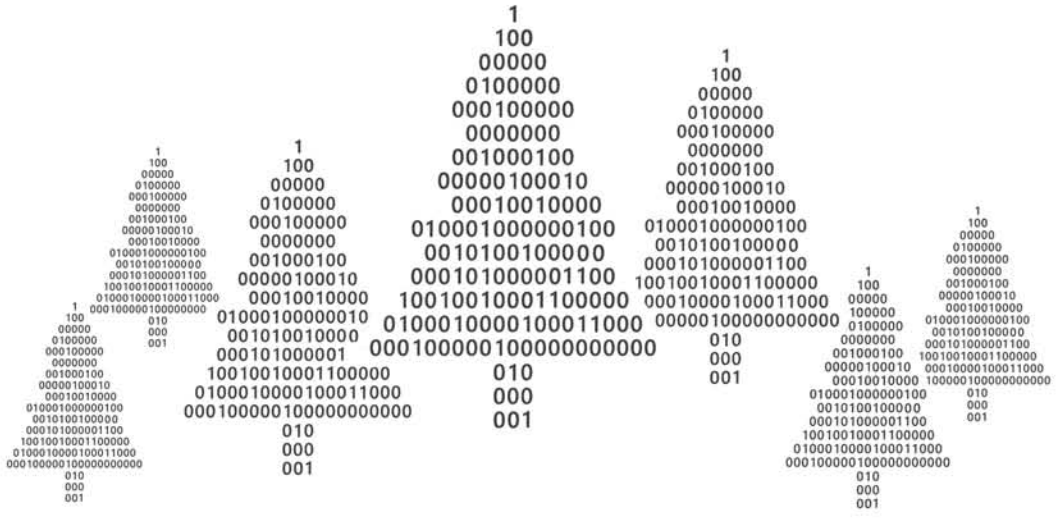
TURING
图灵程序
设计丛书

图解Java多线程设计模式

【日】结城浩 著　侯振龙 杨文轩 译



人民邮电出版社
北　京

图书在版编目（CIP）数据

图解 Java 多线程设计模式 /（日）结城浩著；侯振龙，杨文轩译 . -- 北京：人民邮电出版社，2017.8
（图灵程序设计丛书）
ISBN 978-7-115-46274-9

Ⅰ . ①图… Ⅱ . ①结… ②侯… ③杨… Ⅲ . ① JAVA 语言 – 程序设计 – 图解 Ⅳ . ① TP312.8-64

中国版本图书馆 CIP 数据核字（2017）第 170408 号

内 容 提 要

本书通过具体的 Java 程序，以浅显易懂的语言逐一说明了多线程和并发处理中常用的 12 种设计模式。内容涉及线程的基础知识、线程的启动与终止、线程间的互斥处理与协作、线程的有效应用、线程的数量管理以及性能优化的注意事项等。此外，还介绍了一些多线程编程时容易出现的失误，以及多线程程序的阅读技巧等。在讲解过程中，不仅以图配文，理论结合实例，而且提供了运用模式解决具体问题的练习题和答案，帮助读者加深对多线程和并发处理的理解，并掌握其使用技巧。本书适合对多线程、Java 编程、设计模式以及面向对象开发感兴趣的读者阅读。

◆ 著　　[日] 结城浩
译　　侯振龙　杨文轩
责任编辑　杜晓静
执行编辑　高宇涵　侯秀娟
责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷
◆ 开本：787×1092　1/16
印张：33
字数：944 千字　　2017 年 8 月第 1 版
印数：1–4 000 册　　2017 年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-3943 号

定价：89.00 元

读者服务热线：(010)51095186 转 600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

译者序

提起多线程编程，恐怕许多开发人员都会摇头表示不懂。确实，在校生和刚就职的开发人员往往很少有机会能够实践多线程编程。多数情况下，他们都是在开发框架下编写单线程的业务代码，而多线程的部分则被封装在了框架内部。即使是经验丰富的开发人员也会感叹他们曾经在多线程上栽过的跟头。但不可否认的是，多线程的确是一把利器，活用多线程有助于提高程序的响应性和吞吐量。可以毫不夸张地说，多线程是开发人员在继续“升级”的过程中必须打倒的一只“怪物”。

“设计模式”一词也常常会让开发人员感到畏惧。其实设计模式不过是对代码设计方式的总结和归纳。在我们的代码中，设计模式无处不在，只是我们没有注意到它们而已。善用设计模式可以帮助我们编写出具有高可复用性且松耦合的代码。

那么，将“多线程”与“设计模式”这两个主题放在一起的这本书，恐怕书名就会让许多读者望而却步吧。但是软件开发就是这么一件有趣的事情——随着我们心中的恐惧与日俱增，想要试着挑战的心情也会越来越迫切。

下面就让我们看看这本书都讲了哪些内容吧。

本书整理了 12 种常用的多线程设计模式，“图、文、码”并茂地讲解了它们各自的优缺点、相互的关联以及适用场景。不过这并不表示本书只适合已经掌握多线程编程的开发人员阅读和参考，因为作者还在讲解各种设计模式的过程中体贴地为初学者穿插介绍了多线程的基本知识。相信无论是新手还是老鸟，都能在阅读本书的过程中有所收获。此外，除第 13 章外，本书每章末尾都配有练习题，读者可以通过做题检验自己是否掌握了各章的知识。

本书的另一大特点是在编写代码实现这 12 种设计模式的基础上，还讲解了如何使用 Java 并发包 `java.util.concurrent` 包去实现这些设计模式。`java.util.concurrent` 是自 J2SE 5.0 起加入的包，在实现并发时非常重要，是并发编程必须要掌握的知识点。

另外，请一定不要忘记学习本书附录 B、附录 C 和附录 D 中介绍的知识哦。

相信读者如果掌握本书中的知识和设计模式，再去理解框架代码或是编写 Swing 程序时一定会得心应手。

本书的出版，要感谢合作译者侯振龙以及图灵公司的高编辑和侯编辑。

最后祝大家都能乐享多线程编程！

杨文轩

2017 年 6 月

引言

大家好，我是结城浩。欢迎阅读《图解 Java 多线程设计模式》。

这是一本讲解 Java 多线程及并发处理模式的入门书。

如果我们在程序中巧妙地利用多线程，便能够并发执行多个处理；在 GUI 应用程序中巧妙地利用多线程，便能够提高对用户的响应性；在服务器上的应用程序中巧妙地利用多线程，便能够并发处理多个用户的请求。多线程是重要的编程技术之一。Java 语言从一开始就加入了多线程功能，所以非常便于初学者学习多线程编程。

一些在单线程程序中并不会发生的 Bug 却会在多线程程序中发生，例如数据可能会损坏，程序可能会发生死锁而无法运行。另外，相对于单线程，多线程可能会占用更多的资源。同时多线程程序中发生的 Bug 很难调试，甚至连 Bug 现象的重现都会非常困难。多线程程序的性能优化也是一项非常难的课题。由此可见，相比单线程编程，多线程编程需要注意的地方更多，因此我们在编程时不能随心所欲，而要采用常用的模式。

本书将通过具体的 Java 程序，逐章介绍多线程编程中常用的模式。首先介绍线程的基础知识，随后介绍线程的启动与终止、线程间的互斥处理与协作、线程的有效应用、线程数量的管理、性能优化的注意事项等。此外，本书还将介绍一些多线程编程时容易出现的失误及多线程程序的阅读技巧等。

自 J2SE 5.0 开始，Java 增加了易于多线程编程的类库——`java.util.concurrent` 包。大家在使用该包时，一定要充分理解 Java 多线程的相关内容，否则将无法充分理解该包提供的类的优势，或者会出现使用的类与自己的目的不匹配的危险情况。本书中随处可见使用 `java.util.concurrent` 包时的注意事项或建议，请务必正确地使用该包。

希望读者朋友们能够通过本书加深对多线程及并发处理相关内容的理解，并掌握其使用技巧。

本书的特点

◆多线程模式的讲解

本书第 1 章 ~ 第 12 章的每一章都会讲解一种多线程及并发处理的模式。另外，每章并不仅仅介绍各个模式的内容，还会讲解各个模式相关的 Java 语言功能，从而加深大家对 Java 语言的理解。

◆ Java 语言的示例程序

本书介绍的所有模式都配有具体的 Java 示例程序。为了便于大家通读整个程序，绝大多数示例程序都很短。另外，各示例程序中并无省略内容，每个程序都可单独编译并执行。

◆模式名称的讲解

模式名称均采用英文表述。本书还讲解了各模式名称的英文读法、含义及中文表述。因此，英文不好的人也可以很容易地记住各个模式并理解它们的内容。

◆练习题

除第 13 章外，每章末尾都配有练习题。为了扩展各章所学内容，以及了解这些模式在实际开

发中的应用，请大家一定要做一下这些练习题。附录 A 提供了所有练习题的答案，自学者也可以轻松学习。

◆主要的 API 文档

附录 D 总结了一些线程相关的主要 API 文档。读者在阅读本书时，可以根据需要自行查阅，也可以整体阅读，以复习相关内容。

◆ java.util.concurrent 包的介绍

J2SE 5.0 的标准库中增加了易于多线程编程的 `java.util.concurrent` 包。在本书各章中，笔者会结合示例程序，讲解 `java.util.concurrent` 包中一些主要的类的用法。另外，附录 E 将全面介绍 `java.util.concurrent` 包。

本书的读者

本书适合以下读者阅读。

- 对多线程感兴趣的人
- 对 Java 编程感兴趣的人
- 对设计模式感兴趣的人
- 对面向对象开发感兴趣的人

阅读本书需具备 Java 语言的基础知识。具体而言，需要能够理解类、实例、字段及方法等，并能够独自编译和运行书中提供的 Java 代码。

虽然本书讲解的是设计模式，但必要时也会对 Java 语言进行讲解，因此读者还可以在阅读本书的过程中加深对 Java 语言的理解。即使是对不怎么了解 Java 多线程的读者而言，本书也具有很大的参考价值。

对于 Java 语言零基础的读者来说，在阅读本书之前，可以先阅读笔者之前出版的《Java 语言编程教程（上・下）第 2 次修订版》[①]（见附录 G 中的 [Yuki05]）。

另外，对于想从零开始学习设计模式的读者来说，在阅读本书之前，可以先阅读笔者之前出版的《图解设计模式》[②]（见附录 G 中的 [Yuki04]）。

本书的结构

本书第 1 章 ~ 第 12 章的每一章都会讲解一种设计模式，这些设计模式都是笔者基于如下两个原则从附录 G 中的参考文献里的设计模式之中挑选出来的。

- 与多线程和并发处理相关的设计模式
- 实际编程中常用的设计模式

序章 1 “Java 线程”将通过运行一个小程序，来介绍 Java 多线程的基础知识。

① 原书名为『改訂第 2 版　Java 言語プログラミングレッスン（上・下）』，尚无中文版。——译者注

② 杨文轩译，人民邮电出版社，2017 年 1 月。——译者注

序章 2“多线程程序的评价标准”将整理多线程程序的评价标准。

第 1 章“Single Threaded Execution 模式——能通过这座桥的只有一个人”将介绍多线程编程中最基础的一种设计模式——Single Threaded Execution 模式。该模式可以确保执行处理的线程只能是一个，这样就可以有效防止实例不一致。本章还将深入介绍 Java 语言的 `synchronized` 关键字，并给出计数信号量 `java.util.concurrent.Semaphore` 的示例程序。

第 2 章“Immutable 模式——想破坏也破坏不了”将介绍 Immutable 模式，即实例一旦创建完毕，其内容便不可更改的模式。在该模式下，由于实例不会不一致，所以无需执行互斥处理，程序性能也能提高。本章还将讲述 Java 语言中 `final` 的含义，并给出 `Collections.synchronizedList` 及 `java.util.concurrent.CopyOnWriteArrayList` 的示例程序。

第 3 章“Guarded Suspension 模式——等我准备好哦”将介绍 Guarded Suspension 模式，即在实例进入目标状态之前，防止线程继续执行的模式。该模式也可以防止实例不一致。通过本章还可以练习 Java 语言中的 `wait` 方法和 `notifyAll` 方法的使用。本章还将给出阻塞队列 `java.util.concurrent.LinkedBlockingQueue` 的示例程序。

第 4 章“Balking 模式——不需要就算了”将介绍 Balking 模式，即如果实例未进入目标状态，则中断方法执行的模式。该模式可防止执行无效的等待和多余的方法。

第 5 章“Producer–Consumer 模式——我来做，你来用”将介绍 Producer-Consumer 模式。在该模式下，多个线程能够协调运行。采用该模式时，生成数据的线程与使用数据的线程在并发运行时不会互相抢占。本章还将给出阻塞队列 `java.util.concurrent.ArrayBlockingQueue` 的示例程序。

第 6 章“Read–Write Lock 模式——大家一起读没问题，但读的时候不要写哦”将介绍 Read-Write Lock 模式，该模式会采用灵活的互斥处理。在该模式下，写数据的线程只能有一个，但读数据的线程可以有很多。该模式能够提高程序的整体性能。本章还将给出可重入的 `java.util.concurrent.locks.ReentrantReadWriteLock` 的示例程序。

第 7 章“Thread–Per–Message 模式——这项工作就交给你了”将介绍 Thread-Per-Message 模式，即将处理委托给其他线程的模式。在该模式下，线程可以将任务委托给其他线程，自己则直接处理接下来的工作。该模式能够提高程序的响应性。本章还将介绍 Java 语言中内部类的使用方法，并给出 `java.util.concurrent` 包中 `Executor` 和 `ExecutorService` 的示例程序。

第 8 章“Worker Thread 模式——工作没来就一直等，工作来了就干活”将介绍 Worker Thread 模式，即多个线程通过线程池进行等待，然后按照顺序接受工作并执行的模式。该模式可减少创建线程时的资源消耗，还可以通过调节等待线程的个数来控制可用的资源量。本章还将介绍 AWT 及 Swing（JFC）的线程处理方法，并给出通过 `java.util.concurrent` 包来使用线程池的示例程序。

第 9 章“Future 模式——先给您提货单”将介绍 Future 模式。在该模式下，可以同步获取交给其他线程的任务的结果。该模式适用于调用异步方法的情况。另外，本章还将给出 `java.util.concurrent.Future`、`FutureTask` 及 `Callable` 的示例程序。

第 10 章“Two–Phase Termination 模式——先收拾房间再睡觉”将介绍用于终止线程的 Two-Phase Termination 模式。该模式能够采用合适的终止处理来安全地终止线程。本章还将介绍线程的中断处理，并给出 `java.util.concurrent` 包中 `CountDownLatch`、`CyclicBarrier` 的示例程序。

第 11 章“Thread–Specific Storage 模式——一个线程一个储物柜”将介绍 Thread-Specific

Storage 模式。在该模式下，每个线程都会拥有自己的变量空间。采用该模式时，多个线程之间的变量空间是完全分离的，所以并不需要执行互斥处理。本章还将介绍 `java.lang.ThreadLocal` 类的使用方法。

第 12 章“Active Object 模式——接收异步消息的主动对象”将介绍 Active Object 模式。在该模式下，程序会创建主动对象。该主动对象将接收外部消息，并交由自己的线程来处理。采用该模式时，方法调用和方法执行是彼此分开的。本章还将给出使用了 `java.util.concurrent` 包中的类的示例程序。

第 13 章“总结——多线程编程的模式语言”将采用模式语言的形式归纳本书所介绍的 12 种模式之间的关系。

本书中的示例程序

支持的版本

本书中的示例程序都是基于 Windows 版的 J2SE 5.0（JDK 1.5.0）[①] 编写的，并已在 Windows XP 上进行了确认。

示例程序的获取方法

本书的示例程序可以从以下网址下载（点击“随书下载”）。

`http://www.ituring.com.cn/book/1812`

关于 Main 类

在 Java 中，只要类中定义了以下方法，那么无论该类取什么名字，都可以将其作为程序的起点。

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

但是，为了便于读者理解代码，本书各章的示例程序都是使用 `Main` 类作为程序的起点的。

关于本书中术语的注意事项

接口和 API

接口这个术语有多个意思。

一般而言，在提到“某个类的接口”时，多是指该类持有的方法的集合。当想要对该类执行某些操作和处理时，需要调用这些方法。

但是在 Java 中，也将“使用关键字 `interface` 声明的代码”称为接口。

这两个“接口”的意思有些相似，在使用时容易混淆。因此本书中采用以下方式加以区分。

① 在 J2SE 5.0 以后的环境中，编译和运行结果可能与本书不同。——译者注

- 接口（API）：通常的意思（API 是 Application Programming Interface 的缩写）
- 接口：使用关键字 `interface` 声明的代码

角色

角色是本书中特有的说法。它是指模式（设计模式）中出现的类、接口和实例在该模式中所起的作用。例如，书中会有“由 `MakerThread` 类扮演 Producer 角色”这种表述。请注意，角色的名字与类和接口的名字不一定相同。

致谢

感谢《Java 并发编程：设计原则与模式（第二版）》[①]（见附录 G 中的 [Lea]）一书的作者 Doug Lea，他的著作给了笔者很大帮助。在本书的写作过程中，笔者通过邮件咨询问题时，他都给予了莫大的支持。对此笔者深表感谢。

感谢设计模式邮件列表[②]中的各位参加者。

然后，还要向阅读笔者拙作，包括图书、连载杂志和电子邮件杂志的读者们表示感谢。另外，还要向笔者 Web 主页上的朋友们表示感谢。

笔者在编写本书的原稿、程序以及图示的过程中，还将它们公布在了互联网上，以供大家评审。在互联网上招募的评审人员不限年龄、国籍、性别、住址、职业，所有交流都是通过电子邮件和网络进行的。在此，笔者要向参与本书评审的朋友们表示感谢，特别是对给予了笔者宝贵意见、改进方案，向笔者反馈错误以及一直鼓励笔者的以下各位表示最真挚的感谢（按五十音图排序）：

新真千惠、天野胜、石井胜、石川草子、井芹义博、植田训弘、植松喜孝、宇田川胜俊、宇野敦之、胡田昌彦、大内宽和、大谷晋平、绪方彰、冈庭祐、奥野皓市、小田浩之、大根田雄一、片冈孝浩、镰田淳、萱森孝、神崎雄一郎、木村明治、清田信行、黑川裕之、小松慎一、酒井敦、榊原知香子、坂本善隆、贞池克己、佐藤贵行、佐藤正明、佐山秀晃、泽田大辅、式见彰浩、清水顺、清水宏行、城生贵幸、助田雅纪、铃木健司、高江洲睦、高岛修、高津修一、高野兼一、高桥武士、高安厚思、土居俊彦、中井健介、中岛雷太、中林俊晴、西海秀俊、平田守幸、藤田宗典、藤田幸久、藤山博人、古川洋介、前原正英、松冈正恭、松本成道、三宅喜义、宫本信二、村田贤一郎、茂木正治、八木希仁、山本耕司、山本正和、丁农、吉田慎太郎、鹭崎弘宜。

此外，对其他参与了评审工作的人员也一并表示感谢。

另外，还要向软银出版股份有限公司图书总编野泽喜美男和编辑松本香织表示感谢。

最后要感谢笔者最爱的妻子和两个健康活泼的儿子。跟上你们的步伐简直比多线程的调试还要难啊。

本书献给在笔者上学时教会笔者如何选购参考书的姐姐。

“在书店看到好的参考书，就要赶紧买下来，要是卖出去不就买不到了吗?”

您的这句话简直就是在说这本书嘛。

结城浩

2002 年 6 月 于横滨

① 赵涌等译，中国电力出版社，2004 年 2 月。——编者注

② 亦称“邮件清单”，包含许多接收者地址的一个电子邮件列表之中，主要用来进行信息发布。这里提到的邮件列表地址为 http://www.hyuki.com/dp/dpml.html，现已停止活动。——编者注

写于“修订版”前

《图解 Java 多线程设计模式》拥有众多读者，笔者深感荣幸，在此再次向各位读者表示最真挚的感谢。

在这次修订中，笔者重新全面地审视了本书的内容和表述，还基于 J2SE 5.0 修改了示例程序，并新增了支持 `java.util.concurrent` 的示例程序。本次修订还参考了读者朋友们发送给笔者的无数反馈意见和建议，真心谢谢你们。

感谢对此次修订工作提供支持的软银出版股份有限公司的总编野泽喜美男和编辑中岛绫子。

希望本书也能在读者朋友的工作和学习中发挥些许作用。

结城浩

2006 年 3 月

关于 UML

UML

UML 是将系统可视化、让规格和设计文档化的表现方法，它是 Unified Modeling Language（统一建模语言）的简称。

本书使用 UML 来描述设计模式中的类和实例的关系，所以我们在这里先稍微了解一下 UML，以方便后面的阅读。但是请大家注意，在说明中我们使用的是 Java 语言的术语。例如，讲解时我们会用 Java 中的“字段”（field）取代 UML 中的“属性”（attribute），用 Java 中的“方法”取代 UML 中的“操作”（operation）。

UML 标准的内容非常多，这里我们只对书中使用到的 UML 内容进行讲解。如果想了解更多 UML 内容，请访问以下网站。UML 的规范书也可以从以下网站下载。

- UML Resource Page

  `http://www.omg.org`

- UML Resource Center

  `http://www-306.ibm.com/software/rational/uml`

- UML 技术资料

  `https://www-01.ibm.com/software/cn/rational/index.html`

类图

UML 中的类图（Class Diagram）用于表示类、接口、实例等之间相互的静态关系。虽然名字叫作类图，但是图中并不只有类。

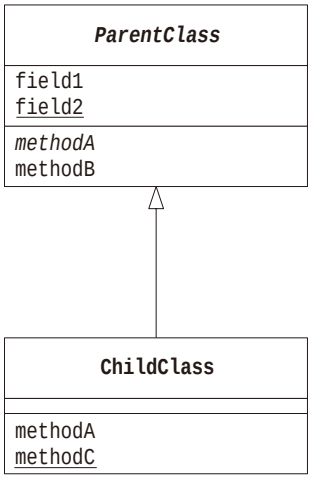
类与层次关系

图 0-1 展示了一段 Java 程序及其对应的类图。

图 0-1 展示类的层次关系的类图

```
abstract class ParentClass {
    int field1;
    static char field2;
    abstract void methodA();
    double methodB() {
        // ...
    }
}

class ChildClass extends ParentClass {
    void methodA() {
        // ...
    }
    static void methodC() {
        // ...
    }
}
```



该图展示了 ParentClass 和 ChildClass 这两个类之间的关系，其中的空心实线箭头表明了两者之间的层次关系，箭头由子类指向父类，换言之，这是表示继承（extends）的箭头。

ParentClass 是 ChildClass 的父类，反过来说，ChildClass 是 ParentClass 的子类。父类也称为基类或超类，子类也称为派生类或继承类。

图中的长方形表示类，长方形内部被横线自上而下分为了如下三个区域。

- 类名
- 字段名
- 方法名

有时，图中除了会写出类名、字段名和方法名等信息外，还会写出其他信息（可见性、方法的参数和类型等）。反之，有时图中也会省略所有不必关注的内容（因此，我们无法确保一定可以根据类图生成源程序）。

abstract 类（抽象类）的名字以斜体方式显示。例如，在图 0-1 中，ParentClass 是抽象类，因此它的名字以斜体方式显示。

static 字段（静态字段）的名字带有下划线。例如，在图 0-1 中，field2 是静态字段，因此它的名字带有下划线。

abstract 方法（抽象方法）的名字以斜体方式显示。例如，在图 0-1 中，ParentClass 类的 methodA 是抽象方法，因此它以斜体方式显示。

static 方法（静态方法）的名字带有下划线。例如，在图 0-1 中 ChildClass 类的 methodC 是静态方法，因此它的名字带有下划线。

▶▶ 小知识：Java 术语与 C++ 术语

Java 术语跟 C++ 术语略有不同。Java 中的字段相当于 C++ 中的成员变量，而 Java 中的方法相当于 C++ 中的成员函数。

▶▶ 小知识：箭头的方向

UML 中规定的箭头方向是从子类指向父类。可能会有人认为子类是以父类为基础的，箭头从父类指向子类会更合理。

关于这一点，按照以下方法去理解有助于大家记住这条规则。在定义子类时需要通过 `extends` 关键字指定父类。因此，子类一定知道父类的定义，而反过来，父类并不知道子类的定义。只有在知道对方的信息时才能指向对方，因此箭头方向是从子类指向父类。

接口与实现

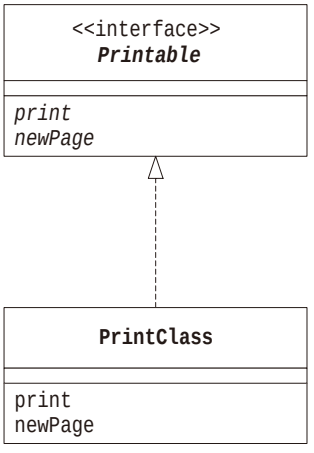
图 0-2 也是类图的示例。该图表示 `PrintClass` 类实现了 `Printable` 接口，其中接口名称为斜体。带有空心三角的虚线箭头代表了接口与实现类的关系，箭头从实现类指向接口。换言之，这是表示实现（`implements`）的箭头。

UML 以 `<<interface>>` 表示 Java 的接口。

图 0-2　展示接口与实现类的类图

```
interface Printable {
    abstract void print();
    abstract void newPage();
}

class PrintClass implements Printable {
    void print() {
        // ...
    }
    void newPage() {
        // ...
    }
}
```



聚合

图 0-3 也是类图的示例。

该图展示了 `Color`（颜色）、`Fruite`（水果）和 `Basket`（果篮）这三个类之间的关系。`Basket` 类中的 `fruites` 字段是可以存放 `Fruite` 类型数据的数组，在一个 `Basket` 类的实例中可以持有多个 `Fruite` 类的实例；`Fruite` 类中的 `color` 字段是 `Color` 类型，一个 `Fruite` 类的实例中只能有一个 `Color` 类的实例。通俗地说就是在篮子中可以放入多个水果，每个水果都有其自身的颜色。

我们将这种“持有”关系称为聚合（aggregation）。只要在一个类中持有另外一个类的实例——无论是一个还是多个——它们之间就是聚合关系。就程序上而言，无论是使用数组、`java.util.ArrayList` 还是其他实现方式，只要在一个类中持有另外一个类的实例，它们之间就是聚合关系。

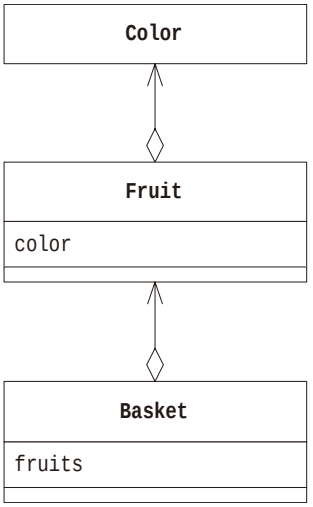
在 UML 中，我们使用带有空心菱形的实线表示聚合关系，因此可以进行联想记忆，将聚合关系想象为在菱形的器皿中装有其他物品。

图 0-3 展示聚合关系的类图

```
class Color {
    // ...
}

class Fruit {
    Color color;
    // ...
}

class Basket {
    Fruit[] fruits;
    // ...
}
```

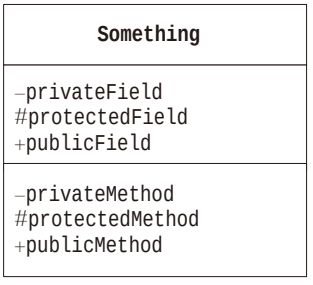


可见性（访问控制）

图 0-4 也是类图的示例。

图 0-4 标识出了可见性的类图

```
class Something {
    private   int privateField;
    protected int protectedField;
    public    int publicField;
    private   void privateMethod() {
    }
    protected void protectedMethod() {
    }
    public    void publicMethod() {
    }
}
```



该图标识出了方法和字段的可见性。在 UML 中可以通过在方法名和字段名前面加上记号来表示可见性。

“+”号表示 `public` 方法和字段，可以从类外部访问这些方法和字段。

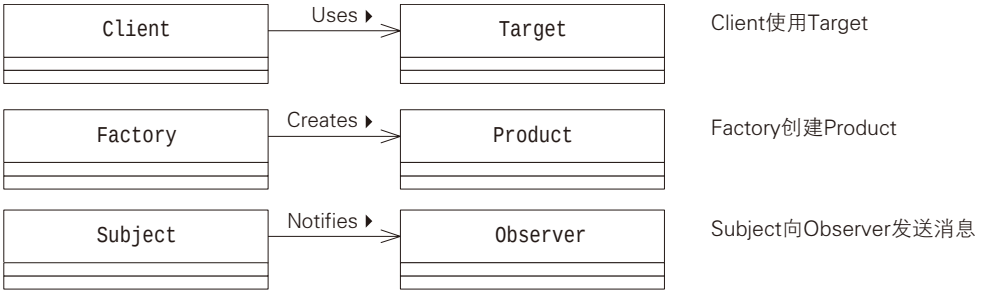
“–”号表示 `private` 方法和字段，无法从类外部访问这些方法和字段。

“#”号表示 `protected` 方法和字段，能够访问这些方法和字段的只能是该类自身、该类的子类以及同一个包中的类。

类的关联

可以在类名前面加上黑三角（▶）表示类之间的关联关系，如图 0-5 所示。

图 0-5 类的关联



时序图

UML 的时序图（Sequence Diagram）用来表示在程序运行时，其内部方法的调用顺序，以及事件的发生顺序。

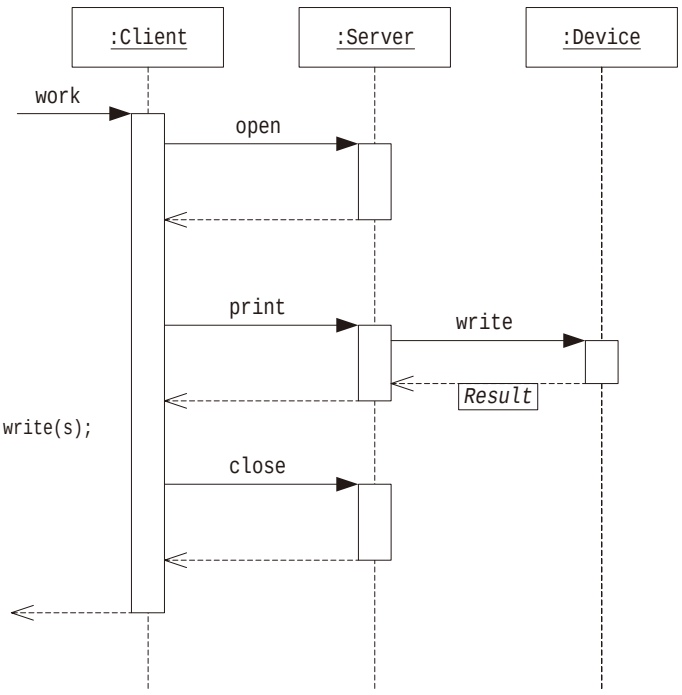
类图表示的是“不因时间流逝而发生变化的关系（静态关系）”，时序图则与之相反，表示的是“随时间流逝而发生变化的关系（动态行为）”。

处理流程与对象间的协作

图 0-6 展示的是时序图的一个例子。

图 0-6 时序图示例（方法的调用）

```
class Client {
    Server server;
    void work() {
        server.open();
        server.print("Hello");
        server.close();
    }
    // ...
}
class Server {
    Device device;
    void open() {
        // ...
    }
    void print(String s) {
        Result result = device.write(s);
        // ...
    }
    void close() {
        // ...
    }
    // ...
}
class Device {
    Result write(String s) {
        // ...
        return result;
    }
}
```



在图 0-6 中，右侧是时序图，左侧是与之对应的代码片段。

该图中共有三个实例，如图中最上方的三个长方形所示。在长方形内部写有类名，类名跟在冒号（：）之后，并带有下划线，如 `:Client`、`:Server`、`:Device`，它们分别代表 `Client` 类、`Server` 类、`Device` 类的实例。

如果需要，还可以在冒号（：）之前表示出实例名，如 `server:Server`。

每个实例都带有一条向下延伸的虚线，我们称其为生命线。这里可以理解为时间从上向下流逝，上面是过去，下面是未来。生命线仅存在于实例的生命周期内。

在生命线上，有一些细长的长方形，它们表示该对象处于某种活动中。

横方向上有许多箭头，请先看带有 `open` 字样的箭头。黑色实线箭头（—►）表示方法的调用，这里表示 `client` 调用 `server` 的 `open` 方法。当 `server` 的 `open` 方法被调用后，`server` 实例处于活动中，因此我们在 `open` 箭头处画出了一个细长的长方形来表示。

而在 `open` 箭头画出的长方形下方，还有一条指向 `client` 实例的虚线箭头（≪┈），它表示返回 `open` 方法。在上图中，我们画出了所有的返回箭头，但是有些时序图也会省略返回箭头。

由于程序控制权已经返回至 `client`，所以表示 `server` 实例处于活动状态的长方形就此结束了。

接着，`client` 实例会调用 `server` 实例的 `print` 方法。不过这次不同的是在 `print` 方法中，`server` 会调用 `device` 实例的 `write` 方法。

这样，我们就将多个实例之间的行为用图示的方式展示出来了。时序图的阅读顺序是沿着生命线从上至下阅读。然后当遇到箭头时，我们可以顺着箭头所指的方向查看实例间的协作。

另外，在本书中，如果返回值很重要，需要像 `Device` 类中 `write` 方法的返回值 `Result` 一样，在箭头下方加上一个长方形表示出来（此方式依据附录 G 中的 [POSA2]）。

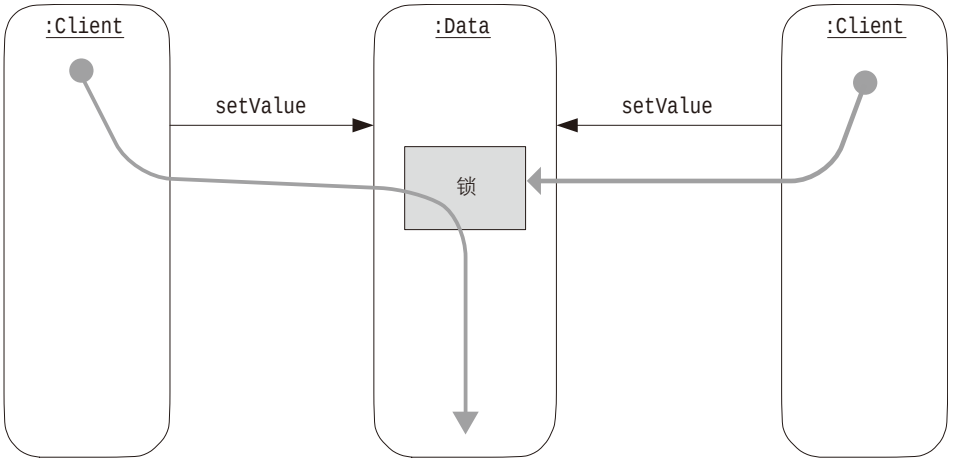
Timethreads 图

本书中，如果用时序图难以表示线程的运行状况，则会采用 Timethreads 图（Timethreads Diagram）来表示。Timethreads 图并不是 UML 中的标准概念，而是附录 G 中的 [Lea] 使用的表示方法。本书主要采用该方法，而对象名则采用 UML 中的表示方法。Timethreads 图能够将线程的运行可视化，易于理解。

图 0-7 是一个简单的 Timethreads 图的示例。在该示例中，有两个线程对 `Data` 类的实例调用了 `setValue` 方法。左侧的线程获取实例的锁并执行 `setValue` 方法，而右侧的线程则在试图获取该锁时陷入阻塞状态。

在 Timethreads 图中，实例用圆角长方形来表示。而 `:Data` 中的长方形表示实例持有的锁。关于锁及线程阻塞的内容，序章 1 将会详细讲解。

图 0-7 Timethreads 图示例



主　页

由 SoftBank Creative 出版的图书信息都发布在以下网站上，请读者自行查阅。

http://www.sbcr.jp/book

有关本书的最新内容，请参照如下网址。

http://www.hyuki.com/dp/dp2.html

这部分由作者本人维护。

目 录

序章 1

Java 线程

1.1 Java 线程

本章主要介绍 Java 线程的一些基础知识。如果读者已经熟悉这些内容，也可阅读一下以便检验自己的掌握程度。本章主要包含以下内容。

- 何谓线程

单线程和多线程、Thread 类、run 方法、start 方法

- 线程的启动

Thread 类、Runnable 接口

- 线程的暂停

sleep 方法

- 线程的互斥处理

synchronized 方法、synchronized 语句、锁

- 线程的协作

等待队列（wait set）、wait 方法、notify 方法、notifyAll 方法

其他内容会根据需要在各章中进行讲解。

另外，线程相关的一些主要 API 在附录 D 中。

1.2 何谓线程

明为跟踪处理流程，实为跟踪线程

阅读程序时，我们会按处理流程来阅读。

首先执行这条语句

↓

然后执行这条语句

↓

接着再执行这条语句……

我们就是按照上面这样的流程阅读程序的。

如果将程序打印出来，试着用笔将执行顺序描画出来，就会发现最终描画出来的是一条弯弯曲曲的长线。

这条长线始终都会是一条。无论是调用方法，还是执行 for 循环、if 条件分支语句，甚至更复杂的处理，都不会对这条长线产生影响。对于这种处理流程始终如一条线的程序，我们称之为单线程程序（single threaded program）。

在单线程程序中，“在某一时间点执行的处理”只有一个。如果有人问起“程序的哪部分正在执行”，我们能够指着程序中的某一处回答说“这里，就是这儿”。这是因为，在单线程程序中，“正在执行程序的主体”只有一个。

线程对应的英文单词 Thread 的本意就是“线”。Java 语言中将此处所说的“正在执行程序的主

体”称为线程[①]。我们在阅读程序时，表面看来是在跟踪程序的处理流程，实际上跟踪的是线程的执行。

单线程程序

这里我们先来执行一个简单的单线程程序。代码清单 I1-1 是一个显示 10 000 次 `Good!` 字符串的单线程程序。

代码清单 I1-1　单线程程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/SingleThread 中

如果你使用的是 Java Development Kit（JDK），请在命令行输入如下内容。

```
javac Main.java ↵
```

接下来，`javac` 命令便会编译源文件 Main.java，并生成类文件 Main.class。

然后，在命令行再输入如下内容。

```
java Main ↵
```

接下来，`java` 命令便会执行该程序，在屏幕上显示 10 000 个 `Good!`，如图 I1-1 所示。

编译与执行的操作方法请参照附录 F。

图 I1-1　运行结果

```
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
（以下省略）
```

Java 程序执行时，至少会有一个线程在运行。代码清单 I1-1 中运行的是被称为**主线程**（main thread）的线程，执行的操作是显示字符串。

在命令行输入如下内容，主线程便会在 Java 运行环境中启动。

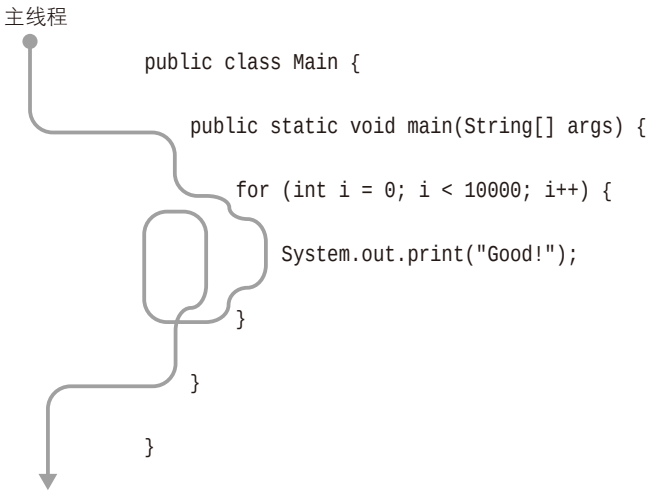
```
java 类名↵
```

然后，主线程会执行命令行中输入的类的 `main` 方法。`main` 方法中的所有处理都执行完后，主线程也就终止了（图 I1-2）。

代码清单 I1-1 中只有一个线程在运行，所以这是一个单线程程序。

① 有时也称为控制线程（thread of control）。

图 I1-2 单线程程序的运行情况



▶▶ 小知识：后台运行的线程

为了便于说明，前面的讲解说的是“只有一个线程在运行”。其实严格来讲，Java 处理的后台也有线程在运行。例如垃圾回收线程、GUI 相关线程等。

多线程程序

由多个线程组成的程序就称为**多线程程序**（multithreaded program）。Java 编程语言从一开始就把多线程处理列入编程规范了。

多个线程运行时，如果跟踪各个线程的运行轨迹，会发现其轨迹就像多条线交织在一起。

假设有人问起“程序的哪部分正在执行”，而我们需要指出程序位置，并回答“这里，就是这儿”。那么在多线程的情况下，一根手指根本不够用，这时需要和线程个数一样多的手指。也就是说，如果有两个线程在运行，那就需要指出两个地方并回答“第一个线程正在这里执行，第二个线程在那里执行”；如果有三个线程，就要指出三个地方；如果有一百个线程，就要指出一百个地方。

当规模大到一定程度时，应用程序中便会自然而然地出现某种形式的多线程。以下便是几种常见示例。

◆GUI 应用程序

几乎所有的 GUI 应用程序中都存在多线程处理。例如，假设用户在使用文本工具编辑较大的文本文件时执行了文字查找操作。那么当文本工具在执行查找时，屏幕上会出现“停止查找”按钮，用户可随时停止查找。此时就需要用到多线程。

（1）执行查找

（2）显示按钮，并在按钮被按下时停止查找

这两个操作是分别交给不同的线程来执行的。这样一来，（1）的操作线程专门执行查找，而（2）的操作线程则专门执行 GUI 操作，因此程序就会比较简单。

◆耗时的 I/O 处理

一般来说，文件与网络的 I/O 处理都非常消耗时间。如果在 I/O 处理期间，程序基本上无法执行其他处理，那么性能将会下降。在这种情况下，就可以使用多线程来解决。如果将执行 I/O 处理的线程和执行其他处理的线程分开，那么在 I/O 处理期间，其他处理也可以同时执行。

◆多个客户端

基本上，网络服务器都需要同时处理多个客户端。但是，如果让服务器端针对多个客户端执行处理，那么程序会变得异常复杂。这种情况下，在客户端连接到服务器时，我们会为该客户端准备一个线程。这样一来，服务器程序就被设计成了好像只处理一个客户端。具体示例将在第 7 章的习题 7-6 中再进行介绍。

▶▶小知识：兼具性能和可扩展性的 I/O 处理

`java.nio` 包中包含兼具性能和可扩展性的 I/O 处理。有了这个包，即便不使用线程，也可以执行兼具性能和可扩展性的 I/O 处理。具体内容请参见 API 文档。

Thread 类的 run 方法和 start 方法

接下来，我们试着编写一个多线程程序。Java 程序运行时，最开始运行的只能是主线程。所以，必须在程序中启动新线程，这才能算是多线程程序。

启动线程时，要使用如下类（一般称为 `Thread` 类）。

```
java.lang.Thread
```

我们来看一下代码清单 I1-2，即 `Thread` 的继承（`extends`）类 `MyThread`。

代码清单 I1-2　表示新线程的 MyThread 类（MyThread.java）

```
public class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Nice!");
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/TwoThreads 中

`MyThread` 类中声明了如下 `run` 方法。

```
public void run() {
    ...
}
```

该方法执行的处理是输出 10 000 次 `Nice!` 字符串。

新启动的线程的操作都编写在 `run` 方法中（run 就是“跑”的意思）。新线程启动后，会调用 `run` 方法。随后，当 `run` 方法执行结束时，线程也会跟着终止。`MyThread` 类中的 `run` 方法写得没有问题，但如果仅是这样，程序什么操作也不会做，所以必须新启动一个线程，调用 `run` 方法才可以。

用于启动线程的代码如代码清单 I1-3 所示。代码清单 I1-3 创建一个 MyThread 的实例，并利用该实例启动新的线程。然后，程序会再执行自身（主线程）的任务，输出 10 000 次 Good!。主线程主要执行如下两个任务。

- 启动输出 Nice! 操作的新线程
- 输出 Good!

代码清单 I1-3　用于启动新线程的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/TwoThreads 中

我们来看一下代码清单 I1-3。通过下面这行语句，主线程会创建 MyThread 类的实例，并将其赋给变量 t。

```
MyThread t = new MyThread();
```

下面这行语句则是由主线程启动新线程。

```
t.start();
```

start 方法是 Thread 类中的方法，用于启动新的线程。

在此需要注意的是，启动新线程时调用的是 start 方法，而不是 run 方法。当然 run 方法是可以调用的，但调用它并不会启动新的线程。

调用 start 方法后，程序会在后台启动新的线程。然后，由这个新线程调用 run 方法。start 方法主要执行以下操作。

- 启动新线程
- 调用 run 方法

start 方法与 run 方法之间的关系如图 I1-3 所示。图中出现了两条线（即图中的灰线）。

而代码清单 I1-2、代码清单 I1-3 的程序运行结果如图 I1-4 所示。

从图 I1-4 中我们可以发现 Good! 字符串和 Nice! 字符串是交织在一起输出的。由于这两个线程是并发运行的，所以结果会像图中这样混在一起。这两个线程负责的操作如下。

- 主线程输出 Good! 字符串
- 新启动的线程输出 Nice! 字符串

代码清单 I1-2、I1-3 的程序中运行着两个线程，所以这是一个多线程程序。

图 I1-3 启动新的线程（start 方法与 run 方法的关系）

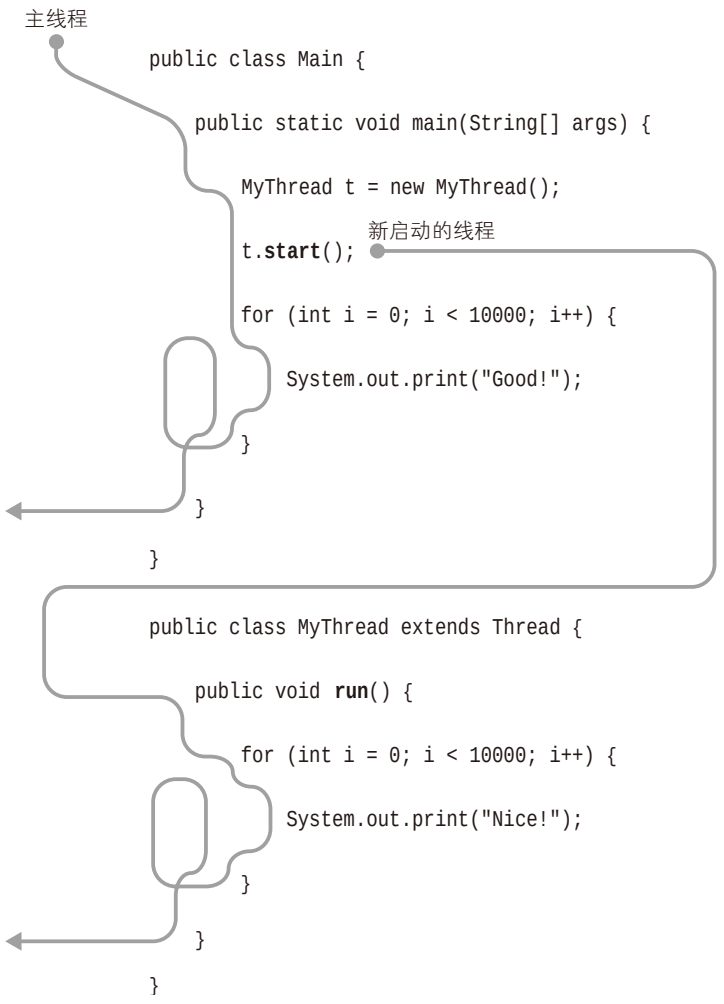


图 I1-4 运行结果示例（Good！ 和 Nice！ 交叉输出）

```
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!
Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!
Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!
Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!
Good!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!Nice!
（以下省略）
```

▶▶ 小知识：顺序、并行与并发

下面简单说明一下顺序、并行与并发这三个概念。

顺序（sequential）用于表示多个操作“依次处理”。比如把十个操作交给一个人处理时，这个人要一个一个地按顺序来处理。

并行（parallel）用于表示多个操作“同时处理”。比如十个操作分给两个人处理时，这两个人就会并行来处理。

并发（concurrent）相对于顺序和并行来说比较抽象，用于表示“将一个操作分割成多个部分并且允许无序处理”。比如将十个操作分成相对独立的两类，这样便能够开始并发处理了。如果一个人来处理，这个人就是顺序处理分开的并发操作，而如果是两个人，这两个人就可以并行处理同一个操作。

本书中的多线程程序都是并发处理的。如果 CPU 只有一个，那么并发处理就是顺序执行的，而如果有多个 CPU，那么并发处理就可能会并行运行。

我们使用的计算机通常情况下只有一个 CPU，所以即便多个线程同时运行，并发处理也只能顺序执行。比如“输出 `Good!` 字符串的线程”和“输出 `Nice!` 字符串的线程”这两个线程就是像下面这样运行的。

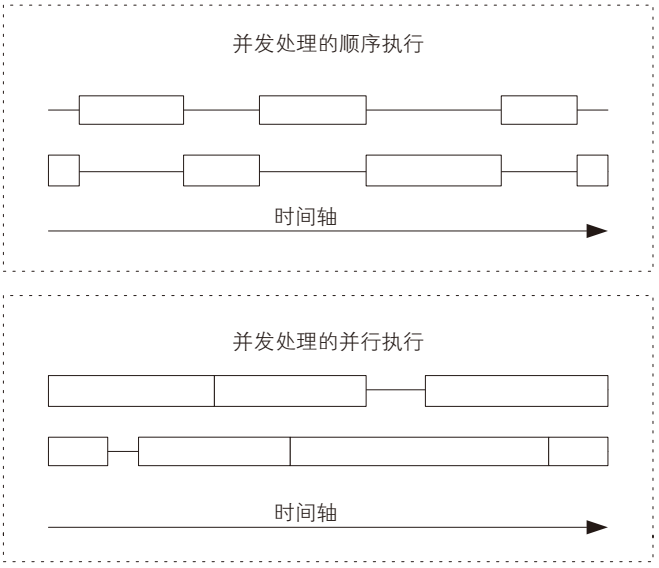
- 输出 `Good!` 字符串的线程稍微运行一下后就停止
  ↓
- 输出 `Nice!` 字符串的线程稍微运行一下后就停止
  ↓
- 输出 `Good!` 字符串的线程稍微运行一下后就停止
  ↓
- 输出 `Nice!` 字符串的线程……

实际上运行的线程就像上面这样在不断切换，顺序执行并发处理。

多线程编程时，即使能够并行执行，也必须确保程序能够完全正确地运行。也就是说，必须正确编写线程的互斥处理和同步处理。

并发处理的顺序执行与并发处理的并行执行示意图如图 I1-5 所示。

图 I1-5　并发处理的顺序执行与并发处理的并行执行



I1.3 线程的启动

下面回到线程启动这个话题上。启动线程的方法有如下两种。

（1）利用 Thread 类的子类的实例启动线程

（2）利用 Runnable 接口的实现类的实例启动线程

下面分别了解一下这两种方法。

线程的启动（1）——利用 Thread 类的子类

这里来学习一下利用 Thread 类的子类的实例来启动线程的方法，即上一节中使用的方法。

PrintThread 类（代码清单 I1-4）表示输出 10 000 次指定字符串的线程。输出的字符串通过构造函数的参数传入，并赋给 message 字段。PrintThread 类被声明为 Thread 的子类。

代码清单 I1-4　表示输出 10 000 次指定字符串的线程的类 PrintThread（PrintThread.java）

```
public class PrintThread extends Thread {
    private String message;
    public PrintThread(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print(message);
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/PrintThread 中

Main 类（代码清单 I1-5）是用于创建上面声明的 PrintThread 类的两个实例并利用它们来启动两个线程的程序。

代码清单 I1-5　利用 PrintThread 类启动 2 个线程（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new PrintThread("Good!").start();
        new PrintThread("Nice!").start();
    }
}
```

该程序收录在本书配套的源代码 Introduction1/PrintThread 中

main 方法创建了 PrintThread 类的实例，并直接（不赋给某个变量）调用了该实例的 start 方法，代码如下。

```
new PrintThread("Good!").start();
```

这是创建 PrintThread 类的实例，并调用该实例的 start 方法的语句。

start 方法会启动新的线程，然后由启动的新线程调用 PrintThread 类的实例的 run 方法。

最终结果就是由新启动的线程执行 10 000 次 Good! 字符串输出。

为了程序简洁，上面的程序只用一条语句启动了线程。但实际上，“创建 PrintThread 的实例”和“启动该实例对应的线程”是两个完全不同的处理。也就是说，即便已经创建了实例，但是如果不调用 start 方法，线程也不会被启动。上面这条语句也可以像下面这样写成两句。

```
Thread t = new PrintThread("Good!");
t.start();
```

另外，这里再提醒大家注意，“PrintThread 的实例”和“线程本身”不是同一个东西。即便创建了 PrintThread 的实例，线程也并没有启动，而且就算线程终止了，PrintThread 的实例也不会消失。

主线程在 Main 类的 main 方法中启动了两个线程。随后 main 方法便会终止，主线程也会跟着终止。但整个程序并不会随之终止，因为启动的两个线程在字符串输出之前是不会终止的。**直到所有的线程都终止后，程序才会终止**。也就是说，当这两个线程都终止后，程序才会终止。

▶▶ 小知识：程序的终止

Java 程序的终止是指除守护线程（Daemon Thread）以外的线程全部终止。守护线程是执行后台作业的线程。我们可以通过 setDaemon 方法把线程设置为守护线程。

创建 Thread 类的子类、创建子类的实例、调用 start 方法——这就是利用 Thread 类的子类启动线程的方法。

线程的启动（2）——利用 Runnable 接口

这里来学习一下利用 Runnable 接口的实现类的实例来启动线程的方法。Runnable 接口包含在 java.lang 包中，声明如下。

```
public interface Runnable {
    public abstract void run();
}
```

Runnable 接口的实现类必须要实现 run 方法[①]。

Printer 类（代码清单 I1-6）表示的是一个输出 10 000 次指定字符串的线程。输出的字符串通过构造函数的参数传入，并赋给 message 字段。由于 Printer 类实现（implements）了 Runnable 接口，所以此时也就无需再将 Printer 类声明为 Thread 类的子类。

代码清单 I1-6　输出指定字符串的 Printer 类（Printer.java）

```
public class Printer implements Runnable {
    private String message;
    public Printer(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
```

① java.util.TimerTask 类（11.8 节）虽然实现了 Runnable 接口，但并未实现 run 方法，类似于这样的类都声明为抽象类。

```
            System.out.print(message);
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/Printer 中

Main 类（代码清单 I1-7）是用于创建两个 Printer 类的实例，并利用它们来启动两个线程的程序。

代码清单 I1-7　利用 Runnable 接口启动两个线程（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new Thread(new Printer("Good!")).start();
        new Thread(new Printer("Nice!")).start();
    }
}
```

该程序收录在本书配套的源代码 Introduction1/Printer 中

在代码清单 I1-7 中，创建 Thread 的实例时，构造函数的参数中会传入 Printer 类的实例，然后会调用 start 方法，启动线程，具体如下。

```
new Thread(new Printer("Good!")).start();
           new Printer("Good!")
new Thread(                    )
                                .
                                 start()
                                        ;
```

这是
新建 Printer 类的实例，再以该实例
为参数创建 Thread 类的实例，
然后调用 Thread 类的实例的
start 方法
的语句。

start 方法会启动新的线程，然后由启动的新线程调用 Printer 类的实例的 run 方法。最终结果就是由新启动的线程执行 10 000 次 Good! 字符串输出。上面这条语句也可以像下面这样写成三句。

```
Runnable r = new Printer("Good!");
Thread t = new Thread(r);
t.start();
```

创建 Runnable 接口的实现类，将实现类的实例作为参数传给 Thread 的构造函数，调用 start 方法——这就是利用 Runnable 接口启动线程的方法。

不管是利用 Thread 类的子类的方法（1），还是利用 Runnable 接口的实现类的方法（2），启动新线程的方法最终都是 Thread 类的 start 方法。

▶▶ 小知识：Thread 类和 Runnable 方法

Thread 类本身还实现了 Runnable 接口，并且持有 run 方法，但 Thread 类的 run 方法主体是空的，不执行任何操作。Thread 类的 run 方法通常都由子类的 run 方法重写（override）。

▶▶ 小知识：java.util.concurrent.ThreadFactory 中的线程创建

java.util.concurrent 包中包含一个将线程创建抽象化的 ThreadFactory 接口。利用该接口，我们可以将以 Runnable 作为传入参数并通过 new 创建 Thread 实例的处理隐藏在

ThreadFactory 内部。典型用法如代码清单 I1-8 所示。默认的 ThreadFactory 对象是通过 Executors.defaultThreadFactory 方法获取的。

此处运行的 Printer 类与代码清单 I1-6 里的 Printer 类相同，运行结果与图 I1-4 相同。

代码清单 I1-8　利用 ThreadFactory 新启动线程（Main.java）

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;

public class Main {
    public static void main(String[] args) {
        ThreadFactory factory = Executors.defaultThreadFactory();
        factory.newThread(new Printer("Nice!")).start();
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/jucThreadFactory 中

I1.4　线程的暂停

下面该稍微休息一下了呢……不过，这里说的是线程休息，不是我们哦。本节将介绍一下让线程暂停运行的方法。

线程 Thread 类中的 sleep 方法能够暂停线程运行，Sleep 也就是“休眠”的意思。sleep 方法是 Thread 类的静态方法。

下面这条语句可以**将当前的线程（执行这条语句的线程）暂停约 1000 毫秒（约 1 秒）**。

```
Thread.sleep(1000);
```

代码清单 I1-9 的程序会输出 10 次 Good! 字符串，而每输出 1 次，线程就暂停约 1000 毫秒（约 1 秒）。也就是每隔约 1 秒就输出 1 次 Good! 字符串（图 I1-6）。

代码清单 I1-9　每隔约 1 秒输出 1 次 Good！的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print("Good!");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

该程序收录在本书配套的源代码 Introduction1/Sleep 中

图 I1-6 运行结果

```
Good!Good!Good!Good!Good!Good!Good!Good!Good!Good!  ←每隔约 1 秒输出 1 次 Good!，共输出 10 次
```

sleep 方法的调用放在了 try...catch 中，这是因为，sleep 方法有可能会抛出 InterruptedException 异常。InterruptedException 异常能够取消线程的处理，详细内容请参见 5.6 节。

代码清单 I1-9 中的 catch 处理是空的，这样即使抛出 InterruptedException 异常，也不会执行任何特殊处理（即忽略该异常）。

本书在模拟“非常耗时的处理”时常会用到 sleep 方法。但在实际程序中，sleep 的使用频率并没有这么高。最多也就是在设计一定时间后自动关闭的对话框，或把按钮按下瞬间的状态显示给用户看时才会用到。

▶▶ 小知识：指定到纳秒（ns）单位

在 sleep 方法中，停止时间也可以指定到纳秒（10^{-9} 秒）单位，语法如下。

```
Thread.sleep( 毫秒 , 纳秒 );
```

不过，通常情况下 Java 平台运行环境无法实现这么精确的控制。具体的精确程度依 Java 平台运行环境而不同。

▶▶ 小知识：如何唤醒呢

如果要中途唤醒被 Thread.sleep 休眠的线程，则可以使用 interrupt 方法。详细内容请参见 5.6 节中的“sleep 方法和 interrupt 方法”部分。

I1.5 线程的互斥处理

多线程程序中的各个线程都是自由运行的，所以它们有时就会同时操作同一个实例。这在某些情况下会引发问题。例如，从银行账户取款时，余额确认部分的代码应该是像下面这样的。

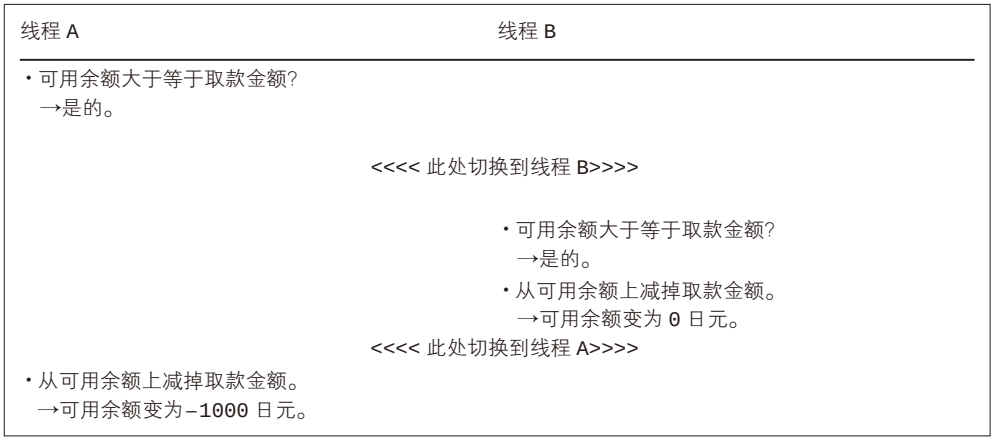
```
if ( 可用余额大于等于取款金额 ) {
    从可用余额上减掉取款金额
}
```

首先确认可用余额，确认是否允许取款。如果允许，则从可用余额上减掉取款金额。这样才不会导致可用余额变为负数。

但是，如果两个线程同时执行这段代码，那么可用余额就有可能会变为负数。

假设可用余额 = 1000 日元，取款金额 = 1000 日元，那么这种情况就如图 I1-7 所示。

图 I1-7 线程 A 执行的两个处理之间插入了线程 B 的处理



（时间流逝顺序为自上而下）

线程 A 和线程 B 同时操作时，有时线程 B 的处理可能会插在线程 A 的“可用余额确认”和“从可用余额上减掉取款金额”这两个处理之间。

这种线程 A 和线程 B 之间互相竞争（race）而引起的与预期相反的情况称为数据竞争（data race）或竞态条件（race condition）。

这时候就需要有一种“交通管制”来协助防止发生数据竞争。例如，如果一个线程正在执行某一部分操作，那么其他线程就不可以再执行这部分操作。这种类似于交通管制的操作通常称为互斥（mutual exclusion）。这种处理就像十字路口的红绿灯，当某一方向为绿灯时，另一方向则一定是红灯。

Java 使用关键字 `synchronized` 来执行线程的互斥处理。

synchronized 方法

如果声明一个方法时，在前面加上关键字 `synchronized`，那么这个方法就只能由一个线程运行。只能由一个线程运行是每次只能由一个线程运行的意思，并不是说仅能让某一特定线程运行。这种方法称为 `synchronized` 方法，有时也称为**同步方法**。

代码清单 I1-10 中的类就使用了 `synchronized` 方法。`Bank`（银行）类中的 `deposit`（存款）和 `withdraw`（取款）这两个方法都是 `synchronized` 方法。

代码清单 I1-10 包含 deposit 和 withdraw 这两个 synchronized 方法的 Bank 类（Bank.java）

```
public class Bank {
    private int money;
    private String name;

    public Bank(String name, int money) {
        this.name = name;
        this.money = money;
    }

    // 存款
    public synchronized void deposit(int m) {
        money += m;
    }
```

```
    // 取款
    public synchronized boolean withdraw(int m) {
        if (money >= m) {
            money -= m;
            return true;    // 取款成功
        } else {
            return false;   // 余额不足
        }
    }

    public String getName() {
        return name;
    }
}
```

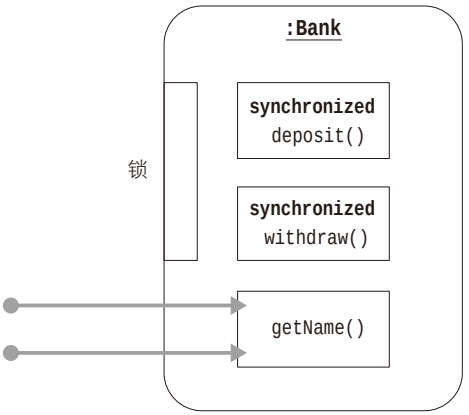
该程序收录在本书配套的源代码 Introduction1/Sync 中

如果有一个线程正在运行 Bank 实例中的 deposit 方法，那么其他线程就无法运行这个实例中的 deposit 方法和 withdraw 方法，需要排队等候。

Bank 类中还有一个 getName 方法。这个方法并不是 synchronized 方法，所以无论其他线程是否正在运行 deposit 或 withdraw，都可以随时运行 getName 方法[①]。

一个实例中的 synchronized 方法每次只能由一个线程运行，而非 synchronized 方法则可以同时由两个以上的线程运行。图 I1-8 展示了由两个线程同时运行 getName 方法的情况。

图 I1-8 多个线程同时运行非 synchronized 方法 getName

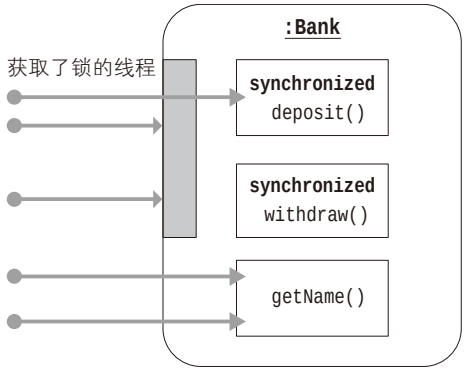


synchronized 方法不允许同时由多个线程运行。在图 I1-8 中，我们在 synchronized 方法左侧放了一个代表“锁”的长方形来表示这点。当一个线程获取了该锁后，长方形这块儿就像筑起的墙一样，可以防止其他线程进入。

图 I1-9 展示了由一个线程运行 deposit 方法的情况。由于该线程获取了锁，所以其他线程就无法运行该实例中的 synchronized 方法。图 I1-9 中，表示锁的长方形被涂成了灰色，这表示该锁已被某一线程获取。

① getName 并未声明为 synchronized 方法，这是因为该方法并未使用多个线程可同时读写的字段。

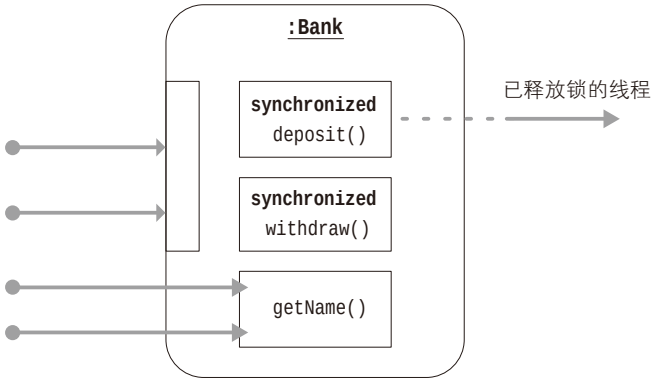
图 I1-9 synchronized 方法每次只能由一个线程运行



请注意，在图 I1-9 中，非 synchronized 的 getName 方法完全不受锁的影响。不管线程是否已经获取锁，都可以自由进入非 synchronized 方法。

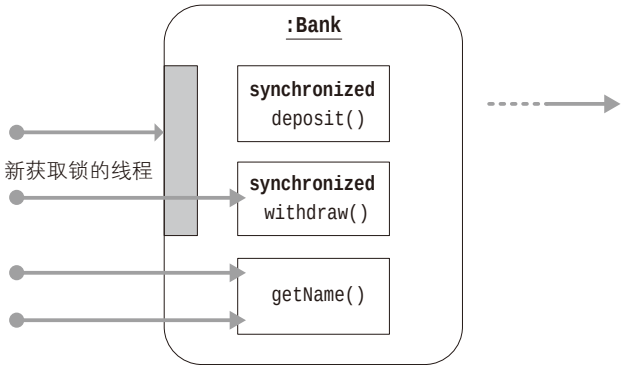
当正在使用 synchronized 方法的线程运行完这个方法后，便会释放锁。图 I1-10 中的长方形锁变为白色表示这个锁已被释放。

图 I1-10 线程运行完 synchronized 方法 deposit 后，释放锁



当锁被释放后，一直等待获取锁的线程中的某一个线程便会获取该锁。但无论何时，获取锁的线程只能是一个。如果等待的线程有很多个，那么没抢到的线程就只能继续等待。图 I1-11 展示的是新获取锁的另一个线程开始运行 synchronized 方法的情况。

图 I1-11 获取锁的另一个线程开始运行 synchronized 方法



每个实例拥有一个独立的锁。因此，并不是说某一个实例中的 synchronized 方法正在执行中，其他实例中的 synchronized 方法就不可以运行了。图 I1-12 展示了 bank1 和 bank2 这两个实例中的 synchronized 方法由不同的线程同时运行的情况。

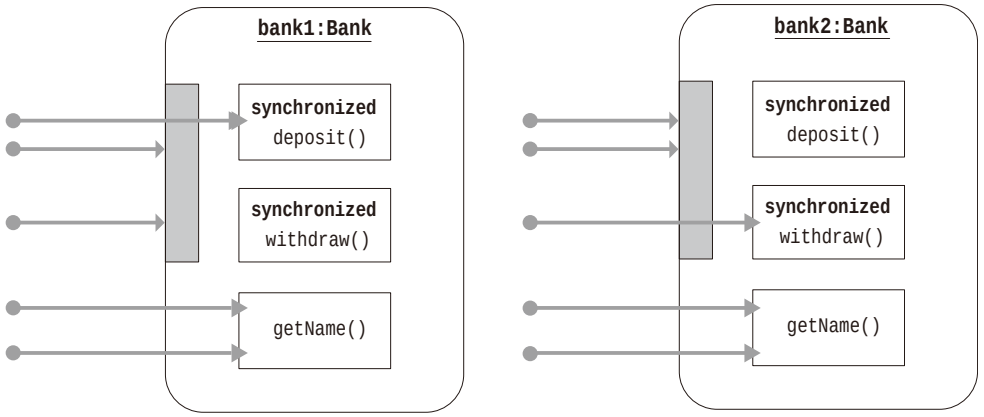
▶▶ 小知识：锁和监视

线程的互斥机制称为**监视**（monitor）。另外，获取锁有时也叫作“拥有（own）监视”或“持有（hold）锁”。

当前线程是否已获取某一对象的锁可以通过 Thread.holdsLock 方法来确认。当前线程已获取对象 obj 的锁时，可使用 assert 来像下面这样表示出来。

```
assert Thread.holdsLock(obj);
```

图 I1-12 每个实例都拥有一个独立的锁



synchronized 代码块

如果只是想让方法中的某一部分由一个线程运行，而非整个方法，则可使用 synchronized 代码块，格式如下所示。

```
synchronized ( 表达式 ) {
    ...
}
```

其中的“表达式”为获取锁的实例。synchronized 代码块用于精确控制互斥处理的执行范围，具体示例将在第 6 章介绍。

◆synchronized 实例方法和 synchronized 代码块

假设有如下 synchronized 实例方法。

```
synchronized void method() {
    ...
}
```

这跟下面将方法体用 synchronized 代码块包围起来是等效的。

```
void method() {
   synchronized (this) {
       ...
   }
}
```

也就是说，synchronized 实例方法是使用 this 的锁来执行线程的互斥处理的。

◆synchronized 静态方法和 synchronized 代码块

假设有如下 synchronized 静态方法。synchronized 静态方法每次只能由一个线程运行，这一点和 synchronized 实例方法相同。但 synchronized 静态方法使用的锁和 synchronized 实例方法使用的锁是不一样的。

```
class Something {
    static synchronized void method() {
        ...
    }
}
```

这跟下面将方法体用 synchronized 代码块包围起来是等效的。

```
class Something {
    static void method() {
        synchronized (Something.class) {
            ...
        }
    }
}
```

也就是说，synchronized 静态方法是使用该类的类对象的锁来执行线程的互斥处理的。Something.clas 是 Something 类对应的 java.lang.class 类的实例。

▶▶ 小知识：synchronized 语句

《Java 编程规范》（见附录 G 中的 [JLS2] 和 [JLS3]）[①] 中将 synchronized 代码块称为 synchronized 语句（synchronized statement）。

关于线程的互斥处理，第 1 章“Single Threaded Execution 模式”中会详细讲解。

I1.6 线程的协作

上一节讲到，如果有一个线程正在运行 synchronized 方法，那么其他线程就无法再运行这个方法了。这就是简单的互斥处理。

① 此书第二版无中文版，第三版由中国电力出版社于 2006 年 7 月引进出版，译者为陈宗斌等。

——编者注

假如我们现在想执行更加精确的控制，而不是单纯地等待其他线程运行终止，例如下面这样的控制。

- 如果空间为空则写入数据；如果非空则一直等待到变空为止
- 空间已为空时，“通知”正在等待的线程

此处是根据“空间是否为空”这个条件来执行线程控制的。Java 提供了用于执行线程控制的 wait 方法、notify 方法和 notifyAll 方法。wait 是让线程等待的方法，而 notify 和 notifyAll 是唤醒等待中的线程的方法。

等待队列——线程休息室

在学习 wait、notify 和 notifyAll 之前，我们先来学习一下等待队列。所有实例都拥有一个等待队列，它是在实例的 wait 方法执行后停止操作的线程的队列。打个比方来说，就是为每个实例准备的线程休息室。

在执行 wait 方法后，线程便会暂停操作，进入等待队列这个休息室。除非发生下列某一情况，否则线程会一直在等待队列中休眠。当下列任意一种情况发生时，线程便会退出等待队列。

- 有其他线程的 notify 方法来唤醒线程
- 有其他线程的 notifyAll 方法来唤醒线程
- 有其他线程的 interrupt 方法来唤醒线程
- wait 方法超时

下面以图配文依次讲解 wait、notify 和 notifyAll。而关于 interrupt 方法，将会在 5.6 节中的“wait 方法和 interrupt 方法”部分进行讲解。关于 wait 方法的超时，将会在 4.6 节讲解。

wait 方法——将线程放入等待队列

wait（等待）方法会让线程进入等待队列。假设我们执行了下面这条语句。

```
obj.wait();
```

那么，当前线程便会暂停运行，并进入实例 obj 的等待队列中。这叫作“线程正在 obj 上 wait”。

如果实例方法中有如下语句（1），由于其含义等同于（2），所以执行了 wait() 的线程将会进入 this 的等待队列中，这时可以说“线程正在 this 上 wait”。

```
wait();          (1)
this.wait();     (2)
```

若要执行 wait 方法，线程必须持有锁（这是规则）。但如果线程进入等待队列，便会释放其实例的锁。整个操作过程如图 I1-13 ~ 图 I1-15 所示。

▶▶ 小知识：等待队列

等待队列是一个虚拟的概念。它既不是实例中的字段，也不是用于获取正在实例上等待的线程的列表的方法。

图 I1-13 获取了锁的线程 A 执行 wait 方法

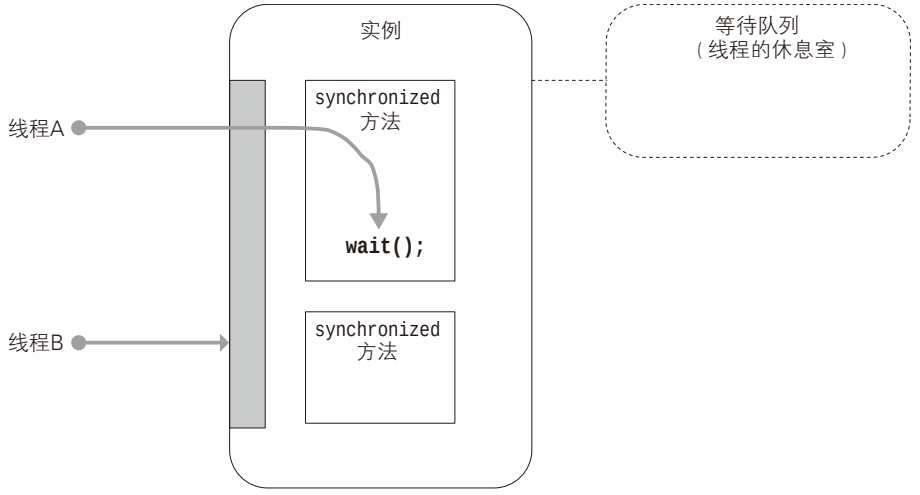


图 I1-14 线程 A 进入等待队列，释放锁

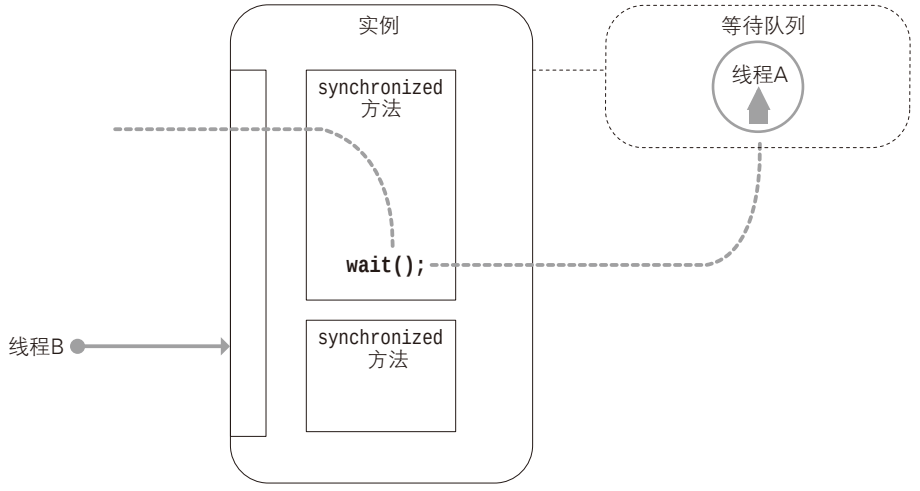
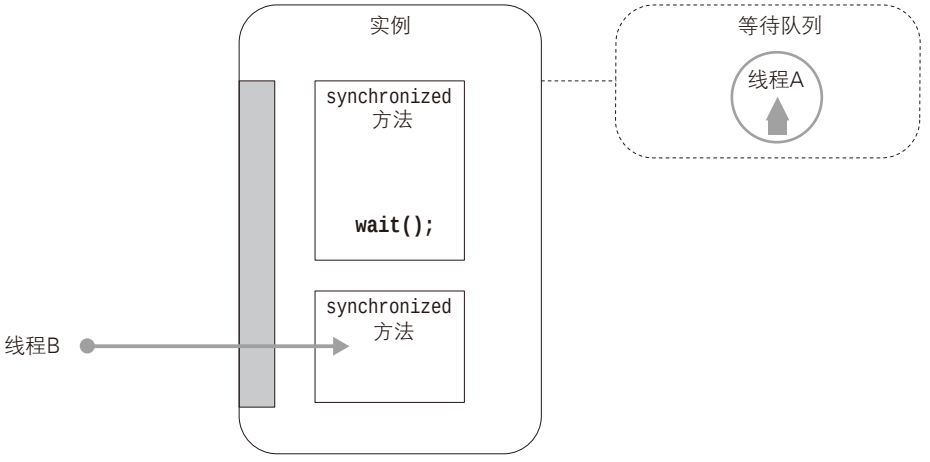


图 I1-15 线程 B 能够获取锁



notify 方法——从等待队列中取出线程

notify（通知）方法会将等待队列中的一个线程取出。假设我们执行了下面这条语句。

```
obj.notify();
```

那么 obj 的等待队列中的一个线程便会被选中和唤醒，然后就会退出等待队列。
整个操作过程如图 I1-16 ~ 图 I1-19 所示。

图 I1-16 获取了锁的线程 B 执行 notify 方法

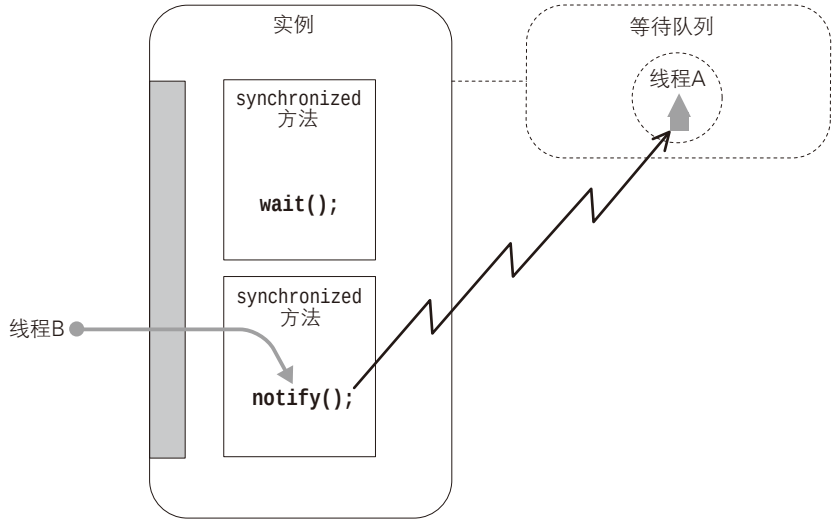


图 I1-17 线程 A 退出等待队列，想要进入 wait 的下一个操作，但刚才执行 notify 的线程 B 仍持有着锁

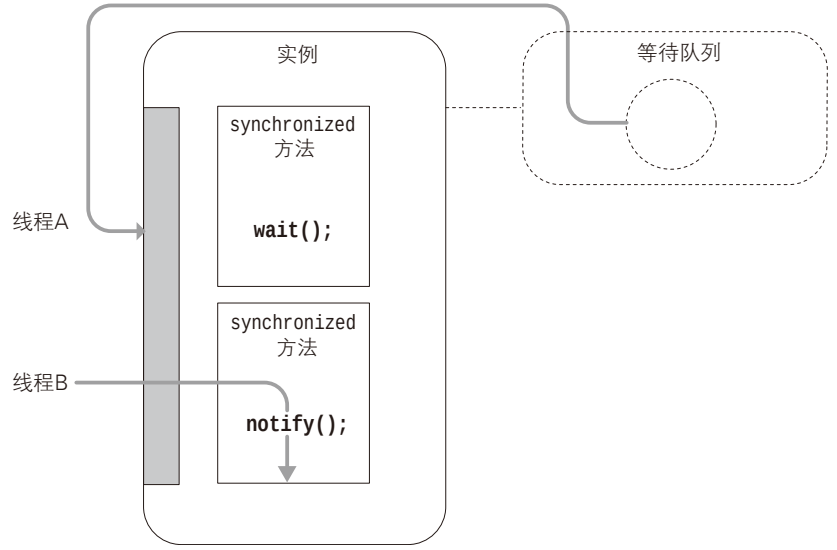


图 I1-18 刚才执行 notify 的线程 B 释放了锁

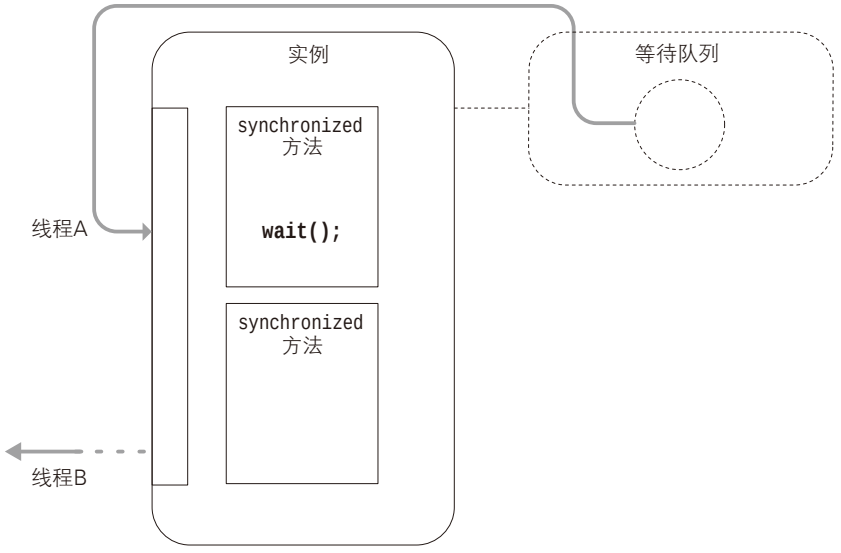
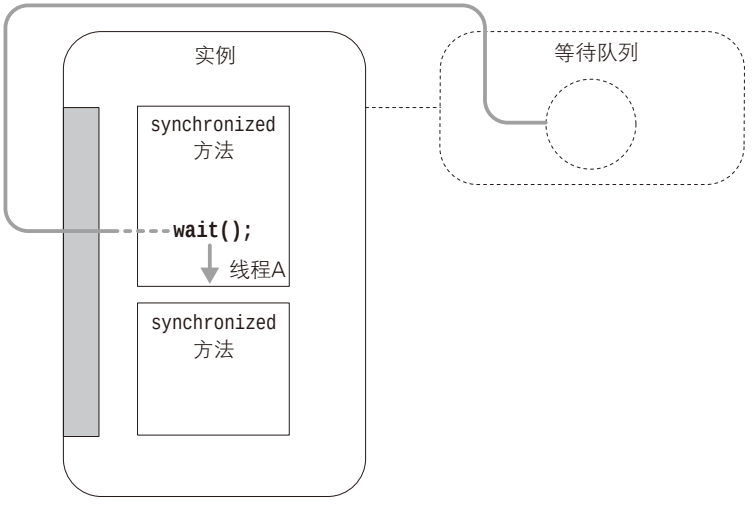


图 I1-19 退出等待队列的线程 A 获取锁，执行 wait 的下一个操作



同 wait 方法一样，若要执行 notify 方法，线程也必须持有要调用的实例的锁（这是规则）。

▶▶ 小知识：执行 notify 后的线程状态

notify 唤醒的线程并不会在执行 notify 的一瞬间重新运行。因为在执行 notify 的那一瞬间，执行 notify 的线程还持有着锁，所以其他线程还无法获取这个实例的锁（图 I1-17）。

▶▶ 小知识：执行 notify 后如何选择线程

假如在执行 notify 方法时，正在等待队列中等待的线程不止一个，对于“这时该如何来选择线程”这个问题规范中并没有作出规定。究竟是选择最先 wait 的线程，还是随机选择，或者采用其他方法要取决于 Java 平台运行环境。因此编写程序时需要注意，最好不要编写依赖于所选线程的程序。

notifyAll 方法——从等待队列中取出所有线程

`notifyAll`（通知大家）方法会将等待队列中的所有线程都取出来。例如，执行下面这条语句之后，在 `obj` 实例的等待队列中休眠的所有线程都会被唤醒。

```
obj.notifyAll();
```

如果简单地在实例方法中写成下面（1）这样，那么由于其含义等同于（2），所以该语句所在方法的实例（`this`）的等待队列中所有线程都会退出等待队列。

```
notifyAll();          （1）
this.notifyAll();     （2）
```

图 I1-20 和图 I1-21 展示了 `notify` 方法和 `notifyAll` 方法的差异。`notify` 方法仅唤醒一个线程，而 `notifyAll` 则唤醒所有线程，这是两者之间唯一的区别。

图 I1-20　notify 方法仅唤醒一个线程，并让该线程退出等待队列

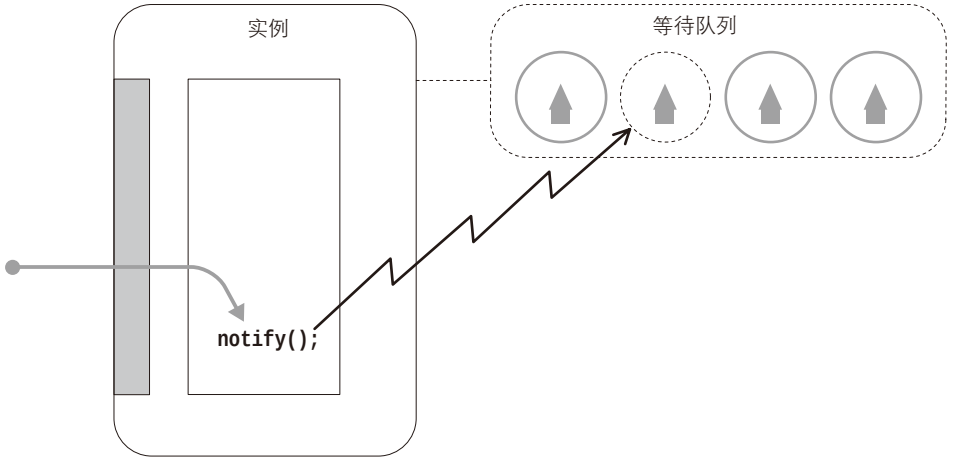
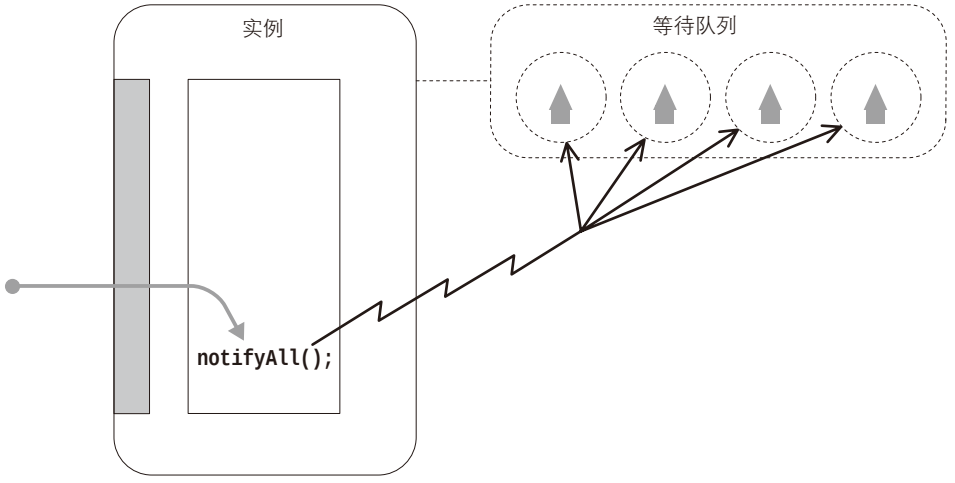


图 I1-21　notifyAll 方法唤醒所有线程，并让所有线程都退出等待队列



同 wait 方法和 notify 方法一样，notifyAll 方法也只能由持有要调用的实例的锁的线程调用。

刚被唤醒的线程会去获取其他线程在进入 wait 状态时释放的锁。但现在锁是在谁的手中呢？对，就是执行 notifyAll 的线程正持有着锁。因此，唤醒的线程虽然都退出了等待队列，但都在等待获取锁，处于阻塞状态。只有在执行 notifyAll 的线程释放锁以后，其中一个幸运儿才能够实际运行。

▶▶ 小知识：如果线程未持有锁会怎样呢

如果未持有锁的线程调用 wait、notify 或 notifyAll，异常 java.lang.IllegalMonitorStateException 会被抛出。

▶▶ 小知识：该使用 notify 方法还是 notifyAll 方法呢

notify 方法和 notifyAll 方法非常相似，到底该使用哪一个呢？实际上，这很难选择。

由于 notify 唤醒的线程较少，所以处理速度要比使用 notifyAll 时快。

但使用 notify 时，如果处理不好，程序便可能会停止。一般来说，使用 notifyAll 时的代码要比使用 notify 时的更为健壮。

除非开发人员完全理解代码的含义和范围，否则使用 notifyAll 更为稳妥。使用 notify 时发生问题的示例将在第 5 章的习题 5-8 中探讨。

wait、notify、notifyAll 是 Object 类的方法

wait、notify 和 notifyAll 都是 java.lang.Object 类的方法，而不是 Thread 类中固有的方法。

下面再来回顾一下 wait、notify 和 notifyAll 的操作。

- obj.wait() 是将当前线程放入 obj 的等待队列中
- obj.notify() 会从 obj 的等待队列中唤醒一个线程
- obj.notifyAll() 会从 obj 的等待队列中唤醒所有线程

换句话说，wait、notify 和 notifyAll 这三个方法与其说是针对线程的操作，倒不如说是针对实例的等待队列的操作。由于所有实例都有等待队列，所以 wait、notify 和 notifyAll 也就成为了 Object 类的方法。

▶▶ 小知识：wait、notify、notifyAll 也是 Thread 类的方法

wait、notify 和 notifyAll 确实不是 Thread 类中固有的方法。但由于 Object 类是 Java 中所有类的父类，所以也可以说 wait、notify 和 notifyAll 都是 Thread 类的方法。

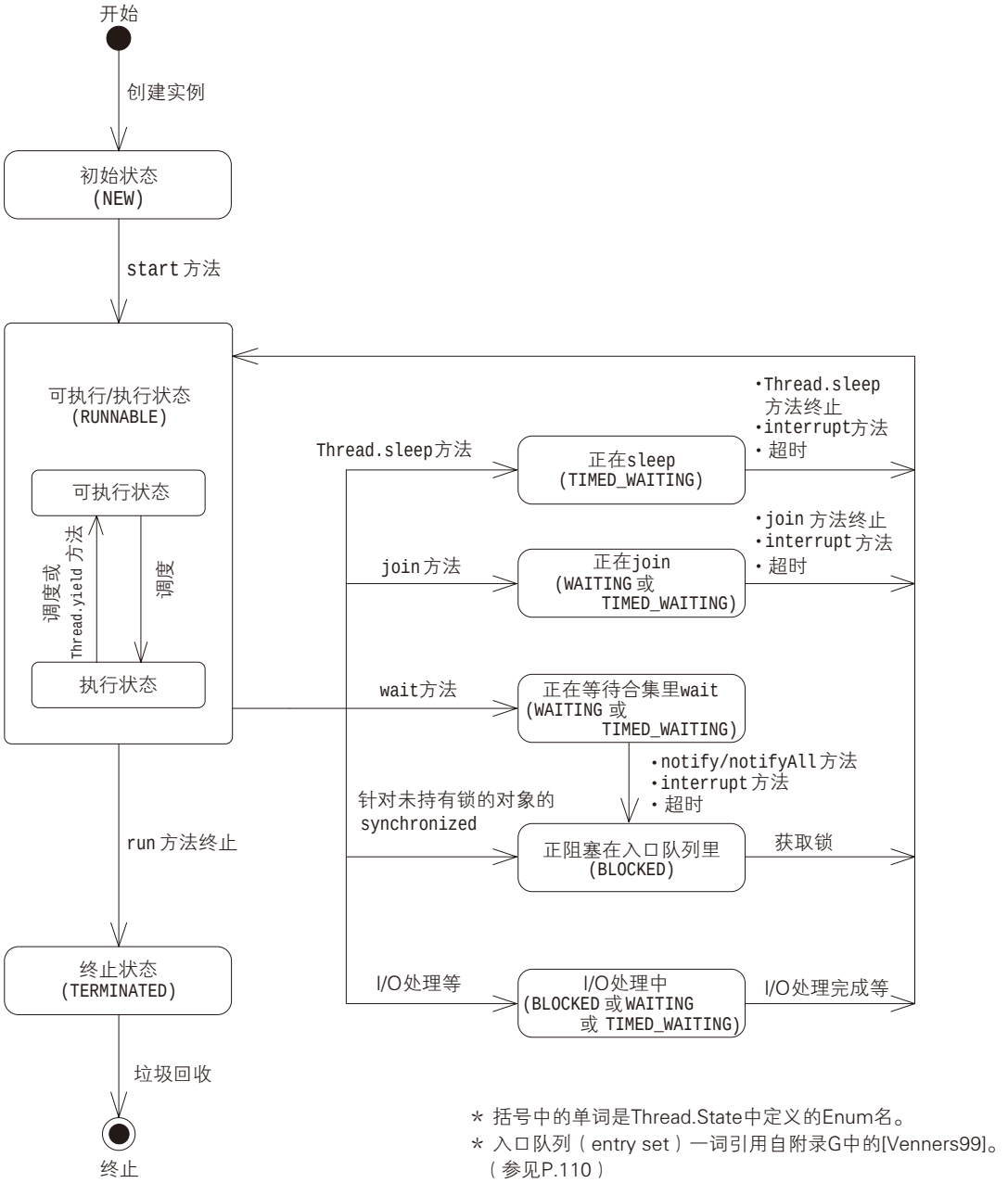
关于 wait、notify 和 notifyAll 的用法，第 3 章“Guarded Suspension 模式”将会详细解说。

I1.7 线程的状态迁移

图 I1-22 中整理了线程的状态迁移，供各位读者参考。图中的线程状态（Thread.Stat 中定

义的 Enum 名）NEW、RUNNABLE、TERMINATED、WAITING、TIMED_WAITING 和 BLOCKED 都能够在程序中查到。各个状态的值都可以通过 Thread 类的 getState 方法获取。

图 I1-22 线程的状态迁移图



＊ 括号中的单词是Thread.State中定义的Enum名。
＊ 入口队列（entry set）一词引用自附录G中的[Venners99]。（参见P.110）

本图制作时参考了附录G中的[Lea][JLS2][JLS3][Holub00]及《JavaWorld》2002年4月刊的内容

I1.8 线程相关的其他话题

下列与线程相关的话题会在相关章节中介绍。

- **用于取消线程处理的中断**
  （`interrupt`、`isInterrupted`、`interrupted`、`InterruptedException`）
  →第 5 章
- **线程的优先级**
  （`setPriority`、`getPriority`）
  →附录 C
- **等待线程终止**
  （`join`）
  →第 5 章、第 10 章

▶▶小知识：“阻塞”和“被阻塞”

有时候，线程会因某些原因而无法继续运行。例如，当某线程 A 欲执行 `synchronized` 方法时，如果其他方法已经获取了该实例的锁，那么线程 A 就无法继续运行了。这种状态就称为“线程 A 阻塞”。

有的读者可能认为这是因外部因素而停止的，应该称为“被阻塞”，这种被动表达更容易理解。虽然本书采用的都是“阻塞”这种表达方式，但这里的“阻塞”跟“被阻塞”含义是相同的。

I1.9 本章所学知识

在本章中，我们学习了 Java 线程的相关内容。

- 何谓线程
- 线程的启动
- 线程的暂停
- 线程的互斥处理
- 线程的协作

请读者试着做一做下面的练习题，测验一下自己是否真正理解了本章所学习的内容。答案在本书附录 A 中，但是请不要直接查看答案，先自己动脑好好思考哦。

I1.10 练习题

答案请参见附录 A（P.338）

●习题 I1-1（基础知识测试）

阅读下面内容，叙述正确请打○，错误请打 ×。

（1）在 Java 程序中，至少有一个线程在运行。

（2）调用 Thread 类的 run 方法后，新的线程就会启动。

（3）start 方法和 run 方法声明在 Runnable 接口中。

（4）有时多个线程会调用同一个实例的方法。

（5）有时多个线程会调用 Thread 类的一个实例的方法。

（6）sleep 方法执行后，在指定时间内所有的线程都会暂停。

（7）某个线程在运行 synchronized 方法时，其他所有线程都会停止运行。

（8）执行 sleep 方法后的线程仅在指定时间内待在等待队列中。

（9）wait 方法的调用语句必须写在 synchronized 方法中。

（10）notifyAll 方法是 java.lang.Object 类的实例方法。

●习题 I1-2（互斥处理）

在图 I1-4 的运行结果示例中，Good! 和 Nice! 字符串并没有像下面这样以字母为单位交错排列。

```
GoNiod!ce!
GooNdi!ce!
```

这是为什么呢？

●习题 I1-3（线程的运行）

当下面的程序（代码清单 I1-11 和代码清单 I1-12）运行时，程序会在输出 1000 个“*”后，再输出 1000 个“+”（图 I1-23）。请问，为什么输出结果并不是“*”和“+”交错混杂的呢？

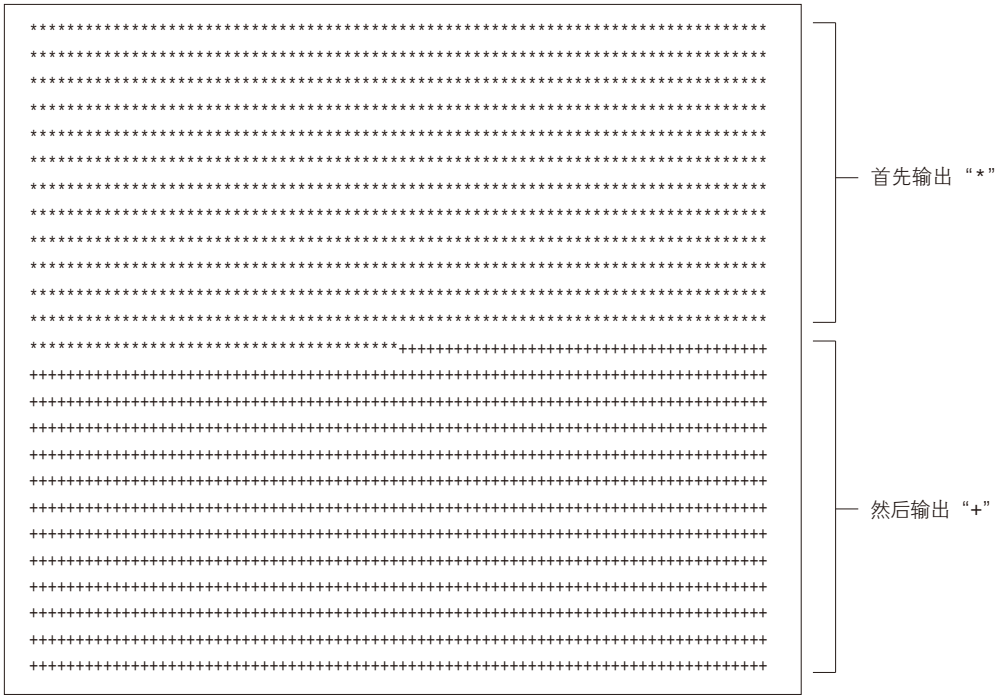
代码清单 I1-11　哪里有问题（Main.java）

```
public class main {
    public static void main(String[] args) {
        new PrintThread("*").run();
        new PrintThread("+").run();
    }
}
```

代码清单 I1-12　用于输出指定字符串的 PrintThread 类（PrintThread.java）

```
public class printThread extends Thread {
    private String message;
    public PrintThread(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.print(message);
        }
    }
}
```

图 I1-23 运行结果



●习题 I1-4（异常检测）

代码清单 I1-13 在本章的 Bank 类（代码清单 I1-10）的基础上，去掉了 synchronized 关键字，并新增了一个 check 方法。check 方法会在可用余额为负数时显示警告消息。请确认当有多个线程使用这个银行（Bank 类）时，可用余额变为负数的可能性。

代码清单 I1-13 确认可用余额变为负数的可能性（Bank.java）

```
public class Bank {
    private int money;
    private String name;
    public Bank(String name, int money) {
        this.name = name;
        this.money = money;
    }
    public void deposit(int m) {
        money += m;
    }
    public boolean withdraw(int m) {
        if (money >= m) {
            money -= m;
            check();
            return true;
        } else {
            return false;
        }
    }
    public String getName() {
        return name;
    }
```

```
    private void check() {
        if (money < 0) {
            System.out.println("可用余额为负数！ money = " + money);
        }
    }
}
```

●习题 I1-5（线程的暂停）

某人写了如下代码，想让启动的线程暂停约 1 秒。但这个代码是错误的，为什么呢？假设下面的 MyThread 就是代码清单 I1-2 中声明的那个类。

```
// 创建实例
MyThread t = new MyThread();
// 启动线程
t.start();
// 暂停已启动的线程
try {
    t.sleep(1000);
} catch (InterruptedException e) {
}
```

●习题 I1-6（互斥处理）

假设存在一个如代码清单 I1-14 这样声明的 Something 类，变量 x,y 表示 Something 类的两个不同实例。请判断下列组合是否允许多个线程同时运行，允许请画√，否则请画 × 。

代码清单 I1-14 Something 类（Something.java）

```
public class Something {
    public                      void iA() { }
    public                      void iB() { }
    public        synchronized void iSyncA() { }
    public        synchronized void iSyncB() { }
    public static               void cA() { }
    public static               void cB() { }
    public static synchronized void cSyncA() { }
    public static synchronized void cSyncB() { }
}
```

（1）x.iA(); 与 x.iA();	（2）x.iA(); 与 x.iB();	（3）x.iA(); 与 x.iSyncA();
（4）x.iSyncA(); 与 x.iSyncA();	（5）x.iSyncA(); 与 x.iSyncB();	（6）x.iSyncA(); 与 y.iSyncA();
（7）x.iSyncA(); 与 y.iSyncB();	（8）x.iSyncA(); 与 Something.cA();	（9）x.iSyncA(); 与 Something.cSyncA();
（10）Something.cSyncA(); 与 Something.cSyncA();	（11）Something.cSyncA(); 与 Something.cSyncB();	（12）x.cSyncA(); 与 y.cSyncB();

序章 2

多线程程序的评价标准

12.1 多线程程序的评价标准

如果你对程序的评价仅仅停留在“这个程序写得好”“这个程序写得差”的程度，那么你就无法深入研究程序。不能只是单纯地指出“好与差”，而应该遵照一个评价标准，指出程序“好在哪儿、差在哪儿”。

本章将根据 Doug Lea 的分类，探讨多线程程序的评价标准。详细内容请参见附录 G 中的 [Lea]。

安全性——不损坏对象

所谓安全性（safety）就是不损坏对象。这是程序正常运行的必要条件之一。

对象损坏只是一种比喻，实际上，对象是内存上的一种虚拟事物，并不会实际损坏。对象损坏是指对象的状态和设计者的原意不一致，通常是指对象的字段的值并非预期值。

以序章 1 中介绍的银行账户为例，假设银行账户的可用余额变为了负数，而设计者此前并没有设想它会变为负数。这时就可以说，表示银行账户的对象“损坏”了。

如果一个类即使被多个线程同时使用，也可确保安全性，那么这个类就称为线程安全（thread-safe）类。由于类库中还存在着非线程安全的类，所以在多线程程序中使用类时一定要特别注意。例如，`java.util.Vector` 类是线程安全的类，而 `java.util.ArrayList` 则是非线程安全的类。一般在 API 文档中能够查到各个类是否是线程安全的。

▶▶ 小知识：线程安全和线程兼容

`ArrayList` 虽然是非线程安全的，但通过执行适当的互斥处理，也可以安全地使用。附录 G 中的 [Bloch] 将此种情况称为线程兼容（thread-compatible）。详细内容请参见 [Bloch] 中的第 52 项。

另外，本书 2.7 节会展示一个对 `ArrayList` 执行适当的互斥处理的程序。

▶▶ 小知识：synchronized 和线程安全

某个线程是线程安全的还是非线程安全的，与这个类的方法是否是 `synchronized` 方法无关。javadoc 生成的 API 文档中也并未明确标注 `synchronized`，因为 `synchronized` 仅是具体编写方法时的信息而已。

生存性——必要的处理能够被执行

生存性（liveness）是指无论是什么时候，必要的处理都一定能够被执行。这也是程序正常运行的必要条件之一（也有人也将 liveness 翻译为“**活性**”）。

即使对象没有损坏，也不代表程序就一定好。极端一点说，假如程序在运行过程中突然停止了，这时，由于处理已经停止，对象的状态就不会发生变化了，所以对象状态也就不会异常。这虽然符合前面讲的“安全性”条件，但无法运行的程序根本没有任何意义。无论是什么时候，必要的处理都一定能够被执行——这就是生存性。

有时候安全性和生存性会相互制约。例如，有时只重视安全性，生存性就会下降。最典型的事例就是死锁（deadlock），即多个线程互相等待对方释放锁的情形。关于死锁的详细内容将在第 1 章“Single Threaded Execution 模式”中讲解。

可复用性——类可重复利用

可复用性（reusability）是指类能够重复利用。这虽然不是程序正常运行的必要条件，但却是提高程序质量的必要条件。

类如果能够作为组件从正常运行的软件中分割出来，那么就说明这个类有很高的可复用性。

在编写多线程程序时，如果能够巧妙地将线程的互斥机制和方针隐藏到类中，那这就是一个可复用性高的程序。J2SE 5.0 中引入的 `java.util.concurrent` 包中就提供了便于多线程编程的可复用性高的类。

性能——能快速、大批量地执行处理

性能（performance）是指能快速、大批量地执行处理。这也不是程序正常运行的必要条件，但却是提高程序质量时应该考虑的条件。

影响性能的因素有好多种。下面是从 Doug Lea 的性能分类中摘录出的主要部分。

吞吐量（throughput）是指单位时间内完成的处理数量。能完成的处理越多，则表示吞吐量越大。

响应性（responsiveness）是指从发出请求到收到响应的时间。时间越短，响应性也就越好。在 GUI 程序中，相比于到处理“结束”时的时间，到处理“开始”时的时间更为重要。前者是指实际处理所花费的时间，而后者是到程序开始响应用户所花费的时间。相比于按下按钮后无任何反应，10 秒后才提示“处理完毕”这种方式，在按下按钮时立刻提示“处理开始”这种方式的响应性更高，即便到处理结束花费的时间稍多一点也没关系。响应性好也称为等待时间（latency）短。

容量（capacity）是指可同时进行的处理数量。例如，服务器能同时处理的客户端数或文件数等。

其他的诸如效率（efficiency）、可伸缩性（scalability）、降级（degradation）等，也可作为性能的评价标准。

有时候，这些要素之间会相互制约（也就是常说的有得必有失）。例如，如果要提高吞吐量，那么在很多情况下，程序的响应性就会下降。就像我们为了提高工作量而目不转睛地干活时，如果有人打招呼，那我们的反应就会慢半拍这样。反应慢半拍也就是说响应性变低了。另外，如果要提高安全性，那么性能就可能会下降（如吞吐量变小）。这就好比是为了防止混乱而减少一次处理的工作量时，一定时间内能处理的工作量自然而然地就变少了。

评价标准总结

安全性（safety）和生存性（liveness）是必须遵守的标准。既不能损坏对象，也一定要执行必要的处理。设计多线程系统时，请务必遵守安全性和生存性这两个标准。

重要的是，还要在满足这两个必要条件的基础上，考虑**如何提高可复用性（reusability）和性能（performance）**。

本书遵从下列评价标准来分析各个模式。

- 安全性和生存性：必要条件
- 可复用性和性能：提高质量

另外，在设计模式时使用的评价标准也称为**约束力（force）**。约束力是设计者面对的“限制”

和“压力”。尤其是在与并发相关的模式中，安全性、生存性、可复用性和性能都应该算是特别重要的约束力。

I2.2 本章所学知识

在本章中，我们学习了多线程程序的评价标准，同时也了解到这些评价标准有时是互相制约的。

下一章将正式开始介绍各个模式。在这之前先来做做下面的练习题，测验一下自己是否真正理解了本章所学习的内容吧。

I2.3 练习题

答案请参见附录 A（P.341）

●习题 I2-1（评价标准应用）

请参照本章讲述的评价标准分析一下下列表述。

（1）在方法中一概加上 synchronized 就“好”。

（2）synchronized 方法中进入了无限循环，这程序“不好”。

（3）由于程序错误，启动了 100 个只是进行无限循环的线程，但这些线程也不过是在循环执行而已，所以还算“好”吧。

（4）这个服务器每次只能连接一个客户端，请将该服务器改“好”一点。

（5）这个查找程序可真够“差”的，一旦查找开始，在全部查找完毕之前都无法取消。

（6）这样的话，线程 A 和线程 B 就都需要互斥处理，这点“不好”。

●习题 I2-2（吞吐量）

如果把线程个数变为 2 倍，那么吞吐量是不是也会变为 2 倍呢？