漫画：volatile对指令重排的影响

2017-12-20 算法爱好者

（点击上方公众号，可快速关注）

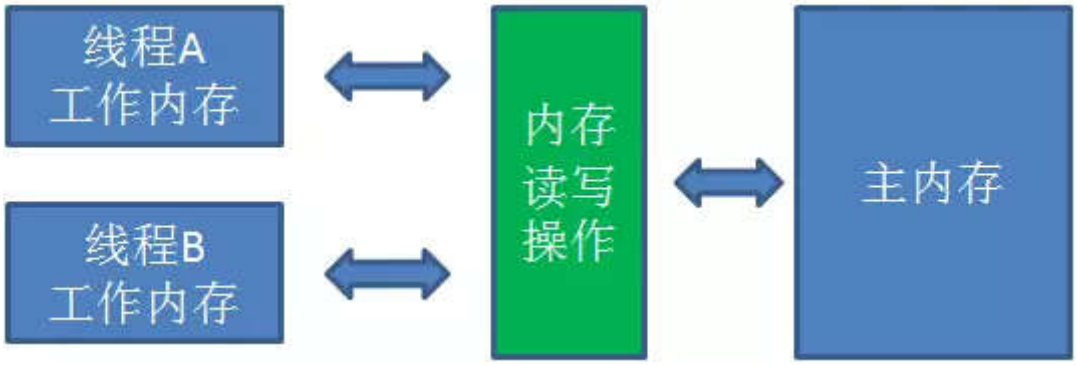
> 来源：伯乐专栏作者/玻璃猫，微信公众号 - 程序员小灰
> 好文投稿， 请点击 → 这里了解详情

上一期介绍了volatile关键字对JVM主内存和工作内存的影响，没看过的小伙伴们可以点击下面链接：

漫画：什么是 volatile 关键字？

实在懒得去看也不要紧，我们简单回顾一下：

volatile是一个轻量级的线程同步机制。它的特性之一，是保证了变量在线程之间的**可见性**。



当一个线程修改了变量的值，新的值会立刻同步到主内存当中。而其他线程读取这个变量的时候，也会从主内存中拉取最新的变量值。

但是volatile并不保证变量更新的**原子性**，在一些场景下，用volatile修饰的变量仍然不是线程安全。

下面，我们来继续今天的主题，讲一讲volatile的其他特性。

大黄，上次你介绍了 volatile 的基本知识，让我收获不小呢。

不过关于 volatile 对指令重排的影响，我还是一知半解，可以继续给我讲讲吗？

当然可以。在继续讲 volatile 之前，我们需要先知道什么是［指令重排］。

什么是指令重排？

指令重排是指JVM在编译Java代码的时候，或者CPU在执行JVM字节码的时候，对现有的指令顺序进行重新排序。

指令重排的目的是为了在不改变程序执行结果的前提下，优化程序的运行效率。需要注意的是，这里所说的不改变执行结果，指的是不改变单线程下的程序执行结果。

然而，指令重排是一把双刃剑，虽然优化了程序的执行效率，但是在某些情况下，会影响到多线程的执行结果。我们来看看下面的例子：

```
boolean contextReady = false;

在线程A中执行:
context = loadContext();
contextReady = true;

在线程B中执行:
while( ! contextReady ){
   sleep(200);
}
doAfterContextReady (context);
```

以上程序看似没有问题。线程B循环等待上下文context的加载，一旦context加载完成，contextReady == true的时候，才执行doAfterContextReady 方法。

但是，如果线程A执行的代码发生了指令重排，初始化和contextReady的赋值交换了顺序：

```
boolean contextReady = false;

在线程A中执行:
contextReady = true;
context = loadContext();
```

```
在线程B中执行:
while( ! contextReady ){
   sleep(200);
}
doAfterContextReady (context);
```

这个时候，很可能context对象还没有加载完成，变量contextReady 已经为true，线程B直接跳出了循环等待，开始执行doAfterContextReady 方法，结果自然会出现错误。

需要注意的是，这里java代码的重排只是为了简单示意，真正的指令重排是在字节码指令的层面。

什么是内存屏障？

内存屏障（Memory Barrier）是一种CPU指令，维基百科给出了如下定义：

A memory barrier, also known as a membar, memory fence or fence instruction, is a type of barrier instruction that causes a CPU or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

翻译结果如下：

内存屏障也称为内存栅栏或栅栏指令，是一种屏障指令，它使CPU或编译器对屏障指令之前和之后发出的内存操作执行一个排序约束。 这通常意味着在屏障之前发布的操作被保证在屏障之后发布的操作之前执行。

内存屏障共分为四种类型：

LoadLoad屏障：
抽象场景：Load1; LoadLoad; Load2
Load1 和 Load2 代表两条读取指令。在Load2要读取的数据被访问前，保证Load1要读取的数据被读取完毕。

StoreStore屏障：

抽象场景：Store1; StoreStore; Store2
Store1 和 Store2代表两条写入指令。在Store2写入执行前，保证Store1的写入操作对其它处理器可见

LoadStore屏障：

抽象场景：Load1; LoadStore; Store2
在Store2被写入前，保证Load1要读取的数据被读取完毕。

StoreLoad屏障：

抽象场景：Store1; StoreLoad; Load2
在Load2读取操作执行前，保证Store1的写入对所有处理器可见。StoreLoad屏障的开销是四种屏障中最大的。

volatile做了什么？

在一个变量被volatile修饰后，JVM会为我们做两件事：

1.在每个volatile写操作前插入**StoreStore**屏障，在写操作后插入**StoreLoad**屏障。
2.在每个volatile读操作前插入**LoadLoad**屏障，在读操作后插入**LoadStore**屏障。

或许这样说有些抽象，我们看一看刚才线程A代码的例子：

boolean contextReady = false;

在线程A中执行:
context = loadContext();
contextReady = true;

我们给contextReady 增加volatile修饰符，会带来什么效果呢？

```
volatile boolean contextReady = false;

在线程A中执行:
context = loadContext();
StoreStore屏障
contextReady = true;
StoreLoad屏障
```

由于加入了StoreStore屏障，屏障上方的普通写入语句 context = loadContext() 和屏障下方的volatile写入语句 contextReady = true 无法交换顺序，从而成功阻止了指令重排序。

```
volatile boolean contextReady = false;

在线程A中执行:
context = loadContext();
StoreStore屏障
contextReady = true;
StoreLoad屏障
```

原来如此，那么刚才所讲的内存屏障和之前所介绍的 Java 语言 happens-before 规则之间，是什么样的关系呢？

happens-before 是 JSR-133 规范之一，内存屏障是 CPU 指令。可以简单认为前者是最终目的，后者是实现手段。

volatile特性之一：

保证变量在线程之间的可见性。可见性的保证是基于CPU的内存屏障指令，被JSR-133抽象为happens-before原则。

volatile特性之二：

阻止编译时和运行时的指令重排。编译时JVM编译器遵循内存屏障的约束，运行时依靠CPU屏障指令来阻止重排。

几点补充：

1. 在使用volatile引入内存屏障的时候，普通读、普通写、volatile读、volatile写会排列组合出许多不同的场景。我们这里只简单列出了其中一种，有兴趣的同学可以查资料进一步学习其他阻止指令重排的场景。

2.volatile除了保证可见性和阻止指令重排，还解决了long类型和double类型数据的**8字节赋值问题**。这个特性相对简单，本文就不详细描述了。

————————END————————

漫画算法系列

- 漫画算法：最小栈的实现
- 漫画算法：判断 2 的乘方
- 漫画算法：找出缺失的整数
- 漫画算法：辗转相除法是什么鬼？
- 漫画算法：什么是动态规划？（整合版）
- 漫画算法：什么是跳跃表？
- 漫画算法：什么是 B 树？
- 漫画算法：什么是 B+ 树？
- 漫画算法：什么是一致性哈希？
- 漫画算法：无序数组排序后的最大相邻差值
- 漫画算法：什么是 Bitmap 算法？
- 漫画算法：Bitmap算法（进阶篇）
- 漫画算法：什么是布隆算法？
- 漫画算法：什么是 A* 寻路算法？
- 漫画算法：什么是 Base64 算法？
- 漫画算法：什么是 MD5 算法？
- 漫画算法：如何破解 MD5 算法？
- 漫画算法：什么是 SHA 系列算法？
- 漫画算法：什么是 AES 算法？
- 漫画算法：AES 算法的底层原理
- 漫画算法：什么是红黑树？
- 漫画算法：什么是HashMap？
- 漫画算法：高并发下的HashMap
- 漫画算法：什么是ConcurrentHashMap？
- 漫画算法：什么是单例设计模式？
- 漫画算法：如何写出更优雅的单例模式？
- 漫画算法：什么是单例模式？（整合版）
- 漫画算法：什么是 volatile 关键字？