一、中断

<http://ifeve.com/java-interrupt-mechanism/>

<http://blog.csdn.net/canot/article/details/51087772>

**1. 引言**

当我们点击某个杀毒软件的取消按钮来停止查杀病毒时，当我们在控制台敲入quit命令以结束某个后台服务时……都需要通过一个线程去取消另一个线程正在执行的任务。Java没有提供一种安全直接的方法来停止某个线程，但是Java提供了中断机制。

如果对Java中断没有一个全面的了解，可能会误以为被中断的线程将立马退出运行，但事实并非如此。中断机制是如何工作的？捕获或检测到中断后，是抛出InterruptedException还是重设中断状态以及在方法中吞掉中断状态会有什么后果？Thread.stop与中断相比又有哪些异同？什么情况下需要使用中断？本文将从以上几个方面进行描述。

**2. 中断的原理**

Java中断机制是一种协作机制，也就是说通过中断并不能直接终止另一个线程，而需要被中断的线程自己处理中断。这好比是家里的父母叮嘱在外的子女要注意身体，但子女是否注意身体，怎么注意身体则完全取决于自己。

Java中断模型也是这么简单，每个线程对象里都有一个boolean类型的标识（不一定就要是Thread类的字段，实际上也的确不是，这几个方法最终都是通过native方法来完成的），代表着是否有中断请求（该请求可以来自所有线程，包括被中断的线程本身）。例如，当线程t1想中断线程t2，只需要在线程t1中将线程t2对象的中断标识置为true，然后线程2可以选择在合适的时候处理该中断请求，甚至可以不理会该请求，就像这个线程没有被中断一样。

java.lang.Thread类提供了几个方法来操作这个中断状态，这些方法包括：

|  |  |
| --- | --- |
| public static boolean interrupted | 测试当前线程是否已经中断。线程的中断状态 由该方法清除。换句话说，如果连续两次调用该方法，则第二次调用将返回 false（在第一次调用已清除了其中断状态之后，且第二次调用检验完中断状态前，当前线程再次中断的情况除外）。 |
| public boolean isInterrupted() | 测试线程是否已经中断。线程的中断状态不受该方法的影响。 |
| public void interrupt() | 中断线程。 |

其中，interrupt方法是唯一能将中断状态设置为true的方法。静态方法interrupted会将当前线程的中断状态清除，但这个方法的命名极不直观，很容易造成误解，需要特别注意。

上面的例子中，线程t1通过调用interrupt方法将线程t2的中断状态置为true，t2可以在合适的时候调用interrupted或isInterrupted来检测状态并做相应的处理。

此外，类库中的有些类的方法也可能会调用中断，如FutureTask中的cancel方法，如果传入的参数为true，它将会在正在运行异步任务的线程上调用interrupt方法，如果正在执行的异步任务中的代码没有对中断做出响应，那么cancel方法中的参数将不会起到什么效果；又如ThreadPoolExecutor中的shutdownNow方法会遍历线程池中的工作线程并调用线程的interrupt方法来中断线程，所以如果工作线程中正在执行的任务没有对中断做出响应，任务将一直执行直到正常结束。

**3. 中断的处理**

既然Java中断机制只是设置被中断线程的中断状态，那么被中断线程该做些什么？

**处理时机**

显然，作为一种协作机制，不会强求被中断线程一定要在某个点进行处理。实际上，被中断线程只需在合适的时候处理即可，如果没有合适的时间点，甚至可以不处理，这时候在任务处理层面，就跟没有调用中断方法一样。“合适的时候”与线程正在处理的业务逻辑紧密相关，例如，每次迭代的时候，进入一个可能阻塞且无法中断的方法之前等，但多半不会出现在某个临界区更新另一个对象状态的时候，因为这可能会导致对象处于不一致状态。

处理时机决定着程序的效率与中断响应的灵敏性。频繁的检查中断状态可能会使程序执行效率下降，相反，检查的较少可能使中断请求得不到及时响应。如果发出中断请求之后，被中断的线程继续执行一段时间不会给系统带来灾难，那么就可以将中断处理放到方便检查中断，同时又能从一定程度上保证响应灵敏度的地方。当程序的性能指标比较关键时，可能需要建立一个测试模型来分析最佳的中断检测点，以平衡性能和响应灵敏性。

**处理方式**

1、 中断状态的管理

一般说来，当可能阻塞的方法声明中有抛出InterruptedException则暗示该方法是可中断的，如BlockingQueue#put、BlockingQueue#take、Object#wait、Thread#sleep等，如果程序捕获到这些可中断的阻塞方法抛出的InterruptedException或检测到中断后，这些中断信息该如何处理？一般有以下两个通用原则：

如果遇到的是可中断的阻塞方法抛出InterruptedException，可以继续向方法调用栈的上层抛出该异常，如果是检测到中断，则可清除中断状态并抛出InterruptedException，使当前方法也成为一个可中断的方法。

若有时候不太方便在方法上抛出InterruptedException，比如要实现的某个接口中的方法签名上没有throws InterruptedException，这时就可以捕获可中断方法的InterruptedException并通过Thread.currentThread.interrupt()来重新设置中断状态。如果是检测并清除了中断状态，亦是如此。

一般的代码中，尤其是作为一个基础类库时，绝不应当吞掉中断，即捕获到InterruptedException后在catch里什么也不做，清除中断状态后又不重设中断状态也不抛出InterruptedException等。因为吞掉中断状态会导致方法调用栈的上层得不到这些信息。

当然，凡事总有例外的时候，当你完全清楚自己的方法会被谁调用，而调用者也不会因为中断被吞掉了而遇到麻烦，就可以这么做。

总得来说，就是要让方法调用栈的上层获知中断的发生。假设你写了一个类库，类库里有个方法amethod，在amethod中检测并清除了中断状态，而没有抛出InterruptedException，作为amethod的用户来说，他并不知道里面的细节，如果用户在调用amethod后也要使用中断来做些事情，那么在调用amethod之后他将永远也检测不到中断了，因为中断信息已经被amethod清除掉了。如果作为用户，遇到这样有问题的类库，又不能修改代码，那该怎么处理？只好在自己的类里设置一个自己的中断状态，在调用interrupt方法的时候，同时设置该状态，这实在是无路可走时才使用的方法。

2、 中断的响应

程序里发现中断后该怎么响应？这就得视实际情况而定了。有些程序可能一检测到中断就立马将线程终止，有些可能是退出当前执行的任务，继续执行下一个任务……作为一种协作机制，这要与中断方协商好，当调用interrupt会发生些什么都是事先知道的，如做一些事务回滚操作，一些清理工作，一些补偿操作等。若不确定调用某个线程的interrupt后该线程会做出什么样的响应，那就不应当中断该线程。

**4. Thread.interrupt VS Thread.stop**

Thread.stop方法已经不推荐使用了。而在某些方面Thread.stop与中断机制有着相似之处。如当线程在等待内置锁或IO时，stop跟interrupt一样，不会中止这些操作；当catch住stop导致的异常时，程序也可以继续执行，虽然stop本意是要停止线程，这么做会让程序行为变得更加混乱。

那么它们的区别在哪里？最重要的就是中断需要程序自己去检测然后做相应的处理，而Thread.stop会直接在代码执行过程中抛出ThreadDeath错误，这是一个java.lang.Error的子类。

在继续之前，先来看个小例子：

|  |  |
| --- | --- |
| 01 | package com.ticmy.interrupt; |

|  |  |
| --- | --- |
| 02 | import java.util.Arrays; |

|  |  |
| --- | --- |
| 03 | import java.util.Random; |

|  |  |
| --- | --- |
| 04 | import java.util.concurrent.TimeUnit; |

|  |  |
| --- | --- |
| 05 | public class TestStop { |

|  |  |
| --- | --- |
| 06 | private static final int[] array = new int[80000]; |

|  |  |
| --- | --- |
| 07 | private static final Thread t = new Thread() { |

|  |  |
| --- | --- |
| 08 | public void run() { |

|  |  |
| --- | --- |
| 09 | try { |

|  |  |
| --- | --- |
| 10 | System.out.println(sort(array)); |

|  |  |
| --- | --- |
| 11 | } catch (Error err) { |

|  |  |
| --- | --- |
| 12 | err.printStackTrace(); |

|  |  |
| --- | --- |
| 13 | } |

|  |  |
| --- | --- |
| 14 | System.out.println("in thread t"); |

|  |  |
| --- | --- |
| 15 | } |

|  |  |
| --- | --- |
| 16 | }; |

|  |  |
| --- | --- |
| 17 |  |

|  |  |
| --- | --- |
| 18 | static { |

|  |  |
| --- | --- |
| 19 | Random random = new Random(); |

|  |  |
| --- | --- |
| 20 | for(int i = 0; i < array.length; i++) { |

|  |  |
| --- | --- |
| 21 | array[i] = random.nextInt(i + 1); |

|  |  |
| --- | --- |
| 22 | } |

|  |  |
| --- | --- |
| 23 | } |

|  |  |
| --- | --- |
| 24 |  |

|  |  |
| --- | --- |
| 25 | private static int sort(int[] array) { |

|  |  |
| --- | --- |
| 26 | for (int i = 0; i < array.length-1; i++){ |

|  |  |
| --- | --- |
| 27 | for(int j = 0 ;j < array.length - i - 1; j++){ |

|  |  |
| --- | --- |
| 28 | if(array[j] < array[j + 1]){ |

|  |  |
| --- | --- |
| 29 | int temp = array[j]; |

|  |  |
| --- | --- |
| 30 | array[j] = array[j + 1]; |

|  |  |
| --- | --- |
| 31 | array[j + 1] = temp; |

|  |  |
| --- | --- |
| 32 | } |

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

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

|  |  |
| --- | --- |
| 35 | return array[0]; |

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

|  |  |
| --- | --- |
| 37 |  |

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

|  |  |
| --- | --- |
| 39 | t.start(); |

|  |  |
| --- | --- |
| 40 | TimeUnit.SECONDS.sleep(1); |

|  |  |
| --- | --- |
| 41 | System.out.println("go to stop thread t"); |

|  |  |
| --- | --- |
| 42 | t.stop(); |

|  |  |
| --- | --- |
| 43 | System.out.println("finish main"); |

|  |  |
| --- | --- |
| 44 | } |

|  |  |
| --- | --- |
| 45 | } |

这个例子很简单，线程t里面做了一个非常耗时的排序操作，排序方法中，只有简单的加、减、赋值、比较等操作，一个可能的执行结果如下：

go to stop thread t
java.lang.ThreadDeath
at java.lang.Thread.stop(Thread.java:758)
at com.ticmy.interrupt.TestStop.main(TestStop.java:44)
finish main
in thread t

这里sort方法是个非常耗时的操作，也就是说主线程休眠一秒钟后调用stop的时候，线程t还在执行sort方法。就是这样一个简单的方法，也会抛出错误！换一句话说，调用stop后，大部分Java字节码都有可能抛出错误，哪怕是简单的加法！

如果线程当前正持有锁，stop之后则会释放该锁。由于此错误可能出现在很多地方，那么这就让编程人员防不胜防，极易造成对象状态的不一致。例如，对象obj中存放着一个范围值：最小值low，最大值high，且low不得大于high，这种关系由锁lock保护，以避免并发时产生竞态条件而导致该关系失效。假设当前low值是5，high值是10，当线程t获取lock后，将low值更新为了15，此时被stop了，真是糟糕，如果没有捕获住stop导致的Error，low的值就为15，high还是10，这导致它们之间的小于关系得不到保证，也就是对象状态被破坏了！如果在给low赋值的时候catch住stop导致的Error则可能使后面high变量的赋值继续，但是谁也不知道Error会在哪条语句抛出，如果对象状态之间的关系更复杂呢？这种方式几乎是无法维护的，太复杂了！如果是中断操作，它决计不会在执行low赋值的时候抛出错误，这样程序对于对象状态一致性就是可控的。

正是因为可能导致对象状态不一致，stop才被禁用。

**5. 中断的使用**

通常，中断的使用场景有以下几个：

点击某个桌面应用中的取消按钮时；

某个操作超过了一定的执行时间限制需要中止时；

多个线程做相同的事情，只要一个线程成功其它线程都可以取消时；

一组线程中的一个或多个出现错误导致整组都无法继续时；

当一个应用或服务需要停止时。

下面来看一个具体的例子。这个例子里，本打算采用GUI形式，但考虑到GUI代码会使程序复杂化，就使用控制台来模拟下核心的逻辑。这里新建了一个磁盘文件扫描的任务，扫描某个目录下的所有文件并将文件路径打印到控制台，扫描的过程可能会很长。若需要中止该任务，只需在控制台键入quit并回车即可。

|  |  |
| --- | --- |
| 01 | package com.ticmy.interrupt; |

|  |  |
| --- | --- |
| 02 | import java.io.BufferedReader; |

|  |  |
| --- | --- |
| 03 | import java.io.File; |

|  |  |
| --- | --- |
| 04 | import java.io.InputStreamReader; |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | public class FileScanner { |

|  |  |
| --- | --- |
| 07 | private static void listFile(File f) throws InterruptedException { |

|  |  |
| --- | --- |
| 08 | if(f == null) { |

|  |  |
| --- | --- |
| 09 | throw new IllegalArgumentException(); |

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

|  |  |
| --- | --- |
| 11 | if(f.isFile()) { |

|  |  |
| --- | --- |
| 12 | System.out.println(f); |

|  |  |
| --- | --- |
| 13 | return; |

|  |  |
| --- | --- |
| 14 | } |

|  |  |
| --- | --- |
| 15 | File[] allFiles = f.listFiles(); |

|  |  |
| --- | --- |
| 16 | if(Thread.interrupted()) { |

|  |  |
| --- | --- |
| 17 | throw new InterruptedException("文件扫描任务被中断"); |

|  |  |
| --- | --- |
| 18 | } |

|  |  |
| --- | --- |
| 19 | for(File file : allFiles) { |

|  |  |
| --- | --- |
| 20 | //还可以将中断检测放到这里 |

|  |  |
| --- | --- |
| 21 | listFile(file); |

|  |  |
| --- | --- |
| 22 | } |

|  |  |
| --- | --- |
| 23 | } |

|  |  |
| --- | --- |
| 24 |  |

|  |  |
| --- | --- |
| 25 | public static String readFromConsole() { |

|  |  |
| --- | --- |
| 26 | BufferedReader reader = new BufferedReader(newInputStreamReader(System.in)); |

|  |  |
| --- | --- |
| 27 | try { |

|  |  |
| --- | --- |
| 28 | return reader.readLine(); |

|  |  |
| --- | --- |
| 29 | } catch (Exception e) { |

|  |  |
| --- | --- |
| 30 | e.printStackTrace(); |

|  |  |
| --- | --- |
| 31 | return ""; |

|  |  |
| --- | --- |
| 32 | } |

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

|  |  |
| --- | --- |
| 34 |  |

|  |  |
| --- | --- |
| 35 | public static void main(String[] args) throws Exception { |

|  |  |
| --- | --- |
| 36 | final Thread fileIteratorThread = new Thread() { |

|  |  |
| --- | --- |
| 37 | public void run() { |

|  |  |
| --- | --- |
| 38 | try { |

|  |  |
| --- | --- |
| 39 | listFile(new File("c:\\")); |

|  |  |
| --- | --- |
| 40 | } catch (InterruptedException e) { |

|  |  |
| --- | --- |
| 41 | e.printStackTrace(); |

|  |  |
| --- | --- |
| 42 | } |

|  |  |
| --- | --- |
| 43 | } |

|  |  |
| --- | --- |
| 44 | }; |

|  |  |
| --- | --- |
| 45 | new Thread() { |

|  |  |
| --- | --- |
| 46 | public void run() { |

|  |  |
| --- | --- |
| 47 | while(true) { |

|  |  |
| --- | --- |
| 48 | if("quit".equalsIgnoreCase(readFromConsole())) { |

|  |  |
| --- | --- |
| 49 | if(fileIteratorThread.isAlive()) { |

|  |  |
| --- | --- |
| 50 | fileIteratorThread.interrupt(); |

|  |  |
| --- | --- |
| 51 | return; |

|  |  |
| --- | --- |
| 52 | } |

|  |  |
| --- | --- |
| 53 | } else { |

|  |  |
| --- | --- |
| 54 | System.out.println("输入quit退出文件扫描"); |

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

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

|  |  |
| --- | --- |
| 57 | } |

|  |  |
| --- | --- |
| 58 | }.start(); |

|  |  |
| --- | --- |
| 59 | fileIteratorThread.start(); |

|  |  |
| --- | --- |
| 60 | } |

|  |  |
| --- | --- |
| 61 | } |

在扫描文件的过程中，对于中断的检测这里采用的策略是，如果碰到的是文件就不检测中断，是目录才检测中断，因为文件可能是非常多的，每次遇到文件都检测一次会降低程序执行效率。此外，在fileIteratorThread线程中，仅是捕获了InterruptedException，没有重设中断状态也没有继续抛出异常，因为我非常清楚它的使用环境，run方法的调用栈上层已经没有可能需要检测中断状态的方法了。

在这个程序中，输入quit完全可以执行System.exit(0)操作来退出程序，但正如前面提到的，这是个GUI程序核心逻辑的模拟，在GUI中，执行System.exit(0)会使得整个程序退出。

**6. 参考资料**

《Java Concurrency in Practice》

《Concurrent Programming in Java Design principles and patterns》

<http://docs.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html>

二、执行器，线程池