- # 要搞懂 volatile 关键字，就靠这 26 张图

小林coding *2020-11-23*

小故事

小艾吃饭路上碰上小牛，忙问：你昨天面大厂面的咋样了？听说他们最喜欢问多线程相关知识。

小牛说：对啊，第一个问题我就讲了20分钟，直接把面试官讲服了。

小艾忙问：什么问题能讲这么久？是不是问你情感经历了？

小牛说：...问的volatile关键字。

小艾说：volatile关键词的作用一般有如下两个：

1. 可见性：当一个线程修改了由volatile关键字修饰的变量的值时，其它线程能够立即得知这个修改。
2. 有序性：禁止编译器关于操作volatile关键词修饰的变量的指令重排序。

你说这两个说了20分钟？口吃？

小牛说：你知道volatile的实现原理吗？

小艾说：缓存一致性协议嘛，这有啥？

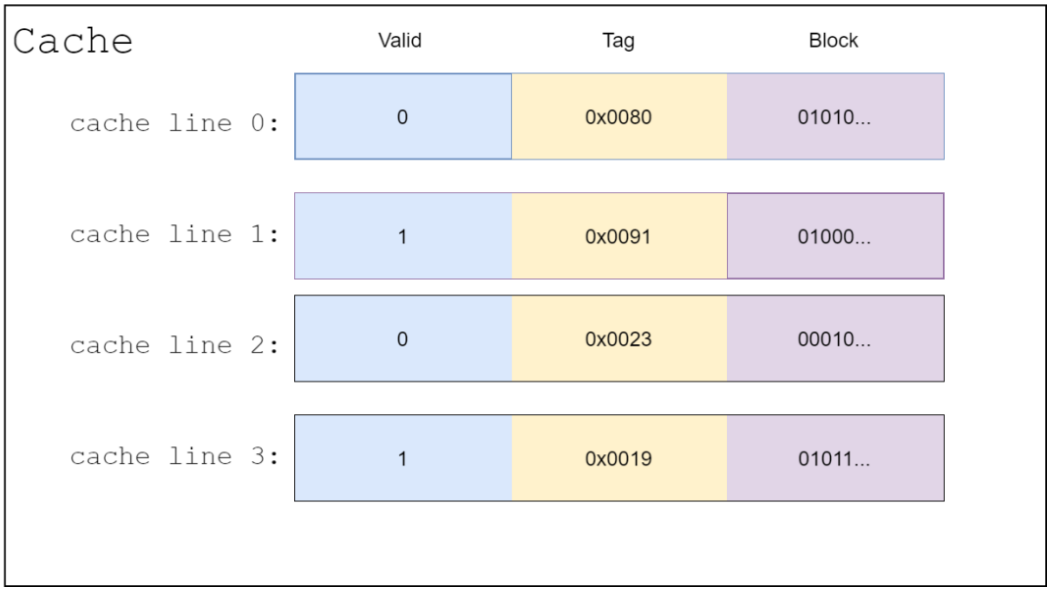
小牛说：既然硬件保证了缓存一致性协议，无论该变量是否被volatile关键词修饰，它都该满足缓存一致性协议呀。你这说的有点自相矛盾哦。

小艾说：那volatile的实现原理是什么？

小牛说：且听我慢慢道来。

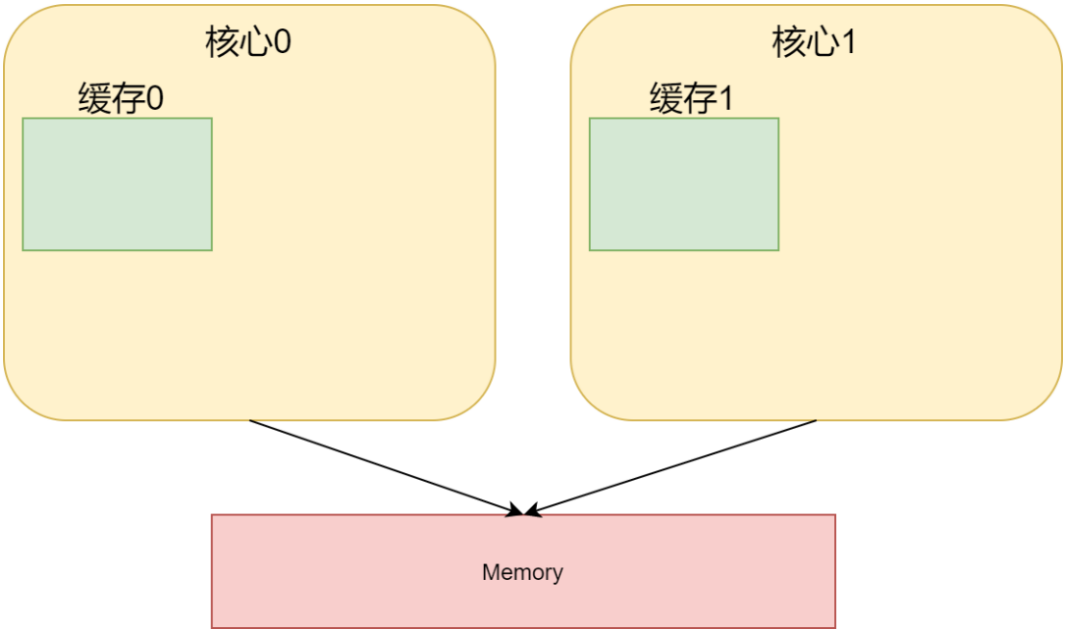
缓存一致性协议

我们知道，现代CPU都是多核处理器。由于cpu核心（Kernel）读取内存数据较慢，于是就有了缓存的概念。我们希望针对频繁读写的某个内存变量，提升本核心的访问速率。因此我们会给每个核心设计缓存区(Cache)，缓存该变量。由于缓存硬件的读写速度比内存快，所以通过这种方式可以提升变量访问速度。
缓存的结构可以如下设计：



缓存结构图

其中，一个缓存区可以分为N个缓存行(Cache line)，缓存行是和内存进行数据交换的最小单位。每个缓存行包含三个部分，其中valid用于标识该数据的有效性。如果有效位为false，CPU核心就从内存中读取，并将对应旧的缓存行数据覆盖，否则使用旧缓存数据；tag用于指示数据对应的内存地址；block则用以存储数据，



多核缓存和内存

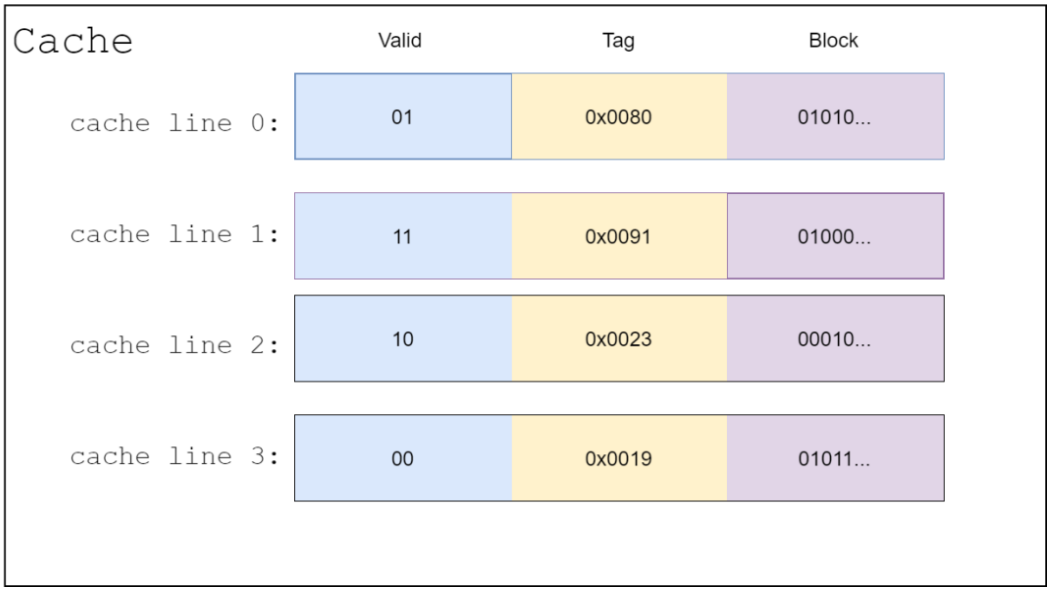
但是，如果涉及到并发任务，多个核心读取同一个变量值，由于每个核心读取的是自己那一部分的缓存，每个核心的缓存数据不一致将会导致一系列问题。缓存一致性的问题根源就在于，对于某个变量，好几个核心对应的缓存区都有，到底哪个是新的数据呢？如果只有一个CPU核心对应的缓存区有该变量，那就没事啦，该缓存肯定是新的。

所以为了保证缓存的一致性，业界有两种思路：

1. 写失效(Write Invalidate)：当一个核心修改了一份数据，其它核心如果有这份数据，就把valid标识为无效；
2. 写更新(Write update)：当一个核心修改了一份数据，其它核心如果有这份数据，就都更新为新值，并且还是标记valid有效。

业界有多种实现缓存一致性的协议，诸如MSI、MESI、MOSI、Synapse、Firefly Dragon Protocol等，其中最为流行的是MESI协议。

MESI协议就是根据写失效的思路，设计的一种缓存一致性协议。为了实现这个协议，原先的缓存行修改如下：



缓存结构图

原先的valid是一个比特位，代表有效/无效两种状态。在MESI协议中，该位改成两位，不再只是有效和无效两种状态，而是有四个状态，分别为：

1. M（Modified）：表示核心的数据被修改了，缓存数据属于有效状态，但是数据只处于本核心对应的缓存，还没有将这个新数据写到内存中。由于此时数据在各个核心缓存区只有唯一一份，不涉及缓存一致性问题；
2. E（Exclusive）：表示数据只存在本核心对应的缓存中，别的核心缓存没这个数据，缓存数据属于有效状态，并且该缓存中的最新数据已经写到内存中了。同样由于此时数据在各个核心缓存区只有一份，也不涉及缓存一致性问题；
3. S（Shared）：表示数据存于多个核心对应的缓存中，缓存数据属于有效状态，和内存一致。这种状态的值涉及缓存一致性问题；
4. I（Invalid）：表示该核心对应的缓存数据无效。

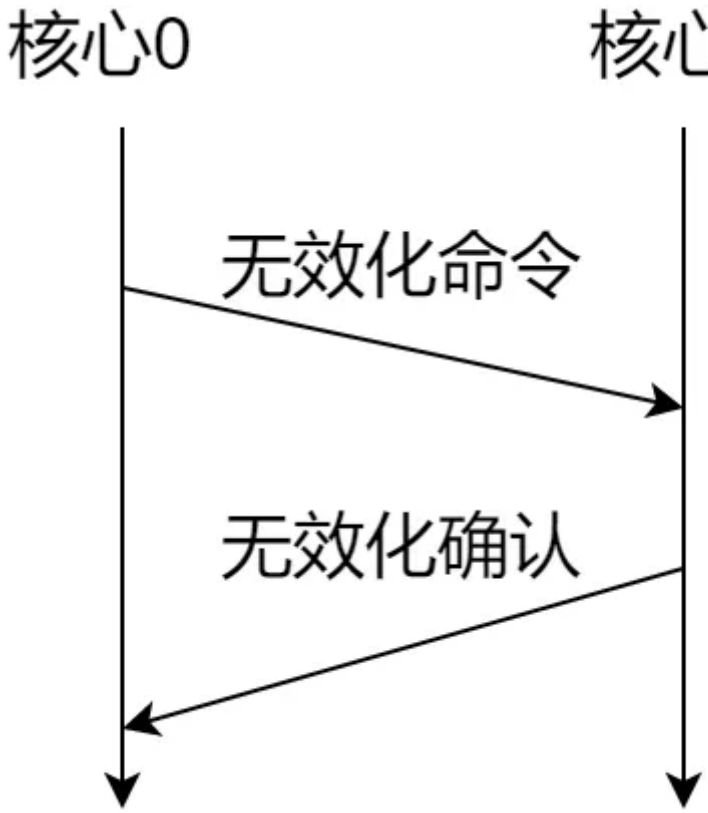
看到这里，大家想必知道为什么这个协议称为MESI协议了吧，它的命名就是取了这四个状态的首字母而已。

为了保证缓存一致性，每个核心要写新数据前，需要确保其他核心已经置同一变量数据的缓存行状态位为Invalid后，再把新数据写到自己的缓存行，并之后写到内存中。

MESI协议包含以下几个行为：

- 读（Read）：当某个核心需要某个变量的值，并且该核心对应的缓存没这个变量时，就会发出读命令，希望别的核心缓存或者内存能给该核心最新的数据；
- 读命令反馈（Read Response）：读命令反馈是对读命令的回应，包含了之前读命令请求的数据。举例来说，Kernel0发送读命令，请求变量a的值，Kernel1对应的缓存区包含变量a，并且该缓存的状态是M状态，所以Kernel1会给Kernel0的读命令发送读命令反馈，给出该值；
- 无效化（Invalidate）：无效化指令是一条广播指令，它告诉其他所有核心，缓存中某个变量已经无效了。如果变量是独占的，只存在某一个核心对应的缓存区中，那就不存在缓存一致性问题了，直接在自己缓存中改了就行，也不用发送无效化指令；
- 无效化确认（Invalidate Acknowledge）：该指令是对无效化指令的回复，收到无效化指令的核心，需要将自己缓存区对应的变量状态改为Invalid，并回复无效化确认，以此保证发送无效化确认的缓存已经无效了；
- 读无效（Read Invalidate）:这个命令是读命令和无效化命令的综合体。它需要接受读命令反馈和无效化确认；
- 写回（Writeback）这个命令的意思是将核心中某个缓存行对应的变量值写回到内存中去。

下图给了个一个应用MESI读写数据的例子。在该图中，假设CPU有两个核心，Kernel0表示第一个核心，Kernel1表示第二个核心。这里给出了Kernel0想写新数据到自己缓存的例子。



MESI工作原理

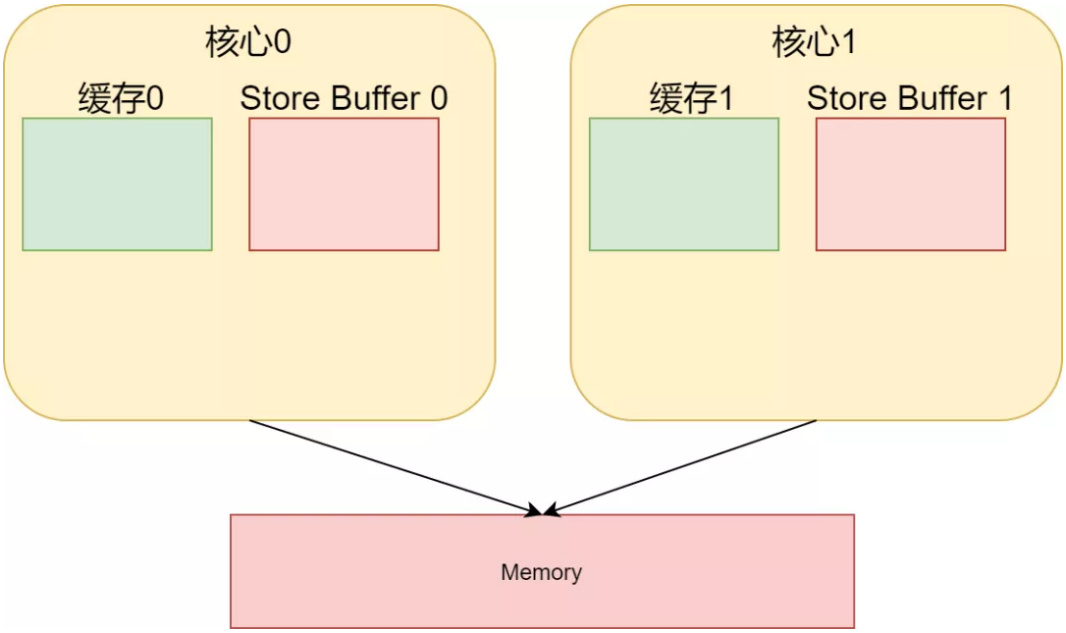
1. 首先Kernel0先完成新数据的创建；
2. Kernel0向全体其他核心发送无效化指令，告诉其他核心其所对应的缓存区中的这条数据已经过期无效。本图例中只有一个其他核心，为Kernel1；
3. 其他核心收到广播消息后，将自己对应缓存的数据的标志位记为无效，然后给Kernel0回确认消息；
4. 收到所有其他Kernel的确认消息后，Kernel0才能将新数据写回到它所对应的缓存结构中去。

根据上图，我们可以发现，影响MESI协议的时间瓶颈主要有两块：

1. 无效化指令：Kernel0需要通知所有的核心，该变量对应的缓存在其他核心中是无效的。在通知完之前，该核心不能做任何关于这个变量的操作。
2. 确认响应：Kernel0需要收到其他核心的确认响应。在收到确认消息之前，该核心不能做任何关于这个变量的操作，需要持续等待其他核心的响应，直到所有核心响应完成，将其对应的缓存行标志位设为Invalid，才能继续其它操作。

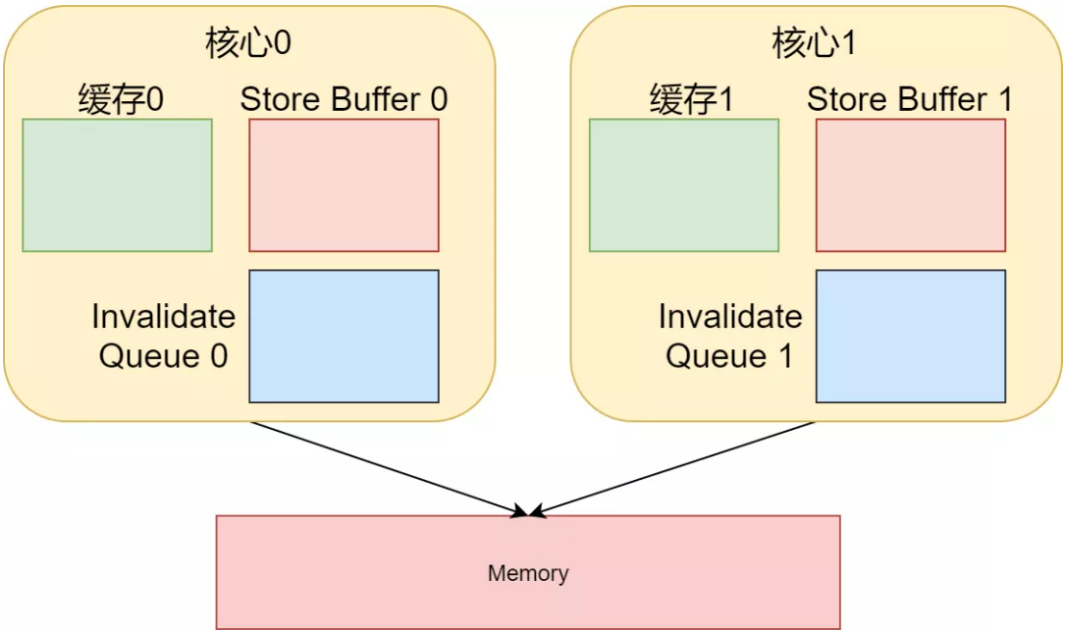
针对这两部分，我们可以进一步优化：

1. 针对无效化指令的加速：在缓存的基础上，引入Store Buffer这个结构。Store Buffer是一个特殊的硬件存储结构。通俗的来讲，核心可以先将变量写入Store Buffer，然后再处理其他事情。如果后面的操作需要用到这个变量，就可以从Store Buffer中读取变量的值，核心读数据的顺序变成Store Buffer → 缓存 → 内存。这样在任何时候核心都不用卡住，做不了关于这个变量的操作了。引入Store Buffer后的结构如下所示：



Store Buffer结构

1. 针对确认响应的加速：在缓存的基础上，引入Invalidate Queue这个结构。其他核心收到Kernel0的Invalidate的命令后，立即给Kernel0回Acknowledge，并把Invalidate这个操作，先记录到Invalidate Queue里，当其他操作结束时，再从Invalidate Queue中取命令，进行Invalidate操作。所以当Kernel0收到确认响应时，其他核心对应的缓存行可能还没完全置为Invalid状态。引入Invalidate Queue后的结构如下所示：



Invalidate Queue结构

缓存一致性协议优化存在的问题

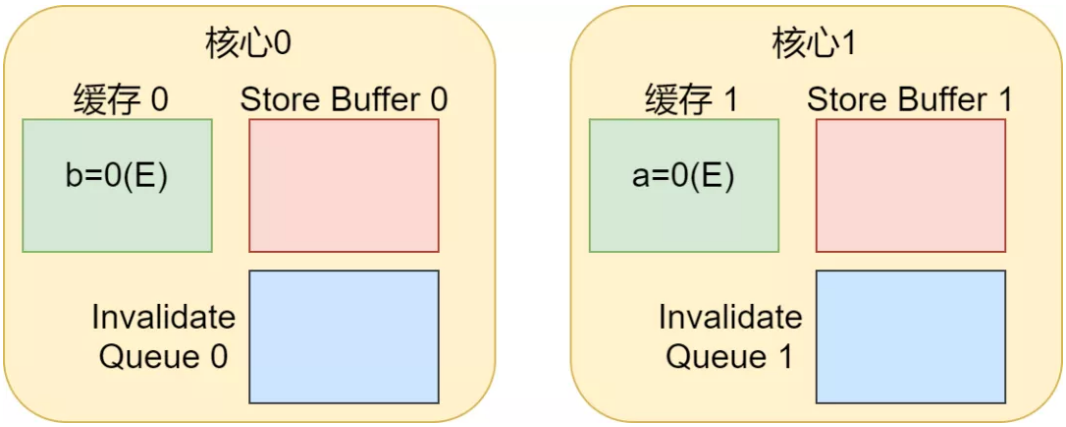
上一节讲了两种缓存一致性协议的加速方式。但是这两个方式却会对缓存一致性导致一定的偏差，下面我们来看一下两个出错的例子：

例子1：关于Store Buffer带来的错误，假设CPU有两个核心，Kernel0表示第一个核心，Kernel1表示第二个核心。

```
...
public void foo(){
    a=1;
    b=1;
}
public void bar(){
    while(b==0) continue;
    assert(a==1):"a has a wrong value!";
}
...
```

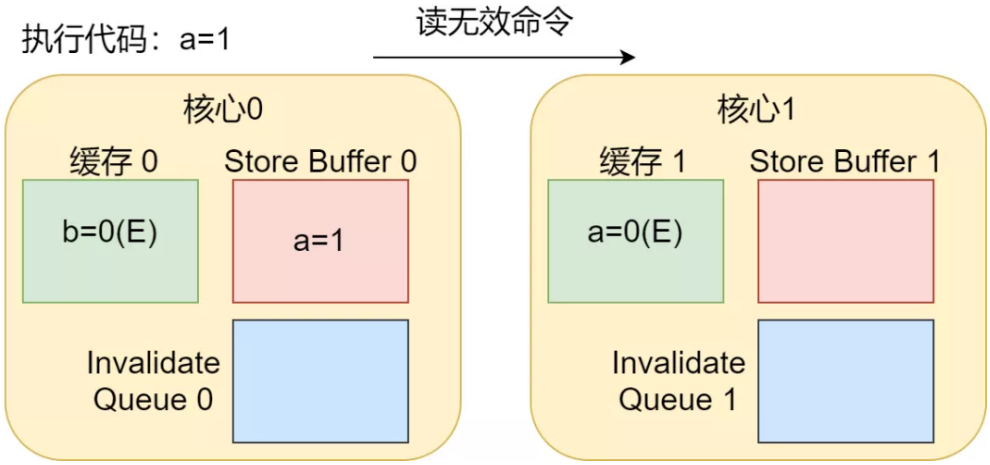
如果Kernel0执行foo()函数，Kernel1执行bar()函数，按照之前我们的理解，如果b变量为1了，那a肯定为1了，assert(a==1)肯定不会报错。但是事实却不是这样的。

假设初始情况是这样的：在执行两个函数前Kernel1的缓存包含变量a=0，不包含缓存变量b，Kernel0的缓存包含变量b=0，不包含缓存变量a。

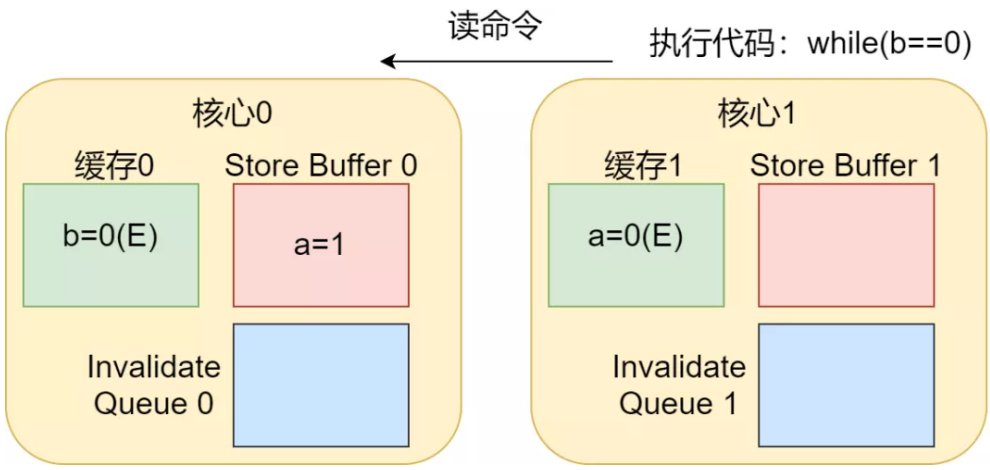


Kernel0执行foo()函数，Kernel1执行bar()函数时，。这样的话计算机的指令程序可能会如下展开：

1. Kernel0执行a=1。由于Kernel0的缓存行不包含变量a，因此Kernel0会将变量a的值存在Store Buffer中，并且向其他Kernel进行read Invalidate操作，通知a变量缓存无效；

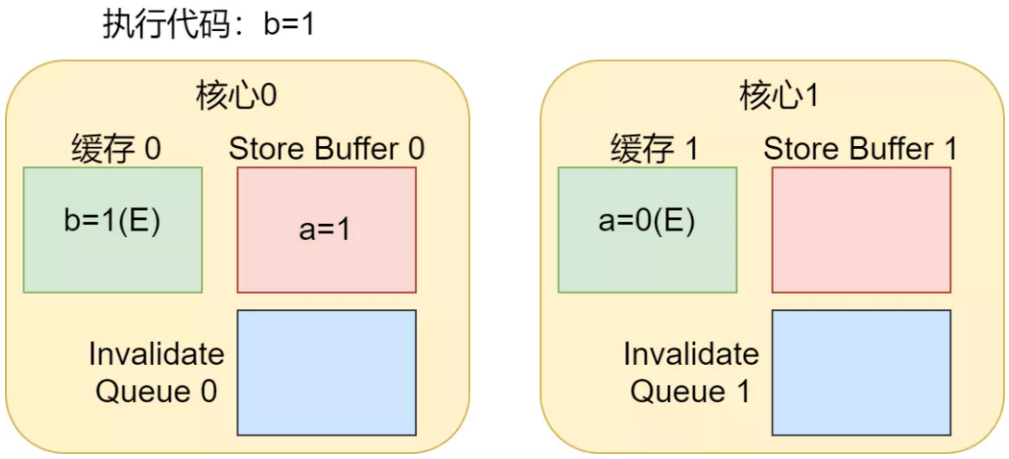


2. Kernel1执行while(b==0)，由于Kernel1的缓存没有变量b，因此它需要发送一个读命令，去找b的值；



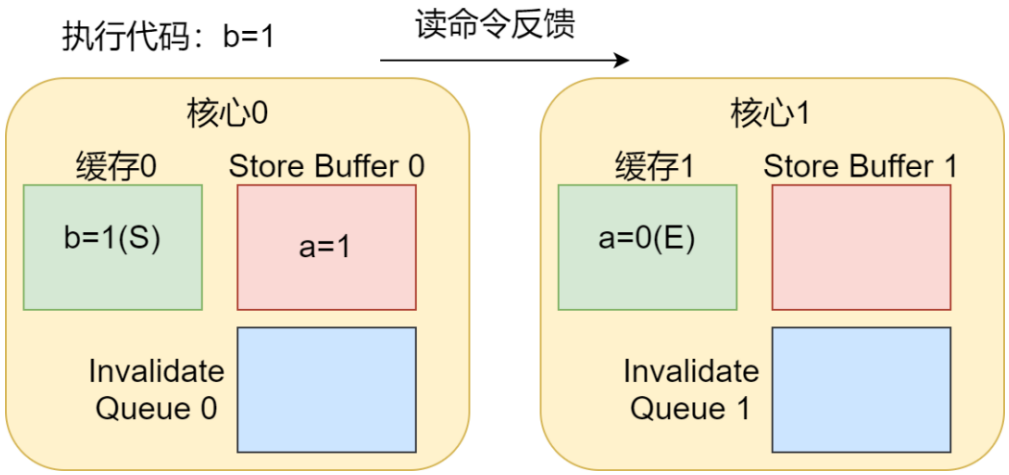
2

3. Kernel0执行b=1，由于Kernel0的缓存中已经有了变量b，而且别的核心没有这个变量的缓存，所以它可以直接更改缓存b的值；



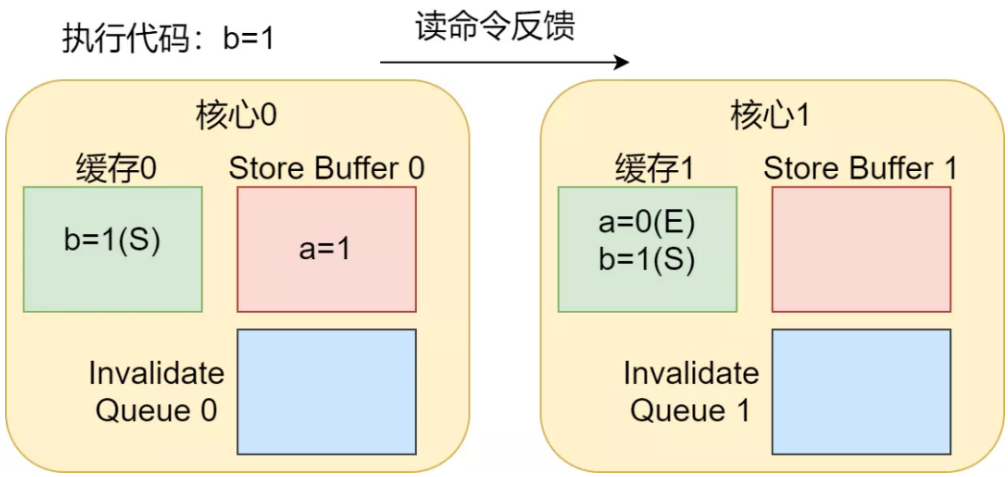
3

4. Kernel0收到读命令后，将最新的b的值发送给Kernel1，并且将变量b的状态由E（独占）改变为S（共享）；



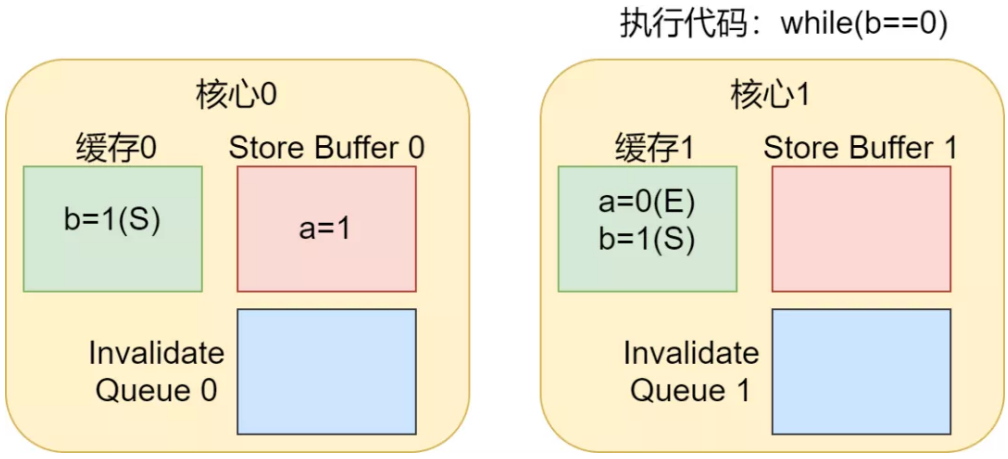
4

5. Kernel1收到b的值后，将其存到自己Kernel对应的缓存区中；



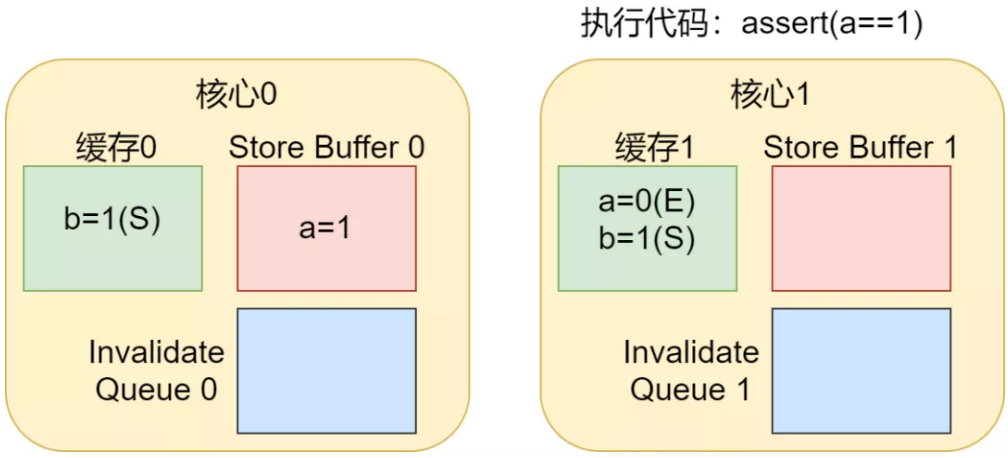
5

6. Kernel1接着执行while(b==0)，因为此时b的新值为1，因此跳出循环；



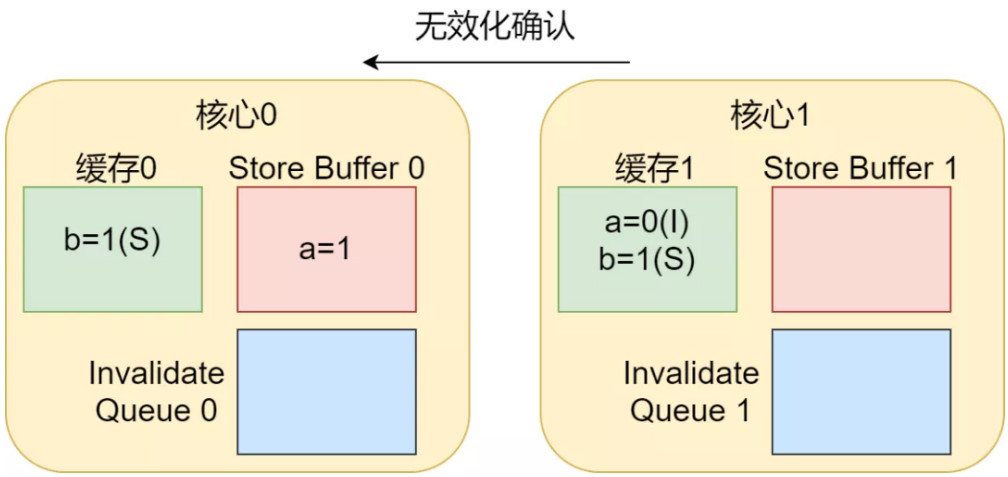
6

7. Kernel1执行assert(a==1)，由于Kernel1缓存中a的值为0，并且是有效的，所以断言出错；



7

8. Kernel1终于收到了第一步Kernel0发送的Invalidate了，赶紧将缓存区的a==1置为invalid，但是为时已晚。



8

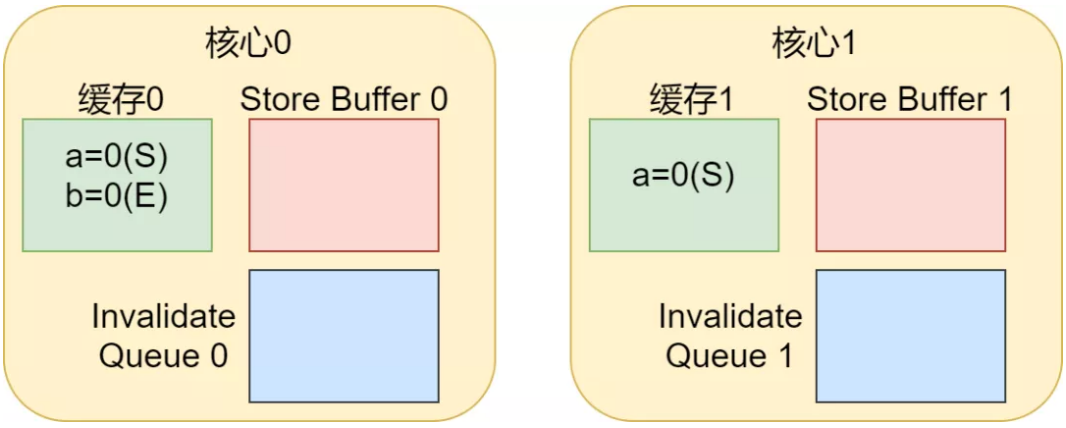
所以我们看到，这个例子出错的原因完全是由Store Buffer这个结构引发的。如果规定将Store Buffer中数据完全刷入到缓存，才能执行对应变量写操作的话，该错误也能避免了。

例子2：关于Invalidate Queue带来的错误，同样假设CPU有两个核心，Kernel0表示第一个核心，Kernel1表示第二个核心。

```
...
public void foo(){
    a=1;
    b=1;
}
public void bar(){
    while(b==0) continue;
    assert(a==1):"a has a wrong value!";
}
...
```

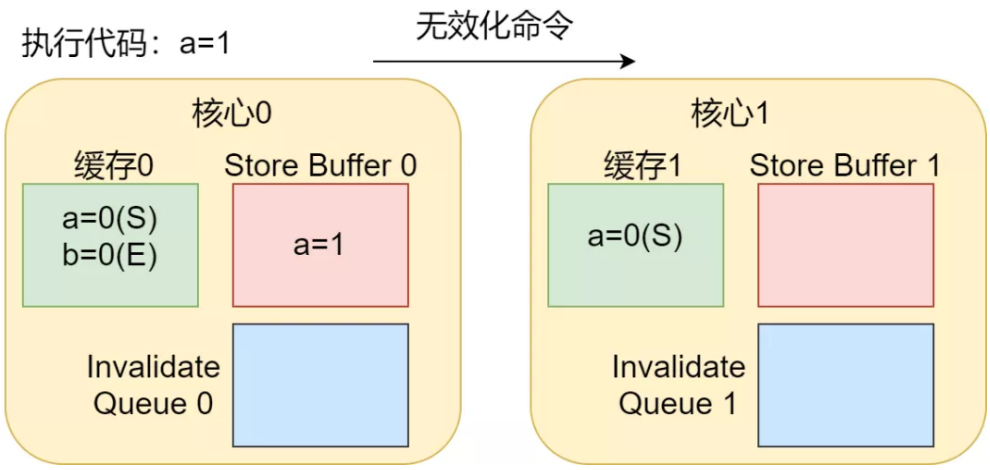
Kernel0执行foo()函数，Kernel1执行bar()函数，猜猜看这次断言会出错吗?

假设在初始情况是这样的：变量a的值在Kernel0和Kernel1对应的缓存区都有，状态为S（共享），初值为0，变量b的值是0，状态为E（独占），只存在于Kernel1对应的缓存区，不存在Kernel0对应的缓存区。假设Kernel0执行foo()函数，Kernel1执行bar()函数时，程序执行过程如下：



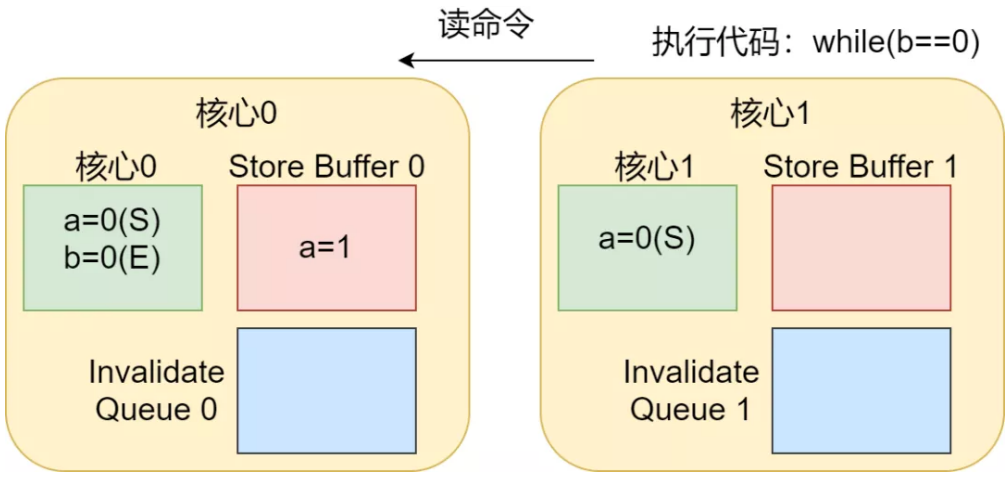
0

1. Kernel0执行a=1，此时由于a变量被更改了，需要给Kernel1发送无效化命令，并且将a的值存储在Kernel0的Store Buffer中；



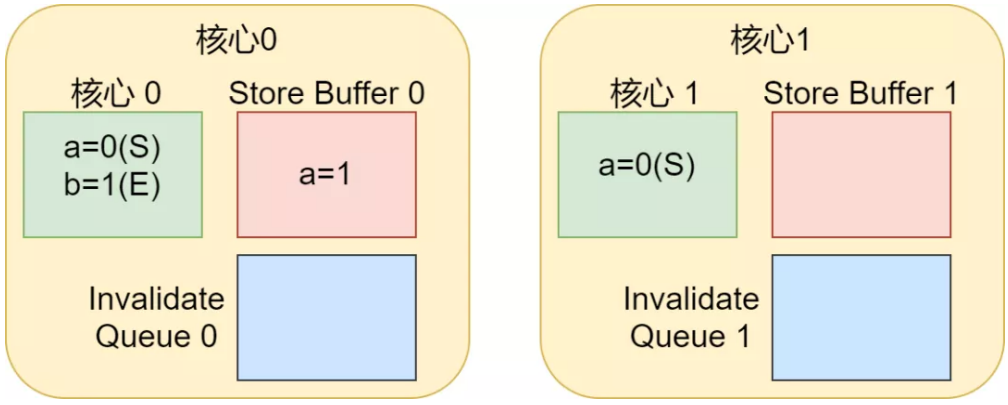
1

2. Kernel1执行while(b==0)，由于Kernel1对应的缓存不包含变量b，它需要发出一个读命令；



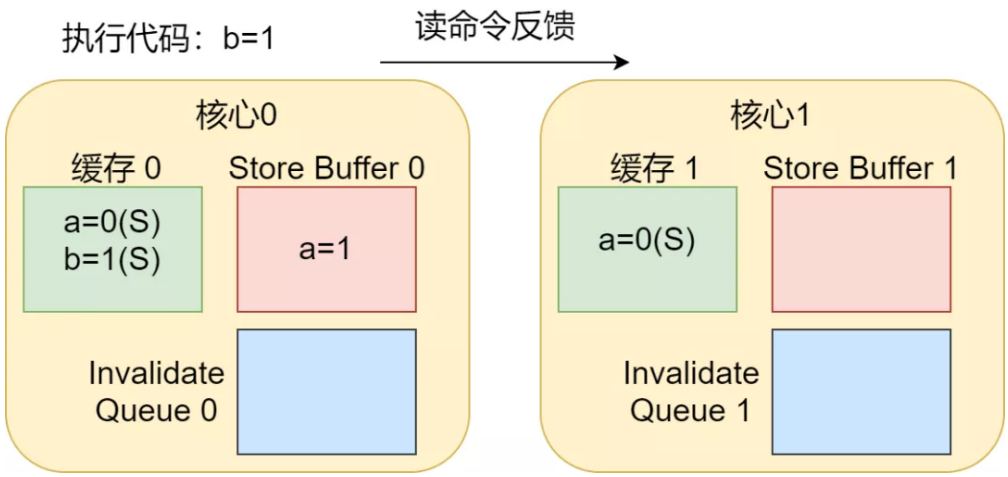
2

3. Kernel0执行b=1，由于是独占的，因此它直接更改自己缓存的值；



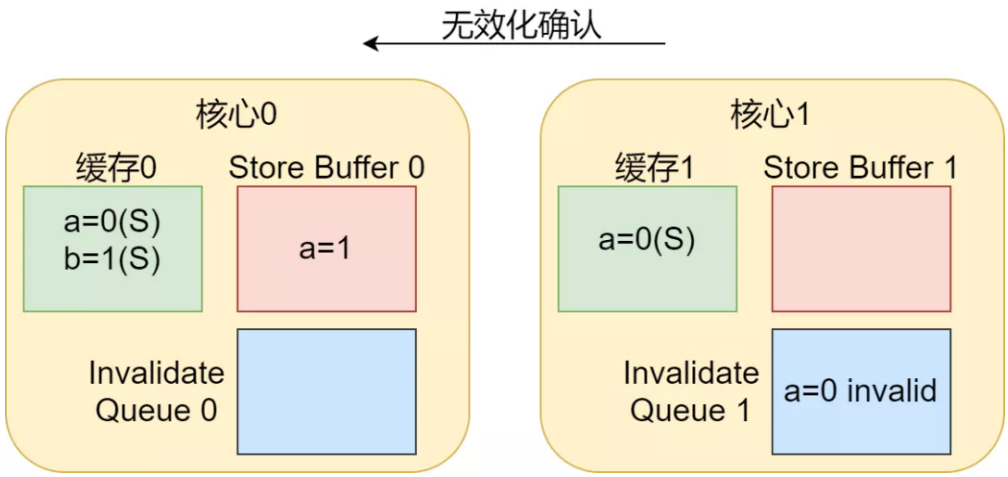
3

4. Kernel0收到读命令，将最新的b的值发送给Kernel1，并且将变量b的状态改变为S（共享）；



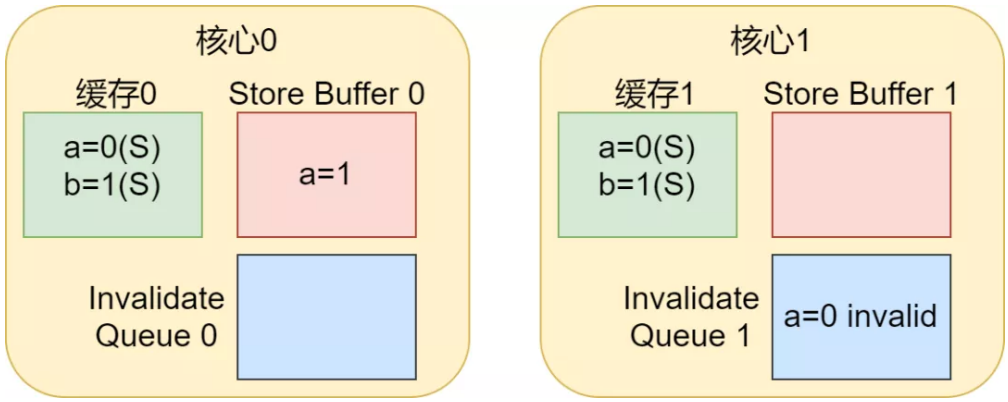
4

5. Kernel1收到Kernel0在第一步发的无效化命令，将这个命令存到Invalidate Queue中，打算之后再处理，并且给Kernel0回确认响应；



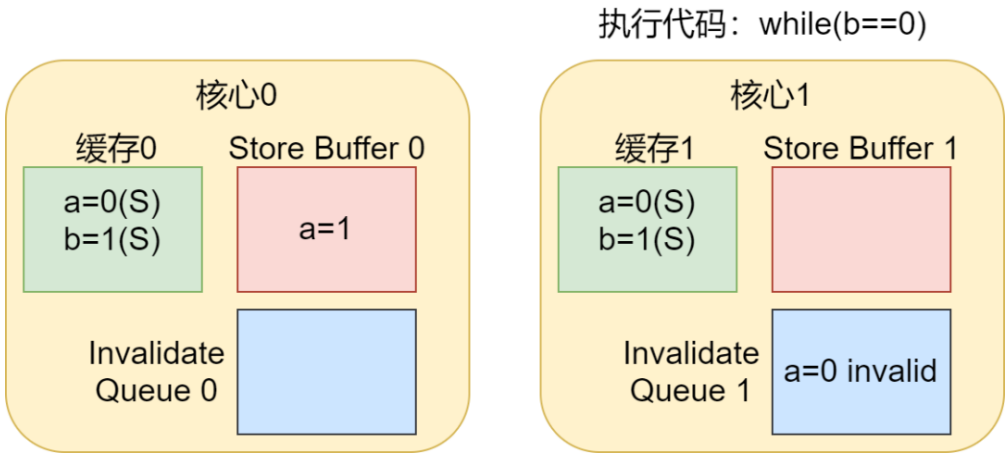
5

6. Kernel1收到包含b值的读命令反馈，把该值存到自己缓存下；



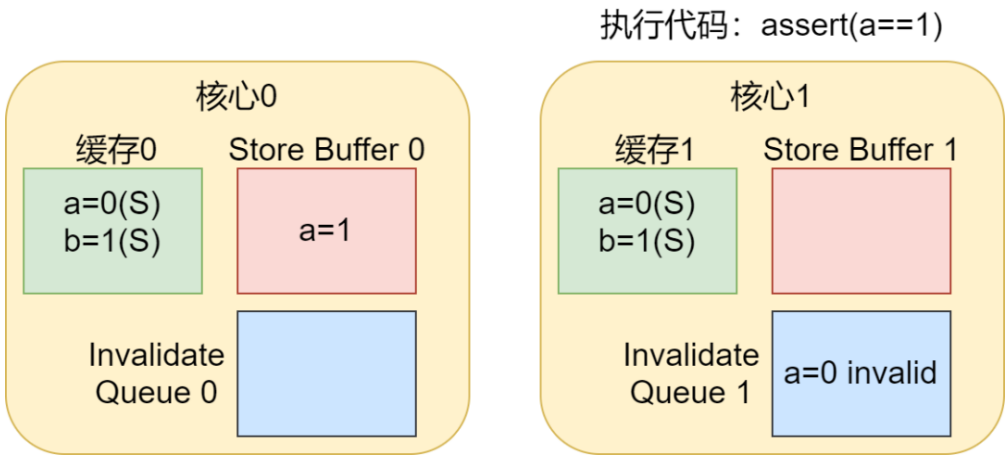
6

7. Kernel1收到b的值之后，打破while循环；



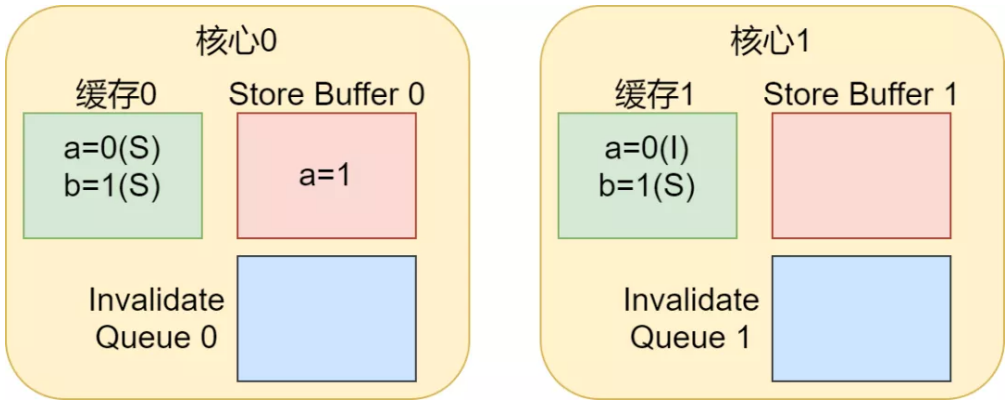
7

8. Kernel1执行assert(a==1)，由于此时Invalidate Queue中的无效化a=0这个缓存值还没执行，因此Kernel1会接着用自己缓存中的a=1这个缓存值，这就出现了问题；



8

9. Kernel1开始执行Invalidate Queue中的命令，将a=0这个缓存值无效化。但这时已经太晚了。



9

所以我们看到，这个例子出错的原因完全是由Invalidate Queue这个结构引发的。如果规定将Invalidate Queue中命令完全处理完，才能执行对应变量读操作的话，该错误也能避免了。

内存屏障

既然刚刚我们遇到了问题，那如何改正呢？这里就终于到了今天的重头戏，内存屏障了。内存屏障简单来讲就是一行命令，规定了某个针对缓存的操作。这里我们来看一下最常见的写屏障和读屏障。

1. 针对Store Buffer：核心在后续变量的新值写入之前，把Store Buffer的所有值刷新到缓存；核心要么就等待刷新完成后写入，要么就把后续的后续变量的新值放到Store Buffer中，直到Store Buffer的数据按顺序刷入缓存。这种也称为内存屏障中的写屏障（Store Barrier）。
2. 针对Invalidate Queue：执行后需等待Invalidate Queue完全应用到缓存后，后续的读操作才能继续执行，保证执行前后的读操作对其他CPU而言是顺序执行的。这种也称为内存屏障中的读屏障（Load Barrier）。

---

volatile中的内存屏障

对于JVM的内存屏障实现中，也采取了内存屏障。JVM的内存屏障有四种，这四种实际上也是上述的读屏障和写屏障的组合。我们来看一下这四种屏障和他们的作用：

1. LoadLoad屏障：对于这样的语句

```
第一大段读数据指令;
LoadLoad;
第二大段读数据指令;
```

LoadLoad指令作用：在第二大段读数据指令被访问前，保证第一大段读数据指令执行完毕

1. StoreStore屏障：对于这样的语句

```
第一大段写数据指令;
StoreStore;
第二大段写数据指令;
```

StoreStore指令作用：在第二大段写数据指令被访问前，保证第一大段写数据指令执行完毕

1. LoadStore屏障：对于这样的语句

```
第一大段读数据指令;
LoadStore;
第二大段写数据指令;
```

LoadStore指令作用：在第二大段写数据指令被访问前，保证第一大段读数据指令执行完毕。

1. StoreLoad屏障：对于这样的语句

```
第一大段写数据指令;
StoreLoad;
第二大段读数据指令;
```

StoreLoad指令作用：在第二大段读数据指令被访问前，保证第一大段写数据指令执行完毕。

针对volatile变量，JVM采用的内存屏障是：

1. 针对volatile修饰变量的写操作：在写操作前插入StoreStore屏障，在写操作后插入StoreLoad屏障；
2. 针对volatile修饰变量的读操作：在每个volatile读操作前插入LoadLoad屏障，在读操作后插入LoadStore屏障；

通过这种方式，就可以保证被volatile修饰的变量具有线程间的可见性和禁止指令重排序的功能了。

总结

讲了这么多，我们来总结一下。

volatile关键字保证了两个性质：

- 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。
- 有序性：对一个volatile变量的写操作，执行在任意后续对这个volatile变量的读操作之前。

单单缓存一致性协议无法实现volatile。

缓存一致性可以通过Store Buffer和Invalidate Queue两种结构进行加速，但这两种方式会造成一系列不一致性的问题。

因此后续提出了内存屏障的概念，分为读屏障和写屏障，以此修正Store Buffer和Invalidate Queu产生的问题。

通过读屏障和写屏障，又发展出了LoadLoad屏障，StoreStore屏障，LoadStore屏障，StoreLoad屏障JVM也是利用了这几种屏障，实现volatile关键字。

这篇相对于林哥的可能更详细一点https://mp.weixin.qq.com/s/kAgjXFogJnSF3Os84g1Aqg

例子二中 变量b 应该是只存在于kernel0的缓存区，文中这里笔误了

首先，感谢楼主的分享，真的学习到了。另外，就是想跟楼主确认以下情况： * Store Buffer的举例，第8步，是不是应该是“a==0置为无效”； * Invalidate Queue的举例，假设说明中，是不是应该是“只存在于Kernel0对应的缓冲区，不存在Kernel1对应的缓冲区”； * Invalidate Queue的举例，第8步，是不是应该是“因此Kernel1会接着用自己缓存中的a=0这个缓存值”；

1.这里确实是笔误了，应该是a==0无效 2.这边有一个读者已经提出来了，已改正 3.这里确实是笔误了

有几个疑问： 1、Store Buffer和Invalidate Queue是为了加速，后面又加上内存屏障，是否就达不到加速的效果了？因为屏障就是等待啊！ 2、多线程共享的变量，是否都要加上volatile修饰？能否讲讲在什么情况下才需要使用volatile修饰？ 3、例子1中，根本原因不是因为kernel1没收到kernel0发出的Invalidate指令导致的？与改变量是读还是写有什么关系？ 4、同样的例子1中，根本原因不是因为kernel1没执行kernel0发出的Invalidate指令导致的？与改变量是读还是写有什么关系？ 以上疑问请多多指教哈

\1. 加了内存屏障，确实达不到加速效果。所以如果是对变量一致性要求没那么高的情况下，不需要用volatile关键字修饰，因此没有内存屏障了。

\2. 关于这个问题，推荐两篇讲的比较好的博文：

\3. kernel1收到kernel0的invalidate了，只不过比较晚，底层的设计不能保证kernel0的invalidate发出后，kernel1能瞬间收到，会有一定的延迟。invalidate操作与改变量未来是用于读操作还是写操作没有什么关系。

\4. 关于invalidate的执行，确实执行了，只不过执行的比较晚 感谢读着的认真阅读和评论！