# 一、可见性、原子性和有序性问题：并发编程Bug的源头

## ①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，但并发程序很多诡异问题的根源也在这里。

## ②源头一：缓存导致的可见性问题

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**。

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。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。

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。

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。

## ③源头二：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**”。

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。

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。

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。

## ④源头三：编译优化带来的有序性问题

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.然后 M 的地址赋值给 instance 变量。

3.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。