这一章学习 NAND FLASH 驱动，和前面的 LCD 驱动一样， Linux 内核里面已经有了一套代码是支持这类存储器的驱动，叫做 MTD(memory technologydevice 内存技术设备)。 他帮我们做了很多工作，从而使增加新的内存设备驱动程序变得更简单。但是这一层代码看起来就不怎么简单了。这一章我们先来了解一下 linux 内核的 MTD 子系统，然后再开始写我们的 nand flash 驱动。

### MTD 子系统

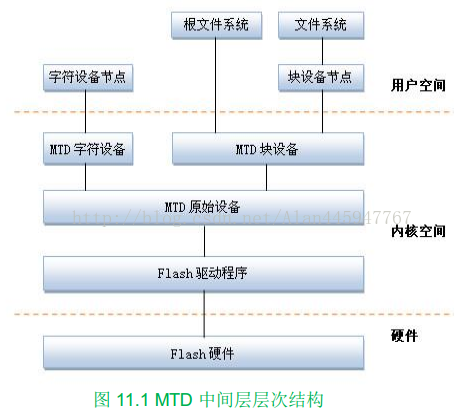
MTD 子系统是在硬件和文件系统层之间的提供了一个抽象的接口，用来访问内存设备（如： ROM、 flash）的中间层，它将内存设备的共有特性抽取出来，从而使增加新的内存设备驱动程序变得更简单。MTD 的源代码都在**/drivers/mtd** 目录中。MTD 中间层细分为四层，按从上到下依次为：设备节点、 MTD 设备层、 MTD原始设备层和硬件驱动层。 MTD 中间层层次结构如图 11.1 所示 ：

#### MTD 子系统

MTD 子系统是在硬件和文件系统层之间的提供了一个抽象的接口，用来访问内存设备（如： ROM、 flash）的中间层，它将内存设备的共有特性抽取出来，从而使增加新的内存设备驱动程序变得更简单。

MTD 的源代码都在/drivers/mtd 目录中。

MTD 中间层细分为四层，按从上到下依次为：设备节点、 MTD 设备层、 MTD原始设备层和硬件驱动层。 MTD 中间层层次结构如图 11.1 所示 ：



#### 设备节点层

通过 mknod 在/dev 子目录下建立 MTD 字符设备（规定主设备号 90）和块设备号（规定的主设备号 31 ）

#### MTD 设备层

这一层分为 MTD 字符设备和 MTD 块设备。 **MTD 字符设备的定义在driver/mtd/mtdchar.c 中，通过 file\_operations mtd\_fops 结构体，注册 lseek，open ， close,read,write,ioctl 等 设 备 操 作 函 数 ； MTD 块 设 备 的 定 义 在driver/mtd/mtdblock.c 中，与字符设备相似，注册设备操作函数的结构体为mtd\_blktrans\_ops，描述块设备的结构体则为 mtdblk\_dev。**

#### MTD 原始层

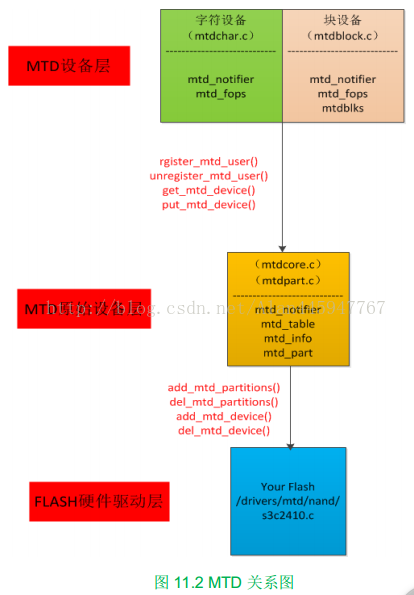
MTD 原始层由两部分组成，**一部分是 MTD 原始设备的通用代码，另一部分是各个特定的 FLASH 的数据，**例如分区。

#### 硬件驱动层

linux MTD 设备的驱动程序，**根据不同的 MTD 设备他们存放在不同目录，如 NAND FLASH 驱 动 存 放 在 /driver/mtd/nand, 而 NOR FLASH 存 放 于/driver/mtd/chips.**

### Linux MTD 系统接口

Linux MTD 各层的接口函数和关键结构体如图 11.2 所示：我们这一小节先来看一下一些结构体，具体的函数我们下一节再结合一个内核包含的 nand flash 驱动来分析。这一节对下一节也是很有紧密联系的哦。



#### MTD 原始设备层

先从 MTD 原始设备层看，因为他正好处于中间层，承上启下，比较关键。先来看第一个结构体： **mtd\_info**

##### mtd\_info 结构体

**mtd\_info 是表示 MTD 原始设备的结构体，每个分区也被认为是一个mtd\_info.**例如有一个 MTD 设备，他有两个分区，系统认为他有两个 mtd\_info。这些 mtd\_info 的指针存放在名为 mtd\_table 的数组里。

mtd\_info 在 include/linux/mtd/mtd.h 里：

struct mtd\_info {

u\_char type; /\* MTD 类型，包括 MTD\_NORFLASH,MTD\_NANDFLASH

等(可参考 mtd-abi.h) \*/

uint32\_t flags; /\* MTD 属性标志， MTD\_WRITEABLE,MTD\_NO\_ERASE 等

(可参考 mtd-abi.h) \*/

uint64\_t size; /\* MTD 设备的大小 \*/

uint32\_t erasesize; /\* MTD 设备的擦除单元大小，对于 NandFlash 来说就

是 Block 的大小 \*/

uint32\_t writesize; /\* 写大小,对 nandFlash 为一页 \*/

uint32\_t oobsize; /\* OOB 字节数 \*/

uint32\_t oobavail; /\* 可用的 OOB 字节数 \*/

unsigned int erasesize\_shift; /\* 默认为 0，不重要 \*/

unsigned int writesize\_shift;/\* 默认为 0，不重要 \*/

unsigned int erasesize\_mask; /\* 默认为 1 ，不重要 \*/

unsigned int writesize\_mask; /\* 默认为 1 ，不重要 \*/

const char \*name; /\* 名字， 不重要\*/

int index; /\* 索引号，不重要 \*/

int numeraseregions; /\* 通常为 1 \*/

struct mtd\_erase\_region\_info \*eraseregions;/\* 可变擦除区域 \*/

void \*priv;/\*设备私有数据指针,对 NandFlash 来说指 nand\_chip 结构体\*/

struct module \*owner; /\* 一般设置为 THIS\_MODULE \*/

/\* 擦除函数 \*/

int (\*erase) (struct mtd\_info \*mtd, struct erase\_info \*instr);

/\* 读写 flash 函数 \*/

int (\*read) (struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen,u\_char \*buf);

int (\*write) (struct mtd\_info \*mtd, loff\_t to, size\_t len, size\_t \*retlen, constu\_char \*buf);

/\* 带 oob 读写 Flash 函数 \*/

int (\*read\_oob) (struct mtd\_info \*mtd, loff\_t from,struct mtd\_oob\_ops \*ops);

int (\*write\_oob) (struct mtd\_info \*mtd, loff\_t to,struct mtd\_oob\_ops \*ops);

int (\*get\_fact\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf, size\_t len);

int (\*read\_fact\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len,size\_t \*retlen, u\_char \*buf);

int (\*get\_user\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf, size\_t len);

int (\*read\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len,size\_t \*retlen, u\_char \*buf);

int (\*write\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len,size\_t \*retlen, u\_char \*buf);

int (\*lock\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len);

int (\*writev) (struct mtd\_info \*mtd, const struct kvec \*vecs, unsigned long count, loff\_t to, size\_t \*retlen);

int (\*panic\_write) (struct mtd\_info \*mtd, loff\_t to, size\_t len, size\_t \*retlen,const u\_char \*buf);

/\* 同步\*/

void (\*sync) (struct mtd\_info \*mtd);

/\* Chip-supported device locking \*/

int (\*lock) (struct mtd\_info \*mtd, loff\_t ofs, uint64\_t len);

int (\*unlock) (struct mtd\_info \*mtd, loff\_t ofs, uint64\_t len);

/\* 电源管理函数 \*/

int (\*suspend) (struct mtd\_info \*mtd);

void (\*resume) (struct mtd\_info \*mtd);

/\* 坏块管理函数 \*/

int (\*block\_isbad) (struct mtd\_info \*mtd, loff\_t ofs);

int (\*block\_markbad) (struct mtd\_info \*mtd, loff\_t ofs);

void (\*unpoint) (struct mtd\_info \*mtd, loff\_t from, size\_t len);

unsigned long (\*get\_unmapped\_area) (struct mtd\_info \*mtd,

unsigned long len,

unsigned long offset,

unsigned long flags);

struct backing\_dev\_info \*backing\_dev\_info;

struct notifier\_block reboot\_notifier; /\* default mode before reboot \*/

/\* ECC 状态信息 \*/

struct mtd\_ecc\_stats ecc\_stats;

int subpage\_sft;

struct device dev;

int usecount;

int (\*get\_device) (struct mtd\_info \*mtd);

void (\*put\_device) (struct mtd\_info \*mtd);

};

在上面提到几个名词， ECC 和 OOB，他们是什么呢?

##### OOB

NAND Flash 每一页大小为(512+16)字节(还有其他格式的 NANDFlash，比如每页大小为(256+8)、 (2048+64)等)，其中的 512 字节就是一般存储数据的区域， 16 字节称为 OOB(Out OfBand)区。通常在 OOB 区存放坏块标记、前面 512字节的 ECC 较验码等。

cramfs、 jffs2 文件系统映像文件中并没有 OOB 区的内容，如果将它们烧入NORFlash 中，则是简单的“平铺”关系；如果将它们烧入 NAND Flash 中，则NANDFlash 的驱动程序首先根据 OOB 的标记略过坏块，然后将一页数据(512字节)写入后，还会计算这 512 字节的 ECC 较验码，最后将它写入 OOB 区，如此循环。

cramfs、 jffs2 文件系统映像文件的大小通常是 512 的整数倍。而 yaffs 文件系统映像文件的格式则跟它们不同，文件本身就包含了 OOB 区的数据(里面有坏块标记、 ECC 较验码、其他 yaffs 相关的信息)。所以烧写时，不需要再计算 ECC 值，首先检查是否坏块(是则跳过)，然后写入 512 字节的数据，最后写入 16 字节的 OOB 数据，如此循环。 yaffs 文件系统映像文件的大小是(512+16)的整数倍。

注意：烧写 yaffs 文件系统映像时，分区上第一个可用的(不是坏块)块也要跳过。

##### ECC

ECC 的全称是 Error Checking and Correction，是一种用于 Nand 的差错检测和修正算法。如果操作时序和电路稳定性不存在问题的话， NAND Flash 出错的时候一般不会造成整个 Block 或是 Page 不能读取或是全部出错，而是整个Page（例如 512Bytes）中只有一个或几个 bit 出错。 ECC 能纠正 1 个比特错误和检测 2 个比特错误，而且计算速度很快，但对 1 比特以上的错误无法纠正，对 2 比特以上的错误不保证能检测。

校验码生成算法： ECC 校验每次对 256 字节的数据进行操作，包含列校验和行校验。对每个待校验的 Bit 位求异或，若结果为 0，则表明含有偶数个 1；若结果为 1 ，则表明含有奇数个 1 。列校验规则如表 1 所示。 256 字节数据形成256 行、 8 列的矩阵，矩阵每个元素表示一个 Bit 位。

算法的 c 语言实现在 drivers/mtd/nand/nand\_ecc.c。

OOB 是每个页都有的数据， 里面存的有 ECC（当然不仅仅）；而 BBT 是一个 FLASH 才有一个；针对每个 BLOCK 的坏块识别则是该块第一页 spare area的第六个字节。

ECC 一般每 256 字节原始数据生成 3 字节 ECC 校验数据，这三字节共 24比特分成两部分： 6 比特的列校验和 16 比特的行校验，多余的两个比特置 1 。（ 512生成两组 ECC，共 6 字节）

当往 NAND Flash 的 page 中写入数据的时候，每 256 字节我们生成一个ECC 校验和，称之为原 ECC 校验和，保存到 PAGE 的 OOB （ out- of-band）数据区中。其位置就是 eccpos[]。校验的时候，根据上述 ECC 生成原理不难推断：将从 OOB 区中读出的原 ECC 校验和新 ECC 校验和按位异或，若结果为 0，则表示不存在错（或是出现了 ECC 无法检测的错误）；若 3 个字节异或结果中存在 11 个比特位为 1 ，表示存在一个比特错误，且可纠正；若 3 个字节异或结果中只存在 1 个比特位为 1 ，表示 OOB 区出错；其他情况均表示出现了无法纠正的错误。

##### mtd\_partition 结构体

存放在内核文件 include/linux/mtd/partitions.h 中

struct mtd\_partition {

char \*name; /\* 分区名 ，自己随便命名 \*/

u\_int32\_t size; /\* 分区大小 \*/

u\_int32\_t offset; /\* 主 MTD 空间内的偏移地址 \*/

u\_int32\_t mask\_flags; /\* 掩码标志 ,不重要 \*/

struct nand\_oobinfo \*oobsel; /\* OOB 布局 \*/

struct mtd\_info \*\*mtdp; /\* 存储 MTD 对象的指针 \*/

}

##### mtd\_part 结构体

在 driver/mtd/mtdpart.c 这里还有一个结构体要说明

struct mtd\_part {

struct mtd\_info mtd; /\* 分区信息, 大部分由 master 决定 \*/

struct mtd\_info \*master; /\* 分区的主分区 \*/

uint64\_t offset; /\* 分区的偏移地址 \*/

struct list\_head list; /\* 将 mtd\_part 链成一个链表 mtd\_partitons \*/

};

看到这里，有同学就要问了，这里的 mtd\_part 和 mtd\_partition 的结构定义好 像 喔 ， 他 们 之 间 有 什 么 关 系 ， 有 什 么 不 同 吗 ？ 等 一 下 后 面 讲add\_mtd\_partitions 函数你就会知道啦。

#### MTD 设备层

MTD 设备主要有 MTD 字符设备（ mtdchar.c）和 MTD 块设备(mtdblock.c)11.2.2.1 mtd\_notifier 结构体MTD 设备通知结构体， notifier 中文 就是通告 的意思 ^-^ ，存 放在include/linux/mtd/mtd.h

struct mtd\_notifier {

/\* 加入 MTD 原始/字符/块设备时执行 \*/

void (\*add)(struct mtd\_info \*mtd);

/\* 移除 MTD 原始/字符/块设备时执行 \*/

void (\*remove)(struct mtd\_info \*mtd);

struct list\_head list;//list 是双向链表，定义在 include/linux/list.h

};

有没有发现这个结构体在设备层和原始设备层都有出现，对的，就是通过这个结构体联系这两层的，具体分析我们下面会慢慢分析，到时你领悟到编写内核的牛人们的思想。

##### MTD 字符设备（ mtdchar.c)

主要看下面几个东西：struct file\_operations mtd\_fops ，这个结构注册了 read， wirte， map， open等函数。写过字符设备人应该都知道。

static const struct file\_operations mtd\_fops = {

.owner = THIS\_MODULE,

.llseek = mtdchar\_lseek,

.read = mtdchar\_read,

.write = mtdchar\_write,

.unlocked\_ioctl = mtdchar\_unlocked\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = mtdchar\_compat\_ioctl,

#endif

.open = mtdchar\_open,

.release = mtdchar\_close,

.mmap = mtdchar\_mmap,

#ifndef CONFIG\_MMU

.get\_unmapped\_area = mtdchar\_get\_unmapped\_area,

#endif

};

init\_mtdchar 设备初始化函数，在这里完成设备节点的注册。

static int \_\_init init\_mtdchar(void)

{

int ret;

/\*

#define MTD\_CHAR\_MAJOR 90

#define MTD\_BLOCK\_MAJOR 31

\*/

/\*\_\_register\_chrdev 注册设备号\*/

ret = \_\_register\_chrdev(MTD\_CHAR\_MAJOR, 0, 1 << MINORBITS,

"mtd", &mtd\_fops);

if (ret < 0) {

pr\_notice("Can't allocate major number %d for "

"Memory Technology Devices.\n", MTD\_CHAR\_MAJOR);

return ret;

}

/\*注册文件系统\*/

ret = register\_filesystem(&mtd\_inodefs\_type);

if (ret) {

pr\_notice("Can't register mtd\_inodefs filesystem: %d\n", ret);

goto err\_unregister\_chdev;

}

return ret;

err\_unregister\_chdev:

\_\_unregister\_chrdev(MTD\_CHAR\_MAJOR, 0, 1 << MINORBITS, "mtd");

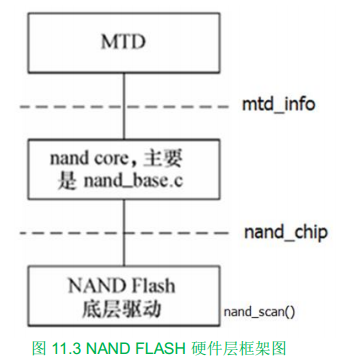
return ret;

}

MTD 块设备与此类似，只不过其设备类型不同而已，这里就不再分析了。

#### FLASH 硬件层

我们的 A8 板上只有 NAND FLASH，这一章的驱动也是在这一层上实现的。LINUX 的 nand flash 驱动都存放在 drivers\mtd\nand 里面，框架如下图 11.3 所示：



其中在 nand\_base.c 文件中，实现了最基本的 read()， write()， read\_oob()，write\_oob() 等等成员函数，这些就不再需要我们去实现了。

mtd\_info 在原始设备层已经介绍过了，其中最后有一个 void \*priv 私有数据指针，在这里 mtd\_info 的私有成员就指向给了 nand\_chip 结构体,在 FLASH 层中，我们也主要是对 nand\_chip 数据结构的一些比较特殊成员的初始化,然后通过 nand\_scan(mtd\_info,1)去设置 mtd\_info 里面的 nand\_chip 的其他成员，以及其他成员函数。

##### nand\_chip 结构体

MTD 使用 nand\_chip 数据结构表示一个 NAND Flash 芯片，这个结构体中包含了关于 NAND Flash 的地址信息、读写方法、 ECC 模式、硬件控制等一些底层机制。我们只有看了这个结构体才能知道什么需要设置。

struct nand\_chip {

void \_\_iomem \*IO\_ADDR\_R; /\* 读 8 位 I/O 线地址 \*/

void \_\_iomem \*IO\_ADDR\_W; /\* 写 8 位 I/O 线地址 \*/

/\* 从芯片中读一个字节 \*/

uint8\_t (\*read\_byte)(struct mtd\_info \*mtd);

/\* 从芯片中读一个字 \*/

u16 (\*read\_word)(struct mtd\_info \*mtd);

/\* 将缓冲区内容写入芯片 \*/

void (\*write\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len);

/\* 读芯片读取内容至缓冲区\*/

void (\*read\_buf)(struct mtd\_info \*mtd, uint8\_t \*buf, int len);

/\* 验证芯片和写入缓冲区中的数据 \*/

int (\*verify\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len);

/\* 选中芯片 \*/

void (\*select\_chip)(struct mtd\_info \*mtd, int chip);

/\* 检测是否有坏块 \*/

int (\*block\_bad)(struct mtd\_info \*mtd, loff\_t ofs, int getchip);

/\* 标记坏块 \*/

int (\*block\_markbad)(struct mtd\_info \*mtd, loff\_t ofs);

/\* 命令、地址、数据控制函数 \*/

void (\*cmd\_ctrl)(struct mtd\_info \*mtd, int dat,unsigned int ctrl);

/\* 设备是否就绪 \*/

int (\*dev\_ready)(struct mtd\_info \*mtd);

/\* 实现命令发送 \*/

void (\*cmdfunc)(struct mtd\_info \*mtd, unsigned command, int column,int page\_addr);

int (\*waitfunc)(struct mtd\_info \*mtd, struct nand\_chip \*this);

/\* 擦除命令的处理 \*/

void (\*erase\_cmd)(struct mtd\_info \*mtd, int page);

/\* 扫描坏块 \*/

int (\*scan\_bbt)(struct mtd\_info \*mtd);

int (\*errstat)(struct mtd\_info \*mtd, struct nand\_chip \*this, int state, intstatus, int page);

/\* 写一页 \*/

int (\*write\_page)(struct mtd\_info \*mtd, struct nand\_chip \*chip,const uint8\_t \*buf, int page, int cached, int raw);

int chip\_delay; /\* 由板决定的延迟时间 \*/

/\* 与具体的 NAND 芯片相关的一些选项，如 NAND\_NO\_AUTOINCR，

NAND\_BUSWIDTH\_16 等 \*/

unsigned int options;

/\* 用位表示的 NAND 芯片的 page 大小，如某片 NAND 芯片

\* 的一个 page 有 512 个字节，那么 page\_shift 就是 9

\*/

int page\_shift;

/\* 用位表示的 NAND 芯片的每次可擦除的大小，如某片 NAND 芯片每次可

\* 擦除 16K 字节(通常就是一个 block 的大小)，那么 phys\_erase\_shift 就是 14

\*/

int phys\_erase\_shift;

/\* 用位表示的 bad block table 的大小，通常一个 bbt 占用一个 block，

\* 所以 bbt\_erase\_shift 通常与 phys\_erase\_shift 相等

\*/

int bbt\_erase\_shift;

/\* 用位表示的 NAND 芯片的容量 \*/

int chip\_shift;

/\* NADN FLASH 芯片的数量 \*/

int numchips;

/\* NAND 芯片的大小 \*/

uint64\_t chipsize;

int pagemask;

int pagebuf;

int subpagesize;

uint8\_t cellinfo;

int badblockpos;

nand\_state\_t state;

uint8\_t \*oob\_poi;

struct nand\_hw\_control \*controller;

struct nand\_ecclayout \*ecclayout; /\* ECC 布局 \*/

/\* ECC 校验结构体，里面有大量的函数进行 ECC 校验 \*/

struct nand\_ecc\_ctrl ecc;

struct nand\_buffers \*buffers;

struct nand\_hw\_control hwcontrol;

struct mtd\_oob\_ops ops;

uint8\_t \*bbt;

struct nand\_bbt\_descr \*bbt\_td;

struct nand\_bbt\_descr \*bbt\_md;

struct nand\_bbt\_descr \*badblock\_pattern;

void \*priv;

};

##### 从 atmel\_nand.c 跟踪分析驱动

通过上一节层次的分析，还有 blabla 讲了一堆结构体，大家可能还是一头雾水，但是这都是基本的应该了解的知识，到了这一节，我们通过被内核收进 nand分支的驱动是怎么写的，并分析上一节没有分析的函数。

注意：以下代码因为版面限制，不重要的代码没有全部贴出来，严重建议学习驱动要建一个 soughtinsight 的工程，然后在里面看代码，怎么建工程在我们的《网蜂科技 A8 开发平台使用教程》有哦。

好了，现在我们开始跟随 atmel 公司写的 nand flash 驱动开始：分析 drivers\mtd\nand\atmel\_nand.c，从入口函数开始：

static int \_\_init atmel\_nand\_init(void)

{

return platform\_driver\_probe(&atmel\_nand\_driver, atmel\_nand\_probe);

}从这里知道这个驱动用的是 platform 的模型，当 match 到会跳转到 probe函数：

static int \_\_init atmel\_nand\_probe(struct platform\_device \*pdev)

{

struct mtd\_info \*mtd; //分配上一节介绍过的结构体

struct nand\_chip \*nand\_chip;

mtd->priv = nand\_chip;

/\*进行一些 nand\_chip 的设置\*/

/\* 第一阶段扫描，找到设备并获取页大小 \*/

/\* 传的是 mtd\_info 类型的参数，在这个函数会对 mtd\_info 进行一些设置 \*/

if (nand\_scan\_ident(mtd, 1, NULL))

{

res = -ENXIO;

goto err\_scan\_ident;

}

/\* 第二阶段扫描 \*/

if (nand\_scan\_tail(mtd)) {

res = -ENXIO;

goto err\_scan\_tail;

}

/\*MTD 设备的注册\*/

res = mtd\_device\_parse\_register(mtd, NULL, &ppdata,host->board.parts, host->board.num\_parts);

}

##### nand\_scan()

在上一节我们有提到这个函数，但是貌似这个 atmel 公司写的驱动好像没有用到它呀，不要急，我们进入这个函数看看在 drivers\mtd\nand\nand\_base.c

int nand\_scan(struct mtd\_info \*mtd, int maxchips)

{

int ret;

if (!mtd->owner && caller\_is\_module())

{

pr\_crit("%s called with NULL mtd->owner!\n", \_\_func\_\_);

BUG();

}

ret = nand\_scan\_ident(mtd, maxchips, NULL);

if (!ret)

ret = nand\_scan\_tail(mtd);

return ret;

}

感觉很短是不是，因为在这里他调用了下面这两个函数。

##### nand\_scan\_ident ()

int nand\_scan\_ident(struct mtd\_info \*mtd, int maxchips, struct nand\_flash\_dev \*table)

{

/\*从传参获取 nand\_chip 结构体内容\*/

struct nand\_chip \*chip = mtd->priv;

/\* Get buswidth to select the correct functions \*/

busw = chip->options & NAND\_BUSWIDTH\_16;

/\*设置默认的配置\*/

nand\_set\_defaults(chip, busw);

/\*获取 FLASH 的类型 \*/

type = nand\_get\_flash\_type(mtd, chip, busw,&nand\_maf\_id, &nand\_dev\_id, table);

/\*选中芯片\*/

chip->select\_chip(mtd, -1);

/\* Check for a chip array \*/

/\* … \*/

/\* Store the number of chips and calc total size for mtd \*/

/\* … \*/

}

这里用到一个面向对象的思想，不防跟踪进来看一下

static void nand\_set\_defaults(struct nand\_chip \*chip, int busw)

{

/\* 检查 chip->chip\_delay 在之前是否被设置过了，如果没有的话

\* 设为默认值 20，如果我们在之前自己的驱动去设置了，

\* 那么就不会被设置为这个默认的参数了 ,以下同理

\*/

if (!chip->chip\_delay)

chip->chip\_delay = 20;

/\* check, if a user supplied command function given \*/

if (chip->cmdfunc == NULL)

chip->cmdfunc = nand\_command;

/\* check, if a user supplied wait function given \*/

if (chip->waitfunc == NULL)

chip->waitfunc = nand\_wait;

if (!chip->select\_chip)

chip->select\_chip = nand\_select\_chip;

if (!chip->read\_byte)

chip->read\_byte = busw ? nand\_read\_byte16 : nand\_read\_byte;

if (!chip->read\_word)

chip->read\_word = nand\_read\_word;

if (!chip->block\_bad)

chip->block\_bad = nand\_block\_bad;

if (!chip->block\_markbad)

chip->block\_markbad = nand\_default\_block\_markbad;

if (!chip->write\_buf)

chip->write\_buf = busw ? nand\_write\_buf16 : nand\_write\_buf;

if (!chip->read\_buf)

chip->read\_buf = busw ? nand\_read\_buf16 : nand\_read\_buf;

if (!chip->scan\_bbt)

chip->scan\_bbt = nand\_default\_bbt;

if (!chip->controller) {

chip->controller = &chip->hwcontrol;

spin\_lock\_init(&chip->controller->lock);

init\_waitqueue\_head(&chip->controller->wq);

}

}

至此我们知道 nand\_scan 函数是以 mtd\_info 为参数探测 NAND flash 的存在，存在的话选中芯片，还有设置 mtd\_info -> nand\_chip 里面的参数。设置过程是如果没配置过的会用默认的配置。默认配置在 nand\_base.c 里面，是一些比较通用的函数，而一些配置要在自己的驱动程序来配置。这里典型的利用了面向对象的继承和重载的思想，不得不佩服写 linux 的这些大牛们。

##### nand\_scan\_tail()

int nand\_scan\_tail(struct mtd\_info \*mtd)

{

struct nand\_chip \*chip = mtd->priv;

/\*和前面一样，如果之前没配置用默认值 \*/

if (!chip->ecc.layout && (chip->ecc.mode != NAND\_ECC\_SOFT\_BCH)) {

switch (mtd->oobsize) {

case 8:

chip->ecc.layout = &nand\_oob\_8;

break;

case 16:

chip->ecc.layout = &nand\_oob\_16;

break;

case 64:

chip->ecc.layout = &nand\_oob\_64;

break;

case 128:

chip->ecc.layout = &nand\_oob\_128;

break;

default:

pr\_warn("No oob scheme defined for oobsize %d\n",mtd->oobsize);

BUG();

}

}

if (!chip->write\_page)

chip->write\_page = nand\_write\_page;

/\* … \*/

/\* 填充 mtd\_info 结构剩下的成员数据\*/

mtd->type = MTD\_NANDFLASH;

mtd->flags = (chip->options & NAND\_ROM) ? MTD\_CAP\_ROM :

MTD\_CAP\_NANDFLASH;

mtd->\_erase = nand\_erase;

mtd->\_point = NULL;

mtd->\_unpoint = NULL;

mtd->\_read = nand\_read;

mtd->\_write = nand\_write;

mtd->\_panic\_write = panic\_nand\_write;

mtd->\_read\_oob = nand\_read\_oob;

mtd->\_write\_oob = nand\_write\_oob;

mtd->\_sync = nand\_sync;

mtd->\_lock = NULL;

mtd->\_unlock = NULL;

mtd->\_suspend = nand\_suspend;

mtd->\_resume = nand\_resume;

mtd->\_block\_isbad = nand\_block\_isbad;

mtd->\_block\_markbad = nand\_block\_markbad;

mtd->writebufsize = mtd->writesize;

/\* propagate ecc info to mtd\_info \*/

/\*…\*/

}

到这里 mtd\_info 的配置都设置好了，因为我们用了 nand\_scan()去设置，所以调用完 mtd\_info 结构体里面的参数，想 read， write 函数就能支持 nand flash了。

##### 调用过程

我们来看一下他的调用过程：

App call read() ->

mtdchar\_read() [struct file\_operations mtd\_fops of mtdchar.c]

->mtd\_read(mtd, \*ppos, len, &retlen, kbuf); [mtdchar\_read（） of mtdchar.c]

->mtd->\_read = nand\_read;

->nand\_do\_read\_ops(mtd, from, &ops)

-> chip->ecc.read\_page(mtd, chip, bufpoi,oob\_required, page);

chip->ecc.read\_page = nand\_read\_page\_hwecc()->chip->read\_buf()

通过这些配置，一层层下来就能对 flash 操作了。

#### add\_mtd\_partitions()

接下来我们继续看这个函数 mtd\_device\_parse\_register()

int mtd\_device\_parse\_register(struct mtd\_info \*mtd, const char \*\*types, struct mtd\_part\_parser\_data \*parser\_data, const struct mtd\_partition \*parts, int nr\_parts)

{

int err;

struct mtd\_partition \*real\_parts;

err = parse\_mtd\_partitions(mtd, types, &real\_parts, parser\_data);

if (err <= 0 && nr\_parts && parts) {

real\_parts = kmemdup(parts, sizeof(\*parts) \* nr\_parts,GFP\_KERNEL);

if (!real\_parts)

err = -ENOMEM;

else

err = nr\_parts;

}

if (err > 0) {

err = add\_mtd\_partitions(mtd, real\_parts, err); /\*调用了这个函数\*/

kfree(real\_parts);

} else if (err == 0) {

err = add\_mtd\_device(mtd);

if (err == 1)

err = -ENODEV;

}

return err;

}

接 下 来 我 们 就 来 仔 细 分 析 这 个 函 数 add\_mtd\_partitions() ， 他 处 于drivers/mtd/mtdpart.c。

add\_mtd\_partitions()看函数名就知道他大概是添加分区的作用。其实他会新建一个 mtd\_part 结构体，将其加入 mtd\_partions 中，并调用 add\_mtd\_device()将此分区作为 MTD 设备加入 radix 树，便于查找。（我们使用的内核是 linux 3.8已经不再使用 mtd\_table 这个变量了） .现在我们明白了 mtd\_part 与 partitions的关系了吧！当然细心同学又留意到了一个新的名词 radix 树,这个是什么呢？等一下看到后面我们就知道。我们来分析一些这个 add\_mtd\_partitions（）都做了些什么。

/ \* \*

此函数将创建 nbparts 个 mtd\_part，其中每个 mtd\_part 包含了一个\*mtd\_info，mtd\_info 的数据内容来自 master 和 parts。程序将 mtd\_info\*加入 radix 树，并将 mtd\_part 中的 list\_head 加入 list

\* /

int add\_mtd\_partitions(struct mtd\_info \*master,const struct mtd\_partition\*parts,int nbparts)

{

struct mtd\_part \*slave; // mtd\_part 结构体指针

uint64\_t cur\_offset = 0; //当前的便移量

int i;

printk(KERN\_NOTICE "Creating %d MTD partitions on \"%s\":\n", nbparts,master->name);

/\*这个 for 循环将分配 nbparts 个 mtd\_part\*/

for (i = 0; i < nbparts; i++) {

/\*分配 mtd\_part 内存\*/

slave = allocate\_partition(master, parts + i, i, cur\_offset);

if (IS\_ERR(slave))

return PTR\_ERR(slave);

/\*加锁\*/

mutex\_lock(&mtd\_partitions\_mutex);

/\*将 mtd\_partitions 加入链表\*/

list\_add(&slave->list, &mtd\_partitions);\

/\*解锁\*/

mutex\_unlock(&mtd\_partitions\_mutex);

/\*加入 mtd\_table， &slave->mtd 是 mtd\_info 结构体\*/

add\_mtd\_device(&slave->mtd);

cur\_offset = slave->offset + slave->mtd.size;

}

return 0;

}

为了解开 radix 树是什么的谜，我们看一下 add\_mtd\_device(&slave->mtd);是如何将 mtd\_info 加入 radix 树的。在 driver/mtd/mtdcore.c 可以找到他

#### add\_mtd\_device()

int add\_mtd\_device(struct mtd\_info \*mtd)

{

struct mtd\_notifier \*not; //mtd\_notifier MTD 设备通知结构体

int i, error;

/\*根据 mtd\_info->type 判度设备类型,对 backing\_dev\_info 赋值\*/

/\*backing\_dev\_info 是 linux 3.x writeback 机制一个重要结构体,该数据结构

描述了 backing\_dev 的所有信息，通常块设备的 request queue 中会包含

backing\_dev 对象.\*/

if (!mtd->backing\_dev\_info) {

switch (mtd->type) {

case MTD\_RAM:

mtd->backing\_dev\_info = &mtd\_bdi\_rw\_mappable;

break;

case MTD\_ROM:

mtd->backing\_dev\_info = &mtd\_bdi\_ro\_mappable;

break;

default:

mtd->backing\_dev\_info = &mtd\_bdi\_unmappable;

break;

}

}

BUG\_ON(mtd->writesize == 0);

/\*上锁\*/

mutex\_lock(&mtd\_table\_mutex);

/\* idr\_get\_new, idr 机制的一个函数，分配 ID 号并将 ID 号和指针关联\*/

do {

if (!idr\_pre\_get(&mtd\_idr, GFP\_KERNEL))

goto fail\_locked;

error = idr\_get\_new(&mtd\_idr, mtd, &i);

} while (error == -EAGAIN);

if (error)

goto fail\_locked;

mtd->index = i;

mtd->usecount = 0;

/\* 这个 bitflip\_threshold 可以参考

\*Documentation/ABI/testing/sysfs-class-mtd,他的值与

\*mtd->ecc\_strength 有关

\* /

/\* default value if not set by driver \*/

if (mtd->bitflip\_threshold == 0)

mtd->bitflip\_threshold = mtd->ecc\_strength;

if (is\_power\_of\_2(mtd->erasesize))

mtd->erasesize\_shift = ffs(mtd->erasesize) - 1;

else

mtd->erasesize\_shift = 0;

if (is\_power\_of\_2(mtd->writesize))

mtd->writesize\_shift = ffs(mtd->writesize) - 1;

else

mtd->writesize\_shift = 0;

mtd->erasesize\_mask = (1 << mtd->erasesize\_shift) - 1;

mtd->writesize\_mask = (1 << mtd->writesize\_shift) - 1;

/\* Some chips always power up locked. Unlock them now \*/

/\*解锁 MTD 设备\*/

if ((mtd->flags & MTD\_WRITEABLE) && (mtd->flags & MTD\_POWERUP\_LOCK)) {

error = mtd\_unlock(mtd, 0, mtd->size);

if (error && error != -EOPNOTSUPP)

printk(KERN\_WARNING "%s: unlock failed, writes may not work\n",mtd->name);

}

/\* Caller should have set dev.parent to match the

\* physical device.

\*/

mtd->dev.type = &mtd\_devtype;

mtd->dev.class = &mtd\_class;

mtd->dev.devt = MTD\_DEVT(i);

dev\_set\_name(&mtd->dev, "mtd%d", i);

dev\_set\_drvdata(&mtd->dev, mtd);

/\*假如 mtd->dev 不是 null，则注册设备.这个跟设备节点注册有关\*/

if (device\_register(&mtd->dev) != 0)

goto fail\_added;

if (MTD\_DEVT(i))

device\_create(&mtd\_class, mtd->dev.parent,MTD\_DEVT(i) + 1,NULL, "mtd%dro", i);

pr\_debug("mtd: Giving out device %d to %s\n", i, mtd->name);

/\*遍历 list 链表将每个 mtd\_notifier 执行 add()函数，对新加入的 mtd 设备操作，通知所有的 MTD user 新的 MTD 设备的到来 \*/

list\_for\_each\_entry(not, &mtd\_notifiers, list)

not->add(mtd);

mutex\_unlock(&mtd\_table\_mutex);

/\* We \_know\_ we aren't being removed, because our caller is still holding us here. So none of this try\_ nonsense, and no bitching about it

either. :) \*/

\_\_module\_get(THIS\_MODULE);

return 0;

fail\_added:

idr\_remove(&mtd\_idr, i);

fail\_locked:

mutex\_unlock(&mtd\_table\_mutex);

return 1;

}

这 些 添 加 通过 遍 历 这个 队 列 ，然 后 添 加进 来 。 搜索 mtd\_notifier 在drivers\mtd\mtd\_blkdevs.c，通过这个结构体到了 MTD 设备层了，看他的.add成员。

static struct mtd\_notifier blktrans\_notifier = {

.add = blktrans\_notify\_add,

.remove = blktrans\_notify\_remove,

};

追踪有

static void blktrans\_notify\_add(struct mtd\_info \*mtd)

{

struct mtd\_blktrans\_ops \*tr;

if (mtd->type == MTD\_ABSENT)

return;

list\_for\_each\_entry(tr, &blktrans\_majors, list)

tr->add\_mtd(tr, mtd);

}

搜索 tr 的原型 mtd\_blktrans\_ops 有

static struct mtd\_blktrans\_ops mtdblock\_tr = {

.name = "mtdblock",

.major = 31,

.part\_bits = 0,

.blksize = 512,

.open = mtdblock\_open,

.flush = mtdblock\_flush,

.release = mtdblock\_release,

.readsect = mtdblock\_readsect,

.writesect = mtdblock\_writesect,

.add\_mtd = mtdblock\_add\_mtd,

.remove\_dev = mtdblock\_remove\_dev,

.owner = THIS\_MODULE,

};

追踪有

static void mtdblock\_add\_mtd(struct mtd\_blktrans\_ops \*tr,struct mtd\_info \*mtd)

{

struct mtd\_blktrans\_dev \*dev = kzalloc(sizeof(\*dev), GFP\_KERNEL);

if (add\_mtd\_blktrans\_dev(dev))

kfree(dev);

}

追踪有

int add\_mtd\_blktrans\_dev(struct mtd\_blktrans\_dev \*new)

{

struct gendisk \*gd;

gd = alloc\_disk(1 << tr->part\_bits);

new->rq = blk\_init\_queue(mtd\_blktrans\_request, &new->queue\_lock);

gd->queue = new->rq;

add\_disk(gd);

}