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

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间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，内存是地上一年（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。

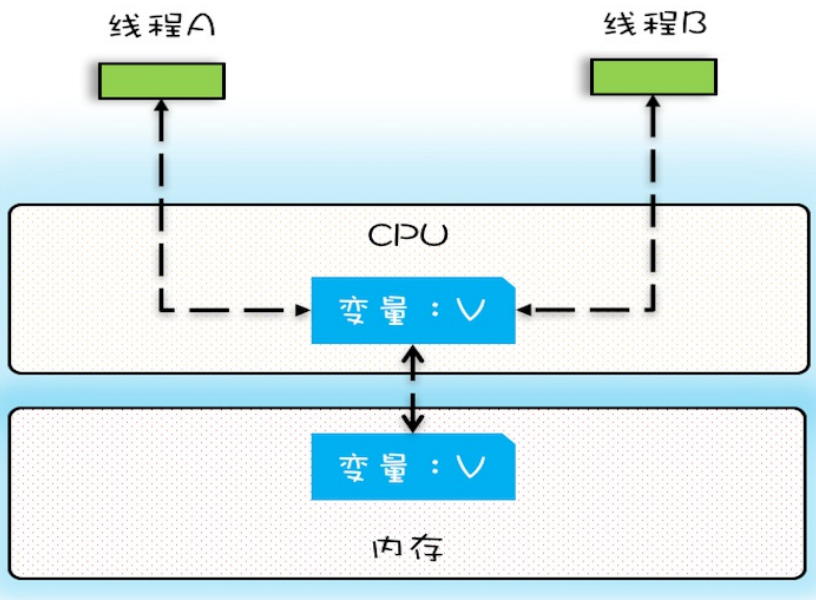
为了合理利用CPU的高性能，平衡这三者的速度差异，计算机体系机构、操作系统、编译程序都做出了贡献，主要体现为：

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）。

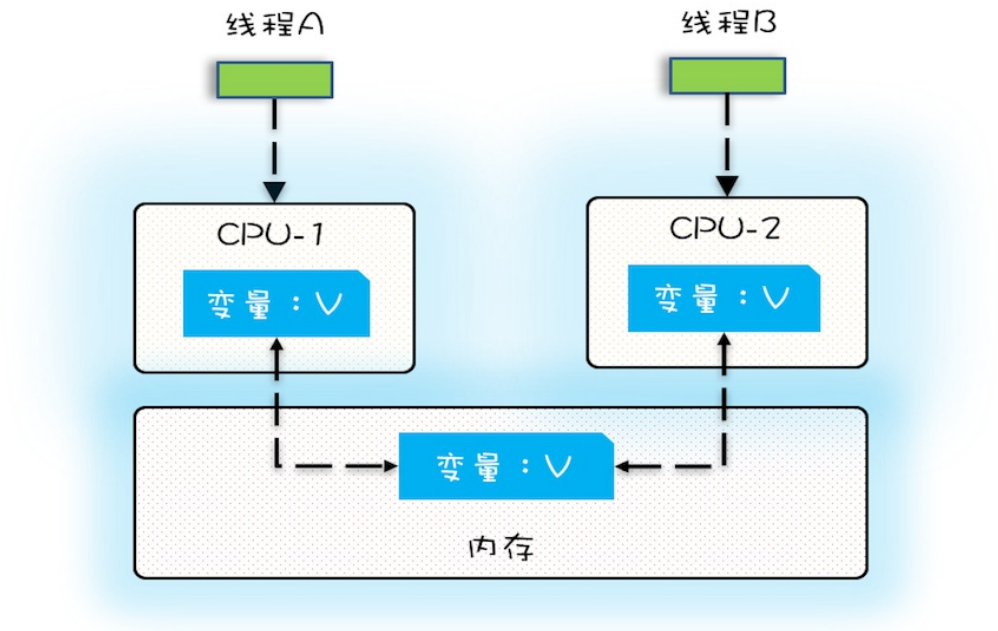


CPU缓存与内存的关系图

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？

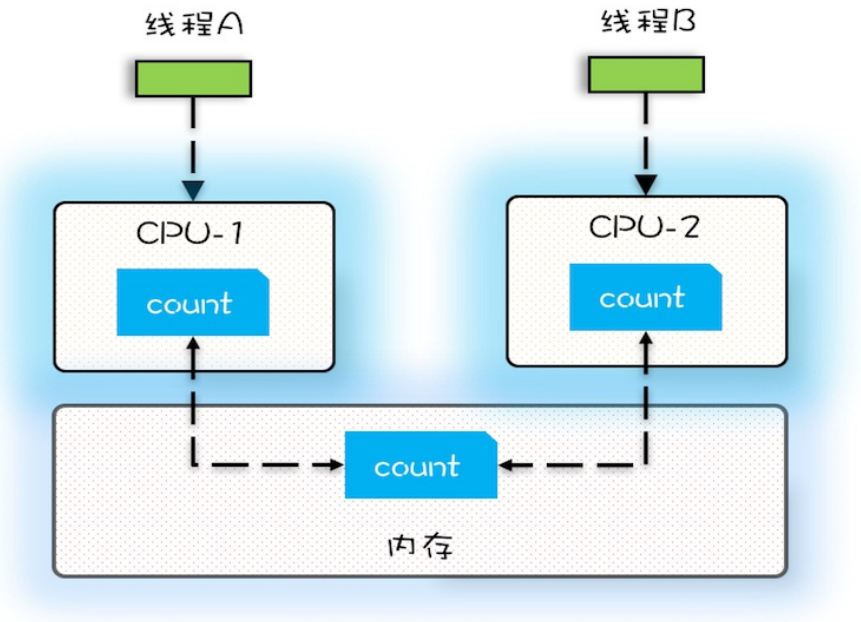
```
public class Test {
  private long count = 0;
  private void add10K() {
    int idx = 0;
    while(idx++ < 10000) {
      count += 1;
    }
  }
  public static long calc() {
    final Test test = new Test();
    // 创建两个线程，执行add()操作
    Thread th1 = new Thread(()->{
      test.add10K();
    });
    Thread th2 = new Thread(()->{
      test.add10K();
    });
    // 启动两个线程
    th1.start();
    th2.start();
    // 等待两个线程执行结束
    th1.join();
    th2.join();
    return count;
  }
}
```

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。

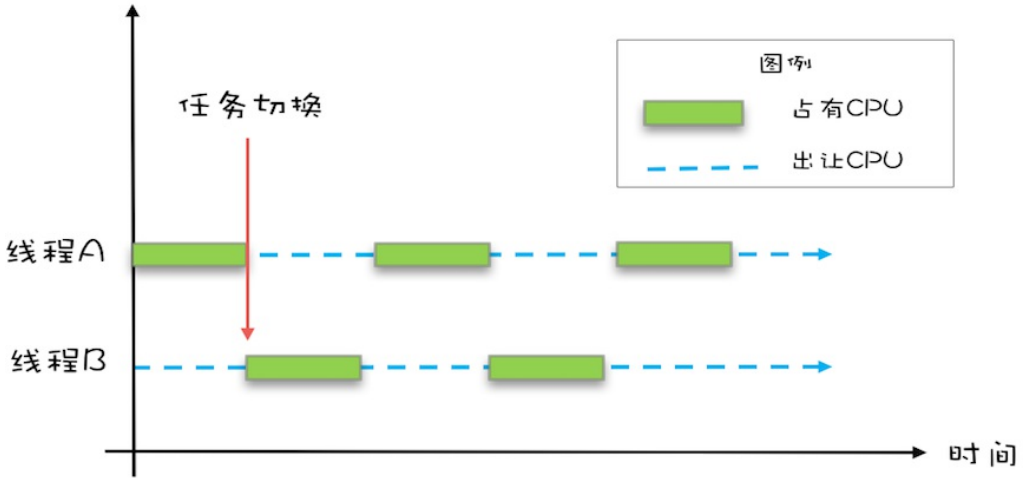


变量count在CPU缓存和内存的分布图

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。

是不是很简单的逻辑？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。

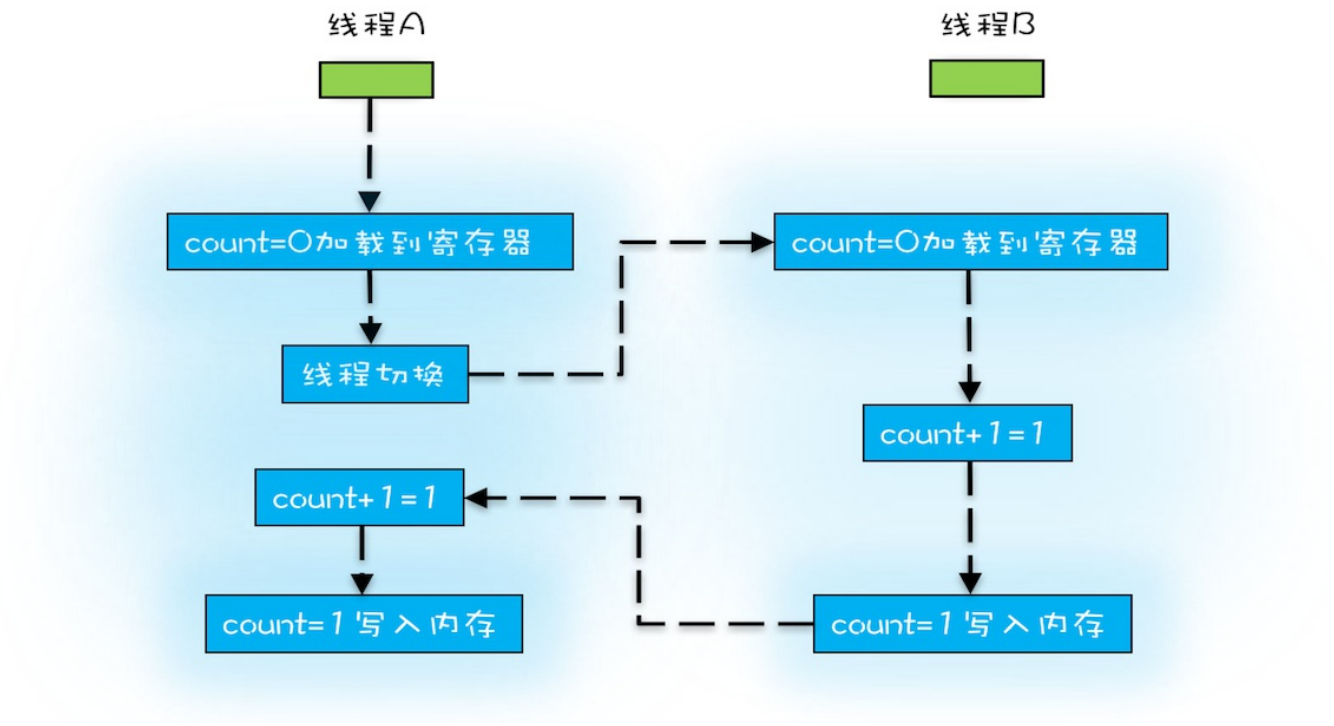
早期的操作系统基于进程来调度CPU，不同进程间是不共享内存空间的，所以进程要做任务切换就要切换内存映射地址，而一个进程创建的所有线程，都是共享一个内存空间的，所以线程做任务切换成本就很低了。现代的操作系统都基于更轻量的线程来调度，现在我们提到的“任务切换”都是指“线程切换”。

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。

```
public class Singleton {
  static Singleton instance;
  static Singleton getInstance(){
    if (instance == null) {
      synchronized(Singleton.class) {
        if (instance == null)
          instance = new Singleton();
        }
    }
    return instance;
  }
}
```

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。

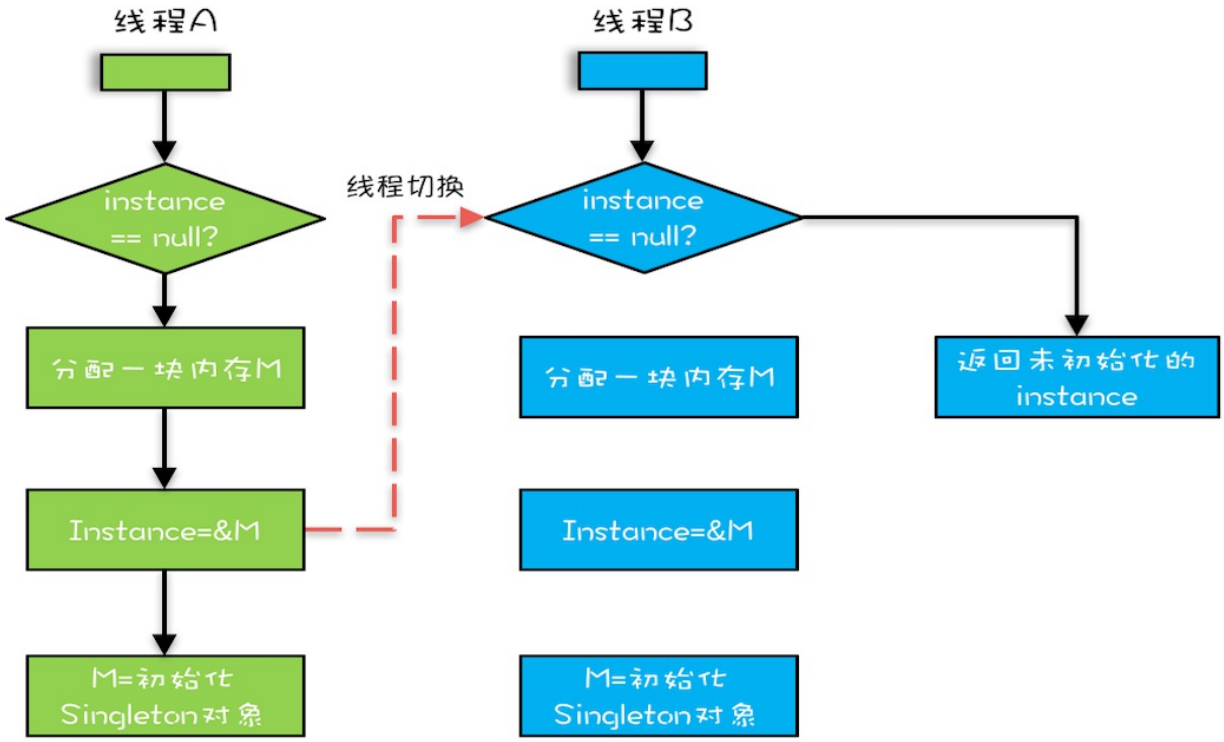
这看上去一切都很完美，无懈可击，但实际上这个getInstance()方法并不完美。问题出在哪里呢？出在new操作上，我们以为的new操作应该是：

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在执行第一个判断时会发现 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，也希望你能够在这方面多思考、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。

精选留言

Jialin

203

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，它是一块比内存更小、读写速度更快的芯片，至于什么时候把数据从缓存写到内存，没有固定的时间，同样地，对于有volatile语义声明的变量，线程A执行完后会强制将值刷新到内存中，线程B进行相关操作时会强制重新把内存中的内容写入到自己的缓存，这就涉及到了volatile的写入屏障问题，当然也就是所谓happen-before问题。

2019-02-28

作者回复

厉害厉害，比我回答的全面多了

2019-02-28

coder

106

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[illegible]

2019-02-28

作者回复

正解

2019-02-28

任大鹏 80

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

2019-02-28

作者回复

正解！感谢回复。

2019-02-28

阿根一世 73

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？

2019-02-28

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

2019-02-28

Blithe 54

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，第一个线程的对象初始化完成先于解锁，第一个线程的解锁先于第二个线程的加锁。所以，真如作者所说第二线程如果过了第一层的判空校验，下步就要加锁，加锁后其实对象在线程一中已经初始化结束，不会出现NPE。但例子的确会出现NPE问题，出现的场景却是，线程二没有进行到第一层判空操作，线程一到了文中说的时间片结束，让出CPU，线程二判空，否，执行对象调用方法。有问题，可加微信交流Blithe-Feng

2019-02-28

作者回复

我看了一下，的确是我的描述有问题。感谢啊！

2019-03-01

CHEN 35

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。

非volatile类型的long和double型变量是8字节64位的，32位机器读或写这个变量时得把人家咔嚓分成两个32位操作，可能一个线程读了某个值的高32位，低32位已经被另一个线程改了。所以官方推荐最好把long\double 变量声明为volatile或是同步加锁synchronize以避免并发问题。

贴一段java文档的说明
https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7

17.7. Non-Atomic Treatment of double and long

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid possible complications.

2019-02-28

作者回复

厉害厉害

2019-03-01

嘎嘎

31

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
正常执行顺序1-2-3
但是CPU重排序后执行顺序可能为1-3-2，那么问题就来了
步骤如下：
1.A、B线程同时进入了第一个if判断
2.A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();
3.然后线程A执行1-> JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance
4.在还没有进行第三步（将instance引用指向内存空间）的时候，线程A离开了synchronized块
5.线程B进入synchronized块，读取到了A线程返回的instance，此时这个instance并未进行物理地址指向，是一个空对象。

有人说将对象设置成volatile，其实也不能完全解决问题。volatile只是保证可见性，并不保证原子性。

现行的比较通用的做法就是采用静态内部类的方式来实现。

```
public class MySingleton {

//内部类
private static class MySingletonHandler{
private static MySingleton instance = new MySingleton();
}

private MySingleton(){}

public static MySingleton getInstance() {
return MySingletonHandler.instance;
}
}
```

2019-02-28

作者回复

厉害，一看就是经验丰富

2019-03-01

xx鼠 28

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。

2019-02-28

作者回复

恭喜你，学会抢答了！

2019-02-28

summer_Day 17

我觉得阿根一世的问题应该是

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

对于synchronized关键字已经对代码块进行加锁了

我理解应该等价于

```
synchronized(Singleton.class) {
if (instance == null) {
分配一块内存 M;
将 M 的地址赋值给 instance 变量;
最后在内存 M 上初始化 Singleton 对象。
}
}
```

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？

2019-02-28

落墨

16

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?

2019-02-28

作者回复

并发程序的诡异之处，就在于：我实在也想不通。

2019-02-28

别皱眉

11

周末了
对留言问题总结一下

------可见性问题------
对于可见性那个例子我们先看下定义:
可见性:一个线程对共享变量的修改，另外一个线程能够立刻看到

并发问题往往都是综合证，这里即使是单核CPU，只要出现线程切换就会有原子性问题。但老师的目的是为了让大家明白什么是可见性
或许我们可以把线程对变量的读可写都看作时原子操作，也就是cpu对变量的操作中间状态不可见，这样就能更加理解什么是可见性了。

------CPU缓存刷新到内存的时机------
cpu将缓存写入内存的时机是不确定的。除非你调用cpu相关指令强刷

------双重锁问题------
如果A线程与B线程如果同时进入第一个分支，那么这个程序就没有问题

如果A线程先获取锁并出现指令重排序时，B线程未进入第一个分支，那么就可能出现空指针问题，这里说可能出现问题是因为当把内存地址赋值给共享变量后，CPU将数据写回缓存的时机是随机的

------ synchronized------
线程在synchronized块中，发生线程切换，锁是不会释放的

------指令优化------
除了编译优化,有一部分可以通过看汇编代码来看，但是CPU和解释器在运行期也会做一部分优化，所以很多时候都是看不到的，也很难重现。

------JMM模型和物理内存、缓存等关系------
内存、cpu缓存是物理存在，jvm内存是软件存在的。
关于线程的工作内存和寄存器、cpu缓存的关系 大家可以参考这篇文章

https://blog.csdn.net/u013851082/article/details/70314778/

------IO操作------
io操作不占用cpu，读文件，是设备驱动干的事，cpu只管发命令。发完命令，就可以干别的事情了。

------寄存器切换------
寄存器是共用的，A线程切换到B线程的时候，寄存器会把操作A的相关内容会保存到内存里，切换回来的时候，会从内存把内容加载到寄存器。可以理解为每个线程有自己的寄存器

请老师帮忙看看，有没问题。希望我的总结能帮到更多人

2019-03-16

作者回复

没问题，总结的太到位了！！！

2019-03-16

我会得到 11

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

2019-02-28

作者回复

后面讲内存模型，会更有意思。

2019-02-28

波波 8

为老师点赞，讲了并发产生的前世今生，通俗易懂又不失深度。

2019-03-01

作者回复

这么夸我，我真的会骄傲的

2019-03-01

黄朋飞 8

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？

2019-02-28

作者回复

对不起，是我没说清楚，这里的缓存，指的是CPU缓存。

2019-02-28

牧童纪年 7

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？

2019-02-28

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？

2019-03-01

rayjun

5

第一个测试代码是不是有点问题，在静态方法中怎么能访问非静态变量呢？

2019-03-03

作者回复

我本地测试的代码是下面这样的，为了说明问题，为了不占用篇幅，做了删减。final Test test = new Test();使用test访问的，所以可以访问

```
public class Test {
private int count = 0;
private void add() {
int idx = 0;
while(idx++ < 10000000) {
count += 1;
}
}
public static int calc() throws Exception {
final Test test = new Test();
Thread th1 = new Thread(()->{
test.add();
});
Thread th2 = new Thread(()->{
test.add();
});

th1.start();
th2.start();
th1.join();
th2.join();
return test.count;
}

public static void main(String[] args) throws Exception {
long c =calc();
System.out.println(c);
```

}
}
2019-03-03

... ...

5

老师，上面的两个线程的例子应该不是可见性导致而是原子性导致的吧！如果是可见性导致的话，我在变量count上加个volatile应该可以解决问题啊！还发现个小问题，非静态变量应该不能直接用于静态方法中吧！

2019-02-28

作者回复

示例代码，只是为了说明问题。考虑到大家手机屏幕的尺寸，能省就省了

这个例子不仅仅是可见性的问题，并发问题往往都是综合证。

2019-03-01

发条橙子 。

5

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两个线程在不同的cpu上操作不太理解， 一个应用的AB两个线程是不是应该处在同个cpu上面 ？？？

2. 如果按照老师所说不同线程在不同cpu上运行， 是不是有个叫并行和并发的概念 。 单个cpu的时候多线程实际上是模拟并发的并行，实际上cpu只能一次执行一个线程，两个线程交替执行。 而到了多核中，可以真正的将两个线程AB同时分给cpu1 .cpu2同时执行，称之为并发？？

3. 我感觉老师第二点原子性中也有包含可见性问题，由于时间片到了， 当把资源读到自己的工作线程中时，由于不可见性，以为自己是最新的导致值不准确，这个也对应了第一个问题， 两个线程是否在同个进程内共享资源

问题有点多， 可能自己的理解有偏差，希望老师指正

2019-02-28

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。

2019-03-01

cfreedomc

5

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，药的好处不必多说，更重要的是对于药的副作用有个清晰的认识才是一个好的医生

最后从这节课后，遇到并发问题，我也可以系统的通过将问题分类到是以上哪种或者哪几种问题去解决问题

如今天的课后题目，在java中 Long类型是64位的，在32位的系统中，Cpu指令要进行多次操作，无法保证原子性

2019-02-28

小呆

4

Count那个应该是1到20000之间吧，而不是10000到20000之间吧？

2019-02-28

作者回复

什么时候会是1呢？

2019-03-01