

# 內部碎片與小文件優化

---



副標題1：文件存入文件系統後佔用多大？

副標題2：文件系統的微觀結構

上篇「系統中的大多數文件有多大？」提到，文件系統中大部分文件其實都很小，中位數一直穩定在 4K 左右，而且這個數字並沒有隨着存儲設備容量的增加而增大。但是存儲設備的總體容量實際是在逐年增長的，，總容量增加而文件大小中位數不變的原因，可能是以下兩種情況：

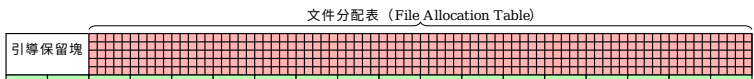
1. 文件數量在增加
2. 大文件的大小在增加

實際上可能是這兩者綜合的結果。這種趨勢給文件系統設計帶來了越來越多的挑戰，因為我們不能單純根據平均文件大小來增加塊大小（block size）優化文件讀寫。微軟的文件系統（FAT系和 NTFS）使用「簇（cluster）」這個概念管理文件系統的可用空間分配，在 Unix 系文件系統中有類似的塊（block）的概念，只不過稱呼不一樣。現代文件系統都有這個塊大小或者簇大小的概念，從而基本的文件空間分配可以獨立於硬件設備本身的扇區大小。塊大小越大，單次分配空間越大，文件系統所需維護的元數據越小，複雜度越低，實現起來也越容易。而塊大小越小，越能節約可用空間，避免內部碎片造成的浪費，但是跟蹤空間所需的元數據也越複雜。

具體塊/簇大小對文件系統設計帶來什麼樣的挑戰？我們先來看一下（目前還在用的）最簡單的文件系統怎麼存文件的吧：

## FAT系文件系統與簇大小

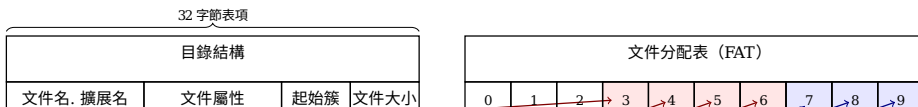
在 FAT 系文件系統(FAT12/16/32/exFAT)中，整個存儲空間除了一些保留扇區之外，被分為兩大塊區域，看起來類似這樣：



前一部分區域放文件分配表（File Allocation Table），後一部分是實際存儲文件和目錄的數據區。數據區被劃分成「簇（cluster）」，每個簇是一到多個連續扇區，然後文件分配表中表項的數量 決定了後面可用空間的簇的數量。文件分配表（FAT）在 FAT 系文件系統中這裏充當了兩個重要作用：

1. **宏觀尺度**：從 CHS 地址映射到線性的簇號地址空間，管理簇空間分配。空間分配器可以掃描 FAT 判斷哪些簇處於空閒狀態，那些簇已經被佔用，從而分配空間。
2. **微觀尺度**：對現有文件，FAT 表中的記錄形成一個單鏈表結構，用來尋找文件的所有已分配簇地址。

比如在根目錄中有 4 個不同大小文件的 FAT16 中，使用 512 字節的簇大小的文件系統，其根目錄結構和 FAT 表可能看起來像下圖這樣：



目錄結構中的文件記錄是固定長度的，其中保存 8.3 長度的文件名，一些文件屬性（修改日期和時間、隱藏文件之類的），文件大小的字節數，和一個起始簇號。起始簇號在 FAT 表中引出一個簇號的單鏈表，順着這個單鏈表能找到存儲文件內容的所有簇。

直觀上理解，FAT表像是數據區域的縮略圖，數據區域有多少簇，FAT表就有多少表項。FAT系文件系統中每個簇有多大，由文件系統總容量，以及 FAT 表項的數量限制。我們來看一下微軟文件系統默認格式化的簇大小（數據來源）：

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		

用於軟盤的時候 FAT12 的簇大小直接等於扇區大小 512B，在容量較小的 FAT16 上也是如此。FAT12 和 FAT16 都被 FAT 表項的最大數量限制（分別是 4068 和 65460），FAT 表本身不會太大。所以上表中可見，隨着設備容量增加，FAT16 需要增加每簇大小，保持同樣數量的 FAT 表項。

到 FAT32 和 exFAT 的年代，FAT 表項存儲 32bit 的簇指針，最多能有接近 4G 個數量的 FAT 表項，從而表項數量理應不再限制 FAT 表大小，使用和扇區大小同樣的簇大小。不過事實上，簇大小仍然根據設備容量增長而增大。FAT32 上 256MiB 到 8GiB 的範圍內使用 4KiB 簇大小，隨後簇大小開始增加；在 exFAT 上 256MiB 到 32GiB 使用 32KiB 簇大小，隨後增加到 128KiB。

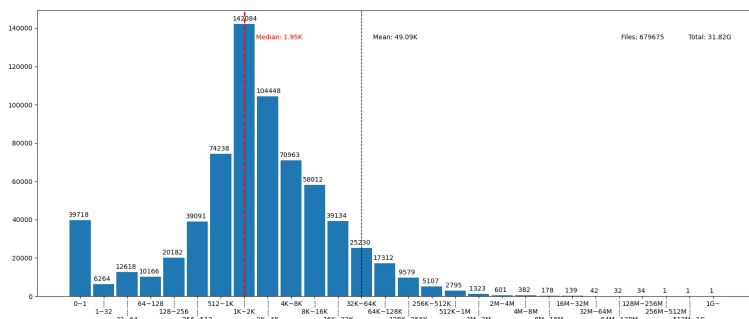
FAT 系的簇大小是可以由用戶在創建文件系統時指定的，大部分普通用戶會使用系統根據存儲設備容量推算的默認值，而存儲設備的生產廠商則可以根據底層存儲設備的特性決定一個適合存儲設備的簇大小。在選擇簇大小時，要考慮取捨，較小的簇意味着同樣容量下更多的簇數，而較大的簇意味着更少的簇數，取捨在於：

**較小的簇：** 優勢是存儲大量小文件時，降低 **內部碎片 (Internal fragmentation)** 的程度，帶來更多可用空間。劣勢是更多 **外部碎片 (External fragmentation)** 導致訪問大文件時來回跳轉降低性能，並且更多簇數也導致簇分配器的性能降低。

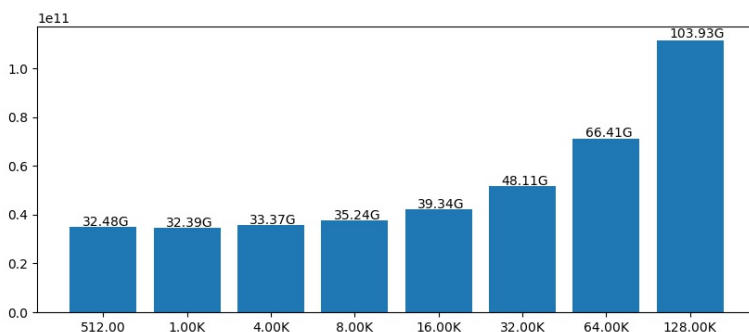
**較大的簇：** 優勢是避免 **外部碎片** 導致的性能損失，劣勢是 **內部碎片** 帶來的低空間利用率。

FAT 系文件系統使用隨着容量增加的簇大小，導致的劣勢在於極度浪費存儲空間。如果文件大小是滿足隨機分佈，那麼大量文件平均而言，每個文件將有半個簇的未使用空間，比如假設一個 64G 的 exFAT 文件系統中存有 8000 個文件，使用 128KiB 的簇大小，那麼平均下來大概會有 500MiB 的空間浪費。實際上如前文系統中的大多數文件有多大？所述，一般系統中的文件大小並非隨機分佈，而是大多數都在大約 1KiB~4KiB 的範圍內，從而造成的空間浪費更為嚴重。

可能有人想說「現在存儲設備的容量都那麼大了，浪費一點點存儲空間換來讀寫性能的話也沒什麼壞處嘛」，於是要考察加大簇大小具體會浪費多少存儲空間。借用前文中統計文件大小的工具和例子，比如我的文件系統中存有 31G 左右的文件，文件大小分佈符合下圖的樣子：



假如把這些文件存入不同簇大小的 FAT32 中，根據簇大小，最終文件系統佔用空間是下圖：



在較小的簇大小時，文件系統佔用接近於文件總大小 31G，而隨着簇大小增長，到使用 128KiB 簇大小的時候空間佔用徒增到 103.93G，是文件總大小的 3.35 倍。如此大的空間佔用源自於目標文件系統 中有大量小文件，每個不足一簇的小文件都要佔用完整一簇的大小。可能注意到上圖 512B 的簇大小時整個文件系統佔用反而比 1KiB 簇大小時的更大，這是因為 512B 簇大小的時候 FAT 表本身的佔用更大。具體數字如下表：

FAT 表				數據	
簇大小	項數	總簇數	總佔用	簇數	FAT簇數
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

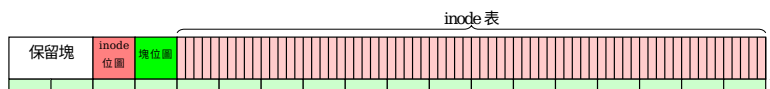
FAT 系文件系統這種對簇大小選擇的困境來源在於，FAT 試圖用同一個數據結構——文件分配表——同時管理 **宏觀尺度** 的可用空間分配和 **微觀尺度** 的文件尋址，這產生兩頭都難以兼顧的矛盾。NTFS 和其它 Unix-like 系統的文件系統都使用塊位圖(block bitmap)跟蹤可用空間分配，將宏觀尺度的空間分配問題和微觀尺度的文件尋址問題分開解決，從而在可接受的性能下允許更小的簇大小和更多的簇數。

## 傳統 Unix 文件系統的塊映射

傳統 Unix 文件系統（下稱 UFS）和它的繼任者們，包括 Linux 的 ext2/3，FreeBSD 的 FFS 等文件系統，使用在 inode 中記錄塊映射（bmap, block mapping）的方式記錄文件存儲的地址範圍。術語上 UFS 中所稱的



「塊 (block)」等價於微軟系文件系統中所稱的「簇 (cluster)」，都是對底層存儲設備中扇區尋址的抽象。



上圖乍看和 FAT 總體結構很像，實際上重要的是「inode表」和「數據塊」兩大區域分別所佔的比例。FAT 系文件系統中，每個簇需要在 FAT 表中有一個表項，所以 FAT 表的大小是每簇大小佔 2 字節 (FAT16) 或 4 字節 (FAT32/exFAT)。假設 exFAT 用 32K 簇大小的話，FAT 表整體大小與數據區的比例大約是 4:32K。UFS 中，在創建文件系統時 `mkfs` 會指定一個 `bytes-per-inode` 的比例，比如 `mkfs.ext4` 默認的 `-i` `bytes-per-inode` 是 32K 於是每 32K 數據空間分配一個 inode，而每個 inode 在 ext4 佔用 256 字節，於是 inode 空間與數據塊空間的比例大約是 256:32K。宏觀上，FAT 表是在 FAT 文件系統中地址前端一段連續空間；而 UFS 中 inode 表的位置 **不一定** 是在存儲設備地址範圍前端連續的空間，至於各個 UFS 如何安排 inode 表與數據塊的宏觀佈局可能今後有空再談，本文所關心的只是 inode 表中存放 inode 獨立於數據塊的存儲空間，兩者的比例在創建文件系統時固定。

UFS 與 FAT 文件系統一點非常重要的區別在於：Unix 文件系統中 **文件名不屬於 inode 記錄的文件元數據**。FAT 系文件系統中文件元數據存儲在目錄結構中，每個目錄表項代表一個文件（除了 VFAT 的長文件名用隱藏目錄表項），佔用 32 字節，引出一個單鏈表表達文件存儲地址；在 UFS 中，目錄內容和 inode 表中的表項和文件地址的樣子像是這樣：

目錄文件 /usr							inode 表										
							類型	權限位	用戶/組	時間戳	文件大小	塊映射					
bin	13	lib	14	share	15	inclu-	13:d	rwxt-xr-x	root:root	mtime ctime atime btime	117K						
-de	16	local	17	src	18	...	14:d	rwxt-xr-x	root:root	mtime ctime atime btime	234K						
							15:d	rwxt-xr-x	root:root	mtime ctime atime btime	6602						

UFS 中每個目錄文件的內容可以看作是單純的（文件名：inode號）構成的數組，最早 Unix v7 的文件系統中文件名長度被限制在 14 字節，後來很快就演變成可以接受更長的文件名只要以 \0 結尾。關於文件的元數據信息，比如所有者和權限位這些，文件元數據並不記錄在目錄文件中，而是記錄在長度規整的 inode 表中。inode 表中 inode 記錄的長度規整這一點非常重要，因

爲知道了 inode 表的位置和 inode 號，可以直接算出 inode 記錄在存儲設備上的地址，從而快速定位到所需文件的元數據信息。在 inode 記錄的末尾有個固定長度的塊映射表，填寫文件的內容的塊地址。

因爲 inode 記錄的長度固定，從而 inode 記錄末尾位置得到塊指針數組的長度也是固定並且有限的，在 Unix v7 FS 中這個數組可以記錄 13 個地址，在 ext2/3 中可以記錄 15 個地址。前文說過，文件系統中大部分文件大小都很小，而少數文件非常大，於是 UFS 中使用間接塊指針的方案，用有限長度的數組表達任意大小的文件。

在 UFS 的 inode 中可以存 13 個地址，其中前 10 個地址用於記錄「直接塊指針（direct block address）」。當文件大小大於 10 塊時，從第 11 塊開始，分配一個「一級間接塊（level 1 indirect block）」，其位置寫在 inode 中第 11 個塊地址上，間接塊中存放塊指針數組。假設塊大小是 4K 而指針大小是 4 字節，那麼一級間接塊可以存放 1024 個直接塊指針。當文件大小超過 1034(=1024+10) 時，再分配一個「二級間接塊（level 2 indirect block）」，存在 inode 中的第 12 個塊地址上，二級間接塊中存放的是一級間接塊的地址，形成一個兩層的指針樹。同理，當二級間接塊也不夠用的時候，分配一個「三級間接塊（level 3 indirect block）」，三級間接塊本身的地址存在 inode 中最後第 13 個塊地址位置上，而三級間接塊內存放指向二級間接塊的指針，形成一個三層的指針樹。UFS 的 inode 一共有 13 個塊地址槽，於是不存在四級間接塊了，依靠上

述最多三級的間接塊構成的指針樹，如果是 4KiB 塊大小的話，每個 inode 最多可以有 \(\mathbf{10+1024+1024^2+1024^3 = 1074791434}\) 塊，最大支持超過 4GiB 的文件大小。

UFS 使用這種 inode 中存儲塊映射引出間接塊樹的形式存儲文件塊地址，這使得 UFS 中定位到文件的 inode 之後查找文件存儲的塊比 FAT 類的文件系統快，因為不再需要去讀取 FAT 表。這種方式另一個特徵是，當文件較大時，讀寫文件前段部分的數據，比如 inode 中記錄的前10塊直接塊地址的時候，比隨後 10~1024 塊一級間接塊要快，同樣的訪問一級間接塊中的數據也比二級和三級間接塊要快。一些 Unix 工具比如 file 判斷文件內容的類型只需要讀取文件前段的內容，在這種記錄方式下也可以比較高效。

## FFS 中的整塊與碎塊

FreeBSD 用的 FFS 基於傳統 UFS 的存儲方式，爲了對抗比較小的塊大小導致塊分配器的性能損失，FFS 創新的使用兩種塊大小記錄文件塊，在此我們把兩種塊大小分佈叫整塊 (block) 和碎塊 (fragment)。整塊和碎塊的大小比例最多是 8:1，也可以是 4:1 或者 2:1，比如可以使用 4KiB 的整塊和 1KiB 的碎塊，或者用 32KiB 的整塊並配有 4KiB 大小的碎塊。寫文件時先把末端不足

一個整塊的內容寫入碎塊中，多個碎塊的長度湊足一個整塊後分配一個整塊並把之前分配的碎塊內容複製到整塊裏。

## ext2 中的碎塊計劃

---

ext2 曾經也計劃過類似 FFS 碎塊的設計，超級塊（superblock）中有個 `s_log_frag_size` 記錄碎塊大小，inode 中也有碎塊數量之類的記錄，不過 ext2 的 Linux/Hurd 實現最終都沒有完成對碎塊的支持，於是超級塊中記錄的碎塊大小永遠等於整塊大小，而 inode 記錄的碎塊永遠為 0。到 ext4 時代這些記錄已經被標爲了過期，不再計劃支持碎塊設計。

在 A Fast File System for UNIX 中介紹了 FFS 的設計思想，最初設計這種整塊碎塊方案時 FFS 默認的整塊是 4KiB 碎塊是 512B，目前 FreeBSD 版本中 `newfs` 命令創建的整塊是 32KiB 碎塊是 4KiB。實驗表明採用這種整塊碎塊兩級塊大小的方案之後，文件系統的空間利用率接近塊大小等於碎塊大小時的 UFS，而塊分配器效率接近塊大小等於整塊大小的 UFS。碎塊大小不應小於底層存儲設備的扇區大小，而 FFS 記錄碎塊的方式使得整塊的大小不能大於碎塊大小的 8 倍。

不考慮稀疏文件（sparse files）的前提下，碎塊記錄只發生在文件末尾，而且在文件系統實際寫入到設備前，內存中仍舊用整塊的方式記錄，避免那些寫入比較

慢而一直在寫入的程序比如日志文件產生大量碎塊到整塊的搬運。

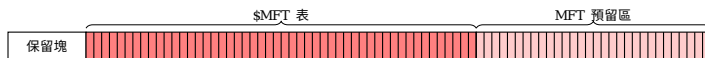
另一種考慮碎塊設計的方式是可以看作 FFS 每次在結束寫入時，會對文件末尾做一次小範圍的碎片整理（defragmentation），將多個碎塊整理成一個整塊。

## NTFS 與區塊（extent）

NTFS 雖然是出自微軟之手，其微觀結構卻和 FAT 很不一樣，某種角度來看更像是一個 UFS 後繼。NTFS 沒有固定位置的 inode 表，但是有一個巨大的文件叫 \$MFT (Master File Table)，整個 \$MFT 的作用就像是 UFS 中 inode 表的作用。NTFS 中的每個文件都在 \$MFT 中存有一個對應的 MFT 表項，MFT 表現有固定長度 1024 字節，整個 \$MFT 文件就是一個巨大的 MFT 表現的數組。每個文件可以根據 MFT 序號在 \$MFT 中找到具體位置。

\$MFT 本身也是個文件，所以它不必連續存放，在 \$MFT 中記錄的第一項文件記錄了 \$MFT 自身的元數據。於是可以先讀取 \$MFT 的最初幾塊，找到 \$MFT 文件存放的地址信息，繼而勾勒出整個 \$MFT 所佔的空間。實際上 Windows 的 NTFS 驅動在創建文件系統時給 \$MFT 預留了很大一片存儲區，Windows XP 之後的碎片整理工具也會非常積極地對 \$MFT 文件本身做碎片整理，於

是通常存儲設備上的 \$MFT 不會散佈在很多地方而是集中在 NTFS 分區靠前的一塊連續位置。於是宏觀而言 NTFS 像是這樣：



## ext4 中的小文件內聯優化

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

ext4 的 inode 存儲方式基本上類似上述 UFS，具體到 inode 而言，ext2/3 中每個 inode 佔用 128 字節，其中末尾有 60 字節存儲塊映射，可以存放 12 個直接塊指針和三級間接塊指針。詳細的 ext2 inode 結構可見 [ext2 文檔](#)。

注意到典型的 Unix 文件系統中，有很多「小」文件小於 60 字節的塊映射大小，而且不止有很多小的普通文件，包括目錄文件、軟鏈接、設備文件之類的特殊 Unix 文件通常也很小。爲了存這些小文件而單獨分配一個塊並在 inode 中記錄單個塊指針顯得很浪費，於是有了 **小文件內聯優化 (small file inlining)**。

一言以蔽之小文件內聯優化就是在 inode 中的 60 字節的塊映射區域中直接存放文件內容。在 inode 前半標誌位 (i\_flags) 中安插一位記錄 (EXT4\_INLINE\_DATA\_FL)，判斷後面這 60 字節的塊映射區是存儲為內聯文件，還是真的存放塊映射。這些被內聯優化掉的小文件磁盤佔用會顯示為 0，因為沒有分配數據塊，但是仍然要佔用完整一個 inode。

# 上述文件系統彙總

文件系統彙總

文件系統	基礎分配單位	常見塊大小	文件尋址方式	支持文件內聯
FAT32	簇	32K	FAT單鏈表	否
exFAT	簇	128K	FAT單鏈表	否
NTFS	MFT項/簇	1K/4K	區塊	900
FFS	inode/碎塊/整塊	128/4K/32K	塊映射	否
Ext4	inode/塊	256/4K	塊映射/區塊樹	~150
xfs	inode/塊	256/4K	區塊樹	僅目錄和符號連接



F2FS	node	4K	塊映射	~3400
reiser3	tree node/blob	4K/4K	塊映射	4k(尾內 聯)
btrfs	tree node/block	16K/4K	區塊樹	~2K(區塊 內聯)
ZFS	ashift/recom 4k/128k	4k/128K	區塊樹	~100(塊 指針內聯)