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

回顾要点（II）

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）

回顾要点（III）

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. 虚拟内存地址、物理内存地址、内存分段、内存分页、内存保护
5. 输入输出设备：接口、设备、硬盘（指标、结构、平均延迟时间、平均寻道时间）

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

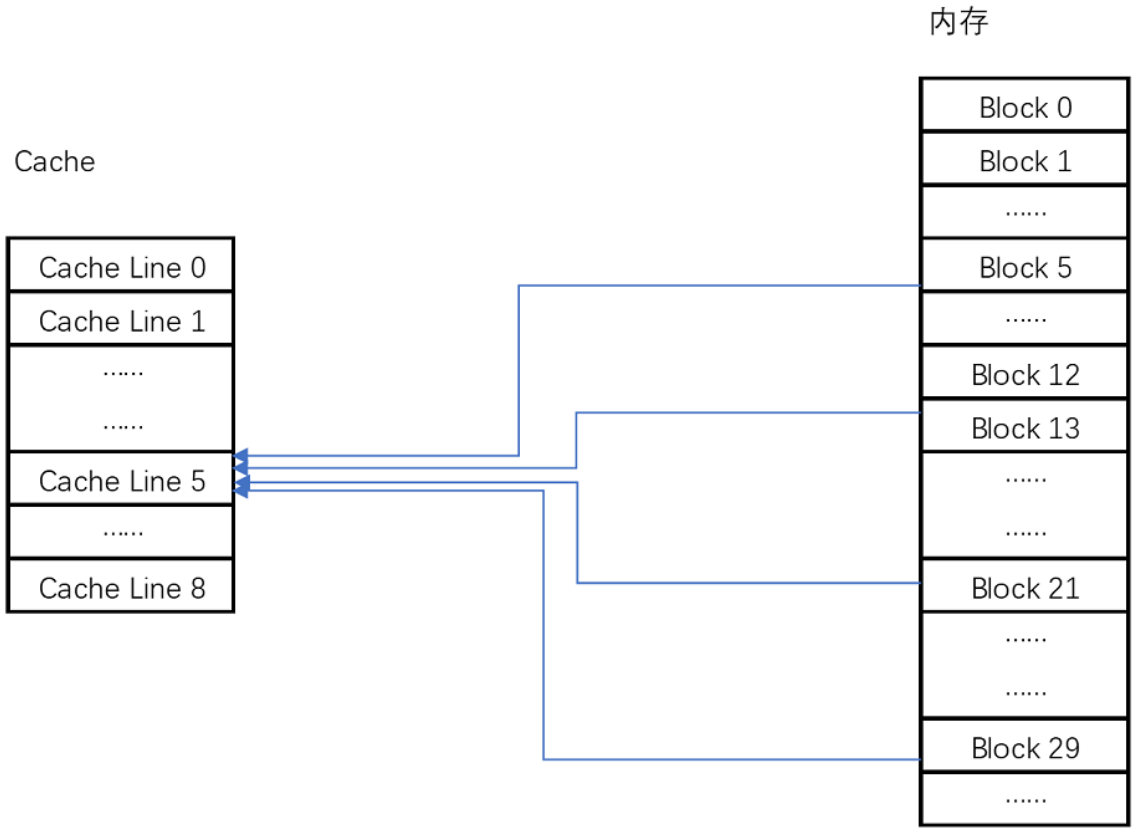
- 理解存储系统层次结构
- 局部性原理
- Cache，Cache line
- 缓存一致性、MESI协议
- 虚拟内存、内存分段、分页
- 内存保护

02

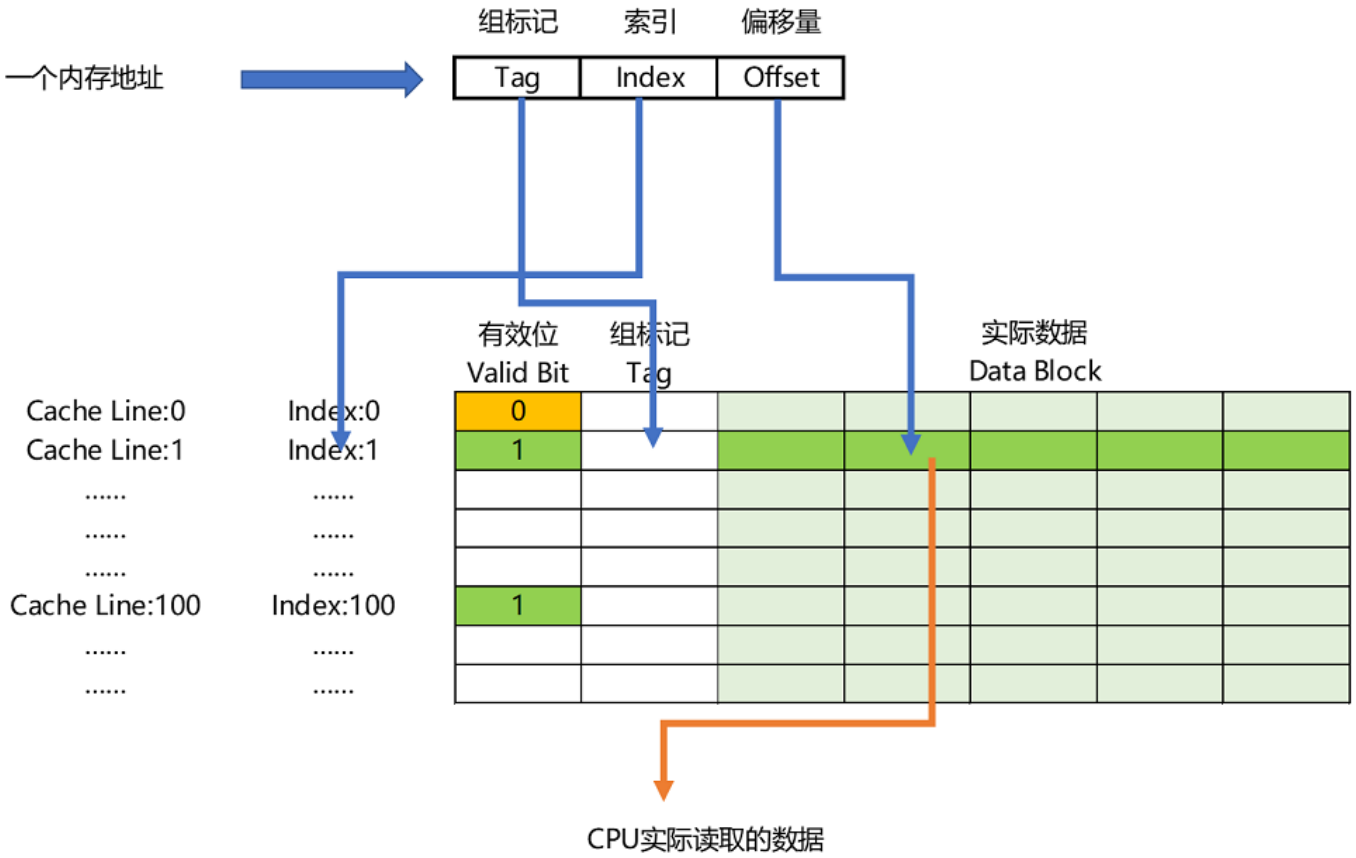
I/O

- 接口和设备
- 硬盘

CPU 如何知道要访问的内存数据，存储在 Cache 的哪个位置呢？



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



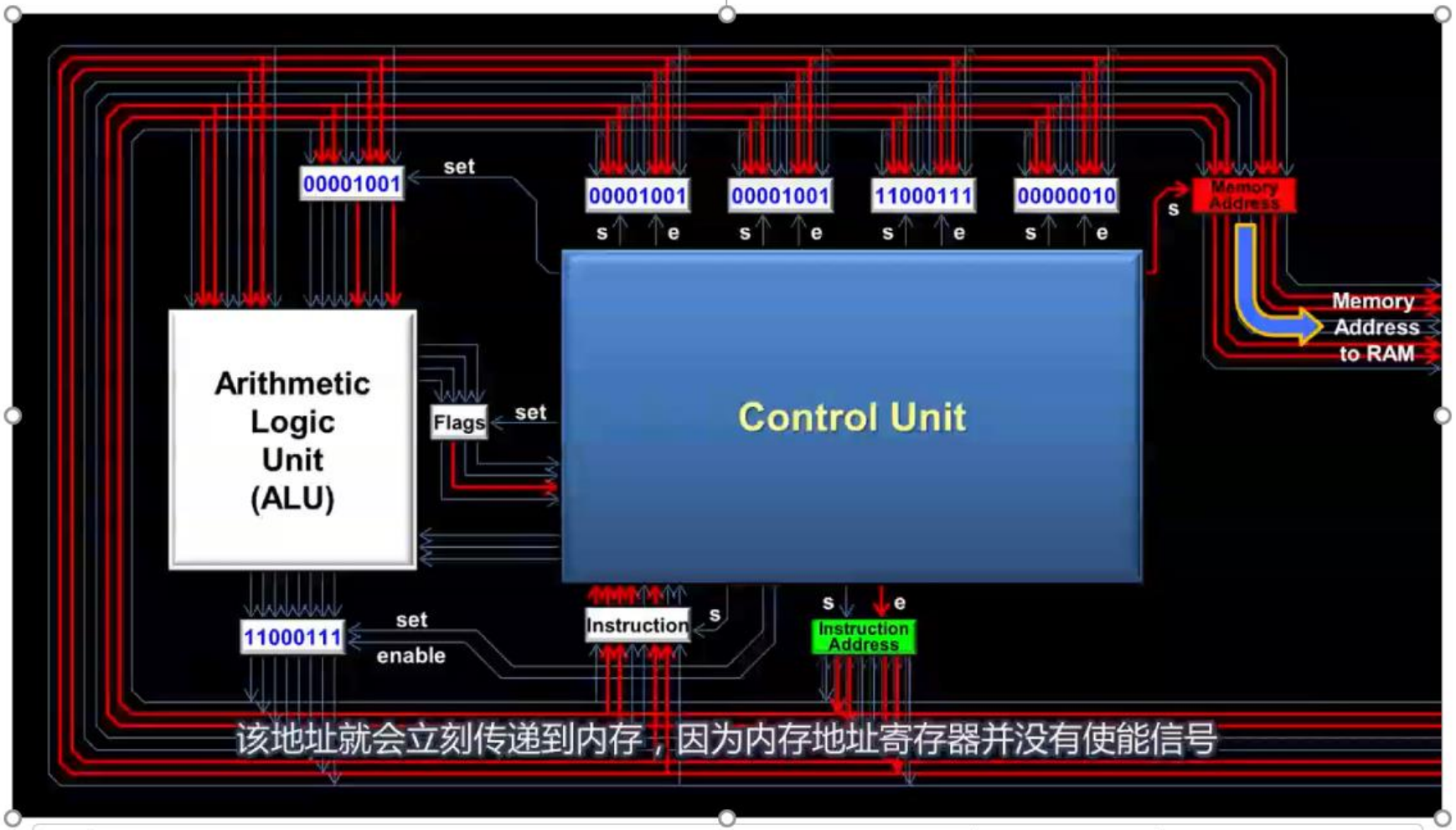
如果内存中的数据已经在 CPU Cache 里了，对一个内存地址的访问，4 个步骤：

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。

算术逻辑运算单元
ALU

控制单元CU

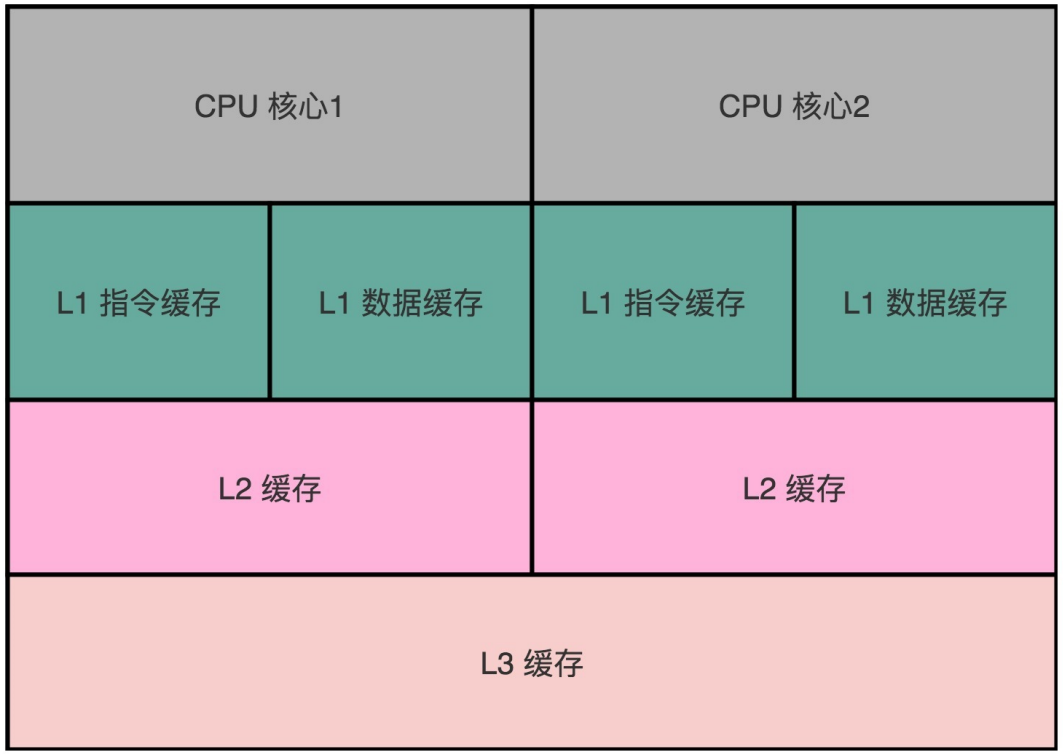


小结

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。

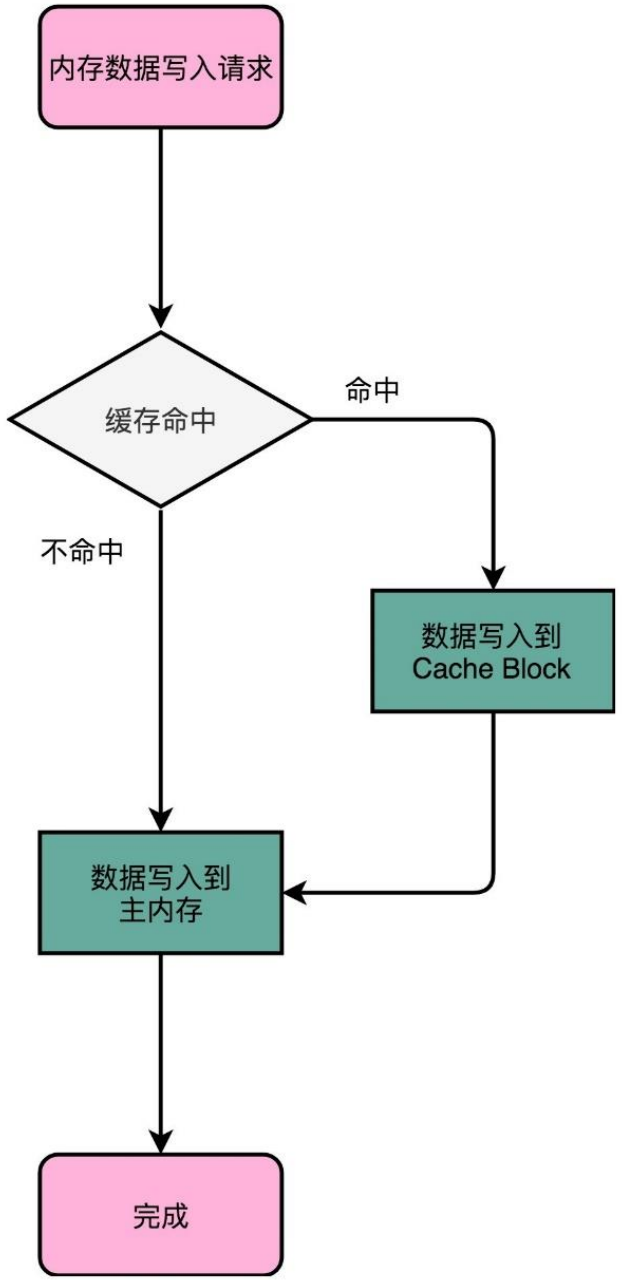
可以将很大的内存地址，映射到很小的 CPU Cache 地址。



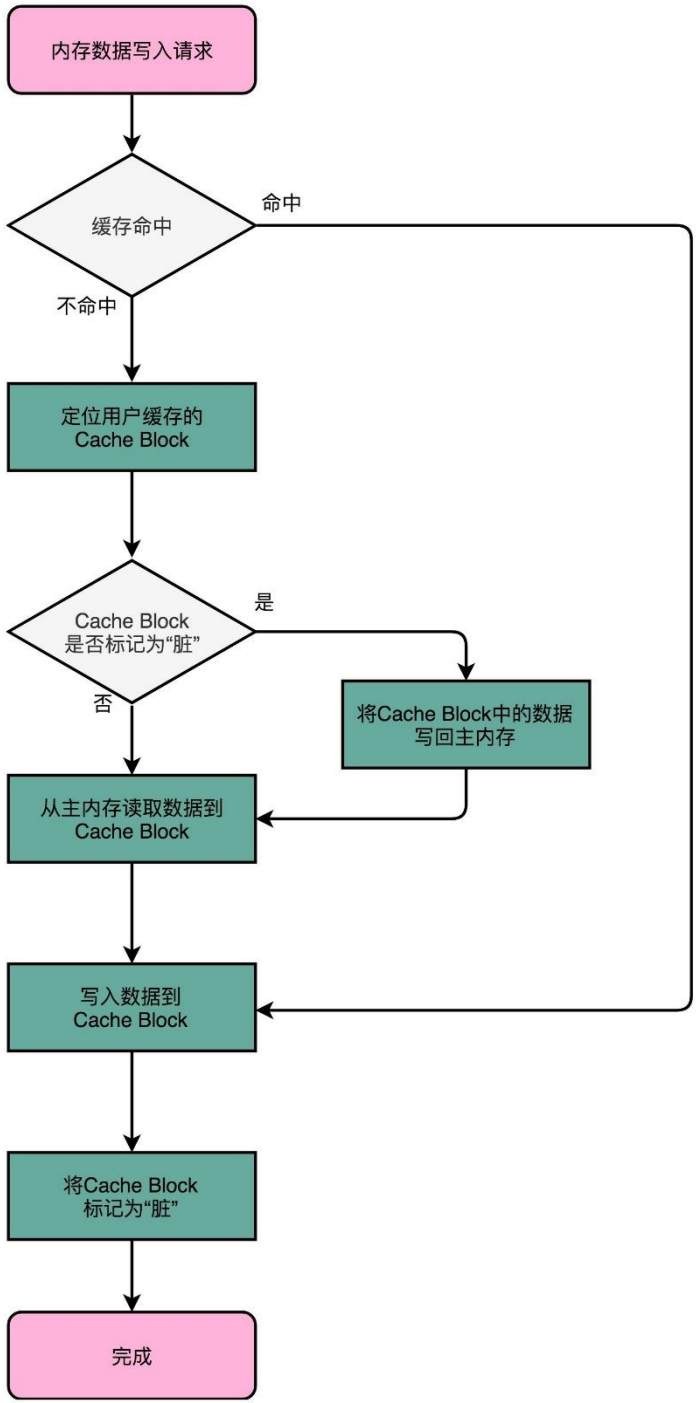
对于二维数组的访问，按行遍历和按列遍历的访问性能是一样的吗?

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

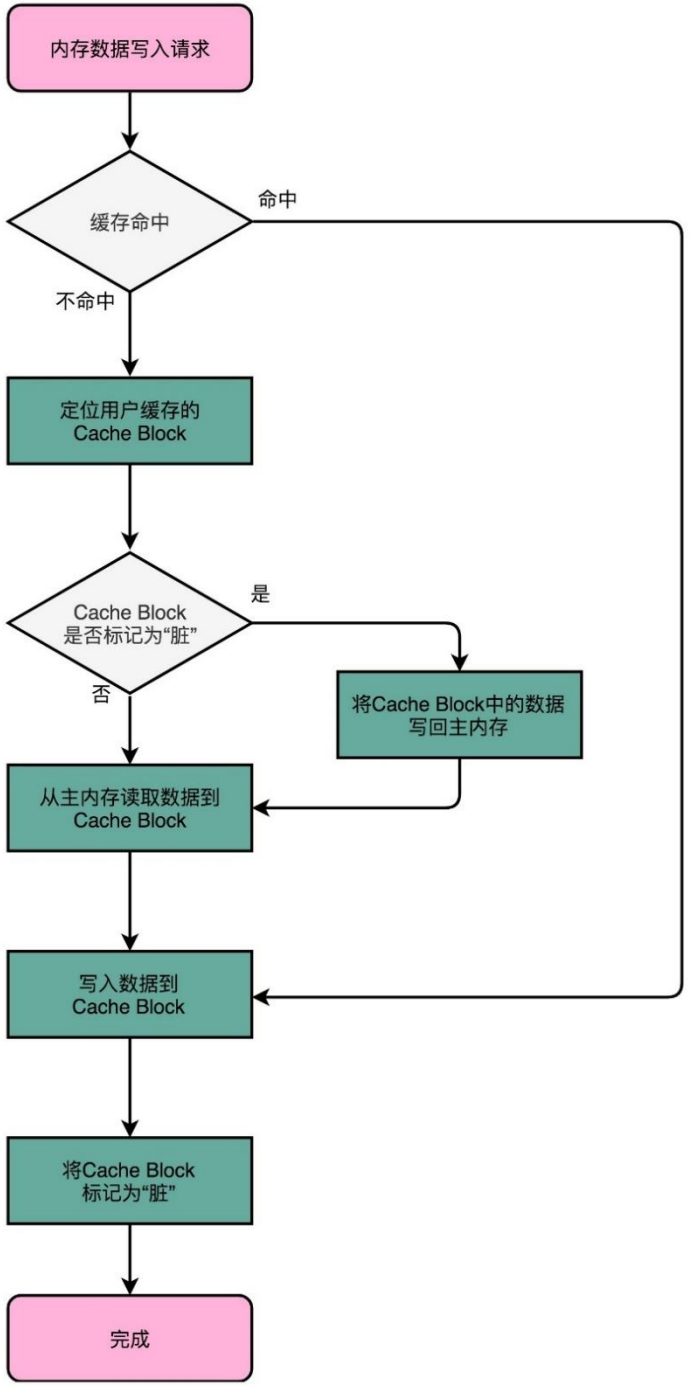
如果要写入的数据块，就在 CPU Cache 里面，那么就只更新 CPU Cache 里面的数据。同时，标记 这个 **Block 是脏（Dirty）**的。即，此时这个 Block 数据，和内存是不一致的。

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。

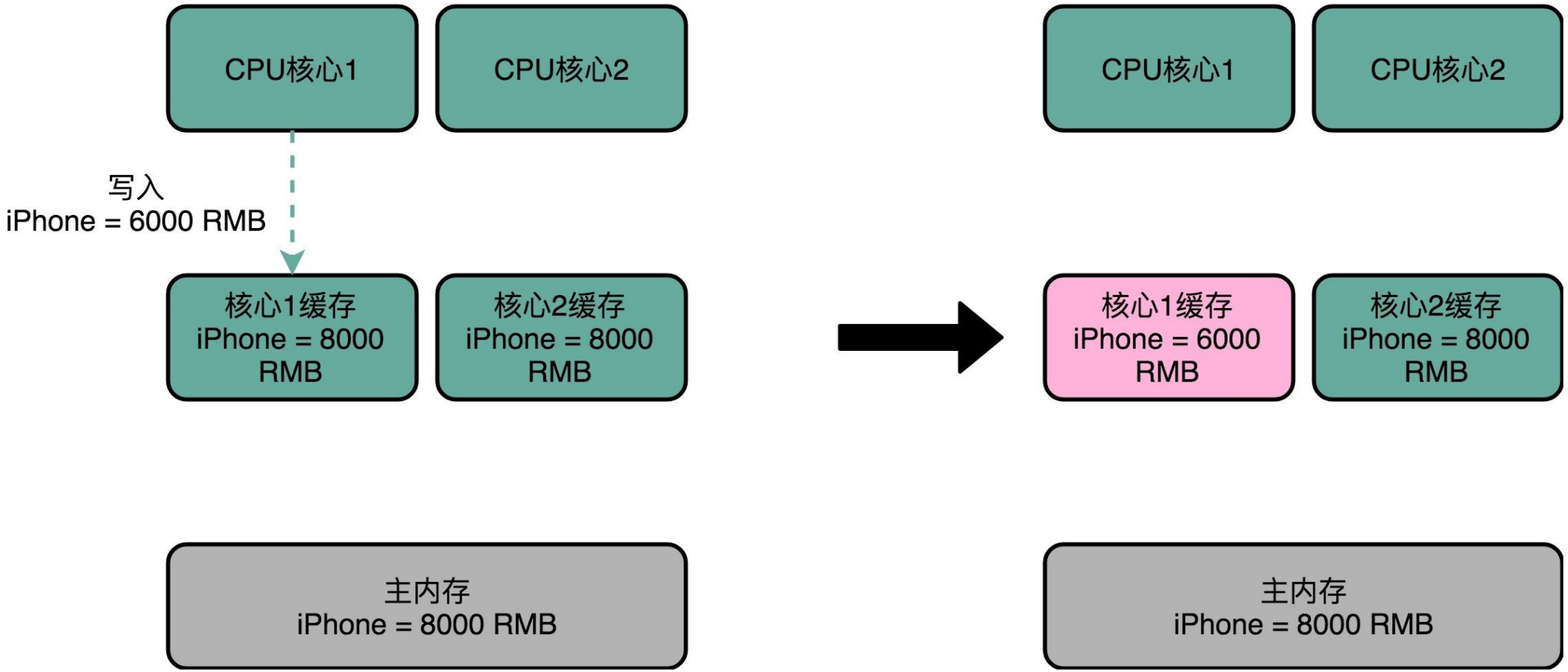
如果是脏的话，要先把这个 Cache Block 里面的数据，写回到主内存里面。然后，再把当前要写入的数据，写入到 Cache 里，同时把 Cache Block 标记成脏的。

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。



缓存一致性


CPU核心1
CPU核心2
写入
iPhone = 6000 RMB
核心1缓存
iPhone = 8000
RMB
核心2缓存
iPhone = 8000
RMB
主内存
iPhone = 8000 RMB
CPU核心1
CPU核心2
核心1缓存
iPhone = 6000
RMB
核心2缓存
iPhone = 8000
RMB
主内存
iPhone = 8000 RMB

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。

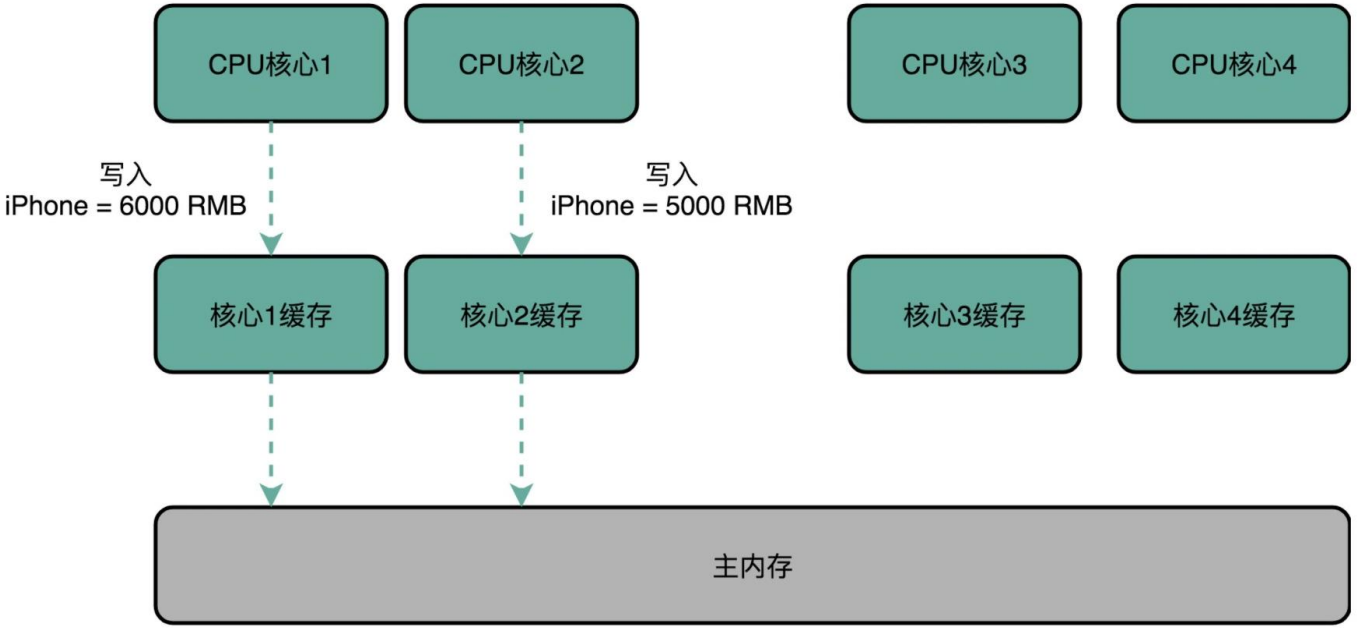
第一点叫**写传播（Write Propagation）**。在一个 CPU 核心里， Cache 数据更新，必须能够传播到其他的对应节点的 Cache Line 里。

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。

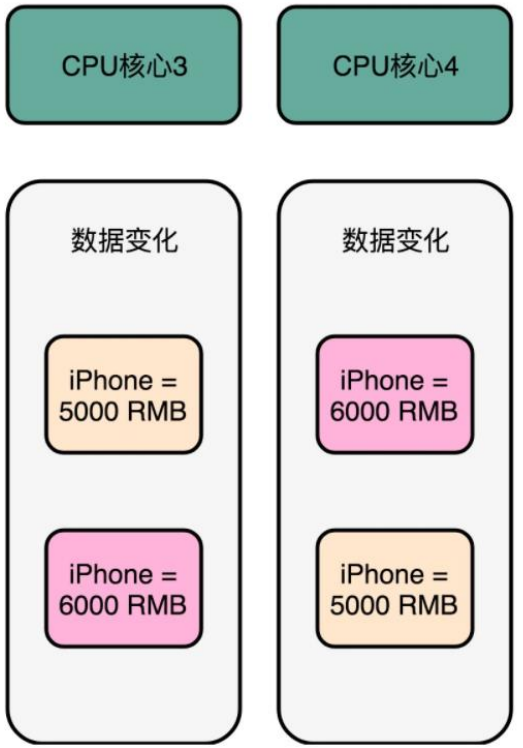
4 核 CPU。1 号核心，先把 iPhone 的价格改成了 6000 块。

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



两个CPU核看到的数据状态变化顺序不一致



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）
S：代表共享（Shared）

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

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。

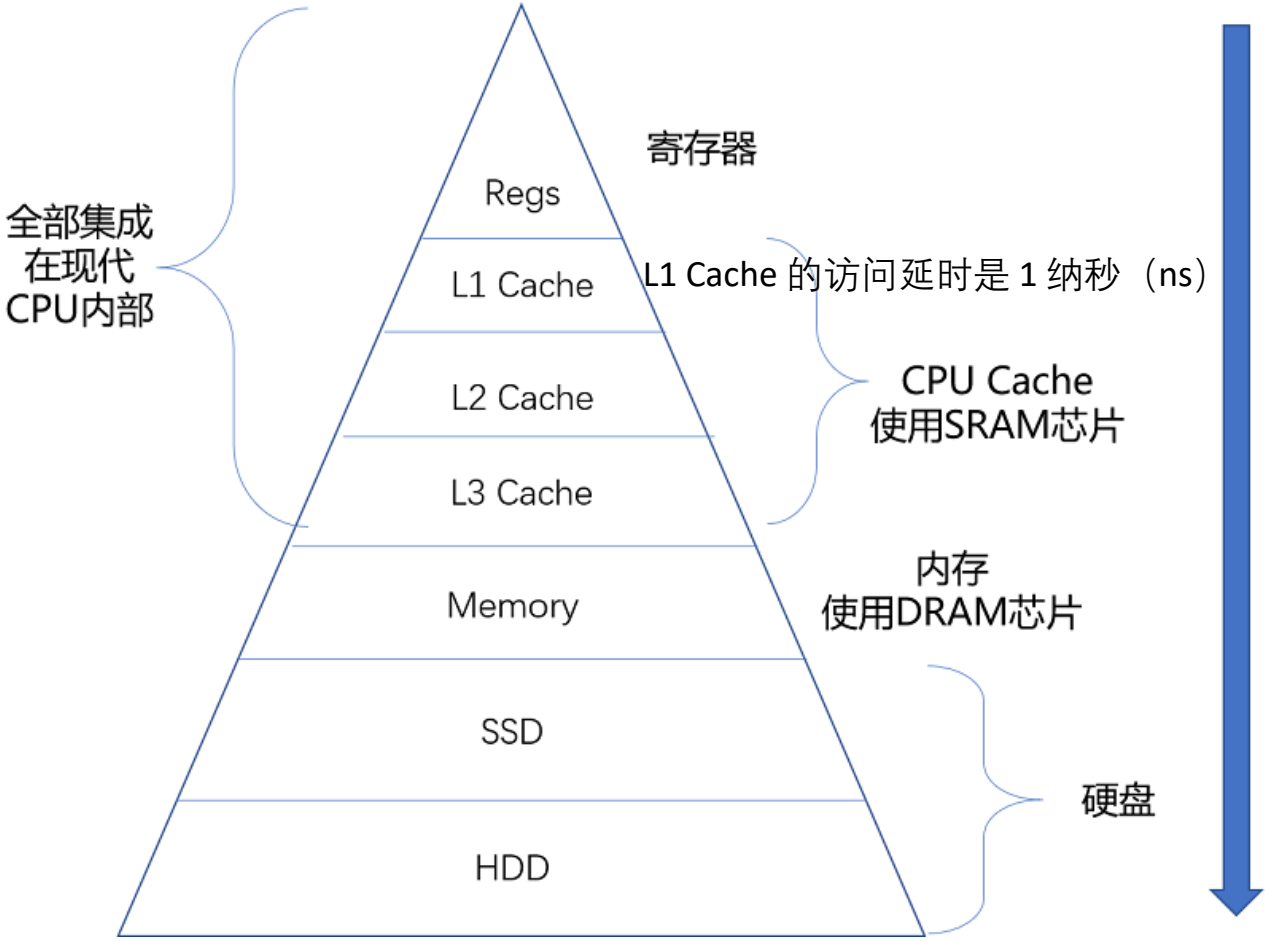
在共享状态下，同样的数据在多个 CPU 核心的 Cache 里都有。所以，**当更新 Cache 数据的时候**，不能直接修改，而是要**向所有的其他 CPU 核心广播一个请求**，要求先把其他 CPU 核心里面的 Cache，都变成无效的状态，然后再更新当前 Cache 里面的数据。这个广播操作，一般**叫作 RFO（Request For Ownership），**也就是**获取当前对应 Cache Block 数据的所有权**

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**

存储器的层次结构



存储器不同层级之间的性能差异和价格差异，在一个数量级以上

每一种存储器设备，只和它相邻的存储设备打交道

存储与I/O系统

01

存储

- 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

I/O

- 接口和设备
- 硬盘

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

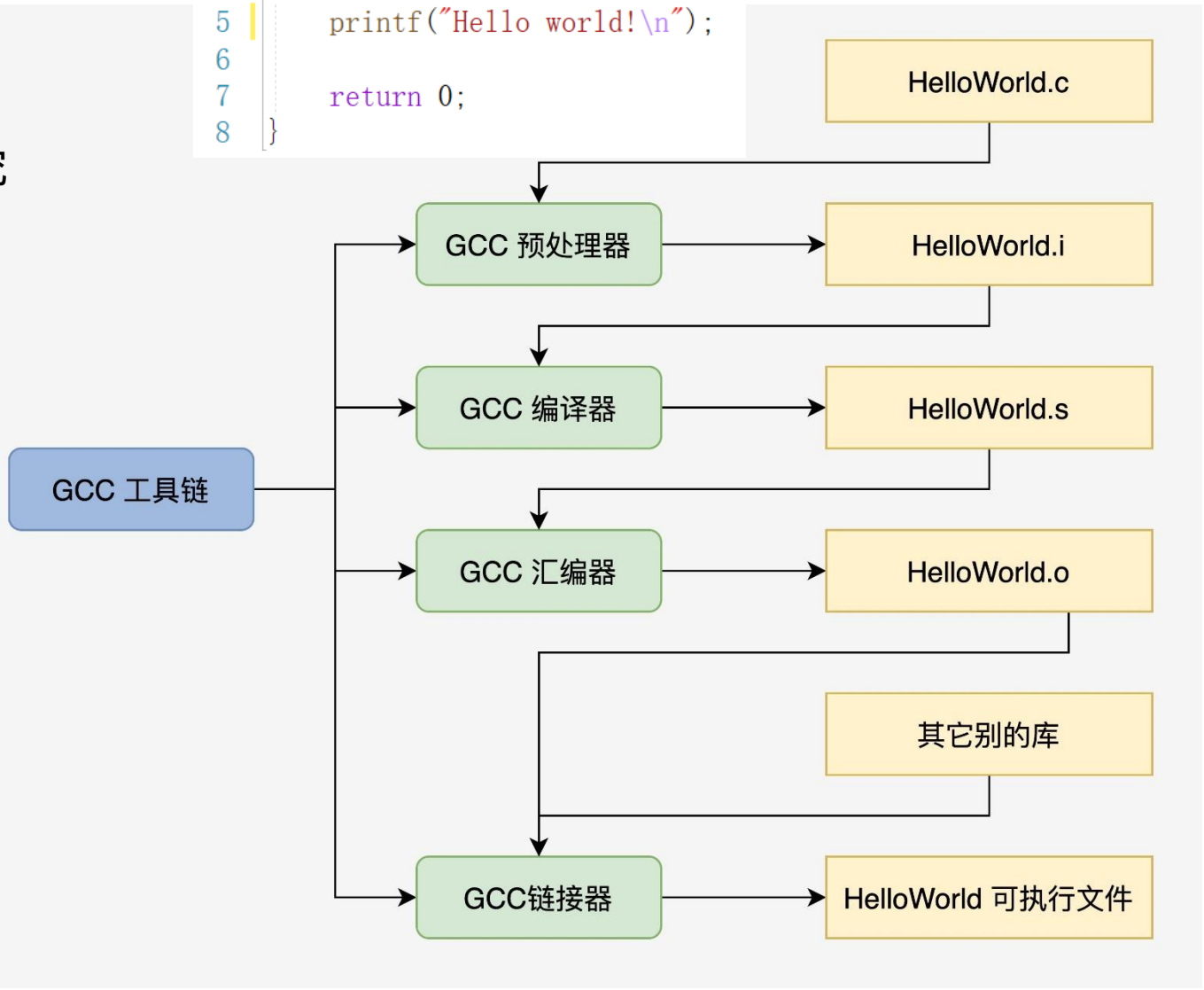
② **gcc HelloWorld.c -S -c HelloWorld.s**
编译：把预处理完的文件进行词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件

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。

```c
#include <stdio.h>
int main()
{
    printf("Hello world!\n");

    return 0;
}
```



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：

① 在物理内存里面，找到一段连续的物理内存空间分配给装载的程序

② 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。

在内存里面，找到一段连续的物理内存空间，分配给要装载的程序

把这段连续的物理内存空间地址，和整个程序指令里指定的内存地址做一个映射

```
$ vim xxxx.c
$ gcc -g -c xxxx.c
$ objdump -d -M intel -S xxxx.o
```

```
#include<stdio.h>
int main()
{
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   rbp
   5:   48 89 e5                mov    rbp,rsp
   8:   48 83 ec 10             sub    rsp,0x10
        srand(time(NULL));
   c:   bf 00 00 00 00          mov    edi,0x0
  11:   e8 00 00 00 00          call   16 <main+0x16>
  16:   89 c7                   mov    edi,eax
  18:   e8 00 00 00 00          call   1d <main+0x1d>
        int r = rand() % 2;
  1d:   e8 00 00 00 00          call   22 <main+0x22>
  22:   99                      cdq
  23:   c1 ea 1f                shr    edx,0x1f
  26:   01 d0                   add    eax,edx
  28:   83 e0 01                and    eax,0x1
  2b:   29 d0                   sub    eax,edx
  2d:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
        int a = 10;
  30:   c7 45 f8 0a 00 00 00    mov    DWORD PTR [rbp-0x8],0xa
        if(r == 0)
  37:   83 7d fc 00             cmp    DWORD PTR [rbp-0x4],0x0
  3b:   75 09                   jne    46 <main+0x46>
                a = 1;
  3d:   c7 45 f8 01 00 00 00    mov    DWORD PTR [rbp-0x8],0x1
  44:   eb 07                   jmp    4d <main+0x4d>
        else
                a = 2;
  46:   c7 45 f8 02 00 00 00    mov    DWORD PTR [rbp-0x8],0x2
        printf("r=%d, a=%d\n",r,a);
  4d:   8b 55 f8                mov    edx,DWORD PTR [rbp-0x8]
  50:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  53:   89 c6                   mov    esi,eax
  55:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # 5c <main+0x5c>
  5c:   b8 00 00 00 00          mov    eax,0x0
  61:   e8 00 00 00 00          call   66 <main+0x66>

        return 0;
  66:   b8 00 00 00 00          mov    eax,0x0
}
  6b:   c9                      leave
  6c:   c3                      ret
wy@wy-virtual-machine:~/文档/Ccode$
```

虚拟内存地址、物理内存地址

指令里用到的内存地址，**虚拟内存地址（Virtual Memory Address）**

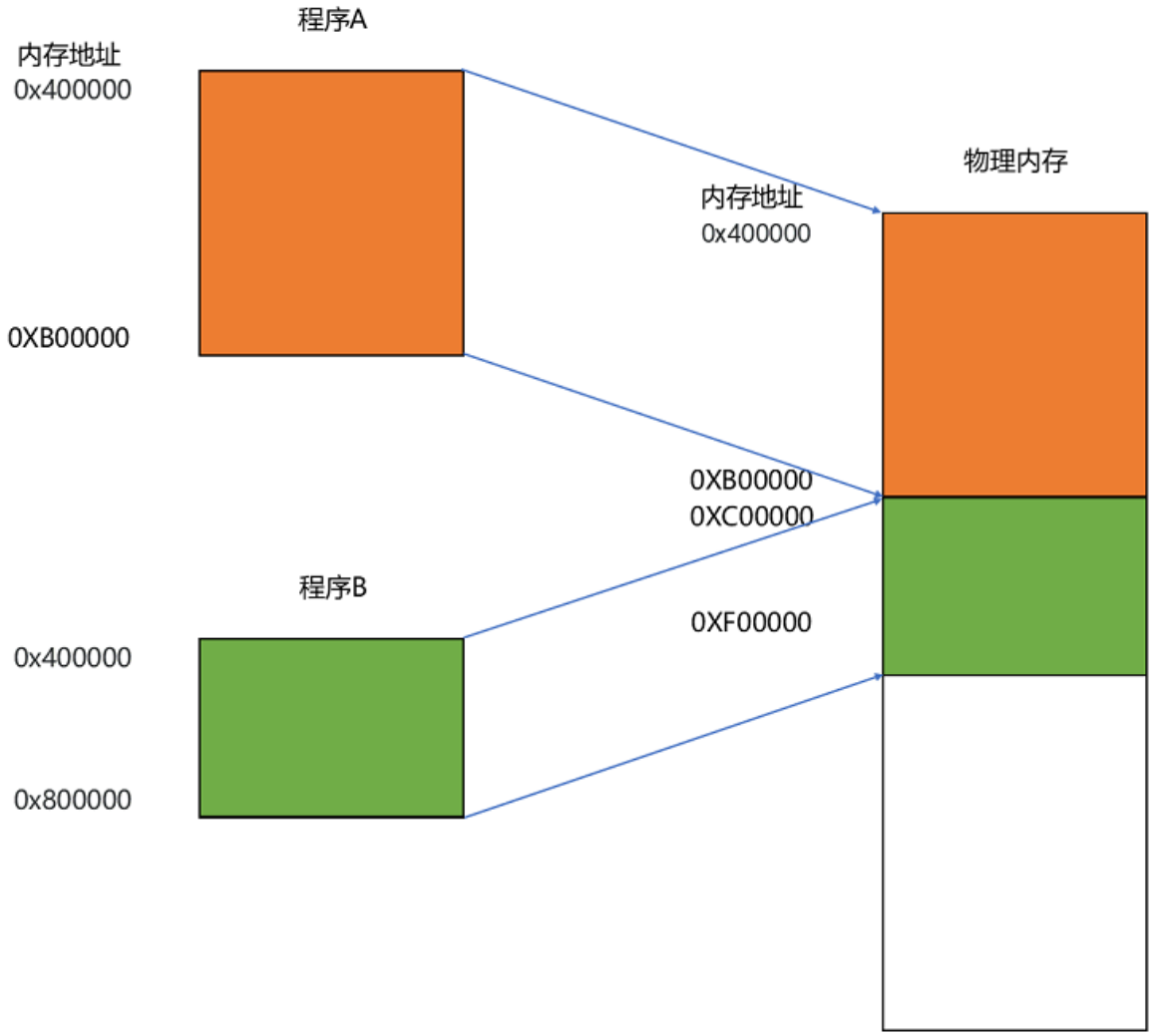
在内存硬件里面的空间地址，**物理内存地址（Physical Memory Address）**

维护一个虚拟内存到物理内存的映射表，实际程序指令执行的时候，会通过虚拟内存地址，找到对应的物理内存地址，然后执行。

因为是连续的内存地址空间，所以只需**维护映射关系的起始地址和对应的空间大小即可。**

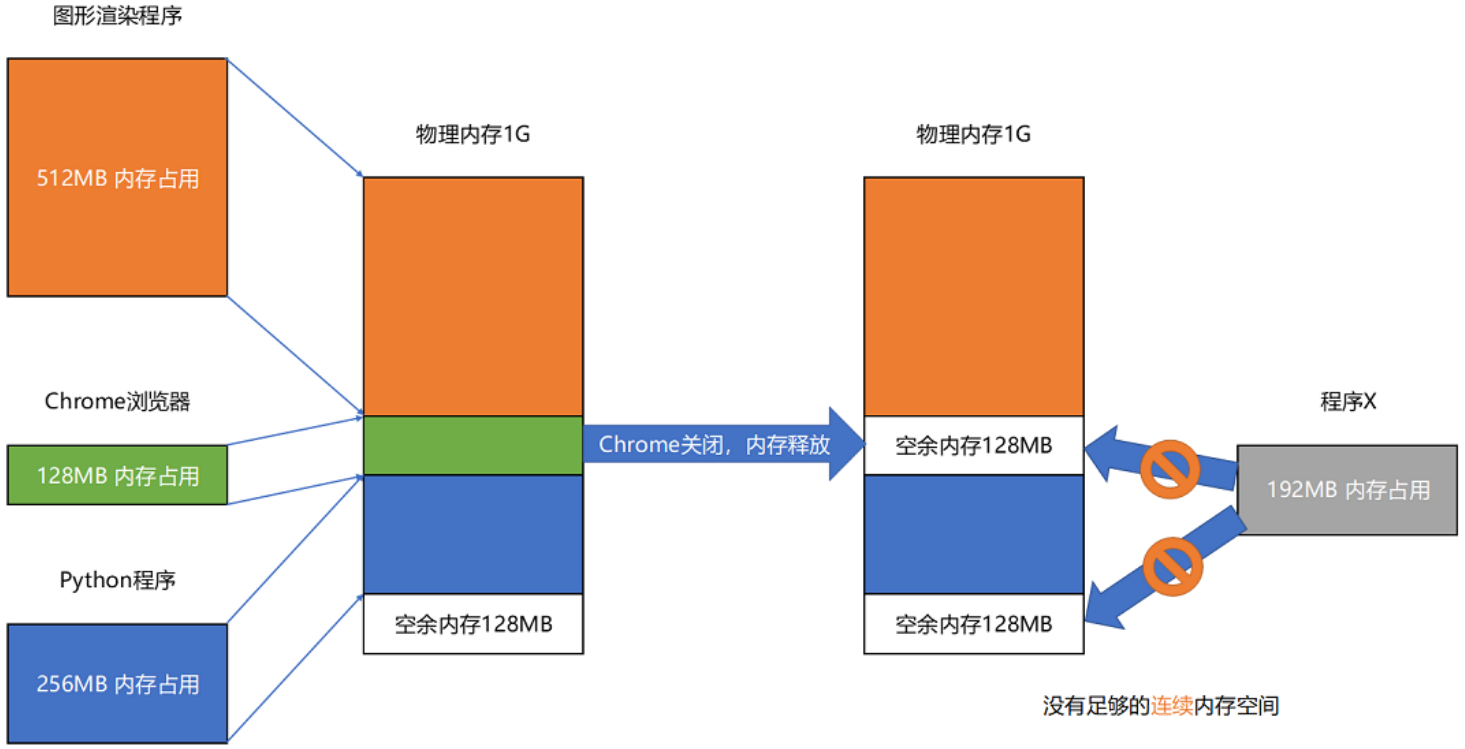
内存分段

分段（Segmentation）：系统分配出来连续的物理内存空间，和虚拟内存地址进行映射



内存分段与内存交换

例。有 1GB 的内存，先启动一个图形渲染程序，占用了 512MB 的内存，接着启动一个 Chrome 浏览器，占用了 128MB 内存，再启动一个 Python 程序，占用了 256MB 内存。这个时候，我们关掉 Chrome，于是空闲内存还有 1024 - 512 - 256 = 256MB



再次装载一个 192MB 的程序，无法加载

内存交换（Memory Swapping）

把 Python 256MB 内存写到硬盘上，然后再从硬盘上读回来，跟在橙色的 512MB 内存后面。Linux swap 硬盘分区，专门给 Linux 操作系统进行内存交换用的

小结

虚拟内存地址、分段，内存交换 → 计算机同时装载运行很多个程序

BUT……硬盘的访问速度比内存慢很多，**每一次内存交换，需要把一大段连续的内存数据写到硬盘上**

所以，如果内存交换的是一个很占空间的程序，整个机器都会卡顿

内存分页（Paging）

当需要进行内存交换的时候，让需要交换写入或者从磁盘装载的数据更少一点，这个办法，在现在计算机的内存管理里面，就叫作内存分页（Paging）

和分配一整段连续的空间给到程序相比，**分页是把整个物理内存空间切成一段段固定尺寸的大小**

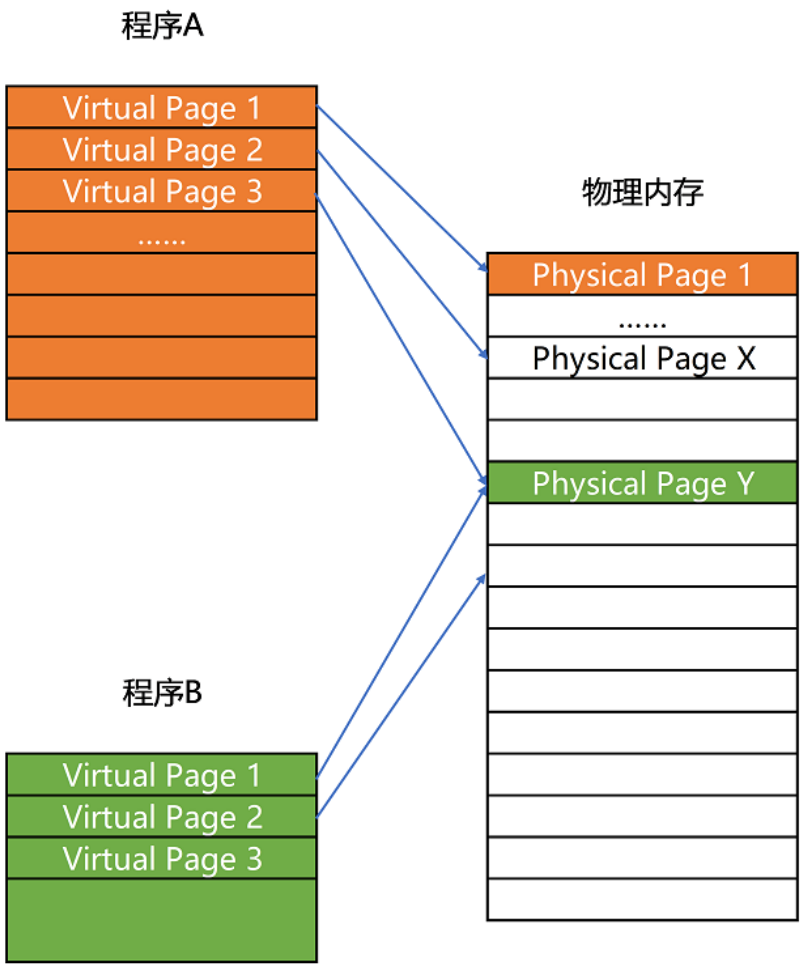
程序所需要占用的虚拟内存空间，也会同样切成一段段固定尺寸的大小。

—— 页（Page）

从虚拟内存地址到物理内存地址的映射，不再整段连续的内存的物理地址，而是按照一个一个页进行。页的尺寸一般远远小于整个程序的大小。**在 Linux 下，通常为 4KB**

预先划分好内存空间，就不存在不能使用的碎片，只有被释放出来的很多 4KB 的页

内存分页



Page未装载时，触发缺页错误
然后装载Page

装载 程序A 的Virtual Page 3
需要进行内存交换

更进一步，在加载程序的时候，不再需要一次性把程序加载到物理内存中。

可以在虚拟内存页和物理内存页之间映射后，**并不真的把页加载到物理内存里，而是只在程序运行时需要用到相应虚拟内存页里面的指令和数据时，再加载到物理内存里。**

当要读取特定的页，却发现数据并没有加载到物理内存里的时候，就会触发一个来自于CPU 的缺页错误（Page Fault）。操作系统会捕捉到这个错误，然后将对应的页，从存放在硬盘上的虚拟内存读取出来，加载到物理内存里。

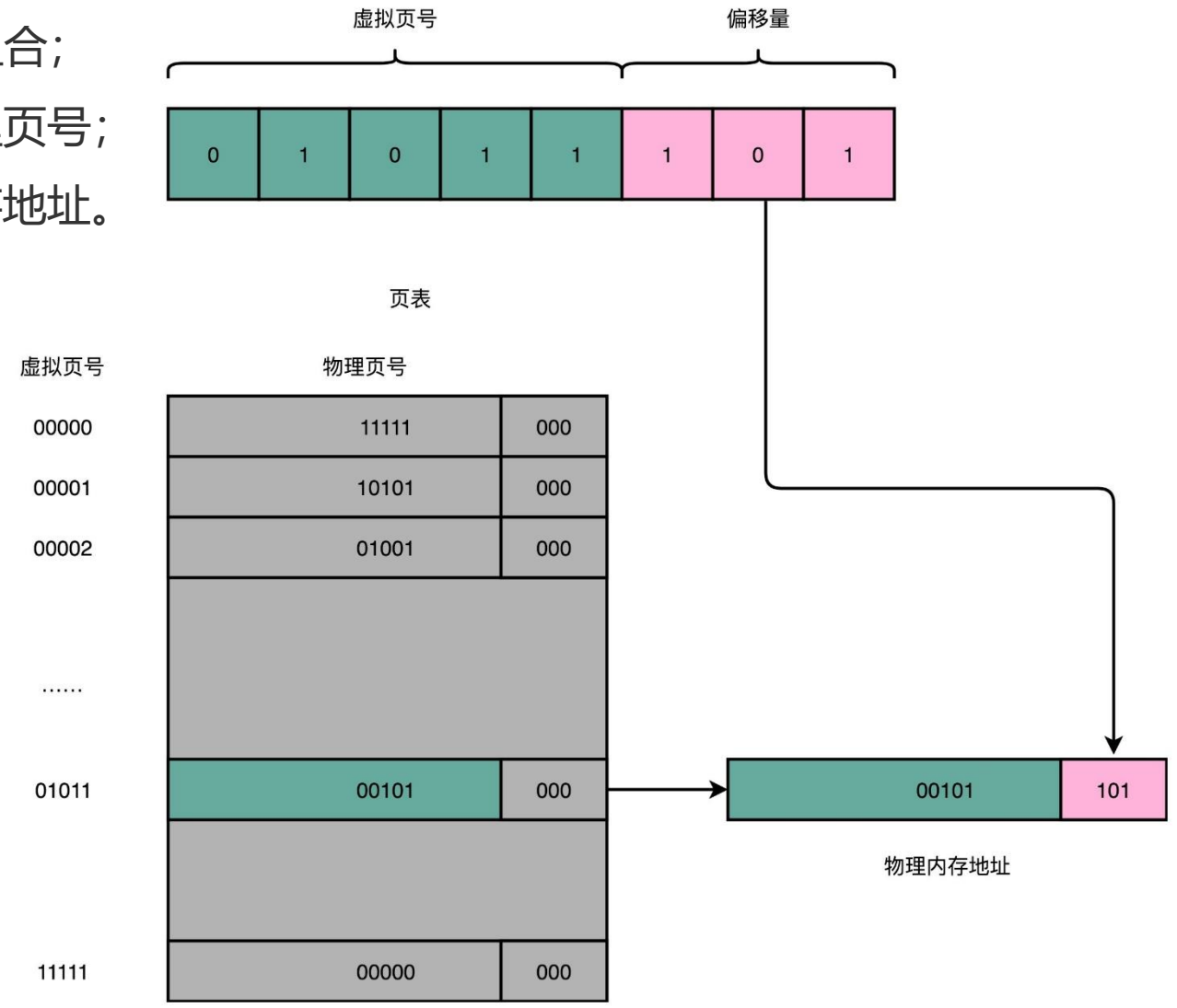
这种方式使得**我们可以运行那些远大于我们实际物理内存的程序**。任何程序都不需要一次性加载完所有指令和数据，只需要加载当前需要用到就行了

内存分页

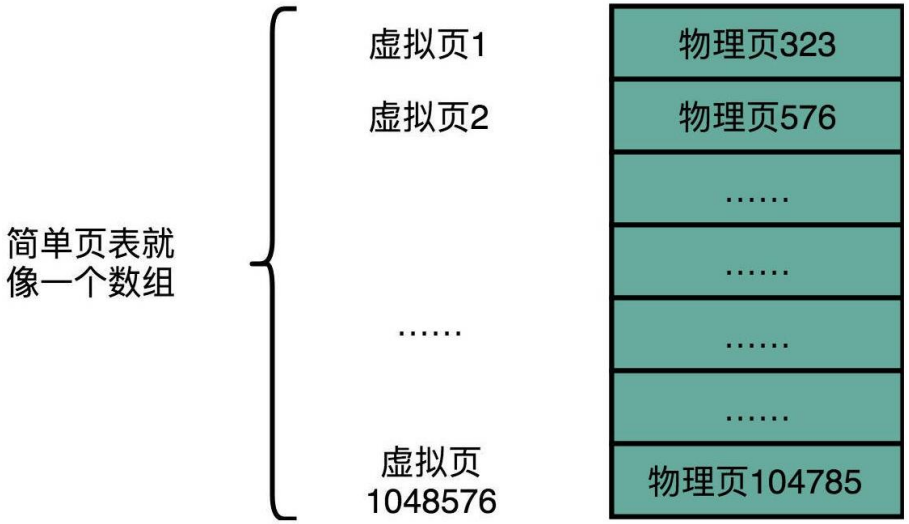
① 把虚拟内存地址，切分成页号和偏移量的组合；

② 从页表里面，查询出虚拟页号所对应的物理页号；

③ 物理页号加上前面的偏移量，得到物理内存地址。



内存分页



任务管理器里的每一个进程，都有属于自己独立的虚拟内存地址空间。这也就意味着，每一个进程都需要这样一个页表，将占用很大的内存。

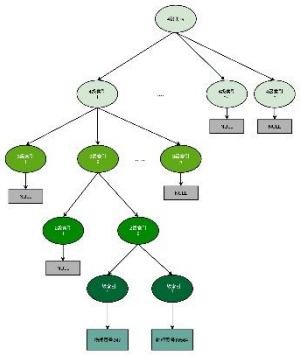
大部分进程所占用的内存是有限的，需要的页也是很有限，只需要去存那些用到的页之间的映射关系（多级页表）

简单页表（数组，紧凑）→ 多级页表 (页表树，稀疏)

4MB / 页表　　约为4MB 的 1/500

32 位的内存地址空间，2^32Bytes，4KB一页，页表一共需要记录 2^20 个到物理页号的映射关系

这个存储关系，就好比一个 2^20 大小的数组。一个页号是完整的 32 位的 4 字节（Byte），这样一个页表就需要 4MB 的空间。



地址变换高速缓存

程序所需要使用的指令，都顺序存放在虚拟内存里面。

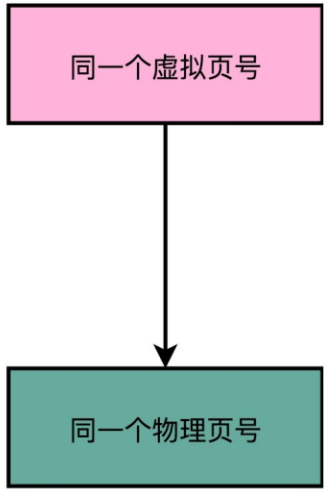
执行的指令，也是一条条顺序执行下去。

对于指令地址和数据的访问，存在“空间局部性”和“时间局部性”。

例如，连续执行 5 条指令，因内存地址都是连续的，**所以这 5 条指令通常都在同一个“虚拟页”里，转换的结果自然也就是同一个物理页号**，就没有必要进行连续5次的内存地址转换。

函数	内存地址	指令
main	14	push rbp
	15	mov rbp, rsp
	……	……
	……	……
	34	call 0 <add>
add	0	push rbp
	1	mov rbp, rsp
	……	……
	……	……
	12	pop rbp
	13	ret
main	39	mov DWORD PTR [rbp-0xc], eax
	3c	
	41	leave
	42	ret

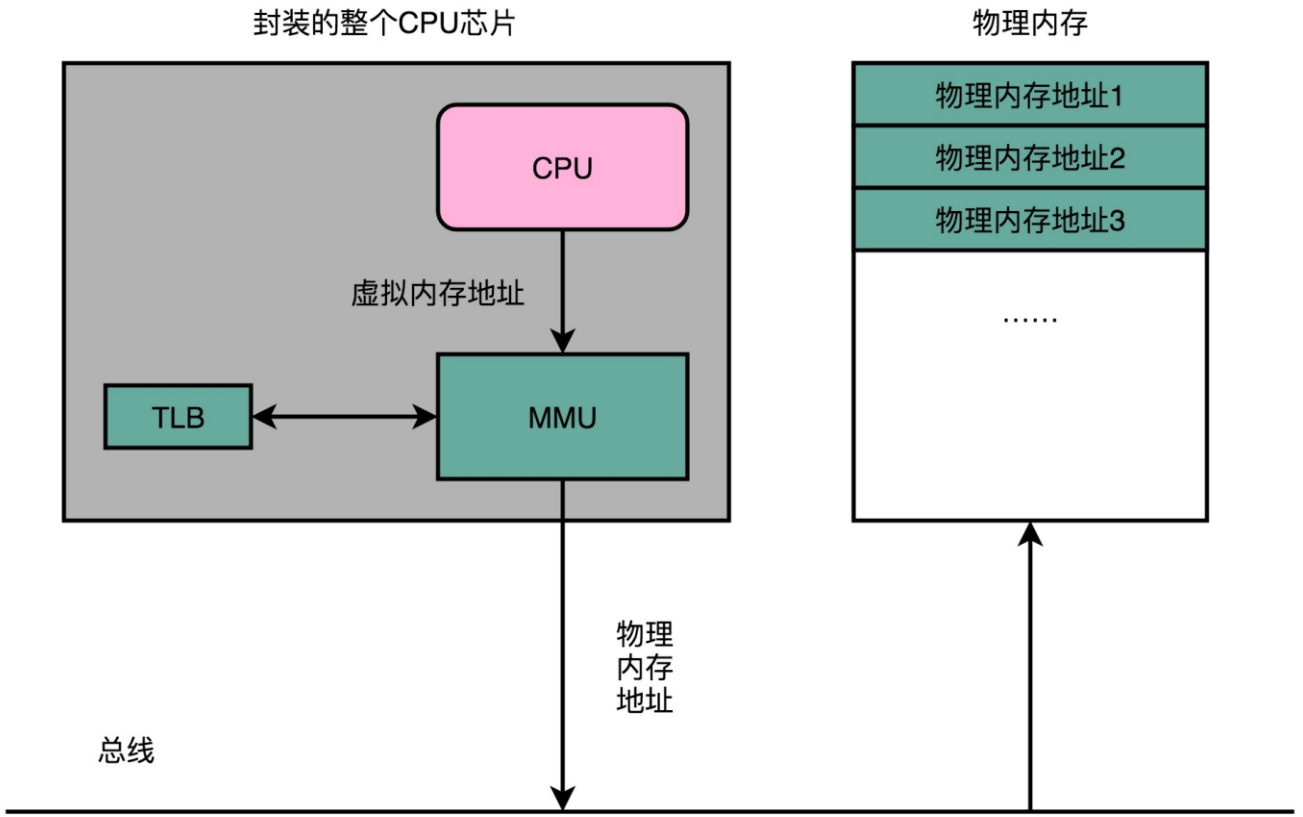
相邻的内存地址



可以把之前的内存转换地址缓存下来，不需要反复去访问内存来进行内存地址转换

地址变换高速缓存（Translation-Lookaside Buffer，TLB）

CPU 里一块缓存芯片，TLB，存放了之前已经进行过地址转换的查询结果。当同样的虚拟地址需要进行地址转换的时候，直接在 TLB 里查询，不需要多次访问内存来完成一次虚拟地址到物理地址的转换。分为指令的 TLB 和数据的 TLB



小结

通过虚拟内存地址、内存交换、内存分页、地址变换等设计，程序本身就不再需要考虑对应的物理内存地址、程序加载、内存管理等问题

任何一个程序，都只需要把内存当成是一块完整而连续的空间来直接使用。

运行一个程序，“必需”的内存是很少的。CPU 只需要执行当前的指令，极限情况下，内存加载一页就成。再大的程序，都可以分成页。在需要用到对应的数据和指令的时候，从硬盘上交换到内存里面来。

以 4K 一页大小，640K 内存也能放下足足 160 页
比尔·盖茨会说出“640K ought to be enough for anyone”

存储与I/O系统

01

存储

- 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

I/O

- 接口和设备
- 硬盘

内存保护

进程的程序，数据，都存放在内存里面

实际程序指令的执行，通过程序计数器里面的地址，去读取内存地址的内容，然后运行对应的指令，使用相应的数据。通过虚拟内存地址和物理内存地址的区分，隔离了各个进程

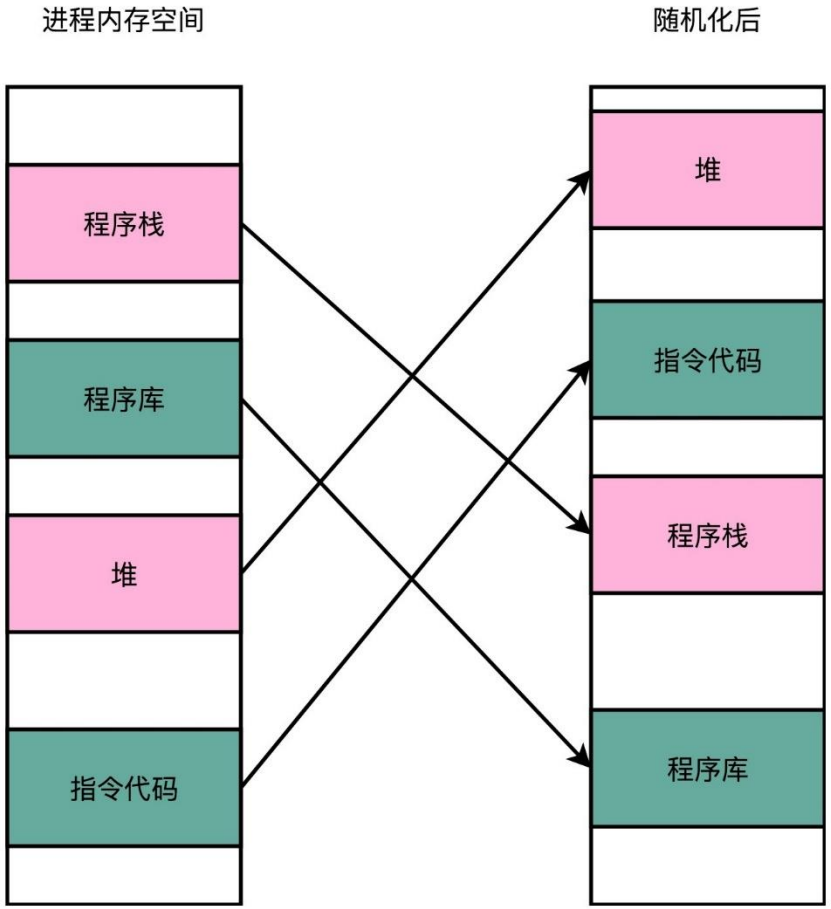
对于内存的管理，计算机还有最底层的安全保护机制。这些机制统称为内存保护（Memory Protection）

可执行空间保护（Executable Space Protection）

对于一个进程使用的内存，只把其中的指令部分设置成“可执行”的，对于其他部分，比如数据部分，不给予“可执行”的权限

把数据部分拿给 CPU，如果这些数据解码后，也能变成一条合理的指令，其实就是可执行的。对于进程里内存空间的执行权限进行控制，可以使得 CPU 只能执行指令区域的代码。对于数据区域的内容，即使找到了其他漏洞想要加载成指令来执行，也会因为没有权限而被阻挡掉

地址空间布局随机化（Address Space Layout Randomization）



“随机化”策略

```
$password = "goodmorning12345";
// 我们的密码是明文存储的

$hashed_password = hash('sha256', password);
// 对应的hash值是 054df97ac847f831f81b439415b2bad05694d1682263599880d7561ee1b77ac
// 但是这个hash值里可以用彩虹表直接“猜出来”原始的密码就是goodmorning12345

$salt = "#21Pb$Hs&Xi923^)?";
$salt_password = $salt.$password;
$hashed_salt_password = hash('sha256', salt_password);
// 这个hash后的slat因为有部分随机的字符串，不会在彩虹表里面出现。
// 261e42d94063b884701149e46eeb42c489c6a6b3d95312e25eee0d008706035f
```

在数据库里，给每一个用户名生成一个随机的、使用了各种特殊字符的盐值（Salt）。这样，我们的哈希值就不再是仅仅使用密码来生成的了，而是密码和盐值放在一起生成的对应的哈希值。哈希值的生成中，包括了一些类似于“乱码”的随机字符串，所以通过彩虹表碰撞来猜出密码的办法就用不了

存储与I/O系统

01

存储

- 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

I/O

- 接口和设备
- 硬盘

输入输出设备（I/O）

输入输出设备，都有两个组成部分:

第一个是它的接口（Interface），

第二个才是实际的 I/O 设备（Actual I/O Device）

硬件设备并不是直接接入到总线上和 CPU 通信的，而是通过接口，用接口连接到总线上，再通过总线和 CPU 通信。

SATA 硬盘，绿色电路板和黄色的齿状部分就是接口(Interface)

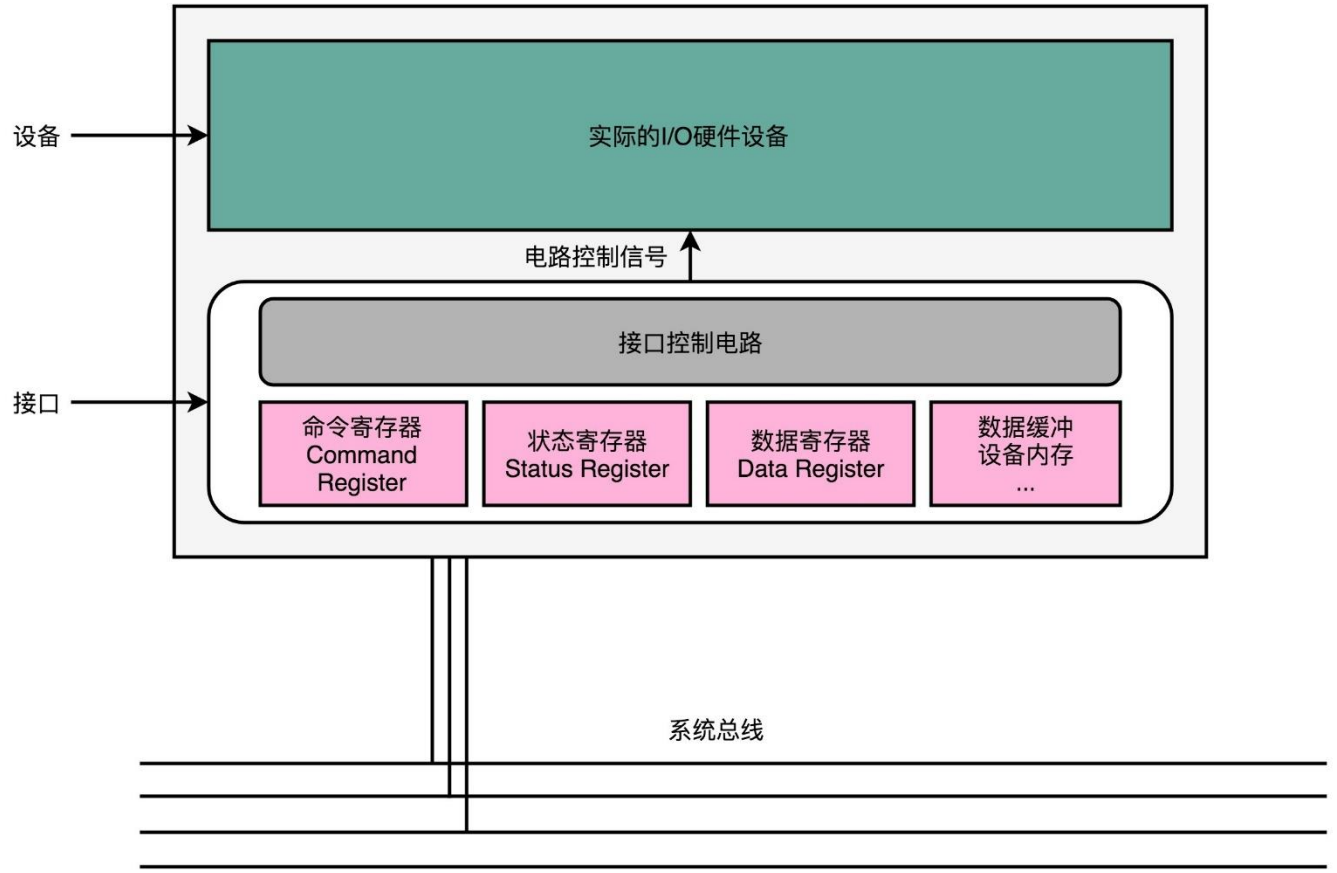
黄色齿状和主板对接的接口，绿色的电路板就是控制电路

输入输出设备（I/O）

数据寄存器。比如要打印的内容是“上”

命令寄存器（Command Register）。CPU 发送一个命令，告诉打印机，要进行打印工作。控制电路会做两个动作：
1）设置状态寄存器里面的状态： not-ready告诉CPU，设备在工作，所以这个时候，CPU 再发送数据或者命令过来，都是没有用的。直到前面的动作已经完成，状态寄存器重新变成了 ready 状态
2）控制打印机进行打印。

数据缓冲。一次性把整个文档“上海体育学院”传输到打印机的内存或者数据缓冲区里面一起打印



输入输出设备（I/O）

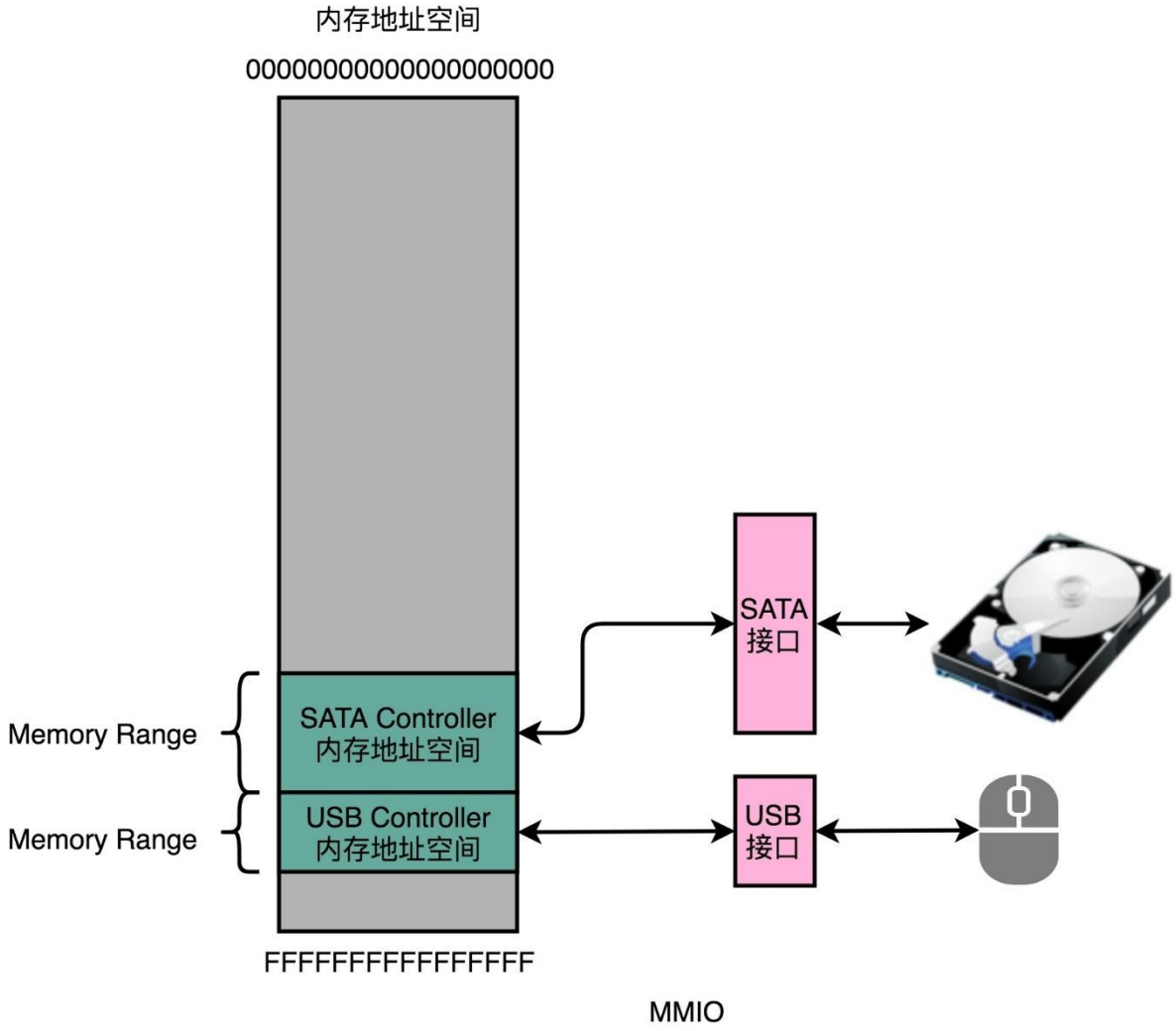
计算机会把 I/O 设备的各个寄存器，以及 I/O 设备内部的内存地址，都映射到主内存地址空间里。

主内存的地址空间里，给不同的 I/O 设备预留一段一段的内存地址

CPU 想要和这些 I/O 设备通信的时候，就往这些地址发送数据。

I/O 设备，会监控地址线，并且在 CPU 往自己地址发送数据的时候，把传输过来的数据，接入到对应的设备里的寄存器和内存里面来。

CPU 无论是向 I/O 设备发送命令、查询状态还是传输数据，都可以通过这样的方式。**内存映射IO**（Memory-Mapped I/O，简称 MMIO）。



硬盘

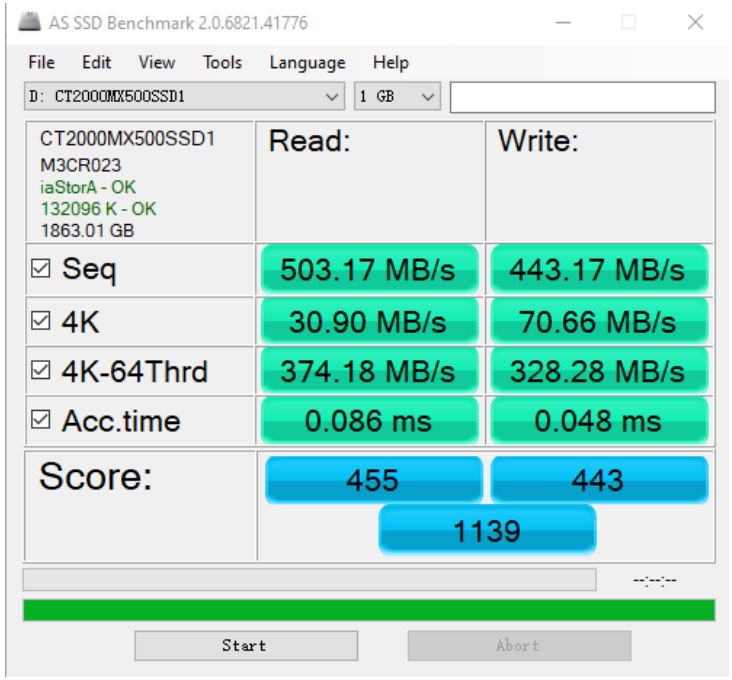
AS SSD 软件，测试硬盘性能

硬盘厂商的性能报告，通常两个指标
一个是响应时间（Response Time）
一个是数据传输率（Data Transfer Rate）

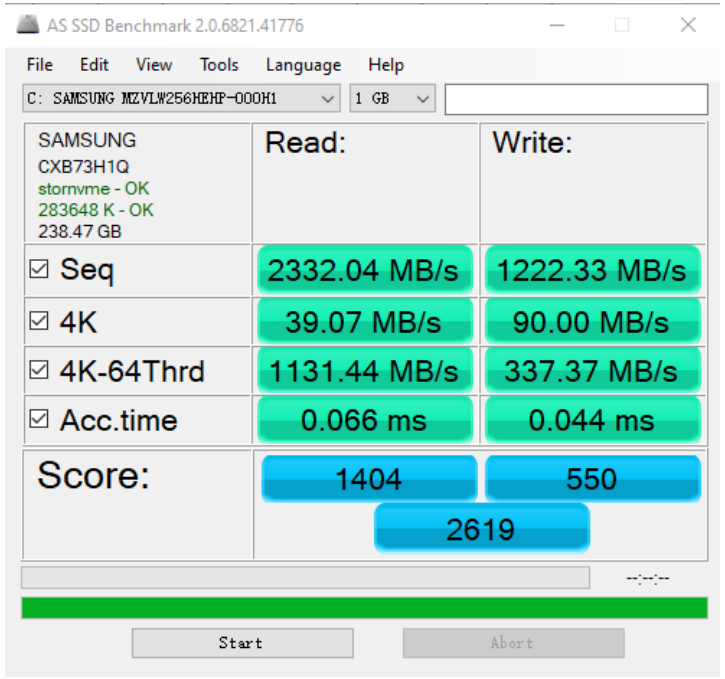
硬盘

现在常用的硬盘有两种：

- HDD 硬盘，机械硬盘， SATA 3.0 的接口，带宽 6Gb/s （每秒传输 768MB ），HDD 硬盘的数据传输率， **200MB/s 左右。**
- SSD 硬盘，一般也被叫作固态硬盘：SATA 3.0 的接口、PCI Express 的接口



AS SSD 测 SATA 接口 SSD 硬盘



AS SSD 测 PCI Express 接口 SSD 硬盘

读—— 2GB/s 左右，约为 HDD 硬盘的 10 倍

写——1.2GB/s

硬盘

Acc.Time：程序发起一个硬盘的写入请求，直到这个请求返回的时间。几十微秒

HDD 的硬盘: 通常会在几毫秒到十几毫秒，几十倍 ~ 几百倍。

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

硬盘

AS SSD 的性能指标 “4K” ，程序随机读取磁盘上**某一个 4KB 大小的数据**，一秒之内可以读取到的数据量

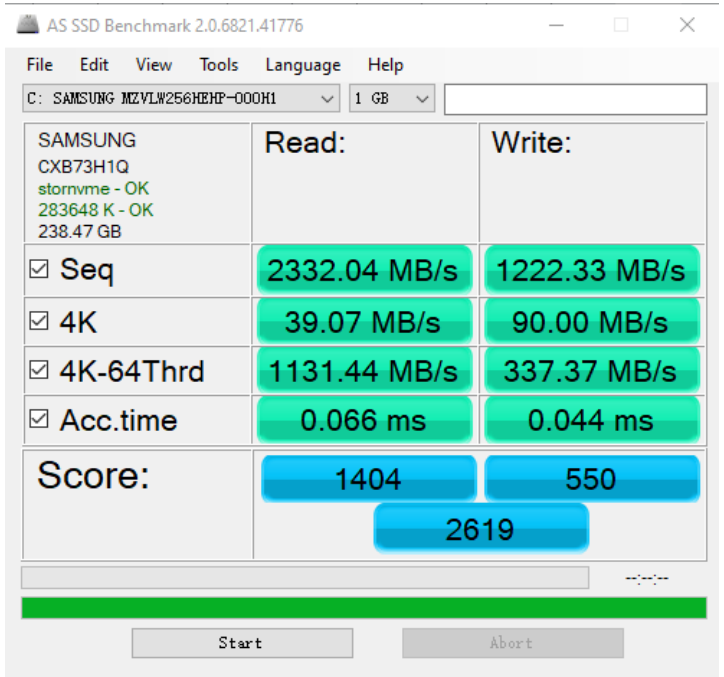
PCI Express 的接口，在随机读写的时候，数据传输率也只 40MB/s 左右，是顺序读写情况下的几十分之一

40MB/s ，一次读取 4KB

40MB / 4KB = 10,000

一秒之内，这块 SSD 硬盘可以随机读取 1 万次的 4KB 的数据；写入，90MB /4KB 差不多是 2 万多次

每秒读写的次数—— IOPS，也就是每秒输入输出操作的次数



AS SSD 测 PCI Express 接口 SSD 硬盘

读—— 2GB/s 左右，约为 HDD 硬盘的 10 倍

写——1.2GB/s

IOPS 和 DTR（Data Transfer Rate，数据传输率）是输入输出性能的核心指标。

HDD 硬盘的 IOPS 通常也就在 100 左右

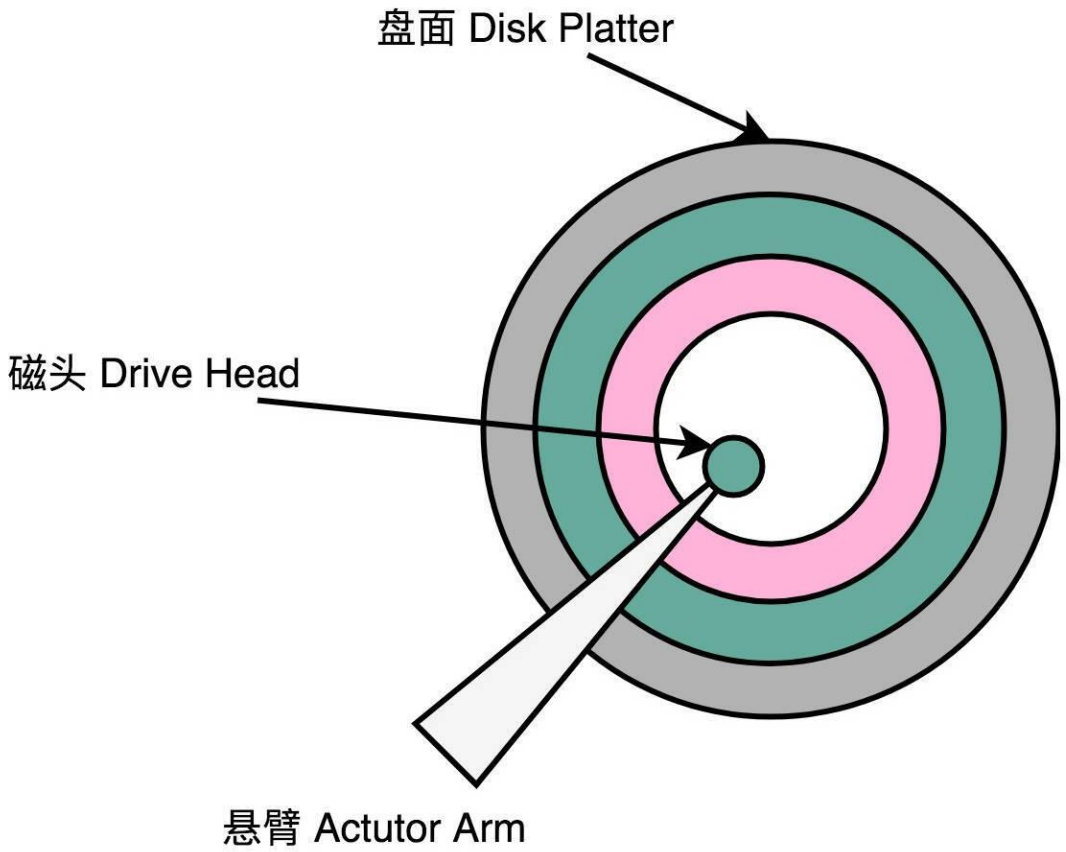
HDD硬盘

一块机械硬盘：盘面、磁头和悬臂三个部件

盘面，通常用铝、玻璃或者陶瓷材质做成光滑盘片，上有一层磁性涂层，记录数据；盘面中心有一受电机控制的转轴，控制盘面旋转

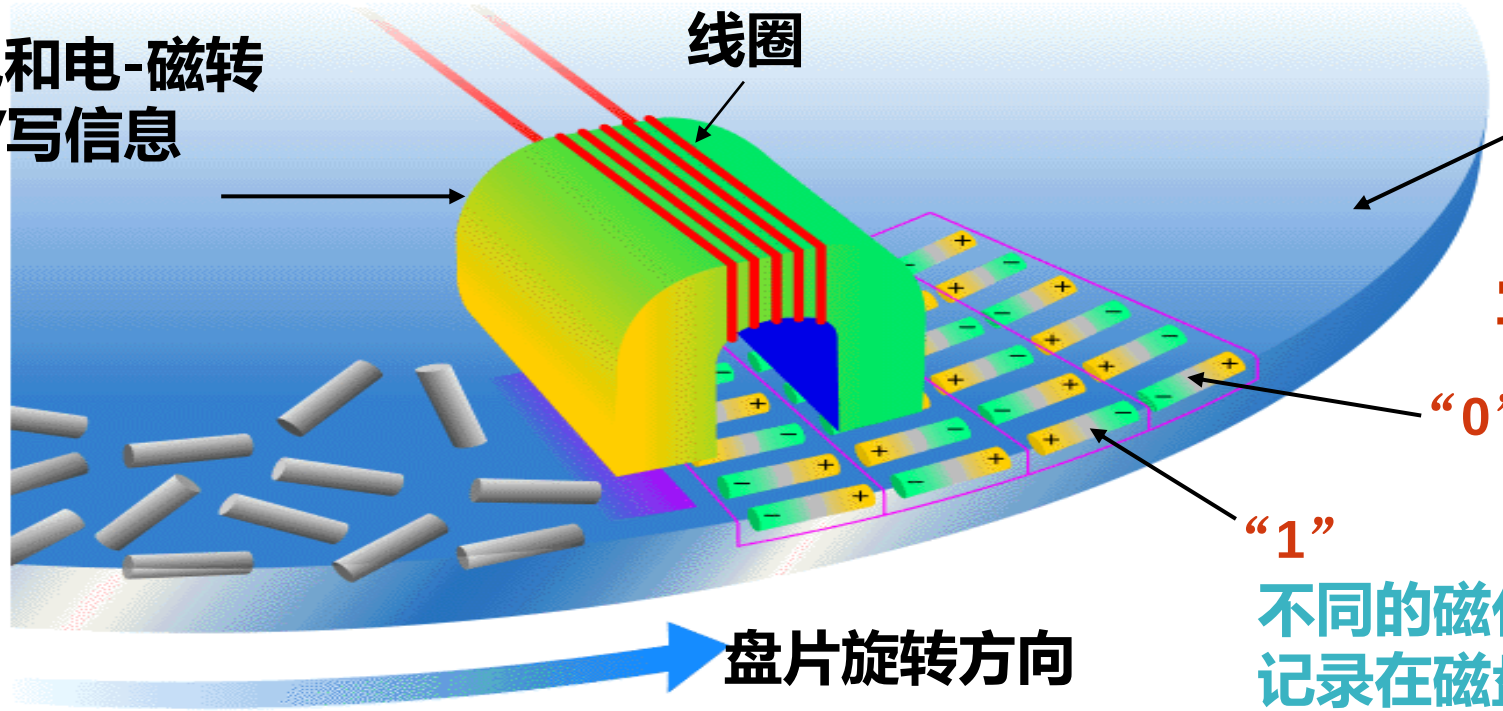
硬盘的转速，盘面中心电机控制的转轴的旋转速度， RPM（Rotations Per Minute）

7200 转， 7200RPM，开机供电之后，硬盘每分钟转上 7200 圈，折算到每一秒钟： 120 圈。



磁盘存储器的信息存储原理

磁头：磁-电和电-磁转换，用于读/写信息



读：磁头固定不动，盘体运动。根据感应电压的不同的极性，可确定读出为0或1。

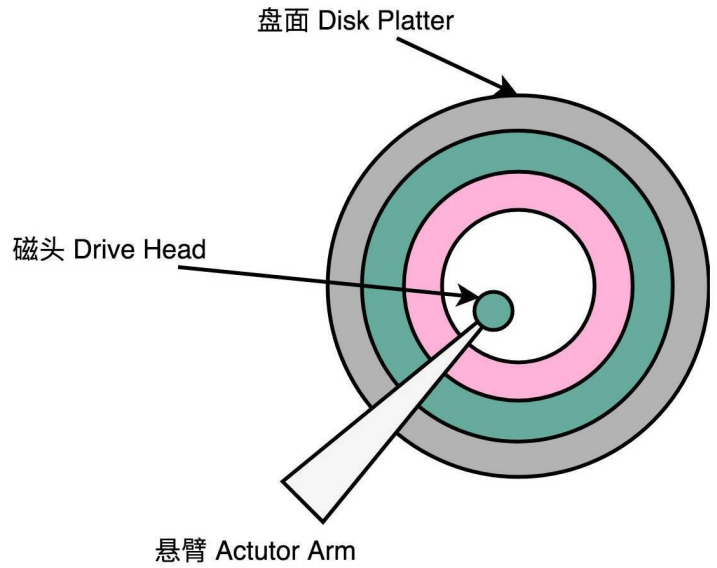
HDD硬盘

一块机械硬盘：盘面、磁头和悬臂三个部件

磁头（Drive Head）。数据通过磁头，从盘面上读取到，通过电路信号传输给硬盘控制电路、硬盘接口，再传输到总线上。

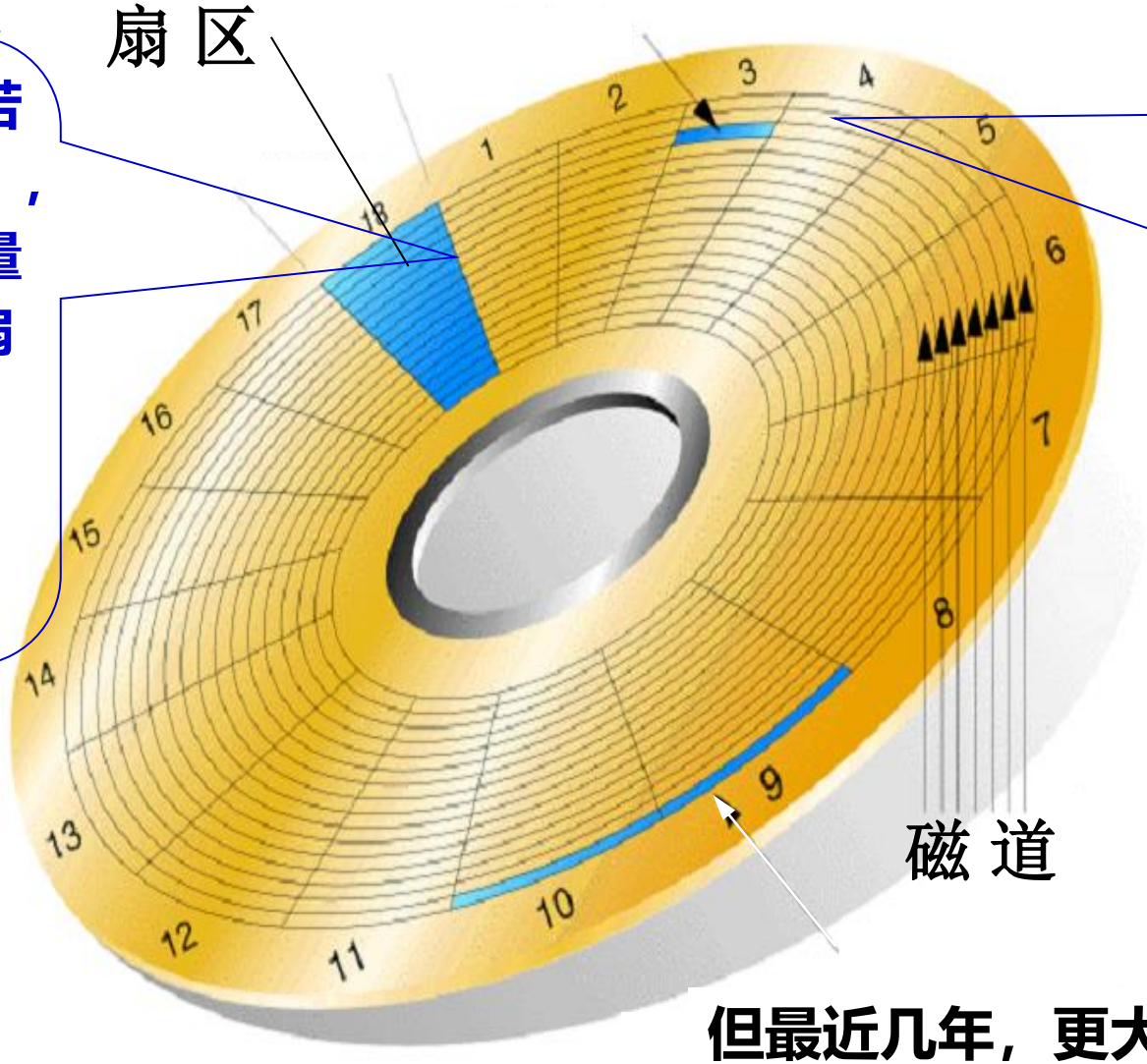
一块硬盘上下堆叠了很多个盘面，
每个盘面的正反两面都有对应的磁头

悬臂（Actutor Arm）。悬臂链接在磁头上，并且在一定范围内把磁头定位到盘面的某个特定的磁道（Track）上。



磁道和扇区

每个磁道被划分为若干段（段又叫扇区），每个扇区的存储容量为512字节。每个扇区都有一个编号

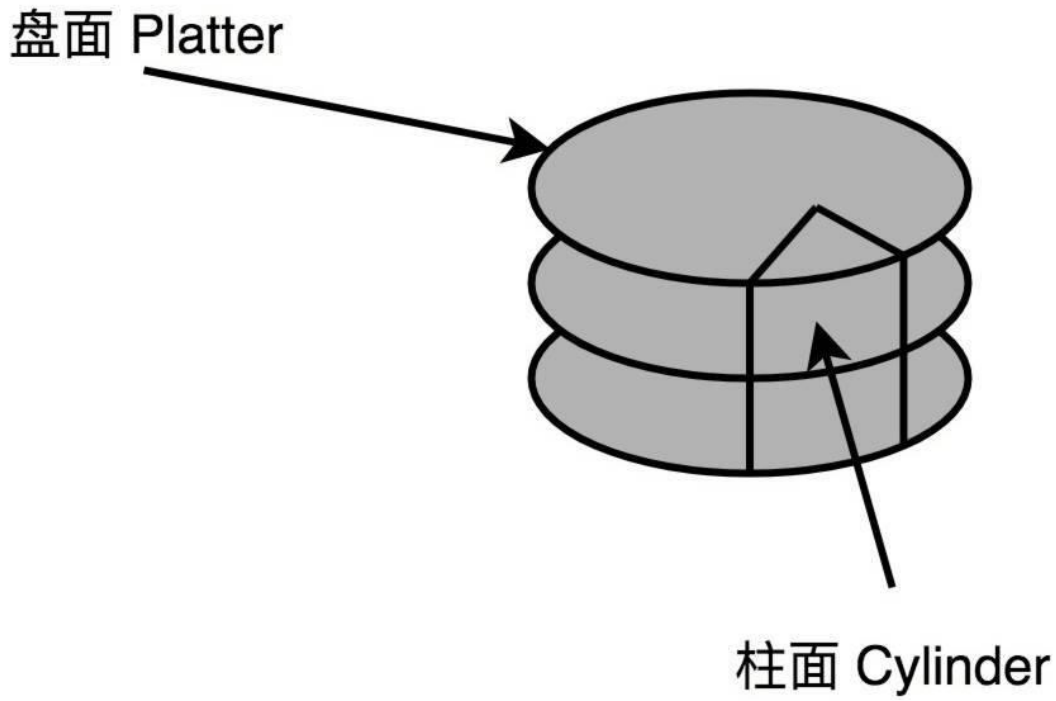


磁盘表面被分为许多同心圆，每个同心圆称为一个磁道。每个磁道都有一个编号，最外面的是0磁道

但最近几年，更大、更高效的4096字节扇区，通常称为4K扇区

HDD硬盘

上下平行的一个一个盘面的相同扇区，柱面（Cylinder）。



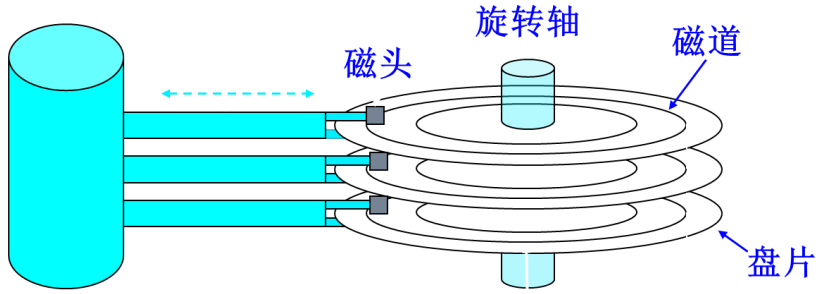
HDD硬盘，平均延时与平均寻道

盘面旋转到某一扇区，悬臂可以定位到某磁道

平均延时（Average Latency）。盘面旋转，把扇区对准悬臂位置的时间。随机情况下，平均找到一个几何扇区的时间为旋转半圈盘面。
7200 转的硬盘，一秒里面，就可以旋转 240 个半圈。那么，这个平均延时就是1s / 240 = **4.17ms**

平均寻道时间（Average Seek Time），盘面选转之后，悬臂定位到扇区某磁道的时间。 HDD 硬盘的平均寻道时间一般**在 4-10ms**

随机在整个硬盘上找一个数据，访问一次数据的时间：
“平均延时 + 寻道时间”需要 8-14 ms



7200 转的硬盘，一秒钟随机的 IO 访问次数：
1s / 8 ms = 125 IOPS 或者 1s / 14ms = 70 IOPS
HDD 硬盘的 IOPS 每秒 100 次左右

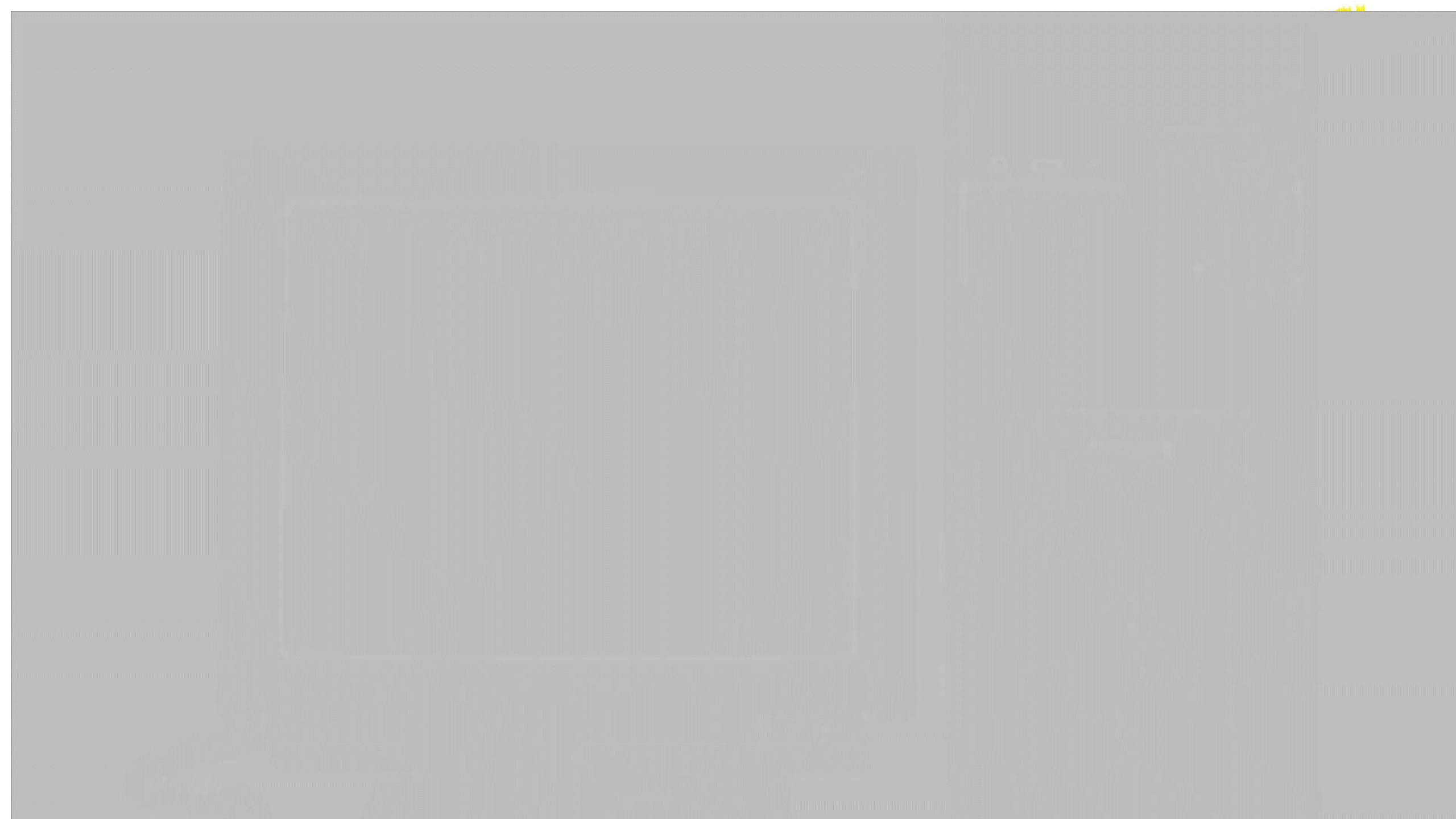
SSD硬盘

只有 100 的 IOPS，很难满足现在互联网海量高并发的请求。所以，今天的数据库，把数据存储在 **SSD 硬盘**上。不过，如果我们把时钟倒播 20 年，没有现在这么便宜的 SSD 硬盘,数据库里面的数据，只能存放在 HDD 硬盘上

SSD 硬盘在 2010 年前后，进入了主流的商业应用

在数据中心里，往往我们会用上 10000 转，乃至 15000 转的硬盘。

访问类型	机械硬盘（HDD硬盘）	固态硬盘（SSD硬盘）
随机读	慢	非常快
随机写	慢	快
顺序写	快	非常快
耐用性（重复擦写）	非常好	差



存储与I/O系统

01

存储

- 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

I/O

- 接口和设备
- 硬盘