带你一次搞明白 Java 多线程(II)

2 线程安全问题

非线程安全主要是指多个线程对同一个对象的实例变量进行操作时,会出现值被更改,值不同步的情况.

线程安全问题表现为三个方面: 原子性,可见性和有序性

2.1 原子性

原子(Atomic)就是不可分割的意思. 原子操作的不可分割有两层含义:

1) 访问(读,写)某个共享变量的操作从其他线程来看,该操作要么已经执行完毕,要么尚未发生, 即其他线程年示到当前操作的中间结果

2) 访问同一组共享变量的原子操作是不能够交错的

如现实生活中从 ATM 机取款, 对于用户来说,要么操作成功,用户拿到钱, 余额减少了,增加了一条交易记录; 要么没拿到钱,相当于取款操作没有发生

Java 有两种方式实现原子性：一种是使用锁；另一种利用处理器的 CAS(Compare and Swap)指令.

锁具有排它性,保证共享变量在某一时刻只能被一个线程访问.

CAS 指令直接在硬件(处理器和内存)层次上实现,看作是硬件锁

2.2 可见性

在多线程环境中，一个线程对某个共享变量进行更新之后，后续其他的线程可能无法立即读到这个更新的结果，这就是线程安全问题的另外一种形式：可见性(visibility).

如果一个线程对共享变量更新后，后续访问该变量的其他线程可以读到更新的结果，称这个线程对共享变量的更新对其他线程可见，否则称这个线程对共享变量的更新对其他线程不可见.

多线程程序因为可见性问题可能会导致其他线程读取到了旧数据(脏数据).

2.3 有序性

有序性(Ordering)是指在什么情况下一个处理器上运行的一个线程所执行的 内存访问操作在另外一个处理器运行的其他线程看来是乱序的(Out of Order).

乱序是指内存访问操作的顺序看起来发生了变化

2.3.1 重排序

在多核处理器的环境下,编写的顺序结构,这种操作执行的顺序可能是没有保障的:

编译器可能会改变两个操作的先后顺序;

处理器也可能不会按照目标代码的顺序执行;

这种一个处理器上执行的多个操作,在其他处理器来看它的顺序与目标代码指定的顺序可能不一样,这种现象称为重排序.

重排序是对内存访问有序操作的一种优化,可以在不影响单线程程序正确的情况下提升程序的性能.但是,可能 对多线程程序的正确性产生影响,即可能导致线程安全问题

重排序与可见性问题类似,不是必然出现的.

与内存操作顺序有关的几个概念:

源代码顺序, 就是源码中指定的内存访问顺序.

程序顺序, 处理器上运行的目标代码所指定的内存访问顺序

执行顺序,内存访问操作在处理器上的实际执行顺序

感知顺序,给定处理器所感知到的该处理器及其他处理器的内存访问操作的顺序

可以把重排序分为指令重排序与存储子系统重排序两种.

指令重排序主要是由 JIT 编译器,处理器引起的，指程序顺序与执行顺序不一样

存储子系统重排序是由高速缓存,写缓冲器引起的，感知顺序与执行顺序 不一致

2.3.2 指令重排序

在源码顺序与程序顺序不一致,或者 程序顺序与执行顺序不一致的情况下,我们就说发生了指令重排序(Instruction Reorder).

指令重排是一种动作,确实对指令的顺序做了调整，重排序的对象指令.

javac 编译器一般不会执行指令重排序，而 JIT 编译器可能执行指令重排序.

处理器也可能执行指令重排序，使得执行顺序与程序顺序不一致.

指令重排不会对单线程程序的结果正确性产生影响,可能导致多线程程序出现非预期的结果.

2.3.3 存储子系统重排序

存储子系统是指写缓冲器与高速缓存.

高速缓存(Cache)是 CPU 中为了匹配与主内存处理速度不匹配而设计的一个高速缓存

写缓冲器(Store buffer, Write buffer)用来提高写高速缓存操作的效率

即使处理器严格按照程序顺序执行两个内存访问操作,在存储子系统的作用下，其他处理器对这两个操作的感知顺序与程序顺序不一致,即这两个操作的顺序顺序看起来像是发生了变化， 这种现象称为存储子系统重排序

存储子系统重排序并没有真正的对指令执行顺序进行调整,而是造成一种指令执行顺序被调整的现象.

存储子系统重排序对象是内存操作的结果.

从处理器角度来看，读内存就是从指定的 RAM 地址中加载数据到寄存器,称为 Load 操作； 写内存就是把数据存储到指定的地址表示的 RAM 存储单元中,称为 Store 操作.内存重排序有以下四种可能:

LoadLoad 重排序,一个处理器先后执行两个读操作 L1 和 L2,其他处理器对两个内存操作的感知顺序可能是 L2->L1

StoreStore 重排序,一个处理器先后执行两个写操作 W1 和 W2,其他

处理器对两个内存操作的感知顺序可能是 W2->W1

LoadStore 重排序,一个处理器先执行读内存操作 L1 再执行写内存操作 W1，其他处理器对两个内存操作的感知顺序可能是 W1->L1

StoreLoad 重排序,一个处理器先执行写内存操作 W1 再执行读内存操作 L1，其他处理器对两个内存操作的感知顺序可能是 L1->W1

内存重排序与具体的处理器微架构有关,不同架构的处理器所允许的内存重排序不同

内存重排序可能会导致线程安全问题.假设有两个共享变量 int data = 0;　boolean　ready = false;

处理器 1	处理器 2
data = 1;　　//S1 ready = true;　　//S2	
	while(!ready){}　　//L3 sout(data);　　//L4

2.3.4 貌似串行语义

JIT 编译器,处理器,存储子系统是按照一定的规则对指令,内存操作的结果进行重排序，给单线程程序造成一种假象----指令是按照源码的顺序执行的.这种假象称为貌似串行语义．并不能保证多线程环境程序的正确性

为了保证貌似串行语义,有数据依赖关系的语句不会被重排序,只有不存在数据依赖关系的语句才会被重排序.如果两个操作(指令)访问同一个变量,且其中一个操作(指令)为写操作,那么这两个操作之间就存在数据依赖关系(Data dependency).

如:

x = 1; y = x + 1; 后一条语句的操作数包含前一条语句的执行结果;

y = x; x = 1; 先读取 x 变量,再更新 x 变量的值;

x = 1; x = 2; 两条语句同时对一个变量进行写操作

如果不存在数据依赖关系则可能重排序,如:

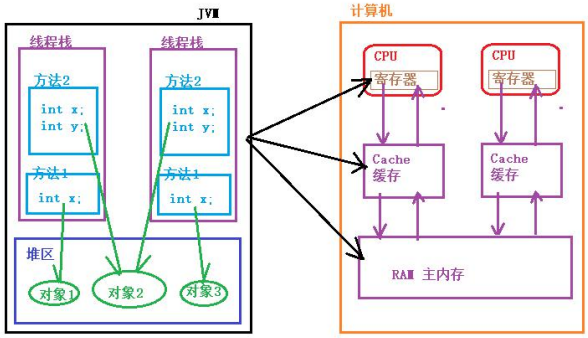
```
double  price = 45.8;
int quantity = 10;
double  sum = price * quantity;
```

存在控制依赖关系的语句允许重排.一条语句(指令)的执行结果会决定另一条语句(指令)能否被执行,这两条语句(指令)存在控制依赖关系(Control Dependency). 如在 if 语句中允许重排,可能存在处理器先执行 if 代码块,再判断 if 条件是否成立

2.3.5 保证内存访问的顺序性

可以使用 volatile 关键字, synchronized 关键字实现有序性

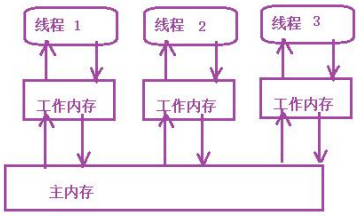
2.4 Java 内存模型



1)每个线程都有独立的栈空间
2)每个线程都可以访问堆内存
3)计算机的CPU不直接从主内存中读取数据，CPU读取数据时,先把主内存的数据读到Cache缓存中，把Cache中 的数据读到Register寄存器中
4)JVM中的共享的数据可能会被分配到Register寄存器中，每个CPU都有自己的Register寄存器，一个CPU不能读取其他CPU上寄存器中 的内容.如果两个线程分别运行在不同的处理器(CPU)上,而这个共享的数据被分配到寄存器上,会产生可见性问题.
5)即使JVM中的共享数据分配到主内存中,也不能保证数据的可见性.CPU不直接对主内存访问,而是通过Cache高速缓存进行的. 一个处理器上运行的线程对数据的更新可能只是更新到处理器的写缓冲器(Store Buffer)，还没有到达Cache缓存,更不用说主内存了.另外一个处理器不能读取到该处理器写缓冲器上的内容，会产生运行在另外一个处理器上的线程无法看到该处理器对共享数据的更新.

6)一个处理器的Cache不能直接读取另外一个处理器的Cache.但是一个处理器可以通过缓存一致性协议(Cache Coherence Protocol)来读取其他处理器缓存中的数据，并将读取的数据更新到该处理器的Cache中.这个过程称为缓存同步. 缓存同步使得一个处理器上运行的线程可以读取到另外一个处理器上运行的线程对共享数据的所做的更新,即保障了可见性.为了保障可见性,必须使一个处理器对共享数据的更新最终被写入该处理器的Cache,这个过程称为冲刷处理器缓存.

可以把Java内存模型抽象为：



规定：
每个线程之间的共享数据都存储在主内存中
每个线程都有一个私有的本地内存(工作内存)，线程的工作内存是抽象的概念,不是真实存在的,它涵盖写缓冲器,寄存器,其他硬件的优化.
每个线程从主内存中把数据读取到本地工作内存中,在工作内存中保存共享数据的副本
线程在自己的工作内存中处理数据，仅对当前线程可见，对其他线程是不可见的