硬盘知识，硬盘逻辑结构

你新买来的硬盘是不能直接使用的，必须对它进行分区并进行格式化的才能储存数据。硬盘分区是操作系统安装过程中经常谈到的话题。对于一些简单的应用，硬盘分区并不成为一种障碍，但对于一些复杂的应用，就不能不深入理解硬盘分区机制的某些细节。硬盘的崩溃经常会遇见，特别是病毒肆虐的时代，关于引导分区的恢复与备份的技巧，你一定要掌握。在使用电脑时，你往往会使用几个操作系统。如何在硬盘中安装多个操作系统？如果你需要了解这方面的知识或是要解决上述问题，这期的"硬盘分区"专题会告诉你答案！

硬盘是现在计算机上最常用的存储器之一。我们都知道，计算机之所以神奇，是因为它具有高速分析处理数据的能力。而这些数据都以文件的形式存储在硬盘里。不过，计算机可不像人那么聪明。在读取相应的文件时，你必须要给出相应的规则。这就是分区概念。分区从实质上说就是对硬盘的一种格式化。当我们创建分区时，就已经设置好了硬盘的各项物理参数，指定了硬盘主引导记录（即 Master Boot Record，一般简称为 MBR）和引导记录备份的存放位置。而对于文件系统以及其他操作系统管理硬盘所需要的信息则是通过以后的高级格式化，即 Format 命令来实现。面、磁道和扇区硬盘分区后，将会被划分为面（Side）、磁道（Track）和扇区（Sector）。需要注意的是，这些只是个虚拟的概念，并不是真正在硬盘上划轨道。先从面说起，硬盘一般是由一片或几片圆形薄膜叠加而成。我们所说，每个圆形薄膜都有两个"面"，这两个面都是用来存储数据的。按照面的多少，依次称为 0 面、1 面、2 面……由于每个面都专有一个读写磁头，也常用 0 头(head)、1 头……称之。按照硬盘容量和规格的不同，硬盘面数(或头数)也不一定相同，少的只有 2 面，多的可达数十面。各面上磁道号相同的磁道合起来，称为一个柱面(Cylinder)。

上面我们提到了磁道的概念。那么究竟何为磁道呢？由于磁盘是旋转的，则连续写入的数据是排列在一个圆周上的。我们称这样的圆周为一个磁道。如果读写磁头沿着圆形薄膜的半径方向移动一段距离，以后写入的数据又排列在另外一个磁道上。根据硬盘规格的不同，磁道数可以从几百到数千不等；一个磁道上可以容纳数 KB 的数据，而主机读写时往往并不需要一次读写那么多，于是，磁道又被划分成若干段，每段称为一个扇区。一个扇区一般存放 512 字节的数据。扇区也需要编号，同一磁道中的扇区，分别称为 1 扇区，2 扇区……

计算机对硬盘的读写，处于效率的考虑，是以扇区为基本单位的。即使计算机只需要硬盘上存储的某个字节，也必须一次把这个字节所在的扇区中的 512 字节全部读入内存，再使用所需的那个字节。不过，在上文中我们也提到，硬盘上面、磁道、扇区的划分表面上是看不到任何痕迹的，虽然磁头可以根据某个磁道的应有半径来对准这个磁道，但怎样才能在首尾相连的一圈扇区中找出所需要的某一扇区呢？原来，每个扇区并不仅仅由 512 个字节组成的，在这些由计算机存取的数据的前、后两端，都另有一些特定的数据，这些数据构成了扇区的界限标志，标志中含有扇区的编号和其他信息。计算机就凭借着这些标志来识别扇区。硬盘的数据结构鎷鎷在上文中，我们谈了数据在硬盘中的存储的一般原理。为了能更深入地了解硬盘，我们还必须对硬盘的数据结构有个简单的了解。硬盘上的数据按照其不同的特点和作用大致可分为 5 部分：MBR 区、DBR 区、FAT 区、DIR 区和 DATA 区。

我们来分别介绍一下：

1．MBR 区鏸鏸MBR（Main Boot Record 主引导记录区）位于整个硬盘的 0 磁道 0 柱面 1 扇区。不过，在总共 512 字节的主引导扇区中，MBR 只占用了其中的 446 个字节，另外的 64 个字节交给了 DPT（Disk Partition Table 硬盘分区表），最后两个字节"55，AA"是分区的结束标志。这个整体构成了硬盘的主引导扇区。

主引导记录中包含了硬盘的一系列参数和一段引导程序。其中的硬盘引导程序的主要作用是检查分区表是否正确并且在系统硬件完成自检以后引导具有激活标志的分区上的操作系统，并将控制权交给启动程序。MBR 是由分区程序（如 Fdisk．exe）所产生的，它不依赖任何操作系统，而且硬盘引导程序也是可以改变的，从而实现多系统共存。

下面，我们以一个实例让大家更直观地来了解主引导记录：

例：80 01 01 00 0B FE BF FC 3F 00 00 00 7E 86 BB 00鏸鏸在这里我们可以看到，最前面的"80"是一个分区的激活标志，表示系统可引导；"01 01 00"表示分区开始的磁头号为 01，开始的扇区号为 01，开始的柱面号为 00；"0B"表示分区的系统类型是 FAT32，其他比较常用的有 04（FAT16）、07（NTFS）；"FE BF FC"表示分区结束的磁头号为 254，分区结束的扇区号为 63、分区结束的柱面号为 764；"3F 00 00 00"表示首扇区的相对扇区号为 63；"7E 86 BB 00"表示总扇区数为 12289622。

2．DBR 区鏸鏸DBR（Dos Boot Record）是操作系统引导记录区的意思。它通常位于硬盘的 0 磁道 1 柱面 1 扇区，是操作系统可以直接访问的第一个扇区，它包括一个引导程序和一个被称为 BPB（Bios Parameter Block）的本分区参数记录表。引导程序的主要任务是当 MBR 将系统控制权交给它时，判断本分区跟目录前两个文件是不是操作系统的引导文件（以 DOS 为例，即是 Io．sys 和 Msdos．sys）。如果确定存在，就把它读入内存，并把控制权 交给该文件。BPB 参数块记录着本分区的起始扇区、结束扇区、文件存储格式、硬盘介质描述符、根目录大小、FAT 个数，分配单元的大小等重要参数。DBR 是由高级格式化程序（即 Format．com 等程序）所产生的。

3．FAT 区鏸鏸在 DBR 之后的是我们比较熟悉的 FAT（File Allocation Table 文件分配表）区。在解释文件分配表的概念之前，我们先来谈谈簇（Cluster）的概念。文件占用磁盘空间时，基本单位不是字节而是簇。一般情况下，软盘每簇是 1 个扇区，硬盘每簇的扇区数与硬盘的总容量大小有关，可能是 4、8、16、32、64……鏸鏸同一个文件的数据并不一定完整地存放在磁盘的一个连续的区域内，而往往会分成若干段，像一条链子一样存放。这种存储方式称为文件的链式存储。由于硬盘上保存着段与段之间的连接信息（即 FAT），操作系统在读取文件时，总是能够准确地找到各段的位置并正确读出。鏸鏸为了实现文件的链式存储，硬盘上必须准确地记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继内容的下一个簇的簇号。对一个文件的最后一簇，则要指明本簇无后继簇。这些都是由 FAT 表来保存的，表中有很多表项，每项记录一个簇的信息。由于 FAT 对于文件管理的重要性，所以 FAT 有一个备份，即在原 FAT 的后面再建一个同样的 FAT。初形成的 FAT 中所有项都标明为"未占用"，但如果磁盘有局部损坏，那么格式化程序会检测出损坏的簇，在相应的项中标为"坏簇"，以后存文件时就不会再使用

这个簇了。FAT 的项数与硬盘上的总簇数相当，每一项占用的字节数也要与总簇数相适应，因为其中需要存放簇号。FAT 的格式有多种，最为常见的是 FAT16 和 FAT32。

4．DIR 区鎷鎷DIR（Directory）是根目录区，紧接着第二 FAT 表（即备份的 FAT 表）之后，记录着根目录下每个文件(目录)的起始单元，文件的属性等。定位文件位置时，操作系统根据 DIR 中的起始单元，结合 FAT 表就可以知道文件在硬盘中的具体位置和大小了。

5．数据(DATA)区鎷鎷数据区是真正意义上的数据存储的地方，位于 DIR 区之后，占据硬盘上的大部分数据空间。

FAT 文件系统原理

一、硬盘的物理结构：

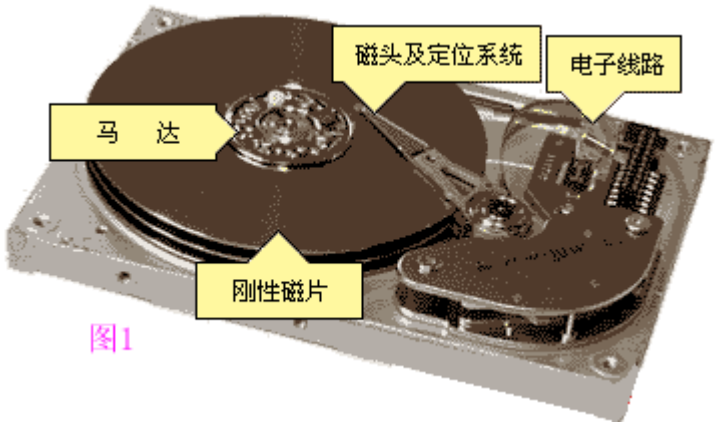


图1

硬盘存储数据是根据电、磁转换原理实现的。硬盘由一个或几个表面镀有磁性物质的金属或玻璃等物质盘片以及盘片两面所安装的磁头和相应的控制电路组成(图 1)，其中盘片和磁头密封在无尘的金属壳中。

硬盘工作时，盘片以设计转速高速旋转，设置在盘片表面的磁头则在电路控制下径向移动到指定位置然后将数据存储或读取出来。当系统向硬盘写入数据时，磁头中"写数据"电流产生磁场使盘片表面磁性物质状态发生改变，并在写电流磁场消失后仍能保持，这样数据就存储下来了；当系统从硬盘中读数据时，磁头经过盘片指定区域，盘片表面磁场使磁头产生感应电流或线圈阻抗产生变化，经相关电路处理后还原成数据。因此只要能将盘片表面处理得更平滑、磁头设计得更精密以及尽量提高盘片旋转速度，就能造出容量更大、读写数据速度更快的硬盘。这是因为盘片表面处理越平、转速越快就能越使磁头离盘片表面越近，提高读、写灵敏度和速度；磁头设计越小越精密就能使磁头在盘片上占用空间越小，使磁头在一张盘片上建立更多的磁道以存储更多的数据。二、硬盘的逻辑结构。
硬盘由很多盘片(platter)组成，每个盘片的每个面都有一个读写磁头。如果有 N 个盘片。就有 2N 个面，对应 2N 个磁头(Heads)，从 0、1、2 开始编号。每个盘片被划分成若干

个同心圆磁道(逻辑上的，是不可见的。)每个盘片的划分规则通常是一样的。这样每个盘片的半径均为固定值 R 的同心圆再逻辑上形成了一个以电机主轴为轴的柱面(Cylinders)，从外至里编号为 0、1、2……每个盘片上的每个磁道又被划分为几十个扇区(Sector)，通常的容量是 512byte，并按照一定规则编号为 1、2、3……形成 Cylinders×Heads×Sector 个扇区。这三个参数即是硬盘的物理参数。我们下面的很多实践需要深刻理解这三个参数的意义。三、磁盘引导原理。3.1 MBR(master boot record)扇区：

计算机在按下 power 键以后，开始执行主板 bios 程序。进行完一系列检测和配置以后。开始按 bios 中设定的系统引导顺序引导系统。假定现在是硬盘。Bios 执行完自己的程序后如何把执行权交给硬盘呢。交给硬盘后又执行存储在哪里的程序呢。其实，称为 mbr 的一段代码起着举足轻重的作用。MBR(master boot record),即主引导记录，有时也称主引导扇区。位于整个硬盘的 0 柱面 0 磁头 1 扇区(可以看作是硬盘的第一个扇区)，bios 在执行自己固有的程序以后就会 jump 到 mbr 中的第一条指令。将系统的控制权交由 mbr 来执行。在总共 512byte 的主引导记录中，MBR 的引导程序占了其中的前 446 个字节(偏移 0H~偏移 1BDH)，随后的 64 个字节(偏移 1BEH~偏移 1FDH)为 DPT(Disk PartitionTable，硬盘分区表)，最后的两个字节"55 AA"(偏移 1FEH~偏移 1FFH)是分区有效结束标志。

MBR 不随操作系统的不同而不同，意即不同的操作系统可能会存在相同的 MBR，即使不同，MBR 也不会夹带操作系统的性质。具有公共引导的特性。

我们来分析一段 mbr。下面是用 winhex 查看的一块希捷 120GB 硬盘的 mbr。

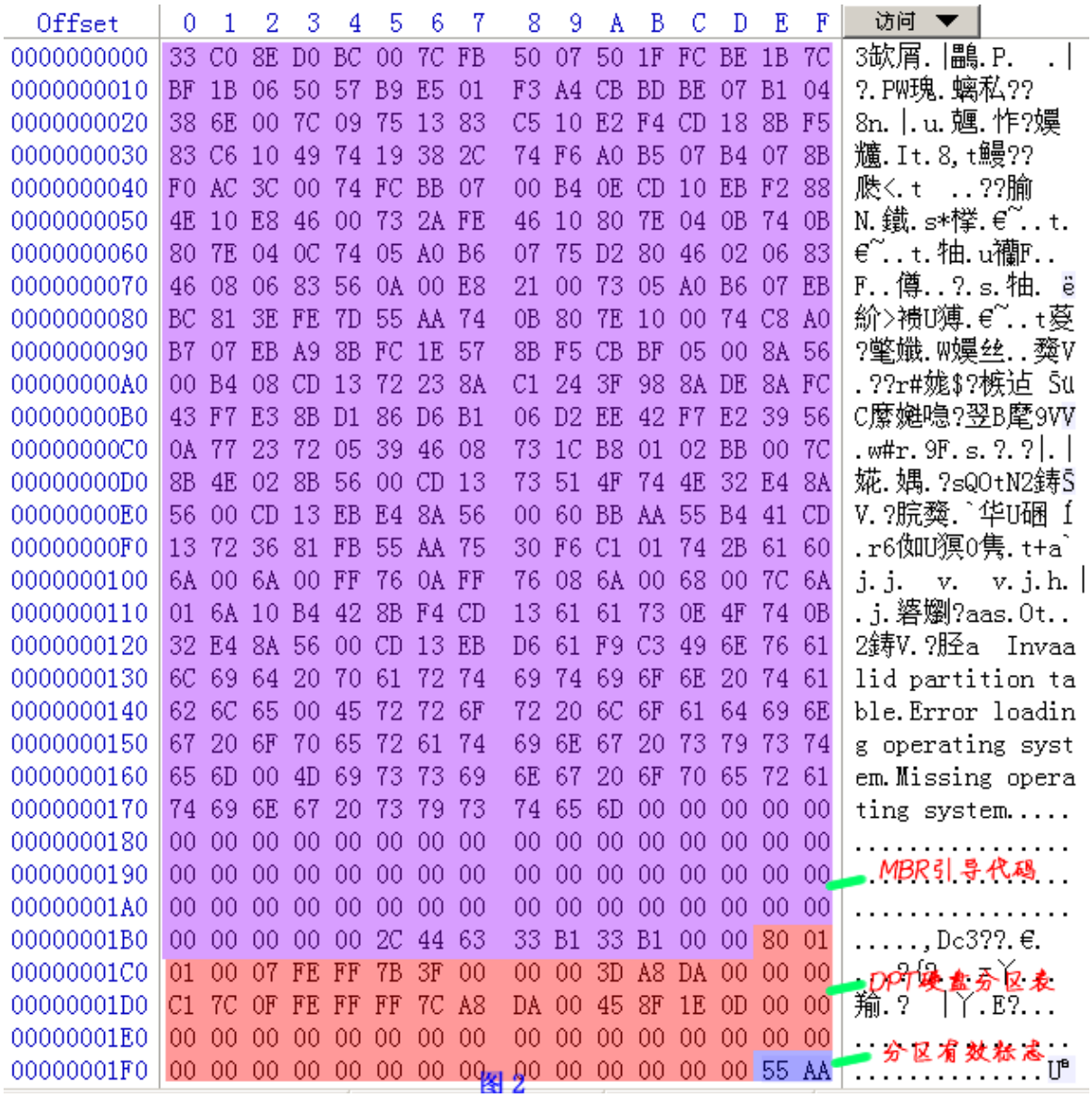


图 2

你的硬盘的 MBR 引导代码可能并非这样。不过即使不同，所执行的功能大体是一样的。这是 wowocock 关于磁盘 mbr 的反编译，已加了详细的注释，感兴趣可以细细研究一下。

我们看 DPT 部分。操作系统为了便于用户对磁盘的管理。加入了磁盘分区的概念。即将一块磁盘逻辑划分为几块。磁盘分区数目的多少只受限于 C～Z 的英文字母的数目，在上图 DPT 共 64 个字节中如何表示多个分区的属性呢? Microsoft 通过链接的方法解决了这个问题。在 DPT 共 64 个字节中，以 16 个字节为分区表项单位描述一个分区的属性。也就是说，第一个分区表项描述一个分区的属性，一般为基本分区。第二个分区表项描述除基本分区外的其余空间，一般而言，就是我们所说的扩展分区。这部分的大体说明见表 1。

[img]http://www.easeus.com.cn/knowledge/fat-elements.htm[/img]
注：上表中的超过 1 字节的数据都以实际数据显示，就是按高位到地位的方式显示。存

储时是按低位到高位存储的。两者表现不同，请仔细看清楚。以后出现的表，图均同。
也可以在 winhex 中看到这些参数的意义：

Master Boot Record, 基础偏移量：0

Offset 偏移	标题	数值
0	Master bootstrap loader code 引导代码	33 C0 8E D0 BC 00 7C FB
	Partition Table Entry #1	
1BE	80 = active partition 活动分区标志	80 80表示活动，00表示非
1BF	Start head 开始磁头	1
1C0	Start sector 开始扇区	1
1C0	Start cylinder 开始柱面	0
1C2	Operating system indicator (hex	07 分区类型标志
1C3	End head 结束磁头	254
1C4	End sector 结束扇区	63
1C4	End cylinder 结束柱面	891
1C6	Sectors preceding partition 1	63 本分区之前的扇区数
1CA	Length of partition 1 in sector	14329917 本分区的扇区总数
	Partition Table Entry #2	
1CE	80 = active partition	00
1CF	Start head	0
1D0	Start sector	1
1D0	Start cylinder	892
1D2	Operating system indicator (hex	0F
1D3	End head	254
1D4	End sector	63
1D4	End cylinder	1023
1D6	Sectors preceding partition 2	14329980
1DA	Length of partition 2 in sector	220106565
	Partition Table Entry #3	
1DE 0H	80 = active partition	00
1DF 1H	Start head	0
1E0 2H	Start sector	0
1E0 2H	Start cylinder	0
1E2 4H	Operating system indicator (hex	00
1E3 5H	End head	0
1E4 6H	End sector	0
1E4 6H	End cylinder	0
1E6 8H	Sectors preceding partition 3	0
1EA CH	Length of partition 3 in sector	0
	Partition Table Entry #4	
1EE	80 = active partition	00
1EF	Start head	0
1F0	Start sector	0
1F0	Start cylinder	0
1F2	Operating system indicator (hex	00
1F3	End head	0
1F4	End sector	0
1F4	End cylinder	0
1F6	Sectors preceding partition 4	0

图3

说明： 每个分区表项占用 16 个字节，假定偏移地址从 0 开始。如图 3 的分区表项 3。分区表项 4 同分区表项 3。
1、0H 偏移为活动分区是否标志，只能选 00H 和 80H。80H 为活动，00H 为非活动。其余值对 microsoft 而言为非法值。

2、重新说明一下(这个非常重要)：大于 1 个字节的数被以低字节在前的存储格式格式(little endian format)或称反字节顺序保存下来。低字节在前的格式是一种保存数的方法，这样，最低位的字节最先出现在十六进制数符号中。例如，相对扇区数字段的值 0x3F000000 的低字节在前表示为 0x0000003F。这个低字节在前的格式数的十进制数为 63。

3、系统在分区时，各分区都不允许跨柱面，即均以柱面为单位，这就是通常所说的分区粒度。有时候我们分区是输入分区的大小为 7000M，分出来却是 6997M，就是这个原因。 偏移 2H 和偏移 6H 的扇区和柱面参数中,扇区占 6 位(bit)，柱面占 10 位(bit)，以偏移 6H 为例，其低 6 位用作扇区数的二进制表示。其高两位做柱面数 10 位中的高两位，偏移 7H 组成的 8 位做柱面数 10 位中的低 8 位。由此可知，实际上用这种方式表示的分区容量是有限的，柱面和磁头从 0 开始编号,扇区从 1 开始编号,所以最多只能表示 1024 个柱面×63 个扇区×256 个磁头×512byte=8455716864byte。即通常的 8.4GB(实际上应该是 7.8GB 左右)限制。实际上磁头数通常只用到 255 个(由汇编语言的寻址寄存器决定),即使把这 3 个字节按线性寻址,依然力不从心。在后来的操作系统中，超过 8.4GB 的分区其实已经不通过 C/H/S 的方式寻址了。而是通过偏移 CH～偏移 FH 共 4 个字节 32 位线性扇区地址来表示分区所占用的扇区总数。可知通过 4 个字节可以表示 2^32 个扇区，即 2TB=2048GB，目前对于大多数计算机而言，这已经是个天文数字了。在未超过 8.4GB 的分区上，C/H/S 的表示方法和线性扇区的表示方法所表示的分区大小是一致的。也就是说，两种表示方法是协调的。即使不协调，也以线性寻址为准。(可能在某些系统中会提示出错)。超过 8.4GB 的分区结束 C/H/S 一般填充为 FEH FFH FFH。即 C/H/S 所能表示的最大值。有时候也会用柱面对 1024 的模来填充。不过这几个字节是什么其实都无关紧要了。
虽然现在的系统均采用线性寻址的方式来处理分区的大小。但不可跨柱面的原则依然没变。本分区的扇区总数加上与前一分区之间的保留扇区数目依然必须是柱面容量的整数倍。(保留扇区中的第一个扇区就是存放分区表的 MBR 或虚拟 MBR 的扇区，分区的扇区总数在线性表示方式上是不计入保留扇区的。如果是第一个分区，保留扇区是本分区前的所有扇区。

附：分区表类型标志如图 4

分区类型标志：

00 空，mocrosoft不允许使用。	63 GNU HURD or Sys
01 FAT32	64 Novell Netware
02 XENIX root	65 Novell Netware
03 XENIX usr	70 Disk Secure Mult
04 FAT16 <32M	75 PC/IX
05 Extended	80 Old Minix
06 FAT16	81 Minix/Old Linux
07 HPFS/NTFS	82 Linux swap
08 AIX	83 Linux
09 AIX bootable	84 OS/2 hidden C:
0A OS/2 Boot Manage	85 Linux extended
0B Win95 FAT32	86 NTFS volume set
0C Win95 FAT32	87 NTFS volume set
0E Win95 FAT16	93 Amoeba
0F Win95 Extended(>8GB)	94 Amoeba BBT
10 OPUS	A0 IBM Thinkpad hidden
11 Hidden FAT12	A5 BSD/386
12 Compaq diagnost	A6 Open BSD
16 HiddenFAT16	A7 NextSTEP
14 Hidden FAT16<32GB	B7 BSDI fs
17 Hidden HPFS/NTFS	B8 BSDI swap
18 AST Windows swap	BE Solaris boot
1B Hidden FAT32	partition
1C Hidden FAT32 partition	C0 DR-DOS/Novell DOS
(using LBA-mode	secured partition
INT 13 extensions)	C1 DRDOS/sec
1E Hidden LBA VFAT partition	C4 DRDOS/sec
24 NEC DOS	C6 DRDOS/sec
3C Partition Magic	C7 Syrinx
40 Venix 80286	DB CP/M/CTOS
41 PPC PreP Boot	E1 DOS access
42 SFS	E3 DOS R/O
4D QNX4.x	E4 SpeedStor
4E QNX4.x 2nd part	EB BeOS fs
4F QNX4.x 3rd part	F1 SpeedStor
50 Ontrack DM	F2 DOS 3.3+ secondary
51 Ontrack DM6 Aux	partition
52 CP/M	F4 SpeedStor
53 oNtRACK DM6 Aux	FE LAN step
54 OnTrack DM6	FF BBT
55 EZ-Drive	
56 Golden Bow	
5C Priam Edisk	
61 Speed Stor	

图4

3.2 扩展分区：

扩展分区中的每个逻辑驱动器都存在一个类似于 MBR 的扩展引导记录(Extended Boot Record, EBR)，也有人称之为虚拟 mbr 或扩展 mbr，意思是一样的。扩展引导记录包括一个扩展分区表和该扇区的标签。扩展引导记录将记录只包含扩展分区中每个逻辑驱动器的第一个柱面的第一面的信息。一个逻辑驱动器中的引导扇区一般位于相对扇区 32 或 63。但是，如果磁盘上没有扩展分区，那么就不会有扩展引导记录和逻辑驱动器。第一个逻辑驱动器的扩展分区表中的第一项指向它自身的引导扇区。第二项指向下一个逻辑驱动器的 EBR。如果不存在进一步的逻辑驱动器，第二项就不会使用，而且被记录成一系列零。如果有附加的逻辑驱动器，那么第二个逻辑驱动器的扩展分区表的第一项会指向它本身的引导扇区。第二个逻辑驱动器的扩展分区表的第二项指向下一个逻辑驱动器的 EBR。扩展分区表的第三项和第四项永远都不会被使用。

通过一幅 4 分区的磁盘结构图可以看到磁盘的大致组织形式。如图 5：

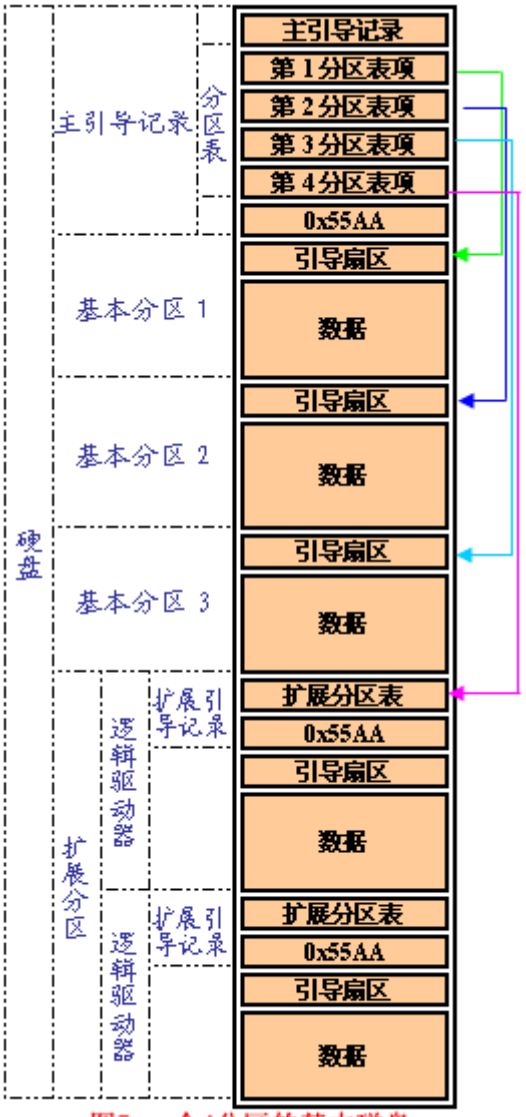


图5 一个4分区的基本磁盘

关于扩展分区，如图 6 所示，扩展分区中逻辑驱动器的扩展引导记录是一个连接表。该图显示了一个扩展分区上的三个逻辑驱动器，说明了前面的逻辑驱动器和最后一个逻辑驱动器之间在扩展分区表中的差异。

除了扩展分区上最后一个逻辑驱动器外，表 2 中所描述的扩展分区表的格式在每个逻辑驱动器中都是重复的：第一个项标识了逻辑驱动器本身的引导扇区，第二个项标识了下一个逻辑驱动器的 EBR。最后一个逻辑驱动器的扩展分区表只会列出它本身的分区项。最后一个扩展分区表的第二个项到第四个项被使用。

[img]http://www.easeus.com.cn/knowledge/fat-elements.htm[/img]

扩展分区表项中的相对扇区数字段所显示的是从扩展分区开始到逻辑驱动器中第一个扇区的位移的字节数。总扇区数字段中的数是指组成该逻辑驱动器的扇区数目。总扇区数字段的值等于从扩展分区表项所定义的引导扇区到逻辑驱动器末尾的扇区数。有时候在磁盘的末尾会有剩余空间，剩余空间是什么呢？我们前面说到，分区是以 1 柱面的容量为分区粒度的，那么如果磁盘总空间不是整数个柱面的话，不够一个柱面的剩下的空间就是剩余空间了，这部分空间并不参与分区，所以一般无法利用。照道理说，磁盘的

物理模式决定了磁盘的总容量就应该是整数个柱面的容量，为什么会有不够一个柱面的空间呢。在我的理解看来，本来现在的磁盘为了更大的利用空间，一般在物理上并不是按照外围的扇区大于里圈的扇区这种管理方式，只是为了与操作系统兼容而抽象出来CHS。可能其实际空间容量不一定正好为整数个柱面的容量。

磁盘阵列(Disk Array)原理

1、为什么需要磁盘阵列?

如何增加磁盘的存取（access）速度，如何防止数据因磁盘的故障而失落及如何有效的利用磁盘空间，一直是电脑专业人员和用户的困扰；而大容量磁盘的价格非常昂贵，对用户形成很大的负担。磁盘阵列技术的产生一举解决了这些问题。

过去十几年来，CPU 的处理速度增加了五十倍有多，内存(memory)的存取速度亦大幅增加，而数据储存装置--主要是磁盘（hard disk）--的存取速度只增加了三、四倍，形成电脑系统的瓶颈，拉低了电脑系统的整体性能（through put），若不能有效的提升磁盘的存取速度，CPU、内存及磁盘间的不平衡将使 CPU 及内存的改进形成浪费。

目前改进磁盘存取速度的的方式主要有两种。一是磁盘快取控制（disk cache controller），它将从磁盘读取的数据存在快取内存（cache memory）中以减少磁盘存取的次数，数据的读写都在快取内存中进行，大幅增加存取的速度，如要读取的数据不在快取内存中，或要写数据到磁盘时，才做磁盘的存取动作。这种方式在单工环境（single-tasking envioronment）如 DOS 之下，对大量数据的存取有很好的性能（量小且频繁的存取则不然），但在多工（multi-tasking）环境之下（因为要不停的作数据交换（swapping）的动作）或数据库（database）的存取（因为每一记录都很小）就不能显示其性能。这种方式没有任何安全保障。

其二是使用磁盘阵列的技术。磁盘阵列是把多个磁盘组成一个阵列，当作单一磁盘使用，它将数据以分段（striping）的方式储存在不同的磁盘中，存取数据时，阵列中的相关磁盘一起动作，大幅减低数据的存取时间，同时有更佳的空间利用率。磁盘阵列所利用的不同的技术，称为 RAID level，不同的 level 针对不同的系统及应用，以解决数据安全的问题。

一般高性能的磁盘阵列都是以硬件的形式来达成，进一步的把磁盘快取控制及磁盘阵列结合在一个控制器(RAID controler 或控制卡上，针对不同的用户解决人们对磁盘输出入系统的四大要求：
(1)增加存取速度；
(2)容错(fault tolerance)，即安全性；
(3)有效的利用磁盘空间；
(4)尽量的平衡 CPU，内存及磁盘的性能差异，提高电脑的整体工作性能。 2.磁盘阵列原理

磁盘阵列中针对不同的应用使用的不同技术，称为 RAID level，RAID 是 Redundent Array of Inexpensive Disks 的缩写，而每一 level 代表一种技术，目前业界公认的标准是 RAID 0~RAID 5。这个 level 并不代表技术的高低，level 5 并不高于 level 3，level 1 也不低过 level 4，至于要选择那一种 RAID level 的产品，纯视用户的操作环境（operating environment）及应用（application）而定，与 level 的高低没有必然的关系。

RAID 0 及 RAID 1 适用于 PC 及 PC 相关的系统如小型的网络服务器（network server）及需要高磁盘容量与快速磁盘存取的工作站等，比较便宜；RAID 3 及 RAID 4 适用于大型电脑及影像、CAD/CAM 等处理；RAID 5 多用于 OLTP（在线事务处理），因有金融机构及大型数据处理中心的迫切需要,故使用较多而较有名气，RAID 2 较少使用，其他如 RAID 6、RAID 7 乃至 RAID 10 等，都是厂商各做各的，并无一致的标准，在此不作说明。介绍各个 RAID level 之前，先看看形成磁盘阵列的两个基本技术：

磁盘延伸（Disk Spanning）：

译为磁盘延伸，能确切的表示 disk spanning 这种技术的含义。如图磁盘阵列控制器，联接了四个磁盘，这四个磁盘形成一个阵列（array），而磁盘阵列的控制器（RAID controller）是将此四个磁盘视为单一的磁盘，如 DOS 环境下的 C：盘。这是 disk spanning 的意义，因为把小容量的磁盘延伸为大容量的单一磁盘，用户不必规划数据在各磁盘的分布，而且提高了磁盘空间的使用率。并使磁盘容量几乎可作无限的延伸；而各个磁盘一起作取存的动作，比单一磁盘更为快捷。很明显的，有此阵列的形成而产生 RAID 的各种技术。

磁盘或数据分段(Disk Striping or Data Striping)：

因为磁盘阵列是将同一阵列的多个磁盘视为单一的虚拟磁盘（virtual disk），所以其数据是以分段（block or segment）的方式顺序存放在磁盘阵列中，数据按需要分段，从第一个磁盘开始放，放到最後一个磁盘再回到第一个磁盘放起，直到数据分布完毕。至于分段的大小视系统而定，有的系统或以 1KB 最有效率，或以 4KB，或以 6KB，甚至是 4MB 或 8MB 的，但除非数据小于一个扇区（sector,即 521bytes），否则其分段应是 512byte 的倍数。因为磁盘的读写是以一个扇区为单位，若数据小于 512bytes，系统读取该扇区后，还要做组合或分组（视读或写而定）的动作，浪费时间。从上图我们可以看出，数据以分段于在不同的磁盘，整个阵列的各个磁盘可同时作读写，故数据分段使数据的存取有最好的效率，理论上本来读一个包含四个分段的数据所需要的时间约=（磁盘的 access time+数据的 tranfer time）X4 次，现在只要一次就可以完成。

若以 N 表示磁盘的数目，R 表示读取，W 表示写入，S 表示可使用空间，则数据分段的性能为：
R：N（可同时读取所有磁盘）
W：N（可同时写入所有磁盘）
S：N（可利用所有的磁盘，并有最佳的使用率）

Disk striping 也称为 RAID 0，很多人以为 RAID 0 没有甚么，其实这是非常错误的观念，因为 RAID 0 使磁盘的输出入有最高的效率。而磁盘阵列有更好效率的原因除数据分段

外，它可以同时执行多个输出入的要求，因为阵列中的每一个磁盘都能独立动作，分段放在不同的磁盘，不同的磁盘可同时作读写，而且能在快取内存及磁盘作并行存取（parallel access）的动作，但只有硬件的磁盘阵列才有此性能表现。

从上面两点我们可以看出，disk spanning 定义了 RAID 的基本形式，提供了一个便宜、灵活、高性能的系统结构，而 disk striping 解决了数据的存取效率和磁盘的利用率问题，RAID 1 至 RAID 5 是在此基础上提供磁盘安全的方案。

RAID 1

RAID 1 是使用磁盘镜像（disk mirroring）的技术。磁盘镜像应用在 RAID 1 之前就在很多系统中使用，它的方式是在工作磁盘（working disk）之外再加一额外的备份磁盘（backup disk），两个磁盘所储存的数据完全一样，数据写入工作磁盘的同时亦写入备份磁盘。磁盘镜像不见得就是 RAID 1，如 Novell Netware 亦有提供磁盘镜像的功能，但并不表示 Netware 有了 RAID 1 的功能。一般磁盘镜像和 RAID 1 有二点最大的不同：

RAID 1 无工作磁盘和备份磁盘之分，多个磁盘可同时动作而有重叠（overlaping）读取的功能，甚至不同的镜像磁盘可同时作写入的动作，这是一种最佳化的方式，称为负载平衡（load-balance）。例如有多个用户在同一时间要读取数据，系统能同时驱动互相镜像的磁盘，同时读取数据，以减轻系统的负载，增加 I/O 的性能。

RAID 1 的磁盘是以磁盘延伸的方式形成阵列，而数据是以数据分段的方式作储存，因而在读取时,它几乎和 RAID 0 有同样的性能。从 RAID 的结构就可以很清楚的看出 RAID 1 和一般磁盘镜像的不同。
下图为 RAID 1,每一笔数据都储存两份：
从图可以看出：
R：N（可同时读取所有磁盘）
W：N/2（同时写入磁盘数）
S：N/2（利用率）

读取数据时可用到所有的磁盘，充分发挥数据分段的优点；写入数据时，因为有备份，所以要写入两个磁盘，其效率是 N/2，磁盘空间的使用率也只有全部磁盘的一半。

很多人以为 RAID 1 要加一个额外的磁盘，形成浪费而不看好 RAID 1，事实上磁盘越来越便宜，并不见得造成负担，况且 RAID 1 有最好的容错（fault tolerence）能力，其效率也是除 RAID 0 之外最好的。

在磁盘阵列的技术上，从 RAID 1 到 RAID 5，不停机的意思表示在工作时如发生磁盘故障， 系统能持续工作而不停顿，仍然可作磁盘的存取，正常的读写数据；而容错则表示即使磁盘故障，数据仍能保持完整，可让系统存取到正确的数据，而 SCSI 的磁盘阵列更可在工作中抽换磁盘，并可自动重建故障磁盘的数据。磁盘阵列之所以能做到容错及不停机， 是因为它有冗余的磁盘空间可资利用，这也就是 Redundant 的意义。 RAID 2

RAID 2 是把数据分散为位（bit）或块（block），加入海明码 Hamming Code，在磁盘阵列中作间隔写入（interleaving）到每个磁盘中，而且地址（address）都一样，也就是在各个磁盘中，其数据都在相同的磁道（cylinder or track）及扇区中。RAID 2 的设计是使用共轴同步（spindle synchronize）的技术，存取数据时，整个磁盘阵列一起动作，在各作磁盘的相同位置作平行存取，所以有最好的存取时间（accesstime），其总线（bus）是特别的设计，以大带宽（band wide）并行传输所存取的数据，所以有最好的传输时间（transfer time）。在大型档案的存取应用，RAID 2 有最好的性能，但如果档案太小，会将其性能拉下来，因为磁盘的存取是以扇区为单位，而 RAID 2 的存取是所有磁盘平行动作，而且是作单位元的存取，故小于一个扇区的数据量会使其性能大打折扣。RAID 2 是设计给需要连续且大量数据的电脑使用的，如大型电脑（mainframe to supercomputer）、作影像处理或 CAD/CAM 的工作站（workstation）等，并不适用于一般的多用户环境、网络服务器（network server），小型机或 PC。

RAID 2 的安全采用内存阵列（memory array）的技术，使用多个额外的磁盘作单位错误校正（single-bit correction）及双位错误检测（double-bit detection）；至于需要多少个额外的磁盘，则视其所采用的方法及结构而定，例如八个数据磁盘的阵列可能需要三个额外的磁盘，有三十二个数据磁盘的高档阵列可能需要七个额外的磁盘。

RAID 3

RAID 3 的数据储存及存取方式都和 RAID 2 一样，但在安全方面以奇偶校验（parity check）取代海明码做错误校正及检测，所以只需要一个额外的校检磁盘（parity disk）。奇偶校验值的计算是以各个磁盘的相对应位作 XOR 的逻辑运算，然后将结果写入奇偶校验磁盘，任何数据的修改都要做奇偶校验计算，

如某一磁盘故障，换上新的磁盘后，整个磁盘阵列（包括奇偶校验磁盘）需重新计算一次， 将故障磁盘的数据恢复并写入新磁盘中；如奇偶校验磁盘故障，则重新计算奇偶校验值，以达容错的要求。

较之 RAID 1 及 RAID 2，RAID 3 有 85%的磁盘空间利用率，其性能比 RAID 2 稍差，因为要做奇偶校验计算；共轴同步的平行存取在读档案时有很好的性能，但在写入时较慢，需要重新计算及修改奇偶校验磁盘的内容。RAID 3 和 RAID 2 有同样的应用方式，适用大档案及大量数据输出入的应用，并不适用于 PC 及网络服务器。 RAID 4

RAID 4 也使用一个校验磁盘，但和 RAID 3 不一样
RAID 4 是以扇区作数据分段，各磁盘相同位置的分段形成一个校验磁盘分段（parity block），放在校验磁盘。这种方式可在不同的磁盘平行执行不同的读取命今，大幅提高磁盘阵列的读取性能；但写入数据时，因受限于校验磁盘，同一时间只能作一次，启动所有磁盘读取数据形成同一校验分段的所有数据分段，与要写入的数据做好校验计算再写入。即使如此，小型档案的写入仍然比 RAID 3 要快，因其校验计算较简单而非作位（bit level）的计算；但校验磁盘形成 RAID 4 的瓶颈，降低了性能，因有 RAID 5 而使得 RAID 4 较少使用。
RAID 5

RAID5 避免了 RAID 4 的瓶颈，方法是不用校验磁盘而将校验数据以循环的方式放在每一个磁盘中，磁盘阵列的第一个磁盘分段是校验值，第二个磁盘至后一个磁盘再折回第一个磁盘的分段是数据，然后第二个磁盘的分段是校验值，从第三个磁盘再折回第二个磁盘的分段是数据，以此类推，直到放完为止。图中的第一个 parity block 是由 A0，A1...，B1，B2 计算出来，第二个 parity block 是由 B3，B4，...，C4，D0 计算出来，也就是校验值是由各磁盘
同一位置的分段的数据所计算出来。这种方式能大幅增加小档案的存取性能，不但可同时读取，甚至有可能同时执行多个写入的动作，如可写入数据到磁盘 1 而其 parity block 在磁盘 2，同时写入数据到磁盘 4 而其 parity block 在磁盘 1，这对联机交易处理（OLTP，On-Line Transaction Processing）如银行系统、金融、股市等或大型数据库的 处理提供了最佳的解决方案（solution），因为这些应用的每一笔数据量小，磁盘输出入频繁而且必须容错。

事实上 RAID 5 的性能并无如此理想，因为任何数据的修改，都要把同一 parityblock 的所有数据读出来修改后，做完校验计算再写回去，也就是 RMW cycle（Read-Modify-Write cycle，这个 cycle 没有包括校验计算）；正因为牵一而动全身，所以：
R：N（可同时读取所有磁盘）
W：1（可同时写入磁盘数）
S：N-1（利用率）

RAID 5 的控制比较复杂，尤其是利用硬件对磁盘阵列的控制，因为这种方式的应用比其他的 RAID level 要掌握更多的事情，有更多的输出入需求，既要速度快，又要处理数据，计算校验值，做错误校正等，所以价格较高；其应用最好是 OLTP，至于用于图像处理等，不见得有最佳的性能。

2、磁盘阵列的额外容错功能：Spare or Standby driver

事实上容错功能已成为磁盘阵列最受青睐的特性，为了加强容错的功能以及使系统在磁盘故障的情况下能迅速的重建数据，以维持系统的性能，一般的磁盘阵列系统都可使用热备份（hot spare or hot standby driver）的功能，所谓热备份是在建立（configure）磁盘阵列系统的时候，将其中一磁盘指定为后备磁盘，此一磁盘在平常并不操作，但若阵列中某一磁盘发生故障时，磁盘阵列即以后备磁盘取代故障磁盘，并自动将故障磁盘的数据重建（rebuild）在后备磁盘之上，因为反应快速，加上快取内存减少了磁盘的存取， 所以数据重建很快即可完成，对系统的性能影响很小。对于要求不停机的大型数据处理中心或控制中心而言，热备份更是一项重要的功能，因为可避免晚间或无人值守时发生磁盘故障所引起的种种不便。
另一个额外的容错功能是坏扇区转移（bad sector reassignment）。坏扇区是磁盘故障的主要原因，通常磁盘在读写时发生坏扇区的情况即表示此磁盘故障，不能再作读写，甚至有很多系统会因为不能完成读写的动作而死机，但若因为某一扇区的损坏而使工作不能完成或要更换磁盘，则使得系统性能大打折扣，而系统的维护成本也未免太高了。坏扇区转移是当磁盘阵列系统发现磁盘有坏扇区时，以另一空白且无故障的扇区取代该扇区， 以延长磁盘的使用寿命，减少坏磁盘的发生率以及系统的维护成本。所以坏扇区转移功能使磁盘阵列具有更好的容错性，同时使整个系统有最好的成本效益比。其他如可外接电池备援磁盘阵列的快取内存，以避免突然断电时数据尚未写回磁盘而损失；

或在 RAID 1 时作写入一致性的检查等，虽是小技术，但亦不可忽视。

3、硬件磁盘阵列还是软件磁盘阵列

市面上有所谓硬件磁盘阵列与软件磁盘阵列之分，因为软件磁盘阵列是使用一块 SCSI 卡与磁盘连接，一般用户误以为是硬件磁盘阵列。以上所述主要是针对硬件磁盘阵列，其与软件磁盘阵列有几个最大的区别：
一个完整的磁盘阵列硬件与系统相接。
内置 CPU，与主机并行运作，所有的 I/O 都在磁盘阵列中完成，减轻主机的工作负载，增加系统整体性能。
有卓越的总线主控（bus mastering）及 DMA（Direct Memory Access）能力，加速数据的存取及传输性能。
与快取内存结合在一起，不但增加数据的存取及传输性能，更因减少对磁盘的存取而增加磁盘的寿命。
能充份利用硬件的特性，反应快速。
软件磁盘阵列是一个程序，在主机执行，透过一块 SCSI 卡与磁盘相接形成阵列，它最大的优点是便宜，因为没有硬件成本（包括研发、生产、维护等），而 SCSI 卡很便宜（亦有的软件磁盘阵列使用指定的很贵的 SCSI 卡）；它最大的缺点是使主机多了很多进程（process），增加了主机的负担，尤其是输出入需求量大的系统。目前市面上的磁盘阵列
系统大部份是硬件磁盘阵列，软件磁盘阵列较少。

4、磁盘阵列卡还是磁盘阵列控制器

磁盘阵列控制卡一般用于小系统，供单机使用。与主机共用电源，在关闭主机电源时存在丢失 Cache 中的数据的的危险。磁盘阵列控制卡只有常用总线方式的接口，其驱动程序与主机、主机所用的操作系统都有关系，有软、硬件兼容性问题并潜在地增加了系统的不安定因素。在更换磁盘阵列卡时要冒磁盘损坏，资料失落，随时停机的风险。

独立式磁盘阵列控制一般用于较大型系统，可分为两种：

单通道磁盘阵列和多通道式磁盘阵列，单通道磁盘阵列只能接一台主机，有很大的扩充限制。多通道磁盘阵列可接多个系统同时使用，以群集（cluster）的方式共用磁盘阵列，这使内接式阵列控制及单接式磁盘阵列无用武之地。目前多数独立形式的磁盘阵列子系统，其本身与主机系统的硬件及操作环境。

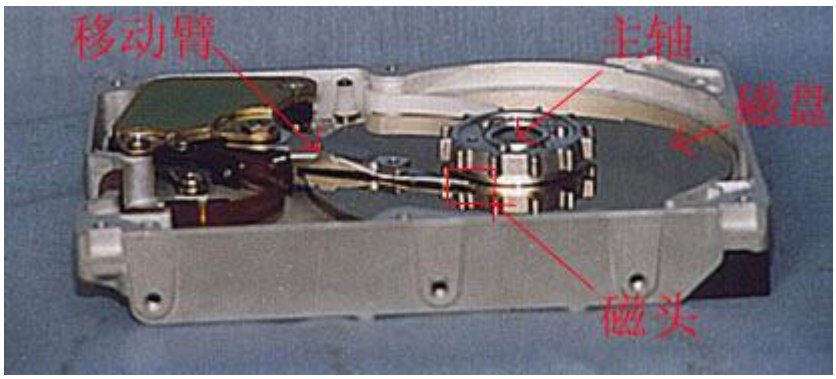
首先，IDE 的性能不会比 SCSI 更高的。特别是在多任务的情况下。一般广告给出的是 最大传送速度，并不是工作速度。同一时期的 IDE 与 SCSI 盘相比，主要是产量比较大， 电路比较简单，所以价格比 SCSI 低很多，但要比性能，则差远了。
RAID 并没有限制使用多少个盘，盘越多越好。
对于 SCSI 结构的 RAID 来说，盘的最大数量与 SCSI 通道（SCSI 总线）的数量有关一般是每个通道最多装 15 个盘（SCSI/3）对于 FC-AL（光纤）则是每个通道 200 个盘当然，要有这样大的磁盘箱才行！

IDE 硬盘详解

要说去年的计算机业，发展最快的是什么？我想，硬盘一定算一号！

记得去年年初,DMA66 刚刚兴起，主流配置还是 13.6G。时隔一年，DMA66 技术已经普及，而且正向这 DMA100 迈进。容量也是翻了一翻还要多，普及配置达到了 30G。其更新换代的速度甚至超过了大名鼎鼎的"摩尔定律"（即集成电路中的晶体管数量每隔 18 个月增加一倍）。然而，面对这突如其来的"硬盘风"不仅普通用户，就连一些 DIYer 也有些招架不住。所以，小弟特写了这篇文章，详细的介绍了当今的硬盘技术，让大家对硬盘有一个系统的理解！

要了解硬盘，就一定要清楚硬盘的工作原理。首先，硬盘主要是由磁盘、移动臂、主轴、磁头和主轴电机组成（见图）。所有的数据都存储在磁盘上，磁盘又固定在主轴上，一般一块硬盘由 1-5 张磁盘组成。主轴底部有一个电机，当硬盘运做时，电机带动主轴，主轴带动磁盘高速旋转，其速度可以达到每分钟几千转、甚至上万转。这时旋转带来的上升的空气将磁盘上的磁头托起，磁头通过磁盘的转动读取数据。移动臂用来固定磁头，让磁头能在磁盘上不同磁道之间来回移动，读取数据。以上，就是硬盘的基本工作原理。



现在大家应该对硬盘有一定了解了，不过这还不够。比如，大家在购买硬盘时经常会碰到 UDMA 、2M 缓存、单碟容量、7200 转等专业名词，这些都是什么意思呢？别着急，下面就为大家解释。

首先要说的就是 UDMA。不过，在这之前我们得先了解一下，什么是 DMA？我们都知道，计算机要工作，都要由 CPU 发出指令，各个部件才做出响应，硬盘也是如此。假如现在 CPU 发出指令，要从硬盘上读取 512K 的数据，设 CPU 每条指令每次能从硬盘读取 1K 的数据，那 CPU 要对硬盘发出 512 次指令。而且硬盘的速度又远远低于 CPU，这就造成 CPU 将长时间等待硬盘的数据，这无疑大大浪费了 CPU。要怎么解决这个问题呢？DMA 技术就应蕴而生了。它的工作原理很简单，就是在主板的南桥芯片中增加了一个 DMA 控制器。DMA 控制器起什么作用呢？同样是上边的例子，当 CPU 要从硬盘上读取的数据时，CPU 只要发出一条指令，告诉 DMA 控制器要读取那一块的数据，由 DMA 控制器来从硬盘上读取数据，读取的数据暂时存放在硬盘的缓存（Cache）上，当数据全部读到缓存上时，DMA 控制器会向 CPU 发出一条回馈信息，告诉 CPU 数据以读完。这

时 CPU 再发第二条条读指令，将缓存上的数据读到内存中。这样 CPU 只发出了两条指令，就完成了这 512K 数据的读取。举个不是很恰当的例子：有一个公司，公司最高领导是总经理，公司中所有的事都由总经理来管，但是像打字、发信、复印等这些琐碎工作，却不用都要总经理来做。这就需要为总经理请个秘书，那些琐碎的小事都交给秘书去做，解放总经理，让总经理有更多的时间做重要的工作。在计算机中也是如此，CPU 就相当于总经理，DMA 控制器就相当于总经理的秘书，DMA 解放了 CPU，减轻了 CPU 的负担，让 CPU 做更重要的工作。所以要清楚一点，DMA 并没有提高硬盘速度，但 DMA 可以大大减少 CPU 占有率，从而提高计算机的整体性能。这才是 DMA 的真正作用！

随着技术的发展，计算机的速度越来越快。可是，硬盘的速度却没什么提高，这时硬盘就成为瓶颈。为了解决这个问题，UDMA 就诞生了。UDMA 的全名叫 Ultra DMA，它是一种接口技术，就是说 UDMA 只能提高硬盘的外部传输速率，而改变不了硬盘的内部传输速率，这点在后边将会进一步说明。最初我们使用的都是 UDMA 33，它是利用脉冲的上沿和下沿传送数据，突发性传输速率达到了 33MB/s。在原来的基础上加入了循环校验（CRC），提高了传输数据的完整性。但是最重要的还是 UDMA 33 完全向下兼容，这对 UDMA 的推广起了关键的作用。不过，随着硬盘技术的不断提高，渐渐的硬盘内部传输速率接近并超过了 33MB/s。这时昆腾和 INTEL 公司在 1998 年又联合推出了 UDMA 66，让突发性传输速率达到了 66MB/s。UDMA 66 的最大特点，就是在原来 40 芯电缆的基础上又增加了 40 根地线电缆，使电缆数达到 80 根，这样做一来增加了一倍传输速率，二来也提高了数据传输的可靠性，保证了数据的完整性。UDMA 66 也向下兼容，虽然采用 80 芯电缆，但接口插针还是 40 针，只是在连接线内部增加了 40 条地线。在这 80 根电缆中第 34 根电缆是断开的，而普通的 40 芯电缆这条电缆是连通的，这有效的区分了 UDMA 66 和 UDMA 33，在检测到这条电缆是否连通后，BIOS 会自动判断是 UDMA 33 还是 UDMA 66。

现在 PC 硬盘内部传输速率最快可以达到 56MB/s，在这个基础上 UDMA 66 基本上是够用了。但技术是发展的，新一代的硬盘将突破 66MB/s 这个上限，于是昆腾在 2001 年 6 月有发表了 UDMA 100 接口标准，这显然是一个面向明天的接口标准，虽然对现在的硬盘没什么作用，但是还是有必要要介绍一下。UDMA 100 仍然使用 80 芯 40 针的数据线，所不同的是 UDMA 100 将突发性传输速率提高到了 100MB/s，同 UDMA 66 一样也向下兼容，而且在兼容性上也做了进一步的改善。

提高了外部传输速率，那内部传输速率又如何提高呢？有两种方法，第一种、是增加磁盘转速，比如从 5400 转提高到现在的 7200 转。第二种、是提高磁盘的单碟容量。那这两种方法都是如何的实现呢？下面就为大家介绍。

在购买硬盘时，经常听说 7200 转和 5400 转。我们说过硬盘工作是靠主轴电机带动磁盘转动，这个速度就是磁盘转动的速度。7200 转和 5400 转的意思就是磁盘在每分钟可以转动 7200 圈和 5400 圈。相对来说转速越快，磁盘的内部传输速率就越快，因为相同时间内磁盘转动越快，磁头经过的磁道就越多，读取的数据就越多。不过，这也不是无止境的，首先、更快的速度就意味着需要更敏感的磁头，其次、提高转速还会带来发热量和噪音。现在硬盘已经可以达到 10000 转，可是由于发热量和噪音是无法忍受的（至少对 PC 来说），现在的技术还没法控制。所以，10000 转硬盘还只能应用在高端服务器的 SCSI 硬盘上。就现在来说，7200 转的硬盘技术已经比较成熟，发热量和噪音都控制

的很好，是现在的首选。5400 转的硬盘虽然也不错，不过由于先天不足只好成为低端产品。

相对于提高转速来说，提高单碟容量的作用相对更突出些。增加单碟容量和提高转速的原理差不多。硬盘的单碟容量增加了，磁盘上单位面积上存储的数据也就增多。也就是说，在相同的时间里，单碟容量越高，磁头读取的数据就越多。这就可以解释为什么在有些评测中，有些 5400 转的硬盘会比一些 7200 转的还快，原因就在于单碟容量。另外，提高单碟容量还有一个重要的作用，那就是可以有效的提高硬盘的容量。就是说在单碟容量越大，磁盘上存储数据就越多，前边说过硬盘是由磁盘组成，因此单碟容量的增加，可以使硬盘容量增加。打个比方吧！同样是迈拓硬盘，迈拓钻石九代单碟容量 10.2G，需要 3 张磁盘组成 30G 硬盘，而迈拓钻石十代单碟容量 15.3G，只用 2 张磁盘可以组成 30G 硬盘。不仅速度上有所提升，而且少用了一张磁盘，成本也会降低，那样价格也就会更低。这也是为什么各大硬盘厂商极力宣传单碟容量的原因。

硬盘还有另一个重要技术，那就是缓存。想必大家一定还记得在前边介绍 DMA 时，曾经介绍过缓存（Cache）的作用。它硬盘中可以说是举足轻重，自然也是硬盘厂商宣传的重点，同样也是我们挑选硬盘的重点。更大的缓存会带来硬盘性能的提升，但更大的缓存就意味着成本的增加，价格的增加。还有更大的缓存也需要好的算法的支持，如果缓存调度算法落后，大缓存不但发挥不出其真正实力，甚至还会影响到整个硬盘。现在硬盘缓存容量以 512K 和 2M 为多，512K 的缓存世面上还会见到，不过都是些低端产品，不值得购买。就现在来说，2M 缓存是一个比较理想的选择，主流硬盘的都是 2M 缓存，所以在购买硬盘还是尽量挑选 2M 缓存的硬盘。

现在的硬盘越出越大，硬盘上存储的数据越来越重要，这样保护好硬盘非常必要了。我们都知道，硬盘的磁头距离硬盘很近（即使在运行的时候也只有 0.01 微米），如果硬盘遇到震动，即使很小的震动，那么磁头撞击磁盘，使磁盘造成损伤，使这块区域就不能使用。但是更糟糕的是，这块损伤会继续扩散，坏磁道将越来越多，那这快硬盘可就快寿终正寝了！为此硬盘厂商想出了多种办法来保护硬盘，比如迈拓公司的 Shock Block 技术，就是把移动臂强度提高 25%，并且把磁头的重量减少 40%。还有，增加硬盘主机座的重量，让硬盘更沉稳。和使用强度更高的合金材料做外壳，一来可以抵受碰撞，二来还可以减少硬盘自身的震动，降低噪音。几乎所有新推出的硬盘都进行了这样的设计，而且也很见效，只是硬盘厂商很少宣传。不过，这些技术并不是万能的，我们还是应该以预防为主！

NTFS 是 Windows NT 引入的新型文件系统，它具有许多新特性。本文旨在探索 NTFS 的底层结构，所叙述的也仅是文件在 NTFS 卷上的分布。NTFS 中，卷中所有存放的数据均在一个叫$MFT 的文件中，叫主文件表(Master File Table)。而$MFT 则由文件记录(File Record)数组构成。File Record 的大小一般是固定的，通常情况下均为 1KB，这个概念相当于 Linux 中的 inode。File Record 在$MFT 文件中物理上是连续的，且从 0 开始编号。$MFT 仅供 File System 本身组织、架构文件系统使用，这在 NTFS 中称为元数据(Metadata)。以下列出 Windows 2000 Release 出的 NTFS 的元数据文件(我将要给出的示例代码的部分输出结果)。

File Record(inode) FileName

---

0 $MFT

1 $MFTMirr

2 $LogFile

3 $Volume

4 $AttrDef

5 .

6 $Bitmap

7 $Boot

8 $BadClus

9 $Secure

10 $UpCase

11 $Extend

Windows??2000 中不能使用 dir 命令(甚至加上/ah 参数)像普通文件一样列出这些元数据文件。实际上 File System Driver(ntfs.sys)维护了一个系统变量 NtfsProtectSystemFiles 用于隐藏这些元数据。默认情况下，这个变量被设为 TRUE，所以使用 dir /ah 将得不到任何文件。知道这个行为后使用 i386kd 修改 NtfsProtectSystemFiles 后即可以列出元数据文件：

kd> x ntfs!NtfsProtect*

fe213498 Ntfs!NtfsProtectSystemFiles

fe21349c Ntfs!NtfsProtectSystemAttributes

kd> dd ntfs!NtfsProtectSystemFiles l 2

fe213498 00000001 00000001

```
kd> ed ntfs!NtfsProtectSystemFiles 0

kd> dd ntfs!NtfsProtectSystemFiles l 2

fe213498 00000000 00000001

kd>

D:\>ver

Microsoft?Windows?2000?[Version 5.00.2195]

D:\>dir /ah $*

驱动器 D 中的卷是 W2KNTFS

卷的序列号是 E831-9D04

D:\ 的目录

2000-04-27 19:31 36,000 $AttrDef

2000-04-27 19:31 0 $BadClus

2000-04-27 19:31 67,336 $Bitmap

2000-04-27 19:31 8,192 $Boot

2000-04-27 19:31 <DIR> $Extend

2000-04-27 19:31 13,139,968 $LogFile

2000-04-27 19:31 27,575,296 $MFT

2000-04-27 19:31 4,096 $MFTMirr

2000-04-27 19:31 131,072 $UpCase

2000-04-27 19:31 0 $Volume

9 个文件 40,961,960 字节

1 个目录 51,863,552 可用字节
```

需要指出的是 ntfs.sys 将元数据文件以一种特殊的方式打开，所以在打开 NtfsProtectSystemFiles 后，如果使用 ReadFile 等产生 IRP_MJ_READ 等 IRP 包时将会导致 Page Fault(详见 Gary Nebbett 的《Windows NT/2000 Native API Reference》)。

以上的讨论均是基于$MFT 文件而讨论的，即基于$MFT 中的 File Record(inode)讨论的。为更好的继续以下的讨论，这儿我列出 File Record Header 的结构：

```
typedef struct {

ULONG Type;

USHORT UsaOffset;

USHORT UsaCount;

USN Usn;

} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;

typedef struct {

NTFS_RECORD_HEADER Ntfs;

USHORT SequenceNumber;

USHORT LinkCount;

USHORT AttributesOffset;

USHORT Flags; // 0x0001 = InUse, 0x0002 = Directory

ULONG BytesInUse;

ULONG BytesAllocated;

ULONGLONG BaseFileRecord;

USHORT NextAttributeNumber;

} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;
```

下面我将讨论如何定位$MFT。稍微有点操作系统知识的人都会知道引导扇区(Boot Sector)，其物理位置为卷中的第一个扇区。以下由 dskprobe.exe(Windows 2000 Resource Kit 中的一个小工具)分析的第一个扇区(当然也可以使用 WinHex 等其他应用

程序):

file: d:\Sector00.bin

Size: 0x00000200 (512)

Address | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF

---------|-----------------------:-----------------------|----------------

00000000 | EB 52 90 4E-54 46 53 20 : 20 20 20 00-02 08 00 00 | ?R?NTFS .....

00000010 | 00 00 00 00-00 F8 00 00 : 3F 00 F0 00-3F 00 00 00 | .....?..?.e.?...

00000020 | 00 00 00 00-80 00 80 00 : 90 C0 41 00-00 00 00 00 | ......惱 A.....

00000030 | 04 00 00 00-00 00 00 00 : 09 1C 04 00-00 00 00 00 | ....

00000040 | F6 00 00 00-01 00 00 00 : 04 9D 31 E8-BB 31 E8 94 | ?.......?杌 1 鑞

. .

. .

. .

000001F0 | 00 00 00 00-00 00 00 00 : 83 A0 B3 C9-00 00 55 AA | ....儭成..U?

这 512 字节为如下的格式:(摘自 Gary Nebbett 书中,本文许多代码均来自或参考此书。)

```
#pragma pack(push, 1)
typedef struct {
UCHAR Jump[3];
UCHAR Format[8];
USHORT BytesPerSector;
UCHAR SectorsPerCluster;
USHORT BootSectors;
```

```
UCHAR Mbz1;

USHORT Mbz2;

USHORT Reserved1;

UCHAR MediaType;

USHORT Mbz3;

USHORT SectorsPerTrack;

USHORT NumberOfHeads;

ULONG PartitionOffset;

ULONG Reserved2[2];

ULONGLONG TotalSectors;

ULONGLONG MftStartLcn;

ULONGLONG Mft2StartLcn;

ULONG ClustersPerFileRecord;

ULONG ClustersPerIndexBlock;

ULONGLONG VolumeSerialNumber;

UCHAR Code[0x1AE];

USHORT BootSignature;

} BOOT_BLOCK, *PBOOT_BLOCK;

#pragma pack(pop)
```

各个字段的详细意义从字段名中即可大致清楚。在 linux-ntfs 的 GNU 工程(http://sf.net/projects/linux-ntfs)中也有详细的文档，限于篇幅我不将其列出。可以使用如下代码读出卷中的第一个扇区：

```
hVolume = CreateFile(drive, GENERIC_READ, FILE_SHARE_READ |
FILE_SHARE_WRITE, 0,
```

OPEN_EXISTING, 0, 0);

ReadFile(hVolume, &bootb, sizeof(bootb), &n, 0);

bootb 是一个 BOOT_BLOCK 结构，在我的卷中如下格式(请对应 Sector00.bin 分析)：

Dump BootBlock at below:

BytesPerSector:200

SectorsPerCluster:8

BootSectors:0

SectorsPerTrack:3F

NumberOfHeads:F0

PartitionOffset:3F

TotalSectors:41C090

MftStartLcn:4

Mft2StartLcn:41C09

ClustersPerFileRecord:F6

ClustersPerIndexBlock:1

VolumeSerialNumber:E8319D04

BootSignature:AA55

以上的 MftStartLcn 其实是$MFT 在卷中的簇(Cluster)号。簇是 NTFS 的基本单位，最小单位。一个只有 1Byte 的文件也要占用一簇的空间。NTFS 使用 LCN(Logical Cluster Number)来代表 NTFS 卷中的物理位置，其简单的从 0 到卷中的总簇数减一进行编号。对于一个特定的文件 NTFS 则使用 VCN(Virtual Cluster Number)来映射 LCN 实现文件的组织。从 MftStartLcn 的值 4 可以知道$MFT 的 LCN 为 4 与 SectorsPerCluster、BytesPerSector 的大小即可定位$MFT 的位置。得到$MFT 的位置后，如果遍历$MFT 中所有的 File Record 即可以得到卷中所有的文件列表(前面已经提到 File Record 只是简单的从 0 开始编号)。也就是说到目前为止已经可以对文件组织有最简单的认识，但如何得到文件的信息呢，如文件名等等。NTFS 中所有文件包括普通的用户文件、元数据文

件均用同样的方式组织数据、属性等。我将 nfi.exe(来自 Windows NT/2000 OEM Support Tools)的输出结果列出，作为我叙述的开始：

D:\>copy con file

testforntfs^Z

已复制 1 个文件。

D:\>nfi d:\file

NTFS File Sector Information Utility.

Copyright © Microsoft Corporation 1999. All rights reserved.

\file

$STANDARD_INFORMATION (resident)

$FILE_NAME (resident)

$DATA (resident)

D:\>echo testforattr>file:ATTR

D:\>nfi d:\file

NTFS File Sector Information Utility.

Copyright © Microsoft Corporation 1999. All rights reserved.

\file

$STANDARD_INFORMATION (resident)

$FILE_NAME (resident)

$DATA (resident)

$DATA ATTR (resident)

nfi 的输出结果$STANDARD_INFORMATION、$FILE_NAME、$DATA 等在 NTFS 中称为属性(Attribute)。属性分为常驻属性(Resident Attribute)与非常驻属性(Nonresident Attribute)。文件的数据也包含在属性中，似乎与属性这个名称有点混谣。不过这又让

NTFS 有了更加统一的组织文件的形式。这也同时让 NTFS 有 MultiStreams 的特性(上面也演示了这个特性)。通过指定的 File Record 定位给定的 Attribute 的实现代码如下：

```
template <class T1, class T2> inline

T1* Padd(T1* p, T2 n) { return (T1*)((char *)p + n); }

PATTRIBUTE FindAttribute(PFILE_RECORD_HEADER file,

ATTRIBUTE_TYPE type, PWSTR name)

{

for (PATTRIBUTE attr = PATTRIBUTE(Padd(file, file->AttributesOffset));

attr->AttributeType != -1;

attr = Padd(attr, attr->Length)) {

if (attr->AttributeType == type) {

if (name == 0 && attr->NameLength == 0) return attr;

if (name != 0 && wcslen(name) == attr->NameLength

&& _wcsicmp(name, PWSTR(Padd(attr, attr->NameOffset))) == 0) return attr;

}

}

return 0;

}
```

Gary Nebbett 提供的这个 FindAttribute 函数在 Attribute name(即第三个参数)不为空串时可能会出现 bug，主要原因是_wcsicmp 对 UNICODE 字符串比较时应该是以\0 结束的标准的 C 字符串。我在提供的代码中已经纠正了这个错误。

下面我将通过使用 SoftICE 来分析这段代码得到$MFT 的$FILE_NAME 属性来得到$MFT 的 file name。这个示例同样适用于得到其它文件的$FILE_NAME(如上面的 file)、还有其它的属性如$DATA 等等。

```
:bpx FindAttribute
```

Break due to BPX FindAttribute (ET=6.89 seconds)

:locals

[EBP-4] +struct ATTRIBUTE * attr = 0x00344D68 <{...}>

[EBP+8] +struct FILE_RECORD_HEADER * file = 0x00344D38 <{...}>

[EBP+C] enum ATTRIBUTE_TYPE type = AttributeFileName (30)

[EBP+10] +unsigned short * name = 0x004041BC <"$MFT">

:?file

struct FILE_RECORD_HEADER * = 0x00344D38 <{...}>

struct NTFS_RECORD_HEADER Ntfs = {...}

unsigned short SequenceNumber = 0x1, "\0\x01"

unsigned short LinkCount = 0x1, "\0\x01"

unsigned short AttributesOffset = 0x30, "\00"

unsigned short Flags = 0x1, "\0\x01"

unsigned long BytesInUse = 0x2D8, "\0\0\x02\xD8"

unsigned long BytesAllocated = 0x400, "\0\0\x04\0"

unsigned quad BaseFileRecord = 0x0, "\0\0\0\0\0\0\0\0"

unsigned short NextAttributeNumber = 0x6, "\0\x06"

file 参数我传入的是$MFT，从$MFT 的 LCN=4 可以得到其在卷中的物理地址，这在上面已说明。你也可以使用 dskprobe(我机子中为第 LCN*SectorsPerCluster=4*8 扇区)得到底下 SoftICE 的输出结果：

d @file //以下的注释可对照文中开头列出的 FILE_RECORD_HEADER 定义。

0023:00344D38 454C4946 0003002A 6D4AC04D 00000000 FILE*...M.Jm....

0023:00344D48 00010001 00010030 000002D8 00000400 ....0.....

|__AttributeOffset

0023:00344D58 00000000 00000000 04340006 0000FA0D .....4.....

0023:00344D68 00000010 00000060 00180000 00000000 ....`.....

-------- --------

| |_指出这个 Attribute 的长度。定义如下。

|_根据 AttributeOffset 得到的 Attribute 头,定义如下。00000010 指出这个 Attribute 为 StandardInformation

0023:00344D78 00000048 00000018 2C1761D0 01BFB03C H.....a.,<...

Attribute 头如下定义：

typedef struct {

ATTRIBUTE_TYPE AttributeType;

ULONG Length;

BOOLEAN Nonresident;

UCHAR NameLength;

USHORT NameOffset;

USHORT Flags; // 0x0001 = Compressed

USHORT AttributeNumber;

} ATTRIBUTE, *PATTRIBUTE;

typedef struct {

ATTRIBUTE Attribute;

ULONG ValueLength;

USHORT ValueOffset;

USHORT Flags; // 0x0001 = Indexed

} RESIDENT_ATTRIBUTE, *PRESIDENT_ATTRIBUTE;

typedef struct {

ULONGLONG DirectoryFileReferenceNumber;

ULONGLONG CreationTime; // Saved when filename last changed

ULONGLONG ChangeTime; // ditto

ULONGLONG LastWriteTime; // ditto

ULONGLONG LastAccessTime; // ditto

ULONGLONG AllocatedSize; // ditto

ULONGLONG DataSize; // ditto

ULONG FileAttributes; // ditto

ULONG AlignmentOrReserved;

UCHAR NameLength;

UCHAR NameType; // 0x01 = Long, 0x02 = Short

WCHAR Name[1];

} FILENAME_ATTRIBUTE, *PFILENAME_ATTRIBUTE;

ATTRIBUTE_TYPE 是一个 Enum 型定义。其中 00000010 为 StandardInformation。30 为 FileName。因为 FileNameAttribute 总是一个常驻 Attribute，所以我将 RESIDENT_ATTRIBUTE 定义也给出。OK，现在可以继续 Dump 下一个 Attribute：

// dd @file+file->AttributeOffset+length(StandardInformationAttribute)

 d @file+30+60

0023:00344DC8 00000030 00000068 00180000 00030000 0...h.....

-------- ------

| |___这里的 NameLength 与 NameOffset 指 FileNameAttribute 名。不要与$MFT FileName 混谣。

|_指出这是一个 FileNameAttribute。

0023:00344DD8 0000004A 00010018 00000005 00050000 J.....

-------- ---- --------

| | |_根据 ValueOffset 的值，得到 FILENAME_ATTRIBUTE 的具体位置。

| |_ValueOffset 值

|_ValueLength 值

0023:00344DE8 2C1761D0 01BFB03C 2C1761D0 01BFB03C .a.,<....a.,<...

0023:00344DF8 2C1761D0 01BFB03C 2C1761D0 01BFB03C .a.,<....a.,<...

0023:00344E08 00004000 00000000 00004000 00000000 .@.....@.....

0023:00344E18 00000006 00000000 00240304 0046004D .....$.M.F.

-- --------
| |___找到$MFT 的 FileName 了吧。

|_NameLength

0023:00344E28 00000054 00000000 00000080 00000190 T.....

0023:00344E38 00400001 00010000 00000000 00000000 ..@.....

这儿给出了 Dump Attribute 的一个具体方法。最后我将给出遍历 File Record 的代码，在给出代码前应该说明一下$MFT 中$BITMAP 属性。NTFS 的这个 Attribute 相当于 LINUX EXT2 的 s_inode_bitmap 数组(Linux 2.0 版本)。所以很容易明白$BITMAP 的作用，即每 bit 指出相应 File Record 的在用情况。以下是 DumpAllFileRecord 的代码：

```
BOOL bitset(PUCHAR bitmap, ULONG i)
{
```

```
return (bitmap[i >> 3] & (1 << (i & 7))) != 0;

}

VOID DumpAllFileRecord()

{

PATTRIBUTE attr = FindAttribute(MFT, AttributeBitmap, 0);

PUCHAR bitmap = new UCHAR[AttributeLengthAllocated(attr)];

ReadAttribute(attr, bitmap);

ULONG n = AttributeLength(FindAttribute(MFT, AttributeData, 0)) / BytesPerFileRecord;

PFILE_RECORD_HEADER file = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);

for (ULONG i = 0; i < n; i++) {

if (!bitset(bitmap, i)) continue;

ReadFileRecord(i, file);

if (file->Ntfs.Type == 'ELIF' && (file->Flags & 3 )) {

attr = FindAttribute(file, AttributeFileName, 0);

if (attr == 0) continue;

PFILENAME_ATTRIBUTE name

= PFILENAME_ATTRIBUTE(Padd(attr, PRESIDENT_ATTRIBUTE(attr)->ValueOffset));

printf("%8lu %.*ws\n", i, int(name->NameLength),name->Name)
```