37-理解CPUCache（上）：“4毫秒”究竟值多少钱？

在这一节内容开始之前，我们先来看一个3行的小程序。你可以猜一猜，这个程序里的循环1和循环2，运行所花费的时间会差多少？你可以先思考几分钟，然后再看我下面的解释。

```
int[] arr = new int[64 * 1024 * 1024];


// 循环1
for (int i = 0; i < arr.length; i++) arr[i] *= 3;


// 循环2
for (int i = 0; i < arr.length; i += 16) arr[i] *= 3
```

在这段Java程序中，我们首先构造了一个64×1024×1024大小的整型数组。在循环1里，我们遍历整个数组，将数组中每一项的值变成了原来的3倍；在循环2里，我们每隔16个索引访问一个数组元素，将这一项的值变成了原来的3倍。

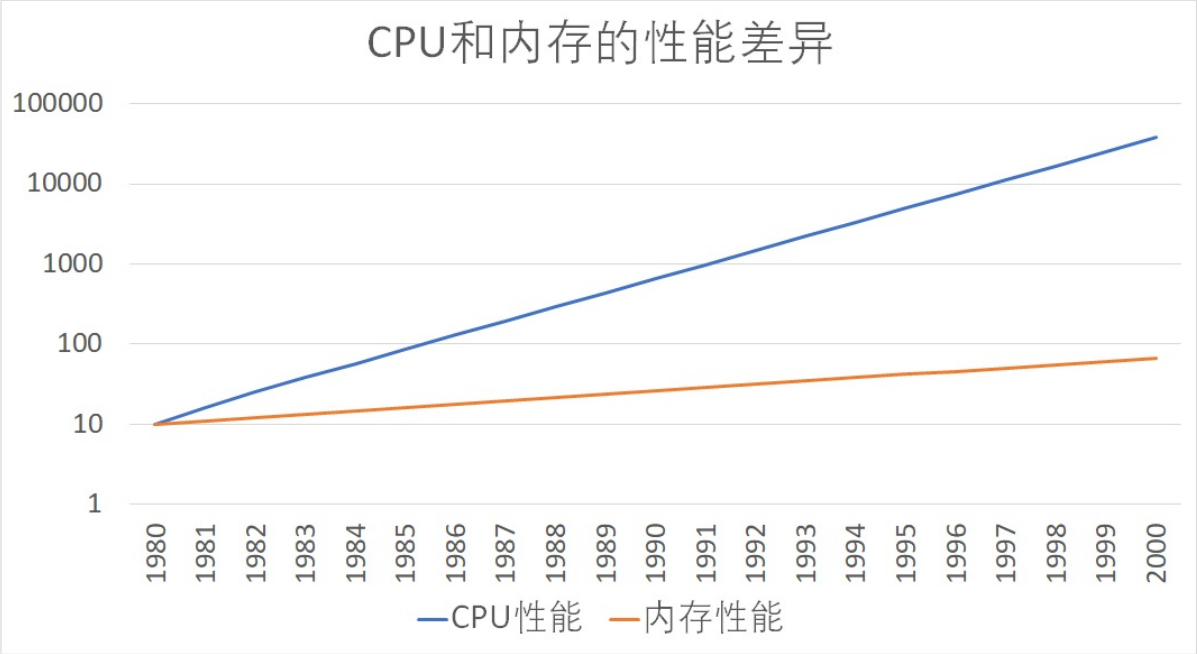
按道理来说，循环2只访问循环1中1/16的数组元素，只进行了循环1中1/16的乘法计算，那循环2花费的时间应该是循环1的1/16左右。但是实际上，循环1在我的电脑上运行需要50毫秒，循环2只需要46毫秒。这两个循环花费时间之差在15%之内。

为什么会有这15%的差异呢？这和我们今天要讲的CPU Cache有关。之前我们看到了内存和硬盘之间存在的巨大性能差异。在CPU眼里，内存也慢得不行。于是，聪明的工程师们就在CPU里面嵌入了CPU Cache（高速缓存），来解决这一问题。

我们为什么需要高速缓存?

按照摩尔定律，CPU的访问速度每18个月便会翻一番，相当于每年增长60%。内存的访问速度虽然也在不断增长，却远没有这么快，每年只增长7%左右。而这两个增长速度的差异，使得CPU性能和内存访问性能的差距不断拉大。到今天来看，一次内存的访问，大约需要120个CPU Cycle，这也意味着，在今天，CPU和内存的访问速度已经有了120倍的差距。

如果拿我们现实生活来打个比方的话，CPU的速度好比风驰电掣的高铁，每小时350公里，然而，它却只能等着旁边腿脚不太灵便的老太太，也就是内存，以每小时3公里的速度缓慢步行。因为CPU需要执行的指令、需要访问的数据，都在这个速度不到自己1%的内存里。

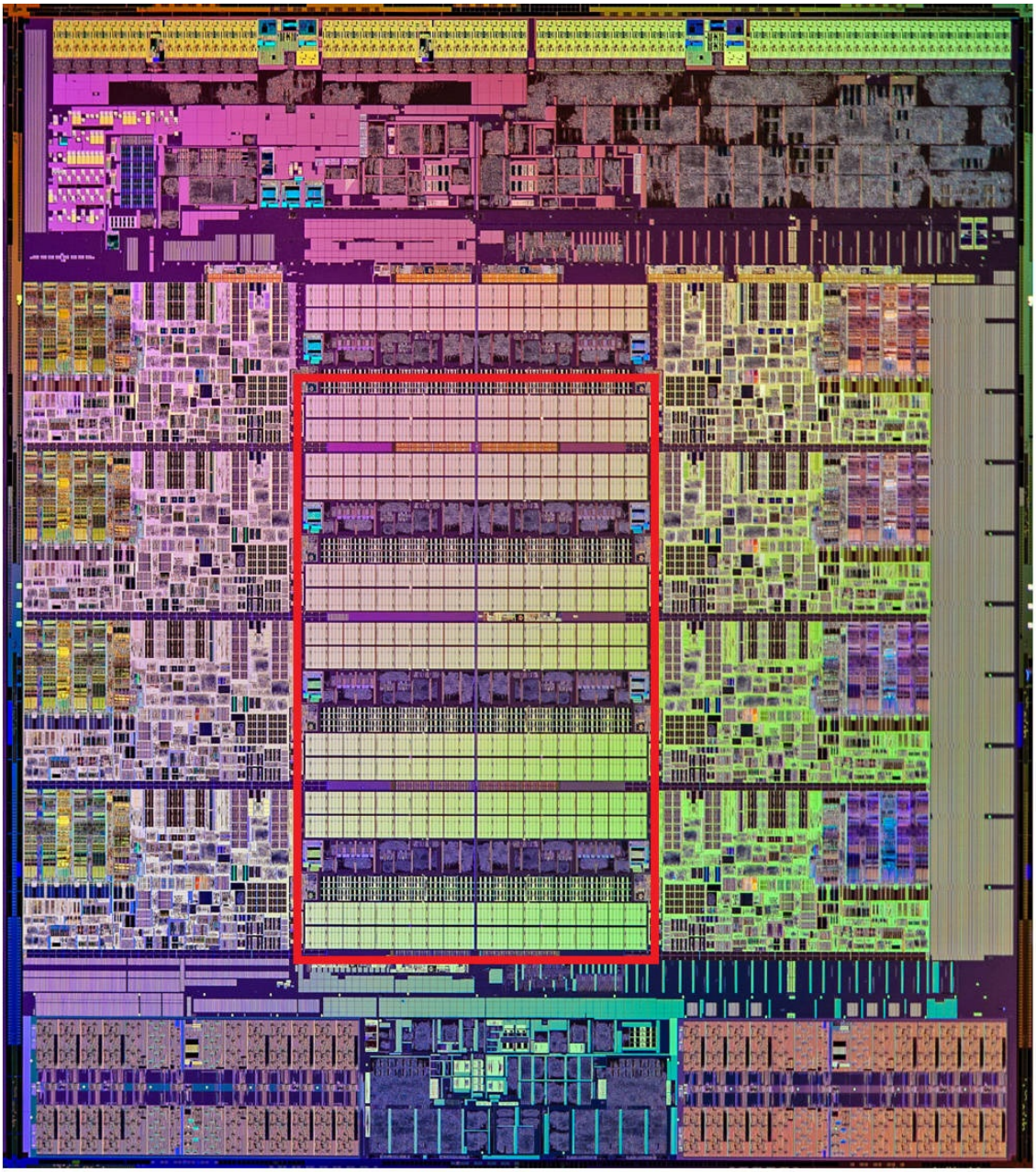


随着时间变迁，CPU和内存之间的性能差距越来越大

为了弥补两者之间的性能差异，我们能真实地把CPU的性能提升用起来，而不是让它在那儿空转，我们在现代CPU中引入了高速缓存。

从CPU Cache被加入到现有的CPU里开始，内存中的指令、数据，会被加载到L1-L3 Cache中，而不是直接由CPU访问内存去拿。在95%的情况下，CPU都只需要访问L1-L3 Cache，从里面读取指令和数据，而无需访问内存。要注意的是，这里我们说的CPU Cache或者L1/L3 Cache，不是一个单纯的、概念上的缓存（比如之前我们说的拿内存作为硬盘的缓存），而是指特定的由SRAM组成的物理芯片。

这里是一张Intel CPU的放大照片。这里面大片的长方形芯片，就是这个CPU使用的20MB的L3 Cache。



现代CPU中大量的空间已经被SRAM占据，图中用红色框出的部分就是CPU的L3 Cache芯片

在这一讲一开始的程序里，运行程序的时间主要花在了将对应的数据从内存中读取出来，加载到CPU Cache里。CPU从内存中读取数据到CPU Cache的过程中，是一小块一小块来读取数据的，而不是按照单个数组元素来读取数据的。这样一小块一小块的数据，在CPU Cache里面，我们把它叫作Cache Line（缓存块）。

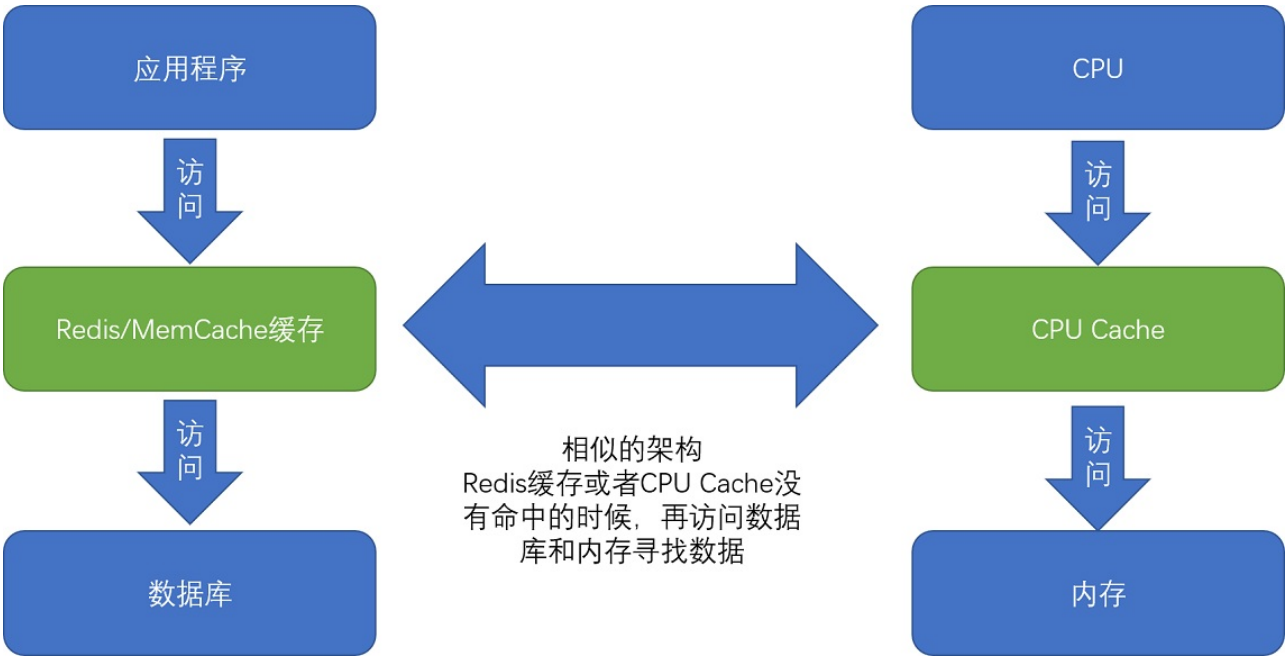
在我们日常使用的Intel服务器或者PC里，Cache Line的大小通常是64字节。而在上面的循环2里面，我们每隔16个整型数计算一次，16个整型数正好是64个字节。于是，循环1和循环2，需要把同样数量的Cache Line数据从内存中读取到CPU Cache中，最终两个程序花费的时间就差别不大了。

知道了为什么需要CPU Cache，接下来，我们就来看一看，CPU究竟是如何访问CPU Cache的，以及CPU Cache是如何组织数据，使得CPU可以找到自己想要访问的数据的。因为Cache作为“缓存”的意思，在很多别的存储设备里面都会用到。为了避免你混淆，在表示抽象的“缓存“概念时，用中文的“缓存”；如果是CPU Cache，我会用“高速缓存“或者英文的“Cache”，来表示。

Cache的数据结构和读取过程是什么样的？

现代CPU进行数据读取的时候，无论数据是否已经存储在Cache中，CPU始终会首先访问Cache。只有当

CPU在Cache中找不到数据的时候，才会去访问内存，并将读取到的数据写入Cache之中。当时间局部性原理起作用后，这个最近刚刚被访问的数据，会很快再次被访问。而Cache的访问速度远远快于内存，这样，CPU花在等待内存访问上的时间就大大变短了。

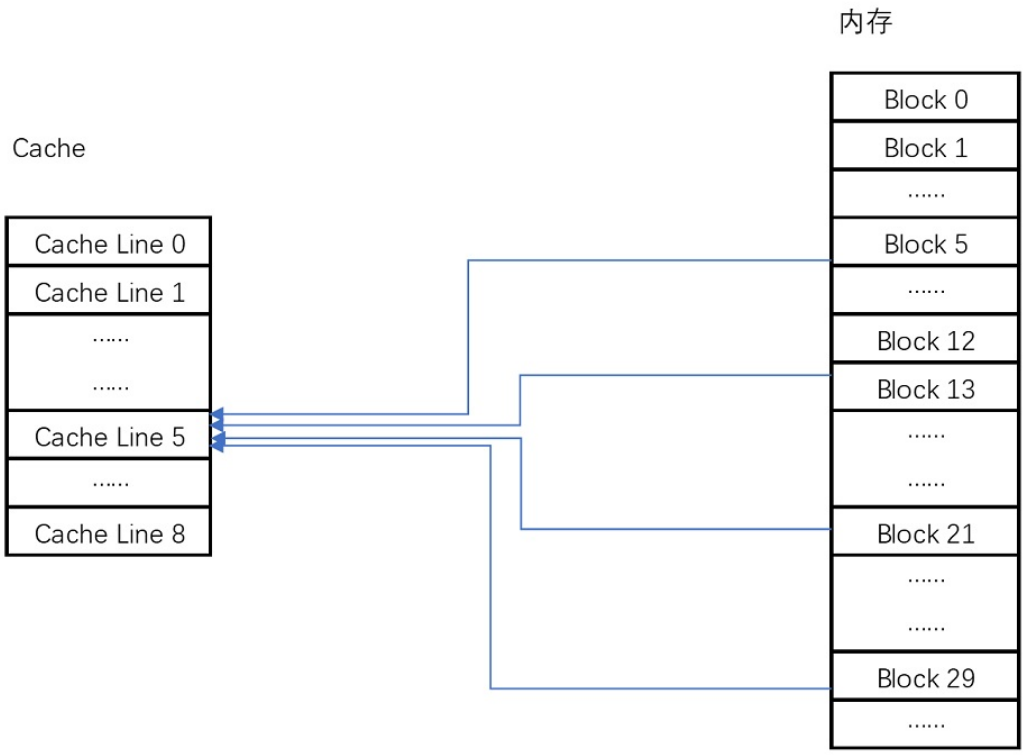


这样的访问机制，和我们自己在开发应用系统的时候，“使用内存作为硬盘的缓存”的逻辑是一样的。在各类基准测试（Benchmark）和实际应用场景中，CPU Cache的命中率通常能达到95%以上。

问题来了，CPU如何知道要访问的内存数据，存储在Cache的哪个位置呢？接下来，我就从最基本的**直接映射Cache**（Direct Mapped Cache）说起，带你来看整个Cache的数据结构和访问逻辑。

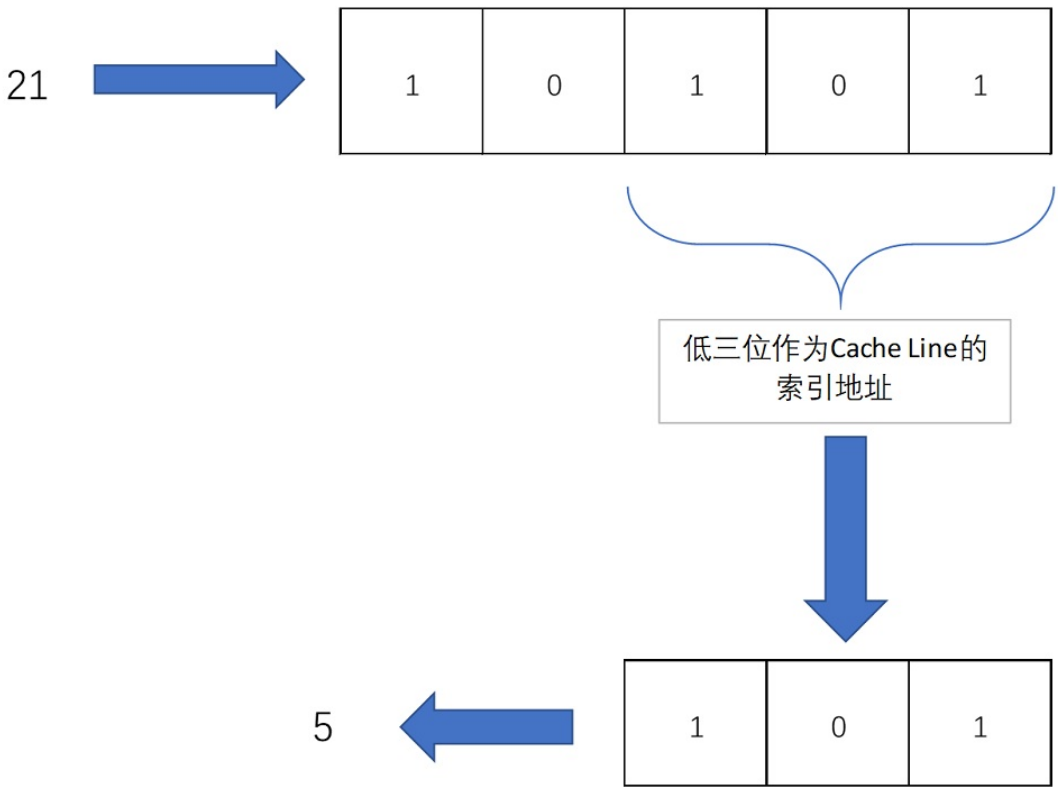
在开头的3行小程序里我说过，CPU访问内存数据，是一小块一小块数据来读取的。对于读取内存中的数据，我们首先拿到的是数据所在的**内存块**（Block）的地址。而直接映射Cache采用的策略，就是确保任何一个内存块的地址，始终映射到一个固定的CPU Cache地址（Cache Line）。而这个映射关系，通常用mod运算（求余运算）来实现。下面我举个例子帮你理解一下。

比如说，我们的主内存被分成0 ~ 31号这样32个块。我们一共有8个缓存块。用户想要访问第21号内存块。如果21号内存块内容在缓存块中的话，它一定在5号缓存块（21 mod 8 = 5）中。



Cache采用mod的方式，把内存块映射到对应的CPU Cache中

实际计算中，有一个小小的技巧，通常我们会把缓存块的数量设置成2的N次方。这样在计算取模的时候，可以直接取地址的低N位，也就是二进制里面的后几位。比如这里的8个缓存块，就是2的3次方。那么，在对21取模的时候，可以对21的2进制表示10101取地址的低三位，也就是101，对应的5，就是对应的缓存块地址。



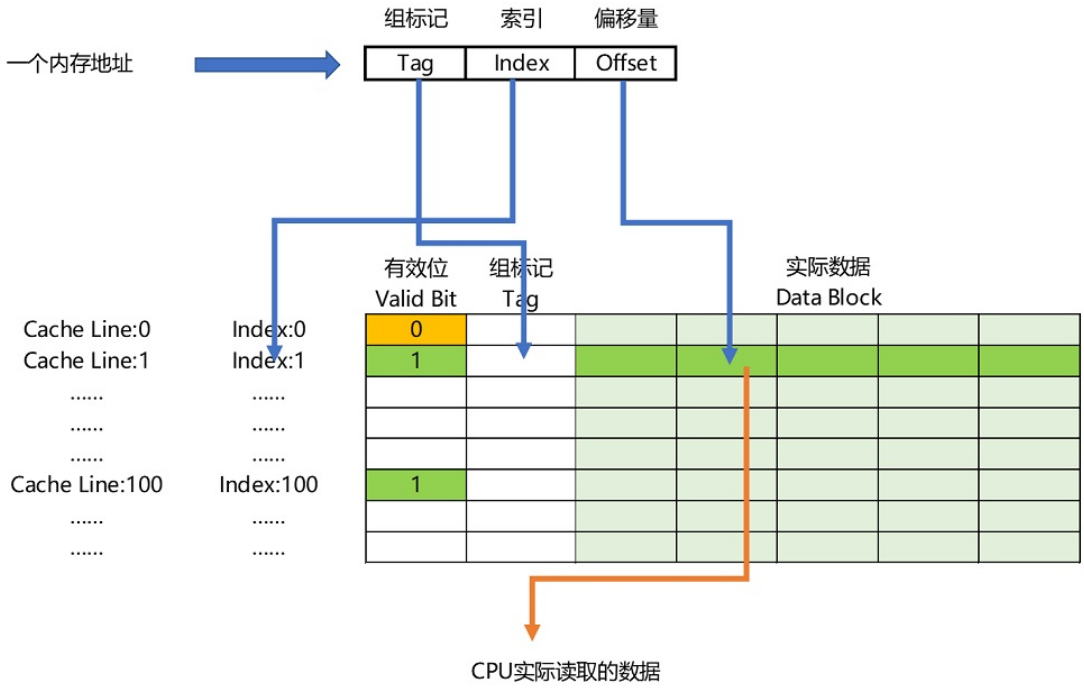
取Block地址的低位，就能得到对应的Cache Line地址，除了21号内存块外，13号、5号等很多内存块的数据，都对应着5号缓存块中。既然如此，假如现在CPU想要读取21号内存块，在读取到5号缓存块的时候，我们怎么知道里面的数据，究竟是不是21号对应的数据呢？同样，建议你借助现有知识，先自己思考一下，然后再看我下面的分析，这样会印象比较深刻。

这个时候，在对应的缓存块中，我们会存储一个**组标记**（Tag）。这个组标记会记录，当前缓存块内存储的数据对应的内存块，而缓存块本身的地址表示访问地址的低N位。就像上面的例子，21的低3位101，缓存块本身的地址已经涵盖了对应的信息、对应的组标记，我们只需要记录21剩余的高2位的信息，也就是10就可以了。

除了组标记信息之外，缓存块中还有两个数据。一个自然是从主内存中加载来的实际存放的数据，另一个是**有效位**（valid bit）。啥是有效位呢？它其实就是用来标记，对应的缓存块中的数据是否是有效的，确保不是机器刚刚启动时候的空数据。如果有效位是0，无论其中的组标记和Cache Line里的数据内容是什么，CPU都不会管这些数据，而要直接访问内存，重新加载数据。

CPU在读取数据的时候，并不是要读取一整个Block，而是读取一个他需要的整数。这样的数据，我们叫作CPU里的一个字（Word）。具体是哪个字，就用这个字在整个Block里面的位置来决定。这个位置，我们叫作偏移量（Offset）。

总结一下，**一个内存的访问地址，最终包括高位代表的组标记、低位代表的索引，以及在对应的Data Block中定位对应字的位置偏移量。**



内存地址到Cache Line的关系

而内存地址对应到Cache里的数据结构，则多了一个有效位和对应的数据，由“**索引 + 有效位 + 组标记 + 数据**”组成。如果内存中的数据已经在CPU Cache里了，那一个内存地址的访问，就会经历这样4个步骤：

1. 根据内存地址的低位，计算在Cache中的索引；
2. 判断有效位，确认Cache中的数据是有效的；
3. 对比内存访问地址的高位，和Cache中的组标记，确认Cache中的数据就是我们要访问的内存数据，从

Cache Line中读取到对应的数据块（Data Block）；
4. 根据内存地址的Offset位，从Data Block中，读取希望读取到的字。

如果在2、3这两个步骤中，CPU发现，Cache中的数据并不是要访问的内存地址的数据，那CPU就会访问内存，并把对应的Block Data更新到Cache Line中，同时更新对应的有效位和组标记的数据。

好了，讲到这里，相信你明白现代CPU，是如何通过直接映射Cache，来定位一个内存访问地址在Cache中的位置了。其实，除了直接映射Cache之外，我们常见的缓存放置策略还有全相连Cache（Fully Associative Cache）、组相连Cache（Set Associative Cache）。这几种策略的数据结构都是相似的，理解了最简单的直接映射Cache，其他的策略你很容易就能理解了。

减少4毫秒，公司挣了多少钱?

刚才我花了很多篇幅，讲了CPU和内存之间的性能差异，以及我们如何通过CPU Cache来尽可能解决这两者之间的性能鸿沟。你可能要问了，这样做的意义和价值究竟是什么？毕竟，一次内存的访问，只不过需要100纳秒而已。1秒钟时间内，足有1000万个100纳秒。别着急，我们先来看一个故事。

2008年，一家叫作Spread Networks的通信公司花费3亿美元，做了一个光缆建设项目。目标是建设一条从芝加哥到新泽西，总长1331公里的光缆线路。建设这条线路的目的，其实是为了将两地之间原有的网络访问延时，从17毫秒降低到13毫秒。

你可能会说，仅仅缩短了4毫秒时间啊，却花费3个亿，真的值吗？为这4毫秒时间买单的，其实是一批高频交易公司。它们以5年1400万美元的价格，使用这条线路。利用这短短的4毫秒的时间优势，这些公司通过高性能的计算机程序，在芝加哥和新泽西两地的交易所进行高频套利，以获得每年以10亿美元计的利润。现在你还觉得这个不值得吗？

其实，只要350微秒的差异，就足够高频交易公司用来进行无风险套利了。而350微秒，如果用来进行100纳秒一次的内存访问，大约只够进行3500次。而引入CPU Cache之后，我们可以进行的数据访问次数，提升了数十倍，使得各种交易策略成为可能。

总结延伸

很多时候，程序的性能瓶颈，来自使用DRAM芯片的内存访问速度。

根据摩尔定律，自上世纪80年代以来，CPU和内存的性能鸿沟越拉越大。于是，现代CPU的设计者们，直接在CPU中嵌入了使用更高性能的SRAM芯片的Cache，来弥补这一性能差异。通过巧妙地将内存地址，拆分成“索引+组标记+偏移量”的方式，使得我们可以将很大的内存地址，映射到很小的CPU Cache地址里。而CPU Cache带来的毫秒乃至微秒级别的性能差异，又能带来巨大的商业利益，十多年前的高频交易行业就是最好的例子。

在搞清楚从内存加载数据到Cache，以及从Cache里读取到想要的数据之后，我们又要面临一个新的挑战了。CPU不仅要读数据，还需要写数据，我们不能只把数据写入到Cache里面就结束了。下一讲，我们就来仔细讲讲，CPU要写入数据的时候，怎么既不牺牲性能，又能保证数据的一致性。

推荐阅读

如果你学有余力，这里有两篇文章推荐给你阅读。

如果想深入了解CPU和内存之间的访问性能，你可以阅读What Every Programmer Should Know About Memory。

现代CPU已经很少使用直接映射Cache了，通常用的是组相连Cache（set associative cache），想要了解组相连Cache，你可以阅读《计算机组成与设计：硬件/软件接口》的5.4.1小节。

课后思考

对于二维数组的访问，按行迭代和按列迭代的访问性能是一样的吗？你可以写一个程序测试一下，并思考一下原因。

欢迎把你思考的结果写在留言区。如果觉得有收获，你也可以把这篇文章分享给你的朋友，和他一起讨论和学习。

精选留言：

- 安排 2019-07-19 07:49:35
  cache和mmu位置关系是怎么样的？哪个在前哪个在后？