伪共享的非标准定义为：缓存系统中是以缓存行（cache line）为单位存储的，当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享。

下面我们就来详细剖析伪共享产生的前因后果。首先，我们要了解什么是缓存系统。

**一、CPU 缓存**

CPU 缓存的百度百科定义为：

复制代码

CPU 缓存（Cache Memory）是位于 CPU 与内存之间的临时存储器，它的容量比内存小的多但是交换速度却比内存要快得多。】

高速缓存的出现主要是为了解决 CPU 运算速度与内存读写速度不匹配的矛盾，因为 CPU 运算速度要比内存读写速度快很多，这样会使 CPU 花费很长时间等待数据到来或把数据写入内存。

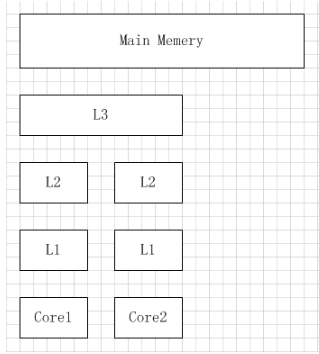
在缓存中的数据是内存中的一小部分，但这一小部分是短时间内 CPU 即将访问的，当 CPU 调用大量数据时，就可避开内存直接从缓存中调用，从而加快读取速度。

复制代码

CPU 和主内存之间有好几层缓存，因为即使直接访问主内存也是非常慢的。如果你正在多次对一块数据做相同的运算，那么在执行运算的时候把它加载到离 CPU 很近的地方就有意义了。

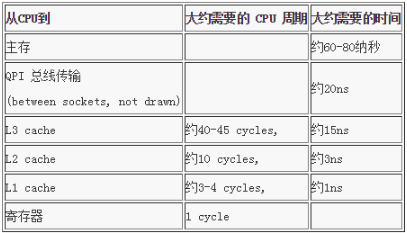
按照数据读取顺序和与 CPU 结合的紧密程度，CPU 缓存可以分为一级缓存，二级缓存，部分高端 CPU 还具有三级缓存。每一级缓存中所储存的全部数据都是下一级缓存的一部分，越靠近 CPU 的缓存越快也越小。所以 L1 缓存很小但很快(译注：L1 表示一级缓存)，并且紧靠着在使用它的 CPU 内核。L2 大一些，也慢一些，并且仍然只能被一个单独的 CPU 核使用。L3 在现代多核机器中更普遍，仍然更大，更慢，并且被单个插槽上的所有 CPU 核共享。最后，你拥有一块主存，由全部插槽上的所有 CPU 核共享。拥有三级缓存的的 CPU，到三级缓存时能够达到 95% 的命中率，只有不到 5% 的数据需要从内存中查询。

多核机器的存储结构如下图所示：



当 CPU 执行运算的时候，它先去 L1 查找所需的数据，再去 L2，然后是 L3，最后如果这些缓存中都没有，所需的数据就要去主内存拿。走得越远，运算耗费的时间就越长。所以如果你在做一些很频繁的事，你要确保数据在 L1 缓存中。

Martin Thompson 给出了一些缓存未命中的消耗数据，如下所示：



**二、MESI 协议及 RFO 请求**

从上一节中我们知道，每个核都有自己私有的 L1,、L2 缓存。那么多线程编程时, 另外一个核的线程想要访问当前核内 L1、L2 缓存行的数据, 该怎么办呢？

有人说可以通过第 2 个核直接访问第 1 个核的缓存行，这是当然是可行的，但这种方法不够快。跨核访问需要通过 Memory Controller（内存控制器，是计算机系统内部控制内存并且通过内存控制器使内存与 CPU 之间交换数据的重要组成部分），典型的情况是第 2 个核经常访问第 1 个核的这条数据，那么每次都有跨核的消耗.。更糟的情况是，有可能第 2 个核与第 1 个核不在一个插槽内，况且 Memory Controller 的总线带宽是有限的，扛不住这么多数据传输。所以，CPU 设计者们更偏向于另一种办法： 如果第 2 个核需要这份数据，由第 1 个核直接把数据内容发过去，数据只需要传一次。

那么什么时候会发生缓存行的传输呢？答案很简单：当一个核需要读取另外一个核的脏缓存行时发生。但是前者怎么判断后者的缓存行已经被弄脏(写)了呢？

下面将详细地解答以上问题. 首先我们需要谈到一个协议—— MESI 协议。现在主流的处理器都是用它来保证缓存的相干性和内存的相干性。M、E、S 和 I 代表使用 MESI 协议时缓存行所处的四个状态：

M（修改，Modified）：本地处理器已经修改缓存行，即是脏行，它的内容与内存中的内容不一样，并且此 cache 只有本地一个拷贝(专有)；

E（专有，Exclusive）：缓存行内容和内存中的一样，而且其它处理器都没有这行数据；

S（共享，Shared）：缓存行内容和内存中的一样, 有可能其它处理器也存在此缓存行的拷贝；

I（无效，Invalid）：缓存行失效, 不能使用。

下面说明这四个状态是如何转换的：

复制代码

初始：一开始时，缓存行没有加载任何数据，所以它处于 I 状态。

本地写（Local Write）：如果本地处理器写数据至处于 I 状态的缓存行，则缓存行的状态变成 M。

本地读（Local Read）：如果本地处理器读取处于 I 状态的缓存行，很明显此缓存没有数据给它。此时分两种情况：(1)其它处理器的缓存里也没有此行数据，则从内存加载数据到此缓存行后，再将它设成 E 状态，表示只有我一家有这条数据，其它处理器都没有；(2)其它处理器的缓存有此行数据，则将此缓存行的状态设为 S 状态。（备注：如果处于M状态的缓存行，再由本地处理器写入/读出，状态是不会改变的）

远程读（Remote Read）：假设我们有两个处理器 c1 和 c2，如果 c2 需要读另外一个处理器 c1 的缓存行内容，c1 需要把它缓存行的内容通过内存控制器 (Memory Controller) 发送给 c2，c2 接到后将相应的缓存行状态设为 S。在设置之前，内存也得从总线上得到这份数据并保存。

远程写（Remote Write）：其实确切地说不是远程写，而是 c2 得到 c1 的数据后，不是为了读，而是为了写。也算是本地写，只是 c1 也拥有这份数据的拷贝，这该怎么办呢？c2 将发出一个 RFO (Request For Owner) 请求，它需要拥有这行数据的权限，其它处理器的相应缓存行设为 I，除了它自已，谁不能动这行数据。这保证了数据的安全，同时处理 RFO 请求以及设置I的过程将给写操作带来很大的性能消耗。

我们从上节知道，写操作的代价很高，特别当需要发送 RFO 消息时。我们编写程序时，什么时候会发生 RFO 请求呢？有以下两种：

1. 线程的工作从一个处理器移到另一个处理器, 它操作的所有缓存行都需要移到新的处理器上。此后如果再写缓存行，则此缓存行在不同核上有多个拷贝，需要发送 RFO 请求了。

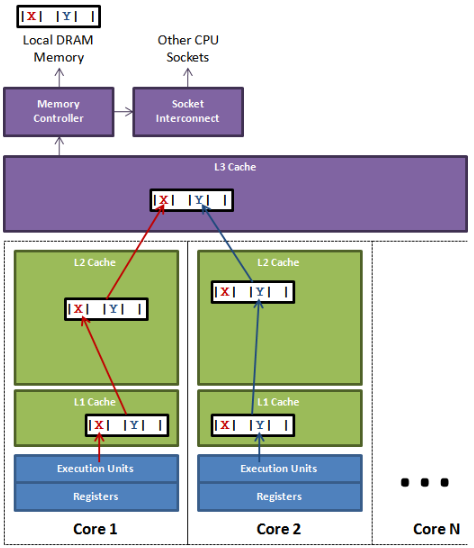
2. 两个不同的处理器确实都需要操作相同的缓存行

接下来，我们要了解什么是缓存行。

**三、缓存行**

在文章开头提到过，缓存系统中是以缓存行（cache line）为单位存储的。缓存行通常是 64 字节（译注：本文基于 64 字节，其他长度的如 32 字节等不适本文讨论的重点），并且它有效地引用主内存中的一块地址。一个 Java 的 long 类型是 8 字节，因此在一个缓存行中可以存 8 个 long 类型的变量。所以，如果你访问一个 long 数组，当数组中的一个值被加载到缓存中，它会额外加载另外 7 个，以致你能非常快地遍历这个数组。事实上，你可以非常快速的遍历在连续的内存块中分配的任意数据结构。而如果你在数据结构中的项在内存中不是彼此相邻的（如链表），你将得不到免费缓存加载所带来的优势，并且在这些数据结构中的每一个项都可能会出现缓存未命中。

如果存在这样的场景，有多个线程操作不同的成员变量，但是相同的缓存行，这个时候会发生什么？。没错，伪共享（False Sharing）问题就发生了！有张 Disruptor 项目的经典示例图，如下：



上图中，一个运行在处理器 core1上的线程想要更新变量 X 的值，同时另外一个运行在处理器 core2 上的线程想要更新变量 Y 的值。但是，这两个频繁改动的变量都处于同一条缓存行。两个线程就会轮番发送 RFO 消息，占得此缓存行的拥有权。当 core1 取得了拥有权开始更新 X，则 core2 对应的缓存行需要设为 I 状态。当 core2 取得了拥有权开始更新 Y，则 core1 对应的缓存行需要设为 I 状态(失效态)。轮番夺取拥有权不但带来大量的 RFO 消息，而且如果某个线程需要读此行数据时，L1 和 L2 缓存上都是失效数据，只有 L3 缓存上是同步好的数据。从前一篇我们知道，读 L3 的数据非常影响性能。更坏的情况是跨槽读取，L3 都要 miss，只能从内存上加载。

表面上 X 和 Y 都是被独立线程操作的，而且两操作之间也没有任何关系。只不过它们共享了一个缓存行，但所有竞争冲突都是来源于共享。

1. **遭遇伪共享**

**见代码。**

**五、如何避免伪共享？**

其中一个解决思路，就是让不同线程操作的对象处于不同的缓存行即可。

那么该如何做到呢？其实在我们注释的那行代码中就有答案，那就是缓存行填充（Padding） 。现在分析上面的例子，我们知道一条缓存行有 64 字节，而 Java 程序的对象头固定占 8 字节(32位系统)或 12 字节( 64 位系统默认开启压缩, 不开压缩为 16 字节)，所以我们只需要填 6 个无用的长整型补上6\*8=48字节，让不同的 VolatileLong 对象处于不同的缓存行，就避免了伪共享( 64 位系统超过缓存行的 64 字节也无所谓，只要保证不同线程不操作同一缓存行就可以)。

伪共享在多核编程中很容易发生，而且非常隐蔽。例如，在 JDK 的 LinkedBlockingQueue 中，存在指向队列头的引用 head 和指向队列尾的引用 tail 。而这种队列经常在异步编程中使有，这两个引用的值经常的被不同的线程修改，但它们却很可能在同一个缓存行，于是就产生了伪共享。线程越多，核越多，对性能产生的负面效果就越大。

由于某些 Java 编译器的优化策略，那些没有使用到的补齐数据可能会在编译期间被优化掉，我们可以在程序中加入一些代码防止被编译优化。如下：

public static long preventFromOptimization(VolatileLong v) {

return v.p1 + v.p2 + v.p3 + v.p4 + v.p5 + v.p6;

}

另外一种技术是使用编译指示，来强制使每一个变量对齐。

下面的代码显式了编译器使用\_\_declspec( align(n) ) 此处 n=64，按照 cache line 边界对齐。

\_\_declspec (align(64)) int thread1\_global\_variable;

\_\_declspec (align(64)) int thread2\_global\_variable;

当使用数组时，在 cache line 尾部填充 padding 来保证数据元素在 cache line 边界开始。如果不能够保证数组按照 cache line 边界对齐，填充数据结构【数组元素】使之是 cache line 大小的两倍。下面的代码显式了填充数据结构使之按照 cache line 对齐。并且通过 \_\_declspec( align(n) ) 语句来保证数组也是对齐的。如果数组是动态分配的，你可以增加分配的大小，并调整指针来对其到 cache line 边界。

复制代码

struct ThreadParams

{

// For the following 4 variables: 4\*4 = 16 bytes

unsigned long thread\_id;

unsigned long v; // Frequent read/write access variable

unsigned long start;

unsigned long end;

// expand to 64 bytes to avoid false-sharing

// (4 unsigned long variables + 12 padding)\*4 = 64

int padding[12];

};

复制代码

除此之外，在网上还有很多对伪共享的研究，提出了一些基于数据融合的方案，有兴趣的同学可以了解下。

**六、对于伪共享，我们在实际开发中该怎么做？**

通过上面大篇幅的介绍，我们已经知道伪共享的对程序的影响。那么，在实际的生产开发过程中，我们一定要通过缓存行填充去解决掉潜在的伪共享问题吗？

其实并不一定。

首先就是多次强调的，伪共享是很隐蔽的，我们暂时无法从系统层面上通过工具来探测伪共享事件。其次，不同类型的计算机具有不同的微架构（如 32 位系统和 64 位系统的 java 对象所占自己数就不一样），如果设计到跨平台的设计，那就更难以把握了，一个确切的填充方案只适用于一个特定的操作系统。还有，缓存的资源是有限的，如果填充会浪费珍贵的 cache 资源，并不适合大范围应用。最后，目前主流的 Intel 微架构 CPU 的 L1 缓存，已能够达到 80% 以上的命中率。

综上所述，并不是每个系统都适合花大量精力去解决潜在的伪共享问题。