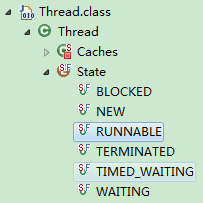
<https://mp.weixin.qq.com/s?__biz=MzIyNDU2ODA4OQ==&mid=2247484002&idx=1&sn=75809357f0ecc639cd1a672f97b9664f&chksm=e80db414df7a3d024d94991064191cca0063eefc94847009565c40598c4e20870dc2b053e40c&scene=21#wechat_redirect>

# 为什么Java线程没有Running状态？

Java虚拟机层面所暴露给我们的状态，与操作系统底层的线程状态是两个不同层面的事。具体而言，这里说的 Java 线程状态均来自于 Thread 类下的 State 这一内部枚举类中所定义的状态：

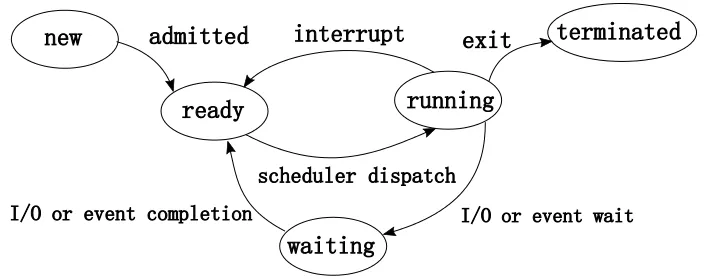


### 什么是 RUNNABLE？

直接看它的 Javadoc 中的说明：

一个在 JVM 中**执行**的线程处于这一状态中。（A thread **executing** in the Java virtual machine is in this state.）

而传统的进（线）程状态一般划分如下：



注：这里的进程指早期的**单线程**进程，这里所谓进程状态实质就是线程状态。

那么 runnable 与图中的 ready 与 running 区别在哪呢？

### 与传统的ready状态的区别

更具体点，javadoc 中是这样说的：

处于 runnable 状态下的线程正在 Java 虚拟机中执行，但它**可能正在等待**来自于操作系统的其它资源，比如处理器。

A thread in the runnable state is executing in the Java virtual machine but **it may be waiting for** other resources from the operating system such as processor.

显然，runnable 状态实质上是包括了 ready 状态的。

甚至还可能有包括上图中的 waiting 状态的部分细分状态，在后面我们将会看到这一点。

### 与传统的running状态的区别

有人常觉得 Java 线程状态中还少了个 running 状态，这其实是把两个不同层面的状态混淆了。对 Java 线程状态而言，不存在所谓的running 状态，它的 runnable 状态包含了 running 状态。

我们可能会问，为何 JVM 中没有去区分这两种状态呢？

现在的**时分**（time-sharing）**多任务**（multi-task）操作系统架构通常都是用所谓的“**时间分片**（time quantum or time slice）”方式进行**抢占式**（preemptive）轮转调度（round-robin式）。

更复杂的可能还会加入优先级（priority）的机制。

这个时间分片通常是很小的，一个线程一次最多只能在 cpu 上运行比如10-20ms 的时间（此时处于 running 状态），也即大概只有0.01秒这一量级，时间片用后就要被切换下来放入调度队列的末尾等待再次调度。（也即回到 ready 状态）

注：如果期间进行了 I/O 的操作还会导致提前释放时间分片，并进入等待队列。

又或者是时间分片没有用完就被抢占，这时也是回到 ready 状态。

这一切换的过程称为线程的**上下文切换**（context switch），当然 cpu 不是简单地把线程踢开就完了，还需要把被相应的执行状态保存到内存中以便后续的恢复执行。

显然，10-20ms 对人而言是很快的，

不计切换开销（每次在1ms 以内），相当于1秒内有50-100次切换。事实上时间片经常没用完，线程就因为各种原因被中断，实际发生的切换次数还会更多。

也这正是**单核 \*CPU 上实现所谓的“**并发\*（concurrent）”的基本原理，但其实是快速切换所带来的假象，这有点类似一个手脚非常快的杂耍演员可以让好多个球同时在空中运转那般。

时间分片也是可配置的，如果不追求在多个线程间很快的响应，也可以把这个时间配置得大一点，以减少切换带来的开销。

如果是多核CPU，才有可能实现真正意义上的并发，这种情况通常也叫**并行**（pararell），不过你可能也会看到这两词会被混着用，这里就不去纠结它们的区别了。

通常，Java的线程状态是服务于监控的，如果线程切换得是如此之快，那么区分 ready 与 running 就没什么太大意义了。

当你看到监控上显示是 running 时，对应的线程可能早就被切换下去了，甚至又再次地切换了上来，也许你只能看到 ready 与 running 两个状态在快速地闪烁。

当然，对于精确的性能评估而言，获得准确的 running 时间是有必要的。

现今主流的 JVM 实现都把 Java 线程一一映射到操作系统底层的线程上，把调度委托给了操作系统，我们在虚拟机层面看到的状态实质是对底层状态的映射及包装。JVM 本身没有做什么实质的调度，把底层的 ready 及 running 状态映射上来也没多大意义，因此，统一成为runnable 状态是不错的选择。

我们将看到，Java 线程状态的改变通常只与自身显式引入的机制有关。

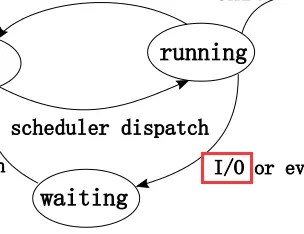
### 当I/O阻塞时

我们知道传统的I/O都是阻塞式（blocked）的，原因是I/O操作比起cpu来实在是太慢了，可能差到好几个数量级都说不定。如果让 cpu 去等I/O 的操作，很可能时间片都用完了，I/O 操作还没完成呢，不管怎样，它会导致 cpu 的利用率极低。

所以，解决办法就是：一旦线程中执行到 I/O 有关的代码，相应线程立马被切走，然后调度 ready 队列中另一个线程来运行。

这时执行了 I/O 的线程就不再运行，即所谓的被阻塞了。它也不会被放到调度队列中去，因为很可能再次调度到它时，I/O 可能仍没有完成。

线程会被放到所谓的等待队列中，处于上图中的 waiting 状态：



当然了，我们所谓阻塞只是指这段时间 cpu 暂时不会理它了，但另一个部件比如硬盘则在努力地为它服务。cpu 与硬盘间是并发的。如果把线程视作为一个 job，这一 job 由 cpu 与硬盘交替协作完成，当在 cpu 上是 waiting 时，在硬盘上却处于 running，只是我们在操作系统层面讨论线程状态时通常是围绕着 cpu 这一中心去述说的。

而当 I/O 完成时，则用一种叫**中断**（interrupt）的机制来通知 cpu：

也即所谓的“**中断驱动**（interrupt-driven）”，现代操作系统基本都采用这一机制。

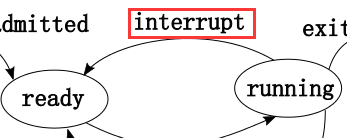
某种意义上，这也是**控制反转**（IoC）机制的一种体现，cpu不用反复去询问硬盘，这也是所谓的“好莱坞原则”—Don’t call us, we will call you.好莱坞的经纪人经常对演员们说：“别打电话给我，（有戏时）我们会打电话给你。”

在这里，硬盘与 cpu 的互动机制也是类似，硬盘对 cpu 说：”别老来问我 IO 做完了没有，完了我自然会通知你的“

当然了，cpu 还是要不断地检查中断，就好比演员们也要时刻注意接听电话，不过这总好过不断主动去询问，毕竟绝大多数的询问都将是徒劳的。

cpu 会收到一个比如说来自硬盘的中断信号，并进入中断处理例程，手头正在执行的线程因此被打断，回到 ready 队列。而先前因 I/O 而waiting 的线程随着 I/O 的完成也再次回到 ready 队列，这时 cpu 可能会选择它来执行。

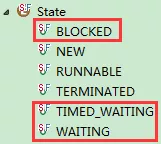
另一方面，所谓的时间分片轮转本质上也是由一个定时器定时中断来驱动的，可以使线程从 running 回到 ready 状态：



比如设置一个10ms 的倒计时，时间一到就发一个中断，好像大限已到一样，然后重置倒计时，如此循环。

与 cpu 正打得火热的线程可能不情愿听到这一中断信号，因为它意味着这一次与 cpu 缠绵的时间又要到头了......奴为出来难，何日君再来？

现在我们再看一下 Java 中定义的线程状态，嘿，它也有 BLOCKED（阻塞），也有 WAITING（等待），甚至它还更细，还有TIMED\_WAITING：

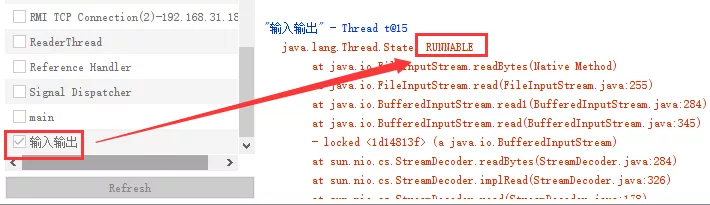


现在问题来了，进行阻塞式 I/O 操作时，Java 的线程状态究竟是什么？是 BLOCKED？还是 WAITING？

可能你已经猜到，既然放到 RUNNABLE 这一主题下讨论，其实状态还是 RUNNABLE。我们也可以通过一些测试来验证这一点：

@Test  
**public** **void** **testInBlockedIOState**() throws InterruptedException {  
    Scanner **in** = **new** Scanner(System.**in**);  
    *// 创建一个名为“输入输出”的线程t*  
    Thread t = **new** Thread(**new** Runnable() {  
        @Override  
        **public** **void** **run**() {  
            **try** {  
                *// 命令行中的阻塞读*  
                String input = **in**.nextLine();  
                System.**out**.println(input);  
            } **catch** (Exception e) {  
                e.printStackTrace();  
            } **finally** {  
              IOUtils.closeQuietly(**in**);  
            }  
        }  
    }, "输入输出"); *// 线程的名字*  
  
    *// 启动*  
    t.start();  
  
    *// 确保run已经得到执行*  
    Thread.sleep(100);  
  
    *// 状态为RUNNABLE*  
    assertThat(t.getState()).isEqualTo(Thread.State.RUNNABLE);  
}

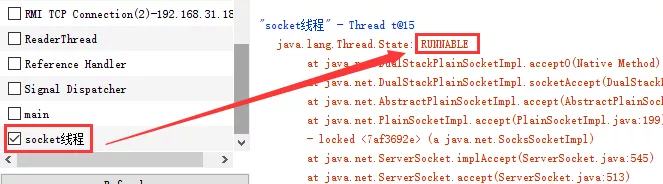
在最后的语句上加一断点，监控上也反映了这一点：



网络阻塞时同理，比如socket.accept，我们说这是一个“阻塞式(blocked)”式方法，但线程状态还是 RUNNABLE。



监控显示：

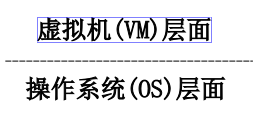


当然，Java 很早就引入了所谓 nio（新的IO）包，至于用 nio 时线程状态究竟是怎样的，这里就不再一一具体去分析了。

至少我们看到了，进行传统上的 IO 操作时，口语上我们也会说“阻塞”，但这个“阻塞”与线程的 BLOCKED 状态是两码事！

### 如何看待RUNNABLE状态？

首先还是前面说的，注意分清两个层面：



虚拟机是骑在你操作系统上面的，身下的操作系统是作为某种资源为满足虚拟机的需求而存在的：



当进行阻塞式的 IO 操作时，或许底层的操作系统线程确实处在阻塞状态，但我们关心的是 JVM 的线程状态。

JVM 并不关心底层的实现细节，什么时间分片也好，什么 IO 时就要切换也好，它并不关心。

前面说到，“处于 runnable 状态下的线程正在\* Java 虚拟机**中执行，但它**可能正在等待\*来自于操作系统的其它资源，比如处理器。”

JVM 把那些都视作资源，cpu 也好，硬盘，网卡也罢，有东西在为线程服务，它就认为线程在“执行”。

你用嘴，用手，还是用什么鸟东西来满足它的需求，它并不关心~

处于 IO 阻塞，只是说 cpu 不执行线程了，但网卡可能还在监听呀，虽然可能暂时没有收到数据：

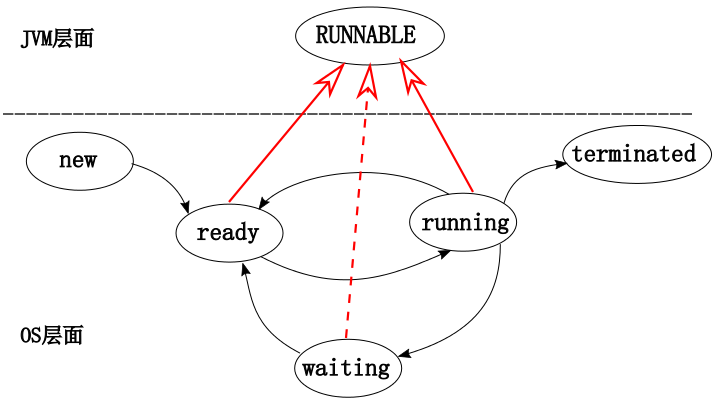
就好比前台或保安坐在他们的位置上，可能没有接待什么人，但你能说他们没在工作吗？

所以 JVM 认为线程还在执行。而操作系统的线程状态是围绕着 cpu 这一核心去述说的，这与 JVM 的侧重点是有所不同的。

前面我们也强调了“Java 线程状态的改变通常只与自身显式引入的机制有关”，如果 JVM 中的线程状态发生改变了，通常是自身机制引发的。

比如 synchronize 机制有可能让线程进入BLOCKED 状态，sleep，wait等方法则可能让其进入 WATING 之类的状态。

它与传统的线程状态的对应可以如下来看：



RUNNABLE 状态对应了传统的 ready， running 以及部分的 waiting 状态。