ZFS 分层架构设计

目录

Contents

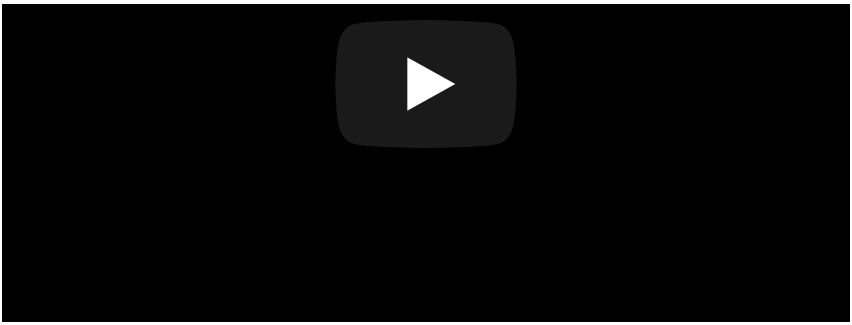
- 早期架构
- 子系统整体架构
- SPA
- VDEV
- ZIO
- MetaSlab

- ARC
- L2ARC
- SLOG
- TOL
- DMU
- ZAP
- DSL
- ZIL
- ZVOL
- ZPL

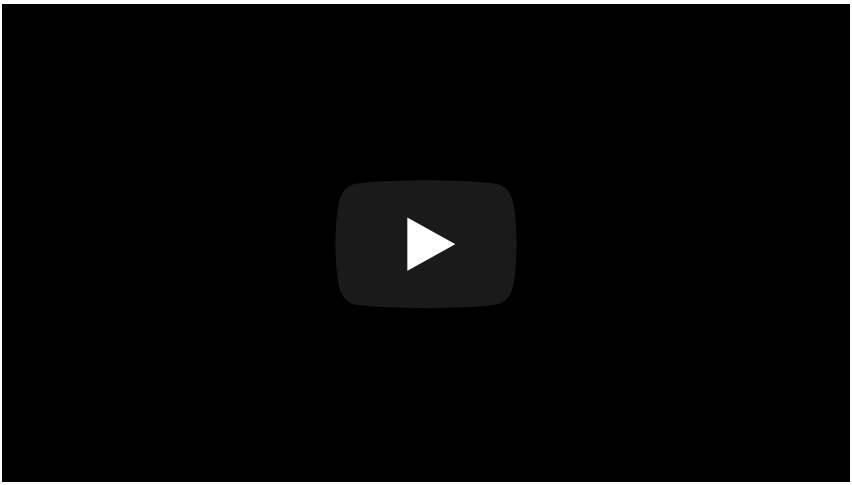
ZFS 在设计之初源自于 Sun 内部多次重写 UFS 的尝试，背负了重构 Solaris 诸多内核子系统的重任，从而不同于 Linux 的文件系统只负责文件系统的功能而把其余功能（比如内存脏页管理， IO调度）交给内核更底层的子系统， ZFS 的整体设计更层次化并更独立，很多部分可能和 Linux/FreeBSD 内核已有的子系统有功能重叠。

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

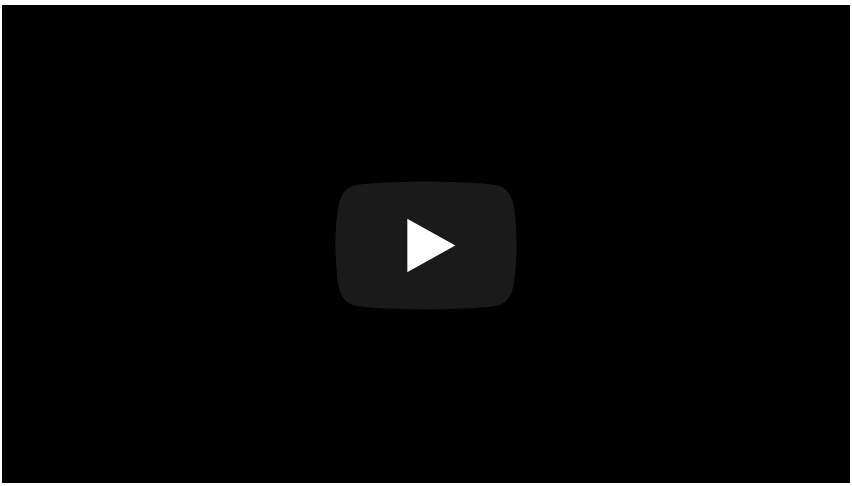


Story Time (Q&A) with Matt and Jeff



ZFS First Mount by Mark Shellenbaum

ZFS past & future by Mark Maybee



早期架构

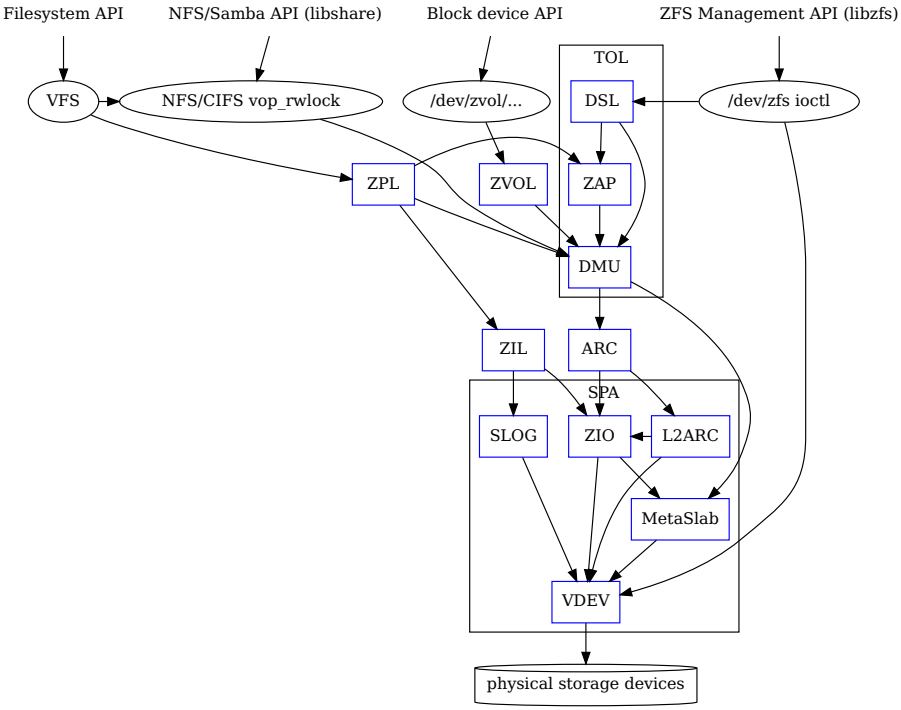
早期 ZFS 在开发时大体可以分为上下三层，分别是 ZPL， DMU 和 SPA ，这三层分别由三组人负责。

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，于是他又从 Solaris 的 Storage Team 抽调了 UFS 部分的负责人 Mark Shellenbaum 和 Mark Maybee 来开发 ZFS。而如今昔日升阳已然日落，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。ZFS 设计中这最初的分工也体现在了 ZFS 现在子系统分层的架构上，继续影响（增强或者限制）ZFS 今后发展的方向。

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（我给方框的子系统加上了超链接）：


Filesystem API
NFS/Samba API (libshare)
Block device API
ZFS Management API (libzfs)
TOL
VFS
NFS/CIFS vop_rwlock
/dev/zvol/...
DSL
/dev/zfs ioctl
ZPL
ZVOL
ZAP
DMU
ZIL
ARC
SPA
SLOG
ZIO
L2ARC
MetaSlab
VDEV
physical storage devices

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 两大部分和其余一些小的子系统。

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 文件系统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 ，在别的 CoW 文件系统比如 Btrfs 和 WAFL 中也有这个特点。 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

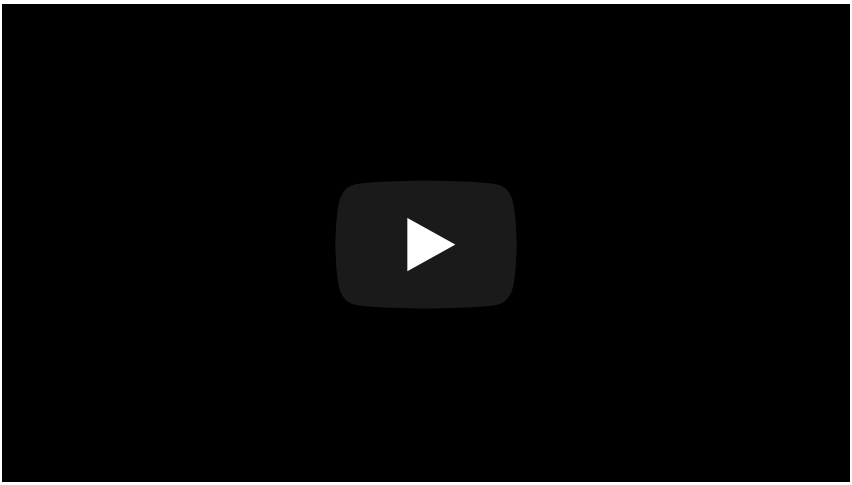
VDEV 在 ZFS 中的作用相当于 Linux 内核的 Device Mapper 层或者 FreeBSD GEOM 层，提供 Stripe/Mirror/RAIDZ 之类的多设备存储池管理和抽象。

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。

除了用于存储数据的 Stripe/Mirror/RAIDZ 之类的 VDEV ，还有一些特殊用途的 VDEV ，包括提供二级缓存的 L2ARC 设备，以及提供 ZIL 高速日志的 SLOG 设备。

ZIO

ZIO Pipeline by George Wilson



ZFS I/O

作用相当于内核的 IO scheduler 和 pagecache write back 机制。 OpenZFS Summit 有个演讲整理了 ZIO 流水线的工作原理。 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 4KiB 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诸如以下操作：

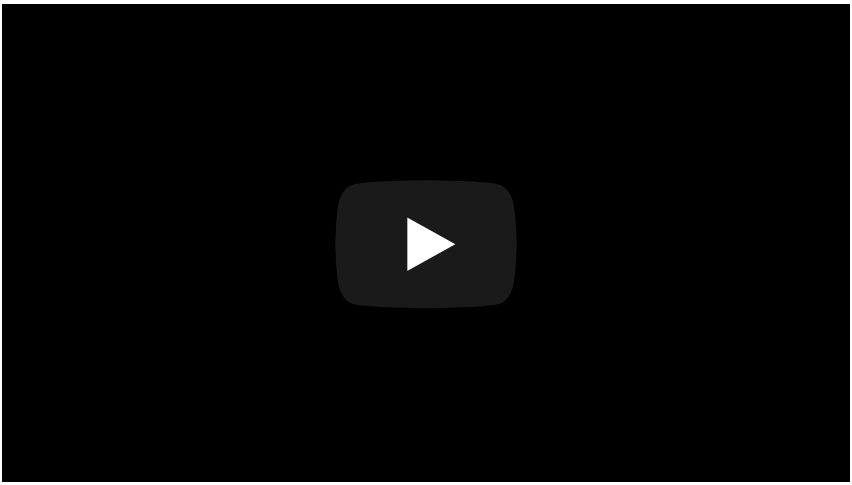
1. 用压缩算法，压缩/解压数据块。
2. 查询 dedup table ，对数据块去重。
3. 加密/解密数据块。
4. 计算数据块的校验和。
5. 如果底层分配器不能分配完整的 128KiB （或 zpool 设置的大小），那么尝试分配多个小块，然后用多个 512B 的指针间接块连起多个小块的，拼装成一个虚拟的大块，这个机制叫 gang block 。通常 ZFS 中用到 gang block 时，整个存储池处于极度空间不足的情况，由 gang block 造成严重性能下降，而 gang block 的意义在于在空间接近要满的时候也能 CoW 写入一些元数据，释放亟需的

存储空间。

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。

MetaSlab

MetaSlab Allocation Performance by Paul Dagnelie



MetaSlab 是 ZFS 的块分配器。 VDEV 把存储设备抽象成存储池之后， MetaSlab 负责实际从存储设备上分配数据块，跟踪记录可用空间和已用空间。

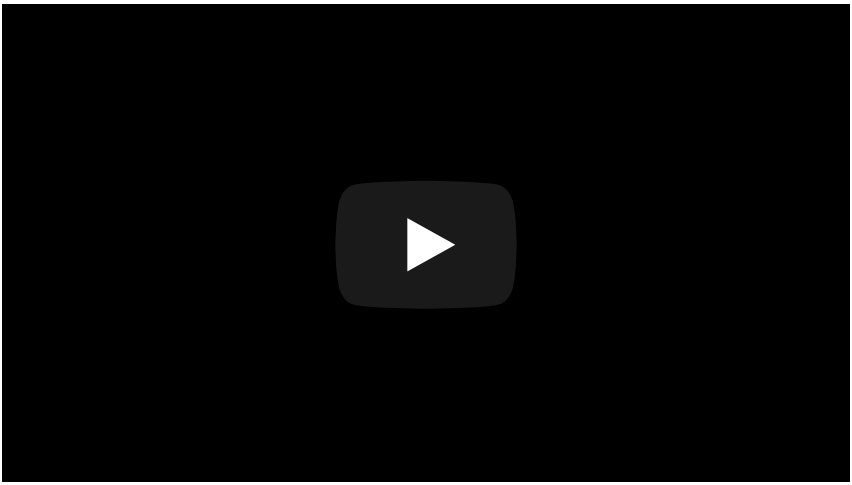
叫 MetaSlab 这个名字是因为 Jeff 最初同时也给 Solaris 内核写过 slab 分配器，一开始设计 SPA 的时候 Jeff 想在 SPA 中也利用 Solaris 的 slab 分配器对磁盘空间使用类似的分配算法。后来 MetaSlab 逐渐不再使用 slab 算法，只有名字留了下来。

MetaSlab 的结构很接近于 FreeBSD UFS 的 cylinder group，或者 ext2/3/4 的 block group，或者 xfs 的 allocation group，目的都是让存储分配策略「局域化」，或者说让近期分配的数据块的物理地址比较接近。在存储设备上创建 zpool 的时候，首先会尽量在存储设备上分配 200 个左右的 MetaSlab，随后给 zpool 增加设备的话使用接近的 MetaSlab 大小。每个 MetaSlab 是连续的一整块空间，在 MetaSlab 内对数据块空间做分配和释放。磁盘中存储的 MetaSlab 的分配情况是按需载入内存的，系统 import zpool 时不需要载入所有 MetaSlab 到内存，而只需加载一小部分。当前载入内存的 MetaSlab 剩余空间告急时，会载入别的 MetaSlab 尝试分配，而从某个 MetaSlab 释放空间不需要载入 MetaSlab。

OpenZFS Summit 也有一个对 MetaSlab 分配器性能的介绍，可以看到很多分配器内的细节。

ARC

ELI5: ZFS Caching Explain Like I'm 5: How the ZFS Adaptive Replacement Cache works



Adaptive Replacement Cache

ARC 的作用相当于 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。最近 FOSDEM 2019 有一个介绍 ZFS ARC 工作原理的视频。

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 平衡的好处之外，也有别的有利的一面。 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。

因为 ARC 工作在 ZIO 上层，所以 ARC 中缓存的数据是经过 ZIO 从存储设备中读取出来之后解压、解密等处理之后的，原始的数据。最近 ZFS 的版本有支持一种新特性叫 Compressed ARC ，打破 ARC 和 VDEV 中间 ZIO 的壁垒，把压缩的数据直接缓存在 ARC 中。这么做是因为压缩解压很快的情况下，压缩的 ARC 能节省不少内存，让更多数据保留在 ARC 中从而提升缓存利用率，并且在有 L2ARC 的情况下也能增加 L2ARC 能存储的缓存。

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。

SLOG

Separate intent LOG

SLOG 是额外的日志记录设备。 SLOG 之于 ZIL 有点像 L2ARC 之余 ARC ， L2ARC 是把内存中的 ARC 放入额外的高速存储设备，而 SLOG 是把原本和别的数据块存储在一起的 ZIL 放到额外的高速存储设备。

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（object）」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 。所有经过 DMU 子系统的对象都有 CoW 语义，也意味着 ZFS 中不能对某些文件可选地关闭 CoW，不能提供数据库应用的 direct IO 之类的接口。

「对象（object）」抽象是 DMU 最重要的抽象，一个对象的大小可变，占用一个或者多个数据块（默认一个数据块 128KiB ）。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。当对象使用多个数据块存储时， DMU 提供间接块（indirect block）来引用这些数据块。 间接块很像传统 Unix 文件系统（Solaris UFS 或者 Linux ext2）中的一级二级三级间接块， 一个间接块存储很多块指针（block pointer），多个间接块形成树状结构，最终一个块指针可以引用到一个对象。 更现代的文件系统比如 ext4/xfs/btrfs/ntfs 提供了 extent 抽象，可以指向一个连续范围的存储块， 而 ZFS 不使用类似 extent 的抽象。DMU 采用间接块而不是 extent，使得 ZFS 的空间分配更趋向碎片化，为了避免碎片化造成的性能影响，需要尽量延迟写入使得一次写入能在磁盘上 尽量连续，这里 ARC 提供的缓存和 ZIO 提供的流水线对延迟写入避免碎片有至关重要的帮助。

有了「对象（object）」的抽象之后， DMU 进一步实现了「对象集（objectset）」的抽象， 一个对象集中保存一系列按顺序编号的 dnode （ ZFS 中类似 inode 的

数据结构），每个 dnode 有足够空间 指向一个对象的最多三个块指针，如果对象需要更多数据块可以使用间接块，如果对象很小也可以直接压缩进 dnode 。随后 DSL 又进一步用对象集来实现数据集（dataset）抽象，提供比如文件系统（filesystem ）、快照（snapshot）、克隆（clone）之类的抽象。一个对象集中的对象可以通过 dnode 编号相互引用， 就像普通文件系统的硬链接引用 inode 编号那样。

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在诊断信息中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 dnode 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 dnode）的引用。

在 ZFS First Mount by Mark Shellenbaum 的8:48 左右 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，也就是说每个文件都有个 ZAP ，其中有叫做 size 呀 owner 之类的键值对，就像是个 JSON 对象那样，这让 ZPL 一开始很容易设计原型并且扩展。然后文件夹内

容列表有另一种数据结构 ZDS（ZFS Directory Service），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和对象集的概念，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。

DSL 管理数据集时，也负责管理数据集上附加的属性。ZFS 每个数据集有个属性列表，这些用 ZAP 存储，DSL 则需要根据数据集的上下级关系，计算出继承的属性，最终指导 ZIO 层面的读写行为。

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。

ZIL

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。 如果要让每次 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 开启日志支持会造成每次文件系统写入操作需要写两次到设备上， 一次写入日志，再一次覆盖文件系统内容；在 ZIL 实现中则不需要重复写两次， 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 文件系统语义的抽象，也就是包括文件、目录、软链接、套接字这些抽象以及 inode 访问时间、权限那些抽象，ZPL 是 ZFS 中对一个普通 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，然后 DMU 对象集对应到 ZPL 下的一个文件系统。 也就是说 ZPL 负责把操作系统 VFS 抽象层的那些文件系统操作接口，翻译映射到基于 DMU 和 ZAP 的抽象上。传统 Unix 中的管道、套接字、软链接之类的没有什么数据内容的东西则在 ZPL 直接用 dnode 实现出来。 ZPL 也需要进一步实现文件权限、所有者、访问日期、扩展属性之类杂七杂八的文件系统功能。

在 ZFS First Mount by Mark Shellenbaum 中介绍了很多在最初实现 ZPL 过程中的坎坷， ZPL 的困难之处在于需要兼容现有应用程序对传统文件系统 API 的使用方式，所以他们需要大量兼容性测试。视频中讲到非常有意思的一件事是， ZFS 在设计时不想重复 Solaris UFS 设计中的很多缺陷，于是实现 VFS API 时有诸多取舍和再设计。 其中他们遇到了 `VOP_RWLOCK` ，这个是 UFS 提供的文件级别读写锁。对一些应用尤其是 NFS 而言，文件读写锁能保证应用层的一致性，而对另一些应用比如数据库而言， 文件锁的粒度太大造成了性能问题。在设计 ZPL 的时候他们不想在 ZFS 中提供 `VOP_RWLOCK` ，这让 NFS 开发者们很难办（要记得 NFS 也是 Solaris

对 Unix 世界贡献的一大发明）。 最终 ZFS 把 DMU 的内部细节也暴露给了 NFS，让 NFS 基于 DMU 的对象创建时间（TXG id）而不是文件锁来保证 NFS 的一致性。结果是现在 ZFS 中也有照顾 NFS 的代码，后来也加入了 Samba/CIFS 的支持，从而在 ZFS 上设置 NFS export 时是通过 ZFS 的机制而非系统原生的 NFS export 机制。