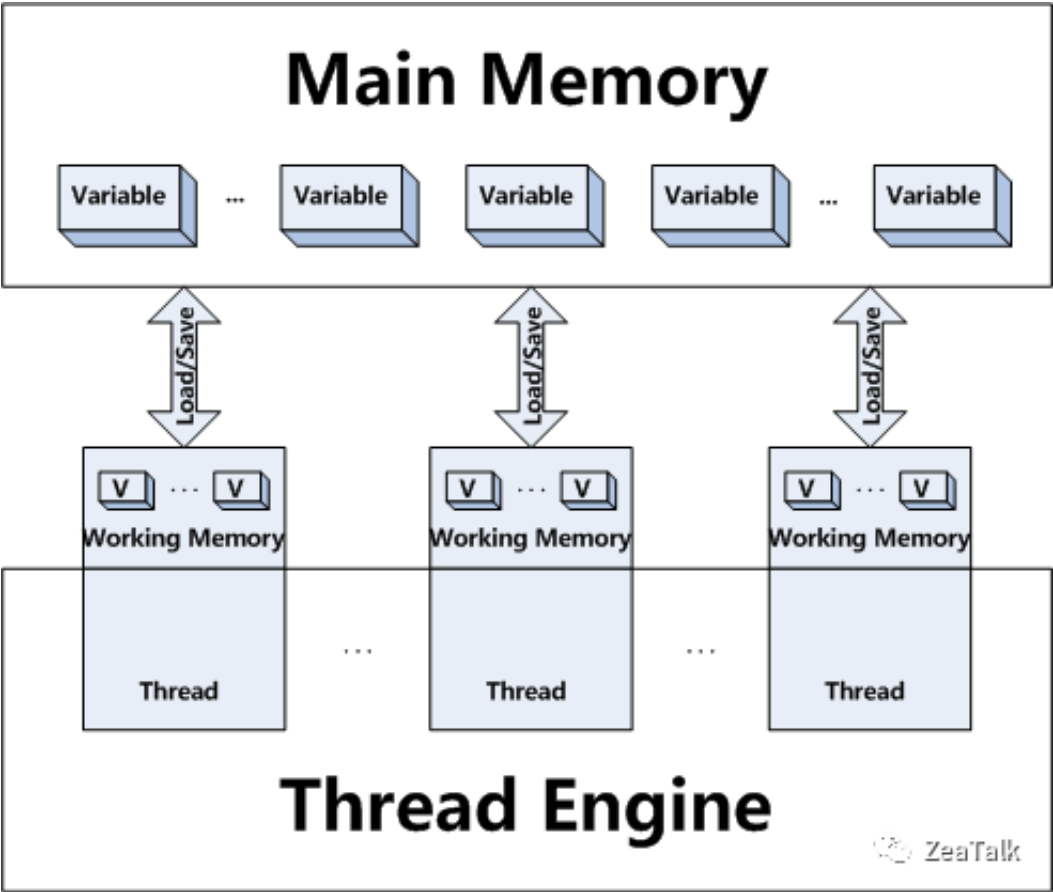


写在前面



Java 内存模型简称 JMM，其实是一个抽象模型，并非具体的存在。如上图想表达的，线程的读写都是直接跟 work memory 打交道，CPU 会通过一种机制将数据 flush 到 main memory 中。那么在 flush 发生之前，不同 CPU 之间就会出现可见性的问题，这很好理解。

熟悉 Java 的大概都知道 **volatile** 关键字是用来解决内存可见性和指令重排序问题的，我们往往知道这两点，但却不知道其中的关联。关于 volatile 也有很多人误认为这是 Java 的特色，其实不然，这涉及到了很多方面，包括 CPU 缓存以及编译器等等，其他语言照样会遇到类似的问题，照样也会提供类似 volatile 的解决方案。然而探究起 volatile 的细节还是有很多不清晰的地方，这些不清晰的地方才是我探究的重点。

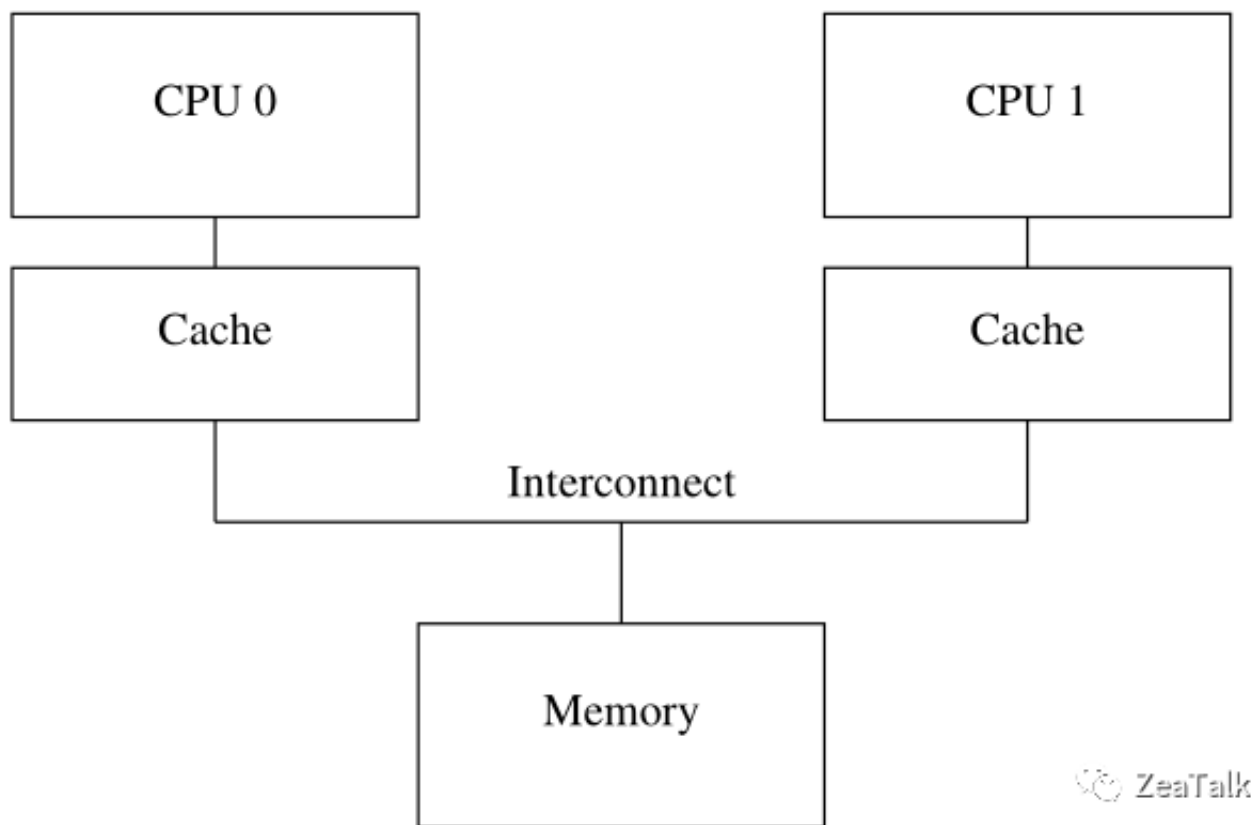
首先我们来看一个问题，CPU 为什么不能按代码逻辑的顺序执行指令？


带着上面的问题，我有幸拜读了 Paul E. McKenney 写的《Memory Barriers: a Hardware View for Software Hackers》，大神写的论文极具亲和力，建议大家多花时间阅读一下。下文就是读完论文后整理的思路，我尽力把一层一层的关系拨开，阐述清楚。

(注：通篇讨论都是基于 SMP（对称多处理器）的架构。)

CPU缓存

一个极简的 CPU 缓存结构是这样的（先不考虑多级缓存）。



 ZeaTalk

看到这个图你是否会想起什么？这似乎就是 JMM 抽象概念的具体写照。显然这里的 Cache 就是所谓的“working memory”，这里的 Memory 就是所谓的“main memory”，不同的 CPU 核心对应不同线程。

CPU 的读（load）实质上就是从缓存中读取数据到寄存器（register）里，在多级缓存的架构中，如果缓存中找不到数据（cache miss），就会层层读取二级缓存三级缓存，一旦所有的缓存里都找不到对应的数据，就要去内存里寻址了。寻址到的数据首先放到寄存器里，其副本会驻留到 CPU 的缓存中。

CPU 的写（store）也是针对缓存作写入。并不会直接和内存打交道，而是通过某种机制实现数据从缓存到内存的写回（write back）。

似乎很麻烦，可为什么需要引入缓存呢？

那是因为相比 CPU 而言，内存的频率实在太低了，况且内存这么大，寻址需要的时间 CPU 真的等不起呀。CPU 一般情况下是整个计算机组成中，频率最高的存在，为了发挥 CPU 的长处就得尽量避免等待。目前的存储设备中 I/O 存储速度由快及慢应该是这样的：

**寄存器 > CPU 缓存（纳秒级） > 内存（微秒级） > 磁盘**

所以 CPU 直接把运算结果放到寄存器上是最具效率的，但寄存器资源十分宝贵，只能存单条或多条指令的中间结果或临时存放的内存地址，很快又会被下一次计算的结果挤占掉。因此 CPU 缓存才是 CPU 存放最终计算结果的的最佳介质。

但引入缓存有副作用，要知道，同一份数据在不同地方保存，数据一致性怎么保证呢？不过 CPU 设计者给出了个通用的方案，MESI。

### **MESI 缓存一致性协议**

之所以叫 MESI，是因为这套方案把一个缓存行（cache line）区分出四种不同的状态标记，他们分别是 Modified、Exclusive、Shared 和 Invalid。这四种状态分别具备一定的意义：

**Modified**：某个缓存行被某个CPU独占，这个CPU对该缓存行做出了修改以后标记的，该缓存行指向的值在该CPU上的副本是全系统最新的；

**Exclusive**：同样也表示某个缓存行被某个CPU独占，但是区别在于，CPU 执行了写回操作，将缓存中最新数据刷到内存里了；

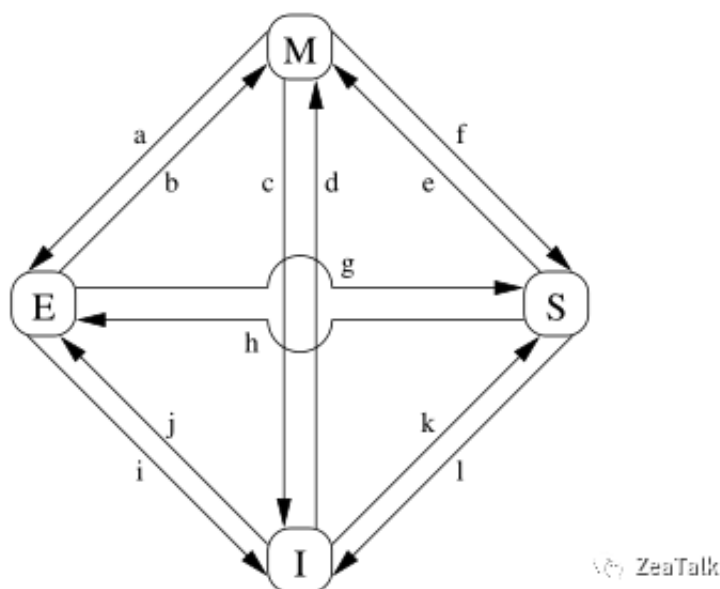
**Shared**：顾名思义，这块缓存行的副本被多个CPU持有；

**Invalid**：缓存行无效，可以当作空缓存行。

这些状态本身是静态的，那么动态来看，又是如何产生状态变化的呢？

首先不同CPU之间也是需要沟通的，这里的沟通是通过在消息总线上传递message实现的。这些在总线上传递的消息有如下几种：

- **Read**：带上数据的物理内存地址发起的读请求消息；
- **Read Response**：Read 请求的响应信息，内部包含了读请求指向的数据；
- **Invalidate**：该消息包含数据的内存物理地址，意思是要让其他如果持有该数据缓存行的 CPU 直接失效对应的缓存行；
- **Invalidate Acknowledge**：CPU 对Invalidate 消息的响应，目的是告知发起 Invalidate 消息的CPU，这边已经失效了这个缓存行啦；
- **Read Invalidate**：这个消息其实是 Read 和 Invalidate 的组合消息，与之对应的响应自然就是一个Read Response 和一系列的 Invalidate Acknowledge；
- **Writeback**：该消息包含一个物理内存地址和数据内容，目的是把这块数据通过总线写回内存里。



这里列举了 12 种不同状态的转换方式。这里只挪列几个我认为比较重要的转换：

- **转换 a**：M to E，CPU 将缓存数据 writeback 到内存，而后状态变更；
- **转换 c**：M to I，CPU 已修改但还没来得及写回内存的缓存行 A，其状态为M，CPU 接收到关于A 的 Read Invalidate 请求后把 A 的数据响应给请求方，并将 A 缓存行失效；
- **转换 d**：I to M，CPU 原子性 read-modify-write 操作一个状态为 I 的缓存行后，状态可以变为 M。变为 M 之前会发出 Read Invalidate 消息，等待 Read Response 和 Invalidate Acknowledge 反馈后，上面的原子操作才算结束；

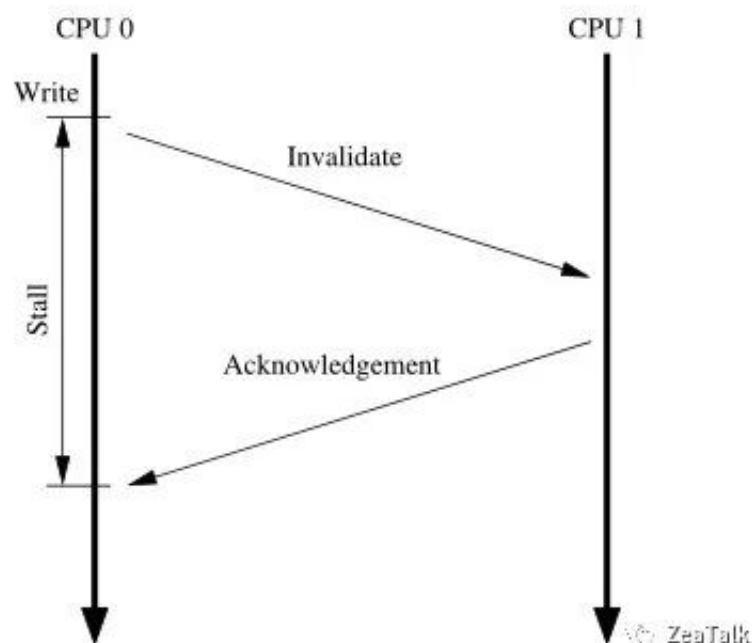
- **转换 e**: S to M, Shared的状态下, 某个CPU发起了对该缓存行的原子性 read-modify-write 操作, 但需要发出 Invalidate 消息, 并等待一系列的 Invalidate Acknowledge 反馈后才能对缓存行修改;
- **转换 f**: M to S, CPU 0 如若持有 M 状态缓存行, 则表示持有了这个数据的最新值, 此时如果 CPU 1 发送了 Read 消息, 那么 CPU 0 就会带上数据反馈一个 Read Response, 那么 M 就会切换成 S; (这里的问题是, 如果多个CPU均持有同一块缓存行, 且状态都为 M, 怎么处理? 目前设计上来说不会出现这种情况, CPU 真正修改数据以前会失效其他 CPU 的同一块缓存行, 在多个 CPU 存在竞争资源的情况下, 只有其中一个 CPU 能最终修改成功。)
- **转换 h**: 实质上是 S to M to E, 结合上文很容易理解;
- **转换 i**: E to I, CPU 0 独占某个数据行, 此时 CPU 1 对同个数据发起了 read-modify-write 的原子操作, CPU 0 持有的这个数据行自然会被失效了;

这些状态的变更都似乎都很容易理解, 我们来总结一些规律性的东西:

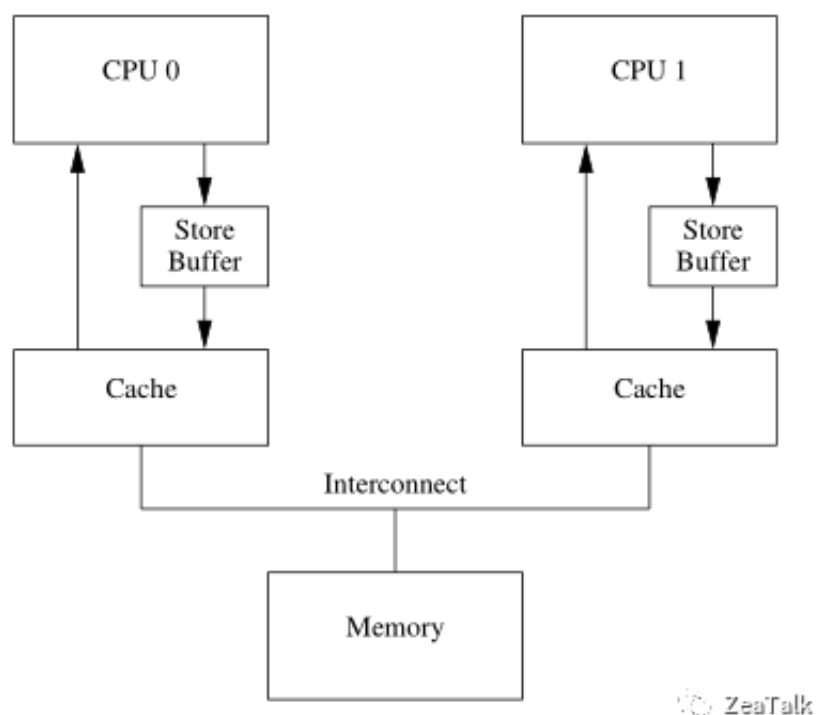
1. 一般情况下, CPU 在对某个缓存行修改之前务必得让其他 CPU 持有的相同数据缓存行失效, 这是基于 Invalidate Acknowledge 消息反馈来判断的;
2. 缓存行为 M 状态, 意味着该缓存行指向的物理内存里的数据, 一定不是最新;
3. 在修改变量之前, 如果CPU持有该变量的缓存, 且为 E 状态, 直接修改; 若状态为 S, 需要在总线上广播 Invalidate; 若CPU不持有该缓存行, 则需要广播 Read Invalidate。

## Store Buffers

这个极简的 CPU 缓存架构存在的问题, 当相当一部分 CPU 持有相同的数据时 (S 状态), 如果其中有一个 CPU 要对其进行修改, 则需要等待其他 CPU 将其共同持有的数据失效, 那么这里就会有空等期 (stall), 这对于频率很高的CPU来说, 简直不能接受!



解决方案就是 Store Buffers。



这是一个 CPU 在真正写入缓存之前的缓冲区，缓冲区作用在于 CPU 无需等待其他 CPU 的反馈，把要写入的数据先丢到 Store Buffer 中，自己可以去处理别的事情，避免了CPU的傻等。

### Store Forwarding

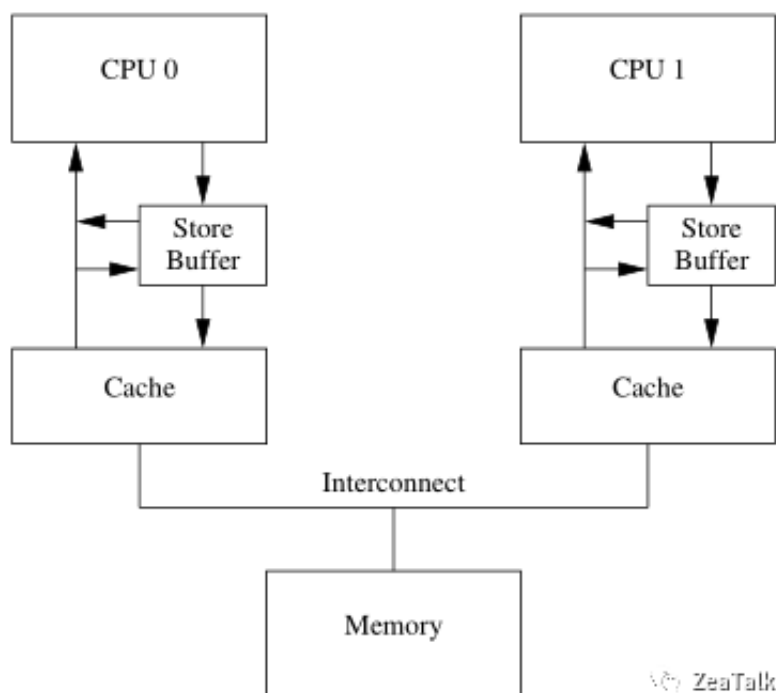
Store Buffer 有效减少了 CPU 写入缓存的空等时间窗口，但也引入了副作用，单个 CPU 在顺序执行指令的过程中，有可能出现，前面的已经执行写入变更，但对后面的代码逻辑不可见。这怎么理解呢？

假设 a, b 初始值为0：

```
1 | a=1
```

第一行a 的赋值在 b 的赋值之前，b 依赖 a的写入结果，若在执行 b 的赋值操作时， a 的写入变更还驻留在 Store Buffer 内，那么 b 无法察觉。此刻 b 会读取 a 的缓存行，那么 b 的最终计算结果便是 0 。这样看来，单个 CPU 的执行顺序性都无法保障，直接影响到了可见性，显然不可接受！

这里解决方案就是 Store Forwarding。



对于同一个 CPU 而言，在读取 a 变量的时候，如若发现 Store Buffer 中有尚未写入到缓存的数据 a，则直接从 Store Buffer 中读取。这就保证了，逻辑上代码执行顺序，也保证了可见性。

## Memory Barriers

通过 Store Forwarding 解决了单个 CPU 执行顺序性和内存可见性问题，但是在全局多 CPU 的环境下，这种内存可见性恐怕就很难保证了。

```
1 | void foo(void)
```

假设上面的 foo 方法被 CPU 0 执行，bar 方法被 CPU 1 执行，也就是我们常说的多线程环境。试想，即便在多线程环境下，foo 和 bar 如若严格按照理想的顺序执行，是无论如何都不会出现 assert failed 的情况的。但往往事与愿违，这种看似很诡异的且有一定几率发生的 assert failed，结合上面所说的 Store Buffer 就一点都不难理解了。

我们来还原 assert failed 的整个过程，假设 a,b 初始值为 0，a 被 CPU0 和 CPU1 共同持有，b 被 CPU0 独占；

1. CPU0 处理 a=1 之前发送 Invalidate 消息给 CPU1，并将其放入 Store Buffer，尚未及时刷入缓存；
2. CPU 0 转而处理 b=1，此时 b=1 直接被刷入缓存；
3. CPU 1 发出 Read 消息读取 b 的值，发现 b 为 1，跳出 while 语句；
4. CPU 1 发出 Read 消息读取 a 的值，发现 a 却为旧值 0，assert failed。

在日常开发过程中也是完全有可能遇到上面的情况，由于 a 的变更对 CPU1 不可见，虽然执行指令的时序没有真正被打乱，但对于 CPU1 来说，这造成了 b=1 先于 a=1 执行的假象，这种看是乱序的问题，通常称为“重排序”。当然上面所说的情况，只是指令重排序的一种可能。

**（由此可以看出，其实指令重排序和内存可见性是相辅相成的）**

解决办法不是没有，Memory Barrier（内存屏障）。借助内存屏障可以很好地保证了顺序一致性。

```
1 | void foo(void)
```

这个屏障可以理解为两条指令之间的栅栏(fence)，比如在上面的 foo 方法中，a 的赋值和 b 的赋值之间势必要执行这个栅栏。这个栅栏有什么用呢？

smp\_mb 首先会使得 CPU 在后续变量变更写入之前，把 Store Buffer 的变更写入 flush 到缓存；CPU 要么就等待 flush 完成后写入，要么就把后续的写入变更放到 Store Buffer 中，直到 Store Buffer 数据顺序刷到缓存。

CPU 的设计者兼顾性能和指令重排序之间做了权衡，认为其实在大多数场景下，多线程环境下的指令重排序和可见性问题是**可以接受的**，并且这有助于 CPU 发挥出该有的性能。如果真的有特殊需求，我们可以借助内存屏障来解决，虽然有一定的代码侵入性，但是这样的 tradeoff 是相当划算的。

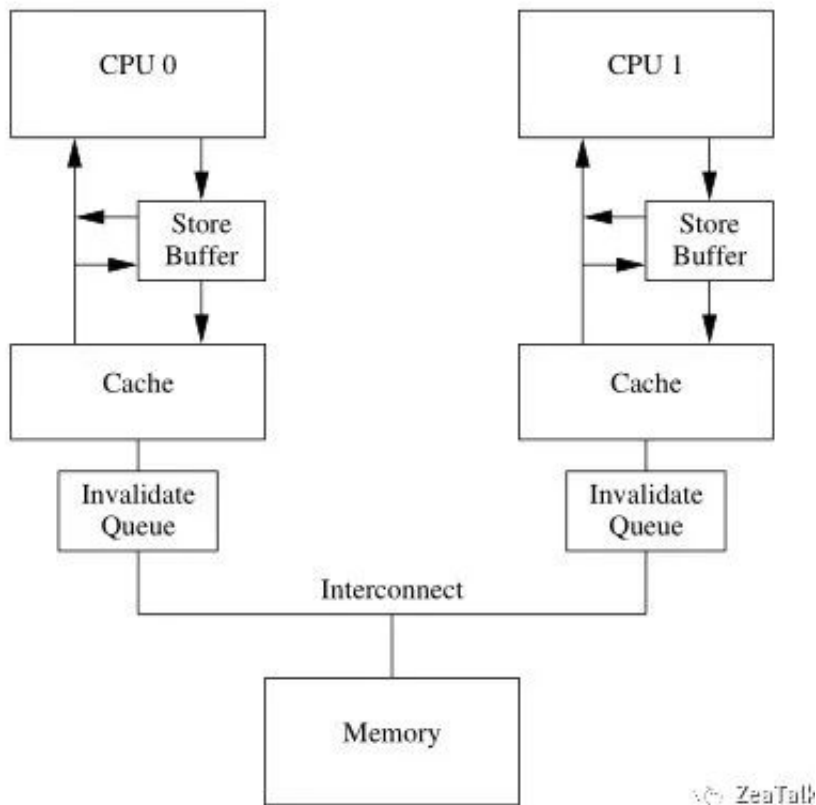
### Invalidate Queues

然而从目前设计看，还依然有问题。试想这么一个场景，CPU 写入一大串数据到 Store Buffer，而这些缓存行均被其他 CPU 持有，那么此时这个 CPU 需要等待一系列的 Invalidate Acknowledge 反馈后才能将这批数据 flush 到缓存行。

这里存在的问题是，Store Buffer 本身很小，如果写入变更指向的变量在 CPU 本地缓存中均是 cache miss 的情况下，变更数量超过了 Store Buffer 能承载的容量，CPU 依然需要等待 Store Buffer 排空后才能继续处理。尤其是执行 Memory Barrier 以后，无论本地缓存是否 cache miss，只要 Store Buffer 还有数据，所有的写入变更都要进入 Store Buffer。这就导致 CPU 依然存在的空等（stall）现象。

CPU 设计者的思路是，尽可能减少 invalidate ack 的时延，以减少 CPU 的无谓等待。目前的方案是，CPU 一旦收到 Invalidate 消息，先是会去缓存中标记该缓存状态为 I，标记完毕后发送 invalidate ack 到消息总线。那如果 CPU 接收到 invalidate 消息，立马反馈 invalidate ack，而 cache line 此时也并非强制要求马上失效，只要确保最终会失效即可，这样的思路是否可以呢？

可以，基于这个思路，Invalidate Queues 应运而生。



每个 CPU 都有一个 Invalidate Queue，用以把需要失效的数据物理地址存储起来，根据这个物理地址，我们可以对缓存行的失效行为“延后执行”。这样做的好处上面也说过，又一次释放了 CPU 的发挥空间，但依然有额外的副作用。继续来看上面的例子：


```
1 void foo(void)
```

引入 Invalidate Queue 后，assert failed 死灰复燃。我们来重现下：

假设 a,b 初始值为0，CPU0 执行 foo 方法，CPU1 执行 bar 方法，b 被CPU0 独占，a 则被 CPU0 和 CPU1 共同持有。

1. CPU0 执行 a=1，由于缓存行状态为 S，需要发送 Invalidate 消息到总线；
2. CPU1 接收到 Invalidate 消息，将数据内存地址放入 Invalidate Queue 后立马反馈 Invalidate Acknowledge；
3. CPU 0 收到反馈后，把 a=1 从 Store Buffer 刷到缓存后，执行 b=1，b 的新值 1 直接被写入到了 CPU 0 的缓存中；
4. CPU1 执行 while 语句，通过发送 Read 指令查询 b 的值，此时 b 为 1，跳出 while；
5. CPU1 执行 assert(a==1)，a 的 invalidate 信息还在 Invalidate Queue 中，CPU1 缓存中的 a 仍然是旧值 0，assert failed。

表象上看依然是指令执行顺序被打乱了，这似乎用 Memory Barrier 也有问题呀，解决方案就是要使用更多的

Memory Barrier 。 

来看代码

```
1 void foo(void)
```

不过这里的 smp\_mb 有更丰富的语义，除了与 Store Buffer 的交互外，一旦执行到 smp\_mb 指令，CPU 首先将本地 Invalidate Queue 的条目全部标记，**并且强制要求 CPU 随后的所有读操作，务必等待 Invalidate Queue 中被标记的条目真正应用到缓存后方能执行**。这就很好解决了上面的重排序问题，但同理，会带来一定程度的性能损耗。

### 读内存屏障 vs 写内存屏障

还有一个小问题，smp\_mb 包含的语义有些“重”，既包含了 Store Buffer 的 flush，又包含了 Invalidate Queue 的等待环节，但现实场景下，我们可能只需要与其中一个数据结构打交道即可。于是，CPU 的设计者把 smp\_mb 屏障进一步拆分，一分为二，smp\_rmb 称之为读内存屏障，smp\_wmb 称之为写内存屏障。他们分别的语义也相应做了简化：

- **smp\_wmb(StoreStore)**：执行后需等待 Store Buffer 中的写入变更 flush 完全到缓存后，后续的写操作才能继续执行，保证执行前后的写操作对其他 CPU 而言是顺序执行的；
- **smp\_rmb(LoadLoad)**：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；

(在上述特定的CPU架构下，StoreStore 和 LoadLoad 屏障完全够用)

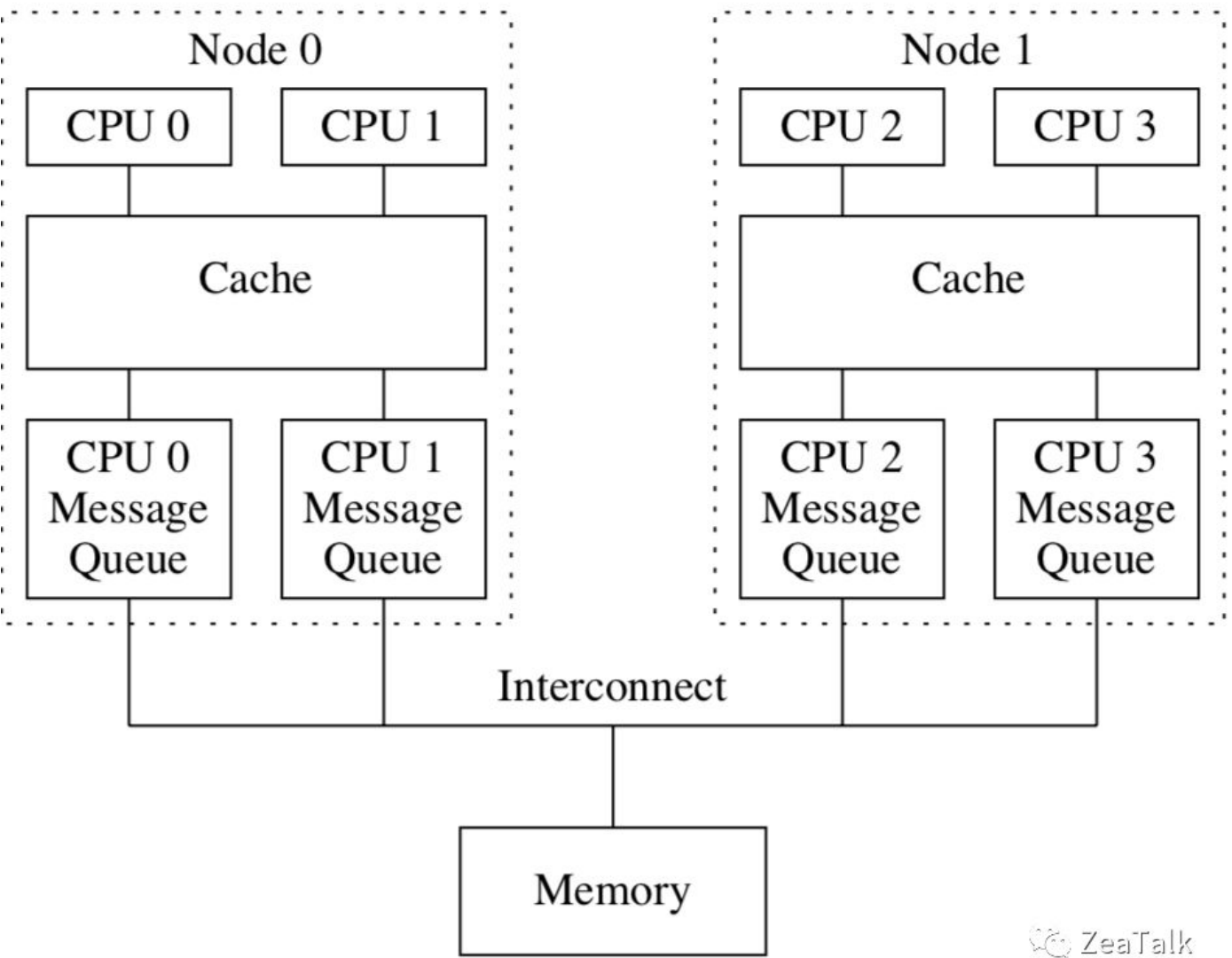


我们稍微对上面的例子再做修改就很容易理解了：

```
1 | void foo(void)
```

如果没有 `smp_wmb`，那么 `foo` 方法对于别的 CPU 而言，`a` 与 `b` 赋值语句的执行顺序是不确定的，可能会导致 `assert failed`；如果没有 `smp_rmb`，那么 `bar` 方法对于其他 CPU 而言，`b` 与 `a` 的 读取指令的执行顺序也是不确定的，也可能导致 `assert failed`。

**CPU 进一步演化--NUCA**



如图，某些 CPU 会维护一个消息队列，用以承载总线上大量的消息。这种架构引出新的问题，对于 `load A -> smp_mb -> store B` 这种情况，这里的屏障是否起作用呢？我们来看个复杂点的例子：

```
1 | void fool(void)
```

这是 Paul 在论文里留下的这个思考题，`a`，`b`，`c` 初始值均为 0，三个方法分别被 CPU0、CPU1 和 CPU2 执行，这里的 `assert` 一定能通过吗？

答案是不一定。

试着来分析下，我们讨论的内存屏障无论是 `smp_wmb` 还是 `smp_rmb`，都是针对 CPU 本地而非全局所有 CPU 而言的屏障逻辑。比如写屏障，是针对 CPU 本地的 Store Buffer，读屏障是针对 CPU 本地的 Invalidate Queue。目前的架构中引入了消息队列，假设 CPU 0 消息队列满载，CPU 1 消息队列空载，这样一来，`a=1` 变更在出消息队列之前就被阻塞，而 `c=1` 相当轻松地走出队列，这就导致 `c=1` 变更先于 `a=1` 抵达 CPU 2，因此 `assert failed`。

之前讨论的屏障针对的要么是 `load A -> smp_mb -> load B`，要么是 `store A -> smp_mb -> store B` 两种情况。由于额外的消息队列存在，对于 `foo2` 方法，正是 `load B -> smp_mb -> store C`，在这种 CPU 的架构下，读写两种内存屏障均不能起到作用。

我们再来看个更复杂的例子：

```
1 void foo1(void)
```

上面三个方法分别被 CPU0、CPU1 和 CPU2 执行，`foo2` 和 `foo3` 都需要依赖 `b` 值的变化才能继续执行。那么这里的 `assert` 能通过吗？

答案是可以的。

由于 `smp_wmb` 的存在，CPU0 中 `a` 和 `e` 的赋值以及 `a` 和 `b` 的赋值对其他 CPU 来说是顺序执行的；`foo2` 和 `foo3` 若能看到 `b` 的值为最新值 1 就能跳出循环，即便这里有消息队列，由于先进先出的特性，这里也是能够感知到 `a=1` 的，因为 `a` 先于 `b`，那么推理可得，断言通过。

但其实问题还没解决，之前也说过，StoreStore 和 LoadLoad 屏障在特定的 CPU 下是非常够用的，但是在这种特殊的架构下，我们可能需要更重的内存屏障以解决一些特定问题。

Java 中的 volatile

回到 Java 语言，JVM 是如何实现自己的内存屏障的？抽象上看 JVM 涉及到的内存屏障有四种：

- **LoadLoad**：两个 Load 操作之间内存屏障，`smp_rmb` 就是典型实现；
- **StoreStore**：两个 Store 操作之间的内存屏障，`smp_wmb` 典型实现；
- **LoadStore**：在 Load 操作和 Store 操作之间的内存屏障；
- **StoreLoad**：在 Store 操作和 Load 操作之间的内存屏障。

在日常开发中，这些屏障的生成逻辑我们可以用一个表总结：

内存屏障	第二项操作			
第一项操作	普通读	普通写	volatile读 同步块入口	volatile写 同步块出口
普通读				LoadStore
普通写				StoreStore
volatile读 同步块入口	LoadLoad	LoadStore	LoadLoad	LoadStore
volatile写 同步块出口			StoreLoad	StoreStore

1. 可以看到默认情况下，普通读写随机组合都是不会出现内存屏障的；
2. volatile 读 与之前的普通读写操作之间不存在内存屏障；
3. volatile 写 与之后的普通读写操作之间不存在内存屏障；

4. 同步块入口 (monitor enter) 和 volatile 读 等价;
5. 同步块出口 (monitor exit) 和 volatile 写 等价;

JVM 是如何分别插入上面四种内存屏障到指令序列之中的呢? 这里的设计相当巧妙。

对于 volatile 读 or monitor enter

```
1 | int t = x; // x 是 volatile 变量
```

对于 volatile 写 or monitor exit

```
1 | <other ops>
```

借助编译器分别对上面两种情况将内存屏障插入到指令序列中。这种看似简洁不拖沓的实现，堪称精妙。

### x86 架构下的实现

上面讨论的四种内存屏障只是抽象层面的，实际上这四种内存屏障在不同平台下的具体映射的指令是不一样的，而这也恰恰是 JVM 跨平台的优势所在。我们接下来着重讨论 X86 平台下，Java volatile 关键字是如何实现防止指令重排序的。

Processor	LoadStore	LoadLoad	StoreStore	StoreLoad	Data dependency orders loads?	Atomic Conditional	Other Atomics	Atomics provide barrier?
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	combine with st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	hwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only
pa-risc	no-op	no-op	no-op	no-op	yes	build from ldcw	ldcv	(N/A)

根据上面表格我们可以看到 x86 平台下，只有 StoreLoad 才有具体的指令对应，而其他三个屏障均是 no-op (空操作)。也就是说，在 x86 的架构下 volatile 内存屏障的生成策略是这样的：

内存屏障	第二项操作			
第一项操作	普通读	普通写	volatile读 同步块入口	volatile写 同步块出口
普通读				
普通写				
volatile读 同步块入口				
volatile写 同步块出口			StoreLoad	

然而，关于 StoreLoad 又有三个具体的指令对应，分别是 mfence、cpuid、以及 locked insn，他们都能很好地实现 StoreLoad 的屏障效果。但毕竟不可能同时用三种指令，这里可能意思是，三种均能达到效果，具体实现交由 JVM 设计者决断。

我们来看看基于 Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, mixed mode) 在 x 86 架构下的 StoreLoad 究竟对应什么指令？

首先写一段简单的代码：

```
1 /**
```

然后我们借助 hsdish（需额外增加参数 -XX:+PrintAssembly）观察 JIT 编译器生成的汇编指令（机器码人类无法阅读），这里千万别忘了禁用解释执行 -Xcomp。

可以看到这里的 StoreLoad 用到的具体指令是

```
1 lock addl $0x0, (%rsp)
```

简单来说，这句指令的作用就是，等待 volatile 变量的写操作对所有的线程可见以后，才继续执行后续指令。假设声明 volatile 变量 a，这个指令具体语义是：

1. 执行 a 的写操作后执行到 StoreLoad 内存屏障；
2. 发出 Lock 指令，锁总线 或 a 的缓存行，那么其他 CPU 不能对已上锁的缓存行有任何操作；
3. 让其他 CPU 持有的 a 的缓存行失效；
4. 将 a 的变更写回主内存，保证全局可见；
5. 上面三步执行完后，该 CPU 方可执行后续操作。

原则上讲 StoreLoad 开销是这四种屏障中最大的，但也兼具其余三种的屏障效果。

## 编译器的重排序

我用很大的篇幅解释了 SMP 系统运行过程中可能出现重排序问题，但是忽略了编译器也有可能是指令重排序的“元凶”。为什么这么说？

我们以 Java 为例，在某些场景下，JIT 编译器为了更好的执行效率，会重新规划出栈和入栈的执行时机，举个例子：

```
1 public void foo() {
```

这个场景下，如果按照编码顺序执行就会显得比较“蠢”，明明先判断 a 再判断 b，在判断 a 之前 b 入局部变量表是无意义的，因为有可能 if 判断不通过，那么 b 根本就用不上，还浪费了寄存器的资源。因此编译器很可能会优化成这样：

```
1 public void foo() {
```

那么这里的指令顺序就被打乱了。

### 为什么 **volatile** 写不具备原子性？

volatile 变量写入并不保证线程安全，也不具备原子性。原因很简单，在执行内存屏障之前，不同 CPU 依旧可以对同一个缓存行持有，一个 CPU 对同一个缓存行的修改不能让另一个 CPU 及时感知，因此出现并发冲突。线程安全还是需要用锁来保障，锁能有效的让 CPU 在同一个时刻独占某个缓存行，执行完并释放锁后，其他 CPU 才能访问该缓存行。

锁既能保证线程安全又能保证内存可见，而 volatile 只能保证内存可见。

### 总结

我们理解了 JMM 模型的具体细节，从 CPU 缓存架构到内存屏障，再到 Java 中 volatile 关键字的处理细节，无一不感叹其中奥妙。为什么代码不能按照编码预定的顺序执行呢？归根结底包含两个方面，一来是 CPU 为了执行效率和性能，接受了多核 CPU 下的代码乱序，这是一种设计妥协；二来编译器为了执行效率和寄存器的利用效率，避免不合理的代码逻辑，也会对编码指令重排序。

经过这一大篇幅的整理，我似乎可以理解设计者的良苦用心。至少在我刚接触 Java 内存模型的时候，甚至理解不到指令重排序对内存可见性的影响。显然，CPU 本身的设计和编译器的有意为之才是真正的内因，重排序是呈现给开发者的表象，而内存可见性则是指令乱序的另一种表现，甚至我认为重排序和内存可见性在某些场景下，其实是一回事。

CPU 在不断追求频率的同时，其架构方案也在不停演化，不管是 Store Buffer 还是 Invalidate Queue，甚至包括 Message Queue，这些组件的引入无疑是在一步一步摆脱 CPU 性能的束缚。频率越高越不能接受无谓的 I/O 等待，解决无谓的等待就需要引入额外的高速的承载元件加以缓冲，引入额外的承载元件会进一步增加数据不一致的可能，这种情况就是本篇所讨论的指令乱序和内存可见性的问题，解决这些问题就需要引入内存屏障，内存屏障本身多种多样，在不同平台下甚至有不同的表现，使用这些内存屏障势必又会制约 CPU 的性能发挥，而这就需要这些开发者因场景而异，做出权衡。

参考资料：

<https://shipilev.net/blog/2014/jmm-pragmatics/>

《Memory Barriers: a Hardware View for Software Hackers》

<https://shipilev.net/blog/2014/on-the-fence-with-dependencies>

<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>

<https://www.scss.tcd.ie/jeremy.jones/CS4021/locks.pdf>

<https://stackoverflow.com/questions/4232660/which-is-a-better-write-barrier-on-x86-lockaddl-or-xchgl>

<https://blogs.oracle.com/dave/instruction-selection-for-volatile-fences-%3a-mfence-vs-lock%3aadd>