Linux MTD 结构分析

作者 :董磊鋆
Email:dongleijun4000@hotmail.com

专有名词：

- MTD：Memory Technology Device，内存技术设备，
- JEDEC：Joint Electron Device Engineering Council，电子电器设备联合会
- CFI：Common Flash Interface，通用 Flash 接口，Intel 发起的一个 Flash 的接口标准
- OOB： out of band，某些内存技术支持 out-of-band 数据——例如，NAND flash 每 512 字节的块有 16 个字节的 extra data，用于纠错或元数据。
- ECC： error correction，某些硬件不仅允许对 flash 的访问，也有 ecc 功能，所有 flash 器件都受位交换现象的困扰。在某些情况下，一个比特位会发生反转或被报告反转了，如果此位真的反转了，就要采用 ECC 算法。
- erasesize： 一个 erase 命令可以擦除的最小块的尺寸
- buswidth：MTD 设备的接口总线宽度
- interleave：交错数，几块芯片平行连接成一块芯片，使 buswidth 变大
- devicetype：芯片类型，x8、x16 或者 x32
- NAND：一种 Flash 技术，参看 NAND 和 NOR 的比较
- NOR：一种 Flash 技术，参看 NAND 和 NOR 的比较
- Wear out：Flash 的擦除次数有限制，一般在 1000,000 次左右，由于过量的擦除，使得 Flash 无效。

体系结构

MTD(memory technology device 内存技术设备)是用于访问 memory 设备(ROM、flash)的 Linux 的子系统。MTD 的主要目的是为了使新的 memory 设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。MTD 的所有源代码在/drivers/mtd 子目录下。

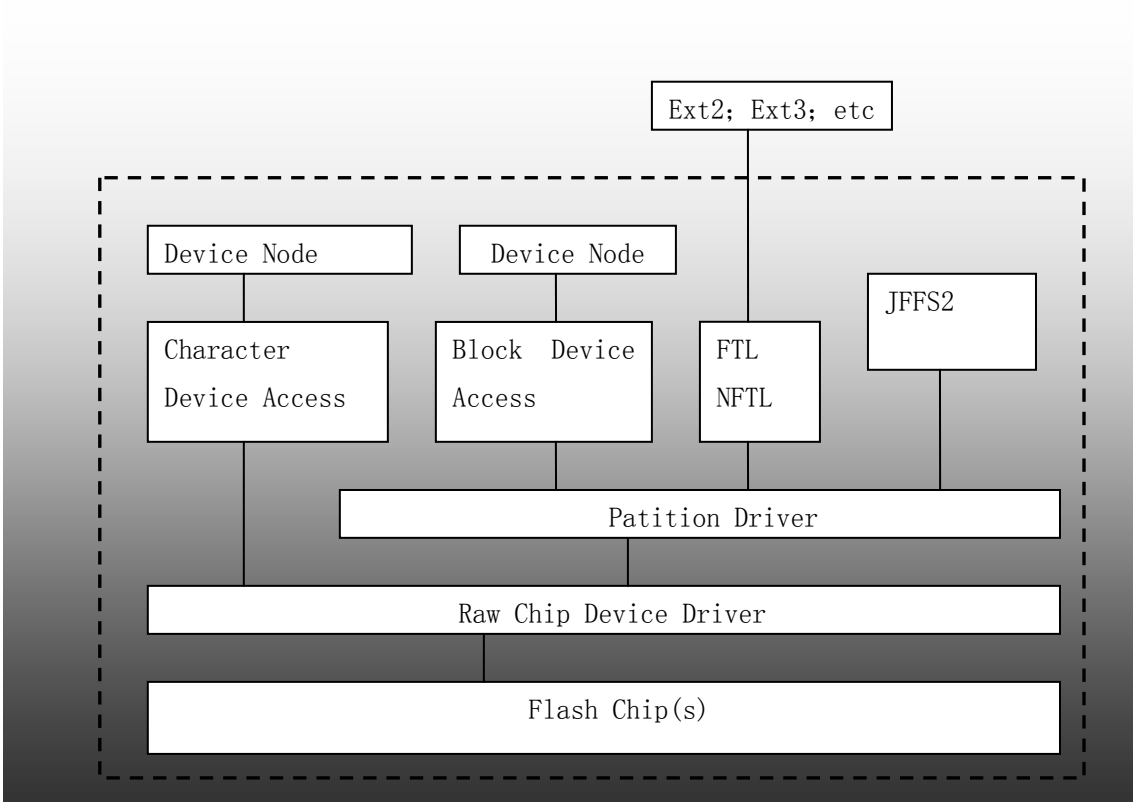

図 1 MTD 体系结构

MTD hardware device drivers

硬件驱动层(Raw Chip Device Driver)：

硬件驱动层负责在 init 时驱动 Flash 硬件，Linux MTD 设备的 NOR Flash 芯片驱动遵循 CFI 接口标准，其驱动程序位于 drivers/mtd/chips 子目录下。NAND 型 Flash 的驱动程序则位于/drivers/mtd/nand 子目录下。

Common Flash Interface (CFI) onboard NOR flash: This is a common solution and is

well-tested and supported, most often using JFFS2 or cramfs file systems.

Flash 型号的探测

为了确定一个 Flash 是否是一个 CFI 使能的 flash memory 器件,首先要往 Flash 的地址 0x55H 写入数据 0x98H，然后从 Flash 的地址 0x10H 处开始连续读取 3 个存储单元中的内容，如果数据总线返回的 3 个存储单元的字符分别为'Q','R'和'Y',那么该器件是一个 CFI 使能的 Flash。在识别 Flash 为 CFI 使能器件后，通过查询命令来读取 CFI 查询结构，这些数据的地址和含义在 cfi_ident.h 文件中。探测 CFI 接口 Flash 设备的程序在文件 cfi_probe.c 中，这些设备的类型为”cfi_probe”。

也可以用 JEDEC（电子电器设备联合会）标准设备模仿 CFI 接口，探测 JEDEC 设备的程序在 jedec_probe.c 中，JEDEC 设备的类型为”jedec_probe”。

CFI 设备和 JEDEC 设备都要用到 gen_probe.c 文件。

不同的制造商使用不同的命令集，目前 Linux 的 MTD 实现的命令集有 AMD/Fujitsu 的标准命令集和 Intel/Sharp 的扩展命令集（兼容 Intel/Sharp 标准命令集）两个，这两个命令集分别在 cfi_cmdset_0002.c 和 cfi_cmdset_0001.c 中实现。
此外还有一些非 CFI 标准的 Flash，其中”jedec”类型的 Flash 的探测程序在 jedec.c 中,”sharp”类型的 Flash 的探测程序在 sharp.c 中”amd_flash”类型的 Flash 的探测程序在 amd_flash.c 中。

最后,还有一些非 Flash 的 MTD,比如 ROM 或 absent(无)设备。这些设备的探测程序在 map_rom.c、map_ram.c 和 map_absent.c 中。

所有类型的芯片都通过 chipreg.c 中的 do_map_probe()程序驱动

Flash 芯片驱动器链表
chipreg.c

关于 MTD 芯片注册的文件，此文件中定义的 chip_drvs_list 是所有芯片类型的驱动器链表，在/drivers/mtd/chips 子目录下的其他文件通过调用 register_mtd_chip_driver()和 unregister_mtd_chip_driver()向此链表中添加或去除 MTD 芯片驱动器。

在/drivers/mtd/map/下的文件根据芯片类型调用 do_map_probe()，do_map_probe()通过 get_mtd_chip_driver()获得符合指定 name 的 MTD 芯片驱动器，再调用获得的芯片驱动器的 probe 程序。

cfi_probe 探测
“cfi_probe”型芯片的探测程序，主要由 cfi_chip_probe()、cfi_probe()、cfi_chip_setup()、qry_present()、cfi_probe_init()和 cfi_probe_exit()这几个函数组成。

cfi_probe()是”cfi_probe”类型芯片的探测程序，它调用通用探测程序 mtd_do_chip_probe()，并将 cfi_chip_probe 作为参数传递给 mtd_do_chip_probe()，mtd_do_chip_probe()将间接调用 cfi_chip_probe 的成员函数 cfi_probe_chip()。cfi_probe()注册在”cfi_probe”芯片的驱动器 cfi_chipdrv 中。

cfi_probe_chip()将调用 qry_present()和 cfi_chip_setup()初始化 cfi_private 结构，qry_presetn()负责验证该MTD设备支持CFI接口，cfi_chip_setup()则读出CFI查询结构中的数据（见cfi.h）

cfi_probe_init()和 cfi_probe_exit()是”cfi_prbe”型芯片驱动器的注册程序和清除程序。

jedec_probe探测

“jedec_probe”型芯片的探测程序，主要由 jedec_probe()、jedec_probe_chip()、cfi_jedec_setup()、jedec_probe_init()和jedec_probe_exit()这几个函数组成。

jedec_probe()是”jedec_probe”类型芯片的探测程序，它调用通用探测程序mtd_do_chip_probe()，并将 jedec_chip_probe 作为参数传递给 mtd_do_chip_probe()，mtd_do_chip_probe()将间接调用 jedec_chip_probe 的成员函数 jedec_probe_chip()。jedec_probe()注册在”jedec_probe”芯片的驱动器jedec_chipdrv中。

jedec_probe_chip()调用cfi_jedec_setup()初始化cfi_private结构，cfi_jedec_setup()根据jedec_probe_init()和jedec_probe_exit()是”cfi_prbe”型芯片驱动器的注册程序和清除程序

gen_probe通用芯片探测程序

由mtd_do_chip_probe()、genprobe_ident_chips()、genprobe_new_chip()、check_cmd_set()和cfi_cmdset_unknown()组成

cfi_probe()或 jedec_probe()调用 mtd_do_chip_probe()，mtd_do_chip_probe()调用genprobe_ident_chips()，genprobe_ident_chips()调用 genprobe_new_chip()，genprobe_new_chip()则调用mtd_do_chip_probe()的参数chip_probe->probe_chip()。

CFI标准命令集

cfi_cmdset_0002.c/ cfi_cmdset_0001.c

CFI 的 AMD 标准命令集，cfi_cmdset_0002()调用 cfi_amdstd_setup()建立 mtd_info，cfi_amdstd_read()、cfi_amdstd_write()、cfi_amdstd_sync()、cfi_amdstd_resume()、cfi_amdstd_erase_onesize()（或 cfi_amdstd_erase_varsize()）、cfi_amdstd_suspend()被注册在 mtd_info 中。其中 cfi_amdstd_read()调用 do_read_onechip()，cfi_amdstd_write()调用 do_write_oneword()，cfi_amdstd_erase_onesize()（或cfi_amdstd_erase_varsize()）调用do_erase_oneblock()。

cfi_amdstd_chipdrv是AMD标准命令集的驱动，它被连接在map->fldrv上，其中的destroy函数指针指向cfi_amdstd_destroy()。

cfi_amdstd_init()和cfi_amdstd_exit()负责AMD标准命令集的初始化和清除工作。

MTD设备层（Patition Drivers）：

用于描述MTD原始设备的数据结构是mtd_info，这其中定义了大量的关于MTD的数据和操作

函数。mtd_table（mtdcore.c）则是所有 MTD 原始设备的列表，mtd_part（mtd_part.c）是用于表示 MTD 原始设备分区的结构，其中包含了 mtd_info，因为每一个分区都是被看成一个 MTD 原始设备加在 mtd_table 中的(struct mtd_info *mtd_table[MAX_MTD_DEVICES];)，mtd_part.mtd_info 中的大部分数据都从该分区的主分区 mtd_part->master 中获得。

调用 add_mtd_device()、del_mtd_device()建立/删除 mtd_info 结构并将其加入/删除 mtd_table（或者调用 add_mtd_partition()、del_mtd_partition()（mtdpart.c）建立/删除 mtd_part 结构并将 mtd_part.mtd_info 加入/删除 mtd_table 中）。

在 linux 的 drivers/mtd/maps 目录下存放了对各种使用 NorFlash 开发板的 Flash 地址划分。这个目录下不包含 Nand Flash 的信息。在这个目录下，并没有找到 Omap 的 map 文件，而找到了 HHarm-R3 的 Flash 文件 s3c2410_llg.c。

要将 MTD 移植到任何开发板上，只需要修改这里的文件。

整个 map 文件要完成的任务序列如下:

- 规划需要建立的分区
- 用 s3c2410_map 代表需要探测的目标范围和探测使用的接口。
- 在探测成功后，建立原始设备上的分区。

下面是部分 map 文件的内容：

```
/*
CPU：ARMS3C2410
Linux version:2.4.18
Flash：Intel E28F128
*/
/*
 * Normal mappings of chips on Samsung s3c2410 in physical memory
 */

#include……
MTD 原始设备的起始地址（PA）
#define WINDOW_ADDR 0x01000000
flash 大小 16M
#define WINDOW_SIZE 0x01600000
总线宽度
#define BUSWIDTH 2
module_init(init_s3c2410);
```

static struct mtd_info *mymtd;

以下是分区的内容，初步规划分了 5 个区 bootloader，kernel，ramdisk，jffs2，cramfs 一共用去了 16M 空间，这里仅仅是规划，还没有提交给设备执行。mtd_info 的 startaddress 为 0x01000000，而且只有 16M，我猜是使用了 32M Flash Bank 的后半段位置。

```
static struct mtd_partition s3c2410_partitions[] = {
    {
            name: “reserved for bootloader”,
            size: 0x040000,
            offset: 0x0,
                        mask_flags: MTD_WRITEABLE,
    },
    {
            name: “reserved for kernel”,
            size: 0x0100000,
            offset: 0x040000,
            mask_flags: MTD_WRITEABLE,
    },
    {
            name: “reserved for ramdisk”,
            size: 0x400000,
            offset: 0x140000,
            mask_flags: MTD_WRITEABLE,

    },
        {
                name: “jffs2(8M)”,
                size: 0x800000,
                offset: 0x800000,
        },
        {
                name: “cramfs(2.75M)”,
                size:0x2c0000,
                offset:0x540000,
        }
```

```
};
struct map_info s3c2410_map = {
        name: “s3c2410 flash device”,
        size: WINDOW_SIZE,
        buswidth: BUSWIDTH,
        read8: s3c2410_read8,
        read16: s3c2410_read16,
        read32: s3c2410_read32,
        copy_from: s3c2410_copy_from,
        write8: s3c2410_write8,
        write16: s3c2410_write16,
        write32: s3c2410_write32,
        copy_to: s3c2410_copy_to,
        map_priv_1:     WINDOW_ADDR,
        map_priv_2:     -1,
};
```

下面是 module 初始化例程,主要任务是对先前已经规划的分区（16M）Flash 空间进行探测，如果探测成功,则将在该范围内建立 5 个分区,并且这些分区添加到 MTD Patition 中。在 linux 中使用了三种探测（probe）方式：CFI,Jedec 和 AMDflash。按照代码中的解释，CFI 是用来探测 CFI 标准的 Flash,Jedec 用来探测 Non-CFI 的 Flash, AMDflash:AMD compatible flash chips (non-CFI).在这个实例中，使用了 Jedec 和 CFI 探测方式，来确保探测的成功。

```
int __init init_s3c2410(void)
{
        s3c2410_map.map_priv_1 = (unsigned long)ioremap(WINDOW_ADDR, WINDOW_SIZE);
        if (!s3c2410_map.map_priv_1) {
                printk(“Failed to ioremap/n”);
                return -EIO;
        }
        mymtd = do_map_probe(“jedec_probe”, &s3c2410_map);
        if (!mymtd)
        mymtd = do_map_probe(“cfi_probe”, &s3c2410_map);
        if (mymtd) {
                mymtd->module = THIS_MODULE;
                mymtd->erasesize = 0x20000; //擦除的大小 INTEL E28F128J3A-150 是
```

```
                128kb
                return add_mtd_partitions(mymtd, s3c2410_partitions,
                sizeof(s3c2410_partitions) / sizeof(struct mtd_partition));
        }
        iounmap((void *)s3c2410_map.map_priv_1);
        return -ENXIO;
}
```

到此，整个 Flash 建立了如下的分区：
前 8M 空间用于 bootloader，kernel，ramdisk，jffs2
后 8M 空间作为系统块设备或者字符设备使用的空间。

On-board memory

Many PC chipsets are incapable of correctly caching system memory above 64M or 512M. A driver exists which allows you to use this memory with the linux-mtd system.

PCMCIA 设备

PCMCIA flash (not CompactFlash but real flash) cards are now supported by the pcmciamtd driver in CVS.

Onboard NAND flash

NAND flash is rapidly overtaking NOR flash due to its larger size and lower cost; JFFS2 support for NAND flash is approaching production quality.

M-Systems' DiskOnChip 2000 and Millennium

The DiskOnChip 2000, Millennium and Millennium Plus devices should be fully supported, using their native NFTL and INFTL 'translation layers'. Support for JFFS2 on DiskOnChip 2000 and Millennium is also operational although lacking proper support for bad block handling.

CompactFlash

CompactFlash emulates an IDE disk, either through the PCMCIA-ATA standard, or by connecting directly to an IDE interface.
As such, it has no business being on this page, as to the best of my knowledge it doesn't have any alternative method of accessing the flash - you have to use the IDE emulation - I mention it here for completeness.

用户模块层(MTD User modules)

目前主要的用户态模块有 4 个: FTL, NFTL, JFFS and MTDBLOCK。

字符设备访问接口

将 Flash 设备模拟成字符设备。

在 drivers\mtd\mtdchar.c 文件中，实现了字符设备驱动接口(mtd_fops)和回调函数(notifier)。

```
static struct file_operations mtd_fops = {
        owner:          THIS_MODULE,
        llseek:         mtd_lseek,      /* lseek */
        read:           mtd_read,       /* read */
        write:          mtd_write,      /* write */
        ioctl:          mtd_ioctl,      /* ioctl */
        open:           mtd_open,       /* open */
        release:        mtd_close,      /* release */
};
```

以及模块的初始化和退出例程：

- init_mtdchar
- cleanup_mtdchar

mtd_ioctl 下判别了以下的 I/O 控制字：

MEMGETREGIONCOUNT，MEMGETREGIONINFO，MEMERASE ，MEMGETINFO，MEMWRITEOOB，MEMLOCK）

mtd_read:

直接直接调用 mtd_info 的 read 函数; mtd_write 直接直接调用 mtd_info 的 write 函数因此，字符设备接口跳过了 patition 这一层。

```
当 count>0 时{
        裁减本次操作大小 len 至 min(MAX_KMALLOC_SIZE,count)，
        申请一块大小为 MAX_KMALLOC_SIZE 的内核空间 kbuf，
        调用 mtd_info->read 将 MTD 设备中的数据读入 kbuf，
        将 kbuf 中的数据拷贝到用户空间 buf，
        count 自减
        释放 kbuf
}
```

mtd_write:

```
当 count>0 时{
```

```
        裁减本次操作大小 len 至 min(MAX_KMALLOC_SIZE,count)，
        申请一块大小为 MAX_KMALLOC_SIZE 的内核空间 kbuf，
        将用户空间 buf 中的数据拷贝到 kbuf，
        调用 mtd_info->write 将 kbuf 中的数据读入 MTD 设备，
        count 自减
        释放 kbuf
}
```

mtd_notifier

MTD 通知器，加入/删除 MTD 设备和原始设备时调用的函数，在设备层，当 MTD 字符设备或块设备注册时，如果定义了 CONFIG_DEVFS_FS，则会将一个 mtd_notifier 加入 MTD 原始设备层的 mtd_notifiers 链表，其中的函数会在两种情况下被调用,一是加入/删除新的 MTD 字符/块设备时，此时调用该 MTD 字符/块设备的 notifier 对下层所有的 MTD 原始设备操作一遍，二是加入/删除新的 MTD 原始设备时，此时调用所有的 notifier 对该原始设备执行一遍。

```
struct mtd_notifier {
void (*add)(struct mtd_info *mtd);       加入时调用
void (*remove)(struct mtd_info *mtd);    删除时调用
        struct mtd_notifier *next;       指向 mtd_notifiers 队列中的下一个
                                         mtd_notifier
};

static void mtd_notify_add(struct mtd_info* mtd)
static void mtd_notify_remove(struct mtd_info* mtd)
```

块设备访问接口

块设备访问接口可以将 Flash 模拟成一个块设备。主要原理是将 Flash 的 erase block 中的数据在内存中建立映射，然后对其进行修改，最后擦除 Flash 上的 block，将内存中的映射块写入 Flash 块。整个过程被称为 read/modify/erase/rewrite 周期。但是，这样做是不安全的，当下列操作序列发生时，read/modify/erase/poweroff，就会丢失这个 block 块的数据。块设备模拟驱动按照 block 号和偏移量来定位文件，因此在 Flash 上除了文件数据，基本没有额外的控制数据。

在 linux 中，driver/mtd/mtdblock.c 文件实现了这个功能。

MTD 设备层的块设备的相关数据结构和程序，其中定义了 MTD 块设备的 notifier 和 MTD 块设备的结构 mtdblk_dev 类型的 mtdblks 数组，该数组中的每个成员与 mtd_table 数组中的每个成员一一对应。注：在分区完毕后，每个分区都作为一个 device 被加入到 mtd_table 中。我

们的例子里有 5 个分区，也就有 5 个设备。每个设备还有自己的缓冲区。

```
struct mtdblk_dev {
        struct mtd_info *mtd; /* Locked */       下层原始设备层的 MTD 设备结构
        int count;
        struct semaphore cache_sem;
        unsigned char *cache_data;       缓冲区（与 MTD 设备块对齐）
        unsigned long cache_offset;      缓冲区内数据在原始设备层 MTD 设备内的偏移
        unsigned int cache_size;         缓冲区大小（通常被设置为 MTD 设备的
erasesize）
        enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;
        缓冲区状态
}*mtdblks[MAX_MTD_DEVICES];
```

do_cached_write
如果缓冲区大小为 0，则调用 mtd_info->write()直接向 MTD 设备中写，
当（len>0）{
        如果是整块 block，则直接向 MTD 设备中写
        否则{
                先调用 write_cached_data()将缓冲内的数据写入 MTD 设备
                在调用 mtd_info->read 从 MTD 设备中将对应块的数据读入缓冲
                调用 erase_write 擦除实际 Flash 块
                将 buf 的数据（非整块）写入缓冲
        }
        len 自减
}

write_cached_data
调用 erase_write。将缓冲的数据写入 Flash。

erase_write
擦写函数，声明一个 erase_info 变量 erase 并给其赋值；调用原始设备层函数 mtd_info->erase(mtd,erase)；调用原始设备层函数 mtd_info->write()。

erase_callback
当擦写完后的回调函数。唤醒等待队列。

handle_mtdblock_request
处理对 MTD 块设备的请求

Flash Translator Layer(FTL)

- 模拟了一个标准块设备
- 在 FTLH 之上，还需要使用文件系统

虚拟块设备

FTL 是在传统的文件系统，如 DOS BPB/FAT，和 flash 存储器之间的一层转换器。通过 FTL 的转换，文件系统可以把 Flash 看成是一个块设备。

如图 2 虚拟块设备显示，FTL 将整块 Flash 分割为一个个 block(size=sector),当文件系统要求对 block 14 进行操作时，FTL 将命令转换为对 Flash 上第 21 块 block 进行的操作。

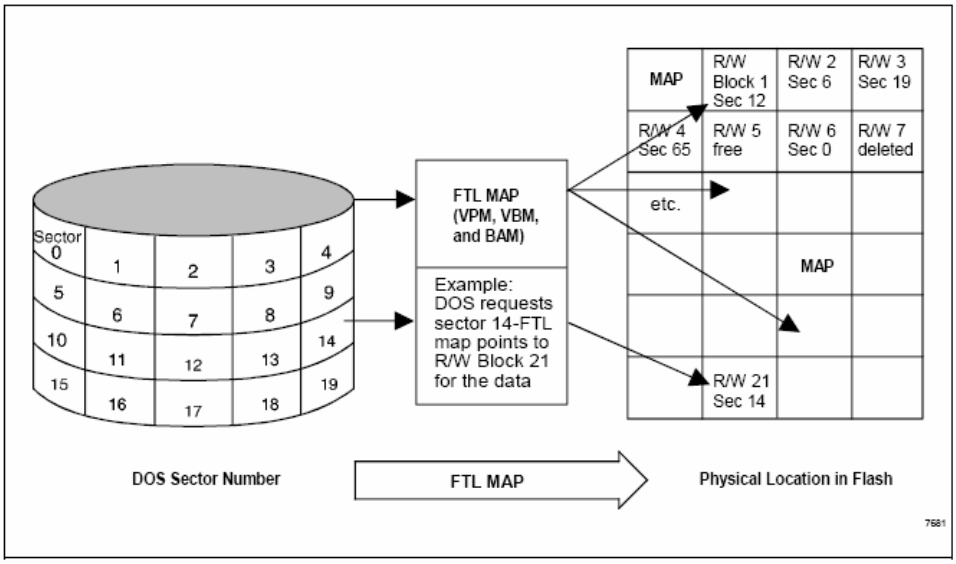

图 2 虚拟块设备

擦除单元(Erase Unit)

每个擦除单元有若干个擦除区(Erase Zone)组成。所谓擦除区是指 Flash 上能执行擦除操作的最小连续地址范围。如果有两片 8-bit 的 Flash 极联成 16-bit 的存储器的话，那么擦除单元由两个擦除区组成，每片 Flash 上各一个。一般擦除区就是由 Flash 芯片手册指定大小的，在 Intel 的芯片上，一般为 32k 或者 64k。

每个擦除单元又可以被若干个大小相等的读写块。这些读写块的大小和文件系统的簇的大小应该是一致的。在 DOS 上，簇为 512 字节，那么读写块也应当为 512 字节。

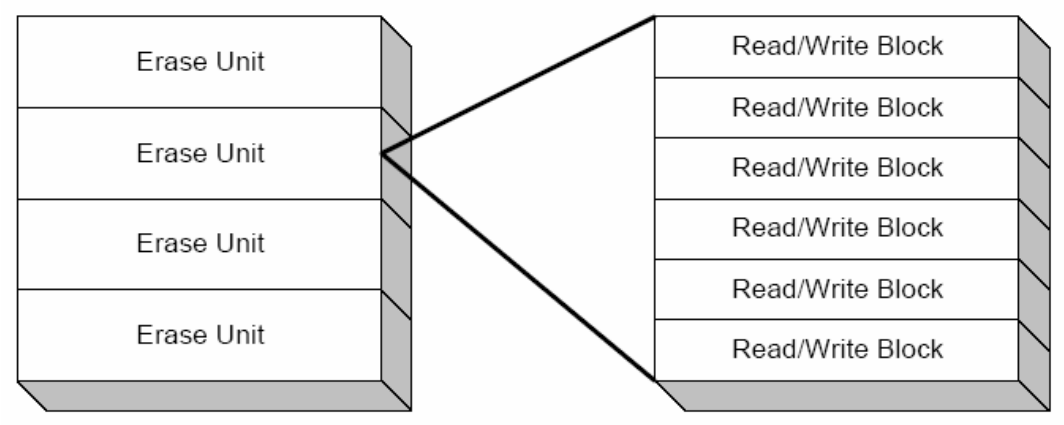

図 3 擦除单元

擦除单元头部(Erase Unit Header,EUH)

所有擦除单元的 0 偏移处都存放着擦除单元头部的数据结构(也有例外的存放地址)erase_unit_header_t。这个数据结构存放着对整个擦除单元的描述信息。

具体信息如下表所示：

Offset	Field	Sz	Description	Example: 4-Meg F008-based Card	
				HEX	Comments
0	LinkTargetTuple	5	PCMCIA Link Target tuple	13 03 43 49 53	(..CIS)
5	DataOrganization Tuple	10	PCMCIA Data Organization tuple	46 39 00 46 54 4C 31 30 30 00	(F9.FTL100.)
15	NumTransferUnits	1	Number of transfer units in partition	01	
16	Reserved	4	Reserved	xx xx	
20	LogicalEUN	2	Logical Erase Unit Number of this block	FF FF	
22	BlockSize	1	Size of a Read/Write Block	09h	2^{9h} = 200h (512 dec)
23	EraseUnitSize	1	Size of an Erase Unit	11h	2^{11h} = 20000h (128k dec)
24	FirstPhysicalEUN	2	Physical Erase Unit where FTL partition begins	00 00	
26	NumEraseUnits	2	Total number of Erase Units in partition	20 00	(0020)
28	FormattedSize	4	Total formatted size of partition	00 10 3C 00	(003C1000)
32	FirstVMAddress	4	First virtual address physically mapped in VBM page on media	00 00 01 00	(00100000)
36	NumVMPages	2	Total number of VBM pages	3D 00	(003D)
38	Flags	1	Special bit-mapped flags	00	
39	Code	1	Code designating EDAC type	FF	None
40	SerialNumber	4	Partition serial number	00 00 00 00	
44	AltEUHOffset	4	Offset of alternative EUH	00 00 00 00	
48	BAMOffset	4	Offset of BAM from start of EUH	44 00 00 00	(00000044)
52	Reserved	12	Reserved	xx	

表 1 擦除单元头部

块分配映射(Block Allocation Map)

当块被分配并且操作后，需要对整个 Flash 中的块使用情况进行跟踪。按照 FTL 的规定中，块分配的跟踪信息可以按照两种方式存储。其中最为常用的方式为：在擦除单元中分配一个一个数组，存储这个擦除单元中所有块的使用情况。这个数组被称为 Block Allocation Map (BAM)。它的每个成员为一个 word 的标识。在 EUH 中的 Flags 域中指定了使用哪种方式来存储块分配信息。

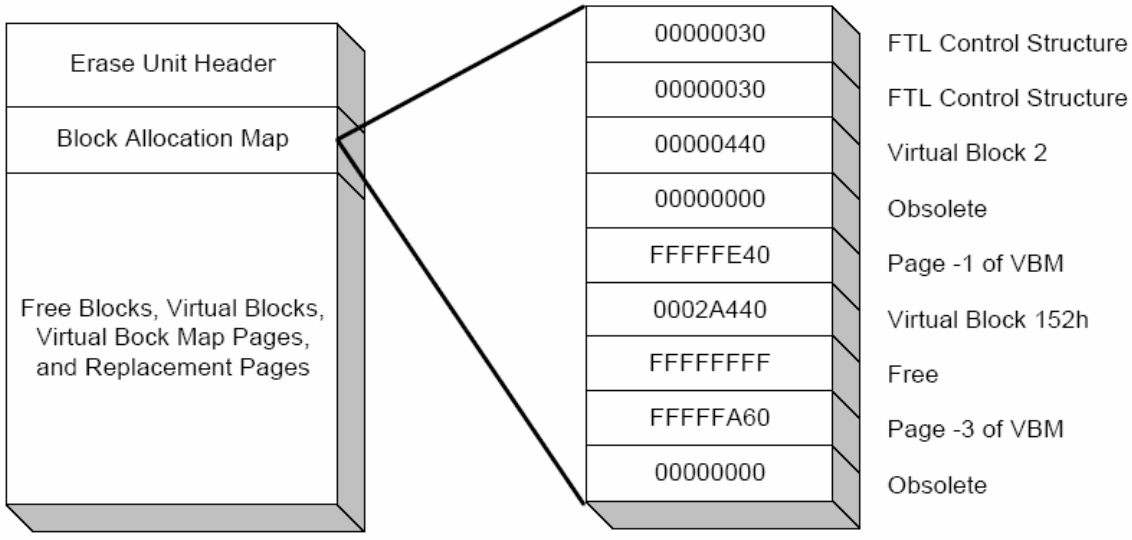

图 4 块分配映射

下表是对各种类型的定义,详细内容请参考 Intel FTL Spec。

32-Bit BAM Entry	Meaning	Description
0xFFFFFFFF	Free	Read/Write block is available for writing.
0xFFFFFFE or 0x00000000	Deleted	Data in block is invalid. This Read/Write block must be erased before it may be reused. The value 0xFFFFFFFE indicates that a write operation on this block was interrupted before completion. The value 0x00000000 (all bits programmed) indicates that the data in this block is obsolete.
0x00000070	Bad	This Read/Write block is unusable and may not be written to or read from.
0x00000030	Control	This Read/Write block contains FTL control structures (e.g., EUH, BAM, ECCs, etc.).
0xXXXXXX40	Data or Map Page	Contains data or a Virtual Map Page.
0xXXXXXX60	Replacement Map Page	Replacement Page for a Virtual Map Page.

图 5 BAM 入口值

虚拟块虚拟地址到逻辑地址的映射

FTL 使用 Virtual Block Map (VBM)来完成虚拟地址到逻辑地址的映射。VBM 由页组成，这些页的大小和操作系统的虚拟内存页大小相等。每个 VBM 页都是有 32-bit 入口组成的数组，这 32-bit 的地址指向实际块所在的地址。VBM 入口是 little-endian 格式的。FTL 将从操作系统传入的 block 号作为数组的 index 来读取数据。

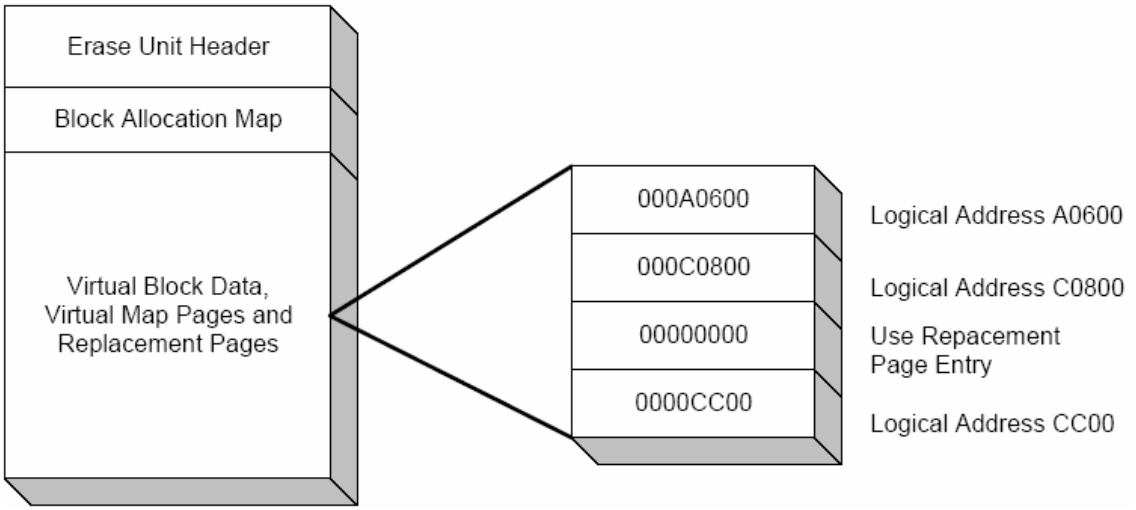

図 6 Virtual Block Map (VBM)

VBM 通常是在 Flash 上被存储和维护的，但是通过更改 First Virtual Mapped Address 的值，可以将部分的 VBM 交给系统在内存中进行维护。

First Virtual Mapped Address

First Virtual Mapped Address (FirstVMAddress)设置了 VBM 可以映射的最低虚拟地址。在这个地址以下的虚拟地址的映射关系将不被存放在 Flash 上。这样，没有被包含的地址映射就需要在系统内存中重新建立。这样做的理由是，虚拟空间的低地址部分会经常被 FAT 更改，如果在 Flash 上建立映射，更新数据操作将花费很多时间。其次，将映射关系部分地存放在内存中，也加快了处理速度。但是这样做会是内存开销增大。

Logical to Physical Mapping

在擦除单元中，LogicalEUN 字段制定了其下所有块的逻辑号。

逻辑地址有两部分组成：

- 逻辑设备号
- 擦除单元中的偏移量

在设备初始化的时候，FTL 把 LogicalEUN 和 PhysicalEUN 进行映射，后者是物理地址。

读操作

FTL 将系统传入的块设备号作为 index 值，查找 VBM 的数组，得到逻辑地址，然后将逻辑地址转换为物理地址，读取数据。

写操作

在写操作执行之前，FTL 必须先找到一块空闲的 block,如果找不到空闲的块，那么会启动擦除单元恢复操作，来清空一些被标记为等待 erase 的块。如果找到空闲块，将空闲块的 BAM 入

口点的值设置成 0xFFFFFFFE（代表处于写操作过程中）。这样即便发生中断，这个块可以被后续处理。然后，将文件系统传入的块数据写入这个块，写完后，将 BAM 中的值更新指向自己。原本的 block 被标记为等待 erase。

擦除单元的恢复

在 FTL 中，写操作不断进行，而在执行写操作时，需要分配空闲块。这样，空闲块在一定阶段会被分配完毕。在这个时候，需要对那些被暂时废弃的，标记为等待 erase 的块执行 erase 操作。由于硬件原因，擦除必须以 erase unit 为单位进行，虽然 erase zone 才是最小擦除单位。

这样，需要寻找一个交换单元来保存擦除单元中有用的数据，而仅仅擦除无效的数据。在完成恢复操作后，交换单元的 LogicalEUN 被设置成被恢复单元的 LogicalEUN，而被恢复单元被设置成交换单元。

FTL AVAILABILITY

M-System TrueFFS* (v3.2 and up are FTL)
M-Systems, Inc. M-Systems, Ltd

SCM SwapFTL* (v3.0 and up are FTL)
SCM Microsystems, Inc. SCM Microsystems, GmbH

SystemSoft FTL
SystemSoft, Inc. (Main) SystemSoft, Inc. (West)

Journalling Flash File System, v2

使用 JFFS2 的原因和目的:

- MTD 的字符接口和块设备接口都存在相当的缺陷(具体将在<比较与分析>章节中讨论)
- FTL 的确做了一定的优化,减少了写操作时间,增加了安全性.但是由于 M-System 限制了 FTL 只能在 PCMCIA 设备上使用。此外，由于 FTL 只是一个解释层，因此需要在其上再使用标准的文件系统，这样的双层结构使运行性能有所降低。

因此,Axis Communications AB 推出了一种单层次,安全高效的文件系统 JFFS2(kernel 2.0)。随后,由 Sweden 的程序员移植到 kernel 2.4 中,再由 RedHat 从 kernel 2.4backport 到 kernel 2.2 进行维护。

Log Structured 文件系统

JFFS2 是基于 Log Structured 结构的文件系统。下面简单介绍一下 Log Structured 文件系统的特性。

Log Structured 文件系统的基本原理

- 数据在存储器上没有固定的存放位置。
- 存储以 node 为单位。每个 node 代表了一次用户的操作。node 是顺序存放在存储器上的。所有的 node 代表了所有有效的用户操作(有效是指用户操作的数据在存储器上仍然有效而没有被其他操作的数据所覆盖)。
- node 中包含了以下信息：version 号码(代表用户操作的顺序，最先的操作号码最小)；file 文件名(该 node 所归属的文件名)；offset(node 数据在存储器中的偏移量)；LEN(node 数据长度)

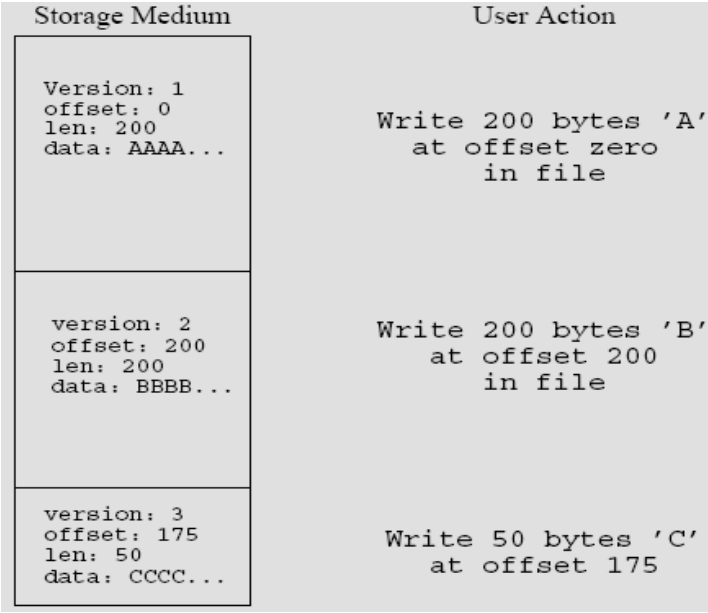

図 7 Log Structured

Log Structured 的回滚

由于 Log Structured 只记录了每次用户的操作，那么在对文件系统进行操作之前，必须将这些操作按照时间顺序回滚一边，这样才能建立文件系统的结构。如図 8 Log Structured 的回滚所示：

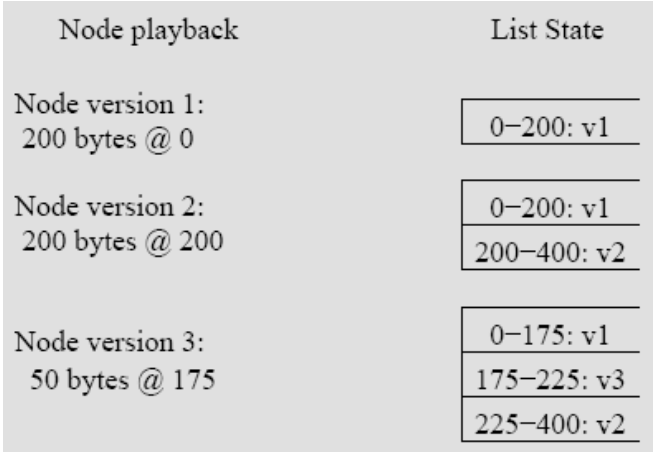

図 8 Log Structured 的回滚

脏 node 和回收机制

越早的操作留下的数据越可能被后续的操作所覆盖。这样，先前的 node 结构就变得无效，形成了脏 node。而随着操作不断线性占用存储器空间，必然会到达存储器地址空间的尽头。在这种情况下，就需要回收那些脏 node。

假设如図 9 的存储器。在这个时刻需要执行回收 node 操作。

図 9

应当从存储器首地址开始将有效的 node 转移到存储器最末的有效区的后面。如図 10。这样就可以在头部形成一个连续的无效存储区。然后在对无效存储区执行 erase 操作，形成如図 11。

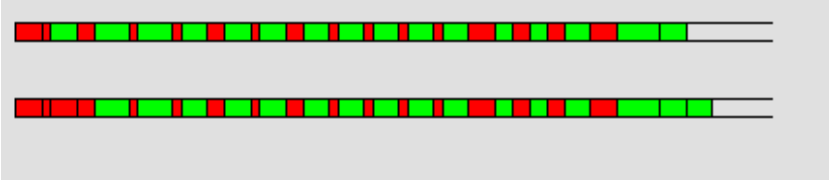

図 10

図 11

原始 Log Structured 的缺陷

这也就是指 JFFS 的缺陷。

1 在下面的情况下，回收机制将变得非常低效。由于严格的线性回收机制，使很多数据搬运操作浪费在有效的 node 上。

図 12 JFFS 低效的回收机制

2 没有把数据进行压缩处理，占用过多的 Flash 空间

3 Parent node 的描述信息和数据存放在一起，占用了空间；还妨碍了 POSIX 硬连接。

这样就诞生了 JFFS2。

JFFS2 的改进

JFFS2 原本只想在 JFFS 基础上增加一些数据压缩的功能，然而由于其他原因使得开发者们认为重新实现这个文件系统为好。

它做了以下改进：

- 改进 log 结构
- 优化的回收机制
- 不同类型的 node
- 优化的存储器使用方式
- 数据压缩功能

log 结构

log 结构通过 block 的链表来记录存储器的使用情况，这里的 block 是指 Flash 手册规定的最小擦除连续空间。在 JFFS 中，log 结构不是很突出，原因是所有的 node 都是按照线性顺序存放，这就是 JFFS 的结构.

在 JFFS2 中就不一样了。它将所有的 erase block 都看成是独立的，node 的存放也是根据 block 的分组来确定地址，而不是一味的线性存放。因此，产生了三种链表：

clean_list：这个链表中所有的 block 都是包含有效数据的 node。
dirty_list：这个链表中所有的 block 部分包含有效数据的 node，其余为无效数据 node，即数据已经被其他操作的数据覆盖。
free_list：这个链表中所有的 block 都是空闲等待分配使用的 node。
这些 node 可能回跨越 blocks。这种 log 结构，会给回收机制带来很大的性能提高。

回收机制

通常情况下，从 dirty_list 取出 block 进行 erase。当 clean_list 中的 block 包含了脏数据后，也可以将其取出，执行 erase。这样避免了顺序的回收扫描。

不同类型的 node

JFFS 只允许一种 node 的存在，而 JFFS2 则允许不同类型的 node 共存。

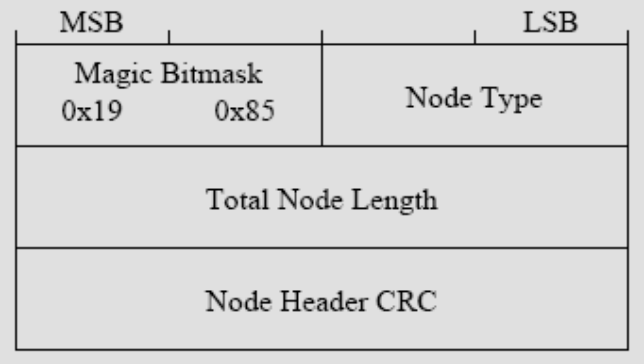

图 13 JFFS2 Common Node Header

Magic Bitmask 唯一地标识了系统中的 node。
出于兼容性考虑，Node Type 有以下定义：
JFFS2_FEATURE_INCOMPAT：on finding a node with this feature mask which is not explicitly supported, a JFFS2 implementation must refuse to mount the file system.

JFFS2 FEATURE ROCOMPAT:a node with this feature mask may be safely ignored by an implementation which does not support it, but the file system must not be written to.

JFFS2 FEATURE RWCOMPAT DELETE:an unsupported node with this mask may be safely ignored and the _le system may be written to. Upon garbage collecting the sector in which it is found, the node should be deleted.

JFFS2 FEATURE RWCOMPAT COPY：an unsupported node with this mask may be safely ignored

and the file system may be written to. Upon garbage collecting the sector in which it is found, the node should be copied intact to a new location.

正常情况下，Node Type 有以下定义：

JFFS2_NODETYPE_INODE：这类 node 和 JFFS 中的原始 node 最相似，代表了一个 inode 的信息。但是，它不包含 parent node 的信息以及文件名。Node 中的数据可以被加密和解密。由于在执行 readpage()操作时，需要解密数据，为了加快操作的速度，node 中的数据量被限制在页大小之内。

JFFS2_NODETYPE_DIRENT:代表一个目录,或者是指向一个 inode 的链接。它包含了目录的 inode 号。如果这是一个链接的话，还会包含被指向的 inode 的号码和链接名。一个链接可以用下面的方式解除。建立一个同名的 dirent node，在 target inode 中设置为 0。在 version 中设置更高的值。

JFFS2_NODETYPE_CLEANMARKER：这类 node 被直接写入刚完成擦除的 block 中，来代表这个块刚被擦除完毕。

Node 的管理

在系统内存中可以存放部分的 node 管理信息。对于设备中的每一个 inode，内存中都有一个 jffs2_inode_cache 结构体与其对应。这个结构体中存放了 inode 号、链接数、指向数据 node 链表首地址的指针。这些结构存放在一个 hash 表中。Hash 表的定位方法非常原始，用 inode 号与 hash 表大小取模。

存储器上每个数据 node 在内存中都由一个 jffs2_raw_node_ref 对应。这个结构相对比较小，因为数据 node 会有很多，这样会使系统内存加大开销。这个结构中，有两个重要的指针：next_in_ino 指向 inode 中的下一个数据 node；next_phys 指向 Flash 上的下一个 node。

对于读一个 inode，JFFS2 会先扫描这个 inode 的数据 node 链表。这个链表尾部的 node 的 next_in_ino 指针将指向 inode 自身，并且 offset 字段被设置成 NULL，这就代表了一个链表的结束。这样，就建立起了一个 inode 的数据 map。

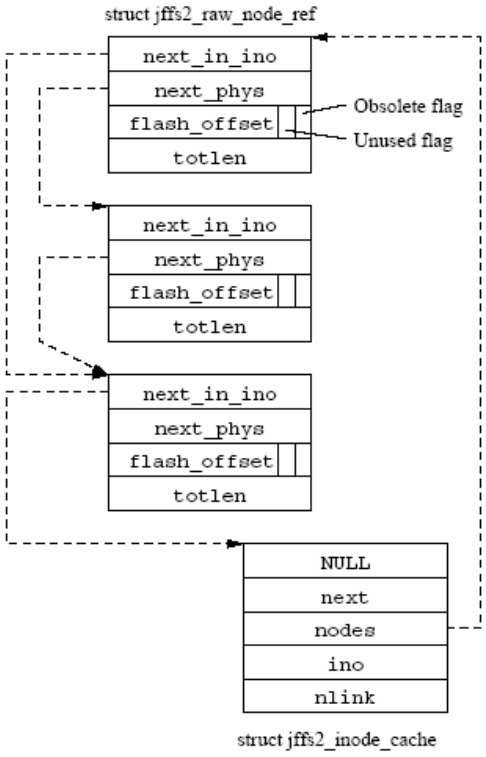

图 14 inode 数据 map

设备节点

通过 mknod 在/dev 子目录下建立 MTD 字符设备节点（主设备号为 90）和 MTD 块设备节点（主设备号为 31），通过访问此设备节点即可访问 MTD 字符设备和块设备。

比较与分析

在以上介绍块设备访问接口，FTL 转换器，JFFS2 文件系统。这三种模块都有各自的优点和缺点。

块设备访问接口

对于块设备访问接口来说，由于遵循 read-modify-erase-rewrite 的操作序列，大大增加了写操作的时间，同样也增加了突然断电的可能性。此外，由于某个文件内部的偏移量被定死在 Flash 的某个地址上，如果这个文件被频繁执行写操作，则很可能发生 Flash Block wear out 的现象。因此，在写操作安全性、效率及 Flash 使用寿命上块设备访问接口处于劣势。

但是，由于块设备访问接口在 Flash 上不需要额外的控制信息存放空间，因此节约了 Flash 的空间，对于只读系统来说，不失为一种高效的访问方法。

FTL

FTL 只是一个转换层，它做的工作其实和块设备驱动做的工作的目的一致，模拟一个块设备。但是它们的实现方式不同。在 FTL 中执行写操作时，使用空闲块来替代旧的数据块，这样不

但省去了擦除的时间开销，降低突然断电的风险，而且将擦除操作平均分配到了 Flash 上的各个块,避免了由于文件频繁写操作而带来的 Flash 块频繁擦除。这样就避免了块的 wear out。因此，在 Flash 使用寿命方面 FTL 较块设备访问接口更具优势。

然而，由于 FTL 只能完成文件系统的部分功能，因此在其上层仍然需要实现完整的文件系统机制。这样一来，就形成了两层文件系统，或者说文件系统+转换层的局面，这会使得文件操作变慢。

此外，FTL 需要在 Flash 上建立控制信息，这将使用掉一部分 Flash 空间。

最后，FTL 还有版权的问题。

JFFS2

JFFS2 完全抛弃了前两种的设计思想，它是一个完整的 log 机制的文件系统，并且直接作用于 Flash。这样，于 FTL 相比省去了转换层的操作；和块设备访问接口相比，解决了 wear out 问题、减少了端点风险。

但是,JFFS2 不仅仅要在 Flash 上为控制信息分配空间,还要在内存中也建立相应的映射关系，这样就在两个层面上增加了资源的开销。FTL 虽然也在内存中建立映射，但是这只是一部分而且也是可以调节的。

参考书籍

[FTL] Intel Corporation, Understanding the Flash Translation Layer (FTL) Speci_cation, (1998). http://developer.intel.com/design/comp/applnots/297816.htm

[LFS] Mendel Rosenblum and John K. Ousterhout,The Design and Implementation of a Log-Structured File System, ACM Transactions on Computer Systems 10(1) (1992) pp. 26{52.

David Woodhouse, Red Hat, Inc.,Jffs2.pdf, http://sources.redhat.com/jffs2/