块设备数据库

**宝德技术研究院 李磊**

# 前言

系统需要了解哪些块设备和字符设备可用,需要维护一个块设备/字符设备的数据库.本文主要是分析块设备数据库,字符设备的数据库和块设备几乎完全一样.

# 基本数据结构

对于块设备,有一个全局变量bdev\_map通过散列来实现块设备数据库.

static struct kobj\_map \*bdev\_map;

这是一个kobj\_map对象

struct kobj\_map {

struct probe {

struct probe \*next;

dev\_t dev;

unsigned long range;

struct module \*owner;

kobj\_probe\_t \*get;

int (\*lock)(dev\_t, void \*);

void \*data;

} \*probes[255];

struct mutex \*lock;

};

这个对象中定义了一个probe指针数组,数组中的每个元素都是散列中的backet.下面分析kobj\_map的成员.

◆next:指向下一个probe对象

◆dev:设备号

◆range:连续的此设备号的个数

◆owner:指向相关模块的指针

◆get:用于获取kobj的函数指针

◆data:指向struct genhd

# 散列表的组织

块设备数据库散列表和内核中其他散列表的组织形式有所区别,其特点有

●backet的值由主设备号决定,在0-255之间

●backet接的链表是单向链表

●链表中的元素是根据range从小到大排列的

散列结构如图一



图一

笔者内核中输出的散列表实际内容如下

**backet [0]: 0xffffffff**

**backet [1]: 0x1 0x1 0x1 0x1 0x1 0x1 0x1 0x1 0xffffffff**

**backet [2]: 0x1 0x100 0xffffffff**

**backet [3]: 0xffffffff**

**backet [4]: 0xffffffff**

**backet [5]: 0xffffffff**

**backet [6]: 0xffffffff**

**backet [7]: 0xffffffff**

**backet [8]: 0x10 0xffffffff**

**backet [9]: 0x1 0x100 0xffffffff**

**backet [10]: 0xffffffff**

我们来看看设备号是如何注册进bdev\_map的，内核通过add\_disk将一个block device注册到bdev\_map中。目前我们只需要关注blk\_register\_region这个函数。

void **add\_disk**(struct gendisk \*disk)

{

disk->flags |= GENHD\_FL\_UP;

**blk\_register\_region**(**MKDEV**(disk->major, disk->first\_minor),

disk->minors, **NULL**, exact\_match, exact\_lock, disk);

**register\_disk**(disk);

**blk\_register\_queue**(disk);

}

blk\_register\_region会调用kobj\_map来实现插入hash操作。

linux内核通过函数blk\_register\_region将设备号添加进入bdev\_map，初看起来这除了查找设备号外，和设备本身并没任何联系。但实际上，有个函数 -- exact\_match应该引起我们的注意。我们来看看这个函数的定义。

static struct kobject \***exact\_match**(dev\_t dev, int \*part, void \*data)

{

struct gendisk \*p = data;

return &p->kobj;

}

这个函数的作用是将data转换成struct gendisk对象后返回其kobject对象。回过头来看看kobj\_map函数，中间这段代码应该引起关注，这里的probe就是exact\_match函数指针。

for (i = 0; i < n; i++, p++) {

p->owner = module;

p->**get** = probe;

p->lock = lock;

p->dev = dev;

p->range = range;

p->data = data;

}

所以在hash表中找到设备号对应的probe对象，就能通过get这个成员找到struct gendisk的kobject对象。内核其他地方只要获取了这个kobject对象就能通过宏to\_disk获取gendisk对象。

**blk\_register\_region**

举个例子，在ramfs文件系统的分配Inode过程中，会根据不同的文件类型给inode赋予不同的文件操作函数集

struct inode \***ramfs\_get\_inode**(struct super\_block \*sb, int mode, dev\_t dev)

{

...

default:

**init\_special\_inode**(inode, mode, dev);

break;

...

｝

void **init\_special\_inode**(struct inode \*inode, umode\_t mode, dev\_t rdev)

{

inode->i\_mode = mode;

if (**S\_ISCHR**(mode)) {

inode->i\_fop = &def\_chr\_fops;

inode->i\_rdev = rdev;

} else if (**S\_ISBLK**(mode)) {

inode->i\_fop = &def\_blk\_fops;

inode->i\_rdev = 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 i\_mode (%o)\n",

mode);

}

在函数init\_special\_inode中可以看到，如果Inode表示的是一个block device，那么i\_fop就赋值为def\_blk\_fops.

const struct file\_operations def\_blk\_fops = {

.**open** = blkdev\_open,

.release = blkdev\_close,

.llseek = block\_llseek,

.**read** = generic\_file\_read,

.**write** = blkdev\_file\_write,

.aio\_read = generic\_file\_aio\_read,

.aio\_write = blkdev\_file\_aio\_write,

.mmap = generic\_file\_mmap,

.fsync = block\_fsync,

.unlocked\_ioctl = block\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = compat\_blkdev\_ioctl,

#endif

.readv = generic\_file\_readv,

.writev = generic\_file\_write\_nolock,

.sendfile = generic\_file\_sendfile,

.splice\_read = generic\_file\_splice\_read,

.splice\_write = generic\_file\_splice\_write,

};

下图是blkdev\_open的函数调用图



图二

查看一下get\_gendisk实现，可以看出此函数在通过kobj\_lookup在bdev\_map中根据设备号找到设备的kobject对象后，然后使用函数to\_disk就获取了gendisk指针。这样一个从设备号获取设备对象的过程就完成了。

struct gendisk \***get\_gendisk**(dev\_t dev, int \*part)

{

struct kobject \*kobj = **kobj\_lookup**(bdev\_map, dev, part);

return kobj ? **to\_disk**(kobj) : **NULL**;

}