面试官：你说你精通Java并发，给我讲讲 volatile

Java极客技术 今天

编者荐语：

详细解读Volatile

以下文章来源于Java技术指北 ， 作者指北君

Java技术指北

回复：java，获取精华资料。专注分享Java技术干货、Java 技术、Spring 全家桶、Jav...

大家好，我是指北君。

PS:最近又赶上跳槽的高峰期，好多粉丝，都问我要有没有最新面试题，我连日加班好多天，终于整理好了，公众号回复 【java极客技术PDF】获取面试宝典吧。

今天来了解一下面试题：你对 volatile 了解多少。要了解 volatile 关键字，就得从 Java 内存模型开始。最后到 volatile 的原理。

一、Java 内存模型 (JMM)

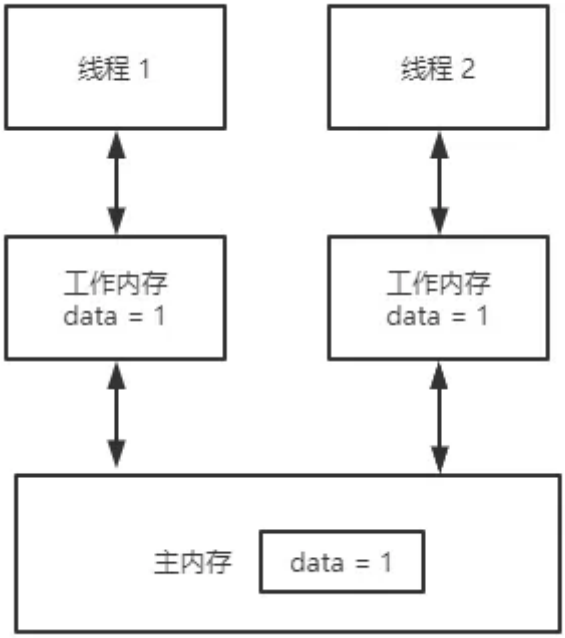
大家都知道 Java 程序可以做到一次编写然后到处运行。这个功劳要归功于 Java 虚拟机。Java 虚拟机中定义了一种 Jva 内存模型（JMM）,用来屏蔽掉各种硬件和操作系统之间内存访问差异，让 Java 程序可以在各个平台中访问变量达到相同的效果。

JMM 的主要目标是定义了程序中变量的访问规则，就是内存中存放和读取变量的一些底层的细节。

JMM 规则

1. 变量包含实例字段，静态字段，构成数组对象的元素，不包含局部变量和方法参数。
2. 变量都存储在主内存上。
3. 每个线程在 CPU 中都有自己的工作内存，工作内存保存了被该线程使用到的变量的主内存副本拷贝。
4. 线程对变量的所有操作都只能在工作内存，不能直接读写主内存的变量。

5. 不同线程之间无法之间访问对方工作内存中的变量。



定义一个静态变量: static int a = 1;

线程 **1** 工作内存	指向	主内存	操作
--	--	a = 1	--
a = 1	<--	a = 1	线程 1 拷贝主内存变量副本
a = 3	--	a = 1	线程 1 修改工作内存变量值
a = 3	-->	a = 3	线程 1 工作内存变量存储到主内存变量

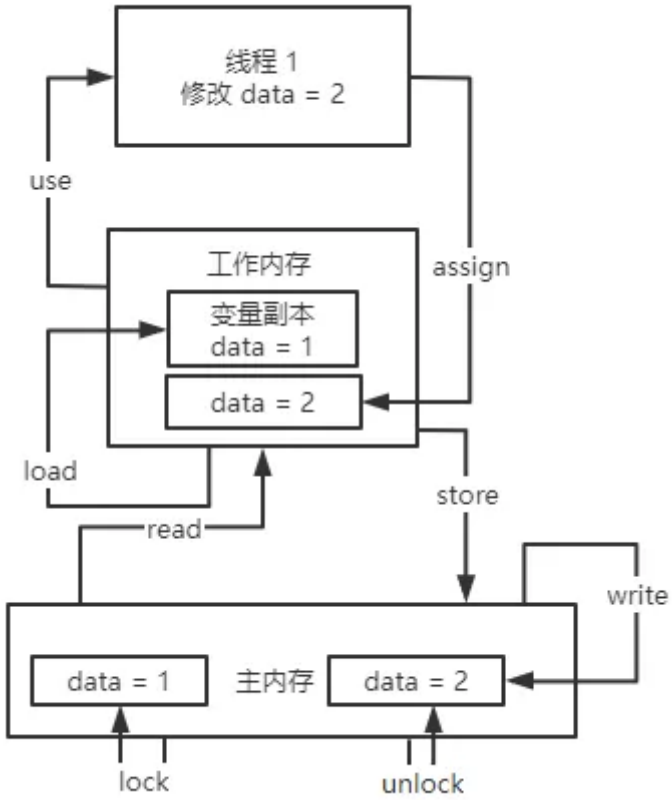
上面的一系列内存操作，在 JMM 中定义了 8 种操作来完成。

JMM 交互

主内存和工作内存之间的交互，JMM 定义了 8 种操作来完成，每个操作都是原子性的。

1. lock (锁定): 作用于主内存变量，把一个变量标识为一条内存独占的状态。
2. unlock (解锁): 作用于主内存变量，把 lock 状态的变量释放出来，释放出来后才能被其他线程锁定。
3. read (读取): 作用于主内存变量，把一个变量的值从主内存传输到工作内存中。
4. load (载入): 作用于工作内存变量，把 read 操作的变量放入到工作内存副本中。
5. use (使用): 作用于工作内存变量，把工作内存中的变量的值传递给执行引擎，每当虚拟机遇到需要这个变量的值的字节码指令时都执行这个操作。

6. assgin (赋值): 作用于工作内存变量，把从执行引擎收到的值赋值给工作内存变量，每当虚拟机遇到需要赋值变量的值的字节码指令时都执行这个操作。
7. store (存储): 作用于工作内存变量，把工作内存中的一个变量值，传送到主内存。
8. write (写入): 作用于主内存变量，把 store 操作的从工作内存取到的变量写入主内存变量中。



从上图中可知，JMM 交互在一条线程中是不会出现任何的问题。但是当有两条线程的时候，线程 1 已经修改了变量的值，但是并未刷新到主内存时，如果此时线程 2 读取变量得到的值并不是线程 1 修改过的数据。

当引入线程 2 的时候 定义一个静态变量: static int a = 1;

操作顺序	线程 **1** 工作内存	线程 **2** 工作内存	指向	主内存	操作
--	--	--	--	a = 1	--
1	a = 1	--	<--	a = 1	线程 1 拷贝主内存变量副本
2	a = 3	--	--	a = 1	线程 1 修改工作内存变量值
3	a = 3	--	-->	a = 1	线程 1 工作内存变量存储到主内存变量，主内存变量还未更新

操作顺序	线程 **1** 工作内存	线程 **2** 工作内存	指向	主内存	操作
4.1	a = 3	a = 1	<--	a = 3	线程 2 拷贝主内存变量副本随后主内存变量更新线程 1 工作内存变量
4.2	a = 3	a = 1	<--	a = 3	线程 1 工作内存变量存储到主内存变量随后线程 2 获取主内存变量副本

下面就可以用 volatile 关键字解决问题。

二、volatile

volatile 可以保证变量对所有线程可见，一条线程修改的值，其他线程对新值可以立即得知。还可以禁止指令的重排序。

可见性

修改内存变量后立刻同步到主内存中，其他的线程立刻得知得益于 Java 的先行发生原则

先行发生原则中的 volatile 原则：一个 volatile 变量的写操作先行于后面发生的这个变量的读操作

定义一个静态变量: static int a = 1;

线程 **1** 工作内存	线程 **2** 工作内存	指向	主内存	操作
--	--	--	a = 1	--
a = 1	--	<--	a = 1	线程 1 拷贝主内存变量副本
a = 3	--	--	a = 1	线程 1 修改工作内存变量值
a = 3	--	-->	a = 1	线程 1 工作内存变量存储到主内存变量
a = 3	a = 3	<--	a = 3	volatile 原则: 主内存变量保存线程A工作内存变量操作在线程 2 工作内存读取主内存变量操作之前

可见性原理

对 volatile 修饰的变量，在执行写操作的时候会多出一条 lock 前缀的指令。JVM 将 lock 前缀指令发送给 CPU，CPU 处理写操作后将最后的值立刻写回主内存，因为有 MESI 缓存一致性协议保证了各个 CPU 的缓存是一致的，所以各个 CPU 缓存都会对总线进行嗅探，本地缓存中的数据是否被别的线程修改了。

如果别的线程修改了共享变量的数据，那么 CPU 就会将本地缓存的变量数据过期掉，然后这个 CPU 上执行的线程在读取共享变量的时候，就会从主内存重新加载最新的数据。

原子性（不保证）

volatile 并不保证变量具有原子性。

```
public class VolatileTest implements Runnable {

    public static volatile int num;

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            num++;
        }

    }

    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            VolatileTest t = new VolatileTest();
            Thread t0 = new Thread(t);
            t0.start();
        }
        System.out.println(num);

    }
}
```

这段代码的结果有可能不是 100000，有可能小于 100000。因为 num++ 并不是原子性的。

有序性（防止指令重排序）

volatile 是通过禁止指令重排序来保证有序性。为了优化程序的执行效率 JVM 在编译 Java 代码的时候或者 CPU 在执行 JVM 字节码的时候，不影响最终结果的前提下会对指令进行重新排序。

编译器会根据以下策略将内存屏障插入到指令中，禁止重排序：

1. 在 volatile 写操作之前插入 StoreStore 屏障。禁止和 StoreStore 屏障之前的普通写操作不会进行重排序。
2. 在 volatile 写操作之后插入 StoreLoad 屏障。禁止和 StoreLoad 屏障之后的 volatile 读写重排序。
3. 在 volatile 读操作之后插入 LoadLoad 屏障。禁止和 LoadLoad 之后的普通读和 volatile 读重排序。
4. 在 volatile 写操作之后插入 LoadStore 屏障。禁止和 LoadStore 屏障之后的普通写操作重排序。

总结

面试被问到 volatile 的时候，可以从 Java 内存模型到原子性、有序性、可见性，最后到 volatile 的原理：内存屏障和 lock 前缀指令。

< END >