

声明

你可以自由地随意修改本文档的任何文字内容及图表，但是如果你在自己的文档中以任何形式直接引用了本文档的任何原有文字或图表并希望发布你的文档，那么你也得保证让所有得到你的文档的人同时享有你曾经享有过的权利。

JFFS2 源代码情景分析（Beta2）

作者在www.linuxforum.net上的ID为shrek2
欢迎补充，欢迎批评指正！

前言（new）	4
第 1 章 jffs2 的数据实体及其内核描述符（improved）	5
数据实体的内核描述符jffs2_raw_node_ref	6
文件的内核描述符jffs2_inode_cache	6
jffs2_raw_dirent数据实体及其上层数据结构	7
jffs2_raw_inode数据实体及其上层数据结构	10
第 2 章 描述jffs2 特性的数据结构（improved）	14
文件系统超级块的u域：jffs2_sb_info数据结构	14
文件索引结点的u域：jffs2_inode_info数据结构	18
打开正规文件后相关数据结构之间的引用关系	19
第 3 章 注册文件系统	21
init_jffs2_fs函数	21
register_filesystem函数	23
第 4 章 挂载文件系统（improved）	25
jffs2_read_super函数	25
jffs2_do_fill_super函数	27
jffs2_do_mount_fs函数	30
jffs2_build_filesystem函数	31
jffs2_scan_medium函数	34
jffs2_scan_eraseblock函数	40
jffs2_scan_inode_node函数	52

jffs2_scan_make_ino_cache函数	55
jffs2_scan_dirent_node函数	56
full_name_hash函数	59
jffs2_add_fd_to_list函数	60
jffs2_build_inode_pass1 函数	61
第 5 章 打开文件时建立inode的方法	63
iget和iget4 函数	63
get_new_inode函数	65
jffs2_read_inode函数	68
jffs2_do_read_inode函数 (improved)	73
jffs2_get_inode_nodes函数	78
第 6 章 jffs2 中写正规文件的方法	88
sys_write函数	89
generic_file_write函数	90
jffs2_prepare_write函数	98
jffs2_commit_write函数	102
jffs2_write_inode_range函数	104
jffs2_write_dnode函数	107
第 7 章 jffs2 中读正规文件的方法	111
jffs2_readpage函数	111
jffs2_do_readpage_nolock函数	111
jffs2_read_inode_range函数	112
jffs2_read_dnode函数	115
第 8 章 jffs2 中符号链接文件的方法表 (new)	120
jffs2_follow_link函数	120
jffs2_getlink函数	121
第 9 章 jffs2 中目录文件的方法表 (new)	122
jffs2_create函数	122
jffs2_new_inode函数	124
jffs2_do_create函数	126
jffs2_do_new_inode函数	129
第 10 章 jffs2 的Garbage Collection	131
jffs2_start_garbage_collect_thread函数	131
jffs2_garbage_collect_thread函数	132
jffs2_garbage_collect_pass函数	135
jffs2_erase_pending_trigger函数	141
第 11 章 讨论和体会	142

什么是日志文件系统，为什么要使用jffs2	142
为什么需要红黑树	142
何时、如何判断数据实体是过时的	143
后记.....	144
附录 用jffs2map2 模块导出文件的数据实体（new）	145
观察根目录文件的数据实体	145
观察符号链接的信息	147
观察正规文件创建后的数据实体	147
观察jffs2_raw_inode数据实体的大小上限	148

前言 (new)

第 1 稿后拜读了情景分析中文件系统的相关章节，将 `ext2` 与 `jffs2` 相类比，显著地加深了对上层文件系统相关概念的理解，尤其是 `VFS` 框架的数据结构的设计思想，比如为了实现良好的可移植性和重用性，上层 `VFS` 框架代码就必须与具体的应用（底层具体文件系统）无关，而这一点恰恰是通过设计中间层的函数指针接口实现的。依靠接口实现的封装性是可移植性的基础。还有 `VFS` 各数据结构的设计目的和设计方法等等，比如只有尽可能地概括各种不同文件系统的共性才能使 `VFS` 具有良好的通用性，同时通过各种数据结构中的 `union`，让具体文件系统的实现来定义、解释、使用其特有数据结构、描述在具体设备上具体文件系统的数据组织格式。“`union` 域反映了各种不同文件系统在高层数据结构上的差异”。

第 2 稿对第 1 稿的改进也主要集中在对 `jffs2` 相关数据结构的理解和 `ext2` 与 `jffs2` 的类比上，这样可以加深对 `jffs2` 数据结构的理解。对于已经看过第 1 稿的朋友，再翻翻第 1、2 章就差不多了。另外有兴趣的话也可以再看看其它新增的章节。

第 1 稿中只涉及了正规文件的访问方法，第 2 稿中补充了符号链接文件和目录文件的相关方法。这些补充可以验证、加深对各种类型文件在 `jffs2` 中的实现方法的理解，比如可以通过目录文件的 `create` 方法看到在创建正规文件时是如何设置 `inode` 的 `u` 域、向父目录文件增加 `jffs2_raw_dirent` 目录项的（具体操作不止这些）。由于我目前的兴趣主要集中在内核中那些与具体应用无关的高层框架上面，而不是与具体应用相关的最底层代码，所以第 1 稿中有关 `jffs2` 某些实现细节的遗留问题暂时还没有继续研究下去，请大家谅解。

感谢论坛上所有鼓励支持我、尤其是那些向我提出问题的朋友，你们的提问促使我研究得更深入一些。完成第 1 稿后我就曾打算通过继续阅读 `mkfs.jffs2` 的源代码、或者编写一个能导出指定的 `jffs2` 文件系统上所有数据结点的模块来加深对 `jffs2` 的理解。在和网友们讨论 `jffs2` 对 `jffs2_raw_inode` 数据结点的最大长度限制时我完成了这个模块，目前它可以导出指定 `jffs2` 文件系统上指定文件的所有数据结点的信息，这样每次从根目录开始就可以逐层得到文件系统目录树中任何一个文件的数据结点信息了。

人们对 `Linux` 的喜爱很大程度上源自于它的可实践性，从而极大地调动研究和使用的积极性。通过这个可以导出一个文件在 `flash` 上的物理实体的模块，`jffs2` 的概念前所未有地清晰、真实，也进一步改正、完善了对目录文件和 `jffs2_raw_dirent` 的理解，有兴趣的朋友可以参见附录及附件源代码，或者进一步改进这个模块。（在第 1 稿中曾错误地认为目录文件是没有 `jffs2_raw_inode` 数据实体的，很抱歉，而实际情况是除了根目录外所有目录都由惟一的、用于描述其类型和其它管理信息的 `jffs2_raw_inode`，与此相关的 `jffs2_full_dnode` 则由 `jffs2_inode_info` 的 `metadata` 域直接指向。）

Let's DIY Linux!

2006 年 1 月 18 日星期三

第 1 章 jffs2 的数据实体及其内核描述符 (improved)

存储于辅存的任何文件都至少包含三种类型的信息：文件的数据本身、描述文件属性的管理信息、以及描述文件在文件系统内部的位置信息。文件的位置信息用于实现“从路径名找到文件”的机制。jffs2 在 flash 上只有两种类型的数据实体：jffs2_raw_inode 和 jffs2_raw_dirent，前者包含文件的管理信息，类似于 ext2 中的磁盘索引结点 ext2_inode，后者用于描述文件在文件系统的位置，类似于 ext2 中的 ext2_dir_entry_2 目录项实体。

与 ext2_inode 可以定位磁盘文件的磁盘块相比，jffs2_raw_inode 没有这种“索引”功能，flash 上文件的数据是由若干离散的 jffs2_raw_inode 数据结点进行描述的。与 ext2_dir_entry_2 类似，jffs2_raw_dirent 也描述了文件名及其索引结点编号之间的映射关系，是文件硬链接的物理实体。

在 ext2 中目录文件所占的磁盘块由其 ext2_inode 进行索引，在一个磁盘块内部为描述其下子目录、子文件的 ext2_dir_entry_2 实体。由于 ext2_dir_entry_2 可指明自身的长度，而且它们在磁盘块内部是连续存放的，所以并不需要描述其所在目录文件的索引结点号。而 jffs2 中目录文件由一个 jffs2_raw_inode 数据实体和若干 jffs2_raw_dirent 数据实体组成，由于目录文件的数据实体之间都是离散存放的，所以每个 jffs2_raw_dirent 中还得描述其所属目录文件的索引结点号，参见下文。

正规文件、符号链接文件、SOCKET/FIFO 文件、设备文件都由一个或多个 jffs2_raw_inode 来表示，而紧随 jffs2_raw_inode 数据结构后的为相关数据块，不同文件所需要的 jffs2_raw_inode 个数及其后数据的内容如下表所示：

文件类型	所需 jffs2_raw_inode 结点的个数	后继数据的内容
目录文件	1（根目录除外）	无
正规文件	>= 1	文件的数据
符号链接文件	1	被链接的文件名
SOCKET/FIFO 文件	1	无
设备文件	1	设备号

另外，所有文件都至少存在一个 jffs2_raw_dirent 数据实体（具体个数由其硬链接个数决定），它们组成其父目录文件的内容。区分 jffs2_raw_dirent 和 jffs2_raw_inode 是为了实现硬链接：在 jffs 版本 1 中就只有类似 jffs2_raw_inode 的一种数据实体，只能实现符号链接。（注：根目录没有父目录了，自然不需要 jffs2_raw_dirent。另外它也没有那个惟一的 jffs2_raw_inode。）

jffs2_raw_dirent 和 jffs2_raw_inode 数据实体都以相同的“头”开始：

```
struct jffs2_unknown_node
{
    /* All start like this */
    jint16_t magic;
    jint16_t nodetype;
```

```

    jint32_t totlen; /* So we can skip over nodes we don't grok */
    jint32_t hdr_crc;
} __attribute__((packed));

```

其中 `nodetype` 指明数据结点的具体类型 `JFFS_NODETYPE_DIRENT` 或者 `JFFS2_NODETYPE_INODE`；`totlen` 为包括后继数据的整个数据实体的总长度；`hdr_crc` 为头部中其它域的 CRC 校验值。另外整个数据结构在内存中以“紧凑”方式进行存储，这样当从 flash 上复制数据实体的头部到该数据结构后，其各个域就能够“各得其所”了。

数据实体的内核描述符 `jffs2_raw_node_ref`

flash 上每个数据实体的位置、长度都由 `jffs2_raw_node_ref` 数据结构描述：

```

struct jffs2_raw_node_ref
{
    struct jffs2_raw_node_ref *next_in_ino;
    struct jffs2_raw_node_ref *next_phys;
    uint32_t flash_offset;
    uint32_t totlen;
};

```

其中 `flash_offset` 表示相应数据实体在 flash 分区上的物理地址，`totlen` 为包括后继数据的总长度。同一个文件的多个 `jffs2_raw_node_ref` 由 `next_in_ino` 组成一个循环链表，链表首为文件内核描述符 `jffs2_inode_cache` 数据结构的 `nodes` 域，链表末尾元素的 `next_in_ino` 则指向 `jffs2_inode_cache`，这样任何一个 `jffs2_raw_node_ref` 元素就都知道自己所在的文件了。

一个 flash 擦除块内所有数据实体的内核描述符由 `next_phys` 域组织成一个链表，其首尾元素分别由擦除块描述符 `jffs2_eraseblock` 数据结构的 `first_node` 和 `last_node` 域指向。

文件的内核描述符 `jffs2_inode_cache`

每一个文件在内核中都由惟一的 `jffs2_inode_cache` 数据结构表示，其最主要的作用就是提供了“文件及其数据之间的映射机制”：

```

struct jffs2_inode_cache {
    struct jffs2_full_dirent *scan_dents;
    struct jffs2_inode_cache *next;
    struct jffs2_raw_node_ref *nodes;
    uint32_t ino;
    int nlink;
    int state;
};

```

ino 为文件的在文件系统中唯一的索引结点号；所有文件内核描述符被组织在一个 `inocache_list` 哈希表中，`next` 用于组织冲突项的链表。

`nlink` 为文件的硬链接个数，在挂载文件系统时会计算指向每个文件的目录项个数，然后赋值给 `nlink`。注意上层 VFS 所使用的 `nlink` 与此不同：在打开文件时会首先将 `jffs2_inode_cache` 的 `nlink` 复制到 `inode` 的 `nlink`，然后对于非叶目录，会根据其下子目录的目录项个数增加 `inode` 的 `nlink`。详见后文。

上层 VFS 的 `inode` 的主要作用之一就是描述文件及其数据之间的映射关系。由于不同文件系统中数据的组织格式不同，所以这种映射关系的描述符自然放到 `inode` 的 `u` 域中，由具体文件系统的方法实现。就 `ext2` 而言，`ext2_inode_info` 的 `i_data[]` 索引文件的磁盘块，而其“物质基础”为文件磁盘索引结点 `ext2_inode` 的 `i_block[]` 数组；就 `jffs2` 而言，根据文件类型的不同由 `jffs2_inode_info` 的 `fragtree/dents/metadata` 描述 flash 数据实体相关上层数据结构的组织，其“物质基础”就是 `jffs2_inode_cache` 的 `nodes` 链表，它组织了文件所有数据实体的内核描述符 `jffs2_raw_node_ref` 数据结构，在挂载文件系统时建立。

在挂载 `jffs2` 文件系统时将遍历整个文件系统（扫描 `jffs2` 文件系统映象所在的整个 flash 分区），为 flash 上每个 `jffs2_raw_dirent` 和 `jffs2_raw_inode` 数据实体创建相应的内核描述符 `jffs2_raw_node_ref`、为每个文件创建内核描述符 `jffs2_inode_cache`，具体过程详见下文。另外在打开文件时，如果是目录文件，则还要为每个 `jffs2_raw_dirent` 创建相应的 `jffs2_full_dirent` 数据结构并组织为链表；如果是正规文件等，则为每个 `jffs2_raw_inode` 创建相应的 `jffs2_full_dnode`、`jffs2_tmp_dnode_info`、`jffs2_node_frag` 数据结构，并组织到红黑树中，详见下文。

jffs2_raw_dirent 数据实体及其上层数据结构

`jffs2_raw_dirent` 数据实体为 `jffs2` 中目录项的表示形式，即硬链接的物理实体。文件的目录项组成了其父目录文件的“数据”。定义如下：

```
struct jffs2_raw_dirent
{
    jint16_t magic;
    jint16_t nodetype;      /* == JFFS_NODETYPE_DIRENT */
    jint32_t totlen;
    jint32_t hdr_crc;
    jint32_t pino;
    jint32_t version;
    jint32_t ino;           /* == zero for unlink */
    jint32_t mctime;
    uint8_t nsize;
    uint8_t type;
    uint8_t unused[2];
    jint32_t node_crc;
    jint32_t name_crc;
    uint8_t name[0];
} __attribute__((packed));
```

紧随jffs2_raw_dirent的是相应文件的文件名，长度由nsize域表示，而name域预留了文件名字符串结束符的空间，ino表示相应文件的索引结点号。如前所述flash上目录文件的若干jffs2_raw_dirent数据实体是离散的，而且也没有类似ext2_inode的“索引”机制，所以就必须由每个jffs2_raw_dirent数据实体表明自己所属的目录文件。pino即是为这个目的设计的，表示该目录项所属的目录文件的索引节点号。另外在挂载文件系统时，会将jffs2_raw_dirent数据实体的描述符加入pino所指文件，即该目录项所属目录文件的jffs2_inode_cache的nodes链表，参见[jffs2_scan_dirent_node函数](#)。

版本号 version 是相对于某一文件内部的概念。任何文件都由若干 jffs2_raw_dirent 或者 jffs2_raw_inode 数据实体组成，修改文件的“某一个区域”时将向 flash 写入新的数据实体，它的 version 总是不断递增的。一个文件的所有数据实体的最高 version 号由其 inode 的 u 域，即 jffs2_inode_info 数据结构中的 highest_version 记录。文件内同一“区域”可能由若干数据实体表示，它们的 version 互不相同，而且除了最新的一个数据结点外，其余的都被标记为“过时”。（另外，按照 jffs2 作者的论文，如果 flash 上数据实体含有相同的数据则允许它们的 version 号相同）

打开目录文件时要创建其VFS的dentry、inode、file对象，在创建inode时要调用super_operations函数指针接口中的read_inode方法，根据相应的内核描述符jffs2_raw_node_ref为每个目录项创建一个上层的jffs2_full_dirent数据结构，并读出jffs2_raw_dirent数据实体后的文件名到jffs2_full_dirent数据结构后面。jffs2_full_dirent组成的链表则由目录文件的索引结点inode.u.dents（即jffs2_inode_info.dents）指向，参见[图 1](#)。

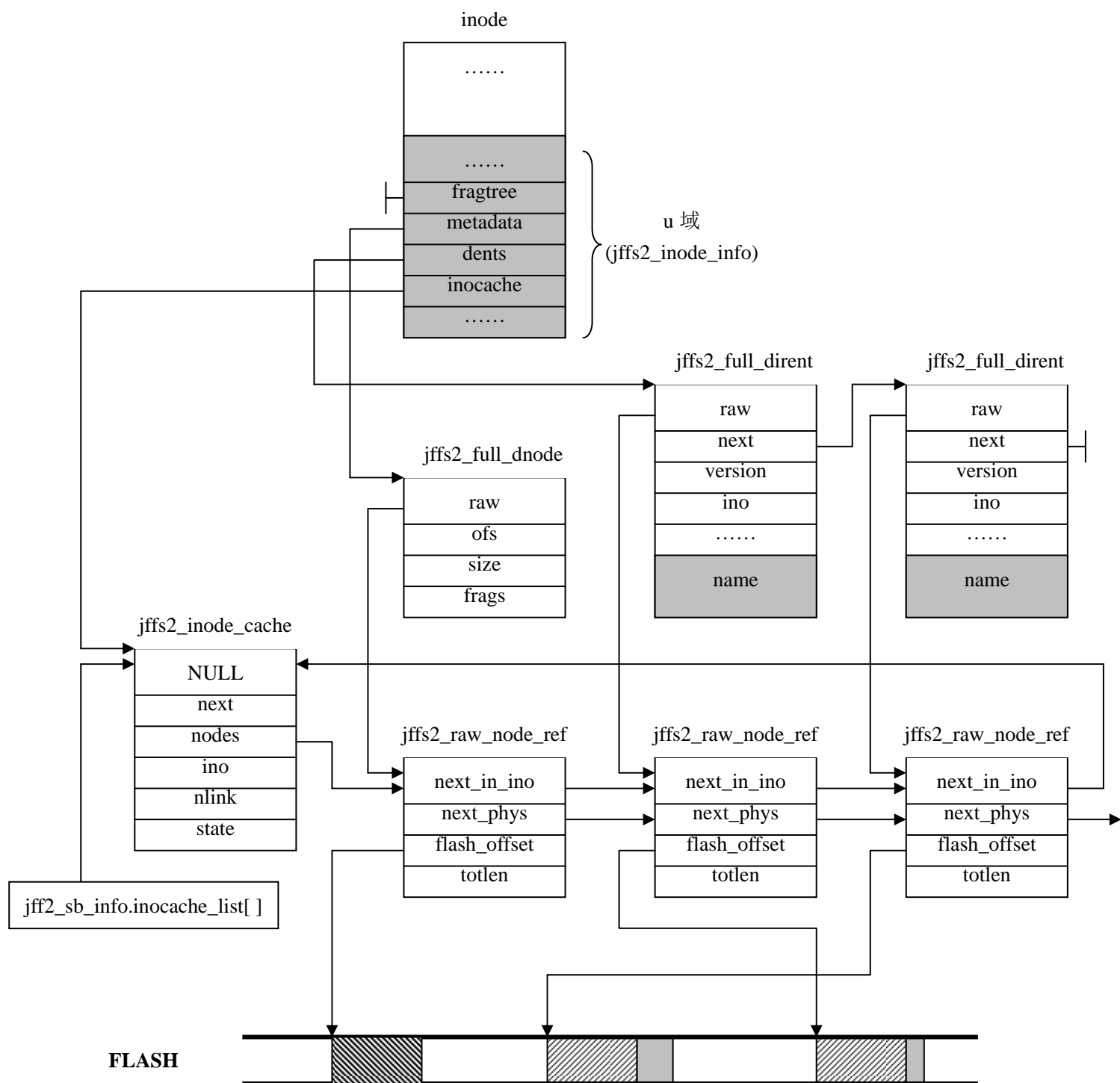
jffs2_full_dirent 数据结构在打开目录文件时才创建，用于保存读出的 jffs2_raw_dirent 数据实体的结果，其定义如下：

```
struct jffs2_full_dirent
{
    struct jffs2_raw_node_ref *raw;
    struct jffs2_full_dirent *next;
    uint32_t version;
    uint32_t ino;           /* == zero for unlink */
    unsigned int nhash;
    unsigned char type;
    unsigned char name[0];
};
```

其中 raw 指向相应的 jffs2_raw_node_ref 结构，紧随其后的为从 flash 上读出的文件名。

总之，在打开一个目录文件后其相关数据结构的关系如下图所示（假设该目录有 2 个目录项，没有画出 file、dentry）。其中对 inode 的 u 域即 jffs2_inode_info 数据结构的解释参见下文。

（注：很抱歉，在第 1 稿中我曾**错误地**认为目录文件只包含jffs2_raw_dirent数据实体，而实际上目录文件、设备文件、符号链接文件都有惟一的jffs2_raw_inode数据实体，在打开文件时为其创建的jffs2_full_dnode由其inode的u域jffs2_inode_info的metadata指向。一个目录文件的数据实体信息可以使用jffs2map2 模块导出，参见[附录](#)）



说明:



jffs2_raw_dirent 实体及紧随其后的文件名



jffs2_raw_inode 实体 (后继无数据)

图 1, 目录文件的相关数据结构

jffs2_raw_inode数据实体及其上层数据结构

jffs2_raw_inode 数据实体用于描述文件的类型、访问权限、其它管理信息和文件的数据（如果存在的话）。由于正规文件、符号链接、设备文件的 jffs2_raw_inode 后都有相应的数据，共同组成一个 flash 上的数据实体，所以在下文中若无特别说明“jffs2_raw_inode”均指该数据结构本身及其后的数据。

```
struct jffs2_raw_inode
{
    jint16_t magic;          /* A constant magic number. */
    jint16_t nodetype;       /* == JFFS2_NODETYPE_INODE */
    jint32_t totlen;         /* Total length of this node (inc data, etc.) */
    jint32_t hdr_crc;
    jint32_t ino;            /* Inode number. */
    jint32_t version;        /* Version number. */
    jint32_t mode;           /* The file's type or mode. */
    jint16_t uid;            /* The file's owner. */
    jint16_t gid;            /* The file's group. */
    jint32_t isize;          /* Total resultant size of this inode (used for truncations) */
    jint32_t atime;          /* Last access time. */
    jint32_t mtime;          /* Last modification time. */
    jint32_t ctime;          /* Change time. */
    jint32_t offset;         /* Where to begin to write. */
    jint32_t csize;          /* (Compressed) data size */
    jint32_t dsize;          /* Size of the node's data. (after decompression) */
    uint8_t compr;           /* Compression algorithm used */
    uint8_t usercompr;       /* Compression algorithm requested by the user */
    jint16_t flags;          /* See JFFS2_INO_FLAG_ */
    jint32_t data_crc;       /* CRC for the (compressed) data. */
    jint32_t node_crc;       /* CRC for the raw inode (excluding data) */
    // uint8_t data[dsize];
} __attribute__((packed));
```

一个正规文件可能由若干 jffs2_raw_inode 数据实体组成，每个数据实体含有文件一个区域的数据。即使文件的同一个区域也可能因为修改而在 flash 上存在多个数据实体，它们都含有相同的 ino，即文件的索引结点编号。

文件在逻辑上被当作一个连续的线性空间，每个 jffs2_raw_inode 所含数据在该线性空间内的偏移由 offset 域表示。注意 offset 为文件内的偏移，而该 jffs2_raw_inode 在 flash 分区中的偏移则由其内核描述符 jffs2_raw_node_ref 的 flash_offset 域表示。

jffs2 支持数据压缩。如果后继数据没有被压缩，则 compr 被设置 JFFS2_COMPR_NONE。压缩前（或解压缩后）的数据长度由 dsize 表示，而压缩后的数据长度由 csize 表示。从后文的相关函数分析中可以看到，在修改文件的已有内容或者写入新内容时，首先要将数据压缩，然后在内存中组装合适的 jffs2_raw_inode

结构，最后再将二者连续地写入 flash。而在读 flash 上的设备结点时首先读出 jffs2_raw_inode 结构，然后根据其中的 csize 域的值，分配合适大小的缓冲区，第二次再读出紧随其后的（压缩了的）数据。在解压缩时则根据 dsize 大小分配合适的缓冲区。另外，如果 jffs2_raw_node 没有后继数据而是代表一个洞，那么 compr 被设置为 JFFS2_COMPR_ZERO。

除了文件头 jffs2_unknown_node 中有 crc 校验值外，在 jffs2_raw_inode 中还有该数据结构本身及其后数据的 crc 校验值。这些校验值在创建 jffs2_raw_inode 时计算，在读出该数据实体时进行验证。

在 ext2 中，磁盘正规文件所占用的所有磁盘块都通过其 ext2_inode 的 i_block[] 进行直接或间接的索引，而 jffs2 中一个正规文件的所有数据实体可能分布在 flash 的任何位置上，每个 jffs2_raw_inode 都“独善其身”地描述自己的后继数据。正因为缺少类似 ext2_inode 的“索引”机制，所以在挂载 jffs2 时才不得不遍历整个 flash 分区，从而找到每个文件的所有数据实体，并建立它们的内核描述符 jffs2_raw_node_ref 并组织到文件的 jffs2_inode_cache 的 nodes 链表中去。

在打开正规文件时要为其 jffs2_raw_inode 数据实体创建相应的内核映像 jffs2_full_dnode（以及临时数据结构 jffs2_tmp_dnode_info）、jffs2_node_frag，并通过后者组织到红黑树中。

```
struct jffs2_full_dnode
{
    struct jffs2_raw_node_ref *raw;
    uint32_t ofs;          /* Don't really need this, but optimisation */
    uint32_t size;
    uint32_t frags;        /* Number of fragments which currently refer
                           to this node. When this reaches zero, the node is obsolete. */
};
```

该数据结构的 ofs 和 size 域用于描述数据实体的后继数据在文件内的逻辑偏移及长度，它们的值来自数据实体 jffs2_raw_inode 的 offset 和 dsize 域。而 raw 指向数据实体的内核描述符 jffs2_raw_node_ref 数据结构。

尤其需要说明的是 frags 域。当打开一个文件时为每个 jffs2_raw_node_ref 创建 jffs2_full_dnode 和 jffs2_node_frag，并由后者插入文件的红黑树中。如果对文件的相同区域进行修改，则将新的数据实体写入 flash 的同时，还要创建相应的 jffs2_raw_node_ref 和 jffs2_full_dnode，并将原有 jffs2_node_frag 数据结构的 node 域指向新的 jffs2_full_dnode，使其 frags 引用计数为 1，而原有的 frags 则递减为 0。

```
struct jffs2_node_frag
{
    rb_node_t rb;
    struct jffs2_full_dnode *node; /* NULL for holes */
    uint32_t size;
    uint32_t ofs;          /* Don't really need this, but optimisation */
};
```

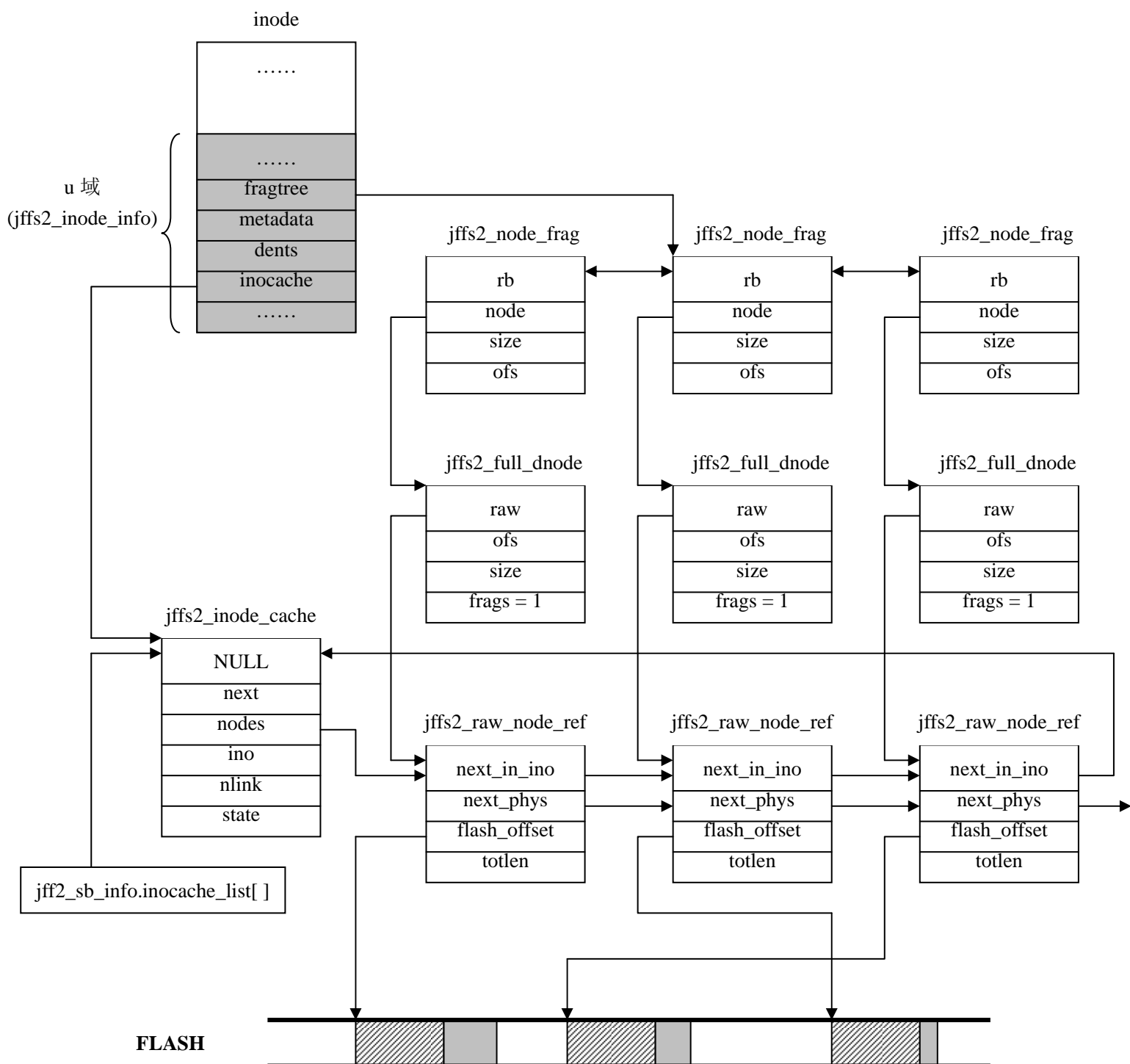
其中 rb 域用于组织红黑树，node 指针指向相应的 jffs2_full_dnode，size 和 ofs 也是从相应的 jffs2_full_dnode

复制而来，表示数据结点所代表的区域在文件内的偏移和长度。

总之，在打开一个正规文件时内核中创建的数据结构之间的关系如下图所示（假设文件由 3 个数据结点组成、没有画出 `file`、`dentry`、临时创建后又被删除的 `jffs2_tmp_dnode_info`）。

需要说明的是下图仅仅针对正规文件。对于目录文件、符号链接、设备文件都只有唯一的 `jffs2_raw_inode` 数据实体，目录文件没有“数据”，后二者的数据分别是被链接文件名和设备结点所代表的设备号。显然这一个数据实体对应的 `jffs2_full_dnode` 就没有必要用红黑树来组织了，而是由 `jffs2_inode_info` 中的 `metedata` 直接指向（在 [jffs2_do_read_inode](#) 函数中首先将它们的 `jffs2_full_dnode` 也向正规文件的那样加入红黑树，然后又改为由 `metadata` 直接指向）。

（另外，文件的目录项 `jffs2_raw_dirent` 数据结点则属于其父目录，所以没有出现在下图中。）



说明:



jffs2_raw_inode 实体及紧随其后的文件数据

图 2，正规文件的相关数据结构

第 2 章 描述jffs2 特性的数据结构（improved）

如果在配置内核时选择对 jffs2 文件系统的支持，则在内核启动时在 init 内核线程上下文中执行 jffs2 的注册工作：将其源代码中定义的 `file_system_type` 类型的变量 `jffs2_fs_type` 注册到由内核全局变量 `file_systems` 指向的链表中去，即用 jffs2 提供的 `jffs2_read_super` 方法来设置 `file_system_type` 类型变量的 `read_super` 函数指针，详见后文“注册文件系统”。如果内核引导命令行“`root=`”指定的设备上含有 jffs2 文件系统映象，则在内核启动时还将 jffs2 文件系统挂载为根文件系统。

在挂载文件系统时内核将为其创建相应的 VFS 对象 `super_block`，进而用具体文件系统注册的方法 `read_super` 读取设备，以填充、设置 `super_block` 数据结构，并建立根目录的 `inode`、`dentry` 等基本 VFS 设施。有些文件系统，比如 `ext2`，在磁盘上就有文件系统的超级块 `ext2_super_block`，因此这个方法主要是读取磁盘上的超级块，将其内容复制到内存中的 `super_block`。而 jffs2 在 flash 上没有超级块实体，所以在这个方法执行其它挂载所必须的操作，比如遍历 flash 为所有的数据实体和文件创建内核描述符、建立文件及其数据的映射关系等等，详见后文“挂载文件系统”。

文件系统超级块的u域：jffs2_sb_info数据结构

如上文所述，上层 VFS 框架的数据结构的设计应该尽可能多地概括各种不同文件系统之间的共性，从而使 VFS 框架具有良好的通用性。但对于各种不同文件系统的个性则用相应的 `union` 域来描述，由具体文件系统的实现来定义、解释、使用这些 `union`，从而实现具体文件系统的操作。在挂载具体文件系统时 `super_block` 的 `union` 域即被实例化为相应文件系统的私有数据结构，对于 jffs2 这个域实例化为 `jffs2_sb_info`：

```
struct jffs2_sb_info {
    struct mtd_info *mtd;
    uint32_t highest_ino;
    uint32_t checked_ino;
    unsigned int flags;
    struct task_struct *gc_task;           /* GC task struct */
    struct semaphore gc_thread_start;      /* GC thread start mutex */
    struct completion gc_thread_exit;      /* GC thread exit completion port */
    struct semaphore alloc_sem;            /* Used to protect all the following fields, and also to protect against
                                           out-of-order writing of nodes. And GC.*/
    uint32_t cleanmarker_size;             /* Size of an _inline_ CLEANMARKER
                                           (i.e. zero for OOB CLEANMARKER */

    uint32_t flash_size;
    uint32_t used_size;
    uint32_t dirty_size;
    uint32_t wasted_size;
    uint32_t free_size;
    uint32_t erasing_size;
    uint32_t bad_size;
```

```

uint32_t sector_size;
uint32_t unchecked_size;
uint32_t nr_free_blocks;
uint32_t nr_erasing_blocks;
uint32_t nr_blocks;
struct jffs2_eraseblock *blocks;      /* The whole array of blocks. Used for getting blocks
                                        * from the offset (blocks[ofs / sector_size]) */
struct jffs2_eraseblock *nextblock;   /* The block we're currently filling */
struct jffs2_eraseblock *gcblock;     /* The block we're currently garbage-collecting */

struct list_head clean_list;          /* Blocks 100% full of clean data */
struct list_head very_dirty_list;     /* Blocks with lots of dirty space */
struct list_head dirty_list;          /* Blocks with some dirty space */
struct list_head erasable_list;        /* Blocks which are completely dirty, and need erasing */
struct list_head erasable_pending_wbuf_list; /* Blocks which need erasing but only after
                                                the current wbuf is flushed */
struct list_head erasing_list;         /* Blocks which are currently erasing */
struct list_head erase_pending_list;    /* Blocks which need erasing now */
struct list_head erase_complete_list;   /* Blocks which are erased and need the clean marker
                                                written to them */
struct list_head free_list;            /* Blocks which are free and ready to be used */
struct list_head bad_list;             /* Bad blocks. */
struct list_head bad_used_list;        /* Bad blocks with valid data in. */

spinlock_t erase_completion_lock;      /* Protect free_list and erasing_list against erase
                                        completion handler */
wait_queue_head_t erase_wait;          /* For waiting for erases to complete */

struct jffs2_inode_cache **inocache_list;
spinlock_t inocache_lock;

/* Sem to allow jffs2_garbage_collect_deletion_dirent to drop the erase_completion_lock while it's holding a
pointer to an obsoleted node. I don't like this. Alternatives welcomed. */
struct semaphore erase_free_sem;

/* Write-behind buffer for NAND flash */
unsigned char *wbuf;
uint32_t wbuf_ofs;
uint32_t wbuf_len;
uint32_t wbuf_pagesize;
struct tq_struct wbuf_task;           /* task for timed wbuf flush */
struct timer_list wbuf_timer;         /* timer for flushing wbuf */

/* OS-private pointer for getting back to master superblock info */

```

```
void *os_priv;
};
```

其中部分域的说明如下，其它域的作用放到代码中说明：

文件系统是在具体设备上的特定数据组织形式。在向设备写入数据前，需要通过文件系统的相应方法将数据组织为特定的格式；在从设备读出数据后，需要通过文件系统的相应方法解释数据，而真正访问设备的工作是由设备驱动完成的。jffs2 是建立在 flash 上的文件系统，所以向 flash 写入、读出数据实体的操作最终通过 flash 驱动程序完成。jffs2_sb_info 的 mtd 域指向整个 flash（注意是包含若干分区的整个 flash）的 mtd_info 数据结构，该数据结构在安装、初始化 flash 设备驱动程序时创建，提供了访问 flash 的方法。从后文的读写操作分析可以看到，在向 flash 写入数据实体 jffs2_raw_inode 或者 jffs2_raw_dirent 及其后的数据时，最终要调用 mtd_info 中的相应方法。

在文件系统所在的设备上索引节点号是惟一的，highest_ino 记录了文件系统内最高的索引结点号。在 ext2 中索引结点 ext2_inode 的编号由其物理位置决定，而且在分配其物理位置时出于效率因素要综合考虑多种因素。而 jffs2 中事情就简单多了：每当新建文件时为之分配的索引结点号即为 highest_ino，并逐一递增该域，参见下文。

flags 为挂载文件系统时指定的各种标志，比如是否以只读方式挂载等等。

再次强调，一种文件系统是具体设备上的一种数据组织格式，因此必须针对具体设备的特点来设计文件系统的数据结构。而文件系统的超级块从宏观上描述了整个文件系统，所以针对具体设备的特点所设计的数据结构正体现在其超级块中。比如 ext2 为用于磁盘的文件系统，所以在其超级块 ext2_sb_info 中包含了磁盘分区上所有块组描述符、块组内索引结点的数量等与磁盘操作相关的数据结构。而 jffs2 是应用于 flash 的文件系统（有关 flash 的使用特点可参见讨论），所以其超级块包含了针对 flash 使用特点的数据结构，主要分为三个部分：为了使各 flash 擦除块都被均衡使用的各种 xxxx_list 链表和 GC 内核线程，以及用于组织所有文件的内核描述符的哈希表 inocache_list，而文件描述符的主要作用就是描述一个文件的所有离散的数据结点的位置等信息。下面我们逐一介绍这三个方面的数据结构。

没看懂

为了尽量推迟写 flash 的时机（也可以提高写一个擦除块的效率），jffs2 使用一个内核线程来执行垃圾回收（GC，即 Garbage Collecting），在需要时回收所有过时的数据结点，比如剩余干净的擦除块数量过低时、卸载 jffs2 时。创建这个内核线程后使用 gc_task 指向其 PCB，这样就可以直接给它发送各种信号从而控制其状态了。相关的数据结构还包括 gc_thread_start 和 gc_thread_exit。信号量 gc_thread_start 用于保证在当前进程（或 init 内核线程）在创建了 GC 内核线程后，在调用 kernel_thread 的函数返回前，GC 内核线程已经运行了；gc_thread_exit 是一个 completion 数据结构，定义如下：

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

其核心是一个等待队列 wait，整个数据结构由 wait.lock 保护。在 jffs2_stop_garbage_collect_thread 函数中通过给 GC 内核线程发送 SIGKILL 信号来结束它，当前执行流在发送完信号后就阻塞在 gc_thread_exit.wait 等待队列上；而 GC 内核线程处理 SIGKILL 信号时将唤醒受阻的执行流并退出，详见后文分析。

为了实现均衡地使用所有擦除块，jffs2 的方法必须记录各擦除块的使用情况和状态。这也就是各 xxxx_list 域和擦除块描述符数组 blocks[] 的目的了。与此相关还包括若干 xxxx_size 的域，重新罗列如下：

```
uint32_t flash_size;
uint32_t used_size;
uint32_t dirty_size;
uint32_t wasted_size;
uint32_t free_size;
uint32_t erasing_size;
uint32_t bad_size;
uint32_t sector_size;
uint32_t unchecked_size;
```

其中 flash_size 和 sector_size 的值在挂载文件系统时由 jffs2_sb_info.mtd 所指向的 flash 板块描述符 mtd_info 中相应的域复制过来，分别代表 jffs2 文件系统所在 flash 分区的大小和擦除块的大小。

在 flash 擦除块描述符 jffs2_eraseblock 中就设计了 used_size、dirty_size、wasted_size、free_size 域，它们分别表示当前擦除块内有效数据实体的空间大小、过时数据实体的空间大小、无法利用的空间大小和剩余空间大小。其中“无法利用的空间”是由于 flash 上数据结点之间必须是 4 字节地址对齐的，因此在数据结点之间可能存在间隙；或者由于填充；或者擦除块尾部的空间无法利用。那么 jffs2_sb_info 中这些域就是分区上所有擦除块相应域的求和。

jffs2_eraseblock 数据结构为擦除块描述符，所有擦除块的描述符都存放在 blocks[] 数组中。另外，根据擦除块的状态（即是否有数据、数据过时情况等信息）还将擦除块描述符组织在不同的 xxxx_list 链表中，以供文件系统的写方法和 GC 使用，从而实现对所有擦除块的均衡使用。根据作者的注释，各个 xxxx_list 域所指向链表的含义如下：

链表	链表中擦除块的性质
clean_list	只包含有效数据结点
very_dirty_list	所含数据结点大部分都已过时
dirty_list	至少含有一个过时数据结点
erasable_list	所有的数据结点都过时需要擦除。但尚未“调度”到 erase_pending_list
erasable_pending_wbuf_list	同 erase_pending_list，但擦除必须等待 wbuf 冲刷后
erasing_list	当前正在擦除
erase_pending_list	当前正等待擦除
erase_complete_list	擦除已完成，但尚未写入 CLEANMARKER
free_list	擦除完成，且已经写入 CLEANMARKER
bad_list	含有损坏单元
bad_used_list	含有损坏单元，但含有数据

最后，由于 jffs2 中不存在类似 ext_inode 的可以提供“索引”文件数据的机制，所以才在挂载文件系统时不得不遍历整个 flash 分区，建立每个文件所包含数据实体的位置和长度信息。数据实体通过其所属文件的内核描述符 jffs2_inode_cache 进行“索引”。所有文件的内核描述符被组织在一张哈希表中，即为 inocache_list 所指向的指针数组。

目录文件的内容由各目录项 `jffs2_raw_dirent` 数据实体组成，实现了“从文件名找到文件”的机制。在打开文件、逐层解析文件的路径名时通过访问父目录文件即可得到当前文件的目录项实体 `jffs2_raw_dirent`，进而得到文件的索引结点号 `ino` (`path_walk` 的细节可参见情景分析)，然后通过 `inocache_list` 哈希表就可以得到其 `jffs2_inode_cache` 的指针，然后通过其 `nodes` 域得到文件所有数据实体的内核描述符 `jffs2_raw_node_ref` 组成的链表，从而最终得到所有数据实体在 flash 上的位置、长度信息。

文件索引结点的u域：jffs2_inode_info数据结构

在打开文件创建其 `inode` 时由具体文件系统提供的 `read_inode` 方法来初始化 `inode` 的 `u` 域。无论底层具体文件系统中“索引结点”的形式如何，上层 VFS 的 `inode` 都完整、惟一描述了文件的所有管理信息（其在文件系统内的组织信息由 `dentry` 描述），其中就必然包括索引文件数据的机制。由于在不同文件系统中数据的组织、存储格式不同，索引文件数据的机制显然应该放到 `inode` 的 `u` 域中。比如 `ext2` 中通过 `ext2_inode_info` 的 `i_data[]` 来索引文件数据所在的磁盘块，该数组内容从 `ext2_inode` 的 `i_block[]` 数组复制过来；而 `jffs2` 中通过 `jffs2_inode_info` 的 `fragtree/metedata`，或者 `dents` 来组织文件的所有数据实体的内核描述符。

```
struct jffs2_inode_info {
    struct semaphore sem;
    uint32_t highest_version;    /* The highest (datanode) version number used for this ino */
    rb_root_t fragtree;         /* List of data fragments which make up the file */

    /* There may be one datanode which isn't referenced by any of the above fragments, if it contains a metadata
    update but no actual data - or if this is a directory inode. This also holds the _only_ dnode for
    symlinks/device nodes, etc. */
    struct jffs2_full_dnode *metadata;
    struct jffs2_full_dirent *dents;    /* Directory entries */

    /* Some stuff we just have to keep in-core at all times, for each inode. */
    struct jffs2_inode_cache *inocache;
    uint16_t flags;
    uint8_t usercompr;
#ifdef LINUX_VERSION_CODE > KERNEL_VERSION(2,5,2)
    struct inode vfs_inode;
#endif
};
```

由于 `inode` 的 `i_sem` 在 `generic_file_write/read` 期间一直被当前执行流持有，用于实现上层用户进程之间的同步。所以在读写操作期间还要使用信号量的话，就必须设计额外的信号量。在 `jffs2_inode_info` 中设计的 `sem` 信号量用于实现底层读写执行流与 GC 之间的同步。（这是因为 GC 的本质就是将有效数据实体的副本写到其它的擦除块中去，即还是通过写入操作完成的，所以需要与其它写入操作同步，详见[后文](#)。）

一个文件的所有数据实体（无论过时与否）都有唯一的 `version` 号。当前所使用的最高 `version` 号由 `highest_version` 域记录。

正规文件包含若干 `jffs2_raw_inode` 数据实体，它们的内核描述符 `jffs2_raw_node_ref` 组成的链表由 `jffs2_inode_cache` 的 `nodes` 指向。在打开文件时还创建相应的 `jffs2_full_dnode` 和 `jffs2_node_frag` 数据结构，并由后者组织在由 `fragtree` 指向的红黑树中。详见图 2。

由于目录文件、符号链接和设备文件只有一个 `jffs2_raw_inode` 数据实体所以没有必要使用红黑树。所以在 `jffs2_do_read_inode` 函数中先象对待正规文件那样先将它们的 `jffs2_full_dnode` 加入红黑树，然后又改为由 `metadata` 直接指向。如果是目录文件，则在打开文件时为数据实体的内核描述符 `jffs2_raw_node_ref` 创建相应的 `jffs2_full_dirent`，并组织为由 `dents` 指向链表，详见图 1。

最后，`inocache` 指向该文件的内核描述符 `jffs2_inode_cache` 数据结构。

打开正规文件后相关数据结构之间的引用关系

如前所述，在挂载文件系统后即创建了 `super_block` 数据结构，为根目录创建了 `inode`、`dentry`，为所有的文件创建了 `jffs2_inode_cache` 及每一个数据实体的描述符 `jffs2_raw_node_ref`；在打开文件时创建 `file`、`dentry`、`inode`，并为数据实体描述符创建相应的 `jffs2_full_dnode` 或者 `jffs2_full_dirent` 等数据结构。打开正规文件后相关数据结构的关系如下图所示：（没有画出根目录的 `inode`）

（注意，虚线框中的数据结构在打开文件时才创建，其它数据结构在挂载文件系统时就已经创建好了）

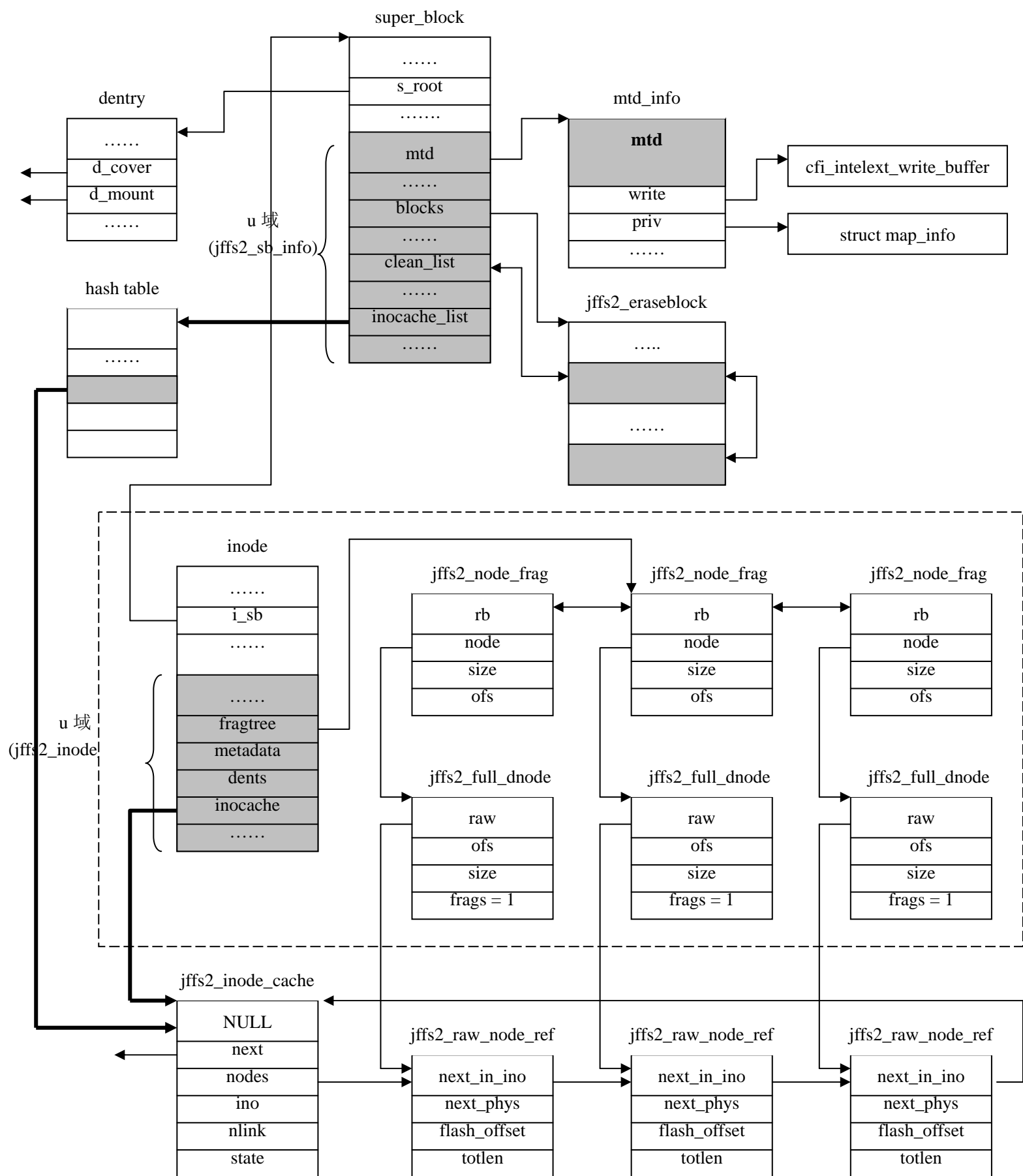


图 3, 打开正规文件后相关数据结构之间的引用关系

第3章 注册文件系统

在 linux 上使用一个文件系统之前必须完成安装和注册。如果在配置内核时选择对文件系统的支持，那么其代码被静态链接到内核中，在内核初始化期间 init 内核线程将完成文件系统的注册。或者在安装文件系统模块时在模块初始化函数中完成注册。

一旦完成注册，这种文件系统的方法对内核就是可用的了，以后就可以用 mount 命令将其挂载到根文件系统的某个目录结点上。而在内核初始化期间注册、挂载根文件系统。在设备上根目录文件由其子目录、子文件的目录项 jffs2_raw_dirent 数据实体组成，根目录在内核中的挂载点为 “/”，相应的 dentry 和 inode 在内核初始化时由 mount_root 函数创建。

init_jffs2_fs函数

在配置内核时选择对 jffs2 的支持，那么 jffs2 的源代码编译后被静态链接入内核映像，在初始化期间 init 内核线程执行 init_jffs2_fs 函数完成 jffs2 的注册：

```
static int __init init_jffs2_fs(void)
{
    int ret;
    printk(KERN_NOTICE "JFFS2 version 2.1. (C) 2001, 2002 Red Hat, Inc., designed by Axis
                        Communications AB.\n");
#ifdef JFFS2_OUT_OF_KERNEL
    /* sanity checks. Could we do these at compile time? */
    if (sizeof(struct jffs2_sb_info) > sizeof(((struct super_block *)NULL)->u)) {
        printk(KERN_ERR "JFFS2 error: struct jffs2_sb_info (%d bytes) doesn't fit in the super_block union
                        (%d bytes)\n", sizeof(struct jffs2_sb_info), sizeof(((struct super_block *)NULL)->u));
        return -EIO;
    }
    if (sizeof(struct jffs2_inode_info) > sizeof(((struct inode *)NULL)->u)) {
        printk(KERN_ERR "JFFS2 error: struct jffs2_inode_info (%d bytes) doesn't fit in the inode union (%d
                        bytes)\n", sizeof(struct jffs2_inode_info), sizeof(((struct inode *)NULL)->u));
        return -EIO;
    }
}
#endif
```

VFS 的 super_block 以及 inode 的最后一个域都为一个共用体，将在挂载文件系统时实例化为具体文件系统的私有数据结构。这里首先检查 super_block 和 inode 的 u 域是否能够容纳 jffs2 的相关私有数据结构。

```
ret = jffs2_zlib_init();
if (ret) {
    printk(KERN_ERR "JFFS2 error: Failed to initialise zlib workspaces\n");
    goto out;
```

```
}
```

jffs2 文件系统支持压缩和解压缩，在将数据实体写入 flash 前可以使用 zlib 库的压缩算法进行压缩，从 flash 读出后进行解压缩。在 `jffs2_zlib_init` 函数中为压缩、解压缩分配空间 `deflate_workspace` 和 `inflate_workspace`。

```
ret = jffs2_create_slab_caches();
if (ret) {
    printk(KERN_ERR "JFFS2 error: Failed to initialise slab caches\n");
    goto out_zlib;
}
```

为数据实体 `jffs2_raw_dirent` 和 `jffs2_raw_inode`、数据实体内核描述符 `jffs2_raw_node_ref`、文件的内核描述符 `jffs2_inode_cache`、`jffs2_full_dnode` 和 `jffs2_node_frag` 等数据结构通过 `kmem_cache_create` 函数创建相应的内存高速缓存（这些数据结构都是频繁分配、回收的对象，因此使用内存高速缓存再合适不过了。另外通过 `slab` 的着色能够使不同 `slab` 内对象的偏移地址不尽相同，从而映射到不同的处理器高速缓存行上）。

```
ret = register_filesystem(&jffs2_fs_type);
if (ret) {
    printk(KERN_ERR "JFFS2 error: Failed to register filesystem\n");
    goto out_slab;
}
return 0;
out_slab:
    jffs2_destroy_slab_caches();
out_zlib:
    jffs2_zlib_exit();
out:
    return ret;
}
```

最后，就是通过 `register_filesystem` 函数向系统注册 `jffs2` 文件系统了。使用 `mount` 命令挂载某一文件系统前，它必须事先已经向系统注册过了。每一个已注册过的文件系统都由数据结构 `file_system_type` 描述：

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*read_super) (struct super_block *, void *, int);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
};
```

所有已注册的文件系统的 `file_system_type` 通过 `next` 域组织成一个链表，链表由内核全局变量 `file_systems` 指向。`name` 域用于描述文件系统的名称，由 `find_filesystem` 函数在链表中查找指定名称的文件系统时使用。

`fs_flags` 指明了文件系统的一些特性，比如文件系统是否只支持一个超级块结构、是否允许用户使用 `mount` 命令挂载等等，详见 `linux/fs.h` 文件。

`file_system_type` 中最重要的域就是函数指针 `read_super` 了。这个函数指针即为上层 VFS 框架与具体文件系统的特定挂载方法之间的接口，VFS 框架中挂载文件系统的代码向下只调用到该函数指针，从而实现与具体文件系统方法的无关性。在注册具体文件系统时实例化该函数指针为相应的方法，从而按照特定格式来创建、初始化文件系统的超级块。

另外根据注释，任何文件系统的 `file_syste_type` 自注册之时其就必须一直存在在内核中，直到其被注销。无论文件系统是否被挂载都不应该释放 `file_system_type` 数据结构。

在 `jffs2` 源代码中实现了所有 `jffs2` 的方法，并通过如下的宏定义了 `file_system_type` 数据结构：

```
static DECLARE_FSTYPE_DEV(jffs2_fs_type, "jffs2", jffs2_read_super);
```

这个宏定义在 `linux/fs.h` 中：

```
#define DECLARE_FSTYPE(var,type,read,flags) \
struct file_system_type var = { \
    name:      type, \
    read_super: read, \
    fs_flags:   flags, \
    owner:      THIS_MODULE, \
}
```

由此可见，在 `jffs2` 源代码文件中定义了 `file_system_type` 类型的变量 `jffs2_fs_type`，其名字为“`jffs2`”，而“`read_super`”方法为 `jffs2_read_super`，它是具体文件系统所提供的各种方法的“总入口”，将在后文挂载文件系统时详细分析。

register_filesystem函数

```
int register_filesystem(struct file_system_type * fs)
{
    int res = 0;
    struct file_system_type ** p;

    if (!fs)
        return -EINVAL;
    if (fs->next)
        return -EBUSY;

    INIT_LIST_HEAD(&fs->fs_supers);
    write_lock(&file_systems_lock);
    p = find_filesystem(fs->name);
```

```

if (*p)    //若已注册过
    res = -EBUSY;
else      //否则，将新的 file_system_type 结构加入到 file_systems 链表的末尾
    *p = fs;
write_unlock(&file_systems_lock);
return res;
}

```

如前所述，所有已注册文件系统的 `file_system_type` 组成一个链表，由内核全局变量 `file_systems` 指向。注册文件系统就是将其 `file_system_type` 加入到这个链表中。当然，在访问链表期间必须首先获得锁 `file_system_locks`。

如果 `name` 命名文件系统已经注册过了，则 `find_filesystems` 函数返回其 `file_system_type` 结构的地址，否则返回内核 `file_systems` 链表的末尾元素 `next` 域的地址：

```

static struct file_system_type **find_filesystem(const char *name)
{
    struct file_system_type **p;
    for (p=&file_systems; *p; p=&(*p)->next)
        if (strcmp((*p)->name, name) == 0)
            break;
    return p;
}

```

由此可见，`register_filesystem` 函数就是将新的文件系统的 `file_system_type` 加入到 `file_systems` 链表的末尾。

第4章 挂载文件系统（improved）

如前所述，在 jffs2 源代码中定义了 `file_system_type` 类型的全局变量 `jffs2_fs_type`，并将其注册到内核的 `file_systems` 链表中去。需要说明的是 `file_system_type` 中的 `read_super` 函数指针所指向的方法是具体文件系统所提供的各种方法的“总入口”，从时间上看各种方法的“引入时机”如下：

1. 在注册时实例化 `read_super` 方法；
2. 在挂载文件系统、初始化超级块时实例化 `super_operations` 方法表；
3. 在打开文件、创建 `inode` 时调用 `super_operations` 方法表的 `read_inode` 方法，从而根据文件的类型将 `inode.i_op` 实例化为具体的 `file_operations` 方法表，比如正规文件的 `jffs2_file_operations` 方法表；
4. 在读写文件时根据 `file_operations` 接口中的函数指针调用 `jffs2` 的具体方法。

在挂载文件系统时内核为之创建 VFS 的 `super_block` 数据结构，以及根目录的 `inode`、`dentry` 等数据结构。挂载根文件系统时函数调用链如下：“>”表示调用）

`mount_root > mount_block_root > sys_mount > do_mount > get_sb_bdev > read_super > jffs2_read_super`

在 `read_super` 函数中将由 `get_empty_super` 函数分配一个 `super_block` 数据结构，稍后调用相应文件系统注册的 `read_super` 方法初始化 `super_block` 数据结构（这个调用链中各个函数的细节可参见情景分析，在此不再赘述）。

jffs2_read_super函数

这个函数在初始化 VFS 超级块对象时为 `flash` 上所有的数据实体和文件建立内核描述符。内核描述符是数据实体和文件的“地图”，由于在 `flash` 中缺少对文件数据的索引机制，所以早在挂载文件系统时必须建立文件及其数据实体的映射关系。而文件的其它 `jffs2` 数据结构，比如 `jffs2_full_dnode` 或 `jffs2_full_dirent` 等要在打开文件时才被创建。

```
static struct super_block *jffs2_read_super(struct super_block *sb, void *data, int silent)
{
    struct jffs2_sb_info *c;
    int ret;
    unsigned long j, k;
    D1(printk(KERN_DEBUG "jffs2: read_super for device %s\n", kdevname(sb->s_dev)));

    if (major(sb->s_dev) != MTD_BLOCK_MAJOR) {
        if (!silent)
            printk(KERN_DEBUG "jffs2: attempt to mount non-MTD device %s\n", kdevname(sb->s_dev));
        return NULL;
    }
```

由 `read_super` 函数调用 `jffs2_read_super` 函数

在 `read_super` 函数中已经将 `super_block.bdev` 设置为 `jffs2` 文件系统所在 `flash` 分区的设备号了，再次检查设备号是否正确。

```

c = JFFS2_SB_INFO(sb);
memset(c, 0, sizeof(*c));
sb->s_op = &jffs2_super_operations;
c->mtd = get_mtd_device(NULL, minor(sb->s_dev));
if (!c->mtd) {
    D1(printk(KERN_DEBUG "jffs2: MTD device #%u doesn't appear to exist\n", minor(sb->s_dev)));
    return NULL;
}

```

JFFS2_SB_INFO 宏返回 **super_block** 的 **u** 域（即 **jffs2_sb_info** 数据结构）的地址。首先将整个 **jffs2_sb_info** 数据结构清空，然后设置文件系统方法表的指针 **s_op** 指向 **jffs2_super_operations** 方法表，它提供了访问整个文件系统的基本方法。

```

static struct super_operations jffs2_super_operations =
{
    read_inode:  jffs2_read_inode,
    put_super:    jffs2_put_super,
    write_super: jffs2_write_super,
    statfs:       jffs2_statfs,
    remount_fs:   jffs2_remount_fs,
    clear_inode:  jffs2_clear_inode
};

```

作为策略的上层 VFS 框架通过各种数据结构中的函数指针接口实现与提供机制的底层具体文件系统的隔离。在 VFS 数据结构中只定义了一个指向具体方法表的指针变量，而由具体文件系统实现方法表中定义的各个函数（当然如果具体文件系统不支持某些方法也可以不用实现，因此上层 VFS 框架在调用函数指针指向的底层方法之前必须首先检查函数指针是否有效，即底层是否支持具体的方法。题外话。），并定义函数指针接口变量，最后将 VFS 数据结构中的方法表指针指向这个函数指针接口即可。

另外一个关键设置就是将 **jffs2_sb_info.mtd** 指向在初始化 flash 设备驱动程序时创建的 **mtd_info** 数据结构，它物理上描述了整个 flash 板块并提供了访问 flash 的底层驱动程序。从后文可见，**jffs2** 方法最终通过调用 flash 驱动程序中将数据实体 **jffs2_raw_dirent** 或 **jffs2_raw_inode** 及后继数据块写入 flash（或从中读出）。

```

j = jiffies;
ret = jffs2_do_fill_super(sb, data, silent);
k = jiffies;
if (ret) {
    put_mtd_device(c->mtd);
    return NULL;
}
printk("JFFS2 mount took %ld jiffies\n", k-j);
return sb;
}

```

真正初始化 VFS 超级块 `super_block` 数据结构、为 flash 上所有数据实体建立内核描述符 `jffs2_raw_node_ref`、为所有文件创建内核描述符 `jffs2_inode_cache` 的任务交给 `jffs2_do_fill_super` 函数完成。

jffs2_do_fill_super函数

```
int jffs2_do_fill_super(struct super_block *sb, void *data, int silent)
{
    struct jffs2_sb_info *c;
    struct inode *root_i;
    int ret;

    c = JFFS2_SB_INFO(sb);
    c->sector_size = c->mtd->erasesize;
    c->flash_size = c->mtd->size;
    if (c->flash_size < 4*c->sector_size) {
        printk(KERN_ERR "jffs2: Too few erase blocks (%d)\n", c->flash_size / c->sector_size);
        return -EINVAL;
    }
    c->cleanmarker_size = sizeof(struct jffs2_unknown_node);
    if (jffs2_cleanmarker_oob(c)) { /* Cleanmarker is out-of-band, so inline size zero */
        c->cleanmarker_size = 0;
    }
}
```

首先根据 `mtd_info` 数据结构的相应域来设置 `jffs2_sb_info` 中与 flash 参数有关的域：擦除块大小和分区大小（`mtd_info` 数据结构在 flash 驱动程序初始化中已创建好）。jffs2 驱动在成功擦除了一个擦除块后，要写入类型为 `CLEANMARKER` 的数据实体来标记擦除成功完成。如果为 NOR flash，则 `CLEANMARKER` 写在擦除块内部，`cleanmarker_size` 即为该数据实体的大小；如果为 NAND flash，则它写在 `oob` (Out_Of_Band) 区间内而不占用擦除块空间，所以将 `cleanmarker_size` 清 0。（NAND flash 可以看作是一组“page”，每个 page 都有一个 oob 空间。在 oob 空间内可以存放 ECC (Error CorreCtion) 代码、或标识含有错误的擦除块的信息、或者与文件系统相关的信息。jffs2 就利用了 oob 来存放 `CLEANMARKER`）

```
if (c->mtd->type == MTD_NANDFLASH) {
    /* Initialise write buffer */
    c->wbuff_pagesize = c->mtd->oobblock;
    c->wbuff_ofs = 0xFFFFFFFF;
    c->wbuff = kmalloc(c->wbuff_pagesize, GFP_KERNEL);
    if (!c->wbuff)
        return -ENOMEM;
    /* Initialize process for timed wbuff flush */
    INIT_TQUEUE(&c->wbuff_task, (void*) jffs2_wbuff_process, (void *)c);
    /* Initialize timer for timed wbuff flush */
    init_timer(&c->wbuff_timer);
    c->wbuff_timer.function = jffs2_wbuff_timeout;
}
```

```

c->wbuf_timer.data = (unsigned long) c;
}

```

NAND flash 由一组 “page” 组成，若干 page 组成一个擦除块。读写操作的最小单元是 page，擦除操作的最小单元是擦除块。flash 描述符 mtd_info 的 oobblock 域即 page 的大小，所以这里分配 oobblock 大小的写缓冲区，以及周期地将该写缓冲区刷新（或同步）到 flash 的内核定时器及一个任务队列元素。由内核定时器周期性地把 jffs2_sb_info.wbuf_task 通过 schedule_task 函数调度给 keventd 执行，相应的回调函数为 jffs2_wbuf_process，它将 jffs2_sb_info.wbuf 写缓冲区的内容写回 flash。（注意，flash 的写操作可能阻塞，因此必须放到进程上下文文中进行，所以交给 keventd 来完成）

```

c->inocache_list = kmalloc(INOCACHE_HASHSIZE * sizeof(struct jffs2_inode_cache *),
                           GFP_KERNEL);
if (!c->inocache_list) {
    ret = -ENOMEM;
    goto out_wbuf;
}
memset(c->inocache_list, 0, INOCACHE_HASHSIZE * sizeof(struct jffs2_inode_cache *));

```

如前所述 flash 上的任何文件都有唯一的内核描述符 jffs2_inode_cache 数据结构，用于建立文件及其数据实体之间的映射关系，在挂载文件系统创建。在打开文件、创建 inode 时，用 inode 的 u 域即 jffs2_inode_info 数据结构的 inocache 域指向它，参见图 3。所有文件的 jffs2_inode_cache 数据结构又被组织到一张哈希表里，由 jffs2_sb_info.inocache_list 指向。

```

if ((ret = jffs2_do_mount_fs(c)))
    goto out_inohash;
ret = -EINVAL;

```

这个函数完成挂载 jffs2 文件系统的绝大部分工作，详见下文分析，这里仅罗列之：

1. 创建擦除块描述符数组 jffs2_sb_info.blocks[] 数组，初始化 jffs2_sb_info 的相应域；
2. 扫描整个 flash 分区，为所有的数据实体建立内核描述符 jffs2_raw_node_ref、为所有的文件创建内核描述符 jffs2_inode_cache；
3. 将所有文件的 jffs2_inode_cache 加入 hash 表，检查 flash 上所有数据实体的有效性（注意，只检查了数据实体 jffs2_raw_dirent 或 jffs2_raw_inode 自身的 crc 校验值，而把后继数据的 crc 校验工作延迟到了真正打开文件时，参见 jffs2_scan_inode_node 函数）；
4. 根据擦除块的内容，将其描述符加入 jffs2_sb_info 中相应的 xxxx_list 链表。

```

D1(printk(KERN_DEBUG "jffs2_do_fill_super(): Getting root inode\n"));
root_i = iget(sb, 1);
if (is_bad_inode(root_i)) {
    D1(printk(KERN_WARNING "get root inode failed\n"));
    goto out_nodes;
}
D1(printk(KERN_DEBUG "jffs2_do_fill_super(): d_alloc_root()\n"));
sb->s_root = d_alloc_root(root_i);

```

```
if (!sb->s_root)
    goto out_root_i;
```

为 flash 上所有文件、所有数据实体创建相应的内核描述符后，就已经完成了挂载 jffs2 文件系统的大部分工作，下面就为根目录“/”创建 VFS 的 inode 和 dentry 了。除根目录外任何文件的 inode 和 dentry 等数据结构都是等到打开文件时才创建。由于根目录文件是 path_walk 解析路径名的出发点，所以早在挂载文件系统时就打开了根目录文件。文件系统超级块 super_block 的 s_root 指针指向根目录的 dentry，而 dentry 的 d_inode 指向其 inode，而 inode 的 i_sb 又指向文件系统超级块 super_block。

创建 inode 的工作由 iget 内联函数完成，注意传递的第二个参数为相应 inode 的索引节点编号，而根目录的索引节点编号为 1。iget 函数的函数调用路径为：

```
iget > iget4 > get_new_inode > super_operations.read_inode (指向 jffs2_read_inode)
```

为文件创建 inode 时，首先根据其索引节点编号 ino 在索引节点哈希表 inode_hashtable 中查找，如果尚未创建，则调用 get_new_inode 函数分配一个 inode 数据结构，并用相应文件系统已注册的 read_super 方法初始化。对于 ext2 文件系统，相应的 ext2_read_inode 函数将读出磁盘索引结点，而对于 jffs2 文件系统，若为目录文件，则为目录文件的所有 jffs2_raw_dirent 目录项创建相应的 jffs2_full_dirent 数据结构并组织为链表，并为其唯一的 jffs2_raw_inode 创建 jffs2_full_dnode 数据结构，并由 jffs2_inode_info 的 metadata 直接指向（对符号链接和设备文件的唯一的 jffs2_raw_inode 的处理与此相同）；若为正规文件，则为数据结点创建相应的 jffs2_full_dnode 和 jffs2_node_frag 数据结构，并由后者组织到红黑树中，最后根据文件的类型设置索引结点方法表指针 inode.i_op/i_fop/i_mapping，详见后文。

```
#if LINUX_VERSION_CODE >= 0x20403
    sb->s_maxbytes = 0xFFFFFFFF;
#endif
    sb->s_blocksize = PAGE_CACHE_SIZE;
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic = JFFS2_SUPER_MAGIC;
    if (!(sb->s_flags & MS_RDONLY))
        jffs2_start_garbage_collect_thread(c);
    return 0;
out_root_i:
    iput(root_i);
out_nodes:
    jffs2_free_ino_caches(c);
    jffs2_free_raw_node_refs(c);
    kfree(c->blocks);
out_inohash:
    kfree(c->inocache_list);
out_wbuf:
    if (c->wbuf)
        kfree(c->wbuf);
    return ret;
```

```
}
```

挂载文件系统的最后还要设置 `jffs2_sb_info` 中的几个域，比如页缓冲区中的页面大小 `s_blocksize`，标识文件系统的“魔数” `s_magic`。另外，就是要启动 GC（Garbage Collecting，垃圾回收）内核线程了。`jffs2` 日志文件系统的特点就是任何修改都会向 `flash` 中写入新的数据结点，而不该动原有的数据结点。当 `flash` 可用擦除块数量低于一定的阈值后，就得唤醒 GC 内核线程回收所有“过时的”数据结点所占的空间了。有关 GC 机制详见第 8 章。

如果成功挂载则返回 0，否则释放所有的描述符及各种获得的空间并返回 `ret` 中保存的错误码。

jffs2_do_mount_fs函数

在这个函数中仅为 `flash` 分区上所有的擦除块分配描述符并初始化各种 `xxxx_list` 链表首部，然后调用 `jffs2_build_filesystem` 函数完成挂载文件系统的绝大部分操作。

```
int jffs2_do_mount_fs(struct jffs2_sb_info *c)
{
    int i;

    c->free_size = c->flash_size;
    c->nr_blocks = c->flash_size / c->sector_size;
    c->blocks = kmalloc(sizeof(struct jffs2_eraseblock) * c->nr_blocks, GFP_KERNEL);
    if (!c->blocks)
        return -ENOMEM;
```

在挂载文件系统之前，认为整个 `flash` 都是可用的，所以设置空闲空间大小为整个 `flash` 分区的大小，并计算擦除块总数。`jffs2_eraseblock` 数据结构是擦除块描述符，这里为分配所有擦除块描述符的空间并初始化：

```
for (i=0; i<c->nr_blocks; i++) {
    INIT_LIST_HEAD(&c->blocks[i].list);
    c->blocks[i].offset = i * c->sector_size;
    c->blocks[i].free_size = c->sector_size;
    c->blocks[i].dirty_size = 0;
    c->blocks[i].wasted_size = 0;
    c->blocks[i].unchecked_size = 0;
    c->blocks[i].used_size = 0;
    c->blocks[i].first_node = NULL;
    c->blocks[i].last_node = NULL;
}
```

其中 `offset` 域为擦除块在 `flash` 分区内的逻辑偏移，`free_size` 为其大小。此时所有记录擦除块使用状况的 `xxxx_size` 域都为 0，它们分别表示（按照代码中的出现顺序）擦除块中过时数据实体所占空间、由于填充和对齐浪费的空间、尚未进行 `crc` 校验的数据实体所占的空间、有效的数据实体所占的空间。一个擦除块内所有数据实体的内核描述符 `jffs2_raw_node_ref` 由其 `next_phys` 域组织成一个链表，其首尾元素分别由

first_node 和 last_node 指向。

```

init_MUTEX(&c->alloc_sem);
init_MUTEX(&c->erase_free_sem);
init_waitqueue_head(&c->erase_wait);
spin_lock_init(&c->erase_completion_lock);
spin_lock_init(&c->inocache_lock);

INIT_LIST_HEAD(&c->clean_list);
INIT_LIST_HEAD(&c->very_dirty_list);
INIT_LIST_HEAD(&c->dirty_list);
INIT_LIST_HEAD(&c->erasable_list);
INIT_LIST_HEAD(&c->erasing_list);
INIT_LIST_HEAD(&c->erase_pending_list);
INIT_LIST_HEAD(&c->erasable_pending_wbuf_list);
INIT_LIST_HEAD(&c->erase_complete_list);
INIT_LIST_HEAD(&c->free_list);
INIT_LIST_HEAD(&c->bad_list);
INIT_LIST_HEAD(&c->bad_used_list);
c->highest_ino = 1;

```

在下面的jffs2_build_filesystem函数将根据所有擦除块的使用情况将各个擦除块的描述符插入不同的链表，这里首先初始化这些链表指针xxxx_list。文件的索引结点号在某个设备上的文件系统内部才唯一，当前flash分区的jffs2 文件系统中最高的索引结点号由jffs2_sb_info的highest_ino域记录，而 1 是根目录“/”的索引结点号。由于jffs2 中不需要象ext2 那样考虑优先将文件数据放到其父目录所在块组中、以及平衡各磁盘块组中目录的数目，所以jffs2 中新文件的索引结点号分配策略非常简单，直接设置为highest_ino即可，并同时递增之。参见[下文](#)。

```

if (jffs2_build_filesystem(c)) {
    D1(printk(KERN_DEBUG "build_fs failed\n"));
    jffs2_free_ino_caches(c);
    jffs2_free_raw_node_refs(c);
    kfree(c->blocks);
    return -EIO;
}
return 0;
}

```

下面就是由 jffs2_build_filesystem 函数真正完成 jffs2 文件系统的挂载工作了。

jffs2_build_filesystem函数

上文罗列了 jffs2_do_mount_fs 函数完成挂载 jffs2 文件系统的绝大部分工作：

1. 创建擦除块描述符数组 jffs2_sb_info.blocks[]数组，初始化 jffs2_sb_info 的相应域；

2. 扫描整个 flash 分区，为所有的数据实体建立内核描述符 `jffs2_raw_node_ref`、为所有的文件创建内核描述符 `jffs2_inode_cache`;
3. 将所有文件的 `jffs2_inode_cache` 加入 `inocache_list` 哈希表，检查 flash 上所有数据结点的有效性;
4. 根据擦除块的内容，将其描述符加入 `jffs2_sb_info` 中相应的 `xxxx_list` 链表

除了第一条外其余的工作都是由 `jffs2_build_filesystem` 函数完成的，我们分段详细分析这个函数：

```
static int jffs2_build_filesystem(struct jffs2_sb_info *c)
{
    int ret;
    int i;
    struct jffs2_inode_cache *ic;

    /* First, scan the medium and build all the inode caches with lists of physical nodes */
    c->flags |= JFFS2_SB_FLAG_MOUNTING;
    ret = jffs2_scan_medium(c);
    c->flags &= ~JFFS2_SB_FLAG_MOUNTING;
    if (ret)
        return ret;
```

由 `jffs2_scan_medium` 函数遍历 flash 分区上的所有的擦除块，读取每一个擦除块上的所有数据实体，建立相应的内核描述符 `jffs2_raw_node_ref`，为每个文件建立内核描述符 `jffs2_inode_cache`，并建立相互连接关系；如果是目录文件，则为其所有目录项创建相应的 `jffs2_full_dirent` 并组织为链表，由 `jffs2_inode_cache` 的 `scan_dents` 域指向；并将 `jffs2_inode_cache` 加入 `inocache_list` 哈希表；最后，根据擦除块的使用情况将其描述符 `jffs2_eraseblock` 加入 `jffs2_sb_info` 中的 `xxxx_list` 链表。详见下文分析。**注意在挂载文件系统期间要设置超级块中的 `JFFS2_SB_FLAG_MOUNTING` 标志。**

剩下的工作分为三个阶段完成：为每个文件计算硬链接计数、删除硬链接为 0 的文件、最后释放每个目录项 `jffs2_raw_dirent` 所对应的上层 `jffs2_full_dirent` 数据结构。

```
D1(printk(KERN_DEBUG "Scanned flash completely\n"));
D1(jffs2_dump_block_lists(c));
/* Now scan the directory tree, increasing nlink according to every dirent found. */
for_each_inode(i, c, ic) {
    D1(printk(KERN_DEBUG "Pass 1: ino #%u\n", ic->ino));
    ret = jffs2_build_inode_pass1(c, ic);
    if (ret) {
        D1(printk(KERN_WARNING "Eep. jffs2_build_inode_pass1 for ino %d returned %d\n",
                        ic->ino, ret));
        return ret;
    }
    cond_resched();
}
```

上面 `jffs2_scan_medium` 已经为 flash 上所有的数据实体和文件创建了内核描述符，并且进一步为所有的目

录项数据实体 `jffs2_raw_dirent` 创建了临时的 `jffs2_full_dirent` 数据结构（它们将在 `jffs2_build_filesystem` 函数的最后删除，目的只是计算所有文件的硬链接计数），目录文件的作用就是实现“从路径名找到文件”的机制，其物质基础就是组成目录文件的各个目录项。在 `path_walk` 逐层解析路径名时将逐一打开各级目录文件，建立其 `dentry` 以及彼此之间的联系，此时就在内核中建立起相应的系统目录树分支。

`for_each_inode` 宏用于访问所有文件的内核描述符，定义如下：

```
#define for_each_inode(i, c, ic) \
    for (i=0; i<INOCACHE_HASHSIZE; i++) \
        for (ic=c->inocache_list[i]; ic; ic=ic->next)
```

所以这里使用这个宏对每一个文件都调用了 `jffs2_build_inode_pass1` 函数，它为目录文件下的所有目录项增加其所指文件的硬链接计数。下面要再次遍历所有文件的内核描述符删除那些 `nlink` 为 0 的文件。如果它是目录，那么还要减小其下的所有子目录、文件的硬链接计数。这就是第 2 个阶段的工作：

上面的操作可能比较耗时，因此 `cond_resched` 宏用于让出 `cpu`，定义如下：

```
#define cond_resched()    do { if need_resched() schedule(); } while(0)
```

```
D1(printk(KERN_DEBUG "Pass 1 complete\n"));
D1(jffs2_dump_block_lists(c));
/* Next, scan for inodes with nlink == 0 and remove them. If they were directories, then decrement the nlink
of their children too, and repeat the scan. As that's going to be a fairly uncommon occurrence, it's not so evil
to do it this way. Recursion bad. */
do {
    D1(printk(KERN_DEBUG "Pass 2 (re)starting\n"));
    ret = 0;
    for_each_inode(i, c, ic) {
        D1(printk(KERN_DEBUG "Pass 2: ino %#u, nlink %d, ic %p, nodes %p\n", ic->ino, ic->nlink, ic,
            ic->nodes));

        if (ic->nlink)
            continue;

        /* XXX: Can get high latency here. Move the cond_resched() from the end of the loop? */
        ret = jffs2_build_remove_unlinked_inode(c, ic);
        if (ret)
            break;

        /* -EAGAIN means the inode's nlink was zero, so we deleted it, and furthermore that it had children and
        their nlink has now gone to zero too. So we have to restart the scan. */
    }
    D1(jffs2_dump_block_lists(c));
    cond_resched();
} while(ret == -EAGAIN);
D1(printk(KERN_DEBUG "Pass 2 complete\n"));
```

最后第 3 阶段要释放 `jffs2_full_dirent` 数据结构了，它们在挂载文件系统时就建立就是为了统计各文件的硬

链接计数，此时其使命已经完成。

```
/* Finally, we can scan again and free the dirent nodes and scan_info structs */
for_each_inode(i, c, ic) {
    struct jffs2_full_dirent *fd;
    D1(printk(KERN_DEBUG "Pass 3: ino #%u, ic %p, nodes %p\n", ic->ino, ic, ic->nodes));

    while(ic->scan_dents) {
        fd = ic->scan_dents;
        ic->scan_dents = fd->next;
        jffs2_free_full_dirent(fd);
    }
    ic->scan_dents = NULL;
    cond_resched();
}
D1(printk(KERN_DEBUG "Pass 3 complete\n"));
D1(jffs2_dump_block_lists(c));

/* Rotate the lists by some number to ensure wear levelling */
jffs2_rotate_lists(c);
return ret;
}
```

先前在 `jffs2_scan_medium` 函数中为每个 `jffs2_raw_dirent` 目录项建立了临时的 `jffs2_full_dirent`，这里逐一删除，同时把每个目录文件的 `jffs2_inode_cache.scan_dents` 域设置为 `NULL`（其它文件的这个域本来就是 `NULL`），以标记数据实体内核描述符 `jffs2_raw_node_ref` 的 `next_in_ino` 域组成的链表的末尾。

（`jffs2_rotate_list` 的作用？及与 `wear leveling` 算法的关系如何？）

jffs2_scan_medium函数

这个函数遍历 flash 分区上的所有的擦除块，执行操作：

1. 读取每一个擦除块上的所有数据实体建立相应的内核描述符 `jffs2_raw_node_ref`；
2. 为每个文件建立内核描述符 `jffs2_inode_cache`，并建立相互连接关系；
3. 为目录文件的所有目录项创建相应的 `jffs2_full_dirent` 并组织为链表，由 `jffs2_inode_cache` 的 `scan_dents` 域指向；（注：这个目录树仅在 `jffs2_build_filesystem` 函数内部使用，在后面通过 `jffs2_build_inode_pass1` 函数计算完所有文件的硬链接个数 `nlink` 后，在 `jffs2_build_filesystem` 函数退出前就被删除了。）
4. 将所有文件的 `jffs2_inode_cache` 加入 `inocache_list` 哈希表；
5. 根据擦除块的使用情况将其描述符 `jffs2_eraseblock` 加入 `jffs2_sb_info` 中的 `xxxx_list` 链表。

```
int jffs2_scan_medium(struct jffs2_sb_info *c)
{
    int i, ret;
    uint32_t empty_blocks = 0, bad_blocks = 0;
    unsigned char *flashbuf = NULL;
```

```

uint32_t buf_size = 0;
size_t pointlen;

if (!c->blocks) {
    printk(KERN_WARNING "EEEE! c->blocks is NULL!\n");
    return -EINVAL;
}
if (c->mtd->point) {
    ret = c->mtd->point(c->mtd, 0, c->mtd->size, &pointlen, &flashbuf);
    if (!ret && pointlen < c->mtd->size) {
        /* Don't muck about if it won't let us point to the whole flash */
        D1(printk(KERN_DEBUG "MTD point returned len too short: 0x%x\n", pointlen));
        c->mtd->unpoint(c->mtd, flashbuf);
        flashbuf = NULL;
    }
    if (ret)
        D1(printk(KERN_DEBUG "MTD point failed %d\n", ret));
}

```

NOR flash 允许“就地运行”（XIP，即 eXecute_In_Place），比如在系统加电时引导程序的前端就是在 flash 上就地运行的。读 NOR flash 的操作与读 sdram 类似，而 flash 驱动中的读方法（read 或者 read_ecc）的本质操作为 memcpy，所以通过内存映射读取 flash 比通过其读方法要节约一次内存拷贝。

如果 NOR flash 驱动程序实现了 point 和 unpoint 方法，则允许建立内存映射。point 函数的第 2、3 个参数指定了被内存映射的区间，而实际被内存映射的区间长度由 pointlen 返回，起始虚拟地址存放在 ~~mtdbuf~~ ^{flashbuf} 所指变量中。这里试图内存映射整个 flash（传递的第 2、3 个参数为 0 和 mtd->size），如果实际被映射的长度 pointlen 小于 flash 大小，则用 unpoint 拆除内存映射。

```

if (!flashbuf) {
    /* For NAND it's quicker to read a whole eraseblock at a time, apparently */
    if (jffs2_cleanmarker_oob(c))
        buf_size = c->sector_size;
    else
        buf_size = PAGE_SIZE;
    D1(printk(KERN_DEBUG "Allocating readbuf of %d bytes\n", buf_size));
    flashbuf = kmalloc(buf_size, GFP_KERNEL);
    if (!flashbuf)
        return -ENOMEM;
}

```

如果 flashbuf 为空，即尚未建立 flash 的直接内存映射，那么需要额外分配一个内核缓冲区用于读出 flash 的内容。对于 NAND flash 根据作者的注释，一次性读出整个擦除块更快，所以缓冲区大小为擦除块大小；对于 NOR flash 该缓冲区大小等于一个内存页框大小。

（由此可见，如果 flash 驱动支持直接内存映射那么在读操作时就无需分配额外的缓冲区了。但是，根据 David

Woodhouse于 2005 年 3 月的文章www.linux-mtd.infradead.org/archive/tech/mtd_info.html, 函数point和unpoint的语义有待精确定义, 所以目前还是不要使用的好。

另外, 只有在读 NOR flash 时才可能会用 point 方法建立内存映射, 而此期间会持有锁而阻塞其它写操作, 所以在读操作完成后应该立即用 unpoint 拆除内存映射、释放锁。所以, 在这里就建立内存映射是否过早, 而应该推迟到真正执行读操作时? 比如在执行读操作时如果可以建立内存映射则通过它读取, 然后拆除之; 否则由 jffs2_flash_read 函数通过 mtd->read 方法读出)

```
for (i=0; i<c->nr_blocks; i++) {
    struct jffs2_eraseblock *jeb = &c->blocks[i];
    ret = jffs2_scan_eraseblock(c, jeb, buf_size?flashbuf:(flashbuf+jeb->offset), buf_size);
    if (ret < 0)
        return ret;
```

jffs2_scan_eraseblock 函数完成了 jff2_scan_medium 函数前 4 条工作, 详见后文。它根据擦除块的内的数据信息返回描述擦除块状态的数值。然后, 就得根据状态信息将擦除块描述符组织到 jffs2_sb_info 的不同的 xxxx_list 链表中去了。具体的工作在一个 switch 结构中完成:

```
ACCT_PARANOIA_CHECK(jeb);
/* Now decide which list to put it on */
switch(ret) {
case BLK_STATE_ALLFF: ALLFF就是all 0xff的意思?
    /* Empty block. Since we can't be sure it was entirely erased, we just queue it for erase
       again. It will be marked as such when the erase is complete. Meanwhile we still count it as
       empty for later checks. */
    empty_blocks++;
    list_add(&jeb->list, &c->erase_pending_list);
    c->nr_erasing_blocks++;
    break;
```

如果该擦除块上为全 1, 即没有任何信息, 当然也没有CLEANMARKER数据实体, 则将该擦除块描述符加入erase_pending_list链表, 同时增加相应的引用计数。该链表中的擦除块即将被擦除, 成功擦除后要在擦除块的开始写入CLEANMARKER (在jffs2_scan_eraseblock函数中返回该值的代码位置)。

```
case BLK_STATE_CLEANMARKER: /* Only a CLEANMARKER node is valid */
    if (!jeb->dirty_size) { /* It's actually free */
        list_add(&jeb->list, &c->free_list);
        c->nr_free_blocks++;
    } else { /* Dirt */
        D1(printk(KERN_DEBUG "Adding all-dirty block at 0x%08x to erase_pending_list\n",
                    jeb->offset));
        list_add(&jeb->list, &c->erase_pending_list);
        c->nr_erasing_blocks++;
    }
    break;
```

如果擦除块中只有一个 CLEANMARKER 数据实体是有效的，而且的确擦除块描述符中 dirty_size 也为 0，即擦除块中没有任何过时数据实体，则将其描述符加入 free_list 中，否则加入 erase_pending_list 中。

```
case BLK_STATE_CLEAN:          /* Full (or almost full) of clean data. Clean list */
    list_add(&jeb->list, &c->clean_list);
    break;
```

如果擦除块中基本上都是有效的数据，则将其加入 clean_list 链表。

```
case BLK_STATE_PARTDIRTY:      /* Some data, but not full. Dirty list. */
/*Except that we want to remember the block with most free space,
and stick it in the 'nextblock' position to start writing to it. Later when we do snapshots, this
must be the most recent block, not the one with most free space. */
    if (jeb->free_size > 2*sizeof(struct jffs2_raw_inode) &&
        (jffs2_can_mark_obsolete(c) || jeb->free_size > c->wbuf_pagesize) &&
        jeb是jffs2中erase block的意思(!c->nextblock || c->nextblock->free_size < jeb->free_size)) {
        /* Better candidate for the next writes to go to */
        if (c->nextblock) {
            c->nextblock->dirty_size += c->nextblock->free_size + c->nextblock->wasted_size;
            c->dirty_size += c->nextblock->free_size + c->nextblock->wasted_size;
            c->free_size -= c->nextblock->free_size;
            c->wasted_size -= c->nextblock->wasted_size;
            c->nextblock->free_size = c->nextblock->wasted_size = 0;
            if (VERYDIRTY(c, c->nextblock->dirty_size)) {
                list_add(&c->nextblock->list, &c->very_dirty_list);
            } else {
                list_add(&c->nextblock->list, &c->dirty_list);
            }
        }
        c->nextblock = jeb;
    } else {
        jeb->dirty_size += jeb->free_size + jeb->wasted_size;
        c->dirty_size += jeb->free_size + jeb->wasted_size;
        c->free_size -= jeb->free_size;
        c->wasted_size -= jeb->wasted_size;
        jeb->free_size = jeb->wasted_size = 0;
        if (VERYDIRTY(c, jeb->dirty_size)) {
            list_add(&jeb->list, &c->very_dirty_list);
        } else {
            list_add(&jeb->list, &c->dirty_list);
        }
    }
    break;
```

如果该擦除块含有至少一个过时的数据实体，那么就把它加入 `dirty_list` 或者 `very_dirty_list` 链表。宏 `VERYDIRTY` 定义如下：

```
#define VERYDIRTY(c, size)    ((size) >= ((c)->sector_size / 2))
```

即如果过时数据实体所占空间超过擦除块的一半大小，则认为该擦除块“很脏”。

`jffs2_sb_info` 的 `nextblock` 指向当前写入操作发生的擦除块。如果当前擦除块的剩余空间大小超过 `nextblock` 所指的擦除块，则将 `nextblock` 指向当前擦除块，而把原先的擦除块加入 `(very)dirty_list`。

（为什么要这样做？为什么在加入 `(very)dirty_list` 前要把擦除块的 `free_size` 和 `wasted_size` 的大小都记入 `dirty_size`？并且将二者清 0。）

```
case BLK_STATE_ALLDIRTY:
```

```
/* Nothing valid - not even a clean marker. Needs erasing. */
/* For now we just put it on the erasing list. We'll start the erases later */
D1(printk(KERN_NOTICE "JFFS2: Erase block at 0x%08x is not formatted.
                It will be erased\n", jeb->offset));
list_add(&jeb->list, &c->erase_pending_list);
c->nr_erasing_blocks++;
break;
```

如果这个擦除块中全都是过时的数据实体，设置没有 `CLEANMARKER`，那么将它加入 `erase_pending_list` 等待擦除。

```
case BLK_STATE_BADBLOCK:
```

```
D1(printk(KERN_NOTICE "JFFS2: Block at 0x%08x is bad\n", jeb->offset));
list_add(&jeb->list, &c->bad_list);
c->bad_size += c->sector_size;
c->free_size -= c->sector_size;
bad_blocks++;
break;
```

最后，如果这个擦除块已经损坏，那么将其加入 `bad_list` 链表，并增加 `bad_size` 计数，减小 `flash` 分区大小计数 `free_size`。

（如何判断擦除块已经损坏？）

```
default:
    printk(KERN_WARNING "jffs2_scan_medium(): unknown block state\n");
    BUG();
} //switch
} //for
```

至此，已经为所有擦除块上的所有数据实体和文件都建立了内核描述符，并且根据擦除块的使用情况将其

描述符加入了合适的 xxxx_list 链表。结束前还得做些额外工作：

```
/* Nextblock dirty is always seen as wasted, because we cannot recycle it now */
if (c->nextblock && (c->nextblock->dirty_size)) {
    c->nextblock->wasted_size += c->nextblock->dirty_size;
    c->wasted_size += c->nextblock->dirty_size;
    c->dirty_size -= c->nextblock->dirty_size;
    c->nextblock->dirty_size = 0;
}
```

jffs2_sb_info 的 nextblock 指向当前正在写入的擦除块（即被写入新的数据结点的擦除块。注意在 jffs2 上数据结点是顺序地写入 flash 的）。根据作者的注释，其过时的数据实体所占的空间被计算入被浪费的空间 wasted_size，而这又是因为当前擦除块没办法被 recycle。什么意思？

wasted_size 指一切不能被 GC 回收的空间
dirty_size 指一切可以被 GC 回收的空间

```
if (!jffs2_can_mark_obsolete(c) && c->nextblock && (c->nextblock->free_size & (c->wbuf_pagesize-1))) {
    /* If we're going to start writing into a block which already contains data, and the end of the data isn't
       page-aligned, skip a little and align it. */

    uint32_t skip = c->nextblock->free_size & (c->wbuf_pagesize-1);
    D1(printk(KERN_DEBUG "jffs2_scan_medium(): Skipping %d bytes in nextblock to
                          ensure page alignment\n", skip));
    c->nextblock->wasted_size += skip;
    c->wasted_size += skip;
    c->nextblock->free_size -= skip;
    c->free_size -= skip;
}
```

根据作者的注释，如果当前正在被写入的擦除块的可用空间的大小不是页地址对齐的，那么跳过其开头的部分空间，到达页地址对齐处。因此，无论擦除块描述符 jffs2_eraseblock 还是 jffs2_sb_info 中的 wasted_size 域都要相应地增加，free_size 域都要相应地减少。

另外需要说明的是，在当前开发板上使用的是 NOR flash，所以 CONFIG_JFFS2_FS_NAND 宏未定义，所以 jffs2_can_mark_obsolete(c) 被定义为 1，所以会跳过这段代码。

```
nr 是 number 的意思
if (c->nr_erasing_blocks) {
    if ( !c->used_size && ((empty_blocks+bad_blocks)!= c->nr_blocks || bad_blocks == c->nr_blocks) ) {
        printk(KERN_NOTICE "Cowardly refusing to erase blocks on filesystem with
                              no valid JFFS2 nodes\n");
        printk(KERN_NOTICE "empty_blocks %d, bad_blocks %d, c->nr_blocks %d\n",
                      empty_blocks, bad_blocks, c->nr_blocks);
        return -EIO;
    }
    jffs2_erase_pending_trigger(c);
}
if (buf_size)
```



```

    kfree(flashbuf);
else
    c->mtd->unpoint(c->mtd, flashbuf);
return 0;
}    buf_size是指用来从Flash中读取数据的缓冲区的大小

```

最后，在挂载完整文件系统后，如果有需要立即擦除的擦除块则通过 `jffs2_erase_pending_trigger` 函数设置文件系统超级块中的 `s_dirt` 标志，并且释放缓存擦除块内容的缓冲区。

jffs2_scan_eraseblock函数

该函数解析一个擦除块：

1. 为擦除块中所有数据实体建立相应的内核描述符 `jffs2_raw_node_ref`;
2. 为擦除块中的每个文件建立内核描述符 `jffs2_inode_cache`，并建立相互连接关系；
3. 为擦除块中的每个目录文件的所有目录项创建相应的 `jffs2_full_dirent` 并组织到 `jffs2_inode_cache` 的 `scan_dents` 链表中去；
4. 将擦除块中所有文件内核描述符加入 `inocache_list` 哈希表。

```

static int jffs2_scan_eraseblock (struct jffs2_sb_info *c, struct jffs2_eraseblock *jeb,
                                unsigned char *buf, uint32_t buf_size) {
    struct jffs2_unknown_node *node;
    struct jffs2_unknown_node crcnode;
    uint32_t ofs, prevofs;
    uint32_t hdr_crc, buf_ofs, buf_len;
    int err;
    int noise = 0;
    int wasempty = 0;
    uint32_t empty_start = 0;
#ifdef CONFIG_JFFS2_FS_NAND
    int cleanmarkerfound = 0;
#endif

    ofs = jeb->offset;
    prevofs = jeb->offset - 1;
    D1(printk(KERN_DEBUG "jffs2_scan_eraseblock(): Scanning block at 0x%x\n", ofs));
#ifdef CONFIG_JFFS2_FS_NAND
    if (jffs2_cleanmarker_oob(c)) {
        int ret = jffs2_check_nand_cleanmarker(c, jeb);
        D2(printk(KERN_NOTICE "jffs2_check_nand_cleanmarker returned %d\n", ret));
        /* Even if it's not found, we still scan to see if the block is empty. We use this information
           to decide whether to erase it or not. */
        switch (ret) {
            case 0:    cleanmarkerfound = 1; break;
            case 1:    break;

```



```

        case 2: return BLK_STATE_BADBLOCK;
        case 3: return BLK_STATE_ALLDIRTY; /* Block has failed to erase min. once */
        default: return ret;
    }
}
#endif
buf_ofs = jeb->offset;

```

设置 ofs 和 buf_ofs 为该擦除块在 flash 分区内的偏移。上面的代码与 NAND 类型的 flash 有关，在此略过。

```

if (!buf_size) {
    buf_len = c->sector_size;
} else {
    buf_len = EMPTY_SCAN_SIZE; //1024
    err = jffs2_fill_scan_buf(c, buf, buf_ofs, buf_len);
    if (err)
        return err;
}

```

通过 jffs2_fill_scan_buf 函数读取 flash 分区上偏移为 buf_ofs、长度为 buf_len 的数据到 buf 缓冲区中。这个函数就是直接调用 jffs2_flash_read 函数，而后者为直接调用 flash 驱动程序 read 方法的宏：

```

#define jffs2_flash_read(c, ofs, len, retlen, buf) ((c)->mtd->read((c)->mtd, ofs, len, retlen, buf))

```

对于 NAND flash 该缓冲区大小等于擦除块大小，所以这里就可以读出整个擦除块的内容；对于 NOR flash，缓冲区大小只等于一个页框，所以这里只能读出擦除块首部一个页面大小的内容，而在后文的 while 循环中逐页读出整个擦除块。

```

/* We temporarily use 'ofs' as a pointer into the buffer/jeb */
ofs = 0;
/* Scan only 4KiB of 0xFF before declaring it's empty */
while(ofs < EMPTY_SCAN_SIZE && *(uint32_t *)(&buf[ofs]) == 0xFFFFFFFF)
    ofs += 4;

```

前面 ofs 和 buf_ofs 都是一个擦除块在 flash 分区内的逻辑偏移，从此开始将 ofs 用作指向缓冲区 buf 内部的指针。如果 buf 中所有的数据都是 0xFF（注意 buf 的长度就是 EMPTY_SCAN_SIZE），则 ofs 到达 EMPTY_SCAN_SIZE 处，否则指向第一个非 1 字节。

```

if (ofs == EMPTY_SCAN_SIZE) {
#ifdef CONFIG_JFFS2_FS_NAND
    if (jffs2_cleanmarker_oob(c)) {
        /* scan oob, take care of cleanmarker */
        int ret = jffs2_check_oob_empty(c, jeb, cleanmarkerfound);
        D2(printk(KERN_NOTICE "jffs2_check_oob_empty returned %d\n", ret));
    }
#endif
}

```

```

        switch (ret) {
        case 0:  return cleanmarkerfound ? BLK_STATE_CLEANMARKER : BLK_STATE_ALLFF;
        case 1:  return BLK_STATE_ALLDIRTY;
        case 2:  return BLK_STATE_BADBLOCK; /* case 2/3 are paranoia checks */
        case 3:  return BLK_STATE_ALLDIRTY; /* Block has failed to erase min. once */
        default: return ret;
        }
    }
}

#endif

    D1(printk(KERN_DEBUG "Block at 0x%08x is empty (erased)\n", jeb->offset));
    return BLK_STATE_ALLFF; /* OK to erase if all blocks are like this */
}

```

如果ofs果然到达EMPTY_SCAN_SIZE处，则认为整个擦除块都是全 1（EMPTY_SCAN_SIZE只有 1k，而擦除块大小比如为 256k），所以返回BLK_STATE_ALLFF。（当从jffs2_scan_eraseblock函数返回到jff2_scan_medium后，[在jff2_scan_medium中根据返回值BLK_STATE_ALLFF将当前擦除块加入jffs2_sb_info.erase_pending_list链表](#)）

```

    if (ofs) {
        D1(printk(KERN_DEBUG "Free space at %08x ends at %08x\n", jeb->offset,
            jeb->offset + ofs));
        DIRTY_SPACE(ofs);
    }
}

```

如果 ofs 没有到达 EMPTY_SCAN_SIZE 处，则指向 buf 中第一个非 1 字节。那么该擦除块开头这 ofs 长度的空间将无法被利用，所以用 DIRTY_SIZE 宏修改 jffs2_eraseblock 和 jffs2_sb_info 的 free_size 和 dirty_size：

```

#define DIRTY_SPACE(x) do { typeof(x) _x = (x); \
    c->free_size -= _x; c->dirty_size += _x; \
    jeb->free_size -= _x ; jeb->dirty_size += _x; \
} while(0)

```

注意从这里开始统计擦除块的使用情况，设置 jffs2_eraseblock 和 jffs2_sb_info 中的 xxxx_size 域。

```

/* Now ofs is a complete physical flash offset as it always was... */
ofs += jeb->offset;

```

原来 ofs 指该 buf 内部的相对地址，而擦除块块的在 flash 分区的逻辑偏移为 jeb->offset，所以加上后者后 ofs 就是在 flash 分区内的逻辑偏移了。下面从已经读出的 buf_len 个数据开始遍历整个擦除块。虽然读出的数据量 buf_len 可能小于擦除块大小 sector_size，但是从下文可知，如果在 buf 的末尾含有一个数据实体的部分数据，则会接着读出 flash 分区中 ofs 开始、长度为 buf_len 的后继数据（ofs 即指向该数据实体的起始偏移）。

```

noise = 10;

```

```

while(ofs < jeb->offset + c->sector_size) {
    D1(ACCT_PARANOIA_CHECK(jeb));
    cond_resched();
}

```

ofs 为当前擦除块的某个字节在 flash 分区内的逻辑偏移，而 “jeb->offset + c->sector_size” 为当前擦除块在分区内的后继字节地址，即在一个循环中遍历整个擦除块。

新一轮循环开始时，ofs 所指字节单元可能包括各种情况。在循环的开始首先处理一些特殊状况：

```

if (ofs & 3) {
    printk(KERN_WARNING "Eep. ofs 0x%08x not word-aligned!\n", ofs);
    ofs = (ofs+3)&~3;
    continue;
}

```

flash 上的数据实体都要求是 4 字节地址对齐的，所以如果如果没有地址对齐，则步进 ofs 到地址对齐处并开始新的循环。

```

if (ofs == prevofs) {
    printk(KERN_WARNING "ofs 0x%08x has already been seen. Skipping\n", ofs);
    DIRTY_SPACE(4);
    ofs += 4;
    continue;
}
prevofs = ofs;

```

循环开始前，prevofs 被设置为 “jeb->offset - 1”，即当前擦除块的前驱字节地址，所以对于第一个非 1 数据字节的位置 ofs 一定不会等于 prevofs。而以后 prevofs 就用于记录已经遍历过的地址。如果当前地址 ofs 已经遍历过，则跳过 4 个字节。

```

if (jeb->offset + c->sector_size < ofs + sizeof(*node)) {
    D1(printk(KERN_DEBUG "Fewer than %d bytes left to end of block. (%x+%x<%x+%x)
        Not reading\n", sizeof(struct jffs2_unknown_node),
        jeb->offset, c->sector_size, ofs, sizeof(*node)));
    DIRTY_SPACE((jeb->offset + c->sector_size)-ofs);
    break;
}

```

再次重申，ofs 为当前擦除块内非 1 字节在 flash 分区中的逻辑偏移，而 sizeof(*node) 为所有的数据实体的头部大小（4 字节）。如果在当前擦除块的末尾无法容纳一个数据实体的头部信息，那么从 ofs 开始到当前擦除块末尾的空间（“jeb->offset + c->sector_size - ofs”）将不会被利用，所以应该计算为 “dirty”。另外，这种情况也代表这当前擦除块遍历完毕，所以通过 break 跳出循环。

```

if (buf_ofs + buf_len < ofs + sizeof(*node)) {

```

```

    buf_len = min_t(uint32_t, buf_size, jeb->offset + c->sector_size - ofs);
    D1(printk(KERN_DEBUG "Fewer than %d bytes (node header) left to end of buf. Reading 0x%x
        at 0x%08x\n", sizeof(struct jffs2_unknown_node), buf_len, ofs));
    err = jffs2_fill_scan_buf(c, buf, ofs, buf_len);
    if (err)
        return err;
    buf_ofs = ofs;
}

```

当从擦除块中读出第一个块长度为 `buf_len` 的数据时，`buf_ofs` 为擦除块的分区偏移。如果先前读出的数据量的末尾没有包含一个完整的数据实体的头部，则从 `ofs` 开始，即该数据实体头部开始，再读出长度 `buf_len` 的数据，以便下面用 `node` 数据结构取出数据实体的头部：

```
node = (struct jffs2_unknown_node *)&buf[ofs-buf_ofs];
```

假设从 `ofs` 开始发现了一个 `jffs2_raw_inode/dirent`，那么应该到达下面发现JFFS2_MAGIC_BITMASK的情况。我们先跳过下面这部分代码。

```

if (*(uint32_t *)&buf[ofs-buf_ofs] == 0xffffffff) {
    uint32_t inbuf_ofs = ofs - buf_ofs + 4;
    uint32_t scanend;

    empty_start = ofs;
    ofs += 4;
    /* If scanning empty space after only a cleanmarker, don't bother scanning the whole block */
    if (unlikely(empty_start == jeb->offset + c->cleanmarker_size &&
        jeb->offset + EMPTY_SCAN_SIZE < buf_ofs + buf_len))
        scanend = jeb->offset + EMPTY_SCAN_SIZE - buf_ofs;
    else
        scanend = buf_len;

    D1(printk(KERN_DEBUG "Found empty flash at 0x%08x\n", ofs));
    while (inbuf_ofs < scanend) {
        if (*(uint32_t *)&buf[inbuf_ofs] != 0xffffffff)
            goto emptyends;
        inbuf_ofs+=4;
        ofs += 4;
    }
    /* Ran off end. */
    D1(printk(KERN_DEBUG "Empty flash ends normally at 0x%08x\n", ofs));
    if (buf_ofs == jeb->offset && jeb->used_size == PAD(c->cleanmarker_size) &&
        !jeb->first_node->next_in_ino && !jeb->dirty_size)
        return BLK_STATE_CLEANMARKER;
    wasempty = 1;
}

```

```

        continue;
    } else if (wasempty) {
        emptyends:
        //printk(KERN_WARNING "Empty flash at 0x%08x ends at 0x%08x\n", empty_start, ofs);
        DIRTY_SPACE(ofs-empty_start);
        wasempty = 0;
        continue;
    }

    if (ofs == jeb->offset && je16_to_cpu(node->magic) == KSAMTIB_CIGAM_2SFFJ) {
        printk(KERN_WARNING "Magic bitmask is backwards at offset 0x%08x.
                               Wrong endian filesystem?\n", ofs);

        DIRTY_SPACE(4);
        ofs += 4;
        continue;
    }

```

如果数据实体的头部 `node` 的第一个字节（魔数）为 `KSAMTIB_CIGAM_2SFFJ`，则说明 `jffs2` 文件系统中数据实体的字节序错误。正常情况下应该等于 `JFFS2_MAGIC_BITMASK`。它们的定义为：

```

/* Values we may expect to find in the 'magic' field */
#define JFFS2_OLD_MAGIC_BITMASK 0x1984
#define JFFS2_MAGIC_BITMASK 0x1985
#define KSAMTIB_CIGAM_2SFFJ 0x5981 /* For detecting wrong-endian fs */
#define JFFS2_EMPTY_BITMASK 0xffff
#define JFFS2_DIRTY_BITMASK 0x0000

    if (je16_to_cpu(node->magic) == JFFS2_DIRTY_BITMASK) {
        D1(printk(KERN_DEBUG "Empty bitmask at 0x%08x\n", ofs));
        DIRTY_SPACE(4);
        ofs += 4;
        continue;
    }

```

如果头部的魔数为 0，则认为无效的数据头部，所以跳过 4 字节。

```

    if (je16_to_cpu(node->magic) == JFFS2_OLD_MAGIC_BITMASK) {
        //printk(KERN_WARNING "Old JFFS2 bitmask found at 0x%08x\n", ofs);
        //printk(KERN_WARNING "You cannot use older JFFS2 filesystems with newer kernels\n");
        DIRTY_SPACE(4);
        ofs += 4;
        continue;
    }

```

如果 jffs 文件系统映象是用版本 1 的工具生成的，那么显然不能与版本 2 的代码工作在一起，所以要跳过整个数据实体头部。

上面已经考虑了魔数的各种其它情况，代码执行到这里魔数就应该为 JFFS2_MAGIC_BITMASK 了。当然，如果还不是，就只能跳过 4 字节开始新的循环。

这种情况也是很有可能发生的，比如当数据实体头部的 crc 校验错误时，从下面可用看到只是简单的步进 ofs 四个字节，在新的循环中又会执行到 “node = (struct jffs2_unknown_node *)&buf[ofs-buf_ofs];” 显然，从 ofs 开始的应该是错误 crc 头部的后继数据，那么就会进入这个 if 分支中。而且这种情况会重复发生，直到整个数据实体都被遍历完。但是，由于 flash 上数据实体的长度也是 4 字节地址对齐的，所以不会影响后继的数据实体！

（这也是我觉得当发现错误的 crc 头部时应该跳过整个数据实体的原因）

```
if (je16_to_cpu(node->magic) != JFFS2_MAGIC_BITMASK) {
    /* OK. We're out of possibilities. Whinge and move on */
    //noisy_printk(&noise, "jffs2_scan_eraseblock(): Magic bitmask 0x%04x not found at 0x%08x:
        0x%04x instead\n", JFFS2_MAGIC_BITMASK, ofs, je16_to_cpu(node->magic));
    DIRTY_SPACE(4);
    ofs += 4;
    continue;
}

/* We seem to have a node of sorts. Check the CRC */
crcnode.magic = node->magic;
crcnode.nodetype = cpu_to_je16( je16_to_cpu(node->nodetype) | JFFS2_NODE_ACCURATE);
crcnode.totlen = node->totlen;
hdr_crc = crc32(0, &crcnode, sizeof(crcnode)-4);
if (hdr_crc != je32_to_cpu(node->hdr_crc)) {
    //noisy_printk(&noise, "jffs2_scan_eraseblock(): Node at 0x%08x {0x%04x, 0x%04x, 0x%08x}
        has invalid CRC 0x%08x (calculated 0x%08x)\n",
    //    ofs, je16_to_cpu(node->magic),
    //    je16_to_cpu(node->nodetype),
    //    je32_to_cpu(node->totlen),
    //    je32_to_cpu(node->hdr_crc),
    //    hdr_crc);
    DIRTY_SPACE(4);
    ofs += 4;
    continue;
}
```

暂时不知道je16_to_cpu或je32_to_cpu的作用，难道还要像计算机网络那样转换数据大小端？

计算数据实体头部的 crc 值，并与其声称的 crc 值向比较。正常情况下应该相同，否则就跳过 4 个字节并开始新的循环。

我觉得应该跳过整个数据实体！（整个数据实体的长度为 node->totlen）

我不觉得，因为node头部中数据都错了，你怎么敢保证node->totlen就是正确的？

```
if (ofs + je32_to_cpu(node->totlen) > jeb->offset + c->sector_size) {
```

```

/* Eep. Node goes over the end of the erase block. */
printk(KERN_WARNING "Node at 0x%08x with length 0x%08x would run over the end of
                    the erase block\n", ofs, je32_to_cpu(node->totlen));
printk(KERN_WARNING "Perhaps the file system was created with the wrong erase size?\n");
DIRTY_SPACE(4);
ofs += 4;
continue;
}

```

jffs2 要求任何数据实体不能跨越一个擦除块。如果这种情况发生了，则跳过 4 字节并开始新的循环。

```

if (!(je16_to_cpu(node->nodetype) & JFFS2_NODE_ACCURATE)) {
    /* Wheee. This is an obsoleted node */
    D2(printk(KERN_DEBUG "Node at 0x%08x is obsolete. Skipping\n", ofs));
    DIRTY_SPACE(PAD(je32_to_cpu(node->totlen)));
    ofs += PAD(je32_to_cpu(node->totlen));
    continue;          PAD是使一个数与四对齐
}

```

因为都或上这个ACCURATE

根据 linux/jffs2.h，所有四种有效的数据结点类型中 JFFS2_NODE_ACCURATE 都有效，否则要跳过整个数据实体，注意头部中 totlen 为包括了后继数据的数据实体总长度。jffs2 文件系统的 flash 中的数据实体有如下 4 种类型：

```

#define JFFS2_NODETYPE_DIRENT      (JFFS2_FEATURE_INCOMPAT |
                                     JFFS2_NODE_ACCURATE | 1)
#define JFFS2_NODETYPE_INODE      (JFFS2_FEATURE_INCOMPAT |
                                     JFFS2_NODE_ACCURATE | 2)
#define JFFS2_NODETYPE_CLEANMARKER (JFFS2_FEATURE_RWCOMPAT_DELETE |
                                     JFFS2_NODE_ACCURATE | 3)
#define JFFS2_NODETYPE_PADDING    (JFFS2_FEATURE_RWCOMPAT_DELETE |
                                     JFFS2_NODE_ACCURATE | 4)

```

下面就得根据头部中的 nodetype 字段判断数据实体的类型。注意，数据实体的头部是从 flash 分区中偏移 ofs 开始的：

```


switch(je16_to_cpu(node->nodetype)) {
case JFFS2_NODETYPE_INODE:
    if (buf_ofs + buf_len < ofs + sizeof(struct jffs2_raw_inode)) {
        buf_len = min_t(uint32_t, buf_size, jeb->offset + c->sector_size - ofs);
        D1(printk(KERN_DEBUG "Fewer than %d bytes (inode node) left to end of buf. Reading
                            0x%x at 0x%08x\n", sizeof(struct jffs2_raw_inode), buf_len, ofs));
        err = jffs2_fill_scan_buf(c, buf, ofs, buf_len);
        if (err)
            return err;
        buf_ofs = ofs;
    }
}

```

```

    node = (void *)buf;
}
err = jffs2_scan_inode_node(c, jeb, (void *)node, ofs);
if (err) return err;
ofs += PAD(je32_to_cpu(node->totlen));
break; //跳出 switch, 开始新一轮大循环

```



首先, 如果数据实体为 `jffs2_raw_inode`, 则由 `jffs2_scan_inode_node` 函数为之创建相应的内核描述符 `jffs2_raw_node_ref`。如果该数据实体为某文件的第一个数据实体, 则该文件的内核描述符 `jffs2_inode_cache` 尚未创建, 则创建之并加入 `inocache_list` 哈希表中, 然后建立数据实体描述符和文件描述符之间的连接关系。参见[下文](#)。

注意, 先前读出的“flash 分区中偏移为 `buf_ofs`、长度为 `buf_len`”的空间的末尾可能只含有部分数据实体, 那么还得继续读出后继的数据块。注意, 读出的后继数据块长度不会小于“`jeb->offset + c->sector_size - ofs`”, 而 `jffs2` 要求一个数据实体不会跨越擦除块边界, 所以一定能读出至少一个完整的数据实体。

扫描完该 `jffs2_raw_inode` 后, 就要递增 `ofs` 为其长度, 用 `break` 跳出 `switch` 结构、开始新一轮循环了。注意 flash 上的数据实体不但起始地址是 4 字节地址对齐的, 而且长度也是 4 字节地址对齐的。递增 `ofs` 前还要由 `PAD` 宏将数据长度向上取整为 4 字节对齐的。

case JFFS2_NODETYPE_DIRENT:

```

if (buf_ofs + buf_len < ofs + je32_to_cpu(node->totlen)) {
    buf_len = min_t(uint32_t, buf_size, jeb->offset + c->sector_size - ofs);
    D1(printk(KERN_DEBUG "Fewer than %d bytes (dirent node) left to end of buf. Reading
        0x%x at 0x%08x\n", je32_to_cpu(node->totlen), buf_len, ofs));
    err = jffs2_fill_scan_buf(c, buf, ofs, buf_len);
    if (err)
        return err;
    buf_ofs = ofs;
    node = (void *)buf;
}
err = jffs2_scan_dirent_node(c, jeb, (void *)node, ofs);
if (err) return err;
ofs += PAD(je32_to_cpu(node->totlen));
break; //跳出 switch, 开始新一轮大循环

```

如果数据实体为 `jffs2_raw_dirent` 目录项, 则由 `jffs2_scan_dirent_node` 函数分析之。详见[下文](#)。其余注意事项与前同。

case JFFS2_NODETYPE_CLEANMARKER:

```

D1(printk(KERN_DEBUG "CLEANMARKER node found at 0x%08x\n", ofs));
if (je32_to_cpu(node->totlen) != c->cleanmarker_size) {
    printk(KERN_NOTICE "CLEANMARKER node found at 0x%08x has totlen 0x%x !=
        normal 0x%x\n", ofs, je32_to_cpu(node->totlen), c->cleanmarker_size);
}

```



```

        DIRTY_SPACE(PAD(sizeof(struct jffs2_unknown_node)));
        ofs += PAD(sizeof(struct jffs2_unknown_node));
    } else if (jeb->first_node) {
        printk(KERN_NOTICE "CLEANMARKER node found at 0x%08x, not first node in block
                           (0x%08x)\n", ofs, jeb->offset);
        DIRTY_SPACE(PAD(sizeof(struct jffs2_unknown_node)));
        ofs += PAD(sizeof(struct jffs2_unknown_node));
    } else {
        struct jffs2_raw_node_ref *marker_ref = jffs2_alloc_raw_node_ref();
        if (!marker_ref) {
            printk(KERN_NOTICE "Failed to allocate node ref for clean marker\n");
            return -ENOMEM;
        }
        marker_ref->next_in_ino = NULL;
        marker_ref->next_phys = NULL;
        marker_ref->flash_offset = ofs | REF_NORMAL;
        marker_ref->totlen = c->cleanmarker_size;
        jeb->first_node = jeb->last_node = marker_ref;
        USED_SPACE(PAD(c->cleanmarker_size));
        ofs += PAD(c->cleanmarker_size);
    }
    break;

```

如果数据实体为 CLEANMARKER，即为一个 `jffs2_unknown_node` 数据结构，首先检查其长度，如果不符合则跳过。另外，CLEANMARKER 是该擦除块成功擦除后写入的第一个数据实体，所以当扫描出它时擦除块描述符的 `first_node` 指针应该为空，否则跳过。

如果一切正常，则为 CLEANMARKER 分配相应的内核描述符并插入擦除块 `first_node` 所指向的链表中去。

```

case JFFS2_NODETYPE_PADDING:
    DIRTY_SPACE(PAD(je32_to_cpu(node->totlen)));
    ofs += PAD(je32_to_cpu(node->totlen));
    break;

```

如果数据实体为填充块，则跳过即可。

正常情况下 flash 上只有上述 4 中数据实体。对于其它特殊数据实体，则需要另外处理（参见 jffs2 作者描述 jffs2 的论文）：

```

default://? ?
    switch (je16_to_cpu(node->nodetype) & JFFS2_COMPAT_MASK) {
    case JFFS2_FEATURE_ROCOMPAT:
        printk(KERN_NOTICE "Read-only compatible feature node (0x%04x) found at offset
                           0x%08x\n", je16_to_cpu(node->nodetype), ofs);

```

```

        c->flags |= JFFS2_SB_FLAG_RO;
    if (!jffs2_is_readonly(c))
        return -EROFS;
    DIRTY_SPACE(PAD(je32_to_cpu(node->totlen)));
    ofs += PAD(je32_to_cpu(node->totlen));
    break;

```

如果发现这种类型的数据实体，那么整个 jffs2 文件系统都只能按照只读的方式挂载，所以设置文件系统超级块的 u 域即 jffs2_sb_info 的 flags 域的 JFFS2_SB_FLAG_RO 标志，同时还得检查挂载 jffs2 时的方式，如果不是按照只读方式挂载的，则返回错误 EROFS（错误值将逆着函数调用链一直向上游传递，从而导致挂载失败）。

```

case JFFS2_FEATURE_INCOMPAT:
    printk(KERN_NOTICE "Incompatible feature node (0x%04x) found at offset 0x%08x\n",
           je16_to_cpu(node->nodetype), ofs);
    return -EINVAL;

```

如果遇到这种类型的数据实体，则直接拒绝挂载 jffs2。

```

case JFFS2_FEATURE_RWCOMPAT_DELETE:
    D1(printk(KERN_NOTICE "Unknown but compatible feature node (0x%04x) found at offset
           0x%08x\n", je16_to_cpu(node->nodetype), ofs));
    DIRTY_SPACE(PAD(je32_to_cpu(node->totlen)));
    ofs += PAD(je32_to_cpu(node->totlen));
    break;
case JFFS2_FEATURE_RWCOMPAT_COPY:
    D1(printk(KERN_NOTICE "Unknown but compatible feature node (0x%04x) found at offset
           0x%08x\n", je16_to_cpu(node->nodetype), ofs));
    USED_SPACE(PAD(je32_to_cpu(node->totlen)));
    ofs += PAD(je32_to_cpu(node->totlen));
    break;

```

如果遇到这两种数据实体，则直接跳过即可。

```

        }//switch
    }//default
}
D1(printk(KERN_DEBUG "Block at 0x%08x: free 0x%08x, dirty 0x%08x, used 0x%08x\n", jeb->offset,
           jeb->free_size, jeb->dirty_size, jeb->used_size));

```

至此，已经遍历完整个擦除块的内容，函数的最后就是要根据擦除块描述符中的统计信息，返回反应其状态的数值了：

```

/* mark_node_obsolete can add to wasted !! */

```

```

if (jeb->wasted_size) {
    jeb->dirty_size += jeb->wasted_size;
    c->dirty_size += jeb->wasted_size;
    c->wasted_size -= jeb->wasted_size;
    jeb->wasted_size = 0;
}

```

如果当前擦除块的 `wasted_size` 域不为 0，则将其算入 `dirty_size`。同时刷新相关统计信息。为什么要这样？

```

if ((jeb->used_size + jeb->unchecked_size) == PAD(c->cleanmarker_size) &&
    !jeb->first_node->next_in_ino && !jeb->dirty_size)
    return BLK_STATE_CLEANMARKER;

```

第一个条件满足表示擦除块中有效数据实体空间和未检查空间等于一个 CLEANMARKER 大小，即只有一个 CLEANMARKER，此时它的内核描述符中 `next_in_ino` 指针一定为 NULL（因为 CLEANMARKER 不属于任何文件），所以第二个条件也满足；第三个条件再次检查的确没有任何过时数据实体，即 `dirty_size` 为 0，则返回 `BLK_STATE_CLEANMARKER`。

```

/* move blocks with max 4 byte dirty space to cleanlist */
else if (!ISDIRTY(c->sector_size - (jeb->used_size + jeb->unchecked_size))) {
    c->dirty_size -= jeb->dirty_size;
    c->wasted_size += jeb->dirty_size;
    jeb->wasted_size += jeb->dirty_size;
    jeb->dirty_size = 0;
    return BLK_STATE_CLEAN;
}

```

参见下面的 `jffs2_scan_inode/dirent_node` 函数分析，它们在扫描完一个有效的 `jffs2_raw_inode` 或 `jffs2_raw_dirent` 数据实体后，分别用 `UNCHECKED_SPACE` 和 `USED_SPACE` 宏增加相应的统计信息。那么遍历完整个擦除块后，擦除块描述符的 `used_size + unchecked_size` 即为其上所有有效数据实体和未检查空间的大小。所以用擦除块大小 `sector_size` 减去这个值就是 `dirty` 和 `wasted` 的空间的大小。`ISDIRTY` 宏用于判断为脏的区域大小 `size` 是否超过 255：

```

/* check if dirty space is more than 255 Byte */
#define ISDIRTY(size) ((size) > sizeof(struct jffs2_raw_inode) + JFFS2_MIN_DATA_LEN)

```

所以，如果 `(c->sector_size - (jeb->used_size + jeb->unchecked_size))` 小于 255，则还认为这个擦除块还是干净的，所以返回 `BLK_STATE_CLEAN`。

有效的数据太多了就认为是CLEAN的

```

else if (jeb->used_size || jeb->unchecked_size)
    return BLK_STATE_PARTDIRTY;
else
    return BLK_STATE_ALLDIRTY;
}

```

否则擦除块为脏。used_size 为有效数据实体的总长度。如果任意不为 0，即擦除块中含有至少一个有效数据实体，则返回“部分脏”数值 BLK_STATE_PARTDIRTY，否则一个有效数据实体也没有，自然应该返回“全脏”数值 BLK_STATE_ALLDIRTY。

jffs2_scan_inode_node函数

该函数为数据实体创建内核描述符 jffs2_raw_node_ref，如果所属文件的内核描述符 jffs2_inode_cache 不存在，则创建之，并建立二者连接关系，再将 jffs2_inode_cache 记录到 inocache_list 哈希表中。参数 ri 为已经读出到内核缓冲区中的 jffs2_raw_inode 数据实体的首址，该数据实体在 flash 分区内的偏移为 ofs。从函数中可用看到数据实体内核描述符 jffs2_raw_node_ref 的 flash_offset 域的值就由 ofs 设置。

```
static int jffs2_scan_inode_node(struct jffs2_sb_info *c, struct jffs2_eraseblock *jeb,
                                struct jffs2_raw_inode *ri, uint32_t ofs)
{
    struct jffs2_raw_node_ref *raw;
    struct jffs2_inode_cache *ic;
    uint32_t ino = je32_to_cpu(ri->ino);
```

首先，从数据实体 jffs2_raw_inode 的 ino 字段获得其所属文件的索引结点数。

```
D1(printk(KERN_DEBUG "jffs2_scan_inode_node(): Node at 0x%08x\n", ofs));
/* We do very little here now. Just check the ino# to which we should attribute
   this node; we can do all the CRC checking etc. later. There's a tradeoff here --
   we used to scan the flash once only, reading everything we want from it into
   memory, then building all our in-core data structures and freeing the extra
   information. Now we allow the first part of the mount to complete a lot quicker,
   but we have to go _back_ to the flash in order to finish the CRC checking, etc.
   Which means that the _full_ amount of time to get to proper write mode with GC
   operational may actually be _longer_ than before. Sucks to be me. */
```

```
raw = jffs2_alloc_raw_node_ref();
if (!raw) {
    printk(KERN_NOTICE "jffs2_scan_inode_node(): allocation of node reference failed\n");
    return -ENOMEM;
}
```

然后，通过 kmem_cache_alloc 函数从 raw_node_ref_slab 中分配一个 jffs2_raw_node_ref 数据结构。[回想在注册 jffs2 文件系统时就已经为所有的内核描述符数据结构创建了相应的高速缓存。](#) [进阶](#)

```
ic = jffs2_get_ino_cache(c, ino);
if (!ic) {
    /* Inocache get failed. Either we read a bogus ino# or it's just genuinely the
       first node we found for this inode. Do a CRC check to protect against the former case */
```

```

uint32_t crc = crc32(0, ri, sizeof(*ri)-8);
if (crc != je32_to_cpu(ri->node_crc)) { 再检查下CRC
    printk(KERN_NOTICE "jffs2_scan_inode_node(): CRC failed on node at 0x%08x: Read 0x%08x,
        calculated 0x%08x\n", ofs, je32_to_cpu(ri->node_crc), crc);
    /* We believe totlen because the CRC on the node _header_ was OK, just the node itself failed. */
    DIRTY_SPACE(PAD(je32_to_cpu(ri->totlen)));
    return 0;
}
ic = jffs2_scan_make_ino_cache(c, ino);
if (!ic) {
    jffs2_free_raw_node_ref(raw);
    return -ENOMEM;
}
}

```

函数的开始就已经得到了数据实体所属文件的索引结点号，这里通过 `jffs2_get_ino_cache` 函数返回相应文件的、组织在 `inocache_list` 哈希表中的 `jffs2_inode_cache` 数据结构。如果不存在则返回 `NULL`，这说明我们遇到了属于该文件的第一个数据实体，所以还得通过 `jffs2_scan_make_ino_cache` 函数为该文件分配一个 `jffs2_inode_cache` 数据结构并加入哈希表。详见下文分析。

这里需要着重说明的是，对于 `jffs2_raw_inode` 数据实体，其 `ino` 即为其所属文件的索引结点号，所以在调用 `jffs2_scan_make_ino_cache` 函数返回其文件的内核描述符时自然传递 `ino`。但是对于 `jffs2_raw_dirent` 目录项数据实体，`ino` 表示其代表的文件的索引结点号，而该目录项所属的目录文件的索引结点号则由 `pino` 指明。目录项的内核描述符自然应该加入其所属目录文件的内核描述符的 `nodes` 链表中去。由下文 `jffs2_scan_dirent_node` 函数可见，其中调用 `jffs2_scan_make_ino_cache` 函数时传递的是 `pino`。

另外，文件的 `jffs2_inode_cache` 数据结构不存在也有可能是由于前面得到的索引结点号 `ino` 错误引起的。所以在创建新的 `jffs2_inode_cache` 数据结构前，首先得验证 `crc` 以保证 `ino` 号是正确的。`jffs2_raw_inode` 数据结构的最后两个域各为长度为 4 字节的 `crc` 校验值 `data_crc` 和 `node_crc`，分别为其后压缩了的数据的 `crc` 校验值和 `jffs2_raw_inode` 数据结构本身的 `crc` 校验值。在计算 `jffs2_raw_inode` 数据结构本身的 `crc` 校验值时要排除这两个域共 8 字节，然后再与读出的 `node_crc` 相比较。

如果发生 `crc` 校验错误，则增加 `jffs2_sb_info` 和 `jffs2_eraseblock` 中的 `dirty_size` 值、减小 `free_size` 值，并直接退出。
一开始的 `free_size` 是整个 `sector_size`，因此要先减小

```

/* Wheee. It worked */
raw->flash_offset = ofs | REF_UNCHECKED;
raw->totlen = PAD(je32_to_cpu(ri->totlen));
raw->next_phys = NULL;

```

如果一切顺利，则初始化 `jffs2_raw_node_ref`，首先将 `flash_offset` 设置为数据实体在 `flash` 分区内的逻辑偏移。该函数的参数 `ofs` 正是这个值。由于 `flash` 上数据结点的总是 4 字节地址对齐的，所以 `jffs2_raw_node_ref` 的 `flash_offset` 的最低两个 bit 总是 0，所以可用利用它们标记相应数据实体的状态。这两位可能的值定义如下：

```

#define REF_UNCHECKED    0        /* We haven't yet checked the CRC or built its inode */
#define REF_OBSOLETE     1        /* Obsolete, can be completely ignored */
#define REF_PRISTINE     2        /* Completely clean. GC without looking */
#define REF_NORMAL       3        /* Possibly overlapped. Read the page and write again on GC */

```

而 PAD 宏定义为: `#define PAD(x) (((x)+3)&~3)`。由此可见, flash 上数据实体的长度也是 4 字节地址对齐的。`next_phys` 用于把擦除块内所有数据实体描述符组织为一个链表, 这里先初始化为 `NULL`, 在下面才加入链表。

尤其需要说明的是, 根据作者的注释这里在挂载文件系统时并没有检查 flash 上数据实体的有效性, 而把 crc 校验的工作推迟到了打开文件、为文件创建 `jffs2_full_dnode` 或者 `jffs2_full_dirent` 时进行 (详见后文“打开文件时读 inode 的方法”一章中的 `jffs2_get_inode_nodes` 函数的[相关部分](#))。所以需要在数据实体的内核描述符中设置 `REF_UNCHECKED` 标志。这样做的目的是加快文件系统的挂载过程, 但是“跑得了和尚跑不了庙”, 在打开文件时还得进行 crc 校验。但是以后并不是一定需要打开所有的文件, 所以推迟 crc 校验还是有好处的。

另外在下文 `jffs2_scan_dirent_node` 函数分析中可以看到, 对目录项数据实体即刻进行了 crc 校验, 所以将其内核描述符设置为 `REF_PRISTINE` (参见 `jffs2_scan_dirent_node` 函数的[相关部分](#))。

然后, 就得建立数据实体内核描述符 `jffs2_raw_node_ref` 和其所属文件描述符 `jffs2_inode_cache` 之间的联系了:

```

raw->next_in_ino = ic->nodes;
ic->nodes = raw;

```

即将该 `jffs2_raw_node_ref` 加入到 `jffs2_inode_cache.nodes` 链表的首部。

```

if (!jeb->first_node)
    jeb->first_node = raw;
if (jeb->last_node)
    jeb->last_node->next_phys = raw;
jeb->last_node = raw;

```

擦除块内所有数据实体的内核描述符通过 `next_phys` 域组织成一个链表, 链表的首尾元素由 `jffs2_eraseblock` 数据结构的 `last_node` 和 `first_node` 指向。

```

D1(printk(KERN_DEBUG "Node is ino %#u, version %d. Range 0x%x-0x%x\n",
    je32_to_cpu(ri->ino), je32_to_cpu(ri->version), je32_to_cpu(ri->offset),
    je32_to_cpu(ri->offset)+je32_to_cpu(ri->dsiz)));
pseudo_random += je32_to_cpu(ri->version);
UNCHECKED_SPACE(PAD(je32_to_cpu(ri->totlen)));
return 0;
}

```

最后,由 `UNCHECKED_SPACE` 宏减少 `jffs2_sb_info` 和 `jffs2_eraseblock` 中的 `free_size` 域,增加 `unchecked_size` 域:

```
#define UNCHECKED_SPACE(x) do { typeof(x) _x = (x); \
    c->free_size -= _x; c->unchecked_size += _x; \
    jeb->free_size -= _x ; jeb->unchecked_size += _x; \
} while(0)
```

`jffs2_scan_inode_node`函数完成后,数据实体和文件的内核描述符之间的关系参见图 2 (为图 2 的一部分)。

jffs2_scan_make_ino_cache函数

这个函数为索引结点号为 `ino` 的文件分配一个新的 `jffs2_inode_cache` 数据结构并加入文件系统 `hash` 表:

```
static struct jffs2_inode_cache *jffs2_scan_make_ino_cache(struct jffs2_sb_info *c, uint32_t ino)
{
    struct jffs2_inode_cache *ic;

    ic = jffs2_get_ino_cache(c, ino);
    if (ic)
        return ic;
    ic = jffs2_alloc_inode_cache();
    if (!ic) {
        printk(KERN_NOTICE "jffs2_scan_make_ino_cache(): allocation of inode cache failed\n");
        return NULL;
    }
    memset(ic, 0, sizeof(*ic));
```

如果 `ino` 对应的 `jffs2_inode_cache` 已经在 `inocache_list` 哈希表中,则直接返回其地址即可。否则通过 `jffs2_alloc_inode_cache` 函数从内核高速缓存中分配一个,然后初始化之:

```
    ic->ino = ino;
    ic->nlink = (void *)ic;
    jffs2_add_ino_cache(c, ic);

    if (ino == 1)
        ic->nlink=1;
    return ic;
}
```

将 `jffs2_inode_cache.ino` 设置为 `ino`, 并通过 `jffs2_add_ino_cache` 函数加入 `inocache_list` 哈希表。最后如果是根目录,则将硬链接计数 `nlink` 设置为 1。这是因为根目录没有父目录了,也就不存在代表其的 `jffs2_raw_dirent` 目录项 (另外从 `jffs2map2` 的结果来看,根目录文件的硬链接计数为 1,正是在这里设置的结果)。

jffs2_scan_dirent_node函数

在挂载文件系统时为已读出的 `jffs2_raw_dirent` 目录项数据实体创建内核描述符 `jffs2_raw_node_ref` 和临时的 `jff2_full_dirent`。如果为相应目录的 `jffs2_inode_cache` 尚未创建则创建之，并建立三者之间的连接关系。

```
static int jffs2_scan_dirent_node(struct jffs2_sb_info *c, struct jffs2_eraseblock *jeb,
                                struct jffs2_raw_dirent *rd, uint32_t ofs)
{
    struct jffs2_raw_node_ref *raw;
    struct jffs2_full_dirent *fd;
    struct jffs2_inode_cache *ic;
    uint32_t crc;
    D1(printk(KERN_DEBUG "jffs2_scan_dirent_node(): Node at 0x%08x\n", ofs));
    /* We don't get here unless the node is still valid, so we don't have to mask in the ACCURATE bit any more. */
    crc = crc32(0, rd, sizeof(*rd)-8);
    if (crc != je32_to_cpu(rd->node_crc)) {
        printk(KERN_NOTICE "jffs2_scan_dirent_node(): Node CRC failed on node at 0x%08x:
                        Read 0x%08x, calculated 0x%08x\n", ofs, je32_to_cpu(rd->node_crc), crc);
        /* We believe totlen because the CRC on the node _header_ was OK, just the node itself failed. */
        DIRTY_SPACE(PAD(je32_to_cpu(rd->totlen)));
        return 0;
    }
}
```

与 `jffs2_raw_inode` 不同，需要为 `jffs2_raw_dirent` 创建额外的内核描述符 `jffs2_full_dirent`。首先进行 `crc` 校验。计算 `jffs2_raw_dirent` 的 `crc` 校验值时排除其后 `node_crc` 和 `name_crc` 两个域，共 8 字节，然后与 `node_crc` 相比较。如果错误则整个数据实体（`jffs2_raw_inode` 及紧随其后的文件名）被认为是 `dirty`，增加 `jffs2_sb_info` 和 `jffs2_eraseblock` 中的 `dirty_size` 值、减小 `free_size` 值，并退出。

```
pseudo_random += je32_to_cpu(rd->version);
fd = jffs2_alloc_full_dirent(rd->nsize+1);
if (!fd) {
    return -ENOMEM;
}
memcpy(&fd->name, rd->name, rd->nsize);
fd->name[rd->nsize] = 0;
```

通过 `jffs2_alloc_full_dirent` 函数从内核高速缓存中分配一个 `jffs2_full_dirent` 数据结构。`jffs2_raw_dirent` 其后跟随的文件名长度为 `nsize`，而为 `jffs2_full_dirent.name` 分配空间时多分配一个字节，用来填充字符串结束符。然后将文件名复制到 `jffs2_full_dirent.name` 所指空间中。

查了下 `crc32`，实际上第一个参数 0 没有任何意义。

```
crc = crc32(0, fd->name, rd->nsize);
if (crc != je32_to_cpu(rd->name_crc)) {
    printk(KERN_NOTICE "jffs2_scan_dirent_node(): Name CRC failed on node at 0x%08x:
                        Read 0x%08x, calculated 0x%08x\n", ofs, je32_to_cpu(rd->name_crc), crc);
}
```



```

D1(printk(KERN_NOTICE "Name for which CRC failed is (now) '%s', ino #d\n",
           fd->name, je32_to_cpu(rd->ino)));
jffs2_free_full_dirent(fd);
/* FIXME: Why do we believe totlen? */
/* We believe totlen because the CRC on the node _header_ was OK, just the name failed. */
DIRTY_SPACE(PAD(je32_to_cpu(rd->totlen)));
return 0;
}

```

接着，就得验证文件名的 **crc 校验值** 了。如果错误，则增加 `jffs2_sb_info` 和 `jffs2_eraseblock` 中的 `dirty_size` 值、减小 `free_size` 值，并退出。接下来就需要为 `jffs2_raw_dirent` 创建内核描述符 `jffs2_raw_node_ref`，如果是指 `pino` 其目录文件的 `jffs2_inode_cache` 尚未创建，则创建之，并建立二者的连接关系。完全类似于 `jffs2_scan_inode_dnode` 函数，在此不再赘述。

```

raw = jffs2_alloc_raw_node_ref();
if (!raw) {
    jffs2_free_full_dirent(fd);
    printk(KERN_NOTICE "jffs2_scan_dirent_node(): allocation of node reference failed\n");
    return -ENOMEM;
}
ic = jffs2_scan_make_ino_cache(c, je32_to_cpu(rd->pino));
if (!ic) {
    jffs2_free_full_dirent(fd);
    jffs2_free_raw_node_ref(raw);
    return -ENOMEM;
}
raw->totlen = PAD(je32_to_cpu(rd->totlen));
raw->flash_offset = ofs | REF_PRISTINE;
raw->next_phys = NULL;
raw->next_in_ino = ic->nodes;
ic->nodes = raw;
if (!jeb->first_node)
    jeb->first_node = raw;
if (jeb->last_node)
    jeb->last_node->next_phys = raw;
jeb->last_node = raw;

```

需要说明的是，由于已经对目录项数据实体进行了 `crc` 校验，所以设置其内核描述符的 `REF_PRISTINE` 标志。与此相比，在 `jffs2_scan_inode_node` 函数中没有对 `jffs2_raw_inode` 数据实体立即进行 `crc` 校验（而是推迟到打开文件时），所以才设置了 `REF_UNCHECKED` 标志（参见 `jffs2_scan_inode_node` 函数的[相关部分](#)）。

在 `jffs2` 中目录项所属的目录文件由其 `pino` 表示，所以在调用 `jffs2_scan_make_ino_cache` 函数返回其所属目录文件的内核描述符时传递 `pino` 而非 `ino`！（类比 `jffs2_scan_inode_node` 函数中传递的是 `ino`，参见[上文](#)）

对于目录项数据实体，还得进一步建立 `jffs2_full_dirent` 和 `jffs2_raw_node_ref`、`jffs2_inode_cache` 之间的连接关系：

```

    fd->raw = raw;
    fd->next = NULL;
    fd->version = je32_to_cpu(rd->version);
    fd->ino = je32_to_cpu(rd->ino);
    fd->nhash = full_name_hash(fd->name, rd->nsize);
    fd->type = rd->type;
    USED_SPACE(PAD(je32_to_cpu(rd->totlen)));
    jffs2_add_fd_to_list(c, fd, &ic->scan_dents);
    return 0;
}

```

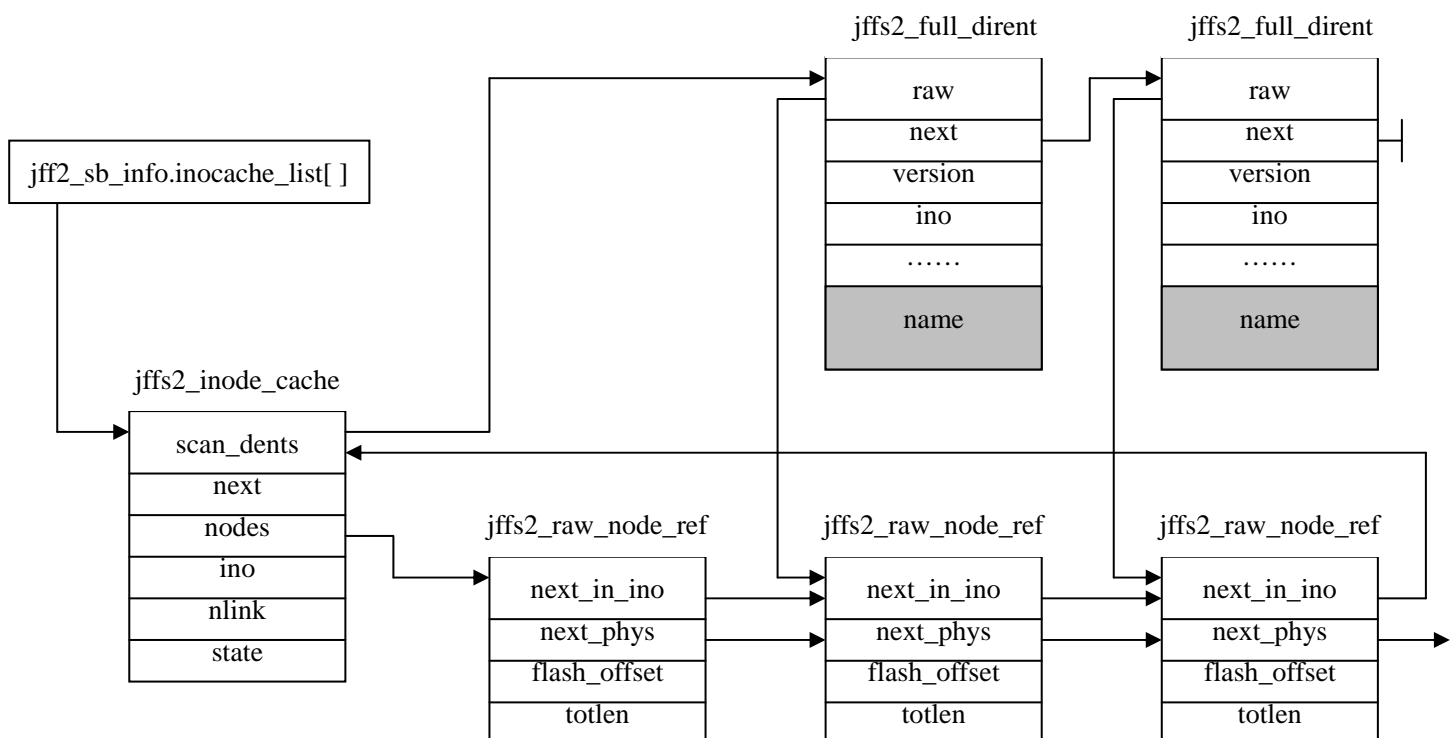
`jffs2_full_dirent` 的 `nhash` 域为根据文件名计算的一个数值，然后在通过 `jffs2_add_fd_to_list` 函数将其插入由 `jffs2_inode_cache.scan_dents` 指向的链表时，按照 `nhash` 由小到大的顺序插入。详见下文。最后在退出前还得用 `USED_SPACE` 宏增加 `jffs2_sb_info` 和当前擦除块的 `jffs2_eraseblock` 的 `used_size` 值、减小 `free_size` 值：

```

#define USED_SPACE(x) do { typeof(x) _x = (x); \
    c->free_size -= _x; c->used_size += _x; \
    jeb->free_size -= _x ; jeb->used_size += _x; \
} while(0)

```

`jffs2_scan_dirent_node` 函数完成后与目录文件相关的数据结构关系如下：



注意上图中 `nodes` 链表中含有目录文件惟一的 `jffs2_raw_inode` 的内核描述符,其相应的上层 `jffs2_full_dnode` 要等到打开目录文件时才会创建,并由目录文件 `inode` 的 `u` 域,即 `jffs2_inode_info` 的 `metadata` 指向。(另外, `nodes` 链表中各描述符的先后顺序完全由 `flash` 上各数据实体的先后顺序决定,每扫描一个就加入链表首部。)

挂载文件系统时从 `jffs2_build_filesystem` 函数到达这个函数的调用路径为:

```
jffs2_build_filesystem > jffs2_scan_medium > jffs2_scan_eraseblock > jffs2_scan_dirent_node
```

从这个函数一直返回到 `jffs2_build_filesystem` 函数后执行流会调用 `jffs2_build_inode_pass1` 函数计算各文件的硬链接计数,随后在 `jffs2_build_filesystem` 函数的第 3 阶段要释放整个 `scan_dents` 链表(参见[jffs2_build_filesystem函数的末尾](#))。与打开目录文件类比,在 `jffs2_do_read_inode` 函数中会再次为所有的目录项创建相应的 `jffs2_full_dirent` 数据结构的链表,由目录文件的 `jffs2_inode_info` 的 `dents` 域指向,参见[图 1](#)。而挂载文件系统时只存在文件的内核描述符 `jffs2_inode_cache`,所以临时链表就由其 `scan_dents` 指向。

full_name_hash函数

`include/linux/dcache.h` 文件中定义的与 `full_name_hash` 函数有关的内联函数如下:

```
/* Name hashing routines. Initial hash value */
/* Hash courtesy of the R5 hash in reiserfs modulo sign bits */
#define init_name_hash()      0

/* partial hash update function. Assume roughly 4 bits per character */
static __inline__ unsigned long
partial_name_hash(unsigned long c, unsigned long prevhash)
{
    return (prevhash + (c << 4) + (c >> 4)) * 11;
}

/* Finally: cut down the number of bits to a int value (and try to avoid losing bits) */
static __inline__ unsigned long
end_name_hash(unsigned long hash)
{
    return (unsigned int) hash;
}

/* Compute the hash for a name string. */
static __inline__ unsigned int
full_name_hash(const unsigned char * name, unsigned int len)
{
    unsigned long hash = init_name_hash();
    while (len--)
```

```

        hash = partial_name_hash(*name++, hash);
    return end_name_hash(hash);
}

```

由此可见，根据文件名的每个字符计算出一个 hash 值，然后这个值被累积到后继字符的 hash 中。直到扫描到文件名的最后一个字符才返回最终的 hash 值。

jffs2_add_fd_to_list函数

就是指将full_dirent按照名称hash值从小到大的顺序插入到链表中。

这个函数将 new 指向的 jffs2_full_dirent 元素加入(*list)指向的链表。

```

void jffs2_add_fd_to_list(struct jffs2_sb_info *c, struct jffs2_full_dirent *new, struct jffs2_full_dirent **list)
{
    struct jffs2_full_dirent **prev = list;
    D1(printk(KERN_DEBUG "jffs2_add_fd_to_list( %p, %p (->%p))\n", new, list, *list));

    while ((*prev) && (*prev)->nhash <= new->nhash) {
        if ((*prev)->nhash == new->nhash && !strcmp((*prev)->name, new->name)) {
            /* Duplicate. Free one */
            if (new->version < (*prev)->version) {
                D1(printk(KERN_DEBUG "Eep! Marking new dirent node obsolete\n"));
                D1(printk(KERN_DEBUG "New dirent is \"%s\"->ino #%%u. Old is \"%s\"->ino #%%u\n",
                    new->name, new->ino, (*prev)->name, (*prev)->ino));
                jffs2_mark_node_obsolete(c, new->raw);
                jffs2_free_full_dirent(new);
            } else {
                D1(printk(KERN_DEBUG "Marking old dirent node (ino #%%u) obsolete\n", (*prev)->ino));
                new->next = (*prev)->next;
                jffs2_mark_node_obsolete(c, ((*prev)->raw));
                jffs2_free_full_dirent(*prev);
                *prev = new;
            }
            goto out;
        }
        prev = &((*prev)->next);
    }
    new->next = *prev;
    *prev = new;
}

```

如果新元素的 nhash 值小于链表首元素的 nhash 值，则步进 prev 指针，否则将新元素插入 prev 指向的位置处。如果新元素的 nhash 值等于(*prev)所指元素的 nhash 值，则进一步比较二者的文件名是否相同。如果也相同，则说明出现了关于同一文件的目录项的重复 jffs2_full_dirent 数据结构，则需要删除版本号较小的那一个：首先通过 jffs2_mark_node_obsolete 函数标记目录项的内存描述符为过时：设置其 flash_offset 的 REF_OBSOLETE 标志（这个函数还进行了许多其它相关操作：刷新所在擦除块描述符和 jffs2_sb_info 中

的 `xxxx_size` 域，甚至要改变当前擦除块所在的 `jffs2_sb_info.xxxx_list` 链表。尚未详细研究)，然后用 `jffs2_free_full_dirent` 函数释放这个 `jffs2_full_dirent` 数据结构。

out:

```
D2(while(*list) {
printk(KERN_DEBUG "Dirent \"%s\" (hash 0x%08x, ino #%u\n",
(*list)->name, (*list)->nhash, (*list)->ino);list = &(*list)->next;}
);
}
```

jffs2_build_inode_pass1 函数

在挂载 `jffs2` 时为所有文件都调用这个函数，第二个参数 `ic` 指向文件的内核描述符。如果它是一个目录文件则增加其下所有目录项所对应文件的硬链接计数 `nlink`。

```
int jffs2_build_inode_pass1(struct jffs2_sb_info *c, struct jffs2_inode_cache *ic)
{
    struct jffs2_full_dirent *fd;
    D1(printk(KERN_DEBUG "jffs2_build_inode building inode #%u\n", ic->ino));

    if (ic->ino > c->highest_ino)
        c->highest_ino = ic->ino;
```

在 `jffs2_build_filesystem` 函数中先由 `jffs2_scan_medium` 函数为目录文件的目录项创建了 `jffs2_full_dirent` 数据结构，它们的链表由 `jffs2_inode_cache` 的 `scan_dents` 域指向。在 `jffs2_build_filesystem` 函数的最后又释放掉了所有目录文件的 `jffs2_full_dirent` 数据结构的链表。临时建立这个链表就是为了 `jffs2_build_inode_pass1` 函数使用。

如果是目录文件，则其 `jffs2_inode_cache` 的 `scan_dents` 指针非空，则遍历相应链表检索目录下的所有子目录。注意，对于非目录文件，`scan_dents` 指针为空，所以就不进入 `for` 循环而直接退出了。

```
/* For each child, increase nlink */
for(fd=ic->scan_dents; fd; fd = fd->next) {
    struct jffs2_inode_cache *child_ic;
    if (!fd->ino)
        continue;
    /* XXX: Can get high latency here with huge directories */

    child_ic = jffs2_get_ino_cache(c, fd->ino);
    if (!child_ic) {
        printk(KERN_NOTICE "Eep. Child \"%s\" (ino #%u) of dir ino #%u doesn't exist!\n",
            fd->name, fd->ino, ic->ino);
        continue;
    }
```

注意目录项中含有两个索引结点号：ino 为其代表的文件的索引结点号，pino 为其所属的目录文件的索引结点号。所以为了返回该目录项所代表的文件内核描述符，象 jffs2_get_ino_cache 函数传递 ino。如果目录项代表的文件的 jffs2_inode_cache 尚未建立，则打印警告信息，并开始新的循环访问下一目录项。

```

if (child_ic->nlink++ && fd->type == DT_DIR) {
    printk(KERN_NOTICE "Child dir \"%s\" (ino #%u) of dir ino #%u appears to be a hard link\n",
               fd->name, fd->ino, ic->ino);
    if (fd->ino == 1 && ic->ino == 1) {
        printk(KERN_NOTICE "This is mostly harmless, and probably caused by creating
                           a JFFS2 image\n");
        printk(KERN_NOTICE "using a buggy version of mkfs.jffs2. Use at least v1.17.\n");
    }
    /* What do we do about it? */
}

D1(printk(KERN_DEBUG "Increased nlink for child \"%s\" (ino #%u)\n", fd->name, fd->ino));
/* Can't free them. We might need them in pass 2 */
}

return 0;
}

```

先前在 jffs2_scan_medium 函数中为所有的文件创建 jffs2_inode_cache 时，nlink 域被设置为 0，它代表指向文件索引结点的目录项的个数。jffs2_build_inode_pass1 函数的核心操作就是“child_ic->nlink++”，即增加目录项所代表文件的硬链接个数。由于子目录、子文件的目录项属于父目录文件，所以为父目录下存在的每个目录项所代表的文件增加硬链接计数。比如，如果文件（无论是否是目录）在 A 目录下，在 B 目录中存在一个硬链接（即 B 目录的目录文件中含有代表该文件的目录项，即在 flash 上存在两个代表该文件的 jffs2_raw_dirent 数据实体（一个属于 A 目录文件，另一个属于 B 目录文件）），那么在遍历 A 目录时其 nlink 由 0 增加为 1，在遍历 B 目录时就会将 nlink 进一步增加为 2。

另外，如果目录项对应一个子目录，则进一步检查该子目录是否为根目录。根据作者的注释这种情况会在使用版本低于 1.17 的 mkfs.jffs2 时出现。

特别需要说明的是，上层 VFS 所使用的“文件硬链接计数”是其 inode 的 nlink，而不是 jffs2_inode_cache 中的 nlink！所以在打开文件时首先用 jffs2_inode_cache 中的 nlink 设置 inode 的 nlink，然后要进一步增加非叶目录的 inode 的 nlink。参见[下文](#)。

第 5 章 打开文件时建立inode的方法

挂载 jffs2 文件系统时，一旦为 flash 上所有文件和数据实体创建了相应的内核描述符后，就已经完成了挂载的大部分工作。剩下的就得为根目录“/”创建 inode 和 dentry 了。创建 inode 的工作由 iget 内联函数完成。在 jffs2_do_fill_super 函数中为根目录创建 inode 的代码摘录如下：

```
D1(printk(KERN_DEBUG "jffs2_do_fill_super(): Getting root inode\n"));
root_i = iget(sb, 1);
if (is_bad_inode(root_i)) {
    D1(printk(KERN_WARNING "get root inode failed\n"));
    goto out_nodes;
}
```

注意传递的第二个参数为相应 inode 的索引节点号，而根目录的索引节点号为 1。iget 函数的函数调用路径为：

```
iget > iget4 > get_new_inode > super_operations.read_inode (指向 jffs2_read_inode)
```

为文件创建 inode 时，首先根据其索引节点编号 ino 在索引节点哈希表 inode_hashtable 中查找，如果尚未创建，则调用 get_new_inode 函数分配一个 inode 数据结构，并用相应文件系统已注册的 read_inode_super 方法初始化。对于 ext2 文件系统，相应的 ext2_read_inode 函数将读出磁盘索引结点，而对于 jffs2 文件系统，若为目录文件，则为目录文件的所有 jffs2_raw_dirent 目录项创建相应的 jffs2_full_dirent 数据结构并组织为链表，并为其惟一的 jffs2_raw_inode 创建 jffs2_full_dnode 数据结构，并由 jffs2_inode_info 的 metadata 直接指向（对符号链接和设备文件的惟一的 jffs2_raw_inode 的处理与此相同）；若为正规文件，则为数据结点创建相应的 jffs2_full_dnode 和 jffs2_node_frag 数据结构，并由后者组织到红黑树中，最后根据文件的类型设置索引结点方法表指针 inode.i_op/i_fop/i_mapping。

iget和iget4 函数

iget 函数返回或者创建与索引结点号 ino 相应的 inode：

```
static inline struct inode *iget(struct super_block *sb, unsigned long ino)
{
    return iget4(sb, ino, NULL, NULL);
}

struct inode *iget4(struct super_block *sb, unsigned long ino, find_inode_t find_actor, void *opaque)
{
    struct list_head * head = inode_hashtable + hash(sb, ino);
    struct inode * inode;
```

文件索引节点号在文件系统所在的设备上是一一的，所以在计算索引节点号的散列值时要传递文件系统超

级块的地址。hash 函数定义于 fs/inode.c:

```
static inline unsigned long hash(struct super_block *sb, unsigned long i_ino)
{
    unsigned long tmp = i_ino + ((unsigned long) sb / L1_CACHE_BYTES);
    tmp = tmp + (tmp >> I_HASHBITS);
    return tmp & I_HASHMASK;
}
```

由此可见它仅把文件系统超级块的地址当作无符号长整型数据来使用。由于每个文件系统超级块的地址不尽相同，所以可用保证不同文件系统内相同的索引结点号在整个操作系统中是不同的。

计算出 ino 对应的散列值后，就可用得到冲突项组成的链表了，链表由 head 指向。下面就用 find_inode 函数返回该链表中 ino 相应的 inode 结构的地址：

```
spin_lock(&inode_lock);
inode = find_inode(sb, ino, head, find_actor, opaque);
if (inode) {
    __iget(inode);
    spin_unlock(&inode_lock);
    wait_on_inode(inode);
    return inode;
}
spin_unlock(&inode_lock);
```

如果这个 inode 已经存在，则返回其地址，并通过 __iget 增加其引用计数（当然 iget 还有其它操作，这里没有分析，可参见情景分析），并且通过 wait_on_inode 函数确保 inode 没有被加锁。如果已经被加锁（inode.i_state 的 I_LOCK 位被设置），则阻塞等待被解锁为止。这个函数很简单，仅罗列其代码如下：

进阶

```
static void __wait_on_inode(struct inode * inode)
{
    DECLARE_WAITQUEUE(wait, current);
    add_wait_queue(&inode->i_wait, &wait);
repeat:
    set_current_state(TASK_UNINTERRUPTIBLE);
    if (inode->i_state & I_LOCK) {
        schedule();
        goto repeat;
    }
    remove_wait_queue(&inode->i_wait, &wait);
    current->state = TASK_RUNNING;
}
```

```
static inline void wait_on_inode(struct inode *inode)
```



```
{
    if (inode->i_state & I_LOCK)
        __wait_on_inode(inode);
}
```

回到 `iget4` 函数，如果相应的 inode 不存在，则通过 `get_new_inode` 函数分配一个新的 inode：

```
/* get_new_inode() will do the right thing, re-trying the search in case it had to block at any point. */
return get_new_inode(sb, ino, head, find_actor, opaque);
}
```

get_new_inode函数

```
/*
 * This is called without the inode lock held.. Be careful.
 *
 * We no longer cache the sb_flags in i_flags - see fs.h
 * -- rmk@arm.uk.linux.org
 */
static struct inode * get_new_inode(struct super_block *sb, unsigned long ino, struct list_head *head,
                                   find_inode_t find_actor, void *opaque)
{
    struct inode * inode;
    inode = alloc_inode();
    if (inode) {
        struct inode * old;
        spin_lock(&inode_lock);
        /* We released the lock, so.. */
        old = find_inode(sb, ino, head, find_actor, opaque);
```

`alloc_inode` 宏定义为：

```
#define alloc_inode() (struct inode *) kmem_cache_alloc(inode_cachep, SLAB_KERNEL))
```

首先从 `inode_cachep` 高速缓存中分配一个 inode 结构，然后在设置该 inode 之前，再次调用 `find_inode` 函数确保此前所需的 inode 仍然没有被创建。

```
if (!old) {
    inodes_stat.nr_inodes++;
    list_add(&inode->i_list, &inode_in_use);
    list_add(&inode->i_hash, head);
```

如果真的需要新的 inode，则将其加入内核链表 `inode_in_use`，并加入索引结点哈希表 `inode_hashtable` 中 `head` 所指的冲突项链表，同时增加内核统计 `inode_stat_nr_inodes++`。

```

inode->i_sb = sb;
inode->i_dev = sb->s_dev;
inode->i_blkbits = sb->s_blocksize_bits;
inode->i_ino = ino;
inode->i_flags = 0;
atomic_set(&inode->i_count, 1);
inode->i_state = I_LOCK;
spin_unlock(&inode_lock);

```

然后，根据传递的参数 sb、s_dev、ino 来设置 inode 中的相应域。由于下面要用文件系统的 read_inode 方法来填充这个 inode，所以设置了 I_LOCK 标志以保证这个过程的原子性。

```

clean_inode(inode);
/* reiserfs specific hack right here.  We don't
** want this to last, and are looking for VFS changes
** that will allow us to get rid of it.
** -- mason@suse.com
*/
if (sb->s_op->read_inode2) {
    sb->s_op->read_inode2(inode, opaque);
} else {
    sb->s_op->read_inode(inode);
}

```

clean_inode 函数继续初始化 inode 其它的域：

```

/*
 * This just initializes the inode fields
 * to known values before returning the inode..
 *
 * i_sb, i_ino, i_count, i_state and the lists have
 * been initialized elsewhere..
 */
static void clean_inode(struct inode *inode)
{
    static struct address_space_operations empty_aops;
    static struct inode_operations empty_iops;
    static struct file_operations empty_fops;
    memset(&inode->u, 0, sizeof(inode->u));
    inode->i_sock = 0;
    inode->i_op = &empty_iops;
    inode->i_fop = &empty_fops;
    inode->i_nlink = 1;

```

```

atomic_set(&inode->i_writecount, 0);
inode->i_size = 0;
inode->i_blocks = 0;
inode->i_generation = 0;
memset(&inode->i_dquot, 0, sizeof(inode->i_dquot));
inode->i_pipe = NULL;
inode->i_bdev = NULL;
inode->i_cdev = NULL;
inode->i_data.a_ops = &empty_aops;
inode->i_data.host = inode;
inode->i_data.gfp_mask = GFP_HIGHUSER;
inode->i_mapping = &inode->i_data;
}

```

其中指向相关方法的指针都被指向空的数据结构，然后在文件系统的 `read_inode` 方法中设置为合适的值。详见下文分析。

```

/*
 * This is special! We do not need the spinlock
 * when clearing I_LOCK, because we're guaranteed
 * that nobody else tries to do anything about the
 * state of the inode when it is locked, as we
 * just created it (so there can be no old holders
 * that haven't tested I_LOCK).
 */
inode->i_state &= ~I_LOCK;
wake_up(&inode->i_wait);
return inode;
}

```

设置完 `inode` 后，根据作者的注释，任何执行流在访问 `inode.i_state` 时必须首先设置 `I_LOCK` 标志。一旦这个标志已经被设置，那么执行流就得阻塞到 `i_wait` 等待队列上。而当前执行流在设置 `inode` 期间时持有 `I_LOCK` 锁的，所以可以保证此时没有其它的执行流，即不存在竞争条件。所以这里清除 `I_LOCK` 标志时无需用自旋锁保护。同时还要唤醒任何因等待 `I_LOCK` 被清除而阻塞在 `i_wait` 等待队列上的执行流。

```

/*
 * Uhhuh, somebody else created the same inode under us. Use the old inode instead of the one we just
 * allocated.
 */
__iget(old);
spin_unlock(&inode_lock);
destroy_inode(inode);
inode = old;
wait_on_inode(inode);

```

```

    }
    return inode;
}

```

另外，如果在 `get_new_inode` 函数执行时已经由其它的执行流创建了所需的 `inode`，则释放先前获得的 `inode` 结构。增加引用计数、等待其“可用”（`I_LOCK` 标志被清除）后直接返回其地址即可。

jffs2_read_inode函数

讲得非常清晰

在 `jffs2` 文件系统源代码文件中定义了类型为 `file_system_type` 的数据结构 `jffs2_fs_type`，其 `read_super` 方法为 `jffs2_read_super`：

```
static DECLARE_FSTYPE_DEV(jffs2_fs_type, "jffs2", jffs2_read_super);
```

然后在 `jffs2` 文件系统的初始化函数 `init_jffs2_fs` 中用 `register_filesystem` 函数向系统注册了 `jffs2` 文件系统，即把 `jffs2_fs_type` 加入 `file_systems` 指向的 `file_system_type` 数据结构的链表。

在挂载 `jffs2` 文件系统时，先前注册的 `read_super` 方法，即 `jffs2_read_super` 函数被调用，在函数的开始就将文件系统超级块的 `s_op` 指针指向了 `jffs2_super_operations` 方法表（参见[“挂载文件系统”](#)）：

```
static struct super_operations jffs2_super_operations =
{
    read_inode:  jffs2_read_inode,
    put_super:   jffs2_put_super,
    write_super: jffs2_write_super,
    statfs:      jffs2_statfs,
    remount_fs:  jffs2_remount_fs,
    clear_inode: jffs2_clear_inode
};
```

其中 `read_inode` 指针正指向 `jffs2_read_inode` 函数，所以在 `get_new_inode` 函数中调用文件系统的 `read_inode` 方法初始化 `inode` 数据结构时这个函数才被调用。

```
void jffs2_read_inode (struct inode *inode)
{
    struct jffs2_inode_info *f;
    struct jffs2_sb_info *c;
    struct jffs2_raw_inode latest_node;
    int ret;

    D1(printk(KERN_DEBUG "jffs2_read_inode(): inode->i_ino == %lu\n", inode->i_ino));

    f = JFFS2_INODE_INFO(inode);
    c = JFFS2_SB_INFO(inode->i_sb);
```

```
jffs2_init_inode_info(f);
```

对于 jffs2 文件系统，inode 的 u 域为 jffs2_inode_info 数据结构，super_block 的 u 域为 jffs2_sb_info 数据结构，先前 get_new_inode 函数中已经设置 inode.i_sb 指向 super_block 了。用宏返回两个 u 域的地址，并且初始化 inode 的 u 域。

除根目录文件外，任何文件都在 flash 上至少有一个 jffs2_raw_inode 数据实体，而每个数据实体中都含有关于该文件的公共信息，比如 i_mode、i_gid、i_uid、i_size、i_atime、i_mtime、i_ctime、i_nlink 等等，所以这些域只需从 flash 中读出一个数据实体即可得到（而硬链接计数 nlink 在 jffs2_build_filesystem 函数中计算）。

如果打开目录文件，则为每个目录项创建 jffs2_full_dirent 并组织为链表，由 jffs2_inode_info 的 dents 域指向，并为其惟一的 jffs2_raw_inode 创建相应的 jffs2_full_dnode，并由 jffs2_inode_info 的 metadata 指向；如果打开正规文件，为每个数据节点创建相应的 jffs2_full_dnode/jffs2_node_frag，并组织为红黑树，树根为 jffs2_inode_info.fragtree。参见图 1，图 2。上述这两个工作都是通过 jffs2_do_read_inode 函数完成的，读出的数据实体存放在 latest_node 中。详见后文分析。

```
ret = jffs2_do_read_inode(c, f, inode->i_ino, &latest_node);
if (ret) {
    make_bad_inode(inode);
    up(&f->sem);
    return;
}
```

如果 jffs2_do_read_inode 函数失败，则通过 make_bad_inode 函数将该 inode 标记为“bad”：

```
/**
 * make_bad_inode - mark an inode bad due to an I/O error
 * @inode: Inode to mark bad
 *
 * When an inode cannot be read due to a media or remote network
 * failure this function makes the inode "bad" and causes I/O operations
 * on it to fail from this point on.
 */
```

```
void make_bad_inode(struct inode * inode)
{
    inode->i_mode = S_IFREG;
    inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
    inode->i_op = &bad_inode_ops;
    inode->i_fop = &bad_file_ops;
}
```

其中主要操作是将索引结点方法指针 i_op 指向 bad_inode_ops 方法表，而在 is_bad_inode 函数中将通过检

查 `i_op` 是否指向 `bad_inode_ops` 方法表来判断 `inode` 是否为 `bad`。比如在 GC 操作中如果发现文件的 `inode` 为 `bad`，则返回 `EIO`。如果一切顺利，就可以根据读到 `last_node` 中的数据实体的信息来设置 `inode` 的相应域了。注意 `nlink` 域为硬链接个数，在挂载文件系统后已经初步计算过。

```
inode->i_mode = je32_to_cpu(latest_node.mode);
inode->i_uid = je16_to_cpu(latest_node.uid);
inode->i_gid = je16_to_cpu(latest_node.gid);
inode->i_size = je32_to_cpu(latest_node.isize);
inode->i_atime = je32_to_cpu(latest_node.atime);
inode->i_mtime = je32_to_cpu(latest_node.mtime);
inode->i_ctime = je32_to_cpu(latest_node.ctime);
inode->i_nlink = f->inocache->nlink;
inode->i_blksize = PAGE_SIZE;
inode->i_blocks = (inode->i_size + 511) >> 9;
```

下面就得根据文件的性质，设置 `inode` 中的方法 `i_op` 和 `i_fop`：

IFMT 应该是inode file mode type ??

```
switch (inode->i_mode & S_IFMT) {
    unsigned short rdev;
case S_IFLNK:
    inode->i_op = &jffs2_symlink_inode_operations;
    break;
```

由此可见在 `jffs2` 中符号链接文件的方法为 `jffs2_symlink_inode_operations`，参见[下文](#)。

```
case S_IFDIR:
{
    struct jffs2_full_dirent *fd;
    for (fd=f->dents; fd; fd = fd->next) {
        if (fd->type == DT_DIR && fd->ino) /* 为目录文件的目录项 */
            inode->i_nlink++;
    }
    /* and '..' */
    inode->i_nlink++;
```

进阶，为啥要添加父目录的nlink数??

在挂载文件系统中、在 `jffs2_build_filesystem` 函数的最后初步为每个文件计算了 `nlink`，此时硬链接计数保存在文件的内核描述符 `jffs2_inode_cache` 中，参见[上文](#)。除根目录外任何文件都至少有一个 `jffs2_raw_dirent` 目录项，所以 `jffs2_inode_cache` 中的 `nlink` 至少为 1，而根目录文件的 `jffs2_inode_cache.nlink` 也为 1（参见[jffs2_scan_make_ino_cache函数](#)）。

需要说明的是上层 VFS 所使用的硬链接计数是其 `inode` 的 `nlink`，而不是文件内核描述符 `jffs2_inode_cache` 的 `nlink`！（对于这一点我目前理解得还不够深入）所以这里还需要进一步增加非叶目录的 `inode` 的 `nlink`：如果目录项对应一个目录，则增加父目录 `inode` 的 `nlink`（但是从 `jffs2map2` 的结果来看在任何目录下都没有“.”或者“..”目录项！值得进一步讨论，参见[附录](#)），所以首先需要判断该目录项是否对应一个目录，

jffs2_full_dirent中的type域从jffs2_raw_dirent的type域复制过来。

```

/* Root dir gets i_nlink 3 for some reason */
if (inode->i_ino == 1)
    inode->i_nlink++;

inode->i_op = &jffs2_dir_inode_operations;
inode->i_fop = &jffs2_dir_operations;
break;
}

```

由此可见在jffs2 中目录文件的方法为jffs2_dir_inode_operations，参见[下文](#)。

```

case S_IFREG:
    inode->i_op = &jffs2_file_inode_operations;
    inode->i_fop = &jffs2_file_operations;
    inode->i_mapping->a_ops = &jffs2_file_address_operations;
    inode->i_mapping->nrpages = 0;
    break;

```

对于正规文件，文件方法表指针 i_fop 被设置为指向 jffs2_file_operations 方法表，而内存映射方法表指针 i_mapping->a_ops 被设置为指向 jffs2_file_address_operations 方法表。从后文就可见读写文件时必须经过这两个方法表中的相关函数。

```

case S_IFBLK:
case S_IFCHR:
    /* Read the device numbers from the media */
    D1(printk(KERN_DEBUG "Reading device numbers from flash\n"));
    if (jffs2_read_dnode(c, f->metadata, (char *)&rdev, 0, sizeof(rdev)) < 0) { /* Eep */
        printk(KERN_NOTICE "Read device numbers for inode %lu failed\n",
               (unsigned long)inode->i_ino);

        up(&f->sem);
        jffs2_do_clear_inode(c, f);
        make_bad_inode(inode);
        return;
    }
    /* FALL THROUGH */

```

设备文件由惟一的 jffs2_raw_inode 数据实体表示，紧随其后的数据为设备号 rdev（类比在 ext2 上设备文件由一个磁盘索引结点表示，其 i_data[] 中第一个元素保存设备号）。另外目录文件、符号链接、设备文件的 jffs2_full_dnode 由 metadata 直接指向而没有加入 fragtree 红黑树中，详见 jffs2_do_read_inode 函数。通过 jffs2_read_dnode 函数读出设备号到 rdev 变量中，详见后文。需要注意的是如果读取设备号成功，则这个 case 后面没有 break，所以会进入下面的代码。

```

case S_IFSOCK:
case S_IFIFO:
    inode->i_op = &jffs2_file_inode_operations;
    init_special_inode(inode, inode->i_mode, kdev_t_to_nr(mk_kdev(rdev>>8, rdev&0xff)));
    break;
default:
    printk(KERN_WARNING "jffs2_read_inode(): Bogus imode %o for ino %lu\n",
            inode->i_mode, (unsigned long)inode->i_ino);
}
up(&f->sem);
D1(printk(KERN_DEBUG "jffs2_read_inode() returning\n"));
}

```

由此可见，对于特殊文件（设备文件、SOCKET、FIFO 文件），它们的 inode 通过 `init_special_inode` 函数来进一步初始化：

```

void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = to_kdev_t(rdev);
        inode->i_cdev = cdget(rdev);
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = to_kdev_t(rdev);
    } else if (S_ISFIFO(mode))
        inode->i_fop = &def_fifo_fops;
    else if (S_ISSOCK(mode))
        inode->i_fop = &bad_sock_fops;
    else
        printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
}

```

其实它们的 inode 的 mode 已经在前面 `jffs2_do_read_inode` 函数读取数据实体后设置好了，这里主要是设置文件方法和设备文件的 `inode.i_rdev` 域。其中参数 `rdev` 经过 `to_kdev_t` 内联函数加以格式转换后就得到 `i_rdev`（该函数及相关的宏定义在 `linux/kdev_t.h`）。

需要说明的是，在 `jffs2` 文件系统中设备文件用 flash 上一个 `jffs2_raw_inode` 数据实体表示。而内核中的 VFS 的索引结点 inode 中设计了 `i_dev` 和 `i_rdev` 两个域，分别表示设备盘索引结点所在的设备的设备号，以及它所代表的设备的设备号。而 `jffs2_raw_inode` 中也没有任何表示设备号的域。这是因为能从 flash 上访问该数据实体就当然知道其所在 flash 分区的设备号；另外紧随该数据实体后的为其所代表的设备的设备号。

jffs2_do_read_inode函数 (improved)

如前文所述，这个函数需要从 flash 中读出一个数据实体以得到文件索引节点的公共信息，同时，如果是目录文件，则为惟一的 jffs2_raw_inode 创建 jffs2_full_dnode，由 jffs2_inode_info 的 metadata 指向；为每个目录项创建 jffs2_full_dirent 并组织为链表，由 jffs2_inode_info 的 dents 域指向；如果是普通文件，为每个 jffs2_raw_inode 数据节点创建相应的 jffs2_full_dnode/jffs2_node_frag，并组织为红黑树，树根为 jffs2_inode_info 的 fragtree。

```
/* Scan the list of all nodes present for this ino, build map of versions, etc. */
int jffs2_do_read_inode(struct jffs2_sb_info *c, struct jffs2_inode_info *f,
                        uint32_t ino, struct jffs2_raw_inode *latest_node)
{
    struct jffs2_tmp_dnode_info *tn_list, *tn;
    struct jffs2_full_dirent *fd_list;
    struct jffs2_full_dnode *fn = NULL;
    uint32_t crc;
    uint32_t latest_mtime, mctime_ver;
    uint32_t mdata_ver = 0;
    size_t retlen;
    int ret;
    D2(printk(KERN_DEBUG "jffs2_do_read_inode(): getting inocache\n"));

    f->inocache = jffs2_get_ino_cache(c, ino);
    D2(printk(KERN_DEBUG "jffs2_do_read_inode(): Got inocache at %p\n", f->inocache));
}
```

由前文可知，在挂载文件系统时（在 jffs2_scan_medium 函数中）已经为 flash 上所有的文件创建了内核描述符 jffs2_inode_cache 并加入了文件系统 inocache_list 哈希表，这里直接根据索引节点号返回其地址即可。

```
if (!f->inocache && ino == 1) {
    /* Special case - no root inode on medium */
    f->inocache = jffs2_alloc_inode_cache();
    if (!f->inocache) {
        printk(KERN_CRIT "jffs2_do_read_inode(): Cannot allocate inocache for root inode\n");
        return -ENOMEM;
    }
    D1(printk(KERN_DEBUG "jffs2_do_read_inode(): Creating inocache for root inode\n"));
    memset(f->inocache, 0, sizeof(struct jffs2_inode_cache));
    f->inocache->ino = f->inocache->nlink = 1;
    f->inocache->nodes = (struct jffs2_raw_node_ref *)f->inocache;
    jffs2_add_ino_cache(c, f->inocache);
}
if (!f->inocache) {
    printk(KERN_WARNING "jffs2_do_read_inode() on nonexistent ino %u\n", ino);
    return -ENOENT;
}
```

```

}
D1(printk(KERN_DEBUG "jffs2_do_read_inode(): ino %#u nlink is %d\n", ino, f->inocache->nlink));

```

如果文件描述符尚未创建则返回错误ENOENT。而根目录文件是惟一没有jffs2_raw_inode数据实体的目录，这段代码表明作者认为先前在挂载文件系统时并没有创建根目录文件的内核描述符，所以如果文件描述符不存在且ino又等于 1，即为根目录，则在这里创建之并加入哈希表。但是我认为即使根目录不存在惟一的那个 jffs2_raw_inode，但是在挂载时也已经为其创建了 jffs2_inode_cache。这是因为在 jffs2_scan_dirent_node函数中也会调用jffs2_scan_make_ino_cache函数创建目录项所在文件的内核描述符，此时传递的是目录项中的pino参数。因此既然根目录不为空，那么挂载文件系统时就会为其创建内核描述符。参见[上文](#)。（进一步，可以尝试注销这段代码以验证我的判断是否正确。）

先前在挂载文件系统时已经为 flash 上所有的数据实体创建了内核描述符 jffs2_raw_node_ref，其中的 flash_offset 为数据实体在 flash 分区内的逻辑偏移，totlen 为其长度。而且同一个文件的 jffs2_raw_node_ref 通过 next_in_ino 域组织成一个链表，链表由文件的 jffs2_inode_cache 的 nodes 域指向。jffs2_get_inode_nodes 函数就可用利用这个链表访问文件的所有数据实体，然后：

1. 为每一个 jffs2_raw_dirent 创建 jffs2_full_dirent，并组织为链表 fd_list
2. 为每一个 jffs2_raw_inode 创建 jffs2_tmp_dnode_info 和 jffs2_full_dnode，并组织为链表 tn_list
标记为 temp node

```

/* Grab all nodes relevant to this ino */
ret = jffs2_get_inode_nodes(c, ino, f, &tn_list, &fd_list, &f->highest_version,
                           &latest_mctime, &mctime_ver);

if (ret) {
    printk(KERN_CRIT "jffs2_get_inode_nodes() for ino %u returned %d\n", ino, ret);
    return ret;
}
f->dents = fd_list;

```

对于目录文件，将其jffs2_full_dirent组成的链表由jffs2_inode_info的dents域指向，参见[图 1](#)。对于其它文件则遍历jffs2_tmp_dnode_info组成的链表，为每一个jffs2_full_dnode创建相应的jffs2_node_frag结构并加入inode.u.fragtree所指向的红黑树。最后释放整个jffs2_tmp_dnode_info链表。

```

while (tn_list) {
    tn = tn_list;
    fn = tn->fn;
    if (f->metadata && tn->version > mdata_ver) {
        D1(printk(KERN_DEBUG "Obsoleting old metadata at 0x%08x\n",
                    ref_offset(f->metadata->raw)));
        jffs2_mark_node_obsolete(c, f->metadata->raw); 将旧的metadata标记为过时
        jffs2_free_full_dnode(f->metadata);
        f->metadata = NULL;
        mdata_ver = 0;
    }
    if (fn->size) {
        jffs2_add_full_dnode_to_inode(c, f, fn);
    }
}

```

```

    } else {
        /* Zero-sized node at end of version list. Just a metadata update */
        D1(printk(KERN_DEBUG "metadata @%08x: ver %d\n", ref_offset(fn->raw), tn->version));
        f->metadata = fn;
        mdata_ver = tn->version;
    }
    tn_list = tn->next;
    jffs2_free_tmp_dnode_info(tn);
} //while

```

在这个循环中遍历 `jffs2_tmp_dnode_info` 的链表，为每个元素所指向的 `jffs2_full_dnode` 数据结构创建相应的 `jffs2_node_frag` 数据结构，并插入以 `jffs2_inode_info.fragtree` 为根红黑树。这个工作是通过函数 `jffs2_add_full_dnode_to_inode` 函数完成的（这个函数涉及红黑树的插入，尚未研究）。

对于正规文件、符号链接、设备文件，它们至少由一个后继带有数据的 `jffs2_raw_inode` 组成，所以这里都组织了红黑树，目录文件、SOCKET、FIFO 文件的 `jffs2_raw_inode` 后没有数据（所以 `fn->size` 等于 0），所以它们的 `jffs2_full_dnode` 直接由 `jffs2_inode_info` 的 `metadata` 指向。

处理完一个 `jffs2_full_dnode`，随即释放相应的 `jffs2_tmp_dnode_info`。由此可见该数据结构是在打开文件期间为处理 `jffs2_full_dnode` 数据结构而临时创建的。

```

if (!fn) {
    /* No data nodes for this inode. */
    if (ino != 1) {
        printk(KERN_WARNING "jffs2_do_read_inode(): No data nodes found for ino #%%u\n", ino);
        if (!fd_list) {
            return -EIO;
        }
        printk(KERN_WARNING "jffs2_do_read_inode(): But it has children so we fake some modes for it\n");
    }
    latest_node->mode = cpu_to_je32(S_IFDIR|S_IRUGO|S_IWUSR|S_IXUGO);
    latest_node->version = cpu_to_je32(0);
    latest_node->atime = latest_node->ctime = latest_node->mtime = cpu_to_je32(0);
    latest_node->isize = cpu_to_je32(0);
    latest_node->gid = cpu_to_je16(0);
    latest_node->uid = cpu_to_je16(0);
    return 0;
}

```

`jffs2` 中只有根目录一个文件没有 `jffs2_raw_inode` 数据实体（哪怕对于新创建、且尚未写入任何数据的正规文件，也在创建当下写入了一个临时的 `jffs2_raw_inode`，相应的 `jffs2_full_dnode` 则由 `metadata` 指向，并在稍后真正第一次写入数据时被标记为过时），所以此时可以直接初始化 `latest_node` 参数所指向的 `jffs2_raw_inode` 数据实体了。否则，就必须通过 `jffs2_flash_read` 函数从 flash 上真正读出一个来：

```

ret = jffs2_flash_read(c, ref_offset(fn->raw), sizeof(*latest_node), &retlen, (void *)latest_node);
if (ret || retlen != sizeof(*latest_node)) {
    printk(KERN_NOTICE "MTD read in jffs2_do_read_inode() failed: Returned
                        %d, %ld of %d bytes read\n", ret, (long)retlen, sizeof(*latest_node));
    /* FIXME: If this fails, there seems to be a memory leak. Find it. */
    up(&f->sem);
    jffs2_do_clear_inode(c, f);
    return ret?ret:-EIO;
}

```

在打开文件时，在 `jffs2_do_read_inode` 函数中除了为数据实体创建相应的数据结构外，还要读取一个数据实体返回给上层的 `jffs2_read_inode` 用于设置文件的 `inode` 数据结构。

（在第 1 稿中这一段没有看明白，现在懂了）对于目录文件有惟一的 `jffs2_raw_inode` 数据实体（对于根目录则前面已经返回），所以上面 `jffs2_get_inode_nodes` 返回参数 `tn_list` 至少含有一个元素，由 `fn` 指向。所以这里正是读出目录文件那个惟一的 `jffs2_raw_inode` 数据实体到 `latest_node` 所指空间中！

```

crc = crc32(0, latest_node, sizeof(*latest_node)-8);
if (crc != je32_to_cpu(latest_node->node_crc)) {
    printk(KERN_NOTICE "CRC failed for read_inode of inode %u at physical location 0x%x\n",
                ino, ref_offset(fn->raw));
    up(&f->sem);
    jffs2_do_clear_inode(c, f);
    return -EIO;
}

```

读出了数据实体后，还要进行 CRC 校验。

最后，还需要修改读出的数据实体的某些域：

```

switch(je32_to_cpu(latest_node->mode) & S_IFMT) {
case S_IFDIR:
    if (mctime_ver > je32_to_cpu(latest_node->version)) {
        /* The times in the latest_node are actually older than mctime in the latest dirent. Cheat. */
        latest_node->ctime = latest_node->mtime = cpu_to_je32(latest_mctime);
    }
    break;

```

前面在调用 `jffs2_get_inode_nodes` 函数时最后两个参数返回目录文件的所有目录项中“最近”的时间，据此修正目录文件惟一的 `jffs2_raw_inode` 中的时间戳。

```

case S_IFREG:
    /* If it was a regular file, truncate it to the latest node's isize */

```

进阶

```
jffs2_truncate_fraglist(c, &f->fragtree, je32_to_cpu(latest_node->isize));
break;
```

(尚未研究这个函数)

```
case S_IFLNK:
```

```
/* Hack to work around broken isize in old symlink code.
   Remove this when dnmw2 comes to his senses and stops
   symlinks from being an entirely gratuitous special case. */
if (!je32_to_cpu(latest_node->isize))
    latest_node->isize = latest_node->dsiz;
/* fall through... */
```

```
case S_IFBLK:
```

```
case S_IFCHR:
```

```
/* Xertain inode types should have only one data node, and it's
   kept as the metadata node */
if (f->metadata) {
    printk(KERN_WARNING "Argh. Special inode #%u with mode 0%o had metadata node\n",
           ino, je32_to_cpu(latest_node->mode));

    up(&f->sem);
    jffs2_do_clear_inode(c, f);
    return -EIO;
}
if (!frag_first(&f->fragtree)) {
    printk(KERN_WARNING "Argh. Special inode #%u with mode 0%o has no fragments\n",
           ino, je32_to_cpu(latest_node->mode));

    up(&f->sem);
    jffs2_do_clear_inode(c, f);
    return -EIO;
}
/* ASSERT: f->fraglist != NULL */
if (frag_next(frag_first(&f->fragtree))) {
    printk(KERN_WARNING "Argh. Special inode #%u with mode 0%o had more than one node\n",
           ino, je32_to_cpu(latest_node->mode));
    /* FIXME: Deal with it - check crc32, check for duplicate node, check times and discard the older
one */

    up(&f->sem);
    jffs2_do_clear_inode(c, f);
    return -EIO;
}
/* OK. We're happy */
f->metadata = frag_first(&f->fragtree)->node;
jffs2_free_node_frag(frag_first(&f->fragtree));
f->fragtree = RB_ROOT;
break;
```

```

    }
    f->inocache->state = INO_STATE_PRESENT;
    return 0;
}

```

对于符号链接，其唯一的 `jffs2_raw_inode` 后带有数据，先前其 `jffs2_full_dnode` 已经通过 `jffs2_node_frag` 加入了红黑树；对于设备文件，其 `flash` 尚设备索引节点的后继数据为设备号，先前也已经被加入红黑树。由于这些文件都只有一个数据实体，红黑树中只有一个节点，所以这里把它们都改为由 `metadata` 直接指向。

jffs2_get_inode_nodes函数

先前在挂载文件系统时已经为 `flash` 上所有的数据实体创建了内核描述符 `jffs2_raw_node_ref`，其中的 `flash_offset` 为数据实体在 `flash` 分区内的逻辑偏移，`totlen` 为其长度。而且同一个文件的 `jffs2_raw_node_ref` 通过 `next_in_ino` 域组织成一个链表，链表由文件的 `jffs2_inode_cache` 的 `nodes` 域指向。

在打开文件时这个函数就可用利用这个链表访问文件的所有数据实体，然后：

1. 为每一个 `jffs2_raw_dirent` 创建 `jffs2_full_dirent`，并组织为链表 `fd_list`，由 `fdp` 参数返回。
2. 为每一个 `jffs2_raw_inode` 创建 `jffs2_tmp_dnode_info` 和 `jffs2_full_dnode`，并组织为链表 `tn_list`，由 `tnp` 参数返回。

另外由于在挂载文件系统、为 `jffs2_raw_inode` 数据实体创建内核描述符时并没有对其后继数据进行 `crc` 校验（所以才在其内核描述符中设置了 `REF_UNCHECKED` 标志），那么现在就到了真正进行 `crc` 校验的时候了。

```

/* Get tmp_dnode_info and full_dirent for all non-obsolete nodes associated
   with this ino, returning the former in order of version */
int jffs2_get_inode_nodes(struct jffs2_sb_info *c, ino_t ino, struct jffs2_inode_info *f,
                          struct jffs2_tmp_dnode_info **tnp, struct jffs2_full_dirent **fdp,
                          uint32_t *highest_version, uint32_t *latest_mctime, uint32_t *mctime_ver)
{
    struct jffs2_raw_node_ref *ref = f->inocache->nodes;
    struct jffs2_tmp_dnode_info *tn, *ret_tn = NULL;
    struct jffs2_full_dirent *fd, *ret_fd = NULL;
    union jffs2_node_union node;
    size_t retlen;
    int err;
    *mctime_ver = 0;

    D1(printk(KERN_DEBUG "jffs2_get_inode_nodes(): ino %#lu\n", ino));
    if (!f->inocache->nodes) {
        printk(KERN_WARNING "Eep. no nodes for ino %#lu\n", ino);
    }
}

```

文件的内核描述符 `jffs2_inode_cache` 描述了文件及其数据之间的映射关系（这里采用较为“原始”的方法，即链表来直接描述映射关系，而打开文件后会在上层 `inode` 的 `u` 域中再次描述文件及其数据的映射关系，

此时就可以采用较为“高级”和多样的描述手段了，比如红黑树 `fragtree`、链表 `dents`、或单一指针 `metadata`）。首先检查 `jffs2_inode_cache` 的 `nodes` 指针不为空。

```
spin_lock_bh(&c->erase_completion_lock);
for (ref = f->inocache->nodes; ref && ref->next_in_ino; ref = ref->next_in_ino) {
    /* Work out whether it's a data node or a dirent node */
    if (ref_obsolete(ref)) {
        /* FIXME: On NAND flash we may need to read these */
        D1(printk(KERN_DEBUG "node at 0x%08x is obsoleted. Ignoring.\n", ref_offset(ref)));
        continue;
    }
    /* We can hold a pointer to a non-obsolete node without the spinlock,
       but _obsolete_ nodes may disappear at any time, if the block they're in gets erased */
    spin_unlock_bh(&c->erase_completion_lock);
```

遍历数据实体的内核描述符链表，在访问链表期间要持有 `jffs2_sb_info` 的 `erase_completion_lock` 自旋锁，并且禁止所有的下半部分（这说明在下半部分中可以同时访问该链表。是谁的下半部分？）。开始新的循环后即可释放该自旋锁；在 `for` 循环的最后、进入新的循环前重新获得该自旋锁。

从后文对写操作的分析可用看到如果对文件进行了任何修改则直接写入新的数据实体，而原有的“过时”的数据实体不做任何改动。在内核中为新的数据实体创建新的内核描述符 `jffs2_raw_node_ref`，同时将原有数据实体的 `jffs2_raw_node_ref` 标记为“过时”（设置其 `flash_offset` 域的 `REF_OBSOLETE` 标志）。

所以在遍历文件的数据实体内核描述符链表时，如果被标记为过时，那么说明相应的 `flash` 数据实体已经失效，则直接跳过之即可。（所以如果打开目录文件，则不会为过时的目录项 `jffs2_raw_dirent` 创建 `jffs2_full_dirent`；如果打开正规文件，则不会为过时的 `jffs2_raw_inode` 创建 `jffs2_tmp_dnode_info` 和 `jffs2_full_dnode`！）

```
cond_resched();
/* FIXME: point() */
err = jffs2_flash_read(c, (ref_offset(ref)), min(ref->totlen, sizeof(node)), &retlen, (void *)&node);
if (err) {
    printk(KERN_WARNING "error %d reading node at 0x%08x in get_inode_nodes()\n",
           err, ref_offset(ref));
    goto free_out;
}
/* Check we've managed to read at least the common node header */
if (retlen < min(ref->totlen, sizeof(node.u))) {
    printk(KERN_WARNING "short read in get_inode_nodes()\n");
    err = -EIO;
    goto free_out;
}
```

`jffs2_flash_read` 函数最终通过调用 `flash` 驱动的 `read_ecc` 或者 `read` 方法读出 `flash` 分区上指定偏移、长度的

数据段。如果支持直接内存映射，那么在读 NOR flash 时可以通过内存映射完成（从而节省 memcpy 的内存拷贝开销）。用 flash 驱动的 point 函数建立内存映射，在读操作完成后再用 unpoint 拆除。

jffs2_flash_read 函数的最后一个域 node 为一个共用体：

```
union jffs2_node_union {
    struct jffs2_raw_inode i;
    struct jffs2_raw_dirent d;
    struct jffs2_unknown_node u;
};
```

由于两种数据实体都含有同样的头部，所以 node 的长度应该为其中最大的 jffs2_raw_inode 数据结构的长度（不包括后继数据）。

分两步读出有效的数据实体：首先读出不包含后继数据的 jffs2_raw_dirent 或者 jffs2_raw_inode 数据实体本身，而其中的 totlen 域为整个数据实体的长度，第二次再读出后继数据。同时，根据头部信息中的 nodetype 字段即可得到数据实体的类型并分配相应的数据结构：为 jffs2_raw_dirent 分配 jffs2_full_dirent，为 jffs2_raw_inode 分配 jffs2_tmp_dnode_info 和 jffs2_full_dnode。

```
switch (je16_to_cpu(node.u.nodetype)) {
case JFFS2_NODETYPE_DIRENT:
    D1(printk(KERN_DEBUG "Node at %08x (%d) is a dirent node\n", ref_offset(ref),
                ref_flags(ref)));
    if (ref_flags(ref) == REF_UNCHECKED) {
        printk(KERN_WARNING "BUG: Dirent node at 0x%08x never got checked? How?\n",
                ref_offset(ref));
        BUG();
    }
    if (retlen < sizeof(node.d)) {
        printk(KERN_WARNING "short read in get_inode_nodes()\n");
        err = -EIO;
        goto free_out;
    }
    if (je32_to_cpu(node.d.version) > *highest_version)
        *highest_version = je32_to_cpu(node.d.version);
    if (ref_obsolete(ref)) {
        /* Obsoleted. This cannot happen, surely? dwmw2 20020308 */
        printk(KERN_ERR "Dirent node at 0x%08x became obsolete while we weren't looking\n",
                ref_offset(ref));
        BUG();
    }
}
```

读出目录项数据实体后首先进行必要的有效性检查：其内核描述符不应该是 REF_UNCHECKED 的（在挂载文件系统时为目录项数据实体创建相应的内核描述符，此时设置其标志为 REF_PRISTINE，参见

`jffs2_scan_dirent_node`函数), 否则为BUG; 如果前面`jffs2_flash_read`函数实际读出的数据量`retlen`小于`jffs2_raw_dirent`数据结构的长度, 则表明读出失败, 所以返回EIO。另外, 还要根据读出的目录项的`version`号来更新其所在目录文件的`jffs2_inode_info.highest_version`。

```
fd = jffs2_alloc_full_dirent(node.d.nsize+1);
if (!fd) {
    err = -ENOMEM;
    goto free_out;
}
memset(fd,0,sizeof(struct jffs2_full_dirent) + node.d.nsize+1);
fd->raw = ref;
fd->version = je32_to_cpu(node.d.version);
fd->ino = je32_to_cpu(node.d.ino);
fd->type = node.d.type;
```

然后, 为目录项实体 `jffs2_raw_dirent` 分配相应的 `jffs2_full_dirent` 数据结构及后继文件名的空间并初始化。而 `jffs2_full_dirent` 数据结构中的域都是从 flash 上目录项数据实体的相应域复制过来的。

```
/* Pick out the mctime of the latest dirent */
if(fd->version > *mctime_ver) {
    *mctime_ver = fd->version;
    *latest_mctime = je32_to_cpu(node.d.mctime);
}
/* memcpy as much of the name as possible from the raw dirent we've already read from the flash
*/
if (retlen > sizeof(struct jffs2_raw_dirent))
    memcpy(&fd->name[0], &node.d.name[0], min((uint32_t)node.d.nsize,
        (retlen-sizeof(struct jffs2_raw_dirent))));
```

先前给 `jffs2_flash_read` 函数传递的待读出的数据长度为 `min(ref->totlen, sizeof(node))`, 而 `node` 的大小为 `jffs2_raw_inode` 的长度, 大于 `jffs2_raw_dirent` 数据结构的长度, 所以先前的读操作至少从 flash 中读取了部分文件名 (甚至是全部的文件名)。所以这里将已读出的部分 (全部) 文件名拷贝到 `jffs2_full_dirent` 的 `name` 所指的空间中。

```
/* Do we need to copy any more of the name directly from the flash?*/
if (node.d.nsize + sizeof(struct jffs2_raw_dirent) > retlen) {
    /* FIXME: point() */
    int already = retlen - sizeof(struct jffs2_raw_dirent);
    err = jffs2_flash_read(c, (ref_offset(ref)) + retlen,
        node.d.nsize - already, &retlen, &fd->name[already]);
    if (!err && retlen != node.d.nsize - already)
        err = -EIO;
    if (err) {
        printk(KERN_WARNING "Read remainder of name in jffs2_get_inode_nodes():
```

```

                                error %d\n", err);
        jffs2_free_full_dirent(fd);
        goto free_out;
    }
}

```

正是由于第一次可能只读出了部分文件名，所以这里可能需要读出剩余的文件名。注意第一次实际读出的数据长度为 `retlen`，那么剩余文件名的起始地址在 flash 分区上的逻辑偏移为 `(ref_offset(ref) + retlen)`，而已经读出的部分文件名长度为 `already`，而 `nsiz` 为完整文件名的长度，所以二者之差为剩余文件名的长度。第二次读操作之间将剩余文件名读入到 `jffs2_full_dirent.name[already]` 所指的地方。

```

fd->nhash = full_name_hash(fd->name, node.d.nsize);
fd->next = NULL;
    /* Wheee. We now have a complete jffs2_full_dirent structure, with
       the name in it and everything. Link it into the list */
    D1(printk(KERN_DEBUG "Adding fd \"%s\", ino %#u\n", fd->name, fd->ino));
    jffs2_add_fd_to_list(c, fd, &ret_fd);
break;

```

最后，需要根据文件名计算一个“散列值”，记录到 `nhash` 域中，然后，通过 `jffs2_add_fd_to_list` 函数根据 `nhash` 值将目录文件的所有目录项的 `jffs2_full_dirent` 数据结构组织在 `ret_fd` 所指向的链表中（这个指针最终由参出返回，然后被设置到 `jffs2_inode_info.dents` 域）。

最后由 `break` 跳出 `switch` 结构，并开始新的 `for` 循环访问当前文件的下一个 flash 数据实体。

case JFFS2_NODETYPE_INODE:

```

    D1(printk(KERN_DEBUG "Node at %08x (%d) is a data node\n", ref_offset(ref), ref_flags(ref)));
    if (retlen < sizeof(node.i)) {
        printk(KERN_WARNING "read too short for dnode\n");
        err = -EIO;
        goto free_out;
    }
    if (je32_to_cpu(node.i.version) > *highest_version)
        *highest_version = je32_to_cpu(node.i.version);
    D1(printk(KERN_DEBUG "version %d, highest_version now %d\n", je32_to_cpu(node.i.version),
        *highest_version));
    if (ref_obsolete(ref)) {
        /* Obsoleted. This cannot happen, surely? dwmw2 20020308 */
        printk(KERN_ERR "Inode node at 0x%08x became obsolete while we weren't looking\n",
            ref_offset(ref));
        BUG();
    }

```

如果该数据实体为 `jffs2_raw_inode`，则于上面处理目录项数据实体类似首先进行有效性检查。由于在挂载文件系统时并没对 `jffs2_raw_inode` 数据实体进行 `crc` 校验而是推迟到了真正打开文件时，所以在其内核描述

符中设置了 `REF_UNCHECKED` 标志（参见 `jffs2_scan_inode_node` 函数的[相关部分](#)）。那么现在打开文件时就到了真正进行 `crc` 校验的时候了：对 `jffs2_raw_inode` 本身和后继数据进行 `crc` 校验。

```
/* If we've never checked the CRCs on this node, check them now. */
if (ref_flags(ref) == REF_UNCHECKED) {
    uint32_t crc;
    struct jffs2_eraseblock *jeb;
    crc = crc32(0, &node, sizeof(node.i)-8);
    if (crc != je32_to_cpu(node.i.node_crc)) {
        printk(KERN_NOTICE "jffs2_get_inode_nodes(): CRC failed on node at 0x%08x: Read
                           0x%08x, calculated 0x%08x\n",
                           ref_offset(ref), je32_to_cpu(node.i.node_crc), crc);
        jffs2_mark_node_obsolete(c, ref);
        spin_lock_bh(&c->erase_completion_lock);
        continue;
    }
}
```

`jffs2_raw_inode` 的最后两个域为其本身及其后数据的 `crc` 校验值，在计算本身的 `crc` 值时要去掉这两个域所占的 8 个字节。如果 `crc` 校验失败，则将这个数据实体的内核描述符标记为“过时”，然后获得 `erase_completion_lock` 自旋锁后开始新的循环。如果 `jffs2_raw_inode` 数据实体本身的 `crc` 校验正确，下面接着对后继数据进行 `crc` 校验：（在挂载文件系统、为数据实体建立内核描述符时已经对数据实体本身进行了 `crc` 校验，这里再次检查就重复了）

`compr`是指该inode所用的压缩算法

```
if (node.i.compr != JFFS2_COMPR_ZERO && je32_to_cpu(node.i.csize)) {
    /* FIXME: point() */
    char *buf = kmalloc(je32_to_cpu(node.i.csize), GFP_KERNEL);
    if (!buf)
        return -ENOMEM;
    err = jffs2_flash_read(c, ref_offset(ref) + sizeof(node.i), je32_to_cpu(node.i.csize),
                          &retlen, buf);
    if (!err && retlen != je32_to_cpu(node.i.csize))
        err = -EIO;
    if (err) {
        kfree(buf);
        return err;
    }
    crc = crc32(0, buf, je32_to_cpu(node.i.csize));
    kfree(buf);
    if (crc != je32_to_cpu(node.i.data_crc)) {
        printk(KERN_NOTICE "jffs2_get_inode_nodes(): Data CRC failed on node at
                           0x%08x: Read 0x%08x, calculated 0x%08x\n",
                           ref_offset(ref), je32_to_cpu(node.i.data_crc), crc);
        jffs2_mark_node_obsolete(c, ref);
        spin_lock_bh(&c->erase_completion_lock);
    }
}
```

```

        continue;
    }
}

```

如果 `node.i.compr` 等于 `JFFS2_COMPR_ZERO`，那么表示该数据实体对应的是一个洞。如果不是洞而且的确存在后继压缩过了数据，则需要进行 `crc` 校验（否则无需校验，则不进入这个 `if` 分支）。`csize` 为压缩后数据的长度。首先从 `flash` 上读出压缩数据，计算出 `crc` 值后即可释放相应缓冲区。如果 `crc` 校验失败，则将这个数据实体的内核描述符标记为“过时”并然后获得 `erase_completion_lock` 自旋锁后开始新的循环。

```

/* Mark the node as having been checked and fix the accounting accordingly */
jeb = &c->blocks[ref->flash_offset / c->sector_size];
jeb->used_size += ref->totlen;
jeb->unchecked_size -= ref->totlen;
c->used_size += ref->totlen;
c->unchecked_size -= ref->totlen;
mark_ref_normal(ref);
} //if (ref_flags(ref) == REF_UNCHECKED)

```

对数据实体进行了 `crc` 校验后，就要通过 `mark_ref_normal` 改变其内核描述符的标志为 `REF_NORMAL`，并且刷新数据实体所在擦除块描述符和文件系统超级块的 `u` 域中的 `used_size` 和 `unchecked_size` 统计信息。

```

tn = jffs2_alloc_tmp_dnode_info();
if (!tn) {
    D1(printk(KERN_DEBUG "alloc tn failed\n"));
    err = -ENOMEM;
    goto free_out;
}
tn->fn = jffs2_alloc_full_dnode();
if (!tn->fn) {
    D1(printk(KERN_DEBUG "alloc fn failed\n"));
    err = -ENOMEM;
    jffs2_free_tmp_dnode_info(tn);
    goto free_out;
}
tn->version = je32_to_cpu(node.i.version);
tn->fn->ofs = je32_to_cpu(node.i.offset);
/* There was a bug where we wrote hole nodes out with csize/dsize swapped. Deal with it */
if (node.i.compr == JFFS2_COMPR_ZERO && !je32_to_cpu(node.i.dsize) &&
    je32_to_cpu(node.i.csize))
    tn->fn->size = je32_to_cpu(node.i.csize);
else // normal case...
    tn->fn->size = je32_to_cpu(node.i.dsize);
tn->fn->raw = ref;
D1(printk(KERN_DEBUG "dnode @%08x: ver %u, offset %04x, dsize %04x\n",

```

```

        ref_offset(ref), je32_to_cpu(node.i.version),
        je32_to_cpu(node.i.offset), je32_to_cpu(node.i.dsize)));
    jffs2_add_tn_to_list(tn, &ret_tn);
    break;

```

接下来为 `jffs2_raw_inode` 分配相应的 `jffs2_tmp_dnode_info` 和 `jffs2_full_dnode` 数据结构并初始化，然后将同文件的 `jffs2_tmp_dnode_info` 组织到 `ret_tn` 所指向的链表中。最后由 `break` 退出 `switch`，开始新的循环。

注意，`jffs2_tmp_dnode_info` 数据结构组成了链表，其 `fn` 域指向相应的 `jffs2_full_dnode`，而后者的 `raw` 域指向数据实体的内核描述符。此时尚需 `jffs2_node_frag` 数据结构才能组织红黑树，这个操作在返回到上层 `jffs2_do_read_inode` 函数中才完成。（而且一旦红黑树组织完毕，`jffs2_tmp_dnode_info` 数据结构的链表即被释放）

一般情况下，文件由 `jffs2_raw_inode` 或者 `jffs2_raw_dirent` 数据实体组成。`default` 分支中处理其它特殊类型的数据节点：**（这些特殊的数据节点的作用是什么？由谁？何时写入？）**

```

default:
    if (ref_flags(ref) == REF_UNCHECKED) {
        struct jffs2_eraseblock *jeb;
        printk(KERN_ERR "Eep. Unknown node type %04x at %08x was marked
            REF_UNCHECKED\n", je16_to_cpu(node.u.nodetype), ref_offset(ref));
        /* Mark the node as having been checked and fix the accounting accordingly */
        jeb = &c->blocks[ref->flash_offset / c->sector_size];
        jeb->used_size += ref->totlen;
        jeb->unchecked_size -= ref->totlen;
        c->used_size += ref->totlen;
        c->unchecked_size -= ref->totlen;
        mark_ref_normal(ref);
    }

```

（如果搞明白了为什么会出现特殊类型的数据实体，）**如果特殊类型的数据实体尚未检查 `crc`，则硬性改为已检查过的。为什么？**

```

node.u.nodetype = cpu_to_je16(JFFS2_NODE_ACCURATE | je16_to_cpu(node.u.nodetype));
if (crc32(0, &node, sizeof(struct jffs2_unknown_node)-4) != je32_to_cpu(node.u.hdr_crc)) {
    /* Hmmm. This should have been caught at scan time. */
    printk(KERN_ERR "Node header CRC failed at %08x. But it must have been OK earlier.\n",
        ref_offset(ref));
    printk(KERN_ERR "Node was: { %04x, %04x, %08x, %08x }\n",
        je16_to_cpu(node.u.magic), je16_to_cpu(node.u.nodetype),
        je32_to_cpu(node.u.totlen), je32_to_cpu(node.u.hdr_crc));
    jffs2_mark_node_obsolete(c, ref);
}

```

对特殊类型的数据实体进行头部的 crc 校验。如果失败，则开始新的循环。否则进一步分析其类型：

```
else switch(je16_to_cpu(node.u.nodetype) & JFFS2_COMPAT_MASK) {
case JFFS2_FEATURE_INCOMPAT:
    printk(KERN_NOTICE "Unknown INCOMPAT nodetype %04X at %08x\n",
           je16_to_cpu(node.u.nodetype), ref_offset(ref));
    /* EEP */
    BUG();
    break;
```

当初在挂载文件系统时如果发现了这种类型的数据实体，则拒绝挂载文件系统。所以现在在打开文件时就不会检查出这种类型的数据实体，否则一定是 BUG。

```
case JFFS2_FEATURE_ROCOMPAT:
    printk(KERN_NOTICE "Unknown ROCOMPAT nodetype %04X at %08x\n",
           je16_to_cpu(node.u.nodetype), ref_offset(ref));
    if (!(c->flags & JFFS2_SB_FLAG_RO))
        BUG();
    break;
```

当初在挂载文件系统时如果发现了这种类型的数据实体，则把文件系统挂载为 RO 的。现在在打开文件时如果发现文件含有这种类型的数据实体，则检查文件系统是否按照 RO 方式挂载的。

```
case JFFS2_FEATURE_RWCOMPAT_COPY:
    printk(KERN_NOTICE "Unknown RWCOMPAT_COPY nodetype %04X at %08x\n",
           je16_to_cpu(node.u.nodetype), ref_offset(ref));
    break;
```

如果是这种类型的数据实体，则不做任何额外操作。

```
case JFFS2_FEATURE_RWCOMPAT_DELETE:
    printk(KERN_NOTICE "Unknown RWCOMPAT_DELETE nodetype %04X at %08x\n",
           je16_to_cpu(node.u.nodetype), ref_offset(ref));
    jffs2_mark_node_obsolete(c, ref);
    break;
```

如果是这种类型的数据实体，则标记其内核描述符为过时即可。在函数的最后，通过返回参数返回 `ret_tn` 和 `tet_fd` 链表的指针。

```
    }
} //switch
spin_lock_bh(&c->erase_completion_lock);
} //for
spin_unlock_bh(&c->erase_completion_lock);
```

```
    *tmp = ret_tn;
    *fdp = ret_fd;
    return 0;
free_out:
    jffs2_free_tmp_dnode_info_list(ret_tn);
    jffs2_free_full_dirent_list(ret_fd);
    return err;
}
```

第 6 章 jffs2 中写正规文件的方法

在打开文件、创建文件的 inode 数据结构时，在 jffs2_read_inode 函数中将正规文件的文件方法设置为：

```
case S_IFREG:
    inode->i_op = &jffs2_file_inode_operations;
    inode->i_fop = &jffs2_file_operations;
    inode->i_mapping->a_ops = &jffs2_file_address_operations;
    inode->i_mapping->npages = 0;
    break;
```

访问文件的方法由 jffs2_file_operation 方法表提供，而文件的内存映射方法由 jffs2_file_address_operation 方法表提供，它们的定义如下：

```
struct file_operations jffs2_file_operations =
{
    .llseek =    generic_file_llseek,
    .open =     generic_file_open,
    .read =     generic_file_read,
    .write =    generic_file_write,
    .ioctl =    jffs2_ioctl,
    .mmap =     generic_file_mmap,
    .fsync =    jffs2_fsync,
#ifdef LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,29)
    .sendfile = generic_file_sendfile
#endif
};
```

```
struct address_space_operations jffs2_file_address_operations =
{
    .readpage =    jffs2_readpage,
    .prepare_write = jffs2_prepare_write,
    .commit_write = jffs2_commit_write
};
```

为了提高读写效率在设备驱动程序层次上设计了缓冲机制，以页面为单位缓存文件的内容。这样做的好处是可用很容易地通过 mmap 系统调用将文件的缓冲页面直接映射到用户进程空间中去，从而实现文件的内存映射。也就是说文件的内存映射是建立在文件缓冲的基础上的。

在 inode 中设计了一个域 i_mapping，它指向一个 address_space 数据结构：

```
struct address_space {
```



```

struct list_head    clean_pages;           /* list of clean pages */
struct list_head    dirty_pages;          /* list of dirty pages */
struct list_head    locked_pages;         /* list of locked pages */
unsigned long       nrpages;              /* number of total pages */
struct address_space_operations *a_ops; /* methods */
struct inode         *host;               /* owner: inode, block_device */
struct vm_area_struct *i_mmap;            /* list of private mappings */
struct vm_area_struct *i_mmap_shared;     /* list of shared mappings */
spinlock_t          i_shared_lock;        /* and spinlock protecting it */
int                 gfp_mask;             /* how to allocate the pages */
};

```

其中的 `clean_pages`、`dirty_pages`、`locked_pages` 分别指向页高速缓存中的相关页面，`i_mmap`、`i_mmap_shared` 指向映射该文件的用户进程的线性区描述符的链表，`host` 指向该数据结构所属的 `inode`，`a_ops` 指向的 `address_space_operations` 方法表提供了文件缓冲机制和设备驱动程序之间的接口（由这个方法表中的函数最终调用设备驱动程序）。

从下文可见，当用户进程访问文件时方法表 `jffs2_file_operation` 中的相应函数会被调用，而它又会进一步调用 `address_space_operations` 方法表中的相关函数来完成与 `flash` 交换数据实体的操作。

sys_write函数

`sys_write` 函数为 `write` 系统调用的处理方法：

```

asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
{
    ssize_t ret;
    struct file * file;
    ret = -EBADF;
    file = fget(fd);

```

进程描述符 PCB 中有一个 `file_struct` 数据结构，其中的 `fd[]` 数组为 `file` 数据结构的指针数组，用进程已经打开的文件号索引。首先通过 `fget` 函数返回与打开文件号 `fd` 相对应的 `file` 结构的地址，同时增加其引用计数。

```

if (file) {
    if (file->f_mode & FMODE_WRITE) {
        struct inode *inode = file->f_dentry->d_inode;
        ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file, file->f_pos, count);
        if (!ret) {
            ssize_t (*write)(struct file *, const char *, size_t, loff_t *);
            ret = -EINVAL;
            if (file->f_op && (write = file->f_op->write) != NULL)

```

```

        ret = write(file, buf, count, &file->f_pos);
    }
}

```

然后通过 `file` 数据结构得到文件索引节点 `inode` 的指针。在进行写操作前首先要由 `locks_verify_area` 检查在待写入的区域上没有已存在的写强制锁。（这也就是“强制”锁名称的来历了：任何写入操作都会执行检查）如果通过了检查，则调用 `file->f_op` 所指方法表中的 `write` 方法。

我们没有分析 `sys_open` 函数，这里仅指出当创建 `file` 对象时会用 `inode.i_fop` 指针设置 `file.f_op`。所以这里调用的就是 `inode.i_fop` 指向的 `jffs2_file_operation` 方法表中的 `generic_file_wirte` 函数。

```

    if (ret > 0)
        dnotify_parent(file->f_dentry, DN_MODIFY);
    fput(file);
}
return ret;
}

```

当写操作完成后，要通过 `fput` 函数减少文件的 `file` 对象的引用计数。

generic_file_write函数

```

/*
 * Write to a file through the page cache.
 *
 * We currently put everything into the page cache prior to writing it.
 * This is not a problem when writing full pages. With partial pages,
 * however, we first have to read the data into the cache, then
 * dirty the page, and finally schedule it for writing. Alternatively, we
 * could write-through just the portion of data that would go into that
 * page, but that would kill performance for applications that write data
 * line by line, and it's prone to race conditions.
 *
 * Note that this routine doesn't try to keep track of dirty pages. Each
 * file system has to do this all by itself, unfortunately.
 *
 * okir@monad.swb.de
 */
ssize_t
generic_file_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    struct address_space *mapping = file->f_dentry->d_inode->i_mapping;
    struct inode *inode = mapping->host;
    unsigned long limit = current->rlim[RLIMIT_FSIZE].rlim_cur;
    current是当前进程

```

通过 file、dentry、inode 结构之间的链接关系，就可以由 file 找到文件的 inode 了。同时得到当前进程在文件大小方面的“资源限制”。

```

loff_t      pos;                                count是待写入的数据量
struct page *page, *cached_page;
ssize_t      written;
long         status = 0;
int          err;
unsigned bytes;

if ((ssize_t) count < 0)
    return -EINVAL;
if (!access_ok(VERIFY_READ, buf, count))
    return -EFAULT;

```

首先进行必要的参数检查：待写入的数据量不能小于 0，而且写入数据所在的用户空间必须是可读的。

```

cached_page = NULL;
down(&inode->i_sem);
inode的锁

```

在写操作开始前要获得信号量 i_sem。根据待写入的数据量一次写入可能要分成若干次操作才能完成，但是在整个写入操作期间当前进程一直持有这个信号量，直到在这个函数退出前（即写入操作完成后）才释放，从而实现了写操作的原子性。

```

pos = *ppos;
err = -EINVAL;
if (pos < 0)
    goto out;

```

任何针对文件的操作都是相对于进程在文件中的上下文，即文件指针 ppos 进行的。

```

err = file->f_error;
if (err) {
    file->f_error = 0;
    goto out;
}

```

执行失败的系统调用在返回用户态（即先前发出该系统调用的用户进程）前可能自动重新执行。比如在执行系统调用时发生阻塞，后因为收到信号而恢复执行，此时从结束阻塞处返回 ERESTARTSYS。在从 ret_from_sys_call 返回后在 do_signal 中处理非阻塞挂起信号，信号处理方法决定了是否自动重新执行先前阻塞过程被打断的系统调用：在由 iret 返回用户态的系统调用封装函数时可以选择修改保存在内核栈中的返回地址，使从“int 0x80”处恢复执行（在 x86 体系结构上），从而在系统调用封装例程中再次发出系统调用、而不是返回用户进程（如果系统调用正常结束，或者相应信号的处理方法不自动重新执行失败的系统调用，则应该返回到封装例程中“int 0x80”之后的指令）。

file 的 `f_error` 域用于记录错误。如果它不为 0，则自动重新执行的系统调用就没有必要重新执行了，直接从 `out` 退出，向上层返回该错误值。

```
written = 0;
```

由下文可见写入操作可能要分多次完成。`written` 记录了当前已经完成的写入量，这里首先清 0。

```
/* FIXME: this is for backwards compatibility with 2.4 */
if (!S_ISBLK(inode->i_mode) && file->f_flags & O_APPEND)
    pos = inode->i_size;
```

如果文件标志 `O_APPEND` 有效（表示只能向文件末尾追加数据），则调整写入位置为文件末尾（即文件的大小）。

```
/*
 * Check whether we've reached the file size limit.
 */
err = -EFBIG;
if (!S_ISBLK(inode->i_mode) && limit != RLIM_INFINITY) {
    if (pos >= limit) {
        send_sig(SIGXFSZ, current, 0);
        goto out;
    }
    if (pos > 0xFFFFFFFFULL || count > limit - (u32)pos) {
        /* send_sig(SIGXFSZ, current, 0); */
        count = limit - (u32)pos;
    }
}
```

接着检查文件的大小是否超过系统的设定值。如果不是允许无限地写入（限制为 `RLIM_INFINITY`，即没有限制），则如果待写入的位置大于文件大小的限制值，则给当前进程发送 `SIGXFSZ` 信号后直接退出，向上层返回的错误码为 `EFBIG`；如果待写入的数据量超过了剩余可写入的数据量，则调整待写入量为允许写入的数据量。

下面的代码与“LFS rule”有关。（它是什么？暂时没有研究。）

```
/*
 * LFS rule
 */
if (pos + count > MAX_NON_LFS && !(file->f_flags & O_LARGEFILE)) {
    if (pos >= MAX_NON_LFS) {
        send_sig(SIGXFSZ, current, 0);
        goto out;
    }
}
```

```

    }
    if (count > MAX_NON_LFS - (u32)pos) {
        /* send_sig(SIGXFSZ, current, 0); */
        count = MAX_NON_LFS - (u32)pos;
    }
}
/*
 * Are we about to exceed the fs block limit ?
 *
 * If we have written data it becomes a short write
 * If we have exceeded without writing data we send
 * a signal and give them an EFBIG.
 *
 * Linus frestrict idea will clean these up nicely..
 */

if (!S_ISBLK(inode->i_mode)) {
    if (pos >= inode->i_sb->s_maxbytes)
    {
        if (count || pos > inode->i_sb->s_maxbytes) {
            send_sig(SIGXFSZ, current, 0); super\_block中规定的文件最大大小
            err = -EFBIG;
            goto out;
        }
        /* zero-length writes at ->s_maxbytes are OK */
    }

    if (pos + count > inode->i_sb->s_maxbytes)
        count = inode->i_sb->s_maxbytes - pos;
} else {
    if (is_read_only(inode->i_rdev)) {
        err = -EPERM;
        goto out;
    }
    if (pos >= inode->i_size) {
        if (count || pos > inode->i_size) { 以字节为单位的文件大小
            err = -ENOSPC;
            goto out;
        }
    }
    if (pos + count > inode->i_size)
        count = inode->i_size - pos;
}

```

```

err = 0;
if (count == 0)
    goto out;

remove_suid(inode); 进阶
inode->i_ctime = inode->i_mtime = CURRENT_TIME;
mark_inode_dirty_sync(inode);

```

只要待写入的数据量不为 0 下面就要开始真正的写操作了。刷新 VFS 的 inode 中的时间戳，并用 mark_inode_dirty_sync 函数将其标记为“脏”，以后它就会被写回到设备索引节点了。

（没有研究 remove_suid 函数即相关的机制，根据情景分析，如果当前进程没有 setuid 权利而且目标文件具有 setuid 和 setgid 属性，则它剥夺目标文件的这些属性，详见上册 P588）

```

if (file->f_flags & O_DIRECT)
    goto o_direct;

```

文件的 O_DIRECT 标志的作用如何？是否代表 IO 设备？尚未研究相关的 generic_file_direct_IO 函数。

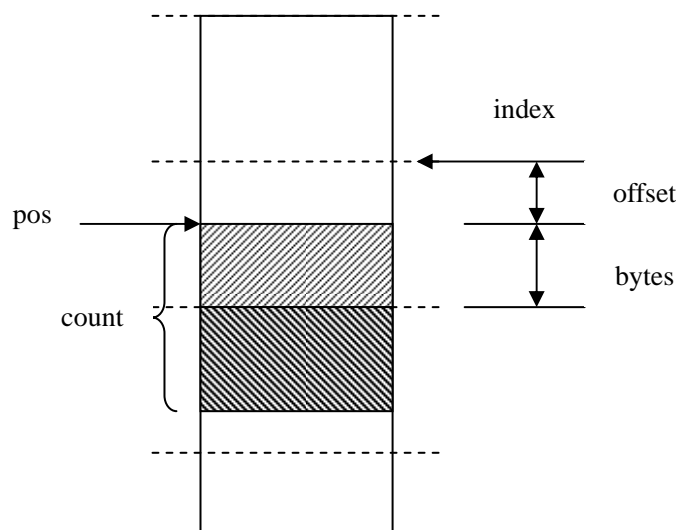
```

do {
    unsigned long index, offset;
    long page_fault;
    char *kaddr;
    /*
     * Try to find the page in the cache. If it isn't there, allocate a free page.
     */
    offset = (pos & (PAGE_CACHE_SIZE - 1)); /* Within page */
    index = pos >> PAGE_CACHE_SHIFT;
    bytes = PAGE_CACHE_SIZE - offset;
    if (bytes > count)
        bytes = count;

```

文件在逻辑上被认为是一个连续的线性空间，可用看作由若干连续页面组成。根据待写入的数据量及初始写入位置可能要在一个循环中分多次完成写入操作，而每次循环都只能针对一个页面写入。

在每次循环的开始都首先计算本次循环涉及的页面、写入位置在页面内的偏移和写入的数据量。pos 为初始写入位置在文件内的偏移，它右移页面大小即得到文件内的页面号 index，对页面大小取整即得到本次写操作在该页面内的偏移 offset，而 bytes 为写入这个页面的数据量，参见下图。如果 bytes 大于剩余待写入的数据量，则调整 bytes 的值。



```

/*
 * Bring in the user page that we will copy from _first_.
 * Otherwise there's a nasty deadlock on copying from the
 * same page as we're writing to, without it being marked
 * up-to-date.
 */
{ volatile unsigned char dummy;
  __get_user(dummy, buf); 读取一个字节
  __get_user(dummy, buf+bytes-1);
}

```

`__get_user` 函数用于访问用户空间，这里通过两个 `__get_usr` 操作读取用户空间写缓冲区的首尾字节。这样可用保证一定为用户空间缓冲区分配了相应的物理页框。作者的注释是什么意思？？

```

status = -ENOMEM; /* we'll assign it later anyway */
page = __grab_cache_page(mapping, index, &cached_page);
/* mapping是address space结构体 */
if (!page)
  break;
/* We have exclusive IO access to the page.. */
if (!PageLocked(page)) {
  PAGE_BUG(page);
}

```

前面计算出了本次循环要写入的页面，这里通过 `__grab_cache_page` 函数返回该页面在内核页高速缓存中对应的物理页框。内核页高速缓存中的所有物理页框的指针被组织在哈希表 `page_hash_table` 中，由于“页面在文件内的偏移”在系统内显然不唯一，所以在计算散列值时要使用文件的 `address_space` 结构的指针 `mapping`（将该指针值当作无符号长整型来使用）。另外，在返回页框描述符时递增了页框的引用计数，等到本次循环结束时再递减。

得到了该页面对应的物理页框的内核描述符 `page` 数据结构的指针后，就要对该页面加锁。在本次循环结束

前再解锁。

值得说明的是，一次循环只操作一个页框，所以加锁的粒度为页框。而写操作由多次循环组成，针对的是整个文件，所以加锁的粒度为整个文件。回想前面在进入循环前就获得了 `inode.i_sem` 信号量，在 `generic_file_write` 函数退出前才释放这个信号量，从而保证整个写文件操作的原子性。而在一次循环中加锁相应的页框，从而保证在循环期间对页框操作的原子性。

```
kaddr = kmap(page);
status = mapping->a_ops->prepare_write(file, page, offset, offset+bytes);
if (status)
    goto sync_failure;
```

获得了页面对应的页框后，在开始真正的写操作前还需要进行一些必须的准备操作，比如如果页框不是“Uptodate”的话就得首先从设备上读出相应的页面（因为本次写操作不一定涵盖整个页面）。对于 `ext2` 文件系统，如果该页框时刚才才分配、并加入页高速缓存的，那么还需要为页框内的磁盘块缓冲区建立相应的描述符 `buffer_head`，所有这些操作都由文件的 `address_space_operation` 方法表中 `prepare_write` 指针指向的函数完成。对于 `jffs2` 文件系统，由于底层 `flash` 驱动并不使用“磁盘块缓冲区”，所以只需要在相应的页框过时、且写入操作没有包括整个页框时读入它。另外如果 `page` 页框在文件内的起始大于文件大小，则本次循环将在文件中造成一个洞（hole），而且后面的写操作并不会描述这个洞，所以得向 `flash` 写入一个 `jffs2_raw_node` 数据实体来描述它。详见后文。（那么在 `ext2` 文件系统中是如何处理洞的？）

另外，`kmap` 函数返回相应页框的内核虚拟地址 `kaddr`。

```
page_fault = __copy_from_user(kaddr+offset, buf, bytes);
flush_dcache_page(page);
```

然后，用 `__copy_from_user` 函数从用户进程空间中读取 `buf` 缓冲区的 `bytes` 个字节到该页框内 `offset` 偏移处。注意 `offset` 和 `bytes` 在循环开始已经被设置为针对当前页框的偏移和写入量。

```
status = mapping->a_ops->commit_write(file, page, offset, offset+bytes);
if (page_fault)
    goto fail_write;
if (!status)
    status = bytes;
if (status >= 0) {
    written += status;
    count -= status;
    pos += status;
    buf += status;
}
```

将待写入的数据从用户空间复制到页高速缓存中的相应页框后，就可以进行真正的写入操作了，它由 `address_space_operation` 方法表的 `commit_write` 所指向的函数完成，即 `jffs2_commit_write` 函数，详见后文分析。在写入操作完成后根据实际写入的量刷新已写入数据量 `written`、剩余写入数据量 `count`、下次写入

位置 `pos` 和用户空间缓冲区指针 `buf`。

在 `ext2` 文件系统上写入是异步的，在 `generic_file_write` 中只需要将待写入数据复制到页高速缓存中的相应页框即可，而 `commit_write` 函数只是将脏页框提交给 `kflushd`，然后由 `kflushd` 内核线程异步地将脏页框刷新回磁盘。在 `jffs2` 文件系统上写入是同步的，在这里立即执行写入操作（由 `flash` 驱动程序提供写入时的阻塞唤醒机制）。

`unlock:`

```
kunmap(page);
/* Mark it unlocked again and drop the page.. */
SetPageReferenced(page);
UnlockPage(page);
page_cache_release(page);
if (status < 0)
    break;
} while (count);
```

在本次循环结束前还要递减页框的引用计数并解锁。

`done:`

```
*ppos = pos;
if (cached_page)
    page_cache_release(cached_page);
/* For now, when the user asks for O_SYNC, we'll actually provide O_DSYNC. */
if (status >= 0) {
    if ((file->f_flags & O_SYNC) || IS_SYNC(inode))
        status = generic_osync_inode(inode, O_SYNC_METADATA|O_SYNC_DATA);
}
```

`out_status:`

```
err = written ? written : status;
```

`out:`

```
up(&inode->i_sem);
return err;
```

在 `generic_file_write` 函数结束前还要刷新文件指针 `ppos` 为最后一次循环后 `pos` 的值，返回总的写入的数据量或者错误码，并释放加在整个文件上的信号量 `inode.i_sem`。

另外，在 `ext2` 文件系统上，如果文件标志 `O_SYNC` 有效，那么表示应该立即把相应文件页高速缓存中的脏页框刷新回磁盘。这个工作由 `generic_osync_inode` 函数完成（尚未研究该函数在 `jffs2` 文件系统中执行的具体操作）。

`fail_write:`

```
status = -EFAULT;
goto unlock;
```

```

sync_failure:
    /*
     * If blocksize < pagesize, prepare_write() may have instantiated a
     * few blocks outside i_size. Trim these off again.
     */
    kunmap(page);
    UnlockPage(page);
    page_cache_release(page);
    if (pos + bytes > inode->i_size)
        vmtruncate(inode, inode->i_size);
    goto done;
o_direct:
    written = generic_file_direct_IO(WRITE, file, (char *) buf, count, pos);
    if (written > 0) {
        loff_t end = pos + written;
        if (end > inode->i_size && !S_ISBLK(inode->i_mode)) {
            inode->i_size = end;
            mark_inode_dirty(inode);
        }
        *ppos = end;
        invalidate_inode_pages2(mapping);
    }
    /*
     * Sync the fs metadata but not the minor inode changes and
     * of course not the data as we did direct DMA for the IO.
     */
    if (written >= 0 && file->f_flags & O_SYNC)
        status = generic_osync_inode(inode, O_SYNC_METADATA);
    goto out_status;
}

```

jffs2_prepare_write函数

在 `generic_file_write` 函数的一次循环中要把 `[start, end]` 区间的数据写入 `pg` 页框，而在写入前必须完成如下准备工作：如果 `pg` 页框的在文件内的起始大于文件大小，则本次循环将在文件中造成一个洞（hole），所以得向 flash 写入一个 `jffs2_raw_node` 数据实体来描述这个洞。另外，如果 `pg` 页框的内容不是最新的，而且写入操作没有包括整个页框，则首先得从 flash 上读出该页框的内容。由 `jffs2_prepare_write` 函数完成这两个工作。

```

int jffs2_prepare_write (struct file *filp, struct page *pg, unsigned start, unsigned end)
{
    struct inode *inode = pg->mapping->host;
    struct jffs2_inode_info *f = JFFS2_INODE_INFO(inode);
    uint32_t pageofs = pg->index << PAGE_CACHE_SHIFT;

```

这里的start是页内偏移offset

```
int ret = 0;
```

如果页框位于页高速缓存，则其描述符 `page` 的 `mapping` 指向其所属文件的 `address_space` 数据结构，而其中的 `host` 即指向其所属文件的 `inode`。页框描述符 `page` 的 `index` 指明页框在相应文件内的页面号，所以 `pageofs` 为页框在文件内的逻辑偏移。

```
down(&f->sem);
```

```
D1(printk(KERN_DEBUG "jffs2_prepare_write()\n"));
```

由于 `inode.i_sem` 在 `generic_file_write/read` 期间一直被当前执行流持有，用以实现原子地读写文件，用于实现上层用户进程之间的同步，而 `jffs2_inode_info.sem` 用于实现底层读写执行流与 GC 之间的同步：

就是说底层的GC也在时不时地修改文件中的nodes链表或其他的属性，因此在prepare_write时就要注意避免竞争

在写文件时 `jffs2_prepare_write` 可能写入代表空洞的数据实体、在 `jffs2_commit_write` 中要写入新的数据实体，而 GC 内核线程每次执行时都将一个有效的数据实体的副本写入新的擦除块，即 GC 操作也是通过写入数据实体完成的。伴随着新数据实体的写入还需要：

1. 创建新的内核描述符 `jffs2_raw_node_ref`，并加入文件描述符 `jffs2_inode_cache` 的 `nodes` 链表
2. 创建新的 `jffs2_full_dnode` 数据结构，并修改红黑树中的相应结点 `jffs2_node_frag` 的 `node` 域指向这个新的数据结构

所以 `jffs2_inode_cache.nodes` 链表及红黑树结点是写操作和 GC 操作的临界资源，故必须采用同步机制避免竞争条件，这也就是设计 `jffs2_inode_info.sem` 的初衷了。（另外更底层访问 flash 芯片时的同步问题由 flash 驱动程序处理。由此可见关键是要分析清楚操作系统各个层次上的竞争条件，比如竞争条件的双方是谁，何时触发竞争条件等，然后使用合适的同步机制加以解决。）

```
if (pageofs > inode->i_size) {
```

```
    /* Make new hole frag from old EOF to new page */
```

```
    struct jffs2_sb_info *c = JFFS2_SB_INFO(inode->i_sb);
```

```
    struct jffs2_raw_inode ri;
```

```
    struct jffs2_full_dnode *fn;
```

```
    uint32_t phys_ofs, alloc_len;
```

```
    D1(printk(KERN_DEBUG "Writing new hole frag 0x%x-0x%x between current EOF and new page\n",
        (unsigned int)inode->i_size, pageofs));
```

如果新写入的页面在文件内的起始位置超过了文件的大小，则此次写入将在文件原有结尾处到该页面起始之间造成一个洞（hole），所以得向 flash 写入一个 `jffs2_raw_node` 数据实体来描述这个洞。

```
ret = jffs2_reserve_space(c, sizeof(ri), &phys_ofs, &alloc_len, ALLOC_NORMAL);
```

```
if (ret) {
```

```
    up(&f->sem);
```

```
    return ret;
```

```
}
```

在向 flash 写入 `jffs2_raw_inode` 数据实体之前得通过 `jffs2_reserve_space` 函数返回 flash 上一个合适的区间，由 `phys_ofs` 和 `alloc_len` 参数返回其位置及长度。（从 flash 上分配空间的操作可能因剩余空间不足而触发 GC，同时在选择擦除块时必须考虑 Wear Levelling 策略。这个函数尚未详细研究）

```

memset(&ri, 0, sizeof(ri));
ri.magic = cpu_to_je16(JFFS2_MAGIC_BITMASK);
ri.nodetype = cpu_to_je16(JFFS2_NODETYPE_INODE);
ri.totlen = cpu_to_je32(sizeof(ri));
ri.hdr_crc = cpu_to_je32(crc32(0, &ri, sizeof(struct jffs2_unknown_node)-4));
ri.ino = cpu_to_je32(f->inocache->ino);
ri.version = cpu_to_je32(++f->highest_version);
ri.mode = cpu_to_je32(inode->i_mode);
ri.uid = cpu_to_je16(inode->i_uid);
ri.gid = cpu_to_je16(inode->i_gid);
ri.isize = cpu_to_je32(max((uint32_t)inode->i_size, pageofs));
ri.atime = ri.ctime = ri.mtime = cpu_to_je32(CURRENT_TIME);
ri.offset = cpu_to_je32(inode->i_size);
ri.dsize = cpu_to_je32(pageofs - inode->i_size);
ri.csize = cpu_to_je32(0);
ri.compr = JFFS2_COMPR_ZERO;
ri.node_crc = cpu_to_je32(crc32(0, &ri, sizeof(ri)-8));
ri.data_crc = cpu_to_je32(0);

```

由代码可以看出，洞仅由一个 `jffs2_raw_inode` 数据实体表示而不需要后继数据，所以其头部中的 `totlen` 就等于该数据结构本身的长度。而 `offset` 和 `dsize` 分别为洞在文件内的起始位置和长度，分别为 `i_size` 和 `pageofs - i_size`。注意，如果数据实体对应一个洞，则设置其 `compr` 为 `JFFS2_COMPR_ZERO`。然后通过 `jffs2_write_dnode` 函数将该数据实体写入 flash，并创建相应的内核描述符 `jffs2_raw_node_ref`，然后组织到链表中去。详见后文。

（其实，从原有文件结尾到该页面的 `start` 前都是一个洞，即文件内部 `[i_size, pageofs + start]` 区间都是洞。但是这里的数据实体只描述了洞的前部分 `[i_size, pageofs]`，而没有包括后部分 `[pageofs, pageofs + start]`。所以我觉得洞的长度应该是 `pageofs - inode->i_size + start`）

```

fn = jffs2_write_dnode(c, f, &ri, NULL, 0, phys_ofs, NULL);
if (IS_ERR(fn)) {
    ret = PTR_ERR(fn);
    jffs2_complete_reservation(c);
    up(&f->sem);
    return ret;
}
ret = jffs2_add_full_dnode_to_inode(c, f, fn);

```

同时，还必须为数据实体创建相应的 `jffs2_full_dnode`、`jffs2_node_frag` 数据结构并刷新红黑树。

（`jffs2_add_full_dnode_to_inode` 函数在前文打开文件、创建 `inode` 时就碰到过，涉及红黑树结点的插入，或者修改已有结点。尚未深入研究）

```

if (f->metadata) {
    jffs2_mark_node_obsolete(c, f->metadata->raw);
}

```

```

    jffs2_free_full_dnode(f->metadata);
    f->metadata = NULL;
}

```

由目录文件的创建正规文件的 `create` 方法可见，在创建一个正规文件时、写入任何有意义的数据前，就首先向 `flash` 中写入了一个 `jffs2_raw_inode` 数据实体，其上层的 `jffs2_full_dnode` 则直接由 `jffs2_inode_info` 的 `metadata` 指向。而等到第一次真正写入有效数据时再将其标记为“过时”，而且以后的 `jffs2_full_dnode` 都组织在 `fragtree` 红黑树中。

```

if (ret) {
    D1(printk(KERN_DEBUG "Eep. add_full_dnode_to_inode() failed in prepare_write,
                      returned %d\n", ret));
    jffs2_mark_node_obsolete(c, fn->raw);
    jffs2_free_full_dnode(fn);
    jffs2_complete_reservation(c);
    up(&f->sem);
    return ret;
}

```

如果加入红黑树失败，则释放 `jffs2_full_dnode`。由于相应的数据实体已经写入 `flash`，所以没有删除其内核描述符，而是将其标记为“过时”。`jffs2_complete_reservation` 函数用于向 GC 内核线程发送 `SIGHUP` 信号，使其唤醒。

```

    jffs2_complete_reservation(c);
    inode->i_size = pageofs;
} //if (pageofs > inode->i_size)

```

如果一切顺利，将文件大小增加洞的长度。注意，洞本身并不属于待写入的数据，所以没有刷新剩余数据量 `count` 和已写入数据量 `written`，而只是调整了文件的大小。（为什么要在这里唤醒 GC 内核线程？调用 `jffs2_complete_reservation` 函数的时机是什么？）

```

/* Read in the page if it wasn't already present, unless it's a whole page */
if (!PageUptodate(pg) && (start || end < PAGE_CACHE_SIZE))
    ret = jffs2_do_readpage_nolock(inode, pg);
D1(printk(KERN_DEBUG "end prepare_write(). pg->flags %lx\n", pg->flags));
up(&f->sem);
return ret;
}

```

最后，如果页框的内容不是“Uptodate”的，而且待写入的区域又不包括整个页框，则必须首先从设备中读出整个页框的内容，然后再写入相应的区间，最后再把整个页框写回。否则页框内不在本次写入范围内的其它数据就会丢失。`jffs2_do_readpage_nolock` 函数的分析详见后文。

jffs2_commit_write函数

本函数将 pg 页框中[start, end]区间的数据写入 flash。首先应该写入一个 jffs2_raw_inode 数据实体，然后再写入数据。同时创建相应的内核描述符 jffs2_raw_node_ref 以及 jffs2_full_dnode 和 jffs2_node_frag，并刷新红黑树：要么插入新结点，要么刷新过时结点（从而使得红黑树只涉及有效的数据实体）。

```
int jffs2_commit_write (struct file *filp, struct page *pg, unsigned start, unsigned end)
{
    /* Actually commit the write from the page cache page we're looking at.
     * For now, we write the full page out each time. It sucks, but it's simple */
    struct inode *inode = pg->mapping->host;
    struct jffs2_inode_info *f = JFFS2_INODE_INFO(inode);
    struct jffs2_sb_info *c = JFFS2_SB_INFO(inode->i_sb);
    struct jffs2_raw_inode *ri;
    int ret = 0;
    uint32_t writtenlen = 0;
    D1(printk(KERN_DEBUG "jffs2_commit_write(): ino %lu, page at 0x%lx, range %d-%d, flags %lx\n",
        inode->i_ino, pg->index << PAGE_CACHE_SHIFT, start, end, pg->flags));

    if (!start && end == PAGE_CACHE_SIZE) {
        /* We need to avoid deadlock with page_cache_read() in
         * jffs2_garbage_collect_pass(). So we have to mark the
         * page up to date, to prevent page_cache_read() from trying to re-lock it. */
        SetPageUptodate(pg);
    }
}
```

进阶

如果写入的范围包括整个页框，那么在写入前就设置页框的“Uptodate”标志。根据作者的注释，其实这样做的目的是为了`避免和 GC 执行的 page_cache_read 发生死锁`。为什么当前写入整个页框时就会发生死锁？死锁是怎么产生的？写入部分页框时会发生死锁么？需要研究 GC。

```
ri = jffs2_alloc_raw_inode();
if (!ri) {
    D1(printk(KERN_DEBUG "jffs2_commit_write(): Allocation of raw inode failed\n"));
    return -ENOMEM;
}
/* Set the fields that the generic jffs2_write_inode_range() code can't find */
ri->ino = cpu_to_je32(inode->i_ino);
ri->mode = cpu_to_je32(inode->i_mode);
ri->uid = cpu_to_je16(inode->i_uid);
ri->gid = cpu_to_je16(inode->i_gid);
ri->isize = cpu_to_je32((uint32_t)inode->i_size); //文件大小
ri->atime = ri->ctime = ri->mtime = cpu_to_je32(CURRENT_TIME);
```

在将数据实体写入 flash 前首先得准备好一个 jffs2_raw_inode 数据实体，并根据文件的索引节点 inode 来设置它。这里只设定了部分域，剩下的与后继数据长度相关的域在下面写入 flash 的函数中再接着设置。这是因为根据数据量可能需要写入多个 jffs2_raw_inode 数据实体，而这又是因为数据实体的后继数据有最大长度限制。但是这些数据实体的 jffs2_raw_inode 中含有关于该文件的相同的信息，而这些相同的信息在这里就可以设置了，剩下与数据长度相关的域及版本号在写入 flash 的函数中才能确定。

下面将 jffs2_raw_inode 及相应的数据一起写入 flash。注意先前在 generic_file_write 函数中用 kmap 函数得到了 pg 页框的内核虚拟地址，所以传递的第四个参数为该页框内写入位置的内核虚拟地址，而第五个参数为写入位置在文件内的逻辑偏移，写入长度为 end - start，实际写入的数据量由 writtenlen 参数返回。另外，在这个函数中还要创建数据实体的内核描述符 jffs2_raw_node_ref 和 jffs2_full_dnode、jffs2_node_frag，并刷新红黑树。详见下文。

```
/* We rely on the fact that generic_file_write() currently kmaps the page for us. */
//传递数据的起始虚拟地址和在文件内的逻辑偏移（jffs2_raw_inode.offset 记录数据在文件内的偏移）
ret = jffs2_write_inode_range(c, f, ri, page_address(pg) + start,
                             (pg->index << PAGE_CACHE_SHIFT) + start, end - start, &writtenlen);

if (ret) {
    /* There was an error writing. */
    SetPageError(pg);
}

if (writtenlen) {
    if (inode->i_size < (pg->index << PAGE_CACHE_SHIFT) + start + writtenlen) {
        inode->i_size = (pg->index << PAGE_CACHE_SHIFT) + start + writtenlen;
        inode->i_blocks = (inode->i_size + 511) >> 9; i_blocks是文件的块数
        inode->i_ctime = inode->i_mtime = je32_to_cpu(ri->ctime);
    }
}

jffs2_free_raw_inode(ri);
```

(pg->index << PAGE_CACHE_SHIFT + start) 为写入位置在文件内的逻辑偏移，而本次写入的数据量为 writtenlen，所以它们的和为新的文件末尾位置。写操作完成后设置文件大小 i_size 为这个值，并释放 jffs2_raw_inode 数据实体。

```
if (start+writtenlen < end) {
    /* generic_file_write has written more to the page cache than we've
       actually written to the medium. Mark the page !Uptodate so that it gets reread */
    D1(printk(KERN_DEBUG "jffs2_commit_write(): Not all bytes written. Marking page !uptodate\n"));
    SetPageError(pg);
    ClearPageUptodate(pg);
}

D1(printk(KERN_DEBUG "jffs2_commit_write() returning %d\n",writtenlen?writtenlen:ret));
return writtenlen?writtenlen:ret;
}
```


由上文可见，在函数的开始如果发现写入范围包括了整个页框则设置了其“Uptodate”标志。如果写入操作完成后发现没有写入额定的数据，即没有写完整个页框，则必须清除页框的“Uptodate”标志。

另外，在 `generic_file_write` 的一个循环中写入一个页框。在一个页框写入后怎么没看到设置其“Uptodate”标志？？

jffs2_write_inode_range函数

该函数将 `jffs2_raw_inode` 及相应的数据一起写入 flash。第四个参数为该页框内写入位置的内核虚拟地址，而第五个参数为写入位置在文件内的逻辑偏移，写入长度为 `end - start`，实际写入的数据量由 `writtenlen` 参数返回。另外，在这个函数中还要创建数据实体的内核描述符 `jffs2_raw_node_ref` 和 `jffs2_full_dnode`、`jffs2_node_frag`，并刷新红黑树。

```
/* The OS-specific code fills in the metadata in the jffs2_raw_inode for us, so that
   we don't have to go digging in struct inode or its equivalent. It should set:
   mode, uid, gid, (starting)isize, atime, ctime, mtime */
int jffs2_write_inode_range(struct jffs2_sb_info *c, struct jffs2_inode_info *f,
                           struct jffs2_raw_inode *ri, unsigned char *buf, 这里的buf就是内存中的page
                           uint32_t offset, uint32_t writelen, uint32_t *retlen)
                           这里的offset是相对于整个文件来说的，即是从文件的第offset字节开始
{
    int ret = 0;
    uint32_t writtenlen = 0;
    D1(printk(KERN_DEBUG "jffs2_write_inode_range(): Ino #%u, ofs 0x%x, len 0x%x\n",
               f->inocache->ino, offset, writelen));
```

因为每个数据实体所携带的数据有长度限制，所以根据待写入数据量可能需要写入多个 `jffs2_raw_inode` 数据实体。但是这些数据实体的 `jffs2_raw_inode` 中含有关于该文件的相同的信息，而这些相同的信息在 `jffs2_commit_write` 函数中就已经设置好了，而剩下的与数据长度相关的域及版本号在这里才能确定。

下面就在一个循环中写入若干数据实体，以便把所有的数据都写入 flash。在写入一个数据实体前可能要首先压缩数据，并设置 `jffs2_raw_inode` 中与后继数据长度相关的域。在写入操作完成后，还要为数据实体创建内核描述符 `jffs2_raw_node_ref` 及相应的 `jffs2_full_dnode` 和 `jffs2_node_frag`，并刷新红黑树：要么插入新结点，要么刷新结点指向新的 `jffs2_node_frag`（从而使得红黑树只与有效数据实体有关），并标记相关区域的原有数据实体为“过时”。

```
while(writelen) {
    struct jffs2_full_dnode *fn;
    unsigned char *comprbuf = NULL;
    unsigned char comprtype = JFFS2_COMPR_NONE;
    uint32_t phys_ofs, alloclen;
    uint32_t datalen, cdatalen;
    D2(printk(KERN_DEBUG "jffs2_commit_write() loop: 0x%x to write to 0x%x\n", writelen, offset));

    ret = jffs2_reserve_space(c, sizeof(*ri) + JFFS2_MIN_DATA_LEN, &phys_ofs, &alloclen,
                              ALLOC_NORMAL);
```



```

if (ret) {
    D1(printk(KERN_DEBUG "jffs2_reserve_space returned %d\n", ret));
    break;
}
down(&f->sem);

```

首先通过 `jffs2_reserve_space` 函数在 flash 上找到一个合适大小的空间，参数 `phys_ofs` 和 `alloclen` 返回该空间的位置和大小。注意传递的第二个参数为 `sizeof(*ri) + JFFS2_MIN_DATA_LEN`，可见数据实体后继数据是有最大长度限制的。在写操作期间要持有信号量 `jffs2_inode_info.sem`，它用于实现写操作和 GC 操作之间的同步，参见[上文](#)。

```

datalen = writelen;
cdatalen = min(alloclen - sizeof(*ri), writelen);
comprbuf = kmalloc(cdatalen, GFP_KERNEL);

```

`writelen` 为剩余待写入的数据量，而给本次写操作分配的空间长度为 `alloclen`，所以本次写操作实际写入的原始数据量为 `cdatalen`。这里按照本次操作的原始写入量分配一个缓冲区用于存放压缩后的待写入数据，然后通过 `jffs2_compress` 函数压缩原始数据，其参数为：原始数据在 `buf` 中，长度为 `datalen`。函数返回后 `datalen` 为实际被压缩了数据量；压缩后的数据存放在 `comprbuf` 中，缓冲区长度为 `cdatalen`。函数返回后为实际被压缩了的数据长度（可能小于 `comprbuf` 的长度）（尚未研究 jffs2 所采用的压缩算法）：

```

if (comprbuf) {
    comprtype = jffs2_compress(buf, comprbuf, &datalen, &cdatalen);
}
if (comprtype == JFFS2_COMPR_NONE) {
    /* Either compression failed, or the allocation of comprbuf failed */
    if (comprbuf)
        kfree(comprbuf);
    comprbuf = buf;
    datalen = cdatalen;
}

```

如果压缩失败，则数据实体的后继数据没有被压缩，所以实际参与压缩的数据量 `datalen` 就等于先前参与压缩的数据量 `cdatalen`，而 `comprbuf` 也指向原始数据缓冲区 `buf`；如果压缩成功，则根据 `jffs2_compress` 函数语义，`datalen` 为实际参与压缩的数据量，`cdatalen` 为压缩后的数据量，下面就可以根据这两个返回参数确定数据实体中与后继数据长度相关的域了：

```

/* Now comprbuf points to the data to be written, be it compressed or not.
   comprtype holds the compression type, and comprtype == JFFS2_COMPR_NONE means
   that the comprbuf doesn't need to be kfree()d. */
ri->magic = cpu_to_je16(JFFS2_MAGIC_BITMASK);
ri->nodetype = cpu_to_je16(JFFS2_NODETYPE_INODE);
ri->totlen = cpu_to_je32(sizeof(*ri) + cdatalen);
ri->hdr_crc = cpu_to_je32(crc32(0, ri, sizeof(struct jffs2_unknown_node)-4));

```

```

ri->ino = cpu_to_je32(f->inocache->ino);
ri->version = cpu_to_je32(++f->highest_version);
ri->isize = cpu_to_je32(max(je32_to_cpu(ri->isize), offset + datalen));
ri->offset = cpu_to_je32(offset);
ri->csize = cpu_to_je32(cdatalen);
ri->dsize = cpu_to_je32(datalen);
ri->compr = comprtype;
ri->node_crc = cpu_to_je32(crc32(0, ri, sizeof(*ri)-8));
ri->data_crc = cpu_to_je32(crc32(0, comprbuf, cdatalen));

```

由此可见，数据实体头部中的 totlen 为数据实体本身及后继数据的长度；同一个文件的所有数据实体的 version 号递增（一个文件的最高 version 号记录在 jffs2_inode_info.highest_verison）；数据实体中的 csize 和 dsize 分别为压缩了的和解压缩后的数据长度；compr 为对压缩算法的描述；node_crc 为数据实体本身的 crc 校验值，data_crc 为后继数据本身的校验值。另外，offset 为该数据实体的后继数据在文件内的逻辑偏移。

```
fn = jffs2_write_dnode(c, f, ri, comprbuf, cdatalen, phys_ofs, NULL);
```

准备好数据实体和后继数据后，就可以将它们顺序地写入 flash 了。jffs2_write_dnode 函数将 jffs2_raw_inode 和压缩过的数据写入 flash 上 phys_ofs 处，同时分配内核描述符 jffs2_raw_node_ref 以及相应的 jffs2_full_dnode，将前者加入文件的链表并返回后者的地址。

```

if (comprtype != JFFS2_COMPR_NONE)
    kfree(comprbuf);
if (IS_ERR(fn)) {
    ret = PTR_ERR(fn);
    up(&f->sem);
    jffs2_complete_reservation(c);
    break;
}
ret = jffs2_add_full_dnode_to_inode(c, f, fn);

```

写入 flash 完成后即可释放缓存压缩数据的缓冲区并释放 jffs2_inode_info.sem 信号量。在 jffs2_write_dnode 函数中创建、注册了数据实体的内核描述符，还要用 jffs2_add_full_dnode_to_inode 函数在文件的红黑树中查找相应数据实体的 jffs2_node_frag 数据结构，如果没找到，则创建新的并插入红黑树；如果找到，则将其改为指向新的 jffs2_full_node，并递减原有 jffs2_full_node 的移用计数，并标记原有数据结点的内核描述为过时。

```

if (f->metadata) {
    jffs2_mark_node_obsolete(c, f->metadata->raw);
    jffs2_free_full_dnode(f->metadata);
    f->metadata = NULL;
}

```

这里还有一个需要考虑的问题。当调用目录文件的 create 方法创建正规文件时，除了向父目录文件写入其目

第6章只讨论写正规文件的方法，因此这些函数应该都是用于正规文件的，所以在这里只需判断一下metadata指针是否有效，有效就可以标记为obsolete
 录项jffs2_raw_dirent数据实体外，还要向flash中写入“代表文件存在”的第一个jffs2_raw_inode数据实体。注意此时并没有需要写入的数据，所以这个jffs2_raw_inode数据实体的dsize域为 0，而且其jffs2_full_dnode由jffs2_inode_info的metadata直接指向而没有组织到fragtree红黑树中，参见jffs2_do_create函数的[相关部分](#)。直到真正进行第一次写操作时将这个数据结点标记为过时的，这也就是上段代码的作用了。另外，由jffs2map2可以观察到当用“echo 1 > 1.txt”创建正规文件 1.txt时，该文件存在两个数据实体，相信被标记为过时的那个就是在创建时写入的，而第二个数据实体含有该文件的数据。参见[附录](#)。

```

if (ret) { /* Eep */
    D1(printk(KERN_DEBUG "Eep. add_full_dnode_to_inode() failed in commit_write, returned
                %d\n", ret));

    jffs2_mark_node_obsolete(c, fn->raw);
    jffs2_free_full_dnode(fn);
    up(&f->sem);
    jffs2_complete_reservation(c);
    break;
}
up(&f->sem);
jffs2_complete_reservation(c);

if (!datalen) {
    printk(KERN_WARNING "Eep. We didn't actually write any data in ffs2_write_inode_range()\n");
    ret = -EIO;
    break;
}
D1(printk(KERN_DEBUG "increasing writtenlen by %d\n", datalen));

writtenlen += datalen;
offset += datalen;
writelen -= datalen;
buf += datalen;
} //while
*retlen = writtenlen;
return ret;
}

```

在函数的最后刷新 writtenlen 和 offset: datalen 在 jffs2_compress 函数返回后为实际参与压缩的数据量，用它递增已写入数据量 writtenlen、待写入数据在文件内的逻辑偏移 offset、待写入数据在缓冲区内的偏移 buf 并递减待写入数据量 writelen。

jffs2_write_dnode函数

jffs2_write_dnode 函数将 ri 所指 jffs2_raw_inode 和 data 缓冲区内长度为 datalen 的数据写入 flash 上 phys_ofs 处，同时分配内核描述符 jffs2_raw_node_ref 以及相应的 jffs2_full_dnode，将前者加入文件的链表并返回后者的地址。

```

/* jffs2_write_dnode - given a raw_inode, allocate a full_dnode for it,
   write it to the flash, link it into the existing inode/fragment list */
struct jffs2_full_dnode *jffs2_write_dnode(struct jffs2_sb_info *c, struct jffs2_inode_info *f,
                                           struct jffs2_raw_inode *ri, const unsigned char *data,
                                           uint32_t datalen, uint32_t flash_ofs, uint32_t *writelen)
{
    struct jffs2_raw_node_ref *raw;
    struct jffs2_full_dnode *fn;
    size_t retlen;
    struct iovec vecs[2];
    int ret;
    unsigned long cnt = 2;

    D1(if(je32_to_cpu(ri->hdr_crc) != crc32(0, ri, sizeof(struct jffs2_unknown_node)-4)) {
        printk(KERN_CRIT "Eep. CRC not correct in jffs2_write_dnode()\n");
        BUG();});

    vecs[0].iov_base = ri;
    vecs[0].iov_len = sizeof(*ri);
    vecs[1].iov_base = (unsigned char *)data;
    vecs[1].iov_len = datalen;

```

显然写入操作分两步完成，依次写入数据实体及后继数据，于是用两个 `iovec` 类型的变量分别指向它们的基址和长度。

```

writecheck(c, flash_ofs); 不知道这个函数干嘛的，全pdf这个函数只出现一次
if (je32_to_cpu(ri->totlen) != sizeof(*ri) + datalen) {
    printk(KERN_WARNING "jffs2_write_dnode: ri->totlen (0x%08x) != sizeof(*ri) (0x%08x) + datalen
        (0x%08x)\n", je32_to_cpu(ri->totlen), sizeof(*ri), datalen);
}

raw = jffs2_alloc_raw_node_ref();
if (!raw)
    return ERR_PTR(-ENOMEM);
fn = jffs2_alloc_full_dnode();
if (!fn) {
    jffs2_free_raw_node_ref(raw);
    return ERR_PTR(-ENOMEM);
}
raw->flash_offset = flash_ofs;
raw->totlen = PAD(sizeof(*ri)+datalen);
raw->next_phys = NULL;
fn->ofs = je32_to_cpu(ri->offset);

```

```

fn->size = je32_to_cpu(ri->dsiz);
fn->frags = 0;
fn->raw = raw;

```

在写入前还要为新的数据实体创建相应的内核描述符 `jff2_raw_node_ref` 和 `jffs2_full_dnode` 数据结构，内描述符的 `flash_offset` 和 `totlen` 为数据实体在 **flash 分区内**的逻辑偏移和整个数据实体的长度。由于此时尚未加入相应文件的链表，所以 `next_phys` 域暂时设置为 `NULL`；后者用于描述后继数据在文件内的位置和长度，所以 `ofs` 和 `size` 域分别为相应数据在文件内的逻辑偏移和长度。由于此时尚未创建相应的 `jffs2_node_frag` 数据结构，所以其 `frags` 域设置为 0。

```

/* check number of valid vecs */
if (!datalen || !data)
    cnt = 1;
ret = jffs2_flash_writew(c, vecs, cnt, flash_ofs, &retlen);

```

前面已经用两个 `iovec` 类型的数据结构描述好了数据实体及后继数据的基地址及长度，这里就可以通过 `jffs2_flash_writew` 函数将它们写入 flash 了（这个函数最终调用 flash 驱动的 `mtd->writew` 方法）。`retlen` 返回实际写入的数据量。如果实际写入的数据量小于整个数据实体的长度、或者函数返回错误，则需要立即处理：

```

if (ret || (retlen != sizeof(*ri) + datalen)) {
    printk(KERN_NOTICE "Write of %d bytes at 0x%08x failed. returned %d, retlen %d\n",
           sizeof(*ri)+datalen, flash_ofs, ret, retlen);
    /* Mark the space as dirtied */
    if (retlen) {
        /* Doesn't belong to any inode */
        raw->next_in_ino = NULL;
        /* Don't change raw->totlen to match retlen. We may have
           written the node header already, and only the data will
           seem corrupted, in which case the scan would skip over
           any node we write before the original intended end of this node */
        raw->flash_offset |= REF_OBSOLETE;
        jffs2_add_physical_node_ref(c, raw);
        jffs2_mark_node_obsolete(c, raw);
    }
}

```

如果实际写入的数据量小于整个数据实体的长度，即认为只写入了部分数据，而数据实体本身认为已完整写入。根据作者的注释此时并不改变头部中的 `totlen` 域，而是将内核描述标记为过时，并照样加入文件的链表。

```

else {
    printk(KERN_NOTICE "Not marking the space at 0x%08x as dirty because the flash driver
                       returned retlen zero\n", raw->flash_offset);
    jffs2_free_raw_node_ref(raw);
}

```

```

    }
    /* Release the full_dnode which is now useless, and return */
    jffs2_free_full_dnode(fn);
    if (writelen)
        *writelen = retlen;
    return ERR_PTR(ret?-EIO);
}
/* Mark the space used */
if (datalen == PAGE_CACHE_SIZE)
    raw->flash_offset |= REF_PRISTINE;
else
    raw->flash_offset |= REF_NORMAL;

```

如果成功写入，则需要设置数据实体的内核描述符的相关标志：如果后继数据为整个页框大小，则设置 `REF_PRISTINE` 标志，至少也是 `REF_NORMAL` 标志。然后，将其内核描述符加入文件的 `jffs2_inode_cache` 的 `nodes` 域指向的链表的首部，并通过 `jffs2_add_physical_node_ref` 函数更新相应 flash 擦除块和文件系统内的相关统计信息：

```

jffs2_add_physical_node_ref(c, raw);
/* Link into per-inode list */
raw->next_in_ino = f->inocache->nodes;
f->inocache->nodes = raw;

D1(printk(KERN_DEBUG "jffs2_write_dnode wrote node at 0x%08x with dsize 0x%x, csize 0x%x,
                    node_crc 0x%08x, data_crc 0x%08x, totlen 0x%08x\n",
                    flash_ofs, je32_to_cpu(ri->dsize), je32_to_cpu(ri->csize),
                    je32_to_cpu(ri->node_crc), je32_to_cpu(ri->data_crc), je32_to_cpu(ri->totlen)));
if (writelen)
    *writelen = retlen;
f->inocache->nodes = raw;
return fn;
}

```

第 7 章 jffs2 中读正规文件的方法

与写正规文件类似，读正规文件时函数调用路径如下：

sys_read > do_generic_file_read > jffs2_readpage > jffs2_do_readpage_unlock > jffs2_do_readpage_nolock

在 do_generic_file_read 函数中需要处理预读，并且在一个循环中通过 inode.i_mapping->a_ops->readpage 方法依次读出文件的各个页面到页高速缓存的相应页框中，而这个方法即为 jffs2_readpage 函数。

jffs2_readpage函数

回顾：文件是由一页页组成的

```
int jffs2_readpage (struct file *filp, struct page *pg)
{
    struct jffs2_inode_info *f = JFFS2_INODE_INFO(pg->mapping->host);
    int ret;

    down(&f->sem);
    ret = jffs2_do_readpage_unlock(pg->mapping->host, pg);
    up(&f->sem);
    return ret;
}

int jffs2_do_readpage_unlock(struct inode *inode, struct page *pg)
{
    int ret = jffs2_do_readpage_nolock(inode, pg);
    unlock_page(pg);
    return ret;
}
```

在 do_generic_file_read 函数中启动一个页面的读操作前，已经通过 lock_page 获得了页高速缓存中页框的锁，所以在读操作完成后还要释放锁。

jffs2_do_readpage_nolock函数

这个函数读出文件中指定的一页内容到其页高速缓存中的相应页框中（页框描述符由 pg 参数指向）。

```
int jffs2_do_readpage_nolock (struct inode *inode, struct page *pg)
{
    struct jffs2_inode_info *f = JFFS2_INODE_INFO(inode);
    struct jffs2_sb_info *c = JFFS2_SB_INFO(inode->i_sb);
    unsigned char *pg_buf;
    int ret;
```

```

D1(printk(KERN_DEBUG "jffs2_do_readpage_nolock(): ino %#lu, page at offset 0x%lx\n", inode->i_ino,
pg->index << PAGE_CACHE_SHIFT));

if (!PageLocked(pg))
    PAGE_BUG(pg);

```

在操作函数调用链的上游、在 `do_generic_file_read` 函数中就已经获得了该页框的锁，否则为 BUG。

```

pg_buf = kmap(pg);
/* FIXME: Can kmap fail? */
ret = jffs2_read_inode_range(c, f, pg_buf, pg->index << PAGE_CACHE_SHIFT,
PAGE_CACHE_SIZE);

```

然后，由 `kmap` 函数返回相应页框的内核虚拟地址，并由 `jffs2_read_inode_range` 函数读入整个页面的内容到该页框中。详见下文。

```

if (ret) {
    ClearPageUptodate(pg);
    SetPageError(pg);
} else {
    SetPageUptodate(pg); //如果读成功，则设置 uptodate 标志
    ClearPageError(pg);
}
flush_dcache_page(pg); 进阶
kunmap(pg);
D1(printk(KERN_DEBUG "readpage finished\n"));
return 0;
}

```

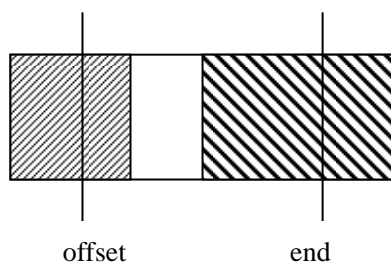
最后，根据读操作完成的情况设置页框的“Uptodate”标志、清除错误标志，或者反之。并由 `kunmap` 函数解除内核页表中对相应页框的映射。

jffs2_read_inode_range函数

`jffs2_read_inode_range` 函数用于从 flash 上读取文件的一个页框中 `[offset, offset + len]` 区域的内容到页高速缓存中。在打开文件、读 inode 时已经为所有有效的 `jffs2_raw_inode` 数据实体创建了相应的 `jffs2_full_dnode`，并由 `jffs2_node_frag` 加入了红黑树，这个函数仅是通过红黑树访问相应的数据实体即可。

（早在打开文件、创建 inode 时就会跳过所有过时的数据实体，另外写入新的数据实体时会修改红黑树中的相应结点的 `node` 指向新数据实体的 `jffs2_node_frag`，那么红黑树中的结点始终指向有效数据实体。flash 上的数据实体一旦过时就再也不会被访问到，最终它所在的擦除块会被 GC 加入待擦除链表。）

第三个参数为缓冲区的内核虚拟地址，第四、五个参数描述文件的一个页中的一个区域。为了帮助理解这个函数的逻辑，可以参考下图所示的具有普遍意义的一种情形：



假设文件的一个页面内容由三部分数据组成：第一块数据的起始即为页面起始，第三块数据结束在页面结尾处，两部分数据之间存在一个空洞。三部分数据都由相应的 `jffs2_raw_inode` 数据实体描述（回想在写操作时 `address_space_operation` 的 `prepare_write` 方法向 flash 写入一个数据实体来描述空洞）。而需要读出的区域始于第一块数据中间、终于第三块数据中间。下面我们就以这种情形为例来分析这个函数的逻辑：

```
int jffs2_read_inode_range(struct jffs2_sb_info *c, struct jffs2_inode_info *f, unsigned char *buf,
                          uint32_t offset, uint32_t len)
```

```
{
    uint32_t end = offset + len;
    struct jffs2_node_frag *frag;
    int ret;
    D1(printk(KERN_DEBUG "jffs2_read_inode_range: ino %#u, range 0x%08x-0x%08x\n",
                f->inocache->ino, offset, offset+len));
```

```
    frag = jffs2_lookup_node_frag(&f->fragtree, offset);
```

由于在打开文件、创建 `inode` 时已经为 `jffs2_raw_inode` 数据实体创建了相应的 `jffs2_full_dnode`，并由 `jffs2_node_frag` 加入了红黑树，所以在读文件时就可以通过红黑树来定位相应的数据实体了。这里首先找到包含 `offset` 的数据实体，或者起始位置 `ofs` 大于 `offset`、但 `ofs` 又是最小的数据实体（`jffs2_lookup_node_frag` 涉及红黑树的查找，尚未详细分析，所以该函数的行为有待确认）。

```
/* XXX FIXME: Where a single physical node actually shows up in two
   frags, we read it twice. Don't do that. */
/* Now we're pointing at the first frag which overlaps our page */
```

由于待读出的区域可能涉及多个数据实体，所以在一个循环中可能需要分多次读出，其中每次循环只涉及一个数据实体中的数据（尽管一次循环待读出的数据可能只是一个数据实体的一部分，但是也要首先读出整个数据实体的内容。）每次循环后递增 `offset` 指针，并从红黑树中返回指向“后继”数据实体的 `frag`。

```
while(offset < end) {
    D2(printk(KERN_DEBUG "jffs2_read_inode_range: offset %d, end %d\n", offset, end));
    if (!frag || frag->ofs > offset) {
        uint32_t holesize = end - offset;
        if (frag) {
            D1(printk(KERN_NOTICE "Eep. Hole in ino %#u fraglist. frag->ofs = 0x%08x,
                          offset = 0x%08x\n", f->inocache->ino, frag->ofs, offset));
            holesize = min(holesize, frag->ofs - offset);
```

```

        D1(jffs2_print_frag_list(f));
    }
    D1(printk(KERN_DEBUG "Filling non-frag hole from %d-%d\n", offset, offset+holesize));
    memset(buf, 0, holesize);
    buf += holesize;
    offset += holesize;
    continue;

```

总结：在读取时碰到“空洞”就将buf中相对应的区域清0即可

在上一个数据实体读出后，frag 即指向其“后继”的数据实体。或者在第一次循环中 frag 指向第一个数据实体。如果该数据实体的数据在文件的相关页面内的逻辑偏移大于 offset，则说明在两次循环的两个数据实体之间存在一个空洞、或者在第一次循环、第一个数据实体前存在一个空洞。此时洞的大小为 ofs - offset。洞内的数据都为 0，所以直接将 buf 中相应长度的区域清 0 即可，同时步进 offset 和 buf 指针为洞的大小，并开始下一轮循环。

```

    } else if (frag->ofs < offset && (offset & (PAGE_CACHE_SIZE-1)) != 0) {
        D1(printk(KERN_NOTICE "Eep. Overlap in ino #%u fraglist. frag->ofs = 0x%08x,
                                offset = 0x%08x\n", f->inocache->ino, frag->ofs, offset));

        D1(jffs2_print_frag_list(f));
        memset(buf, 0, end - offset);
        return -EIO;
    }

```

我也想问

这不正是我假设的第一种情况吗，为什么是非法的呢？！

```

else if (!frag->node) {
    uint32_t holeend = min(end, frag->ofs + frag->size);
    D1(printk(KERN_DEBUG "Filling frag hole from %d-%d (frag 0x%x 0x%x)\n", offset, holeend,
                                frag->ofs, frag->ofs + frag->size));

    memset(buf, 0, holeend - offset);
    buf += holeend - offset;
    offset = holeend;
    frag = frag_next(frag);
    continue;
}

```

jffs2_node_frag 的 node 域指向数据实体的 jffs2_full_dnode 数据结构。在什么情况下会为 NULL？？这种情况也是洞，将读缓冲区中 buf 开始相应长度的区域清 0（另外在 jffs2_prepare_write 中写入洞的数据实体，创建相应内核描述符并加入红黑树）。

```

else {
    uint32_t readlen;
    uint32_t fragofs;          /* offset within the frag to start reading */
    fragofs = offset - frag->ofs;
    readlen = min(frag->size - fragofs, end - offset);

```

```
D1(printk(KERN_DEBUG "Reading %d-%d from node at 0x%x\n", frag->ofs+fragofs,
        frag->ofs+fragofs+readlen, ref_offset(frag->node->raw)));
```

fragofs 为数据起始位置在数据实体内的偏移。在我们假设的情形中，本次循环需要读取数据实体中除 fragofs 之外的数据，所以读出的数据量 readlen 等于数据实体大小 frag->size 减去 fragofs。

第四个参数为什么是这样？？ frag.ofs 和 jffs2_full_dnode.ofs 都是数据在文件内的偏移，相减为 0。那么第四个参数不就是数据在数据实体内部的偏移 fragofs 吗？？从 jffs2_read_dnode 函数分析看，第四个参数的确也是指起始读位置在数据实体内部的偏移。

这里相当于就是在该数据实体中(dnode中的)偏移量offset

```
ret = jffs2_read_dnode(c, frag->node, buf, fragofs + frag->ofs - frag->node->ofs, readlen);
D2(printk(KERN_DEBUG "node read done\n"));
if (ret) {
    D1(printk(KERN_DEBUG "jffs2_read_inode_range error %d\n", ret));
    memset(buf, 0, readlen);
    return ret;
}
buf += readlen;
offset += readlen;
frag = frag_next(frag);
D2(printk(KERN_DEBUG "node read was OK. Looping\n"));
continue;
}
printk(KERN_CRIT "dwmw2 is stupid. Reason #5325\n");
BUG();
} //while
return 0;
}
```

读出操作完成后步进 buf 和 offset 指针，frag_next 宏用于返回红黑树中“后继”数据实体的指针。然后开始新的循环。

jffs2_read_dnode函数

该函数读出 fd 所指数据实体内部[ofs, ofs + len]区域的数据到缓冲区 buf 中。即使可能只需要读出一个数据实体的一部分，但还是首先读出整个数据实体。而从 flash 上读取一个数据实体时分两步进行：首先读出 jffs2_raw_inode，从而获得 csize 和 dsize 信息，然后再分配合适的大小，再读出紧随其后的压缩了的数据，最后再解压缩。最后再将指定区域内的数据复制到 buf 中。

```
int jffs2_read_dnode(struct jffs2_sb_info *c, struct jffs2_full_dnode *fd, unsigned char *buf, int ofs, int len)
{
    struct jffs2_raw_inode *ri;
    size_t readlen;
    uint32_t crc;
```

```

unsigned char *decomprbuf = NULL;
unsigned char *readbuf = NULL;
int ret = 0;

ri = jffs2_alloc_raw_inode();
if (!ri)
    return -ENOMEM;
ret = jffs2_flash_read(c, ref_offset(fd->raw), sizeof(*ri), &readlen, (char *)ri);
if (ret) {
    jffs2_free_raw_inode(ri);
    printk(KERN_WARNING "Error reading node from 0x%08x: %d\n", ref_offset(fd->raw), ret);
    return ret;
}
if (readlen != sizeof(*ri)) {
    jffs2_free_raw_inode(ri);
    printk(KERN_WARNING "Short read from 0x%08x: wanted 0x%x bytes, got 0x%x\n",
        ref_offset(fd->raw), sizeof(*ri), readlen);
    return -EIO;
}
crc = crc32(0, ri, sizeof(*ri)-8);
D1(printk(KERN_DEBUG "Node read from %08x: node_crc %08x, calculated CRC %08x. dsize %x, csize
    %x, offset %x, buf %p\n", ref_offset(fd->raw), je32_to_cpu(ri->node_crc),
    crc, je32_to_cpu(ri->dsize), je32_to_cpu(ri->csize), je32_to_cpu(ri->offset), buf));
if (crc != je32_to_cpu(ri->node_crc)) {
    printk(KERN_WARNING "Node CRC %08x != calculated CRC %08x for node at %08x\n",
        je32_to_cpu(ri->node_crc), crc, ref_offset(fd->raw));
    ret = -EIO;
    goto out_ri;
}
/* There was a bug where we wrote hole nodes out with csize/dsize swapped. Deal with it */
if (ri->compr == JFFS2_COMPR_ZERO && !je32_to_cpu(ri->dsize) && je32_to_cpu(ri->csize)) {
    ri->dsize = ri->csize;
    ri->csize = cpu_to_je32(0);
}

```

如前所述，读取数据实体时首先要读出其 `jffs2_raw_inode` 结构本身的内容，以得到后继数据长度的信息 `csize` 和 `dsize`。首先用 `jffs2_alloc_raw_inode` 函数分配一个 `jffs2_raw_inode` 数据结构，然后由 `jffs2_flash_read` 函数填充之。如果实际读出的数据长度小于 `jffs2_raw_inode` 数据结构本身的长度、或者发生 `crc` 校验错误，则返回错误码 `EIO`。

这里的 `dsize` 不是数据的长度？

```

D1(if(ofs + len > je32_to_cpu(ri->dsize)) {
    printk(KERN_WARNING "jffs2_read_dnode() asked for %d bytes at %d from %d-byte node\n",
        len, ofs, je32_to_cpu(ri->dsize));
    ret = -EINVAL;
}

```

```

        goto out_ri;
    });
    if (ri->compr == JFFS2_COMPR_ZERO) {
        memset(buf, 0, len);
        goto out_ri;
    }
}

```

如果 `compr` 等于 `JFFS2_COMPR_ZERO`，则表示为一个洞，所以只需直接将读缓冲区中 `buf` 偏移开始、长度为 `len` 的空间清 0 即可。处理完洞，下面就根据读出的范围是否涵盖数据实体后继的所有数据、数据是否压缩，分为 4 种情况进行处理：

```

/* Cases:
   Reading whole node and it's uncompressed - read directly to buffer provided, check CRC.
   Reading whole node and it's compressed - read into comprbuf, check CRC and decompress
                                           to buffer provided
   Reading partial node and it's uncompressed - read into readbuf, check CRC, and copy
   Reading partial node and it's compressed - read into readbuf, check checksum,
                                           decompress to decomprbuf and copy */

if (ri->compr == JFFS2_COMPR_NONE && len == je32_to_cpu(ri->dsize)) {
    readbuf = buf;
}

```

第一种情况，如果需要读出后继所有数据，并且数据没有经过压缩，则直接将数据读入到接收缓冲区 `buf` 种即可（而无需通过解压缩缓冲区中转）。

只要不是读出所有的数据，就需要经过中间的缓冲区中转，即使数据也没有被压缩，所以下面根据数据实体的长度 `csize` 分配中间缓冲区：（如果没有压缩，则 `csize` 等于 `dsize`）

```

} else {
    readbuf = kmalloc(je32_to_cpu(ri->csize), GFP_KERNEL);
    if (!readbuf) {
        ret = -ENOMEM;
        goto out_ri;
    }
}

if (ri->compr != JFFS2_COMPR_NONE) {
    if (len < je32_to_cpu(ri->dsize)) { //读出部分压缩数据，需要额外解压缩缓冲区
        decomprbuf = kmalloc(je32_to_cpu(ri->dsize), GFP_KERNEL);
        if (!decomprbuf) {
            ret = -ENOMEM;
            goto out_readbuf;
        }
    } else {
        decomprbuf = buf; //读出全部压缩数据，直接将接收缓冲区当作解压缩缓冲区即可
    }
}

```

```

    }
} else { //读出部分、未压缩的数据
    decomprbuf = readbuf;
}

```

`compr` 不等于 `JFFS2_COMPR_NONE`, 则说明数据被压缩。如果需要读出部分被压缩的数据, 那么还需要另一个容纳解压缩了的数据的缓冲区, 然后再截取其中的数据到接收缓冲区 `buf` 中; 如果要读出全部压缩数据, 则无需截取操作, 所以可以直接将数据解压缩到接收缓冲区 `buf` 中即可; 如果数据没有被压缩, 则首先将全部数据读出到中间缓冲区 `readbuf` 中, 随后在截取其中相应范围的数据到接收缓冲区 `buf` 中。下面通过 `jffs2_flash_read` 函数将**所有**后继数据读到 `readbuf` 中, 并进行 `crc` 校验。

```

D2(printk(KERN_DEBUG "Read %d bytes to %p\n", je32_to_cpu(ri->csize),
    readbuf));
ret = jffs2_flash_read(c, (ref_offset(fd->raw)) + sizeof(*ri), je32_to_cpu(ri->csize), &readlen, readbuf);
if (!ret && readlen != je32_to_cpu(ri->csize))
    ret = -EIO;
if (ret)
    goto out_decomprbuf;
crc = crc32(0, readbuf, je32_to_cpu(ri->csize));
if (crc != je32_to_cpu(ri->data_crc)) {
    printk(KERN_WARNING "Data CRC %08x != calculated CRC %08x for node at %08x\n",
        je32_to_cpu(ri->data_crc), crc, ref_offset(fd->raw));
    ret = -EIO;
    goto out_decomprbuf;
}

```

数据读出后, 如果是压缩了的数据, 则进行解压缩到 `decomprbuf` 中; 如果需要读出的仅是其中的部分数据, 那么还要截取这部分数据:

```

D2(printk(KERN_DEBUG "Data CRC matches calculated CRC %08x\n", crc));
if (ri->compr != JFFS2_COMPR_NONE) {
    D2(printk(KERN_DEBUG "Decompress %d bytes from %p to %d bytes at %p\n", ri->csize, readbuf,
        je32_to_cpu(ri->dsiz), decomprbuf));
    ret = jffs2_decompress(ri->compr, readbuf, decomprbuf, je32_to_cpu(ri->csize),
        je32_to_cpu(ri->dsiz));
    if (ret) {
        printk(KERN_WARNING "Error: jffs2_decompress returned %d\n", ret);
        goto out_decomprbuf;
    }
}
if (len < je32_to_cpu(ri->dsiz)) {
    memcpy(buf, decomprbuf+ofs, len);
}

```

```
out_decomprbuf:
    if(decomprbuf != buf && decomprbuf != readbuf)
        kfree(decomprbuf);
out_readbuf:
    if(readbuf != buf)
        kfree(readbuf);
out_ri:
    jffs2_free_raw_inode(ri);
    return ret;
}
```

第 8 章 jffs2 中符号链接文件的方法表（new）

在 jffs2_read_inode 函数中创建 inode 的最后，会根据文件的类型将 i_op 等指针设置为具体类型文件的方法表。符号链接的方法表为 jffs2_symlink_inode_operations，定义于 fs/jffs2/symlink.c：

```
struct inode_operations jffs2_symlink_inode_operations =
{
    .readlink =      jffs2_readlink,
    .follow_link =   jffs2_follow_link,
    .setattr =       jffs2_setattr
};
```

在 path_walk 函数中逐层解析路径名时，如果当前路径名分量对应的方法表中的 follow_link 指针不为空，则通过 do_follow_link 函数调用符号链接文件的 follow_link 方法“跳转”到真正被链接的目标文件。

由于索引结点号只在当前文件系统内惟一，所以硬链接的目标只能在同文件系统内。与硬链接不同，符号链接的对象可以在其它文件系统中。这是因为符号链接文件的本质类似正规文件，其数据即为被链接的文件名，那么从根文件系统的根目录出发总是可以到达被链接文件的。

由于符号链接文件的目标可能在另外一个文件系统中，所以可想而知在调用具体文件系统的 follow_link 方法跟踪符号链接文件时是一定会首先回到高层 VFS 的框架代码的，然后从那里再通过 path_walk 进入其它的文件系统。在 ext2 中符号链接文件的数据，即被链接的文件名保存在 ext2_inode 的 i_block[] 数组中（和 ext2_inode_info 的 i_data[] 数组中），而 jffs2 中被链接文件名保存在其惟一的 jffs2_raw_inode 数据结点后。不同文件系统只需得到被链接文件名，然后就可以借助 vfs_follow_link 函数来实现各自的 follow_link 方法了。

有关 path_walk 和 do_follow_link 以及 vfs_follow_link 函数的细节可以参见情景分析。

jffs2_follow_link 函数

在 do_follow_link 函数中调用 jffs2_follow_link 函数时传递的第一个参数为指向符号链接本身的 dentry，第二个参数 nd 用于保存后继路径名解析的结果。

```
int jffs2_follow_link(struct dentry *dentry, struct nameidata *nd)
{
    unsigned char *buf;
    int ret;

    buf = jffs2_getlink(JFFS2_SB_INFO(dentry->d_inode->i_sb), JFFS2_INODE_INFO(dentry->d_inode));
    if (IS_ERR(buf))
        return PTR_ERR(buf);
```



```

    ret = vfs_follow_link(nd, buf);
    kfree(buf);
    return ret;
}

```

如前所述，在 jffs2 中只需首先获得被链接文件名，然后就可以直接调用 `vfs_follow_link` 方法了。被链接文件名在其惟一的 `jffs2_raw_inode` 数据实体后，而与之相关的上层 `jffs2_full_dnode` 数据结构由符号链接文件的 `jffs2_inode_info` 的 `metadata` 域直接指向。所以，只需从符号链接文件的 `inode` 出发就可以得到其惟一 `jffs2_raw_inode` 数据实体的内核描述符了，进而读出被链接文件名。这些操作由 `jffs2_getlink` 函数完成。

jffs2_getlink函数

```

/* Core function to read symlink target. */
char *jffs2_getlink(struct jffs2_sb_info *c, struct jffs2_inode_info *f)
{
    char *buf;
    int ret;

    down(&f->sem);
    if (!f->metadata) {
        printk(KERN_NOTICE "No metadata for symlink inode #%u\n", f->inocache->ino);
        up(&f->sem);
        return ERR_PTR(-EINVAL);
    }
    buf = kmalloc(f->metadata->size+1, GFP_USER);
    if (!buf) {
        up(&f->sem);
        return ERR_PTR(-ENOMEM);
    }
    buf[f->metadata->size]=0;

```

符号链接、目录文件和设备文件的惟一的 `jffs2_raw_inode` 的上层数据结构 `jffs2_full_dnode` 由其 `inode` 的 `u` 域，即 `jffs2_full_dnode` 的 `metadata` 域直接指向（参见图 1）。如果该域为空则出错。根据 `jffs2_full_dnode` 中记录的 `jffs2_raw_inode` 的后继数据的大小分配合适的空间，然后就可以通过 [jffs2_read_dnode函数](#) 读出被链接文件名了。注意倒数第二个参数指定待读出数据在相关文件内的逻辑偏移，最后一个参数指明长度：

```

    ret = jffs2_read_dnode(c, f->metadata, buf, 0, f->metadata->size);
    up(&f->sem);
    if (ret) {
        kfree(buf);
        return ERR_PTR(ret);
    }
    return buf;
}

```

第9章 jffs2 中目录文件的方法表（new）

在 `jffs2_read_inode` 函数中创建 `inode` 的最后，会根据文件的类型将 `i_op` 等指针设置为具体类型文件的方法表。目录文件的方法表为 `jffs2_dir_inode_operations`，定义于 `fs/jffs2/dir.c`：

```
struct inode_operations jffs2_dir_inode_operations =
{
    .create =      jffs2_create,
    .lookup =      jffs2_lookup,
    .link =        jffs2_link,
    .unlink =      jffs2_unlink,
    .symlink =     jffs2_symlink,
    .mkdir =       jffs2_mkdir,
    .rmdir =       jffs2_rmdir,
    .mknod =       jffs2_mknod,
    .rename =      jffs2_rename,
    .setattr =     jffs2_setattr,
};
```

这个方法表提供了在一个目录下创建、删除各种类型文件的方法，下面我们分析用于创建正规文件的 `jffs2_create` 方法。

jffs2_create函数

在 `open` 操作中用户可以通过 `O_CREATE` 标志指定当相关文件不存在时创建它。注意 `open` 函数只能创建正规文件。此时函数调用关系为：

`sys_open > filp_open > open_namei > vfs_create > i_op->create`（即 `jffs2_create` 函数）

各层函数的细节可参见情景分析。在新建正规文件时需要创建其 `inode`、向父目录文件中写入其目录项，既然是打开文件则还要创建其 `file`、`dentry`。创建 `dentry` 的工作在 `open_namei` 函数中调用 `vfs_create` 之前，由 `lookup_hash` 函数完成。该函数从 `dentry_cache` 中分配一个新的 `dentry` 数据结构，其 `d_name.name` 指向文件名，并用 `d_parent`、`d_child` 加入系统目录树。此时的 `dentry` 为“负”的，因为 `d_inode` 指针为 `NULL`。然后调用 `jffs2_lookup` 函数尝试从父目录的 `jffs2_inode_info.dents` 队列中查找指定子文件的 `jffs2_full_dirent` 结构（此时当然失败）。

创建 `inode`、向其父目录文件写入相应目录项的工作是由 `vfs_create` 函数完成的，另外就 `jffs2` 而言创建文件时还必须创建相应的内核描述符，并初始化好“代表文件存在”的第一个 `jffs2_raw_inode` 数据实体（该数据实体为“metadata”，仅表文件的存在，由 `jffs2_inode_info.metadata` 直接指向，在第一次写操作时会被设置为过时的，参见正规文件的写操作 `jffs2_write_inode_range` 函数的[相关部分](#)）。`vfs_create` 函数在持有父目录的 `i_zombie` 信号量的情况下调用父目录文件的 `create` 方法，即 `jffs2_create` 函数完成所有具体操作。其第一个

参数指向父目录inode，第二个参数指向已经为新文件创建的dentry（注意文件名已设置好，并已加入文件系统目录树），第三个参数为用户指定的文件访问权限。

```
static int jffs2_create(struct inode *dir_i, struct dentry *dentry, int mode)
{
    struct jffs2_raw_inode *ri;
    struct jffs2_inode_info *f, *dir_f;
    struct jffs2_sb_info *c;
    struct inode *inode;
    int ret;

    ri = jffs2_alloc_raw_inode();
    if (!ri)
        return -ENOMEM;

    c = JFFS2_SB_INFO(dir_i->i_sb);
    D1(printk(KERN_DEBUG "jffs2_create()\n"));

    inode = jffs2_new_inode(dir_i, mode, ri);
    if (IS_ERR(inode)) {
        D1(printk(KERN_DEBUG "jffs2_new_inode() failed\n"));
        jffs2_free_raw_inode(ri);
        return PTR_ERR(inode);
    }
}
```

首先通过 `jffs2_alloc_raw_inode` 函数从 `raw_inode_slab` 高速缓存中分配一个空白的 `jffs2_raw_inode` 数据实体，然后调用 `jffs2_new_inode` 函数完成如下工作：

- 1, 分配、初始化文件的内核描述符 `jffs2_inode_cache` 数据结构；
- 2, 设置 `jffs2_raw_inode` 数据实体；
- 3, 分配、初始化 `inode`；
- 4, 建立 `jffs2_inode_cache` 和 `inode` 的联系，注册 `inode`。

文件的内核描述符实现了文件及其数据实体之间的映射机制，所以早在挂载 `jffs2` 时就已经为设备上的所有文件创建了内核描述符，在新建文件时也必须首先建立其内核描述符。详见下文。

```
inode->i_op = &jffs2_file_inode_operations;
inode->i_fop = &jffs2_file_operations;
inode->i_mapping->a_ops = &jffs2_file_address_operations;
inode->i_mapping->nrpages = 0;
```

接下来就要设置文件的方法表指针了。由于这里创建的是正规文件，所以指针都指向正规文件的相关方法表（这里与 `jffs2_read_inode` 函数的相关代码相同）。

至此，新建正规文件时剩余的工作就是向其父目录中写入相应的目录项 `jffs2_raw_dirent` 数据实体、向 flash 写入“代表其存在”的第一个 `jffs2_raw_inode` 数据实体，以及创建上层的 `jffs2_full_dirent` 和 `jffs2_full_dnode`。

这些工作都由jffs2_do_create函数完成，参见[下文](#)。

```
f = JFFS2_INODE_INFO(inode);
dir_f = JFFS2_INODE_INFO(dir_i);
ret = jffs2_do_create(c, dir_f, f, ri, dentry->d_name.name, dentry->d_name.len);
if (ret) {
    jffs2_clear_inode(inode);
    make_bad_inode(inode);
    iput(inode);
    jffs2_free_raw_inode(ri);
    return ret;
}
```

最后，更新父目录 inode 的相应时间戳为数据实体中的创建时间。最后释放数据实体，并调用 d_instantiate 函数用 d_alias 和 d_inode 域建立 dentry 和 inode 之间的联系。

```
dir_i->i_mtime = dir_i->i_ctime = je32_to_cpu(ri->ctime);

jffs2_free_raw_inode(ri);
d_instantiate(dentry, inode);

D1(printk(KERN_DEBUG "jffs2_create: Created ino %%lu with mode %o, nlink %d(%d). nrpages %ld\n",
    inode->i_ino, inode->i_mode, inode->i_nlink, f->inocache->nlink, inode->i_mapping->nrpages));
return 0;
}
```

jffs2_new_inode函数

该函数创建、初始化新文件的内核描述符 jffs2_inode_cache，初始化文件的 jffs2_raw_inode 数据实体，并创建、设置 inode，最后建立 jffs2_inode_cache 和 inode 的联系，并将 inode 注册到内核相关数据结构中。

```
/* jffs2_new_inode: allocate a new inode and inocache, add it to the hash,
   fill in the raw_inode while you're at it. */
struct inode *jffs2_new_inode (struct inode *dir_i, int mode, struct jffs2_raw_inode *ri)
{
    struct inode *inode;
    struct super_block *sb = dir_i->i_sb;
    struct jffs2_sb_info *c;
    struct jffs2_inode_info *f;
    int ret;

    D1(printk(KERN_DEBUG "jffs2_new_inode(): dir_i %ld, mode 0x%x\n", dir_i->i_ino, mode));

    c = JFFS2_SB_INFO(sb);
```

```

inode = new_inode(sb);
if (!inode)
    return ERR_PTR(-ENOMEM);

```

首先通过定义于 `linux/fs.h` 中的内联函数 `new_inode` 分配一个空白的 `inode`，这个操作又是通过 `get_empty_inode` 函数完成的，它将空白的 `inode` 加入内核的 `inode_in_use` 队列：

```

static inline struct inode * new_inode(struct super_block *sb)
{
    struct inode *inode = get_empty_inode();
    if (inode) {
        inode->i_sb = sb;
        inode->i_dev = sb->s_dev;
        inode->i_blkbits = sb->s_blocksize_bits;
    }
    return inode;
}

f = JFFS2_INODE_INFO(inode);
jffs2_init_inode_info(f);
memset(ri, 0, sizeof(*ri));

```

jffs2 中将 `inode` 的 `u` 域解释为 `jffs2_inode_info` 数据结构，然后其和由参数传递的 `jffs2_raw_inode` 清空。

```

/* Set OS-specific defaults for new inodes */
ri->uid = cpu_to_jel6(current->fsuid);
if (dir_i->i_mode & S_ISGID) {
    ri->gid = cpu_to_jel6(dir_i->i_gid);
    if (S_ISDIR(mode))
        mode |= S_ISGID;
} else {
    ri->gid = cpu_to_jel6(current->fsgid);
}

```

设置新文件的 `uid` 为当前进程实际使用的 `uid`，即 `fsuid`。如果父目录的 `S_ISGID` 标志有效，则新文件继承父目录的 `gid`，否则设置为当前进程实际使用的 `gid`，即 `fsgid`。

```

ri->mode = cpu_to_je32(mode);
ret = jffs2_do_new_inode(c, f, mode, ri);
if (ret) {
    make_bad_inode(inode);
    iput(inode);
    return ERR_PTR(ret);
}

```

```
}
```

文件的内核描述符 `jffs2_inode_cache` 建立了文件及其数据之间的索引机制，在向新建文件写入任何数据实体前必须首先创建好其内核描述符。这个工作通过 `jffs2_do_new_inode` 函数完成，包括设置索引结点号、初始化数据实体描述符的 `nodes` 链表，建立文件描述符与文件 `inode` 的联系并加入超级块的 `inocache_list[]` 哈希表，并进一步设置 `jffs2_raw_inode` 中的相应域，参见[下文](#)。

```
inode->i_nlink = 1;
inode->i_ino = je32_to_cpu(ri->ino);
inode->i_mode = je32_to_cpu(ri->mode);
inode->i_gid = je16_to_cpu(ri->gid);
inode->i_uid = je16_to_cpu(ri->uid);
inode->i_atime = inode->i_ctime = inode->i_mtime = CURRENT_TIME;
ri->atime = ri->mtime = ri->ctime = cpu_to_je32(inode->i_mtime);
inode->i_blksize = PAGE_SIZE;
inode->i_blocks = 0;
inode->i_size = 0;
insert_inode_hash(inode);
return inode;
}
```

设置好新建文件的 `jffs2_raw_inode` 数据实体后，就可以根据其中的信息来设置 `inode` 的相应域了，最后通过 `insert_inode_hash` 函数利用 `inode.i_hash` 域将其加入索引节点哈希表 `inode_hashtable` 的某个队列中。

进阶 重要

jffs2_do_create 函数

在创建正规文件时需要向其父目录写入相应目录项 `jffs2_raw_dirent` 数据实体，同时写入“代表其存在”的第一个 `jffs2_raw_inode` 数据结构，创建其内核描述符 `jffs2_raw_node_ref` 以及上层的 `jffs2_full_dirent` 和 `jffs2_full_dnode` 数据结构。注意此时写入的第一个 `jffs2_raw_inode` 数据结构没有携带任何有效数据，其 `jffs2_full_dnode` 也被 `jffs2_inode_info` 的 `metadata` 域直接指向而没有通过 `jffs2_node_frag` 组织到 `fragtree` 红黑树中。详见下文。

该函数的第 2、3 个参数指向父目录文件、其自身的 `inode`，第 4 个参数指向在 `jffs2_create` 函数中已经设置好的 `jffs2_raw_inode` 数据实体，最后两个参数描述新文件名字符串。

```
int jffs2_do_create(struct jffs2_sb_info *c, struct jffs2_inode_info *dir_f, struct jffs2_inode_info *f,
                   struct jffs2_raw_inode *ri, const char *name, int namelen)
{
    struct jffs2_raw_dirent *rd;
    struct jffs2_full_dnode *fn;
    struct jffs2_full_dirent *fd;
    uint32_t alloclen, phys_ofs;
    uint32_t writtenlen;
    int ret;
```

```

/* Try to reserve enough space for both node and dirent. Just the node will do for now, though */
ret = jffs2_reserve_space(c, sizeof(*ri), &phys_ofs, &alloclen, ALLOC_NORMAL);
D1(printk(KERN_DEBUG "jffs2_do_create(): reserved 0x%x bytes\n", alloclen));
if (ret) {
    up(&f->sem);
    return ret;
}

```

首先通过jffs2_reserve_space函数在flash上分配一个合适大小的空间，其物理地址由phys_ofs参数返回，长度由alloclen参数返回。然后就可以通过[jffs2_write_dnode函数](#)向flash写入“代表文件存在”的jffs2_raw_inode数据实体，并创建相应内核描述符jffs2_raw_node_ref和上层jffs2_full_dnode数据结构了。

```

ri->data_crc = cpu_to_je32(0);
ri->node_crc = cpu_to_je32(crc32(0, ri, sizeof(*ri)-8));          回顾：write_dnode是先写raw_inode，后紧跟其后写一段数据
fn = jffs2_write_dnode(c, f, ri, NULL, 0, phys_ofs, &writtenlen);
D1(printk(KERN_DEBUG "jffs2_do_create created file with mode 0x%x\n",
    je32_to_cpu(ri->mode)));
if (IS_ERR(fn)) {
    D1(printk(KERN_DEBUG "jffs2_write_dnode() failed\n"));
    /* Eeek. Wave bye bye */
    up(&f->sem);
    jffs2_complete_reservation(c);
    return PTR_ERR(fn);
}
/* No data here. Only a metadata node, which will be obsoleted by the first data write */
f->metadata = fn;

```

注意调用jffs2_write_dnode函数时传递的倒数第 3 个参数为 0，因为该数据实体并没有携带任何数据。另外其上层jffs2_full_dnode数据结构由jffs2_inode_info的metadata域直接指向，并且在第一次真正写入文件时被标记为过时的。可参见正规文件的写操作jffs2_write_inode_range函数的[相关部分](#)，以及jffs2map2 观察到的[结果](#)。

```

/* Work out where to put the dirent node now. */
writtenlen = PAD(writtenlen);
phys_ofs += writtenlen;
alloclen -= writtenlen;
up(&f->sem);

```

写完了“代表文件存在”的jffs2_raw_inode数据实体后，接下来就要向父目录文件写入其目录项了。由此可见目录项jffs2_raw_dirent数据实体是“紧接着”刚才的jffs2_raw_inode数据实体写入的。如果先前从flash上分配的空间不足，则再次分配连续的空间：

```

if (alloclen < sizeof(*rd)+namelen) {

```

```

/* Not enough space left in this chunk. Get some more */
jffs2_complete_reservation(c);
ret = jffs2_reserve_space(c, sizeof(*rd)+namelen, &phys_ofs, &alloclen, ALLOC_NORMAL);
if (ret) {
    /* Eep. */
    D1(printk(KERN_DEBUG "jffs2_reserve_space() for dirent failed\n"));
    return ret;
}
}

```

接下来，通过 `jffs2_alloc_raw_dirent` 函数从 `raw_dirent_slab` 高速缓存中分配一个空白的 `jffs2_raw_dirent`，并初始化：

```

rd = jffs2_alloc_raw_dirent();
if (!rd) {
    /* Argh. Now we treat it like a normal delete */
    jffs2_complete_reservation(c);
    return -ENOMEM;
}
down(&dir_f->sem);
rd->magic = cpu_to_je16(JFFS2_MAGIC_BITMASK);
rd->nodetype = cpu_to_je16(JFFS2_NODETYPE_DIRENT);
rd->totlen = cpu_to_je32(sizeof(*rd) + namelen);
rd->hdr_crc = cpu_to_je32(crc32(0, rd, sizeof(struct jffs2_unknown_node)-4));

```

注意 `jffs2_raw_dirent` 数据结点的类型当然为 `JFFS2_NODETYPE_DIRENT`，紧随其后的为新建文件的文件名。

```

rd->pino = cpu_to_je32(dir_f->inocache->ino);
rd->version = cpu_to_je32(++dir_f->highest_version);
rd->ino = ri->ino;
rd->mctime = ri->ctime;
rd->nsize = namelen;
rd->type = DT_REG;
rd->node_crc = cpu_to_je32(crc32(0, rd, sizeof(*rd)-8));
rd->name_crc = cpu_to_je32(crc32(0, name, namelen));

```

参数 `dir_f` 指向父目录文件的 `jffs2_inode_info`。由于 `jffs2` 中组成目录文件的每个目录项不连续，缺乏类似 `ext2` 中通过目录文件的索引结点就可以索引到所有目录项的机制，所以每个目录项除记录自己所代表的文件的索引结点号外，还得用额外的 `pino` 域记录自己所属的目录文件的索引节点号。另外 `type` 域表明目录项所对应的文件的类型，这里为正规文件类型 `DT_REG`。

```

fd = jffs2_write_dirent(c, dir_f, rd, name, namelen, phys_ofs, &writtenlen);
jffs2_free_raw_dirent(rd);

```



```

if (IS_ERR(fd)) {
    /* dirent failed to write. Delete the inode normally
       as if it were the final unlink() */
    jffs2_complete_reservation(c);
    up(&dir_f->sem);
    return PTR_ERR(fd);
}
/* Link the fd into the inode's list, obsoleting an old one if necessary. */
jffs2_add_fd_to_list(c, fd, &dir_f->dents);
jffs2_complete_reservation(c);
up(&dir_f->sem);

return 0;
}

```

设置好目录项 `jffs2_raw_dirent` 数据实体后，就可以通过 `jffs2_write_dirent` 函数将其写入 flash 了，同时创建相应的内核描述符 `jffs2_raw_node_ref` 及上层的 `jffs2_full_dirent` 数据结构，最后通过 [jffs2_add_fd_to_list](#) 函数将其加入父目录文件 `jffs2_inode_info` 的 `dents` 队列。`jffs2_write_dirent` 函数与 `jffs2_write_dnode` 函数很类似，请读者自行阅读。

jffs2_do_new_inode 函数

该函数分配、初始化文件的内核描述符 `jffs2_inode_cache`（包括设置索引结点号、初始化数据实体描述符链表），建立其与文件 `inode` 的联系并加入超级块的 `inocache_list[]` 哈希表，并进一步设置 `jffs2_raw_inode` 的相应域。

```

int jffs2_do_new_inode(struct jffs2_sb_info *c, struct jffs2_inode_info *f,
                      uint32_t mode, struct jffs2_raw_inode *ri)
{
    struct jffs2_inode_cache *ic;

    ic = jffs2_alloc_inode_cache();
    if (!ic) {
        return -ENOMEM;
    }
    memset(ic, 0, sizeof(*ic));

```

首先通过 `jffs2_alloc_inode_cache` 函数从 `inode_cache_slab` 高速缓存中分配一个文件的内核描述符 `jffs2_inode_cache` 数据结构，并清空。

```

    init_MUTEX_LOCKED(&f->sem);
    f->inocache = ic;
    f->inocache->nlink = 1;
    f->inocache->nodes = (struct jffs2_raw_node_ref *)f->inocache;

```

```
f->inocache->ino = ++c->highest_ino;
f->inocache->state = INO_STATE_PRESENT;
```

然后将文件 inode 的 u 域，即 jffs2_inode_info 的 inocache 指向其内核描述符，并完成文件内核描述符的初始化：硬链接计数初始值为 1，索引节点号等于文件系统超级块的 u 域即 jffs2_sb_info 的 hisgest_ino，文件的状态为 **PRESENT**。文件内核描述符的重要作用就是建立文件及其数据实体之间的索引，由于新增文件此时尚未写入任何 jffs2_raw_inode 数据实体，所以描述符的 nodes 域指向其自身。

在 ext2 中文件数据所在的磁盘块由其 ext2_inode 进行索引，而 ext2_inode 在块组的索引节点表里，因此索引节点号就由其所处物理位置决定。为了提高访问效率，一般从其父目录所在块组中分配新建文件的 ext2_inode。如果新建目录文件，则还要考虑均衡每个块组内所含目录的数目。详情可以参考情景分析上册 P568-P571。

但是对 jffs2 而言，在 flash 中并不存在“索引”文件数据实体的“索引节点”，而且 flash 上也没有块组的概念，所以索引节点号的分配策略就简单得多：由于索引节点号在这个文件系统内惟一，所以就用 jffs2_sb_info 的 highest_ino 来保存下一个新建文件的索引节点号，并逐一递增。

```
ri->ino = cpu_to_je32(f->inocache->ino);
D1(printk(KERN_DEBUG "jffs2_do_new_inode(): Assigned ino# %d\n", f->inocache->ino));
```

```
jffs2_add_ino_cache(c, f->inocache);
```

```
ri->magic = cpu_to_je16(JFFS2_MAGIC_BITMASK);
ri->nodetype = cpu_to_je16(JFFS2_NODETYPE_INODE);
ri->totlen = cpu_to_je32(PAD(sizeof(*ri)));
ri->hdr_crc = cpu_to_je32(crc32(0, ri, sizeof(struct jffs2_unknown_node)-4));
ri->mode = cpu_to_je32(mode);
```

```
f->highest_version = 1;
ri->version = cpu_to_je32(f->highest_version);
```

```
return 0;
```

```
}
```

最后，还要通过 jffs2_add_ino_cache 函数将该文件描述符加入 jffs2_sb_info.inocache_list[] 哈希表，并继续设置 jffs2_raw_inode 中的相应域，注意设置 jffs2_raw_inode 数据实体的类型为 JFFS2_NODETYPE_INODE。

第 10 章 jffs2 的 Garbage Collection

在挂载 jffs2 文件系统时、在 jffs2_do_fill_super 函数的最后创建并启动 GC 内核线程，相关代码如下：

```
if (!(sb->s_flags & MS_RDONLY))
    jffs2_start_garbage_collect_thread(c);
return 0;
```

如果 jffs2 文件系统不是以只读方式挂载的，就会有新的数据实体写入 flash。而且 jffs2 文件系统的特点是在写入新的数据实体时并不修改 flash 上原有数据实体，而只是将其内核描述符标记为“过时”。系统运行一段时间后若空白 flash 擦除块的数量小于一定阈值，则 GC 被唤醒用于释放所有过时的数据实体。

为了尽量均衡地使用 flash 分区上的所有擦除块，在选择有效数据实体的副本所写入的擦除块时需要考虑 Wear Levelling 算法。

jffs2_start_garbage_collect_thread 函数

该函数用于创建 GC 内核线程。

/* This must only ever be called when no GC thread is currently running */

int jffs2_start_garbage_collect_thread(struct jffs2_sb_info *c)

```
{
    pid_t pid;
    int ret = 0;
    if (c->gc_task)
        BUG();
    init_MUTEX_LOCKED(&c->gc_thread_start); // 以locked初始化这个semaphore
    init_completion(&c->gc_thread_exit);
    pid = kernel_thread(jffs2_garbage_collect_thread, c, CLONE_FS|CLONE_FILES);
    if (pid < 0) {
        printk(KERN_WARNING "fork failed for JFFS2 garbage collect thread: %d\n", -pid);
        complete(&c->gc_thread_exit);
        ret = pid;
    } else {
        /* Wait for it... */
        D1(printk(KERN_DEBUG "JFFS2: Garbage collect thread is pid %d\n", pid));
        down(&c->gc_thread_start); // 这里直接尝试获取这个semaphore，必定会被阻塞。必须等jffs2_garbage_collect_thread
        // 执行到up函数才可使down函数执行完。
    }
    return ret;
}
```

信号量 `gc_thread_start` 用于保证在当前执行流在创建了 GC 内核线程后、在返回到当前执行流时 GC 内核线程已经运行了。`kernel_thread` 函数创建 GC 内核线程，此时 GC 内核线程与当前执行流谁获得 cpu 还不一定，于是当前执行流在获得 `gc_thread_start` 信号量时会阻塞（先前通过 `init_MUTEX_LOCKED` 宏已将信号量初始化为“不可用”状态），直到 GC 内核线程第一次获得运行后释放该信号量，参见下文。

传递给 `kernel_thread` 函数的第一个参数为新的内核线程所执行的代码，所以 GC 内核线程的行为由函数 `jffs2_garbage_collect_thread` 确定：

jffs2_garbage_collect_thread函数

```
static int jffs2_garbage_collect_thread(void *_c)
{
    struct jffs2_sb_info *c = _c;

    daemonize();
    c->gc_task = current;
    up(&c->gc_thread_start);
    sprintf(current->comm, "jffs2_gcd_mtd%d", c->mtd->index);
```

既然是运行于后台的内核线程，首先就得通过 `daemonize` 函数释放从其父进程获得的一些资源，比如关闭已经打开的控制台设备文件描述符、释放所有的属于用户空间的页面（如果存在的话）、并改换门庭投靠到 `init` 门下等等，其详细分析可参见情景分析。[daemonize的作用应该就是使这个线程脱离原进程，成为独立的线程。避免原进程结束后，子进程也结束。](#)

用 `jffs2_sb_info.gc_task` 来指向 GC 内核线程的 PCB，这样就不用为它创建额外的等待队列了：当 GC 内核线程无事可作时不用加入等待队列，只需从运行队列中删除即可；而在需要唤醒 GC 内核线程时可以通过 `gc_task` 指向通过 `send_sig` 函数向它发送 `SIGHUP` 信号。（类似的还有在 `schedule_timeout` 函数中也是用定时器数据结构 `timer_list` 的 `data` 域指向当前进程的 PCB，而没有使用额外的等到队列，当定时器到时时通过 `wake_up_process` 直接唤醒 `data` 所指向的进程即可）[就是说GC线程直达天听，不需要队列（因为只有一个GC线程？）](#)

释放 `gc_thread_start` 信号量，这将唤醒先前创建 GC 内核线程的执行流。然后给 GC 内核线程起个名字“`jffs2_gcd_mtd#`”，其中最后的数字为 jffs2 文件系统所在的 flash 分区的编号（从 0 开始），比如在我的系统中这个编号为 5（整个 flash 上有 6 个分区：uboot 映象、uboot 参数、分区表、内核映象、work 分区、root 分区）。

通过 `set_user_nice` 函数设置 GC 内核线程的静态优先级为 10 后，就进入它的主体循环了：

```
set_user_nice(current, 10);
for (;;) {
    spin_lock_irq(&current->sigmask_lock);
    siginitsetinv (&current->blocked, sigmask(SIGHUP) | sigmask(SIGKILL) | sigmask(SIGSTOP) |
                  sigmask(SIGCONT));
    recalc_sigpending();
    spin_unlock_irq(&current->sigmask_lock);
```

信号的编号与体系结构相关，定义于 `asm/signal.h`。宏 `sigmask` 返回信号的位索引：


```
#define sigmask(sig) (1UL << ((sig) - 1))
```

这是因为在 Linux 上没有编号为 0 的信号。

GC 内核线程的状态由 `SIGHUP`、`SIGKILL`、`SIGSTOP` 和 `SIGCONT` 四种信号控制，所以每次循环的开始都要做相应的准备工作，以便在进入 `TASK_INTERRUPTIBLE` 状态后接收：由内联函数 `siginitsetinv` 清除其信号屏蔽字 `blocked` 中这四种信号的相应位，使得 GC 内核线程只接收这些信号。`recalc_sigpending` 函数检查当前进程是否有非阻塞的挂起信号，如果有，则设置 PCB 中的 `sigpending` 标志。这个标志由下面的 `signal_pending` 函数检查。非阻塞的挂起信号，非阻塞就是没有被屏蔽，挂起就是还未被处理？

（比较：用户进程的信号处理函数在用户态执行，用户进程不必关心接收信号的细节，只注册信号处理函数。而内核线程运行于内核态，要由内核线程自己来完成信号的接收工作。）

```
if (!thread_should_wake(c)) {
    set_current_state(TASK_INTERRUPTIBLE); 设置为了interruptible后才可被中断，因此这里可能会错失应该被
                                           唤醒的条件
    D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread sleeping...\n"));
    /* Yes, there's a race here; we checked thread_should_wake() before
       setting current->state to TASK_INTERRUPTIBLE. But it doesn't
       matter - We don't care if we miss a wakeup, because the GC thread
       is only an optimisation anyway. */
    schedule();
}
```



每次循环的开始都要判断 GC 内核线程是否真的需要运行。若不需要，那么主动调用调度程序。由于 `jffs2_sb_info.gc_task` 保存了 GC 内核线程 PCB 的地址，所以就不需要创建一个额外的等待队列并将其加入其中了，只需将其状态改为 `TASK_INTERRUPTIBLE` 即可，而在调度程序中会把它从运行队列中删除。

需要说明的是这样进入睡眠的方法存在竞争条件。要想无竞争地进入睡眠就必须在判断条件是否为真前改变进程的状态，这样在判断条件之后、进入调度程序之前，用于唤醒 GC 内核线程的中断执行流能够撤销进入睡眠。但是根据作者的注释即使发生竞争条件也无大碍。

GC 内核线程无事可作时进入 `TASK_INTERRUPTIBLE` 状态，收到 `SIGHUP`、`SIGKILL`、`SIGSTOP` 和 `SIGCONT` 中任何一种信号时恢复到 `TASK_RUNNING` 状态。那么每次恢复运行后都得首先处理所有非阻塞挂起信号，判断唤醒的原因。`dequeue_signal` 取出非阻塞挂起信号中编号最小的那一个：

```
cond_resched(); 在适当的时机让出cpu
/* Put_super will send a SIGKILL and then wait on the sem. */
while (signal_pending(current)) { 检查当前进程是否有信号处理，返回不为0表示有信号需要处理
    siginfo_t info;
    unsigned long signr;
    spin_lock_irq(&current->sigmask_lock);
    signr = dequeue_signal(&current->blocked, &info);
    spin_unlock_irq(&current->sigmask_lock);

    switch(signr) {
```

case SIGSTOP:

```
D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread(): SIGSTOP received.\n"));
set_current_state(TASK_STOPPED);
schedule();
break;
```

如果因收到 SIGSTOP 信号而唤醒，那么 GC 内核线程应该进入 TASK_STOPPED 状态并立刻引发调度。注意当再次恢复执行时，通过 break 语句直接跳出信号处理 while 循环。

case SIGKILL:

```
D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread(): SIGKILL received.\n"));
spin_lock_bh(&c->erase_completion_lock);
c->gc_task = NULL;
spin_unlock_bh(&c->erase_completion_lock);
complete_and_exit(&c->gc_thread_exit, 0);
```

如果因收到 SIGKILL 信号而唤醒，那么 GC 内核线程应该立即结束。当需要结束 GC 内核线程时（比如卸载 jffs2 文件系统时、或者重新按照只读方式挂载时），当前进程通过 jffs2_stop_garbage_collect_thread 函数给它发送 SIGKILL 信号，然后阻塞在 gc_thread_exit.wait 等待队列上；而由 GC 内核线程在处理 SIGKILL 信号时再将这个进程唤醒并完成退出。[主进程阻塞在gc_thread_exit的wait等待队列上；](#)

case SIGHUP:

```
D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread(): SIGHUP received.\n"));
break;
```

如果因收到 SIGHUP 信号而唤醒，那么说明 GC 内核线程有事可作了，所以通过 break 语句直接跳出信号处理 while 循环。在 jffs2_garbage_collect_trigger 函数中向它发送 SIGHUP 信号。

```
default:
    D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread(): signal %ld received\n", signr));
}
} //while
```

最后的 default 分支用于处理 SIGCONT 信号。此时没有直接跳出 while 循环，而是接着处理剩余的非阻塞挂起信号（如果存在的话），直到所有的信号都处理完才结束 while 循环。

（需要说明的是，虽然代码中允许 GC 内核线程接收四种信号，但现有的代码中只使用到了 SIGKILL 和 SIGHUP 两种信号。）

```
/* We don't want SIGHUP to interrupt us. STOP and KILL are OK though. */
spin_lock_irq(&current->sigmask_lock);
siginitsetinv (&current->blocked, sigmask(SIGKILL) | sigmask(SIGSTOP) | sigmask(SIGCONT));
recalc_sigpending();
spin_unlock_irq(&current->sigmask_lock);
D1(printk(KERN_DEBUG "jffs2_garbage_collect_thread(): pass\n"));
```

```

    jffs2_garbage_collect_pass(c);
} //for
}

```

最后，如果 GC 内核线程恢复执行的原因不是希望其退出的话，就说明它有事可做了。GC 操作由 jffs2_garbage_collect_pass 函数完成。另外根据作者的注释，在执行 GC 操作期间不希望收到 SIGHUP 信号，这是为什么？另外在 GC 操作后如何进入 TASK_INTERRUPTIBLE 状态？

jffs2_garbage_collect_pass函数

在一个擦除块中既有有效数据，又有过时数据。那么 GC 的思路就是：挑选一个擦除块，每次 GC 只针对一个有效的数据实体：将它的副本到另外一个擦除块中，从而使得原来的数据实体变成过时的。直到某次 GC 后使得整个擦除块只含有过时的数据实体，就可以启动擦除操作了。

```

/* jffs2_garbage_collect_pass
   Make a single attempt to progress GC. Move one node, and possibly start erasing one eraseblock.
*/
int jffs2_garbage_collect_pass(struct jffs2_sb_info *c)
{
    struct jffs2_eraseblock *jeb;
    struct jffs2_inode_info *fi;
    struct jffs2_raw_node_ref *raw;
    struct jffs2_node_frag *frag;
    struct jffs2_full_dnode *fn = NULL;
    struct jffs2_full_dirent *fd;
    uint32_t start = 0, end = 0, nrfrags = 0;
    uint32_t inum;
    struct inode *inode;
    int ret = 0;

    if (down_interruptible(&c->alloc_sem)) // alloc_sem是保护sb_info的？
        return -EINTR;
    spin_lock_bh(&c->erase_completion_lock); // erase_completion是保护free_list和erasing_list的

```

在 GC 期间要持有信号量 alloc_sem。如果 unchecked_size 不为 0，则不能开始 GC 操作。为什么？

```

while (c->unchecked_size) { // 还有未检查的区域，必须先检查了来
    /* We can't start doing GC yet. We haven't finished checking
       the node CRCs etc. Do it now and wait for it. */
    struct jffs2_inode_cache *ic;
    if (c->checked_ino > c->highest_ino) {
        printk(KERN_CRIT "Checked all inodes but still 0x%x bytes of unchecked space?\n",
                   c->unchecked_size);
        D1(jffs2_dump_block_lists(c));
    }
}

```



```

    BUG();
}
ic = jffs2_get_ino_cache(c, c->checked_ino++);
if (lic) // 从第一个未check的ino_cache开始
    continue;
if (!lic->nlink) {
    D1(printk(KERN_DEBUG "Skipping check of ino %#d with nlink zero\n", ic->ino));
    continue;
}
if (ic->state != INO_STATE_UNCHECKED) {
    D1(printk(KERN_DEBUG "Skipping check of ino %#d already in state %d\n", ic->ino, ic->state));
    continue;
}
spin_unlock_bh(&c->erase_completion_lock);
D1(printk(KERN_DEBUG "jffs2_garbage_collect_pass() triggering inode scan of ino#%d\n", ic->ino));
{
    /* XXX: This wants doing more sensibly -- split the core of jffs2_do_read_inode up */
    struct inode *i = iget(OFNI_BS_2SFFJ(c), ic->ino); // iget函数最终会调用jffs2_read_inode，就是会执行check的操作
    if (is_bad_inode(i)) {
        printk(KERN_NOTICE "Eep. read_inode() failed for ino %u\n", ic->ino);
        ret = -EIO;
    }
    iput(i);
}
up(&c->alloc_sem);
return ret;
}
// 到这里时所有的block都已经被check了

/* First, work out which block we're garbage-collecting */
jeb = c->gcblock; // jeb是erasing block的意思
if (!jeb)
    jeb = jffs2_find_gc_block(c);
if (!jeb) {
    printk(KERN_NOTICE "jffs2: Couldn't find erase block to garbage collect!\n");
    spin_unlock_bh(&c->erase_completion_lock);
    up(&c->alloc_sem);
    return -EIO;
}

```

jffs2_sb_info 的 gcblock 指向当前应该被 GC 的擦除块。如果它为空，则由 jffs2_find_gc_block 函数从含有过时数据实体的擦除块中选择一个（该函数涉及 Wear Levelling 算法，有待深究）。（在第一次运行 GC 前、或完成一个擦除块的 GC 操作后 gcblock 为 NULL，还有其它情况么？）

```

D1(printk(KERN_DEBUG "GC from block %08x, used_size %08x, dirty_size %08x, free_size %08x\n",

```



```

        jeb->offset, jeb->used_size, jeb->dirty_size, jeb->free_size));
D1(if (c->nextblock)
    printk(KERN_DEBUG "Nextblock at %08x, used_size %08x, dirty_size %08x, wasted_size %08x,
        free_size %08x\n", c->nextblock->offset, c->nextblock->used_size, c->nextblock->dirty_size,
        c->nextblock->wasted_size, c->nextblock->free_size)
);

if (!jeb->used_size) {
    up(&c->alloc_sem);
    goto eraseit;
}

```

`used_size` 为擦除块内所有有效数据实体所占的空间。如果它为 0，则说明整个擦除块都过时了，因此可以直接跳到 `eraseit` 处将其描述符加入 `erase_pending_list` 链表。

一个 flash 擦除块内所有数据实体的内核描述符由 `next_phys` 域组织成一个链表，其首尾元素分别由擦除块描述符 `jffs2_eraseblock` 的 `first_node` 和 `last_node` 域指向，而 `gc_node` 指向当前被 GC 的数据实体的描述符。

```

raw = jeb->gc_node; 在jffs2_find_gc_block函数中已经将块内第一个数据实体赋给了gc_node了；
while(ref_obsolete(raw)) {
    D1(printk(KERN_DEBUG "Node at 0x%08x is obsolete... skipping\n", ref_offset(raw)));
    jeb->gc_node = raw = raw->next_phys;
    if (!raw) {
        printk(KERN_WARNING "eep. End of raw list while still supposedly nodes to GC\n");
        printk(KERN_WARNING "erase block at 0x%08x. free_size 0x%08x, dirty_size 0x%08x,
            used_size 0x%08x\n", jeb->offset, jeb->free_size, jeb->dirty_size, jeb->used_size);
        spin_unlock_bh(&c->erase_completion_lock);
        up(&c->alloc_sem);
        BUG();
    }
}

```

当重新选择一个新的擦除块进行 GC 时，`gc_node` 指向该擦除块内第一个数据实体的描述符（见 `jffs2_find_gc_block` 函数）。如果数据实体过时，那么一直步进到第一个不过时的数据实体。这是因为 GC 操作的对象是有效数据实体（就是要将该擦除块内有效的数据实体写一个副本到另外的擦除块中，从而使有效的数据实体变成过时的，最终使整个擦除块上的数据实体都过时，然后就可以擦除整个擦除块了）。

```

D1(printk(KERN_DEBUG "Going to garbage collect node at 0x%08x\n", ref_offset(raw)));
if (!raw->next_in_ino) {
    /* Inode-less node. Clean marker, snapshot or something like that */
    /* FIXME: If it's something that needs to be copied, including something
        we don't grok that has JFFS2_NODETYPE_RWCOMPAT_COPY, we should do so */
    spin_unlock_bh(&c->erase_completion_lock);
    jffs2_mark_node_obsolete(c, raw);
}

```

```

    up(&c->alloc_sem);
    goto eraseit_lock;
}

```

任何文件的数据实体的内核描述符的 `next_in_ino` 用于组织循环链表。所以如果它为 `空`，则说明该数据实体不从属于任何文件。根据注释可能为 `cleanmarker` 或者 `snapshot`，此时标记描述符为过时即可。

这里才开始回收一个正常的数据实体

```

inum = jffs2_raw_ref_to_inum(raw);
D1(printk(KERN_DEBUG "Inode number is %u\n", inum));
spin_unlock_bh(&c->erase_completion_lock);
D1(printk(KERN_DEBUG "jffs2_garbage_collect_pass collecting from block @0x%08x. Node @0x%08x,
                    ino %u\n", jeb->offset, ref_offset(raw), inum));
inode = iget(OFNI_BS_2SFFJ(c), inum);
    OFNI_BS_2SFFJ(c)是从sb_info得到super_block结构体的地址的函数

```

然后通过 `jffs2_raw_ref_to_inum` 函数返回数据实体所在文件的索引节点号，并由 `iget` 函数返回文件的索引结点指针（同时增加其引用计数）。（这岂不是 GC 只操作有 `inode` 的文件了？！而 `inode` 只有在打开文件期间才存在啊？！）

```

if (is_bad_inode(inode)) {
    printk(KERN_NOTICE "Eep. read_inode() failed for ino %u\n", inum);
    /* NB. This will happen again. We need to do something appropriate here. */
    up(&c->alloc_sem);
    iput(inode);
    return -EIO;
}

```

先前在打开文件、创建 `inode` 对象时（在 `jffs2_read_inode` 函数中），如果读取 `flash` 上的数据实体失败，则通过 `make_bad_inode` 函数标记该 `inode` 为 `bad`，其代码如下：

```

ret = jffs2_do_read_inode(c, f, inode->i_ino, &latest_node);
if (ret) {
    make_bad_inode(inode);
    up(&f->sem);
    return;
}

```

读取数据实体失败是由于访问介质错误引起的，所以在 GC 操作中一旦发现这种情况就直接以 `EIO` 退出了。

```

f = JFFS2_INODE_INFO(inode);
down(&f->sem);
/* Now we have the lock for this inode. Check that it's still the one at the head of the list. */

```

在 GC 操作期间必须持有 `jffs2_inode_info.sem` 信号量，它用于同步写操作和 GC 操作。参见 [上文](#)。

```

if (ref_obsolete(raw)) {
    D1(printk(KERN_DEBUG "node to be GC'd was obsoleted in the meantime.\n"));
}

```

```

    /* They'll call again */
    goto upnout;
}

```

在真正开始 GC 操作前还要再次检查该数据实体是否变成过时的了,这是因为前面获得 `jffs2_inode_info.sem` 信号量时可能阻塞,所以需要再次检查。

(下面的代码有待进一步分析,针对数据实体的类型通过相应的 `jffs2_garbage_collect_xxxx` 函数完成 GC 操作。等到把一个擦除块中所有有效的数据实体移走之后,即不含有任何有效数据实体(擦除块描述符中 `used_size` 域等于 0)就可以擦除这个擦除块了:将其描述符加入 `erase_pending_list` 链表并激活擦除过程,并将 `gcblock` 设置为 `NULL`)。

注意每次回收垃圾只尝试回收一个 `raw_inode` 或 `raw_dirent`

```

/* OK. Looks safe. And nobody can get us now because we have the semaphore. Move the block */
if (f->metadata && f->metadata->raw == raw) {
    fn = f->metadata;
    ret = jffs2_garbage_collect_metadata(c, jeb, f, fn);
    goto upnout;
}

/* FIXME. Read node and do lookup? */
for (frag = frag_first(&f->fragtree); frag; frag = frag_next(frag)) {
    if (frag->node && frag->node->raw == raw) {
        fn = frag->node;
        end = frag->ofs + frag->size;
#ifdef 1
        /* Temporary debugging sanity checks, till we're ready to _trust_ the REF_PRISTINE flag stuff */
        if (!nrfrags && ref_flags(fn->raw) == REF_PRISTINE) {
            if (fn->frags > 1)
                printk(KERN_WARNING "REF_PRISTINE node at 0x%08x had %d frags. Tell
                                     dwmw2\n", ref_offset(raw), fn->frags);
            if (frag->ofs & (PAGE_CACHE_SIZE-1) && frag_prev(frag) && frag_prev(frag)->node)
                printk(KERN_WARNING "REF_PRISTINE node at 0x%08x had a previous non-hole
                                     frag in the same page. Tell dwmw2\n", ref_offset(raw));
            if ((frag->ofs+frag->size) & (PAGE_CACHE_SIZE-1) && frag_next(frag) &&
                frag_next(frag)->node)
                printk(KERN_WARNING "REF_PRISTINE node at 0x%08x (%08x-%08x) had a
                                     following non-hole frag in the same page. Tell dwmw2\n",
                                     ref_offset(raw), frag->ofs, frag->ofs+frag->size);
        }
#endif
        if (!nrfrags++)
            start = frag->ofs;
        if (nrfrags == frag->node->frags)
            break; /* We've found them all */
    }
}

```

```

} // for
if (fn) {
    /* We found a datanode. Do the GC */
    if((start >> PAGE_CACHE_SHIFT) < ((end-1) >> PAGE_CACHE_SHIFT)) {
        /* It crosses a page boundary. Therefore, it must be a hole. */
        ret = jffs2_garbage_collect_hole(c, jeb, f, fn, start, end);
    } else {
        /* It could still be a hole. But we GC the page this way anyway */
        ret = jffs2_garbage_collect_dnode(c, jeb, f, fn, start, end);
    }
    goto upnout;
}

/* Wasn't a dnode. Try dirent */
for (fd = f->dents; fd; fd=fd->next) {
    if (fd->raw == raw)
        break;
}

if (fd && fd->ino) {
    ret = jffs2_garbage_collect_dirent(c, jeb, f, fd);
} else if (fd) {
    ret = jffs2_garbage_collect_deletion_dirent(c, jeb, f, fd);
} else {
    printk(KERN_WARNING "Raw node at 0x%08x wasn't in node lists for ino #%%u\n",
            ref_offset(raw), f->inocache->ino);

    if (ref_obsolete(raw)) {
        printk(KERN_WARNING "But it's obsolete so we don't mind too much\n");
    } else {
        ret = -EIO;
    }
}

upnout:
    up(&f->sem);
    up(&c->alloc_sem);
    iput(inode);
eraseit_lock:
    /* If we've finished this block, start it erasing */
    spin_lock_bh(&c->erase_completion_lock);
eraseit:
    if (c->gcblock && !c->gcblock->used_size) {
        D1(printk(KERN_DEBUG "Block at 0x%08x completely obsoleted by GC. Moving to
            erase_pending_list\n", c->gcblock->offset));

        /* We're GC'ing an empty block? */

```

```

    list_add_tail(&c->gcblock->list, &c->erase_pending_list);
    c->gcblock = NULL;
    c->nr_erasing_blocks++;
    jffs2_erase_pending_trigger(c);
}
spin_unlock_bh(&c->erase_completion_lock);
return ret;
}

```

jffs2_erase_pending_trigger函数

这个函数设置 jffs2 的超级块中的 s_dirt 标志：

```

void jffs2_erase_pending_trigger(struct jffs2_sb_info *c)
{
    OFNI_BS_2SFFJ(c)->s_dirt = 1;
}

```

这样当卸载 jffs2 时，在 sync_super 函数中会遍历 super_blocks 队列调用所有 s_dirt 标志有效的超级块的 write_super 方法，对于 jffs2 而言即为 jffs2_write_super 函数：

```

void jffs2_write_super (struct super_block *sb)
{
    struct jffs2_sb_info *c = JFFS2_SB_INFO(sb);
    sb->s_dirt = 0;

    if (sb->s_flags & MS_RDONLY)
        return;

    D1(printk(KERN_DEBUG "jffs2_write_super()\n"));
    jffs2_garbage_collect_trigger(c);
    jffs2_erase_pending_blocks(c);
}

```

在 jffs2_garbage_collect_trigger 函数中向 GC 内核线程发送 SIGHUP 信号。总之在卸载 jffs2 时唤醒 GC 内核线程的步骤如下：

```

sys_umount > do_umount > fsync_dev > sync_supers > s_op->write_super (即 jffs2_write_super) >
jffs2_garbage_collect_trigger > send_sig(SIGHUP, c->gc_task, 1);

```

第 11 章 讨论和体会

什么是日志文件系统，为什么要使用jffs2

上层 VFS 框架通过函数指针接口实现与底层具体文件系统实现相隔离。底层具体文件系统提供在相应设备上按照特定格式访问各种类型文件的方法，并分配函数指针接口的空间（而 VFS 数据结构中只为指向接口的指针分配了空间）。VFS 框架实现了访问、管理文件系统的上层策略，具体文件系统则提供在具体设备上的底层机制。

必须针对具体设备的特点设计具体文件系统的数据存储格式。jffs2 是专门针对 flash 的特点设计的文件系统。flash 为 EEPROM，可以随机读出，但是在写之前必须执行擦除操作，这是因为 flash 的写操作只能将 1 写为 0，所以必须首先将待写区域全部“擦除”为 1。而且擦除操作必须以擦除块为单位进行，即使只改动一个字节的数，也不得不擦除其所在的整个擦除块。

flash 芯片由若干擦除块组成，单个擦除块的寿命（大约 100,000 次）决定了整个 flash 芯片的寿命。所以基于 flash 的文件系统应该使得所有的擦除块都被几乎同等频繁地使用，避免过度使用一部分擦除块而导致整个芯片过早报废。这种算法称为“Wear levelling”。

在 ext2 文件系统中如果修改了文件的某一个区域，那么在写文件前，必须首先找出相应区域所对应的磁盘块，然后视需要读出这个块，再修改，最后异步地写回这个块。而在 jffs2 文件系统中如果修改了文件的某一个区域，那么不会立即删除 flash 上含有该区域数据的原有数据实体，是直接向 flash 分区中写入一个崭新的数据实体，同时将原有数据实体在内核中的描述符标记为“过时”。那么在打开这个文件、为其数据实体创建相应的数据结构时就会跳过所有过时的数据实体，参见 [jffs2_get_inode_nodes](#) 函数。另外，当标记原有数据实体“过时”时，在 [jffs2_mark_node_obsolete](#) 函数（尚未详细分析）中还会同时修改原有数据实体头部的 `nodetype` 区域。这个操作可以从 [jffs2map2](#) 的结果看出来。

flash 上干净擦除块数量低于某一阈值时或者卸载 jffs2 时 GC 就会启用：回收“过时”的数据实体所在的空间。其实每次 GC 内核线程运行时只把被 GC 的擦除块中一个有效数据实体变成过时，当整个擦除块中只含有过时数据实体时进行擦除操作。

jffs2 尽量减少擦除操作：如果每擦除一个擦除块只是修改其中的一个单元，那么效率是最低的。如果需要修改整个擦除块的所有单元，那么执行一次擦除操作的效率就是最高的。在 GC 时会挑选一个含有过时数据实体的擦除块，把其中有效的数据实体写一个副本到另外的擦除块上，从而使有效的数据实体也变成过时的。当整个擦除块变成全脏的时候就可以进行擦除操作了（挑选“被 GC 的”擦除块和“另外的”擦除块都与 Wear Levelling 算法有关）。

为什么需要红黑树

jffs2 文件系统中需要维护数据实体的内核描述符 `jffs2_raw_node_ref` 的链表。假设文件初始数据结点为 N 个，那么多次修改后实际的数据实体个数和链表长度将远远大于 N 。

VFS 的读写函数接口提供文件某个区域的起始位置 `offset` 及长度 `len`。当然可以遍历整个链表，跳过落在这个区域内的过时数据实体、找到有效的数据实体，然后再访问 `flash` 上的有效实体。但是如果 `N` 较大，修改次数较多，那么每次读写时都要遍历整个链表就十分缺乏效率了。

所以需要一种合适的用于快速查找的数据结构及算法，将链表元素以另外的方式组织起来，于是就采用了红黑树。文件个数据实体的内核描述符除了组织在链表中，还通过相应的 `jffs2_full_dnode` 和 `jffs2_node_frag` 组织在红黑树中，树根为 `jffs2_inode_info` 的 `fragtree`。

对文件进行修改时要写入新的数据实体，同时在内核中创建相应的描述符和 `jffs2_full_dnode` 数据结构，在将新实体的描述符加入 `jffs2_inode_cache.nodes` 链表的首部时还要标记原有实体的描述符为过时，同时，还要修改红黑树中相应结点的 `node` 指针指向新的 `jffs2_full_dnode` 结构，同时递减过时的 `jffs2_full_dnode` 的 `frags` 引用计数（原来为 1，现在就为 0 了）。（注意，红黑树中的结点始终指向有效数据实体的相关数据结构）

何时、如何判断数据实体是过时的

在修改文件时写入新的数据实体，此时显然知道原有数据实体是过时的。但这只是整个事情的一小部分。

在挂载文件系统时将扫描整个 `flash` 分区，为所有的数据实体创建内核描述符，此时只对正规文件的数据实体本身进行 `crc` 校验，而把对后继数据的 `crc` 校验延迟到了打开文件时。如果发现数据实体本身的 `crc` 校验错误，则标记其内核描述符为过时的，参见 [jffs2_get_inode_nodes 函数的相关部分](#)。

需要强调的是，在挂载文件系统时要为所有的数据实体创建内核描述符，无论其过时与否！在 `jffs2_mark_node_obsolete` 函数中标记数据实体过时，还会同时修改 `flash` 中数据实体本身头部的 `nodetype`，使得头部的 `crc` 校验失败。所以在挂载文件系统时是知道数据实体是否为过时的。

对于目录文件，在挂载的后期创建目录项的 `jffs2_full_dirent` 数据结构的链表。此时就可以发现目录项数据实体是否过时了：如果它们的名字相同，那么版本号较高的那个为有效的，其它的为过时的。此时只将有效数据实体的 `jffs2_full_dirent` 加入链表，而释放过时的数据结构。详见 [jffs2_add_fd_to_list 函数](#)。

对于正规文件，在打开文件、创建 `inode` 时根据数据实体的内核描述符创建其它数据结构，并组织红黑树。在组织红黑树时就可以发现是否有多个数据实体涉及相同区域的数据了，版本号最高的数据实体为有效的，而其它的都是过时的。此时将红黑树中的结点指向版本号最高的数据实体的 `jffs2_full_dnode` 数据结构，并标记其它数据实体的内核描述符为过时。具体操作在 `jffs2_add_full_dnode_to_inode` 函数中，有待详细分析核实。

后记

本文从 jffs2 文件系统的注册、挂载、文件的打开、正规文件的读写这几个情景出发分析了其源代码的主体框架，描述了 jffs2 文件系统的核心数据结构及之间的相互引用关系。

由于作者时间有限，jffs2 文件系统所特有的垃圾回收（GC）算法、Wear Levelling 算法及红黑树的插入删除操作等方面需要进一步深究，在此罗列如下：

1. 红黑树的插入、删除、查找、平衡，比如 `jffs2_add_full_dnode_to_inode` 函数、`jffs2_lookup_node_frag` 函数。
2. 读写除正规文件之外其它特殊文件的方法，比如目录文件、符号链接文件等等。
3. 关闭文件时释放相应数据结构的过程。
4. 使用 `zlib` 库压缩、解压缩数据的方法。
5. Wear Levelling 算法有关的函数，比如 `jffs2_rotate_list` 和 `jffs2_reserve_space` 函数，以及 GC 时选择一个擦除块的方法。
6. NAND flash 的异步写入机制。

由于作者能力有限，现有文档中存在许多疑问（采用红色字体），也一定有许多错误，欢迎大家补充、批评指正，谢谢！

（从www.infradead.org上可以获得与jffs2 相关的各种资源，包括正在开发中的jffs3）

附录 用jffs2map2 模块导出文件的数据实体（new）

这个模块使得 jffs2 前所未有地真实、清晰，可以很好地促进对相关源代码的理解，发现先前犯下的许多错误认识，呵呵。

jffs2map2 模块可以用于导出指定的 jffs2 文件系统上指定 ino 的文件的所有数据结点的信息，这些信息来自于挂载 jffs2 时创建的文件描述符 jffs2_inode_cache 的 nodes 链表。注意用 jffs2map2 观察指定文件时其可能尚未打开，所以上层的 dentry、inode 尚不存在，因此只能得到比较有限的记载于 jffs2_inode_cache 和各 jffs2_raw_node_ref 中的信息。比如根文件系统映像中根目录下 proc 和 dev 目录下都为空，它们的内容在内核启动时挂载 procfs 和 devfs 后才在内存中动态创建，所以用 jffs2map2 只能看到描述挂载点本身的那个 jffs2_raw_inode 数据实体，而没有任何其它目录项。

注意为了使用 jffs2map2 模块内核必需导出 super_blocks 和 sb_lock 变量。在安装 jffs2map2 模块时可以在命令行通过参数 “jffs2map_sdev” 和 “jffs2map_ino” 分别指定 jffs2 所在的设备，以及指定的文件 ino。如果缺省则将来会输出作为根文件系统的 jffs2 上的根目录的信息。安装 jffs2map2 后，就可以通过 “cat /proc/jffs2map” 来读出指定文件的数据结点信息了。这样从根目录开始我们可以逐层得到任何文件的 ino，进而在安装 jffs2map2 时指定其 ino，从而观察挂载文件系统后相关文件的信息。

目前 jffs2map2 实现得还有些不便于使用，欢迎改进！

1. 每次安装后只能导出一个文件的信息，如果要观察其它文件，则必须首先 rmmod，然后再次 insmod 并同时用 “jffs2map_ino” 指定其它文件的 ino（庆幸开发板上的 bash 支持 history 功能，逃避了重复键入的麻烦:-P）。
2. 由于从 proc 导出数据时一次不能超过一个页框，否则就应该使用 proc 读回调函数接口中的 start、pages 参数来生成多余一个页框的数据了。

观察根目录文件的数据实体

1. 执行操作

```
insmod jffs2map2.o
cat /proc/jffs2map
```

2. 输出结果

```
Display jffs2 with s_dev = (31,4).
The highest ino is 1174, displaying file information with ino 1
ino=1, nlink=0x1, state: unchecked
<pristine,0x1341238,0x30>,jffs2_raw_dirent,ino=0,pino=1,type=UNKNOWN,name: shrek2
<pristine,0x1341208,0x30>,jffs2_raw_dirent,ino=1174,pino=1,type=DIR,name: shrek3
<pristine,0x5906c4,0x2c>,jffs2_raw_dirent,ino=823,pino=1,type=DIR,name: work
<pristine,0x588d50,0x2c>,jffs2_raw_dirent,ino=817,pino=1,type=LNK,name: tmp
<pristine,0x588cd0,0x2c>,jffs2_raw_dirent,ino=816,pino=1,type=LNK,name: var
<pristine,0x565320,0x38>,jffs2_raw_dirent,ino=815,pino=1,type=REG,name: .ramfs.tar.gz
```

```

<pristine,0x3af4,0x30>,jffs2_raw_dirent,ino=133,pino=1,type=DIR,name: yanshou
<pristine,0x3a84,0x2c>,jffs2_raw_dirent,ino=132,pino=1,type=DIR,name: root
<pristine,0x3a14,0x2c>,jffs2_raw_dirent,ino=131,pino=1,type=DIR,name: proc
<pristine,0x38c4,0x2c>,jffs2_raw_dirent,ino=128,pino=1,type=DIR,name: home
<pristine,0x3854,0x2c>,jffs2_raw_dirent,ino=127,pino=1,type=DIR,name: dev
<pristine,0x26b4,0x2c>,jffs2_raw_dirent,ino=88,pino=1,type=DIR,name: etc
<pristine,0x2644,0x2c>,jffs2_raw_dirent,ino=87,pino=1,type=DIR,name:/sbin
<pristine,0x1d40,0x2c>,jffs2_raw_dirent,ino=67,pino=1,type=DIR,name: lib
<pristine,0x1cd0,0x2c>,jffs2_raw_dirent,ino=66,pino=1,type=DIR,name: bin
<pristine,0x16a4,0x2c>,jffs2_raw_dirent,ino=52,pino=1,type=DIR,name: mnt
<pristine,0x0,0x2c>,jffs2_raw_dirent,ino=2,pino=1,type=DIR,name: usr

```

3. 结果分析

由于在安装模块时没有指定 jffs2 所在的设备号，则缺省值为根设备（主号 31，次号 4）；由于没有指定文件的 ino，则缺省值为 1，即输出根目录的信息。

注意打开文件后上层 VFS 所使用的文件硬链接计数为 inode 的 nlink，而这里输出的 nlink 为 jffs2_inode_cache 的 nlink，二者不同（在打开文件时会进一步根据非叶目录下子目录的个数递增父目录的硬链接计数）。

接下来的每一行输出包括如下内容：数据实体的状态、在 flash 上的物理地址、长度、数据实体的类型、目录项所代表的文件的 ino、目录项所属的目录文件的 ino、目录项所对应文件的类型、目录项所对应文件的名字。

由此可见，根目录文件没有那个惟一的 jffs2_raw_inode 数据实体，只包含其下子目录、文件的目录项。另外目录项的 pino 都等于 1，它们的内核描述符都链接在根目录文件的 jffs2_inode_cache 的 nodes 链表中。

另外在控制台执行“ls -l /”的结果如下：

```

total 0
drwxr-xr-x    2 root    root          0 Dec 24  2004 bin
drwxr-xr-x    1 root    root          0 Jan  1 00:00 dev
drwxr-xr-x   26 root    root          0 Jan  1  1970 etc
drwxr-xr-x    4 root    root          0 Jan  1  1970 home
drwxr-xr-x    4 root    root          0 Jul  9  2002 lib
drwxr-xr-x    6 root    root          0 Jan  1  1970 mnt
dr-xr-xr-x   26 root    root          0 Jan  1 00:00 proc
drwxr-xr-x    3 root    root          0 Jan  1  1970 root
drwxr-xr-x    2 root    root          0 Jan  1  1970/sbin
drwxr-xr-x    2 root    root          0 Jan  1  1970 shrek3
lrwxrwxrwx    1 root    root       13 Aug 11  2005 tmp -> mnt/ramfs/tmp
drwxr-xr-x    7 root    root          0 Jan  1  1970 usr
lrwxrwxrwx    1 root    root       13 Aug 11  2005 var -> mnt/ramfs/var
drwxr-xr-x    2 root    root          0 Jan  1  1970 work

```

```
drwxr-xr-x    2 500      root          0 Jan  1  1970 yanshou
```

由此可见在“/”下除了有两个符号链接 `tmp` 和 `var` 外，其余的都是目录文件。注意有一个名为“`shrek3`”的目录，原先其名字叫做“`shrek2`”，后来由“`mv`”改名过来。但是此时仍然可以看到由一个关于 `shrek2` 的目录项，而且其 `ino` 等于 0。

下面我们看一个符号链接 `tmp` 的信息注意其 `ino` 等于 817，它的目标文件为“`mnt/ramfs/tmp`”。

观察符号链接的信息

1. 执行操作

```
rmmod jffs2map2
insmod jffs2map2.o jffs2map_ino=817
cat /proc/jffs2map
```

2. 输出结果

```
Display jffs2 with s_dev = (31,4).
The highest ino is 1174, displaying file information with ino 817
ino=817, nlink=0x1, state: present
<normal,0x588d7c,0x54>,jffs2_raw_inode,ino=817,dsize=0xd,csz=0xd
A symbolic link, the linked file is mnt/ramfs/tmp
```

3. 结果分析

从源代码分析中我们已经知道目录文件、符号链接、设备文件都有惟一的 `jffs2_raw_inode` 数据实体。对于符号链接，其后继数据即为被链接文件名。

这里验证了这个结果。

可以看到 `tmp` 文件只有一个 `jffs2_raw_inode` 数据实体，其后继数据长度为 `0xd`，即等于其后继文件名“`mnt/ramfs/tmp`”的长度。

观察正规文件创建后的数据实体

1. 执行操作

```
rmmod jffs2map2
mkdir /test
echo 1 > /test/1.txt
insmod jffs2map2.o jffs2map_ino=1175（注：事先从根目录下得到新建的 test 目录的 ino 为 1175）
cat /proc/jffs2map

rmmod jffs2map2
insmod jffs2map2.o jffs2map_ino=1176（注：从上一步得到 1.txt 文件的 ino 为 1176）
cat /proc/jffs2map
```

2. 输出结果

Display jffs2 with s_dev = (31,4).

The highest ino is 1176, displaying file information with ino 1175

ino=1175, nlink=0x1, state: present

<pristine,0x13452e8,0x30>,jffs2_raw_dirent,ino=1176,pino=1175,type=REG,name: 1.txt

<normal,0x1345234,0x44>,jffs2_raw_inode,ino=1175,dsize=0x0,csize=0x0

A directory.

Display jffs2 with s_dev = (31,4).

The highest ino is 1176, displaying file information with ino 1176

ino=1176, nlink=0x1, state: present

<normal,0x1345318,0x48>,jffs2_raw_inode,ino=1176,dsize=0x2,csize=0x2

A regular file.

<obsolete,0x13452a4,0x44>,Header CRC 98f7fb1d != calculated CRC 5936d419 for node at 13452a4

Unknown node type

3. 结果分析

从源代码分析中我们已经知道除根目录外任何目录都由惟一的 jffs2_raw_inode 数据实体，其后继没有数据。从这里可以看到这一点。在目录 “/test” 下有一个正规文件 “1.txt”，所以观察 test 目录时可以看到其目录项，其 ino 等于 1176，即 1.txt 文件的 ino；pino 等于 test 目录的 ino；类型为 REG，文件名为 “1.txt”。

进一步观察 1.txt 文件的数据实体，由于在创建时只写入了一个字符 1，所以包括字符串结束符在内文件的长度应该为 2，即 dsize 等于 2。

从目录文件的 create 方法可见在创建正规文件之初就创建了一个临时的 jffs2_raw_inode，其上层的 jffs2_full_dnode 由其 jffs2_inode_info 的 metadata 直接指向。等到第一次真正写入时将其过时，然后再写入真正带有有效数据的 jffs2_raw_inode 数据实体。由于在 jffs2_mark_node_obsolete 函数中标记数据实体的内核描述符为过时，还会同时修改 flash 上的数据实体的头部的 nodetype 域，清除其 JFFS2_NODE_ACCURATE 标志，所以头部的 crc 校验会失败。

这里看到的那个被标记为 obsolete 的数据实体相信就是那个在创建之初写入的临时的 jffs2_raw_inode。

另外可以进一步修改这个文件，比如通过 “echo 22 > 1.txt” 或者 “echo 22 >> 1.txt”，再观察该文件数据实体的信息。再进一步，重启系统，然后再次观察。

观察 jffs2_raw_inode 数据实体的大小上限

在使用 mkfs.jffs2 将 ext2 目录树转换为 jffs2 映像时，可以使用其 “-s” 选项指定正规文件数据结点的大小上限，默认值为一个页框。当数据结点大小上限增加时可以减少 jffs2_raw_inode 数据实体本身的个数，但是这仅是一个方面。

1. 执行操作

mount /dev/mtdblock/3 /work (挂载/dev/mtdblock/3 上的 jffs2 到/work 目录)

rmmmod jffs2map2

```
insmod /home/cqt/jffs2map2.o jffs2map_sdev=7939
cat /proc/jffs2map      (输出新挂载的 jffs2 的根目录的信息)

rmmod jffs2map2
insmod /home/cqt/jffs2map2.o jffs2map_ino=3
cat /proc/jffs2map      (输出新挂载的 jffs2 中 mkfs.jffs22 文件的信息)

cp /work/mkfs.jffs22 /    (将这个文件拷贝到根 jffs2 中。注意这两种 jffs2 有不同的数据实体大小上限!)
rmmod jffs2map2
insmod /home/cqt/jffs2map2.o jffs2map_ino=1177
cat /proc/jffs2map      (输出拷贝到根 jffs2 下 mkfs.jffs22 的信息)
```

2. 输出结果

```
Display jffs2 with s_dev = (31,3).
The highest ino is 3, displaying file information with ino 1
ino=1, nlink=0x1, state: unchecked
<pristine,0x2c0040,0x3c>,jffs2_raw_dirent,ino=2,pino=1,type=DIR,name: telnetd_on_pcm7210
<pristine,0x2c000c,0x34>,jffs2_raw_dirent,ino=0,pino=1,type=UNKNOWN,name: mkfs.jffs2
<pristine,0x4cb0,0x34>,jffs2_raw_dirent,ino=3,pino=1,type=REG,name: mkfs.jffs22
<obsolete,0x0,0x34>,jffs2_raw_dirent,ino=2,pino=1,type=REG,name: mkfs.jffs2
```

```
Display jffs2 with s_dev = (31,3).
The highest ino is 3, displaying file information with ino 3
ino=3, nlink=0x1, state: checkedabsent
<normal,0x9014,0x94c>,jffs2_raw_inode,ino=3,dsize=0x13db,csz=0x907
A regular file.
<normal,0x7dc4,0x1250>,jffs2_raw_inode,ino=3,dsize=0x3000,csz=0x120b
<normal,0x65fc,0x17c8>,jffs2_raw_inode,ino=3,dsize=0x3000,csz=0x1784
<normal,0x4ce4,0x1918>,jffs2_raw_inode,ino=3,dsize=0x3000,csz=0x18d2
```

```
Display jffs2 with s_dev = (31,4).
The highest ino is 1177, displaying file information with ino 1177
ino=1177, nlink=0x1, state: present
<normal,0x134ab88,0x1d4>,jffs2_raw_inode,ino=1177,dsize=0x3db,csz=0x18d
A regular file.
<normal,0x134a390,0x7f8>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x7b1
<normal,0x1349da0,0x5f0>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x5aa
<normal,0x134973c,0x664>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x620
<normal,0x1349060,0x6dc>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x698
<normal,0x134859c,0xac4>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0xa7f
<normal,0x1347c2c,0x970>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x92c
<normal,0x134773c,0x4f0>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x4ac
<normal,0x1347068,0x6d4>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0x690
<normal,0x13465dc,0xa8c>,jffs2_raw_inode,ino=1177,dsize=0x1000,csz=0xa46
```

```
<normal,0x1345d2c,0x8b0>,jffs2_raw_inode,ino=1177,dsize=0x1000,csize=0x86b  
<obsolete,0x1345cb4,0x44>,Header CRC 98f7fb1d C 5936d419 for node at 1345cb4  
Unknown node type
```

3. 结果分析

根文件系统为位于 `/dev/mtdblock/4` 上的 `jffs2`，在使用 `mkfs.jffs2` 制作其映像时采用默认的 `jffs2_raw_inode` 大小上限，即 `0x1000`。再使用 `mkfs.jffs2` 制作一个指定大小上限为 `0x3000` 的 `jffs2` 映像，并把它拷贝到 `flash` 上次号为 3 的那个分区上，并挂载到根 `jffs2` 的 `/work` 目录上。

由其根目录文件的信息知，一个名为 `mkfs.jffs22` 的正规文件的 `ino` 等于 3。进一步观察其数据结点大小，结果看到 `dsize` 等于 `0x3000`，而 `csize` 为压缩后的大小。

最后，将这个文件拷贝到根 `jffs2` 中。注意这两个文件系统所支持的数据实体的大小上限不同！然后再次观察这个文件的信息，结果发现数据实体的大小上限由 `0x3000` 变成了 `0x1000`！（最后那个过时的数据结点相信还是目录文件 `create` 的结果，呵呵。）

（全文完）