内部碎片与小文件优化

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：

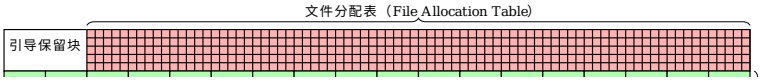
1. 文件数量在增加
2. 大文件的大小在增加

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。

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

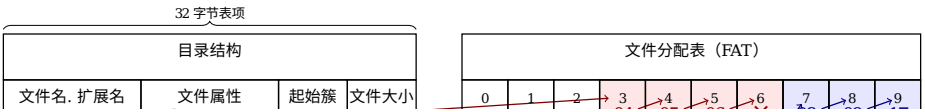
在 FAT 系文件系统(FAT12/16/32/exFAT)中，整个存储空间除了一些保留扇区之外，被分为两大块区域，看起来类似这样：



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	FAT16	FAT32	exFAT	NTFS
< 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 TiB	128KiB	64KiB
> 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 。

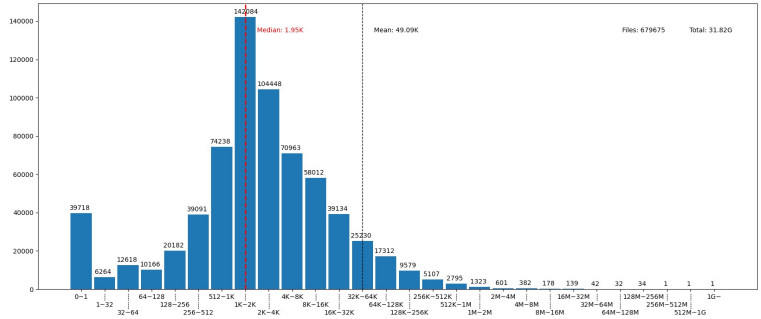
FAT 系的簇大小是可以由用户在创建文件系统时指定的，大部分普通用户会使用系统根据存储设备容量推算的默认值， 而存储设备的生产厂商则可以根据底层存储设备的特性决定一个适合存储设备的簇大小。在选择簇大小时， 要考虑取舍，较小的簇意味着同样容量下更多的簇数，而较大的簇意味着更少的簇数，取舍在于：

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。

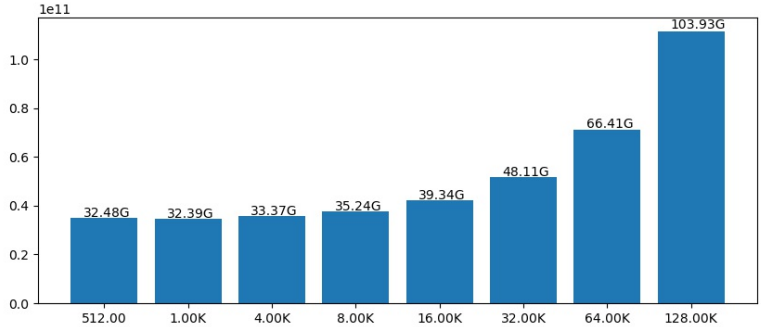
较大的簇： 优势是避免 **外部碎片** 导致的性能损失，劣势是 **内部碎片** 带来的低空间利用率。

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：

簇大小	FAT 表项数	总簇数	总占用	数据簇数	FAT簇数
512.00	63.95M	64.95M	32.48G	63.95M	511.61K

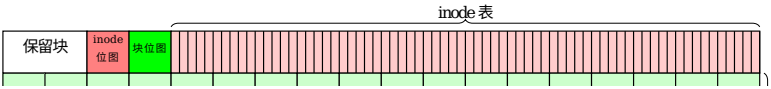
1.00K	32.14M	32.39M	32.39G	32.14M	128.55K
4.00K	8.33M	8.34M	33.37G	8.33M	8.33K
8.00K	4.40M	4.41M	35.24G	4.40M	2.20K
16.00K	2.46M	2.46M	39.34G	2.46M	630.00
32.00K	1.50M	1.50M	48.11G	1.50M	193.00
64.00K	1.04M	1.04M	66.41G	1.04M	67.00
128.00K	831.40K	831.45K	103.93G	831.40K	26.00

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

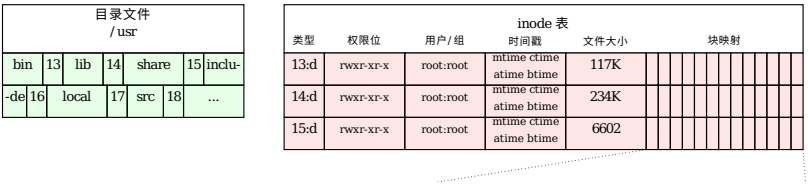
传统 Unix 文件系统（下称 UFS）和它的继任者们，包括 Linux 的 ext2/3 ， FreeBSD 的 FFS 等文件系统，使用在 inode 中记录块映射（bmap, block mapping）的方式记录文件存储的地址范围。 术语上 UFS 中所称的

「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，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 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。

FFS 中的整块与碎块

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布叫整块（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 大小的碎块。写文件时先把末端不足

一个整块的内容写入碎块中， 多个碎块的长度凑足一个整块后分配一个整块并把之前分配的碎块内容复制到整块里。

ext2 中的碎块计划

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 中也有碎块数量之类的记录，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。

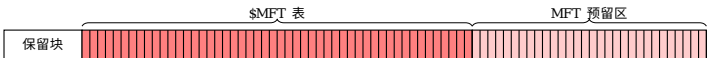
另一种考虑碎块设计的方式是可以看作 FFS 每次在结束写入时，会对文件末尾做一次小范围的碎片整理（defragmentation），将多个碎块整理成一个整块。

NTFS 与区块（extent）

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 表现的数组。每个文件可以根据 MFT 序号在 $MFT 中找到具体位置。

$MFT 本身也是个文件，所以它不必连续存放，在 $MFT 中记录的第一项文件记录了 $MFT 自身的元数据。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：



ext4 中的小文件内联优化

https://lwn.net/Articles/468678/

ext4 的 inode 存储方式基本上类似上述 UFS，具体到 inode 而言， ext2/3 中每个 inode 占用 128 字节，其中末尾有 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 ext2 文档 。

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 （i_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 。

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	支持文件内联
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
Ext4	inode/块	256/4K	块映射/区块树	~150
xfs	inode/块	256/4K	区块树	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)