NTFS文件系统若干技术研究

作者： Beiyu
主页： http://beiyu.bokee.com
电子邮箱： beiyuly@gmail.com
日期： 2007-4-26

Research on Some Technique of NTFS

Author: Beiyu
Home Page: http://beiyu.bokee.com
Email: beiyuly@gmail.com
Date: 2007-4-26

目录

NTFS文件系统若干技术研究

RESEARCH ON SOME TECHNIQUE OF NTFS

目录

表目录

图目录

摘 要

第一章 NTFS介绍

1.1现状
1.2 NTFS特点
1.2.1优点
1.2.2 NTFS的不足
1.3 NTFS未来

第二章 NTFS相关概念理论

2.1 RAID 简介
2.2 NTFS文件系统分区
2.2.1 基本分区
2.2.2 动态分区
2.3 NTFS参数结构介绍
2.3.1 NTFS的DBR
2.3.2 NTFS文件空间分配
2.3.3 NTFS元文件
2.3.4 元文件和DBR参数关系
2.3.5 NTFS的文件和目录
2.3.6 常驻属性与非常驻属性
2.3.7 MFT文件记录结构分析
2.3.7.1基本概念
2.3.7.2文件记录头分析
2.3.7.3标准属性分析
2.3.7.4文件名属性分析
2.3.7.5数据流属性分析
2.3.7.6 位图属性分析
2.3.7.7 $MFT结构总结
2.3.8 $Boot元文件介绍
2.3.9 NTFS索引与目录
2.4 EFS

第三章 NTFS文件基本操作

第四章 NTFS文件系统数据恢复研究

4.1数据恢复原理

4.2常见数据恢复工具
4.3 NTFS数据技术基础
4.3.1基础概念
4.3.1.1硬盘数据结构
4.3.1.2硬盘分区方式
4.3.1.3数据存储原理
4.3.1.4系统启动流程
4.3.2硬盘数据恢复方案分析
4.4数据恢复技术实现
4.5数据恢复经验介绍
4.5.1几个常识性问题
4.5.2技术来源
4.5.3硬盘修复需要理解的基本概念
4.6数据备份介绍

第五章 NTFS相关领域技术介绍

5.1 RH8下最简单编译NTFS模块的方法
5.2 UBUNTU下安全读写NTFS分区格式文件
5.3在FAT32中读写NTFS分区的数据
5.4 DOS下访问NTFS

第六章 结论

参考文献

附录A DOS下访问NTFS分区，查找指定文件的源代码

附录B WINDOWS下NTFS文件恢复源代码

后记

表目录

表格_一-2文件系统比较
表格_一-1NTFS文件系统组织
表格_二-1 BPB参数
表格_二-2 NTFS的缺省簇的大小
表格_二-3从头部开始的偏移长度描述
表格_二-4常驻属性从头部开始的偏移描述
表格_二-5非常驻属性从头部开始的偏移描述
表格_二-6标准索引头的结构
表格_二-7常用索引列表

图目录

图表_一-1操作系统和文件系统支持表
图表_一-2NTFS文件属性示意图
图表_二-1RAID性能比较
图表_二-2 NTFS目录
图表_二-3 NTFS元文件
图表_二-4 MFT空间分配
图表_二-5 NTFS DBR结构
图表_二-6 NTFS BPB结构
图表_二-7 $MFT和DBR关系
图表_二-8 小文件和目录的MFT
图表_二-9 $MFT存储情况
图表_二-10 NTFS常用属性
图表_二-11 小文件示意图1
图表_二-12 小文件示意图2
图表_二-13 小文件的文件记录
图表_二-14 小目录的MFT记录
图表_二-15 存储在两个运行中的非常驻属性
图表_二-16 大目录的MFT记录
图表_二-17 非常驻数据属性的VCN
图表_二-18 非常驻数据属性的VCN-LCN映射
图表_二-19 $MFT文件记录头部结构
图表_二-20标准属性头结构
图表_二-21标准属性的属性结构
图表_二-22 文件属性的含义
图表_二-23文件名属性头结构
图表_二-24文件名属性结构
图表_二-25标志含义图
图表_二-26常见的命名空间
图表_二-27数据流属性头结构
图表_二-28位图属性头
图表_二-29 $MFT的结构示意图
图表_二-30 $Boot元文件属性
图表_二-31 $Boot元文件未命名数据流含义
图表_二-32根目录文件索引
图表_二-33索引项结构示意图
图表_二-34 EFS加密过程
图表_二-35 EFS解密过程

摘 要

随着以NT为内核的Windows 2000/XP的普及，很多个人用户开始用到了NTFS。因此NTFS受到了越来越多的重视。

本文主要介绍NTFS的基本概念，基于NTFS的文件基本操作，NTFS文件数据恢复，NTFS相关领域技术。

关键字：NTFS，文件操作，数据恢复，Windows，DOS

第一章 NTFS介绍

NTFS（New Technology File System）是一个比FAT复杂的多的文件系统，微软Windows NT内核的系列操作系统支持的、一个特别为网络和磁盘配额、文件加密等管理安全特性设计的磁盘格式。

1.1现状

随着以NT为内核的Windows 2000/XP的普及，很多个人用户开始用到了NTFS。NTFS也是以簇为单位来存储数据文件，但NTFS中簇的大小并不依赖于磁盘或分区的大小。簇尺寸的缩小不但降低了磁盘空间的浪费，还减少了产生磁盘碎片的可能。NTFS支持文件加密管理功能，可为用户提供更高层次的安全保证。

Windows NT/2000/XP/2003以上的Windows版本能识别NTFS系统，Windows 9x/Me以及DOS等操作系统都不能直接支持、识别NTFS格式的磁盘，访问NTFS文件系统时需要依靠特殊工具。

	FAT12	FAT16	FAT32	NTFS	WinFS
DOS3.0 以前	√	x	x	x	x
Windows 95 OSR2 之前	√	√	x	x	x
Windows NT	√	√	x	√	x
Windows 2000/XP/2003	√	√	√	√	x
Windows Longhorn	√	√	√	√	√

图表 一-1操作系统和文件系统支持表

下面对目前Windows常用的文件系统作一个比较：

表格 一-2文件系统比较

比较标准	NTFS5	NTFS	FAT32	FAT16
适用操作系统	Windows 2000、XP	Windows NT、2000、XP	Windows 98、Me、2000、XP	DOS、Windows所有版本
磁盘管理限制				
最大分区尺寸	2TB	2TB	理论上2TB	2GB
分区中做多文件数目	接近无限	接近无限	接近无限	小于65000
最大文件尺寸	决定于文件存放分区的尺寸	决定于文件存放分区的尺寸	4GB	2GB
最大簇数	几乎无限	几乎无限	268435456	65535
最大文件名长度	255个英文字符	255个英文字符	255个英文字符	8.3文件名标准，VFAT中为

				255个英文字符
文件系统特征				
Unicode（统一代码）文件命名	由Unicode字符设定	由Unicode字符设定	由系统字符设定	由系统字符设定
系统记录镜像	MFT镜像文件	MFT镜像文件	FAT文件表的第二拷贝	FAT文件表的第二拷贝
引导扇区位置	第一和最后的扇区	第一和最后的扇区	第一扇区	第一扇区
文件属性	标准属性加自定义属性	标准属性加自定义属性	标准属性设定	标准属性设定
交替数据流支持	支持	支持	不支持	不支持
数据压缩支持	支持	支持	不支持	不支持
数据加密支持	支持	不支持	不支持	不支持
对象标识支持	支持	支持	不支持	不支持
磁盘配额支持	支持	不支持	不支持	不支持
稀疏文件支持	支持	不支持	不支持	不支持
重装入点支持	支持	不支持	不支持	不支持
卷装入点支持	支持	不支持	不支持	不支持
性能概述				
安全性构造	支持	支持	不支持	不支持
可恢复性	支持	支持	不支持	不支持
性能	在大分区上（>500MB）性能很高但在有非常多小文件的小分区上性能较低	在大分区上（>500MB）性能很高但在有非常多小文件的小分区上性能较低	在小分区上（<500MB）性能较低	在小分区上（<256MB）性能最高
磁盘空间节约	最大	最大	一般	在大分区上最低
磁盘数据容错性	最大	最大	最低	一般

注意：

1、交替数据流、对象标识、稀疏文件、重装入点和卷装入点的概念非常复杂，而且和普通用户的应用关系不大，列出来只是为了对比特性，故不做进一步解释。

2、表中对大、小分区容量的界定只是一个理论值，实际的分界点要高一些。

Windows 2000/XP提供了分区格式转换工具“Convert.exe”。Convert.exe是Windows 2000附带的一个DOS命令行程序，通过这个工具可以直接在不破坏FAT文件系统的前提下，将FAT转换为NTFS。它的用法很简单，先在Windows 2000环境下切换到DOS命令行窗口，在提示符下键入：D:\>convert 需要转换的盘符 /FS:NTFS。如系统E盘原来为FAT16/32，现在 需

要转换为NTFS，可使用如下格式：D:\>convert e: /FS:NTFS。所有的转换将在系统重新启动后完成。

NTFS支持磁盘数据加密，加密文件系统（Encrpyted File Sysetm，EFS）是NTFS支持的重要特性。

由于NTFS文件分区格式具有良好的安全性，如果你不希望自己在硬盘中的文件被其他人调用或查看，使用权限控制方式加密是非常有效的方法。设置方法非常简单：以系统管理员身份登录，使用鼠标右键单击需要加密的文件夹，选择“Properties”，切换到“Security”选项卡。在“Group of user names”项中设置允许访问的用户只有Administrator和自己。删除其他的所有用户。保存设置退出即可。此后，其他用户将不能访问该文件夹。使用这项功能需要注意的是：一定要保证只有你一个人知道Administrator密码，并且设置其他用户不能属于Administrator。此外，你还可以详细的给每个用户设置权限，包括设置读取权限、写入权限、删除权限等，这样使用起来就更加灵活。你还可以设置权限，控制一个磁盘，或者磁盘分区只为自己使用，这样其他人就不能看到你的任何东西了。

表格 一-1NTFS文件系统组织

分区引导扇区	MFT表	系统文件	文件区域

NTFS的索引非常详细,尽管很利于查找文件，但相当于小型数据库的索引方式对硬件有较高的硬件要求，而且对于较小的分区上存放较多小文件的情况而言，这种检索方式可能反而没有简单的链式快。最常见的情况就是笔记本用户，特别是前两年，笔记本硬盘的速度很慢，硬盘容量又小，而笔记本的性能又普遍低于台式机很多，所以这种情况下如果采用NTFS分区就可能会感觉慢。笔者个人认为，要想体现NTFS分区的性能优势，至少和FAT持平的话，电脑要有如下水准：硬盘的转速最好为7200r/s, CPU主频不低于700MHz，内存不少于256MB，单个分区不小于5GB。前两年有不少台式机都有某些指标不能达到这个水平，难怪有人抱怨NTFS不好了。但就目前的硬件水平而言，NTFS的优势会越来越明显。

1.2 NTFS特点

1.2.1优点

NTFS的四大优点：

1、具备错误预警的文件系统

在NTFS分区中，最开始的16个扇区是分区引导扇区，其中保存着分区引导代码，接着就是主文件表(Master File Table，以下简称MFT)，但如果它所在的磁盘扇区恰好出现损坏，NTFS文件系统会比较智能地将MFT换到硬盘的其他扇区，保证了文件系统的正常使用，也就是保证了Windows的正常运行。而以前的FAT16和FAT32的FAT(文件分配表)则只能固定在分区引导扇区的后面，一旦遇到扇区损坏，那么整个文件系统就要瘫痪。

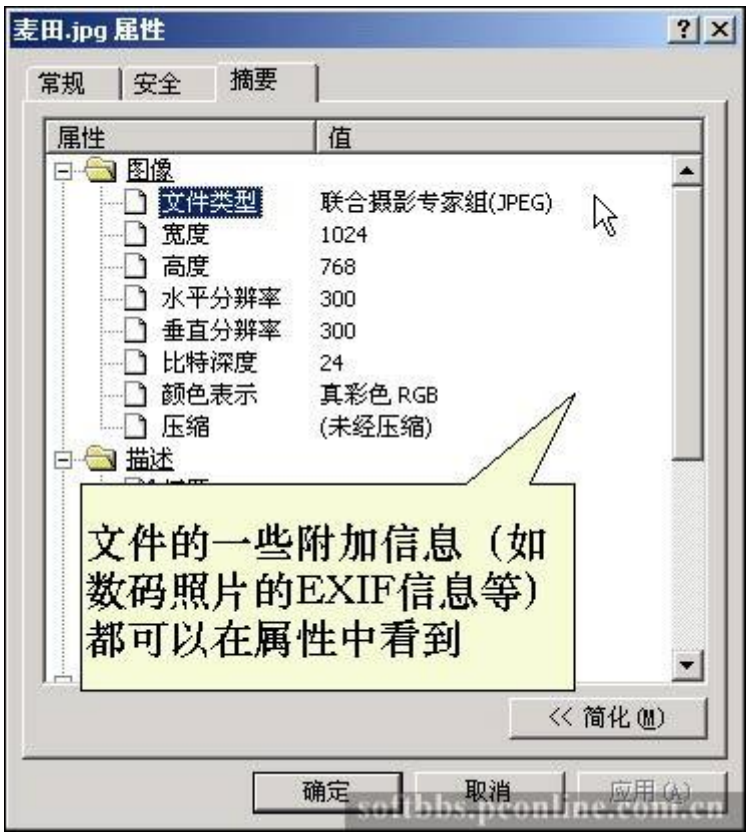
但这种智能移动MFT的做法当然并非十全十美，如果分区引导代码中指向MFT的部分出现错误，那么NTFS文件系统便会不知道到哪里寻找MFT，从而会报告“磁盘没有格式化”这样的错误信息。为了避免这样的问题发生，分区引导代码中会包含一段校验程序，专门负责侦错。

2、文件读取速度更高效!

恐怕很多人都听说NTFS文件系统在安全性方面有很多新功能，但你可否知道:NTFS在文件处理速度上也比FAT32大有提升呢?

对DOS略知一二的读者一定熟悉文件的各种属性:只读、隐藏、系统等。在NTFS文件系统

中，这些属性都还存在，但有了很大不同。在这里，一切东西都是一种属性，就连文件内容也是一种属性。这些属性的列表不是固定的，可以随时增加，这也就是为什么你会在NTFS分区上看到文件有更多的属性，如下图所示：



图表 一-2NTFS文件属性示意图

NTFS文件系统中的文件属性可以分成两种:常驻属性和非常驻属性，常驻属性直接保存在MFT中，像文件名和相关时间信息(例如创建时间、修改时间等)永远属于常驻属性，非常驻属性则保存在MFT之外，但会使用一种复杂的索引方式来进行指示。如果文件或文件夹小于1500字节(其实我们的电脑中有相当多这样大小的文件或文件夹)，那么它们的所有属性，包括内容都会常驻在MFT中，而MFT是Windows一启动就会载入到内存中的，这样当你查看这些文件或文件夹时，其实它们的内容早已在缓存中了，自然大大提高了文件和文件夹的访问速度。

小提示！

为什么FAT的效率不如NTFS高？

FAT文件系统的文件分配表只能列出了每个文件的名称及起始簇，并没有说明这个文件是否存在，而需要通过其所在文件夹的记录来判断，而文件夹入口又包含在文件分配表的索引中。因此在访问文件时，首先要读取文件分配表来确定文件已经存在，然后再次读取文件分配表找到文件的首簇，接着通过链式的检索找到文件所有的存放簇，最终确定后才可以访问。

3、磁盘自我修复功能

NTFS利用一种“自我疗伤”的系统，可以对硬盘上的逻辑错误和物理错误进行自动侦测和修复。在FAT16和FAT32时代，我们需要借助Scandisk这个程序来标记磁盘上的坏扇区，但当发现错误时，数据往往已经被写在了坏的扇区上了，损失已经造成。

NTFS文件系统则不然，每次读写时，它都会检查扇区正确与否。当读取时发现错误，NTFS会报告这个错误;当向磁盘写文件时发现错误，NTFS将会十分智能地换一个完好位置存储数据，操作不会受到任何影响。在这两种情况下，NTFS都会在坏扇区上作标记，以防今后被使用。这种工作模式可以使磁盘错误可以较早地被发现，避免灾难性的事故发生。

4、“防灾赈灾”的事件日志功能

在NTFS文件系统中，任何操作都可以被看成是一个“事件”。比如将一个文件从C盘复制到D盘，整个复制过程就是一个事件。事件日志一直监督着整个操作，当它在目标地——D盘发现了完整文件，就会记录下一个“已完成”的标记。假如复制中途断电，事件日志中就不会记录“已完成”，NTFS可以在来电后重新完成刚才的事件。事件日志的作用不在于它能挽回损失，而在于它监督所有事件，从而让系统永远知道完成了哪些任务，那些任务还没有完成，保证系统不会因为断电等突发事件发生紊乱，最大程度降低了破坏性。

5、附加功能

其实，NTFS还提供了磁盘压缩、数据加密、磁盘配额(在“我的电脑”中右击分区并并行“属性”，进入“配额”选项卡即可设置)、动态磁盘管理等功能，这些功能在很多报刊杂志上介绍的比较多了，这里不再详细介绍。

NTFS提供了为不同用户设置不同访问控制、隐私和安全管理功能。如果你的系统处于一个单机环境，比如家用电脑，那么这些功能对你意义不是很大。

小提示！

从FAT转换过来的NTFS，性能有折扣

如果分区是从FAT32转换为NTFS文件系统的(使用命令为“CONVERT 驱动器盘符 /FS:NTFS”)，不仅MFT会很容易出现磁盘碎片，更糟糕的是，磁盘碎片整理工具往往不能整理这各分区中的MFT，严重影响系统性能。因此，建议将分区直接格式化为NTFS文件系统

1.2.2 NTFS的不足

知道了FAT和NTFS的组织结构后，很容易理解正是因为不同的组织管理方式，导致了完全不同的文件操作方式，从而决定了最终的性能差异。例如当访问一个文件时，由于FAT文件系统的文件分配表只列出了每个文件的名称及起始簇，并没有说明这个文件是否存在，而需要通过其所在文件夹的记录来判断，而文件夹的入口又包含在文件分配表的索引中。因此首先要读取文件分配表来确定文件已经存在，然后再次读取文件分配表找到文件的首簇，最后通过链式的检索找到文件所有的存放簇，最终确定后才可以访问。而在NTFS文件系统中,由于任意文件都在MFT中有详细的记录，所以只要读取MFT中的相关记录文件马上可以使用。这就是NTFS在文件操作性能上要好于FAT的根本原因。事实上， NTFS不断引入的新特性，如综合索引等，仍在进一步拉大这种差距。从技术角度来说这个差距是巨大的，但为何有些人无法感觉到这种差异甚至有相反的感觉呢？

原因也很简单，绝大多数人的计算机应用环境简单，其磁盘操作主要是单文件的操作。再加上很多人的硬盘远不够大，并发操作量也远不够多，因此NTFS与FAT的单个文件操作速度差异往往仅在毫秒之间．根本无法察觉，但如果是在大型的网络服务器上，访问量很多，文件操作频繁，还存在大量并发操作，这种性能差异就会相当惊人。另一方面随着100GB以上容量的硬盘逐渐成为主流，再加上个人使用电脑同时执行多个任务的机会不断增加，这种差距也会越来越明显。那么觉得 NTFS 文件系统更慢的原因是什么呢?

NTFS的索引非常详细,尽管很利于查找文件，但相当于小型数据库的索引方式对硬件有较高的硬件要求，而且对于较小的分区上存放较多小文件的情况而言，这种检索方式可能反而没有简单的链式快。最常见的情况就是笔记本用户，特别是前两年，笔记本硬盘的速度很慢，硬盘容量又小，而笔记本的性能又普遍低于台式机很多，所以这种情况下如果采用NTFS分区就可能会感觉慢。笔者个人认为，要想体现NTFS分区的性能优势，至少和FAT持平的话，电脑要有如下水准：硬盘的转速最好为7200r/s, CPU主频不低于700MHz，内存不少于

256MB，单个分区不小于5GB。前两年有不少台式机都有某些指标不能达到这个水平，难怪有人抱怨NTFS不好了。但就目前的硬件水平而言，NTFS的优势会越来越明显。

1.3 NTFS未来

无论是FAT还是NTFS ,用户访问某个文件都必须通过层次型的目录树结构到达其保存位置，如果不知道文件保存位置，那就只能使用操作系统的搜索功能了。不知道你有没有想过：我们需要的是文件，而不是它的位置，即然如此，为什么要我们必须记住这些“无用”的信息？为解决这个问题，微软将在下一代操作系统中（内部代号Longhorn）中推出传说中的WinFS (windows Future Storage , windwos未来存储）服务，也就是被人们误以为是文件系统的那个东西。

事实上，NTFS将是“Longhorn”的唯一文件系统，WinFS服务可以看作是在NTFS的基础上增加了一个数据库层，这个数据库层以即将出现的SQL Server的“Yukon”版为基础。对于WinFS来说，文件除了我们熟悉的属性，诸如文件名称、大小、日期外，还将通过诸如作者名、图像大小之类的元数据建立索引。系统底层的目录结构仍将存在，但用户使用的将是一个相似文件构成的库（Library）。每个库由一组通过查询WinFS数据库获得的文件构成。如果WinFS确实能将查找文件的依据改变为它是什么，而不是它在哪里的话，无疑这将是基于NTFS文件系统的一个突破。

第二章 NTFS相关概念理论

2.1 RAID 简介

RAID（Redundent Array of Inexpensive Disks）由美国加州伯克利分校的D.A.Patterson教授在1988年提出。

RAID 级	RAID 0	RAID 1	RAID 3	RAID 5	RAID 10	RAID 30	RAID 50
别名	条带	镜像	专用奇偶位条带	分布奇偶位条带	镜像阵列条带	专用奇偶阵列条带	分布奇偶阵列条带
容错性	没有	有	有	有	有	有	有
冗余类型	没有	复制	奇偶位	奇偶位	复制	奇偶位	奇偶位
热备盘选项	没有	有	有	有	有	有	有
需要的磁盘数	一个或多个	只需 2 个	三个或更多	三个或更多	只需 4 个	6、8、10、12、14、16	6、8、10、12、14、16
可用容量	磁盘总容量	磁盘总容量的 50%	（N-1）/N 的磁盘容量	（N-1）/N 的磁盘容量	磁盘总容量的 50%	（N-2）/N 的磁盘容量	（N-2）/N 的磁盘容量

图表 二-1RAID性能比较

WindowsNT/2000XP/2003种的RAID5是一种软RAID。

2.2 NTFS文件系统分区

2.2.1 基本分区

2.2.2 动态分区

2.3 NTFS参数结构介绍

本节主要介绍NTFS的参数概念。

2.3.1 NTFS的DBR

NTFS的DBR和FAT32的DBR作用相同，由MBR得到DBR，再由DBR引导操作系统。在Windows NT/2000/XP/2003上，由DBR调入NTLDR，由NTLDR启动操作系统。

NTFS的引导扇区完成引导和定义分区参数。FAT分区中，即使文件不正确，而BOOT记录正常，分区会显示没有错误。和FAT分区不同，而NTFS分区的BOOT记录不是分区正确与否的充分条件。因为必须MFT中的系统记录（如$MFT等）正常，该分区才能正常访问。NTFS的BPB参数如下表所示：

表格 二-1 BPB参数

字节偏移	长度	常用值	意义
0x0B	字	0x0002	每扇区字节数
0x0D	字节	0x08	每簇扇区数
0x0E	字	0x0000	保留扇区
0x10	3字节	0x000000	总为0
0x13	字	0x0000	NTFS未使用，为0
0x15	字节	0xF8	介质描述
0x16	字	0x0000	总为0
0x18	字	0x3F00	每磁盘扇区数
0x1A	字	0xFF00	磁头数
0x1C	双字	0x3F000000	隐含扇区
0x20	双字	0x00000000	NTFS未使用，为0
0x28	8字节	0x4AF57F0000000000	扇区总数
0x30	8字节	0x0400000000000000	$MFT的逻辑簇号
0x38	8字节	0x54FF070000000000	$MFTMirr的逻辑簇号
0x40	双字	0xF6000000	每MFT记录簇数
0x44	双字	0x01000000	每索引簇数
0x48	8字节	0x14A51B74C91B741C	卷标
0x50	双字	0x00000000	检验和

MFT中的文件记录大小一般是固定的，不管簇的大小是多少，均为1KB。文件记录在MFT文件记录数组中物理上是连续的，且从0开始编号，所以，NTFS是预定义文件系统。MFT仅供系统本身组织、架构文件系统使用，这在NTFS中称为元数据（metadata，是存储在卷上支持文件系统格式管理的数据。它不能被应用程序访问，只能为系统提供服务）。其中最基本的前16个记录是操作系统使用的非常重要的元数据文件。这些元数据文件的名字都以“$”开始，所以是隐藏文件，在Windows 2000/XP中不能使用dir命令（甚至加上/ah参数）像普通文件一样列出。在WINHEX中带有NFI.EXE，用此工具可以显示这些记录与文件的对应关系，下一次再详细解释。

这些元数据文件是系统驱动程序管理卷所必需的，Windows 2000/XP给每个分区赋予一个盘符并不表示该分区包含有Windows 2000/XP可以识别的文件系统格式。如果主文件表损坏，那么该分区在Windows 2000/XP下是无法读取的。为了使该分区能够在Windows 2000/XP下能被识别，就必须首先建立Windows 2000/XP可以识别的文件系统格式即主文件表，这个过程可通过高级格式化该分区来完成。Windows以簇号来定位文件在磁盘上的存储位置，在FAT格式的文件系统中，有关簇号的指针包含在FAT表中，在NTFS中，有关簇号的指针则包

含在$MFT及$MFTMirr文件中。

NTFS使用逻辑簇号（Logical Cluster Number，LCN）和虚拟簇号（Virtual Cluster Number，VCN）来对簇进行定位。LCN是对整个卷中所有的簇从头到尾所进行的简单编号。用卷因子乘以LCN，NTFS就能够得到卷上的物理字节偏移量，从而得到物理磁盘地址。VCN则是对属于特定文件的簇从头到尾进行编号，以便于引用文件中的数据。VCN可以映射成LCN，而不必要求在物理上连续。

在NTFS卷上，跟随在BPB后的数据字段形成一个扩展BPB。这些字段中的数据使得Ntldr能够在启动过程中找到主文件表MFT（Master File Tabl ）。在NTFS卷上，MFT并不象在FAT 16卷和FAT 32卷上一样，被放在一个预定义的扇区中。由于这个原因，如果在MTF的正常位置中有坏扇区的话，就可以把MFT移到别的位置。但是，如果该数据被破坏，就找不到MFT的位置，Windows 2000假设该卷没有被格式化。

因此，如果一个ntfs的卷提示未格式化，可能并未破坏MFT,依据BPB的各字段的意思是可以重建BPB的。

2.3.2 NTFS文件空间分配

NTFS文件按照簇分配。簇的大小必须是物理扇区整数倍，而且总是2的n次幂。

表格 二-2 NTFS的缺省簇的大小

卷大小	每簇的扇区	缺省的簇大小
小于等于512MB	1	512字节
513MB~1024MB(1GB)	2	1024字节(1KB)
1025MB~2048MB(2GB)	4	2048字节(2KB)
大于等于2049MB	8	4KB

从上面可以看出，也就是说不管驱动器多大 NTFS 簇的大小不会超过 4KB。

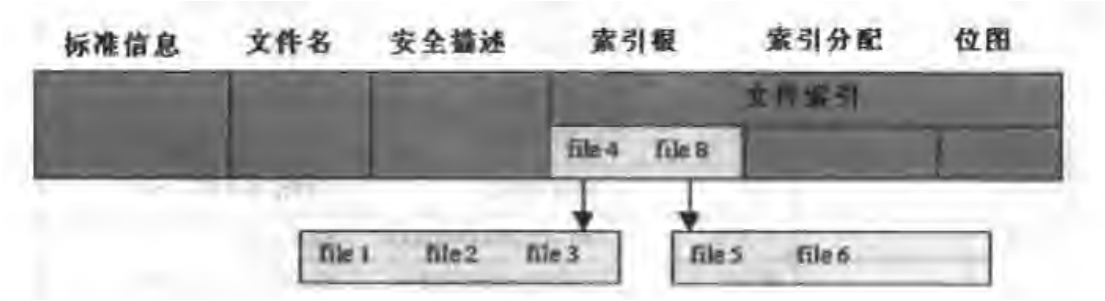
2.3.3 NTFS元文件

NTFS中文件通过主文件表（MFT，Main File Table）确定其在磁盘上的位置。MFT是一个数据库，由一系列文件记录组成。卷中每一个文件都有一个文件记录，其中第一个文件记录称作基本文件记录，里面存储有其他扩展文件记录的信息。主文件表也有自己的文件记录。

NTFS卷上的每个文件都各有一个唯一的64位的文件引用号（File Reference Number，也称文件索引号）。文件引用号由两部分组成：文件号和文件顺序号。文件号48位，对应该文件在MFT中的位置；文件顺序号随着文件记录的重用而增加（考虑到NTFS一致性检查）。

NTFS使用逻辑簇号（Logical Cluster Number，LCN）和虚拟簇号（Virtual Cluster Number）来对簇进行定位。LCN是对整个卷中的簇从头到尾的编号，卷因子乘以LCN可得到卷上的物理字节偏移。VNC是对特定文件的簇从头到尾的编号，以便引用文件中的数据，VNC可以映射成LCN而不要求在物理上连续。

NTFS目录只是一个简单的文件名和文件引用号的索引。如果目录的属性列表小于一个记录长，那么该目录所有信息存储在MFT的记录中，否则大于一个记录长的使用B+树结构进行管理（B+树便于大型目录文件和子目录的快速查找）。如图所示：



图表 二-2 NTFS目录

B-树：一种结构划索引方式。比如建立文件A时，文件系统为其创建索引文件B，由于B的规模仍然太大，为进一步提高速度，又建立了索引的索引文件C，以及索引的索引的索引文件D。这又产生了新问题：B、C、D三个索引文件的对象层次不同，结构不同，操作3个索引文件非常繁琐。所以人们研究使用一种特珠的结构来实现多重索引，B-树就是其中比较成功的方法。而NTFS所使用的“B+树”综合索引方式与其非常类似，由于这些索引的具体实施非常复杂，这里就不详述了。

MFT的基本文件记录中有一指针，指向一个存储非常驻的索引缓冲，包括该目录下所有下一级子目录和文件的外部簇。

NTFS管理是原则：磁盘上任何对象都作为文件管理。文件通过主文件表定位。

MFT文件记录大小固定的1KB。文件记录在MFT文件记录数组中物理上连续，从0开始编号，所以NTFS可以看作预定义文件系统。MFT仅供系统本身组织、架构文件系统使用，这在NTFS中被称为原数据（Metadata）。

那么原数据可以定义为：

存储在卷上支持文件系统各式管理的数据，不能被应用程序访问，只能为系统提供特殊服务。

原数据前16个记录是操作系统使用的非常重要原数据文件，这些文件以“$”开头，为隐藏文件，不能被用户列出。但是能被特殊工具列出，如NFI.EXE。

FAT文件系统簇号指针在FAT表中，而NTFS中簇号指针包含在$MFT和$MFTMirr文件中。

序号	元文件	功能
0	$MFT	主文件表本身
1	$MFTMirr	主文件表的部份镜像
2	$LogFile	日志文件
3	$Volume	卷文件
4	$AttrDef	属性定义列表
5	$Root	根目录
6	$Bitmap	位图文件
7	$Boot	引导文件
8	$BadClus	坏簇文件
9	$Secure	安全文件
10	$UpCase	大写文件
11	$Extend metadata directory	扩展元数据目录
12	$Extend\$Reparse	重解析点文件
13	$Extend\$UsnJrnl	变更日志文件
14	$Extend\$Quota	配额管理文件
15	$Extend\$ObjId	对象 ID 文件
16-23		保留
23+		用户文件和目录

图表 二-3 NTFS元文件

每一个MFT记录都对应着不同的文件，如果一个文件具有多个属性或者分散存储，那么就可能需要多个文件记录。其中第一个记录称作基本文件记录（Basic File Record）。MFT中的第1个记录就是MFT自身。由于MFT文件本身的重要性，为了确保文件系统结构的可靠性，系统专门为它准备了一个镜像文件（$MftMirr），也就是MFT中的第2个记录。

第3个记录是日志文件（$LogFile）。该文件是NTFS为实现可恢复性和安全性而设计的。当系统运行时，NTFS就会在日志文件中记录所有影响NTFS卷结构的操作，包括文件的创建和改变目录结构的命令，例如复制，从而在系统失败时能够恢复NTFS卷。

第4个记录是卷文件（$Volume），它包含了卷名、被格式化的卷的NTFS版本和一个标明该磁盘是否损坏的标志位（NTFS系统以此决定是否需要调用Chkdsk程序来进行修复）。

第5个记录是属性定义表（$AttrDef，attribute definition table），其中存放了卷所支持的所有文件属性，并指出它们是否可以被索引和恢复等。

第6个记录是根目录（\），其中保存了存放于该卷根目录下所有文件和目录的索引。在访问了一个文件后，NTFS就保留该文件的MFT引用，第二次就能够直接进行对该文件的访问。

第7个记录是位图文件（$Bitmap）。NTFS卷的分配状态都存放在位图文件中，其中每一位（bit）代表卷中的一簇，标识该簇是空闲的还是已被分配了的，由于该文件可以很容易的被扩大，所以NTFS的卷可以很方便的动态的扩大，而FAT格式的文件系统由于涉及到FAT表的变化，所以不能随意的对分区大小进行调整。

第8个记录是引导文件（$Boot），它是另一个重要的系统文件，存放着Windows 2000/XP的引导程序代码。该文件必须位于特定的磁盘位置才能够正确地引导系统。该文件是在Format程序运行时创建的，这正体现了NTFS把磁盘上的所有事物都看成是文件的原则。这也意味着虽然该文件享受NTFS系统的各种安全保护，但还是可以通过普通的文件I/O操作来

修改。

第9个记录是坏簇文件（$BadClus），它记录了磁盘上该卷中所有的损坏的簇号，防止系统对其进行分配使用。

第10个记录是安全文件（$Secure），它存储了整个卷的安全描述符数据库。NTFS文件和目录都有各自的安全描述符，为了节省空间，NTFS将具有相同描述符的文件和目录存放在一个公共文件中。

第11个记录为大写文件（$UpCase，upper case file），该文件包含一个大小写字符转换表。

第12个记录是扩展元数据目录（$Extended metadata directory）。

第13个记录是重解析点文件（$Extend\$Reparse）。

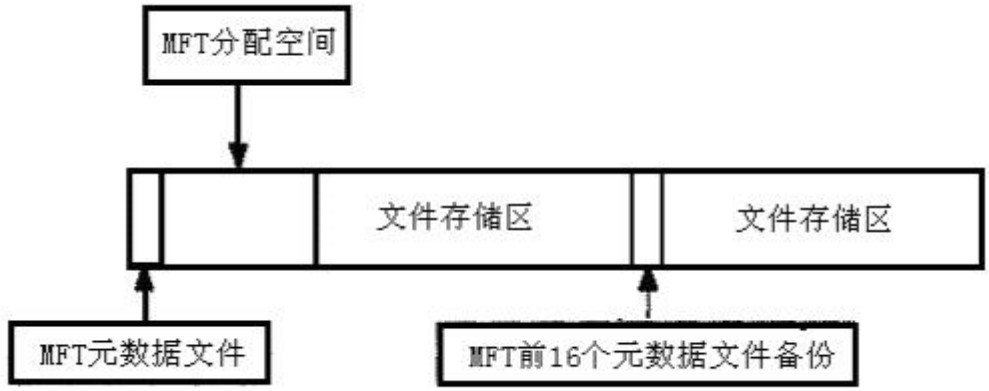
第14个记录是变更日志文件（$Extend\$UsnJrnl）。

第15个记录是配额管理文件（$Extend\$Quota）。

第16个记录是对象ID文件（$Extend\$ObjId）。

第17～23记录是是系统保留记录，用于将来扩展。

MFT的前16个元数据文件是如此重要，为了防止数据的丢失，NTFS系统在该卷文件存储部分的正中央对它们进行了备份，参见下图。



图表 二-4 MFT空间分配

NTFS把磁盘分成了两大部分，其中大约12%分配给了MFT，以满足其不断增长的文件数量。为了保持MFT元文件的连续性，MFT对这12%的空间享有独占权。余下的88%的空间被分配用来存储文件。而剩余磁盘空间则包含了所有的物理剩余空间--MFT剩余空间也包含在里面。MFT空间的使用机制可以这样来描述：当文件耗尽了存储空间时，Windows操作系统会简单地减少MFT空间，并把它分配给文件存储。当有剩余空间时，这些空间又会重新被划分给MFT。虽然系统尽力保持MFT空间的专用性，但是有时不得不做出牺牲。尽管MFT碎片有时是无法忍受的，却无法阻止它的发生。

那么NTFS访问卷过程如下；

1、 当NTFS访问某个卷时，它必须“装载”该卷：NTFS会查看引导文件（在图中的$Boot元数据文件定义的文件），找到MFT的物理磁盘地址。

2、 它就从文件记录的数据属性中获得VCN到LCN的映射信息，并存储在内存中。这个映射信息定位了MFT的运行（run或extent）在磁盘上的位置。

3、 NTFS再打开几个元数据文件的MFT记录，并打开这些文件。如有必要NTFS开始执行它的文件系统恢复操作。在NTFS打开了剩余的元数据文件后，用户就可以开始访问该卷了。

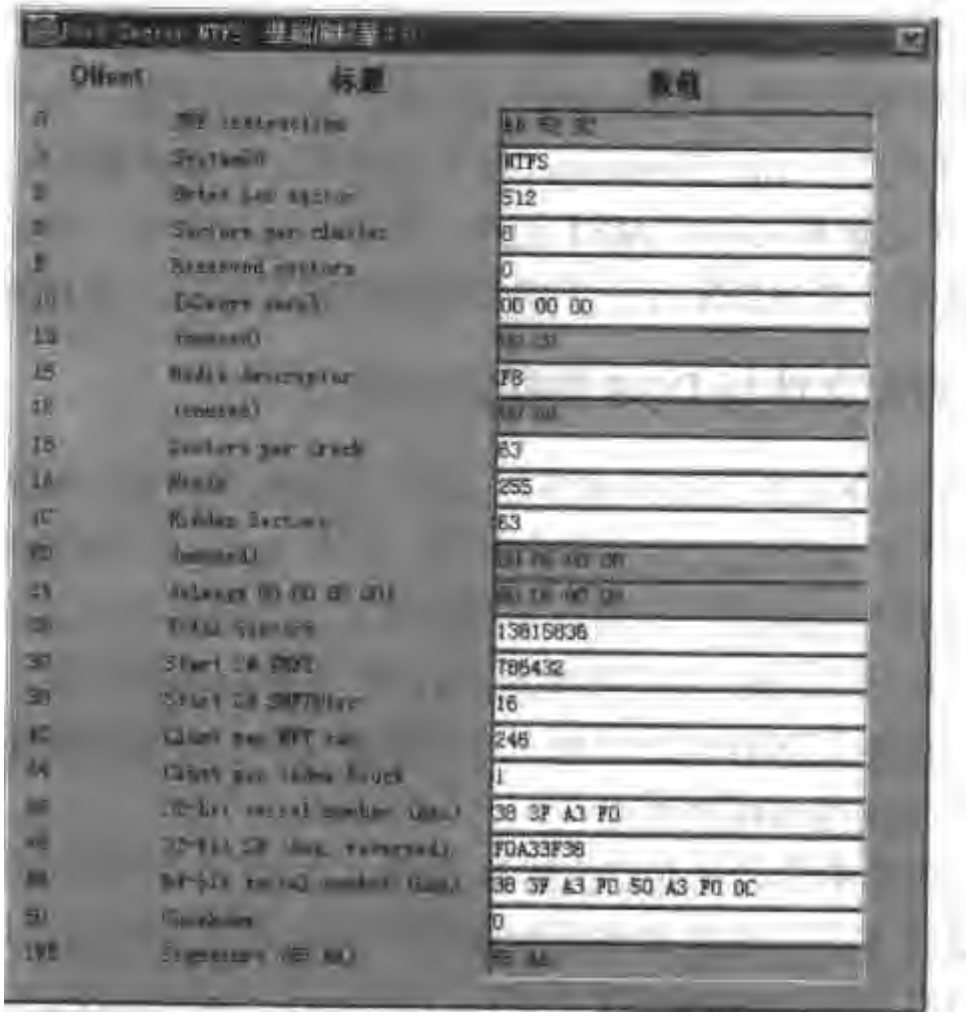
2.3.4 元文件和DBR参数关系

上节提到NTFS把磁盘上的对象都看作是文件。因此DBR和元文件在NTFS中都是文件。

与FAT32文件系统一样，NTFS文件系统得分区或卷的切入点是DBR。NTFS中DBR是本分区的第一个扇区内容，结构如图：

图表 二-5 NTFS DBR结构

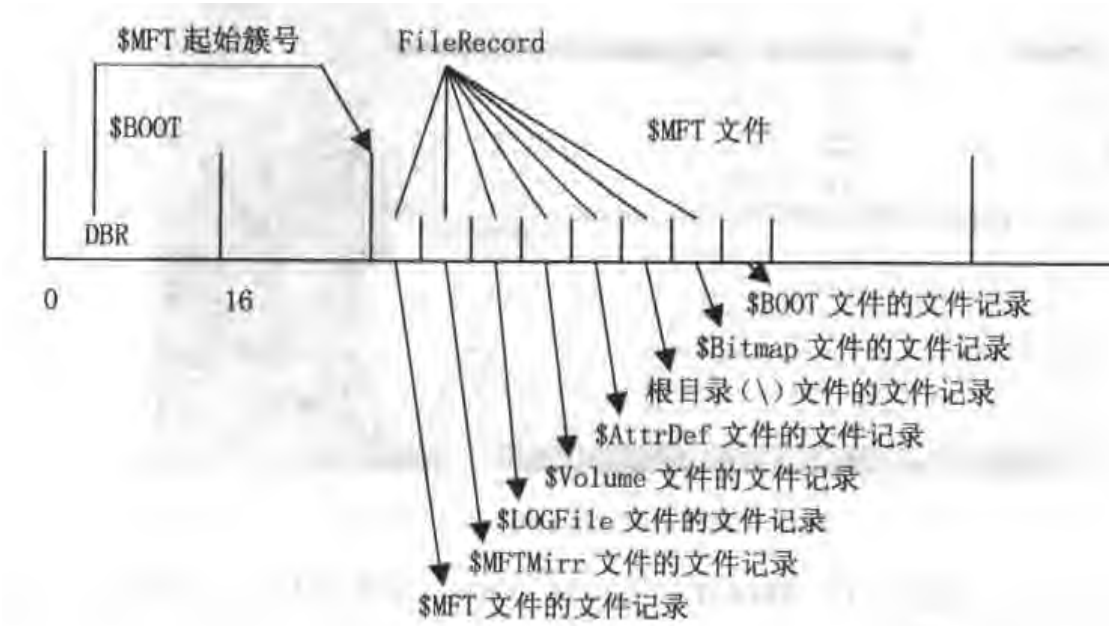
其中的参数BPB如图：



图表 二-6 NTFS BPB结构

第一个扇区DBR是一个文件，也$ROOT文件的第一个扇区。$MFT也是文件，记录$MFT文件信息的是他本身。$MFT的文件记录第一项是$MFT，第8项是$ROOT。

系统通过DBR找到$MFT，然后由$MFT定位和确定$ROOT。在所有的NTFS分区中，$BOOT占用前16扇区。引导分区中$BOOT的代码量一般占用7个扇区（0~6扇区），后面为空，这些代码是系统引导代码。引导分区和非引导分区的1~6扇区内容一致，区别是第0个扇区。如果启动分区的$BOOT文件损坏，可以用其他分区的$BOOT文件恢复。



图表 二-7 $MFT和DBR关系

$MFT的起始位置由DBR的BPB参数确定，$MFTMirr的起始位置也由DBR的BPB参数确定。

在NTFS中，除了DBR本身是预设的，其他文件信息都存储在$MFT中。

2.3.5 NTFS的文件和目录

NTFS将文件作为属性/属性值的集合来处理，这一点与其他文件系统不一样。文件数据就是未命名属性的值，其他文件属性包括文件名、文件拥有者、文件时间标记等。下图显示了一个用于小文件的MFT记录。

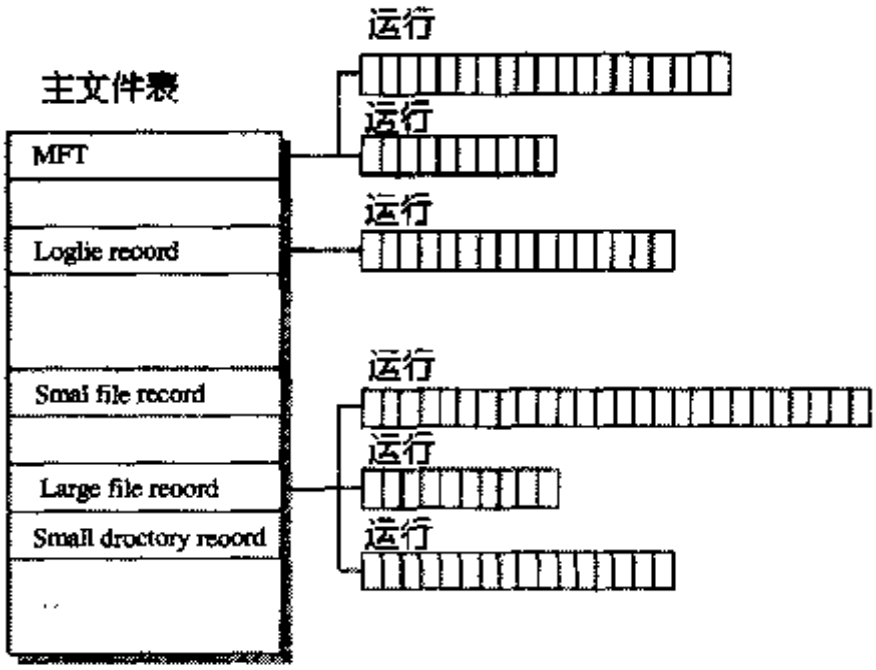
Standard Information 标准信息	file of directory name 文件或目录名	Security Descriptor 安全描述符	Data of Index 数据或索引	（未占用）

图表 二-8 小文件和目录的MFT

每个属性由单个的流（stream）组成，即简单的字符队列。严格地说，NTFS并不对文件进行操作，而只是对属性流进行读写。NTFS提供对属性流的各种操作：创建、删除、读取（字节范围）以及写入（字节范围）。读写操作一般是针对文件的未命名属性的，对于已命名的属性则可以通过已命名的数据流句法来进行操作。

NTFS目录只是一个简单的文件名和文件引用号的索引。如果目录的属性列表小于一个记录长，那么该目录所有信息存储在MFT的记录中，否则大于一个记录长的使用B+树结构进行管理（B+树便于大型目录文件和子目录的快速查找），并且用一个指针指向一个外部簇（Extent，见常驻属性和非常驻属性）。如图表 二-2 NTFS目录所示。

$MFT自身的记录长可能超过一个记录大小，所以存储情况一般如下图：



图表 二-9 $MFT存储情况

NTFS卷上文件的常用属性在下表中列出（并不是所有文件都有所有这些属性）。

属性名	属性描述
$VOLUME_INFORMATION	卷信息：仅存在于$VOLUME 元数据文件中
$VOLUME_NAME	卷名称或卷标识：仅存在于$VOLUME 元数据文件中
$STANDARD_INFORMATION	标准信息：这包括基本文件属性，如只读、存档；时间标记，如文件的创建时间和最近一次修改的时间；有多少目录指向本文件（即它的硬链接数（HARD LINK COUNT））
$FILE_NAME	文件名：这是以 UNICODE 字符表示的，由于 MS-DOS 不能正确识别 WIN32 子系统创建的文件名，当 WIN32 子系统创建一个文件名时，NTFS 会自动生成一个备用的 MS-DOS 文件名，所以一个文件可以有多种文件名属性
$SECURITY_DESCRIPTOR	安全描述符：这是为了向后兼容而保留的，主要用于保护文件以防止未授权访问，但是，WINDOWS 2000/XP 已将所有文件的安全描述符存放在$SECURE 元数据文件中，以便于共享（NTFS 的早期版本将安全描述符与文件目录一起存放，这不利于共享）
$DATA	文件数据：这是文件的内容（在 NTFS 文件系统中，一个文件除了支持文件数据即未命名的属性外，还可支持其他命名属性，即可以有多个数据属性；目录没有默认的数据属性，但是有可选的命名数据属性）
$INDEX_ROOT	索引根
$INDEX_ALLOCATION	索引分配
$BITMAP	位图
$ATTRIBUTE_LIST	属性列表：当一个文件需要使用多个 MFT 文件记录时，这用来表示该文件的属性列表
$OBJECT_ID	对象 ID:一个具有 64 个字节的标识符，其中最低的 16 个字节对卷来说是唯一的（链接跟踪服务为外壳快捷方式及 OLE 链接源文件赋予对象 ID；NTFS 提供 API 来直接通过这些对象 ID 而不是文件名来打开文件）
$REPARSE_POINT	重解析点：存储文件的重解析点数据（NTFS 的软链接与装配点都包括这个属性）
$EA	扩充属性：主要为与 OS/2 兼容，现已使用不多
$EA_INFORMATION	扩充属性信息：主要为与 OS/2 兼容，现已使用不多
$LOGGED_UTILITY_STREAM	EFS 加密属性：主要为实现 EFS（ENCRYPED FILE SYSTEM）而存储内关加密信息如解码密钥、合法访问的用户列表等。

图表 二-10 NTFS常用属性

2.3.6 常驻属性与非常驻属性

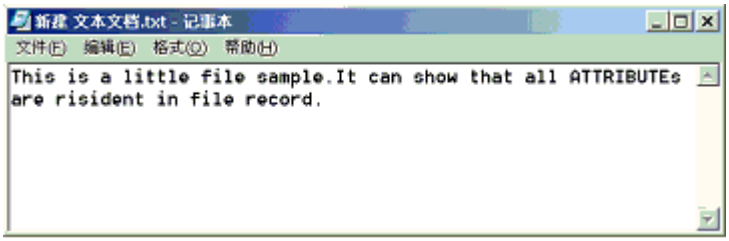
常驻属性（Resident Attribute）：当一个文件很小时，其所有属性和属性值可存放在MFT的文件记录中，这些属性值能直接存放在MFT中属性称为常驻属性。有些属性总是常驻的，这样NTFS才可以确定其他非常驻属性。例如，标准信息属性和根索引就总是常驻属性。

每个属性都是以一个标准头开始的，在头中包含该属性的信息和NTFS通常用来管理属性的信息。该头总是常驻的，并记录着属性值是否常驻、对于常驻属性，头中还包含着属性值

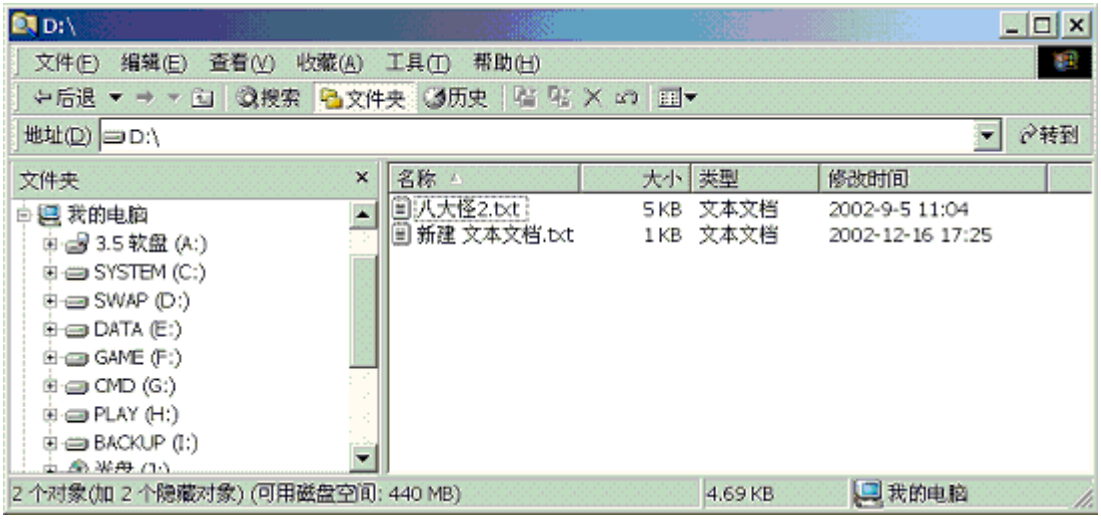
的偏侈量和属性值的长度。

如果属性值能直接存放在MFT中，那么NTFS对它的访问时间就将大大缩短。NTFS只需访问磁盘一次，就可立即获得数据；而不必像FAT文件系统那样，先在FAT表中查找文件，再读出连续分配的单元，最后找到文件的数据。

小文件或小目录的所有属性，均可以在MFT中常驻。小文件的未命名属性可以包括所有文件数据。建立一个小文件如下图所示：



图表 二-11 小文件示意图1

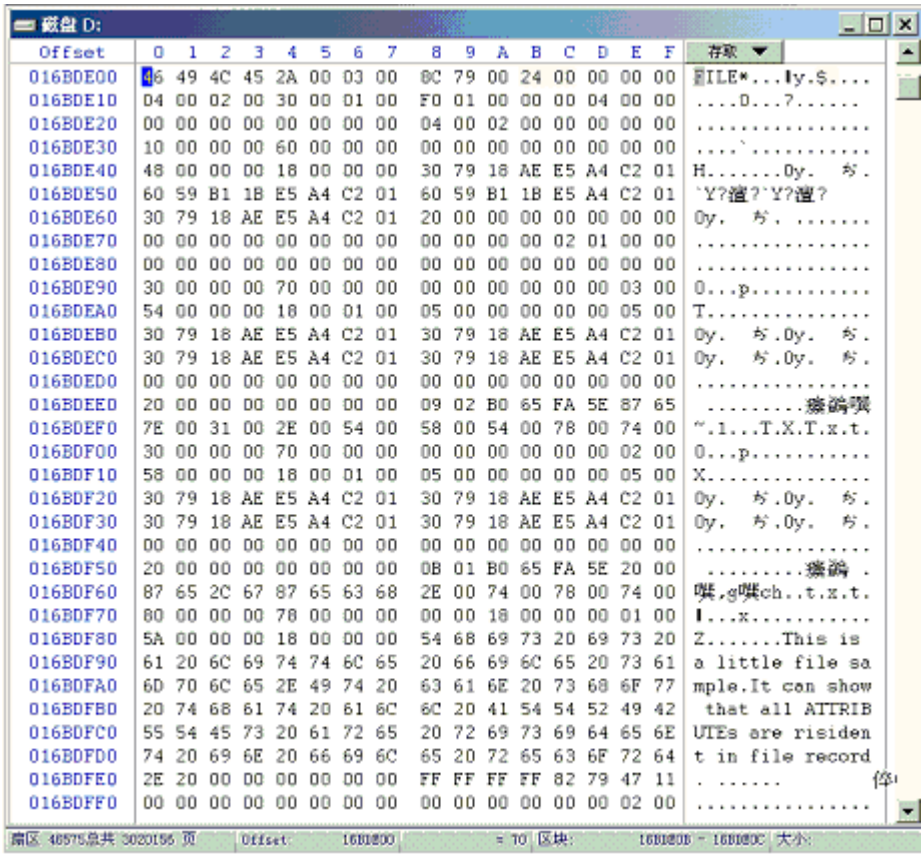


图表 二-12 小文件示意图2

如通过NFI查看文件“新建 文本文档.txt”的文件记录号为36，“nfi d:\新建 文本文档.txt”，显示内容如下：

\新建 文本文档.txt
$STANDARD_INFORMATION (resident)
$FILE_NAME (resident)
$FILE_NAME (resident)
$OBJECT_ID (resident)
$DATA (resident)

从显示内容可以看出文件的全部属性都是常驻属性，包括DATA属性，没有非常驻属性，所以，用WINHEX打开MFT，查看该文件记录，有如下图的内容：



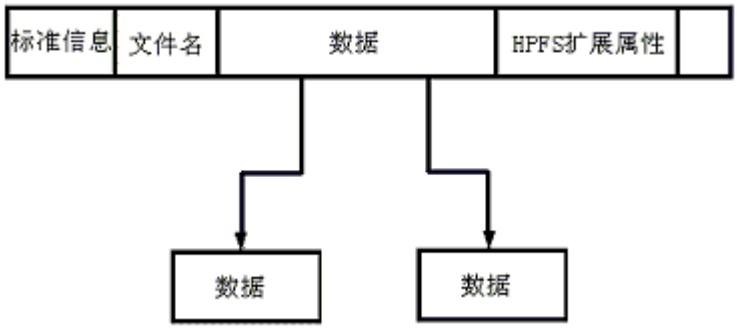
图表 二-13 小文件的文件记录

小目录的索引根属性可以包括其中所有文件和子目录的索引。参见下图

图表 二-14 小目录的MFT记录

大文件或大目录的所有属性，就不可能都常驻在MFT中。如果一个属性（如文件数据属性）太大而不能存放在只有1KB的MFT文件记录中，那么NTFS将从MFT之外分配区域。这些区域通常称为一个运行（run）或一个盘区（extent），它们可用来存储属性值，如文件数据。如果以后属性值又增加，那么NTFS将会再分配一个运行，以便用来存储额外的数据。值存储在运行中而不是在MFT文件记录中的属性称为非常驻属性（nonresident attribute）。NTFS决定了一个属性是常驻还是非常驻的；而属性值的位置对访问它的进程而言是透明的。

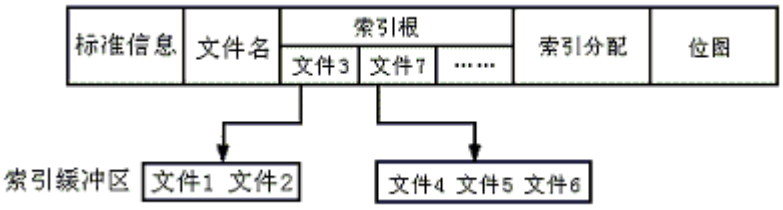
当一个属性为非常驻时，如大文件的数据，它的头部包含了NTFS需要在磁盘上定位该属性值的有关信息。下图显示了一个存储在两个运行中的非常驻属性。



图表 二-15 存储在两个运行中的非常驻属性

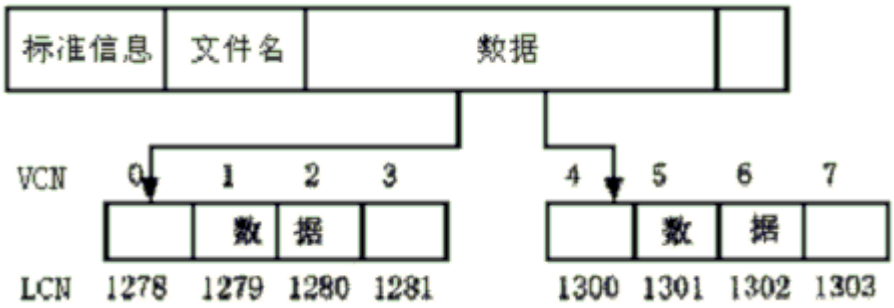
在标准属性中，只有可以增长的属性才是非常驻的。对文件来说，可增长的属性有数据、属性列表等。标准信息和文件名属性总是常驻的。

一个大目录也可能包括非常驻属性（或属性部分），参见图表二-16。在该例中，MFT文件记录没有足够空间来存储大目录的文件索引。其中一部分索引存放在索引根属性中，而另一部分则存放在叫作“索引缓冲区”（index buffer）的非常驻运行中。这里，索引根、索引分配以及位图属性都是简化表示的，这些属性将在后面详细介绍。对目录而言，索引根的头及部分值应是常驻的。



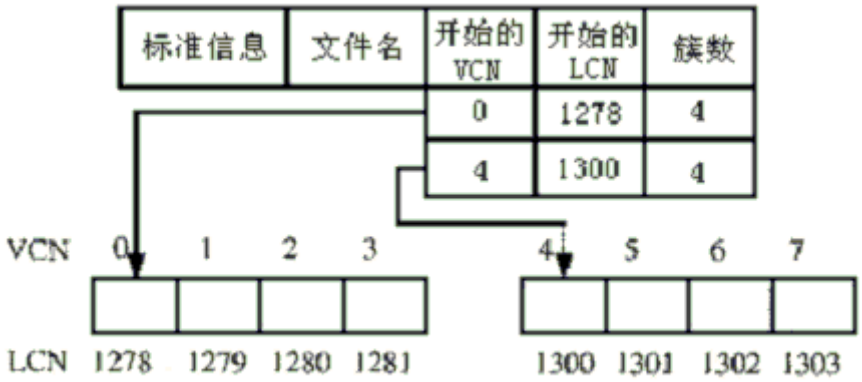
图表 二-16 大目录的MFT记录

当一个文件（或目录）的属性不能放在一个MFT文件记录中，而需要分开分配时，NTFS通过VCN-LCN之间的映射关系来记录运行（run）或盘区情况。LCN用来为整个卷中的簇按顺序从0到n进行编号，而VCN则用来对特定文件所用的簇按逻辑顺序从0到m进行编号。下图显示了一个非常驻数据属性的运行所使用的VCN与LCN编号。



图表 二-17 非常驻数据属性的VCN

当该文件含有超过2个运行时，则第三个运行从VCN8开始，数据属性头部含有前两个运行VCN的映射，这便于NTFS对磁盘文件分配的查询。为了便于NTFS快速查找，具有多个运行文件的常驻数据属性头中包含了VCN-LCN的映射关系，参见下图



图表 二-18 非常驻数据属性的VCN-LCN映射

虽然数据属性常常因太大而存储在运行中，但是其他属性也可能因MFT文件记录没有足够空间而需要存储在运行中。另外，如果一个文件有太多的属性而不能存放在MFT记录中，

那么第二个MFT文件记录就可用来容纳这些额外的属性（或非常驻属性的头）。在这种情况下，一个叫作“属性列表”（attribute list）的属性就加进来。属性列表包括文件属性的名称和类型代码以及属性所在MFT的文件引用。属性列表通常用于太大或太零散的文件，这种文件因VCN-LCN映射关系太大而需要多个MFT文件记录。具有超过200个运行的文件通常需要属性列表。

2.3.7 MFT文件记录结构分析

2.3.7.1基本概念

元文件$MFT是NTFS中最重要的文件，它记录了所有的文件和目录的情况，包括卷的信息、启动文件、$MFT文件本身等卷上的重要信息，如文件名、安全属性、文件大小、存储位置等。元文件$MFT由一系列文件记录组成，每一个记录由头部和属性组成，由“FFFFFFFF”结束，一般大小1KB或者一个簇。属性部分变长，以“FFFFFFFF”结束（严格说是下一个属性开始），大小1KB的记录项中属性部分开始偏移为0x30。

文件除了属性外还包括“流”，“流”是一个字节序，包含了属性的实际值。Windows NT/2000/XP/2003中，命令行访问文件的流语法为“文件名：属性名”，因此在文件名中不能包含“：”。

元数据文件$AttrDef中预定义了常用的文件属性，可以直接使用。标准信息的属性名用10H表示，属性列表中的属性名用20H表示，文件名的属性名用30H表示，把一个常用属性名专门存放在一个文件中，可以大大节省系统开销。

每一个属性分为两部分：标准属性头、内容。

内容部分的结构总是以属性名开始（N字节长），在属性名之后定义该属性是否为常驻属性。当文件属性的数据流就存储在其属性名后时，它就是常驻属性，这样，对于那些流较小且不会增长的文件属性就可以提供更佳的访问次数。如果一个文件属性是非常驻的，那么其流就存储在一个或多个扩展或称为运行中。运行是一个在逻辑簇号上连续的区域。为访问这些运行，NTFS紧跟在文件属性名后存储有一个称为运行列表的表。

常驻和非常驻属性有各自的头结构，他们开始的0x0e偏移内容是一致的，头部结构如下表：

表格 二-3从头部开始的偏移长度描述

0	4	Type	（类型）
4	4	Length	（长度）
8	1	Non-resident flag	（非常驻标志）
9	1	N=Name length	（文件名长度）
A	2	Offset to the content part	（相对内容部分的偏移值）
C	2	Compressed flag	（压缩标志）
E	2	Identificator	（标识）
其他部分（常驻属性或非常驻属性）			

- 文件名长度：00 表示文件属性没有命名。
- 压缩标志：在NTFS中,数据压缩是在文件属性级别上实现的，这就意味着，如果出现意外，你也不会释放出很多的数据。这样，尽管只是对文件进行压缩，但压缩文件同时就意味着其属性数据也一样被压缩。从现在开始， 其头部的安排依赖文件的常驻属性：

对一个常驻属性来说，从头部开始的偏移描述如下：

表格 二-4常驻属性从头部开始的偏移描述

10	4	Length of the stream（流长度）
14	2	Offset to the stream（流偏移）
16	2	Indexed flag（索引标志）

- 索引标志：文件属性通过一个索引入口进行索引。

对于一个非常驻的文件属性，从头部开始的偏移描述如下：

表格 二-5非常驻属性从头部开始的偏移描述

10	8	Starting VCN （起始VCN）
18	8	Last VCN （结束VCN）
20	2	Offset to the runlist （运行列表偏移）
22	2?	Number of compression engine ? （压缩引擎号）
28	8	Allocated size of the stream （为流分配的单元大小）
30	8	Real size of the stream （实际的流大小）
38	8	Initialized data size of the stream （流已初始化大小）

- VCN ： Virtual Cluster Number（虚拟簇号）的缩略词。VCN是一个与非常驻属性相关联的概念。VCN从文件属性流的第一个运行的第一个簇（VCN 0）到最后一个运行的最后一个簇进行编号。 当某个运行列表非常大，文件属性不能放在一个文件记录中时，描述文件的文件属性就会存储在几个文件记录中，运行列表也分成几个小片。起始VCN域和结束VCN域都用于定位其文件记录指示—即运行列表—运行所指定的VCN 。

注意：

如果属性可以放在一个文件记录内，则结束VCN域（这种情况下没有使用）可能是“00 00 00 00 00 00 00 00”。

- 压缩引擎的数量：为达到最好的压缩比率，NTFS可以根据不同类型的数据使用不同的压缩引擎。当前的压缩引擎使用值04。
- 为流分配的单元大小：它几倍于卷上用来存储文件属性流所描述的分配空间。如果流没有压缩，它就是数倍于簇空间大小的实际大小，相反，则比较小。
- 流的实际大小：文件属性流在压缩前的大小。
- 流的初始化大小： 这是文件属性流的压缩后的大小（总是低于分配大小）。如果此流未被压缩，就是它的实际大小 。

注意：

常驻文件属性从不被压缩（也没有压缩引擎号域），因为它的流太小。

- 信息是足够的：名字长+内容部分的偏移值=到流的偏移值(常驻属性)或者到运行列表的偏移值(非常驻悔改)。

2.3.7.2文件记录头分析

$MFT文件记录头部结构如下图所示：

偏移	长度	描　述
0x00	4	固定值，一定是“FILE”
0x04	2	头部大小
0x06	2	固定列表大小
0x08	8	日志文件序列号
0x10	2	序列号（用于记录本文件记录被重复使用的次数，每次文件删除时加 1，跳过 0 值，如果为 0，则保持为 0）
0x12	2	硬连接数，只出现在基本文件记录中，目录所含项数要使用到它
0x14	2	第一个属性的偏移地址
0x16	2	标志字节，1 表示记录使用中，2 表示该记录为目录
0x18	4	文件记录实际大小
0x1C	4	文件记录分配大小
0x20	8	所对应的基本文件记录的文件参考号（扩展文件记录中使用，基本文件记录中为 0，在基本文件记录的属性列表 0X20 属性存储中扩展文件记录的相关信息）
0x28	2	下一个自由 ID 号，当增加新的属性时，将该值分配给新属性，然后该值增加，如果 MFT 记录重新使用，则将它置 0，第一个实例总是 0
0x2A	2	边界，Windows XP 中使用，也就是本记录使用的两个扇区的最后两个字节的值
0x2C	4	Windows XP 中使用，本 MFT 记录号

图表 二-19 $MFT文件记录头部结构

NTFS通过给一个文件创建几个文件属性的放式来实现POSIX的硬连接。每一个文件名属性都有自己的详细信息和父目录，当删除一个硬连接时，相应的文件名从MFT文件记录中删除，当所有的硬连接删除后文件才被完全删除。

2.3.7.3标准属性分析

在$AttrDef中，标准属性的属性头部结构如下图：

偏移	大小	值	描　述
0x00	4	0X10	属性类型（10H，标准属性）
0x04	4	0X60	总长度（包括标准属性头头部本身）
0x08	1	0x00	非常驻标志
0x09	1	0x00	属性名的名称长度
0x0A	2	0x18	属性名的名称偏移
0x0C	2	0x00	标志（似乎已经不再使用，统一放在文件属性中）
0x0E	2	0x00	标识
0x10	4	L	属性长度（L）
0x14	2	0x18	属性内容起始偏移
0x16	1	0x00	索引标志
0x17	1	0x00	填充
0x18	L	0xE0…	从此处开始，共 L 字节为属性值

图表 二-20标准属性头结构

偏移	大小	操作系统	描　述
~	~		标准属性头（已经分析过）
0x00	8		C TIME——文件创建时间
0x08	8		A TIME——文件修改时间
0x10	8		M TIME——MFT 变化时间
0x18	8		R TIME——文件访问时间
0x20	4		文件属性（按照 DOS 术语来称呼，都是文件属性）
0x24	4		文件所允许的最大版本号（0 表示未使用）
0x28	4		文件的版本号（最大版本号为 0，则也为 0）
0x2C	4		类 ID（一个双向的类索引）
0x30	4	Windows 2000	所有者 ID（表示文件的所有者，是文件配额$QUOTA 中$O 和$Q 索引的关键字，为 0 表示未使用磁盘配额）
0x34	4	Windows 2000	安全 ID 是文件$SECURE 中$SII 索引和$SDS 数据流的关键字，注意不要与安全标识相混淆
0x38	8	Windows 2000	本文件所占用的字节数，它是文件所有流占用的总字节数，为 0 表示未使用磁盘配额
0x40	8	Windows 2000	更新系列号（USN），是到文件$USNJRNL 的一个直接的索引，为 0 表示 USN 日志未使用

图表 二-21标准属性的属性结构

文件属性的含义如图：

标　志	二 进 制 位	意　义
0x0001	0000 0000　0000 0001	只读
0x0002	0000 0000　0000 0010	隐含
0x0004	0000 0000　0000 0100	系统
0x0020	0000 0000　0010 0000	存档
0x0040	0000 0000　0100 0000	设备
0x0080	0000 0000　1000 0000	常规
0x0100	0000 0001　0000 0000	临时
0x0200	0000 0010　0000 0000	稀疏文件
0x0400	0000 0100　0000 0000	重解析点
0x0800	0000 1000　0000 0000	压缩
0x1000	0001 0000　0000 0000	脱机
0x2000	0010 0000　0000 0000	未编入索引
0x4000	0100 0000　0000 0000	加密

图表 二-22 文件属性的含义

2.3.7.4文件名属性分析

文件名属性是一种常驻属性，用于存储文件名，紧跟标准属性之后。如$AttrDef定义，其

大小从68bytes到578bytes不等，与最大文件名为255个Unicode字符对应。

文件名属性由一个标准的属性头和可变成的属性内容两部分组成。头结构和标准属性头相同，结构如图：

偏移	大小	值	描　述
0x00	4	0x30	属性类型（30H，文件名属性）
0x04	4	0x68	总长度（包括头部本身）
0x08	1	0x00	非常驻标志
0x09	1	0x00	属性名的名称长度
0x0A	2	0x18	属性名的名称偏移
0x0C	2	0x00	标志（0X0001 表示压缩，0X4000 表示加密，0X8000 表示稀疏文件，常驻属性不会被压缩）
0x0E	2	0x03	标识
0x10	4	0x4A	属性长度（L）
0x14	2	0x18	属性内容起始偏移
0x16	1	0x01	索引标志
0x17	1	0x00	填充
0x18	L	0xE0…	从此处开始，共 L 字节为属性

图表 二-23文件名属性头结构

内容如下图所示：

偏移	大小	值
~	~	标准的属性头结构（见表 4-11）
0x00	8	父目录的文件参考号（即父目录的基本文件记录号，分为两个部分，前 6 个字节 48 位为父目录的文件记录号，此处为 0X05，即根目录，所以$MFT 的父目录为根目录，后 2 个字节为序列号）
0x08	8	文件创建时间
0x10	8	文件修改时间
0x18	8	最后一次的 MFT 更新时间
0x20	8	最后一次的访问时间
0x28	8	文件分配大小
0x30	8	文件实际大小
0x38	4	标志，如目录、压缩、隐藏等
0x3c	4	用于 EAs 和重解析点
0x40	1	以字符计的文件名长度，每字符占用字节数由下一字节命名空间确定，一个字节长度，所以文件名最大为 255 字节长
0x41	1	文件名命名空间，见表 4-16 的详细解释
0x42	2L	以 Unicode 方式表示的文件名

图表 二-24文件名属性结构

文件分配按照簇分配空间，实际的文件大小是指未命名的数据流大小，这也是在Windows下看到的文件大小。标志占用4Bytes，含义如下图：

标　志	二 进 制 位	意　义
0x0001	0000 0000　0000 0001	只读
0x0002	0000 0000　0000 0010	隐含
0x0004	0000 0000　0000 0100	系统
0x0020	0000 0000　0010 0000	存档
0x0040	0000 0000　0100 0000	设备
0x0080	0000 0000　1000 0000	常规
0x0100	0000 0001　0000 0000	临时
0x0200	0000 0010　0000 0000	稀疏文件
0x0400	0000 0100　0000 0000	重解析点
0x0800	0000 1000　0000 0000	压缩
0x1000	0001 0000　0000 0000	脱机
0x2000	0010 0000　0000 0000	未编入索引
0x4000	0100 0000　0000 0000	加密
0x10000000	0001 0000　0000 0000（前两个字节）	目录（从 MFT 文件记录中拷贝的相应的位）
0x20000000	0010 0000　0000 0000（前两个字节）	索引视图（从 MFT 文件记录中拷贝的相应的位）

图表 二-25标志含义图

命名空间是一个有关文件名可以使用的字符标志集。NTFS为了支持旧的程序，为每一个与dos不兼容的文件名分配了一个短文件名。

常见的命名空间如下图：

标志	意　义	描　述
0	POSIX	这是最大的命名空间。它大小写敏感，并允许使用除 NULL (0) 和左斜框 (/) 以外的所有 Unicode 字符作为文件名，文件名最大长度为 255 个字符。有一些字符，如冒号 (:)，在 NTFS 下有效，但 Windows 不让使用，因为 Windows 把它作为多数据流的专用符号了
1	Win32	Win32 是 POSIX 命名空间的一个子集，不区分大小写，可以使用除" * / : < > ? \ | 外的所有 Unicode 字符。另外，文件名不能以句点和空格结束
2	DOS	DOS 是 WIN32 命名空间的一个子集，要求比空格的 ASCII 码要大，且不能使用" * + , / : ; .<=>?\ 等字符，另外其格式是 1~8 个字符的文件名，然后是句点分隔，然后是 1~3 个字符的扩展名
3	Win32 & DOS	该命名空间要求文件名对 Win32 和 DOS 命名空间都有效，这样，文件名就可以在文件记录中只保存一个文件名

图表 二-26常见的命名空间

2.3.7.5数据流属性分析

数据流属性为未命名的非常驻属性，其头部结构如下；

偏移	大小	值	意　义
0x00	4	0x80	属性类型（0x80，数据流属性）
0x04	4	0x48	属性长度（包括本头部的总大小）
0x08	1	0x01	非常驻标志，此处就表示数据流非常驻
0x09	1	0x00	名称长度，$AttrDef 中定义，所以名称长度为 0
0x0A	2	0x0040	名称偏移
0x0C	2	0x00	标志，如 0x0001 为压缩标志，0x4000 为加密标志，0x8000 为稀疏文件标志
0x0E	2	0x0001	标识
0x10	8	0x00	起始 VCN，此处为 0
0x18	8	0x1FF1	结束 VCN，此处为 0X1FF1
0x20	2	0x40	数据运行的偏移
0x22	2	0x00	压缩引擎
0x24	4	0x00	填充
0x28	8	0x1FF2000	为属性值分配大小（按分配的簇的字节数计算）
0x30	8	0x1FF1C00	属性值实际大小
0x38	8	0x1FF1C00	属性值压缩大小
0x40	...	32F21F00000C	数据运行

图表 二-27数据流属性头结构

NTFS最大卷2^{64}-1个簇，XP中NTFS卷最大限制是2^{32}-1个簇。由于MBR的限制，硬盘仅支持2TB大小的分区。超过这个限制需要建立NTFS动态卷。

2.3.7.6 位图属性分析

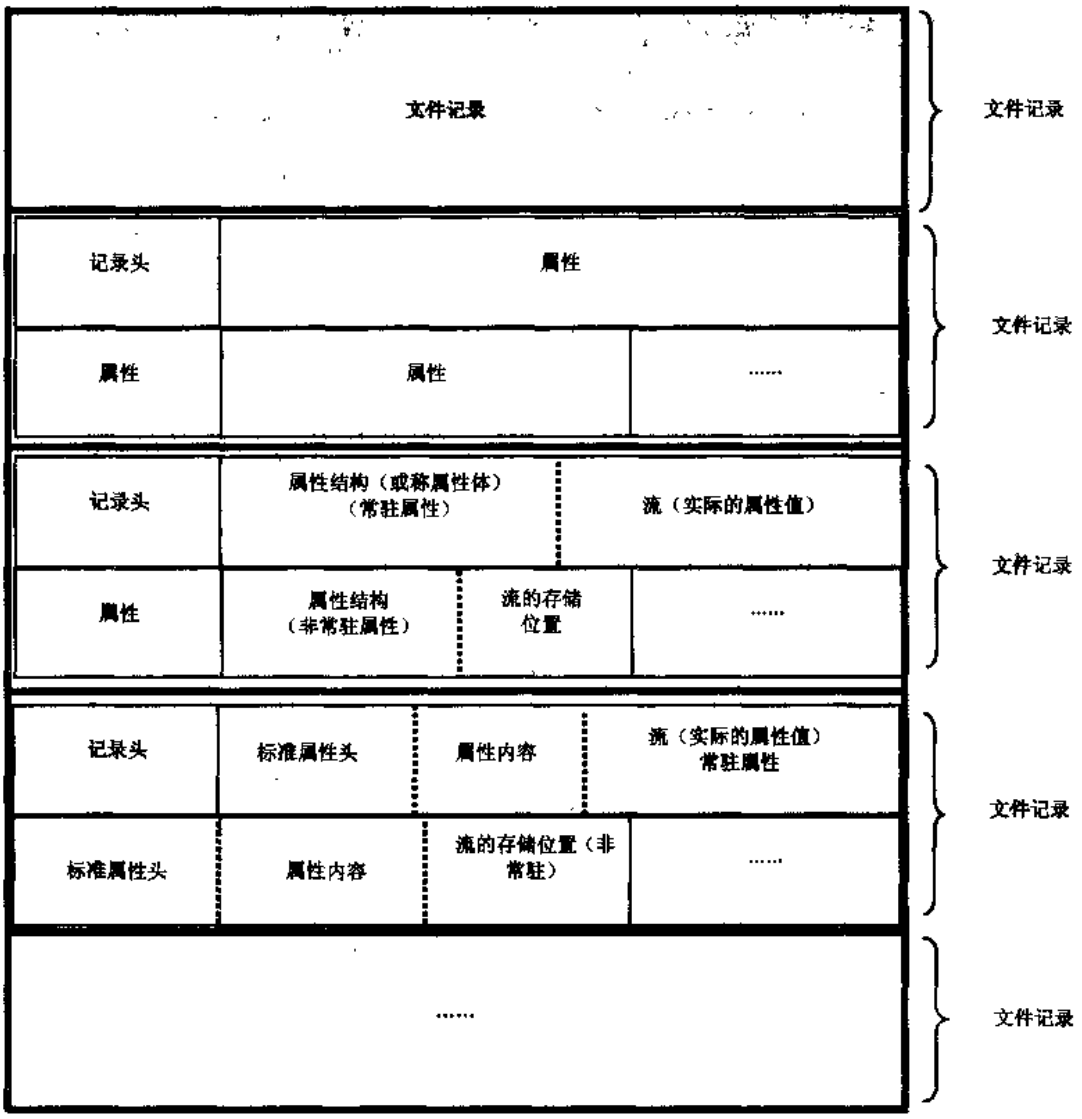
$MFT文件最后一个属性是0xB0位图属性。位图属性是未命名的非常驻属性，属性头结构和数据流属性头类似，如图所示：

偏移	大小	值	意　义
0x00	4	0XB0	属性类型（0xB0，位图属性）
0x04	4	0X48	属性长度（包括本头部的总大小）
0x08	1	0x01	非常驻标志，此处表示位图数据非常驻
0x09	1	0x00	名称长度，$AttrDef 中定义，所以名称长度为 0
0x0A	2	0x40	名称偏移
0x0C	2	0x00	标志，如 0x0001 为压缩标志，0x4000 为加密标志，0x8000 为稀疏文件标志
0x0E	2	0x05	标识
0x10	8	0x00	起始 VCN，此处为 0
0x18	8	0x00	结束 VCN，此处为 0
0x20	2	0x40	数据运行的偏移
0x22	2	0x00	压缩引擎
0x24	4	0x00	填充
0x28	8	0x1000	为属性值分配大小（按分配的簇的字节数计算）
0x30	8	0x1000	属性值实际大小
0x38	8	0x1000	压缩大小
0x40	...	3101404BOF	数据运行

图表 二-28位图属性头

2.3.7.7 $MFT结构总结

总上几节内容，$MFT的结构大致如下图：



图表 二-29 $MFT的结构示意图

因此，$MFT是由一系列的文件记录组成，每个文件记录由一个文件头和一组属性及属性的实际值（流）组成，每个属性由一个属性头和一个属性内容组成，对于常驻属性，流存储在文件记录中，而非常驻属性流的位置存储在文件记录中，流内容在数据区。

2.3.8 $Boot元文件介绍

$Boot用于系统启动的系统文件，包含卷大小、簇和MFT等信息，是唯一不能重新定位（移动）的文件，属性如下图：

类　型	描　述	名　称
0x10	标准信息	标准属性
0x30	文件名	$Boot
0x50	安全描述符	描述安全信息
0x80	数据	未命名

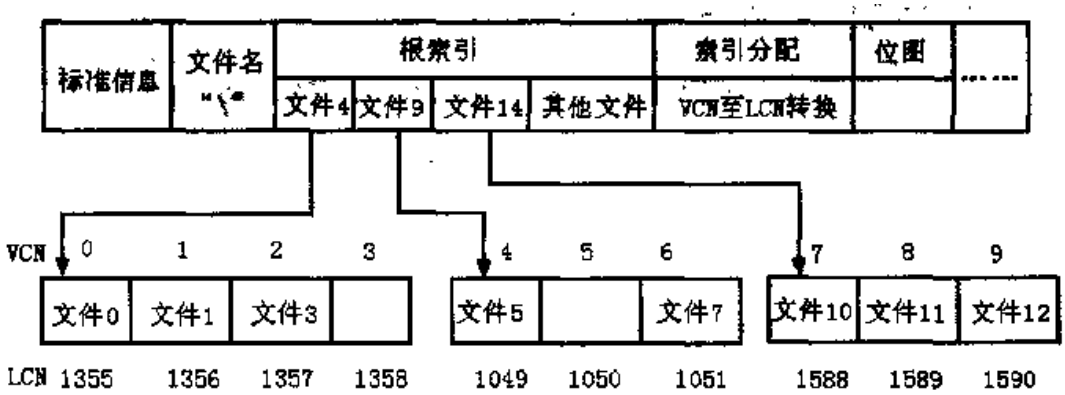
图表 二-30 $Boot元文件属性

偏　移	大　小	描　述
0x0000	3	跳转到引导程序
0x0003	8	系统 ID：“NTFS”
0x000B	2	每扇区字节数
0x000D	1	每簇扇区数
0x000E	7	未使用
0x0015	1	介质描述（硬盘为 F8）
0x0016	2	未使用
0x0018	2	每磁道扇区数
0x001A	2	磁头数
0x001C	8	未使用
0x0024	4	通常是 80 00 80 00
0x0028	8	卷的总扇区数
0x0030	8	$MFT 的 VCN 为 0 所对应的 LCN，即起始簇号
0x0038	8	$MFTMirr 的 VCN 为 0 所对应的 LCN
0x0040	4	每 MFT 记录的簇数（注）
0x0044	4	每索引记录的簇数（注）
0x0048	8	卷序列号
–	–	其他如引导代码、提示信息、有效标志等
0x0200		Windows NT 加载程序

图表 二-31 $Boot元文件未命名数据流含义

2.3.9 NTFS索引与目录

NTFS系统中，文件目录是文件名的一个索引。根目录索引如下图：



图表 二-32根目录文件索引

第一个索引记录都是由一个标准的索引头和一些包含索引键和索引数据的块组成的。索引记录的大小在引导记录 $Boot中定义，一般总是4KB。

标准索引头的结构如下：

表格 二-6标准索引头的结构

偏移	大小	说明
0X00	4	总是“INDX”
0X04	2	更新序号偏移
0X06	2	更新序列号USN的大小与排列，包括第一个字节
0X08	8	日志文件序列号LSN
0X10	8	该索引缓冲在索引分配中的索引VCN
0X18	4	索引入口的偏移（相对于0X18）
0X1C	4	索引入口的大小（相对于0X18）
0X20	4	索引入口的分配大小（相对于0X18）
0X24	1	非页级节点为1（有子索引）
0X25	3	总是0
0X28	2	更新序列号
0X2A	2S-2	更新序列排列

索引头后是索引项，结构如下图：

偏移	大小	描　　述
0x00	8	文件的 MFT 参考号
0x08	2	索引项大小
0x0A	2	文件名偏移
0x0C	2	索引标志
0x0E	2	填充（到 8 字节）
0x10	8	父目录的 MFT 文件参考号
0x18	8	文件创建时间
0x20	8	最后修改时间
0x28	8	文件记录最后修改时间
0x30	8	最后访问时间
0x38	8	文件分配大小
0x40	8	文件实际大小
0x48	8	文件标志
0x50	1	文件名长度（F）
0x51	1	文件名命名空间
0x52	2F	文件名
2F+0x52	P	填充（到 8 字节）
P+2F+0x52	8	子节点索引缓存的 VCN

图表 二-33索引项结构示意图

NTFS有多个索引，常用的如下表：

表格 二-7常用索引列表

名称	索引	说明
$I30	文件名	目录使用
$SDH	安全描述	$SECURE
$SII	安全IDS	$SECURE
$O	对象IDS	$OBJID
$O	所有者IDS	$QUOTA
$Q	配额	$QUOTA
$R	重解析点	$REPARSE

2.4 EFS

EFS是NTFS一个十分重要的特性和功能。

EFS（encrypting file system，文件加密系统）提供一种核心文件加密技术，该技术用于在 NTFS 文件系统卷上存储已加密的文件。加密了文件或文件夹之后，您还可以像使用其他文件和文件夹一样使用它们。

加密对加密该文件的用户是透明的。这表明不必在使用前手动解密已加密的文件，您就可以正常打开和更改文件。

使用 EFS 类似于使用文件和文件夹上的权限。两种方法都可用于限制数据的访问。然而，未经许可对加密文件和文件夹进行物理访问的入侵者将无法阅读这些文件和文件夹中的内容。如果入侵者试图打开或复制已加密文件或文件夹，入侵者将收到拒绝访问消息。文件和文件夹上的权限不能防止未授权的物理攻击。

当一个用户使用EFS去加密文件时，必须存在一个公钥和一个私钥，如果用户没有，EFS服务自动产生一对。对于初级用户来说，即使他完全不懂加密，也能加密文件，可以对单个文件进行加密，也可以对一个文件夹进行加密，这样所有写入文件夹的文件将自动被加密。

一旦用户发布命令加密文件或试图添加一个文件到一个已加密的文件夹中，EFS将进行以下几步：

第一步：NTFS首先在这个文件所在卷的卷信息目录下(这个目录隐藏在根目录下面)创建一个叫做efs0.log的日志文件，当拷贝过程中发生错误时利用此文件进行恢复。

第二步：然后EFS调用CryptoAPI设备环境.设备环境使用Microsoft Base Cryptographic Provider 1.0 产生密匙,当打开这个设备环境后,EFS产生FEK(File Encryption Key,文件加密密匙).FEK的长度为128位（仅US和Canada），这个文件使用DESX加密算法进行加密。

第三步: 获取公有/私有密匙对;如果这个密匙还没有的话(当EFS第一次被调用时),EFS产生一对新的密匙.EFS使用1024位的RSA算法去加密FEK.

第四步：EFS为当前用户创建一个数据解密块Data Decryptong Field(DDF),在这里存放FEK然后用公有密匙加密FEK.

第五步：如果系统设置了加密的代理,EFS同时会创建一个数据恢复块Data Recovery Field (DRF),然后把使用恢复代理密匙加密过的FEK放在DRF.每定义一个恢复代理,EFS将会创建一个Data Recovery Agent(DRA).Winxp没有恢复代理这个功能,所以没有这一步.，这个区域的目的是为了在用户解密文件的中可能解密文件不可用。这些用户叫做恢复代理，恢复代理在EDRP(Encryption Data Recovery Policy,加密数据恢复策略)中定义，它是一个域的安全策略。如果一个域的EDRP没有设置，本地EDRP被使用。在任一种情况下，在一个加密发生时，EDRP必须存在（因此至少有一个恢复代理被定义）。DRF包含使用RSA加密的FEK和恢复代理的公钥。如果在EDRP列表中有多个恢复代理，FEK必须用每个恢复代理的公钥进行加密，

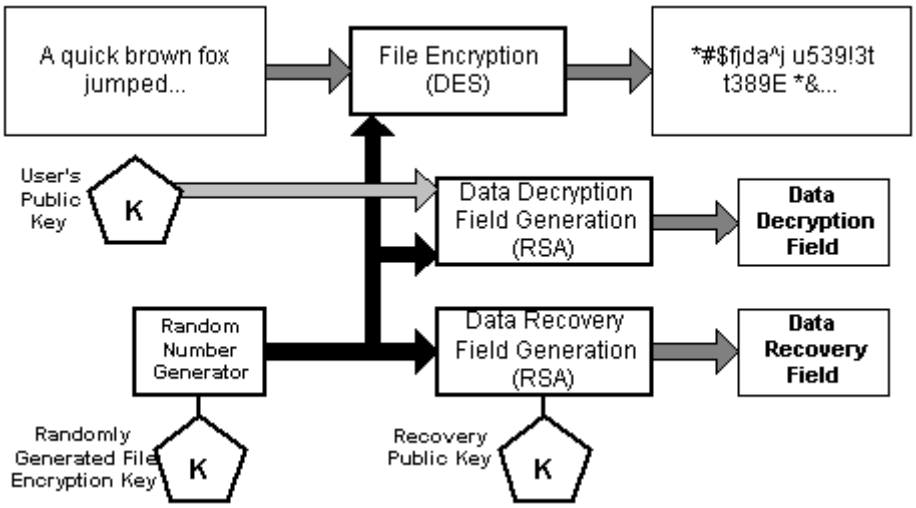
因此，必须为个恢复代理创建一个DRF。

第六步：包含加密数据、DDF及所有DRF的加密文件被写入磁盘。

第七步: 在加密文件所在的文件夹下将会创建一个叫做Efs0.tmp的临时文件.要加密的内容被拷贝到这个临时文件,然后原来的文件被加密后的数据覆盖.在默认的情况下,EFS使用128位的DESX算法加密文件数据,但是Windows还允许使用更强大的的168位的3DES算法加密文件,这是FIPS算法必须打开,因为在默认的情况下它是关闭的.

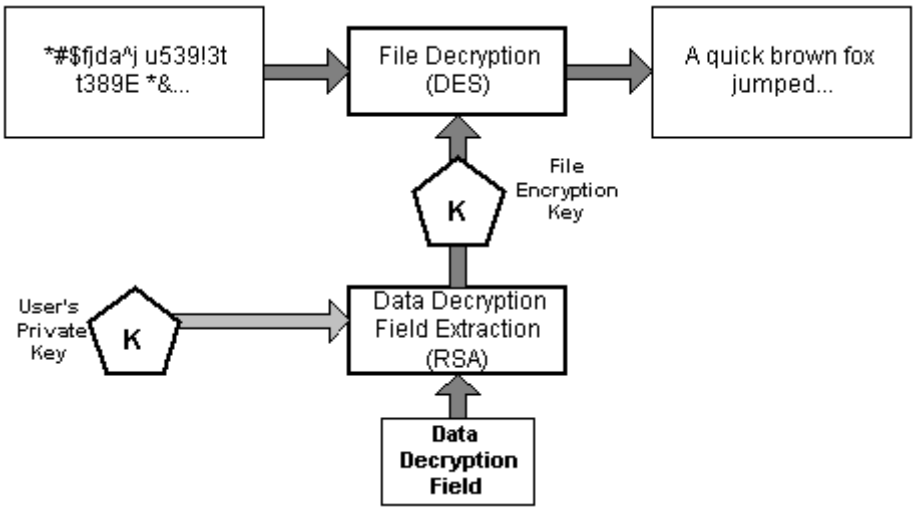
第八步：在第一步中创建的文本文件和第七步中产生的临时文件被删除。

加密过程如图：



图表 二−34 EFS加密过程

解密的过程和加密的过程是相反，如果所示：



图表 二−35 EFS解密过程

第三章 NTFS文件基本操作

NTFS文件基本操作包括文件的创建、读、写、删除等。（未完成）

第四章 NTFS文件系统数据恢复研究

本章主要研究NTFS数据恢复。

4.1数据恢复原理

数据恢复这项工作涵盖的范围很广，各种不同的存储介质在执行数据恢复的时候都会有一些区别，另外数据丢失或损坏的原因也不尽相同。我们讲解面向的对象主要是磁存储介质，如硬盘、软盘以及数据磁带等等。

首先我们需要讲解一下磁存储技术的原理，这有助于我们更深刻的了解数据恢复工作。磁存储技术的工作原理是通过改变磁粒子的极性来在磁性介质上记录数据。在读取数据时，磁头将存储介质上的磁粒子极性转换成相应的电脉冲信号，并转换成计算机可以识别的数据形式。进行写操作的原理也是如此。要使用硬盘等介质上的数据文件，通常需要依靠操作系统所提供的文件系统功能，文件系统维护着存储介质上所有文件的索引。因为效率等诸多方面的考虑，在我们利用操作系统提供的指令删除数据文件的时候，磁介质上的磁粒子极性并不会被清除。操作系统只是对文件系统的索引部分进行了修改，将删除文件的相应段落标识进行了删除标记。同样的，目前主流操作系统对存储介质进行格式化操作时，也不会抹除介质上的实际数据信号。正是操作系统在处理存储时的这种设定，为我们进行数据恢复提供了可能。值得注意的是，这种恢复通常只能在数据文件删除之后相应存储位置没有写入新数据的情况下进行。因为一旦新的数据写入，磁粒子极性将无可挽回的被改变从而使得旧有的数据真正意义上被清除。另外，除了磁存储介质之外，其它一些类型存储介质的数据恢复也遵循同样的原理，例如U盘、CF卡、SD卡等等。因为这些存储设备也和磁盘一样使用类似扇区、簇这样的方式来对数据进行管理。举个例子来说，目前几乎所有的数码相机都遵循DCIM标准，该标准规定了设备以FAT形式来对存储器上的相片文件进行处理。

删除文件，其实是修改文件头的前2个代码。这种修改映射在文件分配表中，就为文件作了删除标记，但文件的内容仍保存在原来的簇，如果不被后来保存的数据覆盖，它就不会从磁盘上抹掉。文件被删除后，既然其数据仍在磁盘上，文件分配表中也有它的信息，这个文件就有恢复的机会，只要找出文件头，并恢复前2个代码，在文件分配表中重新映射一下，这个文件就被恢复了。但是，文件被删除后，如果它所占的簇被存入其他数据，文件头也被覆盖，这个文件在文件分配表中的信息就会被新的文件映射所代替，这个文件一般也就无法恢复了。恢复文件，其实就是用恢复软件的查找分析功能找出文件头，重写前2个代码，并修改文件分配表中的映射记录。仅仅是删除的文件，恢复起来比较容易，如果整个磁盘被格式化了，恢复的困难就更大些，但是只要恢复软件能搜寻，并分析到它的残存的文件头，就有可能利用文件头中的信息，连接文件原来占用的簇，以恢复被删除的文件。然而，如果一个文件的某些簇被其他数据覆盖，即使恢复软件强行把原来占用各簇的数据连接起来恢复文件，但是因为其中的某些簇已不是该文件自身的数据，所以这个恢复后的文件往往无法使用。

相信大家了解了数据恢复的原理之后，就可以很容易的理解为什么使用普通的删除方法，无法彻底和安全的清除数据了。这也是为什么很多企业求助于专业的数据擦除服务公司，请他们使用专业的设备和软件彻底的对企业的敏感数据进行销毁。越来越多的情况证明，只是单纯的对存储介质进行覆写，乃至从物理上破坏存储设备，都不能保证数据不会被恢复出来。在一些拥有尖端设备的实验室中，既使被覆盖多次的磁盘，也可能被还原出最早

存储在上面的磁性信号。这种情况对那些需要恢复他们宝贵数据的用户来说可能是个另人激动的消息，但对于希望保护自己数据的人们来说则恰恰相反。我们希望用户在了解了更多有关数据恢复技术的细节信息之后，能够选择恰当的方式来照管他们的数据。

4.2常见数据恢复工具

目前常见的可以对NTFS数据恢复工具主要有以下几款：

- EasyRecovery Pro
- Revival
- RecoverNT/98
- Recover 4 All
- FinalData

通常数据恢复工具的功能有：

- 修复主引导扇区（MBR）
- 修复BIOS参数块（BPB）
- 修复分区表
- 修复文件分配表（FAT）或主文件表（MFT）
- 修复根目录
- 恢复已经被删除,但是数据没有被清除文件。

这些工具主要特定介绍如下：

1、Harddisk Data Recovery 3.0 硬盘文件系统恢复工具，该工具不象其它修复工具需要写入数据从而有可能把您的硬盘搞得更糟，它可以 在内存重建文件系统，并可以安全的把数据传输到另外一个设备，支持FAT16/32、NTFS。

2、Disk Genius(DiskMan) 2.0 国产全中文硬盘分区表维护软件，软件主要功能及特点：

- 仿WINDOWS纯中文图形界面，支持鼠标操 作；
- 提供比fdisk更灵活的分区操作，支持分区参数编辑；
- 提供强大的分区表重建功能，迅速修复损坏 了的分区表；
- 支持FAT/FAT32分区的快速格式化；
- 在不破坏数据的情况下直接调整FAT/FAT32 分区的大小；
- 自动重建被破坏的硬盘主引导记录；
- 为防止误操作，对于简单的分区动作，在存盘之 前仅更改内存缓冲区，不影响硬盘分区表；
- 能查看硬盘任意扇区，并可保存到文件。
- 可隐藏FAT/FAT32 及NTFS分区。
- 可备份包括逻辑分区表及各分区引导记录在内的所有硬盘分区信息。
- 提供扫描硬盘 坏区功能，报告损坏的柱面。
- 具备扇区拷贝功能。
- 可以彻底清除分区数据。

3、File Scavenger 2.1 能够恢复在NTFS格式下意外删除的文件工具，能够救回的文件不只单一文件，还包括整个目录及压缩 文件，也支持救回来的文件选择在原来所在位置恢复或储存到其它的硬盘。

4、EasyRecovery Pro 6.0是威力非常强大的硬盘数据恢复工具。能够帮你恢复丢失的数据以及重建文件系统。EasyRecovery不会向你的原始驱动器写入任何东东，它主要是在内存中重建文件分区表使数据能够安全地传输到其他驱动器中。你可以从被病毒破坏或是已经格式化的硬盘中恢复数据。该软件可以恢复大于8.4GB的硬盘。支持长文件名。 被破坏的硬盘中像丢

失的引导记录、BIOS参数数据块；分区表；FAT表；引导区都可以由它来进行恢复。
5、Recover98 用来在 WIN 98 下搜索并恢复误删除的文件和子目录的工具，可惜它的试用版，仅能救回3个文件。
6、TiraMisu Data Recovery deMo FAT32 4.03 是恢复硬盘因病毒感染,意外格式化或其它因素所导至的资料损失恢复工具软件,即使电脑以无法开机启动,仍可利用本身所作的启动盘开机来找出所失去的硬盘资料,并让你将这些资料拷贝到其它的硬盘。
7、FinalData 是一个文件恢复程序，利用它能够恢复被删除的重要信息，甚至还能从已经格式化或者已经损坏的磁盘中抽取文件，允许恢复完整的目录并尽量保持其原有的目录结构。
8、多数杀毒软件都有重建分区表的功能。

4.3 NTFS数据技术基础

4.3.1基础概念

4.3.1.1硬盘数据结构

刚出厂一块硬盘，我们是没有办法使用的，你需要将它分区、格式化，然后再安装上操作系统才可以使用。就拿我们一直沿用到现在的Win9x/Me系列来说，我们一般要将硬盘分成主引导扇区、操作系统引导扇区、FAT、DIR和Data等五部分（其中只有主引导扇区是唯一的，其它的随你的分区数的增加而增加）。

1、主引导扇区

主引导扇区位于整个硬盘的0磁道0柱面1扇区，包括硬盘主引导记录MBR（Main Boot Record）和分区表DPT（Disk Partition Table）。其中主引导记录的作用就是检查分区表是否正确以及确定哪个分区为引导分区，并在程序结束时把该分区的启动程序（也就是操作系统引导扇区）调入内存加以执行。至于分区表，很多人都知道，以80H或00H为开始标志，以55AAH为结束标志，共64字节，位于本扇区的最末端。值得一提的是，MBR是由分区程序（例如DOS 的Fdisk.exe）产生的，不同的操作系统可能这个扇区是不尽相同。如果你有这个意向也可以自己去编写一个，只要它能完成前述的任务即可，这也是为什么能实现多系统启动的原因（说句题外话:正因为这个主引导记录容易编写，所以才出现了很多的引导区病毒）。

2、操作系统引导扇区

OBR（OS Boot Record）即操作系统引导扇区，通常位于硬盘的0磁道1柱面1扇区（这是对于DOS来说的，对于那些以多重引导方式启动的系统则位于相应的主分区/扩展分区的第一个扇区），是操作系统可直接访问的第一个扇区，它也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。其实每个逻辑分区都有一个OBR，其参数视分区的大小、操作系统的类别而有所不同。引导程序的主要任务是判断本分区根目录前两个文件是否为操作系统的引导文件（例如MSDOS或者起源于MSDOS的Win9x/Me的IO.SYS和MSDOS.SYS）。如是，就把第一个文件读入内存，并把控制权交予该文件。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（Allocation Unit，以前也称之为簇）的大小等重要参数。OBR由高级格式化程序产生（例如DOS 的Format.com）。

3、文件分配表

FAT(File Allocation Table)即文件分配表，是DOS/Win9x系统的文件寻址系统，为了数据安全起见，FAT一般做两个，第二FAT为第一FAT的备份, FAT区紧接在OBR之后，其大小由本分区的大小及文件分配单元的大小决定。
FAT的格式历来有很多选择，Microsoft 的DOS及Windows采用我们所熟悉的FAT12、FAT16和FAT32格式，但除此以外并非没有其它格式的FAT，像Windows NT、OS/2、UNIX/Linux、Novell等都有自己的文件管理方式。

4、目录区

DIR是Directory即根目录区的简写，DIR紧接在第二FAT表之后,只有FAT还不能定位文件在磁盘中的位置，FAT还必须和DIR配合才能准确定位文件的位置。DIR记录着每个文件（目录）的起始单元（这是最重要的）、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磁盘的具体位置及大小了。在DIR区之后，才是真正意义上的数据存储区，即DATA区。

5、数据区

DATA虽然占据了硬盘的绝大部分空间，但没有了前面的各部分，它对于我们来说，也只能是一些枯燥的二进制代码，没有任何意义。在这里有一点要说明的是，我们通常所说的格式化程序（指高级格式化，例如DOS下的Format程序），并没有把DATA区的数据清除，只是重写了FAT表而已，至于分区硬盘，也只是修改了MBR和OBR，绝大部分的DATA区的数据并没有被改变，这也是许多硬盘数据能够得以修复的原因。但即便如此，如MBR/OBR/FAT/DIR之一被破坏的话，也足够咱们那些所谓的DIY老鸟们忙乎半天了……需要提醒大家的是，如果你经常整理磁盘，那么你的数据区的数据可能是连续的，这样即使MBR/FAT/DIR全部坏了，我们也可以使用磁盘编辑软件（比如DOS下的DiskEdit），只要找到一个文件的起始保存位置，那么这个文件就有可能被恢复（当然了，这需要一个前提，那就是你没有覆盖这个文件……）。

4.3.1.2硬盘分区方式

我们平时说到的分区概念，不外乎三种:主分区、扩展分区和逻辑分区。

主分区是一个比较单纯的分区，通常位于硬盘的最前面一块区域中，构成逻辑C磁盘。在主分区中，不允许再建立其它逻辑磁盘。

扩展分区的概念则比较复杂，也是造成分区和逻辑磁盘混淆的主要原因。由于硬盘仅仅为分区表保留了64个字节的存储空间，而每个分区的参数占据16个字节，故主引导扇区中总计可以存储4个分区的数据。操作系统只允许存储4个分区的数据，如果说逻辑磁盘就是分区，则系统最多只允许4个逻辑磁盘。对于具体的应用，4个逻辑磁盘往往不能满足实际需求。为了建立更多的逻辑磁盘供操作系统使用，系统引入了扩展分区的概念。

所谓扩展分区，严格地讲它不是一个实际意义的分区，它仅仅是一个指向下一个分区的指针，这种指针结构将形成一个单向链表。这样在主引导扇区中除了主分区外，仅需要存储一个被称为扩展分区的分区数据，通过这个扩展分区的数据可以找到下一个分区（实际上也就是下一个逻辑磁盘）的起始位置，以此起始位置类推可以找到所有的分区。无论系统中建立多少个逻辑磁盘，在主引导扇区中通过一个扩展分区的参数就可以逐个找到每一个逻辑磁盘。

需要特别注意的是，由于主分区之后的各个分区是通过一种单向链表的结构来实现链接

的，因此，若单向链表发生问题，将导致逻辑磁盘的丢失。

4.3.1.3数据存储原理

既然要进行数据的恢复，当然数据的存储原理我们不能不提，在这之中，我们还要介绍一下数据的删除和硬盘的格式化相关问题。

1、文件的读取

操作系统从目录区中读取文件信息（包括文件名、后缀名、文件大小、修改日期和文件在数据区保存的第一个簇的簇号），我们这里假设第一个簇号是0023。

操作系统从0023簇读取相应的数据，然后再找到FAT的0023单元，如果内容是文件结束标志（FF），则表示文件结束，否则内容保存数据的下一个簇的簇号，这样重复下去直到遇到文件结束标志。

2、文件的写入

当我们要保存文件时，操作系统首先在DIR区中找到空区写入文件名、大小和创建时间等相应信息，然后在Data区找到闲置空间将文件保存，并将Data区的第一个簇写入DIR区，其余的动作和上边的读取动作差不多。

3、文件的删除

看了前面的文件的读取和写入，你可能没有往下边继续看的信心了，不过放心，Win9x的文件删除工作却是很简单的，简单到只在目录区做了一点小改动——将目录区的文件的第一个字符改成了E5就表示将改文件删除了。

Fdisk和Format的一点小说明

和文件的删除类似，利用Fdisk删除再建立分区和利用Format格式化逻辑磁盘（假设你格式化的时候并没有使用/U这个无条件格式化参数）都没有将数据从DATA区直接删除，前者只是改变了分区表，后者只是修改了FAT表，因此被误删除的分区和误格式化的硬盘完全有可能恢复。

4.3.1.4系统启动流程

各种不同的操作系统启动流程不尽相同，我们这里以Win9x/DOS的启动流程为例。

1、系统加电自检POST过程。POST是Power On Self Test的缩写，也就是加电自检的意思，微机执行内存FFFF0H处的程序（这里是一段固化的ROM程序），对系统的硬件（包括内存）进行检查。

2、读取分区记录和引导记录。当微机检查到硬件正常并与CMOS设置相符后，按照CMOS设置从相应设备启动（我们这里假设从硬盘启动），读取硬盘的分区记录（DPT）和主引导记录（MBR）。

3、读取DOS引导记录。微机正确读取分区记录和主引导记录后，如果主引导记录和分区表校验正确，则执行主引导记录并进一步读取DOS引导记录（位于每一个主分区的第一个扇区），然后执行该DOS引导记录。

4、装载系统隐含文件。将DOS系统的隐含文件IO.SYS入内存，加载基本的文件系统FAT，这时候一般会出现Starting Windows 9x...的标志，IO.SYS将MS.SYS读入内存，并处理System.dat和User.dat文件，加载磁盘压缩程序。

5、实DOS模式配置。系统隐含文件装载完成，微机将执行系统隐含文件，并执行系统配置文件（Config.sys），加载Config.sys中定义的各种驱动程序。
6、调入命令解释程序(Command.com)。系统装载命令管理程序，以便对系统的各种操作命令进行协调管理（我们所使用的Dir、Copy等内部命令就是由Command.com提供的）。
7、执行批处理文件(Autoexec.bat)。微机将一步一步地执行批处理文件中的各条命令。
8、加载Win.com。Win.com负责将Windows下的各种驱动程序和启动执行文件加以执行，至此启动完毕。

数据恢复的基础知识到此就给你介绍得差不多了。如果你领会了以上的这些知识，相信加上工具软件的辅助，恢复丢失的数据不是很困难。

4.3.2硬盘数据恢复方案分析

在NTFS上数据遭到破坏主要有下面几种方式：
1、文件语删除
2、病毒破坏
3、分区表破坏
4、全盘崩溃和分区丢失
5、文件丢失、误格式化
6、文件损坏
7、硬盘被加密或变换
8、文件加密后密码遗忘
9、系统用户密码遗忘的处理
对于以上不同情况应予不同的恢复方案。

1、文件语删除

这可能是最简单同时也是最常见的数据损坏，直接的表述就是一般删除文件后清空了回收站，或按住Shift键删除，要不然就是在“回收站”的“属性”中勾选了“删除时不将文件移入回收站，而是彻底删除”。

恢复在NTFS分区下被误删除的文件。对于这种相对简单的需求，File Scavenger完全就可以胜任。当然，File Scavenger是很具有针对性的——它只能在WinNT/2000系统下使用（同时必须以Administrator用户登录系统），而且只对NTFS格式的分区有效。不过它支持压缩过的NTFS分区或文件夹中文件的恢复，并对格式化过的NTFS分区中的文件也有效（注意:File Scavenger只可以对格式化过的分区中的文件进行恢复，并不能恢复整个被格式化过的分区）。

2、病毒破坏

由于病毒破坏硬盘数据的方法各异，恢复的方案就需要对症下药。以常见的CIH为例。当用户的硬盘数据一旦被CIH病毒破坏后，使用KV3000的F10功能，可修复的程度如下：

- C盘容量为2.1G以上， 原FAT表是32位的，C分区的修复率为98%，D、E、F等分区的修复率为99%, 配合手工C、D、E、F等分区的修复率为100%。
- 硬盘容量为2.1G以下，原FAT表是16位的，C分区的修复率为0%，D、E、F等分区的修复率为99%, 配合手工D、E、F盘的修复率为100%。

因为原C盘是16位的短FAT表，所以C盘的FAT表和根目录下的文件目录都被CIH病毒乱码覆盖了。 KV3000可以把C盘找回来，虽然根目录的文件名字已被病毒乱码覆盖看不到了，但文件的内容影像还存储在C盘内的某些扇区上。推荐用KV3000找回C盘，再用文件修

复软件TIRAMISU.EXE可将C盘内的部分文件影像找回来，如果原存放文件影像的簇是相连的，找回的文件就完整无损。

但对于FAT16的C盘是不是中了CIH就没救呢？可以尝试一下FIXMBR，它可以通过全盘搜索，决定硬盘分区，并重新构造主引导扇区。由于软件只修改主引导扇区记录，对其它扇区不进行写操作，故一般不会带来不安全目录（如果修复得不理想，请DiskEdit等工具进行手工修复）。注意:FIXMBR是一个比较老的程序，对WinNT、Linux以及FAT32考虑得不多。

3、分区表破坏

分区表破坏，可能是数据损坏中除了物理损坏之外最严重的一种灾难性破坏。究其原因，不外乎以下几种:

- 个人误操作删除分区，只要没有进行其它的操作完全可以恢复。
- 安装多系统引导软件或者采用第三方分区工具，有恢复的可能性。
- 病毒破坏，可以部分或者全部恢复。
- 利用Ghost克隆分区/硬盘破坏，只可以部分恢复或者不能恢复。

在Norton Utility系列工具中，功能十分强大，可以恢复分区记录、FAT表，需要注意的是它对硬盘的操作不是只读的，因此你需要每一步都做好Undo文件，这样即使误操作也可以恢复，Norton Disk Doctor配合DiskEdit在分区表不能恢复时也可以恢复部分文件。

最专业的数据恢复公司出的软件，当然很有专业风范，EasyRecovery支持的文件系统格式很多FAT、NTFS都支持，并且有专门的For Novell版本。EasyRecovery对于分区破坏和硬盘意外被格式化都可安全的恢复，你所要做的就是将数据损坏硬盘挂到另外一台电脑上，尽情恢复就是了，不过EasyRecovery对于中文的文件名和目录名效果不是很好（一些乱码，但文章内容绝对是正确的）。

由出品PartitionMagic的PowerQuest公司所出的，硬盘资料复原工具。它是一套恢复硬盘因病毒感染，意外格式化等因素所导致的资料损失工具软件，能将已删除的文件资料找出并恢复，也能找出已重新格式化的硬盘、被破坏的FAT分配表、启动扇区等等，几乎能找出及发现任何在硬盘上的资料（支持FAT16和FAT32及长文件名）。恢复回来的资料能选择在原来所在位置恢复或保存到其它可写入资料的硬盘，也提供了自动备份目录、文件和系统配置文件的功能，能在任何时间恢复）。

4、全盘崩溃和分区丢失

首先重建MBR代码区，再根据情况修正分区表。修正分区表的基本思路是查找以55AA为结束的扇区，再根据扇区结构和后面是否有FAT等情况判定是否为分区表，最后计算填回主分区表，由于需要计算，过程比较烦琐。如果文件仍然无法读取，要考虑用Tiramint等工具进行修复。如果在FAT表彻底崩溃，恢复某个指定文件，可以用DiskEdit或Debug查找已知信息。比如文件为文本，文件中包含“软件狗”，那么把它们转换为内码C8 ED BC FE B9 B7进行查找。

5、文件丢失、误格式化的情况

一般情况下，删除文件仅仅是把文件的首字节，改为E5H，而并不破坏文件本身，因此可以恢复。但对不连续文件要恢复文件链，而由于手工交叉恢复对一般计算机用户来说并不容易，这里就就不讲了，建议用工具处理，如果已经安装了Norton Utilities，可以用它来查找。另外，RecoverNT等工具都是恢复的利器。但是应特别注意，千万不要在发现文件丢失后，在本机安装什么恢复工具，你可能恰恰把文件覆盖掉了。特别是如果你的文件在C盘，发现主要文件被你失手清掉了（比如你按SHIFT删除），你应该马上直接关闭电源，用软盘启动进行恢复或把硬盘串接到其它有恢复工具的机器处理。

6、文件损坏

一般的说，恢复损坏文件须要清楚地了解文件的结构，但这并不是很容易的事情，而这方面的工具也不多。不过，文件如果字节正常，不能正常打开往往是文件头损坏。

7、硬盘被加密或变换

此时千万不要进行FDISK/MBR，SYS等处理，否则数据再也无法找回，一定要反解加密算法，或找到被移走的重要扇区。对于那些加密硬盘数据的病毒，清除时一定要选择能恢复加密数据的可靠杀毒软件。

8、文件加密后密码遗忘

对于很多字处理软件的文件加密和ZIP等压缩包的加密，你是不能靠加密逆过程来完成的，因为那从理论上是异常困难的。目前有一些相关的软件，他们的思想一般都是用一个大字典集中的数据循环用相同算法加密后与密码的密文匹配，直到一致时则说明找到了密码。你可以去寻找这些软件，当然，有些软件是有后门的，比如DOS 下的WPS，Ctrl+qiubojun就是通用密码。Undiskp的作者冯志宏是解文件密码的个中高手，大家不妨去他的主页看看。

9、系统用户密码遗忘的处理

最简单的方法就是用软盘启动（NT的你也可以把盘挂接在其他NT上），找到支持该文件系统结构的软件（比如针对NT的NTFSDOS），利用他把密码文件清掉、或者是COPY出密码档案，用破解软件套字典来处理。前者时间短但所有用户信息丢失，后者时间长，但保全了所有用户信息。对UNIX系统，建议你一定先做一张应急盘。

4.4数据恢复技术实现

（未完成）

4.5数据恢复经验介绍

4.5.1几个常识性问题

1、硬盘逻辑坏道可以修复，而物理坏道不可修复。实际情况是，坏道并不分为逻辑坏道和物理坏道，不知道谁发明这两个概念，反正厂家提供的技术资料中都没有这样的概念，倒是分为按逻辑地址记录的坏扇区和按物理地址记录的坏扇区。
2、硬盘出厂时没有坏道，用户发现坏道就意味着硬盘进入危险状态。实际情况是，每个硬盘出厂前都记录有一定数量的坏道，有些数量甚至达到数千上万个坏扇区，相比之下，用户发现一两个坏道算多大危险？
3、硬盘不认盘就没救，0磁道坏可以用分区方法来解决。实际情况是，有相当部分不认的硬盘也可以修好，而0磁道坏时很难分区。

4.5.2技术来源

1、搜集国外技术资料与国外专业人士交流；

2、购买专业工具软件（有同步技术更新支持）；
3、实践。

4.5.3硬盘修复需要理解的基本概念

在研究硬盘修复和使用专业软件修复硬盘的过程中，必将涉及到一些基本的概念。在这里，高朋根据自己的研究和实践经验，试图总结并解释一些与“硬盘缺陷”相关的概念，与众位读者交流。

Bad sector (坏扇区)

在硬盘中无法被正常访问或不能被正确读写的扇区都称为Bad sector。一个扇区能存储512Bytes的数据，如果在某个扇区中有任何一个字节不能被正确读写，则这个扇区为Bad sector。除了存储512Bytes外，每个扇区还有数十个Bytes信息，包括标识（ID）、校验值和其它信息。这些信息任何一个字节出错都会导致该扇区变“Bad”。例如，在低级格式化的过程中每个扇区都分配有一个编号，写在ID中。如果ID部分出错就会导致这个扇区无法被访问到，则这个扇区属于Bad sector。有一些Bad sector能够通过低级格式化重写这些信息来纠正。

Bad cluster (坏簇)

在用户对硬盘分区并进行高级格式化后，每个区都会建立文件分配表（File Allocation Table, FAT)。FAT中记录有该区内所有cluster（簇）的使用情况和相互的链接关系。如果在高级格式化（或工具软件的扫描）过程中发现某个cluster使用的扇区包括有坏扇区，则在FAT中记录该cluster为Bad cluster，并在以后存放文件时不再使用该cluster,以避免数据丢失。有时病毒或恶意软件也可能在FAT中将无坏扇区的正常cluster标记为Bad cluster, 导致正常cluster不能被使用。这里需要强调的是，每个cluster包括若干个扇区，只要其中存在一个坏扇区，则整个cluster中的其余扇区都一起不再被使用.

Defect (缺陷)

在硬盘内部中所有存在缺陷的部分都被称为Defect。如果某个磁头状态不好，则这个磁头为Defect head。如果盘面上某个Track(磁道)不能被正常访问，则这Track为Defect Track. 如果某个扇区不能被正常访问或不能正确记录数据，则该扇区也称为Defect Sector. 可以认为Bad sector 等同于 Defect sector. 从总的来说，某个硬盘只要有一部分存在缺陷，就称这个硬盘为Defect hard disk.

P-list (永久缺陷表)

现在的硬盘密度越来越高，单张盘片上存储的数据量超过40Gbytes. 硬盘厂家在生产盘片过程极其精密，但也极难做到100%的完美，硬盘盘面上或多或少存在一些缺陷。厂家在硬盘出厂前把所有的硬盘都进行低级格式化，在低级格式化过程中将自动找出所有defect track和defect sector，记录在P-list中。并且在对所有磁道和扇区的编号过程中，将skip（跳过）这些缺陷部分，让用户永远不能用到它们。这样，用户在分区、格式化、检查刚购买的新硬盘时，很难发现有问题。一般的硬盘都在P-list中记录有一定数量的defect, 少则数百，多则数以万计。如果是SCSI硬盘的话可以找到多种通用软件查看到P-list，因为各种牌子的SCSI硬盘使用兼容的SCSI指令集。而不同牌子不同型号的IDE硬盘，使用各自不同的指令集，想查看其P-list要用针对性的专业软件。

G-list (增长缺陷表)

用户在使用硬盘过程中，有可能会发现一些新的defect sector。按“三包”规定，只要出现一个defect sector，商家就应该为用户换或修。现在大容量的硬盘出现一个defect sector概率实在很大，这样的话硬盘商家就要为售后服务忙碌不已了。于是，硬盘厂商设计了一个自动修复机制，叫做Automatic Reallcation。有大多数型号的硬盘都有这样的功能：在对硬盘的读写过程中，如果发现一个defect sector，则自动分配一个备用扇区替换该扇区，并将该扇区及

其替换情况记录在G-list中。这样一来，少量的defect sector对用户的使用没有太大的影响。

也有一些硬盘自动修复机制的激发条件要严格一些，需要用某些软件来判断defect sector, 并通过某个端口（据说是50h）调用自动修复机制。比如常用的Lformat, ADM，DM中的Zero fill，Norton中的Wipeinfo和校正工具，西数工具包中的wddiag, IBM的DFT中的Erase等。这些工具之所以能在运行过后消除了一些“坏道”，很重要的原因就在这Automatic Reallcation（当然还有其它原因），而不能简单地概括这些“坏道”是什么“逻辑坏道”或“假坏道”。如果哪位被误导中毒太深的读者不相信这个事实，等他找到能查看G-list的专业工具后就知道，这些工具运行过后，G-list将会增加多少记录！“逻辑坏道”或“假坏道”有必要记录在G-list中并用其它扇区替换么？

当然，G-list的记录不会无限制，所有的硬盘都会限定在一定数量范围内。如火球系列限度是500，美钻二代的限度是636，西数BB的限度是508，等等。超过限度，Automatic Reallcation就不能再起作用。这就是为何少量的“坏道”可以通过上述工具修复（有人就概括为：“逻辑坏道”可以修复），而坏道多了不能通过这些工具修复（又有人概括为：“物理坏道”不可以修复）。

Bad track (坏道）

这个概念源于十多年前小容量硬盘（100M以下），当时的硬盘在外壳上都贴有一张小表格，上面列出该硬盘中有缺陷的磁道位置（新硬盘也有）。在对这个硬盘进行低级格式化时（如用ADM或DM 5.0等工具,或主板中的低格工具），需要填入这些Bad track的位置, 以便在低格过程中跳过这些磁道。现在的大容量硬盘在结构上与那些小容量硬盘相差极大，这个概念用在大容量硬盘上有点牵强。

国内很多刊物和网上文章中有几个概念：物理坏道，逻辑坏道，真坏道，假坏道，硬坏道，软坏道等但在国外的硬盘技术资料中没有找到对应的英文。

4.6数据备份介绍

虽然数据恢复技术可能将你的损失降到最低，但是，谁愿意在惶恐不安中，边祈祷边按下各类数据恢复软件的“Recover”按钮？也没有人喜欢面对满屏的16进制代码，带着侥幸的心理调整硬盘分区表Data区的信息；所以，最好的数据保全方法应该是备份。

硬盘里有三种东西:数据、应用程序和操作系统。你可备份硬盘中的所有东西，也可只备份数据。

进行完整的备份是最简单的方法，至少从恢复的角度看是这样。如果整个硬盘完全损坏，你不需要从头重装操作系统和应用程序；而是很快就可恢复运行。但是反过来，进行完整备份花的时间更长，要用到更多带有大容量存储空间的移动设备——即所谓外挂驱动器，如CD-R或Zip驱动器。

另一个方法是只备份最重要的东西：数据。你不能很轻易地替换文档、工作表或数据库。备份所有数据的最简单的方法是把所有数据保存到一个地方。

还应备份其它的重要文件，比如那些含有所有设置的文件，包括浏览器的书签（收藏夹）、cookie（含有密码）、地址簿、配置设置、数据项与报告表、模板、宏等。Windows 2000把这些文件都放到一个名叫“Documents And Settings”的新文件夹中，所以找起来很方便。该文件夹中不仅包括My Documents文件夹，还包括了部分关于配置设置、收藏夹和cookie的注册表信息。

第五章 NTFS相关领域技术介绍

5.1 RH8下最简单编译NTFS模块的方法

方法一

- 把附件中的Makefile.ntfs拷到/usr/src/linux-2.4/fs/ntfs下，然后make -f Makefile.ntfs就生成了ntfs.o。
- 在/lib/modules/2.4.18-14/kernel/fs下建立ntfs目录，将ntfs.o移至这个目录。
- 运行 depmod -a

说明：

这个方法是ClearMind@smth.org提出来的。只需单独编译ntfs.o，省时省力。

linux对ntfs的支持还很不成熟，请斟酌使用。

方法二

- 把附件中的Makefile.ntfsCopy到/usr/src/linux-2.4/fs/ntfs下。
- #cp Makefile.ntfs /usr/src/linux-2.4/fs/ntfs
- #cd /usr/src/linux-2.4/fs/ntfs
- #make -f Makefile.ntfs
- #mkdir /lib/modules/2.4.18-14/kernel/fs/ntfs
- #cp ntfs.o /lib/modules/2.4.18-14/kernel/fs/ntfs
- #cd /lib/modules/2.4.18-14/kernel/fs/ntfs
- #insmod ntfs.o

5.2 Ubuntu下安全读写NTFS分区格式文件

虽然在2001年LINUX就支持NTFS格式文件的读取，但是在写文件方面一直不尽人意，软件NTFS-3G可在LINUX下安全读写NTFS分区的文件。安装步骤如下：

1. 先安装FUSE

   http://flomertens.keo.in/debian/ntfs-3g/binary-i386/fuse-utils_2.5.3-1_i386.deb

   http://flomertens.keo.in/debian/ntfs-3g/binary-i386/libfuse2_2.5.3-1_i386.deb

   sudo dpkg -i libfuse2_2.5.3-1_i386.deb fuse-utils_2.5.3-1_i386.deb

2. 再安装NTFS-3G

   http://flomertens.keo.in/debian/ntfs-3g/binary-i386/ntfs-3g_20070714-BETA-1_i386.deb

   sudo dpkg -i ntfs-3g_20070714-BETA-1_i386.deb

3. 将NTFS分区加入到/etc/fstab

   首先检查一下您的机子上有几个NTFS分区，用如下命令即可：

   sudo fdisk -l | grep NTFS

   结果如下：

   /dev/hdc1 * 1 1275 10241406 7 HPFS/NTFS

   /dev/hdc5 1276 3825 20482843+ 7 HPFS/NTFS

说明/dev/hdc1和/dev/hdc5这两个分区是NTFS格式，接下来就将这两个分区加入到/etc/fstab中。

sudo vi /etc/fstab

在文件最后加入下面两行：

/dev/hdc1 /media/diskc ntfs-3g silent，umask=0，locale=en_US.utf8 0 0

/dev/hdc5 /media/diskd ntfs-3g silent，umask=0，locale=en_US.utf8 0 0

然后建立/media/diskc和/media/diskd两个目录

sudo mkdir /media/diskc

sudo mkdir /media/diskd

4. 因为NTFS-3G需要FUSE库的支持，所以需要先加载FUSE，编辑/etc/modules，以便机子启动的时候就加载FUSE

sudo vi /etc/modules

只需在文件最后加入fuse即可。

sudo cat /etc/modules

/etc/modules： kernel modules to load at boot time.

#

This file contains the names of kernel modules that should be loaded

at boot time， one per line. Lines beginning with “#” are ignored.

lp

psmouse

fglrx

fuse

5. 重新启动系统后就可以支持NTFS分区的文件读写了。如果想马上测试一下，可以执行以下步骤：

sudo modprobe fuse

sudo umount -a

sudo mount -a

sudo cd /media/diskd

sudo vi test.txt

6. 结束

5.3在Fat32中读写NTFS分区的数据

NTFS For Win98的主文件（共1个）：ntfs98ro.exe（795K）

下载地址：http://www.winternals.com/demos/ntfs98ro.exe

所需的其他NT或2K的系统文件（共7个）：Autochk.exe、Ntoskrnl.exe Ntdll.dll、Ntfs.sys、C_1252.nls、C_437.nls、L_intl.nls（Win2K中此7个文件共约3.2M）。其中，Ntfs.sys位于/winnt/system32/drivers中；其他文件均位于/winnt/system32中。

注意：是在安装后的NT/2K系统所在分区中，而非在NT/2K的安装盘中！

安装工作：

- 将NTFS For Win98的主文件和所需的七个系统文件均复制到Win98能识别的分区中。
- 执行ntfs98ro.exe，安装NTFS For Win98的主文件。
- 主文件安装成功后会自动执行其间的NTFS Configure（NTFS配置）程序，然后弹出一个名为“NTFS For Windows98 Configuration”的窗口。
- 在上面的文字框内输入（或选择）七个系统文件所在的完整路径（含盘符）；在下面的

文字框中输入你为NTFS分区在Win98环境下所分配盘符（不加冒号）。

注意：

此盘符一定不能和Win98下原有的所有盘符重合！比如在Win98下原有的最后一个分区为G盘，则为NTFS分区分配的盘符必须是H以后(含H) 的任意一个字母。否则会屏蔽掉原有的分区！此分配盘符只在Win98环境下有效；不影响NT/2K原有的分区结构。

- 根据提示，重新启动计算机既可。

5.4 DOS下访问NTFS

NTFSDOS是一个可以制作启动盘的工具，它所制作的启动盘的与众不同之处在于：虽然使用的是MS-DOS，但却可以读写NTFS格式文件系统，所有DOS命令都可以使用在NTFS格式系统上。

该工具可以在http://www.winternals.com/下载。

NTFSdos包括如下功能：

- 对NTFS文件系统的完全读写操作。
- MS-DOS下支持长文件名
- 简单实用的启动盘制作精灵
- Windows NT/2000/XP完全兼容

第六章 结论

（未完成，内容一直处于更新中）

参考文献

[1] 数据恢复技术（第二版）
[2] http://tb.blog.csdn.net/TrackBack.aspx?PostId=914238
[3] NTFS Documentation
[4] http://www.baidu.com
[5]http://www.google.com

附录A dos下访问ntfs分区，查找指定文件的源代码

```
.386
.model small
.stack 0200h
dseg segment USE16
buffer db 3fh*512 dup(?)
dap db 10h,00,01,00,00,00,00,00,00,00,00,00,00,00,00,00
secperclu db ?
MftSectorBase dd ?
SectorsPerFrs dd ?
curreFRS dd ?
namelength db 5
findfilename dw \'n\',\'t\',\'l\',\'d\',\'r\'
findend db \"No Find File!\",24h
findfile db \"File Find OK!\",24h
dseg ends

cseg segment USE16 ;16bit code segment
assume cs:cseg;ds:dseg;es:dseg
start proc
mov ax,dseg
mov ds,ax ;set segment address
mov es,ax
mov si,offset dap ;set pointer to Disk Address Packet
mov [si+6],ax ;set read sector buffer segmeng address
mov byte ptr [si+8],3fh ;set start lba
mov ah,47h
mov dl,80h
int 13h
mov ah,42h
int 13h
mov di,offset buffer ;set pointer to read buffer
mov ax,0aa55h
cmp [di+1feh],ax ;is Boot Record ID?
jnz exit

mov eax,5346544eh
cmp [di+3],eax ;Version is NTFS?
jnz exit

mov ecx,[di+40h] ;cluster per MFT FRS
neg cl
mov eax,1
shl eax,cl ;byte per FRS
```

```
movzx ebx,word ptr [di+0bh];byte per sector
xor edx,edx
div ebx ; eax = sectors per frs
mov SectorsPerFrs, eax
mov eax, [di+30h] ;MFT start cluster
movzx ebx, [di+0dh] ;sector per cluster
mul ebx
add eax,[di+1ch] ;HiddenSectors
mov MftSectorBase, eax ;MftSectorBase= MFT starting sector

mov eax,SectorsPerFrs
mov ebx,23 ;ebx=star frs(start user FRS#)
mul ebx
add eax,MftSectorBase ;FRSstartsector=23#FRS*SectorsPerFrs+MftSectorBase
mov curreFRS,eax ;save FRSstartsector

readfrs:
mov eax,curreFRS
mov ebx,SectorsPerFrs
add eax,ebx ;eax=Next FRS
mov curreFRS,eax
mov [si+2],bx ;set read sector count
mov [si+8],eax ;set start lba
mov ah,47h
mov dl,80h
int 13h
mov ah,42h
int 13h

mov eax,454c4946h
cmp eax,[di] ;is FRS single?
jnz nofrs ;loop exit

xor eax,eax
mov ax,offset buffer
call lacattribre
cmp eax,0
jz readfrs ;go to next FRS

mov ah,9 ;print find file
mov dx,offset findfile
int 21h
jmp exit

nofrs:
mov ah,9 ;print find file end
```

```
mov dx,offset findend
int 21h

exit:
mov ah,4ch
int 21h

;Find an file attribute in an FRS
lacattribre proc
push edi
push esi
push ecx
add ax,[eax+14h] ;FRS FirstAttribute
lacA:
mov ecx,0ffffffffh
cmp [eax],ecx ;is Attribute list end?
jz exitlac
mov ecx,30h ;is filename Attribute?
cmp [eax],ecx
jnz lacB

;find file attribute,cmp is find file name?
movzx ecx,namelength
cmp [eax+58h],cl ;+18h(size FRS attribe)+40h(length offset in filenameattribute)
jnz lacB
mov esi, eax
add si,5ah ;+18h+42h(filename offset in filename attribute)
mov di,offset findfilename
cld
repe cmpsw ; zero flag is set if equal
jnz exitlac ;no find file name
pop ecx
pop esi
pop edi
ret ;return eax=locate Attribute TypeCode

lacB:
add ax,[eax+4] ;next Attribute add eax,[eax+4] attrib length is 4B
jmp lacA

exitlac:
xor eax,eax ;no find file Attribute,set eax=0 exit
pop ecx
pop esi
pop edi
ret
```

```
lacattribre endp
cseg ends
end start
```

附录B Windows下ntfs文件恢复源代码

```
CMFTRecord::CMFTRecord()
{
	m_hDrive = 0;
	m_dwMaxMFTRecSize = 1023; // usual size
	m_pMFTRecord = 0;
	m_dwCurPos = 0;
	m_puchFileData = 0; // collected file data buffer
	m_dwFileDataSz = 0; // file data size , ie. m_pchFileData buffer length
	memset(&m_attrStandard,0,sizeof(ATTR_STANDARD));
	memset(&m_attrFilename,0,sizeof(ATTR_FILENAME));
	m_bInUse = false;;
}
CMFTRecord::~CMFTRecord()
{
	if(m_puchFileData)
		delete m_puchFileData;
	m_puchFileData = 0;
	m_dwFileDataSz = 0;
}
void CMFTRecord::SetDriveHandle(HANDLE hDrive)
{
	m_hDrive = hDrive;
}
int CMFTRecord::SetRecordInfo(LONGLONG  n64StartPos, DWORD dwRecSize, DWORD dwBytesPerCluster)
{
	if(!dwRecSize)
		return ERROR_INVALID_PARAMETER;
	if(dwRecSize%2)
		return ERROR_INVALID_PARAMETER;
	if(!dwBytesPerCluster)
		return ERROR_INVALID_PARAMETER;
	if(dwBytesPerCluster%2)
		return ERROR_INVALID_PARAMETER;
	m_dwMaxMFTRecSize = dwRecSize;
	m_dwBytesPerCluster = dwBytesPerCluster;
	m_n64StartPos =  n64StartPos;
	return ERROR_SUCCESS;
}
int CMFTRecord::ExtractFile(BYTE *puchMFTBuffer, DWORD dwLen, bool bExcludeData)
{
	if(m_dwMaxMFTRecSize > dwLen)
		return ERROR_INVALID_PARAMETER;
	if(!puchMFTBuffer)
```

```
        return ERROR_INVALID_PARAMETER;
NTFS_MFT_FILE      ntfsMFT;
NTFS_ATTRIBUTE   ntfsAttr;
BYTE *puchTmp = 0;
BYTE *uchTmpData =0;
DWORD dwTmpDataLen;
int nRet;
m_pMFTRecord = puchMFTBuffer;
m_dwCurPos = 0;
if(m_puchFileData)
        delete m_puchFileData;
m_puchFileData = 0;
m_dwFileDataSz = 0;
// read the record header in MFT table
memcpy(&ntfsMFT,&m_pMFTRecord[m_dwCurPos],sizeof(NTFS_MFT_FILE));
if(memcmp(ntfsMFT.szSignature,"FILE",4))
        return ERROR_INVALID_PARAMETER; // not the right signature
m_bInUse = (ntfsMFT.wFlags&0x01); //0x01         Record is in use
                                                  //0x02    Record is a directory
//m_dwCurPos = (ntfsMFT.wFixupOffset + ntfsMFT.wFixupSize*2);
m_dwCurPos = ntfsMFT.wAttribOffset;
do
{       // extract the attribute header
        memcpy(&ntfsAttr,&m_pMFTRecord[m_dwCurPos],sizeof(NTFS_ATTRIBUTE));
        switch(ntfsAttr.dwType) // extract the attribute data
        {
                // here I haven' implemented the processing of all the attributes.
                //  I have implemented attributes necessary for file & file data extraction
        case 0://UNUSED
                break;
        case 0x10: //STANDARD_INFORMATION
                nRet = ExtractData(ntfsAttr,uchTmpData,dwTmpDataLen);
                if(nRet)
                        return nRet;
                memcpy(&m_attrStandard,uchTmpData,sizeof(ATTR_STANDARD));
                delete uchTmpData;
                uchTmpData = 0;
                dwTmpDataLen = 0;
                break;
        case 0x30: //FILE_NAME
                nRet = ExtractData(ntfsAttr,uchTmpData,dwTmpDataLen);
                if(nRet)
                        return nRet;
                memcpy(&m_attrFilename,uchTmpData,dwTmpDataLen);
                delete uchTmpData;
                uchTmpData = 0;
```

```
            dwTmpDataLen = 0;
            break;
        case 0x40: //OBJECT_ID
            break;
        case 0x50: //SECURITY_DESCRIPTOR
            break;
        case 0x60: //VOLUME_NAME
            break;
        case 0x70: //VOLUME_INFORMATION
            break;
        case 0x80: //DATA
            if(!bExcludeData)
            {
                nRet = ExtractData(ntfsAttr,uchTmpData,dwTmpDataLen);
                if(nRet)
                    return nRet;

                if(!m_puchFileData)
                {
                    m_dwFileDataSz = dwTmpDataLen;
                    m_puchFileData = new BYTE[dwTmpDataLen];
                    memcpy(m_puchFileData,uchTmpData,dwTmpDataLen);
                }
                else
                {
                    puchTmp = new BYTE[m_dwFileDataSz+dwTmpDataLen];
                        memcpy(puchTmp,m_puchFileData,m_dwFileDataSz);                    memcpy
(puchTmp+m_dwFileDataSz,uchTmpData,dwTmpDataLen);
                    m_dwFileDataSz += dwTmpDataLen;
                    delete m_puchFileData;
                    m_puchFileData = puchTmp;
                }
                delete uchTmpData;
                uchTmpData = 0;
                dwTmpDataLen = 0;
            }
            break;
        case 0x90: //INDEX_ROOT
        case 0xa0: //INDEX_ALLOCATION
            // todo: not implemented to read the index mapped records
            return ERROR_SUCCESS;
            continue;
            break;
        case 0xb0: //BITMAP
            break;
        case 0xc0: //REPARSE_POINT
```

```
                break;
        case 0xd0: //EA_INFORMATION
                break;
        case 0xe0: //EA
                break;
        case 0xf0: //PROPERTY_SET
                break;
        case 0x100: //LOGGED_UTILITY_STREAM
                break;
        case 0x1000: //FIRST_USER_DEFINED_ATTRIBUTE
                break;
        case 0xFFFFFFFF: // END
                if(uchTmpData)
                        delete uchTmpData;
                uchTmpData = 0;
                dwTmpDataLen = 0;
                return ERROR_SUCCESS;
        default:
                break;
        };
        m_dwCurPos += ntfsAttr.dwFullLength; // go to the next location of attribute
    }
    while(ntfsAttr.dwFullLength);
    if(uchTmpData)
        delete uchTmpData;
    uchTmpData = 0;
    dwTmpDataLen = 0;
    return ERROR_SUCCESS;
}
int CMFTRecord::ExtractData(NTFS_ATTRIBUTE ntfsAttr, BYTE *&puchData, DWORD &dwDataLen)
{
    DWORD dwCurPos = m_dwCurPos;
    if(!ntfsAttr.uchNonResFlag)
    {// residence attribute, this always resides in the MFT table itself
        puchData = new BYTE[ntfsAttr.Attr.Resident.dwLength];
            dwDataLen = ntfsAttr.Attr.Resident.dwLength; memcpy(puchData,&m_pMFTRecord
[dwCurPos+ntfsAttr.Attr.Resident.wAttrOffset],dwDataLen);
    }
    else
    {// non-residence attribute, this resides in the other part of the physical drive
        if(!ntfsAttr.Attr.NonResident.n64AllocSize) // i don't know Y, but fails when its zero
                ntfsAttr.Attr.NonResident.n64AllocSize = (ntfsAttr.Attr.NonResident.n64EndVCN -
ntfsAttr.Attr.NonResident.n64StartVCN) + 1;
        dwDataLen = ntfsAttr.Attr.NonResident.n64RealSize;
        puchData = new BYTE[ntfsAttr.Attr.NonResident.n64AllocSize];
        BYTE chLenOffSz; // length & offset sizes
```

```
            BYTE chLenSz; // length size
            BYTE chOffsetSz; // offset size
            LONGLONG n64Len, n64Offset; // the actual lenght & offset
            LONGLONG n64LCN =0; // the pointer pointing the actual data on a physical disk
            BYTE *pTmpBuff = puchData;
            int nRet;
            dwCurPos += ntfsAttr.Attr.NonResident.wDatarunOffset;;
            for(;;)
            {
                  chLenOffSz = 0;
                  memcpy(&chLenOffSz,&m_pMFTRecord[dwCurPos],sizeof(BYTE));
                  dwCurPos += sizeof(BYTE);
                  if(!chLenOffSz)
                        break;
                  chLenSz           = chLenOffSz & 0x0F;
                  chOffsetSz = (chLenOffSz & 0xF0) >> 4;
                  n64Len = 0;
                  memcpy(&n64Len,&m_pMFTRecord[dwCurPos],chLenSz);
                  dwCurPos += chLenSz;
                  n64Offset = 0;
                  memcpy(&n64Offset,&m_pMFTRecord[dwCurPos],chOffsetSz);
                  dwCurPos += chOffsetSz;
                  ////// if the last bit of n64Offset is 1 then its -ve so u got to make it -ve /////
                  if((((char*)&n64Offset)[chOffsetSz-1])&0x80)
                        for(int i=sizeof(LONGLONG)-1;i>(chOffsetSz-1);i--)
                              ((char*)&n64Offset)[i] = 0xff;
                  n64LCN += n64Offset;
                  n64Len *= m_dwBytesPerCluster;
                  ///// read the actual data ////////////////////////////////////////
                  /// since the data is available out side the MFT table, physical drive should be accessed
                  nRet = ReadRaw(n64LCN,pTmpBuff,(DWORD&)n64Len);
                  if(nRet)
                        return nRet;
                  pTmpBuff += n64Len;
            }
      }
      return ERROR_SUCCESS;
}
int CMFTRecord::ReadRaw(LONGLONG n64LCN, BYTE *chData, DWORD &dwLen)
{
      int nRet;
      LARGE_INTEGER n64Pos;
      n64Pos.QuadPart = (n64LCN)*m_dwBytesPerCluster;
      n64Pos.QuadPart += m_n64StartPos;
      //   data is available in the relative sector from the begining od the drive
      //    so point that data
```

```
	nRet = SetFilePointer(m_hDrive,n64Pos.LowPart,&n64Pos.HighPart,FILE_BEGIN);
	if(nRet == 0xFFFFFFFF)
		return GetLastError();
	BYTE *pTmp		= chData;
	DWORD dwBytesRead	=0;
	DWORD dwBytes		=0;
	DWORD dwTotRead		=0;
	while(dwTotRead <dwLen)
	{
		// v r reading a cluster at a time
		dwBytesRead = m_dwBytesPerCluster;
		// this can not read partial sectors
		nRet = ReadFile(m_hDrive,pTmp,dwBytesRead,&dwBytes,NULL);
		if(!nRet)
			return GetLastError();
		dwTotRead += dwBytes;
		pTmp += dwBytes;
	}
	dwLen = dwTotRead;
	return ERROR_SUCCESS;
}
CNTFSDrive::CNTFSDrive()
{
	m_bInitialized = false;
	m_hDrive = 0;
	m_dwBytesPerCluster = 0;
	m_dwBytesPerSector = 0;
	m_puchMFTRecord = 0;
	m_dwMFTRecordSz = 0;
	m_puchMFT = 0;
	m_dwMFTLen = 0;
	m_dwStartSector = 0;
}
CNTFSDrive::~CNTFSDrive()
{
	if(m_puchMFT)
		delete m_puchMFT;
	m_puchMFT = 0;
	m_dwMFTLen = 0;
}
void CNTFSDrive::SetDriveHandle(HANDLE hDrive)
{
	m_hDrive = hDrive;
	m_bInitialized = false;
}
void CNTFSDrive::SetStartSector(DWORD dwStartSector, DWORD dwBytesPerSector)
```

```
{
	m_dwStartSector = dwStartSector;
	m_dwBytesPerSector = dwBytesPerSector;
}
int CNTFSDrive::Initialize()
{
	NTFS_PART_BOOT_SEC ntfsBS;
	DWORD dwBytes;
	LARGE_INTEGER n84StartPos;

	n84StartPos.QuadPart = (LONGLONG)m_dwBytesPerSector*m_dwStartSector;

	// point the starting NTFS volume sector in the physical drive
	DWORD dwCur = SetFilePointer(m_hDrive,n84StartPos.LowPart,&n84StartPos.HighPart,FILE_BEGIN);
	int nRet = ReadFile(m_hDrive,&ntfsBS,sizeof(NTFS_PART_BOOT_SEC),&dwBytes,NULL);
	if(!nRet)
		return GetLastError();
	if(memcmp(ntfsBS.chOemID,"NTFS",4)) // check whether it is realy ntfs
		return ERROR_INVALID_DRIVE;
	m_dwBytesPerCluster = ntfsBS.bpb.uchSecPerClust * ntfsBS.bpb.wBytesPerSec;
	if(m_puchMFTRecord)
		delete m_puchMFTRecord;
	m_dwMFTRecordSz = 0x01<<((-1)*((char)ntfsBS.bpb.nClustPerMFTRecord));
	m_puchMFTRecord = new BYTE[m_dwMFTRecordSz];
	m_bInitialized = true;
	// MFTRecord of MFT is available in the MFTRecord variable
	//   load the entire MFT using it
	nRet = LoadMFT(ntfsBS.bpb.n64MFTLogicalClustNum);
	if(nRet)
	{
		m_bInitialized = false;
		return nRet;
	}
	return ERROR_SUCCESS;
}
int CNTFSDrive::LoadMFT(LONGLONG nStartCluster)
{
	DWORD dwBytes;
	int nRet;
	LARGE_INTEGER n64Pos;
	if(!m_bInitialized)
		return ERROR_INVALID_ACCESS;
	CMFTRecord cMFTRec;
	wchar_t uszMFTName[10];
	mbstowcs(uszMFTName,"$MFT",10);
	// NTFS starting point
```

```
	n64Pos.QuadPart = (LONGLONG)m_dwBytesPerSector*m_dwStartSector;
	// MFT starting point
	n64Pos.QuadPart += (LONGLONG)nStartCluster*m_dwBytesPerCluster;
	//  set the pointer to the MFT start
	nRet = SetFilePointer(m_hDrive,n64Pos.LowPart,&n64Pos.HighPart,FILE_BEGIN);
	if(nRet == 0xFFFFFFFF)
		return GetLastError();

	/// reading the first record in the NTFS table.
	//   the first record in the NTFS is always MFT record
	nRet = ReadFile(m_hDrive,m_puchMFTRecord,m_dwMFTRecordSz,&dwBytes,NULL);
	if(!nRet)
		return GetLastError();

	// now extract the MFT record just like the other MFT table records
	cMFTRec.SetDriveHandle(m_hDrive);
	cMFTRec.SetRecordInfo((LONGLONG)m_dwStartSector*m_dwBytesPerSector, m_dwMFTRecordSz,m_dwBytesPerCluster);
	nRet = cMFTRec.ExtractFile(m_puchMFTRecord,dwBytes);
	if(nRet)
		return nRet;
	if(memcmp(cMFTRec.m_attrFilename.wFilename,uszMFTName,8))
		return ERROR_BAD_DEVICE; // no MFT file available

	if(m_puchMFT)
		delete m_puchMFT;
	m_puchMFT = 0;
	m_dwMFTLen = 0;
	// this data(m_puchFileData) is special since it is the data of entire MFT file
	m_puchMFT = new BYTE[cMFTRec.m_dwFileDataSz];
	m_dwMFTLen = cMFTRec.m_dwFileDataSz;

	// store this file to read other files
	memcpy(m_puchMFT, cMFTRec.m_puchFileData, m_dwMFTLen);

	return ERROR_SUCCESS;
}
int CNTFSDrive::Read_File(DWORD nFileSeq, BYTE *&puchFileData, DWORD &dwFileDataLen)
{
	int nRet;
	if(!m_bInitialized)
		return ERROR_INVALID_ACCESS;
	CMFTRecord cFile;
	// point the record of the file in the MFT table
	memcpy(m_puchMFTRecord,&m_puchMFT[nFileSeq*m_dwMFTRecordSz],m_dwMFTRecordSz);
	// Then extract that file from the drive
	cFile.SetDriveHandle(m_hDrive);
```

```
	cFile.SetRecordInfo((LONGLONG)m_dwStartSector*m_dwBytesPerSector, m_dwMFTRecordSz,m_dwBytesPerCluster);
	nRet = cFile.ExtractFile(m_puchMFTRecord,m_dwMFTRecordSz);
	if(nRet)
		return nRet;
	puchFileData = new BYTE[cFile.m_dwFileDataSz];
	dwFileDataLen = cFile.m_dwFileDataSz;
	// pass the file data, It should be deallocated by the caller
	memcpy(puchFileData,cFile.m_puchFileData,dwFileDataLen);
	return ERROR_SUCCESS;
}
int CNTFSDrive::GetFileDetail(DWORD nFileSeq, ST_FILEINFO &stFileInfo)
{
	int nRet;
	if(!m_bInitialized)
		return ERROR_INVALID_ACCESS;

	if((nFileSeq*m_dwMFTRecordSz+m_dwMFTRecordSz) >= m_dwMFTLen)
		return ERROR_NO_MORE_FILES;
	CMFTRecord cFile;
	// point the record of the file in the MFT table
	memcpy(m_puchMFTRecord,&m_puchMFT[nFileSeq*m_dwMFTRecordSz],m_dwMFTRecordSz);

	// read the only file detail not the file data
	cFile.SetDriveHandle(m_hDrive);
	cFile.SetRecordInfo((LONGLONG)m_dwStartSector*m_dwBytesPerSector, m_dwMFTRecordSz,m_dwBytesPerCluster);
	nRet = cFile.ExtractFile(m_puchMFTRecord,m_dwMFTRecordSz,true);
	if(nRet)
		return nRet;
	// set the struct and pass the struct of file detail
	memset(&stFileInfo,0,sizeof(ST_FILEINFO));
	wcstombs(stFileInfo.szFilename,cFile.m_attrFilename.wFilename,_MAX_PATH);
	stFileInfo.dwAttributes = cFile.m_attrFilename.dwFlags;
	stFileInfo.n64Create = cFile.m_attrStandard.n64Create;
	stFileInfo.n64Modify = cFile.m_attrStandard.n64Modify;
	stFileInfo.n64Access = cFile.m_attrStandard.n64Access;
	stFileInfo.n64Modfil = cFile.m_attrStandard.n64Modfil;
	stFileInfo.n64Size	= cFile.m_attrFilename.n64Allocated;
	stFileInfo.n64Size	/= m_dwBytesPerCluster;
	stFileInfo.n64Size	= (!stFileInfo.n64Size)?1:stFileInfo.n64Size;
	stFileInfo.bDeleted = !cFile.m_bInUse;
	return ERROR_SUCCESS;
}
```

后记

本文的内容有网络上搜集的，也有作者自己研究成果。本文不涉及金钱利益，欢迎读者转载，转载时请保证该文章的完成性。

写本文的初衷是给自己一个比较全面的NTFS资料。目前网络上公开的介绍NTFS文件系统的中文文献很少，或者不全，为了和大家分享我的成果，于是写了这篇论文。

中国的网络已经发展了这么多年了，但是目前的情况比以前要发展的缓慢的多。读者也许发现在国内的搜索引擎搜索一个关键字的时候，同一篇文章的出现率十分的高。这不能不算是中国网络发展的悲哀！遥想当年，很多人都很乐意发表自己的成果，把自己的心得贴出来和大家分享，那是多么美好的时期。如今，得情况是有了成果不愿意公开。这也是中国近年计算机网络技术发展缓慢的原因之一。几句牢骚，如果读者不喜欢请忘记吧。

如果您读了这片文章，希望您能对我多提批评和意见，指出文中的错误。我也希望能够改正错误，得到正确的知识。

由于本文涉及的NTFS相关内容不是很多，所以如果您有什么想法或者成果，希望能和我讨论，一起研究。我的联系方式beiyuly@gmail.com，http://beiyu.bokee.com。

愿你和我一起进步！

Beiyu
2007-4-26

Author: Beiyu
Home Page: http://beiyu.bokee.com
Email: beiyuly@gmail.com
Date: 2007-4-26