Datawhale

Datawhale 开源社区

Datawhale open source community

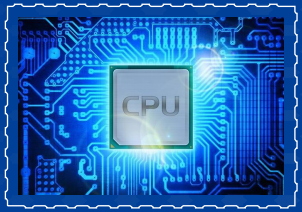
深入理解计算机系统（12）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 12 月 18 日



第 I 部分

存储器的层次结构-I

存储器的层次结构

Datawhale

存储器系统（memory system）是由不同容量、成本和访问时间的存储设备组成的层次结构。在这个层次结构中：CPU 寄存器保存最常用的数据。靠近 CPU 的小的、快速的高速缓存存储器作为相对慢速的主存储器中数据和指令的缓冲区域。主存又作为容量较大、速度较慢的磁盘上数据的缓冲区域。而磁盘又可以作为通过网络中其他机器上数据的缓冲区域。

存储器的层次结构

Datawhale

存储器层次结构对应用程序的性能有者巨大的影响。假如数据存储在寄存器中，那么 0 个时钟周期内就能访问到它们；如果存储在高速缓存中，则需要 $4 \sim 75$ 个周期；如果存储在主存中，需要上百个周期；如果存储在磁盘上，则需要约几千万个周期。

之所以会有这样的差异，是由于不同存储器中使用的存储技术不同。基本的存储技术包括 SRAM 存储器、DRAM 存储器、ROM 存储器、旋转硬盘以及固态硬盘。

随机访问存储器

Datawhale

随机访问存储器（Random-Access Memory, RAM）分为两类：静态的动态的。静态 RAM（SRAM）比动态 RAM（DRAM）更快，但也更贵。SRAM 常用来作为高速缓存存储器。DRAM 常用来作为主存以及图形系统的帧缓冲区。

静态 RAM（SRAM）

Datawhale

SRAM 是存储技术中随机访问存储器（Random-Access Memory, RAM）的一种。SRAM 将每个位存储在一个双稳态（bistable）存储器单元中，其中每个单元用一个六晶体管电路来实现。所谓双稳态，指的是该存储器单元可以无限期地保持在两个不同的电压配置（configuration）之一。也就是说，只要有电，SRAM 存储器单元就会永远保持它的值，即使有干扰扰乱电压，当干扰消除时电路就会恢复到稳定值。

动态 RAM（DRAM）

Datawhale

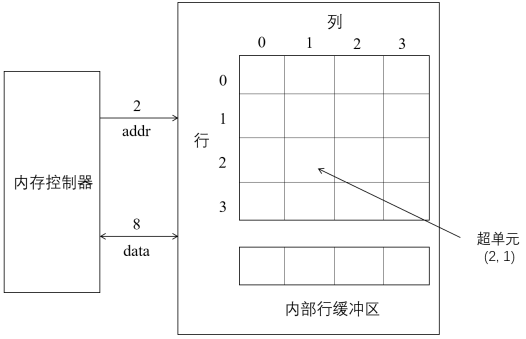
DRAM 将每个位存储为对一个电容的充电，存储器单元由一个电容和一个访问晶体管组成。与 SRAM 不同，DRAM 存储器单元对干扰非常敏感，且受到干扰后永远不会恢复。

有很多原因会导致 DRAM 单元漏电，使得它在 $10 \sim 100$ 毫秒内失去电荷。但由于计算机运行的时钟周期是以纳秒来衡量的，因此相对来说 DRAM 单元电荷保持的时间还是比较长的。由于这个原因，内存系统会周期性地通过读出后重写来刷新内存中的每一位。有些系统还使用纠错码来发现并纠正一个字中任何单个地错误位。

传统 DRAM

Datawhale

DRAM 存储器分为许多种类。在传统的 DRAM 中，DRAM 芯片中的单元（位）被分成 d 个超单元（supercell），每个超单元由 w 个 DRAM 单元组成。所有的超单元被组织成一个 $r \times c$ 的长方形阵列（$rc = d$）。每个超单元有形如 (i, j) 的地址，其中 i 表示行，j 表示列。



传统 DRAM

前面的图中展示的是一个 16×8 的 DRAM 芯片的组织，有 $d = 16$ 个超单元，每个超单元有 $w = 8$ 位，$r = 4$ 行，$c = 4$ 列。信息通过引脚（pin）的外部连接器流入和流出芯片。每个引脚携带一个 1 位的信号。图中展示了两组引脚：8 个 data 引脚，用于传送一个字节到芯片或从芯片传出一个字节；2 个 addr 引脚，用于携带 2 位的行和列单元地址。

每个 DRAM 芯片被连接到某个称为内存控制器（memory controller）的电路上，这个电路可以一次传送 w 位到每个 DRAM 芯片或一次从每个 DRAM 芯片传出 w 位。如果要读出超单元 (i, j) 的内容，内存控制器首先将行地址 i 发送到 DRAM，然后是列地址 j。DRAM 把超单元 (i, j) 的内容发回给控制器作为响应。行地址 i 称为 RAS（Row Access Strobe，行访问选通脉冲）请求。列地址 j 称为 CAS（Column Access Strobe，列访问选通脉冲）请求。

传统 DRAM

Datawhale

例如要从前图中 16×8 的 DRAM 中读出超单元 $(2,1)$，内存控制器发送行地址 2，随后 DRAM 将地址为 2 的整行都复制到一个内部行缓冲区中。接下来内存控制器发送列地址 1，DRAM 就将列地址为 1 的 8 位数据发送到内存控制器。

之所以需要将行地址和列地址分别传送，是因为 DRAM 是以二维阵列的形式组织的，这样做是为了减少引脚的数量。例如如果上面的 128 位 DRAM 被组织成 16 个超单元的线性数组，地址为 $0 \sim 15$，那么芯片将会需要 4 个引脚而不是 2 个。

内存模块

在实际的计算机体系结构中，DRAM 芯片被封装在内存模块（memory module）中并插到主板的扩展槽上。内存模块的基本思想是使用多个 DRAM 芯片，每个芯片中的每个超单元存储主存中的一个字节，并用相应超单元地址为 (i, j) 的 8 个超单元来表示主存中字节地址 A 处的 64 位字。

例如要取出内存地址 A 处的一个字，内存控制器首先将 A 转换成一个超单元地址 (i, j)，并将它发送到内存模块，内存模块再将 i 和 j 广播到每个 DRAM。作为响应，每个 DRAM 输出它的 (i, j) 超单元的 8 位内容。模块中的电路收集这些输出，并把它们合并成一个 64 位字，返回给内存控制器。

磁盘存储

前面介绍的随机访问存储器必须在有电的时候才能存取数据，而磁盘则能够永久存储大量的数据。不过，从磁盘上读信息的时间为毫秒级，比从 DRAM 读数据慢了 10 万倍，比从 SRAM 读数据慢 100 万倍。

磁盘构造

Datawhale

磁盘由盘片（platter）和可以旋转的主轴（spindle）构成。每个盘片都有两个表面，上面覆盖着磁性记录材料。主轴使盘片能以固定的旋转速率旋转，通常为 $5400 \sim 15000$ 转每分钟（“转每分钟”也作为单位，RPM）。

磁盘盘片的表面由一组磁道（track）的同心圆组成。每条磁道由存储数据的扇区（sector）和标识扇区的间隙（gap）构成。其中每个扇区包含相等的数据位（通常为 512 字节），而间隙仅用于标识扇区的格式化位，并不存储数据。

磁盘由一个或多个叠放的盘片组成，并被封装在一个密封的包装里。整个装置称为磁盘驱动器（disk drive），简称为磁盘。通常磁盘又被称为旋转硬盘（rotating disk）或机械硬盘。

磁盘容量

Datawhale

磁盘容量主要由以下因素决定：

1. 记录密度（recording density）（位/英寸）：磁道一英寸的段中可以放入的位数。
2. 磁道密度（track density）（道/英寸）：从盘片中心出发半径上一英寸的段内可以有的磁道数。
3. 面密度（areal density）（位/平方英寸）：记录密度与磁道密度的乘积。

磁盘操作

磁盘通过读写头（read/write head）来读写存储在磁性表面的位，而读写头连接到一个传动臂（actutaor arm）一端。通过沿着半径方向移动传动臂，驱动器可以将读写头定位在盘片的任何磁道上。在这个过程中读写头垂直排列，一致行动。

磁盘操作

Datawhale

磁盘读写数据的时间主要受以下因素影响：

1. 寻道时间：读写数据时，传动臂首先需要将读写头定位到包含目标扇区的磁道上，这个移动过程所花的时间称为寻道时间。寻道时间主要依赖于读写头的位置和传动臂在盘面上移动的速度。经过测量，现代驱动器中的平均寻道时间通常为 3 ～ 9 ms。一次寻道的最大时间可高达 20 ms。
2. 旋转时间：当读写头移动到了目标磁道上，就需要等待目标扇区的第一个位旋转到读写头下。这个过程所花的时间称为旋转时间。旋转时间主要依赖于读写头到达目标磁道时目标扇区的位置以及磁盘的旋转速度。最坏的情况下读写头必须等待磁盘旋转一圈。
3. 传送时间：当目标扇区的第一个位位于读写头下时，驱动器开始读或写该扇区的内容。一个扇区的传送时间依赖于旋转速度和每条磁道的扇区数目。

逻辑磁盘块

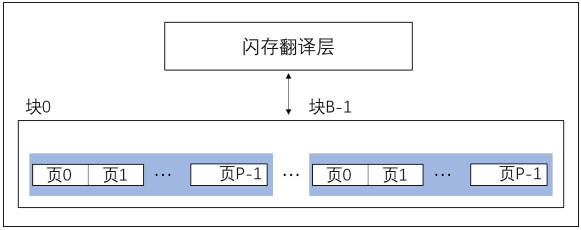
Datawhale

为了对操作系统隐藏磁盘构造的复杂性，在现代磁盘的架构中，通常将盘面上的各个记录区呈现位一个 B 个扇区大小的逻辑块序列，编号为 $0, 1, \ldots, B-1$ 并使用磁盘控制器来维护逻辑块号和实际磁盘扇区之间的映射关系。

当操作系统需要执行一个读写操作，例如读一个磁盘扇区的数据到主存时，操作系统会将命令发送到磁盘控制器，让它读某个逻辑块号。控制器上的固件执行一个快速表查找，将一个逻辑块号翻译成一个三元组：(盘面，磁道，扇区)，这个三元组唯一地表示了对应的物理扇区。随后控制器上的硬件会解释这个三元组，将读写头移动到适当的柱面，等待扇区移动到读写头下，随后将读写头感知到的位放到控制器上的一个小缓冲区中，然后将它们复制到主存中。

固态硬盘

和磁盘不同，固态硬盘（Solid State Disk, SSD）是一种基于闪存的存储技术。一个 SSD 由一个或多个闪存芯片以及闪存翻译层组成。闪存芯片取代了磁盘中机械驱动器。闪存翻译层则扮演与磁盘控制器相同的角色，主要将操作系统对逻辑块的请求翻译成对物理设备的访问。



固态硬盘

Datawhale

在 SSD 中，一个闪存由 B 个块组成，每个块又由 P 页组成。通常，页的大小为 512 字节 $\sim$ 4KB，块由 $32 \sim 128$ 页组成，块的大小为 16 KB ~ 512 KB。

数据在 SSD 中是以页为单位读写的。在写入数据之前，需要对一页所属的块进行擦除操作。所谓擦除，指的是将该块中所有位置为 1。而写入的操作实际上就是将部分位置为 0。

当擦除次数过多后，块会因磨损而损坏。为了减少这种磨损带来的影响，闪存翻译层使用一种平均磨损算法，来将擦除平均到各个块上。

局部性

所谓局部性，指的是在每次引用数据时倾向于引用最近引用过的数据项邻近的数据项，或最近引用过的数据项本身。一个编写良好的程序通常要求具有好的局部性。

局部性分为**时间局部性**和**空间局部性**。良好的时间局部性指的是被引用过一次的数据项会在不远的将来再次被多次引用；良好的空间局部性指的是，如果某个数据在某个位置被引用了一次，那么在不远的将来将引用它附近的内存位置。

局部性举例

在下面的代码中，变量 sum 在每次循环中都被引用一次，因此对于 sum 来说，该程序具有好的时间局部性；对于向量 v 来说，v 中的元素被一个接一个地读取，因此该程序有好的空间局部性。综上，我们说该函数有良好的局部性。

sumvec

```
1 int sumvec(int v[N]) {
2     int i, sum = 0;
3     for (i = 0; i < N; i++) sum += v[i];
4     return sum;
5 }
```

局部性举例

Datawhale

在下面的这个例子中，双重嵌套循环按**行优先顺序**读数组中的元素，由于二维数组也是按照行优先顺序存储的，因此该函数页具有良好的空间局部性。但由于数组中的每一个元素都只被使用了一次，因此该函数不具备良好的时间局部性。

sumarrayrows

```
1 int sumarrayrows(int a[M][N]) {
2     int i, j, sum = 0;
3     for (i = 0; i < M; i++)
4         for (j = 0; j < N; j++)
5             sum += a[i][j];
6     return sum;
7 }
```

局部性举例

Datawhale

如果我们将上面例子中的程序改变一下，将嵌套循环读取数组的方式变为**列优先顺序**，这时候每引用一次数据，系统就要读取一列，空间局部性就变得很差了。

sumarrayrows

```
int sumarrayrows(int a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; i < N; j++)
        for (i = 0; i < N; i++)
            sum += a[i][j];
    return sum;
}
```

步长对局部性的影响

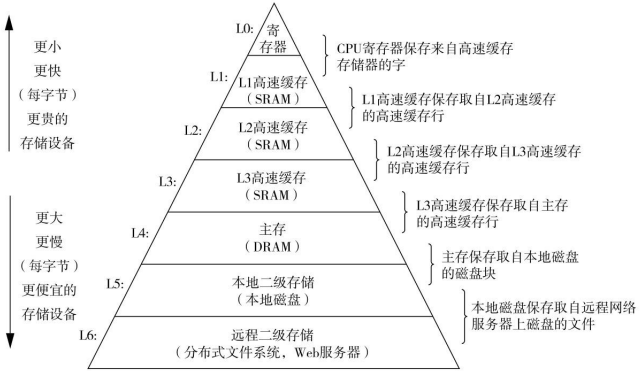
在我们的前两个例子中，程序都是按一个接一个的顺序访问数组，我们称这种访问顺序为“步长为 1 的引用模式”，或称为“顺序引用模式”。在一些循环中，会对一个连续向量每隔 k 个元素进行访问，这时称其为“步长为 k 的引用模式”。

容易看出，步长为 1 的引用模式时空间局部性较好的引用模式。一般来说，步长越大，空间局部性就越差。例如在最后一个例子中，虽然看上去也是一个接一个地引用数组中的元素，但由于引用顺序没有按照数组的行优先存储顺序来引用，所以这实际上是个步长为 N 的引用模式。

存储器的层次结构

Datawhale

为了将不同存储器存储技术之间的差异性与计算机软件对局部性的要求这两者结合互补，人们构造了一种存储器层次结构。如图是一个经典的存储器层次结构：



存储器层次结构中的缓存

Datawhale

一般而言，高速缓存（cache）是作为更大、更慢的存储设备的一个缓冲区域，它相对而言更小、更快速。存储器层次结构的中心思想是，对于每个 k，位于 k 层的更快更小的存储设备作为位于 $k+1$ 层更大更慢的存储设备的缓存。

在存储器层次结构中，第 $k+1$ 层的存储器被划分成连续的数据对象组块（chunk），称为块（block）。每个块都有一个唯一的地址或名字。通常块的大小是固定的，但也可能是可变的（例如存储在 Web 服务器上的远程 HTML 文件）。在第 k 层中，存储器被划分成较少的块的集合，其中每个块的大小与 $k+1$ 层中的块的大小相同。这样，数据就以块大小为传送单元在第 k 层和 $k+1$ 层之间来回复制。

缓存命中

Datawhale

当程序需要第 $k+1$ 层的某个数据对象 d 时，它首先在当前存储在第 k 层的一个块中查找 d。如果 d 刚好缓存在第 k 层中，那就称为**缓存命中**，这时程序直接从第 k 层读取 d。由于第 k 层相对于 $k+1$ 层来说是更块的存储设备，因此读取速度比从第 $k+1$ 层读取更快。

缓存不命中

Datawhale

如果程序在第 k 层中找不到缓存数据对象 d，那么就是**缓存不命中**。当缓存不命中时，就需要从第 k 层中取出包含 d 的那个块，此时如果 k 层的缓存已经满了，那么就可能会覆盖现存的一个块。

覆盖一个现存块的过程称为替换或驱逐这个块。决定该替换哪个块是由缓存的替换策略（replacement policy）控制的。例如，随机替换策略会随机替换一个块；而 LRU 替换策略会选择替换最后被访问的时间距现在最远的块。

缓存不命中的种类

Datawhale

如果一个缓存是空的，那就称它为**冷缓存（cold cache）**。冷缓存对任何数据对象的访问都会不命中，这类不命中称为**强制性不命中（compulsory miss）**或**冷不命中（cold miss）**。冷不命中通常是短暂的事件，不会在存储器的稳定状态中出现。

如果发生了不命中，那么第 k 层的缓存就必须执行某个**放置策略（placement policy）**，来确定把从第 $k+1$ 层中取出来的块放在哪里。最灵活的替换策略是允许来自第 $k+1$ 层的任何块放在第 k 层的任何块中。

缓存不命中的种类

实际上，靠近 CPU 的缓存是通过硬件实现的，实现随机放置块的代价很高，因此硬件缓存通常使用更严格的放置策略。这个策略将第 $k+1$ 层的某个块限制放置在第 k 层块的一个小的子集中（有时只是一个块）。例如可以限制第 $k+1$ 层的块 i 必须放置在第 k 层的块 $(i \ mod \ 4)$ 中。这样第 $k+1$ 层的块 0、4、8 和 12 都会被映射到第 k 层的块 0 中。

使用这种放置策略可能会造成**冲突不命中（conflict miss）**。在这种情况下，部分对象会被映射到同一个缓存块，导致缓存一直不命中。

高速缓存存储器

Datawhale

一直以来，CPU 和主存之间的访存差距逐渐增大，因此系统设计者被迫在 CPU 寄存器文件和主存之间插入高速缓存器。至今 CPU 寄存器文件和主存之间已经有了三个高速缓存器。为了讨论简单，我们接下来假设 CPU 和主存之间只有一个高速缓存。

通用高速缓存存储器组织结构

考虑一个计算机系统，其中每个存储器地址有 m 位，形成 $M = 2^m$ 个不同的地址。这样的一个高速缓存器被组织成一个有 $S = 2^s$ 个高速缓存组的数组。每个组包含 E 个高速缓存行。每个行由一个 $B = 2^b$ 字节的数据块组成。其中还包括有一个有效位，用于指明这个行是否包含有意义的信息。另外有 $t = m - (b + s)$ 个标记位用于唯一地表示存储在这个高速缓存行中的块。

一般而言，高速缓存结构可以用一个四元组 (S, E, B, m) 来描述。高速缓存的大小 C 指的是所有块的大小的和，标记位和有效位不包括在内，因此 $C = S \times E \times B$。

通用高速缓存存储器组织结构

当一条加载指令指示 CPU 从主存地址 A 中读一个字时，它将地址 A 发送到高速缓存。如果高速缓存正保存着地址 A 处那个字的副本，它就立即将那个字发回给 CPU。可以发现，在这个过程中，高速缓存需要知道它自己是否包含地址 A 处那个字的副本。

通用高速缓存存储器组织结构

Datawhale

实际上，高速缓存是通过检查地址位来寻找所请求的字的。在高速缓存结构元组中，S 和 B 将 m 个地址分为了标记、组索引和块偏移三个字段。地址 A 中 s 个组索引位是一个到 S 个组的数组的索引，是一个无符号整数。组索引位告诉我们这个字必须存储在哪个组中。

一旦我们知道了这个字要被放在哪个组中，A 中 t 个标记位就告诉我们这个组中的哪一行包含这个字。当且仅当设置了有效位并且该行的标记位与地址 A 中的标记位相匹配时，组中的这一行才包含这个字。一旦我们在由组索引标识的组中定位了由标号所标识的行，那么 b 个块偏移位给出了在 B 个字节的数据块中的字偏移。

直接映射高速缓存

根据每个组的高速缓存行数 E，高速缓存被分为不同的类。每个组只有一行的高速缓存被称为直接映射高速缓存。

当 CPU 执行一条读内存字 w 的指令时，它向 L1 高速缓存请求这个字。如果 L1 中有 w 的一个缓存的副本，那么缓存命中，L1 将抽取出 w 并将它返回给 CPU。否则缓存不命中，这时 L1 就要向主存请求包含 w 的块的副本。当被请求的块到达时，L1 就将这个块存放在它的一个高速缓存行里，从被存储的块中抽取出字 w，然后将它返回给 CPU。

直接映射高速缓存

在上述过程中，高速缓存的操作分为三步：1）组选择；2）行匹配；3）字抽取。其中组选择较为简单，高速缓存从 w 的地址中抽取出 s 个组索引位。这些位被解释成一个对于于一个组号的无符号整数。

假设已经确定了某个组 i 接下来要确定是否有字 w 的一个副本存储在某个组 i 包含的一个高速缓存行中。由于在直接映射高速缓存中，每个组只有一行。因此当且仅当设置了有效位，而且高速缓存行中的标记 w 与地址中的标记相匹配时，这一行中包含 w 的一个副本。

接下来我们要确定所需要的字在块中是从哪里开始的，这通过块偏移位就可看出。例如块偏移位是 100_2，则表明 w 的副本是从块的字节 4 开始的（假设字长为 4 字节）。

直接映射高速缓存

Datawhale

如果发生了缓存不命中，那么就需要从存储器层次结构的下一层中取出所请求的块并存储在组索引位指示的组中的一个高速缓存行中。如果组中都是有效高速缓存行，那么就用新取出的行替换当前的行。

组相联高速缓存

Datawhale

在组相联高速缓存中，每个组都有多于一个的高速缓存行。通常称 $1 < E < C/B$ 的高速缓存为 E 路组相联高速缓存。

在组相联高速缓存中，组选择的过程与直接映射高速缓存的组选择一样，而行匹配则更复杂些。实际上，我们可以把组相联高速缓存中的每个组都看成一个小的相联存储器。每一组标记位和有效位就对应了一个块的内容。因此相联高速缓存中行匹配的基本思想就是寻找一个有效的行，其标记与地址中的标记相匹配。

组相联高速缓存

当组相联高速缓存中发生不命中时，需要选择行替换策略。常见的行替换策略有最不常使用（Least-Frequently-Used, LFU）策略和最近最少使用（Least Recently-Used, LRU）策略。

全相联高速缓存

全相联高速缓存由一个包含所有高速缓存行的组（$E = C/B$）组成。在全相联高速缓存中，由于只有一个组，不需要进行组选择。此外行匹配和字选择与组相联高速缓存都是一样的。

有关写的问题

如果某个高速缓存更新了某一个字 w 的副本，那么还需要更新 w 在第一层中的副本。最简单的方法是**直写（write-through）**，即立即将 w 写回到低一层中。这样做的缺点是每次都会引起总线流量。

另一中方法是**写回（write-back）**，即尽可能地推迟更新，直到替换算法要驱逐这个更新过的块时再写到低一层中。这样做的缺点是增加了复杂性，因为需要知道某个高速缓存块是否被修改过。

有关写的问题

在写的时候也会出现**写不命中**的问题。一种方法是**写分配**，即加载相应的低一层中的块到高速缓存中，然后更新这个高速缓存块。另一中方法是**非写分配**，即直接把这个字写到低一层中。直写高速缓存通常是非写分配的，写回高速缓存通常是写分配的。