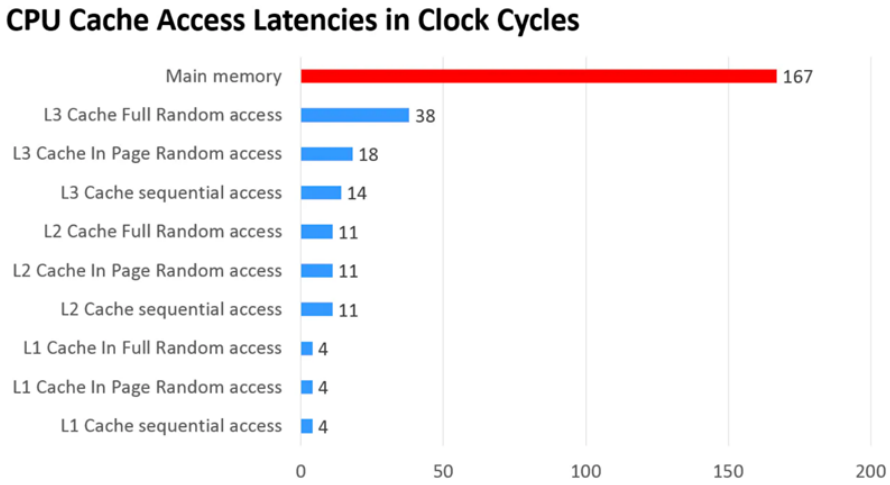
CPU高速缓存（Cache Memory）

CPU高速缓存

CPU缓存即高速缓冲存储器，是位于CPU与主内存间的一种容量较小但速度很高的存储器。由于CPU的速度远高于主内存，CPU直接从内存中存取数据要等待一定时间周期，Cache中保存着CPU刚

用过或循环使用的一部分数据，当CPU再次使用该部分数据时可从Cache中直接调用,减少CPU的等待时间，提高了系统的效率。



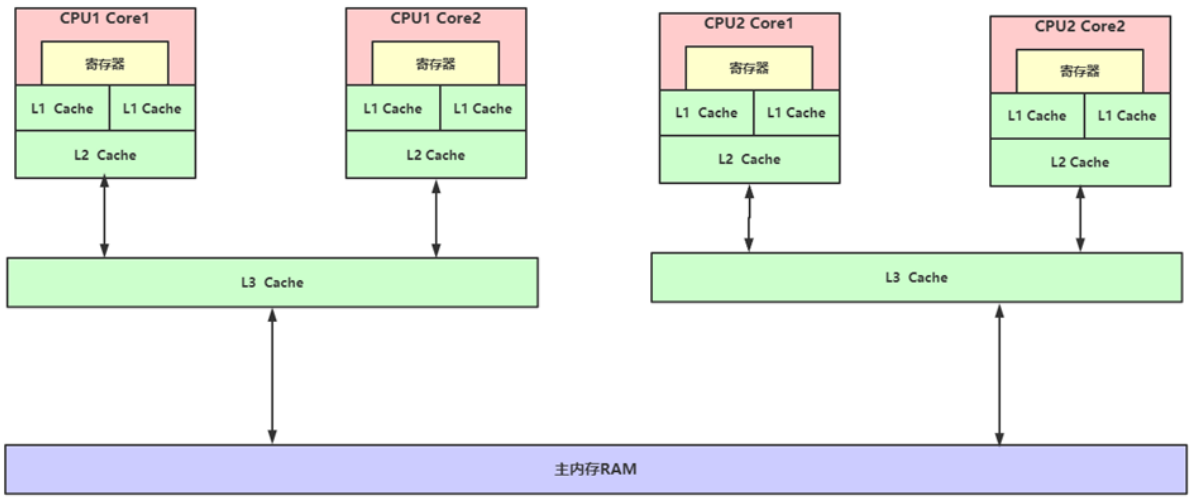
在CPU访问存储设备时，无论是存取数据抑或存取指令，都趋于聚集在一片连续的区域中，这就是局部性原理。

> **时间局部性（Temporal Locality）**：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。
> 比如循环、递归、方法的反复调用等。
> **空间局部性（Spatial Locality）**：如果一个存储器的位置被引用，那么将来他附近的位置也会被引用。
> 比如顺序执行的代码、连续创建的两个对象、数组等。

多CPU多核缓存架构

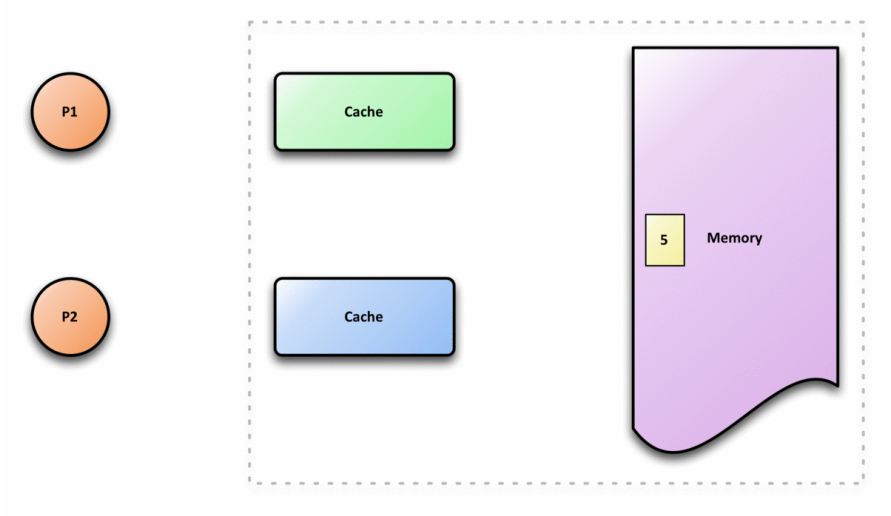
> 物理CPU：物理CPU就是插在主机上的真实的CPU硬件，在Linux下可以数不同的physical id 来确认主机的物理CPU个数。
> 核心数：我们常常会听说多核处理器，其中的核指的就是核心数。在Linux下可以通过cores来确认主机的物理CPU的核心数。
> 逻辑CPU：逻辑CPU跟超线程技术有联系，假如物理CPU不支持超线程的，那么逻辑CPU的数量等于核心数的数量；如果物理CPU支持超线程，那么逻辑CPU的数目是核心数数目的两倍。在Linux下可以通过 processors 的数目来确认逻辑CPU的数量。

现代CPU为了提升执行效率，减少CPU与内存的交互，一般在CPU上集成了多级缓存架构，常见的为三级缓存结构。

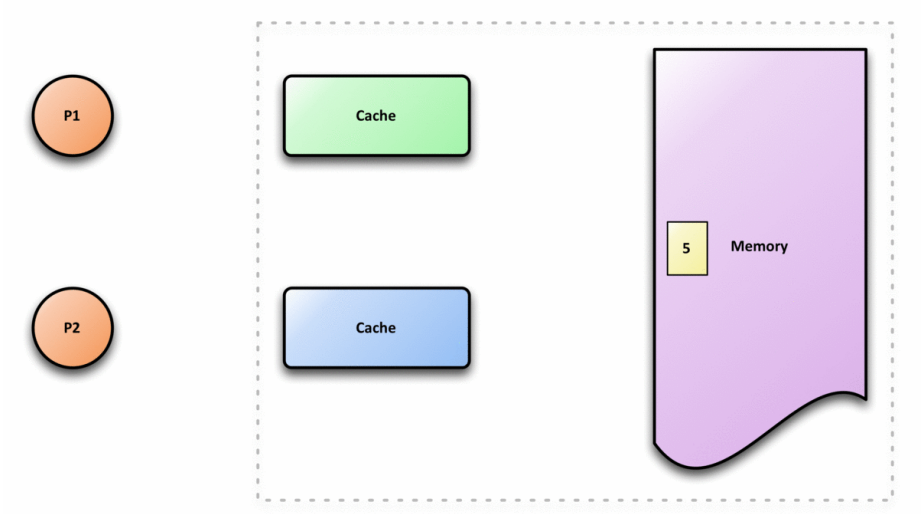


缓存一致性（Cache coherence）

计算机体系结构中，缓存一致性是共享资源数据的一致性，这些数据最终存储在多个本地缓存中。当系统中的客户机维护公共内存资源的缓存时，可能会出现数据不一致的问题，这在多处理系统中的cpu中尤其如此。



在共享内存多处理器系统中，每个处理器都有一个单独的缓存内存，共享数据可能有多个副本:一个副本在主内存中，一个副本在请求它的每个处理器的本地缓存中。当数据的一个副本发生更改时，其他副本必须反映该更改。缓存一致性是确保共享操作数(数据)值的变化能够及时地在整个系统中传播的规程。



缓存一致性的要求

写传播（Write Propagation）

对任何缓存中的数据的更改都必须传播到对等缓存中的其他副本(该缓存行的副本)。

事务串行化（Transaction Serialization）

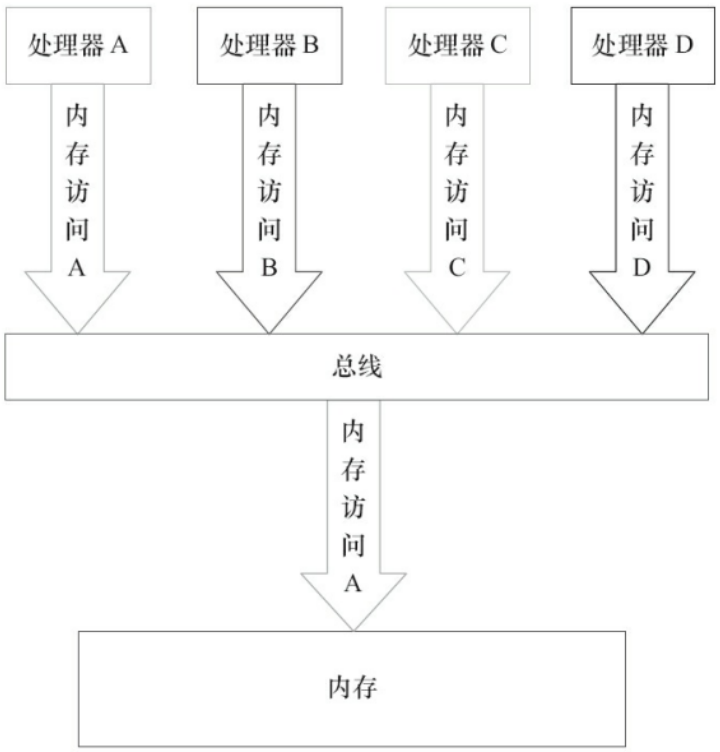
对单个内存位置的读/写必须被所有处理器以相同的顺序看到。理论上，一致性可以在加载/存储粒度上执行。然而，在实践中，它通常在缓存块的粒度上执行。

一致性机制（Coherence mechanisms）

确保一致性的两种最常见的机制是窥探机制（snooping ）和基于目录的机制（directory-based），这两种机制各有优缺点。如果有足够的带宽可用，基于协议的窥探往往会更快，因为所有事务都是所有处理器看到的请求/响应。其缺点是窥探是不可扩展的。每个请求都必须广播到系统中的所有节点，这意味着随着系统变大，(逻辑或物理)总线的大小及其提供的带宽也必须增加。另一方面，目录往往有更长的延迟(3跳 请求/转发/响应)，但使用更少的带宽，因为消息是点对点的，而不是广播的。由于这个原因，许多较大的系统(>64处理器)使用这种类型的缓存一致性。

总线仲裁机制

在计算机中，数据通过总线在处理器和内存之间传递。每次处理器和内存之间的数据传递都是通过一系列步骤来完成的，这一系列步骤称之为总线事务（Bus Transaction）。总线事务包括读事务（Read Transaction）和写事务（WriteTransaction）。读事务从内存传送数据到处理器，写事务从处理器传送数据到内存，每个事务会读/写内存中一个或多个物理上连续的字。这里的关键是，总线会同步试图并发使用总线的事务。在一个处理器执行总线事务期间，总线会禁止其他的处理器和I/O设备执行内存的读/写。



假设处理器A，B和C同时向总线发起总线事务，这时总线仲裁（Bus Arbitration）会对竞争做出裁决，这里假设总线在仲裁后判定处理器A在竞争中获胜（总线仲裁会确保所有处理器都能公平的访问内存）。此时处理器A继续它的总线事务，而其他两个处理器则要等待处理器A的总线事务完成后才能再次执行内存访问。假设在处理器A执行总线事务期间（不管这个总线事务是读事务还是写事务），处理器D向总线发起了总线事务，此时处理器D的请求会被总线禁止。

总线的这种工作机制可以把所有处理器对内存的访问以串行化的方式来执行。在任意时间点，最多只能有一个处理器可以访问内存。这个特性确保了单个总线事务之中的内存读/写操作具有原子性。

原子操作是指不可被中断的一个或者一组操作。处理器会自动保证基本的内存操作的原子性，也就是一个处理器从内存中读取或者写入一个字节时，其他处理器是不能访问这个字节的内存地址。最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

总线锁定

总线锁定就是使用处理器提供的一个 LOCK＃信号，当其中一个处理器在总线上输出此信号时，其它处理器的请求将被阻塞住，那么该处理器可以独占共享内存。

缓存锁定

由于总线锁定阻止了被阻塞处理器和所有内存之间的通信，而输出LOCK#信号的CPU可能只需要锁住特定的一块内存区域，因此总线锁定开销较大。

缓存锁定是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不会在总线上声言LOCK＃信号（总线锁定信号），而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

缓存锁定不能使用的特殊情况：

- 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用总线锁定。
- 有些处理器不支持缓存锁定。

《64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf》中有如下描述：

> The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:
> • Guaranteed atomic operations
> • Bus locking, using the LOCK# signal and the LOCK instruction prefix
> • Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

32位的IA-32处理器支持对系统内存中的位置进行锁定的原子操作。这些操作通常用于管理共享的数据结构(如信号量、段描述符、系统段或页表)，在这些结构中，两个或多个处理器可能同时试图修改相同的字段或标志。处理器使用三种相互依赖的机制来执行锁定的原子操作:

- 有保证的原子操作
- 总线锁定，使用LOCK#信号和LOCK指令前缀
- 缓存一致性协议，确保原子操作可以在缓存的数据结构上执行(缓存锁);这种机制出现在Pentium 4、Intel Xeon和P6系列处理器中

总线窥探(Bus Snooping)

总线窥探(Bus snooping)是缓存中的一致性控制器(snoopy cache)监视或窥探总线事务的一种方案，其目标是在分布式共享内存系统中维护缓存一致性。包含一致性控制器(snooper)的缓存称为snoopy缓存。该方案由Ravishankar和Goodman于1983年提出。

工作原理

当特定数据被多个缓存共享时，处理器修改了共享数据的值，更改必须传播到所有其他具有该数据副本的缓存中。这种更改传播可以防止系统违反缓存一致性。数据变更的通知可以通过总线窥探来完成。所有的窥探者都在监视总线上的每一个事务。如果一个修改共享缓存块的事务出现在总线上，所有的窥探者都会检查他们的缓存是否有共享块的相同副本。如果缓存中有共享块的副本，则相应的窥探者执行一个动作以确保缓存一致性。这个动作可以是刷新缓存块或使缓存块失效。它还涉及到缓存块状态的改变，这取决于缓存一致性协议（cache coherence protocol）。

窥探协议类型

根据管理写操作的本地副本的方式，有两种窥探协议:

Write-invalidate

当处理器写入一个共享缓存块时，其他缓存中的所有共享副本都会通过总线窥探失效。这种方法确保处理器只能读写一个数据的一个副本。其他缓存中的所有其他副本都无效。这是最常用的窥探协议。MSI、MESI、MOSI、MOESI和MESIF协议属于该类型。

Write-update

当处理器写入一个共享缓存块时，其他缓存的所有共享副本都会通过总线窥探更新。这个方法将写数据广播到总线上的所有缓存中。它比write-invalidate协议引起更大的总线流量。这就是为什么这种方法不常见。Dragon和firefly协议属于此类别。

一致性协议（Coherence protocol）

一致性协议在多处理器系统中应用于高速缓存一致性。为了保持一致性，人们设计了各种模型和协议，如MSI、MESI(又名Illinois)、MOSI、MOESI、MERSI、MESIF、write-once、Synapse、Berkeley、Firefly和Dragon协议。

- MSI protocol, the basic protocol from which the MESI protocol is derived.
- Write-once (cache coherency), an early form of the MESI protocol.
- MESI protocol
- MOSI protocol
- MOESI protocol
- MESIF protocol
- MERSI protocol
- Dragon protocol
- Firefly protocol

MESI协议

MESI协议是一个基于写失效的缓存一致性协议，是支持回写（write-back）缓存的最常用协议。也称作**伊利诺伊协议** (Illinois protocol，因为是在伊利诺伊大学厄巴纳-香槟分校被发明的)。与写通过（write through）缓存相比，回写缓冲能节约大量带宽。总是有“脏”（dirty）状态表示缓存中的数据与主存中不同。MESI协议要求在缓存不命中（miss）且数据块在另一个缓存时，允许缓存到缓存的数据复制。与MSI协议相比，MESI协议减少了主存的事务数量。这极大改善了性能。

状态

缓存行有4种不同的状态:

已修改Modified (M)

缓存行是脏的（dirty），与主存的值不同。如果别的CPU内核要读主存这块数据，该缓存行必须回写到主存，状态变为共享(S).

独占Exclusive (E)

缓存行只在当前缓存中，但是干净的--缓存数据同于主存数据。当别的缓存读取它时，状态变为共享；当前写数据时，变为已修改状态。

共享Shared (S)

缓存行也存在于其它缓存中且是未修改的。缓存行可以在任意时刻抛弃。

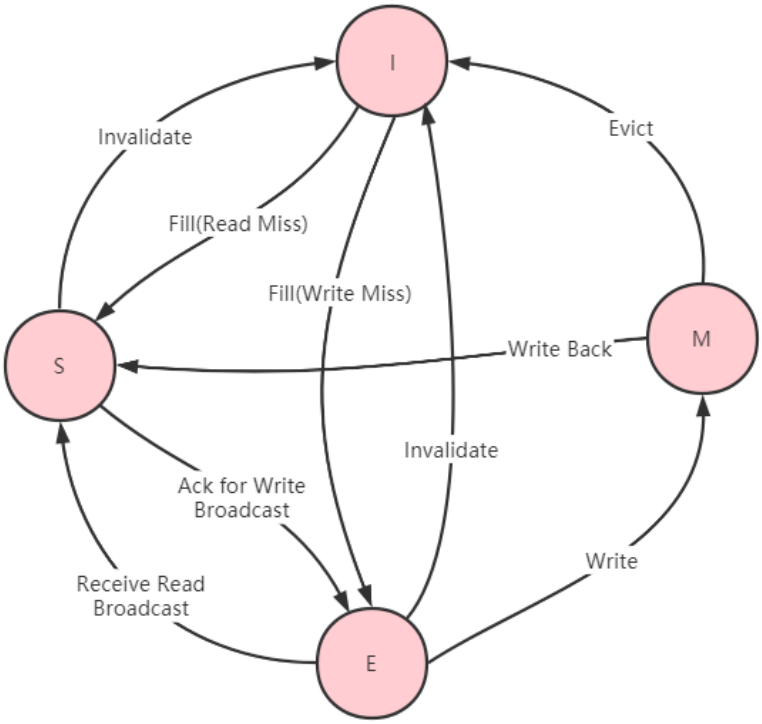
无效Invalid (I)

缓存行是无效的

任意一对缓存，对应缓存行的相容关系:

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

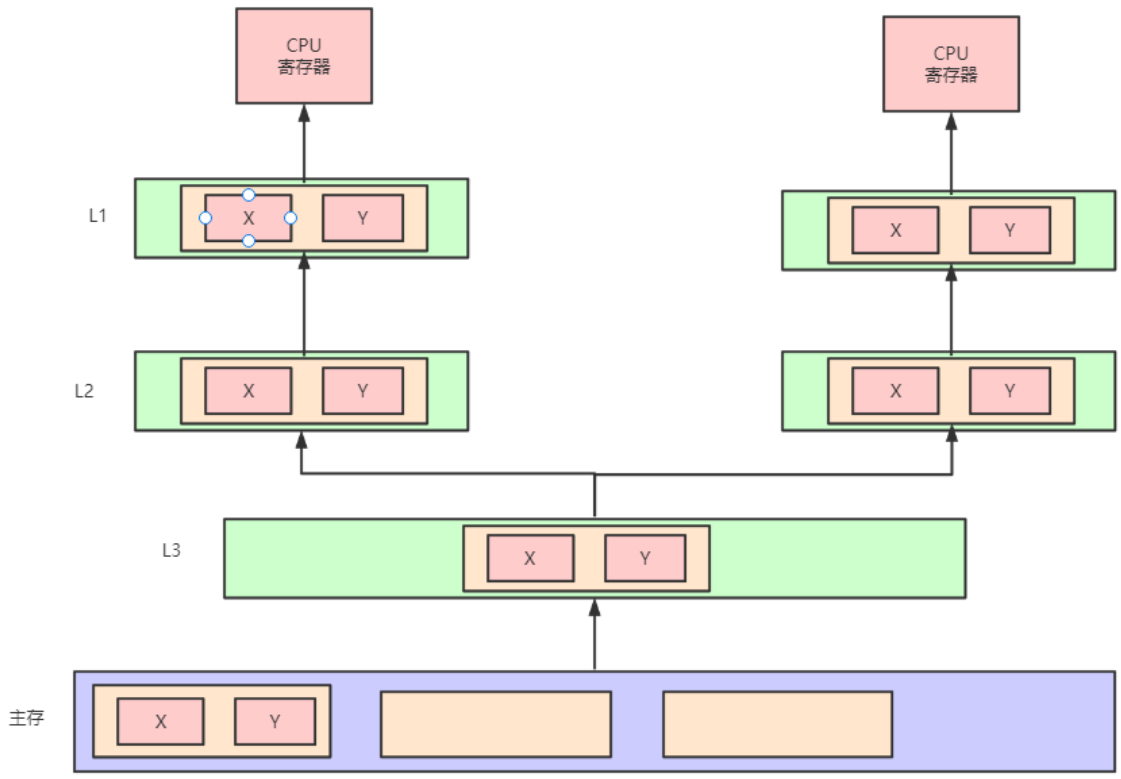
当块标记为 M (已修改), 在其他缓存中的数据副本被标记为I(无效).



MESI State Machine

伪共享的问题

如果多个核的线程在操作同一个缓存行中的不同变量数据，那么就会出现频繁的缓存失效，即使在代码层面看这两个线程操作的数据之间完全没有关系。这种不合理的资源竞争情况就是伪共享（False Sharing）。



linux下查看Cache Line大小

Cache Line大小是64Byte

```
[root@node02 ~]# cd /sys/devices/system/cpu/cpu0/cache/index0
[root@node02 index0]# ls
coherency_line_size  id  level  number_of_sets  physical_line_partition  shared_cpu_list  shared_cpu_map  size  type  ways_of_associativity
[root@node02 index0]# cat coherency_line_size
64
```

或者执行 cat /proc/cpuinfo 命令

```
adx smap clflushopt sha_ni xsaveopt xsavec xgetbv1 virt_ssbd
bogomips        : 3999.99
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

避免伪共享方案

1.缓存行填充

```
class Pointer {
    volatile long x;
    //避免伪共享:  缓存行填充
    long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
```

2.使用 @sun.misc.Contended 注解（java8）

注意需要配置jvm参数：-XX:-RestrictContended

测试

```
public class FalseSharingTest {

    public static void main(String[] args) throws InterruptedException {
        testPointer(new Pointer());
    }

    private static void testPointer(Pointer pointer) throws InterruptedException {
        long start = System.currentTimeMillis();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.x++;
            }
        });

```

```
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000000; i++) {
                pointer.y++;
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(pointer.x+","+pointer.y);

        System.out.println(System.currentTimeMillis() - start);


    }
}


class Pointer {
    // 避免伪共享:  @Contended +  jvm参数: -XX:-RestrictContended  jdk8支持
    //@Contended
    volatile long x;
    //避免伪共享:  缓存行填充
    //long p1, p2, p3, p4, p5, p6, p7;
    volatile long y;
```