第七讲 储存层次结构

701-存储层次结构概况

> 如果CPU爱好哲学，它可能一直在问自己两个问题，
>
> 我要运算的数据从哪里来？我算完的数据又将去向哪里？ 这在计算机世界当中确实是一个非常重要的问题。
>
> 这也正是我们今天所要探讨的主题：存储器。

> 这是我们非常熟悉的冯·诺依曼计算机结构， 那这其中哪些部件和存储功能有关呢？
>
> 存储器自然会是一个，而外部记录介质也是带有存储功能的。
>
> 这两个都比较明显，另外还有一个也带有存储功能那就是运算器。
>
> 因为CPU当中的通用寄存器是包含在运算器当中的，
>
> 通用寄存器中也是用来存放临时的数据，也可以看作是一种存储器。
>
> 那这些就是我们要介绍的构成存储层次结构的主要部件。
>
> 那为了便于描述把这些部件都统称为存储器。
>
> 那我们来看一看计算机当中对存储器有哪些需求。首先存储器当中应该保存了
>
> 将要运行的程序和需要运算的数据，那这些内容在系统通电启动的时候就应该在存储器当中了，
>
> 而且在断电之后也不会丢失。
>
> 那么具有这样特定的存储器我们称之为非易失性存储器。
>
> 而断电后存储的信息就会丢失的存储器则称为易失性存储器。
>
> 那么在现在的计算机系统当中储存和CPU当中的通用寄存器
>
> 都是易失性的存储器，断电之后其中的信息就丢失了。
>
> 而bios芯片和硬盘则是非易失性的存储器，
>
> 所以在系统通电之后CPU必须要从bios芯片开始执行程序，
>
> 然后这段程序把硬盘等设备配置好之后再将更多的程序和数据从硬盘搬运到内存，
>
> 之后CPU才可以在内存里执行程序。
>
> 因此在系统中必须要有非易失性的存储器。
>
> 第二这样的存储器肯定得是可读又可写的。
>
> 那在这个结构当中硬盘和主存都是可读可写的， 而bios芯片则是一个只读的芯片，
>
> 倒不是说它完全不支持写操作，而是对它的写需要 借助特殊的设备或者特殊的操作过程，非常的麻烦，
>
> 无法支持经常性地写入数据。
>
> 第三存储器最好是能够支持随机访问。所谓随机访问是指
>
> 对存储器当中任何一个数据的访问所花费的时间与这个数据所在的位置没有关系。
>
> 那一种典型的非随机访问的设备就是磁带，比如早期要听歌都是要使用磁带的，

假如现在磁带的开头你就可以马上听第一首歌，但是如果你想听

第十首歌，你就得用快进的方式快速的卷过前九首歌才能够听第十首。

这样的访问就是和它的位置有关的，而在计算机程序的运行过程中， 我们会访问到存储器的任意位置，如果不支持随机访问，

这就会对性能造成很大的影响。

在这里对储存的读写和Bios芯片的读都是支持随机访问的。

而硬盘内部实际上是由多个盘片构成，

这些盘片处于高速旋转的状态，并由一个机械的读写头去寻找需要访问的数据的位置。

这就不是随机访问的模式，而且由于期中有机器部件的存在，速度就变得非常的慢。

而在计算机当中存储器的速度也就是访问的时间也是一个非常重要的因素。

CPU的运行速度很快在它需要访问存储器的时候，最好能在

一个时钟周期内就完成数据的访问，不然就会阻碍CPU后续的操作。

而储存的速度是明显高于硬盘的速度的。所以总体看来，

如果我们能找到一个存储器支持随机读写而且是非易失性的，访问时间也很短，

那么就可以考虑只用这样一个存储器和CPU连接，当然我们还要

考虑到是否有足够大的容量以装下所以我们需要的程序和数据， 还有价格是否能够承受，以及功耗是否合适等等。

那既然现状我们是使用了这么多种不同的存储器，虽然是因为没有一个唯一而完美的解决方案，

那么就从性能和价格等方面对它们进行一个比较。

这是30年来具有代表性的CPU，我们每隔十年

选取一个代表，80年的8080，时钟频率 大约为一兆赫兹，其时钟周期是一千纳秒。

90年我们选取了386的一个版本，时钟周期是五十纳秒。

而2000年选择的是奔腾二，时钟周期大约为一点六纳秒。

2010年选择的是Core i7，时钟周期大约是0.24纳秒。

当然我们要注意因为这个时候已经是四核了， 如果每个核，每个时钟周期都需要对外传输数据，

那其实相当于每零点一纳秒就需要传输一个数据。

那么就以这个CPU的时钟周期为指标来看一看外部存储器的特性。

首先我们来看硬盘。

硬盘的访问时间在80年的时候大约是87，我们注意这个单位是毫秒，

毫秒和纳秒差了一百万倍，所以虽然这30年来硬盘的访问速度也有所提升，

大概提升了29倍，但它和CPU的时钟周期完全没有可比性， 相差数百上千万倍。

所以仅从访问时间这一项来看硬盘从一开始就不具备直接和CPU进行交互的能力，

当然它的优势在于容量大而且便宜。

在80年的时候大约每兆字节需要五百美元， 那时典型的硬盘容量是一兆bit，

而在这30年中硬盘单位容量的价格在迅速的下降，下降幅度超过160万倍，

而与此同时硬盘的容量也在迅速的增长，上升的幅度也有150万倍。

这样我们就可以存放更多的程序和数据。

这是硬盘技术进步带来的最大的好处。

当然另外还有一件很有意思的事情，我们发现 价格下降的幅度和容量上升的幅度基本相当，

也就是说现在和30年前相比虽然硬盘的容量和单个字节的价格

有了巨大的变化，但是整个硬盘的价格却基本保持着不变。

那好硬盘作为一个非易失性的存储器自然有它自己的作用。

但是要想和CPU直接交互还得看其他的设备， 这就是DRAM。现在的存储主要是采用DRAM实现。

它的访问时间在1980年大约是375纳秒，

我们注意到这个时候DRAM实际上比CPU的运行的速度还要快一些，

所以这时候并不用担心内存无法及时给CPU提供数据的事情。

而到了90年DRAM的速度已经比CPU的速度慢了，而且后来这个差距越来越大，

到2010年，即使只考虑单核CPU的需求，DRAM的访问时间

也和CPU的时钟周期相差一百倍。那在这同样的这30年里CPU的

时钟频率提升了2500，而DRAM的访问速度却只提升了9倍， 所以这个差距明显是在拉大的。

不过DRAM的进步也同样体现在其容量和价格上，

这30年来其容量足足提升了有12万倍之多， 而成本也几乎有同样比例的压缩，也就是说我们可以

与30年前用同样的价格买到12万倍的容量的内存。

容量扩大自然是一件好事，但是这个性能的差距 到底会带来什么样的影响呢？我们通过一个例子来看一看。

假设CPU直接连接的DRAM构成的主存， 那在CPU当中执行一条指令所需要花的时钟周期大约是这样的，

我们就以奔腾CPU执行指令的五个步骤为例，首先是取指，

我们假设在CPU内部需要花一个周期产生访问存储器的地址， 那么接下来就因为要读存储器而进行等待，

所以这样CPU就会等待一百个时钟周期才能得到这条指令的编码，

然后再花一个周期在CPU内部进行译码，第三步在奔腾当中是生成地址，

第四部则是执行。如果在执行当中需要去 访问存储器读取操作数，那又需要再等待一百个周期，

然后才可以得到操作数，并在CPU内部再用一个周期完成相应的执行工作。

最后画一个周期写回到寄存器，但是如果目的操作数

是存储器的话那还需要花一段时间完成写存储器的操作。

那么这里就可以看到实际上执行一条指令大约要花掉二百多个时钟周期， 而且绝大多数的时间都在等待存储器。

好，我们原先希望一个时钟周期就能完成一条指令，那如果CPU真是这么运行的，

那一个1G赫兹的CPU只相当于时钟主频只有5兆赫兹的CPU的性能了。

这样显然是无法接受的。所以我们必须要考虑如何提升CPU访问存储器的性能。

那既然DRAM的性能提升有限，那么就只能去寻找其它速度更快的存储器， 比如说SRAM就是一种选择。

那么就可以考虑在CPU和DRAM之间，加上一个速度更快的SRAM，

那如果我们能让CPU所需要的程序和数据大部分时间都存放在这个SRAM当中，

那CPU就可以获得快得多的存储器访问时间。

这样一个SRAM的部件，就称为cashe，也就是高速缓存。

那我们来对比一下cache对指令执行时间的影响。

SRAM的访问时间大约可以认为是3个时钟周期，那上面这个指令执行的过程就会变成这- 个样子，

所有读存储器的100个时钟周期都会缩短为大约3个时钟周期， 这样大约只需要总共10个时钟周期就可以完成一条指令了。

相比于之前的性能，有了巨大的提升。

既然SRAM这么好，那么为什么不用SRAM作为主存呢?

要回答这个问题，我们还得来看这张表，我们再加上SRAM的相关参数。

在1980年，SRAM的访问时间大约是300纳秒，比DRAM要快一些，

但是很有限，基本上属于相当的访问时间，但是SRAM的访问速度提升的很快，

三十年间提升了大约两百倍，而这段时间，DRAM只提升了九倍。

在很长一段时间内，SRAM的访问时间与CPU的时钟周期基本相当，

所以在之前学习处理器结构的过程中，我们一直认为 CPU可以在一个时钟周期内从高速缓存当中获得数据，

即使在2010年的时候，SRAM的访问时间也不过是CPU时钟周期的三到四倍，

那这个差距，也完全可以通过继续细化切问流水集来解决。

因此从速度上来说，SRAM作为与CPU直接连接的存储器是合适的。

但是我们能不能直接把它作为主存呢？那我们就得来看看价格是否能够接受了。

我们注意到在2010年，同样容量的SRAM，它的价格是DRAM的

一千倍。如果说我们现在花一千块钱就可以买到一个非常好的内存条，

那你是不是愿意花十万块钱买一个相同容量的SRAM作为内存呢？所以即使仅从价格因素

来考虑，SRAM也无法取代DRAM用来作为内存，而是只能用一个小容量的SRAM

作为高速缓存，保存最常用的程序和数据，以达到性能和价格的平衡。

那是什么时候开始用SRAM作为cache加到CPU和主存之间的呢？ 其实从这张表上我们也能够看出来，

在80年的时候，DRAM还是比CPU的速度要快的，

所以这个时候没有使用高速缓存的需求，而到了90年，DRAM的速度已经比CPU慢了，

所以在我们用作例子的这些英特尔系列的CPU中，

正是在1980年到1990年这个过程中，逐渐产生了高速缓存的需求。

那在这个时期，我们细致地来看， 早期的8088CPU访存实际上需要840纳秒，

而DRAM的延迟是250纳秒，那当然就不需要cache。

286的访存周期大约是200纳秒，DRAM的延迟呢， 是220纳秒，也基本相当，这时候还是不需要cache。

而386的访存周期大约是80纳秒，DRAM的延迟

虽然有所进步，但还是需要190纳秒，这时候已经明显慢于CPU了， 但是386CPU芯片内部并没有设计cache这个部件，

后来也发现这对性能有很大的影响，所以当时是采用 在芯片外再增加一块SRAM芯片作为cache来解决这个问题。

那么到了486的时候，就已经把cache集成到了CPU芯片的内部，

从而缓解CPU和DRAM主存之间的性能差距，

因此，现代的计算机当中都采用了这样层次化的存储结构。

在这个层次结构中，越往上的部件， 容量越小，但速度更快，而单位字节的成本更高，

越靠下的部件，容量更大，但速度更慢，而单位字节的成本更低。

现在，我们已经了解了计算机对存储器的基本需求，

也知道了存储器的基本特性和大致的分类， 那我们就要对其中的细节进行深入的探索，

以及建立完整的计算机存储体系。

702-DRAM和SRAM

SRAM比较快，DRAM比较慢，

SRAM比较贵，DRAM比较便宜，记住这些结论是很容易的， 但是比是什么更重要的是为什么，

那在这一节我们就从电路的级别来探索DRAM 和SRAM为什么会具有他们这样各自的特点。

这就是我们平常在计算机

当中使用的内存条，规范的说法我们应该叫它内存模组。

而一个内存模组实际上就是由一块小电路板再加上若干的DRAM芯片构成。

在这个内存模组上一共有八个DRAM芯片。

然后我们来看DRAM芯片内部的结构。

DRAM芯片内部核心的结构就是这个存储阵列，由若干行和若干列构成。

如果从外部给入行地址，在给入列地址，那么就可以由此选中一个存储单元，

而在一个存储单元当中往往包含着若干个比特， 常见的有四比特或者八比特。

每一个比特都采用了这样一个电路的结构， 这个结构我们称为DRAM的一个基本存储单元，

这个基本单元就通过这个电容来保存了一个比特的信息， 那在DRAM芯片当中还有一些地址译码的逻辑，

根据外部送入的行地址生成行选择信号， 再根据外部送入的列地址，生成列选择信号，

那通过这样的行列选择信号就可以选中对应的存储单元。

其实也就是将这样的电容联通到外部的输入输出线，

那如果是写入就是由外部的数据线驱动对电容进行充电或者放电的操作， 从而完成写入一或者零。

那如果是读操作则是由电容对外部的数据信号线进行驱动，从而完成读出零或者一。

这个电路结构非常简单，但是我们要注意电容是存在漏电效应的，

如果经过一段时间电容上的电荷流失过多，我们就会丢失 它所存的信息。因此

为了保证DRAM的存储的信息的正确性，我们就需要定期对所有的单元进行刷新。

如果这个电容保存的数据信息是一，那么应该补充这个电容的电荷，

而如果它原先保存的是数据零，那么也需要通过刷新让它保持无电荷的状态。

那这就是DRAM的基本结构和运行的原理。

那现在的pc机当中大多是采用DRAM作为主存储器也就是内存，

那当然DRAM也有很多种不同的实现，比如SDRAM，这个s

是同步的缩写，那它是在DRAM的基本结构基础上对输入输出接口进行时钟同步。

从而提高了读写的效率，而现在更为常用的内存 则是在SDARAM的基础上又进行了进一步的改进，

比如说这些年有DDR1，DDR2，DDR3， 这些不同的内存标准都是在SDRAM的基础上进行的改进和升级。

那总体来说由于每个DRAM的基本存储单元非常的简单， 所以它能够做到很高的集成度，而且功耗也比较低，

我们可以用较低的成本制造出大容量的DRAM芯片。

当然它的缺点也跟它的这个结构有关， 因为给予了电容充放电的方式，

所以DRAM的读写速度还是比较慢的，而且它还需要定时刷新，这进一步影响了DRAM读- 写性能。

那么与DRAM相对的，我们还有一种常用的存储单元是SRAM。

我们先来看看SRAM的基本结构，这个结构一共有六个晶体管构成。

我们分别用m1，m2一直到m6进行标记，那这六个晶体管

合在一起才可以保存一个比特，显然比DRAM的一个基本存储单元要复杂得多。

那这个基本存储单元对外有三个接口信号，其中BL和BLbar是用于

传送读写数据信息的，而wl信号则是用于控制外部是否可以访问这个基本存储单元。

那么就通过读写的实力来看一看SRAM是如何工作的。

我们以写入数据1为例来进行分析。

首先这个存储单元中连接的电源盒D分别代表着一和零，

那只要SRAM处于通电状态，这两个信号的值都会保持稳定，

那如果要写入1，就得将BL的信号的值置为1，而与之相对 则需要将BL

bar信号置为零，这样就准备好了要写入的数据，

然后我们需要访问这个SRAM单元，就得把wl信号置为1，一旦wl信号为1，

那它所连接的M5和M6这两个晶体管的gate端就都为1。

那通过这个图标我们就可以看出M6是一个NMOS晶体管，

NMOS晶体管的gate端连接为一时，那这个晶体管应该是连通还是关闭呢？ 这一点我们以前已经学习过，你还记得吗？

那当然对于NMOS来说，如果gate端为一，它应该是处于连通状态。

因此BL信号上的值就会通过M6传递进来， 那么来看一看这根信号控制了哪些晶体管的gate端。

首先它是连到了M2晶体馆的gate端， 那么看M2是一个什么类型的晶体管呢？

它是一个PMOS管，PMOS管的gate端如果为1，那就应该处于关闭的状态。

那因为M2是关闭的，所以从电源这连过来的1不会被传导出来，

那所以这根信号线上的值到底是什么，我们还得来看下面这个晶体管也就是M1。

而M1的gate端和M2的gate端连接的是同一个信号，

所以现在也是1，M1是一个NMOS，所以它处于连通状。

所以这根信号上的值就应该是零， 那么还应该注意到这根信号线同时连接到了M5的这跟晶体管，

而M5的gate端也是由WL信号控制的， 所以M5也是处于联通状态，它会将BL bar信号的0传导进来，

那么我们刚才分析了从D传过来的也是零， 这样就没有冲突，这根信号就可以稳定地维持为0。

那么我们现在来看这根信号又控制了哪些晶体管的gate端。

首先是M4，M4是一个PMOS管，它的gate端为0的时候，

它就处于连通状态，那来自电源的这个1会被传导进来，

与此同时M3的gate端也是0，那这根NMOS晶体管就会处于关闭状态。

所以这根信号与D是不连通的，

那这样从M3和M4的控制来看，这跟信号也应该是1，这和由BL信号传递进来的这个1也- 是没有冲突的。

那这样由外部的BL和BL bar输入的信号 就驱动了这个SRAM存储单元的晶体管达到了一个稳定的状态。

那因为晶体管的工作速度很快，所以这个SRAM的写入的过程也非常快，

那当这几个晶体管状态达到稳定之后我们就可以结束这次写的过程了。

现在我们可以将WL信号由1变为0，从而将M5和M6这两个晶体管关闭，

然后就可以撤销BL和BL bar上的信号了。

那在外部的信号撤销之后，那因为M5和M6这两个晶体管都关闭了，

所以并不是由外部的BL信驱动M2和M1这两个晶体管的gate端，

但是我们注意在刚才写入的过程中，因为M4晶体管处于连通状态，

所以电源所传导的这个1会始终驱动这根信号， 保持M2和M1的gate端为1，

而只要M2和M1的gate端为1，那么由D传过来的这个0就会保持稳定，

而这个0又会反过来驱动M4和M3保持刚才的状态，

从而让这个1也保持稳定，那么在这四个晶体管的相互作用下， 那这个SRAM单元就稳定地保持住了刚才写入的这个数据1。

那现在只要WL信号保持为零，这个SRAM 存储单元当中就会始终存储的这个数据1，

当然前提是要保持通电的状态。

如果断电，电源的这个1消失了，那SRAM当中保存的这个数据也就会丢失。

那在这个基础上我们再来看SRAM的读出过程。

这还是刚才那个SRAM单元，

里面已经保持了一个数据1，那现在外部就来读取这个单元。

这还是需要把WL信号置为1， 那这样M5和M6这两个晶体管都会处于连通的状态，

而此时外部并不驱动BL和BL bar这两根信号，

因此它们将会分别通过M5和M6这两个晶体管被这个SRAM的存储单元所驱动。

BL bar的信号将会被驱动为0，而BL的信号将会被驱动为1。

这样外部在接受到这两根信号之后就可以判断出读出的是一个数据1。

这就是一个SRAM基本单元的结构和工作原理。

那如果我们将很多个这样的基本单元组合起来就可以构成一个存储矩阵。

那从外部可以输入若干个地址线，那通过再增加一些地址译码的驱动电路，

就可以根据地址信号线的值选中存储矩阵当中指定的若干个SRAM的基本存储单元，

由SRAM构成的存储器和DRAM芯片不太一样，它不需要分成行地址和列地址分别选择。

而且相对来说SRAM的设计是比较灵活的， 一个地址所对应的存储单元的数量也许是8个比特，10个比特，

也可以是32个比特，40个比特，64个比特等等都可以。

那通过地址线选中的一组存储单元之后，我们还要确定到底是需要读还是写，

所以外部还需要有读写控制信号，并且有相应的读写控制电路。

而且这些读写控制电路还会将外部传来的数据线连接到SRAM的存储单元上，

那如果现在是写这样来自我外部数据线的信号去驱动存储单元，

而如果是读则将存储单元当中的信号驱动外部的数据线， 那这样的电路一般就称为一个SRAM的存储器，

通常这个SRAM存储器还会有片选信号， 以标明当前是否需要对这个SRAM存储器进行读或者写的操作。

那由此看来因为晶体管的开关速度

远比电容充放电的速度要快，所以相对于DRAM，SRAM有速度快的优势。

但是我们也看到SRAM中要存储一个比特就需要用六个晶体管，

晶体管数量多就会造成芯片的面积大，从而带来集成度低和价格较高的问题。

同时每个晶体管都是要耗电的，晶体管越多功耗也就会越高，这些都是SRAM的缺点。

那现在CPU当中的高速缓存一般都是用SRAM来实现的。

比如这就是一颗四核CPU的版图，

> 在这个芯片当中这些大面积的看起来非常规整的电路实际上都是SRAM，
>
> 所以仅从制造成本上来看这些作为高速缓存SRAM 往往要占到整个CPU的一半或者更多。
>
> 而且由于高速缓存大多和CPU采用相等或者接近的时钟频率，
>
> 所以它们的功耗也非常高，这都是在使用SRAM实现高速缓存时需要注意的问题。
>
> 那么综合比较来看DRAM的 主要优势在于集成度高，功耗低，价格低，
>
> 而SRAM的优势则于速度快，而且不像DRAM需要经常进行刷新，
>
> 所以我们需要根据它们不同的特点，用在不同的场合上。

> 现在我们已经掌握了DRAM和SRAM的内部结构
>
> 以及其工作原理，那我们也就知道了他们的那些特点是如何产生的。
>
> 这对我们理解存储体系是如何构成的有着非常重要的帮助。

703-主存的工作原理

> 主存，我们通常也叫做内存，这是
>
> 计算机当中非常重要的一个部件。冯诺依曼把计算机分成了五大组成部分， 其中存储器实际指的就是这个部件。
>
> 那今天，我们就以PC机当中常用的组成为例， 来看一看这个部件的工作原理。

> 我们还是从DRAM芯片的内部结构开始说起。
>
> DRAM芯片以一个存储阵列为核心， 这个存储阵列以行列的形式组织，
>
> 那么行列的交点就是一个存储单元。
>
> 每个存储单元都有唯一的一组行列地址指定。
>
> 那么这样一个存储单元一般由若干个比特构成，常见的有4比特或者8比特，
>
> 而每一个比特都是由这样的一个基本电路构成。
>
> 通常情况下，DRAM芯片的外观是这样的。
>
> 而在这里，有8个DRAM芯片焊接在这样一块绿色的小电路板上，
>
> 构成了一个内存模组，这也就是我们通常所说的内存条。
>
> 因此，从外部给入了行地址和列地址之后，这些地址会同时送到每一个DRAM芯片，
>
> 从而在每个DRAM芯片当中选中对应的一个存储单元。
>
> 如果每个DRAM芯片送出8个比特，那它就可以向外同时送出64个比特。
>
> 因此，如果从CPU送到内存条一组行列地址， 那内存条就可以返回这组地址所对应的，
>
> 一个64位的数。这就是一个数据在内存当中大致的存放方式。
>
> 当然我们也可以不用内存条的形式， 尤其是对于平板电脑和智能手机，它们本来体积就很小，
>
> 很难容纳内存插条这样大个的组件，而且它们也不一定需要这么多的
>
> DRAM芯片，所以往往是在它们的主板上直接焊接DRAM芯片。
>
> 但不管是哪种情况，一般都是由若干个DRAM芯片构成了计算机的主存储器，
>
> 也被称为内存。那么就来看一看CPU是怎么访问内存的。

我们以SDRAM作为例子来说明这个访问的过程。

SDRAM就是同步的DRAM。

那在计算机内部，CPU通过系统总线连接到了内存控制器，

而内存控制器再通过系统总线连接到了内存条，我们在这里着重画出了其中一个DRAM芯片，

那实际上内存控制器会把相关的地址线、数据线连接到内存条上的各个DRAM芯片，

那当CPU需要访问存储器时，那首先要申请

系统总线，在获得总线控制权后会将地址发到内存控制器中，

对于一个32位CPU，那这个地址一般就是32位的。注意在这个时候， 地址并不会分成行地址和列地址，而是只有一个地址。

然后内存控制器会将这个地址进行分解，形成行地址和列地址等

多个部分。然后内存控制器就会向DRAM芯片发起访存操作。

在这一步，可能会包括两个部分，一是称为预充电的这个操作，

这里打了中括号，代表了这是一个可能有也可能没有的操作，

那么就放在后面再说。然后进行行访问，也就是发出行地址。

通过存储总线发出的行地址会被DRAM芯片当中的 行译码器接收到，就会在存储阵列中选中对应的那一行，

然后这一行当中的所有的存储单元的信号都会被经过放大之后放入到一个缓冲区当中，

那这个过程就会被称为激活，或者是行访问的过程。

那么只有等这个行缓冲区的信号都稳定了，我们才可以进行下一步的操作。

因此，我们需要关注的一个时间，就叫做tRCD，这是从行选

到列选的延迟。我们简单地来看一下这个时序图。假设在这个时钟上升沿，我们发出了行地址，

那么我们必须等待tRCD这么长的时间，才可以去进行 下一步的操作，也就是发出列地址。那这个时间的长短

是由这个DRAM芯片本身的特性决定的。一般来说，它的质量越好，这个时间就越短。

而对同样一个DRAM芯片，它的工作环境越好，这个时间也会越短。

那么等tRCD这段时间过去之后，我们就可以发出列地址了。

这时内存控制器就会把事先准备好的列地址发到DRAM芯片，由列译码器接收，虽然

我们现在是把这个箭头画在这里，实际上内存控制器还是通过存储总线把这个列地址发过去的。

我们另外画这条箭头只是为了看起来比较清晰。那么列译码器收到 列地址之后，就会从缓冲区中选出对应的那一列，

如果现在要进行读操作，那被选中的这个存储单元的数，就会送到数据输出接口上去，

而从发出列地址，到选出对应的存储单元的数这个过程，

就被称为列访问的过程。那么列访问也是需要时间的。

这还是刚才那个时序图。从发出行地址到可以发列地址，中间要等待tRCD这么长的时间，

然后从发出列地址到选中的存储单元的数可以输出，也需要等待一段时间，

这段时间以CL标记，也就是从列选到数据输出的延迟。

而且通常情况下，访问内存都不会只读一个数，而是会连续读出多个数，

那么这些数会每一个时钟周期输出一个，依次地送到数据线上。

也正是因为如此，我们要事先把一整行的数， 都读到缓冲区里，因为它可以把每一个存储单元同时都连到了缓冲区中，

读出一个存储单元也是读，读出一整行来也是读，所花的时间并没有明显的差别。

所以不如一次把一整行都读出来，然后从中选择需要的连续的若干数据送出去，

而且这样还有另外一个好处，如果下一次访存还是在这同一行，那就不需要

重复发这个行地址了。因为对应的行已经在缓冲区当中，只要直接发列地址就可以。

这样就可以大大地缩短访存的时间。那好，现在我们先回到这一次访问。

当DRAM芯片送出数据之后，

内存控制器就会采样对应的数据，然后将采样到的数据再送回到CPU当中去，

那过一段时间CPU又会发出访存的地址，那如果这次要访问的数据

和刚才要访问的数恰好在同一行，那就不需要再重新发行地址，只需要直接发列地址， 从缓冲区中选出对应的单元就可以了。

当然，如果下一次访问所要的数据不是这一行， 那么就需要把激活的这一行关闭，

这个过程我们称为预充电，实际上预充电 最早可以在前一次传输，最后一个数据即将送出的时候开始，

因为我们不确定下一次传输到底会不会在同一行。所以我们有两种可以选择的策略，

一种方式是等到新的传输开始，如果发现要访问的数据 不在已经被激活的这一行，那时再进行预充电，

这也就是刚才在步骤二中，我们提到的那个可能没有的预充电操作。

而另一种方式，则是在一次传输结束后就进行预充电。

这样在下一次的传输是同一行的概率不高的情况下，反而会获得更好的性能。

那么预充电也是需要花一定时间的，这个时间我们记为tRP，

从内存控制器发出预充电的命令到DRAM芯片可以接收下一次行地址， 这段时间就被称为tRP。

所以，虽然一个SDRAM有很多的性能参数，但是其中最关键的

有这么几个，一是tRCD，也就是从行选到列选的延迟时间，

在PC133这个标准当中，这个时间大约是15到23个纳秒，

因为PC133标准的时钟频率是133兆赫兹， 所以相当于2到3个时钟周期。

而CL这个参数，则指明了从列选到数据输出的延迟周期数，

注意它跟tRCD不同，tRCD规定的是一个时间，

单位是纳秒，而CL规定的是一个时钟周期数的延迟，

在PC133标准中，这个参数是2到3个时钟周期，大约相当于15到23纳秒。

第三个重要的性能参数是tRP，也就是行预充电的延迟时间。

同样在PC133标准当中，大约是15到23个纳秒，也相当于2到3个时钟周期。

我们以PC133标准的SDRAM为例，来看一个读操作的典型访问过程。

那么因为时钟频率是133兆赫兹，所以一个时钟周期 是7.5纳秒。那当内存控制器发出行地址之后，

需要等待tRCD时间，才能发出列地址，

然后等待CL个时钟周期，DRAM芯片才会送出第一个数据，

这时内存控制器才可以采样，那么这个时间大约是6个周期，45个纳秒。

随后每个周期DRAM芯片都会送出一个数据， 而每个数据的宽度，则取决于内存条上有几个DRAM芯片，

然后在最后一个数据送出之前，内存控制器可以发出预充电的命令，

再经过tRP的时间，就可以发出下一个访存操作的行地址了。然后后面，

再经过tRCD以及CL的时间，才会得到下一次访存的数据。

因为两次内存访问之间的时间间隔大约是12个周期，90纳秒。

从这里我们也可以看出，虽然内存的时钟频率是7.5纳秒， 但并不意味着只需要7.5纳秒就可以得到想要的数据。

现在我们已经了解了SDRAM内存的基本结构和工作原理。

那近些年来，SDRAM内存又有了长足的进步， 在下一节，我们就来看一看，这些

是如何发展的。

704-主存技术的发展

近些年来SDRAM的标准不断地升级换代，

其性能也得到很大的提升。那为什么 还总是说CPU和内存之间的速度差距越来越大了呢?

这一节我们就来探讨这个问题。

近20年来，流行的内存都采用了SDRAM的结构，需要注意的是

虽然平时我们会说以前用SDRAM内存，现在用DDR内存， 但这种说法是不规范的。所谓DDR内存

也是SDRAM的一种，只不过接口采用了双倍数据率而已。

那么有了DDR SDRAM之后，如果要特指之前的SDRAM，可以说是SDR SDRAM。

SDR就是单数据率的简称，而DDR之后， 又有了DDR2，3，4等后续标准。

为了区分明确，也把第一代的DDR称为 DDR1。这些内存规格后面带的数字，

比如这里的1600，指的是内存接口的等效时钟频率。

借此我们可以计算出内存的传输带宽。

如果按照内存位宽为64比特，由此计算出

的内存传输带宽，早期的pc66是0.53GB/s， 而到了DDR3-1600则达到了12.8GB/s，

十几年里增长了二十倍之多。

然而，很多经典教材和资料中都会提到，近几十年来，

计算机处理器与存储器的性能差距日益加大，

存储器性能的增长不到十倍。那这又是怎么回事呢?

这个说法是否有问题呢？那么

之前我们已经学习过，SDRAM的基本单元是一个电容和一个晶体管。对于SDRAM的读写 对于SDRAM的读写最终体现在了对电容的充放电，

而要减少这个电容的充放电时间，那是非常困难的。

因此，在SDR SDRAM的时代，当时钟频率

提升到了133MHz之后就很难进一步提升了， 那后来的DDR SDRAM又是如何提升性能的呢？

那我们就来看一看内存模组的内部结构。对于一个位宽8 比特的SDR

SDRAM芯片，每次访问其内部的存储阵列都会取出8比特的数据。

而同样位宽的DDR SDRAM芯片，每次访问存储阵列则会取出两倍的数据量。

那这16比特的数据是怎么传输的呢？这就要用到双倍数据率的概念。

所谓的双倍数据率也就是DDR，

相对应的是SDR，SDR是指每个时钟的上升沿传输数据，接收端

也用时钟的上升沿采样数据，而DDR则是在时钟的上升沿和下降沿都传输数据，

这样就可以在同样的时钟频率下传输双倍的数据。

需要说明的是，DDR指的是这种传输方式， 运用在SDRAM内存上，就有了DDR

SDRAM， 但它不仅仅用在内存上，还用在其他很多领域。所以

DDR只是一种传输数据的方式，不能将它等价于我们现在用的内存。

那我们再来看一看DDR SDRAM是如何工作的。

先来看看PC100规格的SDR SDRAM。

它的存储阵列的核心时钟频率是100MHz， 对外接口的时钟频率也是100MHz。

以读操作为例，每个时钟周期从存储阵列中 读一个数据并送到芯片的接口信号上，

从外部看来每个时钟上升沿传输一个数据， 单根数据线上的数据传输率为100MBPS。

而第一代DDR标准及DDR-200， 它的存储阵列的核心时钟频率也是100MHz，

但是每个时钟周期读出两个数据，接口的时钟频率 还是100MHz，但因为在两个时钟沿都传输数据，

所以仍然在一个时钟周期内把这两个数据都传输出去了。

从外部看来每个时钟上升沿和下降沿各传输一个数据，

单根数据线上的数据传输率为200MBPS，这也就是DDR-200的名称由来。

到这里我们应该发现了一个问题， 通常理解的提升内存性能应该是减少读写数据的时间，

就像我们去图书馆借书，原来办手续需要十分钟，

现在图书馆说要提升一倍的服务效率，我以为会只要五分钟了， 但是DDR SDRAM的思路却不一样。

它说，你看这样好不好？办手续呐我们还是保持十分钟， 但你不是要借一本数学书吗？我现在多给你一本语文书，

也就算我们的服务效率提升一倍了。这算不算欺骗消费者呢？

咱们稍后再说。即使是保持办手续还是十分钟这件事情上，它也没有做到。

这张表列出了SDR到DDR1的各代内存的典型规格。

我们看到SDR的核心频率从66M 到100M再到133M，性能肯定是越来越好了。

但是到了DDR1的第一代，也就是DDR-200，

它的核心频率反而下降了，从133MHz降到了100MHz。

其实这也容易理解，要从存储阵列中每次取出双倍的数据，

虽然改动并不大，但对性能还是有影响的，因此速度也就慢了下来。

那DDR-200相对于PC133到底提升了什么呢?

我们先以对PC133的读操作为例，

大概需要15个纳秒进行行选，再用15个纳秒进行列选，

然后才能得到第一个数据。而与这个数据相邻的后续数据则可以在每个时钟周期送出一个，

而从发出地址到传输完这四个数据总共需要60ns。

我们再来看DDR-200， 因为要从存储阵列中取出双倍的数据，所以对存储阵列的访问变慢了，

大概需要20ns的行选，再加20ns的列选，这样才能得到第一个数据。

不过在与这个数据相邻的后续数据则可以在每个时钟周期送出两个，

因此，送出这四个数据，大约需要20ns。DDR-200宣称的

1.6GB/s的传输带宽就是根据最后这一段20ns送出了四个数据计算出来的。

这个指标本身没有错，但是评价内存的性能要看两个指标。

这个指标称为访存带宽，是内存厂商大力宣传的。

另一个重要的指标就是读出第一个数据的时间称为访存延迟，

因为CPU真正需要的很可能只是第一个数据，所以访存延迟非常关键。

这里DDR-200读出第一个数据用了40ns，比PC133还慢了10ns.

即使是看读出这四个数据的总时间， 它也用了总共60ns，和PC133是一样的。

因此DDR-200的性能表现是要比PC133差的。

不过，随着制造工艺的改进到了下一代的DDR-266，情况有了一些改善。

DDR-266的核心频率已经提升到了133MHz，

和PC133一样，但是行选的时间还偏慢一些，这样

读出第一个数据大约需要35ns，不过连续读出四个数据的总时间

已经比PC133要快一些了，大约是50ns。

我们再来看一看行选和列选时间的参数的总表。

等到了DDR-400的时候，最短的行选时间

已经和PC133一样是15ns了，而列选也更快了一些，

缩短到了10ns，因此DDR-400的性能要明显优于PC133了。

但是到了这个时候，核心频率的提升又遇到了困难。

那么怎么继续提升内存性能呢？这就轮到DDR2出场了。

既然提升核心频率有困难，那就再增加预先取出数据的数量，

之前是从存储阵列中取出双倍的数据， 那现在就取出四陪的数据。例如

DDR-400，它的核心频率只有100MHz， 和DDR-200一样，但因为预取了四个数据，

从外界看来数据传输率达到了400MBPS。

当然这时候单靠双倍数据率的传输也不够了。

所以它的接口时钟频率也提升了一倍，变成了200MHz。

这样上下始终沿都传输数据，正好在核心时钟的一个周期内把四个数据都送出去了。

和之前一样，DDR2的第一代标准的

性能也不好，其核心频率比当时流行的DDR-400要低一半，

不但访存延迟大的多，而且访存带宽也是一样的，所以

DDR2-400的性能表现是全面落后于DDR-400的，很快就被市场淘汰了。

随着制造工艺的改进，DDR2的核心频率提高之后，性能才逐渐地超过了DDR1。

不过呢，它的核心频率也没有提高多少。

而从DDR2到DDR3的升级，

基本上也是一样的套路。

这是内存的行选时间的变化图，

图中的每一个红点都是新升级后的第一代。我们可以看出，

内存标准的每次升级都会让行选的时间变长，之后再慢慢地缩短。

从SDR SDRAM的最后一代也就是PC133，到后来多年占据主流地位的

DDR3-1600十几年的时间里行选时间也只是从15ns缩短到了10ns，

降低了大约30%。

这是内存列选时间的变化图，情况基本一样。

我们可以看出内存的发展过程，基本上就是以牺牲访存延迟为代价，努力提高访存带宽。

访存带宽确实有了几十倍的提升，但是访存延迟的优化幅度 很有限。这是由SDRAM的基本单元结构决定的，

也就是那个电容的充放电过程。如果不同材料和结构上做

根本的改变，基于现在的SDRAM结构想要优化访存延迟是很困难的。

所以说从处理器访存的基本要求来看，读出一个

指定数据所需要的时间确实没有多少改善，这个说法是没有问题的。

这就是所谓内存性能越来越高的真相。

如果我是CPU，我想要的数据你并不能快一点给我，

但你却给了我很多我现在用不着的数，这对性能真的有帮助吗?

从现状来看肯定是有的， 那这样的帮助是如何实现的呢?

这就是我们下一节要探讨的问题了。

705-高速缓存的工作原理

> 因为CPU的速度和内存的速度差距越来越大，
>
> 那计算机整体系统的性能，就受到了巨大的影响，
>
> 而高速缓存技术的出现，则挽救了这个局面。那在这一节， 我们就来看一看高速缓存是如何工作的

> 就是计算机的存储层次结构， 高速缓存，也就是Cache对于CPU和主存之间，
>
> 相比于主存，它的容量要小的多，但是速度也快很多，
>
> 但是为什么，在CPU和主存之间，增加这么一个速度很快，但是容量很小的存储部件，
>
> 就可以提升这个计算机系统的性能呢？这就要得益于计算机中运行程序的一个特点，
>
> 这个特点被称为程序的局部性原理。
>
> 我们通过一个例子来进行说明，这是一段很常见的程序， 有两层循环，对一个二位数组进行累加，
>
> 如果sum这个变量是保存在内存中的，那它所在的这个存储单元，就会不断的被访问， 这就称为时间局部性，
>
> 这些对循环变量进行判断 和对循环变量进行递增的指令，也都会被反复执行，
>
> 而另一点，叫作空间局部性，指的是正在被访问的 存储器单元附近的那些数据，也很快会被访问到。
>
> 那么就来看这个数组，它在内存当中，是连续存放的， 从a00，a01，a02，这样一个接一个的存放下去，
>
> 那么在这段循环访问它的时候，访问了a00之后，很快就会访问a01，然后很快会访问- a02，
>
> 这样的特征，就被称为空间局部性， 那Cache就是利用了程序的局部性原理，而设计出来的，
>
> 首先，我们来看Cache对空间局部性的利用， 当CPU要访问主存时，试讲时把地址发给了Cache，
>
> 最开始，Cache里面是没有数据的，所以Cache会把地址再发给主存，然后从主存中- 取得对应的数据，
>
> 但Cache并不会只取回CPU当前需要的那个数据，
>
> 而是会把与这个数据，位置相邻的主存单元里的数据，一并取回来， 这些数据就称为一个数据块，
>
> 那么Cache会从主存里，取回这么一个数据块，存放在自己内部，然后再选出CPU需要- 的那个数据送出去，
>
> 那过一会儿，CPU很可能就会需要刚才那个数据附近的其它数据， 那这时候，这些数据已经再Cache当中了，就可以很快的返回，
>
> 从而提升了访存的性能。第二种情况，是Cache对时间局部性的利用，
>
> 那因为这个数据块暂时会保存在Cache当中，那CPU如果要再次用到
>
> 刚才用过的那个存储单元，那Cache也可以很快的提供这个单元的数据，
>
> 这就是Cache对程序局部性的利用，我们要注意，这些操作都是由硬件完成的，
>
> 对于软件编程人员来说，他编写的程序代码当中，只是访问了主存当中的某个地址，
>
> 而并不知道这个地址对应的存储单元，到底是放在主存当中，还是在Cache当中，

如果这个不太好理解的话，那我来打个比方，

我们可以把主存看作一个图书馆，里面可能有几千万册的藏书， 那CPU就像一个来借书的人，那他给出了一个书号，希望借到这本书，

从 :3:47 开始播放视频并学习脚本3:47

那可能要花几个小时，才能找到这本书，拿出来交给借书的人。

那过一会儿，借书的人把这本书还了以后，管理员又把它放回到仓库当中去，再过一会儿，这- 个人又要借这本书，

那管理人员又要花几个小时，从仓库里再找到这本书，拿出来， 后来管理员发现，这样的工作实在是太低效了，

于是就在这个仓库的外面，借阅的柜台旁边，准备了一个柜子，把刚才借过的书，暂时先存放- 在这个柜子里，

有些书在很短的时间内，会被多次借阅， 所以这本书放在这个临时柜子里的时候，经常就会被借阅到，

那管理员就可以很快的把这本书交给借阅的人，而不用去大库里面，花几个小时去寻找。

于此同时，管理员还发现了另一个现象，那就是有一个人来借了一本明朝的故事，

然后过一会儿，就会再来借清朝的故事，在过一会儿，可能又来借宋朝的故事，

而这些书，在仓库里面，都是紧挨着排放的， 所以他干脆这么做，当要从仓库里取出一本书的时候，

就把和这个书在同一层书架上的所有的书，都一次性拿出来， 这样并不会比取一本书多花多少时间，

而对于借书的人来说，他并不会知道管理员做了这件事情， 对他来说，还是给出了一个书号，然后过一会儿得到这本书，

他只会发现，借书的速度变快了，这个类比差不多就是Cache所做的事情。

那我们再来整理一下Cache的访问过程， 那现在CPU发出度请求，都是直接发给Cache了，

然后Cache这个硬件的部件，会检查这个数据，是否在Cache当中，

如果是，就称为Cache命中，然后从Cache当中取出对应的数据，返回给CPU就- 可以了。

但是如果这个数不在Cache中，我们就称为Cache失效， 那这时候，就要由Cache这个部件，向主存发起读请求，

这个过程CPU是不知情的，它仍然是在等待Cache返回数据，

那Cache向主存发出读请求之后，就会等待主存将数据返回，

这个过程会很长，那么当包含这个数据的一整个数据块返回之后，Cache就会更新

自己内部的内容，并将CPU需要的那个数据返回给CPU，这样就完成了一次Cache读- 的操作。

那么Cache这个部件内部是如何组织的呢?

Cache主要组成部分是一块SRAM，当然还有配套的一些控制逻辑电路，

那这个SRAM的组织形式，像这个表格，它会分为很多行，

那么在这个事例的结构当中，一共有16行， 每一行当中有一个比特，是有效位，还有若干个比特是标签，

然后其它的位置，都是用来存放从内存取回来的数据块。

在这个例子当中，一个数据块是16个字节， 那么还是通过一个例子，来看一看这个Cache是如何运行的，

我们就用这个表格来代表Cache，假设现在这个Cache里面全是空的， 有效位为0，代表它对应的这一行没有数据，

那么现在来执行这四条指令，第一步，我们要访问2011H这个内存地址，

并取出对应的字节，放在AL寄存器当中去，那CPU就会把这个地址发给Cache，

那因为现在Cache全是空的，所以显然没有命中，那Cache就会向内存发起一次读操- 作的请求。

但我们要注意，因为Cache一次要从内存中读出一个数据块，而现在这个

Cache的结构，一个数据块是16个字节，所以它发出的地址，都是16个字节对齐的，

所以这时，Cache向主存发出的地址，是2010H， 这个地址是16个字节对齐的，

而且从它开始的16个字节的这个数据块当中，包含了2011H这个地址单元，

当Cache把这个数据块读回来之后，会分配到表项1中，那么在这个表项当中，

这个字节就是2010所对应的数据，这个字节就是2011所对应的数据， 所以Cache会将这个字节返回给CPU，

但是Cache为什么要把这个数据块，放在了表项1当中呢？ 我们详细来看一看，

CPU在执行这条指令的时候，Cache收到的地址，实际上是2011H，

那因为现在一个数据块当中，包括16个字节，那最后的

这个16进制数，正好用来指令这16个字节当中的哪一个字节，是当前所需的，

因此，我们取回的这个数据块，要放在哪一个表项当中，就要靠前面的一个地址来决定。

那么在现在的Cache设计当中，一般来说，都是用剩下的这些地址当中，

最低的那几位，来决定到底把这个数据块放在哪一个Cache行中，

那我们现在由16个表项，所以也需要4位的地址来决定，

那因此，现在剩下的最低的4位地址，就正好是这个16进制数了，

这个数是1，所以Cache就决定把这个数据块放在表项1的Cache行里，

那现在还剩下8位的地址，我们也必须记录下来，

不然以后就搞不清楚，这个Cache行里存放的数据，到底是对应哪一个地址的，

所以剩下的地址，不管有多少，都要存放在标签这个域当中。

当然，在把数据块取回之后，还需要把这个有效位置为1，

那这样，我们通过这个表格，就可以明确的知道，当前的这个数据块，是从2010这个地址

读出来的，那在这个Cache行中的第一个字节，就是CPU现在做需要的那个字节了。

那把这个字节取出来，交给CPU，这条指令对应的读操作就完成了。

然后我们再来看第二条指令，这条指令要读取4011H这个地址，

那这次我们就来看一看，Cache在收到这个地址后，会做哪些操作。

开始收到4011这个地址后，首先应该找到这个地址对应的Cache行在哪里，

那它会用这一部分的地址来索引行，所以找到的还是表项1，

然后检查有效位，是1，代表这一行当中的数据是有效的。

但这并不能说明它所需要的数据就在这一行里面。接下来还需要比较标签位，

那把地址当中的高位40H和这个标签位进行比较，结果发现不相等，

那就说明这行当中的数据不是4011对应的那个数据块， 因此Cache还是需要向储存放出访存请求，

那发出的访存地址应该是什么呢？你先考虑一下。

那还有一点要注意的，就是等会取回的这个数据块，也还是需要放在这个表项1当中，

所以会覆盖现在Cache当中的这个数据块，而且等会还会把这个标签位也改成40H。

那假设现在Cache已经把对应的数据块从储存当中读回来了， 并且完成了对这个Cache行的替换操作，

那之后Cache就可以根据地址当中的最低四位找到对应的字节， 那现在还是第一个字节，

把这个字节B1H返回给CPU，就完成了这条指令的读操作了。

然后我们再来看第三条指令，那请先想一想访问这个地址的时候，Cache会

去检查哪一个表项，又会进行什么样的操作呢？请先思考一下， 然后我会快速地给出结果，就不再详细地解释了。

其实这一次

会访问表项3，然后是不命中的，然后读取内存地址3730

的数据块，并填到表项3中，然后返回其中第二个字节

C2H，这样就完成了第三条指令。然后我们再来看第四条指令，

这条指令的地址是401F，那么Cache会首先找到对应的行，

因为这部分地址是1，所以索引到的还是表项1 ，然后查看有效位，确定这一行的数据是有效的。

下一步是比较标签，那么都是40，所以标签也是匹配的，

这样就可以确认这个地址对应的数据就在这个Cache行当中，那我们就称为Cach- e命中，

最后再根据地址的最低几位，找到对应的字节， 那这个BFH就是CPU需要的这个数据了，

把这个数据返回之后，这条指令也就完成了。

那现在我们就了解了Cache读操作的几种典型的情况， 一种是没有命中，而且对应的表项是空的时候；

第二是没有命中，但是对应的表项已经被占用的情况，

还有就是命中了的情况。那看完了读，我们再来看一看写的情况。

当CPU要写一个数据的时候，也会先送到Cache， 那这时也会有命中和失效两种情况。

如果Cache命中，我们可以采用一种叫写穿透的策略，把这个数据写入

Cache中命中的那一行的同时，也写入主存当中对应的单元， 那这样就保证了Cache中的数据和主存中的数据始终是一致的。

但是因为访问主存比较慢，这样的操作效率是比较低的， 所以我们还可以用另一种策略叫做写返回，

那这时只需要把数据写到Cache当中，而不写回主存， 只有当这个数据块被替换的时候，才把数据写回主存。

那这样做的性能显然会好一些， 因为有可能会对同一个单元连续进行多次的写，

那这样只用将最后一次写的结果在替换时，写回主存就可以了，大大减少了访问主存的次数。

但是要完成这样的功能，Cache这个部件就会变得复杂得多， 那同样地，在Cache失效的时候，也有两种写策略，

一种叫做“写不分配”，那因为Cache失效，所以要写的那个存储单元不在Cach- e当中，

写不分配的策略就是直接将这个数据写到对应的主存单元里； 而另一种策略叫“写分配”，那就是先将这个数据

所在的数据块读到Cache里以后，再将数据写到Cache里。

那写不分配的策略实现起来是很简单的，但是性能并不太好，

而写分配的策略，虽然第一次写的时候操作复杂一些，

还是要将这个块读到Cache里以后再写入，看起来比写不分配要慢一点， 但是根据局部性的原理，

现在写过的这个数据块过一会很可能会被使用，所以提前 把它分配到Cache当中后，会让后续的访存性能大大提升。

因此在现代的Cache设计当中，写穿透和写不 分配这两种比较简单的策略往往是配套使用的，

用于那些对性能要求不高，但是希望设计比较简单的系统；

而大多数希望性能比较好的Cache，都会采用写返回和写分配这一套组合的策略。

那除此之外，在对Cache进行写的过程中，如何去查找分配和替换Cache中的

表项，都是和刚才介绍过的读操作的情形是一样的，就不再重复描述了。

高速缓存的基本原理并不复杂，

现在我们就可以构造出能够正常工作的高速缓存了。

但是如果希望高速缓存能够高效地工作，真正提升计算机的性能， 就还需要解决很多的细节问题，

之后，我们一一探索。

706-高速缓存的组织结构

高速缓存是一个非常精细的部件，

想让它高效地工作，就得在设计时，进行仔细地权衡。

想要设计出一个优秀的高速缓存部件， 我们就得从几个基本概念开始入手。

我们再来看一看Cache的访问过程。如果CPU向Cache发出了读请求，

那Cache就会检查对应的数据是否在自己内部，如果在，

我们就称为Cache命中。那有项性能指标就叫做命中率，

这说的是CPU所有访存请求中发生了Cache命中的请求所占的比例，

那如果命中就从Cache内部向CPU返回数据。

那从Cache中将命中的数据返回的时间，就称为命中时间， 这也是一个重要的性能参数。

现在的CPU当中，一级开始的命中时间大约是1到3 个周期，二级开始的命中时间大约是5到20个周期。

那如果数据不在Cache里，我们就称为Cache失效，那对应的就是失效率。

失效率和命中率加在一起，肯定是百分之一百。

那在失效之后，Cache会向主存发起读请求，那么等待

从主存中返回读数据的这个时间，我们就称为失效代价。

那现在通常需要在100到300个时钟周期之后，才能得到读的数据。

那要评价访存的性能，我们经常会用到平均访存时间这个指标，

这个指标就是用刚才介绍的那几个参数推算出来的。

平均访存时间就等于命中时间，加上失效代价乘以失效率，那想要提高访存的性能，

我们就得降低平均访存时间，那要做的就是分别降低这三个参数。

这就是提高访存性能的主要途径。

其中想要降低命中时间，就要尽量将Cache的容量做得小一些， Cache的结构也不要做得太复杂。

但是小容量的结构简单的Cache，又很容易发生失效，

这样就会增加平均访存时间。其中如果要减少失效代价，

要么是提升主存的性能，要么是在当前的高速缓存和主存之间再增加一级高速缓存。

那在新增的那级高速缓存当中，也需要面临这些问题。

所以这三个途径并不是独立的，它们是交织在一起， 相互影响。那我们先重点来看一看命中率这个因素。

如果有的Cache的命中率是97%，

我们假设命中时间是3个周期，失效代价是300个周期，

那么平均访存时间就是12个周期。那如果我们有一个方法

可以在不影响命中时间和失效代价的情况下，将命中率提高到99%， 那这时候的平均访存时间就降低到了6个周期。

所以虽然命中率只提高了2%，看起来并不起眼， 但是访存性能却提高了一倍，这是非常大幅度的提升。

所以对于现在的Cache来说，能够提高一点点命中率，都可以带来很好的性能提升。

那哪些因素会影响命中率呢?

或者我们反过来看，Cache失效会由哪些原因造成? 有一种叫做义务失效，

那从来没有访问过的数据块，肯定就不在Cache里， 所以第一次访问这个数据块所发生的失效，就称为义务失效，

义务失效是很难避免的。第二种失效原因称为容量失效，

那如果这个程序现在所需的数据块已经超过了这个Cache容量的总和，

那不管我们怎么去精巧地设计这个Cache，总会发生失效。

当然容量失效可以通过扩大Cache来缓解， 但是增加了Cache容量之后，一方面会增加成本，

另一方面可能也会影响到命中时间，所以也需要综合地考虑。

第三种失效，称为冲突失效，也就是在Cache并没有满的情况下，

因为我们将多个存储器的位置映射到了同一个Cache行，导致位置上的冲突而带来的失效。

那我们重点就来看一看如何解决这个问题。

那这个问题就成为Cache的映射策略。

那这是一块存储器的区域，我们还是按照每16个 字节一个数据块的形式把这块存储器的区域画出来，

所以这里标记的是这个数据块的起始地址， 每个地址之间正好相差16。那如果我们有一个8表象的Cache，

那么就采用我们之前介绍过的那种存放方法。

地址为0的数据块是要发在表象0当中， 而地址为080的数据块，也得放在表象0当中，

同样地址为100的数据块，也要存放在这个表象中。

这其实就是把内存分成每8个数据块为一组，任何一组当中的第0个

数据块，都会被放在表象0，第一个数据块都会被放在表象1， 那这样的Cache的映射策略就叫做直接映射。

它的优点在于硬件结构非常简单， 我们只要根据地址就可以知道对应的数据块应该放在哪个表象。

但是它的问题也很明显，如果我们在程序当中正好要交替地不断访问两个数据，

不妨称为数据A和数据B，那如果数据A在地址为0的这个 数据块当中，而数据B在地址080的这个数据块当中，

那在访问到数据A时，会把地址0的这个数据块调到Cache当中来，

然后把对应的数据A交给CPU，然后CPU又需要访问数据B，

其后开始发现，数据B不在Cache当中，所以又要把080对应的这个数据块调进来，

替换掉原有的表象0当中的数据，然后再将数据B交给CPU， 然后接下来CPU又访问到数据A，

那Cache又要把地址为0的这个数据块调进来，再次覆盖表象0。

那如果CPU不断地在交替访问数据A和数据B，

那这段时间的Cache访问每一次都是不命中的，这样的访存性能还不如没有Cache，

而这时Cache当中其他的行，都还是空着的，根本没有发挥作用。

所以这就是这个映射策略上的问题。

那为了解决这个问题，我们可以做一些改进，在不增加 Cache总的容量情况下，我们可以将这8个Cache行分为两组，

这就是二路组相联的Cache。

那这样刚才那种交替地访问数据A和数据B的情况，就不会有问题了，

因为CPU访问数据A时，就会把地址0对应的数据块放在这里，

而接下来在访问数据B时，就会把地址080对应的数据块放在这里，

然后再反复地访问数据A和数据B，都会在Cache中命中， 这样访存的性能就会非常好。

当然如果CPU在交替地访问这三个数据块当中的数据，

那么二路组相联的Cache又会出现连续不命中的情况，所以我们还可以对它进一步切分。

这就是一个四路组相联的Cache。

那我们是不是可以无限制地切分下去呢?

这倒是可以的，如果这个Cache总共只有8行，而我们把它分成八路组相联，

那也就是说，内存当中任一个数据块都可以放到这个Cache当中的任何一个行中，

而不用通过地址的特征来约定固定放在哪一个行。

那这样结构的Cache就叫做全相联的Cache。

这样的设计灵活性显然是最高的，但是它的控制逻辑也会变得非常的复杂。

我们假设CPU发了一个地址，Cache要判断这个地址是否在自己内部，

那它就需要把可能包含这个地址的Cache行当中的标签取出来进行比较，

对于直接映射的Cache，只需要取一个标签来比较就行， 二路组相联的时候，就需要取两个标签同时进行比较，

四路组相联的时候就需要取出四个标签来比较， 而在全相联的情况下，那就需要把所有行当中的标签都取出来比较。

这样的比较需要选用大量的硬件电路，既增加了延迟，又增加了功耗，

如果划分的路数太多，虽然有可能降低了失效率，但是却增加了命中时间，

这样就得不偿失了。而且话又说回来，增加了路数，还不一定能够降低失效率，

因为在多路组相联的Cache当中，同一个数据块可以放在不同的地方，

那如果这些地方都已经被占用了，就需要去选择一行替换出去， 这个替换算法设计得好不好，就对性能有很大的影响。

如果这个Cache选择替换出去的行，恰恰总是过一会就要使用到的那个数据块，

那这样性能的表现就会很差。现在常见的Cache替换算法有这几种，

最简单的是随机替换，这个性能显然不会很好。

然后还有轮转替换，也就是按照事先设定好的顺序依次地进行替换，

那如果是四路组相联，上一次替换了第0路，这一次就替换第1路，下一次就替换第2路。

这个从硬件设计设计上来说比较简单，但是性能也一般。

性能比较好的替换算法，是最近最少使用的替换算法，简称为LRU，

那它需要额外的硬件来记录访问的历史信息，

在替换的时候，选择距离现在最长时间没有被访问的那个Cache行进行替换，

在使用中，这种方法的性能表现比较好，但是其硬件的设计也相当的复杂。

所以映射策略和替换算法都需要在性能和实现代价之间进行权衡。

那我们再来看看一些Cache设计的实例， 在X86系列CPU当中，486是最早在CPU芯片内部集成了Cache的，

但它使用的是一个指令和数据共用的Cache， 这个Cache有一个很明显的缺点，那就是指令和数据的局部性会互相影响，

因为指令和数据一般是存放在内存中的不同区域，所以它们各自具有局部性。

> 利用在执行一个要操作大量数据的程序， 那这些数据就会很快地沾满Cache，把其中的指令都挤出去了，
>
> 那在这个时候，执行一条指令，取指的阶段很可能是Cache不命中，需要等待访问存储器，
>
> 那就需要花很长的时间，而在执行阶段，去取操作数时，却往往会
>
> 命中Cache，那虽然这段时间比较短，但是整个指令执行的时间还是很长。
>
> 所以到了后来的奔腾，就把指令和数据分成了两个独立的Cache， 这样它们各自的局部性就不会相互影响了，
>
> 现在大多数CPU的一级Cache都会采用这样的形式。
>
> 那这个是现在比较先进的Core
>
> i7，它内部采用了多级Cache的结构， 其中一级Cache是指令和数据分离的各32K个Byte，
>
> 采用了8路组相联的形式，命中时间是4个周期，
>
> 所以在CPU的流水线当中，访问Cache也需要占多个流水席。
>
> 那么在这个4核的i7当中，每个处理器核还有自己独享的二级Cache，
>
> 二级Cache就不再分成指令和数据两个部分了，
>
> 因为它的容量比较大，指令和数据之间的相互影响就不那么明显。
>
> 但是二级Cache的命中时间也比较长，需要11个周期，那i7 CPU的流水线
>
> 总共也就16级左右，肯定是没有办法和二级Cache直接协同工作的。
>
> 这也是为什么一级Cache不能做得很大的一个重要原因。
>
> 那在二级Cache之下，还有一个三级Cache，这是由四个核共享的，总共8兆个字节，
>
> 三级Cache采用了16路组相联的结构，
>
> 而且容量也很大，达到了8兆个字节，这又导致它的命中时间很长，需要30到40个周期，
>
> 但它这样的结构命中率会很高，这样就很少需要去访问主存了。
>
> 那我们可以看到这三级的Cache，它的命中时间从4个周期、11个周期到40个周期，
>
> 那我们再考虑到主存的100到300个周期， 这就可以看出，这个多级Cache加主存的结构就拉开了明显的层次，
>
> 在各自的设计时就可以有不同的侧重，相互配合来提升整个系统的性能。

> 高速缓存的研究已经持续了很长时间， 直到今天仍然是一个研究的热点，
>
> 那只不过之前的研究对象是CPU和主存之间的这一级高速缓存，
>
> 而现在的研究对象则是由多级高速缓存组成的这么一个多层次的结构，
>
> 那不管怎么样，高速缓存技术的研究给我们带来了在可控成本
>
> 之下，尽可能高的系统性能。

707-存储容量的计算

> 对于存储器，我们既关心它的性能，
>
> 也关心它的容量，那么现在，在存储器容量的计算方面，却有了一些争议。
>
> 这又是怎么回事呢？我们一起来看一看。

我们来看一个关于U盘的怪事，有一天我想把我电脑上的一些文件随身带走

我先查看了一下文件的大小总共有3.8GB， 于是我就找了一个4G的U盘，这应该可以轻松地存下了。

于是我插上U盘就开始复制文件。

结果，弹出了这么个对话框，说空间不足。

我明明是个4G的U盘，结果系统却说可用大小总共只有3.72GB， 存不下我这3.8G的文件。这到底是怎么回事呢?

于是我就点右键，看这个U盘的属性，确实显示是3.72GB。

这里系统也统计了这个U盘总共有多少个字节， 然后就自己算一算，4GB应该是多少个字节，那么学计算机的都知道，

所谓32位的CPU可以访问4Gbyte（位）内存空间。

这就是说，指，32位的信号总共可以产生2的32次方个地址。

你一次可以访问2的32次方个byte。所以4G个byte就等于2的32次方个b- yte，

也就相当于4乘以1024的3次方， 那总共应该是四十二亿九千四百多万个字节，

而这个u盘足足给我少个两亿九千多万个字节，这不是太明显的偷工减料嘛?

于是我就向生产厂商投诉，说你们给我的东西怎么少了这么多呢? 我们原本买了一个4GB的U盘，你上面标的也是4GB

为什么实际只有3.72GB呢？可没想到，人家慢条斯理地跟我说了一大番道理。

说我们这个就是4GB，是这么算的。

当然你看到的这个41的字节并没有错，但我们的计算方法是这样的， 这41个字节就大约等于4乘以一千的3次方，这就是4GB。

其实，我们还多送了你一百多万个字节呢，

你占了便宜，反倒来投诉我们偷工减料，这又是何道理?

这我哪能轻易罢休，计算机里明明应该用1024作为底数。

你怎么能1000作为底数呢，所以你应该给够我1乘以1024的3次方，这才是一个4G- B的U盘。

不然所谓32位的地址可以访问4GB的存储空间岂不成为了一句笑话? 于是我们双方各执一词，谁也说服不了谁。

那虽然刚才讲的这个故事是虚构的。

但是这样的争议却是真实存在的，那么就从头来看一看。

刚才提到的4GB，其中B，也就是byte， 是一个数据的单位，而G则是这个单位的前缀。

比如说，我们经常使用的长度1km也就是一公里，或者叫一千米。

那它就相当于1000m，m就是米，这是长度的单位，

而K就是一个单位前缀，用来指代1000。那么类似的，1KG就是一千克。

那我们在学习到的计算机的知识中也经常有这样的单位前缀。

比如说我们介绍高速缓存的容量，我们会说它是32KB，

这就等于32乘以1024，在这里K指的是2的十次方。

而M，也就是我们平常所说的兆，指的是2的20次方，

而G指的是2的30次方，因此我们说一个内存是4GB的，

这就是说它一共有4乘以1024的三次方这么多byte。

但是按刚才U盘的生产厂商的说法，4GB 的u盘则是指4乘以1000的三次方的byte。

因此，至少从市值看来，4GB的u盘和4GB的内存，

虽然它们用了同样的容量标记，但实际上它们真正的容量是不一样的。

另外，如果你用同样的方法检查硬盘的容量，你也会发现硬盘的容量，

似乎也没有它标称的那么多。这也是同样的原因。好，我们在计算机系统中检查硬盘的属性。

肯定会发现，它不到1T，那这个矛盾到底是怎么回事呢?

究竟是谁对谁错呢？另外除了存储的容量，还有一些别的地方，

也用到了这些前缀，那他们究竟又表达的是什么含义呢?

要确定这些单位前缀的含义，我们就得从国际单位开始说起。

第一个是温度的单位，kelvin，用k来表示。

第二个是时间的单位，秒，用s来表示。

第三个是长度的单位，米，用m来表示。

第四个是质量的单位，千克，用kg来表示。

这里我们要注意到，其实克也是一个单位，

而千则是一个单位前缀，另外还有几个其他的单位，我们就不一一说明了。

接着我们来看国际单位制的前缀，刚才我们已经提到了k这种前缀。

它所代表的是一千，也就是10的三次方，

因为一千的这个数量并不大，人的日常生活中经常会遇到，所以这个前缀很早就确定下来了

我们要注意的是这个k是小写的。

当然随着社会的发展进步，人们所需要记录和计算的数也越来越大，

这也需要有新的单位前缀。比k更大的就是M，就相当于1000的平方，

更大一级的是G，相当于1000的三次方。

再高一级的是T，是1000的四次方，这都是在1960年确定的。

也就是说在那个时代这个量级的单位的前缀就基本够用了。

当然在后来，这些前缀又不够用了的时候，我们又必须要加新的。

75年增加了P和E，分别是1000的5次方和6次方。

到了91年，又增加了Z和Y，分别是1000的7次方和8次方。

这就是目前最大的单位前缀了，那么有大也有小。

当我们想表达更小的数时，就需要有另一个方向的前缀。

比如说m就是1000的负1次方，也就是千分之一。

那mm就是我们常用的毫米，而ms就是毫秒。

这些也都是人所能直接感知的范围内，所以也在很早就确定下来了。

后来在60年确定了micro这个前缀， 在中文里称之为”微“，

对应的有微米、微秒、微克等等，那它是用一个希腊字母μ来表示的，

当然在手写的情况下，是没有问题的。

而如果计算机里来输入，则会有一些困难。如果条件允许，我们都应该用插入特殊字符等方式，

正确地输入μ这个字母，只有在确实无法输入的情况下， 也可以用英文字母”u“来代替。

那么同一年，它规定了n这个前缀， 也就是”纳“，现在的集成电路设计都是在纳米这个尺度上。

而1G赫兹的时钟频率对应的时钟周期， 也就是纳秒，那比n更小一级的前缀是p，

中文里译作”皮“，那由晶体管构成的逻辑门的延迟现在都是在皮秒量级的。

那再往后，64年由设了f和a这两个单位前缀， 分别是1000的负5次方和1000的负6次方，

再到了91年，又增加了z和y，分别是1000的负7次方和1000的负8次方。

那这些就是现在国际公认的单位和单位前缀。

而在计算机领域使用情况有所不同， 内部存储器并没有遵守国际单位制的要求，

而在外部存储器的容量标记上则是遵循了国际单位制的标准。

从这个角度来看，那似乎是外部存储器的这个标记方法才更合理一些。

那内部存储器为什么要采取1024为底呢？这个原因其实也很明确。

刚才我们也已经解释过，因为计算器内部采取二进制的计数，

其实最主要的就是CPU的运算器采用二进制的计算的方式，

比如说15位的地址就正好可以表示2的15次方那么多个字节，

这15位地址线所能表示的每一个地址都正好对应了sm中的一个字节，

这样在计算的时候就自然而然应该用1024为底，

而对于外部存储器，它和CPU的距离较远，不像高速缓存和内存那样，

由CPU直接的操作，所以他们并没有必要采用1024为底的计算方法。

而其他一些参数，比如数据传输率、时钟频率

虽然也在计算机内部使用，但它和刚才提到的地址宽度并没有直接的联系，

而都只是在说在一个单位的时间内要完成多少事情，比如说以太网的传输率

100兆bps是指每秒钟要传输100兆个比特，所以这里的兆，

指的就是1000的二次方，而CPU的主频则对应了时钟的信号每秒钟要翻转多少次，

那么3G赫兹就是指每秒钟有30亿个时钟周期。

因此我们可以看到，计算机当中大多数的单位前缀 都还是采用十进制的形式，并且符合国际单位制的要求。

只有在计算内部存储器容量时，采用了二进制的形式，这样的做法都是合理的，但唯一的问- 题在于，

只有采取1024为底的前缀，也都使用了 与国际单位制额前缀相同的字母，

针对这些单位前缀使用上的矛盾，也有一些官方的说法。

那美国国家标准协会以及电器和电子工程师协会， 在1968年出过标准，

关于K，它给了两个定义，一就是和国际单位制相同的，是指1000，

第二条，它特别说明了如果是用来形容计算机存储的容量，

那这个前缀又可以表示2的十次方也就是1024。

类似的，M，也有两个定义。

然而在现实中，大家仍然是各行其是，并没有按照它们的规定执行， 也就按照内存一套，外存一套，各说各话。

因此到后来，IEEE的标准委员会也通过美国国家标准技术研究员发布了声明，

大概意思是说，针对现状， IEEE的标准将会将会使用大家公认的国际单位制前缀的定义，

那么mega，也就是我们说的兆，也就是指100万，不再有2的20次方这样另一个定义

当然这里也说了，如果能够明确指出，是以2为底的， 也可以作为个例单独处理，直到有一种新的标准

专门用于二进制的单位前缀。

那现在这种单位前缀的标准也已经制定出来了，

是由国际标准化组织ISO和国际电工委员会IEC制定的， 比如说Ki指1024，

而Mi就是指1024的平方，Gi则是指1024的三次方， 以此类推，

因此现在标记为500GB的硬盘，如果要对应到计算机 内部存储器所标记的存储容量，

应该是466GiB，好，我们现在常说的标记为512MB

的内存，如果想不引起混淆，那就应该标记出512MiB， 在这一套标准的影响下，

就不再有混淆的问题了

本节小结：存储器容量计算

标准现在已经有了， 但是执行的怎么样，那就是另一回事了。

现在的CPU和内存的厂商， 好像也没有太大的动力去改变原有的存储容量标记方式。

但是对于我们来说，已经了解了它内部的真相到底是怎么回事。

至少，不会把我们自己给搞糊涂了。