线程中断机制（interrupt）

优雅地中断线程，是一门艺术

众所周知，Thread.stop、Thread.suspend、Thread.resume都已经被废弃了。因为它们太暴力了，是不安全的，这种暴力中断线程是一种不安全的操作。举个例子来说明其可能造成的问题：

public class ThreadTest {

public static void main(String[] args) throws InterruptedException {

StopThread thread = new StopThread();

thread.start();

// 休眠1秒，确保线程进入运行

Thread.sleep(1000);

// 暂停线程

thread.stop();

// thread.interrupt();

// 确保线程已经销毁

while (thread.isAlive()) { }

// 输出结果

thread.print();

}

private static class StopThread extends Thread {

private int x = 0; private int y = 0;

@Override

public void run() {

// 这是一个同步原子操作

synchronized (this) {

++x;

try {

// 休眠3秒,模拟耗时操作

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

++y;

}

}

public void print() {

System.out.println("x=" + x + " y=" + y);

}

}

}

上述代码中，run方法里是一个同步的原子操作，x和y必须要共同增加，然而这里如果调用thread.stop()方法强制中断线程，输出如下：

x=1 y=0

没有异常，也破坏了我们的预期。如果这种问题出现在我们的程序中，会引发难以预期的异常。因此这种不安全的方式很早就被废弃了，取而代之的是interrupt()。上述代码如果采用thread.interrupt()方法，输出结果如下：

x=1 y=1

java.lang.InterruptedException: sleep interrupted

at java.lang.Thread.sleep(Native Method)

at ThreadTest$StopThread.run(ThreadTest.java:28)

x=1,y=1这个结果是符合我们的预期的，同时还抛出了个异常，这个异常下文详说。

Interrupt()它基于[ 一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。]思想，是一个比较温柔的做法，它更类似一个标志位。其实作用不是中断线程，而是[ 通知线程应该中断了 ]，具体到底中断还是继续运行，应该由被通知的线程自己处理。

Interrupt()并不能真正地中断线程，这点要谨记。需要被调用地线程自己进行配合才行。也就是说，一个线程如果由被中断地需求，那么就需要这样做：

在正常运行任务时，经常检查本线程地中断标志位，如果被设置了中断标志就自行停止线程。

在调用阻塞方法时正确处理InterruptedException异常。（例如：catch异常后就结束线程。）

先看下Thread类interrupt相关地几个方法：

// 核心 interrupt 方法

public void interrupt() {

if (this != Thread.currentThread()) // 非本线程，需要检查权限

checkAccess();

synchronized (blockerLock) {

Interruptible b = blocker;

if (b != null) {

interrupt0(); // 仅仅设置interrupt标志位

b.interrupt(this); // 调用如 I/O 操作定义的中断方法

return;

}

}

interrupt0();

}

// 静态方法，这个方法有点坑，调用该方法调用后会清除中断状态。

public static boolean interrupted() {

return currentThread().isInterrupted(true);

}

// 这个方法不会清除中断状态

public boolean isInterrupted() {

return isInterrupted(false);

}

// 上面两个方法会调用这个本地方法，参数代表是否清除中断状态

private native boolean isInterrupted(boolean ClearInterrupted);

首先讲interrupt()方法：

1. Interrupt中断操作时，非自身打断需要先检测是否有中断权限，这由jvm的安全机制配置；
2. 如果线程处于sleep、wait、join等状态，那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常；
3. 如果线程处于I/O阻塞状态，将会抛出ClosedByInterruptException（IOException的子类）异常；
4. 如果线程在Selector上被阻塞，select方法将立即返回。
5. 如果非以上情况，将直接标记interrupt状态；

注意：interrupt操作不会打断所有阻塞，只有上述阻塞情况才在JVM的打断范围内，如处于锁阻塞的线程，不会受interrupt中断；

阻塞情况下中断，抛出异常后线程回复非中断状态，即interrupted=false

输出：

public class ThreadTest {

public static void main(String[] args) throws InterruptedException {

Thread t = new Thread(new Task("1"));

t.start();

t.interrupt();

}

static class Task implements Runnable{

String name;

public Task(String name) {

this.name = name;

}

@Override

public void run() {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("thread has been interrupt!");

}

System.out.println("isInterrupted: " + Thread.currentThread().isInterrupted());

System.out.println("task " + name + " is over");

}

}

}

thread has been interrupt!

isInterrupted: false

task 1 is over

调用Thread.interrupted()方法后线程恢复非中断状态

public class ThreadTest {

public static void main(String[] args) throws InterruptedException {

Thread t = new Thread(new Task("1"));

t.start();

t.interrupt();

}

static class Task implements Runnable{

String name;

public Task(String name) {

this.name = name;

}

@Override

public void run() {

System.out.println("first :" + Thread.interrupted());

System.out.println("second:" + Thread.interrupted());

System.out.println("task " + name + " is over");

}

}

}

输出结果：

first :true

second:false

task 1 is over

上述两种隐含的状态恢复操作，是符合常理的，因为线程标记为中断后，用户没有真正中断线程，必然将其恢复为false。理论上Thread.interrupted()调用后，如果已中断，应该执行退出操作，不会重复调用。