再有人问你volatile是什么，就把这篇文章发给他

原创 Hollis Hollis 2018-08-13

收录于话题
#和并发编程有关的那点事儿

13个

在再有人问你Java内存模型是什么，就把这篇文章发给他中我们曾经介绍过，Java语言为了解决并发编程中存在的原子性、可见性和有序性问题，提供了一系列和并发处理相关的关键字，比如synchronized、volatile、final、concurren包等。在前一篇文章中，我们也介绍了synchronized的用法及原理。本文，来分析一下另外一个关键字——volatile。

本文就围绕 volatile 展开，主要介绍 volatile 的用法、 volatile 的原理，以及 volatile 是如何提供可见性和有序性保障的等。

volatile 这个关键字，不仅仅在Java语言中有，在很多语言中都有的，而且其用法和语义也都是不尽相同的。尤其在C语言、C++以及Java中，都有 volatile 关键字。都可以用来声明变量或者对象。下面简单来介绍一下Java语言中的 volatile 关键字。

volatile的用法

volatile 通常被比喻成"轻量级的 synchronized "，也是Java并发编程中比较重要的一个关键字。和 synchronized 不同， volatile 是一个变量修饰符，只能用来修饰变量。无法修饰方法及代码块等。

volatile 的用法比较简单，只需要在声明一个可能被多线程同时访问的变量时，使用 volatile 修饰就可以了。

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
```

```
        }
        return singleton;
        }
    }
```

如以上代码，是一个比较典型的使用双重锁校验的形式实现单例的，其中使用 `volatile` 关键字修饰可能被多个线程同时访问到的singleton。

volatile的原理

在再有人问你Java内存模型是什么，就把这篇文章发给他中我们曾经介绍过，为了提高处理器的执行速度，在处理器和内存之间增加了多级缓存来提升。但是由于引入了多级缓存，就存在缓存数据不一致问题。

但是，对于 `volatile` 变量，当对 `volatile` 变量进行写操作的时候，JVM会向处理器发送一条lock前缀的指令，将这个缓存中的变量回写到系统主存中。

但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现 缓存一致性协议

缓存一致性协议：每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

所以，如果一个变量被 `volatile` 所修饰的话，在每次数据变化之后，其值都会被强制刷入主存。而其他处理器的缓存由于遵守了缓存一致性协议，也会把这个变量的值从主存加载到自己的缓存中。这就保证了一个 `volatile` 在并发编程中，其值在多个缓存中是可见的。

volatile与可见性

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得到修改的值。

我们在再有人问你Java内存模型是什么，就把这篇文章发给他中分析过：Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也

无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。所以，就可能出现线程1改了某个变量的值，但是线程2不可见的情况。

前面的关于 `volatile` 的原理中介绍过了，Java中的 `volatile` 关键字提供了一个功能，那就是被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。因此，可以使用 `volatile` 来保证多线程操作时变量的可见性。

volatile与有序性

有序性即程序执行的顺序按照代码的先后顺序执行。

我们在再有人问你Java内存模型是什么，就把这篇文章发给他中分析过：除了引入了时间片以外，由于处理器优化和指令重排等，CPU还可能对输入代码进行乱序执行，比如 `load->add->save` 有可能被优化成 `load->save->add` 。这就是可能存在有序性问题。

而 `volatile` 除了可以保证数据的可见性之外，还有一个强大的功能，那就是他可以禁止指令重排优化等。

普通的变量仅仅会保证在该方法的执行过程中所依赖的赋值结果的地方都能获得正确的结果，而不能保证变量的赋值操作的顺序与程序代码中的执行顺序一致。

volatile可以禁止指令重排，这就保证了代码的程序会严格按照代码的先后顺序执行。这就保证了有序性。被 `volatile` 修饰的变量的操作，会严格按照代码顺序执行， `load->add->save` 的执行顺序就是：load、add、save。

volatile与原子性

原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。

我们在Java的并发编程中的多线程问题到底是怎么回事儿中分析过：线程是CPU调度的基本单位。CPU有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去CPU使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。

在上一篇文章中，我们介绍 synchronized 的时候，提到过，为了保证原子性，需要通过字节码指令 monitorenter 和 monitorexit ，但是 volatile 和这两个指令之间是没有任何关系的。

所以， volatile 是不能保证原子性的。

在以下两个场景中可以使用 volatile 来代替 synchronized ：

> 1、运算结果并不依赖变量的当前值，或者能够确保只有单一的线程会修改变量的值。
> 2、变量不需要与其他状态变量共同参与不变约束。

除以上场景外，都需要使用其他方式来保证原子性，如 synchronized 或者 concurrent包 。

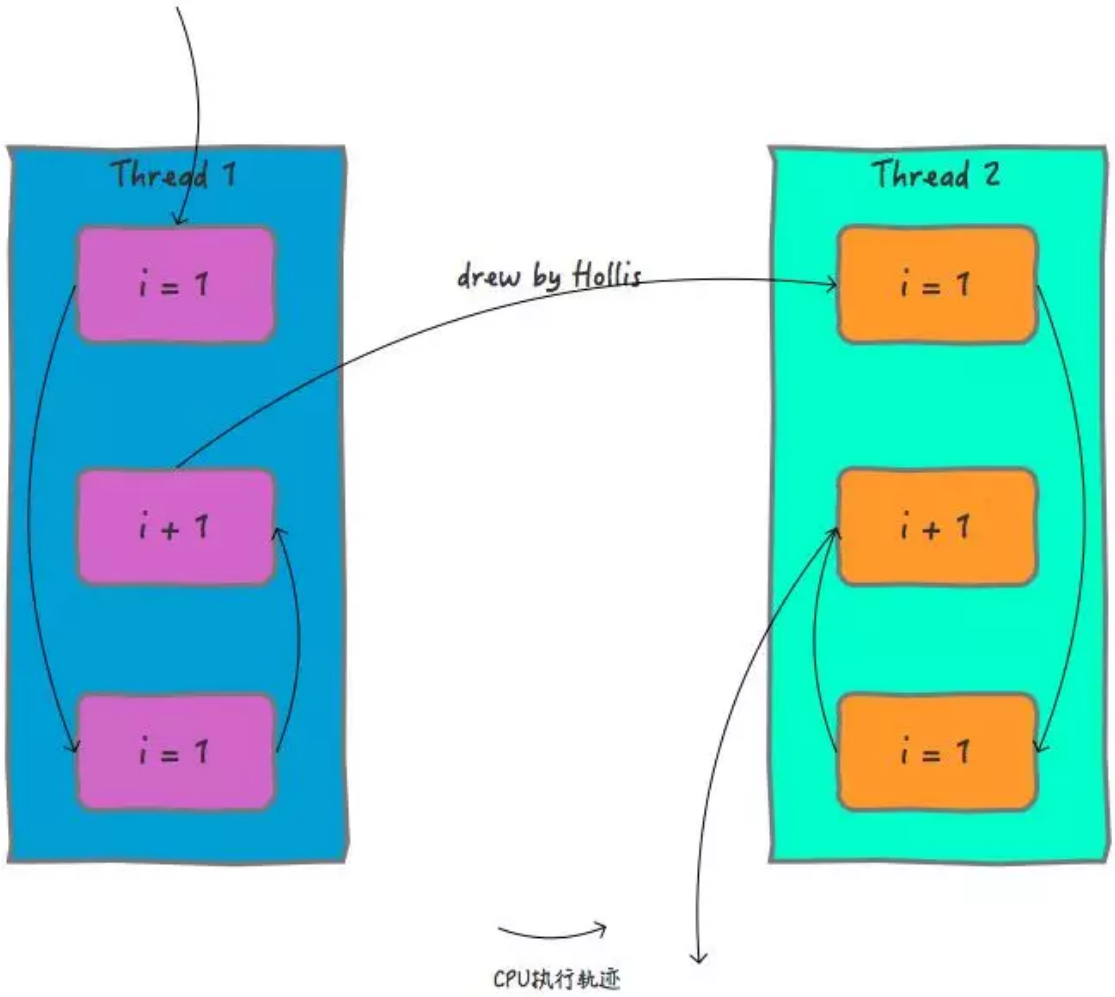
我们来看一下volatile和原子性的例子：

```
public class Test {
    public volatile int i = 0;

    public void increase() {
        i++;
    }

    public static void main(String[] args) {
        final Test test = new Test();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }

        while(Thread.activeCount()>1)  //保证前面的线程都执行完
            Thread.yield();
        System.out.println(test.i);
    }
}
```

以上代码比较简单，就是创建10个线程，然后分别执行1000次 i++ 操作。正常情况下，程序的输出结果应该是10000，但是，多次执行的结果都小于10000。这其实就是 volatile 无法满足原子性的原因。

为什么会出现这种情况呢，那就是因为虽然volatile可以保证 `i` 在多个线程之间的可见性。但是无法保证 `i++` 的原子性。

`i++` 操作，一共有三个步骤： `load i` ， `add i` ，`save i` 。在多线程场景中，如果这三个步骤无法按照顺序执行的话，那么就会出现问题。



如上图，两个线程同时执行 `i++` 操作，如果允许指令重排，我们期望的结果是3，但是实际执行结果可能是2，甚至可能是1。

总结与思考

我们介绍过了 `volatile` 关键字和 `synchronized` 关键字。现在我们知道，`synchronized` 可以保证原子性、有序性和可见性。而 `volatile` 却只能保证有序性和可见性。

那么，我们再来看一下双重校验锁实现的单例，已经使用了 `synchronized` ，为什么还需要 `volatile` ？

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
}
```

答案，我们在下一篇文章：既生synchronized，何生volatile中介绍，敬请期待。

欢迎你给出你的思考。

- MORE | 更多精彩文章 -

- **TCP 三次握手原理，你真的理解吗?**
- **推荐一个很好的公众号：漫话编程**
- **如何给女朋友解释什么是分布式和集群**
- **深入理解Java中的synchronized原理**

如果你看到了这里，说明你喜欢本文。

那么请长按二维码，关注Hollis

转发朋友圈，是对我最大的支持。

喜欢此内容的人还喜欢

【对线面试官】Spring基础

面试造火箭

国内外大厂oncall情况大比拼！谷歌员工竟然抢着oncall?

程序员八卦

回家后的一些感悟

阿猫读书