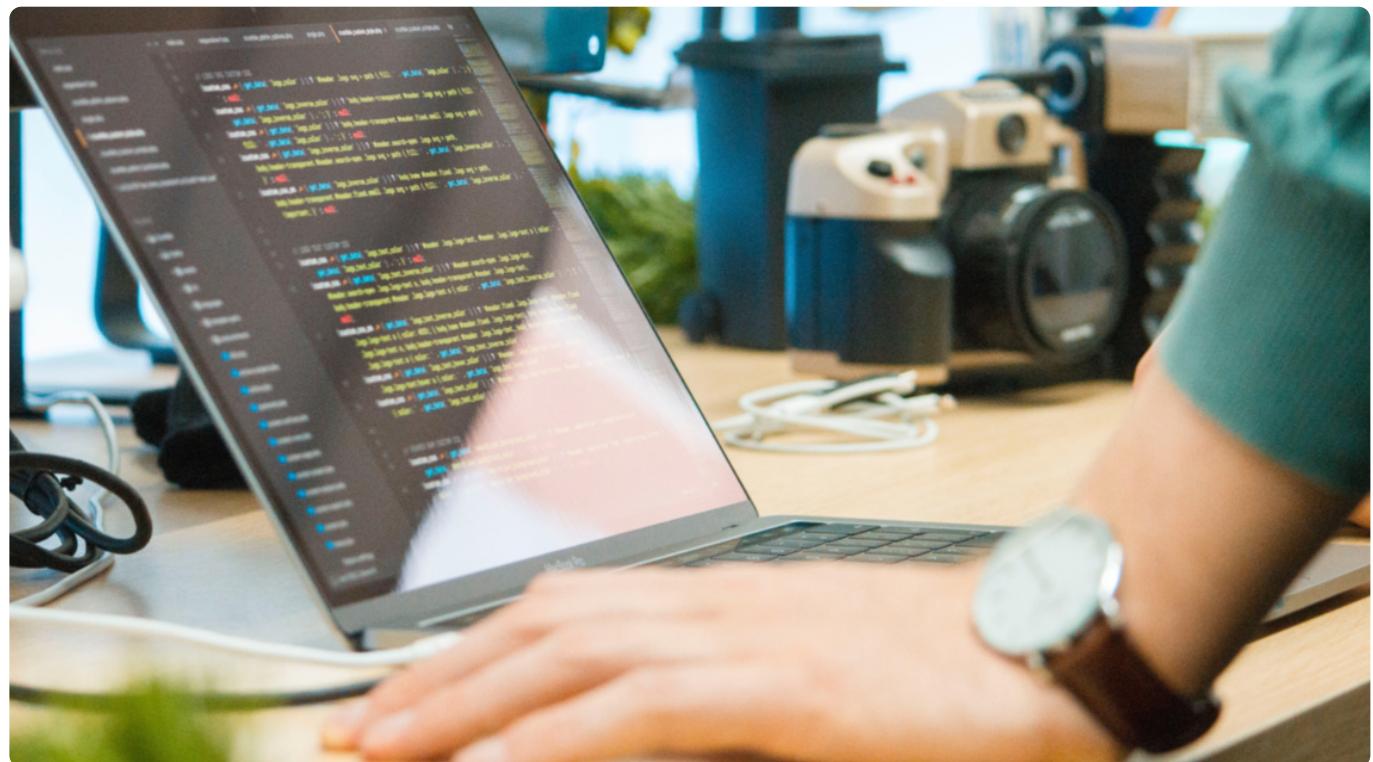


## 第17讲 | 一个线程两次调用start()方法会出现什么情况？

2018-06-14 杨晓峰

Java核心技术36讲

[进入课程 >](#)



讲述：黄洲君

时长 10:01 大小 4.59M



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，可以认为它是个 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：

 复制代码

```
1 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。

 复制代码

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

这种实现有利有弊，总体上来说，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：

 复制代码

```
1 Runnable task = () -> {System.out.println("Hello World!");};  
2 Thread myThread = new Thread(task);  
3 myThread.start();  
4 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：

 复制代码

```
1 Future future = Executors.newFixedThreadPool(1)  
2 .submit(task)  
3 .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：

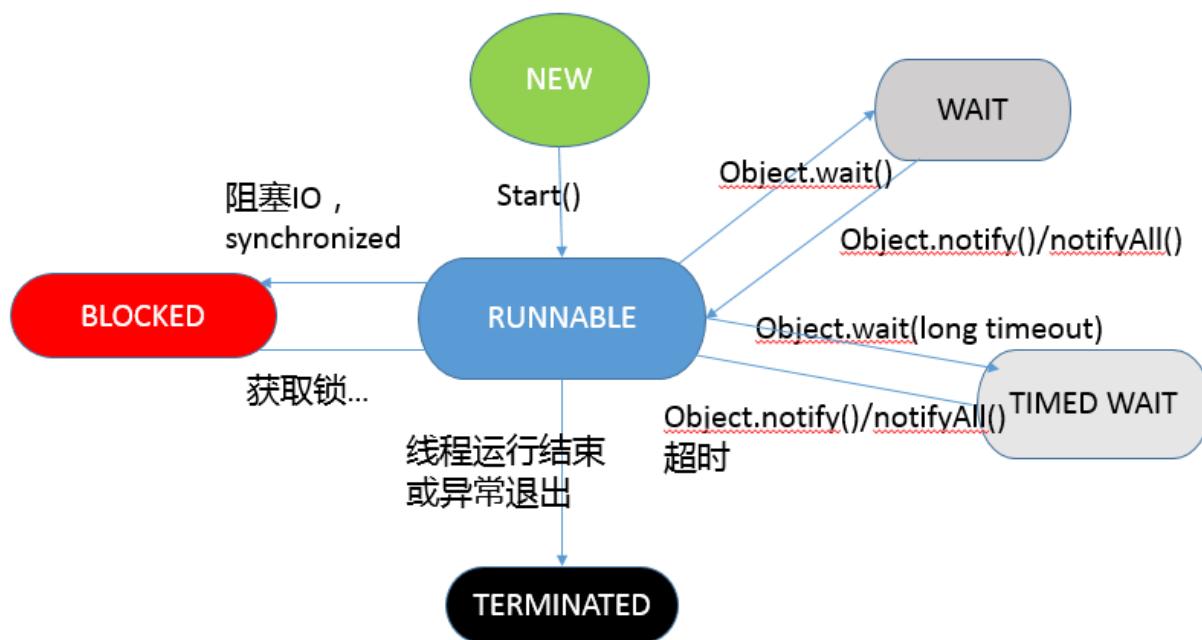
线程自身的方法，除了 start，还有多个 join 方法，等待线程结束；yield 是告诉调度器，主动让出 CPU；另外，就是一些已经被标记为过时的 resume、stop、suspend 之

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。**

 复制代码

```
1 Thread daemonThread = new Thread();
2 daemonThread.setDaemon(true);
3 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。

 复制代码

```
1 // 推荐
2 while ( isCondition() ) {
3     waitForAConfiton(...);
4 }
5
6 // 不推荐，可能引入 bug
7 if ( isCondition() ) {
8     waitForAConfiton(...);
9 }
10
```

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 提供的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。

 复制代码

```
1 static class ThreadLocalMap {
2     static class Entry extends WeakReference<ThreadLocal<?>> {
```

```
3     /** The value associated with this ThreadLocal. */
4     Object value;
5     Entry(ThreadLocal<?> k, Object v) {
6         super(k);
7         value = v;
8     }
9 }
10 // ...
11 }
```

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：

 复制代码

```
1 private void set(ThreadLocal<?> key, Object value) {
2     Entry[] tab = table;
3     int len = tab.length;
4     int i = key.threadLocalHashCode & (len-1);
5
6     for (Entry e = tab[i];; ...) {
7         //...
8         if (k == null) {
9             // 替换废弃条目
10            replaceStaleEntry(key, value, i);
11            return;
12        }
13    }
14
15    tab[i] = new Entry(key, value);
16    int sz = ++size;
17 // 扫描并清理发现的废弃条目，并检查容量是否超限
18    if (!cleanSomeSlots(i, sz) && sz >= threshold)
19        rehash(); // 清理废弃条目，如果仍然超限，则扩容（加倍）
20 }
```

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，所以通常都会建议，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。

请你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。



# Java核心技术36讲

— 前 Oracle 首席工程师  
带你修炼 Java 内功 —

杨晓峰 前 Oracle 首席工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇 第16讲 | synchronized底层如何实现？什么是锁的升级、降级？](#)

[下一篇 第18讲 | 什么情况下Java程序会产生死锁？如何定位、修复？](#)

## 精选留言 (38)

写留言



风动静泉

2018-06-14

39

一课一练：

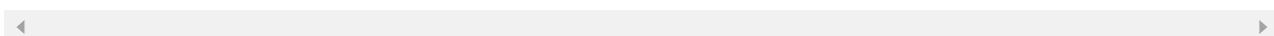
使用了两种方式获取当前程序的线程数。

- 1、使用线程管理器MXBean
- 2、直接通过线程组的activeCount

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...

展开 ▼

作者回复: 不错





qpm

2018-06-14

33

做了一个test分析老师的问题，观察到的情况如下：

JVM 启动 Hello World的线程分析

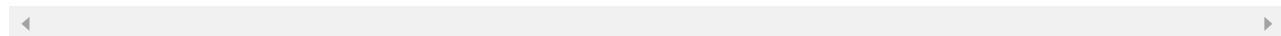
环境：

macOS + jdk8

检测获得...

展开 ▾

作者回复: 不错



行者

2018-06-14

12

“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。”

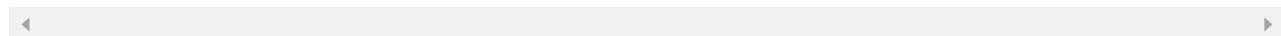
老师，Entry继承的是WeakReference，这个是弱引用吧。

main:

```
System.out.println("hello world");...
```

展开 ▾

作者回复: 前面是翻译错了，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



爱折腾的老...

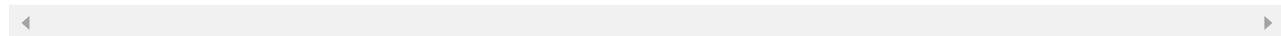
2018-06-14

10

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threadlocal里面的值是不是不会被回收？

展开 ▾

作者回复: 嗯，线程池一般不建议和thread local配合...



黄启航

2018-07-09

6

杨老师您好，我有个疑问：

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示地触发，否则就要等待线程的结束"。

...

展开 ▾



三木子

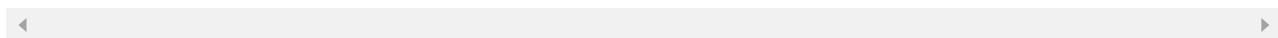
2018-06-15

4

现在觉得踩坑是一种很好学习方法

展开 ▾

作者回复: 同意



tyson

2018-06-14

4

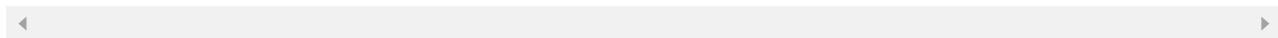
- 1、站在应用程序方面，只创建了一个线程。
- 2、站在jvm方面，肯定还有gc等其余线程。

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...

展开 ▾

作者回复: 不错



苦行僧

2018-12-30

3

边看老师的讲课 边反思工程代码

展开 ▾



锐

2018-06-14

2

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...

展开 ▾

作者回复: �恩，为了生命周期的需求



**TonyEasy**

2018-06-18

1

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？

作者回复: 不是的，工作线程一般不退出的，复用的是类似Runnable这种



**mongo**

2018-06-15

1

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。

展开 ▾

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，全面使用并发的那些东西的，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，去实现用例代码。面试中大体也就够了，毕竟项目经验不是教程能解决的



**sunlight00...**

2018-06-14

1

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😊

展开 ▼

作者回复: 不用了, 明确移除是好习惯



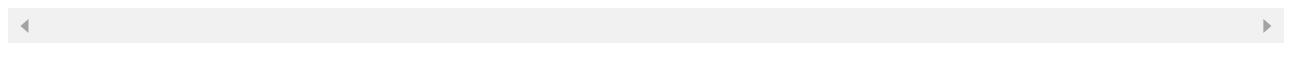
**食指可爱多**

2018-06-14

1

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jstack看看当前jvm的线程堆栈。

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



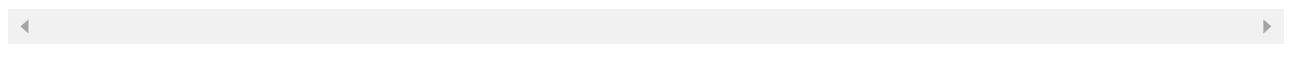
**Eason**

2018-06-14

1

“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? ?

作者回复: wait和blocked是不同的



**Zach\_**

2019-04-28

1

这是怎么检测出来 有多少个线程在后台运行的啊?

展开 ▼



**流光**

2019-04-11

1

```
创建单例线程, Future o = Executors.newSingleThreadExecutor().submit() -> {  
    return "asdf";  
};
```

这样更优雅些吧

展开 ▼



QQ怪

2019-04-01



跟我们讲下协程主要是干什么的，好吗

展开 ▼



DL

2019-03-15



老师您好，对threadLocal的OOM有一些疑问。既然Map中的key是弱引用，是不是意味着只要我们能够在编程中注意将new出来的对象的强引用去掉（=null），后续的gc会自动清理掉弱引用key，对于key=null的value的清理在后续的set等方法中会清理。这样即使在线程复用的情况下如何会出现OOM呢？

展开 ▼



峻铭

2018-12-11



我还是没明白为什么不能start两次，知其然不知其所以然



王子瑞Ailil...

2018-11-26



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...

展开 ▼