author: Пешеходов Андрей aka fresco (filesystems@nm.ru)
released: 05.12.2007
modified: 30.07.2008

Архитектура ZFS

Лицензионные замечания

Большую часть документа составляет авторский перевод официальной документации Sun Micrisystems, Inc. и комментариев в исходниках ZFS, немного материала взято из блогов сотрудников этой компании, и совсем уж незначительные добавления составляют авторские ремарки. Получить консультации по порядку лицензирования материала у сообщества OpenSolaris автору не удалось, поэтому, во избежание претензий, этот документ распространяется под лицензией PDL (ее текст доступен по адресу http://opensolaris.org/os/community/documentation/license/pdl_version_101.odt) и принадлежит компании Sun Micrisystems, Inc.

От автора

Для понимания материала читателю совершенно необходимо ознакомится с основами функционирования и администрирования ZFS. Русский перевод "Руководства администратора ZFS" доступен здесь: http://dlc.sun.com/osol/g11n/downloads/docs/current/doc-ru-RU-ZFSADMIN-20070301.tar.bz2 . Материал не ординарен и достаточно сложен для понимания; в связи с этим документ разбит на 2 части: в первой, являющейся своего рода справочным руководством, описан дисковый формат ZFS, во второй описаны некоторые детали алгоритмов работы этой файловой системы.

Введение

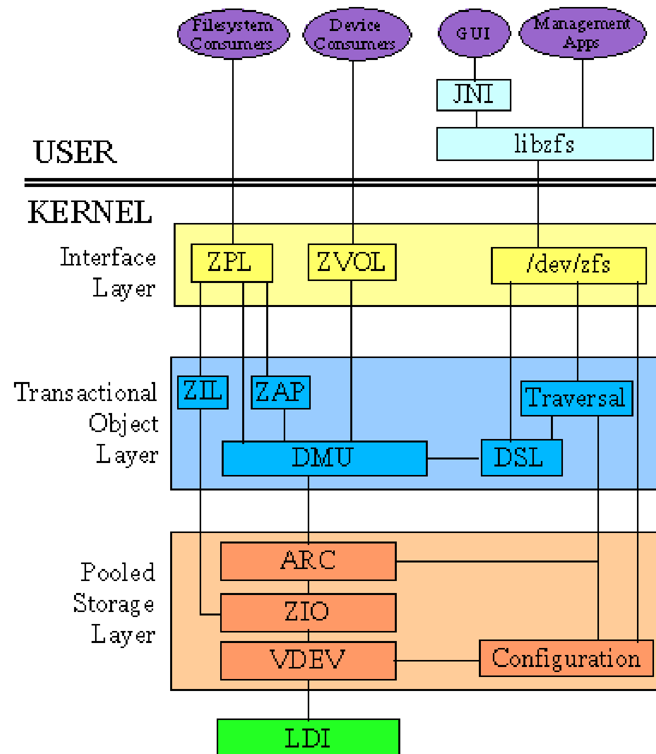
ZFS — это новая технология в ФС-индустрии, обеспечивающая, относительно конкурирующих продуктов, существенно более высокие объемы хранилищ, целостность пользовательских данных, постоянную корректность метаданных, великолепно масштабируемую производительность и удаленную real-time репликацию.

ZFS отходит от принципов традиционных файловых систем за счет отказа от концепции томов. Вместо них ZFS предлагает более сложные storage-пулы, состоящие из одного или более записываемых носителей данных (дисков). Носители могут добавляться и удаляться из пула по мере необходимости. Файловые системы динамически расширяются или урезаются без переразметки конкретных носителей.

ZFS обеспечивает постоянно согласованный дисковый формат, используя модель COW-транзакций (copy on write). Эта модель гарантирует, что дисковые данные не перезаписываются достаточно длительное время, и все обновления в ФС вносятся атомарно.

Программное обеспечение ZFS разбито на 7 основных компонент: SPA (storage pool allocator), DSL (data and snapshot layer), DMU (data management unit), ZAP (ZFS attribute processor), ZPL (ZFS POSIX layer), ZIL (ZFS intent log) и ZVOL (ZFS volume). Дисковые структуры, ассоциированные с каждым из этих компонентов, описывается в следующих главах: SPA — главы 1 и 2, DSL — глава 5, DMU (3), ZAP (4),ZPL (6), ZIL (7), ZVOL (8).

Картинка иллюстрирует роль каждого из этих компонентов и глобальные взаимосвязи между ними.



Часть 1. Дисковая структура ZFS

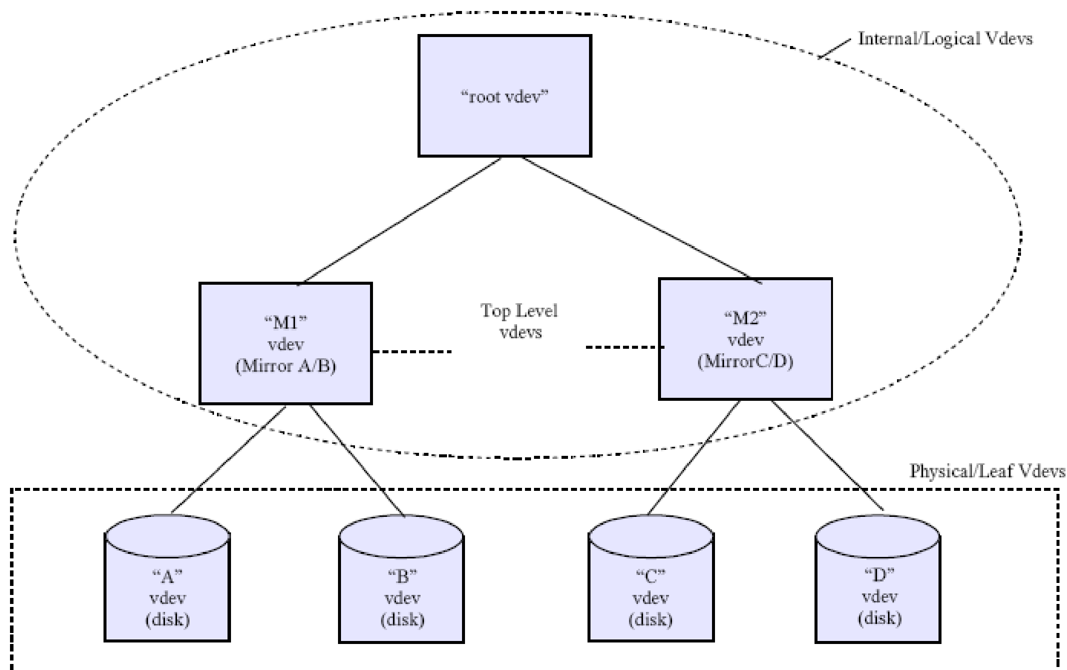
Глава 1. Виртуальные устройства (vdev's), vdev-метки и boot-блок

1.1 Виртуальные устройства

Пулы ZFS выполнены в виде совокупности виртуальных устройств двух типов: физических (называемых также листовыми vdev's) и логических (внутренних vdev's). Физические vdev's — это записываемые блочные устройства (например диски), логические — концептуальные группы физических.

Vdev's выстраиваются в дерево, листьями которого являются физические устройства. Все пулы имеют специальные логический vdev — корень этого дерева.

Все прямые потомки корневого vdev (физические или логические) называются виртуальными устройствами верхнего уровня (top-level vdev).



1.2 Метки виртуальных устройств

Каждое физическое виртуальное устройство содержит структуру размером 256 kb, называемую vdev-меткой. Эта метка содержит информацию о данном устройстве и о всех vdev'ах, с которыми оно разделяет vdev's верхнего уровня. К примеру, метка, хранящаяся на устройстве C имеет информацию о vdev'ах C, D и M2. Строение метки подробно описывается в секции 1.3.

Vdev-метка выполняет 2 задачи: обеспечивает доступ к содержимому пула и используется для проверки целостности и доступности пула. Для обеспечения постоянной целостности самой метки применяется избыточность и модель пошаговой фиксации изменений. Избыточность состоит в том, что каждое физическое устройство пула хранит 4 копии метки (одинаковы в пределах vdev'a, но, понятно, не пула). Изменения в метку вносятся атомарно в 2 шага.

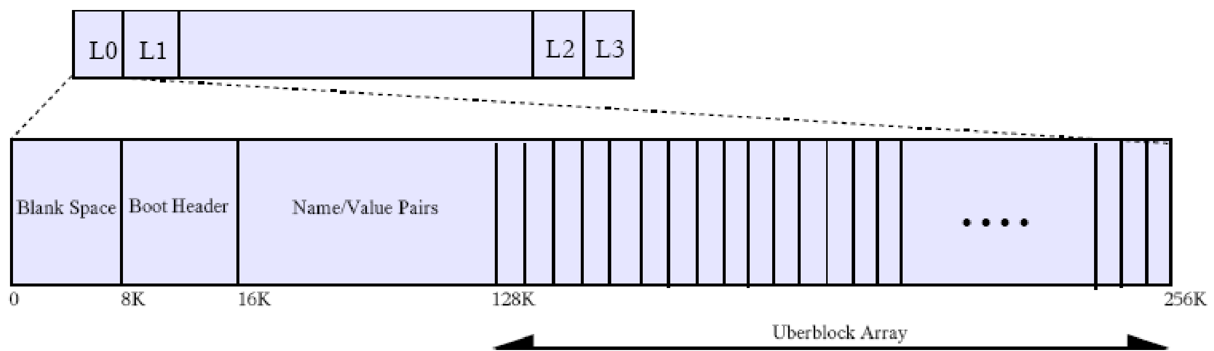
Кроме небольшого времени внесения обновлений все 4 копии метки полностью идентичны и любая из них может быть использована кодом SPA. ZFS помещает 2 копии в начале устройства, и 2 — в конце. Т.к. разрушение диска (или случайная перезапись) чаще всего происходит на некотором протяженном участке, размещение копий меток со значительным пространственным разносом обеспечивает большую надежность.

Положение меток на диске фиксировано и вычисляется в момент добавления устройства в пул. Т.о., к ним не применима семантика COW-транзакций, которая используется почти для всех объектов ZFS — следовательно, метки при обновлении перезаписываются и потенциально подвержены разрушению.

Обновления меток фиксируется в 2 шага. Сначала обновляются нулевая и вторая метки, далее — первая и третья.

1.3 Детали реализации vdev's

Метка состоит из четырех частей: 8 kb пустого пространства, 8 kb заголовка загрузочного блока, 112 kb пар имя/значение (далее NV-пар) и 128k 1-килобайтных uberblock-структур.



Структура vdev-метки определена в /usr/src/uts/common/fs/zfs/sys/vdev_impl.h.

```
typedef struct vdev_label {
      char                    vl_pad[VDEV_SKIP_SIZE];            /* 8K*/
      vdev_boot_header_t      vl_boot_header;                    /* 8K */
      vdev_phys_t             vl_vdev_phys;                      /* 112K */
      char                    vl_uberblock[VDEV_UBERBLOCK_RING]; /* 128K */
} vdev_label_t;                                                  /* 256K */
```

ZFS поддерживает 2 распространенных стандарта разметки диска: VTOC (Volume Table Of Content) и EFI. В то время, как EFI-разметка не записывается на диск как часть слайса, располагая особым зарезервированным дисковым пространством, VTOC-информация должна храниться в первых 8 kb диска. Поэтому vdev-метка начинается с неиспользуемых блоков.

Заголовок загрузочного блока — 8-килобайтная структура, зарезервированная для будущего использования. Определена в /usr/src/uts/common/fs/zfs/sys/vdev_impl.h и будет описана ниже.

Следующие 112 kb содержат список NV-пар, которые описывают данное и связанные с ним виртуальные устройства. Формат списка определен в /usr/src/uts/common/fs/zfs/sys/vdev_impl.h.

```
typedef struct vdev_phys {
      char              vp_nvlist[VDEV_PHYS_SIZE - sizeof (zio_block_tail_t)];
      zio_block_tail_t  vp_zbt;
} vdev_phys_t;
```

Видно, что это просто массив символов, заканчивающийся "хвостовым" блоком с контрольной суммой. Тип zio_block_tail_t описан в /usr/src/uts/common/fs/zfs/sys/zio.h

```
typedef struct zio_block_tail {
      uint64_t    zbt_magic;  /* magic */
      zio_cksum_t zbt_cksum;  /* 256-битная контрольная сумма */
} zio_block_tail_t;
```

zio_cksum_t определен в /usr/src/uts/common/fs/zfs/sys/spa.h как массив из четырех 64-битных чисел

```
typedef struct zio_cksum {
      uint64_t    zc_word[4];
} zio_cksum_t;
```

NV-пары хранятся в виде XDR-кодированных NV-списков. Подробнее о XDR кодировании и NV-списках можно узнать из man-страниц libnvpair (3LIB) и nvlist_free (3NVPAIR). В этом списке хранятся следующие параметры:

- "version", тип DATA_TYPE_UINT64 — версия дискового формата, в настоящее время 10 (в /usr/src/uts/common/sys/fs/zfs.h определена константа SPA_VERSION)
- "name", тип DATA_TYPE_STRING — имя пула, за которым числится данное устройство
- "state", тип DATA_TYPE_UINT64 — состояние пула, принимает значения из перечисления pool_state_t, определенного в /usr/src/uts/common/sys/fs/zfs.h

```
/*
 * Состояния пула. Следующие состояния записываются на диск как
 * часть нормального жизненного цикла SPA: ACTIVE, EXPORTED, DESTROYED,
 * SPARE, L2CACHE. Остальные состояния есть просто программные
 * абстракции, используемые на различных уровнях для связывания
 * состояний.
 */
typedef enum pool_state {
      POOL_STATE_ACTIVE = 0,        /* Используется */
      POOL_STATE_EXPORTED,          /* Явно экспортирован*/
      POOL_STATE_DESTROYED,         /* Явно уничтожен */
      POOL_STATE_SPARE,             /* Зарезервирован для горячей замены */
      POOL_STATE_L2CACHE,           /* Устройство ARC 2-го уровня */
      POOL_STATE_UNINITIALIZED,     /* Внутреннее состояние spa_t */
      POOL_STATE_IO_FAILURE,        /* Внутреннее состояние pool */
      POOL_STATE_UNAVAIL,           /* Внутреннее состояние libzfs */
      POOL_STATE_POTENTIALLY_ACTIVE /* Внутреннее состояние libzfs */
} pool_state_t;
```

- "txg", тип DATA_TYPE_UNIT64 — номер группы транзакций, в которой была записана эта метка

- "pool_guid", тип DATA_TYPE_UNIT64 — глобальный уникальный идентификатор пула
- "top_guid", тип DATA_TYPE_UNIT64 — глобальный уникальный идентификатор vdev верхнего уровня для данного поддерева
- "guid", тип DATA_TYPE_UNIT64 — глобальный уникальный идентификатор этого vdev
- "vdev_tree", тип DATA_TYPE_NVLIST — структура nvlist, используемая рекурсивно для представления структуры дерева vdev. Содержит описание для всех связанных с данным виртуальных устройств поддерева. На иллюстрации ниже показана структура vdev_tree для виртуального устройства M1 пула, изображенного выше:

```
vdev_tree
type='mirror'
id=1
guid=16593009660401351626
metaslab_array = 13
metaslab_shift = 22
ashift = 9
asize =519569408
children[0]
    vdev_tree
    type='disk'
    id=2
    guid=6649981596953412974
    path='/dev/dsk/c4t0d0'
    devid='id1,sd@SSEAGATE_ST373453LW_3HW0J0FJ00007404E4NS/a'
children[1]
    vdev_tree
    type='disk'
    id=3
    guid=3648040300193291405
    path='/dev/dsk/c4t1d0'
    devid='id1,sd@SSEAGATE_ST373453LW_3HW0HLAW0007404D6MN/a'
```

Каждый vdev_tree содержит элементы, описанные ниже. Заметьте, что не все из них характерны для всех типов vdev's. NV-список каждого vdev-дерева может содержать только поднабор этих элементов. Полный перечень определен в /usr/src/uts/common/sys/fs/zfs.h.

- "type", тип DATA_TYPE_STRING — строка-идентификатор типа vdev, в настоящее время возможны следующие типы:

Тип	**Определение**
"disk"	листовой vdev, блочное хранилище
"file"	листовой vdev, файловое хранилище (loop device)
"mirror"	внутренний vdev, зеркало
"RAID-Z"	внутренний vdev, RAID-Z
"replacing"	внутренний vdev, упрощенная реализация зеркала, используется ZFS при замене одного диска другим
"root"	внутренний vdev, корень vdev-дерева

- id, тип DATA_TYPE_UINT64 — индекс данного vdev в специальном массиве его родителя
- guid, тип DATA_TYPE_UINT64 — уникальный глобальный идентификатор этого элемента vdev-дерева

- "path", DATA_TYPE_STRING — путь к устройству, используется только для листовых vdev's
- "devid", DATA_TYPE_ STRING — ID устройства для данного элемента, используется только в листовых vdev's
- "metaslab_array", тип DATA_TYPE_UINT64 — номер объекта, содержащего массив заголовков метаслабов данного виртуального устройства. Каждый элемент этого массива (ma[i]), в свою очередь, является номером объекта для карты пространства i-того metaslab. Metaslab в ZFS — это нечто вроде группы размещения из XFS. Каждый диск разбит на множество metaslab со своими структурами учета пространства. Подробнее — см. во второй части статьи.
- "metaslab_shift", тип DATA_TYPE_UINT64 — log2 от размера metaslab диска.
- "ashift", тип DATA_TYPE_UINT64 — log2 от размера минимального выделяемого элемента для данного vdev'a верхнего уровня. 10 для RAID-Z-конфигурации, 9 в остальных случаях.
- "asize", тип DATA_TYPE_UINT64 — Количество места, которое может быть выделено из данного vdev'a верхнего уровня.
- "children", тип DATA_TYPE_NVLIST_ARRAY — массив NV-списков для каждого потомка данного элемента vdev-дерева (только для внутренних vdev's).

Сразу за списками NV-пар в vdev-метке следует массив uber-блоков. Uber-блок — это часть метки, содержащая информацию, необходимую для доступа к содержимому пула; в некотором роде он подобен суперблоку. Только один uber-блок считается активным в настоящий момент времени — это uber-блок с наибольшим номером группы транзакций и корректной SHA-256 контрольной суммой.

Для гарантии доступа к активному uber-блоку он никогда не перезаписывается. Вместо этого обновленный uber-блок записывается на место следующего элемента кольцевого списка (т.е. применяется COW-журналирование). Структура uber-блока определена в /usr/src/uts/common/fs/zfs/sys/uberblock_impl.h.

```
struct uberblock {
      uint64_t    ub_magic;
      uint64_t    ub_version;
      uint64_t    ub_txg;
      uint64_t    ub_guid_sum;
      uint64_t    ub_timestamp;
      blkptr_t    ub_rootbp;
};
```

ub_magic

Показывает, что устройство содержит ZFS.

ub_version

Используется для хранения номера версии дискового формата, в котором записаны данные этого устройства. В настоящее время 10 (в /usr/src/uts/common/sys/

fszfs.h определена константа SPA_VERSION)

ub_txg

Все данные, вносимые в ZFS, записываются в рамках какой-либо группы транзакций, каждая их которых имеет свой номер. Поле ub_txg указывает на группу транзакций, в рамках которой был записан данный uber-блок. Номер ub_txg должен быть больше или равен номеру "txg", хранимому в NV-списке данной метки для признания ее валидной.

ub_guid_sum

Поле служит для проверки доступности данного виртуального устройства в пределах пула. При открытии пула ZFS обходит все листовые vdev's и подсчитывает сумму все guid (тех, что хранятся в паре guid списка NV-пар). Эта вычисленная сумма и сверяется с ub_guid_sum.

ub_timestamp

Содержит время записи uber-блока в формате UTC.

ub_rootbp

Здесь хранится структура blkptr_t, указывающая положение MOS (Meta Object Set, будут описаны ниже).

1.4 Boot-блок

Сразу за нулевой и первой метками располагается 3.5 Mb дискового пространства, зарезервированного для будущего использования — т.н. boot-блок.

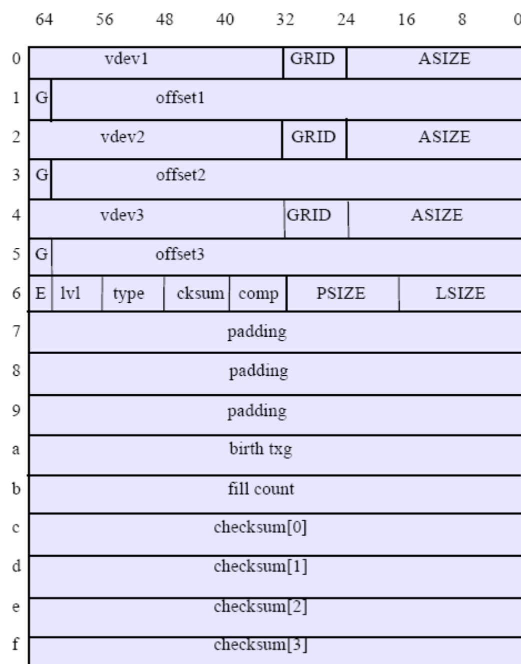
Глава 2. Blockpointers и косвенные блоки

Данные передаются между диском и памятью в элементах, называемых блоками. Указатель на блок, 128-байтная структура, используется для описания координат, проверки и определения свойств блоков данных на диске.

Формат структуры blkptr_t описан в /usr/src/uts/common/fs/zfs/sys/spa.h.

```
typedef struct blkptr {
      dva_t       blk_dva[3]; /* 128-bit Data Virtual Address */
      uint64_t    blk_prop;   /* размер, сжатие, тип, и т.д. */
      uint64_t    blk_pad[3]; /* зарезервировано */
      uint64_t    blk_birth;  /* номер группы транзакций */
      uint64_t    blk_fill;   /* fill count */
      zio_cksum_t blk_cksum;  /* 256-битная контрольная сумма */
} blkptr_t;

/* Бинарная структура DVA — два 64-битных слова */
typedef struct dva {
      uint64_t    dva_word[2];
} dva_t;
```

На иллюстрации показана логическая структура DVA



2.1 Data Virtual Address

DVA (виртуальный адрес данных) — есть комбинация ID листового виртуального устройства и номера блока на нем. ZFS обеспечивает возможность хранения до 3-х копий данных, адресуемых этим указателем, причем каждая будет располагаться по своему DVA. Данные в копиях полностью идентичны. Эта возможность введена для предохранения наиболее критичных метаданных от разрушений диска. С ZFS версии 6 таким же образом можно резервировать и пользовательские данные.

vdev — 32-битный integer, уникально идентифицирующий физическое виртуальное устройство, содержащее этот блок. offset —63-битное значение смещения блока в пределах диска (первые 4 Mb — метки и загрузочный блок — пропускаются).

Порция offset хранит смещение в секторах (512 байт). Смещение в байтах от начала слайса считается так:

phys_block_address = (offset << 9) + 0x400000

Фактически выполняется умножение на 512 и прибавление 4 Mb.

2.2 GRID

Зарезервировано под дополнительную информацию о формате RAID-Z, пока не используется.

2.3 Gang

Gand-блоком (gang — набор, комплект) называется блок, содержащий указатели на блоки. Gang-блоки используются в ситуации сильной фрагментации, когда невозможно выделить непрерывный участок требуемого размера. В таком случае выделяется несколько блоков меньшего размера (в сумме — требуемый размер), указатели на которые помещаются в gang-блок. Указатель на этот gang-блок возвращается заказчику, давая тому ощущение единого блока. Установленный бит G в маске blk_prop структуры blkptr_t является индикаторм того, что данный blockpointer указывает на gang-блок.

Gang-блок имеет 512 байт в размере и содержит до 3 указателей на блоки, завершающиеся 64-битной контрольной суммой. Его формат описан в /usr/src/uts/common/fs/zfs/sys/zio.h, соответствующие макросы определены там же:

```
typedef struct zio_gbh {
      blkptr_t          zg_blkptr[SPA_GBH_NBLKPTRS];
      uint64_t          zg_filler[SPA_GBH_FILLER];
      zio_block_tail_t  zg_tail;
} zio_gbh_phys_t;
```

zg_filler — выравнивание.

2.4 Контрольная сумма

По умолчанию ZFS контролирует целостность всех блоков данных и метаданных с помощью контрольных сумм, подсчитываемых различными алгоритмами. На примененный к данному блоку алгоритм указывает 8-битная порция cksum в DVA указателя. Она может принимать следующие значения (см. /usr/src/uts/common/fs/zfs/sys/zio.h):

```
enum zio_checksum {
      ZIO_CHECKSUM_INHERIT = 0,     /* Наследуется */
      ZIO_CHECKSUM_ON,              /* fletcher2 */
      ZIO_CHECKSUM_OFF,
      ZIO_CHECKSUM_LABEL,           /* SHA-256 */
      ZIO_CHECKSUM_GANG_HEADER,     /* SHA-256 */
      ZIO_CHECKSUM_ZILOG,           /* fletcher2 */
      ZIO_CHECKSUM_FLETCHER_2,      /* fletcher2 */
      ZIO_CHECKSUM_FLETCHER_4,      /* fletcher4 */
      ZIO_CHECKSUM_SHA256,          /* SHA-256 */
      ZIO_CHECKSUM_FUNCTIONS
};

#define     ZIO_CHECKSUM_ON_VALUE   ZIO_CHECKSUM_FLETCHER_2
#define     ZIO_CHECKSUM_DEFAULT    ZIO_CHECKSUM_ON
```

256-битная контрольная сумма подсчитывается для каждого блока с помощью алгоритма, определенного в cksum (кроме блоков, для которых cksum равно ZIO_CHECKSUM_OFF). Gang- и zilog-блоки (см. ниже) сами содержат свою контрольную сумму (порция _tail соответствующей структуры), для остальных блоков она хранится в указателе.

2.5 Сжатие

ZFS поддерживает несколько алгоритмов сжатия данных, конкретный алгоритм определяется порцией comp в DVA указателя на блок. Она может принимать следующие значения (см. /usr/src/uts/common/fs/zfs/sys/zio.h):

```
enum zio_compress {
      ZIO_COMPRESS_INHERIT = 0,
      ZIO_COMPRESS_ON,              /* LZJB */
      ZIO_COMPRESS_OFF,
      ZIO_COMPRESS_LZJB,            /* LZJB */
      ZIO_COMPRESS_EMPTY,           /* Без компрессии */
      ZIO_COMPRESS_GZIP_1,          /* GZIP */
      ZIO_COMPRESS_GZIP_2,
      ZIO_COMPRESS_GZIP_3,
      ZIO_COMPRESS_GZIP_4,
      ZIO_COMPRESS_GZIP_5,
      ZIO_COMPRESS_GZIP_6,
      ZIO_COMPRESS_GZIP_7,
      ZIO_COMPRESS_GZIP_8,
      ZIO_COMPRESS_GZIP_9,
      ZIO_COMPRESS_FUNCTIONS
};

#define     ZIO_COMPRESS_ON_VALUE   ZIO_COMPRESS_LZJB
#define     ZIO_COMPRESS_DEFAULT    ZIO_COMPRESS_OFF
```

2.6 Размер блока

Размер блока описывается тремя различными полями DVA.

- lsize — логический размер, размер данных без учета компрессии, RAID-Z и расходов на gang-блоки
- psize — физический размер блока на диске после компрессии
- asize — выделенный размер, размер данных с учетом всех gang-заголовков и RAID-Z информации

Если сжатие отключено и данная ZFS не является RAID-Z хранилищем, значения всех трех полей буду равны. Все размеры считаются в 512-байтных секторах.

2.7 Порядок следования байт

ZFS инвариантна к порядку следования байт (big или little endian), т.е. позволяет перемещать пулы между машинами с разными CPU-архитектурами. Бит E в DVA указателя определяет порядок следования байт в блоке: 1 — little endian, 0 — big endian. Данные всегда записываются в родном для данной архитектуры формате, в случае перемещении пула на другую машину данные конвертируются при чтении.

2.8 Тип

Поле type в DVA указателя определяет тип DMU-бъекта, данные которого содержатся в блоке. Возможные значения определены в /usr/src/uts/common/fs/zfs/sys/dmu.h (подробнее см. ниже).

2.9 Широта

Порция lvl содержит "широту" данного блока в структуре косвенной адресации. См. ниже.

2.10 Степень наполнения

Порция fill содержит количество ненулевых указателей под данным указателем. Для блоков данных это 1, т.е. под ним нет указателей. Для блоков косвенной адресации это поле принимает некоторое осмысленное значение. См. ниже.

2.11 Транзакция рождения

Номер группы транзакций, во время которой блок был создан

Глава 3. Data Management Unit

Модуль управления данными объединяет блоки в логические элементы, называемые объектами, которые в последствии могут группироваться DMU в наборы объектов (objsets).

3.1 Объекты

За исключением небольшой доли инфраструктуры, описанной ранее, все сущности в ZFS есть объекты. Возможные типы объектов ZFS перечислены в /usr/src/uts/common/fs/zfs/sys/dmu.h

```
typedef enum dmu_object_type {
      DMU_OT_NONE,                 /* Unallocated объект */

      /* general: */
      DMU_OT_OBJECT_DIRECTORY,     /* DSL-объект, каталог, ZAP-объект */
      DMU_OT_OBJECT_ARRAY,         /* Объект используется для хранения
                                    * массива номеров объектов */
      DMU_OT_PACKED_NVLIST,        /* Упакованный NV-список */
      DMU_OT_PACKED_NVLIST_SIZE,   /* UINT64 */
      DMU_OT_BPLIST,               /* Список указателей на блоки,
                                    * используется для хранения списка
                                    * блоков, удаленных в последнем
                                    * снапшоте (deadlist), и списка
                                    * отложенно освобожденных блоков
                                    * (deffered free list) */
      DMU_OT_BPLIST_HDR,           /* Заголовок BPLIST, хранит структуру
                                    * bplist_phys_t */
```

```
	/* spa: */
	DMU_OT_SPACE_MAP_HEADER,	/* UINT64 */
	DMU_OT_SPACE_MAP,		/* Список используемых SPA дисковых
					 * блоков */

	/* zil: */
	DMU_OT_INTENT_LOG,		/* Intent-лог */

	/* dmu: */
	DMU_OT_DNODE,			/* dnode */
	DMU_OT_OBJSET,			/* Набор объектов */

	/* dsl: */
	DMU_OT_DSL_DIR,
	DMU_OT_DSL_DIR_CHILD_MAP,	/* UINT64 */
	DMU_OT_DSL_DS_SNAP_MAP,		/* Снапшот-информация для набора
					 * данных, DLS ZAP объект */
	DMU_OT_DSL_PROPS,		/* Свойства для объекта DSL каталога */
	DMU_OT_DSL_DATASET,

	/* zpl: */
	DMU_OT_ZNODE,			/* ZNODE */
	DMU_OT_ACL,			/* ACL */
	DMU_OT_PLAIN_FILE_CONTENTS,	/* Содержимое простого файла */
	DMU_OT_DIRECTORY_CONTENTS,	/* ZAP-объект каталога ZPL */
	DMU_OT_MASTER_NODE,		/* ZAP-объект главного узла ZPL,
					 * головной объект , применяемый для
					 * идентификации корневого каталога,
					 * delete-очереди и версии ФС */
	DMU_OT_UNLINKED_SET,		/* Очередь удалений, которые
					 * выполнялись на момент сбоя ФС.
					 * При следующем монтировании ФС все
					 * файлы/каталоги из этой очереди
					 * будут удалены  */

	/* zvol: */
	DMU_OT_ZVOL,			/* UINT8 */
	DMU_OT_ZVOL_PROP,		/* ZAP */

	/* Только для тестирования! */
	DMU_OT_PLAIN_OTHER,		/* UINT8 */
	DMU_OT_UINT64_OTHER,		/* UINT64 */
	DMU_OT_ZAP_OTHER,		/* ZAP */

	/* Новые типы объектов */
	DMU_OT_ERROR_LOG,		/* ZAP */
	DMU_OT_SPA_HISTORY,		/* UINT8 */
	DMU_OT_SPA_HISTORY_OFFSETS,	/* spa_his_phys_t */
	DMU_OT_POOL_PROPS,		/* ZAP */
	DMU_OT_DSL_PERMS,		/* ZAP */
	DMU_OT_NUMTYPES
} dmu_object_type_t;
```

Любой объект описывается 512-байтной структурой dnode, которая определяет и организует набор блоков, составляющих объект (/usr/src/uts/common/fs/zfs/sys/dnode.h):

```
/* Макрос определяет, в чем (байтах или секторах) измеряется
```

```
 * dn_used */
#define     DNODE_FLAG_USED_BYTES   (1<<0)

typedef struct dnode_phys {
      uint8_t dn_type;              /* dmu_object_type_t */
      uint8_t dn_indblkshift;       /* log2(indirect block size) */
      uint8_t dn_nlevels;           /* 1=dn_blkptr->data blocks */
      uint8_t dn_nblkptr;           /* длина dn_blkptr */
      uint8_t dn_bonustype;         /* тип данных в bunus-буфере */
      uint8_t dn_checksum;          /* тип ZIO_CHECKSUM */
      uint8_t dn_compress;          /* тип ZIO_COMPRESS */
      uint8_t dn_flags;             /* DNODE_FLAG_* */
      uint16_t dn_datablkszsec;     /* размер блока данных в секторах */
      uint16_t dn_bonuslen;         /* длина dn_bonus */
      uint8_t dn_pad2[4];

      uint64_t dn_maxblkid;         /* наибольший ID выделенного блока */
      uint64_t dn_used;             /* количество байт или секторов
                                     * дискового пространства */

      uint64_t dn_pad3[4];

      /* Поля переменной длины */

      blkptr_t dn_blkptr[1];
      uint8_t dn_bonus[DN_MAX_BONUSLEN];  /* bonus-буфер */
} dnode_phys_t;
```

dn_type
Тип DMU-объекта

dn_indblkshift и **dn_datablkszsec**
ZFS поддерживает различные размеры косвенных блоков и блоков данных. Возможны значения от 512 байт до 128 kb. dn_indblkshift содержит log2 от размера (в байтах) косвенного блока. dn_datablkszsec — размер блока данных в секторах (от 1 до 256).

dn_nblkptr и **dn_blkptr**
Массив dn_blkptr может содержать от 1 до 3 указателей на блоки. Устанавливается во время создания dnode и остается неизменным до момента его удаления. dn_nblkptr хранит количество указателей в массиве dn_blkptr.

dn_nlevels
Количество уровней, составляющих данный объект. Эти уровни обычно называются уровнями косвенности. dnode имеет ограниченное количество указателей на блоки, и даже при максимальном размере блока (128 kb) и трех указателях размер объекта был бы ограничен 384 kb. Для поддержки больших объектов используются косвенные блоки. Как и в традиционных файловых системах, косвенным блоком в ZFS называется блок, содержащий указатели на другие блоки. Количество указателей зависит только от размера косвенного блока и может достигать 1024 штук. По мере увеличения размера объекта могут добавляться новые косвенные блоки и даже уровни. ZFS поддерживает до 6 уровней косвенности — таким образом, максимальный размер файла ограничен

2^64 байт. Широтой блока называется номер уровня, на котором он расположен — от 0 до 5, причем нулевым считается уровень блоков с данными.

dn_maxblkid

Блок в пределах объекта идентифицируется по ID. Блоки на каждом уровне нумеруются числом от нуля до N. Поле dn_maxblkid содержит номер последнего блока на уровне данных (нулевом). Это позволяет ZFS быстро отыскивать нужные косвенные блоки для достижения требуемого блока данных. Рассмотрим, к примеру, объект со 128-kb косвенными блоками — каждый из них содержит 1024 указателя. Для достижения блока 16360 нулевого уровня нам необходим 15-й блок 1-го уровня:

level1_blk_id = 16360 % 1024 = 15

Это несложное вычисление может быть рекурсивно применено и для большего количества уровней. Таким образом, образуется упрощенная древо-подобная структура, имеющая несколько большую производительность поиска произвольного блока относительно линейного перебора косвенных ссылок.

dn_used

Поле хранит сумму значений asize всех указателей на блоки для данного объекта. Размерность (байты или секторы) определяется макросом DNODE_FLAG_USED_BYTES.

dn_bonus, **dn_bonustype** и **dn_bonuslen**

Bonus-буфер, в зависимости от типа объекта, может содержать от 64 до 320 байт данных (текущая длина лежит в dn_bonuslen). dn_bonustype определяет тип данных в bonus-буфере и может принимать некоторые значения из перечисления dmu_object_type_t.

Тип	Определение	Структура метаданных
DMU_OT_PACKED_NVLIST_SIZE	Размер (в байтах) объекта типа DMU_OT_PACKED_NVLIST	uint64_t
DMU_OT_SPACE_MAP_HEADER	Заголовок карты пространства	space_map_obj_t
DMU_OT_DSL_DIR	Объект DSL каталога, используется для определения отношений между связанными наборами данных (datasets)	dsl_dir_phys_t
DMU_OT_DSL_DATASET	DSL dataset объект применяется для организации снапшотов и статистики использование для объектов типа DMU_OT_OBJSET	dsl_dataset_phys_t
DMU_OT_ZNODE	Метаданные ZPL	znode_phys_t

3.2 Наборы объектов

Наборы объектов используется в ZFS для группировки связанных объектов в

файловой системе, снапшоте, клоне или томе. Набор определяется 1 kb структурой objset_phys_t (см. /usr/src/uts/common/fs/zfs/sys/dmu.h).

```
typedef struct objset_phys {
      dnode_phys_t os_meta_dnode;
      zil_header_t os_zil_header;
      uint64_t os_type;

      char os_pad[1024 - sizeof (dnode_phys_t) - sizeof (zil_header_t)
                                                      - sizeof (uint64_t)];
} objset_phys_t;
```

os_type

DMU поддерживает несколько типов наборов объектов, каждый из которых имеет четко определенный формат объектов. Возможные типы перечислены в /usr/src/uts/common/fs/zfs/sys/dmu.h:

```
typedef enum dmu_objset_type {
      DMU_OST_NONE,           /* Не инициализированный набор */
      DMU_OST_META,           /* Набор объектов DSL */
      DMU_OST_ZFS,            /* Набор объектов ZPL */
      DMU_OST_ZVOL,           /* Набор объектов ZVOL */
      DMU_OST_OTHER,          /* Только для тестирования! */
      DMU_OST_ANY,            /* Будьте осторожны! */
      DMU_OST_NUMTYPES
} dmu_objset_type_t;
```

os_zil_header

Заголовок журнала (intetn log в терминах ZFS) для данного набора объектов.

meta_dnode

Как было упомянуто ранее, каждый объект в ZFS описывается структурой dnode_phys_t. Совокупность структур dnode_phys_t, описывающих объекты в наборе, хранится в объекте, описываемом мета-dnode. Данные в нем форматированы в виде массива структур dnode_phys_t.

Каждый объект в пределах набора уникально определяется 64-битным числом, называемым номером объекта. Это просто индекс в массиве структур dnode_phys_t.

Часть 4. DSL

DSL (Dataset and Snapshot Layer — уровень наборов данных и снапшотов) обеспечивает механизмы описания взаимоотношений и свойств наборов объектов. Перед тем, как начать обсуждение DSL, сделаем небольшой обзор особенностей наборов объектов ZFS.

ZFS предоставляет возможность создавать 4 вида датасетов:

- Файловая система: хранит и организует объекты в доступном, совместимом с POSIX формате.
- Клон: модифицируемая копия ФС, начальное состояние которой определяется некоторым снапшотом.
- Снапшот: read-only срез файловой системы, клона или тома в некий момент времени.
- Том: логический том, экспортируемый ZFS в виде блочного устройства.

ZFS поддерживает несколько операций и/или конфигураций, которые вызывают сложные зависимости и взаимоотношения между наборами объектов. Задача DSL — управлять этими связями.

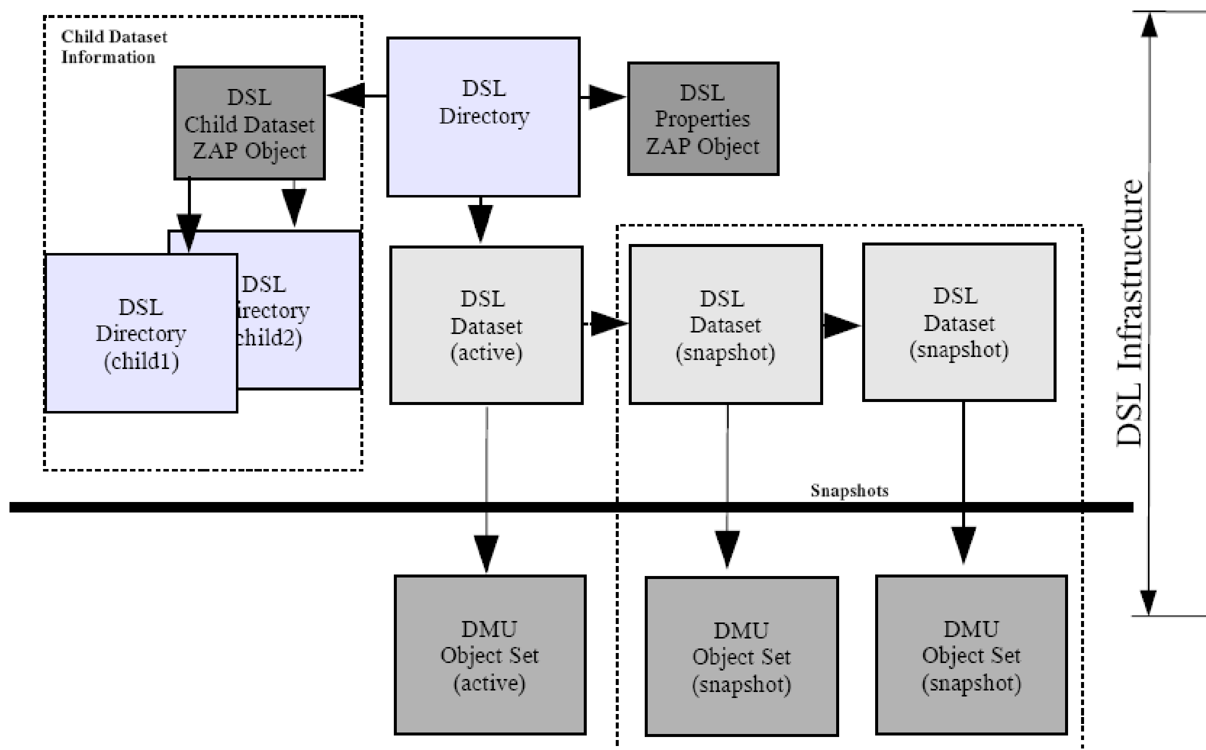
- Клоны: связаны со снапшотами, от которых они происходят. Как только клон создан, снапшот, на котором он базируется, не может быть удален до удаления клона
- Снапшот есть зафиксированный во времени образ данных в наборе объектов. Файловая система, клон или том не могут быть удалены, пока не удалены все их снапшоты
- Потомки: ZFS поддерживает иерархически-структурированные наборы объектов внутри датасета. Потомок зависит от существования родителя. Родитель также не может быть уничтожен до удаления всех потомков.

4.1 Инфраструктура DSL

Каждый набор объектов, представляемый в DSL — это набор данных. Датасет управляет статистикой расхода пространства, содержит информацию о размещении объектов и отслеживает любые внутренние зависимости.

Наборы данных группируются в месте с образованием каталогов наборов данных. Эти каталоги управляют группировкой наборов данных и свойствами, ассоциированными с такой группировкой. DSL-каталог всегда имеет только один активный датасет. Все остальные наборы под DSL-каталогом связаны с активным через снапшоты, клоны, или зависимости родитель-потомок.

Следующая картинка показывает инфраструктуру DSL и наглядно демонстрирует, каким образом объекты устанавливают взаимосвязи через наборы данных и каталоги DSL. Наверху в центре показана DSL директория верхнего уровня, а прямо под ней — активный датасет, представляющий файловую систему. От него отходит связанный список снапшотов, снятых в различные моменты времени. Каждая dataset-структура указывает на соответствующий набор объектов DMU, который содержит данные объектов. Слева от DSL каталога расположен ZAP-объект со списком зависимостей родитель-потомок. Справа находится ZAP-объект со свойствами датасетов в этом DSL-каталоге.



4.2 Детали реализации DSL

DSL реализован как набор объектов типа DMU_OST_META. Его часто называют Meta Object Set, или MOS. На каждый пул приходится строго один MOS, на который прямо указывает uber-блок. Он содержит описатели для всех файловых систем данного пула, а также их снапшотов и клонов.

В MOS существует один особенный объект, расположенный по индексу 1 в массиве dnodes. Он называется объектным каталогом, и все остальные объекты в MOS могут быть достигнуты по набору ссылок, начинающемуся в этом каталоге. Логически это ZAP-объект, содержащий три атрибута в NV-списке: "root_datadset", "config" и "sync_bplist".

- "root_dataset": атрибут содержит 64-битный объектный номер корневого DSL-каталога (специальный объект типа DMU_OT_DSL_DIR, содержащий ссылки на все датасеты верхнего уровня для данного пула)
- "config": атрибут хранит номер объекта типа DMU_OT_PACKED_NVLIST, содержащего конфигурационные параметры пула в XDR-кодированном NV-списке. Его содержание аналогично списку имя/значение, описанному в пункте 1.3
- "sync_bplist": хранит номер объекта типа DMU_OT_SYNC_BPLIST, который вмещает список указателей на блоки, которые необходимо удалить на следующей транзакции

4.3 Устройство датасета

Датасет хранится как объект типа DMU_OT_DSL_DATASET. В объектах этого типа используется bonus-буфер в dinode_phys_t для хранения структуры

dsl_dataset_phys_t (определена в /usr/src/uts/common/fs/zfs/sys/dsl_dataset.h):

```
typedef struct dsl_dataset_phys {
      uint64_t ds_dir_obj;
      uint64_t ds_prev_snap_obj;
      uint64_t ds_prev_snap_txg;
      uint64_t ds_next_snap_obj;
      uint64_t ds_snapnames_zapobj; /* номер ZAP-объекта со списком имен
                                     * снапшотов; 0, если данный датасет
                                     * сам снапшот */
      uint64_t ds_num_children;
      uint64_t ds_creation_time;    /* время создания в UTC-формате */
      uint64_t ds_creation_txg;
      uint64_t ds_deadlist_obj;
      uint64_t ds_used_bytes;
      uint64_t ds_compressed_bytes;
      uint64_t ds_uncompressed_bytes;
      uint64_t ds_unique_bytes;     /* количество уникальных байт;
                                     * актуально только для снапшотов и,
                                     * по сути, содержит размер */
      /*
       * The ds_fsid_guid is a 56-bit ID that can change to avoid
       * collisions.  The ds_guid is a 64-bit ID that will never
       * change, so there is a small probability that it will collide.
       */
      uint64_t ds_fsid_guid;
      uint64_t ds_guid;
      uint64_t ds_flags;
      blkptr_t ds_bp;
      uint64_t ds_pad[8];           /* pad out to 320 bytes for good measure */
} dsl_dataset_phys_t;
```

ds_dir_obj

Номер объекта для родительского DSL-каталога.

ds_prev_snap_obj

Если датасет представляет файловую систему, том или клон, это поле содержит номер объекта последнего снапшота (или 0 если снапшотов нет). Если сам датасет есть снапшот — здесь лежит номер предыдущего снапшота или 0 — если такового не имеется.

ds_prev_snap_txg

Номер группы транзакций, во время которой был сделан предыдущий снапшот.

ds_next_snap_obj

Используется только в снапшотах и содержит номер объекта для последнего по времени снапшота.

ds_snapnames_zapobj

Содержит номер ZAP-объекта, в котором хранятся NV-пары для каждого снапшота в данном датасете. Каждая пара содержит имя снапшота и номер ассоциированного с ним DSL-dataset объекта.

ds_num_children

Вегда 0, если не снапшот. Иначе это количество ссылок на данный снапшот: 1 из следующего снапшота или активного датасета, если такого нет, плюс по одной из каждого клона, происходящего из этого снапшота. По сути — количество потомков.

ds_creation_time

Время создания датасета.

ds_creation_txg

Номер группы транзакций, в рамках которой был создан датасет.

ds_deadlist_obj

Номер объекта для списка указателей, блоки которых были удалены со времени последнего снапшота.

ds_used_bytes

Количество байт, используемых набором объектов этого датасета.

ds_compressed_bytes

Количество сжатых байт, используемых набором объектов этого датасета.

ds_uncompressed_bytes

Количество разжатых байт, используемых набором объектов этого датасета.

ds_unique_bytes

На момент создания снапшота его содержимое идентично активной копии данных. По мере внесения изменений в файловую систему количество уникальных блоков в данном снапшоте растет — увеличивается это поле. Оно равно 0 для ФС, клонов и томов.

ds_fsid_guid

Уникальный идентификатор среди смонтированных в данный момент датасетов. Может меняться при его последующих открытиях.

ds_guid

Глобальный идентификатор датасета. Не меняется на протяжении его жизненного цикла.

ds_flags

Флаги. Используется для маркировки процесса восстановления датасета программой 'zfs restore'.

ds_bp

Указатель на блок, содержащий набор объектов датасета.

4.4 Устройство DSL-каталога

Объект DSL-каталога содержит структуру dsl_dir_phys_t (см. /usr/src/uts/common/fs/zfs/sys/dsl_dir.h) в своем bonus-буфере.

```
typedef struct dsl_dir_phys {
      uint64_t dd_creation_time;    /* не используется */
      uint64_t dd_head_dataset_obj;
      uint64_t dd_parent_obj;
      uint64_t dd_clone_parent_obj;
      uint64_t dd_child_dir_zapobj;

      /*
       * how much space our children are accounting for; for leaf
       * datasets, == physical space used by fs + snaps
       */
      uint64_t dd_used_bytes;
      uint64_t dd_compressed_bytes;
      uint64_t dd_uncompressed_bytes;

      /* Administrative quota setting */
      uint64_t dd_quota;

      /* Administrative reservation setting */
      uint64_t dd_reserved;
      uint64_t dd_props_zapobj;
      uint64_t dd_deleg_zapobj;     /* dataset delegation permissions */
      uint64_t dd_pad[20];          /* pad out to 256 bytes for good measure */
} dsl_dir_phys_t;
```

dd_head_dataset_obj
Номер объекта активного датасета.

dd_parent_obj
Номер объекта родительского DSL-каталога.

dd_clone_parent_obj
Для клонов — номер объекта исходного снапшота.

dd_child_dir_zapobj
Номер ZAP-объекта с NV-парами для каждого потомка этого DSL-каталога.

dd_used_bytes, **dd_compressed_bytes** и **dd_uncompressed_bytes**
Количество использованных, сжатых и разжатых байт, для всех датасетов этого каталога: включая снапшоты и дочерние датасеты.

dd_quota
Назначенные квоты. Если таковых нет, здесь хранятся естественные ограничения для датасетов этого DSL-каталога (определяемые, например, размером пула).

dd_reserved
Количество зарезервированного пространства для датасетов этого каталога

dd_props_zapobj

Номер ZAP-объекта со свойствами всех датасетов этого DSL-каталога. Содержит только не наследуемые или локально установленные свойства.

Свойство	**Определение**	**Значения**
aclinherit	Управляет порядком наследования для датасетов	discard = 0 noallow = 1 passtrhrough = 3 secure = 4 (default)
aclmode	Контролирует chmod и манеру создания файлов/ каталогов в датасете	discard = 0 groupmask = 2 (default) passtrhrough = 3
atime	Управляет модификацией времени доступа в датасете	off = 0 on = 1 (default)
checksum	Подсчет контрольной суммы для всех датасетов в DSL-каталоге	off = 0 on = 1 (default)
compressio n	Компрессия данных для всех датасетов DSL-каталога	off = 0 (default) on = 1
devices	Определяет, могут ли открываться файлы устройств на датасете	devices = 0 nodevices = 1 (default)
exec	Определяет, могут ли исполняться файлы на датасете	exec = 1(default) noexec = 0
mountpoint	Путь к точке монтирования датасетов DSL-каталога	строка
quota	Квоты по пространству для всех датасетов DSL-каталога	размер квот в байтах, или 0 при отсутствии квот (default)
readonly	Только для чтения	readonly = 1 readwrite = 0 (default)
recordsize	Размер блока для всех объектов в пределах датасетов данного DSL-каталога	байты
reservation	Количество зарезервированного пространства для данного DSL-каталога, включая все дочерние датасеты и DSL-каталоги	байты
setuid	Определяет,может ли быть представлен на датасете бит set-UID	setuid = 1 (default) nosetuid = 0
sharenfs	Определяет, можгут ли датасеты Dsl-каталога быть расшарены через NFS	строка — любые доступные опции NFS
snapdir	Определяет, будет ли скрыта директория .zfs	hidden = 0 vivble = 1 (default)
volblocksize	Для томов — размер блока задается в момент создания тома	от 512 байт до 128 kb, по умолчанию — 8 kb
volsize	Размер тома	байты
zoned	Определяет, управляется ли датасет через локальную зону	on = 1 off = 0 (default)

Глава 5. ZAP

ZAP — ZFS Attribute Processor — это модуль, расположенный над DMU и управляющий ZAP-объектами. ZAP-объект есть DMU-объект, предназначенный для хранения атрибутов в виде NV-пар. Порция name атрибута есть заканчивающаяся нулем строка длиной до 256 байт. Порция value атрибута — массив целых чисел, чей размер ограничен только размером блока данных ZAP. ZAP-объекты используются для хранения свойств датасета, навигации по файловой системе, хранения параметров пула и других целей. ZAP-объекты бывают следующих типов:

- DMU_OT_OBJECT_DIRECTORY
- DMU_OT_DSL_DIR_CHILD_MAP
- DMU_OT_DSL_DS_SNAP_MAP
- DMU_OT_DSL_PROPS
- DMU_OT_DIRECTORY_CONTENTS
- DMU_OT_MASTER_NODE
- DMU_OT_DELETE_QUEUE
- DMU_OT_ZVOL_PROP

ZAP-объекты могут существать в двух форматах — microzap и fatzap объекты. Microzap — это облегченная версия fatzap, обеспечивают простой и быстрый механизм поиска в небольшом количестве элементов атрибутов (attribute entries). Fatzap больше подходят для ZAP-объектов, содержащих множество элементов атрибутов.

ZFS будет использовать microzap при соблюдении всех перечисленных ниже условий, иначе используется fatzap:

- все NV-пары умещаются в один блок; максимальный размер блока ZFS равен 128 kb, и он может содержать до 2047 элементов microzap
- порция value всех атрибутов содержит значение типа uint64_t
- порция name всех атрибутов хранит не более 50 байт (включая завершающий ноль)

Первые 64 бита тела любого ZAP-объекта используются для определения его типа. Возможные типы перечислены в /usr/src/uts/common/fs/zfs/sys/zap_impl.h:

Метка	Определение	Значение
ZBT_LEAF	Блок используется для fatzap. Этот идентификатор используется для всех блоков fatzap, кроме первого	((1ULL << 63) + 0)
ZBT_HEADER	Используется для fatzap. Этим идентификатором помечается первый блок в fatzap	((1ULL << 63) + 1)
ZBT_MICRO	Этот блок используется для элементов microzap	((1ULL << 63) + 3)

5.1 Microzap

Microzap реализует простой механизм хранения небольшого количества атрибутов. Он полностью умещается в одном блоке, содержащем массив microzap-элементов. Каждый атрибут представлен одним таким элементом. См.

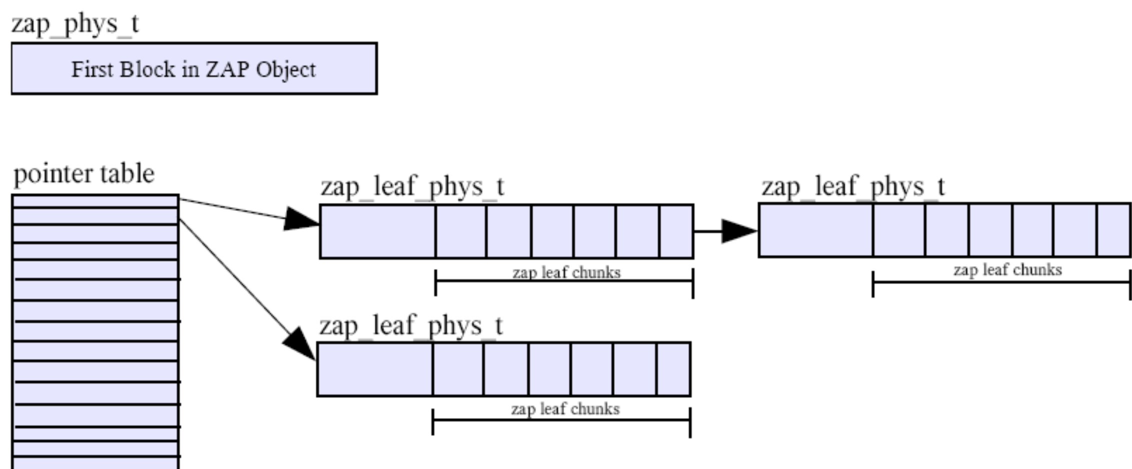
/usr/src/uts/common/fs/zfs/sys/zap_impl.h:

```
/* microzap-элемент */
typedef struct mzap_ent_phys {
      uint64_t mze_value;     /* Значение атрибута */
      uint32_t mze_cd;        /* Используется для различения объектов
                               * с одинаковым хэшем. При отсутствии
                               * коллизий равно 0 */
      uint16_t mze_pad;
      char mze_name[MZAP_NAME_LEN]; /* Имя атрибута */
} mzap_ent_phys_t;

/* Собственно microzap */
typedef struct mzap_phys {
      uint64_t mz_block_type; /* Тип блока --  ZBT_MICRO */
      uint64_t mz_salt;       /* Индикатор типа хэш-функции (если они
                               * различны для разных ZAP-объектов) */
      uint64_t mz_pad[6];
      mzap_ent_phys_t mz_chunk[1];
} mzap_phys_t;
```

5.2 Fat ZAP

Fatzap реализует гибкую архитектуру для хранения большого количества атрибутов, а также атрибутов с длинными именами и комплексными значениями.

Элементы fatzap-объекта выстраиваются в порядке, определяемом значением хэша имени атрибута, который используется в качестве индекса в таблице указателей.



Первый блок fatzap-объекта занимает 128-kb структура zap_phys_t (/usr/src/uts/common/fs/zfs/sys/zap_impl.h):

```
typedef struct zap_phys {
      uint64_t zap_block_type;      /* Тип блока = ZBT_HEADER */
      uint64_t zap_magic;           /* ZAP_MAGIC */

      struct zap_table_phys {       /* Описывается положение таблицы
```

```
                                * указателей */
            uint64_t zt_blk;        /* Номер начального блока */
            uint64_t zt_numblks;    /* Количество блоков */
            uint64_t zt_shift;      /* Количество битов в значении хэша,
                                * которые используются как индекс в
                                * таблице указателей. Если таблица
                                * встроена -- всегда 13.  */
            uint64_t zt_nextblk;    /* Используются при изменении
                                * размера таблицы */
            uint64_t zt_blks_copied;      /* То же */
      } zap_ptrtbl;

      uint64_t zap_freeblk;
      uint64_t zap_num_leafs;
      uint64_t zap_num_entries;
      uint64_t zap_salt;
} zap_phys_t;
```

В зависимости от размера таблицы указателей она может содержаться либо в остаточной части первого блока, либо будет адресоваться структурой zap_ptrtbl. В первом случае таблица достижима через макросы:

```
/*
 * Встроенная таблица указателей начинается на половине блока:
 * block size / entry size (2^3) / 2
 */
#define     ZAP_EMBEDDED_PTRTBL_SHIFT(zap) (FZAP_BLOCK_SHIFT(zap) - 3 – 1)

/*
 * Через этот макрос мы обращаемся к отдельным элементам таблицы
 * по их индексу
 */
#define     ZAP_EMBEDDED_PTRTBL_ENT(zap, idx) \
      ((uint64_t *)(zap)->zap_f.zap_phys) \
      [(idx) + (1<<ZAP_EMBEDDED_PTRTBL_SHIFT(zap))]
```

Проясним назначение некоторых полей структуры zap_phys_t.

zap_freeblk

Номер первого доступного ZAP-блока, который может быть использован для размещения нового zap_leaf.

zap_num_leafs

Количество структур zap_leaf_phys_t, содержащихся в данном ZAP-объекте.

zap_salt

Индикатор типа хэш-функции (если они различны для разных ZAP-объектов).

zap_num_entries

Количество атрибутов, хранимых в данном ZAP-объекте.

Таблица указателей — это хэш-таблица, которая использует chaining-метод для обработки коллизий. Каждый хэш-пакет содержит 64-битный ID блока нулевого

уровня (см. главу 3 о ID блоков) первого элемента цепочки, хэшируемой здесь. Хэш-пакет элемента определяется с использованием нескольких первых бит 64-битного хэша ZAP-элемента, вычисляемого на основе имени атрибута. Это значение применяется для индексирования в таблице указателей, называется префиксом и хранится в старших битах 64-битного вычисленного хеша. Количество этих бит определяется значением zt_shift.

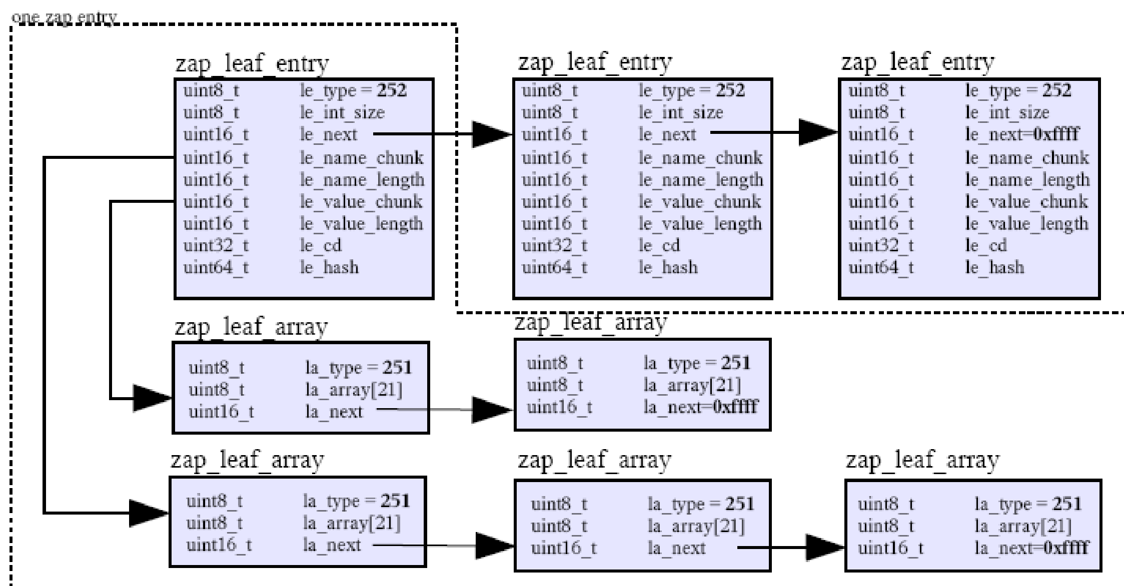
zap_leaf_phys_t это структура, адресуемые по указателям из таблицы. Из-за возможности возникновения коллизий эти структуры приходится собирать в связанный список. Каждая структура состоит из заголовка, хэш-таблицы и некоторого количества кусков (см. /usr/src/uts/common/fs/zfs/sys/zap_leaf.h):

```
/* ZAP-лист */
typedef struct zap_leaf_phys {
      struct zap_leaf_header {
            uint64_t lh_block_type; /* Тип блока -- ZBT_LEAF */
            uint64_t lh_pad1;
            uint64_t lh_prefix;     /* хэш-префикс этого листа */
            uint32_t lh_magic;      /* ZAP_LEAF_MAGIC */
            uint16_t lh_nfree;      /* количество свободных кусков */
            uint16_t lh_nentries;   /* количество элементов */
            uint16_t lh_prefix_len; /* количество битов, используемых
                                     * для ID */
            uint16_t lh_freelist;   /* заголовок списка свободных
                                     * кусков */
            uint8_t lh_pad2[12];
      } l_hdr;                      /* 2 24-byte chunks */

      uint16_t l_hash[1];           /* хэш-таблица */
} zap_leaf_phys_t;

/* Типы кусков */
typedef enum zap_chunk_type {
      ZAP_CHUNK_FREE = 253,
      ZAP_CHUNK_ENTRY = 252,
      ZAP_CHUNK_ARRAY = 251,
      ZAP_CHUNK_TYPE_MAX = 250
} zap_chunk_type_t;

/* Структуры кусков */
typedef union zap_leaf_chunk {
      struct zap_leaf_entry {
            uint8_t le_type;              /* всегда ZAP_CHUNK_ENTRY */
            uint8_t le_int_size;          /* size of ints */
            uint16_t le_next;             /* следующий элемент в хэш-цепочке */
            uint16_t le_name_chunk;       /* первый кусок имени атрибута */
            uint16_t le_name_length;      /* длина имени */
            uint16_t le_value_chunk;      /* первый кусок значения атрибута */
            uint16_t le_value_length;     /* длина значения */
            uint32_t le_cd;               /* collision differentiator */
            uint64_t le_hash;             /* хэш имени */
      } l_entry;
      struct zap_leaf_array {
            uint8_t la_type;              /* всегда ZAP_CHUNK_ARRAY */
            uint8_t la_array[ZAP_LEAF_ARRAY_BYTES];
            uint16_t la_next;             /* индекс следующего куска или
                                           * CHAIN_END */
      } l_array;
```

```
        struct zap_leaf_free {
              uint8_t lf_type;              /* всегда ZAP_CHUNK_FREE */
              uint8_t lf_pad[ZAP_LEAF_ARRAY_BYTES];
              uint16_t lf_next;             /* индекс следующего свободного
                                             * списка или CHAIN_END */
        } l_free;
} zap_leaf_chunk_t;
```

Заголовок ZAP-листа хранится в структуре zap_leaf_header. Каждый лист (или цепочка листов) хранит ZAP-элементы, первые ls_prefix_len бит которых содержат значение хэша lh_prefix. За заголовком следует 8-kb хэш-таблица листа. Элементы этой таблицы адресуют куски данных типа zap_leaf_entry. 12 бит хэша, следующие за lh_prefix, используются для индексации в этой таблице. Каждый элемент в таблице содержит 16-битное целое — индекс массиве zap_leaf_chunk.

Каждый лист содержит массив кусков. Они бывают трех типов: zap_leaf_entry, zap_leaf_array и zap_leaf_free. Каждый атрибут представлен некоторым количеством этих кусков: одним zap_leaf_entry и несколькими zap_leaf_array.



zap_leaf_entry

Элементы хэш-таблицы листа указывают на куски этого типа, а они, в свою очередь, указывают на куски типа zap_leaf_array, что хранят собственно NV-пару. Поле le_next указывает на следующий элемент в цепочки кусков. Цепочка образуется при возникновении коллизий в хэш-таблице.

zap_leaf_array

Кусок этого типа хранит часть (сейчас — 21 байт) имени или значения атрибута. Значения типа integer всегда хранятся в big endian формате — не зависимо от архитектуры.

Глава 6. ZPL

ZPL, ZFS POSIX Layer, позволяет DMU-объектам выглядеть как файловая система POSIX.

ZPL представляет файловую систему как набор объектов типа DMU_OST_ZFS. Все снапшоты, клоны и файловые системы реализованы как наборы объектов этого типа. ZPL использует хорошо известный формат для объединения объектов в наборы; именно его мы и рассмотрим в этом разделе.

6.1 Формат файловой системы ZPL

Набор объектов ZPL имеет один объект с фиксированным местоположением и номером. Он называется master node и всегда имеет номер 1. Maser node представляет собой ZAP-объект, содержащий 3 атрибута: DELETE_QUEUE, VERSION и ROOT.

DELETE_QUEUE

Содержит 64-битный номер для объекта очереди удаления. Очередь удаления поддерживает список удалений, которые выполнялись на момент сбоя файловой системы. Перед следующим монтированием ФС все удаления из этой очереди выполняются.

VERSION

Версия ZPL-формата. На момент написания статьи здесь стоял номер 3.

ROOT

Содержит 64-битный номер для объекта корневого каталога.

6.2 Каталоги и их обход

Каталоги файловой системы реализованы в виде ZAP-объектов типа DMU_OT_DIRECTORY. Каждый такой объект содержит пары имен и номеров объектов для элементов каталога. Все объекты файловой системы содержат структуру znode_phys_t в bonus-буфере (/usr/uts/common/fs/zfs/sys/zfs_znode.h) — данные для вызова stat().

```
/*
 * Это постоянная часть znode. Она хранится в bonus-буфере файла.
 * Короткие символические ссылки хранятся там же
 */
typedef struct znode_phys {
      uint64_t zp_atime[2];   /* время последнего доступа */
      uint64_t zp_mtime[2];   /* время последней изменения модификации
                               * файла */
      uint64_t zp_ctime[2];   /* время последнего изменения */
      uint64_t zp_crtime[2];  /* время создания */
      uint64_t zp_gen;        /* номер группы транзакций на момент
                               * создания */
      uint64_t zp_mode;       /* mode-биты */
      uint64_t zp_size;       /* размер файла */
      uint64_t zp_parent;     /* родительский каталог ("..") */
      uint64_t zp_links;      /* количество ссылок на файл */
```

```
      uint64_t zp_xattr;      /* DMU-объект для xattrs */
      uint64_t zp_rdev;       /* dev_t для VBLK и VCHR файлов */
      uint64_t zp_flags;      /* флаги */
      uint64_t zp_uid;        /* владелец */
      uint64_t zp_gid;        /* группа */
      uint64_t zp_pad[4];     /* зарезервировано */
      zfs_znode_acl_t zp_acl; /* ACL */
} znode_phys_t;
```

Биты 13-16 поля zp_mode используются для определения типа файла. Возможные значения:

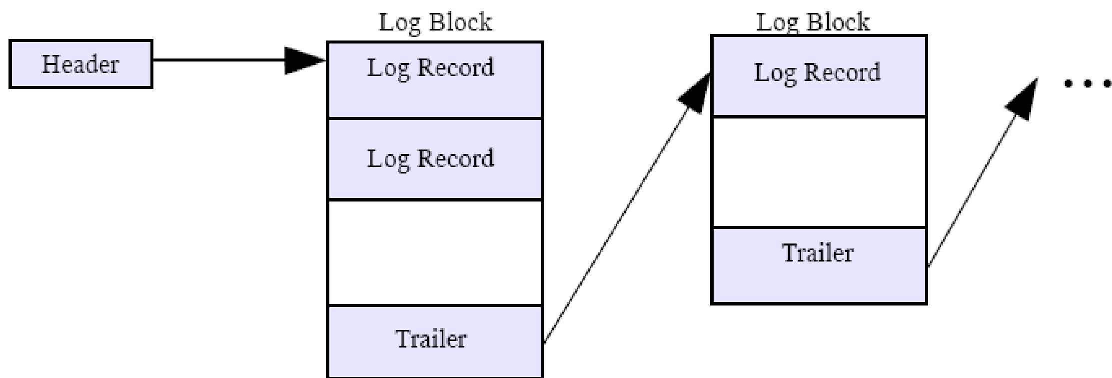
Тип	Определение	Значение битов 13-16
S_IFIFO	FIFO-канал	0x1
S_IFCHR	Символьное устройство	0x2
S_IFDIR	Каталог	0x4
S_IFBLK	Блочное устройство	0x6
S_IFREG	Регулярный файл	0x8
S_IFLNK	Символическая ссылка	0xA
S_IFSOCK	Сокет	0xC
S_IFDOOR	"Дверь"	0xD
S_IFPORT	Порт события	0xE

Поле zp_xattr содержит номер ZAP-объекта, являющегося скрытым каталогом атрибутов. Он обрабатывается аналогично нормальному каталогу, но скрыт от приложений и требует туннеля в файл через openat() для открытия.

Глава 7. ZIL —журнал ZFS

ZIL, ZFS Intent Log, сохраняет информацию об изменяющих ФС транзакциях в объеме, достаточном для их завершения или атомарного отката в случае сбоя. Они накапливаются в памяти до тех пор, пока группа транзакций DMU не зафиксирует их в стабильном пуле, или пока они не будут сброшены в журнал (также в пуле) в результате fsync().

Существует один ZIL на файловую систему. На диске он состоит из трех частей: заголовка, ZIL-блоков и ZIL-записей. Блоки хранят множество записей и объединяются в цепочки. Каждый ZIL-блок содержит указатель (трейлер, типа blkptr_t) на следующий блок в цепи. Блоки могут быть разного размера. ZIL-заголовок указывает на первый блок в цепи; фиксированного местоположения на пуле ZIL не имеет.



Структура заголовка определена в /usr/src/uts/common/fs/zfs/sys/zil.h:

```
typedef struct zil_header {
      uint64_t zh_claim_txg;  /* номер группы транзакций (txg) */
      uint64_t zh_replay_seq; /* highest replayed sequence number */
      blkptr_t zh_log;        /* указатель на следующий блок в цепи */
      uint64_t zh_claim_seq;  /* highest claimed sequence number */
      uint64_t zh_pad[5];
} zil_header_t;

typedef struct zil_trailer {
      uint64_t zit_pad;
      blkptr_t zit_next_blk;        /* следующий блок в цепи */
      uint64_t zit_nused;           /* использовано байт в log-блоке */
      zio_block_tail_t zit_bt;      /* block trailer (хвост блока) */
} zil_trailer_t;
```

Каждый набор объектов имеет свой собственный журнал. Заголовок для журнала набора N хранится в N-ом объекте SPA's intent_log objset'a. Заголовок указывает на цепочку журнальных блоков, каждый из которых содержит журнальные записи (т.е. транзакции), следующие за трейлером. Формат записей зависит от их типа, однако все они начинаются с однотипной структуры, с помощью которой определяется тип, длина и txg.

Log block trailer — структура в конце заголовка журнала и каждого журнального блока. Поле zit_bt содержит контрольную сумму, которая в журнале равна номеру последовательности данного журнального блока. Сумма = 0 не верна.

Структура полей log-записи была тщательно выверена — поэтому они имеют одинаковый размер и выравнивание как на sparc, так и на intel архитектурах. Любая log-запись начинается с общей секции (структура lr_t), за которой следуют специфичные для конкретного типа транзакции данные.

```
typedef struct {              /* общий заголовок log-записей */
      uint64_t    lrc_txtype; /* тип транзакции */
      uint64_t    lrc_reclen; /* длина записи */
      uint64_t    lrc_txg;    /* номер группы транзакций DMU */
      uint64_t    lrc_seq;    /* номер последовательности */
```

```
} lr_t;
```

ZIL определяет следующие типы транзакций (/usr/src/uts/common/fs/zfs/sys/zil.h):

```
#define     TX_CREATE   1           /* Создать файл */
#define     TX_MKDIR    2           /* Создать каталог */
#define     TX_MKXATTR  3           /* Создать XATTR-каталог */
#define     TX_SYMLINK  4           /* Создать символическую ссылку */
#define     TX_REMOVE   5           /* Удалить файл */
#define     TX_RMDIR    6           /* Удалить каталог */
#define     TX_LINK     7           /* Создать жесткую ссылку */
#define     TX_RENAME   8           /* Переименовать файл */
#define     TX_WRITE    9           /* Записать файл */
#define     TX_TRUNCATE 10          /* Урезать файл */
#define     TX_SETATTR  11          /* Установить атрибуты файла */
#define     TX_ACL      12          /* Установить ACL */
#define     TX_MAX_TYPE 13
```

Структуры соответствующих log-записей определены в /usr/src/uts/common/fs/zfs/sys/zil.h.

Глава 8. ZVOL

ZVOL (ZFS Volume) обеспечивает механизм для создания логических томов. Тома ZFS экспортируются как блочные устройства. Внутри ZFS тома представляются набором объектов типа DMU_OST_ZVOL и имеют очень простой формат — состоят всего лишь из двух объектов: типа DMU_OT_ZVOL_PROP (свойства тома) и DMU_OT_ZVOL (данные).

Часть 2. Алгоритмы

Подведем небольшой итог: уровень ZPL занимается преобразованием имен файлов в номера объектов, DMU получает из объектов их DVA, а SPA по DVA определяет непосредственное положение данных файла на конкретном носителе.

Учет и выделение блоков

Карты пространства

Любая файловая система должна отслеживать 2 важные вещи: где находятся ее данные, и где — свободное место.

В принципе, отслеживание свободного места на так уж необходимо: каждый блок или выделен, или свободен, и free-карта может быть вычислена путем вычитания из карты всего диска занятых блоков, а allocated-карта может быть

получена обходом всей ФС от корневого каталога.

Однако на практике решение проблемы таким способом не реально, т.к. займет слишком много времени для файловой системы сколь нибудь значительного размера. Для эффективного выделения и освобождения блоков ФС должна отслеживать свободное пространство. Посмотрим, как эта задача решается в традиционных ФС и какая схема применена в ZFS.

Битовые карты

Наиболее распространенный способ представления свободного пространства — битовые карты. Bitmap — есть просто массив битов, где N-й бит показывает состояние N-го блока. При размере блока в 4 kb накладные расходы на хранение битовых карт пренебрежимо малы (1/4096*8 =0.003%, 1 байт на 8 блоков). Для ФС объемом 1Gb битовые карты занимают всего 32 kb — такой кусок данных легко помещается в памяти и может быть быстро просканирован. Для 1 Tb это уже 32 Mb — все еще помещается в памяти, но обход будет занимать достаточно длительное время. Для 1 Pb битовые карты займут 32 Gb — неприемлемо много для большинства машин.

Кроме того, битовые карты не масштабируются.

Очевидный выход — разбить карту на несколько частей. К примеру, для ФС в 1 Pb свободное пространство может быть представлено миллионом 32-kb битовых карт. А обобщающая информация (миллион integers — сколько места в каждой битовой карте) помещается в памяти. Так достаточно просто найти карту с необходимым количеством места, и быстро ее просканировать.

Однако существует фундаментальная проблема — битовые карты должны обновляться не только при выделении, но и при освобождении блоков. Простая команда типа "rm -rf" может вызвать освобождение множества блоков по всему диску. В нашем примере, удаление 4 Gb данных (миллиона 4-kb блоков) потребует, в худшем случае, чтения, модификации и фиксации каждой из миллиона битовых карт. 2 миллиона дисковых запросов для освобождения каких-то 4 Gb — совершенно неприемлемо.

Это главная причина плохой масштабируемости битовых карт — никакая схема кэширования не справится с хаотичным удалением сколь-нибудь значительного количества данных и все равно потребует множества запросов к диску.

Другое распространенное решение — сбалансированные деревья внешнего поиска, B-деревья экстентов. Выделение блоков с использованием B-деревьев гораздо более эффективно, однако, сталкиваясь с хаотичным освобождением, деревья остаются столь же слабы, как и битовые карты.

Отложенные освобождения

Один из путей избежать этих недостатков — откладывать обновление битовых карт или B-деревьев и хранить список недавно освобожденных блоков. При достижении этим списком некоторого размера, указанные в нем освобождения вносятся в битовые карты или деревья. Не идеально, но должно помочь.

Карты пространства: log-structured списки освобождений

Вернемся к давней идее журнально-структурированных файловых систем: если вместо того, что бы каждый раз вносить содержимое журнала транзакций в файловую систему, мы заставим сам журнал быть файловой системой?

Тот же вопрос может быть поставлен в отношении списка отложенных освобождений: если вместо того, что бы периодически фиксировать отложенные освобождения в битовых картах или B-деревьях, мы заставим список освобождений быть представлением свободного пространства?

Именно эта схема и применена в ZFS. ZFS разбивает пространство на каждом виртуальном устройстве на несколько сотен регионов, называемых metaslabs. Каждый метаслаб имеет собственную карту пространства. Эта карта есть просто журнал выделений и освобождений, записанных в порядке внесения. Т.о., случайные освобождения здесь столь же эффективны, как и последовательные — требуется только удаление одного экстента из space map объекта. Аналогично, выделение экстента — просто его добавление в space map объект (естественно, с установленным битом, показывающим, что это не освобождение).

Когда ZFS решает выделить блок на конкретном метаслабе, первым делом она читает с диска его карту пространства и проигрывает все внесенные туда размещения и освобождения, выстраивая в памяти AVL-дерево свободных экстентов, отсортированное по смещению. AVL-дерево (самая эффективная из существующих древовидных структур) очень хорошо приспособлено для быстрого поиска больших участков свободного пространства.

Такие карты пространства имеют несколько замечательных свойств:

- Они не требуют инициализации — наличие карты без элементов говорит только о том, что ни освобождений, ни размещений еще не было — т.е. все пространство свободно.
- Они великолепно масштабируются.
- Они практически не имеют недостатков — одинаково эффективны как для освобождения, так и для размещения блоков.
- Их эффективность не зависит от степени заполненности пула.

Наконец нельзя не заметить, что когда карта пространства полностью заполнена, она состоит из единственного экстента. Следовательно, с ростом степени заполненности пула карта пространства только уменьшается, выдавая дополнительное свободное место под актуальные данные.

Выделение блоков в ZFS

Политика выделения блоков — центральная составляющая любой файловой системы. Она затрагивает не только производительность, но также административную модель (например, stripe-конфигурацию) и даже некоторые возможности ядра ФС — семантику транзакций, сжатие, разделение блоков между снапшотами. Тем более важно сделать выделение блоков эффективным.

В ZFS политика размещения блоков состоит из трех компонентов:

1. Выбор устройства
2. Выбор метаслаба
3. Выбор блока

По дизайну эти три политики независимы и расширяемы — т.е. могут быть пересмотрены без изменения дискового формата, что дает известную гибкость в будущем развитии ZFS.

1. Выбор устройства (aka dinamic striping)

Первая задача — выбор устройства. Цель — распределить нагрузку между всеми устройствами пула и получить максимальную пропускную способность без введения лишних сущностей типа stripe-массивов. Добавляя новый диск вы автоматически увеличиваете производительность.

Выбрать устройство можно разными способами. Любая политика здесь работоспособна, включая просто случайный выбор, однако есть несколько практических соображений:

- Если устройство было добавлено в пул, оно будет относительно пустым. Для устранения этой неуравновешенности политика ZFS начинает предпочитать для выделения блоков именно такое устройство. Это решение позволяет более равномерно нагружать устройства в пуле и делает механизм выделения блоков более предсказуемым и масштабируемым.
- При прочих равных, круговое чередование дисков является хорошим решением. Однако очень важно правильно выбрать гранулярность (объем данных, после вывода которого происходит переключение на следующее устройство). Когда она слишком велика (к примеру, 1 Gb), при последовательной записи чаще всего будет нагружено только одно устройство. Если гранулярность мала (1 блок), мы теряем преимущества использования встроенного буфера диска. Опытом было установлено, что переключение на следующее устройство после вывода каждых 512 Kb данных является хорошим решением для современного поколения дисковых устройств.
- Для блоков журнала наилучшей политикой было бы переключение между устройствами каждый раз при записи такого блока, т.к. эти блоки очень коротко живущие и мы не ожидаем, что когда нибудь станем читать их.

Поэтому идеально было бы оптимизировать именно производительность записи ZIL-блоков.

- Вообще говоря, возможно мы захотим иметь различные политики выбора устройства для разных типов данных: большие последовательные, маленькие случайные, мимолетные (как записи журнальных блоков) и dnodes (группировка для пространственной плотности может быть полезна). Это благодатная почва для исследований.
- Если одно из устройств замедлилось или разрушено по некоторым причинам, оно должно быть исключено из выбора. Работа над этим ведется.

2. Выбор метаслаба

ZFS разбивает каждое устройство на несколько сотен регионов, называемых метаслабами. Выбрав устройство, ZFS становится перед выбором оптимального метаслаба. Интуитивно понятно, что наилучшем будет наиболее свободный метаслаб, однако есть и еще несколько факторов:

- Современные диски имеют однородную битовую плотность и постоянную угловую скорость. Следовательно, области на внешней кромке диска будут обрабатываться быстрее, нежели на внутренней, диаметр которой обычно в 2 раза меньше. По сути, ZFS выбирает наиболее быстрый и свободный метаслаб.
- На относительно пустом пуле ZFS старается использовать сначала наиболее быстрые области дисков, что не только увеличивает производительность, но и противодействует фрагментации.

Все эти условия перебираются в функции metaslab_weight() (см. metaslab.c), которая назначает вес каждому метаслабу. Алгоритм выбора после вызова этой функции очень прост — всегда используется регион с наибольшим весом.

3. Выбор блока

В настоящее врем ZFS не производит какого-либо оптимизированного выбора блока в метаслабе — берется просто первый попавшийся. В будущем это решение будет пересмотрено, и для каждого типа рабочей нагрузки будет назначен свой алгоритм выбора блока. Код выбора блока заранее спроектирован с учетом этого расширения (см. space_map_ops в space_map.h).

Реализация снапшотов

Снапшоты — далеко не новая концепция, давно реализованная в нескольких файловых системах. Снапшот есть срез файловой системы в некоторый момент времени. Снапшоты ZFS предназначены для тех же целей, что и в других ФС: сделав backup-снапшот, вы получаете согласованную (persistent), статичную точку восстановления данных.

От прочих снапшоты ZFS отличает то, что они лишены практически всех

ограничений. Пользователь может создавать их в любом количестве, с любой частотой, и именовать по своему усмотрению. Создание снапшота — операция неизменной длительности, а их наличие никак не влияет на производительность остальной файловой системы. Время удаления снапшота, как будет показано, пропорционально количеству измененных со времени его создания блоков.

Снапшоты ZFS реализованы на уровне DSL.

Окружение

ZFS реализована на диске в виде единого дерева блоков — с данными в листовых узлах и метаданными во внутренних (в основном это косвенные блоки, но есть также и структуры dnode_phys_t). При сбросе модифицированного блока данных на диск файловая система записывает его в новое место. Далее необходимо внести указатель на новое смещение в родительский узел этого блока — и он также пишется по новому смещению. Этот процесс продолжается вплоть до корня дерева, хранящегося по фиксированному смещению — он просто перезаписывается.

Структуры данных

Снапшот ZFS есть тип датасета, представляемого на диске структурой типа dsl_dataset_phys_t. Сами снапшоты поддерживаются через две структуры данных: поле blk_bitrh в blkptr_t, и список "мертвых" блоков, ассоциированных с каждой файловой системой и снапшотом (указатель ds_deadlist_obj в dsl_dataset_phys_t).

При обновлении указателя на блок в его родителе файловая система также записывает время этого обновления (в виде номера группы транзакций — txg), и модифицирует uber-блок.

"Мертвый" список содержит массив указателей на блоки, которые были актуальны в предыдущем снапшоте, однако удалены в текущем (или файловой системе).

Создание снапшота

В принципе, все, что мы должны сделать для создания снапшота — это сохранить старый uber-блок до его перезаписи, т.к. он указывает на актуальные, не модифицированные данные. Фактически же, мы сохраняем не uber-блок, а корень поддерева, который представляет файловую систему (структура типа objset_phys_t, или, вообще говоря, любой указатель на dsl_dataset_phys_t). Таким образом, каждая файловая система имеет свои собственные, независимые от других ФС снапшоты. Снапшоты каждой ФС отслеживаются в двусвязном списке (указатели ds_prev_snap_obj и ds_next_snap_obj), отсортированном по времени создания. Сама файловая система находится в хвосте списка. Снапшоты имеют выбираемые администратором имена, хранящиеся в каталого-подобной структуре, управляемой с помощью ZAP-объекта, адресуемого по указателю ds_snapnames_zapobj.

При создании снапшота его dead-лист инициализируется данными из мертвого списка файловой системы, а тот, в свою очередь, опустошается.

За создание снапшота отвечает функция dsl_dataset_snapshot_sync().

Освобождение блоков

При модификации или освобождении некоторого блока файловой системы DMU вызывает функцию dsl_dataset_block_born(), которая определяет, может ли ФС действительно освободить данный блок, или должна всего лишь разместить новую копию в другом месте. Блок освобождается только в случае отсутствия ссылок на него в каких-либо объектах ZFS, что определяется путем сравнения времени рождения блока (blk_birth) с временем создания самого последнего снапшота (ds_prev_snap_txg). Если блок был размещен после создания самого последнего снапшота, значит тот не может ссылаться на него, и блок может быть освобожден.

Если последний снапшот создан после размещения блока, он (блок) добавляется в dead-лист.

Удаление снапшота

За удаление снапшота отвечает функция dsl_dataset_destroy_sync(), которая должна определить список блоков на освобождение, а также подкорректировать мертвые списки.

ZFS проходит по мертвому списку следующего снапшота (или ФС) и сравнивает время рождения каждого блока с временем создания предыдущего снапшота. Если блок появился после предыдущего снапшота, он не освобождается, а добавляется в текущий мертвый список. Если блок был рожден после предыдущего снапшота, он может быть освобожден. Операция завершается заменой dead-листа следующего снапшота на мертвый список текущего. Далее снапшот удаляется из двусвязного списка и каталога имен снапшотов.

Выводы

ZFS очень необычная файловая система, и сравнивать ее с чем либо достаточно трудно. Первое, что бросается в глаза — исключительное удобство администрирования. По этому параметру, пожалуй, ZFS оставляет далеко позади все системы хранения данных предыдущего поколения, доступные для ОС семейства *NIX.

Выигрыш в производительности же удастся получить только на серверах с быстрыми CPU и большими объемами оперативной памяти. В документации к ZFS сказано, что минимальный объем RAM для ее использования равен 1Gb, однако по опыту можно утверждать, что она начинает опережать конкурентов лишь на компьютерах с памятью не менее 2 Gb — что не удивительно, т.к. ZFS ориентирована именно на такие мощности. Дело в том, что на современных

машинах самым узким местом в производительности файловой системы является сам диск, а именно — объем побочного ввода/вывода — то есть чтение и запись метаданных. На сокращение этого параметра и ориентированы все подсистемы ZFS, требующей, поэтому, достаточно большого дискового кэша. На быстрой многопроцессорной машине с памятью от 2Gb такие особенности, как сжатие данных и журнально-структурированные карты пространства начинают играть на руку производительности, позволяя ZFS оставить позади такие файловые системы, как ufs2, reiser4, и ext4.

Файловая система ZFS доступна для Sun Solaris 10 и OpenSolaris с 2005 года, для FreeBSD-7.0-RELEASE с февраля 2008 года.

Ситуация с ZFS в Linux довольно плачевна. Доступен лишь порт на FUSE, работающий нестабильно и достаточно медленно. О полноценном kernel-space драйвере мечтать не приходится — прежде всего из-за ограничений GPL.

Приложение 1. Версии ZFS

Version 1

Первая базовая версия ZFS, в компании Sun Microsystems появилась 31.10.2005. В течение следующих 6 месяцев внутреннего использования были внесены несколько серьезных изменения в дисковый формат, не отразившихся, однако, на номере версии. Первыми официальными релизами с ZFS вирсии 1 были:

- Solaris 10 Update 2
- Solaris Nevada Build 36

Более ранние неофициальные релизы не поддерживают эту версию ZFS, несмотря на то, что дисковый номер один и тот же — в официальную "единицу" внесены следующие изменения:

- 6389368: Fat ZAP стал использовать 16-kb блоки
- 6390677: проверка номера версии может вызвать модернизацию

Version 2

В этой версии поддерживаются т.н. "Ditto Blocks" — реплицированные метаданные. Благодаря древовидной дисковой структуре ZFS, неисправимая ошибка в листовом блоке может быть обработана без особых потерь, в то время как подобная же ошибка в метаданных пула может разрушить все хранилище. В версии 2 появилась возможность хранить до трех копий каждого блока — вне зависимости от избыточности пула. К примеру, на mirror-пуле наиболее критичные метаданные будут храниться в трех копиях на каждой стороне зеркала — всего 6 репликаций. Указатели на копии расположены в структуре blockrun_t. В дальнейшем эту возможность планируется использовать для репликации и пользовательских данных на базе датасетов (а не только пулов, как это делается сейчас).

Репликация метаданных была интегрирована в обновление за номером 6420698 и вышла со следующими релизами:

- Solaris Nevada Build 38
- Solaris 10 Update 2 (Build 09)

Version 3

Эта версия ZFS включает поддержку следующих возможностей:

- Резервирование дисков для горячей замены (фикс номер 6405966)
- RAID-Z двойной четности (RAID-Z2, фикс 6417978)
- Усовершенствованный подсчет потерь на RAID-Z (фикс 6288488)

ZFS версии 3 интегрирована в релизы:

- Solaris Nevada Build 42
- Solaris 10 Update 3 Build 3

Version 4

Введена одна новая возможность — история zpool. Состоит из двух изменений:

- 6529406: история изменений в подсистеме zpool, необходимая для дампа
- 6343741: история подкоманд утилиты zpool

Интегрирована в релиз Solaris Nevada Build 62 и Solaris 10 Update 4.

Version 5

Поддерживается gzip-компрессия для датасетов ZFS (фикс 6536606), релиз Solaris Nevada Build 62.

Version 6

Поддерживается свойство пула "bootfs". Изменения 4929890 (поддержка загрузки с ZFS для платформы x86) и 6479807 (свойства пула) интегрированы в релиз Solaris Nevada Build 62.

Во FreeBSD-7.0-RELEASE, ZFS которой идентифицирует себя как "версия 6", введен атрибут "copies", определяющий, сколько копий каждого блока данных будет храниться в пуле (для критичных метаданных эта возможность была введена еще в версии 2), однако в официальной документации сообщества OpenSolaris/ZFS об этом ничего не говорится.

Version 7

ZIL (ZFS Intent Log) удовлетворяет потребность некоторых приложений в синхронном сбросе данных в ФС, удерживая полную информацию о системном вызове в некоторой выделенной области пула, которому принадлежит файловая система. Фикс номер 6339640 добавляет возможность размещать ZIL на выделенных устройствах (например, NVRAM или отдельном диске). Интегрирована в Solaris Nevada Build 68.

Version 8

Поддержка делегирования администрирования пулов и файловых систем ZFS обычным пользователям — фикс 6349470, релиз Solaris Nevada Build 69.

Version 9

Внесены следующие новые возмжности:

- Дополнительно к командам установки квот и резервирования, эта версия поддерживает команды "zfs set refquota" и "zfs set refreservation", аналогичные уже имеющимся, но не затрагивающие нижележащие датасеты (снапшоты и клоны) в ограничении пространства (фикс 6431277)
- Изменения в порядке резервирования пространства при создании эмулируемых томов ZFS (фикс 6483677)
- Поддержка CIFS-сервера OpenSolaris (фикс 6617183)

Релиз Solaris Nevada Community Edition, Build 77.

Version 10

В пул может быть добавлено кэш-устройство, составляющее дополнительный уровень кэширования между памятью и основным хранилищем. Это очень существенно увеличивает производительность хаотичного чтения относительно статичных данных. Фикс 6536054 интегрирован в релиз Solaris Nevada Community Edition, Build 78.

Version 11

Существенно улучшена производительность очистки/ресинхронизации пула. Новый драйвер может быть использован и с томами более старых форматов, однако обновление пулов улучшает производительность очистки (scrubbing) при большом количестве файловых систем, снапшотов и клонов.

Фикс 6343667 доступен в Solaris Nevada Build 94, а также в FreeBSD-7.1-RELEASE.

Источники

1. ZFS on-disk specification (http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf)

2. http://www.opensolaris.org/os/community/zfs/version/N, где N — номер версии от 1 до 11
3. Исходники OpenSolaris от 07.01.2008
4. Jeff Bonwick's blog (http://blogs.sun.com/bonwick/)
5. Matthew Ahrens blog (http://blogs.sun.com/ahrens/)

Свежую версию этого документа, а также аналогичные по тематике статьи и переводы можно найти на www.filesystems.nm.ru