Java内存模型

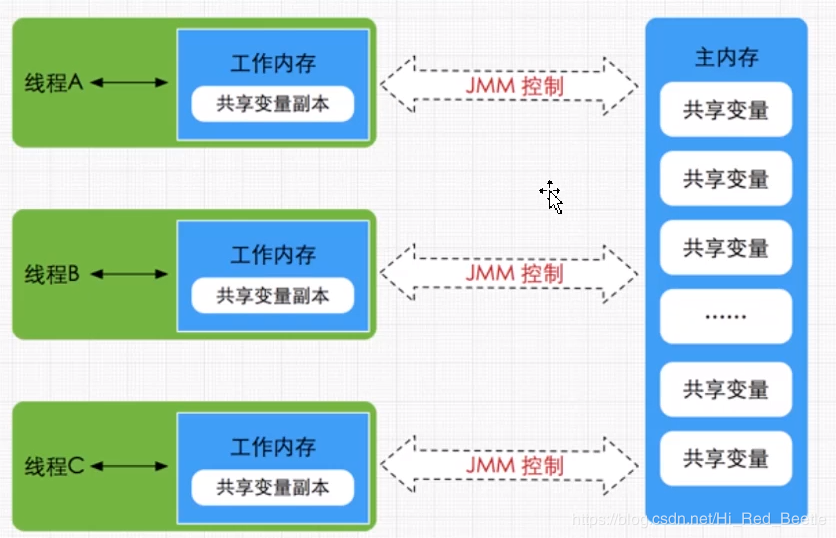
一、概念

JMM——Java Memeory Model，即Java内存模型，也可以说是JVM内存模型。它的主要作用就是用于定义变量的访问规则。这里的变量是指所有线程可以共享的变量，比如成员变量，不能是局部变量。因为局部变量（在方法内）是放在栈中的，而每个线程都有属于自己的栈，所以局部变量不是线程共享的。

JMM将内存划分为两个区：主内存区和工作内存区。可能这里你会问内存空间不是分为堆、栈、方法区等这些区域吗？其实都没有错，只是从不同的角度进行的划分而已。就比如说人从性别的角度可以划分为男人和女人，但从年龄的角度划分又可以划分为老人、中年人、年轻人、小孩等。下面具体说一下这两个区域:

主内存区：存放真实的变量

工作内存区：存放的是主内存中变量的副本，供各个线程使用。工作内存是各个线程私有的，每个线程有属于自己的工作内存。



**这里需要注意两点：**

1.各个线程只能访问自己私有的工作内存，不能访问其他线程的工作内存，也不能访问主内存；

2.不同线程之间，可以通过主内存间接的访问其他线程的工作内存

明确了上面的概念后我们看一下下面这段代码，用下面这段代码证明一下内存模型的真实存在：

public class JMMTest {

private static boolean initFlag = false;

public static void main(String[] args) throws InterruptedException {

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("waiting date...");

while (!initFlag){

}

System.out.println("==========success");

}

}).start();

//为了保证线程1先执行完再执行线程2，所以在这里让主线程睡两秒。

Thread.sleep(2000);

new Thread(new Runnable() {

@Override

public void run() {

prepareData();

}

}).start();

}

private static void prepareData() {

System.out.println("prapareing data...");

initFlag = true;

System.out.println("prepare date end...");

}

}

首先我们在JMMTest这个类中创建了一个共享的成员变量initFlag，接着我们创建了两个线程来模拟工作中遇到的业务场景：线程2用来准备数据，当initFlag被线程2置为ture时说明数据已经准备好了，线程2运行完毕；线程1用来处理数据，它会通过while (!initFlag)不断的去判断数据是否已经就绪，如果没有就绪就一致执行while循环，但当数据被线程2准备好后（initFlag=true),它就会跳出循环，通过输出====success模拟对数据处理成功。所以我们预想的输出结果应该是：

1.waiting date...

2.prapareing data...

3.prepare date end...

4.==========success

那么实际情况如何呢？你可以拷贝一下代码运行一下，我这里先给出输出结果：

1.waiting date...

2.prapareing data...

3.prepare date end...

可以看到==========success始终没有打印出来，并且程序一直处于运行状态。这和我们预想的结果不一样啊？怎么回事？这其实就是java内存模型的效果。initFlag这个共享变量除了在主内存中有一份外，在两个线程的工作内存中都分别会有一份副本，当线程2修改了initFlag=true时，它只是修改了自己工作内存中副本值（修改后会更新主内存的值为最新值），而线程1工作内存中的initFlag的值依旧是false，也就是说线程1没有感知到线程2对共享变量initFlag的修改（没有从主内存中拉取最新值）。那么怎样才能让某一个线程对共享变量的修改立马让其他线程感知到呢？就是给共享变量加一个volatile的关键字，这就是我们经常听说的volatile“可见性”原理，具体的原理细节我会在下面详细展开，在这里你先不放试一下给initFlag加个volatile看一看是不是和预想的结果一致。

二、JMM数据的原子操作

**2.1 JMM数据的原子操作**

上面说了这么多，那么一个变量是如何从主内存拷贝到工作内存、又如何从工作内存同步回主内存的呢？Java内存模型定义了以下8种操作来实现，下前面的每一个操作都是原子的、不可分割的：

1.read:作用于主内存变量，将主内存中的变量读取（拷贝）到工作内存（大范围内，因为整个工作内存空间很大，这个阶段只是将其放在了工作内存中的某处，还没有给到变量i的副本）

2.load:作用于工作内存的变量，将2中读取的变量拷贝到变量副本中（小范围，精准定位到某个内存地址/变量上）

3.use:作用于工作内存的变量，将工作内存中的变量副本传递给线程去使用

4.assign：作用于工作内存的变量，将线程正在使用的变量（线程正在使用的变量也就是此刻CPU正在参与运算的变量），传递给工作内存中的变量副本

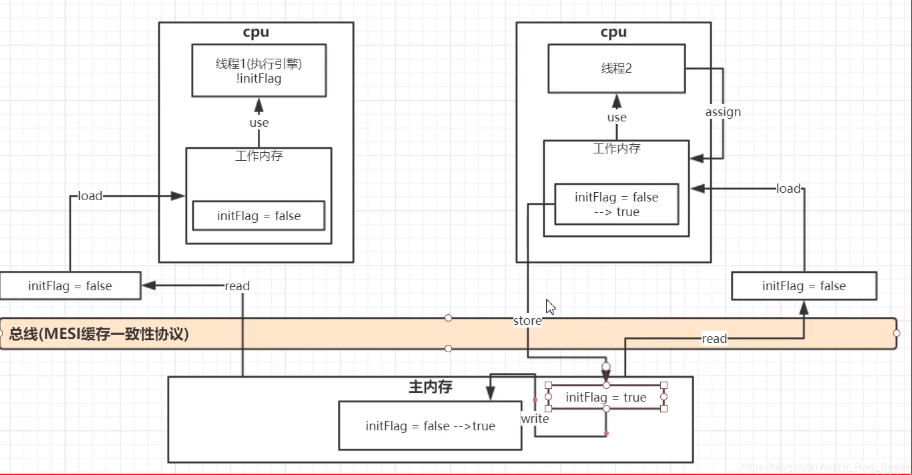
5.store:作用于工作内存的变量，将工作内存中变量副本的值传递到主内存中（大范围内,因为整个主内存空间很大，这个阶段只是将其放在了主内存中的某处，还没有把值给到真正的变量i）

6.write:作用于主内存的变量，将6中传递过来的变量副本作为一个主内存中的变量进行存储（小范围，精准定位到真正的变量i上）

7.lock：作用于主内存的变量，将主内存中的变量标识为一条线程的独占状态。

8.unlock：作用于主内存变量，解除线程的独占状态，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

我们还是以上面的代码为例子看一下共享变量initFlag是如何在工作内存和主内存中流转的，先看在没加volatile关键字之前的流转图（图中省略了lock和unlock操作，不影响我们分流转过程）：



首先刚开始initFlag在主内存中的初始值是false，之后线程1会通过read——>load将读到工作内存中，然后线程1不停的执行while循环（use）。线程2也是通过read——>load将初始值为false的initFlag从主内存读到属于自己的工作内存中,之后cpu运转将false修改为true（use），修改完之后通过assign将修改后的值再赋给工作内存，此时工作内存中的initFlag=true;再之后通过store——>write将工作内存中的新值写回给主内存，此时主内存中的initFlag变为true。但是这个时候虽然主内存中的initFlag已经为true了，但是属于线程1的工作内存中initFlag已久为false，并没有什么机制在主内存的变量变为最新值后通知给线程1，这就导致了线程1一直在哪里空转占用cpu资源。这也就解释了我们上面程序中打印出的结果中始终没有输出success.

**2.2 volatile可见性原理**

那么我们再看一下给initFlag加了volatile关键字以后与上面的流程有什么不同？为什么加了volatile以后就可以实现线程间的“可见性”呢？主要有以下两点：

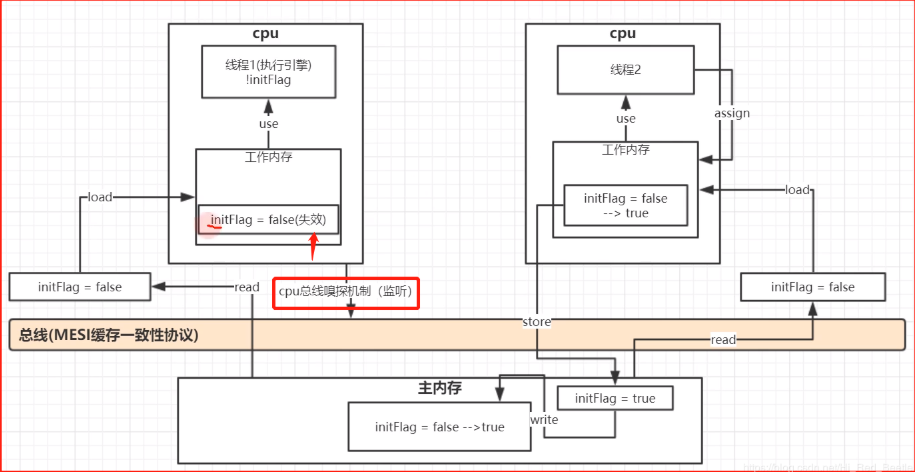
加了volatile后会将当前处理器缓存的的共享变量（工作内存中的数据）立即写回主内存（即立刻执行store和write操作）；

这个写回主内存的操作会引起其他线程工作内存中的共享变量无效（MESI协议）。

我们对上面的两点再做更进一步的分析：

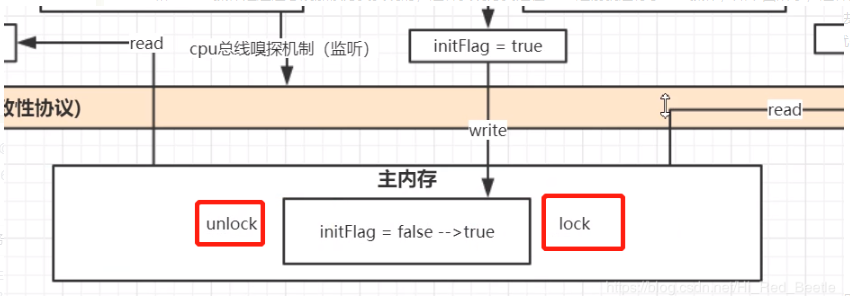
1.volatile的底层实现其实是通过汇编语言的lock前缀指令，只要我们给一个共享变量加了volatile关键字，那么lock指令就会锁定工作内存中的某个地址（这个地址正是存储了共享变量的地址），每当工作内存中的这个变量的值发生变化的时候就会出发lock指令立马写回到主内存，这是lock指令的第一个作用；

2.工作内存在写回数据到主内存的时候会经过总线（总线的作用就是传输数据，两块内存不能凭空气传递数据啊，肯定需要传输介质，总线就是这个介质的作用），lock指令会触发MESI协议（这个协议的具体作用就是其他运行中线程会去监听总线的状态——即总线嗅探机制，当一个线程将工作内存中的数据经过总线写回主内存时，那么在总线上进行监听的其他线程就会感知到数据的变化，进而将自己工作内存中的数据置为失效），这就是lock指令的第二个作用。以上就volatile实现“可见性”的深层次原理。我们通过图来直观的感受一下：

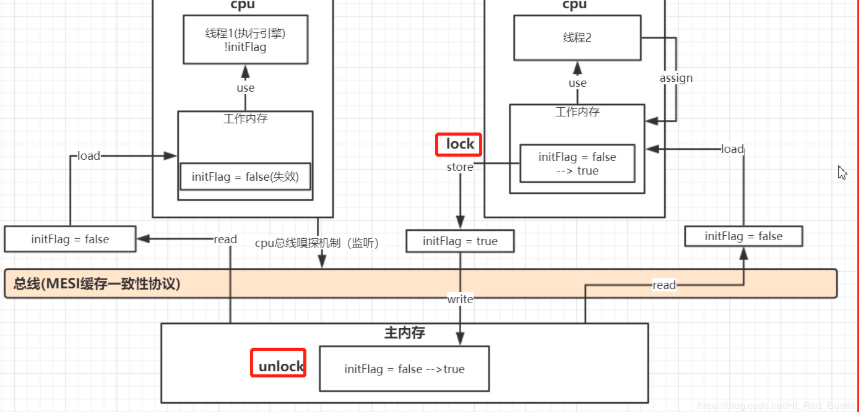


**2.3 lock和unlock**

上面的8个原子操作中没有具体说lock和unlock，这里单拿出来聊一聊，因为这两个操作相比于其他操作涉及的东西多一些。在早些时候lock和unlock操作是通过总线加锁方式实现的，这种实现方式是在read之前就进行了lock操作，如下图所示，这种方式的弊端之一就是性能低下，因为这样的话一个cpu在从主内存读取数据到工作内存回对整个过程加锁，这样的话其他cpu就没有办法从主内存读取这个数据，直到这个cpu使用完数据后释放掉锁之后其他cpu才能继续读取数据。这样多核cpu“并行”执行任务的优点就体现不出来，因为这样的话，虽然有多个cpu,但其实在执行任务的时候还是“串行”执行。



而volatile方式的lock操作是在store之前进行的，这个锁的粒度相比于上面的粒度要小很多，因为上面的锁它的作用范围横跨了主内存和工作内存交换数据的整个过程（read、load、use、assign、store、write）,而这个锁只涉及store和write两个过程，这样效率得到了很大的提升。



清楚了上面的概念后我们再思考一个问题：lock和unlock操作可不可以省略？其实我们在上面讲的时候就没有在流程图中画这两个操作，感觉也很好的实现了共享变量的可见性，但仔细分析你就会发现没有这两个操作是不行的，主要原因还是“高并发”会引起的问题，思考下面两个场景：

假如线程1和线程2同时往主内存中回写数据，如果没有锁机制是不是会导致数据出错。

假如线程2执行完了store操作（已经写入主内存中，但是还没有赋值给initFlag变量，此时的initFlag依旧是false),而正在此时，线程1进行了read操作，那么它拿到的值依旧时initFlag=false的值。

所以综上，lock和unlock是必须要有的操作。

三、volatile

其实volatile就是JVM提供的一个轻量级的同步机制，它的主要作用就是以下三个：

防止jvm对long/double等64位的非原子性协议进行误操作（读取半个类型）；

可以使变量对所有的线程立即可见（某一个线程如果修改了工作内存中的变量副本，那么加上volatile之后，该变量就会立刻同步到其他线程的工作内存中去）；

禁止指令的“重排序”优化。

上面的第一点我就不展开细说了，博文的这部分内容我们主要就volatile对并发编程中的三大特性可见性、原子性、有序性具体展开讲一讲。上面的第二点和第三点分别就是对应的可见性和有序性，这两点volatile都可以保证，但原子性volatile是保证不了的。

**3.1可见性**

关于volatile可以保证可见性的原理我们在上面的“2.2volatile可见性原理”中已经做了详细的分析，这里不在赘述。

**3.2原子性**

首先原子性指的就是不可分割的操作，比如：.num = 10;就是一个原子操作，因为它不能在分割了，而int num = 10;就是一个非原子操作，因为它可以进一步分割为两个操作——>int num;num = 10；我们通过下面的代码来验证volatile不可以保证原子性：

public class TestVolatile {

static volatile int num = 0;

public static void main(String[] args) throws InterruptedException {

for (int i = 0; i < 100; i++) {

//每个线程将num累加30000次，在线程安全的前提下，最后num结果应该是300万。

new Thread(new Runnable() {

@Override

public void run() {

for(int j = 0; j < 30000; j++){

num++;//不是一个原子操作，可以进一步拆分成如下1、2操作

/\*\*

\* num = num + 1:

\* 1.num + 1

\* 2.num = 1的结果

\*

\* 2个线程同时执行num+1(假设此时num=10)

\* 一个线程：10+1 ->11

\* 另一个线程：10+1 ->11

\* 结果：漏加

\*/

}

}

}).start();

}

//让main线程先睡几秒，以保证上面的3万个线程都执行完毕再输出结果

Thread.sleep(5000);

System.out.println("num=" + num);

}

}

上面的代码在执行完成后，一般情况下都会小于300万，原因就是num++不是原子操作，他会进一步拆分成两个原子操作，这样在多线程中就可能会出现漏加的情况，导致最后结果小于预期值。程序里面虽然num用volatile修饰了，但它依然保证不了num++是原子操作，这就是volatile不可保证原子性的具体体现。为了在高并发场景下此类问题的发生我们可以把num++的这个操作放在一个方法内执行，然后给这个方法加上synchronized关键字。或则可以使用原子包java.util.concurrent.aotmic包中的AtomicInteger类，该类能够保证原子性的核心是因为提供了compareAndSet()方法，该方法提供了CAS算法（无锁算法）:

将static volatile int num = 0;

改写为static AtomicInteger num = new AtomicInteger(0);

将num++改为num.incrementAndGet();

**3.3有序性**

在讲有序性之前我们需要先搞懂重排序的概念：

重排序：CPU为了提高执行效率,有时候会将某些语句拆分成原子操作，然后对这些原子操作做进一步排序（排序的对象就是原子操作）。

重排序原则：重排序后不会影响“单线程的执行结果”。

我们看一下大家都熟悉“双重检查机制的单例模式”代码：

//双重检查式的懒汉式单例模式

public class Singleton {

private static Singleton instance = null;

private Singleton(){ }

public static Singleton getInstance(){

if(instance == null){

synchronized (Singleton.class){

if(instance == null){

instance = new Singleton();//不是一个原子性操作

}

}

}

return instance;

}

}

在单线程下以上代码没有任何问题，但是在多线程高并发的情况下以上代码可能会出现问题，原因是instance = new Singleton()不是一个原子性操作，会在执行时拆分成以下几个原子操作：

1.JVM会分配内存地址和空间

2.使用构造方法实例化对象

3.instance = 第一步中分配好的内存地址

根据重排序的规则可知，以上3个动作在真正执行时可能123，也可能是132，因为这样排序后在单线程下不会对结果造成任何影响，所以是可以进行重排序的。但是如果在多线程环境下，使用132可能出现如下问题：假设线程A刚刚执行完以下步骤（即刚执行1、3，但还没有执行2）

    1.JVM会分配内存地址和空间0x123

    3.nstance = 第一步中分配好的内存地址：instance=0x123

此时线程B进入单例程序的if判断，直接会得到instance对象（注意，此时的instance是刚才线程A并没有new出来的对象），就去使用该对象，例如调用对象的方法instance.xxx()必然报错，因为此时虽然给对象分配了内存地址，但是却没有把new出来的对象放到这个地址上。解决方案就是禁止此程序使用132的重排序，那么在instance前面加上volatile关键字即可实现：

private volatile static Singleton instance = null;

以上就是volatile保证有序性的具体体现。

最后作为了解，volite其实是通过“内存屏障”来防止重排序问题的。