# 线程中断

## 中断的作用

一个线程在未正常结束之前, 被强制终止是很危险的事情。因为它可能带来完全预料不到的严重后果，比如会**带着自己所持有的锁而永远的休眠**等。所以Thread.suspend, Thread.stop等方法都被Deprecated了。

既然不应该强制让线程结束，但是有时候又必须让一个线程终止，或者是让它结束某种等待的状态。比较安全的做法有两种：

1. 使用wait、notify机制；
2. 给线程发送一个中断信号。

通过这两种方式，让线程自己决定该如果处理。

中断的使用场景是什么样的呢？

* 在某个子线程中为了等待一些特定条件的到来, 调用了Thread.sleep(10000), 预期线程睡10秒之后自己醒来, 但是如果这个特定条件提前到来的话, 需要通知一个处于Sleep的线程。
* 线程通过调用子线程的join方法阻塞自己以等待子线程结束, 但是子线程运行过程中发现自己没办法在短时间内结束, 于是它需要想办法告诉主线程别不用再等待了。

一个简单的例子

public class **MyThread** extends **Thread**{

public void **run**(){

while(true){

if(Thread.**currentThread**().**isInterrupted**()){

System.out.**println**("Interrupted");

}

else{

System.out.**println**("Doing jobs");

}

}

}

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

MyThread myThread = new **MyThread**();

myThread.**start**();

Thread.**sleep**(3000);

myThread.**interrupt**();

}

}

main线程sleep(3000)的过程中，线程myThread没有被中断，一直输出Doing jobs。当执行了myThread.interrupt()方法后，myThread中的终端标志被设置为true，也就是Thread.**currentThread**().**isInterrupted**()返回值为true。线程并不会因为改变了中断标志而停止。线程会一直在输出Interrupted。

**没有任何语言方面的需求要求一个被中断的程序应该终止。**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。

线程需要通过不断的检查自己的中断状态，这样可以决定自己接下来应该做什么。

但是当线程被阻塞的时候，比如被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞时。调用线程的interrput()方法，可想而知，**没有占用CPU运行的线程是不可能给自己的中断状态置位的。这就会产生一个InterruptedException异常。**

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，他们都可能永久的阻塞线程（比如accept一直没有连接，那么这个线程就阻塞到了这里）。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。**

Thread.interrupt()方法不会中断一个正在运行的线程。这个方法实际上完成的是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞，那么，它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。

因此，如果线程被上述几种方法阻塞，正确停止线程的方式是**设置共享变量，并调用interrupt()方法**。如果线程没有被阻塞，线程调用interrupt方法将不起作用，因为它只是修改了中断状态；如果线程被阻塞了，那么线程会得到异常（线程应当事先预备好处理此异常），接着线程就可以逃离阻塞状态。在任何一种情况中，最后线程都将检查共享变量然后再停止。

public class **WhyInterupt** extends **Thread** {

volatile boolean stop = false;

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

WhyInterupt thread = new **WhyInterupt**();

System.out.**println**("Starting thread");

thread.**start**();

Thread.**sleep**(3000);

System.out.**println**("Asking thread to stop");

thread.stop = true; *//线程阻塞时，也就是线程调用了sleep()方法时，是不会检查这个变量的。紧接着调用interrupt方法会让线程抛出异常，线程捕获异常并处理，下一个while的条件就不会满足，退出while，线程也就结束了。*

thread.**interrupt**();

Thread.**sleep**(3000);

System.out.**println**("Stopping application");

return;

}

@Override

public void **run**() {

while(!stop){ *//stop为true的时候退出while*

System.out.**println**("Thread running");

try {

Thread.**sleep**(2000);

} catch (InterruptedException e) {

System.out.**println**("Thread interrupted");

*// e.printStackTrace();*

}

}

System.out.**println**("Thread exiting under request");

}

}

把握几个重点：stop变量、run方法中的sleep()、interrupt()、InterruptedException。串接起来就是这个意思：当我们在run方法中调用sleep（或其他阻塞线程的方法）时，如果线程阻塞的时间过长，比如10s，那**在这10s内，线程阻塞，run方法不被执行**，但是如果在这10s内，**stop被设置成true，表明要终止这个线程，但是，现在线程是阻塞的，它的run方法不能执行，自然也就不能检查stop，所以线程不能终止**，这个时候，我们就可以用interrupt()方法了：我们在thread.stop = true;语句后调用thread.interrupt()方法，该方法将在线程阻塞时抛出一个中断信号，该信号将被catch语句捕获到，一旦捕获到这个信号，线程就提前终结自己的阻塞状态，这样，它就能够再次运行run 方法了，然后检查到stop = true，while循环就不会再被执行，在执行了while后面的清理工作之后，run方法执行完 毕，线程终止。

但是需要注意的是，不是所有的阻塞方法收到中断后都可以取消阻塞状态, 输入和输出流类会阻塞等待 I/O 完成，但是它们不抛出 InterruptedException，而且在被中断的情况下也不会退出阻塞状态。

## 使用中断

判断线程的中断是否停止，Java提供了两个方法：

1. interrupted()：测试当前线程是否已经中断；
2. isInterrupted()：测试线程是否已经中断。

**中断是通过调用Thread.interrupt()方法来做的. 这个方法通过修改了被调用线程的中断状态来告知那个线程, 说它被中断了. 对于非阻塞中的线程, 只是改变了中断状态, 即Thread.isInterrupted()将返回true; 对于可取消的阻塞状态中的线程, 比如等待在这些函数上的线程, Thread.sleep(), Object.wait(), Thread.join(), 这个线程收到中断信号后, 会抛出InterruptedException, 同时会把中断状态置回为true.但调用Thread.interrupted()会对中断状态进行复位。**

public static boolean **interrupted**() {}

public boolean **isInterrupted**() {}

interrupted方法是测试当前线程是否已经中断，当前线程的意思是调用了interrupted方法的线程。

public static void **main**(String[] args) {

MyThread myThread = new **MyThread**();

myThread.**start**();

myThread.**interrupt**();

System.out.**println**(myThread.**interrupted**()); *//false*

}

输出为false是因为这里的当前线程是main线程，main线程并没有中断。

那么我们让main线程调用interrupted方法输出结果

public static void **main**(String[] args) {

Thread.**currentThread**().**interrupt**();

System.out.**println**(Thread.**interrupted**()); *//ture*

System.out.**println**(Thread.**interrupted**()); *//false*

}

这么写是因为interrupted方法判断的是调用它的线程是否中断，在这里是判断main线程，所以要调用main线程的interrupted方法，用Thread.currentThread().interrupe()停止main线程。

当然如果要判断其他线程可以这样写：

public static void **main**(String[] args) {

MyThread myThread = new **MyThread**();

myThread.**start**();

myThread.**interrupt**();

System.out.**println**(myThread.**test**()); //true

System.out.**println**(myThread.**test**()); //false

}

class **MyThread** extends **Thread**{

private int i = 5;

@Override

public synchronized void **run**() {

return;

}

boolean **test**(){

return Thread.**interrupted**();

}

}

两次调用interrupted，第一次输出true，第二次输出false。因为interrupted方法会清除线程中断的状态。（并不是总是两次输出不一样，比如线程如果没有被中断，那么两次都会输出false）

和interrupted相比，isInterrupted方法是非static的，并且该方法不会清除线程中断的状态，比如下面的例子输出两个true。

public static void **main**(String[] args) {

MyThread myThread = new **MyThread**();

myThread.**start**();

myThread.**interrupt**();

System.out.**println**(myThread.**isInterrupted**()); *//true*

System.out.**println**(myThread.**isInterrupted**()); *//true*

}

# synchronized作用说明

关键字synchronized拥有锁重入的功能，也就是使用synchronized时，当一个线程得到一个对象锁后，再次请求此对象锁时是可以获得该对象的锁的。

synchronized void **f1**(){

**f2**();

}

synchronized void **f2**(){

return;

}

当一个线程A，调用了一个对象o的synchronized方法f时，A线程就获得了f方法所在**对象的锁（**synchronized修饰方法锁住的是对象的本身，也是this**）**，其他线程必须等待线程A执行完之后才可以调用f方法。如果另一个线程B调用了o中另一个synchronized方法ff时，由于线程获得的是对象的锁，所以B也要等待A线程执行完，释放对象锁之后才可以调用。（注意这里说的是synchronized方法）

class **SynchronizedObject**{

private int num = 5;

public synchronized void **addI**(String username) {

try {

if (username.**equals**("a")) {

num = 100;

System.out.**println**("set a");

Thread.**sleep**(200);

} else {

num = 200;

System.out.**println**("set b");

}

System.out.**println**(username + " " + num);

} catch (Exception e) {

e.**printStackTrace**();

}

}

}

class **MyThread1** extends **Thread**{

private SynchronizedObject object;

public **MyThread1**(SynchronizedObject object){

this.object = object;

}

@Override

public void **run**() {

object.**addI**("a");

}

}

class **MyThread2** extends **Thread**{

private SynchronizedObject object;

public **MyThread2**(SynchronizedObject object){

this.object = object;

}

@Override

public void **run**() {

object.**addI**("b");

}

}

上面这个例子，synchronized修饰的是addI方法，如果执行下面的main函数，可以看到输出结果是

set a

a 100

set b

b 200

因为synchronized的addI方法会得到对象的锁，只有一个线程执行完才可以执行其他的线程。

public static void **main**(String[] args) {

SynchronizedObject object = new **SynchronizedObject**();

MyThread1 thread1 = new **MyThread1**(object);

thread1.**start**();

MyThread2 thread2 = new **MyThread2**(object);

thread2.**start**();

return;

}

如果执行下面的例子，输出结果就不是严格有序的了

set a

set b

b 200

a 100

因为每个线程锁住的是不同的对象，所以不存在一个线程要等待另一个线程执行完。

public static void **main**(String[] args) {

SynchronizedObject object1 = new **SynchronizedObject**();

SynchronizedObject object2 = new **SynchronizedObject**();

MyThread1 thread1 = new **MyThread1**(object1);

thread1.**start**();

MyThread2 thread2 = new **MyThread2**(object2);

thread2.**start**();

return;

}

如果我们对synchronized的使用稍加修改，比如

public void **addI**(String username) {

synchronized (this) {

...

}

}

public void **addI**(String username) {

synchronized (SynchronizedObject.class) {

...;

}

}

static public void **addI**(String username){

...;

}

这三个效果又完全不一样了。对于第一个，效果和之前synchronized修饰方法是一样的，因为它也是锁住对象。而第二个锁住的并不是对象，而是Class，即使new多个**SynchronizedObject，**锁住的却是Class，所以一定能保证一个线程执行完毕，下一个线程才开始执行。静态的同步方法也就是对Class的锁。

synchronized修饰方法和修饰代码块的效果：

synchronized void **f**()

{

*//放在方法上面就锁住,整个方法里面所有的语句!*

*//一个线程进来了,其他线程就进不来,除非被wait,或者强制结束!*

}

void f()

{

synchronized(this)

{

*//如果放在这里,只能说,这个代码块里面的语句被锁!*

*//1线程进来在这里里面执行,2就进不来,会在这个代码块外面候着,除非1被挂起!*

*//但是method方法,所有线程都可以进来!*

}

}

不在synchronized块中的代码是异步执行的，在synchronized块中的就是同步执行的。

public void **f**(){

for(int i = 0; i < 10; i++)

out.**println**("i");

synchronized (this){

for(int i = 0; i < 10; i++)

out.**println**("i");

}

}

这个方法，如果有两个线程A和B同时执行，非synchronized部分的for循环，A和B可以交替的执行，但是synchronized部分的代码两个线程同步执行，一个线程执行完之后另一个线程才可以执行。

如果同步代码块使用synchronized(this)对对象进行锁定，一个类中多个方法都使用synchronized(this)，虽然可以实现同步，但是每个方法被会受到阻塞，影响程序效率。同步代码块可以不锁定this，那么只要synchronized代码块锁定的目标不同，这些代码块就可以异步执行。

public void **f**(){

String strLock = new **String**("");

synchronized (strLock){

...;

}

}

上面的代码，strLock每次都是一个新的对象，synchronized每次锁定的都是新的对象，所以这段代码始终无法同步执行。

## 避免对会在常量池中的对象加锁

String对象会存放在常量池中，常量池中的对象是相同的对象，对这类对象加锁无法达到想要的效果。

synchronized是对对象加锁，int等基本类型能当做对象来进行加锁，所以不用考虑。

## 锁对象改变

String lock = "123"

public void **f**(){

synchronized(lock){

lock = "456";

Thread.**sleep**(2000);

}

}

public static void **main**(String[] args){

Thread thread1 = new **Thread**();

Thread thread2 = new **Thread**();

a.**start**();

Thread.**sleep**(50);

b.**start**();

}

由于在synchronized同步块内部修改了lock对象，线程a和线程b锁定的不是同一个对象，所以thread1和thread2是异步的。

如果删除Thread.sleep(50)这句，thread1和thread2锁定的就都是"123"这个对象了，两个线程就是同步的。

对于锁对象的修改，如果只是修改了对象内部的一些属性，而对象本身没有变化（比如重新赋值导致地址的变化），那么认为锁定的是同一个对象。

# 线程间通信

## wait、notify

wait和notify、notifyAll这三个方法用于协调多个线程对共享数据的存取，所以**必须在synchronized语句块内使用**。synchronized关键字用于保护共享数据，阻止其他线程对共享数据的存取，但是这样程序的流程就很不灵活了，如何才能在当前线程还没退出synchronized数据块时让其他线程也有机会访问共享数据呢？此时就用这三个方法来灵活控制。

wait()方法使当前线程**暂停执行并释放对象锁**，让其他线程可以进入synchronized数据块，当前线程被放入对象等待池中。当调用notify()方法后，将从对象的等待池中移走一个任意的线程并放到锁标志等待池中，只有锁标志等待池中线程能够获取锁标志；如果锁标志等待池中没有线程，则notify()不起作用。notifyAll()则从对象等待池中移走所有等待那个对象的线程并放到锁标志等待池中。

执行notify()方法后，当前线程**不会马上释放该对象锁**（wait是立即就释放锁的），所以呈wait状态的线程并不能马上获取该对象锁，要等到执行notify()方法的线程将程序执行完，也就是推迟synchronized代码块之后，当前线程才会释放锁。

这三个方法属于Object的一部分，不属于Thread。wait()方法将线程挂起，而不是等待，只有在notify和notifyAll时才唤醒。

当线程调用wait方法后，如果线程调用了interrupe方法会抛出InterruptedException。

## 管道流 pipeStream

管道流分为：

1. PipedInputStream和PipedOutputStream；
2. PipedReader和PipedWrite。

# Lock

synchronized与wait、notify和notifyAll结合使用可以实现等待、通知模式，利用ReentrantLock和Condition也可以实现同样的功能。而且使用wait、notify模式时，被通知的线程是JVM随机选择的，会出现很大的效率问题。Condition类能提供更好的灵活性，可以实现多路通知功能，也就是在一个Lock对象中创建多个Condition实例，线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在线程调度上更加的灵活。

## demo

public class **ConditionService** {

private ReentrantLock lock = new **ReentrantLock**();

private Condition condition = lock.**newCondition**();

public void **toAwait**(){

try{

lock.**lock**();

System.out.**println**("await start: " + System.**currentTimeMillis**());

condition.**await**();

System.out.**println**("await end: " + System.**currentTimeMillis**());

} catch (InterruptedException e) {

e.**printStackTrace**();

} finally {

lock.**unlock**();

}

}

public void **toAwake**(){

try{

lock.**lock**();

System.out.**println**("awake start: " + System.**currentTimeMillis**());

condition.**signal**();

}finally {

lock.**unlock**();

}

}

}

public class **TestCondition** {

public static void **main**(String[] args) {

ConditionService service = new **ConditionService**();

MyThread myThread = new **MyThread**(service);

myThread.**start**();

try {

Thread.**sleep**(1000);

} catch (InterruptedException e) {

e.**printStackTrace**();

}

service.**toAwake**();

return;

}

}

class **MyThread** extends **Thread**{

private ConditionService service;

public **MyThread**(ConditionService service){

this.service = service;

}

@Override

public void **run**() {

service.**toAwait**();

}

}

/\*

await start: 1523591654019

awake start: 1523591655019

await end: 1523591655019

\*/

# 参考

## java多线程核心编程

## 理解java线程的中断(interrupt)

https://blog.csdn.net/canot/article/details/51087772