Btrfs vs ZFS 實現 snapshot 的差異

目錄

> zfs 這個東西倒是名不符實。叫 z storage stack 明顯更符合。 叫 fs 但不做 fs 自然確實會和 btrfs 有很大出入。
> 我反而以前還好奇為什麼 btrfs 不弄 zvol， 直到我意識到這東西真是一個 fs，名符奇實。
> —— 某不愿透露姓名的 Ext2FSD 開發者

Btrfs 和 ZFS 都是開源的寫時拷貝（Copy on Write, CoW）文件系統，都提供了相似的子卷管理和 快照 (snapshot）的功能。網上有不少文章都評價 ZFS 實現 CoW FS 的創新之處，進而想說「 Btrfs 只是 Linux/GPL 陣營對 ZFS 的拙劣抄襲」。或許（在存儲領域人盡皆知而在領域外）鮮有人知，在 ZFS 之前就有 NetApp 的商業產品 WAFL (Write Anywhere File Layout) 實現了

CoW 語義的文件系統，並且集成了快照和卷管理之類的功能。描述 btrfs 原型設計的 論文 和 發表幻燈片 也明顯提到 WAFL 比提到 ZFS 更多一些，應該說 WAFL 這樣的企業級存儲方案纔是 ZFS 和 btrfs 共同的靈感來源，而無論是 ZFS 還是 btrfs 在其設計中都汲取了很多來自 WAFL 的經驗教訓。

我一開始也帶着「Btrfs 和 ZFS 都提供了類似的功能，因此兩者必然有類似的設計」這樣的先入觀念，嘗試去使用這兩個文件系統，卻經常撞上兩者細節上的差異，導致使用時需要不盡相同的工作流，或者看似相似的用法有不太一樣的性能表現，又或者一邊有的功能，比如 ZFS 的在線去重（in-band dedup），Btrfs 的 reflink，在另一邊沒有的情況，進而需要不同細粒度的子卷劃分方案。後來看到了 LWN 的這篇《A short history of btrfs》讓我意識到 btrfs 和 ZFS 雖然表面功能上看起來類似，但是實現細節上完全不一樣，所以需要不一樣的用法，適用於不一樣的使用場景。

爲了更好地理解這些差異，我四處蒐羅這兩個文件系統的實現細節，於是有了這篇筆記，記錄一下我查到的種種發現和自己的理解。~~（或許會寫成一個系列？還是先別亂挖坑不填。）~~ 只是自己的筆記，所有參閱的資料文檔都是二手資料，沒有深挖過源碼，還參雜了自己的理解，於是難免有和事實相違的地方，如有寫錯，還請留言糾正。

1 Btrfs 的子卷和快照

關於寫時拷貝（CoW）文件系統的優勢，我們爲什麼要用 btrfs/zfs 這樣的寫時拷貝文件系統， 而不是傳統的文件系統設計，或者寫時拷貝文件系統在使用時有什麼區別之類的，網上同樣也能找到很多介紹 ，這裏不想再討論。這裏假設你用過 btrfs/zfs 至少一個的快照功能，知道它該怎麼用， 並且想知道更多細節，判斷怎麼用那些功能才合理。

先從兩個文件系統中（表面上看起來）比較簡單的 btrfs 的子卷（subvolume）和快照（snapshot）說起。 關於子卷和快照的常規用法、推薦佈局之類的話題就不細說了，網上能找到很多不錯的資料，比如 btrfs wiki 的 SysadminGuide 頁 和 Arch wiki 上 Btrfs#Subvolumes 頁都有不錯的參考價值。

1.1 子卷和快照的術語

在 btrfs 中，存在於存儲媒介中的只有「子卷」的概念，「快照」只是個創建「子卷」的方式， 換句話說在 btrfs 的術語裏，子卷（subvolume）是個名詞，而快照（snapshot）是個動詞。 如果脫離了 btrfs 術語的上下文，或者不精確地稱呼的時候，也經常有文檔把 btrfs 的快照命令創建出的子卷叫做一個快照，所以當提到快照

的時候，根據上下文判斷這裏是個動詞還是名詞， 把名詞的快照當作用快照命令創建出的子卷就可以了。或者我們可以理解爲， **互相共享一部分元數據（metadata）的子卷互爲彼此的快照（名詞）** ， 那麼按照這個定義的話，在 btrfs 中創建快照（名詞）的方式其實有兩種：

1. 用 `btrfs subvolume snapshot` 命令創建快照
2. 用 `btrfs send` 命令並使用 `-p` 參數發送快照，並在管道另一端接收

`btrfs send` 命令的 `-p` 與 `-c`

這裏也順便提一下 `btrfs send` 命令的 `-p` 參數和 `-c` 參數的差異。 只看 btrfs-send(8) 的描述的話：

-p <parent>
send an incremental stream
from parent to subvol

-c <clone-src>
use this snapshot as a clone
source for an incremental
send (multiple allowed)

看起來這兩個都可以用來生成兩個快照之間的差分，只不過 -p 只能指定一個「parent」， 而 -c 能指定多個「clone source」。在 unix stackexchange 上有人寫明了這兩個的異同。使用 -p 的時候，產生的差分首先讓接收端用 subvolume snapshot 命令對 parent 子卷創建一個快照， 然後發送指令將這個快照修改成目標子卷的樣子，而使用 -c 的時候，首先在接收端用 subvolume create 創建一個空的子卷，隨後發送指令在這個子卷中填充內容，其數據塊儘量共享 clone source 已有的數據。 所以 `btrfs send -p` 在接收端產生是有共享元數據的快照，而 `btrfs send -c` 在接收端產生的是僅僅共享數據而不共享元數據的子卷。

定義中「互相共享一部分 **元數據**」比較重要，因爲除了快照的方式之外， btrfs 的子卷間也可以通過 reflink 的形式共享數據塊。我們可以對一整個子卷（甚至目錄）執行 `cp -r --reflink=always` ，創建出一個副本，副本的文件內容通過 reflink 共享原本的數據，但不共享元數據，這樣創建出的就不是快照。

說了這麼多，其實關鍵的只是 btrfs 在傳統 Unix 文件系統的「目錄/文件/inode」 這些東西之外只增加了一個「子卷」的新概念，而子卷間可以共享元數據或者數據， 用快照命令創建出的子卷就是共享一部分元數據。

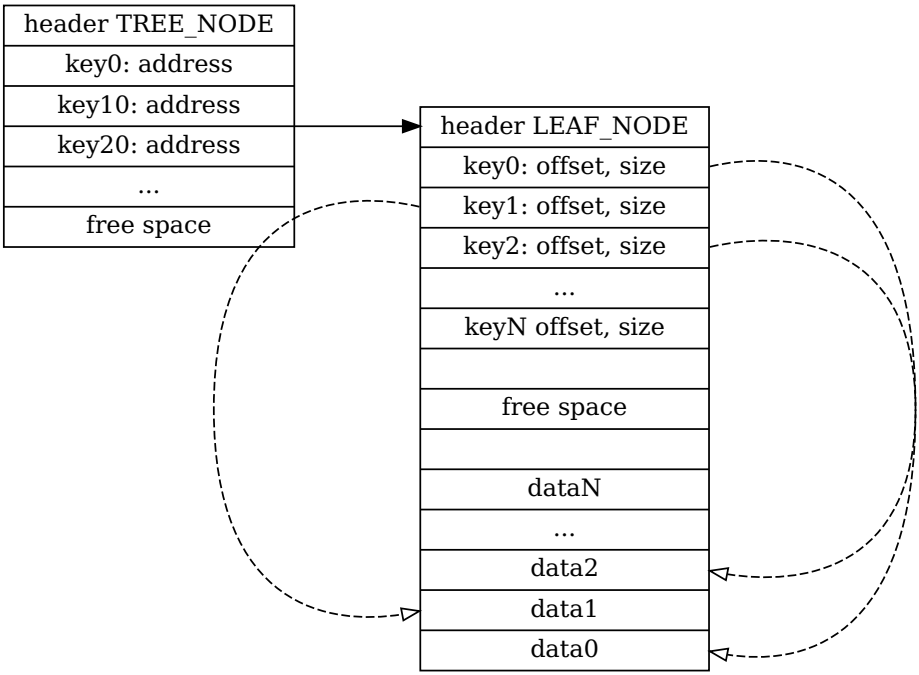
1.2 於是子卷在存儲介質中是如何記錄的呢？

首先要說明， btrfs 中大部分長度可變的數據結構都是 CoW B-tree ，一種經過修改適合寫時拷貝的B樹結構，所以在 on-disk format 中提到了很多個樹。這裏的樹不是指文件系統中目錄結構樹，而是寫時拷貝B樹（CoW B-tree，下文簡稱B樹），如果不關心B樹細節的話可以把 btrfs 所說的一棵樹理解爲關係數據庫中的一個表， 和數據庫的表一樣 btrfs 的樹的長度可變，然後表項內容根據一個 key 排序。

B樹結構由索引 key 、中間節點和葉子節點構成。每個 key 是一個 `(uint64_t object_id, uint8_t item_type, uint64_t item_extra)` 這樣的三元組，

二元組每一項的具体含义由 item_type 定義。 key 二元組構成了對象的概念，每個對象（object）在樹中用一個或多個表項（item）描述，同 object_id 的表項共同描述一個對象。B樹中的 key 只用來比較大小而不必連續，從而 object_id 也不必連續，只是按大小排序。有一些預留的 object_id 不能用作別的用途，他們的編號範圍是 -255ULL 到 255ULL，也就是表中前 255 和最後 255 個編號預留。

B樹中間節點和葉子節點結構大概像是這個樣子：



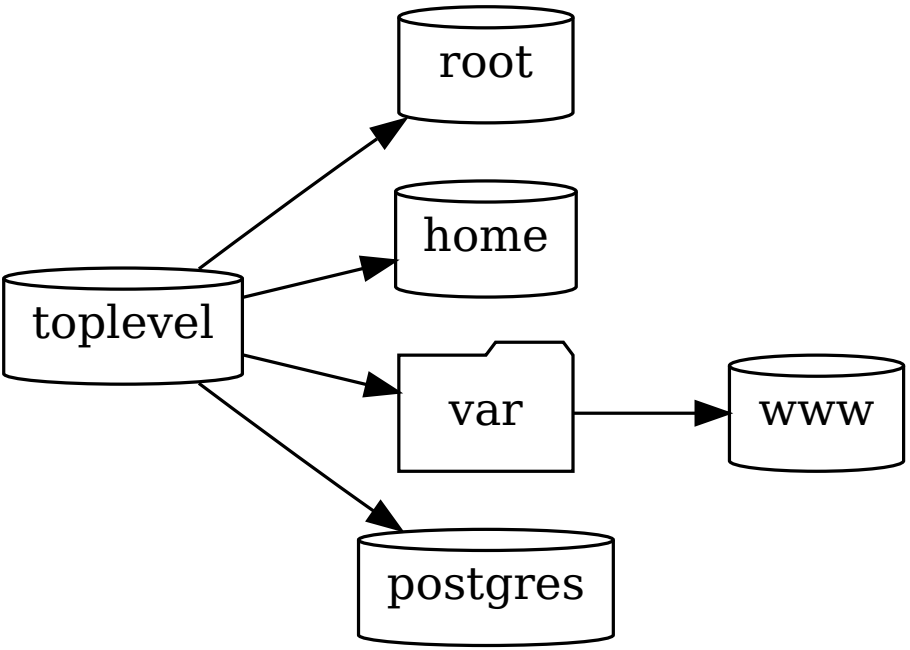
由此，每個中間節點保存一系列 key 到葉子節點的

指針，而葉子節點內保存一系列 item ，每個 item 固定大小，並指向節點內某個可變大小位置的 data 。從而邏輯上一棵B樹可以包含任何類型的 item ，每個 item 都可以有可變大小的附加數據。通過這樣的B樹結構，可以緊湊而靈活地表達很多數據類型。

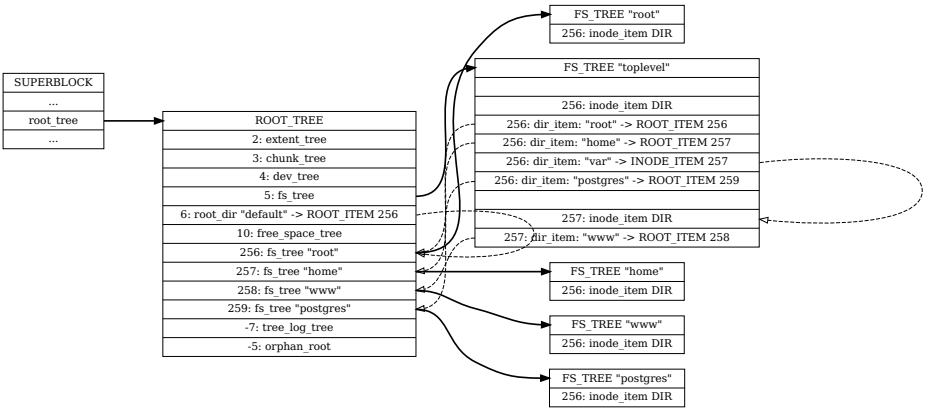
有這樣的背景之後，比如在 SysadminGuide 這頁的 Flat 佈局 有個子卷佈局的例子。

```
toplevel         (volume root direc
tory, not to be mounted by default)
    +-- root       (subvolume root
directory, to be mounted at /)
    +-- home       (subvolume root
directory, to be mounted at /home)
    +-- var        (directory)
    |   \-- www    (subvolume root
directory, to be mounted at /var/ww
w)
    \-- postgres   (subvolume root
directory, to be mounted at /var/li
b/postgresql)
```

用圓柱體表示子卷的話畫成圖大概是這個樣子：



上圖例子中的 Flat 佈局在 btrfs 中大概是這樣的數據結構， 其中實線箭頭是B樹一系列中間節點和葉子節點，邏輯上指向一棵B樹，虛線箭頭是根據 inode 號之類的編號的引用：



上圖中已經隱去了很多和本文無關的具體細節，所有這些細節都可以通過 btrfs inspect-internal 的 dump-super 和 dump-tree 查看到。

ROOT_TREE 中記錄了到所有別的B樹的指針，在一

些文檔中叫做 tree of tree roots。「所有別的B樹」 舉例來說比如 2 號 extent_tree，3 號 chunk_tree， 4 號 dev_tree，10 號 free_space_tree，這些B樹都是描述 btrfs 文件系統結構非常重要的組成部分，但是在本文關係不大， 今後有機會再討論它們。在 ROOT_TREE 的 5 號對象有一個 fs_tree，它描述了整個 btrfs pool 的頂級子卷，也就是圖中叫 toplevel 的那個子卷（有些文檔用定冠詞稱 the FS_TREE 的時候就是在說這個 5 號樹，而不是別的子卷的 FS_TREE）。除了頂級子卷之外，別的所有子卷的 object_id 在 256ULL 到 -256ULL 的範圍之間，對子卷而言 ROOT_TREE 中的這些 object_id 也同時是它們的 子卷 id，在內核掛載文件系統的時候可以用 subvolid 找到它們，別的一些對子卷的操作也可以直接用 subvolid 表示一個子卷。 ROOT_TREE 的 6 號對象描述的不是一棵樹，而是一個名叫 default 的特殊目錄，它指向 btrfs pool 的默認掛載子卷。最初 mkfs 的時候，這個目錄指向 ROOT_ITEM 5，也就是那個頂級子卷，之後可以通過命令 `btrfs subvolume set-default` 修改它指向別的子卷，這裏它被改爲指向 ROOT_ITEM 256 亦即那個名叫 "root" 的子卷。

每一個子卷都有一棵自己的 FS_TREE （有的文檔中叫 file tree），一個 FS_TREE 相當於傳統 Unix 文件系統中的一整個 inode table，只不過它除了包含 inode 信息之外還包含所有文件夾內容。在 FS_TREE 中，object_id 同時也是它所描述對象的 inode 號，所以 btrfs 的 **子卷有互相獨立的 inode 編號**，不同子卷中的文件或目錄可以擁有相同的 inode。 或許有人不太清楚

子卷間 inode 編號獨立意味着什麼，簡單地說，這意味

子卷間 inode 編號獨立意味着什麼，簡單地說，這意味着你不能跨子卷創建 hard link，不能跨子卷 mv 移動文件而不產生複製操作。不過因爲 reflink 和 inode 無關，可以跨子卷創建 reflink，也可以用 reflink + rm 的方式快速「移動」文件（這裏移動加引號是因爲 inode 變了，傳統上不算移動）。

FS_TREE 中一個目錄用一個 inode_item 和多個 dir_item 描述， inode_item 是目錄自己的 inode，那些 dir_item 是目錄的內容。 dir_item 可以指向別的 inode_item 來描述普通文件和子目錄， 也可以指向 root_item 來描述這個目錄指向一個子卷。有人或許疑惑，子卷就沒有自己的 inode 麼？其實如果看 數據結構定義 的話 `struct btrfs_root_item` 結構在最開頭的地方包含了一個 `struct btrfs_inode_item` 所以 root_item 也同時作爲子卷的 inode，不過用戶通常看不到這個子卷的 inode，因爲子卷在被（手動或自動地）掛載到目錄上之後， 用戶會看到的是子卷的根目錄的 inode。

比如上圖 FS_TREE toplevel 中，有兩個對象，第一個 256 是（子卷的）根目錄，第二個 257 是 "var" 目錄，256 有4個子目錄，其中 "root" "home" "postgres" 這三個指向了 ROOT_TREE 中的對應子卷，而 "var" 指向了 inode 257。然後 257 有一個子目錄叫 "www" 它指向了 ROOT_TREE 中 object_id 爲 258 的子卷。

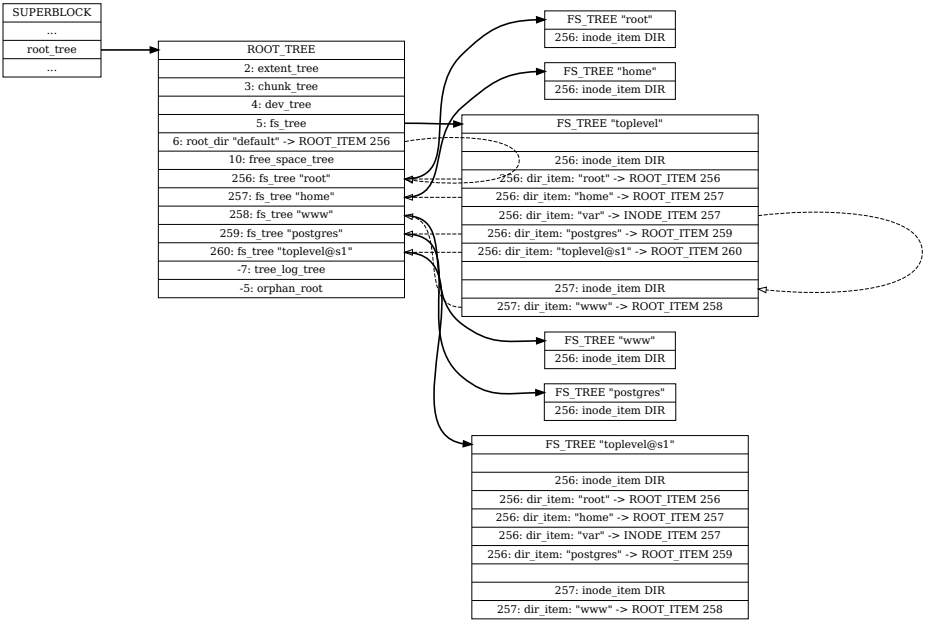
1.3 那麼快照又是如何記錄的

1.3 那麼快照又是如何記錄的呢？

以上是子卷、目錄、 inode 在 btrfs 中的記錄方式，你可能想知道，如何記錄一個快照呢？ 特別是，如果對一個包含子卷的子卷創建了快照，會得到什麼結果呢？ 如果我們在上面的佈局基礎上執行：

```
 1  btrfs subvolume snapshot toplevel to
plevel/toplevel@s1
```

那麼產生的數據結構大概如下所示：


SUPERBLOCK
...
root_tree
...
ROOT_TREE
2: extent_tree
3: chunk_tree
4: dev_tree
5: fs_tree
6: root_dir "default" -> ROOT_ITEM 256
10: free_space_tree
256: fs_tree "root"
257: fs_tree "home"
258: fs_tree "www"
259: fs_tree "postgres"
260: fs_tree "toplevel@s1"
-7: tree_log_tree
-5: orphan_root
FS_TREE "root"
256: inode_item DIR
FS_TREE "home"
256: inode_item DIR
FS_TREE "toplevel"
256: inode_item DIR
256: dir_item: "root" -> ROOT_ITEM 256
256: dir_item: "home" -> ROOT_ITEM 257
256: dir_item: "var" -> INODE_ITEM 257
256: dir_item: "postgres" -> ROOT_ITEM 259
256: dir_item: "toplevel@s1" -> ROOT_ITEM 260
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258
FS_TREE "www"
256: inode_item DIR
FS_TREE "postgres"
256: inode_item DIR
FS_TREE "toplevel@s1"
256: inode_item DIR
256: dir_item: "root" -> ROOT_ITEM 256
256: dir_item: "home" -> ROOT_ITEM 257
256: dir_item: "var" -> INODE_ITEM 257
256: dir_item: "postgres" -> ROOT_ITEM 259
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258

在 ROOT_TREE 中增加了 260 號子卷，其內容複製自 toplevel 子卷，然後 FS_TREE toplevel 的 256 號 inode 也就是根目錄中增加一個 dir_item 名叫 *toplevel@s1* 它指向 ROOT_ITEM 的 260 號子卷。這裏看似是完整複製了整個 FS_TREE 的內容，這是因爲 CoW b-tree 當只有一個葉子節點時就複製整個葉子節點。如果子卷內容再多一些，除了葉子之外還有中間節點， 那麼只有被修改的葉子和其上的中間節點需要複製。從而創建快照的開銷基本上是 O(level of FS_TREE)，而B樹的高度一般都能維持在很低的程度，所以快照創建速度近乎是常數開銷。

從子卷和快照的這種實現方式，可以看出：**雖然子卷可以嵌套子卷，但是對含有嵌套子卷的子卷做快照的語義有些特別**。上圖中我沒有畫 *toplevel@s1* 下的各個子卷到對應 ROOT_ITEM 之間的虛線箭頭， 是因爲這時候如果你嘗試直接跳過 *toplevel* 掛載 *toplevel@s1* 到掛載點， 會發現那些子卷沒有被自動掛載，更奇怪的是那些子卷的目錄項也不是個普通目錄， 嘗試往它們中放東西會得到無權訪問的錯誤，對它們能做的唯一事情是手動將別的子卷掛載在上面。 推測原因在於這些子目錄並

不是真的目錄，沒有對應的目錄的 inode ，試圖查看它

們的 inode 號會得到 2 號，而這是個保留號不應該出現在 btrfs 的 inode 號中。 每個子卷創建時會記錄包含它的上級子卷，用 `btrfs subvolume list` 可以看到每個子卷的 top level subvolid ，猜測當掛載 A 而 A 中嵌套的 B 子卷記錄的上級子卷不是 A 的時候， 會出現上述奇怪行爲。嵌套子卷的快照還有一些別的奇怪行爲，大家可以自己探索探索。

建議用平坦的子卷佈局

因爲上述嵌套子卷在做快照時的特殊行爲， 我個人建議是 **保持平坦的子卷佈局** ，也就是說：

1. 只讓頂層子卷包含其它子卷，除了頂層子卷之外的子卷只做手工掛載，不放嵌套子卷
2. 只在頂層子卷對其它子卷做快照，不快照頂層子卷
3. 雖然可以在頂層子卷放子卷之外的東西（文件或目錄），不過因爲想避免對頂層子卷做快照， 所以避免在頂層子卷放普通文件。

btrfs 的子卷可以設置「可寫」或者「只讀」，在創建一個快照的時候也可以通過 `-r` 參數創建出一個只讀快照。通常只讀快照可能比可寫的快照更有用，因爲 `btrfs send` 命令只接受只讀快照作爲參考點。子卷可以有兩種方式切換它是否只讀的屬性，可以通過 `btrfs property set <subvol> ro` 直接修改是否只讀，也

可以對只讀子卷用 `btrfs subvolume snapshot` 創建出可寫子卷，或者反過來對可寫子卷創建出只讀子卷。

只讀快照也有些特殊的限制，在 SysadminGuide#Special_Cases 就提到一例，你不能把只讀快照用 mv 移出包含它的目錄，雖然你能用 mv 給它改名或者移動包含它的目錄 到別的地方。 btrfs wiki 上給出這個限制的原因是子卷中記錄了它的上級， 所以要移動它到別的上級需要修改這個子卷，從而只讀子卷沒法移動到別的上級（ 不過我還沒搞清楚子卷在哪兒記錄了它的上級，記錄的是上級目錄還是上級子卷）。不過這個限制可以通過 對只讀快照在目標位置創建一個新的只讀快照，然後刪掉原位置的只讀快照來解決。

2 ZFS 的文件系統、快照、克隆及其它

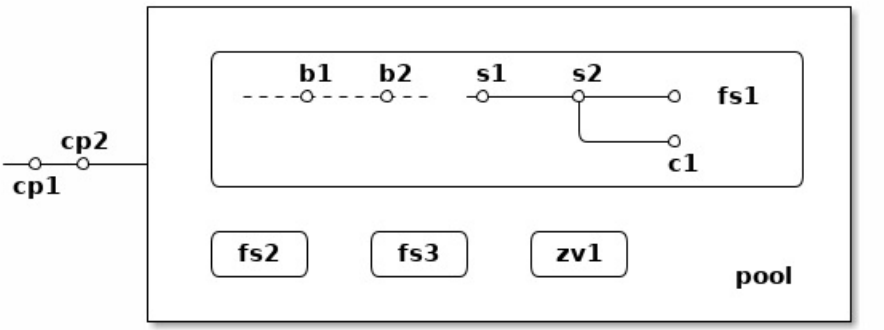
Btrfs 給傳統文件系統只增加了子卷的概念，相比之下 ZFS 中類似子卷的概念有好幾個，據我所知有這些：

- 數據集（dataset）
- 文件系統（filesystem）
- 快照（snapshot）
- 克隆（clone）

- 書籤（bookmark）：從 ZFS on Linux v0.6.4 開

始
- 檢查點（checkpoint）：從 ZFS on Linux v0.8.0 開始

梳理一下這些概念之間的關係也是最初想寫下這篇筆記的初衷。先畫個簡圖，隨後逐一講講這些概念：



上圖中，假設我們有一個 pool，其中有 3 個文件系統叫 fs1~fs3 和一個 zvol 叫 zv1，然後文件系統 fs1 有兩個快照 s1 和 s2，和兩個書籤 b1 和 b2。pool 整體有兩個檢查點 cp1 和 cp2。這個簡圖將作爲例子在後面介紹這些概念。

2.1 ZFS 設計中和快照相關的一些術語和概念

數據集

ZFS 中把文件系統、快照、克隆、zvol 等概念統稱

爲數據集（dataset）。 一些文檔和介紹中把文件系統叫做數據集，大概因爲在 ZFS 中，文件系統是最先創建並且最有用的數據集。

在 ZFS 的術語中，把底層管理和釋放存儲設備空間的叫做 ZFS 存儲池（pool）， 簡稱 zpool ，其上可以容納多個數據集，這些數據集用類似文件夾路徑的語法 `pool_name/dataset_path@snapshot_name` 這樣來稱呼。 存儲池中的數據集一同共享可用的存儲空間，每個數據集單獨跟蹤自己所消耗掉的存儲空間。

數據集之間有類似文件夾的層級父子關係，這一點有用的地方在於可以在父級數據集上設定一些 ZFS 參數， 這些參數可以被子級數據集繼承，從而通過層級關係可以方便地微調 ZFS 參數。在 btrfs 中目前還沒有類似的屬性繼承的功能。

zvol 的概念和本文關係不大，可以參考我上一篇 ZFS 子系統筆記中 ZVOL 的說明 。用 zvol 能把 ZFS 當作一個傳統的卷管理器，繞開 ZFS 的 ZPL（ZFS Posix filesystem Layer） 層。在 Btrfs 中可以用 loopback 塊設備某種程度上模擬 zvol 的功能。

文件系統

創建了 ZFS 存儲池後，首先要在其中創建文件系統（filesystem），才能在文件系統中存儲文件。 容易看出 ZFS 文件系統的概念直接對應 btrfs 中的子卷。文件系統（filesystem）這個術語， 從命名方式來看或許是想

要和（像 Solaris 的 SVM 或者 Linux 的 LVM 這樣的）傳統的卷管理器 與其上創建的多個文件系統（Solaris UFS 或者 Linux ext）這樣的上下層級做類比。 從 btrfs 的子卷在內部結構中叫作 FS_TREE 這一點可以看出，至少在 btrfs 早期設計中大概也是把子卷稱爲 filesystem 做過類似的類比的。 和傳統的卷管理器與傳統文件系統的上下層級不同的是， ZFS 和 btrfs 中由存儲池跟蹤和管理可用空間， 做統一的數據塊分配和釋放，沒有分配的數據塊算作整個存儲池中所有 ZFS 文件系統或者 btrfs 子卷的可用空間。

與 btrfs 的子卷不同的是， ZFS 的文件系統之間是完全隔離的，（除了後文會講的 dedup 方式之外）不可以共享任何數據或者元數據。一個文件系統還包含了隸屬於其中的快照（snapshot）、 克隆（clone）和書籤（bookmark）。在 btrfs 中一個子卷和對其創建的快照之間雖然有父子關係， 但是在 ROOT_TREE 的記錄中屬於平級的關係。

上面簡圖中 pool 裏面包含 3 個文件系統，分別是 fs1~3 。

快照

ZFS 的快照對應 btrfs 的只讀快照，是標記數據集在某一歷史時刻上的只讀狀態。 和 btrfs 的只讀快照一樣， ZFS 的快照也兼作 send/receive 時的參考點。 快照隸屬

於一個數據集，這說明 ZFS 的文件系統或者 zvol 都可以

創建快照。

ZFS 中快照是排列在一個時間線上的，因爲都是只讀快照，它們是數據集在歷史上的不同時間點。 這裏說的時間不是系統時鐘的時間，而是 ZFS 中事務組（TXG, transaction group）的一個序號。 整個 ZFS pool 的每次寫入會被合併到一個事務組，對事務組分配一個嚴格遞增的序列號， 提交一個事務組具有類似數據庫中事務的語義：要麼整個事務組都被完整提交，要麼整個 pool 處於上一個事務組的狀態，即使中間發生突然斷電之類的意外也不會破壞事務語義。 因此 ZFS 快照就是數據集處於某一個事務組時的狀態。

如果不滿於對數據集進行的修改，想把整個數據集恢復到之前的狀態，那麼可以回滾（rollback）數據集到一個快照。回滾操作會撤銷掉對數據集的所有更改，並且默認參數下只能回滾到最近的一個快照。 如果想回滾到更早的快照，可以先刪掉最近的幾個，或者可以使用 `zfs rollback -r` 參數刪除中間的快照並回滾。

除了回滾操作，還可以直接只讀訪問到快照中的文件。 ZFS 的文件系統中有個隱藏文件夾叫 ".zfs" ，所以如果只想回滾一部分文件，可以從 ".zfs/snapshots/SNAPSHOT-NAME" 中把需要的文件複製出來。

比如上面簡圖中 fs1 就有 `pool/fs1@s1` 和 `pool/fs1@s2` 這兩個快照， 那麼可以在 fs1 掛載點下 `.zfs/snapshots/s1` 的路徑直接訪問到 s1 中的內容。

克隆

ZFS 的克隆（clone）有點像 btrfs 的可寫快照。因爲 ZFS 的快照是只讀的，如果想對快照做寫入，那需要先用 `zfs clone` 從快照中建出一個克隆，創建出的克隆和快照共享元數據和數據， 然後對克隆的寫入不影響數據集原本的寫入點。 創建了克隆之後，作爲克隆參考點的快照會成爲克隆的依賴，克隆存在期間無法刪除掉作爲其依賴的快照。

一個數據集可以有多個克隆，這些克隆都獨立於數據集當前的寫入點。使用 `zfs promote` 命令可以把一個克隆「升級」成爲數據集的當前寫入點，從而數據集原本的寫入點會調轉依賴關係， 成爲這個新寫入點的一個克隆，被升級的克隆原本依賴的快照和之前的快照會成爲新數據集寫入點的快照。

比如上面簡圖中 fs1 有 c1 的克隆，它依賴於 s2 這個快照，從而 c1 存在的時候就不能刪除掉 s2 。

書籤

這是 ZFS 一個比較新的特性，ZFS on Linux 分支從 v0.6.4 開始支持創建書籤的功能。

書籤（bookmark）特性存在的理由是基於這樣的事實：原本 ZFS 在 send 兩個快照間的差異的時候，比如 send S1 和 S2 之間的差異，在發送端實際上只需要 S1 中記錄的時間戳（TXG id），而不需要 S1 快照的數據，

就可以計算出 S1 到 S2 的差異。在接收端則需要 S1 的完

整數據，在其上根據接收到的數據流創建 S2 。 因此在發送端，可以把快照 S1 轉變成書籤，只留下時間戳元數據而不保留任何目錄結構或者文件內容。 書籤只能作爲增量 send 時的參考點，並且在接收端需要有對應的快照，這種方式可以在發送端節省很多存儲。

通常的使用場景是，比如你有一個筆記本電腦，上面有 ZFS 存儲的數據，然後使用一個服務器上 ZFS 作爲接收端，定期對筆記本上的 ZFS 做快照然後 send 給服務器。在沒有書籤功能的時候， 筆記本上至少得保留一個和服務器上相同的快照，作爲 send 的增量參考點，而這個快照的內容已經在服務器上，所以筆記本中存有相同的快照只是在浪費存儲空間。 有了書籤功能之後，每次將定期的新快照發送到服務器之後，就可以把這個快照轉化成書籤，節省存儲開銷。

檢查點

這也是 ZFS 的新特性， ZFS on Linux 分支從 v0.8.0 開始支持創建檢查點。

簡而言之，檢查點（checkpoint）可以看作是整個存儲池級別的快照，使用檢查點能快速將整個存儲池都恢復到上一個狀態。 這邊有篇文章介紹 ZFS checkpoint 功能的背景、用法和限制 ，可以看出當存儲池中有檢查點的時候很多存儲池的功能會受影響（比如不能刪除 vdev 、不能處於 degraded 狀態、不能 scrub 到當前存儲池中已經釋放而在檢查點還在引用的數據塊）， 於是檢查點功能設計上更多是給系統管理員準備的用於調整

整個 ZFS pool 時的後悔藥， 調整結束後日用狀態下應該刪除掉所有檢查點。

2.2 ZFS 的概念與 btrfs 概念的對比

先說書籤和檢查點，因爲這是兩個 btrfs 目前完全沒有的功能。

書籤功能完全圍繞 ZFS send 的工作原理，而 ZFS send 位於 ZFS 設計中的 DSL 層面，甚至不關心它 send 的快照的數據是來自文件系統還是 zvol 。在發送端它只是從目標快照遞歸取數據塊，判斷 TXG 是否老於參照點的快照，然後把新的數據塊全部發往 send stream ；在接收端也只是完整地接收數據塊， 不加以處理，。與之不同的是 btrfs 的 send 的工作原理是工作在文件系統的只讀子卷層面， 發送端在內核代碼中根據目標快照的 b 樹和參照點快照的 generation 生成一個 diff （可以通過 `btrfs subvolume find-new` 直接拿到這個 diff ），然後在用戶態代碼中根據 diff 和參照點、目標快照的兩個只讀子卷的數據產生一連串修改文件系統的指令， 指令包括創建文件、刪除文件、讓文件引用數據塊（保持 reflink ）等操作；在接收端則完全工作在用戶態下， 根據接收到的指令重建目標快照。可見 btrfs send 需要在發送端讀取參照點快照的數據（比如找到 reflink 引用），從而 btrfs 沒法（或者很難）實現書籤功能。

檢查點也是 btrfs 目前沒有的功能。 btrfs 目前不能

對頂層子卷做遞歸的 snapshot，btrfs 的子卷也沒有類似 ZFS 數據集的層級關係和可繼承屬性，從而沒法實現類似檢查點的功能。

除了書籤和檢查點之外，剩下的概念可以在 ZFS 和 btrfs 之間有如下映射關係：

ZFS 文件系統:	btrfs 子卷
ZFS 快照:	btrfs 只讀快照
ZFS 克隆:	btrfs 可寫快照

對 ZFS 數據集的操作，大部分也可以找到對應的對 btrfs 子卷的操作。

zfs list:	`btrfs subvolume list`
zfs create:	`btrfs subvolume create`
zfs destroy:	`btrfs subvolume delete`
zfs rename:	`mv`
zfs snapshot:	`btrfs subvolume snapshot -r`
zfs rollback:	這個在 btrfs 需要對只讀快照創建出可寫的快照（用 snapshot 命令，或者直接修改讀寫屬

性），然後改名或者調整掛載點

zfs diff:	`btrfs subvolume find-new`
zfs clone:	`btrfs subvolume snapshot`
zfs promote:	和 rollback 類似，可以直接調整 btrfs 子卷的掛載點

可見雖然功能上類似，但是至少從管理員管理的角度而言， zfs 對文件系統、快照、克隆的劃分更爲清晰，對他們能做的操作也更爲明確。這也是很多從 ZFS 遷移到 btrfs ，或者反過來從 btrfs 換用 zfs 時，一些人困惑的起源（甚至有人據此說 ZFS 比 btrfs 好在 cli 設計上）。

不過 btrfs 子卷的設計也使它在系統管理上有了更大的靈活性。比如在 btrfs 中刪除一個子卷不會受制於別的子卷是否存在，而在 zfs 中要刪除一個快照必須先保證先摧毀掉依賴它的克隆。 再比如 btrfs 的可寫子卷沒有主次之分，而 zfs 中一個文件系統和其克隆之間有明顯的區別，所以需要 promote 命令調整差異。還有比如 ZFS 的文件系統只能回滾到最近一次的快照， 要回滾到更久之前的快照需要刪掉中間的快照，並且回滾之後原本的文件系統數據和快照數據就被丟棄了；而 btrfs 中因爲回滾操作相當於調整子卷的掛載，所以不需要刪掉快照， 並且回滾之後原本的子卷和快照還可以繼續保留。

加上 btrfs 有 reflink ，這給了 btrfs 在使用中更大的靈活性，可以有一些 zfs 很難做到的用法，比如想從快

靈活性，可以有一些 ZFS 很難做到的用法。 比如想從快照中打撈出一些虛擬機鏡像的歷史副本，而不想回滾整個快照的時候，在 btrfs 中可以直接 `cp --reflink=always` 將鏡像從快照中複製出來，此時的複製將和快照共享數據塊； 而在 zfs 中只能用普通 cp 複製，會浪費很多存儲空間。

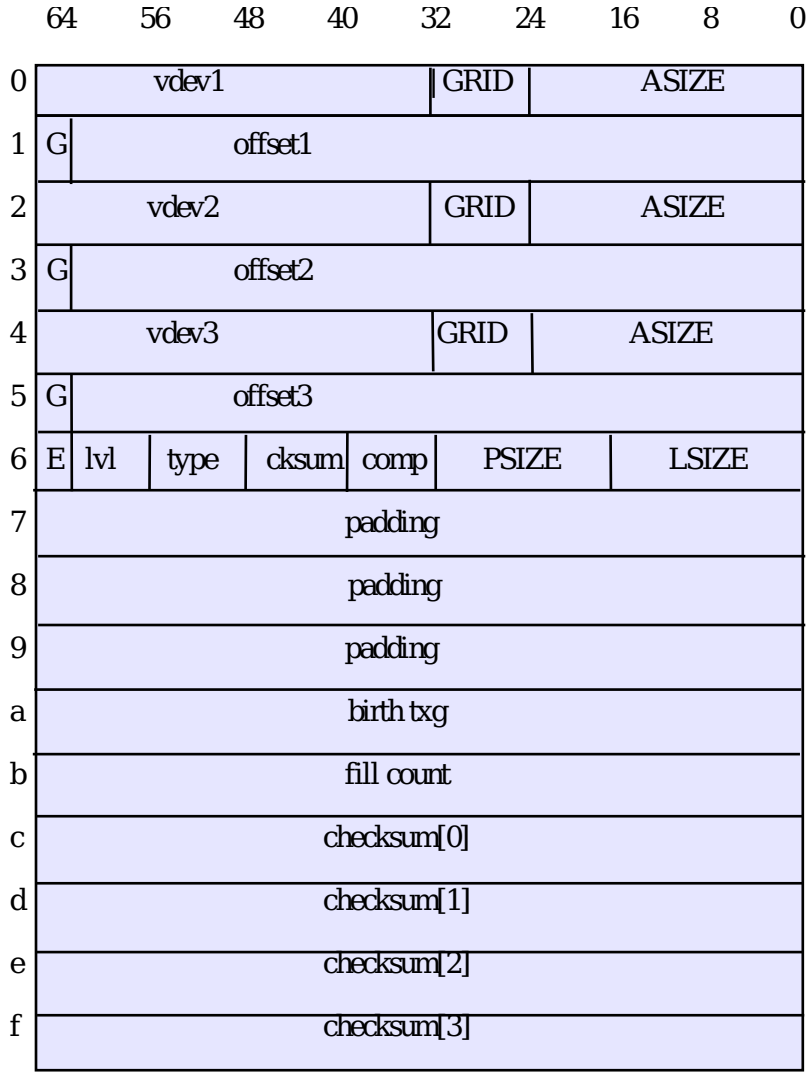
2.3 ZFS 中是如何存儲這些數據集的呢

要講到存儲細節，首先需要 瞭解一下 ZFS 的分層設計 。不像 btrfs 基於現代 Linux 內核，有許多現有文件系統已經實現好的基礎設施可以利用， 並且大體上只用到一種核心數據結構（CoW的B樹）；ZFS 則脫胎於 Solaris 的野心勃勃， 設計時就分成很多不同的子系統，逐步提升抽象層次， 並且每個子系統都發明了許多特定需求下的數據結構來描述存儲的信息。 在這裏和本文內容密切相關的是 ZPL 、 DSL 、 DMU 這些 ZFS 子系統。

Sun 曾經寫過一篇 ZFS 的 On disk format 對理解 ZFS 如何存儲在磁盤上很有幫助，雖然這篇文檔是針對 Sun 還在的時候 Solaris 的 ZFS ，現在 ZFS 的內部已經變化挺大，不過對於理解本文想講的快照的實現方式還具有參考意義。這裏藉助這篇 ZFS On Disk Format 中的一些圖示來解釋 ZFS 在磁盤上的存儲方式。

ZFS 的塊指針

ZFS 中用的 128 字節塊指針



要理解 ZFS 的磁盤結構首先想介紹一下 ZFS 中的塊指針（block pointer，blkptr_t），結構如右圖所

指針（block pointer，blkptr_t），結構如右圖所示。 ZFS 的塊指針用在 ZFS 的許多數據結構之中，當需要從一個地方指向任意另一個地址的時候都會 插入這樣的一個塊指針結構。大多數文件系統中也有類似的指針結構，比如 btrfs 中有個8字節大小的邏輯地址（logical address），一般也就是個 4字節 到 16字節 大小的整數寫着扇區號、塊號或者字節偏移，在 ZFS 中的塊指針則是一個巨大的128字節（不是 128bit !）的結構體。

128字節塊指針的開頭是3個數據虛擬地址（DVA, Data Virtual Address），每個 DVA 是 128bit，其中記錄這塊數據在什麼設備（vdev）的什麼偏移（offset）上佔用多大（asize)，有 3個 DVA 槽是用來存儲最多3個不同位置的副本。然後塊指針還記錄了這個塊用什麼校驗算法（ cksum ）和什麼壓縮算法（comp），壓縮前後的大小（PSIZE/LSIZE），以及256bit的校驗和（checksum）。

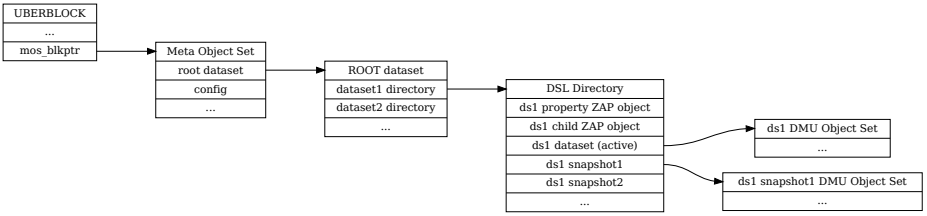
當需要間接塊（indirect block）時，塊指針中記錄了間接塊的層數（lvl），和下層塊指針的數量（fill）。一個間接塊就是一個數據塊中包含一個塊指針的數組，當引用的對象很大需要很多塊時，間接塊構成一棵樹狀結構。

塊指針中還有和本文關係很大的一個值 birth txg，記錄這個塊指針誕生時的整個 pool 的 TXG id 。一次 TXG 提交中寫入的數據塊都會有相同的 birth txg，這個相當於 btrfs 中 generation 的概念。 實際上現在的 ZFS 塊指針似乎記錄了兩個 birth txg，分別在圖中的9行和a行的位置， 一個 physical 一個 logical，用於 dedup 和 device removal，值得注意的是塊指針裏只有 birth txg

device removal。值得注意的是塊指針裏只有 birth txg，沒有引用計數或者別的機制做引用，這對後面要講的東西很關鍵。

DSL 的元對象集

理解塊指針和 ZFS 的子系統層級之後，就可以來看看 ZFS 存儲在磁盤上的具體結構了。 因爲涉及的數據結構種類比較多，所以先來畫一張邏輯上的簡圖，其中箭頭只是某種引用關係不代表塊指針， 方框也不是結構體細節：

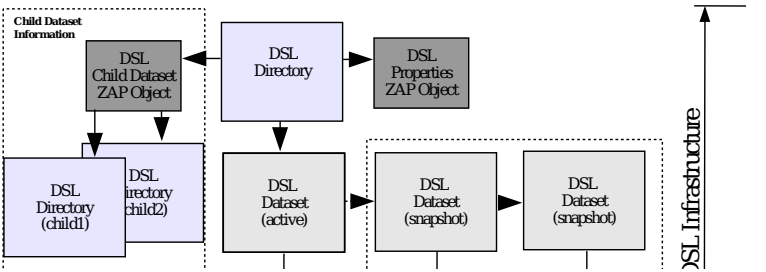


如上簡圖所示，首先 ZFS pool 級別有個 uberblock，具體每個 vdev 如何存儲和找到這個 uberblock 今後有空再聊，這裏認爲整個 zpool 有唯一的一個 uberblock。從 uberblock 有個指針指向元對象集（MOS, Meta Object Set），它是個 DMU 的對象集，它包含整個 pool 的一些配置信息，和根數據集（root dataset）。根數據集再包含整個 pool 中保存的所有頂層數據集，每個數據集有一個 DSL Directory 結構。然後從每個數據集的 DSL Directory 可以找到一系列子數據集和一系列快照等結

構。最後每個數據集有個 active 的 DMU 對象集，這是整個文件系統的當前寫入點。每個快照也指向一個各自

的 DMU 對象集。

DSL 層的每個數據集的邏輯結構也可以用下面的圖表達（來自 ZFS On Disk Format ）：



ZFS On Disk Format 中 4.1 節的 DSL infrastructure

ZFS On Disk Format 中 4.2 節的 Meta Object Set

需要記得 ZFS 中沒有類似 btrfs 的 CoW b-tree 這樣的統一數據結構，所以上面的這些設施是用各種不同的數據結構表達的。 尤其每個 Directory 的結構可以包含一個 ZAP 的鍵值對存儲，和一個 DMU 對象。 可以理解爲， DSL 用 DMU 對象集（Objectset）表示一個整數（uinit64_t 的 dnode 編號）到 DMU 對象的映射，然後用 ZAP 對象表示一個名字到整數的映射，然後又有很多額外的存儲於 DMU 對象中的 DSL 結構體。如果我們畫出不同的指針和不同的結構體，那麼會得到一個稍顯複雜的圖，見右邊「ZFS On Disk Format 中 4.2 節的 Meta Object Set」，圖中還只畫到了 root_dataset 爲止。

看到這裏，大概可以理解在 ZFS 中創建一個 ZFS 快照的操作其實很簡單：找到數據集的 DSL Directory 中當

前 active 的 DMU 對象集指針，創建一個表示 snapshot 的 DSL dataset 結構，指向那個 DMU 對象集，然後快照就建好了。因爲今後對 active 的寫入會寫時複製對應的 DMU 對象集，所以 snapshot 指向的 DMU 對象集不會變化。

3 創建快照這麼簡單麼？那麼刪除快照呢？

按上面的存儲格式細節來看， btrfs 和 zfs 中創建快照似乎都挺簡單的，利用寫時拷貝，創建快照本身沒什麼複雜操作。

如果你也聽到過別人介紹 CoW 文件系統時這麼講，是不是會覺得似乎哪兒少了點什麼。創建快照是挺簡單的， **直到你開始考慮如何刪除快照** ……

或者不侷限在刪除單個快照上， CoW 文件系統因爲寫時拷貝，每修改一個文件內容或者修改一個文件系統結構， 都是分配新數據塊，然後考慮是否要刪除這個數據替換的老數據塊，此時如何決定老數據塊能不能刪呢？ 刪除快照的時候也是同樣，快照是和別的文件系統有共享一部分數據和元數據的， 所以顯然不能把快照引

用到的數據塊都直接刪掉，要考察快照引用的數據塊是否還在別的地方被引用着， 只能刪除那些沒有被引用的

數據。

深究「如何刪快照」這個問題，就能看出 WAFL 、btrfs 、 ZFS 甚至別的 log-structured 文件系統間的關鍵區別，從而也能看到另一個問題的答案： **爲什麼 btrfs 只需要子卷的抽象，而 zfs 搞出了這麼多抽象概念？** 帶着這兩個疑問，我們來研究一下這些文件系統的塊刪除算法。

3.1 日誌結構文件系統中用的垃圾回收算法

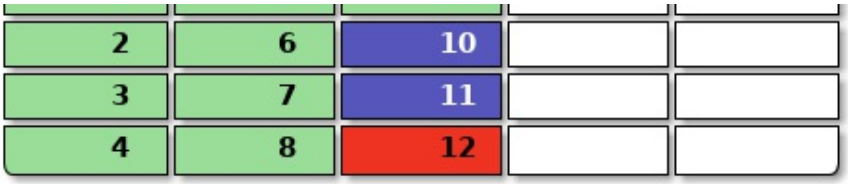
講 btrfs 和 zfs 用到的刪除算法之前，先講一下日誌結構（log-structured）文件系統中的垃圾回收（ GC, Garbage Collection）算法。對熟悉編程的人來說，講到空間釋放算法，大概首先會想到 GC ，因爲這裏要解決的問題乍看起來很像編程語言的內存管理中 GC 想要解決的問題：有很多指針相互指向很多數據結構，找其中沒有被引用的垃圾然後釋放掉。

首先要澄清一下 日誌結構文件系統（log-structured file system） 的定義，因爲有很多文件系統用日誌，而用了日誌的不一定是日誌結構文件系統。 在維基百科上有個頁面介紹 日誌結構文件系統 ，還有個 列表列出了一些日誌結構文件系統 。通常說，整個文件系統的存儲結構都組織成一個大日誌的樣子，就說這個文件系統是日誌結構的， 這包括很多早期學術研究的文件

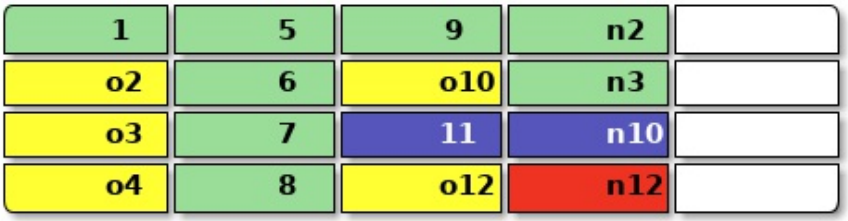
系統，以及目前 NetBSD 的 LFS 、Linux 的 NILFS ，用在光盤介質上的 UDF ，還有一些專門爲閃存優化的 JFFS 、 YAFFS 以及 F2FS 。日誌結構文件系統不包括那些用額外日誌保證文件系統一致性，但文件系統結構不在日誌中的 ext4 、 xfs 、 ntfs 、 hfs+ 。

簡單來說，日誌結構文件系統就是把存儲設備當作一個大日誌，每次寫入數據時都添加在日誌末尾， 然後用寫時複製重新寫入元數據，最後提交整個文件系統結構。因爲這裏用了寫時複製，原本的數據塊都還留着，所以可以很容易實現快照之類的功能。從這個特徵上來說，寫時拷貝文件系統（CoW FS）像 btrfs/zfs 這些在一些人眼中也符合日誌結構文件系統的特徵， 所以也有人說寫時拷貝文件系統算是日誌結構文件系統的一個子類。不過日誌結構文件系統的另一大特徵是利用 GC 回收空間，這裏是本文要講的區別，所以在我看來不用 GC 的 btrfs 和 zfs 不算是日誌結構文件系統。

舉個例子，比如下圖是一個日誌結構文件系統的磁盤佔用，其中綠色是數據，藍色是元數據（比如目錄結構和 inode），紅色是文件系統級關鍵數據（比如最後的日誌提交點），一開始可能是這樣，有9個數據塊， 2個元數據塊，1個系統塊：



現在要覆蓋 2 和 3 的內容，新寫入 n2 和 n3，再刪除 4 號的內容，然後修改 10 裏面的 inode 變成 n10 引用這些新數據，然後寫入一個新提交 n12，用黃色表示不再被引用的垃圾，提交完大概是這樣：



日誌結構文件系統需要 GC 比較容易理解，寫日誌嘛，總得有一個「添加到末尾」的寫入點，比如上面圖中的 n12 就是當前的寫入點。空盤上連續往後寫而不 GC 總會遇到空間末尾，這時候就要覆蓋寫空間開頭， 就很難判斷「末尾」在什麼地方，而下一次寫入需要在哪裏了。 這時文件系統也不知道需要回收哪些塊（圖中的 o2 o3 o4 o10 和 o12），因爲這些塊可能被別的地方還繼續引用着，需要等到 GC 時掃描元數據來判斷。

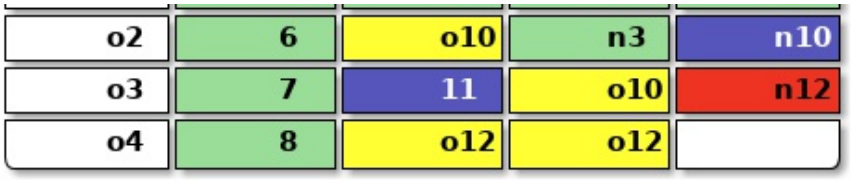
和內存管理時的 GC 不同的一點在於，文件系統的 GC 肯定不能停下整個世界跑 GC，也不能把整個地址空間對半分然後 Mark-and-Sweep，這些在內存中還尚可的簡單策略直接放到文件系統中絕對是性能災難。所以文件系統的 GC 需要並行的後臺 GC，並且需要更細粒度

的分塊機制能在 Mark-and-Sweep 的時候保持別的地方可以繼續寫入數據而維持文件系統的正常職能。

通常文件系統的 GC 是這樣，先把整個盤分成幾個段（segment）或者區域(zone)，術語不同不過表達的概念類似， 然後 GC 時挑一個老段，掃描文件系統元數據找出要釋放的段中還被引用的數據塊，搬運到日誌末尾，最後整個釋放一段。 搬運數據塊時，也要調整文件系統別的地方對被搬運的數據塊的引用。

物理磁盤上一般有扇區的概念，通常是 512B 或者 4KiB 的大小，在文件系統中一般把連續幾個物理塊作爲一個數據塊， 大概是 4KiB 到 1MiB 的數量級，然後日誌結構文件系統中一個段(segment)通常是連續的很多塊，數量級來看大約是 4MiB 到 64MiB 這樣的數量級。相比之下 ufs/ext4/btrfs/zfs 的分配器通常還有 block group 的概念， 大概是 128MiB 到 1GiB 的大小。可見日誌結構文件系統的段，是位於數據塊和其它文件系統 block group 中間的一個單位。段大小太小的話，會顯著增加空間管理需要的額外時間空間開銷，而段大小太大的話， 又不利於利用整個可用空間，這裏的抉擇有個平衡點。

繼續上面的例子，假設上面文件系統的圖示中每一列的4塊是一個段，想要回收最開頭那個段， 那麼需要搬運還在用的 1 到空閒空間，順帶修改引用它的 n10 ，最後提交 n12 ：



要掃描並釋放一整段，需要掃描整個文件系統中別的元數據（圖中的 n12 和 n10 和 11）來確定有沒有引用到目標段中的地址，可見釋放一個段是一個 \(O(N)\) 的操作，其中 N 是元數據段的數量，按文件系統的大小增長， 於是刪除快照之類可能要連續釋放很多段的操作在日誌文件系統中是個 \(O(N^2)\) 甚至更昂贵的操作。 在文件系統相對比較小而系統內存相對比較大的時候，比如手機上或者PC讀寫SD卡，大部分元數據塊（ 其中包含塊指針）都能放入內存緩存起來的話，這個掃描操作的開銷還是可以接受的。 但是對大型存儲系統顯然掃描並釋放空間就不合適了。

段的抽象用在閃存類存儲設備上的一點優勢在於，閃存通常也有擦除塊的概念，比寫入塊的大小要大， 是連續的多個寫入塊構成，從而日誌結構的文件系統中一個段可以直接對應到閃存的一個擦除塊上。 所以閃存設備諸如U盤或者 SSD 通常在底層固件中用日誌結構文件系統模擬一個塊設備，來做寫入平衡。 大家所說的 SSD 上固件做的 GC ，大概也就是這樣一種操作。

基於段的 GC 還有一個顯著缺陷，需要掃描元數據，複製搬運仍然被引用到的塊，這不光會增加設備寫入，還需要調整現有數據結構中的指針，調整指針需要更多

寫入，同時又釋放更多數據塊， F2FS 等一些文件系統設計中把這個問題叫 Wandering Tree Problem ，在 F2FS

設計中是通過近乎「作弊」的 NAT 轉換表 放在存儲設備期待的 FAT 所在位置，不僅能讓需要掃描的元數據更集中，還能減少這種指針調整導致的寫入。

不過基於段的 GC 也有一些好處，它不需要複雜的文件系統設計，不需要特殊構造的指針， 就能很方便地支持大量快照。一些日誌結構文件系統比如 NILFS 用這一點支持了「連續快照（continuous snapshots）」，每次文件系統提交都是自動創建一個快照，用戶可以手動標記需要保留哪些快照， GC 算法則排除掉用戶手動標記的快照之後，根據快照創建的時間，先從最老的未標記快照開始回收。 即便如此， GC 的開銷（CPU時間和磁盤讀寫帶寬）仍然是 NILFS 最爲被人詬病的地方，是它難以被廣泛採用的原因。 爲了加快 NILFS 這類日誌文件系統的 GC 性能讓他們能更適合於普通使用場景，也有許多學術研究致力於探索和優化 GC ，使用更先進的數據結構和算法跟蹤數據塊來調整 GC 策略，比如這裏有一篇 State-of-the-art Garbage Collection Policies for NILFS2 。

3.2 WAFL 早期使用的可用空間位圖數組

從日誌結構文件系統使用 GC 的困境中可以看出，文件系統級別實際更合適的， 可能不是在運行期依賴掃描

元數據來計算空間利用率的 GC，而是在創建快照時或者寫入數據時就預先記錄下快照的空間利用情況，從而可以細粒度地跟蹤空間和回收空間，這也是 WAFL 早期實現快照的設計思路。

WAFL 早期記錄快照佔用數據塊的思路從表面上來看也很「暴力」，傳統文件系統一般有個叫做「位圖（bitmap）」的數據結構，用一個二進制位記錄一個數據塊是否佔用，靠掃描位圖來尋找可用空間和已用空間。WAFL 的設計早期中考慮既然需要支持快照，那就把記錄數據塊佔用情況的位圖，變成快照的數組。於是整個文件系統有個 256 大小的快照利用率數組，數組中每個快照記錄自己佔用的數據塊位圖，文件系統中最多能容納 255 個快照。

	block1	block2	block3	block4	block5	...	block N
filesystem	1	1	1	1	1	...	N
snapshot1	1	0	1	1	1	...	N
snapshot2	1	1	1	0	1	...	0
...	1	0	1	0	1	...	0
snapshot255	1	0	0	0	1	...	0

上面每個單元格都是一個二進制位，表示某個快照有沒有引用某個數據塊。有這樣一個位圖的數組之後，就可以直接掃描位圖判斷出某個數據塊是否已經佔用，可以找出尚未被佔用的數據塊用作空間分配，也可以方便地計算每個快照引用的空間大小或者獨佔的空間大小，估算刪除快照後可以釋放的空間。

需要注意的是，文件系統中可以有非常多的塊，從而位圖數組比位圖需要更多的元數據來表達。比如估算

一下傳統文件系統中一塊可以是 4KiB 大小，那麼跟蹤空間利用的位圖需要 1bit/4KiB ， 1TiB 的盤就需要 32MiB 的元數據來存放位圖； 而 WAFL 這種位圖數組即便限制了快照數量只能有255個，仍需要 256bit/4KiB 的空間開銷， 1TiB 的盤需要的元數據開銷陡增到 8GiB ，這些還只是單純記錄空間利用率的位圖數組，不包括別的元數據。

使用這麼多元數據表示快照之後，創建快照的開銷也相應地增加了，需要複製整個位圖來創建一個新的快照， 按上面的估算 1TiB 的盤可能需要複製 32MiB 的位圖，這不再是一瞬能完成的事情， 期間可能需要停下所有對文件系統的寫入等待複製完成。 位圖數組在存儲設備上的記錄方式也很有講究，當刪除快照時希望能快速讀寫上圖中的一整行位圖， 於是可能希望每一行位圖的存儲方式在磁盤上都儘量連續， 而在普通的寫入操作需要分配新塊時，想要按列的方式掃描位圖數組，找到沒有被快照佔用的塊， 從而上圖中按列的存儲表達也希望在磁盤上儘量連續。 WAFL 的設計工程師們在位圖數組的思路下，實現了高效的數據結構讓上述兩種維度的操作都能快速完成， 但是這絕不是一件容易的事情。

位圖數組的表達方式也有其好處，比如除了快照之外，也可以非常容易地表達類似 ZFS 的克隆和獨立的文件系統這樣的概念，這些東西和快照一樣，佔用僅有的 256 個快照數量限制。 這樣表達的克隆可以有數據塊和

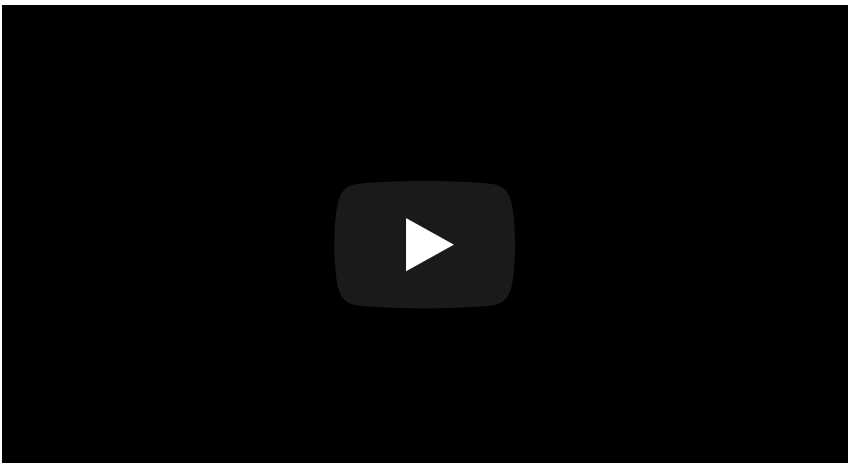
別的文件系統共享，文件系統之間也可以有類似 reflink 的機制共享數據塊，在位圖數組的相應位置將位置1即

可。

使用位圖數組的做法，也只是 WAFL 早期可能採用的方式，由於 WAFL 本身是閉源產品， 難以獲知它具體的工作原理。哈佛大學和 NetApp 的職員曾經在 FAST10 (USENIX Conference on File and Storage Technologies) 上發表過一篇講解高效跟蹤和使用 back reference 的論文，叫 Tracking Back References in a Write-Anywhere File System ，可以推測在新一代 WAFL 的設計中可能使用了類似 btrfs backref 的實現方式，接下來會詳細介紹。

3.3 ZFS 中關於快照和克隆的空間跟蹤算法

How ZFS snapshots really work And why they perform well (usually)



幻燈片可以從這裏下載

OpenZFS 的項目領導者，同時也是最初設計 ZFS 中

OpenZFS 的項目領導者，同時也是最初設計 ZFS 中 DMU 子系統的作者 Matt Ahrens 在 DMU 和 DSL 中設計並實現了 ZFS 獨特的快照的空間跟蹤算法。他也在很多地方發表演講，講過這個算法的思路和細節， 比如右側就是他在 BSDCan 2019 做的演講 How ZFS snapshots really work And why they perform well (usually) 的 YouTube 視頻。

其中 Matt 講到了三個刪除快照的算法，分別可以叫做「烏龜算法」、「兔子算法」、「豹子算法」， 接下來簡單講講這些算法背後的思想和實現方式。

烏龜算法：概念上 ZFS 如何刪快照

烏龜算法沒有實現在 ZFS 中，不過方便理解 ZFS 在概念上如何考慮快照刪除這個問題，從而幫助理解 後面的兔子算法和豹子算法。

要刪除一個快照， ZFS 需要找出這個快照引用到的「獨佔」數據塊，也就是那些不和別的數據集或者快照共享的 數據塊。 ZFS 刪除快照基於這幾點條件：

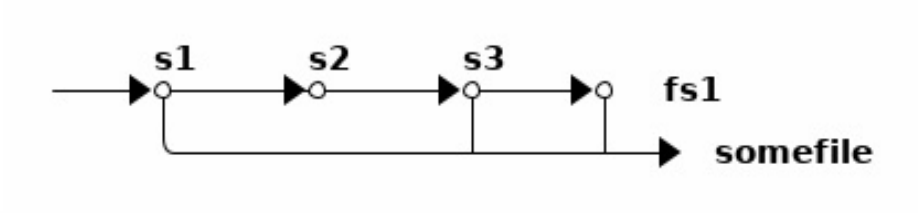
1. ZFS 快照是只讀的。創建快照之後無法修改其內容。
2. ZFS 的快照是嚴格按時間順序排列的，這裏的時間指 TXG id ，即記錄文件系統提交所屬事務組的嚴格遞增序號。
3. ZFS 不存在 reflink 之類的機制，從而在某個時間點刪除掉的數據塊，不可能在此之後的快照中

點刪除掉的數據塊，不可能在比它更後面的快照中「復活」。

第三點關於 reflink 造成的數據復活現象可能需要解釋一下，比如在（支持 reflink 的） btrfs 中有如下操作：

```
 1  btrfs subvolume snapshot -r fs s1
 2  rm fs/somefile
 3  btrfs subvolume snapshot -r fs s2
 4  cp --reflink=always s1/somefile fs/s
omefile
 5  btrfs subvolume snapshot -r fs s3
```

我們對 fs 創建了 s1 快照，刪除了 fs 中某個文件，創建了 s2 快照，然後用 reflink 把剛剛刪除的文件從 s1 中複製出來，再創建 s3 。如此操作之後，按時間順序有 s1、s2、s3 三個快照：



其中只有 s2 不存在 somefile ，而 s1 、 s3 和當前的 fs 都有，並且都引用到了同一個數據塊。 於是從時間線來看， somefile 的數據塊在 s2 中「死掉」了，又在 s3 中「復活」了。

而 ZFS (目前還）不支持 reflink ，所以沒法像這樣讓數據塊復活。一旦某個數據塊在某個快照中「死」

了，就意味着它在隨後的所有快照中都不再被引用到了。

ZFS 的快照具有的上述三點條件，使得 ZFS 的快照刪除算法可以基於 birth time。回顧上面 ZFS 的塊指針中講到，ZFS 的每個塊指針都有一個 birth txg 屬性，記錄這個塊誕生時 pool 所在的 txg。於是可以根據這個 birth txg 找到快照所引用的「獨佔」數據塊然後釋放掉它們。

具體來說，烏龜算法可以這樣刪除一個快照：

1. 在 DSL 層找出要刪除的快照（我們叫他 s），它的前一個快照（叫它 ps），後一個快照（叫它 ns），分別有各自的 birth txg 叫 s.birth, ps.birth, ns.birth。
2. 遍歷 s 的 DMU 對象集指針所引出的所有塊指針。這裏所有塊指針在邏輯上構成一個由塊指針組成的樹狀結構，可以有間接塊組成的指針樹，可以有對象集的 dnode 保存的塊指針，這些都可以看作是樹狀結構的中間節點。
    1. 每個樹節點的指針 bp，考察如果 bp.birth <= ps.birth，那麼這個指針和其下所有指針都還被前一個快照引用着，需要保留這個 bp 引出的整個子樹。
    2. 按定義 bp.birth 不可能 > s.birth。
    3. 對所有滿足 ps.birth < bp.birtu <= s.birth 的 bp，需要去遍歷 ns 的相應塊指針（同樣文件的同樣偏移位置），看是否還在引用 bp。
        - 如果有在，繼續遞歸往下考察樹狀結

- 如果存在，繼續遞歸在下一層級的樹結構中 bp 的所有子節點指針。因爲可能共享了這個 bp 但 CoW 了新的子節點。
- 如果不存在，說明下一個快照中已經刪了 bp 。這時可以確定地說 bp 是 s 的「獨佔」數據塊。

3. 釋放掉所有找到的 s 所「獨佔」的數據塊。

上述算法的一些邊角情況可以自然地處理，比如沒有後一個快照時使用當前數據集的寫入點， 沒有前一個快照時那麼不被後一個快照引用的數據塊都是當前要刪除快照的獨佔數據塊。

分析一下烏龜算法的複雜度的話，算法需要分兩次，讀 s 和 ns 中引用到的所有 ps 之後創建的數據塊的指針，重要的是這些讀都是在整個文件系統範圍內的隨機讀操作，所以速度非常慢……

兔子算法：死亡列表算法（ZFS早期）

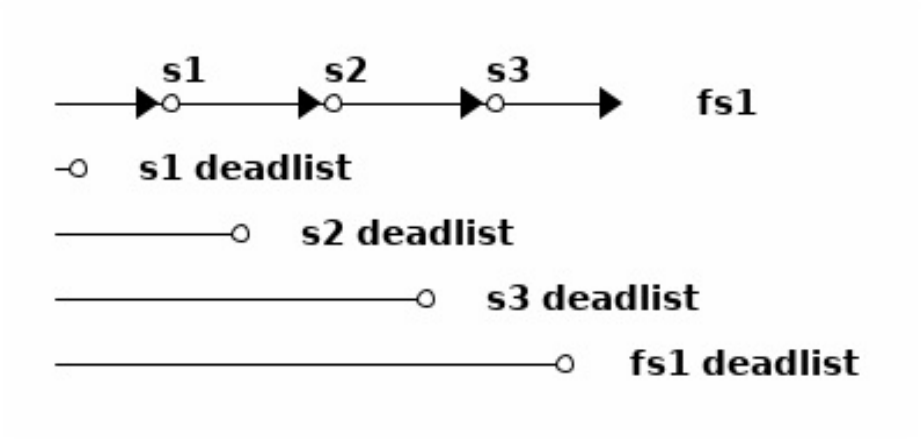
可以粗略地認爲烏龜算法算是用 birth txg 優化代碼路徑的 GC 算法，利用了一部分元數據中的 birth txg 信息來避免掃描所有元數據，但是概念上仍然是在掃描元

數據找出快照的獨佔數據塊， 而非記錄和跟蹤快照的數據塊，在最壞的情況下仍然可能需要掃描幾乎所有元數據

據。

兔子算法基於烏龜算法的基本原理，在它基礎上跟蹤快照所引用數據塊的一些信息， 從而很大程度上避免了掃描元數據的開銷。ZFS 在早期使用這個算法跟蹤數據集和快照引用數據塊的情況。

兔子算法爲每個數據集（文件系統或快照）增加了一個數據結構，叫死亡列表（dead list）， 記錄 **前一個快照中還活着，而當前數據集中死掉了的數據塊指針**，換句話說就是在本數據集中「殺掉」的數據塊。舉例畫圖大概是這樣



上圖中有三個快照和一個文件系統，共 4 個數據集。每個數據集維護自己的死亡列表， 死亡列表中是那些在該數據集中被刪掉的數據塊。於是兔子算法把烏龜算法所做的操作分成了兩部分， 一部分在文件系統刪除數據時記錄死亡列表，另一部分在刪除快照時根據死亡列表釋放需要釋放的塊。

在當前文件系統刪除數據塊（不再被當前文件系統引用）時，負責比對 birth txg 維護當前文件系統的死亡列表。每刪除一個數據塊，指針爲 bp 時，判斷 bp.birth

列表。每刪除一個數據塊，指針爲 bp 時，判斷 bp.birth 和文件系統最新的快照（上圖爲 s3）的 birth：

- bp.birth <= s3.birth： 說明 bp 被 s3 引用，於是將 bp 加入 fs1 的 deadlist
- bp.birth > s3.birth：說明 bp 指向的數據塊誕生於 s3 之後，可以直接釋放 bp 指向的塊。

創建新快照時，將當前文件系統（圖中 fs1）的死亡列表交給快照，文件系統可以初始化一個空列表。

刪除快照時，我們有被刪除的快照 s 和前一個快照 ps 、後一個快照 ns ，需要讀入當前快照 s 和後一個快照 ns 的死亡列表：

1. 對 s.deadlist 中的每個指針 bp
    - 複製 bp 到 ns.deadlist
2. 對 ns.deadlist 中的每個指針 bp （其中包含了上一步複製來的）
    - 如果 bp.birth > ps.birth ，釋放 bp 的空間
    - 否則保留 bp

換個說法的話， **死亡列表記錄的是每個數據集需要負責刪除，但因爲之前的快照還引用着所以不能刪除的數據塊列表** 。從當前文件系統中刪除一個數據塊時，這個職責最初落在當前文件系統身上，隨後跟着創建新快照職責被轉移到新快照上。 每個負責的數據集根據數據

塊的出生時間是否早於之前一個快照來判斷現在是否能立刻釋放該塊， 刪除一個快照時則重新評估自己負責的和下一個快照負責的數據塊的出生時間

和下一個快照真實的數據塊的出生時間。

從所做的事情來看，兔子算法並沒有比烏龜算法少做很多事情。烏龜算法刪除一個快照， 需要遍歷當前快照和後一個快照兩組數據塊指針中，新寫入的部分；兔子算法則需要遍歷當前快照和後一個快照兩個死亡列表中，新刪除的塊指針。 但是實際兔子算法能比烏龜算法快不少，因爲維護死亡列表的操作只在文件系統刪除數據時和刪除快照時， 順序寫入，並且刪除快照時也只需要順序讀取死亡列表。在磁盤這種塊設備上，順序訪問能比隨機訪問有數量級的差異。

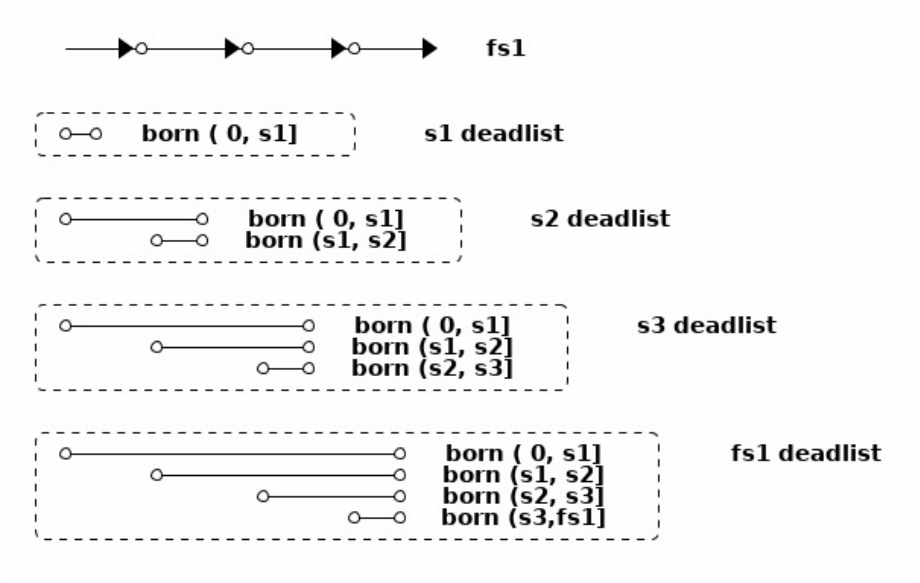
不過記錄死亡列表也有一定存儲開銷。最差情況下，比如把文件系統寫滿之後，創建一個快照， 再把所有數據都刪掉，此時文件系統引用的所有數據塊的塊指針都要保存在文件系統的死亡列表中。 按 ZFS 默認的 128KiB 數據塊大小，每塊需要 128 字節的塊指針，存儲這些死亡列表所需開銷可能要 整個文件系統大小的 1/1024 。如果用 4KiB 的數據塊大小，所需開銷則是 1/32 ， 1TiB 的盤會有 32GiB 拿來存放這些塊指針，將高於用位圖數組所需的存儲量。

豹子算法：死亡列表的子列表

豹子算法是 ZFS 後來在 2009 年左右實現的算法。在兔子算法中就可以看到，每次刪除快照操作死亡列表的時候， 都需要掃描死亡列表中的塊指針，根據指針中記錄的 birth txg 做判斷是否能直接釋放或是需要保留到另一個快照的死亡列表。 於是豹子算法的思路是，在死亡列表中記錄塊指針時，就把其中的塊指針按 birth txg 分

列表中記錄塊指針時，就把其中的塊指針按 birth txg 分成子列表（sublist）。

比如上面兔子算法中那4個死亡列表，可以這樣拆成子列表：



這樣拆成子列表之後，每次從死亡列表中釋放數據塊都能根據出生時間找到對應的子列表， 然後連續釋放整個子列表。每次合併死亡列表時，也能直接用單鏈表穿起需要合併的子列表，不需要複製塊指針。

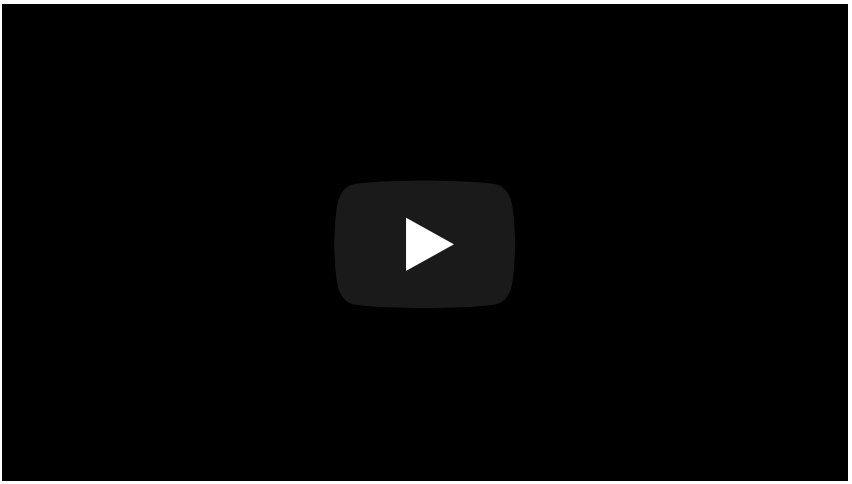
死亡列表並不在跟蹤快照的獨佔大小，而是在跟蹤快照所需負責刪除的數據塊大小， 從這個數值可以推算出快照的獨佔大小之類的信息。 有了按出生時間排列的死亡列表子列表之後，事實上給任何一個出生時間到死亡時間的範圍， 都可以找出對應的幾個子列表，從而根據子列表的大小可以快速計算出每個快照範圍的「獨佔」數據塊、 「共享」數據塊等大小，這不光在刪除快照時很有用，也可以用來根據大小估算 zfs send 或者別的基於快照操作時需要的時間。

的基於快照操作時需要的時間。

從直覺上理解，雖然 ZFS 沒有直接記錄每個數據塊屬於哪個數據集，但是 ZFS 跟蹤記錄了每個數據塊的歸屬信息，也就是說由哪個數據集負責釋放這個數據塊。在文件系統中刪除數據塊或者快照時，這個歸屬信息跟着共享數據塊轉移到別的快照中，直到最終被釋放掉。

生存日誌：ZFS 如何管理克隆的空間佔用

Fast Clone Deletion by Sara Hartse



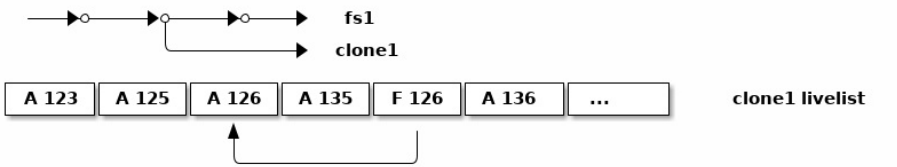
以上三種算法負責在 ZFS 中跟蹤快照的空間佔用，它們都基於數據塊的誕生時間，所以都假設 ZFS 中對數據塊的分配是位於連續的快照時間軸上。但是明顯 ZFS 除了快照和文件系統， 還有另一種數據集可能分配數據塊，那就是 克隆 ，於是還需要在克隆中使用不同的算法單獨管理因克隆而分配的數據塊。 OpenZFS Summit

2017 有個演講 Fast Clone Deletion by Sara Hartse 解釋了其中的細節。

首先克隆的存在本身會鎖住克隆引用到的快照，不能刪除這些被依賴的快照， 所以克隆無須擔心靠快照共享的數據塊的管理問題。因此克隆需要管理的，是從快照分離之後， 新創建的數據塊。

和烏龜算法一樣，原理上刪除克隆的時候可以遍歷克隆引用的整個 DMU 對象集，找出其中晚於快照的誕生時間的數據塊，然後釋放它們。也和烏龜算法一樣， 這樣掃描整個對象集的開銷很大，所以使用一個列表來記錄數據塊指針。 克隆管理新數據塊的思路和快照的兔子算法維持死亡列表的思路相反， 記錄所有新誕生的數據塊，這個列表叫做「生存日誌（livelist）」。

克隆不光要記錄新數據塊的誕生，還要記錄新數據塊可能的死亡，所以磁盤上保存的生存日誌雖然叫 livelist ，但不像死亡列表那樣是列表的形式，而是日誌的形式，而內存中保存的生存日誌則組織成了棵 自平衡樹（AVLTree） 來加速查找。



磁盤上存儲的生存日誌如上圖，每個表項記錄它是分配（A）或者刪除（F）一個數據塊，同時記錄數據塊的地址。 這些記錄在一般情況下直接記錄在日誌末尾，

隨着對克隆的寫入操作而不斷增長，長到一定程度則從內存中的 AVL Tree 直接輸出一個新的生存日誌替代掉舊的，合併其中對應的分配和刪除操作。

生存日誌可以無限增長，從而要將整個生存列表載入內存也有不小的開銷，這裏的解決方案有點像快照管理中用 豹子算法改進兔子算法的思路，把一個克隆的整個生存日誌也按照數據塊的誕生時間拆分成子列表。Sara Hartse 的演講 Fast Clone Deletion 中繼續解釋了其中的細節和優化方案，感興趣的可以看看。

3.4 btrfs 的空間跟蹤算法：引用計數與反向引用

理解了 ZFS 中根據 birth txg 管理快照和克隆的算法之後，可以發現它們基於的假設難以用於 WAFL 和 btrfs。 ZFS 嚴格區分文件系統、快照、克隆，並且不存在 reflink，從而可以用 birth txg 判斷數據塊是否需要保留，而 WAFL 和 btrfs 中不存在 ZFS 的那些數據集分工，又想支持 reflink，可見單純基於 birth txg 不足以管理 WAFL 和 btrfs 子卷。

讓我們回到一開始日誌結構文件系統中基於垃圾回收（GC）的思路上來，作爲程序員來看， 當垃圾回收的性能不足以滿足當前需要時，大概很自然地會想到：引

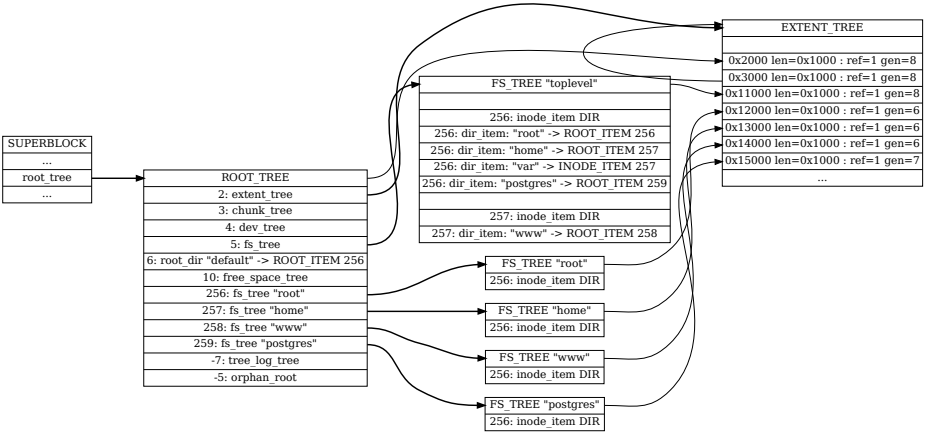
用計數（reference counting）。 編程語言中用引用計數作爲內存管理策略的缺陷是：強引用不能成環， 這在文件系統中看起來不是很嚴重的問題，文件系統總體上看是個樹狀結構，或者就算有共享的數據也是個 上下層級分明的有向圖，很少會使用成環的指針，以及文件系統記錄指針的時候也都會區分指針的類型， 根據指針類型可以分出強弱引用。

EXTENT_TREE 和引用計數

btrfs 中就是用引用計數的方式跟蹤和管理數據塊的。引用計數本身不能保存在 FS_TREE 或者指向的數據塊中，因爲這個計數需要能夠變化，對只讀快照來說整個 FS_TREE 都是只讀的。 所以這裏增加一層抽象，btrfs 中關於數據塊的引用計數用一個單獨的 CoW B樹來記錄，叫做 EXTENT_TREE ，保存於 ROOT_TREE 中的 2 號對象位置。

btrfs 中每個塊都是按 區塊（extent） 的形式分配的，區塊是一塊連續的存儲空間，而非 zfs 中的固定大小。每個區塊記錄存儲的位置和長度， 以及這裏所說的引用計數。所以本文最開始講 Btrfs 的子卷和快照 中舉例的那個平坦佈局，如果畫上 EXTENT_TREE 大概像是下

圖這樣，其中每個粗箭頭是一個區塊指針，指向磁盤中的邏輯地址，細箭頭則是對應的 EXTENT_TREE 中關於這塊區塊的描述：


SUPERBLOCK
...
root_tree
...
ROOT_TREE
2: extent_tree
3: chunk_tree
4: dev_tree
5: fs_tree
6: root_dir "default" -> ROOT_ITEM 256
10: free_space_tree
256: fs_tree "root"
257: fs_tree "home"
258: fs_tree "www"
259: fs_tree "postgres"
-7: tree_log_tree
-5: orphan_root
FS_TREE "toplevel"
256: inode_item DIR
256: dir_item: "root" -> ROOT_ITEM 256
256: dir_item: "home" -> ROOT_ITEM 257
256: dir_item: "var" -> INODE_ITEM 257
256: dir_item: "postgres" -> ROOT_ITEM 259
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258
FS_TREE "root"
256: inode_item DIR
FS_TREE "home"
256: inode_item DIR
FS_TREE "www"
256: inode_item DIR
FS_TREE "postgres"
256: inode_item DIR
EXTENT_TREE
0x2000 len=0x1000 : ref=1 gen=8
0x3000 len=0x1000 : ref=1 gen=8
0x11000 len=0x1000 : ref=1 gen=8
0x12000 len=0x1000 : ref=1 gen=6
0x13000 len=0x1000 : ref=1 gen=6
0x14000 len=0x1000 : ref=1 gen=6
0x15000 len=0x1000 : ref=1 gen=7
...

btrfs 中關於 `chattr +C` 關閉了 CoW 的文件的處理

2020年2月20日補充

這裏從 EXTENT_TREE 的記錄可以看出，每個區塊都有引用計數記錄。對用 `chattr +C` 關閉了 CoW 的文件而言，文件數據同樣還是有引用計數，可以和別的文件或者快照共享文件數據的。 這裏的特殊處理在於，每次寫入一個 nocow 的文件的時候，考察這個文件指向區塊的引用計數， 如果引用計數 >1 ，表示這個文件的區塊發生過 reflink ，那會對文件內容做一次 CoW 斷開 reflink 並寫入新位置；如果引用計數 =1 ，那麼直接原地寫入文件內容而不 CoW 。於是 nocow 的文件仍然能得到 reflink 和 snapshot 的功能， 使用這些功能仍然會造成文件碎片並伴隨性能損失，只是在引用計數爲 1 的時

候不發生 CoW 。

包括 ROOT_TREE 和 EXTENT_TREE 在內，btrfs 中所有分配的區塊（extent）都在 EXTENT_TREE 中有對應的記錄，按區塊的邏輯地址索引。從而給定一個區塊，能從 EXTENT_TREE 中找到 ref 字段描述這個區塊有多少引用。不過 ROOT_TREE 、 EXTENT_TREE 和別的一些 pool-wide 數據結構本身不依賴引用計數的，這些數據結構對應的區塊的引用計數總是 1 ，不會和別的樹共享區塊；從 FS_TREE 開始的所有樹節點都可以共享區塊，這包括所有子卷的元數據和文件數據，這些區塊對應的引用計數可以大於 1 表示有多處引用。

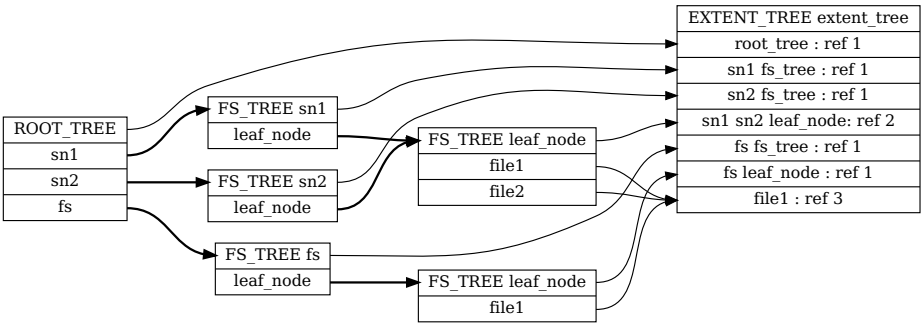
EXTENT_TREE 按區塊的邏輯地址索引，記錄了起始地址和長度，所以 EXTENT_TREE 也兼任 btrfs 的空間利用記錄，充當別的文件系統中 block bitmap 的職責。比如上面例子中的 extent_tree 就表示 `[0x2000,0x4000)` `[0x11000,0x16000)` 這兩段連續的空間是已用空間，剩下的空間按定義則是可用空間。爲了加速空間分配器， btrfs 也有額外的 free space cache 記錄在 ROOT_TREE 的 10 號位置 free_space_tree 中，不過在 btrfs 中這個 free_space_tree 記錄的信息只是緩存，必要時可以通過 `btrfs check --clear-space-cache` 扔掉這個緩存重新掃描 extent_tree 並重建可用空間記錄。

比如我們用如下命令創建了兩個文件，通過 reflink 讓它們共享區塊，然後創建兩個快照， 然後刪除文件系統中的 file2 ：

```
1  write fs/file1
2  cp --reflink=always fs/file1 fs/file2

3  btrfs subvolume snapshot fs sn1
4  btrfs subvolume snapshot fs sn2
5  rm fs/file2
```

經過以上操作之後，整個 extent_tree 的結構中記錄的引用計數大概如下圖所示：



上圖簡化了一些細節，實際上每個文件可以引用多個區塊（文件碎片）， 其中每個對區塊的引用都可以指明引用到具體某個區塊記錄的某個地址偏移和長度， 也就是說文件引用的區塊可以不是區塊記錄中的一整個區塊，而是一部分內容。

圖中可見，整個文件系統中共有5個文件路徑可以訪問到同一個文件的內容，分別是 `sn1/file1`, `sn1/file2`, `sn2/file1`, `sn2/file2`, `fs/file1` ， 在

extent_tree 中， sn1 和 sn2 可能共享了一個 B樹 葉子節點，這個葉子節點的引用計數爲 2 ，然後每個文件的內容都指向同一個 extent ，這個 extent 的引用計數爲 3

。

刪除子卷時，通過引用計數就能準確地釋放掉子卷所引用的區塊。具體算法挺符合直覺的：

1. 從子卷的 FS_TREE 往下遍歷
    - 遇到引用計數 >1 的區塊，減小該塊的計數即可，不需要再遞歸下去
    - 遇到引用計數 =1 的區塊，就是子卷獨佔的區塊，需要釋放該塊並遞歸往下繼續掃描

大體思路挺像上面介紹的 ZFS 快照刪除的烏龜算法，只不過根據引用計數而非 birth txg 判斷是否獨佔數據塊。性能上說， btrfs 的B樹本身內容就比較緊湊，FS_TREE 一個結構就容納了文件 inode 和引用的區塊信息， EXTENT_TREE 按地址排序也比較緊湊，所以刪除算法的隨機讀寫不像 ZFS 的烏龜算法那麼嚴重， 實際實現代碼裏面也可能通過 btrfs generation 做一些類似基於 birth txg 優化的快速代碼路徑。 即便如此，掃描 FS_TREE 仍然可能需要耗時良久，這個遞歸的每一步操作都會記錄在 ROOT_TREE 中專門的結構，也就是說刪除一個子卷的操作可以執行很長時間並跨越多個 pool commit。 `btrfs subvolume delete` 命令默認也只是記錄下這個刪除操作，然後就返回一句類似： `Delete subvolume (no-commit): /subvolume/path` 的輸出，不會等刪除操作執行結束。 相比之下 ZFS 那邊刪除

一個快照或文件系統必須在一個 txg 內執行完，沒有中間過程的記錄， 所以如果耗時很久會影響整個 pool 的寫入，於是 ZFS 那邊必須對這些操作優化到能在一個 txg

內執行完的程度(摧毀克隆方面 ZFS 還有 async_destroy 優化 可能有些幫助)。

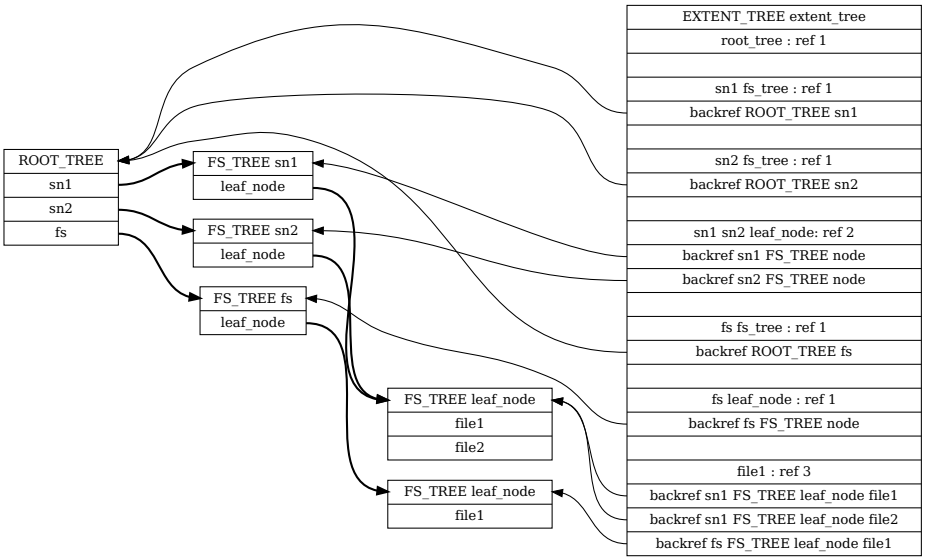
只需要引用計數就足夠完成快照的創建、刪除之類的功能，也能支持 reflink 了（仔細回想， reflink 其實就是 reference counted link 嘛），普通讀寫下也只需要引用計數。 但是只有引用計數不足以知道區塊的歸屬，不能用引用計數統計每個子卷分別佔用多少空間， 獨佔多少區塊而又共享多少區塊。上面的例子就可以看出，所有文件都指向同一個區塊，該區塊的引用計數爲 3 ，而文件系統中一共有 5 個路徑能訪問到該文件。可見從區塊根據引用計數反推子卷歸屬信息不是那麼一目瞭然的。

反向引用（back reference）

單純從區塊的引用計數難以看出整個文件系統所有子卷中有多少副本。 也就是說單有引用計數的一個數字還不夠，需要記錄具體反向的從區塊往引用源頭指的引用，這種結構在 btrfs 中叫做「反向引用（back reference，簡稱 backref）」。所以在上圖中每一個指向 EXTENT_TREE 的單向箭頭，在 btrfs 中都有記錄一條反向引用，通過反向引用記錄能反過來從被指針指向的位置找回到記錄指針的地方。

反向引用（backref）是 btrfs 中非常關鍵的機制，在 btrfs kernel wiki 專門有一篇頁面 Resolving Extent Backrefs 解釋它的原理和實現方式。

對上面的引用計數的例子畫出反向引用的指針大概是這樣：



EXTENT_TREE 中每個 extent 記錄都同時記錄了引用到這個區塊的反向引用列表。反向引用有兩種記錄方式：

1. 普通反向引用（Normal back references）。記錄這個指針來源所在是哪顆B樹、 B樹中的對象 id 和對象偏移。
    - 對文件區塊而言，就是記錄文件所在子卷、inode、和文件內容的偏移。
    - 對子卷的樹節點區塊而言，就是記錄該區塊的上級樹節點在哪個B樹的哪個位置開始。
2. 共享反向引用（Shared back references）。記錄這個指針來源區塊的邏輯地址。
    - 無論對文件區塊而言，還是對子卷的樹節點區塊而言，都是直接記錄了保存這個區塊指針的上層樹節點的邏輯地址。

有兩種記錄方式是因爲它們各有性能上的優缺點：

普通反向引用: 因爲通過對象編號記錄，所以當樹節點 CoW 改變了地址時不需要調整地址， 從而在普通的讀寫和快照之類的操作下有更好的性能， 但是在解析反向引用時需要額外一次樹查找。 同時因爲這個額外查找，普通反向引用也叫間接反向引用。

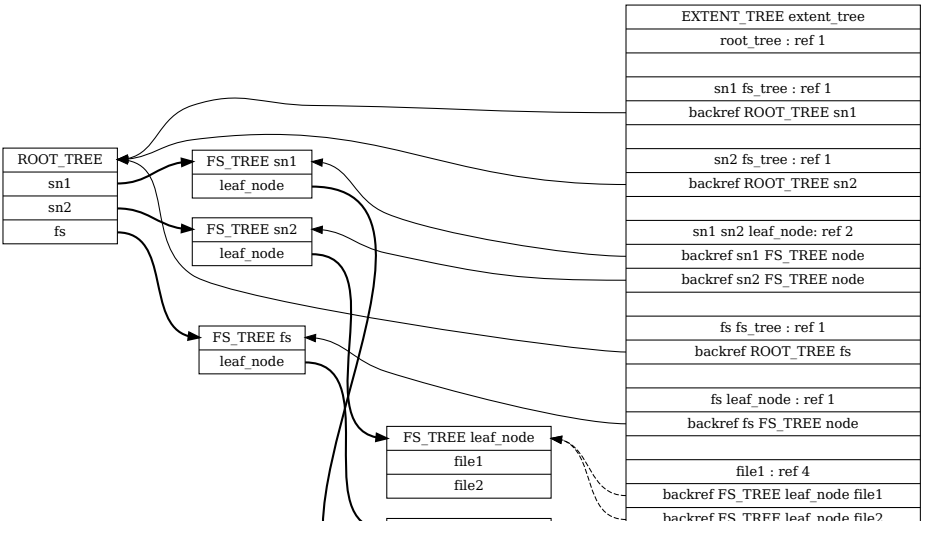
共享反向引用: 因爲直接記錄了邏輯地址，所以當這個地址的節點被 CoW 的時候也需要調整這裏記錄的地址。 在普通的讀寫和快照操作下，調整地址會增加寫入從而影響性能，但是在解析反向引用時更快。

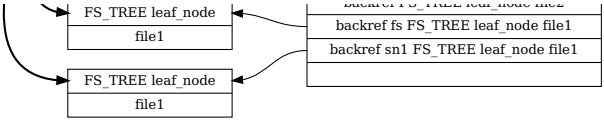
通常通過普通寫入、快照、 reflink 等方式創建出來的引用是普通反向引用， 由於普通反向引用記錄了包含它的B樹，從而可以說綁在了某棵樹比如某個子卷上， 當這個普通反向引用指向的對象不再存在，而這個反向引用還在通過別的途徑共享時， 這個普通反向引用會轉換共享反向引用；共享反向引用在存在期間不會變回普通反向引用。

比如上圖反向引用的例子中，我們先假設所有畫出的反向引用都是普通反向引用，於是圖中標爲 file1 引用數爲 3 的那個區塊有 3 條反向引用記錄，其中前兩條都指向 cp1 裏面的文件，分別是 cp1/file1 和 cp1/file2。

指向 sn1 裏面的文件，分別是 sn1/file1 和 sn1/file2，然後 sn1 和 sn2 共享了 FS_TREE 的葉子節點。

假設這時我們刪除 sn1/file2，執行了代碼 `rm sn1/file2` 之後：





那麼 sn1 會 CoW 那個和 sn2 共享的葉子節點，有了新的屬於 sn1 的葉子，從而斷開了原本 file1 中對這個共享葉子節點的兩個普通反向引用，轉化成共享反向引用（圖中用虛線箭頭描述）， 並且插入了一個新的普通反向引用指向新的 sn1 的葉子節點。

遍歷反向引用(backref walking)

有了反向引用記錄之後，可以給定一個邏輯地址，從 EXTENT_TREE 中找到地址的區塊記錄， 然後從區塊記錄中的反向引用記錄一步步往回遍歷，直到遇到 ROOT_TREE，最終確定這個邏輯地址的區塊在整個文

ROOT_TREE，最終確定這個邏輯地址的區塊在整個文件系統中有多少路徑能訪問它。 這個遍歷反向引用的操作，在 btrfs 文檔和代碼中被稱作 backref walking 。

比如還是上面的反向引用圖例中 sn1 和 sn2 完全共享葉子節點的那個例子，通過 backref walking ，我們能從 file1 所記錄的 3 個反向引用，推出全部 5 個可能的訪問路徑。

backref walking 作爲很多功能的基礎設施，從 btrfs 相當早期（3.3內核）就有，很多 btrfs 的功能實際依賴 backref walking 的正確性。列舉一些需要 backref walking 來實現的功能：

1. qgroup

   btrfs 的子卷沒有記錄子卷的磁盤佔用開銷，靠引用計數來刪除子卷， 所以也不需要詳細統計子卷的空間佔用情況。不過對一些用戶的使用場景，可能需要統計子卷空間佔用。由於 可能存在的共享元數據和數據，子卷佔用不能靠累計加減法的方式算出來，所以 btrfs 有了 qgroup 和 quota 功能，用來統計子卷或者別的管理粒度下的佔用空間情況。爲了實現 qgroup ，需要 backref walking 來計算區塊共享的情況。

2. send

   btrfs send 在計算子卷間的差異時，也通過 backref walking 尋找能靠 reflink 共享的區塊，從而避免傳輸數據。

3. balance/scrub

   balance 和 scrub 都會調整區塊的地址，通過 backref walking 能找到所有引用到這個地址的位置並正確修改地址。

4. check

   當需要打印診斷信息的時候，除了提供出錯的數據所在具體地址之外，通過 backref walking 也能提供受影響的文件路徑之類的信息。

btrfs 的 reflink-aware defrag

這裏想提一下 btrfs 一直計劃中，但是還沒有成功實現的 reflink-aware defrag 。文件碎片一直是 CoW 文件系統的大問題，對 btrfs 和對 ZFS 都是同樣。ZFS 完全不支持碎片整理， 而 btrfs 目前只提供了文件級別的碎片整理，這會切斷現有的 reflink 。計劃中的 reflink-aware defrag 也是基於 backref walking ，根據區塊引用的碎片程度，整理碎片而某種程度上保持 reflink 。btrfs 曾經實現了這個，但是因爲 bug 太多不久就取消了相關功能，目前這個工作處於停滯階段。

可見 backref walking 的能力對 btrfs 的許多功能都非常重要（不像 ZPL 的 dnode 中記錄的 parent dnode 那樣只用於診斷信息 ）。不過 backref walking 根據區塊共享的情況的不同，也可能導致挺大的運行期開銷

塊共享的情況的不同，也可能導致提大的建行期開銷，包括算法時間上的和內存佔用方面的開銷。 比如某個子卷中有 100 個文件通過 reflink 共享了同一個區塊，然後對這個子卷做了 100 個快照， 那麼對這一個共享區塊的 backref walking 結果可能解析出 10000 個路徑。可見隨着使用 reflink 和快照， backref walking 的開銷可能爆炸式增長。最近 btrfs 郵件列表也有一些用戶彙報，在大量子卷 和通過 reflink 做過 dedup 的 btrfs 文件系統上 send 快照時，可能導致內核分配大量內存甚至 panic 的情形，在 5.5 內核中 btrfs send 試圖控制 send 時 clone reference 的數量上限來緩解這種邊角問題。

值得再強調的是，在沒有開啓 qgroup 的前提下，正常創建刪除快照或 reflink ，正常寫入和覆蓋區塊之類的文件系統操作，只需要引用計數就足夠，雖然可能需要調整反向引用記錄（尤其是共享反向引用的地址），但是不需要動用 backref walking 這樣的重型武器。

4 ZFS vs btrfs 的 dedup 功能現狀

上面討論 ZFS 的快照和克隆如何跟蹤數據塊時，故意避開了 ZFS 的 dedup 功能，因爲要講 dedup 可能需要先理解引用計數在文件系統中的作用，而 btrfs 正好用了引用計數。 於是我們再回來 ZFS 這邊，看看 ZFS 的

dedup 是具體如何運作的。

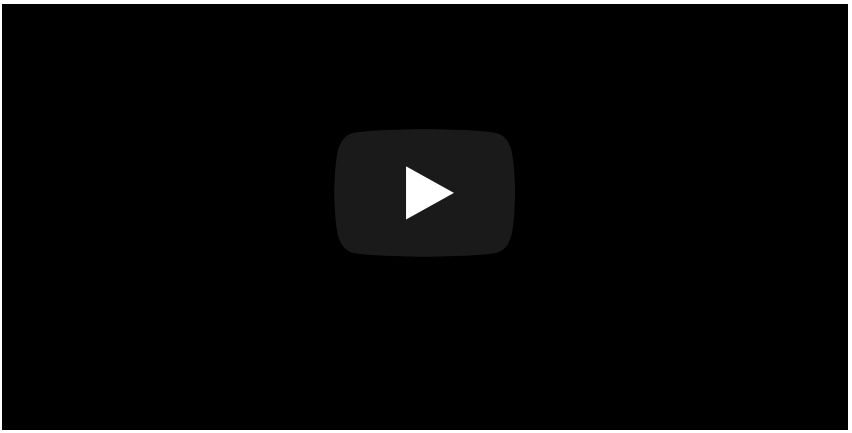
稍微瞭解過 btrfs 和 ZFS 兩者的人，或許有不少 btrfs 用戶都眼饞 ZFS 有 in-band dedup 的能力，可以在寫入數據塊的同時就去掉重複數據，而 btrfs 只能「退而求其次」地選擇第三方 dedup 方案，用外部工具掃描已經寫入的數據，將其中重複的部分改爲 reflink 。又或許有不少 btrfs 用戶以爲 zfs 的 dedup 就是在內存和磁盤中維護一個類似 Bloom filter 的結構，然後根據結果對數據塊增加 reflink ，從而 zfs 內部大概一定有類似 reflink 的設施，進一步質疑爲什麼 btrfs 還遲遲沒有實現這樣一個 Bloom filter 。 或許還有從 btrfs 轉移到 ZFS 的用戶有疑惑， 爲什麼 ZFS 還沒有暴露出 reflink 的用戶空間接口 ，或者既然 ZFS 已經有了 dedup ， 能不能臨時開關 dedup 來提供類似 reflink 式的共享數據塊 而避免 ZFS 長期開 dedup 導致的巨大性能開銷。

看過上面 ZFS 中關於快照和克隆的空間跟蹤算法 之後我們會發現，其實 ZFS 中並沒有 能對應 btrfs reflink 的功能，而是根據數據塊指針中的 birth txg 來跟蹤快照和克隆的共享數據塊的。這引來更多疑惑：

4.1 ZFS 是如何實現 dedup 的？

Dedup Performance by Matt Ahrens

[redacted]



ZFS 是在 Sun/OpenSolaris 壽命相當晚期的 2009 年獲得的 dedup 功能，就在 Oracle 收購 Sun，OpenSolaris 分裂出 Illumos 從而 ZFS 分裂出 Oracle ZFS 和 OpenZFS 的時間點之前。因此 關於 ZFS dedup 如何實現的文檔相對匱乏，大部分介紹 ZFS 的文檔或者教程會講到 ZFS dedup 的用法，但是對 dedup 的實現細節、性能影響、乃至使用場景之類的話題就很少提了（甚至很多教程講了一堆用法之後說類似，「我評估之後覺得我不需要開 dedup，你可以自己評估一下」這樣的建議）。

OpenZFS Summit 2017 上 Matt 有個演講，主要內容關於今後如何改進 dedup 性能的計劃，其中講到的計劃還沒有被具體實現，不過可以窺探一下 dedup 現在在 ZFS 中是如何工作的。 Chris 的博客也有兩篇文章《

What I can see about how ZFS deduplication seems to work on disk》和《An important addition to how ZFS deduplication works on the disk》介紹了他對此的認識，在這裏我也嘗試來總結一下 ZFS dedup 特性如

的認識，在這裏我也會試圖總結一下 ZFS dedup 特性如何工作。

ZFS dedup 是存儲池級別（pool-wide）開關的特性，所以大概在 MOS 之類的地方有存儲一個特殊的數據結構， 叫 DeDup Table 簡稱 DDT 。DDT 目前是存儲設備上的一個 hash table ，因爲是存儲池級別的元數據，所以在 ZFS 中存儲了三份完全一樣的 DDT ，DDT 的內容是大概如下結構：

Check	DVA(Data Virtual Address)	Refc
0x12345678	vdev=1 addr=0x45671234	3
0x5678efab	vdev=2 addr=0x37165adb	0
0x98765432	vdev=1 addr=0xac71be12	1
0xabcd1234	vdev=0 addr=0xc1a2231d	5
… …	… …	… …

DDT 中對每個數據塊存有3個東西：數據塊的 checksum 、DVA （就是 ZFS 的塊指針 中的 DVA）和引用計數。在存儲池開啓 dedup 特性之後，每次新寫入一個數據塊，都會先計算出數據塊的 checksum ，然後查找 DDT ，存在的話增加 DDT 條目的引用計數，不存在的話插入 DDT 條目。每次釋放一個數據塊，同樣需要查找 DDT 調整引用計數。

除了 DDT 之外，文件系統中記錄的塊指針中也有個特殊標誌位記錄這個塊是否經過了 DDT 。讀取數據不需要經過 DDT ，但是子卷、克隆或者文件系統正常刪除數據塊的時候， 需要根據塊指針中的標誌位判斷是否需要

檢查和調整 DDT 。

從而關於 dedup 的實現可以得知以下一些特點：

- 開啓 dedup 之後，每個寫入操作放大成 3+1 個隨機位置的寫入操作，每個刪除操作變成 1 個寫入操作。沒有 dedup 時刪除塊並不需要立刻寫入，只需要記錄在內存中並在 MOS 提交的時候調整磁盤佔用情況即可。
- 只有開啓 dedup 期間寫入的數據塊纔會參與 dedup 。對已經有數據的存儲池，後來開啓的 dedup 不會影響已經寫好的數據，從而即使後來新的寫入與之前的寫入有重複也得不到 dedup 效果。 DDT 中沒有記錄的數據塊不會參與 dedup 。換句話說 DDT 中那些引用計數爲 1 的記錄也是必須存在的，否則這些數據塊沒有機會參與 dedup 。
- 關閉 dedup 之後，只要 DDT 中還存有數據，那麼對這些數據的刪除操作仍然有性能影響。

從直覺上可以這樣理解：在 ZFS 中每個數據塊都有其「歸屬」，沒有 dedup 的時候，數據塊歸屬於某個數據集（文件系統、快照、克隆）， 該數據集需要負責釋放該數據塊或者把從屬信息轉移到別的數據集（快照）

上。 而在開啓 dedup 期間，產生的寫入的數據塊實際歸屬於 DDT 而不是任何一個數據集，數據集需要查詢和調整 DDT 中記錄的引用計數來決定是否能釋放數據塊。

乍看起來 DDT 貌似挺像 btrfs 的 EXTENT_TREE，但是本質上 EXTENT_TREE 是根據區塊地址排序的，而 DDT 因爲是個 hashtable 所以是根據 checksum 排序的。並且 EXTENT_TREE 中記錄的區塊可以是任意大小，而 DDT 中記錄的數據塊是固定大小的，所以碎片不嚴重的情況下 DDT 要比 EXTENT_TREE 多記錄很多數據塊。這些區別都非常影響操作 DDT 時的性能。

DDT 本身是個 DMU 對象，所以對 DDT 的讀寫也是經過 DMU 的 CoW 讀寫，從而也經過 ARC 的緩存。想要有比較合理的 dedup 性能，需要整個 DDT 都儘量保持在內存 ARC 或者 L2ARC 緩存中，於是 dedup 特性也有了非常佔用內存的特點。每個 DDT 表項需要大概 192 字節來描述一個（默認 128KiB 大小的）數據塊，由此可以估算一下平均每 2TiB 的數據需要 3GiB 的內存來支持 dedup 的功能。

Matt 的視頻中後面講到優化 ZFS dedup 的一些思路，大體上未來 ZFS 可以做這些優化：

1. DDT 在內存中仍然是 hashtable，在存儲介質上則換成類似 ZIL 的日誌結構，讓 DDT 儘量保持在內存中，並且繞過 DMU 減少寫入放大。
2. 給 DDT 表項瘦身，從192字節縮減到接近64字節。
3. 當遇到內存壓力時，從 DDT 中隨機剔除掉引用計數爲 1 的表項。被剔除的表項沒有了未來參與 dedup 的可能性，但是能減輕內存壓力。剔除引用計數爲 1 的表項仍然可以維持數據塊的歸屬信息（處理上當作是沒有 dedup 的形式），但是引用

計數更高的表項沒法剔除。

這些優化策略目的是想讓 dedup 的性能損失能讓更多使用場景接受。不過因爲缺乏開發者意願， 目前這些策略還只是計劃，沒有實現在 ZFS 的代碼中。

因爲以上特點， ZFS 目前 dedup 特性的適用場景極爲有限，只有在 IO 帶寬、內存大小都非常充裕， 並且可以預見到很多重複的數據的時候適合。聽說過的 ZFS dedup 的成功案例是，比如提供虛擬機服務的服務商，在宿主文件系統上用 ZFS 的 zvol 寄宿虛擬機的磁盤鏡像，客戶在虛擬機內使用其它文件系統。大部分客戶可能用類似版本的操作系統， 從而宿主機整體來看有很多 dedup 的潛質。不過這種應用場景下，服務商很可能偏向選擇 CephFS 這樣的分佈式文件系統提供虛擬機鏡像存儲，而不是 ZFS 這樣侷限在單系統上的本地文件系統。

4.2 btrfs 的 dedup

btrfs 目前沒有內建的 dedup 支持，但是因爲有 reflink 所以可以通過第三方工具在事後掃描文件塊來實現 dedup 。這一點乍看像是某種將就之策，實際上瞭解了 ZFS dedup 的實現之後可以看出這個狀況其實更靈活。

在 btrfs 中實現 in-band dedup 本身不算很複雜，

增加一個內存中的 bloom filter 然後按情況插入 reflink 的正常思路就夠了。在 btrfs kernel wiki 中有篇筆記 提到已經有了實驗性的 in-band dedup 內核支持的實現。這個實現已經越來越成熟，雖然還有諸多使用限制， 不過實現正確性上問題不大，遲遲沒有辦法合併進主線內核的原因更多是性能上的問題。

如果 btrfs 有了 in-band dedup 這樣系統性的 dedup 方案，那麼不可避免地會增加文件系統中使用 reflink 的數量。這將會暴露出 backref walking 這樣的基礎設施中許多潛在的邊角情況下的性能瓶頸。 前面解釋過 backref walking 操作是個挺大開銷的操作，並且開銷隨着快照和 reflink 的使用而爆炸式增長。直到最近的 btrfs 更新仍然在試圖優化和改善現有 backref walking 的性能問題，可以預測 btrfs 的內建 dedup 支持將需要等待這方面更加成熟。

5 結論和展望

不知不覺圍繞 btrfs 和 zfs 的快照功能寫了一大篇，前前後後寫了一個半月， 文中提及的很多細節我自己也沒有自信，如果有錯誤還請指出。

稍微列舉一些我覺得比較重要的結論，算是 TL;DR 的 takeaway notes 吧：

- ZFS 的快照非常輕量。完全可以像 NILFS2 的連續快照那樣，每小時一個快照，每天24小時，每年

快照那樣，每小時一個快照，每天24小時，每年365天不間斷地創建快照，實際似乎也有公司是這樣用的。如此頻繁的快照不同於 NILFS2 等文件系統提供的連續快照，但是也沒有那些日誌結構文件系統實現連續快照所需承擔的巨大 GC 開銷。 並且 ZFS 可以沒有額外開銷地算出快照等數據集的空間佔用之類的信息。

- btrfs 的快照相對也很輕量，比 LVM 和 dm-thin 的快照輕便很多，但是不如 ZFS 的快照輕，因爲 btrfs 有維護反向引用的開銷。 btrfs 要得知子卷的空間佔用情況需要開啓 qgroup 特性，這會對一些需要 backref walking 的操作有一些額外性能損失。
- btrfs 對快照和 reflink 沒有限制，日常桌面系統下使用也不太會遇到性能問題。 不過系統性地（自動化地）大量使用快照和 reflink ，在一些操作下可能會有性能問題，值得注意。
- 因爲沒有 reflink ， ZFS 的數據集劃分需要一些前期計劃。 ZFS 中共享元數據的方式只有快照， 所以要儘量多細分文件系統，方便以後能利用到快照特性，劃分的粒度大致按照可能要回滾快照的粒度來。 btrfs 有 reflink ，於是這裏有很多自由度，即便前期計劃不夠詳細也可以通過 reflink 相對快速調整子卷結構。
- dedup 在 zfs 和 btrfs 都是個喜憂參半的特性，開啓前要仔細評估可能的性能損失。ZFS dedup 的成功案例是，比如虛擬機服務的服務商，在宿主文件系統上用 ZFS 寄宿虛擬機的磁盤鏡像，客戶在虛擬機可能用類似版本的操作系統，從而宿主機整

體來看有很多 dedup 的潛質。一般桌面場景下 dedup 的收益不明顯，反而有巨大內存和IO帶寬開銷。

- 相比 btrfs，ZFS 更嚴格地遵守 CoW 文件系統「僅寫一次」的特點，甚至就算遇到了數據塊損壞， 修復數據塊的時候也只能在原位寫入。 btrfs 因爲有反向引用所以在這方面靈活很多。
- ZFS 不支持對單個文件關閉 CoW，所有文件（以及所有 zvol）都經過 DMU 層有 CoW 語義，這對一些應用場景有性能影響。btrfs 可以對單個文件關閉 CoW，但是關閉 CoW 同時也丟失了寫文件的事務性語義。
- ZFS 不支持碎片整理，靠 ARC 加大緩存來解決碎片帶來的性能問題。 btrfs 有 defrag，但是目前的實現會切斷 reflink。

最後關於 ZFS 沒有 reflink 也沒有反向引用的情況，想引用幾段話。

FreeBSD 的發起人之一，FreeBSD 的 FFS 維護者，Kirk McKusick 曾經在 OpenZFS developer summit 2015 這麼說過：

> I decided I'd add a wish list since I have a whole bunch of people here that could actually possibly consider doing this. Both competitors of ZFS, which are

basically WAFL and BTRFS, kind of maintained back pointers. And back pointers allow a lot of things like disk migration, you can go through and tune up file layout, if you're working with direct-mapped flash it allows you to do that effectively. This has been a long -- and I understand big debate with the ZFS people and I'm not going to try and talk about that -- but there's a very nice paper that I've cited here, "Tracking Back References in a Write Anywhere File System", that is it integrates keeping track of the back pointers in a way that would work very well with ZFS. And so the cost is low, the cost of actually using it is a little higher, but it's not unreasonable. So there's the reference to that paper and if any of you are contemplating that you should read the paper because if

> nothing else it's a great paper.

Kirk McKusick 呼籲 ZFS 開發者們考慮在 ZFS 中實現類似 backref 的基礎設施，從而可能在未來獲得更多有用的特性。

和 ZFS 實現 backref 相關的一點是目前 ZFS 的塊指針的組織結構。對此 ZFS 的 ZPL 層原作者之一的 Mark Shellenbaum 在 OpenZFS developer summit 2016 也曾說過這樣的話：

> (Q: Are there any things that we that we have regretted we did?) A: I guess not so much on the ZPL, but with the way block pointers maybe weren't fully virtualized, you know that things like that.

以及 ZFS 的最初的作者 Jeff 在 OpenZFS developer summit 2015 也曾說過：

> ... and then certainly one thing i'd always wish we had done but there really were always implementation

> difficulties was true virtual block addressing. Because it would made dedup simpler, or would have made you know compression of data, defragging, all that kind of stuff simpler. That would have been really nice to have. But we never did the way that was sort of tracable in terms of both the cost and the transactional semantics.

ZFS 這些開發者元老們都希望 ZFS 能有某種類似 backref 的機制，或者讓塊指針記錄的地址更抽象的機制。

關於這一點，ZFS 最重要的作者 Matt 如何看的呢？Matt 近期似乎沒有發表過看法，但是熟悉 ZFS 的人可能聽到過 Matt 一直在計劃的另一項 ZFS 特性中看出些端倪，叫 BP rewrite ，或者 BP virtualization 。從 Matt 還在 Sun 的時候開始，就試圖在 ZFS 中實現 BP rewrite 特性，提供某種系統性的基礎設施，能夠快速地找到並

改寫大量數據塊指針。 在網上搜索很多 ZFS 功能的實現細節，最終都會帶到關於 BP rewrite 的討論（甚至可以說論戰）中。 Matt 最近給 OpenZFS 實現的兩項功能，toplevel vdev removal 和 raidz expansion 如果有 BP

rewrite 將會容易很多，而他們目前是在沒有 BP rewrite 的前提下，通過一連串額外抽象實現的。

從 BP rewrite 這個兔子洞中，還能引出更多 btrfs 和 ZFS 關於設備管理的差異，這個有待今後再談。