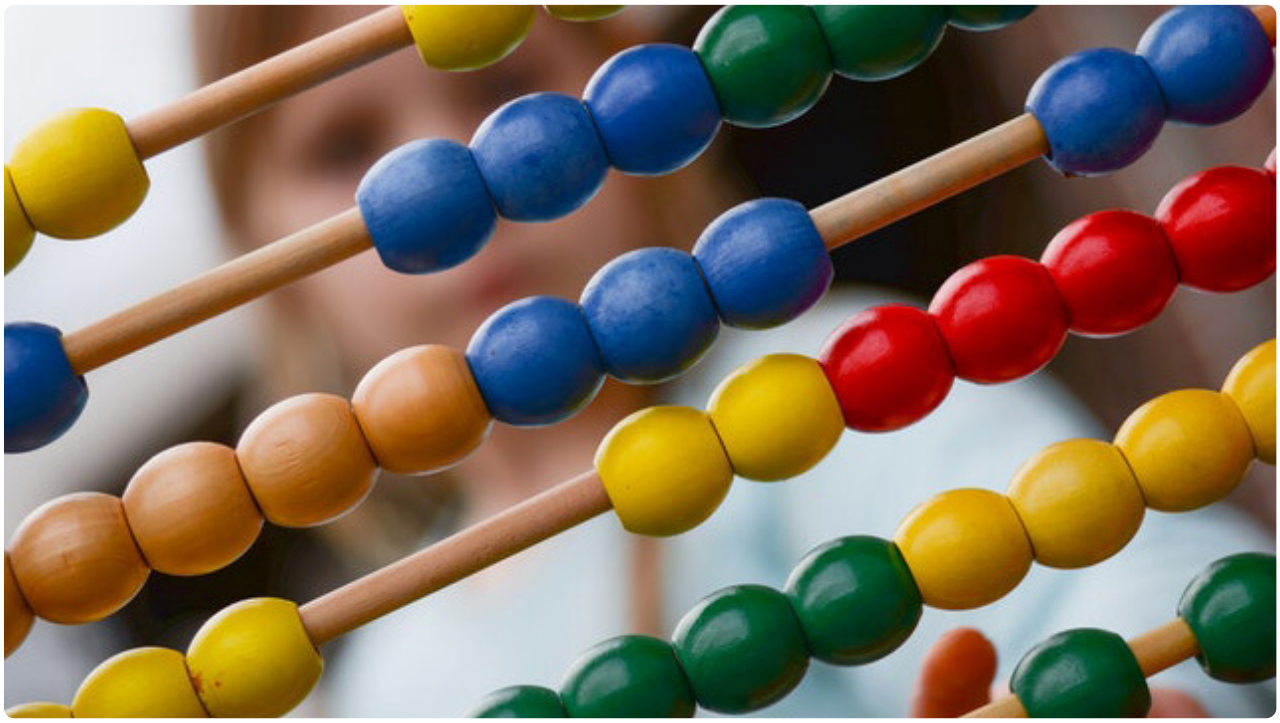
11 眼见不实—可见性

更新时间：2019-10-03 12:02:15



> 人生的价值，并不是用时间，而是用深度去衡量的。
>
> ——列夫·托尔斯泰

本节介绍并发三大特性的可见性。并发编程路上可谓困难重重。不过没有关系，道高一尺，魔高一丈。我们现在讲解的所有问题，都有能降伏住他的武器。但要想做常胜将军，那就要做到知己知彼。我们只要搞清楚有哪些问题，问题的根本原因是什么，困难才会迎刃而解。

由于我们的程序在绝大多数情况下是单线程运行的，另外即使是多线程，如果对象是无状态的，也不会有线程安全的问题。所以 JVM 更多会考虑单线程的需求。这也就造就了多线程程序在共享资源访的访问上存在问题。比如本节所讨论的可见性。

1. 什么是可见性

可见性指的是，某个线程对共享变量进行了修改，其它线程能够立刻看到修改后的最新值。乍一听这个定义，你可能会觉得这不是废话吗？变量被修改了，线程当然能够立刻读取到！否则即使单线程的程序也会出问题啊！没错，变量被修改后，在本线程中确实能够立刻被看到，但并不保证别的线程会立刻看到。原因就是编程领域经典的两大难题之一----缓存一致性。

我们看一个例子，代码如下:

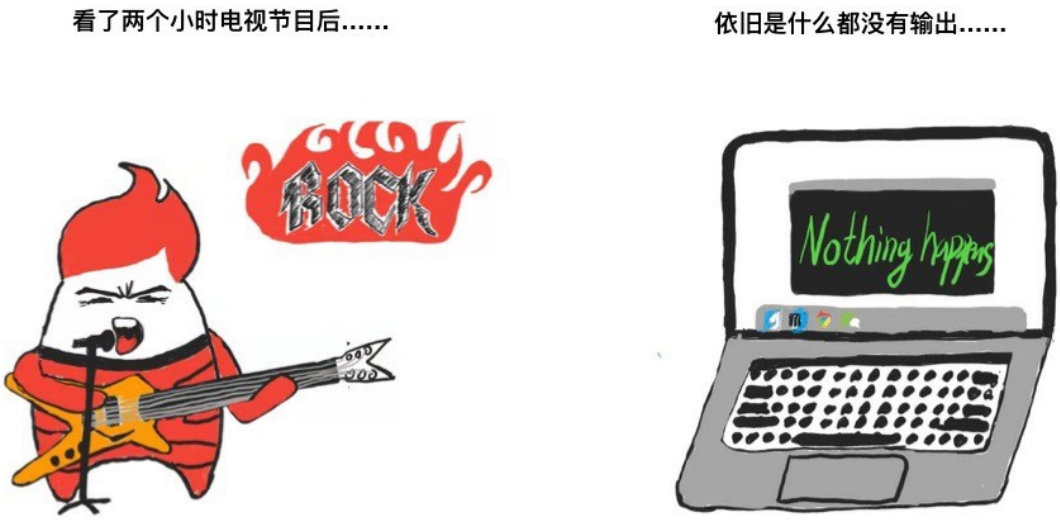
```
public class visibility {
    private static class ShowVisibility implements Runnable{
        public static Object o = new Object();
        private Boolean flag = false;
        @Override
        public void run() {
            while (true) {
                if (flag) {
                    System.out.println(Thread.currentThread().getName()+":"+flag);
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ShowVisibility showVisibility = new ShowVisibility();
        Thread blindThread = new Thread(showVisibility);
         blindThread.start();
        //给线程启动的时间
        Thread.sleep(500);
        //更新flag
        showVisibility.flag=true;
        System.out.println("flag is true, thread should print");
        Thread.sleep(1000);
        System.out.println("I have slept 1 seconds. I guess there was nothing printed ");
    }
}
```

这段代码很简单，ShowVisibility 实现 Runnable 接口，在 run 方法中判断成员变量 flag 值为 true 时进行打印。main 方法中通过 showVisibility 对象启动一个线程。主线程等待 0.5 秒后，改变 showVisibility 中 flag 的值为 true。按正常思路，此时 blindThread 应该开始打印。但是，实际情况并非如此。运行此程序，输出如下：

```
flag is true, thread should print
I have slept 1 seconds. I guess there was nothing printed
```

没错，flag 改为 true 后，blindThread 没有任何打印。也就是说 blindThread 并没有观察到到 flag 的值变化。为了测试 blindThread 到底多久能看到 flag 的变化，我决定先看会电视，可是等我刷完一集《乐队的夏天》回来，还是没有任何输出。

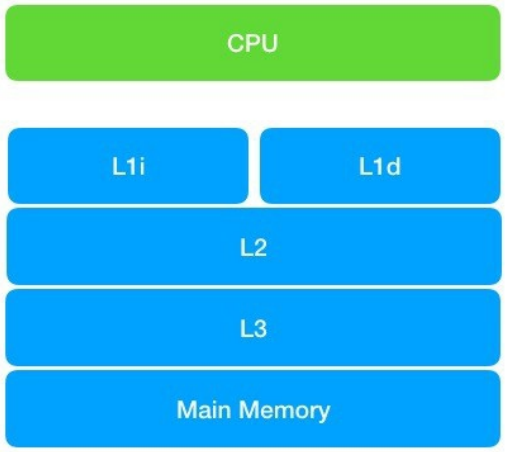


是不是很神奇？是不是很玄学？作为程序员，你一定碰到过怎么都找不出原因的 bug，最后归于玄学。其实作为代码来说，不会有什么玄学。遇到的所有问题一定有其原因。只不过有些隐藏得很深，我们很难发现。或者也可能限于自己的认知，苦苦思考也找不到答案。

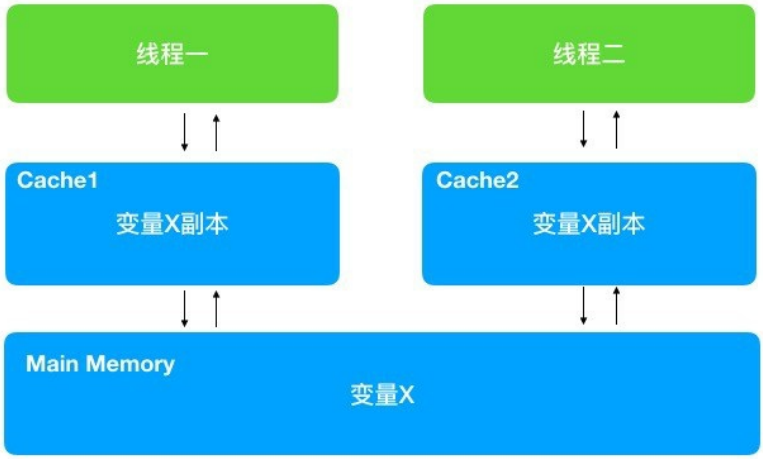
回到例子的问题本身来，执行结果完全违背我们的直觉。如果是单线程程序，做了一个变量的修改，那么程序是立即就能看到的。然而在多线程程序中并非如此。原因是 CPU 为提高计算的速度，使用了缓存。

2. CPU 缓存模型

大家一定都知道摩尔定律。根据定律，CPU 每 18 个月速度将会翻一番。CPU 的计算速度提升了，但是内存的访问速度却没有什么大幅度的提升。这就好比一个脑瓜很聪明程序员，接到需求后很快就想好程序怎么写了。但是他的电脑性能很差，每敲一行代码都要反应好久，导致完成编码的时间依旧很长。所以人再聪明没有用，瓶颈在计算机的速度上。CPU 计算也是同样的道理，瓶颈出现在对内存的访问上。没关系，我们可以使用缓存啊，这已经是路人皆知的手段了。CPU 更狠一点，用了 L1、L2、L3，一共三级缓存。其中 L1 缓存根据用途不同，还分为 L1i 和 L1d 两种缓存。如下图：



缓存的访问速度是主存的几分之一，甚至几十分之一。通过缓存，极大的提高了 CPU 计算速度。CPU 会先从主存中复制数据到缓存，CPU 在计算的时候就可以从缓存读取数据了，在计算完成后再把数据从缓存更新回主存。这样在计算期间，就无须访问主存了，速度大大提升。加上缓存后，CPU 的数据访问如下：



我们再回头看上文的例子。blindThread 线程启动后，就进入 while 循环中，一直进行运算，运算时把 flag 从主存拿到了自己线程中的缓存，此后就会一直从缓存中读取 flag 的值。即便是main线程修改了 flag 的值。但是 blindThread 线程的缓存并未更新，所以取到的还一直是之前的值。导致 blindThread 线程一致也不会有输出。

3. 最低安全性

在前面的例子中，blindThread 线程读取到flag的值是之前有效的 false。但其现在已经失效了。也就是说 blindThread 读取到了失效数据。虽然线程在未做同步的时候会读取到失效值，但是起码这个值是曾经存在过的。这称之为最低安全性。我猜你一定会问，难道线程还能读取到从来没有设置过的值吗？是的，对于 64 位类型的变量 long 和 double，JVM 会把读写操作分解为两个 32 位的操作。如果两个线程分别去读和写，那么在读的时候，可能写线程只修改了一个 32 位的数据。此时读线程会读取到原来数值一个 32 位的数值和新的数值一个 32 位的数值。两个不同数值各自的一个 32 位数值合在一起会产生一个新的数值，没有任何线程设置过的数值。这就好比马和驴各一半的基因，会生出骡子一样。此时，就违背了最低安全性。

4. 初识 volatile 关键字

要想解决可见性问题其实很简单。第一种方法就是解决一切并发问题的方法–同步。不过读和写都需要同步。

此外还有一个方法会简单很多，使用 volatile 关键字。

我们把例子中下面这行代码做一下修改。

```
private Boolean flag = false;
```

改为：

```
private volatile Boolean flag = false;
```

我们再次运行。现在程序居然可以正常输出了！是不是很简单的修改？

volatile 修饰的变量，在发生变化的时候，其它线程会立刻觉察到，然后从主存中取得更新后的值。volatile 除了简洁外，还有个好处就是它不会加锁，所以不会阻塞代码。关于 volatile 更多的知识我们后面还会做详细讲解。现在我们只要知道他能够以轻量级的方式实现同步就可以了。

5. 总结

本节我们学习了可见性。如果不了解可见性，我们写出的并发代码，可能会出现各种违背逻辑的现象。现在我们已经弄清了问题产生的原因以及如何去解决，所以可见性的问题也没什么可怕的。开发遇到问题时不要慌，所有的问题都有其产生的原因，找到原因再对症下药，保准药到病除。

开发工作中，我会遇到一些同事，遇到问题后不去分析问题产生的原因，先是自己猜测，试着乱改。发现自己不能解决后，网上搜索。找到相关帖子或文章，也不看原因是什么，直接复制粘贴代码，又是一顿试。即使这样最后解决了问题，我想对于他来说也是毫无收获的。我们不管遇到什么难题，一定不能乱了阵脚，还是从分析问题入手。最终解决问题一定是基于你分析出的原因。而不是靠猜测和盲目乱试。

}

10 有福同享，有难同当—原子性

12 什么？还有这种操作！—有序性