853769620

1、面试题

java的内存模型是什么？能结合内存模型说一下volatile的工作原理吗？指令重排序，内存栅栏，happen-before等概念是指的什么意思？

2、面试官心里分析

我作为面试官的话，也是经常问volatile的，确实是，但是确实比较凸显一个人的java基本功，而且一般聊到并发编程这块，一般都会问一下这个问题。说实话，我遇到过的人里，能把volatile讲清楚的很少很少；你如果去网上查文章和博客，能把volatile看明白的，我觉得你很厉害，文章或者是博客，几乎我没见过一个真正讲明白的，写的很好的。

这个volatile，说实话吧，不推荐大家大量使用，为什么呢？因为volatile背后的原理还是比较高级的，如果你用了，有两种情况会发生：第一种，你很牛，你明白volatile啥意思，用了，没问题；第二种，你就略微懂一点点，结果还瞎用volatile，就导致出问题。

但是无论是哪种情况用了volatile，都有一个很大的问题，对于你研发普通的业务系统来说，要明白一个道理，铁打的硬盘，流水的兵，可能以后你都走了，结果某个人来接替你，他可不一定熟悉volatile这个东西，那可能他维护的时候就会出问题。

所以就我个人经验来说，我不推荐在业务系统里用volatile，你可以找别的方式来保证并发安全性，但是如果是那种底层框架或者分布式系统研发团队，里面个个都是牛人，走了一个，进来的也是比较厉害的，这种精英团队，可以用volatile。

面试的时候，volatile还是很高频的会问到的，所以大家知道肯定要知道，自己要结合内存模型能回答一下，但是你说用，但是还是少用。

3、面试题

**3.1 操作系统内存模型**

先聊下内存模型这个事儿。

操作系统这个层面上，所有指令都是cpu里执行的，但是执行指令的时候肯定是要读写数据的，数据一般都放主存里，但是直接读写内存速度不够快，所以一般会在cpu里放高速缓存，就是说主存的数据会放一份在cpu内部的高速缓存中，这样cpu指令直接对cpu内部的高速缓存里的数据读写就可以了。

举个例子，i++

但是这样就会带来数据一致性的问题，因为每次cpu都是从内存读取数据到cpu高速缓存，操作完以后，再写回主存里。比如i++，这个操作吧，一般就是从主存读取i（i = 1）到cpu高速缓存里，然后对这个值累加，i = 2，先写入高速缓存，接着写回主存。

但是如果是多线程，可不是这么玩儿的，多线程的时候，在多cpu场景下，可能两个线程会将主存的i = 1都读到高速缓存里，然后都累加，i = 2，接着写回自己的高速缓存，然后刷回主存，此时就会导致主存里的数据是i = 2，而不是我们期望的i = 3。

可以用总线lock锁机制，这个上一讲还提到过，但是这个机制其实很重，会导致只有一个cpu可以操作主存，别的cpu都没法操作了。所以最新的cpu里，都不会用总线锁了，一般都会用优化的缓存一致性协议（MSI），就是某个cpu写数据的时候，发现写的是共享变量，会通知其他cpu这个数据在他们内部的高速缓存是无效的，让其他cpu读这个变量的时候从写数据的那个cpu缓存里读取，保证大家的缓存是一致的。当然，也有一种实现，是说修改变量的线程将数据从自己缓存立即刷新到内存，然后其他cpu发现自己的缓存行失效，如果要读写数据的时候重新从内存里加载到缓存里来。

并发问题里，有3个至关重要的点，原子性、可见性、有序性

**（1）原子性**

这个很简单，就是说比如i++这个操作吧，根据操作系统内存模型，是需要先从主存中读取数据到cpu高速缓存中，然后给它加1后写入高速缓存，最后再从高速缓存刷入主存，这个过程是有很多个步骤的。如果这个过程能保证绝对可以执行成功，不会出现任何意外，那么就是原子性。

你可以保证并发操作的原子性，无论多少个线程怎么i++，最后都是不断累加1的，数据不会错，就是原子性的

**（2）可见性**

//线程1

int i = 1;

i = 5; -> 还放在高速缓存里，还没写入主存

//线程2

int j = i; -> 从主存加载i的值，还是1，将i = 1的值，赋值给了j，j = 1

比如上面的代码，如果两个线程在不同的cpu运行，此时可能会出现说，线程1将i的值赋值为1，然后写入了cpu高速缓存，同时可能写入了主存；接着将i再次更新为5，然后写入了cpu高速缓存，但是还没来得及写入主存；此时线程2要将i的值赋值给j，那么就会从主存中读取i的值，此时还是1呢，所以就会将1这个值赋值给j

这就是不可见了，因为线程1做的i = 5的修改停留在cpu高速缓存里，还没来得及写入主存，导致线程2没看到。

**（3）有序性**

同时还有一个问题是指令重排序，编译器和指令器，有的时候为了提高代码执行效率，会将指令重排序，就是说比如下面的代码

//线程1:

prepare(); // 准备资源

flag = true;

//线程2:

while(!flag){

Thread.sleep();

}

execute(); // 基于准备好的资源执行操作

重排序之后，让flag = true先执行了，会导致线程2直接跳过while等待，执行某段代码，结果prepare()方法还没执行，资源还没准备好呢，此时就会导致代码逻辑出现异常。

**3.2 java内存模型**

Java内存模型，规定的东西跟操作系统是相关的，规定了所有变量的值都在主存中，每个线程都有自己的工作内存（类似前面说的cpu高速缓存），线程对变量的操作都必须在工作内存中完成，是不能直接对主存进行操作的，而且每个线程不能访问其他线程的工作内存，其实在上面都能找到对应的概念。

所以先熟悉了操作系统内存模型，再看java内存模型，就能看得懂了。

另外，你要明白一个线程对工作内存和主存的操作模型，read（从主存读取），load（将主存读取到的值写入工作内存），use（从工作内存读取数据来计算），assign（将计算好的值重新赋值到工作内存中），store（将工作内存数据写入主存），write（将store过去的变量值赋值给主存中的变量）

**（1）原子性**

举个例子，i = 1，必须线程先在自己的工作内存中将i赋值为1，然后再写入主存中，不能直接修改主存的。而且java中仅仅保证i = 1这种基本数字类型的赋值操作，是原子的，如果是什么i = y，i++这种，都需要先将某个值从主存读入工作内存，操作过后再写回主存，都不是原子的。

voaltile是不能保证原子性的，这个稍后来分析。

原子性，实际上只能靠synchronized和lock来解决，就是仅仅在一段时间内允许一个线程获取锁，然后执行某段代码，操作完之后强制将工作内存数据刷回主存，其他线程获取锁之后立马可以看到数据然后使用。

**（2）可见性**

volatile是用来保证可见性的，就是加了volatile关键字的变量，你在修改之后，会立即刷入主存，接着其他线程会感知到强制从主存读取最新的值，可以保证可见性。

**（3）有序性**

java中有一个happens-before原则：

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

上面这8条原则的意思很显而易见，就是程序中的代码如果满足这个条件，就一定会按照这个规则来保证指令的顺序。

但是如果没满足上面的规则，那么就可能会出现指令重排，就这个意思。这8条原则是避免说出现乱七八糟扰乱秩序的指令重排，要求是这几个重要的场景下，比如是按照顺序来，但是8条规则之外，可以随意重排指令。

**3.3 volatile的作用**

**（1）volatile对可见性的影响**

volatile run = false;

// 线程1，先执行

run=true // 已经变成1了

// 线程2，后执行

// 读取到的还是0

while(!run) {

// 无限循环

}

如果不用volatile，可能导致run变量的修改对其他线程是不可见的，所以上面的代码，不一定能保证说修改了run变量的值之后，理解让线程2跳出来

但是只要你加了volatile关键词，保证了每个线程对一个变量的修改，立即都是对其他线程是可见的，只要线程1一旦修改了run = true，然后理解就是其他线程会看到最新的值，然后while循环就会立即跳出

就是某个线程对一个变量先修改了值，然后另外一个线程去读取那个变量一定会看到最新的值，这个叫做可见性

有的时候我们会用上面的代码来让一个线程控制另外一个线程终止无限循环然后结束，但是如果直接这么搞可能就是有问题的，因为线程1读取了run到自己工作内存，然后依据工作内存中的run=true不断的循环；而线程2虽然将run的值修改为了false而且写入了主存，但是线程1可能确实就是没感知到，因为还是一直在依赖自己工作内存中的run = true在工作。

如果加了volatile关键字之后，可以保证的是，一个线程对数据的写操作，会立即同步到主存，其他线程也是直接从主存来读取的，就是忽略了cpu高速缓存的作用了。因为加了volatile之后，相当于是在cpu层面加了lock指令。这个lock指令，会基于缓存一致性协议来实现，让cpu写数据到高速缓存之后，立即同步到主存，同时其他cpu发现这个共享数据被修改之后，就会标记自己高速缓存中的那个数据失效了，那么如果要对这个失效数据进行修改的时候，会强制重新从内存里把这个数据读到自己缓存里来，再进行修改。

比如上面的场景，如果对i这个共享变量加了volatile关键字之后，线程1修改i++，会强制立即将工作内存中的值刷入主存，而且会导致线程2的工作内存中的i变量的缓存行无效，因为线程2的i变量缓存行无效了，此时就会重新从主存中读取run变量的最新值，就是i = 1。

所以说，volatile是可以保证共享变量的可见性的。

只要是对volatile变量并发操作

某个线程，先修改了值，立即刷回主存；如果另外一个线程此时再来读，会直接从主存里读到最新的值；如果值钱另外一个线程已经将这个变量的值读到自己的工作内存里去了，此时其他线程要从工作内存里读的时候，会发现那个变量的缓存行已经失效了，此时会重新从主存里加载最新的值

**（2）volatile对原子性的影响**

网上没有一篇文章对volatile关于原子性的事情说清楚了，其实坦白说，很多人完全不理解volatile的作用，所以出现了瞎用的情况，比如说下面的例子：

volatile int i = 0

// 线程1

i++

// 线程2

i++

如果说你以为volatile是轻量级锁的人，很傻X，我个人是绝对不认可volatile是一把锁；可以保证多线程并发修改一个共享资源的话 -> JDK傻乎乎的搞了什么synchronized、ReentrantLock、Semaphore。

如果你以为加了个volatile就是可以保证万无一失，那你真的是too young too simple

如果某一时刻，两个线程都先后将这个i的值，i = 0加载到工作内存里来了，结果可能线程1都执行到use步骤了，此时已经将i = 0弄出来准备加1了；同时线程2在这个瞬间就完成了use、assign、store、write等步骤，i = 1，同时写回了主存，并且让线程1的工作内存缓存行失效了；但是然而有什么用呢？线程1此时不需要工作内存了，之前use已经从工作内存读取了数据，所以也同时将i加1变成了1，然后接着assign、store、write回了主存，所以相当于写了两次1回主存。

看明白了java内存模型，你就理解了所谓的可见性是啥，原子性是啥了。

可见性是说，你可能某一次读取是错的，但是下一次再次读取的时候，一定会发现失效的缓存行，重新从主存去read和load进行最新的值；原子性是说，你还是可能会出现某一次读取到的值是旧值，没人告诉你一定能保证100%没问题。

正是对这块的不理解，导致几乎没几个程序员能用好volatile关键字的。

**（3）volatile对有序性的影响**

volatile可以在一定程度上禁止部分指令重排序

//线程1:

prepare(); // 准备资源

volatile flag = true;

//线程2:

while(!flag){

sleep()

}

execute(); // 基于准备好的资源执行操作

比如这个例子，如果用volatile来修饰flag变量，一定可以让prepare()指令在flag = true之前先执行，这就禁止了指令重排。因为volatile要求的是，volatile前面的代码一定不能指令重排到volatile变量操作后面，volatile后面的代码也不能指令重排到volatile前面。

看完了上面的东西，就可以最后说一说这几个概念了，指令重排已经说过了，编译器和指令器为了效率，有的时候会对指令重新排序，但是指令重排的时候一定会遵守happens-before这套规则，如果代码不符合happens-before规则，那么就可以随意进行指令重排，否则就要按照happens-before规则的顺序来走。

内存屏障，其实就是volatile关键字加了之后，就会加一个操作系统层面的lock指令，这就形成了一个所谓内存屏障，或者是内存栅栏吧。其实就是起到一个作用，一个是禁止指令重排，就是上面说的；另外就是说写的时候一定会强制刷主存；写完之后一定会导致其他cpu的高速缓存失效

**3.4 volatile的使用场景**

我觉得说到这里，大家基本上对于如何用volatile关键字就呼之欲出了吧

说白了，就看看上面那个例子，如果说，你可以忍受没有原子性，但是只要保证可见性就好，比如while(run)的时候，某一次run的值没读到最新的是ok的，但是只要下一次读一定能立马读到最新值就好了，那用volatile其实很合适。

如果你要的是i++这种复杂操作要实现线程并发安全，那要么是CAS的AtomicInteger这种，要么是synchronized或者lock加锁，因为只有加锁是能禁止别的线程执行这段操作的。用volatile是不靠谱的。