jffs 文件系统分析

Copyright . 2003 by 郭健
E-mail:guo1975@163.net
uclinux-2.4
Version 1.0.0，2003-9-28

摘要：本文主要分析了 uclinux 2.4 内核的 jffs 文件系统机制。希望能对基于 uclinux 开发产品的广大工程师有所帮助。

关键词：uclinux　vfs jffs

欢迎各位指出文档中的错误与疑问

一、flash 读写的特殊性

对于嵌入式系统，flash 是很常见的一种设备，而大部分的嵌入式系统都是把文件系统建立在 flash 之上，由于对 flash 操作的特殊性，使得在 flash 上的文件系统和普通磁盘上的文件系统有很大的差别，对 flash 操作的特殊性包括：

（1） 不能对单个字节进行擦除，最小的擦写单位是一个 block，有时候也称为一个扇区。典型的一个 block 的大小是 64k。不同的 flash 会有不同，具体参考 flash 芯片的规范。

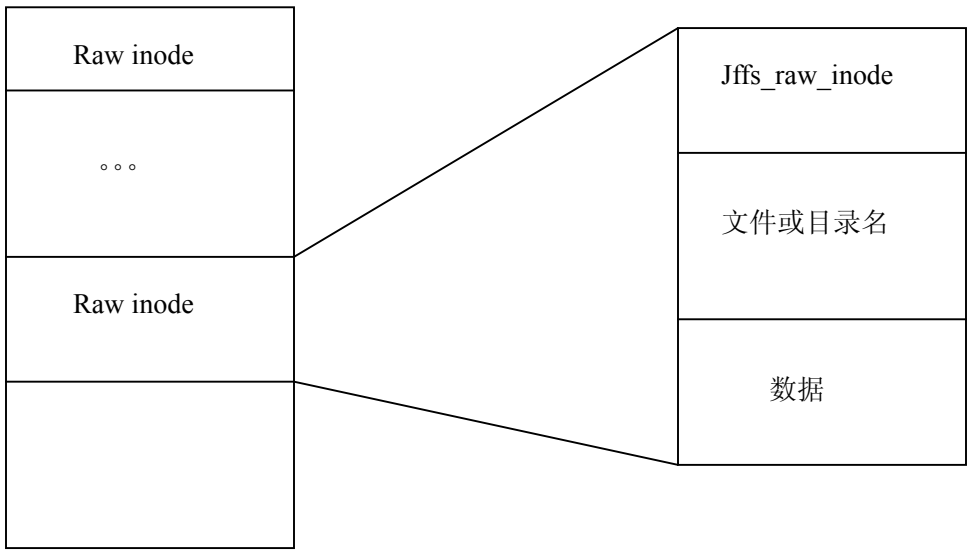
（2） 写操作只能对一个原来是空（也就是该地址的内容是全 f）的位置操作，如果该位置非空，写操作不起作用，也就是说如果要改写一个原来已经有内容的空间，只能是读出该 sector 到 ram，在 ram 中改写，然后写整个 sector。

由于这些特殊写，所以在 flash 这样的设备上建立文件也有自己独特的特点，下面我们就以 jffs 为例进行分析。

二、jffs 体系结构介绍

1、存储结构

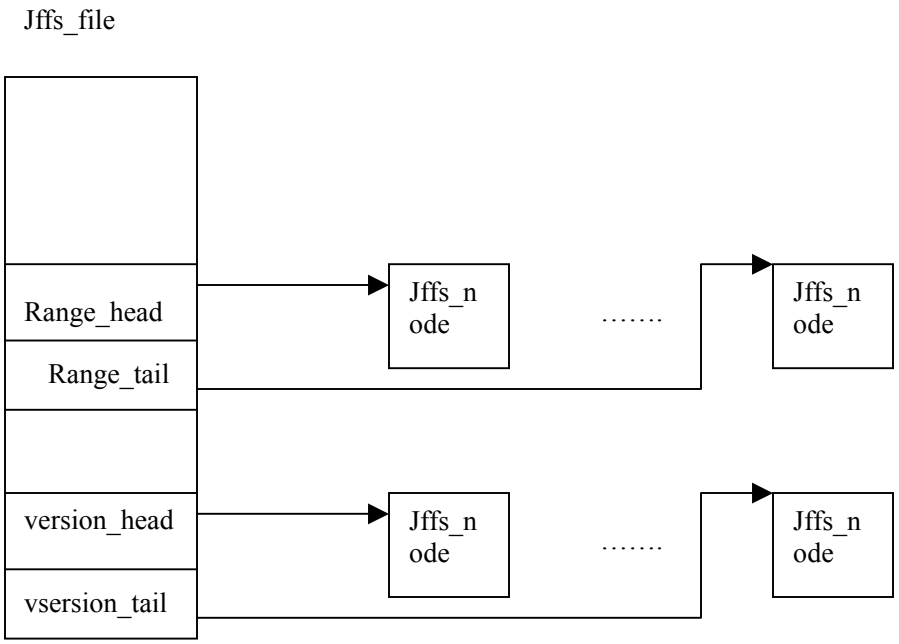
在 jffs 中，所有的文件和目录是一样对待的，都是用一个 jffs_raw_inode 来表示



整个 flash 上就是由一个一个的 raw inode 排列组成，一个目录只有一个 raw inode，对于文件则是由一个或多个 raw inode 组成。

2、文件组成

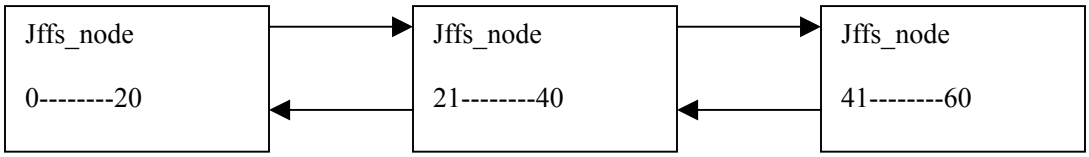
在文件系统 mount 到 flash 设备上的时候，会扫描 flash，从而根据 flash 上的所有属于一个文件的 raw inode 建立一个 jffs_file 结构以及 node list。
下面的图显示了一个文件的组成



一个文件是由若干个jffs_node组成，每一个jffs_node是根据flash上得jffs_raw_inode而建立的，jffs_file主要维护两个链表
版本链表：主要是描述该 node 创建的早晚，就是说 version_head 指向的是一个最老的 node，也就意味着垃圾回收的时候最该回收的就是这个最老的 node。
区域链表：这个链表主要是为读写文件创建的，version_head 指向的 node 代表的文件数据区域是 0~~~n-1 之后依次的节点分别是 n~~~m-1 m~~~~o-1 …….其中 n<m<o …….。当你从文件的 0 位置读的话，那么定位到 verision_head 指向的第一个 node，当你从文件的中间位置读的话，那么定位到区域链表的某一个中间的 node 上。

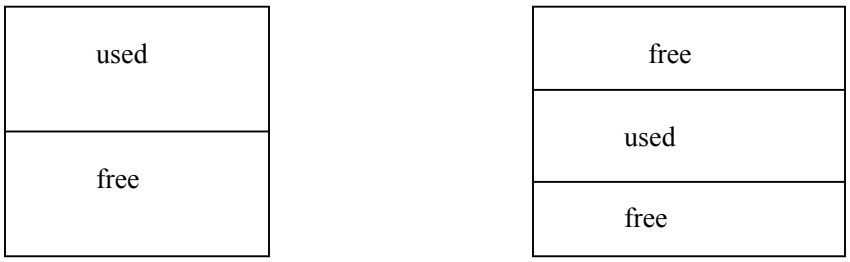
3、操作

对文件的读操作应该是比较简单，但是写操作，包括更改文件名等操作都是引起一个新的 jffs_node 的诞生，同时要写一个相映的 raw inode 到 flash 上，这样的操作有可能导致前面的某个 jffs_node 上面的数据完全失效，从而导致对应 flash 上的 raw inode 的空间成为 dirty。
下面举一个例子可能会更清楚一些。



一个文件的 range list 是由上面的三个 jffs_node 组成，当我们做如下写操作的时候
lseek(fd, 10, SEEK_SET);
write(fd, buf,40);
第一个和最后一个 node 被截短了，第二个 node 完全被新数据替换，该 node 会从链表上摘下来，flash 上空间变成 dirty。如果做如下写操作的时候
lseek(fd, 23, SEEK_SET);
write(fd, buf,5);
此时，第二个 node 被分裂成两个 node，同时产生一个新的 node，range 链表的元素变成五个。

4、垃圾回收



我们的 flash 上的内容基本上是有两种情况，一种是前面一段是 used，后面是 free 的，还有一个是中间一段是 used，flash 的开始和底部都是 free 的，但是不管如何，如果符合了垃圾回收的条件，就要启动垃圾回收。Used 的空间中，有一部分是我们真正需要的数据，还有一部分由于我们的对文件的写，删除等操作而变成 dirty 的空间，我们垃圾回收的目标就是把这些 dirty 的空间变成 free 的，从而可以继续使用。

Used 的空间是有一个个的 jffs_fm 的链表组成，垃圾回收也总是从 used 的最顶部开始，如果 jffs_fm 不和任何的 jffs_node 相关，那么我们就认为 jffs_fm 代表的这块 flash 空间是 dirty 的，找到了完整的至少一个 sector 的 dirty 空间，就可以把这个 sector 擦掉，从而增加一个 sector 的 free 空间。

三、数据结构分析

这些结构不会是每一个成员变量都作解释，有的英语注释说的很清楚了，有些会在下面的相关的代码解释

1、struct jffs_control

```
/* A struct for the overall file system control.  Pointers to
   jffs_control structs are named `c' in the source code.  */
struct jffs_control
{
    struct super_block *sb;          /* Reference to the VFS super block.  */
    struct jffs_file *root;      /* The root directory file.  */
    struct list_head *hash;          /* Hash table for finding files by ino.  */
    struct jffs_fmcontrol *fmc;    /* Flash memory control structure.  */
    __u32 hash_len;              /* The size of the hash table.  */
    __u32 next_ino;              /* Next inode number to use for new files.  */
    __u16 building_fs;           /* Is the file system being built right now?  */
    struct jffs_delete_list *delete_list; /* Track deleted files.  */
    pid_t thread_pid;       /* GC thread's PID */
    struct task_struct *gc_task;    /* GC task struct */
    struct completion gc_thread_comp; /* GC thread exit mutex */
    __u32 gc_minfree_threshold;  /* GC trigger thresholds */
    __u32 gc_maxdirty_threshold;
    __u16 gc_background;         /* GC currently running in background */
};
```

解释：

（1）为了快速由 inode num 找到文件的 struct jffs_file 结构，所以建立了长度为 hash_len 的哈西表，hash 指向了该哈西表

（2）在 jffs 中，不论目录还是普通文件，都有一个 struct jffs_file 结构表示，成员变量 root 代表根文件。

（3）成员变量 delete_list 是为了删除文件而建立，只是在将文件系统 mount 到设备上而扫描 flash 的时候使用。

（4）文件号最小是 1，分配给了根，此后，每创建一个文件 next_no 就会加一。当文件删除之后，也不会回收文件号，毕竟当 next_no 到达最大值的时候，flash 恐怕早就挂拉。

2、struct jffs_fmcontrol

很显然，这是一个描述整个 flash 使用情况的结构

```
struct jffs_fmcontrol
{
    __u32 flash_size;
    __u32 used_size;
```

```
    __u32 dirty_size;
    __u32 free_size;
    __u32 sector_size;
    __u32 min_free_size;    /* The minimum free space needed to be able
                               to perform garbage collections.    */
    __u32 max_chunk_size; /* The maximum size of a chunk of data.    */
    struct mtd_info *mtd; //指向 mtd 设备
    struct jffs_control *c;
    struct jffs_fm *head;
    struct jffs_fm *tail;
    struct jffs_fm *head_extra;
    struct jffs_fm *tail_extra;
    struct semaphore biglock;
};
```

解释：

（1）整个 flash 上的空间=flash_size，已经使用了 used_size 的空间，在 used_size 中一共有 dirty_size 是 dirty 的，dirty 也就是说在垃圾回收的时候可以回收的空间，free_size 是你能够使用的 flash 上的空间

（2）整个 flash 上的所有 used_size 是通过一个 struct jffs_fm 的链表来管理的，head 和 tail 分别指向了最老和最新的 flash chunk

（3）head_extra 和 tail_extra 是在扫描 flash 的时候使用

（4）jffs 中，对一个节点的数据块的大小是有限制的，最大是 max_chunk_size

3、struct jffs_fm

```
/* The struct jffs_fm represents a chunk of data in the flash memory.    */
struct jffs_fm
{
    __u32 offset;                   //在 flash 中的偏移
    __u32 size;                     //大小
    struct jffs_fm *prev;     //形成双向链表
    struct jffs_fm *next;
    struct jffs_node_ref *nodes; /* USED if != 0.    */
};
```

解释：

（1）由于对文件的多次读写，一个 struct jffs_fm 可能会属于多个 struct jffs_node 结构，所以成员变量 nodes 代表了所有属于同一个 jffs_fm 的 jffs_node 的链表

（2）如果 nodes==NULL，说明该 jffs_fm 不和任何 node 关联，也就是说该 fm 表示的区域是 dirty 的。

4、struct jffs_node

不论文件或是目录，flash 上都是用 jffs_raw_inode 来表示，而 struct jffs_node 则是其在内存中的体现

```
/* The RAM representation of the node.    The names of pointers to
```

```
    jffs_nodes are very often just called `n' in the source code.    */
struct jffs_node
{
    __u32 ino;                  /* Inode number.    */
    __u32 version;              /* Version number.    */
    __u32 data_offset;    /* Logic location of the data to insert.    */
    __u32 data_size;        /* The amount of data this node inserts.    */
    __u32 removed_size; /* The amount of data that this node removes.    */
    __u32 fm_offset;        /* Physical location of the data in the actual
                               flash memory data chunk.    */
    __u8 name_size;          /* Size of the name.    */
    struct jffs_fm *fm; /* Physical memory information.    */
    struct jffs_node *version_prev;
    struct jffs_node *version_next;
    struct jffs_node *range_prev;
    struct jffs_node *range_next;
};
```

解释：

（1）每一次对文件的写操作都会形成一个新的 version 的节点，成员变量 version 表明了该节点的版本号，创建第一个 node 的时候，verion = 1,此后由于对文件的写操作，而创建新的 node 的时候,version 就会加一。同文件号的道理一样，version 也不会回收。

（2）一个文件是由若干节点组成，这些节点组成双象链表，所以该结构中的 struct jffs_node *得成员变量都是为这些双向链表而设立的

（3）data_offset 是逻辑偏移，也就是文件中的偏移，而 fm_offset 表明该节点的数据在 jffs_fm 上的偏移

5、struct jffs_file

该结构代表一个文件或者目录

```
/* The RAM representation of a file (plain files, directories,
    links, etc.).    Pointers to jffs_files are normally named `f'
    in the JFFS source code.    */
struct jffs_file
{
    __u32 ino;       /* Inode number.    */
    __u32 pino;      /* Parent's inode number.    */
    __u32 mode;      /* file_type, mode    */
    __u16 uid;       /* owner    */
    __u16 gid;       /* group    */
    __u32 atime;    /* Last access time.    */
    __u32 mtime;    /* Last modification time.    */
    __u32 ctime;    /* Creation time.    */
    __u8 nsize;      /* Name length.    */
    __u8 nlink;      /* Number of links.    */
    __u8 deleted; /* Has this file been deleted?    */
```

```
    char *name;      /* The name of this file; NULL-terminated.    */
    __u32 size;      /* The total size of the file's data.    */
    __u32 highest_version; /* The highest version number of this file.    */
    struct jffs_control *c;
    struct jffs_file *parent;      /* Reference to the parent directory.    */
    struct jffs_file *children; /* Always NULL for plain files.    */
    struct jffs_file *sibling_prev; /* Siblings in the same directory.    */
    struct jffs_file *sibling_next;
    struct list_head hash;       /* hash list.    */
    struct jffs_node *range_head;      /* The final data.    */
    struct jffs_node *range_tail;      /* The first data.    */
    struct jffs_node *version_head; /* The youngest node.    */
    struct jffs_node *version_tail; /* The oldest node.    */
};
```

解释：

（1）一个文件是由一系列不同的版本的节点组成的，而 highest_version 是最高版本。

（2）一个文件维护两个双向链表，一个反映版本的情况，一个反映文件的区域，version_head 和 version_tail 分别指向了最老和最新的节点，range_head 指向文件中逻辑偏移为 0 的节点，沿着该链表，可以读出整个文件的内容。

（3）在 jffs 中，所有的文件形成一个树，树的根是 jffs_control 结构中的 root，它是唯一的。通过每个 jffs_file 中的 parent，children，sibling_prev，sibling_next 指针可以把所有文件（包括目录）形成一个树

6、struct jffs_raw_inode

这是真正写到 flash 上的一个表示文件（目录）的一个节点的结构

```
/* The JFFS raw inode structure: Used for storage on physical media.    */
/* Perhaps the uid, gid, atime, mtime and ctime members should have
    more space due to future changes in the Linux kernel. Anyhow, since
    a user of this filesystem probably have to fix a large number of
    other things, we have decided to not be forward compatible.    */
struct jffs_raw_inode
{
    __u32 magic;          /* A constant magic number.    */
    __u32 ino;            /* Inode number.    */
    __u32 pino;           /* Parent's inode number.    */
    __u32 version;       /* Version number.    */
    __u32 mode;           /* The file's type or mode.    */
    __u16 uid;            /* The file's owner.    */
    __u16 gid;            /* The file's group.    */
    __u32 atime;          /* Last access time.    */
    __u32 mtime;          /* Last modification time.    */
    __u32 ctime;          /* Creation time.    */
    __u32 offset;        /* Where to begin to write.    */
    __u32 dsize;          /* Size of the node's data.    */
    __u32 rsize;          /* How much are going to be replaced?    */
```

```
    __u8 nsize;             /* Name length.    */
    __u8 nlink;             /* Number of links.    */
    __u8 spare : 6;      /* For future use.    */
    __u8 rename : 1;    /* Rename to a name of an already existing file?    */
    __u8 deleted : 1; /* Has this file been deleted?    */
    __u8 accurate;        /* The inode is obsolete if accurate == 0.    */
    __u32 dchksum;        /* Checksum for the data.    */
    __u16 nchksum;        /* Checksum for the name.    */
    __u16 chksum;         /* Checksum for the raw inode.    */
};
```

四、jffs 的挂接

1、定义jffs文件系统

```
static DECLARE_FSTYPE_DEV(jffs_fs_type, "jffs", jffs_read_super);
```

2、注册文件系统

```
tatic int __init init_jffs_fs(void)
```

这个函数主要是建立 struct jffs_fm 和 struct jffs_node 的专用的缓冲区队列，然后通过 register_filesystem(&jffs_fs_type)注册 jffs 文件系统。

3、read super

当通过命令 mount -t jffs /dev/mtdblock0 /mnt/flash 将文件系统 mount 到设备上的时候，通过 sys_mount 系统调用进入内核，并通过具体的文件系统的 read_super 函数建立起 vfs 的各种数据结构。

```
/* Called by the VFS at mount time to initialize the whole file system.    */
static struct super_block * jffs_read_super(struct super_block *sb, void *data, int silent)
{
    kdev_t dev = sb->s_dev;
    struct inode *root_inode;
    struct jffs_control *c;
//jffs 文件系统要求 mount 的设备必须是 mtd
    if (MAJOR(dev) != MTD_BLOCK_MAJOR) {
        printk(KERN_WARNING "JFFS: Trying to mount a "
                "non-mtd device.\n");
        return 0;
    }
```

```
    sb->s_blocksize = PAGE_CACHE_SIZE;          //设定块的大小
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->u.generic_sbp = (void *) 0;
    sb->s_maxbytes = 0xFFFFFFFF;       //Maximum size of the files

//通过 jffs_build_fs 扫描整个 flash，然后通过 flash 上的内容建立完整的文件树，对于 jffs 文件系统，所有的文件都在 ram 中有对应的结构，不论该文件是否打开
    /* Build the file system.    */
    if (jffs_build_fs(sb) < 0) {       //该函数下面具体分析
        goto jffs_sb_err1;
    }

    /*
     * set up enough so that we can read an inode
     */
    sb->s_magic = JFFS_MAGIC_SB_BITMASK;       //设置文件系统魔术
    sb->s_op = &jffs_ops;                                  //设置 super block 的操作方法

//jffs 文件系统最小的 inode number 是 JFFS_MIN_INO=1，这里建立根的 inode 结构
//对于一个表示 jffs 文件的 inode 结构，inode->u.generic_ip 是指向一个表示该文件的 struct jffs_file 结构。通过 jffs_read_inode，可以将根的 inode 设置好，包括上面的 inode->u.generic_ip，还有 inode->i_op inode->i_fop
    root_inode = iget(sb, JFFS_MIN_INO);
    if (!root_inode)
            goto jffs_sb_err2;

//这里建立根的 dentry 结构
    /* Get the root directory of this file system.    */
    if (!(sb->s_root = d_alloc_root(root_inode))) {
        goto jffs_sb_err3;
    }

//获得 sb 中 jffs_control 的指针
    c = (struct jffs_control *) sb->u.generic_sbp;

    /* Set the Garbage Collection thresholds */
//当 flash 上的 free size 小于 gc_minfree_threshold 的时候，会启动垃圾回收，以便释放一些空间
    /* GC if free space goes below 5% of the total size */
    c->gc_minfree_threshold = c->fmc->flash_size / 20;

    if (c->gc_minfree_threshold < c->fmc->sector_size)
        c->gc_minfree_threshold = c->fmc->sector_size;
//当 flash 上的 dirty size 大于 gc_maxdirty_threshold 的时候，会启动垃圾回收，以便
```

释放一些空间

```
    /* GC if dirty space exceeds 33% of the total size. */
    c->gc_maxdirty_threshold = c->fmc->flash_size / 3;

    if (c->gc_maxdirty_threshold < c->fmc->sector_size)
        c->gc_maxdirty_threshold = c->fmc->sector_size;

//启动垃圾回收的内核线程
    c->thread_pid = kernel_thread (jffs_garbage_collect_thread,
                                   (void *) c,
                                   CLONE_FS | CLONE_FILES | CLONE_SIGHAND);

    return sb;
}
```

4、初始化fs，建立文件树

```
/* This is where the file system is built and initialized.    */
int jffs_build_fs(struct super_block *sb)
{
    struct jffs_control *c;
    int err = 0;

//创建 jffs_control 和 jffs_fmcontrol 结构，并初始化 jffs_control 中的哈西表，根据 mount 的 mtd 设备，初始化 jffs_fmcontrol
    if (!(c = jffs_create_control(sb->s_dev))) {
        return -ENOMEM;
    }
    c->building_fs = 1;         //标示目前正在 building fs
    c->sb = sb;

    //通过 jffs_scan_flash 扫描整个 flash，建立相关的 fs 的结构，下面会详细分析
    if ((err = jffs_scan_flash(c)) < 0) {
        if(err == -EAGAIN){
            //如果发现 flipping bits，则重新扫描，所谓 flipping bits 是由于在 erase sector 的时候，突然断电而造成 flash 上该扇区内容不确定
            jffs_cleanup_control(c);       //清除发现 flipping bits 之前创建的结构
            if (!(c = jffs_create_control(sb->s_dev))) {
                return -ENOMEM;
            }
            c->building_fs = 1;
            c->sb = sb;

            if ((err = jffs_scan_flash(c)) < 0) {     //重新扫描
                goto jffs_build_fs_fail;
            }
```

```
        }else{
            goto jffs_build_fs_fail;
        }
    }

//在 flash 上有所有文件和目录的 jffs_raw_inode 结构，但是没有根文件的结点，所以我们一般要通过 jffs_add_virtual_root 手动创建根文件的相关结构。jffs_find_file 是通过 inode number 在哈西表中查找该 jffs_file
    if (!jffs_find_file(c, JFFS_MIN_INO)) {
        if ((err = jffs_add_virtual_root(c)) < 0) {
            goto jffs_build_fs_fail;
        }
    }
//由于各种原因，扫描结束后，可能有些文件是要删除的，下面的代码执行删除任务
    while (c->delete_list) {
        struct jffs_file *f;
        struct jffs_delete_list *delete_list_element;

        if ((f = jffs_find_file(c, c->delete_list->ino))) {
            f->deleted = 1;
        }
        delete_list_element = c->delete_list;
        c->delete_list = c->delete_list->next;
        kfree(delete_list_element);
    }

//有些节点被标记 delete，那么我们要去掉这些 deleted nodes
    if ((err = jffs_foreach_file(c, jffs_possibly_delete_file)) < 0) {
        printk(KERN_ERR "JFFS: Failed to remove deleted nodes.\n");
        goto jffs_build_fs_fail;
    }
//去掉 redundant nodes
    jffs_foreach_file(c, jffs_remove_redundant_nodes);

//从扫描的所有的 jffs_node 和 jffs_file 结构建立文件树
    if ((err = jffs_foreach_file(c, jffs_insert_file_into_tree)) < 0) {
        printk("JFFS: Failed to build tree.\n");
        goto jffs_build_fs_fail;
    }
//根据每一个文件的版本链表，建立文件的区域链表
    if ((err = jffs_foreach_file(c, jffs_build_file)) < 0) {
        printk("JFFS: Failed to build file system.\n");
        goto jffs_build_fs_fail;
    }
```

```
//建立 vfs 和具体文件系统的关系
    sb->u.generic_sbp = (void *)c;
    c->building_fs = 0;         //标示 building fs  结束

    return 0;

jffs_build_fs_fail:
    jffs_cleanup_control(c);
    return err;
} /* jffs_build_fs()    */
```

5、扫描flash

jffs_scan_flash 是一个很长的函数，下面我们只是描述函数的结构

```
static int jffs_scan_flash(struct jffs_control *c)
{
    pos = 0    //pos  表示当前 flash 上扫描的位置

    通过 check_partly_erased_sectors 函数检查 flipping bits

    while (读到 flash 最后一个 byte) {

        //从当前位置读从一个 u32
        switch (flash_read_u32(fmc->mtd, pos)) {
            case JFFS_EMPTY_BITMASK:
```

如果读到的字节是 JFFS_EMPTY_BITMASK 也就是 0xffffffff，那么该位置上 flash 是 free 的，我们还没有使用它，接着就会用一个 4k 的 buffer 去读直到不是 JFFS_EMPTY_BITMASK 的位置停止。

```
            case JFFS_DIRTY_BITMASK:
```

如果读到的字节是 JFFS_DIRTY_BITMASK 也就是 0x00000000，那么读出所有的连续的 0x00000000,分配一个 jffs_fm 结构表示该区域，但是 jffs_fm->nodes 为空，也就是标示该区域为 dirty，并把该 jffs_fm 连接到 jffs_fmcontrol 的双向链表中。一般这种区域是由于到了 flash 的末尾，剩余的空间不够写一个 jffs_raw_inode 结构，所以全部写 0

```
            case JFFS_MAGIC_BITMASK:
```

找到一个真正的 jffs_raw_inode 结构，将该 raw indoe 读出来，如果是一个 bad raw inode(例如校验错误等等)，那么分配一个 jffs_fm 结构表示该区域，但是 jffs_fm->nodes 为空，也就是标示该区域为 dirty；如果是一个 good inode，那么建立 jffs_node 结构和 jffs_fm 结构，并把该 jffs_fm 连接到 jffs_fmcontrol 的双向链表中，然后把 jffs_node 插入到 jffs_file 的 version list 中，表明该 node 的文件的 jffs_file 结构先通过哈西表查找，如果没有则创建，一般来说，如果这个 jffs_node 是扫描到

的该文件的第一个节点，那么就需要创建 jffs_file 结构，此后就可以通过哈西表找到该 jffs_file 结构。

```
            }

        }
```

解释：

（1）通过上面的循环，可以建立所有的文件的 jffs_file 结构，并且 version list 已经建好，但是 range list 还没有建立，文件还不能正常读写

（2）通过上面的循环，可以建立表示 flash 使用情况的 jffs_fmcontrol 结构，并且所有的 used_size 都已经通过 jffs_fm 联接成链表。

}

五、文件打开

本身文件的打开对 jffs 文件系统下的文件是没有什么实际的意义，因为在 mount 的时候就会 scan 整个 flash 而建立文件树，所有的文件都其实是打开的了。需要留意的是：

1、创建一个文件

可以通过 open 函数创建一个文件，只要设定相映的 flag。本操作是通过 jffs_create 完成，很显然，该函数最直观的效果是向 flash 写入一个 jffs_raw_inode 及其文件名，当然也要维护文件树的完整性。

```
static int jffs_create(struct inode *dir, struct dentry *dentry, int mode)
{

//获得 create 文件的那个目录的 jffs_file 结构，我们前面说过 inode 结构的 inode->u.generic_ip 指向她的 jffs_file 结构。
    dir_f = (struct jffs_file *)dir->u.generic_ip;

    c = dir_f->c;

//分配一 jffs_node 的结构，该结构是该 jffs_file 的第一个 node
    if (!(node = jffs_alloc_node())) {
        D(printk("jffs_create(): Allocation failed: node == 0\n"));
        return -ENOMEM;
    }

    down(&c->fmc->biglock);

    node->data_offset = 0;
    node->removed_size = 0;
```

```
//初始化向 flash 上写的 jffs_raw_inode 结构
    raw_inode.magic = JFFS_MAGIC_BITMASK;
    raw_inode.version = 1;          //第一个 version 是一
    。。。略过部分代码
    raw_inode.deleted = 0;

//将 raw inode  和文件名写进 flash
    if ((err = jffs_write_node(c, node, &raw_inode,
                        dentry->d_name.name, 0, 0, NULL)) < 0) {
        D(printk("jffs_create(): jffs_write_node() failed.\n"));
        jffs_free_node(node);
        goto jffs_create_end;
    }

//在 jffs_insert_node 中，建立 jffs_file 和这个新产生的 node 的关系
    if ((err = jffs_insert_node(c, 0, &raw_inode, dentry->d_name.name,
                         node)) < 0) {
        goto jffs_create_end;
    }

    /* Initialize an inode.    */
    inode = jffs_new_inode(dir, &raw_inode, &err);
    if (inode == NULL) {
        goto jffs_create_end;
    }
    err = 0;
//设定各种操作函数集合
    inode->i_op = &jffs_file_inode_operations;
    inode->i_fop = &jffs_file_operations;
    inode->i_mapping->a_ops = &jffs_address_operations;
    inode->i_mapping->nrpages = 0;

    d_instantiate(dentry, inode);

} /* jffs_create()    */
```

2、jffs_lookup

在打开文件的过程中，需要在目录中搜索，这里调用 jffs_lookup

```
static struct dentry *
jffs_lookup(struct inode *dir, struct dentry *dentry)
{

    struct jffs_control *c = (struct jffs_control *)dir->i_sb->u.generic_sbp;
```

```
    struct inode *inode = NULL;
    len = dentry->d_name.len;
    name = dentry->d_name.name;

    down(&c->fmc->biglock);

    r = -ENAMETOOLONG;
    if (len > JFFS_MAX_NAME_LEN) {    //名字是否超过 jffs 要求的最大值
        goto jffs_lookup_end;
    }

    r = -EACCES;
//获得目录的 jffs_file 结构
    if (!(d = (struct jffs_file *)dir->u.generic_ip)) {
        D(printk("jffs_lookup(): No such inode! (%lu)\n",
                 dir->i_ino));
        goto jffs_lookup_end;
    }

//下面的用注视代替了原码
    if ((len == 1) && (name[0] == '.')) {
        处理当前目录.的情况，因为目录的 jffs_file 已经找到，所以直接调用 iget
找到它的 inode 结构
    } else if ((len == 2) && (name[0] == '.') && (name[1] == '.')) {
        处理..的情况，上层目录的文件号可以通过 jffs_file 的 pino 找到，所以调用
iget 找到它上层目录的 inode 结构
    } else if ((f = jffs_find_child(d, name, len))) {
        正常情况，通过 jffs_find_child 找到该文件的 jffs_file 结构，也就找到了文
件号，于是可以通过 iget 函数根据文件号找到该文件的 inode 结构
    } else {
        找不到文件
    }

//维护 vfs 结构的一致性
    d_add(dentry, inode);

    up(&c->fmc->biglock);
    return NULL;

} /* jffs_lookup()    */
```

3、truncate

对于 truncate，是通过 jffs_setattr 实现，此处掠过

4、generic_file_open

一般的文件打开是调用 generic_file_open。

六、文件读写

对 jffs 的文件的读写是使用 page cache 的，jffs 层上具体的读函数使用了通用的 generic_file_open，address_space 结构描述了 page cache 中的页面，对于页面的操作，jffs 是这样定义的

```
static struct address_space_operations jffs_address_operations = {
    readpage: jffs_readpage,
    prepare_write: jffs_prepare_write,
    commit_write: jffs_commit_write,
};

static int jffs_readpage(struct file *file, struct page *page)
{
    这个函数很简单，通过 jffs_read_data 读出一个页面大小的内容
}

int jffs_read_data(struct jffs_file *f, unsigned char *buf, __u32 read_offset,
                __u32 size)
{
//写的偏移不能大于文件的大小
    if (read_offset >= f->size) {
        D(printk("    f->size: %d\n", f->size));
        return 0;
    }

//首先要沿着 range list 找到该 offset 所在的 node
    node = f->range_head;
    while (pos <= read_offset) {
        node_offset = read_offset - pos;
        if (node_offset >= node->data_size) {
            pos += node->data_size;
            node = node->range_next;
        }
        else {
            break;
        }
    }

//下面的循环读入缓冲区
    while (node && (read_data < size)) {
```

```
        int r;
        if (!node->fm) {
            /* This node does not refer to real data.    */
            r = min(size - read_data,
                        node->data_size - node_offset);
            memset(&buf[read_data], 0, r);
        }
```

从 offset 所在的 node 开始，读出一个页面的数据，根据 node 的 data_size 的大小，可能一个 node 就高定，也许要读一系列的 node

```
        else if ((r = jffs_get_node_data(f, node, &buf[read_data],
                                node_offset,
                                size - read_data,
                                f->c->sb->s_dev)) < 0) {
            return r;
        }
        read_data += r;
        node_offset = 0;
        node = node->range_next;
    }

    return read_data;
}
```

对于 jffs_prepare_write，只是保证该页面是正确的，如果需要更新，那末就重新读入该页

```
static ssize_t jffs_prepare_write(struct file *filp, struct page *page,
                          unsigned from, unsigned to)
{
    if (!Page_Uptodate(page) && (from || to < PAGE_CACHE_SIZE))
        return jffs_do_readpage_nolock(filp, page);

    return 0;
} /* jffs_prepare_write() */

static ssize_t jffs_commit_write(struct file *filp, struct page *page,
                         unsigned from, unsigned to)
{
        void *addr = page_address(page) + from;

        loff_t pos = (page->index<<PAGE_CACHE_SHIFT) + from;

        return jffs_file_write(filp, addr, to-from, &pos);
} /* jffs_commit_write() */

static ssize_t jffs_file_write(struct file *filp, const char *buf, size_t count,
        loff_t *ppos)
```

```
{

    err = -EINVAL;

//检查是否是正规文件
    if (!S_ISREG(inode->i_mode)) {
        D(printk("jffs_file_write(): inode->i_mode == 0x%08x\n",
                 inode->i_mode));
        goto out_isem;
    }
//只要是正常打开的文件 inode->u.generic_ip 应该指向她的 jffs_file 结构
    if (!(f = (struct jffs_file *)inode->u.generic_ip)) {
        D(printk("jffs_file_write(): inode->u.generic_ip = 0x%p\n",
                 inode->u.generic_ip));
        goto out_isem;
    }

    c = f->c;

//因为对文件 node 的大小有限制，所以如果写的数目非常大，那么我们会产生若干个 node 来完成一次的写操作，this_count 就是本次写操作的数据量
    thiscount = min(c->fmc->max_chunk_size - sizeof(struct jffs_raw_inode), count);
//通过一个 while 循环，完成所有的写操作
    while (count) {
//分配一个 jffs_node 结构
        if (!(node = jffs_alloc_node())) {
            err = -ENOMEM;
            goto out;
        }
//设定该 node 的在整个文件的逻辑偏移
        node->data_offset = pos;
        node->removed_size = 0;

//初始化 raw_node
        raw_inode.magic = JFFS_MAGIC_BITMASK;
        。。。。略过部分代码
        raw_inode.deleted = 0;

//我们在某一个文件偏移位置写入数据，除非是在文件末尾，要不然的话，我们需要覆盖调部分内容，remove_size 指明了该 node 要覆盖的数据的大小，也就是说从该文件偏移处起，前面节点的 remove_size 大小的空间要被 remove，数据将被新的 node 代替。
        if (pos < f->size) {
            node->removed_size = raw_inode.rsize = min(thiscount, (__u32)(f->size - pos));
```

```
            }

//通过 jffs_write_node 将 jffs_raw_inode   文件名   数据写入 flash
            if ((err = jffs_write_node(c, node, &raw_inode, f->name,
                                       (const unsigned char *)buf,
                                       recoverable, f)) < 0) {
                jffs_free_node(node);
                goto out;
            }
//调整位置
            written += err;
            buf += err;
            count -= err;
            pos += err;

//将新生成的 node 插入到 jffs_file 结构的 node list 中
            if ((err = jffs_insert_node(c, f, &raw_inode, 0, node)) < 0) {
                goto out;
            }

            thiscount = min(c->fmc->max_chunk_size - sizeof(struct jffs_raw_inode),
count);
        }
    out:
        up(&c->fmc->biglock);

//更新 vfs 结构上的信息
        if (pos > inode->i_size) {
            inode->i_size = pos;
            inode->i_blocks = (inode->i_size + 511) >> 9;
        }
        inode->i_ctime = inode->i_mtime = CURRENT_TIME;
//将该文件的 inode 挂入 sb 的 dirty list
        mark_inode_dirty(inode);
        invalidate_inode_pages(inode);

    out_isem:
        return err;
} /* jffs_file_write()    */
```

七、垃圾回收

1、垃圾回收的内核线程

```
int jffs_garbage_collect_thread(void *ptr)
{
      //主循环
     for (;;) {
// 看看是否需要睡眠，一般有两种情况，一种是自由空间的数量 < MIN_FREE_BYTES 同时至少有一个 sector 的 flash 空间是 dirty 的，还有一种情况是 dirty 空间的数量 > MAX_DIRTY_BYTES
          if (!thread_should_wake(c))
               set_current_state (TASK_INTERRUPTIBLE);
//我们垃圾回收的内核线程优先级很低，调用 schedule 看一看是否有其他进程的可以调度
          schedule();
//信号处理部分
          while (signal_pending(current)) {
               switch(signr) {
               case SIGSTOP:
                    set_current_state(TASK_STOPPED);
                    schedule();
                    break;
               case SIGKILL:
                    c->gc_task = NULL;
                    complete_and_exit(&c->gc_thread_comp, 0);
               case SIGHUP:
                    set_current_state(TASK_INTERRUPTIBLE);
                    schedule_timeout(2*HZ);
                    break;
               }
          }
          down(&fmc->biglock);
          c->gc_background = 1;
//如果从垃圾回收点处开始，有全部是 dirty 的 sector，那么将通过 jffs_try_to_erase 将该扇区擦除，变为 free
          if ((erased = jffs_try_to_erase(c)) < 0) {
               printk(KERN_WARNING "JFFS: Error in "
                         "garbage collector: %ld.\n", erased);
          }
//只要至少擦掉一个 sector，我们就结束 gc thread
          if (erased)
               goto gc_end;
//如果自由空间等于 0，没办法拉，只好自杀
```

```
            if (fmc->free_size == 0) {
                send_sig(SIGQUIT, c->gc_task, 1);
                goto gc_end;
            }
//如果执行到此处，则说明具备垃圾回收的条件，但是从垃圾回收点处开始的那一个 sector 不是完全 dirty 的，需要搬移部分的 raw inode
            if ((result = jffs_garbage_collect_next(c)) < 0) {
                printk(KERN_ERR "JFFS: Something "
                                "has gone seriously wrong "
                                "with a garbage collect: %d\n", result);
            }
//至此至少回收了一个 sector  本次任务结束，继续睡眠
        gc_end:
            c->gc_background = 0;
            up(&fmc->biglock);
        } /* for (;;) */
} /* jffs_garbage_collect_thread() */
```