

JFFS/JFFS2 文件系统

本文原文由站长编写，人民邮电出版社出版

电子版版权由 RTEMS.net 所有，转载请注明：来源 <http://www.rtems.net>，作者 ray@rtems.net

JFFS 文件系统是瑞典 Axis Communications AB 为嵌入式系统开发的日志文件系统。JFFS1 应用在 Linux2.2 以上版本中，JFFS2 在 Linux2.4 内核和 Ecos 中。Linux 的实现中，JFFS 必须建立在 MTD 驱动程序的上层（如图 7-13 所示）。这里 MTD 的作用是为 JFFS 提供操作 NAND 或者 NOR 芯片的接口，具体例程可以参考 7.3 和 7.4 两节。

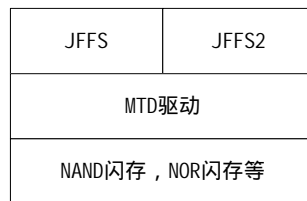


图 7-13 文件系统层次

JFFS 是针对以闪存为存储介质的嵌入式系统，所以充分考虑了闪存的物理局限性，使用了尽可能高效的日志系统。和前面介绍的 TrueFFS 以及其他中间层驱动相比，JFFS 是专门针对闪存的文件系统，这个文件系统除了有日志功能，还包含了前面在 TrueFFS 章节中介绍的负载平衡、垃圾收集等功能。另外一个重要特点是这个文件系统是源代码公开的，方便了学习和使用。此外使用了日志系统，使文件系统更加可靠。

日志文件系统的主要设计思想是跟踪文件系统的变化而不是文件系统的内容。日志文件系统中，存储系统上面有一系列节点记录了对文件的操作。日志节点上面记录的信息包括：

- 和日志节点关联的文件的标示符。
- 日志节点的序列号（version）。
- 当前节点的 uid，gid 等信息。
- 其他关于文件内容分布的信息。

图 7-14 所示说明了日志文件节点的功能

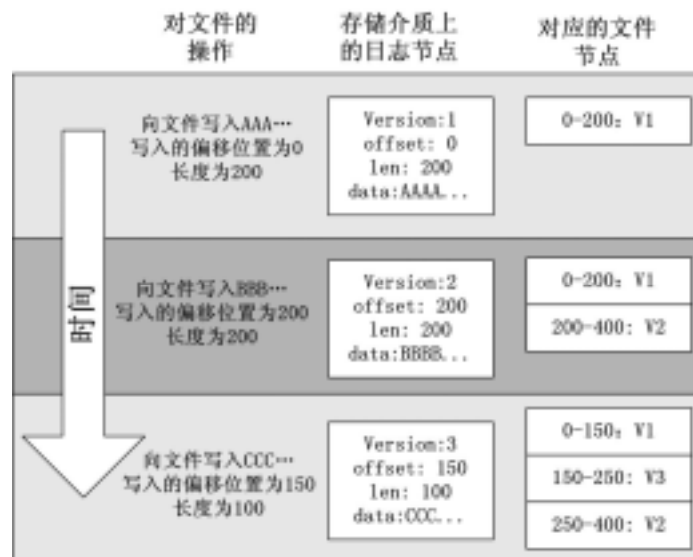


图 7-14 日志文件系统

图 7-14 中，左边是对文件的操作，中间是对应日志节点的变化，右边是操作的目标文件上面和信息的变化。第一次对文件进行操作的时候，在闪存中建立了 version 号为 1 的节点，在节点中记录了在在偏移位置为 0 的地方写入了“AAA.....”长度为 200。此时在对应的文件中建立了文件和一号日志节点的对应关系。第二次对文件的操作是在偏移位置为 200 的地方写入数据“BBBB.....”长度为 200，这时在闪存中建立 version 号为 2 的日志节点并且在文件中建立对应的关联，这时实际对应的文件如该图右边所示。第三次操作在该文件偏移位置为 150 的地方写入“CCC.....”长度为 100，并且建立 3 号节点以及该节点和文件的关联。

从图 7-14 中可以看出，日志节点记录的是对文件作的修改信息，而且还能看出有了日志文件系统，在一定程度上能对文件进行的操作回滚。这里由于对文件的反复操作又可能造成早期的日志节点“过时”，比如在上面的操作中，第四次操作里在 0-200 的位置写入 DDD....，那么日志节点 1 所记录的信息对文件的当前信息没有任何价值，这个时候就将日志节点 1 称为“过时”(脏数据)。文件系统中不断加入新文件，不断对旧文件进行操作，那么文件系统就会被占满；这个时候就需要释放“过时”的日志节点，释放过时日志节点的工作称为“垃圾收集”(注意这里的垃圾收集的概念和 TrueFFS)中的垃圾收集不同。

JFFS 中要进行垃圾收集的时候，不是简单地将过时的节点释放掉就可以的。前面也提到过，NAND Flash 的擦写操作是以块为单位的，所以必须将一个设备块中有用的数据拷贝到文件系统的空闲块中，直到剩下的数据都是过时的数据才进行擦写释放。

1 日志文件系统存储结构

JFFS 是一种纯日志文件系统。linux 中，所谓文件系统是一系列存放在存储介质上的节点。在 JFFS1 中，只有一种日志节点 struct jffs_raw_inode 用于在闪存芯片中存放数据，节点的定义如下：

```
struct jffs_raw_inode
{
    __u32 magic;        /* 魔数 */
    __u32 ino;         /* I 节点编号 */
    __u32 pino;        /* 该节点的父节点编号 */
    __u32 version;     /* Version 号 */
    __u32 mode;        /* 文件类型 */
    __u16 uid;         /* 属主 */
    __u16 gid;         /* 文件属组 */
    __u32 atime;       /* 最后访问时间 */
    __u32 mtime;       /* 最后修改时间 */
    __u32 ctime;       /* 创建时间 */
    __u32 offset;      /* 数据偏移 */
    __u32 dsiz e;      /* 数据长度 */
    __u32 rsiz e;      /* 要删除的数据长度 */
    __u8 nsiz e;       /* 名称长度 */
    __u8 nlink;        /* 文件链接数 */
    __u8 spare : 6;    /* 保留位 */
    __u8 rename : 1;   /* 是否需要更名? */
}
```

```

__u8 deleted : 1;    /* 是否被删除 */
__u8 accurate;    /* 是否是可用的数据 */
__u32 dchksum;    /* 数据校验码 */
__u16 nchksum;    /* 名称校验码 */
__u16 chksum;    /* 节点信息校验码 */
};

```

每个节点中有一个域 `__u32 ino` 和一个 `i` 节点相关联, `__u32 mode` 记录了文件的类型, `jffs_raw_inode` 也包含了一个 `version` 字段, 记录了日志节点的序列号。因为一个 `ino` 可能和多个日志节点相关联, 使用序列号就能将日志节点排序。此外 `uid`, `gid`, `mtime`, `atime`, `mtim` 记录了文件的用户信息和文件的修改信息等。

每个节点也包含了文件的数据信息。如果文件非空, 那么 `jffs_raw_inode` 也会记录数据的偏移量。这主要是因为 JFFS 中一个大文件可能有多个日志节点与之相关, 每个日志节点只管理文件中的一部分数据。

`__u8 accurate` 这个字段是在 JFFS 中实现先写后清零的关键字段。发生写操作的时候, 往往并不是将原来的节点清零, 而是将其标记为脏节点, 也就是将 `accurate` 赋值为 0。



图 7-15 文件在闪存中的存放方式

在闪存中, 数据的存放方式如图 7-15 所示, 紧接在 `jffs_raw_inode` 后面存放的是文件/目录的名称, 长度为 `nsize` (通常一个文件中只有一个 `jffs_raw_inode` 中 `nsize` 不为 0)。在名称后面就是数据域, 这个域的长度使用 `dsize` 来描述。当文件系统加载的时候会扫描整个芯片, 读入 `jffs_raw_inode` 的信息, 然后根据该信息构建文件系统。

发生修改操作的时候, 将分配一个新的 `jffs_raw_inode`, 并且将要写的数据写入对应区域。然后原来存放数据的块标记成脏。当系统无法分配新的节点进行写操作的时候就必须进行垃圾收集。也就是从头开始扫描日志节点, 将标记为脏的节点回收。值得注意的是, 因为闪存的写操作只能以块为单位, 所以如果标记为脏的节点和有效节点同处于一个块中的时候, 必须将有效节点先移到其他块中。这里涉及到一些算法, 就不做详细的介绍了。

文件加载的时候, 在内存中使用 `struct jffs_file` 表示, 每个文件的数据实际存放在一个以 `range_head` 开头, 以 `range_tail` 结尾的 `jffs_node` 列表中。结构 `jffs_node` 的定义如下:

```

struct jffs_node
{
    __u32 ino;        /* 节点编号 */
    __u32 version;    /* Version 编号 */
    __u32 data_offset; /* I 节点记录的数据块在文件中的偏移 */
    __u32 data_size;  /* 数据长度 */
    __u32 removed_size; /* I 节点记录的要清除的数据长度 */
    __u32 fm_offset;  /* 数据的物理地址 */
};

```

```

__u8 name_size; /* 名称长度 */
struct jffs_fm *fm; /* 闪存的物理信息 */
struct jffs_node *version_prev;
struct jffs_node *version_next;
struct jffs_node *range_prev;
struct jffs_node *range_next;
};

```

从上面的数据结构可以看出每个节点包含了节点号,节点内数据多少,节点到物理闪存的映射等信息。对应的 jffs_file 内 jffs_node 的组织如图 7-16 所示,可以看出所有的 jffs_node 组成一个链表,

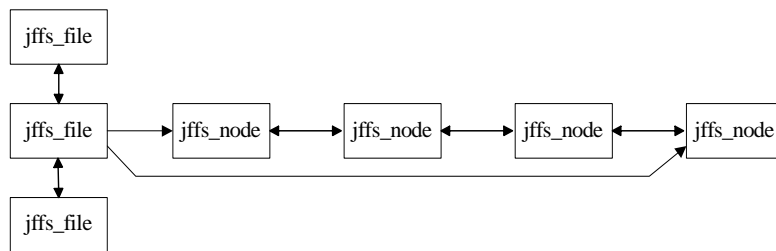


图 7-16 JFFS 数据结构示意图

此外 jffs 中包括了一个“ version ”号,这个号是一个序列号,每个文件的第一个 jffs_node 的号是 1,此后每增加一个节点,序列号加 1。removed_size 字段表示了将被删除的数据大小,这个常常用在减少文件大小的时候。

jffs_node 中,和物理存储设备直接关联的是一个 struct jffs_fm 类型的字段。这个结构体记录了一个闪存的存储块,包括数据在文件中的偏移,数据块大小。所有的闪存块是由 struct jffs_fmcontrol 统一管理的。

2 JFFS2

JFFS2 是 Redhat 公司基于 JFFS 开发的闪存文件系统,它主要是针对 RedHat 公司的嵌入式产品 eCos 开发的嵌入式文件系统,当然 JFFS2 也可以使用在 Linux, ucLinux 中。JFFS2 克服了 JFFS 中的一些缺点:

- 使用了基于哈希表的日志节点结构,大大加快了对节点的操作速度。
- 支持数据压缩。
- 提供了“写平衡”支持。
- 支持多种节点类型(数据 I 节点,目录 I 节点等);而 JFFS 只支持一种节点。
- 提高了对闪存的利用率,降低了内存的消耗。

JFFS2 中定义了多种节点,但是每种节点都包含下面的信息:

```

struct jffs2_unknown_node
{
    __u16 magic; /*作为 nodetype 的补充*/
    __u16 nodetype; /*节点类型*/
    __u32 totlen; /* 节点总长度 */
    __u32 hdr_crc; /*CRC 校验码*/
};

```

如图 7-17 可以看出这些数据在存储器中整齐地排列:

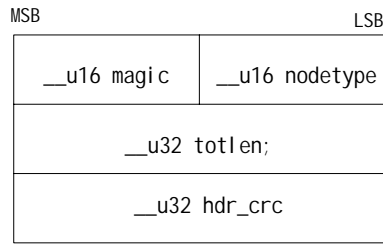


图 7-17 数据结构内存表示

这里 magic 的最左边两位用来表示节点类型，作为对 nodetype 的补充，表示的类型包括：

- JFFS2_FEATURE_INCOMPAT 不兼容节点
- JFFS2_FEATURE_ROCOMPAT 可以进行节点不支持的操作，但是不能对节点进行写操作。
- JFFS2_FEATURE_RWCOMPAT_DELETE 如果是不支持的节点并且含有这个标示位，表示节点已经无用，可以删除
- JFFS2_FEATURE_RWCOMPAT_COPY 如果是不支持的节点，并且含有这个标示位，表示节点可以被写入。

1. 目录节点的定义如下：

```

struct jffs2_raw_dirent
{
    __u16 magic;
    __u16 nodetype; /* 节点类型，设置为 JFFS2_NODETYPE_DIRENT */
    __u32 totlen;
    __u32 hdr_crc; /*jffs2_unknown_node 部分的 CRC 校验*/
    __u32 pino; /*上层目录节点的编号*/
    __u32 version;
    __u32 ino; /* 节点编号，如果是 0 表示没有链接的节点*/
    __u32 mctime; /*创建时间*/
    __u8 nsize; /*大小*/
    __u8 type;
    __u8 unused[2];
    __u32 node_crc; /*校验码*/
    __u32 name_crc;
    __u8 name[0]; /*名称*/
};

```

数据结构中最后的一个字段是长度为 0 的名称字段，这并没有为 name 分配空间，name 的实际存放在 jffs2_raw_dirent 的后面，长度为 nsize。

2. 数据节点：

```

struct jffs2_raw_inode
{
    __u16 magic;
    __u16 nodetype; /*设置为 JFFS2_NODETYPE_INODE*/
    __u32 totlen; /*节点的总长度（包括有效数据）*/
    __u32 hdr_crc; /*jffs2_unknown_node 部分的 CRC 校验*/
    __u32 ino;
};

```

```

__u32 version;
__u32 mode;      /*文件的类型*/
__u16 uid;
__u16 gid;
__u32 isize;    /*实际长度*/
__u32 atime;
__u32 mtime;
__u32 ctime;
__u32 offset;   /*对应数据在文件中的起始位置 */
__u32 csize;    /*压缩数据的长度 */
__u32 dsize;    /*数据有效长度 */
__u8 compr;     /*当前压缩算法 */
__u8 usercompr; /*用户指定的压缩算法 */
__u16 flags;    /*标志位 */
__u32 data_crc; /*数据校验码 */
__u32 node_crc; /*头节点的交验码*/
}

```

和 JFFS1 中定义的数据节点类似，但是下面的几个字段不同：

- 没有父亲节点编号，没有文件名称。
- 对节点作了优化，能放在一个页面里面
- 增加了压缩功能。

3. 可靠性支持：

对闪存进行擦写操作的时候如果突然掉电，可能会有一部分数据没有擦写干净。为了解决这个问题，JFFS2 对块操作的时候，如果操作成功，会在块的开始做上标记，通过这个标记表明块内的数据处于一致状态。

4. 内存使用：

JFFS2 中 I 节点的信息并没有全部存放在内存里面。mount 操作时，会为节点建立映射表，但是这个映射表并不全部存放在内存里面，存放在内存中的节点信息是一个缩小尺寸的结构体——struct jffs2_raw_node_ref，它的定义如下：

```

struct jffs2_raw_node_ref
{
    struct jffs2_raw_node_ref *next_in_ino; /* 链表指针*/
    struct jffs2_raw_node_ref next_phys; /*在物理上相邻的块*/
    __u32 flash_offset; /*在 Flash 块中的偏移，一般为 0*/
    __u32 totlen;
};

```

jffs2_raw_node_ref 信息在内存中通过 jffs2_inode_cache 结构进行管理：

```

struct jffs2_inode_cache {
    struct jffs2_scan_info *scan; /* 在扫描链表的时候存放临时信息，在扫描结束以后设置成 NULL*/
    struct jffs2_inode_cache *next;
    struct jffs2_raw_node_ref *nodes;
    __u32 ino;
    int nlink; /*和当前 I 节点链接的节点数目*/
};

```

内存中 jffs2_inode_cache 和 jffs2_raw_node_ref 的关系如图 7-18 所示：

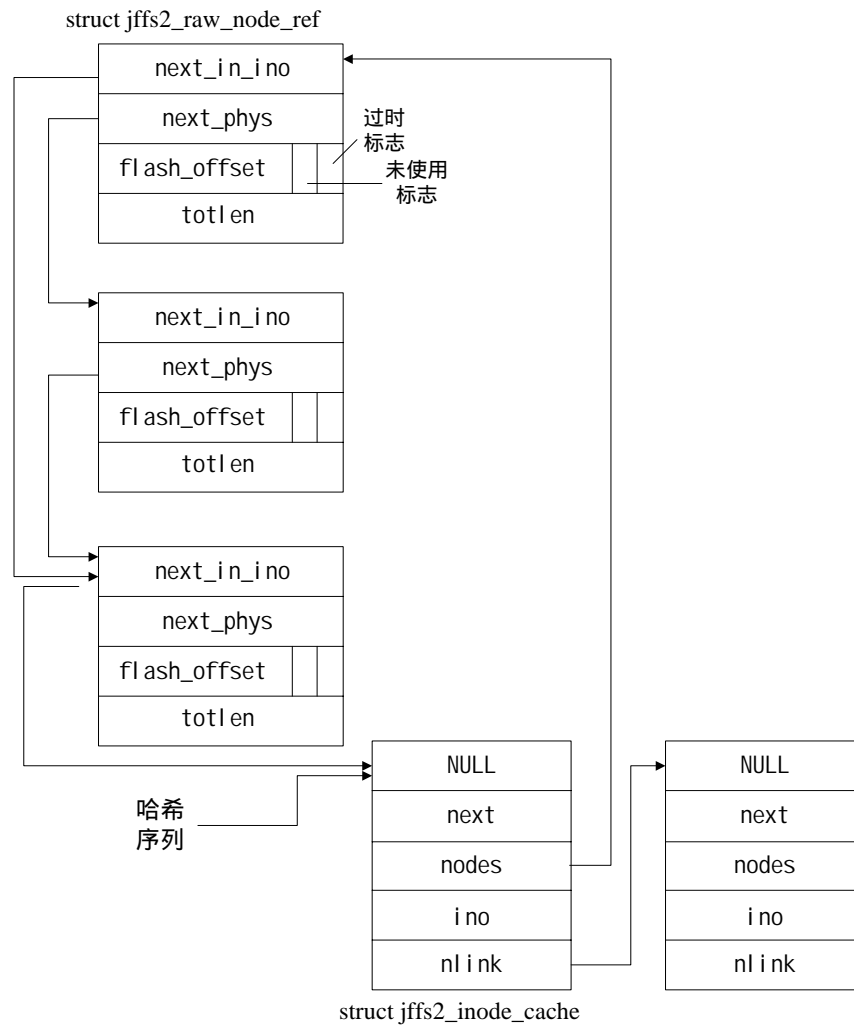


图 7-18 jffs2_inode_cache 和 jffs2_raw_node_ref 关系示意

系统中使用结构体 jffs2_sb_info 来管理所有的节点链表和闪存块，这个结构相当于 UNIX/Linux 系统中的超级块，定义如下：

```

struct jffs2_sb_info {
    struct mtd_info *mtd;
    __u32 highest_ino;
    unsigned int flags;
    spinlock_t nodelist_lock;
    struct task_struct *gc_task; /* 垃圾收集任务指针 */
    struct semaphore gc_thread_start; /* 垃圾收集线程使用的互斥变量*/
    struct completion gc_thread_exit; /* 垃圾收集结束的信号量 */
    struct semaphore alloc_sem; /* 用于在垃圾收集的时候保护数据*/
    __u32 flash_size; /*Flash 相关信息*/
    __u32 used_size;
    __u32 dirty_size;
    __u32 free_size;
    __u32 erasing_size;
}
    
```

```

__u32 bad_size;
__u32 sector_size;
__u32 nr_free_blocks;
__u32 nr_erasing_blocks;
__u32 nr_blocks;
struct jffs2_eraseblock *blocks;      /* 存放所有块的数组头指针 */
struct jffs2_eraseblock *nextblock;   /* 目前正在写入的块 */

struct jffs2_eraseblock *gcblock;     /* 目前正在进行垃圾收集的块 */

    struct list_head clean_list;       /* 包含所有数据都是正确
数据的块（干净块）的链表 */
    struct list_head dirty_list;       /* 包含有脏数据的链表 */
    struct list_head erasing_list;     /* 正在擦写的块的链表 */
    struct list_head erase_pending_list; /* 需要擦写的块的链表 */
    struct list_head erase_complete_list; /* 完成擦写的块的链表，
准备做“干净”标志*/
    struct list_head free_list;        /* 空闲块链表，所有的块都
可以被用来存放数据 */
    struct list_head bad_list;         /* 不可用的块的链表，闪存中的坏块 */
    struct list_head bad_used_list;    /* 不可用块的链表，但是里
面有数据，只能读，不能写 */
    spinlock_t erase_completion_lock;  /* 对链表操作的互斥变量 */
    wait_queue_head_t erase_wait;     /* 等待擦写结束的信号量 */
    struct jffs2_inode_cache *inocache_list[INOCACHE_HASHSIZE]; /*内存 i 节点哈希表*/
    spinlock_t inocache_lock;
};

```

在这里可以看到 `struct jffs2_inode_cache*inocache_list[INOCACHE_HASHSIZE]` 字段就是我们刚才所说的内存节点链表。此外为了管理方便，JFFS2 还有其他的链表，在程序中用斜体表示出来，它们的用途可以参考对应注释。

5. 垃圾收集

JFFS2 使用了多个级别的待回收块队列。在垃圾收集的时候先看 `bad_used_list` 链表中是否有节点，如果有，先回收这个链表的节点，因为这个链表中的节点由于闪存的物理原因很快要失效了。做完了 `bad_used_list` 链表的回收，然后回收 `dirty_list` 链表。垃圾收集操作的主要工作是将数据块里面的有效数据移动到空闲块中，然后清除脏数据块，最后将数据块从 `dirty_list` 链表中摘除，并且放入空闲块链表。此外可以回收的队列还包括 `erasable_list`、`very_dirty_list` 等。`jffs2_sb_info` 中也有几个字段用于对垃圾收集线程进行互斥控制。由于 JFFS2 中使用了多种节点，所以在进行垃圾收集的时候也必须对不同的节点进行不同的操作。JFFS2 进行垃圾收集时也对闪存文件系统中的不连续数据块进行整理。垃圾收集的详细代码请读者参考 JFFS2 源代码中的文件 `gc.c`，这里就不详细描述了。

6. 写平衡

JFFS2 中写平衡策略是在垃圾收集中实现的，垃圾收集的时候会读取系统时间，通过系统时间产生一个伪随机数。使用这个伪随机数结合不同的待回收链表选择要进行回收的链表。使用了这个平衡策略以后能提供较好的写平衡效果。

