

# ZFS 分层架构设计



## 目次

### Contents

- 早期架构.....
- 子系统整体架构.....
- SPA.....
- VDEV.....
- ZIO.....
- ARC.....

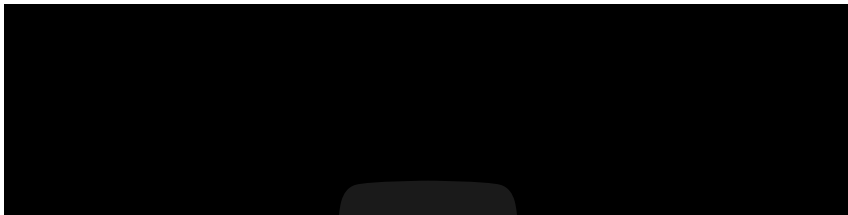
- L2ARC.....
- TOL.....
- DMU.....
- ZAP.....
- DSL.....
- ZIL.....
- ZVOL.....
- ZPL.....

ZFS 在设计之初源自于 Sun 内部多次重写 UFS 的尝试，背负了重构 Solaris 诸多内核子系统的重任，从而不同于 Linux 的文件系统只负责文件系统的功能而把其余功能（比如内存脏页管理，IO调度）交给内核更底层的子系统，ZFS 的整体设计更层次化并更独立，很多部分可能和 Linux 内核已有的子系统有功能重叠。

似乎很多关于 ZFS 的视频演讲和幻灯片有讲到类似的子系统架构，但是找了半天也没找到网上关于这个的说明文档。于是写下这篇笔记试图从 ZFS 的早期开发历程开始，记录一下 ZFS 分层架构中各个子系统之间的分工。也有几段 OpenZFS Summit 视频佐以记录那段历史。

The Birth of ZFS by Jeff Bonwick

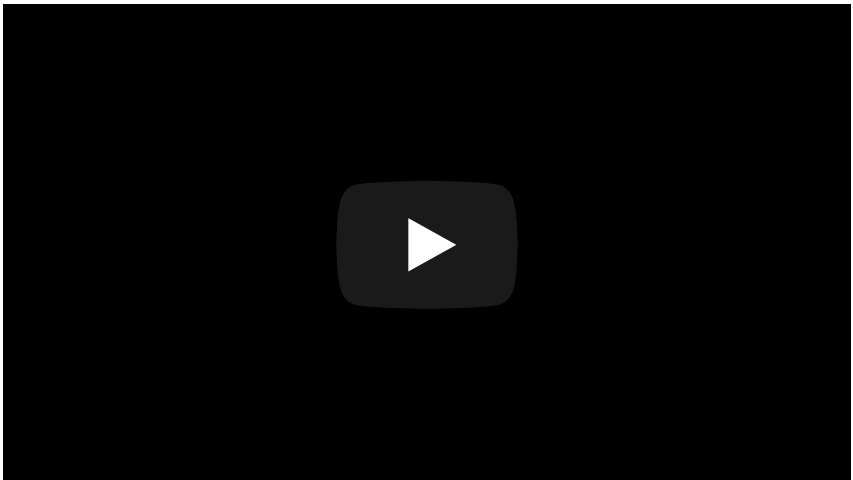
---





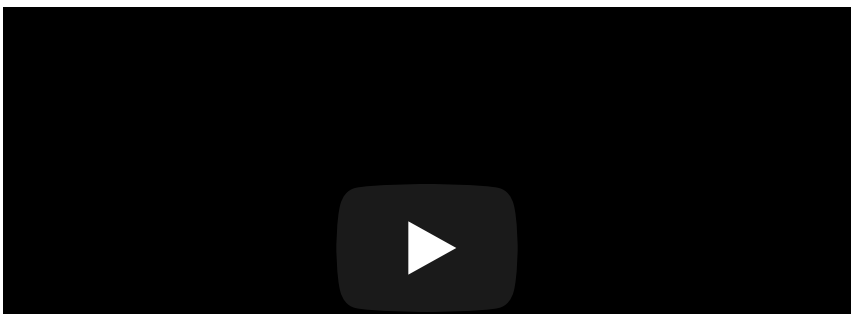
## Story Time (Q&A) with Matt and Jeff

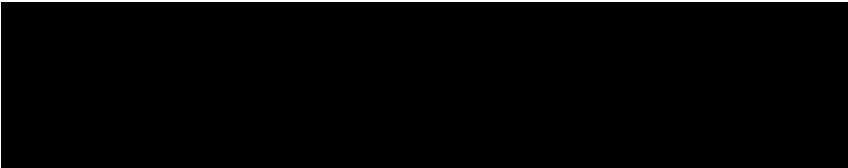
---



## ZFS First Mount by Mark Shellenbaum

---





ZFS past & future by Mark Maybee

---



## 早期架构

---

早期 ZFS 在开发时大体可以分为上下三层，分别是 ZPL，DMU 和 SPA，这三层分别由三组人负责。

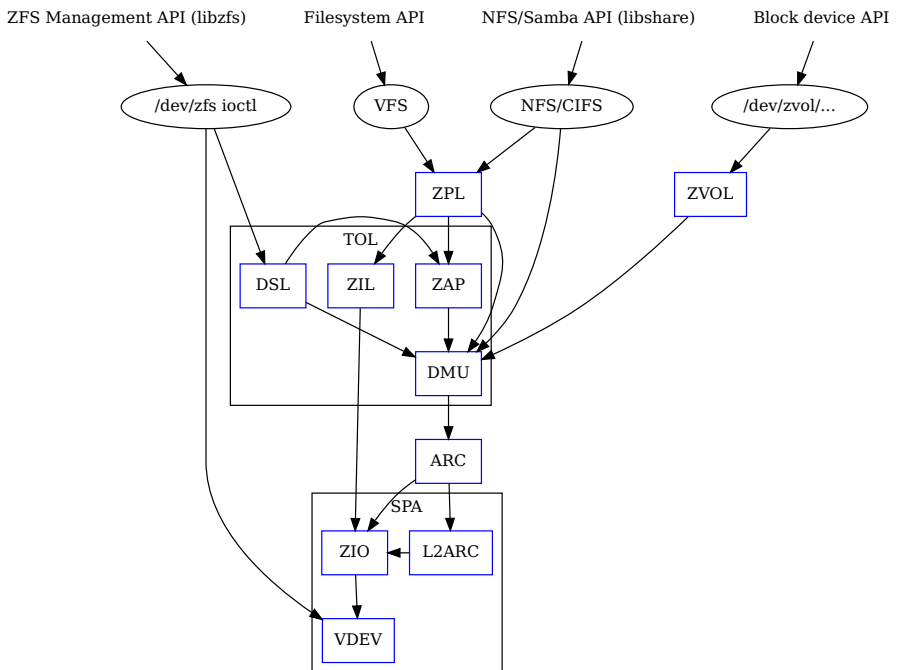
最初在 Sun 内部带领 ZFS 开发的是 Jeff Bonwick，他首先有了对 ZFS 整体架构的构思，然后游说 Sun 高层，亲自组建起了 ZFS 开发团队，招募了当时刚从大学毕业的 Matt Ahrens。作为和 Sun 高层谈妥的条件，Jeff 也必须负责 Solaris 整体的 Storage & Filesystem Team，于是他也从 Solaris 的 Storage Team 抽调了 UFS 部分的负责人 Mark Shellenbaum 和 Mark Maybee 来开发 ZFS。而如今 Jeff 成立了独立公司继续开拓服务器存储领域，Matt 是 OpenZFS 项目的负责人，两位 Mark 则留在了 Sun/Oracle 成为了 Oracle ZFS 分支的维护者。

在开发早期，作为分工，Jeff 负责 ZFS 设计中最底层的 SPA，提供多个存储设备组成的存储池抽象；Matt 负责 ZFS 设计中最至关重要的 DMU 引擎，在块设备基础上提供具有事务语义的对象存储；而两位 Mark 负责 ZFS 设计中直接面向用户的 ZPL，在 DMU 基础上提供完整 POSIX 文件系统语义。

## 子系统整体架构

---

首先 ZFS 整体架构如下图，其中圆圈是 ZFS 给内核层的外部接口，方框是 ZFS 内部子系统：



接下来从底层往上介绍一下各个子系统的全称和职能。

# SPA

---

## Storage Pool Allocator

从内核提供的多个块设备中抽象出存储池的子系统。SPA 进一步分为 ZIO 和 VDEV 两大部分。

SPA 对 DMU 提供的接口不同于传统的块设备接口，完全利用了 CoW FS 对写入位置不敏感的特点。传统的块设备接口通常是写入时指定一个写入地址，把缓冲区写到磁盘指定的位置上，而 DMU 可以让 SPA 做两种操作：

1. write ， DMU 交给 SPA 一个数据块的内存指针， SPA 负责找设备写入这个数据块，然后返回给 DMU 一个 block pointer 。
2. read ， DMU 交给 SPA 一个 block pointer ，

SPA 读取设备并返回给 DMU 完整的数据块。

也就是说，在 DMU 让 SPA 写数据块时，DMU 还不知道 SPA 会写入的地方，这完全由 SPA 判断，这一点通常被称为 Write Anywhere。反过来 SPA 想要对一个数据块操作时，也完全不清楚这个数据块用于什么目的，属于什么文件或者文件系统结构。

# VDEV

---

## Virtual DEvice

作用相当于 Linux 内核的 Device Mapper 层或者 FreeBSD GEOM 层，提供 Stripe/Mirror/RAIDZ 之类的多设备存储池管理和抽象。ZFS 中的 vdev 形成一个树状结构，在树的底层是从内核提供的物理设备，其上是虚拟的块设备。每个虚拟块设备对上对下都是块设备接口，除了底层的物理设备之外，位于中间层的 vdev 需要负责地址映射、容量转换等计算过程。

# ZIO

---

## ZFS I/O



作用相当于内核的 IO scheduler 和 pagecache write back 机制。ZIO 内部使用流水线和事件驱动机制，避免让上层的 ZFS 线程阻塞等待在 IO 操作上。ZIO 把一个上层的写请求转换成多个写操作，负责把这些写操作合并到 transaction group 提交事务组。ZIO 也负责将读写请求按同步还是异步分成不同的读写优先级并实施优先级调度，在 [OpenZFS 项目 wiki 页](#) 有一篇描述 ZIO 调度的细节。

除了调度之外，ZIO 层还负责在读写流水线中拆解和拼装数据块。上层 DMU 交给 SPA 的数据块有固定大小，目前默认是 128KiB，pool 整体的参数可以调整块大小在 8KiB 到 8MiB 之间。ZIO 拿到整块大小的数据块之后，在流水线中可以对数据块做如下操作：

1. 用压缩算法，压缩/解压数据块。
2. 查询 dedup table，对数据块去重。
3. 如果底层分配器不能分配完整的 128KiB（或别的大小），那么尝试分配多个小块，多个用 512B 的指针间接块连起多个小块的 [gang block](#) 拼成一个大块。

可见经过 ZIO 流水线之后，数据块不再是统一大小，这使得 ZFS 用在 4K 对齐的磁盘或者 SSD 上有了一些新的挑战。

## Adaptive Replacement Cache

作用相当于 Linux/Solaris/FreeBSD 中传统的 page/buffer cache。和传统 pagecache 使用 LRU (Least Recently Used) 之类的算法剔除缓存页不同，ARC 算法试图在 LRU 和 LFU(Least Frequently Used) 之间寻找平衡，从而复制大文件之类的线性大量 IO 操作不至于让缓存失效率猛增。

不过 ZFS 采用它自有的 ARC 一个显著缺点在于，不能和内核已有的 pagecache 机制相互配合，尤其在系统内存压力很大的情况下，内核与 ZFS 无关的其余部分可能难以通知 ARC 释放内存。所以 ARC 是 ZFS 消耗内存的大户之一（另一个是可选的 dedup table），也是 ZFS 性能调优的重中之重。

当然，ZFS 采用 ARC 不依赖于内核已有的 pagecache 机制除了 LFU 平衡之外，也有其有利的一面。系统中多次读取因 snapshot 或者 dedup 而共享的数据块的话，在 ZFS 的 ARC 机制下，同样的 block pointer 只会被缓存一次；而传统的 pagecache 因为基于 inode 判断是否有共享，所以即使这些文件有共享页面（比如 btrfs/xfs 的 reflink 形成的），也会多次读入内存。Linux 的 btrfs 和 xfs 在 VFS 层面有共用的 reflink 机制之后，正在努力着手改善这种局面，而 ZFS 因为 ARC 所以从最初就避免了这种浪费。

和很多传言所说的不同，ARC 的内存压力问题不仅在 Linux 内核会有，在 FreeBSD 和 Solaris/Illumos 上也是同样，这个在 ZFS First Mount by Mark

Shellenbaum 的问答环节 16:37 左右有提到。其中 Mark Shellenbaum 提到 Matt 觉得让 ARC 并入现有 pagecache 子系统的工作量太大，难以实现。

## L2ARC

---

### Level 2 Adaptive Replacement Cache

这是用 ARC 算法实现的二级缓存，保存于高速存储设备上。常见用法是给 ZFS pool 配置一块 SSD 作为 L2ARC 高速缓存，减轻内存 ARC 的负担并增加缓存命中率。

## TOL

---

### Transactional Object Layer

这一部分子系统在数据块的基础上提供一个事务性的对象语义层，这里事务性是指，对对象的修改处于明确的状态，不会因为突然断电之类的原因导致状态不一致。TOL 中最主要的部分是 DMU 层。

# DMU

---

## Data Management Unit

在块的基础上提供「对象」的抽象。每个「对象」可以是一个文件，或者是别的 ZFS 内部需要记录的东西。

DMU 这个名字最初是 Jeff 想类比于操作系统中内存管理的 MMU(Memory Management Unit)，Jeff 希望 ZFS 中增加和删除文件就像内存分配一样简单，增加和移除块设备就像增加内存一样简单，由 DMU 负责从存储池中分配和释放数据块，对上提供事务性语义，管理员不需要管理文件存储在什么存储设备上。这里事务性语义指对文件的修改要么完全成功，要么完全失败，不会处于中间状态，这靠 DMU 的 CoW 语义实现。

DMU 实现了对象级别的 CoW 语义，从而任何经过了 DMU 做读写的子系统都具有了 CoW 的特征，这不仅包括文件、文件夹这些 ZPL 层需要的东西，也包括文件系统内部用的 spacemap 之类的设施。相反，不经过 DMU 的子系统则可能没法保证事务语义。这里一个特例是 ZIL，一定程度上绕过了 DMU 直接写日志。说一定程度是因为 ZIL 仍然靠 DMU 来扩展长度，当一个块写满日志之后需要等 DMU 分配一个新块，在分配好的块内写日志则不需要经过 DMU。

上面提到 SPA 的时候也讲了 DMU 和 SPA 之间不同于普通块设备抽象的接口，这使得 DMU 按整块的大小分配空间。当对象的大小超过一个固定的块大小时

（4K~8M，默认128K），DMU 采用了传统 Unix 文件系统的间接块（indirect block）的方案，不同于更现代的文件系统如 ext4/xfs/btrfs/ntfs/hfs+ 这些使用 extent 记录连续的物理地址分配。间接块简单来说就是写满了 block pointer 的块组成的树状结构。DMU 采用间接块而不是 extent，使得 ZFS 的空间分配更趋向碎片化。

上面也提到因为 SPA 和 DMU 分离，SPA 完全不知道数据块用于什么目的；这一点其实对 DMU 也是类似，DMU 虽然能从某个对象找到它所占用的数据块，但是 DMU 完全不知道这个对象在文件系统或者存储池中是用来存储什么的。当 DMU 读取数据遇到坏块（block pointer 中的校验和与 block pointer 指向的数据块内容不一致）时，它知道这个数据块在哪儿（具体哪个设备上的哪个地址），但是不知道这个数据块是否和别的对象共享，不知道搬动这个数据块的影响，也没法从对象反推出文件系统路径，（除了明显开销很高地扫一遍整个存储池）。所以 DMU 在遇到读取错误（普通的读操作或者 scrub/resilver 操作中）时，只能选择在同样的地址，原地写入数据块的备份（如果能找到或者推算出备份的话）。

或许有人会疑惑，既然从 SPA 无法根据数据地址反推出对象，在 DMU 也无法根据对象反推出文件，那么 zfs 在遇到数据损坏时是如何给出损坏的文件路径的呢？这其实基于 ZPL 的一个黑魔法：在 dnode 记录父级

inode 的编号。因为是个黑魔法，这个记录不总是对的，所以只能用于诊断信息，不能基于这个实现别的文件系统功能。

# ZAP

## ZFS Attribute Processor

在 DMU 提供的「对象」抽象基础上提供紧凑的 name/value 映射存储，从而文件夹内容列表、文件扩展属性之类的都是基于 ZAP 来存。ZAP 在内部分为两种存储表达：microZAP 和 fatZAP。

一个 microZAP 占用一整块数据块，能存 name 长度小于 50 字符并且 value 是 uint64\_t 的表项，每个表项 64 字节。fatZAP 则是个树状结构，能存更多更复杂的东西。可见 microZAP 非常适合表述一个普通大小的文件夹里面包含到很多普通文件 inode（ZFS 是 inode）的引用。

在 ZFS First Mount by Mark Shellenbaum 中提到，最初 ZPL 中关于文件的所有属性（包括访问时间、权限、大小之类所有文件都有的）都是基于 ZAP 来存，然后文件夹内容列表有另一种数据结构 ZDS，后来常见的文件属性在 ZPL 有了专用的紧凑数据结构，而 ZDS 则渐渐融入了 ZAP。

# DSL

---

## Dataset and Snapshot Layer

数据集和快照层，负责创建和管理快照、克隆等数据集类型，跟踪它们的写入大小，最终删除它们。由于 DMU 层面已经负责了对象的写时复制语义，所以 DSL 层面不需要直接接触写文件之类来自 ZPL 的请求，无论有没有快照对 DMU 层面一样采用写时复制的方式修改文件数据。不过在删除快照和克隆之类的时候，则需要 DSL 参与计算没有和别的数据集共享的数据块并且删除它们。

除了管理数据集，DSL 层面也提供了 zfs 中 send/receive 的能力。ZFS 在 send 时从 DSL 层找到快照引用到的所有数据块，把它们直接发往管道，在 receive 端则直接接收数据块并重组数据块指针。因为 DSL 提供的 send/receive 工作在 DMU 之上，所以在 DSL 看到的数据块是 DMU 的数据块，下层 SPA 完成的数据压缩、加密、去重等工作，对 DMU 层完全透明。所以在最初的 send/receive 实现中，假如数据块已经压缩，需要在 send 端经过 SPA 解压，再 receive 端则重新压缩。最近 ZFS 的 send/receive 逐渐打破 DMU 与 SPA 的壁垒，支持了直接发送已压缩或加密的数据块的能力。

## ZFS Intent Log

记录两次完整事务语义提交之间的日志，用来加速实现 fsync 之类的文件事务语义。

原本 CoW 的文件系统不需要日志结构来保证文件系统结构的一致性，在 DMU 保证了对象级别事务语义的前提下，每次完整的 transaction group commit 都保证了文件系统一致性，挂载时也直接找到最后一个 transaction group 从它开始挂载即可。不过在 ZFS 中，做一次完整的 transaction group commit 是个比较耗时的操作，在写入文件的数据块之后，还需要更新整个 object set，然后更新 meta-object set，最后更新 uberblock，为了满足事务语义这些操作没法并行完成，所以整个 pool 提交一次需要等待好几次磁盘写操作返回，短则一两秒，长则几分钟，如果事务中有要删除快照等非常耗时的操作可能还要等更久，在此期间提交的事务没法保证一致。

对上层应用程序而言，通常使用 fsync 或者 fdatasync 之类的系统调用，确保文件内容本身的事务一致性。如果能让每次 fsync/fdatasync 等待整个 transaction group commit 完成，那会严重拖慢很多应用程序，而如果它们不等待直接返回，则在突发断电时没有保证一致性。从而 ZFS 有了 ZIL，记录两次 transaction group 的 commit 之间发生的 fsync，突然



断电后下次 import zpool 时首先找到最近一次 transaction group，在它基础上重放 ZIL 中记录的写请求和 fsync 请求，从而满足 fsync API 要求的事务语义。

显然对 ZIL 的写操作需要绕过 DMU 直接写入数据块，所以 ZIL 本身是以日志系统的方式组织的，每次写 ZIL 都是在已经分配的 ZIL 块的末尾添加数据，分配新的 ZIL 块仍然需要经过 DMU 的空间分配。

传统日志型文件系统中对 data 开日志需要写两次，一次写入日志，再一次覆盖文件系统内容；在 ZIL 实现中则不需要写两次，DMU 让 SPA 写入数据之后 ZIL 可以直接记录新数据块的 block pointer，所以使用 ZIL 不会导致传统日志型文件系统中双倍写入放大的问题。

# ZVOL

---

## ZFS VOLume

有点像 loopback block device，暴露一个块设备的接口，其上可以创建别的 FS。对 ZFS 而言实现 ZVOL 的意义在于它是比文件更简单的接口，所以在实现完整 ZPL 之前，一开始就先实现了 ZVOL，而且早期 Solaris 没有 thin provisioning storage pool 的时候可以用 ZVOL 模拟很大的块设备，当时 Solaris 的 UFS 团队用它来测试 UFS 对 TB 级存储的支持情况。

---

因为 ZVOL 基于 DMU 上层，所以 DMU 所有的文件系统功能，比如 snapshot / dedup / compression 都可以用在 ZVOL 上，从而让 ZVOL 上层的传统文件系统也具有类似的功能。并且 ZVOL 也具有了 ARC 缓存的能力，和 dedup 结合之下，非常适合于在一个宿主机 ZFS 上提供对虚拟机文件系统镜像的存储，可以节省不少存储空间和内存占用开销。

# ZPL

---

## ZFS Posix Layer

提供符合 POSIX 文件系统的语义，也就是包括文件、目录这些抽象以及 inode 属性、权限那些，对一个普通 FS 而言用户直接接触的部分。ZPL 可以说是 ZFS 最复杂的子系统，也是 ZFS 作为一个文件系统而言最关键的部分。

ZPL 的实现中直接使用了 ZAP 和 DMU 提供的抽象，比如每个 ZPL 文件用一个 DMU 对象表达，每个 ZPL 目录用一个 ZAP 对象表达。

