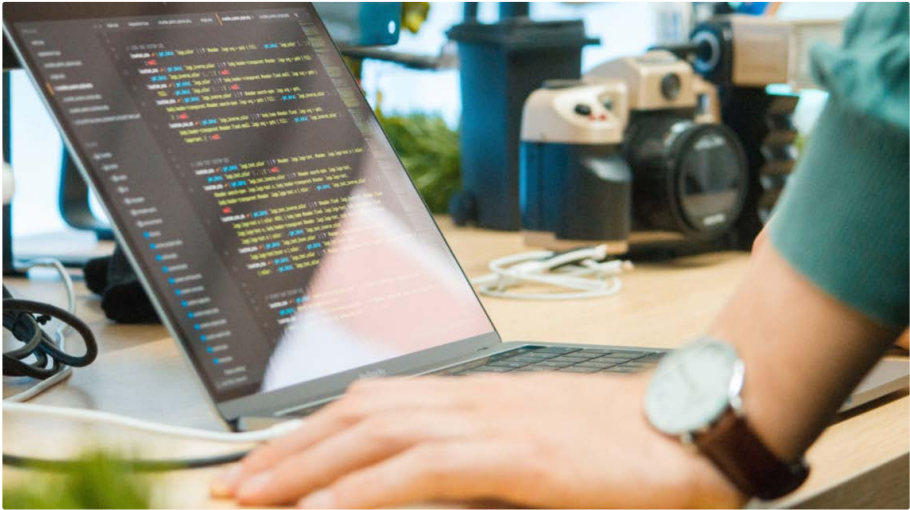


第17讲 | 一个线程两次调用start()方法会出现什么情况？

2018-06-14 杨晓峰





第17讲 | 一个线程两次调用start()方法会出现什么情况？

杨晓峰

- 00:18 / 10:01

今天我们来深入聊聊线程，相信大家对于线程这个概念都不陌生，它是Java并发的基础元素，理解、操纵、诊断线程是Java工程师的必修课，但是你真的掌握线程了吗？

今天我要问你的问题是，[一个线程两次调用start\(\)方法会出现什么情况？](#) [谈谈线程的生命周期和状态转移。](#)

典型回答

Java的线程是不允许启动两次的，第二次调用必然会抛出IllegalThreadStateException，这是一种运行时异常，多次调用start被认为是编程错误。

关于线程生命周期的不同状态，在Java 5以后，线程状态被明确定义在其公共内部枚举类型java.lang.Thread.State中，分别是：

- 新建（NEW），表示线程被创建出来还没真正启动的状态，可以认为它是Java内部状态。
- 就绪（RUNNABLE），表示该线程已经在JVM中执行，当然由于执行需要计算资源，它可能是正在运行，也可能还在等待系统分配给它CPU片段，在就绪队列里面排队。
- 在其他一些分析中，会额外区分一种状态RUNNING，但是从Java API的角度，并不能表示出来。
- 阻塞（BLOCKED），这个状态和我们前面两讲介绍的同步非常相关，阻塞表示线程在等待Monitor lock，比如，线程试图通过synchronized去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态。
- 等待（WAITING），表示正在等待其他线程采取某些操作。一个常见的场景是类似生产者消费者模式，发现任务条件尚未满足，就让当前消费者线程等待（wait），另外的生产者线程去准备任务数据，然后通过类似notify等动作，通知消费线程可以继续工作了。Thread.join()也会令线程进入等待状态。
- 计时等待（TIMED\_WAIT），其进入条件和等待状态类似，但是调用的是存在超时条件的方法，比如wait或join等方法的指定超时版本，如下面示例：

```
public final native void wait(long timeout) throws InterruptedException;
```

- 终止（TERMINATED），不管是意外退出还是正常执行结束，线程已经完成使命，终止运行，也有人把这个状态叫作死亡。

在第二次调用start()方法的时候，线程可能处于终止或者其他（非NEW）状态，但是不论如何，都是不可以再次启动的。

考点分析

今天的问题可以算是个常见的面试热身题目，前面的给出的典型回答，算是对基本状态和简单流转的一个介绍，如果觉得还不够直观，我在下面分析会对比一个状态图进行介绍。总的来说，理解线程对于我们日常开发或者诊断分析，都是不可或缺的基础。

面试官可能会以此为契机，从各种不同角度考察你对线程的掌握：

- 相对理论一些的面试官可能会问你线程到底是什么以及Java底层实现方式。
- 线程状态的切换，以及和锁等并发工具类的互动。
- 线程编程时容易踩的坑与建议等。

可以看出，仅仅是一个线程，就有非常多的内容需要掌握。我们选择重点内容，开始进入详细分析。

知识扩展

首先，我们来整体看一下线程是什么？

从操作系统的角度，可以简单认为，线程是系统调度的最小单元，一个进程可以包含多个线程，作为任务的真正运作者，有自己的栈（Stack）、寄存器（Register）、本地存储（Thread Local）等，但是会和进程内其他线程共享文件描述符、虚拟地址空间等。

在具体实现中，线程还分为内核线程、用户线程，Java的线程实现其实是与虚拟机相关的。对于我们最熟悉的Sun/Oracle JDK，其线程也经历了一个演进过程，基本上在Java 1.2之后，JDK已经抛弃了所谓的[Green Thread](#)，也就是用户调度的线程，现在的模型是一对一映射到操作系统内核线程。

如果我们来看Thread的源码，你会发现其基本操作逻辑大都是以JNI形式调用的本地代码。

```
private native void start0();
private native void setPriority0(int newPriority);
private native void interrupt0();
```

这种实现有利有弊。总体来说，Java语言得益于精细粒度的线程和相关的并发操作，其构建高扩展性的大型应用的能力已经毋庸置疑。但是，其复杂性也提高了并发编程的门槛，近几年的Go语言等提供了协程（[coroutine](#)），大大提高了构建并发应用的效率。于此同时，Java也在[Loom](#)项目中，孕育新的类似轻量级用户线程（Fiber）等机制，也许在不久的将来就可以在新版JDK中使用到它。

下面，我来分析下线程的基本操作。如何创建线程想必你已经非常熟悉了，请看下面的例子：

```
Runnable task = () -> {System.out.println("Hello World!");};
Thread myThread = new Thread(task);
myThread.start();
myThread.join();
```

我们可以直接扩展Thread类，然后实例化。但在本例中，我选取了另外一种方式，就是实现一个Runnable，将代码逻辑放在Runnable中，然后构建Thread并启动（start），等待结束（join）。

Runnable的好处是，不会受Java不支持类多继承的限制，重用代码实现，当我们需要重复执行相应逻辑时优点明显。而且，也能更好的与现代Java并发库中的Executor之类框架结合使用，比如将上面start和join的逻辑完全写成下面的结构：

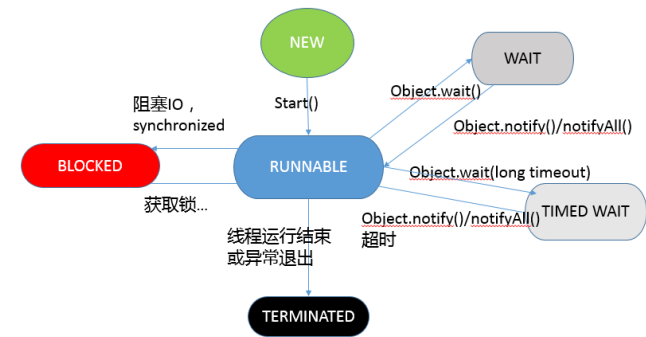
```
Future future = Executors.newFixedThreadPool(1)
    .submit(task)
    .get();
```

这样我们就不用操心线程的创建和管理，也能利用Future等机制更好地处理执行结果。线程生命周期通常和业务之间没有本质联系，混淆实现需求和业务需求，就会降低开发的效率。

从线程生命周期的状态开始展开，那么在Java编程中，有哪些因素可能影响线程的状态呢？主要有：

- 线程自身的方法，除了start，还有多个join方法，等待线程结束；yield是告诉调度器，主动让出CPU；另外，就是一些已经被标记为过时的resume、stop、suspend之类，据我所知，在JDK最新版本中，destory/stop方法将被直接移除。
- 基类Object提供了一些基础的wait/notify/notifyAll方法。如果我们持有某个对象的Monitor锁，调用wait会让当前线程处于等待状态，直到其他线程notify或者notifyAll。所以，本质上是提供了Monitor的获取和释放的能力，是基本的线程间通信方式。
- 并发类库中的工具，比如CountDownLatch.await()会让当前线程进入等待状态，直到latch被基数为0，这可以看作是线程间通信的Signal。

我这里画了一个状态和方法之间的对应图：



Thread和Object的方法，听起来简单，但是实际应用中被证明非常晦涩、易错，这也是为什么Java后来又引入了并发包。总的来说，有了并发包，大多数情况下，我们已经不再需要去调用wait/notify之类的方法了。

前面谈了不少理论，下面谈谈线程API使用，我会侧重于平时工作学习中，容易被忽略的一些方面。

先来看看守护线程（Daemon Thread），有的时候应用中需要一个长期驻留的服务程序，但是不希望其影响应用退出，就可以将其设置为守护线程，如果JVM发现只有守护线程存在时，将结束进程。具体可以参考下面代码段。注意，必须在线程启动之前设置。

```
Thread daemonThread = new Thread();
daemonThread.setDaemon(true);
daemonThread.start();
```

再来看看[Spurious wakeup](#)。尤其是在多核CPU的系统中，线程等待存在一种可能，就是在没有任何线程广播或者发出信号的情况下，线程就被唤醒，如果处理不当就可能出现诡异的并发问题，所以我们在等待条件过程中，建议采用下面模式来书写。

```
// 推荐
while ( isCondition() ) {
    waitforkConftion(...);
}

// 不推荐, 可能引入bug
if ( isCondition() ) {
    waitforkConftion(...);
}
```

Thread.onSpinWait()，这是Java 9中引入的特性。我在[专栏第16讲](#)给你留的思考题中，提到“自旋锁”（spin-wait, busy-waiting），也可以认为其不算是一种锁，而是一种针对短期等待的性能优化技术。“onSpinWait()”没有任何行为上的保证，而是对JVM的一个暗示，JVM可能会利用CPU的pause指令进一步提高性能，性能特别敏感的应用可以关注。

再有就是慎用[ThreadLocal](#)，这是Java提供了一种保存线程私有信息的机制，因为其在整个线程生命周期内有效，所以可以方便地在一个线程关联的不同业务模块之间传递信息，比如事务ID、Cookie等上下文相关信息。

它的实现结构，可以参考[源码](#)，数据存储于线程相关的ThreadLocalMap，其内部条目是弱引用，如下面片段。

```
static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;
        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
    // ...
}
```

当Key为null时，该条目就变成“废弃条目”，相关“value”的回收，往往依赖于几个关键点，即set、remove、rehash。

下面是set的示例，我进行了精简和注释：

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);

    for (Entry e = tab[i];; ...) {
        //...
        if (k == null) {
            // 替换废弃条目
            replaceStaleEntry(key, value, i);
            return;
        }

        tab[i] = new Entry(key, value);
        int sz = ++size;
        // 扫描并清理发现的废弃条目，并检查容量是否超限
        if (!cleanSomeSlots(i, sz) && sz >= threshold)
            rehash(); // 清理废弃条目，如果仍然超限，则扩容（加倍）
    }
}
```

具体的清理逻辑是实现在cleanSomeSlots和expungeStaleEntry之中，如果你有兴趣可以自行阅读。

结合[专栏第4讲](#)介绍的引用类型，我们会发现一个特别的地方，通常弱引用都会和引用队列配合清理机制使用，但是ThreadLocal是个例外，它并没有这么做。

这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应ThreadLocalMap！这就是很多OOM的来源，所以通常会建议，应用一定要自己负责remove，并且不要和线程池配合，因为worker线程往往是不会退出的。

今天，我介绍了线程基础，分析了生命周期中的状态和各种方法之间的对应关系，这也有助于我们更好地理解synchronized和锁的影响，并介绍了一些需要注意的操作，希望对你有所帮助。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天我准备了一个有意思的问题，写一个最简单的打印HelloWorld的程序，说说看，运行这个应用，Java至少会创建几个线程呢？然后思考一下，如何明确验证你的结论，真实情况很可能令你大跌眼镜哦。

请在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



风动静泉	2018-06-14
<p>一课一练：</p> <p>使用了两种方式获取当前程序的线程数。</p> <p>1、使用线程管理器MXBean</p> <p>2、直接通过线程组的activeCount</p> <p>第二种需要注意不断向上找父线程组，否则只能获取当前线程组，结果是1</p> <p>结论：</p> <p>使用以上两种方式获取的线程总数都是5个。</p> <p>main</p> <p>Attach Listener</p> <p>Signal Dispatcher</p> <p>Finalizer</p> <p>Reference Handler</p> <p>此外，如果使用的IDE是IDEA 直接运行会多一个Monitor Ctrl-break线程，这个是IDE的原因。debug模式下不会有这个线程。</p> <p>作者回复</p>	2018-06-14
不错	2018-06-14
qpm	2018-06-14
<p>做了一个test老师的问题，观察到的情况如下：</p> <p>JVM 启动 Hello World的线程分析</p> <p>环境：</p> <p>macOS + jdk8</p> <p>检测获得</p> <p>Thread[Reference Handler,10,system]</p> <p>Thread[Finalizer,8,system]</p> <p>Thread[main,5,main]</p> <p>Thread[Signal Dispatcher,9,system]</p> <p>Hello World!</p> <p>其中：</p> <p>Reference Handler：处理引用对象本身的垃圾回收</p> <p>Finalizer：处理用户的Finalizer方法</p> <p>Signal Dispatcher：外部jvm命令的转发器</p> <p>在jdk6环境中</p> <p>还有一个Attach Listener的线程</p> <p>是负责接收外部命令的，如jmap、jstack</p> <p>作者回复</p>	2018-06-14
不错	2018-06-14
行者	2018-06-14
<p>“我们会发现一个特别的地方，通常幻象引用都会和引用队列配合清理机制使用，但是 ThreadLocal 是个例外，它并没有这么做。”</p> <p>老师，Entry继承的是WeakReference，这个是弱引用吧。</p> <p>main:</p> <pre>System.out.println("hello world"); ThreadGroup group = Thread.currentThread().getThreadGroup(); ThreadGroup topGroup = group; while (group != null) {     topGroup = group;     group = group.getParent(); } int nowThreads = topGroup.activeCount(); Thread[] lstThreads = new Thread[nowThreads]; topGroup.enumerate(lstThreads); for (int i = 0; i &lt; nowThreads; i++) {     System.out.println("线程number: " + i + " = " + lstThreads[i].getName()); } out: 线程number: 0 = Reference Handler // 计算对象是否可达? 线程number: 1 = Finalizer // 回收对象时触发的finalize方法? 线程number: 2 = Signal Dispatcher // 线程调度员 线程number: 3 = main 线程number: 4 = Monitor Ctrl-Break // 监控器，锁相关</pre> <p>作者回复</p>	2018-06-14

前面是翻译单了，已经修正；后面大家用了很多方法，基本都可以。主要目的是结合前面的介绍加深理解	2018-06-14
爱折腾的老斑鸠	
theadlocal里面的值如果是线程池的线程里面设置的，当任务完成，线程归还线程池时，这个theadlocal里面的值是不是不会被回收？	2018-06-14
作者回复	
嗯，线程池一般不建议和thread local配合...	2018-06-14
三木子	
现在觉得踩坑是一种很好学习方法	2018-06-15
作者回复	
同意	2018-06-17
tyson	
1、站在应用程序方面，只创建了一个线程。 2、站在jvm方面，肯定还有gc等其余线程。	2018-06-14
总结： 1、线程是系统调度的最小单元，应该是进程吧。线程是操作系统的资源，在运行的时候会打开文件描述符等。 2、resume、stop、suspend等已经被废弃了 3、线程的等待和唤醒，建议使用reentrantlock的condition wait/notify方法 4、可以使用线程的join方法、countdownlatch、cyclicbarrier、future等进行线程的等待	
作者回复	2018-06-14
不错	
锐	
通常弱引用都会和引用队列配合清理机制使用，但是 ThreadLocal 是个例外，它并没有这么做。	2018-06-14
这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应 ThreadLocalMap！这就是很多 OOM 的来源	
这个平时还真没注意	
作者回复	2018-06-14
嗯，为了生命周期的需求	
sunlight001	
threadlocal在放入值之后，在get出来之后，需要做remove操作，我这么理解对么？以前写的程序都没remove◆◆	2018-06-14
作者回复	2018-06-14
不用了，明确移除是好习惯	
Eason	
“比如，线程试图通过 synchronized 去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态”这个例子换一个理解，感觉也是在等待其他线程做某些操作。在“阻塞”中也是在“等待”中？	2018-06-14
作者回复	
wait和blocked是不同的	2018-06-14
黄启航	
杨老师您好，我有个疑问：	2018-07-09
文章最后说“弱引用都会和引用队列配合清理工作，但是Threadlocal是个例外，它并没有这么做。这意味着，废弃项目的回收依赖显式地触发，否则就要等待线程的结束”。	
我的疑问：既然没有利用引用队列来实现自动清除，那TheadLocalMap内部的Entry继承WeakReference有何用意？能起到什么作用？	
tracer	
有讲解那五个线程的资料吗？	2018-06-27
作者回复	
不知道，源码...	2018-06-28
jacy	
Threadlocal进行线程隔离，线程拥有自己的数据空间，synchronize进行线程同步。 另外想问老师，虚假唤醒的深层次原因是啥呢？	2018-06-20
TonyEasy	
老师，我有一点疑问，在线程池复用线程时，对同一线程调用多次.start()方法，为何不报错呢？	2018-06-18
作者回复	2018-06-19
工作线程一般不退出的	
TonyEasy	
老师，我有一点疑问，在线程池里复用线程时是不是对同一个线程调用了多次.start()方法呢？	2018-06-18
作者回复	2018-06-20
不是的，工作线程一般不退出的，复用的是类似Runnable这种	
扫地僧的功夫梦	

调用notify()/notifyAll()方法线程是变为阻塞状态吧，因为线程还没获取到锁。	2018-06-17
作者回复	
已回复，不是的	2018-06-19
三木子	
看了17讲回来留言threadlocal	2018-06-15
mongo	
杨老师请教你，关于高并发和线程池，我刚刚入门，工作中没有涉及过这一块。我阅读了oracle java tutorial high level concurrency 章节，阅读并粗略理解了《并发编程实践》这本书，想进一步清晰我的理解，我现在苦于在实践练习方面不知道怎么进行。老师有什么具体可行的思路指点一下吗？留言圈里有好多大神，在这里同时也请教其他的朋友。谢谢老师，谢谢大家。	2018-06-15
作者回复	
下面章节就会覆盖这部分，我谈下自己的思路：大部分工程师是没有机会在工作中，全面使用开发的那些东西的，尤其是反馈读者中初学者不少；所以，我建议有个整体性体系有个了解，分清大体都有什么；然后可以选些实践场景，去实现用代码。面试中大体也就够了，毕竟项目经验不是教程能解决的	2018-06-17
肖一林	
threadlocal和线程池结合的问题真的没考虑过	2018-06-14
作者回复	
线程池里的线程生命周期长	2018-06-14
Miaozhe	
问个问题，NIO 2的异步是不是利用协程的原理设计的？它实际运行的是多线程吗？	2018-06-14
作者回复	
我理解不是一回事，openjdk目前没有协程，Loom过程在做相关事情	2018-06-17
tyson	
1、站在应用程序方面，只创建了一个线程。 2、站在jvm方面，肯定还有gc等其余线程。	2018-06-14
总结： 1、线程是系统调度的最小单元，应该是进程吧。线程是操作系统的资源，在运行的时候会打开文件描述符等。 2、resume、stop、suspend等已经被废弃了 3、线程的等待和唤醒，建议使用reentrantlock的condition wait/notify方法 4、可以使用线程的join方法、countdownlatch、cyclicbarrier、future等进行线程的等待	
作者回复	
正解	2018-06-14
雷霖源的爸爸	
老师今天这课后题，又打脸了平时工作不仔细的地方，我首先想到的是好歹得sleep一下或打个断点用类似visualvm的工具看下，或者top之类数一下，赶着出门，回来试下	2018-06-14
作者回复	
主要是为了加深理解，这种也就是老学究关心，哈哈	2018-06-14
高杰	
有几个弱引用，虚引用的地方，音频和文字对不上，把我搞晕了。应该有2个线程，还有jvm的gc线程？还有第三个线程吗？	2018-06-14
作者回复	
翻译修正了，谢谢指出；你试试用比如最简单的stack查看下，不止这些哦	2018-06-14
甘建新	
线程得内存分配是怎么样子的呢？	2018-06-14
作者回复	
后边虚拟机那边介绍	2018-06-14
hanmshashou	
weak 应该不是幻象引用吧	2018-06-14
作者回复	
汗颜，已修正	2018-06-14
灰飞灰猪不会灰飞 烟灭	
老师 future模式是怎么异步返回结果的呢？是不是把每个线程的运行结果放到queue中，然后轮询queue返回结果？	2018-06-14
作者回复	
是说FutureTask的实现吗？我记得是有区别的	2018-06-15
食指可爱多	
我了解确定线程有：任务线程，Main线程，垃圾回收线程，还有些线程没细心关注名字和用途，惭愧了。可以在业务线程中等待，然后在命令行用stack看看当前jvm的线程堆栈。	2018-06-14
作者回复	
其他就包括我们前面章节说过的finalizer，各种cleaner等，还有事件处理等	2018-06-14

