.out.println("Main: End of the example");
```

工作原理

本例使用 **AtomicIntegerArray** 对象实现了下面两个不同的任务。

- ◆ **Incrementer** 任务: 这个类使用 **getAndIncrement()**方法增加数组中所有元素的值。
- ◆ **Decrementer** 任务: 这个类使用 **getAndDecrement()**方法减少数组中所有元素的值。

在 **Main** 类中, 创建了 1,000 个元素的 **AtomicIntegerArray** 数组, 执行了 100 个 **Incrementer** 任务和 100 个 **Decrementer** 任务。在任务的结尾, 如果没有不一致的错误, 数组中的所有元素值都必须是 0。执行程序后将会看到, 程序只将最后的消息打印到控制台, 因为所有元素值为 0。

更多信息

现今, Java 还提供了另一个原子数组类, 即 **AtomicLongArray** 类, 它的方法与 **AtomicIntegerArray** 类相同。

这些原子数组还提供了其他方法。

- ◆ **get(int i)**: 返回数组中由参数指定位置的值。
- ◆ **set(int I, int newValue)**: 设置由参数指定位置的新值。

参见

- ◆ 参见 6.8 节。

第 7 章

定制并发类

本章将讲解下列内容：

- ◆ 定制 ThreadPoolExecutor 类
- ◆ 实现基于优先级的 Executor 类
- ◆ 实现 ThreadFactory 接口生成定制线程
- ◆ 在 Executor 对象中使用 ThreadFactory
- ◆ 定制运行在定时线程池中的任务
- ◆ 通过实现 ThreadFactory 接口为 Fork / Join 框架生成定制线程
- ◆ 定制运行在 Fork/Join 框架中的任务
- ◆ 实现定制 Lock 类
- ◆ 实现基于优先级的传输队列
- ◆ 实现自己的原子对象

7.1 简介

Java 并发 API 提供了大量接口和类来实现并发应用程序。这些接口和类既包含了底层机制，如 **Thread** 类、**Runnable** 接口或 **Callable** 接口、**synchronized** 关键字，也包含了高层机制，如在 Java 7 中增加的 **Executor** 框架和 **Fork / Join** 框架。尽管如此，在开发应用程序时，仍会发现已有的 Java 类无法满足需求。

这时，我们就需要基于 Java 提供类和接口来实现自己的定制并发工具。一般来说，我

们可以：

- ◆ 实现一个接口以拥有接口定义的功能，例如，**ThreadFactory** 接口；
- ◆ 覆盖类的一些方法，改变这些方法的行为，来满足需求，例如，覆盖 **Thread** 类的 **run()** 方法，它默认什么都不做，可以被用来覆盖以提供预期的功能。

在本章中，我们将学习如何改变一些 Java 并发 API 的行为，而不需要从头设计一个并发框架。并且在今后的应用程序开发中，可以使用本章中的代码作为定制并发类的起始点。

7.2 定制 ThreadPoolExecutor 类

Executor 框架是一种将线程的创建和执行分离的机制。它基于 **Executor** 和 **ExecutorService** 接口，及这两个接口的实现类 **ThreadPoolExecutor** 展开。**Executor** 有一个内部线程池，并提供了将任务传递到池中线程以获得执行的方法。可传递的任务有如下两种：

- ◆ 通过 **Runnable** 接口实现的任务，它不返回结果；
- ◆ 通过 **Callable** 接口实现的任务，它返回结果。

在这两种情况下，只需要传递任务到执行器，执行器即可使用线程池中的线程或新创建的线程来执行任务。执行器也决定了任务执行的时间。

本节将学习如何覆盖 **ThreadPoolExecutor** 类，通过范例来计算任务在执行器中执行的时间，执行结束后在控制台输出有关执行器的统计信息。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **MyExecutor** 的类，并继承 **ThreadPoolExecutor** 类。

```
public class MyExecutor extends ThreadPoolExecutor {
```

2. 声明一个私有 **ConcurrentHashMap** 属性，其泛型参数为 **String** 和 **Date**，命名为 **startTimes**。

```
private ConcurrentHashMap<String, Date> startTimes;
```

3. 实现类构造器。使用 **super** 关键字调用父类构造器，然后初始化 **startTime** 属性。

```
public MyExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable>
    workQueue) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
        workQueue);
    startTimes=new ConcurrentHashMap<>();
}
```

4. 覆盖 **shutdown()** 方法。将已执行过的任务、正在执行的任务和等待执行的任务的信息输出到控制台。接下来，使用 **super** 关键字调用父类的 **shutdown()** 方法。

```
@Override
public void shutdown() {
    System.out.printf("MyExecutor: Going to shutdown.\n");
    System.out.printf("MyExecutor: Executed tasks:
        %d\n",getCompletedTaskCount());
    System.out.printf("MyExecutor: Running tasks:
        %d\n",getActiveCount());
    System.out.printf("MyExecutor: Pending tasks:
        %d\n",getQueue().size());
    super.shutdown();
}
```

5. 覆盖 **shutdownNow()** 方法。将已执行过的任务、正在执行的任务和等待执行的任务的信息输出到控制台。接下来，使用 **super** 关键字调用父类的 **shutdownNow()** 方法。

```
@Override
public List<Runnable> shutdownNow() {
    System.out.printf("MyExecutor: Going to immediately
        shutdown.\n");
    System.out.printf("MyExecutor: Executed tasks:
        %d\n",getCompletedTaskCount());
    System.out.printf("MyExecutor: Running tasks:
        %d\n",getActiveCount());
    System.out.printf("MyExecutor: Pending tasks:
```

```

        %d\n", getQueue().size());
    return super.shutdownNow();
}

```

6. 覆盖 **beforeExecute()** 方法。输出将要执行的线程的名字、任务的哈希码(Hash Code)、开始日期存放到 **HashMap** 中，它是以任务的哈希码值作为主键的。

```

@Override
protected void beforeExecute(Thread t, Runnable r) {
    System.out.printf("MyExecutor: A task is beginning: %s :
        %s\n", t.getName(), r.hashCode());
    startTimes.put(String.valueOf(r.hashCode()), new Date());
}

```

7. 覆盖 **afterExecute()** 方法。将任务的执行结果输出到控制台，用当前时间减去存放在并发 **HashMap** 中的起始日期来计算任务的运行时间。

```

@Override
protected void afterExecute(Runnable r, Throwable t) {
    Future<?> result=(Future<?>)r;
    try {
        System.out.printf("*****\n");
        System.out.printf("MyExecutor: A task is finishing.\n");
        System.out.printf("MyExecutor: Result: %s\n",result.get());
        Date startDate=startTimes.remove(String.valueOf(r.
            hashCode()));
        Date finishDate=new Date();
        long diff=finishDate.getTime()-startDate.getTime();
        System.out.printf("MyExecutor: Duration: %d\n",diff);
        System.out.printf("*****\n");
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

```

8. 创建名为 **SleepTwoSecondsTask** 的类并实现 **Callable** 接口，这个接口的泛型参数是 **String** 类。实现 **call()** 方法，使当前线程休眠 2s，然后将当前日期转换为字符串，并返回。

```

public class SleepTwoSecondsTask implements Callable<String> {
    public String call() throws Exception {
        TimeUnit.SECONDS.sleep(2);
    }
}

```

```

        return new Date().toString();
    }
}

```

9. 创建名为 **Main** 的主类，并添加 **main()**方法

```

public class Main {
    public static void main(String[] args) {

```

10. 创建一个 **MyExecutor** 对象，命名为 **myExecutor**。

```

MyExecutor myExecutor=new MyExecutor(2, 4, 1000, TimeUnit.
    MILLISECONDS, new LinkedBlockingDeque<Runnable>());

```

11. 创建一个 **Future** 对象列表 **results**，**Future** 对象的泛型参数为 **String** 类，这个列表用来存放即将传递给执行器的任务的返回结果。

```
List<Future<String>> results=new ArrayList<>();i;
```

12. 提交 10 个 **Task** 对象。

```

for (int i=0; i<10; i++) {
    SleepTwoSecondsTask task=new SleepTwoSecondsTask();
    Future<String> result=myExecutor.submit(task);
    results.add(result);
}

```

13. 使用 **get()**方法得到前 5 个任务的执行结果，并将结果输出到控制台。

```

for (int i=0; i<5; i++){
    try {
        String result=results.get(i).get();
        System.out.printf("Main: Result for Task %d :
            %s\n",i,result);
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

```

14. 使用 **shutdown()**方法完成 **executor** 的执行。

```
myExecutor.shutdown();
```

15. 使用 **get()**方法得到后面 5 个任务的执行结果，并将结果输出到控制台。

```
for (int i=5; i<10; i++) {
    try {
        String result=results.get(i).get();
        System.out.printf("Main: Result for Task %d :
            %s\n",i,result);
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

16. 使用 **awaitTermination()**方法等待执行器的完成。

```
try {
    myExecutor.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

17. 将程序执行结束的信息输出到控制台。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

本节实现了一个定制的执行器，它继承了 **ThreadPoolExecutor** 类并覆盖了它的 4 个方法。**beforeExecute()**和**afterExecute()**方法被用来计算任务的运行时间。**beforeExecute()**方法在任务开始前执行。本例中，使用 **HashMap** 存放任务的开始时间。**afterExecute()**方法在任务结束后执行。通过这两个方法分别获得了任务的开始时间和结束时间，它们的时间间隔就是当前任务的执行时间。我们也覆盖了 **shutdown()**和 **shutdownNow()**方法，将执行器执行的任务的统计信息输出到控制台：

- ◆ 通过调用 **getCompletedTaskCount()**方法获得已执行过的任务数；
- ◆ 通过调用 **getActiveCount()**方法获得正在执行的任务数。

对于等待执行的任务，执行器将它们存放在阻塞队列中，调用阻塞队列的 **size()**方法就可以获得等待执行的任务数。**SleepTwoSecondsTask** 类实现了 **Callable** 接口，它让执行线程休眠 2s。**Main** 主类传递了 10 个任务到执行器中，并使用它和其他类来演示它们的特性。

运行程序，我们将看到每个任务的执行时间，同时也会看到通过调用 **shutdown()** 方法而输出统计的信息。

参见

- ◆ 参见 4.2 节。
- ◆ 参见 7.5 节。

7.3 实现基于优先级的 Executor 类

在 Java 并发 API 的第一个版本中，我们必须创建并运行应用程序中的所有线程。在 Java5 中，伴随 Executor 框架的出现，引入了一种新的并发任务机制。

使用 Executor 框架，只需要实现任务并将它们传递到执行器中，然后执行器将负责创建执行任务的线程，并执行这些线程。

执行器内部使用一个阻塞式队列存放等待执行的任务，并按任务到达执行器时的顺序进行存放。另一个可行的替代方案是使用优先级队列存放新的任务，这样，如果有高优先级的新任务到达执行器，它将在其他正在等待的低优先级的线程之前被执行。

本节将学习如何实现一个执行器，范例使用优先级队列存放传递给它的任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **MyPriorityTask** 的类，并实现 **Runnable** 和 **Comparable** 接口，泛型参数为 **MyPriorityTask** 接口。

```
public class MyPriorityTask implements Runnable,  
Comparable<MyPriorityTask> {
```

2. 声明一个名为 **priority** 的私有 **int** 属性。

```
private int priority;
```

3. 声明一个名为 **name** 的私有 **String** 属性。

```
private String name;
```

4. 实现类构造器来初始化属性。

```
public MyPriorityTask(String name, int priority) {
    this.name=name;
    this.priority=priority;
}
```

5. 实现一个方法返回优先级属性的值。

```
public int getPriority() {
    return priority;
}
```

6. 实现在 **Comparable** 接口中声明 **compareTo()** 方法。它接收一个 **MyPriorityTask** 对象作为参数，然后比较当前和参数对象的优先级值。让高优先级的任务先于低优先级的任务执行。

```
@Override
public int compareTo(MyPriorityTask o) {
    if (this.getPriority() < o.getPriority()) {
        return 1;
    }
    if (this.getPriority() > o.getPriority()) {
        return -1;
    }
    return 0;
}
```

7. 实现 **run()** 方法。使当前线程休眠 2s。

```
@Override
public void run() {
    System.out.printf("MyPriorityTask: %s Priority :
        %d\n", name, priority);
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```

        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

8. 创建名为 **Main** 的主类，并添加 **main()** 方法。

```

public class Main {
    public static void main(String[] args) {

```

9. 创建一个 **ThreadPoolExecutor** 对象，命名为 **executor**。传入参数 **PriorityBlockingQueue** 的泛型参数为 **Runnable** 接口，它用来存放等待执行的任务。

```

ThreadPoolExecutor executor=new ThreadPoolExecutor(2,2,1,TimeUnit.SECONDS,new PriorityBlockingQueue<Runnable>());

```

10. 传递 4 个任务到执行器，使用循环计算器作为任务的优先级。使用 **execute()** 方法发送任务到执行器。

```

for (int i=0; i<4; i++) {
    MyPriorityTask task=new MyPriorityTask ("Task "+i,i);
    executor.execute(task);
}

```

11. 使当前线程休眠 1s。

```

try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

12. 再传递 4 个任务到执行器，使用循环计算器作为任务的优先级。使用 **execute()** 方法发送任务到执行器。

```

for (int i=4; i<8; i++) {
    MyPriorityTask task=new MyPriorityTask ("Task "+i,i);
    executor.execute(task);
}

```

13. 使用 **shutdown()**方法关闭执行器。

```
executor.shutdown();
```

14. 使用 **awaitTermination()**方法等待执行器结束。

```
try {
    executor.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

15. 将程序执行结束的信息输出到控制台。

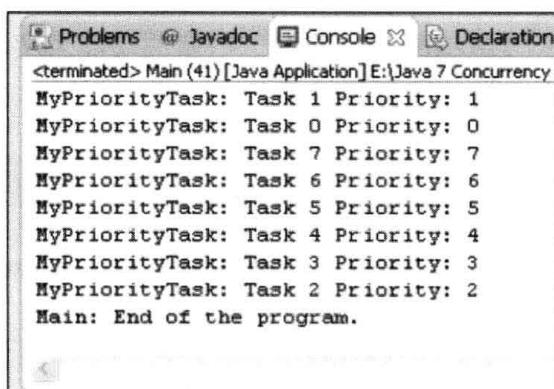
```
System.out.printf("Main: End of the program.\n");
```

工作原理

把一个普通的执行器转换为基于优先级的执行器是非常简单的，只需要把 **PriorityBlockingQueue** 对象作为其中一个传入参数，并且要求它的泛型参数是 **Runnable** 接口即可。使用这种执行器的时候，存放在优先队列中的所有对象必须实现 **Comparable** 接口。

任务类 **MyPriorityTask** 实现了 **Runnable** 接口以成为执行任务，也实现了 **Comparable** 接口以被存放在优先队列中。这个类有一个 **priority** 属性用来存放任务的优先级。一个任务的优先级属性值越高，它越早被执行。**compareTo()**方法决定了优先队列中的任务顺序。**Main** 主类传递了 8 个不同优先级的任务到执行器。发送到的第一个任务是第一个被执行的任务。当执行器空闲并等待任务时，第一批任务到达，它们将立即被执行。这里我们只创建了两个线程执行器，所以前两个任务将被首批执行。接下来，剩余任务基于它们优先级被依次执行。

下面的截图显示了这个例子的执行情况。



更多信息

可以配置 **Executor** 使用 **BlockingQueue** 接口的任意实现，比较有趣的一个是 **DelayQueue**。这个类用来存放带有延迟激活的元素，提供了只返回活动对象的方法。可以使用 **ScheduledThreadPoolExecutor** 类定制自己的类。

参见

- ◆ 参见 4.2 节。
- ◆ 参见 7.2 节。
- ◆ 参见 6.3 节。

7.4 实现 ThreadFactory 接口生成定制线程

工厂模式（**Factory Pattern**）在面向对象编程中是一个应用广泛的设计模式。它是一种创建模式（**Creational Pattern**），目标是创建一个类并通过这个类创建一个或多个类的对象。当创建一个类的对象时，使用工厂类而不是 **new** 操作符。

通过工厂模式，我们能够将对象创建集中化，这样做的好处是：改变对象的创建方式将会变得很容易，并且针对限定资源还可以限制创建对象的数量。例如，通过工厂模式生成了一个类型的 **N** 个对象，就很容易获得创建这些对象的统计数据。

Java 提供了 **ThreadFactory** 接口来实现 **Thread** 对象工厂。Java 并发 API 的一些高级辅助类，像 Executor 框架或 Fork / Join 框架，都使用了线程工厂来创建线程。

线程工厂在 Java 并发 API 中的另一个应用是 **Executors** 类。它提供了大量方法来创建不同类型的 **Executor** 对象。

本节的范例将通过继承 **Thread** 类并增加一些新的功能来实现一个新的线程类，同时实现一个线程工厂来生成这个新的线程类对象。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个类名为 **MyThread** 的类，并继承 **Thread** 类。

```
public class MyThread extends Thread {
```

2. 声明三个私有 **Date** 属性，分别命名为 **creationDate**、**startDate** 和 **finishDate**。

```
private Date creationDate;
private Date startDate;
private Date finishDate;
```

3. 实现类构造器。它接收将要被执行的 **Runnable** 类型对象 **target** 作为参数。然后保存线程的创建时间。

```
public MyThread(Runnable target, String name ) {
    super(target,name);
    setCreationDate();
}
```

4. 实现 **run()** 方法。保存线程的开始时间，然后调用父类的 **run()** 方法，保存执行的结果时间。

```
@Override
public void run() {
    setStartDate();
    super.run();
    setFinishDate();
}
```

5. 实现一个方法来设置 **creationDate** 属性的值。

```
public void setCreationDate() {
    creationDate=new Date();
}
```

6. 实现一个方法来设置 **startDate** 属性的值。

```
public void setStartDate() {
    startDate=new Date();
```

```
}
```

7. 实现一个方法来设置 **finishDate** 属性的值。

```
public void setFinishDate() {
    finishDate=new Date();
}
```

8. 实现一个名为 **getExecutionTime()** 的方法，用来计算线程开始和结束的时间差。

```
public long getExecutionTime() {
    return finishDate.getTime()-startDate.getTime();
}
```

9. 覆盖 **toString()** 方法，返回线程的创建时间和执行时间。

```
@Override
public String toString() {
    StringBuilder buffer=new StringBuilder();
    buffer.append(getName());
    buffer.append(": ");
    buffer.append(" Creation Date: ");
    buffer.append(creationDate);
    buffer.append(" : Running time: ");
    buffer.append(getExecutionTime());
    buffer.append(" Milliseconds.");
    return buffer.toString();
}
```

10. 创建 **MyThreadFactory** 类，实现 **ThreadFactory** 接口。

```
public class MyThreadFactory implements ThreadFactory {
```

11. 声明一个名为 **counter** 的私有 **int** 属性。

```
private int counter;
```

12. 声明一个名为 **prefix** 的私有 **String** 属性。

```
private String prefix;
```

13. 实现类构造器，初始化属性。

```
public MyThreadFactory (String prefix) {
```

```

    this.prefix=prefix;
    counter=1;
}

```

14. 实现 **newThread()** 方法。创建 **MyThread** 对象并增加 **counter** 属性。

```

@Override
public Thread newThread(Runnable r) {
    MyThread myThread=new MyThread(r,prefix+"-"+counter);
    counter++;
    return myThread;
}

```

15. 创建 **MyTask** 类实现 **Runnable** 接口。然后实现 **run()**方法，使当前线程休眠 2s。

```

public class MyTask implements Runnable {
    @Override
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

16. 创建名为 **Main** 的主类，并添加 **main()**方法。

```

public class Main {
    public static void main(String[] args) throws Exception {
}

```

17. 创建 **MyThreadFactory** 对象。

```

MyThreadFactory myFactory=new MyThreadFactory("MyThreadFactory
");

```

18. 创建任务对象。

```

MyTask task=new MyTask();

```

19. 使用工厂的 **newThread()**方法创建 **MyThread** 对象以执行任务。

```
Thread thread=myFactory.newThread(task);
```

20. 启动线程并等待它执行结束。

```
thread.start();
thread.join();
```

21. 使用 **toString()**方法输出关于线程的信息。

```
System.out.printf("Main: Thread information.\n");
System.out.printf("%s\n",thread);
System.out.printf("Main: End of the example.\n");
```

工作原理

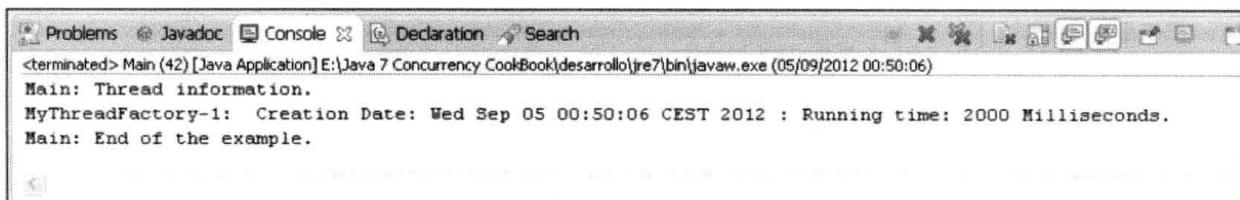
本节实现了一个定制的 **MyThread** 类，它继承了 **Thread** 类。**MyThread** 类有三个属性分别存放它的创建时间，执行起始时间和执行结束时间。**getExecutionTime()**方法使用了执行起始时间和结束时间两个属性，返回线程执行的时间。最后，覆盖了 **toString()**方法来生成有关线程的信息。

在实现了自己的线程类之后，我们通过实现 **ThreadFactory** 接口来完成工厂类，用来创建刚实现的线程类对象。如果使用工厂作为独立对象那么就可以不必实现 **ThreadFactory** 接口，但是如果这个工厂要与 Java 并发 API 的其他类一起使用，就必须实现 **ThreadFactory** 接口。实现 **ThreadFactory** 接口只有一个方法，即 **newThread()**方法，它接收一个 **Runnable** 对象作为参数并返回一个执行 **Runnable** 对象的 **Thread** 对象。在这个例子中，返回一个 **MyThread** 对象。

为了检查这两个类，我们需要实现 **MyTask** 类，用其实现 **Runnable** 接口。这是在 **MyThread** 线程中被执行的任务。**MyTask** 对象能让执行线程休眠 2s。

在 **main()**方法中，通过 **MyThreadFactory** 工厂创建了 **MyThread** 对象来执行任务。运行这个程序，在控制台能看到执行线程的创建时间和执行时间的消息。

下面的截图显示了本例的运行结果。



更多信息

Java 并发 API 提供了 **Executor** 类来生成执行线程，生成的执行线程通常是 **ThreadPoolExecutor** 的对象。也可以使用这个类的 **defaultThreadFactory()** 方法获取 **ThreadFactory** 接口的最基本实现，这个工厂能够生成基本的线程对象，并且生成的线程都属于同一个线程组对象。

当然，你可以在程序中自由使用 **ThreadFactory** 接口，而不必拘泥于 Executor 框架。

7.5 在 Executor 对象中使用 ThreadFactory

在上一节“实现 ThreadFactory 接口生成定制线程”中，我们引入了工厂模式并提供了实现一个线程工厂的实例，该线程工厂实现了 **ThreadFactory** 接口。

Executor 框架是一种分离线程的创建和执行的机制。它是基于 **Executor** 和 **ExecutorService** 接口，以及这两个接口的实现类 **ThreadPoolExecutor** 展开的。它有一个内部线程池，并提供了将任务传递到池中线程以获得执行的方法。可传递的任务有如下两种：

- ◆ 通过 **Runnable** 接口实现的任务，它不返回结果；
- ◆ 通过 **Callable** 接口实现的任务，它返回结果。

Executor 框架内部使用了 **ThreadFactory** 接口来生成新的线程。本节，我们将学习如何实现自己的线程类，以及用来生成这个线程类对象的线程工厂类，同时也将学习如何在执行器中使用工厂类，使执行器将执行创建的线程。

准备工作

阅读上节，实现一个 **ThreadFactory** 接口来生成定制线程。

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 把 7.4 节中的类文件 **MyThread**、**MyThreadFactory**、和 **MyTask** 复制到项目中，

本节范例也将使用它们。

2. 创建名为 **Main** 的主类，并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) throws Exception {
```

3. 创建一个新的 **MyThreadFactory** 对象，命名为 **threadFactory**。

```
MyThreadFactory threadFactory=new MyThreadFactory
("MyThreadFactory");
```

4. 使用 **Executors** 类的 **newCachedThreadPool()**方法创建一个新 **Executor** 对象。传递刚创建的工厂对象作为参数。新的 **Executor** 对象将使用工厂创建必须的线程，它将执行 **MyThread** 线程。

```
ExecutorService executor=Executors.newCachedThreadPool
(threadFactory);
```

5. 创建一个新 **Task** 对象，并使用 **submit()**方法将其传递到执行器。

```
MyTask task=new MyTask();
executor.submit(task);
```

6. 使用 **shutdown()**方法关闭执行器。

```
executor.shutdown();
```

7. 使用 **awaitTermination()**方法等待执行器结束。

```
executor.awaitTermination(1, TimeUnit.DAYS);
```

8. 在控制台输出一条消息表示程序结束。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

在上一节的“范例实现”部分中，已经详细地解释了 **MyThread**、**MyThreadFactory** 和 **MyTask** 类的工作原理。

本例的 **main()**方法使用 **newCachedThreadPool()**方法创建了一个 **Executor** 对象，并

以创建的工厂对象作为传入参数，所以创建的 **Executor** 对象将使用这个工厂创建它需要的线程，并且将执行 **MyThread** 类的线程对象。

运行这个程序，会看到关于线程开始时间和执行时间的信息。下面的截图显示了本例的输出结果。

```
<terminated> Main (43) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo\re7\bin\javaw.exe (05/09/2012 00:56:53)
Main: End of the program.
Thread: MyThreadFactory-1: Creation Date: Wed Sep 05 00:56:54 CEST 2012 : Running time: 2000 Milliseconds.
```

参见

- ◆ 参见第 7 章“定制并发类”之“实现 ThreadFactory 接口生成定制线程”一节。

7.6 定制运行在定时线程池中的任务

定时线程池（**Scheduled Thread Pool**）是 Executor 框架基本线程池的扩展，允许在一段时间后定时执行任务。**ScheduledThreadPoolExecutor** 类不仅实现了这个功能，还允许执行下列两类任务。

- ◆ 延迟任务（Delayed Task）：这类任务在一段时间后仅执行一次。
- ◆ 周期性任务（Periodic Task）：这类任务在一段延迟时间后周期性地执行。

延迟任务能够执行实现 **Callable** 和 **Runnable** 接口的两类对象，周期性任务仅能执行实现 **Runnable** 接口的对象。所有由定时线程池执行的任务都必须实现 **RunnableScheduledFuture** 接口。本节将学习如何实现 **RunnableScheduledFuture** 接口来执行延迟任务和周期性任务。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个类名为 **MyScheduledTask** 的类，泛型参数为 V。它继承了 FutureTask 类，并实现了 RunnableScheduledFuture 接口。

```
public class MyScheduledTask<V> extends FutureTask<V> implements
RunnableScheduledFuture<V> {
```

2. 声明一个名为 **task** 的私有 RunnableScheduledFuture 属性。

```
private RunnableScheduledFuture<V> task;
```

3. 声明一个名为 **executor** 的私有 ScheduledThreadPoolExecutor 属性。

```
private ScheduledThreadPoolExecutor executor;
```

4. 声明一个名为 **period** 的私有 long 属性。

```
private long period;
```

5. 声明一个名为 **startDate** 的私有 long 属性。

```
private long startDate;
```

6. 实现类构造器。它接收将被任务执行的 **Runnable** 对象，并且返回执行的结果。

RunnableScheduledFuture 任务将用来创建 **MyScheduledTask** 任务对象，**ScheduledThreadPoolExecutor** 对象将执行这个任务。这个方法先调用父类的构造器，然后存放 **task** 和 **executor** 属性。

```
public MyScheduledTask(Runnable runnable, V result,
RunnableScheduledFuture<V> task, ScheduledThreadPoolExecutor
executor) {
super(runnable, result);
this.task=task;
this.executor=executor;
}
```

7. 实现 **getDelay()** 方法。如果是周期性任务且 **startDate** 属性值不等于 0，则计算 **startDate** 属性和当前时间的时间差作为返回值。否则返回存放在 **task** 属性中的延迟值。需要注意的是，返回结果需要被转化为参数指定的时间单位。

```
@Override
public long getDelay(TimeUnit unit) {
```

```

    if (!isPeriodic()) {
        return task.getDelay(unit);
    } else {
        if (startDate==0){
            return task.getDelay(unit);
        } else {
            Date now=new Date();
            long delay=startDate-now.getTime();
            return unit.convert(delay, TimeUnit.MILLISECONDS);
        }
    }
}

```

8. 实现 **compareTo()**方法。调用原来任务的 **compareTo()**方法。

```

@Override
public int compareTo(Delayed o) {
    return task.compareTo(o);
}

```

9. 实现 **isPeriodic()**方法。调用原来任务的 **isPeriodic()**方法。

```

@Override
public boolean isPeriodic() {
    return task.isPeriodic();
}

```

10. 实现 **run()**方法。如果是周期性任务，则需要用任务下一次执行的开始时间更新它的 **startDate** 属性，即用当前时间加上周期间隔作为下一次执行的开始时间。然后，再次增加任务到 **ScheduledThreadPoolExecutor** 对象的队列中。

```

@Override
public void run() {
    if (isPeriodic() && (!executor.isShutdown())) {
        Date now=new Date();
        startDate=now.getTime()+period;
        executor.getQueue().add(this);
    }
}

```

11. 在控制台输出当前时间，然后调用 **runAndReset()**方法执行任务，并且将任务执行后的当前时间输出到控制台。

```

        System.out.printf("Pre-MyScheduledTask: %s\n", new Date());
        System.out.printf("MyScheduledTask: Is Periodic:
                           %s\n", isPeriodic());
        super.runAndReset();
        System.out.printf("Post-MyScheduledTask: %s\n", new Date());
    }
}

```

12. 实现 **setPeriod()**方法来设置任务的周期。

```

public void setPeriod(long period) {
    this.period=period;
}

```

13. 创建一个名为 **MyScheduledThreadPoolExecutor** 的类，它继承了一个 **ScheduledThreadPoolExecutor** 类，用来执行 **MyScheduledTask** 任务。

```

public class MyScheduledThreadPoolExecutor extends
    ScheduledThreadPoolExecutor {

```

14. 实现类的构造器，仅调用父类的构造器。

```

public MyScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize);
}

```

15. 实现 **decorateTask()**方法。它接收将要被执行的 **Runnable** 对象作为参数，另一个参数是 **RunnableScheduledFuture** 任务，用来执行这个 **Runnable** 对象。这个方法使用这些对象构造并返回 **MyScheduledTask** 任务。

```

@Override
protected <V> RunnableScheduledFuture<V> decorateTask(Runnable
    runnable,
    RunnableScheduledFuture<V> task) {
    MyScheduledTask<V> myTask=new MyScheduledTask<V>(runnable,
        null, task, this);
    return myTask;
}

```

16. 覆盖 **scheduledAtFixedRate()**方法。调用父类的这个方法，把返回对象强制类型转换为一个 **MyScheduledTask** 对象，并使用 **setPeriod()**方法设置任务的周期。

```

@Override
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
    long initialDelay, long period, TimeUnit unit) {
ScheduledFuture<?> task= super.scheduleAtFixedRate(command,
    initialDelay, period, unit);
MyScheduledTask<?> myTask=(MyScheduledTask<?>)task;
myTask.setPeriod(TimeUnit.MILLISECONDS.convert(period,unit));
return task;
}

```

17. 创建名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

18. 实现 **run()**方法。在任务开始时输出相应消息，然后使当前线程休眠 2 秒，在任务结束时输出结束消息。

```

@Override
public void run() {
    System.out.printf("Task: Begin.\n");
    try {
        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Task: End.\n");
}

```

19. 创建名为 **Main** 的主类，并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) throws Exception{
```

20. 创建 **MyScheduledThreadPoolExecutor** 对象，命名为 **executor**。创建时使用了 2 作为参数，这样线程池中将有两个线程。

```
MyScheduledThreadPoolExecutor executor=new
MyScheduledThreadPoolExecutor(2);
```

21. 创建一个 **Task** 类型的对象并命名为 **task**。在控制台输出当前时间。

```
Task task=new Task();
```

```
System.out.printf("Main: %s\n", new Date());
```

22. 使用 **schedule()**方法发送一个延迟任务到执行器。这个任务将延迟 1 秒后被执行。

```
executor.schedule(task, 1, TimeUnit.SECONDS);
```

23. 使主线程休眠 3 秒。

```
TimeUnit.SECONDS.sleep(3);
```

24. 创建另一个 **Task** 对象。在控制台再次输出当前时间。

```
task=new Task();
System.out.printf("Main: %s\n", new Date());
```

25. 使用 **scheduleAtFixedRate()**方法发送一个周期性任务到执行器。这个任务将延迟 1 秒后被执行，然后每 3 秒执行一次。

```
executor.scheduleAtFixedRate(task, 1, 3, TimeUnit.SECONDS);
```

26. 使主线程休眠 10 秒。

```
TimeUnit.SECONDS.sleep(10);
```

27. 使用 **shutdown()**方法关闭执行器。使用 **awaitTermination()**方法等待执行器结束。

```
executor.shutdown();
executor.awaitTermination(1, TimeUnit.DAYS);
```

28. 在控制台输出一条消息表示程序结束。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

在本节中，我们实现了定制的任务类 **MyScheduledTask**，它能在 **ScheduledThreadPoolExecutor** 对象里执行。该类继承了 **FutureTask** 类并实现了 **RunnableScheduledFuture** 接口。之所以实现 **RunnableScheduledFuture** 接口是因为所有在定时执行器中执行的任务都必须实现这个接口；之所以继承 **FutureTask** 类是因为该类提供了 **RunnableScheduledFuture** 接口中声明的方法的有效实现。所有上面提到的接口和类都使用了泛型参数化，泛型参数是任务返回的数据类型。

为了在定时执行器中使用 **MyScheduledTask** 任务，我们必须在 **MyScheduledThreadPoolExecutor** 类中覆盖 **decorateTask()** 方法。**MyScheduledThreadPoolExecutor** 类继承了 **ScheduledThreadPoolExecutor**，**decorateTask()** 方法默认返回的是 **ScheduledThreadPoolExecutor** 实现的缺省定时任务，覆盖这个方法会将缺省定时任务替换为 **MyScheduledTask** 任务。所以，当实现自定义的定时任务时，必须实现自定义的定时执行器。

decorateTask() 方法只是使用传入参数来创建 **MyScheduledTask** 对象；传入的 **Runnable** 对象将在任务中执行，执行结果也将被任务返回。在本例中，任务不会返回一个结果，所以结果参数使用了 **null** 作为返回值。这个方法默认返回的任务将执行 **Runnable** 对象。覆盖后新返回的对象将在池中替换默认的任务对象，并传入参数执行器用来执行这个任务。在本例中，使用关键字 **this** 来引用创建任务的执行器。

MyScheduledTask 类既可以执行延迟任务也可以执行周期性任务。我们已经实现了 **getDelay()** 和 **run()** 方法，它们具有执行这两种任务的所有必需逻辑。

对于 **getDelay()** 方法，定时执行器调用它来确定是否必须执行一个任务。这个方法在延迟任务和周期性任务中表现不一样。像之前提到的，**MyScheduledTask** 类构造器接收的是 **ScheduledRunnableFuture** 默认实现的对象，用来执行传入的 **Runnable** 对象，并把它保存为 **MyScheduledTask** 类的属性，以供 **MyScheduledTask** 类的其他方法和数据进行访问。如果执行的是一个延迟任务，**getDelay()** 方法返回传入任务的延迟值，如果执行的是周期性任务，**getDelay()** 方法返回 **startDate** 属性与当前时间的时间差。

run() 方法，即执行任务的方法。周期性任务的一个特殊性是，它必须添加到执行器的队列中作为新任务，才能被再次执行。所以，如果正在执行一个周期性任务，需用当前时间与任务执行周期的和重置 **startDate** 属性值，并将这个任务再次加入到执行器队列中。**startDate** 属性存放任务下一次执行的时间。接下来，使用 **FutureTask** 类提供的 **runAndReset()** 方法来执行方法。在延迟任务中，不需要再次把任务放入执行器队列，因为它仅执行一次。

备注：必须考虑执行器已关闭时的情况。在这种情况下，周期性任务不需要再加入到执行器队列中。

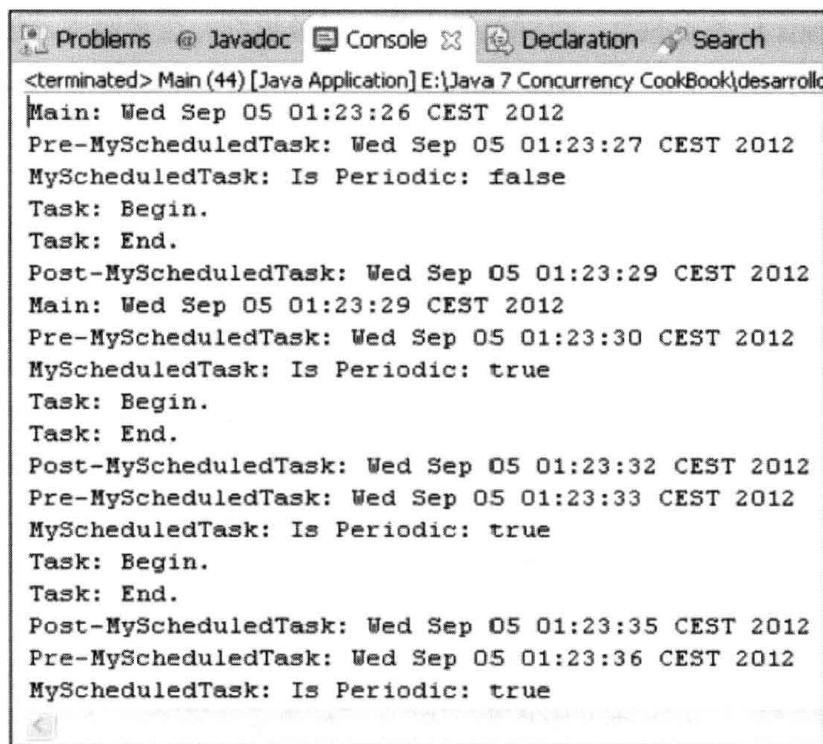
最后，在 **MyScheduledThreadPoolExecutor** 类中，我们覆盖了 **scheduleAtFixedRate()** 方法。如前所述，对于周期性任务，**startDate** 属性必须被重置，而这需要用到任务的周期，此时我们并没有初始化它。所以我们必须覆盖 **scheduleAtFixedRate()** 方法，因为它接收任务的周期值，并传入到 **MyScheduledTask** 类里。

Task 类使得范例更完整，它实现了 **Runnable** 接口，也是在定时执行器中执行的任务。

Main 主类创建了一个 **MyScheduledThreadPoolExecutor** 执行器对象并发送给它下面的两个任务；

- ◆ 一个延迟任务，在当前时间 1 秒后执行；
- ◆ 一个周期性任务，第一次在当前时间 1 秒后执行，接下来每 3 秒执行一次。

下面的截图显示了本例的部分执行结果。我们将发现两类任务已经被正确地执行了。



```

Problems @ Javadoc Console Declaration Search
<terminated> Main (44) [Java Application] E:\Java 7 Concurrency CookBook\desarrollo
Main: Wed Sep 05 01:23:26 CEST 2012
Pre-MyScheduledTask: Wed Sep 05 01:23:27 CEST 2012
MyScheduledTask: Is Periodic: false
Task: Begin.
Task: End.
Post-MyScheduledTask: Wed Sep 05 01:23:29 CEST 2012
Main: Wed Sep 05 01:23:29 CEST 2012
Pre-MyScheduledTask: Wed Sep 05 01:23:30 CEST 2012
MyScheduledTask: Is Periodic: true
Task: Begin.
Task: End.
Post-MyScheduledTask: Wed Sep 05 01:23:32 CEST 2012
Pre-MyScheduledTask: Wed Sep 05 01:23:33 CEST 2012
MyScheduledTask: Is Periodic: true
Task: Begin.
Task: End.
Post-MyScheduledTask: Wed Sep 05 01:23:35 CEST 2012
Pre-MyScheduledTask: Wed Sep 05 01:23:36 CEST 2012
MyScheduledTask: Is Periodic: true

```

更多信息

ScheduledThreadPoolExecutor 类提供了另外一种 **decorateTask()** 方法，它接收一个 **Callable** 对象（而不是 **Runnable** 对象）作为参数。

参见

- ◆ 参见 4.7 节。
- ◆ 参见 4.8 节。

7.7 通过实现 ThreadFactory 接口为 Fork/Join 框架生成定制线程

Java7 中最有趣的特性之一是 Fork/Join 框架。它是 **Executor** 和 **ExecutorService** 接口的实现，这两个接口能够允许我们执行 **Callable** 和 **Runnable** 任务，而不需要去关注执行这些任务的具体线程。

这个执行器用于执行可以分拆成更小任务体的任务，它的主要组件如下。

- ◆ 一种特殊类型任务，由 **ForkJoinTask** 类来实现。
- ◆ 两种操作，其中通过 **fork** 操作将一个任务分拆为多个子任务，而通过 **join** 操作等待这些子任务结束。
- ◆ 工作窃取算法（Work-Stealing Algorithm），用来对线程池的使用进行优化。当一个任务等待它的子任务时，执行这个任务的线程可以被用来执行其他任务。

Fork / Join 框架的主类是 **ForkJoinPool** 类。从内部实现来说，它有下面两个元素：

- ◆ 一个任务队列，存放的是等待被执行的任务；
- ◆ 一个执行这些任务的线程池。

在本节，我们将学习如何实现一个定制的工作线程（Worker Thread），它被 **ForkJoinPool** 类使用，此外我们还将学习如何通过工厂模式来使用它。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **MyWorkerThread** 的类，继承 **ForkJoinWorkerThread** 类。

```
public class MyWorkerThread extends ForkJoinWorkerThread {
```

2. 声明并创建一个私有 **ThreadLocal** 属性，命名为 **taskCounter**，并指定泛型参数是 **Integer**。

```
private static ThreadLocal<Integer> taskCounter=new
    ThreadLocal<Integer>();
```

3. 实现类构造器。

```
protected MyWorkerThread(ForkJoinPool pool) {
    super(pool);
}
```

4. 覆盖 **onStart()** 方法。调用它的父类方法，并输出一条消息到控制台，然后设置本线程的 **taskCounter** 属性值为 0。

```
@Override
protected void onStart() {
    super.onStart();
    System.out.printf("MyWorkerThread %d: Initializing task
        counter.\n",getId());
    taskCounter.set(0);
}
```

5. 覆盖 **onTermination()** 方法。输出本线程的 **taskCounter** 属性值到控制台。

```
@Override
protected void onTermination(Throwable exception) {
    System.out.printf("MyWorkerThread %d:
        %d\n",getId(),taskCounter.get());
    super.onTermination(exception);
}
```

6. 实现 **addTask()** 方法。增加 **taskCounter** 属性的值。

```
public void addTask(){
    int counter=taskCounter.get().intValue();
    counter++;
    taskCounter.set(counter);
}
```

7. 创建名为 **MyWorkerThreadFactory** 的类，它实现了 **ForkJoinWorkerThreadFactory**

接口。实现 **newThread()** 方法，用来创建并返回一个 **MyWorkerThread** 对象。

```
public class MyWorkerThreadFactory implements
ForkJoinWorkerThreadFactory {
@Override
public ForkJoinWorkerThread newThread(ForkJoinPool pool) {
    return new MyWorkerThread(pool);
}
}
```

8. 创建一个名为 **MyRecursiveTask** 的类，它继承泛型参数 Integer 的 **RecursiveTask** 类。

```
public class MyRecursiveTask extends RecursiveTask<Integer> {
```

9. 声明一个名为 **array** 的私有 **int** 数组。

```
private int array[];
```

10. 声明两个私有 **int** 属性，名字分别为 **start** 和 **end**。

```
private int start, end;
```

11. 实现类构造器来初始化属性。

```
public Task (int array[],int start, int end) {
    this.array=array;
    this.start=start;
    this.end=end;
}
```

12. 实现 **compute()** 方法，用于对数组中从起始位到指定位的所有元素求和。首先，它将正在执行任务的线程转换为 **MyWorkerThread** 类型，然后使用 **addTask()** 方法增加线程属性 **taskCounter** 的值。

```
@Override
protected Integer compute() {
    Integer ret;
    MyWorkerThread thread=(MyWorkerThread) Thread.currentThread();
    thread.addTask();
}
```

13. 实现 **addResults()** 方法。它计算并返回通过参数传入的两个任务的结果之和。

```

private Integer addResults(Task task1, Task task2) {
    int value;
    try {
        value = task1.get().intValue() + task2.get().intValue();
    } catch (InterruptedException e) {
        e.printStackTrace();
        value=0;
    } catch (ExecutionException e) {
        e.printStackTrace();
        value=0;
    }
}

```

14. 使线程休眠 10 毫秒，并返回任务的结果。

```

try {
    TimeUnit.MILLISECONDS.sleep(10);
} catch (InterruptedException e) {
    e.printStackTrace();
}
return value;
}

```

15. 创建名为 **Main** 的主类，并实现 **main()** 方法。

```

public class Main {
    public static void main(String[] args) throws Exception {
}

```

16. 创建一个名为 **factory** 的 **MyWorkerThreadFactory** 对象。

```
MyWorkerThreadFactory factory=new MyWorkerThreadFactory();
```

17. 创建一个名为 **pool** 的 **ForkJoinPool** 对象。把之前创建的工厂对象传递到构造器。

```
ForkJoinPool pool=new ForkJoinPool(4, factory, null, false);
```

18. 创建一个有 100,000 个整数的数组 **array**。初始化所有元素值为 1。

```

int array[] = new int[100000];

for (int i=0; i<array.length; i++) {
    array[i]=1;
}

```

19. 创建一个 **Task** 对象，用来对这个数组的所有元素求和。

```
MyRecursiveTask task=new MyRecursiveTask(array,0,array.length);
```

20. 使用 **execute()**方法将求和任务发送到 **pool**。

```
pool.execute(task);
```

21. 使用 **join()**方法等待任务结束。

```
task.join();
```

22. 使用 **shutdown()**方法关闭 **pool** 对象。

```
pool.shutdown();
```

23. 使用 **awaitTermination()**方法等待执行器的结束。

```
pool.awaitTermination(1, TimeUnit.DAYS);
```

24. 使用 **get()**方法在控制台输出任务的结果。

```
System.out.printf("Main: Result: %d\n",task.get());
```

25. 在控制台输出一条消息表示程序的结束。

```
System.out.printf("Main: End of the program\n");
```

工作原理

在 Fork/Join 框架里使用的线程被称为工作线程（Worker Thread）。Java 提供了 **ForkJoinWorkerThread** 类，它继承了 **Thread** 类并实现了可在 Fork/Join 框架里使用的工作线程。

本节范例实现了 **MyWorkerThread** 类，它继承了 **ForkJoinWorkerThread** 类，并覆盖了类的两个方法。这里我们在每个工作线程中实现一个计数器，以统计一个工作线程已执行了多少任务，然后我们使用了 **ThreadLocal** 类型的计数器。这样每个线程都有自己的计数器，这对于程序员来讲是透明的。

覆盖 **ForkJoinWorkerThread** 类的 **onStart()**方法，以初始化任务计数器。当工作线程开始执行这个方法时会被自动调用。也覆盖了 **onTermination()**方法，用来输出任务计数器

中的值到控制台。当工作线程完成执行这个方法时也会被自动调用。我们也实现了一个 **addTask()** 方法，用来增加每个线程的任务计数器。

与 Java 并发 API 中的所有执行器一样，**ForkJoinPool** 类也使用工厂模式来创建它的线程，所以如果想在 **ForkJoinPool** 类中使用 **MyWorkerThread** 线程，必须实现自己的线程工厂。在 Fork / Join 框架里，这个工厂必须实现 **ForkJoinPool.ForkJoinWorkerThreadFactory** 接口。针对这个目的，我们实现了 **MyWorkerThreadFactory** 类。它只有一个 **newThread()** 方法，用来创建一个新的 **MyWorkerThread** 对象。

最后，我们需要使用刚创建的工厂来初始化 **ForkJoinPool** 类，这是在 **Main** 主类中使用 **ForkJoinPool** 类的构造器来实现的。

下面的截图显示了这个程序的部分输出结果。

```

Problems @ Javadoc Console Declaration Search
<terminated> Main (45) [Java Application] E:\Java 7 Concurrency CookBook
MyWorkerThread 10: Initializing task counter.
MyWorkerThread 11: Initializing task counter.
MyWorkerThread 8: 513
MyWorkerThread 10: 511
MyWorkerThread 11: 511
MyWorkerThread 9: 512
Main: Result: 100000
Main: End of the program

```

可以看到，**ForkJoinPool** 对象执行了 4 个工作线程，以及每个线程执行的任务数。

更多信息

当一个线程正常结束或抛出 **Exception** 异常时，**ForkJoinWorkerThread** 类提供的 **onTermination()** 方法都会被自动调用。这个方法接收一个 **Throwable** 对象作为参数。如果参数为 **null** 值，工作线程正常结束；如果参数有一个值，线程将抛出异常。所以我们必须编写相应的代码来处理这种异常情况。

参见

- ◆ 参见 5.2 节。

- ◆ 参见 1.13 节。

7.8 定制运行在 Fork / Join 框架中的任务

Executor 框架分离了任务的创建和执行。在这个框架下，只需要实现 **Runnable** 对象和 **Executor** 对象，并将 **Runnable** 对象发送给执行器即可，这样，执行器将创建执行这些任务的线程，并对其进行管理直到线程终止。

Java7 在 Fork / Join 框架中提供了一个特殊形式的执行器。这个框架旨在通过分拆和合并技术，把任务分拆为更小的任务。在一个任务中，我们需要检查待解决问题的规模。如果问题的规模比制定的规模大，则需要把问题分拆为两个或多个任务，并使用 Fork / Join 框架执行这些任务。如果问题的规模小于已制定的规模，直接在任务里解决问题即可。另外，可以选择是否返回结果。Fork / Join 框架通过工作窃取算法（Work-Stealing Algorithm），提升了这类问题的整体性能。

Fork / Join 框架的主类是 **ForkJoinPool** 类。从内部来讲，它有下面两个元素：

- ◆ 一个任务队列，存放的是等待被执行的任务；
- ◆ 一个执行这些任务的线程池。

默认情况下，**ForkJoinPool** 类执行的任务是 **ForkJoinTask** 类的对象。我们也可以传递 **Runnable** 和 **Callable** 对象到 **ForkJoinPool** 类中，但是它们不会利用 **Fork / Join** 框架的优势。一般来说，我们将 **ForkJoinTask** 的两种子类传递给 **ForkJoinPool** 对象。

- ◆ **RecursiveAction**: 用于任务不返回结果的情况。
- ◆ **RecursiveTask**: 用于任务返回结果的情况。

在本节，我们将学习如何通过继承 **ForkJoinTask** 类来实现 Fork / Join 框架下的定制任务。这个任务实现计算自身的执行时间并在控制台输出，所以我们能够控制它的进一步演变。也可以实现自己的 Fork / Join 任务来输出日志信息，并获得任务中使用到的资源，或对任务结果做进一步的处理。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **MyWorkerTask** 的类，并继承 **ForkJoinTask** 类，且泛型参数为 Void 类型。

```
public abstract class MyWorkerTask extends ForkJoinTask<Void> {
```

2. 声明一个名为 **name** 的私有 **String** 属性，存放任务的名字。

```
private String name;
```

3. 实现类构造器来初始化属性。

```
public MyWorkerTask(String name) {
    this.name=name;
}
```

4. 实现 **getRawResult()** 方法。这是一个 **ForkJoinTask** 类的抽象方法，当 **MyWorkerTask** 任务不返回任何结果时，这个方法必须返回 **null** 值。

```
@Override
public Void getRawResult() {
    return null;
}
```

5. 实现 **setRawResult()** 方法。这是 **ForkJoinTask** 类的另一个抽象方法，用于当 **MyWorkerTask** 任务不返回任何结果时，设置方法体为空。

```
@Override
protected void setRawResult(Void value) {

}
```

6. 实现 **exec()** 方法。这是类的主方法。本例将任务的逻辑委托到 **compute()** 方法。计算这个方法的执行时间并输出到控制台。

```
@Override
protected boolean exec() {
    Date startDate=new Date();
```

```

        compute();
        Date finishDate=new Date();
        long diff=finishDate.getTime()-startDate.getTime();
        System.out.printf("MyWorkerTask: %s : %d Milliseconds to
            complete.\n",name,diff);
        return true;
    }
}

```

7. 实现 **getName()**方法来返回任务名。

```

public String getName() {
    return name;
}

```

8. 声明抽象方法 **compute()**。如之前提到的，这个方法实现的是任务的逻辑，**MyWorkerTask** 类的子类必须实现这个方法。

```
protected abstract void compute();
```

9. 创建一个名为 **Task** 的类，继承 **MyWorkerTask** 类。

```
public class Task extends MyWorkerTask {
```

10. 声明一个名为 **array** 的私有 int 数组。

```
private int array[];
```

11. 实现类构造器来初始化属性。

```

public Task(String name, int array[], int start, int end) {
    super(name);
    this.array=array;
    this.start=start;
    this.end=end;
}

```

12. 实现 **compute()**方法。该方法增加由开始和结束属性决定的数组的元素块。如果元素块中有超过 100 个元素，则分拆这个块为两部分，并创建两个 **Task** 对象来处理各部分。使用 **invokeAll()**方法传递这些任务到 **pool**。

```

protected void compute() {
    if (end-start>100){

```

```
int mid=(end+start)/2;
Task task1=new Task(this.getName()+"1",array,start,mid);
Task task2=new Task(this.getName()+"2",array,mid,end);
invokeAll(task1,task2);
```

13. 如果元素块中有少于 100 个元素，则使用一个 for 循环对所有的元素加 1。

```
    } else {  
        for (int i=start; i<end; i++) {  
            array[i]++;  
        }  
    }
```

14. 使当前线程休眠 50 毫秒。

```
    try {
        Thread.sleep(50);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

15. 创建名为 **Main** 的主类，并实现 **main()**方法。

```
public class Main {  
    public static void main(String[] args) throws Exception {
```

16. 创建一个有 10,000 元素的 int 数组。。

```
int array[] = new int[10000];
```

17. 创建一个名为 pool 的 ForkJoinPool 对象。

```
ForkJoinPool pool=new ForkJoinPool();
```

18. 创建一个 **Task** 对象对数组所有元素值加 1。构造器参数 **Task** 作为任务名, **array** 对象, 0 和 10000 来表明这个任务要处理整个数组。

```
Task task=new Task("Task",array,0,array.length);
```

19. 使用 **execute()**方法传递任务到 **pool**。

```
pool.invoke(task);
```

20. 使用 **shutdown()**方法关闭 **pool**。

```
pool.shutdown();
```

21. 在控制台输出一条消息表明程序结束。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

在本节，我们继承 **ForkJoinTask** 类而实现了 **MyWorkerTask** 类。这是我们的任务基类，它能在 **ForkJoinPool** 中执行，能使用执行器提供的所有优点（如工作窃取算法）。这个类与 **RecursiveAction** 和 **RecursiveTask** 类相当。

当继承 **ForkJoinTask** 类时，必须实现下列三个方法。

- ◆ **setRawResult()**: 被用来设置任务的结果。当任务不返回任何结果时，方法为空。
- ◆ **getRawResult()**: 被用来获取任务的结果。当任务不返回任何结果时，方法必须返回 **null** 值。
- ◆ **exec()**: 实现任务的逻辑。本例将逻辑委托到了抽象方法 **compute()** 中（跟 **RecursiveAction** 和 **RecursiveTask** 类一样），在 **exec()**方法中测量这个方法执行的时间，并输出到控制台。

最后，在本例的 **main** 主类中，创建了有 10,000 个元素的数组、一个 **ForkJoinPool** 执行器和一个 **Task** 对象，用来处理整个数组。运行程序，会发现不同的任务分别在控制台输出了各自的执行时间。

参见

- ◆ 参见 5.2 节。
- ◆ 参见 7.7 节。

7.9 实现定制 Lock 类

锁是 Java 并发 API 提供的最基本的同步机制之一。程序员用锁来保护代码的临界区（Critical Section），所以同一时间只有一个线程能执行临界区代码。它提供了下列两种操作：

- ◆ **lock()**: 当要访问临界区代码时调用这个操作。如果另一个线程正在运行临界区代码，其他线程将被阻塞直到被访问临界区的锁唤醒。
- ◆ **unlock()**: 在临界区代码结尾调用这个操作，以允许其他线程来访问这部分临界区代码。

在 Java 并发 API 中，锁是使用 **Lock** 接口来声明的，并且有一些类实现了 **Lock** 接口，如 **ReentrantLock** 类。

本节将学习如何实现自定义 **Lock** 对象，通过实现 **Lock** 接口，来保护临界区代码。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建名为 **MyAbstractQueuedSynchronizer** 的类，继承自 **AbstractQueuedSynchronizer** 类。

```
public class MyAbstractQueuedSynchronizer extends
AbstractQueuedSynchronizer {
```

2. 声明一个名为 **state** 的私有 **AtomicInteger** 属性。

```
private AtomicInteger state;
```

3. 实现类构造器来初始化属性。

```
public MyAbstractQueuedSynchronizer() {
    state=new AtomicInteger(0);
```

```
}
```

4. 实现 **tryAcquire()**方法。它用来把 **state** 变量的值从 0 改为 1。如果成功，返回 **true**，否则返回 **false**。

```
@Override
protected boolean tryAcquire(int arg) {
    return state.compareAndSet(0, 1);
}
```

5. 实现 **tryRelease()**方法。它用来把 **state** 变量的值从 1 改为 0。如果成功，返回 **true**，否则返回 **false**。

```
@Override
protected boolean tryRelease(int arg) {
    return state.compareAndSet(1, 0);
}
```

6. 创建一个名为 **MyLock** 的类，并实现 **Lock** 接口。

```
public class MyLock implements Lock{
```

7. 声明一个名为 **sync** 的私有 **AbstractQueuedSynchronizer** 属性。

```
private AbstractQueuedSynchronizer sync;
```

8. 实现类构造器，并创建一个新的 **MyAbstractQueueSynchronizer** 对象来初始化 **sync** 属性。

```
public MyLock() {
    sync=new MyAbstractQueueSynchronizer();
```

9. 实现 **lock()**方法。然后在方法中调用 **sync** 对象的 **acquire()**方法。

```
@Override
public void lock() {
    sync.acquire(1);
}
```

10. 实现 **lockInterruptibly()**方法。然后在方法中调用 **sync** 对象的 **acquireInterruptibly()**

方法。

```
@Override
public void lockInterruptibly() throws InterruptedException {
    sync.acquireInterruptibly(1);
}
```

11. 实现 **tryLock()** 方法。然后在方法中调用 **sync** 对象的 **tryAcquireNanos()** 方法。

```
@Override
public boolean tryLock() {
    try {
        return sync.tryAcquireNanos(1, 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
        return false;
    }
}
```

12. 实现另一种 **tryLock()** 方法。它带有两个参数：一个名为 **time** 的 **long** 型参数和一个名为 **unit** 的 **TimeUnit** 类型参数。然后在方法中调用 **sync** 对象的 **tryAcquireNanos()** 方法。

```
@Override
public boolean tryLock(long time, TimeUnit unit)
    throws InterruptedException {
    return sync.tryAcquireNanos(1, TimeUnit.NANOSECONDS.
        convert(time, unit));
}
```

13. 实现 **unlock()** 方法。然后在方法中调用 **sync** 对象的 **release()** 方法。

```
@Override
public void unlock() {
    sync.release(1);
}
```

14. 实现 **newCondition()** 方法。创建 **sync** 对象内部类 **ConditionObject** 的一个新对象。

```
@Override
public Condition newCondition() {
    return sync.new ConditionObject();
}
```

15. 创建一个名为 **Task** 的类，实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

16. 声明一个名为 **lock** 的私有 **MyLock** 属性。

```
private MyLock lock;
```

17. 声明一个名为 **name** 的私有 String 属性。

```
private String name;
```

18. 实现类构造器并初始化其属性。

```
public Task(String name, MyLock lock) {
    this.lock=lock;
    this.name=name;
}
```

19. 实现 **run()** 方法。获取锁，然后让线程休眠 2 秒，释放 **lock** 对象。

```
@Override
public void run() {
    lock.lock();
    System.out.printf("Task: %s: Take the lock\n",name);
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.printf("Task: %s: Free the lock\n",name);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

20. 创建名为 **Main** 的主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) {
```

21. 创建一个名为 **lock** 的 **MyLock** 对象。

```
MyLock lock=new MyLock();
```

22. 创建并执行 10 个 **Task** 任务。

```
for (int i=0; i<10; i++) {
    Task task=new Task("Task-"+i,lock);
    Thread thread=new Thread(task);
    thread.start();
}
```

23. 使用 **tryLock()**方法获取锁。等待 1 秒如果取不到锁，就在控制台输出一条消息后重新去尝试获取锁。

```
boolean value;
do {
    try {
        value=lock.tryLock(1, TimeUnit.SECONDS);
        if (!value) {
            System.out.printf("Main: Trying to get the Lock\n");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
        value=false;
    }
} while (!value);
```

24. 在控制台输出一条消息表示获得了锁，然后释放线程。

```
System.out.printf("Main: Got the lock\n");
lock.unlock();
```

25. 在控制台输出一条消息表示程序线束。

```
System.out.printf("Main: End of the program\n");
```

工作原理

Java 并发 API 提供了 **AbstractQueuedSynchronizer** 类，它用来实现带有锁或信号特性的同步机制。顾名思义，**AbstractQueuedSynchronizer** 是一个抽象类。它提供操作来对临界区代码的访问进行控制，并对等待访问临界区代码的阻塞线程队列进行管理。它有下面两个方法。

- ◆ **tryAcquire()**: 当访问临界区代码时调用这个方法。如果访问成功，返回 **true** 值；否则返回 **false** 值。
- ◆ **tryRelease()**: 当释放对临界区代码的访问时调用这个方法。如果释放成功，返回 **true** 值；否则返回 **false** 值。

在这两个方法中，我们需要实现对临界区代码的并发访问控制。在例子中，我们实现了继承自 **AbstractQueuedSynchronizer** 类的 **MyQueuedSynchronizer** 类，并覆盖了这两个抽象方法，在覆盖的时候使用了 **AtomicInteger** 变量对临界区代码的访问进行控制。锁可以被获取的时候，变量值为 0，这时允许一个线程访问临界区代码；锁在不可用的时候，变量值为 1，这时不允许任何线程访问临界区代码。

我们使用 **AtomicInteger** 类的 **compareAndSet()** 方法试图把第一个参数的值设置为第二个参数指定值。在实现 **tryAcquire()** 方法的时候，又使用 **compareAndSet()** 方法试图把原子变量从 0 设置为 1。同样，在实现 **tryRelease()** 方法的时候，我们使用 **compareAndSet()** 方法试图把原子变量从 1 设置为 0。

必须实现 **AbstractQueuedSynchronizer** 抽象类，因为这个类的其他实现（如在 **ReentrantLock** 中使用的）被实现为私有内部类，所以无法访问到这些内部类。

接下来，我们实现了 **MyLock** 类，它实现了 **Lock** 接口，并且有一个 **MyQueuedSynchronizer** 属性。在实现 **Lock** 接口的所有方法中，我们具体使用了 **MyQueuedSynchronizer** 对象的方法。

最后，我们应用 **Task** 类，实现了 **Runnable** 接口，并使用一个 **MyLock** 对象访问临界区。临界区将正在访问它的休眠线程 2 秒钟。**main** 主类创建了一个 **MyLock** 对象，并运行了 10 个 **Task** 对象，这 10 个 **Task** 对象共享 **MyLock** 对象。**main** 主类也使用了 **tryLock()** 方法来尝试获得这个锁。

运行范例，我们可以看到只有一个线程能够访问临界区代码，当这个线程结束时，另一个线程才能访问临界区代码。

通过定制锁，能够获得它的使用情况、控制临界区的锁定时间，还可以实现高级同步机制。例如，对仅在特定时间内才可用的资源进行访问控制。

更多信息

AbstractQueuedSynchronizer 抽象类提供了两个方法用来管理锁状态。**getState()** 和 **setState()**。这两个方法接收并返回锁状态的整型值。你可能已经使用这些方法而不是

AtomicInteger 属性来存储锁状态。

Java 并发 API 提供了另一个类来实现同步机制，即 **AbstractQueuedLongSynchronizer** 抽象类，它与 **AbstractQueuedSynchronizer** 抽象类是一样的，只是使用了一个 **long** 属性来存储线程的状态而已。

参见

- ◆ 参见 2.5 节。

7.10 实现基于优先级的传输队列

Java 7 API 提供了几种用于并发应用程序的数据结构。我们要重点关注以下两种结构。

◆ **LinkedTransferQueue**: 适用于拥有生产者—消费者结构的程序中。在这些应用程序中，有一个或多个数据生产者和数据消费者，然后这些生产者 / 消费者却共享着一个数据结构。生产者将数据存放到数据结构中，消费者则从数据结构中取出数据。如果数据结构为空，消费者被阻塞直到数据结构中有可用的数据。如果数据结构已满，则生产者被阻塞直到数据结构有可用的空间可以存放生产者将要存放进来的数据。

◆ **PriorityBlockingQueue**: 在这个数据结构中，元素按顺序存储。这些元素必须实现 **Comparable** 接口，并实现接口中定义的 **compareTo** 方法。当插入一个元素时，它会与已有元素进行比较直至找到它的位置。

LinkedTransferQueue 中的元素按到达的先后顺序进行存储，所以早到的被优先消费。你可能面临这样的一个场景：你想开发一个生产者 / 消费者程序，数据是按某种优先级（而不是按到达的时间顺序）被消费。本节将学习如何实现一个数据结构，范例用来解决数据结构中的元素是按优先级排序的生产者 / 消费者问题，优先级高的先被处理。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **MyPriorityTransferQueue** 的类，继承自 **PriorityBlockingQueue** 类并实现 **TransferQueue** 接口。

```
public class MyPriorityTransferQueue<E> extends
    PriorityBlockingQueue<E> implements
    TransferQueue<E> {
```

2. 声明一个名为 **counter** 的私有 **AtomicInteger** 属性，用来存储等待消费元素的消费者数量。

```
private AtomicInteger counter;
```

3. 声明一个名为 **transferred** 的私有 **LinkedBlockingQueue** 属性。

```
private LinkedBlockingQueue<E> transferred;
```

4. 声明一个名为 **lock** 的私有 **ReentrantLock** 属性。

```
private ReentrantLock lock;
```

5. 实现类构造器来初始化它的属性。

```
public MyPriorityTransferQueue() {
    counter=new AtomicInteger(0);
    lock=new ReentrantLock();
    transferred=new LinkedBlockingQueue<E>();
}
```

6. 实现 **tryTransfer()** 方法。这个方法尝试立即将元素发送到一个正在等待的消费者。如果没有等待中的消费者，该方法返回 **false**。

```
@Override
public boolean tryTransfer(E e) {
    lock.lock();
    boolean value;
    if (counter.get()==0) {
        value=false;
    } else {
```

```

        put(e);
        value=true;
    }
    lock.unlock();
    return value;
}

```

7. 实现 **transfer()**方法。这个方法尝试立即将元素发送到一个正在等待的消费者。如果没有等待中的消费者，该方法将元素存储到 **transferred** 队列，并等待出现试图获得元素的第一个消费者。在这之前，线程将被阻塞。

```

@Override
public void transfer(E e) throws InterruptedException {
    lock.lock();
    if (counter.get()!=0) {
        put(e);
        lock.unlock();
    } else {
        transferred.add(e);
        lock.unlock();
        synchronized (e) {
            e.wait();
        }
    }
}

```

8. 实现 **tryTransfer()**方法，它接收 3 个参数：第一个参数用以表示生产和消费的元素，第二个参数表示如果没有消费者则等待一个消费者的时间，第三个参数表示等待时间的单位。如果有消费者在等待，它就立即发送元素。否则，将参数指定的时间转换为毫秒并使用 **wait()**方法让线程休眠。当消费者取元素时，如果线程仍在 **wait()**方法中休眠，将使用 **notify()**方法去唤醒它，这个实现在下面的部分讲解。

```

@Override
public boolean tryTransfer(E e, long timeout, TimeUnit unit)
    throws InterruptedException {
    lock.lock();
    if (counter.get()!=0) {
        put(e);
        lock.unlock();
        return true;
    } else {

```

```

        transferred.add(e);
        long newTimeout= TimeUnit.MILLISECONDS.convert(timeout,
            unit);
        lock.unlock();
        e.wait(newTimeout);
        lock.lock();
        if (transferred.contains(e)) {
            transferred.remove(e);
            lock.unlock();
            return false;
        } else {
            lock.unlock();
            return true;
        }
    }
}

```

9. 实现 **hasWaitingConsumer()**方法。使用 **counter** 属性的值来计算该方法的返回值，如果 **counter** 大于 0，返回 **true**，否则返回 **false**。

```

@Override
public boolean hasWaitingConsumer() {
    return (counter.get() != 0);
}

```

10. 实现 **getWaitingConsumerCount()**方法。返回 **counter** 属性的值。

```

@Override
public int getWaitingConsumerCount() {
    return counter.get();
}

```

11. 实现 **take()**方法。该方法将被准备消费元素的消费者调用。首先，获得之前定义的锁，然后增加正在等待的消费者的数量。

```

@Override
public E take() throws InterruptedException {
    lock.lock();
    counter.incrementAndGet();
}

```

12. 如果在 **transferred** 队列中没有元素，则释放锁并尝试使用 **take()**方法从队列中取得一个元素并再次获取锁。如果队列中没有元素，该方法将让线程休眠直至有元素可被

消费。

```
E value=transferred.poll();
if (value==null) {
    lock.unlock();
    value=super.take();
    lock.lock();
```

13. 否则，从 **transferred** 队列中取出 **value** 元素，并唤醒可能在等待元素被消费的线程。

```
} else {
    synchronized (value) {
        value.notify();
    }
}
```

14. 减少正在等待的消费者的数量并释放锁。

```
counter.decrementAndGet();
lock.unlock();
return value;
}
```

15. 实现一个名为 **Event** 的类，并实现 **Comparable** 接口，其泛型参数为 **Event** 类。

```
public class Event implements Comparable<Event> {
```

16. 声明一个名为 **thread** 的私有 **String** 属性来存储创建事件的线程的名字。

```
private String thread;
```

17. 声明一个名为 **priority** 的私有 **int** 属性来存储事件的优先级。

```
private int priority;
```

18. 实现类构造器来初始化它的属性。

```
public Event(String thread, int priority){
    this.thread=thread;
    this.priority=priority;
}
```

19. 实现 **getThread()** 方法来返回线程 **thread** 属性的值。

```
public String getThread() {
    return thread;
}
```

20. 实现 **getPriority()** 方法来返回优先级 **priority** 属性的值。

```
public int getPriority() {
    return priority;
}
```

21. 实现 **compareTo()** 方法。该方法用来比较当前事件和传入事件。如果当前事件比传入事件的优先级值大，则返回-1；如果当前事件比传入事件的优先级值小，则返回 1；如果两个事件有相同的优先级值，则返回 0。如此便得到按优先级倒序排列的一个列表，有更高优先级的事件将被存储在队列的头部。

```
public int compareTo(Event e) {
    if (this.priority > e.getPriority()) {
        return -1;
    } else if (this.priority < e.getPriority()) {
        return 1;
    } else {
        return 0;
    }
}
```

22. 实现一个名为 **Producer** 的类，并实现 **Runnable** 接口。

```
public class Producer implements Runnable {
```

23. 声明一个名为 **buffer** 的私有 **MyPriorityTransferQueue** 属性，它的泛型参数是 **Event** 类，用来存储由这个类生成的事件。

```
private MyPriorityTransferQueue<Event> buffer;
```

24. 实现类构造器来初始化它的属性。

```
public Producer(MyPriorityTransferQueue<Event> buffer) {
```

```

        this.buffer=buffer;
    }
}

```

25. 实现类的 **run()**方法。创建 100 个 **Event** 对象，使用创建序号作为优先级（事件创建的越晚优先级越高），并使用 **put()**方法将它们插入到队列中。

```

@Override
public void run() {
    for (int i=0; i<100; i++) {
        Event event=new Event(Thread.currentThread().getName(),i);
        buffer.put(event);
    }
}

```

26. 实现一个名为 **Consumer** 的类，并实现 **Runnable** 接口。

```
public class Consumer implements Runnable {
```

27. 声明一个名为 **buffer** 的私有 **MyPriorityTransferQueue** 属性，它的泛型参数是 **Event** 类，用来获得由这个类消费的事件。

```
private MyPriorityTransferQueue<Event> buffer;
```

28. 实现类构造器来初始化它的属性。

```
public Consumer(MyPriorityTransferQueue<Event> buffer) {
    this.buffer=buffer;
}
```

29. 实现 **run()**方法。使用 **take()**方法消费 1002 个 **Event**（例子中所有产生的事件），并在控制台输出产生事件的线程的名称以及事件的优先级 **priority**。

```

@Override
public void run() {
    for (int i=0; i<1002; i++) {
        try {
            Event value=buffer.take();
            System.out.printf("Consumer: %s: %d\n",value.
                getThread(),value.getPriority());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}
}
```

30. 创建名为 **Main** 的主类，并实现 **main()**方法。

```
public class Main {
    public static void main(String[] args) throws Exception {
```

31. 创建一个名为 **buffer** 的 **MyPriorityTransferQueue** 对象。

```
MyPriorityTransferQueue<Event>buffer=new MyPriorityTransferQueue<Event>();
}
```

32. 创建一个 **Producer** 任务，并运行 10 个线程来执行这些任务。

```
Producer producer=new Producer(buffer);

Thread producerThreads[]=new Thread[10];
for (int i=0; i<producerThreads.length; i++) {
    producerThreads[i]=new Thread(producer);
    producerThreads[i].start();
}
```

33. 创建并运行一个 **Consumer** 任务。

```
Consumer consumer=new Consumer(buffer);
Thread consumerThread=new Thread(consumer);
consumerThread.start();
```

34. 在控制台输出实际的消费者数。

```
System.out.printf("Main: Buffer: Consumer count: %d\n",buffer.
    .getWaitingConsumerCount());
```

35. 使用 **transfer()**方法把一个事件传递给消费者。

```
Event myEvent=new Event("Core Event",0);
buffer.transfer(myEvent);
System.out.printf("Main: My Event has been transferred.\n");
```

36. 使用 **join()**方法等待生产者执行结束。

```

for (int i=0; i<producerThreads.length; i++) {
    try {
        producerThreads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

37. 线程休眠 1 秒钟。

```
TimeUnit.SECONDS.sleep(1);
```

38. 在控制台输出实际的消费者数。

```
System.out.printf("Main: Buffer: Consumer count: %d\n", buffer.
    getWaitingConsumerCount());
```

39. 使用 **transfer()** 方法传递另一个事件。

```
myEvent=new Event("Core Event 2",0);
buffer.transfer(myEvent);
```

40. 使用 **join()** 方法等待消费者线程执行结束。

```
consumerThread.join();
```

41. 在控制台输出一条消息表示程序结束。

```
System.out.printf("Main: End of the program\n");
```

工作原理

本节实现了 **MyPriorityTransferQueue** 数据结构。该结构用在生产者—消费者问题中，但是里面的元素按优先级进行排序而不是按到达的先后顺序排序。因为 Java 不允许多继承，所以第一考虑是使用基类。**MyPriorityTransferQueue** 类继承了 **PriorityBlockingQueue** 类，并实现了把元素按优先级插入到结构中的操作。**MyPriorityTransferQueue** 类也实现了 **TransferQueue** 接口，用来增加与生产者—消费者相关的操作方法。

MyPriorityTransferQueue 类有下面 3 个属性。

- ◆ 一个名为 **counter** 的 **AtomicInteger** 属性：用来存储等待从数据结构中读取元素的

消费者的数量。当消费者调用 **take()** 方法从数据结构中取元素时，**counter** 属性增加 1。当消费者完成 **take()** 方法的执行，**counter** 属性减 1。**counter** 用在 **hasWaitingConsumer()** 和 **getWaitingConsumerCount()** 方法的实现中。

- ◆ 一个名为 **lock** 的 **ReentrantLock** 属性：用来控制访问上述实现的生产者—消费者的操作，实现仅有一个线程可操作数据结构的目标。

- ◆ 一个 **LinkedBlockingQueue** 列表：存储已传递（尚未被消费）的元素。

我们在 **MyPriorityTransferQueue** 类中实现了几个方法，这些方法都在 **TransferQueue** 接口中进行声明的，而 **take()** 方法在所继承的 **PriorityBlockingQueue** 类中则有默认实现。这些方法中的两个已在之前部分描述过，下面是其他方法的介绍。

- ◆ **tryTransfer(E e)**：这个方法尝试直接发送一个元素到消费者。如果有消费者正在等待，这个方法将把元素存入到可被立即消费的优先级队列中，然后返回 **true**。如果没有消费者在等待，则返回 **false**。

- ◆ **transfer(E e)**：这个方法直接传递一个元素到一个消费者。如果有消费者正在等待，这个方法将把元素存入到可被立即消费的优先级队列中。否则，元素被存储在已转移（尚未被消费）的元素列表中，线程将被阻塞直至元素被消费。当线程休眠时，必须释放锁，否则队列被阻塞。

- ◆ **tryTransfer(E e, long timeout, TimeUnit unit)**：这个方法与 **transfer()** 方法相似，但是线程阻塞的时间由参数决定。当线程休眠时，必须释放锁，否则队列被阻塞。

- ◆ **take()**：这个方法返回下一个要被消费的元素。如果在转移列表 **transferred** 中有元素，那么就从这个列表中获取将要被消费的元素。否则，就从优先队列中获取。

在这个数据结构之后，我们实现了 **Event** 类，它是存储在数据结构中的元素类。**Event** 类有两个属性存储生产者 ID（线程的名称）和事件的优先级 **priority**，它实现了 **Comparable** 接口，这是数据结构的一个需求。

接下来，实现了生产者 **Producer** 和消费者 **Consumer** 类。在本例中，有 10 个生产者和消费者，它们共享相同的缓冲区。每个生产者生成 100 个带有递增优先级的事件，所以有更高优先级的事件最后被生成。

本例的 **main** 主类创建了 **MyPriorityTransferQueue** 对象、10 个生产者和 1 个消费者，并且使用 **MyPriorityTransferQueue** 类对象 **buffer** 的 **transfer()** 方法传递两个事件到缓冲区。

下面的截图显示了这个程序的执行结果。

```

Problems @ Javadoc Console E:\Java 7
<terminated> Main (48) [Java Application] E:\Java 7
Main: Buffer: Consumer count: 0
Consumer: Thread-1: 99
Consumer: Core Event: 0
Consumer: Thread-2: 99
Consumer: Thread-5: 99
Consumer: Thread-6: 99
Consumer: Thread-7: 99
Consumer: Thread-0: 99
Consumer: Thread-3: 99
Consumer: Thread-9: 99
Consumer: Thread-8: 99
Consumer: Thread-4: 99
Consumer: Thread-5: 98
Consumer: Thread-6: 98
Consumer: Thread-1: 98
Consumer: Thread-2: 98

```

我们可以看到，有更高优先级的事件先被消费，当 **transferred** 转移列表中有元素的时候，消费者会先消费高优先级的事件。

参见

- ◆ 参见 6.4 节。
- ◆ 参见 6.3 节。

7.11 实现自己的原子对象

从 Java 5 开始就已经引入了原子变量（Atomic Variable），它提供对单个变量的原子操作。当线程使用原子变量执行操作时，类的实现包括了一个机制来检查操作是否在单步内结束。简单来讲，就是操作获取变量值，然后通过本地变量来改变值，接着尝试改旧值为新值。如果旧值未变，则执行改变。否则，方法重新执行。

本节将学习如何继承一个原子对象和如何实现两个遵守原子对象机制保证所有操作在单步内结束的方法。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **ParkingCounter** 的类，继承 **AtomicInteger** 类。

```
public class ParkingCounter extends AtomicInteger {
```

2. 声明一个名为 **maxNumber** 的私有 **int** 属性，用来存储停车场中可停放汽车的最大数量。

```
private int maxNumber;
```

3. 实现类构造器来初始化它的属性。

```
public ParkingCounter(int maxNumber) {
    set(0);
    this.maxNumber=maxNumber;
}
```

4. 实现 **carIn()**方法。如果它的值小于指定的最大值，这个方法就增加车的数量。构建一个无限循环并使用 **get()**方法取得内部计数器的值。

```
public boolean carIn() {
    for (;;) {
        int value=get();
```

5. 如果内部计数器的值等于可停放汽车的最大数量 **maxNumber** 属性，计数器不可以被继续增加（即停车场已满，车无法再进入），然后返回 **false**。

```
if (value==maxNumber) {
    System.out.printf("ParkingCounter: The parking lot is
                      full.\n");
    return false;
```

6. 否则，将内部计数器的值加 1 作为一个新值，并使用 **compareAndSet()**方法将内部计数器设置为这个新值。如果 `changed` 结果返回 `false`，那么内部计数器不被增加，所以会再次循环。如果 `changed` 结果返回 `true`，意味着计数器已被更新，然后直接返回 `true`。

```
    } else {
        int newValue=value+1;
        boolean changed=compareAndSet(value,newValue);
        if (changed) {
            System.out.printf("ParkingCounter: A car has
                entered.\n");
            return true;
        }
    }
}
```

7. 实现 **carOut()** 方法。如果车的数量大于零，这个方法将减少车的数量。构建一个无限循环并使用 **get()** 方法取得内部计数器的值。

```
public boolean carOut() {  
    for (;;) {  
        int value=get();  
        if (value==0) {  
            System.out.printf("ParkingCounter: The parking lot is  
                empty.\n");  
            return false;  
        } else {  
            int newValue=value-1;  
            boolean changed=compareAndSet(value,newValue);  
            if (changed) {  
                System.out.printf("ParkingCounter: A car has gone  
                    out.\n");  
                return true;  
            }  
        }  
    }  
}
```

8. 创建一个名为 **Sensor1** 的类，实现 **Runnable** 接口。

```
public class Sensor1 implements Runnable {
```

9. 声明一个名为 **counter** 的私有 **ParkingCounter** 属性。

```
private ParkingCounter counter;
```

10. 实现类构造器来初始化它的属性。

```
public Sensor1(ParkingCounter counter) {  
    this.counter=counter;  
}
```

11. 实现 **run()** 方法。调用几次 **carIn()** 和 **carOut()** 方法。

```
@Override  
public void run() {  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
    counter.carOut();  
    counter.carOut();  
    counter.carOut();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
}
```

12. 创建一个名为 **Sensor2** 的类，实现 **Runnable** 接口。

```
public class Sensor2 implements Runnable {
```

13. 声明一个名为 **counter** 的私有 **ParkingCounter** 属性。

```
private ParkingCounter counter;
```

14. 实现类构造器来初始化它的属性。

```
public Sensor2(ParkingCounter counter) {  
    this.counter=counter;  
}
```

15. 实现 **run()** 方法。调用几次 **carIn()** 和 **carOut()** 方法。

```
@Override  
public void run() {  
    counter.carIn();  
    counter.carOut();  
    counter.carOut();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
    counter.carIn();  
}
```

16. 创建名为 **Main** 的主类并添加 **main()** 方法

```
public class Main {  
    public static void main(String[] args) throws Exception {
```

17. 创建一个名为 **counter** 的 **ParkingCounter** 对象。

```
ParkingCounter counter=new ParkingCounter(5);
```

18. 创建并执行一个 **Sensor1** 任务和一个 **Sensor2** 任务。

```
Sensor1 sensor1=new Sensor1(counter);  
Sensor2 sensor2=new Sensor2(counter);  
  
Thread thread1=new Thread(sensor1);  
Thread thread2=new Thread(sensor2);  
  
thread1.start();  
thread2.start();
```

19. 等待两个任务执行结束。

```
thread1.join();  
thread2.join();
```

20. 在控制台输出实际的 **counter** 值。

```
System.out.printf("Main: Number of cars: %d\n", counter.get());
```

21. 在控制台输出一个消息表示程序结束。

```
System.out.printf("Main: End of the program.\n");
```

工作原理

ParkingCounter 类继承自 **AtomicInteger** 类，它带有两个原子操作 **carIn()** 和 **carOut()**。本例模拟一个系统来控制停车场内的汽车数量。停车场能接纳的汽车数由 **maxNumber** 属性表示。

carIn() 方法将停车场现有汽车数与最大停车数相比较。如果相等，则汽车不可以进入停车场，并返回 **false**。否则，使用下面的原子操作指令。

1. 将原子对象的值赋给一个本地变量。
2. 将本地变量值增加 1 作为新值，并把这个新值赋给另一个不同的变量。
3. 使用 **compareAndSet()** 方法尝试使用新值替换旧值。如果返回 **true**，作为参数的旧值就是当前内部计数器的值，所以计数器值将发生改变。这个操作是以原子方式执行的，将返回 **true**，如 **carIn()** 方法。如果 **compareAndSet()** 返回 **false**，作为参数的旧值已不是当前内部计数器的值（另外一个线程已修改过它），所以这个操作就不是以原子方式执行的。操作将重新开始直到它能够以原子方式完成。

carOut() 方法与 **carIn()** 相似。我们还实现了两个 **Runnable** 对象，这两个对象使用 **carIn()** 和 **carOut()** 方法模拟停车场的活动。当运行这个程序时，能够看到停车场永不会超出设定的最大停车数。

参见

- ◆ 参见 6.8 节。

第 8 章

测试并发应用程序

本章将讲解下列内容：

- ◆ 监控 **Lock** 接口
- ◆ 监控 **Phaser** 类
- ◆ 监控执行器框架
- ◆ 监控 **Fork / Join** 池
- ◆ 输出高效的日志信息
- ◆ 使用 **FindBugs** 分析并发代码
- ◆ 配置 **Eclipse** 调试并发代码
- ◆ 配置 **NetBeans** 调试并发代码
- ◆ 使用 **MultithreadedTC** 测试并发代码

8.1 简介

对应用程序进行测试是一个关键的任务。在将应用程序交付给最终客户使用前，必须检验应用程序的正确性。我们使用测试流程来证明应用程序的正确性。测试是软件开发的基本环节，也称为**质量保证（Quality Assurance）**过程，可以找到大量文章介绍关于测试过程的不同方法，这些方法可用于应用程序开发当中。也有大量的类包（如单元测试框架**JUnit**）以及应用程序（如 Apache JMeter），它们可用来以一种自动化方式测试 Java 应用，在并发应用程序开发中进行测试则显得更加关键。

并发应用程序实际上是多个线程共享数据结构并相互交互，这在很大程度上增加了测试的难度。在测试并发应用程序时，要面对的最大的问题是线程的执行是不确定的（**Non-Deterministic**），我们无法保证线程执行的顺序，所以很难重现错误。

在本章，我们将学习下列内容：

- ◆ 如何获取并发应用程序中元素的信息，这些信息有助于测试应用程序；
- ◆ 如何使用 IDE（Integrated Development Environment）和其他工具（如 FindBugs）来测试应用程序；
- ◆ 如何使用类库（如 MultithreadedTC）来进行自动化测试。

8.2 监控 Lock 接口

Lock 接口是 Java 并发 API 同步代码块的基本机制之一。它定义了临界区（**Critical Section**）。临界区是同一时间只能被一个线程执行的共享资源的代码块。这种机制是通过 **Lock** 接口和 **ReentrantLock** 类而实现的。

在本节，我们将学习 **Lock** 对象提供的信息以及如何获取这些信息。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **MyLock** 的类，继承 **ReentrantLock** 类。

```
public class MyLock extends ReentrantLock {
```

2. 实现 **getOwnerName()** 方法。它使用了 **Lock** 类受保护的（访问权限为 **protected**）方法 **getOwner()**，返回当前获得了锁（如果有）的线程的名字。

```
public String getOwnerName() {
    if (this.getOwner() == null) {
```

```

        return "None";
    }
    return this.getOwner().getName();
}

```

3. 实现 **getThreads()** 方法。它使用了 **Lock** 类受保护的方法 **getQueuedThreads()**，返回正等待获取此锁的线程列表。

```

public Collection<Thread> getThreads() {
    return this.getQueuedThreads();
}

```

4. 创建一个名为 **Task** 的类，并且实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

5. 声明一个名为 **lock** 的私有 **Lock** 属性。

```
private Lock lock;
```

6. 实现类的构造器，用来初始化它的属性。

```

public Task (Lock lock) {
    this.lock=lock;
}

```

7. 实现 **run()** 方法。创建一个 5 次循环。

```

@Override
public void run() {
    for (int i=0; i<5; i++) {

```

8. 使用 **lock()** 方法取得锁，并在控制台输出一条消息。

```

lock.lock();
System.out.printf("%s: Get the Lock.\n", Thread.
    currentThread().getName());

```

9. 将线程休眠 500 毫秒。调用 **unlock()** 方法释放锁并在控制台输出一条消息。

```

try {
    TimeUnit.MILLISECONDS.sleep(500);
    System.out.printf("%s: Free the Lock.\n", Thread.

```

```
        currentThread().getName());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

10. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```
public class Main {  
    public static void main(String[] args) throws Exception {
```

11. 创建一个名为 **lock** 的 **MyLock** 对象。

```
MyLock lock=new MyLock();
```

12. 创建一个长度为 5 的线程数组。

```
Thread threads []=new Thread[5];
```

13. 创建并启动 5 个线程来执行 5 个 **Task** 对象。

```
for (int i=0; i<5; i++) {  
    Task task=new Task(lock);  
    threads[i]=new Thread(task);  
    threads[i].start();  
}
```

14. 创建一个 15 次的循环。

```
for (int i=0; i<15; i++) {
```

15. 在控制台输出锁的拥有者的名字。

```
System.out.printf("Main: Logging the Lock\n");
System.out.printf("*****\n");
System.out.printf("Lock: Owner : %s\n", lock.getOwnerName());
```

16. 输出正等待获取此锁的线程数和名称。

```

out.printf("Lock: Queued Threads: %s\n", lock.hasQueuedThreads());
if (lock.hasQueuedThreads()) {
    System.out.printf("Lock: Queue Length: %d\n", lock.
        getQueueLength());
    System.out.printf("Lock: Queued Threads: ");
    Collection<Thread> lockedThreads=lock.getThreads();
    for (Thread lockedThread : lockedThreads) {
        System.out.printf("%s ", lockedThread.getName());
    }
    System.out.printf("\n");
}

```

17. 输出 **Lock** 对象的公平 (Fair) 模式和状态信息。

```

System.out.printf("Lock: Fairness: %s\n", lock.isFair());
System.out.printf("Lock: Locked: %s\n", lock.isLocked());
System.out.printf("*****\n");

```

18. 将线程休眠 1 秒。

```

TimeUnit.SECONDS.sleep(1);
}
}
}

```

工作原理

本节范例通过继承 **ReentrantLock** 类实现 **MyLock** 类，用来返回 **ReentrantLock** 类的受保护数据，**ReentrantLock** 类中这些数据的默认访问权限是 `protected`，所以这些数据可能无法访问。**MyLock** 类实现了如下方法。

- ◆ **getOwnerName()**: 只有一个线程能够执行 **Lock** 对象保护的临界区。锁存储了执行临界区代码的线程。**ReentrantLock** 类的受保护方法 **getOwner()** 返回这个线程的名称。
- ◆ **getThreads()**: 当一个线程正在执行临界区代码时，其他试图进入的线程将被休眠直至它们能够执行临界区代码。

ReentrantLock 类的受保护方法 **getQueuedThreads()** 返回等待执行临界区代码的线程列表。这个方法直接返回 **getQueuedThreads()** 方法的返回结果。

我们已使用 **ReentrantLock** 类实现的其他方法如下。

- ◆ **hasQueuedThreads()**: 返回的 `boolean` 值表明是否有线程正在等待获取锁。

- ◆ **getQueueLength()**: 返回正在等待获取锁的线程数。
- ◆ **isLocked()**: 返回的 boolean 值表示这个锁是否被一个线程占有。
- ◆ **isFair()**: 返回的 boolean 值表示该锁是否为公平模式。

更多信息

ReentrantLock 类还提供了其他方法来获取 **Lock** 对象信息。

- ◆ **getHoldCount()**: 返回当前线程获取到锁的次数。
- ◆ **isHeldByCurrentThread()**: 返回的 boolean 值表示当前线程是否正持有此锁。

参见

- ◆ 参见 2.5 节。
- ◆ 参见 7.9 节。

8.3 监控 Phaser 类

Java 并发 API 提供的最复杂最强大的功能之一是使用 **Phaser** 类执行并发阶段任务 (**Concurrent Phased Task**)。当有并发任务被分为几步时这一机制非常有用。**Phaser** 类提供了在每步结束时同步等待其他线程的机制，因此，只有所有线程都执行完各自的第一步操作之后，才会开始执行接下来的第二步操作。

在本节，我们将学习 **Phaser** 类的状态信息，以及如何获取这些信息。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE (比如 NetBeans)，都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

2. 声明一个名为 **time** 的私有 **int** 属性。

```
private int time;
```

3. 声明一个名为 **phaser** 的私有 **Phaser** 属性。

```
private Phaser phaser;
```

4. 实现类的构造器，用来初始化它的属性。

```
public Task(int time, Phaser phaser) {
    this.time=time;
    this.phaser=phaser;
}
```

5. 实现 **run()** 方法。使用 **phaser** 属性的 **arrive()** 方法启动任务。

```
@Override
public void run() {
    phaser.arrive();
```

6. 在控制台输出一条消息表示阶段一开始，然后让线程休眠 **time** 属性指定的时长。接着在控制台输出一条消息表示阶段一结束，然后调用 **phaser** 属性的方法 **arriveAndAwaitAdvance()** 与其他的线程同步。

```
System.out.printf("%s: Entering phase 1.\n", Thread.
    currentThread().getName());
try {
    TimeUnit.SECONDS.sleep(time);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.printf("%s: Finishing phase 1.\n", Thread.
    currentThread().getName());
phaser.arriveAndAwaitAdvance();
```

7. 为阶段二和阶段三重复上述行为。在阶段三结束时，使用 **arriveAndDeregister()** 方法，而不是 **arriveAndAwaitAdvance()** 方法。

```
System.out.printf("%s: Entering phase 2.\n", Thread.
    currentThread().getName());
try {
    TimeUnit.SECONDS.sleep(time);
```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("%s: Finishing phase 2.\n", Thread.
        currentThread().getName());
    phaser.arriveAndAwaitAdvance();
    System.out.printf("%s: Entering phase 3.\n", Thread.
        currentThread().getName());
    try {
        TimeUnit.SECONDS.sleep(time);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("%s: Finishing phase 3.\n", Thread.
        currentThread().getName());
    phaser.arriveAndDeregister();
}

```

8. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```

public class Main {
    public static void main(String[] args) throws Exception {

```

9. 创建一个名为 **phaser** 的新 **Phaser** 对象，有 3 个参与者。

```
    Phaser phaser=new Phaser(3);
```

10. 创建并启动 3 个线程来执行 3 个任务对象。

```

    for (int i=0; i<3; i++) {
        Task task=new Task(i+1, phaser);
        Thread thread=new Thread(task);
        thread.start();
    }
}

```

11. 创建一个 10 步循环，输出 **phaser** 对象的信息。

```
for (int i=0; i<10; i++) {
```

12. 输出关于已注册方的信息、**phaser** 的阶段、已到达方和未到达方。

```

    for(int i=0; i<10; i++) {
        System.out.printf("*****\n");
        System.out.printf("Main: Phaser Log\n");
        System.out.printf("Main: Phaser: Phase: %d\n", phaser.

```

```

        getPhase());
System.out.printf("Main: Phaser: Registered Parties:
    %d\n",phaser.getRegisteredParties());
System.out.printf("Main: Phaser: Arrived Parties:
    %d\n",phaser.getArrivedParties());
System.out.printf("Main: Phaser: Unarrived Parties:
    %d\n",phaser.getUnarrivedParties());
System.out.printf("*****\n");

```

13. 让线程休眠 1 秒钟。

```

        TimeUnit.SECONDS.sleep(1);
    }
}
}

```

工作原理

本节范例在 **Task** 类中实现了一个阶段任务 (**Phase Task**)。该任务有 3 个阶段并使用一个 **Phaser** 接口与其他 **Task** 对象进行同步。主类启动 3 个任务并且当这些任务都在执行各自阶段时，主类在控制台输出关于 **phaser** 对象状态的信息。使用下列方法得到 **phaser** 对象的状态。

- ◆ **getPhase()**: 这个方法返回 **phaser** 对象的当前阶段。
- ◆ **getRegisteredParties()**: 这个方法返回使用 **phaser** 对象作为同步机制的任务数。
- ◆ **getArrivedParties()**: 这个方法返回在一个阶段结束时已到达的任务数。
- ◆ **getUnarrivedParties()**: 这个方法返回在一个阶段结束时未到达的任务数。

下面的截图展示了范例执行的部分结果。

```

Problems @ Javadoc Console Declarat
<terminated> Main (53) [Java Application] E:\Java 7 Concurrer
*****
Main: Phaser Log
Main: Phaser: Phase: 2
Main: Phaser: Registered Parties: 3
Thread-2: Finishing phase 2.
Main: Phaser: Arrived Parties: 2
Main: Phaser: Unarrived Parties: 3
*****
Thread-0: Entering phase 3.
Thread-1: Entering phase 3.
Thread-2: Entering phase 3.
Thread-0: Finishing phase 3.
*****
Main: Phaser Log
Main: Phaser: Phase: 3
Main: Phaser: Registered Parties: 2
Main: Phaser: Arrived Parties: 0
Main: Phaser: Unarrived Parties: 2
*****

```

参见

- ◆ 参见 3.6 节。

8.4 监控执行器框架

执行器框架（**Executor Framework**）提供了一套机制把任务的实现和创建与管理执行这些任务的线程分开。如果使用一个执行器（Executor），必须实现 **Runnable** 对象并将其发送到执行器。执行器负责管理线程，当有任务到达时，它将尝试使用线程池中的线程来执行这个任务，从而避免创建新的线程。**Executor** 接口和它的实现类 **ThreadPoolExecutor** 提供了这套机制。

在本节，我们将学习能够获取到的 **ThreadPoolExecutor** 类的状态信息有哪些，以及如何获取这些信息。

准备工作

本节的范例是在 EclipseIDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

2. 声明一个名为 **milliseconds** 的私有 **long** 属性。

```
private long milliseconds;
```

3. 实现类的构造器，用来初始化它的属性。

```
public Task (long milliseconds) {
    this.milliseconds=milliseconds;
}
```

4. 实现 **run()**方法。让线程休眠由 **milliseconds** 属性指定的毫秒数。

```

@Override
public void run() {
    System.out.printf("%s: Begin\n", Thread.currentThread().
        getName());
    try {
        TimeUnit.MILLISECONDS.sleep(milliseconds);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("%s: End\n", Thread.currentThread().
        getName());
}

```

5. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```

public class Main {
    public static void main(String[] args) throws Exception {

```

6. 调用 **Executors** 工厂类的 **newCachedThreadPool()**方法创建一个新的 **Executor** 对象。

```

ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.
    newCachedThreadPool();

```

7. 创建并提交 10 个 **Task** 任务对象到 **executor** 对象。用随机数来初始化这些任务对象。

```

Random random=new Random();
for (int i=0; i<10; i++) {
    Task task=new Task(random.nextInt(10000));
    executor.submit(task);
}

```

8. 创建一个 5 次循环。在每一次循环中，调用 **showLog()**方法输出关于 **executor** 的信息并让线程休眠 1 秒钟。

```

for (int i=0; i<5; i++) {
    showLog(executor);
    TimeUnit.SECONDS.sleep(1);
}

```

9. 调用 **shutdown()**方法关闭 **executor**。

```
executor.shutdown();
```

10. 创建另一个 5 次循环。在每一次循环中，调用 **showLog()**方法输出关于 **executor** 的信息并让线程休眠 1 秒钟。

```
for (int i=0; i<5; i++) {
    showLog(executor);
    TimeUnit.SECONDS.sleep(1);
}
```

11. 调用 **awaitTermination()**方法等待 **executor** 结束。

```
executor.awaitTermination(1, TimeUnit.DAYS);
```

12. 在程序结束时输出一条消息。

```
System.out.printf("Main: End of the program.\n");
}
```

13. 实现接收 **Executor** 作为参数的 **showLog()**方法。输出线程池的大小、任务数和执行器 **executor** 的状态。

```
private static void showLog(ThreadPoolExecutor executor) {
    System.out.printf("*****\n");
    System.out.printf("Main: Executor Log");
    System.out.printf("Main: Executor: Core Pool Size:
        %d\n", executor.getCorePoolSize());
    System.out.printf("Main: Executor: Pool Size: %d\n", executor.
        getPoolSize());
    System.out.printf("Main: Executor: Active Count:
        %d\n", executor.getActiveCount());
    System.out.printf("Main: Executor: Task Count: %d\n", executor.
        getTaskCount());
    System.out.printf("Main: Executor: Completed Task Count:
        %d\n", executor.getCompletedTaskCount());
    System.out.printf("Main: Executor: Shutdown: %s\n", executor.
        isShutdown());
    System.out.printf("Main: Executor: Terminating:
        %s\n", executor.isTerminating());
    System.out.printf("Main: Executor: Terminated: %s\n", executor.
        isTerminated());
```

```

        System.out.printf("*****\n");
    }
}

```

工作原理

在本节，我们实现了一个任务，这个任务阻塞它的执行线程随机毫秒数。接下来，发送 10 个任务到执行器中，并等待它们执行结束，在控制台输出关于执行器的状态信息。使用下列方法获取 **Executor** 对象的状态。

- ◆ **getCorePoolSize()**: 这个方法返回一个 **int** 整数，表示线程池中的核心线程数。它是当执行器不执行任务时，在内部线程池中的最小线程数。
- ◆ **getPoolSize()**: 这个方法返回一个 **int** 整数，它是内部线程池的实际大小。
- ◆ **getActiveCount()**: 这个方法返回一个 **int** 整数，它是当前正在执行任务的线程数。
- ◆ **getTaskCount()**: 这个方法返回一个 **long** 长整数，它是计划将被执行的任务数。
- ◆ **getCompletedTaskCount()**: 这个方法返回一个 **long** 长整数，它是已被执行器执行的且已完成执行的任务数。
- ◆ **isShutdown()**: 当调用了执行器的 **shutdown()** 方法来结束它的执行时，方法会返回一个 **boolean** 布尔值。
- ◆ **isTerminating()**: 当执行器正在执行 **shutdown()** 方法但还没有执行完成时，这个方法返回一个 **boolean** 布尔值。
- ◆ **isTerminated()**: 当执行器已完成执行时，方法返回一个 **boolean** 布尔值。

参见

- ◆ 参见 4.2 节。
- ◆ 参见 7.2 节。
- ◆ 参见 7.3 节。

8.5 监控 Fork/Join 池

执行器框架（**Executor Framework**）提供了一套机制把任务的实现和创建与管理执行这些任务的线程分开。Java 7 引入了执行器框架的一个扩展，针对一些特定问题，比其他

解决方案(如直接使用 **Thread** 对象或 **Executor** 框架)具有更高的性能, 这就是 **Fork / Join** 框架 (**Fork / Join Framework**)。

Fork / Join 框架被设计用来解决能够被分解成更小任务的问题, 通过分治技术(**Divide and Conquer Technique**), 采用 **fork()**和 **join()**方法运行操作。实现这种行为的主类是 **ForkJoinPool** 类。

在本节, 我们将学习能够获取到的 **ForkJoinPool** 类信息有哪些, 以及如何获取这些信息。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE (比如 NetBeans), 都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类, 并继承 **RecursiveAction** 类。

```
public class Task extends RecursiveAction{
```

2. 声明一个名为 **array** 的私有 **int** 数组属性, 用来存储将要增加的元素。

```
private int array[];
```

3. 声明两个分别名为 **start** 和 **end** 的私有 **int** 属性, 用来存储任务将要处理的元素块的开始和结束位置。

```
private int start;
private int end;
```

4. 实现类的构造器, 用来初始化它的属性。

```
public Task (int array[], int start, int end) {
    this.array=array;
    this.start=start;
    this.end=end;
}
```

5. 实现 **compute()**方法, 它包含了任务的主要步骤。如果任务要处理的元素多于 100

个，则拆分元素集为两部分，再创建两个任务来执行这两部分。调用 **fork()** 方法开始执行，然后用 **join()** 方法等待这两个任务的结束。

```
protected void compute() {
    if (end-start>100) {
        int mid=(start+end)/2;
        Task task1=new Task(array,start,mid);
        Task task2=new Task(array,mid,end);

        task1.fork();
        task2.fork();

        task1.join();
        task2.join();
    }
}
```

6. 如果任务要处理的元素少于或等于 100 个，则通过循环将每个元素加 1，在每次操作后让线程休眠 5 毫秒，然后继续执行元素加 1 的操作。

```
} else {
    for (int i=start; i<end; i++) {
        array[i]++;
        try {
            Thread.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

7. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) throws Exception {
```

8. 创建一个名为 **pool** 的 **ForkJoinPool** 对象。

```
ForkJoinPool pool=new ForkJoinPool();
```

9. 创建一个名为 **array** 能够容纳 10,000 个元素的 **int** 数组。

```
int array[] = new int[10000];
```

10. 创建一个新 **Task** 对象来处理整个数组。

```
Task task1 = new Task(array, 0, array.length);
```

11. 调用 **execute()** 方法发送任务到线程池中去执行。

```
pool.execute(task1);
```

12. 当任务未执行完成时，调用 **showLog()** 方法输出 **ForkJoinPool** 类的状态信息并让线程休眠 1 秒钟。

```
while (!task1.isDone()) {
    showLog(pool);
    TimeUnit.SECONDS.sleep(1);
}
```

13. 调用 **shutdown()** 方法关闭池。

```
pool.shutdown();
```

14. 调用 **awaitTermination()** 方法等待池的结束。

```
pool.awaitTermination(1, TimeUnit.DAYS);
```

15. 调用 **showLog()** 方法输出 **ForkJoinPool** 类的状态信息，并在控制台输出一条消息表示程序结束。

```
showLog(pool);
System.out.printf("Main: End of the program.\n");
```

16. 实现 **showLog()** 方法。它接收一个 **ForkJoinPool** 对象作为参数，并输出关于它的状态和正在执行的线程和任务的信息。

```
private static void showLog(ForkJoinPool pool) {
    System.out.printf("*****\n");
    System.out.printf("Main: Fork/Join Pool log\n");
    System.out.printf("Main: Fork/Join Pool: Parallelism:");
    System.out.printf("%d\n", pool.getParallelism());
    System.out.printf("Main: Fork/Join Pool: Pool Size:
```

```

        %d\n", pool.getPoolSize());
System.out.printf("Main: Fork/Join Pool: Active Thread Count:
        %d\n", pool.getActiveThreadCount());
System.out.printf("Main: Fork/Join Pool: Running Thread Count:
        %d\n", pool.getRunningThreadCount());
System.out.printf("Main: Fork/Join Pool: Queued Submission:
        %d\n", pool.getQueuedSubmissionCount());
System.out.printf("Main: Fork/Join Pool: Queued Tasks:
        %d\n", pool.getQueuedTaskCount());
System.out.printf("Main: Fork/Join Pool: Queued Submissions:
        %s\n", pool.hasQueuedSubmissions());
System.out.printf("Main: Fork/Join Pool: Steal Count:
        %d\n", pool.getStealCount());
System.out.printf("Main: Fork/Join Pool: Terminated :
        %s\n", pool.isTerminated());
System.out.printf("*****\n");
}

```

工作原理

在本节，我们使用一个 **ForkJoinPool** 类和一个继承了 **RecursiveAction** 类的 **Task** 类，实现了一个任务——对一个数组中的元素加 1。**RecursiveAction** 类型的任务是能够在 **ForkJoinPool** 类中执行的一种。当任务正处理数组时，在控制台输出关于 **ForkJoinPool** 类的状态信息。使用下列方法获取 **ForkJoinPool** 类的状态信息。

- ◆ **getPoolSize()**: 这个方法返回一个 **int** 整数，表示 **Fork / Join** 线程池中工作者线程 (**Worker Thread**) 的数量。
- ◆ **getParallelism()**: 这个方法返回为池建立的期望的并发级别。
- ◆ **getActiveThreadCount()**: 这个方法返回当前正在执行任务的线程数。
- ◆ **getRunningThreadCount()**: 这个方法返回当前正在工作且未在任何同步机制中被阻塞的线程数。
- ◆ **getQueuedSubmissionCount()**: 这个方法返回已经提交到线程池但未开始执行的任务数。
- ◆ **getQueuedTaskCount()**: 这个方法返回已经提交到线程池且已开始执行的任务数。
- ◆ **hasQueuedSubmissions()**: 这个方法返回一个 **boolean** 布尔值，表示在线程池中是否有未开始执行的等待任务。

- ◆ **getStealCount()**: 这个方法返回一个 **long** 长整数，表示一个工作者线程从另一个线程中偷得的任务的次数。
- ◆ **isTerminated()**: 这个方法返回一个 **boolean** 布尔值，表示 **Fork / Join** 池是否已经完成执行。

参见

- ◆ 参见 5.2 节。
- ◆ 参见 7.7 节。
- ◆ 参见 7.8 节。

8.6 输出高效的日志信息

日志系统（**Log System**）是将信息输出到一个或多个目标上的一种机制。一个日志器（**Logger**）有下面几个组件。

- ◆ 一个或多个处理器（**Handler**）：处理器决定目标和日志消息的格式。可把日志消息输出到控制台上、写到文件中或保存到数据库中。
- ◆ 一个名称（**Name**）：一般来说，类中的日志记录器的名称是基于它的包名和类名的。
- ◆ 一个级别（**Level**）：日志消息有一个关联的级别来表示它的重要性。日志记录器也有一个级别用来决定它要输出什么级别的消息。日志记录器仅输出与它的级别相同重要或更重要的消息。

使用日志系统有下面两个主要目的：

- ◆ 当捕获到异常时尽可能多地输出信息，这有助于定位并解决错误；
- ◆ 输出关于程序正在执行的类和方法的信息。

在本节，我们将学习如何使用 **java.util.logging** 包提供的类来将一个日志系统增加到并发应用程序中。

准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如

NetBeans), 都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **MyFormatter** 的类, 继承 **java.util.logging.Formatter** 类。然后, 实现抽象 **format()** 方法。它以 **LogRecord** 对象为参数, 返回一个带有日志消息的 String 对象。

```
public class MyFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        StringBuilder sb=new StringBuilder();
        sb.append("[ "+record.getLevel()+" ] - ");
        sb.append(new Date(record.getMillis())+" : ");
        sb.append(record.getSourceClassName()+"."+record.
            getSourceMethodName()+" : ");
        sb.append(record.getMessage()+"\n");
        return sb.toString();
    }
}
```

2. 创建一个名为 **MyLogger** 的类。

```
public class MyLogger {
```

3. 声明一个名为 **handler** 的私有静态 **Handler** 属性。

```
private static Handler handler;
```

4. 实现公开的静态 **getLogger()** 方法, 创建用来输出日志消息的 **Logger** 对象。它接收一个名为 **name** 的字符串参数。

```
public static Logger getLogger(String name){
```

5. 调用 **Logger** 类的 **getLogger()** 方法, 传递参数 **name**, 得到与之关联的 **java.util.logging.Logger** 日志器。

```
Logger logger=Logger.getLogger(name);
```

6. 使用 **setLevel()** 方法设置日志级别, 这里输出所有的日志消息。

```
logger.setLevel(Level.ALL);
```

7. 如果 **handler** 属性值为 **null**, 就创建一个新的 **FileHandler** 对象, 用来输出日志消息到 **recipe8.log** 文件中。调用 **setFormatter()**方法, 为 **handler** 指定一个 **MyFormatter** 对象作为一个格式化器。

```
try {
    if (handler==null) {
        handler=new FileHandler("recipe8.log");
        Formatter format=new MyFormatter();
        handler.setFormatter(format);
    }
}
```

8. 如果没有一个处理程序与 **Logger** 对象相关联, 就调用 **addHandler()**方法增加之。

```
if (logger.getHandlers().length==0) {
    logger.addHandler(handler);
}
} catch (SecurityException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

9. 返回已创建的 **Logger** 对象。

```
return logger;
}
```

10. 创建一个名为 **Task** 的类, 实现 **Runnable** 接口。它用来测试 **Logger** 对象的任务。

```
public class Task implements Runnable {
```

11. 实现 **run()**方法。

```
@Override
public void run() {
```

12. 声明一个名为 **logger** 的 **Logger** 对象。调用 **MyLogger** 类的 **getLogger()**方法, 传递自身的类名作为参数来初始化 **Logger** 对象。

```
Logger logger= MyLogger.getLogger(this.getClass().getName());
```

13. 使用 **entering()**方法输出一条日志消息表示方法开始执行, 然后使线程休眠 2 秒钟。

```
logger.entering(Thread.currentThread().getName(), "run()");
```

```

Sleep the thread for two seconds
try {
    TimeUnit.SECONDS.sleep(2);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

14. 使用 **exiting()**方法输出一条日志消息表示方法执行结束。

```

logger.exiting(Thread.currentThread().getName(),
               "run()", Thread.currentThread());
}

```

15. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```

public class Main {
    public static void main(String[] args) {

```

16. 声明一个名为 **logger** 的 **Logger** 对象。调用 **MyLogger** 类的 **getLogger()**方法，以字符串 **Core** 作为参数来初始化 **Logger** 对象。

```
Logger logger=MyLogger.getLogger("Core");
```

17. 使用 **entering()**方法输出一条日志消息表示主程序开始执行。

```
logger.entering("Core", "main()", args);
```

18. 创建一个 **Thread** 数组存储 5 个线程。

```
Thread threads[]=new Thread[5];
```

19. 创建 5 个 **Task** 对象和 5 个线程来执行它们。输出日志消息表明将启动一个新线程且线程已创建。

```

for (int i=0; i<threads.length; i++) {
    logger.log(Level.INFO,"Launching thread: "+i);
    Task task=new Task();
    threads[i]=new Thread(task);
    logger.log(Level.INFO,"Thread created: "+ threads[i].
               getName());
    threads[i].start();
}

```

20. 输出一条日志消息表明已创建了线程。

```
logger.log(Level.INFO, "Ten Threads created."+
    "Waiting for its finalization");
```

21. 调用 **join()** 方法等待 5 个线程执行结束。在每个线程结束后，输出一条日志消息。

```
for (int i=0; i<threads.length; i++) {
    try {
        threads[i].join();
        logger.log(Level.INFO, "Thread has finished its
            execution", threads[i]);
    } catch (InterruptedException e) {
        logger.log(Level.SEVERE, "Exception", e);
    }
}
```

22. 调用 **exiting()** 方法输出一条日志消息表示主程序执行结束。

```
logger.exiting("Core", "main()");
}
```

工作原理

在本节，我们使用了来自 Java 日志 API 的 **Logger** 类在并发应用程序中输出日志消息。首先，实现 **MyFormatter** 类，为日志消息指定格式。这个类继承了 **Formatter** 类，它声明了抽象方法 **format()**。这个方法接收一个 **LogRecord** 对象，用来格式化给定的日志记录，并返回格式化后的字符串。在这个实现类中，已使用 **LogRecord** 类的下列方法来获取日志消息的信息。

- ◆ **getLevel()**: 返回消息级别。
- ◆ **getMillis()**: 返回发送消息到 **Logger** 对象时的时间(从 1970 开始计算的毫秒数)。
- ◆ **getSourceClassName()**: 返回发送消息到 **Logger** 的类的名称。
- ◆ **getSourceMessageName()**: 返回发送消息到 **Logger** 的方法的名称。

getMessage() 方法返回日志消息。**MyLogger** 类实现了静态方法 **getLogger()**，它创建一个 **Logger** 对象，并指定一个 **Handler** 对象来使用 **MyFormatter** 格式器输出应用程序的日志消息到 **recipe8.log** 文件中。本范例创建了带有静态方法 **getLogger()** 的 **Logger** 对象。这个方法根据所传递的参数返回不同的对象。因以仅创建了一个 **Handler** 对象，所以所有

的 **Logger** 对象都将日志消息输出到同一个文件中。已配置 **logger** 对象输出所有的日志消息，因此不需要考虑它的日志级别。

最后，实现了一个 **Task** 对象和一个主程序，输出不同的日志消息到日志文件中。使用了下列方法。

- ◆ **entering()**: 输出 **FINER** 级别的消息表示方法开始执行。
- ◆ **exiting()**: 输出 **FINER** 级别的消息表示方法停止执行。
- ◆ **log()**: 输出带有指定级别的消息。

更多信息

当使用一个日志系统时，需要重点考虑下列两点。

- ◆ **输出必需的信息**: 如果输出的日志信息太少，日志器将徒劳无功。如果输出的日志信息太多，将产生过大的日志文件，不仅不易于管理还难于从中获取必需的信息。
- ◆ **为消息设定恰当的级别**: 如果使用过高级别或过低级别消息，将会让阅读日志文件的人感到迷惑。在一个错误情况中很难知道发生了什么问题，或者得到了引起错误的过多信息，而无法快速找到主要原因。

也有其他类库比 **java.util.logging** 包提供了更完全的日志系统，如 **Log4j** 或 **slf4j** 类库。但是 **java.util.logging** 包是 Java API 的一部分，并且它的所有方法都是安全的，所以用在并发应用程序中没有问题。

参见

- ◆ 参见 6.2 节。
- ◆ 参见 6.3 节。
- ◆ 参见 6.4 节。
- ◆ 参见 6.5 节。
- ◆ 参见 6.6 节。
- ◆ 参见 6.7 节。

8.7 使用 FindBugs 分析并发代码

静态代码分析工具（**Static Code Analysis Tool**）是一套通过分析应用程序的源代码来查出潜在错误的工具集。这些工具，如 **Checkstyle**、**PMD** 或 **FindBugs**，提供了预定义的最佳实践的规则集来分析源代码、查找源代码是否违反了这些规则。这些工具的目标是，在应用程序正式投产前发现错误或定位性能瓶颈之处。编程语言通常提供了这类工具，Java 语言也不例外。分析 Java 代码的工具之一是 **FindBugs**。它是一个开源工具，包含了一系列规则来分析 Java 并发代码。

在本节，我们将学习如何使用 **FindBugs** 工具来分析 Java 并发应用程序。

准备工作

开始本节前，先从 FindBug 项目的网页 <http://findbugs.sourceforge.net/> 下载 FindBugs。可以下载一个独立应用程序或者 Eclipse 插件。在本节中，我们将使用独立版本。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task** 的类，并实现 **Runnable** 接口。

```
public class Task implements Runnable {
```

2. 声明一个名为 **lock** 的私有 **ReentrantLock** 属性。

```
private ReentrantLock lock;
```

3. 实现类的构造器。

```
public Task(ReentrantLock lock) {  
    this.lock=lock;  
}
```

4. 实现 **run()** 方法。获取对锁的控制，使线程休眠 2 秒后，释放锁。

```
@Override  
public void run() {
```

```

lock.lock();
try {
    TimeUnit.SECONDS.sleep(1);
    lock.unlock();
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

5. 实现范例的主类，创建 Main 主类，并实现 main()方法。

```

public class Main {
    public static void main(String[] args) {

```

6. 声明并创建一个名为 **lock** 的 **ReentrantLock** 对象。

```
        ReentrantLock lock=new ReentrantLock();
```

7. 创建 10 个 **Task** 对象和 10 个用来执行这些任务的线程。调用 **run()** 方法来启动线程。

```

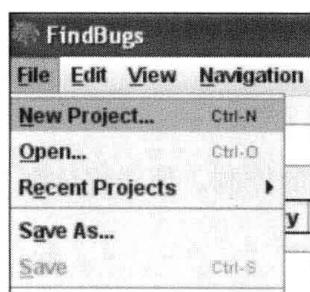
        for (int i=0; i<10; i++) {
            Task task=new Task(lock);
            Thread thread=new Thread(task);
            thread.run();
        }
    }
}

```

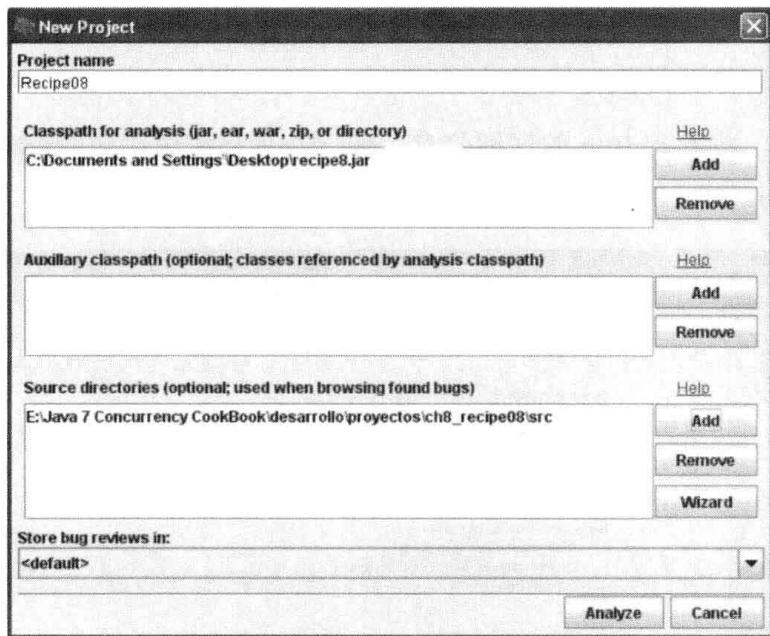
8. 将项目导出为一个.jar 文件，命名为 **recipe8.jar**。使用集成开发工具的菜单项或者用 **javac** 和 **jar** 命令来编译压缩应用程序。

9. 在 **Windows** 平台上通过运行 **findbugs.bat** 命令来启动 **FindBugs** 独立程序，在 **Linux** 平台上则通过执行 **findbugs.sh** 命令来启动。

10. 使用 **File** 菜单中的 **NewProject** 选项来创建一个新项目。



11. **FindBugs** 应用程序将显示一个窗口来配置项目。在 **ProjectName** 文本框中输入文本 **Recipe08**。在 **Classpath for analysis** 文本框中加入 **jar** 文件，在 **Source directories** 文本框中添加例子源代码的目录，如下图所示。



12. 单击 **Analyze** 按钮创建新项目并分析它的代码。
13. **FindBugs** 应用程序显示了代码分析的结果。在本例中，它发现了两个 Bug。
14. 单击一个 Bug，将在右边栏区域看到 Bug 对应的源代码和屏幕底部对 Bug 的描述。

工作原理

下面的截图显示了 **FindBugs** 的分析结果。

Group bugs by:	Category	Bug Kind	Bug Pattern	↔	Bug Rank	Designation	
Bugs (2)							
└ Multithreaded correctness (2)							
└ Lock not released on all paths (1)							
└ Method does not release lock on all exception paths (1)							
└ com.packtpub.java7.concurrency.chapter8.recipe06.task.Task.run() does not r							
└ Method invokes run() (1)							
└ Invokes run on a thread (did you mean to start it instead?) (1)							
└ com.packtpub.java7.concurrency.chapter8.recipe06.core.Core.main(String[]) e							

分析检测到在程序中有下列两个潜在 Bug。

- ◆ 一个在 **Task** 类的 **run()** 方法中。抛出 **InterruptedException** 异常时，任务不会释放锁，因为它执行不到 **unlock()** 方法。这可能会引起应用程序死锁。
- ◆ 另一个是 **Main** 类的 **main()** 方法。因为已经直接调用了一个线程的 **run()** 方法，但是没有调用 **start()** 方法开始执行线程。

双击一个 Bug，将看到 Bug 的详细信息。像已在项目配置中包含的源代码参考一样，将看到检测到的 Bug 对应的源代码。下面的截图显示了一个范例。

```

19     * Create a Lock
20     */
21     ReentrantLock lock=new ReentrantLock();
22
23     /**
24      * Executes the threads. There is a problem with this
25      * block of code. It uses the run() method instead of
26      * the start() method.
27      */
28     for (int i=0; i<10; i++) {
29         Task task=new Task(lock);
30         Thread thread=new Thread(task);
31         thread.run();
32     }
33
34 }
35
36 }
37

```

更多信息

注意：**FindBugs** 只能检测出一些问题（与并发代码相关或不相关）。例如，在 **Task** 类的 **run()** 方法中删除调用 **unlock()** 方法的代码，然后重新分析，**FindBugs** 将不会警告已在任务中获取到锁但永远不会释放它。

使用这个静态代码分析工具有助于提高代码的质量，但是不要期望它检测出代码中的所有 Bug。

参见

- ◆ 参见 8.9 节。

8.8 配置 Eclipse 调试并发代码

无论使用什么编程语言，如今几乎每个程序员都使用 IDE 集成开发环境来创建应用程序。IDE 提供了大量已集成的有趣的功能，如：

- ◆ 项目管理；
- ◆ 代码自动生成；
- ◆ 文档自动生成；
- ◆ 集成版本控制系统；
- ◆ 测试应用程序的调试器；
- ◆ 创建项目或应用元素的各种向导。

IDE 中最有帮助的特性之一就是调试器，它可以一步步地执行应用程序并分析程序中的对象和变量的值。

如果使用 Java 编程语言，**Eclipse** 是最流行的 IDE 之一。它有一个集成的调试器，用来测试应用程序。缺省时，当你调试一个并发应用程序并且发现一个断点，调试器仅停止有断点的线程而其他线程继续执行。

在本节，我们将学习如何改变 **Eclipse** 的配置来帮助测试并发应用程序。

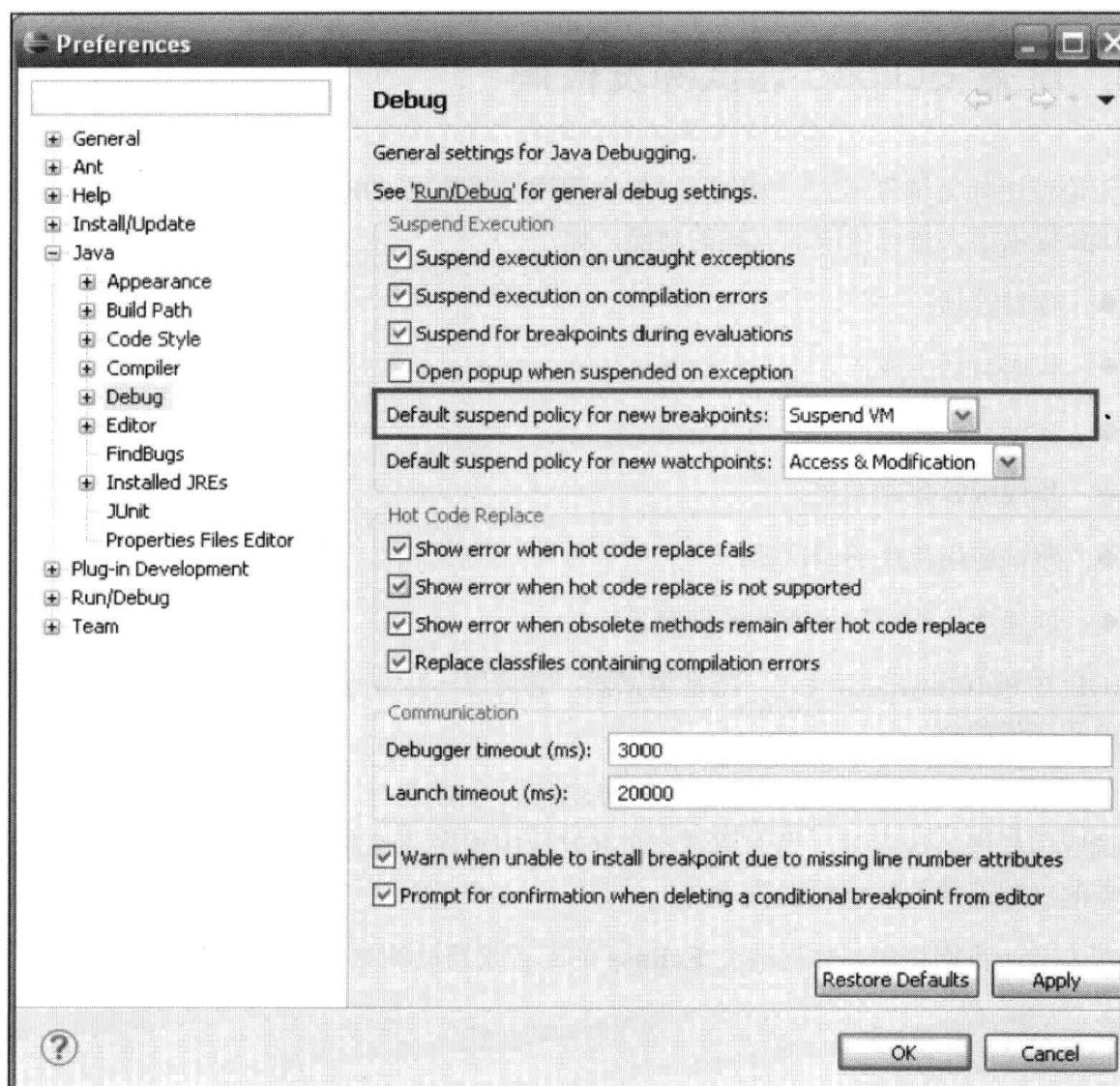
准备工作

本节的范例是在 Eclipse IDE 里完成的。无论你使用 Eclipse 还是其他的 IDE（比如 NetBeans），都可以打开这个 IDE 并且创建一个新的 Java 工程。

范例实现

按照接下来的步骤实现本节的范例。

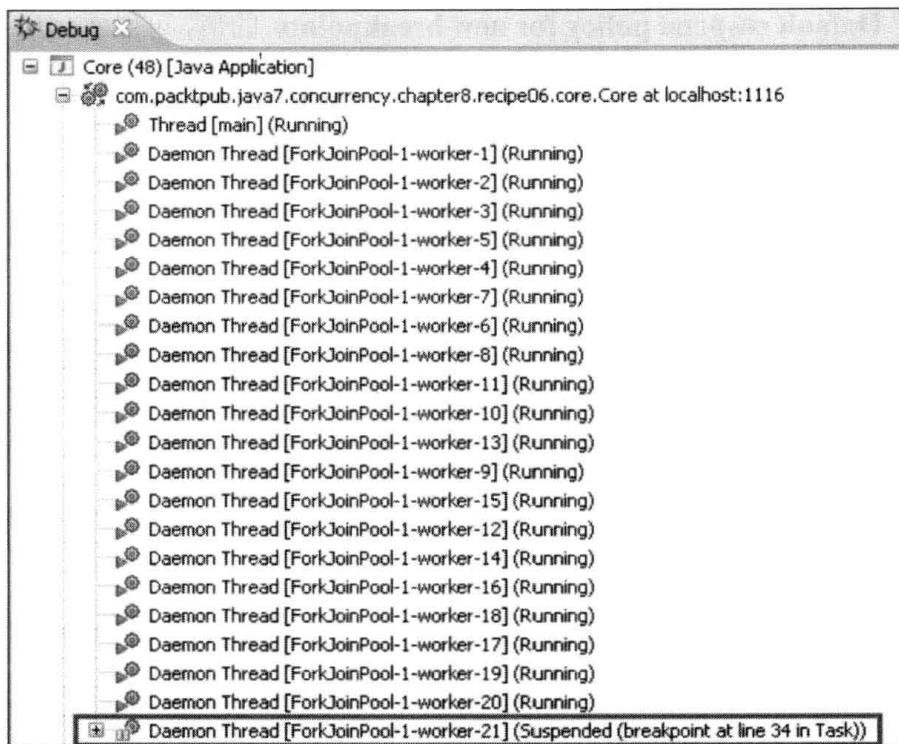
1. 选择菜单项 **Window**，然后单击 **Preferences** 选项。
2. 在左边菜单栏展开 **Java** 选项。
3. 在左边菜单栏选择 **Debug** 选项。下面的截图显示了相应窗口。



4. 设置 **Default suspend policy for new breakpoints** 的值为 **Suspend VM**（上述截图中带框的部分）。
5. 单击 **OK** 按钮确认更改。

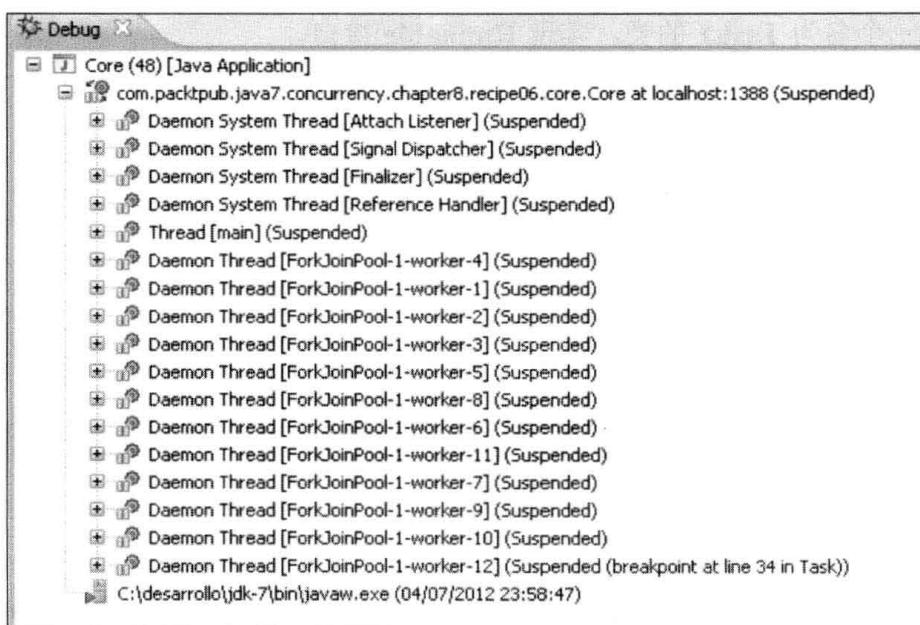
工作原理

如本节中提到的，缺省时，当你在 **Eclipse** 中调试并发应用程序并且发现一个断点时，调试器仅暂停有断点的线程而其他线程继续执行。下面的截图显示了一个实例。



可以看到，只有 **worker-21** 被暂停（图中标框的部分）而其他线程仍在执行。但是，如果改变 **Default suspend policy for new breakpoints** 的值为 **Suspend VM**，当调试一个并发应用程序到一个断点时，所有线程都会暂停执行。

下面的截图显示了一个实例。



通过改变 **Default suspend policy for new breakpoints** 的值，可以看到所有线程都被暂停了，可以继续调试任何想要的线程。应建议根据应用程序的需要来选择最能满足需求的线程暂停策略。

8.9 配置 NetBeans 调试并发代码

如今，开发应用程序时，选用相应的软件工具是非常必要的。应用程序需要满足公司的质量标准，未来易于修改且费用低耗时少。为了实现这个目标，使用同一界面下集成了多个工具（编译器和调试器）来促进应用程序开发的 IDE 极为重要。

如果使用 Java 编程语言，NetBeans 也是最流行的 IDE 之一。它有一个集成的调试器，用来测试应用程序。

在本节，我们将学习如何改变 NetBeans 的配置来帮助测试并发应用程序。

准备工作

需要先安装 NetBeans IDE 集成开发环境，然后启动 NetBeans 并创建一个新的 Java 项目。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **Task1** 的类，实现 **Runnable** 接口。

```
public class Task1 implements Runnable {
```

2. 声明两个名为 **lock1** 和 **lock2** 的私有 **Lock** 属性。

```
private Lock lock1, lock2;
```

3. 实现类的构造器，用来初始化它的属性。

```
public Task1 (Lock lock1, Lock lock2) {  
    this.lock1=lock1;  
    this.lock2=lock2;  
}
```

4. 实现 **run()** 方法。调用 **lock()** 方法获取到对 **lock1** 锁对象的控制，并在控制台输出

一条消息表示已获取 **lock1** 锁。

```
@Override
public void run() {
    lock1.lock();
    System.out.printf("Task 1: Lock 1 locked\n");
}
```

5. 调用 **lock()**方法获取到对 **lock2** 锁对象的控制，并在控制台输出一条消息表示已获取 **lock2** 锁。然后，释放这两个锁对象。先释放 **lock2** 锁对象，接着释放 **lock1** 锁对象。

```
lock2.lock();
System.out.printf("Task 1: Lock 2 locked\n");
lock2.unlock();
lock1.unlock();
}
```

6. 创建一个名为 **Task2** 的类，实现 **Runnable** 接口。

```
public class Task2 implements Runnable{
```

7. 声明两个名为 **lock1** 和 **lock2** 的私有 **Lock** 属性。

```
private Lock lock1, lock2;
```

8. 实现类的构造器，用来初始化它的属性。

```
public Task2(Lock lock1, Lock lock2) {
    this.lock1=lock1;
    this.lock2=lock2;
}
```

9. 实现 **run()**方法。调用 **lock()**方法获取对 **lock2** 锁对象的控制，在控制台输出一条消息表示已获取 **lock2** 锁。

```
@Override
public void run() {
    lock2.lock();
    System.out.printf("Task 2: Lock 2 locked\n");
```

10. 使用 **lock()**方法获取对 **lock1** 锁对象的控制，在控制台输出一条消息表示已获取 **lock1** 锁。

```
lock1.lock();
System.out.printf("Task 2: Lock 1 locked\n");
```

11. 释放这两个锁对象。先释放 **lock1** 锁对象，接着释放 **lock2** 锁对象。

```
lock1.unlock();
lock2.unlock();
}
```

12. 实现范例的主类，创建 **Main** 主类，并实现 **main()**方法。

```
public class Main {
```

13. 创建两个名为 **lock1** 和 **lock2** 的 **Lock** 锁对象。

```
Lock lock1, lock2;
lock1=new ReentrantLock();
lock2=new ReentrantLock();
```

14. 创建一个名为 **task1** 的 **Task1** 对象。

```
Task1 task1=new Task1(lock1, lock2);
```

15. 创建一个名为 **task2** 的 **Task2** 对象。

```
Task2 task2=new Task2(lock1, lock2);
```

16. 使用两个线程执行两个任务。

```
Thread thread1=new Thread(task1);
Thread thread2=new Thread(task2);

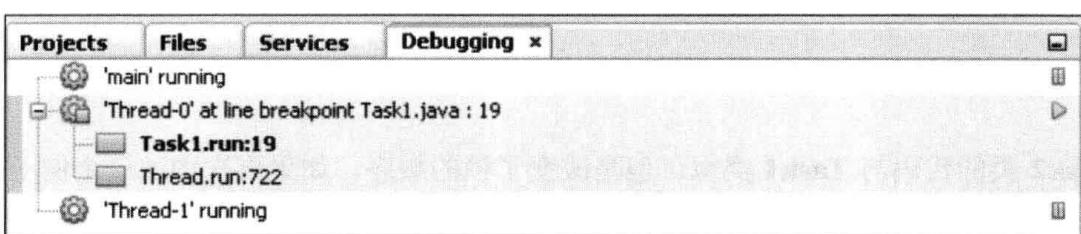
thread1.start();
thread2.start();
```

17. 当两个任务未完成执行时，每 500 毫秒在控制台输出一条消息。调用 **isAlive()**方法检查线程是否依然在执行。

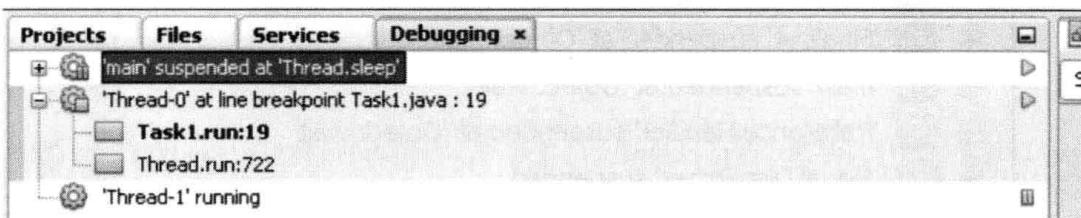
```
while ((thread1.isAlive()) && (thread2.isAlive())) {
    System.out.println("Main: The example is"+ "running");
    try {
```

```
        TimeUnit.MILLISECONDS.sleep(500);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

18. 在第一次调用 **Task1** 类的 **run()**方法中的 **println()**方法上增加一个断点。
 19. 调试程序。将在 **NetBeans** 主窗口左上角看到调试窗口。下面的截图显示了这个窗口。执行 **Task1** 对象的线程在休眠，因为它们已到达断点且其他线程正在运行。



20. 暂停主线程的执行。选择线程并右键单击，选择 **Suspend** 选项。下面的截图显示了调试窗口。



21. 恢复先前的两个暂停的线程。选择每一个线程并右键单击，然后选择 **Resume** 选项。

工作原理

当使用 **NetBeans** 调试并发应用程序到达一个断点时, **NetBeans** 暂停线程, 在左上角显示正在运行的线程的 **Debugging** 窗口。

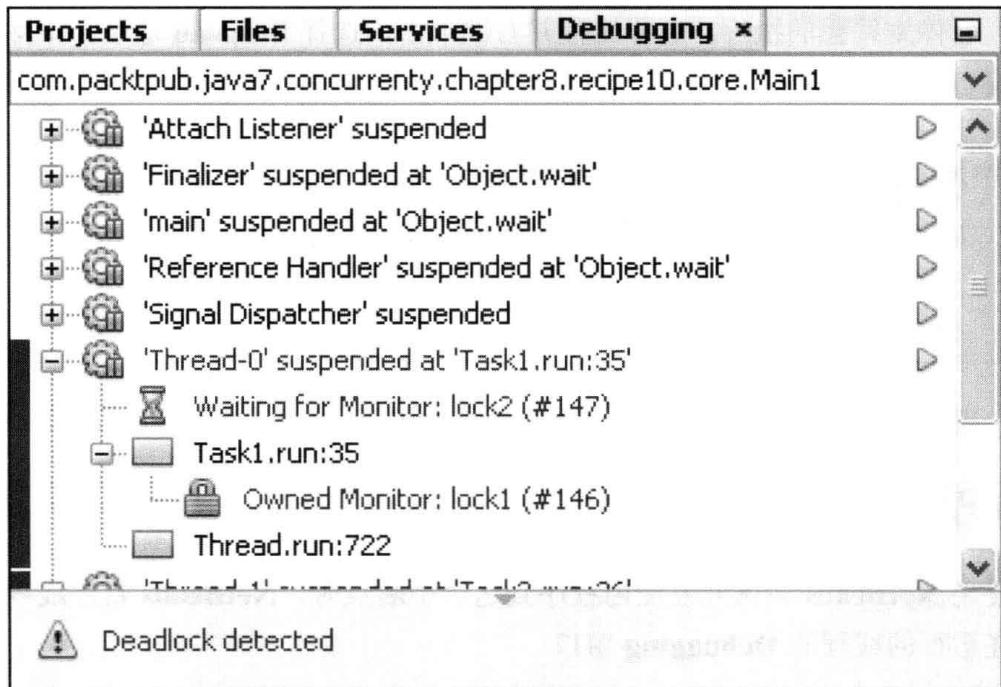
使用 **Pause** 或 **Resume** 选项暂停或恢复当前正在运行的线程。使用 **Variables** 标签可以看到当前线程的变量或属性的值。

NetBeans 也包含一个死锁探测器。当在 **Debug** 菜单中选择 **CheckforDeadlock** 选项时, **NetBeans** 对当前正在调试的应用程序执行一个分析来决定是否有死锁存在。这个范例提供了一个清晰的死锁: 第一个线程首先获取锁 **lock1**, 接着获取锁 **lock2**; 第二个线程按逆序

获取这两个锁。插入断点将引发死锁，但是如果使用 **NetBeans** 死锁检测器，将发现没有死锁，所以应该谨慎使用这个选项。使用 **synchronized** 关键字在两个任务中改变锁并再次调试。**Task1** 的代码如下：

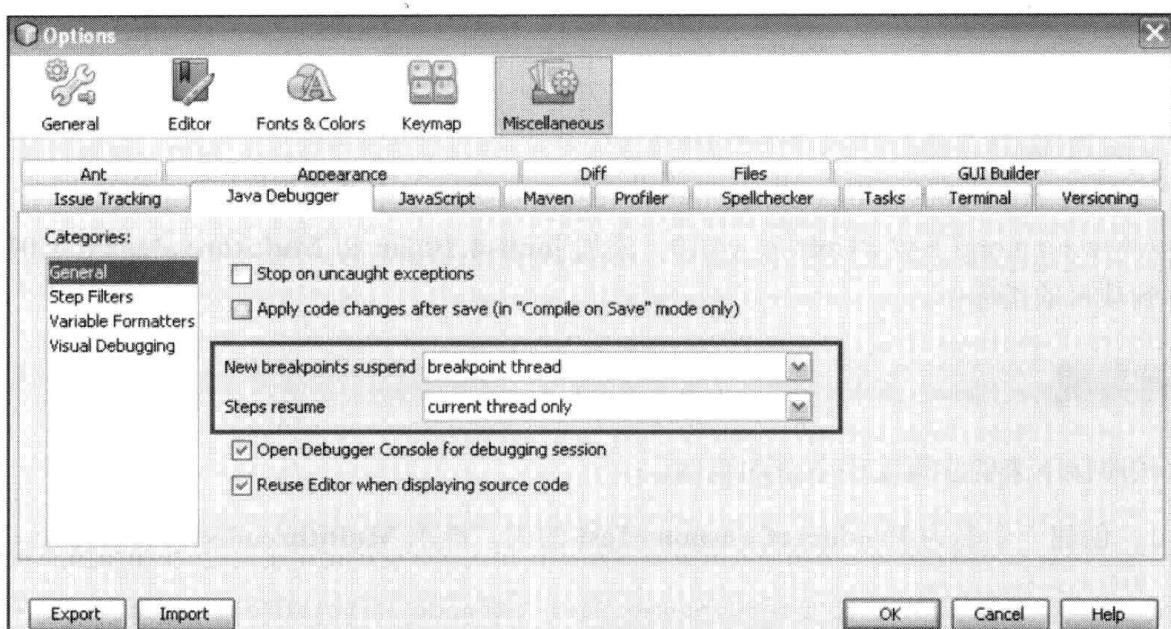
```
@Override
public void run() {
    synchronized(lock1) {
        System.out.printf("Task 1: Lock 1 locked\n");
        synchronized(lock2) {
            System.out.printf("Task 1: Lock 2 locked\n");
        }
    }
}
```

Task2 类的代码与 **Task1** 类似，但是改变了锁的顺序。如果再次调试这个例子，将又会得到一个死循环。但是在这个例子中，它会由死锁检测器检测到，如下图所示。



更多信息

NetBeans 还提供了很多其他选项来配置调试器。选择 **Tools** 菜单下的 **Options** 选项，接着选择 **Miscellaneous** 选项和 **Java Debugger** 标签进行查看。下面的截图显示了这个窗口。



在这个窗口上有两个选项可用来控制之前描述的行为。

- ◆ **New breakpoints suspend:** 使用这个选项，可以配置 **NetBeans** 在线程中发现一个断点时，只暂停有断点的线程还是暂停所有线程。
- ◆ **Steps resume:** 使用这个选项，可以配置 **NetBeans** 在恢复线程时，只恢复当前线程还是恢复所有线程。

两个选项已在之前的截图中标出。

参见

- ◆ 参见 8.8 节。

8.10 使用 MultithreadedTC 测试并发代码

MultithreadedTC 是一个测试并发应用程序的 Java 类库，它的主要目标是解决不确定的并发应用程序存在的问题。而我们无法控制线程的执行顺序。为了这个目标，**MultithreadedTC** 用一个内部节拍器（metronome）来控制应用程序不同线程的执行顺序。这些测试线程被实现为类的方法。

在本节，我们将学习如何使用 **MultithreadedTC** 类库来为 **LinkedTransferQueue** 类进

行测试。

准备工作

需要从 <http://code.google.com/p/multithreadedtc/> 网站上下载 **MultithreadedTC** 库，从 <http://www.junit.org/> 下载 **JUnit** 库 v4.10。添加 **junit-4.10.jar** 和 **MultithreadedTC-1.01.jar** 文件到项目的类库中。

范例实现

按照接下来的步骤实现本节的范例。

1. 创建一个名为 **ProducerConsumerTest** 的类，继承 **MultithreadedTestCase** 类。

```
public class ProducerConsumerTest extends MultithreadedTestCase {
```

2. 声明一个名为 **queue** 的私有 **LinkedTransferQueue** 属性，泛型参数为 **String** 类型。

```
private LinkedTransferQueue<String> queue;
```

3. 实现 **initialize()** 方法。这个方法不接收任何参数而且也不返回值，它调用父类的 **initialize()** 方法，接着初始化 **queue** 属性。

```
@Override
public void initialize() {
    super.initialize();
    queue=new LinkedTransferQueue<String>();
    System.out.printf("Test: The test has been initialized\n");
}
```

4. 实现 **thread1()** 方法。它将实现第一个消费者的逻辑。调用 **queue** 的 **take()** 方法，接着在控制台输出返回值。

```
public void thread1() throws InterruptedException {
    String ret=queue.take();
    System.out.printf("Thread 1: %s\n",ret);
}
```

5. 实现 **thread2()** 方法。它将实现第二个消费者的逻辑。在 **take()** 方法中使用 **waitForTick()** 方法等待直至第一个线程休眠。接着，调用 **queue** 的 **take()** 方法，并在控制

台输出返回值。

```
public void thread2() throws InterruptedException {
    waitForTick(1);
    String ret=queue.take();
    System.out.printf("Thread 2: %s\n",ret);
}
```

6. 实现 **thread3()** 方法。它将实现一个生产者的逻辑。在 **take()** 方法中使用两次 **waitForTick()** 方法等待直至两个消费者都被阻塞。接着，调用 **queue** 的 **put()** 方法插入两个字符串到 **queue** 中。

```
public void thread3() {
    waitForTick(1);
    waitForTick(2);
    queue.put("Event 1");
    queue.put("Event 2");
    System.out.printf("Thread 3: Inserted two elements\n");
}
```

7. 实现 **finish()** 方法。在控制台输出一条消息表示测试已执行完成。使用 **assertEquals()** 方法来检查两个事件是否已被消费（被消费 **queue** 的大小为 0）。

```
public void finish() {
    super.finish();
    System.out.printf("Test: End\n");
    assertEquals(true, queue.size()==0);
    System.out.printf("Test: Result: The queue is empty\n");
}
```

8. 实现范例的主类，创建 **Main** 主类，并实现 **main()** 方法。

```
public class Main {
    public static void main(String[] args) throws Throwable {
```

9. 创建一个名为 **test** 的 **ProducerConsumerTest** 对象。

```
ProducerConsumerTest test=new ProducerConsumerTest();
```

10. 调用 **TestFramework** 类的 **runOnce()** 方法来执行测试。

```
System.out.printf("Main: Starting the test\n");
```

```
TestFramework.runOnce(test);
System.out.printf("Main: The test has finished\n");
```

工作原理

本节范例使用 **MultithreadedTC** 库为 **LinkedTransferQueue** 类实现了一个测试。可以使用这个库和它的节拍器测试任何并发应用程序和类。在这个例子中，使用两个消费者和一个生产者实现了典型的生产者—消费者问题。我们要测试的是进入缓冲区中的第一个 **String** 对象被到达缓冲区的第一个消费者消费，进入缓冲区中的第二个 **String** 对象被到达缓冲区的第二个消费者消费。

MultithreadedTC 库基于 **JUnit** 库展开，**JUnit** 则是 Java 中用来实现单元测试的最常用的类库。如果使用 **MultithreadedTC** 库来实现一个测试，必须继承 **MultithreadedTestCase** 类。**MultithreadedTestCase** 类继承了包含检查测试结果的所有方法的 **junit.framework.Assert** 类，但是它没有继承 **junit.framework.TestCase** 类，所以不能够将 **MultithreadedTC** 测试和其他 **JUnit** 的测试集成在一起。

接着，实现下列方法。

- ◆ **initialize()**: 这个方法的实现是可选的。测试一开始它被执行，所以可以使用它来初始化在测试中将要使用的对象。
- ◆ **finish()**: 这个方法的实现是可选的。当结束测试时它才被执行，所以可以使用它来关闭或释放测试中用到的资源，或检查测试结果。
- ◆ **实现测试的方法**: 这些方法含有测试的主要逻辑，必须以 **thread** 关键字开始并跟随一个字符串。比如 **thread1()** 方法。

使用 **waitForTick()** 方法来控制线程的执行顺序。**waitForTick()** 方法接收一个 **int** 类型作为参数，并让执行方法的线程休眠，直至运行测试的所有线程都被阻塞。当所有线程都被阻塞时，**MultithreadedTC** 库通过调用 **waitForTick()** 方法来恢复被阻塞的线程。

waitForTick() 方法的传入参数是一个整数，它被用来控制线程的执行顺序。**MultithreadedTC** 库的节拍器（**Metronome**）有一个内部计数器，当所有线程被阻塞后，**MultithreadedTC** 库的计数器就会根据被阻塞的 **waitForTick()** 方法调用时所指定的数字大小来不断进行设置，然后比这个数字小的线程就会先开始执行，以此类推，这样就能够控制线程的执行顺序了。

在内部实现机制中，当 **MultithreadedTC** 库开始执行一个测试时，首先执行 **initialize()**

方法。接着，它为每个以 **thread** 关键字开头的方法创建一个线程，如方法 **thread1()**、**thread2()** 和 **thread3()**，当所有线程执行完时，再执行 **finish()** 方法。为了执行测试，需要调用 **TestFramework** 类的 **runOnce()** 方法。

更多信息

如果 **MultithreadedTC** 库检测到测试的所有线程均被阻塞，但是没有一个线程在 **waitForTick()** 方法中被阻塞，那么测试将被当成是死锁状态并且抛出 **java.lang.IllegalStateException** 异常。

参见

- ◆ 参见 8.7 节。

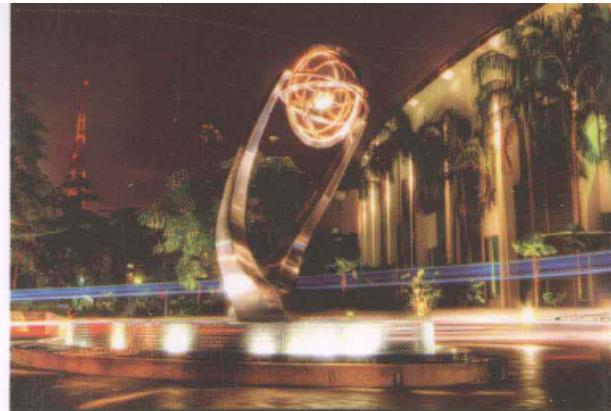
Java 7并发编程 实战手册

Java 7的发布为并发编程带来了一系列激动人心的新功能。这些新功能会尽可能地使应用程序的并行任务达到最佳性能。本书通过具体的基于任务的实例，详细地介绍了Java 7 并发API中的所有元素。有了本书，你将能够轻松自如地实现和利用这些新功能。

本书介绍了Java 7并发API中最重要、最实用的功能，你可以直接在应用程序中使用它们，以实现最佳性能。这些最核心的技术将帮助你应对基本的线程管理和任务管理，以及全新的Fork/Join框架、任务之间的同步机制、数据结构、定制等。

本书面向的读者：

如果你是一名Java程序员并且渴望了解Java 7最新的并发特性，同时希望能够进一步掌握多线程的知识，那么本书为你量身打造。



通过本书，你将学到：

- 在深入学习更高级的并发任务之前，先掌握好基本的线程管理和同步知识；
- 深入学习Java 7全新的并发特性，包括Phaser类和Fork/Join框架；
- 用真实的范例定制一些最实用的Java 并发API的类；
- 学习使用高级的Java工具来管理线程之间的同步；
- 学习并发应用程序中必须使用的数据结构，以避免出现数据冲突的问题；
- 在开发并发应用程序时，学会利用附录中提供的各种技巧来解决问题。



人民邮电出版社 - 信息技术分社
<http://weibo.com/ptpitbooks>

美术编辑：王建国

分类建议：计算机 / 程序设计 / Java
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-33529-6

定价：59.00 元