终止线程的三种方法   
  
    有三种方法可以使终止线程。   
  
    1.  使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。   
  
    2.  使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。   
  
    3.  使用interrupt方法中断线程。   
1. 使用退出标志终止线程   
  
    当run方法执行完后，线程就会退出。但有时run方法是永远不会结束的。如在服务端程序中使用线程进行监听客户端请求，或是其他的需要循环处理的任务。在这种情况下，一般是将这些任务放在一个循环中，如while循环。如果想让循环永远运行下去，可以使用while（true）{……}来处理。但要想使while循环在某一特定条件下退出，最直接的方法就是设一个boolean类型的标志，并通过设置这个标志为true或false来控制while循环是否退出。下面给出了一个利用退出标志终止线程的例子。   
  
package chapter2;   
  
public class ThreadFlag extends Thread   
{   
    public volatile boolean exit = false;   
  
    public void run()   
    {   
        while (!exit);   
    }   
    public static void main(String[] args) throws Exception   
    {   
        ThreadFlag thread = new ThreadFlag();   
        thread.start();   
        sleep(5000); // 主线程延迟5秒   
        thread.exit = true;  // 终止线程thread   
        thread.join();   
        System.out.println("线程退出!");   
    }   
}   
  
  
    在上面代码中定义了一个退出标志exit，当exit为true时，while循环退出，exit的默认值为false.在定义exit时，使用了一个Java关键字volatile，这个关键字的目的是使exit同步，也就是说在同一时刻只能由一个线程来修改exit的值，   
  
    2. 使用stop方法终止线程   
  
    使用stop方法可以强行终止正在运行或挂起的线程。我们可以使用如下的代码来终止线程：   
  
thread.stop();   
  
  
    虽然使用上面的代码可以终止线程，但使用stop方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，因此，并不推荐使用stop方法来终止线程。   
  
    3. 使用interrupt方法终止线程   
  
    使用interrupt方法来终端线程可分为两种情况：   
  
    （1）线程处于阻塞状态，如使用了sleep方法。   
  
    （2）使用while（！isInterrupted（））{……}来判断线程是否被中断。   
  
    在第一种情况下使用interrupt方法，sleep方法将抛出一个InterruptedException例外，而在第二种情况下线程将直接退出。下面的代码演示了在第一种情况下使用interrupt方法。   
  
package chapter2;   
  
public class ThreadInterrupt extends Thread   
{   
    public void run()   
    {   
        try   
        {   
            sleep(50000);  // 延迟50秒   
        }   
        catch (InterruptedException e)   
        {   
            System.out.println(e.getMessage());   
        }   
    }   
    public static void main(String[] args) throws Exception   
    {   
        Thread thread = new ThreadInterrupt();   
        thread.start();   
        System.out.println("在50秒之内按任意键中断线程!");   
        System.in.read();   
        thread.interrupt();   
        thread.join();   
        System.out.println("线程已经退出!");   
    }   
}   
  
  
    上面代码的运行结果如下：   
  
    在50秒之内按任意键中断线程!   
  
    sleep interrupted   
    线程已经退出!   
  
  
    在调用interrupt方法后， sleep方法抛出异常，然后输出错误信息：sleep interrupted.   
  
    注意：在Thread类中有两个方法可以判断线程是否通过interrupt方法被终止。一个是静态的方法interrupted（），一个是非静态的方法isInterrupted（），这两个方法的区别是interrupted用来判断当前线是否被中断，而isInterrupted可以用来判断其他线程是否被中断。因此，while （！isInterrupted（））也可以换成while （！Thread.interrupted（））。

## [如何停止JAVA线程](http://www.cnblogs.com/gpcuster/archive/2010/01/18/1650273.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

如何停止java的线程一直是一个困恼我们开发多线程程序的一个问题。这个问题最终在Java5的[java.util.concurrent](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)中得到了回答：使用[interrupt()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "interrupt()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)，让线程在run方法中停止。

#### 简介

在Java的多线程编程中，[java.lang.Thread](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)类型包含了一些列的方法[start()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "start()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank), [stop()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "stop()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank), [stop(Throwable)](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "stop(java.lang.Throwable)" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank) and [suspend()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "suspend()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank), [destroy()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "destroy()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank) and [resume()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "resume()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)。通过这些方法，我们可以对线程进行方便的操作，但是这些方法中，只有[start()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "start()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)方法得到了保留。

在Sun公司的一篇文章《[Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank) 》中详细讲解了舍弃这些方法的原因。那么，我们究竟应该如何停止线程呢？

#### 建议使用的方法

在《[Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank) 》中，建议使用如下的方法来停止线程：

    private volatile Thread blinker;   
    public void stop() {   
        blinker = null;   
    }   
    public void run() {   
        Thread thisThread = Thread.currentThread();   
        while (blinker == thisThread) {   
            try {   
                thisThread.sleep(interval);   
            } catch (InterruptedException e){   
            }   
            repaint();   
        }   
    }

关于使用volatile关键字的原因，请查看[http://java.sun.com/docs/books/jls/second\_edition/html/classes.doc.html#36930](http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html" \l "36930" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)。

##### 当线程处于非运行（Run）状态

当线程处于下面的状况时，属于非运行状态：

当sleep方法被调用。

当wait方法被调用。

当被I/O阻塞，可能是文件或者网络等等。

当线程处于上述的状态时，使用前面介绍的方法就不可用了。这个时候，我们可以使用[interrupt()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "interrupt()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)来打破阻塞的情况，如：

public void stop() {

Thread tmpBlinker = blinker;

blinker = null;

if (tmpBlinker != null) {

tmpBlinker.interrupt();

}

}

当[interrupt()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "interrupt()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)被调用的时候，[InterruptedException](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/InterruptedException.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)将被抛出，所以你可以再run方法中捕获这个异常，让线程安全退出：

try {

....

wait();

} catch (InterruptedException iex) {

throw new RuntimeException("Interrupted",iex);

}

#### 阻塞的I/O

当线程被I/O阻塞的时候，调用[interrupt()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "interrupt()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)的情况是依赖与实际运行的平台的。在Solaris和Linux平台上将会抛出InterruptedIOException的异常，但是Windows上面不会有这种异常。所以，我们处理这种问题不能依靠于平台的实现。如：

package com.cnblogs.gpcuster

import java.net.\*;import java.io.\*;

public abstract class InterruptibleReader extends Thread {

private Object lock = new Object( );

private InputStream is;

private boolean done;

private int buflen;

protected void processData(byte[] b, int n) { }

class ReaderClass extends Thread {

public void run( ) {

byte[] b = new byte[buflen];

while (!done) {

try {

int n = is.read(b, 0, buflen);

processData(b, n);

} catch (IOException ioe) {

done = true;

}

}

synchronized(lock) {

lock.notify( );

}

}

}

public InterruptibleReader(InputStream is) {

this(is, 512);

}

public InterruptibleReader(InputStream is, int len) {

this.is = is;

buflen = len;

}

public void run( ) {

ReaderClass rc = new ReaderClass( );

synchronized(lock) {

rc.start( );

while (!done) {

try {

lock.wait( );

} catch (InterruptedException ie) {

done = true;

rc.interrupt( );

try {

is.close( );

} catch (IOException ioe) {}

}

}

}

}

}

另外，我们也可以使用[InterruptibleChannel](http://java.sun.com/j2se/1.5.0/docs/api/java/nio/channels/InterruptibleChannel.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)接口。 实现了[InterruptibleChannel](http://java.sun.com/j2se/1.5.0/docs/api/java/nio/channels/InterruptibleChannel.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)接口的类可以在阻塞的时候抛出[ClosedByInterruptException](http://java.sun.com/j2se/1.5.0/docs/api/java/nio/channels/ClosedByInterruptException.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)。如：

package com.cnblogs.gpcuster

import java.io.BufferedReader;import java.io.FileDescriptor;import java.io.FileInputStream;import java.io.InputStream;import java.io.InputStreamReader;import java.nio.channels.Channels;

public class InterruptInput {

static BufferedReader in = new BufferedReader(

new InputStreamReader(

Channels.newInputStream(

(new FileInputStream(FileDescriptor.in)).getChannel())));

public static void main(String args[]) {

try {

System.out.println("Enter lines of input (user ctrl+Z Enter to terminate):");

System.out.println("(Input thread will be interrupted in 10 sec.)");

// interrupt input in 10 sec

(new TimeOut()).start();

String line = null;

while ((line = in.readLine()) != null) {

System.out.println("Read line:'"+line+"'");

}

} catch (Exception ex) {

System.out.println(ex.toString()); // printStackTrace();

}

}

public static class TimeOut extends Thread {

int sleepTime = 10000;

Thread threadToInterrupt = null;

public TimeOut() {

// interrupt thread that creates this TimeOut.

threadToInterrupt = Thread.currentThread();

setDaemon(true);

}

public void run() {

try {

sleep(10000); // wait 10 sec

} catch(InterruptedException ex) {/\*ignore\*/}

threadToInterrupt.interrupt();

}

}

}

这里还需要注意一点，当线程处于写文件的状态时，调用[interrupt()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html" \l "interrupt()" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)不会中断线程。

#### 参考资料

[How to Stop a Thread or a Task](http://forward.com.au/javaProgramming/HowToStopAThread.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

[Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

****不提倡的stop()方法****  
臭名昭著的stop()停止线程的方法已不提倡使用了,原因是什么呢?  
　当在一个线程对象上调用stop()方法时，这个线程对象所运行的线程就会立即停止，并抛出特殊的ThreadDeath()异常。这里的“立即”因为太“立即”了，  
假如一个线程正在执行：

synchronized void {

x = 3;

y = 4;

}

　　由于方法是同步的，多个线程访问时总能保证x,y被同时赋值，而如果一个线程正在执行到x = 3;时，被调用了 stop()方法，即使在同步块中，它也干脆地stop了，这样就产生了不完整的残废数据。而多线程编程中最最基础的条件要保证数据的完整性，所以请忘记 线程的stop方法，以后我们再也不要说“停止线程”了。

　　 如何才能“结束”一个线程？

**interupt()中断线程**

一个线程从运行到真正的结束，应该有三个阶段：

1. 正常运行.
2. 处理结束前的工作,也就是准备结束.
3. 结束退出.

那么如何让一个线程结束呢？既然不能调用stop，可用的只的interrupt()方法。但interrupt()方法只是改变了线程的运行状态，如何让它退出运行？对于一般逻辑，只要线程状态已经中断，我们就可以让它退出，这里我们定义一个线程类ThreadA,所以这样的语句可以保证线程在中断后就能结束运行：

while(!isInterrupted()){

正常逻辑

}

,一个测试类,ThreadDemo  
　　这样ThreadDemo调用interrupt()方法，isInterrupted()为true，就会退出运行。但是如果线程正在执行wait,sleep,join方法，你调用interrupt()方法，这个逻辑就不完全了。  
我们可以这样处理:

public void run(){

while(!isInterrupted()){

try{

正常工作

}catch(InterruptedException e){

//nothing

}

}

}

}

想一想，如果一个正在sleep的线程，在调用interrupt后，会如何？wait方法检查到isInterrupted()为true，抛出异常， 而你又没有处理。而一个抛出了InterruptedException的线程的状态马上就会被置为非中断状态，如果catch语句没有处理异常，则下一 次循环中isInterrupted()为false，线程会继续执行，可能你N次抛出异常，也无法让线程停止。  
这个错误情况的实例代码  
ThreadA

IMG_256  
IMG_257public class ThreadA extends Thread {  
IMG_258   int count=0;  
IMG_259   public void run(){  
IMG_260       System.out.println(getName()+"将要运行...");  
IMG_261       while(!this.isInterrupted()){  
IMG_262           System.out.println(getName()+"运行中"+count++);  
IMG_263           try{  
IMG_264               Thread.sleep(400);  
IMG_265           }catch(InterruptedException e){  
IMG_266               System.out.println(getName()+"从阻塞中退出...");  
IMG_267               System.out.println("this.isInterrupted()="+this.isInterrupted());  
IMG_268  
IMG_269           }  
IMG_270       }  
IMG_271       System.out.println(getName()+"已经终止!");  
IMG_272   }  
IMG_273}  
IMG_274

ThreadDemo

IMG_275public class ThreadDemo {  
IMG_276      
IMG_277    public static void main(String argv[])throws InterruptedException{  
IMG_278        ThreadA ta=new ThreadA();  
IMG_279        ta.setName("ThreadA");  
IMG_280        ta.start();  
IMG_281        Thread.sleep(2000);  
IMG_282        System.out.println(ta.getName()+"正在被中断...");  
IMG_283        ta.interrupt();  
IMG_284        System.out.println("ta.isInterrupted()="+ta.isInterrupted());  
IMG_285    }  
IMG_286  
IMG_287}  
IMG_288

　那么如何能确保线程真正停止？在线程同步的时候我们有一个叫“二次惰性检测”(double check)，能在提高效率的基础上又确保线程真正中同步控制中。那么我把线程正确退出的方法称为“双重安全退出”，即不以isInterrupted ()为循环条件。而以一个标记作为循环条件：  
正确的ThreadA代码是:

IMG_289   
IMG_290  
IMG_291public class ThreadA extends Thread {  
IMG_292    private boolean isInterrupted=false;  
IMG_293   int count=0;  
IMG_294     
IMG_295   public void interrupt(){  
IMG_296       isInterrupted = true;  
IMG_297       super.interrupt();  
IMG_298      }  
IMG_299     
IMG_300   public void run(){  
IMG_301       System.out.println(getName()+"将要运行...");  
IMG_302       while(!isInterrupted){  
IMG_303           System.out.println(getName()+"运行中"+count++);  
IMG_304           try{  
IMG_305               Thread.sleep(400);  
IMG_306           }catch(InterruptedException e){  
IMG_307               System.out.println(getName()+"从阻塞中退出...");  
IMG_308               System.out.println("this.isInterrupted()="+this.isInterrupted());  
IMG_309  
IMG_310           }  
IMG_311       }  
IMG_312       System.out.println(getName()+"已经终止!");  
IMG_313   }  
IMG_314}  
IMG_315

    在java多线程编程中，线程的终止可以说是一个必然会遇到的操作。但是这样一个常见的操作其实并不是一个能够轻而易举实现的操作，而且在某些场景下情况会变得更复杂更棘手。

        Java标准API中的Thread类提供了stop方法可以终止线程，但是很遗憾，这种方法不建议使用，原因是这种方式终止线程中断临界区代码执行，并会释放线程之前获取的监控器锁，这样势必引起某些对象状态的不一致（因为临界区代码一般是原子的，不会被干扰的），具体原因可以参考资料[1]。这样一来，就必须根据线程的特点使用不同的替代方案以终止线程。根据停止线程时线程执行状态的不同有如下停止线程的方法。

# 1 处于运行状态的线程停止

        处于运行状态的线程就是常见的处于一个循环中不断执行业务流程的线程，这样的线程需要通过设置停止变量的方式，在每次循环开始处判断变量是否改变为停止，以达到停止线程的目的，比如如下代码框架：

****[java]**** [view plain](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "view plain" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)[copy](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "copy" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

1. **private** **volatile** Thread blinker;
2. **public** **void** stop() {
3. blinker = **null**;
4. }
5. **public** **void** run() {
6. Thread thisThread = Thread.currentThread();
7. **while** (blinker == thisThread) {
8. **try** {
9. //业务流程
10. } **catch** (Exception e){}
11. }
12. }<span style="font-size:12px;">
13. </span>

        如果主线程调用该线程对象的stop方法，blinker对象被设置为null，则线程的下次循环中blinker！=thisThread，因而可以退出循环，并退出run方法而使线程结束。将引用变量blinker的类型前加上volatile关键字的目的是防止编译器对该变量存取时的优化，这种优化主要是缓存对变量的修改，这将使其他线程不会立刻看到修改后的blinker值，从而影响退出。此外，Java标准保证被volatile修饰的变量的读写都是原子的。

        上述的Thread类型的blinker完全可以由更为简单的boolean类型变量代替。

# 2 即将或正在处于非运行态的线程停止

        线程的非运行状态常见的有如下两种情况：

可中断等待：线程调用了sleep或wait方法，这些方法可抛出InterruptedException；

Io阻塞：线程调用了IO的read操作或者socket的accept操作，处于阻塞状态。

## 2.1 处于可中断等待线程的停止

        如果线程调用了可中断等待方法，正处于等待状态，则可以通过调用Thread的interrupt方法让等待方法抛出InterruptedException异常，然后在循环外截获并处理异常，这样便跳出了线程run方法中的循环，以使线程顺利结束。

        上述的stop方法中需要做的修改就是在设置停止变量之后调用interrupt方法：

****[java]**** [view plain](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "view plain" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)[copy](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "copy" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

1. **private** **volatile** Thread blinker;
2. **public** **void** stop() {
3. Thread tmp = blinker;
4. blinker = **null**;
5. **if**(tmp!=**null**){
6. tmp.interrupt();
7. }
8. }

         特别的，Thread对象的interrupt方法会设置线程的interruptedFlag，所以我们可以通过判断Thread对象的isInterrupted方法的返回值来判断是否应该继续run方法内的循环，从而代替线程中的volatile停止变量。这时的上述run方法的代码框架就变为如下：

****[java]**** [view plain](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "view plain" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)[copy](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "copy" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

1. **public** **void** run() {
2. **while** (!Thread.currentThread().isInterrupted()) {
3. **try** {
4. //业务流程
5. } **catch** (Exception e){}
6. }
7. }

        需要注意的是Thread对象的isInterrupted不会清除interrupted标记，但是Thread对象的interrupted方法（与interrupt方法区别）会清除该标记。

## 2.2 处于IO阻塞状态线程的停止

         Java中的输入输出流并没有类似于Interrupt的机制，但是Java的InterruptableChanel接口提供了这样的机制，任何实现了InterruptableChanel接口的类的IO阻塞都是可中断的，中断时抛出ClosedByInterruptedException，也是由Thread对象调用Interrupt方法完成中断调用。IO中断后将关闭通道。

        以文件IO为例，构造一个可中断的文件输入流的代码如下：

****[java]**** [view plain](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "view plain" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)[copy](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "copy" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

1. **new** InputStreamReader(
2. Channels.newInputStream(
3. (**new** FileInputStream(FileDescriptor.in)).getChannel())));

         实现InterruptableChanel接口的类包括FileChannel,ServerSocketChannel, SocketChannel, Pipe.SinkChannel andPipe.SourceChannel，也就是说，原则上可以实现文件、Socket、管道的可中断IO阻塞操作。

        虽然解除IO阻塞的方法还可以直接调用IO对象的Close方法，这也会抛出IO异常。但是InterruptableChanel机制能够使处于IO阻塞的线程能够有一个和处于中断等待的线程一致的线程停止方案。

# 3 处于大数据IO读写中的线程停止

         处于大数据IO读写中的线程实际上处于运行状态，而不是等待或阻塞状态，因此上面的interrupt机制不适用。线程处于IO读写中可以看成是线程运行中的一种特例。停止这样的线程的办法是强行close掉io输入输出流对象，使其抛出异常，进而使线程停止。

        最好的建议是将大数据的IO读写操作放在循环中进行，这样可以在每次循环中都有线程停止的时机，这也就将问题转化为如何停止正在运行中的线程的问题了。

# 4 在线程运行前停止线程

         有时，线程中的run方法需要足够健壮以支持在线程实际运行前终止线程的情况。即在Thread创建后，到Thread的start方法调用前这段时间，调用自定义的stop方法也要奏效。从上述的停止处于等待状态线程的代码示例中，stop方法并不能终止运行前的线程，因为在Thread的start方法被调用前，调用interrupt方法并不会将Thread对象的中断状态置位，这样当run方法执行时，currentThread的isInterrupted方法返回false，线程将继续执行下去。

         为了解决这个问题，不得不自己再额外创建一个volatile标志量，并将其加入run方法的最开头：

****[java]**** [view plain](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "view plain" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)[copy](http://blog.csdn.net/mikeszhang/article/details/8751355" \o "copy" \t "http://m.blog.csdn.net/anhuidelinger/article/details/_blank)

1. **public** **void** run() {
2. **if** (myThread == **null**) {
3. **return**; // stopped before started.
4. }
5. **while**(!Thread.currentThread().isInterrupted()){
6. //业务逻辑
7. }
8. }

         还有一种解决方法，也可以在run中直接使用该自定义标志量，而不使用isInterrupted方法判断线程是否应该停止。这种方法的run代码框架实际上和停止运行时线程的一样。