Java 线程

第一章　关于本教程

本教程有什么内容？

本教程研究了线程的基础知识 — 线程是什么、线程为什么有用以及怎么开始编写使用线程的简单程序。

我们还将研究更复杂的、使用线程的应用程序的基本构件 — 如何在线程之间交换数据、如何控制线程以及线程如何互相通信。

我应该学习这个教程吗？

本教程适用于拥有丰富 **Java** 语言应用知识，但又没有多少多线程或并发性经验的 **Java** 程序员。

学习完本教程之后，您应该可以编写一个使用线程的简单程序。您还应该可以阅读并理解以简单方法使用线程的程序。

关于作者

Brian Goetz 是 **developerWorks Java** 技术专区的一名定期专栏作家，而且他在过去的 **15** 年里一直是专业软件开发人员。他是 **Quiotix** 的首席顾问，这是一家位于加利福尼亚州洛斯阿尔托斯市（**Los Altos**）的软件开发和咨询公司。

在流行的业界出版物上可以看到 **Brian** 发表和即将发表的文章。

可以通过 **brian@quiotix.com** 联系 **Brian**。

第二章　线程基础

什么是线程？

几乎每种操作系统都支持进程的概念 —— 进程就是在某种程度上相互隔离的、独立运行的程序。

线程化是允许多个活动共存于一个进程中的工具。大多数现代的操作系统都支持线程，而且线程的概念以各种形式已存在了好多年。**Java** 是第一个在语言本身中显式地包含线程的主流编程语言，它没有把线程化看作是底层操作系统的工具。

有时候，线程也称作轻量级进程。就象进程一样，线程在程序中是独立的、并发的执行路径，每个线程有它自己的堆栈、自己的程序计数器和自己的局部变量。但是，与分隔的进程相比，进程中的线程之间的隔离程度要小。它们共享内存、文件句柄和其它每个进程应有的状态。

进程可以支持多个线程，它们看似同时执行，但互相之间并不同步。一个进程中的多个线程共享相同的内存地址空间，这就意味着它们可以访问相同的变量和对象，而且它们从同一堆中分配对象。尽管这让线程之间共享信息变得更容易，但您必须小心，确保它们不会妨碍同一进程里的其它线程。

Java 线程工具和 **API** 看似简单。但是，编写有效使用线程的复杂程序并不十分容易。因为有多个线程共存在相同的内存空间中并共享相同的变量，所以您必须小心，确保您的线程不会互相干扰。

每个 Java 程序都使用线程

每个 **Java** 程序都至少有一个线程 — 主线程。当一个 **Java** 程序启动时，**JVM** 会创建主线程，并在该线程中调用程序的 **main()** 方法。

JVM 还创建了其它线程，您通常都看不到它们 — 例如，与垃圾收集、对象终止和其它 **JVM** 内务处理任务相关的线程。其它工具也创建线程，如 **AWT**（抽象窗口工具箱（**Abstract Windowing Toolkit**））或 **Swing UI** 工具箱、**servlet** 容器、应用程序服务器和 **RMI**（远程方法调用（**Remote Method Invocation**））。

为什么使用线程？

在 Java 程序中使用线程有许多原因。如果您使用 Swing、servlet、RMI 或 Enterprise JavaBeans（EJB）技术，您也许没有意识到您已经在使用线程了。

使用线程的一些原因是它们可以帮助：

- 使 UI 响应更快
- 利用多处理器系统
- 简化建模
- 执行异步或后台处理

响应更快的 UI

事件驱动的 UI 工具箱（如 AWT 和 Swing）有一个事件线程，它处理 UI 事件，如击键或鼠标点击。

AWT 和 Swing 程序把事件侦听器与 UI 对象连接。当特定事件（如单击了某个按钮）发生时，这些侦听器会得到通知。事件侦听器是在 AWT 事件线程中调用的。

如果事件侦听器要执行持续很久的任务，如检查一个大文档中的拼写，事件线程将忙于运行拼写检查器，所以在完成事件侦听器之前，就不能处理额外的 UI 事件。这就会使程序看来似乎停滞了，让用户不知所措。

要避免使 UI 延迟响应，事件侦听器应该把较长的任务放到另一个线程中，这样 AWT 线程在任务的执行过程中就可以继续处理 UI 事件（包括取消正在执行的长时间运行任务的请求）。

利用多处理器系统

多处理器（MP）系统比过去更普及了。以前只能在大型数据中心和科学计算设施中才能找到它们。现在许多低端服务器系统 — 甚至是一些台式机系统 — 都有多个处理器。

现代操作系统，包括 Linux、Solaris 和 Windows NT/2000，都可以利用多个处理器并调度线程在任何可用的处理器上执行。

调度的基本单位通常是线程；如果某个程序只有一个活动的线程，它一次只能在一个处理器上运行。如果某个程序有多个活动线程，那么可以同时调度多个线程。在精心设计的程序中，使用多个线程可以提高程序吞吐量和性能。

简化建模

在某些情况下，使用线程可以使程序编写和维护起来更简单。考虑一个仿真应用程序，您要在其中模拟多个实体之间的交互作用。给每个实体一个自己的线程可以使许多仿真和对应用程序的建模大大简化。

另一个适合使用单独线程来简化程序的示例是在一个应用程序有多个独立的事件驱动的组件的时候。例如，一个应用程序可能有这样一个组件，该组件在某个事件之后用秒数倒计时，并更新屏幕显示。与其让一个主循环定期检查时间并更新显示，不如让一个线程什么也不做，一直休眠，直到某一段时间后，更新屏幕上的计数器，这样更简单，而且不容易出错。这样，主线程就根本无需担心计时器。

异步或后台处理

服务器应用程序从远程来源（如套接字）获取输入。当读取套接字时，如果当前没有可用数据，那么对 **`SocketInputStream.read()`** 的调用将会阻塞，直到有可用数据为止。

如果单线程程序要读取套接字，而套接字另一端的实体并未发送任何数据，那么该程序只会永远等待，而不执行其它处理。相反，程序可以轮询套接字，查看是否有可用数据，但通常不会使用这种做法，因为会影响性能。

但是，如果您创建了一个线程来读取套接字，那么当这个线程等待套接字中的输入时，主线程就可以执行其它任务。您甚至可以创建多个线程，这样就可以同时读取多个套接字。这样，当有可用数据时，您会迅速得到通知（因为正在等待的线程被唤醒），而不必经常轮询以检查是否有可用数据。使用线程等待套接字的代码也比轮询更简单、更不易出错。

简单，但有时有风险

虽然 Java 线程工具非常易于使用，但当您创建多线程程序时，应该尽量避免一些风险。

当多个线程访问同一数据项（如静态字段、可全局访问对象的实例字段或共享集合）时，需要确保它们协调了对数据的访问，这样它们都可以看到数据的一致视图，而且相互不会干扰另一方的更改。为了实现这个目的，Java 语言提供了两个关键字：synchronized 和 volatile。我们将稍后在本教程中研究这些关键字的用途和意义。

当从多个线程中访问变量时，必须确保对该访问正确地进行了同步。对于简单变量，将变量声明成 volatile 也许就足够了，但在大多数情况下，需要使用同步。

如果您将要使用同步来保护对共享变量的访问，那么必须确保在程序中所有访问该变量的地方都使用同步。

不要做过头

虽然线程可以大大简化许多类型的应用程序，过度使用线程可能会危及程序的性能及其可维护性。线程消耗了资源。因此，在不降低性能的情况下，可以创建的线程的数量是有限制的。

尤其在单处理器系统中，使用多个线程不会使主要消耗 CPU 资源的程序运行得更快。

示例：使用一个线程用于计时，并使用另一个线程完成工作

以下示例使用两个线程，一个用于计时，一个用于执行实际工作。主线程使用非常简单的算法计算素数。

在它启动之前，它创建并启动一个计时器线程，这个线程会休眠十秒钟，然后设置一个主线程要检查的标志。十秒钟之后，主线程将停止。请注意，共享标志被声明成 volatile。

```
/**
```

```
 * CalculatePrimes -- calculate as many primes as we can in ten seconds
 */

public class CalculatePrimes extends Thread {

   public static final int MAX_PRIMES = 1000000;
   public static final int TEN_SECONDS = 10000;

   public volatile boolean finished = false;

   public void run() {
      int[] primes = new int[MAX_PRIMES];
      int count = 0;

      for (int i=2; count<MAX_PRIMES; i++) {

         // Check to see if the timer has expired
         if (finished) {
            break;
         }

         boolean prime = true;
         for (int j=0; j<count; j++) {
            if (i % primes[j] == 0) {
               prime = false;
               break;
            }
         }

         if (prime) {
            primes[count++] = i;
            System.out.println("Found prime: " + i);
         }
      }
   }

   public static void main(String[] args) {
      CalculatePrimes calculator = new CalculatePrimes();
      calculator.start();
      try {
         Thread.sleep(TEN_SECONDS);
      }
```

```
        catch (InterruptedException e) {
            // fall through
        }

        calculator.finished = true;
    }
}
```

小结

Java 语言包含了内置在语言中的功能强大的线程工具。您可以将线程工具用于：

- 增加 GUI 应用程序的响应速度
- 利用多处理器系统
- 当程序有多个独立实体时，简化程序逻辑
- 在不阻塞整个程序的情况下，执行阻塞 I/O

当使用多个线程时，必须谨慎，遵循在线程之间共享数据的规则，我们将在共享对数据的访问中讨论这些规则。所有这些规则归结为一条基本原则：不要忘了同步。

第三章　线程的生命

创建线程

在 Java 程序中创建线程有几种方法。每个 Java 程序至少包含一个线程：主线程。其它线程都是通过 Thread 构造器或实例化继承类 Thread 的类来创建的。

Java 线程可以通过直接实例化 Thread 对象或实例化继承 Thread 的对象来创建其它线程。在线程基础中的示例（其中，我们在十秒钟之内计算尽量多的素数）中，我们通过实例化 CalculatePrimes 类型的对象（它继承了 Thread），创建了一个线程。

当我们讨论 Java 程序中的线程时，也许会提到两个相关实体：完成工作的实际线程或代表线程的 Thread 对象。正在运行的线程通常是由操作系统创建的；Thread 对象是由 Java VM 创建的，作为控制相关线程的一种方式。

创建线程和启动线程并不相同

在一个线程对新线程的 **Thread** 对象调用 **start()** 方法之前，这个新线程并没有真正开始执行。**Thread** 对象在其线程真正启动之前就已经存在了，而且其线程退出之后仍然存在。这可以让您控制或获取关于已创建的线程的信息，即使线程还没有启动或已经完成了。

通常在构造器中通过 **start()** 启动线程并不是好主意。这样做，会把部分构造的对象暴露给新的线程。如果对象拥有一个线程，那么它应该提供一个启动该线程的 **start()** 或 **init()** 方法，而不是从构造器中启动它。（请参阅参考资料，获取提供此概念更详细说明的文章链接。）

结束线程

线程会以以下三种方式之一结束：

- 线程到达其 **run()** 方法的末尾。
- 线程抛出一个未捕获到的 **Exception** 或 **Error**。
- 另一个线程调用一个弃用的 **stop()** 方法。弃用是指这些方法仍然存在，但是您不应该在新代码中使用它们，并且应该尽量从现有代码中除去它们。

当 **Java** 程序中的所有线程都完成时，程序就退出了。

加入线程

Thread API 包含了等待另一个线程完成的方法：**join()** 方法。当调用 **Thread.join()** 时，调用线程将阻塞，直到目标线程完成为止。

Thread.join() 通常由使用线程的程序使用，以将大问题划分成许多小问题，每个小问题分配一个线程。本章结尾处的示例创建了十个线程，启动它们，然后使用 **Thread.join()** 等待它们全部完成。

调度

除了何时使用 **Thread.join()** 和 **Object.wait()** 外，线程调度和执行的计时是不确定的。如果两个线程同时运行，而且都不等待，您必须假设在任何两个指令之间，其它线程都可以运行并修改程序变量。如果线程要访问其它线程可以看见的变量，如从静态字段（全局变量）直接或间接引用的数据，则必须使用同步以确保数据一致性。

在以下的简单示例中，我们将创建并启动两个线程，每个线程都打印两行到 **System.out**：

```
public class TwoThreads {

    public static class Thread1 extends Thread {
        public void run() {
            System.out.println("A");
            System.out.println("B");
        }
    }

    public static class Thread2 extends Thread {
        public void run() {
            System.out.println("1");
            System.out.println("2");
        }
    }

    public static void main(String[] args) {
        new Thread1().start();
        new Thread2().start();
    }
}
```

我们并不知道这些行按什么顺序执行，只知道“**1**”在“**2**”之前打印，以及“**A**”在“**B**”之前打印。输出可能是以下结果中的任何一种：

- 1 2 A B
- 1 A 2 B
- 1 A B 2
- A 1 2 B
- A 1 B 2
- A B 1 2

不仅不同机器之间的结果可能不同，而且在同一机器上多次运行同一程序也可能生成不同结果。永远不要假设一个线程会在另一个线程之前执行某些操作，除非您已经使用了同步以强制一个特定的执行顺序。

休眠

Thread API 包含了一个 **sleep()** 方法，它将使当前线程进入等待状态，直到过了一段指定时间，或者直到另一个线程对当前线程的 **Thread** 对象调用了 **Thread.interrupt()**，从而中断了线程。当过了指定时间后，线程又将变成可运行的，并且回到调度程序的可运行线程队列中。

如果线程是由对 **Thread.interrupt()** 的调用而中断的，那么休眠的线程会抛出 **InterruptedException**，这样线程就知道它是由中断唤醒的，就不必查看计时器是否过期。

Thread.yield() 方法就象 **Thread.sleep()** 一样，但它并不引起休眠，而只是暂停当前线程片刻，这样其它线程就可以运行了。在大多数实现中，当较高优先级的线程调用 **Thread.yield()** 时，较低优先级的线程就不会运行。

CalculatePrimes 示例使用了一个后台线程计算素数，然后休眠十秒钟。当计时器过期后，它就会设置一个标志，表示已经过了十秒。

守护程序线程

我们提到过当 **Java** 程序的所有线程都完成时，该程序就退出，但这并不完全正确。隐藏的系统线程，如垃圾收集线程和由 **JVM** 创建的其它线程会怎么样？我们没有办法停止这些线程。如果那些线程正在运行，那么 **Java** 程序怎么退出呢？

这些系统线程称作守护程序线程。**Java** 程序实际上是在它的所有非守护程序线程完成后退出的。

任何线程都可以变成守护程序线程。可以通过调用 **Thread.setDaemon()** 方法来指明某个线程是守护程序线程。您也许想要使用守护程序线程作为在程序中创建的后台线程，如计时器线程或其它延迟的事件线程，只有当其它非守护程序线程正在运行时，这些线程才有用。

示例：用多个线程分解大任务

在这个示例中，**TenThreads** 显示了一个创建了十个线程的程序，每个线程都执行一部分工作。该程序等待所有线程全部完成，然后收集结果。

```
/**
 * Creates ten threads to search for the maximum value of a large matrix.
 * Each thread searches one portion of the matrix.
 */
public class TenThreads {

   private static class WorkerThread extends Thread {
      int max = Integer.MIN_VALUE;
      int[] ourArray;

      public WorkerThread(int[] ourArray) {
         this.ourArray = ourArray;
      }

      // Find the maximum value in our particular piece of the array
      public void run() {
         for (int i = 0; i < ourArray.length; i++)
            max = Math.max(max, ourArray[i]);
      }

      public int getMax() {
         return max;
      }
   }

   public static void main(String[] args) {
      WorkerThread[] threads = new WorkerThread[10];
      int[][] bigMatrix = getBigHairyMatrix();
```

```
        int max = Integer.MIN_VALUE;

        // Give each thread a slice of the matrix to work with
        for (int i=0; i < 10; i++) {
            threads[i] = new WorkerThread(bigMatrix[i]);
            threads[i].start();
        }

        // Wait for each thread to finish
        try {
            for (int i=0; i < 10; i++) {
                threads[i].join();
                max = Math.max(max, threads[i].getMax());
            }
        }
        catch (InterruptedException e) {
            // fall through
        }

        System.out.println("Maximum value was " + max);
    }
}
```

小结

就象程序一样，线程有生命周期：它们启动、执行，然后完成。一个程序或进程也许包含多个线程，而这些线程看来互相单独地执行。

线程是通过实例化 **Thread** 对象或实例化继承 **Thread** 的对象来创建的，但在对新的 **Thread** 对象调用 **start()** 方法之前，这个线程并没有开始执行。当线程运行到其 **run()** 方法的末尾或抛出未经处理的异常时，它们就结束了。

sleep() 方法可以用于等待一段特定时间；而 **join()** 方法可能用于等到另一个线程完成。

第四章　无处不在的线程

谁创建线程？

即使您从未显式地创建一个新线程，您仍可能会发现自己在使用线程。线程被从各种来源中引入到我们的程序中。

有许多工具可以为您创建线程，如果要使用这些工具，应该了解线程如何交互，以及如何防止线程互相干扰。

AWT 和 Swing

任何使用 AWT 或 Swing 的程序都必须处理线程。AWT 工具箱创建单个线程，用于处理 UI 事件，任何由 AWT 事件调用的事件侦听器都在 AWT 事件线程中执行。

您不仅必须关心同步对事件侦听器和其它线程之间共享的数据项的访问，而且还必须找到一种方法，让由事件侦听器触发的长时间运行任务（如在大文档中检查拼写或在文件系统中搜索一个文件） 在后台线程中运行，这样当该任务运行时，UI 就不会停滞了（这可能还会阻止用户取消操作）。这样做的一个好的框架示例是 SwingWorker 类（请参阅参考资料）。

AWT 事件线程并不是守护程序线程；这就是通常使用 System.exit() 结束 AWT 和 Swing 应用程序的原因。

使用 TimerTask

JDK 1.3 中，TimerTask 工具被引入到 Java 语言。这个便利的工具让您可以稍后在某个时间执行任务（例如，即从现在起十秒后运行一次任务），或者定期执行任务（即，每隔十秒运行任务）。

实现 Timer 类非常简单：它创建一个计时器线程，并且构建一个按执行时间排序的等待事件队列。

TimerTask 线程被标记成守护程序线程，这样它就不会阻止程序退出。

因为计时器事件是在计时器线程中执行，所以必须确保正确同步了针对计时器任务中使用的任何数据项的访问。

在 **CalculatePrimes** 示例中，并没有让主线程休眠，我们可以使用 **TimerTask**，方法如下：

```
public static void main(String[] args) {
    Timer timer = new Timer();

    final CalculatePrimes calculator = new CalculatePrimes();
    calculator.start();

    timer.schedule(
           new TimerTask() {
               public void run()
               {
                   calculator.finished = true;
               }
           }, TEN_SECONDS);
}
```

servlet 和 JavaServer Pages 技术

servlet 容器创建多个线程，在这些线程中执行 **servlet** 请求。作为 **servlet** 编写者，您不知道（也不应该知道）您的请求会在什么线程中执行；如果同时有多个对相同 **URL** 的请求入站，那么同一个 **servlet** 可能会同时在多个线程中是活动的。

当编写 **servlet** 或 **JavaServer Pages (JSP)** 文件时，必须始终假设可以在多个线程中并发地执行同一个 **servlet** 或 **JSP** 文件。必须适当同步 **servlet** 或 **JSP** 文件访问的任何共享数据；这包括 **servlet** 对象本身的字段。

实现 RMI 对象

RMI 工具可以让您调用对在其它 JVM 中运行的对象进行的操作。当调用远程方法时，RMI 编译器创建的 RMI 存根会打包方法参数，并通过网络将它们发送到远程系统，然后远程系统会将它们解包并调用远程方法。

假设您创建了一个 RMI 对象，并将它注册到 RMI 注册表或者 Java 命名和目录接口（Java Naming and Directory Interface (JNDI)）名称空间。当远程客户机调用其中的一个方法时，该方法会在什么线程中执行呢？

实现 RMI 对象的常用方法是继承 UnicastRemoteObject。在构造 UnicastRemoteObject 时，会初始化用于分派远程方法调用的基础结构。这包括用于接收远程调用请求的套接字侦听器，和一个或多个执行远程请求的线程。

所以，当接收到执行 RMI 方法的请求时，这些方法将在 RMI 管理的线程中执行。

小结

线程通过几种机制进入 Java 程序。除了用 Thread 构造器中显式创建线程之外，还可以用许多其它机制创建线程：

- AWT 和 Swing
- RMI
- java.util.TimerTask 工具
- servlet 和 JSP 技术

第五章　共享对数据的访问

共享变量

要使多个线程在一个程序中有用，它们必须有某种方法可以互相通信或共享它们的结果。

让线程共享其结果的最简单方法是使用共享变量。它们还应该使用同步来确保值从一个线程正确传播到另一个线程，以及防止当一个线程正在更新一些相关数据项时，另一个线程看到不一致的中间结果。

线程基础中计算素数的示例使用了一个共享布尔变量，用于表示指定的时间段已经过去了。这说明了在线程间共享数据最简单的形式是：轮询共享变量以查看另一个线程是否已经完成执行某项任务。

存在于同一个内存空间中的所有线程

正如前面讨论过的，线程与进程有许多共同点，不同的是线程与同一进程中的其它线程共享相同的进程上下文，包括内存。这非常便利，但也有重大责任。只要访问共享变量（静态或实例字段），线程就可以方便地互相交换数据，但线程还必须确保它们以受控的方式访问共享变量，以免它们互相干扰对方的更改。

任何线程可以访问所有其作用域内的变量，就象主线程可以访问该变量一样。素数示例使用了一个公用实例字段，叫做 **finished**，用于表示已经过了指定的时间。当计时器过期时，一个线程会写这个字段；另一个线程会定期读取这个字段，以检查它是否应该停止。注：这个字段被声明成 **volatile**，这对于这个程序的正确运行非常重要。在本章的后面，我们将看到原因。

受控访问的同步

为了确保可以在线程之间以受控方式共享数据，**Java** 语言提供了两个关键字：**synchronized** 和 **volatile**。

Synchronized 有两个重要含义：它确保了一次只有一个线程可以执行代码的受保护部分（互斥，**mutual exclusion** 或者说 **mutex**），而且它确保了一个线程更改的数据对于其它线程是可见的（更改的可见性）。

如果没有同步，数据很容易就处于不一致状态。例如，如果一个线程正在更新两个相关值（比如，粒子的位置和速率），而另一个线程正在读取这两个值，有可能在第一个线程只写了一个值，还没有写另一个值的时候，调度第二个线程运行，这样它就会看到一个旧值和一个新值。同步让我们可以定义必须原子地运行的代码块，这样对于其他线程而言，它们要么都执行，要么都不执行。

同步的原子执行或互斥方面类似于其它操作环境中的临界段的概念。

确保共享数据更改的可见性

同步可以让我们确保线程看到一致的内存视图。

处理器可以使用高速缓存加速对内存的访问（或者编译器可以将值存储到寄存器中以便进行更快的访问）。在一些多处理器体系结构上，如果在一个处理器的高速缓存中修改了内存位置，没有必要让其它处理器看到这一修改，直到刷新了写入器的高速缓存并且使读取器的高速缓存无效。

这表示在这样的系统上，对于同一变量，在两个不同处理器上执行的两个线程可能会看到两个不同的值！这听起来很吓人，但它却很常见。它只是表示在访问其它线程使用或修改的数据时，必须遵循某些规则。

Volatile 比同步更简单，只适合于控制对基本变量（整数、布尔变量等）的单个实例的访问。当一个变量被声明成 **volatile**，任何对该变量的写操作都会绕过高速缓存，直接写入主内存，而任何对该变量的读取也都绕过高速缓存，直接取自主内存。这表示所有线程在任何时候看到的 **volatile** 变量值都相同。

如果没有正确的同步，线程可能会看到旧的变量值，或者引起其它形式的数据损坏。

用锁保护的原子代码块

Volatile 对于确保每个线程看到最新的变量值非常有用，但有时我们需要保护比较大的代码片段，如涉及更新多个变量的片段。

同步使用监控器（**monitor**）或锁的概念，以协调对特定代码块的访问。

每个 **Java** 对象都有一个相关的锁。同一时间只能有一个线程持有 **Java** 锁。当线程进入 **synchronized** 代码块时，线程会阻塞并等待，直到锁可用，当它可用时，就会获得这个锁，然后执行代码块。当控制退出受保护的代码块时，即到达了代码块末尾或者抛出了没有在 **synchronized** 块中捕获的异常时，它就会释放该锁。

这样，每次只有一个线程可以执行受给定监控器保护的代码块。从其它线程的角度看，该代码块可以看作是原子的，它要么全部执行，要么根本不执行。

简单的同步示例

使用 **synchronized** 块可以让您将一组相关更新作为一个集合来执行，而不必担心其它线程中断或看到计算的中间结果。以下示例代码将打印“**1 0**”或“**0 1**”。如果没有同步，它还会打印“**1 1**”（或“**0 0**”，随便您信不信）。

```
public class SyncExample {
  private static lockObject = new Object();
  private static class Thread1 extends Thread {
    public void run() {
      synchronized (lockObject) {
        x = y = 0;
        System.out.println(x);
      }
    }
  }

  private static class Thread2 extends Thread {
    public void run() {
      synchronized (lockObject) {
```

```
      x = y = 1;
      System.out.println(y);
    }
  }
 }

 public static void main(String[] args) {
  new Thread1().run();
  new Thread2().run();
 }
}
```

在这两个线程中都必须使用同步，以便使这个程序正确工作。

Java 锁定

Java 锁定合并了一种互斥形式。每次只有一个线程可以持有锁。锁用于保护代码块或整个方法，必须记住是锁的身份保护了代码块，而不是代码块本身，这一点很重要。一个锁可以保护许多代码块或方法。

反之，仅仅因为代码块由锁保护并不表示两个线程不能同时执行该代码块。它只表示如果两个线程正在等待相同的锁，则它们不能同时执行该代码。

在以下示例中，两个线程可以同时不受限制地执行 **setLastAccess()** 中的 **synchronized** 块，因为每个线程有一个不同的 **thingie** 值。因此，**synchronized** 代码块受到两个正在执行的线程中不同锁的保护。

```
public class SyncExample {
 public static class Thingie {

  private Date lastAccess;

  public synchronized void setLastAccess(Date date) {
   this.lastAccess = date;
  }
 }
```

```
  public static class MyThread extends Thread {
    private Thingie thingie;

    public MyThread(Thingie thingie) {
      this.thingie = thingie;
    }

    public void run() {
      thingie.setLastAccess(new Date());
    }
  }

  public static void main() {
    Thingie thingie1 = new Thingie(),
      thingie2 = new Thingie();

    new MyThread(thingie1).start();
    new MyThread(thingie2).start();
  }
}
```

同步的方法

创建 **synchronized** 块的最简单方法是将方法声明成 **synchronized**。这表示在进入方法主体之前，调用者必须获得锁：

```
public class Point {
  public synchronized void setXY(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

对于普通的 **synchronized** 方法，这个锁是一个对象，将针对它调用方法。对于静态 **synchronized** 方法，这个锁是与 **Class** 对象相关的监控器，在该对象中声明了方法。

仅仅因为 **setXY()** 被声明成 **synchronized** 并不表示两个不同的线程不能同时执行 **setXY()**，只要它们调用不同的 **Point** 实例的 **setXY()** 就可同时执行。对于一个 **Point** 实例，一次只能有一个线程执行 **setXY()**，或 **Point** 的任何其它 **synchronized** 方法。

同步的块

synchronized 块的语法比 **synchronized** 方法稍微复杂一点，因为还需要显式地指定锁要保护哪个块。**Point** 的以下版本等价于前一页中显示的版本：

```
public class Point {
  public void setXY(int x, int y) {
    synchronized (this) {
      this.x = x;
      this.y = y;
    }
  }
}
```

使用 **this** 引用作为锁很常见，但这并不是必需的。这表示该代码块将与这个类中的 **synchronized** 方法使用同一个锁。

由于同步防止了多个线程同时执行一个代码块，因此性能上就有问题，即使是在单处理器系统上。最好在尽可能最小的需要保护的代码块上使用同步。

访问局部（基于堆栈的）变量从来不需要受到保护，因为它们只能被自己所属的线程访问。

大多数类并没有同步

因为同步会带来小小的性能损失，大多数通用类，如 **java.util** 中的 **Collection** 类，不在内部使用同步。这表示在没有附加同步的情况下，不能在多个线程中使用诸如 **HashMap** 这样的类。

通过每次访问共享集合中的方法时使用同步，可以在多线程应用程序中使用 **Collection** 类。对于任何给定的集合，每次必须用同一个锁进行同步。通常可以选择集合对象本身作为锁。

下一页中的示例类 **SimpleCache** 显示了如何使用 **HashMap** 以线程安全的方式提供高速缓存。但是，通常适当的同步并不只是意味着同步每个方法。

Collections 类提供了一组便利的用于 **List**、**Map** 和 **Set** 接口的封装器。您可以用 **Collections.synchronizedMap** 封装 **Map**，它将确保所有对该映射的访问都被正确同步。

如果类的文档没有说明它是线程安全的，那么您必须假设它不是。

示例：简单的线程安全的高速缓存

如以下代码样本所示，**SimpleCache.java** 使用 **HashMap** 为对象装入器提供了一个简单的高速缓存。**load()** 方法知道怎样按对象的键装入对象。在一次装入对象之后，该对象就被存储到高速缓存中，这样以后的访问就会从高速缓存中检索它，而不是每次都全部地装入它。对共享高速缓存的每个访问都受到 **synchronized** 块保护。由于它被正确同步，所以多个线程可以同时调用 **getObject** 和 **clearCache** 方法，而没有数据损坏的风险。

```
public class SimpleCache {
  private final Map cache = new HashMap();

  public Object load(String objectName) {
    // load the object somehow
  }

  public void clearCache() {
    synchronized (cache) {
      cache.clear();
    }
  }

  public Object getObject(String objectName) {
    synchronized (cache) {
      Object o = cache.get(objectName);
```

```
      if (o == null) {
        o = load(objectName);
        cache.put(objectName, o);
      }
    }

    return o;
  }
}
```

小结

由于线程执行的计时是不确定的，我们需要小心，以控制线程对共享数据的访问。否则，多个并发线程会互相干扰对方的更改，从而损坏数据，或者其它线程也许不能及时看到对共享数据的更改。

通过使用同步来保护对共享变量的访问，我们可以确保线程以可预料的方式与程序变量进行交互。

每个 **Java** 对象都可以充当锁，**synchronized** 块可以确保一次只有一个线程执行由给定锁保护的 **synchronized** 代码。

第六章　同步详细信息

互斥

在共享对数据的访问中，我们讨论了 **synchronized** 块的特征，并在实现典型互斥锁（即，互斥或临界段）时说明了它们，其中每次只有一个线程可以执行受给定锁保护的代码块。

互斥是同步所做工作的重要部分，但同步还有其它几种特征，这些特征对于在多处理器系统上取得正确结果非常重要。

可见性

除了互斥，同步（如 **volatile**）强制某些可见性约束。当对象获取锁时，它首先使自己的高速缓存无效，这样就可以保证直接从主内存中装入变量。

同样，在对象释放锁之前，它会刷新其高速缓存，强制使已做的任何更改都出现在主内存中。

这样，会保证在同一个锁上同步的两个线程看到在 **synchronized** 块内修改的变量的相同值。

什么时候必须同步？

要跨线程维护正确的可见性，只要在几个线程之间共享非 **final** 变量，就必须使用 **synchronized**（或 **volatile**）以确保一个线程可以看见另一个线程做的更改。

可见性同步的基本规则是在以下情况中必须同步：

- 读取上一次可能是由另一个线程写入的变量
- 写入下一次可能由另一个线程读取的变量

用于一致性的同步

除了用于可见性的同步，从应用程序角度看，您还必须用同步来确保一致性得到了维护。当修改多个相关值时，您想要其它线程原子地看到这组更改 — 要么看到全部更改，要么什么也看不到。这

适用于相关数据项（如粒子的位置和速率）和元数据项（如链表中包含的数据值和列表自身中的数据项的链）。

考虑以下示例，它实现了一个简单（但不是线程安全的）的整数堆栈：

```
public class UnsafeStack {
  public int top = 0;
  public int[] values = new int[1000];

  public void push(int n) {
    values[top++] = n;
  }

  public int pop() {
    return values[--top];
  }
}
```

如果多个线程试图同时使用这个类，会发生什么？这可能是个灾难。因为没有同步，多个线程可以同时执行 **push()** 和 **pop()**。如果一个线程调用 **push()**，而另一个线程正好在递增了 **top** 并要把它用作 **values** 的下标之间调用 **push()**，会发生什么？结果，这两个线程会把它们的新值存储到相同的位置！当多个线程依赖于数据值之间的已知关系，但没有确保只有一个线程可以在给定时间操作那些值时，可能会发生许多形式的数据损坏，而这只是其中之一。

对于这种情况，补救办法很简单：同步 **push()** 和 **pop()** 这两者，您将防止线程执行相互干扰。

请注意，使用 **volatile** 还不够 — 需要使用 **synchronized** 来确保 **top** 和 **values** 之间的关系保持一致。

递增共享计数器

通常，如果正在保护一个基本变量（如一个整数），有时只使用 **volatile** 就可以侥幸过关。但是，如果变量的新值派生自以前的值，就必须使用同步。为什么？考虑这个类：

```
public class Counter {
  private int counter = 0;

  public int  get()     { return counter; }
  public void set(int n) { counter = n; }
  public void increment() {
    set(get() + 1);
  }
}
```

当我们要递增计数器时，会发生什么？请看 **increment()** 的代码。它很清楚，但不是线程安全的。如果两个线程试图同时执行 **increment()**，会发生什么？计数器也许会增加 **1**，也许增加 **2**。令人惊奇的是，把 **counter** 标记成 **volatile** 没有帮助，使 **get()** 和 **set()** 都变成 **synchronized** 也没有帮助。

设想计数器是零，而两个线程同时执行递增操作代码。这两个线程会调用 **Counter.get()**，并且看到计数器是零。现在两个线程都对它加一，然后调用 **Counter.set()**。如果我们的计时不太凑巧，那么这两个线程都看不到对方的更新，即使 **counter** 是 **volatile**，或者 **get()** 和 **set()** 是 **synchronized**。现在，即使计数器递增了两次，得到的值也许只是一，而不是二。

要使递增操作正确运行，不仅 **get()** 和 **set()** 必须是 **synchronized**，而且 **increment()** 也必需是 **synchronized**！否则，调用 **increment()** 的线程可能会中断另一个调用 **increment()** 的线程。如果您不走运，最终结果将会是计数器只增加了一次，不是两次。同步 **increment()** 防止了这种情况的发生，因为整个递增操作是原子的。

当循环遍历 **Vector** 的元素时，同样如此。即使同步了 **Vector** 的方法，但在循环遍历时，**Vector** 的内容仍然会更改。如果要确保 **Vector** 的内容在循环遍历时不更改，必须同步整个代码块。

不变性和 final 字段

许多 **Java** 类，包括 **String**、**Integer** 和 **BigDecimal**，都是不可改变的：一旦构造之后，它们的状态就永远不会更改。如果某个类的所有字段都被声明成 **final**，那么这个类就是不可改变的。（实

际上，许多不可改变的类都有非 **final** 字段，用于高速缓存以前计算的方法结果，如 **String.hashCode()**，但调用者看不到这些字段。）

不可改变的类使并发编程变得非常简单。因为不能更改它们的字段，所以就不需要担心把状态的更改从一个线程传递到另一个线程。在正确构造了对象之后，可以把它看作是常量。

同样，**final** 字段对于线程也更友好。因为 **final** 字段在初始化之后，它们的值就不能更改，所以当在线程之间共享 **final** 字段时，不需要担心同步访问。

什么时候不需要同步

在某些情况中，您不必用同步来将数据从一个线程传递到另一个，因为 **JVM** 已经隐含地为您执行同步。这些情况包括：

- 由静态初始化器（在静态字段上或 **static{}** 块中的初始化器）初始化数据时
- 访问 **final** 字段时
- 在创建线程之前创建对象时
- 线程可以看见它将要处理的对象时

死锁

只要您拥有多个进程，而且它们要争用对多个锁的独占访问，那么就有可能发生死锁。如果有一组进程或线程，其中每个都在等待一个只有其它进程或线程才可以执行的操作，那么就称它们被死锁了。

最常见的死锁形式是当线程 **1** 持有对象 **A** 上的锁，而且正在等待与 **B** 上的锁，而线程 **2** 持有对象 **B** 上的锁，却正在等待对象 **A** 上的锁。这两个线程永远都不会获得第二个锁，或者释放第一个锁。它们只会永远等待下去。

要避免死锁，应该确保在获取多个锁时，在所有的线程中都以相同的顺序获取锁。

性能考虑事项

关于同步的性能代价有许多说法 — 其中有许多是错的。同步，尤其是争用的同步，确实有性能问题，但这些问题并没有象人们普遍怀疑的那么大。

许多人都使用别出心裁但不起作用的技巧以试图避免必须使用同步，但最终都陷入了麻烦。一个典型的示例是双重检查锁定模式（请参阅参考资料，其中有几篇文章讲述了这种模式有什么问题）。这种看似无害的结构据说可以避免公共代码路径上的同步，但却令人费解地失败了，而且所有试图修正它的尝试也失败了。

在编写并发代码时，除非看到性能问题的确凿证据，否则不要过多考虑性能。瓶颈往往出现在我们最不会怀疑的地方。投机性地优化一个也许最终根本不会成为性能问题的代码路径 — 以程序正确性为代价 — 是一桩赔本的生意。

同步准则

当编写 **synchronized** 块时，有几个简单的准则可以遵循，这些准则在避免死锁和性能危险的风险方面大有帮助：

- **使代码块保持简短。Synchronized** 块应该简短 — 在保证相关数据操作的完整性的同时，尽量简短。把不随线程变化的预处理和后处理移出 **synchronized** 块。
- **不要阻塞。**不要在 **synchronized** 块或方法中调用可能引起阻塞的方法，如 **InputStream.read()**。
- **在持有锁的时候，不要对其它对象调用方法。**这听起来可能有些极端，但它消除了最常见的死锁源头。

第七章　其它线程 API 详细信息

wait()、notify() 和 notifyAll() 方法

除了使用轮询（它可能消耗大量 CPU 资源，而且具有计时不精确的特征），Object 类还包括一些方法，可以让线程相互通知事件的发生。

Object 类定义了 wait()、notify()和 notifyAll()方法。要执行这些方法，必须拥有相关对象的锁。

Wait() 会让调用线程休眠，直到用 Thread.interrupt() 中断它、过了指定的时间、或者另一个线程用 notify() 或 notifyAll() 唤醒它。

当对某个对象调用 notify() 时，如果有任何线程正在通过 wait() 等待该对象，那么就会唤醒其中一个线程。当对某个对象调用 notifyAll() 时，会唤醒所有正在等待该对象的线程。

这些方法是更复杂的锁定、排队和并发性代码的构件。但是，notify() 和 notifyAll() 的使用很复杂。尤其是，使用 notify() 来代替 notifyAll() 是有风险的。除非您确实知道正在做什么，否则就使用 notifyAll()。

与其使用 wait() 和 notify() 来编写您自己的调度程序、线程池、队列和锁，倒不如使用 util.concurrent 包（请参阅参考资料），这是一个被广泛使用的开放源码工具箱，里面都是有用的并发性实用程序。JDK 1.5 将包括 java.util.concurrent 包；它的许多类都派生自 util.concurrent。

线程优先级

Thread API 让您可以将执行优先级与每个线程关联起来。但是，这些优先级如何映射到底层操作系统调度程序取决于实现。在某些实现中，多个 — 甚至全部 — 优先级可能被映射成相同的底层操作系统优先级。

在遇到诸如死锁、资源匮乏或其它意外的调度特征问题时，许多人都想要调整线程优先级。但是，通常这样只会把问题移到别的地方。大多数程序应该完全避免更改线程优先级。

线程组

ThreadGroup 类原本旨在用于把线程集合构造成组。但是，结果证明 **ThreadGroup** 并没有那样有用。您最好只使用 **Thread** 中的等价方法。

ThreadGroup 确实提供了一个有用的功能部件（**Thread** 中目前还没有）：**uncaughtException()** 方法。线程组中的某个线程由于抛出了未捕获的异常而退出时，会调用 **ThreadGroup.uncaughtException()** 方法。这就让您有机会关闭系统、将一条消息写到日志文件或者重新启动失败的服务。

SwingUtilities

虽然 **SwingUtilities** 类不是 **Thread API** 的一部分，但还是值得简单提一下。

正如前面提到的，**Swing** 应用程序有一个 **UI** 线程（有时叫称为事件线程），所有 **UI** 活动都必须在这个线程中发生。有时，另一个线程也许想要更新屏幕上某样东西的外观，或者触发 **Swing** 对象上的一个事件。

SwingUtilities.invokeLater() 方法可以让您将 **Runnable** 对象传送给它，并且在事件线程中执行指定的 **Runnable**。它的同类 **invokeAndWait()** 会在事件线程中调用 **Runnable**，但 **invokeAndWait()** 会阻塞，直到 **Runnable** 完成执行之后。

```
void showHelloThereDialog() throws Exception {
   Runnable showModalDialog = new Runnable() {
      public void run() {
         JOptionPane.showMessageDialog(myMainFrame, "Hello There");
      }
   };
   SwingUtilities.invokeLater(showModalDialog);
}
```

对于 **AWT** 应用程序，**java.awt.EventQueue** 还提供了 **invokeLater()** 和 **invokeAndWait()**。

第八章　结束语和参考资料

结束语

每个 Java 程序都使用线程，不论您知道与否。如果您正在使用 Java UI 工具箱（AWT 或 Swing）、Java Servlet、RMI、JavaServer Pages 或 Enterprise JavaBeans 技术，您可能没有意识到您正在使用线程。

在许多情况中，您可能想要显式地使用线程以提高程序的性能、响应速度或组织。这些情况包括：

- 在执行耗时较长的任务时，使用户界面的响应速度更快
- 利用多处理器系统以并行处理多个任务
- 简化仿真或基于代理的系统的建模
- 执行异步或后台处理

虽然线程 API 很简单，但编写线程安全的程序并不容易。在线程之间共享变量时，必须非常小心，以确保正确同步了对它们的读写访问。当写一个可能接下来由另一个线程读取的变量，或者读取可能由另一个线程写过的变量时，必须使用同步以确保对数据的更改在线程之间是可见的。

当使用同步保护共享变量时，必须确保不仅使用了同步，而且读取器和写入器在同一个监控器上同步。而且，如果依赖对象的状态在多个操作中保持相同，或者依赖多个变量互相保持一致（或者，与它们过去的值一致），那么必须使用同步来强制实现这一点。但简单地同步一个类中的每一个方法并不能使它变成线程安全的 — 只会使它更容易发生死锁。

参考资料

下载

- 请研究 Doug Lea 的 util.concurrent 包（http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html），它包含了许多用于构建有效并发应用程序的有用的类。

文章和教程

- “Synchronization and the Java Memory Model”（http://gee.cs.oswego.edu/dl/cpj/jmm.html）摘录自 Doug Lea 撰写的着重讨论 synchronized 实际意义的一书。
- 在“Writing multithreading Java applications”（*developerWorks*，2001 年 2 月，http://www-106.ibm.com/developerworks/library/j-thread.html）一文中，Alex Roetter 概述了 Java 多线程化中涉及的问题，并提供了常见问题的解决方案。
- Brian Goetz 撰写的“Threading lightly, Part 1: Synchronization is not the enemy”（*developerWorks*，2001 年 7 月，http://www-106.ibm.com/developerworks/library/j-threads1/）研究了如何管理并发应用程序的性能。
- Jeff Friesen 撰写的“Achieve strong performance with threads”（*JavaWorld*，2002 年 5 月，http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-java101.html）是关于使用线程的一个四部分教程。
- “Double-checked locking: Clever, but broken”（*JavaWorld*，2001 年 2 月，http://www.javaworld.com/jw-02-2001/jw-0209-double.html）详细研究了 Java Memory Model，以及在某些情况下同步失败的惊人后果。
- 线程安全性是棘手的问题。“Java theory and practice: Safe construction techniques”（*developerWorks*，2002 年 6 月，http://www-106.ibm.com/developerworks/library/j-jtp0618.html）提供了一些安全地构造对象的提示。
- 在“Threads and Swing”（http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html）中，Sun 公司的技术人员研究了安全地使用 Swing 的规则，并且引入了有用的 SwingWorker 类。

推荐书籍

- Doug Lea 的 Concurrent Programming in Java，第二版（Addison-Wesley，1999 年，http://www.amazon.com/exec/obidos/ASIN/0201310090/none0b69）是关于围绕在 Java 应用程序中多线程编程的难解问题的权威书籍。
- Paul Hyde 的 Java Thread Programming（http://www.amazon.com/exec/obidos/ASIN/0672315858/none0b69）是关于许多现实世界中多线程问题的好教程和参考大全。
- Allen Holub 的 Taming Java Threads（http://www.amazon.com/exec/obidos/ASIN/1893115100/none0b69）一书是 Java 线程编程难题的有趣介绍。

其它参考资料

- util.concurrent 包正在根据 Java Community Process JSR 166（http://www.jcp.org/jsr/detail/166.jsp）进行正式化，以便包含在 JDK 的 1.5 发行版中。
- Foxtrot 项目（http://foxtrot.sourceforge.net/）是另一个在 Swing 应用程序中使用线程的方法，它可能更简单。
- 在 developerWorks Java 技术专区（http://www-106.ibm.com/developerworks/java/）中，您会找到几百篇关于 Java 编程的各个方面的文章。
- 请访问 developerWorks Java 技术教程页面（http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle?OpenDocument&Count=500/），以获取 developerWorks 中免费教程的完整清单。