链滴

计算机中存储体系的设计

作者：vcjmhg
原文链接：https://ld246.com/article/1588142182996
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20181013.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="概述">概述</h2>
在计算机运行过程中,存储器是各种信息存储和交换的中心，而计算机所有存储器所构成的存储系
更是整个计算机系统的核心组成部分。在一台计算机中通常有多个存储器：<code>主存储器</code
、<code>Cache</code>、通 <code>用寄存器</code>、<code>磁盘寄存器</code>、<code
各种缓冲存储器</code>、<code>光盘存储器</code> 等。
为了评定不同存储器的性能差异，人们制定了一些主要的性能指标:速度，容量和价格

其中速度我们用存储器的访问周期、读出时间、频带宽度等来进行表示。<s
rong>容量用字节 B、千字节 KB、兆字节 MB 和千兆字节 GB 等表示。价格</s
rong>则是用单位容量的价钱表示，例如 <code>$C/bit</code>。
讲了这么多存储器的内容，那什么叫做 <code>存储体系</code> 哪？
下边我们引入存储体系的定义：
<blockquote>
两个或两个以上速度、容量和价格各不相同的存储器用硬件、软件、或软件与硬件相结合的方法
接建立起来的一个系统。
</blockquote>
简单来说，就是一个将不同类型的存储器用软件或者硬件方法结合成一个整体。该系统对程序员
明，并且，从应用程序员来看，它仅仅是一个存储器。这个存储器的速度接近速度最快的
个存储器，存储容量与容量最大的那个存储器相等，单位容量的<str
ng>价格接近最便宜的那个存储器。
为了更加便于理解存储体系速度、容量和价格的关系，我们画出下图用更加严谨的方式来进行表
。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-edd9ae4f.png?imageView2/2/interlace/1/format/jpg"></
>
符号说明
M_i:i\text{号存储器}
T_i\text{：}i\text{存储器的速度性能}
S_i\text{：}i\text{号存储器的容量性能}
C_i\text{：}i\text{号存储的价格}
各个存储器的各个性能指标和其构成的存储体系的性能指标之间的关系为：
T≈min（T1，T2，……，Tn），用存储周期表示
S＝max（S1，S2，……，Sn），容量用MB或GB表示
C≈min（C1，C2，……，Cn），价格用每位的价格表示
<h2 id="存储体系的分类">存储体系的分类</h2>
在计算机系统中一般来说存储器类别主要有 Cache、主存储器和辅助存储器三类组成，按照不同
组合方式构成了两种存储体系：

<code>Chache存储体系</code>：由 Cache 和主存储器构成

主要目的：提高存储器速度，或者说是为了贪图 Cache 的存储速度
系统程序员看：速度接近 Cache，存储器容量等于主存，每位的价格接近主存储器

<code>虚拟存储体系</code>：由主存储器和磁盘存储器构成

主要目的：扩大存储器容量，或者说贪图磁盘存储器的容量。
应用程序员看：速度接近主存储器，存储容量是虚拟地址空间，每位价格接近磁盘存储器。

<blockquote>

看到这里可能大家会有疑问，三种存储器两两组合方式组中结果不是应该是三种吗？
或者说应该还有一种组合方式即 Cache+ 磁盘存储器？？
</blockquote>
坦白来讲此种存储体系理论上讲是可以有的，但是实际应用中，由于 Cache 和虚拟存储器的速差别太大，强行相互结合，根本不可能发挥出 Cache 存储器的速度优势，实际上还会大大拖累 Cache 的运行。因此此种组合方式在实际应用过程中根本不可能存在。

p>
<h2 id="存储效率">存储效率</h2>
关于速度的评定我们一般通过，访问周期、存取周期、存储周期、存取时间等来进行表示。
关于这些指标的计算我们详细可以参照百度百科
提到了存储器的速度，我们就提另一个指标 <code>存储效率</code>。
首先我们给出命中率的定义：
<blockquote>
在M_1存储器中达到的访问效率，给出以下公式</p

</blockquote>
=N_1/\left(N_1+N_2 \right)
其中：
 N_1 是M_1</spn>存储器访问到的效率。
N_2是对M_2</san>存储器的访问次数
进而我们引出存储系统的访问效率：
e=\frac{T1}{T}=\frac{T1}{H\cdot T1+\left(1-H \right) \cdot T2}=frac{1}{H+\left(1-H \right) \cdot \frac{T2}{T1}}=f\left(H,\frac{T2}{T1} \right)
从上边的公式中我们可以得出提高存储系统的两条途径：

提高命中率 H
两个存储器的速度差别不应太大

在实际操作中有时第二条途径做不多（如虚拟存储器），因此，在实际操作过程中我们通过<strog>提高命中率来提高存取效率。
<blockquote>
那此处可能就又有一个问题：如何提高命中率？
</blockquote>
我们一般采用 <code>预取技术</code> 来提高命中率
措施：在未命中时，把M_2存储器中相邻几个单元成的一个数据块都取出来送入到M_1存储其中。
在引入 <code>预取技术</code> 之后我们可以的出新的命中率公式：
H^{\prime}=\frac{H+n-1}{n}
其中：
H^{\prime}是采用预取技术之后的命中率
H是原来的命中率
n是数据块大小与数据重复使用次数的乘积。
为了更加深入的理解命中率我们做下边一个例题
<blockquote>
例：在一个 Cache 存储系统中，当 Cache 的块大小为一个字时，命中率为 H＝0.8；假设数据重复利用率为 5，Cache 的块大小为 4 个字

计算 Cache 存储系统的命中率是多少？
假设T_2=5T_1，分别计算访问效率。

</blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-f37f33ae.png?imageView2/2/interlace/1/format/jpg"></>
<h2 id="存储体系原理">存储体系原理</h2>
存储体系具体操作过程中可能会遇到容量不足或者速度不够</strng>的问题，下边我们分成两个专题分别对其进行讨论和解决。
<h2 id="容量不足的问题">容量不足的问题</h2>
第一种方案：增加主存容量
解决容量不足的问题，我们最容易想到的便是增加主存容量。正如缺啥补，缺少容量我们增加主存容量必然可以在一定程度上解决容量不足的问题，但是提到任何一种方案，考虑经济因素便是耍流氓。我们要知道主存的价格并不是非常便宜，随着主存容量的增加，存储器价总量必然增加。从而导致单位容量存储器价格增加。显然这不是最终的解决的方案。
第二种方案：采用两级存储器
既然仅仅增加主存容量我们不能根本上解决该问题，因此我们也提出了第二种方案来进行补充--主存和辅存结合在一起采用两级存储结构，利用低价格的辅存来扩充存储容量。/p>
当然能够用辅存扩充容量且能达到主存的速度，其背后是有如下依据的：
首先计算机中所有的信息可以分为如下三类：

活跃的信息，即当前正在使用的信息
待命的信息，将要使用的信息
静止的信息，已被使用而不再处理的信息

因此我们按照不同的使用频率，可以将不同的信息放置到不同的存储器中。可将活跃的信息和部待命的信息放在主存中，其余部分放到辅存，以减少主存容量的要求，从而实现扩充容量的同时又不显著的增加成本。
由于信息本身的使用频率不是固定的，或者说同一个信息在不同的时间段可能被分为不同的类别因此在使用两级存储之后，主辅存之间的信息调出和调入可能会有一些问题。
程序的调入和调出由程序员考虑和安排，一定程度上增加了程序员的负担。因此使用辅助机构自定位，从而引出了虚拟存储器。
可以将高速辅存（如磁盘）伪装成主存访问，信息在主存、辅存之间的调进和调出完全由辅助机自动完成。
<h2 id="速度不足的问题">速度不足的问题</h2>
存储器的速度往往是整个计算机速度的第一个瓶颈。
方法一，同样也是从主存入手，提高主存的速度。此法也在采用，但此法随存储器速度的提高成也会显著增加。
方法二也是引入二级存储结构，只不过引入的是 <code>Cache+主存</code> 此体系的结构如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-6605ec56.png?imageView2/2/interlace/1/format/jpg"></>
让 CPU 直接与它速度匹配的 Cache 访问，此法也需要 CPU 和 cache 之间利用辅助机构来完成ache 与主存之间信息的调入与调出。
<h2 id="辅助机构的功能">辅助机构的功能</h2>
<blockquote>
上边解决两个问题都需要用到辅助机构，那么我们到此处可能会有疑问了，那辅助机构到底有什功能那？
</blockquote>
首先，第一个是地址映像功能：

解决M_2 中的信息采用何种规则调入到<span class
"language-math">M_1中（即调入规则的问题）。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-c82a7172.png?imageView2/2/interlace/1/format/jpg"></
>
第二个功能叫做地址变换功能：
根据映像规则，将包括M_2在内的大空间的地址转
成 CPU 能够直接访问的M_1中的地址（即地址变换问题
。
第三个功能是进行页面替换：
在M_1中装满信息的条件下，采用何种算法，算出
用M_1的部分信息，使<span class="language-math"
M_2中的部分功能调到M_1中（即替换算法问
）。
关于这三部分功能下边我们会分成三个小专题分别来进行讲述。
<h2 id="存储体系中的页式管理">存储体系中的页式管理</h2>
因为存储器本身的物理结构并不适合信息的存储于组织，因此我们在对存储器使用之前需要对其
象出一层逻辑结构来进行信息组织，因此此处我们引入 <code>页式管理</code> 的概念。
<h2 id="页的定义">页的定义</h2>
首先我们要明白 <code>页式管理</code> 中"页"是何含义。
<code>页式管理</code> 中将虚拟存储空间和实际存储空间</
trong>等分成固定大小的页，使虚拟页可装入主存中不同的实际页面位置。</
>
<h2 id="页式管理中的地址表示">页式管理中的地址表示</h2>
<code>页式管理</code> 将物理的存储结构进行一层抽象，引入了“页”来进行管理，从而
得对存储地址的访问也需要某种新的地址表示。

虚地址（逻辑地址，程序地址）：包括M_2在内的
空间地址。

<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-97cdace5.png?imageView2/2/interlace/1/format/jpg"></
>
其中，
N_v：虚页号
N_r:页内地址
<ol start="2">
实地址（物理地址）：为 CPU 能直接访问的M_1中
地址。

<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-37d114e4.png?imageView2/2/interlace/1/format/jpg"></
>
其中：
n_v:实页号
n_r:页内地址
当然引用两种地址表示不是目的，为了解决 <code>虚地址</code> 和 <code>实地址</code
，因此下边我们引入了 <code>页表</code> 的概念。
<blockquote>
页表是一种特殊的数据结构，放在系统空间的页表区，存放逻辑页与物理页帧的对应关系。
</blockquote>
<code>页表</code> 具有以下特点：

<code>页表</code> 所需行数和虚页号数相等，虚页号与页表行号对应，因此无需虚页号字

。
<code>页表</code> 中每行内容分为两个字段：实页号n_v
/span>以及装入位(1 位，其中 0 表示该页已装入，1 表示该页未被装入)

针对上边的两个特点我们可以设计出结构如下的页表：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-0e294614.png?imageView2/2/interlace/1/format/jpg"></
>
此时可能会有同学问了，虚页号到哪里去了？？
这个很容易解释，根据第一个特点，虚页号和页表行号一致的，而页表行号实际上是从零开始逐
“+1 “的。也即我们看到的上边那个两行的页表实际上等同于下边这种三行结构的页表。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-92ee8494.png?imageView2/2/interlace/1/format/jpg"></
>
页式管理的地址变换（关键）
① 根据N_v去查页表中的某一行 m。
② 查该行的装入位。
③ 装入位=1 时，命中。表示该虚页已装入。

从该行中送出n_v（实页号）。
再将N_r直送n_r<
span>，
即完成N_vN_r →
_v n_r。

④ 装入位=0 时，失效，表示该虚页未装入M_1中。
/p>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-3fe7f96a.png?imageView2/2/interlace/1/format/jpg"></
>
为了加深理解我们做下边一个例题，
<blockquote>
某虚拟存储器共 8 个页面，每页为 1024 个字，实际主存为 4096 个字，采用页表法进行地址映
。映象表的内容如下表所示：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-50b9b0a3.png?imageView2/2/interlace/1/format/jpg"><
p>
1）列出会发生页面失效的全部虚页号；
2）按以下虚地址计算主存实地址：
0、3728、1023、1024、2055、7800、4096、6800
</blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/04/image-d23c9f87.png?imageView2/2/interlace/1/format/jpg"></
>
<h2 id="地址映象及其变换">地址映象及其变换</h2>
在上一小节，我们简单介绍了虚地址和实地址之间的转换。但实际上如果存储系统使用的是不同
址映像方式的话，可能地址变换方式可能也会有很大的不同。一般来说我们常用的地址映像方式有以
四种：

全相联
直接相联
组相联
段相联

下边我们分四个小专题分别对其进行介绍。
<h2 id="全相联映象及其变换">全相联映象及其变换</h2>
首先我们给出`全相联映象的含义：
<blockquote>
对辅存中的任何一个页面都可以放到主存中任何一个页面的映像规则，称为全相联映射。
</blockquote>
按照其定义我们给出以下示意图：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-2c822dbf.png?imageView2/2/interlace/1/format/jpg"></>
我们不难 <code>全相联映射</code> 具有如下特点：
辅存中的任何一个页面都可能映射到主存中。
<h3 id="地址变换">地址变换</h3>
地址表示的示意图如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-12a8677f.png?imageView2/2/interlace/1/format/jpg"></>
当然上边具体变换方法不只一种，除了上一章节介绍的 <code>页表法</code> 外还有一种方叫做 <code>目录表法</code>
<h3 id="全相联目录表法">全相联目录表法</h3>
该种方法要求用相联存储器作为目录表（相联存储器是一种可按照内容的特征字段来访问的一种储器，是一个小容量高速存储器）。
目录表的结构具有如下特点：

目录表的行数与主存页面数相等
目录表中每行的内容：

N_v为相联比较字段；
n_v为主存页号（非相联比较字段）。

具体变换过程如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-87c56bf6.png?imageView2/2/interlace/1/format/jpg">></>
当然为了便于理解，我们画出其详细的变换过程
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-2e293f02.png?imageView2/2/interlace/1/format/jpg">></>
具体过程如下：

将虚地址中的N_V送目录表中去进行相联比较（一个span class="language-math">t_m）。
当有某个比较器比较相等(比较的时候是比较的是N_v</span相关联的字段)时，将该行n_V送出，同时N_r→n_r，实现了N_vN_r→n_vn_r的变换命中）.
若没有相等的，不命中，等待调入。

通过以上过程我们不难分析出全相连目录表法具有如下特点：

产生页面冲突的可能性很小（因为页表的行数和主存行数相同）；

与页表放入主存中相比，查表的速度快（因为目录表是放到相联存储器中的，输入高速存储器）

当然该方法也有以下缺点：

可扩展性差（因为相联存储器容量在一定程度上限制了主存的扩展，不能太大超过相联存储器的数）
主存容量增加时，目录表的造价高，速度降低（成本原因，在工程上很常见）。

<h2 id="直接映象及其变换">直接映象及其变换</h2>
该方法现将辅存按照主存大小分为若干块，在辅存的每一块都有与主存相同的页面数，辅存每块内的页面只能调入到与主存同的页面上。
该方法与全相联映射相比我们发现，直接映象辅存中的某块具体某一页能够调度的位置是固定的或者说是固定的几个位置）。
下面我们画出直接映象的规则示意图：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2020/04/image-4e52f0e7.png?imageView2/2/interlace/1/format/jpg"></>
其中：

N_d ：块号
N_v'：块内页号

从图中我们可以看到，该方法先将辅存按主存大小分为若干块，在辅存的每块内都有与主存相同页面数，辅存每块内的页面只能调入到与主存相同的页面上（可以简单理解为一个萝卜一个坑 :smile:: mile:）
<h3 id="地址变换-">地址变换</h3>
在展开说明直接映象的地址变换之前我们先了解下直接相联请情况下，实地址和虚地是如何表示的。
由于直接映象引入了**“块”，因此我们在给定虚地址时肯定要给出块号*，然后我们通过块号（N_d）、块内页号(N_v')以及页内地址(N_r)三部来最终定位到具体的数据地址，此处我们给出示意图便于理解。
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2020/04/image-cb3a129f.png?imageView2/2/interlace/1/format/jpg"></>
实地址和硬件是高度相关的，因此基本不会发生变化，和前边一样由两部分组成实页号（n_v）页内地址（n_r，示意图如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2020/04/image-37d114e4.png?imageView2/2/interlace/1/format/jpg"></>
同样的，由于引入了“块”的概念因此在进行地址变换的时候我们需要一个新的表--<code>块</code>。
<code>块表</code> 具有如下特点：

块表长度与主存页面数相等。
块表行中的内容：块号N_d。

<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2020/04/image-d59afb4b.png?imageView2/2/interlace/1/format/jpg"></>
有了前边两点的铺垫之后，我们下边可以详细讲述一下地址变换的过程：

给出地址变换的一个简单示意图：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-61769047.png?imageView2/2/interlace/1/format/jpg"></>

根据块内页号N_v’去查块表中的N_v’行，获取到对应的块号
将虚地址中的块号N_d与所选块表中的 <span class "language-math">N_d比较。
相同时则命中，直接将N_v’→n_v，N_r→n_r。
不同时，则表示未命中

从上边的分析中我们可以总结出直接相连映象具有如下特点：

可将查表与访存同时进行，有利于访问速度的提高（命中时）。
产生页面冲突的可能性极大（因无灵活的存放余地）。

<h2 id="组相联映象及地址变换">组相联映象及地址变换</h2>
<code>组相连映像</code> 则是先将主存分为页面数相的若干组，再将辅存按主存划分为若干区，组内采用全相联映象，组间采用直接映。
这个很明显通过分组融合了全相连映象和直接映象，说起来还有点小期待:happy::happy:。
同样的，我们先给出示意图：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-734f4fe3.png?imageView2/2/interlace/1/format/jpg"></>
其中：

Nd:区号
s :组号
q:组内页号——辅存
s':组号
q’组内页号——主存

从图中我们可以明显看到：组间是直接相联的，而组内则是全相联。
<h3 id="地址变换--">地址变换</h3>
同样的我们先给出其地址的表示方法
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-eccfd77a.png?imageView2/2/interlace/1/format/jpg"></>
由于映象方式的改变，我们在进行地址变换的时候，需要引入一个新的表--code>随机存储器表</code>。
该表具有如下特点：

表的行数与组数相等（本例 2 组，即 2 行）。
每 行 大 字 段 数与组内页面数相等（本例 2 个 ）。
每 个 大 字 段 又分为三个小字段。
每个大字段还有一个比较器。

为了便以理解我们画出其示意图：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-17e30143.png?imageView2/2/interlace/1/format/jpg"></

>
具体的变换过程如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-6b8a591f.png?imageView2/2/interlace/1/format/jpg"></>

根据虚地址中的 s 去查 RAM 表中的某一行。
将虚地址中的N_d、q 同时送各比较器与所选行中的span class="language-math">N_d、q 进行比较。
当有一个比较器相等时：

将 s→s’ （组间直接）。
将相等大字段中 q’送出作 q’ 。
再将N_r→n_r,即实现了将虚址(N_d,s ,q ,N_r)→(s’, q’,n_r)命中时的地址变换</l>

最后我们总结出组相联映象的特点：
既有直接映象中对号入座部分(组间直接)，可减少查表范围，缩短查表时间，又有全相联中的灵存放规则(组内全相联)，从而可降低页面冲突。
缺点： 控制机构复杂
<h2 id="段相联映象">段相联映象</h2>
所谓 <code>段相连映象</code>，其实跟组相联特别相似--对主辅的划分与组相联映象相同，为区分两种不同的映象规则，将组相联中的组改为段，段间采用全相联，段内采用直接映象。
其示意图如下：
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-f72400f4.png?imageView2/2/interlace/1/format/jpg"></>
<img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/file/2020/04/image-6c206cf6.png?imageView2/2/interlace/1/format/jpg"></>
<h2 id="四种映象规则的关系">四种映象规则的关系</h2>

在组相联映象中，当每组只有一页时，此时的组相联就是直接映象。
当把主存只分为一个组时，此时的组相联也就是全相联映象，即直接映象和全相联映象是组相联象的两个特例。

在段相联映象中，当每段只有一页时，此时的段相联映象就是全相联映象。
当把主存只分为一个段时，此时的段相联也就是直接映象。

<h2 id="总结">总结</h2>
本文主要从存储体系由来、分类以及原理角度来讲解一个存储体系的设计，努力做到全面。当然于个人水平有限，文章难免可能会有错误，如若发现，恳请指出，不胜感激。