Java并发之线程中断

前面的几篇文章主要介绍了线程的一些最基本的概念，包括线程的间的冲突及其解决办法，以及线程间的协作机制。本篇主要来学习下Java中对线程中断机制的实现。在我们的程序中经常会有一些不达到目的不会退出的线程，例如：我们有一个下载程序线程，该线程在没有下载成功之前是不会退出的，若此时用户觉得下载速度慢，不想下载了，这时就需要用到我们的线程中断机制了，告诉线程，你不要继续执行了，准备好退出吧。当然，线程在不同的状态下遇到中断会产生不同的响应，有点会抛出异常，有的则没有变化，有的则会结束线程。本篇将从以下两个方面来介绍Java中对线程中断机制的具体实现：

- Java中对线程中断所提供的API支持
- 线程在不同状态下对于中断所产生的反应

一、Java中对线程中断所提供的API支持

在以前的jdk版本中，我们使用stop方法中断线程，但是现在的jdk版本中已经不再推荐使用该方法了，反而由以下三个方法完成对线程中断的支持。

```java
public boolean isInterrupted()

public void interrupt()

public static boolean interrupted()
```

每个线程都一个状态位用于标识当前线程对象是否是中断状态。isInterrupted是一个实例方法，主要用于判断当前线程对象的中断标志位是否被标记了，如果被标记了则返回true表示当前已经被中断，否则返回false。我们也可以看看它的实现源码：

```java
public boolean isInterrupted() {
        return isInterrupted(false);
}
```

```java
private native boolean isInterrupted(boolean ClearInterrupted);
```

底层调用的本地方法isInterrupted，传入一个boolean类型的参数，用于指定调用该方法之后是否需要清除该线程对象的中断标识位。从这里我们也可以看出来，调用isInterrupted并不会清除线程对象的中断标识位。

interrupt是一个实例方法，该方法用于设置当前线程对象的中断标识位。

interrupted是一个静态的方法，用于返回当前线程是否被中断。

```java
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

```java
private native boolean isInterrupted(boolean ClearInterrupted);
```

该方法用于判断当前线程是否被中断，并且该方法调用结束的时候会清空中断标识位。下面我们看看线程所处不同状态下对于中断操作的反应。

二、线程在不同状态下对于中断所产生的反应

线程一共6种状态，分别是NEW，RUNNABLE，BLOCKED，WAITING，TIMED_WAITING，TERMINATED（Thread类中有一个State枚举类型列举了线程的所有状态）。下面我们就将把线程分别置于上述的不同种状态，然后看看我们的中断操作对它们的影响。

1、NEW和TERMINATED

线程的new状态表示还未调用start方法，还未真正启动。线程的terminated状态表示线程已经运行终止。这两个状态下调用中断方法来中断线程的时候，Java认为毫无意义，所以并不会设置线程的中断标识位，什么事也不会发生。例如：

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

    Thread thread = new MyThread();
    System.out.println(thread.getState());

    thread.interrupt();

    System.out.println(thread.isInterrupted());
}
```

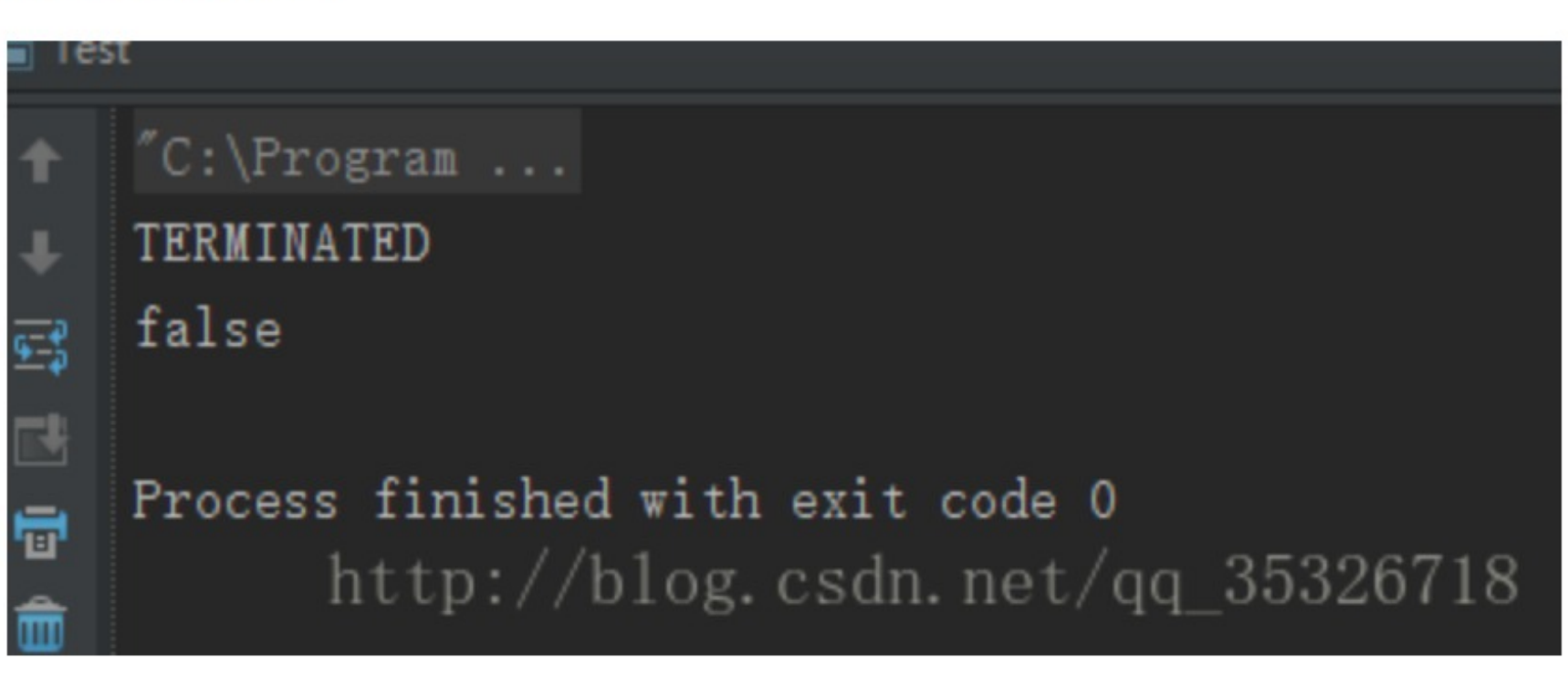
输出结果如下：

```
"C:\Program ...
NEW
false

Process finished with exit code 0
```

terminated状态：

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

     Thread thread = new MyThread();
     thread.start();

     thread.join();
     System.out.println(thread.getState());

     thread.interrupt();

    System.out.println(thread.isInterrupted());

}
```

输出结果如下：



```
"C:\Program ...
TERMINATED
false

Process finished with exit code 0
```

从上述的两个例子来看，对于处于new和terminated状态的线程对于中断是屏蔽的，也就是说中断操作对这两种状态下的线程是无效的。

2、RUNNABLE

如果线程处于运行状态，那么该线程的状态就是RUNNABLE，但是不一定所有处于RUNNABLE状态的线程都能获得CPU运行，在某个时间段，只能由一个线程占用CPU，那么其余的线程虽然状态是RUNNABLE，但是都没有处于运行状态。而我们处于RUNNABLE状态的线程在遭遇中断操作的时候只会设置该线程的中断标志位，并不会让线程实际中断，想要发现本线程已经被要求中断了则需要用程序去判断。例如：

```java
/*先定义一个线程类*/
public class MyThread extends Thread{

    @Override
    public void run(){
        while(true){
            //do something
        }
    }
}
```

```java
/*main函数启动线程*/
public static void main(String[] args) throws InterruptedException {

    Thread thread = new MyThread();
    thread.start();

    System.out.println(thread.getState());

    thread.interrupt();
    Thread.sleep(1000);//等到thread线程被中断之后
    System.out.println(thread.isInterrupted());

    System.out.println(thread.getState());
}
```

我们定义的线程始终循环做一些事情，主线程启动该线程并输出该线程的状态，然后调用中断方法中断该线程并再次输出该线程的状态。总的输出结果如下：

```
"C:\Program ...
RUNNABLE
true
RUNNABLE
```

可以看到在我们启动线程之后，线程状态变为RUNNABLE，中断之后输出中断标志，显然中断位已经被标记，但是当我们再次输出线程状态的时候发现，线程仍然处于RUNNABLE状态。很显然，处于RUNNBALE状态下的线程即便遇到中断操作，也只会设置中断标志位并不会实际中断线程运行。那么问题是，既然不能直接中断线程，我要中断标志有何用处？
这里其实Java将这种权力交给了我们的程序，Java给我们提供了一个中断标志位，我们的程序可以通过if判断中断标志位是否被设置来中断我们的程序而不是系统强制的中断。例如：

```java
/*修改MyThread类的run方法*/
public void run(){
    while(true){
        if (Thread.currentThread().isInterrupted()){
            System.out.println("exit MyThread");
            break;
        }
    }
}
```

线程一旦发现自己的中断标志为被设置了，立马跳出死循环。这样的设计好处就在于给了我们程序更大的灵活性。

3、BLOCKED

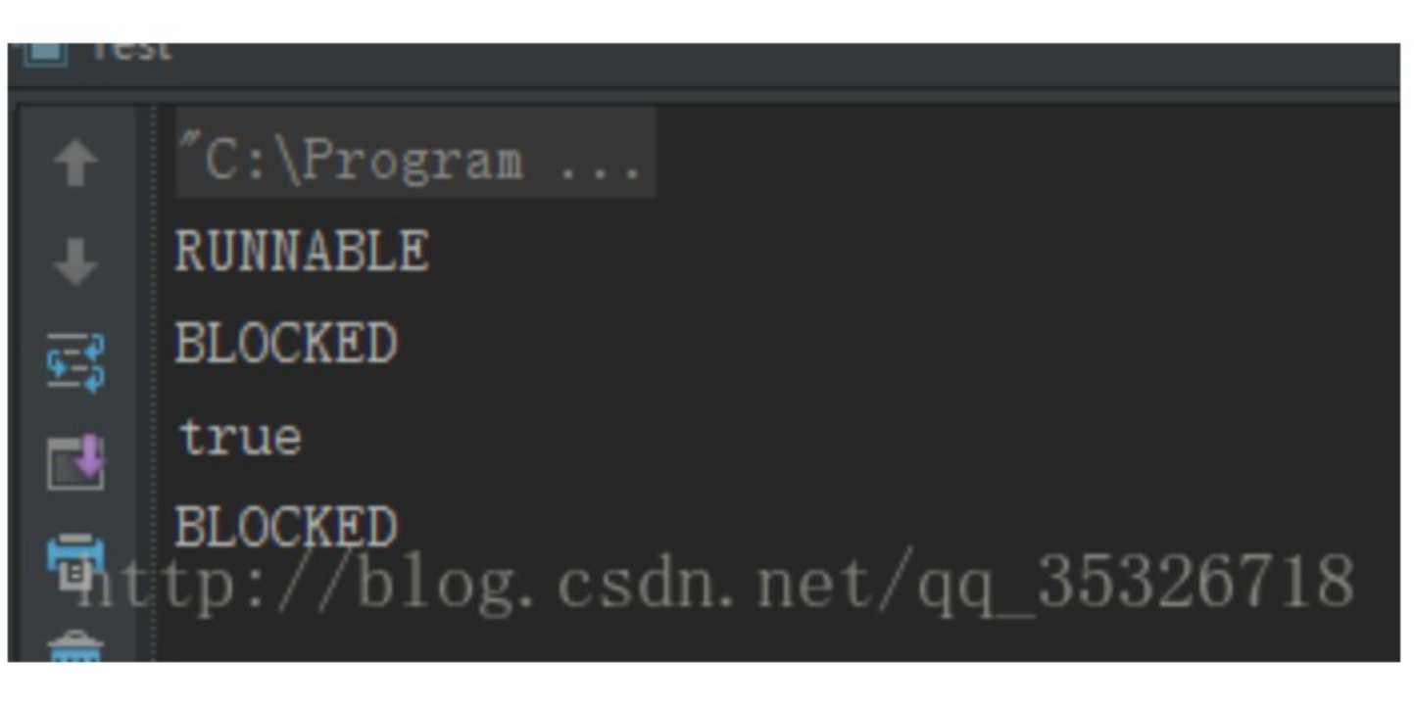
当线程处于BLOCKED状态说明该线程由于竞争某个对象的锁失败而被挂在了该对象的阻塞队列上了。那么此时发起中断操作不会对该线程产生任何影响，依然只是设置中断标志位。例如：

```java
/*自定义线程类*/
public class MyThread extends Thread{

    public synchronized static void doSomething(){
        while(true){
            //do something
        }
    }
    @Override
    public void run(){
        doSomething();
    }
}
```

这里我们自定义了一个线程类，run方法中主要就做一件事情，调用一个有锁的静态方法，该方法内部是一个死循环（占用该锁让其他线程阻塞）。

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

    Thread thread1 = new MyThread();
    thread1.start();

    Thread thread2 = new MyThread();
    thread2.start();

    Thread.sleep(1000);
    System.out.println(thread1.getState());
    System.out.println(thread2.getState());

    thread2.interrupt();
    System.out.println(thread2.isInterrupted());
    System.out.println(thread2.getState());
}
```

在我们的主线程中，我们定义了两个线程并按照定义顺序启动他们，显然thread1启动后便占用MyThread类锁，此后thread2在获取锁的时候一定失败，自然被阻塞在阻塞队列上，而我们对thread2进行中断，输出结果如下：



```
"C:\Program ...
RUNNABLE
BLOCKED
true
BLOCKED
```

从输出结果看来，thread2处于BLOCKED状态，执行中断操作之后，该线程仍然处于BLOCKED状态，但是中断标志位却已被修改。这种状态下的线程和处于RUNNABLE状态下的线程是类似的，给了我们程序更大的灵活性去判断和处理中断。

4、WAITING/TIMED_WAITING

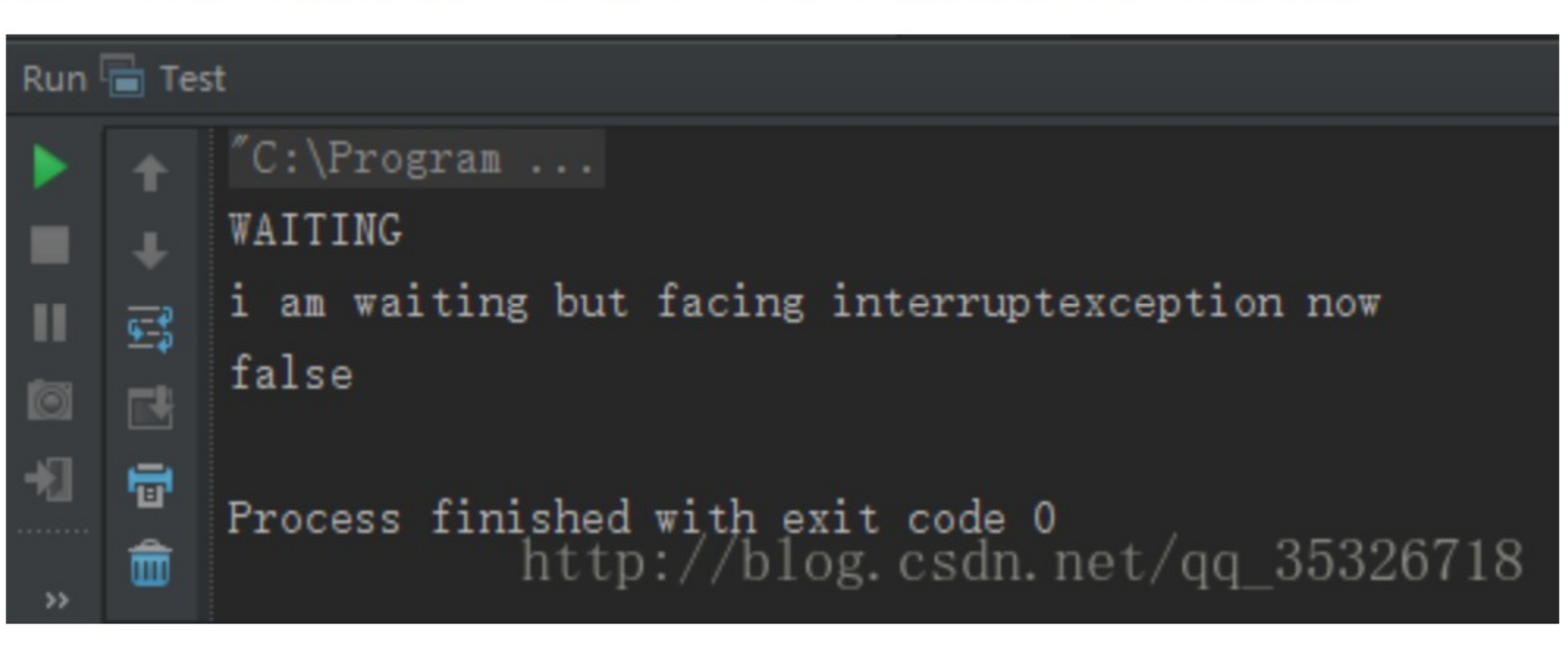
这两种状态本质上是同一种状态，只不过TIMED_WAITING在等待一段时间后会自动释放自己，而WAITING则是无限期等待，需要其他线程调用notify方法释放自己。但是他们都是线程在运行的过程中由于缺少某些条件而被挂起在某个对象的等待队列上。当这些线程遇到中断操作的时候，会抛出一个InterruptedException异常，并清空中断标志位。例如：

```java
/*定义一个线程类*/
public class MyThread extends Thread{

    @Override
    public void run(){
        synchronized (this){
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("i am waiting but facing interruptexception now");
            }
        }
    }
}
```

我们定义了一个线程类，其中run方法让当前线程阻塞到条件队列上，并且针对InterruptedException 进行捕获，如果遇到InterruptedException 异常则输出一行信息。

```java
/*main函数启动该线程*/
public static void main(String[] args) throws InterruptedException {

    Thread thread = new MyThread();
    thread.start();

    Thread.sleep(500);
    System.out.println(thread.getState());
    thread.interrupt();
    Thread.sleep(1000);
    System.out.println(thread.isInterrupted());
}
```

在main线程中我们启动一个MyThread线程，然后对其进行中断操作。



```
"C:\Program ...
WAITING
i am waiting but facing interruptexception now
false

Process finished with exit code 0
```

从运行结果看，当前程thread启动之后就被挂起到该线程对象的条件队列上，然后我们调用interrupt方法对该线程进行中断，输出了我们在catch中的输出语句，显然是捕获了InterruptedException异常，接着就看到该线程的中断标志位被清空。

综上所述，我们分别介绍了不同种线程的不同状态下对于中断请求的反应。NEW和TERMINATED对于中断操作几乎是屏蔽的，RUNNABLE和BLOCKED类似，对于中断操作只是设置中断标志位并没有强制终止线程，对于线程的终止权利依然在程序手中。WAITING/TIMED_WAITING状态下的线程对于中断操作是敏感的，他们会抛出异常并清空中断标志位。