09 | Java线程（上）：Java线程的生命周期

2019-03-19 王宝令

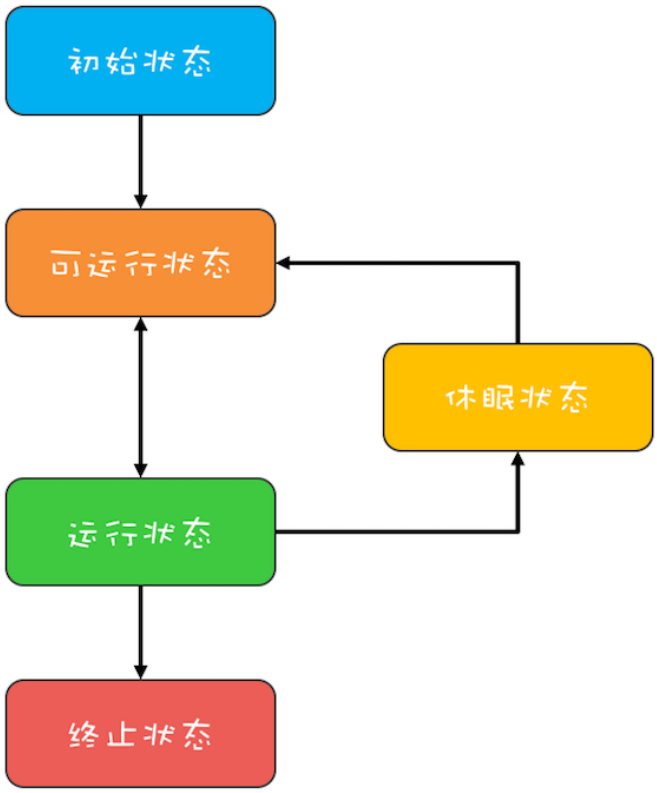
在Java领域，实现并发程序的主要手段就是多线程。线程是操作系统里的一个概念，虽然各种不同的开发语言如Java、C#等都对其进行了封装，但是万变不离操作系统。Java语言里的线程本质上就是操作系统的线程，它们是一一对应的。

在操作系统层面，线程也有“生老病死”，专业的说法叫有生命周期。对于有生命周期的事物，要学好它，思路非常简单，只要能搞懂**生命周期中各个节点的状态转换机制**就可以了。

虽然不同的开发语言对于操作系统线程进行了不同的封装，但是对于线程的生命周期这部分，基本上是雷同的。所以，我们可以先来了解一下通用的线程生命周期模型，这部分内容也适用于很多其他编程语言；然后再详细有针对性地学习一下Java中线程的生命周期。

通用的线程生命周期

通用的线程生命周期基本上可以用下图这个“五态模型”来描述。这五态分别是：**初始状态、可运行状态、运行状态、休眠状态**和**终止状态**。



通用线程状态转换图——五态模型

这“五态模型”的详细情况如下所示。

1. **初始状态**，指的是线程已经被创建，但是还不允许分配CPU执行。这个状态属于编程语言特有的，不过这里所谓的被创建，仅仅是在编程语言层面被创建，而在操作系统层面，真正的线程还没有创建。
2. **可运行状态**，指的是线程可以分配CPU执行。在这种状态下，真正的操作系统线程已经被成功创建了，所以可以分配CPU执行。
3. 当有空闲的CPU时，操作系统会将其分配给一个处于可运行状态的线程，被分配到CPU的线程的状态就转换成了**运行状态**。
4. 运行状态的线程如果调用一个阻塞的API（例如以阻塞方式读文件）或者等待某个事件（例如条件变量），那么线程的状态就会转换到**休眠状态**，同时释放CPU使用权，休眠状态的线程永远没有机会获得CPU使用权。当等待的事件出现了，线程就会从休眠状态转换到可运行状态。
5. 线程执行完或者出现异常就会进入**终止状态**，终止状态的线程不会切换到其他任何状态，进入终止状态也就意味着线程的生命周期结束了。

这五种状态在不同编程语言里会有简化合并。例如，C语言的POSIX Threads规范，就把初始状态和可运行状态合并了；Java语言里则把可运行状态和运行状态合并了，这两个状态在操作系统调度层面有用，而JVM层面不关心这两个状态，因为JVM把线程调度交给操作系统处理了。

除了简化合并，这五种状态也有可能被细化，比如，Java语言里就细化了休眠状态（这个下面我

们会详细讲解）。

Java中线程的生命周期

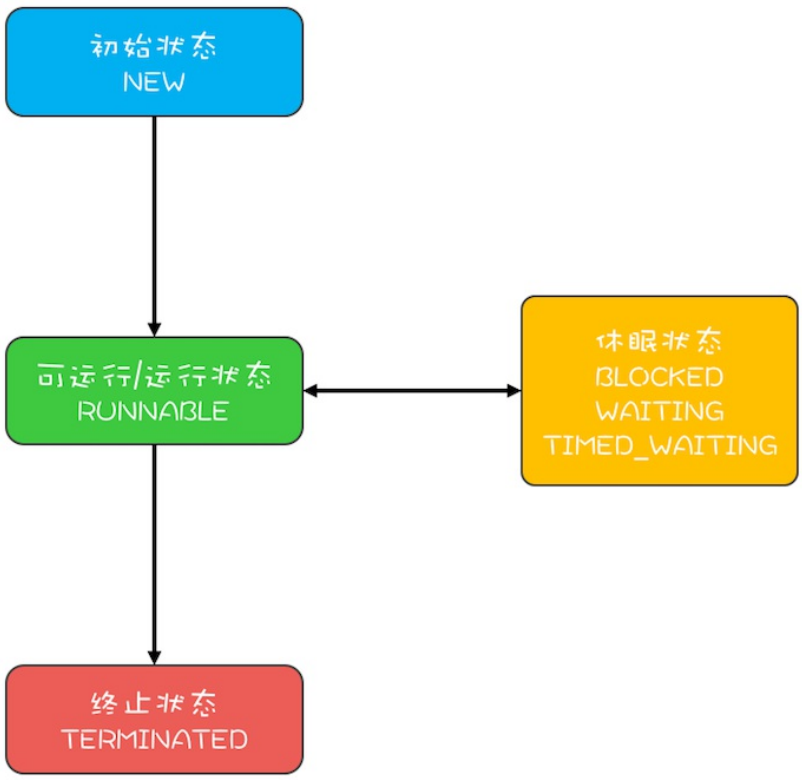
介绍完通用的线程生命周期模型，想必你已经对线程的“生老病死”有了一个大致的了解。那接下来我们就来详细看看Java语言里的线程生命周期是什么样的。

Java语言中线程共有六种状态，分别是：

1. NEW（初始化状态）
2. RUNNABLE（可运行/运行状态）
3. BLOCKED（阻塞状态）
4. WAITING（无时限等待）
5. TIMED_WAITING（有时限等待）
6. TERMINATED（终止状态）

这看上去挺复杂的，状态类型也比较多。但其实在操作系统层面，Java线程中的BLOCKED、WAITING、TIMED_WAITING是一种状态，即前面我们提到的休眠状态。也就是说**只要Java线程处于这三种状态之一，那么这个线程就永远没有CPU的使用权**。

所以Java线程的生命周期可以简化为下图：



Java中的线程状态转换图

其中，BLOCKED、WAITING、TIMED_WAITING可以理解为线程导致休眠状态的三种原因。那

具体是哪些情形会导致线程从RUNNABLE状态转换到这三种状态呢？而这三种状态又是何时转换回RUNNABLE的呢？以及NEW、TERMINATED和RUNNABLE状态是如何转换的？

1. RUNNABLE与BLOCKED的状态转换

只有一种场景会触发这种转换，就是线程等待synchronized的隐式锁。synchronized修饰的方法、代码块同一时刻只允许一个线程执行，其他线程只能等待，这种情况下，等待的线程就会从RUNNABLE转换到BLOCKED状态。而当等待的线程获得synchronized隐式锁时，就又会从BLOCKED转换到RUNNABLE状态。

如果你熟悉操作系统线程的生命周期的话，可能会有个疑问：线程调用阻塞式API时，是否会转换到BLOCKED状态呢？在操作系统层面，线程是会转换到休眠状态的，但是在JVM层面，Java线程的状态不会发生变化，也就是说Java线程的状态会依然保持RUNNABLE状态。**JVM层面并不关心操作系统调度相关的状态**，因为在JVM看来，等待CPU使用权（操作系统层面此时处于可执行状态）与等待I/O（操作系统层面此时处于休眠状态）没有区别，都是在等待某个资源，所以都归入了RUNNABLE状态。

而我们平时所谓的Java在调用阻塞式API时，线程会阻塞，指的是操作系统线程的状态，并不是Java线程的状态。

2. RUNNABLE与WAITING的状态转换

总体来说，有三种场景会触发这种转换。

第一种场景，获得synchronized隐式锁的线程，调用无参数的Object.wait()方法。其中，wait()方法我们在上一篇讲解管程的时候已经深入介绍过了，这里就不再赘述。

第二种场景，调用无参数的Thread.join()方法。其中的join()是一种线程同步方法，例如有一个线程对象thread A，当调用A.join()的时候，执行这条语句的线程会等待thread A执行完，而等待中的这个线程，其状态会从RUNNABLE转换到WAITING。当线程thread A执行完，原来等待它的线程又会从WAITING状态转换到RUNNABLE。

第三种场景，调用LockSupport.park()方法。其中的LockSupport对象，也许你有点陌生，其实Java并发包中的锁，都是基于它实现的。调用LockSupport.park()方法，当前线程会阻塞，线程的状态会从RUNNABLE转换到WAITING。调用LockSupport.unpark(Thread thread)可唤醒目标线程，目标线程的状态又会从WAITING状态转换到RUNNABLE。

3. RUNNABLE与TIMED_WAITING的状态转换

有五种场景会触发这种转换：

1. 调用**带超时参数**的Thread.sleep(long millis)方法；
2. 获得synchronized隐式锁的线程，调用**带超时参数**的Object.wait(long timeout)方法；
3. 调用**带超时参数**的Thread.join(long millis)方法；

4. 调用**带超时参数**的LockSupport.parkNanos(Object blocker, long deadline)方法；
5. 调用**带超时参数**的LockSupport.parkUntil(long deadline)方法。

这里你会发现TIMED_WAITING和WAITING状态的区别，仅仅是触发条件多了**超时参数**。

4. 从NEW到RUNNABLE状态

Java刚创建出来的Thread对象就是NEW状态，而创建Thread对象主要有两种方法。一种是继承Thread对象，重写run()方法。示例代码如下：

```
// 自定义线程对象
class MyThread extends Thread {
  public void run() {
    // 线程需要执行的代码
    ......
  }
}
// 创建线程对象
MyThread myThread = new MyThread();
```

另一种是实现Runnable接口，重写run()方法，并将该实现类作为创建Thread对象的参数。示例代码如下：

```
// 实现Runnable接口
class Runner implements Runnable {
  @Override
  public void run() {
    // 线程需要执行的代码
    ......
  }
}
// 创建线程对象
Thread thread = new Thread(new Runner());
```

NEW状态的线程，不会被操作系统调度，因此不会执行。Java线程要执行，就必须转换到RUNNABLE状态。从NEW状态转换到RUNNABLE状态很简单，只要调用线程对象的start()方法就可以了，示例代码如下：

```
MyThread myThread = new MyThread();
// 从NEW状态转换到RUNNABLE状态
myThread.start();
```

5. 从RUNNABLE到TERMINATED状态

线程执行完 run() 方法后，会自动转换到TERMINATED状态，当然如果执行run()方法的时候异常抛出，也会导致线程终止。有时候我们需要强制中断run()方法的执行，例如 run()方法访问一个很慢的网络，我们等不下去了，想终止怎么办呢？Java的Thread类里面倒是有个stop()方法，不过已经标记为@Deprecated，所以不建议使用了。正确的姿势其实是调用interrupt()方法。

那stop()和interrupt()方法的主要区别是什么呢？

stop()方法会真的杀死线程，不给线程喘息的机会，如果线程持有ReentrantLock锁，被stop()的线程并不会自动调用ReentrantLock的unlock()去释放锁，那其他线程就再也没机会获得ReentrantLock锁，这实在是太危险了。所以该方法就不建议使用了，类似的方法还有suspend()和 resume()方法，这两个方法同样也都不建议使用了，所以这里也就不多介绍了。

而interrupt()方法就温柔多了，interrupt()方法仅仅是通知线程，线程有机会执行一些后续操作，同时也可以无视这个通知。被interrupt的线程，是怎么收到通知的呢？一种是异常，另一种是主动检测。

当线程A处于WAITING、TIMED_WAITING状态时，如果其他线程调用线程A的interrupt()方法，会使线程A返回到RUNNABLE状态，同时线程A的代码会触发InterruptedException异常。上面我们提到转换到WAITING、TIMED_WAITING状态的触发条件，都是调用了类似wait()、join()、sleep()这样的方法，我们看这些方法的签名，发现都会throws InterruptedException这个异常。这个异常的触发条件就是：其他线程调用了该线程的interrupt()方法。

当线程A处于RUNNABLE状态时，并且阻塞在java.nio.channels.InterruptibleChannel上时，如果其他线程调用线程A的interrupt()方法，线程A会触发java.nio.channels.ClosedByInterruptException这个异常；而阻塞在java.nio.channels.Selector上时，如果其他线程调用线程A的interrupt()方法，线程A的java.nio.channels.Selector会立即返回。

上面这两种情况属于被中断的线程通过异常的方式获得了通知。还有一种是主动检测，如果线程处于RUNNABLE状态，并且没有阻塞在某个I/O操作上，例如中断计算圆周率的线程A，这时就得依赖线程A主动检测中断状态了。如果其他线程调用线程A的interrupt()方法，那么线程A可以通过isInterrupted()方法，检测是不是自己被中断了。

总结

理解Java线程的各种状态以及生命周期对于诊断多线程Bug非常有帮助，多线程程序很难调试，出了Bug基本上都是靠日志，靠线程dump来跟踪问题，分析线程dump的一个基本功就是分析线程状态，大部分的死锁、饥饿、活锁问题都需要跟踪分析线程的状态。同时，本文介绍的线程生命周期具备很强的通用性，对于学习其他语言的多线程编程也有很大的帮助。

你可以通过 jstack 命令或者Java VisualVM这个可视化工具将JVM所有的线程栈信息导出来，完整的线程栈信息不仅包括线程的当前状态、调用栈，还包括了锁的信息。例如，我曾经写过一个死锁的程序，导出的线程栈明确告诉我发生了死锁，并且将死锁线程的调用栈信息清晰地显示出来了（如下图）。导出线程栈，分析线程状态是诊断并发问题的一个重要工具。

```
Found one Java-level deadlock:
=============================
"T2":
  waiting to lock monitor 0x0000000002fcbac8 (object 0x000000076c4534a8, a org.i7.cp.lesson.one.Account),
  which is held by "T1"
"T1":
  waiting to lock monitor 0x0000000002fcbc28 (object 0x000000076c4534b8, a org.i7.cp.lesson.one.Account),
  which is held by "T2"

Java stack information for the threads listed above:
===================================================
"T2":
        at org.i7.cp.lesson.one.Account.transfer(Account.java:15)
        - waiting to lock <0x000000076c4534a8> (a org.i7.cp.lesson.one.Account)
        - locked <0x000000076c4534b8> (a org.i7.cp.lesson.one.Account)
        at org.i7.cp.lesson.one.Account.lambda$main$1(Account.java:31)
        at org.i7.cp.lesson.one.Account$$Lambda$2/519569038.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)
"T1":
        at org.i7.cp.lesson.one.Account.transfer(Account.java:15)
        - waiting to lock <0x000000076c4534b8> (a org.i7.cp.lesson.one.Account)
        - locked <0x000000076c4534a8> (a org.i7.cp.lesson.one.Account)
        at org.i7.cp.lesson.one.Account.lambda$main$0(Account.java:28)
        at org.i7.cp.lesson.one.Account$$Lambda$1/314337396.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

发生死锁的线程栈

课后思考

下面代码的本意是当前线程被中断之后，退出while(true)，你觉得这段代码是否正确呢？

```
Thread th = Thread.currentThread();
while(true) {
  if(th.isInterrupted()) {
    break;
  }
  // 省略业务代码无数
  try {
    Thread.sleep(100);
  }catch (InterruptedException e){
    e.printStackTrace();
  }
}
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

姜戈

81

可能出现无限循环，线程在sleep期间被打断了，抛出一个InterruptedException异常，try catch

捕捉此异常，应该重置一下中断标示，因为抛出异常后，中断标示会自动清除掉！

```
Thread th = Thread.currentThread();
while(true) {
if(th.isInterrupted()) {
break;
}
// 省略业务代码无数
try {
Thread.sleep(100);
}catch (InterruptedException e) {
Thread.currentThread().interrupt();
e.printStackTrace();
}
}
```

2019-03-19

作者回复

2019-03-19

Geek_961eed

34

希望作者讲解一下每一期的思考题！

2019-03-19

虎虎

24

我的一位长辈曾告诉我，没有真正学不会的知识或者技术，只是缺乏好的老师。

有的人可以把复杂的知识讲明白，但是讲解的过程却也是晦涩难懂，不免落了下成。

而学习王老师的课，我一直都觉得很轻松。云淡风轻地就把并发知识抽丝剥茧，确是更显功力。另一方面，我觉得人的大脑更喜欢接受这些平易近人的文字。看似浅近的文字，却更能带领我深入的思考，留下更深刻的印象。反观一些看起来高端大气上档次的论述，让人觉得云山雾罩，好不容易看懂了，但看过后却什么也想不起来了。大概是读文章的时候脑细胞都用来和晦涩的文字做斗争了，已经没有空间去思考和记忆了。

再次感谢王老师给大家带来优秀的课程。

2019-03-19

作者回复

看来我没必要写的很装了

2019-03-19

thas

10

interrupt是中断的意思，在单片机开发领域，用于接收特定的事件，从而执行后续的操作。Java线程中，（通常）使用interrupt作为线程退出的通知事件，告知线程可以结束了。
interrupt不会结束线程的运行，在抛出InterruptedException后会清除中断标志（代表可以接收下

一个中断信号了），所以我想，interrupt应该也是可以类似单片机一样作为一种通知信号的，只是实现通知的话，Java有其他更好的选择。
因InterruptedException退出同步代码块会释放当前线程持有的锁，所以相比外部强制stop是安全的（已手动测试）。sleep、join等会抛出InterruptedException的操作会立即抛出异常，wait在被唤醒之后才会抛出异常（就像阻塞一样，不被打扰）。
另外，感谢老师提醒，I/O阻塞在Java中是可运行状态，并发包中的lock是等待状态。

2019-03-19

作者回复

能和硬件中断联系起来

2019-03-19

悟 6

老师 stop方法直接杀掉线程了，什么不会释放锁呢

2019-03-19

作者回复

我也不知道搞jvm的人咋想的

2019-03-19

alias cd=rm -rf 5

思考题，不能中断循环，异常捕获要放在while循环外面

2019-03-19

作者回复

你这也是个办法

2019-03-19

Tristan 4

为什么实战高并发程序设计医术中写道“Tread.stop()方法在结束线程时，会直接终止线程，并且会释放这个线程所持有的锁”，而您文中所写的“果线程持有 synchronized 隐式锁，也不会释放”？？

2019-04-14

作者回复

是我的错，我确认了一下，隐式锁可以释放。多谢多谢！！！

2019-04-14

Junzi 4

当发起中断之后，Thread.sleep(100);会抛出InterruptedException异常，而这个抛出这个异常会清除当前线程的中断标识，导致th.isInterrupted()一直都是返回false的。

InterruptedException - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

2019-03-26

作者回复

2019-03-28

向往的生活 4

当线程 A 处于 WAITING、TIMED_WAITING 状态时，如果其他线程调用线程 A 的 interrupt() 方法，会使线程 A 返回到 RUNNABLE 状态，同时线程 A 的代码会触发 InterruptedException 异常。此时如果线程A获取不到锁，岂不是会立马又变成BLOCKED 状态？

2019-03-19

作者回复

我估计不会有中间的runnable，只是换个队列而已

2019-03-20

海鸿 3

如果线程处于阻塞状态（BLOCKED）,此时调用线程的中断方法，线程会又如何反应?
是否会像等待状态一样抛异常?
还是会像运行状态一样被标记为已中断状态?
还是不受到任何影响?
麻烦老师解答一下

2019-03-19

作者回复

阻塞态的线程不响应中断，并发包里的锁有方法能够响应中断

2019-03-20

刘晓林 3

感谢老师提醒，原来jvm层面的线程状态和os层面上的线程状态是不一样的，i/o挂起在jvm也是runable状态。另外并发包的lock其实是处于waitting状态。
但是有个疑问，jvm中blocked状态的线程和waitting状态的线程，除了处在不同的队列之外，还有没有什么区别呀？我这里问的区别包括jvm和os两个层面，谢谢老师

2019-03-19

作者回复

block不能响应中断，os里应该都是休眠状态，因为都不能获得cpu使用权

2019-03-20

缪文@有赞 2

```
if(th.isInterrupted()) {
break;
}
```

其实这段代码完全没必要啊，在捕获中断异常后，直接break就好了

2019-04-02

忠艾一生 2

这段代码中的线程对象并没有调用th.interrupt（）,只是调用了sleep()方法，此时线程并没有中断，也不会发生异常，sleep（）过后，线程继续自动执行。所以也不会进入到if代码块。
不知道我说的对不对啊老师。。。

2019-03-19

^_^

1

```
class MyThread extends Thread {
@Override
public void run() {
}
}
```

2019-03-28

ren

1

老师。那么jvm在进行gc的时候的停顿所有线程(stw) 这个期间 jvm中的线程应该属于生命周期的哪一个状态呢？ 我看到有资料讲的是 jvm中的线程 会因为jvm设置的安全点和安全区域 执行test指令产生一个自陷异常信号 这个指令应该是汇编中的触发线程中断的 那么之后的恢复成运行状态也都是交给操作系统层面来实现的吗？

2019-03-26

作者回复

线程调度是交给操作系统的，stw期间你看到的线程状态和stw之前应该是一样的，java里的线程状态是给你看的，没必要让你看到不该看的。但是stw期间操作系统层面的状态应该都是阻塞态，不允许调度。这个要看jvm的具体实现

2019-03-27

linqw

1

老师，不知道能否在理论讲解清楚的同时也能补上对源码的分析，比如线程a的interrupt方法被其他线程调用，有两种形式检测，异常和使用isInterrupted检测，但是内部原理还是感觉不清楚不明白，根据异常它是如何中断的？还有java有阻塞和等待状态，但是没能理解java为什么要将其区分开来，比如阻塞是在获取不到锁阻塞，会在锁对象中的队列排队，wait等待状态，不是也会在调用的对象队列中排队么？不太清楚为什么要怎么做？

2019-03-24

作者回复

我尽量不讲源码，讲源码的书有好多，感兴趣的可以去参考。也不回拿出汇编来讲解怎么实现的，网上也有很多。听完这个专栏去再去看代码，你会觉得很简单。

区分这么多状态的原因我也没有深究，可能是历史原因，如果并发包里的锁也搞一状态，可能会更乱

2019-03-25

Dylan

1

老师，Java调用阻塞API时，Java层面是runnable，那仍然占用CPU吗，此时此线程在操作系统中是什么状态呢？这个问题好几个人都在问，能详细解释下吗？

2019-03-24

作者回复

不占cpu，操作系统里是阻塞状态。

2019-03-24

靠人品去赢

1

（1）Interrupt()只是一个通知可以中断并不能真的去中断线程

（2）IsIntrrupt()方法是检查有没有中断

于是就会出现一种状况，线程没有中断，while就一直循环。关于try catch会不会重置中断标记，让其中断不了这个我就不了解，不知道具体原理（是异常错误那部分吗？）。

还有作者用Jstack命令和Java visualVM工具检查死锁真的是一个很棒的检查死锁的思路。

2019-03-21

Docker

1

测试过了，确实一个线程获取不到锁，线程状态为blocked

2019-03-20

Zach_

1

```
public class TestThread {

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

Worker t = new Worker();
t.start();

Thread.sleep(2000);

System.out.println("-1-1-1-");
t.interrupt();
System.out.println("000000");
Thread.sleep(2000);
t.stop();
System.out.println("000111");
Thread.sleep(2000);
t.join();
System.out.println("111111");
}

}

class Worker extends Thread {

@Override
public void run() {
int i = 0;
while (i<20) {
if (Thread.currentThread().isInterrupted()) {
```

```
break;
}
++i;
System.out.println(Thread.currentThread().getName() + "i: " + i);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
}
}
```

忽然发现极客时间网页版的留言窗口好小啊，都看不到自己上面写的东西...

---

1. 如果worker中没有sleep方法，则调用th.interrupt()方法会真正的中断th线程，并且不会抛出InterruptException 但是该演示代码不能体现锁的释放；
2. 如果worer中有sleep方法，则调用th.interrupt()方法会抛 java.lang.InterruptException(), 是针对sleep方法抛出的
同样的Object的wait() wait(带参) 也会抛出java.lang.InterruptException()而从当前的wait/blocked状态被中断（唤醒）
那也就是说，throws InterruptedException 的方法 在线程被调用interrupt()方法后，会被从当前状态中断
至于调用interrupy()方法后线程的状态属于哪种，取决于interrupt方法前的执行的方法使得当前线程处于哪种状态，
老师的总结很到位，需要好好理解，感受~！
3. 无论worder的run中有没有slee()方法，stop都会直接中断线程，当前演示代码也无法演示锁没有被释放
4. join()总是在等待被调用的线程执行完毕
5. while循环放在try里面, 在调用th.interrupt之后，可以有效捕获InterruptException 从而使th线程中断

说的有点多了， 大家多多讨论~！~！~！

2019-03-20

作者回复

好认真

2019-03-22