

## 09 | Java线程（上）：Java线程的生命周期

2019-03-19 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 14:00 大小 12.83M



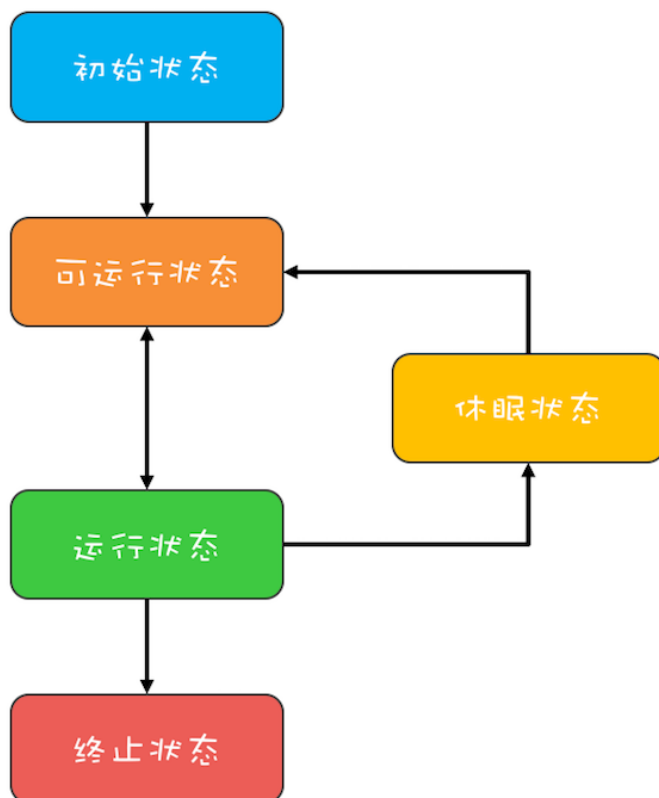
在 Java 领域，实现并发程序的主要手段就是多线程。线程是操作系统里的一个概念，虽然各种不同的开发语言如 Java、C# 等都对其进行了封装，但是万变不离操作系统。Java 语言里的线程本质上就是操作系统的线程，它们是一一对应的。

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

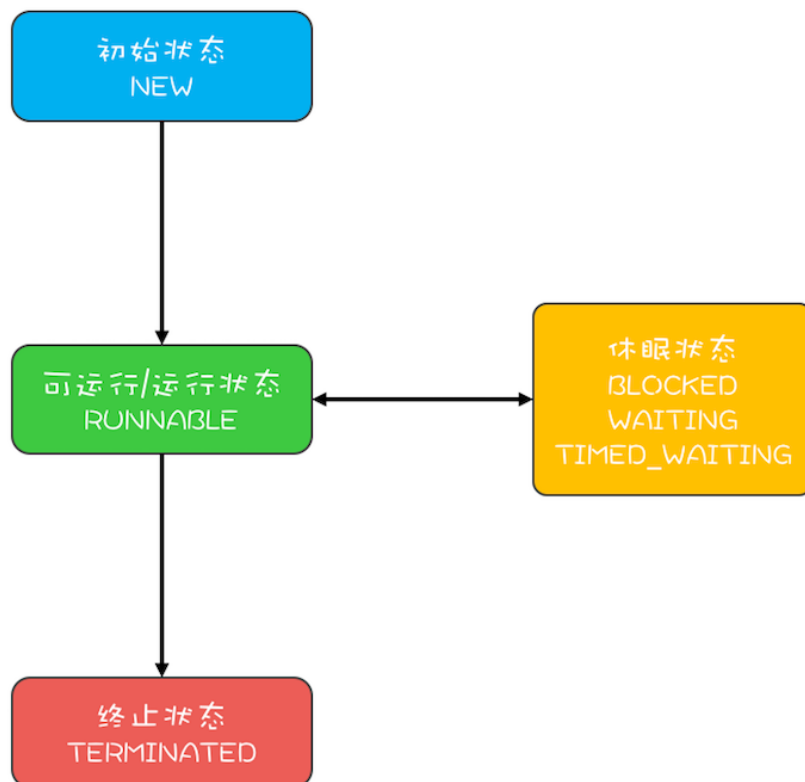
介绍完通用的线程生命周期模型，想必你已经对线程的“生老病死”有了一个大致的了解。那接下来我们就来详细看看 Java 语言里的线程生命周期是什么样的。

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：

 复制代码

```
1 // 自定义线程对象
2 class MyThread extends Thread {
3     public void run() {
4         // 线程需要执行的代码
5         .....
6     }
7 }
8 // 创建线程对象
9 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：

 复制代码

```
1 // 实现 Runnable 接口
2 class Runner implements Runnable {
3     @Override
4     public void run() {
5         // 线程需要执行的代码
6         .....
7     }
8 }
9 // 创建线程对象
10 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：

 复制代码

```
1 MyThread myThread = new MyThread();
2 // 从 NEW 状态转换到 RUNNABLE 状态
3 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有机会获得 `ReentrantLock` 锁，这实在是太危险了。所以该方法就不建议使用，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 0x0000000002fcbcb8 (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.
```



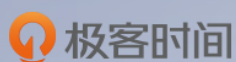
### 课后思考

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？

 复制代码

```
1 Thread th = Thread.currentThread();
2 while(true) {
3     if(th.isInterrupted()) {
4         break;
5     }
6     // 省略业务代码无数
7     try {
8         Thread.sleep(100);
9     }catch (InterruptedException e){
10         e.printStackTrace();
11     }
12 }
```

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。



# Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 08 | 管程：并发编程的万能钥匙

下一篇 10 | Java线程（中）：创建多少线程才是合适的？

## 精选留言 (53)

写留言



姜戈

2019-03-19

78

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) {...
```

展开

作者回复:



Geek\_961ee...

2019-03-19

33

希望作者讲解一下每一期的思考题！

展开



虎虎

2019-03-19

23

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...

展开

作者回复: 看来我没必要写的很装了



thas

2019-03-19

👍 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...  
展开 ∨

作者回复: 能和硬件中断联系起来👍👍👍



悟

2019-03-19

👍 6

老师 stop方法直接杀掉线程了，什么不会释放锁呢

展开 ∨

作者回复: 我也不知道搞jvm的人咋想的



alias cd=...

2019-03-19

👍 5

思考题，不能中断循环，异常捕获要放在while循环外面

展开 ∨

作者回复: 你这也是个办法



Tristan

2019-04-14

👍 4

为什么实战高并发程序设计术中写道“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”？？

展开 ∨

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!!!



向往的生活

2019-03-19

👍 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？

作者回复: 我估计不会有中间的runnable，只是换个队列而已



Junzi

2019-03-26

👍 3

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.

展开 ∨

作者回复: 🙏



海鸿

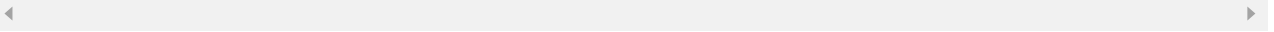
2019-03-19

👍 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 😊

展开 ▾

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



刘晓林

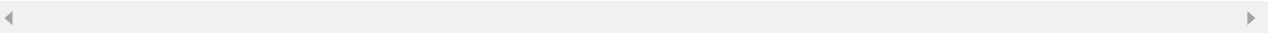
2019-03-19

👍 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

展开 ▾

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



缪文@有赞

2019-04-02

👍 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



忠艾一生

2019-03-19

👍 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。。。



^\_^

2019-03-28

👍 1

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

```
}  
}  
展开 ∨
```



ren

2019-03-26

👍 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？

展开 ∨

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

◀ ▶



linqw

2019-03-24

👍 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？

展开 ∨

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

◀ ▶



Dylan

2019-03-24

👍 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？

展开 ▾

作者回复: 不占cpu，操作系统里是阻塞状态。



**Docker**

2019-03-20

👍 1

测试过了，确实一个线程获取不到锁，线程状态为blocked

展开 ▾



**Zach\_**

2019-03-20

👍 1

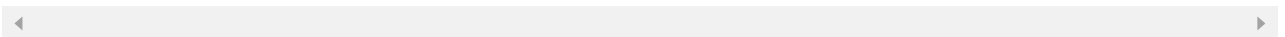
```
public class TestThread {
```

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

```
    ...
```

展开 ▾

作者回复: 好认真👍👍👍



**小华**

2019-03-19

👍 1

isInterrupted方法只是检测线程是否被标记为了中断状态，而不会改变线程的中断状态，代码中，没有调用线程的interrupted方法，所以线程的中断状态为false，while循环不会退出，也不会抛InterruptedException

展开 ▾