

ZFS 分層架構設計



Table of Contents

Contents

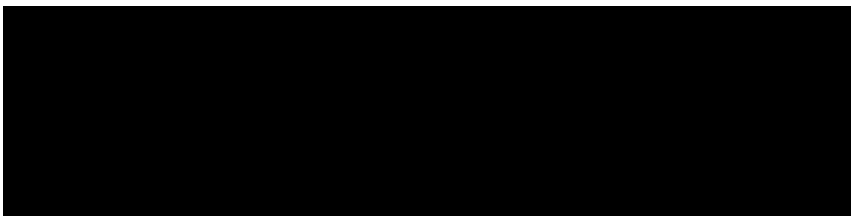
- 早期架構.....
- 子系統整體架構.....
- SPA.....
- VDEV.....
- ZIO.....
- ARC.....

- L2ARC.....
- TOL.....
- DMU.....
- ZAP.....
- DSL.....
- ZIL.....
- ZVOL.....
- ZPL.....

ZFS 在設計之初源自於 Sun 內部多次重寫 UFS 的嘗試，背負了重構 Solaris 諸多內核子系統的重任，從而不同於 Linux 的文件系統只負責文件系統的功能而把其餘功能（比如內存髒頁管理，IO調度）交給內核更底層的子系統，ZFS 的整體設計更層次化並更獨立，很多部分可能和 Linux 內核已有的子系統有功能重疊。

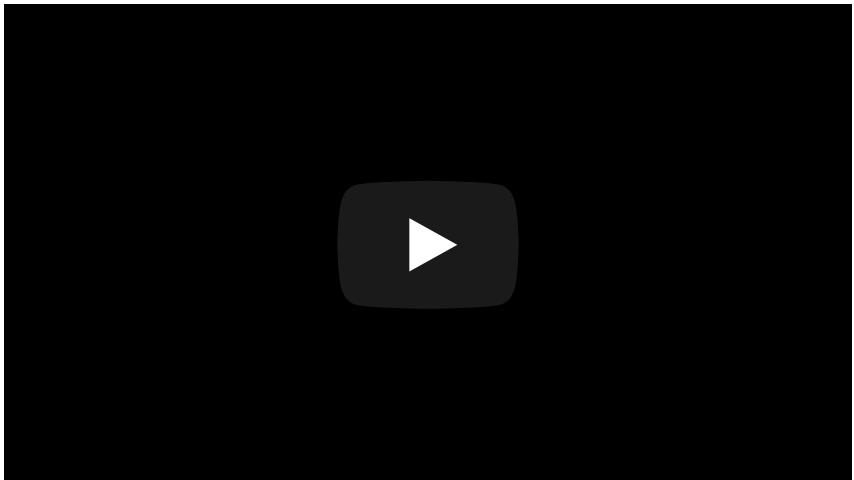
似乎很多關於 ZFS 的視頻演講和幻燈片有講到類似的子系統架構，但是找了半天也沒找到網上關於這個的說明文檔。於是寫下這篇筆記試圖從 ZFS 的早期開發歷程開始，記錄一下 ZFS 分層架構中各個子系統之間的分工。也有幾段 OpenZFS Summit 視頻佐以記錄那段歷史。

The Birth of ZFS by Jeff Bonwick

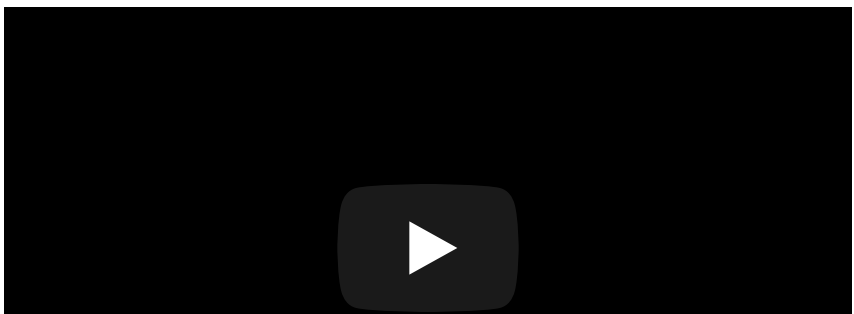


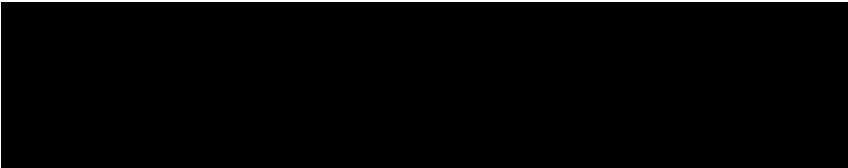


Story Time (Q&A) with Matt and Jeff



ZFS First Mount by Mark Shellenbaum





ZFS past & future by Mark Maybee



早期架構

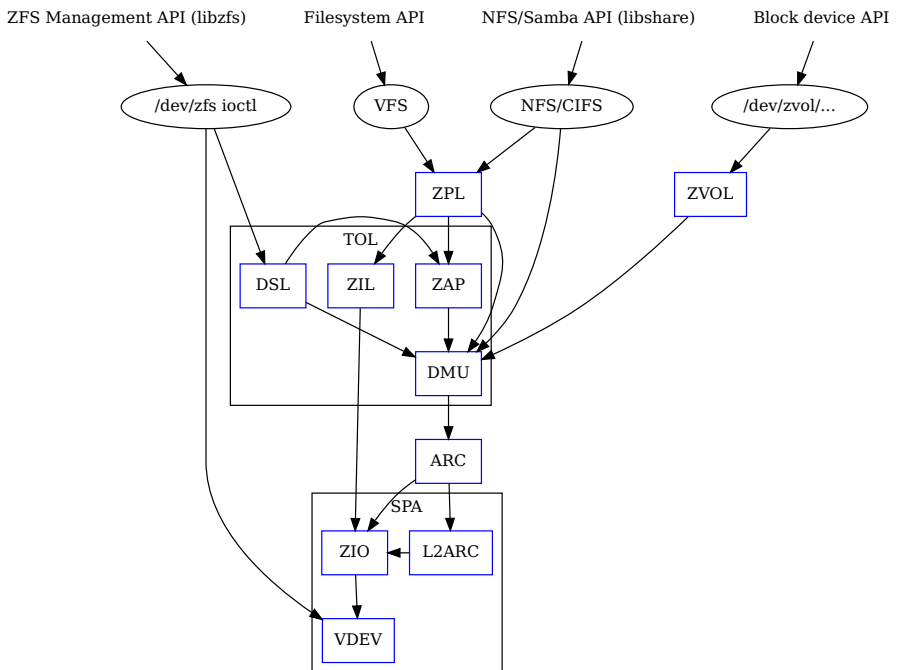
早期 ZFS 在開發時大體可以分爲上下三層，分別是 ZPL，DMU 和 SPA，這三層分別由三組人負責。

最初在 Sun 內部帶領 ZFS 開發的是 Jeff Bonwick，他首先有了對 ZFS 整體架構的構思，然後遊說 Sun 高層，親自組建起了 ZFS 開發團隊，招募了當時剛從大學畢業的 Matt Ahrens。作為和 Sun 高層談妥的條件，Jeff 也必須負責 Solaris 整體的 Storage & Filesystem Team，於是他也從 Solaris 的 Storage Team 抽調了 UFS 部分的負責人 Mark Shellenbaum 和 Mark Maybee 來開發 ZFS。而如今 Jeff 成立了獨立公司繼續開拓服務器存儲領域，Matt 是 OpenZFS 項目的負責人，兩位 Mark 則留在了 Sun/Oracle 成為了 Oracle ZFS 分支的維護者。

在開發早期，作為分工，Jeff 負責 ZFS 設計中最底層的 SPA，提供多個存儲設備組成的存儲池抽象；Matt 負責 ZFS 設計中最至關重要的 DMU 引擎，在塊設備基礎上提供具有事務語義的對象存儲；而兩位 Mark 負責 ZFS 設計中直接面向用戶的 ZPL，在 DMU 基礎上提供完整 POSIX 文件系統語義。

子系統整體架構

首先 ZFS 整體架構如下圖，其中圓圈是 ZFS 給內核層的外部接口，方框是 ZFS 內部子系統：



接下來從底層往上介紹一下各個子系統的全稱和職能。

SPA

Storage Pool Allocator

從內核提供的多個塊設備中抽象出存儲池的子系統。SPA 進一步分爲 ZIO 和 VDEV 兩大部分。

SPA 對 DMU 提供的接口不同於傳統的塊設備接口，完全利用了 CoW FS 對寫入位置不敏感的特點。傳統的塊設備接口通常是寫入時指定一個寫入地址，把緩衝區寫到磁盤指定的位置上，而 DMU 可以讓 SPA 做兩種操作：

1. write ， DMU 交給 SPA 一個數據塊的內存指針， SPA 負責找設備寫入這個數據塊，然後返回給 DMU 一個 block pointer 。
2. read ， DMU 交給 SPA 一個 block pointer ，

SPA 讀取設備並返回給 DMU 完整的數據塊。

也就是說，在 DMU 讓 SPA 寫數據塊時，DMU 還不知道 SPA 會寫入的地方，這完全由 SPA 判斷，這一點通常被稱為 Write Anywhere。反過來 SPA 想要對一個數據塊操作時，也完全不清楚這個數據塊用於什麼目的，屬於什麼文件或者文件系統結構。

VDEV

Virtual DEvice

作用相當於 Linux 內核的 Device Mapper 層或者 FreeBSD GEOM 層，提供 Stripe/Mirror/RAIDZ 之類得多設備存儲池管理和抽象。ZFS 中的 vdev 形成一個樹狀結構，在樹的底層是從內核提供的物理設備，其上是虛擬的塊設備。每個虛擬塊設備對上對下都是塊設備接口，除了底層的物理設備之外，位於中間層的 vdev 需要負責地址映射、容量轉換等計算過程。

ZIO

ZFS I/O

作用相當於內核的 IO scheduler 和 pagecache write back 機制。ZIO 內部使用流水線和事件驅動機制，避免讓上層的 ZFS 線程阻塞等待在 IO 操作上。ZIO 把一個上層的寫請求轉換成多個寫操作，負責把這些寫操作合併到 transaction group 提交事務組。ZIO 也負責將讀寫請求按同步還是異步分成不同的讀寫優先級並實施優先級調度，在 [OpenZFS 項目 wiki 頁](#) 有一篇描述 ZIO 調度的細節。

除了調度之外，ZIO 層還負責在讀寫流水線中拆解和拼裝數據塊。上層 DMU 交給 SPA 的數據塊有固定大小，目前默認是 128KiB，pool 整體的參數可以調整塊大小在 8KiB 到 8MiB 之間。ZIO 拿到整塊大小的數據塊之後，在流水線中可以對數據塊做如下操作：

1. 用壓縮算法，壓縮/解壓數據塊。
2. 查詢 dedup table，對數據塊去重。
3. 如果底層分配器不能分配完整的 128KiB（或別的大小），那麼嘗試分配多個小塊，多個用 512B 的指針間接塊連起多個小塊的 [gang block](#) 拼成一個大塊。

可見經過 ZIO 流水線之後，數據塊不再是統一大小，這使得 ZFS 用在 4K 對齊的磁盤或者 SSD 上有了一些新的挑戰。

Adaptive Replacement Cache

作用相當於 Linux/Solaris/FreeBSD 中傳統的 page/buffer cache。和傳統 pagecache 使用 LRU (Least Recently Used) 之類的算法剔除緩存頁不同，ARC 算法試圖在 LRU 和 LFU(Least Frequently Used) 之間尋找平衡，從而複製大文件之類的線性大量 IO 操作不至於讓緩存失效率猛增。

不過 ZFS 採用它自有的 ARC 一個顯著缺點在於，不能和內核已有的 pagecache 機制相互配合，尤其在系統內存壓力很大的情況下，內核與 ZFS 無關的其餘部分可能難以通知 ARC 釋放內存。所以 ARC 是 ZFS 消耗內存的大戶之一（另一個是可選的 dedup table），也是 ZFS 性能調優的重中之重。

當然，ZFS 採用 ARC 不依賴於內核已有的 pagecache 機制除了 LFU 平衡之外，也有其有利的一面。系統中多次讀取因 snapshot 或者 dedup 而共享的數據塊的話，在 ZFS 的 ARC 機制下，同樣的 block pointer 只會被緩存一次；而傳統的 pagecache 因為基於 inode 判斷是否有共享，所以即使這些文件有共享頁面（比如 btrfs/xfs 的 reflink 形成的），也會多次讀入內存。Linux 的 btrfs 和 xfs 在 VFS 層面有共用的 reflink 機制之後，正在努力着手改善這種局面，而 ZFS 因為 ARC 所以從最初就避免了這種浪費。

和很多傳言所說的不同，ARC 的內存壓力問題不僅在 Linux 內核會有，在 FreeBSD 和 Solaris/Illumos 上也是同樣，這個在 ZFS First Mount by Mark

Shellenbaum 的問答環節 16:37 左右有提到。其中 Mark Shellenbaum 提到 Matt 覺得讓 ARC 併入現有 pagecache 子系統的工作量太大，難以實現。

L2ARC

Level 2 Adaptive Replacement Cache

這是用 ARC 算法實現的二級緩存，保存於高速存儲設備上。常見用法是給 ZFS pool 配置一塊 SSD 作為 L2ARC 高速緩存，減輕內存 ARC 的負擔並增加緩存命中率。

TOL

Transactional Object Layer

這一部分子系統在數據塊的基礎上提供一個事務性的對象語義層，這裏事務性是指，對對象的修改處於明確的狀態，不會因為突然斷電之類的原因導致狀態不一致。TOL 中最主要的部分是 DMU 層。

DMU

Data Management Unit

在塊的基礎上提供「對象」的抽象。每個「對象」可以是一個文件，或者是別的 ZFS 內部需要記錄的東西。

DMU 這個名字最初是 Jeff 想類比於操作系統中內存管理的 MMU(Memory Management Unit)，Jeff 希望 ZFS 中增加和刪除文件就像內存分配一樣簡單，增加和移除塊設備就像增加內存一樣簡單，由 DMU 負責從存儲池中分配和釋放數據塊，對上提供事務性語義，管理員不需要管理文件存儲在什麼存儲設備上。這裏事務性語義指對文件的修改要麼完全成功，要麼完全失敗，不會處於中間狀態，這靠 DMU 的 CoW 語義實現。

DMU 實現了對象級別的 CoW 語義，從而任何經過了 DMU 做讀寫的子系統都具有了 CoW 的特徵，這不僅包括文件、文件夾這些 ZPL 層需要的東西，也包括文件系統內部用的 spacemap 之類的設施。相反，不經過 DMU 的子系統則可能沒法保證事務語義。這裏一個特例是 ZIL，一定程度上繞過了 DMU 直接寫日誌。說一定程度是因為 ZIL 仍然靠 DMU 來擴展長度，當一個塊寫滿日誌之後需要等 DMU 分配一個新塊，在分配好的塊內寫日誌則不需要經過 DMU。

上面提到 SPA 的時候也講了 DMU 和 SPA 之間不同於普通塊設備抽象的接口，這使得 DMU 按整塊的大小分配空間。當對象的大小超過一個固定的塊大小時

（4K~8M，默認128K），DMU 採用了傳統 Unix 文件系統的間接塊（indirect block）的方案，不同於更現代的文件系統如 ext4/xfs/btrfs/ntfs/hfs+ 這些使用 extent 記錄連續的物理地址分配。間接塊簡單來說就是寫滿了 block pointer 的塊組成的樹狀結構。DMU 採用間接塊而不是 extent，使得 ZFS 的空間分配更趨向碎片化。

上面也提到因為 SPA 和 DMU 分離，SPA 完全不知道數據塊用於什麼目的；這一點其實對 DMU 也是類似，DMU 雖然能從某個對象找到它所佔用的數據塊，但是 DMU 完全不知道這個對象在文件系統或者存儲池中是用來存儲什麼的。當 DMU 讀取數據遇到壞塊（block pointer 中的校驗和與 block pointer 指向的數據塊內容不一致）時，它知道這個數據塊在哪兒（具體哪個設備上的哪個地址），但是不知道這個數據塊是否和別的對象共享，不知道搬動這個數據塊的影響，也沒法從對象反推出文件系統路徑，（除了明顯開銷很高地掃一遍整個存儲池）。所以 DMU 在遇到讀取錯誤（普通的讀操作或者 scrub/resilver 操作中）時，只能選擇在同樣的地址，原地寫入數據塊的備份（如果能找到或者推算出備份的話）。

或許有人會疑惑，既然從 SPA 無法根據數據地址反推出對象，在 DMU 也無法根據對象反推出文件，那麼 zfs 在遇到數據損壞時是如何給出損壞的文件路徑的呢？這其實基於 ZPL 的一個黑魔法：在 dnode 記錄父級

dnode 的編號。因為是個黑魔法，這個記錄不總是對的，所以只能用於診斷信息，不能基於這個實現別的文件系統功能。

ZAP

ZFS Attribute Processor

在 DMU 提供的「對象」抽象基礎上提供緊湊的 name/value 映射存儲，從而文件夾內容列表、文件擴展屬性之類的都是基於 ZAP 來存。ZAP 在內部分為兩種存儲表達：microZAP 和 fatZAP。

一個 microZAP 佔用一整塊數據塊，能存 name 長度小於 50 字符並且 value 是 uint64_t 的表項，每個表項 64 字節。fatZAP 則是個樹狀結構，能存更多更複雜的東西。可見 microZAP 非常適合表述一個普通大小的文件夾裏面包含到很多普通文件 inode（ZFS 是 dnode）的引用。

在 ZFS First Mount by Mark Shellenbaum 中提到，最初 ZPL 中關於文件的所有屬性（包括訪問時間、權限、大小之類所有文件都有的）都是基於 ZAP 來存，然後文件夾內容列表有另一種數據結構 ZDS，後來常見的文件屬性在 ZPL 有了專用的緊湊數據結構，而 ZDS 則漸漸融入了 ZAP。

DSL

Dataset and Snapshot Layer

數據集和快照層，負責創建和管理快照、克隆等數據集類型，跟蹤它們的寫入大小，最終刪除它們。由於 DMU 層面已經負責了對象的寫時複製語義，所以 DSL 層面不需要直接接觸寫文件之類來自 ZPL 的請求，無論有沒有快照對 DMU 層面一樣採用寫時複製的方式修改文件數據。不過在刪除快照和克隆之類的時候，則需要 DSL 參與計算沒有和別的數據集共享的數據塊並且刪除它們。

除了管理數據集，DSL 層面也提供了 zfs 中 send/receive 的能力。ZFS 在 send 時從 DSL 層找到快照引用到的所有數據塊，把它們直接發往管道，在 receive 端則直接接收數據塊並重組數據塊指針。因為 DSL 提供的 send/receive 工作在 DMU 之上，所以在 DSL 看到的數據塊是 DMU 的數據塊，下層 SPA 完成的數據壓縮、加密、去重等工作，對 DMU 層完全透明。所以在最初的 send/receive 實現中，假如數據塊已經壓縮，需要在 send 端經過 SPA 解壓，再 receive 端則重新壓縮。最近 ZFS 的 send/receive 逐漸打破 DMU 與 SPA 的壁壘，支持了直接發送已壓縮或加密的數據塊的能力。

ZFS Intent Log

記錄兩次完整事務語義提交之間的日誌，用來加速實現 fsync 之類的文件事務語義。

原本 CoW 的文件系統不需要日誌結構來保證文件系統結構的一致性，在 DMU 保證了對象級別事務語義的前提下，每次完整的 transaction group commit 都保證了文件系統一致性，掛載時也直接找到最後一個 transaction group 從它開始掛載即可。不過在 ZFS 中，做一次完整的 transaction group commit 是個比較耗時的操作，在寫入文件的數據塊之後，還需要更新整個 object set，然後更新 meta-object set，最後更新 uberblock，爲了滿足事務語義這些操作沒法並行完成，所以整個 pool 提交一次需要等待好幾次磁盤寫操作返回，短則一兩秒，長則幾分鐘，如果事務中有要刪除快照等非常耗時的操作可能還要等更久，在此期間提交的事務沒法保證一致。

對上層應用程序而言，通常使用 fsync 或者 fdatasync 之類的系統調用，確保文件內容本身的事務一致性。如果要讓每次 fsync/fdatasync 等待整個 transaction group commit 完成，那會嚴重拖慢很多應用程序，而如果它們不等待直接返回，則在突發斷電時沒有保證一致性。從而 ZFS 有了 ZIL，記錄兩次 transaction group 的 commit 之間發生的 fsync，突然

斷電後下次 import zpool 時首先找到最近一次 transaction group，在它基礎上重放 ZIL 中記錄的寫請求和 fsync 請求，從而滿足 fsync API 要求的事務語義。

顯然對 ZIL 的寫操作需要繞過 DMU 直接寫入數據塊，所以 ZIL 本身是以日誌系統的方式組織的，每次寫 ZIL 都是在已經分配的 ZIL 塊的末尾添加數據，分配新的 ZIL 塊仍然需要經過 DMU 的空間分配。

傳統日誌型文件系統中對 data 開日誌需要寫兩次，一次寫入日誌，再一次覆蓋文件系統內容；在 ZIL 實現中則不需要寫兩次，DMU 讓 SPA 寫入數據之後 ZIL 可以直接記錄新數據塊的 block pointer，所以使用 ZIL 不會導致傳統日誌型文件系統中雙倍寫入放大的問題。

ZVOL

ZFS VOLUME

有點像 loopback block device，暴露一個塊設備的接口，其上可以創建別的 FS。對 ZFS 而言實現 ZVOL 的意義在於它是比文件更簡單的接口，所以在實現完整 ZPL 之前，一開始就先實現了 ZVOL，而且早期 Solaris 沒有 thin provisioning storage pool 的時候可以用 ZVOL 模擬很大的塊設備，當時 Solaris 的 UFS 團隊用它來測試 UFS 對 TB 級存儲的支持情況。

因爲 ZVOL 基於 DMU 上層，所以 DMU 所有的文件系統功能，比如 snapshot / dedup / compression 都可以用在 ZVOL 上，從而讓 ZVOL 上層的傳統文件系統也具有類似的功能。並且 ZVOL 也具有了 ARC 緩存的能力，和 dedup 結合之下，非常適合於在一個宿主機 ZFS 上提供對虛擬機文件系統鏡像的存儲，可以節省不少存儲空間和內存佔用開銷。

ZPL

ZFS Posix Layer

提供符合 POSIX 文件系統的語義，也就是包括文件、目錄這些抽象以及 inode 屬性、權限那些，對一個普通 FS 而言用戶直接接觸的部分。ZPL 可以說是 ZFS 最複雜的子系統，也是 ZFS 作爲一個文件系統而言最關鍵的部分。

ZPL 的實現中直接使用了 ZAP 和 DMU 提供的抽象，比如每個 ZPL 文件用一個 DMU 對象表達，每個 ZPL 目錄用一個 ZAP 對象表達。

