

01 | 可见性、原子性和有序性问题：并发编程Bug的源头

2019-02-28 王宝令



讲述：王宝令

时长 15:28 大小 14.18M



如果你细心观察的话，你会发现，不管是哪一门编程语言，并发类的知识都是在高级篇里。换句话说，这块知识点其实对于程序员来说，是比较进阶的知识。我自己这么多年学习过来，也确实觉得并发是比较难的，因为它会涉及到很多的底层知识，比如若你对操作系统相关的知识一无所知的话，那去理解一些原理就会费些力气。这是我们整个专栏的第一篇文章，我说这些话的意思是如果你在中遇到自己没想通的问题，可以去查阅资料，也可以在评论区找我，以保证你能够跟上学习进度。

你我都知道，编写正确的并发程序是一件极困难的事情，并发程序的 Bug 往往会诡异地出现，然后又诡异地消失，很难重现，也很难追踪，很多时候都让人很抓狂。但要快速而又精准地解决“并发”类的疑难杂症，你就要理解这件事情的本质，追本溯源，深入分析这些 Bug 的源头在哪里。

那为什么并发编程容易出问题呢？它是怎么出问题的？今天我们就重点聊聊这些 Bug 的源头。

并发程序幕后的故事

这些年，我们的 CPU、内存、I/O 设备都在不断迭代，不断朝着更快的方向努力。但是，在这个快速发展的过程中，有一个**核心矛盾一直存在，就是这三者的速度差异**。CPU 和内存的速度差异可以形象地描述为：CPU 是天上一天，内存是地上一小时（假设 CPU 执行一条普通指令需要一天，那么 CPU 读写内存得等待一年的时间）。内存和 I/O 设备的速度差异就更大了，内存是天上一天，I/O 设备是地上十年。

程序里大部分语句都要访问内存，有些还要访问 I/O，根据木桶理论（一只水桶能装多少水取决于它最短的那块木板），程序整体的性能取决于最慢的操作——读写 I/O 设备，也就是说单方面提高 CPU 性能是无效的。

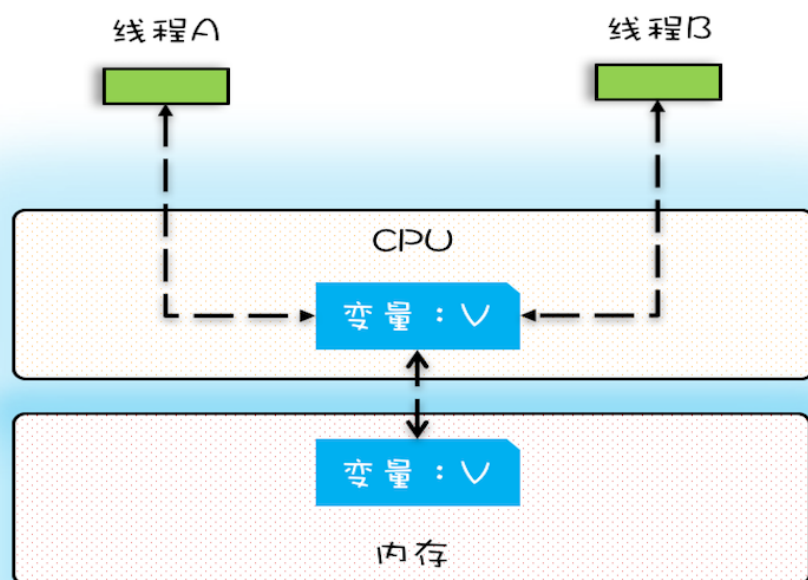
为了合理利用 CPU 的高性能，平衡这三者的速度差异，计算机体系机构、操作系统、编译程序都做出了贡献，主要体现为：

1. CPU 增加了缓存，以均衡与内存的速度差异；
2. 操作系统增加了进程、线程，以分时复用 CPU，进而均衡 CPU 与 I/O 设备的速度差异；
3. 编译程序优化指令执行次序，使得缓存能够得到更加合理地利用。

现在我们几乎所有的程序都默默地享受着这些成果，但是天下没有免费的午餐，并发程序很多诡异问题的根源也在这里。

源头之一：缓存导致的可见性问题

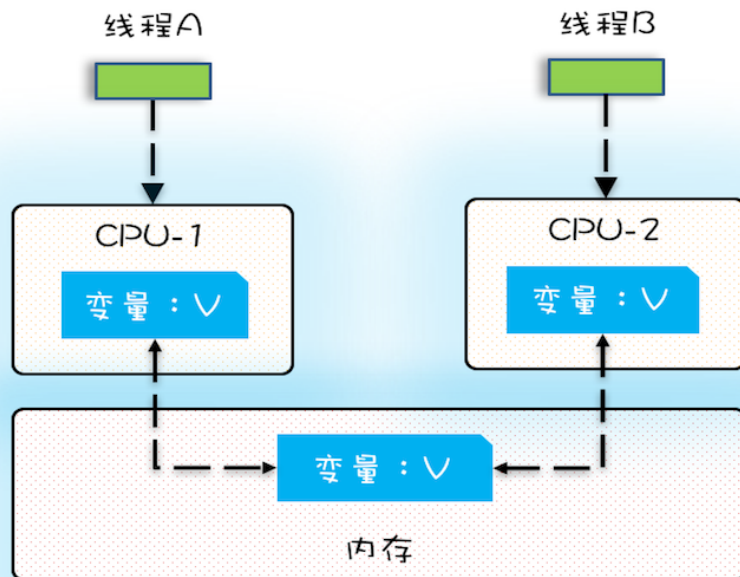
在单核时代，所有的线程都是在一颗 CPU 上执行，CPU 缓存与内存的数据一致性容易解决。因为所有线程都是操作同一个 CPU 的缓存，一个线程对缓存的写，对另外一个线程来说一定是可见的。例如在下面的图中，线程 A 和线程 B 都是操作同一个 CPU 里面的缓存，所以线程 A 更新了变量 V 的值，那么线程 B 之后再访问变量 V，得到的一定是 V 的最新值（线程 A 写过的值）。



CPU 缓存与内存的关系图


一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称为**可见性**。

多核时代，每颗 CPU 都有自己的缓存，这时 CPU 缓存与内存的数据一致性就没那么容易解决了，当多个线程在不同的 CPU 上执行时，这些线程操作的是不同的 CPU 缓存。比如下图中，线程 A 操作的是 CPU-1 上的缓存，而线程 B 操作的是 CPU-2 上的缓存，很明显，这个时候线程 A 对变量 V 的操作对于线程 B 而言就不具备可见性了。这个就属于硬件程序员给软件程序员挖的“坑”。



多核 CPU 的缓存与内存关系图

下面我们再用一段代码来验证一下多核场景下的可见性问题。下面的代码，每执行一次 `add10K()` 方法，都会循环 10000 次 `count+=1` 操作。在 `calc()` 方法中我们创建了两个线程，每个线程调用一次 `add10K()` 方法，我们来想一想执行 `calc()` 方法得到的结果应该是多少呢？

 复制代码

```
1 public class Test {
2     private long count = 0;
3     private void add10K() {
4         int idx = 0;
5         while(idx++ < 10000) {
6             count += 1;
7         }
8     }
9     public static long calc() {
10         final Test test = new Test();
11         // 创建两个线程，执行 add() 操作
12         Thread th1 = new Thread(()->{
13             test.add10K();
14         });
15         Thread th2 = new Thread(()->{
16             test.add10K();
17         });
18         // 启动两个线程
19         th1.start();
20         th2.start();
21         // 等待两个线程执行结束
```



```

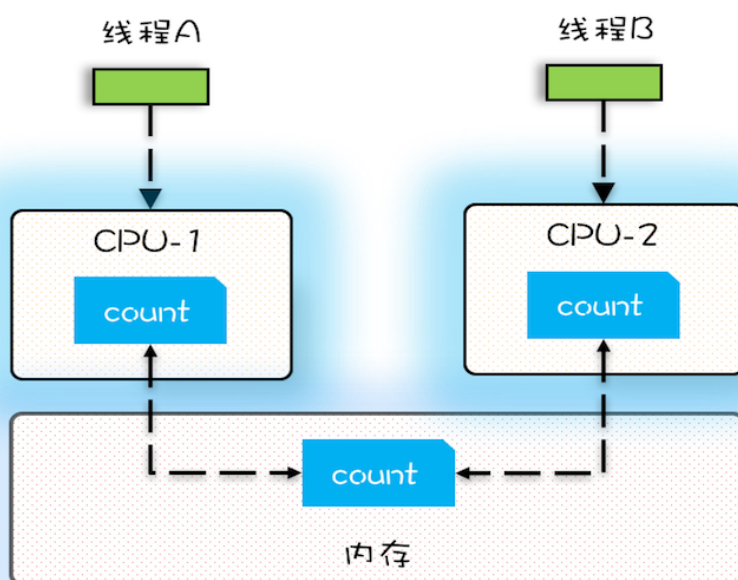
22     th1.join();
23     th2.join();
24     return count;
25 }
26 }

```

直觉告诉我们应该是 20000，因为在单线程里调用两次 add10K() 方法，count 的值就是 20000，但实际上 calc() 的执行结果是个 10000 到 20000 之间的随机数。为什么呢？

我们假设线程 A 和线程 B 同时开始执行，那么第一次都会将 count=0 读到各自的 CPU 缓存里，执行完 count+=1 之后，各自 CPU 缓存里的值都是 1，同时写入内存后，我们会发现内存中是 1，而不是我们期望的 2。之后由于各自的 CPU 缓存里都有了 count 的值，两个线程都是基于 CPU 缓存里的 count 值来计算，所以导致最终 count 的值都是小于 20000 的。这就是缓存的可见性问题。

循环 10000 次 count+=1 操作如果改为循环 1 亿次，你会发现效果更明显，最终 count 的值接近 1 亿，而不是 2 亿。如果循环 10000 次，count 的值接近 20000，原因是两个线程不是同时启动的，有一个时差。

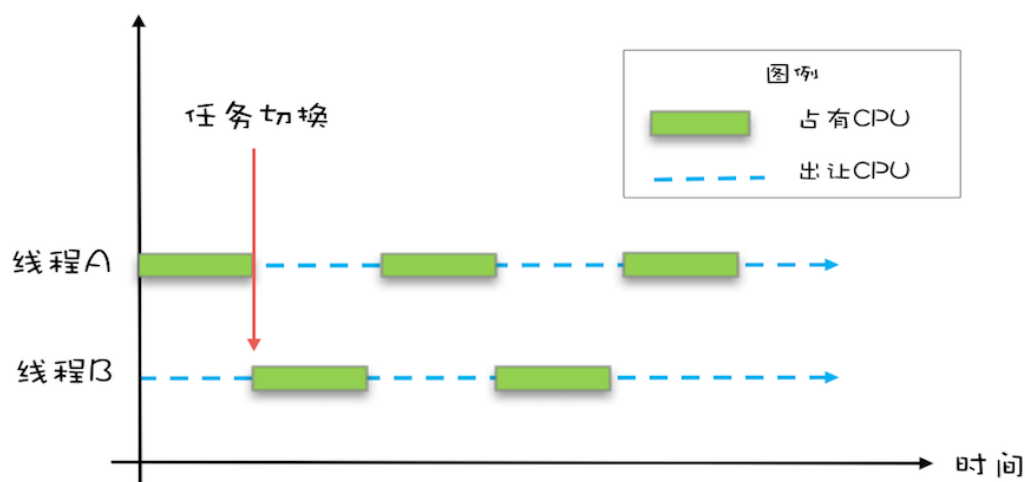


变量 count 在 CPU 缓存和内存的分布图

源头之二：线程切换带来的原子性问题

由于 IO 太慢，早期的操作系统就发明了多进程，即便在单核的 CPU 上我们也可以一边听着歌，一边写 Bug，这个就是多进程的功劳。

操作系统允许某个进程执行一小段时间，例如 50 毫秒，过了 50 毫秒操作系统就会重新选择一个进程来执行（我们称为“任务切换”），这个 50 毫秒称为“时间片”。



线程切换示意图

在一个时间片内，如果一个进程进行一个 IO 操作，例如读个文件，这个时候该进程可以把自己标记为“休眠状态”并出让 CPU 的使用权，待文件读进内存，操作系统会把这个休眠的进程唤醒，唤醒后的进程就有机会重新获得 CPU 的使用权了。

这里的进程在等待 IO 时之所以会释放 CPU 使用权，是为了让 CPU 在这段等待时间里可以做别的事情，这样一来 CPU 的使用率就上来了；此外，如果这时有另外一个进程也读文件，读文件的操作就会排队，磁盘驱动在完成一个进程的读操作后，发现有排队的任务，就会立即启动下一个读操作，这样 IO 的使用率也上来了。

是不是很简单逻辑？但是，虽然看似简单，支持多进程分时复用在操作系统的发展史上却具有里程碑意义，Unix 就是因为解决了这个问题而名噪天下的。

早期的操作系统基于进程来调度 CPU，不同进程间是不共享内存空间的，所以进程要做任务切换就要切换内存映射地址，而一个进程创建的所有线程，都是共享一个内存空间的，

所以线程做任务切换成本就很低了。现代的操作系统都基于更轻量的线程来调度，现在我们提到的“任务切换”都是指“线程切换”。

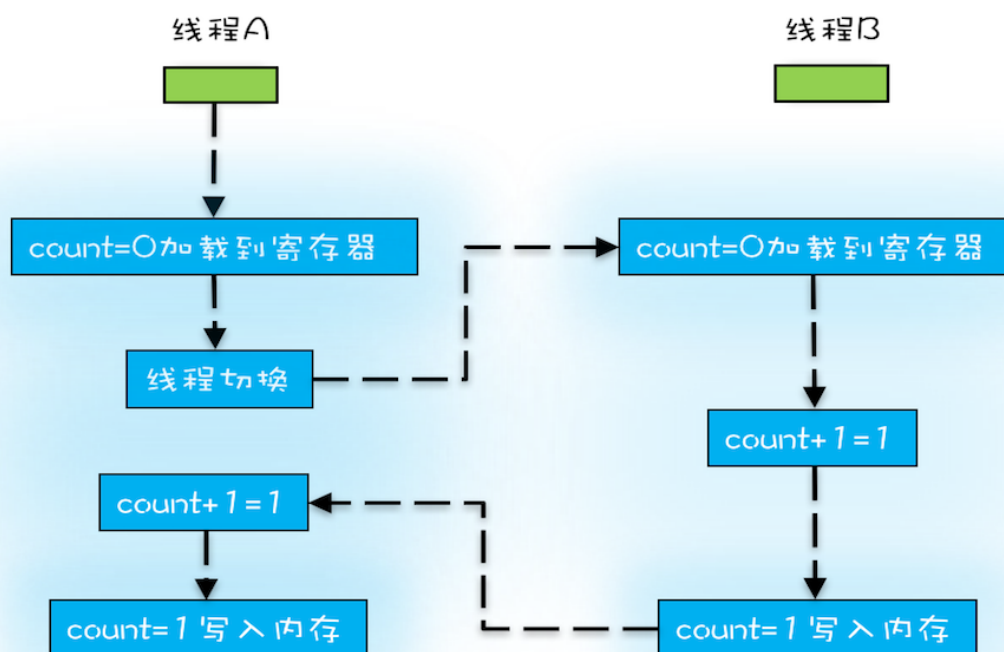
Java 并发程序都是基于多线程的，自然也会涉及到任务切换，也许你想不到，任务切换竟然也是并发编程里诡异 Bug 的源头之一。任务切换的时机大多数是在时间片结束的时候，我们现在基本都使用高级语言编程，高级语言里一条语句往往需要多条 CPU 指令完成，例如上面代码中的 `count += 1`，至少需要三条 CPU 指令。

指令 1：首先，需要把变量 `count` 从内存加载到 CPU 的寄存器；

指令 2：之后，在寄存器中执行 `+1` 操作；

指令 3：最后，将结果写入内存（缓存机制导致可能写入的是 CPU 缓存而不是内存）。

操作系统做任务切换，可以发生在任何一条**CPU 指令**执行完，是的，是 CPU 指令，而不是高级语言里的一条语句。对于上面的三条指令来说，我们假设 `count=0`，如果线程 A 在指令 1 执行完后做线程切换，线程 A 和线程 B 按照下图的序列执行，那么我们会发现两个线程都执行了 `count+=1` 的操作，但是得到的结果不是我们期望的 2，而是 1。



非原子操作的执行路径示意图


我们潜意识里面觉得 `count+=1` 这个操作是一个不可分割的整体，就像一个原子一样，线程的切换可以发生在 `count+=1` 之前，也可以发生在 `count+=1` 之后，但就是不会发生在中间。我们把一个或者多个操作在 CPU 执行的过程中不被中断的特性称为原子性。

CPU 能保证的原子操作是 CPU 指令级别的，而不是高级语言的操作符，这是违背我们直觉的地方。因此，很多时候我们需要在高级语言层面保证操作的原子性。

源头之三：编译优化带来的有序性问题

那并发编程里还有没有其他有违直觉容易导致诡异 Bug 的技术呢？有的，就是有序性。顾名思义，有序性指的是程序按照代码的先后顺序执行。编译器为了优化性能，有时候会改变程序中语句的先后顺序，例如程序中：“a=6；b=7；” 编译器优化后可能变成“b=7；a=6；”，在这个例子中，编译器调整了语句的顺序，但是不影响程序的最终结果。不过有时候编译器及解释器的优化可能导致意想不到的 Bug。

在 Java 领域一个经典的案例就是利用双重检查创建单例对象，例如下面的代码：在获取实例 `getInstance()` 的方法中，我们首先判断 `instance` 是否为空，如果为空，则锁定 `Singleton.class` 并再次检查 `instance` 是否为空，如果还为空则创建 `Singleton` 的一个实例。

 复制代码

```
1 public class Singleton {
2     static Singleton instance;
3     static Singleton getInstance(){
4         if (instance == null) {
5             synchronized(Singleton.class) {
6                 if (instance == null)
7                     instance = new Singleton();
8             }
9         }
10        return instance;
11    }
12 }
```

假设有两个线程 A、B 同时调用 `getInstance()` 方法，他们会同时发现 `instance == null`，于是同时对 `Singleton.class` 加锁，此时 JVM 保证只有一个线程能够加锁成功（假设是线程 A），另外一个线程则会处于等待状态（假设是线程 B）；线程 A 会创建一个 `Singleton` 实例，之后释放锁，锁释放后，线程 B 被唤醒，线程 B 再次尝试加锁，此时是可以加锁成功的，加锁成功后，线程 B 检查 `instance == null` 时会发现，已经创建过 `Singleton` 实例了，所以线程 B 不会再创建一个 `Singleton` 实例。

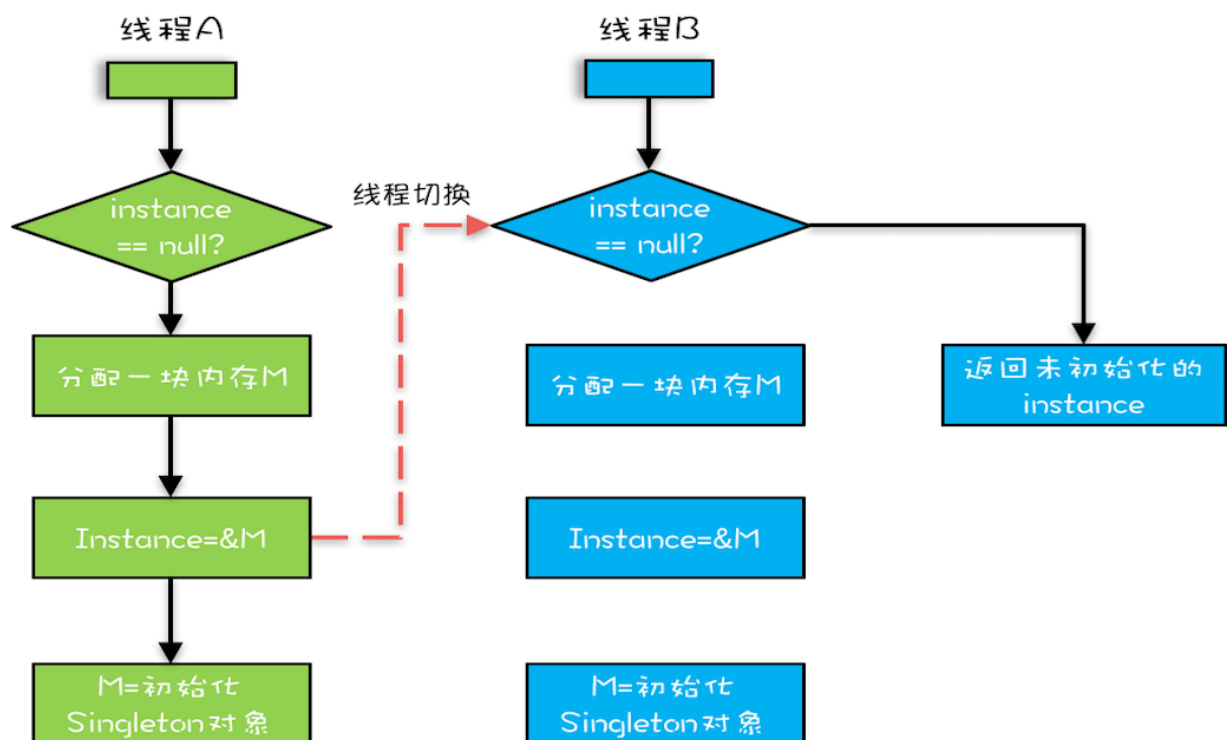
这看上去一切都很完美，无懈可击，但实际上这个 `getInstance()` 方法并不完美。问题出在哪里呢？出在 `new` 操作上，我们以为的 `new` 操作应该是：

1. 分配一块内存 `M`；
2. 在内存 `M` 上初始化 Singleton 对象；
3. 然后 `M` 的地址赋值给 `instance` 变量。

但是实际上优化后的执行路径却是这样的：

1. 分配一块内存 `M`；
2. 将 `M` 的地址赋值给 `instance` 变量；
3. 最后在内存 `M` 上初始化 Singleton 对象。

优化后会导致什么问题呢？我们假设线程 A 先执行 `getInstance()` 方法，当执行完指令 2 时恰好发生了线程切换，切换到了线程 B 上；如果此时线程 B 也执行 `getInstance()` 方法，那么线程 B 会发现 `instance != null`，所以直接返回 `instance`，而此时的 `instance` 是没有初始化过的，如果我们这个时候访问 `instance` 的成员变量就可能触发空指针异常。



双重检查创建单例的异常执行路径

总结

要写好并发程序，首先要知道并发程序的问题在哪里，只有确定了“靶子”，才有可能把问题解决，毕竟所有的解决方案都是针对问题的。并发程序经常出现的诡异问题看上去非常无厘头，但是深究的话，无外乎就是直觉欺骗了我们，**只要我们能够深刻理解可见性、原子性、有序性在并发场景下的原理，很多并发 Bug 都是可以理解、可以诊断的。**

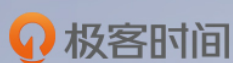
在介绍可见性、原子性、有序性的时候，特意提到**缓存**导致的可见性问题，**线程切换**带来的原子性问题，**编译优化**带来的有序性问题，其实缓存、线程、编译优化的目的和我们写并发程序的目的是相同的，都是提高程序性能。但是技术在解决一个问题的同时，必然会带来另外一个问题，所以**在采用一项技术的同时，一定要清楚它带来的问题是什么，以及如何规避。**

我们这个专栏在讲解每项技术的时候，都会尽量将每项技术解决的问题以及产生的问题讲清楚，也希望你能在这方面多思考、多总结。

课后思考

常听人说，在 32 位的机器上对 long 型变量进行加减操作存在并发隐患，到底是不是这样呢？现在相信你一定能分析出来。

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」，10位好友免费读，邀请订阅更有**现金**奖励。

上一篇 学习攻略 | 如何才能学好并发编程？

下一篇 02 | Java内存模型：看Java如何解决可见性和有序性问题

精选留言 (124)

写留言



Jialin

2019-02-28

48

对于双重锁的问题，我觉得任大鹏分析的蛮有道理，线程A进入第二个判空条件，进行初始化时，发生了时间片切换，即使没有释放锁，线程B刚要进入第一个判空条件时，发现条件不成立，直接返回instance引用，不用去获取锁。如果对instance进行volatile语义声明，就可以禁止指令重排序，避免该情况发生。

对于有些同学对CPU缓存和内存的疑问，CPU缓存不存在于内存中的，它是一块比内存...
展开

作者回复: 厉害厉害，比我回答的全面多了



任大鹏

2019-02-28

47

对于阿根一世同学的那个疑问，我个人认为CPU时间片切换后，线程B刚好执行到第一次判断instance==null，此时不为空，不用进入synchronized里，就将还未初始化的instance返回了

作者回复: 正解！感谢回复。



json

2019-02-28

47

long类型64位，所以在32位的机器上，对long类型的数据操作通常需要多条指令组合出来，无法保证原子性，所以并发的时候会出问题□□□

作者回复: 正解



阿根一世

2019-02-28

👍 46

对于双重锁检查那个例子，我有一个疑问，A如果没有完成实例的初始化，锁应该不会释放的，B是拿不到锁的，怎么还会出问题呢？

作者回复: 后面好多同学已经帮我作答了，教好学生，饿死师傅啊



QIANQIANCH...

2019-02-28

👍 13

刚看过《java并发实战》，又是看了个开始就看不下去了😂😂，希望订阅专栏可以跟老师和其他童鞋一起坚持学习并发编程😂😂

思考题：在32位的机器上对long型变量进行加减操作存在并发隐患的说法是正确的。原因就是文章里的bug源头之二：线程切换带来的原子性问题。...

展开 ∨

作者回复: 厉害厉害



xx鼠

2019-02-28

👍 12

Singleton instance改为volatile或者final就完美了，这里面其实涉及Java的happen-before原则。

作者回复: 恭喜你，学会抢答了！



summer_Da...

2019-02-28

👍 7

我觉得阿根一世的问题应该是
`synchronized(Singleton.class) {`

```
if (instance == null)
    instance = new Singleton();
}...
```

展开 ▾



落墨

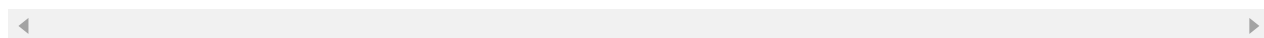
2019-02-28

👍 6

老师,运行文中的测试代码,有时会出现9000多的结果,不知道是什么原因?

展开 ▾

作者回复: 并发程序的诡异之处,就在于:我实在也想不通。



黄朋飞

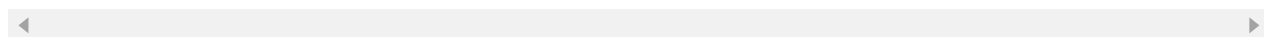
2019-02-28

👍 6

老师你好,请问文章中的缓存和内存什么区别,缓存不是在内存中存放着吗?

展开 ▾

作者回复: 对不起,是我没说清楚,这里的缓存,指的是CPU缓存。



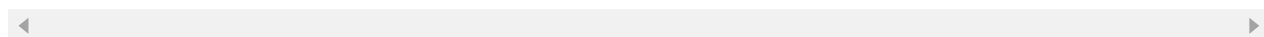
我会得到

2019-02-28

👍 6

零点一过刚好看到更新,果断一口气读完,带劲!可见性,原子性,有序性,操作系统作为基础,内存模型,机器指令,编译原理,一个都不能少,开始有点意思了👉

作者回复: 后面讲内存模型,会更有意思。



Blithe

2019-02-28

👍 5

对于阿根一世的提问,以及文中作者的描述,我有自己的看法。阿根一世的提问是对的,作者的描述是有误的,但作者的结论是正确的。

我的解释如下:两个线程都过了第一层判空后,第二个线程不会出现文中说的空指针异

常。因为JSR-133中的happens-before规则。1.一个线程中的每个操作先于线程中的后续操作。2.对一个锁的解锁先于随后对这个锁的解锁。3.传递行。综合以上三条规则，第...

展开 ▾

作者回复: 我看了一下，的确是我的描述有问题。感谢啊！



发条橙子 ...

2019-02-28

👍 4

老师，我有几个问题希望老师指点，也是涉及到操作系统的：

1. 操作系统是以进程为单位共享资源，以线程单位进行调用。多个线程共享一个进程的资源。一个java应用占一个进程（jvm的内存模型的资源也在这个进程中），一个进程占一个cpu，所以老师所说的多核cpu缓存，每个cpu有自己的缓存，AB两个线程在不...

展开 ▾

作者回复: 进程和线程的关系，你可以看看操作系统原理。进程不占有CPU。操作系统会把CPU分配给线程。分到CPU的线程就能执行。

并行，是同一时刻，两个线程都在执行。并发，是同一时刻，只有一个执行，但是一个时间段内，两个线程都执行了。



嘎嘎

2019-02-28

👍 4

针对阿根一世的问题，问题其实出现在new Singleton()这里。

这一行分对于CPU来讲，有3个指令：

- 1.分配内存空间
- 2.初始化对象
- 3.instance引用指向内存空间...

展开 ▾

作者回复: 厉害，一看就是经验丰富



牧童纪年

2019-02-28

👍 3

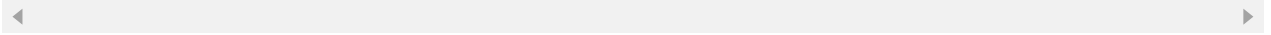
王老师，你文章中讲的 优化指令的执行次序 使得缓存能够更加合理的利用是什么意思？

展开 ∨

作者回复: 比如第1行：a=8

第1000行：a=a*2;

这个时候，把他们放到一起执行，是不是就能更好的利用缓存了？



cfreedomc

2019-02-28

👍 3

今天主要学习了并发编程中三种类型的问题

- 1.缓存导致的可见性问题
- 2.线程切换导致的原子性问题
- 3.编译优化带来的有序性问题

也是让我认识到我们编程其实和医生看病一样，项目就是病人，当你给病人开药时，药...

展开 ∨



Reloaded

2019-02-28

👍 2

对于创建单例那个问题，可以在声明变量的时候加上 volatile 关键字，有两个作用,一是可以禁止指令重排序(源头三)，二是保证了程序的可见性(源头一)



Geek_df121...

2019-02-28

👍 2

volatile关键字可以禁止指令重排序，可以解决大部分人的疑惑

展开 ∨



牧童纪年

2019-02-28

👍 2

对于Jialin同学的volatile 语义的理解，我不太认可，volatile关键字 应该是no cached，所以不会出现“强制重新把内存中的内容写入到自己的缓存”这一说，都是写操作完事后，如果不发生任务切换，则直接将修改的最新的值刷新到内存中的把？@王老师，你说呢？

展开 ∨

作者回复: volatile 语义下一期会讲。但是不会涉及具体如何实现的。如果大家感兴趣，可以参考《Java并发编程的艺术》



terry

2019-02-28

👍 2

讲得真是通俗易懂，尤其是结合三张图的说明，真的是足够清晰了。之前都是靠去记一些概念和规则，老师讲的是通过原理结合图例去说明，并讲述了背景以及硬件和操作系统设计的原因，以及产生的新问题，这回理解了。

展开 ∨



0928

2019-02-28

👍 2

long和double类型的变量是64位，但是在32位JVM中，32位的JVM会将64位数据的读写操作分为2次32位的读写操作来进行，这就导致了long、double类型的变量在32位虚拟机中是非原子操作所以在并发时存在数据不一致的问题。

展开 ∨