# 处理 InterruptedException

**简介：** 很多 Java™ 语言方法，例如 Thread.sleep() 和 Object.wait()，都可以抛出InterruptedException。您不能忽略这个异常，因为它是一个检查异常（checked exception）。但是应该如何处理它呢？在本月的 [Java 理论与实践中](http://www.ibm.com/developerworks/cn/views/java/articles.jsp?view_by=search&search_by=Java+%E7%90%86%E8%AE%BA%E4%B8%8E%E5%AE%9E%E8%B7%B5)，并发专家 Brian Goetz 将解释 InterruptedException 的含义，为什么会抛出 InterruptedException，以及在捕捉到该异常时应该怎么做。

这样的情景您也许并不陌生：您在编写一个测试程序，程序需要暂停一段时间，于是调用 Thread.sleep()。但是编译器或 IDE 报错说没有处理检查到的 InterruptedException。InterruptedException 是什么呢，为什么必须处理它？

对于 InterruptedException，一种常见的处理方式是 “生吞（swallow）” 它 —— 捕捉它，然后什么也不做（或者记录下它，不过这也好不到哪去）—— 就像后面的 [清单 4](http://www.ibm.com/developerworks/cn/java/j-jtp05236.html#code4) 一样。不幸的是，这种方法忽略了这样一个事实：这期间可能发生中断，而中断可能导致应用程序丧失及时取消活动或关闭的能力。

**阻塞方法**

当一个方法抛出 InterruptedException 时，它不仅告诉您它可以抛出一个特定的检查异常，而且还告诉您其他一些事情。例如，它告诉您它是一个*阻塞（blocking）*方法，如果您响应得当的话，它将尝试消除阻塞并尽早返回。

阻塞方法不同于一般的要运行较长时间的方法。一般方法的完成只取决于它所要做的事情，以及是否有足够多可用的计算资源（CPU 周期和内存）。而阻塞方法的完成还取决于一些外部的事件，例如计时器到期，I/O 完成，或者另一个线程的动作（释放一个锁，设置一个标志，或者将一个任务放在一个工作队列中）。一般方法在它们的工作做完后即可结束，而阻塞方法较难于预测，因为它们取决于外部事件。阻塞方法可能影响响应能力，因为难于预测它们何时会结束。

阻塞方法可能因为等不到所等的事件而无法终止，因此令阻塞方法*可取消* 就非常有用（如果长时间运行的非阻塞方法是可取消的，那么通常也非常有用）。可取消操作是指能从外部使之在正常完成之前终止的操作。由 Thread 提供并受 Thread.sleep() 和Object.wait() 支持的中断机制就是一种取消机制；它允许一个线程请求另一个线程停止它正在做的事情。当一个方法抛出InterruptedException 时，它是在告诉您，如果执行该方法的线程被中断，它将尝试停止它正在做的事情而提前返回，并通过抛出InterruptedException 表明它提前返回。 行为良好的阻塞库方法应该能对中断作出响应并抛出 InterruptedException，以便能够用于可取消活动中，而不至于影响响应。

**线程中断**

每个线程都有一个与之相关联的 Boolean 属性，用于表示线程的*中断状态（interrupted status）*。中断状态初始时为 false；当另一个线程通过调用 Thread.interrupt() 中断一个线程时，会出现以下两种情况之一。如果那个线程在执行一个低级可中断阻塞方法，例如 Thread.sleep()、 Thread.join() 或 Object.wait()，那么它将取消阻塞并抛出 InterruptedException。否则， interrupt() 只是设置线程的中断状态。 在被中断线程中运行的代码以后可以轮询中断状态，看看它是否被请求停止正在做的事情。中断状态可以通过 Thread.isInterrupted() 来读取，并且可以通过一个名为 Thread.interrupted() 的操作读取和清除。

中断是一种协作机制。当一个线程中断另一个线程时，被中断的线程不一定要立即停止正在做的事情。相反，中断是礼貌地请求另一个线程在它愿意并且方便的时候停止它正在做的事情。有些方法，例如 Thread.sleep()，很认真地对待这样的请求，但每个方法不是一定要对中断作出响应。对于中断请求，不阻塞但是仍然要花较长时间执行的方法可以轮询中断状态，并在被中断的时候提前返回。 您可以随意忽略中断请求，但是这样做的话会影响响应。

中断的协作特性所带来的一个好处是，它为安全地构造可取消活动提供更大的灵活性。我们很少希望一个活动立即停止；如果活动在正在进行更新的时候被取消，那么程序数据结构可能处于不一致状态。中断允许一个可取消活动来清理正在进行的工作，恢复不变量，通知其他活动它要被取消，然后才终止。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp05236.html#ibm-pcon)

**处理 InterruptedException**

如果抛出 InterruptedException 意味着一个方法是阻塞方法，那么调用一个阻塞方法则意味着您的方法也是一个阻塞方法，而且您应该有某种策略来处理 InterruptedException。通常最容易的策略是自己抛出 InterruptedException，如清单 1 中 putTask() 和getTask() 方法中的代码所示。 这样做可以使方法对中断作出响应，并且只需将 InterruptedException 添加到 throws 子句。

**清单 1. 不捕捉 InterruptedException，将它传播给调用者**

|  |
| --- |
| public class TaskQueue {  private static final int MAX\_TASKS = 1000;  private BlockingQueue<Task> queue  = new LinkedBlockingQueue<Task>(MAX\_TASKS);  public void putTask(Task r) **throws InterruptedException** {  queue.put(r);  }  public Task getTask() **throws InterruptedException** {  return queue.take();  }  } |

有时候需要在传播异常之前进行一些清理工作。在这种情况下，可以捕捉 InterruptedException，执行清理，然后抛出异常。清单 2 演示了这种技术，该代码是用于匹配在线游戏服务中的玩家的一种机制。 matchPlayers() 方法等待两个玩家到来，然后开始一个新游戏。如果在一个玩家已到来，但是另一个玩家仍未到来之际该方法被中断，那么它会将那个玩家放回队列中，然后重新抛出InterruptedException，这样那个玩家对游戏的请求就不至于丢失。

**清单 2. 在重新抛出 InterruptedException 之前执行特定于任务的清理工作**

|  |
| --- |
| public class PlayerMatcher {  private PlayerSource players;  public PlayerMatcher(PlayerSource players) {  this.players = players;  }  public void matchPlayers() **throws InterruptedException** {  try {  Player playerOne, playerTwo;  while (true) {  playerOne = playerTwo = null;  // Wait for two players to arrive and start a new game  playerOne = players.waitForPlayer(); // could throw IE  playerTwo = players.waitForPlayer(); // could throw IE  startNewGame(playerOne, playerTwo);  }  }  catch (InterruptedException e) {  // If we got one player and were interrupted, put that player back  if (playerOne != null)  players.addFirst(playerOne);  // Then propagate the exception  **throw e;**  }  }  } |

**不要生吞中断**

有时候抛出 InterruptedException 并不合适，例如当由 Runnable 定义的任务调用一个可中断的方法时，就是如此。在这种情况下，不能重新抛出 InterruptedException，但是您也不想什么都不做。当一个阻塞方法检测到中断并抛出 InterruptedException 时，它清除中断状态。如果捕捉到 InterruptedException 但是不能重新抛出它，那么应该保留中断发生的证据，以便调用栈中更高层的代码能知道中断，并对中断作出响应。该任务可以通过调用 interrupt() 以 “重新中断” 当前线程来完成，如清单 3 所示。至少，每当捕捉到 InterruptedException 并且不重新抛出它时，就在返回之前重新中断当前线程。

**清单 3. 捕捉 InterruptedException 后恢复中断状态**

|  |
| --- |
| public class TaskRunner implements Runnable {  private BlockingQueue<Task> queue;  public TaskRunner(BlockingQueue<Task> queue) {  this.queue = queue;  }  public void run() {  try {  while (true) {  Task task = queue.take(10, TimeUnit.SECONDS);  task.execute();  }  }  catch (InterruptedException e) {  // Restore the interrupted status  **Thread.currentThread().interrupt();**  }  }  } |

处理 InterruptedException 时采取的最糟糕的做法是生吞它 —— 捕捉它，然后既不重新抛出它，也不重新断言线程的中断状态。对于不知如何处理的异常，最标准的处理方法是捕捉它，然后记录下它，但是这种方法仍然无异于生吞中断，因为调用栈中更高层的代码还是无法获得关于该异常的信息。（仅仅记录 InterruptedException 也不是明智的做法，因为等到人来读取日志的时候，再来对它作出处理就为时已晚了。） 清单 4 展示了一种使用得很广泛的模式，这也是生吞中断的一种模式：

**清单 4. 生吞中断 —— 不要这么做**

|  |
| --- |
| // Don't do this  public class TaskRunner implements Runnable {  private BlockingQueue<Task> queue;  public TaskRunner(BlockingQueue<Task> queue) {  this.queue = queue;  }  public void run() {  try {  while (true) {  Task task = queue.take(10, TimeUnit.SECONDS);  task.execute();  }  }  catch (InterruptedException swallowed) {  /\* DON'T DO THIS - RESTORE THE INTERRUPTED STATUS INSTEAD \*/  }  }  } |

如果不能重新抛出 InterruptedException，不管您是否计划处理中断请求，仍然需要重新中断当前线程，因为一个中断请求可能有多个 “接收者”。标准线程池 （ThreadPoolExecutor）worker 线程实现负责中断，因此中断一个运行在线程池中的任务可以起到双重效果，一是取消任务，二是通知执行线程线程池正要关闭。如果任务生吞中断请求，则 worker 线程将不知道有一个被请求的中断，从而耽误应用程序或服务的关闭。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp05236.html#ibm-pcon)

**实现可取消任务**

语言规范中并没有为中断提供特定的语义，但是在较大的程序中，难于维护除取消外的任何中断语义。取决于是什么活动，用户可以通过一个 GUI 或通过网络机制，例如 JMX 或 Web 服务来请求取消。程序逻辑也可以请求取消。例如，一个 Web 爬行器（crawler）如果检测到磁盘已满，它会自动关闭自己，否则一个并行算法会启动多个线程来搜索解决方案空间的不同区域，一旦其中一个线程找到一个解决方案，就取消那些线程。

仅仅因为一个任务是可取消的，并不意味着需要*立即* 对中断请求作出响应。对于执行一个循环中的代码的任务，通常只需为每一个循环迭代检查一次中断。取决于循环执行的时间有多长，任何代码可能要花一些时间才能注意到线程已经被中断（或者是通过调用Thread.isInterrupted() 方法轮询中断状态，或者是调用一个阻塞方法）。 如果任务需要提高响应能力，那么它可以更频繁地轮询中断状态。阻塞方法通常在入口就立即轮询中断状态，并且，如果它被设置来改善响应能力，那么还会抛出 InterruptedException。

惟一可以生吞中断的时候是您知道线程正要退出。只有当调用可中断方法的类是 Thread 的一部分，而不是 Runnable 或通用库代码的情况下，才会发生这样的场景，清单 5 演示了这种情况。清单 5 创建一个线程，该线程列举素数，直到被中断，这里还允许该线程在被中断时退出。用于搜索素数的循环在两个地方检查是否有中断：一处是在 while 循环的头部轮询 isInterrupted() 方法，另一处是调用阻塞方法 BlockingQueue.put()。

**清单 5. 如果知道线程正要退出的话，则可以生吞中断**

|  |
| --- |
| public class PrimeProducer extends Thread {  private final BlockingQueue<BigInteger> queue;  PrimeProducer(BlockingQueue<BigInteger> queue) {  this.queue = queue;  }  public void run() {  try {  BigInteger p = BigInteger.ONE;  while (!Thread.currentThread().isInterrupted())  queue.put(p = p.nextProbablePrime());  } catch (InterruptedException consumed) {  /\* Allow thread to exit \*/  }  }  public void cancel() { interrupt(); }  } |

**不可中断的阻塞方法**

并非所有的阻塞方法都抛出 InterruptedException。输入和输出流类会阻塞等待 I/O 完成，但是它们不抛出 InterruptedException，而且在被中断的情况下也不会提前返回。然而，对于套接字 I/O，如果一个线程关闭套接字，则那个套接字上的阻塞 I/O 操作将提前结束，并抛出一个 SocketException。java.nio 中的非阻塞 I/O 类也不支持可中断 I/O，但是同样可以通过关闭通道或者请求 Selector上的唤醒来取消阻塞操作。类似地，尝试获取一个内部锁的操作（进入一个 synchronized 块）是不能被中断的，但是 ReentrantLock支持可中断的获取模式。

**不可取消的任务**

有些任务拒绝被中断，这使得它们是不可取消的。但是，即使是不可取消的任务也应该尝试保留中断状态，以防在不可取消的任务结束之后，调用栈上更高层的代码需要对中断进行处理。清单 6 展示了一个方法，该方法等待一个阻塞队列，直到队列中出现一个可用项目，而不管它是否被中断。为了方便他人，它在结束后在一个 finally 块中恢复中断状态，以免剥夺中断请求的调用者的权利。（它不能在更早的时候恢复中断状态，因为那将导致无限循环 —— BlockingQueue.take() 将在入口处立即轮询中断状态，并且，如果发现中断状态集，就会抛出 InterruptedException。）

**清单 6. 在返回前恢复中断状态的不可取消任务**

|  |
| --- |
| public Task getNextTask(BlockingQueue<Task> queue) {  boolean interrupted = false;  try {  while (true) {  try {  return queue.take();  } catch (InterruptedException e) {  interrupted = true;  // fall through and retry  }  }  } finally {  if (interrupted)  Thread.currentThread().interrupt();  }  } |

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp05236.html#ibm-pcon)

**结束语**

您可以用 Java 平台提供的协作中断机制来构造灵活的取消策略。各活动可以自行决定它们是可取消的还是不可取消的，以及如何对中断作出响应，如果立即返回会危害应用程序完整性的话，它们还可以推迟中断。即使您想在代码中完全忽略中断，也应该确保在捕捉到 InterruptedException 但是没有重新抛出它的情况下，恢复中断状态，以免调用它的代码无法获知中断的发生。