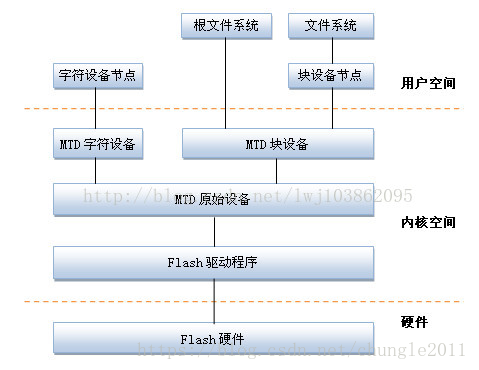
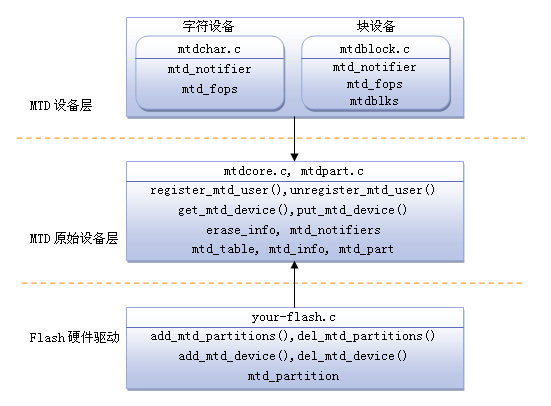
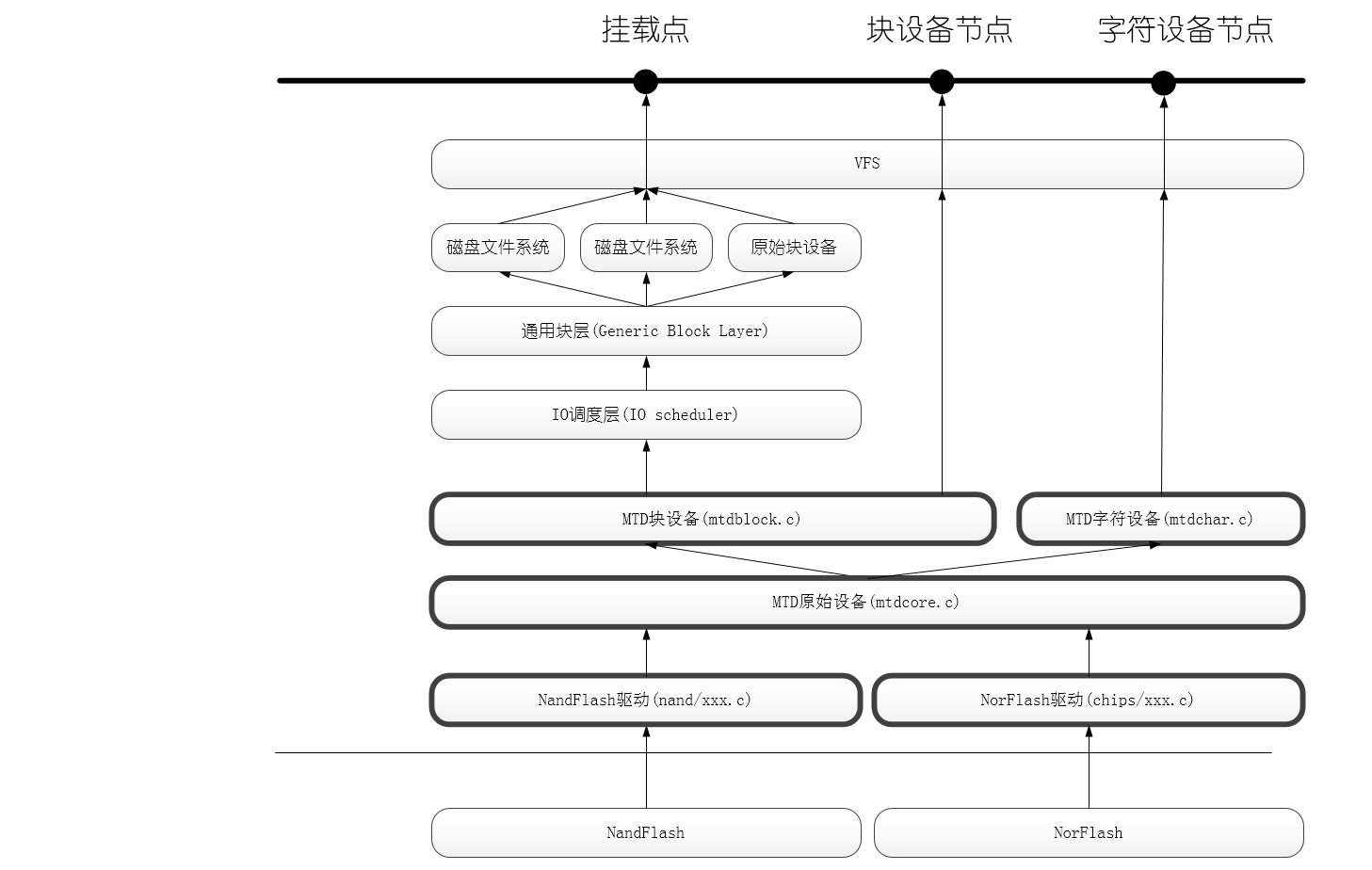
# Linux MTD系统

MTD，Memory Technology Device即内存技术设备，在Linux内核中，引入MTD层为NOR FLASH和NAND FLASH设备提供统一接口。MTD将文件系统与底层FLASH存储器进行了隔离。







MTD设备通常可分为四层，从上到下依次是：设备节点、MTD设备层、MTD原始设备层、硬件驱动层。

Flash硬件驱动层：Flash硬件驱动层负责对Flash硬件的读、写和擦除操作。MTD设备的Nand Flash芯片的驱动则drivers/mtd/nand/子目录下,Nor Flash芯片驱动位于drivers/mtd/chips/子目录下。

MTD原始设备层：用于描述MTD原始设备的数据结构是mtd\_info，它定义了大量的关于MTD的数据和操作函数。其中mtdcore.c:  MTD原始设备接口相关实现，mtdpart.c :  MTD分区接口相关实现。由MTD原始设备的通用代码+特定的Flash数据组成。mtd\_info、mtd\_part、mtd\_partition以及mtd\_partitions等对象及其操作方法就属于这一层，对应的文件是"drivers/mtd/mtdcore.c"。类似于i2c驱动框架中的核心层。

MTD设备层：基于MTD原始设备，linux系统可以定义出MTD的块设备（主设备号31）和字符设备（设备号90）。其中drivers/mtd/mtdchar.c :  MTD字符设备接口相关实现，drivers/mtd/mtdblock.c : MTD块设备接口相关实现。块设备模拟：MTD提供一个称谓mtdblock的块驱动程序，它在闪存上模拟一块硬盘，你可以将任何文件系统(如：ext2)放在模拟的闪存磁盘上，mtdblock隐藏了复杂的闪存访问过程(比如写之前先删除相关扇区的内容)，被mtdblock创建的设备节点命名为/dev/mtdblock/X，其中X是分区号。字符设备模拟：mtdchar是底层闪存设备呈现出线性特点，与文件系统的块设备特性不同，mtdchar建立的设备节点命名为/dev/mtd/X，其中X为分区号，例如，写入引导程序: dd if=bootloader.bin of=/dev/mtd/0 ；一个原始mtdchar分区的使用示例是POST错误日志，另外一个嵌入式系统使用字符闪存分区的例子是保存类似于PC的CMOS、EEPROM信息。

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

### 核心结构

* mtd\_info描述原始设备层的一个分区的结构, 描述一个设备或一个多分区设备中的一个分区
* ~~mtd\_table管理原始设备层的mtd\_info的数组~~
* mtd\_part表示一个分区, 其中的struct mtd\_info mtd描述该分区的信息， 一个物理Flash设备可以有多于1个mtd\_part，每个mtd\_part都对应一个mtd\_info。
* mtd\_partition描述一个分区表, 通过管理mtd\_part以及每一个mtd\_part中的mtd\_info来描述所有的分区，一个物理Flash设备只有一个mtd\_partition
* mtd\_partitions是一个list\_head对象，用于管理mtd\_partition们
* map\_info描述一个NOR Flash设备
* nand\_chip描述一个NAND Flash设备

### 核心方法

* add\_mtd\_device()/del\_mtd\_device()注册/注销一个MTD设备
* add\_mtd\_partitions()/del\_mtd\_partitions()注册注销一个或多个分区表,
* do\_map\_probe()用来根据传入的参数匹配一个map\_info对象的驱动，比如CFI接口或JEDEC接口的NOR Flash，并返回一个mtd\_info以便注册分区信息。
* nand\_scan()：NAND flash使用这个API来匹配驱动。

MTD数据结构：

1.Linux内核使用mtd\_info结构体表示MTD原始设备，这其中定义了大量关于MTD的数据和操作函数（后面将会看到），所有的mtd\_info结构体存放在mtd\_table[更新的kernel里面被改为了mtd\_idr]结构体数据里。在/drivers/mtd/mtdcore.c里：

1. struct mtd\_info \*mtd\_table[MAX\_MTD\_DEVICES];   [old]

2. static DEFINE\_IDR(mtd\_idr); [new]

2.Linux内核使用mtd\_part结构体表示分区，其中mtd\_info结构体成员用于描述该分区，大部分成员由其主分区mtd\_part->master决定，各种函数也指向主分区的相应函数。每一个mtd\_part都会调用add\_mtd\_device一次，每次增加一个mtd\_info结构，如果没有使用mtd\_part，那么整块nand就只有一个mtd\_info

1. struct mtd\_part {
2. struct mtd\_info mtd;        /\* 分区信息, 大部分由master决定 \*/
3. struct mtd\_info \*master;    /\* 分区的主分区 \*/
4. uint64\_t offset;            /\* 分区的偏移地址 \*/
5. int index;                  /\* 分区号 (Linux3.0后不存在该字段) \*/
6. struct list\_head list;      /\* 将mtd\_part链成一个链表mtd\_partitons \*/
7. int registered;
8. };

# 核心结构与方法详述

## mtd\_info

本身是没有list\_head来供内核管理，对mtd\_info对象的管理是通过mtd\_part来实现的。mtd\_info对象属于原始设备层，里面的很多函数接口内核已经实现了。mtd\_info中的read()/write()等操作是MTD设备驱动要实现的主要函数，在NORFlash或NANDFlash中的驱动代码中几乎看不到mtd\_info的成员函数，即这些函数对于Flash芯片是透明的，因为Linux在MTD的下层实现了针对NORFlash和NANDFlash的通用的mtd\_info函数。

114 struct mtd\_info {

115 u\_char type;

116 uint32\_t flags;

117 uint64\_t size; // Total size of the MTD

118

123 uint32\_t erasesize;

131 uint32\_t writesize;

132

142 uint32\_t writebufsize;

143

144 uint32\_t oobsize; // Amount of OOB data per block (e.g. 16)

145 uint32\_t oobavail; // Available OOB bytes per block

146

151 unsigned int erasesize\_shift;

152 unsigned int writesize\_shift;

153 /\* Masks based on erasesize\_shift and writesize\_shift \*/

154 unsigned int erasesize\_mask;

155 unsigned int writesize\_mask;

156

164 unsigned int bitflip\_threshold;

165

166 // Kernel-only stuff starts here.

167 const char \*name;

168 int index;

169

170 /\* ECC layout structure pointer - read only! \*/

171 struct nand\_ecclayout \*ecclayout;

172

173 /\* the ecc step size. \*/

174 unsigned int ecc\_step\_size;

175

176 /\* max number of correctible bit errors per ecc step \*/

177 unsigned int ecc\_strength;

178

179 /\* Data for variable erase regions. If numeraseregions is zero,

180 \* it means that the whole device has erasesize as given above.

181 \*/

182 int numeraseregions;

183 struct mtd\_erase\_region\_info \*eraseregions;

184

185 /\*

186 \* Do not call via these pointers, use corresponding mtd\_\*()

187 \* wrappers instead.

188 \*/

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

190 int (\*\_point) (struct mtd\_info \*mtd, loff\_t from, size\_t len,

191 size\_t \*retlen, void \*\*virt, resource\_size\_t \*phys);

192 int (\*\_unpoint) (struct mtd\_info \*mtd, loff\_t from, size\_t len);

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

194 unsigned long len,

195 unsigned long offset,

196 unsigned long flags);

197 int (\*\_read) (struct mtd\_info \*mtd, loff\_t from, size\_t len,

198 size\_t \*retlen, u\_char \*buf);

199 int (\*\_write) (struct mtd\_info \*mtd, loff\_t to, size\_t len,

200 size\_t \*retlen, const u\_char \*buf);

201 int (\*\_panic\_write) (struct mtd\_info \*mtd, loff\_t to, size\_t len,

202 size\_t \*retlen, const u\_char \*buf);

203 int (\*\_read\_oob) (struct mtd\_info \*mtd, loff\_t from,

204 struct mtd\_oob\_ops \*ops);

205 int (\*\_write\_oob) (struct mtd\_info \*mtd, loff\_t to,

206 struct mtd\_oob\_ops \*ops);

207 int (\*\_get\_fact\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf,

208 size\_t len);

209 int (\*\_read\_fact\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from,

210 size\_t len, size\_t \*retlen, u\_char \*buf);

211 int (\*\_get\_user\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf,

212 size\_t len);

213 int (\*\_read\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from,

214 size\_t len, size\_t \*retlen, u\_char \*buf);

215 int (\*\_write\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t to,

216 size\_t len, size\_t \*retlen, u\_char \*buf);

217 int (\*\_lock\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from,

218 size\_t len);

219 int (\*\_writev) (struct mtd\_info \*mtd, const struct kvec \*vecs,

220 unsigned long count, loff\_t to, size\_t \*retlen);

221 void (\*\_sync) (struct mtd\_info \*mtd);

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

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

224 int (\*\_is\_locked) (struct mtd\_info \*mtd, loff\_t ofs, uint64\_t len);

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

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

227 int (\*\_suspend) (struct mtd\_info \*mtd);

228 void (\*\_resume) (struct mtd\_info \*mtd);

229 /\*

230 \* If the driver is something smart, like UBI, it may need to maintain

231 \* its own reference counting. The below functions are only for driver.

232 \*/

233 int (\*\_get\_device) (struct mtd\_info \*mtd);

234 void (\*\_put\_device) (struct mtd\_info \*mtd);

235

236 /\* Backing device capabilities for this device

237 \* - provides mmap capabilities

238 \*/

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

240

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

242

243 /\* ECC status information \*/

244 struct mtd\_ecc\_stats ecc\_stats;

245 /\* Subpage shift (NAND) \*/

246 int subpage\_sft;

247

248 void \*priv;

249

250 struct module \*owner;

251 struct device dev;

252 int usecount;

253 };

struct mtd\_info  
--115-→MTD设备类型，有MTD\_RAM，MTD\_ROM、MTD\_NORFLASH、MTD\_NAND\_FLASH  
--116-->读写及权限标志位，有MTD\_WRITEABLE、MTD\_BIT\_WRITEABLE、MTD\_NO\_ERASE、MTD\_UP\_LOCK  
--117-->MTD设备的大小  
--123-->主要的擦除块大小，NandFlash就是"块"的大小  
--131-->最小可写字节数，NandFlash一般对应"页"的大小  
--144-->一个block中的OOB字节数  
--145-->一个block中可用oob的字节数  
--171-->ECC布局结构体指针  
--190-->针对eXecute-In-Place，即XIP  
--192-->如果这个指针为空，不允许XIP  
--197-->读函数指针  
--199-->写函数指针  
--248-->私有数据

## mtd\_part

内核管理分区的链表节点，通过它来实现对mtd\_info对象的管理。

41 struct mtd\_part {

42 struct mtd\_info mtd;

43 struct mtd\_info \*master;

44 uint64\_t offset;

45 struct list\_head list;

46 };

struct mtd\_part  
--42-->对应的mtd\_info对象  
--43-->父对象指针  
--44-->偏移量  
--45-->链表节点

## mtd\_partition

描述一个分区

39 struct mtd\_partition {

40 const char \*name; /\* identifier string \*/

41 uint64\_t size; /\* partition size \*/

42 uint64\_t offset; /\* offset within the master MTD space \*/

43 uint32\_t mask\_flags; /\* master MTD flags to mask out for this partition \*/

44 struct nand\_ecclayout \*ecclayout; /\* out of band layout for this partition (NAND only) \*/

45 };

mtd\_partition  
--40-->分区名  
--41-->分区大小，使用MTDPART\_SIZ\_FULL表示使用全部空间  
--42-->分区在master设备中的偏移量。MTDPART\_OFS\_APPEND表示从上一个分区结束的地方开始，MTDPART\_OFS\_NXTBLK表示从下一个擦除块开始; MTDPART\_OFS\_RETAIN表示尽可能向后偏，把size大小的空间留下即可  
--43-->权限掩码，MTD\_WRITEABLE表示将父设备的只读选项变成可写(可写分区要求size和offset要erasesize对齐，eg MTDPART\_OFS\_NEXTBLK)  
--44-->NANDFlash的OOB布局，OOB是NANDFlash中很有用空间，比如yaffs2就需要将坏块信息存储在OOB区域

## mtd\_partitions

链表头，将所有的mtd\_partition连接起来。

36 /\* Our partition linked list \*/

37 static LIST\_HEAD(mtd\_partitions);

下图是关键API的调用关系。

mtd\_add\_partition()  
   └── add\_mtd\_device()  
add\_mtd\_partitions()  
   └── add\_mtd\_device()

## add\_mtd\_device()

分配并初始化一个mtd对象。

367 334 int add\_mtd\_device(struct mtd\_info \*mtd)

335 {

336 struct mtd\_notifier \*not;

337 int i, error;

338

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

340 switch (mtd->type) {

341 case MTD\_RAM:

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

343 break;

344 case MTD\_ROM:

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

346 break;

347 default:

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

349 break;

350 }

351 }

355

356 i = idr\_alloc(&mtd\_idr, mtd, 0, 0, GFP\_KERNEL);

357 if (i < 0)

358 goto fail\_locked;

359

360 mtd->index = i;

361 mtd->usecount = 0;

362

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

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

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

366

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

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

369 else

370 mtd->erasesize\_shift = 0;

371

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

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

374 else

375 mtd->writesize\_shift = 0;

376

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

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

379

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

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

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

387 }

388

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

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

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

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

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

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

399

400 if (MTD\_DEVT(i))

401 device\_create(&mtd\_class, mtd->dev.parent,

402 MTD\_DEVT(i) + 1,

403 NULL, "mtd%dro", i);

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

409 not->add(mtd);

417 return 0;

424 }

add\_mtd\_device()  
--395-->设置MTD设备的名字  
--396-->设置私有数据，将mtd地址藏到device->device\_private->void\* driver\_data  
--408-->遍历所有的mtd\_notifier，将其添加到通知链

## mtd\_add\_partition()

通过将一个mtd\_part对象注册到内核，将mtd\_info对象注册到内核，即为一个设备添加一个分区。

537 int mtd\_add\_partition(struct mtd\_info \*master, const char \*name,

538 long long offset, long long length)

539 {

540 struct mtd\_partition part;

541 struct mtd\_part \*p, \*new;

542 uint64\_t start, end;

543 int ret = 0;

545 /\* the direct offset is expected \*/

546 if (offset == MTDPART\_OFS\_APPEND ||

547 offset == MTDPART\_OFS\_NXTBLK)

548 return -EINVAL;

549

550 if (length == MTDPART\_SIZ\_FULL)

551 length = master->size - offset;

552

553 if (length <= 0)

554 return -EINVAL;

555

556 part.name = name;

557 part.size = length;

558 part.offset = offset;

559 part.mask\_flags = 0;

560 part.ecclayout = NULL;

561

562 new = allocate\_partition(master, &part, -1, offset);

563 if (IS\_ERR(new))

564 return PTR\_ERR(new);

565

566 start = offset;

567 end = offset + length;

568

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

570 list\_for\_each\_entry(p, &mtd\_partitions, list)

571 if (p->master == master) {

572 if ((start >= p->offset) &&

573 (start < (p->offset + p->mtd.size)))

574 goto err\_inv;

575

576 if ((end >= p->offset) &&

577 (end < (p->offset + p->mtd.size)))

578 goto err\_inv;

579 }

580

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

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

583

584 add\_mtd\_device(&new->mtd);

585

586 return ret;

591 }

## add\_mtd\_partitions()

添加一个分区表到内核，一个MTD设备一个分区表

626 int add\_mtd\_partitions(struct mtd\_info \*master,

627 const struct mtd\_partition \*parts,

628 int nbparts)

629 {

630 struct mtd\_part \*slave;

631 uint64\_t cur\_offset = 0;

632 int i;

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

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

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

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

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

648 }

649

650 return 0;

651 }

# 用户空间编程

MTD设备提供了字符设备和块设备两种接口，对于字符设备接口，在"drivers/mtd/mtdchar.c"中实现了，比如，用户程序可以直接通过ioctl()回调相应的驱动实现。其中下面的几个是这些操作中常用的结构，这些结构是对用户空间开放的，类似于输入子系统中的input\_event结构。

### mtd\_info\_user

//include/uapi/mtd/mtd-abi.h

125 struct mtd\_info\_user {

126 \_\_u8 type;

127 \_\_u32 flags;

128 \_\_u32 size; /\* Total size of the MTD \*/

129 \_\_u32 erasesize;

130 \_\_u32 writesize;

131 \_\_u32 oobsize; /\* Amount of OOB data per block (e.g. 16) \*/

132 \_\_u64 padding; /\* Old obsolete field; do not use \*/

133 };

### mtd\_oob\_buf

描述NandFlash的OOB（Out Of Band）信息。

35 struct mtd\_oob\_buf {

36 \_\_u32 start;

37 \_\_u32 length;

38 unsigned char \_\_user \*ptr;

39 };

### erase\_info\_user

25 struct erase\_info\_user {

26 \_\_u32 start;

27 \_\_u32 length;

28 };

### 实例

mtd\_oob\_buf oob;

erase\_info\_user erase;

mtd\_info\_user meminfo;

/\* 获得设备信息 \*/

if(0 != ioctl(fd, MEMGETINFO, &meminfo))

perror("MEMGETINFO");

/\* 擦除块 \*/

if(0 != ioctl(fd, MEMERASE, &erase))

perror("MEMERASE");

/\* 读OOB \*/

if(0 != ioctl(fd, MEMREADOOB, &oob))

perror("MEMREADOOB");

/\* 写OOB??? \*/

if(0 != ioctl(fd, MEMWRITEOOB, &oob))

perror("MEMWRITEOOB");

/\* 检查坏块 \*/

if(blockstart != (ofs & (~meminfo.erase + 1))){

blockstart = ofs & (~meminfo.erasesize + 1);

if((badblock = ioctl(fd, MEMGETBADBLOCK, &blockstart)) < 0)

perror("MEMGETBADBLOCK");

else if(badblock)

/\* 坏块代码 \*/

else

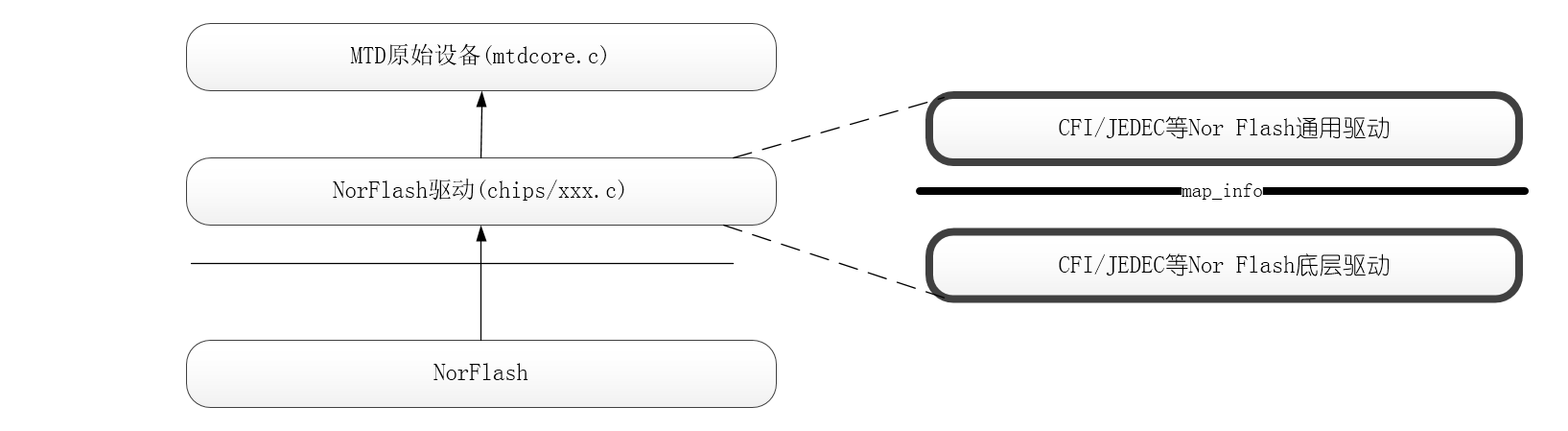
/\* 好块代码 \*/

}

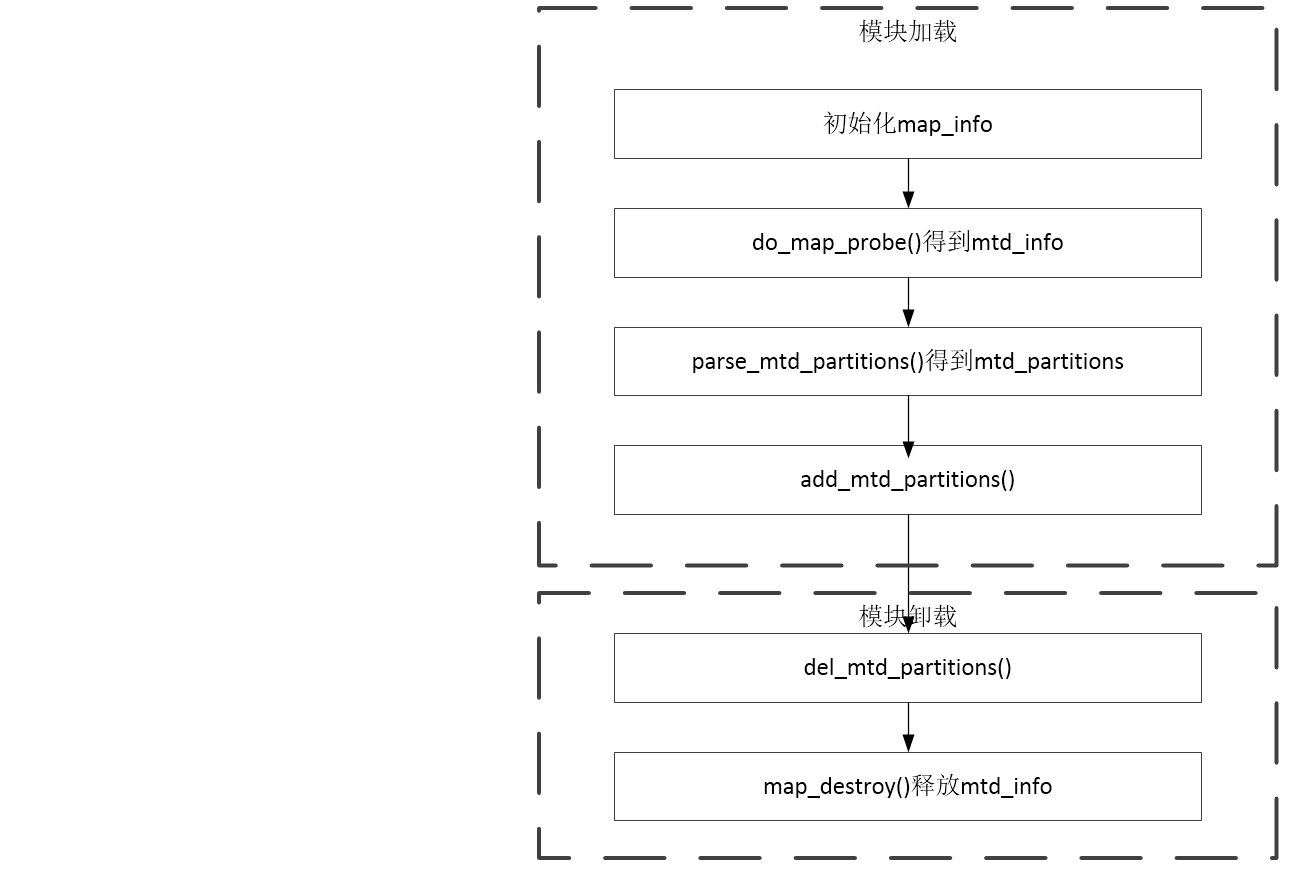
NANDFlash和NORFlash都是基于MTD框架编写的，由于MTD框架中通用代码已经在内核中实现了，所以驱动开发主要是进行MTD框架中的的开发。

# NOR Flash驱动

下图就是NORFlash驱动在MTD驱动框架中的位置



基于上述的MTD框架, Flash驱动都变的十分的简单, 因为当下Flash的操作接口已经很统一, a, 相应的代码在"drivers/mtd/chips"中文件实现，所以在设备驱动层, 留给驱动工程师的工作就大大的减少了。  
基于MTD子系统开发NOR FLash驱动，只需要构造一个map\_info类型的对象并调用do\_map\_probe()来匹配内核中已经写好的驱动，比如CFI接口的驱动或JEDEC接口的驱动。当下编写一个NorFlash驱动的工作流程如下



### map\_info

208 struct map\_info {

209 const char \*name;

210 unsigned long size;

211 resource\_size\_t phys;

212 #define NO\_XIP (-1UL)

214 void \_\_iomem \*virt;

215 void \*cached;

217 int swap; /\* this mapping's byte-swapping requirement \*/

218 int bankwidth;

243 void (\*set\_vpp)(struct map\_info \*, int);

245 unsigned long pfow\_base;

246 unsigned long map\_priv\_1;

247 unsigned long map\_priv\_2;

248 struct device\_node \*device\_node;

249 void \*fldrv\_priv;

250 struct mtd\_chip\_driver \*fldrv;

251 };

struct map\_info  
--210-->NOR Flash设备的容量  
--211-->NOR Flash在物理地址空间中的地址  
--214-->由物理地址映射的虚拟地址  
--218-->总线宽度，NOR Flash是有地址总线的，所以才能片上执行，一般都是8位或16位宽

构造好一个map\_info对象之后，接下来的工作就是匹配驱动+注册分区表

### do\_map\_probe()

这个API用来根据传入的参数匹配一个map\_info对象的驱动，比如CFI接口或JEDEC接口的NOR Flash。这个函数的接口如下：

struct mtd\_info \*do\_map\_probe(const char \*name, struct map\_info \*map)

对于常用的NorFlash标准，这个函数的调用方式如下：

do\_map\_probe("cfi\_probe", &xxx\_map\_info);

do\_map\_probe("jedec\_probe",&xxx\_map\_info);

do\_map\_probe("map\_rom",&xxx\_map\_info);

匹配了设备驱动，可以发现一个map\_info对象中没有mtd\_partitions相关的信息，对于一个NOR Flash的分区信息，需要通过do\_map\_probe返回的mtd\_info对象来注册到内核。这里我们可以先调用parse\_mtd\_partitions()查看Flash上已有的分区信息，获取了分区信息之后再调用add\_mtd\_partitions()将分区信息写入内核

### NOR Flash驱动模板

#define WINDOW\_SIZE ...

#define WINDOW\_ADDR ...

static struct map\_info xxx\_map = {

.name = "xxx flash",

.size = WINDOW\_SIZE,

.bankwidth = 1,

.phys = WINDOW\_ADDR,

};

static struct mtd\_partition xxx\_partitions[] = {

.name = "Drive A",

.offset = 0,

.size = 0x0e000,

};

#define NUM\_PARTITIONS ARRAY\_SIZE(xxx\_partitions)

static struct mtd\_info \*mymtd;

static int \_\_init init\_xxx\_map(void)

{

int rc = 0;

xxx\_map.virt = ioremap\_nocache(xxx\_map.phys, xxx\_map.size);

if(!xxx\_map.virt){

printk(KERN\_ERR"Failed to ioremap\_nocache\n");

rc = -EIO;

goto err2;

}

simple\_map\_init(&xxx\_map);

mymtd = do\_map\_probe("jedec\_probe", &xxx\_map);

if(!mymtd){

rc = -ENXIO;

goto err1;

}

mymtd->owner = THIS\_MODULE;

add\_mtd\_partitions(mymtd, xxx\_partitions, NUM\_PARTITIONS);

return 0;

err1:

map\_destroy(mymtd);

iounmap(xxx\_map.virt);

err2:

return rc;

}

static void \_\_exit cleanup\_xxx\_map(void)

{

if(mymtd){

del\_mtd\_partitions(mymtd);

map\_destroy(mymtd);

}

if(xxx\_map.virt){

iounmap(xxx\_map.virt);

xxx\_map.virt = NULL;

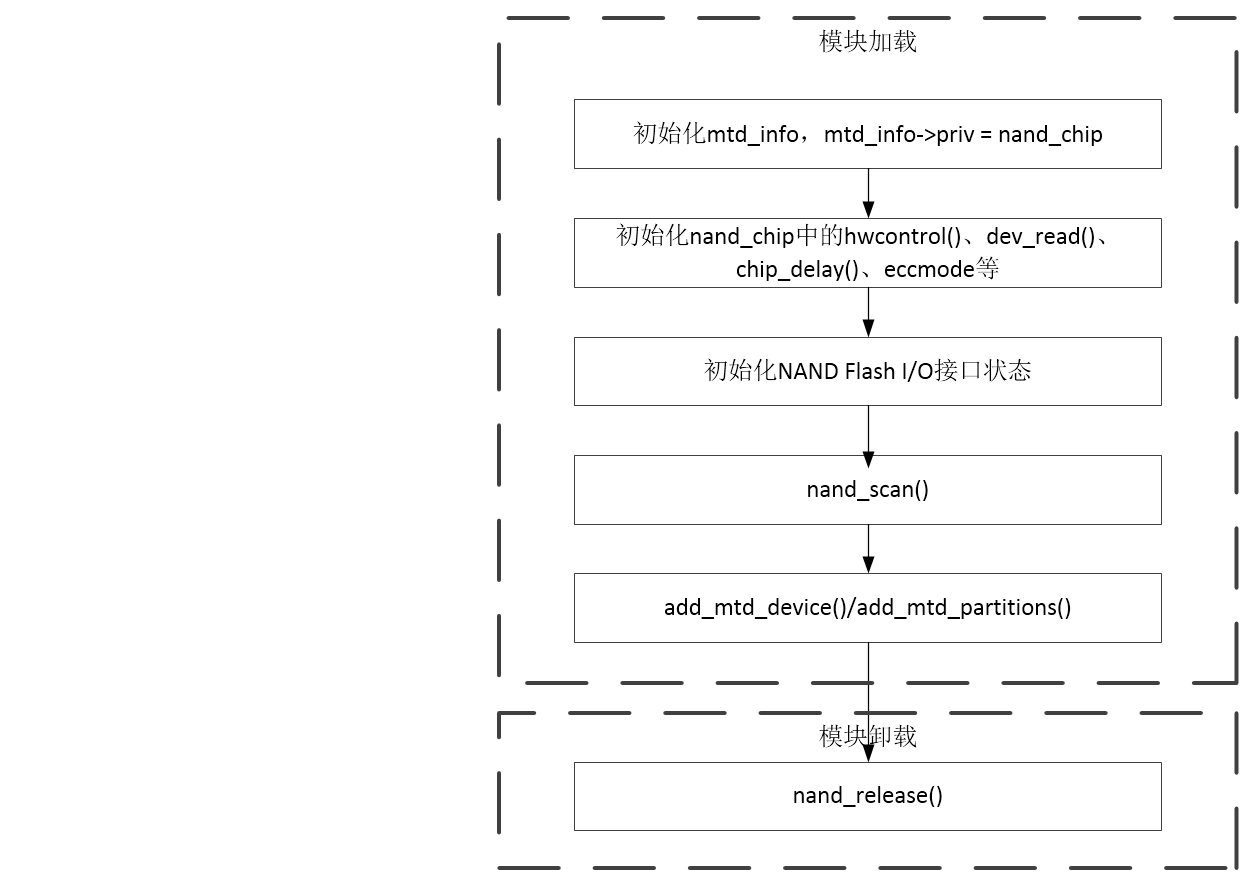
}

}

# Nand Flash驱动

下图就是基于MTD框架的NandFlash驱动的位置。

Nand Flash和NOR Flash类似，内核中已经在"drivers/mtd/nand/nand\_base.c"中实现了通用的驱动程序，驱动开发中不需要再实现mtd\_info中的read, write, read\_oob, write\_oob等接口，只需要构造并注册一个nand\_chip对象， 这个对象主要描述了一片flash芯片的相关信息，包括地址信息，读写方法，ECC模式，硬件控制等一系列底层机制。当下，编写一个NandFlash驱动的工作流程如下：



### nand\_chip

这个结构描述一个NAND Flash设备，通常藏在mtd\_info->priv中，以便在回调其中的接口的时候可以找到nand\_chip对象。

547 struct nand\_chip {

548 void \_\_iomem \*IO\_ADDR\_R;

549 void \_\_iomem \*IO\_ADDR\_W;

550

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

578

579 int chip\_delay;

580 unsigned int options;

581 unsigned int bbt\_options;

582

583 int page\_shift;

584 int phys\_erase\_shift;

585 int bbt\_erase\_shift;

586 int chip\_shift;

587 int numchips;

588 uint64\_t chipsize;

589 int pagemask;

590 int pagebuf;

591 unsigned int pagebuf\_bitflips;

592 int subpagesize;

593 uint8\_t bits\_per\_cell;

594 uint16\_t ecc\_strength\_ds;

595 uint16\_t ecc\_step\_ds;

596 int badblockpos;

597 int badblockbits;

598

599 int onfi\_version;

600 struct nand\_onfi\_params onfi\_params;

601

602 int read\_retries;

603

604 flstate\_t state;

605

606 uint8\_t \*oob\_poi;

607 struct nand\_hw\_control \*controller;

608

609 struct nand\_ecc\_ctrl ecc;

610 struct nand\_buffers \*buffers;

611 struct nand\_hw\_control hwcontrol;

612

613 uint8\_t \*bbt;

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

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

616

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

618

619 void \*priv;

620 };

struct nand\_chip  
--609-->NAND芯片的OOB分布和模式，如果不赋值，则会使用内核默认的OOB  
--580-->与具体的NAND 芯片相关的一些选项，如NAND\_BUSWIDTH\_16 等，可以参考<Linux/mtd/nand.h>  
--583-->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 ；  
--584-->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 ；  
--585-->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；  
--587-->表示系统中有多少片NAND 芯片；  
--588-->NAND 芯片的大小；  
--589-->计算page number 时的掩码，总是等于chipsize/page 大小 － 1 ；  
--590-->用来保存当前读取的NAND 芯片的page number ，这样一来，下次读取的数据若还是属于同一个page ，就不必再从NAND 芯片读取了，而是从data\_buf 中直接得到；  
--596-->表示坏块信息保存在oob 中的第几个字节。对于绝大多数的NAND 芯片，若page size> 512，那么坏块信息从Byte 0 开始存储，否则就存储在Byte 5 ，即第六个字节。  
--619-->私有数据

### nand\_scan()

准备好了一个nand\_chip，接下来的工作就是匹配驱动+注册分区表。  
NAND flash使用nand\_scan()来匹配驱动，这个函数会读取NAND芯片的ID，并根据mtd->priv即nand\_chip中的成员初始化mtd\_info。如果要分区，则以mtd\_info和mtd\_partition为参数调用add\_mtd\_partitions来添加分区信息。

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

### NAND Flash驱动模板

#define CHIP\_PHYSICAL\_ADDRESS ...

#define NUM\_PARTITIONS 2

static struct mtd\_partition partition\_info[] = {

{

.name = "Flash partition 1",

.info = 0,

.size = 8 \* 1024 \* 1024,

},

{

.name = "Flash partition 2",

offset = MTDPART\_OFS\_NEXT,

size = MTDPART\_SIZ\_FULL,

},

};

int \_\_init board\_init(void)

{

struct nand\_chip \*this;

int err = 0;

/\* 为MTD设备对象和nand\_chip分配内存 \*/

board\_mtd = kmalloc(sizeof(struct mtd\_info) + sizeof(struct nand\_chip),GFP\_KERNEL);

if(!board\_mtd){

printk("Unable to allocate NAND MTD device structure\n");

err = -ENOMEM;

goto out;

}

/\* 初始化结构体 \*/

memset((char \*)board\_mtd, 0 ,sizeof(struct mtd\_info) + sizeof(struct nand\_chip));

/\* 映射物理地址 \*/

baseaddr = (unsigned long) ioremap(CHIP\_PHYSICAL\_ADDRESS,1024);

if(!baseaddr){

printk("Ioremap to access NAND Chip failed\n");

err = -EIO;

goto out\_mtd;

}

/\* 获取私有数据(nand\_chip)指针 \*/

this = (struct nand\_chip \*)(&board\_mtd[1]);

/\* 将nand\_chip赋予mtd\_info私有指针 \*/

board\_mtd->priv = this;

/\* 设置NAND Flash的IO基地址 \*/

this->IO\_ADDR\_R = baseaddr;

this->IO\_ADDR\_W = baseaddr;

/\* 硬件控制函数 \*/

this->cmd\_ctrl = board\_hwcontrol;

/\* 初始化设备ready函数 \*/

this->dev\_ready = board\_dev\_ready;

/\* 扫描以确定设备的存在 \*/

if(nand\_scan(board\_mtd, 1)){

err = -ENXIO;

goto = out\_ior;

}

/\* 添加分区 \*/

add\_mtd\_partitions(board\_mtd,partition\_info,NUM\_PARTITIONS);

goto out;

out\_ior:

iounmap((void \*)baseaddr);

out\_mtd:

kfree(board\_mtd);

out:

return err;

}

static void \_\_exit board\_cleanup(void)

{

/\* 释放资源，注销设备 \*/

nand\_release(board\_mtd);

/\* unmap物理地址 \*/

iounmap((void \*)baseaddr);

/\* 释放MTD设备结构体 \*/

kfree(board\_mtd);

}

/\* 硬件控制 \*/

static void board\_hwcontrol(struct mtd\_info \*mtd, int dat,unsigned int ctrl)

{

...

if(ctrl & NAND\_CTRL\_CHANGE){

if(ctrl & NAND\_NCE){

}

}

...

}

/\*返回ready状态\*/

static int board\_dev\_ready(struct mtd\_info \*mtd)

{

return xxx\_read\_ready\_bit();

}

参考：

<https://blog.csdn.net/chungle2011/article/details/80585542>

<https://blog.csdn.net/hanmengaidudu/article/details/17120923>

<http://www.embeddedlinux.org.cn/emb-linux/kernel-driver/201703/30-6356.html>

<http://www.linux-mtd.infradead.org/faq/general.html>

<https://blog.csdn.net/geekcome/article/details/7202503>

<https://blog.csdn.net/geekcome/article/details/7205713>

https://blog.csdn.net/geekcome/article/details/7207908