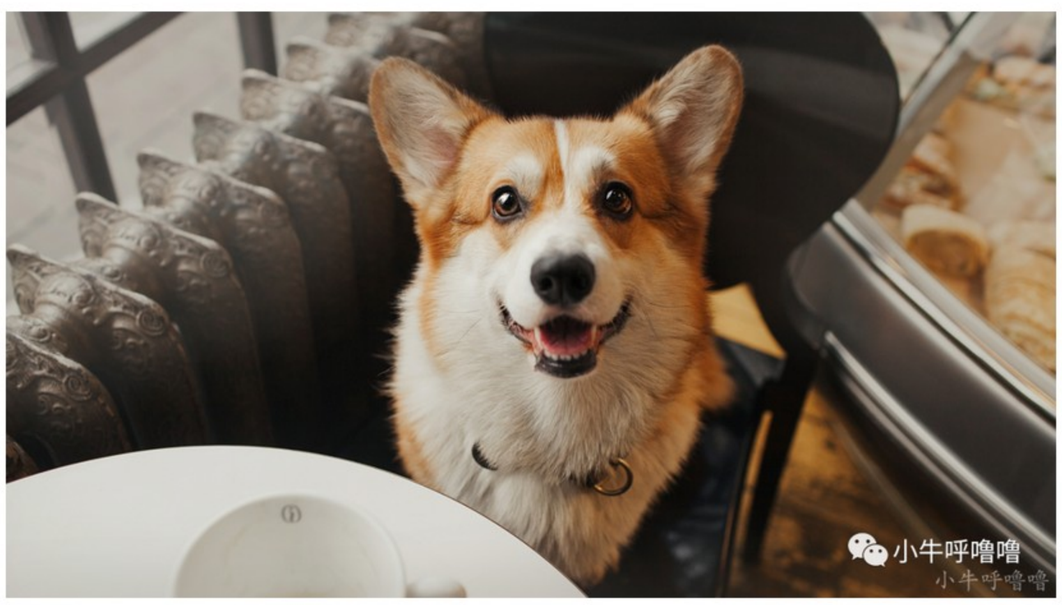
volatile关键字在并发中有哪些作用?

原创 小牛呼噜噜 小牛呼噜噜 2022-10-21 00:08

收录于合集

#Java 进阶 14 #Java 21



前言

大家好，读过呼噜噜之前的一篇文章https://mp.weixin.qq.com/s/iyXN4WyGAV_J4d8zeiG7yA的小伙伴,知道了由于计算机为了充分利用CPU的高性能，以及各个硬件 存取速度巨大的差异带来的一系列问题

1. 为了充分压榨CPU的性能，**CPU 会对指令乱序执行或者语言的编译器会指令重排**，让CPU一直工作不停歇，但同时会导致有序性问题。
2. 为了平衡CPU的寄存器和内存的速度差异，计算机的CPU 增加了高速缓存，但同时导致了可见性问题
3. 为了平衡CPU 与 I/O 设备的速度差异，操作系统增加了进程、线程概念，以分时复用CPU，但同时导致了原子性问题。

Java 是最早尝试提供内存模型的编程语言。由于Java 语言是跨平台的，另外各个操作系统总存在一些差异，Java在物理机器的基础上抽象出一个**内存模型(JMM)**，来简化和管理并发程序。我们都知道Java并发的三大特性：`原子性，可见性，有序性`

- 原子性指的是一个不可以被分割的操作，即这个操作在执行过程中不能被中断，要么全部不执行，要么全部执行。且一旦开始执行，不会被其他线程打断。
- 可见性 指的是一个线程修改了共享变量后，其他线程能立即感知这个变量被修改。
- 有序性 指程序按照代码的先后顺序执行。在Java内存模型中，为了提升效率是允许编译器和处理器对指令进行重排序，当然重排序不会影响单线程的运行结果，但是对多线程会有影响

那么本文我们就聊聊**关键字volatile**，`可能是 Java 中最微妙和最难用的关键字`,看看其在Java内存模型中是如何保证并发操作的原子性、可见性、有序性的?

> 读本文前，可参考JMMhttps://mp.weixin.qq.com/s/iyXN4WyGAV_J4d8zeiG7yA里的每张图，先理解JMM内存模型

什么是volatile关键字

volatile是Java中用于修饰变量的关键字，其**可以保证该变量的可见性以及顺序性，但是无法保证原子性。更准确地说是**`volatile关键字`**只能保证单操作的原子性**，比如`x=1`**，但是无法保证复合操作的原子性**，比如`x++`

其为Java提供了一种轻量级的同步机制：`保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知`。相比于`synchronized关键字`（synchronized通常称为重量级锁），volatile更轻量级，开销低，因为它不会引起`线程上下文的切换和调度`。

保证可见性

可见性：是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看到修改的值。我们一起来看一个例子：

```
public class VisibilityTest {
    private boolean flag = true;

    public void change() {
        flag = false;
        System.out.println(Thread.currentThread().getName() + "，已修改flag=false");
    }

    public void load() {
        System.out.println(Thread.currentThread().getName() + "，开始执行.....");
        int i = 0;
        while (flag) {
            i++;
        }
        System.out.println(Thread.currentThread().getName() + ",结束循环");
    }

    public static void main(String[] args) throws InterruptedException {
        VisibilityTest test = new VisibilityTest();

        // 线程threadA模拟数据加载场景
        Thread threadA = new Thread(() -> test.load(), "threadA");
        threadA.start();

        // 让threadA执行一会儿
        Thread.sleep(1000);
        // 线程threadB 修改 共享变量flag
        Thread threadB = new Thread(() -> test.change(), "threadB");
        threadB.start();

    }
}
```

其中：threadA 负责循环，threadB负责修改 共享变量flag，如果flag=false时，threadA 会结束循环，但是上面的例子会死循环！原因是`threadA无法立即读取到共享变量flag修改后的值`。我们只需 `private volatile boolean flag = true;`，加上volatile关键字threadA就可以立即退出循环了。

其中Java中的volatile关键字提供了一个功能：`那就是被volatile修饰的变量P被修改后，JMM会把该线程本地内存中的这个变量P，立即强制刷新到主内存中去，导致其他线程中的volatile变量P缓存无效，也就是说其他线程使用volatile变量P在时，都是从主内存刷新的最新数据`。而**普通变量的值在线程间传递的时候一般是通过主内存以共享内存的方式实现的；**

因此，可以使用`volatile`来保证多线程操作时变量的可见性。除了`volatile`，`Java`中的`synchronized`和`final`两个关键字 以及各种 `Lock`也可以实现可见性。加锁的话， 当一个线程进入 `synchronized`代码块后，线程获取到锁，会清空本地内存，然后从主内存中拷贝共享变量的最新值到本地内存作为副本，执行代码，又将修改后的副本值刷新到主内存中，最后线程释放锁。

保证有序性

有序性，顾名思义即程序执行的顺序按照代码的先后顺序执行。但现代的计算机中CPU中为了能够让指令的执行尽可能地同时运行起来，提示计算机性能，采用了**指令流水线**。一个 CPU 指令的执行过程可以分成 4 个阶段：`取指、译码、执行、写回`。这 4 个阶段分别由 4 个独立物理执行单元来完成。

理想的情况是：指令之间无依赖，可以使流水线的并行度最大化 但是如果两条指令的前后存在依赖关系，比如数据依赖，控制依赖等，此时后一条语句就必需等到前一条指令完成后，才能开始。所以CPU为了提高流水线的运行效率，对无依赖的前后指令做适当的乱序和调度，即现代的计算机中CPU是**乱序执行**指令的

另一方面，只要不会改变程序的运行结果，Java编译器是可以通过**指令重排**来优化性能。然而，重排可能会影响本地处理器缓存与主内存交互的方式，可能导致在多线程的情况下发生"细微"的BUG。

指令重排一般可以分为如下三种类型：

- 编译器优化重排序，编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
- 指令级并行重排序，现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
- 内存系统重排序，由于处理器使用缓存和读 / 写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。这并不是显式的将指令进行重排序，只是因为缓存的原因，让指令的执行看起来像乱序。

从 Java 源代码到最终执行的指令序列，一般会经历下面三种重排序：

变量初始化赋值

我们一起来看一个例子，让大家体悟 `volatile关键字` 的**禁止指令重排**的作用：

```
int i = 0;
int j = 0;
int k = 0;
i = 10;
j = 1;
```

对于上面的代码我们正常的执行流程是：

- 初始化i
- 初始化j
- 初始化k
- i赋值
- j赋值

但由于指令重排序问题，代码的执行顺序未必就是编写代码时候的顺序。语句可能的执行顺序如下：

- 初始化i
- i赋值
- 初始化j
- j赋值
- 初始化k

指令重排对于非原子性的操作，在不影响最终结果的情况下，其拆分成的原子操作可能会被重新排列执行顺序,提升性能。**指令重排不会影响单线程的执行结果，但是会影响多线程并发执行的结果正确性**。但当我们用 `volatile` 修饰变量k时：

```
int i = 0;
int j = 0;
volatile int k = 0;
i = 10;
j = 1;
```

这样会保证上面代码执行顺序：变量i和j的初始化，在 `volatile int k = 0` 之前，变量i和j的赋值操作在 `volatile int k = 0` 后面

懒汉式单例 -- 双重校验锁 volatile版

我们可以使用 `volatile关键字` 去阻止重排 `volatile变量` 周围的读写指令，这种操作通常称为 *memory barrier*（内存屏障），详情可见：**https://mp.weixin.qq.com/s/TyiCfVMeeDwa-2hd9N9XJQ** 中 `懒汉式单例 -- 双重校验锁 volatile版`

隐藏特性

volatile关键字除了禁止指令重排的作用，还有一个特性：当线程向一个 `volatile 变量` 写入时，在线程写入之前的其他所有变量（包括 `非volatile变量` ）也会刷新到主内存。

当线程读取一个 `volatile变量` 时，它也会读取其他所有变量（包括 `非volatile变量` ）与 `volatile变量` 一起刷新到主内存。尽管这是一个重要的特性，但是我们不应该过于依赖这个特性，来"自动"使周围的变量变得 `volatile` ，若是我们想让一个变量是 `volatile` 的，我们编写程序的时候需要非常明确地用 `volatile关键字` 来修饰。

无法保证原子性

volatile关键字无法保证原子性，更准确地说是 `volatile关键字` **只能保证单操作的原子性，**比如 `x=1` **，但是无法保证复合操作的原子性，**比如 `x++`

所谓原子性：即一个或者多个操作作为一个整体，要么全部执行，要么都不执行，并且操作在执行过程中不会被线程调度机制打断；而且这种操作一旦开始，就一直运行到结束，中间不会有任何上下文切换（context switch）

```
int  = 0;   //语句1，单操作,原子性的操作

i++;        //语句2，复合操作，非原子性的操作
```

其中：语句2 `i++` 其实在Java中执行过程，可以分为3步：

1. i 被从局部变量表（内存）取出，
2. 压入操作栈（寄存器），操作栈中自增
3. 使用栈顶值更新局部变量表（寄存器更新写入内存）

执行上述3个步骤的时候是可以进行线程切换的，或者说是可以被另其他线程的 这3 步打断的，因此 `语句2` 不是一个原子性操作

volatile版 i++

我们再来看一个例子：

```
public class Test1 {

    public static volatile int val;

    public static void add() {
        for (int i = 0; i < 1000; i++) {
            val++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(Test1::add);
        Thread t2 = new Thread(Test1::add);
        t1.start();
        t2.start();
        t1.join();//等待该线程终止
        t2.join();
        System.out.println(val);
    }
}
```

2个线程各循环2000次，每次 `+1` ，如果 `volatile关键字` 能够保证原子性，预期的结果是 `2000` ，但实际结果却是： `1127` ，而且多次执行的结果都不一样，可以发现 `volatile关键字` 无法保证原子性。

synchronized版 i++

我们可以利用 `synchronized关键字` 来解决上面的问题：

```
public class SynchronizedTest {
    public static int val;

    public synchronized static void add() {
        for (int i = 0; i < 1000; i++) {
            val++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(SynchronizedTest::add);
        Thread t2 = new Thread(SynchronizedTest::add);
        t1.start();
        t2.start();
        t1.join();//等待该线程终止
        t2.join();
        System.out.println(val);
    }
}
```

运行结果： `2000`

Lock版 i++

我们还可以通过加锁来解决上述问题：

```
public class LockTest {

    public static int val;

    static Lock lock = new ReentrantLock();

    public static void add() {

        for (int i = 0; i < 1000; i++) {

            lock.lock();//上锁
            try {
                val++;
            }catch(Exception e) {
                e.printStackTrace();
            }finally {
                lock.unlock();//解锁
            }

        }

    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(LockTest::add);
        Thread t2 = new Thread(LockTest::add);
        t1.start();
        t2.start();
        t1.join();//等待该线程终止
        t2.join();
        System.out.println(val);
    }

}
```

运行结果：`2000`

Atomic版 i++

Java从JDK 1.5开始提供了`java.util.concurrent.atomic`包（以下简称Atomic包），这个包中的原子操作类, 靠`CAS循环`的方式来保证其原子性，是一种用法简单、性能高效、线程安全地更新一个变量的方式。

这些类可以保证多线程环境下，当某个线程在执行atomic的方法时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择一个线程执行。

我们来用`atomic`包来解决volatile原子性的问题：

```
public class AtomicTest {
    public static AtomicInteger val = new AtomicInteger();

    public static void add() {
        for (int i = 0; i < 1000; i++) {
            val.getAndIncrement();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(AtomicTest::add);
        Thread t2 = new Thread(AtomicTest::add);
        t1.start();
        t2.start();
        t1.join();//等待该线程终止
        t2.join();
        System.out.println(val);
    }
}
```

运行结果：`2000`, 如果我们维护现有的项目，如果遇到volatile变量最好将其替换为Atomic 变量,除非你真的特别了解volatile。Atomic 就不展开说了，先挖个坑，以后补上

volatile 原理

当大家仔细读完上文的`懒汉式单例 -- 双重校验锁 volatile版`，会发现`volatile关键字`修饰变量后，我们反汇编后会发现 多出了`lock`前缀指令，lock前缀指令在汇编中 LOCK指令前缀功能如下：

- 被修饰的汇编指令成为"原子的"
- 与被修饰的汇编指令一起提供"内存屏障"效果（`lock指令`可不是内存屏障）

内存屏障主要分类：

1. 一类是可以**强制读取主内存，强制刷新主内存的内存屏障**，叫做`Load屏障`和`Store屏障`
2. 另一类是**禁止指令重排序的内存屏障**，主要有四个分别叫做`LoadLoad屏障`、`StoreStore屏障`、`LoadStore屏障`、StoreLoad屏障

这4个屏障具体作用：

- LoadLoad屏障：（指令Load1; LoadLoad; Load2），在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
- LoadStore屏障：（指令Load1; LoadStore; Store2），在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
- StoreStore屏障：（指令Store1; StoreStore; Store2），在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
- StoreLoad屏障：（指令Store1; StoreLoad; Load2），在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能

对于volatile操作而言，其操作步骤如下：

- 每个volatile写入之前，插入一个 **StoreStore**，写入以后插入一个 **StoreLoad**
- 每个volatile读取之前，插入一个 **LoadLoad**，读取之后插入一个**LoadStore**

我们再总结以下，用`volatile关键字`修饰变量后，主要发生的变化有哪些？：

1. 当一个线程修改了 volatile 修饰的变量，当修改后的变量写回主内存时，其他线程能立即看到最新值。即volatile关键字保证了并发的可见性

   > - 使用volatile关键字修饰共享变量后，每个线程要操作该变量时会从主内存中将变量拷贝到本地内存作为副本，但当线程操作完变量副本，会强制将修改的值立即写入主内存中。
   > - 然后通过 **CPU总线嗅探机制**告知其他线程中该变量副本**全部失效**，（在CPU层，一个处理器的缓存回写到内存会导致其他处理器的缓存行无效），若其他线程需要该变量，必须重新从主内存中读取。

2. 在x86的架构中，volatile关键字 底层 含有lock前缀的指令，与被修饰的汇编指令一起提供"内存屏障"效果，禁止了指令重排序，保证了并发的有序性

   > 确保一些特定操作执行的顺序,让cpu必须按照顺序执行指令，即当指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

3. volatile关键字无法保证原子性，**更准确地说是**`volatile关键字`**只能保证单操作的原子性**，比如`x=1`，**但是无法保证复合操作的原子性**，比如`x++`。

   > 有人可能问赋值操作是原子操作，本来就是原子性的，用volatile修饰有什么意义？在Java数据类型足够大的情况下（在 Java 中 **long** 和 **double** 类型都是 64 位），写入变量的过程分两步进行，就会发生 **Word tearing（字分裂）**情况。JVM 被允许将64位数量的读写作为两个单独的32位操作执行，这增加了在读写过程中发生上下文切换的可能性，多线程的情况下可能会出现值会被破坏的情况
   >
   > 在缺乏任何其他保护的情况下，用 **volatile** 修饰符定义一个 **long** 或 **double** 变量，可阻止字分裂情况

参考资料：

《On Java 8》

《Java并发编程》

《深入理解JVM虚拟机》

http://ifeve.com/jmm-cookbook-reorderings

http://ifeve.com/jmm-cookbook-mb

本篇文章到这里就结束啦，很感谢你能看到最后，如果觉得文章对你有帮助，别忘记关注我！

收录于合集 #Java 进阶 14

‹ 上一篇
Java内存模型(JMM)详解

下一篇 ›
Synchronized关键字详解

阅读原文

分享 收藏

小牛呼噜噜

89篇原创内容

关注