Thread.currentThread.interrupt()

转载 2016年10月09日 17:10:29 2301

Thread.currentThread.interrupt() 只对阻塞线程起作用，
当线程阻塞时，调用interrupt方法后，该线程会得到一个interrupt异常，可以通过对该异常的处理而退出线程
对于正在运行的线程，没有任何作用！

先看收集了别人的文章，全面的了解下java的中断：

中断线程

线程的thread.interrupt()方法是中断线程，将会设置该线程的中断状态位，即设置为true，中断的结果线程是死亡、还是等待新的任务或是继续运行至下一步，就取决于这个程序本身。线程会不时地检测这个中断标示位，以判断线程是否应该被中断（中断标示值是否为true）。它并不像stop方法那样会中断一个正在运行的线程。

判断线程是否被中断

判断某个线程是否已被发送过中断请求，请使用Thread.currentThread().isInterrupted()方法（因为它将线程中断标示位设置为true后，不会立刻清除中断标示位，即不会将中断标设置为false），而不要使用thread.interrupted()（该方法调用后会将中断标示位清除，即重新设置为false）方法来判断，下面是线程在循环中时的中断方式：

```
while(!Thread.currentThread().isInterrupted() && more work to do){
    do more work
}
```

如何中断线程

如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、thread.wait、1.5中的condition.await、以及可中断的通道上的 I/O 操作方法后可进入阻塞状态），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法（sleep、join、wait、1.5中的condition.await及可中断的通道上的 I/O 操作方法）调用处抛出InterruptedException异常，并且在抛出异常后立即将线程的中断标示位清除，即重新设置为false。抛出异常是为了线程从阻塞状态醒过来，并在结束线程前让程序员有足够的时间来处理中断请求。

注，synchronized在获锁的过程中是不能被中断的，意思是说如果产生了死锁，则不可能被中断（请参考后面的测试例子）。与synchronized功能相似的reentrantLock.lock()方法也是一样，它也不可中断的，即如果发生死锁，那么reentrantLock.lock()方法无法终止，如果调用时被阻塞，则它一直阻塞到它获取到锁为止。但是如果调用带超时的tryLock方法reentrantLock.tryLock(long
timeout, TimeUnit unit)，那么如果线程在等待时被中断，将抛出一个InterruptedException异常，这是一个非常有用的特性，因为它允许程序打破死锁。你也可以调用reentrantLock.lockInterruptibly()方法，它就相当于一个超时设为无限的tryLock方法。

没有任何语言方面的需求一个被中断的线程应该终止。中断一个线程只是为了引起该线程的注意，被中断线程可以决定如何应对中断。某些线程非常重要，以至于它们应该不理会中断，而是在处理完抛出的异常之后继续执行，但是更普遍的情况是，一个线程将把中断看作一个终止请求，这种线程的run方法遵循如下形式：

复制代码

```
public void run() {
    try {
        ...
        /*
         * 不管循环里是否调用过线程阻塞的方法如sleep、join、wait，这里还是需要加上
         * !Thread.currentThread().isInterrupted()条件，虽然抛出异常后退出了循环，显
         * 得用阻塞的情况下是多余的，但如果调用了阻塞方法但没有阻塞时，这样会更安全、更及时。
         */
        while (!Thread.currentThread().isInterrupted()&& more work to do) {
            do more work
        }
    } catch (InterruptedException e) {
```

```
        //线程在wait或sleep期间被中断了
    } finally {
        //线程结束前做一些清理工作
    }
}
```

复制代码

上面是while循环在try块里，如果try在while循环里时，因该在catch块里重新设置一下中断标示，因为抛出InterruptedException异常后，中断标示位会自动清除，此时应该这样：

复制代码

```
public void run() {
    while (!Thread.currentThread().isInterrupted()&& more work to do) {
        try {
            ...
            sleep(delay);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();//重新设置中断标示
        }
    }
}
```

复制代码

底层中断异常处理方式

另外不要在你的底层代码里捕获InterruptedException异常后不处理，会处理不当，如下：

复制代码

```
void mySubTask(){
    ...
    try{
        sleep(delay);
    }catch(InterruptedException e){}//不要这样做
    ...
}
```

复制代码

如果你不知道抛InterruptedException异常后如何处理，那么你有如下好的建议处理方式：
1、在catch子句中，调用Thread.currentThread.interrupt()来设置中断状态（因为抛出异常后中断标示会被清除），让外界通过判断Thread.currentThread().isInterrupted()标示来决定是否终止线程还是继续下去，应该这样做：

复制代码

```
void mySubTask() {
    ...
    try {
        sleep(delay);
    } catch (InterruptedException e) {
        Thread.currentThread().isInterrupted();
    }
    ...
}
```

复制代码

2、或者，更好的做法就是，不使用try来捕获这样的异常，让方法直接抛出：

```
void mySubTask() throws InterruptedException {
    ...
    sleep(delay);
    ...
}
```

中断应用

使用中断信号量中断非阻塞状态的线程

中断线程最好的，最受推荐的方式是，使用共享变量（shared variable）发出信号，告诉线程必须停止正在运行的任务。线程必须周期性的核查这一变量，然后有秩序地中止任务。Example2描述了这一方式：

复制代码

```
class Example2 extends Thread {
    volatile boolean stop = false;// 线程中断信号量

    public static void main(String args[]) throws Exception {
        Example2 thread = new Example2();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        // 设置中断信号量
        thread.stop = true;
        Thread.sleep(3000);
        System.out.println("Stopping application...");
    }

    public void run() {
        // 每隔一秒检测一下中断信号量
        while (!stop) {
            System.out.println("Thread is running...");
            long time = System.currentTimeMillis();
            /*
             * 使用while循环模拟 sleep 方法，这里不要使用sleep，否则在阻塞时会 抛
             * InterruptedException异常而退出循环，这样while检测stop条件就不会执行，
             * 失去了意义。
             */
            while ((System.currentTimeMillis() - time < 1000)) {}
        }
        System.out.println("Thread exiting under request...");
    }
}
```

复制代码

使用thread.interrupt()中断非阻塞状态线程

虽然Example2该方法要求一些编码，但并不难实现。同时，它给予线程机会进行必要的清理工作。这里需注意一点的是需将共享变量定义成volatile 类型或将对它的一切访问封入同步的块/方法（synchronized blocks/methods）中。上面是中断一个非阻塞状态的线程的常见做法，但对非检测isInterrupted()条件会更简洁：

复制代码

```
class Example2 extends Thread {
    public static void main(String args[]) throws Exception {
        Example2 thread = new Example2();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        // 发出中断请求
        thread.interrupt();
        Thread.sleep(3000);
        System.out.println("Stopping application...");
    }

    public void run() {
        // 每隔一秒检测是否设置了中断标示
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread is running...");
            long time = System.currentTimeMillis();
            // 使用while循环模拟 sleep
            while ((System.currentTimeMillis() - time < 1000) ) {
            }
        }
        System.out.println("Thread exiting under request...");
    }
}
```

到目前为止一切顺利！但是，当线程等待某些事件发生而被阻塞，又会发生什么？当然，如果线程被阻塞，它便不能核查共享变量，也就不能停止。这在许多情况下会发生，例如调用Object.wait()、ServerSocket.accept()和DatagramSocket.receive()时，这里仅举出一些。

他们都可能永久的阻塞线程。即使发生超时，在超时期满之前持续等待也是不可行和不适当的，所以，要使用某种机制使得线程更早地退出被阻塞的状态。下面就来看一下中断阻塞线程技术。

使用thread.interrupt()中断阻塞状态线程

Thread.interrupt()方法不会中断一个正在运行的线程。这一方法实际上完成的是，设置线程的中断标示位，在线程受到阻塞的地方（如调用sleep、wait、join等地方）抛出一个异常InterruptedException，并且中断状态也将被清除，这样线程就得以退出阻塞的状态。下面是具体实现：

```
class Example3 extends Thread {
    public static void main(String args[]) throws Exception {
        Example3 thread = new Example3();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        thread.interrupt();// 等中断信号量设置后再调用
        Thread.sleep(3000);
        System.out.println("Stopping application...");
    }

    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread running...");
            try {
                /*
                 * 如果线程阻塞，将不会去检查中断信号量stop变量，所 以thread.interrupt()
                 * 会使阻塞线程从阻塞的地方抛出异常，让阻塞线程从阻塞状态逃离出来，并
                 * 进行异常块进行 相应的处理
                 */
                Thread.sleep(1000);// 线程阻塞，如果线程收到中断操作信号将抛出异常
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted...");
                /*
                 * 如果线程在调用 Object.wait()方法，或者该类的 join() 、sleep()方法
                 * 过程中受阻，则其中断状态将被清除
                 */
                System.out.println(this.isInterrupted());// false

                //中不中断由自己决定，如果需要真真中断线程，则需要重新设置中断位，如果
                //不需要，则不用调用
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Thread exiting under request...");
    }
}
```

一旦Example3中的Thread.interrupt()被调用，线程便收到一个异常，于是逃离了阻塞状态并确定应该停止。上面我们还可以使用共享信号量来替换!Thread.currentThread().isInterrupted()条件，但不如它简洁。

死锁状态线程无法被中断

Example4试着去中断处于死锁状态的两个线程，但这两个线都没有收到任何中断信号（抛出异常），所以interrupt()方法是不能中断死锁线程的，因为锁定的位置根本无法抛出异常：

```
class Example4 extends Thread {
    public static void main(String args[]) throws Exception {
```

```
        final Object lock1 = new Object();
        final Object lock2 = new Object();
        Thread thread1 = new Thread() {
            public void run() {
                deathLock(lock1, lock2);
            }
        };
        Thread thread2 = new Thread() {
            public void run() {
                // 注意，这里在交换了一下位置
                deathLock(lock2, lock1);
            }
        };
        System.out.println("Starting thread...");
        thread1.start();
        thread2.start();
        Thread.sleep(3000);
        System.out.println("Interrupting thread...");
        thread1.interrupt();
        thread2.interrupt();
        Thread.sleep(3000);
        System.out.println("Stopping application...");
    }

    static void deathLock(Object lock1, Object lock2) {
        try {
            synchronized (lock1) {
                Thread.sleep(10);// 不会在这里死掉
                synchronized (lock2) {// 会锁在这里，虽然阻塞了，但不会抛异常
                    System.out.println(Thread.currentThread());
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

复制代码

中断I/O操作

然而，如果线程在I/O操作进行时被阻塞，又会如何？I/O操作可以阻塞线程一段相当长的时间，特别是牵扯到网络应用时。例如，服务器可能需要等待一个请求（request），又或者，一个网络应用程序可能要等待远端主机的响应。

实现此InterruptibleChannel接口的通道是可中断的：如果某个线程在可中断通道上因调用某个阻塞的 I/O 操作（常见的操作一般有这些：serverSocketChannel. accept()、socketChannel.connect、socketChannel.open、socketChannel.read、socketChannel.write、fileChannel.read、fileChannel.write）而进入阻塞状态，而另一个线程又调用了该阻塞线程的
interrupt 方法，这将导致该通道被关闭，并且已阻塞线程接将会收到ClosedByInterruptException，并且设置已阻塞线程的中断状态。另外，如果已设置某个线程的中断状态并且它在通道上调用某个阻塞的 I/O 操作，则该通道将关闭并且该线程立即接收到 ClosedByInterruptException；并仍然设置其中断状态。如果情况是这样，其代码的逻辑和第三个例子中的是一样的，只是异常不同而已。

如果你正使用通道（channels）（这是在Java 1.4中引入的新的I/O API），那么被阻塞的线程将收到一个ClosedByInterruptException异常。但是，你可能正使用Java1.0之前就存在的传统的I/O，而且要求更多的工作。既然这样，Thread.interrupt()将不起作用，因为线程将不会退出被阻塞状态。Example5描述了这一行为。尽管interrupt()被调用，线程也不会退出被阻塞状态，比如ServerSocket的accept方法根本不抛出异常。

很幸运，Java平台为这种情形提供了一项解决方案，即调用阻塞该线程的套接字的close()方法。在这种情形下，如果线程被I/O操作阻塞，当调用该套接字的close方法时，该线程在调用accept地方法将接收到一个SocketException（SocketException为IOException的子异常）异常，这与使用interrupt()方法引起一个InterruptedException异常被抛出非常相似，（注，如果是流因读写阻塞后，调用流的close方法也会被阻塞，根本不能调用，更不会抛IOExcepiton，此种情况下怎样中断？我想可以转换为通道来操作流可以解决，比如文件通道）。下面是具体实现：

复制代码

```
class Example6 extends Thread {
    volatile ServerSocket socket;

    public static void main(String args[]) throws Exception {
        Example6 thread = new Example6();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        Thread.currentThread().interrupt();// 再调用interrupt方法
        thread.socket.close();// 再调用close方法
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }
        System.out.println("Stopping application...");
    }

    public void run() {
        try {
            socket = new ServerSocket(8888);
        } catch (IOException e) {
            System.out.println("Could not create the socket...");
            return;
        }
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Waiting for connection...");
            try {
                socket.accept();
            } catch (IOException e) {
                System.out.println("accept() failed or interrupted...");
                Thread.currentThread().interrupt();//重新设置中断标示位
            }
        }
        System.out.println("Thread exiting under request...");
    }
}
```

复制代码

一、没有任何语言方面的需求一个被中断的线程应该终止。中断一个线程只是为了引起该线程的注意，被中断线程可以决定如何应对中断。

二、对于处于sleep，join等操作的线程，如果被调用interrupt()后，会抛出InterruptedException，然后线程的中断标志位会由true重置为false，因为线程为了处理异常已经重新处于就绪状态。

三、不可中断的操作，包括进入synchronized段以及Lock.lock()，inputSteam.read()等，调用interrupt()对于这几个问题无效，因为它们都不抛出中断异常。如果拿不到资源，它们会无限期阻塞下去。

对于Lock.lock()，可以改用Lock.lockInterruptibly()，可被中断的加锁操作，它可以抛出中断异常。等同于等待时间无限长的Lock.tryLock(long time, TimeUnit unit)。

对于inputStream等资源，有些(实现了interruptibleChannel接口)可以通过close()方法将资源关闭，对应的阻塞也会被放开。

首先，看看Thread类里的几个方法：

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。
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。

上面列出了与中断有关的几个方法及其行为，可以看到interrupt是中断线程。如果不了解Java的中断机制，这样的一种解释极容易造成误解，认为调用了线程的interrupt方法就一定会中断线程。

其实，Java的中断是一种协作机制。也就是说调用线程对象的interrupt方法并不一定就中断了正在运行的线程，它只是要求线程自己在合适的时机中断自己。每个线程都有一个boolean的中断状态（这个状态不在Thread的属性上），interrupt方法仅仅只是将该状态置为true。

比如对正常运行的线程调用interrupt()并不能终止他，只是改变了interrupt标示符。

一般说来，如果一个方法声明抛出InterruptedException，表示该方法是可中断的,比如wait,sleep,join，也就是说可中断方法会对interrupt调用做出响应（例如sleep响应interrupt的操作包括清除中断状态，抛出InterruptedException）,异常都是由可中断方法自己抛出来的，并不是直接由interrupt方法直接引起的。

Object.wait, Thread.sleep方法，会不断的轮询监听 interrupted 标志位，发现其设置为true后，会停止阻塞并抛出 InterruptedException异常。

看了以上的说明，对java中断的使用肯定是会了，但我想知道的是阻塞了的线程是如何通过interuppt方法完成停止阻塞并抛出interruptedException的，这就要看Thread中native的interuppt0方法了。

第一步学习Java的JNI调用Native方法。

第二步下载openjdk的源代码，找到目录结构里的openjdk-src\jdk\src\share\native\java\lang\Thread.c文件。

复制代码

```
#include "jni.h"
#include "jvm.h"

#include "java_lang_Thread.h"

#define THD "Ljava/lang/Thread;"
#define OBJ "Ljava/lang/Object;"
#define STE "Ljava/lang/StackTraceElement;"

#define ARRAY_LENGTH(a) (sizeof(a)/sizeof(a[0]))

static JNINativeMethod methods[] = {
    {"start0",           "()V",        (void *)&JVM_StartThread},
    {"stop0",            "(" OBJ ")V", (void *)&JVM_StopThread},
    {"isAlive",          "()Z",        (void *)&JVM_IsThreadAlive},
    {"suspend0",         "()V",        (void *)&JVM_SuspendThread},
    {"resume0",          "()V",        (void *)&JVM_ResumeThread},
    {"setPriority0",     "(I)V",       (void *)&JVM_SetThreadPriority},
    {"yield",            "()V",        (void *)&JVM_Yield},
    {"sleep",            "(J)V",       (void *)&JVM_Sleep},
    {"currentThread",    "()" THD,     (void *)&JVM_CurrentThread},
    {"countStackFrames", "()I",        (void *)&JVM_CountStackFrames},
    {"interrupt0",       "()V",        (void *)&JVM_Interrupt},
    {"isInterrupted",    "(Z)Z",       (void *)&JVM_IsInterrupted},
    {"holdsLock",        "(" OBJ ")Z", (void *)&JVM_HoldsLock},
    {"getThreads",        "()[" THD,   (void *)&JVM_GetAllThreads},
    {"dumpThreads",      "([" THD ")[[" STE, (void *)&JVM_DumpThreads},
};

#undef THD
#undef OBJ
#undef STE

JNIEXPORT void JNICALL
Java_java_lang_Thread_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls, methods, ARRAY_LENGTH(methods));
}
```

复制代码

暂时还看不太懂，先去学习一下C的一些基础。

未完待续...
