Broadview®
www.broadview.com.cn

· Java多线程编程实战系列 ·

本书用Java诠释多线程编程的“三十六计”——多线程设计模式。
每个设计模式的讲解都附有实战案例及源码解析，从理论到实战经验，
全面呈现常用多线程设计模式的来龙去脉。

Java 多线程编程实战指南

（设计模式篇）

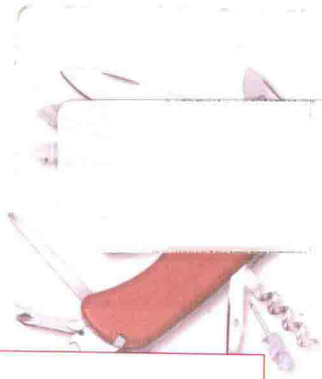
黄文海 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Java多线程编程实战指南

（设计模式篇）



技术内容用术语讲解是目前大部分书籍容易犯的毛病，导致技术类书籍成了催眠书，有幸和文海对技术进行交流，他擅长用生活中的语言和例子讲解复杂的技术，所以我认定，文海的书不会让读者失望。

北大青鸟中博软件学院校长　张帅

Java多线程是多任务Java应用的核心，本书总结了大量多线程使用场景，比较系统地分析了各种场景下的设计模式。作者将其多年的软件开发经验汇集成章，加以细致的讲解，并配合丰富的实例，让人印象深刻。相信读者定能从中受益良多。

华为技术有限公司系统工程师　陆多俊

本书源码下载地址：

http://github.com/Viscent/javamtp

http://www.broadview.com.cn/27006

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：付　睿
封面设计：李　玲

上架建议：程序设计>Java

ISBN 978-7-121-27006-2

定价：59.00元

Java 多线程编程实战指南

（设计模式篇）

黄文海 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着 CPU 多核时代的到来，多线程编程在充分利用计算资源、提高软件服务质量方面扮演了越来越重要的角色。而解决多线程编程中频繁出现的普遍问题可以借鉴设计模式所提供的现成解决方案。然而，多线程编程相关的设计模式书籍多采用 C++作为描述语言，且书中所举的例子多与应用开发人员的实际工作相去甚远。本书采用 Java（JDK1.6）语言和 UML 为描述语言，并结合作者多年工作经历的相关实战案例，介绍了多线程环境下常用设计模式的来龙去脉：各个设计模式是什么样的及其典型的实际应用场景、实际应用时需要注意的事项以及各个模式的可复用代码实现。

本书适合有一定 Java 多线程编程基础、经验的读者。

图书在版编目（CIP）数据

Java 多线程编程实战指南：设计模式篇/黄文海著. 北京：电子工业出版社，2015.10
（Java 多线程编程实战系列）
ISBN 978-7-121-27006-2

Ⅰ. ①J… Ⅱ. ①黄… Ⅲ. ①JAVA 语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆 CIP 数据核字(2015)第 195631 号

责任编辑：付　睿
印　　刷：中国电影出版社印刷厂
装　　订：中国电影出版社印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：17.5　字数：381 千字
版　　次：2015 年 10 月第 1 版
印　　次：2015 年 10 月第 1 次印刷
印　　数：3000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐序

欣闻文海兄弟的《Java 多线程编程实战指南》一书即将出版，心里感到非常激动和兴奋。与文海相识于 2014 年，某一天 InfoQ 中文站的运营编辑给我转发了一封读者投稿的邮件，标题是《Java 多线程编程模式实战指南之 Active Object 模式》。读完了稿件后立刻决定发布到 InfoQ 中文站上，因为这篇文章无论从内容选取、技术方向还是文字水平都是我见过的上乘之作。文章发布后也收到了很多读者的反馈，而该文章的作者正是文海。此后，文海又在 InfoQ 中文站上连载了多篇关于 Java 多线程设计模式相关的文章，均获得了不错的读者评价。

本书正是文海多年来工作经验的总结之作。众所周知，目前 Java 并发领域的经典好书大部分都是外版作品。不过值得欣喜的是，近一两年来，也有一些不错的国内开发者开始编写这个领域的图书，口碑也相当不错。文海的这部著作针对 Java 并发编程但又不局限于这个领域，它将 Java 多线程编程与设计模式这两大主题有机地结合到了一起。实际上，目前市场上虽然既有关于 Java 多线程编程的图书，也有关于设计模式的图书，但这两类图书内容之间却难以产生交集。介绍 Java 多线程的图书会专门讲解多线程编程的方方面面，而介绍设计模式的图书一般又会以经典的 23 种设计模式为蓝本，同时辅以一些简单的代码示例进行解读，难以让读者真正领会设计模式在实际开发中所起的作用。这本《Java 多线程编程实战指南》正是这两个领域的集大成者，它不仅深入透彻地分析了 Java 多线程编程的方方面面，还将其与设计模式有机地结合到了一起，形成了主动对象模式、两阶段终止模式、生产者/消费者模式、流水线模式、线程池模式等对实际项目开发会起到积极指导作用的诸多模式。可以这么说，本书不仅会向大家介绍 Java 多线程开发的难点与重点，还会探讨在某些场景下该使用哪种模式，这样做会给项目带来什么好处。从这个意义上来说，本书是 Java 多线程开发与设计模式理论的集大成者，相信会给广大的 Java 开发者带来切实的帮助。

目前已经是多核普及的时代，程序员也一定要编写面向多核的代码。虽然传统的 SSH（特指 Struts+Spring+Hibernate）依然还在发挥着重要的作用，但不得不说的是，作为一名有追求的

Java 开发者，眼光不应该局限于此。每一名有理想的 Java 开发者都应该系统学习有关多线程编程的知识，这不仅涉及程序语言与库的学习，还需要了解现代硬件体系架构（如 CPU、缓存、内存等），同时辅以恰当的设计模式，这样才能在未来游刃有余、得心应手。

虽然本人已经出版过多本技术图书，但为别人的书写序还是第一次。因此，在写这篇序之前我通读了该书的全部章节。事实也印证了之前的猜想，文海的这本书绝对是他本人的心血结晶之作，诸多的实际经验相信会给你带来不一样的感受。诚然，目前 Java 开发相关的技术图书已然汗牛充栋，但我相信，这本《Java 多线程编程实战指南》应该是每一个对代码有追求、对模式有见地的读者书架上不可或缺的一本书。

InfoQ 中文站 Java 主编：张龙

2015 年 9 月 14 日于北京

前言

随着现代 CPU 的生产工艺从提升 CPU 主频频率转向多核化，即在一块芯片上集成多个 CPU 内核（Core），以往那种靠 CPU 自身处理能力的提升所带来的软件计算性能提升的“免费午餐”不复存在。在此背景下，多线程编程在充分利用计算资源、提高软件服务质量方面扮演了越来越重要的角色。然而，多线程编程并非一个简单地使用多个线程进行编程的数量问题，其又有自身的问题。好比俗话说“一个和尚打水喝，两个和尚挑水喝，三个和尚没水喝”，简单地使用多个线程进行编程可能导致更加糟糕的计算效率。

设计模式相当于软件开发领域的“三十六计”，它为特定背景下反复出现的问题提供了一般性解决方案。多线程相关的设计模式为我们恰当地使用多线程进行编程并达到提升软件服务质量这一目的提供了指引和参考。当然，设计模式不是菜谱。即便是菜谱，我们也不能指望照着菜谱做就能做出一道美味可口的菜肴，但我们又不能因此而否认菜谱存在的价值。

可惜的是，国外与多线程编程相关的设计模式书籍多数采用 C++作为描述语言，且书中所举的例子又多与应用开发人员的实际工作经历相去甚远。本书作为国内第一本多线程编程相关设计模式的原创书籍，希望能够为 Java 开发者普及多线程相关的设计模式开一个头。

本书采用 Java（JDK1.6）语言和 UML（Unified Modeling Language）为描述语言，并结合作者多年工作经历的相关实战案例，介绍了多线程环境下常用设计模式的来龙去脉：各个设计模式是什么样的及其典型的实际应用场景、实际应用时需要注意的相关事项以及各个模式的可复用代码实现。

本书第 1 章对多线程编程基础进行了回顾，虽然该章讲的是基础，但重点仍然是强调“实战”。所谓“温故而知新”，有一定多线程编程基础、经验的读者也不妨快速阅读一下本章，说不定有新的收获。

本书第 3 章到第 14 章逐一详细讲解了多线程编程相关的 12 个常用设计模式。针对每个设计

模式，相应章节会从以下几个方面进行详细讲解。

模式简介。这部分简要介绍了相应设计模式的由来及核心思想，以便读者能够快速地对相应设计模式有个初步认识。

模式的架构。这部分会从静态（类及类与类之间的结构关系）和动态（类与类之间的交互）两个角度对相应设计模式进行详细讲解。模式架构分别使用 UML 类图（Class Diagram）和序列图（Sequence Diagram）对模式的静态和动态两个方面进行描述。

实战案例解析。在相应设计模式架构的基础上，本部分会给出相关的实战案例并对其进行解析。不同于教科书式的范例，实战案例强调的是“实战”这一背景。因此实战案例解析中，我们会先提出实际案例中我们面临的实际问题，并在此基础上结合相应设计模式讲解相应设计模式是如何解决这些问题的。实战案例解析中我们会给出相关的 Java 代码，并讲解这些代码与相应设计模式的架构间的对应关系，以便读者进一步理解相应设计模式。为了便于读者进行实验，本书给出的实战案例代码都力求做到可运行。实战案例解析有助于读者进一步理解相应的设计模式，并体验相应设计模式的应用场景。建议读者在阅读这部分时先关注重点，即实战案例中我们要解决哪些问题，相应设计模式又是如何解决这些问题的，以及实战案例的代码与相应设计模式的架构间的对应关系。而代码中其与设计模式非强相关的细节则可以稍后关注。

模式的评价与实现考量。这部分会对相应设计模式在实现和应用过程中需要注意的一些事项、问题进行讲解，并讨论应用相应设计模式所带来的好处及缺点。该节也会讨论相应设计模式的典型应用场景。

可复用实现代码。这部分给出相应设计模式的可复用实现代码。编写设计模式的可复用代码有助于读者进一步理解相应设计模式及其在实现和应用过程中需要注意的相关事项和问题，也便于读者在实际工作中应用相应设计模式。

Java 标准库实例。考虑到 Java 标准库的 API 设计过程中已经应用了许多设计模式，本书尽可能地给出相应设计模式在 Java API 中的应用情况。

相关模式。设计模式不是孤立存在的，一个具体的设计模式往往和其他设计模式之间存在某些联系。这部分会描述相应设计模式与其他设计模式之间存在的关系。这当中可能涉及 GOF 的设计模式，这类设计模式并不在本书的讨论范围之内。有需要的读者，请自行参考相关书籍。

本书的源码可以从 http://github.com/Viscent/javamtp 下载或博文视点官网 http://www.broadview.com.cn 相关图书页面下载。

目录

第1章

Java多线程编程实战基础

温故而知新

——《论语·为政》

有一定的多线程编程基础和工作经验的读者，也不妨继续往下看，看后或许会有新的发现。这一章的内容并非纯粹的理论“基础”，它更加强调“实战”。

1.1 无处不在的线程

进程（Process）代表运行中的程序。一个运行的Java程序就是一个进程。从操作系统的角度来看，线程（Thread）是进程中可独立执行的子任务。一个进程可以包含多个线程，同一个进程中的线程共享该进程所申请到的资源，如内存空间和文件句柄等。从JVM的角度来看，线程是进程中的一个组件（Component），它可以看作执行Java代码的最小单位。Java程序中任何一段代码总是执行在某个确定的线程中的。JVM启动的时候会创建一个main线程，该线程负责执行Java程序的入口方法（main方法）。如清单1-1所示的代码展示了Java程序中的代码总是由某个确定的线程运行的。

清单1-1. Java代码的执行线程

```
public class JavaThreadAnywhere {

    public static void main(String[] args) {
        System.out.println("The main method was executed by thread:"
            + Thread.currentThread().getName());
        Helper helper = new Helper("Java Thread Anywhere");
        helper.run();
    }

    static class Helper implements Runnable {
        private final String message;
```

```
        public Helper(String message) {
            this.message = message;
        }

        private void doSomething(String message) {
            System.out.println("The doSomething method was executed by thread:"
                + Thread.currentThread().getName());
            System.out.println("Do something with " + message);
        }

        @Override
        public void run() {
            doSomething(message);

        }
    }

}
```

如清单 1-1 所示的程序运行时输出如下：

```
The main method was executed by thread:main
The doSomething method was executed by thread:main
Do something with Java Thread Anywhere
```

从上面的输出可以看出，类 JavaThreadAnywhere 的 main 方法以及类 Helper 的 doSomething 方法都是由 main 方法负责执行的。

在多线程编程中，弄清楚一段代码具体是由哪个（或者哪种）线程去负责执行的这点很重要，这关系到性能问题、线程安全问题等。本书的后续章节会体现这点。

Java 中的线程可以分为守护线程（Daemon Thread）和用户线程（User Thread）。用户线程会阻止 JVM 的正常停止[1]，即 JVM 正常停止前应用程序中的所有用户线程必须先停止完毕；否则 JVM 无法停止。而守护线程则不会影响 JVM 的正常停止，即应用程序中有守护线程在运行也不影响 JVM 的正常停止。因此，守护线程通常用于执行一些重要性不是很高的任务，例如用于监视其他线程的运行情况。

1.2 线程的创建与运行

在 Java 语言中，一个线程就是一个 java.lang.Thread 类的实例。因此，在 Java 语言中创建一个线程就是创建一个 Thread 类的实例，当然这离不开内存的分配。创建一个 Thread 实例与创建其他类的实例所不同的是，JVM 会为一个 Thread 实例分配两个调用栈（Call Stack）所需的内存空间。这两个调用栈一个用于跟踪 Java 代码间的调用关系，另一个用于跟踪 Java

1 即通过 System.exit 调用停止 JVM，而非强行停止 JVM（如在 Linux 系统下使用 kill 命令停止 Java 进程）。

代码对本地代码（即 Native 代码，通常是 C 代码）的调用关系。

一个 Thread 实例通常对应两个线程。一个是 JVM 中的线程（或称之为 Java 线程），另一个是与 JVM 中的线程相对应的依赖于 JVM 宿主机[2]操作系统的本地（Native）线程。启动一个 Java 线程只需要调用相应 Thread 实例的 start 方法即可。线程启动后，当相应的线程被 JVM 的线程调度器调度到运行时，相应 Thread 实例的 run 方法会被 JVM 调用，如清单 1-2 所示。

清单 1-2. Java 线程的创建与运行

```
public class JavaThreadCreationAndRun {

    public static void main(String[] args) {
        System.out.println("The main method was executed by thread:"
           + Thread.currentThread().getName());
        Helper helper = new Helper("Java Thread Anywhere");

        //创建一个线程
        Thread thread = new Thread(helper);

        //设置线程名
        thread.setName("A-Worker-Thread");

        //启动线程
        thread.start();
    }

    static class Helper implements Runnable {
        private final String message;

        public Helper(String message) {
            this.message = message;
        }

        private void doSomething(String message) {
            System.out.println("The doSomething method was executed by thread:"
               + Thread.currentThread().getName());
            System.out.println("Do something with " + message);
        }

        @Override
        public void run() {
            doSomething(message);
                                \
        }
    }
}
```

清单 1-2 中，我们通过直接 new 一个 Thread 类实例来创建一个线程。Thread 类的其中一个构造器支持传入一个 java.lang.Runnable 接口实例，当相应线程启动时该实例的 run 方法会被 JVM 调用。

2 即运行 JVM 的主机。

如清单 1-2 所示的程序运行时输出如下：

```
The main method was executed by thread:main
The doSomething method was executed by thread:A-Worker-Thread
Do something with Java Thread Anywhere
```

与清单 1-1 所示的代码相比，同样的类 Helper 的同一个方法 doSomething 此时是由名为 A-Worker-Thread 的线程而非 main 线程负责执行。这是因为清单 1-1 中，类 Helper 的 run 方法由 main 线程所执行的 main 方法直接调用，而清单 1-2 中类 Helper 的 run 方法我们并没有在代码中直接对其进行调用，而是由 JVM 通过其创建的线程（线程名为 A-Worker-Thread）进行调用。

清单 1-2 中，对线程对象的 start 方法的调用（thread.start()）这段代码是运行在 main 方法中的，而 main 方法是由 main 线程负责执行的。因此，这里我们所创建的线程 thread 就可以看成是 main 线程的一个子线程，而 main 线程就是该线程的父线程。

Java 语言中，子线程是否是一个守护线程取决于其父线程：默认情况下父线程是守护线程则子线程也是守护线程，父线程是用户线程则子线程也是用户线程。当然，父线程在创建子线程后，启动子线程之前可以调用 Thread 实例的 setDaemon 方法来修改线程的这一属性。

Thread 类自身是一个实现 java.lang.Runnable 接口的对象，我们也可以通过定义一个 Thread 类的子类来创建线程，自定义的线程类要覆盖其父类的 run 方法，如清单 1-3 所示。

清单 1-3. 以创建 Thread 子类的方式创建线程

```
public class ThreadCreationViaSubclass {

    public static void main(String[] args) {
        Thread thread = new CustomThread();
        thread.start();

    }

    static class CustomThread extends Thread {

        @Override
        public void run() {
            System.out.println("Running...");
        }

    }

}
```

1.3 线程的状态与上下文切换

Java 语言中，一个线程从其创建、启动到其运行结束的整个生命周期可能经历若干个状态，如图 1-1 所示。

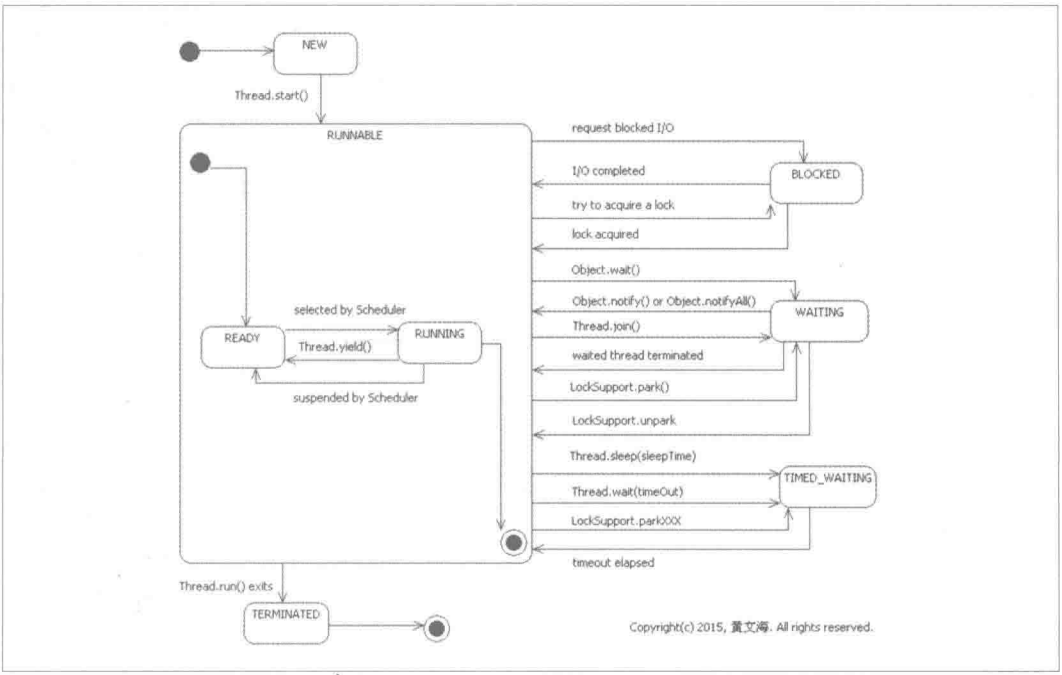


图 1-1. Java 线程的状态

Java 线程的状态可以通过调用相应 Thread 实例的 getState 方法获取。该方法的返回值类型 Thread.State 是一个枚举类型（Enum）。Thread.State 所定义的线程状态包括以下几种。

- **NEW**：一个刚创建而未启动的线程处于该状态。由于一个线程实例只能够被启动一次，因此一个线程只可能有一次处于该状态。
- **RUNNABLE**：该状态可以看成是一个复合的状态。它包括两个子状态：READY 和 RUNNING。前者表示处于该状态的线程可以被 JVM 的线程调度器（Scheduler）进行调度而使之处于 RUNNING 状态。后者表示处于该状态的线程正在运行，即相应线程对象的 run 方法中的代码所对应的指令正在由 CPU 执行。当 Thread 实例的 yield 方法被调用时或者由于线程调度器的原因，相应线程的状态会由 RUNNING 转换为 READY。

- **BLOCKED**：一个线程发起一个阻塞式 I/O（Blocking I/O）操作[3]后，或者试图去获得一个由其他线程持有的锁时，相应的线程会处于该状态。处于该状态的线程并不会占用 CPU 资源。当相应的 I/O 操作完成后，或者相应的锁被其他线程释放后，该线程的状态又可以转换为 RUNNABLE。

- **WAITING**：一个线程执行了某些方法调用之后就会处于这种无限等待其他线程执行特定操作的状态。这些方法包括：Object.wait()、Thread.join()和 LockSupport.park()。能够使相应线程从 WAITING 转换到 RUNNABLE 的相应方法包括：Object.notify()、Object.notifyAll()和 LockSupport.unpark(thread)。

- **TIMED_WAITING**：该状态和 WAITING 类似，差别在于处于该状态的线程并非无限等待其他线程执行特定操作，而是处于带有时间限制的等待状态。当其他线程没有在指定时间内执行该线程所期望的特定操作时，该线程的状态自动转换为 RUNNABLE。

- **TERMINATED**：已经执行结束的线程处于该状态。由于一个线程实例只能够被启动一次，因此一个线程也只可能有一次处于该状态。Thread 实例的 run 方法正常返回或者由于抛出异常而提前终止都会导致相应线程处于该状态。

从上述描述可知，一个线程在其整个生命周期中，只可能一次处于 NEW 状态和 TERMINATED 状态。而一个线程的状态从 RUNNABLE 状态转换为 BLOCKED、WAITING 和 TIMED_WAITING 这几个状态中的任何一个状态都意味着上下文切换（Context Switch）的产生。

上下文切换类似于我们接听手机电话的场景。比如，当我们正在接听一个电话并与对方讨论某件事情的时候，这时突然有另外一个来电。通常这个时候我们会跟对方说："我先接个电话，你别挂断"，并记下与之的讨论进行到什么程度了。然后，接听新的来电并告诉对方稍后会回拨并将该来电挂断。接着，我们又继续先前的讨论。如果在接听新来电之前，我们没有特意记下当前的讨论进展到什么程度，等我们接听新的来电后再回过头继续讨论时可能得问对方"刚才我们讲到哪里了"这样的问题。

多线程环境中，当一个线程的状态由 RUNNABLE 转换为非 RUNNABLE（BLOCKED、WAITING 或者 TIMED_WAITING）时，相应线程的上下文信息（即所谓的 Context，包括 CPU 的寄存器和程序计数器在某一时间点的内容等）需要被保存，以便相应线程稍后再次进入 RUNNABLE 状态时能够在之前的执行进度的基础上继续前进。而**一个线程的状态由非 RUNNABLE 状态进入 RUNNABLE 状态时可能涉及恢复之前保存的线程上下文信息并在此基础上前进。**这个对线程的上下文信息进行保存和恢复的过程就被称为上下文切换。

3 如文件读写和阻塞式 Socket 读写。

上下文切换会带来额外的开销，这包括保存和恢复线程上下文信息的开销、对线程进行调度的 CPU 时间开销以及 CPU 缓存内容失效（即 CPU 的 L1 Cache、L2 Cache 等）的开销[4]。

在 Linux 平台下，我们可以使用 perf 命令来监视 Java 程序运行过程中的上下文切换情况。例如，我们可以使用 perf 命令来监视如清单 1-2 所示的程序的运行，相应的命令如下：

```
perf stat -e cpu-clock,task-clock,cs,cache-references,cache-misses java
JavaThreadCreationAndRun
```

上述命令中，参数 e 的值中的 cs 表示要监视被监视程序的上下文切换的数量。上述命令执行后输出的内容类似如下：

```
73.719681 cpu-clock (msec)
73.704906 task-clock (msec)       #    0.956 CPUs utilized
434 cs                            #    0.006 M/sec
2,361,905 cache-references        #   32.045 M/sec
419,134 cache-misses              #   17.746 % of all cache refs

0.077060925 seconds time elapsed
```

由此可见，如清单 1-2 所示的程序的这次运行一共产生了 434 次上下文切换。

Windows 平台下，我们可以使用 Windows 自带的工具 perfmon[5]来监视 Java 程序运行过程中的上下文切换情况。

1.4 线程的监视

一个真实的 Java 系统运行时往往有上百个线程在运行，如果没有相应的工具可以对这些线程进行监视，那么这些线程对于我们来说就成了黑盒。而我们在开发过程中进行代码调试、定位问题，甚至是定位线上环境（生产环境）中的问题时往往都需要将线程变为白盒，即我们要能够知道系统中特定时刻存在哪些线程、这些线程处于什么状态以及这些线程具体是在做什么事情这些信息。

JDK 自带的工具 jvisualvm[6]可以实现线程的监视，它适合于在开发和测试环境下监视 Java 系

4 线程所执行的代码从 CPU 缓存中访问其所需的变量值比从主内存(RAM)中访问相应变量的值要快得多，但是上下文切换会导致相关线程所访问的 CPU 缓存内容失效，这使得相关线程稍后被重新调度到运行时其不得不再次访问主内存中的变量以重新创建 CPU 缓存内容。

5 相应的可执行文件为：Windows 安装目录\System32\perfmon.exe。

6 以 Linux 版 JDK 为例，jvisualvm 工具相应的可执行文件为：JDK 主目录\bin\jvisualvm。

统中的线程情况。图 1-2 展示了使用 jvisualvm 监视一个运行的 Eclipse 实例中的线程情况，这包括有哪些线程，以及这些线程的状态和调用栈。

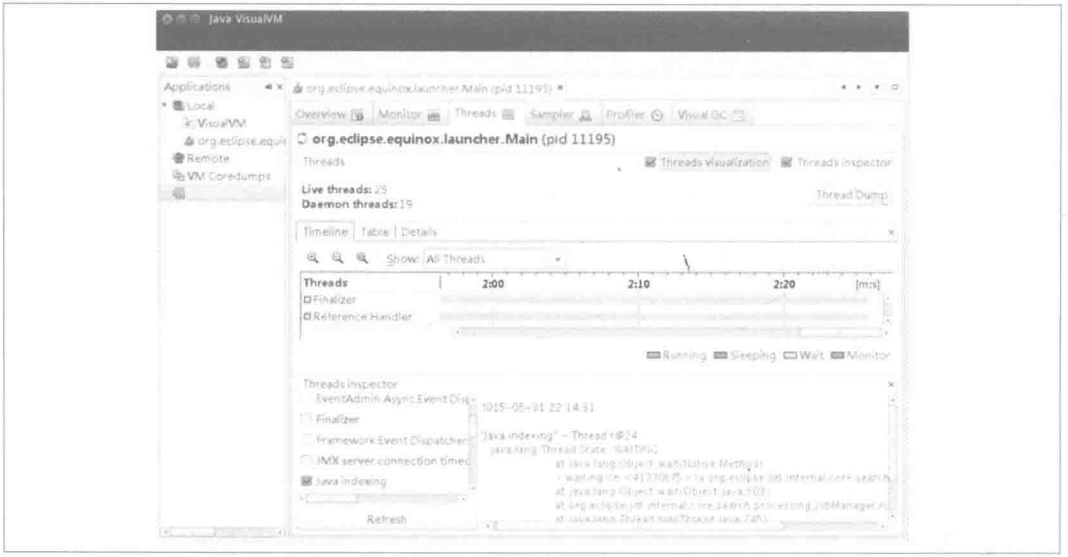


图 1-2. 使用 jvisualvm 监视 Java 线程

当然，如果是线上环境，我们可能不便使用 jvisualvm。此时可以使用 JDK 自带的另外一个工具 jstack[7]。jstack 是一个命令行工具，通过它可以获取指定 Java 进程的线程信息。例如，假设某个运行的 Eclipse 实例对应的 Java 进程 ID（PID）为 11195，那么通过以下命令我们可以获取该 Eclipse 实例中的线程情况。

```
~/apps/java/jdk1.6.0_45/bin/jstack 11195
```

上述命令的部分输出如图 1-3 所示。

在 Java 8 中，我们还可以使用 Java Mission Control（JMC）这个工具来监视 Java 线程，如图 1-4 所示。

7 以 Linux 版 JDK 为例，jstack 工具相应的可执行文件为：JDK 主目录\bin\jstack。

```
viscent@viscent:~/apps/java/jdk1.6.0_45/bin$ jstack 11195
2015-05-31 22:38:08
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.72-b04 mixed mode):

"Worker-13" prio=10 tid=0x00007f9eb40e8000 nid=0x32a7 in Object.wait() [0x00007f9ed043
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000dae68e48> (a org.eclipse.core.internal.jobs.WorkerPool
        at org.eclipse.core.internal.jobs.WorkerPool.sleep(WorkerPool.java:188)
        - locked <0x00000000dae68e48> (a org.eclipse.core.internal.jobs.WorkerPool)
        at org.eclipse.core.internal.jobs.WorkerPool.startJob(WorkerPool.java:220)
        at org.eclipse.core.internal.jobs.Worker.run(Worker.java:51)

"Worker-12" prio=10 tid=0x000000000110b000 nid=0x329a in Object.wait() [0x00007f9ea383
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000dae68e48> (a org.eclipse.core.internal.jobs.WorkerPool
        at org.eclipse.core.internal.jobs.WorkerPool.sleep(WorkerPool.java:188)
        - locked <0x00000000dae68e48> (a org.eclipse.core.internal.jobs.WorkerPool)
        at org.eclipse.core.internal.jobs.WorkerPool.startJob(WorkerPool.java:220)
        at org.eclipse.core.internal.jobs.Worker.run(Worker.java:51)

"pool-2-thread-1" prio=10 tid=0x00007f9f3dc29800 nid=0x3284 waiting on condition [0x00
   java.lang.Thread.State: TIMED_WAITING (parking)
```

图 1-3. 使用 jstack 监视 Java 线程

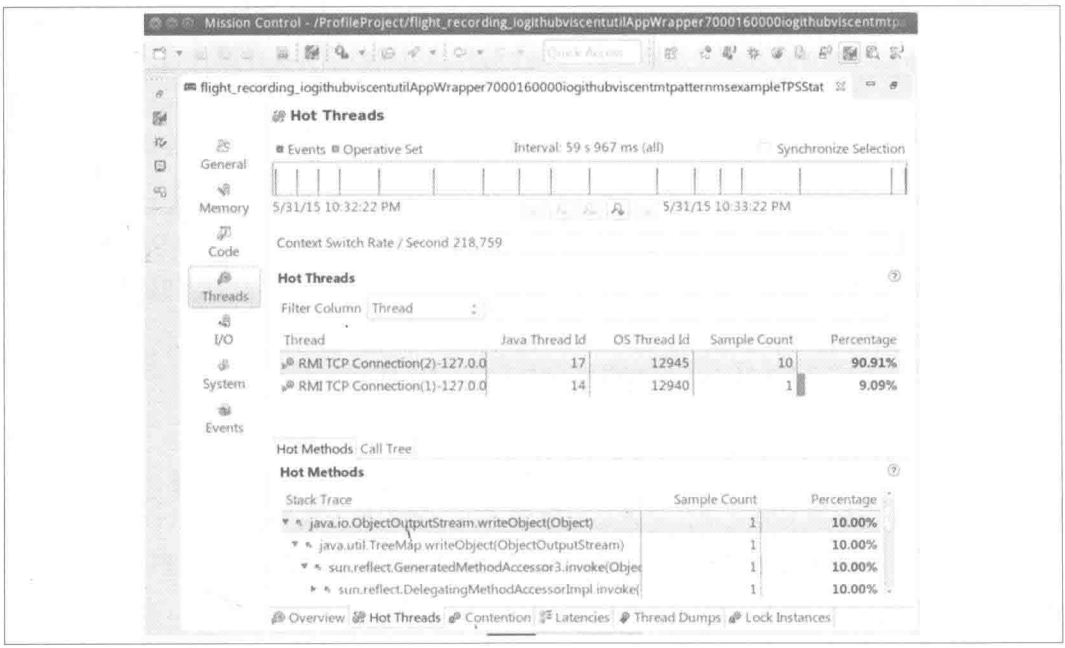


图 1-4. 使用 JMC 监视 Java 线程

1.5 原子性、内存可见性和重排序——重新认识 synchronized 和 volatile

原子（Atomic）操作指相应的操作是单一不可分割的操作。例如，对 int 型变量 count 执行 counter++的操作就不是原子操作。这是因为 counter++实际上可以分解为 3 个操作：1）读取变量 counter 的当前值，2）拿 counter 的当前值和 1 做加法运算，3）将 counter 的当前值增加 1 后的值赋值给 counter 变量。

在多线程环境中，非原子操作可能会受其他线程的干扰。比如，上述例子如果没有对相应的代码进行同步（Synchronization）处理，则可能出现在执行第 2 个操作的时候 counter 的值已经被其他线程修改了，因此这一步的操作所使用的 counter 变量的“当前值”其实已经是过期的了。当然，synchronized 关键字可以帮助我们实现操作的原子性，以避免这种线程间的干扰情况。

synchronized 关键字可以实现操作的原子性，其本质是通过该关键字所包括的临界区（Critical Section）的排他性保证在任何一个时刻只有一个线程能够执行临界区中的代码，这使得临界区中的代码代表了一个原子操作。这一点，读者可能已经很清楚。但是，synchronized 关键字所起的另外一个作用——保证内存的可见性（Memory Visibility），也是值得我们回顾的。

CPU 在执行代码的时候，为了减少变量访问的时间消耗可能将代码中访问的变量的值缓存到该 CPU 的缓存区（如 L1 Cache、L2 Cache 等）中。因此相应代码再次访问某个变量时，相应的值可能是从 CPU 缓存区而不是主内存中读取的。同样地，代码对这些被缓存过的变量的值的修改也可能仅是被写入 CPU 缓存区，而没有被写回主内存。由于每个 CPU 都有自己的缓存区，因此一个 CPU 缓存区中的内容对于其他 CPU 而言是不可见的。这就导致了在其他 CPU 上运行的其他线程可能无法“看到”该线程对某个变量值所做的更改。这就是所谓的内存可见性。

synchronized 关键字的另外一个作用就是它保证了一个线程执行临界区中的代码时所修改的变量值对于稍后执行该临界区中的代码的线程来说是可见的。这对于保证多线程代码的正确性来说非常重要。

而 volatile 关键字也能够保证内存可见性。即一个线程对一个采用 volatile 关键字修饰的变量的值的更改对于其他访问该变量的线程而言总是可见的。也就是说，其他线程不会读到一个“过期”的变量值。因此，有人将 volatile 关键字与 synchronized 关键字所代表的内部锁做比较，将其称为轻量级的锁。这种称呼其实并不恰当，volatile 关键字只能保证内存可见性，它并不能像 synchronized 关键字所代表的内部锁那样能够保证操作的原子性。volatile 关键字

实现内存可见性的核心机制是当一个线程修改了一个 volatile 修饰的变量的值时，该值会被写入主内存（即 RAM）而不仅仅是当前线程所在的 CPU 的缓存区，而其他 CPU 的缓存区中存储的该变量的值也会因此而失效（从而得以更新为主内存中该变量的相应值）。这就保证了其他线程访问该 volatile 修饰的变量时总是可以获取该变量的最新值。

volatile 关键字的另外一个作用是它禁止了指令重排序（Re-order）[8]。编译器和 CPU 为了提高指令的执行效率可能会进行指令重排序，这使得代码的实际执行方式可能不是按照我们所认为的方式进行的。例如下面的实例变量初始化语句：

```
private SomeClass someObject = new SomeClass();
```

上述语句所做的事情非常简单：1）创建类 SomeClass 的实例，2）将类 SomeClass 的实例的引用赋值给变量 someObject。但是由于指令重排序的作用，这段代码的实际执行顺序可能是：1）分配一段用于存储 SomeClass 实例的内存空间，2）将对该内存空间的引用赋值给变量 someObject，3）创建类 SomeClass 的实例。因此，当其他线程访问 someObject 变量的值时，其得到的仅是指向一段存储 SomeClass 实例的内存空间的引用而已，而该内存空间相应的 SomeClass 实例的初始化可能尚未完成，这就可能导致一些意想不到的结果。而禁止指令重排序则可以使得上述代码按照我们所期望的顺序（正如代码所表达的顺序）来执行。

禁止指令重排序虽然导致编译器和 CPU 无法对一些指令进行可能的优化，但是它某种程度上让代码的执行看起来更符合我们的期望。

本书涉及的代码也有不少地方使用了 volatile 关键字。读者需要注意这个关键字对多线程代码的正确性所起的作用。

与 synchronized 相比，前者既能保证操作的原子性，又能保证内存可见性。而 volatile 仅能保证内存可见性。但是，前者会导致上下文切换，而后者不会。

1.6 线程的优势和风险

多线程编程具有以下优势。

- **提高系统的吞吐率（Throughput）**。多线程编程使得一个进程中可以有多个并发（Concurrent，即同时进行的）的操作。例如，当一个线程因为 I/O 操作而处于等待时，其他线程仍然可以执行其操作。
- **提高响应性（Responsiveness）**。使用多线程编程的情况下，对于 GUI 软件（如桌面

8 有关指令重排序的进一步内容，请参考 Java 内存模型（JMM，Java Memory Model）的相关资料。

应用程序）而言，一个慢的操作（比如从服务器上下载一个大的文件）并不会导致软件的界面出现被“冻住”的现象而无法响应用户的其他操作；对于 Web 应用程序而言，一个请求的处理慢了并不会影响其他请求的处理。

- **充分利用多核（Multicore）CPU 资源**。如今多核 CPU 的设备越来越普及，就算是手机这样的消费类设备也普遍使用多核 CPU。实施恰当的多线程编程有助于我们充分利用设备的多核 CPU 资源，从而避免了资源浪费。

- **最小化对系统资源的使用**。一个进程中的多个线程可以共享其所在进程所申请的资源（如内存空间），因此使用多个线程相比于使用多个进程进行编程来说，节约了对系统资源的使用。

- **简化程序的结构**。线程可以简化复杂应用程序的结构。

多线程编程也有自身的问题与风险，包括：

- **线程安全（Thread Safe）问题**。多个线程共享数据的时候，如果没有采取相应的并发访问控制措施，那么就可能产生数据一致性问题，如读取脏数据（过期的数据）、丢失更新（某些线程所做的更新被其他线程所做的更新覆盖）等。使用锁（包括 synchronized 关键字和 ReentrantLock 等）能够保证线程安全是大家所熟知的，本书后续章节则会提供一些不借助锁而实现线程安全的方案。

- **线程的生命特征（Thread Liveness）问题**。一个线程从其创建到运行结束的整个生命周期会经历若干个状态。从单个线程的角度来看，RUNNABLE 状态是我们所期望的状态。但实际上，代码编写不适当可能导致某些线程一直处于等待其他线程释放锁的状态（BLOCKED 状态），即产生了死锁（Dead Lock）。例如，线程 T1 拥有锁 L1，并试图去获得锁 L2，而此时线程 T2 拥有锁 T2 而试图去获得锁 L1，这就导致线程 T1 和 T2 一直处于等待对方释放锁而一直又得不到锁的状态。当然，一直忙碌的线程也可能会出现问题，它可能面临活锁（Live Lock）问题，即一个线程一直在尝试某个操作但就是无法进展，这就好比室内窗户上的苍蝇向着阳光飞却一直也没能飞出去一样。另外，线程是一种稀缺的计算资源，一个系统所拥有的 CPU 数量相比于该系统中存在的线程数量而言总是少之又少的。某些情况可能出现线程饥饿（Starvation）的问题，即某些线程永远无法获取 CPU 执行的机会而永远处于 RUNNABLE 状态的 READY 子状态。

- **上下文切换**。由于 CPU 资源的稀缺性，上下文切换可以看作是多线程编程的必然副产物，它增加了系统的消耗，不利于系统的吞吐率。

- **可靠性**。多线程编程一方面可以有利于可靠性，例如某个线程意外提前终止了，但这并不影响其他线程继续其处理。另一方面，线程是进程的一个组件，它总是存在于特定的进程中的，如果这个进程由于某种原因意外提前终止了，比如某个 Java 进程由于

内存泄漏导致 JVM 奔溃而意外终止了，那么该进程中所有的线程也就随之而无法继续运行。因此，从提高软件的可靠性的角度来看，某些情况下可能要考虑多进程多线程的编程方式[9]，而非简单的单进程多线程方式。

1.7 多线程编程常用术语

本书常用的多线程编程方面的术语如表 1-1 所示。

表 1-1. 多线程编程常用术语

术　语	释　义	备　注
任务（Task）	把线程比作公司员工的话，那么任务就可以被看作员工所要完成的工作内容。任务与线程并非一一对应的，通常一个线程可以用来执行多个任务。任务是一个相对的概念。一个文件可以被看作一个任务，一个文件中的多个记录可以被看作一个任务，多个文件也可以被看作一个任务	
并发（Concurrent）	表示多个任务在同一时间段内被执行。这些任务并不是顺序执行的，而往往是以交替的方式被执行	
并行（Parallel）	表示多个任务在同一时刻被执行	
客户端线程（Client Thread）	从面向对象编程的角度来看，一个类总是对外提供某些服务（这也是这个类存在的意义）。其他类是通过调用该类的相应方法来使用这些服务的。因此，一个类的方法的调用方代码就被称为该类的客户端代码。相应地，执行客户端代码的线程就被称为客户端线程。因此，客户端线程也是一个相对的概念，某个类的客户端线程随着执行该方法调用方代码的线程的变化而变化	清单 1-2 中，类 Helper 的 doSomething 方法的客户端线程就是名为 A-Worker-Thread 的线程
工作者线程（Worker Thread）	工作者线程是相对于客户端线程而言的。它表示客户端线程之外的用于特定用途的其他线程	清单 1-4 展示了一个工作者线程的示例代码

9 例如，一个系统被分解为多个模块，每个模块是一个 Java 进程（程序）。各个模块间采用网络进行通信。

续表

术　语	释　义	备　注
上下文切换（Context Switch）	多线程环境中，当一个线程的状态由RUNNABLE转换为非RUNNABLE（BLOCKED、WAITING或者TIMED_WAITING）时，相应线程的上下文信息（即所谓的Context，包括CPU的寄存器和程序计数器在某一时间点的内容等）需要被保存以便相应线程稍后再次进入。RUNNABLE状态时能够在之前的执行进度的基础上继续前进。而一个线程的状态由非RUNNABLE状态进入RUNNABLE状态时也就可能涉及恢复之前保存的线程上下文信息并在此基础上前进。这个对线程的上下文信息进行保存和恢复的过程就被称为上下文切换	
显式锁（Explicit Lock）	指在Java代码中可以使用和控制的锁，即不是编译器和CPU内部使用的锁。包括synchronized关键字和java.util.concurrent.locks.Lock接口的所有实现类	
线程安全（Thread Safe）	一段操纵共享数据的代码能够保证在同一时间内被多个线程执行而仍然保持其正确性的，就被称为是线程安全的	清单1-5展示了一个非线程安全的计数器类。该类的increment方法被多个线程同时调用的话可能导致计数器的计数结果不正确。而清单1-6展示了线程安全的计数器类，该类可以由多个线程同时使用而不影响其计数的正确性

清单1-4. 工作者线程示例代码

```
public class WorkerThread {

    public static void main(String[] args) {
        Helper helper = new Helper();
        helper.init();

        // 此处，helper的客户端线程为main线程
        helper.submit("Something...");
```

```
    }

    static class Helper {
        private final BlockingQueue<String> workQueue = new
                ArrayBlockingQueue<String>(100);

        // 用于处理队列 workQueue 中的任务的工作者线程
        private final Thread workerThread = new Thread() {

            @Override
            public void run() {
                String task = null;
                while (true) {
                    try {
                        task = workQueue.take();
                    } catch (InterruptedException e) {
                        break;
                    }
                    System.out.println(doProcess(task));

                }
            }

        };

        public void init() {
            workerThread.start();
        }

        protected String doProcess(String task) {
            return task + "->processed.";
        }

        public void submit(String task) {
            try {
                workQueue.put(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                ;
            }
        }

    }

}
```

清单 1-5. 非线程安全的计数器

```
/**
 * 非线程安全的计数器。
 *
 * @author Viscent Huang
 *
 */
public class NonThreadSafeCounter {
    private int counter = 0;

    public void increment() {
        counter++;
```

```
    }

    public int get() {
        return counter;
    }
}
```

清单 1-6. 线程安全的计数器

```
/**
 * 线程安全的计数器。
 *
 * @author Viscent Huang
 *
 */
public class ThreadSafeCounter {
    private int counter = 0;

    public void increment() {
        synchronized (this) {
            counter++;
        }
    }

    public int get() {
        synchronized (this) {
            return counter;
        }

    }
}
```

第2章
设计模式简介

Nothing in the world is single,
世上哪有什么孤零零？
All things by a law divine
万物由于自然律，
In one another's being mingle…
都必融汇于一种精神。
——《Love's Philosophy》雪莱，1819年

2.1 设计模式及其作用

设计模式（Design Pattern）是软件设计中给定背景（Context）下普遍存在的问题一般性可复用的解决方案[1]。这个定义可能有点抽象，但是我们可以先了解一下设计模式的历史由来以对其有个感性的认识。

一般认为模式（Pattern）起源于 Christopher Alexander 所提出的建筑上的概念，后来有人开始将其借用到软件行业。这其中最为人所熟知的是 Erich Gamma 等 4 人（这 4 人亦被称为 Gang of Four，GOF）于 1994 年所出版的经典之作《设计模式：可复用的面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*）。实际上，国人也不必舍近求远，我们可以从大家耳熟能详的三十六计中去直接感受一下设计模式是个什么东西。

1 引自维基百科："A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design"。

中国古代兵法著作《三十六计》中许多计策都是大家耳熟能详的，例如调虎离山计、离间计、反间计等。《三十六计》中的每个计策都可以看成一个关于计策的模式。例如，其中的借刀杀人计的描述如下：

敌已明，友未定，引友杀敌。不自出力，以《损》推演。

意思是说，目前形势谁是敌人已经明朗而谁是友人尚未明确。在此情形下，利用“友人”去消灭敌人，自己则可以不用亲自动手，而能够坐收渔人之利。这则计策是根据《易经》中的损卦推演出来的。

借刀杀人计的描述就可以反映出设计模式的几个要素。首先是所谓给定背景，它指明了什么情况下可以运用某个计策（模式）。借刀杀人计中的“敌已明，友未定”（谁是敌人已明朗，谁是友人尚未明确）就说明了该计策（模式）的运用背景。其次是解决方案，当然解决方案的背后是问题。对于借刀杀人计而言，要解决的问题很明显是消灭敌人。而相应的解决方案是借他人（即“友人”）之手消灭敌人而不必自己亲自动手：由于敌人已明确，而“友人”尚未明确，借“友人”之手去消灭敌人，则这两方无论谁受打击对己方而言都是有利而无害的。接着是隐藏在计策（模式）背后的思想：《三十六计》中的计策源自《易经》中的卦象（思想）。例如，借刀杀人计就是根据《易经》中的第 41 卦（损卦）推演出来的。

另外，计策和计策之间往往不是孤立的，而是需要相互配合使用的。例如，离间计（其核心是破坏敌人内部关系，使他们疏远）和反间计（其核心是发现敌方的间谍后不揭穿而是利用其传递假情报）就经常配合着使用。《三国演义》中的“蒋干盗书”的故事就是综合使用了反间计和离间计以削弱曹操的势力：赤壁大战前夕，曹操手下的谋士蒋干要去东吴劝降周瑜。周瑜则故意在书房中让蒋干盗得一封捏造的由曹操水军将领蔡瑁、张允写给周瑜的降书，以利用其携带假情报给曹操，离间曹操与蔡瑁、张允之间的关系。曹操果真上了当，斩了精通水战的蔡瑁、张允。另外，三十六计专门有一计叫连环计，其核心就是环环相扣地使用其他各个计策。连环计更加明确地表明各个计策之间不是孤立的，在特定情况下它们是相互联系的。

此时回到正题，就不难理解设计模式了：我们可以将设计模式看作是软件设计领域的三十六计。

正如三十六计中的各个计策之间并非孤立，而是可以相互联系的，设计模式与设计模式之间也不是孤立的。在解决实际问题的过程中，我们往往需要综合使用多个设计模式，而不是单靠某个设计模式，这点从本书所提供的各个实战案例中可见一斑。

隐藏在三十六计背后的是《易经》中的思想，而隐藏在设计模式背后的是面向对象设计的思想[2]。

设计模式是一种可复用的设计方案，它并不是可以直接使用的代码。因此，这就涉及设计模式的实现问题，即如何将设计方案转化为可执行的代码。当然，这也就涉及编程语言，具体来说是面向对象的编程语言。因此，一方面设计模式依赖于编程语言，因为最终我们要将设计模式转化为具体的可执行的代码。另一方面，设计模式具有语言独立性。一个设计模式可以使用 Java 语言实现，也能使用 C++、C#等其他面向对象编程语言实现。当然，在实现设计模式时我们也必须考虑所选用的编程语言自身的特点。由于本书选用 Java 语言作为实现语言，因此对设计进行描述时有时会直接从语言的层面出发。读者若需要以其他语言作为实现语言也不妨，只要在具体实现时充分考虑语言自身的特点即可。考虑到设计模式本身并非可直接使用的代码，而设计模式的实现又离不开代码，本书在介绍各个具体设计模式的时候会尽可能给出相应模式的可复用代码，以便读者理解和应用相应设计模式。

设计模式可以为我们解决设计问题提供可借鉴的成熟解决方案。设计模式是在广泛实践的基础上提炼出来的设计方案。这也就意味着，许多设计模式已经被广泛运用在许多软件的设计之中。例如，Java 标准库 API 的设计就运用了许多设计模式。本书也正是有鉴于此，在介绍每个具体的设计模式时，有专门的一节介绍 Java 标准库对相应设计模式的运用。

设计模式为软件开发人员之间阐述和沟通设计方案提供了共用的词汇。团队开发中，软件的设计必然涉及设计方案的讨论与沟通。一方面，负责设计的人员之间需要进行方案的沟通。另一方面，最终负责落实设计方案的人员（编码人员）也需要理解设计方案，这当然也涉及相应的阐述和沟通。而设计模式为开发人员之间阐述和沟通设计方案提供了一个统一的基础，即大家都能理解且理解一致的词汇。这种词汇在软件设计过程中所起的作用，正如其他专业术语如继承、多态等在我们工作过程中所起的作用一样。比如，团队成员在讨论一个关于如何扩展我们手头上没有源代码的 Java 类的问题时，有人提出“我们自己新建一个类，该类继承自 XX 类”，此时大家便都明白可以怎么做了。

设计模式可以作为描述软件架构的一种方式。设计模式构建了开发人员之间阐述和沟通设计方案的词汇，因此它自然可以用来描述软件的架构。有了设计模式，我们在描述系统架构的时候便可以使用“本模块运用了 XX 设计模式”之类的语句。

2 概括为 SOLID（Single Responsibility Principle 单一职责原则、Open-Closed Principle 开闭原则、Liskov Substitution Principle 里氏替换原则、Interface Segregation Principle 接口隔离原则以及 Dependency Inversion Principle 依赖倒置原则）原则。

2.2 多线程设计模式简介

随着 CPU 的生产工艺从提高 CPU 主频频率转向多核化[3],以往那种靠 CPU 主频频率提升所带来的软件性能提升的“免费午餐”[4]不复存在。这使得多线程编程在充分发挥系统的 CPU 资源以及提升软件性能方面起到了越来越重要的作用。然而，多线程编程本身又会引入开销及其他问题，如较之单线程顺序编程的复杂性、线程安全问题、死锁、活锁以及上下文切换开销等。多线程设计模式是多线程编程领域的设计模式，它可以帮助我们解决多线程编程中的许多问题。本书介绍的多线程设计模式在按其有助于解决的多线程编程相关的问题可粗略分类如下。

- **不使用锁的情况下保证线程安全**： Immutable Object（不可变对象）模式（第 3 章）、Thread Specific Storage（线程特有存储）模式（第 10 章）、Serial Thread Confinement（串行线程封闭）模式（第 11 章）。
- **优雅地停止线程**：Two-phase Termination（两阶段终止）模式（第 5 章）。
- **线程协作**：Guarded Suspension（保护性暂挂）模式（第 4 章）、Producer-Consumer（生产者/消费者）模式（第 7 章）。
- **提高并发性（Concurrency）、减少等待**：Promise（承诺）模式（第 6 章）、Active Object（主动对象）模式（第 8 章）、Pipeline（流水线）模式（第 13 章）。
- **提高响应性（Responsiveness）**：Master-Slave（主仆）模式（第 12 章）、Half-sync/Half-async（半同步/半异步）模式（第 14 章）。
- **减少资源消耗**：Thread Pool（线程池）模式（第 9 章）、Serial Thread Confinement（串行线程封闭）模式（第 11 章）。

读者现在不必急于理解上述分类的具体含义，可以在阅读完本书的相应章节后再回头来看这个分类。从上面的分类中我们可以看出，Serial Thread Confinement（串行线程封闭）模式（第 11 章）被划分到了两个类别中，这是因为该模式既能在不引入锁的情况下保证线程安全，也有助于减少上下文切换所带来的系统资源消耗。

---

3 即一块芯片上集成多个处理器。

4 参见《The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software》一文：http://www.gotw.ca/publications/concurrency-ddj.htm。

2.3 设计模式的描述

设计的模式通常都有形式化的格式来描述。但不管采用哪种格式，其表达的内容都脱离不了上述设计模式的几个要素。

描述设计模式的一种常见格式包含以下几个方面。

- **别名**（Alias）。该部分指出相应模式的其他名字。
- **背景**（Context）。该部分列出相应设计模式所要解决的问题所产生的背景，或者应用相应设计模式的情景。
- **问题**（Problem）。该部分指出相应设计模式所要解决的问题。
- **解决方案**（Solution）。该部分描述了设计模式对相应问题所提出的解决方案。
- **结果**（Consequences）。该部分对设计模式进行评价，主要描述了使用相应模式会带来哪些好处以及可能存在哪些弊端。
- **相关模式**（Related Patterns）。该部分描述相应模式与其他模式之间的关系。

本书第15章就是采用这种格式对本书所介绍的12个多线程方面的设计模式进行描述。不过，这一章采用这种格式进行描述的目的是方便读者在需要时进行快速参考。而为了方便讲解各个设计模式，本书在其他章中使用了以下格式进行描述。

- **模式简介**。该部分主要对相应模式的核心思想和本质进行描述，以便读者对其形成基本的认识。
- **模式架构**。该部分会描述相应模式涉及哪些类（参与者）以及这些类之间的关系，并在此基础上描述这些类之间是如何协作从而解决相应模式所要解决的问题。也就是说，该部分从静态（类及类与类之间的关系）和动态（类之间的协作）两个角度描述了解决方案，因此被称为模式架构。
- **实战案例解析**。该部分以笔者实际工作经历中应用相应模式所解决过的问题为例子，讲解了应用相应设计模式的典型场景（背景）以及相应设计模式是如何解决相关问题的。
- **模式的评价与考量**。该部分会对相应设计模式进行评价，分析其优势和弊端，并对其实现和运用过程中需要注意的一些事项、重要问题及解决方法进行描述。例如，第10章介绍的Thread Specific Storage（线程特有存储）模式可以在不引入锁的情况下实现线程安全。但是其在实际使用过程中却可能导致内存泄漏这样严重的问题。
- **可复用实现代码**。该部分会给出相应模式的可复用实现代码。编写模式的可复用代码可以帮助读者进一步理解相应的设计模式及其实现时需要注意的问题。另一方面，也便于

读者在实际工作中运用相应的设计模式。

- **Java 标准库实例**。该部分描述了 Java 标准库对相应设计模式的使用情况。Java 标准库已经实现了本书介绍的部分设计模式，如 Promise（承诺）模式（第 6 章）和 Thread Specific Storage（线程特有存储）模式（第 10 章）。该部分重点讲解的是 Java 标准库对相应设计模式的使用而不是实现情况。
- **相关模式**。该部分描述相应模式与其他模式之间的关系。

第3章

Immutable Object（不可变对象）模式

3.1 Immutable Object 模式简介

多线程共享变量的情况下，为了保证数据一致性，往往需要对这些变量的访问进行加锁。而锁本身又会带来一些问题和开销。Immutable Object 模式使得我们可以在不使用锁的情况下，既保证共享变量访问的线程安全，又能避免引入锁可能带来的问题和开销。

多线程环境中，一个对象常常会被多个线程共享。这种情况下，如果存在多个线程并发地修改该对象的状态或者一个线程访问该对象的状态而另外一个线程试图修改该对象的状态，我们不得不做一些同步访问控制以保证数据一致性。而这些同步访问控制，如显式锁（Explicit Lock）和 CAS（Compare and Swap）操作，会带来额外的开销和问题，如上下文切换、等待时间和 ABA 问题等。Immutable Object 模式的意图是通过使用对外可见的状态不可变的对象（即 Immutable Object），使得被共享对象“天生”具有线程安全性，而无须额外地同步访问控制。从而既保证了数据一致性，又避免了同步访问控制所产生的额外开销和问题，也简化了编程。

所谓状态不可变的对象，即对象一经创建，其对外可见的状态就保持不变，例如 Java 中的 String 和 Integer。这点固然容易理解，但还不足以指导我们在实际工作中运用 Immutable Object 模式。下面我们看一个典型应用场景，这不仅有助于我们理解它，也有助于在实际的环境中运用它。

一个车辆管理系统要对车辆的位置信息进行跟踪，我们可以对车辆的位置信息建立如清单 3-1 所示的模型。

清单 3-1. 状态可变的位置信息模型（非线程安全）

```
public class Location {

    private double x;
    private double y;

    public Location(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setXY(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

当系统接收到新的车辆坐标数据时，需要调用 Location 的 setXY 方法来更新位置信息。显然，清单 3-1 中的 setXY 是非线程安全的，因为对坐标数据 x 和 y 的写操作不是一个原子操作。setXY 被调用时，如果在 x 写入完毕，而 y 开始写之前有其他线程来读取位置信息，则该线程可能读到一个被追踪车辆根本不曾经过的位置。为了使 setXY 方法具备线程安全性，我们需要借助锁进行访问控制。虽然被追踪车辆的位置信息总是在变化，但是我们也可以将位置信息建模为状态不可变的对象，如清单 3-2 所示。

清单 3-2. 状态不可变的位置信息模型

```
public final class Location {
    public final double x;
    public final double y;

    public Location(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

使用状态不可变的位置信息模型时，如果车辆的位置发生变动，则更新车辆的位置信息是通过替换整个表示位置信息的对象（即 Location 实例）来实现的[1]，如清单 3-3 所示。

1 实际上 ConcurrentMap 内部的确涉及了锁的使用。这里只是为了展示车辆位置信息的改变是通过新建和替换整个 Location 实例来现实的。这个过程本身不涉及锁。

清单 3-3. 在使用不可变对象的情况下更新车辆的位置信息

```
public class VehicleTracker {

    private Map<String, Location> locMap = new ConcurrentHashMap<String, Location>();

    public void updateLocation(String vehicleId, Location newLocation) {
        locMap.put(vehicleId, newLocation);
    }

}
```

因此，所谓状态不可变的对象并非指被建模的现实世界实体的状态不可变，而是我们在建模的时候的一种决策：现实世界实体的状态总是在变化的，但我们可以用状态不可变的对象来对这些实体进行建模。

3.2 Immutable Object 模式的架构

Immutable Object 模式将现实世界中状态可变的实体建模为状态不可变对象，并通过创建不同的状态不可变的对象来反映实现世界实体的状态变更。

Immutable Object 模式的主要参与者有以下几种。其类图如图 3-1 所示。

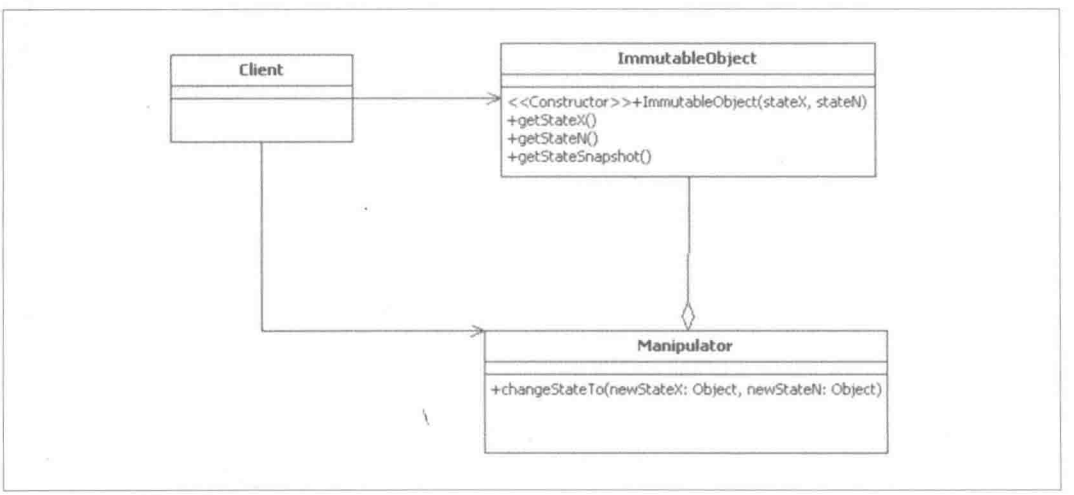


图 3-1. Immutable Object 模式的类图

- **ImmutableObject**：负责存储一组不可变状态。该参与者不对外暴露任何可以修改其状态的方法，其主要方法及职责如下。
  - **getStateX，getStateN**：这些 getter 方法返回其所属 ImmutableObject 实例所维护的

状态相关变量的值。这些变量在对象实例化时通过其构造器的参数获得值。

- **getStateSnapshot**：返回其所属 ImmutableObject 实例维护的一组状态的快照。

- **Manipulator**：负责维护 ImmutableObject 所建模的现实世界实体状态的变更。当相应的现实实体状态变更时，该参与者负责生成新的 ImmutableObject 的实例，以反映新的状态。

  - **changeStateTo**：根据新的状态值生成新的 ImmutableObject 的实例。

不可变对象的使用主要包括以下几种类型。

- **获取单个状态的值**：调用不可变对象的相关 getter 方法即可实现。
- **获取一组状态的快照**：不可变对象可以提供一个 getter 方法，该方法需要对其返回值做防御性复制或者返回一个只读的对象，以避免其状态对外泄露而被改变。
- **生成新的不可变对象实例**：当被建模对象的状态发生变化的时候，创建新的不可变对象实例来反映这种变化。

Immutable Object 模式的典型交互场景的序列图如图 3-2 所示。

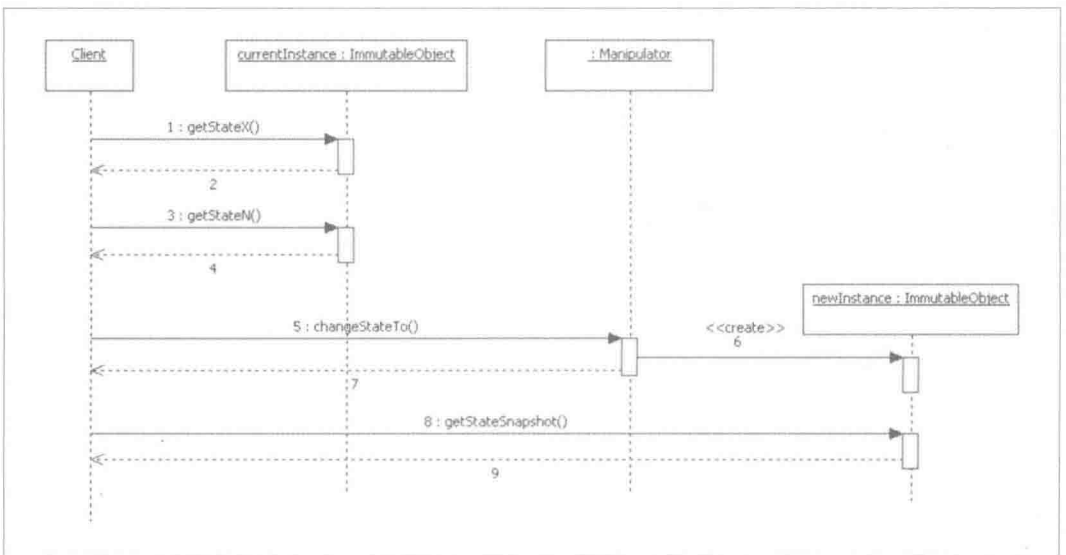


图 3-2. Immutable Object 模式的典型交互场景

第 1～4 步：客户端代码获取当前 ImmutableObject 实例的各个状态值。

第 5 步：客户端代码调用 Manipulator 的 changeStateTo 方法来更新应用的状态。

第 6、7 步：changeStateTo 方法创建新的 ImmutableObject 实例以反映应用的新状态，并返回。

第 8、9 步：客户端代码获取新的 ImmutableObject 实例的状态快照。

一个严格意义上不可变对象要满足以下所有条件。

1. 类本身使用 final 修饰：防止其子类改变其定义的行为。

2. 所有字段都是用 final 修饰的：使用 final 修饰不仅仅是从语义上说明被修饰字段的引用不可改变。更重要的是这个语义在多线程环境下由 JMM（Java Memory Model）保证了被修饰字段所引用对象的初始化安全，即 final 修饰的字段在其他线程可见时，它必定是初始化完成的。相反，非 final 修饰的字段由于缺少这种保证，可能导致一个线程“看到”一个字段的时候，它还未被初始化完成，从而可能导致一些不可预料的结果。

3. 在对象的创建过程中，this 关键字没有泄露给其他类：防止其他类（如该类的内部匿名类）在对象创建过程中修改其状态。

4. 任何字段，若其引用了其他状态可变的对象（如集合、数组等），则这些字段必须是 private 修饰的，并且这些字段值不能对外暴露。若有相关方法要返回这些字段值，应该进行防御性复制（Defensive Copy）。

3.3 Immutable Object 模式实战案例解析

某彩信网关系统在处理由增值业务提供商（VASP，Value-Added Service Provider）下发给手机终端用户的彩信消息时，需要根据彩信接收方号码的前缀（如 1381234）选择对应的彩信中心（MMSC，Multimedia Messaging Service Center），然后转发消息给选中的彩信中心，由其负责对接电信网络将彩信消息下发给手机终端用户。彩信中心相对于彩信网关系统而言，它是一个独立的部件，二者通过网络进行交互。这个选择彩信中心的过程，我们称之为路由（Routing）。而手机号前缀和彩信中心的这种对应关系，被称为路由表。路由表在软件运维过程中可能发生变化。例如，业务扩容带来的新增彩信中心、为某个号码前缀指定新的彩信中心等。虽然路由表在该系统中是由多线程共享的数据，但是这些数据的变化频率并不高。因此，即使是为了保证线程安全，我们也不希望对这些数据的访问进行加锁等并发访问控制，以免产生不必要的开销和问题。这时，Immutable Object 模式就派上用场了。

维护路由表可以被建模为一个不可变对象，如清单 3-4 所示。

清单 3-4. 使用不可变对象维护路由表

```
/**
 * 彩信中心路由规则管理器
 * 模式角色：ImmutableObject.ImmutableObject
 */
public final class MMSCRouter {
    // 用 volatile 修饰，保证多线程环境下该变量的可见性
    private static volatile MMSCRouter instance = new MMSCRouter();
       //维护手机号码前缀到彩信中心之间的映射关系
    private final Map<String, MMSCInfo> routeMap;

    public MMSCRouter() {
        // 将数据库表中的数据加载到内存，存为 Map
        this.routeMap = MMSCRouter.retrieveRouteMapFromDB();
    }

    private static Map<String, MMSCInfo> retrieveRouteMapFromDB() {
        Map<String, MMSCInfo> map = new HashMap<String, MMSCInfo>();
        // 省略其他代码
        return map;
    }

    public static MMSCRouter getInstance() {

        return instance;
    }

    /**
     * 根据手机号码前缀获取对应的彩信中心信息
     *
     * @param msisdnPrefix
     *          手机号码前缀
     * @return 彩信中心信息
     */
    public MMSCInfo getMMSC(String msisdnPrefix) {
        return routeMap.get(msisdnPrefix);

    }

    /**
     * 将当前 MMSCRouter 的实例更新为指定的新实例
     *
     * @param newInstance
     *          新的 MMSCRouter 实例
     */
    public static void setInstance(MMSCRouter newInstance) {
        instance = newInstance;
    }

    private static Map<String, MMSCInfo> deepCopy(Map<String, MMSCInfo> m) {
        Map<String, MMSCInfo> result = new HashMap<String, MMSCInfo>();
        for (String key : m.keySet()) {
            result.put(key, new MMSCInfo(m.get(key)));
        }
        return result;
    }
```

```
    public Map<String, MMSCInfo> getRouteMap() {
        //做防御性复制
        return Collections.unmodifiableMap(deepCopy(routeMap));
    }

}
```

而彩信中心的相关数据，如彩信中心设备编号、URL、支持的最大附件尺寸也被建模为一个不可变对象，如清单 3-5 所示。

清单 3-5. 使用不可变对象表示彩信中心信息

```
/**
 * 彩信中心信息
 *
 * 模式角色：ImmutableObject.ImmutableObject
 */
public final class MMSCInfo {
    /**
     * 设备编号
     */
    private final String deviceID;
    /**
     * 彩信中心 URL
     */
    private final String url;
    /**
     * 该彩信中心允许的最大附件大小
     */
    private final int maxAttachmentSizeInBytes;

    public MMSCInfo(String deviceID, String url, int maxAttachmentSizeInBytes) {
        this.deviceID = deviceID;
        this.url = url;
        this.maxAttachmentSizeInBytes = maxAttachmentSizeInBytes;
    }

    public MMSCInfo(MMSCInfo prototype) {
        this.deviceID = prototype.deviceID;
        this.url = prototype.url;
        this.maxAttachmentSizeInBytes = prototype.maxAttachmentSizeInBytes;
    }

    public String getDeviceID() {
        return deviceID;
    }

    public String getUrl() {
        return url;
    }

    public int getMaxAttachmentSizeInBytes() {
        return maxAttachmentSizeInBytes;
    }

}
```

彩信中心信息变更的频率也同样不高。因此，当彩信网关系统通过网络（Socket 连接）被通知到这种彩信中心信息本身或者路由表变更时，网关系统会重新生成新的 MMSCInfo 和 MMSCRouter 来反映这种变更，如清单 3-6 所示。

清单 3-6. 处理彩信中心、路由表的变更

```
/**
 * 与运维中心（Operation and Maintenance Center）对接的类
 * 模式角色：ImmutableObject.Manipulator
 */
public class OMCAgent extends Thread{

  @Override
  public void run() {                                    \
      boolean isTableModificationMsg=false;
      String updatedTableName=null;
       while(true){
          //省略其他代码
          /*
           * 从与 OMC 连接的 Socket 中读取消息并进行解析，
           * 解析到数据表更新消息后，重置 MMSCRouter 实例。
           */
          if(isTableModificationMsg){
              if("MMSCInfo".equals(updatedTableName)){
                  MMSCRouter.setInstance(new MMSCRouter());
              }
          }
          //省略其他代码
       }
  }

}
```

上述代码会调用 MMSCRouter 的 setInstance 方法来替换 MMSCRouter 的实例为新创建的实例。而新创建的 MMSCRouter 实例通过其构造器会生成多个新的 MMSCInfo 的实例。

本案例中，MMSCInfo 是一个严格意义上的不可变对象。虽然 MMSCRouter 对外提供了 setInstance 方法用于改变其静态字段 instance 的值，但它仍然可被视作一个等效的不可变对象。这是因为 setInstance 方法仅仅改变 instance 变量指向的对象，而 instance 变量采用 volatile 修饰保证了其在多线程之间的内存可见性，所以这意味着 setInstance 对 instance 变量的改变无须加锁也能保证线程安全。而其他代码在调用 MMSCRouter 的相关方法获取路由信息时也无须加锁。

从图 3-1 的类图上看，OMCAgent 类（见清单 3-6）是一个 Manipulator 参与者实例，而 MMSCInfo、MMSCRouter 是一个 ImmutableObject 参与者实例。通过使用不可变对象，我们既可以应对路由表、彩信中心这些不是非常频繁的变更，又可以使系统中使用路由表的代码免于并发访问控制的开销和问题。

3.4 Immutable Object 模式的评价与实现考量

不可变对象具有天生的线程安全性，多个线程共享一个不可变对象的时候无须使用额外的并发访问控制，这使得我们可以避免显式锁等并发访问控制的开销和问题，简化了多线程编程。

Immutable Object 模式特别适用于以下场景。

- **被建模对象的状态变化不频繁**：正如本章案例所展示的，这种场景下可以设置一个专门的线程（Manipulator 参与者所在的线程）用于在被建模对象状态变化时创建新的不可变对象。而其他线程则只是读取不可变对象的状态。此场景下的一个小技巧是 Manipulator 对不可变对象的引用采用 volatile 关键字修饰，既可以避免使用显式锁（如 synchronized），又可以保证多线程间的内存可见性。

- **同时对一组相关的数据进行写操作，因此需要保证原子性**：此场景为了保证操作的原子性，通常的做法是使用显式锁。但若采用 Immutable Object 模式，将这一组相关的数据“组合”成一个不可变对象，则对这一组数据的操作就可以无须加显式锁也能保证原子性，这既简化了编程，又提高了代码运行效率。本章开头所举的车辆位置跟踪的例子正是这种场景。

- **使用某个对象作为安全的 HashMap 的 Key**：我们知道，一个对象作为 HashMap 的 Key 被“放入”HashMap 之后，若该对象状态变化导致了其 Hash Code 的变化，则会导致后面在用同样的对象作为 Key 去 get 的时候无法获取关联的值，尽管该 HashMap 中的确存在以该对象为 Key 的条目。相反，由于不可变对象的状态不变，因此其 Hash Code 也不变。这使得不可变对象非常适于用作 HashMap 的 Key。

Immutable Object 模式实现时需要注意以下几个问题。

- **被建模对象的状态变更比较频繁**：此时也不见得不能使用 Immutable Object 模式。只是这意味着频繁创建新的不可变对象，因此会增加 JVM 垃圾回收（Garbage Collection）的负担和 CPU 消耗，我们需要综合考虑：被建模对象的规模、代码目标运行环境的 JVM 内存分配情况、系统对吞吐率和响应性的要求。若这几个方面因素综合考虑都能满足要求，那么使用不可变对象建模也未尝不可。

- **使用等效或者近似的不可变对象**：有时创建严格意义上的不可变对象比较难，但是尽量向严格意义上的不可变对象靠拢也有利于发挥不可变对象的好处。

- **防御性复制**：如果不可变对象本身包含一些状态需要对外暴露，而相应的字段本身又是可变的（如 HashMap），那么返回这些字段的方法还是需要做防御性复制，以避免外部代码修改了其内部状态。正如清单 3-4 的代码中的 getRouteMap 方法所展示的。

3.5 Immutable Object 模式的可复用实现代码

Immutable Object 模式不便于实现可复用的代码。我们需要根据实际应用具体实现。

3.6 Java 标准库实例

在多线程环境中，遍历一个集合（Collection，如 java.util.Vector）对象时，即便被遍历的对象本身是线程安全的，开发人员仍然不得不引入锁，以防止遍历过程中该集合的内部结构被其他线程改变（如删除或者插入了一个新的元素）而导致出错，如清单 3-7 所示。

清单 3-7. 遍历线程安全的集合时加锁

```
Vector vector = null;

// 此处以 vector 本身为锁，防止遍历过程中的其他线程改变其内部结构
synchronized (vector) {
    for (int i = 0; i < vector.size(); i++) {
        doSomethingWith(vector.get(i));
    }
}
```

为了保证线程安全而在遍历时对集合对象进行加锁，但这在某些情形下可能并不合适，比如系统中对该集合的插入和删除的操作频率远比遍历操作的频率要高。JDK 1.5 中引入的类 java.util.concurrent.CopyOnWriteArrayList 应用了 Immutable Object 模式，使得对 CopyOnWriteArrayList 实例进行遍历时不用加锁也能保证线程安全。当然，CopyOnWriteArrayList 也不是“万能”的，它是专门针对遍历操作的频率比添加和删除操作更加频繁的场景设计的。CopyOnWriteArrayList 的源码（骨架）如清单 3-8 所示。

清单 3-8. JDK 类 CopyOnWriteArrayList 的源码（骨架）

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    //省略其他代码

    /** The array, accessed only via getArray/setArray. */
    private volatile transient Object[] array;

    /**
     * Gets the array.  Non-private so as to also be accessible
     * from CopyOnWriteArraySet class.
     */
    final Object[] getArray() {
        return array;
    }

    /**
```

```
     * Sets the array.
     */
    final void setArray(Object[] a) {
        array = a;
    }

    //省略其他代码

    //遍历集合
    public Iterator<E> iterator() {
        return new COWIterator<E>(getArray(), 0);
    }

    //添加元素
    public boolean add(E e) {
     final ReentrantLock lock = this.lock;
     lock.lock();
     try {
         Object[] elements = getArray();
         int len = elements.length;

            //复制原数组，并在此基础上将新数组的最后一个元素设置为要添加的元素
         Object[] newElements = Arrays.copyOf(elements, len + 1);
         newElements[len] = e;

               //直接将 array 变量设置为新的数组
         setArray(newElements);
         return true;
     } finally {
         lock.unlock();
     }
    }

    //省略其他代码
}
```

从清单 3-8 的代码可见，CopyOnWriteArrayList 内部维护一个名为 array 的实例变量用于存储集合的各个元素。在集合中添加一个元素的时候，CopyOnWriteArrayList 会生成一个新的数组，并将集合中现有的元素都复制到新的数组，然后将新的数组的最后一个元素设置为要添加的元素。这个新的数组会直接被赋值给 array 实例变量。这里，实例变量 array 引用的数组就是一个等效的 Immutable Object[2]，其内容一旦确定下来就不再被改变。因此，遍历的 CopyOnWriteArrayList 维护各个元素的时候，直接根据 array 实例变量生成一个 Iterator 实例即可，而无须加锁[3]。

2 之所以称之等效，是因为 CopyOnWriteArrayList 的实例变量 array 是个数组，而数组的各个元素的值是可替换的。因此，CopyOnWriteArrayList 的实例变量 array 并非严格意义上的 Immutable Object。

3 虽然 CopyOnWriteArrayList 的 add 方法为了保证复制实例变量 array 引用的老数组时的线程安全里使用了锁，但是只要对 CopyOnWriteArrayList 进行遍历时不加锁就已经达到了 CopyOnWriteArrayList 的设计目标（即使其适用于遍历比修改操作更加频繁的场景）。

3.7 相关模式

3.7.1 Thread Specific Storage 模式（第 10 章）

Immutable Object 模式使得我们可以在不使用显式锁的情况下保证线程安全。Thread Specific Storage（线程特有存储）模式也可以帮我们达到相同的效果，只不过二者的具体实现方式不同。

3.7.2 Serial Thread Confinement 模式（第 11 章）

Serial Thread Confinement 模式也可以使我们在不使用显式锁的情况下保证线程安全。只不过，在使用 Serial Thread Confinement 模式来实现线程安全的时候，Serial Thread Confinement 模式用到的队列本身实际上涉及了显式锁。因此，使用 Serial Thread Confinement 模式来保证线程安全实际上是试图用一种更小的锁开销去（队列所涉及的锁开销）替代另外一种可能更大的锁开销（工作者线程如果采用锁所带来的开销）。

3.8 参考资源

1．Brian Göetz. Java theory and practice: To mutate or not to mutate?. http://www.ibm.com/developerworks/java/library/j-jtp02183/index.html.

2．Brian Göetz et al. Java Concurrency In Practice. Addison Wesley,2006.

3．Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

4．Java Language Specification. 17.5. final Field Semantics. http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html.

第4章

Guarded Suspension（保护性暂挂）模式

4.1 Guarded Suspension 模式简介

多线程编程中，为了提高并发性，往往将一个任务分解为不同的部分，将其交由不同的线程来执行。这些线程间相互协作时，仍然可能会出现一个线程去等待另一个线程完成一定的操作，其自身才能继续运行的情形。这好比汽车行驶过程中油量不足时，司机只好到加油站等工作人员将油加满才能继续行驶。

Guarded Suspension 模式可以帮助我们解决上述的等待问题。该模式的核心思想是如果某个线程执行特定的操作前需要满足一定的条件，则在该条件未满足时将该线程暂停运行（即暂挂线程，使其处于等待（WAITING）状态，直到该条件满足时才继续该线程的运行）。这里，读者可能会想到 wait/notify[1]。的确，wait/notify 可以用来实现 Guarded Suspension 模式。但是，Guarded Suspension 模式还要解决 wait/notify 所解决的问题之外的问题。

4.2 Guarded Suspension 模式的架构

Guarded Suspension 模式的核心是一个受保护方法（Guarded Method）。该方法执行其所要真正执行的操作时需要满足特定的条件（Predicate，以下称之为保护条件）。当该条件不满足时，执行受保护方法的线程会被挂起进入等待（WAITING）状态，直到该条件满足时该线程才会继续运行。此时，受保护方法才会真正执行其所要执行的操作。为方便起见，以下称受保护

1 指 java.lang.Object 的 wait()和 notify()方法。

方法所要真正执行的操作为目标动作。

Guarded Suspension 模式的主要参与者有以下几种。其类图如图 4-1 所示。

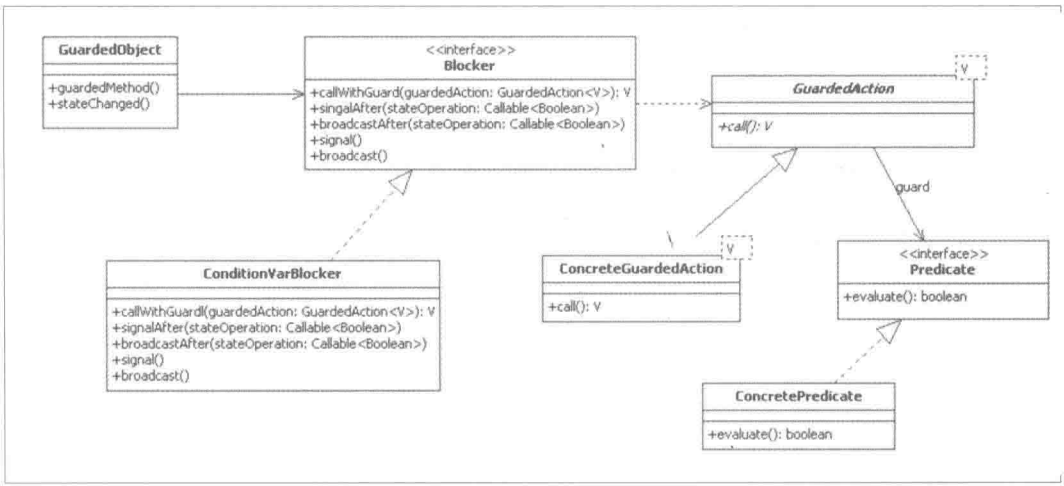


图 4-1. Guarded Suspension 模式的类图

- **GuardedObject**：包含受保护方法的对象。其主要方法及职责如下。
  - **guardedMethod**：受保护方法。
  - **stateChanged**：改变 GuardedObject 实例状态的方法。该方法负责在保护条件成立时唤醒受保护方法的执行线程。
- **GuardedAction**：抽象了目标动作，并关联了目标动作所需的保护条件。其主要方法及职责如下。
  - **call**：用于表示目标动作的方法。
- **ConcreteGuardedAction**：应用程序所实现的具体目标动作及其关联的保护条件。
- **Predicate**：抽象了保护条件。其主要方法及职责如下。
  - **evaluate**：用于表示保护条件的方法。
- **ConcretePredicate**：应用程序所实现的具体保护条件。
- **Blocker**：负责对执行 guardedMethod 方法的线程进行挂起和唤醒，并执行 ConcreteGuardedAction 所实现的目标操作。其主要方法及职责如下。

- **callWithGuard**：负责执行目标操作和暂挂当前线程。
- **signalAfter**：负责执行其参数指定的动作和唤醒由该方法所属 Blocker 实例所暂挂的线程中的一个线程。
- **signal**：负责唤醒由该方法所属 Blocker 实例所暂挂的线程中的一个线程。
- **broadcastAfter**：负责执行其参数指定的动作和唤醒由该方法所属 Blocker 实例所暂挂的所有线程。
- **broadcast**：负责唤醒由该方法所属 Blocker 实例暂挂的所有线程。

- **ConditionVarBlocker**：基于 Java 条件变量（java.util.concurrent.locks.Condition）实现的 Blocker。

受保护方法的内部处理逻辑如图 4-2 所示的序列图。

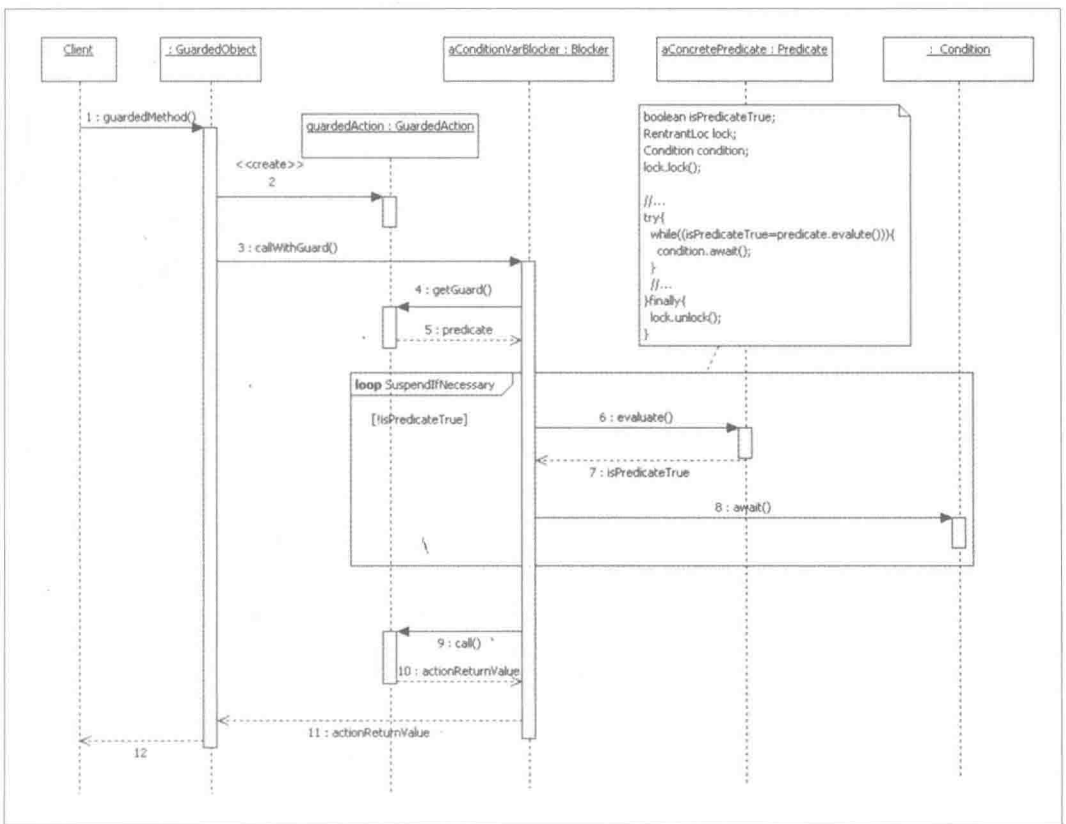


图 4-2. 受保护方法的内部处理逻辑

第 1 步：客户端代码调用受保护方法 guardedMethod。

第 2 步：guardedMethod 方法创建 GuardedAction 实例 guardedAction。

第 3 步：guardedMethod 方法以 guardedAction 为参数调用 Blocker 实例的 callWithGuard 方法。

第 4、5 步：callWithGuard 方法调用 guardedAction 的 getGuard 方法获取保护条件 Predicate。

第 6~8 步：这几个步骤是个循环。该循环会判断保护条件是否成立（第 6、7 步）。若保护条件成立，则该循环退出。否则，该循环会将当前线程暂挂使其处于等待（WAITING）状态（第 8 步）。当其他线程唤醒该被暂挂的线程后（即第 8 步的方法调用返回），该循环仍然继续检测保护条件，并重复上述逻辑。

第 9、10 步：callWithGuard 方法调用 guardedAction 的 call 方法来执行目标动作，并记录 call 方法的返回值 actionReturnValue。

第 11 步：callWithGuard 方法将 actionReturnValue 作为其返回值返回给调用方。

第 12 步：guardedMethod 方法返回。

受保护方法的执行线程被暂挂后，当保护条件成立时，其他线程需要唤醒该线程，如图 4-3 所示的序列图。

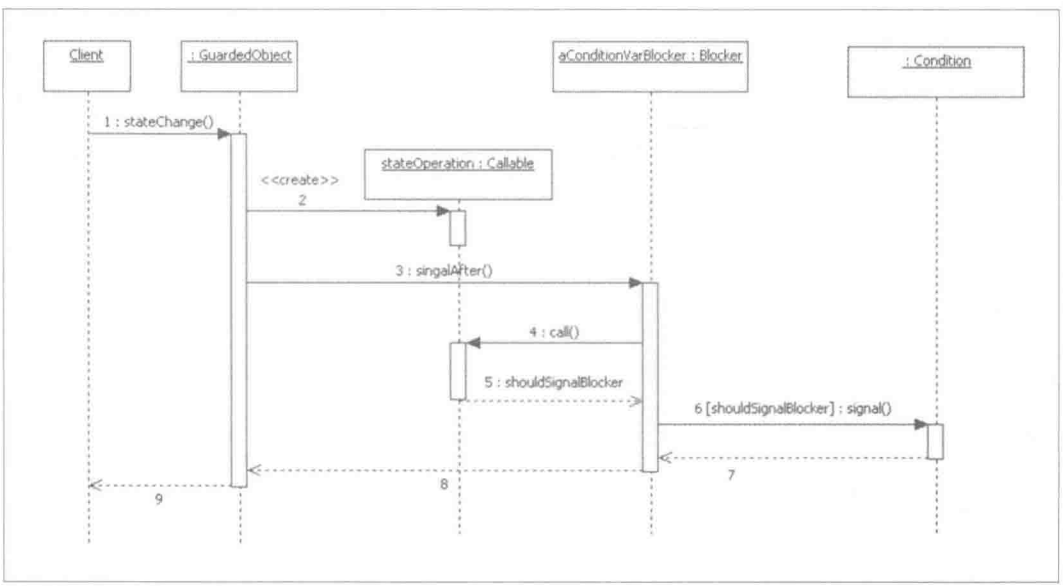


图 4-3. 被暂挂的线程的唤醒

第 1 步：客户端代码调用 stateChanged 方法改变 GuardedObject 实例的状态。这些状态包含了保护条件所关心的状态。

第 2 步：stateChanged 方法创建 java.util.concurrent.Callable 实例 stateOperation。stateOperation 封装了改变 GuardedObject 实例状态所需的操作。

第 3 步：stateChanged 方法以 stateOperation 为参数，调用 Blocker 实例的 signalAfter 方法。

第 4、5 步：signalAfter 方法调用 stateOperation 对象的 call 方法以改变 GuardedObject 实例的状态，并记录其返回值 shouldSignalBlocker。

第 6、7 步：signalAfter 方法在 shouldSignalBlocker 值为 true 时，调用 java.util.concurrent.locks.Condition 实例的 signal 方法来唤醒被该 Condition 实例所暂挂的线程中的一个线程。

第 8 步：signalAfter 方法返回。此时，执行受保护方法的线程可能已经被唤醒(取决于 shouldSignalBlocker 的值是否为 true)。

第 9 步：stateChanged 方法返回。

4.3 Guarded Suspension 模式实战案例解析

某系统有个告警[2]功能模块。该模块的主要功能是将其接收到的告警信息发送给告警服务器。该模块中的类 AlarmAgent 负责与告警服务器进行对接。AlarmAgent 的 sendAlarm 方法负责通过网络连接（Socket 连接）将告警信息发送到告警服务器。AlarmAgent 创建了一个专门的线程用于其与告警服务器建立网络连接。因此，sendAlarm 方法被调用的时候，连接线程可能还没有完成网络连接的建立。此时，sendAlarm 方法应该等待连接线程建立好网络连接。另外，即便连接线程建立好了网络连接，中途也可能由于某些原因出现与告警服务器断连的情况。此时，sendAlarm 方法需要等待心跳（Heartbeat）任务[3]重新建立好连接才能上报告警信息。也就是说，sendAlarm 方法必须在 AlarmAgent 与告警服务器的网络连接建立成功的情况下才能执行其所要执行的操作。若 AlarmAgent 与告警服务器的连接未建立(或者连接中断)，sendAlarm 方法的执行线程应该暂挂直到连接建立完毕（或者恢复）。

上述问题可以应用 Guarded Suspension 模式来解决，如清单 4-1 所示。

2 所谓告警，类似于智能手机监视其电池的电量。当可用电量少于特定值（如 14%）时，手机就会提醒用户电量不足。

3 心跳任务通过定时（如每 2s）执行一个动作（如给服务器发送一个请求）来检测目标服务是否可用。

清单 4-1. AlarmAgent 类源码

```
/**
 * 负责连接告警服务器，并发送告警信息至告警服务器
 *
 */
public class AlarmAgent {
    // 用于记录 AlarmAgent 是否连接上告警服务器
    private volatile boolean connectedToServer = false;

    // 模式角色：GuardedSuspension.Predicate
    private final Predicate agentConnected = new Predicate() {
        @Override
        public boolean evaluate() {
            return connectedToServer;
        }
    };

    // 模式角色：GuardedSuspension.Blocker
    private final Blocker blocker = new ConditionVarBlocker();

    // 心跳定时器
    private final Timer heartbeatTimer = new Timer(true);

    // 省略其他代码

    /**
     * 发送告警信息
     * @param alarm 告警信息
     * @throws Exception
     */
    public void sendAlarm(final AlarmInfo alarm) throws Exception {
        // 可能需要等待，直到 AlarmAgent 连接上告警服务器（或者连接中断后重新连上服务器）
        // 模式角色：GuardedSuspension.GuardedAction
        GuardedAction<Void> guardedAction = new GuardedAction<Void>(agentConnected)
{
            public Void call() throws Exception {
                doSendAlarm(alarm);
                return null;
            }
        };

        blocker.callWithGuard(guardedAction);
    }

    // 通过网络连接将告警信息发送给告警服务器
    private void doSendAlarm(AlarmInfo alarm) {
        // 省略其他代码
        Debug.info("sending alarm " + alarm);

        //模拟发送告警至服务器的耗时
        try {
            Thread.sleep(50);
        } catch (Exception e) {

        }
    }
```

```
public void init() {
    // 省略其他代码

    // 告警连接线程
    Thread connectingThread = new Thread(new ConnectingTask());

    connectingThread.start();

    heartbeatTimer.schedule(new HeartbeatTask(), 60000, 2000);
}

public void disconnect() {
    // 省略其他代码
    Debug.info("disconnected from alarm server.");
    connectedToServer = false;
}

protected void onConnected() {
    try {
        blocker.signalAfter(new Callable<Boolean>() {
            @Override
            public Boolean call() {
                connectedToServer = true;
                Debug.info("connected to server");
                return Boolean.TRUE;
            }
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected void onDisconnected() {
    connectedToServer = false;
}

//负责与告警服务器建立网络连接
private class ConnectingTask implements Runnable{
    @Override
    public void run() {
        // 省略其他代码

        // 模拟连接操作耗时
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            ;
        }

        onConnected();
    }
}

/**
 * 心跳定时任务：定时检查与告警服务器的连接是否正常，发现连接异常后自动重新连接
 */
private class HeartbeatTask extends TimerTask {
    // 省略其他代码
```

```
        @Override
        public void run() {
            // 省略其他代码

            if (!testConnection()) {
                onDisconnected();
                reconnect();
            }

        }

        private boolean testConnection() {
            // 省略其他代码

            return true;
        }

        private void reconnect() {
            ConnectingTask connectingThread = new ConnectingTask();

            //直接在心跳定时器线程中执行
            connectingThread.run();
        }

    }
}
```

如清单 4-1 所示代码中的 sendAlarm 方法会等待其所属的 AlarmAgent 实例连接上告警服务器才调用 doSendAlarm 方法真正将告警信息上报到告警服务器。若 sendAlarm 方法被调用时，AlarmAgent 实例与告警服务器的连接未建立完毕或者连接中断了，则 sendAlarm 方法会使当前线程阻塞，直到其他线程唤醒该被阻塞的线程且 AlarmAgent 实例与告警服务器的连接建立完毕。可见，AlarmAgent 类相当于 Guarded Suspension 模式中的 GuardedObject 参与者实例，其 sendAlarm 方法是个保护方法。sendAlarm 方法的目标动作是 doSendAlarm 所执行的操作（上报告警信息至告警服务器）。sendAlarm 方法的保护条件是 AlarmAgent 实例与告警服务器已建立网络连接（即 connectedToServer 的值为 true）。

AlarmAgent 的实例变量 agentConnected 相当于 Predicate 参与者实例。AlarmAgent 的实例变量 blocker 相当于 Blocker 参与者实例。

AlarmAgent 的 onConnected 方法相当于 GuardedObject 参与者的 stateChanged 方法。当告警连接线程建立起与告警服务器的连接，或者心跳定时任务在网络连接中断后重新连接完毕时，onConnected 方法都会被调用到。此时，onConnected 方法通过调用 Blocker 实例 blocker 的 signalAfter 方法[4]，将标志变量 connectedToServer 置为 true，并“通知”blocker 去唤醒被暂挂

4 本案例中，AlarmAgent 的 sendAlarm 方法由专门的一个线程调用（详情可见第 14 章的案例），因此这里唤醒被暂挂的 sendAlarm 方法执行线程时调用 signalAfter 方法即可，而无须调用 broadcastAfter 方法。

的 sendAlarm 方法执行线程。

清单 4-1 代码所引用的其他类的源码，如清单 4-2、清单 4-3、清单 4-4 和清单 4-5 所示。

清单 4-2. Predicate 接口源码

```
public interface Predicate {
    boolean evaluate();
}
```

清单 4-3. GuardedAction 类源码

```
public abstract class GuardedAction<V> implements Callable<V> {
    protected final Predicate guard;

    public GuardedAction(Predicate guard) {
        this.guard = guard;
    }
}
```

清单 4-4. Blocker 接口源码

```
public interface Blocker {

    /**
     * 在保护条件成立时执行目标动作；否则阻塞当前线程，直到保护条件成立。
     * @param guardedAction 带保护条件的目标动作
     * @return
     * @throws Exception
     */
    <V> V callWithGuard(GuardedAction<V> guardedAction) throws Exception;

    /**
     * 执行 stateOperation 所指定的操作后，决定是否唤醒本 Blocker
     * 所暂挂的所有线程中的一个线程。
     *
     * @param stateOperation
     *          更改状态的操作，其 call 方法的返回值为 true 时，该方法才会唤醒被暂挂的线程
     */
    void signalAfter(Callable<Boolean> stateOperation) throws Exception;

    void signal() throws InterruptedException;

    /**
     * 执行 stateOperation 所指定的操作后，决定是否唤醒本 Blocker
     * 所暂挂的所有线程。
     *
     * @param stateOperation
     *          更改状态的操作，其 call 方法的返回值为 true 时，该方法才会唤醒被暂挂的线程
     */
    void broadcastAfter(Callable<Boolean> stateOperation) throws Exception;
}
```

清单 4-5. ConditionVarBlocker 类源码

```
public class ConditionVarBlocker implements Blocker {
    private final Lock lock;
```

```
private final Condition condition;

public ConditionVarBlocker(Lock lock) {

    this.lock = lock;
    this.condition = lock.newCondition();
}

public ConditionVarBlocker() {

    this.lock = new ReentrantLock();
    this.condition = lock.newCondition();
}

public <V> V callWithGuard(GuardedAction<V> guardedAction) throws Exception {
    lock.lockInterruptibly();
    V result;
    try {
        final Predicate guard = guardedAction.guard;
        while (!guard.evaluate()) {
            condition.await();
        }
        result = guardedAction.call();
        return result;
    } finally {
        lock.unlock();
    }
}

public void signalAfter(Callable<Boolean> stateOperation) throws Exception {
    lock.lockInterruptibly();
    try {
        if (stateOperation.call()) {
            condition.signal();
        }
    } finally {
        lock.unlock();
    }

}

public void broadcastAfter(Callable<Boolean> stateOperation) throws Exception {
    lock.lockInterruptibly();
    try {
        if (stateOperation.call()) {
            condition.signalAll();
        }
    } finally {
        lock.unlock();
    }

}

public void signal() throws InterruptedException {
    lock.lockInterruptibly();
    try {
        condition.signal();

    } finally {
```

```
            lock.unlock();
        }
    }
}
```

4.4 Guarded Suspension 模式的评价与实现考量

Guarded Suspension 模式使应用程序避免了样板式代码。Guarded Suspension 模式的 Blocker 参与者所实现的线程挂起与唤醒功能固然可以由应用代码直接使用 wait/notify 或者 java.util.concurrent.locks.Condition 来实现，但是这里面涉及几个比较容易犯错的重要技术细节（下文会提到）。这些细节如果散落在应用代码中，则会增加出错的概率。另外，应用直接编写代码来正确实现这些技术细节往往导致许多样板式代码。这不仅增加了代码编写的工作量，而且也增加了出错的概率。相反，Guarded Suspension 模式的 Blocker 参与者封装了这些易错的技术细节，从而减少了应用代码的编写量和出错的概率。

关注点分离（Separation of Concern）。Guarded Suspension 模式中的各个参与者各自仅关注本模式所要解决的问题中的一个方面，各个参与者的职责是高度内聚（Cohesive）的。这使得 Guarded Suspension 模式便于理解和应用，其实现代码也便于阅读和维护。应用开发人员只需要根据应用的需要实现 GuardedObject、ConcretePredicate 和 ConcreteGuardedAction 这几个必须由应用实现的参与者即可，而其他的参与者的实现都是可复用的。

可能增加 JVM 垃圾回收的负担。为了使 GuardedAction 实例的 call 方法能够访问保护方法 guardedMethod 的参数，我们需要利用闭包（Closure）。因此，GuardedAction 实例可能是在保护方法中创建的。这意味着，每次保护方法被调用的时候都会有个新的 GuardedAction 实例被创建。而这会增加 JVM 内存池 Eden 区域内存的占用，从而可能增加 JVM 垃圾回收的负担。如果应用程序所在 JVM 的内存池 Eden 区域空间比较小，则需要特别注意 GuardedAction 实例创建可能导致的垃圾回收负担。

可能增加上下文切换（Context Switch）。严格来说，这点与 Guarded Suspension 模式本身无关。只不过，不管如何实现 Guarded Suspension 模式，只要这里面涉及线程的暂挂和唤醒就会引起上下文切换。如果频繁出现保护方法被调用时保护条件不成立，那么保护方法的执行线程就会频繁地被暂挂和唤醒，而导致频繁的上下文切换。过于频繁的上下文切换会过多消耗系统的 CPU 时间，从而降低系统处理能力。

Blocker 实现类中封装的几个易错的重要技术细节介绍如下。

4.4.1 内存可见性和锁泄漏（Lock Leak）

保护条件中涉及的变量牵涉读线程和写线程进行共享访问。保护方法的执行线程是读线程，它读取这些变量以判断保护条件是否成立。而写线程是受保护对象实例的 stateChanged 方法的执行线程，它会去更改这些变量的值。因此，对保护条件涉及的变量的访问应该使用锁进行保护，以保证写线程对这些变量所做的更改，读线程能够“看到”相应的值。从清单 4-5 中的代码可以看出，这点已经被封装在 Blocker 实例的几个方法中了。应用代码只需要在创建 Blocker 实例时在其构造器中指定恰当的锁实例即可。

ConditionVarBlocker 类（代码见清单 4-5）为了保证保护条件中涉及的变量的内存可见性而引入 ReentrantLock 锁。使用该锁时需要注意临界区中的代码无论是执行正常还是出现异常，进入临界区前获得的锁实例都应该被释放。否则，就会出现锁泄漏现象：锁对象被某个线程获得，但永远不会被该线程释放，导致其他线程无法获得该锁。为了避免锁泄漏，使用 ReentrantLock 的临界区代码总是需要按照如下格式来编写：

```
//获得锁
lock.lockInterruptibly();
        try {
            //临界区代码

        } finally {
//在 finally 块中释放锁，保证锁总是会被释放的
            lock.unlock();
        }
```

4.4.2 线程过早被唤醒

ConditionVarBlocker 类的 callWithGuard 方法对 Condition 实例的 await 方法是放在一个 while 循环中的，而不是在 if 语句中。这是因为 Condition 实例的 await 方法使得当前线程暂挂后，其他线程调用了 Condition 实例的 signal 或者 signalAll 方法固然可以唤醒被暂挂的线程，但是这并不能保证此时保护条件就是成立的。因此，这个时候 callWithGuard 方法应该继续检测保护条件，直到保护条件成立。

ConditionVarBlocker 类是基于 Condition 接口实现的。由 callWithGuard 方法的签名可知，一个 Condition 实例可以对应多个保护方法。假设某 Condition 实例 aCondition 对应多个保护条件：predicateA 和 predicateB。当某个线程 threadA 发现 predicateA 成立时，它调用了 aCondition 的 notifyAll 方法，那么此时所有被 aCondition 暂挂的线程都会被唤醒。这些被唤醒的线程中的某个线程 threadB 需要等待的保护条件是 predicateB，显然这时 predicateB 不一定成立。也

就是，这时线程 threadB 被“过早”地唤醒了[5]。

导致线程被“过早”地唤醒的因素还有所谓的欺骗性唤醒（Spurious Wakeup），即在没有其他任何线程调用 Condition 实例的 notify/notifyAll 方法的情况下，Condition 实例的 await 方法就返回了。

因此，为了应对被暂挂线程可能“过早”地被唤醒的情形，callWithGuard 方法对保护条件的检测和对 Condition 实例的 await 方法的调用总是被放在一个 while 循环而非 if 语句中。代码样板如下所示：

```
lock.lockInterruptibly();

try {
    while (保护条件不成立) {
        condition.await();
    }

    执行目标动作

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

在应用 Guarded Suspension 模式的时候还需要注意嵌套监视器锁死问题（Nested Monitor Lockout）。

4.4.3 嵌套监视器锁死

ConditionVarBlocker 实例的 callWithGuard 方法和 signalAfter/signal/broadcastAfter/broadcast 方法本身涉及了由 java.util.concurrent.locks.Lock 实例实现的同步块。如果这些方法又在另外一个同步块代码中被调用，那么这就形成了嵌套同步。而嵌套同步可能产生锁死的问题（类似死锁）。下面看一个嵌套同步导致锁死的示例代码（见清单 4-6）。

清单 4-6. 嵌套监视器锁死示例代码

```
public class NestedMonitorLockoutExample {

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

5 这种情形下，调用 notify 方法也是类似的。这是因为 notify 方法唤醒的一个线程具体是等待哪个保护条件的线程，这点是无法保证的。因此，也可能出现这样的情形：保护条件 predicateA 成立时，某线程调用 Condition 实例的 notify 方法所唤醒的线程恰是之前被该 Condition 实例所阻塞的等待保护条件 predicateB 的线程。

```
        final Helper helper = new Helper();
        Debug.info("Before calling guaredMethod.");

        Thread t = new Thread(new Runnable() {

            @Override
            public void run() {
                String result;
                result = helper.xGuarededMethod("test");
                Debug.info(result);
            }

        });
        t.start();

        final Timer timer = new Timer();

        // 延迟 50ms 调用 helper.stateChanged 方法
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                helper.xStateChanged();
                timer.cancel();
            }

        }, 50, 10);

    }

    private static class Helper {
        private volatile boolean isStateOK = false;
        private final Predicate stateBeOK = new Predicate() {

            @Override
            public boolean evaluate() {
                return isStateOK;
            }

        };

        private final Blocker blocker = new ConditionVarBlocker();

        public synchronized String xGuarededMethod(final String message) {
            GuardedAction<String> ga = new GuardedAction<String>(stateBeOK) {

                @Override
                public String call() throws Exception {
                    return message + "->received.";
                }

            };
            String result = null;
            try {
                result = blocker.callWithGuard(ga);
            } catch (Exception e) {
                e.printStackTrace();
            }
            return result;
        }
```

```
        public synchronized void xStateChanged() {
            try {
                blocker.signalAfter(new Callable<Boolean>() {

                    @Override
                    public Boolean call() throws Exception {
                        isStateOK = true;
                        Debug.info("state ok.");
                        return Boolean.TRUE;
                    }

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

        }
    }
}
```

如清单 4-6 所示的程序，如果我们将其 xGuardedMethod 方法和 xStateChanged 方法的 synchronized 关键字去掉，那么运行该程序可以得到类似如下的输出：

```
[2015-04-03 22:51:31.742][INFO][main]:Before calling guaredMethod.
[2015-04-03 22:51:31.798][INFO][Timer-0]:state ok.
[2015-04-03 22:51:31.799][INFO][Thread-0]:test->received.
```

但是，如果保留 xGuardedMethod 方法和 xStateChanged 方法的 synchronized 关键字，那么上述程序运行的时候，程序的输出始终只有上面显示的输出的第 1 行。这说明 xGuardedMethod 方法的调用一直没有返回。

xGuardedMethod 方法最终会调用到 java.util.concurrent.locks.Condition 实例的 await 方法。Condition 实例的 await 方法在其被调用之后返回之前会释放与其所属的 Condition 实例所关联的锁，但是 xGuardedMethod 方法本身所获得的锁（即 guardedMethod 方法所属的 NestedMonitorLockoutExample 实例）并没有被释放。而 Condition 实例的 await 方法要返回需要其他线程调用 Condition 实例的 signal/signalAll 方法。但是，调用 xStateChanged 方法所需获得的锁此时还是被 xGuardedMethod 方法的执行线程所保留。这就意味着其他线程无法通过调用 xStateChanged 方法使得 xGuardedMethod 方法所调用的 Condition 实例的 await 方法返回，而这点又导致 xStateChanged 方法调用时所需的锁无法被获得。这就形成了锁死，如图 4-4 所示。

从如图 4-4 所示的调用栈（Call Stack）可知，线程 Thread-0 和线程 Timer-0 一直都处于这样一个状态：前者拥有一个锁，而后者等待获取前者拥有的锁，但是前者又等待后者唤醒，之后才会释放其拥有的锁。

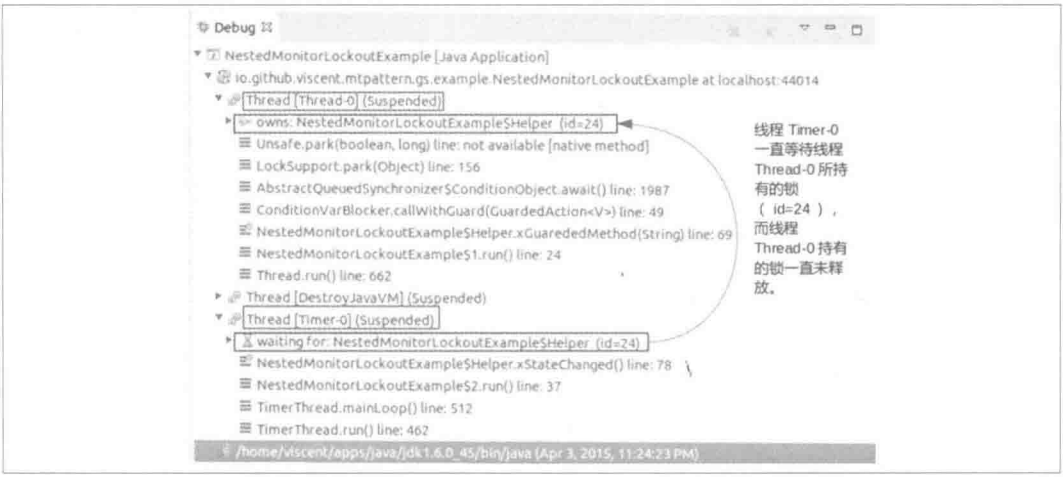


图 4-4. 嵌套监视器锁死的线程示例

如果我们能够避免不必要的嵌套同步，那么嵌套监视器锁死的问题也就可以避免。ConditionVarBlocker 的构造器支持传入一个 java.util.concurrent.locks.Lock 实例正是出于避免不必要的嵌套同步的考虑。该构造器使得 ConditionVarBlocker 调用 Condition 实例的相关方法时可以使用指定的 Lock 实例，而不是 ConditionVarBlocker 实例自己创建的 Lock 实例。

4.5 Guarded Suspension 模式的可复用实现代码

本章案例代码（见清单 4-2、清单 4-3、清单 4-4 和清单 4-5）所实现的 Guarded Suspension 模式的几个参与者（Predicate、GuardedAction、Blocker 和 ConditionVarBlocker）都是可复用的。在此基础上，应用代码只需要根据应用自身的需要实现 GuardedObject、ConcretePredicate 和 ConcreteGuardedAction 这几个参与者即可。

本章案例所使用的 Blocker 实现类 ConditionVarBlocker 是基于 java.util.concurrent.locks.Condition 实现的，如果必要的话，读者也可以编写自己的实现类，而本章案例的其他可复用代码可以不加修改。

4.6 Java 标准库实例

JDK 1.5 开始提供的阻塞队列类 java.util.concurrent.LinkedBlockingQueue 就使用了 Guarded Suspension 模式。该类的 take 方法用于从队列中取出一个元素。如果 take 方法被调用时，队列是空的，则当前线程会被阻塞；直到队列不为空时，该方法才返回一个出队列的元素。只

不过 LinkedBlockingQueue 在实现 Guarded Suspension 模式时，直接使用了 java.concurrent.locks.Condition。

4.7 相关模式

Guarded Suspension 模式是多线程设计模式中的一个基础模式，不仅在应用程序中使用频繁，而且也有其他模式会用到它。

4.7.1 Promise 模式（第 6 章）

Promise 模式中，客户端代码调用 Promise 实例的 getResult 方法时，如果异步任务尚未执行完毕，则 getResult 方法会使当前线程阻塞，直到异步任务处理完毕或者出现异常。

4.7.2 Producer-Consumer 模式（第 7 章）

Producer-Consumer 模式中，当消费者线程所需的“产品”暂时没有时，消费者线程会等待直到生产者线程“生产”了新的“产品”。

4.8 参考资源

1．Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

2．Doug Lea. Concurrent Programming in Java: Design Principles and Patterns, Second Edition. Addison Wesley, 1999.

3．JDK 标准库源码. http://www.docjar.com/html/api/java/util/concurrent/ LinkedBlockingQueue.java.html.

第5章

Two-phase Termination（两阶段终止）模式

5.1 Two-phase Termination 模式简介

停止线程是一个目标简单而实现却不那么简单的任务。首先，Java 没有提供直接的 API 用于停止线程[1]。此外，停止线程还有一些额外的细节需要考虑，如待停止的线程处于阻塞（如等待锁）或者等待状态（等待其他线程）、尚有未处理完的任务等。

Two-phase Termination 模式通过将停止线程这个动作分解为准备阶段和执行阶段这两个阶段，提供了一种通用的用于优雅[2]地停止线程的方法。

准备阶段。该阶段的主要动作是“通知”目标线程（欲停止的线程）准备进行停止。这一步会设置一个标志变量用于指示目标线程可以准备停止了。但是，由于目标线程可能正处于阻塞状态（等待锁的获得）、等待状态（如调用 Object.wait）或者 I/O（如 InputStream.read）等待等状态，即便设置了这个标志，目标线程也无法立即“看到”这个标志而做出相应动作。因此，这一阶段还需要通过调用目标线程的 interrupt 方法，以期望目标线程能够通过捕获相关的异常侦测到该方法调用，从而中断其阻塞状态、等待状态。对于能够对 interrupt 方法调用做出响应的方法（参见表 5-1），目标线程代码可以通过捕获这些方法抛出的 InterruptedException 来侦测线程停止信号。但也有一些方法（如 InputStream.read）并不对 interrupt 调用做出响应，此时需要我们手工处理，如同步的 Socket I/O 操作中通过关闭 socket，

1 ava.lang.Thread 类的 stop 方法早已被不提倡使用了。

2 所谓“优雅”是指可以等要停止的线程在其处理完待处理的任务后才停止，而不是强行停止。

使处于 I/O 等待的 socket 抛出 java.net.SocketException。

表 5-1. 能够对 Thread.interrupt 做出响应的一些方法

方法（或者类）	响应 interrupt 调用抛出的异常
Object.wait() 、Object.wait(long timeout) 、Object.wait (long timeout, int nanos)	InterruptedException
Thread.sleep(long millis) 、Thread.sleep(long millis, int nanos)	InterruptedException
Thread.join()、Thread.join(long millis) 、Thread.Join (long millis, int nanos)	InterruptedException
java.util.concurrent.BlockingQueue.take()	InterruptedException
java.util.concurrent.locks.Lock.lockInterruptibly()	InterruptedException
java.nio.channels.InterruptibleChannel	java.nio.channels.ClosedByInterruptException

执行阶段。该阶段的主要动作是检查准备阶段所设置的线程停止标志和信号，在此基础上决定线程停止的时机，并进行适当的“清理”操作。

5.2 Two-phase Termination 模式的架构

Two-phase Termination 模式的主要参与者有以下几种。其类图如图 5-1 所示。

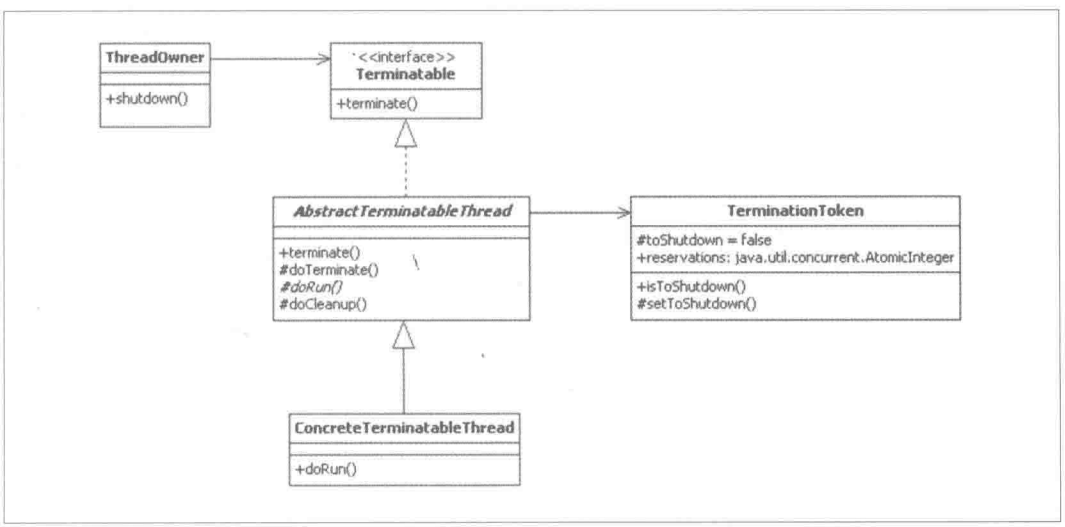


图 5-1. Two-phase Termination 模式的类图

- **ThreadOwner**：目标线程的拥有者。Java 语言中，并没有线程拥有者的概念，但是线程的背后是其要处理的任务或者其所提供的服务，因此我们不能在不清楚某个线程具体是做什么的情况下贸然将其停止。一般地，我们可以将目标线程的创建者视为该线程的拥有者，并假定其“知道”目标线程的工作内容，可以安全地停止目标线程。
- **Terminatable**：可停止线程的抽象。其主要方法及职责如下。
  - **terminate**：请求目标线程停止。
- **AbstractTerminatableThread**：可停止的线程。其主要方法及职责如下。
  - **terminate**：设置线程停止标志，并发送停止“信号”给目标线程。
  - **doTerminate**：留给子类实现线程停止时所需的一些额外操作，如目标线程代码中包含 Socket I/O，子类可以在该方法中关闭 Socket 以达到快速停止线程，而不会使目标线程等待 I/O 完成才能侦测到线程停止标记。
  - **doRun**：线程处理逻辑方法。留给子类实现线程的处理逻辑。相当于 Thread.run()，只不过该方法中无须关心停止线程的逻辑，因为这个逻辑已经被封装在 TerminatableThread 的 run 方法中了。
  - **doCleanup**：留给子类实现线程停止后可能需要的一些清理动作。
- **TerminationToken**：线程停止标志。toShutdown 用于指示目标线程可以停止了。reservations 可用于反映目标线程还有多少数量未完成的任务，以支持等目标线程处理完其任务后再行停止。
- **ConcreteTerminatableThread**：由应用自己实现的 AbstractTerminatableThread 参与者的实现类。该类需要实现其父类的 doRun 抽象方法，在其中实现线程的处理逻辑，并根据应用的实际需要覆盖（Override）其父类的 doTerminate 方法、doCleanup 方法。

准备阶段的序列图如图 5-2 所示。

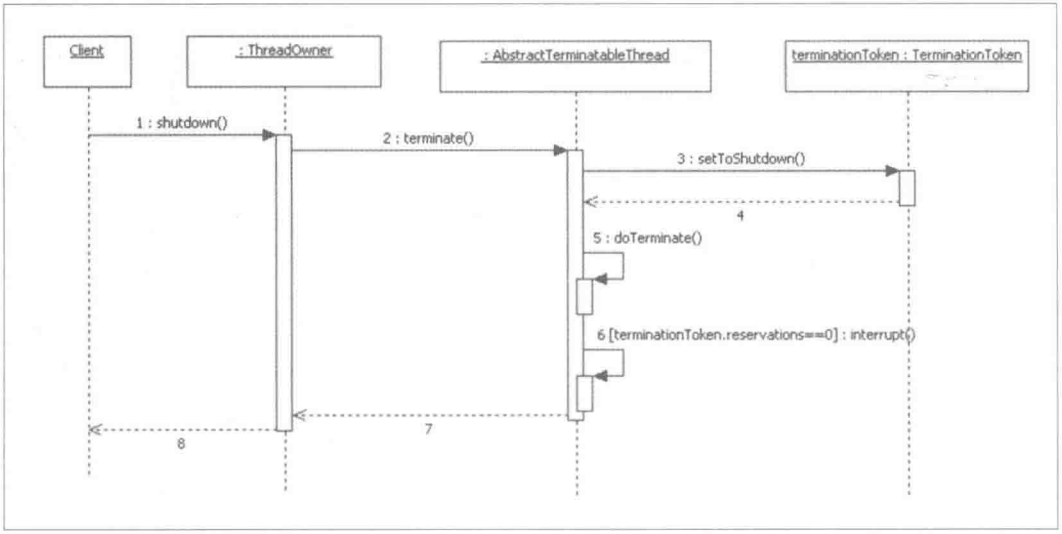


图 5-2. 准备阶段的序列图

第 1 步：客户端代码调用线程拥有者的 shutdown 方法。

第 2 步：shutdown 方法调用目标线程的 terminate 方法。

第 3、4 步：terminate 方法将 terminationToken 的 toShutdown 标志设置为 true。

第 5 步：terminate 方法调用由 AbstractTerminatableThread 子类实现的 doTerminate 方法，使得子类可以为停止目标线程做一些其他必要的操作。

第 6 步：若 terminationToken 的 reservations 属性值为 0，则表示目标线程没有未处理完的任务或者 ThreadOwner 在停止线程时不关心其是否有未处理的任务。此时，terminate 方法会调用目标线程的 interrupt 方法。

第 7 步：terminate 方法调用结束。

第 8 步：shutdown 调用返回，此时目标线程可能还仍然在运行。

执行阶段由目标线程的 run 方法去检查 terminationToken 的 toShutdown 属性、reservations 属性的值，并捕获由 interrupt 方法调用抛出的相关异常以决定是否停止线程。在线程停止前由 AbstractTerminatableThread 子类实现的 doCleanup 方法会被调用。

5.3 Two-phase Termination 模式实战案例解析

某系统的告警功能被封装在一个模块中。告警模块的入口类是 AlarmMgr。其他模块（业务模块）需要发送告警信息时只需要调用 AlarmMgr 的 sendAlarm 方法即可。该方法将告警信息缓存入队列，由专门的告警发送线程负责调用 AlarmAgent 的相关方法发送告警。AlarmAgent 类负责与告警服务器对接，它通过网络连接将告警信息发送至告警服务器。

告警发送线程是一个用户线程（User Thread），因此在系统的停止过程中，该线程若未停止则会阻止 JVM 正常关闭。所以，在系统停止过程中我们必须主动去停止告警发送线程，而非依赖 JVM。为了能够尽可能快地以优雅的方式将告警发送线程停止，我们需要处理以下两个问题。

1．当告警缓存队列非空时，需要将队列中已有的告警信息发送至告警服务器。

2．由于缓存告警信息的队列是一个阻塞队列（ArrayBlockingQueue），在该队列为空的情况下，告警发送线程会一直处于等待状态。这会导致其无法响应我们关闭线程的请求。

上述问题可以通过使用 Two-phase Termination 模式来解决。

AlarmMgr 相当于图 5-1 中的 ThreadOwner 参与者实例，它是告警发送线程（对应实例变量 alarmSendingThread）的拥有者。系统停止过程中调用其 shutdown 方法（AlarmMgr.getInstance().shutdown()）即可请求告警发送线程停止。其代码如清单 5-1 所示。

清单 5-1. AlarmMgr 类源码

```
/**
 * 告警功能入口类
 * 模式角色：Two-phaseTermination.ThreadOwner
 */
public class AlarmMgr {
    // 保存 AlarmMgr 类的唯一实例
    private static final AlarmMgr INSTANCE = new AlarmMgr();

    private volatile boolean shutdownRequested = false;

    //告警发送线程
    private final AlarmSendingThread alarmSendingThread;

    //私有构造器
    private AlarmMgr() {
        alarmSendingThread = new AlarmSendingThread();

    }

    //返回类 AlarmMgr 的唯一实例
```

```
    public static AlarmMgr getInstance() {
        return INSTANCE;
    /**
     * 发送告警
     * @param type 告警类型
     * @param id 告警编号
     * @param extraInfo 告警参数
     * @return 由type+id+extraInfo唯一确定的告警信息被提交的次数。-1表示告警管理器已被关闭。
     */
    public int sendAlarm(AlarmType type, String id, String extraInfo) {
        Debug.info("Trigger alarm " + type + "," + id + ',' + extraInfo);
        int duplicateSubmissionCount = 0;
        try {
            AlarmInfo alarmInfo = new AlarmInfo(id, type);
            alarmInfo.setExtraInfo(extraInfo);
            duplicateSubmissionCount = alarmSendingThread.sendAlarm(alarmInfo);
        } catch (Throwable t) {
            t.printStackTrace();
        }

        return duplicateSubmissionCount;
    }

    public void init() {
        alarmSendingThread.start();
    }

    public synchronized void shutdown() {
        if (shutdownRequested) {
            throw new IllegalStateException("shutdown already requested!");
        }

        alarmSendingThread.terminate();
        shutdownRequested = true;
    }

}
```

告警发送线程类 AlarmSendingThread 的源码，如清单 5-2 所示。

清单 5-2. AlarmSendingThread 类源码

```
//模式角色：Two-phaseTermination.ConcreteTerminatableThread
public class AlarmSendingThread extends AbstractTerminatableThread {

    private final AlarmAgent alarmAgent = new AlarmAgent();

    //告警队列
    private final BlockingQueue<AlarmInfo> alarmQueue;
    private final ConcurrentMap<String, AtomicInteger> submittedAlarmRegistry;

    public AlarmSending
        alarmQueue = new ArThread() {
            rayBlockingQueue<AlarmInfo>(100);

        submittedAlarmRegistry = new ConcurrentHashMap<String, AtomicInteger>();
```

```
        alarmAgent.init();
    }

    @Override
    protected void doRun() throws Exception {
        AlarmInfo alarm;
        alarm = alarmQueue.take();
        terminationToken.reservations.decrementAndGet();

        try {
            //将告警信息发送至告警服务器
            alarmAgent.sendAlarm(alarm);
        } catch (Exception e) {
            e.printStackTrace();
        }

        /*
         * 处理恢复告警：将相应的故障告警从注册表中删除，使得相应故障恢复后若再次出现相同故障，
         * 该故障信息能够上报到服务器
         */
        if (AlarmType.RESUME == alarm.type) {
            String key = AlarmType.FAULT.toString() + ':' + alarm.getId() + '@'
                + alarm.getExtraInfo();
            submittedAlarmRegistry.remove(key);

            key = AlarmType.RESUME.toString() + ':' + alarm.getId() + '@'
                + alarm.getExtraInfo();
            submittedAlarmRegistry.remove(key);
        }

    }

    public int sendAlarm(final AlarmInfo alarmInfo) {
        AlarmType type = alarmInfo.type;
        String id = alarmInfo.getId();
        String extraInfo = alarmInfo.getExtraInfo();

        if (terminationToken.isToShutdown()) {
            // 记录告警
            System.err.println("rejected alarm:" + id + "," + extraInfo);
            return -1;
        }

        int duplicateSubmissionCount = 0;
        try {

            AtomicInteger prevSubmittedCounter;

            prevSubmittedCounter = submittedAlarmRegistry.putIfAbsent(
              type.toString()+ ':' + id + '@' + extraInfo, new AtomicInteger(0));
            if (null == prevSubmittedCounter) {
                terminationToken.reservations.incrementAndGet();
                alarmQueue.put(alarmInfo);
            } else {

                // 故障未恢复，不用重复发送告警信息给服务器，故仅增加计数
                duplicateSubmissionCount = prevSubmittedCounter.incrementAndGet();
            }
```

```
        } catch (Throwable t) {
            t.printStackTrace();
        }

        return duplicateSubmissionCount;
    }

    @Override
    protected void doCleanup(Exception exp) {
        if (null != exp && !(exp instanceof InterruptedException)) {
            exp.printStackTrace();
        }
        alarmAgent.disconnect();
    }

}
```

从上面的代码可以看出，AlarmSendingThread 每接受一个告警信息放入缓存队列便将 terminationToken 的 reservations 值增加 1，而每发送一个告警到告警服务器则将 terminationToken 的 reservations 值减少 1。这为我们可以在停止告警发送线程前确保队列中现有的告警信息会被处理完毕提供了线索：AbstractTerminatableThread 的 run 方法会根据 terminationToken 的 reservations 是否为 0 来判断待停止的线程已无未处理的任务，或者无须关心其是否有待处理的任务。

AbstractTerminatableThread 的源码见清单 5-3。

清单 5-3. AbstractTerminatableThread 类源码

```
/**
 * 可停止的抽象线程。
 *
 * 模式角色：Two-phaseTermination.AbstractTerminatableThread
 *
 * @author Viscent Huang
 */
public abstract class AbstractTerminatableThread extends Thread implements
    Terminatable {

    // 模式角色：Two-phaseTermination.TerminationToken
    public final TerminationToken terminationToken;

    public AbstractTerminatableThread() {
        this(new TerminationToken());
    }

    /**
     *
     * @param terminationToken
     *          线程间共享的线程终止标志实例
     */
    public AbstractTerminatableThread(TerminationToken terminationToken) {
        super();
        this.terminationToken = terminationToken;
        terminationToken.register(this);
```

```
}

/**
 * 留给子类实现其线程处理逻辑。
 *
 * @throws Exception
 */
protected abstract void doRun() throws Exception;

/**
 * 留给子类实现。用于实现线程停止后的一些清理动作。
 *
 * @param cause
 */
protected void doCleanup(Exception cause) {
    // 什么也不做
}

/**
 * 留给子类实现。用于执行线程停止所需的操作。
 */
protected void doTerminiate() {
    // 什么也不做
}

@Override
public void run() {
    Exception ex = null;
    try {
        for (;;) {

            // 在执行线程的处理逻辑前先判断线程停止的标志。
            if (terminationToken.isToShutdown()
                && terminationToken.reservations.get() <= 0) {
                break;
            }
            doRun();
        }

    } catch (Exception e) {
        // 使得线程能够响应 interrupt 调用而退出
        ex = e;
    } finally {
        try {
            doCleanup(ex);
        } finally {
            terminationToken.notifyThreadTermination(this);
        }
    }
}

@Override
public void interrupt() {
    terminate();
}

/*
 * 请求停止线程。
```

```
     *
     * @see io.github.viscent.mtpattern.tpt.Terminatable#terminate()
     */
    @Override
    public void terminate() {
        terminationToken.setToShutdown(true);
        try {
            doTerminiate();
        } finally {

            // 若无待处理的任务，则试图强制终止线程
            if (terminationToken.reservations.get() <= 0) {
                super.interrupt();
            }
        }
    }

    public void terminate(boolean waitUtilThreadTerminated) {
        terminate();
        if (waitUtilThreadTerminated) {
            try {
                this.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

AbstractTerminatableThread 是一个可复用的 Terminatable 参与者实例。其 terminate 方法完成了线程停止的准备阶段。该方法首先将 terminationToken 的 toShutdown 属性设置为 true，指示目标线程可以准备停止了。但是，此时目标线程可能处于一些阻塞（Blocking）方法的调用，如调用 Object.sleep、InputStream.read 等，无法检测该变量的值。调用目标线程的 interrupt 方法可以使一些阻塞方法（参见表 5-1）抛出异常从而使目标线程停止。但也有些阻塞方法如 InputStream.read 并不对 interrupt 方法调用做出响应，此时需要由 AbstractTerminatableThread 的子类实现 doTerminate 方法，在该方法中实现一些关闭目标线程所需的额外操作。例如，在 Socket 同步 I/O 中通过关闭 socket 使得使用该 socket 的线程若处于 I/O 等待会抛出 SocketException。因此，terminate 方法下一步调用 doTerminate 方法。接着，若 terminationToken.reservations 的值为非正数（表示目标线程无待处理任务，或者我们不关心其是否有待处理任务），则 terminate 方法会调用目标线程的 interrupt 方法，强制目标线程的阻塞方法中断，从而强制终止目标线程。

执行阶段在 AbstractTerminatableThread 的 run 方法中完成。该方法通过对 TerminationToken 的 toShutdown 属性和 reservations 属性的判断或者通过捕获由 interrupt 方法调用而抛出的异常来终止线程，并在线程终止前调用由 AbstractTerminatableThread 子类实现的 doCleanup 方法用于执行一些清理动作。

在执行阶段，由于 AbstractTerminatableThread.run 方法每次执行线程处理逻辑（通过调用 doRun 方法实现）前都先判断下 toShutdown 属性和 reservations 属性的值，在目标线程处理完待处理的任务后（此时 reservations 属性的值为非正数）目标线程 run 方法也就退出了 while 循环。因此，线程的处理逻辑方法将不再被调用，从而使本案例在不使用 Two-phase Termination 模式的情况下停止目标线程存在的两个问题得以解决（目标线程停止前可以保证处理完待处理的任务——发送队列中现有的告警信息到服务器）和规避（目标线程发送完队列中现有的告警信息后，doRun 方法不再被调用，从而避免了队列为空时 BlockingQueue.take 调用导致的阻塞）。

由上可知，准备阶段、执行阶段需要通过 TerminationToken 作为“中介”来协调二者的动作。TerminationToken 的源码如清单 5-4 所示。

清单 5-4. TerminationToken 类源码

```
/**
 * 线程停止标志。
 *
 * @author Viscent Huang
 *
 */
public class TerminationToken {

    // 使用 volatile 修饰，以保证无须显式锁的情况下该变量的内存可见性
    protected volatile boolean toShutdown = false;
    public final AtomicInteger reservations = new AtomicInteger(0);

    /*
     * 在多个可停止线程实例共享一个 TerminationToken 实例的情况下，该队列用于记录那些共享
     * TerminationToken 实例的可停止线程，以便尽可能减少锁的使用的情况下，实现这些线程的停止。
     */
    private final Queue<WeakReference<Terminatable>> coordinatedThreads;

    public TerminationToken() {
        coordinatedThreads = new ConcurrentLinkedQueue<WeakReference<Terminatable>> ();
    }

    public boolean isToShutdown() {
        return toShutdown;
    }

    protected void setToShutdown(boolean toShutdown) {
        this.toShutdown = true;
    }

    protected void register(Terminatable thread) {
        coordinatedThreads.add(new WeakReference<Terminatable>(thread));
    }

    /**
     * 通知 TerminationToken 实例：共享该实例的所有可停止线程中的一个线程停止了，
     * 以便其停止其他未被停止的线程。
```

```
     * @param thread
     *          已停止的线程
     */
    protected void notifyThreadTermination(Terminatable thread) {
        WeakReference<Terminatable> wrThread;
        Terminatable otherThread;
        while (null != (wrThread = coordinatedThreads.poll())) {
            otherThread = wrThread.get();
            if (null != otherThread && otherThread != thread) {
                otherThread.terminate();
            }
        }
    }

}
```

5.4 Two-phase Termination 模式的评价与实现考量

Two-phase Termination 模式使得我们可以对各种形式的目标线程进行优雅的停止。如目标线程调用了能够对 interrupt 方法调用做出响应的阻塞方法、目标线程调用了不能对 interrupt 方法调用做出响应的阻塞方法、目标线程作为消费者处理其他线程生产的“产品”在其停止前需要处理完现有“产品”等。Two-phase Termination 模式实现的线程停止可能出现延迟，即客户端代码调用完 ThreadOwner.shutdown 后，该线程可能仍在运行。

本章案例展示了一个可复用的 Two-phase Termination 模式实现代码。读者若要加深对该模式的理解或者自行实现该模式，需要注意以下几个问题。

5.4.1 线程停止标志

本章案例使用了 TerminationToken 作为目标线程可以准备停止的标志。从清单 5-4 的代码我们可以看到，TerminationToken 使用了 toShutdown 这个 boolean 变量作为主要的停止标志，而非使用 Thread.isInterrupted()。这是因为，调用目标线程的 interrupt 方法无法保证目标线程的 isInterrupted() 方法返回值为 true；目标线程可能调用一些代码，它们捕获 InterruptedException 后没有通过调用 Thread.currentThread().interrupt() 保留线程中断状态。另外，toShutdown 这个变量为了保证内存可见性而又能避免使用显式锁的开销，采用了 volatile 修饰。这点也很重要，笔者曾经见过一些采用 boolean 变量作为线程停止标志的代码，只是这些变量没有用 volatile 修饰，对其访问也没有加锁，这就可能无法停止目标线程。

另外，某些场景下多个可停止线程实例可能需要共用一个线程停止标志。例如，多个可停止线程实例“消耗”同一个队列中的数据。当该队列为空且不再有新的数据入队列的时候，“消耗”该队列数据的所有可停止线程都应该被停掉。AbstractTerminatableThread 类（源码见清

单 5-3）的构造器支持传入一个 TerminationToken 实例就是为了支持这种场景。

5.4.2 生产者-消费者问题中的线程停止

在多线程编程中，许多问题和一些多线程编程模式都可以看作生产者-消费者问题。停止处于生产者-消费者问题中的线程，需要考虑更多的问题：需要注意线程的停止顺序。如果消费者线程比生产者线程先停止则会导致生产者生产的新“产品”无法被处理，而如果先停止生产者线程又可能使消费者线程处于空等待（如生产者、消费者采用阻塞队列中转“产品”）。并且，停止消费者线程前是否考虑要等待其处理完所有待处理的任务或者将这些任务做个备份也是个问题。本章案例部分地展示生产者-消费者问题中线程停止的处理，其核心就是通过使用 TerminationToken 的 reservations 属性：生产者每“生产”一个产品，Two-phase Termination 模式的客户端代码要使 reservations 属性值增加 1（即调用 terminationToken.reservations.incrementAndGet()）；消费者线程每处理一个产品，该线程的线程处理逻辑方法 doRun 要使 reservations 属性值减少 1（即调用 terminationToken.reservations.decrementAndGet()）。当然，在停止消费者线程时如果我们不关心其待处理的任务，Two-phase Termination 模式的客户端代码可以忽略对 reservations 变量的操作。清单 5-5 展示了一个完整的停止生产者-消费者问题中的线程的例子。

清单 5-5. 停止生产者-消费者问题中的线程的例子

```
public class SomeService {
    private final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(
        100);

    private final Producer producer = new Producer();
    private final Consumer consumer = new Consumer();

    private class Producer extends AbstractTerminatableThread {
        private int i = 0;

        @Override
        protected void doRun() throws Exception {
            queue.put(String.valueOf(i++));
            consumer.terminationToken.reservations.incrementAndGet();
        }

    };

    private class Consumer extends AbstractTerminatableThread {

        @Override
        protected void doRun() throws Exception {
            String product = queue.take();

            System.out.println("Processing product:" + product);

            // 模拟执行真正操作的时间消耗
```

```
            try {
                Thread.sleep(new Random().nextInt(100));
            } catch (InterruptedException e) {
                ;
            } finally {
                terminationToken.reservations.decrementAndGet();
            }

        }

    }

    public void shutdown() {

        //生产者线程停止后再停止消费者线程
        producer.terminate(true);
        consumer.terminate();
    }

    public void init() {
        producer.start();
        consumer.start();
    }

    public static void main(String[] args) throws InterruptedException {
        SomeService ss = new SomeService();
        ss.init();
        Thread.sleep(500);
        ss.shutdown();
    }
}
```

5.4.3 隐藏而非暴露可停止的线程

为了保证可停止的线程不被其他代码误停止，一般我们将可停止线程隐藏在线程拥有者背后，而使系统中其他代码无法直接访问该线程，正如本案例代码（见清单 5-1）所展示：AlarmMgr 定义了一个 private 字段 alarmSendingThread 用于引用告警发送线程（可停止的线程），系统中的其他代码只能通过调用 AlarmMgr 的 shutdown 方法来请求该线程停止，而非通过引用该线程对象自身来停止它。

5.5 Two-phase Termination 模式的可复用实现代码

本章案例代码（见清单 5-3、清单 5-4）所实现的 Two-phase Termination 模式的几个参与者 AbstractTerminatableThread 和 TerminationToken 都是可复用的。在此基础上，应用代码只需要在定义 AbstractTerminatableThread 的子类（或匿名类）时实现 doRun 方法，在该方法中实现线程的处理逻辑。另外，应用代码如果需要在目标线程处理完待处理的任务后再停止，则需要注意 TerminationToken 实例的 reservations 属性值的增加和减少。

5.6 Java 标准库实例

类 java.util.concurrent.ThreadPoolExecutor 就使用了 Two-phase Termination 模式来停止其内部维护的工作者线程。当客户端代码调用 ThreadPoolExecutor 实例的 shutdown 方法请求其关闭时，ThreadPoolExecutor 会先将其运行状态设置为 SHUTDOWN。工作者线程的 run 方法会判断其所属的 ThreadPoolExecutor 实例的运行状态。若 ThreadPoolExecutor 实例的运行状态为 SHUTDOWN，则工作者线程会一直取工作队列中的任务进行执行，直到工作队列为空时该工作者线程就停止了。可见，ThreadPoolExecutor 实例的停止过程也是分为准备阶段（设置其运行状态为 SHUTDOWN）和执行阶段（工作者队列取空工作队列中的任务，然后终止线程）。

5.7 相关模式

Two-phase Termination 模式是一个应用比较广泛的基础多线程设计模式。凡是涉及应用自身实现线程的代码，都可能需要使用该模式。

5.7.1 Producer-Consumer 模式（第 7 章）

Producer-Consumer 模式中，生产者线程、消费者线程的停止可能需要使用 Two-phase Termination 模式。

5.7.2 Master-Slave 模式（第 12 章）

Master-Slave 模式中，工作者线程的停止可能需要使用 Two-phase Termination 模式。

5.8 参考资源

1．Brian Göetz et al.Java Concurrency In Practice.Addison Wesley,2006.

2．Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

3．类 ThreadPoolExecutor 源码. http://www.docjar.com/html/api/java/util/concurrent/ ThreadPoolExecutor.java.html.

第 6 章

Promise（承诺）模式

6.1 Promise 模式简介

Promise 模式是一种异步编程模式[1]。它使得我们可以先开始一个任务的执行，并得到一个用于获取该任务执行结果的凭据对象，而不必等待该任务执行完毕就可以继续执行其他操作。等到我们需要该任务的执行结果时，再调用凭据对象的相关方法来获取。这样就避免了不必要的等待，增加了系统的并发性。这好比我们去小吃店，同时点了鸭血粉丝汤和生煎包。当我们点餐付完款后，我们拿到手的其实只是一张可借以换取相应食品的收银小票（凭据对象）而已，而不是对应的实物。由于鸭血粉丝汤可以较快制作好，故我们可以凭收银小票即刻兑换到。而生煎包的制作则比较耗时，因此我们可以先吃拿到手的鸭血粉丝汤，而不必饿着肚子等生煎包出炉再一起吃。等到我们把鸭血粉丝汤吃得差不多的时候，生煎包可能也出炉了，这时我们再凭收银小票去换取生煎包，如图 6-1 所示。

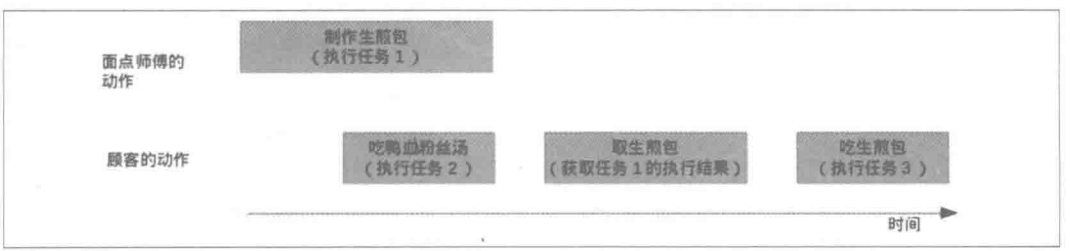


图 6-1. Promise 模式的日常生活例子

1 其实该模式也支持同步方式的编程，下文会提到这点。

6.2 Promise 模式的架构

Promise 模式中，客户端代码调用某个异步方法所得到的返回值仅是一个凭据对象（该对象被称为 Promise，意为“承诺”）。凭借该对象，客户端代码可以获取异步方法相应的真正任务的执行结果。为了讨论方便，下文我们称异步方法对应的真正的任务为异步任务。

Promise 模式的主要参与者有以下几种。其类图如图 6-2 所示。

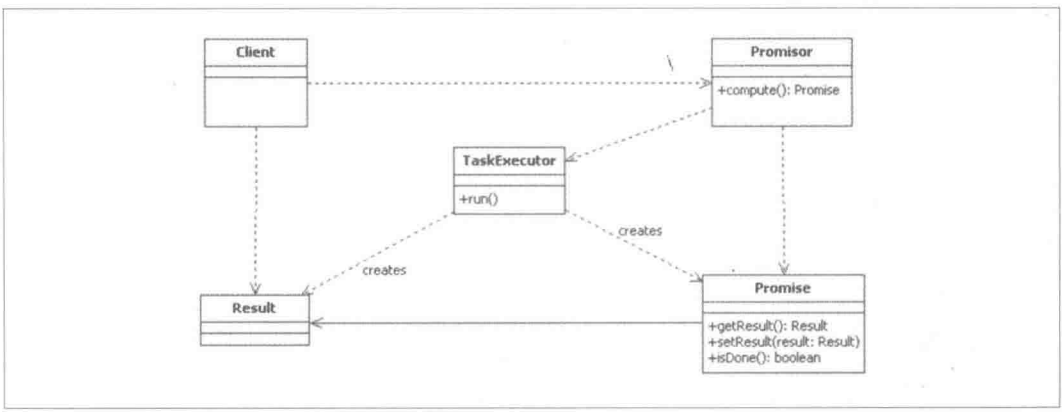


图 6-2. Promise 模式的类图

- **Promisor**：负责对外暴露可以返回 Promise 对象的异步方法，并启动异步任务的执行。其主要方法及职责如下。
  - **compute**：启动异步任务的执行，并返回用于获取异步任务执行结果的凭据对象。
- **Promise**：包装异步任务处理结果的凭据对象。负责检测异步任务是否处理完毕、返回和存储异步任务处理结果。其主要方法及职责如下。
  - **getResult**：获取与其所属 Promise 实例关联的异步任务的执行结果。
  - **setResult**：设置与其所属 Promise 实例关联的异步任务的执行结果。
  - **isDone**：检测与其所属 Promise 实例关联的异步任务是否执行完毕。
- **Result**：负责表示异步任务处理结果。具体类型由应用决定。
- **TaskExecutor**：负责真正执行异步任务所代表的计算，并将其计算结果设置到相应的 Promise 实例。其主要方法及职责如下
  - **run**：执行异步任务所代表的计算。

客户端代码获取异步任务处理结果的过程如图 6-3 所示的序列图。

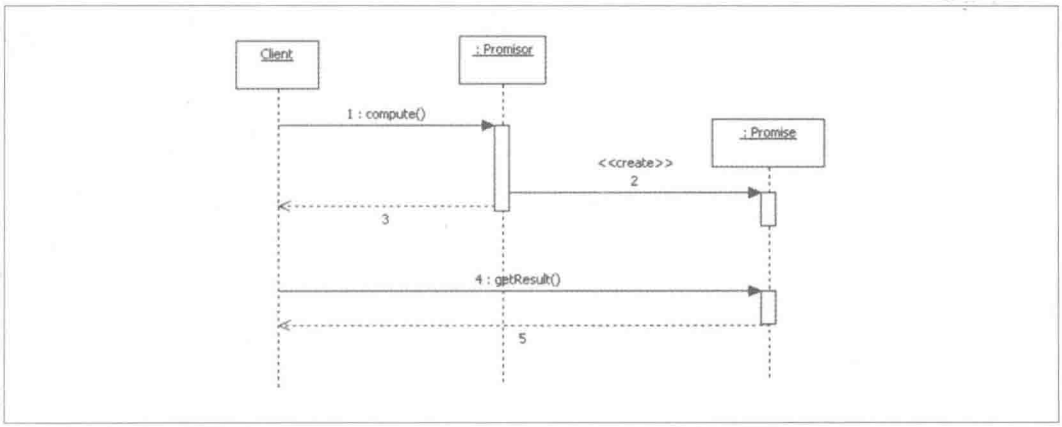


图 6-3. 获取异步任务的处理结果

第 1 步：客户端代码调用 Promisor 的异步方法 compute。

第 2、3 步：compute 方法创建 Promise 实例作为该方法的返回值，并返回。

第 4 步：客户端代码调用其所得到的 Promise 对象的 getResult 方法来获取异步任务处理结果。如果此时异步任务执行尚未完成，则 getResult 方法会阻塞（即调用方代码的运行线程暂时处于阻塞状态）。

异步任务的真正执行以及其处理结果的设置如图 6-4 所示的序列图。

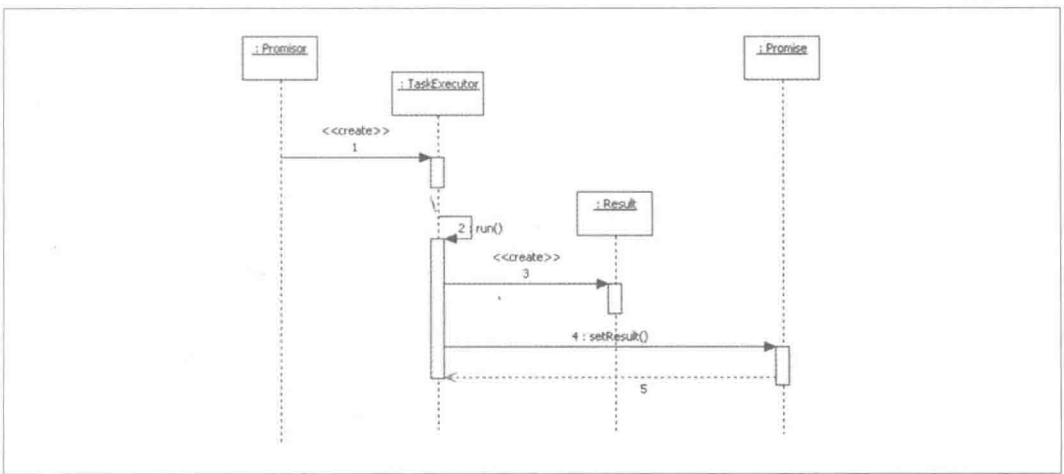


图 6-4. 设置异步任务的处理结果

第 1 步：Promisor 的异步方法 compute 创建 TaskExecutor 实例。

第 2 步：TaskExecutor 的 run 方法被执行（可以由专门的线程或者线程池[2]来调用 run 方法）。

第 3 步：run 方法创建表示其执行结果的 Result 实例。

第 4、5 步：run 方法将其处理结果设置到相应的 Promise 实例上。

6.3 Promise 模式实战案例解析

某系统的一个数据同步模块需要将一批本地文件上传到指定的目标 FTP 服务器上。这些文件是根据页面中的输入条件查询数据库的相应记录生成的。在将文件上传到目标服务器之前，需要对 FTP 客户端实例进行初始化（包括与对端服务器建立网络连接、向服务器发送登录用户和向服务器发送登录密码）。而 FTP 客户端实例初始化这个操作比较耗时间，我们希望它尽可能地在本地文件上传之前准备就绪。因此我们可以引入异步编程，使得 FTP 客户端实例初始化和本地文件上传这两个任务能够并发执行，减少不必要的等待。另一方面，我们不希望这种异步编程增加了代码编写的复杂性。这时，Promise 模式就可以派上用场了：先开始 FTP 客户端实例的初始化，并得到一个获取 FTP 客户端实例的凭据对象。在不必等待 FTP 客户端实例初始化完毕的情况下，每生成一个本地文件，就通过凭据对象获取 FTP 客户端实例，再通过该 FTP 客户端实例将文件上传到目标服务器上。代码如清单 6-1 所示[3]。

清单 6-1. 数据同步模块的入口类

```
public class DataSyncTask implements Runnable {
    private final Map<String, String> taskParameters;

    public DataSyncTask(Map<String, String> taskParameters) {
        this.taskParameters = taskParameters;
    }

    @Override
    public void run() {
        String ftpServer = taskParameters.get("server");
        String ftpUserName = taskParameters.get("userName");
        String password = taskParameters.get("password");

        //先开始初始化 FTP 客户端实例
        Future<FTPClientUtil> ftpClientUtilPromise = FTPClientUtil.newInstance(
            ftpServer, ftpUserName, password);
```

2 在第 9 章中我们会提到线程池（Thread Pool）模式。这里，读者可以简单地将线程池理解成 Java 中的类 java.util.concurrent.ThreadPoolExecutor。

3 本案例的实际代码与该清单所示的并不完全一致，这是为了讨论方便。在第 13 章我们会展示该案例的实际代码。

```
        //查询数据库生成本地文件
        generateFilesFromDB();

        FTPClientUtil ftpClientUtil = null;
        try {
            // 获取初始化完毕的 FTP 客户端实例
            ftpClientUtil = ftpClientUtilPromise.get();
        } catch (InterruptedException e) {
            ;
        } catch (ExecutionException e) {
            throw new RuntimeException(e);
        }

        // 上传文件
        uploadFiles(ftpClientUtil);

        //省略其他代码
    }

    private void generateFilesFromDB() {
        // 省略其他代码
    }

    private void uploadFiles(FTPClientUtil ftpClientUtil) {
        Set<File> files = retrieveGeneratedFiles();
        for (File file : files) {
            try {
                ftpClientUtil.upload(file);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

    }

    private Set<File> retrieveGeneratedFiles() {
        Set<File> files = new HashSet<File>();
        // 省略其他代码
        return files;
    }

}
```

从清单 6-1 的代码中可以看出，DataSyncTask 类的 run 方法先开始 FTP 客户端实例的初始化，并得到获取相应 FTP 客户端实例的凭据对象 ftpClientUtilPromise。接着，它直接开始查询数据库并生成本地文件。而此时，FTP 客户端实例的初始化可能尚未完成。在本地文件生成之后，run 方法通过调用 ftpClientUtilPromise 的 get 方法来获取相应的 FTP 客户端实例。此时，如果相应的 FTP 客户端实例的初始化仍未完成，则该调用会阻塞，直到相应的 FTP 客户端实例的初始化完成或者失败。run 方法获取到 FTP 客户端实例后，调用其 upload 方法将文件上传到指定的 FTP 服务器。

清单 6-1 代码所引用的 FTP 客户端工具类 FTPClientUtil 的代码如清单 6-2 所示。

清单 6-2. FTP 客户端工具类源码

```
//模式角色：Promise.Promisor、Promise.Result
public class FTPClientUtil {
    private final FTPClient ftp = new FTPClient();

    private final Map<String, Boolean> dirCreateMap = new HashMap<String, Boolean>();

    private FTPClientUtil() {

    }

    //模式角色：Promise.Promisor.compute
    public static Future<FTPClientUtil> newInstance(final String ftpServer,
        final String userName, final String password) {

        Callable<FTPClientUtil> callable = new Callable<FTPClientUtil>() {

            @Override
            public FTPClientUtil call() throws Exception {
                FTPClientUtil self = new FTPClientUtil();
                self.init(ftpServer, userName, password);
                return self;
            }

        };

        //task 相当于模式角色：Promise.Promise
        final FutureTask<FTPClientUtil> task = new FutureTask<FTPClientUtil>(
            callable);

        /*
        下面这行代码与本案例的实际代码并不一致，这是为了讨论方便。
        下面新建的线程相当于模式角色：Promise.TaskExecutor
        */
        new Thread(task).start();
        return task;
    }

    private void init(String ftpServer, String userName, String password)
        throws Exception {

        FTPClientConfig config = new FTPClientConfig();
        ftp.configure(config);

        int reply;
        ftp.connect(ftpServer);

        System.out.print(ftp.getReplyString());

        reply = ftp.getReplyCode();

        if (!FTPReply.isPositiveCompletion(reply)) {
            ftp.disconnect();
            throw new RuntimeException("FTP server refused connection.");
        }
        boolean isOK = ftp.login(userName, password);
        if (isOK) {
```

```
            System.out.println(ftp.getReplyString());

        } else {
            throw new RuntimeException("Failed to login." + ftp.getReplyString());
        }

        reply = ftp.cwd("~/subspsync");
        if (!FTPReply.isPositiveCompletion(reply)) {
            ftp.disconnect();
            throw new RuntimeException("Failed to change working directory.reply:"
                + reply);
        } else {

            System.out.println(ftp.getReplyString());
        }

        ftp.setFileType(FTP.ASCII_FILE_TYPE);

    }

    public void upload(File file) throws Exception {
        InputStream dataIn = new BufferedInputStream(new FileInputStream(file),
            1024 * 8);
        boolean isOK;
        String dirName = file.getParentFile().getName();
        String fileName = dirName + '/' + file.getName();
        ByteArrayInputStream checkFileInputStream = new ByteArrayInputStream(
            "".getBytes());

        try {
            if (!dirCreateMap.containsKey(dirName)) {
                ftp.makeDirectory(dirName);
                dirCreateMap.put(dirName, null);
            }

            try {
                isOK = ftp.storeFile(fileName, dataIn);
            } catch (IOException e) {
                throw new RuntimeException("Failed to upload " + file, e);
            }
            if (isOK) {
                ftp.storeFile(fileName + ".c", checkFileInputStream);

            } else {

            throw new RuntimeException("Failed to upload " + file + ",reply:" +
                    "," + ftp.getReplyString());
            }
        } finally {
            dataIn.close();
        }

    }

    public void disconnect() {
        if (ftp.isConnected()) {
            try {
                ftp.disconnect();
            } catch (IOException ioe) {
```

```
                    // 什么也不做
                }
            }
        }
    }
```

FTPClientUtil 类封装了 FTP 客户端，其构造方法是 private 修饰的，因此其他类无法通过 new 来生成相应的实例，而是通过其静态方法 newInstance 来获得实例。不过 newInstance 方法的返回值并不是一个 FTPClientUtil 实例，而是一个可以获取 FTPClientUtil 实例的凭据对象 java.util.concurrent.Future（具体说是 java.util.concurrent.FutureTask，它实现了 java.util.concurrent.Future 接口）实例。因此，FTPClientUtil 既相当于 Promise 模式中的 Promisor 参与者实例，又相当于 Result 参与者实例。而 newInstance 方法的返回值 java.util.concurrent.FutureTask 实例既相当于 Promise 参与者实例，又相当于 TaskExecutor 参与者实例：newInstance 方法的返回值 java.util.concurrent.FutureTask 实例不仅负责该方法真正处理结果（初始化完毕的 FTP 客户端实例）的存储和获取，还负责执行异步任务（调用 FTPClientUtil 实例的 init 方法），并设置任务的处理结果。

从如清单 6-2 所示的 Promise 客户端代码（DataSyncTask 类的 run 方法）来看，使用 Promise 模式的异步编程并没有本质上增加编程的复杂性：客户端代码的编写方式与同步编程并没有太大差别，唯一一点差别就是获取 FTP 客户端实例的时候多了一步对 java.util.concurrent.FutureTask 实例的 get 方法的调用。

6.4 Promise 模式的评价与实现考量

Promise 模式既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性：有关异步编程的细节，如创建新的线程或者提交任务到线程池执行等细节，都被封装在 Promisor 参与者实例中，而 Promise 的客户端代码则无须关心这些细节，其编码方式与同步编程并无本质上差别。这点正如清单 6-1 代码所展示的，客户端代码仅仅需要调用 FTPClientUtil 的 newInstance 静态方法，再调用其返回值的 get 方法，即可获得一个初始化完毕的 FTP 客户端实例。这本质上还是同步编程。当然，客户端代码也不能完全无视 Promise 模式的异步编程这一特性：为了减少客户端代码在调用 Promise 的 getResult 方法时出现阻塞的可能，客户端代码应该尽可能早地调用 Promisor 的异步方法，并尽可能晚地调用 Promise 的 getResult 方法。这当中间隔的时间可以由客户端代码用来执行其他操作，同时这段时间可以给 TaskExecutor 用于执行异步任务。

Promise 模式一定程度上屏蔽了异步、同步编程的差异。前文我们一直说 Promisor 对外暴露的 compute 方法是个异步方法。事实上，如果 compute 方法是一个同步方法，那么 Promise

模式的客户端代码的编写方式也是一样的。也就是说，无论 compute 方法是一个同步方法还是异步方法，Promise 客户端代码的编写方式都是一样的。例如，本章案例中 FTPClientUtil 的 newInstance 方法如果改成同步方法，我们只需要将其方法体中的语句 new Thread(task).start();改为 task.run();即可。而该案例中的其他代码无须更改。这就在一定程度上屏蔽了同步、异步编程的差异。而这可以给代码调试或者问题定位带来一定的便利。比如，我们的本意是要将 compute 方法设计成一个异步方法，但在调试代码的时候发现结果不对，那么我们可以尝试临时将其改为同步方法。若此时原先存在的问题不再出现，则说明问题是 compute 方法被编码为异步方法后所产生的多线程并发访问控制不正确导致的。

6.4.1 异步方法的异常处理

如果 Promisor 的 compute 方法是个异步方法，那么客户端代码在调用完该方法后异步任务可能尚未开始执行。另外，异步任务运行在自己的线程中，而不是 compute 方法的调用方线程中。因此，异步任务执行过程中产生的异常无法在 compute 方法中抛出。为了让 Promise 模式的客户端代码能够捕获到异步任务执行过程中出现的异常，一个可行的办法是让 TaskExecutor 在执行任务捕获到异常后，将异常对象“记录”到 Promise 实例的一个专门的实例变量上，然后由 Promise 实例的 getResult 方法对该实例变量进行检查。若该实例变量的值不为 null，则 getResult 方法抛出异常。这样，Promise 模式的客户端代码通过捕获 getResult 方法抛出的异常即可“知道”异步任务执行过程中出现的异常。JDK 中提供的类 java.util.concurrent.FutureTask 就是采用这种方法对 compute 异步方法的异常进行处理的。

6.4.2 轮询（Polling）

客户端代码对 Promise 的 getResult 的调用可能由于异步任务尚未执行完毕而阻塞，这实际上也是一种等待。虽然我们可以通过尽可能早地调用 compute 方法并尽可能晚地调用 getResult 方法来减少这种等待的可能性，但是它仍然可能会出现。某些场景下，我们可能根本不希望进行任何等待。此时，我们需要在调用 Promise 的 getResult 方法之前确保异步任务已经执行完毕。因此，Promise 需要暴露一个 isDone 方法用于检测异步任务是否已执行完毕。JDK 提供的类 java.util.concurrent.FutureTask 的 isDone 方法正是出于这种考虑，它允许我们在“适当”的时候才调用 Promise 的 getResult 方法（相当于 FutureTask 的 get 方法）。

6.4.3 异步任务的执行

本章案例中，异步任务的执行我们是通过新建一个线程，由该线程去调用 TaskExecutor 的 run 方法来实现的（见清单 6-2）。这只是为了讨论方便。如果系统中同时存在多个线程调用

Promisor 的异步方法，而每个异步方法都启动了各自的线程去执行异步任务，这可能导致一个 JVM 中启动的线程数量过多，增加了线程调度的负担，从而反倒降低了系统的性能。因此，如果 Promise 模式的客户端并发量比较大，则需要考虑由线程池负责执行 TaskExecutor 的 run 方法来实现异步任务的执行。例如，如清单 6-2 所示的异步任务如果改用线程池去执行，我们只需要将代码改为类似如清单 6-3 所示的代码即可。

清单 6-3. 用线程池执行异步任务

```
public class FTPClientUtil {
    private volatile static ThreadPoolExecutor threadPoolExecutor;

    static {
        threadPoolExecutor = new ThreadPoolExecutor(1,Runtime.getRuntime()
            .availableProcessors() * 2,
            60,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(10), new ThreadFactory() {
                public Thread newThread(Runnable r) {
                    Thread t = new Thread(r);
                    t.setDaemon(true);
                    return t;
                }
            }, new ThreadPoolExecutor.CallerRunsPolicy());
    }

    private final FTPClient ftp = new FTPClient();
    private final Map<String, Boolean> dirCreateMap = new HashMap<String, Boolean>();

    //私有构造器
    private FTPClientUtil() {

    }

    public static Future<FTPClientUtil> newInstance(final String ftpServer,
        final String userName, final String password) {

        Callable<FTPClientUtil> callable = new Callable<FTPClientUtil>() {

            @Override
            public FTPClientUtil call() throws Exception {
                FTPClientUtil self = new FTPClientUtil();
                self.init(ftpServer, userName, password);
                return self;
            }

        };
        final FutureTask<FTPClientUtil> task = new FutureTask<FTPClientUtil>(
            callable);

        threadPoolExecutor.execute(task);
        return task;
    }

    private void init(String ftpServer, String userName, String password)
        throws Exception {
```

```
        //省略与清单 6-2 中相同的代码
    }

    public void upload(File file) throws Exception {
        //省略与清单 6-2 中相同的代码
    }

    public void disconnect() {
        //省略与清单 6-2 中相同的代码
    }

}
```

6.5 Promise 模式的可复用实现代码

JDK1.5 开始提供的接口 java.util.concurrent.Future 可以看成是 Promise 模式中 Promise 参与者的抽象，其声明如下：

```
public interface Future<V>
```

该接口的类型参数 V 相当于 Promise 模式中的 Result 参与者。该接口定义的方法及其与 Promise 参与者相关方法之间的对应关系如表 6-1 所示。

表 6-1. 接口 java.util.concurrent.Future 与 Promise 参与者的对应关系

接口 java.util.concurrent.Future 的方法	Promise 参与者的方法	功　能
get()	getResult()	获取异步任务的执行结果
isDone()	isDone()	检查异步任务是否执行完毕

接口 java.util.concurrent.Future 的实现类 java.util.concurrent.FutureTask 可以看作 Promise 模式的 Promise 参与者实例。

如清单 6-2 所示的代码中的异步方法 newInstance 展示了如何使用 java.util.concurrent.FutureTask 来作为 Promise 参与者。

6.6 Java 标准库实例

JAX-WS 2.0 API 中用于支持调用 Web Service 的接口 javax.xml.ws.Dispatch 就使用了 Promise 模式。该接口用于异步调用 Web Service 的方法声明如下：

```
Response<T> invokeAsync(T msg)
```

该方法不等对端服务器给响应就返回了（即实现了异步调用 Web Service），从而避免了 Web

Service 客户端进行不必要的等待。而客户端需要其调用的 Web Service 的响应时，可以调用 invokeAsync 方法的返回值的相关方法来获取。invokeAsync 的返回值类型为 javax.xml.ws.Response，它继承自 java.util.concurrent.Future。因此，javax.xml.ws.Dispatch 相当于 Promise 模式中的 Promisor 参与者实例，其异步方法 invokeAsync(T msg)的返回值相当于 Promise 参与者实例。

6.7 相关模式

6.7.1 Guarded Suspension 模式（第 4 章）

Promise 模式的客户端代码调用 Promise 的 getResult 方法获取异步任务处理结果时，如果异步任务已经执行完毕，则该调用会直接返回。否则，该调用会阻塞直到异步任务处理结束或者出现异常。这种通过线程阻塞而进行的等待可以看作 Guarded Suspension 模式的一个实例。只不过，一般情况下 Promise 参与者我们可以直接使用 JDK 中提供的类 java.util.concurrent.FutureTask 来实现，而无须自行编码。关于 java.util.concurrent.FutureTask 如何实现通过阻塞去等待异步方法执行结束，感兴趣的读者可以去阅读 JDK 标准库的源码。

6.7.2 Active Object 模式（第 8 章）

Active Object 模式可以看成是包含了 Promise 模式的复合模式。其 Proxy 参与者相当于 Promise 模式的 Promisor 参与者。Proxy 参与者的异步方法返回值相当于 Promise 模式的 Promise 参与者实例。Active Object 模式的 Scheduler 参与者相当于 Promise 模式的 TaskExecutor 参与者。

6.7.3 Master-Slave 模式（第 12 章）

Master-Slave 模式中，Slave 参与者返回其对子任务的处理结果可能需要使用 Promise 模式。此时，Slave 参与者相当于 Promise 模式的 Promisor 参与者，其 subService 方法的返回值是一个 Promise 模式的 Promise 参与者实例。

6.7.4 Factory Method 模式[4]

Promise 模式中的 Promisor 参与者可以看成是 Factory Method 模式的一个例子：Promisor 的异

4 Factory Method 模式是 GOF 设计模式中的一个，不在本书的讨论范围中。

步方法可以看成一个工厂方法，该方法的返回值是一个 Promise 实例。

6.8 参考资源

1．Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

2．JDK 标准库源码.http://www.docjar.com/html/api/java/util/concurrent/FutureTask.java.html.

3．JAX-WS 2.0 介绍.http://www.oracle.com/technetwork/articles/javase/jax-ws-2-141894.html.

4．Erich Gamma 等.设计模式：可复用面向对象软件的基础（英文版）.机械工业出版社，2002.

第7章

Producer-Consumer（生产者/消费者）模式

7.1 Producer-Consumer 模式简介

数据的提供方可形象地称为数据的生产者，它“生产”了数据，而数据的加工方则相应地被称为消费者，它“消费”了数据。实际上，生产者“生产”数据的速率和消费者“消费”数据的速率往往是不均衡的，比如数据的“生产”要比其“消费”快。为了避免数据的生产者和消费者中处理速率快的一方需要等待处理速率慢的一方，Producer-Consumer 模式通过在数据的生产者和消费者之间引入一个通道（Channel，暂时可以将其简单地理解为一个队列）对二者进行解耦（Decoupling）：生产者将其“生产”的数据放入通道，消费者从相应通道中取出数据进行“消费”（处理），生产者和消费者各自运行在各自的线程中，从而使双方处理速率互不影响。

Producer-Consumer 模式可以看成是设计模式的设计模式。本书介绍的许多模式都可以看作是 Producer-Consumer 模式的一个实例。

7.2 Producer-Consumer 模式的架构

Producer-Consumer 模式的核心是通过通道对数据（或任务）的生产者和消费者进行解耦，使二者不直接交互，从而使二者的处理速率相对来说互不影响。为讨论方便，以下将生产者往通道中存储的数据（或者任务）称为“产品”。

Producer-Consumer 模式的主要参与者有以下几种。其类图如图 7-1 所示。

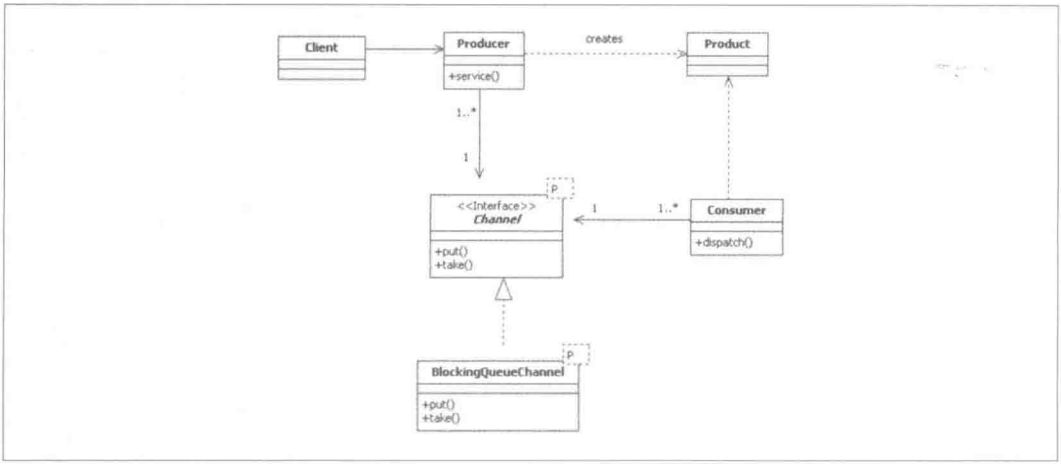


图 7-1. Producer-Consumer 模式的类图

- **Producer**：生产者，负责生成相应的“产品”并将其存入通道。其主要方法及职责如下。
  - **service**：生产者对外暴露的服务方法。
- **Product**：生产者所“生产”的数据或者任务。其具体类型由应用程序决定。
- **Channel**：对通道的抽象。通道充当生产者和消费者之间的缓冲区用于“产品”传递，它可以是有存储容量限制的。该接口的类型参数 P 相当于表示“产品”的类型。该接口的主要方法及职责如下。
  - **put**：将“产品”存入通道。
  - **take**：从通道中取出一个“产品”。
- **BlockingQueueChannel**：基于阻塞队列（Blocking Queue）的 Channel 实现。其主要方法及职责如下。
  - **put**：将“产品”存入通道。当队列满时，该方法会将当前线程挂起直到队列非满。
  - **take**：从通道中取出一个“产品”。当队列为空时，该方法通常会将当前线程挂起直到队列非空。
- **Consumer**：消费者，负责对“产品”进行处理。其主要方法及职责如下。
  - **dispatch**：从通道中获取“产品”，并对其进行处理。

Producer-Consumer 模式的“产品”“生产”序列图如图 7-2 所示。

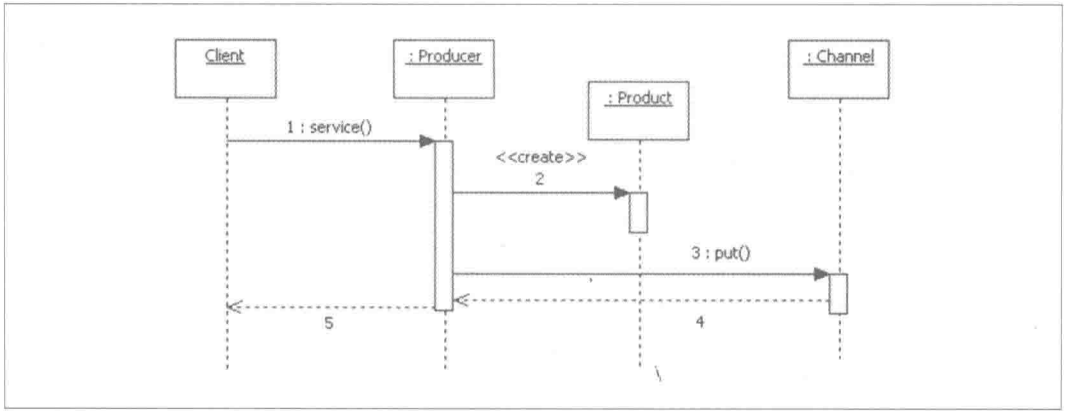


图 7-2. Producer-Consumer 模式的“产品”“生产”序列图

第 1 步：客户端代码调用 Producer 参与者实例的 service 方法。

第 2～4 步：serivce 方法生成相应的“产品”，并调用 Channel 参与者实例的 put 方法将“产品”存入通道。

第 5 步：service 方法返回。

Producer 参与者实例通常运行在客户端线程中。而 Consumer 参与者实例则运行在其专门的工作者线程中。Producer-Consumer 模式的“产品”“消费”序列图如图 7-3 所示。

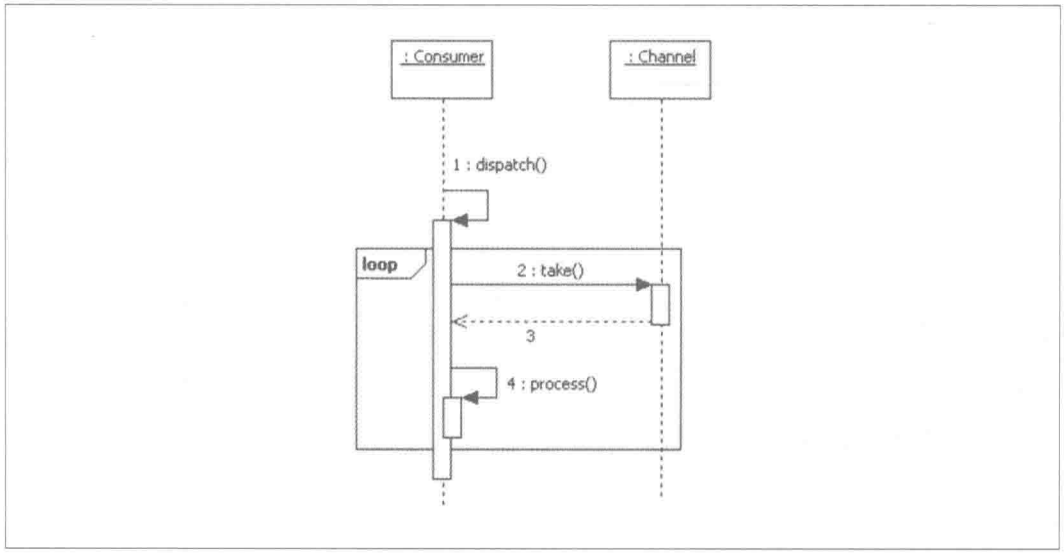


图 7-3. Producer-Consumer 模式的“产品”“消费”序列图

第 1 步：Consumer 参与者实例的工作者线程开始运行，其 dispatch 方法被调用。dispatch 方法会重复执行第 2～4 步。

第 2～4 步：dispatch 方法调用 Channel 参与者实例的 take 方法取出一个“产品”，并对其进行相应的处理。

从上面的描述可以看出，生产者和消费者运行在不同的线程中。生产者“生产”好“产品”后，只需要将其存入通道，而无须等待消费者对其处理完毕。同样，消费者只管从通道中取出“产品”进行处理，而不必等待生产者“生产”“产品”。当然，对于消费者而言，当通道为空的时候，它还是需要等待生产者“生产”新的“产品”的。这样，尽管生产者和消费者的处理速率不同，但是二者并不直接依赖和等待对方从而使二者的处理速率相对来说互不影响对方。

7.3 Producer-Consumer 模式实战案例解析

某内容管理系统需要支持对文档附件中的文件（格式包括 Word、PDF）进行全文检索（Full-text Search）。该系统中，附件会被上传到专用的文件服务器上，对附件进行全文检索的功能模块也是部署在文件服务器上的。因此，与一份文档相关联的附件被上传到文件服务器之后，我们还需要对这些附件生成相应的索引文件以供后面对附件进行全文检索时使用。对附件生成索引的过程包括文件 I/O（读取附件文件和写索引文件）和一些计算（如进行分词），该过程相对于将上传的附件保存到磁盘中而言也快不到哪里。因此，我们不希望对附件生成索引文件这个操作的快慢影响系统用户的体验（如增加了用户等待系统给出操作反馈的时间）。此时，Producer-Consumer 模式可以排上用场：我们可以把负责附件存储的线程看作生产者，其“产品”是一个已经保存到磁盘的文件。另外，我们引入一个负责对已存储的附件文件生成相应索引文件的线程，该线程就相当于消费者，它“消费”了上传到文件服务器的附件文件。

该案例的代码如清单 7-1 所示。其中，负责对上传的附件进行存储的类 AttachmentProcessor，它相当于 Producer-Consumer 模式中的 Producer 参与者，负责对附件文件生成索引文件的线程 indexingThread 则相当于 Producer-Consumer 模式中的 Consumer 参与者。AttachmentProcessor 的实例变量 channel 相当于 Channel 参与者实例。AttachmentProcessor 将上传的附件保存完毕后，就将相应的文件存入通道 channel，便返回了，它不会等待该文件相应的索引文件的生成，因此减少了系统用户的等待时间。而相应文件对应的索引文件由 Consumer 的工作者线程 indexingThread 负责生成。工作者线程 indexingThread 使用了 Two-phase Termination 模式（参见第 5 章）以实现该线程的优雅停止。

清单 7-1. 对附件生成全文检索所需的索引文件

```
//模式角色：Producer-Consumer.Producer
public class AttachmentProcessor {
    private final String ATTACHMENT_STORE_BASE_DIR =
        "/home/viscent/tmp/attachments/";

    // 模式角色：Producer-Consumer.Channel
    private final Channel<File> channel = new BlockingQueueChannel<File>(
       new ArrayBlockingQueue<File>(200));

    // 模式角色：Producer-Consumer.Consumer
    private final AbstractTerminatableThread indexingThread = new
        AbstractTerminatableThread() {

        @Override                                    \
        protected void doRun() throws Exception {
            File file = null;

            file = channel.take();
            try {
                indexFile(file);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                terminationToken.reservations.decrementAndGet();
            }

        }

        // 根据指定文件生成全文搜索所需的索引文件
        private void indexFile(File file) throws Exception {
            // 省略其他代码

            // 模拟生成索引文件的时间消耗
            Random rnd = new Random();
            try {
                Thread.sleep(rnd.nextInt(100));
            } catch (InterruptedException e) {
                ;
            }
        }

    };

    public void init() {
        indexingThread.start();
    }

    public void shutdown() {
        indexingThread.terminate();
    }

    public void saveAttachment(InputStream in, String documentId,
       String originalFileName) throws IOException {
        File file = saveAsFile(in, documentId, originalFileName);
        try {
            channel.put(file);
```

```
        } catch (InterruptedException e) {
            ;
        }
        indexingThread.terminationToken.reservations.incrementAndGet();
    }

    private File saveAsFile(InputStream in, String documentId,
        String originalFileName) throws IOException {
        String dirName = ATTACHMENT_STORE_BASE_DIR + documentId;
        File dir = new File(dirName);
        dir.mkdirs();
        File file = new File(dirName + '/'
            + Normalizer.normalize(originalFileName, Normalizer.Form.NFC));

        // 防止目录跨越攻击
        if (!dirName.equals(file.getCanonicalFile().getParent())) {
            throw new SecurityException("Invalid originalFileName:"
                + originalFileName);
        }

        BufferedOutputStream bos = null;
        BufferedInputStream bis = new BufferedInputStream(in);
        byte[] buf = new byte[2048];
        int len = -1;
        try {
            bos = new BufferedOutputStream(new FileOutputStream(file));
            while ((len = bis.read(buf)) > 0) {
                bos.write(buf, 0, len);
            }
            bos.flush();
        } finally {
            try {
                bis.close();
            } catch (IOException e) {
                ;
            }
            try {
                if (null != bos) {
                    bos.close();
                }
            } catch (IOException e) {
                ;
            }
        }

        return file;
    }

}
```

清单 7-1 所引用的接口 Channel 和类 BlockingQueueChannel 的源码如清单 7-2 和清单 7-3 所示。

清单 7-2. 接口 Channel 的源码

```
/**
 * 对通道参与者进行抽象。
 *
```

```
 * @author Viscent Huang
 *
 * @param <P>
 *          “产品”类型
 */
public interface Channel<P> {
    /**
     * 从通道中取出一个“产品”。
     *
     * @return “产品”
     * @throws InterruptedException
     */
    P take() throws InterruptedException;

    /**
     * 往通道中存储一个“产品”。
     *
     * @param product
     *          “产品”
     * @throws InterruptedException
     */
    void put(P product) throws InterruptedException;
}
```

清单 7-3. 类 BlockingQueueChannel 的源码

```
/**
 * 基于阻塞队列的通道实现。
 *
 * @author Viscent Huang
 *
 * @param <P> “产品”类型
 */
public class BlockingQueueChannel<P> implements Channel<P> {
    private final BlockingQueue<P> queue;

    public BlockingQueueChannel(BlockingQueue<P> queue) {
        this.queue = queue;
    }

    @Override
    public P take() throws InterruptedException {

        return queue.take();
    }

    @Override
    public void put(P product) throws InterruptedException {
        queue.put(product);

    }

}
```

7.4 Producer-Consumer 模式的评价与实现考量

Producer-Consumer 模式使得“产品”的生产者和消费者各自的处理能力（速率）相对来说互不影响。生产者只需要将其“生产”的“产品”放入通道中就可以继续处理，而不必等待相应的“产品”被消费者处理完毕。而消费者运行在其自身的工作者线程中，它只管从通道中取“产品”进行处理，而不必关心这些“产品”由谁“生产”以及如何“生产”这些细节。因而消费者的处理能力相对来说又不影响生产者，同时又与生产者是松耦合（Loose Coupling）的关系。另一方面，当消费者处理能力比生产者处理能力大的时候，可能出现通道为空的情形，此时消费者的工作者线程会被暂挂直到生产者“生产”了新的“产品”。此时出现了事实上的消费者等待生产者的情形。类似地，当消费者的处理能力小于生产者的处理能力时，通道可能会满，导致生产者线程被暂挂直到消费者“消费”了通道中的部分“产品”而腾出了存储空间。此时出现了事实上的生产者等待消费者的情形。因此，我们说生产者和消费者各自的处理能力相互不影响是相对的。

7.4.1 通道积压

Producer-Consumer 模式中，消费者的处理能力往往低于生产者的处理能力。此情形下随着时间的推移，通道中存储的“产品”会越来越多而出现积压，这好比工厂的生产能力比较大，但是其生产的产品的销售情况却不容乐观。为了更好地平衡生产者和消费者的处理能力，我们需要对消费者处理过慢的情形进行一定的处理。常见的方法包括以下两种。

- 使用有界阻塞队列。使用有界阻塞队列（如 ArrayBlockingQueue 和带容量限制的 LinkedBlockingQueue）作为 Channel 参与者的实现可以实现将消费者处理压力“反弹”给生产者的效果，从而使消费者处理负荷过大时相应的生产者的处理能力也下降一定程度以达到平衡二者处理能力的目的。当消费者处理能力低于生产者的处理能力时，作为通道的有界阻塞队列会逐渐积压到队列满，此时生产者线程会被阻塞直到相应的消费者“消费”了队列中的一些“产品”使得队列非满。也就是出现了生产者的步伐等待消费者的步伐的情形。

- 使用带流量控制的无界阻塞队列。使用无界阻塞队列（如不带容量限制 LinkedBlockingQueue）作为 Channel 参与者的实现也可以实现平衡生产者和消费者的处理能力。这通常是借助流量控制实现的，即对同一时间内可以有多少个生产者线程往通道中存储“产品”进行限制，从而达到平衡生产者和消费者的处理能力的目的，如清单 7-4 所示。

清单 7-4. 基于 Semaphore 的支持流量控制的通道实现

```
/**
 * 基于 Semaphore 的支持流量控制的通道实现。
 *
 * @author Viscent Huang
 *
 * @param <P> “产品”类型
 */
public class SemaphoreBasedChannel<P> implements Channel<P> {
    private final BlockingQueue<P> queue;
    private final Semaphore semaphore;

    /**
     *
     * @param queue 阻塞队列，通常是一个无界阻塞队列。
     * @param flowLimit 流量限制数
     */
    public SemaphoreBasedChannel(BlockingQueue<P> queue, int flowLimit) {
        this.queue = queue;
        this.semaphore = new Semaphore(flowLimit);
    }

    @Override
    public P take() throws InterruptedException {
        return queue.take();
    }

    @Override
    public void put(P product) throws InterruptedException {
        semaphore.acquire();
        try {
            queue.put(product);
        } finally {
            semaphore.release();
        }
    }
}
```

7.4.2 工作窃取算法

Producer-Consumer 模式中的通道通常可以使用队列来实现。一个通道可以对应一个或者多个队列实例。本章案例中（代码见清单 7-1），一个通道仅对应一个队列（ArrayBlockingQueue）实例。这意味着，如果有多个消费者从该通道中获取“产品”，那么这些消费者的工作者线程实际上是在共享同一个队列实例，而这会导致锁的竞争，即修改队列的头指针时所需要获得的锁而导致的竞争。如果一个通道实例对应多个队列实例，那么就可以实现多个消费者线程从通道中取“产品”的时候访问的是各自的队列实例。此时，各个消费者线程修改队列的头指针并不会导致锁竞争。

一个通道实例对应多个队列实例的时候，当一个消费者线程处理完该线程对应的队列中的“产

品”时，它可以继续从其他消费者线程对应的队列中取出“产品”进行处理，这样就不会导致该消费者线程闲置，并减轻其他消费者线程的负担。这就是工作窃取（Work Stealing）算法的思想。清单 7-5 展示了一个工作窃取算法的示例代码。

清单 7-5. 工作窃取算法示例代码

```
/**
 * 工作窃取算法示例。该类使用 Two-phase Termination 模式（参见第 5 章）。
 *
 * @author Viscent Huang
 *
 */
public class WorkStealingExample {
    private final WorkStealingEnabledChannel<String> channel;
    private final TerminationToken token = new TerminationToken();

    public WorkStealingExample() {
        int nCPU = Runtime.getRuntime().availableProcessors();
        int consumerCount = nCPU / 2 + 1;

        @SuppressWarnings("unchecked")
        BlockingDeque<String>[] managedQueues = new LinkedBlockingDeque[consumerCount];

        // 该通道实例对应了多个队列实例 managedQueues
        channel = new WorkStealingChannel<String>(managedQueues);

        Consumer[] consumers = new Consumer[consumerCount];
        for (int i = 0; i < consumerCount; i++) {
            managedQueues[i] = new LinkedBlockingDeque<String>();
            consumers[i] = new Consumer(token, managedQueues[i]);

        }

        for (int i = 0; i < nCPU; i++) {
            new Producer().start();
        }

        for (int i = 0; i < consumerCount; i++) {

            consumers[i].start();
        }

    }

    public void doSomething() {

    }

    public static void main(String[] args) throws InterruptedException {
        WorkStealingExample wse;
        wse = new WorkStealingExample();

        wse.doSomething();
        Thread.sleep(3500);

    }
```

```
    private class Producer extends AbstractTerminatableThread {
        private int i = 0;

        @Override
        protected void doRun() throws Exception {
            channel.put(String.valueOf(i++));
            token.reservations.incrementAndGet();
        }

    }

    private class Consumer extends AbstractTerminatableThread {
        private final BlockingDeque<String> workQueue;

        public Consumer(TerminationToken token, BlockingDeque<String> workQueue) {
            super(token);                                    \
            this.workQueue = workQueue;
        }

        @Override
        protected void doRun() throws Exception {

            /*
             * WorkStealingEnabledChannel 接口的 take(BlockingDequepreferedQueue) 方法
             * 实现了工作窃取算法
             */
            String product = channel.take(workQueue);

            System.out.println("Processing product:" + product);

            // 模拟执行真正操作的时间消耗
            try {
                Thread.sleep(new Random().nextInt(50));
            } catch (InterruptedException e) {
                ;
            } finally {
                token.reservations.decrementAndGet();
            }

        }

    }

}
```

清单 7-5 中引用的接口 WorkStealingEnabledChannel 和类 WorkStealingChannel 的源码分别如清单 7-6、清单 7-7 所示。类 AbstractTerminatableThread 和类 TerminationToken 的源码请见清单 5-3、清单 5-4。

清单 7-6. 接口 WorkStealingEnabledChannel 的源码

```
public interface WorkStealingEnabledChannel<P> extends Channel<P> {
    P take(BlockingDeque<P> preferredQueue) throws InterruptedException;
}
```

清单 7-7. 类 WorkStealingChannel 的源码

```
public class WorkStealingChannel<T> implements WorkStealingEnabledChannel<T> {
    // 受管队列
    private final BlockingDeque<T>[] managedQueues;

    public WorkStealingChannel(BlockingDeque<T>[] managedQueues) {
        this.managedQueues = managedQueues;
    }

    @Override
    public T take(BlockingDeque<T> preferredQueue) throws InterruptedException {

        // 优先从指定的受管队列中取“产品”
        BlockingDeque<T> targetQueue = preferredQueue;
        T product = null;

        // 试图从指定的队列队首取“产品”
        if (null != targetQueue) {
            product = targetQueue.poll();
        }

        int queueIndex = -1;

        while (null == product) {
            queueIndex = (queueIndex + 1) % managedQueues.length;
            targetQueue = managedQueues[queueIndex];
            // 试图从其他受管队列的队尾“窃取”“产品”
            product = targetQueue.pollLast();
            if (preferredQueue == targetQueue) {
                break;
            }
        }

        if (null == product) {

            // 随机“窃取”其他受管队列的“产品”
            queueIndex = (int) (System.currentTimeMillis() % managedQueues.length);
            targetQueue = managedQueues[queueIndex];
            product = targetQueue.takeLast();
            System.out.println("stealed from " + queueIndex + ":" + product);
        }

        return product;
    }

    @Override
    public void put(T product) throws InterruptedException {
        int targetIndex = (product.hashCode() % managedQueues.length);
        BlockingQueue<T> targetQueue = managedQueues[targetIndex];
        targetQueue.put(product);
    }

    @Override
    public T take() throws InterruptedException {
        return take(null);
    }

}
```

7.4.3 线程的停止

一个具体的 Producer-Consumer 模式实现通常可以看作一个服务。如果该服务中的 Producer 参与者实例也有其工作者线程，那么该服务的停止就涉及 Producer 参与者和 Consumer 参与者的两种工作者线程的停止。此时，我们需要注意这两种线程的停止顺序：如果先停止 Consumer 参与者的工作者线程则会导致 Producer 参与者新“生产”的“产品”无法被处理；如果先停止 Producer 参与者的工作者线程又可能使 Consumer 参与者的工作者线程处于空等待。并且，停止 Consumer 参与者的工作者线程前是否考虑要等待其处理完所有待处理的“产品”或者将这些“产品”做个备份也是个问题。总的来说，我们可以借助 Two-phase Termination 模式（第 5 章）来先停止 Producer 参与者的工作者线程。当某个服务的所有 Producer 参与者的工作者线程都停止之后，再停止该服务涉及的 Consumer 参与者的工作者线程。相关代码请参阅清单 5-5。

7.4.4 高性能高可靠性的 Producer-Consumer 模式实现

本章中我们给出的 Producer-Consumer 模式实现可以说是一个比较一般的实现。如果应用程序对准备采用 Producer-Consumer 模式实现的服务有较高的性能和可靠性的要求，那么不妨考虑使用开源的 Producer-Consumer 模式实现库 LMAX Disruptor。详情请参阅本章相应的参考资源。

7.5 Producer-Consumer 模式的可复用实现代码

JDK 1.5 引入的标准库类 java.util.concurrent.ThreadPoolExecutor 可以看成是 Producer-Consumer 模式的可复用实现。ThreadPoolExecutor 内部维护的工作队列和工作者线程相当于 Producer-Consumer 模式的 Channel 参与者和 Consumer 参与者。而 ThreadPoolExecutor 的客户端代码则相当于 Producer 参与者。利用 ThreadPoolExecutor 实现 Producer-Consumer 模式，应用代码只需要完成以下几件事情。

1.【必需】创建 Runnable 实例（任务），该实例相当于“产品”。

2.【必需】客户端代码调用 ThreadPoolExecutor 实例的 submit 方法提交一个任务，这相当于 Producer 往通道中放入一个“产品”。

有关 ThreadPoolExecutor 的进一步信息，可参考 Thread Pool 模式（第 9 章）。

7.6 Java 标准库实例

Java 标准库中的类 java.io.PipedOutStream 和 java.io.PipedInputStream 允许一个线程以 I/O 的形式输出数据给另外一个线程。这里，java.io.PipedOutStream、java.io.PipedInputStream 分别相当于 Producer-Consumer 模式的 Producer 参与者和 Consumer 参与者。而 java.io.PipedOutStream 内部维护的缓冲区则相当于 Producer-Consumer 模式的 Channel 参与者。

7.7 相关模式

Producer-Consumer 模式可以看作模式的模式，即许多模式可以看作该模式的一个实例，这里不一一列举。

7.7.1 Guarded Suspension 模式（第 4 章）

Producer-Consumer 模式中，当队列为满时 Producer 参与者的线程会被阻塞直到队列不为满；当队列为空时 Consumer 参与者的线程会被阻塞直到队列不为空。这两种线程的阻塞与唤醒的实现可以借助 Guarded Suspension 模式实现。

7.7.2 Thread Pool 模式（第 9 章）

Thread Pool 模式也可以看作 Producer-Consumer 模式的一个实例。

7.8 参考资源

1. Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

2. Brian Göetz et al.Java Concurrency In Practice.Addison Wesley,2006.

3. 高性能线程间消息传递库 LMAX Disruptor.https://github.com/LMAX-Exchange/disruptor.

第 8 章

Active Object（主动对象）模式

8.1 Active Object 模式简介

Active Object 模式是一种异步编程模式。它通过对方法的调用（Method Invocation）与方法的执行（Method Execution）进行解耦（Decoupling）来提高并发性。若以任务的概念来说，Active Object 模式的核心则是它允许任务的提交（相当于对异步方法的调用）和任务的执行（相当于异步方法的真正执行）分离。这有点类似于 System.gc()这个方法：客户端代码调用完 gc()后，一个进行垃圾回收的任务被提交，但此时 JVM 并不一定进行了垃圾回收，而可能是在 gc()方法调用返回后的某段时间才开始执行任务——回收垃圾。我们知道，System.gc()的调用方代码是运行在自己的线程上（通常是 main 线程派生的子线程），而 JVM 的垃圾回收这个动作则由专门的工作者线程（垃圾回收线程）来执行。换而言之，System.gc()这个方法所代表的动作（其所定义的功能）的调用方和执行方是运行在不同的线程中的，从而提高了并发性。

在进一步介绍 Active Object 模式之前，我们可先简单地将其核心理解为一个名为 ActiveObject 的类，该类对外暴露了一些异步方法，如图 8-1 所示。

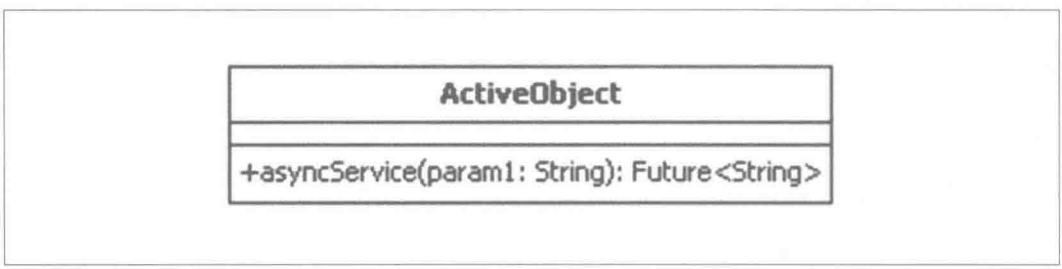


图 8-1. ActiveObject 对象示例

asyncService 方法的调用方和执行方运行在各自的线程上。在多线程环境下，asyncService 方法会被多个线程调用。这时所需的线程安全控制被封装在 asyncService 方法背后，因此调用方代码无须关心这点，从而简化了调用方代码：从调用方代码来看，调用一个 Active Object 对象的方法与调用普通 Java 对象的方法并无实质性差别，如清单 8-1 所示。

清单 8-1. Active Object 方法调用示例

```
ActiveObject ao=...;
Future<String> future = ao.asyncService("data");
//执行其他操作

String result = future.get();
System.out.println(result);
```

8.2 Active Object 模式的架构

当 Active Object 模式对外暴露的异步方法被调用时，与该方法调用相关的上下文信息，包括被调用的异步方法名(或其代表的操作)、客户端代码所传递的参数等，会被封装成一个对象。该对象被称为方法请求（Method Request）。方法请求对象会被存入 Active Object 模式所维护的缓冲区（Activation Queue）中，并由专门的工作者线程负责根据其包含的上下文信息执行相应的操作。也就是说，方法请求对象是由客户端线程（Client Thread）通过调用 Active Object 模式对外暴露的异步方法生成的，而方法请求所代表的操作则由专门的工作者线程来执行，从而实现了方法的调用与执行的分离，产生了并发。

Active Object 模式的主要参与者有以下几种。其类图如图 8-2 所示。

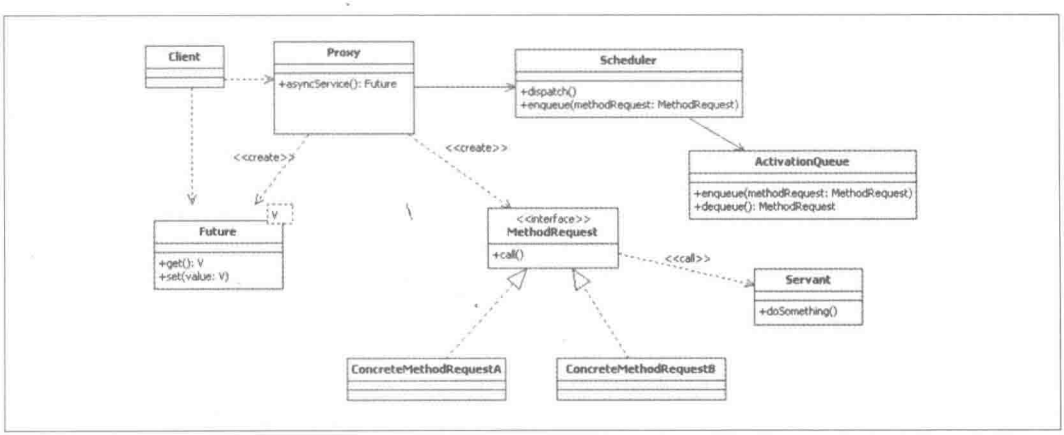


图 8-2. Active Object 模式的类图

- **Proxy**：负责对外暴露异步方法接口。其主要方法及职责如下。
  - **asyncService**：该异步方法负责创建与该方法相应的 MethodRequest 参与者实例，并将其提交给 Scheduler 参与者实例。该方法的返回值是一个 Future 参与者实例，客户端代码可以通过它获取异步方法对应的任务的执行结果。
- **MethodRequest**：负责将客户端代码对 Proxy 实例的异步方法的调用封装为一个对象。该对象保留了异步方法的名称及客户端代码传递的参数等上下文信息。它使得 Proxy 的异步方法的调用和执行分离成为可能。其主要方法及职责如下。
  - **call**：根据其所属 MethodRequest 实例所包含的上下文信息调用 Servant 实例的相应方法。
- **ActivationQueue**：缓冲区，用于临时存储由 Proxy 的异步方法被调用时所创建的 MethodRequest 实例。其主要方法及职责如下。
  - **enqueue**：将 MethodRequest 实例放入缓冲区。
  - **dequeue**：从缓冲区中取出一个 MethodRequest 实例。
- **Scheduler**：负责将 Proxy 的异步方法所创建的 MethodRequest 实例存入其维护的缓冲区中，并根据一定的调度策略，对其维护的缓冲区中的 MethodRequest 实例进行执行。其调度策略可以根据实际需要来定，如 FIFO、LIFO 和根据 MethodRequest 中包含的信息所定的优先级等。其主要方法及职责如下。
  - **enqueue**：接受一个 MethodRequest 实例，并将其存入缓冲区。
  - **dispatch**：反复地从缓冲区中取出 MethodRequest 实例进行执行。
- **Servant**：负责 Proxy 所暴露的异步方法的具体实现。其主要方法及职责如下。
  - **doSomething**：执行 Proxy 所暴露的异步方法对应的任务。
- **Future**：负责存储和获取 Active Object 异步方法的执行结果。其主要方法及职责如下：
  - **get**：获取异步方法对应的任务的执行结果。
  - **set**：设置异步方法对应的任务的执行结果。

Active Object 模式的序列图如图 8-3 所示。

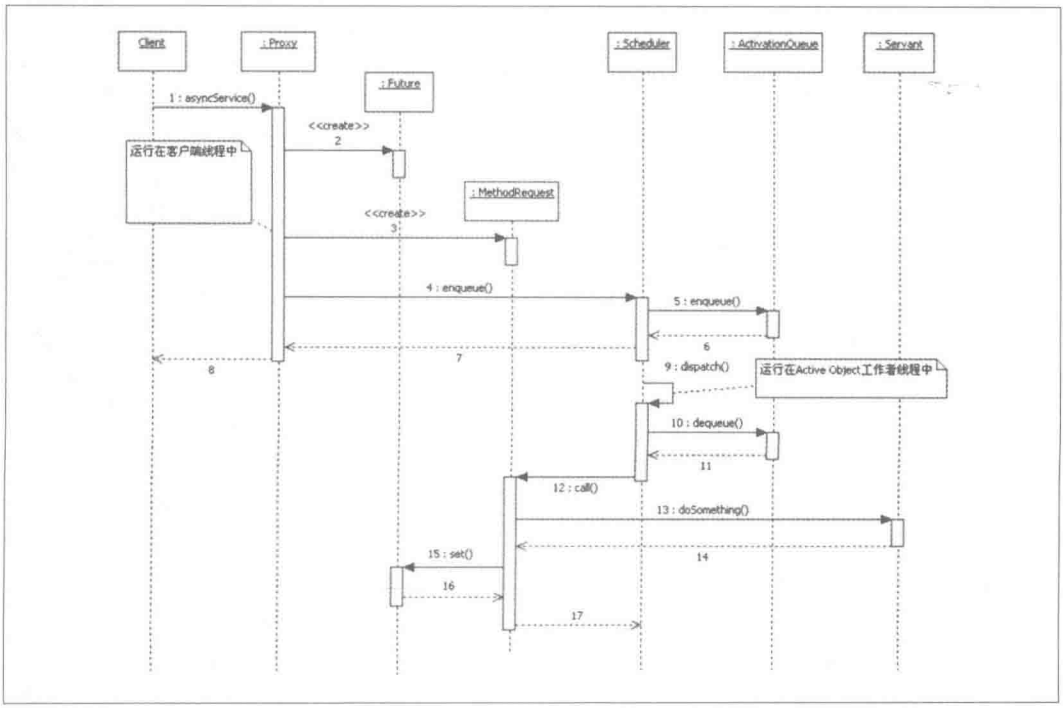


图 8-3. Active Object 模式的序列图

第 1 步：客户端代码调用 Proxy 的异步方法 asyncService。

第 2～7 步：asyncService 方法创建 Future 实例作为该方法的返回值，并将客户端代码对该方法的调用封装为 MethodRequest 对象。然后以所创建的 MethodRequest 对象作为参数调用 Scheduler 的 enqueue 方法，以将 MethodRequest 对象存入缓冲区。Scheduler 的 enqueue 方法会调用 Scheduler 所维护的 ActivationQueue 实例的 enqueue 方法，将 MethodRequest 对象存入缓冲区。

第 8 步：asyncService 返回其所创建的 Future 实例。

第 9 步：Scheduler 实例采用专门的工作者线程运行 dispatch 方法。

第 10～12 步：dispatch 方法调用 ActivationQueue 实例的 dequeue 方法，获取一个 MethodRequest 对象。然后调用 MethodRequest 对象的 call 方法。

第 13～16 步：MethodRequest 对象的 call 方法调用与其关联的 Servant 实例的相应方法 doSomething，并将 Servant.doSomething 方法的返回值设置到 Future 实例上。

第 17 步：MethodRequest 对象的 call 方法返回。

上述步骤中，第 1～8 步是运行在 Active Object 的客户端线程中的，这几个步骤实现了将客户端代码对 Active Object 所提供的异步方法的调用封装成对象（MethodRequest），并将其存入缓冲区（ActivationQueue）。这几个步骤实现了任务的提交。第 9～17 步是运行在 Active Object 的工作者线程中，这些步骤实现从缓冲区中读取 MethodRequest，并对其进行执行，实现了任务的执行。从而实现了 Active Object 对外暴露的异步方法的调用与执行的分离。

如果客户端代码关心 Active Object 的异步方法的返回值，则可以在其需要时，调用 Future 实例的 get 方法来获得异步方法的真正执行结果。

8.3 Active Object 模式实战案例解析

某电信软件有一个彩信短号模块。其主要功能是实现手机用户给其他手机用户发送彩信时，接收方号码可以填写为对方的短号。例如，用户 13612345678 给其同事 13787654321 发送彩信时，可以将接收方号码填写为对方的短号，如 776，而非真实的号码。

该模块处理接收到的下发彩信请求的一个关键操作是，查询数据库以获得接收方短号对应的真实号码（长号）。该操作可能因为数据库故障而失败，从而使整个请求无法继续被处理。而数据库故障是可恢复的故障，因此在短号转换为长号的过程中如果出现数据库异常，可以先将整个下发彩信请求消息缓存到磁盘中，等到数据库恢复后，再从磁盘中读取请求消息，进行重试。为方便起见，我们可以通过 Java 的对象序列化 API，将表示下发彩信的对象序列化到磁盘文件中从而实现请求缓存。下面我们讨论这个请求缓存操作还需要考虑的其他因素，以及 Active Object 模式如何帮助我们满足这些考虑。

首先，请求消息缓存到磁盘中涉及文件 I/O 这种慢的操作，我们不希望它在请求处理的主线程（即 Web 服务器的工作者线程）中执行。因为这样会使该模块的响应延时增大，降低系统的响应性，并使得 Web 服务器的工作者线程因等待文件 I/O 而降低了系统的吞吐量。这时，异步处理就派上用场了。Active Object 模式可以帮助我们实现请求缓存这个任务的提交和执行分离：任务的提交是在 Web 服务器的工作者线程中完成的，而任务的执行（包括序列化对象到磁盘文件中等操作）则是在 Active Object 工作者线程中执行的。这样，请求处理的主线程在侦测到短号转长号失败时即可触发对当前彩信下发请求进行缓存，接着继续其请求处理，如给客户端响应。而此时，当前请求消息可能正在被 Active Object 线程缓存到文件中，如图 8-4 所示。

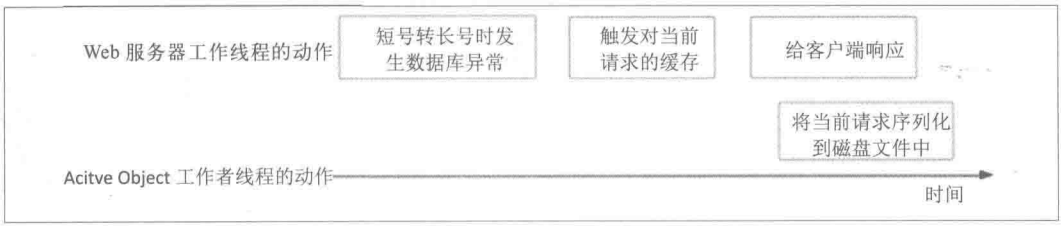


图 8-4. 异步实现缓存

其次，每个短号转长号失败的彩信下发请求消息会被缓存为一个磁盘文件，但我们不希望这些缓存文件被存在同一个子目录下，而是希望多个缓存文件会被存储到多个子目录中。每个子目录最多可以存储指定个数（如 2000 个）的缓存文件。若当前子目录已存满，则新建一个子目录存放新的缓存文件，直到该子目录也存满，依此类推。当这些子目录的个数到达指定数量（如 100 个）时，最老的子目录（连同其下的缓存文件，如果有的话）会被删除，从而保证子目录的个数也是固定的。显然，在并发环境下，实现这种控制需要一些并发访问控制（如通过锁来控制），但是我们不希望这种控制暴露给处理请求的其他代码。而 Active Object 模式中的 Proxy 参与者可以帮助我们封装并发访问控制。

下面，我们看该案例的相关代码通过应用 Active Object 模式在实现缓存功能时满足上述两个目标。首先看请求处理的入口类，该类就是本案例的 Active Object 模式的客户端代码，如清单 8-2 所示。

清单 8-2. 彩信下发请求处理的入口类

```
public class MMSDeliveryServlet extends HttpServlet {

    private static final long serialVersionUID = 5886933373599895099L;

    @Override
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        //将请求中的数据解析为内部对象
        MMSDeliverRequest mmsDeliverReq = this.parseRequest(req.getInputStream());
        Recipient shortNumberRecipient = mmsDeliverReq.getRecipient();
        Recipient originalNumberRecipient = null;

        try {
            // 将接收方短号转换为长号
            originalNumberRecipient = convertShortNumber(shortNumberRecipient);
        } catch (SQLException e) {

            // 接收方短号转换为长号时发生数据库异常，触发请求消息的缓存
            AsyncRequestPersistence.getInstance().store(mmsDeliverReq);

            // 省略其他代码

            resp.setStatus(202);
```

```
    private MMSDeliverRequest parseRequest(InputStream reqInputStream) {
        MMSDeliverRequest mmsDeliverReq = new MMSDeliverRequest();
        //省略其他代码
        return mmsDeliverReq;
    }

    private Recipient convertShortNumber(Recipient shortNumberRecipient)
            throws SQLException {
        Recipient recipent = null;
        //省略其他代码
        return recipent;
    }

}
```

清单 8-2 中的 doPost 方法在侦测到短号转换过程中发生的数据库异常后，通过调用 AsyncRequestPersistence 类的 store 方法触发对彩信下发请求消息的缓存。这里，AsyncRequestPersistence 类是彩信下发请求缓存入口类，它相当于 Active Object 模式中的 Proxy 参与者。尽管本案例涉及的是一个并发环境，但从清单 8-2 中的代码可见，AsyncRequestPersistence 类的客户端代码无须处理多线程同步问题。这是因为多线程同步问题被封装在 AsyncRequestPersistence 类之后。

AsyncRequestPersistence 类的代码如清单 8-3 所示。

清单 8-3. AsyncRequestPersistence 类源码

```
//模式角色：ActiveObject.Proxy
public class AsyncRequestPersistence implements RequestPersistence {
    private static final long ONE_MINUTE_IN_SECONDS = 60;
    private final Logger logger;
    private final AtomicLong taskTimeConsumedPerInterval = new AtomicLong(0);
    private final AtomicInteger requestSubmittedPerIterval = new AtomicInteger(0);

    //模式角色：ActiveObject.Servant
    private final DiskbasedRequestPersistence delegate = new
        DiskbasedRequestPersistence();

    //模式角色：ActiveObject.Scheduler
    private final ThreadPoolExecutor scheduler;

    //用于保存 AsyncRequestPersistence 的唯一实例
    private static class InstanceHolder {
        final static RequestPersistence INSTANCE = new AsyncRequestPersistence();
    }

    //私有构造器
    private AsyncRequestPersistence() {
        logger = Logger.getLogger(AsyncRequestPersistence.class);
        scheduler = new ThreadPoolExecutor(1, 3, 60 * ONE_MINUTE_IN_SECONDS,
            TimeUnit.SECONDS,
            //模式角色：ActiveObject.ActivationQueue
            new ArrayBlockingQueue<Runnable>(200), new ThreadFactory() {
                @Override
```

```
            public Thread newThread(Runnable r) {
                Thread t;
                t = new Thread(r, "AsyncRequestPersistence");
                return t;
            }

        });

    scheduler
        .setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());

    // 启动队列监控定时任务
    Timer monitorTimer = new Timer(true);
    monitorTimer.scheduleAtFixedRate(new TimerTask() {

        @Override
        public void run() {
            if (logger.isInfoEnabled()) {

                logger.info("task count:" + requestSubmittedPerIterval
                    + ",Queue size:" + scheduler.getQueue().size()
                    + ",taskTimeConsumedPerInterval:"
                    + taskTimeConsumedPerInterval.get() + " ms");
            }

            taskTimeConsumedPerInterval.set(0);
            requestSubmittedPerIterval.set(0);

        }
    }, 0, ONE_MINUTE_IN_SECONDS * 1000);

}

//获取类 AsyncRequestPersistence 的唯一实例
public static RequestPersistence getInstance() {
    return InstanceHolder.INSTANCE;
}

@Override
public void store(final MMSDeliverRequest request) {
    /*
     * 将对 store 方法的调用封装成 MethodRequest 对象，并存入缓冲区。
     */
    //模式角色：ActiveObject.MethodRequest
    Callable<Boolean> methodRequest = new Callable<Boolean>() {
        @Override
        public Boolean call() throws Exception {
            long start = System.currentTimeMillis();
            try {
                delegate.store(request);
            } finally {
                taskTimeConsumedPerInterval.addAndGet(
                    System.currentTimeMillis()- start);
            }

            return Boolean.TRUE;
        }
```

```
            };
            scheduler.submit(methodRequest);

            requestSubmittedPerIterval.incrementAndGet();
        }
}
```

AsyncRequestPersistence 类所实现的接口 RequestPersistence 定义了 Active Object 对外暴露的异步方法：store 方法。由于本案例不关心请求缓存的结果，故该方法没有返回值。其代码如清单 8-4 所示。

清单 8-4. RequestPersistence 接口源码

```
public interface RequestPersistence {

    void store(MMSDeliverRequest request);
}
```

AsyncRequestPersistence 类的实例变量 scheduler 相当于 Active Object 模式中的 Scheduler 参与者实例。这里我们直接使用了 JDK1.5 引入的 Executor Framework 中的 ThreadPoolExecutor。在 ThreadPoolExecutor 类实例化时，其构造器的第 5 个参数（BlockingQueue<Runnable> workQueue）我们指定了一个有界阻塞队列：new ArrayBlockingQueue<Runnable>(200)。该队列相当于 Active Object 模式中的 ActivationQueue 参与者实例。

AsyncRequestPersistence 类的实例变量 delegate 相当于 Active Object 模式中的 Servant 参与者实例。

AsyncRequestPersistence 类的 store 方法利用匿名类生成一个 java.util.concurrent.Callable 实例 methodRequest。该实例相当于 Active Object 模式中的 MethodRequest 参与者实例。利用闭包（Closure），该实例封装了对 store 方法调用的上下文信息（包括调用参数、所调用的方法对应的操作信息）。AsyncRequestPersistence 类的 store 方法通过调用 scheduler 的 submit 方法，将 methodRequest 送入 ThreadPoolExecutor 所维护的缓冲区（阻塞队列）中。确切地说，ThreadPoolExecutor 是 Scheduler 参与者的一个“近似”实现。ThreadPoolExecutor 的 submit 方法相对于 Scheduler 的 enqueue 方法，该方法用于接纳 MethodRequest 对象，以将其存入缓冲区。当 ThreadPoolExecutor 当前使用的线程数量小于其核心线程数量时，submit 方法所接收的任务会直接被新建的线程执行。当 ThreadPoolExecutor 当前使用的线程数量大于其核心线程数时，submit 方法所接收的任务才会被存入其维护的阻塞队列中。不过，ThreadPoolExecutor 的这种任务处理机制，并不妨碍我们将它用作 Scheduler 的实现。

methodRequest 的 call 方法会调用 delegate 的 store 方法来真正实现请求缓存功能。delegate 实例对应的类 DiskbasedRequestPersistence 是请求消息缓存功能的真正实现者。其代码如清单

8-5 所示。

清单 8-5. DiskbasedRequestPersistence 类的源码

```
public class DiskbasedRequestPersistence implements RequestPersistence {
    // 负责缓存文件的存储管理
    private final SectionBasedDiskStorage storage = new SectionBasedDiskStorage();
    private final Logger logger = Logger.getLogger(
        DiskbasedRequestPersistence.class);
    @Override
    public void store(MMSDeliverRequest request) {
        // 申请缓存文件的文件名
        String[] fileNameParts = storage.apply4Filename(request);
        File file = new File(fileNameParts[0]);
        try {
            ObjectOutputStream objOut = new ObjectOutputStream(new
                FileOutputStream(file));
            try {
                objOut.writeObject(request);
            } finally {
                objOut.close();
            }
        } catch (FileNotFoundException e) {
            storage.decrementSectionFileCount(fileNameParts[1]);
            logger.error("Failed to store request", e);
        } catch (IOException e) {
            storage.decrementSectionFileCount(fileNameParts[1]);
            logger.error("Failed to store request", e);
        }

    }

    class SectionBasedDiskStorage {
        private Deque<String> sectionNames = new LinkedList<String>();
        /*
         * Key->value: 存储子目录名->子目录下缓存文件计数器
         */
        private Map<String, AtomicInteger> sectionFileCountMap
            = new HashMap<String, AtomicInteger>();
        private int maxFilesPerSection = 2000;
        private int maxSectionCount = 100;
        private String storageBaseDir = System.getProperty("user.dir") + "/vpn";

        private final Object sectionLock = new Object();

        public String[] apply4Filename(MMSDeliverRequest request) {
            String sectionName;
            int iFileCount;
            boolean need2RemoveSection = false;
            String[] fileName = new String[2];
            synchronized (sectionLock) {
                //获取当前的存储子目录名
                sectionName = this.getSectionName();
                AtomicInteger fileCount;
                fileCount = sectionFileCountMap.get(sectionName);
                iFileCount = fileCount.get();
                //当前存储子目录已满
                if (iFileCount >= maxFilesPerSection) {
```

```
                if (sectionNames.size() >= maxSectionCount) {
                    need2RemoveSection = true;
                }
                //创建新的存储子目录
                sectionName = this.makeNewSectionDir();
                fileCount = sectionFileCountMap.get(sectionName);

            }
            iFileCount = fileCount.addAndGet(1);

        }

        fileName[0] = storageBaseDir + "/" + sectionName + "/"
            + new DecimalFormat("0000").format(iFileCount) + "-"
            + request.getTimeStamp().getTime() / 1000 + "-"
            + request.getExpiry()
            + ".rq";
        fileName[1] = sectionName;

        if (need2RemoveSection) {
            //删除最老的存储子目录
            String oldestSectionName = sectionNames.removeFirst();
            this.removeSection(oldestSectionName);
        }

        return fileName;
    }

    public void decrementSectionFileCount(String sectionName) {
        AtomicInteger fileCount = sectionFileCountMap.get(sectionName);
        if (null != fileCount) {
            fileCount.decrementAndGet();
        }
    }

    private boolean removeSection(String sectionName) {
        boolean result = true;
        File dir = new File(storageBaseDir + "/" + sectionName);
        for (File file : dir.listFiles()) {
            result = result && file.delete();
        }
        result = result && dir.delete();
        return result;
    }

    private String getSectionName() {
        String sectionName;

        if (sectionNames.isEmpty()) {
            sectionName = this.makeNewSectionDir();

        } else {
            sectionName = sectionNames.getLast();
        }

        return sectionName;
    }

    private String makeNewSectionDir() {
```

```
            String sectionName;
            SimpleDateFormat sdf = new SimpleDateFormat("MMddHHmmss");
            sectionName = sdf.format(new Date());
            File dir = new File(storageBaseDir + "/" + sectionName);
            if (dir.mkdir()) {
                sectionNames.addLast(sectionName);
                sectionFileCountMap.put(sectionName, new AtomicInteger(0));
            } else {
                throw new RuntimeException("Cannot create section dir " +
                    sectionName);
            }

            return sectionName;
        }
    }
}
```

methodRequest 的 call 方法的调用者代码是运行在 ThreadPoolExecutor 所维护的工作者线程中的，这就保证了 store 方法的客户端和真正的执行方是分别运行在不同的线程中的：服务器工作者线程负责触发请求消息缓存，ThreadPoolExecutor 所维护的工作者线程负责将请求消息序列化到磁盘文件中。

DiskbasedRequestPersistence 类的 store 方法中调用的 SectionBasedDiskStorage 类的 apply4Filename 方法包含了一些多线程同步控制代码（见清单 8-5）。这部分控制由于是封装在 DiskbasedRequestPersistence 的内部类中的，对于该类之外的代码是不可见的。因此，AsyncRequestPersistence 的客户端代码无法知道该细节，这体现了 Active Object 模式对并发访问控制的封装。

8.4 Active Object 模式的评价与实现考量

Active Object 模式通过将方法的调用与执行分离，实现了异步编程。有利于提高并发性，从而提高系统的吞吐率。

Active Object 模式还有个好处是它可以将任务（MethodRequest）的提交（调用异步方法）和任务的执行策略（Execution Policy）分离。任务的执行策略被封装在 Scheduler 的实现类之内，因此它对外是“不可见”的，一旦需要变动也不会影响其他代码，从而降低了系统的耦合性。任务的执行策略可以反映以下一些问题。

- 采用什么顺序去执行任务，如 FIFO、LIFO，或者基于任务中包含的信息所定的优先级？
- 多少个任务可以并发执行？
- 多少个任务可以被排队等待执行？

- 如果有任务由于系统过载被拒绝，此时哪个任务该被选中作为牺牲品，应用程序该如何被通知到？
- 任务执行前、执行后需要执行哪些操作？

这意味着，任务的执行顺序可以和任务的提交顺序不同，可以采用单线程也可以采用多线程去执行任务等。

当然，好处的背后总是隐藏着代价，Active Object 模式实现异步编程也有其代价。该模式的参与者有 6 个之多，其实现过程也包含了不少中间的处理：MethodRequest 对象的生成、MethodRequest 对象的移动（进出缓冲区）、MethodRequest 对象的运行调度和线程上下文切换等。这些处理都有其空间和时间的代价。因此，Active Object 模式适合于分解一个比较耗时的任务（如涉及 I/O 操作的任务）：将任务的发起和执行进行分离，以减少不必要的等待时间。

虽然模式的参与者较多，但正如本章案例的实现代码所展示的，其中大部分的参与者我们可以利用 JDK 自身提供的类来实现，以节省编码时间，如表 8-1 所示。

表 8-1. 使用 JDK 现有类实现 Active Object 的一些参与者

参与者名称	可以借用的 JDK 类	备　注
Scheduler	Java Executor Framework 中的 java.util.concurrent.ExecutorService 接口的相关实现类，如 java.util.concurrent.ThreadPool Executor	ExecutorService 接口所定义的 submit(Callable<T> task) 方法相当于图 8-2 中的 enqueue 方法
ActivationQueue	java.util.concurrent.LinkedBlockingQueue	若 Scheduler 采用 java.util. concurrent. ThreadPoolExecutor，则 java.util.concurrent. LinkedBlocking Queue 实例作为 ThreadPool Executor 构造器的参数传入即可
MethodRequest	java.util.concurrent.Callable 接口的实现类	Callable 接口比起 Runnable 接口的优势在于它定义的 call 方法有返回值，便于将该返回值传递给 Future 实例。通常使用 Callable 接口的匿名实现类即可
Future	java.util.concurrent.Future	ExecutorService 接口所定义的 submit(Callable<T> task) 方法的返回值类型就是 java.util.concurrent. Future

8.4.1 错误隔离

错误隔离指一个任务的处理失败不影响其他任务的处理。每个 MethodRequest 实例可以看作一个任务。那么，Scheduler 的实现类在执行 MethodRequest 时需要注意错误隔离。选用 JDK 中现成的类（如 ThreadPoolExecutor）来实现 Scheduler 的一个好处就是这些类可能已经实现了错误隔离。而如果自己编写代码实现 Scheduler，用单个 Active Object 工作者线程逐一执行所有任务，则需要特别注意线程的 run 方法的异常处理，确保不会因为个别任务执行时遇到一些运行时异常而导致整个线程终止。如清单 8-6 所示的示例代码。

清单 8-6. 自己动手实现 Scheduler 的错误隔离示例代码

```
public class CustomScheduler implements Runnable {
    private LinkedBlockingQueue<Runnable> activationQueue = new
        LinkedBlockingQueue<Runnable>();

    @Override
    public void run() {
        dispatch();
    }

    public <T> Future<T> enqueue(Callable<T> methodRequest) {
        final FutureTask<T> task = new FutureTask<T>(methodRequest) {

            @Override
            public void run() {
                try {
                   super.run();
                //捕获所有可能抛出的对象，避免该任务运行失败而导致其所在的线程终止。
                } catch (Throwable t) {
                   this.setException(t);
                }
            }

        };

        try {
            activationQueue.put(task);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return task;
    }

    public void dispatch() {
        while (true) {
            Runnable methodRequest;
            try {
                methodRequest = activationQueue.take();

                //防止个别任务执行失败导致线程终止的代码在 run 方法中
                methodRequest.run();
            } catch (InterruptedException e) {
                // 处理该异常
```

```
                }
            }
        }
}
```

8.4.2 缓冲区监控

如果 ActivationQueue 是有界缓冲区，则对缓冲区的当前大小进行监控无论是对于运维还是测试来说都有其意义。从测试的角度来看，监控缓冲区有助于确定缓冲区容量的建议值（合理值）。如清单 8-3 所示的代码，即通过定时任务周期性地调用 ThreadPoolExecutor 的 getQueue 方法对缓冲区的大小进行监控。当然，在监控缓冲区的时候，往往只需要大致的值，因此在监控代码中要注意避免不必要的锁。

8.4.3 缓冲区饱和处理策略

当任务的提交速率大于任务的执行速率时，缓冲区可能逐渐积压到满。这时新提交的任务会被拒绝。无论是自己编写代码还是利用 JDK 现有类来实现 Scheduler，对于缓冲区满时新任务提交失败，我们需要一个处理策略用于决定此时哪个任务会成为“牺牲品”。若使用 ThreadPoolExecutor 来实现 Scheduler 有个好处，是它已经提供了几个缓冲区饱和处理策略的实现代码，应用代码可以直接调用。如清单 8-3 所示的代码，本章案例中我们选择了在任务的提交方线程中执行被拒绝的任务作为处理策略。

java.util.concurrent.RejectedExecutionHandler 接口是 ThreadPoolExecutor 对缓冲区饱和处理策略的抽象，JDK 中提供的具体实现类如表 8-2 所示。

表 8-2. JDK 提供的缓冲区饱和处理策略实现类

实 现 类	所实现的处理策略
ThreadPoolExecutor.AbortPolicy	直接抛出异常
ThreadPoolExecutor.DiscardPolicy	丢弃当前被拒绝的任务（而不抛出任何异常）
ThreadPoolExecutor.DiscardOldestPolicy	将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务
ThreadPoolExecutor.CallerRunsPolicy	在任务的提交方线程中运行被拒绝的任务

当然，对于 ThreadPoolExecutor 而言，其工作队列满不一定就意味着新提交的任务会被拒绝。当其最大线程池大小大于其核心线程池大小时，工作队列满的情况下，新提交的任务会用所有核心线程之外的新增线程来执行，直到工作者线程数达到最大线程数时，新提交的任务才会被拒绝。

8.4.4 Scheduler 空闲工作者线程清理

如果 Scheduler 采用多个工作者线程(如采用 ThreadPoolExecutor 这样的线程池)来执行任务，则可能需要清理空闲的线程以节约资源。清单 8-3 的代码就是直接使用了 ThreadPoolExecutor 的现有功能，在初始化实例时通过指定其构造器的第 3、4 个参数（ long keepAliveTime、TimeUnit unit)，告诉 ThreadPoolExecutor 对于核心工作者线程以外的线程，若已经空闲了指定时间，则将其清理掉。

8.5 Active Object 模式的可复用实现代码

尽管利用 JDK 中的现成类可以极大地简化 Active Object 模式的实现，但如果需要频繁地在不同场景下使用 Active Object 模式，则需要一套更利于复用的代码，以节约编码的时间和使代码更加易于理解。清单 8-7 展示了一段基于 Java 动态代理的可复用的 Active Object 模式 Proxy 参与者实现代码。

清单 8-7. 可复用的 Active Object 模式 Proxy 参与者实现

```
/**
 * Active Object 模式 Proxy 参与者的可复用实现。
 * 模式角色：ActiveObject.Proxy
 * @author Viscent Huang
 */
public abstract class ActiveObjectProxy {

    private static class DispatchInvocationHandler implements InvocationHandler {
        private final Object delegate;
        private final ExecutorService scheduler;

        public DispatchInvocationHandler(Object delegate,
            ExecutorService executorService) {
            this.delegate = delegate;
            this.scheduler = executorService;
        }

        private String makeDelegateMethodName(final Method method,
            final Object[] arg) {
            String name = method.getName();
            name = "do" + Character.toUpperCase(name.charAt(0))
                    + name.substring(1);

            return name;
        }

        @Override
        public Object invoke(final Object proxy, final Method method,
            final Object[] args) throws Throwable {

            Object returnValue = null;
            final Object delegate = this.delegate;
```

```
        final Method delegateMethod;

        //如果拦截到的被调用方法是异步方法，则将其转发到相应的 doXXX 方法
        if (Future.class.isAssignableFrom(method.getReturnType())) {
            delegateMethod = delegate.getClass().getMethod(
                                makeDelegateMethodName(method, args),
                                method.getParameterTypes());

            final ExecutorService scheduler = this.scheduler;

            Callable<Object> methodRequest = new Callable<Object>() {
                @Override
                public Object call() throws Exception {
                    Object rv = null;

                    try {
                        rv = delegateMethod.invoke(delegate, args);
                    } catch (IllegalArgumentException e) {
                        throw new Exception(e);
                    } catch (IllegalAccessException e) {
                        throw new Exception(e);
                    } catch (InvocationTargetException e) {
                        throw new Exception(e);
                    }
                    return rv;
                }
            };
            Future<Object> future = scheduler.submit(methodRequest);
            returnValue = future;

        } else {

            //若拦截到的方法调用不是异步方法，则直接转发

           delegateMethod = delegate.getClass()
           .getMethod(method.getName(),method.getParameterTypes());

            returnValue = delegateMethod.invoke(delegate, args);
        }

        return returnValue;
    }
}

/**
 * 生成一个实现指定接口的 Active Object proxy 实例。
 * 对 interf 所定义的异步方法的调用会被转发到 servant 的相应的 doXXX 方法。
 * @param interf 要实现的 Active Object 接口
 * @param servant Active Object 的 Servant 参与者实例
 * @param scheduler Active Object 的 Scheduler 参与者实例
 * @return Active Object 的 Proxy 参与者实例
 */
public static <T> T newInstance(Class<T> interf, Object servant,
    ExecutorService scheduler) {
     @SuppressWarnings("unchecked")
     T f = (T) Proxy.newProxyInstance(interf.getClassLoader(),
           new Class[] { interf }, new DispatchInvocationHandler(servant,
           scheduler));
```

```
        return f;
    }
}
```

清单 8-7 的代码实现了可复用的 Active Object 模式的 Proxy 参与者 ActiveObjectProxy。ActiveObjectProxy 通过使用 Java 动态代理，动态生成指定接口的代理对象。对该代理对象的异步方法（即返回值类型为 java.util.concurrent.Future 的方法）的调用会被 ActiveObjectProxy 实现 InvocationHandler（DispatchInvocationHandler）所拦截，并转发给 ActiveObjectProxy 的 newInstance 方法中指定的 Servant 处理。

使用 ActiveObjectProxy 实现 Active Object 模式，应用代码只需要调用 ActiveObjectProxy 的静态方法 newInstance 即可。应用代码调用 newInstance 方法需要指定以下参数：

1）指定 Active Object 模式对外暴露的接口，该接口作为第 1 个参数传入。

2）创建 Active Object 模式对外暴露的接口的实现类。该类的实例作为第 2 个参数传入。

3）指定一个 java.util.concurrent.ExecutorService 实例。该实例作为第 3 个参数传入。

如清单 8-8 所示的代码展示了通过使用 ActiveObjectProxy 快速实现 Active Object 模式。

清单 8-8. 基于可复用的 API 快速实现 Active Object 模式

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

    SampleActiveObject sao = ActiveObjectProxy.newInstance(
            SampleActiveObject.class, new SampleActiveObjectImpl(),
            Executors.newCachedThreadPool());
    Future<String> ft = sao.process("Something", 1);

    Thread.sleep(50);

    System.out.println(ft.get());
}
```

8.6 Java 标准库实例

类 java.util.concurrent.ThreadPoolExecutor 可以看成是 Active Object 模式的一个通用实现。ThreadPoolExecutor 自身相当于 Active Object 模式的 Proxy 和 Scheduler 参与者实例。ThreadPoolExecutor 的 submit 方法相当于 Active Object 模式对外暴露的异步方法。该方法的唯一参数（java.util.concurrent.Callable 或者 java.lang.Runnable）可以看作是 MethodRequest 参与者实例。该方法的返回值（java.util.concurrent.Future）相当于 Future 参与者实例，而 ThreadPoolExecutor 的构造方法中需要传入的 BlockingQueue 实例相当于 ActivationQueue 参与者实例。

8.7 相关模式

8.7.1 Promise 模式（第 6 章）

Active Object 模式的 Proxy 参与者相当于 Promise 模式中的 Promisor 参与者，其 asyncService 异步方法的返回值类型 Future 相当于 Promise 模式中的 Promise 参与者。

8.7.2 Producer-Consumer 模式（第 7 章）

整体上看，Active Object 模式可以看作 Producer-Consumer 模式的一个实例：Active Object 模式的 Proxy 参与者可以看成 Producer-Consumer 模式中的 Producer 参与者，它“生产”了 MethodRequest 参与者这种“产品”；Scheduler 参与者可以看成 Producer-Consumer 模式中的 Consumer 参与者，它“消费”了 Proxy 参与者所“生产”的 MethodRequest。

8.8 参考资源

1．维基百科 Active Object 模式词条.http://en.wikipedia.org/wiki/Active_object.

2．Douglas C. Schmidt 对 Active Object 模式的定义.http://www.laputan.org/pub/sag/act-obj.pdf.

3．Schmidt, Douglas et al. Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

4．Java theory and practice: Decorating with dynamic proxies. http://www.ibm.com/developerworks/java/library/j-jtp08305/index.html.

第9章

Thread Pool（线程池）模式

9.1 Thread Pool 模式简介

一个系统中的线程相对于其所要处理的任务而言，总是一种非常有限的资源。线程不仅在其执行任务时需要消耗 CPU 时间和内存等资源，线程对象（Thread 实例）本身以及线程所需的调用栈（Call Stack）也占用内存，并且 Java 中创建一个线程往往意味着 JVM 会创建相应的依赖于宿主机操作系统的本地线程（Native Thread）。因此，为每个或者每一批任务创建一个线程以对其进行执行，通常是一种奢侈而不现实的事情。比较常见的一种做法是复用一定数量的线程，由这些线程去执行不断产生的任务。绝大多数的 Web 服务器就是采用这种方法。例如，Tomcat 服务器复用一定数量的线程用于处理其接收到的请求。

Thread Pool 模式的核心思想是使用队列对待处理的任务进行缓存，并复用一定数量的工作者线程去取队列中的任务进行执行。

Thread Pool 模式的本质是使用极其有限的资源去处理相对无限的任务。这好比一个生意兴隆的饭店，虽然每天顾客不断，但饭店却不可能因为来一批客人就增加一个服务员。相反，服务员的人数还是那么多，只不过饭店生意好的时候，服务员们比较忙碌，生意不好时，服务员们比较空闲。

JDK 1.5 引入的标准库类 java.util.concurrent.ThreadPoolExecutor 就是 Thread Pool 模式的一个实现。读者如果对 ThreadPoolExecutor 有所了解，也不妨继续阅读本章。毕竟，使用 Thread Pool 模式不仅仅是会使用 ThreadPoolExecutor 提供的相关 API 那么简单，其背后还有不少风险和问题需要注意。

9.2 Thread Pool 模式的架构

Thread Pool 模式的主要参与者有以下几种。其类图如图 9-1 所示。

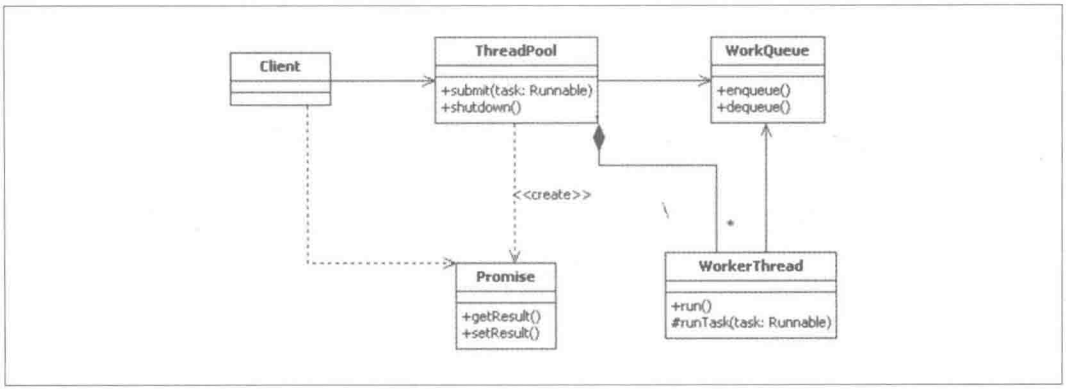


图 9-1. Thread Pool 模式的类图

- **ThreadPool**：负责接收和存储任务以及工作者线程的生命周期管理。其主要方法及职责如下。
  - **submit**：用于接收一个任务，客户端代码调用该方法向线程池提交一个任务。
  - **shutdown**：关闭线程池对外提供的服务。
- **Promise**：可借以获取相应任务执行结果的凭据对象。其主要方法及职责如下。
  - **getResult**：获取相应任务的执行结果。
  - **setResult**：设置相应任务的执行结果。
- **WorkQueue**：工作队列。实现任务的缓存。其主要方法及职责如下。
  - **enqueue**：将任务存入队列。
  - **dequeue**：从队列中取出一个任务。
- **WorkerThread**：负责任务执行的工作者线程。其主要方法及职责如下。
  - **run**：逐一从工作队列中取出任务执行。
  - **runTask**：执行指定的任务。

客户端代码向线程池提交任务的序列图如图 9-2 所示。

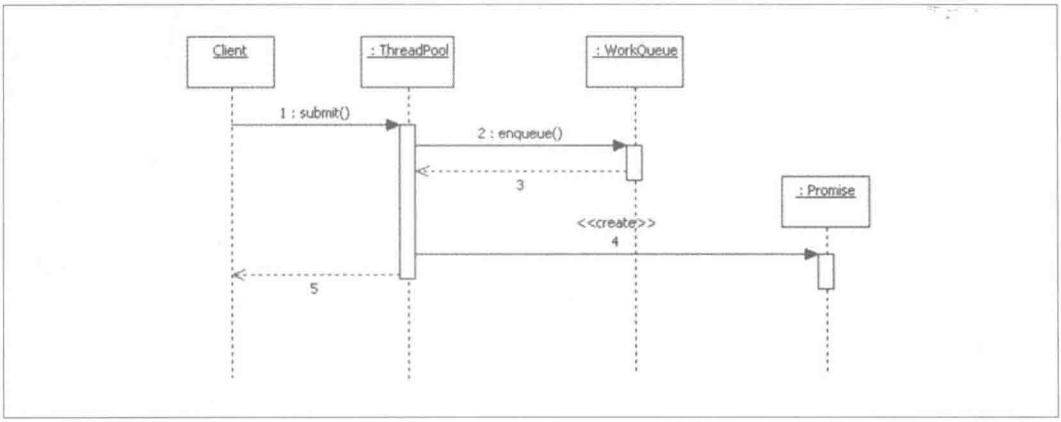


图 9-2. 客户端代码提交任务的序列图

第 1 步：客户端代码调用 ThreadPool 参与者实例的 submit 方法提交一个任务。

第 2、3 步：submit 方法调用 WorkQueue 参与者实例的 enqueue 方法将任务缓存。

第 4、5 步：submit 方法创建与当前任务相应的 Promise 参与者实例，并将其作为返回值返回。

线程池执行任务的序列图如图 9-3 所示。

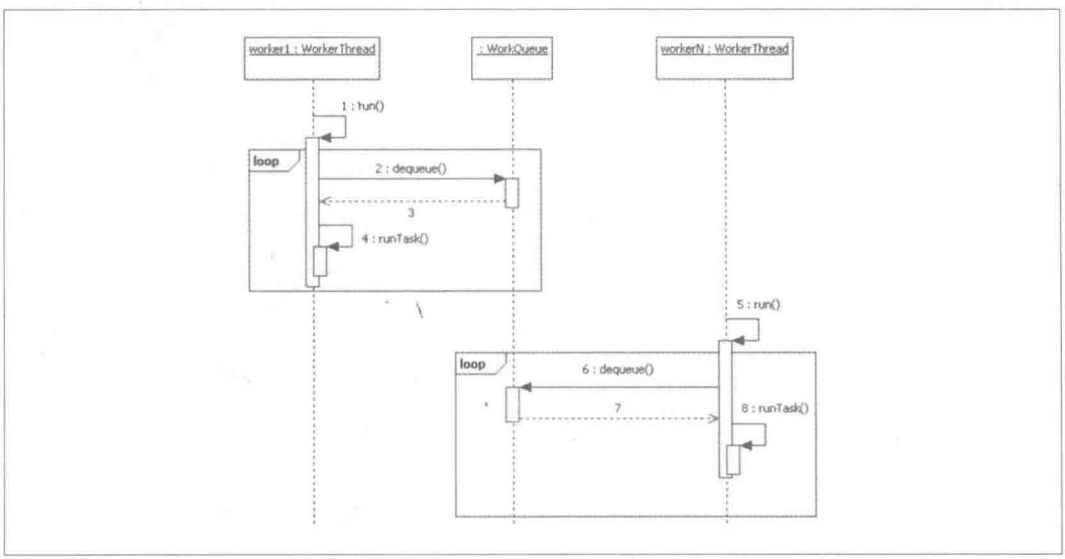


图 9-3. 线程池执行任务的序列图

第 1 步：某个工作者线程开始运行，其 run 方法被 JVM 调用。

第 2～4 步：这几个步骤是 run 方法中的循环。该循环重复地从工作队列中取出一个任务执行。若工作队列中暂时没有任务，则当前工作者线程会被挂起，直到工作队列中有新的任务。

第 5 步：其他工作者线程开始运行，相应线程的 run 方法被 JVM 调用。

第 6～8 步：类似于上述第 2～4 步，其他工作者线程取工作队列中的任务执行。

9.3 Thread Pool 模式实战案例解析

某系统在用户执行一些关键的操作前要求其输入一个验证码。验证码是一串随机数字，由该系统的服务器端代码生成并通过短信发送给用户。

服务端代码实现短信发送功能需要调用其他系统提供的 Web 服务（Web Service）。从设计上看，我们希望一个名为 SmsVerficationCodeSender 的类负责验证码的生成和相应短信的下发。这样，系统在发送验证码短信时只需要调用 SmsVerficationCodeSender 实例的相应方法即可。考虑到 SmsVerficationCodeSender 的客户端代码（即该系统需要发送验证码短信的代码）是运行在服务器的请求处理线程中的，我们不希望发送验证码短信时所涉及的网络 I/O 这种相对慢的操作影响服务器请求处理线程执行其他操作（如继续处理其他请求）。因此，SmsVerficationCodeSender 需要采用专门的工作者线程负责验证码短信的发送。另外，考虑到系统的并发量，每次发送验证码短信都启动一个专门的工作者线程显然是过于昂贵。

这里，Thread Pool 模式就可以派上用场。我们可以创建一个包含若干个工作者线程的线程池，系统需要下发验证码短信时就创建一个相应的任务，并将其提交给这个线程池执行。由于向线程池提交任务这个操作可即刻返回，因此验证码短信发送的快慢并不会影响需要发送验证码的线程继续处理。代码如清单 9-1 所示。

清单 9-1. 验证码短信下发源码

```
public class SmsVerficationCodeSender {
    private static final ExecutorService EXECUTOR = new ThreadPoolExecutor(1,
        Runtime.getRuntime().availableProcessors(), 60, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r, "VerfCodeSender");
                t.setDaemon(true);
                return t;
            }

        }, new ThreadPoolExecutor.DiscardPolicy());

    /**
```

```
     * 生成并下发验证码短信到指定的手机号码。
     *
     * @param msisdn 短信接收方号码。
     */
    public void sendVerificationSms(final String msisdn) {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                //生成强随机数验证码
                int verificationCode = ThreadSpecificSecureRandom.INSTANCE
                    .nextInt(999999);
                DecimalFormat df = new DecimalFormat("000000");
                String txtVerCode = df.format(verificationCode);

                //发送验证码短信
                sendSms(msisdn, txtVerCode);
            }

        };

        EXECUTOR.submit(task);

    }

    private void sendSms(String msisdn, String verificationCode) {
        System.out.println("Sending verification code " + verificationCode + " to "
            + msisdn);

        // 省略其他代码
    }

}
```

清单 9-1 中的类 SmsVerficationCodeSender 的 sendVerificationSms 方法被调用时会创建一个包含验证码生成和验证码发送这两个操作的任务（Runnable 实例），并将该任务提交给线程池实例 EXECUTOR 执行。

清单 9-1 所引用的类 ThreadSpecificSecureRandom 使用了 Thread Specific Storage 模式（参见第 10 章），其源码参见清单 10-2。

9.4 Thread Pool 模式的评价与实现考量

Thread Pool 模式通过复用一定数量的工作者线程去执行不断被提交的任务，节约了线程这种有限而昂贵的资源。Thread Pool 模式还可以带来以下好处。

- **抵消线程创建的开销，提高响应性。**创建线程的消耗不仅包括线程对象本身以及其调用栈所需的内存空间，也包括创建依赖于 JVM 宿主机操作系统的本地线程。因此，线程创建是一个昂贵的操作。Thread Pool 模式通过事先创建一部分线程使得被提交的任务可以立即被执行而无须等待工作者线程的创建，从而提高了响应性。另外，由于一个工作者

线程可以为多个任务服务，因此创建工作者线程的开销被平摊到其执行的所有任务中。一个工作者线程执行的任务越多，那么其创建的开销就越明显地被抵消。

- **封装了工作者线程生命周期管理**。线程池本身负责了其工作者线程的生命周期的管理，包括何时创建工作者线程、创建多少工作者线程以及何时销毁工作者线程。这使得 Thread Pool 模式的客户端代码往往只需要关心任务的提交及其处理结果，而无须关心工作者线程的生命周期。如果我们不再需要某个线程池，那么只需要客户端代码调用线程池实例的 shutdown 方法即可。
- **减少销毁线程的开销**。JVM 销毁一个已停止的线程也有其时间上的开销。Thread Pool 模式使我们避免了频繁地创建工作者线程，因此避免了频繁地销毁已停止的线程，从而减少了相应开销。

虽然理解和使用 Thread Pool 模式相对简单，但在实战中使用 Thread Pool 模式却涉及比较多的风险和问题。由于自行实现 Thread Pool 模式比较容易出错，并且 Java 自 JDK 1.5 开始已经提供了 ThreadPoolExecutor 这个线程池实现类。因此，下面我们讨论的有关使用 Thread Pool 模式的风险和问题都是基于 ThreadPoolExecutor 这个线程池实现类的。

9.4.1 工作队列的选择

工作队列通常可以在有界队列（Bounded Queue）、无界队列（Unbound Queue）和直接交接队列（SynchronousQueue）之间选择。

以无界队列（如 LinkedBlockingQueue）作为工作队列，虽然工作队列本身并不限制线程池中等待运行的任务的数量，但工作队列中实际可容纳的任务数取决于任务本身对资源的使用情况。例如，线程池的客户端代码在创建向线程池提交的任务对象（Runnable）的时候同时创建了该任务对象所需引用的其他对象，而这些被引用的对象占用的内存空间比较大。这样一来，随着工作队列中这样的任务对象越来越多，这些任务对象所导致的内存占用也越来越多（这些任务还没有被执行，因此其占用的内存空间不能被垃圾回收）。极端的情况下，这种情形还能导致 JVM 内存溢出，从而影响了这个 Java 应用程序，而不仅仅是使用该线程池的对象。因此，无界工作队列可能导致系统的不稳定，适合在任务占用的内存空间以及其他稀缺资源比较少的情况下使用。

如果应用程序确实需要比较大的工作队列容量，而又想避免无界工作队列可能导致的问题，不妨考虑 SynchronousQueue。SynchronousQueue 实现上并不使用缓存空间。由于 ThreadPoolExecutor 内部实现任务提交的时候调用的是工作队列（BlockingQueue 接口的实现类）的非阻塞式入队列方法（offer 方法），因此，在使用 SynchronousQueue 作为工作队列的前提下，客户端代码向线程池提交任务时，而线程池中又没有空闲的线程能够从

SynchronousQueue 队列实例中取一个任务，那么相应的 offer 方法调用就会失败（即任务没有被存入工作队列）。此时，ThreadPoolExecutor 会新建一个新的工作者线程用于对这个入队列失败的任务进行处理（假设此时线程池的大小还未达到其最大线程池大小）。所以，使用 SynchronousQueue 作为工作队列，工作队列本身并不限制待执行的任务的数量。但此时需要限定线程池的最大大小为一个合理的有限值，而不是 Integer.MAX_VALUE，否则可能导致线程池中的工作者线程的数量一直增加到系统资源所无法承受为止。本章案例就是采用了这种方法来配置其所用的线程池。

以有界队列（如 ArrayBlockingQueue、有界的 LinkedBlockingQueue）作为工作队列则可以限定线程池中待执行任务的数量，这在一定程度上可以限制资源的消耗。通常，使用有界队列作为工作队列需要指定线程池的最大大小为一个合理有限值，而不是 Integer.MAX_VALUE。其理由类似上面使用 SynchronousQueue 作为工作队列的情形。而有界工作队列加上有限数量的工作者线程则可能导致死锁。有关线程池的死锁问题，下文会提到。有界队列适合在提交给线程池执行的各个任务之间是相互独立（而非有依赖关系）的情况下使用。

9.4.2 线程池大小调校

线程池大小指线程池中的工作者线程的数量。线程池大小太大会消耗过多的资源，并增加上下文切换。线程池大小太小，又可能导致无法充分利用 CPU 资源，使任务处理的吞吐率过低。因此，实战中使用线程池时需要寻找一个合理（或者所谓最佳）的线程池大小。合理的线程池大小取决于该线程池所要处理的任务的特性、系统资源状况以及任务所使用的稀缺资源状况等因素。

系统资源状况主要考虑系统 CPU 个数以及 JVM 堆内存的大小。通常，线程池的大小不是硬编码（Hard-coded）在代码中，而是可配置的（如通过配置文件配置）或者动态计算出来的。动态计算出来的线程池大小通常是基于 CPU 个数计算的。在 Java 中我们可以调用 java.lang.Runtime 类的 availableProcessors 方法获取 JVM 宿主机 CPU 个数。下面为了讨论方便，我们用 N_{cpu} 表示系统的 CPU 个数。

任务的特性主要考虑任务是 CPU 密集型、I/O 密集型，还是混合型（同时包含较多计算和 I/O 操作）。对于 CPU 密集型任务，相应的线程池的大小可以考虑设置为 $N_{cpu}+1$。这里，之所以线程池的大小比 CPU 的个数还多 1 个，是因为考虑到即便是 CPU 密集型的任务其执行线程也可能在某一时刻由于某种原因，如缺页中断（Page Fault）而出现等待。此时，一个额外的线程可以继续使用 CPU 时间片。对于 I/O 密集型任务，相应的线程池大小可以考虑设置得相对大一点。这是因为 I/O 密集型任务执行过程中等待 I/O 的时间相对于其使用 CPU 的时间长，而处于 I/O 等待状态的线程并不会消耗 CPU 资源，因此相应的线程池的大小调成大于 N_{cpu} 的

数字，比如 $2\times N_{cpu}$。另外，对于 I/O 密集型任务我们需要注意 I/O 操作会引起上下文切换[1]。这就意味着进行 I/O 操作的线程越多由 I/O 操作引起的上下文切换也越多。因此，对于 I/O 密集型任务不妨将相应的线程池的核心线程池大小（Core Pool Size）设置为 1，并将其最大线程池大小（Maximum Pool Size）设置为 $2\times N_{cpu}$。这样，如果线程池只需要一个工作者线程就可以轻松处理提交给其的任务，那么此时由 I/O 操作导致的上下文切换是最少的；如果一个工作者线程无法满足任务处理的需要，那么 ThreadPoolExecutor 会逐渐增加工作者线程的数量，直到其达到 $2\times N_{cpu}$。清单 9-2 展示了一个设置 I/O 密集型任务相应的线程池大小的例子。对于混合型任务，可以将任务进行相应的分解，将其分解为 CPU 密集型和 I/O 密集型两种子任务，这些子任务采用各自的线程池执行。

清单 9-2. 设置 I/O 密集型任务相应的线程池的大小示例

```
public class ThreadPoolSize4IOIntensiveTask {

    public static void main(String[] args) {

        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
                //核心线程池大小为 1
                1,
                //最大线程池大小为 2*Ncpu
            Runtime.getRuntime().availableProcessors() * 2,
            60,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(200));

        threadPool.submit(new IOIntensiveTask());
    }

    // 某个 I/O 密集型任务
    private static class IOIntensiveTask implements Runnable {

        @Override
        public void run() {

            // 执行大量的 I/O 操作

        }

    }

}
```

尽管所谓合理或者最佳的线程池大小本身也是不精确的，但是，上述方法还是有些理想化。不过其好处是容易实施。实际上，商用软件往往会规定某个软件在其运行过程中对 CPU 的使用率不能超过某个限定值（如 75%）。因此，如果要进一步“精确”地设置线程池的大小，我们可能需要考虑目标 CPU 使用率，即我们期望软件运行过程中会保持多少平均 CPU 使用率。

1 确切地说是阻塞式 I/O 会引起上下文切换。

另外，如果任务本身是混合型而又不太好将其拆分成 CPU 密集型和 I/O 密集型的子任务的话，我们也可以考虑不拆分，只是在设置线程池大小的时候要将该任务涉及的等待时间，如等待外部服务器的 HTTP 响应，以及该任务真正执行计算（使用了 CPU 时间片）的时间，考虑进来。

下面给出一个计算线程合理大小的公式，如下所示。

$$S = N_{cpu} \times U_{cpu} \times (1 + \frac{WT}{ST})$$

上述公式中，S 为线程池的合理大小，N_{cpu} 为 CPU 个数，U_{cpu} 为目标 CPU 使用率，WT 为任务执行线程进行等待的时间，ST 为任务执行线程使用 CPU 进行计算的时间。其中，WT 和 ST 这个时间值可以借助工具（如 jvisualvm）计算出相应值。

另外，任务执行过程中使用到的一些稀缺资源，如数据库连接，也会对线程池的合理大小产生影响。比如，与数据库打交道的任务其执行过程中需要获取数据库连接。虽然数据库连接多数是从数据库连接池中取得的，而不是每次执行这种任务都新建一个数据库连接，但是从数据库服务器的角度来看，一个数据库服务器能够对外提供的数据库连接始终是有限的。在应用程序采用集群部署的情况下，可能有多台主机共同访问同一个数据库服务器。因此，相对于该集群环境中的一台主机而言，其能够使用的数据库连接资源就显得更加有限。所以，此情形下线程池的合理大小实际上还要受可用的数据库连接数的限制。

总而言之，设置线程池的合理大小不是一件能精确做到的事情。重要的是线程池的大小要可以配置，并且其配置值要考虑到系统可用 CPU 资源以及其他稀缺资源等因素。上述关于调整线程池大小的方法为我们合理设置线程池的大小提供了一个参考，使得线程池大小调校有了一个起点和依据。

9.4.3 线程池监控

线程池的大小、工作队列的容量、线程空闲时间限制这些线程池的属性虽然我们可通过配置的方式进行指定（而不是在代码中硬编码），但是所指定的值是否恰当就需要通过监控来判断。例如，如果我们选择有界队列作为工作队列，那么这个队列的容量以多少为宜呢，这需要在软件测试过程中对线程池进行监控来确定。另外，考虑到测试环境和软件实际运行环境总是存在差别的，出于软件运维的考虑我们也可能需要对线程池进行监控。ThreadPoolExecutor 类提供了对线程池进行监控的相关方法，如表 9-1 所示。

表 9-1. ThreadPoolExecutor 提供的线程池监控相关方法

方　法	用　途
getPoolSize()	获取当前线程池大小
getQueue()	返回工作队列实例，通过该实例可获取工作队列的当前大小
getLargestPoolSize()	获取工作者线程数曾经达到的最大数，该数值有助于确认线程池的最大大小设置是否合理
getActiveCount()	获取线程池中当前正在执行任务的工作者线程数（近似值）
getTaskCount()	获取线程池到目前为止所接收到的任务数（近似值）
getCompletedTaskCount()	获取线程池到目前为止已经处理完毕的任务数（近似值）

9.4.4 线程泄漏

线程泄漏（Thread Leak）指线程池中的工作者线程意外中止，使得线程池中实际可用的工作者线程变少。如果线程泄漏持续存在，那么线程池中的工作者线程会越来越少，最终使得线程池无法处理提交给其的任务。线程泄漏通常是由于线程对象的 run 方法中异常处理没有捕获 RuntimeException 和 Error 导致 run 方法意外返回，使得相应线程提前中止。尽管 Java 中的 ThreadPoolExecutor 已经对线程泄漏进行了预防，但是，实战中我们需要注意另外一种可以事实上造成线程泄漏的场景：如果线程池中的某个工作者线程执行的任务涉及外部资源等待，如等待网络 I/O，而该任务又没有对这种等待指定时间限制。那么，外部资源如果一直没有返回该任务所等待的结果，就会导致执行该任务的工作者线程一直处于等待状态而无法执行其他任务，这就形成了事实上的线程泄漏。

9.4.5 可靠性与线程池饱和处理策略

如果我们在创建 ThreadPoolExecutor 实例的时候指定了有界队列作为工作队列，那么当线程池中的工作队列满，并且工作者线程数量已达到最大工作者线程数（线程池的最大大小）时，线程池就处于饱和状态。此时，新提交给线程池的任务就会被拒绝（无法提交成功）。但是，从线程池的客户端代码的角度来看，其为了提高计算的可靠性，必需考虑如何应对这种任务提交被拒绝的情形，即线程池饱和处理策略。ThreadPoolExecutor 已经提供了线程池饱和处理策略的接口和一些预定义的实现类。

接口 java.util.concurrent.RejectedExecutionHandler 对线程池饱和处理策略进行了抽象，其声明如清单 9-2 所示。当一个提交给 ThreadPoolExecutor 实例的任务被拒绝时，相应的 ThreadPoolExecutor 实例会调用其 RejectedExecutionHandler 实例的 rejectedExecution 方法以执行线程池饱和处理策略。JDK 提供了该接口的几个实现类，如清单 9-3 所示。

清单 9-3. RejectedExecutionHandler 接口声明

```
public interface RejectedExecutionHandler {

    /**
     *
     * @param r 提交失败的任务
     * @param 试图执行任务的线程池
     * @throws RejectedExecutionException
     */
    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

从表 9-2 可以看出，JDK 中预置的线程池饱和处理策略中只有 ThreadPoolExecutor.CallerRunsPolicy 能够实现补救，即提交失败的任务可以获得再次（或者更多）执行的机会。其他的饱和处理策略都是放弃提交失败的任务，而 ThreadPoolExecutor.AbortPolicy 则是 ThreadPoolExecutor 默认的线程池饱和处理策略。如果使用 ThreadPoolExecutor.CallerRunsPolicy 作为线程池饱和处理策略，需要注意它可能会引起线程安全问题。假设某种任务其执行过程中使用了非线程安全对象，并且不需要与其他线程共享任何对象，此时我们完全可以考虑使用最大工作者线程数为 1 的 ThreadPoolExecutor 实例来执行这些任务。此时，这些任务的相关代码可以采用单线程的方式去编写，而无须考虑数据同步和线程安全。这种情况下，如果我们采用 ThreadPoolExecutor.CallerRunsPolicy 作为该 ThreadPoolExecutor 实例的线程饱和处理策略则可能引起线程安全问题。这是因为此情形下提交失败的任务会通过 ThreadPoolExecutor.CallerRunsPolicy 实例被客户端线程重新执行，而客户端线程与该 ThreadPoolExecutor 实例中的唯一的工作者线程可能形成两个并发的线程，从而引发线程安全。

表 9-2. JDK 提供的线程池饱和处理策略实现类

实 现 类	所实现的处理策略
ThreadPoolExecutor.AbortPolicy	直接抛出异常
ThreadPoolExecutor.DiscardPolicy	丢弃当前被拒绝的任务（而不抛出任何异常）
ThreadPoolExecutor.DiscardOldestPolicy	将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务
ThreadPoolExecutor.CallerRunsPolicy	在客户端线程中执行被拒绝的任务

当然，我们也可以采用其他支持补救的线程池饱和处理策略。比如，本章案例中采用的人工

重试的补救办法。本章案例中（代码参见清单 9-1），我们创建线程池实例 EXECUTOR 时通过 ThreadPoolExecutor 的构造器的第 7 个参数指定的线程池饱和处理策略是 ThreadPoolExecutor.DiscardPolicy，这意味着某个发送验证码短信的任务提交失败时该任务就直接被抛弃。尽管如此，我们可以稍后由人工发起一个重试（由要接收验证码短信的用户通过网页操作）重新提交一个验证码发送的任务。此外，我们也可以考虑自行实现自动重试的处理策略。清单 9-4 展示了一个自定义的线程池饱和处理策略，它支持将提交失败的任务重新放入线程池的工作队列等待处理。当然，由于线程池饱和时其工作队列是满的，对工作队列的阻塞式入队列方法（put 方法）的调用需要等待到相应队列非满时才能返回，因此该处理策略可能导致线程池的客户端线程因等待线程池工作队列非空而被阻塞。如果提交给线程池的任务执行过程中又会提交新的任务给同一个线程池，而这两个任务之间又有依赖关系，则该处理策略可能导致线程池死锁。

清单 9-4. 支持将提交失败的任务重新入工作队列的线程池饱和处理策略

```
/**
 * 该线程池饱和处理策略支持将提交失败的任务重新放入线程池工作队列。
 *
 * @author Viscent Huang
 *
 */
public class ReEnqueueRejectedExecutionHandler implements
    RejectedExecutionHandler {

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        if (executor.isShutdown()) {
            return;
        }

        try {
            executor.getQueue().put(r);
        } catch (InterruptedException e) {
            ;
        }

    }

}
```

如果我们采用无界队列作为线程池的工作队列，那么上述的线程池饱和处理策略就不适用了，因为这种情况下提交给线程池的任务总是可以被放入工作队列而不会被拒绝（除非 ThreadPoolExecutor 实例已经被关闭）。尽管无界工作队列本身没有存储限制，但是队列中存储的对象所占用的资源，如内存空间，是有限的。因此，此时我们也需要考虑线程池事实上的饱和状态，即线程池究竟可以容纳多少待处理的任务。

9.4.6 死锁

如果线程池中执行的任务在其执行过程中又会向同一个线程池提交另外一个任务，而前一个任务的执行结束又依赖于后一个任务的执行结果，那么当线程池中所有的线程都处于这种等待其他任务的处理结果，而这些线程所等待的任务仍然还在工作队列中的时候，由于线程池已经没有可以对工作队列中的任务进行处理的工作者线程，这种等待就会一直持续下去而形成死锁（Deadlock）。

因此，适合提交给同一线程池实例执行的任务是相互独立的任务，而不是彼此有依赖关系的任务。

要执行彼此有依赖关系的任务可以考虑将不同类型的任务交给不同的线程池实例执行，或者对负责任务执行的线程池实例进行如下配置。

配置 1：设置线程池的最大大小为一个有限值，而不是默认值 Integer.MAX_VALUE。

配置 2：使用 SynchronousQueue 作为工作队列。

配置 3：使用 ThreadPoolExecutor.CallerRunsPolicy 作为线程池饱和处理策略。

采用上述配置的线程池之所以能够避免死锁，是因为：当线程池中的一个工作者线程（ThreadA）执行某个任务（TaskA）时，该任务向同一个线程池提交了另外一个任务（TaskB），而 TaskA 的执行结束依赖于 TaskB 的处理结果。若此时线程池已满（这有赖于上述配置 1），则 TaskB 入工作队列会失败（这有赖于上述配置 2）。这时，TaskB 就由于线程池饱和（工作者队列“满”并且线程池也满）而被拒绝。但巧妙的一点是，此时 TaskB 可以由当前线程池中提交该任务的工作者线程（即 ThreadA）自身去执行（这有赖于上述配置 3），即存在依赖关系的两个任务 TaskA 和 TaskB 此时由线程池中的同一个工作者线程执行，因此避免了死锁，如清单 9-5 所示。

清单 9-5. 避免线程池死锁的一种方法

```
public class ThreadPoolDeadLockAvoidance {
    private final ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
            1,
            //最大线程池大小为 1（有限数值）:
            1,
        60, TimeUnit.SECONDS,
        //工作队列为 SynchronousQueue:
        new SynchronousQueue<Runnable>(),
        //线程池饱和处理策略为 CallerRunsPolicy:
        new ThreadPoolExecutor.CallerRunsPolicy());

    public static void main(String[] args) {
        ThreadPoolDeadLockAvoidance me = new ThreadPoolDeadLockAvoidance();
```

```
        me.test("<This will NOT deadlock>");
    }

    public void test(final String message) {
        Runnable taskA = new Runnable() {

            @Override
            public void run() {
                System.out.println("Executing TaskA...");
                Runnable taskB = new Runnable() {

                    @Override
                    public void run() {
                        System.out.println("TaskB processes " + message);
                    }

                };
                Future<?> result = threadPool.submit(taskB);

                try {
        // 等待 TaskB 执行结束才能继续执行 TaskA，使 TaskA 和 TaskB 成为有依赖关系的两个任务
                    result.get();
                } catch (InterruptedException e) {
                    ;
                } catch (ExecutionException e) {
                    e.printStackTrace();
                }

                System.out.println("TaskA Done.");

            }

        };

        threadPool.submit(taskA);
    }

}
```

9.4.7 线程池空闲线程清理

线程池中长期处于空闲状态（即没有在执行任务）的工作者线程会浪费宝贵的线程资源。因此，清理一部分这样的线程可以节约有限的资源。ThreadPoolExecutor 支持将其核心工作者线程以外的空闲线程进行清理。创建 ThreadPoolExecutor 实例时，我们可以在其构造器的第 3、4 个参数中指定一个空闲持续时间。核心工作者线程以外的工作者线程空闲了指定时间以后，ThreadPoolExecutor 就可以将其清理掉。

9.5 Thread Pool 模式的可复用实现代码

Java 标准库类 java.util.concurrent.ThreadPoolExecutor 就是 Thread Pool 模式的一个可复用的实现。利用 ThreadPoolExecutor 实现 Thread Pool 模式，应用代码只需要完成以下几件事情。

1.【必需】创建一个 ThreadPoolExecutor 实例。根据应用程序的需要，创建 ThreadPoolExecutor 实例时指定一个合适的线程池饱和处理策略。

2.【必需】创建 Runnable 实例用于表示待执行的任务，并调用 ThreadPoolExecutor 实例的 submit 方法提交任务。

3.【可选】使用 ThreadPoolExecutor 实例的 submit 方法返回值获取相应任务的执行结果。

Thread Pool 模式在实现上需要考虑比较多的问题与风险，因此我们尽量不要自己去实现该模式，而是要使用 JDK 中的现成类 ThreadPoolExecutor。

9.6 Java 标准库实例

Java Swing 中类 javax.swing.SwingWorker 可用于执行耗时较长的任务。该类使用了 Thread Pool 模式。SwingWorker 内部维护了一个线程池（ThreadPoolExecutor 实例），该线程池包含了若干个工作者线程用于执行提交给 SwingWorker 的任务。

9.7 相关模式

9.7.1 Two-phase Termination 模式（第 5 章）

Thread Pool 模式可能会使用 Two-phase Termination 模式来实现其工作者线程的停止。

9.7.2 Promise 模式（第 6 章）

Thread Pool 模式中的 Promise 参与者相当于 Promise 模式中的 Promise 参与者。

9.7.3 Producer-Consumer 模式（第 7 章）

Thread Pool 模式可以看成是 Producer-Consumer 模式的一个实例。Thread Pool 模式中的

ThreadPool 参与者和 WorkerThread 参与者分别相当于 Producer-Consumer 模式的 Producer 参与者和 Consumer 参与者。

9.8 参考资源

1．Java theory and practice: Thread pools and work queues. http.//www.ibm.com/developerworks/library/j-jtp0730/index.html.

2．Brian Göetz et al.Java Concurrency In Practice.Addison Wesley,2006.

3．ThreadPoolExecutor 类源码. http://grepcode.com/file/repository.grepcode.com/ java/ root/ jdk/openjdk/6-b27/java/util/concurrent/ThreadPoolExecutor.java#ThreadPoolExecutor.

第 10 章

Thread Specific Storage（线程特有存储）模式

10.1 Thread Specific Storage 模式简介

多线程相关的问题如线程安全、死锁等归根究底是多线程共享变量导致的。有鉴于此，Thread Specific Storage 模式通过不共享变量实现了线程安全，并由此自然地避免了与锁的消耗及与之有关的问题。

这里，读者可能会想到只要将线程需要访问的变量作为线程对象的私有实例变量，即可实现各个线程访问各自的变量，相互间不共享变量，如清单 10-1 所示。

清单 10-1. 用线程对象的私有实例变量实现线程间不共享变量

```
public class ThreadPrivateMember {

    public static void main(String[] args) throws InterruptedException {
        XThread thread;
        for (int i = 0; i < 3; i++) {
            thread = new XThread("message-" + i);
            thread.start();
        }

        Thread.sleep(50);

    }

    private static class XThread extends Thread {
        private final String message;

        public XThread(String message) {
            this.message = message;
        }
```

```
        @Override
        public void run() {
            for (int i = 0; i < 3; i++) {
                System.out.println(message);
            }
        }

    }
}
```

使用线程对象的私有变量这种方法可以看成是 Thread Specific Storage 模式的一个最小实现。但是，这种方法的通用性不够。毕竟代码之间调用关系并不总是这种 run 方法去访问线程的私有变量。

如果多个线程都需要使用某个类 TSObject，而为了实现线程安全和避免与锁相关的问题我们又不希望这些线程共享 TSObject 的实例。那么我们可以使这些线程中的每一个线程都获得一个且仅一个 TSObject 实例。这样的 TSObject 实例就被称为线程特有对象（Thread Specific Object），它只会被一个线程持有，该线程对其状态的修改不会影响到其他线程。并且，为了隐藏相关实现细节以便于这些线程获取所需的线程特有对象，我们引入一个线程特有对象的代理类 TSObjectProxy。不同的线程使用同一个 TSObjectProxy 实例可以获得所需的线程特有对象。这样就形成了一种效果：不同的线程使用统一的访问接入点（TSObjectProxy）可以获取该线程所特有的 TSObject 实例。这就是 Thread Specific Storage 模式的核心思想，如图 10-1 所示。

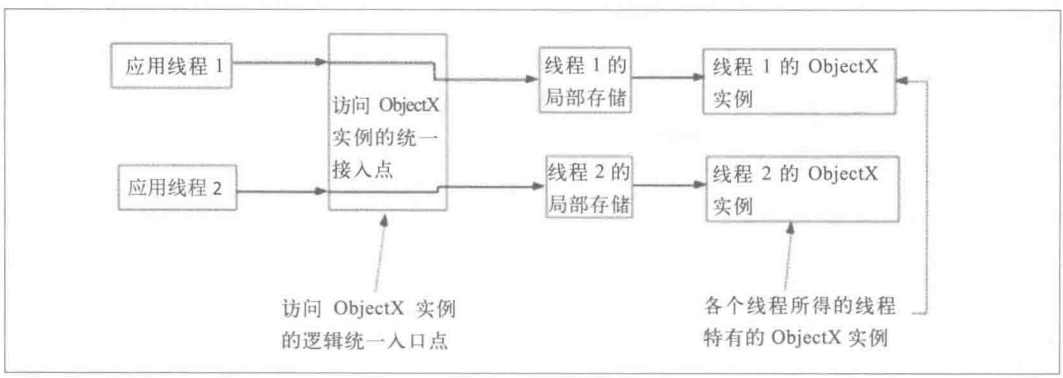


图 10-1. Thread Specific Storage 模式核心思想示意图

Thread Specific Storage 模式的核心思想有点像家长们聊天时称呼自己的小孩为我家宝宝的情形。虽然张三说我家宝宝是夜猫子，李四说我家宝宝晚上 10 点准时睡觉，但是作为听众的我们明白那个夜猫子是张三的孩子张小宝，而那个准时睡觉的是李四的孩子李大宝。

Java 1.2 引入的标准库类 java.lang.ThreadLocal 就相当于图 10-1 中的访问线程特有变量的逻辑

统一入口点。读者如果对 ThreadLocal 已经有所理解，也不妨继续往下看。本章其他节的内容也有助于读者深入理解 ThreadLocal，并了解在实战中应用 ThreadLocal 需要注意的重要问题。

10.2 Thread Specific Storage 模式的架构

Thread Specific Storage 模式的基础是为每个线程“装备”一个线程特有存储（Thread Specific Storage）。线程特有存储可以理解为一个 Map。该 Map 存放了若干个条目（Entry）。每个条目中的值（Value）为一个线程特有对象，键（Key）为用于获取相应线程特有对象的 TSObjectProxy 实例。客户端代码通过 TSObjectProxy 实例获取其所需的线程特有对象。逻辑上，每个线程都有其线程特有存储。因此，不同线程使用同一个 TSObjectProxy 实例可以获取到不同的线程特有对象实例。同一个线程使用不同 TSObjectProxy 实例可以获取不同的线程特有对象实例。

Thread Specific Storage 模式的主要参与者有以下几种。其类图如图 10-2 所示。

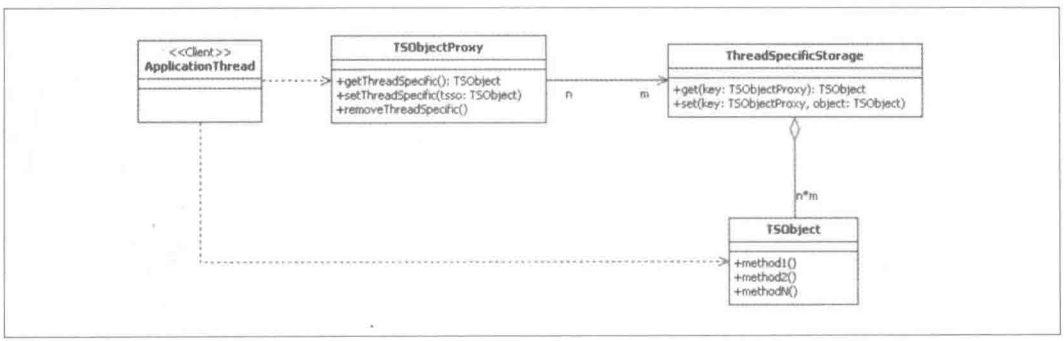


图 10-2. Thread Specific Storage 模式的类图

- **ApplicationThread**：Thread Specific Storage 模式的客户端，表示各个应用线程。
- **TSObjectProxy**：用于访问线程特有对象的代理对象。其主要方法及职责如下。
  - **getThreadSpecific**：获取与其所属 TSObjectProxy 实例相关联的线程特有对象实例。
  - **setThreadSpecific**：建立其所属 TSObjectProxy 实例与指定线程特有对象实例的关联。
  - **removeThreadSpecific**：删除其所属 TSObjectProxy 实例与线程特有对象实例的关联。
- **TSObject**：表示线程特有对象。具体的类型由应用决定。
- **ThreadSpecificStorage**：线程特有存储。该参与者实例可以理解为一个 Map，每个线程

都有这样一个 Map。这个 Map 存储了 TSObjectProxy 实例到 TSObject 实例的映射。其主要方法及职责如下。

- **get**：获取与指定 TSObjectProxy 实例关联的 TSObject 实例。
- **set**：设置指定 TSObjectProxy 实例与指定 TSObject 实例的关联关系。

Java 标准库中的类 java.lang.ThreadLocal 可以看成是 TSObjectProxy 参与者的一个通用实现。ThreadLocal 的类型参数 T 可以看成是 TSObject 参与者。

Thread Specific Storage 模式的序列图如图 10-3 所示。

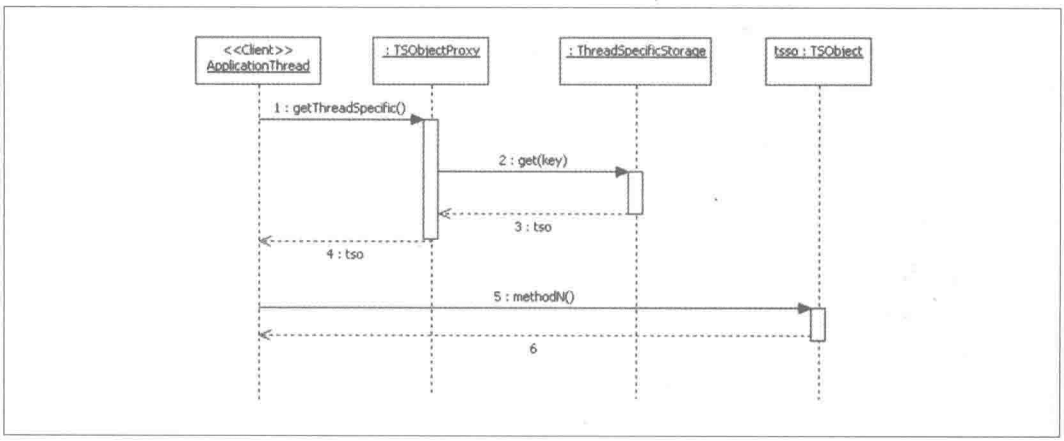


图 10-3. Thread Specific Storage 模式的序列图

第 1 步：客户端代码调用 TSObjectProxy 实例的 getThreadSpecific 方法。

第 2、3 步：getThreadSpecific 方法调用与当前线程关联的 ThreadSpecificStorage 实例的 get 方法，并得到其返回值 tso。

第 4 步：getThreadSpecific 方法将 tso 作为其返回值返回。

第 5、6 步：客户端代码调用 tso 的相关方法。

上述第 2、3 步是最为关键的一步，该步骤需要获取与当前线程关联的 ThreadSpecificStorage 实例。我们可以借助线程对象的私有实例变量（参见如清单 10-1 所示的代码）来实现 ThreadSpecificStorage 实例与某个线程的关联。ThreadLocal 类就是采用这种方法。

10.3 Thread Specific Storage 模式实战案例解析

某系统需要支持验证码短信功能。该系统的用户进行一些重要操作的时候，该系统会生成一个验证码，并将其通过短信发送给用户。验证码是一个 6 位数的随机数字。这里，为了提高安全性，验证码的生成需要使用 java.security.SecureRandom 这种强随机数生成器，而非 java.math.Random 这种伪随机数生成器。但是，使用 SecureRandom 可能会涉及以下几个问题。

一、SecureRandom 实例的初始化（主要是初始化种子）可能比较耗时间。这点在 JVM 的宿主机操作系统为 Linux 系统时更为明显。这与 SecureRandom 的内部实现有关。因此，我们希望能够复用 SecureRandom 实例，而不是每次需要生成一个验证码的时候就生成一个 SecureRandom 实例。

二、SecureRandom 用于生成随机整数的 nextInt 方法最终会调用一个由 SecureRandom 自身定义的 synchronized 方法。这意味着，nextInt 方法的调用实际上会涉及锁。因此，如果多个线程共用同一个 SecureRandom 实例，那么当一个线程正在调用该实例的 nextInt 方法生成随机数的时候，其他线程只能等待。但是，我们不希望看到这种等待：由于验证码生成后需要通过短信发送给用户，而该系统下发短信给用户涉及网络 I/O 这种相对慢的操作，我们希望验证码的生成能够尽量快，从而不耽误短信的下发。

尽管 SecureRandom 是线程安全的，但是综合分析上述两个问题，我们决定不在多个线程间共享 SecureRandom 实例，并且也不每次生成验证码的时候就创建一个 SecureRandom。这时，我们可以借用 Thread Specific Storage 模式：让每个需要生成验证码的线程生成一个且仅一个 SecureRandom 实例。

因此，本案例生成验证码时会使用如清单 10-2 所示的基于 Thread Specific Storage 模式的强随机数生成器代码。

清单 10-2. 基于 Thread Specific Storage 模式的强随机数生成器

```
public class ThreadSpecificSecureRandom {
    //该类的唯一实例
    public static final ThreadSpecificSecureRandom INSTANCE = new
        ThreadSpecificSecureRandom();

    /*
    SECURE_RANDOM 相当于模式角色：ThreadSpecificStorage.TSObjectProxy。
    SecureRandom 相当于模式角色：ThreadSpecificStorage.TSObject。
    */
    private static final ThreadLocal<SecureRandom> SECURE_RANDOM = new
        ThreadLocal<SecureRandom>() {

        @Override
```

```
        protected SecureRandom initialValue() {
            SecureRandom srnd;
            try {
                srnd = SecureRandom.getInstance("SHA1PRNG");
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
                srnd = new SecureRandom();
            }
            return srnd;
        }

    };

    // 私有构造器
    private ThreadSpecificSecureRandom() {

    }

    public int nextInt(int upperBound) {
        SecureRandom secureRnd = SECURE_RANDOM.get();
        return secureRnd.nextInt(upperBound);
    }

    public void setSeed(long seed) {
        SecureRandom secureRnd = SECURE_RANDOM.get();
        secureRnd.setSeed(seed);
    }

}
```

清单 10-2 中的类 ThreadSpecificSecureRandom 是一个单例类。尽管该类的唯一实例会被多个需要生成验证码的线程所共享，但是这些线程调用其 nextInt 方法的时候所用到的 SecureRandom 实例却是每个线程各自持有的线程特有对象。

客户端使用 ThreadSpecificSecureRandom 类来生成验证码并下发验证码短信的代码如清单 10-3 所示。

清单 10-3. 基于 Thread Specific Storage 模式的强随机数生成器客户端代码

```
public class SmsVerficationCodeSender {
    private static final ExecutorService EXECUTOR = new ThreadPoolExecutor(1,
        Runtime.getRuntime().availableProcessors(), 60, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r, "VerfCodeSender");
                t.setDaemon(true);
                return t;
            }

        }, new ThreadPoolExecutor.DiscardPolicy());

    public static void main(String[] args) {
        SmsVerficationCodeSender client = new SmsVerficationCodeSender();
```

```
        client.sendVerificationSms("18912345678");
        client.sendVerificationSms("18712345679");
        client.sendVerificationSms("18612345676");

        try {
       Thread.sleep(100);
       } catch (InterruptedException e) {
           ;
           }
       }

    public void sendVerificationSms(final String msisdn) {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                //生成强随机数验证码
                int verificationCode = ThreadSpecificSecureRandom.INSTANCE
                    .nextInt(999999);
                DecimalFormat df = new DecimalFormat("000000");
                String txtVerCode = df.format(verificationCode);

                //发送验证码短信
                sendSms(msisdn, txtVerCode);
            }

        };

        EXECUTOR.submit(task);

    }

    private void sendSms(String msisdn, String verificationCode) {
        System.out.println("Sending verification code " + verificationCode + " to "
            + msisdn);

        // 省略其他代码
    }

}
```

10.4 Thread Specific Storage 模式的评价与实现考量

Thread Specific Storage 模式提升了计算效率。Thread Specific Storage 模式使得我们可以在不使用锁的情况下实现线程安全，从而避免了锁的开销以及由锁带来的相关问题，如上下文切换、死锁等。

Thread Specific Storage 模式易于使用。Thread Specific Storage 模式通过引入 TSObjectProxy 这个参与者隐藏了其相关实现细节，客户端代码只需要与 TSObjectProxy 参与者实例打交道即可获取所需的线程特有对象。

Thread Specific Storage 模式有以下几个弊端。

Thread Specific Storage 模式隐藏了系统结构。Thread Specific Storage 模式的 TSObjectProxy 参与者一方面使得客户端代码变得简单，另一方面也隐藏了应用中的各个对象间的关系，从而可能使应用更加难于理解。

Thread Specific Storage 模式鼓励使用全局对象。Thread Specific Storage 模式的客户端代码通常是多个线程共用一个 TSObjectProxy（ThreadLocal）实例获取其所需的线程特有对象。这个共用的 TSObjectProxy 实例就相当于一个全局变量，而全局变量的使用与目前广为接受的观念是相左的。

Thread Specific Storage 模式的常见使用场景包括以下几个。

场景一、需要使用非线程安全对象，但又不希望引入锁：如果多个线程需要使用非线程安全的对象，而我们又不希望该对象被多个线程共享，因为共享往往意味着需要引入锁以保证线程安全。此时可以使用 Thread Specific Storage 模式，使得各个线程拥有其特有的非线程安全对象实例。该场景的一个典型例子是将非线程安全的 SimpleDateFormat 实例用 ThreadLocal 包装，以供多个线程使用而不必引入锁，如清单 10-4 所示。

清单 10-4. 基于 Thread Specific Storage 模式实现 SimpleDateFormat 的线程安全

```
public class ThreadSpecificDateFormat {
    private static final ThreadLocal<SimpleDateFormat> TS_SDF = new
        ThreadLocal<SimpleDateFormat>() {

        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat();
        }

    };

    public static Date parse(String timeStamp, String format)
        throws ParseException {
        final SimpleDateFormat sdf = TS_SDF.get();
        sdf.applyPattern(format);
        Date date = sdf.parse(timeStamp);
        return date;
    }

    public static void main(String[] args) throws ParseException {
        Date date = ThreadSpecificDateFormat.parse("20150501123040",
            "yyyyMMddHHmmss");
        System.out.println(date);

    }

}
```

场景二、使用线程安全对象，但希望避免其使用的锁的开销和相关问题：线程安全的对象虽然可以被多个线程共享，但是由于其可能使用了锁来保证线程安全[1]，而某些情况下我们可能不希望看到锁的开销以及由锁可能引起的相关问题（如死锁）。此时，我们可以将线程安全的对象看成非线程安全的对象来应用 Thread Specific Storage 模式。因此，这种场景就转化成场景一。只不过，此时使用 Thread Specific Storage 模式的主要意图在于避免锁的开销，当然线程安全也是有保障的。本章案例对 Thread Specific Storage 模式的使用就属于这种场景。

场景三、隐式参数传递：线程特有对象在一个具体的线程中，它是线程全局可见的。某个类的方法中设置的线程特有对象对于该方法调用的其他类的方法也是可见的。这就可以形成隐式传递参数的效果，即一个类的方法调用另一个类的方法时，前者向后者传递数据可以借助 ThreadLocal 而不必通过方法参数传递。不过，也有的观点认为隐式参数传递使得系统难于理解。如清单 10-5 所示代码展示了这种使用场景。

场景四、特定于线程的单例（Singleton）模式：广为使用的单例模式所实现的是，对于一个 JVM 中的一个类加载器而言，某个类有且仅有一个实例。如果对于某个类，希望每个线程有且仅有该类的一个实例，那么就可以使用 Thread Specific Storage 模式。Thread Specific Storage 模式中，同一个应用线程多次调用同一个 TSObjectProxy 实例所得到的线程特有对象实例都是同一个实例（只要该线程没有调用 TSObjectProxy 实例的 setThreadSpecific 方法替换过线程特有对象实例）。

清单 10-5. 使用 Thread Specific Storage 模式实现参数传递的效果

```
public class ImplicitParameterPassing {

    public static void main(String[] args) throws InterruptedException {
        ClientThread thread;
        BusinessService bs = new BusinessService();
        for (int i = 0; i < Integer.valueOf(args[0]); i++) {
            thread = new ClientThread("test", bs);
            thread.start();
            thread.join();
        }

    }

}

class ClientThread extends Thread {
    private final String message;
    private final BusinessService bs;
    private static final AtomicInteger SEQ = new AtomicInteger(0);
```

1 第 3 章中介绍的 Immutable Object 模式可以在不使用锁的情况下保证线程安全，所以线程安全的对象不一定就涉及锁。

```
    public ClientThread(String message, BusinessService bs) {
        this.message = message;
        this.bs = bs;
    }

    @Override
    public void run() {
        Context.INSTANCE.setTransactionId(SEQ.getAndIncrement());
        bs.service(message);
    }

}

class Context {
    private static final ThreadLocal<Integer> TS_OBJECT_PROXY = new
        ThreadLocal<Integer>();

    // Context 类的唯一实例
    public static final Context INSTANCE = new Context();

    // 私有构造器
    private Context() {

    }

    public Integer getTransactionId() {
        return TS_OBJECT_PROXY.get();
    }

    public void setTransactionId(Integer transactionId) {
        TS_OBJECT_PROXY.set(transactionId);
    }

    public void reset() {
        TS_OBJECT_PROXY.remove();
    }

}

class BusinessService {

    public void service(String message) {
        int transactionId = Context.INSTANCE.getTransactionId();
        System.out.println("processing transaction " + transactionId
            + "'s message:" + message);
    }
}
```

实际使用 Thread Specific Storage 模式需要注意以下几个重要的问题。为方便起见，下面的讨论都是基于使用 ThreadLocal 作为 Thread Specific Storage 模式的实现。

10.4.1 线程池环境下使用 Thread Specific Storage 模式

在线程池环境下使用 Thread Specific Storage 模式需要谨慎。这是因为线程池环境下使用线程

特有对象可能导致数据错乱的现象。线程池环境下，一个工作者线程往往先后用于执行多个不同的任务。对于线程池中的一个工作者线程而言，如果它所执行的一个任务更换或者修改了该线程的线程特有对象，那么该工作者线程继续执行其他任务的时候，这些任务所“看到”的是一个新的线程特有对象。这种场景下，如果线程特有对象的状态与当前工作者线程所处理的任务有关，则会导致其他任务在执行的时候“看到”了本不属于它们的数据，产生了错乱。

因此，在线程池环境下使用线程特有对象需要考虑在适当的时间和地方清理线程特有对象，以便同一个工作者线程在处理其他任务的时候不会产生数据错乱。清理线程特有对象只需要调用获取相应线程特有对象实例时所用的 ThreadLocal 实例的 remove 方法即可。当然，如果我们进一步分析就不难发现，这种情形下使用 Thread Specific Storage 模式所得到的线程特有对象其实更像是“任务特有对象”。因为每个任务都需要自己的一个线程特有对象实例。

另一方面，线程池环境下使用 Thread Specific Storage 模式也不一定就会产生数据错乱。本章案例（相关代码参见清单 10-3）就是个例子。只不过，该案例涉及的线程特有对象 ThreadSpecificSecureRandom 是用于生成随机数字的，客户端代码只关心其生成的数字是否是随机的，而并不关心具体是哪个 ThreadSpecificSecureRandom 实例负责产生随机数字。因此，这种情形下，即便是在线程池环境中，使用 Thread Specific Storage 模式也不会产生数据错乱问题。

10.4.2 内存泄漏与伪内存泄漏

内存泄漏（Memory Leak）指由于对象永远无法被垃圾回收导致其占用的 JVM 内存无法被释放。持续的内存泄漏会导致 JVM 可用内存逐渐减少，并最终可能导致 JVM 内存溢出（Out of Memory）直到 JVM 宕机。

伪内存泄漏（ Memory Pseudo-leak）类似于内存泄漏。所不同的是，伪内存泄漏中对象所占用的内存在其不再被使用后的相当长的时间仍然无法被回收，甚至可能永远无法被回收。也就是说，伪内存泄漏中对象占用的内存空间可能会被回收，也可能永远无法被回收（此时，就变成了内存泄漏）。

在服务器环境（如 Apache Tomcat）下使用 Thread Specific Storage 模式需要注意 ThreadLocal 可能导致内存泄漏和伪内存泄漏。下面我们分析 ThreadLocal 导致内存泄漏与伪内存泄漏的场景和原因，并在此基础上给出相应的参考解决方案。

首先，我们先看下 ThreadLocal 是如何实现 Thread Specific Storage 模式的。JDK 标准库类 java.lang.Thread 的实例变量 threadLocals 会引用一个 ThreadLocal.ThreadLocalMap 实例。该实例相当于 Thread Specific Storage 模式的 ThreadSpecificStorage 参与者实例，从效果上看可以

将其理解为一个 WeakHashMap。该 WeakHashMap 包含若干个条目（Entry）。每个条目的键（Key）为指向一个 ThreadLocal 实例的弱引用（Weak Reference），值（Value）指向线程特有对象（TSObject）实例，如图 10-4 所示。

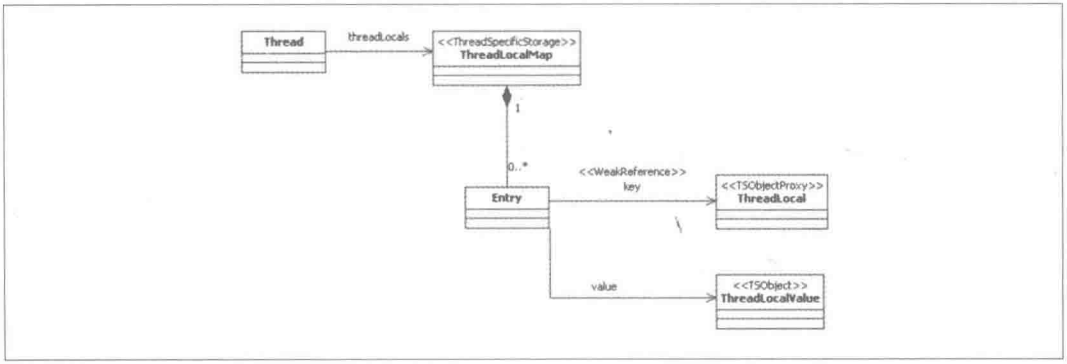


图 10-4. ThreadLocal 对 Thread Specific Storage 模式的实现

ThreadLocalMap 的条目对 ThreadLocal 实例的引用是弱引用，因此它不会阻止垃圾回收器（Garbage Collector）将其引用的 ThreadLocal 实例回收。而 ThreadLocalMap 的条目对线程特有对象实例的引用是强引用（Strong Reference），因此它会阻止垃圾回收器将其引用的线程特有对象实例回收。

对于某个 ThreadLocal 实例而言，如果在某一个时间段内该实例除了 ThreadLocalMap 条目对其有可达的引用（Reachable Reference）外，没有其他可达的引用，那么垃圾回收器就可以将该 ThreadLocal 实例回收。此时，先前引用该 ThreadLocal 实例的 ThreadLocalMap 条目由于其 Key 的值变为 null，就成了一条无效的条目（Stale Entry）。这种无效的条目（即 Key 值为 null，Value 值指向一个线程特有对象）在其所属的 ThreadLocalMap 有新增条目时可能被删除掉[2]。如果某个线程引用的 ThreadLocalMap 实例产生无效条目后，某段时间内该线程处于非运行状态，则该线程引用的 ThreadLocalMap 实例就没有新增的条目，因此其所有的无效条目在该时间段内也无法被删除。如果该线程一直处于非运行状态，则该线程引用的 ThreadLocalMap 实例的无效条目永远无法被删除。所以，这种情形就会导致伪内存泄漏。

对于某个 ThreadLocal 实例而言，如果在某一时间段内系统中除了 ThreadLocalMap 条目对其有可达的引用（弱引用）外，还有其他可达的强引用，那么，相应的 ThreadLocalMap 条目不会被删除，再加上 ThreadLocalMap 条目对线程特有对象的引用是强引用，此时只要引用该 ThreadLocalMap 实例的线程存在，则该条目会阻止垃圾回收器将相应的线程特有对象实例回

2 这是从效果的角度来说的，要了解 ThreadLocalMap 的实际处理可参考 ThreadLocal 类的源码。

收。如果引用该 ThreadLocalMap 条目的线程永远存在，则该 ThreadLocalMap 条目引用的 ThreadLocal 实例及其对应的线程特有对象实例都无法被垃圾回收，这就产生了内存泄漏。

下面以 Apache Tomcat 6.0.37 为例分析产生内存泄漏和伪泄漏的具体场景。

Tomcat 服务器内部会维护一个线程池。该线程池中的工作者线程用于处理各个 Web 应用程序的请求。因此，某个 Tomcat 服务器中部署的一个 Web 应用程序停止（如停止应用和卸载应用）后，负责请求处理的线程池中的各个工作者线程仍然存在。

如清单 10-6 所示代码展示了一个使用 ThreadLocal 导致内存泄漏的例子。MemoryLeakingServlet 类的 ThreadLocal 变量 TL_COUNTER 对应的线程特有对象类型 Counter 是一个由应用自身定义的类。Tomcat 中，应用自身定义的类由 Tomcat 的类加载器（Class Loader）WebAppClassLoader 负责加载。Java 中，某个类的实例会持有对该类（类本身也是一个对象）的引用。而类对象又会持有对其进行加载的类加载器对象的引用。因此，Counter 实例会持有对 WebAppClassLoader 的可达引用。并且，Java 中的类加载器会持有对其所加载的所有类的引用。MemoryLeakingServlet 类也是由 WebAppClassLoader 负责加载的，因此 WebAppClassLoader 会持有对 MemoryLeakingServlet 类的引用，而 MemoryLeakingServlet 类通过其静态变量 TL_COUNTER 持有对 ThreadLocal 实例的引用。故 WebAppClassLoader 持有对 ThreadLocal 实例的可达引用。在这种引用关系（如图 10-5 所示）下，当 MemoryLeakingServlet 所在的 Web 应用程序停止时，由于 TL_COUNTER 对应的 ThreadLocal 实例除了 Tomcat 服务器的工作者线程对其持有可达引用外，还有 WebAppClassLoader 对其有可达引用，而工作者线程又持有对该 ThreadLocal 实例对应的 Counter 实例的可达强引用。因此，这种场景就会导致 ThreadLocal 实例、Counter 实例以及 WebAppClassLoader 所加载的所有类都无法被垃圾回收，这就产生了内存泄漏。如果 MemoryLeakingServlet 类所在的 Web 应用程序被反复停止和启动，则上述内存泄漏会导致每次该 Web 应用程序启动时所加载的类无法被垃圾回收，最终导致 JVM 内存的 PermGen 区域被耗尽（对应的异常消息为：java.lang.OutOfMemoryError: PermGen space）。

清单 10-6. ThreadLocal 内存泄漏示例代码

```
public class MemoryLeakingServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private final static ThreadLocal<Counter> TL_COUNTER = new ThreadLocal<Counter>()
{
        @Override
        protected Counter initialValue() {
            return new Counter();
        }

    };
```

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

         PrintWriter pwr = resp.getWriter();
         pwr.write(String.valueOf(TL_COUNTER.get().getAndIncrement()));
         pwr.close();

    }

}

class Counter {
    private int i=0;

    public int getAndIncrement(){
         return (i++);
    }
}
```

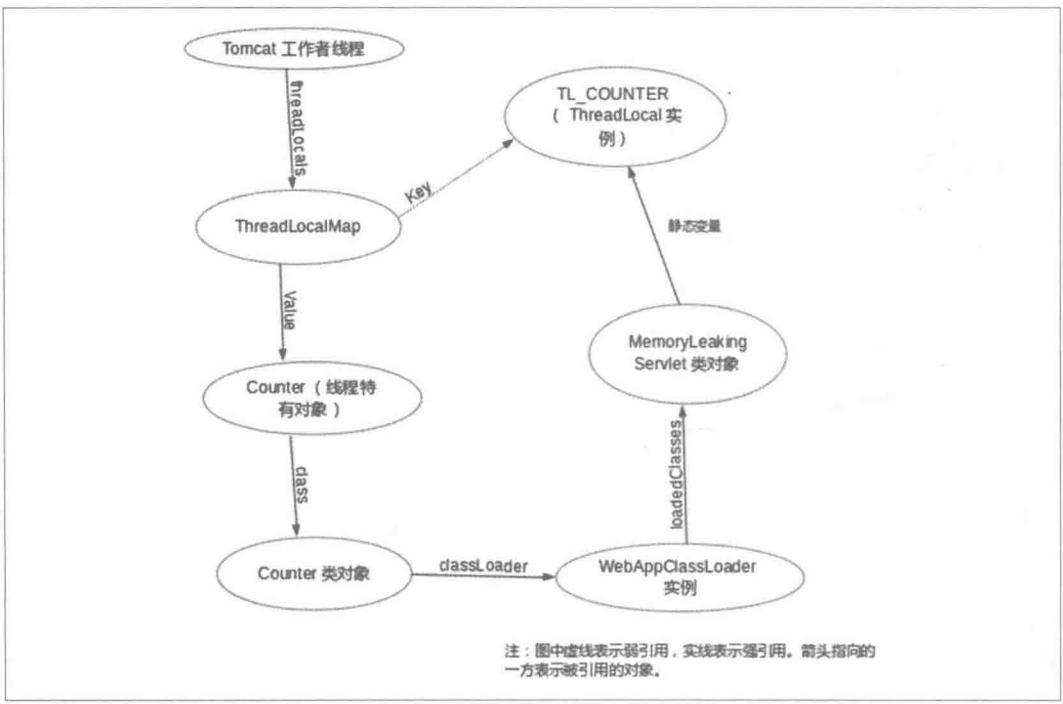


图 10-5. 清单 10-6 代码涉及的对象引用关系

如果把清单 10-6 中的 ThreadLocal 实例对应的线程特有对象改为某个 JDK 标准库类的实例（参见清单 10-7），则会形成伪内存泄漏。这是因为，在 Tomcat 环境中 JDK 标准库类是由 Tomcat 的 StandardClassLoader 负责加载的。当 MemoryPseduoLeakingServlet 所在的 Web 应用程序被停止的时候，StandardClassLoader 并没有持有对 MemoryPseduoLeakingServlet 类的引用。因

此，MemoryPseduoLeakingServlet 类并未持有对 ThreadLocal 实例 TL_COUNTER 的可达引用。即，此时 ThreadLocal 实例除了线程池中的工作线程对其持有可达的弱引用外，系统中没有其他可达的引用。这就形成了上文提到的伪泄漏的条件。

清单 10-7. ThreadLocal 伪内存泄漏示例代码

```
public class MemoryPseduoLeakingServlet extends HttpServlet {
     private static final long serialVersionUID = 1L;

         private final static ThreadLocal<AtomicInteger> TL_COUNTER=new
             ThreadLocal<AtomicInteger>(){
              @Override
        protected AtomicInteger initialValue() {
             return new AtomicInteger();
        }

         };

         @Override
         protected void doGet(HttpServletRequest req, HttpServletResponse resp)
             throws ServletException, IOException {


              PrintWriter pwr = resp.getWriter();
              pwr.write(String.valueOf(TL_COUNTER.get().getAndIncrement()));
              pwr.close();

         }

}
```

综上所述，在服务器环境下使用 ThreadLocal 导致内存泄漏的一个关键条件是 Web 应用程序的类加载器持有对 ThreadLocal 实例的可达引用（参见图 10-5）。在此基础上，我们就可以找到相应的规避方法，相关代码参见清单 10-8。

清单 10-8. 可防止 ThreadLocal 内存泄漏的工具类

```
/**
 * 支持统一清理不再被使用的 ThreadLocal 变量的 ThreadLocal 子类。
 *
 * @author Viscent Huang
 *
 * @param <T> 相应的线程特有对象类型
 */
public class ManagedThreadLocal<T> extends ThreadLocal<T> {

    /*
     * 使用弱引用，防止内存泄漏。
     * 使用 volatile 修饰保证内存可见性。
     */
    private static volatile Queue<WeakReference<ManagedThreadLocal<?>>> instances =
        new ConcurrentLinkedQueue<WeakReference<ManagedThreadLocal<?>>>();

    private volatile ThreadLocal<T> threadLocal;
```

```
private ManagedThreadLocal(final InitialValueProvider<T> ivp) {

    this.threadLocal = new ThreadLocal<T>() {

        @Override
        protected T initialValue() {
            return ivp.initialValue();
        }

    };
}

public static <T> ManagedThreadLocal<T> newInstance(
   final InitialValueProvider<T> ivp) {
    ManagedThreadLocal<T> mtl = new ManagedThreadLocal<T>(ivp);

    // 使用弱引用来引用 ThreadLocalProxy 实例，防止内存泄漏。
    instances.add(new WeakReference<ManagedThreadLocal<?>>(mtl));
    return mtl;
}

public static <T> ManagedThreadLocal<T> newInstance() {
    return newInstance(new ManagedThreadLocal.InitialValueProvider<T>());
}

public T get() {
    return threadLocal.get();
}

public void set(T value) {
    threadLocal.set(value);
}

public void remove() {
    if (null != threadLocal) {
        threadLocal.remove();
        threadLocal = null;
    }
}

/**
 * 清理该类所管理的所有 ThreadLocal 实例。
 */
public static void removeAll() {
    WeakReference<ManagedThreadLocal<?>> wrMtl;
    ManagedThreadLocal<?> mtl;
    while (null != (wrMtl = instances.poll())) {
        mtl = wrMtl.get();
        if (null != mtl) {
            mtl.remove();
        }
    }
}

public static class InitialValueProvider<T> {
    protected T initialValue(){

        //默认值为 null
```

```
            return null;
        }
    }

}
```

如清单 10-8 所示的 ManagedThreadLocal 是 ThreadLocal 的子类，故其使用方法与 ThreadLocal 基本相同。ManagedThreadLocal 的静态方法 removeAll 用于打破该类的所有实例对其实例变量 threadLocal 的引用。这就消除了产生 ThreadLocal 内存泄漏的关键条件。通常，我们可以在 ServletContextListener 实例的 contextDestroyed 方法中调用 ManagedThreadLoca.removeAll。这样，当某个 Web 应用程序停止的时候，该应用所使用的所有 ThreadLocal 实例都会被清理掉（假设该应用程序都通过 ManagedThreadLocal 来访问线程特有对象）。清单 10-9 展示了一个使用 ManagedThreadLocal 来防止 ThreadLocal 内存泄漏的例子。

清单 10-9. 使用 ManagedThreadLocal 类防止 ThreadLocal 内存泄漏

```
public class MemoryLeakPreventingServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private final static ManagedThreadLocal<Counter> TL_COUNTER = ManagedThreadLocal
        .newInstance(new ManagedThreadLocal.InitialValueProvider<Counter>() {
            @Override
            protected Counter initialValue() {
                return new Counter();
            }
        });

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        PrintWriter pwr = resp.getWriter();
        pwr.write(String.valueOf(TL_COUNTER.get().getAndIncrement()));
        pwr.close();

    }
}
```

10.5 Thread Specific Storage 模式的可复用实现代码

JDK 1.2 引入的标准库类 java.lang.ThreadLocal<T>可以看成是 Thread Specific Storage 模式的可复用实现。ThreadLocal 类相当于 Thread Specific Storage 模式的 TSObjectProxy 参与者。其类型参数 T 相当于 TSObject 参与者。利用 ThreadLocal 实现 Thread Specific Storage 模式，应用代码只需要完成以下几件事情。

1.【必需】创建 ThreadLocal 的子类（或者匿名子类）。

2.【可选，但通常是需要的】在 ThreadLocal 的子类中覆盖其父类的 initialValue 方法，用于定义初始的线程特有对象实例。

需要注意的是，类型为 ThreadLocal 的变量，其声明通常采用 static final 修饰。不同的线程采用同一个 ThreadLocal 实例即可获取所需的线程特有对象实例，因此类型为 ThreadLocal 的变量定义为类变量（用 static 修饰）即可，而无须定义为实例变量（不使用 static 修饰）。

10.6 Java 标准库实例

JDK 1.7 中引入的标准库类 java.util.concurrent.ThreadLocalRandom 就使用了 Thread Specific Storage 模式。ThreadLocalRandom 是 ThreadLocal 的一个子类，其 current 方法返回一个属于当前线程的 ThreadLocalRandom 实例。这里，current 方法的返回值就是一个线程特有对象实例。因此，不同线程调用 ThreadLocalRandom 实例的相关方法获取下一个随机数的时候，相互之间不影响。这比多个线程共享一个 java.math.Random 实例来获取随机数效率要高，尽管 Random 类也是线程安全的。

10.7 相关模式

10.7.1 Immutable Object 模式（第 3 章）

Thread Specific Storage 模式通过不共享变量实现线程安全。Immutable Object 模式则是使被共享的变量所引用的对象的状态不可变来实现线程安全。二者都是在无须使用显式锁的情况下保证线程安全。

10.7.2 Proxy（代理）模式[3]

Thread Specific Storage 模式的 TSObjectProxy 参与者相当于 Proxy 模式的 Proxy 参与者。

10.7.3 Singleton（单例）模式[4]

Thread Specific Storage 模式可以用来实现特定于线程的单例模式，即实现每个线程只生成某个类的一个且仅一个实例。

---

3 GOF 设计模式，不在本书讨论范围之内。详情可参考本章“参考资源”。

4 GOF 设计模式，不在本书讨论范围之内。详情可参考本章“参考资源”。

10.8 参考资源

1. Schmidt, Douglas et al. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

2. ThreadLocal 类源码. http://grepcode.com/file/repository.grepcode.com/ java/root/jdk/openjdk/6-b27/java/lang/ThreadLocal.java#ThreadLocal.

3. ThreadLocalRandom 类源码. http://grepcode.com/file/repository.grepcode.com/java/root/ jdk/openjdk/7u40-b43/java/util/concurrent/ThreadLocalRandom.java#ThreadLocalRandom.

4. Erich Gamma 等. 设计模式：可复用面向对象软件的基础（英文版）. 机械工业出版社，2002.

第 11 章

Serial Thread Confinement（串行线程封闭）模式

11.1 Serial Thread Confinement 模式简介

如果并发任务的执行涉及某个非线程安全对象，而我们又希望因此而引入锁，那么我们可以考虑使用 Serial Thread Confinement 模式。

Serial Thread Confinement 模式的核心思想是通过将多个并发的任务存入队列实现任务的串行化，并为这些串行化的任务创建唯一的一个工作者线程进行处理。因此，这个唯一的工作者线程所访问的非线程安全对象由于只有一个线程访问它，对其的访问自然无须加锁，从而避免了锁的开销及由锁可能引发的问题。

当然，如果我们对并发任务访问的非线程安全对象进行加锁，也能实现任务的串行化从而实现线程安全，另外 Serial Thread Confinement 模式串行化并发任务所使用的队列本身也会涉及锁。因此，Serial Thread Confinement 模式的本质是使用一个开销更小的锁（串行化并发任务时所用队列涉及的锁）去替代另一个可能的开销更大的锁（为保障并发任务所访问的非线程安全对象可能引入的锁）。

11.2 Serial Thread Confinement 模式的架构

Serial Thread Confinement 模式借助队列将并发任务串行化，并创建一个且仅一个工作者线程对这些串行化的任务进行处理。这使得我们可以不必对这些任务执行时所访问的非线程安全对象进行加锁。

Serial Thread Confinement 模式的主要参与者有以下几种。其类图如图 11-1 所示。

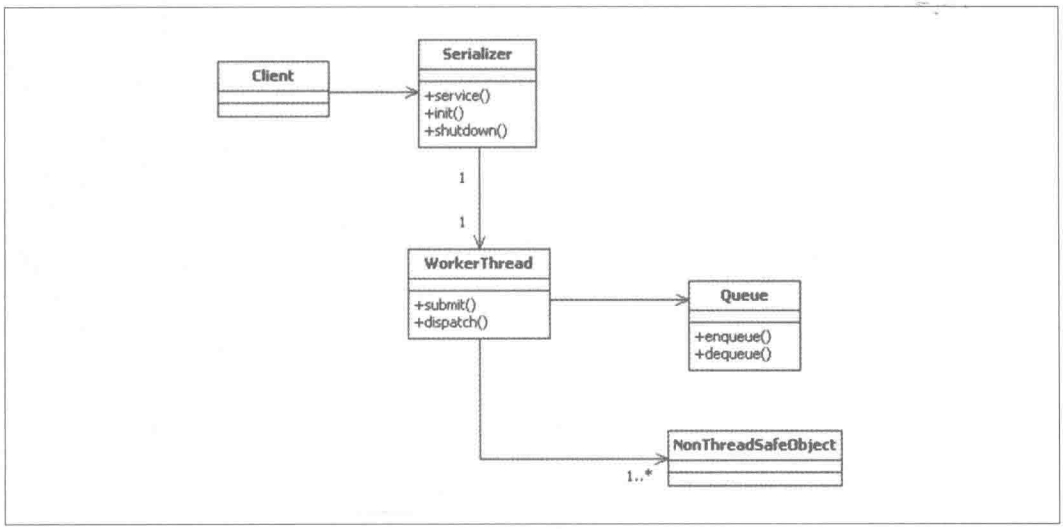


图 11-1. Serial Thread Confinement 模式的类图

- **Serializer**：负责对外暴露服务接口以及工作者线程的生命周期管理，并将客户端对其的并发服务调用转换为相应的任务以实现服务串行化。其主要方法及职责如下：
  - **service**：表示 Serial Thread Confinement 模式对外暴露的服务方法，该方法将客户端对其的并发调用串行化。
  - **init**：初始化 Serial Thread Confinement 模式对外暴露的服务。该方法可以启动工作者线程。
  - **shutdown**：停止 Serial Thread Confinement 模式对外暴露的服务。该方法可以停止工作者线程。
- **WokerThread**：工作者线程，负责接收 Serializer 提交的并发任务以及这些任务的执行。其主要方法及职责如下。
  - **submit**：用于 Serializer 提交并发任务，并将这些任务串行化。
  - **dispatch**：用于执行串行化的任务。
- **Queue**：Serializer 和 WorkerThread 间的缓冲区，用于实现并发任务的串行化。其主要方法及职责如下。

- **enqueue**：用于并发任务入队列。
- **dequeue**：用于任务出队列。

- **NonThreadSafeObject**：工作者线程执行任务时所需访问的非线程安全对象。

Serial Thread Confinement 模式的序列图如图 11-2 所示。

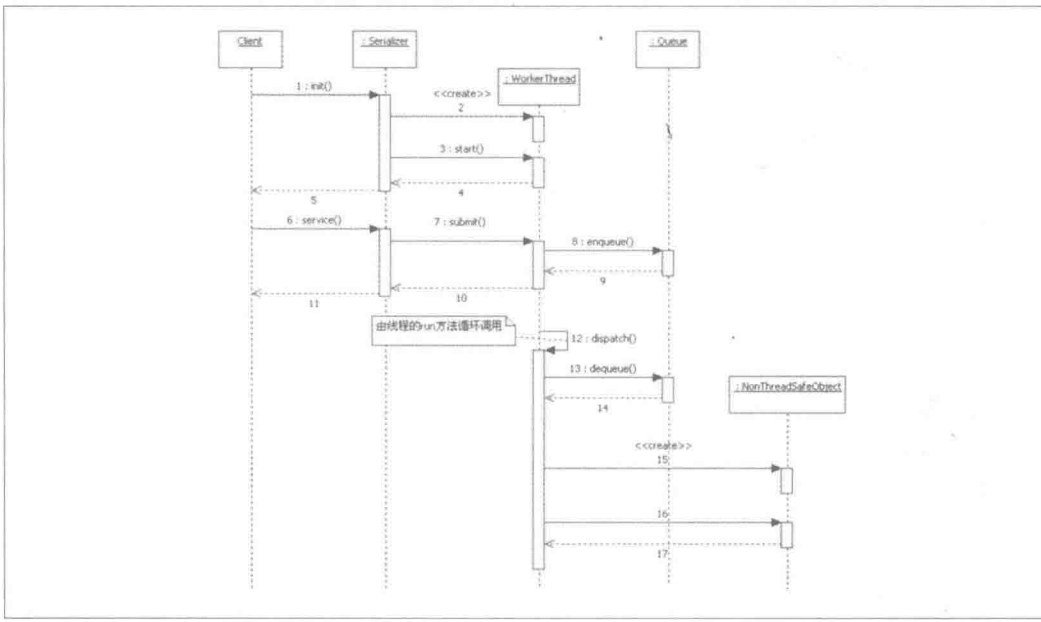


图 11-2. Serial Thread Confinement 模式的序列图

第 1 步：客户端代码调用 Serializer 的 init 方法初始化 Serializer 提供的服务。

第 2~4 步：init 方法创建并启动其唯一的工作者线程实例。

第 5 步：init 方法返回。

第 6 步：客户端代码调用 Serializer 实例的服务方法 service。

第 7 步：service 方法调用其工作者线程实例的 submit 方法提交一个任务。

第 8、9 步：submit 方法将其接收到的任务存入队列，实现并发任务的串行化。

第 10 步：submit 方法返回。

第 11 步：service 方法返回。

第 12 步：工作者线程运行。

第 13、14 步：工作者线程取队列中的任务。

第 15~17 步：工作者线程在执行任务过程创建并访问非线程安全对象实例。

11.3 Serial Thread Confinement 模式实战案例解析

某系统需要支持从内网的某台 FTP 服务器上下载一批文件的功能。该功能的实现会用到一款开源 FTP 客户端组件。该 FTP 客户端组件并非线程安全，因此如果多个线程共享其实例可能引起数据错乱。另外，系统中会有多个线程需要从服务器上下载文件，并且我们不希望这些需要下载文件的线程等待要下载的文件下载完毕后才能进行其他处理。因此，这里我们需要使用异步编程。如果我们采用多个线程去并发下载一批文件，并使其中的每个线程都持有一个特有的 FTP 客户端实例（即使用 Thread Specific Storage 模式，参见第 10 章）。那么，线程安全是得到保障了。但这样意味着某一时刻该系统与对端 FTP 服务器建立了多个网络连接。而这种情况除非必要的时候，否则我们不希望其出现。当然，使用锁也能够实现 FTP 客户端的线程安全，但是，这样一来从服务器上下载文件其实是串行的（逐一下载文件）。并且，文件下载过程中涉及的网络 I/O、文件 I/O 都会引起上下文切换，为保障 FTP 客户端线程安全而引入的锁[1]增加了额外的上下文切换，从而增加系统的负担。

因此，这里我们可以使用 Serial Thread Confinement 模式。它可以帮助我们达成以下几个目标。

一、异步编程。这使得客户端线程不必等待要下载的文件下载完毕便可以继续其他处理。

二、不借助锁而实现线程安全。这使得负责文件下载的工作者线程可以安全地使用上述 FTP 客户端实例，而又不会增加不必要的上下文切换。

使用 Serial Thread Confinement 模式实现文件下载，本质也是串行下载一批文件。表面上看，这点与使用锁实现 FTP 客户端线程安全所导致的结果无异。但实际上，我们是采用一个开销更小的锁（队列涉及的锁）去替代一个开销更大的锁（为保障非线程安全对象的线程安全而引入的锁）。当然，这里我们主要考虑的是整个系统的吞吐率，而不仅仅是文件下载服务本身的计算效率。

本案例实现文件下载服务的类如清单 11-1 所示。

1 确切地说，是被征用的锁（Contented Lock）会引起上下文切换。没有被征用的锁不会引起上下文切换。

清单 11-1. 文件下载服务类源码

```
/*
 * 实现 FTP 文件下载。
 * 模式角色：SerialThreadConfiment.Serializer
 */
public class MessageFileDownloader {

    // 模式角色：SerialThreadConfiment.WorkerThread
    private final WorkerThread workerThread;

    public MessageFileDownloader(String outputDir, final String ftpServer,
        final String userName, final String password) {

        workerThread = new WorkerThread(outputDir, ftpServer, userName, password);

    }

    public void init() {
        workerThread.start();
    }

    public void shutdown() {
        workerThread.terminate();
    }

    public void downloadFile(String file) {
        workerThread.download(file);
    }

    // 模式角色：SerialThreadConfiment.WorkerThread
    private static class WorkerThread extends AbstractTerminatableThread {

        // 模式角色：SerialThreadConfiment.Queue
        private final BlockingQueue<String> workQueue;
        private final Future<FTPClient> ftpClientPromise;
        private final String outputDir;

        public WorkerThread(String outputDir, final String ftpServer,
            final String userName, final String password) {
            this.workQueue = new ArrayBlockingQueue<String>(100);
            this.outputDir = outputDir + '/';

            this.ftpClientPromise = new FutureTask<FTPClient>(
                new Callable<FTPClient>() {

                    @Override
                    public FTPClient call() throws Exception {
                        FTPClient ftpClient = initFTPClient(ftpServer, userName,
                            password);
                        return ftpClient;
                    }

                });

            new Thread((FutureTask<FTPClient>) ftpClientPromise).start();
        }
```

```
public void download(String file) {
    try {
        workQueue.put(file);
        terminationToken.reservations.incrementAndGet();
    } catch (InterruptedException e) {
        ;
    }
}

private FTPClient initFTPClient(String ftpServer, String userName,
   String password) throws Exception {
    FTPClient ftpClient = new FTPClient();

    FTPClientConfig config = new FTPClientConfig();
    ftpClient.configure(config);

    int reply;
    ftpClient.connect(ftpServer);

    System.out.print(ftpClient.getReplyString());

    reply = ftpClient.getReplyCode();

    if (!FTPReply.isPositiveCompletion(reply)) {
        ftpClient.disconnect();
        throw new RuntimeException("FTP server refused connection.");
    }

    boolean isOK = ftpClient.login(userName, password);
    if (isOK) {
        System.out.println(ftpClient.getReplyString());

    } else {
        throw new RuntimeException("Failed to login."
           + ftpClient.getReplyString());
    }

    reply = ftpClient.cwd("~/messages");
    if (!FTPReply.isPositiveCompletion(reply)) {
        ftpClient.disconnect();
        throw new RuntimeException("Failed to change working
            directory.reply:" + reply);
    } else {

        System.out.println(ftpClient.getReplyString());
    }

    ftpClient.setFileType(FTP.ASCII_FILE_TYPE);
    return ftpClient;

}

@Override
protected void doRun() throws Exception {
    String file = workQueue.take();

    OutputStream os = null;
    try {
        os = new BufferedOutputStream(new FileOutputStream(outputDir +
```

```
                    file));
                ftpClientPromise.get().retrieveFile(file, os);
            } finally {
                if (null != os) {
                    try {
                        os.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                terminationToken.reservations.decrementAndGet();
            }


        @Override
        protected void doCleanup(Exception cause) { \
            try {
                ftpClientPromise.get().disconnect();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }

    }
}
```

MessageFileDownloader 类相当于 Serial Thread Confinement 模式的 Serializer 参与者。其静态内部类 WorkerThread 相当于 WorkerThread 参与者。静态类 WorkerThread 的实例变量 workQueue 相当于 Queue 参与者实例。清单 11-2 展示了 MessageFileDownloader 类的一个示例客户端源码。

清单 11-2. MessageFileDownloader 类的示例客户端源码

```
public class SampleClient {
    private static final MessageFileDownloader DOWNLOADER;

    static {
        DOWNLOADER = new MessageFileDownloader("/home/viscent/tmp/incoming",
            "192.168.1.105", "datacenter", "abc123");
        DOWNLOADER.init();
    }

    public static void main(String[] args) {
        DOWNLOADER.downloadFile("abc.xml");
        // 执行其他操作
    }

}
```

11.4 Serial Thread Confinement 模式的评价与实现考量

Serial Thread Confinement 模式可以帮助我们在不使用锁的情况下实现线程安全。但是，在实际使用时需要注意 Serial Thread Confinement 模式自身的开销：将任务串行化所涉及的进出队列以及 Serializer 创建向 WorkerThread 提交的任务对象这些动作都有时间和空间的开销。

Serial Thread Confinement 模式的本质是通过使用一个开销更小的锁来替代另一个开销更大的锁以实现线程安全。因此，实际应用时我们需要注意比较锁的开销：如果采用锁去保障对某个非线程安全对象的访问的线程安全，那么这个锁的开销比起 Serial Thread Confinement 模式中使用的队列涉及的锁哪个开销更大些？

Serial Thread Confinement 模式的典型应用场景包括以下两个。

- **需要使用非线程安全对象，但又不希望引入锁**：任务的执行涉及非线程安全对象，如果采用锁去保证对这些对象访问的线程安全，这些锁的开销比起将任务通过队列中转涉及锁的开销更大的话，那么我们可以使用 Serial Thread Confinement 模式。本章案例就属于这种场景。
- **任务的执行涉及 I/O 操作，但我们不希望过多的 I/O 线程增加上下文切换**：I/O 操作，如文件 I/O、网络 I/O 会增加系统的上下文切换。因此，如果涉及 I/O 的线程越多，那么系统的处理效率可能反而越低。这是由于过多的上下文切换消耗了过多的系统资源。本章案例之所以采用单线程去处理文件下载，也是出于减少上下文切换的考虑。

任务的处理结果

如果客户端代码关心任务的处理结果，那么我们可以借用 Promise 模式（参见第 6 章）。此时，我们可以让 Serializer 的 service 方法返回一个 Promise（java.util.concurrent.Future）实例，客户端代码通过该实例可以获取相应任务的处理结果。但是，这样做需要注意一点：多个客户端线程共享同一个 Serializer 实例意味着多个线程会等待同一个线程（Serializer 实例所管理的工作者线程）的处理结果。如果 WorkerThread 处理过慢，或者客户端代码在 Serializer 的 service 方法调用与获取任务的处理结果之间的时间间隔太短，使得 WorkerThread 没有足够的时间执行相应任务，那么这可能导致客户端线程等待时间过长。当然，我们也可以考虑生成多个 Serializer 实例。这样，每个 Serializer 实例的客户端线程在获取任务处理结果时若需要等待也只是等待其调用 Serializer 实例对应的工作者线程。不过这样一来，我们又需要考虑工作者线

程的数量是否会太多。

11.5 Serial Thread Confinement 模式的可复用实现代码

下面的代码展示了 Serial Thread Confinement 模式的一个可复用实现。如清单 11-3、清单 11-4 和清单 11-5 所示。

清单 11-3. Serializer 参与者的可复用实现代码

```
/**
 * Serial Thread Confinement 模式 Serializer 参与者可复用实现。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *         Serializer 向 WorkerThread 所提交的任务对应的类型
 * @param <V>
 *         service 方法的返回值类型
 */
public abstract class AbstractSerializer<T, V> {
    private final TerminatableWorkerThread<T, V> workerThread;

    public AbstractSerializer(BlockingQueue<Runnable> workQueue,
        TaskProcessor<T, V> taskProcessor) {
        workerThread = new TerminatableWorkerThread<T, V>(workQueue, taskProcessor);
    }

    /**
     * 留给子类实现。用于根据指定参数生成相应的任务实例。
     *
     * @param params
     *         参数列表
     * @return 任务实例。用于提交给 WorkerThread。
     */
    protected abstract T makeTask(Object... params);

    /**
     * 该类对外暴露的服务方法。该类的子类需要定义一个命名含义比该方法更为具体的方法（如
     * downloadFile）。
     * 含义具体的服务方法（如 downloadFile）可直接调用该方法。
     *
     * @param params
     *         客户端代码调用该方法时所传递的参数列表
     * @return 可借以获取任务处理结果的 Promise（参见第 6 章，Promise 模式）实例。
     * @throws InterruptedException
     */
    protected Future<V> service(Object... params) throws InterruptedException {
        T task = makeTask(params);
        Future<V> resultPromise = workerThread.submit(task);
        return resultPromise;
```

```
    }

    /**
     * 初始化该类对外暴露的服务。
     */
    public void init() {
        workerThread.start();
    }

    /**
     * 停止该类对外暴露的服务。
     */
    public void shutdown() {
        workerThread.terminate();
    }

}
```

清单 11-4. WorkerThread 参与者的可复用实现代码

```
/**
 * Serial Thread Confinement 模式 WorkerThread 参与者可复用实现。
 * 该类使用了 Two-phase Termination 模式（参见第 5 章）。
 * @author Viscent Huang
 *
 * @param <T>
 *         Serializer 向 WorkerThread 所提交的任务对应的类型
 * @param <V>
 *         表示任务执行结果的类型
 */
public class TerminatableWorkerThread<T, V> extends AbstractTerminatableThread {
    private final BlockingQueue<Runnable> workQueue;

    //负责真正执行任务的对象
    private final TaskProcessor<T, V> taskProcessor;

    public TerminatableWorkerThread(BlockingQueue<Runnable> workQueue,
        TaskProcessor<T, V> taskProcessor) {
        this.workQueue = workQueue;
        this.taskProcessor = taskProcessor;
    }

    /**
     * 接收并行任务，并将其串行化。
     *
     * @param task
     *         任务
     * @return 可借以获取任务处理结果的 Promise（参见第 6 章，Promise 模式）实例。
     * @throws InterruptedException
     */
    public Future<V> submit(final T task) throws InterruptedException {
        Callable<V> callable = new Callable<V>() {

            @Override
            public V call() throws Exception {
                return taskProcessor.doProcess(task);
            }
```

```
        };

        FutureTask<V> ft = new FutureTask<V>(callable);
        workQueue.put(ft);

        terminationToken.reservations.incrementAndGet();
        return ft;
    }

    /**
     * 执行任务的处理逻辑
     */
    @Override
    protected void doRun() throws Exception {
        Runnable ft = workQueue.take();
        try {
            ft.run();
        } finally {
            terminationToken.reservations.decrementAndGet();
        }

    }

}
```

清单 11-5. TaskProcessor 接口源码

```
/**
 * 对任务处理的抽象。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *          表示任务的类型
 * @param <V>
 *          表示任务处理结果的类型
 */
public interface TaskProcessor<T, V> {
    /**
     * 对指定任务进行处理。
     *
     * @param task
     *          任务
     * @return 任务处理结果
     * @throws Exception
     */
    V doProcess(T task) throws Exception;
}
```

利用本章的可复用代码实现 Serial Thread Confinement 模式，应用程序只需要完成以下几件事情。

1.【必需】定义 Serializer 提交给 WorkerThread 的任务对应的类型。

2.【必需】定义 AbstractSerializer 的子类，并实现其父类定义的 makeTask 抽象方法。另外该子类需要定义一个名字含义比 service 更为具体的服务方法。该方法可直接调用其父类的 service 方法。

3.【必需】定义 TaskProcessor 接口的实现类。

清单 11-6 展示了一个使用代码实现 Serial Thread Confinement 模式的例子。

清单 11-6. 使用可复用代码实现 Serial Thread Confinement 模式示例代码

```
public class ReusableCodeExample {

    public static void main(String[] args) throws InterruptedException,
       ExecutionException {
        SomeService ss = new SomeService();
        ss.init();
        Future<String> result = ss.doSomething("Serial Thread Confinement", 1);

        Thread.sleep(50);

        System.out.println(result.get());

        ss.shutdown();

    }

    private static class Task {
        public final String message;
        public final int id;

        public Task(String message, int id) {
            this.message = message;
            this.id = id;
        }
    }

    private static class SomeService extends AbstractSerializer<Task, String> {

        public SomeService() {
            super(new ArrayBlockingQueue<Runnable>(100),
                new TaskProcessor<Task, String>() {

                    @Override
                    public String doProcess(Task task) throws Exception {
                        System.out.println("[" + task.id + "]:" + task.message);
                        return task.message + " accepted.";
                    }

                });
        }

        @Override
        protected Task makeTask(Object... params) {
            String message = (String) params[0];
            int id = (Integer) params[1];
```

```
            return new Task(message, id);
        }
        public Future<String> doSomething(String message, int id)
            throws InterruptedException {
            Future<String> result = null;

            result = service(message, id);

            return result;
        }

    }

}
```

11.6 Java 标准库实例

Swing 中的实用工具类 SwingUtilities 就使用了 Serial Thread Confinement 模式。该类的 invokeLater 方法使得多个应用线程能够并发提交与 Swing GUI 有关的任务。而这些任务仅由唯一的一个线程（即 Swing 的 Event Loop 线程）去负责执行。从而保障了 Swing GUI 层的线程安全。这里，SwingUtilities 类相当于 Serializer 参与者，其 invokeLater 方法相当于 Serializer 参与者的 service 方法。而 Swing Event Loop 线程则相当于 WorkerThread 参与者。

11.7 相关模式

11.7.1 Immutable Object 模式（第 3 章）

Immutable Object 模式也可以在不使用锁的情况下实现线程安全。其核心是使被各个线程所共享对象的状态不可变。

11.7.2 Promise 模式（第 6 章）

如果客户端代码关心任务的处理结果，那么我们可以借用 Promise 模式（参见第 6 章）。此时，可以让 Serializer 的 service 方法返回一个 Promise（java.util.concurrent.Future）实例，客户端通过该实例可以获取相应任务的处理结果。

11.7.3 Producer-Consumer 模式（第 7 章）

Serial Thread Confinement 模式可以看成 Producer-Consumer 模式的一个实例。Serial Thread Confinement 模式的 Serializer 参与者、Queue 参与者和 WorkerThread 参与者分别相当于

Producer-Consumer 模式的 Producer 参与者、Channel 参与者和 Consumer 参与者。

11.7.4 Thread Specific Storage（线程特有存储）模式（第 10 章）

Thread Specific Storage 模式也可以在不使用锁的情况下实现线程安全。其核心是使每个线程都生成一个且仅一个某个非线程安全对象的实例，各个线程之间不共享该非线程安全对象的实例。

11.8 参考资源

1．Brian Göetz et al. Java Concurrency In Practice. Addison Wesley, 2006.

2．Doug Lea. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition. Addison Wesley, 1999.

3．SwingUtilities 源码. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/ openjdk/6-b27/javax/swing/SwingUtilities.java#SwingUtilities.

第 12 章

Master-Slave（主仆）模式

12.1 Master-Slave 模式简介

Master-Slave 模式是一个基于分而治之（Divide and conquer）思想的设计模式。其核心思想是将一个任务（原始任务）分解为若干个语义等同（Semantically-identical）的子任务，并由专门的工作者线程来并行执行这些子任务。原始任务的处理结果是通过整合各个子任务的处理结果而形成的。而这些与分而治之相关的处理细节对于原始任务的提交方来说又是不可见的。因此，Master-Slave 模式既提高计算效率，又实现了信息隐藏。

12.2 Master-Slave 模式的架构

为了提高计算效率，我们往往将一个规模比较大的任务分解为若干个子任务，并使用专门的工作者线程来执行这些子任务。然后将各个子任务的处理结果整合为原始任务的处理结果。但是，对于服务的客户端代码而言，有关任务的分解、子任务处理结果的合并以及工作者线程的管理等细节都应该是不可见的，以便于提高服务代码的可维护性和简化客户端代码。

Master-Slave 模式通过在子任务和服务的客户端代码之间引入一个协调性对象（Master）解决了上述问题。Master-Slave 模式的主要参与者有以下几种。其类图如图 12-1 所示。

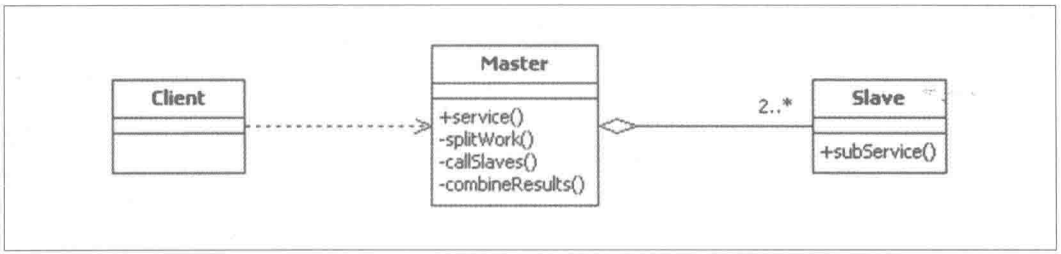


图 12-1. Master-Slave 模式的类图

- **Master**：负责原始任务的分解、子任务的派发和子任务处理结果的合并。其主要方法及职责如下。
  - **service**：这是 Master 参与者对外暴露的接口，用于接收原始任务，并返回其处理结果。
  - **splitWork**：将原始任务分解成若干个语义等同的子任务。
  - **callSlaves**：将各个子任务派发给各个 Slave 实例进行处理。
  - **combineResults**：将各个子任务的处理结果进行整合，形成原始任务的处理结果。
- **Slave**：负责子任务的处理。其主要方法及职责如下。
  - **subService**：异步方法，负责执行子任务的处理逻辑。

Master-Slave 模式的序列图如图 12-2 所示。

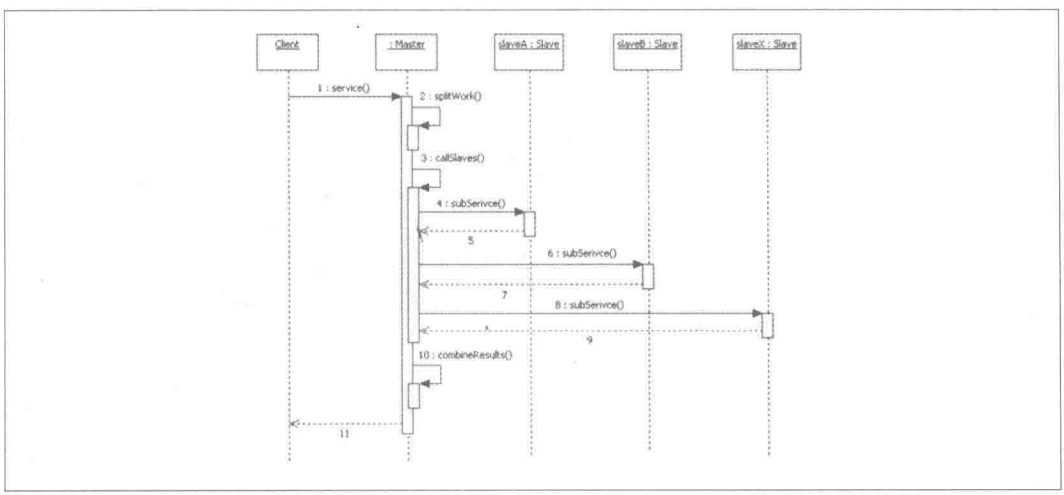


图 12-2. Master-Slave 模式的序列图

第 1 步：客户端代码调用 Master 参与者实例的 service 方法提交一个任务（原始任务）。

第 2 步：service 方法调用 splitWork 方法将原始任务分解为若干个等同语义的子任务。

第 3 步：service 方法调用 callSlaves 方法，进行子任务的派发。

第 4~9 步：callSlaves 方法逐一调用各个 Slave 参与者实例的 subService 方法将子任务派发给相应的 Slave 参与者实例进行处理。

第 10 步：service 方法调用 combineResults 方法，将各个子任务的处理结果整合成原始任务的处理结果。

第 11 步：service 方法返回原始任务的处理结果。

由如图 12-2 所示的序列图可知，Master-Slave 模式的客户端代码仅与 Master 参与者的 service 方法打交道，因此与分而治之相关的细节对于客户端代码是不可见的。

12.3 Master-Slave 模式实战案例解析

某基于 Web Service 的电信系统需要一个系统流量统计工具。该工具的统计依据是该系统运行过程中产生的接口日志文件。接口日志文件记录了该系统接收到的请求、该系统返回给客户端的响应以及该系统调用外部系统时涉及的请求和响应的相关数据。其格式如下：

> 操作时间戳|协议类型（SOAP/REST/HTTP）|记录类型（请求/响应）|接口名称|操作名称|源设备名|目标设备名|消息唯一标识|本机 IP 地址|主叫号码|被叫号码

图 12-3 展示了一个示例接口日志文件。

```
2015-02-01 14:13:45.039|SOAP|request|SMS|sendSms|OSG|ESB|00200033375|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.067|SOAP|response|SMS|sendSmsRsp|ESB|OSG|00200033375|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.100|SOAP|request|SMS|sendSms|ESB|NIG|00210033376|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.396|SOAP|response|SMS|sendSmsRsp|NIG|ESB|00210033379|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.047|REST|request|Location|getLocation|OSG|ESB|00200008326|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.060|REST|request|Location|getLocation|ESB|NIG|00210008327|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.113|REST|response|Location|getLocationRsp|NIG|ESB|00210008330|192.168.1.102|13612345678|136712345670
2015-02-01 14:13:45.234|REST|response|Location|getLocationRsp|ESB|OSG|00200008328|192.168.1.102|13612345678|136712345670
```

图 12-3. 示例接口日志文件

另外，流量统计工具要支持统计指定时间段内的系统流量。因此，我们把该工具分为两个模块：Linux Shell 脚本模块和 Java 模块。Linux Shell 脚本模块根据指定的时间段查找相应的接口日志文件，并将所有符合要求的接口日志文件的文件名通过标准输出传递给 Java 模块。Java

模块根据其标准输入（System.in）中指定的接口日志文件名读取相应的接口日志文件进行流量统计。

实现上述统计工具需要考虑以下几个问题。

首先，该系统的接口日志文件被统一存储在日志文件服务器上。有时我们可能需要在日志服务器上直接统计系统流量。因此，我们希望统计工具占用的系统资源（CPU 时间和内存）尽可能地低，以免其运行干扰了日志服务器。

其次，统计工具的统计依据——接口日志文件可能包含大量的记录。以该系统单节点的流量为 100TPS（Transaction per Second，即 1s 内系统能够处理的请求数量），平均某个请求的处理会产生 4 条接口日志记录为例，那么 1h 会产生上千万条接口日志记录（100×3600×4=1440000）。因此，统计工具的统计速率要尽可能快，以免在急需统计结果的时候等待过久。

当然，上述两个需求是矛盾的。我们只能在统计速率和资源消耗之间寻求一个平衡点。这里，我们可以使用 Master-Slave 模式：某个时间段内涉及的接口日志记录总数可能达上千万，因此原始任务的规模比较大。考虑到每个接口日志文件包含的记录个数最多为 N 条（如 N 为 10000），我们可以以文件为单位进行任务分解，将若干个日志文件合为一个子任务，并利用专门的工作者线程去执行这些子任务。再汇总各个子任务的统计结果并可得到我们所需要的最终结果。

统计工具的 Java 模块的源码如清单 12-1 所示。

清单 12-1. 流量统计工具的 Java 模块源码

```
public class TPSStat {

    public static void main(String[] args) throws Exception {
        // 接口日志文件所在目录
        String logBaseDir = args[0];

        // 忽略的操作名列表
        String excludedOperationNames = "";

        // 指定要统计在内的操作名列表
        String includedOperationNames = "*";

        // 指定要统计在内的目标设备名
        String destinationSysName = "*";

        int argc = args.length;

        if (argc > 2) {
            excludedOperationNames = args[1];
        }
```

```
        if (argc > 3) {
            excludedOperationNames = args[2];
        }

        if (argc > 4) {
            destinationSysName = args[3];
        }

        Master processor = new Master(logBaseDir, excludedOperationNames,
            includedOperationNames, destinationSysName);

        BufferedReader fileNamesReader = new BufferedReader(new InputStreamReader(
            System.in));

        ConcurrentMap<String, AtomicInteger> result = processor
            .calculate(fileNamesReader);

        for (String timeRange : result.keySet()) {
            System.out.println(timeRange + "," + result.get(timeRange));
        }

    }

    // 模式角色：Master-Slave.Master
    private static class Master {
        private final String logFileBaseDir;
        private final String excludedOperationNames;
        private final String includedOperationNames;
        private final String destinationSysName;

        // 每次派发给某个 Slave 线程的文件个数
        private static final int NUMBER_OF_FILES_FOR_EACH_DISPATCH = 5;
        private static final int WORKER_COUNT = Runtime.getRuntime()
            .availableProcessors();

        public Master(String logFileBaseDir, String excludedOperationNames,
            String includedOperationNames, String destinationSysName) {
            this.logFileBaseDir = logFileBaseDir;
            this.excludedOperationNames = excludedOperationNames;
            this.includedOperationNames = includedOperationNames;
            this.destinationSysName = destinationSysName;
        }

        public ConcurrentMap<String, AtomicInteger> calculate(
            BufferedReader fileNamesReader) throws IOException {
            ConcurrentMap<String, AtomicInteger> repository = new
                ConcurrentSkipListMap<String, AtomicInteger>();

            // 创建工作者线程
            Worker[] workers = createAndStartWorkers(repository);

            // 指派任务给工作者线程
            dispatchTask(fileNamesReader, workers);

            // 等待工作者线程处理结束
            for (int i = 0; i < WORKER_COUNT; i++) {
                workers[i].terminate(true);
            }
```

```
        // 返回处理结果
        return repository;
    }

    private Worker[] createAndStartWorkers(
        ConcurrentMap<String, AtomicInteger> repository) {
        Worker[] workers = new Worker[WORKER_COUNT];
        Worker worker;
        UncaughtExceptionHandler eh = new UncaughtExceptionHandler() {

            @Override
            public void uncaughtException(Thread t, Throwable e) {
                e.printStackTrace();

            }

        };

        for (int i = 0; i < WORKER_COUNT; i++) {
            worker = new Worker(repository, excludedOperationNames,
                includedOperationNames, destinationSysName);
            workers[i] = worker;
            worker.setUncaughtExceptionHandler(eh);
            worker.start();
        }

        return workers;
    }

    private void dispatchTask(BufferedReader fileNamesReader, Worker[] workers)
        throws IOException {

        String line;
        Set<String> fileNames = new HashSet<String>();

        int fileCount = 0;
        int workerIndex = -1;
        BufferedReader logFileReader;
        while ((line = fileNamesReader.readLine()) != null) {

            fileNames.add(line);
            fileCount++;
            if (0 == (fileCount % NUMBER_OF_FILES_FOR_EACH_DISPATCH)) {

                // 工作者线程间的负载均衡：采用简单的轮询选择 worker
                workerIndex = (workerIndex + 1) % WORKER_COUNT;
                logFileReader = makeReaderFrom(fileNames);

                Debug.info("Dispatch " + NUMBER_OF_FILES_FOR_EACH_DISPATCH
                    + " files to worker:" + workerIndex);
                workers[workerIndex].submitWorkload(logFileReader);

                fileNames = new HashSet<String>();
                fileCount = 0;
            }

        }

        if (fileCount > 0) {
```

```
                logFileReader = makeReaderFrom(fileNames);
                workerIndex = (workerIndex + 1) % WORKER_COUNT;
                workers[workerIndex].submitWorkload(logFileReader);
            }
        }

        private BufferedReader makeReaderFrom(final Set<String> logFileNames) {
            BufferedReader logFileReader;

            InputStream in = new SequenceInputStream(new Enumeration<InputStream>() {
                private Iterator<String> iterator = logFileNames.iterator();

                @Override
                public boolean hasMoreElements() {
                    return iterator.hasNext();
                }

                @Override
                public InputStream nextElement() {
                    String fileName = iterator.next();
                    InputStream in = null;
                    try {
                        in = new FileInputStream(logFileBaseDir + fileName);
                    } catch (FileNotFoundException e) {
                        throw new RuntimeException(e);
                    }
                    return in;
                }

            });
            logFileReader = new BufferedReader(new InputStreamReader(in));
            return logFileReader;
        }

    }

    // 模式角色：Master-Slave.Slave
    private static class Worker extends AbstractTerminatableThread {
        private static final Pattern SPLIT_PATTERN = Pattern.compile("\\|");
        private final ConcurrentMap<String, AtomicInteger> repository;
        private final BlockingQueue<BufferedReader> workQueue;

        private final String selfDevice = "ESB";
        private final String excludedOperationNames;
        private final String includedOperationNames;
        private final String destinationSysName;

        public Worker(ConcurrentMap<String, AtomicInteger> repository,
            String excludedOperationNames, String includedOperationNames,
            String destinationSysName) {
            this.repository = repository;
            workQueue = new ArrayBlockingQueue<BufferedReader>(100);
            this.excludedOperationNames = excludedOperationNames;
            this.includedOperationNames = includedOperationNames;
            this.destinationSysName = destinationSysName;
        }

        public void submitWorkload(BufferedReader taskWorkload) {
            try {
```

```
            workQueue.put(taskWorkload);
            terminationToken.reservations.incrementAndGet();
        } catch (InterruptedException e) {
            ;
        }

    }

    @Override
    protected void doRun() throws Exception {
        BufferedReader logFileReader = workQueue.take();

        String interfaceLogRecord;
        String[] recordParts;
        String timeStamp;
        AtomicInteger reqCounter;
        AtomicInteger existingReqCounter;
        int i = 0;

        try {
            while ((interfaceLogRecord = logFileReader.readLine()) != null) {
                recordParts = SPLIT_PATTERN.split(interfaceLogRecord, 0);

                // 避免 CPU 占用过高
                if (0 == ((++i) % 100000)) {
                    Thread.sleep(80);
                    i = 0;
                }

                // 跳过无效记录（如果有的话）
                if (recordParts.length < 7) {
                    continue;
                }

                // 只考虑表示发送请求给 selfDevice 所指定的系统的记录
                if (("request".equals(recordParts[2])) &&

                (recordParts[6].startsWith(selfDevice))) {

                    timeStamp = recordParts[0];

                    timeStamp = new String(timeStamp.substring(0,
                        19).toCharArray());

                    String operName = recordParts[4];

                    reqCounter = repository.get(timeStamp);
                    if (null == reqCounter) {
                        reqCounter = new AtomicInteger(0);
                        existingReqCounter = repository
                            .putIfAbsent(timeStamp, reqCounter);
                        if (null != existingReqCounter) {
                            reqCounter = existingReqCounter;
                        }
                    }

                    if (isSrcDeviceEEligible(recordParts[5])) {
```

```
                        if (excludedOperationNames.contains(operName + ',')) {

                            continue;

                        }

                        if ("*".equals(includedOperationNames)) {
                            reqCounter.incrementAndGet();

                        } else {
                            if (includedOperationNames.contains(operName +
                                ',')) {
                                reqCounter.incrementAndGet();
                            }
                        }
                    }
                }
            }

        } finally {
            terminationToken.reservations.decrementAndGet();
            logFileReader.close();
        }

    }

    // 判断目标设备名是否在待统计之列
    private boolean isSrcDeviceEEligible(String sourceNE) {

        boolean result = false;

        if ("*".equals(destinationSysName)) {

            result = true;

        } else if (destinationSysName.equals(sourceNE)) {

            result = true;

        }

        return result;
    }
}
}
```

清单 12-1 中的类 Master 相当于 Master-Slave 模式中的 Master 参与者实例，它将一组接口日志文件派发给 Slave 线程处理。类 Master 的 calculate 方法相当于图 12-1 中的 service 方法。类 Master 的 dispatchTask 方法对原始任务进行分解，并将分解得来的子任务派发给 Slave 线程处理。因此，calculate 方法相当于图 12-1 中的 splitWork 方法和 callSlaves 方法。清单 12-1 中的类 Slave 相当于 Master-Slave 模式中的 Slave 参与者实例，它负责对接口日志文件中的记录进行统计。每个 Slave 实例都是一个可停止的线程，其 submitWorkload 方法相当于图 12-1 中的 subService 方法。本案例中，我们使用了一个 java.util.concurrent.ConcurrentMap 实例，

用于存储各个子任务的处理结果。因此，Slave 实例往 ConcurrentMap 实例存储其处理结果的过程也正是原始任务的处理结果的形成过程。

本案例的主要计算集中在 Slave 线程上。为了避免 Slave 线程占用过多的资源，我们主要采取了两个措施。一个是，Master 类所创建的 Slave 线程数量为 JVM 所在主机的 CPU 个数（通过 Runtime.getRuntime().availableProcessors()获取）；另一个是 Slave 线程的 doRun 方法每处理完 10 万条记录就会休眠 80ms，这是为了避免其使得 CPU 过于繁忙。

清单 12-1 中的类 Slave 继承自 AbstractTerminatableThread 类。它使用了第 5 章介绍的 Two-phase Termination 模式。相关源码请见清单 5-3。

12.4 Master-Slave 模式的评价与实现考量

Master-Slave 模式的应用场景包括以下 3 个。

- **并行计算**（Parallel Computation）。该场景使用 Master-Slave 模式是为了提升计算性能。本章案例就属于该运用场景。在此情形下，原始任务的处理结果是通过组合每个 Slave 实例的处理结果形成的。
- **容错处理**（Fault Tolerance）。该场景使用 Master-Slave 模式是为了提高计算的可靠性。在此情形下，原始任务的处理结果是任意一个 Slave 实例的成功处理结果（那些处理失败的 Slave 实例无法为 Master 返回结果）。
- **计算精度**（Computational Accuracy）。该场景使用 Master-Slave 模式是为了提高计算的精确程度。在此情形下，原始任务的处理结果是所有 Slave 实例中处理结果不精确性最低的一个结果。

上述 3 种场景，只有第 1 种场景中所有 Slave 参与者实例都使用一个实现类，而后面两种场景下，不同 Slave 参与者实例对应着不同的实现类。此时，为了保证 Slave 参与者当实例在数量、类型上如果有变动对 Master 可能产生的影响最小，我们可能需要在 Slave 参与者中使用 Strategy（策略）模式，使得所有的 Slave 实现类都有一个共同的接口，如图 12-4 所示。

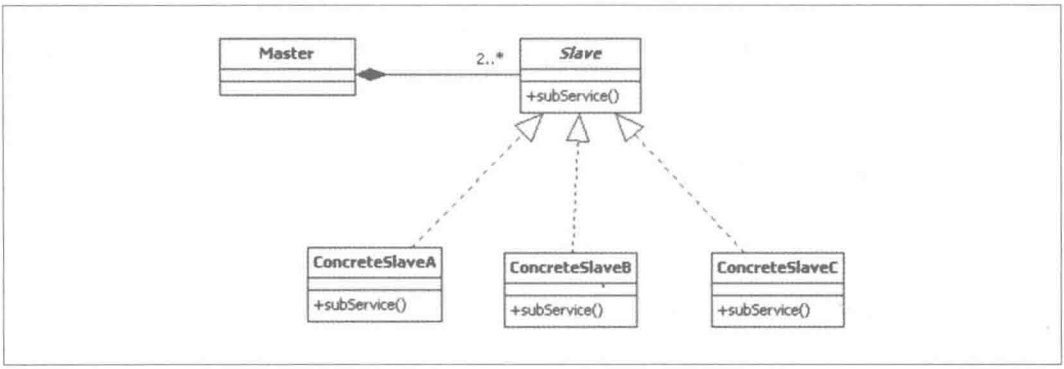


图 12-4. 实现同一接口的 Slave 参与者

使用 Master-Slave 模式能够带来以下几点好处。

- **可交换性（Exchangeability）和可扩展性（ Extensibility）**。如上文所述，如果我们使所有的 Slave 参与者实现类拥有共同的接口，那么替换某个 Slave 实例、增加一个 Slave 实例对 Master 参与者产生的影响很小。当然，客户端代码也不会受此影响。Master 进行任务分解、任务派发和对子任务处理结果的合并所涉及的算法如果有改变也不会影响到 Slave 参与者和客户端代码。
- **提升计算性能**。Master-Slave 模式实现恰当的话可以提升计算性能。这里，我们需要注意原始任务的分解、子任务的派发、子任务处理结果的合并以及 Slave 线程的管理都有其自身的时间和空间消耗。

Master-Slave 模式的架构虽然简单易于理解，但是在实现上该模式却不太简单。实现 Master-Slave 模式还需要考虑以下几个问题。

12.4.1 子任务的处理结果的收集

Master 参与者需要收集各个子任务的处理结果，才能生成原始任务的处理结果。收集子任务的处理结果通常有两种方法。一种是使用存储仓库（Repository）。所谓的存储仓库是一个 Master 参与者和 Slave 参与者都能够访问的数据结构。Slave 参与者实例将其子任务的处理结果存入存储仓库。Master 参与者可以等待各个 Slave 参与者实例处理完毕后从存储仓库中获取各个子任务的处理结果。本章案例就是采用这种方法收集子任务的处理结果。清单 12-1 中的类 Master 的实例变量 repository（其类型为 ConcurrentMap）就是一个用于收集子任务处理结果的存储仓库。另外一种方法是使用第 6 章中介绍的 Promise 模式。这里，我们可以使 Slave 参与者的 subService 方法的返回值为 Promise 模式中的 Promise 参与者实例。通常可以使 subService 方法的返回值为一个 java.util.concurrent.Future 实例。这样，Master 参与者可以通过调用各个 Slave 参与者实例的 subService 方法的返回值的 get()方法来获取子任务的处理结果。

12.4.2 Slave 参与者实例的负载均衡与工作窃取

在并行计算的场景中，为了使各个 Slave 参与者实例能够充分发挥其作用又不至于使其中某个负载过重，在子任务派发的时候我们需要注意各个 Slave 参与者实例所得到的子任务的规模和数量是否均衡。

如果原始任务的规模事先可知，那么 Master 参与者可以根据原始任务的规模和 Slave 参与者实例的个数算出的平均数进行子任务的派发[1]。这时，每个 Slave 参与者实例被分配到的子任务的总规模是相等的。在并行计算场景中，一个 Slave 参与者实例通常就是一个线程。为了避免增加 JVM 线程调度的负担，Slave 线程的数量一般要根据 JVM 所在主机的 CPU 个数来定。通常，Slave 线程数量只比 CPU 个数大一点。

如果原始任务的规模事先不可知，则 Master 参与者在派发子任务的时候可能要采用某种负载均衡算法来使得各个 Slave 线程所分配到的子任务的规模和数量是均衡的。本章案例就是属于这种情形。流量统计工具的 Java 模块是从标准输入中读取 Linux Shell 脚本模块输出的接口日志文件名，我们不想等到 Linux Shell 脚本模块输出其查找到的所有接口日志文件的文件名后再进行统计，因此对于 Java 模块而言，原始任务的规模是其事先不可知的。故其使用了简单的轮询（Round-Robin）算法（代码见清单 12-1）来保持各个 Slave 线程的子任务负载均衡。

上述的负载均衡着眼点在于子任务的派发上。Master-Slave 模式可以看成 Producer-Consumer 模式的一个实例。Master-Slave 模式的 Master 参与者和 Slave 参与者分别相当于 Producer-Consumer 模式（参见第 7 章）中的 Producer 参与者和 Consumer 参与者。因此，从 Slave 实例运行的角度来看，我们可以使用第 7 章中介绍的工作窃取算法来动态调整各个 Slave 实例的计算负载。

12.4.3 可靠性与异常处理

Master 参与者的实现类需要注意处理其与 Slave 参与者的通信异常以及 Slave 参与者对子任务处理的异常。

Master 参与者与 Slave 参与者的通信异常指 Master 参与者实例调用 Slave 参与者实例的 subService 方法时出现的异常。

由于 Slave 参与者实例是运行在工作者线程中的，而 Master 参与者实例是运行在客户端线程中的，后者要获取前者抛出的异常有点困难（详情参见第 6 章）。如果 Master 参与者利用上

1 如果原始任务规模很大，则可能需要将一组子任务派发给某个 Slave 参与者线程处理。

文所述的基于 Promise 模式的方法来获取子任务的处理结果，那么获取 Slave 参与者实例的处理异常就变得非常简单：Master 参与者实例调用 subService 返回值的 get()方法获取子任务处理结果的时候，如果 get()方法抛出异常则说明相应的子任务处理失败。

为了提高计算的可靠性，Master 参与者在侦测到上述异常时可以考虑由其自身重新执行处理失败的子任务，即让处理失败的子任务运行在客户端线程中，而不是在 Slave 参与者的工作者线程中。这类似于 ThreadPoolExecutor 提供的 java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy 这个任务提交失败处理策略。下文提到的 Master-Slave 模式的可复用代码中涉及的 SubTaskFailureException 异常类(代码见清单 12-8)和 RetryInfo 类(代码见清单 12-9)为这种重试机制提供了支持。

12.4.4 Slave 线程的停止

在并行计算场景中，每个 Slave 参与者实例就是一个线程。通常，这些线程会使用阻塞队列（BlockingQueue）来接收子任务，而线程则从阻塞队列中取出子任务（即调用 BlockingQueue 的 take()方法）进行执行。而 BlockingQueue 的 take()在队列为空时会使当前线程一直处于等待状态，因此当 Slave 线程处理完分配给其的所有子任务时，Slave 线程仍然未停止。此时，我们可以使用 Two-phase Termination 模式（参见第 5 章）来实现 Slave 线程在处理完分配给其的子任务后停止。本章案例就是采用这种方法来停止 Slave 线程。

12.5 Master-Slave 模式的可复用实现代码

Master-Slave 模式的可复用代码实现起来有些复杂，需要借助 Template（模板）模式、Strategy（策略）模式和 Factory Method（工厂方法）模式[2]。如图 12-5 所示的类图展示了 Master-Slave 模式的可复用代码所涉及的主要类及其间的关系。

2 GOF 设计模式，不在本书讨论范围之内。详情可参考《设计模式：可复用面向对象软件的基础（英文版）》一书。

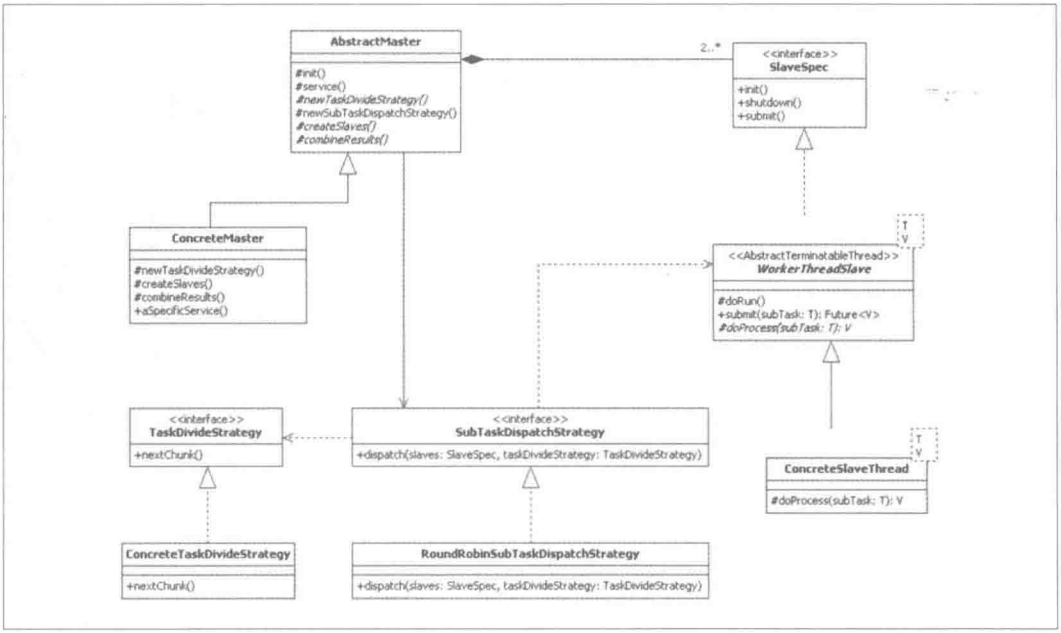


图 12-5. Master-Slave 模式的可复用代码涉及的主要类及其间的关系

如图 12-5 所示的类图中各个类的主要职责及其间关系如下。

- **TaskDivideStrategy**：原始任务分解算法策略，负责封装原始任务分解算法。这里，我们使用了 Strategy（策略）模式将一个算法封装为一个对象。其 nextChunk 方法每被调用一次会返回一个子任务，直到没有剩余的子任务。相应代码如清单 12-2 所示。
- **ConcreteTaskDivideStrategy**：TaskDivideStrategy 接口的实现类，由应用来实现。ConcreteMaster 类所实现的 newTaskDivideStrategy 方法返回该类的实例。
- **SubTaskDispatchStrategy**：子任务派发算法策略，负责封装子任务派发算法。这里，我们使用了 Strategy（策略）模式将一个算法封装为一个对象。其 dispatch 方法会循环调用 TaskDivideStrategy 实例的 nextChunk 方法来获取各个子任务，并将这些子任务派发给各个 Slave 实例。相应代码如清单\12-3 所示。
- **RoundRobinSubTaskDispatchStrategy**：SubTaskDispatchStrategy 接口的一个实现类，基于轮询算法（Round-Robin）的子任务派发算法策略。相应代码如清单 12-4 所示。当然，我们也可以根据应用的需要创建基于其他算法（如基于加权值的算法）的 SubTaskDispatchStrategy 接口实现类。
- **SlaveSpec**：对 Slave 参与者进行抽象。相应代码如清单 12-5 所示。
- **WorkerThreadSlave**：一个基于工作者线程的 Slave 参与者可复用抽象实现类。这里我们

使用了 Two-phase Termination 模式（参见第 5 章）来使 Slave 线程可优雅地停止。相应代码如清单 12-6 所示。WokerThreadSlave 的抽象方法 doProcess 用于其子类实现子任务处理逻辑。子任务处理抛出的异常会被 WokerThreadSlave 转换为 SubTaskFailureException 异常（相应代码见清单 12-8）。SubTaskFailureException 异常使得我们在子任务处理失败后可以进行重试。重试所需的信息包括处理失败的子任务及其处理逻辑都被封装在 RetryInfo 类（相应代码见清单 12-9）中。SubTaskFailureException 异常实例引用了相应的 RetryInfo 实例，Master-Slave 模式的客户端代码在获取子任务的处理结果时可以通过捕获 java.util.concurrent.ExecutionException 异常获得 SubTaskFailureException 异常实例进而获得 RetryInfo 实例。如清单 12-10 所示的代码的 combineResults 方法展示了一个基于 SubTaskFailureException 的异常处理逻辑。

- **ConcreteSlaveThread**：由应用所实现的 WorkerThreadSlave 的子类。其 doProcess 方法实现了对子任务的处理逻辑。
- **AbstractMaster**：Master 参与者的一个可复用实现。该类所做的事情主要有以下几种。
  - 调用其子类实现的 createSlaves 方法来创建 Slave 参与者实例。
  - 调用其子类实现的 newTaskDivideStrategy 方法来创建一个原始任务分解算法策略对象。
  - 调用其子类实现的 newSubTaskDispatchStrategy 方法来创建一个子任务派发算法策略对象。该类默认创建的子任务派发算法策略对象是一个 RoundRobinSubTaskDispatchStrategy 实例。
  - 调用子任务派发算法策略对象的 dispatch 方法来派发各个子任务。
  - 调用其子类实现的 combineResults 方法来合并子任务的处理结果。

这里，我们使用了 Factory Method（工厂方法）模式使得 AbstractMaster 创建的 Slave 参与者实例、原始任务分解算法策略实例和子任务派发算法策略实例的具体实现类可以由子类（应用代码）来指定。相应代码如清单 12-7 所示。

- **ConcreteMaster**：一个由应用实现的 AbstractMaster 类的子类。该类除了实现其父类定义的几个抽象方法外，还定义一个服务方法。该服务方法对外呈现一个名称含义更为具体的接口。实现上，它直接调用其父类的 service 方法。

清单 12-2. TaskDivideStrategy 接口源码

```
/**
 * 对原始任务分解算法策略的抽象。
 *
 * @author Viscent Huang
```

```
 *
 * @param <T>
 *          子任务类型
 */
public interface TaskDivideStrategy<T> {
    /**
     * 返回下一个子任务。 若返回值为 null，则表示无后续子任务。
     *
     * @return 下一个子任务
     */
    T nextChunk();
}
```

清单 12-3. SubTaskDispatchStrategy 接口源码

```
/**
 * 对子任务派发算法策略的抽象。
 * @author Viscent Huang
 *
 * @param <T> 子任务类型
 * @param <V> 子任务处理结果类型
 */
public interface SubTaskDispatchStrategy<T,V> {
    /**
     * 根据指定的原始任务分解策略，将分解得来的各个子任务派发给一组 Slave 参与者实例。
     *
     * @param slaves 可以接受子任务的一组 Slave 参与者实例
     * @param taskDivideStrategy 原始任务分解策略
     * @return iterator。遍历该 iterator 可得到用于获取子任务处理结果的 Promise（参见第 6 章，
     * Promise 模式）实例。
     * @throws InterruptedException 当 Slave 工作者线程被中断时抛出该异常。
     */
    Iterator<Future<V>> dispatch(Set<? extends SlaveSpec<T, V>> slaves,
        TaskDivideStrategy<T> taskDivideStrategy) throws InterruptedException;
}
```

清单 12-4. RoundRobinSubTaskDispatchStrategy 类源码

```
/**
 * 简单的轮询（Round-Robin）派发算法策略。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *          原始任务类型
 * @param <V>
 *          子任务处理结果类型
 */
public class RoundRobinSubTaskDispatchStrategy<T, V> implements
    SubTaskDispatchStrategy<T, V> {

    @SuppressWarnings("unchecked")
    @Override
    public Iterator<Future<V>> dispatch(Set<? extends SlaveSpec<T, V>> slaves,
        TaskDivideStrategy<T> taskDivideStrategy) throws InterruptedException {
        final List<Future<V>> subResults = new LinkedList<Future<V>>();
        T subTask;
```

```
        Object[] arrSlaves = slaves.toArray();
        int i = -1;
        final int slaveCount = arrSlaves.length;
        Future<V> subTaskResultPromise;

        while (null != (subTask = taskDivideStrategy.nextChunk())) {
            i = (i + 1) % slaveCount;
            subTaskResultPromise = ((WorkerThreadSlave<T, V>) arrSlaves[i])
                .submit(subTask);
            subResults.add(subTaskResultPromise);
        }

        return subResults.iterator();
    }

}
```

清单 12-5. SlaveSpec 接口源码

```
/**
 * 对 Master-Slave 模式 Slave 参与者的抽象。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *          子任务类型
 * @param <V>
 *          子任务处理结果类型
 */
public interface SlaveSpec<T, V> {
    /**
     * 用于 Master 向其提交一个子任务。
     *
     * @param task
     *          子任务
     * @return 可借以获取子任务处理结果的 Promise（参见第 6 章，Promise 模式）实例。
     * @throws InterruptedException
     */
    Future<V> submit(final T task) throws InterruptedException;

    /**
     * 初始化 Slave 实例提供的服务
     */
    void init();

    /**
     * 停止 Slave 实例对外提供的服务
     */
    void shutdown();
}
```

清单 12-6. WorkerThreadSlave 类源码

```
/**
 * 基于工作者线程的 Slave 参与者通用实现。
 *
 * @author Viscent Huang
```

```
 *
 * @param <T>
 *        子任务类型
 * @param <V>
 *        子任务处理结果类型
 */
public abstract class WorkerThreadSlave<T, V> extends
    AbstractTerminatableThread implements SlaveSpec<T, V> {
    private final BlockingQueue<Runnable> taskQueue;

    public WorkerThreadSlave(BlockingQueue<Runnable> taskQueue) {
        this.taskQueue = taskQueue;
    }

    public Future<V> submit(final T task) throws InterruptedException {

        FutureTask<V> ft = new FutureTask<V>(new Callable<V>() {

            @Override
            public V call() throws Exception {
                V result;

                try {
                    result = doProcess(task);
                } catch (Exception e) {
                    SubTaskFailureException stfe = newSubTaskFailureException(task, e);
                    throw stfe;
                }

                return result;
            }

        });
        taskQueue.put(ft);
        terminationToken.reservations.incrementAndGet();
        return ft;
    }

    private SubTaskFailureException newSubTaskFailureException(final T subTask,
        Exception cause) {
        RetryInfo<T, V> retryInfo = new RetryInfo<T, V>(subTask, new Callable<V>() {
            @Override
            public V call() throws Exception {
                V result;
                result = doProcess(subTask);
                return result;
            }
        });

        return new SubTaskFailureException(retryInfo, cause);
    }

    /**
     * 留给子类实现。用于实现子任务的处理逻辑。
     *
     * @param task
     *        子任务
     * @return 子任务的处理结果
```

```
     * @throws Exception
     */
    protected abstract V doProcess(T task) throws Exception;

    @Override
    protected void doRun() throws Exception {
        try {
            Runnable task = taskQueue.take();
            task.run();
        } finally {
            terminationToken.reservations.decrementAndGet();
        }

    }

    @Override
    public void init() {
        start();
    }

    @Override
    public void shutdown() {
        terminate(true);
    }
}
```

清单 12-7. AbstractMaster 类源码

```
/**
 * Master-Slave 模式 Master 参与者的可复用实现。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *          子任务对象类型
 * @param <V>
 *          子任务的处理结果类型
 * @param <R>
 *          原始任务处理结果类型
 */
public abstract class AbstractMaster<T, V, R> {
    protected volatile Set<? extends SlaveSpec<T, V>> slaves;

    // 子任务派发算法策略
    private volatile SubTaskDispatchStrategy<T, V> dispatchStrategy;

    public AbstractMaster() {
    }

    protected void init() {
        slaves = createSlaves();
        dispatchStrategy = newSubTaskDispatchStrategy();
        for (SlaveSpec<T, V> slave : slaves) {
            slave.init();
        }
    }

    /**
```

```
     * 对子类暴露的服务方法。该类的子类需要定义一个比该方法命名更为具体的服务方法（如
     * downloadFileService）。
     * 由命名含义具体的服务方法（如 downloadFileService）调用该方法。
     * 该方法使用了 Template（模板）模式、Strategy（策略）模式。
     *
     * @param params
     *          客户端代码传递的参数列表
     */
    protected R service(Object... params) throws Exception {
        final TaskDivideStrategy<T> taskDivideStrategy =
            newTaskDivideStrategy(params);

        /*
         * 对原始任务进行分解，并将分解得来的子任务派发给 Slave 参与者实例。 这里使用了 Strategy
         *（策略）模式：原始任务分解和子任务派发
         * 这两个具体的计算是通过调用需要的算法策略（对象）实现的。
         */
        Iterator<Future<V>> subResults = dispatchStrategy.dispatch(slaves,
            taskDivideStrategy);

        // 等待 Slave 实例处理结束
        for (SlaveSpec<T, V> slave : slaves) {
            slave.shutdown();
        }

        // 合并子任务的处理结果
        R result = combineResults(subResults);
        return result;
    }

    /**
     * 留给子类实现。用于创建原始任务分解算法策略。
     *
     * @param params
     *          客户端代码调用 service 方法时传递的参数列表
     */
    protected abstract TaskDivideStrategy<T> newTaskDivideStrategy(
        Object... params);

    /**
     * 用于创建子任务派发算法策略。默认使用轮询（Round-Robin）派发算法。
     *
     * @return 子任务派发算法策略实例。
     */
    protected SubTaskDispatchStrategy<T, V> newSubTaskDispatchStrategy() {
        return new RoundRobinSubTaskDispatchStrategy<T, V>();
    }

    /**
     * 留给子类实现。用于创建 Slave 参与者实例。
     *
     * @return 一组 Slave 参与者实例。
     */
    protected abstract Set<? extends SlaveSpec<T, V>> createSlaves();

    /**
```

```
     * 留给子类实现。用于合并子任务的处理结果。
     *
     * @param subResults
     *          各个子任务处理结果
     * @return 原始任务的处理结果
     */
    protected abstract R combineResults(Iterator<Future<V>> subResults);

}
```

清单 12-8. SubTaskFailureException 类源码

```
/**
 * 表示子任务处理失败的异常。
 *
 * @author Viscent Huang
 */
@SuppressWarnings("serial")
public class SubTaskFailureException extends Exception {

    /**
     * 对处理失败的子任务进行重试所需的信息
     */
    @SuppressWarnings("rawtypes")
    public final RetryInfo retryInfo;

    @SuppressWarnings("rawtypes")
    public SubTaskFailureException(RetryInfo retryInfo, Exception cause) {
        super(cause);
        this.retryInfo = retryInfo;
    }

}
```

清单 12-9. RetryInfo 类源码

```
/**
 * 对处理失败的子任务进行重试所需的信息。
 *
 * @author Viscent Huang
 *
 * @param <T> 子任务类型
 * @param <V> 子任务处理结果类型
 */
public class RetryInfo<T, V> {
    public final T subTask;
    public final Callable<V> redoCommand;

    public RetryInfo(T subTask, Callable<V> redoCommand) {
        this.subTask = subTask;
        this.redoCommand = redoCommand;
    }
}
```

使用可复用代码实现 Master-Slave 模式，应用代码需要完成以下几件事情。

1.【必需】创建 TaskDivideStrategy 接口的实现类，在该类中实现原始任务分解算法。

2.【必需】创建 AbstractMaster 的子类。该子类除了实现其父类定义的几个抽象方法外，还要定义服务方法，该服务方法的名字比其父类的 service 方法含义更为具体。

3.【必需】创建 WorkerThreadSlave 的子类。在该子类中实现其父类的 doProcess 方法。当然，我们也可以自己编写 SlaveSpec 接口的实现类。

4.【可选】创建 SubTaskDispatchStrategy 的实现类。在该类中实现子任务派发算法。AbstactMaster 默认使用 RoundRobinTaskDispatchStrategy。

清单 12-10 展示了一个使用 Master-Slave 模式可复用代码的例子。

清单 12-10. 基于 Master-Slave 模式可复用代码的质数生成器

```
public class ParallelPrimeGenerator {

    public static void main(String[] args) throws Exception {
        PrimeGeneratorService primeGeneratorService = new PrimeGeneratorService();

        Set<BigInteger> result = primeGeneratorService.generatePrime(Integer
            .valueOf(args[0]));
        System.out.println("Generated " + result.size() + " prime:");
        System.out.println(result);

    }

}

class Range {
    public final int lowerBound;
    public final int upperBound;

    public Range(int lowerBound, int upperBound) {
        if (upperBound < lowerBound) {
            throw new IllegalArgumentException(
                "upperBound should not be less than lowerBound!");
        }
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }

    @Override
    public String toString() {
        return "Range [" + lowerBound + ", " + upperBound + "]";
    }

}

/*
 * 质数生成器服务。模式角色：Master-Slave.Master
 */
class PrimeGeneratorService extends
   AbstractMaster<Range, Set<BigInteger>, Set<BigInteger>> {
```

```
public PrimeGeneratorService() {
    this.init();
}

// 创建子任务分解算法实现类
@Override
protected TaskDivideStrategy<Range> newTaskDivideStrategy(
    final Object... params) {

    final int numOfSlaves = slaves.size();
    final int originalTaskScale = (Integer)params[0];
    final int subTaskScale = originalTaskScale / numOfSlaves;
    final int subTasksCount = (0 == (originalTaskScale % numOfSlaves)) ?
        numOfSlaves: numOfSlaves + 1;

    TaskDivideStrategy<Range> tds = new TaskDivideStrategy<Range>() {
        private int i = 1;

        @Override
        public Range nextChunk() {
            int upperBound;
            if (i < subTasksCount) {
                upperBound = i * subTaskScale;
            } else if (i == subTasksCount) {
                upperBound = originalTaskScale;
            } else {
                return null;
            }

            int lowerBound = (i - 1) * subTaskScale + 1;
            i++;

            return new Range(lowerBound, upperBound);
        }

    };
    return tds;
}

// 创建 Slave 线程
@Override
protected Set<? extends SlaveSpec<Range, Set<BigInteger>>> createSlaves() {
    Set<PrimeGenerator> slaves = new HashSet<PrimeGenerator>();
    for (int i = 0; i < Runtime.getRuntime().availableProcessors(); i++) {
        slaves.add(new PrimeGenerator(new ArrayBlockingQueue<Runnable>(2)));
    }
    return slaves;
}
// 组合子任务的处理结果
@Override
protected Set<BigInteger> combineResults(
    Iterator<Future<Set<BigInteger>>> subResults) {

    Set<BigInteger> result = new TreeSet<BigInteger>();

    while (subResults.hasNext()) {
```

```
            try {
                result.addAll(subResults.next().get());
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                Throwable cause = e.getCause();
                if (SubTaskFailureException.class.isInstance(cause)) {
                    @SuppressWarnings("rawtypes")
                    RetryInfo retryInfo = ((SubTaskFailureException)
                        cause).retryInfo;
                    Object subTask = retryInfo.subTask;
                    Debug.info("failed subTask:" + subTask);
                    e.printStackTrace();
                }
            }

        }

        return result;

    }

    // 使 Master 子类对外保留一个含义具体的方法名
    public Set<BigInteger> generatePrime(int upperBound) throws Exception {
        return this.service(upperBound);
    }

    /*
     * 质数生成器。模式角色：Master-Slave.Slave
     */
    private static class PrimeGenerator extends
        WorkerThreadSlave<Range, Set<BigInteger>> {

        public PrimeGenerator(BlockingQueue<Runnable> taskQueue) {
            super(taskQueue);
        }

        @Override
        protected Set<BigInteger> doProcess(Range range) throws Exception {

            Set<BigInteger> result = new TreeSet<BigInteger>();
            BigInteger start = BigInteger.valueOf(range.lowerBound);
            BigInteger end = BigInteger.valueOf(range.upperBound);

            while (-1 == (start = start.nextProbablePrime()).compareTo(end)) {
                result.add(start);
            }

            return result;
        }

    }

}
```

12.6 Java 标准库实例

Java 标准库中没有使用 Master-Slave 模式。

12.7 相关模式

12.7.1 Two-phase Termination 模式（第 5 章）

Master-Slave 模式的 Slave 参与者使用 Two-phase Termination 模式来实现其线程的停止。

12.7.2 Promise 模式（第 6 章）

Master-Slave 模式可能使用 Promise 模式来实现 Slave 返回其子任务处理结果。此时，Slave 参与者相当于 Promise 模式中的 Promisor 参与者，其 subService 方法的返回值是一个 Promise 模式中的 Promise 参与者实例。

与 Master-Slave 模式有关的 GOF 设计模式有后面的 3 个模式。

12.7.3 Strategy（策略）模式

实现 Master-Slave 模式的可复用代码时，可使用 Strategy 模式将原始任务分解算法和子任务派发算法封装为相应的对象。

12.7.4 Template（模板）模式

实现 Master-Slave 模式的可复用代码实现时，可使用 Template 模式封装 Master 参与者的处理步骤：分解原始任务、派发子任务、创建 Slave 参与者实例和合并各个子任务的处理结果。

12.7.5 Factory Method（工厂方法）模式

实现 Master-Slave 模式的可复用代码时，可使用 Factory Method 模式来创建 Slave 参与者实例、原始任务分解算法策略实例和子任务分解算法策略实例。

12.8 参考资源

1．Schmidt, Douglas et al. Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley, 2000.

2．Erich Gamma 等. 设计模式：可复用面向对象软件的基础（英文版）. 机械工业出版社，2002.

第 13 章

Pipeline（流水线）模式

13.1 Pipeline 模式简介

Pipeline 模式的核心思想是将一个任务处理分解为若干个处理阶段（Stage），其中每个处理阶段的输出作为下一个处理阶段的输入，并且各个处理阶段都有相应的工作者线程去执行相应的计算。因此，处理一批任务时，各个任务的各个处理阶段是并行（Parallel）的。通过并行计算，Pipeline 模式使应用程序能够充分利用多核 CPU 资源，提高其计算效率。

假设有一批任务（T1、T2、T3、T4），其中每个任务的处理可分解为 3 个处理阶段。虽然，对于这一批任务中的某一个任务而言，其处理仍然是串行的，即完成一个任务的处理要依次执行各个处理阶段，但从整体任务上看，各个处理阶段的执行是并行的。比如，处理阶段 1 的工作者线程执行完任务 T1 相应的计算后，其处理结果会被提交给处理阶段 2 作为输入；当处理阶段 2 的工作者线程正在执行任务 T1 的相应的计算时，处理阶段 1 正在执行任务 T2 相应的计算，此时这两个处理阶段是并行的，如图 13-1 所示。

Pipeline 模式类似于生活中顾客去饭店吃饭的情形：从顾客的迎接和安排座位、点菜、菜肴烹煮、上菜到买单，每个阶段都有相应的工作人员（迎宾、服务员、厨师、传菜员和收银员）负责，而不是由一名工作人员对一桌客人负责到底，从而减少了顾客的等待，提高了服务效率。

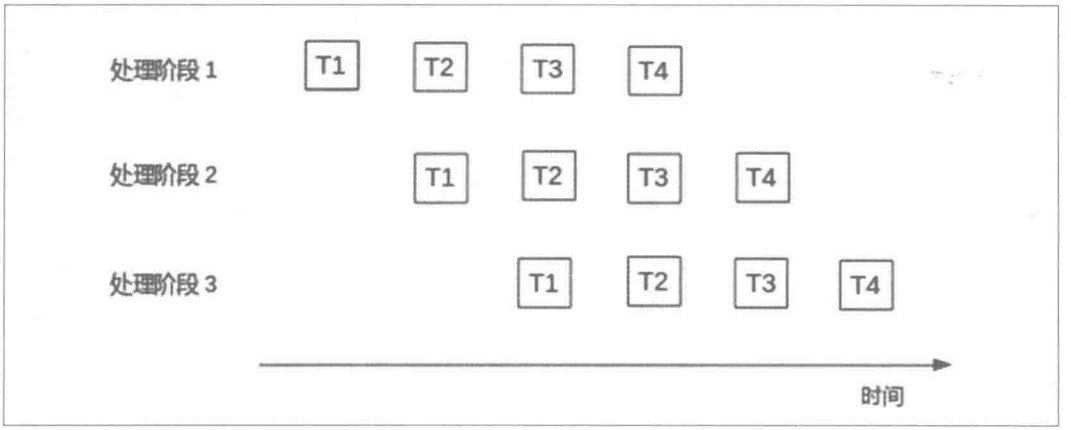


图 13-1. Pipeline 模式示意图

13.2 Pipeline 模式的架构

Pipeline 模式将任务的处理分解为若干个处理阶段，并将其中的每个阶段抽象为一个对象。这些表示处理阶段的对象都有其工作者线程负责对输入进行处理，并将输出作为下一个处理阶段的输入。

按照处理阶段中是否包含并发处理阶段，Pipeline 模式可分为线性 Pipeline 和非线性 Pipeline 两种，如图 13-2 所示。图 13-2 中的非线性 Pipeline 中的第 3 个处理阶段包含了两个可并发执行的操作。

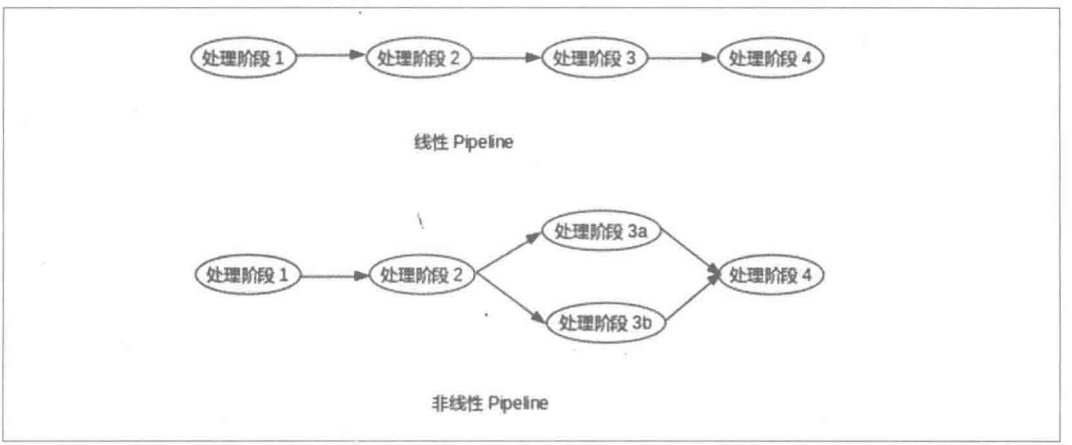


图 13-2. Pipeline 模式的分类

Pipeline 模式的主要参与者有以下几种。其类图如图 13-3 所示。

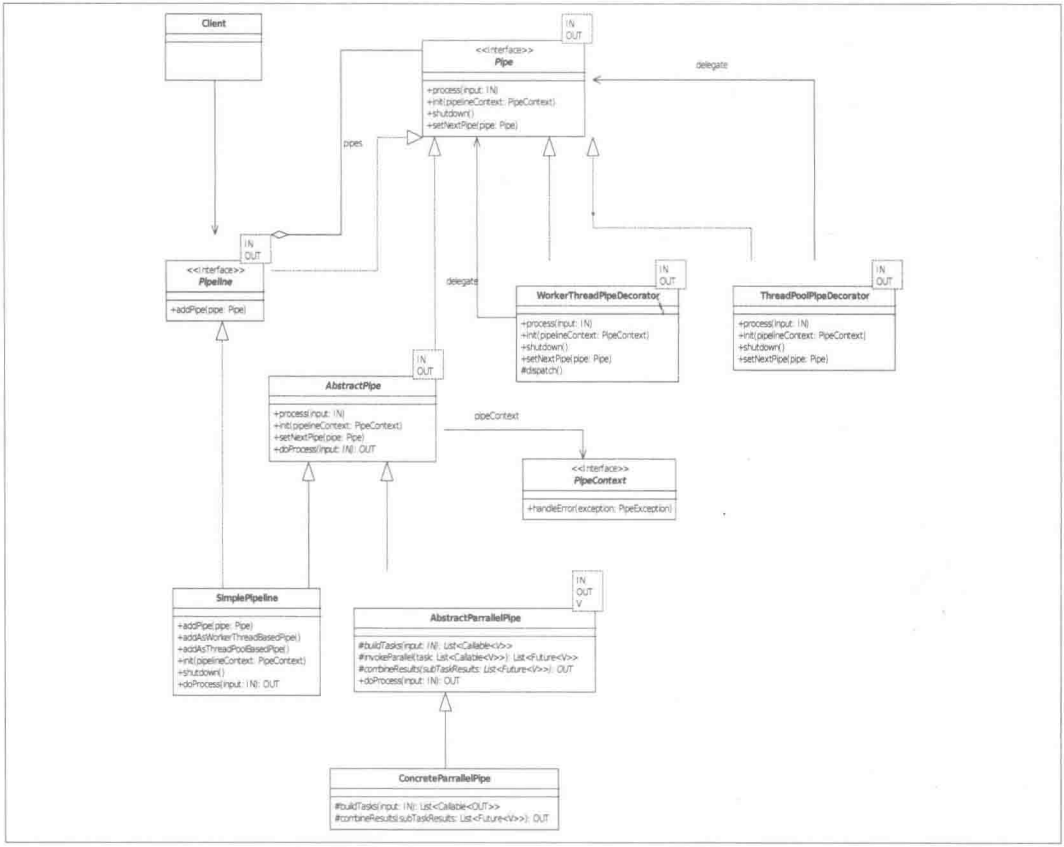


图 13-3. Pipeline 模式的类图

- **Pipe**：对处理阶段的抽象。负责对输入进行处理，并将输出作为下一个处理阶段的输入。因此，一个 Pipe 实例可以理解为包含输入、输出和处理的三元组。其主要方法及职责如下。
  - **process**：用于接收前一个处理阶段的处理结果，作为该处理阶段的输入。
  - **init**：初始化当前处理阶段对外提供的服务。
  - **shutdown**：关闭当前处理阶段对外提供的服务。
  - **setNextPipe**：设置当前处理阶段的下一个处理阶段。
- **PipeContext**：对各个处理阶段的计算环境进行抽象，主要用于异常处理。其主要方法

及职责如下。

- **handleError**：用于对处理阶段抛出的异常进行处理。

- **AbstractPipe**：Pipe 接口的抽象实现类。其主要方法及职责如下。
  - **process**：接收前一个处理阶段的输入，并调用其子类实现的 doProcess 方法对输入元素进行处理。相应的处理结果会被提交给下一个处理阶段作为输入。
  - **init**：保存对其参数中指定的 PipeContext 实例的引用，子类可根据需要覆盖该方法以实现其服务的初始化。
- **shutdown**：默认实现什么也不做。子类可根据需要覆盖该方法实现服务停止。
  - **setNextPipe**：设置当前处理阶段的下一个处理阶段。
  - **doProcess**：留给子类实现的抽象方法。用于子类实现其对输入元素的处理逻辑。
- **WorkerThreadPipeDecorator**：基于工作者线程的 Pipe 实现类。该 Pipe 实例会将接收到的输入元素存入队列，由指定个数的工作者线程对队列中的输入元素进行处理。该类自身主要负责工作者线程的生命周期的管理。它通过调用 delegate 实例变量所指定的 Pipe 实例（以下称为委托 Pipe 实例）的相应方法实现 Pipe 接口中定义的各个方法。其主要方法及职责如下。
  - **process**：接收前一个处理阶段的输入，并将其存入队列，由工作者线程运行时取出进行处理。
  - **init**：启动工作者线程，并调用委托 Pipe 实例的 init 方法。
  - **shutdown**：停止工作者线程，并调用委托 Pipe 实例的 shutdown 方法。
  - **setNextPipe**：调用委托 Pipe 实例的 setNextPipe 方法。
  - **dispatch**：取队列中的输入元素，并调用委托 Pipe 实例的 process 方法对其进行处理。
- **ThreadPoolPipeDecorator**：基于线程池的 Pipe 实现类。该类主要实现用线程池中的工作者线程去执行对各个输入元素的处理。它通过 delegate 实例变量所指定的 Pipe 实例（以下称为委托 Pipe 实例）的相应方法实现 Pipe 接口中定义的各个方法。其主要方法及职责如下。
  - **process**：接收前一个处理阶段的输入，并向线程池提交一个对该输入进行相应处理的任务。

- **init**：用委托 Pipe 实例的 init 方法。
- **shutdown**：关闭当前 Pipe 实例对外提供的服务，并调用委托 Pipe 实例的 shutdown 方法。
- **setNextPipe**：调用委托 Pipe 实例的 setNextPipe 方法。

- **AbstractParallelPipe**：AbstractPipe 的子类，支持并行处理的 Pipe 实现类。该类对其每个输入元素(原始任务)生成相应的一组子任务，并以并行的方式去执行这些子任务。各个子任务的执行结果会被合并为相应原始任务的输出结果。其主要方法及职责如下。
  - **buildTasks**：留给子类实现的抽象方法。用于根据指定的输入构造一组子任务。
  - **combineResults**：留给子类实现的抽象方法。对各个并行子任务的处理结果进行合并，形成相应输入元素的输出结果。
  - **invokeParallel**：实现以并行的方式执行一组任务。
  - **doProcess**：实现该类对其输入的处理逻辑。
- **ConcreteParallelPipe**：由应用定义的 AbstractParallelPipe 的子类。其主要方法及职责如下。
  - **buildTasks**：根据指定的输入构造一组子任务。
  - **combineResults**：对各个并行子任务的处理结果进行合并，形成相应输入元素的输出结果。
- **Pipeline**：对复合 Pipe 的抽象。一个 Pipeline 实例可包含多个 Pipe 实例。其主要方法及职责如下。
  - **addPipe**：往该 Pipeline 实例中添加一个 Pipe 实例。
- **SimplePipeline**：基于 AbstractPipe 的 Pipeline 接口的一个简单实现类。其主要方法及职责如下。
  - **addPipe**：往该 Pipeline 实例中添加一个 Pipe 实例。
  - **addAsWorkerThreadBasedPipe**：将指定的 Pipe 实例用 WorkerThreadPipeDecorator 实例包装后加入 Pipeline 实例。
  - **addAsThreadPoolBasedPipe**：将指定的 Pipe 实例用 ThreadPoolPipeDecorator 实例包装后加入 Pipeline 实例。

Pipeline 模式的服务初始化序列图如图 13-4 所示。

第 1 步：客户端代码创建 Pipeline 参与者实例。

第 2~4 步：客户端代码创建其所需的各个 Pipe 参与者实例。

第 5 步：客户端代码调用 Pipeline 实例的 add 方法添加其创建的 Pipe 实例。

第 6 步：客户端代码创建 PipeContext 参与者实例。

第 7 步：客户端代码调用 Pipeline 实例的 init 方法。

第 8~13 步：init 方法调用其所属 Pipeline 实例所包含的各个 Pipe 实例的 init 方法，以初始化这些 Pipe 实例所提供的服务。

第 14 步：init 方法调用返回。

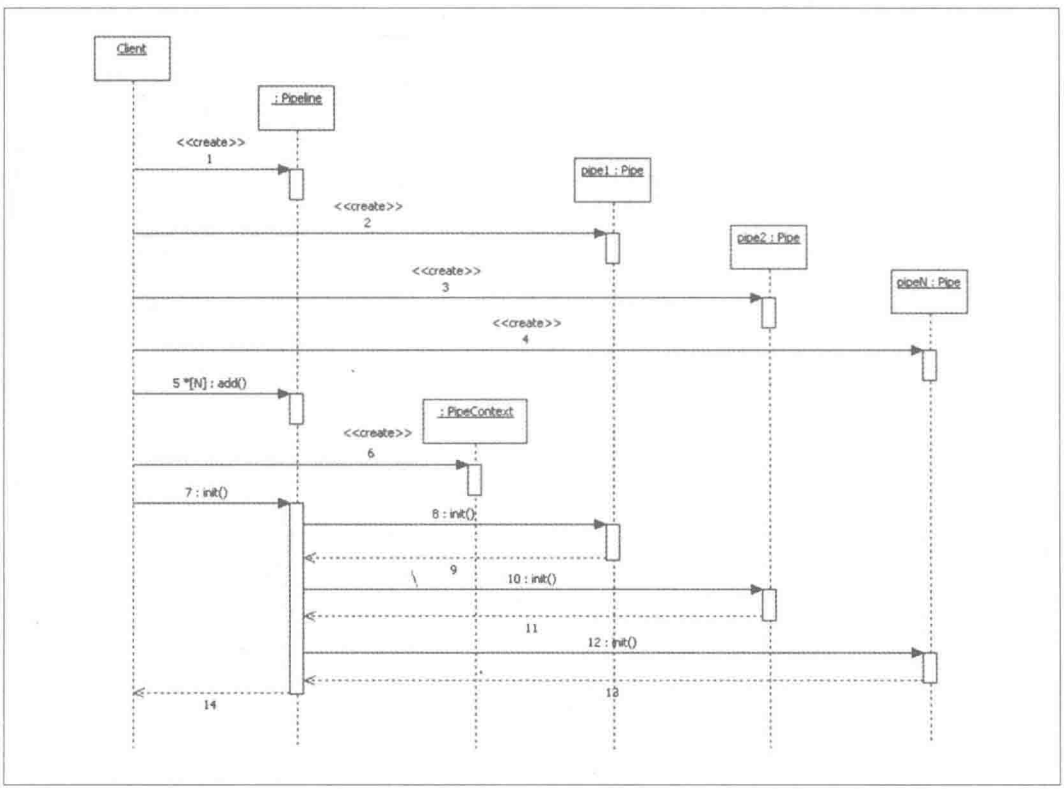


图 13-4. Pipeline 模式的服务初始化序列图

如图 13-5 所示的序列图展示了客户端代码调用基于 WorkerThreadPipeDecorator 和 AbstractPipe 实现的 Pipeline 模式服务。

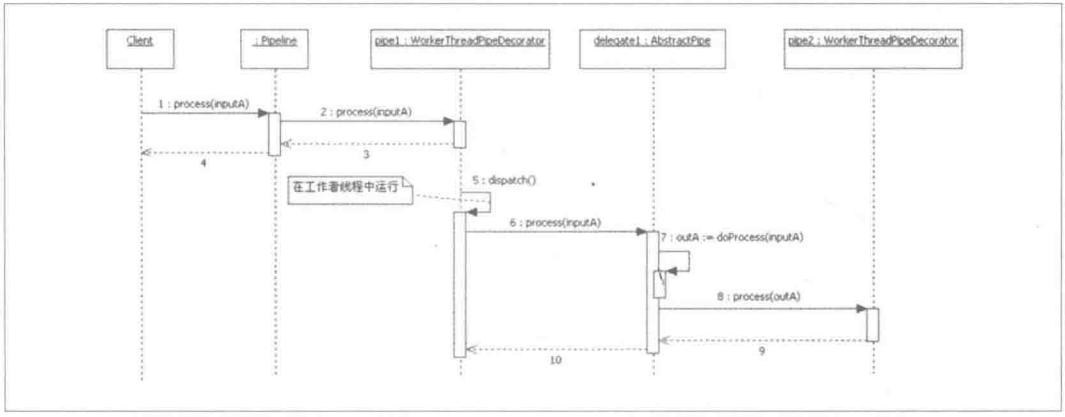


图 13-5. Pipeline 模式的服务调用序列图

第 1 步：客户端代码调用 Pipeline 参与者实例的 process 方法提交一个任务。

第 2、3 步：process 方法调用当前 Pipeline 实例所包含的第一个 Pipe 实例的 process 方法。

第 4 步：process 方法返回。

第 5 步：Pipeline 实例中的第一个 Pipe 实例开始其任务处理，其 dispatch 方法被工作者线程（或者线程池中的工作者线程）调用。

第 6 步：dispatch 方法调用 pipe1 的委托 Pipe 实例 delegate1 的 process 方法。

第 7 步：delegate1 的 process 方法调用其 doProcess 方法对输入元素 inputA 进行处理，相应处理结果为 outA。

第 8、9 步：delegate1 的 process 方法获取其下一个 Pipe 实例（同时也是 pipe1 的下一个 Pipe 实例）pipe2，并调用 pipe2 的 process 方法，将 outA 作为输入元素提交给 pipe2 处理。

第 10 步：pipe1 的 process 方法返回。

13.3 Pipeline 模式实战案例解析

某系统需要一个数据同步的定时任务。该定时任务将数据库中符合指定条件的记录数据以文件的形式 FTP 传输（同步）到指定的主机上。该定时任务需要满足以下几个要求。

1．每个数据文件最多只包含 N（如 10000，具体可配置）条记录；当一个数据文件被写满时，其他待写记录会被写入新的数据文件。

2．每个数据文件可能需要被传输到多台主机上。

3．本地要保留同步过的数据文件的备份。

因此，该定时任务所要做的事情主要包括：查询数据库（以下称为 Stage1）、根据数据库查询结果集生成本地数据文件（以下称为 Stage2）、FTP 传输各个数据文件到指定的主机（支持多台主机，以下称为 Stage3）、备份传输完毕（或者失败）的数据文件（以下称为 Stage4）。显然，该任务处理涉及比较多的 I/O 操作：查询数据库和传输数据文件均涉及网络 I/O 和文件 I/O，备份传输过的文件涉及文件 I/O。所以，该任务的处理逻辑不适宜用单线程实现。因为采用单线程模型上述处理步骤只能是按顺序串行执行，而这会使各个处理步骤所涉及的 I/O 等待的负面影响放大。另外，如果采用多个线程，每个线程中仍然是按顺序串行处理（每个线程先后执行 Stage2、Stage3 和 Stage4）也是不适合的，这样会导致执行 Stage2 时多个线程需要征用同一个数据文件（因为这个文件还没有被写满）。

这里，Pipeline 模式可以派上用场：采用一个线程去负责 Stage1 的执行。其余处理步骤（Stage2、Stage3 和 Stage4）中的每一个步骤都有专门的工作者线程去负责处理。这样，从个体任务上看，上述处理步骤虽然仍然是按顺序串行处理，但从整体任务上看几个步骤却是并行的，从而提升了计算效率。该定时任务的代码如清单 13-1 所示。

清单 13-1. 数据同步定时任务源码

```
public class DataSyncTask implements Runnable {

    public void run() {
        ResultSet rs = null;
        SimplePipeline<RecordSaveTask, String> pipeline = buildPipeline();
        pipeline.init(pipeline.newDefaultPipeContext());

        Connection dbConn = null;
        try {
            dbConn = getConnection();
            rs = qryRecords(dbConn);

            processRecords(rs, pipeline);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (null != dbConn) {
                try {
                    dbConn.close();
                } catch (SQLException e) {
                    ;
                }
```

```
            }
        }
        pipeline.shutdown(360, TimeUnit.SECONDS);
    }

    private ResultSet qryRecords(Connection dbConn) throws Exception {

        dbConn.setReadOnly(true);
        PreparedStatement ps = dbConn
            .prepareStatement(
                "select id,productId,packageId,msisdn,operationTime,operationType,"
                    + "effectiveDate,dueDate from subscriptions order by
                     operationTime",
                ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
        ResultSet rs = ps.executeQuery();
        return rs;
    }

    private static Connection getConnection() throws Exception {
        Connection dbConn = null;

        Class.forName("org.hsqldb.jdbc.JDBCDriver");

        dbConn = DriverManager.getConnection(
            "jdbc:hsqldb:hsql://192.168.1.105:9001/viscent-test", "SA", "");
        return dbConn;
    }

    private static Record makeRecordFrom(ResultSet rs) throws SQLException {
        Record record = new Record();

        record.setId(rs.getInt("id"));
        record.setProductId(rs.getString("productId"));
        record.setPackageId(rs.getString("packageId"));
        record.setMsisdn(rs.getString("msisdn"));
        record.setOperationTime(rs.getTimestamp("operationTime"));
        record.setOperationType(rs.getInt("operationType"));
        record.setEffectiveDate(rs.getTimestamp("effectiveDate"));
        record.setDueDate(rs.getTimestamp("dueDate"));
        return record;
    }

    private static class RecordSaveTask {
        public final Record[] records;
        public final int targetFileIndex;
        public final String recordDay;

        public RecordSaveTask(Record[] records, int targetFileIndex) {
            this.records = records;
            this.targetFileIndex = targetFileIndex;
            this.recordDay = null;
        }

        public RecordSaveTask(String recordDay, int targetFileIndex) {
            this.records = null;
            this.targetFileIndex = targetFileIndex;
            this.recordDay = recordDay;
        }
```

```
}

@SuppressWarnings("unchecked")
private SimplePipeline<RecordSaveTask, String> buildPipeline() {
    /*
     * 线程池的本质是重复利用一定数量的线程，而不是针对每个任务都有一个专门的工作者线程。
     * 这里，各个 Pipe 的初始化完全可以在上游 Pipe 初始化完毕后再初始化其后继 Pipe，而不必多
     * 个 Pipe 同时初始化。
     * 因此，这个初始化的动作可以由一个线程来处理。该线程处理完各个 Pipe 的初始化后，可以继续
     * 处理之后可能产生的任务，如错误处理。
     * 所以，上述这些先后产生的任务可以由线程池中的一个工作者线程从头到尾负责执行。
     */
    final ExecutorService helperExecutor = Executors.newSingleThreadExecutor();

    final SimplePipeline<RecordSaveTask, String> pipeline = new
       SimplePipeline<RecordSaveTask, String>(helperExecutor);

    // 根据数据库记录生成相应的数据文件
    Pipe<RecordSaveTask, File> stageSaveFile = new AbstractPipe<RecordSaveTask,
        File>() {

        @Override
        protected File doProcess(RecordSaveTask task) throws PipeException {
            final RecordWriter recordWriter = RecordWriter.getInstance();
            final Record[] records = task.records;
            File file;
            if (null == records) {
                file = recordWriter.finishRecords(task.recordDay,
                   task.targetFileIndex);
            } else {
                try {
                    file = recordWriter.write(records, task.targetFileIndex);
                } catch (IOException e) {
                    throw new PipeException(this, task, "Failed to save
                       records.", e);
                }
            }
            return file;
        }
    };

    /*
     * 由于这里的几个 Pipe 都是处理 I/O 的，为了避免使用锁（以减少不必要的上下文切换） 但又能
     * 保证线程安全，故每个 Pipe 都采用单线程处理。
     * 若各个 Pipe 要改用线程池来处理，需要注意：1）线程安全 2）死锁。
     */
    pipeline.addAsWorkerThreadBasedPipe(stageSaveFile, 1);

    final String[][] ftpServerConfigs = retrieveFTPServConf();

    final ThreadPoolExecutor ftpExecutorService = new ThreadPoolExecutor(1,
       ftpServerConfigs.length, 60, TimeUnit.SECONDS,
       new ArrayBlockingQueue<Runnable>(100), new RejectedExecutionHandler() {
            @Override
           public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
                if (!executor.isShutdown()) {
                    try {
```

```
                    executor.getQueue().put(r);
                } catch (InterruptedException e) {
                    ;
                }
            }
        }

    });

// 将生成的数据文件传输到指定的主机上。
Pipe<File, File> stageTransferFile = new AbstractParallelPipe<File, File,
    File>(new SynchronousQueue<File>(), ftpExecutorService) {
    final Future<FTPClientUtil>[] ftpClientUtilHolders = new
        Future[ftpServerConfigs.length];

    @Override
    public void init(PipeContext pipeCtx) {
        super.init(pipeCtx);

        String[] ftpServerConfig;
        for (int i = 0; i < ftpServerConfigs.length; i++) {
            ftpServerConfig = ftpServerConfigs[i];
            ftpClientUtilHolders[i] = FTPClientUtil.newInstance(
                ftpServerConfig[0], ftpServerConfig[1], ftpServerConfig[2]);
        }

    }

    @Override
    protected List<Callable<File>> buildTasks(final File file) {
        List<Callable<File>> tasks = new LinkedList<Callable<File>>();

        for (Future<FTPClientUtil> ftpClientUtilHolder :
            ftpClientUtilHolders) {
            tasks.add(new ParallelTask(ftpClientUtilHolder, file));
        }

        return tasks;
    }

    @Override
    protected File combineResults(List<Future<File>> subTaskResults)
        throws Exception {
        if (0 == subTaskResults.size()) {
            return null;
        }
        File file = null;

        file = subTaskResults.get(0).get();

        return file;
    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        super.shutdown(timeout, unit);

        ftpExecutorService.shutdown();
```

```
            try {
                ftpExecutorService.awaitTermination(timeout, unit);
            } catch (InterruptedException e1) {
                ;
            }
            for (Future<FTPClientUtil> ftpClientUtilHolder :
                ftpClientUtilHolders) {
                try {
                    ftpClientUtilHolder.get().disconnect();
                } catch (Exception e) {
                    ;
                }
            }
        }

        class ParallelTask implements Callable<File> {
            public final Future<FTPClientUtil> ftpUtilHodler;
            public final File file2Transfer;

            public ParallelTask(Future<FTPClientUtil> ftpUtilHodler,
                File file2Transfer) {
                this.ftpUtilHodler = ftpUtilHodler;
                this.file2Transfer = file2Transfer;
            }

            @Override
            public File call() throws Exception {
                File transferedFile = null;
                ftpUtilHodler.get().upload(file2Transfer);
                transferedFile = file2Transfer;
                return transferedFile;
            }

        }

    };

    pipeline.addAsWorkerThreadBasedPipe(stageTransferFile, 1);

    // 备份已经传输的数据文件
    Pipe<File, Void> stageBackupFile = new AbstractPipe<File, Void>() {
        @Override
        protected Void doProcess(File transferedFile) throws PipeException {
            RecordWriter.backupFile(transferedFile);
            return null;
        }

        @Override
        public void shutdown(long timeout, TimeUnit unit) {

            // 所有文件备份完毕后，清理掉空文件夹
            RecordWriter.purgeDir();
        }

    };

    pipeline.addAsWorkerThreadBasedPipe(stageBackupFile,1);

    return pipeline;
```

```
    }

    private String[][] retrieveFTPServConf() {
        String[][] ftpServerConfigs = new String[][] {
            { "192.168.1.105", "datacenter", "abc123" }
            ,
            { "192.168.1.104", "datacenter", "abc123" }
//          ,
//          { "192.168.1.103", "datacenter", "abc123" }
            };
        return ftpServerConfigs;
    }

    private void processRecords(ResultSet rs,
        Pipeline<RecordSaveTask, String> pipeline) throws Exception {
        Record record;
        Record[] records = new Record[Config.RECORD_SAVE_CHUNK_SIZE];
        int targetFileIndex = 0;
        int nextTargetFileIndex = 0;
        int recordCountInTheDay = 0;
        int recordCountInTheFile = 0;
        String recordDay = null;
        String lastRecordDay = null;
        SimpleDateFormat sdf = new SimpleDateFormat("yyMMdd");

        while (rs.next()) {
            record = makeRecordFrom(rs);

            lastRecordDay = recordDay;

            recordDay = sdf.format(record.getOperationTime());

            if (recordDay.equals(lastRecordDay)) {
                records[recordCountInTheFile] = record;
                recordCountInTheDay++;
            } else {
                // 实际已发生的不同日期记录文件切换
                if (null != lastRecordDay) {
                    if (recordCountInTheFile >= 1) {
                        pipeline.process(new RecordSaveTask(Arrays.copyOf(records,
                            recordCountInTheFile), targetFileIndex));
                    } else {
                        pipeline
                            .process(new RecordSaveTask(lastRecordDay, targetFileIndex));
                    }

                    // 在此之前，先将 records 中的内容写入文件
                    records[0] = record;
                    recordCountInTheFile = 0;
                } else {
                    // 直接赋值
                    records[0] = record;
                }

                recordCountInTheDay = 1;
            }

            if (nextTargetFileIndex == targetFileIndex) {
```

```
                recordCountInTheFile++;
                if (0 == (recordCountInTheFile % Config.RECORD_SAVE_CHUNK_SIZE)) {
                    pipeline.process(new RecordSaveTask(Arrays.copyOf(records,
                        recordCountInTheFile), targetFileIndex));
                    recordCountInTheFile = 0;
                }
            }

            nextTargetFileIndex = (recordCountInTheDay) /
                Config.MAX_RECORDS_PER_FILE;
            if (nextTargetFileIndex > targetFileIndex) {
                // 预测到将发生同日期记录文件切换
                if (recordCountInTheFile > 1) {
                    pipeline.process(new RecordSaveTask(Arrays.copyOf(records,
                        recordCountInTheFile), targetFileIndex));
                } else {
                    pipeline.process(new RecordSaveTask(recordDay, targetFileIndex));
                }

                recordCountInTheFile = 0;

                targetFileIndex = nextTargetFileIndex;
            } else if (nextTargetFileIndex < targetFileIndex) {
                // 实际已发生的异日期记录文件切换，recordCountInTheFile 保持当前值
                targetFileIndex = nextTargetFileIndex;
            }

        }

        if (recordCountInTheFile > 0) {
            pipeline.process(new RecordSaveTask(Arrays.copyOf(records,
                recordCountInTheFile), targetFileIndex));
        }

    }

}
```

清单 13-1 中的数据同步定时任务 DataSyncTask 创建了一个 Pipeline 实例。该 Pipeline 实例包含了 3 个 Pipe 实例，它们分别对应 buildPipeline 方法中的 3 个局部变量：stageSaveFile、stageTransferFile 和 stageBackupFile。表 13-1 描述了这 3 个 Pipe 实例。

表 13-1. 数据同步定时任务中使用的 Pipe 实例

Pipe 实例名称	输　入	处　理	输　出	线 程 模 型
stageSaveFile	一组数据库记录（类型为 RecordSaveTask）	将输入的一组数据库记录存入数据文件	若当前数据文件已写满，则输出该写满的文件（类型为 File）；否则，输出 null	单工作者线程。若使用多线程模型，必然导致多个线程征用同一个数据文件（文件未写满），而这又需要引入锁。使用单线程模型则可以避免锁的使用

续表

Pipe 实例名称	输　入	处　理	输　出	线 程 模 型
stageTransferFile	已写满的数据文件（类型 File）	将输入的数据文件以并行方式 FTP 传输至指定的多台 FTP 主机	若文件传输成功，则输出已传输过的数据文件（类型为 File）；否则输出 null	单工作者线程。该 Pipe 处理过程会用到一款非线程安全的开源 FTP 客户端组件，使用单线程模型可以在不引入锁的情况下确保使用该组件时的线程安全
stageBackupFile	已传输成功的文件（类型 File）	将输入的文件移动到备份目录	null	单工作者线程

stageSaveFile 将其接收到的多组数据库记录中的数据写入数据文件。若当前数据文件已经写满，则该文件（如 fileX）会被作为 stageSaveFile 的输出提交给下一个 Pipe 实例 stageTransferFile 作为其输入。当 stageTransferFile 正将其一个输入元素文件（如 fileX）传输到多台 FTP 主机时，stageSaveFile 可能正在将其新接收到的其他组数据库记录写入新的数据文件。因此，这就形成了这两个逻辑上有先后依赖关系（数据库记录包含的数据必须先写到文件中，相应的数据才能进行 FTP 传输）的处理步骤，从整体上产生了并行计算。同理，当 stageTransferFile 将其输入中的一个文件（如 fileX）传输完毕后，该文件作为其输出被提交给下一个 Pipe 实例 stageBackupFile 作为输入。

如清单 13-1 所示代码引用的类 FTPClientUtil 和 RecordWriter 如清单 13-2、清单 13-3 所示，其他类参见本章第 5 节的代码（清单 13-5 至清单 13-13）。

从清单 13-2、清单 13-3 中的代码可以看出，这些代码都是按照单线程模型去编写的，这降低了编码难度，并减少了上下文切换。当然，这得益于我们在创建上述几个 Pipe 实例的时候采用了单工作者线程。

清单 13-2. FTPClientUtil 类源码

```
//模式角色：Promise.Promisor、Promise.Result
public class FTPClientUtil {
    private final FTPClient ftp = new FTPClient();

    private final Map<String,Boolean> dirCreateMap=new HashMap<String,Boolean>();
    /*
     * helperExecutor 是个静态变量，这使得 newInstance 方法在生成不同的 FTPClientUtil 实例时
     * 可以共用同一个线程池。
     * 模式角色：Promise.TaskExecutor
```

```
 */
private volatile static ExecutorService helperExecutor;

static {

    helperExecutor = new ThreadPoolExecutor(1,
            Runtime.getRuntime()
        .availableProcessors() * 2,
        60,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(10), new ThreadFactory() {
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setDaemon(true);
                return t;
            }
        }, new ThreadPoolExecutor.CallerRunsPolicy());
}

//私有构造器
private FTPClientUtil() {

}

//模式角色：Promise.Promisor.compute
public static Future<FTPClientUtil> newInstance(final String ftpServer,final
     String userName,final String password) {

    Callable<FTPClientUtil> callable = new Callable<FTPClientUtil>() {

        @Override
        public FTPClientUtil call() throws Exception {
            FTPClientUtil self = new FTPClientUtil();
            self.init( ftpServer, userName, password);
            return self;
        }

    };

    //task 相当于模式角色：Promise.Promise
    final FutureTask<FTPClientUtil> task = new
        FutureTask<FTPClientUtil>(callable);
    helperExecutor.execute(task);
    return task;
}

private void init(String ftpServer,String userName,String password) throws
    Exception {

    FTPClientConfig config = new FTPClientConfig();
    ftp.configure(config);

    int reply;
    ftp.connect(ftpServer);

    System.out.print(ftp.getReplyString());

    reply = ftp.getReplyCode();
```

```
        if (!FTPReply.isPositiveCompletion(reply)) {
            ftp.disconnect();
            throw new RuntimeException("FTP server refused connection.");
        }
        boolean isOK = ftp.login(userName, password);
        if (isOK) {
            System.out.println(ftp.getReplyString());

        } else {
            throw new RuntimeException("Failed to login." + ftp.getReplyString());
        }

        reply = ftp.cwd("~/subspsync");
        if (!FTPReply.isPositiveCompletion(reply)) {
            ftp.disconnect();
            throw new RuntimeException("Failed to change working
                directory.reply:"+reply);
        } else {

            System.out.println(ftp.getReplyString());
        }

        ftp.setFileType(FTP.ASCII_FILE_TYPE);

    }


    public void upload(File file) throws Exception {
        InputStream dataIn = new BufferedInputStream(new
            FileInputStream(file),1024*8);
        boolean isOK;
        String dirName = file.getParentFile().getName();
        String fileName=dirName + '/' + file.getName();
        ByteArrayInputStream checkFileInputStream = new
            ByteArrayInputStream("".getBytes());


        try{
            if(!dirCreateMap.containsKey(dirName)){
                ftp.makeDirectory(dirName);
                dirCreateMap.put(dirName, null);
            }

            try {
                isOK = ftp.storeFile(fileName, dataIn);
            } catch (IOException e) {
                throw new RuntimeException("Failed to upload " + file, e);
            }
            if (isOK) {
                ftp.storeFile(fileName + ".c", checkFileInputStream);

            } else {

                throw new RuntimeException("Failed to upload " + file + ",reply:" +
                    "," + ftp.getReplyString());
            }
        }finally{
```

```
            dataIn.close();
        }

    }


    public void disconnect() {
        if(ftp.isConnected()) {
     try {
       ftp.disconnect();
     } catch(IOException ioe) {
       //什么也不做

     }
    }
     }
}
```

清单 13-3. RecordWriter 类源码

```
public class RecordWriter {

    private static final RecordWriter INSTANCE = new RecordWriter();

    // HashMap 不是线程安全的，但 RecordWriter 实例是在单线程中使用的，因此不会产生问题。
    private static Map<String, PrintWriter> printWriterMap = new HashMap<String,
        PrintWriter>();

    private static String baseDir;
    private static final char FIELD_SEPARATOR = '|';

    // SimpleDateFormat 不是线程安全的，但 RecordWriter 实例是在单线程中使用的，因此不会产生问题。
    private static final SimpleDateFormat DIRECTORY_NAME_FORMATTER = new
        SimpleDateFormat("yyMMdd");
    private static final SimpleDateFormat sdf = new SimpleDateFormat(
       "yyyy-MM-dd HH:MM:ss");

    private static final DecimalFormat FILE_INDEX_FORMATTER = new DecimalFormat(
       "0000");

    private static final int RECORD_JOIN_SIZE = Config.RECORD_JOIN_SIZE;

    private static final FieldPosition FIELD_POS = new FieldPosition(
       DateFormat.Field.DAY_OF_MONTH);

    //私有构造器
    private RecordWriter() {
        baseDir = System.getProperty("user.home") + "/tmp/subspsync/";

    }
    public static RecordWriter getInstance() {
        return INSTANCE;
    }

    public File write(Record[] records, int targetFileIndex) throws IOException {
        if (null == records || 0 == records.length) {
            throw new IllegalArgumentException("records is null or empty");
        }
```

```
int recordCount = records.length;

String recordDay;

recordDay = DIRECTORY_NAME_FORMATTER.format(records[0].getOperationTime());

String fileKey = recordDay + '-' + targetFileIndex;

PrintWriter pwr = printWriterMap.get(fileKey);

if (null == pwr) {
    File file = new File(baseDir + '/' + recordDay + "/subspsync-gw-"
        + FILE_INDEX_FORMATTER.format(targetFileIndex) + ".dat");
    File dir = file.getParentFile();
    if (!dir.exists() && !dir.mkdirs()) {
        throw new IOException("No such directory:" + dir);
    }

    pwr = new PrintWriter(new BufferedWriter(new FileWriter(file, true),
        Config.WRITER_BUFFER_SIZE));

    printWriterMap.put(fileKey, pwr);
}
StringBuffer strBuf = new StringBuffer(40);
int i = 0;
for (Record record : records) {
    i++;

    pwr.print(String.valueOf(record.getId()));
    pwr.print(FIELD_SEPARATOR);
    pwr.print(record.getMsisdn());
    pwr.print(FIELD_SEPARATOR);
    pwr.print(record.getProductId());
    pwr.print(FIELD_SEPARATOR);

    pwr.print(record.getPackageId());
    pwr.print(FIELD_SEPARATOR);
    pwr.print(String.valueOf(record.getOperationType()));
    pwr.print(FIELD_SEPARATOR);

    strBuf.delete(0, 40);
    pwr.print(sdf.format(record.getOperationTime(), strBuf, FIELD_POS));
    pwr.print(FIELD_SEPARATOR);

    strBuf.delete(0, 40);

    pwr.print(sdf.format(record.getEffectiveDate(), strBuf, FIELD_POS));
    strBuf.delete(0, 40);
    pwr.print(FIELD_SEPARATOR);
    pwr.print(sdf.format(record.getDueDate(), strBuf, FIELD_POS));
    pwr.print('\n');

    if (0 == (i % RECORD_JOIN_SIZE)) {
        pwr.flush();
        i = 0;
        // Thread.yield();
    }
```

```
        }

        if (i > 0) {
            pwr.flush();
        }

        File file = null;

        // 处理当前文件中的最后一组记录
        if (recordCount < Config.RECORD_SAVE_CHUNK_SIZE) {
            pwr.close();
            file = new File(baseDir + '/' + recordDay + "/subspsync-gw-"
                + FILE_INDEX_FORMATTER.format(targetFileIndex) + ".dat");
            printWriterMap.remove(fileKey);
        }
        return file;
    }

    public File finishRecords(String recordDay, int targetFileIndex) {
        String fileKey = recordDay + '-' + targetFileIndex;
        PrintWriter pwr = printWriterMap.get(fileKey);
        File file = null;
        if (null != pwr) {
            pwr.flush();
            pwr.close();
            file = new File(baseDir + '/' + recordDay + "/subspsync-gw-"
                + FILE_INDEX_FORMATTER.format(targetFileIndex) + ".dat");
            printWriterMap.remove(fileKey);
        }
        return file;
    }

    public static void backupFile(final File file) {
        String recordDay = file.getParentFile().getName();
        File destFile = new File(baseDir + "/backup/" + recordDay);

        if (!destFile.exists()) {
            if (!destFile.mkdirs()) {
                return;
            }
        }

        destFile = new File(destFile, file.getName());

        if (!file.renameTo(destFile)) {
            throw new RuntimeException("Failed to backup file " + file);
        }

        file.delete();

    }
    public static void purgeDir() {
        File[] dirs = new File(baseDir).listFiles();
        for (File dir : dirs) {
            if (dir.isDirectory() && 0 == dir.list().length) {
                dir.delete();
            }
        }
    }
```

}

13.4 Pipeline 模式的评价与实现考量

Pipeline 模式可以对有依赖关系的任务实现并行处理。并行和并发编程中，为了提高并发性我们往往需要将规模较大的任务分解成若干个规模较小的子任务，这些子任务间通常没有依赖关系。而 Pipeline 模式则在允许子任务间存在依赖关系的条件下实现并行计算：Pipeline 模式中的每个 Pipe 的输入元素可以看成提交给该 Pipe 的一个任务。这样，前一个 Pipe 实例的任务处理结果就是下一个 Pipe 实例的输入任务。所以，从个体任务上看 Pipeline 模式对各个任务的处理是顺序的。由于每个 Pipe 实例都有其工作者线程负责任务处理，当一个 Pipe 实例处理其上游 Pipe 实例提交的某个任务时，其上游 Pipe 实例已经在处理其接收到的其他任务。因此，从整体任务上看，Pipeline 对任务的处理又是并行的。

Pipeline 模式为使用单线程模型编程提供了便利。多线程编程总的来说是复杂的，不仅代码编写比较复杂，出现问题时也不好定位。多线程程序出现非预期结果时，开发人员不仅要考虑算法是否正确，还要考虑是否是多线程相关问题（如线程安全）导致非预期的结果。相反，单线程编程就显得相对简单。Pipeline 模式非常便于我们采用单线程模型实现对子任务的处理。比如，子任务的处理涉及非线程安全对象或者涉及阻塞 I/O（Blocking I/O）操作时，我们不希望引入锁从而避免其增加上下文切换。此时单线程编程值得考虑，本章案例就是这样一个例子。Pipeline 模式中，要对某种子任务采用单线程处理，开发人员只需要调用 SimplePipeline 的 addAsWorkerThreadBasedPipe 方法将负责该子任务处理的 Pipe 实例加入 Pipeline 即可，如清单 13-1 所示。

Pipeline 模式中，每个 Pipeline 实例都是一个 Pipe 实例。因此，我们不仅可以往一个 Pipeline 实例中添加 Pipe 实例，也可以添加其他 Pipeline 实例。这就允许应用程序复用特定的 Pipe 实例组合（Pipeline 实例），或者改变一个 Pipeline 实例内部各个 Pipe 实例的编排而外界不受此影响。因此，Pipeline 模式带来了一定的灵活性。

当然，Pipeline 模式提供的优势的背后也隐藏着代价。Pipeline 模式中各个处理阶段所使用的工作者线程或者线程池、表示各个处理阶段的输入/输出对象的创建和移动（进出队列）都有其自身的时间和空间开销。因此，Pipeline 模式适合于处理规模较大的任务，否则可能得不偿失：Pipeline 模式可能带来的好处无法抵消其背后的代价。

13.4.1 Pipeline 的深度

下面以线性 Pipeline 为例讨论 Pipeline 的深度问题，非线性 Pipeline 同样可参考之。一个 Pipeline 实例中包含的 Pipe 实例的个数被称为 Pipeline 的深度。通常，我们需要考虑 Pipeline 的深度与 JVM 宿主机的 CPU 个数（以下称为 N_{CPU}）间的关系。如果当前 Pipeline 实例所处理的任务多属 CPU 密集型，那么 Pipeline 的深度最好不超过 N_{CPU}。如果 Pipeline 实例所处理的任务多属 I/O 密集型，那么 Pipeline 的深度最好不超过 $2\times N_{CPU}$。

按照上述规则，如果 Pipeline 的深度太大，或者 Pipeline 的深度并未“超标”但 Pipeline 中有不少 Pipe 实例需要使用多个工作者线程，那么此时我们可以考虑使用线程池来实现各个 Pipe 实例的任务处理。实现这点，我们只需要调用 SimplePipeline 实例的 addAsThreadPoolBasedPipe 方法添加相关 Pipe 实例即可。另外，使用基于线程池的 Pipe 实例需要注意线程池可能导致的一些问题，下文会提到这点。

13.4.2 基于线程池的 Pipe

当 Pipeline 的深度比较大，或者其深度不大但不少 Pipe 实例需要多个工作者线程负责任务处理时，为了避免创建过多的线程而增加资源的消耗以及上下文切换，我们可能需要考虑使用基于线程池的 Pipe，即一个 Pipeline 实例中的多个 Pipe 实例共用一个（或者多个）线程池负责各个 Pipe 的任务处理。清单 13-4 展示了一个基于线程池的 Pipe 例子。使用同一个线程池负责执行提交给不同 Pipe 实例的任务，其本质是一个 Pipeline 实例中的多个 Pipe 实例作为相应线程池的客户端向线程池提交任务。因此，此情形下，无论是实现 Pipeline 模式还是使用 Pipeline 模式都要注意各个 Pipe 实例给线程池提交的任务之间是否存在依赖关系。这是因为同一个线程池实例执行的各个任务若存在依赖关系，则可能导致线程池死锁。有关线程池死锁的进一步信息，请参考第 9 章。

清单 13-4. 使用基于线程池的 Pipe 示例代码

```
public class ThreadPoolBasedPipeExample {
    public static void main(String[] args) {
        final ThreadPoolExecutor executorSerivce;
        executorSerivce = new ThreadPoolExecutor(1, Runtime.getRuntime()
            .availableProcessors() * 2, 60, TimeUnit.MINUTES,
            new SynchronousQueue<Runnable>(),
            new ThreadPoolExecutor.CallerRunsPolicy());

        final SimplePipeline<String, String> pipeline = new SimplePipeline<String,
            String>();
        Pipe<String, String> pipe = new AbstractPipe<String, String>() {
            @Override
            protected String doProcess(String input) throws PipeException {
                String result = input + "->[pipe1," +
                    Thread.currentThread().getName() + "]";
```

```
            System.out.println(result);
            return result;
        }
    };

    pipeline.addAsThreadPoolBasedPipe(pipe, executorSerivce);

    pipe = new AbstractPipe<String, String>() {

        @Override
        protected String doProcess(String input) throws PipeException {
            String result = input + "->[pipe2," +
                Thread.currentThread().getName() + "]";
            System.out.println(result);
            try {
                Thread.sleep(new Random().nextInt(100));
            } catch (InterruptedException e) {
                ;
            }
            return result;
        }
    };

    pipeline.addAsThreadPoolBasedPipe(pipe, executorSerivce);

    pipe = new AbstractPipe<String, String>() {

        @Override
        protected String doProcess(String input) throws PipeException {
            String result = input + "->[pipe3," +
                Thread.currentThread().getName() + "]";
            ;
            System.out.println(result);
            try {
                Thread.sleep(new Random().nextInt(200));
            } catch (InterruptedException e) {
                ;
            }
            return result;
        }

        @Override
        public void shutdown(long timeout, TimeUnit unit) {

            // 在最后一个 Pipe 中关闭线程池
            executorSerivce.shutdown();
            try {
                executorSerivce.awaitTermination(timeout, unit);
            } catch (InterruptedException e) {
                ;
            }
        }
    };

    pipeline.addAsThreadPoolBasedPipe(pipe, executorSerivce);

    pipeline.init(pipeline.newDefaultPipeContext());

    int N = 100;
```

```
        try {
            for (int i = 0; i < N; i++) {
                pipeline.process("Task-" + i);
            }
        } catch (IllegalStateException e) {
            ;
        } catch (InterruptedException e) {
            ;
        }

        pipeline.shutdown(10, TimeUnit.SECONDS);

    }

}
```

如清单 13-4 所示代码运行后输出类似如下的消息（省略部分输出）。

```
Task-0->[pipe1,pool-1-thread-1]
Task-1->[pipe1,pool-1-thread-2]
Task-4->[pipe1,pool-1-thread-5]
Task-3->[pipe1,pool-1-thread-4]
Task-1->[pipe1,pool-1-thread-2]->[pipe2,pool-1-thread-8]
Task-2->[pipe1,pool-1-thread-3]
Task-6->[pipe1,main]
Task-8->[pipe1,main]
Task-8->[pipe1,main]->[pipe2,main]
Task-4->[pipe1,pool-1-thread-5]->[pipe2,pool-1-thread-5]
Task-0->[pipe1,pool-1-thread-1]->[pipe2,pool-1-thread-7]
……此处省略部分输出
Task-94->[pipe1,pool-1-thread-8]->[pipe2,pool-1-thread-8]
Task-93->[pipe1,pool-1-thread-1]->[pipe2,pool-1-thread-1]
Task-97->[pipe1,main]->[pipe2,main]->[pipe3,pool-1-thread-3]
Task-98->[pipe1,main]
Task-98->[pipe1,main]->[pipe2,main]
Task-96->[pipe1,pool-1-thread-5]->[pipe2,pool-1-thread-5]->[pipe3,pool-1-threa
d-5]
Task-93->[pipe1,pool-1-thread-1]->[pipe2,pool-1-thread-1]->[pipe3,pool-1-threa
d-1]
Task-94->[pipe1,pool-1-thread-8]->[pipe2,pool-1-thread-8]->[pipe3,pool-1-threa
d-8]
Task-95->[pipe1,pool-1-thread-7]->[pipe2,pool-1-thread-7]->[pipe3,pool-1-threa
d-3]
Task-98->[pipe1,main]->[pipe2,main]->[pipe3,pool-1-thread-7]
Task-99->[pipe1,pool-1-thread-4]
Task-99->[pipe1,pool-1-thread-4]->[pipe2,pool-1-thread-4]
Task-99->[pipe1,pool-1-thread-4]->[pipe2,pool-1-thread-4]->[pipe3,pool-1-threa
d-8]
```

13.4.3 错误处理

Pipe 实例对其任务进行处理过程中抛出的异常可能需要在相应 Pipe 实例之外进行处理。此时，异常处理通常有两种方式。一种是本章案例所使用的方法，即各个 Pipe 实例捕获到异常后调用 PipeContext 实例的 handleError 进行错误处理。另一种方法是创建一个专门负责错误处理的 Pipe 实例，其他 Pipe 实例捕获异常后提交相关数据给该 Pipe 实例处理。

13.4.4 可配置的 Pipeline

本章案例中，我们以代码的方式将若干个 Pipe 实例添加到 Pipeline 实例中。如果有必要的话，我们也可以借助配置文件等可配置的方式实现以动态的方式创建 Pipeline 实例，并往其中添加所需的 Pipe 实例。

13.5 Pipeline 模式的可复用实现代码

下面的代码展示了 Pipeline 模式的一个可复用实现，如清单 13-5 至清单 13-13 所示。

清单 13-5. Pipe 接口源码

```
/**
 * 对处理阶段的抽象。
 * 负责对输入进行处理，并将输出作为下一个处理阶段的输入。
 *
 * @author Viscent Huang
 *
 * @param <IN>
 *          输入类型
 * @param <OUT>
 *          输出类型
 */
public interface Pipe<IN, OUT> {
    /**
     * 设置当前 Pipe 实例的下一个 Pipe 实例。
     *
     * @param nextPipe
     *          下一个 Pipe 实例
     */
    void setNextPipe(Pipe<?, ?> nextPipe);

    /**
     * 初始化当前 Pipe 实例对外提供的服务。
     *
     * @param pipeCtx
     */
    void init(PipeContext pipeCtx);

    /**
```

```
     * 停止当前 Pipe 实例对外提供的服务。
     *
     * @param timeout
     * @param unit
     */
    void shutdown(long timeout, TimeUnit unit);

    /**
     * 对输入元素进行处理，并将处理结果作为下一个 Pipe 实例的输入。
     */
    void process(IN input) throws InterruptedException;
}
```

清单 13-6. AbstractPipe 类源码

```
/**
 * Pipe 的抽象实现类。
 * 该类会调用其子类实现的 doProcess 方法对输入元素进行处理，并将相应的输出作为
 * 下一个 Pipe 实例的输入。
 * @author Viscent Huang
 *
 * @param <IN>
 *        输入类型
 * @param <OUT>
 *        输出类型
 */
public abstract class AbstractPipe<IN, OUT> implements Pipe<IN, OUT> {
    protected volatile Pipe<?, ?> nextPipe = null;
    protected volatile PipeContext pipeCtx;

    @Override
    public void init(PipeContext pipeCtx) {
        this.pipeCtx = pipeCtx;

    }

    @Override
    public void setNextPipe(Pipe<?, ?> nextPipe) {
        this.nextPipe = nextPipe;

    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        // 什么也不做
    }

    /**
     * 留给子类实现。用于子类实现其任务处理逻辑。
     *
     * @param input
     *        输入元素（任务）
     * @return 任务的处理结果
     * @throws PipeException
     */
    protected abstract OUT doProcess(IN input) throws PipeException;
```

```
    @SuppressWarnings("unchecked")
    public void process(IN input) throws InterruptedException {

        try {

            OUT out = doProcess(input);
            if (null != nextPipe) {
                if (null != out) {
                    ((Pipe<OUT, ?>) nextPipe).process(out);
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (PipeException e) {
            pipeCtx.handleError(e);
        }
    }
}
```

清单 13-7. WorkerThreadPipeDecorator 类源码

```
/**
 * 基于工作者线程的 Pipe 实现类。
 * 提交到该 Pipe 的任务由指定个数的工作者线程共同处理。
 * 该类使用了 Two-phase Termination 模式（参见第 5 章）。
 *
 * @author Viscent Huang
 *
 * @param <IN>
 *          输入类型
 * @param <OUT>
 *          输出类型
 */
public class WorkerThreadPipeDecorator<IN, OUT> implements Pipe<IN, OUT> {
    protected final BlockingQueue<IN> workQueue;
    private final Set<AbstractTerminatableThread> workerThreads = new
        HashSet<AbstractTerminatableThread>();
    private final TerminationToken terminationToken = new TerminationToken();

    private final Pipe<IN, OUT> delegate;

    public WorkerThreadPipeDecorator(Pipe<IN, OUT> delegate, int workerCount) {
        this(new SynchronousQueue<IN>(), delegate, workerCount);
    }

    public WorkerThreadPipeDecorator(BlockingQueue<IN> workQueue,
        Pipe<IN, OUT> delegate, int workerCount) {
        if (workerCount <= 0) {
            throw new IllegalArgumentException("workerCount should be positive!");
        }

        this.workQueue = workQueue;
        this.delegate = delegate;
        for (int i = 0; i < workerCount; i++) {
            workerThreads.add(new AbstractTerminatableThread(terminationToken) {
                @Override
                protected void doRun() throws Exception {
                    try {
```

```
                        dispatch();
                    } finally {
                        terminationToken.reservations.decrementAndGet();
                    }

                }
            });
        }

    }

    protected void dispatch() throws InterruptedException {
        IN input = workQueue.take();
        delegate.process(input);
    }

    @Override
    public void init(PipeContext pipeCtx) {
        delegate.init(pipeCtx);
        for (AbstractTerminatableThread thread : workerThreads) {
            thread.start();
        }
    }

    @Override
    public void process(IN input) throws InterruptedException {
        workQueue.put(input);
        terminationToken.reservations.incrementAndGet();
    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        for (AbstractTerminatableThread thread : workerThreads) {
            thread.terminate();
            try {
                thread.join(TimeUnit.MILLISECONDS.convert(timeout, unit));
            } catch (InterruptedException e) {
            }
        }
        delegate.shutdown(timeout, unit);
    }

    @Override
    public void setNextPipe(Pipe<?, ?> nextPipe) {
        delegate.setNextPipe(nextPipe);
    }

}
```

清单 13-8. ThreadPoolPipeDecorator 类源码

```
/**
 * 基于线程池的 Pipe 实现类。
 *
 * @author Viscent Huang
 *
 * @param <IN>
 *          输入类型
 * @param <OUT>
```

```
 *          输出类型
 */
public class ThreadPoolPipeDecorator<IN, OUT> implements Pipe<IN, OUT> {
    private final Pipe<IN, OUT> delegate;
    private final ExecutorService executorSerivce;

    //线程池停止标志。
    private final TerminationToken terminationToken;
    private final CountDownLatch stageProcessDoneLatch = new CountDownLatch(1);

    public ThreadPoolPipeDecorator(Pipe<IN, OUT> delegate,
        ExecutorService executorSerivce) {
        this.delegate = delegate;
        this.executorSerivce = executorSerivce;
        this.terminationToken = TerminationToken.newInstance(executorSerivce);
    }

    @Override
    public void init(PipeContext pipeCtx) {
        delegate.init(pipeCtx);

    }

    @Override
    public void process(final IN input) throws InterruptedException {

        Runnable task = new Runnable() {
            @Override
            public void run() {
                int remainingReservations = -1;
                try {
                    delegate.process(input);
                } catch (InterruptedException e) {
                    ;
                } finally {
                    remainingReservations = terminationToken.reservations
                        .decrementAndGet();
                }

                if (terminationToken.isToShutdown() && 0 == remainingReservations) {
                    stageProcessDoneLatch.countDown();
                }
            }

        };

        executorSerivce.submit(task);

        terminationToken.reservations.incrementAndGet();

    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        terminationToken.setIsToShutdown();

        if (terminationToken.reservations.get() > 0) {
            try {
                if (stageProcessDoneLatch.getCount() > 0) {
```

```
                    stageProcessDoneLatch.await(timeout, unit);
                }
            } catch (InterruptedException e) {
                ;
            }
        }

        delegate.shutdown(timeout, unit);
    }

    @Override
    public void setNextPipe(Pipe<?, ?> nextPipe) {
        delegate.setNextPipe(nextPipe);
    }

    /**
     * 线程池停止标志。
     * 每个 ExecutorService 实例对应唯一的一个 TerminationToken 实例。
     * 这里使用了 Two-phase Termination 模式（第 5 章）的思想来停止多个 Pipe 实例所共用的
     * 线程池实例。
     * @author Viscent Huang
     *
     */
    private static class TerminationToken extends
        io.github.viscent.mtpattern.tpt.TerminationToken {
        private final static ConcurrentMap<ExecutorService, TerminationToken>
            INSTANCES_MAP = new ConcurrentHashMap<ExecutorService, TerminationToken>();

        // 私有构造器
        private TerminationToken() {

        }

        void setIsToShutdown() {
            this.toShutdown = true;
        }

        static TerminationToken newInstance(ExecutorService executorSerivce) {
            TerminationToken token = INSTANCES_MAP.get(executorSerivce);
            if (null == token) {
                token = new TerminationToken();
                TerminationToken existingToken = INSTANCES_MAP.putIfAbsent(
                    executorSerivce, token);
                if (null != existingToken) {
                    token = existingToken;
                }
            }
            return token;
        }
    }

}
```

清单 13-9. AbstractParallelPipe 类源码

```
/**
 * 支持并行处理的 Pipe 实现类。
 * 该类对每个输入元素生成一组子任务，并以并行的方式去执行这些子任务。
```

```
 * 各个子任务的执行结果会被合并为相应输入元素的输出结果。
 *
 * @author Viscent Huang
 *
 * @param <IN>
 *        输入类型
 * @param <OUT>
 *        输出类型
 *
 * @param <V>
 *        并行子任务的处理结果类型
 */
public abstract class AbstractParallelPipe<IN, OUT, V> extends
    AbstractPipe<IN, OUT> {
     private final ExecutorService executorService;

     public AbstractParallelPipe(BlockingQueue<IN> queue,
         ExecutorService executorService) {
          super();
          this.executorService = executorService;
     }

     /**
      * 留给子类实现。用于根据指定的输入元素 input 构造一组子任务。
      *
      * @param input
      *        输入元素
      * @param <V>
      *        子任务的处理结果类型
      * @return 一组子任务
      */
     protected abstract List<Callable<V>> buildTasks(IN input) throws Exception;

     /**
      * 留给子类实现。对各个子任务的处理结果进行合并，形成相应输入元素的输出结果。
      *
      * @param subTaskResults
      *        子任务处理结果列表
      * @return 相应输入元素的处理结果
      * @throws Exception
      */
     protected abstract OUT combineResults(List<Future<V>> subTaskResults)
         throws Exception;

     /**
      * 以并行的方式执行一组子任务。
      *
      * @param tasks
      *        一组子任务
      * @return 一组可以借以获取并行任务中各个任务处理结果的 Promise（参见第 6 章，Promise 模式）
      * 实例。
      */
     protected List<Future<V>> invokeParallel(List<Callable<V>> tasks)
         throws Exception {
          return executorService.invokeAll(tasks);
     }
```

```
    @Override
    protected OUT doProcess(final IN input) throws PipeException {
        OUT out = null;
        try {
            out = combineResults(invokeParallel(buildTasks(input)));
        } catch (Exception e) {
            throw new PipeException(this, input, "Task failed", e);
        }
        return out;
    }

}
```

清单 13-10. Pipeline 接口源码

```
/**
 * 对复合 Pipe 的抽象。 一个 Pipeline 实例可包含多个 Pipe 实例。
 *
 * @author Viscent Huang
 *
 * @param <IN>
 *          输入类型
 * @param <OUT>
 *          输出类型
 */
public interface Pipeline<IN, OUT> extends Pipe<IN, OUT> {

    /**
     * 往该 Pipeline 实例中添加一个 Pipe 实例。
     *
     * @param pipe
     *          Pipe 实例
     */
    void addPipe(Pipe<?, ?> pipe);
}
```

清单 13-11. SimplePipeline 类源码

```
public class SimplePipeline<T, OUT> extends AbstractPipe<T, OUT> implements
    Pipeline<T, OUT> {

    private final Queue<Pipe<?, ?>> pipes = new LinkedList<Pipe<?, ?>>();

    private final ExecutorService helperExecutor;

    public SimplePipeline() {
        this(Executors.newSingleThreadExecutor(new ThreadFactory() {

            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r, "SimplePipeline-Helper");
                t.setDaemon(true);
                return t;
            }

        }));

    }
```

```
public SimplePipeline(final ExecutorService helperExecutor) {
    super();
    this.helperExecutor = helperExecutor;
}

@Override
public void shutdown(long timeout, TimeUnit unit) {
    Pipe<?, ?> pipe;

    while (null != (pipe = pipes.poll())) {
        pipe.shutdown(timeout, unit);
    }

    helperExecutor.shutdown();

}

@Override
protected OUT doProcess(T input) throws PipeException {
    // 什么也不做
    return null;
}

@Override
public void addPipe(Pipe<?, ?> pipe) {

    // Pipe 间的关联关系在 init 方法中建立
    pipes.add(pipe);
}

public <INPUT, OUTPUT> void addAsWorkerThreadBasedPipe(
   Pipe<INPUT, OUTPUT> delegate, int workerCount) {
    addPipe(new WorkerThreadPipeDecorator<INPUT, OUTPUT>(delegate,
        workerCount));
}

public <INPUT, OUTPUT> void addAsThreadPoolBasedPipe(
   Pipe<INPUT, OUTPUT> delegate, ExecutorService executorSerivce) {
    addPipe(new ThreadPoolPipeDecorator<INPUT, OUTPUT>(delegate,
       executorSerivce));
}

@Override
public void process(T input) throws InterruptedException {

    @SuppressWarnings("unchecked")
    Pipe<T, ?> firstPipe = (Pipe<T, ?>) pipes.peek();

    firstPipe.process(input);
}

@Override
public void init(final PipeContext ctx) {

    LinkedList<Pipe<?, ?>> pipesList = (LinkedList<Pipe<?, ?>>) pipes;
    Pipe<?, ?> prevPipe = this;
    for (Pipe<?, ?> pipe : pipesList) {
        prevPipe.setNextPipe(pipe);
        prevPipe = pipe;
```

```
        }

        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (Pipe<?, ?> pipe : pipes) {
                    pipe.init(ctx);
                }

            }

        };

        helperExecutor.submit(task);
    }

    public PipeContext newDefaultPipeContext() {
        return new PipeContext() {
            @Override
            public void handleError(final PipeException exp) {
                helperExecutor.submit(new Runnable() {

                    @Override
                    public void run() {
                        exp.printStackTrace();
                    }

                });
            }

        };
    }
}
```

清单 13-12. PipeContext 接口源码

```
/**
 * 对各个处理阶段的计算环境进行抽象，主要用于异常处理。
 *
 * @author Viscent Huang
 *
 */
public interface PipeContext {
    /**
     * 用于对处理阶段抛出的异常进行处理.
     *                              \
     * @param exp
     */
    void handleError(PipeException exp);
}                                    .
```

清单 13-13. PipeException 类源码

```
public class PipeException extends Exception {
    private static final long serialVersionUID = -2944728968269016114L;
    /**
     * 抛出异常的 Pipe 实例。
     */
    public final Pipe<?, ?> sourcePipe;
```

```
    /**
     * 抛出异常的 Pipe 实例在抛出异常时所处理的输入元素。
     */
    public final Object input;

    public PipeException(Pipe<?, ?> sourcePipe, Object input, String message) {
        super(message);
        this.sourcePipe = sourcePipe;
        this.input = input;
    }

    public PipeException(Pipe<?, ?> sourcePipe, Object input, String message,
        Throwable cause) {
        super(message, cause);
        this.sourcePipe = sourcePipe;
        this.input = input;
    }
}
```

利用本章的可复用代码实现 Pipeline 模式，应用程序只需要完成以下几件事情。

1.【必需】创建 Pipeline 实例。

2.【必需】创建表示各个处理阶段的 Pipe 实例，并将其添加到 Pipeline 实例中。

3.【必需】初始化 Pipeline 实例。

13.6 Java 标准库实例

Java 标准库中没有使用 Pipeline 模式的实例。

13.7 相关模式

13.7.1 Serial Thread Confinement 模式（第 11 章）

Pipeline 模式中，如果某个 Pipe 实例使用的线程模型是单个工作者线程，那么该 Pipe 实例就可以看作是一个 Serial Thread Confinement 模式的实例。此时，Pipe 实例同时实现了 Serial Thread Confinement 模式中的所有参与者。

13.7.2 Master-Slave 模式（第 12 章）

Pipeline 模式中的 AbstractParallelPipe 参与者可以将其接收到的一个输入元素（原始任务）分解为若干个子任务，并以并行的方式去执行这些子任务。各个子任务处理结果经过合并形成

原始任务的输出。因此，AbstractParallelPipe 参与者可以看成是 Master-Slave 模式的一个实现。

13.7.3 Composite 模式[1]

Pipeline 模式中，每个 Pipeline 实例都是一个 Pipe 实例，因此我们可以往 Pipeline 实例中添加 Pipe 实例和其他 Pipeline 实例。因此，Pipeline 参与者可以看成是 Composite 模式的一个实例。

13.8 参考资源

1. THE PIPELINE PATTERN. http://www.informit.com/articles/article.aspx? p=366887& seqNum=8.

2. Apache Commons Pipeline 开源项目. http://commons.apache.org/ sandbox/commons- pipeline/.

3. Erich Gamma 等. 设计模式：可复用面向对象软件的基础（英文版）. 机械工业出版社，2002.

1 GOF 设计模式，不在本书讨论范围，详情请参阅本章相应的参考资源。

第 14 章

Half-sync/Half-async（半同步/半异步）模式

14.1 Half-sync/Half-async 模式简介

Half-sync/Half-async 模式集成了同步编程和异步编程的优势，它通过同步任务和异步任务的共同协作来完成一个计算，既保持了同步编程的简单性，又充分发挥异步编程在提高系统并发性方面的优势。该模式就如一个称职的管理者，知晓不同下属各自的优势和弱点，在指派任务的时候能够根据下属的优势和弱点扬长避短，并使各个下属相互协作，共同完成工作。

14.2 Half-sync/Half-async 模式的架构

Half-sync/Half-async 模式是一个分层架构。它包含 3 个层次：异步任务层、同步任务层和队列层。Half-sync/Half-async 模式的核心思想是如何将系统中的任务进行恰当的分解，使各个子任务落入合适的层次中。

低级的任务（如与用户界面有关的任务）或者耗时较短的任务可以安排在异步任务层。而高级的任务（如数据库访问）或者耗时较长的任务可以安排在同步任务层。而异步任务层和同步任务层这两层之间的协作通过队列层进行解耦（Decoupling）：队列层负责异步任务层和同步任务层之间的数据交换。

Half-sync/Half-async 模式的主要参与者有以下几种。其类图如图 14-1 所示。

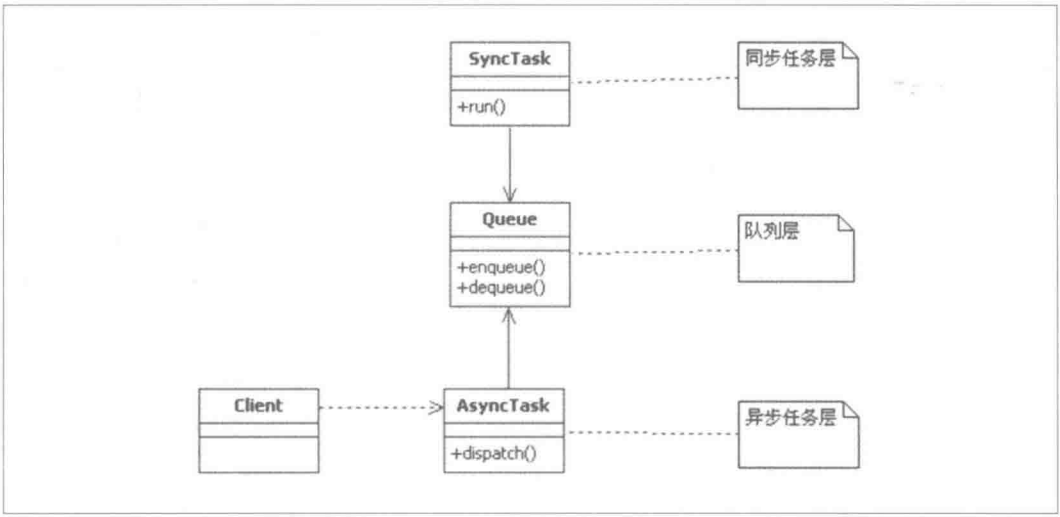


图 14-1. Half-sync/Half-async 模式的类图

- **AsyncTask**：异步任务，负责接收来自客户端的输入，对其进行初步处理，并通过队列与相应的同步任务通信。其主要方法及职责如下。
  - **dispatch**：对输入进行初步处理，并构造相应消息放入队列由相应的同步任务进行处理。
- **Queue**：队列，异步任务层和同步任务层进行通信的中介。其主要方法及职责如下。
  - **enqueue**：消息入队列。
  - **dequeue**：消息出队列。
- **SyncTask**：同步任务，负责处理队列中的消息所对应的计算。其主要方法及职责如下。
  - **run**：执行同步任务。

Half-sync/Half-async 模式的序列图如图 14-2 所示。

第 1 步：客户端代码调用 AsyncTask 的 dispatch 方法。

第 2、3 步：dispatch 方法对其输入进行初步处理后，构造相应消息放入队列，并返回。

第 4 步：dispatch 方法返回。

第 5 步：同步任务的 run 方法被 JVM 或者线程池调用。

第 6、7 步：同步任务的 run 方法取队列中的消息，并根据消息执行相应操作。

上述步骤中，第 1～4 步是运行在客户端线程中的，第 5～7 步是运行在后台线程（工作者线程）中的。

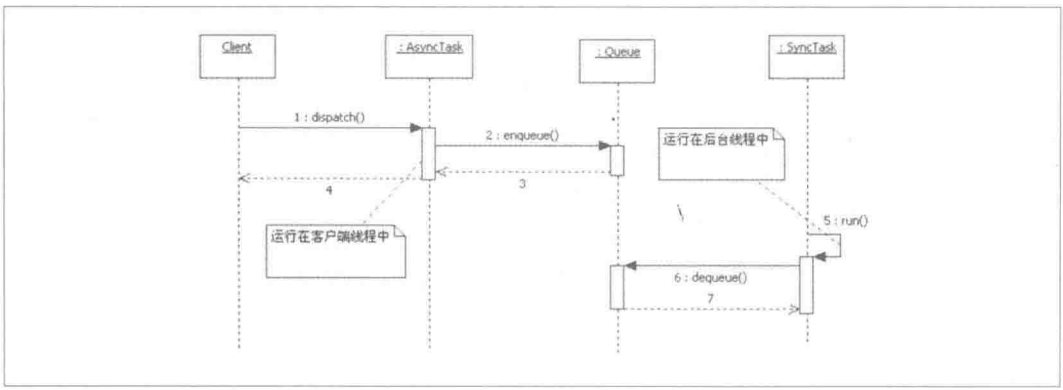


图 14-2. Half-sync/Half-async 模式的序列图

14.3 Half-sync/Half-async 模式实战案例解析

某系统的告警功能被封装在一个模块中。告警模块的主要功能是将其接收到的告警信息发送到告警服务器上。该模块的入口类是 AlarmMgr。其他模块（业务模块）需要发送告警信息时只需调用 AlarmMgr 的 sendAlarm 方法即可。告警模块中的类 AlarmAgent 负责与告警服务器对接，它通过网络连接将告警信息上报到告警服务器。从业务模块的角度来看，告警模块需要解决如下两个问题。

一、告警信息最终是要通过网络连接发送到告警服务器上的。而网络 I/O 是个相对慢的操作，我们不希望业务模块在调用 AlarmMgr.sendAlarm 时受此影响。也就是说，业务模块对 AlarmMgr.sendAlarm 的调用应该尽可能快地返回，这样不影响业务模块的处理能力。

二、当某个告警产生的条件（通常是故障）持续存在时，相应的告警信息会反复地被发送到告警模块。业务模块需要在某条告警信息被重复发送到告警模块达到指定次数后，不再在其日志中打印详细的告警信息，而只是打印一个概要信息。因此，AlarmMgr 的 sendAlarm 方法需要返回要发送的告警信息当前被重复发送的次数。

上述问题涉及两个操作，前者是个相对慢的操作（网络 I/O），后者是个相对快的操作（计数）。因此，这里我们可以应用 Half-sync/Half-async 模式：AlarmMgr 可以看成 AsyncTask 参与者实例。其 sendAlarm 方法相当于 dispatch 方法。该方法先对告警信息进行初步处理——计算

该告警信息被重复发送到告警模块的次数，再将告警信息放入队列，由同步层的任务（告警发送线程）将告警信息上报至告警服务器。这样，sendAlarm 方法是在业务模块线程中执行的，其消耗的时间主要就在计算告警信息被重复发送的次数以及告警信息入队列。而真正将告警信息通过网络连接上报到服务器的操作是由同步层中的告警发送线程（工作者线程）执行的。因此，这就产生了业务线程和告警发送线程并发执行各自的操作，后者执行的快慢不会影响前者的执行。故上述第一个问题也得以解决。

清单 14-1 展示了一个示例业务模块对告警模块的调用。

清单 14-1. 示例业务模块对告警模块的调用

```
public class SampleAlarmClient {
    private final static Logger logger = Logger
       .getLogger(SampleAlarmClient.class);

    // 告警日志抑制阈值
    private static final int ALARM_MSG_SUPRESS_THRESHOLD = 10;

    static {

        // 初始化告警模块
        AlarmMgr.getInstance().init();
    }

    public static void main(String[] args) {
        SampleAlarmClient alarmClient = new SampleAlarmClient();
        Connection dbConn = null;
        try {
            dbConn = alarmClient.retrieveDBConnection();

        } catch (Exception e) {
            final AlarmMgr alarmMgr = AlarmMgr.getInstance();

            // 告警被重复发送至告警模块的次数
            int duplicateSubmissionCount;
            String alarmId = "00000010000020";
            final String alarmExtraInfo = "operation=GetDBConnection;detail=Failed
                to get DB connection:" + e.getMessage();

            duplicateSubmissionCount = alarmMgr.sendAlarm(AlarmType.FAULT, alarmId,
                alarmExtraInfo);
            if (duplicateSubmissionCount < ALARM_MSG_SUPRESS_THRESHOLD) {
                logger.error("Alarm[" + alarmId + "] raised,extraInfo:"
                    + alarmExtraInfo);
            } else {
                if (duplicateSubmissionCount == ALARM_MSG_SUPRESS_THRESHOLD) {
                    logger.error("Alarm[" + alarmId + "] was raised more than "
                        + ALARM_MSG_SUPRESS_THRESHOLD
                        + " times, it will no longer be logged.");
                }
            }
            return;
        }
```

```
        alarmClient.doSomething(dbConn);

    }

    // 获取数据库连接
    private Connection retrieveDBConnection() throws Exception {
        Connection cnn = null;

        // 省略其他代码

        return cnn;
    }

    private void doSomething(Connection conn) {
        // 省略其他代码
    }

}
```

如清单 14-1 所示，业务模块在侦测到故障后，会调用 AlarmMgr 的 sendAlarm 方法发送告警信息。AlarmMgr 类的源码如清单 14-2 所示。

清单 14-2. AlarmMgr 类源码

```
/**
 * 告警功能入口类
 * 模式角色：HalfSync/HalfAsync.AsyncTask
 * 模式角色：Two-phaseTermination.ThreadOwner
 */
public class AlarmMgr {
    // 保存 AlarmMgr 类的唯一实例
    private static final AlarmMgr INSTANCE = new AlarmMgr();

    private volatile boolean shutdownRequested = false;

    //告警发送线程
    private final AlarmSendingThread alarmSendingThread;

    private AlarmMgr() {
        alarmSendingThread = new AlarmSendingThread();

    }

    public static AlarmMgr getInstance() {
        return INSTANCE;
    }

    /**
     * 发送告警
     * @param type 告警类型
     * @param id 告警编号
     * @param extraInfo 告警参数
     * @return 由 type+id+extraInfo 唯一确定的告警信息被提交的次数。-1 表示告警管理器已被关闭。
     */
    public int sendAlarm(AlarmType type, String id, String extraInfo) {
        Debug.info("Trigger alarm " + type + "," + id + ',' + extraInfo);
```

```
        int duplicateSubmissionCount = 0;
        try {
            AlarmInfo alarmInfo = new AlarmInfo(id, type);
            alarmInfo.setExtraInfo(extraInfo);
            duplicateSubmissionCount = alarmSendingThread.sendAlarm(alarmInfo);
        } catch (Throwable t) {
            t.printStackTrace();
        }

        return duplicateSubmissionCount;
    }

    public void init() {
        alarmSendingThread.start();
    }

    public synchronized void shutdown() {
        if (shutdownRequested) {
            throw new IllegalStateException("shutdown already requested!");
        }

        alarmSendingThread.terminate();
        shutdownRequested = true;
    }

}
```

从清单 14-2 的代码可知，AlarmMgr 的 sendAlarm 方法计算其接收的告警信息被重复发送到告警模块的次数是通过调用告警发送线程 AlarmSendingThread 的 sendAlarm 方法实现的。尽管这个计数的代码不是直接写在 AlarmMgr 的 sendAlarm 方法内，它实际上是运行在 AlarmMgr 的 sendAlarm 方法的执行线程（即业务线程）中。AlarmSendingThread 类的源码如清单 14-3 所示。

清单 14-3. AlarmSendingThread 类源码

```
/*
 * 告警发送线程。
 * 模式角色：HalfSync/HalfAsync.AsyncTask
 * 模式角色：Two-phaseTermination.ConcreteTerminatableThread
 */
public class AlarmSendingThread extends AbstractTerminatableThread {

    private final AlarmAgent alarmAgent = new AlarmAgent();

    /*
     * 告警队列。
     * 模式角色：HalfSync/HalfAsync.Queue
     */
    private final BlockingQueue<AlarmInfo> alarmQueue;
    private final ConcurrentMap<String, AtomicInteger> submittedAlarmRegistry;

    public AlarmSendingThread() {

        alarmQueue = new ArrayBlockingQueue<AlarmInfo>(100);
```

```
        submittedAlarmRegistry = new ConcurrentHashMap<String, AtomicInteger>();

        alarmAgent.init();
    }

    @Override
    protected void doRun() throws Exception {
        AlarmInfo alarm;
        alarm = alarmQueue.take();
        terminationToken.reservations.decrementAndGet();

        try {
            //将告警信息发送至告警服务器
            alarmAgent.sendAlarm(alarm);
        } catch (Exception e) {
            e.printStackTrace();
        }

        /*
         * 处理恢复告警：将相应的故障告警从注册表中删除，使得相应故障恢复后若再次出现相同故障，该
         * 故障信息能够上报到服务器。
         */
        if (AlarmType.RESUME == alarm.type) {
            String key = AlarmType.FAULT.toString() + ':' + alarm.getId() + '@'
                + alarm.getExtraInfo();
            submittedAlarmRegistry.remove(key);

            key = AlarmType.RESUME.toString() + ':' + alarm.getId() + '@'
                + alarm.getExtraInfo();
            submittedAlarmRegistry.remove(key);
        }

    }

    public int sendAlarm(final AlarmInfo alarmInfo) {
        AlarmType type = alarmInfo.type;
        String id = alarmInfo.getId();
        String extraInfo = alarmInfo.getExtraInfo();

        if (terminationToken.isToShutdown()) {
            // 记录告警
            System.err.println("rejected alarm:" + id + "," + extraInfo);
            return -1;
        }

        int duplicateSubmissionCount = 0;
        try {

            AtomicInteger prevSubmittedCounter;

        prevSubmittedCounter
            submittedAlarmRegistry.putIfAbsent(type.toString()
                + ':' + id + '@' + extraInfo, new AtomicInteger(0));
            if (null == prevSubmittedCounter) {
                terminationToken.reservations.incrementAndGet();
                alarmQueue.put(alarmInfo);
            } else {
```

```
                // 故障未恢复，不用重复发送告警信息给服务器，故仅增加计数
                duplicateSubmissionCount = prevSubmittedCounter.incrementAndGet();
            }
        } catch (Throwable t) {
            t.printStackTrace();
        }

        return duplicateSubmissionCount;
    }

    @Override
    protected void doCleanup(Exception exp) {
        if (null != exp && !(exp instanceof InterruptedException)) {
            exp.printStackTrace();
        }
        alarmAgent.disconnect();
    }

}
```

告警发送线程类 AlarmSendingThread 相当于 Half-sync/Half-async 模式的 SyncTask 参与者实例，同时它还应用了 Two-phase Termination 模式（参见第 5 章）。其线程处理逻辑方法 doRun 先从告警队列中取出告警信息，然后调用类 AlarmAgent 的 sendAlarm 方法将告警信息通过网络上报到告警服务器。这个调用是运行在告警线程这个工作者线程中的，而不是运行在业务线程中的。因此，这里上报告警的快慢并不影响业务模块的处理能力[1]，使得上述第一个问题得以解决。AlarmSendingThread 的实例变量 alarmQueue 相当于 Half-sync/Half-async 模式的 Queue 参与者实例，它充当了告警消息缓冲区的作用。

AlarmAgent 类的源码如清单 14-4 所示。

清单 14-4. AlarmAgent 类源码

```
/**
 * 负责连接告警服务器，并发送告警信息至告警服务器
 *
 */
public class AlarmAgent {
    // 用于记录 AlarmAgent 是否连接上告警服务器
    private volatile boolean connectedToServer = false;

    // 模式角色：GuardedSuspension.Predicate
    private final Predicate agentConnected = new Predicate() {
        @Override
        public boolean evaluate() {
            return connectedToServer;
        }
    };

    // 模式角色：GuardedSuspension.Blocker
    private final Blocker blocker = new ConditionVarBlocker();
```

1 这其实是相对的。若同步层处理过慢，最终也可能拖慢异步层的处理。下文会提到这点。

```
// 心跳定时器
private final Timer heartbeatTimer = new Timer(true);

// 省略其他代码

/**
 * 发送告警信息
 * @param alarm 告警信息
 * @throws Exception
 */
public void sendAlarm(final AlarmInfo alarm) throws Exception {
    // 可能需要等待，直到AlarmAgent连接上告警服务器（或者连接中断后重新连上服务器）
    // 模式角色：GuardedSuspension.GuardedAction
    GuardedAction<Void> guardedAction = new GuardedAction<Void>(agentConnected) {
        public Void call() throws Exception {
            doSendAlarm(alarm);
            return null;
        }
    };

    blocker.callWithGuard(guardedAction);
}

// 通过网络连接将告警信息发送给告警服务器
private void doSendAlarm(AlarmInfo alarm) {
    // 省略其他代码
    Debug.info("sending alarm " + alarm);

    // 模拟发送告警至服务器的耗时
    try {
        Thread.sleep(50);
    } catch (Exception e) {

    }
}

public void init() {
    // 省略其他代码

    // 告警连接线程
    Thread connectingThread = new Thread(new ConnectingTask());

    connectingThread.start();

    heartbeatTimer.schedule(new HeartbeatTask(), 60000, 2000);
}

public void disconnect() {
    // 省略其他代码
    Debug.info("disconnected from alarm server.");
    connectedToServer = false;
}

protected void onConnected() {
    try {
        blocker.signalAfter(new Callable<Boolean>() {
            @Override
```

```
                public Boolean call() {
                    connectedToServer = true;
                    Debug.info("connected to server");
                    return Boolean.TRUE;
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    protected void onDisconnected() {
        connectedToServer = false;
    }

    // 负责与告警服务器建立网络连接
    private class ConnectingTask implements Runnable {
        @Override
        public void run() {
            // 省略其他代码

            // 模拟连接操作耗时
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                ;
            }

            onConnected();
        }
    }

    /**
     * 心跳定时任务：定时检查与告警服务器的连接是否正常，发现连接异常后自动重新连接。
     */
    private class HeartbeatTask extends TimerTask {
        // 省略其他代码

        @Override
        public void run() {
            // 省略其他代码

            if (!testConnection()) {
                onDisconnected();
                reconnect();
            }

        }

        private boolean testConnection() {
            // 省略其他代码

            return true;
        }

        private void reconnect() {
            ConnectingTask connectingThread = new ConnectingTask();
            // 直接在心跳定时器线程中执行
```

```
                connectingThread.run();
            }

        }
}
```

AlarmAgent 应用了 Guarded Suspension 模式（参见第 4 章）。相关代码见清单 4-2、清单 4-3、清单 4-4 和清单 4-5。

14.4 Half-sync/Half-async 模式的评价与实现考量

Half-sync/Half-async 模式既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性。Half-sync/Half-async 模式通过把低级任务或者耗时较短的任务安排在异步层，减少了客户端的等待，有利于提升系统的吞吐率。而高级任务或耗时较长的任务被安排在同步层，这使得我们可以在不影响客户端代码处理性能的情况下保持了同步编程的简单性。

独立的并发访问控制策略。Half-sync/Half-async 模式的分层架构使得各个层次的代码可以有各自的并发访问控制策略。这似乎很显而易见，但是它却蕴含了 Half-sync/Half-async 模式的几个重要的应用场景。例如，在 Java Swing 中，为了避免 GUI 组件（如按钮）死锁（Dead lock），Swing 的 GUI 层的代码特意采用单线程模型（Swing 的 GUI 层代码运行在 Event Loop 线程中）。这样就可以避免使用锁，也就自然地避免了死锁。但是，这同时也带来另外的问题：如果要在 Swing 的 Event Loop 线程中执行耗时较长的任务（如从服务器上下载大文件），那么 GUI 就会被“冻住”而无法响应用户操作。为了解决这个问题，JDK 1.6 引入了 SwingWorker 类。该类应用了 Half-sync/Half-async 模式。SwingWorker 就像一个转换器，其一端（异步层）连接着 Swing 的单线程 GUI 层，另一端（同步层）连接着多线程的应用代码层（如从服务器上下载文件的代码）。这样，耗时较长的应用代码仍然可以多线程执行，比如多个线程分段从服务器上下载大文件，而不影响 Swing GUI 层的单线程模式。

另外一个典型的例子是，Half-sync/Half-async 模式的异步层对接的是多线程环境，因此该层通常需要并发访问控制措施，如显式锁。而同步层是一个单线程环境，因此它无须任何并发访问控制即可保证线程安全。例如，同步层的代码需要调用一些非线程安全的 API，而我们又不想因此引入显式锁，同时又要保证线程安全。此时，这种异步层对接多线程环境，同步层采用单线程模型的 Half-sync/Half-async 模式就可以派上用场。这种场景可以使我们用单线程编程这种方式去编写同步层的代码，从而使同步层的代码进一步简单化。

14.4.1 队列积压

Half-sync/Half-async 模式可以看成是 Producer-Consumer 模式的一个实例。AsyncTask 和 SyncTask 分别相当于 Producer-Consumer 模式的 Producer 和 Consumer 参与者。Producer-Consumer 模式在实际应用中常见的一个问题是 Consumer 处理“产品”的速率往往比 Producer“生产”“产品”的速率慢。如果同步层的处理速率过慢的情况持续存在，那么队列就会逐渐积压。如果 Producer 和 Consumer 采用有界队列进行通信，那么当队列积压到队列满的时候就会导致 Producer 的处理速率受到影响。这是因为 Producer 在“生产”好“产品”将其放入队列的时候，需要等待队列非满（由满变成不满的状态）。如果 Producer 和 Consumer 采用无界队列进行通信，那么队列持续积压到一定程度可能就导致内存溢出，这是因为队列中存储的大量元素占用了大量内存无法被垃圾回收（Garbage collect）掉。

因此，在使用 Half-sync/Half-async 模式的时候，需要根据实际情况在有界队列和无界队列之间做出选择。选择有界队列有个隐含的好处：当队列积压到满的情况下，Producer 会因其需要等待队列非满而降低处理速率，从而在一定程度上减轻了 Consumer 的处理负担[2]。有界队列可以使用 java.util.concurrent.ArrayBlockingQueue。ArrayBlockingQueue 用于存储队列元素的存储空间是预先分配的，这意味着它在使用过程中内存开销较小（无须动态申请存储空间）。无界队列可以使用 java.util.concurrent.LinkedBlockingQueue。LinkedBlockingQueue 用于存储队列元素的存储空间是在其使用过程中动态分配的，因此它可能会增加 JVM 垃圾回收的负担。

14.4.2 避免同步层处理过慢

同步层中的高级任务往往涉及 I/O 这种比较慢的操作，如网络读/写、数据库操作以及文件读/写等。这些任务执行的快慢往往取决于其依赖的外部资源（如数据库服务器）。而这些外部资源又往往不在我们的控制范围之内。因此要避免同步层处理过慢，还要从同步层自身的设计入手。例如，某同步任务需要通过广域网发送请求给服务器。网速、对端服务器性能等因素在极端的情况下可能导致执行该同步任务的线程由于等待对端响应而全部被阻塞，从而使得队列中的新增元素没有空闲线程去处理。这种情形下，该同步任务可能需要改用非阻塞 I/O（如基于 Java NIO 的 API），并采用专门的线程去处理 I/O 等待事件。

2 这当然也是相对的。如果队列持续满，而 Producer 又不断地“生产”“产品”，最终也可能导致 Producer 线程全部被阻塞，导致其无法对外服务。

14.5 Half-sync/Half-async 模式的可复用实现代码

如清单 14-5 所示的代码展示了一个可复用的 Half-sync/Half-async 模式实现。清单 14-5 中的 AsyncTask 类相当于 Half-sync/Half-async 模式的 AsyncTask 参与者。其 dispatch 方法会先调用 onPreExecute 方法，以执行耗时较短的任务，对客户端的输入做一些初步处理。然后，dispatch 方法会把 doInBackround 方法中实现的耗时较长的任务提交到同步层，由 java.util.concurrent.Executor 实现类负责执行。AsyncTask 类的 executor 实例变量相当于 Half-sync/Half-async 模式的 SyncTask 参与者，它负责执行 doInBackground 中实现的同步任务。

总而言之，AsyncTask 类的 dispatch 方法运行在异步层中，其调用的 onPreExecute 方法用于执行耗时较短的任务。AsyncTask 类的 doInBackground 方法用于执行耗时较长的任务，它运行在同步层的工作者线程中。

清单 14-5. Half-sync/Half-async 模式的可复用实现

```
/**
 * Half-sync/Half-async 模式的可复用实现。
 *
 * @author Viscent Huang
 *
 * @param <V>
 *          同步任务的处理结果类型
 */
public abstract class AsyncTask<V> {

    // 相当于 Half-sync/Half-async 模式的同步层：用于执行异步层提交的任务。
    private volatile Executor executor;
    private final static ExecutorService DEFAULT_EXECUTOR;

    static {
        DEFAULT_EXECUTOR = new ThreadPoolExecutor(1, 1, 8, TimeUnit.HOURS,
            new LinkedBlockingQueue<Runnable>(), new ThreadFactory() {

                @Override
                public Thread newThread(Runnable r) {
                    Thread thread = new Thread(r, "AsyncTaskDefaultWorker");

                    // 使该线程在 JVM 关闭时自动停止
                    thread.setDaemon(true);
                    return thread;
                }

            }, new RejectedExecutionHandler() {


                /**
                 * 该 RejectedExecutionHandler 支持重试。
                 * 当任务被 ThreadPoolExecutor 拒绝时，
                 * 该 RejectedExecutionHandler 支持
```

```
                 * 重新将任务放入 ThreadPoolExecutor
                 * 的工作队列（这意味着，此时客户端代码
                 * 需要等待 ThreadPoolExecutor 的队列非满）。
                 */
                @Override
                public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
                    if (!executor.isShutdown()) {
                        try {
                            executor.getQueue().put(r);
                        } catch (InterruptedException e) {
                            ;
                        }
                    }

                }

            });
    }

    public AsyncTask(Executor executor) {
        this.executor = executor;
    }

    public AsyncTask() {
        this(DEFAULT_EXECUTOR);
    }

    public void setExecutor(Executor executor) {
        this.executor = executor;
    }

    /**
     * 留给子类实现耗时较短的任务，默认实现什么也不做。
     *
     * @param params
     *          客户端代码调用 dispatch 方法时所传递的参数列表
     */
    protected void onPreExecute(Object... params) {
        // 什么也不做
    }

    /**
     * 留给子类实现。用于实现同步任务执行结束后所需执行的操作。默认实现什么也不做。
     *
     * @param result
     *          同步任务的处理结果
     */
    protected void onPostExecute(V result) {
        // 什么也不做
    }

    protected void onExecutionError(Exception e) {
        e.printStackTrace();
    }

    /**
     * 留给子类实现耗时较长的任务（同步任务），由后台线程负责调用。
```

```
     *
     * @param params
     *          客户端代码调用 dispatch 方法时所传递的参数列表
     * @return 同步任务的处理结果
     */
    protected abstract V doInBackground(Object... params);

    /**
     * 对外（其子类）暴露的服务方法。该类的子类需要定义一个比该方法命名更为具体的服务方法（如
     * downloadLargeFile）。
     * 该命名具体的服务方法（如 downloadLargeFile）可直接调用该方法。
     *
     * @param params
     *          客户端代码传递的参数列表
     * @return 可借以获取任务处理结果的 Promise（参见第 6 章，Promise 模式）实例。
     */
    protected Future<V> dispatch(final Object... params) {
        FutureTask<V> ft = null;

        // 进行异步层初步处理
        onPreExecute(params);

        Callable<V> callable = new Callable<V>() {
            @Override
            public V call() throws Exception {
                V result;
                result = doInBackground(params);
                return result;
            }

        };

        ft = new FutureTask<V>(callable) {

            @Override
            protected void done() {
                try {
                    onPostExecute(this.get());
                } catch (InterruptedException e) {
                    onExecutionError(e);
                } catch (ExecutionException e) {
                    onExecutionError(e);
                }
            }

        };

        // 提交任务到同步层处理
        executor.execute(ft);

        return ft;
    }

}
```

使用 AsyncTask 类来实现 Half-sync/Half-async 模式，应用代码只需要创建一个 AsyncTask 类

的子类（或者匿名子类），并在子类中完成以下几个“填空”任务。

1.【必需】定义一个含义具体的服务方法名，该方法可直接调用父类的 dispatch 方法。

2.【必需】实现父类的抽象方法 doInBackground 方法。

3.【可选】根据应用的实际需要覆盖父类的 onPreExecute 方法、onPostExecute 方法和 onExecutionException 方法。

清单 14-6 展示了一个使用 AsyncTask 类实现的 Half-sync/Half-async 模式客户端代码。

清单 14-6. 基于可复用代码实现的 Half-sync/Half-async 模式客户端示例

```
public class SampleAsyncTask {

    public static void main(String[] args) {
        XAsyncTask task = new XAsyncTask();
        List<Future<String>> results = new LinkedList<Future<String>>();

        try {
            results.add(task.doSomething("Half-sync/Half-async", 1));

            results.add(task.doSomething("Pattern", 2));

            for (Future<String> result : results) {
                Debug.info(result.get());
            }

            Thread.sleep(200);
        } catch (Exception e) {

            e.printStackTrace();
        }

    }

    private static class XAsyncTask extends AsyncTask<String> {

        @Override
        protected String doInBackground(Object... params) {
            String message = (String) params[0];
            int sequence = (Integer) params[1];
            Debug.info("doInBackground:" + message);
            return "message " + sequence + ":" + message;
        }

        @Override
        protected void onPreExecute(Object... params) {
            String message = (String) params[0];
            int sequence = (Integer) params[1];
            Debug.info("onPreExecute:[" + sequence + "]" + message);
        }

        public Future<String> doSomething(String message, int sequence) {
            if (sequence < 0) {
```

```
                throw new IllegalArgumentException("Invalid sequence:" + sequence);
            }
            return this.dispatch(message, sequence);
        }
    }
}
```

如清单 14-6 所示的程序运行的输出类似如下。

```
[2015-04-22 20:26:19.333][INFO][main]:onPreExecute:[1]Half-sync/Half-async
[2015-04-22 20:26:19.335][INFO][main]:onPreExecute:[2]Pattern
[2015-04-22
20:26:19.336][INFO][AsyncTaskDefaultWorker]:doInBackground:Half-sync/Half-async
[2015-04-22 20:26:19.336][INFO][main]:message 1:Half-sync/Half-async
[2015-04-22 20:26:19.337][INFO][AsyncTaskDefaultWorker]:doInBackground:Pattern
[2015-04-22 20:26:19.337][INFO][main]:message 2:Pattern
```

从上面的输出可以看出，onPreExecute 方法是运行在 main 线程（客户端线程）中的，而 doInBackground 方法是运行在后台线程 AsyncTaskDefaultWorker（工作者线程）中的。

14.6 Java 标准库实例

Java Swing 为了避免其 GUI 组件（如按钮）出现死锁，而特意采用单线程模型。所有的 GUI 组件都是运行在唯一的一个线程，即 Event Loop 线程中的。在 Event Loop 线程中运行耗时较长的任务会导致用户界面被“冻住”而无法响应用户操作。为了解决这个问题，JDK 1.6 引入了 SwingWorker 类。该类应用了 Half-sync/Half-async 模式。SwingWorker 使得一些耗时较短的任务运行在 Event Loop 线程中，而耗时较长的任务运行在其维护的后台线程中。因此，SwingWorker 相当于 Half-sync/Half-async 模式的 AsyncTask 参与者实例。其 doInBackground 方法相当于 AsyncTask 参与者的 dispatch 方法。而 SwingWorker 内部维护的后台线程则相当于 SyncTask 参与者实例，它们负责真正执行耗时较长的任务。

14.7 相关模式

14.7.1 Two-phase Termination 模式（第 5 章）

Half-sync/Half-async 模式的同步层可能使用 Two-phase Termination 模式来创建可停止的线程用于执行同步任务。本章案例中的告警发送线程就是这样一个例子。

14.7.2 Producer-Consumer 模式（第 7 章）

Half-sync/Half-async 模式可以看成是 Producer-Consumer 模式的一个实例。Half-sync/Half-async 模式的 AsyncTask 参与者和 SyncTask 参与者分别相当于 Producer-Consumer 模式的 Producer 参与者和 Consumer 参与者。

14.7.3 Active Object 模式（第 8 章）

Half-sync/Half-async 模式可以实现减少客户端代码等待的效果。这点，使用 Active Object 模式也可以实现。Half-sync/Half-async 模式与 Active Object 模式的区别要从二者的意图来理解。前者的意图在于通过将异步任务和同步任务的结合和解耦来实现简单和性能的平衡。而后者的意图是通过解耦方法的调用（Invocation）与执行（Execution）来提高并发性。二者的相同之处在于借助异步编程来提高性能。不同之处在于前者更加强调哪些任务适合放入异步层。事实上，如果我们将如图 14-1 所示的类图中的同步层和队列层合起来看，就不难发现这两层合起来可以用 Active Object 模式实现。

本章案例中 AlarmMgr 的 sendAlarm 方法接收到告警信息后需要立即计算该告警信息被重复发送到告警模块的次数。因此，这时使用 Half-sync/Half-async 模式更为明了：将计数任务放入异步层。

14.7.4 Thread Pool 模式（第 9 章）

Half-sync/Half-async 模式的同步层可能需要使用线程池，以便采用一定数量的线程就可以完成同步任务的执行，减少 JVM 线程调度的负担。

14.8 参考资源

1．Schmidt, Douglas et al. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

2．SwingWorker 源码. http://grepcode.com/file/repository.grepcode.com/java/root/ jdk/openjdk/6-b27/javax/swing/SwingWorker.java#SwingWorker.

3．Android AsyncTask API 文档. http://developer.android.com/reference/android/os/ AsyncTask.Html.

4．Android AsyncTask 源码. https://github.com/android/platform_frameworks_base/blob/master/core/java/android/os/AsyncTask.java.

第 15 章

模式语言

15.1 模式与模式间的联系

设计模式并不是孤立的，一个设计模式往往和其他设计模式存在某些关联，如图 15-1 所示。

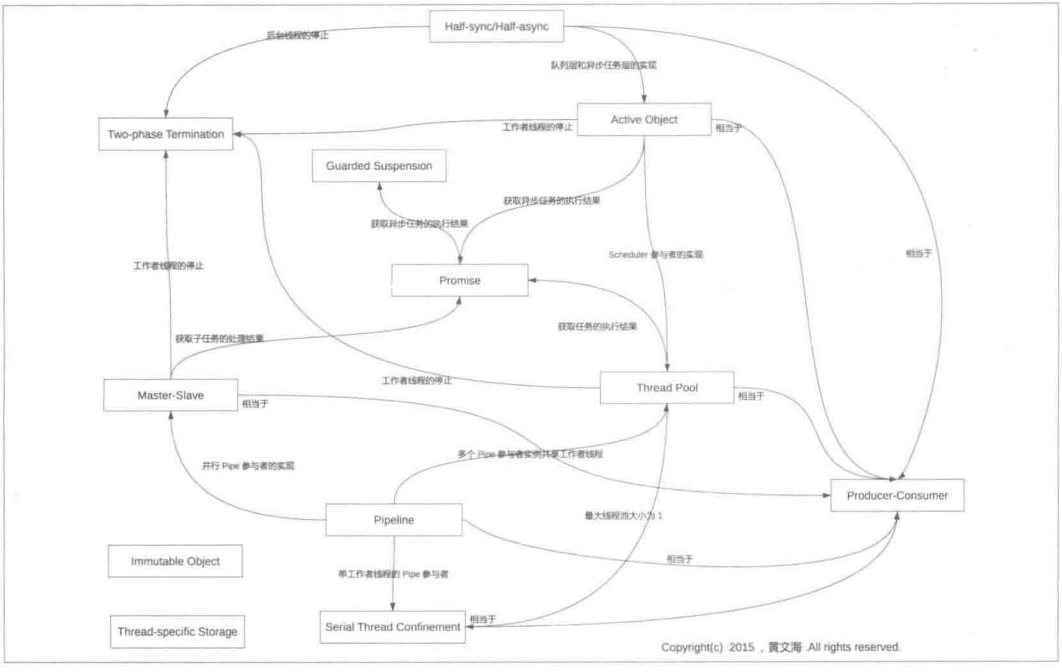


图 15-1　设计模式间的联系

设计模式之间的关系可以概括为：支持、变体、组合和备选这 4 种关系。

支持[1]（Support）。具体的一个设计模式能够解决特定的问题，但是应用特定的设计模式本身也会引入特定的问题，而这些问题往往可以使用另外一些设计模式来解决，即在此情景下一个设计模式为另外一个设计模式解决其面临的问题提供了支持。这好比一种药物可以治疗某种疾病，但是它本身又会对人体产生一些副作用（如伤害肝脏），因此医生为病人开相应药物的时候又开了另外用于抵消该药物副作用的其他药物。例如，使用 Master-Slave 模式（第 12 章）可以提升服务的响应性，但其应用本身也会带来一些问题：Master-Slave 模式中，Master 参与者需要获取由各个 Slave 参与者实例的工作者线程执行的子任务的处理结果，而这点可以通过使用 Promise 模式（第 6 章）来实现。另外，Master 参与者需要提供一个用于停止各个 Slave 参与者的工作者线程的接口（方法），该接口的实现可以借助 Two-phase Termination 模式（第 5 章）。

变体（Variant）。在特定的情况下，一个设计模式可以看作另外一个设计模式的特例（变体）。这好比正方体可以看作长和宽相等的“特殊”长方体，这里我们可以说正方体是长方体的一个变体，相应的“特定情况”是“长和宽相等”。例如，就 Thread Pool 模式（第 9 章）而言，当其最大线程池大小为 1 时，相应的 Thread Pool 模式实现就等效于一个 Serial Thread Confinement 模式（第 11 章）实现。此时，这两个模式无论是从架构上看还是从解决的问题（线程安全问题）上看都是相似的。

组合（Combination）。俗话说“一把钥匙开一把锁”。如果我们把一个具体的设计模式比作一把锁，而把我们要解决的问题比作要开的门，那么我们不能指望用一把锁来开启所有要开启的门。在实际解决问题的过程中，我们往往需要使用多个设计模式。在此情景下，涉及的各个设计模式之间的关系即为组合关系，即它们在特定场景下组合在一起来解决问题。例如，本书第 10 章（Thread Specific Storage 模式）提到的实战案例中的验证码短信问题，生成随机验证码的时候我们使用了 Thread Specific Storage 模式以避免共享 ecureRandom 实例可能造成不必要等待的问题；另外，将验证码通过短信发送到用户手机的时候，我们使用了 Thread Pool 模式（第 9 章）以避免在并发用户量大的时候下发大量验证码短信导致系统创建的负责短信发送的线程过多。

备选（Alternative）。俗话说“条条大路通罗马”。一个（类）特定的问题往往可以采用不同的方法来解决。不同的方法采用不同的方式去解决、各自有其适用的条件和场景。例如，多线程编程经常需要解决的一个问题是线程安全的问题。本书提供了可以解决该问题的 3 个设计模式：Immutable Object 模式（第 3 章）、Thread Specific Storage 模式（第 10 章）和 Serial

1 也有文献称之为改良（Refinement）。但笔者以为这样称呼不太能够全面反映相应的模式间关系。例如，Master-Slave 模式中，Master 需要管理 Slave 参与者的工作者线程的生命周期，其停止工作者线程的实现可以借助 Two-phase Termination 模式。此时，称 Two-phase Termination 模式与 Master-Slave 模式之间的关系为“支持”比“改良”更贴切一些。

Thread Confinement 模式（第 11 章）。也就是说，对于线程安全问题的解决，我们有 3 个设计模式可选择。那么，这 3 个设计模式此时就形成了备选关系。

15.2 Immutable Object（不可变对象）模式

别名

该模式也被称为 Immutable（不可变）模式。

背景

多个线程共享一个对象的实例。

问题

当被共享的对象相应的现实世界实体的状态变更时，系统对此要有所反映。但是，通过直接更改该被共享的对象的状态来反映这个变更通常会导致锁的引入以保证线程安全。

解决方案

将相应现实世界实体建模为状态不可变的对象。当相应实现世界实体的状态变更时，系统通过创建新的对象实例来反映这种状态变更，而不是更改对象本身的状态。

结果

- 多个线程可以在不使用锁的情况下，以线程安全的方式去访问共享对象。
- 可能导致频繁的对象创建。

相关模式

Thread Specific Storage 模式（第 10 章）和 Serial Thread Confinement 模式（第 11 章）也可以在不引入锁的情况下确保线程安全。

15.3 Guarded Suspension（保护性暂挂）模式

别名

该模式也被称为 Guarded Waits（受保护等待）模式。

背景

一个方法欲执行的操作(目标动作)所需的前提条件可能暂时无法满足而稍后可能得以满足。

问题

多线程环境中，某个对象的方法被调用时，该方法欲执行的操作所需的状态暂时没有得到满足，而稍后可能得以满足。因此，此时如果该方法返回或者抛出异常，则会迫使客户端代码对其不期望的结果进行处理。

解决方案

多线程环境中，当某个对象的方法（受保护方法）执行其欲执行的操作所需的状态（保护条件）未被满足时，将当前线程暂挂直到其他线程改变了该对象的状态使得保护条件得以满足时，被暂挂的线程得以唤醒。

结果

- 使应用程序避免了样板式（Boilerplate）代码。
- 实现了关注点分离（Separation of Concern）。
- 可能增加 JVM 垃圾回收的负担。
- 可能增加上下文切换（Context Switch）。

相关模式

Promise 模式(第 6 章)和 Producer-Consumer 模式(第 7 章)实现过程中可能需要使用 Guarded Suspension 模式。

15.4 Two-phase Termination（两阶段终止）模式

别名

无。

背景

系统需要防止用户线程（User Thread）阻止 JVM 正常关闭。

问题

守护线程（Daemon Thread）不会阻止 JVM 正常关闭，而用户线程会阻止 JVM 正常关闭。因此，正常关闭 JVM 时需要先将用户线程停止。但是，停止一个用户线程时，我们希望该线程能够在其处理完待处理的任务后再行停止。

解决方案

将线程的停止分为两个阶段：准备阶段和执行阶段。准备阶段主要实现线程停止的标志的设置，执行阶段主要实现线程停止标志的检测并在线程处理完待处理的任务后停止线程。

结果

- **实现了线程的优雅停止**：线程可以在其处理完待处理的任务后再行停止，而非粗暴地停止。
- **可能延迟线程的停止**：待停止的线程可能是在其处理完待处理的任务后再停止的，而这可能需要一定的等待时间。

相关模式

Producer-Consumer 模式（第 7 章）和 Master-Slave 模式（第 12 章）可能需要使用 Two-phase Termination 模式以实现其工作者线程的停止。

15.5 Promise（承诺）模式

别名

该模式也被称为 Future（期货）模式。

背景

一个对象需要使用另外一个对象的某个方法（以下称为目标方法）的返回值。

问题

目标方法需要消耗较长的处理时间才能返回表示其处理结果的值。在该方法返回之前，客户端代码会被阻塞而无法进行其他处理。

解决方案

使用异步编程，将目标方法的返回值改为一个凭据对象，而不是表示目标方法真正处理结果的对象（结果对象）。客户端代码通过调用凭据对象的某个方法来获取目标方法的结果对象。在此基础上，采用专门的工作者线程或者线程池去执行目标方法所进行的计算。

结果

- 既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性——客户端代码的编写方式与同步编程无本质差别。
- 一定程度上屏蔽了异步编程和同步编程的差异：无论目标方法是异步方法还是同步方法，它都不影响客户端代码的编写方式。

相关模式

目标方法可以看成 Factory Method 模式[2]中的工厂方法。

凭据对象用于获取结果对象的方法可能需要等待目标方法对应的计算完成才能返回，这可以使用 Guarded Suspension 模式（第 4 章）来实现。

Active Object 模式（第 8 章）可以看成是包含了 Promise 模式的复合模式，其异步方法的返回值就是一个凭据对象。

15.6 Producer-Consumer（生产者/消费者）模式

别名

无。

背景

数据（任务）的提供方的处理能力（速率）与相应的使用方的处理能力（速率）不均衡，或者我们不希望二者的处理能力（速率）相互影响。

问题

数据（任务）的提供方和使用方运行在同一个线程中会导致一方处理能力（速率）的大小对

2 Factory Method 模式是 GOF 设计模式中的一个，不在本书的讨论范围。

另外一方产生影响，即造成等待。

解决方案

在数据（任务）的提供方和使用方之间引入一个作为缓冲区的通道，从而使数据（任务）的提供方和使用方可以运行在各自的线程之中。

结果

- 数据（任务）的提供方和使用方的处理能力相对来说互不影响。
- 关注点分离（Separation of Concern）：数据（任务）的提供方只需要将数据（任务）存入通道即可，它无须关心谁对数据（任务）进行处理；而数据（任务）的使用方只需要从通道中取出数据（任务）进行处理而无须关心是谁将其存入通道的。

相关模式

许多模式可看成 Producer-Consumer 模式的一个实例。

15.7 Active Object（主动对象）模式

别名

该模式也被称为 Concurrent Object 模式。

背景

客户端代码需要访问独立的线程控制（Thread of Control）对象。

问题

客户端代码需要使用某个类提供的服务，但是不希望等待相应的服务调用完成后才能继续其他处理，以避免响应性和吞吐率受此服务调用的影响。

解决方案

将服务方法的调用（Invocation）和执行（Execution）进行解耦（Decoupling）。客户端代码调用某个服务方法时，该方法并不立即执行相应的服务操作，而是生成表示相应服务操作的对象（任务）并将其存入缓存区，由专门的工作者线程取缓冲区中的任务进行执行。

结果

- 有利于提高并发性，从而提高系统的吞吐率。
- 使调试变得复杂。

相关模式

Active Object 模式可看成 Producer-Consumer 模式（第 7 章）的一个实例。

Active Object 模式使用了 Promise 模式（第 6 章）以实现客户端代码获取异步任务的处理结果。

15.8 Thread Pool（线程池）模式

别名

无。

背景

多线程环境中，新的任务不断产生。

问题

为每个新的任务都创建一个线程去负责处理过程会导致线程不断地被创建和销毁，这会增加系统的资源消耗和上下文切换。

解决方案

将待处理的任务存入缓冲区，并创建一定数量的工作者线程，复用这些工作者线程使其从缓冲区中取出任务执行。

结果

- 抵消线程创建的开销，提高系统的响应性。
- 封装了工作者线程生命周期管理。
- 减少销毁线程的开销。
- 不恰当的使用可能导致死锁。

相关模式

Thread Pool 模式可看成 Producer-Consumer 模式（第 7 章）的一个实例。

Thread Pool 模式可以使用 Two-phase Termination 模式(第 5 章)来实现其工作者线程的停止。

15.9 Thread Specific Storage（线程特有存储）模式

别名

该模式也被称为 Thread Local Storage 模式。

背景

多个线程需要访问同一个非线程安全对象。或者，使用线程安全的对象，但希望能够避免其使用的锁的开销。

问题

多个线程访问同一个非线程安全对象（TSObject）可能产生线程安全问题，而我们又不希望因此而引入锁，以便能够避免锁的开销和相关问题。

解决方案

使每个线程获得一个（且仅一个）该线程所特有的 TSObject 实例，各个线程仅访问各自的 TSObject 实例，一个 TSObject 实例不会被多个线程共享。

结果

- 在不引入锁的情况下实现了对非线程安全对象访问的线程安全。
- 易于使用。
- 隐藏了系统结构，可能使系统难于理解。
- 鼓励了全局对象的使用。
- 不恰当的使用可能导致内存泄漏。

相关模式

Immutable Object 模式（第 3 章）和 Serial Thread Confinement 模式（第 11 章）也能够在不引

入锁的情况下确保线程安全。

15.10 Serial Thread Confinement（串行线程封闭）模式

别名

无。

背景

异步编程中，工作者线程需要访问非线程安全对象，而我们又不希望因此而引入锁。

问题

系统对某种并发任务的处理涉及非线程安全对象的访问，而我们又不希望因此而引入锁，以便能够避免锁的开销和相关问题。

解决方案

将并发任务通过队列串行化，再创建唯一的一个工作者线程对队列中的任务进行执行。

结果

- 在不引入锁的情况下实现了对非线程安全对象访问的线程安全。
- 如果客户端代码关心任务的处理结果，那么可能导致多个客户端线程等待同一个工作者线程的处理结果。

相关模式

Serial Thread Confinement 模式可看成 Producer-Consumer 模式（第 7 章）的一个实例。

Immutable Object 模式（第 3 章）和 Thread Specific Storage 模式（第 10 章）也能够在不引入锁的情况下确保线程安全。

如果客户端代码关心任务的处理结果，那么我们可以借用 Promise 模式（第 6 章）来实现这点。

15.11 Master-Slave（主仆）模式

别名

该模式也被称为 Boss-Worker（老板-伙计）模式。

背景

一个任务被分解为等同语义（Semantically-identical）的若干个子任务。

问题

分而治之（Divide and Conquer）是解决许多问题的一个通用原则。将一个任务（原始任务）分解为若干个子任务，再让这些子任务独立执行。然后将各个子任务的处理结果组合成原始任务的处理结果。这个过程需要处理好以下几个方面。

- 客户端代码不应该知道其调用的服务是基于分而治之的计算。
- 无论是客户端代码还是子任务，它们都应该不依赖于任务分解和子任务处理结果合并的算法。

解决方案

在服务的客户端代码和子任务的处理之间引入一个协调性的对象（即 Master）。有关分而治之的相关细节被封装在 Master 里面。各个子任务由专门的工作者线程负责处理。

结果

- **提升了计算性能**：子任务可以并行执行。
- **可交换性（Exchangeability）和可扩展性（Extensibility）**：替换某个 Slave 实例、增加一个 Slave 实例对 Master 参与者产生的影响很小。

相关模式

Master-Slave 模式可看成 Producer-Consumer 模式（第 7 章）的一个实例。

Master-Slave 模式中的 Master 参与者可能会使用 Promise 模式（第 6 章）以获取子任务的处理结果。

15.12 Pipeline（流水线）模式

别名

无。

背景

多线程编程中，规模较大的任务的处理可以分解为若干个存在依赖关系的子任务。

问题

规模较大的任务（原始任务）的处理可能比较耗时。如果对原始任务进行纵向分解，即分解得来的子任务中的每个任务的处理又包括若干个步骤，那么即使我们采用若干个工作者线程去负责执行子任务的执行也仍然避免不了一个子任务的处理中所出现的等待（一个处理步骤的开始要等待前一个处理步骤的完成）。

解决方案

对原始任务进行横向分解，即将一个任务的处理分解为若干个处理阶段（Stage），其中每个处理阶段的输出作为下一个处理阶段的输入，并且各个处理阶段都有相应的工作者线程去执行相应计算。

结果

- 可以对有依赖关系的任务实现并行处理。
- 为局部使用单线程模型编程提供了便利。
- 具备任务处理逻辑安排的灵活性。

相关模式

Pipeline 模式中的处理阶段可能会使用 Serial Thread Confinement 模式（第 11 章），以实现任务处理的线程安全。

Pipeline 模式可以借助 Master-Slave 模式（第 12 章）实现某个处理阶段的并行处理。

15.13 Half-sync/Half-async（半同步/半异步）模式

别名

无。

背景

某计算同时涉及低级（或耗时较短）任务和高级（或耗时较长）任务。

问题

低级（或耗时较短）的任务可以直接在客户端线程中执行，但是高级（或耗时较长）任务在客户端线程中执行则会增加客户端线程的等待从而减少吞吐率并降低响应性。

解决方案

采用分层架构。将低级（或耗时较短）任务放在异步层由客户端线程执行，高级（或耗时较长）任务放在同步层由专门的后台工作者线程执行。异步层和同步层不直接通信，而是通过队列层进行通信。

结果

- 既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性。
- 各层代码可以使用独立的并发访问控制策略。

相关模式

Half-sync/Half-async 模式可看成 Producer-Consumer 模式（第 7 章）的一个实例。

Half-sync/Half-async 模式可能会使用 Two-phase Termination 模式（第 5 章）来停止后台工作者线程。

Half-sync/Half-async 模式的队列层和同步层合起来可以使用 Active Object 模式（第 8 章）来实现。

Thread Pool 模式（第 9 章）可以用来实现同步层任务的执行。

附录 A

本书常用 UML 图指南

A.1 UML 简介

UML（Unified Modeling Language，统一建模语言）是软件工程领域一种标准化[1]通用图形化建模语言，它被设计用于以可视化的方式来描述软件系统。UML 类似于大家所熟悉的 E-R 图（Entity-Relationship Diagram）。E-R 图是对数据库进行建模的图形化语言，而 UML 是对面向对象的软件系统进行建模的图形化语言。

UML 的名字中之所以带有“统一”(Unified)字样，是因为它综合了 Grady Booch、Ivar Jacobson 和 James Rumbaugh 这 3 人提出的建模方法，并在此基础上进一步发展。

UML 定义的图形可以分为结构型（Structural UML Diagrams）和行为型（Behavioral UML Diagrams）两种类型。

结构型 UML 图包括：类图（Class Diagram）、组件图（Component Diagram）、复合结构图（Composite Structure Diagram）、部署图（Deployment Diagram）、对象图（Object Diagram）、包图（Package Diagram）和轮廓图（Profile Diagram）。

行为型 UML 图包括：活动图（Activity Diagram）、通信图（Communication Diagram）、交互概览图（Interaction Overview Diagram）、序列图（Sequence Diagram）、状态图（State Diagram）、时限图（Timing Diagram）和用例图（Use Case Diagram）。

1 UML 于 2005 年被国际标准化组织（ISO）发布为标准。

A.2 类图（Class Diagram）

类图用于描述类以及类与类之间的关系。它是从静态的角度对系统进行描述。

A.2.1 类的属性、方法和立体型（Stereotype）

图 A-1 展示了一个用于对队列进行建模的类图。类图中包含的信息包括类名、类型参数、立体型（Stereotype）、属性列表和方法列表。

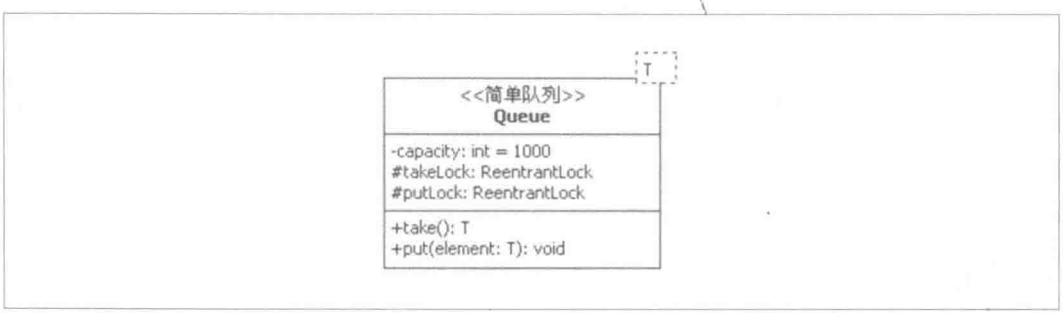


图 A-1. 对队列进行建模的类图

UML 使用矩形框来表示类。矩形框的第 1 个方格中的文字表示类名。类名上方用书名号括起来的文字被称为立体型（Stereotype），它是用来对相应的类做出额外的说明。

矩形框右上角虚线框中的文字表示相应的类型参数，对应到 Java 语言就是指范型（Generic）参数。

矩形框的第 2 个方格表示类的属性列表。这个方格往往只是列出相应类的关键[2]属性，而非全部属性。属性列表中的每个属性的声明格式如下：

[属性的可见性]属性名[:属性值类型][=属性的初始值]

其中，属性的可见性及其表示符号如表 A-1 所示。

2 所谓关键是相对于 UML 作者当前所要阐述的话题而言的。

表 A-1. UML 属性/方法的可见性符号列表

符　号	可 见 性	对应的 Java 语言可见性关键字
+	Public	public
#	Protected	protected
-	Private	private
~	Package	package

例如，图 A-1 中的属性声明"-capacity: int = 1000"表示属性 capacity 的可见性为 private，其值类型为 int，初始值为 1000。

矩形框的第 3 个方格表示类的方法列表。这个方格往往只列出相应类的关键方法，而非全部方法。例如，类的私有方法除非有特别的需要，否则通常不列出来。方法列表中的每个方法的声明格式如下：

[方法的可见性]方法名([参数列表])[:返回值类型]

其中，方法名使用斜体表示，说明相应的方法为抽象（abstract）方法。方法的可见性及其表示符号如表 A-1 所示。参数列表的个数如下：

[(参数名 1: 参数 1 类型)][,(参数名 2: 参数 2 类型)][,(参数名 3: 参数 3 类型)]...

例如，图 A-1 中的方法声明"+put(element:T): void"表示方法 put 的可见性为 public，其返回值类型为 void（即没有返回值）。该方法仅包含一个类型为 T（对于图 A-1 中的类而言，T 是相应类的类型参数）的参数 element。

综上所述，图 A-1 表示的队列可以转化为相应的 Java 代码（骨架），如清单 A-1 所示。

清单 A-1. 图 A-1 所建模的类对应的 Java 代码（骨架）

```
public class Queue<T> {

    private int capacity = 1000;
    private ReentrantLock takeLock;
    private ReentrantLock putLock;

    public T take() {
        T element = null;

        return element;

    }

    public void put(T element) {

    }
}
```

类图也可以用来表示接口。从 UML 图作图工具的角度来看，UML 中接口通常可以使用类名以斜体注明的类图来表示或者在类图的立体型中注明“interface”。

A.2.2 类与类之间的关系

UML 中可以表示的类和类之间的关系包括泛化（Generalization）、实现（Realization）、关联（Association）、依赖（Dependency）、组合（Composition）和聚集（Aggregate）。

泛化关系表示类（接口）与类（接口）之间的继承关系，即一个类（接口）是另外一个类（接口）的子类。它相当于 Java 语言层面的 extends 关键字所表达的语义。在 UML 中，泛化关系使用单箭头实线来表示，如图 A-2 所示。

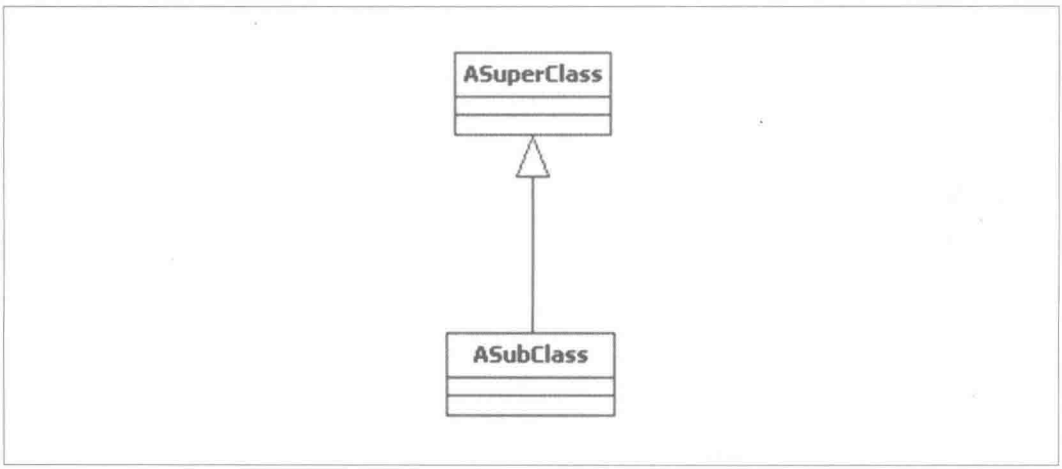


图 A-2. 类与类之间的泛化（继承）关系

图 A-2 表示类 ASubClass 继承自类 ASuperClass，即类 ASubClass 是类 ASuperClass 的一个子类。

实现关系用于表示某个类实现了哪些接口，或者说某个接口有哪些实现类。在 UML 中，实现关系使用单箭头虚线来表示，如图 A-3 所示。

图 A-3 表示类 LightSwtich（代表光控开关）实现了 Switch 接口（代表开关），或者说类 LightSwtich 是 Switch 接口的一个实现，相应的语义为光控开关是开关的一种具体实现。

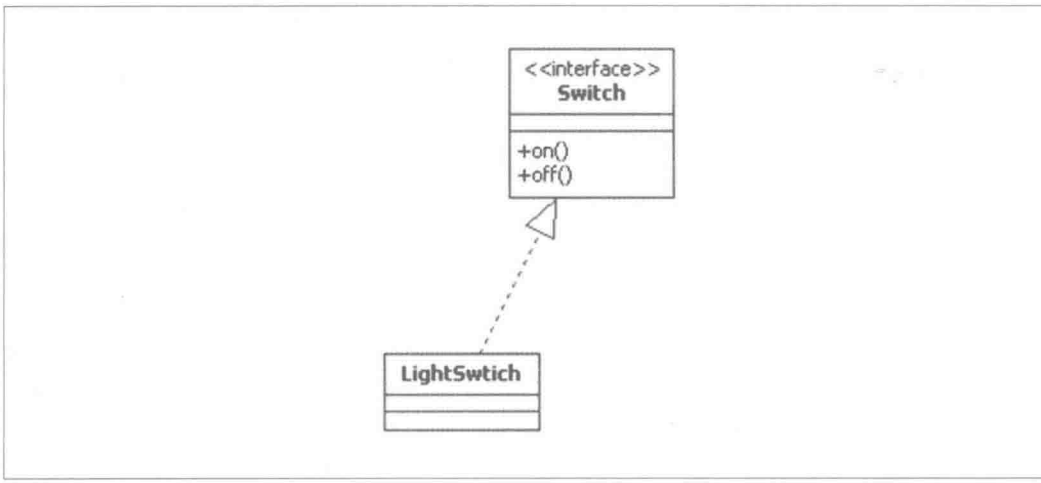


图 A-3. 类与接口之间的实现关系

类与类之间的关联关系从语义的角度可以理解为一个类拥有某个属性，该属性的类型为另外一个类，那么这两个类之间的关系即为关联关系。关联关系从代码的层次可以理解为一个类拥有某个实例变量，该变量的类型为另外一个类。UML 中，类和类之间的关联关系使用实线（连接线）来表示。例如，图 A-4 展示了类 Person（表示人）和类 MobilePhone（表示手机）之间的关联关系。这个关联关系从语义上说明一个人可以拥有多部手机（或者没有手机），而一部手机只属于一个人（机主）。如图 A-4 所示的两个类对应的代码（骨架）如清单 A-2 所示。

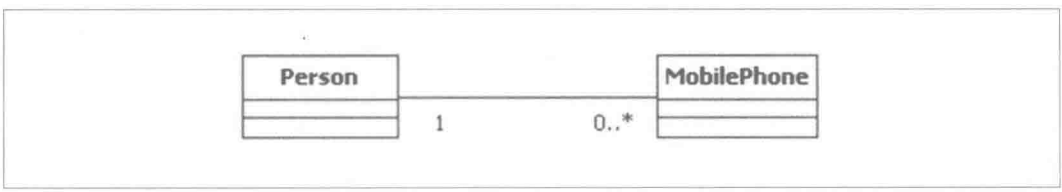


图 A-4. 类与类之间的关联关系

清单 A-2. 图 A-4 所建模的类对应的 Java 代码（骨架）

```
public class Person {
    private Set<MobilePhone> mobilePhones = new HashSet<MobilePhone>();

}

class MobilePhone {
    private Person owner = null;

}
```

表示类与类之间的关联关系的连接线的两端可以带有数字，用于表示相关类之间的数量关系（Multiplicity）。例如，一个人可以有多个住址（代表相应的建筑物），一个住址代表的建筑物可以住多个人。这种数量关系的声明格式为：

([下界][..上界])

例如，“0..*”表示任意个。

表示类与类之间的关联关系的连接线的两端还可以带有箭头。箭头表示可导航性（Navigability），即通过一个类可以访问到另外一个类的可能性。存在关联关系的两个类之间的可导航性可能是单向的也可能是双向的。例如，图 A-4 表示的两个类之间的连接线的两端都没有箭头，这可以理解为连接线所连接的两个类之间存在双向导航关系。就图 A-4 而言，这种双向导航关系表现为对于某个人（Person）我们可以知道他拥有多少部手机，对于某部手机而言我们可以知道它的机主是何人。

聚集（Aggregate）关系是一种特殊的关联关系，或者说它是一种更加具体的关联关系。它表示“有一个”（has a）的关系。在 UML 中，聚集关系使用一端带宝石符号的连接线来表示。例如，大学里的一个系（Department）可以拥有若干个教师（Teacher）。这里面的系与教师之间的关联关系可以用图 A-5 表示。

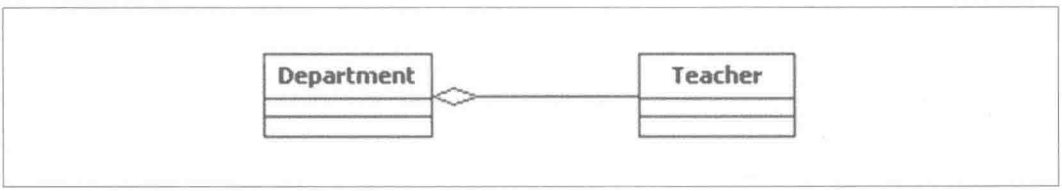


图 A-5. 类与类之间的聚集关系

注意到聚集关系也是一种关联关系这个事实，可以让我们明白上述适用于关联关系的一些概念（如数量关系）同样也适用于聚集关系。

组合（Composition）也是一种特殊的关联关系。在 UML 中，组合关系使用带实心宝石符号的箭头表示。组合关系是一种更为严格的聚集关系。它与聚集关系的区别在于具有这种关联关系的两个类的实例的生命周期的差别。在聚集关系中，一个类的实例的销毁不影响另外一个相关类的实例的存在。例如，大学里的一个系如果不存在了（如直接解散或者并到其他系中去了）并不影响这个系拥有的教师的存在情况（比如，这些教师可以到其他系去任教）。而组合关系则更为严格，它表示其中一个类的实例的销毁，会导致与该类有组合关系的类的实例的销毁。例如，一个文件夹（Folder）的删除意味着该文件夹下的所有文件（File）也被删

除，这种关系可以用图 A-6 表示。

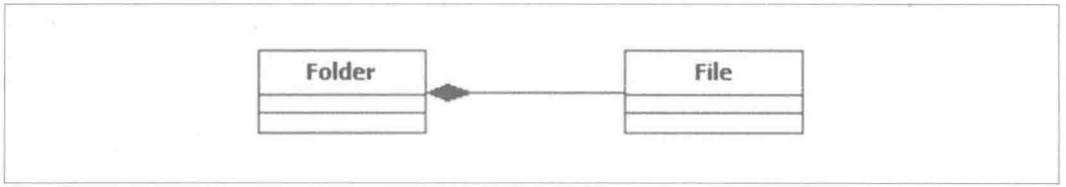


图 A-6. 类与类之间的组合关系

依赖（Dependency）关系是类与类之间的一种比较宽松的关系。这种关系通常用于表示两个类之间的除上述关系（泛化关系、实现关系和关联关系[3]）之外的关系。在 UML 中，依赖关系使用带箭头的虚线表示。图 A-7 展示了密码生成器（PasswordGenerator）和密码（Password）之间的依赖关系。这里，这种依赖关系表示类 PasswordGenerator 的 newPassword 方法会使用到类 Password。

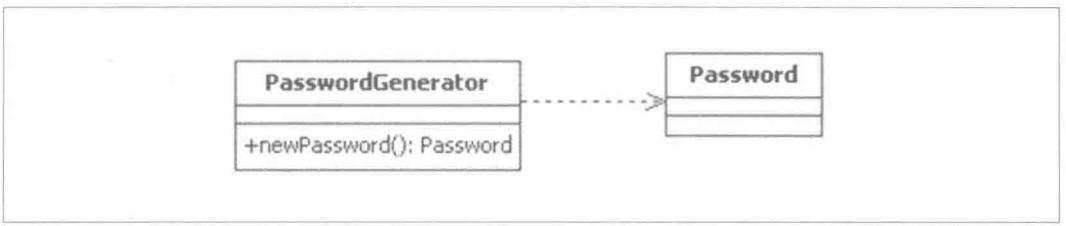


图 A-7. 类与类之间的依赖关系

A.3 序列图（Sequence Diagram）

序列图用于描述方法之间的调用关系，它是从动态的角度对系统进行描述。图 A-8 展示了一个使用 GPS（全球定位系统）客户端设备获取设备当前位置（Location）的过程所涉及的方法调用。

序列图中，方框表示对象。方框中冒号（:）前的文本表示对象标志，冒号后的文本表示类名。带箭头的实线表示方法调用。实线箭头指向的一方表示被调用的方法所属的类。带箭头的实线上方的文本表示被调用的方法（包括方法名、方法参数和返回值）。带箭头的虚线表示相应方法调用返回。序列图中，对象的创建可以使用特殊的方法调用来表示，即实线箭头上方的方法名为空。

3 聚集关系和组合关系都是一种特殊的关联关系。

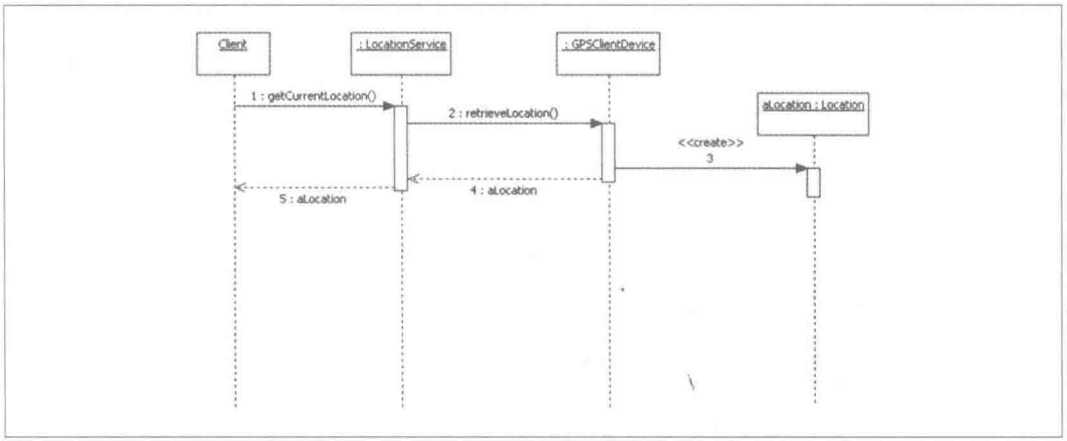


图 A-8. 序列图

图 A-8 描述的方法调用及其关系相应的代码（骨架）如清单 A-3 所示。

清单 A-3. 图 A-8 对应的 Java 代码（骨架）

```
public class LocationExample {

    public static void main(String[] args) {
        LocationService ls = new LocationService();
        Location location = ls.getCurrentLocation();

        // 其他代码
    }

}

class LocationService {
    private final GPSClientDevice gpsClient = new GPSClientDevice();

    public Location getCurrentLocation() {
        Location aLocation;
        aLocation = gpsClient.retrieveLocation();
        return aLocation;
    }
}

 class GPSClientDevice {
    public Location retrieveLocation(){
        Location aLocation = new Location();
        //其他代码
        return aLocation;
    }
}

class Location {
// 其他代码
}
```

参考文献

1．Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition.Wiley, 2002.

2．Schmidt, Douglas et al. Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

3．Brian Göetz et al. Java Concurrency In Practice. Addison Wesley,2006.

4．Erich Gamma 等. 设计模式：可复用面向对象软件的基础（英文版）. 机械工业出版社，2002.

5．Doug Lea. Concurrent Programming in Java: Design Principles and Patterns, Second Edition. Addison Wesley, 1999.