聊聊zfs中的核心数据结构

存储内核技术交流

微信扫描二维码，关注我的公众号

分布式存储技术研...

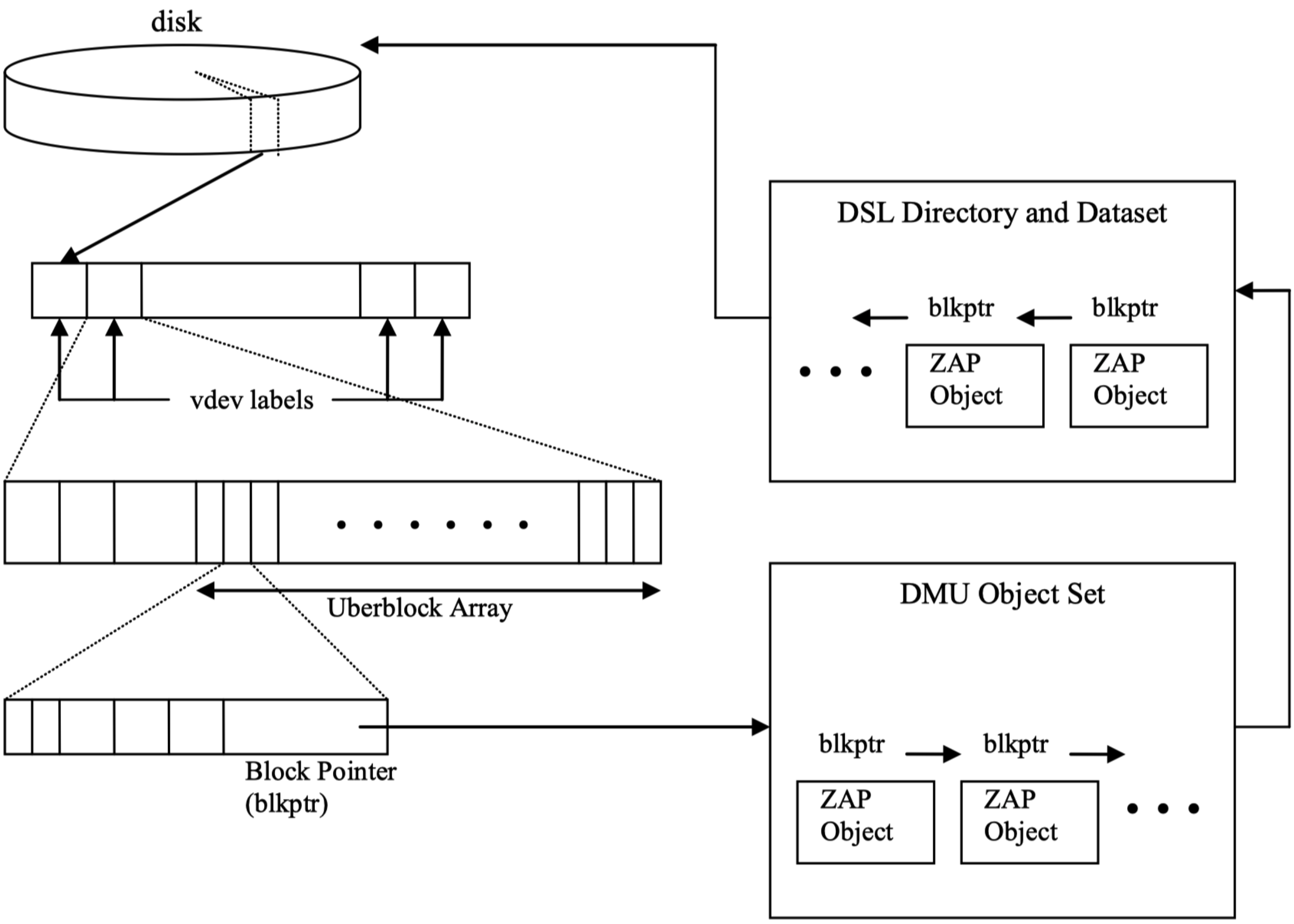
群号：672152841

扫一扫二维码，加入群聊。

QQ

开源存储问题解答社区:https://github.com/perrynzhou/deep-dive-storage-in-china

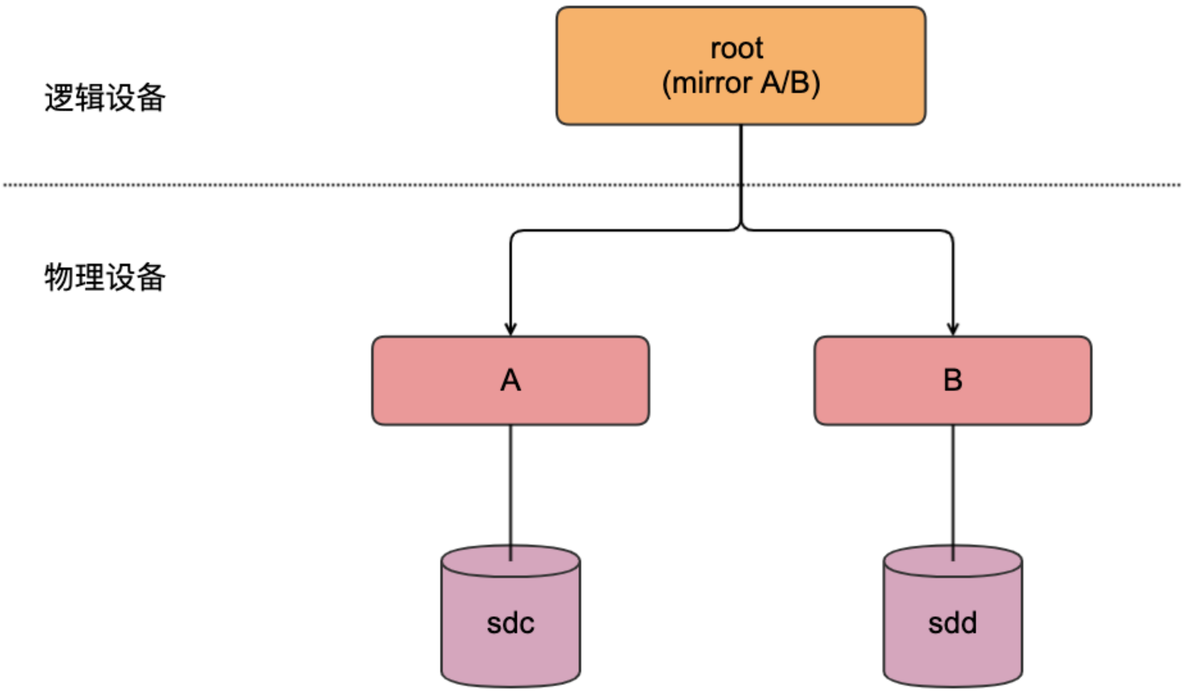
ZFS数据结构之间关系



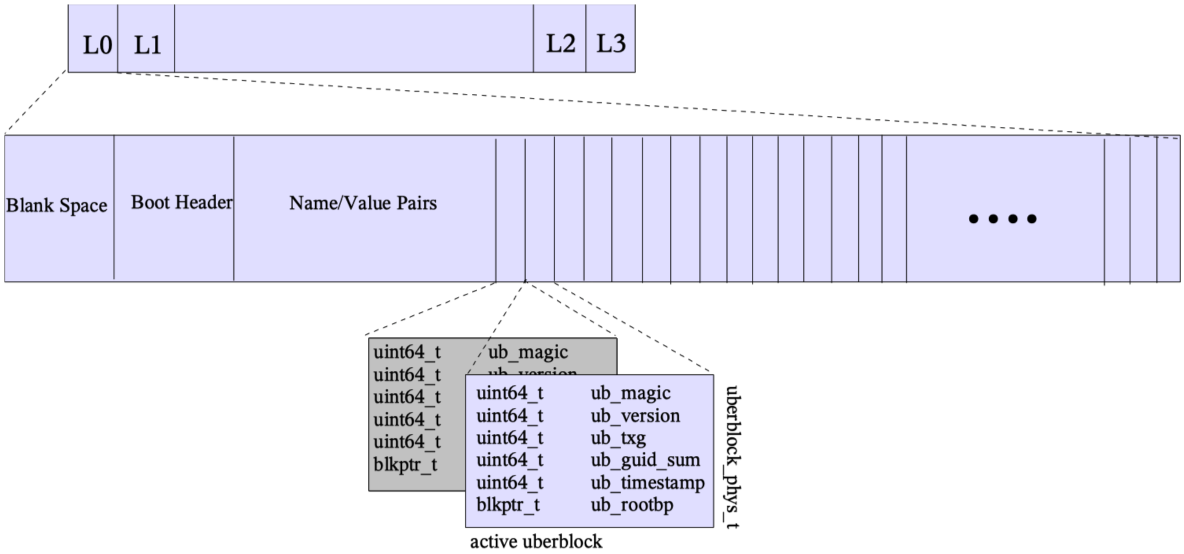
ZFS中核心数据结构

`uberblock_t`结构

- `zfs`中的分为逻辑设备和物理设备，逻辑和物理设备构成一棵树，非叶子节点是逻辑设备，叶子节点是物理磁盘设备。逻辑设备负责`zfs`中的`镜像/软raid`存储策略。



- uberblock_t可以理解为本地文件系统的超级快。在zfs中的vdev label中存储,vdev label存储4份，在物理磁盘的头部和尾部存储。在任何时候uberblock_t活跃只会有一个。第三个vdev label L2和第三个vdev label L3之间是数据存储区。




```
// vdev label的数据结构
typedef struct vdev_label {
        char                vl_pad1[VDEV_PAD_SIZE];             /*  8K */
        vdev_boot_envblock_t     vl_be;                         /*  8K */
        vdev_phys_t         vl_vdev_phys;                       /* 112K */
        char                vl_uberblock[VDEV_UBERBLOCK_RING];  /* 128K */
} vdev_label_t;                                              /* 256K total */


#define UBERBLOCK_MAGIC          0x00bab10c
typedef struct uberblock uberblock_t;
struct uberblock {
        // 存储设备和设备上数据的唯一标识
        uint64_t        ub_magic;
        // 磁盘的格式唯一标识
        uint64_t        ub_version;
        // zfs中所有写的操作都会添加到事务组，每个事务关联一个事务组编号，ub_txg是事务
组的编号。
        uint64_t        ub_txg;
        // 用来校验zfs pool中的vdev。当zfs pool打开的时候，zfs遍历所有叶子节点的磁
盘统计所有的guid,如果磁盘出现问题，这个时候就知道了
        uint64_t        ub_guid_sum;
        // uberblock的写入时间
        uint64_t        ub_timestamp;
        // ub_rootbp包含了MOS(meta-object set)的位置信息，根数据块的位置。
        blkptr_t        ub_rootbp;
        // uberblock软版本号，定为5000ULL
        uint64_t        ub_software_version;
        // ub_mmp_magic定义为0xa11cea11
        uint64_t        ub_mmp_magic;
        uint64_t        ub_mmp_delay;
```

```
        uint64_t        ub_mmp_config;
        uint64_t        ub_checkpoint_txg;
};
```

- 我们可以使用`zdb`命令dump出`zfs`中的uberblock的信息，dump命令参考`zgdb -vvv`.dump出来的信息包含了`pool的名称/主机名称/pool_guid/设备数`。这里仅仅会有一个磁盘，整个树中只会有一个叶子节点。dump出信息也包含叶子节点的物理 磁盘块的块大小(ashift=12,代表块大小是10的12次方)。

```
// 查看整个zfs pool的列表
$ zpool list
NAME      SIZE  ALLOC   FREE  CKPOINT  EXPANDSZ   FRAG    CAP  DEDUP
HEALTH  ALTROOT
sample  7.50G  40.7M  7.46G        -         -     0%     0%  1.00x
ONLINE  -

// 查看sample的pool的uberblock的情况
$ zdb -uuuvv sample

Uberblock:
        magic = 0000000000bab10c
        version = 5000
        txg = 4582
        guid_sum = 8780584948530658020
        timestamp = 1648490620 UTC = Mon Mar 28 14:03:40 2022
        mmp_magic = 00000000a11cea11
        mmp_delay = 0
        mmp_valid = 0
        rootbp = DVA[0]=<0:3c000:1000> DVA[1]=<0:2003c000:1000> DVA[2]=
<0:447d1000:1000> [L0 DMU objset] fletcher4 uncompressed unencrypted LE
contiguous unique triple size=1000L/1000P birth=4582L/4582P fill=71
cksum=2d00322c4:acd446c5f02:14c3c3fb6966a1:1aa3b027a1563db6
        checkpoint_txg = 0

// zfs pool的设备树信息
$ zdb -vvv
sample:
    version: 5000
    name: 'sample'
    state: 0
    txg: 4
    pool_guid: 3521762337703789458
    errata: 0
    hostname: 'CentOS8-Dev'
    com.delphix:has_per_vdev_zaps
    vdev_children: 1
    vdev_tree:
```

```
        type: 'root'
        id: 0
        guid: 3521762337703789458
        create_txg: 4
        children[0]:
            type: 'disk'
            id: 0
            guid: 5258822610826868562
            path: '/dev/sdc1'
            devid: 'ata-CentOS8-Dev-1_SSD_1MSJN73KVBY4Z8R3MKE4-part1'
            phys_path: 'pci-0000:00:1f.2-ata-4'
            whole_disk: 1
            metaslab_array: 128
            metaslab_shift: 29
            ashift: 12
            asize: 8574730240
            is_log: 0
            create_txg: 4
            com.delphix:vdev_zap_leaf: 66
            com.delphix:vdev_zap_top: 67
    features_for_read:
        com.delphix:hole_birth
        com.delphix:embedded_data
```

blkptr_t结构

- blkptr_t用来基于Data Virtual Address(DMA)追踪磁盘上的数据块block指针。

```
typedef struct blkptr {
        dva_t           blk_dva[SPA_DVAS_PER_BP]; /* Data Virtual Addresses
*/
        uint64_t        blk_prop;       /* size, compression, type, etc
*/
        uint64_t        blk_pad[2];     /* Extra space for the future
*/
        uint64_t        blk_phys_birth; /* txg when block was allocated
*/
        uint64_t        blk_birth;      /* transaction group at birth
*/
        uint64_t        blk_fill;       /* fill count
*/
        zio_cksum_t     blk_cksum;      /* 256-bit checksum
*/
} blkptr_t;
```

dnode_phys_t结构

- `DMU`层管理数据块Block然后按照类型分组为很多对象，这些对象的定义是由 `dnode_phys_t` 定义。

```
/*
 * VARIABLE-LENGTH (LARGE) DNODES
 *
 * The motivation for variable-length dnodes is to eliminate the overhead
 * associated with using spill blocks.  Spill blocks are used to store
 * system attribute data (i.e. file metadata) that does not fit in the
 * dnode's bonus buffer. By allowing a larger bonus buffer area the use of
 * a spill block can be avoided.  Spill blocks potentially incur an
 * additional read I/O for every dnode in a dnode block. As a worst case
 * example, reading 32 dnodes from a 16k dnode block and all of the spill
 * blocks could issue 33 separate reads. Now suppose those dnodes have size
 * 1024 and therefore don't need spill blocks. Then the worst case number
 * of blocks read is reduced from 33 to two--one per dnode block.
 *
 * ZFS-on-Linux systems that make heavy use of extended attributes benefit
 * from this feature. In particular, ZFS-on-Linux supports the xattr=sa
 * dataset property which allows file extended attribute data to be stored
 * in the dnode bonus buffer as an alternative to the traditional
 * directory-based format. Workloads such as SELinux and the Lustre
 * distributed filesystem often store enough xattr data to force spill
 * blocks when xattr=sa is in effect. Large dnodes may therefore provide a
 * performance benefit to such systems. Other use cases that benefit from
 * this feature include files with large ACLs and symbolic links with long
 * target names.
 *
 * The size of a dnode may be a multiple of 512 bytes up to the size of a
 * dnode block (currently 16384 bytes). The dn_extra_slots field of the
 * on-disk dnode_phys_t structure describes the size of the physical dnode
 * on disk. The field represents how many "extra" dnode_phys_t slots a
 * dnode consumes in its dnode block. This convention results in a value of
 * 0 for 512 byte dnodes which preserves on-disk format compatibility with
 * older software which doesn't support large dnodes.
 *
 * Similarly, the in-memory dnode_t structure has a dn_num_slots field
 * to represent the total number of dnode_phys_t slots consumed on disk.
 * Thus dn->dn_num_slots is 1 greater than the corresponding
 * dnp->dn_extra_slots. This difference in convention was adopted
 * because, unlike on-disk structures, backward compatibility is not a
 * concern for in-memory objects, so we used a more natural way to
 * represent size for a dnode_t.
 *
 * The default size for newly created dnodes is determined by the value of
 * the "dnodesize" dataset property. By default the property is set to
 * "legacy" which is compatible with older software. Setting the property
 * to "auto" will allow the filesystem to choose the most suitable dnode
```

```
 * size. Currently this just sets the default dnode size to 1k, but future
 * code improvements could dynamically choose a size based on observed
 * workload patterns. Dnodes of varying sizes can coexist within the same
 * dataset and even within the same dnode block.
 */

typedef struct dnode_phys {
        uint8_t dn_type;                /* dmu_object_type_t */
        uint8_t dn_indblkshift;         /* ln2(indirect block size) */
        uint8_t dn_nlevels;             /* 1=dn_blkptr->data blocks */
        uint8_t dn_nblkptr;             /* length of dn_blkptr */
        uint8_t dn_bonustype;           /* type of data in bonus buffer */
        uint8_t dn_checksum;            /* ZIO_CHECKSUM type */
        uint8_t dn_compress;            /* ZIO_COMPRESS type */
        uint8_t dn_flags;               /* DNODE_FLAG_* */
        uint16_t dn_datablkszsec;       /* data block size in 512b sectors
*/
        uint16_t dn_bonuslen;           /* length of dn_bonus */
        uint8_t dn_extra_slots;         /* # of subsequent slots consumed
*/
        uint8_t dn_pad2[3];

        /* accounting is protected by dn_dirty_mtx */
        uint64_t dn_maxblkid;           /* largest allocated block ID */
        uint64_t dn_used;               /* bytes (or sectors) of disk space
*/

        /*
         * Both dn_pad2 and dn_pad3 are protected by the block's MAC. This
         * allows us to protect any fields that might be added here in the
         * future. In either case, developers will want to check
         * zio_crypt_init_uios_dnode() and
zio_crypt_do_dnode_hmac_updates()
         * to ensure the new field is being protected and updated properly.
         */
        uint64_t dn_pad3[4];

        /*
         * The tail region is 448 bytes for a 512 byte dnode, and
         * correspondingly larger for larger dnode sizes. The spill
         * block pointer, when present, is always at the end of the tail
         * region. There are three ways this space may be used, using
         * a 512 byte dnode for this diagram:
         *
         * 0       64      128     192     256     320     384     448
(offset)
         * +---------------+---------------+---------------+-------+
         * | dn_blkptr[0]  | dn_blkptr[1]  | dn_blkptr[2]  | /     |
         * +---------------+---------------+---------------+-------+
         * | dn_blkptr[0]  | dn_bonus[0..319]                      |
         * +---------------+-----------------------+---------------+
```

```
         * | dn_blkptr[0]  | dn_bonus[0..191]      | dn_spill      |
         * +---------------+-----------------------+---------------+
         */
        union {
                blkptr_t dn_blkptr[1+DN_OLD_MAX_BONUSLEN/sizeof
(blkptr_t)];
                struct {
                        blkptr_t __dn_ignore1;
                        uint8_t dn_bonus[DN_OLD_MAX_BONUSLEN];
                };
                struct {
                        blkptr_t __dn_ignore2;
                        uint8_t __dn_ignore3[DN_OLD_MAX_BONUSLEN -
                            sizeof (blkptr_t)];
                        blkptr_t dn_spill;
                };
        };
} dnode_phys_t;
```

`objset_phys_t`结构

- `objset_phys_t`描述了一对象

```
typedef struct objset_phys {
        dnode_phys_t os_meta_dnode;
        zil_header_t os_zil_header;
        uint64_t os_type;
        uint64_t os_flags;
        uint8_t os_portable_mac[ZIO_OBJSET_MAC_LEN];
        uint8_t os_local_mac[ZIO_OBJSET_MAC_LEN];
        char os_pad0[OBJSET_PHYS_SIZE_V2 - sizeof (dnode_phys_t)*3 -
            sizeof (zil_header_t) - sizeof (uint64_t)*2 -
            2*ZIO_OBJSET_MAC_LEN];
        dnode_phys_t os_userused_dnode;
        dnode_phys_t os_groupused_dnode;
        dnode_phys_t os_projectused_dnode;
        char os_pad1[OBJSET_PHYS_SIZE_V3 - OBJSET_PHYS_SIZE_V2 -
            sizeof (dnode_phys_t)];
} objset_phys_t;
```

`mzap_phys_t`结构

- `mzap_phys_t`是ZAP层的对象，用来存储object的扩展属性 。

```c
typedef struct mzap_ent_phys {
        uint64_t mze_value;
        uint32_t mze_cd;
        uint16_t mze_pad;       /* in case we want to chain them someday */
        char mze_name[MZAP_NAME_LEN];
} mzap_ent_phys_t;

typedef struct mzap_phys {
        uint64_t mz_block_type; /* ZBT_MICRO */
        uint64_t mz_salt;
        uint64_t mz_normflags;
        uint64_t mz_pad[5];
        mzap_ent_phys_t mz_chunk[1];
        /* actually variable size depending on block size */
} mzap_phys_t;
```

dsl_dir_phys_t结构

- dnode_t中的bonus变量存储了少量的数据类型，其中就包括了dsl_dir_phys_t这是DMU_OT_DSL_DIR类型的对象。

```c
typedef struct dsl_dir_phys {
        uint64_t dd_creation_time; /* not actually used */
        uint64_t dd_head_dataset_obj;
        uint64_t dd_parent_obj;
        uint64_t dd_origin_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_flags;
        uint64_t dd_used_breakdown[DD_USED_NUM];
        uint64_t dd_clones; /* dsl_dir objects */
```

```
        uint64_t dd_pad[13]; /* pad out to 256 bytes for good measure */
} dsl_dir_phys_t;
```

`dsl_dataset_phys_t`结构

- `dnode_t`中的第二个`bonus`变量就是`dsl_dataset_phys_t`.

```
typedef struct dsl_dataset_phys {
        uint64_t ds_dir_obj;            /* DMU_OT_DSL_DIR */
        uint64_t ds_prev_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_prev_snap_txg;
        uint64_t ds_next_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_snapnames_zapobj;   /* DMU_OT_DSL_DS_SNAP_MAP 0 for
snaps */
        uint64_t ds_num_children;       /* clone/snap children; ==0 for
head */
        uint64_t ds_creation_time;      /* seconds since 1970 */
        uint64_t ds_creation_txg;
        uint64_t ds_deadlist_obj;       /* DMU_OT_DEADLIST */
        /*
         * ds_referenced_bytes, ds_compressed_bytes, and
ds_uncompressed_bytes
         * include all blocks referenced by this dataset, including those
         * shared with any other datasets.
         */
        uint64_t ds_referenced_bytes;
        uint64_t ds_compressed_bytes;
        uint64_t ds_uncompressed_bytes;
        uint64_t ds_unique_bytes;       /* only relevant to snapshots */
        /*
         * 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;              /* DS_FLAG_* */
        blkptr_t ds_bp;
        uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
        uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
        uint64_t ds_userrefs_obj;       /* DMU_OT_USERREFS */
        uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
} dsl_dataset_phys_t;
```

`znode_phys_t`结构

- 这个是存储在`dnode_t`中的`bonus`中的buffer中，包含了文件或者目录的基本元数据信息.

```
/*
 * This is a deprecated data structure that only exists for
 * dealing with file systems create prior to ZPL version 5.
 */
typedef struct znode_phys {
        uint64_t zp_atime[2];           /*  0 - last file access time */
        uint64_t zp_mtime[2];           /* 16 - last file modification time */
        uint64_t zp_ctime[2];           /* 32 - last file change time */
        uint64_t zp_crtime[2];          /* 48 - creation time */
        uint64_t zp_gen;                /* 64 - generation (txg of creation)
*/
        uint64_t zp_mode;               /* 72 - file mode bits */
        uint64_t zp_size;               /* 80 - size of file */
        uint64_t zp_parent;             /* 88 - directory parent (`..') */
        uint64_t zp_links;              /* 96 - number of links to file */
        uint64_t zp_xattr;              /* 104 - DMU object for xattrs */
        uint64_t zp_rdev;               /* 112 - dev_t for VBLK & VCHR files
*/
        uint64_t zp_flags;              /* 120 - persistent flags */
        uint64_t zp_uid;                /* 128 - file owner */
        uint64_t zp_gid;                /* 136 - owning group */
        uint64_t zp_zap;                /* 144 - extra attributes */
        uint64_t zp_pad[3];             /* 152 - future */
        zfs_acl_phys_t zp_acl;          /* 176 - 263 ACL */
        /*
         * Data may pad out any remaining bytes in the znode buffer, eg:
         *
         * |<---------------------- dnode_phys (512) ------------------------
>|
         * |<-- dnode (192) --->|<----------- "bonus" buffer (320) ----------
>|
         *                      |<---- znode (264) ---->|<---- data (56) ----
>|
         *
         * At present, we use this space for the following:
         *  - symbolic links
         *  - 32-byte anti-virus scanstamp (regular files only)
         */
} znode_phys_t;
```