多道程序调度

多道程序设计是在计算机内存中同时存放几道相互独立的程序（只有将程序放到内存 CPU 才会执行），使它们在管理程序控制之下，相互穿插地运行（内存中多道程序轮流地或时获分地占有 CPU），交替地执行（单 CPU 情况），使他们共享 CPU 和系统中的各种资源。当某一程序因为某种原因不能继续执行时（如等待外部设备输入输出数据或者其他中断处理），操作系统的管理程序将会让 CPU 执行内存中的另一道程序，如此可以相对减少 CPU 和其他外部设备的空闲时间（即处于忙碌状态），从而提高计算机的使用效率。

优点：

（1）提高 CPU 的利用率；

（2）提高内存和 I/O 设备的利用率；

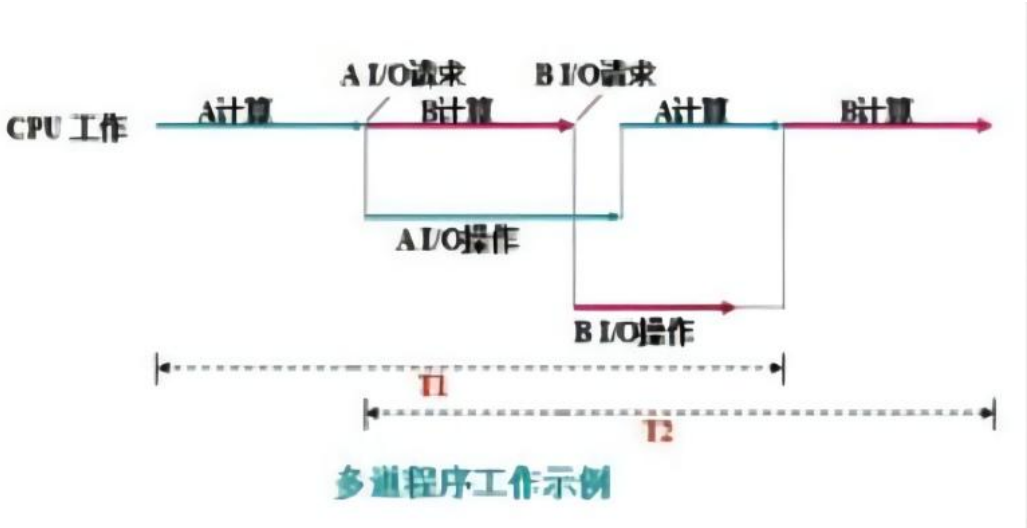
（3）增加系统吞吐量。

特征：

（1）多道：计算机中同时存放几道相互独立的程序；

（2）宏观上并行：同时进入系统的几道程序都处于运行过程中，即它们先后开始六各自的执行，但都未运行完毕；

（3）微观上串行：内存中的多道程序轮流地或分时地占有 CPU，交替地执行（单 CPU 情况）。



虚拟内存技术

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

现代所有用于一般应用的操作系统都对普通的应用程序使用虚拟内存技术，老一些的操作系统，如 DOS 和1980年代的 Windows，或者那些1960年代的大型机，一般都没有虚拟内存的功能。

直接使用物理内存的状态下就会产生一些问题：

1. **内存空间利用率的问题**

   各个进程对内存的使用会导致内存碎片化，当要用 malloc 分配一块很大的内存空间时，可能会出现虽然有足够多的空闲物理内存，却没有足够大的连续空闲内存这种情况，东一块西一块的内存碎片就被浪费掉了。

2. **读写内存的安全性问题**

   物理内存本身是不限制访问的，任何地址都可以读写，而现代操作系统需要实现不同的页面具有不同的访问权限，例如只读的数据等等。

3. **进程间的安全问题**

   各个进程之间没有独立的地址空间，一个进程由于执行错误指令或是恶意代码都可以直接修改其它进程的数据，甚至修改内核地址空间的数据，这是操作系统所不愿看到的。

4. **内存读写的效率问题**

   当多个进程同时运行，需要分配给进程的内存总和大于实际可用的物理内存时，需要将其他程序暂时拷贝到硬盘当中，然后将新的程序装入内存运行。由于大量的数据频繁装入装出，内存的使用效率会非常低。

虚拟内存技术成功的解决了直接使用物理内存会出现的问题，比如物理内存中离散式存储，虚拟内存中连续存储解决了物理内存碎片化资源利用率过低的问题；每个进程只能访问自己独立的用户空间而内核空间是共用的解决了进程间的安全问题；缺页异常和选择牺牲页的算法提高了内存读写的效率等等。

内存一致性

对于单个处理器，很容易发现独立操作可以以并行内存操作按程序顺序执行，例如，读取按程序顺序返回最后一次写入的值。

对称多处理器 (SMP) 系统的多种内存一致模型

- **顺序一致（Sequential consistency）**：同一个线程的原子操作还是按照 happens-before 关系，但不同线程间的执行关系是任意。
- **松弛一致（Relaxed consistency，允许某种类型的重排序）**：如果某个操作只要求是原子操作，除此之外，不需要其它同步的保障，就可以使用 Relaxed ordering。程序计数器是一种典型的应用场景。
- **弱一致（Weak consistency）**：读写任意排序，受显式的内存屏障限制。

如果任何执行的结果与所有处理器的操作以某种顺序执行的结果相同，并且每个单独处理器的操作按程序建立的顺序出现，则称多处理器系统是顺序一致的。

1 内存模型介绍

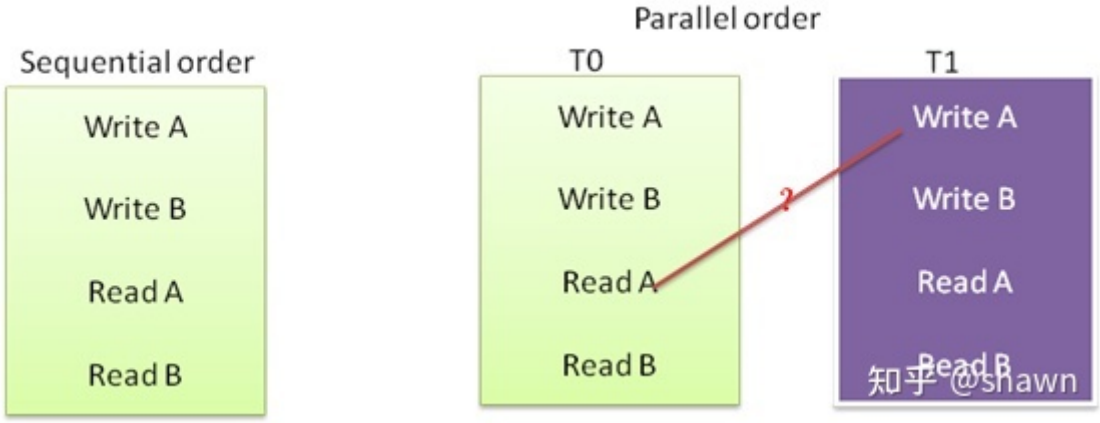
1.1 Sequential 内存一致性模型

上个世纪70年代，Lamport 提出了顺序一致性（Sequential consistency）的概念，他把基本不可实现的全局精确时钟要求转换到了局部时钟要求。Sequential 模型有三个特点：

（1） 所有的内存访问都是原子操作（中间不应该有write buffer和 cache）。

（2） CPU 核内的访问**严格按照程序代码顺序**进行，对于顺序执行 (in-order) CPU 严格按照程序要求的顺序执行对内存的存取（load和store）操作，对于乱序执行（out-of-order）cpu 也需要遵循该要求：严格按照程序要求的顺序完成对 load 和 store 指令的提交和退休(retire)。

（3） 多核之间代码可以以任意的顺序进行交织运行（但每个核内还是严格的顺序执行）



在 sequential 模型下，上述问题的运行结果就只能是先 A=1 ,然后才发生 flag=1 ,所以不可能打印出 A=0 的结果。

Sequentia l模型的优点是实现简单（对硬件和软件人员都简单），但问题是性能太差，无法发挥硬件的特性, 所以现在的 CPU 几乎无一例外地偏离了 Sequential 模型。

1.2 Relaxed 内存一致性模型

为了表达的方便，我们用 X->Y 来表示先必须完成 X 操作后才能执行 Y 操作。

前面介绍的 Sequential 模型就是要求严格保持 4 种可能的内存读写顺序场景（即使这些操作之间没有控制上的，数据上的，流水线上的依赖关系）：

宽松模型的关键思想是允许乱序执行读取 (load) 和写入 (store) 操作,通过使用同步操作来实现处理器之间的同步。宽松模型的方式可以多种多样的，我们可以根据放松了哪种读取和写入顺序来进行分类。

（1） 放松W->R顺序：我们就得到了 强顺序模型 TSO (total store ordering)，它允许 CPU 先执行读操作然后在执行写操作而不严格按照代码的指示顺序来进行。由于这种模型保持了写入操作之间的顺序，所以很多在 Sequential 模型下能够运行的代码也能在TSO模型下正常运行。

（2） 放松W->W顺序：我们就得到了 PSO (partial store ordering) 模型，允许多个写操作也被打乱顺序。

（3） 放松 R->W 和 R->R 顺序：将会得到很多模型，包括弱内存模型 WMO（Weak Memory Ordering），released 模型，等。

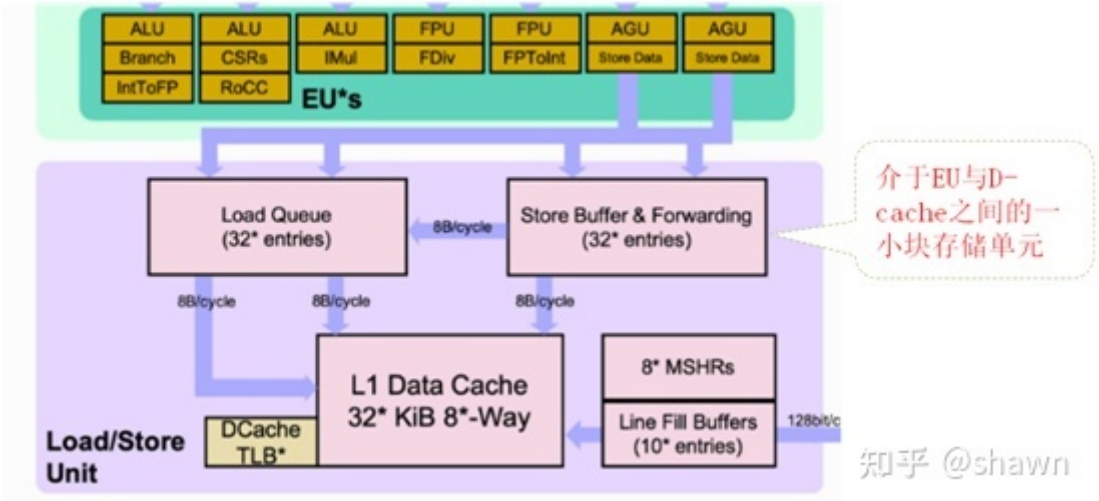
通过放松这些顺序约束，处理器性能可以得到显著的提升。

【注：这些顺序放松但依旧保证了在本 CPU 与程序代码一致的存取顺序，但在其它 CPU 节点看来顺序就可能被打乱了】

2 内存模型背后的原因

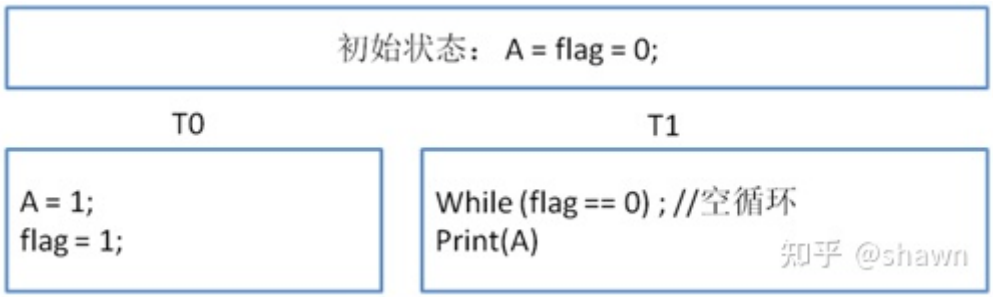
我们知道 CPU 访问主 memory 需要的开销非常大（上100个cycles），如果 CPU 想存储一个数据到 memory 而该数据又不在 data cache 中，则这个过程是相当耗时的，大大地降低了 CPU 性能。为了进一步加快内存访问速度，CPU 设计中引入了 store buffer 这个部件，store buffer是位于执行单元（比如浮点执行单元FPU, 算术逻辑单元ALU等）与 data cache之间的小一块存储单元。(借用一下

BOOMv3中的一个架构图)



使用 store buffer 的好处不仅在于加快存储的速度，另外还可以利用 store buffer 实现乱序执行以及实现推断执行 (speculative execution) 失败后的回退 roll back，等

当CPU需要写(store)数据时, 数据不直接写到 memory 或者 cache 中，而是先写到store buffer 中，store buffer 负责后续以某种次序写入 L1 data Cache 。我们还是用前面那个例子来解释一下什么情况下会输出另外一种结果：



假设 T0 在 CPU0 上执行，T1 在 CPU1 上执行：

- CPU0 将 A=1 写入 store buffer。
- CPU0 将 flag=1 写入 store buffer。
- CPU1 读取 flag 的值，由于 flag 新值还在 CPU0 的 store buffer 里面，所以 CPU1 看到的 flag 值依然是 0 。所以 CPU1 继续自旋等待。
- CPU0 的 store buffer 将 flag 的值写入 Cache，此时 A 的值依然在 store buffer 里面没有更新到 Cache（比如因为 A 没有在 cache 中而 flag 在 Cache 中，cache 的一致性保证了 CPU1 处看到的 flag=1）。
- CPU1 发现 flag 的值是1，退出循环。然后读取 A 的值，此时 A 的值是旧的数据，也就是 0。因为 A 的新值还在 CPU0 的 store buffer 里面，所以打印出 A=0。

####

缓存一致性

缓存的基本定律

如果我们只处理读操作，那么事情会很简单，因为所有级别的缓存都遵守以下规律，我称之为：

> ***基本定律***：在任意时刻，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

一旦我们允许写操作，事情就变得复杂一点了。这里有两种基本的写模式：直写（write-through）和回写（write-back）。直写更简单一点：我们透过本级缓存，直接把数据写到下一级缓存（或直接到内存）中，如果对应的段被缓存了，我们同时更新缓存中的内容（甚至直接丢弃），就这么简单。这也遵守前面的定律：缓存中的段永远和它对应的内存内容匹配。

回写模式就有点复杂了。缓存不会立即把写操作传递到下一级，而是仅修改本级缓存中的数据，并且把对应的缓存段标记为“脏”段。脏段会触发回写，也就是把里面的内容写到对应的内存或下一级缓存中。回写后，脏段又变“干净”了。当一个脏段被丢弃的时候，总是先要进行一次回写。回写所遵循的规律有点不同。

> ***回写定律***：当所有的脏段被回写后，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

换句话说，回写模式的定律中，我们去掉了“在任意时刻”这个修饰语，代之以弱化一点的条件：要么缓存段的内容和内存一致（如果缓存段是干净的话），要么缓存段中的内容最终要回写到内存中（对于脏缓存段来说）。

直接模式更简单，但是回写模式有它的优势：它能过滤掉对同一地址的反复写操作，并且，如果大多数缓存段都在回写模式下工作，那么系统经常可以一下子写一大片内存，而不是分成小块来写，前者的效率更高。

问题

当计算机加入了缓存，cpu 读取数据的时候首先会从缓存读取，如果缓存中不存在，则从主存读取，同时把该数据加入到缓存中，这样下次再次使用的时候就可以直接从缓存中读取。当修改了某些数据，先把修改后的数据写入到缓存中，然后再刷到主存中，以此来提高效率。

这样的过程在单线程的环境下是不会出现问题的，但是在多线程环境下就会出现问题，现在的计算机几乎都是多个核心的，多个线程运行在不同的核心中，每个核心都有自己的缓存。这时就会出现多个 cpu 同时修改了同一个数据的问题，这就是著名的缓存一致性问题。

早期 cpu 中，为了解决缓存一致性问题，计算机厂商们通过在消息总线上加锁来解决的，也就是说同时只有一个cpu能操作同一块数据。这样的后果就是，加锁期间其他 cpu 无法访问内存，导致效率低下，因此出现了第二种解决方案，就是通过缓存一致性协议来解决缓存一致性问题。

缓存一致性协议（MESI）

MESI 是取自缓存行（Cache line，缓存中存储数据的单元）中数据的四种状态的英文首字母，缓存行中数据具有四种状态，它们分别是：

> - Modified（修改）：数据有效，数据被修改了，和内存中数据不一致，数据只存在于本Cache中。
> - Exclusive（独享）：数据有效，数据和内存中的数据一致，数据只存在于本Cache中。
> - Shared（共享）：数据有效，数据和内存中的数据一致，数据存在多个Cache中。
> - Invalid（无效）：数据无效，一旦数据被标记为无效，那效果就等同于它从来没被加载到缓存中。

例如现在有一个 project，需要四个人（core）参与到 coding 工作，为了版本控制，我们将代码传到 github（内存）上，四个人分别是甲乙丙丁，对应着 core0，core1，core2 和 core3，每个人都有自己的计算机（local cache），平时在自己的计算机上 coding。

现在，假设甲从 github（内存）上下载（load）了某一个类文件（cache block）到他的计算机（甲的 local cache），而乙丙丁的计算机并没有这个类文件（cache block）的副本，这时候甲独占着这个类文件的副本，并且和 github（内存）上一样，所以此时这个类文件的状态是exclusive。

假设这时候丁也从 github上下载了这个类文件到他的计算机上，这时候，这个类文件在甲和丁两个人的计算机上都有副本，并且和github 上一样，所以此时这个类文件的状态是 shared。

假设现在丁在自己的计算机上对这个类文件进行了修改，此时，甲的计算机上的这个类文件就过时了，成了 invalid 状态，而这个类文件的最新副本在丁的计算机上，且只有丁的计算机上有这个类文件的最新副本，同时这个类文件和 github 上的内容不一致，是脏的（dirty），所以这个类文件的状态就是 modified。

MESI状态迁移

如果当前core的cache block状态是invalid

那么当前core读取这个cache block（local read）的时候，考虑两种情况：
　　如果其他core的cache上没有这个cache block的副本，那么从内存中加载，此时只有当前core拥有和内存中一致的cache block副本，所以转移到exclusive状态；
　　如果其他core的cache上有这个cache block的副本，那么从LLC中加载，此时不止一个core拥有和内存中一致的cache block副本，所以转移到shared状态。

当前core修改这个cache block（local write）的时候，这个cache block的内容与内存中不一致（dirty），并且只有当前core拥有这个最新的副本，所以转移到modified状态；

其他core读取这个cache block（remote read）以及其他core修改这个cache block（remote write）的时候，对当前core的cache中的cache block没有任何影响。

如果当前core的cache block状态是exclusive，

那么当前core读取这个cache block（local read）的时候，当前core的cache block的状态还是exclusive，因为local read不会让其他core拥有这个cache block的副本，并且也不会对这个cache block进行修改

当前core修改这个cache block（local write）的时候，因为对这个cache block进行了修改，也就和内存中不一致了，所以当前core的cache block转移到modified状态

其他core读取这个cache block（remote read）的时候，其他core的cache中也会拥有这个cache block的副本，所以此时当前core的cache block转移到shared状态

其他core修改这个cache block（remote write）的时候，当前core的cache block数据过时，不再有效，所以当前core的cache block转移到invalid状态

如果当前core的cache block处于shared状态

那么当前core对cache block的读取（local read）以及其他core对cache block的读取（remote read）对当前core的cache block状态无影响，所以还是shared状态

当前core对cache block的修改（local write）会使得当前core的cache block内容与内存中不一致，所以会转移到modified状态

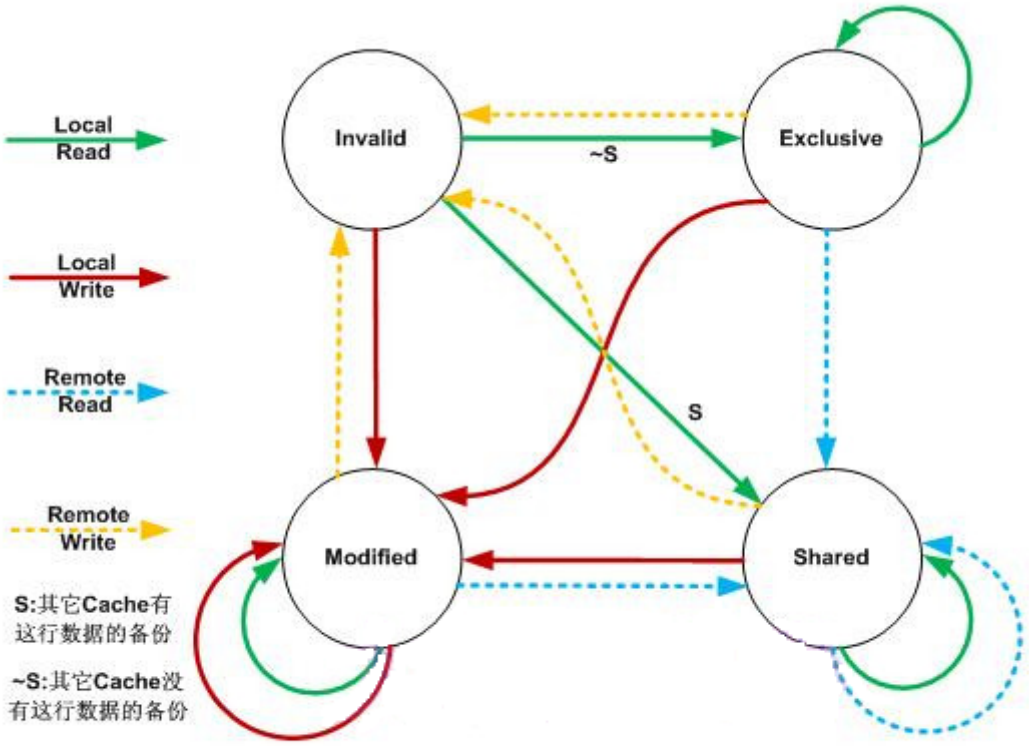
其他core对cache block的修改（remote write）会使得当前core的cache block内容过时失效，所以转移到invalid状态

如果当前core的cache block处于modified状态

那么当前core对cache block的读取（local read）以及当前core对cache block的修改（local write）对当前core的cache block状态不产生影响，所以还是modified状态

其他core对cache block的读取（remote read）会导致其他core拥有这个cache block的最新副本，所以此时不再是当前core独占，所以转移到shared状态

其他core对cache block的修改（remote write）会导致当前core的cache block内容过时失效，所以转移到invalid状态



MESI协议状态迁移图

闪存

闪存是一种电可擦除可编程只读存储器（Electrically Erasable ProgrammableRead Only Memory，EEPROM），具有非易失、读写速度快、抗震、低功耗、体积小等特性，目前己广泛应用于嵌入式系统、航空航天、消费电子等领域。[1]

闪存存储器主要分为NOR型和NAND型两种，NOR型闪存有独立的地址线和数据线，它支持按位进行访问，具有高可靠性且随机读取速度较快，但NOR闪存的擦除和写操作速度较慢、容量小、价格昂贵，主要用于存储程序代码并在内存中直接运行。NOR闪存在手机上得到了广泛的应用。NAND闪存相对于NOR型闪存拥有更大的容量，适合进行数据存储。

此外，它类似于传统的机械硬盘，对于小数据块的操作较慢，对于大数据块操作较快。NAND闪存根据其芯片单元所能存储的比特位数又可分为单级晶胞(SLC)和多级晶胞(MLC)两类。MLC技术具有显著的存储密度优越性，相对于SLC每个单元仅能存储1位比特，MLC可以存储多位比特，但MLC在速度和可靠性方面还有一定的提升空间。

原理

闪存结合了EPROM的高密度和EEPROM结构的变通性的优点。

EPROM是指其中的内容可以通过特殊手段擦去，然后重新写入。其基本单元电路如下图所示。常采用浮空栅雪崩注入式MOS电路，简称为FAMOS。它与MOS电路相似，是在N型基片上生长出两个高浓度的P型区，通过欧姆接触分别引出源极S和漏极D。在源极和漏极之间有一个多晶硅栅极浮空在 绝缘层中，与四周无直接电气联接。这种电路以浮空栅极是否带电来表示存1或者0，浮空栅极带电后（例如负电荷），就在其下面，源极和漏极之间感应出正的导电沟道，使MOS管导通，即表示存入0.若浮空栅极不带电，则不能形成导电沟道，MOS管不导通，即存入1。

EEPROM基本存储单元电路的工作原理如图2.2所示。与EPROM相似，它是在EPROM基本单元电路的浮空栅极的上面再生成一个浮空栅，前者称为第一级浮空栅，后者称为第二级浮空栅。可给第二级浮空栅引出一个电极，使第二级浮空栅极接某一电压VG。若VG为正电压，第一浮空栅极与漏极之间产生隧道效应，使电子注入第一浮空栅极，即编程写入。若使VG为负电压，强使第一浮空栅极的电子散失，即擦除。擦除后可重新写入。

闪存的基本单元电路与EEPROM类似，也是由双层浮空栅MOS管组成。但是第一层栅介质很薄，作为隧道氧化层。写入方法与EEPROM相同，在第二级浮空栅加正电压，使电子进入第一级浮空栅。读出方法与EPROM相同。擦除方法是在源极加正电压利用第一级浮空栅与漏极之间的隧道效应，将注入到浮空栅的负电荷吸引到源极。由于利用源极加正电压擦除，因此各单元的源极联在一起，这样，擦除不能按字节擦除，而是全片或者分块擦除。随着半导体技术的改进，闪存也实现了单晶体管设计，主要就是在原有的晶体管上加入浮空栅和选择栅，

NAND闪存阵列分为一系列128kB的区块(block)，这些区块是NAND器件中最小的可擦除实体。擦除一个区块就是把所有的位(bit)设置为“1”(而所有字节(byte)设置为FFh)。有必要通过编程，将已擦除的位从“1”变为“0”。最小的编程实体是字节(byte)。一些NOR闪存能同时执行读写操作(见下图1)。虽然NAND不能同时执行读写操作，它可以采用称为“映射(shadowing)”的方法，在系统级实现这一点。这种方法在个人电脑上已经沿用多年，即将BIOS从速率较低的ROM加载到速率较高的RAM上。

分类

Flash又分NAND Flash和NOR Flash，NOR型存储内容以编码为主，其功能多与运算相关；NAND型主要功能是存储资料，如数码相机中所用的记忆卡。

现在大部分的SSD都是用来存储不易丢失的资料，所以SSD存储单元会选择NAND Flash芯片。这里我们讲的就是SSD中的NAND Flash芯片。

Nor Flash：主要用来执行片上程序

优点：具有很好的读写性能和随机访问性能，因此它先得到广泛的应用；

缺点：单片容量较小且写入速度较慢，决定了其应用范围较窄。

NAND Flash：主要用在大容量存储场合

优点：优秀的读写性能、较大的存储容量和性价比，因此在大容量存储领域得到了广泛的应用；

缺点：不具备随机访问性能。

NAND 规则

- Flash都不支持覆盖，即**写入操作只能在空或已擦除的单元内进行。**更改数据时，将整页拷贝到缓存（Cache）中修改对应页，再把更改后的数据挪到新的页中保存，将原来位置的页标记为无效页；
- **以page为单位写入，以Block为单位擦除**；擦除Block前需要先对里面的有效页进行搬迁。
- 每个Block都有擦除次数限制（有寿命），擦除次数过多会成为坏块（bad block）

写入放大

一个page有三种状态：空；无效数据；有效数据。

向SSD写入数据时，以page为单位，但只能向空的page里写。如果要向存有无效数据的page里写，则需要先进行擦除，将无效数据的page变为空page，再写入。但是擦除数据时，是以block为单位的。

举例：向一个block写入一个page的新数据时，一个block里已经没有空的page了，只有有效数据以及无效数据的page，所以主控就把新数据以及有效数据读到缓存形成一个新的block，再擦除原先的block，再把新block写回去。这个操作带来的写入放大就是: 实际写一个page的4KB的数据，造成了整个block共512KB的写入操作，这就是128倍的写入放大（WA，Write Amplification）。写入放大造成了SSD实际对NAND的写入量大于主机要求的写入量。

同时，原本仅须一步写入page的操作变成：缓存读取新数据以及有效数据→闪存擦除→缓存写入，造成延迟大大添加，速度变慢。
所以说，写入放大是影响SSD随机写入性能和寿命的关键因素。以100%随机4KB来写入，眼下的大多数SSD主控，在最坏的情况下WA能够达到100以上。假设是100%持续的从低LBA写入到高LBA的话，WA能够做到1，实际使用中写入放大会介于这两者之间。

当一个block里的page只有空、有效数据两种状态时，在写入数据的时候不需要进行擦除，trim就是通知SSD在空闲状态时，将无效数据擦除，以减少SSD在写入数据时需要进行的数据擦除工作。

写入放大倍数 = 闪存写入数据量 / 主控写入数据量 = 实际写入数据量 / 要求写入数据量

一些知识

HDD

HDD是指机械硬盘，是传统普通的硬盘。

介质：采用磁性碟片来存储。

包括：盘片、磁头、磁盘旋转轴及控制电机、磁头控制器、数据转接器、接口、缓存。

机械式硬盘最大速率约为100MB/s，由于容易发热等原因已经无法再进一步提升速度，所以引入了固态硬盘

SSD

SSD（Solid State Drives）是固态硬盘。

介质：采用闪存颗粒来存储。

包括：控制单元、存储单元（DRAM芯片/FLASH芯片）。

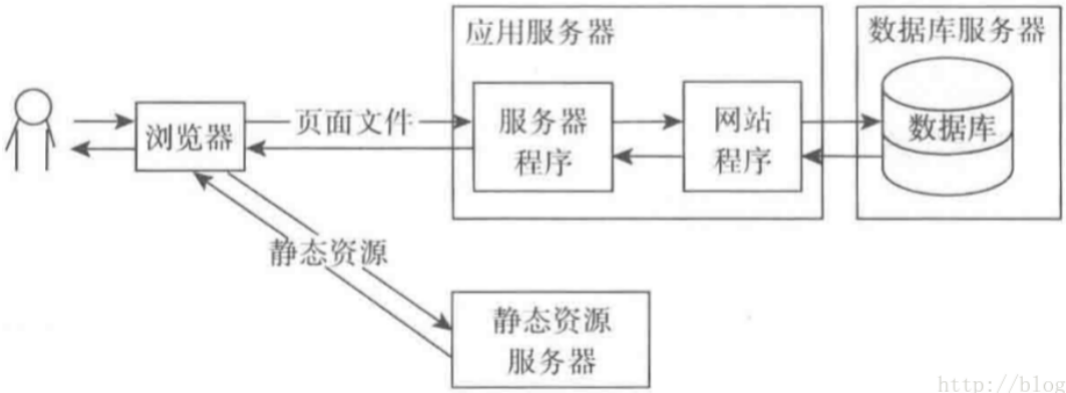
性能区别

HDD是机械式寻找数据，所以**防震**远低于SSD，**数据寻找时间**也远低于SSD

高并发的解决方案

1.应用和静态资源分离

刚开始的时候应用和静态资源是保存在一起的，当并发量达到一定程度的时候就需要将静态资源保存到专门的服务器中，静态资源主要包括图片、视频、js、css和一些资源文件等，这些文件因为没有状态所以分离比较简单，直接存放到响应的服务器就可以了，一般会使用专门的域名去访问。
通过不同的域名可以让浏览器直接访问资源服务器而不需要再访问应用服务器了。架构图如下：



2.页面缓存

页面缓存是将应用生成的页面缓存起来，这样就不需要每次都生成页面了，从而可以节省大量的CPU资源，如果将缓存的页面放到内存中速度就更快了。如果使用Nginx服务器就可以使用它自带的缓存功能，当然也可以使用专门的Squid 服务器。页面缓存的默认失效机制一班都是按缓存时间处理的，当然也可以在修改数据之后手动让相应的缓存失效。

页面缓存主要是使用在数据很少发生变化的页面，但是很多页面是大部分数据都很少发生变化，而其中很少一部分数据变化频率却非常高，比如说一个显示文章的页面，正常来说完全可以静态化，但是如果文章后面有"顶"和"踩"的功能而且显示的有响应的数量，这个数据的变化频率就比较高了，这就会影响静态化。这个问题可以用先生成静态页面然后使用Ajax来读取并修改响应的数据，这样就可以一举两得来，既可以使用页面缓存也可以实时显示一些变化频率高的数据来。

其实大家都知道，效率最高、消耗最小的就是纯静态化的html页面，所以我们尽可能使我们的网站上的页面采用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，我们无法全部手动去挨个实现，于是出现了我们常见的信息发布系统CMS，像我们常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。

除了门户和信息发布类型的网站，对于交互性要求很高的社区类型网站来说，尽可能的静态化也是提高性能的必要手段，将社区内的帖子、文章进行实时的静态化，有更新的时候再重新静态化也是大量使用的策略，像Mop的大杂烩就是使用了这样的策略，网易社区等也是如此。

同时，html静态化也是某些缓存策略使用的手段，对于系统中频繁使用数据库查询但是内容更新很小的应用，可以考虑使用html静态化来实现，比如论坛中论坛的公用设置信息，这些信息目前的主流论坛都可以进行后台管理并且存储再数据库中，这些信息其实大量被前台程序调用，但是更新频率很小，可以考虑将这部分内容进行后台更新的时候进行静态化，这样避免了大量的数据库访问请求。

3.集群与分布式

集群是每台服务器都具有相同的功能，处理请求时调用那台服务器都可以，主要起分流作用。

分布式是将不同的业务放到不同的服务器中，处理一个请求可能需要用到多台服务器，这样就可以提高一个请求的处理速度，而且集群和分布式也可以同时使用。

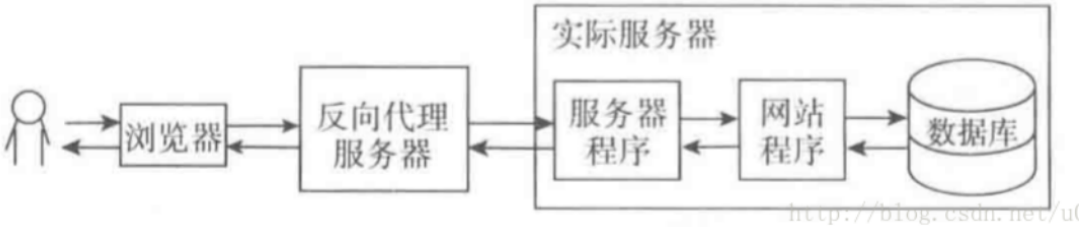
集群有两个方式：一种是在静态资源集群。另一种是应用程序集群。静态资源集群比较简单。应用程序集群在处理过程中最核心的问题就是Session 同步问题。

Session 同步有两种处理方式：一种是在Session 发生变化后自动同步到其他服务器，另一种就是用个程序统一管理Session。所有集群的服务器都使用同一个Session，Tomcat 默认使用就是第一种方式，通过简单的配置就可以实现，第二种方式可以使用专门的服务器安装Mencached等高效的缓存程序统一来管理session，然后再应用程序中通过重写Request并覆盖getSession 方法来获取制定服务器中的Session。

对于集群来说还有一个核心的问题就是负载均衡，也就是接收到一个请求后具体分配到那个服务器去处理的问题，这个问题可以通过软件处理也可以使用专门的硬件解决。

4. 反向代理

反向代理指的是客户端直接访问的服务器并不真正提供服务，它从别的服务器获取资源然后将结果返回给用户。



反向代理服务器和代理服务器的区别

代理服务器的作用是代我门获取想要的资源然后将结果返回给我们，所要获取的资源是我门主动告诉代理服务器的，比如，我门想访问Facebook，但是直接访问不了，这时就可以让代理服务器访问，然后将结果返回给我们。

反向代理服务器是我门正常访问一台服务器的时候，服务器自己去调用了别的服务器资源并将结果返回给我们，我门自己并不知道。

代理服务器是我们主动使用的，是为我们服务的，他不需要有自己的域名；反向代理服务器是服务器自己试用的，我门并不知道，它有自己的域名，我门访问它和访问正常的网址没有任何区别。

反向代理服务器主要有三个作用：

1. 可以作为前端服务器跟实际处理请求的服务器集成；
2. 可以做负载均衡
3. 转发请求，比如说可以将不同类型的资源请求转发到不同的服务器去处理。

5. CDN

cdn 其实是一种特殊的集群页面缓存服务器，他和普通集群的多台页面缓存服务器相比，主要是它存放的位置和分配请求的方式有点特殊。CDN 服务器是分布在全国各地的，当接收到用户请求后会将请求分配到最合适的CDN服务器节点获取数据。比如联通的用户分配到联通的节点，上海的用户分配到上海的节点。

CDN的每个节点其实就是一个页面缓存服务器，如果没有请求资源的缓存就会从主服务器获取，否则直接返回缓存的页面。

CDN分配请求（负载均衡）的方式是用专门的CDN域名解析服务器在解析域名的时候就分配好的。一般的做法是在ISP哪里试用CNAME将域名解析到一个特定的域名，然后再将解析到的那个域名用专门的CDN服务器解析道相应的CDN节点。如图。

第二步访问CDN的DNS服务器是应为CNAME记录的目标域名使用NS记录指向了CDN的DNS服务器。CDN的每个节点可能也是集群了多台服务器。

6. 底层的优化

前面说的所有都是架构都是建立在最前面介绍的基础结构之上的。很多地方都需要通过网络传输数据，如果可以加快网络传输的速度，那将会让整个系统得到改善。

7. 数据库集群和库表散列

大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是我们需要使用数据库集群或者库表散列。

在数据库集群方面，很多数据库都有自己的解决方案，Oracle、Sybase等都有很好的方案，常用的MySQL提供的Master/Slave也是类似的方案，您使用了什么样的DB，就参考相应的解决方案来实施即可。

上面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用DB类型的限制，于是我们需要从应用程序的角度来考虑改善系统架构，库表散列是常用并且最有效的解决方案。我们在应用程序中安装业务和应用或者功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再按照一定的策略对某个页面或者功能进行更小的数据库散列，比如用户表，按照用户ID进行表散列，这样就能够低成本的提升系统的性能并且有很好的扩展性。sohu的论坛就是采用了这样的架构，将论坛的用户、设置、帖子等信息进行数据库分离，然后对帖子、用户按照板块和ID进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统随时增加一台低成本的数据库进来补充系统性能。

8. 小结

网站架构的整个演变过程主要是围绕大数据和高并发这两个问题展开的，解决方案主要分为使用缓存和多资源两种类型。多资源主要指多存储（包括多内存）、多CPU和多网络，对于多资源来说又可以分为单个资源处理一个完整的请求和多个资源合作处理一个请求两种类型，如多存储和多CPU中的集群和分布式，多网络中的CDN和静态资源分离。理解了整个思路之后就抓住了架构演变的本质，而且自己可能还可以设计出更好的架构。

处理高并发的六种方法

1：系统拆分，将一个系统拆分为多个子系统，用dubbo来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，这样就可以抗高并发。

2：缓存，必须得用缓存。大部分的高并发场景，都是读多写少，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟人家redis轻轻松松单机几万的并发啊。没问题的。所以你可以考的虑考虑你的项目里，那些承载主要请求读场景，怎么用缓存来抗高并发。

3：MQ(消息队列)，必须得用MQ。可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改，疯了。那高并发绝对搞挂你的系统，人家是缓存你要是用redis来承载写那肯定不行，数据随时就被LRU(淘汰掉最不经常使用的)了，数据格式还无比简单，没有事务支持。所以该用mysql还得用mysql啊。那你咋办？用MQ吧，大量的写请求灌入MQ里，排队慢慢玩儿，后边系统消费后慢慢写，控制在mysql承载范围之内。所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用MQ来异步写，提升并发性。MQ单机抗几万并发也是ok的。

4：分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来抗更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高sql跑的性能。

5：读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，主库写入，从库读取，搞一个读写分离。读流量太多的时候，还可以加更多的从库。

6：solrCloud:
SolrCloud(solr 云)是Solr提供的分布式搜索方案，可以解决海量数据的 分布式全文检索，因为搭建了集群，因此具备高可用的特性，同时对数据进行主从备份，避免了单点故障问题。可以做到数据的快速恢复。并且可以动态的添加新的节点，再对数据进行平衡,可以做到负载均衡：

多线程比单线程更慢的例子

例子一:

单核单处理器,开一个线程跑循环输出10万条打印信息,开100个线程输出10万条打印信息.

后者比前者慢,因为输出端是临界资源,线程抢占的时间大,单线程则无需抢占

例子二:

网络服务器处理,每个请求开一个线程,请求的处理时间极短,迅速返回,一次提交10万个请求,则有10万次线程创建和销毁

对应于一个工作线程处理这10万条请求

后者比前者肯定快

例子三：创建线程的开销大于线程的工作开销。当线程超过一定数量的时候，线程的调度将会变成很大的开销，反而会让性能降低，所以要适当使用多线程，不能滥用

CPU密集型 vs IO密集型

CPU密集型（CPU-bound）

CPU密集型也叫计算密集型，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，CPU Loading很高。

在多重程序系统中，大部份时间用来做计算、逻辑判断等CPU动作的程序称之CPU bound。例如一个计算圆周率至小数点一千位以下的程序，在执行的过程当中绝大部份时间用在三角函数和开根号的计算，便是属于CPU bound的程序。

CPU bound的程序一般而言CPU占用率相当高。这可能是因为任务本身不太需要访问I/O设备，也可能是因为程序是多线程实现因此屏蔽掉了等待I/O的时间。

IO密集型（I/O bound）

IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高。

I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力。

我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

总之，计算密集型程序适合C语言多线程，I/O密集型适合脚本语言开发的多线程。