

1 概述

YAFFS (Yet Another Flash File System) 文件系统是专门针对 NAND flash 设计的嵌入式文件系统，目前有 YAFFS 和 YAFFS2 两个版本，两个版本的主要区别之一在于 YAFFS2 能够更好的支持大容量的 NAND flash 芯片。YAFFS 文件系统有些类似于 JFFS/JFFS2 文件系统，与之不同的是 JFFS/JFFS2 文件系统最初是针对 NOR flash 的应用场合设计的，而 NOR flash 和 NAND flash 本质上有较大的区别，尽管 JFFS/JFFS2 文件系统也能应用于 NAND flash，但由于它在内存占用和启动时间方面针对 NOR 的特性做了一些取舍，所以 YAFFS2 对 NAND flash 来说通常才是最优的选择方案。

2 相关概念

分析 YAFFS2 之前，把 NAND flash 相关概念介绍下：NAND flash 由块(block)组成，块又由页(page)构成，擦除时以块为单位，读写时以页为单位，页又包含数据区和空闲区(OOB, Out-Of-Band)，而 Page 在 YAFFS2 中被称为 Chunk，其中的数据区用来存放实际的数据，OOB 用来存放附加信息实现 NAND flash 的管理。以 T8000 AXMPFUA 单板使用的 NAND flash 为例，每块 Block: 128 pages，每页 Page: (8K + 448) bytes，数据区为 8K，OOB 为 448bytes，如图 1 所示：

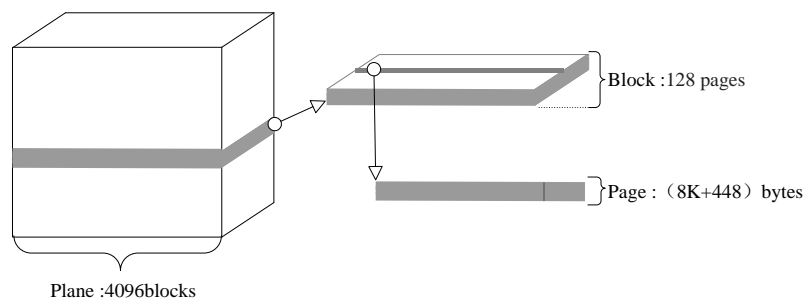


图 1 NAND flash 物理结构

3 数据结构

struct yaffs_dev 是 YAFFS2 文件系统最核心的数据结构，表示 YAFFS2 文件系统的超级块，它建立了整个文件系统的层次结构，并衔接 VFS 层和 MTD 层，与 struct super_block、struct mtd_info 的关系如图 2 所示：

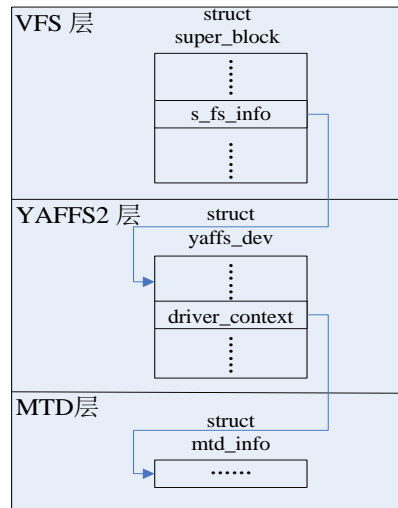


图2 yaffs_dev与super_block、mtd_info层次关系

下面围绕 struct yaffs_dev 这个最核心的数据结构开始，分段介绍它的含义，进而引出其他重要的数据结构：

```
struct yaffs_dev {
    struct yaffs_param param;
```

■param: 存储文件系统重要的一些参数，以及与 MTD 层的接口函数

```
struct yaffs_param {
    .....
    /*
     * Entry parameters set up way early. Yaffs sets up the rest.
     * The structure should be zeroed out before use so that unused
     * and default values are zero.
     */
    int inband_tags; /* Use unband tags */
    u32 total_bytes_per_chunk; /* Should be >= 512, does not need to
                               be a power of 2 */
    int chunks_per_block; /* does not need to be a power of 2 */
    int spare_bytes_per_chunk; /* spare area size */
    int start_block; /* Start block we're allowed to use */
    int end_block; /* End block we're allowed to use */
    int n_reserved_blocks; /* Tuneable so that we can reduce
                           * reserved blocks on NOR and RAM. */
    int n_caches; /* If <= 0, then short op caching is disabled,
                  * else the number of short op caches.
                  */
    .....
    int no_tags_ecc; /* Flag to decide whether or not to do ECC
                     * on packed tags (yaffs2) */
    int is_yaffs2; /* Use yaffs2 mode on this device */
    .....
    int refresh_period; /* How often to check for a block refresh */
    /* Checkpoint control. Can be set before or after initialisation */
    u8 skip_checkpoint_rd;
    u8 skip_checkpoint_wr;
    .....
    /* NAND access functions (Must be set before calling YAFFS) */
    int (*erase_fn) (struct yaffs_dev *dev, int flash_block);
    .....
    int (*write_chunk_tags_fn) (struct yaffs_dev *dev,
                               int nand_chunk, const u8 *data,
                               const struct yaffs_ext_tags *tags);
    int (*read_chunk_tags_fn) (struct yaffs_dev *dev,
                               int nand_chunk, u8 *data,
                               struct yaffs_ext_tags *tags);
    .....
    int wide_tnodes_disabled; /* Set to disable wide tnodes */
} ? end yaffs_param ? ;
```

□inband_tags: 标志位，默认为 0，即采用 OOB（out of band）方式存储 tags，

可以通过挂载时指定 inband-tags 选项改变默认值

□total_bytes_per_chunk: 每个 chunk 总的字节数

□chunks_per_block: 每个 block 总的 chunk 数

□spare_bytes_per_chunk: 每个 chunk 包含 OOB 块的字节数

□start_block: 第一个可以使用的 block

□end_block: 最后一个可以使用的 block

□n_reserved_blocks: 为 GC 保留的 block 阈值

□n_caches: 缓冲区的个数, YAFFS2 为减少数据的碎片以及提高性能为每个文件的写入提供了 cache

□no_tags_ecc: 标志位, 默认为 0, 即 tags 中包括 ECC 纠错信息, 可以通过内核配置改变默认值, CONFIG_YAFFS_DISABLE_TAGS_ECC

□is_yaffs2: 标志位, 默认为 0, 即 YAFFS, 在挂载的过程中会根据识别的 mtd->writesize 自动转换成 YAFFS2

□refresh_period: 刷新周期, 刷新目的主要找出最旧的处于 YAFFS_BLOCK_STATE_FULL 状态的 block, 供 GC 作为 gc_block 使用

□skip_checkpoint_rd: 标志位, 默认为 0, 支持读取 checkpoint, 提高挂载速度的一个功能可以通过挂载时指定挂载选项 no-checkpoint-read、no-checkpoint 修改默认值

□skip_checkpoint_wr: 标志位, 默认为 0, 支持写入 checkpoint, 提高挂载速度的一个功能可以通过挂载时指定挂载选项 no-checkpoint-write、no-checkpoint 修改默认值

□write_chunk_tags_fn: 函数指针, 在挂载的文件系统的时候会被初始, NAND flash 写入接口函数:

param->write_chunk_tags_fn = nandmtd2_write_chunk_tags;

□read_chunk_tags_fn: 函数指针, 在挂载的文件系统的时候会被初始, NAND flash 读取接口函数:

param->write_chunk_tags_fn = nandmtd2_write_chunk_tags;

□erase_fn: 函数指针, 在挂载的文件系统的时候会被初始, NAND flash 擦除 block 接口函数: param->erase_fn = nandmtd_erase_block;

□wide_tnodes_disabled: 标志位, 默认值为 0, 采用动态位宽, 通过内核配置修改可采用静态位宽 CONFIG_YAFFS_DISABLE_WIDE_TNODES

```
/* Context storage. Holds extra OS specific data for this device */
void *os_context;
void *driver_context;
```

■os_context:指向yaffs_linux_context结构指针, 该结构存储YAFFS2运行环境, 如下:

```
struct yaffs_linux_context {
    struct list_head context_list; /* List of these we have mounted */
    struct yaffs_dev *dev;
    struct super_block *super;
    struct task_struct *bg_thread; /* Background thread for this device */
    int bg_running;
    struct mutex gross_lock; /* Gross locking mutex*/
    u8 *spare_buffer; /* For mtdif2 use. Don't know the buffer size
                       * at compile time so we have to allocate it.
                       */
    struct list_head search_contexts;
    void (*put_super_fn) (struct super_block *sb);

    struct task_struct *readdir_process;
    unsigned mount_id;
};
```

□context_list: 通过该字段加入到yaffs_context_list全局链表中

- dev: 指向YAFFS2文件系统超级块的指针
- super: 指向VFS层超级块的指针
- bg_thread: YAFFS2后台垃圾回收线程的指针
- bg_running: 启动和停止垃圾回收线程的标志位, 1: 启动, 0: 停止
- gross_lock: 互斥锁, 保护整个超级块关键字段的互斥访问, 粒度比较大
- spare_buffer: OOB块的缓冲区
- search_contexts: 通过该字段把所有Directory Search Context组成链表
- yaffs_mtd_put_super: 卸载文件系统时被调用来清理super_block
- readdir_process: 解决使用NFS死锁问题加入的
- mount_id: 每个NAND flash分区挂载YAFFS2都分配不同的ID号

■driver_context: 指向mtd_info结构指针, mtd_info是MTD子系统核心的数据结构, 主要是对底层硬件驱动进行封装, 这里不再介绍

```
/* Runtime parameters. Set up by YAFFS. */
int data_bytes_per_chunk;
```

■data_bytes_per_chunk: 每个chunk总的字节数, 和前面提到的total_bytes_per_chunk一样

```
/* Non-wide tnode stuff */
u16 chunk_grp_bits; /* Number of bits that need to be resolved
                    *if the tnodes are not wide enough.*/
u16 chunk_grp_size; /* == 2^^chunk_grp_bits */

/* Stuff to support wide tnodes */
u32 tnode_width;
u32 tnode_mask;
u32 tnode_size;
```

■chunk_grp_bits : 采用静态位宽时超过tnode_width宽度之后的位数, 采用动态位宽值恒为0

■chunk_grp_size: 由chunk_grp_bits转化而来的大小

■tnode_width: 采用静态位宽默认是16, 采用动态位宽是由整个NAND flash中chunk数目计算得到

■tnode_mask: 位宽的mask, 主要用于快速获取chunk id号

■tnode_size: YAFFS_NTNODES_LEVEL0节点所占用的内存大小, 单位:byte

```
/* Stuff for figuring out file offset to chunk conversions */
u32 chunk_shift; /* Shift value */
u32 chunk_div; /* Divisor after shifting: 1 for 2^n sizes */
u32 chunk_mask; /* Mask to use for power-of-2 case */

int is_mounted;
int read_only;
int is_checkpointed;
```

■chunk_shift: 主要用来计算logical chunk index以及logical chunk offset

■chunk_div: 作用同chunk_shift, 主要用于chunk大小不是2次幂的情况

■chunk_mask: 作用同chunk_shift, 组合起来计算logical chunk offset

■is_mounted: 标志位, 文件系统挂载时被置位

```
/* Runtime checkpointing stuff */
int checkpoint_page_seq; /* running sequence number of chkpt pages */
int checkpoint_byte_count;
int checkpoint_byte_offs;
```

```

u8 *ckpt_buffer;
int ckpt_open_write;
int blocks_in_ckpt;
int ckpt_cur_chunk;
int ckpt_cur_block;
int ckpt_next_block;
int *ckpt_block_list;
int ckpt_max_blocks;
int checkpoint_blocks_required; /* Number of blocks needed to store
                                * current checkpoint set */

```

Checkpoint是为提高挂载速度而引入的功能，作用同JFFS2的EBS，以空间来换取时间，卸载时通过在NAND flash上保存文件系统超级块快照，挂载时获取快照信息可以快速还原系统的状态。

- ckpt_page_seq: 控制checkpoint的写入或读出时的chunk序列号
- ckpt_byte_count: 写入或读出checkpoint信息的字节数
- ckpt_byte_offs: ckpt_buffer缓存区的偏移量
- ckpt_buffer: 写入或读出checkpoint的缓存区，大小为data_bytes_per_chunk字节
- ckpt_open_write: 标志位，yaffs2_ckpt_open时传入的，决定checkpoint读写属性
- blocks_in_ckpt: checkpoint使用的block的数量
- ckpt_cur_chunk: 当前block已经使用掉的chunk数量
- ckpt_cur_block: 当前正在使用的block编号
- ckpt_next_block: 当前正在使用的下一个block编号
- ckpt_block_list: 在读checkpoint时使用，该数组中保留使用的block编号，checkpoint信息读取完毕，根据这个数组保存的block编号，更改block的状态为YAFFS_BLOCK_TATE_CHECKPOINT，以便卸载时检测到这样状态的block时擦除
- ckpt_max_blocks: checkpoint能使用的block最大数量
- checkpoint_blocks_required: checkpoint信息所需的block数量，看该字段的计算大致就知道checkpoint需要保存文件系统的哪些信息

```

/* Block Info */
struct yaffs_block_info *block_info;
u8 *chunk_bits; /* bitmap of chunks in use */
unsigned block_info_alt:1; /* allocated using alternative alloc */
unsigned chunk_bits_alt:1; /* allocated using alternative alloc */
int chunk_bit_stride; /* Number of bytes of chunk_bits per block.
                      * Must be consistent with chunks_per_block.
                      */
int n_erased_blocks;
int alloc_block; /* Current block being allocated off */
u32 alloc_page;
int alloc_block_finder; /* Used to search for next allocation block */

```

■block_info: 指向yaffs_block_info类型的指针数组，数组每一项表示一块擦除块的统计信息，由yaffs_block_info表示，该信息在运行时只存在于内存中，当YAFFS2被卸载时，该数据当作checkpoint信息被记录下来，在下一次挂载时被读出并恢复，具体定义如下：

```

struct yaffs_block_info {
    int soft_del_pages:10; /* number of soft deleted pages */
    int pages_in_use:10; /* number of pages in use */
    unsigned block_state:4; /* One of the above block states.
                           /* NB use unsigned because enum is sometimes
                           * an int */
    u32 needs_retiring:1; /* Data has failed on this block,
                           /* need to get valid data off and retire */
    u32 skip_erased_check:1; /* Skip the erased check on this block */

```

```

u32 gc_prioritise:1; /* An ECC check or blank check has failed.
                      Block should be prioritised for GC */
u32 chunk_error_strikes:3; /* How many times we've had ecc etc
                           failures on this block and tried to reuse it */

#ifdef CONFIG_YAFFS_YAFFS2
u32 has_shrink_hdr:1; /* This block has at least one shrink header */
u32 seq_number; /* block sequence number for yaffs2 */
#endif

} ? end_yaffs_block_info ? ;

```

□soft_del_pages: 在YAFFS2包含两种类型的删除，delete和soft delete。前者用于文件内容的更新，比如修改文件中的部分内容，这时YAFFS2会分配新的chunk，将修改后的内容写入新的chunk中，原来chunk的内容自然就没用了，所有将pages_in_use减1，并修改chunk_bits。后者用于文件的删除，YAFFS2在删除文件时，只是删除该文件在内存中的一些描述结构，而被删除文件所占用的chunk不会立即释放，不会删除文件的内容，在后续的文件系统操作中一般也不会把这些chunk分配出去，直到系统进行垃圾收集时才有选择地释放这些chunk。

soft_del_pages就表示soft delete的chunk数目

□pages_in_use: 该擦除块中被使用的chunk数目，包括已经被soft delete的chunk

□block_state: 该擦除块的状态，比如，YAFFS_BLOCK_STATE_FULL表示该擦除块中所有的chunk已经被分配完，YAFFS_BLOCK_STATE_DIRTY表示该擦除块中所有的chunk已经被delete可以被擦除了，YAFFS_BLOCK_STATE_CHECKPOINT表示该擦除块保存的是checkpoint信息，YAFFS_BLOCK_STATE_EMPTY表示空闲的擦除块

□needs_retiring: 标志位，chunk_error_strikes次数超过3次以上就会置位，该擦除块需要重新回收擦除

□skip_erased_check: 标志位，置0时需要对该擦除块进行检测，一般只检测该擦除块的第一个chunk，置1时跳过对该擦除块的检测，可以通过CONFIG_YAFFS_ALWAYS_CHECK_CHUNK_ERASED设置

□gc_prioritise: 标志位，该块发生过ECC校验错误或check失败，需要在垃圾回收时优先擦除

□chunk_error_strikes: 发生ECC校验错误的次数

□has_shrink_hdr: 标志位，置0时表示该擦除块上的存储的文件被没有发生过截断truncate，即文件大小没有发生过变化resize，否则必须在文件的objectheader标识，同时该object header所在擦除块上也标识，即has_shrink_hdr置1，

□seq_number: 序列号，表示擦除块被使用的先后顺序，序号越小越早被使用，在挂载时起到非常重要的作用

■chunk_bits: 指向擦除块位图的指针，每一位对应一个chunk，置0表示没有被使用，置1表示在使用中

■block_info_alt: 标志位，采用kmalloc分配block_info所使用的内存则置0，采用vmalloc分配则置1

■chunk_bits_alt: 标志位，采用kmalloc分配chunk_bits所使用的内存则置0，采用vmalloc分配则置1

■chunk_bit_stride: 位宽，擦除块中每个chunk占用一位，总的位宽应等于chunks_per_block，但要按byte对齐

■n_erased_blocks: 空闲擦除块的数目

- alloc_block: 当前正在被写入的擦除块在block_info指针数组中的下标
- alloc_page: 当前正在被写入的擦除块中chunk的顺序号
- alloc_block_finder: 记录下一个被选择写入的擦除块在block_info指针数组中的下标

```

/* Object and Tnode memory management */
void *allocator;
int n_obj;
int n_tnodes;
int n_hardlinks;
struct yafts_obj_bucket obj_bucket[YAFFS_NOBJECT_BUCKETS];
u32 bucket_finder;
int n_free_chunks;

```

■allocator:指向struct yafts_allocator结构的指针，YAFFS2文件系统实现的一个内存分配器，主要为struct yafts_allocator、struct yafts_tnode分配高速缓存替代原有的slab，struct yafts_allocator定义如下：

```

struct yafts_allocator {
    int n_tnodes_created;
    struct yafts_tnode *free_tnodes;
    int n_free_tnodes;
    struct yafts_tnode_list *alloc_tnode_list;

    int n_obj_created;
    struct list_head free_objs;
    int n_free_objects;
    struct yafts_obj_list *allocated_obj_list;
};

```

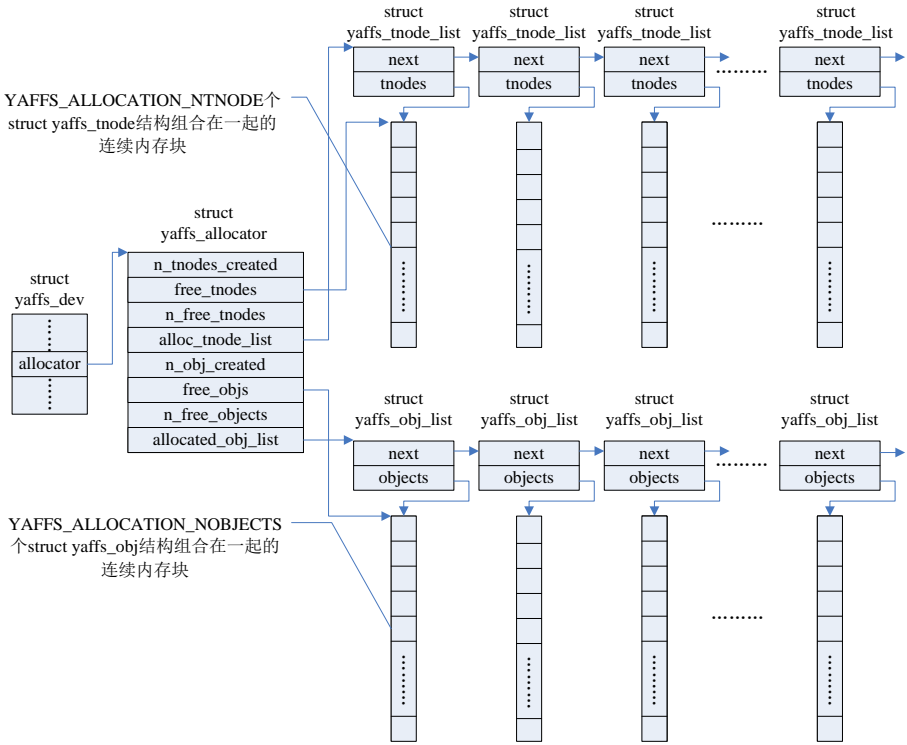


图3 yafts_allocator分配器数据结构之间的关系

- n_tnodes_created: 统计值，分配过总的struct yafts_tnode数目
- free_tnodes: 指向一块连续的内存区域，这块连续内存被分成YAFFS_ALLOCATION_NTNODE个struct yafts_tnode结构大小的内存
- n_free_tnodes: 分配器中空闲tnodes的数目
- alloc_tnode_list: 把yafts_tnode连续的内存区域存放到此链表中

- n_obj_created: 统计值，分配过总的struct yaffs_obj数目
- free_objs: 指向一块连续的内存区域，这块连续内存被分成YAFFS_ALLOCATION_NOOBJECTS个struct yaffs_obj结构大小的内存
- n_free_objects: 分配器中空闲objects的数目
- allocated_obj_list: 把yaffs_obj连续的内存区域存放到此链表中

这里引入了两个最核心的数据结构struct yaffs_tnode和struct yaffs_obj，下面分别来介绍：

```
/*----- Tnode ----- */
struct yaffs_tnode {
    struct yaffs_tnode *internal[YAFFS_NTNODES_INTERNAL];
};
```

这是一个长度为YAFFS_NTNODES_INTERNAL的指针数组。据此结构创建的节点树最底层的节点成为叶子节点，中间的是非叶子节点。不管是叶子节点还是非叶节点，都是这个结构。当节点为非叶节点时，数组中的每个元素都指向下一层子节点；当节点为叶子节点时，该数组拆分为YAFFS_NTNODES_LEVEL0个tnode_width位长的短整数，该短整数就是文件内容在flash 上的存储位置。最底层节点即叶节点中tnode_width的位宽决定了文件系统所能寻址的FLASH最大空间，假设tnode_width=16，即可以表示65 536 个chunk。对于chunk的大小为8KB，这种情况下所能寻址的最大FLASH空间是512MB。

文件系统通过节点树查找文件是非常简单和方便的。中间节点使用的Tnode每组有8个指针，需要3位二进制代码对其进行索引，同时，叶节点使用的Tnode每组有16个指针，所以需要4 位二进制代码对其进行索引。当节点树结构Tnode刚开始建立时，仅建立最底层Lowest-Level Tnode，当File所配置的Chunk数超过16个时，此Tree会建立一个中间节点internal Tnode，并将其第0 个internal[0]指向原本的Lowest-Level Tnode。当读取的Chunk数愈来愈多时，会一直新增Tnode，且节点树也会越来越高，如图4所示。

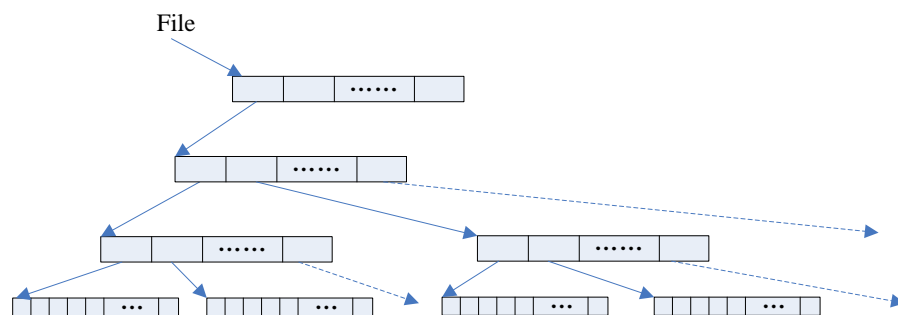


图3 文件的Tnode节点树

Tnode节点树建立了文件内偏移物理chunk之间的映射，YAFFS2靠logical chunk index 来检索Tnode tree，从而得到physical chunk index。比如说读文件时，使用者给出文件内的offset，YAFFS2根据chunk size计算出该offset在文件的第几个chunk中，即计算出logical chunk index，然后在Tnode tree 中找到相应的physical chunk index。


```

struct yaffs_obj {
    u8 deleted:1;          /* This should only apply to unlinked files. */
    u8 soft_del:1;        /* it has also been soft deleted */
    u8 unlinked:1;        /* An unlinked file.*/
    u8 fake:1;            /* A fake object has no presence on NAND. */
    /*.....*/
    u8 dirty:1;           /* the object needs to be written to flash */
    u8 valid:1;           /* When the file system is being loaded up, this
                           * object might be created before the data
                           * is available
                           * ie. file data chunks encountered before
                           * the header.
                           */
    /*.....*/
    struct list_head hash_link; /* list of objects in hash bucket */
    /* directory structure stuff */
    /* also used for linking up the free list */
    struct yaffs_obj *parent;
    struct list_head siblings;
    /*.....*/
    /* Where's my object header in NAND? */
    int hdr_chunk;
    int n_data_chunks; /* Number of data chunks for this file. */
    u32 obj_id;        /* the object id value */
    /*.....*/
    enum yaffs_obj_type variant_type;
    union yaffs_obj_var variant;
} ? end yaffs_obj ? ;

```

在YAFFS2中，不管是文件、目录或者是链接，在内存中都由struct yaffs_obj来描述。这里介绍几个关键的字段：

□deleted、soft_del、unlinked：这三个字段用于描述该文件对象在删除过程中所处的阶段。在删除文件时，首先要将文件从原目录移至一个特殊的系统目录/unlinked，以此拒绝应用程序对该文件的访问，此时将unlinked置1；然后判断该文件长度是否为0，如果为0，该文件就可以直接删除，此时将deleted置1；如果不为0，就将deleted和soft_del都置1，表明该文件数据所占据的chunk还没有释放，要留待后续垃圾回收处理。

□fake：创建fake directory时被置1，有两种directory：一种是普通的directory，具有object header，另外一种fake directory，没有object header，只存在于内存中。它们是root、lost+found、deleted与unlinked

□dirty：文件被修改时置1

□valid：文件object header写入的时候置1，没有object header时置0，表示文件并未真正生效

□hash_link：通过该字段把文件对象挂接到散列表中

□parent：指向父目录的yaffs_obj对象

□siblings：在同一个目录的yaffs_obj对象通过该字段组成双向循环链表

□hdr_chunk：每个文件在flash上都有一个object header，存储着该文件的大小、所有者、创建修改时间等信息，hdr_chunk就是该文件头在flash上的chunk号，object header一旦被写入chunk，就不能再修改，只能在另一个chunk中写入一个新的object header

□obj_id：每一个文件系统对象都被赋予一个唯一的编号，作为对象标识，也用于将该对象挂入一个散列表，加快对象的搜索速度。

□variant_type、variant：前者表示该对象的类型，是目录、普通文件还是链接文

件，后者是一个联合体，根据对象类型的不同有不同的解释，双向循环链表，
obj->variant.dir_variant.children 记录着该目录下的所有子目录和文件

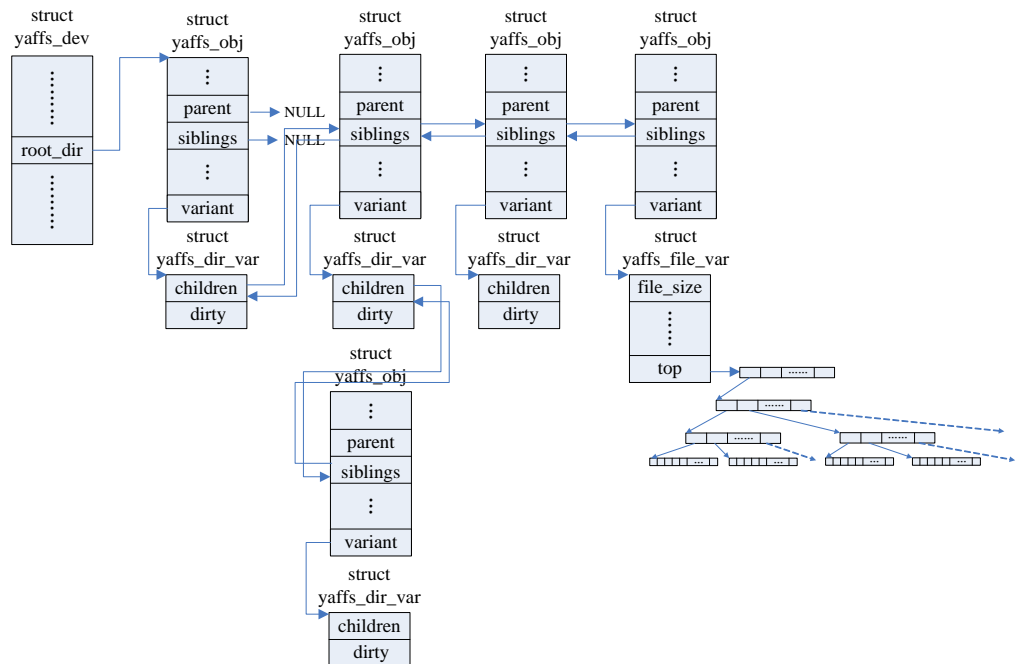


图4 YAFFS2文件系统目录层次结构

```

/* ----- Object structure ----- */
/* This is the object structure as stored on NAND */
struct yaffs_obj_hdr {
    enum yaffs_obj_type type;
    /* Apply to everything */
    int parent_obj_id;
    u16 sum_no_longer_used; /* checksum of name. No longer used */
    YCHAR name[YAFFS_MAX_NAME_LENGTH + 1];
    /* The following apply to all object types except for hard links */
    u32 yst_mode; /* protection */
    u32 yst_uid;
    u32 yst_gid;
    u32 yst_atime;
    u32 yst_mtime;
    u32 yst_ctime;
    /* File size applies to files only */
    int file_size;
    /* Equivalent object id applies to hard links only. */
    int equiv_id;
    /* Alias is for symlinks only. */
    YCHAR alias[YAFFS_MAX_ALIAS_LENGTH + 1];
    u32 yst_rdev; /* stuff for block and char devices (major/min) */
    u32 win_ctime[2];
    u32 win_atime[2];
    u32 win_mtime[2];
    u32 inband_shadowed_obj_id;
    u32 inband_is_shrink;
    u32 reserved[2];
    int shadows_obj; /* This object header shadows the
                     specified object if > 0 */
    /* is_shrink applies to object headers written when we make a hole. */
    u32 is_shrink;
} ? end yaffs_obj_hdr ? ;

```

struct yaffs_obj_hdr表示object header的结构，该结构只存在于flash，每个文件都有一个object header，存储着该文件的大小、所有者、创建修改时间等信息。object header一旦被写入chunk，就不能再修改。如果文件被修改，只能在另一个

chunk 中写入一个新的object header。关键的字段：

□type: 关联文件的类型，包括普通文件、目录还是链接文件等

□name: 文件名，object name

□is_shrink: 所关联的文件发生过resize。如果file 发生过resize，必须在file 的object header 上标识出这一点；同时必须在object header所在block上也标识出这一点。关于shrink header的说明可以见作者的原文：

[Yaffs] Improving/ignoring yaffs2 shrink header handling

<http://www.yaffs.net/lurker/message/20070430.103622.962952a8.fr.html>

上述邮件列表提到的下面这些操作：

1. Create file. Writes object header
2. Write 5MB of data
3. Truncate file to 1MB: write object header (flen=1MB)
4. Seek to 2MB
5. Write 1MB of data.
6. Close file: write object header (flen = 3MB).

如果垃圾回收没有擦除上述操作涉及到的chunk。在挂载扫描FLASH的过程中，根据扫描的顺序，在所有操作结束后，会有 $(5M-3M) = 2MB$ 的chunk被错误的放入了tnode tree 中。这就是作者提到的必须要shrink flag的原因，问题可以这样解决：

◇在扫描完步骤1、2涉及到的所有chunk后，有5MB chunk被插入到了tnode tree。

◇当扫描到步骤3产生的object header后，看到shrink flag被设置，则resize file，完成后只剩下1MB 的chunk在tnode tree中。

这样上述问题就解决了。要注意的是object header的shrink flag不能丢失，那么shrink flag应该“活”到什么时候呢？答案简单，那就是“活”到步骤3 resize下来的chunk被擦除掉。但是，根据object header上的信息，找到步骤3 resize下来的chunk，那几乎是不可能完成的。所以YAFFS2在做垃圾回收时，碰到包含有shrink flag的擦除块后，只有在它之前分配的所有擦除块均被擦除了，才擦除它。也不延长它的生存期，因为如果延长，则无法确定什么时候应该擦除。干脆要么把它放在那，要么等在它之前分配的擦除块均擦除了，再擦除它。对于没包含shrink flag 的block，则没有这一约束。

■n_obj: 文件系统中已经分配的struct yaffs_obj的数量，计算checkpoint信息时使用

■n_tnodes: 文件系统中已经分配的struct yaffs_tnode的数量，计算checkpoint信息时使用

■obj_bucket: struct yaffs_obj对象的散列表，以obj_id为键，便于文件查找和操作，如图5所示：

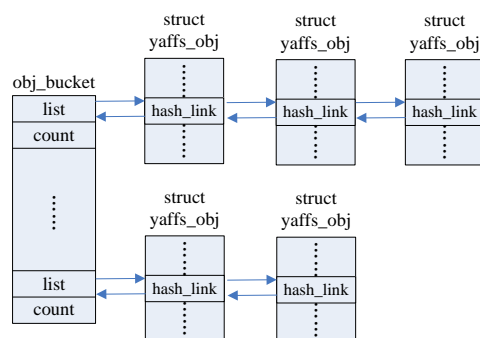


图5 struct yaffs_obj对象散列表

■bucket_finder: 散列表中最短链表的索引

■n_free_chunks: flash中空闲的chunk数量

```
/* Garbage collection control */
u32 *gc_cleanup_list; /* objects to delete at the end of a GC. */
u32 n_clean_ups;
unsigned has_pending_prioritised_gc; /* We think this device might
                                     have pending prioritised gcs */
unsigned gc_disable;
unsigned gc_block_finder;
unsigned gc_dirtiest;
unsigned gc_pages_in_use;
unsigned gc_not_done;
unsigned gc_block;
unsigned gc_chunk;
unsigned gc_skip;
```

■gc_cleanup_list: 数组, 保存垃圾回收时可以删除的yaffs_obj对象obj_id

■n_clean_ups: 上面提到的数组下标

■has_pending_prioritised_gc: 标志位, 设备存在优先回收的擦除块

■gc_disable: 标志位, 置1禁止垃圾回收, 置0使能垃圾回收, 主要用于垃圾回收时的同步, 防止垃圾回收的重入, 进入垃圾流程置1, 退出时置0

■gc_block_finder: 存储可以进行垃圾回收擦除块的编号

■gc_dirtiest: 存储最脏的擦除块的编号

■gc_pages_in_use: 被选中垃圾回收的擦除块有效数据使用的page数目不能超过一定阈值, 否则代价太大, 需要把有效数据搬移到空闲擦除的page中

■gc_not_done: 跳过垃圾回收的次数

■gc_block: 正在被垃圾回收的擦除块

■gc_chunk: 垃圾回收时会判断每个chunk是否有效, gc_chunk表示正在被检查的那个chunk, 有效的话需要把该chunk的数据搬移到其他空闲擦除块的chunk上

■gc_skip: 没有使用该字段

```
/* Special directories */
struct yaffs_obj *root_dir;
struct yaffs_obj *lost_n_found;
/* Stuff for background deletion and unlinked files. */
struct yaffs_obj *unlinked_dir; /* Directory where unlinked and deleted
                                files live. */
struct yaffs_obj *del_dir; /* Directory where deleted objects are
                             sent to disappear. */
```

■root_dir、lost_n_found、unlinked_dir、del_dir、unlinked_deletion: 这几种目录都是fake directory, 只存在于内存中, 没有object header, 该类目录禁止改名、禁止删除, 文件对象的fake的字段置1

```
struct yaffs_cache *cache;
int cache_last_use;
```

■cache: 现实的文件操作中存在大量的short sequential reads, and short sequential writes, 页缓存能提高short sequential reads类型的读性能, 而为了提高short sequential writes类型的写性能, YAFFS2文件系统实现了自己的文件缓存。yaffs_guts_initialise函数对文件系统要使用的cache进行初始化, 缓存的数目有dev->param.n_caches决定, 缓存的大小由dev->param.total_bytes_per_chunk决定, 写入的数据小于total_bytes_per_chunk时就暂时缓存到cache中, 达到total_bytes_per_chunk再写入到FLASH。

■cache_last_use: cache被使用的次数, 通过该字段可以看出文件操作是否存在频繁的short

sequential writes

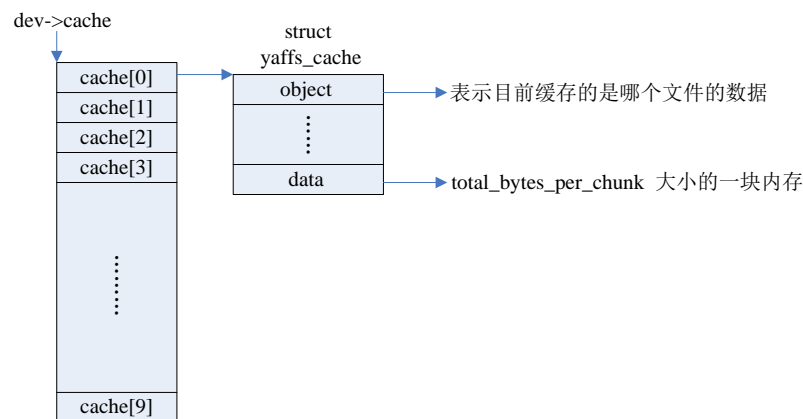


图6 yaffs文件系统的cache结构

```
/* Temporary buffer management */
struct yaffs_buffer temp_buffer[YAFFS_N_TEMP_BUFFERS];
```

■temp_buffer: YAFFS2实现了一个临时的buffer，buffer大小为total_bytes_per_chunk，主要用来临时存放从chunk中读出的数据

```
/* yaffs2 runtime stuff */
unsigned seq_number; /* Sequence number of currently
                        allocating block */
unsigned oldest_dirty_seq;
unsigned oldest_dirty_block;
```

■seq_number: 当被用作序列号时，sequence Number表示擦除块被使用的前后顺序，越小则表示被使用的越早，反之亦然。擦除块的sequence number在gc与上电scan时起到了非常重要的作用。

■oldest_dirty_seq: 最小的seq_number号也就是最老的seq_number号

■oldest_dirty_block: 最小的seq_number号对应的擦除块编号

```
/* Dirty directory handling */
struct list_head dirty_dirs; /* List of dirty directories */
```

■dirty_dirs: 需要同步更新的文件对象挂到这个链表下

最后介绍下struct yaffs_ext_tags结构， struct yaffs_ext_tags结构就是用来chunk的归属信息定义如下，主要字段：

```
struct yaffs_ext_tags {
    unsigned validity0;
    unsigned chunk_used; /* Status of the chunk: used or unused */
    unsigned obj_id; /* If 0 this is not used */
    unsigned chunk_id; /* If 0 this is a header, else a data chunk */
    unsigned n_bytes; /* Only valid for data chunks */
    /* The following stuff only has meaning when we read */
    enum yaffs_ecc_result ecc_result;
    unsigned block_bad;
    /* YAFFS 1 stuff */
    unsigned is_deleted; /* The chunk is marked deleted */
    unsigned serial_number; /* Yaffs1 2-bit serial number */
    /* YAFFS2 stuff */
    unsigned seq_number; /* The sequence number of this block */
    /* Extra info if this is an object header (YAFFS2 only) */
    unsigned extra_available; /* Extra info available if not zero */
    unsigned extra_parent_id; /* The parent object */
    unsigned extra_is_shrink; /* Is it a shrink header? */
    unsigned extra_shadows; /* Does this shadow another object? */
    enum yaffs_obj_type extra_obj_type; /* What object type? */
    unsigned extra_length; /* Length if it is a file */
    unsigned extra_equiv_id; /* Equivalent object for a hard link */
    unsigned validity1;
} ? end yaffs_ext_tags ? ;
```

- **chunk_used**: 为0表示该chunk未分配使用；为1表示该chunk已分配使用
- **obj_id**: 为0表示该chunk未分配使用；非0表示文件的obj_id信息，通过该字段可知道该chunk属于哪个文件
- **chunk_id**: 为0表示该chunk存储的是一个文件的object header；非0表示该chunk存储的是data chunk，表示文件logical chunk index，可以根据这个字段计算所保存的数据在文件内的偏移
- **n_bytes**: 表示该chunk包含的数据量，单位是byte
- **seq_number**: 等同于擦除块的seq_number，表示擦除块被使用的先后顺序，block中所有chunk的该字段是一样的，所以扫描时可以根据此信息对block进行排序
- **extra_obj_type**: 表示该chunk中存储的文件类型，extra_available为1才能使用这个信息

4 关键流程

4.1 挂载流程

①初始化与VFS层的接口:

```
sb->s_op = &yaffs_super_ops;
inode->i_op = &yaffs_dir_inode_operations;
inode->i_fop = &yaffs_dir_operations;
```

②初始化与驱动层的接口:

```
param->write_chunk_tags_fn = nandmtd2_write_chunk_tags;
param->read_chunk_tags_fn = nandmtd2_read_chunk_tags;
param->bad_block_fn = nandmtd2_mark_block_bad;
param->query_block_fn = nandmtd2_query_block;
param->erase_fn = nandmtd_erase_block;
param->initialise_flash_fn = nandmtd_initialise;
```

上面两步等于把VFS、YAFFS2、MTD三层的通道搭建起来。

③初始化yaffs_dev结构:

主要在yaffs_guts_initialise函数完成，具体包含以下部分，

- ◇初始化擦除块信息: yaffs_init_blocks
- ◇初始化cache和buffer: yaffs_init_tmp_buffers
- ◇初始化tnodes和objs: yaffs_init_tnodes_and_objs
- ◇初始化根目录: yaffs_create_initial_dir

④ yaffs2_checkpoint_restore流程:

基础设施搭建完毕，后面就是扫描flash在内存中建立完整的文件视图，YAFFS2首先尝试从checkpoint中恢复文件系统的信息，

- ◇yaffs2_checkpoint_open为读出checkpoint信息做一些初始化工作，首先创建一个checkpoint_buffer，初始化checkpoint_block_list数组；
- ◇yaffs2_rd_checkpoint_validity_marker读取checkpoint头部信息，并判断是否正确；
- ◇yaffs2_rd_checkpoint_dev读取struct yaffs_checkpoint_dev结构的信息，用来填充yaffs_dev结构相应的字段，以及flash上各个擦除块yaffs_block_info信息，还有chunk_bits表示flash上各个chunk位图；
- ◇yaffs2_rd_checkpoint_objs读取各个文件的信息，如果是目录就建立目录结构，如果是文件就建立Tnodes树；
- ◇yaffs2_rd_checkpoint_validity_marker读出checkpoint尾部信息，并判断是否正确；
- ◇yaffs2_rd_checkpoint_sum读出校验和，并和计算的比较是否一致。

⑤ yaffs2_scan_backwards流程:

如果上述5个步骤都没有错误，则从checkpoint挂载成功。否则就扫描整个flash，

- ◇扫描整个flash OOB信息，按照bi->seq_number顺序对block进行排序；

- ◇其次从seq_number最大的block开始往seq_number减小的方向扫描;
 - ◇最后从physical chunk index大的chunk开始, 并向减小的方向扫描;
- 扫描的功能主要由yaffs2_scan_chunk函数完成:
- ◇读取chunk的tags信息, 根据tags.chunk_used判断该chunk是空闲的还是被使用的; 根据tags.chunk_id判断是data chunk, 还是object header
 - ◇如果是data chunk, 分几种情况:
 - a) 根据tags.obj_id查找散列表, 找到object对象, 则把tags.chunk_id转换成文件内的偏移, 拿这个偏移和文件对象的shrink_size比较, 如果在文件的大小范围内则调用yaffs_put_chunk_in_file把该chunk加入到对应的tnodes tree中, 否则删除该chunk; 这一般发生在正常关闭文件的情况下, 也即先扫描到object header, 后扫描到文件的数据 chunk, 这种情况文件的实际大小应该由扫描碰到的第一个object header所记录的大小决定;
 - b) 根据tags.obj_id查找散列表, 不存在则创建新的object对象, 并插入到散列表中, 则把tags.chunk_id转换成文件内的偏移, 拿这个偏移和文件对象的shrink_size比较, 如果在文件的大小范围内则调用yaffs_put_chunk_in_file把该chunk加入到对应的tnodes tree中, 否则删除该chunk; 这一般发生在非正常关闭文件的情况下, 也即先扫描到data chunk, 后扫描到文件的object header, 这种情况文件大小由扫描到的第一个data chunk决定。
 - ◇如果是object header, 分几种情况:
 - a) object header被扫描到, object散列表中并无对应的文件, 那么根据tags.obj_id、tags.extra_obj_type或者tags.obj_id、oh->type创建新的object对象, 并使用object header的信息初始化object对象, 并根据oh->parent_obj_id创建父亲的object对象, 把两者联系起来;
 - b) object header被扫描到, object散列表中已经存在对应的object对象, 且对象已经有关联的object header, 说明这个object header是过时的, 所以就直接删除该chunk;

4.2 打开流程

YAFFS2文件的打开主要由yaffs_iget函数完成:

根据yaffs_obj的信息填充inode:

```
inode->i_size = obj->variant.file_variant.file_size;
inode->i_ino = obj->obj_id;
```

如果是普通文件:

```
inode->i_op = &yaffs_file_inode_operations;
inode->i_fop = &yaffs_file_operations;
inode->i_mapping->a_ops =
    &yaffs_file_address_operations;
```

如果是目录文件:

```
inode->i_op = &yaffs_dir_inode_operations;
inode->i_fop = &yaffs_dir_operations;
```

4.3 读取流程

YAFFS2文件的读取主要由yaffs_readpage函数完成: 根据pg->index计算文件内偏移, 然后转换成logical chunk index, 最后在Tnode tree 中找到相应的physical chunk index, 读取该chunk的数据到页缓存中。

读取的过程涉及到YAFFS2内部的cache, 首先检测读取的数据是边界否chunk对齐, 大小是否data_bytes_per_chunk对齐, 如果边界对齐就不需要内部的cache直接从chunk中读取数据到页缓存中; 否则在cache查找该文件是否有对应的cache, 如果有就直接从cache读取, 没有对应的cache则分配一个cache, 从chunk中读到cache中, 然后再从cache到页缓存中。

4.4 写入流程

YAFFS2文件的写入操作主要由yaffs_file_write函数完成: 根据pg->index计算文件内偏移, 然后转换成logical chunk index; 然后分配空闲chunk, 建立logical chunk index和physical chunk index, 把页缓存中的写入数据到分配到的chunk, 并把该chunk插入到tnode tree。

写入的流程也涉及到YAFFS2内部的cache, 首先检测写入的数据是边界否chunk对齐, 大小是否data_bytes_per_chunk对齐, 如果边界对齐就不需要内部的cache直接从页缓存写入chunk中; 否则在cache查找该文件是否有对应的cache, 如果有就直接写入cache中, 如果cache中的数据达到data_bytes_per_chunk就同步到flash; 没有对应的cache则分配一个cache, chunk中读到cache中, 然后再把数据写入cache中。

4.5 删除流程

YAFFS2文件的删除采取软删除主要由yaffs_unlink函数完成, 对于不同的文件具体划分: 普通文件对应yaffs_del_file函数, 目录文件对应yaffs_del_dir函数。普通文件如果对应的有数据即n_data_chunks大于0, 则把文件移到unlink目录, 并释放内存中tnode tree, in->deleted置1, obj->dirty 置1, obj->unlinked 置1, 并写入一个新的object header, oh->file_size = 0, chunk留待垃圾回收擦除; 如果没有对应的数据则把文件移到deleted目录, 则删除空的tnode tree, 最后释放object, 并写入一个新的object header, oh->file_size = 0。目录文件比较简单, 最后释放object, 并写入一个新的object header表示文件被删除。

4.6 垃圾回收

4.6.1 回收时机

- ①在yaffs_wr_data_obj入口调用yaffs_check_gc进行垃圾回收;
- ②在yaffs_bg_thread_fn垃圾回收线程中进行回收, 该线程会定时被唤醒, 根据剩余空间的多少, 定时间隔会动态调整, 该线程是非实时的, 最后调用yaffs_check_gc进行垃圾回收

4.6.2 回收策略

根据空闲擦除块的数量判断空间是否紧张, 分为如下两种策略:

- ①aggressive garbage collection, 比较紧急的回收, yaffs_find_gc_block找符合条件的擦除块条件比较宽松, 具体体现在以下两个指标的计算上:

threshold = dev->param.chunks_per_block;

iterations = dev->internal_end_block - dev->internal_start_block + 1;

第一个指标表示擦除块dev->gc_pages_in_use的阈值, 这个阈值越大擦除块越好选择, 第二个指标表示选择擦除块的范围, 比较紧急的时候要搜索所有的擦除块找合适的进行回收。

- ②"leisurely" garbage collection, 非紧急的回收, yaffs_find_gc_block找符合条件的擦除块条件比较严格, 具体体现在以下两个指标的计算上:

threshold = background ? (dev->gc_not_done + 2) * 2 : 0;

iterations = (dev->internal_end_block - dev->internal_start_block + 1) / 16 + 1;

第一个指标表示要找全脏的, 无有效chunk的块进行回收, 第二个指标表示只在很小的范围内搜索这样的擦除块。

4.6.3 回收过程

- ①首先选择适合进行回收的擦除块, 由yaffs_find_gc_block来完成, 当block不包含shrink flag时, 则它适合gc; 如果擦除块包含shrink flag标志, 则已完成chunk分配且最早被使用过的block适合gc。
- ②如果该擦除块还有有效的chunk, 则要将正在使用的chunk挪到空闲的block上, 来腾出这

个比较脏的block，由yaffs_gc_process_chunk完成。

③当整个擦除块无有效chunk时，yaffs_block_became_dirty调用yaffs_erase_block把该擦除块擦除掉。

5 调试手段

5.1 打印开关

在yaffs_trace.h定义了一些宏，这些宏控制YAFFS2文件系统的调试打印，可以在YAFFS_TRACE_ALWAYS中添加这些响应的位，比如想观察垃圾回收运行流程的调试打印，可以设置YAFFS_TRACE_ALWAYS为0xf0000040，也可以修改yaffs_yaffs2.c文件的yaffs_trace_Mask打开这些调试选项：

```
#define YAFFS_TRACE_OS 0x00000002
#define YAFFS_TRACE_ALLOCATE 0x00000004
#define YAFFS_TRACE_SCAN 0x00000008
#define YAFFS_TRACE_BAD_BLOCKS 0x00000010
#define YAFFS_TRACE_ERASE 0x00000020
#define YAFFS_TRACE_GC 0x00000040
#define YAFFS_TRACE_WRITE 0x00000080
#define YAFFS_TRACE_TRACING 0x00000100
#define YAFFS_TRACE_DELETION 0x00000200
#define YAFFS_TRACE_BUFFERS 0x00000400
#define YAFFS_TRACE_NANDACCESS 0x00000800
#define YAFFS_TRACE_GC_DETAIL 0x00001000
#define YAFFS_TRACE_SCAN_DEBUG 0x00002000
#define YAFFS_TRACE_MTD 0x00004000
#define YAFFS_TRACE_CHECKPOINT 0x00008000

#define YAFFS_TRACE_VERIFY 0x00010000
#define YAFFS_TRACE_VERIFY_NAND 0x00020000
#define YAFFS_TRACE_VERIFY_FULL 0x00040000
#define YAFFS_TRACE_VERIFY_ALL 0x000f0000

#define YAFFS_TRACE_SYNC 0x00100000
#define YAFFS_TRACE_BACKGROUND 0x00200000
#define YAFFS_TRACE_LOCK 0x00400000
#define YAFFS_TRACE_MOUNT 0x00800000

#define YAFFS_TRACE_ERROR 0x40000000
#define YAFFS_TRACE_BUG 0x80000000
#define YAFFS_TRACE_ALWAYS 0xf0000000
```

5.2 转储工具

Nanddump工具在mtd-util工具套件中可以找到，使用这个工具可以转储出NAND flash的镜像数据，供观察分析，有如下选项：

--help	Display this help and exit
--version	Output version information and exit
-a --forcebinary	Force printing of binary data to tty
-c --canonicalprint	Print canonical Hex+ASCII dump
-f file --file=file	Dump to file
-i --ignoreerrors	Ignore errors
-l length --length=length	Length
-n --noecc	Read without error correction
-o --omitbad	Omit bad blocks from the dump
-b --omitbad	Omit bad blocks from the dump

```
-p          --prettyprint      Print nice (hexdump)
-q          --quiet             Don't display progress and status
messages
-s addr     --startaddress=addr Start address
```

转储的结果如下，可以对照源码的数据结构进行分析：

```
OOB Data: ff 00 00 01 10 00 59 65 56 00 10 10 00 00 00 10
OOB Data: 00 00 80 00 f3 00 cf c3 f3 00 00 00 00 00 50 00
OOB Data: 00 00 50 ff ff ff 65 55 a5 ff ff ff ff ff ff ff
OOB Data: ff ff ff ff ff ff ff 0c fc ff ff ff ff ff ff ff
0x00001000: 83 06 00 08 d3 02 01 02 83 98 e7 44 d3 02 01 02
0x00001010: 83 9a d7 0a 08 fd 00 80 84 40 fc 53 08 f1 00 80
0x00001020: d3 02 01 c2 93 92 99 8c 45 00 01 a3 d7 02 a4 41
0x00001030: 83 00 00 00 09 90 00 00 c3 00 21 10 09 f1 00 81
0x00001040: 08 f1 00 81 06 00 00 d3 09 f1 00 81 84 00 00 c2
0x00001050: 18 f3 00 80 83 90 00 10 09 f1 00 80 d3 02 01 c2
0x00001060: 18 92 09 4a 08 f1 00 80 f7 08 84 04 14 c9 ff 82
0x00001070: 83 00 00 00 09 f1 00 81 08 f1 00 81 c7 30 30 87
```

6 参考资料

- 【1】 Flash 存储管理在嵌入式系统中的实现.pdf
- 【2】 NAND 闪存文件系统 YAFFS 的研究.pdf
- 【3】 yaffs2 源码分析.pdf