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

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 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。

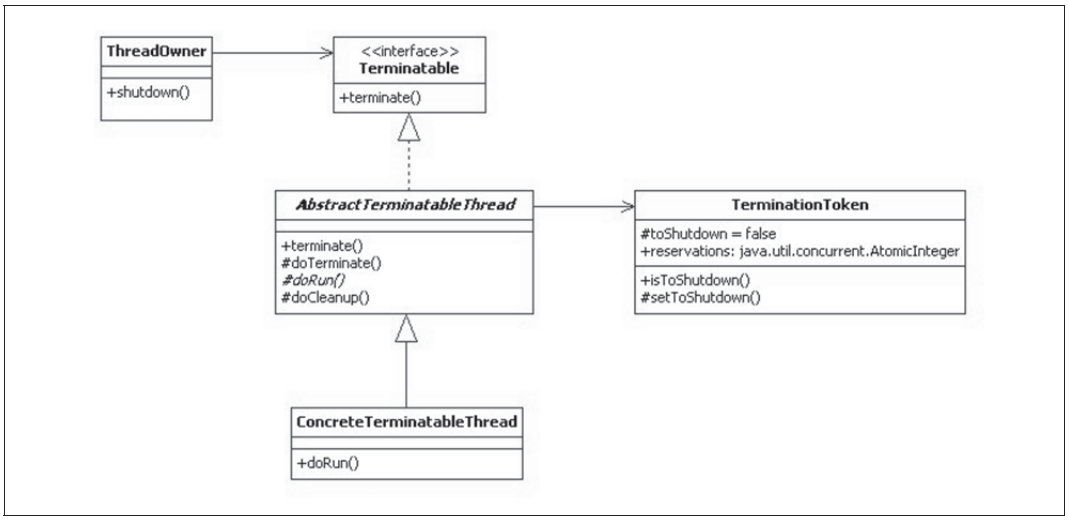

图 5-1. Two-phase Termination 模式的类图

- **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。

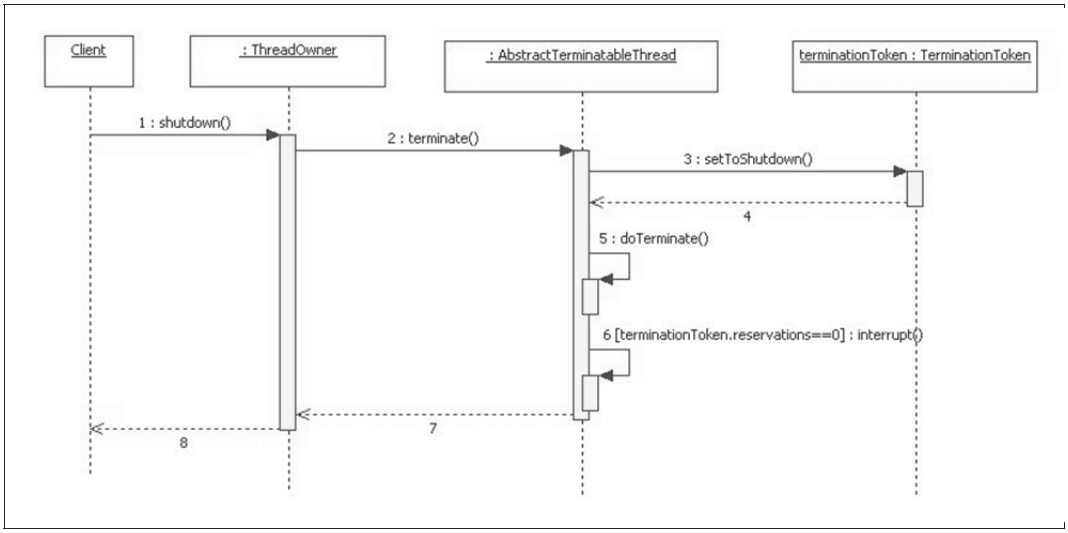

图 5-2. 准备阶段的序列图

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 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。

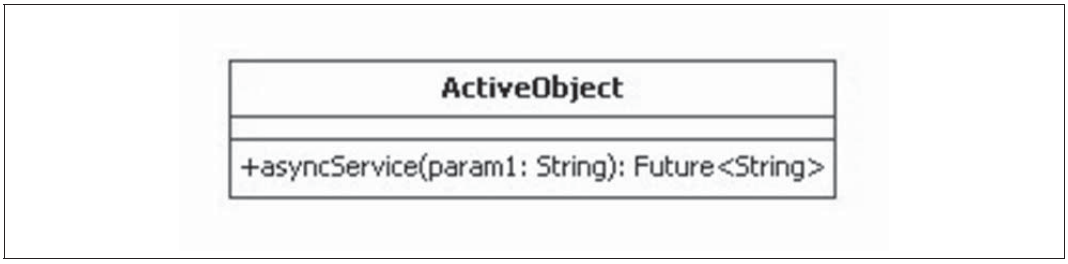

图 8-1. ActiveObject 对象示例

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。

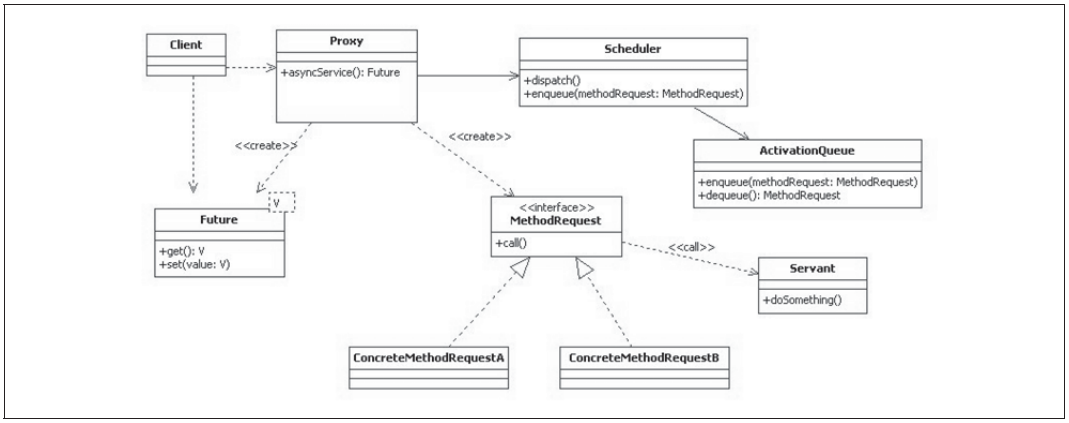

图 8-2. Active Object 模式的类图

- **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。

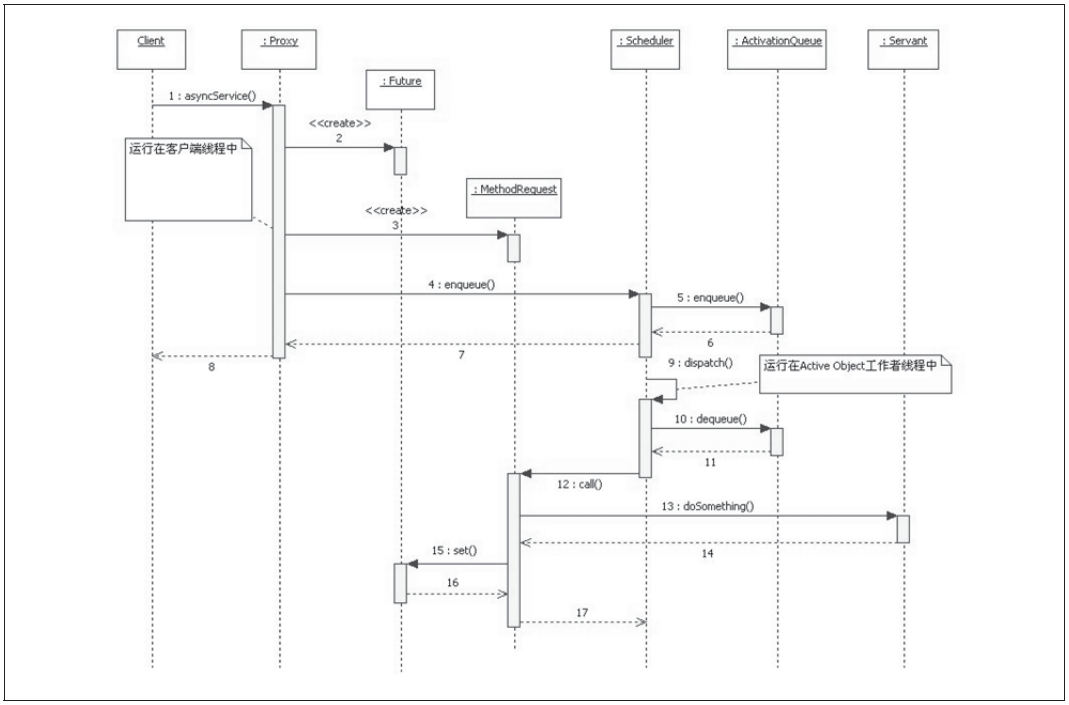

图 8-3. Active Object 模式的序列图

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。

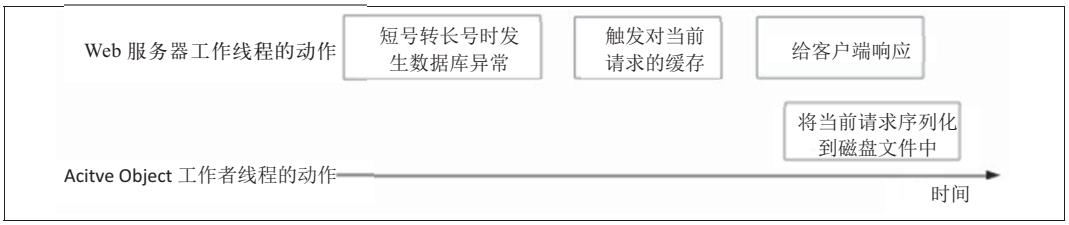

图 8-4. 异步实现缓存

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.