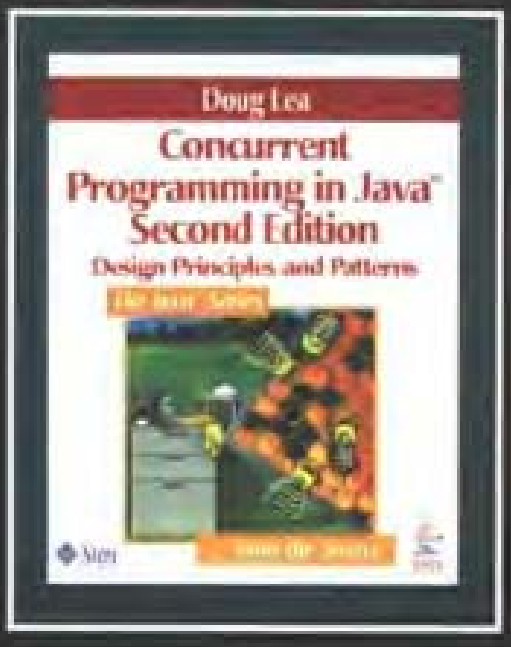
开 发 大 师 系 列

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java 并发编程
设计原则与模式
（第二版）

[美] Doug Lea 著
赵涌 齐科科 郑承豫 郭明亮 译



- Sun 公司 Java 核心技术图书之一
- 适合希望掌握 Java 并发编程的中高级程序员
- 涵盖了有关并发软件开发诸多方面的模式

中国电力出版社
www.infopower.com.cn

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java 并发编程
设计原则与模式（第二版）

本书全面介绍了如何使用 Java 2 平台进行并发编程，较上一版新增和扩展的内容包括：

- 存储模型
- 取消
- 可移植的并行编程
- 实现并发控制的工具类

Java 平台提供了一套广泛而功能强大的 API，工具和技术。内建支持线程是它的一个强大的功能。这一功能为使用 Java 编程语言的程序员提供了并发编程这一诱人但同时也非常具有挑战性的选择。

本书通过帮助读者理解有关并发编程的模式及其利弊，向读者展示了如何更精确地使用 Java 平台的线程模型。

这里，读者将通过使用 java.lang.Thread 类、synchronized 和 volatile 关键字，以及 wait、notify 和 notifyAll 方法，学习如何初始化、控制和协调并发操作。此外，本书还提供了有关并发编程的全方位的详细内容，例如限制和同步、死锁和冲突、依赖于状态的操作控制、异步消息传递和控制流、协作交互，以及如何创建基于 Web 的服务和计算型服务。

本书的读者对象是那些希望掌握并发编程的中高级程序员。从设计模式的角度，本书提供了标准的设计技巧，以创建和实现用来解决一般性并发编程问题的组件。贯串全书的大量示例代码详细地阐述了在讨论中所涉及到的并发编程理念的细微之处。

Doug Lea 是面向对象技术和软件复用的前沿专家之一。他和 Sun 实验室开展合作研究长达五年之久。Lea 是 SUNY Oswego 大学计算机科学系的教授。他是计算机应用纽约先进技术中心（New York Center for Advanced Technology in Computer Application）的软件工程实验室主任，也是 Syracuse 电气与计算机工程系的副教授。此外，他还是《Object-Oriented System Development》（Addison-Wesley, 1993）一书的作者之一。他在 New Hampshire 大学获得了学士、硕士和博士学位。

责任编辑 / 朱恩从
封面设计 / 王红卿

ISBN 7-5083-1828-5

9 787508 318288 >

ISBN 7-5083-1828-5
定价：35.00 元

开 发 大 师 系 列

Concurrent Programming in Java
Design Principles and Patterns Second Edition

Java 并发编程
设计原则与模式
（第二版）

[美] Doug Lea 著
赵涌 齐科科 郑承豫 郭明亮 译

中国电力出版社

Concurrent Programming in Java : Design Principles and Patterns Second Edition (ISBN 0-201-31009-0)
Doug lea
Authorized translation from the English language edition, entitled Concurrent Programming in Java : Design Principles and Patterns Second Edition, published by Addison Wesley Longman, Inc.,

图书在版编目（CIP）数据
Java 并发编程：设计原则与模式 /（美）利著；赵涌等译. —2 版. —北京：中国电力出版社，2003
ISBN 7-5083-1828-5
Ⅰ.J... Ⅱ.①利...②赵... Ⅲ.JAVA 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 108178 号

书　　名：Java并发编程：设计原则与模式
编　　著：（美）道格•利
翻　　译：赵涌 等
责任编辑：朱恩从
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：18　　字　　数：417千字
书　　号：ISBN 7-5083-1828-5
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：35.00 元

译 者 序

自 20 世纪 90 年代以来，并发编程一直是软件设计的热门话题。随着大型企业软件的演化，以及 Java 与.net 的斗法，并发编程的概念正被不断地强化，其涵盖的内容也不断扩大，在所有有关软件和编程的杂志和论文中，它几乎独占鳌头。

由于 Java 语言在设计之初就考虑到了并发编程的因素，所以在和 C++的竞争中，它占了很大的优势。虽然 J2EE 规范了一些并发编程的底层设计，使得开发人员可以更专注于业务逻辑的实现，但作为大型软件的开发人员或架构师，对并发编程的理解和运用仍是必需的，而且可以说是永无止境的。

作为 Java 1.5 并发 JSR 的带头人，Doug Lea 编写的这本《Java 并发编程：设计原则与模式》可以称为这方面的经典之作。随着 util.concurrent 包的广泛商用化，以及加入 Java 1.5 标准库的临近，潜在的读者数量正在倍增。虽然国外读者对这本书的深度和广度赞誉甚高，但由于 Doug Lea 的严密论述而导致的语言相对深奥的问题也正日渐显露。因此，为了大家能更好地理解本书所述内容，我们在翻译的时候尽量使用意译，将英文中复杂的句法尽量简化。

翻译组的成员包括赵涌、齐科科、任文捷、关承豫和郭明亮等。具体负责的工作和个人介绍如下：

- ✧ 赵涌，负责总体审稿和校对。具有多年的编程经验，从 1998 年开始由 C/C++编程转为 Java 编程，一直在电信领域编写和设计分布式应用软件。现在对 Java 服务器端的模式和架构非常感兴趣。
- ✧ 齐科科，负责第 1 章的翻译工作。是开源软件爱好者，blogger。兴趣方向：并行与分布式计算。
- ✧ 任文捷，负责第 2 章的翻译工作。2001 年毕业于哈尔滨工业大学计算机系统结构研究室，硕士学历，主要研究方向为 Linux 操作系统健壮性分析。现在就职于联想研究院，主要研究方向是语音识别、自然语言理解和人机交互界面等。
- ✧ 关承豫，负责第 3 章的翻译工作。外资 Java 软件研发中心高级程序员，开源软件的爱好者。研究方向：面向对象设计、设计模式、企业级软件设计和 GUI 相关技术。
- ✧ 郭明亮，负责第 4 章的翻译工作。多年 Java 开发经验，具有丰富的实际项目经验，参与过多个大型 Java 项目的开发与设计。目前兴趣：分布式计算。

本书由 JavaResearch.org 组织翻译和总体协调，我们还在 JavaResearch.org 的论坛中专门开辟了“译作支持区”（http://www.javaresearch.org/forum/forum.jsp?column=481），大家在阅读本书过程中所遇到的任何问题或迸发的任何感想都可以在那里发表，直接和译者及更多本书的读者进行交流。有关本书的一些勘误，我们将在论坛中及时发布和更新。由于水平有限，书中可能还存在着错误和不足，敬请读者朋友批评指正。

Java 研究组织《Java 并发编程：设计原则与模式》翻译组

2003 年 10 月

致　谢

这本书开始于我在 1995 年春天编写的一小套 Web 页面。那时，作为实验性的开发阶段，我试图把自己早期使用 Java 并发特性的尝试变得更有实际意义。在演变过程中，我首先在 World Wide Web 上扩展、扩充以及删除了一些反映我和其他人有关 Java 并发编程的不断积累的经验模式，一直到现在的成书，书中涵盖了有关并发软件开发的诸多方面的模式。至今，那些网页还在，不过现在它只作为书中所阐述概念的补充材料。

在从页面到成书的过程中，许多有识之士给予了评注、建议、勘错报告以及意见交流。这些人包括：Ole Agesen、Tatsuya Aoyagi、Taranov Alexander、Moti Ben-Ari、Peter Buhr、Bruce Chapman、Il-Hyung Cho、Colin Cooper、Kelly Davis、Bruce Eckel、Yacov Eckel、Saleh Elmohamed、Ed Falis、Randy Farmer、Glenn Goldstein、David Hanson、Jyrki Heikkinen、Alain Hsiung、Jerry James、Johannes Johannsen、Istvan Kiss、Ross Knippel、Bil Lewis、Sheng Liang、Jonathan Locke、Steve MacDonald、Hidehiko Masuhara、Arnulf Mester、Mike Mills、Trevor Morris、Bill Pugh、Andrew Purshottam、Simon Roberts、John Rose、Rodney Ryan、Joel Rosi-Schwartz、Miles Sabin、Aamod Sane、Beverly Sanders、Doug Schmidt、Kevin Shank、Yukari Shirota、David Spitz、David Stoutamire、Henry Story、Sumana Srinivasan、Satish Subramanian、Jeff Swartz、Patrick Thompson、Volker Turau、Dennis Ulrich、Cees Vissar、Bruce Wallace、Greg Wilson、Grant Woodside、Steve Yen、Dave Yost，以及其他通过匿名电子邮件发送评注的人们。

Ralph Johnson 模式研讨会的成员（尤其是 Brian Foote 和 Ian Chai）通读了一些模式的早期形式，并提出了许多改进意见。纽约城模式组（New York City Patterns Group）的 Raj Datta、Sterling Barrett 和 Philip Eskelin，硅谷模式组（Silicon Valley Patterns Group）的 Russ Rufer、Ming Kwok、Mustafa Ozgen、Edward Anderson 和 Don Chin 对第二版的早期版本也做出了宝贵贡献。

在时间很紧张的情况下，第一版和第二版书稿的正式和非正式的审阅人也都提出了有帮助的意见和建议。他们是 Ken Arnold、Josh Bloch、Joseph Bowbeer、Patrick Chan、Gary Craig、Desmond D'Souza、Bill Foote、Tim Harrison、David Henderson、Tim Lindholm、Tom May、Oscar Nierstrasz、James Robins、Greg Travis、Mark Wales、Peter Welch 和 Deborra Zukowski。需要特别感谢 Tom Cargill，他提出了很多见解和修正，而且容许在本书中阐述其特有的通知模式（Specific Notification pattern）。此外，还要特别感谢 David Holmes，除了其他贡献，他还帮助开发和扩展了在第二版中所使用的教程的素材。

没有 Sun 实验室的慷慨支持，就不会有本书。我在这里要特别感谢 Jos Marlowe 和 Steve Heller，他们为我提供了在有趣而激动人心的研究开发项目中协同工作的机会。

最后，还要感谢 Kathy、Keith 和 Colin 为这一切所付出的忍耐。

Doug Lea，1999 年 9 月

目　　录

第1章

面向对象的并发编程

本书讨论了一些可以在Java™编程语言中使用的思考、设计和实现并发程序的方法。本书讨论的大多数内容都假定你是一个有经验的开发者，熟悉面向对象（object-oriented，OO）编程，但是对于并发编程的知识了解得不多。当然，对于有着相反经历的读者，即熟悉其他语言的并发编程，但是对 Java 语言中的相关部分知之甚少的读者，也可以从此书中得到不少帮助。

本书粗分为四章（也许称为部分更为合适）。第 1 章，我们从对一些常用构件的简要介绍开始，然后为并发的面向对象的编程建立一个概念基础：并发性是如何与对象结合起来的，设计需求的结果是如何影响对象和组件的构造的，如何使用一些常用的设计模式来构建解决方案。

接下来的三章主要是围绕着如何使用（和避免）Java 编程语言所提供的三种常用的并发构件进行阐述的：

独占（Exclusion）。可以通过阻止多个并发行为间的有害干扰来维护对象状态的一致性。通常使用同步（synchronized）的方法。

状态依赖（State dependence）。是否可以触发、阻止、延迟或是恢复某些行为是由一些对象是否处在这些行为可能成功或是已经成功的状态上决定的。通常，状态依赖关系使用***监视器（monitor）***方法实现，如 Object.wait、Object.notify 和 Object.notifyAll。

创建线程（Creating threads）。使用线程(Thread)对象来创建和管理并发操作。

每章都包含一系列主要的节，而每一节又都有各自独立的主题。它们分别讲述了高级的设计准则和策略，并发构件的技术细节，封装了常用方法的工具，以及用来解决特定并发问题的设计模式等一系列问题。大多数节的结尾都附带了一份有关进阶阅读的文章列表，它们包括了所讨论主题的更多信息。本书的线上支持包含了一些在线资料的链接，还有本书的更新、勘误和代码示例。可以通过如下网址访问：

```
http://java.sun.com/Series
```

或是

```
http://gee.cs.oswego.edu/dl/cpj
```

如果你已经熟悉了这些相关的基础知识，则可以按照章节的顺序深入学习每一个主题。虽然大多数读者可能更希望根据各自不同的顺序学习本书，但是，由于多数有关并发的概念和技术互相之间都是紧密关联的，因此很难完全独立地学习和理解某个单独的章节或是主题。然而，你可以使用一种广度优先的方法，在开始对某个部分进行详细深入地学习之前，

先大致浏览一下每一章（包括本章）。在有选择地阅读某些材料之后，本书后面的很多内容都可以通过与这些材料上附带的交叉引用访问到。

你可以现在就开始尝试着使用这一技巧，跳过如下的预备知识。

术语（Terminology）。本书使用标准面向对象的术语约定：程序定义了***方法***（所实现的***操作***）和***成员变量***（所表示的***属性***），这些变量保存了特定***类*** 的所有***实例***（对象）。

面向对象程序中的交互通常与程序各个部分的职责相关。一个***客户（client）***对象通常期待某个动作的执行，而一个***服务（server）***对象通常包含着用来执行这个动作的代码。在这里，术语***客户*** 和***服务*** 指的是它们的通常含义，而不是特指分布式的客户/服务器结构。一个客户可以是任何对象，它向另一个对象发送请求。而一个服务也可以是任何对象，它接受这个请求。大多数对象都同时在扮演着客户和服务这两种角色。通常情况下，我们不必关心一个对象是客户或者还是服务，或是同时属于这两种类型，它通常被称作一个***主体（host）***，而那些由它调用的对象常常被称作是***辅助对象（helper）*** 或***对等体（peer）***。同时，当讨论到如同 `obj.msg(arg)` 形式的调用方式时，接受者（变量 `obj` 所代表的对象）则被称作***目标（target）*** 对象。

本书尽力避免讨论那些偶尔出现并且与并发操作没有直接关系的类和包，同时不涉及某些特殊框架下的有关并发控制的技术细节，诸如 Enterprise JavaBeans™ 和 Servlets。但有些时候，它确实使用了一些与 Java 平台相关的商业软件和产品。

代码（Code listing）。本书包含了一系列看上去有些恼人的、短小但可运行的例子，通过不断修改这些例子来说明书中涉及到的大多数技术和模式。这样做的目的是为了可以描述得更清晰，而不是想让本书变得枯燥。如果使用其他的具体例子，这些并发构件可能会由于过于细微而迷失在那些复杂的代码之中。通过重用相同的例子，并突出设计和实现方面的主要问题，可以使得细小但关键的差别变得更加明显。同时，本书的内容还包括了简要的代码和类的片断，描述了有关实现技术，但是它们也许并不完整甚至无法编译，这样的类都会在代码清单一开始的注释部分中被指明。

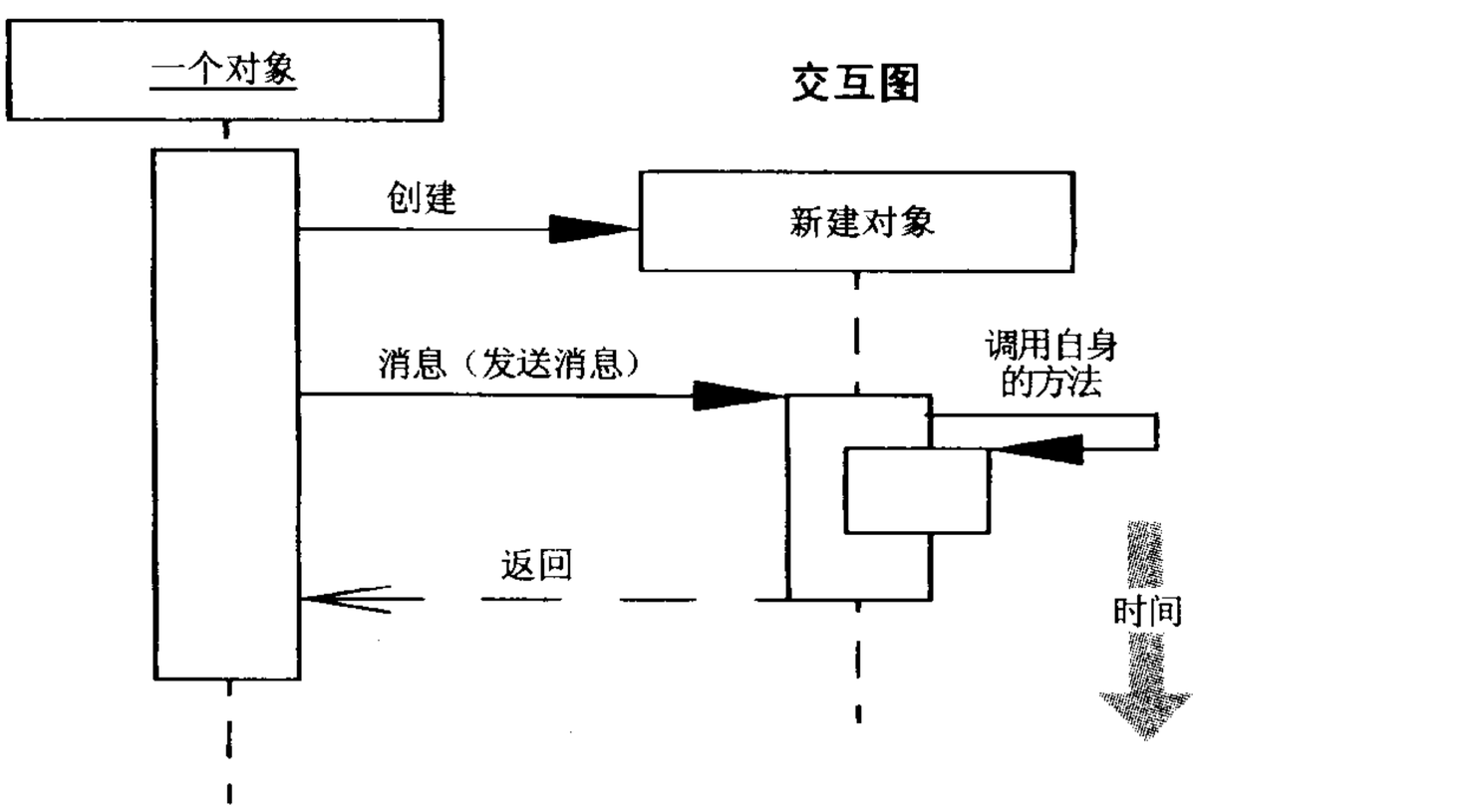
有时代码中会省略一些import语句、访问限定符，甚至某些方法和成员变量，这是因为，这些内容可以从上下文的关系中推断出来，或是不会对相应的功能产生影响。只要没有特殊的理由去限定子类的访问，`protected`限定符就被用来作为那些非公有元素的默认属性。这保证了并发类设计的可扩展性（参见 § 1.3.4和 § 3.3.3）。默认情况下，类没有访问限定符。有时，示例代码没有使用标准的格式，这样做的目的是为了使它们被安排在一页上或是突出我们感兴趣的某些主要内容。

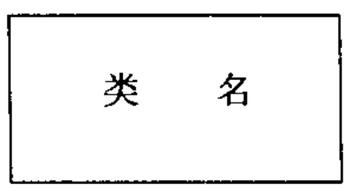
书中的所有例子的代码都可以从网上获得。书中的大多数技术和模式都用一个单独的代码实例加以说明，其中给出了它们的典型形式和用法。线上支持还包括了额外的例子，用于说明其他的一些细微变化，还有一些指向其他用法的链接。你还可以在网上找到一些更长的例子，它们更适合浏览和测试，而不是作为代码阅读。

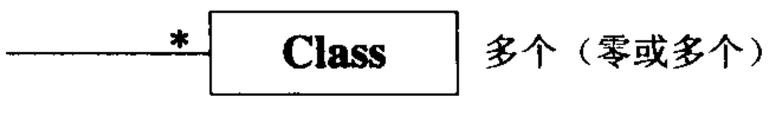
线上支持还包括了一个链接，指向一个有用的包 `util.concurrent`，其中包括一些书中讨论的实用类的产品级版本。这些代码运行在 Java 2 平台上，并且在 1.2.x 版本下进行了测试。线上支持中还包括了一些零散的讨论、说明和脚注，简要地记载了从之前版本到现有版本的修改，还有在写作本书时想到的、会在未来做出的修正，以及在实现时会遇到的一

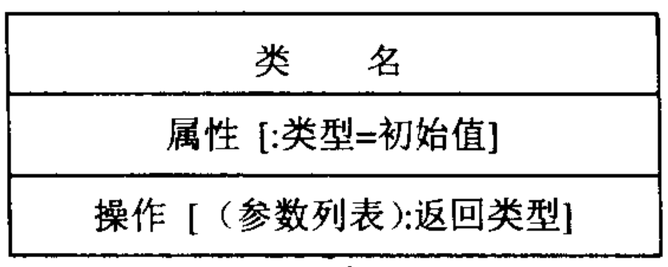
些奇怪但值得注意的问题。可以从在线支持中得到有关这些内容的最新数据。

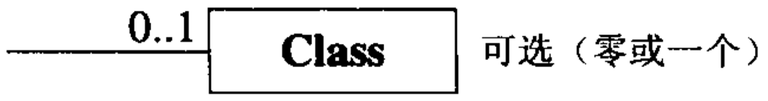
图例。我们使用标准的 UML 表示法来绘制交互图和类图（可以参考 § 1.1.3 的进一步说明）。本页的表示法图（经 Martin Fowler 允许）演示了图例在本书中的样子。除此之外，我们没有使用其他的 UML 表示法、方法和术语。

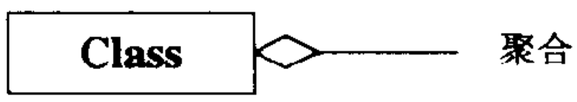


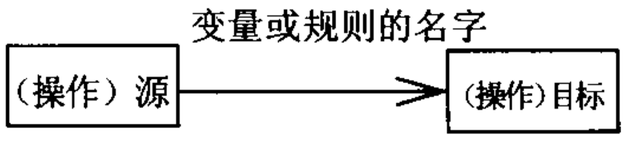


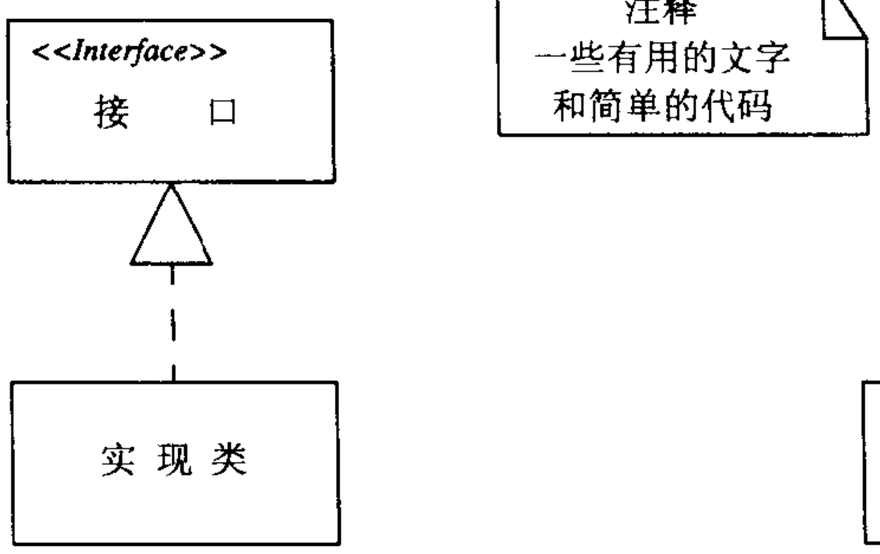


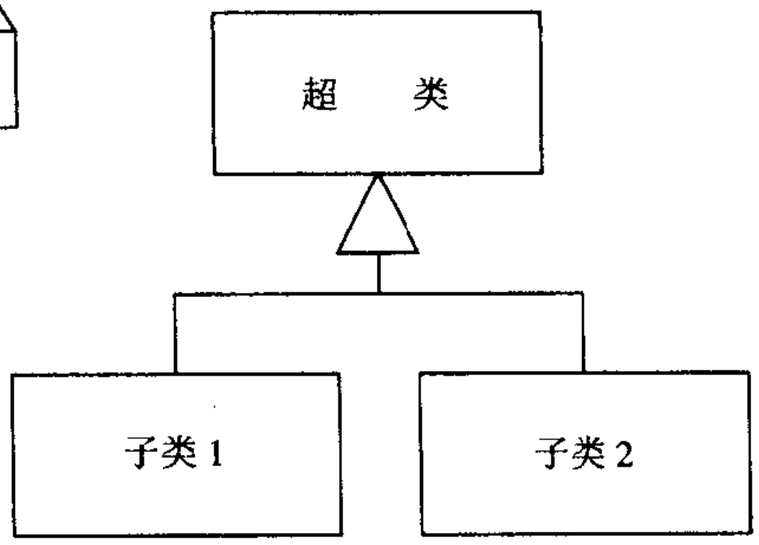






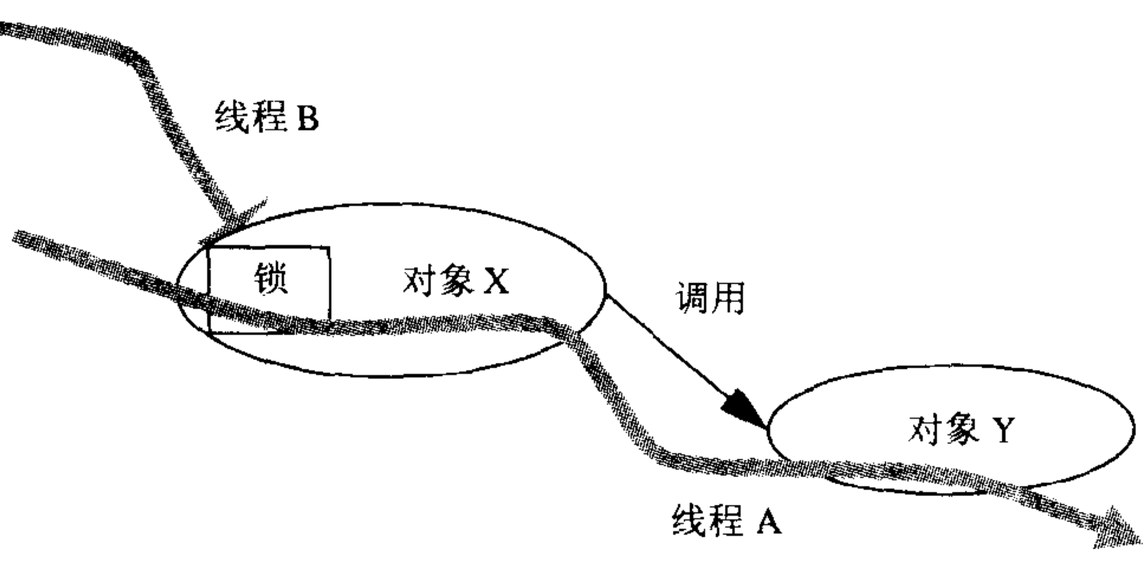






其他的大多数图例用来描绘线程时序关系（timethread），那些随意绘制的灰色曲线用来跟踪线程调用多个对象的轨迹。扁平的箭头表示阻塞。椭圆形的物体表示对象，有时当中也包含了一些有关内部特性的说明，比如锁、成员变量和一部分代码。在对象中间的那些细的线条（有时标以说明）指明了对象间的关系（通常是引用，或是潜在的调用关系）。下图使用了一个并无实际意义的例子来说明线程 A 已经取得了对象 X 上的锁，因而正在执行对象

Y 上的某些方法，这里的 Y 就是 X 的一个辅助对象。同时，当线程 B 想执行对象 X 的某个方法时将被阻塞。



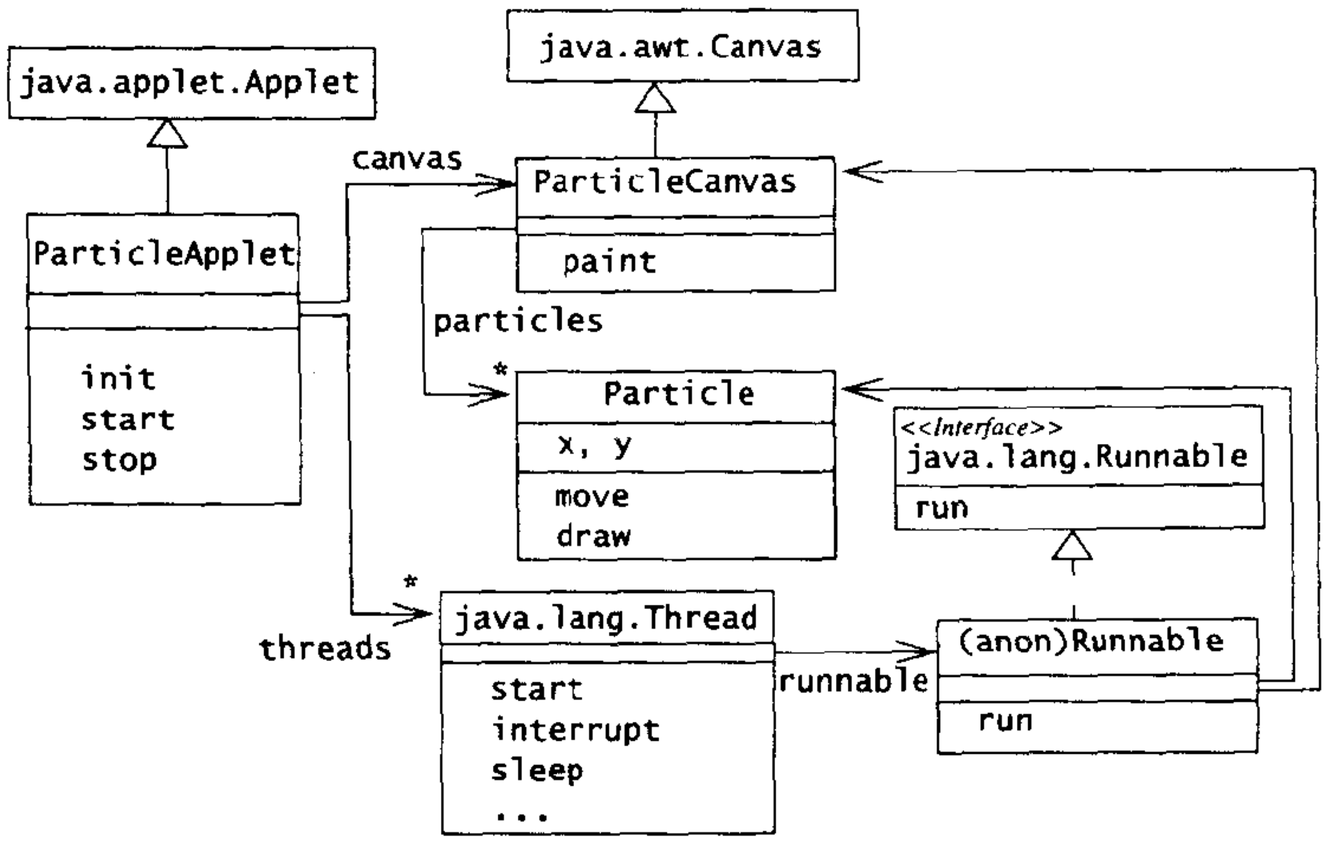
1.1 使用并发构件

本节将通过一个例子介绍几种基础的并发支持构件，然后大致地介绍 Thread 类的一些基本方法。其他的并发构件也会在引入它们的时候加以简要说明，完整的技术细节会在后面的章节中详细阐述（主要是 §2.2.1 和 §3.2.2）。通常情况下，并发程序也会使用一些 Java 编程语言所独有的特性，因此我们在使用的时候会对它们进行简要地回顾。

1.1.1 粒子 Applet

ParticleApplet 是一个用来显示随机移动粒子的 Applet。除了介绍相关的并发构件之外，这个例子也举例说明了一些在 GUI 程序中使用线程将遇到的问题。为了使得这个例子看上去更有吸引力、更真实，也许还需要进行一些润色工作。作为一个作业，你会在修改与添加的工作中得到乐趣的。

作为一个典型的GUI程序，ParticleApplet 使用多个辅助类来完成其中主要的工作。在讨论 ParticleApplet 之前，我们先大致介绍一下 Particle 和 ParticleCanvas 的结构。



1.1.1.1 粒子

类 Particle 定义了一个用于移动物体的完全假想的模型。每个粒子（particle）用它的（x,y）位置表示。同时，每个粒子包括了一个用来随机移动它自身位置的方法和一个可以通过给定的 java.awt.Graphics 对象来绘制其自身（一个小方块）的方法。

虽然 Particle 对象内部并不包含任何并发操作，但是它的方法可能被多个并发行为调用。当一个操作正在执行一个 move 操作而几乎同时另一个操作正在调用 draw 方法时，我们就需要保证 draw 方法所描绘的是 Particle 所在的准确位置。这里，我们要求 draw 方法可以在 move 方法被调用之前或是之后使用粒子的当前位置值进行绘制。举例而言，当 draw 操作使用 move 方法调用前的 y 值和 move 方法调用后的 x 值绘制一个粒子的图形，就会产生一个概念性的错误。如果我们允许这种情况出现，那 draw 方法就可能把一个粒子绘制在其并没有出现过的位置上。

我们可以通过使用 synchronized 关键字来防止这种情况的发生，synchronized 可以修饰一个方法或是一个代码块。Object 类（和它的子类）的每一个实例都会在进入一个同步（synchronized）方法前加锁，并在离开这个方法时自动地释放这把锁。对于代码块的操作也是一样，只是同步代码块需要一个参数来指明对哪一个对象加锁。最常用的参数是 this，它意味着锁住当前正在执行的方法所属的对象。当一个线程持有了一把锁后，其他线程必须阻塞，等待这个线程释放这把锁。加锁对于非同步的方法不起作用，因此即便另一个线程持有这个锁，这个非同步方法也可以执行。

通过保证**同步**在同一个对象上的方法或代码块操作的***原子性*** ***(atomicity)***，加锁机制可以同时提供多种保护措施，包括对上层和底层冲突的保护。原子操作被作为一个整体来执行，这样它们就不会被插入的其他线程的操作打断。但是，就像将在 § 1.3.2 和第 2 章所提到的那样，过多的锁操作也会产生线程活跃性问题，最终导致程序停止。我们在这里并不过多地讨论此类问题，暂时使用一些简单的规则来排除这些干扰。

- 永远只是在更新对象的成员变量时加锁。
- 永远只是在访问有可能被更新对象的成员变量时才加锁。
- 永远不要在调用其他对象的方法时加锁。

在使用这些规则时有许多例外和细节需要注意，但是对于我们的 Particle 类而言，这些暂时已经足够了。

```
import java.util.Random;

class Particle {
  protected int x;
  protected int y;
  protected final Random rng = new Random();

  public Particle(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
```

```
public synchronized void move() {
  x += rng.nextInt(10) - 5;
  y += rng.nextInt(20) - 10;
}

public void draw(Graphics g) {
  int lx, ly;
  synchronized (this) { lx = x; ly = y; }
  g.drawRect(lx, ly, 10, 10);
  }
}
```

注意：

- 我们使用 final 关键字修饰随机数生成器 rng，这样做的目的是为了确保这个成员变量的引用不会被改变。这样一来，它就不会被我们的加锁规则所影响。许多并发程序中都大量地使用了 final 关键字，以帮助将减少同步需要的设计意图自动在文档中突出说明（参见 §2.1）。
- draw 方法需要同时得到一对一致的 x 和 y 值。由于一个方法一次只能返回一个值，但是我们在这里却需要同时获得 x 和 y 的值，因此不能简单地把变量的访问封装在一个同步方法中。相反，我们使用一个同步代码块代替（可以查阅 §2.4 来寻找其他的方法）。
- 可以看到 draw 方法遵守了我们上面提到的那个重要规则，在调用其他对象上的方法时释放了锁（这个方法是 g.drawRect）。而 move 方法看似违背了我们的规则，它调用了 rng.nextInt 方法。然而在这里，我们这样做是有它的原因的，因为每一个 Particle 都包含其自身的 rng 对象，因而 rng 对象是 Particle 的一部分，所以它应该不能被看成规则中所描述的“其他对象”。§2.3 描述了适用于这种情况下的更通用的条件，并且讨论了在这种情况下许多需要考虑的因素。

1.1.1.2 ParticleCanvas

ParticleCanvas 是 java.awt.Canvas 的一个简单子类，它为所有粒子提供了一个绘制的区域。它的主要职责是，当其 paint 方法被调用时，调用所有粒子的 draw 方法。

但是，ParticleCanvas 本身并不负责创建和管理这些粒子。ParticleCanvas 对象可以被动或是主动地得到这些粒子对象。这里，我们选择前一种方法。

实例变量 particles 是一个数组，保存了已存在的 Particle 对象。这个变量在需要的时候由 Applet 负责设置，被 paint 方法使用。我们同样应用了默认的规则，使用了两个简单的、同步的 get 和 set 方法［也可以称为*访问方法（accessor）*和*指派*方法（assignment）］来访问 particles 变量，而避免了直接访问 particles 变量本身。为了简化程序并使其被正确地使用，这个 particles 变量不允许被设为 null，而是被初始化为一个空数组。

```
class ParticleCanvas extends Canvas {

  private Particle[] particles = new Particle[0];

  ParticleCanvas(int size) {
    setSize(new Dimension(size, size));
  }

  // intended to be called by applet
  protected synchronized void setParticles(Particle[] ps) {
    if (ps == null)
      throw new IllegalArgumentException("Cannot set null");

    particles = ps;
  }

  protected synchronized Particle[] getParticles() {
    return particles;
  }

  public void paint(Graphics g) { // override Canvas.paint
    Particle[] ps = getParticles();

    for (int i = 0; i < ps.length; ++i)
      ps[i].draw(g);

  }
}
```

1.1.1.3 ParticleApplet

类 Particle 和 ParticleCanvas 可以作为几个其他的程序的基础使用。但是对于 ParticleApplet 而言我们所要做的是：使每一个粒子以一种自治的“持续”的方式运动，并且更新显示这些粒子的位置。为了遵循标准 Applet 的编程约定，当外部程序调用（通常在 Web 浏览器内部被调用）Applet.start 方法之后，这些行为才会开始，直到 Applet.stop 被调用后才会结束（我们也可以通过添加按钮来允许用户来控制粒子动画的开始和结束）。

有多种方法可以实现这一切，最简单的一种是为每个粒子设置一个单独的循环，并且每一个循环都在一个单独的线程中执行。

在新线程中执行的操作必须被定义在实现了 java.lang.Runable 接口的类中。这个接口只包含了一个 run 方法。它没有参数，没有返回值，也不会抛出需要检查的异常：

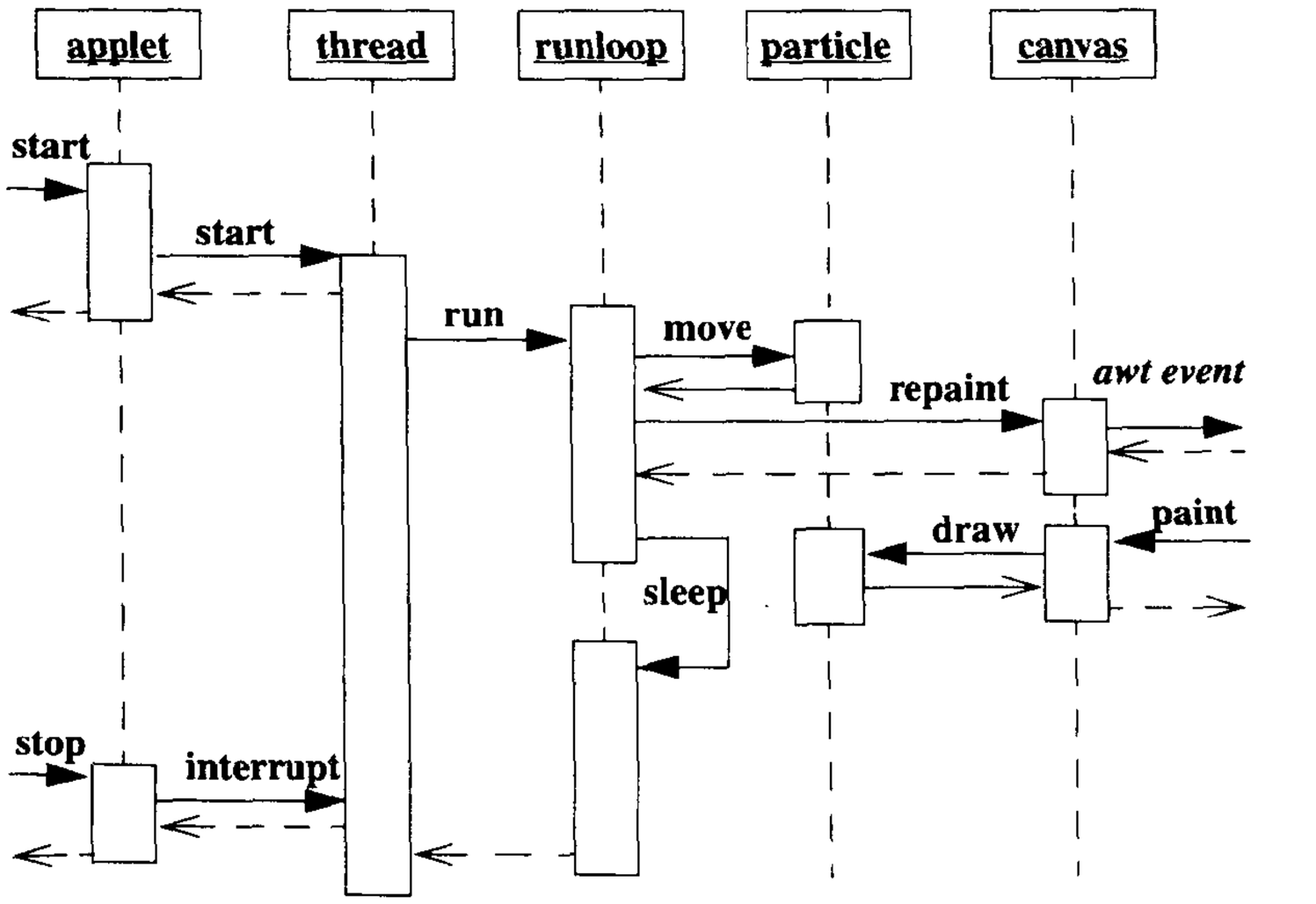
```
public interface java.lang.Runnable {
  void run();
}
```

接口(interface)封装了一系列相一致的服务和属性［广义上，它们可以被称为一个***角色（role）***］，但在接口中并没有特别指明这些功能应该由哪个对象或是代码来实现。接口比类更为抽象，因为它没有包含任何有关实现的信息和代码。它们所做的只是定义一些公共操作的***签名（signature）***（包括名字、参数、结果类型和异常），没有对调用这些接口的对象

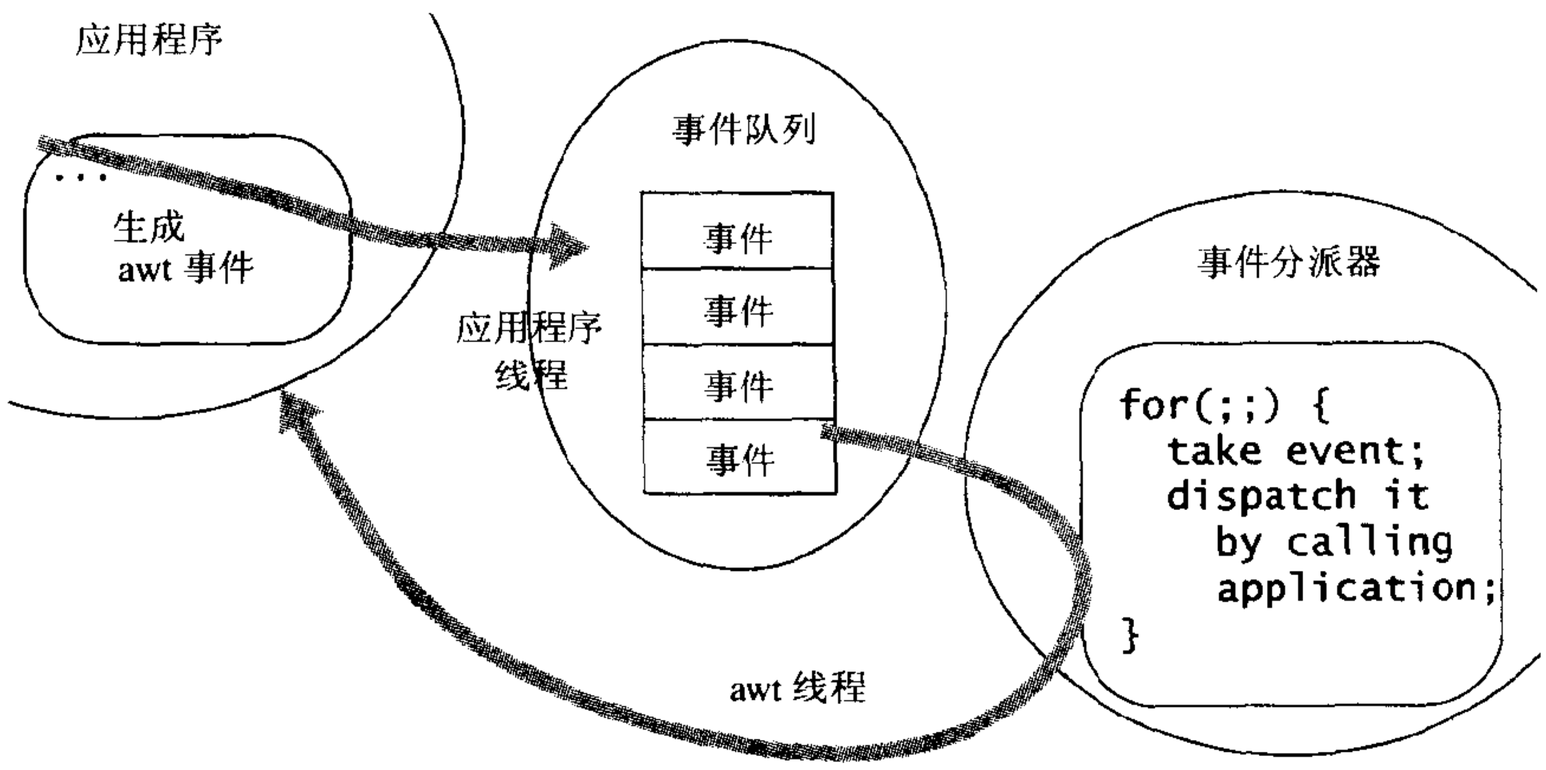
的类进行约束。除了包含一个 run 方法之外，实现 Runnable 接口的类和普通的类并无差别。

每一个 Thread 类的实例维护了执行和管理那些组成其动作的调用序列所需要的控制状态。Thread 类的最常用的构造函数接收一个 Runnable 对象作为它的参数，在线程被启动时会调用这个 Runnable 对象的 run 方法。由于任何类都可以实现 Runnable 接口，因此通常一种很方便且实用的做法就是把一个 Runnable 作为一个匿名内部类实现。

类 ParticleApplet 利用这种方法实现线程，从而使得粒子移动，并且在 Applet 结束时取消这些粒子的运动。我们可以通过覆盖标准 Applet 的 start 和 stop 方法来达到这一目的（虽然这两个方法与 Thread.start 和 Thread.stop 重名，但是两者并无任何关联）。



上面的交互图（interaction diagram）说明了 Applet 执行期间的主要消息序列。除了这些显式创建的线程之外，这个 Applet 还同 AWT 事件线程交互，这一点会在 § 4.1.4 做更多的描述。交互图右边省略的部分是一个生产者-消费者模型，它大致采用如下的形式：



```
public class ParticleApplet extends Applet {
  protected Thread[] threads = null; // null when not running
  protected final ParticleCanvas canvas
                                    = new ParticleCanvas(100);
  public void init() { add(canvas); }
protected Thread makeThread(final Particle p) { // utility
  Runnable runloop = new Runnable() {
    public void run() {
      try {
        for(;;) {
          p.move();
          canvas.repaint();
          Thread.sleep(100); // 100msec is arbitrary
        }
      }
      catch (InterruptedException e) { return; }
    }
  };
  return new Thread(runloop);
}

public synchronized void start() {
  int n = 10; // just for demo
  if (threads == null) { // bypass if already started
    Particle[] particles = new Particle[n];
    for (int i = 0; i < n; ++i)
      particles[i] = new Particle(50, 50);
    canvas.setParticles(particles);

    threads = new Thread[n];
    for (int i = 0; i < n; ++i) {
      threads[i] = makeThread(particles[i]);
      threads[i].start();
    }
  }
}

public synchronized void stop() {
  if (threads != null) { // bypass if already stopped
      for (int i = 0; i < threads.length; ++i)
        threads[i].interrupt();
      threads = null;
    }
  }
}
```

注意：

- 我们在 makeThread 的操作中定义了一个“无限”循环（可能有人更喜欢使用“while(true)”语句），只有当前线程被中断时这个循环才会退出。在每一次

循环中包含如下操作：粒子移动和告知画布进行重绘，这样这些粒子的移动才会被显示出来，在这之后的一段时间内什么都不做，从而降低工作的速度，以便配合人类的视觉感知速率。Thread.sleep 会使得当前的线程暂停，而之后它被系统时钟唤醒。

- 内部类之所以方便和实用的一个原因是：它们可以直接获取所有适当的上下文变量（在这里是 p 和 canvas）而不需要创建一个额外的类来保存它们。不过，在拥有这些便利的同时，内部类也带来一些小小的副作用：这些可以被内部类直接获得的方法参数和本地变量都必须被声明为 final，这样做的目的是为了保证这些变量的值都可以被无歧义地获得。否则，如果在 makeThread 方法内部生成了 Runnable 对象之后 p 被重新赋值，那么当 Runnable 执行的时候便会无法确定是应该使用 p 原先的值还是新赋的值。
- canvas.repaint 方法并不会直接调用 canvas.paint 方法，而是在 AWT 事件队列（java.awt.EventQueue）中添加了一个 UpdateEvent 对象。（在系统内部，这个操作会被优化，以便可以消除重复的事件。）java.awt.EventDispatchThread 的实例会异步地从队列中取出这些事件，然后分派给指定的接受者，（最终）调用 canvas.paint 方法。这个线程连同其他系统生成的线程也会存在于某些“名义上”的单线程程序中。
- 直到 Thread.start 方法被调用后，生成线程所包含的操作才会开始工作。
- 在 §3.1.2 中我们会讨论到，有很多方法可以让一个线程的工作停止。最简单的莫过于让 run 方法自然终止。但是对于无限循环的方法而言，最好的选择是使用 Thread.interrupt 方法。（通过 InterruptedException 异常）一个被中断的线程会自动从下列方法中退出：Object.wait、Thread.join 和 Thread.sleep。调用者可以通过捕获这个异常并采取适当的措施来关闭线程。在这里，runloop 中的 catch 语句只是使得 run 方法退出，从而使得这个线程结束。
- 这里的 start 和 stop 方法都声明为 synchronized，从而可以避免并发的开始或停止操作。即便这些方法需要处理大量工作（包括调用其他的对象），这里的加锁机制也可以很好地完成开始到结束和结束再开始的状态转换工作。变量 threads 的空值（null）可以方便地用来表示当前的处理状态。

1.1.2 线程机制

线程是一个独立于其他线程执行的调用序列，它可以共享底层的系统资源，如文件，并且可以访问在同一个程序中构造的其他对象（参见 §1.2.2）。java.lang.Thread 对象用来记录和控制这种行为。

每个程序至少包括一个线程——用来运行在 JVM（Java Virtual Machine，Java 虚拟机）启动时作为参数给出的类中的 main 方法。在初始化 JVM 的时候，其他的内部后台线程也会被启动。在不同的 JVM 实现中，这些线程的数目和特性并不相同。然而，所有的用户线