优秀引用

尚硅谷JUC并发编程（对标阿里P6-P7）之volatile

Java中不可或缺的关键字「volatile」

全面理解Java的内存模型（JMM）

1、概述

在多线程编程中，确保线程安全和正确的执行顺序是非常重要的。由于多线程环境下，不同线程之间共享内存资源，因此对这些资源的访问必须进行同步以避免出现竞态条件等问题。Java中提供了多种方式来实现同步，其中 `volatile` 是一种非常轻量级的同步机制。

`volatile` 直译过来是“**不稳定的**”，意味着被其修饰的属性可能随时发生变化。该关键字为Java提供了一个轻量级的同步机制：**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**。相较于我们熟知的重量级锁 `synchronized`，`volatile` 更轻量级，因为它**不会引起上下文切换和线程调度**。

`volatile`关键字的特性主要有以下几点：

1. **保证可见性**：当一个变量被声明为 `volatile` 时，所有线程都可以看到它的最新值，即每次读取都是从**主内存**中获取最新值，而不是从线程的本地缓存中获取旧值；
2. **保证有序性**： `volatile` 关键字可以禁止指令重排序。编译器和CPU为了提高代码执行效率，可能会对指令进行**重排序**，这可能会导致线程安全问题。但是，当一个变量被声明为 `volatile` 时，编译器和CPU会禁止对它进行指令重排序，保证指令执行的正确顺序；
3. **无法保证原子性**： `volatile` 关键字并不能保证操作过程中的有序性，如果需要保证一系列操作的原子性，仍然需要借助锁机制进行限制。

2、可见性保证

2.1、什么是可见性

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，其他县城能够立即看到修改的值。

2.2、例子举证

我们通过一个循环的例子进行举证，大致是使用一个变量标识一个 `while` 循环，通过新线程修改这个标识，进而查看循环是否会结束。接下来将会对未加上和加上 `volatile` 进行举例查看结果。

1. 未加 `volatile` 的普通flag。

```
public class VolatileSeeTest {
    static boolean flag = true;

    public static void main(String[] args) throws
InterruptedException {
        new Thread(() -> {
            int i = 1;
            while (flag) {
                if (i == 1) {
                    System.out.println("掉坑里了……");
                    i = 0;
                }
            }
            System.out.println("我出来啦！.(flag此时为" + flag);
        }, "t1").start();

        // 等待确保上面t1线程已经执行
        Thread.sleep(1000L);

        flag = false;
        System.out.println("好小子，速速跳出坑来.(flag此时为" + flag);
```

```
    }
}

================================================

此时结果打印：
掉坑里了……
好小子，速速跳出坑来.(flag此时为false)
(程序还未结束，代表t1线程还在死循环中)

================================================
```

2. 加上 `volatile` 的flag。

```
public class VolatileSeeTest {
    static volatile boolean flag = true;

    public static void main(String[] args) throws
InterruptedException {
        new Thread(() -> {
            int i = 1;
            while (flag) {
                if (i == 1) {
                    System.out.println("掉坑里了……");
                    i = 0;
                }
            }
            System.out.println("我出来啦！.(flag此时为" + flag);
        }, "t1").start();

        // 等待确保上面t1线程已经执行
        Thread.sleep(1000L);

        flag = false;
        System.out.println("好小子，速速跳出坑来.(flag此时为" + flag);
    }
}

================================================

掉坑里了……
好小子，速速跳出坑来.(flag此时为false)
我出来啦！.(flag此时为false)

Process finished with exit code 0(程序已经结束)

================================================
```

2.3、结果解析

针对于第一种没有 `volatile` 关键字修饰的情况，很明显 **主线程** 对 `flag` 变量的修改对 `t1` 线程并不可见，导致 `t1` 线程中的循环并未跳出。这是因为 **主线程** 和 `t1` 线程中分别都对 `flag` 变量进行了拷贝，备份到了各自中的本地缓存(也叫做工作内存或本地内存)中，当两个线程读取 `flag` 变量时都是从本地缓存中读取，**主线程** 中对 `flag` 变量进行的操作对 `t1` 线程并不可见，导致每次 `t1` 线程读取 `flag` 变量时都是初始保存的 `false`。

> 🌈
>
> 根本原因是因为没有 `volatile` 关键字修饰的变量并没有及时的从主存中读取最新值和往主存中写入自己修改的值，如果其他线程要访问这个变量，它们可能会直接从自己的本地缓存中读取这个变量的值，而不是从主内存中读取，导致在多线程环境下不同线程之间的数据出现不一致情况。

针对于第二种添加了 `volatile` 关键字修饰的情况，通过结果我们可以看出 `t1` 线程成功跳出了循环最终程序结束，证明了 `volatile` 关键字是可以保证可见性的。这是因为被 `volatile` 修饰的 `flag` 变量被修改后，`JMM` 会把该线程本地缓存中的这个 `flag` 变量**立即强制刷新到主内存**中去，导致 `t1` 线程中的 `flag` 变量缓存无效，也就是说其他线程使用 `volatile` 修饰的 `flag` 变量时，都是从主内存刷新的最新数据。

3、有序性保证

3.1、什么是有序性

所谓的有序性，顾名思义就是程序执行的顺序按照指定的顺序先后执行。

3.2、什么是重排序

现代的计算机为了提高性能，在程序运行过程中常常会对指令进行重排序，这就涉及到了为此诞生的 **流水线技术**。

所谓的 **流水线技术**，就是指一个CPU指令的执行过程可以分为4个阶段：取指、译码、执行、写回。它的原理是在不影响程序运行结果的情况下，指令1还没有执行完，就可以开始执行指令2，而不用等到指令1执行结束之后再执行指令2，这样就大大提高了效率。

但是在多线程的情况下，指令重排可能会影响本地缓存和主存之间交互的方式，造成乱序问题最终导致数据错乱。指令重排一般可以分为下面三种类型：

- **编译器优化重排**。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。
- **指令并行重排**。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**(即后一个执行的语句无需依赖前面执行的语句的结果)，处理器可以改变语句对应的机器指令的执行顺序。
- **内存系统重排**。由于处理器使用缓存和读写缓存冲区，这使得加载(load)和存储(store)操作看上去可能是在乱序执行，因为三级缓存的存在，导致内存与缓存的数据同步存在时

间差。

3.3、例子举证

了解过单例模式的小伙伴可能都了解过，双重校验锁有一个volatile版本的：

```
public class Singleton {

    // 私有构造方法
    private Singleton() {}
    // 使用volatile禁止单例对象创建时的重排序
    private static volatile Singleton instance;

   // 对外提供静态方法获取该对象
    public static Singleton getInstance() {
        // 第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
        if(instance == null) {
            synchronized (Singleton.class) {
                // 抢到锁之后再次判断是否为空
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

有小伙伴可能会问到不是已经上了锁并且都进行判断了嘛，怎么还会有并发问题，还得加上 `volatile` 关键字解决的。这就得扯到在多线程环境下对 `instant` 对象实例化时计算机对其的指令重排了：

当一个线程执行到第一次判空时，由于 `instant` 还没有被初始化，因此会进入同步块中进行初始化操作。但是，在初始化过程中，由于指令重排序的影响， `instant` 可能会被先分配空间并赋值，然后再进行构造函数的初始化操作。此时，如果有另外一个线程进入了第一次判空，并且发现 `instant` 不为 `null`，就会直接返回一个尚未完成初始化的实例，从而导致并发问题。

4、无法保证原子性

4.1、什么是原子性

原子性是指一个操作或者一系列操作要么全部执行成功，要么全部不执行，不会出现部分执行的情况。

4.2、例子举证

常见的非原子性操作便是自增操作，因为自增操作在指令层面可以分为三步：

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）。

我们对 **volatile** 修饰的变量进行自增操作，通过查看结果来验证这一特性：

```
public class VolatileAtomicTest {

    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();
        // 等待线程运算结束
        t1.join();
        t2.join();
        // 打印结果
        System.out.println(val);
    }
}
```

按照正常情况，最终输出的应该是2000，但是我们运行起来会发现结果并不如意，绝大多数情况下都会低于2000，从而验证了 **volatile** 并不能保证原子性。这是因为多线程环境下，可能 **线程t1** 正在进行 `第i次` 的 **取值-运算-赋值** 操作时，另外一个 **线程t2** 已经完成了操作并提交到了主存中，主存就会通知 **线程t1** 本地缓存中的数据已经过时，从而丢弃手中正在进行的对数据的操作，去获取最新的数据，导致 **线程t1** 要开始 `第i+1次` 运算从而浪费了 `第i次` 的运算机会，导致最终的结果没有达到我们预想的2000。

> 🌈
>
> 原子性的保证可以通过 synchronized、Lock、Atomic

5、内存屏障

5.1、什么是内存屏障

内存屏障，也称内存栅栏，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，是的词典之前的所有读写操作都执行后才可以开始执行词典之后的操作，避免代码的重排序。

内存屏障其实就是一种JVM指令，Java内存模型的重排规则会**要求Java编译器在生成JVM指令时插入特定的内存屏障指令**，通过这些内存屏障指令，volatile实现了Java内存模型中的可见性和有序性。

> 🌈
>
> 通过对有 volatile 关键字修饰的变量进行操作的代码进行反编译我们会发现，在 volatile 范围内多了个**lock前缀指令**，这里简单介绍一下这一指令的作用。

当一个变量被volatile修饰后，它在读写时会使用一种特殊的机器指令（lock前缀指令），这个指令可以保证多个线程在读写这个变量时不会出现问题：

1. 写volatile变量时，会先把变量的值写入到CPU缓存中，然后再把缓存中的数据写入到主内存中，这样其他线程就能看到最新的值了。
2. 读volatile变量时，会从主内存中读取最新的值，而不是从CPU缓存中读取，这样就能保证不会拿到过期的值了。

此外，由于lock前缀指令会对指定的内存区域加锁，保证了对该变量的**读写操作**的原子性，避免了出现竞态条件。

5.2、不同内存屏障的作用

对于内存屏障的分类其实分有两种，其中一种常见的便是对内存屏障的粗分：

- **读屏障**：用于确保在读取共享变量之前，先要读取该变量之前的所有操作的结果；
- **写屏障**：用于确保在写入共享变量之后，后续的所有操作都不能被重排序到写操作之前。

细分之下，内存屏障又分为四种：

- **LoadLoad屏障**：
    - 保证在读取共享变量之前，先要读取该变量之前的所有操作的结果。
    - 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**：
    - 保证在读取共享变量之前，先要读取该变量之前的所有操作的结果，并且在写入共享变量之后，后续的所有操作都不能被重排序到写操作之前。
    - 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**：
    - 保证在写入共享变量之后，后续的所有写操作都不能被重排序到写操作之前。
    - 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**：
    - 保证在写入共享变量之后，后续的所有读操作都不能被重排序到写操作之前。
    - 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**

6、volatile和synchronized的区别

volatile和synchronized都可以保证多线程之间的可见性和原子性，但是它们之间有以下几点不同：

1. volatile只能保证可见性和有序性，不能保证原子性。而synchronized既可以保证可见性和有序性，也可以保证原子性。
2. volatile不会阻塞线程，而synchronized会阻塞线程。
3. volatile只能修饰变量，而synchronized可以修饰方法和代码块。
4. volatile只能保证单次读/写的原子性，不能保证多次读/写的原子性。而synchronized可以保证多次读/写的原子性。

	可见性保证	原子性保证	有序性保证	阻塞线程	可修饰对象	多次操作原子性
volatile(轻量)	✓	✗	✗	✗	变量	✗
synchronized(重量)	✓	✓	✓	✓	方法、代码块	✓

7、使用场景

7.1、多线程共享变量

在多线程环境下，多个线程可能同时访问同一个变量。如果这个变量没有被声明为 `volatile`，那么每个线程都会从自己的缓存中读取这个变量的值，而不是从主内存中读取。这样就可能会出现一个线程修改了变量的值，但是其他线程并没有及时得到变量的更新，导致程序出现错误。

使用 `volatile` 声明变量可以保证每个线程都从主内存中读取变量的值，而不是从自己的缓存中读取。这样就可以保证多个线程访问同一个变量时的可见性和正确性。

7.2、双重检查锁定

双重检查锁定(Double-checked locking)是一种延迟初始化的技术，常用于单例模式的实现。在双重检查锁定模式中，首先检查是否已经实例化，如果没有实例化，则进行同步代码块，再次检查是否已经实例化，如果没有则进行实例化。

但是在没有使用volatile修饰共享变量的情况下，可能会出现线程安全问题。因为在实例化对象时，可能会出现指令重排的情况，导致其他线程在检查对象是否为null时，得到的是一个尚未完全初始化的对象。

使用volatile声明共享变量可以禁止指令重排，从而保证双重检查锁定模式的正确性。

7.3、状态标志

当一个变量被用于表示某个状态时，例如线程是否终止、是否可以执行某项操作等，需要使用volatile来保证操作的可见性和正确性。

在多线程环境下，一个线程修改了状态变量的值，其他线程需要及时得到变量的更新，以保证程序的正确性。