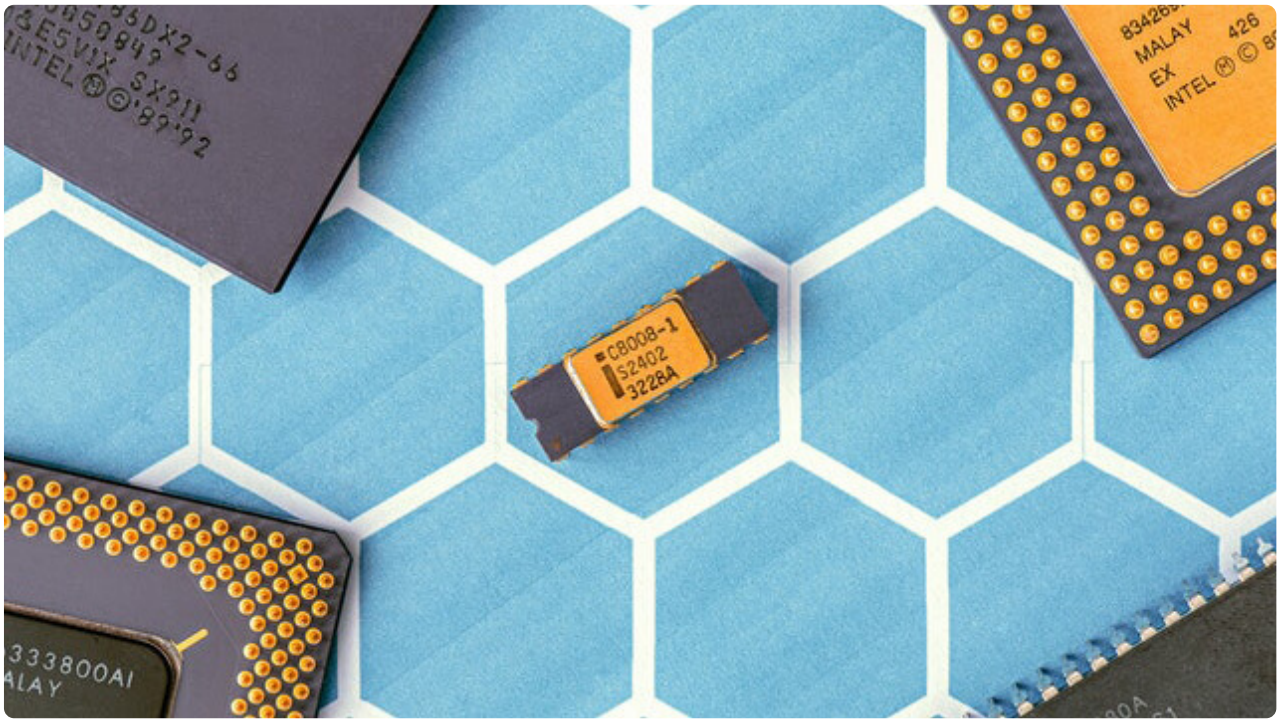
12 什么？还有这种操作！—有序性

更新时间：2019-10-11 09:58:18



> 理想必须要人们去实现它，它不但需要决心和勇敢而且需要知识。
>
> ——吴玉章

前面我们学习了原子性和可见性。相比较而言，可见性更难理解一点，但是由于缓存已经在日常编程中大量被使用，我们并不陌生，所以理解起来也没什么难度。不过本节要讲的有序性，我们之前并没有接触过相关的知识，理解起来会比较抽象。

1. 什么是有序性

有序性指的是代码在运行期间保证按照编写的顺序。这句话看起来和可见性的定义一样，好像又是一句废话。你一定在想，代码当然是按照编写顺序执行的，否则那还不乱套了？其实并不是这样，代码执行的顺序还真不一定和你编写的顺序一致。多线程开发复杂就复杂在和我们的认知相违背，我们如果在做多线程开发前不一一搞清楚，那么所编写出的并发代码一定是漏洞百出。

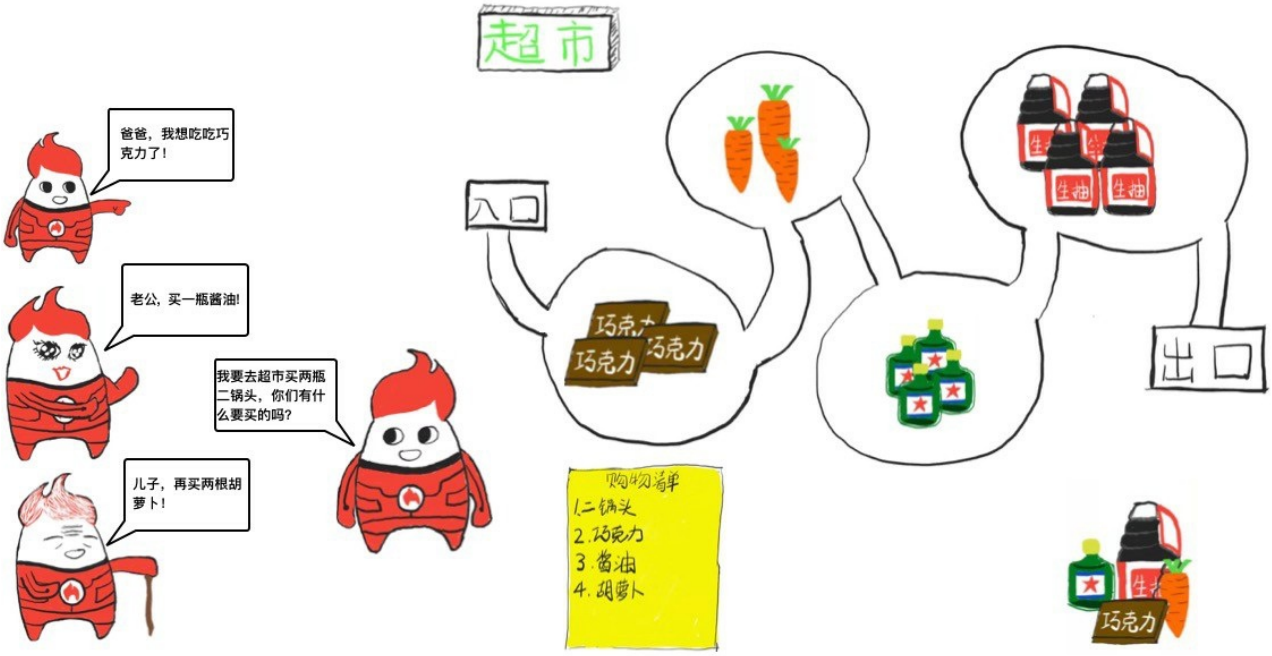
2. 指令重排序

说到有序性，我们一定会提到指令重排序。CPU 为了提高运行效率，可能会对编译后代码的指令做一些优化，这些优化不能保证 100% 符合你编写代码在正常编译后的顺序执行。但是一定能保证代码执行的结果和按照编写顺序执行的结果是一致的。

指令重排序并不是毫无约束的随意改变代码执行顺序，而是需要符合指令间的依赖关系，否则会造成程序执行结果错误。

我们接下来通过一个例子来理解指令重排序的必要性。

星期六早上，你要去超市进行采购，你自己想买两斤小龙虾，你儿子和你说要一袋巧克力，然后你老婆说家里没有酱油了买一瓶，你妈又说买两根胡萝卜。那么你到了超市会死板的按照小龙虾、巧克力、酱油、胡萝卜的顺序去采购吗？当然不会，你肯定会大致规划好路线，从离超市入口最近的货架开始采购，避免走回头路。不管你采购的顺序如何，最终你肯定会保证所有人给你的需求全部实现。CPU 也是如此，虽然是机器，但它也会规划更为合理的执行方式，确保程序运行正确的情况下，提高效率。



我们再来看一个不能重排序的例子。还是去超市采购，你妈和你说，如果买不到西葫芦，才买胡萝卜。那么买西葫芦和胡萝卜这两个步骤就不能改变。否则假如我们先去了胡萝卜货架，发现自己没买到西葫芦，就会买胡萝卜，然后又执行了买西葫芦。最后的结果就是错误的 ---- 我们既买了西葫芦也买了胡萝卜。

指令重排序的优化，仅仅对单线程程序确保安全。如果在并发的情况下，程序没能保证有序性，程序的执行结果往往会出乎我们的意料。另外注意，指令重排序，并不是代码重排序。我们的代码被编译后，一行代码可能会对应多条指令，所以指令重排序更为细粒度。

3. 单例实现遇到的有序性问题

我们在实现单例的时候，有一种方式叫做双重判断。首先判断 instance 是不是为空，如果为空进入同步代码块初始化 instance，否而直接返回 instance。初始化 instance 时再次判断 instance 是否为空，避免了在进入同步代码块这段时间有线程抢先一步完成了 instance 初始化。代码如下：

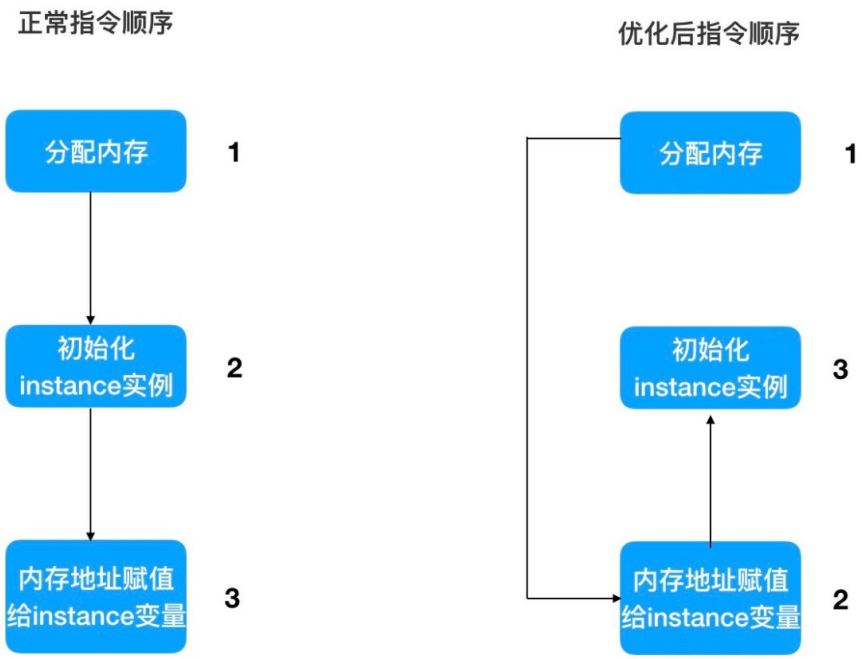
```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

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

}
```

这种单例的实现方式，看似在提高效率的同时，做到了天衣无缝。其实不然，因为 instance = new Singleton (); 这一行代码会被编译为三条指令，正常指令顺序如下：

1. 为 instance 分配一块内存 A
2. 在分配的内存 A 上初始化 instance 实例
3. 把内存 A 的地址赋值给 instance 变量

而编译器优化后可能会变成：

1. 为 instance 分配一块内存 A
2. 把内存 A 的地址赋值给 instance 变量
3. 在分配的内存 A 上初始化 instance 实例



可以看出在优化后第 2 和第 3 步调换了位置。调换后单线程运行是没有问题的。但是换做多线程，假如线程 A 正在初始化 instance，此时执行完第 2 步，正在执行第三步。而线程 B 执行到 if (instance == null) 的判断，那么线程 B 就会直接得到未初始化好的 instance，而此时线程 B 使用此 instance 显然是有问题的。

要解决本例的有序性问题很简单，我们只需要为 instance 声明时增加 volatile 关键字，volatile 修饰的变量是会保证读操作一定能读到写完的值。

总结

有序性是在多线程的情况下，确保 CPU 不对我们需要保证顺序性的代码进行重排序的。我们可以通过 sychronized 或者 volatile 来确保有序性。至此，关于多线程的三大特性已经学习完成。我们在多线程开发过程中要牢记原子性、可见性、有序性。千万不要以写单线程程序的思路来开发多线程，处理好这三大特性，多线程开发的大部分问题都会得以解决。下一节我们会来学习 Java 内存模型，其实所有的线程安全性都来自于它。

}

11 眼见不实—可见性

13 问题的根源—Java内存模型简介