# **并发问题产生的三大根源与解决sync锁**

目录

[1. 并发问题产生的三大根源 1](#_Toc2071)

[1.1. 原子性 CPU缓存会导致可见问题 指令重排序 1](#_Toc18001)

[2. 并发问题根源之一：CPU切换线程执导致的原子性问题 1](#_Toc32144)

[2.1. 加锁解决原子性 和atomcatic类型 2](#_Toc14144)

[3. 并发问题根源之二：缓存导致的可见性问题 2](#_Toc3158)

[3.1. 二、高速缓存的产生 缓存失效问题 2](#_Toc25933)

[3.2. Sync和Volide解决可见性 3](#_Toc30128)

[4. 三、指令优化 3](#_Toc18145)

[4.1. 并发问题根源之三：指令优化导致的重排序问题 4](#_Toc4017)

[5. Synchronized和volatile 对区别 4](#_Toc30155)

[5.1. 最简单方法加锁 sync 5](#_Toc24780)

## 原子性 CPU缓存会导致可见问题 指令重排序

因为CPU可以切换线程执行指令所以会导致我们的一些操作会失去原子性，因为CPU缓存会导致可见问题、还有CPU的一些指令重排序优化也会导致一些问题，关于这些我们将在并发问题的三大根源一文中继续探讨。

# 并发问题根源之一：CPU切换线程执导致的原子性问题

首先我们先理解什么叫原子性，原子性就指是把一个操作或者多个操作视为一个整体，在执行的过程不能被中断的特性叫原子性。

## 加锁解决原子性 和atomcatic类型

# 并发问题根源之二：缓存导致的可见性问题

在有了高速缓存之后，CPU的执行操作数据的过程会是这样的，CPU首先会从内存把数据拷贝到CPU缓存区。

然后CPU再对缓存里面的数据进行更新等操作，最后CPU把缓存区里面的数据更新到内存。

磁盘、内存、CPU缓存会按如下形式协作。

## **二、高速缓存的产生 缓存失效问题**

**告诉cache和多cpu cache都会导致这个问题。**

为了减少CPU等待IO的时间，让CPU有更多的时间是花在运算上，最简单的思路就是减少IO等待的时间，基于这个思路所以就有了高速缓存增加了高速缓存（L1,L2,L3,主存）。

当处理器发出内存访问请求时，会先查看高速缓存内是否有请求数据。如果存在（命中），则不需要访问内存直接返回该数据；如果不存在（失效），则要先把内存中的相应数据载入缓存，再将其返回处理器。

缓存导致的可见性问题就是指我们在操作CPU缓存过程中，由于多个CPU缓存之间独立不可见的特性，导致共享变量的操作结果无法预期。

在单核CPU时代，因为只有一个核心控制器，所以只会有一个CPU缓存区，这时各个线程访问的CPU缓存也都是同一个，在这种情况一个线程把共享变量更新到CPU缓存后另外一个线程是可以马上看见的，因为他们操作的是同一个缓存，所以他们操作后的结果不存在可见性问题。

而随着CPU的发展，CPU逐渐发展成了多核，CPU可以同时使用多个核心控制器执行线程任务，当然CPU处理同时处理线程任务的速度也越来越快了，但随之也产生了一个问题，多核CPU每个核心控制器工作的时候都会有自己独立的CPU缓存，每个核心控制器都执行任务的时候都是操作的自己的CPU缓存，CPU1与CPU2它们之间的缓存是相互不可见的。

这种情况下多个线程操作共享变量就因为缓存不可见而带来问题，多线程的情况下线程并不一定是在同一个CUP上执行，它们如果同时操作一个共享变量，但因为在不同的CPU执行所以他们只能查看和更新自己CPU缓存里的变量值，线程各自的执行结果对于别的线程来说是不可见的，所以在并发的情况下会因为这种缓存不可见的情况会导致问题出现。

## Sync和Volide解决可见性

# 三、指令优化

进程和线程本质上是增加并行的任务数量来提升CPU的利用率，缓存是通过把IO时间减少来提升CPU的利用率，而指令顺序优化的初衷的初衷就是想通过调整CPU指令的执行顺序和异步化的操作来提升CPU执行指令任务的效率。

指令顺序优化可能发生在编译、CPU指令执行、缓存优化几个阶，其优化原则就是只要能保证重排序后不影响单线程的运行结果，那么就允许指令重排序的发生。其重排序的大体逻辑就是优先把CPU比较耗时的指令放到最先执行，然后在这些指令执行的空余时间来执行其他指令，就像我们做菜的时候会把熟的最慢的菜最先开始煮，然后在这个菜熟的时间段去做其它的菜，通过这种方式减少CPU的等待，更好的利用CPU的资源。

## **并发问题根源之三：指令优化导致的重排序问题**

下面的程序代码如果init（）方法的代码经过了指令重排序后，两个方法在两个不同的线程里面调用就可能出现问题。

# Synchronized和volatile 对区别

## 最简单方法加锁 sync

volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的

volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性

volatile按照语法规范，只是表明**同一线程中对两个volatile的操作要保序**。拿volatile去解决多线程问题，其实是依赖于**“锁操作也是对volatile进行操作”这一假设**得到的取巧的结果，**如果你没加锁，光靠volatile也是错的**。

现在是不推荐拿volatile解决多线程问题的，这是个历史遗留问题。请看新一点的、更权威更正确的教材。

如果需要一个更大范围的原子性可以使用synchronized来实现，synchronized块之间的操作。

可以使用volatile保证可见性，也可以使用关键字synchronized和final。

有序性：在本线程中所有的操作都是有序的；在另一个线程中，看来所有的操作都是无序的，就可需要使用具有天然有序性的volatile保持有序性，因为其禁止重排序。

Volatile 的特性：

保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。（实现可见性）

禁止进行指令重排序。（实现有序性）

volatile 只能保证对单次读/写的原子性，i++ 这种操作不能保证原子性

Synchronized

Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：

原子性：确保线程互斥的访问同步代码；

可见性：保证共享变量的修改能够及时可见，其实是通过Java内存模型中的 “对一个变量unlock操作之前，必须要同步到主内存中；如果对一个变量进行lock操作，则将会清空工作内存中此变量的值，在执行引擎使用此变量前，需要重新从主内存中load操作或assign操作初始化变量值” 来保证的

有序性：有效解决重排序问题，即 “一个unlock操作先行发生(happen-before)于后面对同一个锁的lock操作”；

首先想到是那就在使用synchronized作用在静态方法：

虽然这样简单粗暴解决，但会导致这个方法比较效率低效，导致程序性能严重下降，那是不是还有其他更优的解决方案呢？

那么问题找到了，那怎么去解决呢？那就禁止不允许初始化阶段步骤2 、3发生重排序，刚好Volatile 禁止指令重排，从而使得双重检测真正发挥作用。

volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的

volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性

使用volatile而不是synchronized的唯一安全的情况是类中只有一个可变的域