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）

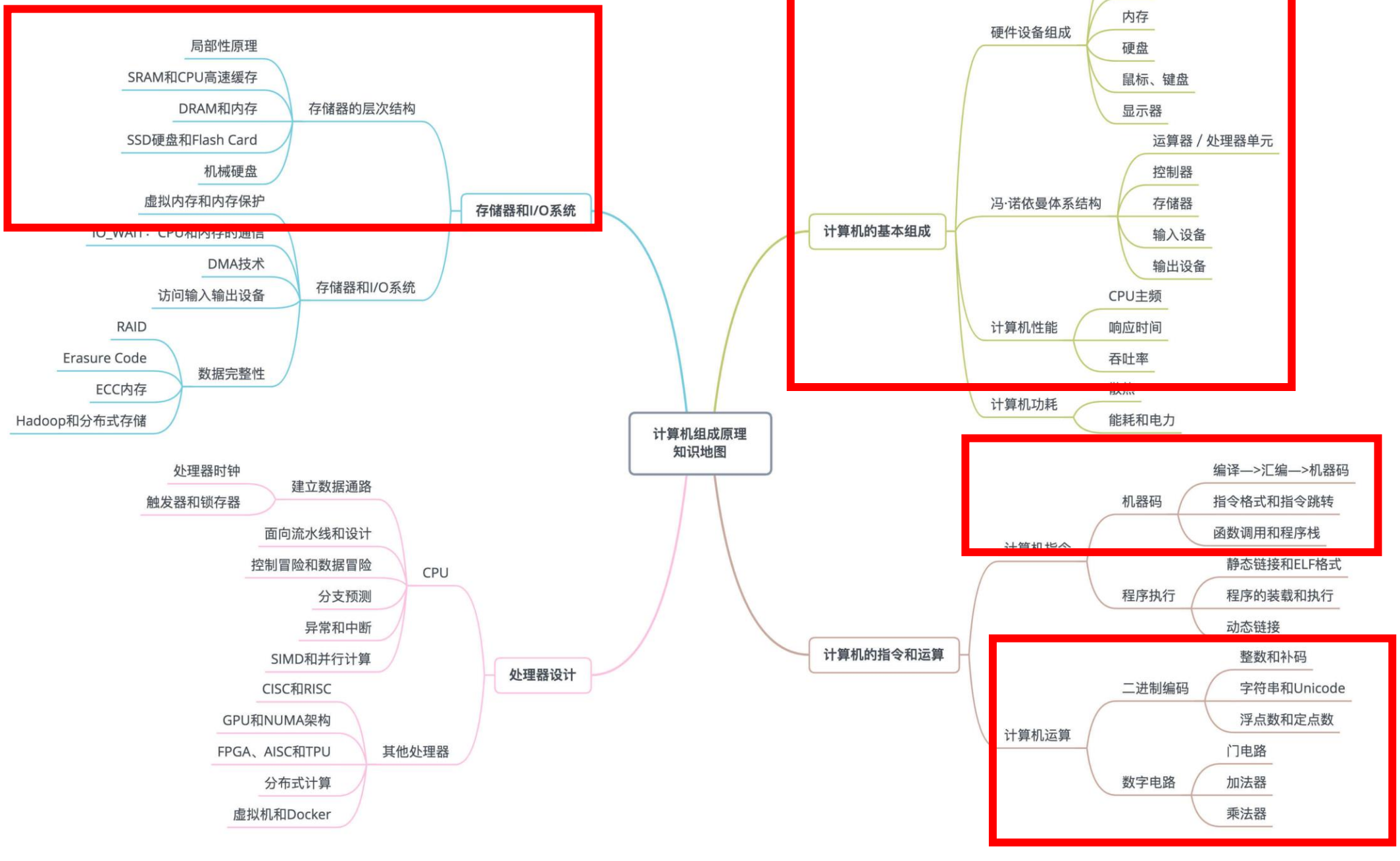
回顾要点（II）

存储与I/O系统

1. 理解存储系统层次结构
2. 局部性原理

其他

1. EXCEL应用
2. 网络与数据库应用


计算机组成原理
知识地图
存储器和I/O系统
存储器的层次结构
局部性原理
SRAM和CPU高速缓存
DRAM和内存
SSD硬盘和Flash Card
机械硬盘
存储器和I/O系统
虚拟内存和内存保护
IO_WAIT：CPU和内存的通信
DMA技术
访问输入输出设备
数据完整性
RAID
Erasure Code
ECC内存
Hadoop和分布式存储
处理器设计
CPU
建立数据通路
处理器时钟
触发器和锁存器
面向流水线和设计
控制冒险和数据冒险
分支预测
异常和中断
SIMD和并行计算
其他处理器
CISC和RISC
GPU和NUMA架构
FPGA、AISC和TPU
分布式计算
虚拟机和Docker
计算机的基本组成
硬件设备组成
CPU
主板
内存
硬盘
鼠标、键盘
显示器
冯·诺依曼体系结构
运算器 / 处理器单元
控制器
存储器
输入设备
输出设备
计算机性能
CPU主频
响应时间
吞吐率
计算机功耗
能耗和电力
计算机的指令和运算
计算机指令
机器码
编译—>汇编—>机器码
指令格式和指令跳转
函数调用和程序栈
程序执行
静态链接和ELF格式
程序的装载和执行
动态链接
计算机运算
二进制编码
整数和补码
字符串和Unicode
浮点数和定点数
数字电路
门电路
加法器
乘法器

课程考核方式（考试课）

4学分

课堂表现（10%）

作业（20%）

平时考勤（10%）

闭卷考试（60%）

存储与I/O系统

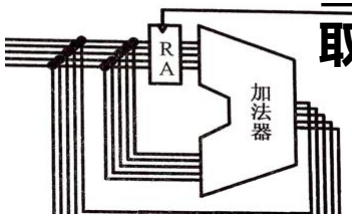
01

存储与I/O系统

- 理解存储系统层次结构
- 局部性原理

02

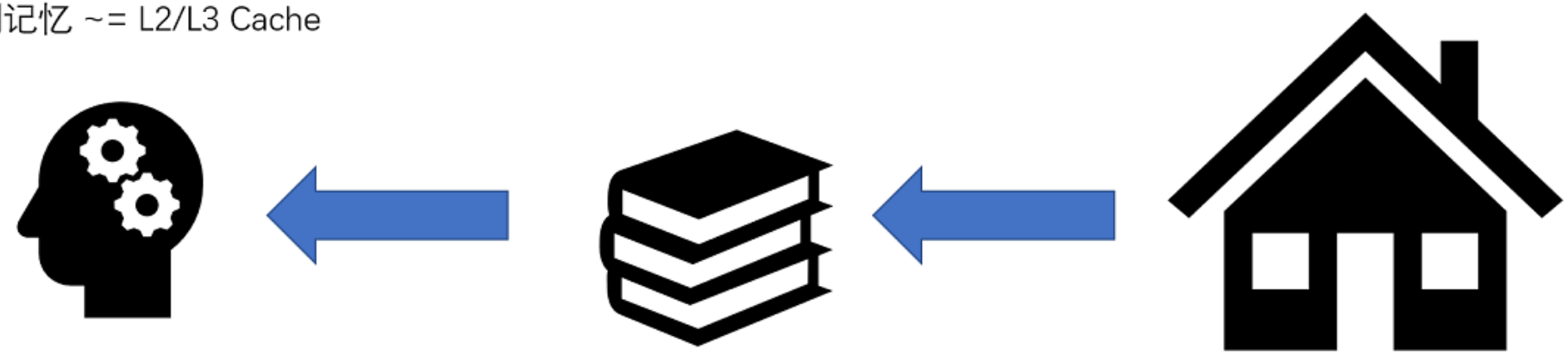
当要执行算术逻辑操作时，控制器首先将**内存里的数据取到寄存器中**，然后运算器**再从寄存器中取出进行运算**



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）

理解存储器的层次结构

理解存储器的层次结构

SRAM ——“静态”存储器

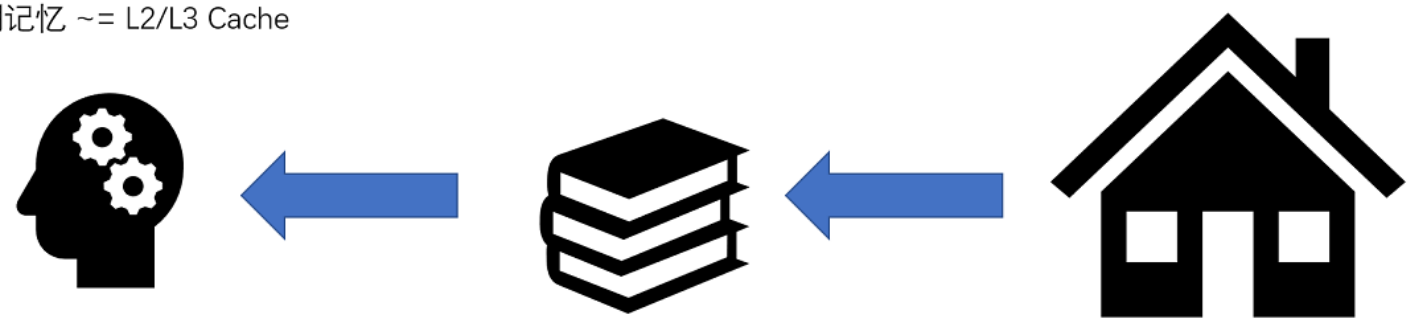
- 只要处在通电状态，里面的数据就可以保持存在。一旦断电，里面的数据就会丢失。
- 在 SRAM 里面，一个比特的数据，需要 6 ~ 8 个晶体管。所以 SRAM 的存储密度不高。同样的物理空间下，能够存储的数据有限。
- SRAM 的电路简单，访问速度非常快。

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）

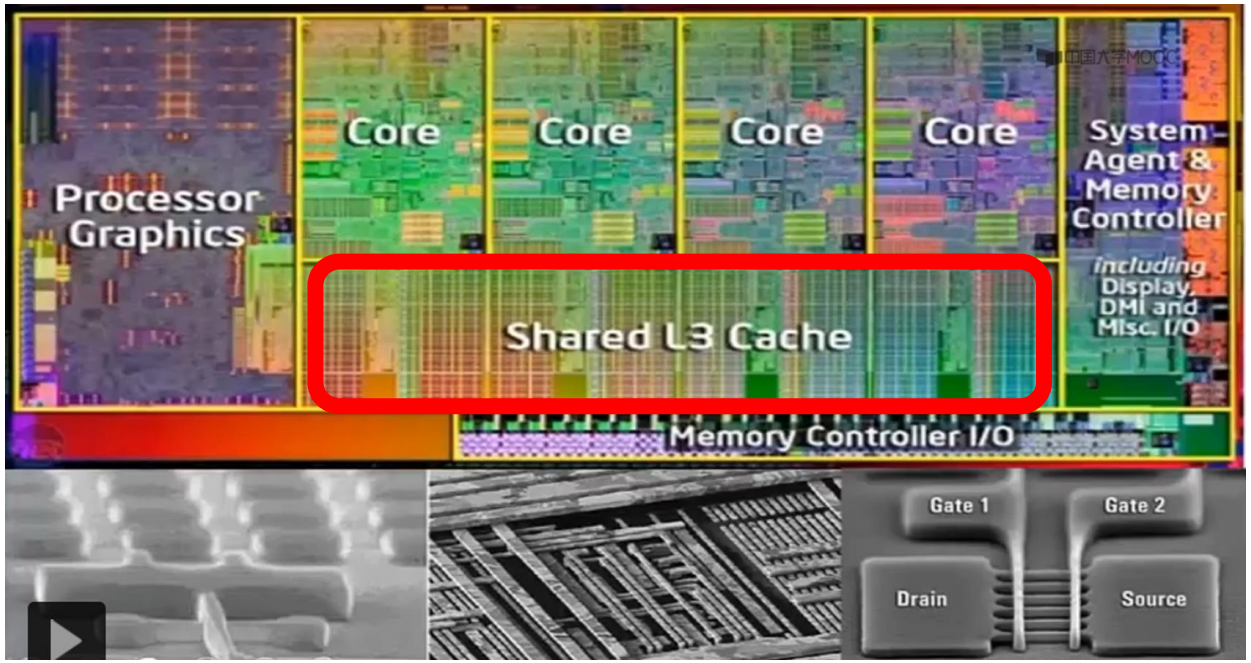
L1 Cache 往往嵌在 CPU 核心的内部。每个 CPU 核心有一块属于自己的 L1 高速缓存，通常分成**指令缓存和数据缓存**，分开存放 CPU 所用指令和数据

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。


Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including Display, DMI and Misc. I/O
Shared L3 Cache
Memory Controller I/O
Gate 1
Gate 2
Drain
Source

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）

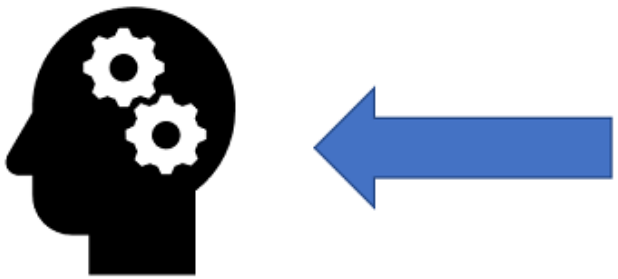
- DRAM 的一个比特，只需要一个晶体管和一个电容就能存储。所以，DRAM 在**同样的物理空间下**，能够存储的数据也就更多，也就是存储的“密度”更大，比 SRAM 芯片便宜不少。
- 数据是存储在电容里的，电容会不断漏电，所以需要定时刷新充电，才能保持数据不丢失。需要靠不断地“刷新”，才能保持数据被存储起来。

- 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

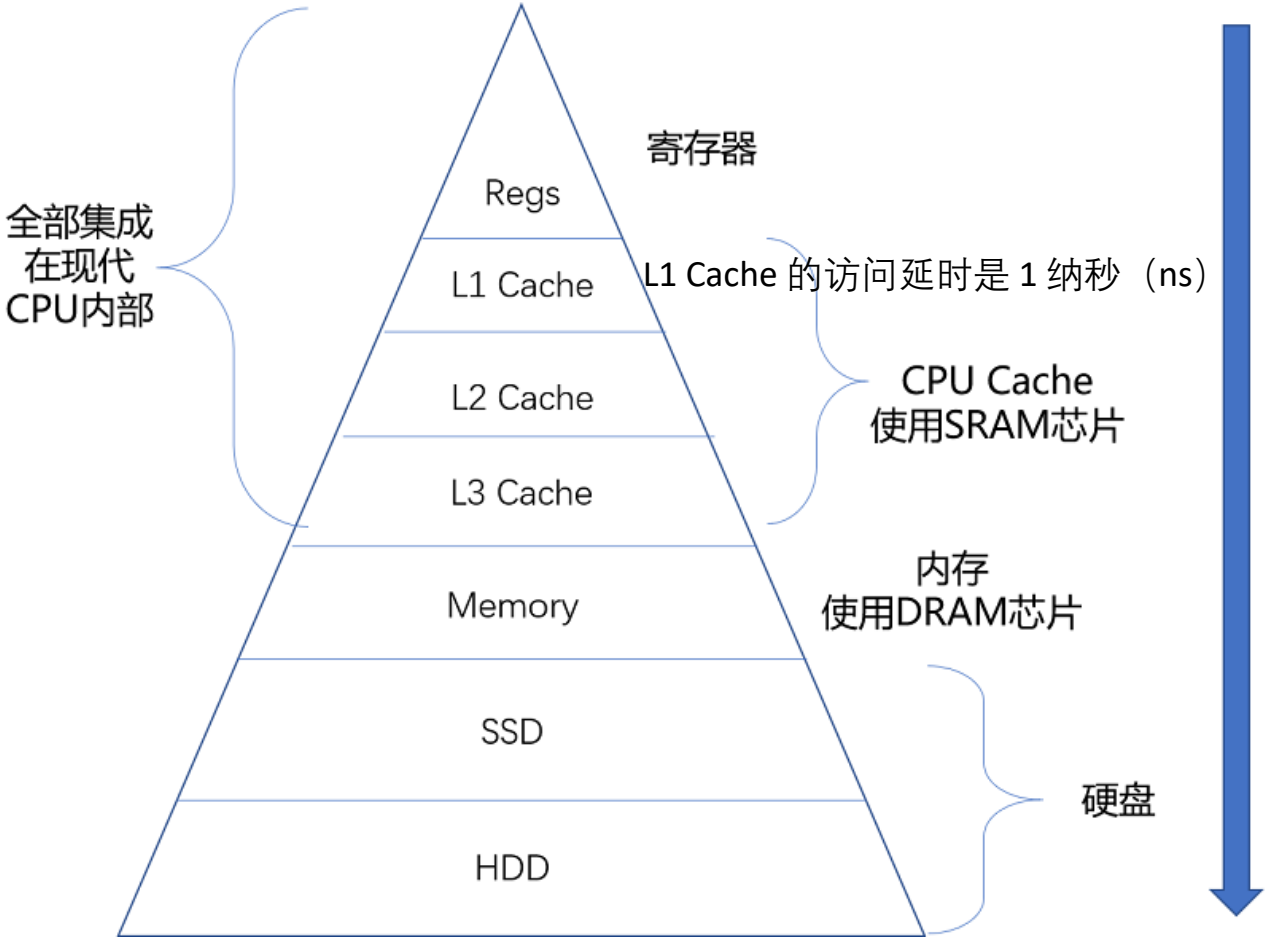
大脑 ~= CPU
正在处理 ~= 寄存器
短期记忆 ~= L1 Cache
长期记忆 ~= L2/L3 Cache

书/资料 ~= 数据
书桌/书房 ~= 内存

图书馆 ~= SSD/HDD硬盘



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

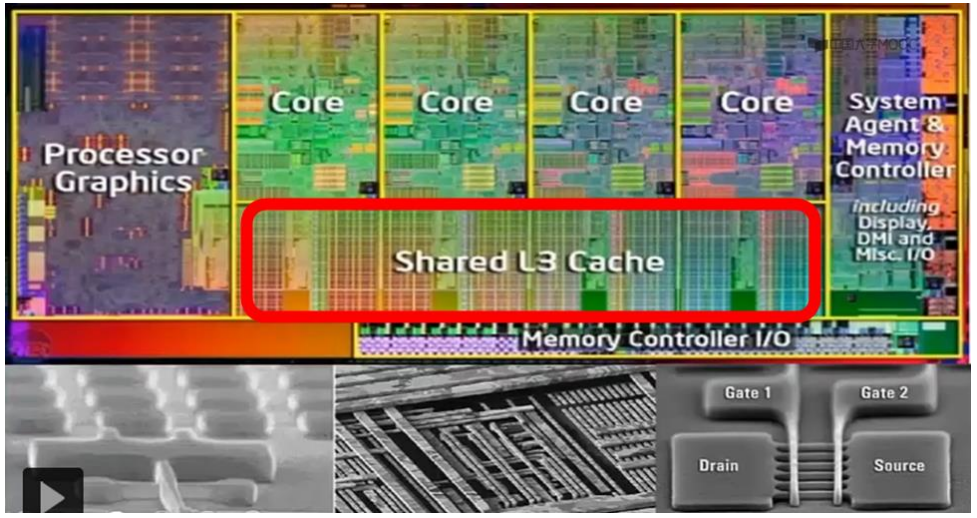
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Intle 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



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?

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

02

局部性原理 Principle of Locality

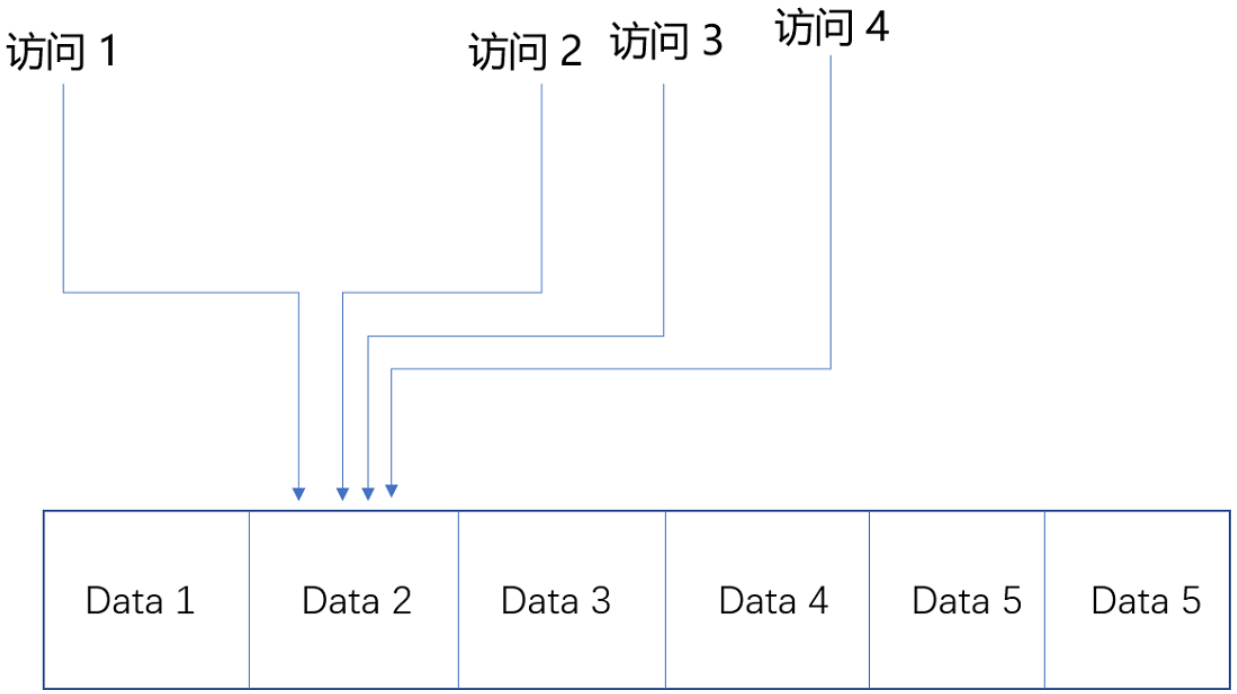
利用局部性原理，来制定管理和访问数据的策略：

- 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）

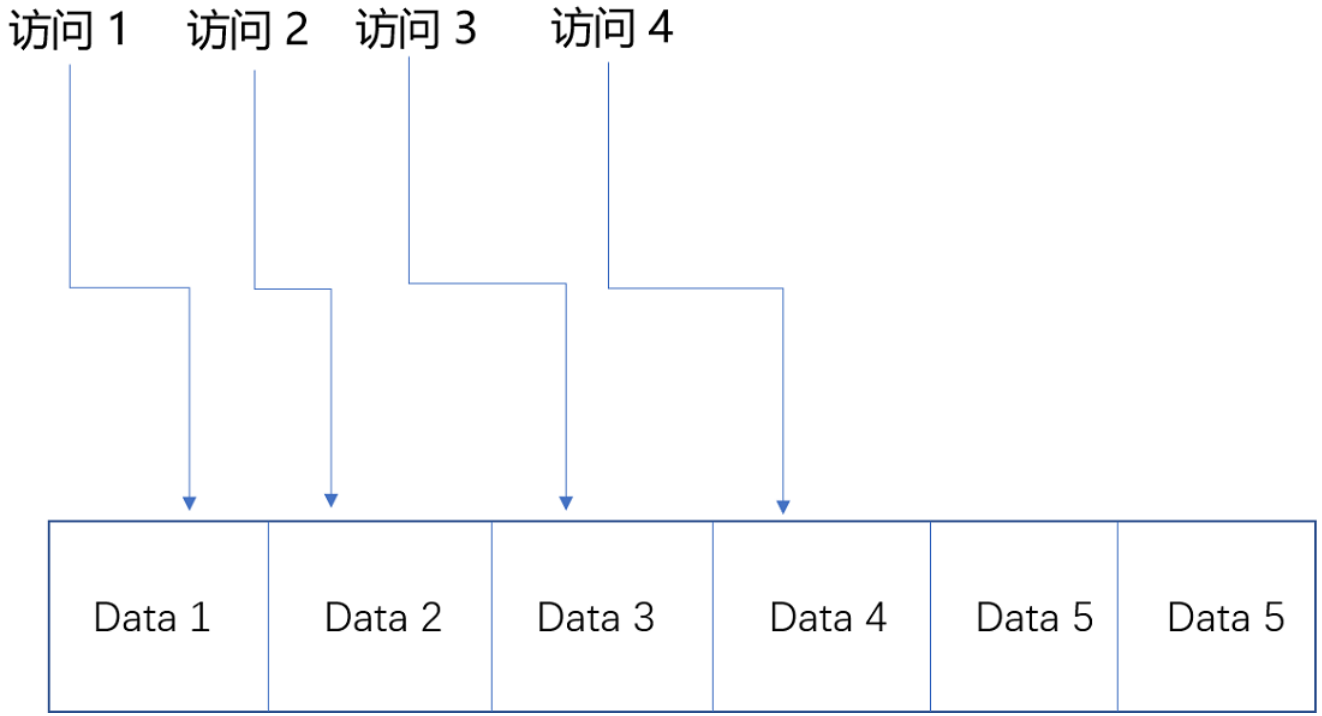
  如果一个数据被访问了，那么它在短时间内还会被再次访问



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。



- 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。

时间局部性

类似亚马逊这样的电商网站。
我们假设里面有 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 亿 × 4MB）的数据存储。

如果把数据都放在内存里面，需要 3600 万美元（ = 2400TB/1MB × 0.015 美元 = **3600 万美元**）。

但是，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 万美元 × 1% + 2400TB / 1MB × 0.00004 美元），是原来成本的 1.3% 左右。

时间局部性

把有用户访问过的数据，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）

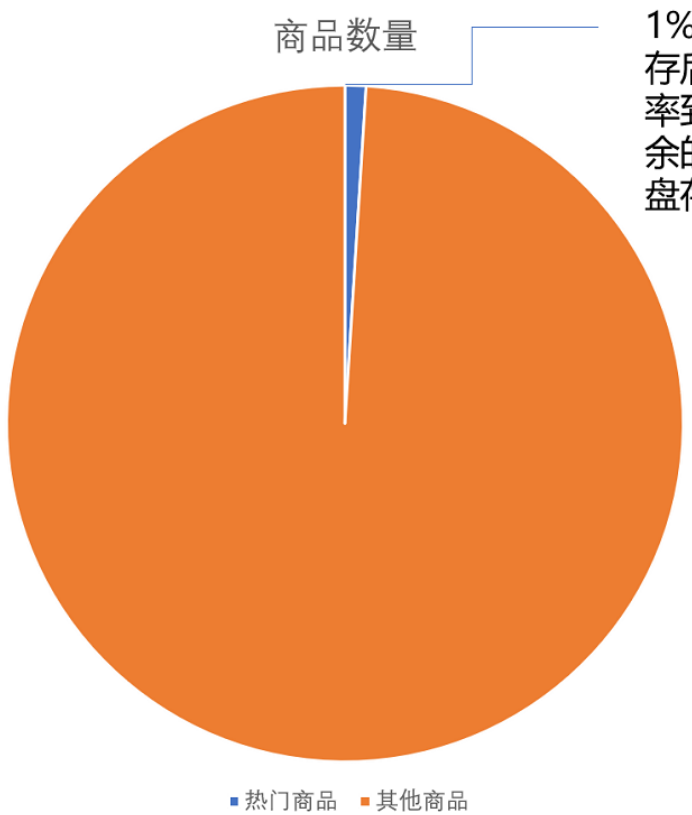
内存的随机访问请求需要 100ns，访问一次内存需要100ns, 那么**1秒**可以访问1s/100ns=10,000,000次
在极限情况下，内存可以支持 **1秒**1000 万次随机访问。

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（ = 24TB/8GB × 1s/100ns）访问。

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（ = 2400TB/4TB × 1s/10ms ）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

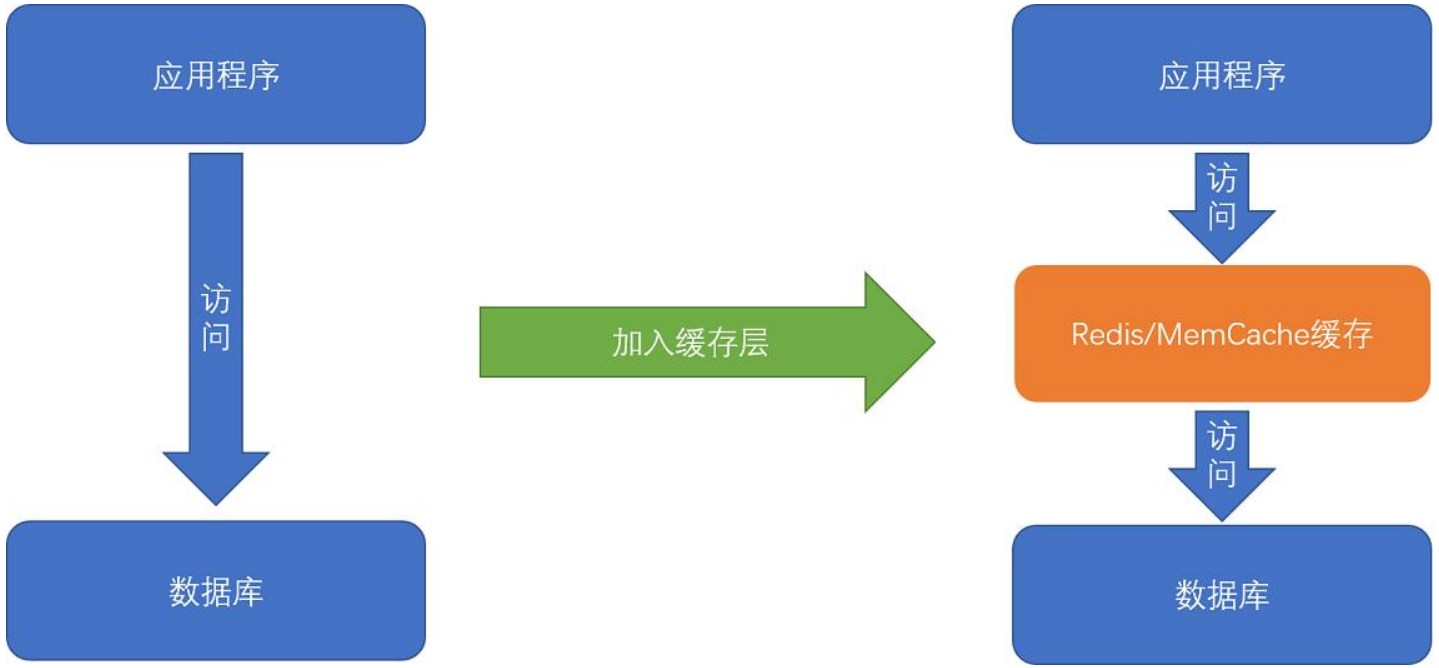


1%的热门商品放入缓存后，如果缓存命中率到达50%，那么剩余的数据使用HDD磁盘存储就够了

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止要考虑数据，还需要考虑维护数据结构的空间，而缓存的命中率和访问请求也要考虑均值和峰值的问题。

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。将热点数据加载并保留在速度更快的存储设备里面，用更低的成本来支撑服务器。

快速估算来判断这个添加缓存的策略是否能够满足我们的需求，以及在估算的服务器负载的情况下，需要规划多少硬件设备。“估算 + 规划”的能力，是每一个期望成长为架构师的工程师，必须掌握的能力。



通过使用 Redis 或者 Memcache 这样的开源软件，在数据库前面提供一层缓存的数据，来缓解数据库面临的压力，提升服务端的程序性能。