Hardware White Paper

Designing Hardware for Microsoft® Operating Systems

Microsoft Extensible Firmware Initiative FAT32 File System Specification

FAT: General Overview of On-Disk Format

Version 1.03, December 6, 2000
Microsoft Corporation

译注：

刚完成的项目内容涉及 FAT 文件系统，因此在查阅手册的同时把文档翻译了一下，希望能对那些和我一样初次使用 FAT 的朋友有所帮助。因本人对 FAT 并不十分了解，翻译所做的也只是文字表面工作，出现各种错误在所难免，因此：**本文只适用于初学者作为了解材料，需要获得技术信息的朋友请查阅原版英文资料！**

发现错误的朋友请mail yuwh@amoi.com.cn指正，本人将不胜感激。

Microsoft Extensible Firmware Initiative FAT32 File System Specification

IMPORTANT-READ CAREFULLY:

……

原文长 5 页，大意是告诫人们不要使用 D 版云云，此略。

本文的一些约定

以字符“0x”开头的数字为 16 进制，若开头没有字符“0x”则表明该数字为 10 进制。

本文的程序使用 C 语言书写，书写风格可能与教科书中严格定义的有所出入。

一些变量在程序中没有注明其数据类型是 16-bit 还是 32-bit，因为我们知道你有能力正确地完成这些数据类型之间的转换，并保证在转换 32-bit 为 16-bit 的过程中不会造成数据丢失。同时请注意，所有的数据类型均是无符号类型(UNSIGNED)，不要尝试使用有符号整形（signed integer types）来进行 FAT 运算，否则一些 FAT 卷将会因此而出错。

概述（适用于所有 FAT 类型）

起先所有的 FAT 文件系统都是为 IBM PC 机器而设计的，这说明了一个重要的问题：FAT 文件系统在磁盘上的数据是以“小端” （little-endian） 结构存储的。我们使用 4 个 8-bit 的字节 -- 起始字节为

byte[0],结束字节为 byte[3] -- 来存储一个 32-bit 的 FAT 项(FAT entry)。然后分别给这 32 位编号为 00-31，从下表我们可以清楚地看到这 32 位是如何排序的（最低位为 00）.

```
byte[3]   3 3 2 2 2 2 2 2
          1 0 9 8 7 6 5 4

byte[2]   2 2 2 2 1 1 1 1
          3 2 1 0 9 8 7 6

byte[1]   1 1 1 1 1 1 0 0
          5 4 3 2 1 0 9 8

byte[0]   0 0 0 0 0 0 0 0
          7 6 5 4 3 2 1 0
```

这对于那些使用“大端”（big-endian）存储结构的机器就显得尤为重要，因为在磁盘存取数据之前，必须先完成 big-endian 和 little-endian 之间的转换。

每个 FAT 文件系统由 4 部分组成，这些基本区域按如下顺序排列：

0 - 保留区（Reserved Region）

1 - FAT 区(FAT Region)

2 - 根目录区（Root Directory Region, FAT32 卷没有此域）

3 - 文件和目录数据区(File and Directory Data Region)

启动扇区与 BPB

BPB(BIOS Parameter Block)是 FAT 文件系统中第一个重要的数据结构，它位于该 FAT 卷的第一个扇区，同时也属于 FAT 文件系统基本区域的保留区。这个扇区又叫做“启动扇区”、“保留扇区”、“0 扇区”，众多的叫法都说明一个相同的问题：该扇区是 FAT 卷的第一个扇区。

这是 FAT 文件系统中第一个让人感到迷惑的地方，对于 MS-DOS 1.x 的版本，启动扇区中并没有 BPB 这么一个东西，FAT 文件系统的最早期版本只有两种不同的格式：使用于单面或双面的 360K 5 寸软盘。这两种格式是通过 FAT 的第一个字节（FAT[0]的低 8 位）来区分的。

在 MS-DOS 2.x 以后，启动扇区里增加了 BPB 用于区分磁盘介质，同时不再支持老的磁盘介质区分方式（用 FAT 的第一个字节来区分），所有的 FAT 文件系统卷必须在启动扇区中包含 BPB。

这又是一个迷惑人的地方，BPB 具体是什么样的？在 MS-DOS 2.x 的定义中，每个 FAT 卷的扇区数不能多于 65536(每个扇区 512 字节的话最多 32M)，这一限定是由于定义“总扇区数”的变量本身是一个 16-bit 的数据类型。这一个限制在 MS-DOS 3.x 中有所改进，它使用一个 32-bit 的变量来存储“总扇区数”。

在 Win95 操作系统，确切的说应该是在 OSR2（OEM Service Release 2）出现的时候 BPB 的内容有了新的变化，在这一版本中引入了新的 FAT 类型 —— FAT32。在 FAT16 中，由于 FAT 表的大小限制了有效的簇数（cluster），同时也就限制了磁盘空间的大小，如果每个扇区为 512 字节的话，那么 FAT16 格式只能支持到 2G。FAT32 的引入改变了这一状况，不再需要增加分区来管理大于 2G 的硬盘。

FAT32 的 BPB 内容和 FAT12/FAT16 的内容在 BPB_ToSet32 区域以前完全一致，而从偏移量 36 开始他们的内容有所区别,具体内容要看 FAT 类型为 FAT12/FAT16 还是 FAT32(后面的内容会提到如何区分 FAT 格式)这点保证了在启动扇区中包含一个完整的 FAT12/FAT16 或 FAT32 的 BPB 内容，这么做是为了达到最好的兼容性，同时也为了保证所有的 FAT 文件系统驱动程序能正确地识别和驱动不同 FAT 格式，并让他们良好地工作，因为他们包含了现有的全部内容。

NOTE: 在以下的描述中，凡名称与 BPB_开头的域都是 BPB 的一部分，凡名称与 BS_开头的项都是启动

扇区(boot sector)的一部分，而不是真正属于 BPB 内容。下面是 FAT 0 扇区的内容，BPB 也包含其中。

启动扇区与 BPB 结构

名称	Offset (byte)	大小 (byte)	描述
BS_jmpBoot	0	3	跳转指令。指向启动代码，允许以下两种形式： **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90** 和 **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??** 0x??表示该字节可以为任意 8-bit 值，这是 Intel x86 架构 3 字节的无条件转移指令，跳转到操作系统的启动代码，这些启动代码往往紧接 BPB 后面 0 扇区里的剩余字节，当然也可能位于其他扇区。以上的两种形式任取。**jmpBoot[0] = 0xEB** 是较常用的一种格式
BS_OEMName	3	8	建议值为“MSWIN4.1” 此域经常引起人们的误解，其实这只是一个字符串而已，Microsoft 的操作系统似乎并不关心此域。但其他厂商的 FAT 驱动程序可能会检测此项，这就是为什么建议将此域设为 “MSWIN4.1”的原因，这样可以尽量避免兼容性的问题。你可以更改它的内容，但这有可能造成某些 FAT 驱动程序无法识别该磁盘。很多情况下该域用于显示格式化该 FAT 卷的操作系统的名称。
BPB_BytsPerSec	11	2	每扇区字节数，取值只能是以下的几种情况：512、1024、2048 或是 4096，设置为 512 将取得最好的兼容性，目前有很多的 FAT 代码都是硬性的规定每扇区字节数为 512，而不是实际检测该此域的值，Microsoft 的操作系统能够很好地支持 1024、2048 和 4096 各种数值。 NOTE: 请勿曲解此处“最好的兼容性”的意思，如果某些存储介质的物体特性决定其值为 N,那么你就必须使用该数值 N,该 N 值一定是小于或是等于 4096。那么取得“最好的兼容性”的办法就是使用该特定的 N 值。
BPB_SecPerClus	13	1	每簇扇区数，其值必须是 2 的整数次方（该整数必须>=0），如 1、2、4、8、16、32、64 或 128，同时还必须保证每簇的字节数不超过 32K，既：保证（BPB_BytsPerSec * BPB_SecPerClus <= 32K (1024*32) 该值大于 32K 是绝对不允许的，虽然有些版本的操作系统支持每簇字节数最大到 64K，但很多应用程序的安装程序都无法在这样的 FAT 文件系统上正常运行。
BPB_RsvdSecCnt	14	2	保留区中保留扇区的数目，保留扇区从 FAT 卷的第一个扇区开始，此域不能为 0，对于 FAT12 和 FAT16 必须为 1，FAT32 的典型取值为 32,目前很多 FAT 程序都是硬性规定 FAT12/FAT16 的保留扇区数为 1，而不对此域进行实际的检测，Microsoft 的操作系统支持任何非零的值.
BPB_NumFATs	16	1	此卷中 FAT 表的份数。任何 FAT 格式此域都建议为 2 。虽然此域取值为其他>=1 的数值也是合法的，但是很多 FAT 程序和部分操作系统对于此项不为 2 的时候将无法正常工

			作。当不为 2 时，Microsoft 的操作系统仍能良好的工作。但仍然**强烈建议**此域为 2 。 选择此项的标准值为 2 的原因是为了提供一份 FAT 表的备份，当其中一个 FAT 表所在的扇区被损坏时我们可以从备份的 FAT 表中读取正确的数据。但是对于一些非磁盘介质的存储器（如 FLASH 卡）,这一特性变得毫无用处，如果想使用 1 个 FAT 表来节省空间，那么带来的问题将是某些操作系统无法识别该 FAT 卷。
BPB_RootEntCnt	17	2	对于 FAT12 和 FAT16 此域包含根目录中的目录项数（每个项长度为 32 bytes），对于 FAT32,此项必须为 0。对于 FAT12 和 FAT16,此数值乘以 32 必须为 BPB_BytsPerSec 的偶数倍，为了达到最好的兼容性，FAT12/FAT16 应该取值为 512.
BPB_TotSec16	19	2	早期版本中 16-bit 的总扇区数，这里的总扇区数包括 FAT 卷上四个基本区的全部扇区，此域可以为 0，若此域为 0，那么 BPB_TotSec32 必须非 0，对于 FAT32,此域必须为 0。对于 FAT12/FAT16，此域填写总扇区数， 如果该数值小于 0x10000 的话，BPB_TotSec32 必须为 0。
BPB_Media	21	1	对于“固定”（不可移动）存储介质而言，0xF8 是标准值，对于可移动存储介质，经常使用的数值是 0xF0,此域合法的取值可以取 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE 和 0xFF。另外要提醒的一点是，无论此域写入什么数值，同时也必须在 FAT[0]的低字节写入相同的值，这是因为早期的 MSDOS 1.x 使用该字节来判定是何种存储介质。
BPB_FATSz16	22	2	FAT12/FAT16 一个 FAT 表所占的扇区数，对于 FAT32 此域必须为零，在 BPB_FATSz32 中有指定其 FAT 表的大小
BPB_SecPerTrk	24	2	每磁道扇区数，用于 BIOS 中断 0x13，此域只对于有“特殊形状”（由磁头和柱面分割为若干磁道）的存储介质有效，同时必须可以调用 BIOS 的 0x13 中断得到此数值
BPB_NumHeads	26	2	磁头数，用于 BIOS 的 0x13 中断， 类似于上面的 BPB_SecPerTrk,只有对特殊的介质才有效,此域包含一个至少为 1 的数值，比如 1.4M 的软盘此域为 2
BPB_HiddSec	28	4	在此 FAT 分区之前所隐藏的扇区数，必须使得调用 BIOS 的 0x13 中断可以得到此数值，对于那些没有分区的存储介质，此域必须为 0，具体使用什么值由操作系统决定。
BPB_TotSec32	32	4	该卷总扇区数（32-bit），这里的总扇区数包括 FAT 卷上四个基本区的全部扇区， 此域可以为 0,若此域为 0，BPB_TotSec16 必须为非 0，对于 FAT32,此域一定是非 0。对于 FAT12/FAT16,如果总扇区数大于或等于 0x10000 的话，此域就是总扇区数，同时 BPB_TotSec16 的值为 0

从 Offset 36 开始 FAT12/FAT16 的内容开始区别于 FAT32,现在分两个表格列出，下表为 FAT12/FAT16 的内容：

名称	Offset (byte)	大小 (byte)	描述
BS_drvNum	36	1	用于 BIOS 中断 0x13 得到磁盘驱动器参数，（0x00 为软盘，0x80 为硬盘）。 NOTE：此域的值实际上由操作系统来决定
BS_Reserved1 （壹）	37	1	保留（供 NT 使用），格式化 FAT 卷时必须把此域设置为 0
BS_BootSig	38	1	扩展引导标记（0x29）,用于指明此后的 3 个域可用
BS_VolID	39	4	卷标序列号,此域以 BS_VolLab 一起可以用来检测磁盘是否正确,FAT 文件系统可以用此判断连接的可移动磁盘是否正确，此域往往是由时间和日期组成的一个 32 位值。
BS_VolLab	43	11	磁盘卷标，此域必须与根目录中 11 字节长的卷标一致。 NOTE： FAT 文件系统必须保证在根目录的卷标文件更改或是创建的同时，此域的内容能得到及时的更新，当 FAT 卷没有卷标时，此域的内容为“NO NANM ”
BS_FilSysType	54	8	以下的几种之一:“ FAT12 ”、“FAT16 ”、“FAT32 ”. NOTE： 不少人错误的认为 FAT 文件系统的类型由此域来确定，仔细点你就能发现此域并不是 BPB 的一部分，只是一个字符串而已，Microsoft 的操作系统并不使用此域来确定 FAT 文件的类型，因为它常常被写错或是根本就不存在，后面将讨论如何来检测一个 FAT 文件系统的类型，但不管如何，建议您在此域填写正确的信息，因为一些非 Microsoft 的操作系统会检测此域。

下表为 FAT32 的内容：

名称	Offset (byte)	大小 (byte)	描述
BPB_FATSz32	36	4	一个 FAT 表所占的扇区数，此域为 FAT32 特有，同时 BPB_FATSz16 必须为 0。
BPB_ExtFlags	40	2	此域 FAT32 特有。 Bits 0-3:不小于 0 的活动 FAT（active FAT）数目，只有在镜像(mirroring)禁止时才有效。 Bits 4-6: 保留。 Bits 7: -- 0 表示 FAT 实时镜像到所有的 FAT 表中。 -- 1 表示只有一个活动的 FAT 表，这个表就是 bits 0-3 所指定的那个。 Bits 8-15: 保留
BPB_FSVer	42	2	此域 FAT32 特有。高位为 FAT32 的主版本号，底位为次版本号，这个版本号是为了以后更高级 FAT 版本考虑，假设当前的操作系统所能支持的 FAT32 版本号为 0:0。那么该操作系统检测到此域不为 0 时，它便会忽略这个 FAT 卷，因为它的版本号比系统能支持的版本要高。
BPB_RootClus	44	4	此域 FAT32 特有。根目录所在第一个簇的簇号，通常该数值为 2，但不是必须为 2。 NOTE： 磁盘工具在改变根目录的位置时，必须想办法让磁盘上第一个非坏簇作为根目录的第一个簇（比如第 2 簇，除

			非它已经被标记为坏簇)，这样的话，如果此域正好为 0 的话磁盘检测工具也能轻松的找到根目录所在簇的位置。
BPB_FSInfo	48	2	此域 FAT32 特有。保留区中 FAT32 卷 FSINFO 结构所占的扇区数，通常为 1。 **NOTE：** 在 Backup Boot 中会有一个 FSINFO 的备份，但该备份只是更新其中的指针，也就是说无论是主引导记录还是备份引导记录都是指向同一个 FSINFO 结构
BPB_BkBootSec	50	2	此域 FAT32 特有。如果不为 0，表示在保留区中引导记录的备份数据所占的扇区数，通常为 6。同时不建议使用 6 以外的其他数值。
BPB_Reserved	52	12	此域 FAT32 特有。用于以后 FAT 的扩展使用，对于 FAT32, 此域用 0 填充。
BS_DrvNum	64	1	与 FAT12/FAT16 的定义相同，只不过两者位于启动扇区不同位置而已。
BS_Reserved1	65	1	与 FAT12/FAT16 的定义相同，只不过两者位于启动扇区不同位置而已。
BS_BootSig	66	1	与 FAT12/FAT16 的定义相同，只不过两者位于启动扇区不同位置而已。
BS_VolID	67	4	与 FAT12/FAT16 的定义相同，只不过两者位于启动扇区不同位置而已。
BS_FilSysType	71	11	与 FAT12/FAT16 的定义相同，只不过两者位于启动扇区不同位置而已。
BS_FilSysType	82	8	通常设置为“FAT32　”，请参照 FAT12/FAT16 部分关于此域的陈述，该域的内容和 FAT 类型的判定无关。

关于 FAT 启动扇区还有一点重要的说明：我们假设里面的内容是按字节排序的，那么扇区[510]的内容一定是 0x55，扇区[511]的内容一定是 0xAA。

NOTE： 很多 FAT 的资料文档会错误地把 0xAA55 说成是“启动扇区最后两字节的内容”，这样的陈述是正确的如果 -- 仅仅是如果 -- BPB_BytsPerSec 的值为 512 的话。若是 BPB_BytsPerSec 的值大于 512，该标记的位置并没有变（虽然在启动扇区的最后两个字节写上 0xAA55 完全没有问题）。

关于 BPB_TotSec16/32 这里再作一点补充:假设现在我们有一块磁盘或一个分区,它的扇区数为 DskSz，如果 BPB_aTotSec（BPB_TotSec16 或是 BPB_TotSec32 其中不为 0 的那个）的值*小于或等于* DskSz 并不会使该 FAT 卷在使用中出现什么错误，实际上，BPB_TotSec16/32 的值不要比 DskSz 小得离谱就不会有什么错误。

这样做将造成磁盘空间的浪费，程序本身并不会认为该 FAT 卷存在什么错误。但是，如果 BPB_TotSec16/32 的值比 DskSz 大的话将会使 FAT 卷遭到严重的损坏，因为它超出了存储介质或是磁盘分区的边界。当 BPB_TotSec16/32 的值比 DskSz 大时，一些数据将不幸地被丢失。

FAT 数据结构(FAT Data Structure)

接下来一个重要的数据结构就是 FAT 表(File Allocation Table) ，它是一一对应于数据区簇号的列表。

文件系统分配磁盘空间按簇来分配的。因此，文件占用磁盘空间时，基本单位不是字节而是簇，即使某个文件只有一个字节，操作系统也会给他分配一个最小单元——即一个簇。为了可以将磁盘空间有序地分配给相应的文件，而读取文件的时候又可以从相应的地址读出文件，我们把数据区空间分成

BPB_BytsPerSec * BPB_SecPerClus 字节长的簇来管理， FAT 表项的大小与 FAT 类型有关，FAT12 的 表项为 12-bit，FAT16 为 16-bit，而 FAT32 则为 32-bit。对于大文件，需要分配多个簇。同一个文件的数据并不一定完整地存放在磁盘中一个连续的区域内，而往往会分成若干段，像链子一样存放。这种存储方式称为文件的链式存储。为了实现文件的链式存储，文件系统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，FAT 表的对应表项中记录着它所代表的簇的有关信息：诸如是否空，是否是坏簇，是否已经是某个文件的尾簇等。

以 FAT16 为例说明 FAT 区的结构如下：

表项	示例代码	描述
0	FFF8	磁盘标识字，必须为 FFF8
1	FFFF	第一簇已经被占用
2	0003	0000h : 可用簇
3	0004	0002h - FFFEFh: 已用簇，表项中存放文件下
……	……	个簇的簇号
N	FFFF	FFFF0h - FFF6h : 保留簇
N+1	0000	FFF7h : 坏簇
……	……	FFF8h - FFFFh : 文件的最后一簇

FAT 的项数与硬盘上的总簇数相关（因为每一个项要代表一个簇，簇越多当然需要的 FAT 表项越多），每一项占用的字节数也与总簇数有关（因为其中需要存放簇号，簇号越大当然每项占用的字节数就大）

这里说一下 FAT 目录，其实它和普通的文件并没有什么不一样的地方，只是多了一个表示它是目录的属性（attrib），另外就是目录所链接的内容是一个 32 字节的目录项（32-byte FAT directory entries 后面有具体讨论）。除此之外，目录和文件没什么区别。FAT 表是根据簇数和文件对应的。第一个存放数据的簇是簇 2。

簇 2 的第一个扇区（磁盘的数据区）根据 BPB 来计算，首先我们计算根目录所占的扇区数：

RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;

因为 FAT32 的 BPB_RootEntCnt 为 0， 所以对于 FAT32 卷 RootDirSectors 的值也一定是 0。上式中的 32 是每个目录项所占的字节数。计算结果四舍五入。

数据区的起始地址，簇 2 的第一个扇区由下面公式计算：

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
FirstDataSector = BPB_RsvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;
```

NOTE: 扇区号指的是针对卷中包含 BPB 的第一个扇区的偏移量（包含 BPB 的第一个扇区是扇区 0），并不是必须直接和磁盘的扇区相对应。因为卷的扇区 0 并不一定就是磁盘的扇区 0。

给一个合法的簇号 N，该簇的第一个扇区号（针对 FAT 卷扇区 0 的偏移量）由下式计算：

FirstSectorofCluster = ((N - 2) * BPB_SecPerClust) + FirstDataSector;

NOTE: 因为 BPB_SecPerClus 总是 2 的整数次方（1，2，4，8，……）这意味这 BPB_SecPerClus 的乘除法运算可以通过移位（SHIFT）来进行。在当前 Intel x86 架构 2 进制的机器上乘法（MULT）和除法（DIV）的机器指令非常的繁杂和庞大，而使用移位来运算则会相对的快许多。

FAT 类型辨别（FAT Type Determination）

这是一个经常产生错误的地方，并且常常会出现诸如 “off by 1”， “off by 2”， “off by 10”和 “massively off”的错误，事实上，FAT 类型的检测非常简单，FAT 的类型 ——FAT12,或是 FAT16 或是 FAT32 —— 只能通过 FAT 卷中簇的数量来判定，没有其他办法。

请仔细阅读本段的每一个细节，每个词都很关键。比如“簇数(count of cluster)”并不是指“最大可取得的簇的数量（maximum valid cluster number)”，因为数据区的第一个簇是簇 2 而不是 0 或 1。

首先我们讨论这个“簇数”是如何计算的，它完全根据 BPB 的内容来确定，我们先计算根目录所占的扇区数（前面已经有叙述）。

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec  -  1)) / BPB_BytsPerSec;
```

FAT32 的 RootDirSectors 为 0。

接下来我们检测数据区中扇区数：

```
If (BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
If (BPB_TotSec16 != 0)
    TotSec = BPB_TotSec16;
Else
    TotSec = BPB_TotSec32;
DataSec = TotSec  -  (BPB_RsvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
```

计算簇数：

```
CountofClusters = DataSec / BPB_SecPerClus;
```

请记住计算结果四舍五入。

现在我们就可以判定 FAT 的类型了，这部分请仔细阅读，*否则会导致 off-by-one 的错误*

在下面的程序中， “<”和 “<=”是不一样的，另外请注意数字不要弄错。

```
If (CountofCluster < 4085) {
    /* FAT 类型是 FAT12 */
}
Else if (CountofCluster < 65525) {
    /* FAT 类型是 FAT16 */
}
Else {
    /* FAT 类型是 FAT32*/
}
```

这是检测 FAT 类型的唯一办法。世上不存在簇数大于 4084 的 FAT12 卷，也不存在簇数小于 4085 或是大于 65524 的 FAT16 卷，同样没有哪个 FAT32 卷的簇数小于 65525。如果你坚持要违背这个规则来创建一个 FAT 卷那么 Microsoft 的操作系统将无法对此卷进行操作，因为它不认为这是 FAT 文件系统。

NOTE: 如前面所说，目前有很多 FAT 的代码有一些错误，常常会出现 off by 1，2，8，10 或是 off by 16 的错误 。因此，为和现存的所有代码取得最好的兼容性，强烈建议在格式化 FAT 文件系统时，尽量使簇数的取值不要接近 4085 或 65525，最好能和这分割点的值相差 16 或更多。

同时请注意这里的簇数（Count of Cluster）是指数据区所占簇的数量（the count of data clusters）从簇 2 开始算起，而“最大可用的簇数”（Maximum valid cluster number for the volume）是簇数 + 1，“包括保留簇的簇数”（count of cluster including the two reserved cluster）则为 簇数 + 2。

FAT 的另一个重要计算公式：给一个簇号 N，它位于 FAT 表的什么位置呢？对于 FAT16 和 FAT32 都比较容易计算，而 FAT12 则会复杂一点：

```
If (BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
If (FATType == FAT16)
    FATOffset = N * 2;
Else if (FATType == FAT32)
    FATOffset = N * 4;
ThisFATSecNum = BPB_RsvdSecCnt + (FATOffset / BPB_BytsPerSec);
ThisFATEntOffset = REM (FATOffset / BPB_BytsPerSec);
```

REM(…)为求余操作符，就是求 FATOffset 除以 BPB_BytsPerSec 的余数。ThisFATSecNum 是 FAT 表中包含簇 N 的扇区数，如果你想得到第二个 FAT 表中的扇区数，只要加上 FATSz 就是，如果想的到第三个 FAT 表中的扇区数，只需要加上 FATSz * 2，依次类推。

现在你得到扇区数 ThisFATSecNum(记住这是针对 FAT 卷扇区 0 的偏移量)，假设把该值读入到一个指定的 8-bit SecBuff，同时假定数据类型 WORD 是一个 16-bit 的不无符号类型，而 DWORD 是一个 32-bit 的无符号类型。

```
If (FATType == TAF16)
    FAT16ClusEntryVal = *((WORD *) & SecBuff[ThisFATEntOffset]);
Else
    FAT32ClusEntryVal = (*((DWORD *) & SecBuff[ThisFATEntOffset]) & 0x0FFFFFF;
```

取得该扇区的内容。

设置该扇区的值使用如下算式：

```
If (FATType == FAT16)
    *((WORD *) &SecBuff[ThisFATEntOffset]) = FAT16ClusEntryVal;
Else {
    FAT32ClusEntryVal = FAT32ClusEntryVal & 0x0FFFFFFF;
    *((DWORD *) & SecBuff[ThisFATEntOffset]) =
        (*((DWORD *) & SecBuff[ThisFATEntOffset])) & 0xF0000000;
    *((DWORD *) & SecBuff[ThisFATEntOffset]) =
        (*((DWORD *) & SecBuff[ThisFATEntOffset])) | FAT32ClusEntryVal;
}
```

我们看看上述FAT 代码是如何工作的，实际上每个FAT32的FAT表项只有28-bit可以使用，它的高4位保留，这4位只有在被格式化的时候会被使用到，在格式化时整个FAT32单元的32-bit都被设为0，包括高位的4-bit。

另外要说明的一点，这也是经常被混淆的地方，因为FAT32表项实际上被使用的只有28-bit而不是32-bit。比如，以下几个FAT32簇的值为0x10000000,0xF0000000和0x00000000都表示该簇为空，因为程序忽略了高位4-bit的值。如果当前簇的值为0x30000000，你想要把数值0x0FFFFFF7写入当前簇来标记坏簇，那么当你的操作结束后该簇的值实际上是0x3FFFFFF7,因为你必须舍去0x0FFFFFF7这个坏簇标记高位的4-bit.

因为BPB_BytsPerSec的值一定能够被2和4整除，对于FAT16/FAT32来说，你不必担心元素会超越扇区的边界，但对于FAT12,你就必须小心了。

FAT12的代码会显得复杂一点，因为它每个元素占1.5个字节（12-bit）。

```
If (FATType == FAT12)
    FATOffset = N + (N / 2);
/* 注意算式并没有乘以浮点数1.5。除以2的值四舍五入 */
    ThisFATSecNum = BPB_RsvdSecCnt + (FATOffset / BPB_BytsPerSec);
    ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

现在我们必须考虑扇区边界的情况。

```
If (ThisFATEntOffset == (BPB_BytsPerSec - 1)) {
    /* 这个簇跨越了FAT的扇区边界，处理这个问题有很多方法， */
    /* 其中最简单的一种就是当FAT类型为FAT12则同时读取两个 */
    /* 扇区的内容到内存中（如果你想读取扇区N,同时你把扇区 */
    /* N+1的内容也读入内存中来，除非扇区N是FAT的最后一个扇 */
    /* 区）这样做实际上是避免扇区的边界检测               */
}
```

现在我们可以象FAT16一样使用WORD数据类型来对FAT12的数据进行读取，但仍需要注意，如果簇号为偶数，我们取16-bit中的低12-bit，如果是奇数则取高12-bit，

```
FAT12ClusEntryVal = *((WORD *) & SecBuff[ThisFATEntOffset]);
If (N & 0x0001) {
    FAT12ClusEntryVal = FAT12ClusEntryVal >> 4      /* 簇号为偶数 */
    *((WORD *) & SecBuff[ThisFATEntOffset]) =
        (*((WORD *) & SecBuff[ThisFATEntOffset])) & 0x000F;
}
Else {
    FAT12ClusEntryVal = FAT12ClusEntryVal & 0x0FFF;     /* 簇号为奇数 */
    *((WORD *) &SecBuff[ThisFATEntOffset]) =
    (*((WORD *) &SecBuff[ThisFATEntOffset])) & 0xF000;
}
*((WORD *) &SecBuff[ThisFATEntOffset]) =
(*((WORD *) &SecBuff[ThisFATEntOffset])) | FAT12ClusEntryVal;
```

NOTE: 这里的 >>为右移操作符，并往高 4 位填充 0； <<为左移操作符，同时低 4 位用 0 填充。

数据区中的文件是按以下方式与 FAT 表相对应的：数据区中文件存放的第一个簇的簇号被记录在目录项中，文件就是根据这个簇号和 FAT 表相关联，数据区中文件的位置由前面讨论的 FirstSectorofCluster 来计算。

对于一个文件大小为 0 —— 一个没有数据的文件 —— 在目录项中分配的第一个簇的簇号为 0，此簇（参见前面讨论的 ThisFATSecNum 和 ThisFATEntOffset）的内容要么是一个 EOC 标记（End of Cluster chain 簇链表结束标记）要么就是该文件下一个簇的簇号。EOC 的值和 FAT 类型有关（假设 FATContent 是需要检测看是否包含 EOC 标记的簇的内容）

```
isEOF = FALSE;
if (FATType == FAT12) {
    if (FATContent >= 0x0FF8)
        isEOF = TRUE;
}
Else if (FATType == FAT16) {
    If (FATContent >= 0xFFF8)
        isEOF = TRUE;
```

```
}
Else if (FATType == FAT32) {
     If (FATContent >= 0x0FFFFFF8)
          isEOF = TRUE;
}
```

注意：包含 EOC 标记的簇属于当前文件并且是当前文件的最后一个簇。Microsoft 的操作系统设置 EOC 标记时 FAT12 使用 0x0FFF,FAT16 使用 0xFFFF,FAT32 使用 0x0FFFFFFF，但有一些运行于 Microsoft 系统的工具并不使用这个值。

还有一个特殊的标记就是“坏簇（BAT CLUSTER）”标记，任何包含“坏簇”标记的簇都不应该被列到剩余簇的范畴内，这个“坏簇”标记对 FAT12 是 0x0FF7, FAT16 是 0xFFF7, FAT32 是 0x0FFFFFF7。另外，这些坏簇看起来也象是丢失的簇——们似乎已经分配出去，因为它们的值不为 0，但同时它们又不属于任何文件。磁盘修复程序一定要认出这些被标记坏簇标记的“丢失簇”，并且不去修改它们的内容。

NOTE: 对于 FAT12 和 FAT16 而言，坏簇标记不可能是某个已分配簇的簇号，但对于 FAT32 而言，0x0FFFFFF7 就有可能是某个簇的簇号，因此 FAT32 文件系统必须避免把 0x0FFFFFF7 这个数字分配出去作为某个簇的簇号。

FAT 表中剩余簇的列表就是卷中所有内容为 0 的簇的列表。这些数据必须尽早取得并记录下来以表示剩余的簇是已经被使用的。这个列表并没有存储在卷中的任何一个地方，他必须在系统挂上（mount）该卷时由 FAT 扫描程序获得内容为 0 的簇的列表。FAT32 的 BPB_FSInfo 扇区*可能会* 包含剩余簇的数量，请参阅 FAT32 关于 BPB_FSInfo 扇区的讨论部分。

在 FAT 卷起始部分的两个保留扇区到底是做什么用的呢？第一个保留簇 FAT[0]，它的低位 8-bit 为 BPB_Media,剩余的位用 1 填充，比如：BPB_Media 的内容为 0xF8，那么 FAT12 的内容为 0x0FF8，FAT16 为 0xFFF8,FAT32 的内容为 0x0FFFFFF8.第二个保留簇 FAT[1]在格式化的时候被填充 EOC 标记。FAT12 卷此域不用，其值始终为 EOC 标记。FAT16 和 FAT32 此域的高 2-bit 可以被用于标记磁盘是否为“脏”（下面描述），剩余的位均用 1 填充。请注意 FAT16 和 FAT32 这两位的位置是不一样的，因为是高位 2-bit.

对于 FAT16:

```
ClnShutBitMask = 0x8000;
HrdErrBitMask = 0x4000;
```

对于 FAT32:

```
ClnShutBitMask = 0x08000000;
HrdErrBitMask = 0x04000000;
```

Bit ClnShutBitMask - 如果此位为 1，那么卷是“干净”的。 如果为 0，那么卷是“脏”的。意味着系统在上次卸载（unmount）此卷时没有正常地断开连接，此时建议使用 ChkDsk/Scandisk 等工具来检测磁盘是否有错误。

Bit HrdErrBitMask - 如果此位为 1，表示没有发生磁盘读/写错误。如果为 0，表示系统在上次挂载该卷时有发生过磁盘读/写错误，此时建议运行 ChkDsk/Scandisk 等工具来扫描磁盘表面看看是否有出现新的坏簇。

关于 FAT 表这里还有两个重要的问题：

1. FAT 表的结束扇区不一定就是 FAT 表的最后一个扇区，FAT 表的结尾扇区位于簇号为 CountofClusters + 1（参阅前面关于 CountofClusters 的计算式）的簇中。这个扇区未必在 FAT 表的最后面。FAT 程序不应该尝试着去访问 CountofClusters + 1 以后的簇. FAT 格式化程序应该把这个簇号后面的所有簇用 0 填充。
2. BPB_FATSz16(对于 FAT32 为 BPB_FATSz32)的值会比它实际需要的大，也就是说，在 FAT 表中可能有部分扇区没有被使用。因此，FAT 表的结束扇区都是由 CountofClusters + 1 来 计

算得到，而不是使用 BPB_FATSz16/32 来计算。FAT 程序不因该尝试去访问这些“额外”的扇区。FAT 格式化程序因该把这些扇区用 0 来填充。

初始化 FAT 卷（FAT Volume Initialization）

读到这里，细心的读者一定会发现一个有趣的问题。前面说过 FAT 的类型（FAT12, FAT16 或是 FAT32）是根据总的簇数来辨别——并且数据区中最大可取得的扇区数由 FAT 表的大小来决定——那么当一个磁盘还没有格式化时，我们无法得到这 BPB 数据，这时是如何检测并计算出正确的值来放到 BPB_SecPerClus 和 BPB_FATSz16 或是 BPB_FATSz32 中呢？Microsoft 的操作系统使用一些固定的值和表格配合一个巧妙的算法来完成这些工作。

Microsoft 只在软盘上使用 FAT12 文件系统，因为软盘的种类很少，并且都有固定的参数，格式化时使用的是一张简单的表：

“如果这是一张这种格式的软盘，那么它的 BPB 看起来就是这样子。(If it is a floppy of this type, then the BPB look like this.)”

对于 FAT12 格式的计算相对简单，所有写到 BPB 中的值都可以在一张纸上用手算出来的（当然得小心使 cluster 的值始终小于 4085），如果存储介质的容量大于 4M，那就别再麻烦 FAT12 了，只需要把 BPB_SecPerClus 的值改小一点，这个卷就成为 FAT16 了。

本节以下部分描述如何驱动每扇区 512 字节的 FAT 卷。如果扇区大小不是这样的话，你将不能使用这个表格和算法。不同的磁盘的扇区大小不一。这里根据磁盘容量的大小选择一个“合适的值”来简单地区分 FAT 类型。如果磁盘容量小于该值就是 FAT16；如果大于或等于该值，那么就是 FAT32。Windows 操作系统选择该值为 512MB，任何小于 512MB 的卷都是 FAT16 卷，任何大于或等于 512MB 的卷都是 FAT32 卷。

这里请特别注意，别过早地下结论。

有很多 FAT16 卷的容量都大于 512MB，因为有很多不同的办法可以强制把磁盘格式化成 FAT 16 格式而不是象默认的那样格式化成 FAT32 格式，并且不同的 FAT 程序遵循不同的规定来格式化磁盘。这里我们讨论的只是 MS-DOS 和 Windows 默认情况下是如何处理未格式化的磁盘。这里有两个表格，一个用于 FAT16, 另外一个用于 FAT32。这个表其中的一项是根据每扇区 512 字节的磁盘空间大小计算而来（该数值将写到 BPB_TotSec16 或是 BPBSec32 中），还有一个值用来设置 BPB_SecPerClus。

```
struct DSKSZTOSECPERCLUS {
  DWORD       DiskSize;
  BYTE        SecPerClusVal;
};
/*
* 这个表使用于 FAT16，注意这个表包含了大于 512MB 的情况，虽然默认
* 情况下使用的都是<512MB。通过比较磁盘大小的是否小于或等于表中的
* 第一项 DiskSize 来决定采用哪个值。为了能够让这张表能够正确地工作，
* BPB_RsvdSecCnt 的值必须为 1， BPB_NumFATs 的值必须为 2，还有
* BPB_RootEntCnt 的值必须为 512，如果这些值不符的话，必须改变表中第
* 一项 DiskSize 的值否则磁盘的簇数可能会小于 FAT16 所要求的数值。
*/
    DSKSZTOSECPERCLUS DskTableFAT16 [ ] = {
        {8400,     0}, /* 磁盘容量最大为 4.1 MB, SecPerClusVal 的值为 0 表示这是一个错误*/
        { 32680,       2}, /*磁盘容量最大为 16 MB, 1k cluster */
        { 262144,      4}, /*磁盘容量最大为 128 MB, 2k cluster */
```

```
        { 524288, 8}, /*磁盘容量最大为 256 MB, 4k cluster */
        { 1048576,    16}, /*磁盘容量最大为 512 MB, 8k cluster */
        /* 除非强制使用 FAT16，否则以下数据不使用 */
        { 2097152,    32}, /*磁盘容量最大为 1 GB, 16k cluster */
        { 4194304,    64}, /*磁盘容量最大为 2 GB, 32k cluster */
        { 0xFFFFFFFF, 0} /*磁盘容量超过 2GB, SecPerClusVal 的值为 0 意味着这是一个错误*/
    };

/*
* 这个表使用于 FAT32，注意这个表包含了小于 512 的情况，虽然默认
* 情况下使用的都是 >= 512MB，通过比较磁盘大小的是否小于或等于
* 表中的第一项 DiskSize 来决定采用哪个值。为了使这个表能正确地工作，
* BPB_RsvdSecCnt 的值必须为 32，BPB_NumFATs 的值必须为 2，如果这
* 些值不符的话，必须改变表中第一项 DiskSize 的值否则磁盘的簇数可能
* 会小于 FAT32 所要求的数值。
*/
    DSKSZTOSECPERCLUS DskTableFAT32 [ ] = {
      { 66600, 0},/*磁盘容量最大为32.5 MB, SecPerClusVal的值为0表示这是一个错误 */
      { 532480,   1}, /*磁盘容量最大为260 MB, .5k cluster */
      { 16777216, 8}, /*磁盘容量最大为8 GB, 4k cluster */
      { 33554432, 16}, /*磁盘容量最大为16 GB, 8k cluster */
      { 67108864, 32}, /*磁盘容量最大为32 GB, 16k cluster */
      { 0xFFFFFFFF, 64}/* 磁盘容量超过32GB, 32k cluster */
    };
```

这样，只要给出磁盘的大小和 FAT 的类型就可以确定 BPB_SecPerClus 的值，现在我们唯一所缺的就是 FATSz16 或 FATSz32 的大小。这里我们假设 BPB_RootEntCnt, BPB_RsvdSecCnt 和 BPB_NumFATs 的值已经按照上面的约定正确地设置。同时我们还假设 DiskSize 就是我们要写到 BPB_TotSec32 或是 BPB_TotSec16 的值。

```
    RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec  -  1)) / BPB_BytsPerSec;
    TmpVal1 = DskSize  -  (BPB_RsvdSecCnt + RootDirSectors);
    TmpVal2 = (256 * BPB_SecPerClus) + BPB_NumFATs;
    If(FATType == FAT32)
        TmpVal2 = TmpVal2 / 2;
    FATSz = (TMPVal1 + (TmpVal2  -  1)) / TmpVal2;
    If(FATType == FAT32) {
        BPB_FATSz16 = 0;
        BPB_FATSz32 = FATSz;
    } else {
        BPB_FATSz16 = LOWORD(FATSz);
        /* FAT16 的 BPB 没有 BPB_FATSz32 */
    }
```

不要花费太多的工夫去琢磨上面的代码是如何工作的，这个算法的原理十分复杂，放在这里的目的只是说明 Microsoft 的操作系统是如何实现磁盘初始化的。但是，上面的代码并不完美，在一些偶然的情况

下，它会使 FAT16 的 FATSz 比实际需要的多 2 个扇区，或使 FAT32 的 FATSz 多 8 个扇区，不过它永远不会使 FATSz 的值比实际需要的小。因为 FATSz 的值比实际的大并不会影响文件系统正常工作，唯一的缺点就是造成很少一部分磁盘空间的浪费。由于以上代码是如此的简单可靠，这在很大程度上弥补了它的不足。

FAT32 FSInfo 扇区结构和备份启动扇区

FAT32 的 FAT 表可能会非常的庞大，而不像 FAT16 的值被限制在 128K 或 FAT12 被限制在 6K 范围内一样，因此，在 FAT32 卷中存放着 “最新”的剩余簇的数量，在 API 函数想知道剩余空间时（比如在 DIR 命令的最后显示剩余空间）不必马上去计算该数值。FSInfo 的扇区号存放在 BPB_FSInfo 中，对于 Microsoft 的操作系统此值为 1。下表是 FSInfo 的结构

名称	Offset (byte)	大小 (byte)	描述
FSI_LeadSig	0	4	值为 0x41615252,这个标记用来表示该扇区为 FSInfo 扇区。
FSI_Reserved1	4	480	保留为以后扩展使用，FAT32 格式化程序应该把此域全部设置为 0，当前版本的 FAT 程序不可以访问该域。
FSI_StrucSig	484	4	值为 0x61417272, 更具体地表明该扇区已经被使用
FSI_Free_Count	488	4	保存最新的剩余簇数量，如果为 0xFFFFFFFF 表示剩余簇未知，需要重新计算，初此之外其他的值都可以用，而且不要求十分精确，但必须保证其值<=磁盘所有的簇数。
FSI_Nxt_free	492	4	该域为 FAT 驱动程序提供一条有利的线索,它告诉驱动程序从哪里开始寻找剩余簇。因为 FAT32 的 FAT 表可能非常的庞大,如果已经分配的簇很多的话要从头开始查找剩余簇将耗费大量时间。通常这个值被设定为驱动程序最后分配出去的簇号。如果值为 0xFFFFFFFF，那么驱动程序必须从簇 2 开始查找，除此之外其他的值都可以使用，当然前提是这个值必须合法的。
FSI_Reserved2	496	12	保留为以后扩展使用，FAT32 格式化程序应该把此域全部设置为 0，当前版本的 FAT 程序不可以访问该域。
FSI_TrailSig	508	4	值为 0xAA550000, 此结束标记用来表示这是一个 FSInfo 扇区，注意次域的高两位的偏移量为 510 和 511，这和启动扇区在相同偏移处的标记是一样的。

FAT32 区别于 FAT12/FAT16 另外一个地方就是 BPB_BkBootSec,FAT12/FAT16 有可能由于丢失启动扇区的内容而使得整个卷都无法访问，这是一个“单点错误（single point of failure)”，为了避免这种严重的后果，FAT32 中引进了 BPB_BkBootSec，FAT32 在扇区号为 6 的地方完整地拷贝了一份启动扇区的备份，包括 BPB 的内容。

当启动扇区的内容被损坏后，磁盘修复程序只需要把启动备份扇区中的数据拷回到启动扇区即可，即使在启动扇区被损坏的情况下，磁盘驱动程序仍然可以在更换硬盘之前正常访问该卷。

在第二种情况下——扇区 0 被损坏——这就是为什么 BPB_BkBootSec 的值为什么必须为 6 的原因，当扇区 0 不可读时，很多不同的操作系统都是硬性检查 FAT32 卷在扇区 6 的启动备份。保存在扇区 6 中的是一个完整的启动记录。Microsoft 的 FAT32 的“启动扇区”实际上是由 3 个长度为 512 字节的扇区组成，在 BPB_BkBootSec 扇区开始的启动备份中完整地包含了这 3 个扇区。FSInfo 也在其中，虽然在备份中 BPB_FSInfo 的内容和 0 扇区中 BPB_FSInfo 所指向的是同一个 FSInfo 结构。

NOTE: 这三个扇区和启动扇区一样也在偏移量为 510 和 511 的地方包含标记 0xAA55（参看前面的叙

述)。

FAT 目录结构（FAT Directory Structure）

这里我们先忽略长目录项的情况，只讨论短目录项。

FAT 目录其实就是一个由 32-bytes 的线性表构成的“文件”。根目录（root directory）是一个特殊的目录，它存在于每一个 FAT 卷中。对于 FAT12/16,根目录存储在磁盘中固定的地方，它紧跟在最后一个 FAT 表后。根目录的扇区数也是固定的，可以根据 BPB_RootEntCnt 计算得出（参见前文计算公式），对于 FAT12/16,根目录的扇区号是相对该 FAT 卷第一个扇区（0 扇区）的偏移量。

FirstRootDirSecNum = BPB_RsvdSecCnt + (BPB_NumFATs * BPB_FATSz16);

FAT32 的根目录由簇链组成，其扇区数不确定，这点和普通的文件相同，根目录的第一个扇区号存储在 BPB_RootClus 中，根目录不同于其他的目录，没有日期和时间戳，也没有目录名(“/” 并不是其目录名),同时根目录里没有 “.” 和 “..”这两个目录项,根目录另一个特殊的地方在于,根目录中有一个设置了 ATTR_VOLUME_ID 位(见下表)的文件,这个文件在整个 FAT 卷中是唯一的.

FAT 的 32-byte 目录项结构

名称	Offset (byte)	大小 (byte)	描述
DIR_Name	0	11	短文件名
DIR_Attr	11	1	文件属性: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID 前两个属性位为保留位,在文件创建时应把这两位设为 0,在以后的使用中不能读写和更改.
DIR_NTRes	12	1	保留给 Window NT 使用,在文件创建时设置该位为 0, 在以后的使用中不能读写和更改.
DIR_CrtTimeTeenth	13	1	文件创建时间的毫秒级时间戳, 由于 DIR_CrtTime 的精度为 2 秒,所以此域的有效值在 0-199 之间
DIR_CrtTime	14	2	文件创建时间
DIR_CrtDate	16	2	文件创建日期
DIR_LastAccDate	18	2	最后访问日期,请注意并没有最后访问时间域,而只有日期,这日期是指文件被读或写的日期,如果是写,该日期还应该被写到 DIR_WrtDate 中
DIR_FstClusHI	20	2	该目录项簇号的高位字(FAT12/16 此位为 0)
DIR_WrtTime	22	2	最后写的时间,文件创建被认作写.

DIR_WrtDate	24	2	最后写的日期,文件创建被认作写.
DIR_FstClusLO	26	2	该目录项簇号的低位字.
DIR_FileSize	28	2	文件大小,由 32-byte 双字组成.

DIR_NAME[0]

此处特别注释目录项的第一个字节(DIR_NAME[0]).

• 如果 DIR_Name[0] == 0xE5, 则此目录为空(目录项不包含文件和目录)

• 如果 DIR_Name[0] == 0x00, 则此目录为空(同 0xE5),并且此后的不再分配有目录项(此后所有的 DIR_Name[0]均为 0).

不同于 0xE5,如果 DIR_Name[0]的值为 0，那么 FAT 程序将不会再去检测其后续的磁盘空间，因为这些空间都是空闲的。

•如果 DIR_Name[0] == 0x05,则文件名在该位的实际值为 0xE5,0xE5 是日文中合法的字符,当需要用 0xE5 来作为 DIR_Name[0]时使用 0x05 来代替，避免程序误认为该目录项为空。

DIR_Name 域实际由两部分组成：8 个字符的主文件名和 3 个字符的扩展名。两部分如果字符数不够的话由空格(0x20)填充（trailing space padded）.

DIR_Name[0]不允许为 0x20,主文件名和扩展名之间的间隔 ‘.’并不真实的存在于 DIR_Name 中，小写字母不允许出现在 DIR_Name 中(这些字符因不同的国家和地区而已).

• 以下字符不允许出现在 DIR_Name 中的任何位置:

0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D,还有 0x7C。

以下是一些例子显示用户输入的文件名如何以 DIR_Name 对应:

“foo.bar” -> “FOO BAR”
“FOO.BAR” -> “FOO BAR”
“Foo.Bar” -> “FOO BAR”
“foo” -> “FOO ”
“foo.” -> “FOO ”
“PICKLE.A” -> “PICKLE A ”
“prettybg.big” -> “PRETTYBGBIG”
“ .big” -> 非法, DIR_Name[0] 不能为 0x20。

在 FAT 的目录中，所有的文件名都是唯一的,上面例子中的三个文件名看起来似乎各有千秋,事实上它们都是相同的文件名,在同一个目录中,只能有一个 DIR_Name 被设置为 “FOO BAR”。

文件属性(DIR_Attr):

ATTR_READ_ONLY : 对这个文件的写操作将会失败.
ATTR_HIDDEN: 正常模式显示该目录列表时不显示该文件.
ATTR_SYSTEM: 是系统文件.
ATTR_VOLUME_ID: 在一个 FAT 卷中,只能有一个 “文件”设置此位,并且该文件必须在根目录中,该文件的文件名实际上就是该卷的卷标,并且该文件的 DIR_FstClusHI 和 DIR_FstClusLO 必须为 0(卷标文件不分配空间).
ATTR_DIRECTORY: 目录
ATTR_ARCHIEVE 此属性用于支持一些备份程序,当文件创建,改名或写入时,FAT 文件系统会设置此位,备份程序可以利用此位来判断卷中的哪些程序在上次备份到现在有更改过.

另外 ATTR_LONG_NAME 位实际上表明此 “文件”实际上为另外一个有长文件名的文件的一部分,在下

个章节我们会详细讨论长文件名的情形

创建一个目录时，该“文件” 的 ATTR_DIRECTORY 位被置位同时 DIR_FileSize 被设置为 0，ATTR_DIRECTORY 被置位的文件不使用 DIR_FileSize 域并且该域通常为 0(目录所占空间为其起始簇到 EOC 结束的簇链所占的空间)，每个目录项分配一个簇(除非是 FAT12/16 的根目录)，将 DIR_FstClusLO 和 DIR_FstClusHI 的值设置为该簇的簇号，然后在 FAT 表中为该簇设置一个 EOC 标志，并把该簇的每一个字节设置为 0，如果这是根目录，那么你的工作就完成了（根目录没有‘.’和 ‘.. ’），否则，你必须在该目录空间（就是刚刚分配的那个簇）的头两个 32-bytes 创建两个特殊的目录项。

第一个目录项的 DIR_Name 设置为：

“.　　“

第二个目录项的 DIR_NAME 设置为：

“..　　“

我们称这两个目录为“点”和“点点”，这两个目录的 DIR_FileSize 均设置为 0，同时两个目录的时间和日期标志也设置为以包含它们本身的目录相同，将”.”目录项的 DIR_ClusLO 和 DIR_ClusHI 设置为与它自身所在的目录（包含这两个“.”和“..”目录项的目录）相同。

最后把“..”目录项的 DIR_ClusLO 和 CLUS_ClusHI 设置为与刚刚创建的目录项所在目录的第一个簇号（如果刚创建的目录在根目录则这些值为 0，FAT32 也是如此）。

“.”和”..”的特征可以概括如下：

“.”目录指向该目录本身。

“..”目录指向该目录的上级目录。

时间和日期格式（Date and Time Formats）

很多 FAT 文件系统不支持时间/日期而只支持 DIR_WrtTime 和 DIR_WrtDate，因此，DIR_CrtTimeMil, DIR_CrtTime, DIR_CrtDate 和 DIR_LastAccDate 实际上都是可选的域，只有 DIR_WrtTime 和 DIR_WrtDate 是*必须*的。如果其他的域不支持，那么这些不支持的域必须在文件创建时把它设置为 0，以后文件的读写操作时不能更改这些域。

日期格式. 每个 FAT 目录项的日期由 16-bit 构成，这一时间是相对于 1980 年 1 月 1 号的偏差量，其格式如下：（bit 0 为最低有效位，bit 15 为最高有效位）

Bits 0-4　　日期，可以在 1-31 区间取值。

Bits 5-8　　月份，1 代表一月份，可以在 1-12 区间取值。

Bits 9-15　　年份，从 1980 开始计算，可以在 0-127 区间取值（1980-2107）。

时间格式. 每个 FAT 目录项的时间由 16-bit 构成，每两秒为一个单位，其格式如下：（bit 0 为最低有效位，bit 15 为最高有效位）：

Bits 0-4　　秒，（2 秒为 1 单位），可以在 0-29 区间取值。（0-58 秒）

Bits 5-10　　分钟，可以在 0-59 区间取值。

Bits 11-15　　小时，可以在 0-23 区间取值。

因此可以取得的时间为午夜 00:00:00 到 23:59:58

长目录项（FAT Long Directory Entries）

在 FAT 文件系统中增加长目录项很重要的一点在于它要添加到现有的 FAT 文件系统中：

- 对于早期的 MS-DOS 版本透明。设计长目录项一个很基本的要求是使得在早期的 MS-DOS/Windows 上运行的 API 不能轻易地“找到”长目录项。只有一类 MS-DOS API 可以“找到”长目录项，那就是当配合全字符匹配（比如*.*）和全属性匹配（比如 0xFF）使用时的 FCB-based-find APIs。在 Windows

95 以后的 MS-DOS/Windows 版本，再没有哪个 API 可以“找到”单一的长目录项。

- 长目录项在存储介质上的位置接近和它配对的短目录项。长目录项紧跟在该短目录项后有助于增强了 FAT 文件系统的稳定性。
- 当被低版本的磁盘维护程序检测到时，长目录项的存在并不会影响现有文件的数据完整性。通常磁盘维护程序并不使用 MS-DOS 的 API 来读取存储介质上的数据，而是读取物理或是逻辑的扇区数据并根据自己的程序判断这些目录项都包含些什么内容，检测程序使用试探法（heuristics）来检测磁盘是否有错，并通过不同的步骤来“修复”这些看似被“损坏”的文件。长目录项加入到现有文件系统后必须保证它们被 Windows 95 以前的系统“修复”过后不被损坏。

为了实现*部分可访问性*（locality-of-access）和透明性，我们给短目录项设置特殊的属性，于是它们就成了长目录项。如前面叙述，长目录项的就是拥有以下属性普通的短目录项：

```
ATTR_LONG_NAME          ATTR_READ_ONLY |
                        ATRR_HIDDEN |
                        ATTR_SYSTEM |
                        ATTR_VOLUME_ID
```

长目录名的子项（sub-component）属性如下：

```
ATTR_LONG_NAME_MASK     ATTR_READ_ONLY |
                        ATTR_HIDDEN |
                        ATTR_SYSTEM |
                        ATTR_VOLUME_ID |
                        ATTR_DIRECTORY |
                        ATTR_ARCHIEVE
```

文件系统遇到这样的目录项将会对其特别处理，把它作为是和其配对的短目录项的一部分。每个长目录项由以下结构组成：

FAT 长目录项结构（FAT Long Directory Entry Structure）

名称	Offset (byte)	大小 (byte)	描述
LDIR_Ord	0	1	该长目录项在本组（long dir set）中的序号，如果标记为 0x40(LAST_LONG_ENTRY)则表明是该组的最后一个长目录项项。长目录项必须以 LDIR_Ord 开始。
LDIR_Name1	1	10	长文件名子项的第 1-5 个字符。
LDIR_Attr	11	1	属性必须为 ATTR_LONG_NAME
LDIR_Type	12	1	如果为 0 表明是长文件名的子项。 NOTE: 其他值保留供以后扩展使用,如果非零表明是其他目录类型
LDIR_Chksum	13	1	短文件名的校验和
LDIR_Name2	14	12	长文件名子项的第 6-11 个字符。
LDIR_FstClusLO	26	2	必须为 0，这是 FAT 的“第一个簇”，对于长目录项，此域没有任何意义，只是为了兼容早期的程序。
LDIR_Name3	28	4	长文件名子项的第 12-13 个字符。

长/短目录项的组织与联结（Organization and Association of Short & Long Directory Entries）

每个长目录组（a set of long entries）总是与其紧跟着的短目录相配对，它们相配对的原因是：在低版本的 MS-DOS/Windows 系统中，只有短目录是可见的，没有短目录配对的长目录完全不可见。而失去配

对的短目录，长目录就会成为“孤立”项。以下的图表描述了长/短目录项之间是如何关联的。

长目录项总是紧跟在短目录项后面，并且在物理（介质）上是连续的，文件系统往往会通过一些其他的手段来检测长目录项是否有短目录项与之相配对。

长目录项的序号（Sequence Of Long Directory Entries）

项	序号
第 N 个（最后一个）长目录项	LAST_LONG_ENTRY(0x40) ｜ N
…其他的长目录项	…
第一个长目录项	1
长目录项之前的短目录项	（无）

首先，在一组长目录项中每个目录项的序号是唯一的，并且每组的最后一个目录项使用一个标志来表明这是最后一个长目录单元，LDIR_Ord 域就是用作此目的。第一个目录项中 LDIR_Ord 的值为一，第 N 个目录项的值为 N 与 LAST_LONG_ENTRY 相或的结果(LAST_LONG_ENTRY(0x40) ｜ N)，记住 LDIR_Ord 的值不能为 0x00 和 0xE5,因为这些值常常被文件系统用来表示该目录项是“空的”或是一个簇的“最后”一个目录项。在 LDIR_Ord 的取值范围不包含这两个数值，而必须是从 1 到 N ｜ LAST_LONG_ENTRY.否则这组长目录项就被“损坏”并被文件系统认为是孤立的。

然后，文件系统会在目录建立时计算出一个 8-bit 的校验和，这个校验和由短目录项文件名中 11 个字符计算得出，然后保存在每一个长目录项中，如果保存在长目录项中的校验和与计算结果不同，这个目录项便会被认为是孤立的，如果早期版本的 MS-DOS/Windows 挂上(mount)了含有长目录项的磁盘，那么在改文件名时只有短目录名会被改变，这样便会出现长目录项中的校验和与计算结果不符的现象。

校验和的算法如下，用 C 语言描述：

```
//------------------------------------------------------------------------
// ChkSum ()
// 返回一个字节长度的无符号校验和，此校验和由一无符号字节数组计算得出，此数组长度必须为
// 11，同时假设数组包含的是 MS-DOS 格式的文件名
// Passed: pFcbName      指向长度为 11 的无符号字节数组指针，
// Returns: Sum          由 pFcbName 数组计算得出的 8-bit 无符号校验和
//------------------------------------------------------------------------
unsigned char ChkSum (unsigned char *pFcbName)
{
  short FcbNameLen;
  unsigned char Sum;

  Sum = 0;
  for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {
    // NOTE: 运算为无符号字符循环由移
    Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
  }
  return (Sum);
}
```

最后我们得出这样的结论：短目录项的结构包含最后访问日期，创建日期，第一个簇的簇号和文件.大小，当然，还包括早期的 MS-DOS/Windows 可以识别的短目录名。长目录项不再重复这些可以从短目录项取得的信息，因而有更多的空间去容纳其他信息，这些空间主要作用于存储长目录名。与长目录项配对的短目录项所包含的文件名我们也称作“别名”（alias, alias name）。

长文件名在长目录项中的存储（Storage of a Long-Name Within Long Directory Entries）

一个目录项往往无法容纳长文件名所有的字符，因而需要将长文件名存储在多个目录项中。在任何情况下，这些长目录项都相互独立（disjoint），以下的例子将说明长文件名如何在多个目录项中存储。长文件名以 NUL 字符结尾，剩余的空间由 0xFFFF 填充，这也可用于检测长文件名空间是否被破坏。如果长文件名的字符刚好能够填满目录项（比如 13 的整数倍），那么文件名末尾将不添加 NUL 字符和 0xFFFF。

假如我们创建一个文件名为“The quick brown.fox”的文件，下面举例说明文件名如何存储在多个长目录项中。大多数的文件名都是按以这种方式存储的：

第二个长目录项（最后一项）

42h	w	n	.	f	o	0Fh	00h	chk-sum	x
0000h	FFFFh	FFFFh	FFFFh	FFFFh	0000h	FFFFh			FFFFh

第一个长目录项

01	T	h	e		q	0Fh	00h	chk-sum	u
i	c	k		b	0000h	r			o

短目录项

T	H	E	Q	U	I	~	1	F	0	X	20h	NT	Rsvd	Ctrated time
Created Date		Last Access Date		0000h		Last Modified Time		Last Modified Date		First Cluster		File Size		

短文件名是根据长文件名“自动生成”的，算法在后面章节讨论。

命名规则和字符设置(Name Limits and Character Sets)

短目录项

短目录名由 8 个字符的主文件名和 3 字符的扩展名以及可选的点号(“.”)组成。路径的总长度不能超过 80 个字符（路径 64 字符 + 驱动器名称 3 字符 + 8.3 格式文件名 12 字符 + NUL），构成文件名的字符可以是任何可结合的字母（combination character），数字，或是 ASCII 码中大于 127 的字符，并且允许使用以下的特殊字符：

$ % ' - _ @ ~ ` ! ( ) { } ^ # &

短文件名在文件创建时以系统指定的 OEM 代码页（code page）存储在短目录项中，短目录项依旧保存着 OEM 代码页以兼容早期版本的文件系统。OEM 字符是单字节的 8-bit 字符或是以特定代码页配对的 DBCS（数据库管理系统）字符。

短文件名的每个字符在传送给文件系统之前会先被转换为大写字母，因此它们原来的名字也就丢失，这就带来一个问题：在很多 OEM 的代码页里，小写字母和大写字母并不是一一对应的，也就是说可能有多个小写字母对应同一个大写字母，而文件系统并不保存小写字母所能包含的信息，这使得一些表面看起来似乎不一样的文件名，再转换为大写字母之后却成了同一个名字。

长目录项

长文件名可以允许最多 255 个字符，不包括 NUL 字符。路径的总长度不能超过 260 个字符(包含 NUL)，前文规定组成短文件名的字符都可以用在长文件名当中，并且在长文件名中可以使用多个点号(“.”)，同时空格也可以作为长文件名的一部分。另外以下六个字符只允许在长文件名中使用，在短文件名中是不允许出现的：

+ , ; = [ ]

长文件名的中间部分可以有空格出现，而开头和结尾的空格将被文件系统忽略。

开头和中间部分可以使用点号，而结尾的点号将被忽略。

长文件名以 UNICODE 存储在长目录项中，UNICODE 为 16-bit 字符，UNICODE 字符是无法存储在短

目录项中的，因为短目录项只能存储 8-bit 的 OEM 或 DBCS 字符。

文件系统存储长文件名之前并不需要把小写字母转化为大写，从而保存了文件名原始的信息，UNICODE 避免了一些 OEM 代码页会将多个小写字母对应同一个大写字符而带来的问题。

长/短目录名的匹配(Name Matching In Short & Long Names)

所有短目录项中的短文件名被统称为“短文件名空间”(short name space)，而所有长目录项中包含的长文件名统称为“长文件名空间”(long name space)，它们共同组成统一的命名空间，在这个空间中不存在重复的文件名，那就是说：在同一个目录中，无论是长文件名还是短文件名，一个文件名只能出现一次。具体点说，虽然长文件名保存了文件名的大小写信息，但不允许两个文件使用相同的文件名，虽然这两个文件保存在磁盘介质上的名称是不一样的。如果当前目录中存在一个短目录名为“FOOBAR”或是长文件名为“FooBar”的文件，您将无法再创建一个名为“foobar”的文件。

文件系统中的所有搜索操作（比如查找，打开，创建，删除，改名）都是不区分大小写的（case-insensitive），打开文件“FOOBAR”的操作将打开“FooBar”或“foobar”，如果它们中有一个存在的话，如果用“FOOBAR”作为匹配模式来查找的话，上述的两个文件将同样被查找到。这一规则同样适用于那些扩展字符。

短文件名搜索只检查短文件名部分，而长文件名搜索将检查长文件名和短文件名两部分。在遍历目录时，一旦短目录项在缓存中遇到以之配对的长文件名，文件系统便会把长目录项的文件名子部放入到缓存中，长文件名操作将会先查找缓存中的长文件名然后在查找短文件名。

无论是以 OEM 字符还是 UNICODE 字符存储，当存储在介质中的字符无法被转换为合适的 OEM 或 ANSI 代码页时，它们都将被“翻译”成字符“_”（下画线）返回给用户——但存储介质中的字符并没有改变。在所有的 OEM 代码页和 ANSI 中，这个字符都是相同的。

文件名转换以及长文件名（Naming Conventions and Long Names）

API 允许调用者给文件或目录命名，但它不允许调用者只改变文件的短文件名，这是因为文件系统由长文件名和短文件名共同组成统一的命名空间，可见文件系统的命名空间不支持重复的文件名。也就是说，忽略大小写，一个文件的长文件名不能和另外一个文件的短文件名相同，这一限制是为避免用户或应用程序把这些文件混淆。为了使这一特性对用户透明，当创建了一个唯一的长文件名之后，系统将自动创建一个唯一的短文件名。

根据长文件名来自动产生短文件名的技术是在 Windows NT 之后才有的，自动生成的短文件名分为*基本名（basis-name）*和可选的*数字结尾(numeric-tail)*

（注：8.3 格式为 主文件名(8).扩展名(3)，主文件名由基本名和数字结尾两部分共同组成）

基本名的生成算法（The Basis-Name Generation Algorithm）

生成基本名的算法可概述如下，大体描述了短文件名是如何根据长文件名自动生成的，它因该按以下步骤执行：

1. 把 UNICODE 转换为大写之后传送给文件系统
2. 将大写的 UNICODE 转换为 OEM 代码页

```
if (OEM 代码页中不存在转换为大写字母后的 UNICODE 字符)  或
     (OEM 字符不符合 8.3 命名规范)
{
     将该字符转换为 OEM 字符‘_’（下画线）。
     设置“lossy conversion”标记。
}
```

3. 去掉长文件名开头和中间的空格。
4. 去掉长文件名开头的点。
5. while（不是长文件名的结尾）&&（该字符不是点）
   （字符总数小于 8）
   {
   将字符复制到基本名的主文件名部分。
   }
6. 如果在长文件名的最后一个点后面还有扩展名，则在基本名的主文件名后面插入“.”
7. 扫描长文件名中的最后一个点
   if （ 找到在文件名当中的最后一个点）
   {
   while（ 不是长文件名的结尾）
   &&（字符总数小于 3）
   {
   将字符拷贝到基本名的扩展名部分
   }
   }

数字结尾的生成算法（The Numeric-Tail Generation Algorithm）

If （没有设置“lossy conversion”标志）
&&（长文件名符合 8.3 格式名字转换）
&&（基本名不与其他现有的短文件名冲突）
{
短文件名只有基本名而没有数字结尾。
}
else
{
在基本名的主文件名结尾部分插入~n，这个 n 的选择必须使得文件名不与现有的短文件名冲突，并且主文件名的字符不能超过 8 个字符。
}

“~n”的取值范围在“~1”到“~999999”之间，“n”的取值为具有相同 basis-name 的文件系列的序列号的下一个号码，比如：存在以下几个短文件名：LETTER~1.DOC 和 LETTER~2.DOC,那么下一个文件的名字就是 LETTER~3.DOC.现假设存在这两个文件名：LETTER~1.DOC 和 LETTER~3.DOC,那么，下一个文件的名字将为 LETTER~2.DOC。然而程序并不是绝对的按这种方式来计算 n 值，当一个目录中包含大量的类似这样混杂的文件名时，程序会使用另外一种更快速的算法来计算 n 值，n 的数值可能和前面算法的结果不一样，但保证基本名一样拥有相同的开头并与“~n”结尾。

长目录项在低版本 FAT 中的状况（Effect of Long Directory Entries on Down Level Versions of FAT）

长文件名主要用于硬盘，但它同样也支持移动存储，长文件名的引入并不影响对现有文件系统的兼容性。低版本的文件系统**读取**数据时不会有任何兼容性的问题。在使用长文件名之前现有的数据不需要进行

任何的转换，所有的文件都将原样保留。只有在创建长文件名时才会创建一个长目录项。在相邻的目录空间不可用的情况下，要给现有的文件增加一个长文件名则必须把该目录项移到别处去。

长目录项在低版本系统上是隐藏的，就如它的系统文件一样，这足以避免用户一些误操作带来的麻烦，用户可以用 8.3 格式文件名将文件拷贝到别的地方，在原地方存放新的文件不会有任何的不妥。

有趣的地方在于当磁盘放到低版本的文件系统中，并将原文件改变的情况。这很有可能会影响到长文件名，因为低版本的文件系统会忽略长目录项。同时也无法保证长文件名和 8.3 格式的短文件名相配对。

低版本的文件系统只有在搜索卷标时才会找到长目录项，在低版本的文件系统中，如果根目录中卷标文件的位置不是在所有的长目录项之前，那么显示的卷标将会不正确，这是因为长目录项也设置了卷标位，这确实糟糕，还好这不是一个很严重的问题。

如果用户试图删除卷标，那么这很可能删除一个有长文件名的文件，发生这种情况的可能比较小，并且这很容易由系统判断出来。被删除的长文件名将不再有效，因为其中一个或多个长目录项被标记为已经删除。如果被删除的目录项被重新使用的话，那么长目录项属性位的设置将是错误的。

在低版本的系统中重命名一个文件，只有短文件名会被修改。命名空间中长短文件名必须保持一致，因此长文件名不可见也算是见好事。长目录项中的校验和可以用来检测此类型的改动，长文件名只有再验证无误的情况下才会被使用。如果更改过的文件名计算出来的校验和刚好和以前的相等，这将会产生错误，因为检验和算法并没有跨越短文件名命名空间。

更改 8.3 格式的短文件名同样不能和其他文件的长文件名重复，否则在忽略大小写的情况下，低版本的文件系统便可能创建一个以其他文件的长文件名相同的短文件名，为避免这种情况发生，自动生成 8.3 格式短文件名的程序将直接把长文件名转化为大写后映射成 8.3 格式文件名。

如果一个文件被删除，那么其长目录项变成孤立项，如果此时创建一个新文件，则长文件名将被错误地以此文件配对，以前面文件重命名时讨论的情况相同，由 8.3 文件名计算得到的校验和将避免错误的发生。

确认目录的内容（Validating The Contents of a Directory）

由于 FAT 目录结构中某些位保留供以后版本使用，本节内容主要帮助磁盘维护程序在处理这些保留位时，如何检查文件的“正确性”。

1. *不要*试图去查看标有“reserved”标志的区域，同时我们假定：如果这些区域的值不为 0，那么它们就是“坏的”
2. 如果标志“reserved”区域的值不为 0（我们已经假设它们被“损坏”），*不要*将这些区域的值设置为 0，这些区域是设计为*保留*而不是*取值必须为 0*，这些区域是为以后的文件系统扩展而保留的，检测程序应该忽略它们，这样的程序在后期版本的文件系统中也能正确的运行。
3. 在检测一个目录项是长目录项还是短目录项时，首先应该使用 A_LONG 属性位来判定。以下是判断长/短目录项的一个正确算法：

```
if ((( LDIR_Attr & ATTR_LONG_NAME_MASK) == ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
      /* 找到一个活动的长文件名子部(long name sub-component) */
}
```

4. 同时使用短目录项的第 3 和第 4 为来判定短目录项的类型，以下是判定目录类型的一个正确算法：

```
if ((( LDIR_Attr & ATTR_LONG_NAME_MASK) != ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
      if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == 0x00)
          /* 找到一个文件 */
      else if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == ATTR_DIRECTORY)
```

```
            /* 找到一个目录 */
        else if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == ATTR_VOLUME_ID)
            /* 找到一个一个卷标 */
        else
            /* 找到一个非法的目录项 */
    }
```

5. 不要认为文件“类型”区域的值非 0 的目录项就是坏的目录项，也不要强行把“类型”区域的值设置为 0。
6. 记得使用长目录项的“校验和”。“起始簇”的值设置为 0 是正确的，虽然这在以后可能会被更改。

FAT 目录应该注意的其他地方(Other Notes Relating to FAT Directories)

- 长目录项在所有类型的 FAT 中都是相同的。详情请参阅前面章节。
- DIR_FileSize 是一个 32-bit 的区域,FAT 驱动程序一定不能创建一个大于 0x100000000 的簇链，这么长的簇链将使最后一个簇的最后一个字节无法分配给文件,也就是说文件的长度不能大于 0xFFFFFFFF,这是所有 FAT 文件系统一个基本的限定：FAT 卷中文件大小不能超过 0xFFFFFFFF(4,294,967,295)字节。
- 同样，FAT 驱动程序不允许一个目录（作为其他文件容器的文件）的大小超过 65536 * 32(2,097,152)字节。

NOTE: 目录中文件的个数并不受限制，目录的大小受其容量的影响而与目录里的内容无任何关系，这主要由于两个原因：

1. FAT 目录没有排序和索引，因而在同一目录下放置大量文件的做法是极其愚蠢的。否则类似创建新目录的操作（需要在现有的文件中查找以确认新创建的文件名不会重复）将变得非常缓慢。
2. 包括 Microsoft 的在内，有很多的应用程序都将文件数放在一个长度为 16-bit 的双字变量中，因此一个目录中的文件数不能大于 16-bit 所能容纳的值（65535）。

Khalai

2004. 9 于 夏新