Volatile禁止指令重排

1、指令重排有序性:

计算机在执行程序时,为了提高性能,编译器和处理器常常会做指令重排,一般分为以下三种:



单线程环境里面确保程序最终执行结果和代码顺序执行结果一致。

处理器在进行指令重排序时必须考虑指令之间的*数据依赖性*

多线程环境中线程交替执行,由于编译器指令重排的存在,两个线程使用的变量能否保证一致性是无法确认的,结果无法预测。

指令重排案例分析one:

指令重排案例分析two:

x = 2 y = 1	https://blog.csdr.net/longgegiaojie90
x = a;	y = b;
b = 1;	a = 2;
线程1	线程2
如果编译器对这段程序代码执行重排优化后	,可能出現下列情况
x = 0 y = 0	
b = 1;	a = 2;
x = a;	y = b;
线程1	线程2

指令重排案例分析three:

```
public class BanCommandReSortSeq {
   int a = 0;
   boolean flag = false;
   public void methodOne() {
      a = 1;
      // 语句1
      flag = true;
      // 语句2
      // methodOne发生指令重排,程序执行顺序可能如下:
      // flag = true;
      // 语句2
      // a = 1;
     // 语句1
   public void methodTwo() {
     if (flag) {
         a = a + 5;
         // 语句3
      System.out.println("methodTwo ret a = " + a);
   }
   // 多线程环境中线程交替执行,由于编译器指令重排的存在,两个线程使用的变量能否保证一致性是无
法确认的,结果无法预测。
  // 多线程交替调用会出现如下场景:
   // 线程1调用methodOne,如果此时编译器进行指令重排
  // methodOne代码执行顺序变为: 语句2(flag=true) -> 语句1(a=5)
   // 线程2调用methodTwo,由于flag=true,如果此时语句1还没有执行(语句2 -> 语句3 -> 语句
1 ),那么执行语句3的时候a的初始值=0
```

2、禁止指令重排底层原理:

volatile实现禁止指令重排优化,从而避免多线程环境下程序出现乱序执行的现象。

先了解下概念,内存屏障(Memory Barrier)又称内存栅栏,是一个CPU指令,它的作用有两个:

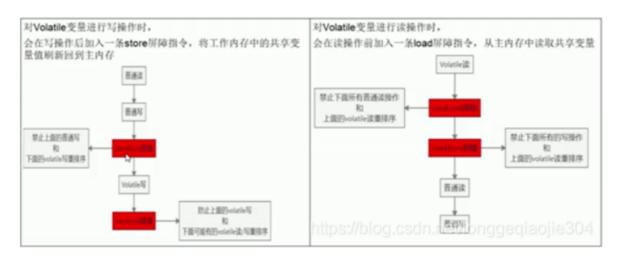
- 保证特定操作执行的顺序性;
- 保证某些变量的内存可见性 (利用该特性实现volatile内存可见性)

volatile实现禁止指令重排优化底层原理:

由于编译器和处理器都能执行指令重排优化。如果在指令间插入一条Memory Barrier则会告诉编译器和CPU,不管什么指令都不能和这条Memory Barrier指令重排,也就是说通过插入内存屏障,就能禁止在内存屏障前后的指令执行重排优化。内存屏障另外一个作用就是强制刷出各种CPU的缓存数据,因此任何CPU上的线程都能读取到这些数据的最新版本。

左边: 写操作场景: 先LoadStore指令, 后LoadLoad指令。

右边:读操作场景:先LoadLoad指令,后LoadStore指令。



3、volatile使用场景

单例模式(DCL-Double Check Lock双端检锁机制)

如果此时你也把volatile禁止指令重排底层原理也解释清楚了,面试官可能会接着问你,你知道volatile 使用场景吗?

单例模式(DCL-Double Check Lock双端检锁机制)就是它的使用场景

认准一手QQ3195303913微信wxywd8