在上篇文章《多线程的使用——Thread类和Runnable接口》中提到中断线程的问题。在JAVA中，曾经使用stop方法来停止线程，然而，该方法具有固有的不安全性，因而已经被抛弃(Deprecated)。那么应该怎么结束一个进程呢？官方文档中对此有详细说明：[《为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume？》](http://download.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html)。在此引用stop方法的说明：

1. Why is Thread.stop deprecated?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

大概意思是：

因为该方法本质上是不安全的。停止一个线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 ThreadDeath 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。此行为可能是微妙的，难以察觉，也可能是显著的。不像其他的未检查异常，ThreadDeath异常会在后台杀死线程，因此，用户并不会得到警告，提示他的程序可能已损坏。这种损坏有可能在实际破坏发生之后的任何时间表现出来，也有可能在多小时甚至在未来的很多天后。

在文档中还提到，程序员不能通过捕获ThreadDeath异常来修复已破坏的对象。具体原因见原文。

既然stop方法不建议使用，那么应该用什么方法来代理stop已实现相应的功能呢？

**1、通过修改共享变量来通知目标线程停止运行**

大部分需要使用stop的地方应该使用这种方法来达到中断线程的目的。

这种方法有几个要求或注意事项：

（1）目标线程必须有规律的检查变量，当该变量指示它应该停止运行时，该线程应该按一定的顺序从它执行的方法中返回。

（2）该变量必须定义为volatile，或者所有对它的访问必须同步(synchronized)。

例如：

假如你的applet包括start,stop,run几个方法：

1. **private** Thread blinker;
3. **public** **void** start() {
4. blinker = **new** Thread(**this**);
5. blinker.start();
6. }
8. **public** **void** stop() {
9. blinker.stop();  // UNSAFE!
10. }
12. **public** **void** run() {
13. Thread thisThread = Thread.currentThread();
14. **while** (**true**) {
15. **try** {
16. thisThread.sleep(interval);
17. } **catch** (InterruptedException e){
18. }
19. repaint();
20. }
21. }

你可以使用如下方式避免使用Thread.stop方法：

1. **private** **volatile** Thread blinker;
3. **public** **void** stop() {
4. blinker = **null**;
5. }
7. **public** **void** run() {
8. Thread thisThread = Thread.currentThread();
9. **while** (blinker == thisThread) {
10. **try** {
11. thisThread.sleep(interval);
12. } **catch** (InterruptedException e){
13. }
14. repaint();
15. }
16. }

**2、通过Thread.interrupt方法中断线程**

通常情况下，我们应该使用第一种方式来代替Thread.stop方法。然而以下几种方式应该使用Thread.interrupt方法来中断线程（该方法通常也会结合第一种方法使用）。

一开始使用interrupt方法时，会有莫名奇妙的感觉：难道该方法有问题？

API文档上说，该方法用于"Interrupts this thread"。请看下面的例子：

1. **package** com.polaris.thread;
3. **public** **class** TestThread **implements** Runnable{
5. **boolean** stop = **false**;
6. **public** **static** **void** main(String[] args) **throws** Exception {
7. Thread thread = **new** Thread(**new** TestThread(),"My Thread");
8. System.out.println( "Starting thread..." );
9. thread.start();
10. Thread.sleep( 3000 );
11. System.out.println( "Interrupting thread..." );
12. thread.interrupt();
13. System.out.println("线程是否中断：" + thread.isInterrupted());
14. Thread.sleep( 3000 );
15. System.out.println("Stopping application..." );
16. }
17. **public** **void** run() {
18. **while**(!stop){
19. System.out.println( "My Thread is running..." );
20. // 让该循环持续一段时间，使上面的话打印次数少点
21. **long** time = System.currentTimeMillis();
22. **while**((System.currentTimeMillis()-time < 1000)) {
23. }
24. }
25. System.out.println("My Thread exiting under request..." );
26. }
27. }

运行后的结果是：

Starting thread...

My Thread is running...

My Thread is running...

My Thread is running...

My Thread is running...

Interrupting thread...

线程是否中断：true

My Thread is running...

My Thread is running...

My Thread is running...

Stopping application...

My Thread is running...

My Thread is running...

……

应用程序并不会退出，启动的线程没有因为调用interrupt而终止，可是从调用isInterrupted方法返回的结果可以清楚地知道该线程已经中断了。那位什么会出现这种情况呢？到底是interrupt方法出问题了还是isInterrupted方法出问题了？在Thread类中还有一个测试中断状态的方法（静态的）interrupted，换用这个方法测试，得到的结果是一样的。由此似乎应该是interrupt方法出问题了。于是，在网上有一篇文章：《 Java Thread.interrupt 害人！ 中断JAVA线程》，它详细的说明了应该如何使用interrupt来中断一个线程的执行。

实际上，在JAVA API文档中对该方法进行了详细的说明。该方法实际上只是设置了一个中断状态，当该线程由于下列原因而受阻时，这个中断状态就起作用了：

（1）如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法，或者该类的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法过程中受阻，则其中断状态将被清除，它还将收到一个InterruptedException异常。这个时候，我们可以通过捕获InterruptedException异常来终止线程的执行，具体可以通过return等退出或改变共享变量的值使其退出。

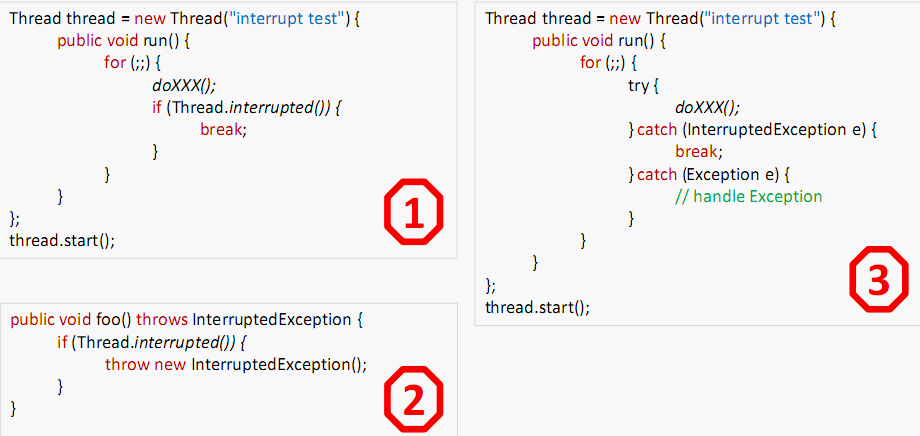
（2）如果该线程在可中断的通道上的 I/O 操作中受阻，则该通道将被关闭，该线程的中断状态将被设置并且该线程将收到一个 ClosedByInterruptException。这时候处理方法一样，只是捕获的异常不一样而已。

其实对于这些情况有一个通用的处理方法：

1. **package** com.polaris.thread;
3. **public** **class** TestThread2 **implements** Runnable{
5. **boolean** stop = **false**;
6. **public** **static** **void** main(String[] args) **throws** Exception {
7. Thread thread = **new** Thread(**new** TestThread2(),"My Thread2");
8. System.out.println( "Starting thread..." );
9. thread.start();
10. Thread.sleep( 3000 );
11. System.out.println( "Interrupting thread..." );
12. thread.interrupt();
13. System.out.println("线程是否中断：" + thread.isInterrupted());
14. Thread.sleep( 3000 );
15. System.out.println("Stopping application..." );
16. }
17. **public** **void** run() {
18. **while**(!stop){
19. System.out.println( "My Thread is running..." );
20. // 让该循环持续一段时间，使上面的话打印次数少点
21. **long** time = System.currentTimeMillis();
22. **while**((System.currentTimeMillis()-time < 1000)) {
23. }
24. **if**(Thread.currentThread().isInterrupted()) {
25. **return**;
26. }
27. }
28. System.out.println("My Thread exiting under request..." );
29. }
30. }

因为调用interrupt方法后，会设置线程的中断状态，所以，通过监视该状态来达到终止线程的目的。

总结：程序应该对线程中断作出恰当的响应。响应方式通常有三种：（来自温绍锦（昵称：温少）：http//www.cnblogs.com/jobs/）

[](http://img1.51cto.com/attachment/201008/160457761.png)

注意：interrupted与isInterrupted方法的区别（见API文档）

引用一篇文章

来自随心所欲<http://redisliu.blog.sohu.com/131647795.html>的《Java的interrupt机制》

当外部线程对某线程调用了thread.interrupt()方法后，java语言的处理机制如下：

如果该线程处在可中断状态下，（调用了xx.wait()，或者Selector.select(),Thread.sleep()等特定会发生阻塞的api），那么该线程会立即被唤醒，同时会受到一个InterruptedException，同时，如果是阻塞在io上，对应的资源会被关闭。如果该线程接下来不执行“Thread.interrupted()方法（不是interrupt），那么该线程处理任何io资源的时候，都会导致这些资源关闭。当然，解决的办法就是调用一下interrupted()，不过这里需要程序员自行根据代码的逻辑来设定，根据自己的需求确认是否可以直接忽略该中断，还是应该马上退出。

如果该线程处在不可中断状态下，就是没有调用上述api，那么java只是设置一下该线程的interrupt状态，其他事情都不会发生，如果该线程之后会调用行数阻塞API，那到时候线程会马会上跳出，并抛出InterruptedException，接下来的事情就跟第一种状况一致了。如果不会调用阻塞API，那么这个线程就会一直执行下去。除非你就是要实现这样的线程，一般高性能的代码中肯定会有wait()，yield()之类出让cpu的函数，不会发生后者的情况。