内部碎片与小文件优化

Table of Contents

目录

副标题1：文件存入文件系统后占用多大？

副标题2：文件系统的微观结构

上篇「系统中的大多数文件有多大？」提到，文件系统中大部分文件其实都很小，中位数一直稳定在 4K 左右，而且这个数字并没有随着存储设备容量的增加而增大。但是存储设备的总体容量实际是在逐年增长的，，总容量增加而文件大小中位数不变的原因，可能是以下两种情况：

1. 文件数量在增加
2. 大文件的大小在增加

实际上可能是这两者综合的结果。这种趋势给文件系统设计带来了越来越多的挑战，因为我们不能单纯根据平均文件大小来增加块大小（block size）优化文件读写。微软的文件系统（FAT系和 NTFS）使用「簇（cluster）」这个概念管理文件系统的可用空间分配，在 Unix 系文件系统中有类似的块（block）的概念，只不过称呼不一样。现代文件系统都有这个块大小或者簇大小的概念，从而基本的文件空间分配可以独立于硬件设备本身的扇区大小。块大小越大，单次分配空间越

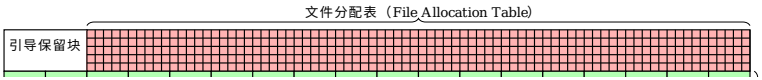
大，文件系统所需维护的元数据越小，复杂度越低，实现起来也越容易。 而块大小越小，越能节约可用空间，避免内部碎片造成的浪费，但是跟踪空间所需的元数据也越复杂。

按块分配的文件系统

具体块/簇大小对文件系统设计带来什么样的挑战？我们先来看一下（目前还在用的）最简单的文件系统怎么存文件的吧：

FAT系文件系统与簇大小

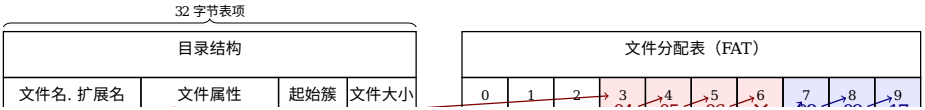
在 FAT 系文件系统(FAT12/16/32/exFAT)中，整个存储空间除了一些保留扇区之外，被分为两大块区域，看起来类似这样：



前一部分区域放文件分配表（File Allocation Table），后一部分是实际存储文件和目录的数据区。 数据区被划分成「簇（cluster）」，每个簇是一到多个连续扇区，然后文件分配表中表项的数量 决定了后面可用空间的簇的数量。文件分配表（FAT）在 FAT 系文件系统中这里充当了两个重要作用：

1. **宏观尺度** ：从 CHS 地址映射到线性的簇号地址空间，管理簇空间分配。空间分配器可以扫描 FAT 判断哪些簇处于空闲状态，那些簇已经被占用，从而分配空间。
2. **微观尺度** ：对现有文件，FAT 表中的记录形成一个单链表结构，用来寻找文件的所有已分配簇地址。

比如在根目录中有 4 个不同大小文件的 FAT16 中，使用 512 字节的簇大小的文件系统，其根目录结构和 FAT 表可能看起来像下图这样：



目录结构中的文件记录是固定长度的，其中保存 8.3 长度的文件名，一些文件属性（修改日期和时间、 隐藏文件之类的），文件大小的字节数，和一个起始簇号。起始簇号在 FAT 表中引出一个簇号的单链表， 顺着这个单链表能找到存储文件内容的所有簇。

直观上理解，FAT表像是数据区域的缩略图，数据区域有多少簇，FAT表就有多少表项。 FAT系文件系统中每个簇有多大，由文件系统总容量，以及 FAT 表项的数量限制。 我们来看一下微软文件系统默认格式化的簇大小（数据来源）：

Volume Size	FA	FA	ex	NT
< 8 MiB			4KiB	4KiB
8 MiB – 16 MiB	512B		4KiB	4KiB
16 MiB – 32 MiB	512B	512B	4KiB	4KiB
32 MiB – 64 MiB	1KiB	512B	4KiB	4KiB
64 MiB – 128 MiB	2KiB	1KiB	4KiB	4KiB
128 MiB – 256 MiB	4KiB	2KiB	4KiB	4KiB
256 MiB – 512 MiB	8KiB	4KiB	32KiB	4KiB
512 MiB – 1 GiB	16KiB	4KiB	32KiB	4KiB
1 GiB – 2 GiB	32KiB	4KiB	32KiB	4KiB
2 GiB – 4 GiB	64KiB	4KiB	32KiB	4KiB
4 GiB – 8 GiB		4KiB	32KiB	4KiB
8 GiB – 16 GiB		8KiB	32KiB	4KiB
16 GiB – 32 GiB		16KiB	32KiB	4KiB
32 GiB – 16TiB			128KiB	4KiB
16 TiB – 32 TiB			128KiB	8KiB
32 TiB – 64 TiB			128KiB	16KiB
64 TiB – 128 TiB			128KiB	32KiB
128 TiB – 256			128KiB	64KiB

TiB
> 256 TiB

用于软盘的时候 FAT12 的簇大小直接等于扇区大小 512B ，在容量较小的 FAT16 上也是如此。 FAT12 和 FAT16 都被 FAT 表项的最大数量限制（分别是 4068 和 65460 ），FAT 表本身不会太大。 所以上表中可见，随着设备容量增加， FAT16 需要增加每簇大小，保持同样数量的 FAT 表项。

到 FAT32 和 exFAT 的年代，FAT 表项存储 32bit 的簇指针，最多能有接近 4G 个数量的 FAT 表项，从而表项数量理应不再限制 FAT 表大小，使用和扇区大小同样的簇大小。不过事实上， 簇大小仍然根据设备容量增长而增大。 FAT32 上 256MiB 到 8GiB 的范围内使用 4KiB 簇大小，随后簇大小开始增加；在 exFAT 上 256MiB 到 32GiB 使用 32KiB 簇大小，随后增加到 128KiB 。

FAT 系的簇大小是可以由用户在创建文件系统时指定的，大部分普通用户会使用系统根据存储设备容量推算的默认值， 而存储设备的生产厂商则可以根据底层存储设备的特性决定一个适合存储设备的簇大小。在选择簇大小时， 要考虑取舍，较小的簇意味着同样容量下更多的簇数，而较大的簇意味着更少的簇数，取舍在于：

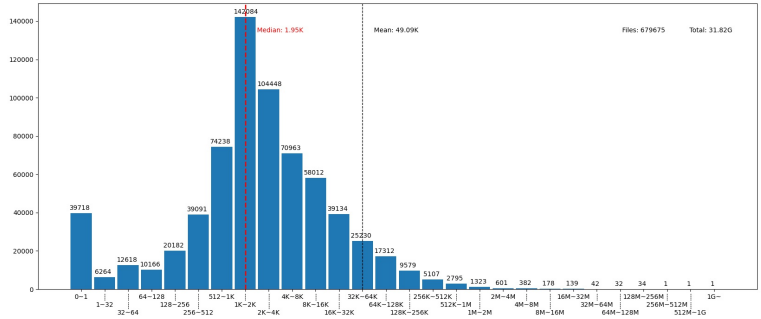
较小的簇： 优势是存储大量小文件时，降低 **内部碎片（Internal fragmentation）** 的程度，带来更多可用空间。劣势是更多 **外部碎片（External fragmentation）** 导致访问大文件时

来回跳转降低性能，并且更多簇数也导致簇分配器的性能降低。

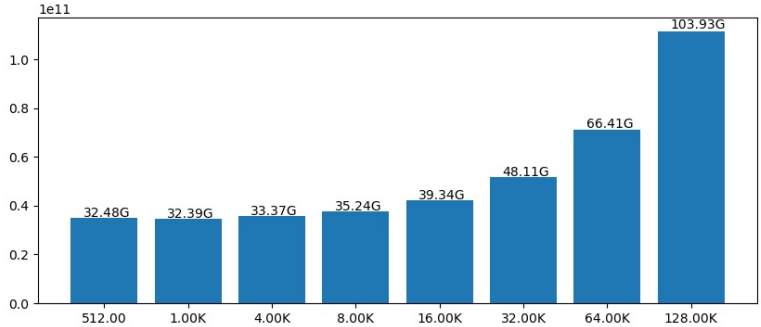
较大的簇: 优势是避免 **外部碎片** 导致的性能损失，劣势是 **内部碎片** 带来的低空间利用率。

FAT 系文件系统使用随着容量增加的簇大小，导致的劣势在于极度浪费存储空间。如果文件大小是满足随机分布， 那么大量文件平均而言，每个文件将有半个簇的未使用空间，比如假设一个 64G 的 exFAT 文件系统中存有 8000 个文件，使用 128KiB 的簇大小，那么平均下来大概会有 500MiB 的空间浪费。实际上如前文 系统中的大多数文件有多大？ 所述，一般系统中的文件大小并非随机分布，而是大多数都在大约 1KiB~4KiB 的范围内，从而造成的空间浪费更为严重。

可能有人想说「现在存储设备的容量都那么大了，浪费一点点存储空间换来读写性能的话也没什么坏处嘛」， 于是要考察加大簇大小具体会浪费多少存储空间。借用前文中统计文件大小的工具和例子， 比如我的文件系统中存有 31G 左右的文件，文件大小分布符合下图的样子：



假如把这些文件存入不同簇大小的 FAT32 中，根据簇大小，最终文件系统占用空间是下图：



在较小的簇大小时，文件系统占用接近于文件总大小 31G ，而随着簇大小增长，到使用 128KiB 簇大小的时候空间占用徒增到 103.93G ，是文件总大小的 3.35 倍。如此大的空间占用源自于目标文件系统 中有大量小文件，每个不足一簇的小文件都要占用完整一簇的大小。可能注意到上图 512B 的簇大小时整个文件系统占用反而比 1KiB 簇大小时的更大，这是因为 512B 簇大小的时候 FAT 表本身的占用更大。具体数字如下表：

FA 表项	据 簇

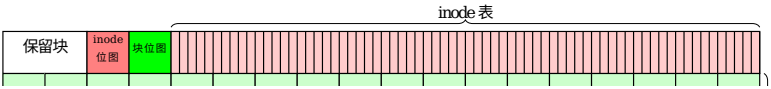
簇大小	数	总簇数	占用	数	FAT簇数
512.00	63.95M	64.95M	32.48G	63.95M	511.61K
1.00K	32.14M	32.39M	32.39G	32.14M	128.55K
4.00K	8.33M	8.34M	33.37G	8.33M	8.33K
8.00K	4.40M	4.41M	35.24G	4.40M	2.20K
16.00K	2.46M	2.46M	39.34G	2.46M	630.00
32.00K	1.50M	1.50M	48.11G	1.50M	193.00
64.00K	1.04M	1.04M	66.41G	1.04M	67.00
128.00K	831.40K	831.45K	103.93G	831.40K	26.00

FAT 系文件系统这种对簇大小选择的困境来源在于， FAT 试图用同一个数据结构——文件分配表——同时管理 **宏观尺度** 的可用空间分配和 **微观尺度** 的文件寻址，这产生两头都难以兼顾的矛盾。 NTFS 和其它 Unix-like 系统的文件系统都使用块位图(block bitmap)跟踪可用空间分配， 将宏观尺度的空间分配问题和微观尺度的文件寻址问题分开解决，从而在可接受的性能下允许更小的簇大小和更多的簇数。

传统 Unix 文件系统的块映射

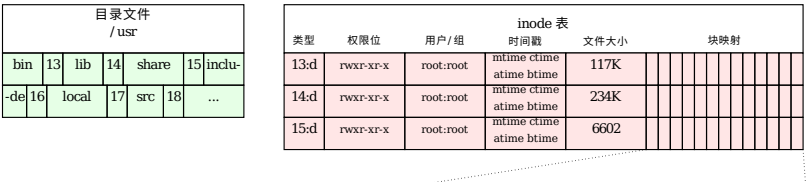
传统 Unix 文件系统（下称 UFS）和它的继任者们，包括 Linux 的 ext2/3 ， FreeBSD 的 FFS 等文件系统，使用在 inode 中记录块映射（bmap, block mapping）

的方式记录文件存储的地址范围。 术语上 UFS 中所称的「块（block）」等价于微软系文件系统中所称的「簇（cluster）」， 都是对底层存储设备中扇区寻址的抽象。



上图乍看和 FAT 总体结构很像，实际上重要的是「inode表」和「数据块」两大区域分别所占的比例。FAT 系文件系统中，每个簇需要在 FAT 表中有一个表项，所以 FAT 表的大小是每簇大小占 2字节 （FAT16）或 4 字节（FAT32/exFAT）。假设 exFAT 用 32K 簇大小的话， FAT 表整体大小与数据区的比例大约是 4:32K 。UFS 中，在创建文件系统时 `mkfs` 会指定一个 `bytes-per-inode` 的比例，比如 mkfs.ext4 默认的 `-i bytes-per-inode` 是 32K 于是每 32K 数据空间分配一个 inode ，而每个 inode 在 ext4 占用 256 字节，于是 inode 空间与数据块空间的比例大约是 256:32K 。宏观上，FAT 表是在 FAT 文件系统中地址前端一段连续空间；而 UFS 中 inode 表的位置 **不一定** 是在存储设备地址范围前端连续的空间，至于各个 UFS 如何安排 inode 表与数据块的宏观布局可能今后有空再谈，本文所关心的只是 inode 表中存放 inode 独立于数据块的存储空间，两者的比例在创建文件系统时固定。

UFS 与 FAT 文件系统一点非常重要的区别在于：Unix 文件系统中 **文件名不属于 inode 记录的文件元数据**。FAT 系文件系统中文件元数据存储在目录结构中，每个目录表项代表一个文件（除了 VFAT 的长文件名用隐藏目录表项），占用 32 字节，引出一个单链表表达文件存储地址；在 UFS 中，目录内容和 inode 表中的表项和文件地址的样子像是这样：



UFS 中每个目录文件的内容可以看作是单纯的（文件名：inode号）构成的数组，最早 Unix v7 的文件系统中文件名长度被限制在 14 字节，后来很快就演变成可以接受更长的文件名只要以 `\0` 结尾。关于文件的元数据信息，比如所有者和权限位这些，文件元数据并不记录在目录文件中，而是记录在长度规整的 inode 表中。inode 表中 inode 记录的长度规整这一点非常重要，因

为知道了 inode 表的位置和 inode 号，可以直接算出 inode 记录在存储设备上的地址，从而快速定位到所需文件的元数据信息。 在 inode 记录的末尾有个固定长度的块映射表，填写文件的内容的块地址。

因为 inode 记录的长度固定，从而 inode 记录末尾位置得到块指针数组的长度也是固定并且有限的， 在 Unix v7 FS 中这个数组可以记录 13 个地址，在 ext2/3 中可以记录 15 个地址。前文说过，文件系统中大部分文件大小都很小，而少数文件非常大，于是 UFS 中使用间接块指针的方案，用有限长度的数组表达任意大小的文件。

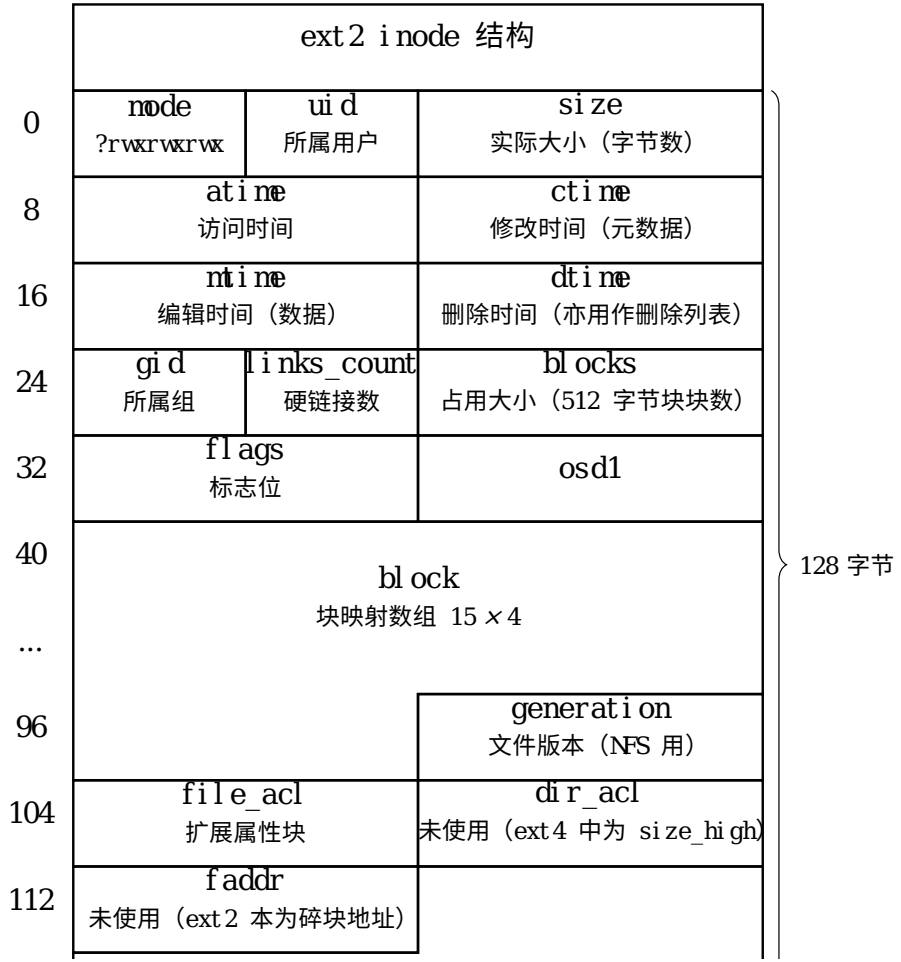
在 UFS 的 inode 中可以存 13 个地址，其中前 10 个地址用于记录「直接块指针（direct block address）」。当文件大小大于 10 块时，从第 11 块开始，分配一个「一级间接块（level 1 indirect block）」，其位置写在 inode 中第 11 个块地址上，间接块中存放块指针数组。 假设块大小是 4K 而指针大小是 4 字节，那么一级间接块可以存放 1024 个直接块指针。当文件大小超过 1034(=1024+10) 时，再分配一个「二级间接块（level 2 indirect block）」， 存在 inode 中的第 12 个块地址上，二级间接块中存放的是一级间接块的地址，形成一个两层的指针树。 同理，当二级间接块也不够用的时候，分配一个「三级间接块（level 3 indirect block）」， 三级间接块本身的地址存在 inode 中最后第 13 个块地址位置上，而三级间接块内存放指向二级间接块的指针， 形成一个三层的指针树。 UFS 的 inode 一共有 13 个块地址槽，于是不存在四级间接块了， 依靠上

述最多三级的间接块构成的指针树，如果是 4KiB 块大小的话，每个 inode 最多可以有 \(10+1024+1024^2+1024^3 = 1074791434\) 块，最大支持超过 4GiB 的文件大小。

UFS 使用这种 inode 中存储块映射引出间接块树的形式存储文件块地址，这使得 UFS 中定位到文件的 inode 之后查找文件存储的块比 FAT 类的文件系统快，因为不再需要去读取 FAT 表。这种方式另一个特征是，当文件较大时，读写文件前段部分的数据，比如 inode 中记录的前10块直接块地址的时候，比随后 10~1024 块一级间接块要快一些，同样的访问一级间接块中的数据也比二级和三级间接块要快一些。一些 Unix 工具比如 `file` 判断文件内容的类型只需要读取文件前段的内容，在这种记录方式下也可以比较高效。

以 ext2/3 为例，补充一下扩展属性和希疏文件

ext2/3 的 inode 存储方式基本上类似上述 UFS ，具体到它们的 inode 结构而言， ext2/3 中每个 inode 占用 128 字节，其中有 60 字节存储块映射，可以存放 12 个直接块指针和三级间接块指针。 详细的 ext2 inode 结构可见下图，结构来自 ext2 文档 。



结构中 osd1 和 osd2 两大块被定义为是系统实现相关（OS Dependent）的区域，具体到 Linux 上的 ext2/3 实现中 osd2 的部分区域被拿来保存 uid/gid/size/blocks 等位数不够的数据的高位，具体参考后文 ext4 的 inode 结构。

传统上 Unix 的文件系统支持扩展属性（xattr, eXtended ATTRibutes）和希疏文件（sparse files）这两个比较少用的高级特性，每个 Unix 分支的 UFS 实现在

实现这两个特性的方面都略有不同， 所以拿 ext2/3 来举例说明一下扩展属性和希疏文件的存储方式。

首先扩展属性是保存在 inode 中，文件系统本身不太关心，但是对系统别的部分而言需要保存的关于文件的一些属性。 一开始扩展属性主要是保存 POSIX 访问控制列表（ACL），不同于普通的 POSIX 权限位（permissions flags）， POSIX ACL 提供了更细粒度的文件权限的控制。后来扩展属性也用来存储 NFS 、SELinux 或者别的子系统用的属性。不同于「权限位」或者「修改时间」这些文件系统内置属性是保存在固定的 inode 结构体中，扩展属性在文件系统 API 中是以键值对（key-value pair）的方式存储的。

在 ext2/3 的 inode 中如果文件有扩展属性，是保存在额外的扩展属性块中，然后块指针写在 `inode.i_file_acl` 里面。扩展属性块是文件系统块大小，只能有一块，而且 API 限制所有扩展属性「键值对」的大小加起来不能超过 4KiB 。这个限制使得 ext2/3 有了对扩展属性支持不佳的评价。

支持扩展属性的需求一开始不那么紧迫，随着 ext2 被部署到企业级应用中，对扩展属性的需求逐渐紧迫起来。 像 POSIX ACL 和 SELinux 所需的扩展属性有个特点是它们经常递归地设置在文件夹树上，很多文件会直接继承父级文件夹设置的扩展属性。后来 ext2 有了个 ea_inode 的特性，用单独的特殊 inode 保存扩展属性，这个 inode 不存在于任何文件夹，其文件内容是引用它的文件的扩展属性，很多普通文件可以共享一个 ea_inode 来表达相同的扩展属性。

希疏文件（sparse files）是文件地址范围内有部分地址空间没有分配对应存储块的文件， 常用于存储磁盘镜像文件之类地址范围很大，而有很多空洞的文件。 在 ext2/3 的块映射中，对希疏文件如果有文件地址没有分配块，则把该块的指针存成 0 的方式表达。 可见在表达希疏文件的时候， ext2/3 虽然不需要分配块，但是仍然要存储空指针。

总体而言对扩展属性和希疏文件的支持一开始并不在 ext2/3 文件系统的原始设计中， 这两个较少使用的特性都是后来增加的，如果大量使用这两个特性可能带来一些性能问题。

FFS 中的整块与碎块设计

FreeBSD 用的 FFS 基于传统 UFS 的存储方式，为了对抗比较小的块大小导致块分配器的性能损失， FFS 创新的使用两种块大小记录文件块，在此我们把两种块大小分别叫整块（block）和碎块（fragment）。 整块和碎块的大小比例最多是 8:1，也可以是 4:1 或者 2:1，比如可以使用 4KiB 的整块和 1KiB 的碎块，或者用 32KiB 的整块并配有 4KiB 的碎块。写文件时先把末端不足一个整块的内容写入碎块中， 之后多个碎块的长度凑足一个整块后再分配一个整块并把之前分配的碎块内容复制到整块里。

另一种考虑碎块设计的方式是可以看作 FFS 每次在结束写入时，会对文件末尾做一次小范围的碎片整理（defragmentation），将多个碎块整理成一个整块。就像普通的碎片整理，这种积攒多个碎块直到构成一个整块， 然后搬运碎块写入整块的操作，会造成一定程度的写入放大；不过因为对碎块的碎片整理只针对碎块， 所以多次写入不会像普通的碎片整理那样产生多次写入放大，而只会发生一次。

ext2 中实现碎块的计划

---

ext2 曾经也计划过类似 FFS 碎块的设计，超级块（superblock）中有个 s_log_frag_size 记录碎块大小， inode 中也有碎块地址 i_faddr 之类的记录，不过 ext2 的 Linux/Hurd 实现最终都没有完成对碎块的支持，于是超级块中记录的碎块大小永远等于整块大小，而 inode 记录的碎块永远为 0 。到 ext4 时代这些记录已经被标为了过期，不再计划支持碎块设计。

在 A Fast File System for UNIX 中介绍了 FFS 的设计思想，最初设计这种整块碎块方案时 FFS 默认的整块是 4KiB 碎块是 512B ，目前 FreeBSD 版本中 newfs 命令创建的整块是 32KiB 碎块是 4KiB 。实验表明采用这种整块碎块两级块大小的方案之后， 文件系统的空间利用率接近块大小等于碎块大小时的 UFS ，而块分配器效率

接近块大小等于整块大小的 UFS 。碎块大小不应小于底层存储设备的扇区大小，而 FFS 记录碎块的方式使得整块的大小不能大于碎块大小的 8 倍。

不考虑希疏文件（sparse files）的前提下，碎块记录只发生在文件末尾，而且在文件系统实际写入到设备前， 内存中仍旧用整块的方式记录，避免那些写入比较慢而一直在写入的程序（比如日志文件）产生大量碎块到整块的搬运。

F2FS 的对闪存优化

打乱一下历史发展顺序，介绍完 FAT 系和传统 Unix 系文件系统的微观结构之后，这里时间线跳到现代， 稍微聊一下 F2FS 的微观结构的实现方式。打乱时间线的原因等我们看完 F2FS 的设计就知道了。

三星的 F2FS 是为有闪存控制器（FTL）的闪存类存储设备优化的日志结构（log-structured）文件系统。「为有闪存控制器（FTL）的闪存类存储设备优化」这一点听起来比较商业噱头，换个说法 F2FS 实际做的是一个写入模式从宏观上看很像 FAT32/exFAT 的日志结构文件系统。

关于 日志结构文件系统的实现思路在我很久之前的一篇博客中稍微有介绍到，传统上日志结构文件系统是为了优化硬盘类存储设备上的写入速度而设计的， 写入时能保持在一大段空闲空间内连续写入，避免写入时发

生寻道。这一点设计虽然是对硬盘的优化， 但是反而对后来的闪存类存储设备而言更为友好。 之前也有文章分别介绍 硬盘 和 闪存 的特点，稍微总结性概括一下的话闪存类存储设备相比硬盘有如下特点：

1. 读取和写入的非对称性。读取可以任意地址寻址无须在意寻道开销，而写入必须顺序写入来减少写放大。
2. 写入和擦除的非对称性。读写的时候是基本扇区（比如4K）大小，而擦除的时候一次性擦除更大一块（比如 128K~8M）。

传统上的日志结构文件系统中，每修改一个文件，无论是修改文件数据还是元数据， 都分配新块写入在日志结构末尾，然后重新写入新的文件元数据（比如间接块和 inode ）和文件系统关键结构 （比如 inode 表等），避免任何覆盖写入的操作。不做覆盖写入本身对 FTL 的闪存很友好， 但是每次内容变化都需要重新写入整个文件系统元数据中涉及到的树结构，直到树根，这会造成一部分写放大。 F2FS 的设计者把传统日志结构文件系统中，文件系统的关键树结构在整个存储空间中不断迁移的现象叫做 漫游树问题（wandering tree problem） 。

F2FS 的设计认识到，实际配有 FTL 的存储设备一般为了支持类似 FAT 的写入模式，会区别对待地址范围内前面一小部分 FAT 表占用的地址空间和后面 FAT 的数据区所在的地址空间，通常 FAT 表占用的地址空间 允许高效地随机地址写入。基于 FTL 闪存这样的存储特点，F2FS 有了解决漫游树问题的方案： 在地址空间前面一小

部分通常是 FAT 表的范围内，放入一个节点地址转换表（NAT，Node Address Table）， 随后当文件内容或者 inode 改变的时候只需要更新 NAT 中记录的地址，就可以切断漫游树需要更新到树根的特点， 避免每次修改一个文件都需要重新写入核心树结构导致的写入放大。于是 F2FS 的宏观结构看起来像是下图：

超级块（SB, superblock）中记录关于文件系统不变的元信息，传统 UFS 中那些会变化的统计信息（比如共有多少 inode ，其中用了多少 inode ）单独分出来记录，叫做检查点（ CP, checkpoint），随后是段信息表（SIT, segment information table）记录每一段的信息（比如写入指针）， 接下来是上述节点地址转换表（NAT, Node Address Table），是一个地址数组， 记录每个节点（node）的地址。

F2FS 中节点（node）的概念包含两种节点，一是传统 UFS 的 inode 节点， 二是当需要间接地址块时的间接节点。从而当很大的文件使用了间接块记录地址的时候， 修改了间接块地址只需要修改间接块节点在 NAT 中的地址，而不需要修改 inode 本身。

F2FS 的 inode 节点散布在主数据区（main area）中，每个 inode 和普通文件块一样大， 比传统 UFS 的 inode 要大得多。比如 ext2 的 inode 是 128 字节， ext4

的 inode 是 256 字节，而常见的 F2FS 的 inode 有 4KiB 大小。如此大的 inode 仍然使用传统 UFS 的块映射的方式存放文件地址，可以估算除了文件属性之类的元数据需要的近 100 字节空间外， F2FS 的 inode 可以存储非常多的块指针。实际上 4KiB 中可存放最多 923 个直接块指针， 外加 2 个一级间接节点，2 个二级间接节点，1 个三级间接节点。

NAT 表大小和节点数量来看， F2FS 需要的节点数量大于传统 UFS 的 inode 数量（因为每个 inode 都是一个 F2FS 节点，而存储大文件用的间接块也是 F2FS 节点），但是小于主数据区的块数量（因为 inode 和别的数据块都占用主数据区）。从而 F2FS 的 NAT 表大小远小于同样块大小的 exFAT 的 FAT 表大小。因为 F2FS 使用 4KiB 块大小而 exFAT 一般使用 32KiB 或者 128KiB 这样的块大小，所以通常 F2FS 的 NAT 大小大概正好较小于同设备上用 exFAT 时 FAT 的大小。 NAT 大小这一点对 F2FS 所说的闪存类存储设备优化很关键，一旦 NAT 超过了 FAT 大小，就可能难以享受针对 FAT 类闪存设备对 FAT 系文件系统的特殊优化了。

因为 F2FS 使用 4KiB 的 inode ，使得 F2FS 很适合做小文件内联优化，对很小的文件（大约小于 3400 字节）可以直接将文件内容写入 inode 中用来存放块映射的那部分空间，避免对文件内容单独分配数据块。 F2FS 也支持文件夹和符号链接等特殊文件的小文件内联，以及支持扩展属性内联存储在 inode 中。

从 F2FS 的设计中可以看出， F2FS 是一个拼命表现得像是 FAT 的日志结构文件系统（LFS）， 进而可以说是 FAT 和 UFS 特点的结合，比起后继的那些为了减少硬盘寻道而优化的现代文件系统而言， F2FS 的设计更古典朴素一些。

连续区块分配的文件系统

前述几个文件系统都是按块分配的文件系统，意味着它们的文件元数据中以某种方式逐一记录了所有数据块的地址。 FAT 系文件系统用 FAT 表中的单链表穿起了文件簇的地址， ext2/3 和 FFS 和 F2FS 这些基于传统 UFS 设计的文件系统用块映射的方式记录了所有块地址。

随着存储设备不断增大而块大小基本保持不变，导致的结果是存储文件的数据块经常有连续地址的多块构成， 从而在按块分配的文件系统中要逐一记录连续的块地址，导致较大的元数据。 并且逐块分配的文件系统也通常不考虑块与块之间是否连续，导致 **外部碎片** 降低硬盘上的读写效率。

新的文件系统设计中为了解决这个问题，通常使用区块分配器，一次分配连续的多块存放文件， 并且在文件元数据中也以（起始地址，区块长度）的方式记录连

续的多块存储空间。 这种记录连续多块地址的方式通常叫做 区块（extent） 每个文件的内容由多个区块构成，接下来介绍一些使用区块记录文件地址的文件系统。

最早使用区块方式记录文件地址，并且使用连续分配的分配器算法避免产生外部碎片的文件系统 可能是 SGI 的 EFS (Extent Filesystem)，这种设计在其继任的 SGI XFS 中被传承并且发扬光大。

XFS 的区块记录

XFS 在线文档的图示

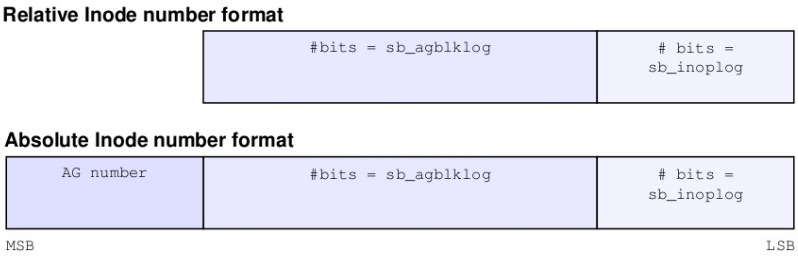
XFS 在线文档 中提供了针对 XFS 数据结构相当明确的图示，本文中使用关于 XFS 的图示全都来自这里。

首先 XFS 中没有 UFS 那样的固定位置的 inode 表，而是把小块的 inode 表散布在整个存储空间中。 每次需要新的空间保存 inode 的时候，一次分配 64 个 inode 所需的数据块，而每个 inode 是 256 字节大小， 从而一个存放 inode 表的数据块大概占用 16KiB 。

传统 UFS 中用 inode 号表示在 inode 表内的数组偏移可以快速定位到 inode ，也可以通过扫描 inode 表的方式找到文件系统中的所有 inode 。而 XFS 中通过两种

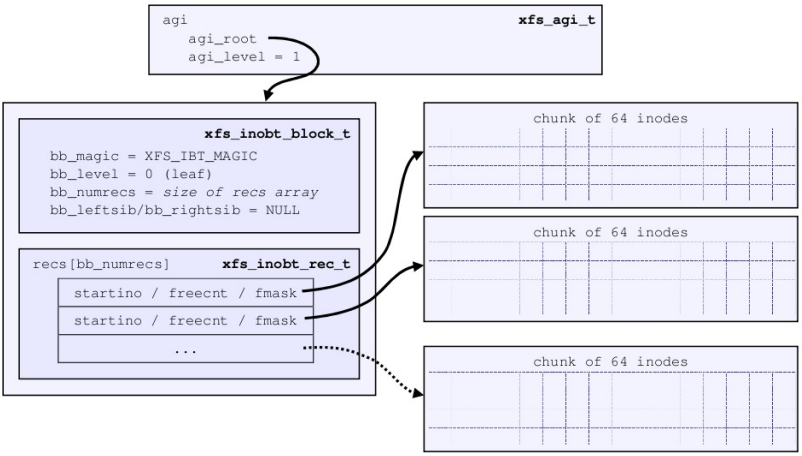
方式支持快速定位 inode 。

1. 给定 inode 号找 inode 的时候， XFS 把 inode 所在数据块的地址和 inode 在数据块中的数组下标直接 编码在了 inode 号 中。比如使用 32位 inode 号的时候，可能有 26 位记录数据块地址，6位记录数据块内的 64 个 inode 中的哪一个。具体使用 inode 号的多少位作为数据块地址，还有多少位作为下标，要看超级块中的 sb_inoplog 记录。这种 inode 号的编码方式使得 XFS 的 inode 号通常非常大，并且32位CPU架构上使用 XFS 时的 inode 号与64位CPU架构上的 XFS 并不兼容。

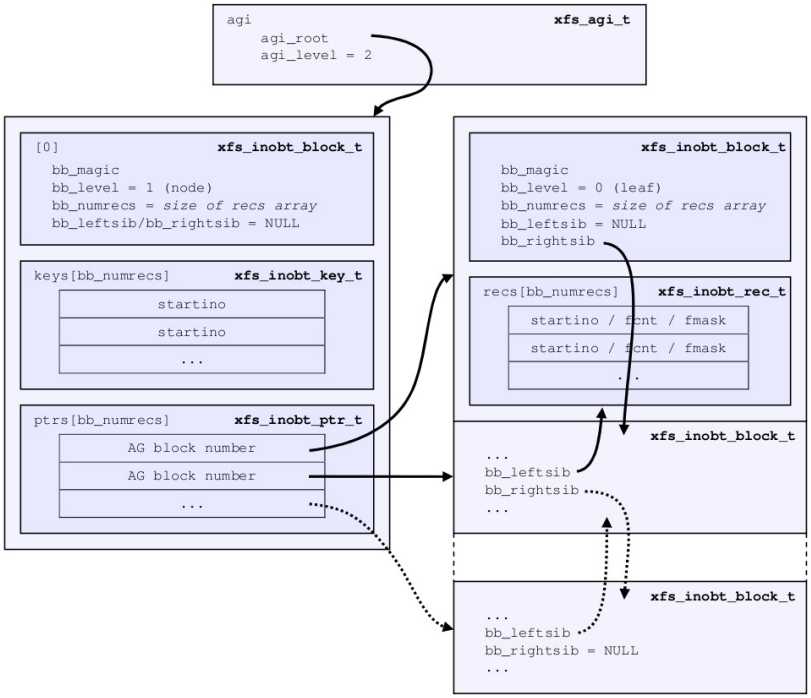


2. 用平衡树（B+Tree）记录所有保存 inode 的数据块的地址和利用率信息，叫做 inode B+tree ，从而扫描 inode b+tree 可以遍历文件系统中所有在使用的 inode 。

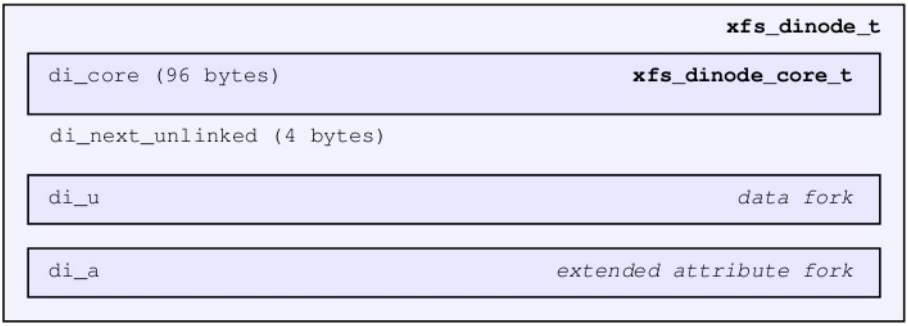
   高度只有一级的 inode B+Tree 结构类似这样：



高度为二级的 inode B+Tree 结构类似这样：

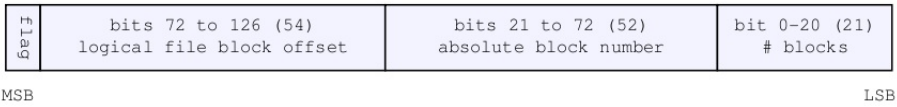


定位到 inode 之后，每个文件至少 256 字节的 inode 又分为三大部分： inode core 、数据分支（data fork）、扩展属性分支（extended attribute fork）。



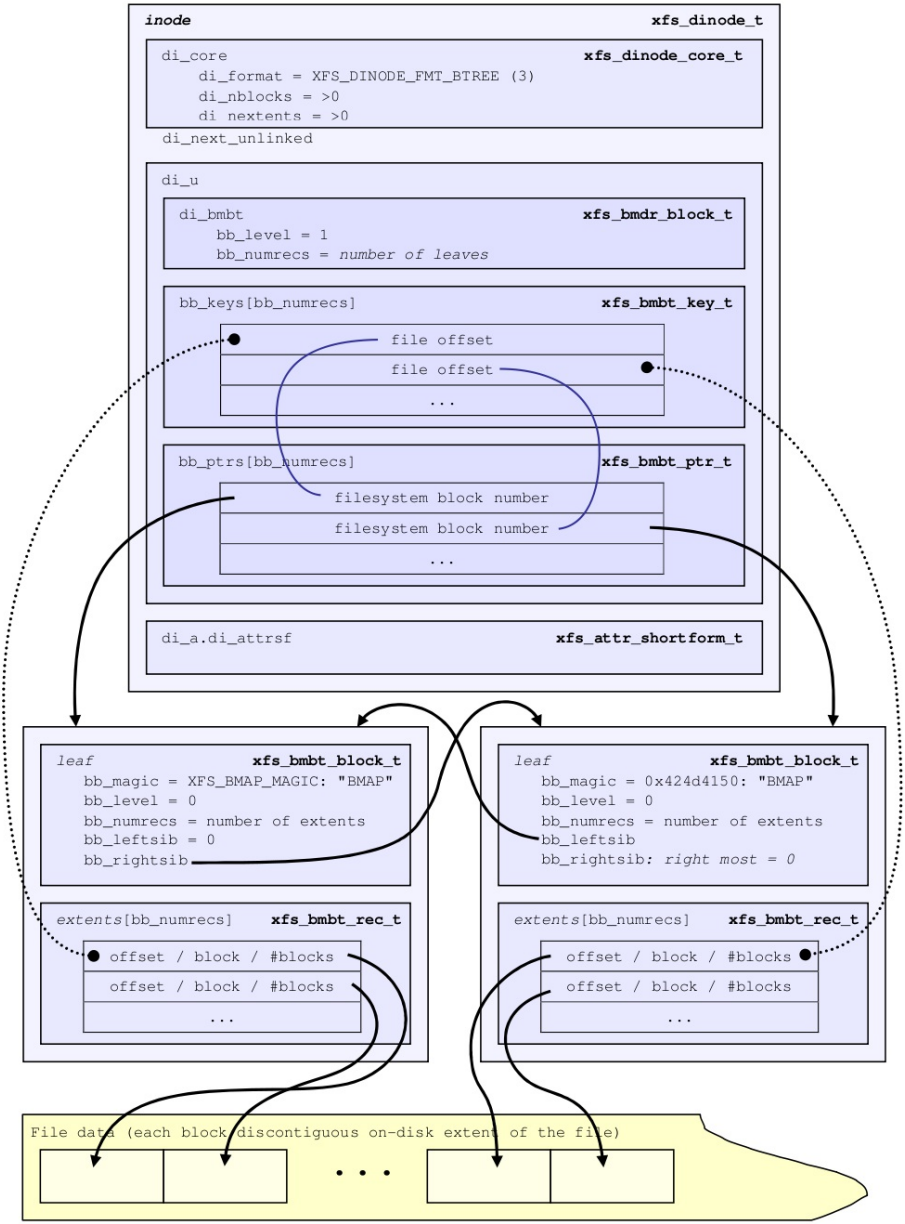
其中 XFS 的 inode core 像是 ext 的 inode 中去掉块映射的部分，记录文件元数据， 数据分支记录文件的地址信息，扩展属性分支则记录文件的扩展属性。XFS 的 inode 结构是有版本号区分的，在 v4 版本结构体中 inode core 占用 96 字节，在 v5 版本结构体中占用 176 字节，从而 inode 剩余的空间可以用来存储数据分支和扩展属性。

在 inode 的数据分支中用区块的方式记录文件地址，每个区块是这样一个 128 位（32字节）的结构：



结构记录了文件的（文件内起始偏移、块地址、块数）三部分信息，和一个标志位。标志位可以用来区分这个区块是 普通区块还是预分配区块（unwritten），用来支持 fallocate 预分配空间。通过跳过一部分起始偏移， 这种区块记录方式可以自然地表达希疏文件，而不需要像 ext2/3 那样记录空指针。 每个区块记录可以表示 2^21 块连续的数据块。

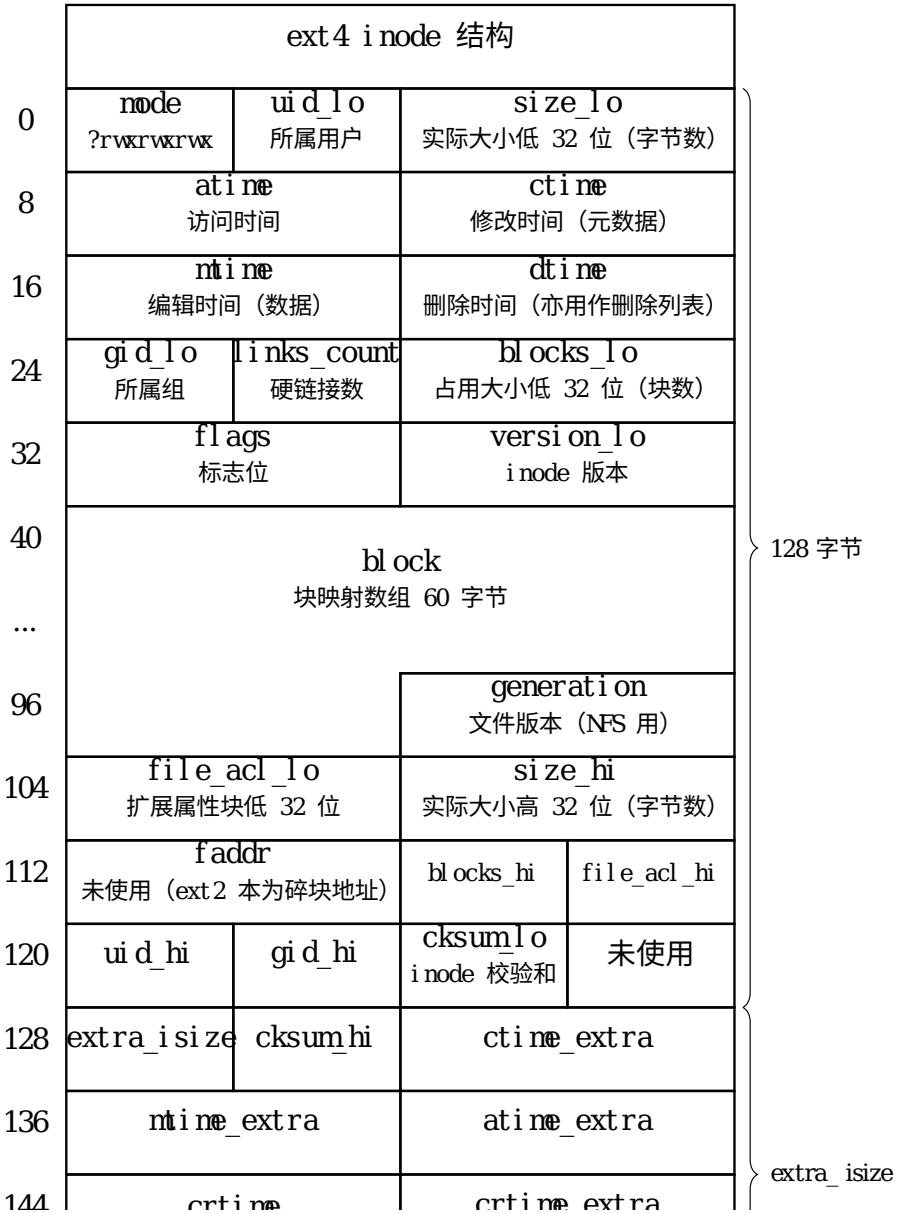
如果文件数据的所有区块记录都可以塞入 inode 的数据分支中，那么 XFS 用 inode 中的区块列表记录这些区块，否则 XFS 会分配区块的 B+Tree ，把这些区块信息写入 B+Tree 的叶节点中。 单层的区块 B+Tree 看起来像是这样：



对扩展属性的支持也是类似， 当扩展属性比较小的时候 XFS 可以将扩展属性列表直接存入 inode 的扩展属性分支中，当扩展属性较大的时候 XFS 分配间接数据块保存扩展属性的 B+Tree 。

ext4 中的小文件内联优化

ext4 中首先将 ext2/3 的 128 字节 inode 扩展到了默认 256 字节大小，整个结构类似下图：



对比上面 ext2 的 inode，ext4 中用 osd2 的位置额外存了很多数据的高位，扩展了那些数据的位数，随后在增加的结构中加入了几个时间戳的纳秒支持。之后总

共 256 字节的 inode 保存了大约 156 的 inode 结构后还有大概 100 字节空间剩余，ext4 用这些剩余空间保存扩展属性。

inode 中 60 字节的 block 数组在 ext2/3 中是用来保存块映射，而在 ext4 中保存区块结构。 ext4 中每个区块记录占用 12 字节，多个区块记录和 XFS 一样以树的形式构成。

注意到典型的 Unix 文件系统中，有很多「小」文件小于 60 字节的块映射大小，而且不止有很多小的普通文件， 包括目录文件、软链接、设备文件之类的特殊 Unix 文件通常也很小。为了存这些小文件而单独分配一个块并在 inode 中记录单个块指针显得很浪费，于是有了 **小文件内联优化 (small file inlining)** 。

一言以蔽之小文件内联优化就是在 inode 中的 60 字节的块映射区域中直接存放文件内容。 在 inode 前半标志位 （flags）中安插一位记录 (EXT4_INLINE_DATA_FL)，判断后面这 60 字节的块映射区是存储为内联文件，还是真的存放块映射。这些被内联优化掉的小文件磁盘占用会显示为 0 ，因为没有分配数据块，但是仍然要占用完整一个 inode 。

ext4 实现的小文件内联还利用了扩展属性占用的空间，当 60 字节的块映射区还不足存放文件内容时， ext4 会分配一个特殊的扩展属性 “system.data” 直接保存文件内容。

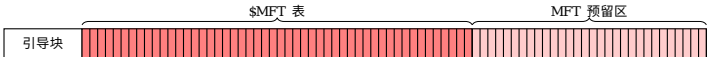
对较小的目录文件，inode 中的 60 字节 block 空间可以直接保存目录的内容。对符号链接文件， 这部分空间可以直接保存符号链接的目标，这两种特殊文件可以视作小文件内联优化的特例。

NTFS 的 MFT 表

NTFS 虽然是出自微软之手，其微观结构却和 FAT 很不一样，某种角度来看更像是一个 UFS 后继。 NTFS 没有固定位置的 inode 表，但是有一个巨大的文件叫 $MFT (Master File Table ），整个 $MFT 的作用就像是 UFS 中 inode 表的作用。NTFS 中的每个文件都在 $MFT 中存有一个对应的 MFT 表项， MFT 表现有固定长度 1024 字节，整个 $MFT 文件就是一个巨大的 MFT 表项构成的数组。每个文件可以根据 MFT 序号作为数组下标在 $MFT 中找到具体位置。 网上经常有人说 NTFS 中所有东西都是文件，包括 $MFT 也是个文件，这其实是在说所有东西包括 $MFT 本身都在 $MFT 中有个占用 1KiB 的文件记录。

$MFT 本身也是个文件，所以它不必连续存放也没有固定的开始位置，在引导块中记录了 $MFT 的起始位置，然后在 $MFT 起始位置中记录的第一项文件记录了关于 $MFT 自身的元数据，也就包含 $MFT 占用的别的空间的信息。 于是可以先读取 $MFT 的最初几块，找到 $MFT 文件存放的地址信息，继而勾勒出整个 $MFT 所占的空间。 实际上 Windows 的 NTFS 驱动在创建文件系统

时给 $MFT 预留了很大一片存储区， Windows XP 之后的碎片整理工具也会非常积极地对 $MFT 文件本身做碎片整理，于是通常存储设备上的 $MFT 不会散布在很多地方而是集中在 NTFS 分区靠前的一块连续位置。于是宏观而言 NTFS 像是这样：



上述文件系统汇总

文件系统汇总

文件系统	基础分配单位	常见块大小	文件寻址方式	小文件内联
FAT32	簇	32K	FAT单链表	否
exFAT	簇	128K	FAT单链表	否
NTFS	MFT项/簇	1K/4K	区块	~900
FFS	inode/碎块/整块	128/4K/32K	块映射	否

ext2/3	inode/块	128/4K	块映射	否
ext4	inode/块	256/4K	块映射/区块树	~150
xfs	inode/块	256/4K	区块树	~80，仅目录和符号连接
F2FS	node	4K	块映射	~3400
reiser3	tree node/blob	4K/4K	块映射	4k(尾内联)
btrfs	tree node/block	16K/4K	区块树	2K(区块内联)
ZFS	ashift/recordsize	4K/128K	区块树	~100(块指针内联)