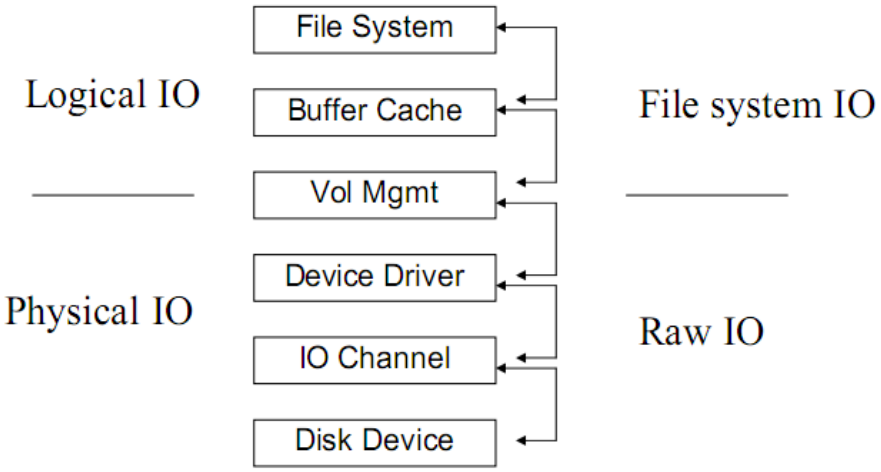
感谢博主 “和你在一起”http://pengjiaheng.iteye.com/ 给我们提供了这么一个优秀的 IO 专题。

整理：“yaocoder” http://yaocoder.blog.51cto.com/

说说 IO（一）- IO 的分层

IO 性能对于一个系统的影响是至关重要的。一个系统经过多项优化以后，瓶颈往往落在数据库；而数据库经过多种优化以后，瓶颈最终会落到 IO。而 IO 性能的发展，明显落后于 CPU 的发展。Memcached 也好，NoSql 也好，这些流行技术的背后都在直接或者间接地回避 IO 瓶颈，从而提高系统性能。

IO 系统的分层：



1. 三层结构

上图层次比较多，但总的就是三部分。**磁盘**（存储）、**VM**（卷管理）和**文件系统**。专有名词不好理解，打个比方说：磁盘就相当于一块待用的空地；LVM 相当于空地上的围墙（把空地划分成多个部分）；文件系统则相当于每块空地上建的楼房（决定了有多少房间、房屋编号如何，能容纳多少人住）；而房子里面住的人，则相当于系统里面存的数据。

- **文件系统—数据如何存放？**

  对应了上图的 File System 和 Buffer Cache。

  File System（文件系统）：解决了空间管理的问题，即：数据如何存放、读取。

Buffer Cache：解决数据缓冲的问题。对读，进行 cache，即：缓存经常要用到的数据；对写，进行 buffer，缓冲一定数据以后，一次性进行写入。

- **VM—磁盘空间不足了怎么办？**

  对应上图的 Vol Mgmt。

  VM 其实跟 IO 没有必然联系。他是处于文件系统和磁盘（存储）中间的一层。**VM 屏蔽了底层磁盘对上层文件系统的影响**。当没有 VM 的时候，文件系统直接使用存储上的地址空间，因此文件系统直接受限于物理硬盘，这时如果发生磁盘空间不足的情况，对应用而言将是一场噩梦，不得不新增硬盘，然后重新进行数据复制。而 VM 则可以实现动态扩展，而对文件系统没有影响。另外，VM 也可以把多个磁盘合并成一个磁盘，对文件系统呈现统一的地址空间，这个特性的杀伤力不言而喻。

- **存储—数据放在哪儿？如何访问？如何提高 IO 速度？**

  对应上图的 Device Driver、IO Channel 和 Disk Device

  数据最终会放在这里，因此，效率、数据安全、容灾是这里需要考虑的问题。而提高存储的性能，则可以直接提高物理 IO 的性能

2. Logical IO vs Physical IO

逻辑 IO 是操作系统发起的 IO，这个数据可能会放在磁盘上，也可能会放在内存（文件系统的 Cache）里。

物理 IO 是设备驱动发起的 IO，这个数据最终会落在磁盘上。

逻辑 IO 和物理 IO 不是一一对应的。

说说 IO（二）- IO 模型

这部分的东西在网络编程经常能看到，不过在所有 IO 处理中都是类似的。

IO 请求的两个阶段：

等待资源阶段：IO 请求一般需要请求特殊的资源（如磁盘、RAM、文件），当资源被上一个使用者使用没有被释放时，IO 请求就会被阻塞，直到能够使用这个资源。

使用资源阶段：真正进行数据接收和发生。

举例说就是**排队**和**服务**。

在**等待数据**阶段，IO 分为阻塞 IO 和非阻塞 IO。

阻塞 IO：资源不可用时，IO 请求一直阻塞，直到反馈结果（有数据或超时）。

非阻塞 IO：资源不可用时，IO 请求离开返回，返回数据标识资源不可用

在**使用资源**阶段，IO 分为同步 IO 和异步 IO。

同步 IO：应用阻塞在发送或接收数据的状态，直到数据成功传输或返回失败。

异步 IO：应用发送或接收数据后立刻返回，数据写入 OS 缓存，由 OS 完成数据发送或接收，并返回成功或失败的信息给应用。

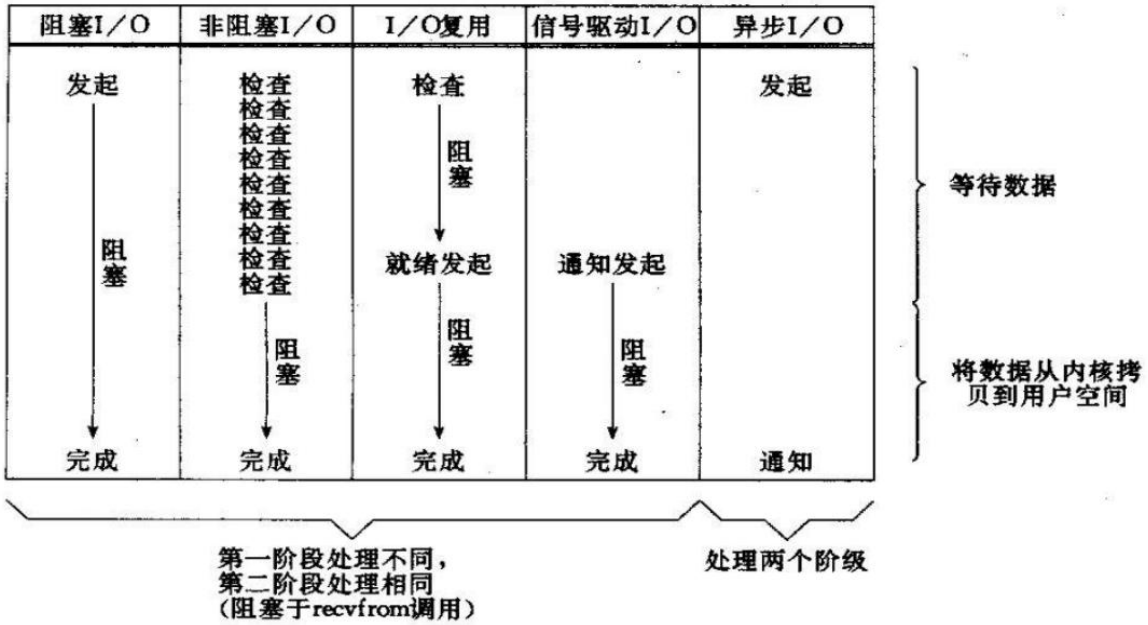


图 6.6　五个 I/O 模型的比较

按照 Unix 的 5 个 IO 模型划分

- 阻塞 IO
- 非阻塞 IO
- IO 复用
- 信号驱动的 IO
- 异步 IO

从性能上看，异步 IO 的性能无疑是最好的。

各种 IO 的特点

- **阻塞 IO**：使用简单，但随之而来的问题就是会形成阻塞，需要独立线程配合，而这些线程在大多数时候都是没有进行运算的。Java 的 BIO 使用这种方式，问题带来的问题很明显，一个 Socket 需要一个独立的线程，因此，会造成线程膨胀。

- **非阻塞 IO**：采用轮询方式，不会形成线程的阻塞。Java 的 NIO 使用这种方式，对比 BIO 的优势很明显，可以使用一个线程进行所有 Socket 的监听（select）。大大减少了线程数。

- **同步 IO**：同步 IO 保证一个 IO 操作结束之后才会返回，因此同步 IO 效率会低一些，但是对应用来说，编程方式会简单。Java 的 BIO 和 NIO 都是使用这种方式进行数据处理。
- **异步 IO**：由于异步 IO 请求只是写入了缓存，从缓存到硬盘是否成功不可知，因此异步 IO 相当于把一个 IO 拆成了两部分，一是发起请求，二是获取处理结果。因此，对应用来说增加了复杂性。但是异步 IO 的性能是所有很好的，而且异步的思想贯穿了 IT 系统放放面面。

说说 IO（三）- IO 性能的重要指标

最重要的三个指标

IOPS

IOPS，即每秒钟处理的 IO 请求数量。IOPS 是随机访问类型业务（OLTP 类）很重要的一个参考指标。

- **一块物理硬盘能提供多少 IOPS？**

从磁盘上进行数据读取时，比较重要的几个时间是：**寻址时间**（找到数据块的起始位置），**旋转时间**（等待磁盘旋转到数据块的起始位置），**传输时间**（读取数据的时间和返回的时间）。其中寻址时间是固定的（磁头定位到数据的存储的扇区即可），旋转时间受磁盘转速的影响，传输时间受数据量大小的影响和接口类型的影响（不用硬盘接口速度不同），但是在随机访问类业务中，他的时间也很少。因此，在硬盘接口相同的情况下，IOPS 主要受限于寻址时间和传输时间。以一个 15K 的硬盘为例，寻址时间固定为 4ms，传输时间为 60s/15000*1/2=2ms，忽略传输时间。1000ms/6ms=167 个 IOPS。

- **OS 的一次 IO 请求对应物理硬盘一个 IO 吗？**

在没有文件系统、没有 VM（卷管理）、没有 RAID、没有存储设备的情况下，这个答案还是成立的。但是当这么多中间层加进去以后，这个答案就不是这样了。物理硬盘提供的 IO 是有限的，也是整个 IO 系统存在瓶颈的最大根源。所以，如果一块硬盘不能提供，那么多块在一起并行处理，这不就行了吗？确实是这样的。可以看到，**越是高端的存储设备的 cache 越大，硬盘越多，一方面通过 cache 异步处理 IO，另一方面通过盘数增加，尽可能把一个 OS 的 IO 分布到不同硬盘上，从而提高性能**。文件系统则是在 cache 上会影响，而 VM 则可能是一个 IO 分布到多个不同设备上（Striping）。

所以，**一个 OS 的 IO 在经过多个中间层以后，发生在物理磁盘上的 IO 是不确定的。可能是一对一个，也可能一个对应多个**。

- **IOPS 能算出来吗？**

对单块磁盘的 IOPS 的计算没有没问题，但是当系统后面接的是一个存储系统时、考虑不同读写比例，IOPS 则很难计算，而需要根据实际情况进行测试。主要的因素有：

- **存储系统本身有自己的缓存**。缓存大小直接影响 IOPS，理论上说，缓存越大能 cache 的东西越多，在 cache 命中率保持的情况下，IOPS 会越高。
- **RAID 级别**。不同的 RAID 级别影响了物理 IO 的效率。
- **读写混合比例**。对读操作，一般只要 cache 能足够大，可以大大减少物理 IO，而都在 cache 中进行；对写操作，不论 cache 有多大，最终的写还是会落到磁盘上。因此，100%写的 IOPS 要小于 100%的读的 IOPS。同时，100%写的 IOPS 大致等同于存储设备能提供的物理的 IOPS。
- **一次 IO 请求数据量的多少**。一次读写 1KB 和一次读写 1MB，显而易见，结果是完全不同的。

当时上面 N 多因素混合在一起以后，IOPS 的值就变得扑朔迷离了。所以，一般需要通过实际应用的测试才能获得。

IO Response Time

即 IO 的响应时间。IO 响应时间是从操作系统内核发出一个 IO 请求到接收到 IO 响应的时间。因此，IO Response time 除了包括磁盘获取数据的时间，还包括了操作系统以及在存储系统内部 IO 等待的时间。一般看，随 IOPS 增加，因为 IO 出现等待，IO 响应时间也会随之增加。对一个 OLTP 系统，10ms 以内的响应时间，是比较合理的。下面是一些 IO 性能示例：

- **一个 8K 的 IO 会比一个 64K 的 IO 速度快**，因为数据读取的少些。
- **一个 64K 的 IO 会比 8 个 8K 的 IO 速度快**，因为前者只请求了一个 IO 而后者是 8 个 IO。
- **串行 IO 会比随机 IO 快**，因为串行 IO 相对随机 IO 说，即便没有 Cache，串行 IO 在磁盘处理上也会少些操作。

需要注意，IOPS 与 IO Response Time 有着密切的联系。一般情况下，IOPS 增加，说明 IO 请求多了，IO Response Time 会相应增加。但是会出现 IOPS 一直增加，但是 IO Response Time 变得非常慢，超过 20ms 甚至几十 ms，这时候的 IOPS 虽然还在提高，但是意义已经不大，因为整个 IO 系统的服务时间已经不可取。

Throughput

为吞吐量。这个指标衡量标识了最大的数据传输量。如上说明，**这个值在顺序访问或者大数据量访问的情况下会比较重要**。尤其在大数据量写的时候。

吞吐量不像 IOPS 影响因素很多，吞吐量一般受限于一些比较固定的因素，如：网络带宽、IO 传输接口的带宽、硬盘接口带宽等。一般他的值就等于上面几个地方中某一个的瓶颈。

一些概念

IO Chunk Size

即单个 IO 操作请求数据的大小。一次 IO 操作是指从发出 IO 请求到返回数据的过程。IO Chunk Size 与应用或业务逻辑有着很密切的关系。比如像 Oracle 一类数据库，由于其 block size 一般为 8K，读取、写入时都此为单位，因此，8K 为这个系统主要的 IO Chunk Size。**IO Chunk Size**

小，考验的是 IO 系统的 IOPS 能力；**IO Chunk Size** 大，考验的时候 IO 系统的 IO 吞吐量。

Queue Deep

熟悉数据库的人都知道，SQL 是可以批量提交的，这样可以大大提高操作效率。IO 请求也是一样，IO 请求可以积累一定数据，然后一次提交到存储系统，这样一些相邻的数据块操作可以进行合并，减少物理 IO 数。而且 Queue Deep 如其名，就是设置一起提交的 IO 请求数量的。一般 Queue Deep 在 IO 驱动层面上进行配置。

Queue Deep 与 IOPS 有着密切关系。Queue Deep 主要考虑批量提交 IO 请求，自然只有 IOPS 是瓶颈的时候才会有意义，如果 IO 都是大 IO，磁盘已经成瓶颈，Queue Deep 意义也就不大了。一般来说，IOPS 的峰值会随着 Queue Deep 的增加而增加(不会非常显著)，Queue Deep 一般小于 256。

随机访问（随机 IO）、顺序访问（顺序 IO）

随机访问的特点是**每次 IO 请求的数据在磁盘上的位置跨度很大**（如：分布在不同的扇区），因此 N 个非常小的 IO 请求（如：1K），必须以 N 次 IO 请求才能获取到相应的数据。

顺序访问的特点跟随机访问相反，**它请求的数据在磁盘的位置是连续的**。当系统发起 N 个非常小的 IO 请求（如：1K）时，因为一次 IO 是有代价的，系统会取完整的一块数据（如 4K、8K），所以当第一次 IO 完成时，后续 IO 请求的数据可能已经有了。这样可以减少 IO 请求的次数。这也就是所谓的预取。

随机访问和顺序访问同样是有应用决定的。如数据库、小文件的存储的业务，大多是随机 IO。而视频类业务、大文件存取，则大多为顺序 IO。

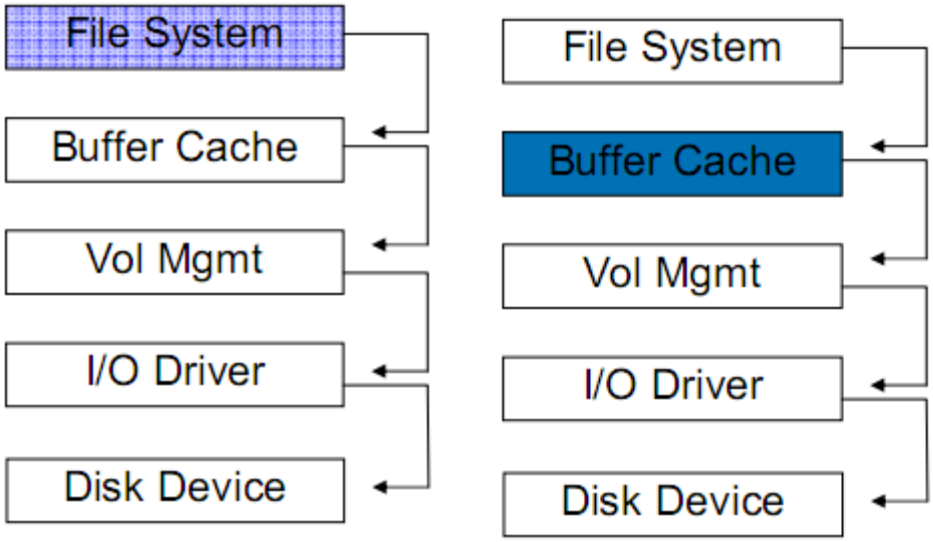
选取合理的观察指标：

以上各指标中，不同的应用场景需要观察不同的指标，因为应用场景不同，有些指标甚至是没有意义的。

随机访问和 IOPS: 在随机访问场景下，IOPS 往往会到达瓶颈，而这个时候去观察 Throughput，则往往远低于理论值。

顺序访问和 Throughput：在顺序访问的场景下，Throughput 往往会达到瓶颈（磁盘限制或者带宽），而这时候去观察 IOPS，往往很小。

说说 IO（四）- 文件系统

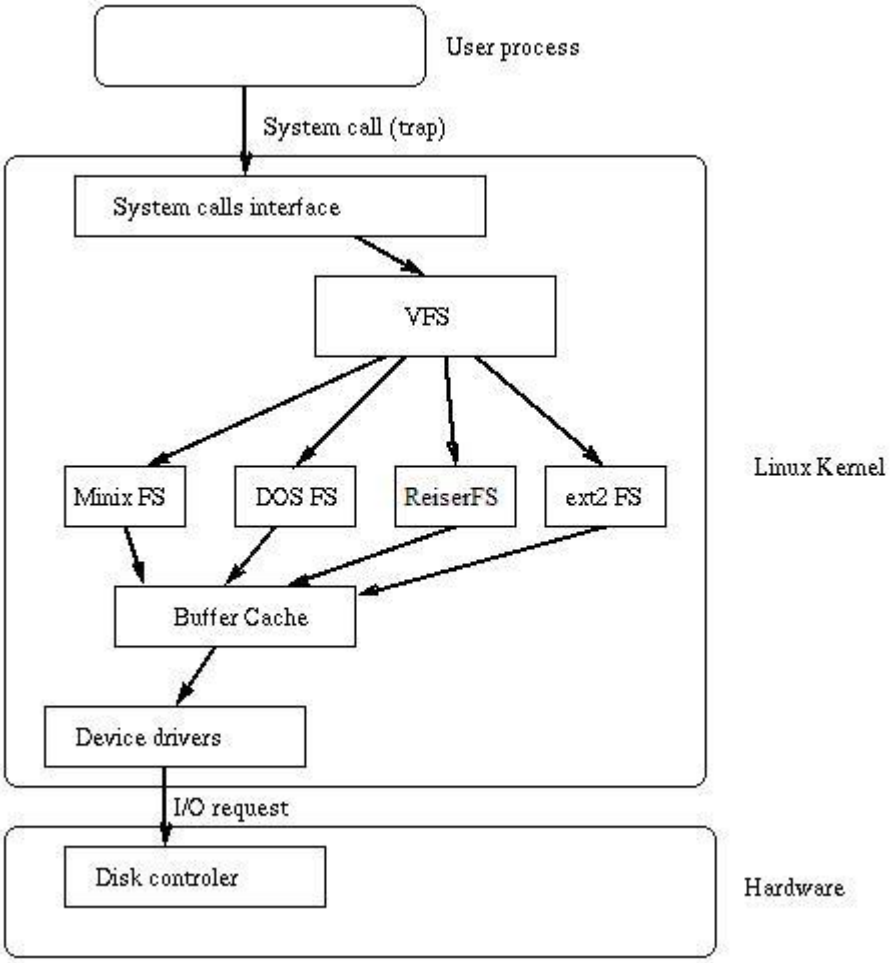


文件系统各有不同，其最主要的目标就是解决磁盘空间的管理问题，同时提供高效性、安全性。如果在分布式环境下，则有相应的分布式文件系统。Linux 上有 ext 系列，Windows 上有 Fat 和 NTFS。如图为一个 linux 下文件系统的结构。

其中 VFS（Virtual File System）是 Linux Kernel 文件系统的一个模块，简单看就是一个 Adapter，对下屏蔽了下层不同文件系统之间的差异，对上为操作系统提供了统一的接口.

中间部分为各个不同文件系统的实现。

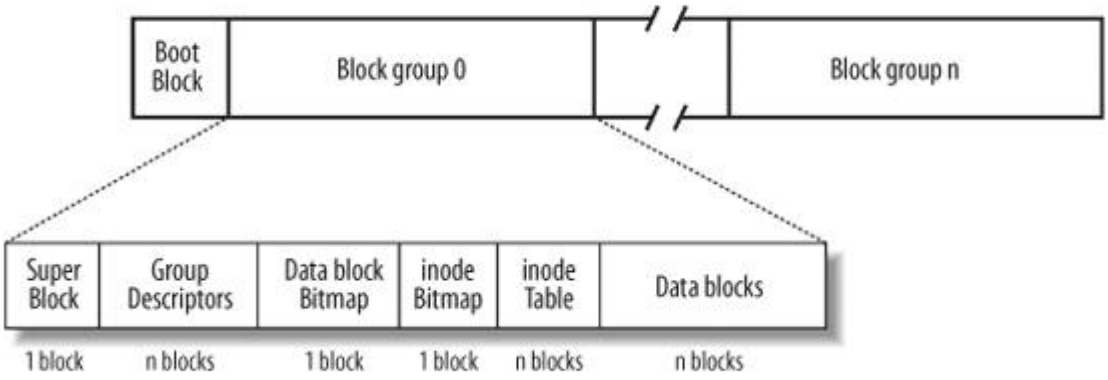
再往下是 Buffer Cache 和 Driver。



文件系统的结构

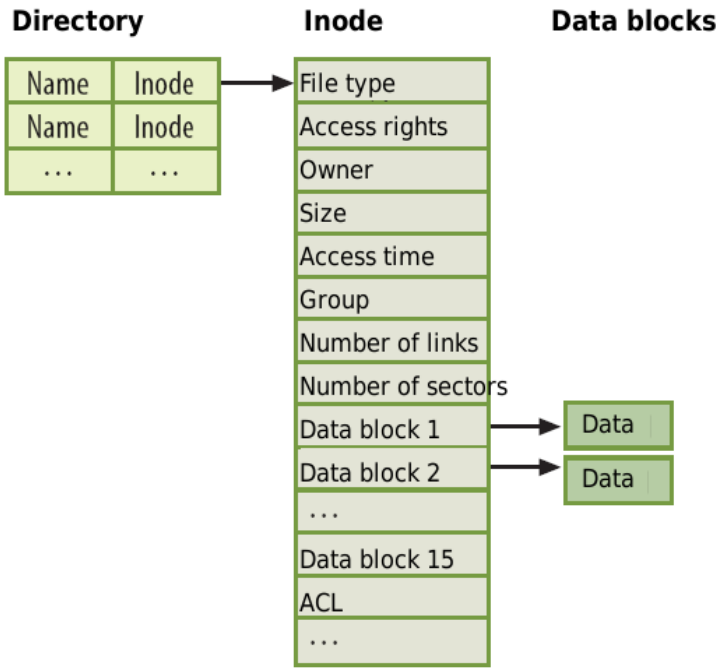
各种文件系统实现方式不同，因此性能、管理性、可靠性等也有所不同。下面为 Linux Ext2（Ext3）的一个大致文件系统的结构。

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



Boot Block 存放了引导程序。

Super Block 存放了整个文件系统的一些全局参数，如：卷名、状态、块大小、块总数。他在文件系统被 mount 时读入内存，在 umount 时被释放。

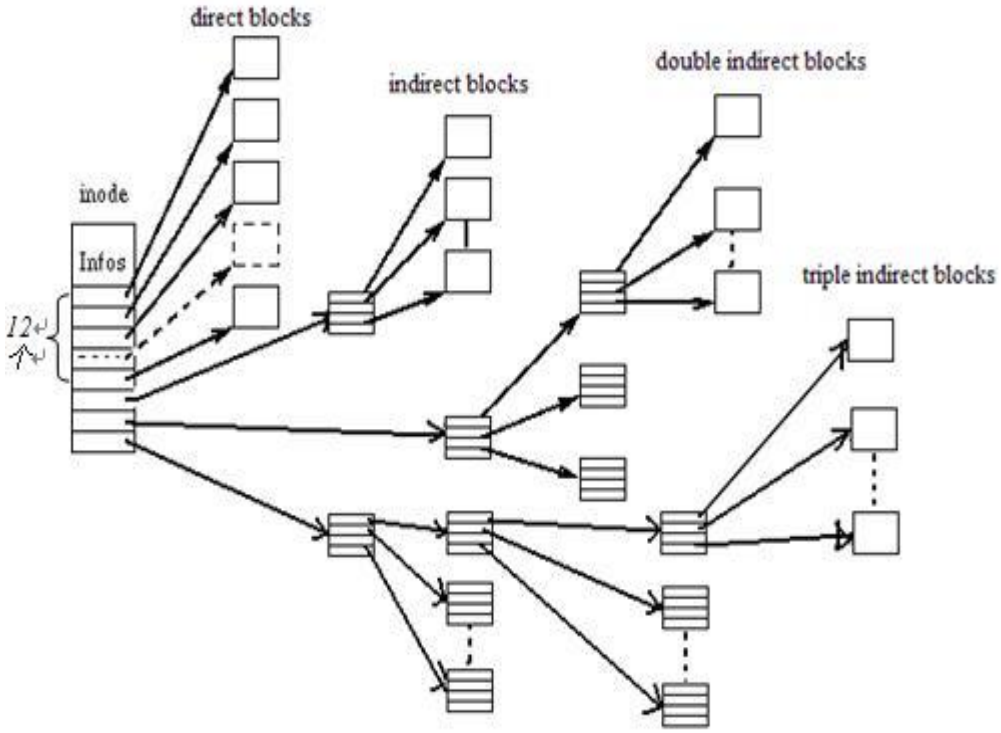


上图描述了 Ext2 文件系统中很重要的三个数据结构和他们之间的关系。

Inode：Inode 是文件系统中最重要的一个结构。如图，他里面记录了文件相关的所有信息，也就是我们常说的 meta 信息。包括：文件类型、权限、所有者、大小、atime 等。Inode 里面也保存了指向实际文件内容信息的索引。其中这种索引分几类：

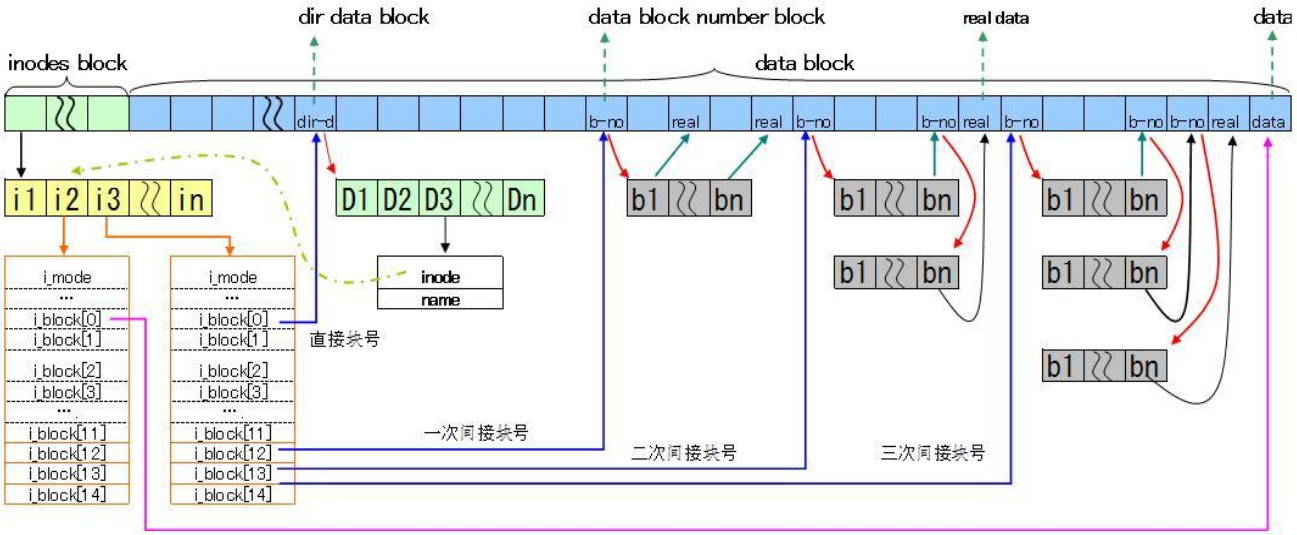
- 直接索引：直接指向实际内容信息，公有 12 个。因此如果，一个文件系统 block size 为 1k，那么直接索引到的内容最大为 12k
- 间接索引
- 两级间接索引
- 三级间接索引

如图：



图〖三〗：*inode*结构示意图

Directory 代表了文件系统中的目录，包括了当前目录中的所有 Inode 信息。其中每行只有两个信息，一个是文件名，一个是其对应的 Inode。需要注意，Directory 不是文件系统中的一个特殊结构，他实际上也是一个文件，有自己的 Inode，而它的文件内容信息里面，包括了上面看到的那些文件名和 Inode 的对应关系。如下图：



Data Block 即存放文件的时间内容块。Data Block 大小必须为磁盘的数据块大小的整数倍，磁盘一般为 512 字节，因此 Data Block 一般为 1K、2K、4K。

Buffer Cache

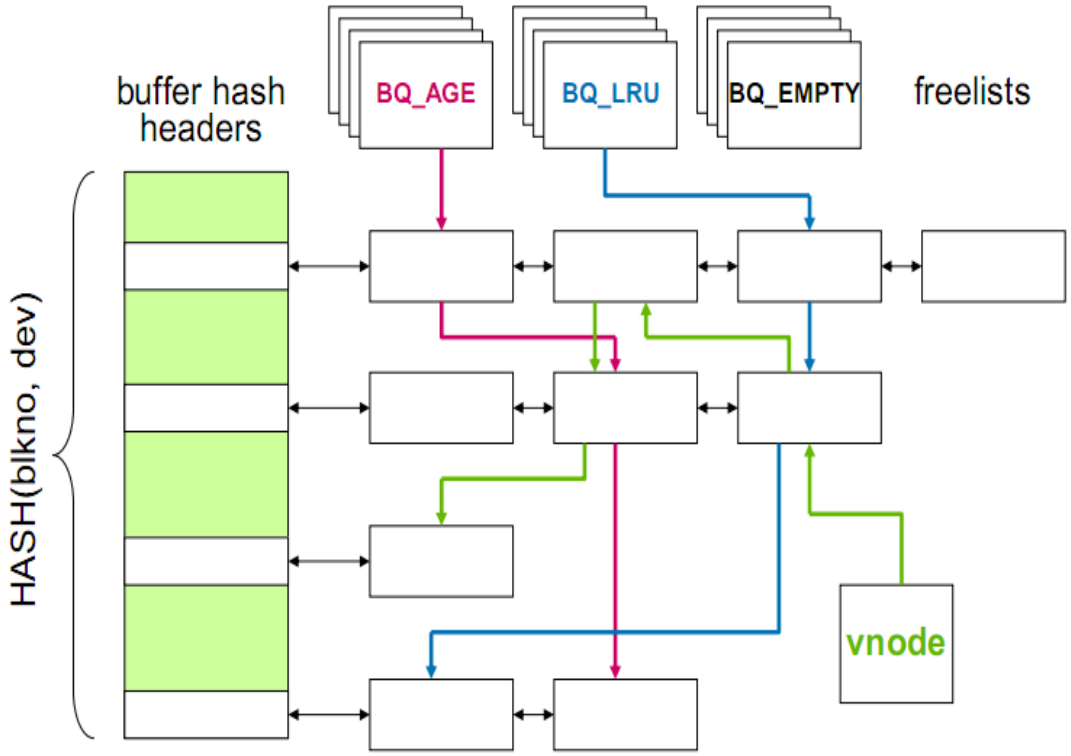
Buffer & Cache

虽然 Buffer 和 Cache 放在一起了，但是在实际过程中 Buffer 和 Cache 是完全不同了。Buffer 一般对于写而言，也叫“缓冲区”，缓冲使得多个小的数据块能够合并成一个大数据块，一次性写入；Cache 一般对于读而且，也叫“缓存”，避免频繁的磁盘读取。如图为 Linux 的 free 命令，其中也是把 Buffer 和 Cache 进行区分，这两部分都算在了 free 的内存。

```
$ free
             total       used       free     shared    buffers     cached
Mem:      49452900   21214516   28238384          0     356164   10787376
-/+ buffers/cache:   10070976   39381924
Swap:     51511288          0   51511288
```

Buffer Cache

Buffer Cache 中的缓存，本质与所有的缓存都是一样，数据结构也是类似，下图为 VxSF 的一个 Buffer Cache 结构。



这个数据结构与 memcached 和 Oracle SGA 的 buffer 何等相似。左侧的 hash chain 完成数据块的寻址，上方的的链表记录了数据块的状态。

Buffer vs Direct I/O

文件系统的 Buffer 和 Cache 在某些情况下确实提高了速度，但是反之也会带来一些负面影响。一方面文件系统增加了一个中间层，另外一方面，当 Cache 使用不当、配置不好或者有些业务无法获取 cache 带来的好处时，cache 则成为了一种负担。

- 适合 Cache 的业务：串行的大数据量业务，如：NFS、FTP。
- 不适合 Cache 的业务：随机 IO 的业务。如：Oracle，小文件读取。

块设备、字符设备、裸设备

这几个东西看得很晕，找了一些资料也没有找到很准确的说明。

从硬件设备的角度来看，

- 块设备就是以块（比如磁盘扇区）为单位收发数据的设备，它们支持缓冲和随机访问（不必顺序读取块，而是可以在任何时候访问任何块）等特性。块设备包括硬盘、CD-ROM 和 RAM 盘。
- 字符设备则没有可以进行物理寻址的媒体。字符设备包括串行端口和磁带设备，只能逐字符地读取这些设备中的数据。

从操作系统的角度看（对应操作系统的设备文件类型的 b 和 c），

```
# ls -l /dev/*lv
brw-------    1 root     system           22,     2 May 15 2007  lv
crw-------    2 root     system           22,     2 May 15 2007  rlv
```

- **块设备能支持缓冲和随机读写**。即读取和写入时，可以是任意长度的数据。最小为 1byte。对块设备，你可以成功执行下列命令：dd if=/dev/zero of=/dev/vg01/lv bs=1 count=1。即：在设备中写入一个字节。硬件设备是不支持这样的操作的（最小是 512），这个时候，操作系统首先完成一个读取（如 1K，操作系统最小的读写单位，为硬件设备支持的数据块的整数倍），再更改这 1k 上的数据，然后写入设备。
- **字符设备只能支持固定长度数据的读取和写入**，这里的长度就是操作系统能支持的最小读写单位，如 1K，所以块设备的缓冲功能，这里就没有了，需要使用者自己来完成。由于读写时不经过任何缓冲区，此时执行 dd if=/dev/zero of=/dev/vg01/lv bs=1 count=1，这个命令将会出错，因为这里的 bs（block size）太小，系统无法支持。如果执行 dd if=/dev/zero of=/dev/vg01/lv bs=1024 count=1，则可以成功。这里的 block size 有 OS 内核参数决定。

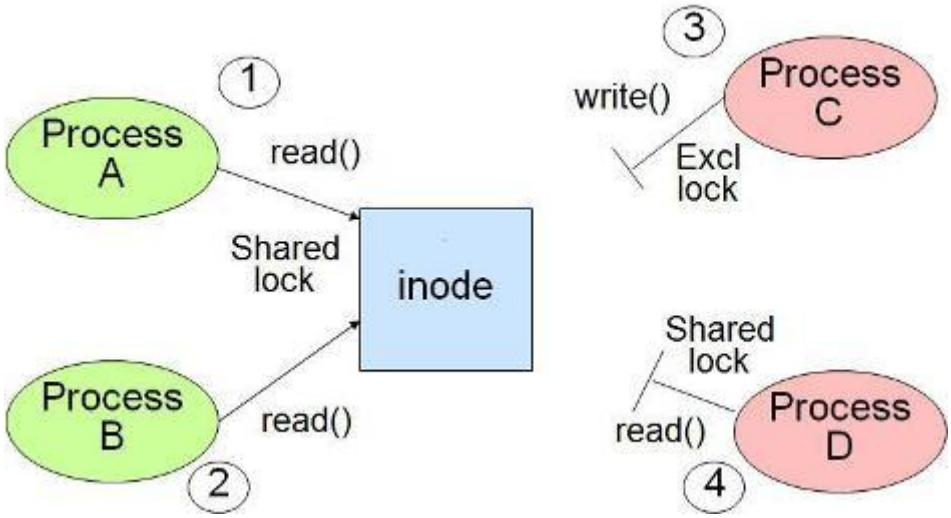
如上，相比之下，字符设备在使用更为直接，而块设备更为灵活。文件系统一般建立在块设备上，而为了追求高性能，使用字符设备则是更好的选择，如 Oracle 的裸设备使用。

裸设备

裸设备也叫裸分区，就是没有经过格式化、没有文件系统的一块存储空间。可以写入二进制内容，但是内容的格式、其中信息的组织等问题，需要使用它的人来完成。文件系统就是建立在裸设备之上，并完成裸设备空间的管理。

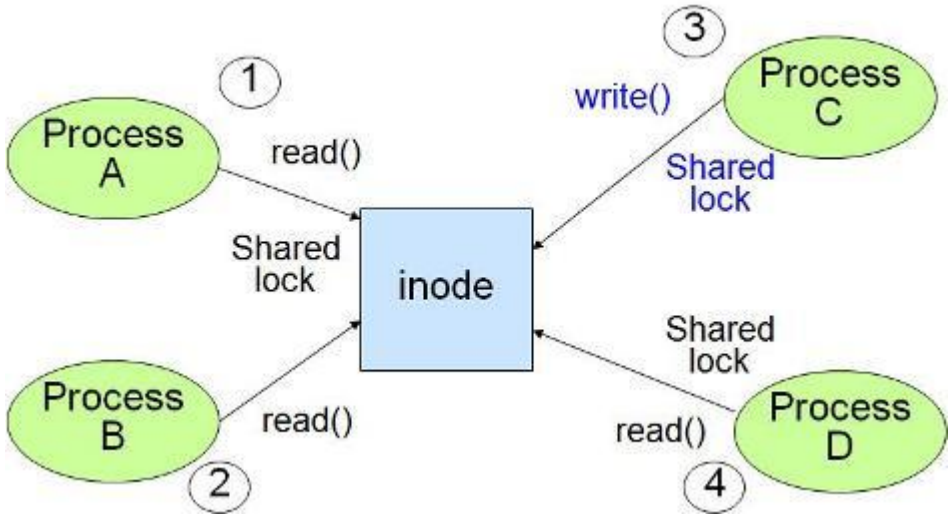
CIO

CIO 即并行 IO（Concurrent IO）。在文件系统中，当某个文件被多个进程同时访问时，就出现了 Inode 竞争的问题。一般地，读操作使用的共享锁，即：多个读操作可以并发进行，而写操作使用排他锁。当锁被写进程占用时，其他所有操作均阻塞。因此，当这样的情况出现时，整个应用的性能将会大大降低。如图：



CIO 就是为了解决这个问题。而且 CIO 带来的性能提高直逼裸设备。当文件系统支持 CIO 并开启 CIO 时，CIO 默认会开启文件系统的 Direct IO，即：让 IO 操作不经过 Buffer 直接进行底层数据操作。由于不经过数据 Buffer，在文件系统层面就无需考虑数据一致性的问题，因此，读写操作可以并行执行。

在最终进行数据存储的时候，所有操作都会串行执行，CIO 把这个事情交个了底层的 driver。



[说说 IO（五）- 逻辑卷管理]

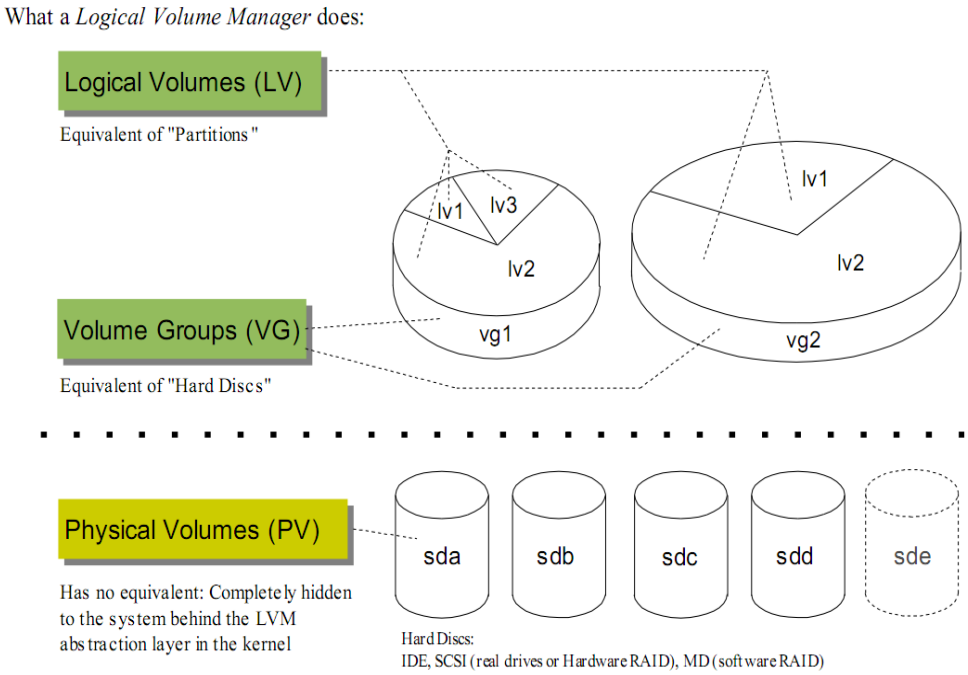
LVM（逻辑卷管理），位于操作系统和硬盘之间，LVM 屏蔽了底层硬盘带来的复杂性。最简单的，LVM 使得 N 块硬盘在 OS 看来成为一块硬盘，大大提高了系统可用性。

LVM 的引入，使得文件系统和底层磁盘之间的关系变得更为灵活，而且更方便关系。LVM 有以下特点：

- 统一进行磁盘管理。按需分配空间，提供动态扩展。
- 条带化（Striped）
- 镜像（mirrored）
- 快照（snapshot）

LVM 可以做动态磁盘扩展，想想看，当系统管理员发现应用空间不足时，敲两个命令就完成空间扩展，估计做梦都要笑醒：）

LVM 的磁盘管理方式



LVM 中有几个很重要的概念：

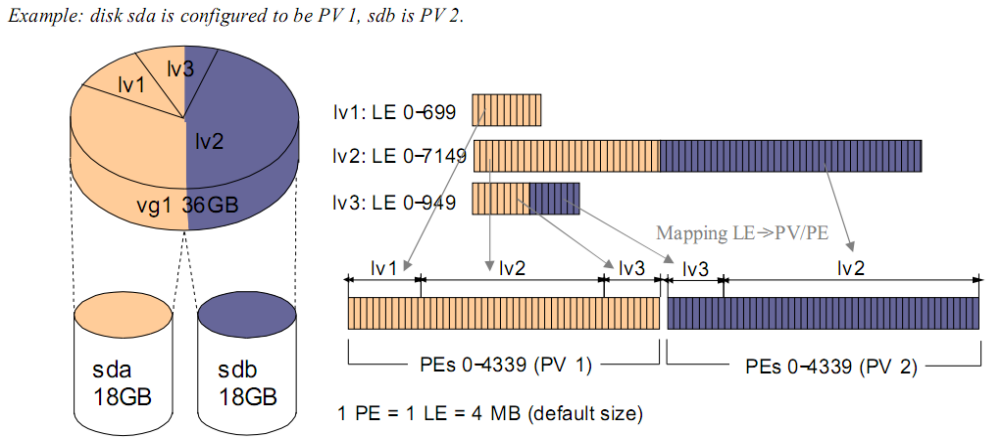
- **PV（physical volume）**：物理卷。在 LVM 中，一个 PV 对应就是操作系统能看见的一块物理磁盘，或者由存储设备分配操作系统的 lun。一块磁盘唯一对应一个 PV，PV 创建以后，说明这块空间可以纳入到 LVM 的管理。创建 PV 时，可以指定 PV 大小，即可以把整个磁盘的部分纳入 PV，而不是全部磁盘。这点在表面上看没有什么意义，但是如果主机后面接的是存储设备的话就很有意义了，因为存储设备分配的 lun 是可以动态扩展的，只有当 PV 可以动态扩展，这种扩展性才能向上延伸。
- **VG（volume group）**：卷组。一个 VG 是多个 PV 的集合，简单说就是一个 VG 就是一个磁盘资源池。VG 对上屏蔽了多个物理磁盘，上层是使用时只需考虑空间大小的问题，而 VG 解决的空间的如何在多个 PV 上连续的问题。
- **LV（logical volume）**：逻辑卷。LV 是最终可供使用卷，LV 在 VG 中创建，有了 VG，LV 创建是只需考虑空间大小等问题，对 LV 而言，他看到的是一直联系的地址空间，不用考虑多块硬盘的问题。

有了上面三个，LVM 把单个的磁盘抽象成了一组连续的、可随意分配的地址空间。除上面三个概念外，还有一些其他概念：

- **PE（physical extend）**： 物理扩展块。LVM 在创建 PV，不会按字节方式去进行空间管理。而是按 PE 为单位。PE 为空间管理的最小单位。即：如果一个 1024M 的物理盘，LVM 的 PE 为 4M，那么 LVM 管理空间时，会按照 256 个 PE 去管理。分配时，也是按照分配了多少 PE、剩余多少 PE 考虑。

- **LE（logical extend）**：逻辑扩展块。类似 PV，LE 是创建 LV 考虑，当 LV 需要动态扩展时，每次最小的扩展单位。

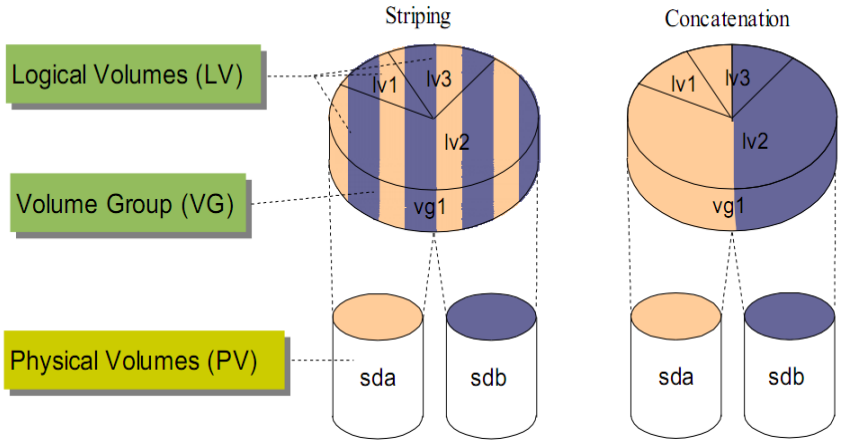
对于上面几个概念，无需刻意去记住，当你需要做这么一个东西时，这些概念是自然而然的。PV 把物理硬盘转换成 LVM 中对于的逻辑（解决如何管理物理硬盘的问题），VG 是 PV 的集合（解决如何组合 PV 的问题），LV 是 VG 上空间的再划分（解决如何给 OS 使用空间的问题）；而 PE、LE 则是空间分配时的单位。



如图，为两块 18G 的磁盘组成了一个 36G 的 VG。此 VG 上划分了 3 个 LV。其 PE 和 LE 都为 4M。其中 LV1 只用到了 sda 的空间，而 LV2 和 LV3 使用到了两块磁盘。

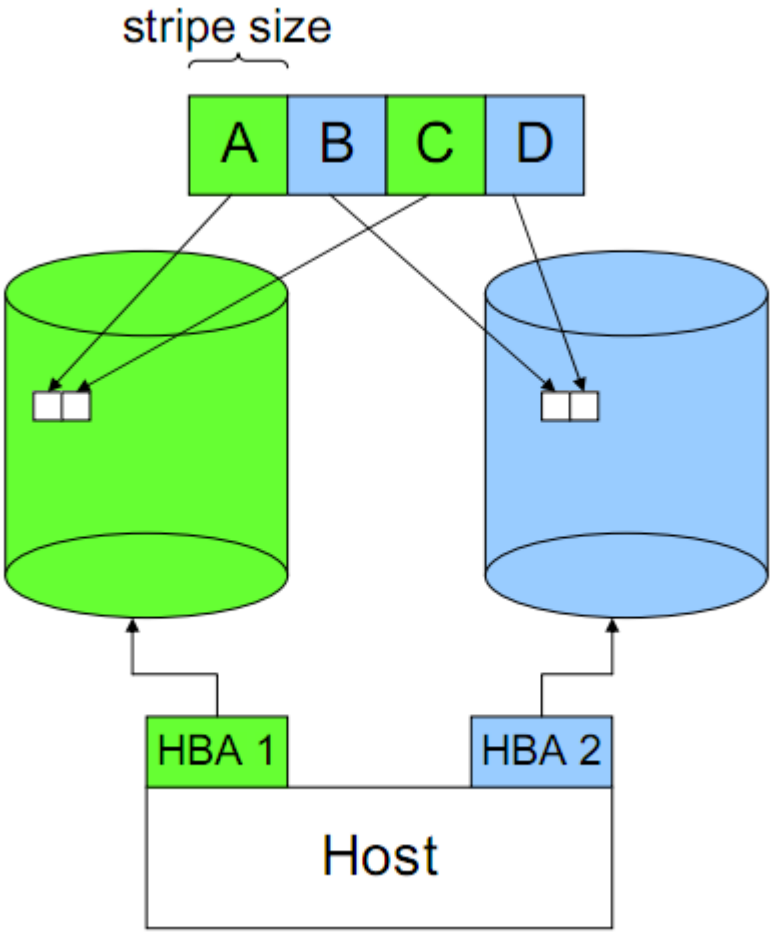
串联、条带化、镜像

Example for concatenation & striping of two disks to one logical disk:



串联（Concatenation）： 按顺序使用磁盘，一个磁盘使用完以后使用后续的磁盘。

条带化（Striping）： 交替使用不同磁盘的空间。条带化使得 IO 操作可以并行，因此是提高 IO 性能的关键。另外，Striping 也是 RAID 的基础。如：VG 有 2 个 PV，LV 做了条带数量为 2 的条带化，条带大小为 8K，那么当 OS 发起一个 16K 的写操作时，那么刚好这 2 个 PV 对应的磁盘可以对整个写入操作进行并行写入。



Striping 带来好处有：

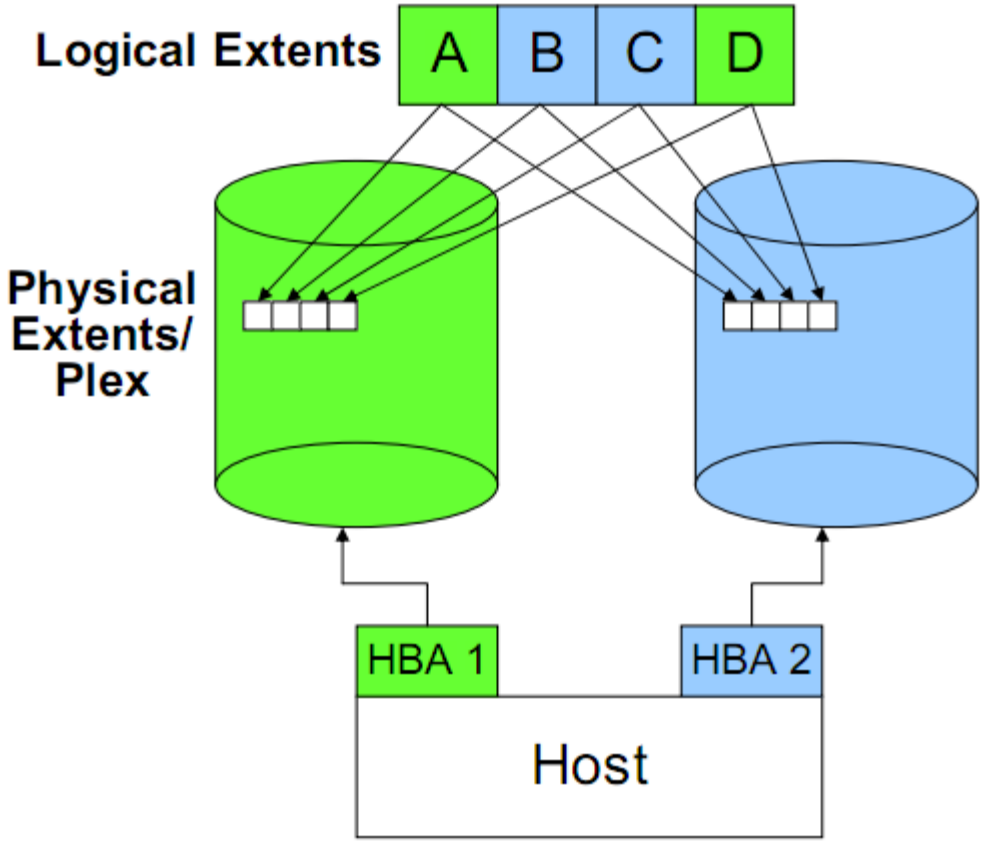
- 并发进行数据处理。读写操作可以同时发送在多个磁盘上，大大提高了性能。

Striping 带来的问题：

- 数据完整性的风险。Striping 导致一份完整的数据被分布到多个磁盘上，任何一个磁盘上的数据都是不完整，也无法进行还原。一个条带的损坏会导致所有数据的失效。因此这个问题只能通过存储设备来弥补。
- 条带大小的设定很大程度决定了 Striping 带来的好处。如果条带设置过大，一个 IO 操作最终还是发生在一个磁盘上，无法带来并行的好处；当条带设置国小，本来一次并行 IO 可以完成的事情会最终导致了多次并行 IO。

镜像（mirror）

如同名字。LVM 提供 LV 镜像的功能。即当一个 LV 进行 IO 操作时，相同的操作发生在另外一个 LV 上。这样的功能为数据的安全性提供了支持。如图，一份数据被同时写入两个不同的 PV。



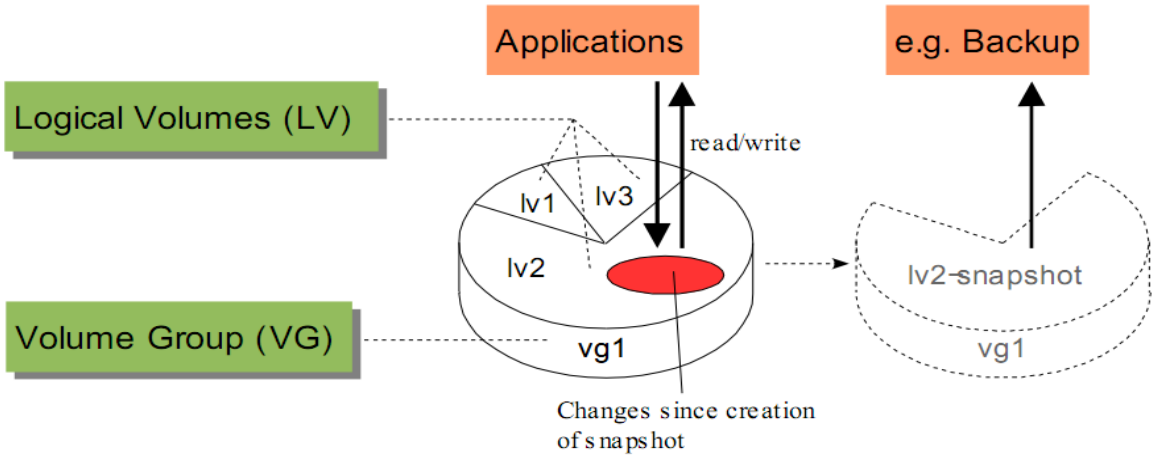
使用 mirror 时，可以获得一些好处：

- 读取操作可以从两个磁盘上获取，因此读效率会更好些。
- 数据完整复杂了一份，安全性更高。

但是，伴随也存在一些问题：

- 所有的写操作都会同时发送在两个磁盘上，因此实际发送的 IO 是请求 IO 的 2 倍
- 由于写操作在两个磁盘上发生，因此一些完整的写操作需要两边都完成了才算完成，带来了额外负担。
- 在处理串行 IO 时，有些 IO 走一个磁盘，另外一些 IO 走另外的磁盘，一个完整的 IO 请求会被打乱，LVM 需要进行 IO 数据的合并，才能提供给上层。像一些如预读的功能，由于有了多个数据获取同道，也会存在额外的负担。

快照（Snapshot）



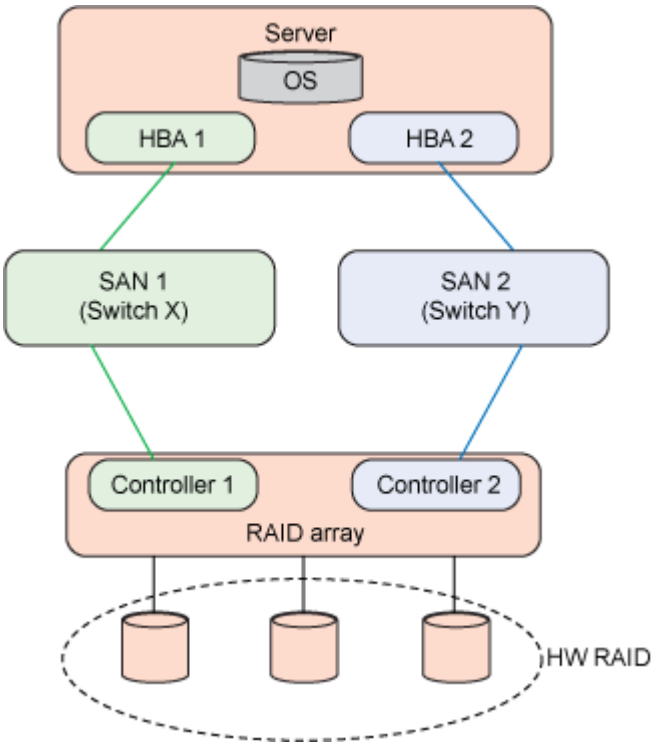
快照如其名，他保存了某一时间点磁盘的状态，而后续数据的变化不会影响快照，因此，快照是一种备份很好手段。

但是快照由于保存了某一时间点数据的状态，因此在数据变化时，这部分数据需要写到其他地方，随着而来回带来一些问题。关于这块，后续存储也涉及到类似的问题，后面再说。

说说 IO（六）- Driver & IO Channel

这部分值得一说的是多路径问题。IO 部分的高可用性在整个应用系统中可以说是最关键的，应用层可以坏掉一两台机器没有问题，但是如果 IO 不通了，整个系统都没法使用。如图为一个典型的 SAN 网络，从主机到磁盘，所有路径上都提供了冗余，以备发生通路中断的情况。

- OS 配置了 2 块光纤卡，分别连不同交换机
- SAN 网络配置了 2 个交换机
- 存储配置了 2 个 Controller，分别连不同交换机



如上图结构，由于存在两条路径，对于存储划分的一个空间，在 OS 端会看到两个（两块磁盘或者两个 lun）。可怕的是，OS 并不知道这两个东西对应的其实是一块空间，如果路径再多，则 OS 会看到更多。还是那句经典的话，“计算机中碰到的问题，往往可以通过增加的一个中间层来解决”，于是有了多路径软件。他提供了以下特性：

- 把多个映射到同一块空间的路径合并为一个提供给主机
- 提供 fail over 的支持。当一条通路出现问题时，及时切换到其他通路
- 提供 load balance 的支持。即同时使用多条路径进行数据传送，发挥多路径的资源优势，提高系统整体带宽。

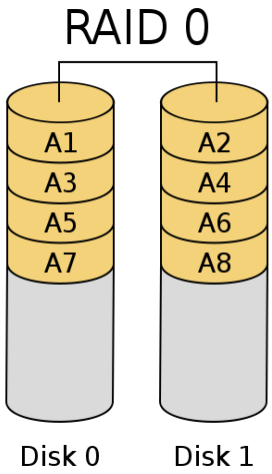
Fail over 的能力一般 OS 也可能支持，而 load balance 则需要与存储配合，所以需要根据存储不同配置安装不同的多通路软件。
多路径除了解决了高可用性，同时，多条路径也可以同时工作，提高系统性能。

说说 IO（七）- RAID

Raid 很基础，但是在存储系统中占据非常重要的地位，所有涉及存储的书籍都会提到 RAID。RAID 通过磁盘冗余的方式提高了可用性和可高性，一方面增加了数据读写速度，另一方面增加了数据的安全性。

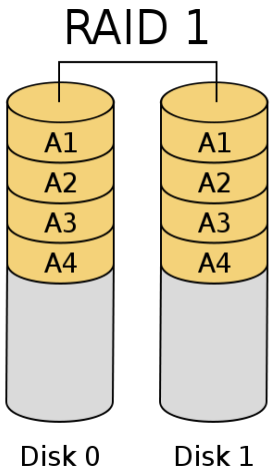
RAID 0

对数据进行条带化。使用两个磁盘交替存放连续数据。因此可以实现并发读写，但带来的问题是如果一个磁盘损坏，另外一个磁盘的数据将失去意义。RAID 0 最少需要 2 块盘。



RAID 1

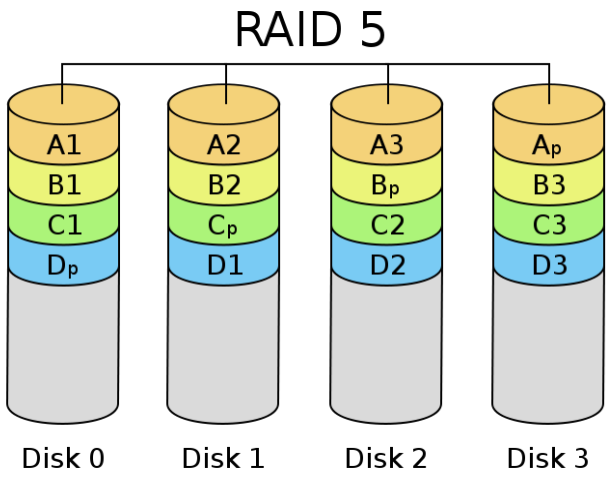
对数据进行镜像。数据写入时，相同的数据同时写入两块盘。因此两个盘的数据完全一致，如果一块盘损坏，另外一块盘可以顶替使用，RAID 1 带来了很好的可靠性。同时读的时候，数据可以从两个盘上进行读取。但是 RAID 1 带来的问题就是空间的浪费。两块盘只提供了一块盘的空间。RAID 1 最少需要 2 块盘。



RAID 5 和 RAID 4

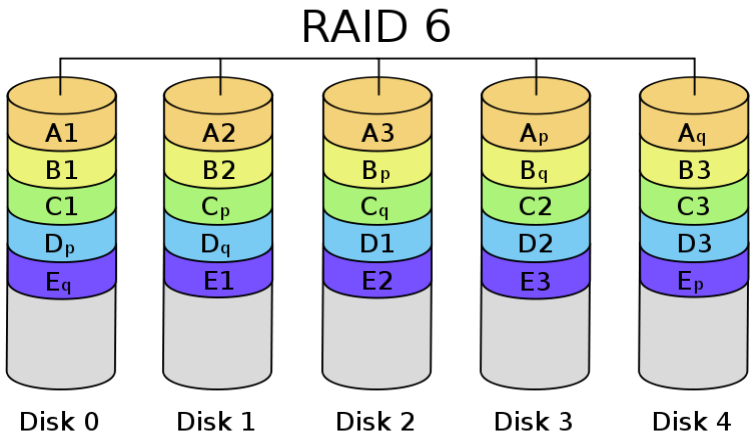
使用多余的一块校验盘。数据写入时，RAID 5 需要对数据进行计算，以便得出校验位。因此，在写性能上 RAID 5 会有损失。但是 RAID 5 兼顾了性能和安全性。当有一块磁盘损坏时，RAID 5 可以通过其他盘上的数据对其进行恢复。

如图可以看出，右下角为 p 的就是校验数据。可以看到 RAID 5 的校验数据依次分布在不同的盘上，这样可以避免出现热点盘(因为所有写操作和更新操作都需要修改校验信息，如果校验都在一个盘做，会导致这个盘成为写瓶颈，从而拖累整体性能，RAID 4 的问题）。RAID 5 最少需要 3 块盘。



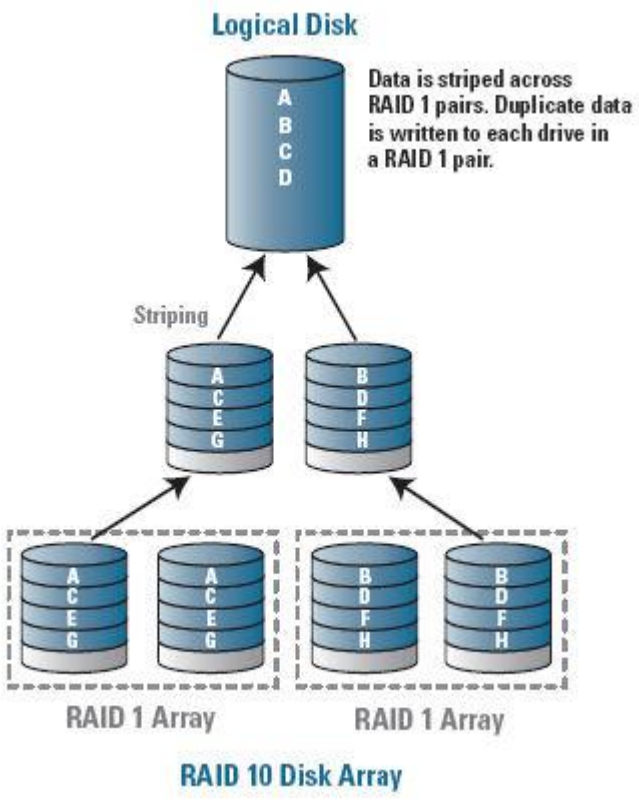
RAID 6

RAID 6 与 RAID 5 类似。但是提供了两块校验盘（下图右下角为 p 和 q 的）。安全性更高，写性能更差了。RAID 6 最少需要 4 块盘。



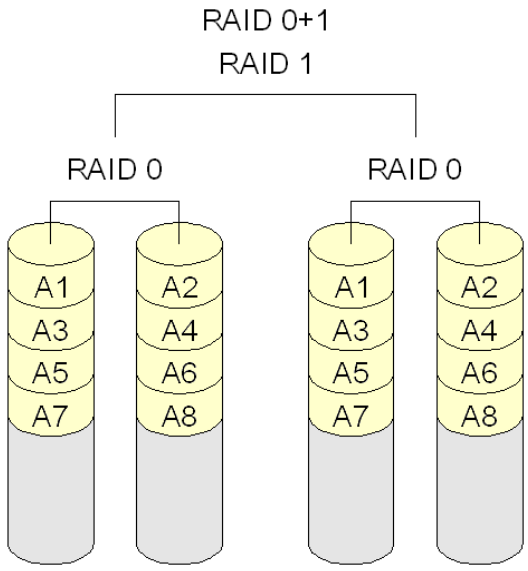
RAID 10（Striped mirror）

RAID 10 是 RAID 0 和 RAID 1 的结合，同时兼顾了二者的特点，提供了高性能，但是同时空间使用也是最大。RAID 10 最少需要 4 块盘。



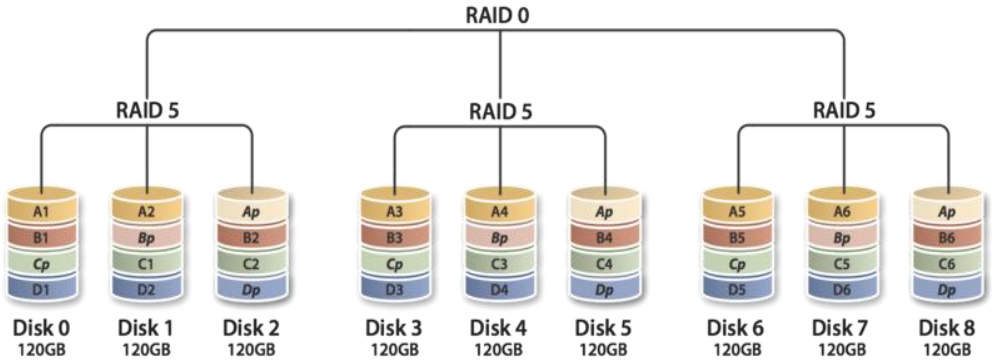
需要注意，使用 RAID 10 来称呼其实很容易产生混淆，因为 RAID 0+1 和 RAID 10 基本上只是两个数字交换了一下位置，但是对 RAID 来说就是两个不同的组成。因此，更容易理解的方式是“Striped mirrors”，即：条带化后的镜像——RAID 10；或者“mirrored stripes”，即：镜像后的条带化。比较 RAID 10 和 RAID 0+1，虽然最终都是用到了 4 块盘，但是在数据组织上有所不同，从而带来问题。RAID 10 在可用性上是要高于 RAID 0+1 的：

- RAID 0+1 任何一块盘损坏，将失去冗余。如图 4 块盘中，右侧一组损坏一块盘，左侧一组损坏一块盘，整个盘阵将无法使用。而 RAID 10 左右各损坏一块盘，盘阵仍然可以工作。
- RAID 0+1 损坏后的恢复过程会更慢。因为先经过的 mirror，所以左右两组中保存的都是完整的数据，数据恢复时，需要完整恢复所以数据。而 RAID 10 因为先条带化，因此损坏数据以后，恢复的只是本条带的数据。如图 4 块盘，数据少了一半。



RAID 50

RAID 50　同 RAID 10，先做条带化以后，在做 RAID 5。兼顾性能，同时又保证空间的利用率。RAID 50 最少需要 6 块盘。

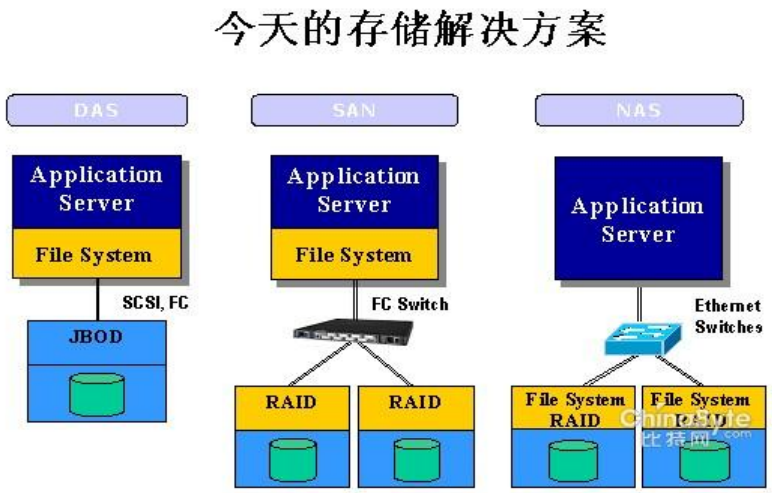


总结：

- RAID 与 LVM 中的条带化原理上类似，只是实现层面不同。在存储上实现的 RAID 一般有专门的芯片来完成，因此速度上远比 LVM 块。也称硬 RAID。
- 如上介绍，RAID 的使用是有风险的，如 RAID 0，一块盘损坏会导致所有数据丢失。因此，在实际使用中，高性能环境会使用 RAID 10，兼顾性能和安全；一般情况下使用 RAID 5（RAID 50），兼顾空间利用率和性能；

说说 IO（八）- 三分天下

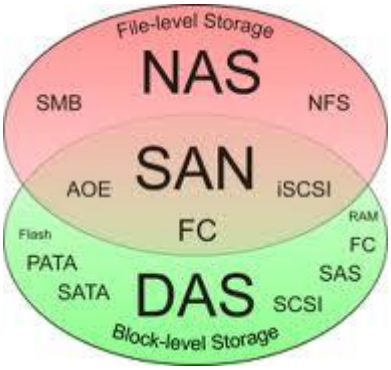
DAS、SAN 和 NAS



为了满足人们不断扩大的需求，存储方案也是在发展的。而 DAS、SAN、NAS 直接反映这种反映了这种趋势。

- **单台主机**。在这种情况下，存储作为主机的一个或多个磁盘存在，这样局限性也是很明显的。由于受限于主机空间，一个主机只能装一块到几块硬盘，而硬盘空间时受限的，当磁盘满了以后，你不得不为主机更换更大空间的硬盘。
- **独立存储空间**。为了解决空间的问题，于是考虑把磁盘独立出来，于是有了 DAS（Direct Attached Storage），即：直连存储。DAS 就是一组磁盘的集合体，数据读取和写入等也都是由主机来控制。但是，随之而来，DAS 又面临了一个他无法解决的问题——存储空间的共享。接某个主机的 JBOD（Just a Bunch Of Disks，磁盘组），只能这个主机使用，其他主机无法用。因此，如果 DAS 解决空间了，那么他无法解决的就是如果让空间能够在多个机器共享。**因为 DAS 可以理解为与磁盘交互，DAS 处理问题的层面相对更低。使用协议都是跟磁盘交互的协议**
- **独立的存储网络**。为了解决共享的问题，借鉴以太网的思想，于是有了 SAN（Storage Area Network），即：存储网络。对于 SAN 网络，你能看到两个非常特点，一个就是光纤网络，另一个是光纤交换机。**SAN 网络由于不会直接跟磁盘交互，他考虑的更多是数据存取的问题，因此使用的协议相对 DAS 层面更高一些。**
  - 光纤网络：对于存储来说，与以太网很大的一个不同就是他对带宽的要求非常高，因此 SAN 网络下，光纤成为了其连接的基础。而其上的光纤协议相比以太网协议而言，也被设计的更为简洁，性能也更高。
  - 光纤交换机：这个类似以太网，如果想要做到真正的“网络”，交换机是基础。

- 网络文件系统。存储空间可以共享，那文件也是可以共享的。NAS（Network attached storage）相对上面两个，看待问题的层面更高，NAS 是在文件系统级别看待问题。因此他面的不再是存储空间，而是单个的文件。因此，当 NAS 和 SAN、DAS 放在一起时，很容易引起混淆。**NAS 从文件的层面考虑共享，因此 NAS 相关协议都是文件控制协议。**
    - NAS 解决的是文件共享的问题；SAN（DAS）解决的是存储空间的问题。
    - NAS 要处理的对象是文件；SAN（DAS）要处理的是磁盘。
    - 为 NAS 服务的主机必须是一个完整的主机（有 OS、有文件系统，而存储则不一定有，因为可以他后面又接了一个 SAN 网络），他考虑的是如何在各个主机直接高效的共享文件；为 SAN 提供服务的是存储设备（可以是个完整的主机，也可以是部分），它考虑的是数据怎么分布到不同磁盘。
    - NAS 使用的协议是控制文件的（即：对文件的读写等）；SAN 使用的协议是控制存储空间的（即：把多长的一串二进制写到某个地址）



如图，对 NAS、SAN、DAS 的组成协议进行了划分，从这里也能很清晰的看出他们之间的差别。

NAS：涉及 SMB 协议、NFS 协议，都是网络文件系统的协议。

SAN：有 FC、iSCSI、AOE，都是网络数据传输协议。

DAS：有 PATA、SATA、SAS 等，主要是磁盘数据传输协议。

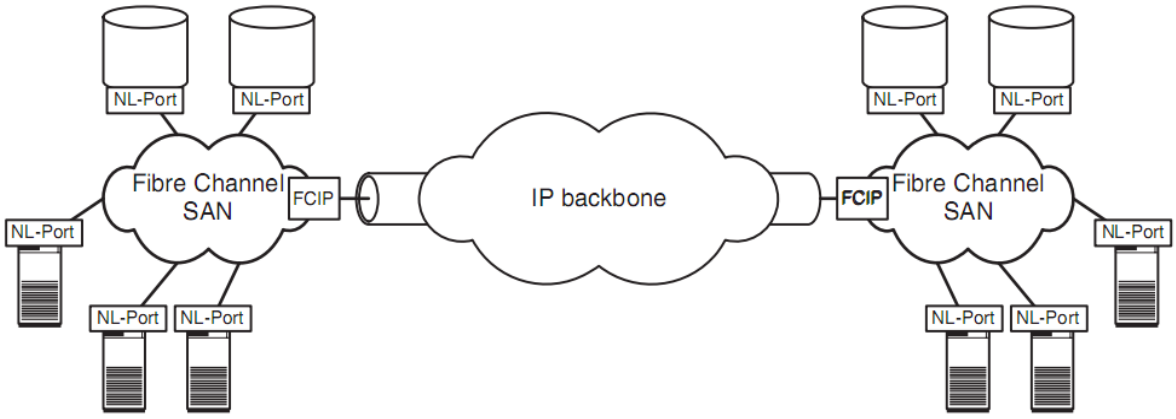
从 DAS 到 SAN，在到 NAS，在不同层面对存储方案进行的补充，也可以看到一种从低级到高级的发展趋势。而现在我们常看到一些分布式文件系统（如 hadoop 等）、数据库的 sharding 等，从存储的角度来说，则是在 OS 层面（应用）对数据进行存储。从这也能看到一种技术发展的趋势。

跑在以太网上的 SAN

SAN 网络并不是只能使用光纤和光纤协议，当初之所以使用 FC，传输效率是一个很大的问题，但是以太网发展到今天被不断的完善、加强，带宽的问题也被不断的解决。因此，以太网上的 SAN 或许会成为一个趋势。

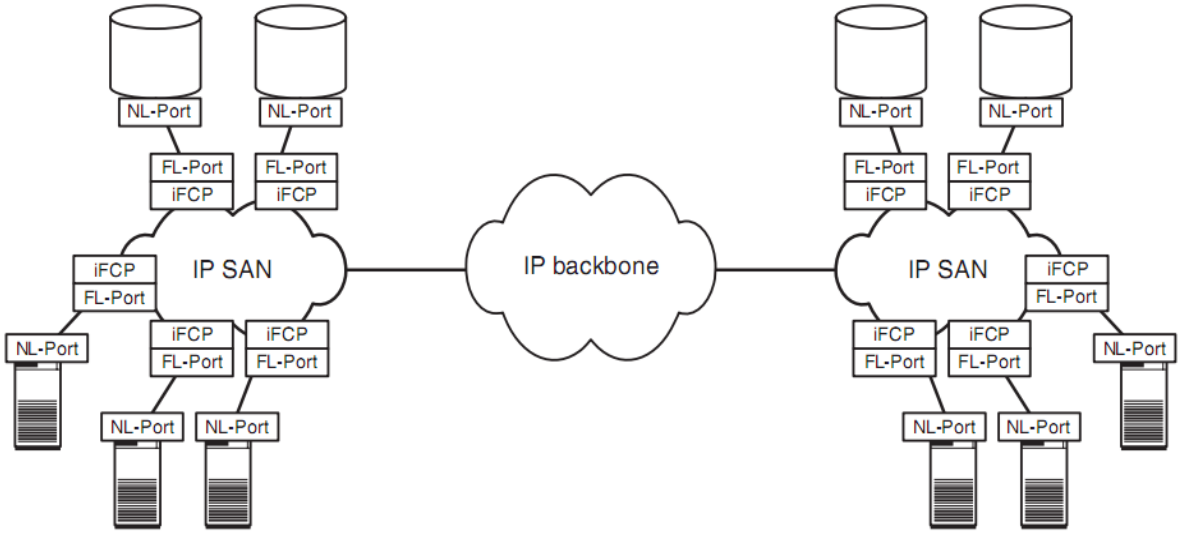
- FCIP

如图两个 FC 的 SAN 网络，通过 FCIP 实现了两个 SAN 网络数据在 IP 网络上的传输。这个时候 SAN 网络还是以 FC 协议为基础，还是使用光纤。



- iFCP

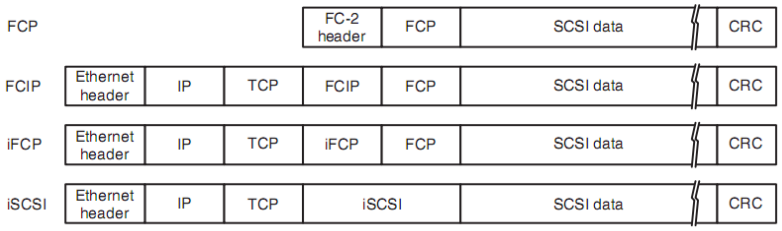
通过 iFCP 方式，SAN 网络由 FC 的 SAN 网络演变为 IP SAN 网络，整个 SAN 网络都基于了 IP 方式。但是主机和存储直接使用的还是 FC 协议。只是在接入 SAN 网络的时候通过 iFCP 进行了转换



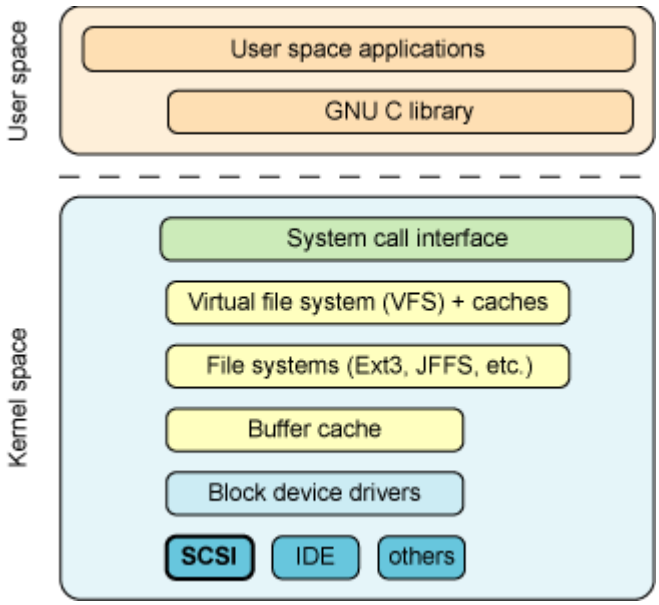
- iSCSI

iSCSI 是比较主流的 IP SAN 的提供方式，而且其效率也得到了认可。

	End device	Fabric services	Transport
FCP	Fibre Channel	Fibre Channel	Fibre Channel
FCIP	Fibre Channel	Fibre Channel	IP/Ethernet
iFCP	Fibre Channel	IP/Ethernet	IP/Ethernet
iSCSI	IP/Ethernet	IP/Ethernet	IP/Ethernet

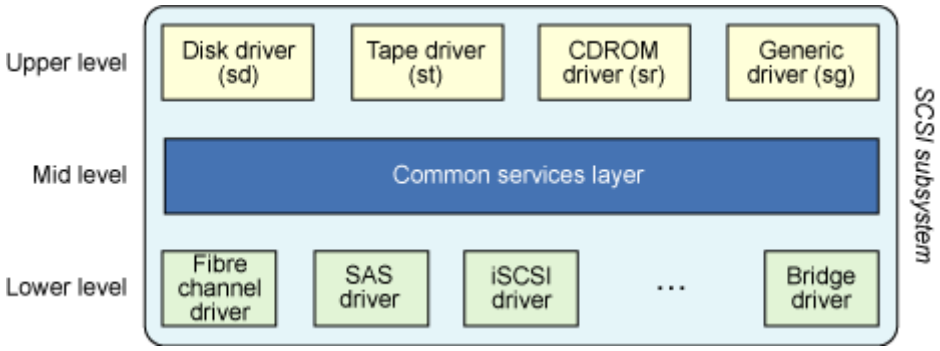


对于 iSCSI，最重要的一点就是 SCSI 协议。SCSI（Small Computer Systems Interface）协议是计算机内部的一个通用协议。是一组标准集，它定义了与大量设备（主要是与存储相关的设备）通信所需的接口和协议。如图，SCSI 为 block device drivers 之下。



从 SCIS 的分层来看，共分三层：

高层：提供了与 OS 各种设备之间的接口，实现把 OS 如：Linux 的 VFS 请求转换为 SCSI 请求
中间层：实现高层和底层之间的转换，类似一个协议网关。
底层：完成于具体物理设备之间的交互，实现真正的数据处理。


Upper level
Disk driver (sd)
Tape driver (st)
CDROM driver (sr)
Generic driver (sg)
Mid level
Common services layer
Lower level
Fibre channel driver
SAS driver
iSCSI driver
...
Bridge driver
SCSI subsystem