16 让你眼见为实—volatile详解

更新时间：2019-10-22 10:05:02

> 人要有毅力，否则将一事无成。
>
> ——居里夫人

上一节我们讲解了 Atomic 变量。Atomic 以更为轻量的方式实现原子性。不过也存在其局限性，只能应用于特定的场景。本节我们将讲解的 volatile 关键字，则是用来解决可见性、有序性问题。上一章讲解可见性问题时，已经简单提到过 volatile 关键字。被 volatile 关键字修饰的变量，会确保值的变化被其它线程所感知，从而从主存中取得该变量最新的值。此外，在 happans-before 原则中有一条 volatile 变量原则，阐述了 vlatile 如何确保有序性。

1. volatile 效果

我们先通过之前的例子来回顾下 volatile 的作用，例子很简单，主线程试图通过修改 flag 的值，来触发 visableThread 线程打印自己线程 name。代码如下：

```
private static class ShowVisibility implements Runnable{
    public static Object o = new Object();
    private volatile Boolean flag = false;
    @Override
    public void run() {
      while (true) {
        if (flag) {
          System.out.println(Thread.currentThread().getName()+":"+flag);
        }
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    ShowVisibility showVisibility = new ShowVisibility();
    Thread visableThread = new Thread(showVisibility);
     visableThread.start();
    //给线程启动的时间
    Thread.sleep(500);
    //更新flay
    showVisibility.flag=true;
    System.out.println("flag is true, thread should print");
    Thread.sleep(1000);
    System.out.println("I have slept 1 seconds. Is there anything printed ?");
  }
}
```

代码中使用 volatile 修饰 flag 变量。这确保在多个线程并发时，任何一个线程改变了 flag 的值都会立即被其它线程所看到。以上程序 main 线程修改了 flag 值后，visableThread 能够立即打印出自己的线程 name。但如果我们把 flag 前的 volatile 去掉，可以看到 main 线程修改了 flag 值后，visableThread 也不会有任何输出。也就是说 visableThread 并不知道 flag 值已经被修改。

原因在之前文章中也已经分析过，为了提高计算效率，CPU 会从缓存中取得 flag 值。但是主存中 flag 值的变化，visableThread 线程并不知道，导致其缓存和主存不一致，获取到的是失效的 flag 值。

2. 理解 volatile

volatile 关键字可以用来修饰实例变量和类变量。被 volatile 修饰后，该变量或获得以下特性：

1. 可见性。任何线程对其修改，其它线程马上就能读到最新值；
2. 有序性。禁止指令重排序。

有序性 **可见性**

之前章节我们讲解过，CPU 为了提升速度，采用了缓存，因此造成了多个线程缓存不一致的问题，这也是可见性的根源。为了解决缓存一致性，我们需要了解缓存一致性协议。MESI 协议是目前主流的缓存一致性协议。此协议会保证，写操作发生时，线程独占该变量的缓存，CPU 并且会通知其它线程对于该变量所在的缓存段失效。只有在独占操纵完成之后，该线程才能修改此变量。而此时由于其它缓存全部失效，所以就不存在缓存一致性问题。而其它线程的读取操作，需要等写入操作完成，恢复到共享状态。

volatile 是如何做到以上机制的呢？我们可以看一下对 volatile 修饰变量的赋值操作，编译成指令后的代码：

```
mov    0x20(%rsp),%rsi

mov    %rax,%r10

shr    $0x3,%r10

mov    %r10d,0xc(%rsi)

shr    $0x9,%rsi

movabs $0x7f55dd1cb000,%rdi

movb   $0x0,(%rsi,%rdi,1)

lock addl $0x0,(%rsp)
```

我们不需要理解以上指令，只需要关注最后一行。可以看到最后一行使用了 lock 关键字。lock 的作用是在其有效的范围内锁住总线，从而执行该行代码线程所在的处理器能够独占资源。由于总线被锁定，开销很大的。所以新的 CPU 实现已经不会锁住总线，而是锁定变量所在的缓存区域，就像上文描述的 MESI 协议，从而保证了数据的可见性。

volatile 的有序性则是通过内存屏障。所谓的内存屏障就是在屏障前的所有指令可以重排序的，屏障之后的指令也可以重排序，但是重排序的时候不能越过内存屏障。也就是说内存屏障前的指令不会被重排序到内存屏障之后，反之亦然。

volatile 能够保证变量的可见性和有序性，但是并不能保证原子性。比如我们用 volatile 修饰了变量 i，多线程并发执行 i++。假如有 10 个线程，每个线程执行 1 万次 i++，那么最后 i 的结果肯定不是 10 万。因为 i++ 实际为三步操作：

1. 从主存取得 i 的值，存入缓存；

2. 为 i 加 1；
3. 赋给 i，写入主存。

这三步在没有原子性保证时多线程并发，就会导致不同线程同时执行了步骤 1，读取到了一样的 n 值，从而造成了重复的 +1 操作。多次 i++ 操作但只为 i 增加了 1。从试验结果可以明显的看出 volatile 并不会保证原子性。

3. volatile 的使用场景

了解 volatile 原理之后，我们总结一下 volatile 的特性和局限性。

volatile 能为我们提供如下特性：

1. 确保实例变量和类变量的可见性；
2. 确保 volatile 变量前后代码的重排序以 volatile 变量为界限。

volatile 的局限性：

1. volatile 的可见性和有序性只能作用于单一变量；
2. volatile 不能确保原子性；
3. volatile 不能作用于方法，只能修饰实例或者类变量。

volatile 的以上特点，决定了它的使用场景是有限的，并不能完全取代 synchronized 同步方式。一般使用 volatile 的场景是代码中通过某个状态值 flag 做判断，flag 可能被多个线程修改。如果不使用 volatile 修饰，那么 flag 不能保证最新的值被每个线程读取到。而在使用 volatile 修饰后，任何线程对 flag 的修改，都立刻对其它线程可见。此外其它线程看到 flag 变化时，所有对 flag 操作前的代码都已生效，这是 volatile 的有序性确保的。

正是由于 volatile 有如上局限性，所以我们只能在上述场景或者其它适合的场景使用 volatile。反推 volatile 不适用的场景如下：

1. 一个变量或者多个变量的原子性操作；
2. 不以 volatile 变量操作作为分界线的有序性保证。

volatile 无法解决的问题最终还得通过 sychronized 或者其它加锁方式来确保同步。

4. 总结

本节我们深入讲解了 volatile 关键字。不但学习了 volatile 的特性和原理，并了解了 volatile 的局限性。我们在开发中最常用的用法，是使用 volatile 修饰作为标识判断的变量。以确保任何线程对它的修改都能立即被其它线程看到，从而正确触发判断逻辑。volatile 可以在特定的场景下高效解决并发问题。不过由于自身的局限性，很多时候还是需要依靠 synchronized 或它加锁方式来实现同步。下一节我们就来看看一直被提及的 synchronized 关键字如何使用及其原理。

}