在Java中，每个线程都是[Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)类的实例。并发应用中一般有两种不同的线程创建策略。

* 直接控制线程的创建和管理，每当应用程序需要执行一个异步任务的时候就为其创建一个线程
* 将线程的管理从应用程序中抽象出来作为执行器，应用程序将任务传递给执行器，有执行器负责执行。

这一节，我们将讨论Thread对象，有关Executors将在[高级并发对象](http://ifeve.com/high-level-concurrency-objects/)一节中讨论。

**定义并启动一个线程**

应用程序在创建一个线程实例时，必须提供需要在线程中运行的代码。有两种方式去做到这一点：

* 提供一个Runnable对象。[Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html)对象仅包含一个run()方法，在这个方法中定义的代码将在会线程中执行。将Runnable对象传递给Thread类的构造函数即可，如下面这个[HelloRunnable](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java)的例子：

|  |  |  |
| --- | --- | --- |
| 01 | public class HelloRunnable implements Runnable { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public void run() { |
| 04 | System.out.println("Hello from a thread!"); | |

|  |  |  |
| --- | --- | --- |
| 05 | } | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | public static void main(String args[]) { |
| 08 | (new Thread(new HelloRunnable())).start(); | |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | } |

* 继承Thread类。Thread类自身已实现了Runnable接口，但它的run()方法中并没有定义任何代码。应用程序可以继承与Thread类，并复写run()方法。如例子[HelloThread](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java)

|  |  |  |
| --- | --- | --- |
| 01 | public class HelloThread extends Thread { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public void run() { |
| 04 | System.out.println("Hello from a thread!"); | |

|  |  |  |
| --- | --- | --- |
| 05 | } | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | public static void main(String args[]) { | |
| 08 | (new HelloThread()).start(); |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | } |

需要注意的是，上述两个例子都需要调用Thread.start()方法来启动一个新的线程。 哪一种方式是我们应该使用的？相对来说，第一种更加通用，因为Runnable对象可以继承于其他类（Java只支持单继承，当一个类继承与Thread类后，就无法继承与其他类）。第二种方法更易于在简单的应用程序中使用，但它的局限就是：你的任务类必须是Thread的子类。这个课程更加聚焦于第一种将Runnable任务和Thread类分离的方式。不仅仅是因为这种方式更加灵活，更因为它更适合后面将要介绍的高级线程管理API。 Thread类定义了一些对线程管理十分有用的的方法。在这些方法中，有一些静态方法可以给当前线程调用，它们可以提供一些有关线程的信息，或者影响线程的状态。而其他一些方法可以由其他线程进行调用，用于管理线程和Thread对象。我们将在下面的章节中，深入探讨这些内容。

**使用Sleep方法暂停一个线程**

使用Thread.sleep()方法可以暂停当前线程一段时间。这是一种使处理器时间可以被其他线程或者运用程序使用的有效方式。sleep()方法还可以用于调整线程执行节奏（见下面的例子）和等待其他有执行时间需求的线程（这个例子将在下一节演示）。

在Thread中有两个不同的sleep()方法，一个使用毫秒表示休眠的时间，而另一个是用纳秒。由于操作系统的限制休眠时间并不能保证十分精确。休眠周期可以被interrups所终止，我们将在后面看到这样的例子。不管在任何情况下，我们都不应该假定调用了sleep()方法就可以将一个线程暂停一个十分精确的时间周期。

[SleepMessages](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SleepMessages.java)程序为我们展示了使用sleep()方法每四秒打印一个信息的例子

|  |  |
| --- | --- |
| 01 | public class SleepMessages { |
| 02 | public static void main(String args[]) | |

|  |  |  |
| --- | --- | --- |
| 03 | throws InterruptedException { | |
| 04 | String importantInfo[] = { |

|  |  |  |
| --- | --- | --- |
| 05 | "Mares eat oats", | |
| 06 | "Does eat oats", |

|  |  |
| --- | --- |
| 07 | "Little lambs eat ivy", |
| 08 | "A kid will eat ivy too" | |

|  |  |  |
| --- | --- | --- |
| 09 | }; | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | for (int i = 0; |
| 12 | i < importantInfo.length; | |

|  |  |
| --- | --- |
| 13 | i++) { |
| 14 | //Pause for 4 seconds | |

|  |  |  |
| --- | --- | --- |
| 15 | Thread.sleep(4000); | |
| 16 | //Print a message |

|  |  |  |
| --- | --- | --- |
| 17 | System.out.println(importantInfo[i]); | |
| 18 | } |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

main()方法声明了它有可能抛出InterruptedException。当其他线程中断当前线程时，sleep()方法就会抛出该异常。由于这个应用程序并没有定义其他的线程，所以并不用关心如何处理该异常。

**中断（Interrupts）**

[原文链接](http://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)，[译文链接](http://ifeve.com/java-interrupt)，**译者**： 蘑菇街-小宝   **校对**：梁海舰  
中断是给线程的一个指示，告诉它应该停止正在做的事并去做其他事情。一个线程究竟要怎么响应中断请求取决于程序员，不过让其终止是很普遍的做法。这是本文重点强调的用法。

一个线程通过调用对被中断线程的Thread对象的interrupt()方法，发送中断信号。为了让中断机制正常工作，被中断的线程必须支持它自己的中断(即要自己处理中断)

**中断支持**

线程如何支持自身的中断？这取决于它当前正在做什么。如果线程正在频繁调用会抛InterruptedException异常的方法，在捕获异常之后，它只是从run()方法中返回。例如，假设在SleepMessages的例子中，关键的消息循环在线程的Runnable对象的run方法中，代码可能会被修改成下面这样以支持中断：

|  |  |  |
| --- | --- | --- |
| 01 | for (int i = 0; i < importantInfo.length; i++) { | |
| 02 | // Pause for 4 seconds |

|  |  |
| --- | --- |
| 03 | try { |
| 04 | Thread.sleep(4000); | |

|  |  |
| --- | --- |
| 05 | } catch (InterruptedException e) { |
| 06 | // We've been interrupted: no more messages. | |

|  |  |  |
| --- | --- | --- |
| 07 | return; | |
| 08 | } |

|  |  |
| --- | --- |
| 09 | // Print a message |
| 10 | System.out.println(importantInfo[i]); | |

|  |  |
| --- | --- |
| 11 | } |

许多会抛InterruptedException异常的方法(如sleep()），被设计成接收到中断后取消它们当前的操作，并在立即返回。

如果一个线程长时间运行而不调用会抛InterruptedException异常的方法会怎样？ 那它必须周期性地调用Thread.interrupted()方法，该方法在接收到中断请求后返回true。例如：

|  |  |  |
| --- | --- | --- |
| 1 | for (int i = 0; i < inputs.length; i++) { | |
| 2 | heavyCrunch(inputs[i]); |

|  |  |
| --- | --- |
| 3 | if (Thread.interrupted()) { |
| 4 | // We've been interrupted: no more crunching. | |

|  |  |  |
| --- | --- | --- |
| 5 | return; | |
| 6 | } |

|  |  |
| --- | --- |
| 7 | } |

在这个简单的例子中，代码只是检测中断，并在收到中断后退出线程。在更复杂的应用中，抛出一个InterruptedException异常可能更有意义。

|  |  |
| --- | --- |
| 1 | if (Thread.interrupted()){ |
| 2 | throw new InterruptedException(); | |

|  |  |
| --- | --- |
| 3 | } |

这使得中断处理代码能集中在catch语句中。

**中断状态标记**

中断机制通过使用称为中断状态的内部标记来实现。调用Thread.interrupt()设置这个标记。当线程通过调用静态方法Thread.interrupted()检测中断时，中断状态会被清除。非静态的isInterrupted()方法被线程用来检测其他线程的中断状态，不改变中断状态标记。

按照惯例，任何通过抛出一个InterruptedException异常退出的方法，当抛该异常时会清除中断状态。不过，通过其他的线程调用interrupt()方法，中断状态总是有可能会立即被重新设置。

**Joins**

Join()方法可以让一个线程等待另一个线程执行完成。若t是一个正在执行的Thread对象，

|  |  |
| --- | --- |
| 1 | t.join(); |

将会使当前线程暂停执行并等待t执行完成。重载的join()方法可以让开发者自定义等待周期。然而，和sleep()方法一样join()方法依赖于操作系统的时间处理机制，你不能假定join()方法将会精确的等待你所定义的时长。

如同sleep()方法，join()方法响应中断并在中断时抛出InterruptedException。

**一个简单的线程例子**

下面这个简单的例子将会把这一节的一些概念放到一起演示。[SimpleThreads](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SimpleThreads.java)程序有两个线程组成，第一个是主线程，它从创建了一个线程并等待它执行完成。如果MessageLoop线程执行了太长时间，主线程将会将其中断。

MessageLoop现场将会打印一系列的信息。如果中断在它打印完所有信息前发生，它将会打印一个特定的消息并退出。

[view source](http://ifeve.com/oracel-thread-objects/#viewSource)

[print](http://ifeve.com/oracel-thread-objects/#printSource)[?](http://ifeve.com/oracel-thread-objects/#about)

|  |  |  |
| --- | --- | --- |
| 01 | public class SimpleThreads { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | // Display a message, preceded by |
| 04 | // the name of the current thread |

|  |  |  |
| --- | --- | --- |
| 05 | static void threadMessage(String message) { | |
| 06 | String threadName = |

|  |  |  |
| --- | --- | --- |
| 07 | Thread.currentThread().getName(); | |
| 08 | System.out.format("%s: %s%n", |

|  |  |  |
| --- | --- | --- |
| 09 | threadName, | |
| 10 | message); |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | private static class MessageLoop | |
| 14 | implements Runnable { |

|  |  |
| --- | --- |
| 15 | public void run() { |
| 16 | String importantInfo[] = { | |

|  |  |  |
| --- | --- | --- |
| 17 | "Mares eat oats", | |
| 18 | "Does eat oats", |

|  |  |
| --- | --- |
| 19 | "Little lambs eat ivy", |
| 20 | "A kid will eat ivy too" | |

|  |  |
| --- | --- |
| 21 | }; |
| 22 | try { | |

|  |  |
| --- | --- |
| 23 | for (int i = 0; |
| 24 | i < importantInfo.length; | |

|  |  |
| --- | --- |
| 25 | i++) { |
| 26 | // Pause for 4 seconds | |

|  |  |  |
| --- | --- | --- |
| 27 | Thread.sleep(4000); | |
| 28 | // Print a message |

|  |  |  |  |
| --- | --- | --- | --- |
| 29 | threadMessage(importantInfo[i]); | | |
| 30 | | } |

|  |  |
| --- | --- |
| 31 | } catch (InterruptedException e) { |
| 32 | threadMessage("I wasn't done!"); | |

|  |  |  |
| --- | --- | --- |
| 33 | } | |
| 34 | } |

|  |  |  |
| --- | --- | --- |
| 35 | } | |
| 36 |  |

|  |  |  |
| --- | --- | --- |
| 37 | public static void main(String args[]) | |
| 38 | throws InterruptedException { |

|  |  |
| --- | --- |
| 39 |  |
| 40 | // Delay, in milliseconds before | |

|  |  |
| --- | --- |
| 41 | // we interrupt MessageLoop |
| 42 | // thread (default one hour). | |

|  |  |  |
| --- | --- | --- |
| 43 | long patience = 1000 \* 60 \* 60; | |
| 44 |  |

|  |  |  |
| --- | --- | --- |
| 45 | // If command line argument | |
| 46 | // present, gives patience |

|  |  |
| --- | --- |
| 47 | // in seconds. |
| 48 | if (args.length > 0) { | |

|  |  |
| --- | --- |
| 49 | try { |
| 50 | patience = Long.parseLong(args[0]) \* 1000; | |

|  |  |
| --- | --- |
| 51 | } catch (NumberFormatException e) { |
| 52 | System.err.println("Argument must be an integer."); | |

|  |  |  |
| --- | --- | --- |
| 53 | System.exit(1); | |
| 54 | } |

|  |  |  |
| --- | --- | --- |
| 55 | } | |
| 56 |  |

|  |  |
| --- | --- |
| 57 | threadMessage("Starting MessageLoop thread"); |
| 58 | long startTime = System.currentTimeMillis(); |

|  |  |  |
| --- | --- | --- |
| 59 | Thread t = new Thread(new MessageLoop()); | |
| 60 | t.start(); |

|  |  |
| --- | --- |
| 61 |  |
| 62 | threadMessage("Waiting for MessageLoop thread to finish"); | |

|  |  |  |
| --- | --- | --- |
| 63 | // loop until MessageLoop | |
| 64 | // thread exits |

|  |  |
| --- | --- |
| 65 | while (t.isAlive()) { |
| 66 | threadMessage("Still waiting..."); | |

|  |  |  |
| --- | --- | --- |
| 67 | // Wait maximum of 1 second | |
| 68 | // for MessageLoop thread |

|  |  |
| --- | --- |
| 69 | // to finish. |
| 70 | t.join(1000); |

|  |  |  |
| --- | --- | --- |
| 71 | if (((System.currentTimeMillis() - startTime) > patience) | |
| 72 | && t.isAlive()) { |

|  |  |  |
| --- | --- | --- |
| 73 | threadMessage("Tired of waiting!"); | |
| 74 | t.interrupt(); |

|  |  |  |
| --- | --- | --- |
| 75 | // Shouldn't be long now | |
| 76 | // -- wait indefinitely |

|  |  |  |
| --- | --- | --- |
| 77 | t.join(); | |
| 78 | } |

|  |  |
| --- | --- |
| 79 | } |
| 80 | threadMessage("Finally!"); | |

|  |  |  |
| --- | --- | --- |
| 81 | } | |
| 82 | } |