

Welcome

数据科学与大数据技术专业

计算机系统基础

上海体育学院经济管理学院

Wu Ying

回顾要点 (I)



理解计算机

1. 个人计算机的硬件组成
2. 存储程序 原理
3. 可编程
4. 冯·诺依曼体系结构
5. 总线
6. 0 - 1 序列 指挥电路动作
7. 高级语言、汇编语言、机器码
8. 计算机发展阶段、特点以及应用
9. 总线概述、总线分类、系统总线分类、总线仲裁
10. 位、字节、地址、存储单元、地址总线位数与寻址范围

理解运算

1. 电路怎样实现运算，与或非逻辑运算
2. 传统逻辑、布尔代数、香农开关
3. CPU的演化
4. 与主存一起完成自动加法计算

深入 CPU 和 主存

1. 一条指令的执行过程
2. 程序，多条指令的连续执行
3. 冯·诺依曼机的基本工作原理
4. 指令和机器码（指令的分类、格式）
5. 模型机
6. 指令周期（Instruction Cycle）、机器周期（CPU/Machine Cycle）、时钟周期（Clock Cycle）
7. 微操作
8. 抽象

指令集和系统抽象层次

1. 指令集 + 指令集体系结构 = 指令系统
2. 计算机系统的抽象层次、不同用户
3. 操作系统、用户接口、编译与解释

计算机系统

1. 硬件系统及软件系统的抽象层次
2. 软件系统及其分类（系统软件、支持软件、应用软件）
3. 应用操作：Windows tips、文字处理软件（Word、记事本、Markdown等）
4. 系统性能评价
程序执行时间、MIPS、Amdahl定律

数据的机器表示与处理

1. 编码、数字化
2. 数制、进制转换
3. 定点小数与定点整数
4. 浮点表示，规格化，IEEE754
5. 定点数的编码
6. 原码表示
7. 补码表示、特殊数据的补码表示
8. 整数的表示
9. 整数与浮点数类型转换
10. 浮点数的加法、精度损失
11. 数据在内存的排列顺序、位运算
12. 非数值型数据编码：字符汉字、多媒体（声音、图形图像、视频动画、存储容量、压缩、文件格式）

存储与I/O系统

1. 理解存储系统层次结构（寄存器、cache、内存、外存；SRAM, DRAM）
2. 局部性原理（时间局部性、空间局部性、LRU（Least Recently Used）缓存算法、缓存命中率）
3. Cache，弥补CPU与内存之间的性能鸿沟，**Cache Line缓存块**、缓存一致性、MESI协议
4. 虚拟内存、内存分段、内存分页、内存保护

其他

1. EXCEL应用
2. 网络与数据库应用

存储与I/O系统

01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

理解存储器的层次结构

大脑 ~= CPU
正在处理 ~= 寄存器
短期记忆 ~= L1 Cache
长期记忆 ~= L2/L3 Cache

书/资料 ~= 数据
书桌/书房 ~= 内存

图书馆 ~= SSD/HDD硬盘

CPU Cache , SRAM (Static Random-Access Memory, 静态随机存取存储器)



DRAM (Dynamic Random Access Memory, 动态随机存取存储器)

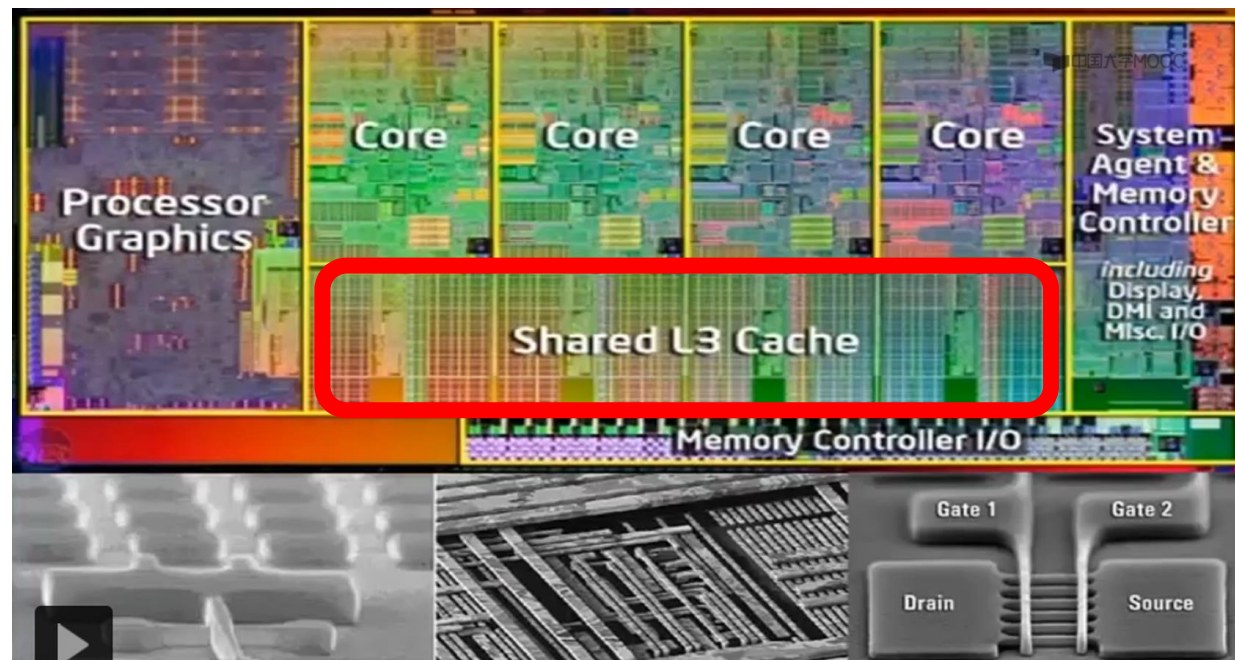
L1 Cache 往往嵌在 CPU 核心的内部。每个 CPU 核心有一块属于自己的 L1 高速缓存，通常分成**指令缓存和数据缓存**，分开存放 CPU 所用指令和数据

L2 Cache 同样是每个 CPU 核心都有的，不过它往往不在 CPU 核心的内部。所以 L2 Cache 的访问速度会比 L1 稍微慢一些。

L3 Cache，通常多个 CPU 核心共用，尺寸会更大一些，访问速度自然更慢一些。

当我们自己记忆中没有资料的时候，可以从书桌或者书架上拿书来翻阅。

这个过程中就相当于，**数据从内存中加载到 CPU 的寄存器和 Cache 中**，然后通过“大脑”，也就是 CPU，进行处理和运算。



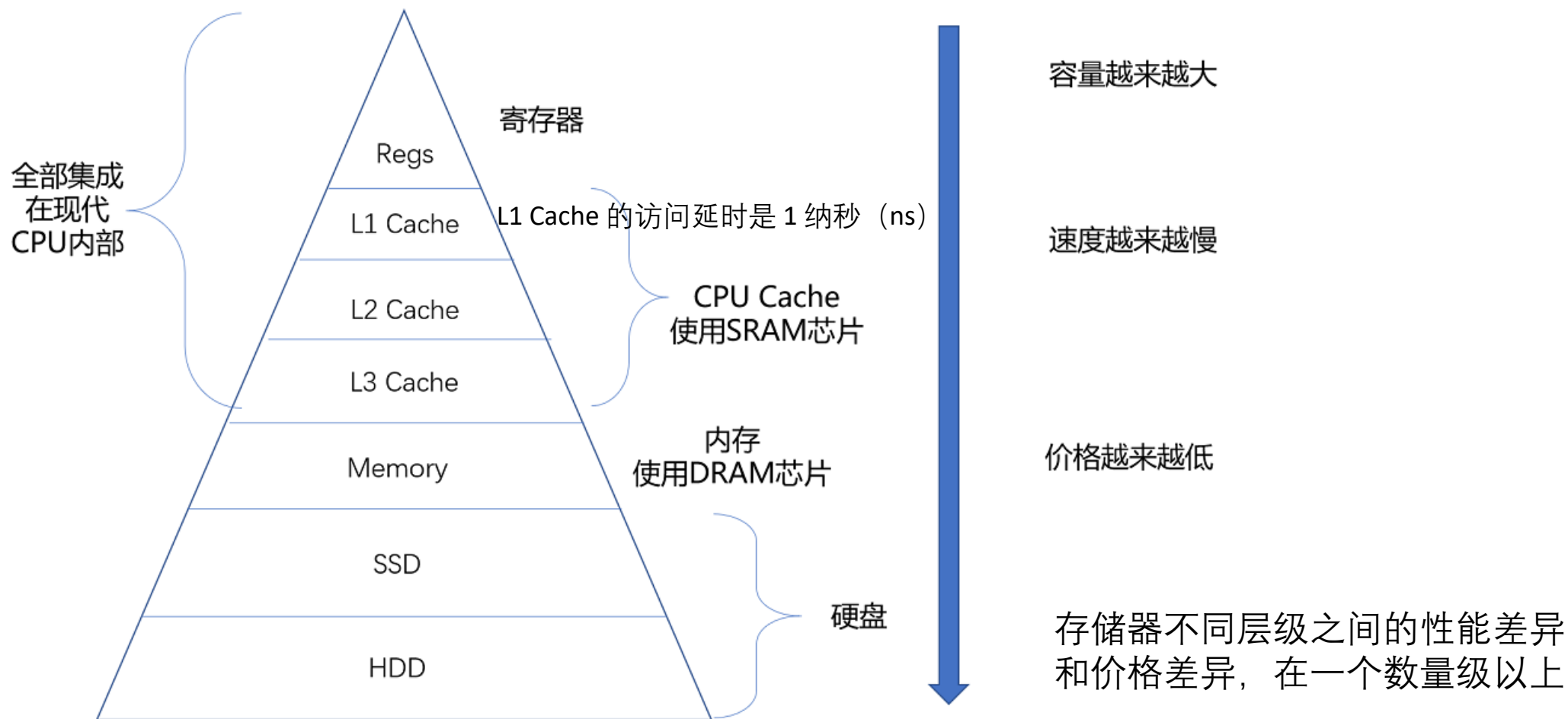
理解存储器的层次结构

内存芯片DRAM和 Cache(SRAM)在性能和价格上的差异

DRAM (Dynamic Random Access Memory, 动态随机存取存储器)

- DRAM 的一个比特，只需要一个晶体管和一个电容就能存储。所以，DRAM 在**同样的物理空间下**，能够存储的数据也就更多，也就是存储的“密度”更大，比 SRAM 芯片便宜不少。
- 数据是存储在电容里的，电容会不断漏电，所以需要定时刷新充电，才能保持数据不丢失。需要靠不断地“刷新”，才能保持数据被存储起来。
- DRAM 的数据访问电路和刷新电路都比 SRAM 更复杂，所以访问延时也就更长
- 同样断电丢失数据
- SRAM 更贵，速度更快。DRAM 更便宜，容量更大

存储器的层次结构



每一种存储器设备, 只和它相邻的存储设备打交道

存储器的层次结构

存储器	硬件介质	单位成本(美元/MB)	随机访问延时	说明
L1 Cache	SRAM	7	1ns	
L2 Cache	SRAM	7	4ns	访问延时15x L1 Cache
Memory	DRAM	0.015	100ns	访问延时15X SRAM, 价格1/40 SRAM
Disk	SSD(NAND)	0.0004	150μs	访问延时 1500X DRAM, 价格 1/40 DRAM
Disk	HDD	0.00004	10ms	访问延时 70X SSD, 价格 1/10 SSD

L1 Cache 256K

L2 Cache 1MB

L3 Cache 12MB

一共 13MB 的存储空间, 如果按照 7 美元 /1MB 的价格计算, 91 美元。

内存有 8GB, 容量是 CPU Cache 的 600 多倍, 120 美元。如果按目前京东上的价格, 恐怕不到 40 美元

128G 的 SSD 和 1T 的 HDD, 现在的价格加起来也不会超过 100 美元。虽然容量是内存的 16 倍乃至 128 倍, 但是它们的访问速度却不到内存的 1/1000

$$1s=1000ms=1000*1000us=1000*1000*1000ns$$

实际在进行电脑硬件配置的时候, 会去组合配置各种存储设备

https://colin-scott.github.io/personal_website/research/interactive_latency.html

存储器的层次结构

Intel i5-8265U 的 CPU (4 核)

这块 CPU 每个核有 32KB、**一共 128KB 的 L1 指令 Cache**

每个核还有 32KB，**一共 128KB 的 L1 数据 Cache**，指令 Cache 和数据 Cache 都是采用 8 路组相连的放置策略。

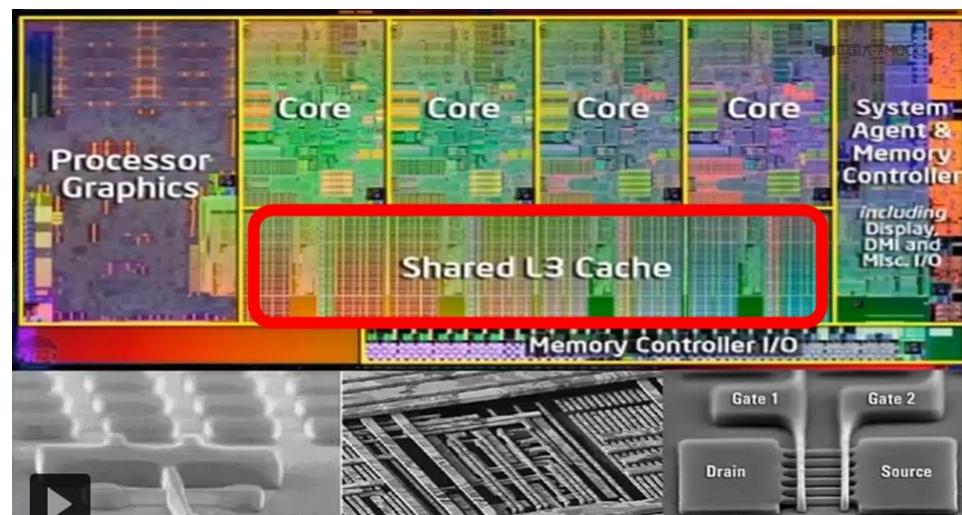
每个核有 256KB，一共 1MB 的 L2 Cache。L2 Cache 是用 4 路组相连的放置策略。

最后还有一块多个核心共用的 12MB 的 L3 Cache，采用的是 12 路组相连的放置策略。

8GB 的内存一块

128G 的 SSD 硬盘

1T 的 HDD 硬盘



存储与I/O系统

01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

存储器的层次结构

性能和价格的巨大差异，给我们工程师带来了一个挑战：

能不能既享受 CPU Cache 的速度，又享受内存、硬盘巨大的容量和低廉的价格呢？



局部性原理 Principle of Locality

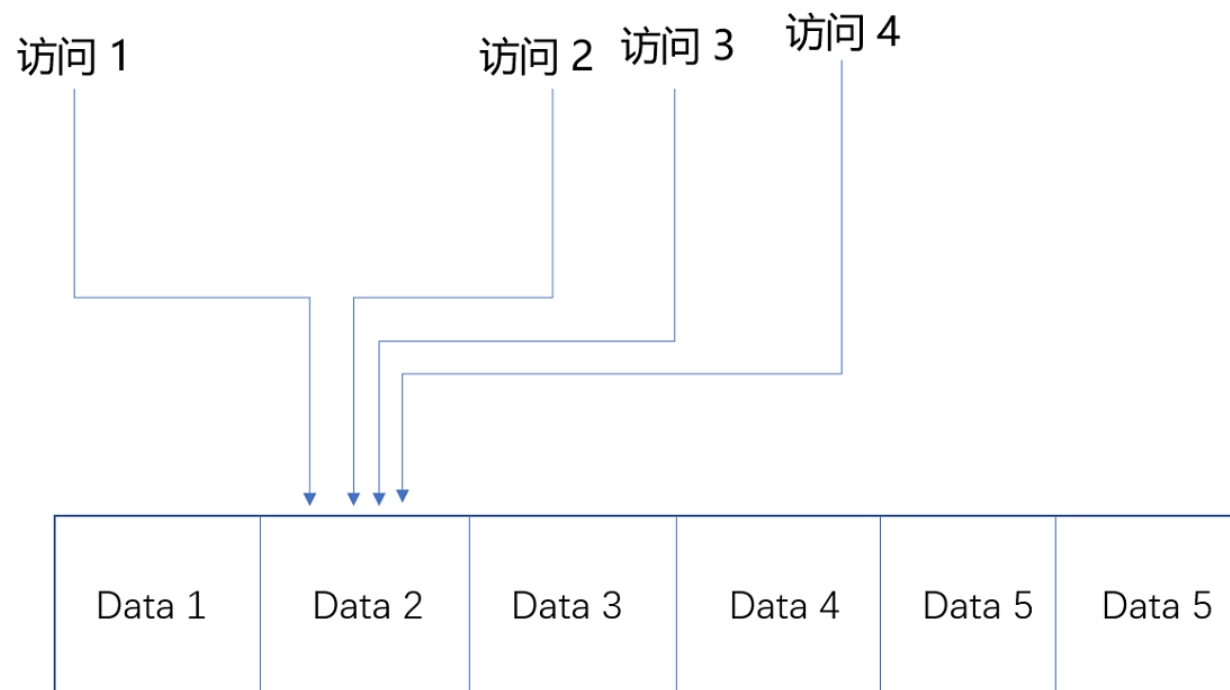
利用局部性原理，来制定管理和访问数据的策略：

- 时间局部性 (temporal locality)
- 空间局部性 (spatial locality)

局部性原理 Principle of Locality

- 时间局部性 (temporal locality)

如果一个数据被访问了，那么它在短时间内还会被再次访问



存储与I/O系统

01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

存储器的层次结构

存储器	硬件介质	单位成本(美元/MB)	随机访问延时	说明
L1 Cache	SRAM	7	1ns	
L2 Cache	SRAM	7	4ns	访问延时15x L1 Cache
Memory	DRAM	0.015	100ns	访问延时15X SRAM, 价格1/40 SRAM
Disk	SSD(NAND)	0.0004	150μs	访问延时 1500X DRAM, 价格 1/40 DRAM
Disk	HDD	0.00004	10ms	访问延时 70X SSD, 价格 1/10 SSD

L1 Cache 256K

L2 Cache 1MB

L3 Cache 12MB

一共 13MB 的存储空间, 如果按照 7 美元 /1MB 的价格计算, 91 美元。

内存有 8GB, 容量是 CPU Cache 的 600 多倍, 120 美元。如果按目前京东上的价格, 恐怕不到 40 美元

128G 的 SSD 和 1T 的 HDD, 现在的价格加起来也不会超过 100 美元。虽然容量是内存的 16 倍乃至 128 倍, 但是它们的访问速度却不到内存的 1/1000

$$1s=1000ms=1000*1000us=1000*1000*1000ns$$

实际在进行电脑硬件配置的时候, 会去组合配置各种存储设备

https://colin-scott.github.io/personal_website/research/interactive_latency.html



局部性原理应用，LRU算法与缓存命中率

类似亚马逊这样的电商网站，假设有 6 亿件商品，如果每件商品需要 4MB 的存储空间（考虑到商品图片的话，4MB 已经是一个相对较小的估计了），那么一共需要 2400TB（ $= 6 \text{ 亿} \times 4\text{MB}$ ）的数据存储。

如果把数据都放在内存里面，需要 3600 万美元（ $= 2400\text{TB} \times 0.015 \text{ 美元/1MB} = \mathbf{3600 \text{ 万美元}}$ ）。

但这 6 亿件商品中，不是每一件商品都会被经常访问。比如说，有 Kindle 电子书这样的热销商品，也一定有基本无人问津的商品，比如偏门的缅甸语词典

如果**只在内存里放前 1% 的热门商品**，也就是 600 万件热门商品，而把剩下的商品，放在机械式的 HDD 硬盘上，那么，要的存储成本就下降到 **45.6 万美元**（ $= 3600 \text{ 万美元} \times 1\% + 2400\text{TB} \times 0.00004 \text{ 美元/1MB}$ ），是原来成本的 1.3% 左右。



LRU算法与缓存命中率

把用户访问过的数据，加载到内存中，一旦内存里面放不下了，就把**最长时间没有在内存中被访问过的数据，从内存中移走**——LRU (Least Recently Used) 缓存算法。

热门商品被访问得多，就会始终被保留在内存里，而冷门商品被访问得少，就只存放在 HDD 硬盘上，数据的读取也都是直接访问硬盘。即使加载到内存中，也会很快被移除。

越是热门的商品，越容易在内存中找到

访问的数据中，可以在内存缓存中找到的，占有多大比例？

LRU 缓存策略的缓存命中率 (Hit Rate/Hit Ratio)



局部性原理应用，LRU算法与缓存命中率

以亚马逊 2017 年 3 亿的用户数来看，估算每天的活跃用户为 1 亿，这 1 亿用户每人平均会访问 100 个商品，那么**平均每秒访问的商品数量，就是 12 万次**。

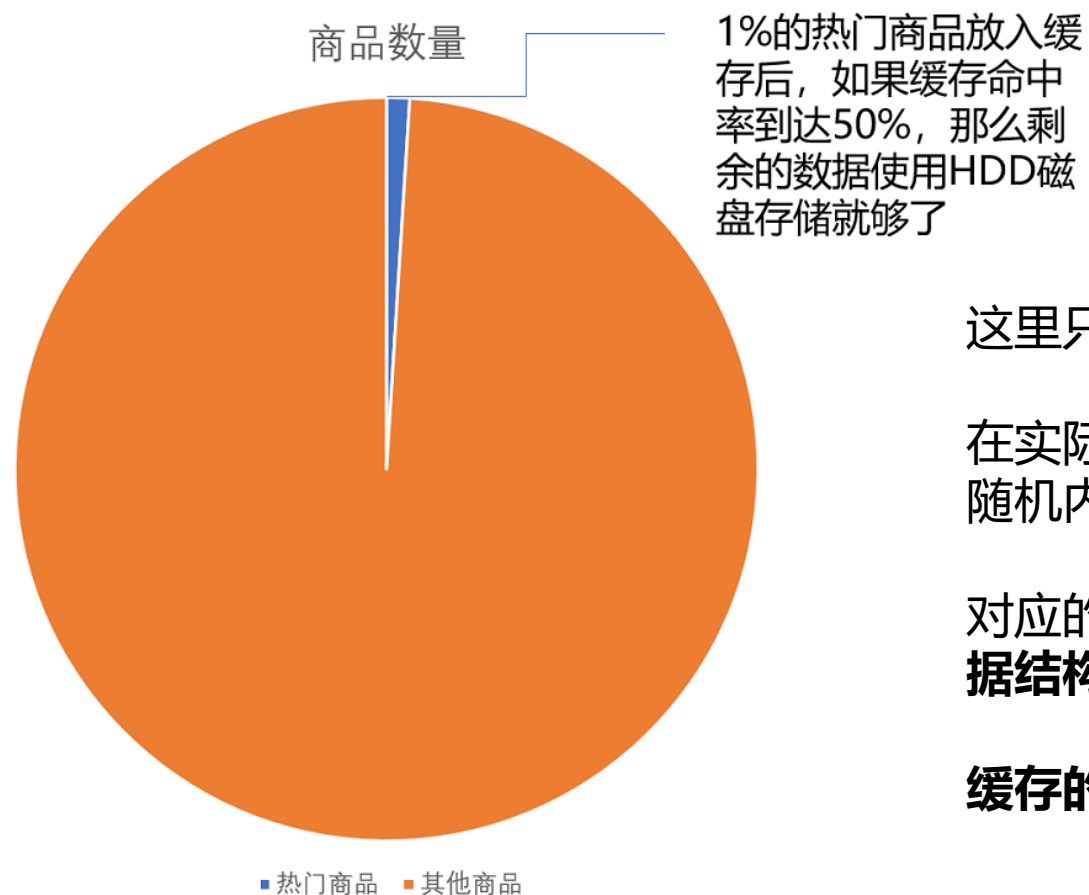
如果数据没有命中内存，对应的数据请求就要访问到HDD 磁盘了。一块 HDD 硬盘只能支撑每秒 100 次的随机访问，2400TB 的数据，以 4TB 一块磁盘来计算，有 600 块磁盘，**也就是能支撑每秒 6 万次**（ $= 2400\text{TB}/4\text{TB} \times 1\text{s}/10\text{ms}$ ）的随机访问。

至少要 50% 的缓存命中率，HDD 磁盘才能支撑对应的访问次数。不然的话，我们要么选择添加更多数量的 HDD 硬盘，做到每秒 12 万次的随机访问，或者将 HDD 替换成 SSD 硬盘，让单个硬盘可以支持更多的随机访问请求

内存的随机访问请求需要 100ns，访问一次内存需要100ns, 那么**1秒**可以访问 $1\text{s}/100\text{ns}=10,000,000$ 次在极限情况下，**内存可以支持 1秒1000 万次随机访问**。

我们用了 24TB 内存，如果 8G 一条的话，意味着有 3000 条内存，可以支持每秒 300 亿次（ $= 24\text{TB}/8\text{GB} \times 1\text{s}/100\text{ns}$ ）访问。

局部性原理应用，LRU算法与缓存命中率



这里只是一个简单的估算。

在实际的应用程序中，**查看一个商品的数据可能不止一次的随机内存或者随机磁盘的访问。**

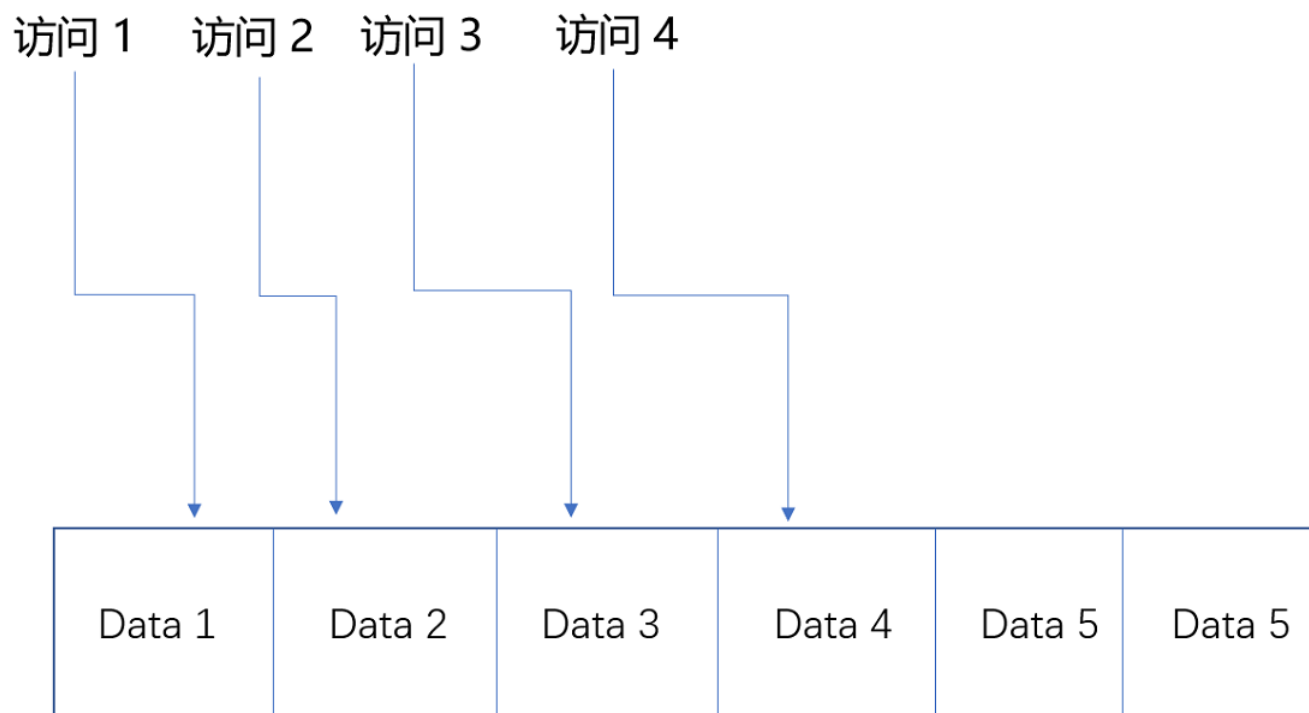
对应的数据存储空间也**不仅要考虑数据，还需要考虑维护数据结构的空间**

缓存的命中率和访问请求也要考虑均值和峰值

局部性原理 Principle of Locality

- 空间局部性 (spatial locality)

如果一个数据被访问了，那么**和它相邻的数据**也很快会被访问。



小结

- 访问次数多的数据，放在贵但是快一点的存储器里
- 访问次数少的数据，放在慢但是大一点的存储器里
- 组合使用内存、SSD 硬盘以及 HDD 硬盘，可以用最低的成本提供实际所需要的数据存储、管理和访问的需求。

局部性的存在，可以在应用开发中使用**缓存**这个有利的武器，组合利用不同层次的存储设备进行存储器的硬件规划。

需要考虑硬件的成本、访问的数据量以及访问的数据分布，根据对这些数据的估算，组合不同层次的存储器，**用尽可能低的成本支撑所需要的服务器压力。**

600块磁盘，通常很多服务器，一个主板上可以有8块至12块硬盘的接口，用50台服务器提供服务。

存储与I/O系统

01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

```

7 int main()
8 {
9     int i;
10    int n = 256 * 1024 * 1024;
11    clock_t timestart, timeend;
12    int* arr;
13
14    arr = (int*)malloc(sizeof(int) * n);
15
16    timestart = clock();
17    // 循环1
18    for (i = 0; i < n; i++)
19        arr[i] *= 3;
20    timeend = clock();
21    printf("time: %lfs\n", (double)(timeend - timestart) / CLOCKS_PER_SEC);
22
23    timestart = clock();
24    // 循环2
25    for (int i = 0; i < n; i += 16)
26        arr[i] *= 3;
27    timeend = clock();
28    printf("time: %lfs\n", (double)(timeend - timestart) / CLOCKS_PER_SEC);
29
30    free(arr); // 释放第一维指针
31 }
32

```

Microsoft Visual Stu

```

time: 0.540000s
time: 0.121000s
D:\Ccode\test2\Debug\te
按任意键关闭此窗口...

```

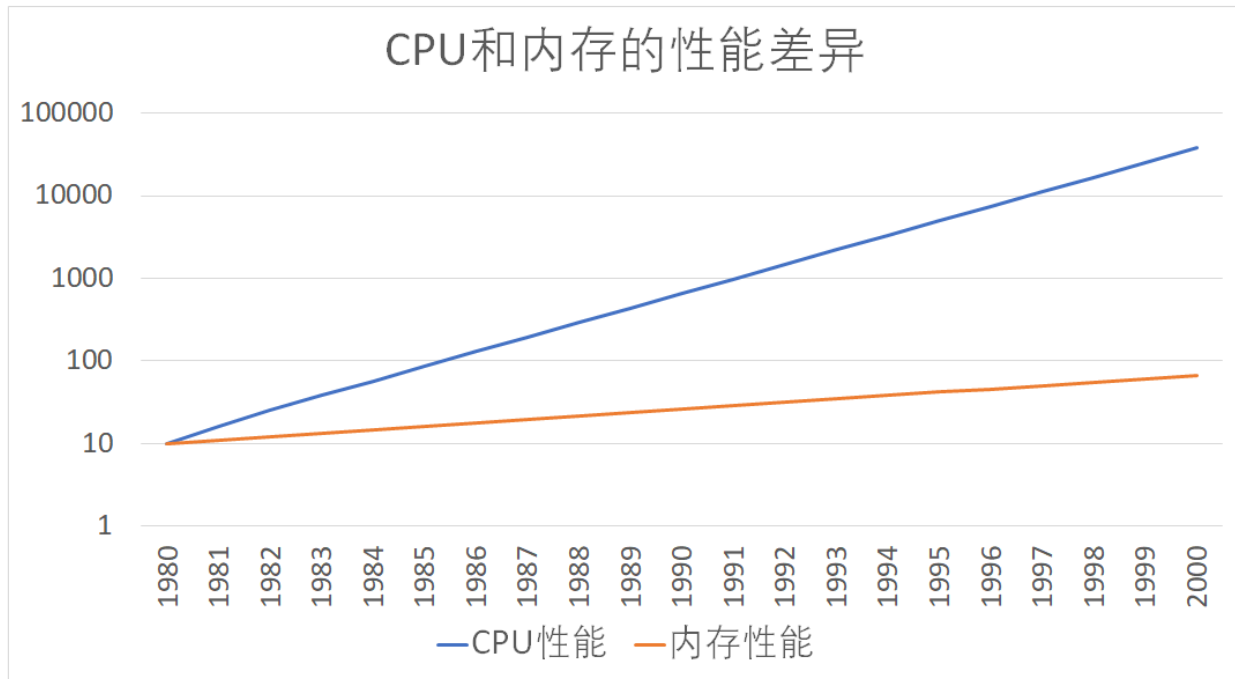
循环2并非循环1时间的1/16

在CPU 眼里，内存也慢得不行

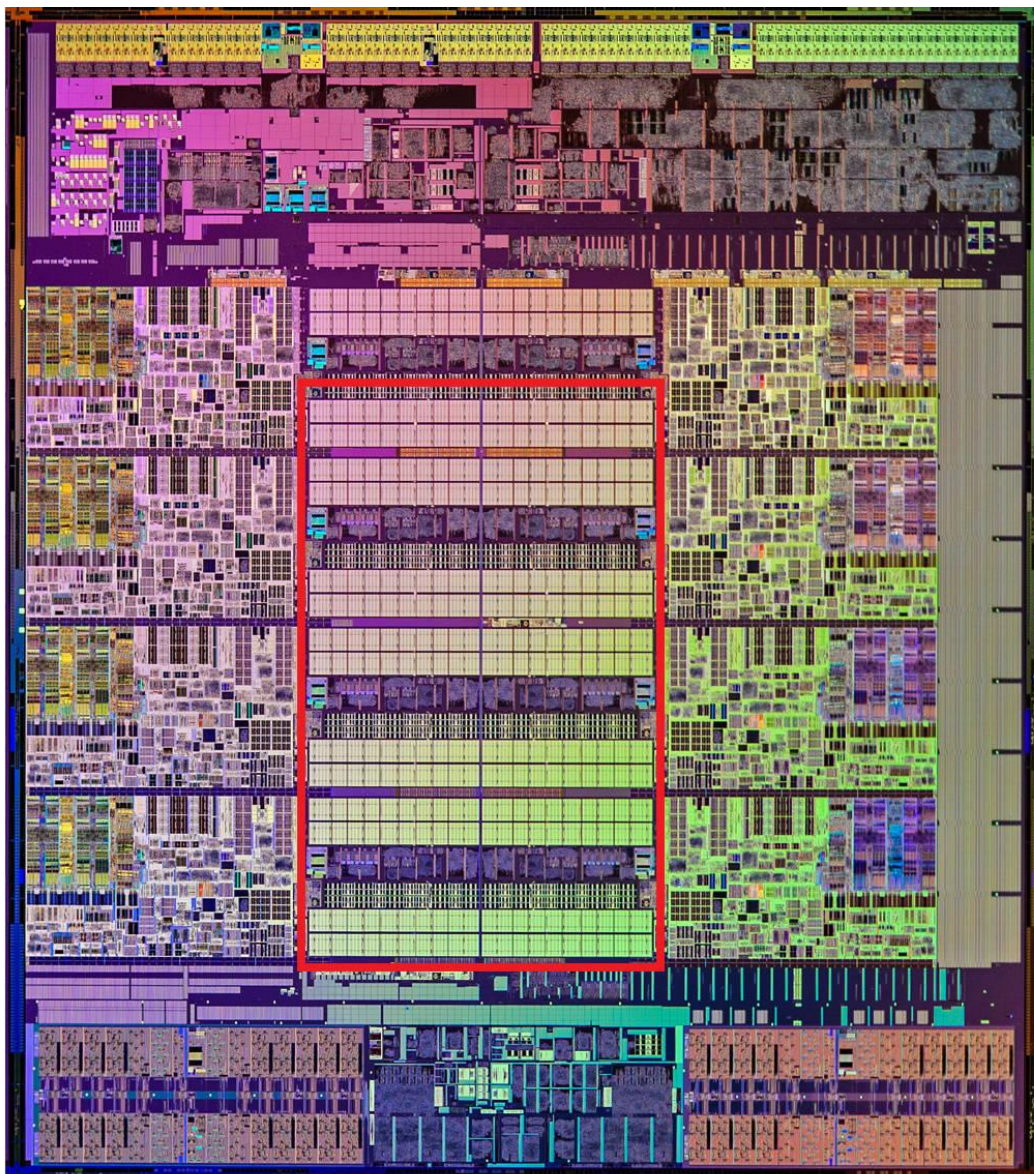
按照摩尔定律，CPU 的访问速度每 18 个月便会翻一番，相当于每年增长 60%，但内存的访问速度虽然也在不断增长，每年只增长 7% 左右。

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。

目前一次内存的访问，大约需要 120 个 CPU Cycle，这也意味着，CPU 和内存的访问速度已经有了 120 倍的差距。



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，即在 95% 的情况下，CPU 都只需要访问 L1-L3 Cache (SRAM芯片)，从里面读取指令和数据，而无需访问内存



20MB的L3 cache，占据CPU相当比例的面积

CPU 从内存中读取数据到 CPU Cache，一小块一小块读取，并非按照单个数组元素来读取

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），日常使用的 Intel 服务器或者 PC，通常是 64 字节

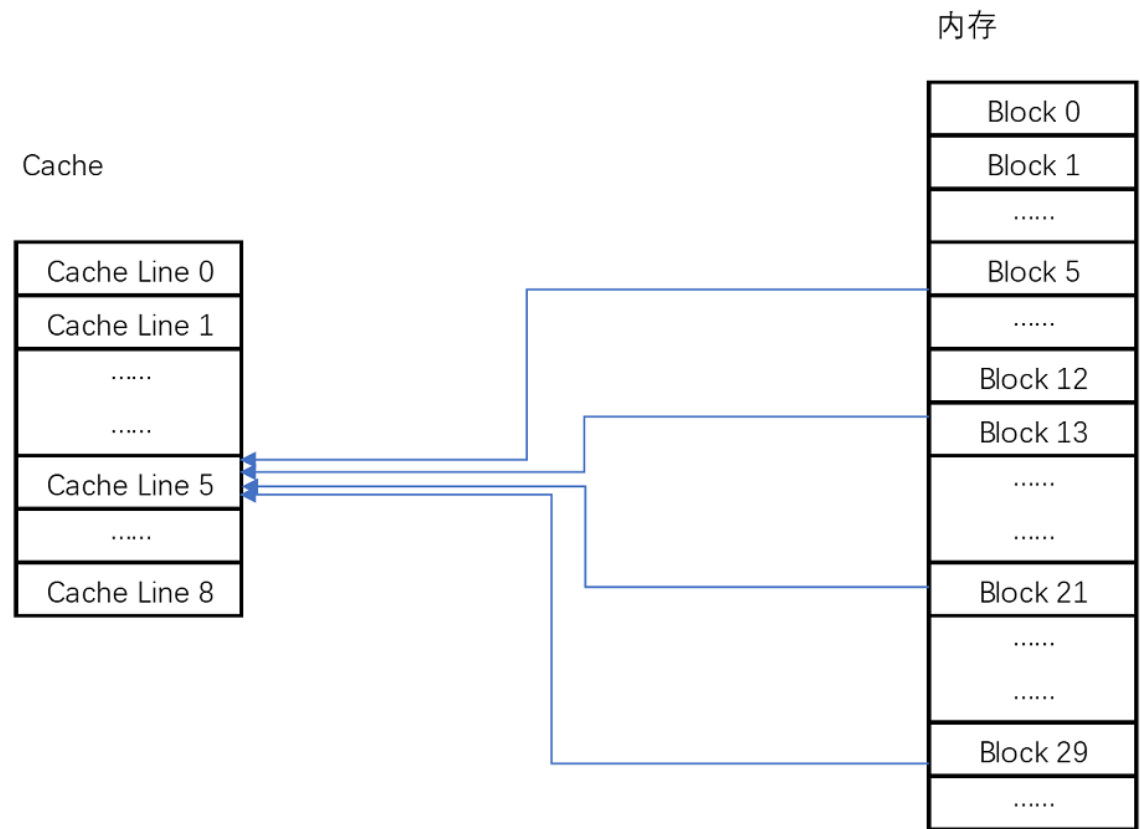
“缓存”——很多别的存储设备也有此概念。为避免混淆，在表示一般的“缓存”用中文的“缓存”；

如果是 CPU Cache，特指CPU内部的缓存，一般用“高速缓存”或者英文“Cache”表示。

循环 2 里面，每隔 16 个整型数，取出数组元素计算一次，16 个整型数正好是 64 个字节。所以循环 1 和循环 2，把同样数量的 Cache Line 数据从内存中读取到 CPU Cache 中，故而循环2并非循环1时间的1/16

CPU 如何知道要访问的内存数据，存储在 Cache 的哪个位置呢？

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。**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）**



例，主内存被分成 0 ~ 31 号，32 个块。一共有 8 个高速缓存块

用户想要访问第 21 号内存块，如果 21 号内存块内容在缓存中的话，一定在 5 号缓存块 ($21 \bmod 8 = 5$)

通常，缓存块的数量是 2 的 N 次方，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，21 的二进制 10101，取低 3 位

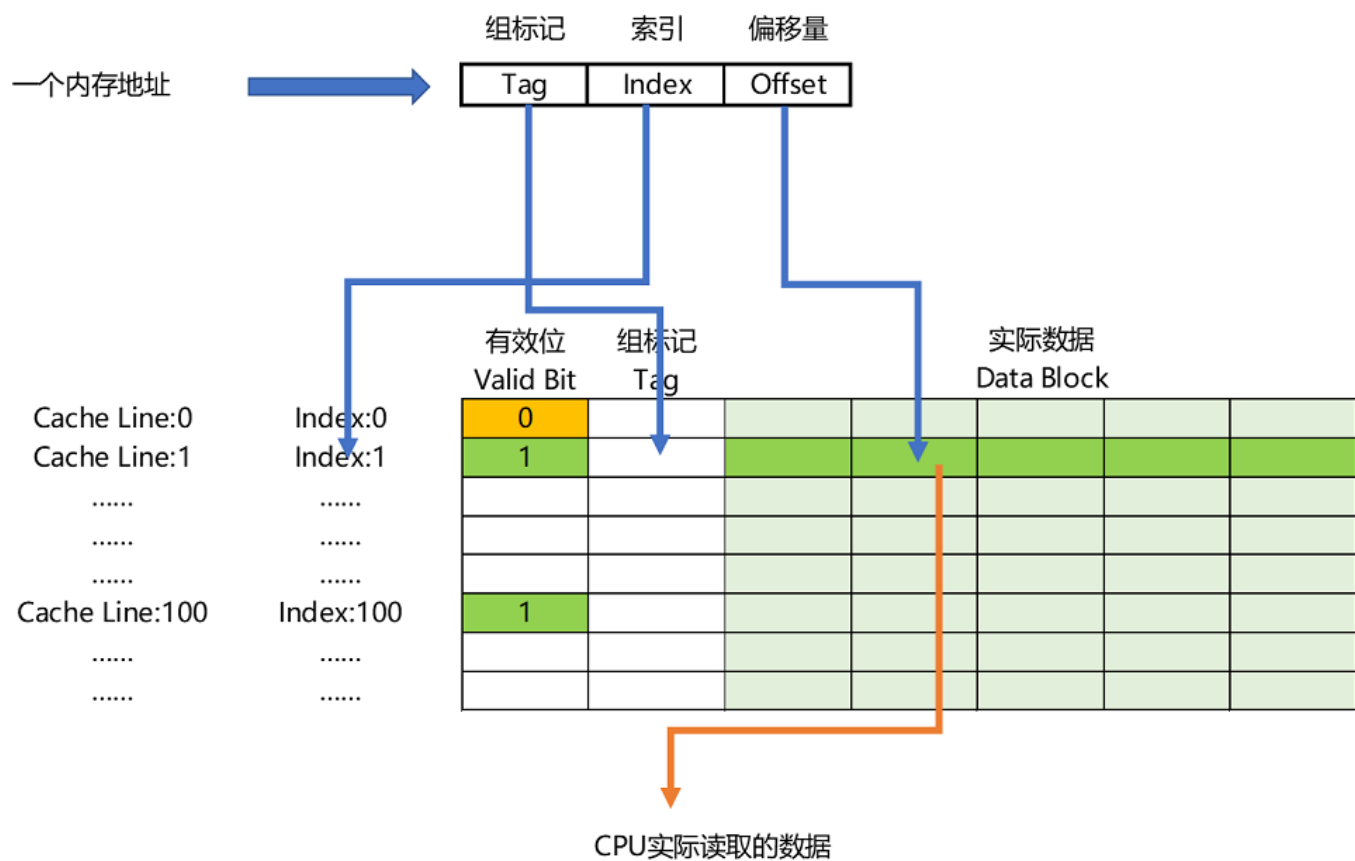
除了 21 号内存块外，29号、13 号、5 号内存块的数据，也对应着 5 号缓存块 Cache Line

Cache缓存块中，会存储一个组标记（Tag），记录当前缓存块内存储的数据**对应的内存块**

除了组标记信息之外，缓存块中还有两个数据：

一个是从主内存中加载来的实际存放的数据；

一个是有效位（valid bit），用来标记缓存块中的数据是否有效，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。**

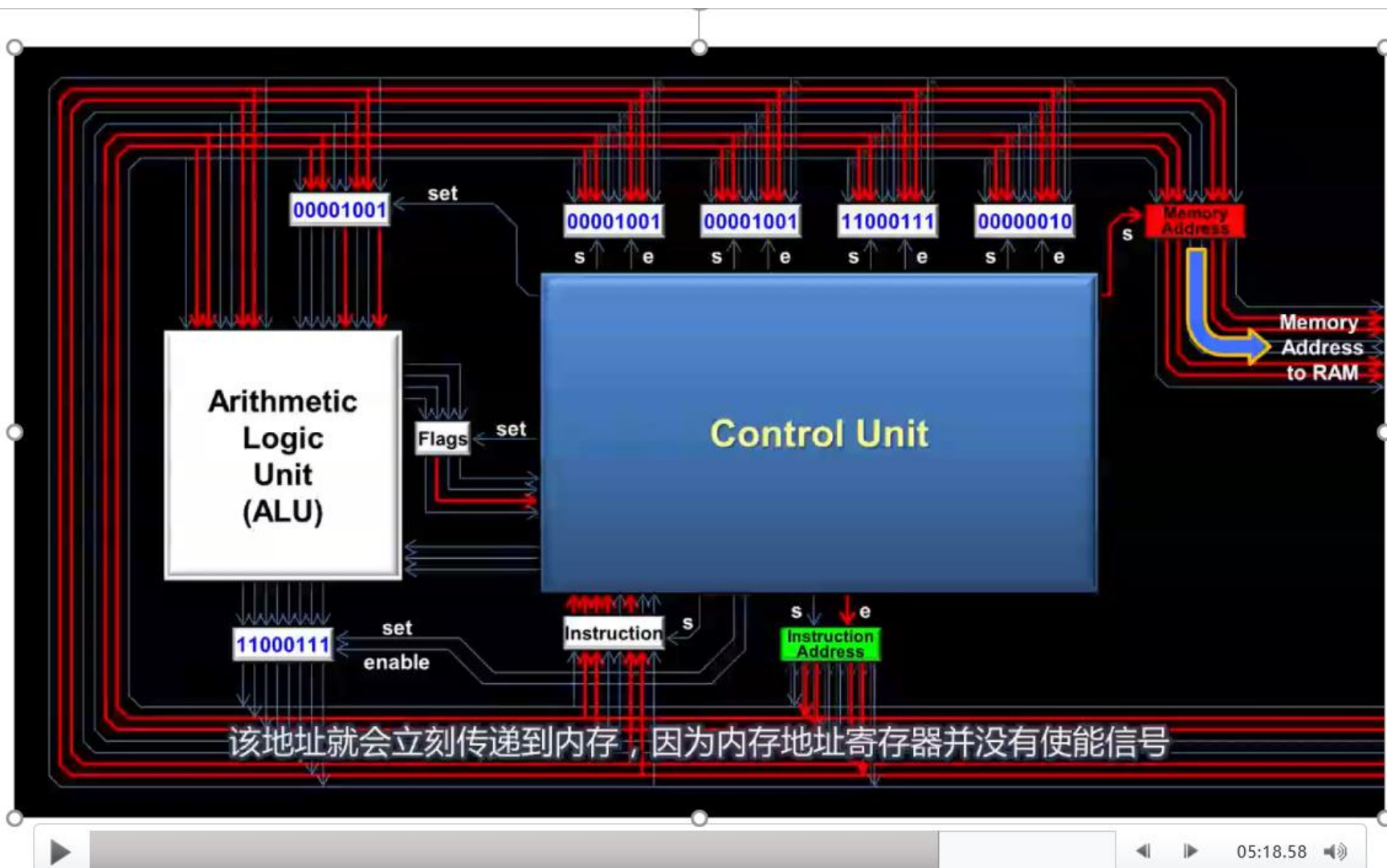


如果内存中的数据已经在 CPU Cache 里了，对一个内存地址的访问，4 个步骤：

- 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。

控制单元CU



算术逻辑运算单元
ALU

控制单元CU



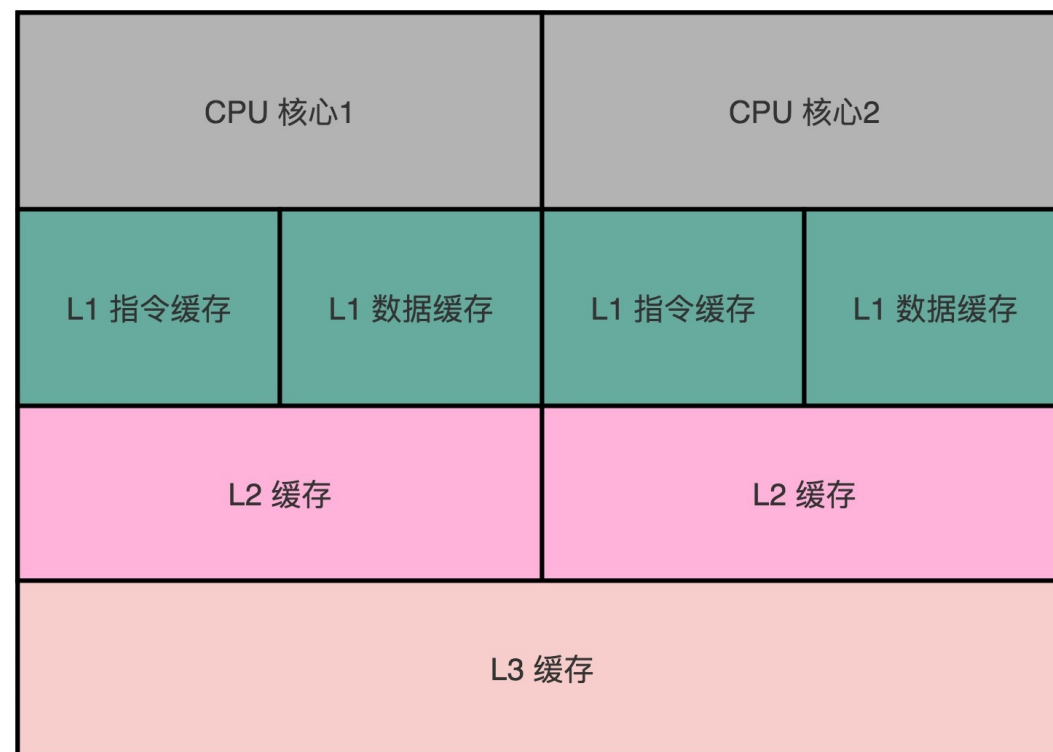
当算术逻辑单元向寄存器输出数据时

小结

CPU 和内存之间的性能差异，导致访问内存（DRAM 芯片）成为程序的性能瓶颈，CPU Cache 尽可能解决这两者之间的性能鸿沟

CPU，**直接映射 Cache**，定位一个内存访问地址在 Cache 中的位置。缓存放置策略还有全相连 Cache（Fully Associative Cache）、**组相连 Cache（Set Associative Cache）（现代CPU常用）**，等其他几种策略。

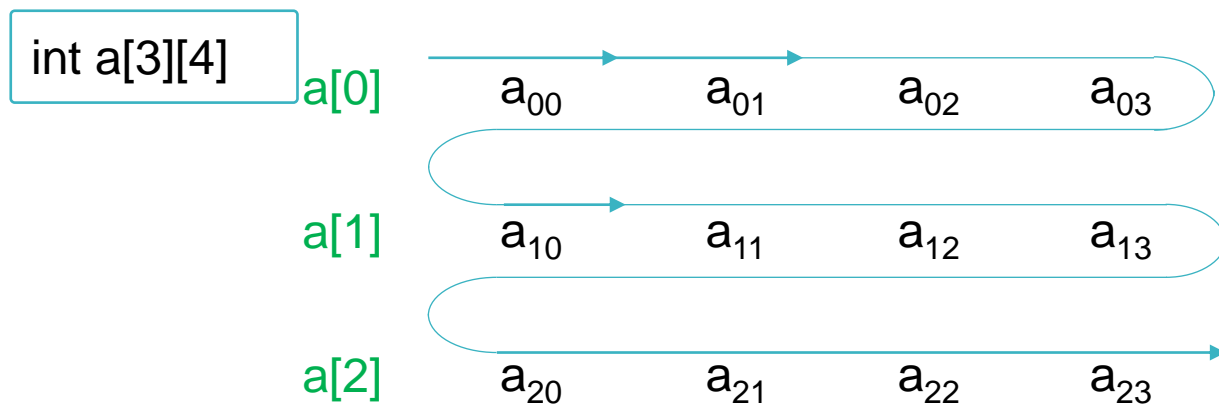
可以将很大的内存地址，映射到很小的 CPU Cache 地址。





对于二维数组的访问，按行遍历和按列遍历的访问性能是一样的吗？

C语言中，二维数组的元素在内存中是按行连续存放的，是线性的。



数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 $4 \text{ Byte} \times (3 \times 4) = 48 \text{ Byte}$ 。

2000	a[0][0]	第0行元素
2004	a[0][1]	
2008	a[0][2]	
2012	a[0][3]	
2016	a[1][0]	第1行元素
2020	a[1][1]	
2024	a[1][2]	
2028	a[1][3]	
2032	a[2][0]	第2行元素
2036	a[2][1]	
2040	a[2][2]	
2044	a[2][3]	

按行遍历，可以很好的利用**Cache line** 数据块，从高速缓存中读取数据

```


4 #include<string.h>
5 #include<stdlib.h>
6 #define N 10000
7 int main()
8 {
9     int i;
10    int j;
11    clock_t timestart, timeend;
12
13    int** arr = (int**)malloc(N * sizeof(int*));
14
15    for (i = 0; i < N; i++)
16    {
17        arr[i] = (int*)malloc(N * sizeof(int));
18    }
19
20    //arr = (int(*)[100])malloc(sizeof(int*) * 100);
21
22    timestart = clock();
23    // 循环1
24    for (i = 0; i < N; i++)
25        for(j=0;j<N;j++)
26            arr[i][j] = 10;
27    timeend = clock();
28    printf("time: %lfs\n", (double)(timeend - timestart) / CLOCKS_PER_SEC);

```

```

29
30    timestart = clock();
31    // 循环2
32    for (j = 0; j < N; j++)
33        for (i = 0; i < N; i++)
34            arr[i][j] = 10;
35    timeend = clock();
36    printf("time: %lfs\n", (double)(timeend - timestart) / CLOCKS_PER_SEC);
37
38    for (i = 0; i < N; i++)
39    {
40        free(arr[i]);
41    }
42    free(arr); //释放内存
43    arr = NULL;
44 }
45

```

 Microsoft Visual Studio 调试

```

time: 0.128000s
time: 0.928000s

```



从2007年开始华尔街开始用计算机取代人工进行交易处理，这时候开始交易的速度就取决于网络速度

主流网络公司提供的网络速度很不稳定，今天是16毫秒明天是17毫秒

当时最快的Verizon电信是14.65毫秒可以完成一次订单传输

华尔街的交易员斯皮维在仔细研究了芝加哥到新泽西的电缆路线后，在经过详细的计算后，他重新规划了一条管道铺设路线，比早前的电缆缩短了100英里。

2008年，斯皮维找到建筑商，说明了自己要铺设一条网络管道，并且除了不可避免的弯路之外任何一条弯路都不能存在，哪怕是任何天然的屏障都要想尽办法解决，于是他们凿开了阿勒格尼山脉...

他说服了Jim Barksdale为了他这条自己估值超过3亿美金的电缆工程筹钱，他们创立了Spread Networks、Northeastern ITS、Job 8等空壳公司来秘密进行工程

他们计算出一家华尔街银行利用这条线路，可以带来的现金与期货之间的简单套利能达到200亿美元，于是为了筹集资金他们找到200个愿意提前付款的高频交易员，支付了未来5年一共1400万美元的费用，这相当于当时电信线路的10倍价格，就这样他们集资到了28亿美元，在经历了种种困难后，这条总长825英里的线路终于在2010年的7月竣工，他们把信号传输时间缩短到了13毫秒



对于高频交易公司来说，光纤的2/3 光速还是太慢了。

Jump Trading（跳跃交易）公司在全球最大期货交易所芝加哥商品交易所数据中心对面，买了一块 12 万平方米的空地，没盖楼炒房，也不是为了风水，就架**微波通信基站**，为了把交易请求传到芝加哥商品交易所快0.07毫秒



2013 年Jump Trading 买下英国一处原来供北约使用的微波塔，只为把数据更快传到伦敦商品交易所

美国证监会在2010年的一份文件中，提出“高频交易”往往会同时满足以下特征：

1. 使用超高速的复杂计算机系统下单
2. 使用 co-location 和直连交易所的数据通道
3. 平均每次持仓时间极短
4. 大量发送和取消委托订单
5. 收盘时基本保持平仓（不持仓过夜）

高频交易复杂的性质决定了它只能由专业的交易员发起。正如美国作家Michael Lewis 在他的非虚构畅销书《高频交易员》（Flash Boys）所说，**数据传输的速度决定着交易的价格**。交易员们不顾一切，拼的就是速度。而在美国，这个竞争已经到了微秒的级别。

套利背后的基本设想是某些金融工具的价格根本上是相互联系的。

所以如果发现本应价值相同的两种金融工具当前的价格存在差异，那么应该买入便宜的，或者卖空较贵的。当便宜的价格上涨或者贵的价格下跌，高频交易商就可盈利。

由于很难预测会发生二者中的哪种情况（或者由于其他因素两种工具的价格均发生了变化），最佳的策略是同时买入和卖出它们。这样无论发生何种情况，高频交易商都能盈利

高频交易公司并不是同时只买卖一个公司的股票，一般是同时买卖上千支股票，以及不同交易策略所需要交易的股票指数、期货、外汇等。

交易具体怎么进行的？

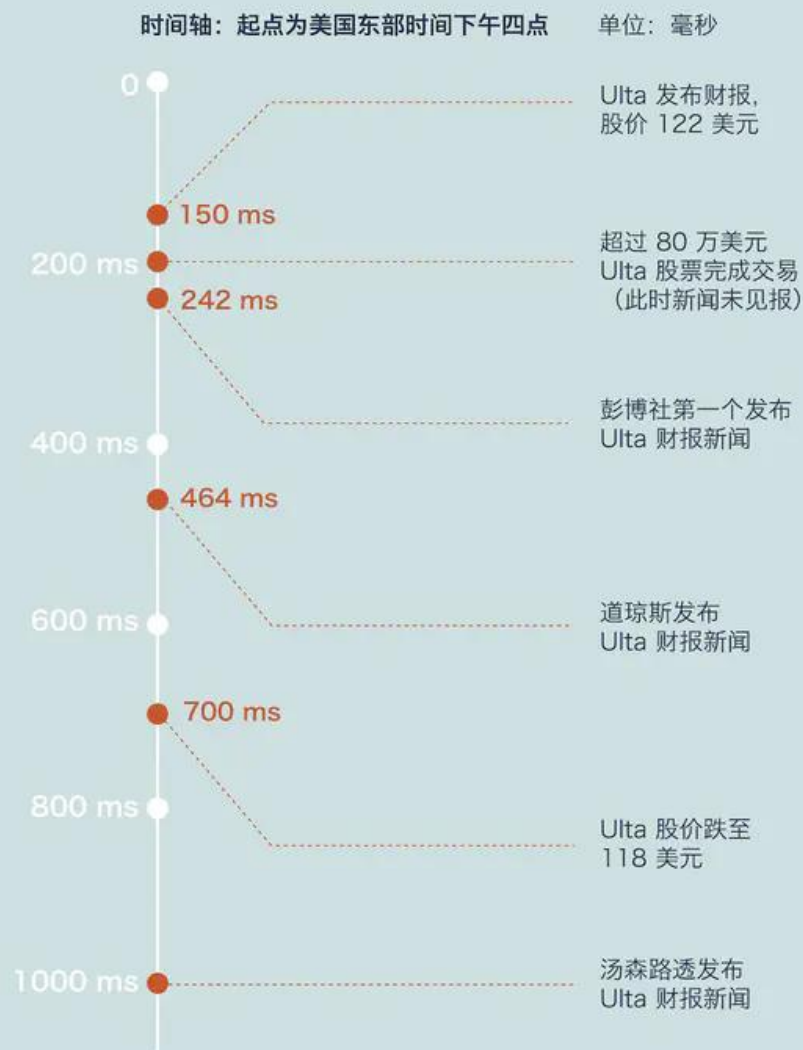
《华尔街日报》曾披露过这样一个例子，2014 年 12 月 5 日美国东部时间下午 4 点，化妆品零售商 Ulta 发布财报，当时股价为 122 美元。接下来事情的发展成了这样：

4 点 0 分 1 秒，汤森路透发布 Ulta 财报新闻。
我们可以很放心的说，等到这时候，还没有任何人类交易员读完财报标题。尽管只过了 0.85 秒

高频交易公司现在要等彭博社发布信息才能开始交易，会比之前晚 0.192 秒



与时间赛跑的高频交易



资料来源: 华尔街日报

一笔交易快几毫秒，这是交易过程中的数字游戏，和公司价值并没有任何关系

今天做高频交易、让计算机直接管理基金的公司越来越多，每家都很快。从某种程度上说，市场更公平了，但参与者的回报也大不如前



2015年，等到李奥自立门户开始交易时，他再次认识到了硬件和系统对交易速度的致命影响。

那时，李奥的程序已经稳定运行了半个月，结果有一天突然开始亏钱了。他一检查，才发现自己前一天有个程序没关，占用了CPU。李奥平时开两个程序，那天开了三个，没想到两个程序开始互相竞争CPU。

“我X，我第一次意识到（硬件）居然会有这么重要。”他说。

这次教训后，李奥花了一两个月的时间去研究CPU和系统。他慢慢领悟到，高频交易拼速度，写代码可能占50%，但外围因素很关键：机房，机器，对各种系统要求都很高。

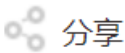
“有时候你自己优化C++，搞半天优化一个毫秒，很开心。买个好的CPU，可能就优化五毫秒。（这些事）别人不会出来跟你说的，你得自己去琢磨。”他说。

在上海某期货公司为李奥提供技术支持的L对李奥的自学能力赞叹不已。“通常在高频领域，做策略的不懂技术，做软件的不懂硬件，他一个人就搞定了。”L说，“有次调整服务器，我们期货公司的运维工程师都没搞定（我找了两个人来看），结果他自己搞定了。”

对于李奥爱钻研的精神，他的太太R深有体会。她告诉我，李奥平时在家，哪怕是跟她一起看电视时，基本都是电脑不离手地在写代码。

AI开始抢饭碗了 贝莱德裁员改用机器人选股

2017-04-03 08:01:28来源:智能电子集成



分享

[摘要] 投资管理公司贝莱德 (BlackRock) 已开始改用人工智能(AI)来做程序化选股, 预计裁掉数十名基金经理人和分析师

全球最大的投资管理公司贝莱德 (BlackRock) 已开始改用人工智能(AI)来做程序化选股, 该公司表示, 机器人选股将建立更自动化的流程, 取代现有人力, 预计裁掉数十名基金经理人和分析师, 节省的人力支出可达数千万美元。

人工智能对各行业的影响已经在逐渐浮现。在投资领域程序化选股早已行之有年, 近年来市场结构转变, 指数型基金崛起, 都带给主动式基金压力, 不得不降低收费吸引客户, 这也直接促使投资管理公司开始考量庞大的人力投资问题。

贝莱德创办人兼首席执行官 Laurence D. Fink表示, 经过多年讨论, 他们决定投入大量的机器以替代人力。他强调, 未来贝莱德将会持续押注在机器人理财顾问、大数据 甚至是人工智能等领域。

上个月底贝莱德提出了一个计划, 将利用更多演算法及数字模型来选择股票, 并积极管理共同基金。这项改革采用更量化的策略, 将传统主动管理的 2 千亿美元的资产中, 挪移出 80 亿美元转为更便宜的产品, 当然也会使约 30 名左右, 包括经理人及分析师等员工被解雇。

选股业务的资金将会转移到新的量化部门, 预计规模有 740 亿美元, 透过自动化流程为投资人提供 9 个风险更低的基金。该公司也表示, 此次裁员也有助于节省约 2,500 万美元费用。近年来, 贝莱德的主动式基金表现并不好, 甚至低于业界水准, 自 2012 年开始 就积极聘用顶尖人才, 试图重振业务。不过在去年, 旗下的 4 家对冲基金表现也是史上最差。

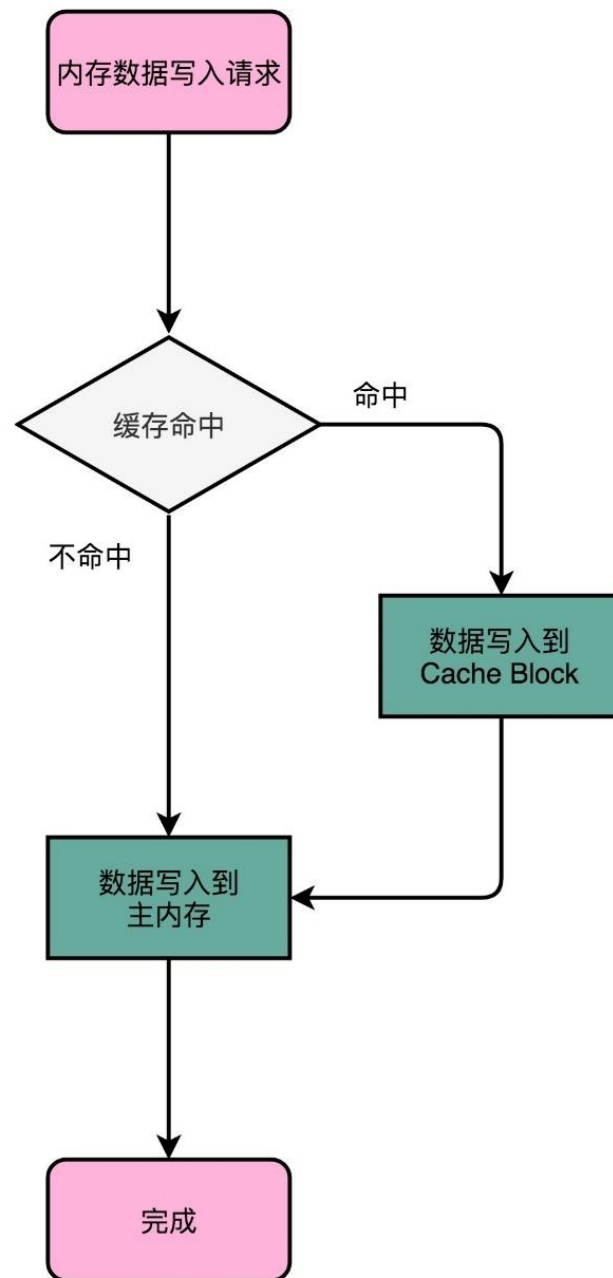
2016年, 贝莱德聘请了资深基金经理人Mark Wiseman在最艰难的时期接手。尽管未来必须裁员, 但贝莱德表示未来也会需要更多具备数据分析能力的资深人才。

其实， 350 微秒的差异，足够高频交易公司进行无风险套利。

350 微秒，内存访问一次100 纳秒，大约进行 3500 次。引入 CPU Cache 之后，可以进行的数据访问次数，可以在8万多次以上，提升了数十倍。

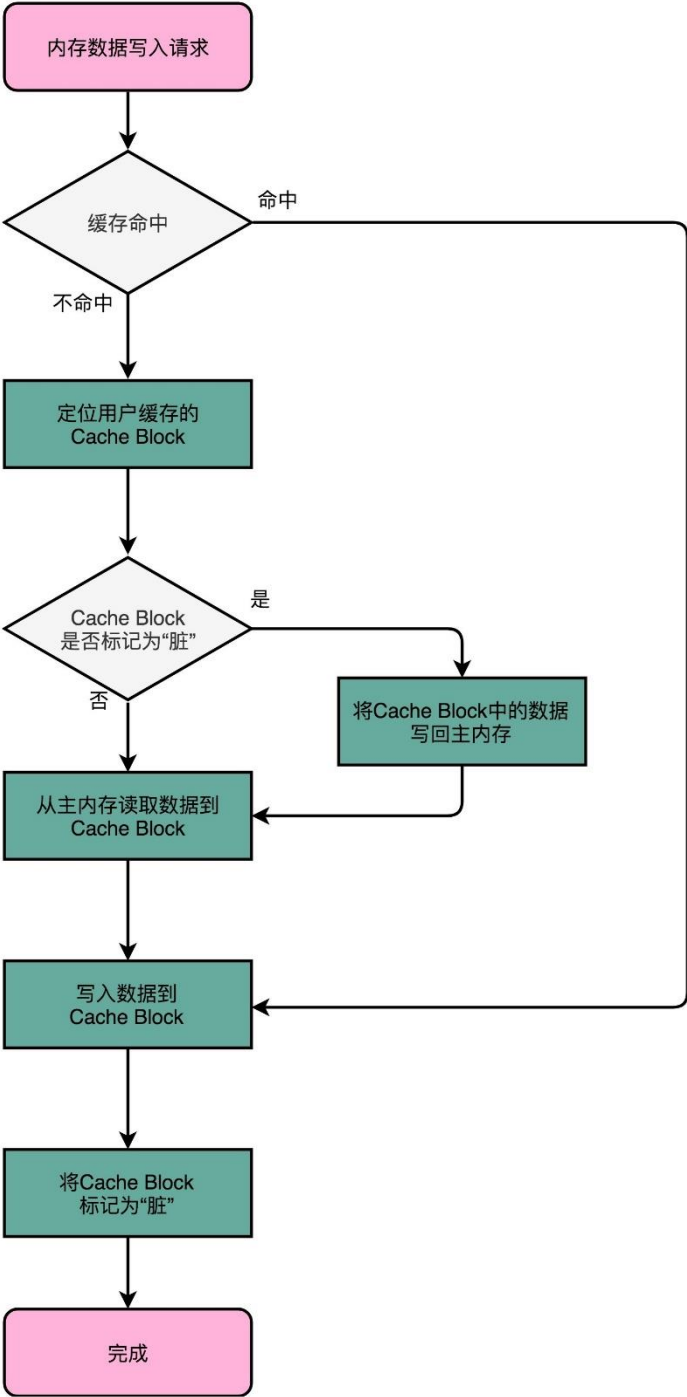
CPU 不仅要读数据，还需要写数据

写直达 (Write-Through)。每一次数据都要写入到主内存里面。写入前，先判断数据是否已经在 Cache 里面了。如果数据已经在 Cache 里面了，先把数据更新到 Cache 里面，再写入到主内存里面；如果数据不在 Cache 里，就只更新主内存。



写回 (Write-Back)。只写到 CPU Cache 里。只有当 CPU Cache 里面的数据要被“替换”的时候，才把数据写入到主内存里面去。

在写回这个策略里，大量的操作都能够命中缓存，大部分时间里，都不需要读写主内存，性能会比写直达的效果好很多



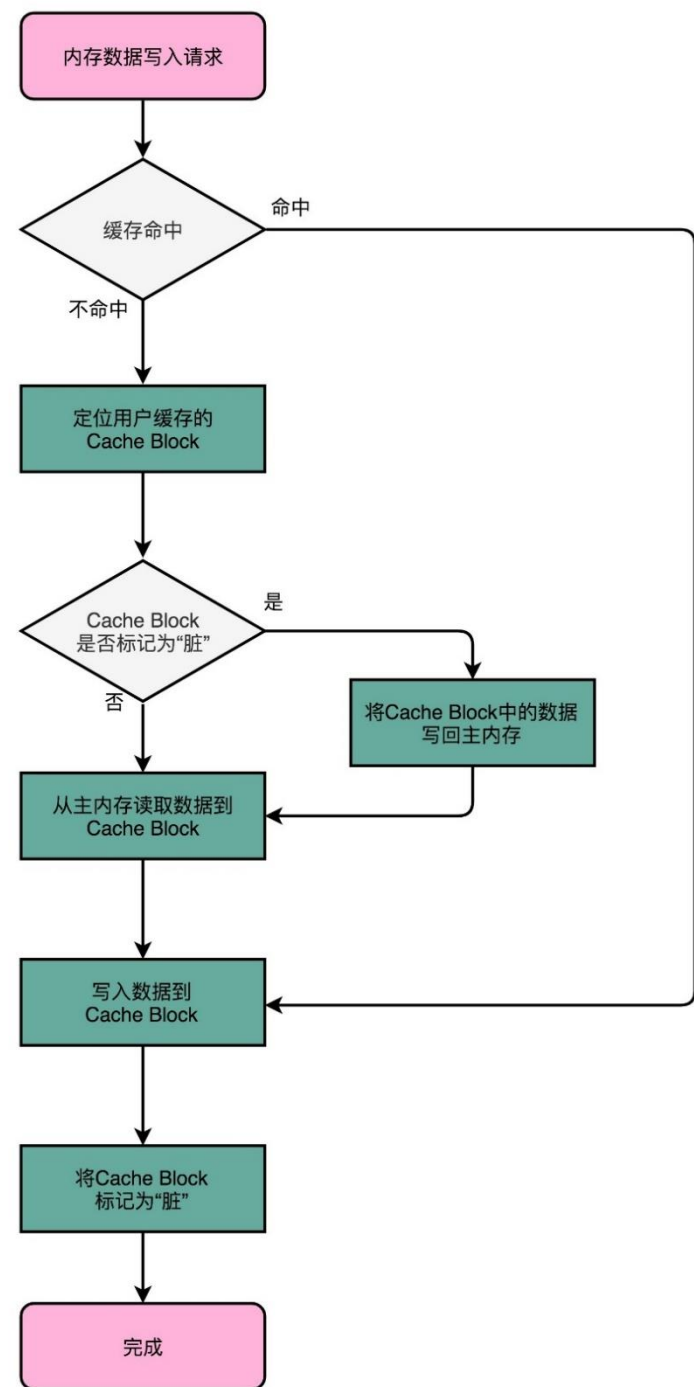
如果要写入的数据块，就在 CPU Cache 里面，那么就只更新 CPU Cache 里面的数据。同时，标记这个 **Block 是脏 (Dirty)** 的。即，此时这个 Block 数据，和内存是不一致的。

如果要写入的数据所对应的 Cache Block，已经存放了别的内存的数据。那么就要看一看，Block 里面的数据有没有被标记成脏的。

如果是脏的话，要先把这个 Cache Block 里面的数据，写回到主内存里面。然后，再把当前要写入的数据，写入到 Cache 里，同时把 Cache Block 标记成脏的。

如果 Block 里面的数据没有被标记成脏的，那么我们直接把数据写入到 Cache 里面，然后再把 Cache Block 标记成脏的就好了。

在用了写回这个策略之后，**加载内存数据到 Cache 里面的时候，也要多一步同步脏 Cache 的动作**：如果加载内存里面的数据到 Cache 的时候，发现 Cache Block 里面有脏标记，我们也要先把 Cache Block 里的数据写回到主内存，才能加载数据覆盖掉 Cache。



存储与I/O系统

01

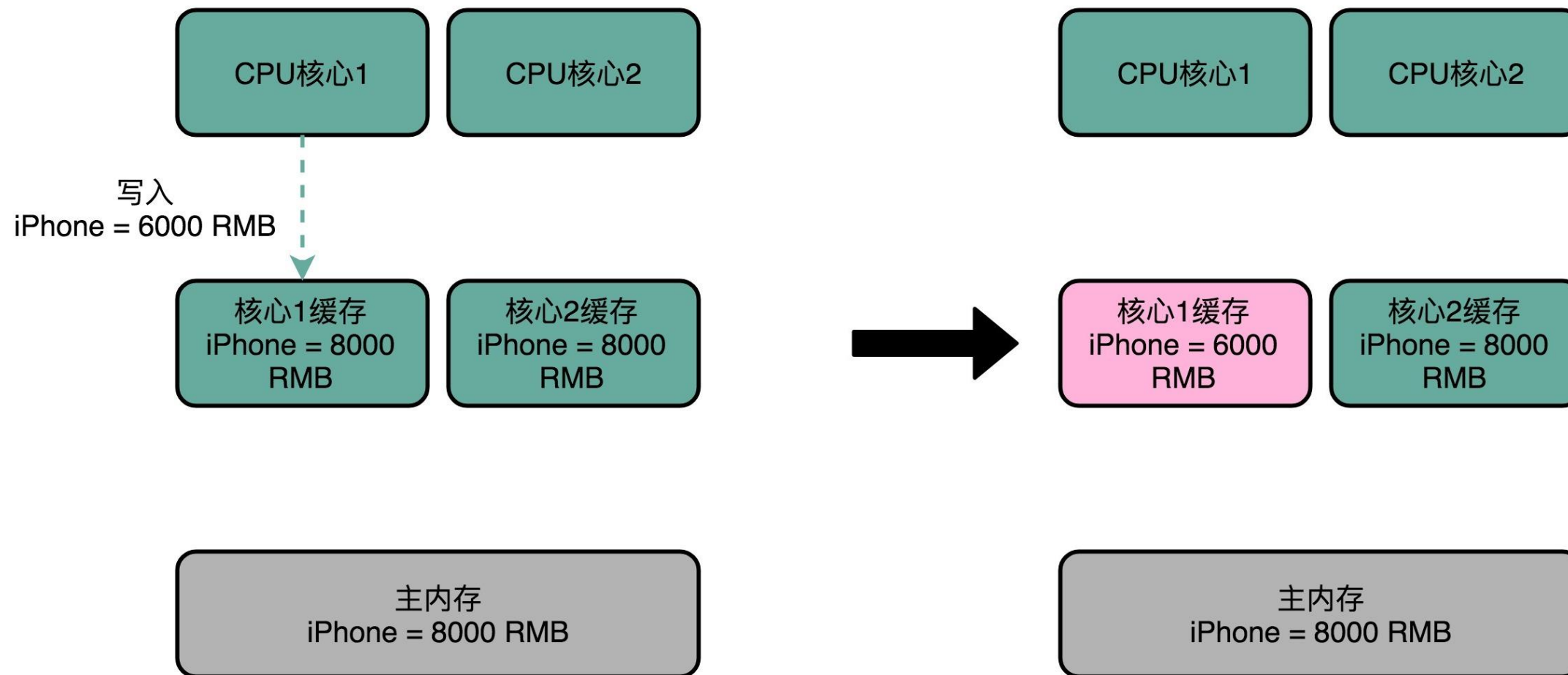
存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

比尔·盖茨在上世纪 80 年代 —— “640K ought to be enough for anyone”

缓存一致性





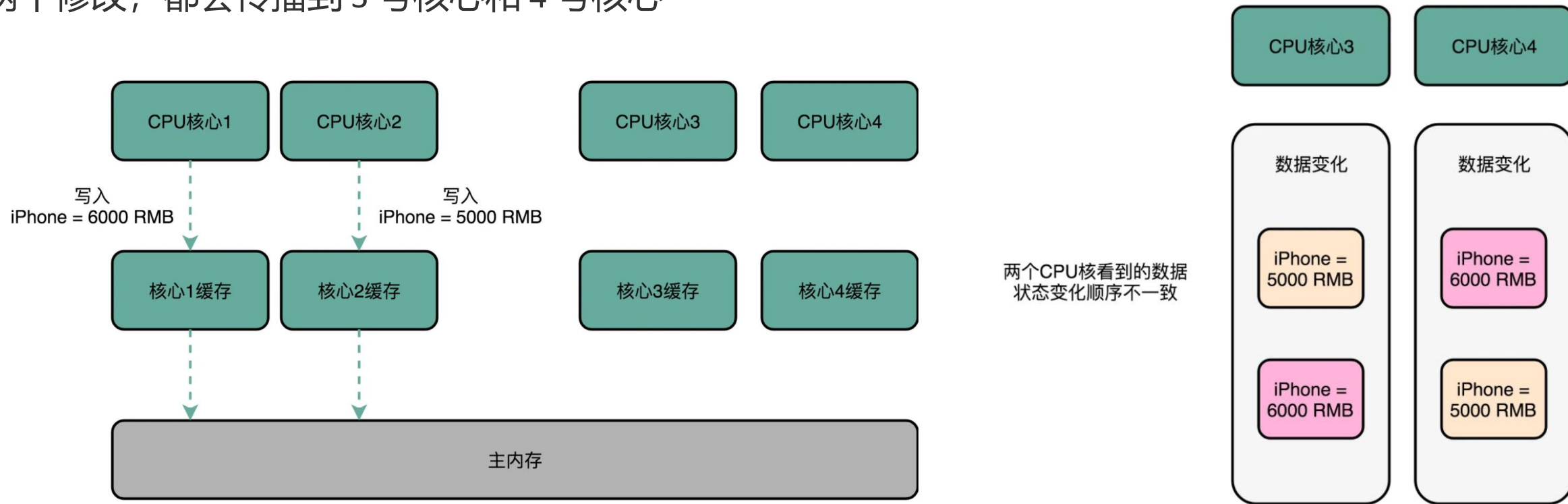
缓存一致性

为了解决这个缓存不一致的问题，需要有一种机制，来同步两个不同核心里面的缓存数据。

第一点叫**写传播 (Write Propagation)**。在一个 CPU 核心里，Cache 数据更新，必须能够传播到其他的对应节点的 Cache Line 里。

第二点叫**事务的串行化 (Transaction Serialization)**，在一个 CPU 核心里面的读取和写入，在其他的节点看起来，顺序是一样的。

4 核 CPU。1 号核心，先把 iPhone 的价格改成了 6000 块。
差不多在同一个时间，2 号核心把 iPhone 的价格改成了 5000 块。
两个修改，都会传播到 3 号核心和 4 号核心



需要的是，从 1 号到 4 号核心，都能看到相同顺序的数据变化。称之为实现了**事务的串行化**

缓存一致性

事务的串行化，不仅仅是缓存一致性中所必须的。平时所用到的系统当中，最需要保障事务串行化的就是数据库。

多个不同的连接去访问数据库的时候，必须保障事务的串行化，做不到事务的串行化的数据库，无法作为可靠的商业数据库来使用。

在 CPU Cache 里做到事务串行化：

第一点，一个 CPU 核心对于数据的操作，同步通信给到其他 CPU 核心。

第二点，如果两个 CPU 核心里有同一个数据的 Cache，那么对于这个 Cache 数据的更新，需要有一个“锁”的概念。只有拿到了对应 Cache Block 的“锁”之后，才能进行对应的数据更新。



MESI 协议，Pentium时代，被引入到 Intel CPU 中

对 Cache Line 的四个不同的标记：

M：代表已修改（Modified）——“脏”的 Cache Block，内容已经更新，但还没有写回到主内存里

E：代表独占（Exclusive） **无论独占状态还是共享状态，缓存里面的数据都是“干净”的，Cache Block 里面的数据和主内存里面的数据是一致的**
S：代表共享（Shared）

I：代表已失效（Invalidated）——数据已经失效，不可以相信这个 Cache Block 里面的数据

独占状态下，该Cache Line 只加载进入了当前 CPU 核的 Cache 里。其他的 CPU 核，并没有加载对应的数据到自己的 Cache 里。这个时候，如果要向独占的 Cache Block 写入数据，可以自由地写入数据，而不需要告知其他 CPU 核。

在独占状态下的数据，如果收到了一个来自总线的读取缓存请求，它就会变成共享状态。因为这个时候另外一个 CPU 核心，要把该 Cache Block的数据，从内存加载到了它的 Cache 里。

在共享状态下，同样的数据在多个 CPU 核心的 Cache 里都有。所以，当更新 Cache 数据的时候，不能直接修改，而是要向所有的其他 CPU 核心广播一个请求，要求先把其他 CPU 核心里面的 Cache，都变成无效的状态，然后再更新当前 Cache 里面的数据。这个广播操作，一般叫作 RFO（Request For Ownership），也就是获取当前对应 Cache Block 数据的所有权

在多线程里面用到的读写锁。在共享状态下，大家都可以并行去读对应的数据。但是如果要写，就需要通过一个锁，获取当前写入位置的所有权。



小结

缓存一致性（写传播、事务串行化）  MESI 协议

存储与I/O系统

01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理
- Cache, Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页

02

比尔·盖茨在上世纪 80 年代 —— “640K ought to be enough for anyone”

软件系统——操作系统、编译系统

程序编译过程

可以手动控制编译流程，从而留下中间文件方便研究

① `gcc HelloWorld.c -E -o HelloWorld.i`

预处理：加入头文件，替换宏

② `gcc HelloWorld.c -S -c HelloWorld.s`

编译：把预处理完的文件进行词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件

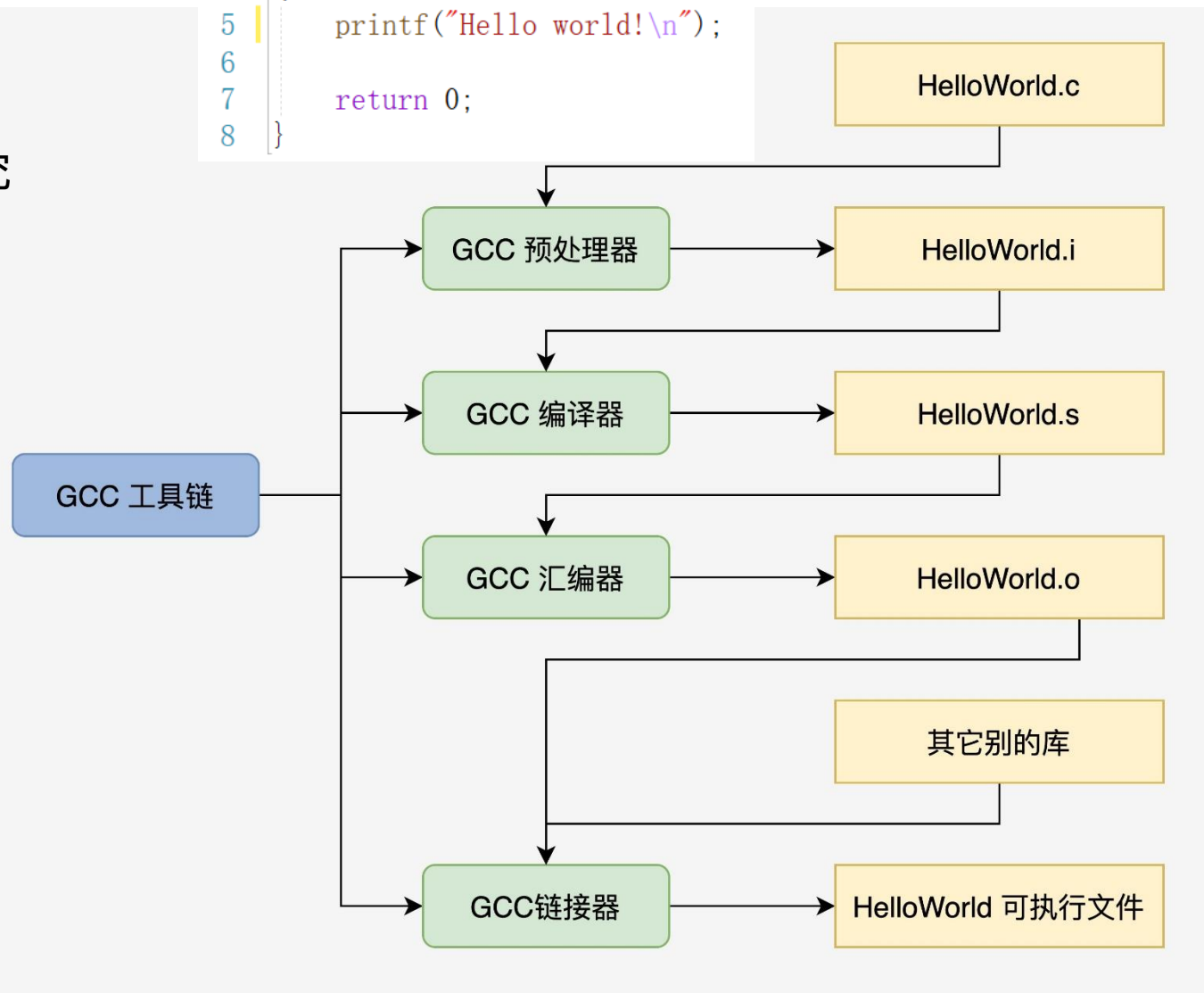
③ `gcc HelloWorld.c -c HelloWorld.o`

汇编：将汇编代码转换成可以执行的机器指令。大部分汇编语句对应一条机器指令，有的汇编语句对应多条机器指令。

④ `gcc HelloWorld.c -o HelloWorld`

链接：目标文件已经是二进制文件，与可执行文件的组织形式类似，只是有些函数和全局变量的地址还未找到，程序不能执行。链接的作用就是找到这些目标地址，将所有的目标文件组织成一个可以执行的二进制文件。

```
2 #include <stdio.h>
3 int main()
4 {
5     printf("Hello world!\n");
6
7     return 0;
8 }
```



第一，可执行程序加载后占用的内存空间应该是连续的
一条条指令连续地存储在一起

第二，需要同时加载很多个程序，并且不会让程序自己选择在内存中加载的位置



**要满足这两个基本的要求，可以在内存里面，找到一段连续的物理内存空间分配给装载的程序
把这段连续的物理内存空间地址，和程序指令的内存地址做一个映射。**