内存屏障

#技术/杂项

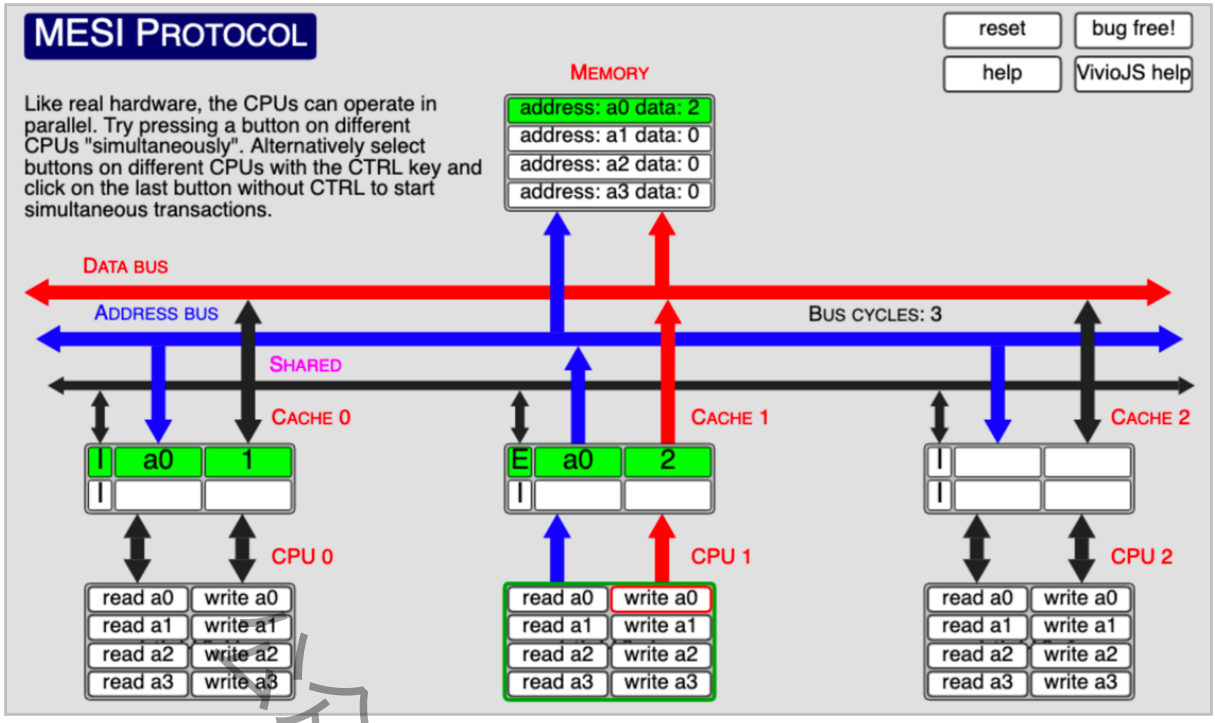
需要先了解 CPU Cache 基础。

MESI

MESI 的词条在这里：MESI protocol - Wikipedia，它是一种缓存一致性维护协议。MESI 表示 Cache Line 的四种状态，`modified`, `exclusive`, `shared`, `invalid`。

- `modified`：CPU 拥有该 Cache Line 且将其做了修改，CPU 需要保证在重用该 Cache Line 存其它数据前，将修改的数据写入主存，或者将 Cache Line 转交给其它 CPU 所有；
- `exclusive`：跟 `modified` 类似，也表示 CPU 拥有某个 Cache Line 但还未来得及对它做出修改。CPU 可以直接将里面数据丢弃或者转交给其它 CPU
- `shared`：Cache Line 的数据是最新的，可以丢弃或转交给其它 CPU，但当前 CPU 不能对其进行修改。要修改的话需要转为 `exclusive` 状态后再进行；
- `invalid`：Cache Line 内的数据为无效，也相当于没存数据。CPU 在找空 Cache Line 缓存数据的时候就是找 `invalid` 状态的 Cache Line；

有个超级棒的可视化工具，能看到 Cache 是怎么在这四个状态之间流转的：VivioJS MESI animation help。Address Bus 和 Data Bus 都是所有 CPU 都能监听到变化。比如 CPU 0 要读数据会把请求先发去 Address Bus，Memory 和其它 CPU 都会收到这次请求。Memory 通过 Data Bus 发数据时候也是所有 CPU 都会收到数据。我理解这就是能实现出来 Bus snooping的原因。另外这个工具上可以使用鼠标滚轮上下滚，看每个时钟下数据流转过程。



后续内容以及图片大多来自 Is Parallel Programming Hard, And, If So, What Can You Do About It? 这本书的附录 C。因为 MESI 协议本身非常复杂，各种状态流转很麻烦，所以这本书里对协议做了一些精简，用比较直观的方式来介绍这个协议，好处是让理解更容易。如果想知道协议的真实样貌需要去看上面提到的 WIKI。

协议

- Read: 读取一条物理内存地址上的数据；
- Read Response： 包含 `Read` 命令请求的数据结果，可以是主存发来的，也可以是其它 CPU 发来的。比如被读的数据在别的 CPU 的 Cache Line 上处于 `modified` 状态，这个 CPU 就会响应 `Read` 命令
- Invalidate：包含一个物理内存地址，用于告知其它所有 CPU 在自己的 Cache 中将这条地址对应的 Cache Line 清理；
- Invalidate Acknowledge：收到 `Invalidate`，在清理完自己的 Cache 后，CPU 需要回应 `Invalidate Acknowledge`；
- Read Invalidate：相当于将 `Read` 和 `Invalidate` 合起来发送，一方面收到请求的 CPU 要考虑构造 `Read Response` 还要清理自己的 Cache，完成后回复 `Invalidate Acknowledge`，即要回复两次；
- Writeback：包含要写的数据地址，和要写的数据，用于让对应数据写回主存或写到某个别的地方。

发起 `Writeback` 一般是因为某个 CPU 的 Cache 不够了，比如需要新 Load 数据进来，但是 Cache 已经满了。就需要找一个 Cache Line 丢弃掉，如果这个被丢弃的 Cache Line 处于

`modified` 状态，就需要触发一次 `WriteBack`，可能是把数据写去主存，也可能写入同一个 CPU 的更高级缓存，还有可能直接写去别的 CPU。比如之前这个 CPU 读过这个数据，可能对这个数据有兴趣，为了保证数据还在缓存中，可能就触发一次 `Writeback` 把数据发到读过该数据的 CPU 上。

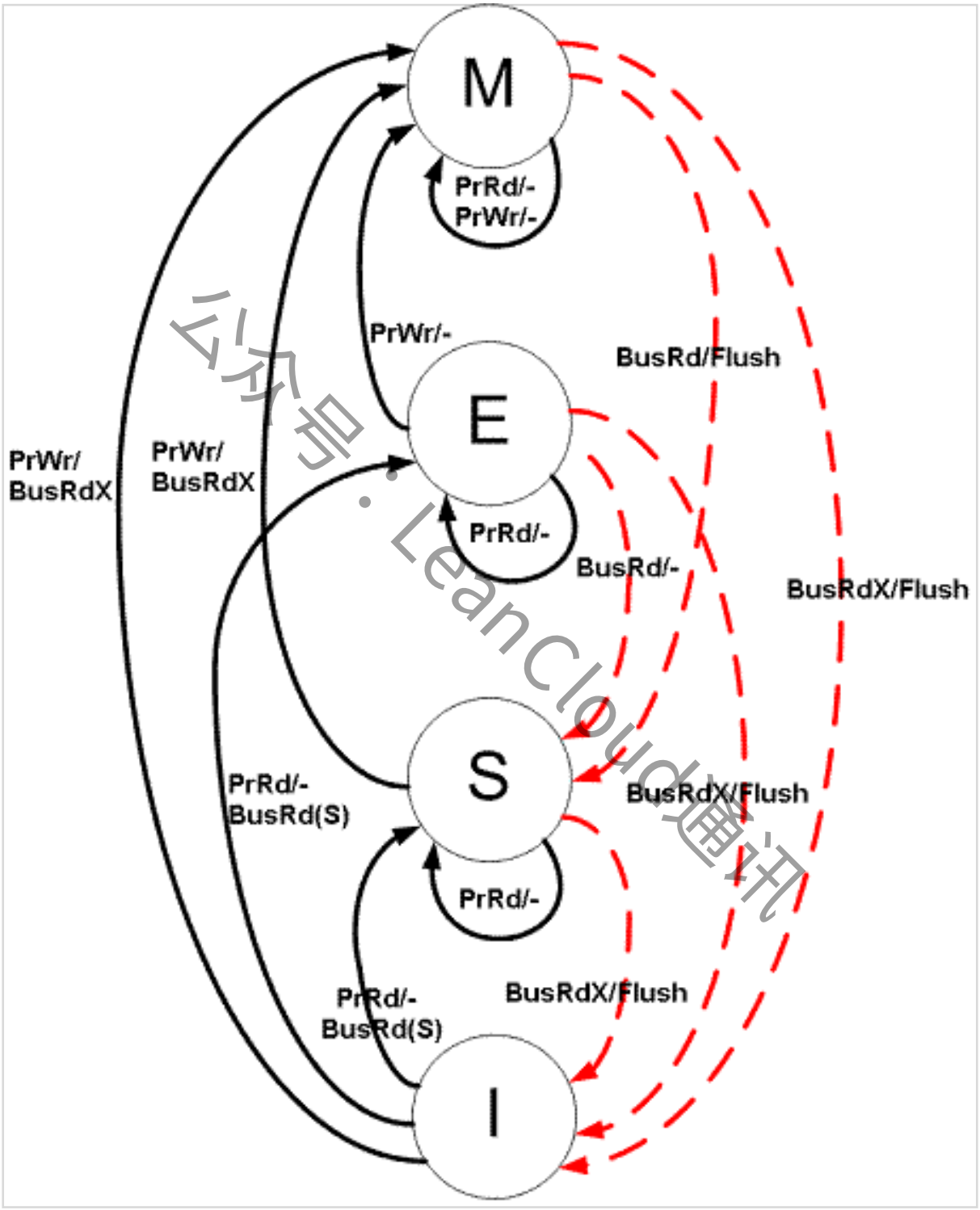
举例：

			CPU Cache				Memory	
Sequence #	CPU #	Operation	0	1	2	3	0	8
0		Initial State	–/I	–/I	–/I	–/I	V	V
1	0	Load	0/S	–/I	–/I	–/I	V	V
2	3	Load	0/S	–/I	–/I	0/S	V	V
3	0	Invalidation	8/S	–/I	–/I	0/S	V	V
4	2	RMW	8/S	–/I	0/E	–/I	V	V
5	2	Store	8/S	–/I	0/M	–/I	I	V
6	1	Atomic Inc	8/S	0/M	–/I	–/I	I	V
7	1	Writeback	8/S	8/S	–/I	–/I	V	V

左边是操作执行顺序，CPU 是执行操作的 CPU 编号。Operation 是执行的操作。RMW 表示读、修改、写。Memory 那里 V 表示内存数据是 Valid。

0. 一开始所有缓存都是 Invalid;
1. CPU 0 通过 `Read` 消息读 0 地址数据，0 地址所在 Cache Line 变成 `Shared` 状态;
2. CPU 3 再执行 `Read` 读 0 地址，它的 0 地址所在 Cache Line 也变成 `Shared`;
3. CPU 0 又从 Memory 读取 8 地址，替换了之前存 0 地址的 Cache Line。8 地址所在 Cache Line 也标记为 `Shared`。
4. CPU 2 因为要读取并修改 0 地址数据，所以发送 `Read Invalidate` 请求，首先 Load 0 地址数据到 Cache Line，再让当前 Cache 了 0 地址数据的 CPU 3 的 Cache 变成 `Invalidate`;
5. CPU 2 修改了 0 地址数据，0 地址数据在 Cache Line 上进入 `Modified` 状态，并且 Memory 上数据也变成 Invalid 的；
6. CPU 1 发送 `Read Invalidate` 请求，从 CPU 2 获取 0 地址的最新修改，并设置 CPU 2 上 Cache Line 为 `Invalidate`。CPU 1 在读取到 0 地址最新数据后对其进行修改，Cache Line 进入 `Modified` 状态。**注意**这里 CPU 2 没有 `Writeback` 0 地址数据到 Memory
7. CPU 1 读取 8 地址数据，因为自己的 Cache Line 满了，所以 `Writeback` 修改后的 0 地址数据到 Memory，读 8 地址数据到 Cache Line 设置为 `Shared` 状态。此时 Memory 上 0 地址数据进入 Valid 状态

真实的 MESI 协议非常复杂，MESI 因为是缓存之间维护数据一致性的协议，所以它所有请求都分为两端，请求来自 CPU 还是来自 Bus。请求来源不同在不同状态下也有不同结果。下面图片来自 wiki MESI protocol - Wikipedia，只是贴一下大概瞧瞧就好。



MOESI 和 MESIF

MOESI protocol - Wikipedia

MESIF protocol - Wikipedia

multithreading - What is the benefit of the MOESI cache coherency protocol over MESI? - Stack Overflow

Memory Barrier

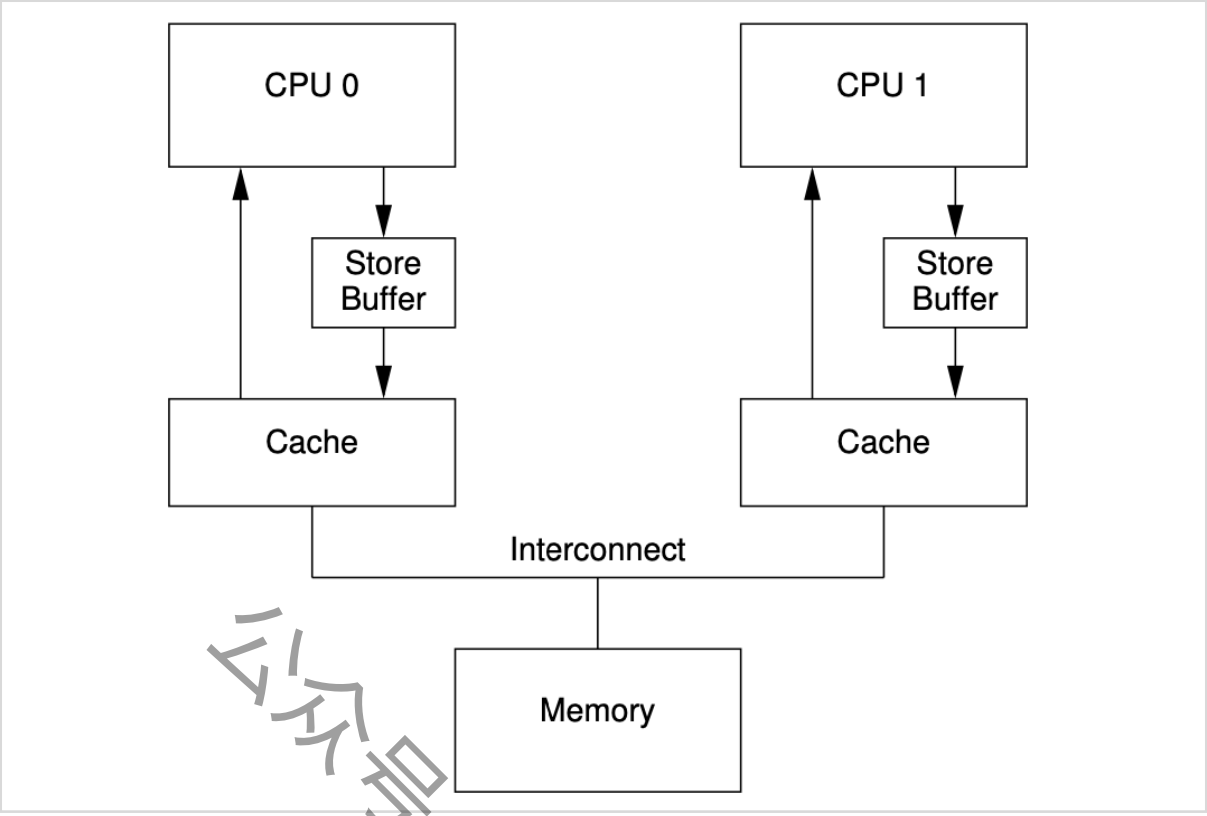
Store Buffer

假设 CPU 0 要写数据到某个地址，有两种情况：

1. CPU 0 已经读取了目标数据所在 Cache Line，处于 `Shared` 状态；
2. CPU 0 的 Cache 中还没有目标数据所在 Cache Line；

第一种情况下，CPU 0 只要发送 `Invalidate` 给其它 CPU 即可。收到所有 CPU 的 `Invalidate Ack` 后，这块 Cache Line 可以转换为 `Exclusive` 状态。第二种情况下，CPU 0 需要发送 `Read Invalidate` 到所有 CPU，拥有最新目标数据的 CPU 会把最新数据发给 CPU 0，并且会标记自己的这块 Cache Line 为无效。

无论是 `Invalidate` 还是 `Read Invalidate`，CPU 0 都得等其他所有 CPU 返回 `Invalidate Ack` 后才能安全操作数据，这个等待时间可能会很长。因为 CPU 0 这里只是想写数据到目标内存地址，它根本不关心目标数据在别的 CPU 上当前值是什么，所以这个等待是可以优化的，办法就是用 Store Buffer:



每次写数据时一方面发送 `Invalidate` 去其它 CPU，另一方面是将新写的数据内容放入 Store Buffer。等到所有 CPU 都回复 `Invalidate Ack` 后，再将对应 Cache Line 数据从 Store Buffer 移除，写入 CPU 实际 Cache Line。

除了避免等待 `Invalidate Ack` 外，Store Buffer 还能优化 `Write Miss` 的情况。比如即使只用一个 CPU，如果目标待写内存不在 Cache，正常来说需要等待数据从 Memory 加载到 Cache 后 CPU 才能开始写，那有了 Store Buffer 的存在，如果待写内存现在不在 Cache 里可以不用等待数据从 Memory 加载，而是把新写数据放入 Store Buffer，接着去执行别的操作，等数据加载到 Cache 后再把 Store Buffer 内的新写数据写入 Cache。

另外对于 `Invalidate` 操作，有没有可能两个 CPU 并发的去 `Invalidate` 某个相同的 Cache Line?

这种冲突主要靠 Bus 解决，可以从前面 MESI 的可视化工具看到，所有操作都得先访问 Address Bus，访问时会锁住 Address Bus，所以一段时间内只有一个 CPU 会操作 Bus，会操作某个 Cache Line。但是两个 CPU 可以不断相互修改同一个内存数据，导致同一个 Cache Line 在两个 CPU 上来回切换。

Store Forwarding

前面图上画的 Store Buffer 结构还有问题，主要是读数据的时候还需要读 Store Buffer 里的数

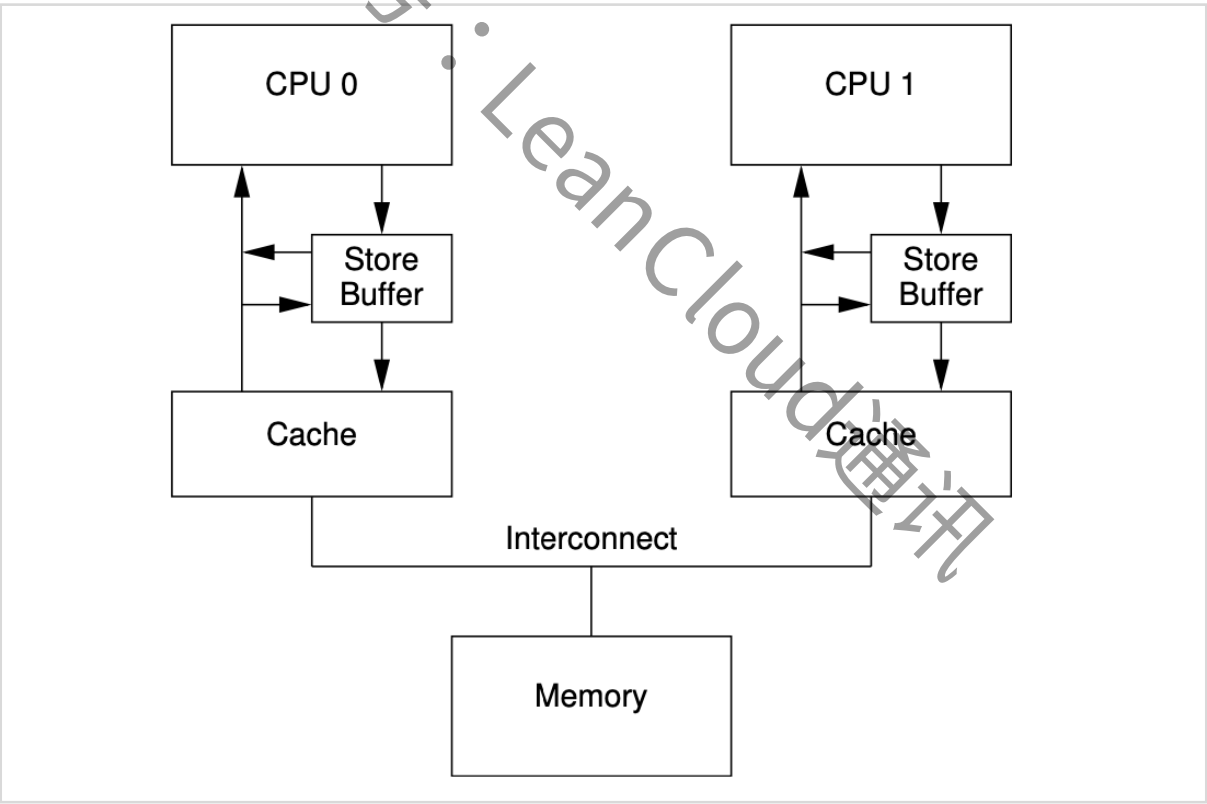
据，而不是写完 Store Buffer 就结束了。

比如现在有这个代码，a 一开始不在 CPU 0 内，在 CPU 1 内，值为 0。b 在 CPU 0 内：

```
a = 1;
b = a + 1;
assert(b == 2);
```

CPU 0 因为没缓存 a，写 a 为 1 的操作要放入 Store Buffer，之后需要发送 `Read Invalidate` 去 CPU 1。等 CPU 1 发来 a 的数据后 a 的值为 0，如果 CPU 0 在执行 `a + 1` 的时候不去读取 Store Buffer，则执行完 b 的值会是 1，而不是 2，导致 assert 出错。

所以更正常的结构是下图：



另外一开始 CPU 0 虽然就是想写 a 的值 为 1，根本不关心它现在值是什么，但也不能直接发送 `Invalidate` 给其它 CPU。因为 a 所在 Cache Line 上可能不只 a 在，可能还有别的数据在，如果直接发送 `Invalidate` 会导致 Cache Line 上不属于 a 的数据丢失。所以 `Invalidate` 只有在 Cache Line 处于 `Shared` 状态，准备向 `Exclusive` 转变时才会使用。

Write Barrier

```
// CPU 0 执行 foo(), 拥有 b 的 Cache Line
void foo(void)
{
    a = 1;
    b = 1;
}
// CPU 1 执行 bar(), 拥有 a 的 Cache Line
void bar(void)
{
    while (b == 0) continue;
    assert(a == 1);
}
```

对 CPU 0 来说，一开始 Cache 内没有 a，于是发送 `Read Invalidate` 去获取 a 所在 Cache Line 的修改权。a 写入的新值存在 Store Buffer。之后 CPU 0 就可以立即写 `b = 1` 因为 b 的 Cache Line 就在 CPU 0 上，处于 `Exclusive` 状态。

对 CPU 1 来说，它没有 b 的 Cache Line 所以需要先发送 `Read` 读 b 的值，如果此时 CPU 0 刚好写完了 `b = 1`，CPU 1 读到的 b 的值就是 1，就能跳出循环，此时如果还未收到 CPU 0 发来的 `Read Invalidate`，或者说收到了 CPU 0 的 `Read Invalidate` 但是只处理完 `Read` 部分给 CPU 0 发回去 a 的值即 `Read Response` 但还未处理完 `Invalidate`，也即 CPU 1 还拥有 a 的 Cache Line，CPU 0 还是不能将 a 的写入从 Store Buffer 写到 CPU 0 的 Cache Line 上。这样 CPU 1 上 a 读到的值就是 0，从而触发 assert 失败。

上述问题原因就是 Store Buffer 的存在，如果没有 Write Barrier，写入操作可能会乱序，导致后一个写入提前被其它 CPU 看到。

这里可能的一个疑问是，上述问题能出现意味着 CPU 1 在收到 `Read Invalidate` 后还未处理完就能发 `Read` 请求给 CPU 0 读 b 变量的 Cache Line，感觉上似乎不合理，因为似乎 Cache 应该是收到一个请求处理一个请求才对。这里可能有理解的盲区，我猜测是因为 `Read Invalidate` 实际分为两个操作，一个 `Read` 一个 `Invalidate`，`Read` 可以快速返回，但是 `Invalidate` 操作可能比较重，比如需要写回主存，那 Cache 可能有什么优化能允许等待执行完 `Invalidate` 返回 `Invalidate Ack` 前再收到 CPU 发来的轻量级的 `Read` 操作时可以把 `Read` 先丢出去，毕竟 CPU 读操作对 Cache 来说只需要转发，`Invalidate` 则是真的要 Cache 去操作自己的标志之类的，做的事情更多。

上面问题解决办法就是 Write Barrier，其作用是将 Write Barrier 之前所有操作的 Cache Line 都打上标记，Barrier 之后的写入操作不能直接操作 Cache Line 而也要先写 Store Buffer 去，只是这种拥有 Cache Line 但因为 Barrier 关系也写入 Store Buffer 的 Cache Line 不用打特殊标记。等 Store Buffer 内带着标记的写入因为收到 `Invalidate Ack` 而能写 Cache Line 后，这些没有打标记的写入操作才能写入 Cache Line。

相同代码带着 Write Barrier：

```
// CPU 0 执行 foo(), 拥有 b 的 Cache Line
void foo(void)
{
    a = 1;
    smp_wmb();
    b = 1;
}
// CPU 1 执行 bar(), 拥有 a 的 Cache Line
void bar(void)
{
    while (b == 0) continue;
    assert(a == 1);
}
```


此时对 CPU 0 来说，a 写入 Store Buffer 后带着特殊标记，b 的写入也得放入 Store Buffer。这样如果 CPU 1 还未返回 `Invalidate Ack`，CPU 0 对 b 的写入在 CPU 1 上就不可见。CPU 1 发来的 `Read` 读取 b 拿到的一直是 0。等 CPU 1 回复 `Invalidate Ack` 后，Ack 的是 a 所在 Cache Line，于是 CPU 0 将 Store Buffer 内 `a = 1` 的写入写到自己的 Cache Line，在从 Store Buffer 内找到所有排在 a 后面不带特殊标记的写入，即 `b = 1` 写入自己的 Cache Line。这样 CPU 1 再读 b 就会拿到新值 1，而此时 a 在 CPU 1 上因为回复过 `Invalidate Ack`，所以 a 会是 `Invalidate` 状态，重新读 a 后得到 a 值为 1。assert 成功。

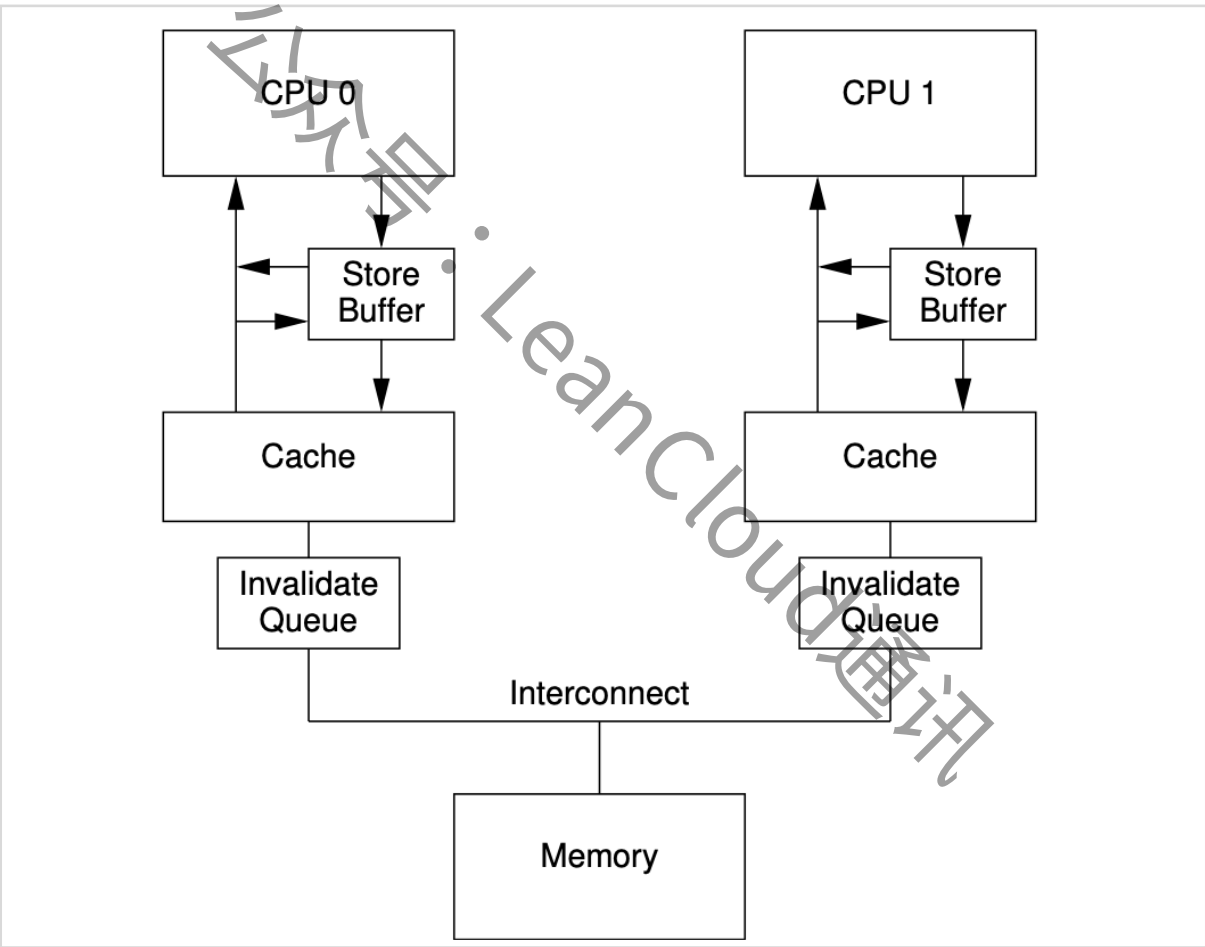
Invalidate Queue

每个 CPU 上 Store Buffer 都是有限的，当 Store Buffer 被写满之后，后续写入就必须等 Store Buffer 有位置后才能再写。就导致了性能问题。特别是 Write Barrier 的存在，一旦有 Write Barrier，后续所有写入都得放入 Store Buffer 会让 Store Buffer 排队写入数量大幅度增加。所以需要缩短写入请求在 Store Buffer 的排队时间。

之前提到 Store Buffer 存在原因就是等待 `Invalidate Ack` 可能较长，那缩短 Store Buffer 排队时间办法就是尽快回复 `Invalidate Ack`。`Invalidate` 操作时间长来自两方面：

1. 如果 Cache 特别繁忙，比如 CPU 有大量的在 Cache 上的读取、写入操作，可能导致 Cache 错过 `Invalidate` 消息，导致 `Invalidate` 延迟 (我认为是收到总线上信号后如果来不及处理可以丢掉，等信号发送方待会重试)
2. 可能短时间到来大量的 `Invalidate`，导致 Cache 来不及处理这么多 `Invalidate` 请求。每个还得回复 `Invalidate Ack`，也会占用总线通信时间

于是解决办法就是为每个 CPU 再增加一个 Invalidate Queue。收到 `Invalidate` 请求后将请求放入队列，并立即回复 `Ack`。



这么做导致的问题也是显而易见的。一个被 Invalidate 的 Cache Line 本来应该处于 `Invalidate` 状态，CPU 不该读、写里面数据的，但因为 `Invalidate` 请求被放入队列，CPU 还认为自己可以读写这个 Cache Line 而在操作老旧数据。从上图能看到 CPU 和 Invalidate Queue 在 Cache 两端，所以跟 Store Buffer 不同，CPU 不能去 Invalidate Queue 里查一个 Cache Line 是否被 Invalidate，这也是为什么 CPU 会读到无效数据的原因。

另一方面，Invalidate Queue 的存在导致如果要 Invalidate 一个 Cache Line，得先把 CPU 自

己的 Invalidate Queue 清理干净，或者至少有办法让 Cache 确认一个 Cache Line 在自己这里状态是非 Invalidate 的。

Read Barrier

因为 Invalidate Queue 的存在，CPU 可能读到旧值，场景如下：

```
// CPU 0 执行 foo(), a 处于 Shared, b 处于 Exclusive
void foo(void)
{
    a = 1;
    smp_wmb();
    b = 1;
}
// CPU 1 执行 bar(), a 处于 Shared 状态
void bar(void)
{
    while (b == 0) continue;
    assert(a == 1);
}
```

CPU 0 将 `a = 1` 写入 Store Buffer，发送 `Invalidate` (不是 `Read Invalidate`，因为 a 是 `Shared` 状态) 给 CPU 1。CPU 1 将 `Invalidate` 请求放入队列后立即返回了，所以 CPU 0 很快能将 1 写入 a、b 所在 Cache Line。CPU 1 再去读 b 的时候拿到 b 的新值 0，读 a 的时候认为 a 处于 `Shared` 状态于是直接读 a，拿到 a 的旧值比如 0，导致 assert 失败。最后，即使程序运行失败了，CPU 1 还需要继续处理 Invalidate Queue，把 a 的 Cache Line 设置为无效。

解决办法是加 Read Barrier。Read Barrier 起作用不是说 CPU 看到 Read Barrier 后就立即去处理 Invalidate Queue，把它处理完了再接着执行剩下东西，而只是标记 Invalidate Queue 上的 Cache Line，之后继续执行别的指令，直到看到下一个 Load 操作要从 Cache Line 里读数据了，CPU 才会等待 Invalidate Queue 内所有刚才被标记的 Cache Line 都处理完才继续执行下一个 Load。比如标记完 Cache Line 后，又有新的 `Invalidate` 请求进来，因为这些请求没有标记，所以下一次 Load 操作是不会等他们的。

```
// CPU 0 执行 foo(), a 处于 Shared, b 处于 Exclusive
```

```
void foo(void)
{
    a = 1;
    smp_wmb();
    b = 1;
}
// CPU 1 执行 bar(), a 处于 Shared 状态
void bar(void)
{
    while (b == 0) continue;
    smp_rmb();
    assert(a == 1);
}
```

有了 Read Barrier 后，CPU 1 读到 b 为 0后，标记所有 Invalidate Queue 上的 Cache Line 继续运行。下一个操作是读 a 当前值，于是开始等所有被标记的 Cache Line 真的被 Invalidate 掉，此时再读 a 发现 a 是 `Invalidate` 状态，于是发送 `Read` 到 CPU 0，拿到 a 所在 Cache Line 最新值，assert 成功。

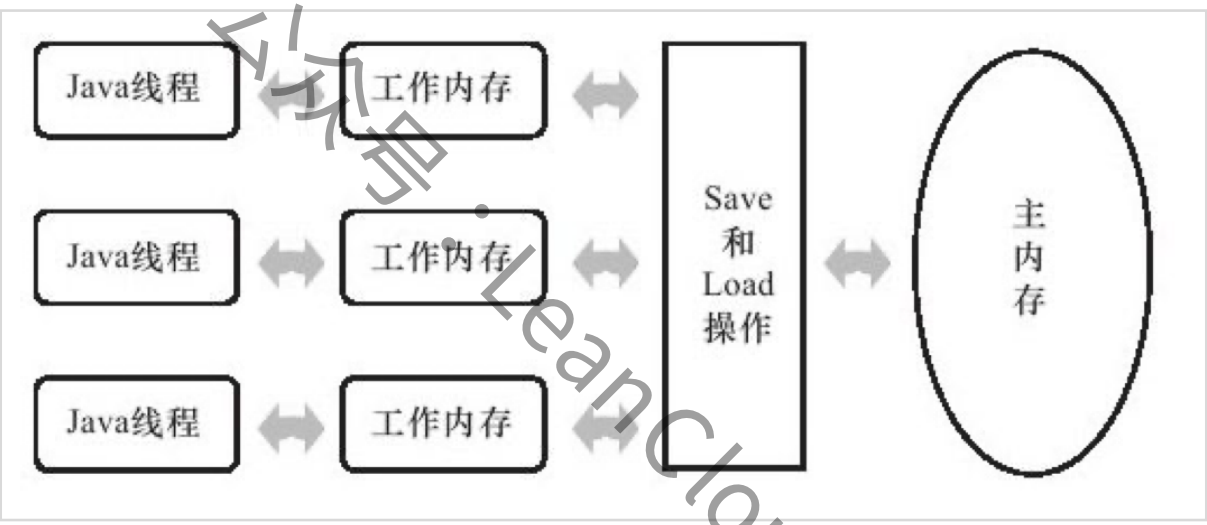
除了 Read Barrier 和 Write Barrier 外还有合二为一的 Barrier。作用是让后续写操作全部先去 Store Buffer 排队，让后续读操作都得先等 Invalidate Queue 处理完。

看到这里需要补充一下，本节对 Memory Barrier 的描述和前面 MESI 的描述是类似的，都是对真实世界做了一层简化，在这个超级简化的模型上管中窥豹的看看为什么要有 Memory Barrier。而实际 CPU Cache 的架构极为复杂，每种不同架构的 CPU 又有自己各自的取舍，也几乎没人直接用 MESI 协议，都使用的是它的变种。所以我觉得在我们通过这个极简模型对 Memory Barrier 出现原因有了大致了解后我们需要跳出这个模型。我们需要将 Memory Barrier 当做一个接口，当做是应用层程序员和 CPU、甚至编译器之间的一个交流工具，站在接口的角度上去看它。比如 Write Barrier 含义就是后续的写操作不能重排到 Write Barrier 之前，Read Barrier 是后续读操作不能被重排到 Read Barrier 之前。两者合二为一的 Barrier 是不准 Barrier 后面的任何读写操作被重排到 Barrier 之前。从分层的角度来说我们也不用去考虑 Write Barrier 是去刷了 Store Buffer 还是刷了 Invalidate Queue，只要知道它作为接口表达出来的含义即可。另外指令重排除了内存访问原因之外，编译器也会做静态的指令重排，即编译出来的代码就是重排过的，CPU 在执行指令的时候也会有乱序执行的情况。咱们这里只是从缓存访问作为切入点介绍内存屏障，介绍它存在的意义，而内存屏障实际对编译器和 CPU 执行顺序都有效。所以不能把用到 Barrier 的地方都去套上面的 CPU 模型。不然看到 Barrier 时候可能会有很多很难理解的地方，因为上面只是内存屏障起作用的一个方面，且给出的 CPU 模型很简化。

Java Memory Model (JMM)

Java 为了能在不同架构的 CPU 上运行，提炼出一套自己的内存模型，定义出来 Java 程序该怎么样和这个抽象的内存模型进行交互，定义出来程序的运行过程，什么样的指令可以重排，什么样的不行，指令之间可见性如何等。相当于是规范出来了 Java 程序运行的基本规范。这个模型定义会很不容易，它要有足够弹性，以适应各种不同的硬件架构，让这些硬件在支持 JVM 时候都能满足运行规范；它又要足够严谨，让应用层代码编写者能依靠这套规范，知道程序怎么写才能在各种系统上运行都不会有歧义，不会有并发问题。

在著名的 《深入理解 Java 虚拟机》一书的图 12-1 指出了在 JMM 内，线程、主内存、工作内存的关系。图片来自该书的 Kindle 版：



从内存模型一词就能看出来，这是对真实世界的模拟。图中 Java 线程对应的就是 CPU，工作内存对应的就是 CPU Cache，Java 提炼出来的一套 Save、Load 指令对应的就是缓存一致性协议，就是 MESI 等协议，最后主内存对应的就是 Memory。真实世界的硬件需要根据自身情况去向这套模型里套。

JMM 完善于 JSR-133，现在一般会把详细说明放在 Java Language 的 Spec 上，比如 Java11 的话在：Chapter_17._Threads and Locks。在这些说明之外，还有个特别出名的 Cookbook，叫 The JSR-133 Cookbook for Compiler Writers。看上去是说写给 Java 的 Compiler 看，让 Compiler 知道如何能遵循 JSR-133 描绘的 JMM 要求。对我们上层 Java 使用者来说，这个 Cookbook 就是探索 JVM 与内存、Cache 交互的途径。这里面水很深，介绍了很多东西，我这里只记录跟本文相关的内容。

JVM 上的 Memory Barrier

JVM 按前后分别有读、写两种操作以全排列方式一共提供了四种 Barrier，名称就是左右两边

操作的名字拼接。比如 `LoadLoad` Barrier 就是放在两次 Load 操作中间的 Barrier，`LoadStore` 就是放在 Load 和 Store 中间的 Barrier。Barrier 类型及其含义如下：

- `LoadLoad`，操作序列 Load1, LoadLoad, Load2，用于保证访问 Load2 的读取操作一定不能重排到 Load1 之前。类似于前面说的 `Read Barrier`，需要先处理 Invalidate Queue 后再读 Load2；
- `StoreStore`，操作序列 Store1, StoreStore, Store2，用于保证 Store1 及其之后写出的数据一定先于 Store2 写出，即别的 CPU 一定先看到 Store1 的数据，再看到 Store2 的数据。可能会有一次 Store Buffer 的刷写，也可能通过所有写操作都放入 Store Buffer 排序来保证；
- `LoadStore`，操作序列 Load1, LoadStore, Store2，用于保证 Store2 及其之后写出的数据被其它 CPU 看到之前，Load1 读取的数据一定先读入缓存。甚至可能 Store2 的操作依赖于 Load1 的当前值。这个 Barrier 的使用场景可能和上一节讲的 Cache 架构模型很难对应，毕竟那是一个极简结构，并且只是一种具体的 Cache 架构，而 JVM 的 Barrier 要足够抽象去应付各种不同的 Cache 架构。如果跳出上一节的 Cache 架构来说，我理解用到这个 Barrier 的场景可能是说某种 CPU 在写 Store2 的时候，认为刷写 Store2 到内存，将其它 CPU 上 Store2 所在 Cache Line 设置为无效的速度要快于从内存读取 Load1，所以做了这种重排。
- `StoreLoad`，操作序列 Store1, StoreLoad, Load2，用于保证 Store1 写出的数据被其它 CPU 看到后才能读取 Load2 的数据到缓存。如果 Store1 和 Load2 操作的是同一个地址，StoreLoad Barrier 需要保证 Load2 不能读 Store Buffer 内的数据，得是从内存上拉取到的某个别的 CPU 修改过的值。`StoreLoad` 一般会认为是最重的 Barrier 也是能实现其它所有 Barrier 功能的 Barrier。

  对上面四种 Barrier 解释最好的是来自这里：jdk/MemoryBarriers.java at 6bab0f539fba8fb441697846347597b4a0ade428 · openjdk/jdk · GitHub，感觉比 JSR-133 Cookbook 里的还要细一点。我在这里也抄一下吧：

  LoadLoad

  > The sequence Load1; LoadLoad; Load2 ensures that Load1's data are loaded before data accessed by Load2 and all subsequent load instructions are loaded. In general, explicit LoadLoad barriers are needed on processors that perform speculative loads and/or out-of-order processing in which waiting load instructions can bypass waiting stores. On processors that guarantee to always preserve load ordering, these barriers amount to no-ops.

  LoadStore

  > The sequence Load1; LoadStore; Store2 ensures that Load1's data are loaded before all data associated with Store2 and subsequent store instructions are flushed. LoadStore

> barriers are needed only on those out-of-order processors in which waiting store instructions can bypass loads.

StoreLoad

> The sequence Store1; StoreLoad; Load2 ensures that Store1's data are made visible to other processors (i.e., flushed to main memory) before data accessed by Load2 and all subsequent load instructions are loaded. StoreLoad barriers protect against a subsequent load incorrectly using Store1's data value rather than that from a more recent store to the same location performed by a different processor.
> Because of this, on the processors discussed below, a StoreLoad is strictly necessary only for separating stores from subsequent loads of the same location(s) as were stored before the barrier. StoreLoad barriers are needed on nearly all recent multiprocessors, and are usually the most expensive kind. Part of the reason they are expensive is that they must disable mechanisms that ordinarily bypass cache to satisfy loads from write-buffers. This might be implemented by letting the buffer fully flush, among other possible stalls.

StoreStore

> The sequence Store1; StoreStore; Store2 ensures that Store1's data are visible to other processors (i.e., flushed to memory) before the data associated with Store2 and all subsequent store instructions. In general, StoreStore barriers are needed on processors that do not otherwise guarantee strict ordering of flushes from write buffers and/or caches to other processors or main memory.

为什么这一堆 Barrier 里 `StoreLoad` 最重？

所谓的重实际就是跟内存交互次数，交互越多延迟越大，也就是越重。`StoreStore`，`LoadLoad` 两个都不提了，因为它俩要么只限制读，要么只限制写，也即只有一次内存交互。只有 `LoadStore` 和 `StoreLoad` 看上去有可能对读写都有限制。但 `LoadStore` 里实际限制的更多的是读，即 Load 数据进来，它并不对最后的 Store 存出去数据的可见性有要求，只是说 Store 不能重排到 Load 之前。而反观 `StoreLoad`，它是说不能让 Load 重排到 Store 之前，这么一来得要求在 Load 操作前刷写 Store Buffer 到内存。不去刷 Store Buffer 的话，就可能导致先执行了读取操作，之后再刷 Store Buffer 导致写操作实际被重排到了读之后。而数据一旦刷写出去，别的 CPU 就能看到，看到之后可能就会修改下一步 Load 操作的内存导致 Load 操作的内存所在 Cache Line 无效。如果允许 Load 操作从一个可能被 Invalidate 的 Cache Line 里读数据，则表示 Load 从实际意义上来说被重排到了 Store 之前，因为这个数据可能是 Store 前就在 Cache 中的，相当于读操作提前了。为了避免这种事发生，Store 完成后一定要去处理 Invalidate Queue，去判断自己 Load 操作的内存所在 Cache Line 是否被设置为无

效。这么一来为了满足 `StoreLoad` 的要求，一方面要刷 Store Buffer，一方面要处理 Invalidate Queue，则最差情况下会有两次内存操作，读写分别一次，所以它最重。

`StoreLoad` 为什么能实现其它 Barrier 的功能？

这个也是从前一个问题结果能看出来的。`StoreLoad` 因为对读写操作均有要求，所以它能实现其它 Barrier 的功能。其它 Barrier 都是只对读写之中的一个方面有要求。

不过这四个 Barrier 只是 Java 为了跨平台而设计出来的，实际上根据 CPU 的不同，对应 CPU 平台上的 JVM 可能可以优化掉一些 Barrier。比如很多 CPU 在读写同一个变量的时候能保证它连续操作的顺序性，那就不用加 Barrier 了。比如 `Load x; Load x.field` 读 x 再读 x 下面某个 field，如果访问同一个内存 CPU 能保证顺序性，两次读取之间的 Barrier 就不再需要了，根据字节码编译得到的汇编指令中，本来应该插入 Barrier 的地方会被替换为 `nop`，即空操作。在 x86 上，实际只有 `StoreLoad` 这一个 Barrier 是有效的，x86 上没有 Invalidate Queue，每次 Store 数据又都会去 Store Buffer 排队，所以 `StoreStore`，`LoadLoad` 都不需要。x86 又能保证 Store 操作都会走 Store Buffer 异步刷写，Store 不会被重排到 Load 之前，`LoadStore` 也是不需要的。只剩下一个 `StoreLoad` Barrier 在 x86 平台的 JVM 上被使用。

x86 上怎么使用 Barrier 的说明可以在 openjdk 的代码中看到，大致上是：

```
enum Membar_mask_bits {
    StoreStore = 1 << 3,
    LoadStore  = 1 << 2,
    StoreLoad  = 1 << 1,
    LoadLoad   = 1 << 0
  };

  // Serializes memory and blows flags
  void membar(Membar_mask_bits order_constraint) {
    // We only have to handle StoreLoad
    if (order_constraint & StoreLoad) {
      // All usable chips support "locked" instructions which suffice
      // as barriers, and are much faster than the alternative of
      // using cpuid instruction. We use here a locked add [esp-C],0.
      // This is conveniently otherwise a no-op except for blowing
      // flags, and introducing a false dependency on target memory
      // location. We can't do anything with flags, but we can avoid
      // memory dependencies in the current method by locked-adding
```

```
      // somewhere else on the stack. Doing [esp+C] will collide with
      // something on stack in current method, hence we go for [esp-C].
      // It is convenient since it is almost always in data cache, for
      // any small C.  We need to step back from SP to avoid data
      // dependencies with other things on below SP (callee-saves, for
      // example). Without a clear way to figure out the minimal safe
      // distance from SP, it makes sense to step back the complete
      // cache line, as this will also avoid possible second-order effects
      // with locked ops against the cache line. Our choice of offset
      // is bounded by x86 operand encoding, which should stay within
      // [-128; +127] to have the 8-byte displacement encoding.
      //
      // Any change to this code may need to revisit other places in
      // the code where this idiom is used, in particular the
      // orderAccess code.

      int offset = -VM_Version::L1_line_size();
      if (offset < -128) {
        offset = -128;
      }

      lock();
      addl(Address(rsp, offset), 0);// Assert the lock# signal here
    }
  }
```


来自这里：jdk/assembler_x86.hpp at 9a69bb807beb6693c68a7b11bee435c0bab7ceac · openjdk/jdk · GitHub
看到 x86 下使用的是 lock 来实现 `StoreLoad`，并且只处理了 `StoreLoad`。

volatile

JVM 上对 Barrier 的一个主要应用是在 `volatile` 关键字的实现上。对这个关键字的实现 Oracle 有这么一段描述：

> Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to

> other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

来自 Oracle 对 Atomic Access 的说明：Atomic Access。大致上就是说被 `volatile` 标记的变量需要维护两个特性：

- 可见性，每次读 `volatile` 变量总能读到它最新值，即最后一个对它的写入操作，不管这个写入是不是当前线程完成的。
- 禁止指令重排，也即维护 `happens-before` 关系，对 `volatile` 变量的写入不能重排到写入之前的操作之前，从而保证别的线程看到写入值后就能知道写入之前的操作都已经发生过；对 `volatile` 的读取操作一定不能被重排到后续操作之后，比如我需要读 `volatile` 后根据读到的值做一些事情，做这些事情如果重排到了读 `volatile` 之前，则相当于没有满足读 `volatile` 需要读到最新值的要求，因为后续这些事情是根据一个旧 `volatile` 值做的。

  需要看到两个事情，一个是禁止指令重排不是禁止所有的重排，只是 `volatile` 写入不能向前排，读取不能向后排。别的重排还是会允许。另一个是禁止指令重排实际也是为了去满足可见性而附带产生的。所以 `volatile` 对上述两个特性的维护就能靠 Barrier 来实现。

  假设约定 Normal Load, Normal Store 对应的是对普通引用的修改。好比有 `int a = 1;` 那 `a = 2;` 就是 Normal Store，`int b = a;` 就有一次对 a 的 Normal Load。如果变量带着 `volatile` 修饰，那对应的读取和写入操作就是 Volatile Load 或者 Volatile Store。`volatile` 对代码生成的字节码本身没有影响，即 Java Method 生成的字节码无论里面操作的变量是不是 `volatile` 声明的，生成的字节码都是一样的。`volatile` 在字节码层面影响的是 Class 内 Field 的 `access_flags`(参看 Java 11 The Java Virtual Machine Specification 的 4.5 节)，可以理解为当看到一个成员变量被声明为 `volatile`，Java 编译器就在这个成员变量上打个标记记录它是 `volatile` 的。JVM 在将字节码编译为汇编时，如果碰见比如 `getfield`, `putfield` 这些字节码，并且发现操作的是带着 `volatile` 标记的成员变量，就会在汇编指令中根据 JMM 要求插入对应的 Barrier。

  根据 `volatile` 语义，我们依次看下面操作次序该用什么 Barrier，需要说明的是这里前后两个操作需要操作不同的变量：

- Normal Store, Volatile Store。即先写一个普通变量，再写一个带 `volatile` 的变量。这种很明显是得用 StoreStore Barrier。
- Volatile Store, Volatile Store。也明显是 StoreStore，因为第二次修改被别的 CPU 看到时需要保证这次写入之前的写入都能被看到。
- Nolmal Load, Volatile Store。得用 LoadStore，避免 Store 操作重排到 Load 之前。
- Volatile Load, Volatile Store。得用 LoadStore，原因同上。

上面四种情况要用 Barrier 的原因统一来说就是前面 Oracle 对 Atomic Access 的说明，写一个 `volatile` 的变量被别的 CPU 看到时，需要保证写这个变量操作之前的操作都能完成，不管前一个操作是读还是写，操作的是 `volatile` 变量还是不是。如果 Store 操作做了重排，排到了前一个操作之前，就会违反这个约定。所以 `volatile` 变量操作是在 Store 操作前面加 Barrier，而 Store 后如果是 Normal 变量就不用 Barrier 了，重不重排都无所谓：

- Volatile Store, Normal Load
- Volatile Store, Normal Store

对于 `volatile` 变量的读操作，为了满足前面提到 `volatile` 的两个特性，为了避免后一个操作重排到读 `volatile` 操作之前，所以对 `volatile` 的读操作都是在读后面加 Barrier：

- Volatile Load, Volatile Load。得用 LoadLoad。
- Volatile Load, Normal Load。得用 LoadLoad。
- Volatile Load, Normal Store。得用 LoadStore。

而如果后一个操作是 Load，则不需要再用 Barrier，能随意重排：

- Normal Store, Volatile Load。
- Normal Load, Volatile Load。

最后还有个特别的，前一个操作是 Volatile Store，后一个操作是 Volatile Load：

- Volatile Store, Volatile Load。得用 StoreLoad。因为前一个 Store 之前的操作可能导致后一个 Load 的变量发生变化，后一个 Load 操作需要能看到这个变化。

还剩下四个 Normal 的操作，都是随意重排，没影响：

- Normal Store, Normal Load
- Normal Load, Normal Load
- Normal Store, Normal Store
- Normal Load, Normal Store

这些使用方式和 Java 下具体操作的对应表如下：

Required barriers	*2nd operation*			
1st operation	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

图中 Monitor Enter 和 Monitor Exit 分别对应着进出 `synchronized` 块。Monitor Ender 和 Volatile Load 对应，使用 Barrier 的方式相同。Monitor Exit 和 Volatile Store 对应，使用 Barrier 的方式相同。

总结一下这个图，记忆使用 Barrier 的方法非常简单，只要是写了 `volatile` 变量，为了保证对这个变量的写操作被其它 CPU 看到时，这个写操作之前发生的事情也都能被别的 CPU 看到，那就需要在写 `volatile` 之前加入 Barrier。避免写操作被向前重排导致 `volatile` 变量已经写入了被别的 CPU 看到了但它前面写入过、读过的变量却没有被别的 CPU 感知到。写入变量被别的 CPU 感知到好说，这里读变量怎么可能被别的 CPU 感知到呢？主要是

在读方面，只要是读了 `volatile` 变量，为了保证后续基于这次读操作而执行的操作能真的根据读到的最新值做接下来的事情，需要在读操作之后加 Barrier。

在此之外加一个特殊的 Volatile Store, Volatile Load，为了保证后一个读取能看到因为前一次写入导致的变化，所以需要加入 `StoreLoad` Barrier。

JMM 说明中，除了上面表中讲的 `volatile` 变量相关的使用 Barrier 地方之外，还有个特殊地方也会用到 Barrier，是 `final` 修饰符。在修改 `final` 变量和修改别的共享变量之间，要有一个 `StoreStore` Barrier。例如 `x.finalField = v; StoreStore; sharedRef = x;` 下面是一组操作举例，看具体什么样的变量被读取、写入的时候需要使用 Barrier。

最后可以看一下 JSR-133 Cookbook 里给的例子，大概感受一下操作各种类型变量时候 Barrier 是怎么加的：

Java	Instructions

class X {
  int a, b;
  volatile int v, u;
  void f() {
  int i, j;
 	
  i = a;
  j = b;
  i = v;
 	
  j = u;
 	
  a = i;
  b = j;
 	
  v = i;
 	
  u = j;
 	
  i = u;
 	
  j = b;
  a = i;
  }
}
	
	load a
	load b
	load v
	  LoadLoad
	load u
	  LoadStore
	store a
	store b
	  StoreStore
	store v
	  StoreStore
	store u
	  StoreLoad
	load u
	  LoadLoad
	  LoadStore
	load b
	store a

volatile 的可见性维护

总结来说，`volatile` 可见性包括两个方面：

1. 写入的 `volatile` 变量在写完之后能被别的 CPU 在下一次读取中读取到；
2. 写入 `volatile` 变量之前的操作在别的 CPU 看到 `volatile` 的最新值后一定也能被看到；

对于第一个方面，主要通过：

1. 读取 `volatile` 变量不能使用寄存器，每次读取都要去内存拿
2. 禁止读 `volatile` 变量后续操作被重排到读 `volatile` 之前

对于第二个方面，主要是通过写 `volatile` 变量时的 Barrier 保证写 `volatile` 之前的操作先于写 `volatile` 变量之前发生。

最后还一个特殊的，如果能用到 `StoreLoad` Barrier，写 `volatile` 后一般会触发 Store Buffer 的刷写，所以写操作能「立即」被别的 CPU 看到。

一般提到 `volatile` 可见性怎么实现，最常听到的解释是「写入数据之后加一个写 Barrier 去刷缓存到主存，读数据之前加入 Barrier 去强制从主存读」。

从前面对 JMM 的介绍能看到，至少从 JMM 的角度来说，这个说法是不够准确的。一方面 Barrier 按说加在写 `volatile` 变量之前，不该之后加 Barrier。而读 `volatile` 是在之后会加 Barrier，而不在之前。另一方面关于 「刷缓存」的表述也不够准确，即使是 `StoreLoad` Barrier 刷的也是 Store Buffer 到缓存里，而不是缓存向主存去刷。如果待写入的目标内存在当前 CPU Cache，即使触发 Store Buffer 刷写也是写数据到 Cache，并不会触发 Cache 的 Writeback 即向内存做同步的事情，同步主存也没有意义因为别的 CPU 并不一定关心这个值；同理，即使读 `volatile` 变量后有 Barrier 的存在，如果目标内存在当前 CPU Cache 且处于 Valid 状态，那读取操作就立即从 Cache 读，并不会真的再去内存拉一遍数据。

需要补充的是无论是 `volatile` 还是普通变量在读写操作本身方面完全是一样的，即读写操作都交给 Cache，Cache 通过 MESI 及其变种协议去做缓存一致性维护。这两种变量的区别就只在于 Barrier 的使用上。

volatile 读取操作是 Free 的吗

在 x86 下因为除了 `StoreLoad` 之外其它 Barrier 都是空操作，但是读 `volatile` 变量并不是完全无开销，一方面 Java 的编译器还是会遵照 JMM 要求在本该加入 Barrier 的汇编指令处填入 `nop`，这会阻碍 Java 编译器的一些优化措施。比如本来能进行指令重排的不敢进行指令重排等。另外因为访问的变量被声明为 `volatile`，每次读取它都得从内存(或 Cache) 要，而不能把 `volatile` 变量放入寄存器反复使用。这也降低了访问变量的性能。可以看看这里：

- Psychosomatic, Lobotomy, Saw: JMM Cookbook Footnote: NOOP Memory Barriers on x86 are NOT FREE
- Are volatile reads really free? - Marc's Blog
- java - Are volatile variable 'reads' as fast as normal reads? - Stack Overflow

理想情况下对 `volatile` 字段的使用应当多读少写，并且尽量只有一个线程进行写操作。不过读 `volatile` 相对读普通变量来说也有开销存在，只是一般不是特别重。

回顾 False Sharing 里的例子

CPU Cache 基础 这篇文章内介绍了 False Sharing 的概念以及如何观察到 False Sharing 现象。其中有个关键点是为了能更好的观察到 False Sharing，得将被线程操作的变量声明为 `volatile`，这样 False Sharing 出现时性能下降会非常多，但如果去掉 `volatile` 性能下降比率就会减少，这是为什么呢？

简单来说如果没有 `volatile` 声明，也即没有 Barrier 存在，每次对变量进行修改如果当前变量所在内存的 Cache Line 不在当前 CPU，那就将修改的操作放在 Store Buffer 内等待目标 Cache Line 加载后再实际执行写入操作，这相当于写入操作在 Store Buffer 内做了积累，比如 `a++` 操作不是每次执行都会向 Cache 里执行加一，而是在 Cache 加载后直接执行比如加 10，加 100，从而将一批加一操作合并成一次 Cache Line 写入操作。而有了 `volatile` 声明，有了 Barrier，为了保证写入数据的可见性，就会引入等待 Store Buffer 刷写 Cache Line 的开销。当目标 Cache Line 还未加载入当前 CPU 的 Cache，写数据先写 Store Buffer，但看到例如 `StoreLoad` Barrier 后需要等待 Store Buffer 的刷写才能继续执行下一条指令。还是拿 `a++` 来说，每次加一操作不再能积累，而是必须等着 Cache Line 加载，执行完 Store Buffer 刷写后才能继续下一个写入，这就放大了 Cache Miss 时的影响，所以出现 False Sharing 时 Cache Line 在多个 CPU 之间来回跳转，在被修改的变量有了 `volatile` 声明后会执行的更慢。

再进一步说，我是在我本机做测试，我的机器是 x86 架构的，在我的机器上实际只有 `StoreLoad` Barrier 会真的起作用。我们去 open jdk 的代码里看看 `StoreLoad` Barrier 是怎么加上的。

先看这里，JSR-133 Cookbook 里定义了一堆 Barrier，但 JVM 虚拟机上实际还会定义更多一些 Barrier 在 src/hotspot/share/runtime/orderAccess.hpp。

每个不同的系统或 CPU 架构会使用不同的 `orderAccess` 的实现，比如 linux x86 的在 src/hotspot/os_cpu/linux_x86/orderAccess_linux_x86.hpp，BSD x86 和 Linux x86 的类似在 src/hotspot/os_cpu/bsd_x86/orderAccess_bsd_x86.hpp，都是这样定义的：

```
inline void OrderAccess::loadload()   { compiler_barrier(); }
inline void OrderAccess::storestore() { compiler_barrier(); }
inline void OrderAccess::loadstore()  { compiler_barrier(); }
inline void OrderAccess::storeload()  { fence();            }

inline void OrderAccess::acquire()    { compiler_barrier(); }
inline void OrderAccess::release()    { compiler_barrier(); }

inline void OrderAccess::fence() {
   // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
  __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
#else
  __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  compiler_barrier();
}
```


`compiler_barrier()` 只是为了不做指令重排，但是对应的是空操作。看到上面只有 `StoreLoad` 是实际有效的，对应的是 `fence()`，看到 `fence()` 的实现是用 `lock`。为啥用 `lock` 在前面贴的 `assembler_x86` 的注释里有说明。

之后 `volatile` 变量在每次修改后，都需要使用 `StoreLoad` Barrier，在解释执行字节码的代码里能看到。src/hotspot/share/interpreter/bytecodeInterpreter.cpp，看到是执行 `putfield` 的时候如果操作的是 `volatile` 变量，就在写完之后加一个 `StoreLoad` Barrier。我们还能找到 `MonitorExit` 相当于对 `volatile` 的写入，在 JSR-133 Cookbook 里有说过，在 openjdk 的代码里也能找到证据在 src/hotspot/share/runtime/objectMonitor.cpp。

JSR-133 Cookbook 还提到 `final` 字段在初始化后需要有 `StoreStore` Barrier，在 src/hotspot/share/interpreter/bytecodeInterpreter.cpp 也能找到。

这里问题又来了，按 JSR-133 Cookbook 上给的图，连续两次 `volatile` 变量写入中间不该用的是 `StoreStore` 吗，从上面代码看用的怎么是 `StoreLoad`。从 JSR-133 Cookbook 上给的 `StoreLoad` 是说 `Store1; StoreLoad; Load2` 含义是 Barrier 后面的所有读取操作都不能重排在 Store1 前面，并不是仅指紧跟着 Store1 后面的那次读，而是不管隔多远只要有读取都不能做重排。所以我理解拿 `volatile` 修饰的变量来说，写完 `volatile` 之后，程序总有某个位置会

去读这个 `volatile` 变量，所以写完 `volatile` 变量后一定总对应着 `StoreLoad` Barrier，只是理论上存在比如只写 `volatile` 变量但从来不读它，这时候才可能产生 `StoreStore` Barrier。当然这个只是我从 JDK 代码上和实际测试中得到的结论。

怎么观察到上面说的内容是不是真的呢？我们需要把 JDK 编码结果打印出来。可以参考这篇文章。简单来说有两个关键点：

- 启动 Java 程序时候带着这些参数：`-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly`
- 需要想办法下载或编译出来 hsdis，放在 `JAVA_HOME` 的 `jre/lib` 下面

如果缺少 `hsdis` 则会在启动程序时候看到：

```
Could not load hsdis-amd64.dylib; library not loadable; PrintAssembly is
disabled
```

之后我们去打印之前测试 False Sharing 例子中代码编译出来的结果，可以看到汇编指令中，每次执行完写 `volatile` 的 `valueA` 或者 `valueB` 后面都跟着 `lock` 指令，即使 JIT 介入后依然如此，汇编指令大致上类似于：

```
0x000000010f9b180: lock addl $0x0,(%rsp)     ;*putfield valueA
                                               ; -
cn.leancloud.filter.service.SomeClassBench::testA@2 (line 22)
```


内存屏障在 JVM 的其它应用

Atomic 的 LazySet

跟 Barrier 相关的还一个有意思的，是 Atomic 下的 LazySet 操作。拿最常见的 `AtomicInteger` 为例，里面的状态 `value` 是个 `volatile` 的 `int`，普通的 `set` 就是将这个状态修改为目标值，修改后因为有 Barrier 的关系会让其它 CPU 可见。而 `lazySet` 与 `set` 对比是这样：

```
public final void set(int newValue) {
    value = newValue;
```

```
}
public final void lazySet(int newValue) {
    unsafe.putOrderedInt(this, valueOffset, newValue);
}
```

对于 `unsafe.putOrderedInt()` 的内容 Java 完全没给出解释，但从添加 `lazySet()` 这个功能的地方： Bug ID: JDK-6275329 Add lazySet methods to atomic classes，能看出来其作用是在写入 `volatile` 状态前增加 `StoreStore` Barrier。它只保证本次写入不会重排到前面写入之前，但本次写入什么时候能刷写到内存是不做要求的，从而是一次轻量级的写入操作，在特定场景能优化性能。

ConcurrentLinkedQueue 下的黑科技

简单介绍一下这个黑科技。比如现在有 a b c d 四个 `volatile` 变量，如果无脑执行:

```
a = 1;
b = 2;
c = 3;
d = 4;
```

会在每个语句中间加上 Barrier。直接上面这样写可能还好，都是 `StoreStore` 的 Barrier，但如果写 `volatile` 之后又有一些读 `volatile` 操作，可能 Barrier 就会提升至最重的 `StoreLoad` Barrier，开销就会很大。而如果对开始的 a b c 写入都是用写普通变量的方式写入，只对最后的 d 用 `volatile` 方式更新，即只在 `d = 4` 前带上写 Barrier，保证 `d = 4` 被其它 CPU 看见时，a、b、c 的值也能被别的 CPU 看见。这么一来就能减少 Barrier 的数量，提高性能。

JVM 里上一节介绍的 `unsafe` 下还有个叫 `putObject` 的方法，用来将一个 `volatile` 变量以普通变量方式更新，即不使用 Barrier。用这个 `putObject` 就能做到上面提到的优化。

`ConcurrentLinkedQueue` 是 Java 标准库提供的无锁队列，它里面就用到了这个黑科技。因为是链表，所以里面有个叫 `Node` 的类用来存放数据，`Node` 连起来就构成链表。`Node` 内有个被 `volatile` 修饰的变量指向 `Node` 存放的数据。`Node` 的部分代码如下：

```
private static class Node<E> {
```

```
    volatile E item;
    volatile Node<E> next;
    Node(E item) {
        UNSAFE.putObject(this, itemOffset, item);
    }
    ....
}
```

因为 Node 被构造出来后它得通过 `cas` 操作队尾 `Node` 的 `next` 引用接入链表，接入成功之后才需要被其它 CPU 看到，在 `Node` 刚构造出来的时候，`Node` 内的 `item` 实际不会被任何别的线程访问，所以看到 `Node` 的构造函数可以直接用 `putObject` 更新 `item`，等后续 `cas` 操作队列队尾 `Node` 的 `next` 时候再以 `volatile` 方式更新 `next`，从而带上 Barrier，更新完成后 `next` 的更新包括 `Node` 内 `item` 的更新就都被别的 CPU 看到了。从而减少操作 `volatile` 变量的开销。

这个无锁的链表实现的黑科技横飞，很多实现细节很考验对 `volatile` 的理解。

除了 `ConcurrentLinkedQueue` 我另一个看过的黑科技横飞的是 Disruptor，False Sharing，`putOrderedXXX`，`putXXXVolatile` 四处可见，可以作为参考看看。

其它参考

- http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
- Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software
- Memory Barriers Are Like Source Control Operations
- multithreading - Why is a store-load barrier considered expensive? - Stack Overflow
- Memory Reordering Caught in the Act
- https://stackoverflow.com/questions/15360598/what-does-a-loadload-barrier-really-do
- https://www.infoq.com/articles/memory_barriers_jvm_concurrency
- Mechanical Sympathy: Memory Barriers/Fences