# 一、驱动模块

## 1、设备类型

设备分为三种类型：块设备、字符设备、网络设备。

块设备（blkdev）是可寻址，寻址以块为单位，块大小随设备不同而不同；块设备通常支持重定位操作（即对数据的随机访问），例如：硬盘，光盘，flash等。

字符设备（cdev）是不可寻址，仅提供数据的流式访问，就是一个个字符，或者一个个字节，例如：键盘、鼠标等。miscdev简化的字符设备，提供杂项设备。

网络设备最常见的是以太网，提供对网络的访问。

如图所示：



## 2、模块例程

Linux内核是模块化组成，允许驱动模块编译进内核（无法卸载），也允许内核在运行时动态加载或卸载驱动模块。

下面是一个简单的驱动模块driver\_module.c，后续都会以该模块作为基础进行测试相关内容：

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

static int \_\_init driver\_module\_init(void)

{

printk("This is a driver module!\n");

return 0;

}

static void \_\_exit driver\_module\_exit(void)

{

printk("driver module is exit!\n");

}

module\_init(driver\_module\_init);

module\_exit(driver\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("coolice");

实现相应的Makefile，其内容如下：

#this module is a example of driver

PWD=$(shell pwd)

LINUX\_SRC\_ROOT=/xxx/linux\_v3.10.y/Trunk/src

obj-m := driver\_test.o

driver\_test-y := driver\_module.o

module:

make -C $(LINUX\_SRC\_ROOT) M=$(PWD) modules

clean:

make -C $(LINUX\_SRC\_ROOT) M=$(PWD) clean

编译完成后，就会在相同目录下产生一个driver\_test.ko。

## 3、模块参数

Linux允许驱动程序声明参数，从而用户可以在系统启动或者模块装载时再指定参数值。它提供了参数的形式：

### （1）module\_param

module\_param(name, type, perm)

name：参数名

type：参数数据类型

perm：参数在sysfs下对应文件权限，可以是八进制格式

### （2）module\_param\_named

module\_param\_named(name, variable, type, perm)

name：外部可见的参数名称

variable：参数对应的内部变量名称

type：参数类型

perm：sysfs文件权限

### （3）module\_param\_string

module\_param\_string(name, string, len, perm)

name：外部可见的参数名称

string：对应内部变量参数

len：string缓存区长度

perm：sysfs文件权限

### （4）module\_param\_array

module\_param\_array(name, type, nump, perm)

name：外部参数以及对应内部参数变量名

type：数据类型

nump：该整型存放数组项数

perm：sysfs文件权限

可以使用MODULE\_PARM\_DESC()描述参数。

在driver\_module驱动例程中添加上面的模块参数，如下：

static int m\_param = 0;

module\_param(m\_param, int, 0600);

static int m\_param\_name=0;

module\_param\_named(param\_name, m\_param\_name, int, 0600);

static char m\_param\_string[32];

module\_param\_string(param\_string, m\_param\_string, 32, 0644);

MODULE\_PARM\_DESC(param\_string, "param string test.");

static int m\_param\_array[32];

static int nr\_array = 0;

module\_param\_array(m\_param\_array, int, &nr\_array, 0644);

重新编译模块，加载后打印如下：

m\_param:1

m\_param\_name:1

m\_param\_string:1234

m\_param\_array: 1 2 3 4

# 二、设备模型

2.6内核增加了一个引人注目的新特性——统一设备模型。它提供了一个独立的机制专门来表示设备，并描述其在系统中的拓扑结构，具有如下优点：

A．代码重复最小化

B．提供诸如引用计数这样的统一机制

C．列举系统中所有的设备，观察他们的状态，并且查看它们连接的总线

D．将系统中的全部设备结构以树的形式完整、有效的展现出来（包括所有的总线和内部连接）

E．将设备和其对应的驱动联系起来，反之亦然

F．设备按照类型加以归类，比如分类为输入设备，而无需理解物理设备的拓扑结构

G．沿设备树的叶子向其根的方向依次遍历，以保证能以正确顺序关闭各设备的电源

下图是摘自《深入理解linux内核》的层次模型例子:



## 1、kobject

### （1）数据机构

Linux设备模型的核心是使用Bus、Class、Device、Driver四个核心数据结构，将大量的、不同功能的硬件设备（以及驱动该硬件设备的方法），以树状结构的形式，进行归纳、抽象，从而方便Kernel的统一管理。

将这些设备数据结构共同的功能统一抽象出来，于是就有了kobject（kernel object）,它由struct kobject结构体表示，定义如下（include/linux/kobject.h）：

struct kobject {

const char \*name; //kobject名称

struct list\_head entry; //kobject插入的链表

struct kobject \*parent; //指向kobject的父对象

struct kset \*kset; //指向kobject属于的kset

struct kobj\_type \*ktype; //指向kobject对象的类型

struct sysfs\_dirent \*sd; //指向与kobject相对应的sysfs文件dentry数据结构

struct kref kref; //引用计数

unsigned int state\_initialized:1;

unsigned int state\_in\_sysfs:1;

unsigned int state\_add\_uevent\_sent:1;

unsigned int state\_remove\_uevent\_sent:1;

unsigned int uevent\_suppress:1;

};

Kobject通常是嵌入其他结构中，其单独使用意义不大。当kobject被嵌入到其他结构中时，该结构便拥有了kobject提供的标准功能。主要功能如下：

A．通过parent指针，可以将所有Kobject以层次结构的形式组合起来。

B．使用一个引用计数（reference count），来记录Kobject被引用的次数，并在引用次数变为0时把它释放。

C．和sysfs虚拟文件系统配合，将每一个Kobject及其特性，以文件的形式，开放到用户空间

### （2）操作接口

创建kobject对象：

struct kobject \*kobject\_create(void)

struct kobject \*kobject\_create\_and\_add(const char \*name, struct kobject \*parent)

递增和递减引用计数：

struct kobject \*kobject\_get(struct kobject \*kobj)

void kobject\_put(struct kobject \*kobj)

## 2、kset

kset是同类kobject对象的集合体。把它看成一个容器，将所有相关的kobject对象置于同一位置，其定义如下（include/linux/kobject.h）：

struct kset {

struct list\_head list; //包含在kset中的kobject链表的首部

spinlock\_t list\_lock;

struct kobject kobj; //嵌入的kobject结构

const struct kset\_uevent\_ops \*uevent\_ops; //处理集合中kobject对象的热插拔操作

};

## 3、ktype

ktype描述kobject所具有的普通特性。其定义如下（include/linux/kobject.h）：

struct kobj\_type {

void (\*release)(struct kobject \*kobj); //当kobject引用计数减到0时要被调用的析构函数

const struct sysfs\_ops \*sysfs\_ops; //描述sysfs文件读写时的操作

struct attribute \*\*default\_attrs; //指向一个attribute结构体数组，kobject相关的默认属性

const struct kobj\_ns\_type\_operations \*(\*child\_ns\_type)(struct kobject \*kobj);

const void \*(\*namespace)(struct kobject \*kobj);

};

# 三、sysfs和procfs

## 1、sysfs

sysfs文件系统是一个处于内存中的虚拟文件系统，为我们提供了kobject对象层次结构的试图。sysfs根目录/sys下包含很多目录，其意义如下：

devices：系统中设备拓扑结构视图，直接映射出了内核设备结构体的组织层次。该目录是最重要的目录将设备模型导出到用户空间，目录结构就是系统中实际的设备拓扑，其他目录中的很多数据都是devices下面的符号链接。

dev：维护一个按字符设备和块设备的主次号码（major:minor），是真实设备（devices）下的软链接文件

bus：内核设备按总线分层放置的目录结构，devices中的所有设备都是连接到某种总线之下，统一设备模型的一部分

class：内核设备按照功能分类的设备模型，统一设备模型的一部分

block：系统中当前所有块设备所在，已经过时，保留为了向前兼容，新的已经移到class/block下。

firmware：系统加载固件机制的对用户空间的接口

fs：描述系统中所有文件系统，包括文件系统本身和文件系统分类存放的已挂载点。部分调试还在/proc/sys/fs下。

kernel：内核所有可调整参数的位置。部分还在/proc/sys/kernel。

module：系统中所有模块的信息

power：系统中电源选项

### （1）sysfs添加和删除kobject

每一个kobject都会对应sysfs中的一个目录，因此在将Kobject添加到Kernel时，create\_dir接口会调用sysfs文件系统的创建目录接口，创建和Kobject对应的目录。主要操作接口：

int kobject\_add(struct kobject \*kobj, struct kobject \*parent, const char \*fmt, ...)

struct kobject \*kobject\_create\_and\_add(const char \*name, struct kobject \*parent)

void kobject\_del(struct kobject \*kobj)

### （2）sysfs添加文件

kobject被映射为文件目录，而且所有的对象层次结构都优雅的、一个不少的映射成sys下的目录结构，但是没有提供实际的数据文件，这部分由kobject的属性attribute决定。

kobject的属性就是特定数据类型变量的属性，它可以是任何东西，名称、一个内部变量、一个字符串等等。kobject的所有属性，都在它对应的sysfs目录下以文件的形式呈现。这些文件一般是可读、写的，而kernel中定义了这些属性的模块，会根据用户空间的读写操作，记录和返回这些attribute的值。例如某个driver定义了一个变量，却希望用户空间程序可以修改该变量，以控制driver的运行行为，那么就可以将该变量以sysfs attribute的形式开放出来。

A．默认属性

ktype字段提供了默认的文件集合，其中的default\_attrs字段描述的属性负责将内核数据映射成sysfs中的文件。

struct attribute {

const char \*name; //属性名称

umode\_t mode; //权限

};

最终出现在sysfs文件中的文件名就是上面的name。

ktype的sysfs\_ops字段描述属性的操作方法：

struct sysfs\_ops {

ssize\_t (\*show)(struct kobject \*, struct attribute \*,char \*); //读取sysfs文件时调用

ssize\_t (\*store)(struct kobject \*,struct attribute \*,const char \*, size\_t); //写sysfs文件时该方法调用

const void \*(\*namespace)(struct kobject \*, const struct attribute \*);

};

B．创建和删除新属性

由kobject的ktype提供默认属性一般够用，遇到特殊情况内核通过下面接口实现在默认集合之上在添加新属性：

int sysfs\_create\_file(struct kobject \* kobj, const struct attribute \* attr)

删除一个属性：

void sysfs\_remove\_file(struct kobject \*kobj, const struct attribute \*attr);

### （3）sysfs约定

当前sysfs文件系统代替以前需要有ioctl()（作用于设备节点）和procfs文件系统完成的功能。为了保持简洁、直观，开发者需要遵守一下约定：

A．sysfs属性应该保持每个文件只导出一个值，该值应该是文本形式而且映射为简单C语言类型

B．sysfs要以一个清晰的层次组织数据，父子关系要正确才能将kobject层次结构直观映射到sysfs树中。

C．sysfs提供内核到用户空间的服务，类似用户空间的ABI作用，任何情况下都不应该改变现有的文件。

### （4）内核事件层

它实现了内核到用户的消息通知系统，借助kobject和sysfs实现证明相当理想。内核事件由内核空间传递到用户空间需要经过netlink，它是一个用于传送网络信息的多点传送套接字，方法就是用户空间实现一个系统后台服务用于监听套接字，处理任何读到的信息，并将事件传送到系统栈里。

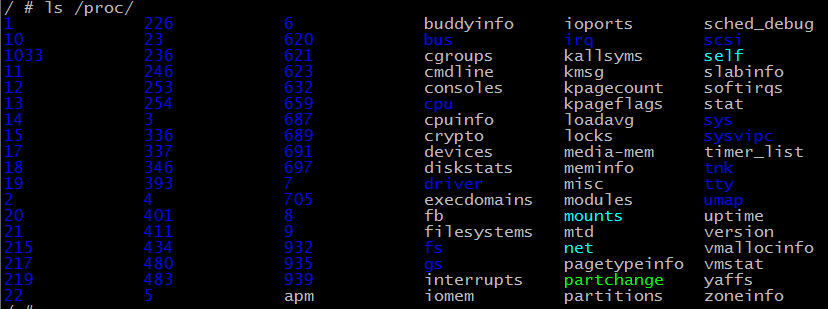
在内核代码中向用户空间发送信号使用函数kobject\_uevent()：

int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action)



## 2、procfs

procfs文件系统是进程文件系统的缩写，也是一个伪文件系统，是一种特殊的、只存在于内存的文件系统，开始是主要用于用户空间访问进程信息，如今经过不断的发展，其已发展成一个用户空间与内核交换数据修改系统行为的接口。



通过这些文件我们可以得到计算机系统的一些基本信息；而一些数字目录，每个数字对应一个进程的pid，目录里面包含的文件描述着这个进程的方方面面，当然这些文件都是只读的，我们并不能更改这些文件，只能用于获取进程的运行信息。

## 3、sysfs测试例程

在第一章节模块例程的基础上添加sysfs测试例子，其源码如下：

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#define DEVICE\_MODEL 1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* defined marco \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//自定义模块属性

struct devm\_attribute {

struct attribute attr;

ssize\_t (\*show)(char \*buf);

ssize\_t (\*store)(const char \*buf, size\_t count);

};

#define to\_devm\_attr(\_attr) container\_of(\_attr, struct devm\_attribute, attr)

#if DEVICE\_MODEL

static struct kset \*devm\_kset = NULL;

static struct kobject \*devm\_kobj = NULL;

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* defined function \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#if DEVICE\_MODEL

//自定义属性具体实现

static unsigned int show = 0;

static ssize\_t show\_devm\_attr(char \*buf)

{

return sprintf(buf, "%d\n", show);

}

static ssize\_t store\_devm\_attr(const char \*buf, size\_t count)

{

if(buf[0] == '0')

show = 0;

else

show = 1;

return count;

}

static struct devm\_attribute devm\_attr =

{

.attr = {

.name = "devm\_attr",

.mode = 0666,

},

.show = show\_devm\_attr,

.store = store\_devm\_attr,

};

//内核提供的标准属性接口，内容自己实现

static ssize\_t devm\_attr\_show(struct kobject \*kobj, struct attribute \*attr, char \*buf)

{

ssize\_t ret = 0;

struct devm\_attribute \*devm\_attr = to\_devm\_attr(attr);

if(devm\_attr->show)

ret = devm\_attr->show(buf);

return ret;

}

static ssize\_t devm\_attr\_store(struct kobject \*kobj, struct attribute \*attr, const char \*buf, size\_t count)

{

ssize\_t ret = 0;

struct devm\_attribute \*devm\_attr = to\_devm\_attr(attr);

if(devm\_attr->store)

ret = devm\_attr->store(buf, count);

return ret;

}

static struct sysfs\_ops devm\_sysfs\_ops =

{

.show = devm\_attr\_show,

.store = devm\_attr\_store,

};

static struct kobj\_type devm\_ktype =

{

.sysfs\_ops = &devm\_sysfs\_ops,

};

static int devm\_register(void)

{

//在sys目录下创建自己的一级目录

devm\_kset = kset\_create\_and\_add("devm", NULL, NULL);

if(!devm\_kset)

return -1;

//实际使用的模块目录

devm\_kobj = kobject\_create\_and\_add("devtest", &devm\_kset->kobj);

if(!devm\_kobj)

goto dev\_kobj\_err;

devm\_kobj->kset = devm\_kset;

devm\_kobj->ktype = &devm\_ktype;

//添加属性文件

if(sysfs\_create\_file(devm\_kobj, &devm\_attr.attr))

printk("sysfs create file failed\n");

return 0;

dev\_kobj\_err:

kset\_unregister(devm\_kset);

return -1;

}

static void devm\_unregister(void)

{

sysfs\_remove\_file(devm\_kobj, &devm\_attr.attr);

kobject\_del(devm\_kobj);

if(devm\_kset)

kset\_unregister(devm\_kset);

}

#endif

static int \_\_init driver\_module\_init(void)

{

printk("This is a driver module!, parameter:\n");

#if DEVICE\_MODEL

if(devm\_register())

printk("create device model failed\n");

#endif

return 0;

}

static void \_\_exit driver\_module\_exit(void)

{

printk("driver module is exit!\n");

#if DEVICE\_MODEL

devm\_unregister();

#endif

}

module\_init(driver\_module\_init);

module\_exit(driver\_module\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("coolice");

测试结果如下：

/ # ls /sys/devm/

Devtest

/ # cat /sys/devm/devtest/devm\_attr

1

/ # echo 0 > /sys/devm/devtest/devm\_attr

/ # cat /sys/devm/devtest/devm\_attr

0

# 四、bus/class/device/driver

## 1、bus

总线是CPU和一个或多个设备之间信息交互的通道。为了方便设备模型的抽象，所有的设备都应该连接到总线。

Bus模块的主要功能如下：

1. bus的注册和注销

B．本bus下有device或device\_driver注册到内核时的处理

C．本bus下有device或device\_driver从内核注销时的处理

D．device\_driver的probe处理

E．管理bus下的所有device和device\_driver

Bus由struct bus\_type结构体定义（include/linux/device.h），细节如下：

struct bus\_type {

const char \*name; //bus的名称，会在sysfs中以目录的形式存在

const char \*dev\_name; //用于子系统枚举设备，类似("foo%u", dev->id)，如sda1 sda2

struct device \*dev\_root; //默认的总线父设备

struct bus\_attribute \*bus\_attrs; //总线默认属性

struct device\_attribute \*dev\_attrs; //总线所有设备默认属性

struct driver\_attribute \*drv\_attrs; //总线所有设备驱动默认属性

/\* 一个由具体的bus driver实现的回调函数。当任何属于该Bus的device或者device\_driver添加到内核时，内核都会调用该接口，如果新加的device或device\_driver匹配上了自己的另一半的话，该接口要返回非零值，此时Bus模块的核心逻辑就会执行后续的处理\*/

int (\*match)(struct device \*dev, struct device\_driver \*drv);

/\*一个由具体的bus driver实现的回调函数。当任何属于该Bus的device，发生添加、移除或者其它动作时，Bus模块的核心逻辑就会调用该接口，以便bus driver能够修改环境变量\*/

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

/\*下面两个回调函数的存在是非常有意义的。可以想象一下，如果需要probe（其实就是初始化）指定的device话，需要保证该device所在的bus是被初始化过、确保能正确工作的。这就要就在执行device\_driver的probe前，先执行它的bus的probe。remove的过程相反\*/

int (\*probe)(struct device \*dev);

int (\*remove)(struct device \*dev);

void (\*shutdown)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct dev\_pm\_ops \*pm; //电源管理

struct iommu\_ops \*iommu\_ops;

struct subsys\_private \*p; //驱动私有数据，只有驱动可以访问它

struct lock\_class\_key lock\_key;

};

struct subsys\_private {

struct kset subsys; //子系统目录的kset

struct kset \*devices\_kset; //子系统里devices目录的kset

struct list\_head interfaces; //该bus的所有interface

struct mutex mutex;

struct kset \*drivers\_kset; //子系统里drivers目录的kset

struct klist klist\_devices; //本bus下所有devices的链表

struct klist klist\_drivers; //本bus下所有device\_driver的链表

struct blocking\_notifier\_head bus\_notifier;

unsigned int drivers\_autoprobe:1; //bus下的drivers或device是否自动probe

struct bus\_type \*bus; //指向bus

struct kset glue\_dirs;

struct class \*class; //指向class

};

内核存在三种比较特殊的bus：system bus\virtual bus\platform bus，不是实际存在的bus（像USB/PCI等），为了方便设备模型抽象而虚构的。

system bus：旧版内核用于抽象设备（CPU、Timer等），而在新版内核中被抛弃，只是为了兼容而存在，不建议使用。

virtual bus：一个比较新的bus，主要用来抽象那些虚拟设备，所谓的虚拟设备，是指不是真实的硬件设备，而是用软件模拟出来的设备，在/sys/devices/virtual/目录下可以看到相关虚拟设备。

Platform bus：它主要抽象集成在CPU（SOC）中的各种设备。这些设备直接和CPU连接，通过总线寻址和中断的方式，和CPU交互信息。

## 2、class

在Linux设备模型中，Class的概念非常类似面向对象程序设计中的Class（类），它主要是集合具有相似功能或属性的设备，这样就可以抽象出一套可以在多个设备之间共用的数据结构和接口函数。因而从属于相同Class的设备的驱动程序，就不再需要重复定义这些公共资源，直接从Class中继承即可。

主动功能：

A．在sys/class目录下，创建一个class类目录

B．在本目录下创建一个属于该class设备的符号连接，这样就可以在本class目录下，访问该设备的所有特性。

C．device在sysfs目录下，也会创建一个subsystem的符号链接，连接到本class的目录。

Class由struct class结构体定义（include/linux/device.h 332），细节如下：

struct class {

const char \*name; //class名字，会出现在sys/class/目录下

struct module \*owner;

struct class\_attribute \*class\_attrs; //class默认属性，sys/class/xxx下创建相应的属性文件

struct device\_attribute \*dev\_attrs; //属于这个class的设备默认属性，sys/class/xxx下创建相应的属性文件

struct bin\_attribute \*dev\_bin\_attrs; //属于这个class的设备默认二进制属性

struct kobject \*dev\_kobj; //表示该class下的设备在/sys/dev/下的目录

int (\*dev\_uevent)(struct device \*dev, struct kobj\_uevent\_env \*env); //当该class下有设备发生变化时，会调用class的uevent回调函数

char \*(\*devnode)(struct device \*dev, umode\_t \*mode);

void (\*class\_release)(struct class \*class);

void (\*dev\_release)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct kobj\_ns\_type\_operations \*ns\_type;

const void \*(\*namespace)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

struct subsys\_private \*p; //同bus的subsys\_private

};

## 3、device与driver

device抽象系统中所有的硬件设备，描述它的名字、属性、从属的Bus、从属的Class等信息。而用driver抽象硬件设备的驱动程序，它包含设备初始化、电源管理相关的接口实现。

Linux内核中的驱动开发就是要开发指定的软件以驱动指定的设备，通过device和device\_driver两个数据结构分别表示它们，主要完成如下功能：

A．设备及设备驱动在kernel中的抽象、使用和维护

B．设备即设备驱动的注册、加载、初始化原理

C．设备模型在实际驱动开发过程中的使用方法。

device数据结构的详细定义如下（include/linux/device.h）：

struct device {

struct device \*parent; //该设备的父设备，一般是该设备所从属的bus、controller等设备

struct device\_private \*p; //一个用于struct device的私有数据结构指针，该指针中会保存子设备链表、用于添加到bus/driver/prent等设备中的链表头等等

struct kobject kobj; //该数据结构对应的struct kobject

const char \*init\_name; //该设备的初始名称，可以初始化给定，也可以由bus name + device ID方式创造

const struct device\_type \*type;

struct mutex mutex;

struct bus\_type \*bus; //设备归属的总线

struct device\_driver \*driver; //设备对应的驱动

void \*platform\_data; //保存具体的平台相关数据，在driver的probe中比较常见

struct dev\_pm\_info power;

struct dev\_pm\_domain \*pm\_domain;

#ifdef CONFIG\_PINCTRL

struct dev\_pin\_info \*pins;

#endif

#ifdef CONFIG\_NUMA

int numa\_node; /\* NUMA node this device is close to \*/

#endif

u64 \*dma\_mask; /\* dma mask (if dma'able device) \*/

u64 coherent\_dma\_mask;

struct device\_dma\_parameters \*dma\_parms;

struct list\_head dma\_pools;

struct dma\_coherent\_mem \*dma\_mem;

#ifdef CONFIG\_CMA

struct cma \*cma\_area;

#endif

/\* arch specific additions \*/

struct dev\_archdata archdata;

struct device\_node \*of\_node; /\* associated device tree node \*/

struct acpi\_dev\_node acpi\_node; /\* associated ACPI device node \*/

dev\_t devt; //设备号（Major和Minor组成），用于在sys/dev/下面相应的设备目录

u32 id; /\* device instance \*/

spinlock\_t devres\_lock;

struct list\_head devres\_head;

struct klist\_node knode\_class;

struct class \*class; //该设备属于的class

const struct attribute\_group \*\*groups; //设备默认的attribute集合，注册时自动在sysfs中创建对应的文件

void (\*release)(struct device \*dev);

struct iommu\_group \*iommu\_group;

};

driver数据结构体的详细描述如（include/linux/device.h）：

struct device\_driver {

const char \*name; //该driver的名字

struct bus\_type \*bus; //该driver对应的bus

struct module \*owner;

const char \*mod\_name; /\* used for built-in modules \*/

bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

const struct of\_device\_id \*of\_match\_table;

const struct acpi\_device\_id \*acpi\_match\_table;

int (\*probe) (struct device \*dev); //寻找匹配设备接口

int (\*remove) (struct device \*dev); //移除设备

void (\*shutdown) (struct device \*dev);

int (\*suspend) (struct device \*dev, pm\_message\_t state);

int (\*resume) (struct device \*dev);

const struct attribute\_group \*\*groups; //driver默认的attribute集合，注册时自动在sysfs中创建对应的文件

const struct dev\_pm\_ops \*pm;

struct driver\_private \*p; //驱动私有数据，其他驱动不能访问

};

在设备模型的框架下，驱动开发主要完成2个步骤：

A．分配一个struct device类型的变量，填充必要的信息后，把它注册到内核中

B．分配一个struct device\_driver类型的变量，填充必要的信息后，把它注册到内核中

内核会调用device\_driver的probe将device和device\_driver进行匹配操作，其要求如下：

A．device和device\_driver必须具备相同的名称，内核才能完成匹配操作，进而调用device\_driver中的相应接口。这里的同名，作用范围是同一个bus下的所有device和device\_driver

B．device和device\_driver必须挂载在一个bus之下

C． driver开发者可以在struct device变量中，保存描述设备特征的信息，如寻址空间、依赖的GPIOs等，因为device指针会在执行probe等接口时传入，这时driver就可以根据这些信息，执行相应的逻辑操作了。

D．使用内核已经封装过的接口，如platform等，简单、易用。

内核执行驱动probe发生在一下几种情况：

A．将device注册到内核时自动触发（device\_register, device\_add, device\_create\_vargs, device\_create）

B．将driver注册到内核时自动触发（driver\_register）

C．手动查找同一bus下的所有device\_driver，如果有和指定device同名的driver，执行probe操作（device\_attach）

D．手动查找同一bus下的所有device，如果有和指定driver同名的device，执行probe操作（driver\_attach）

E．自行调用driver的probe接口，并在该接口中将该driver绑定到某个device结构中----即设置dev->driver（device\_bind\_driver）

## 4、例子分析

下面以网卡stmmac驱动为例，对设备模型进行简单讲解，该驱动通过platform的方式注册到内核。

### （1）platform总线注册

首先，注册platform总线。

kernel\_init -> kernel\_init\_freeable -> do\_basic\_setup -> driver\_init(驱动模型初始化) -> platform\_bus\_init()：

int \_\_init platform\_bus\_init(void)

{

int error;

early\_platform\_cleanup();

error = device\_register(&platform\_bus); //注册platform bus设备，主要完成kobject初始化，sys下面相关目录创建

if (error)

return error;

error = bus\_register(&platform\_bus\_type); //注册platform bus总线，主要完成kobject初始化，sys下面相关目录创建

if (error)

device\_unregister(&platform\_bus);

return error;

}

经过初始化后，在sys目录下可以看到相应信息：

ls /sys/devices/目录下多了一个platform目录

ls sys/bus/目录下多了一个platform目录

/ # ls /sys/bus/platform/

devices drivers\_autoprobe uevent

drivers drivers\_probe

platform\_bus和platform\_bus\_type的初始化如下：

struct device platform\_bus = {

.init\_name = "platform",

};

struct bus\_type platform\_bus\_type = {

.name = "platform",

.dev\_attrs = platform\_dev\_attrs,

.match = platform\_match, //该函数用于匹配通过platform接口注册的设备和驱动

.uevent = platform\_uevent,

.pm = &platform\_dev\_pm\_ops,

};

static int platform\_match(struct device \*dev, struct device\_driver \*drv)

{

struct platform\_device \*pdev = to\_platform\_device(dev);

struct platform\_driver \*pdrv = to\_platform\_driver(drv);

/\* 先尝试OF(open firmware)设备树方式进行匹配 \*/

if (of\_driver\_match\_device(dev, drv))

return 1;

/\* 然后尝试ACPI接口方式匹配 \*/

if (acpi\_driver\_match\_device(dev, drv))

return 1;

/\* 接着进行ID表的方式进行匹配 \*/

if (pdrv->id\_table)

return platform\_match\_id(pdrv->id\_table, pdev) != NULL;

/\* 最后使用简单的驱动和设备名字方式进行匹配 \*/

return (strcmp(pdev->name, drv->name) == 0);

}

### （2）注册stmmac设备

在drivers/net/ethernet/stmmac/stmmac\_main.c中stmmac\_init\_module接口完成设备和驱动的注册，本小节讲解设备注册部分：

首先，定义一个platform\_device结构体：

static struct platform\_device stmmac\_platform\_device = {

.name = STMMAC\_RESOURCE\_NAME, //“stmmaceth”

.id = 0,

.dev = {

.platform\_data = &stmmac\_ethernet\_platform\_data, //前面已经讲过，平台特有数据

.dma\_mask = &stmmac\_dmamask,

.coherent\_dma\_mask = DMA\_BIT\_MASK(32),

.release = stmmac\_platform\_device\_release,

},

.num\_resources = ARRAY\_SIZE(stmmac\_resources), //相关资源

.resource = stmmac\_resources,

};

static int \_\_init stmmac\_init\_module(void)

{

int ret;

…

ret = platform\_device\_register(&stmmac\_platform\_device);

…

return ret;

}

int platform\_device\_register(struct platform\_device \*pdev)

{

device\_initialize(&pdev->dev); //初始化platform设备的设备模型kobject等信息

arch\_setup\_pdev\_archdata(pdev);

return platform\_device\_add(pdev); //注册设备

}

int platform\_device\_add(struct platform\_device \*pdev)

{

int i, ret;

…

if (!pdev->dev.parent)

pdev->dev.parent = &platform\_bus; //注册设备的父设备为platform\_bus

pdev->dev.bus = &platform\_bus\_type; //注册设备挂载的总线为platform\_bus

…

ret = device\_add(&pdev->dev); //向内核添加设备

if (ret == 0)

return ret;

…

}

int device\_add(struct device \*dev)

{

…

/\* 以下注册设备模型通用层，会在/sys/devices/platform下产生相应的设备目录: ls /sys/devices/platform/stmmaceth.0/ \*/

error = kobject\_add(&dev->kobj, dev->kobj.parent, NULL);

if (error)

goto Error;

/\* notify platform of device entry \*/

if (platform\_notify)

platform\_notify(dev);

error = device\_create\_file(dev, &uevent\_attr);

if (error)

goto attrError;

…

bus\_probe\_device(dev); //总线为设备匹配对应的驱动，该函数调用device\_attach

…

}

int device\_attach(struct device \*dev)

{

…

//扫描总线所有驱动，查找匹配该设备的驱动

ret = bus\_for\_each\_drv(dev->bus, NULL, dev, \_\_device\_attach);

pm\_request\_idle(dev);

…

}

static int \_\_device\_attach(struct device\_driver \*drv, void \*data)

{

struct device \*dev = data;

/\* static inline int driver\_match\_device(struct device\_driver \*drv,

struct device \*dev)

{

return drv->bus->match ? drv->bus->match(dev, drv) : 1;

}

该处的bus->match就是上一节的platform\_bus\_type. platform\_match函数

\*/

if (!driver\_match\_device(drv, dev))

return 0;

return driver\_probe\_device(drv, dev); //完成设备驱动绑定，调用驱动的probe实现对设备相关硬件初始化。

}

static int really\_probe(struct device \*dev, struct device\_driver \*drv)

{

…

dev->driver = drv;

…

/\* 如果总线有probe则通过总线probe完成设备初始化，如果没有则直接调用驱动的probe初始化 \*/

if (dev->bus->probe) {

ret = dev->bus->probe(dev);

if (ret)

goto probe\_failed;

} else if (drv->probe) {

ret = drv->probe(dev);

if (ret)

goto probe\_failed;

}

…

}

### （3）注册stmmac驱动

在drivers/net/ethernet/stmmac/stmmac\_main.c中stmmac\_init\_module接口完成设备和驱动的注册，本小节讲解driver注册部分：

首先，定义一个platform\_driver结构体;

static struct platform\_driver stmmac\_driver = {

.probe = stmmac\_dvr\_probe, //platform驱动probe函数

.remove = stmmac\_dvr\_remove,

.driver = {

.name = STMMAC\_RESOURCE\_NAME, //“stmmaceth”需要和device注册的一致

.owner = THIS\_MODULE,

.pm = &stmmac\_pm\_ops,

},

};

static int \_\_init stmmac\_init\_module(void)

{

int ret;

…

ret = platform\_driver\_register(&stmmac\_driver); //注册platform驱动

…

return ret;

}

int platform\_driver\_register(struct platform\_driver \*drv)

{

drv->driver.bus = &platform\_bus\_type; //配置驱动总线为platform bus

if (drv->probe)

drv->driver.probe = platform\_drv\_probe; //设置driver驱动的probe，该接口会调用drv->probe，即stmmac\_dvr\_probe

if (drv->remove)

drv->driver.remove = platform\_drv\_remove;

if (drv->shutdown)

drv->driver.shutdown = platform\_drv\_shutdown;

return driver\_register(&drv->driver); //驱动注册

}

int driver\_register(struct device\_driver \*drv)

{

…

other = driver\_find(drv->name, drv->bus); //判断总线是否已经添加了该驱动

if (other) {

printk(KERN\_ERR "Error: Driver '%s' is already registered, "

"aborting...\n", drv->name);

return -EBUSY;

}

ret = bus\_add\_driver(drv); //向总线添加设备，并扫描是否有匹配的设备

if (ret)

return ret;

…

}

int bus\_add\_driver(struct device\_driver \*drv)

{

…

//驱动初始化相应的设备模型

klist\_init(&priv->klist\_devices, NULL, NULL);

priv->driver = drv;

drv->p = priv;

priv->kobj.kset = bus->p->drivers\_kset;

error = kobject\_init\_and\_add(&priv->kobj, &driver\_ktype, NULL,

"%s", drv->name);

if (error)

goto out\_unregister;

klist\_add\_tail(&priv->knode\_bus, &bus->p->klist\_drivers);

if (drv->bus->p->drivers\_autoprobe) {

error = driver\_attach(drv); //驱动匹配相应的设备

if (error)

goto out\_unregister;

}

…

}

int driver\_attach(struct device\_driver \*drv)

{

//扫描总线上的所有设备，查找该驱动匹配的所有设备

return bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach);

}

static int \_\_driver\_attach(struct device \*dev, void \*data)

{

struct device\_driver \*drv = data;

if (!driver\_match\_device(drv, dev)) //是否匹配

return 0;

if (dev->parent) /\* Needed for USB \*/

device\_lock(dev->parent);

device\_lock(dev);

/\*

如果匹配，调用相应的probe初始化设备，见上面一节分析。

really\_probe 调用dev->bus->probe（platform\_drv\_probe）函数调用stmmac\_dvr\_probe() 完成stmmac网卡驱动注册，实现初始化。

\*/

if (!dev->driver)

driver\_probe\_device(drv, dev);

device\_unlock(dev);

if (dev->parent)

device\_unlock(dev->parent);

return 0;

}

# 五、字符设备

字符设备是面向流的设备，一个字节一个字节读写设备，按照先后顺序进行，不能随机读取设备内存中的某一个数据。所以字符设备处理起来相应容易些，它不需要复杂的缓冲策略，也不涉及磁盘高速缓存。

## 1、数据结构

### （1）cdev

字符设备由cdev结构体定义（include/linux/cdev.h）：

struct cdev {

struct kobject kobj; //内嵌kobject

struct module \*owner; //指向实现驱动程序模块的指针

const struct file\_operations \*ops; //指向驱动程序文件操作列表

struct list\_head list; //与字符设备文件对应的索引节点链表头

dev\_t dev; //给设备驱动程序所分配的初始主设备号和次设备号

unsigned int count; //给设备驱动程序所分配的设备号范围的大小

};

list字段是双向循环链表的首部，该链表用于收集相同字符设备驱动程序所对应的字符设备文件的索引节点。

### （2）char\_device\_struct

为了记录目前已经分配了哪些字符设备号，内核使用散列表chrdevs来表示，由结构体char\_device\_struct定义（fs/char\_dev.c）

static struct char\_device\_struct {

struct char\_device\_struct \*next; //指向同主设备号的下一个元素

unsigned int major; //主设备号

unsigned int baseminor; //设备号范围的初始次设备号

int minorct; //次设备号范围大小(一般1个)，范围：baseminor ~ baseminor + minorct

char name[64]; //字符设备驱动程序名字

struct cdev \*cdev; /\* will die \*/ //指向字符设备cdev

} \*chrdevs[CHRDEV\_MAJOR\_HASH\_SIZE]; //每一项代表一个主设备号

#define CHRDEV\_MAJOR\_HASH\_SIZE 255

表示最多支持255个主设备号散列表，而实际没有这么多可以使用，当主设备号超过255时，通过major\_to\_index接口转化到对应的散列表。两个不同的设备可能共享同一个主设备号，但是次设备号必然不同。

### （3）kobj\_map

为了管理已经注册了的字符设备，内核使用散列表probe来管理，总共255个表项，由上面的0~255主设备号进行索引，每个probe类型对象都拥有一个已经注册的主设备号和次设备号。其定义如下（drivers/base/map.c）：

struct kobj\_map {

struct probe {

struct probe \*next; //散列表中的下一个元素

dev\_t dev; //初始设备号（主次设备号）

unsigned long range; //设备号范围的大小，范围：dev~dev+range

struct module \*owner;

kobj\_probe\_t \*get; //探测谁拥有这个设备号范围

int (\*lock)(dev\_t, void \*); //增加设备号范围内拥有者的引用计数器

void \*data; //设备好范围内拥有者的私有数据，指向字符设备cdev

} \*probes[255];

struct mutex \*lock; //访问加锁

};

## 2、 操作接口

### （1）register\_chrdev

字符设备驱动程序通过register\_chrdev（\_\_register\_chrdev）接口完成注册，其实现如下：

int \_\_register\_chrdev(unsigned int major, unsigned int baseminor,

unsigned int count, const char \*name,

const struct file\_operations \*fops)

{

…

/\* 申请可用的设备号 \*/

cd = \_\_register\_chrdev\_region(major, baseminor, count, name);

if (IS\_ERR(cd))

return PTR\_ERR(cd);

/\* 申请字符设备，并完成kobject设备模型初始化 \*/

cdev = cdev\_alloc();

if (!cdev)

goto out2;

…

/\* 注册字符设备 \*/

err = cdev\_add(cdev, MKDEV(cd->major, baseminor), count);

if (err)

goto out;

cd->cdev = cdev;

…

}

### （2）\_\_register\_chrdev\_region

如果major=0，则动态申请，从空闲处申请相应minor的设备号；如果非0，则从对应major处申请相应的minor设备号，已经被使用则返回错误。整个散列表的管理结构如图所示：



baseminor次设备号管理按照递增的方式进行。

### （3）cdev\_add

添加一个字符设备到内核，其实就添加到上面的kobj\_map散列表里。整个散列表的管理结构如图所示：



同一个probe下dev的major可能不同，通过major%255得到probe的index。

# 六、块设备

系统中能够随机访问固定大小数据片的硬件设备称为块设备，最常见的块设备硬盘，此外还有光盘、闪存等。它们一般都已安装文件系统的方式使用，这也是块设备一般的访问方式。内核管理块设备比字符设备要细致得多，需要考虑的问题和完成的工作相对字符设备来说要复杂许多。涉及很多内核模块如图所示：



例如当进程发起一个read系统调用时，内核发生的基本步骤：

A．调用一个合适的VFS函数，将文件描述符和文件内的偏移量传递给它

B．VFS函数会优先从磁盘缓存中查询是否存在，不存在再访问磁盘

C．从磁盘读取数据就必须确定数据的具体物理位置，内核映射层（具体文件系统）主要完成：

确定文件所在文件系统的块大小，并根据文件块的大小计算所请求数据的长度。

调用一个具体文件系统的函数，访问文件的磁盘节点，然后根据逻辑块号确定所请求数据在磁盘上的位置。

D．利用通用块层启动I/O操作来传递所请求的数据

E．I/O调度程序根据预先定义的内核策略将待处理的I/O数据传送请求进行归类，主要作用是把物理介质上相邻的数据请求聚集在一起。

F．最后块设备驱动程序向磁盘控制器的硬件接口发送适当命令，进行实际的数据传送。

## 1、块设备处理

块设备中的数据存储涉及了许多内核模块，每个模块采用不同长度的块来管理磁盘数据，如图所示：



### （1）扇区

硬件块设备控制器采用称为“扇区”的固定长度的块来传送数据。因此，I/O调度程序和块设备驱动程序必须管理数据扇区。

在大部分磁盘设备中，扇区的大小是512B，但是一些设备使用更大的扇区（1024和2048字节），扇区是数传送的基本单元，不允许传送少于一个扇区数据，但可以同时传送多个相邻的扇区。

Linux中扇区大小按惯例都设为512B，如果一个块设备使用更大的扇区，就需要底层块驱动程序进行相应的变换。

### （2）块

虚拟文件系统、映射层和文件系统将磁盘数据存放在称为“块”的逻辑单元中。一个块对应文件系统中一个最小的磁盘存储单元。

Linux中块大小必须是2的幂，是扇区大小的整数倍，而且不能超过一个页框。块设备的块大小不是唯一的，创建磁盘文件系统时可以选择合适的块大小。每个块都有自己的块缓冲区（如上面图中的磁盘高速缓冲区），当内核从磁盘读出一个块时将硬件设备中所获得的值来填充相应的块缓冲区，当内核向磁盘中写入一个块时用相关块缓冲区的实际值更新硬件设备上相应的一组相邻字节。

### （3）段

块设备驱动程序能够处理数据的“段”：一个段就是一个内存页或内存页的一部分，它们包含磁盘上物理相邻的数据块。

对磁盘的每个I/O操作就是磁盘与一些RAM单元之间互相传送一些相邻扇区的内容，大多数情况下通过DMA方式进行。块设备驱动程序只要向磁盘控制器发送一些适当的命令就可以触发一次数据传送；一旦完成数据的传送，控制器就会发出一个中断通知块设备驱动程序。

新的分散-聚集DMA传送方式使磁盘可以与一些非连续的内存区相互传送数据，而块设备驱动程序必须能够处理称为段的数据存储单元，一次分散-聚集DMA操作可能同时传送几个段。

### （4）页

磁盘高速缓存作用于磁盘数据的“页”上，每页正好装一个页框中。

## 2、通用块层

通用块层主要处理来自系统中的所有块设备发出的请求。以前的核心数据结构是buffer\_head，现在则是bio。

### （1）buffer\_head结构体

按照前面说的每个块都有一个与之对应的缓冲区，相当于磁盘块在内存中的表示。内核通过buffer\_head数据结构（缓冲区头）来描述块对应的缓冲区，包含了内核操作缓冲区所需要的全部信息。详细定义如下（include/linux/buffer\_head.h）：

struct buffer\_head {

unsigned long b\_state; /\* 缓冲区状态标志，位图形式 \*/

struct buffer\_head \*b\_this\_page;/\* 页面中的缓冲区 \*/

struct page \*b\_page; /\* 存储缓冲区中的页面 \*/

sector\_t b\_blocknr; /\* 起始块号 \*/

size\_t b\_size; /\* 映射的大小 \*/

char \*b\_data; /\* 页面内的数据指针 \*/

struct block\_device \*b\_bdev; /\* 相关联的块设备 \*/

bh\_end\_io\_t \*b\_end\_io; /\* I/O完成方法 \*/

void \*b\_private; /\* reserved for b\_end\_io \*/

struct list\_head b\_assoc\_buffers; /\* 相关的映射链表 \*/

struct address\_space \*b\_assoc\_map; /\* 相关的地址空间 \*/

atomic\_t b\_count; /\* 缓冲区使用计数 \*/

};

缓冲区头不仅描述了从磁盘块到物理内存的映射，而且还是所有块I/O操作的容器。作为容器，存在两个明显的弊端：

第一，缓冲区头是一个很大且不易控制的数据结构体，而且对数据操作不方面和不清晰。对内核来说更倾向操作页面结构，因为页操作简便，同时效率更高。

第二，作为I/O容器使用时，缓冲区头会促使内核把大块数据的I/O操作分解为多个buffer\_head结构体进行操作。

因此，内核引入了一种新型、灵活并且轻量级的容器—bio结构体。

### （2）bio结构体

目前，内核中块I/O操作的基本容器由bio结构体表示，是通用层的核心数据结构。该结构体代表了正在现场执行的以片段链表形式组织的块I/O操作，一个片段是一小块连续的内存缓冲区，这样就不需要保证单个缓冲区一定要连续。通过片段来描述缓冲区，即使一个缓冲区分散在内存多个位置上，bio结构体也能对内存保证I/O操作的执行。该结构体详细定义如下（include/linux/blk\_types.h）：

struct bio {

sector\_t bi\_sector; /\* 第一个磁盘扇区地址 \*/

struct bio \*bi\_next; /\* 链接到请求队列中的下一个bio \*/

struct block\_device \*bi\_bdev; /\* 指向块设备描述符的指针 \*/

unsigned long bi\_flags; /\* bio的状态标志 \*/

unsigned long bi\_rw; /\* 低位读写标志，高位代表优先级 \*/

unsigned short bi\_vcnt; /\* bio\_vec数组中段的数目 \*/

unsigned short bi\_idx; /\* bio\_vec数组中段的当前索引值 \*/

unsigned int bi\_phys\_segments; /\* 合并自后bio中物理段的数目 \*/

unsigned int bi\_size; /\* 剩余需要传送的字节数目 \*/

unsigned int bi\_seg\_front\_size; //第一个可合并的段大小

unsigned int bi\_seg\_back\_size; //最后一个可合并的段大小

bio\_end\_io\_t \*bi\_end\_io; //bio的I/O操作结束时调用的方法

void \*bi\_private; //拥有者的私有方法

unsigned int bi\_max\_vecs; /\* bio的bio\_vec数组中允许的最大段数 \*/

atomic\_t bi\_cnt; /\* bio引用计数器 \*/

struct bio\_vec \*bi\_io\_vec; /\* 指向bio的bio\_vec数组中的段的指针 \*/

struct bio\_set \*bi\_pool;

/\*

\* We can inline a number of vecs at the end of the bio, to avoid

\* double allocations for a small number of bio\_vecs. This member

\* MUST obviously be kept at the very end of the bio.

\*/

struct bio\_vec bi\_inline\_vecs[0]; /\* 内嵌bio向量 \*/

};

bio中的每个段是由一个bio\_vec数据结构描述的，包含了一个特定I/O操作需要使用到的所有片段，结构体详细定义如下：

struct bio\_vec {

struct page \*bv\_page; //指向片段的页框中页描述符的指针

unsigned int bv\_len; //段的字节长度

unsigned int bv\_offset; //页框中段数据的偏移量

};

每个bio\_vec是一个形式为<page, offset, len>的向量，它描述的是一个特定片段：片段所在物理页，块在物理页中的偏移位置，从给定偏移量开始的块长度。在每个给定的块I/O操作中，bi\_vcnt用来描述bio\_vec数组中的向量数目，bi\_idx指向数组中的当前bio\_vec片段，可以通过bi\_idx跟踪I/O操作完成进度。它们的关系如图所示：



buffer\_head和bio结构体之间存在显著差异：bio结构体代表的是I/O操作，它可以包括内存中的一个或多个页；buffer\_head结构体代表的是缓冲区，描述的仅仅是磁盘中的一个块。在当前设置中，当bio结构体描述当前正在使用的I/O操作时，buffer\_head结构体仍然需要包含缓冲区信息，内核通过这两种结构分别保存各自的信息。

### （3）请求队列

当向通用层提交一个I/O操作请求时，内核首先调用bio\_alloc函数分配一个新的bio描述符并进行初始化，接着调用generic\_make\_request函数（通用块层的主要入口点）将bio请求插入到与块设备相关的请求队列q中，然后I/O调度程序启动使块设备驱动程序从队列中获取请求并将其送入对应的块设备上去。

## 3、I/O调度程序

如果简单以通用块层产生I/O请求的次序直接将请求发送给块设备，性能将急剧下降。为了优化寻址操作，内核即不会简单的按请求次序，也不会立即将其提交给磁盘。相反，它会在提交前，先执行名为合并与排序的预操作，这种方式几大提高系统的整体性能。在内核中负责提交I/O请求的子系统称为I/O调度程序。

为了防止块设备驱动程序被挂起，每个I/O操作都是异步处理的。特别是块设备驱动程序是中断驱动的：通用块层调用I/O调度程序产生一个新的块设备请求或扩展一个已有的块设备请求，然后终止。在恰当的时间，块设备驱动程序会调用一个所谓的策略例程选择一个待处理的请求，发送给块设备。当块设备操作完成时会产生一个中断，如果需要，中断处理程序就会调用策略例程去处理队列中的另一个请求。

每个块设备驱动程序都维持着自己的请求队列，通过结构体request\_queue（struct request\_queue）表示，它是一个双向链表，其元素就是请求描述符request（include/linux/blkdev.h）。

每个块设备的待处理请求都是一个请求描述符来表示，每个请求包含一个或多个bio结构：通用块层创建一个仅包含一个bio结构的请求，然后I/O调度程序向初始的bio增加一个新段，要么将另外一个bio结构连接到请求中，从而“扩展”该请求。

当向请求队列增加一条新的请求时，通用块层会调用I/O调度程序来确定请求队列中新请求的确切位置。I/O调度程序也被称为电梯调度，衍生4种不同算法：预测算法、最后期限算法、完全公平队列算法、Noop算法。所有算法都使用一个调度队列（电梯算法），队列中包含的所有请求按照设备驱动程序应当处理的顺序进行排序。

Noop算法：最简单的I/O调度算法，没有排序队列：新的请求通常被插在调度队列的开头或末尾，下一个要处理的请求总是队列中的第一个请求。

CFQ算法：主要目标是在触发I/O请求的所有进程中确保磁盘I/O宽带的公平分配。算法使用许多个排序队列（默认64个）存放不同进程发出的请求。当算法处理一个请求时，内核将线程组标识符PID转化为队列的索引值；然后算法将一个新的请求插入该队列的末尾。

deadline算法：除了调度队列外，还使用了4个队列，其中两个排序队列分别包含读请求和写请求，另外连个包含相同读写请求但经过“最后期限”排序过的。引入这些队列是为了避免请求饿死，由于电梯策略优先处理与上一个所处理的请求最近的请求，因而就会对某个请求忽略很长一段时间。该算法本质就是一个超时定时器，当请求被传给电梯算法时开始计时。

预测算法：linux提供的最复杂的一种I/O调度算法，是deadline算法的一个演变：实现了三个队列，并为每个请求设置超时时间，主要改进是增加了预测启发能力。

内核使用的缺省电梯算法可在引导时通过内核参数elevator=<name>进行再设置。

## 4、驱动程序

块设备驱动程序是linux块子系统中的最底层组件。从I/O调度程序中获得请求，然后按要求处理这些请求。

块设备驱动程序是设备驱动程序模型的组成部分，因此每个块设备驱动程序对应一个device\_driver类型的描述符，块设备驱动程序的每个设备都有一个device描述符相关联。这些只是驱动模型的一部分，块I/O子系统通过block\_device结构体描述了每个块设备存放的附加信息。

### （1）数据结构block\_device

每个块设备都由一个block\_device结构的描述符来表示，其具体定义如下（include/linux/fs.h）：

struct block\_device {

dev\_t bd\_dev; //块设备的主设备号和此设备号

int bd\_openers; //计数器，统计块设备已经被打开多少次

struct inode \* bd\_inode; //指向bdev文件系统中块设备对应的文件索引节点指针

struct super\_block \* bd\_super; //指向bdev文件系统中块设备对应的超级块

struct mutex bd\_mutex; //打开关闭互斥锁

struct list\_head bd\_inodes; //节点链表

void \* bd\_claiming;

void \* bd\_holder; //块设备描述符的当前所有者

int bd\_holders; //对bd\_holder字段多次设置的次数

bool bd\_write\_holder;

#ifdef CONFIG\_SYSFS

struct list\_head bd\_holder\_disks;

#endif

struct block\_device \* bd\_contains; //如果块设备是一个分区，则指向整个磁盘的块设备描述符；否则指向该块设备描述符

unsigned bd\_block\_size; //块大小

struct hd\_struct \* bd\_part; //指向分区描述符的指针，如果该块设备不是一个分区则为NULL

unsigned bd\_part\_count; //计数器，统计包含在块设备中的分区已经被打开了多少次

int bd\_invalidated; //当需要读块设备的分区表时设置的标志

struct gendisk \* bd\_disk; //指向块设备中磁盘的gendisk结构指针

struct request\_queue \* bd\_queue; //块设备请求队列

struct list\_head bd\_list; //块设备描述符链表的指针

unsigned long bd\_private; //指向块设备持有者的私有数据指针

/\* The counter of freeze processes \*/

int bd\_fsfreeze\_count;

/\* Mutex for freeze \*/

struct mutex bd\_fsfreeze\_mutex;

};

所有的块设备描述符被插入一个全局链表中，链表首部由变量all\_bdevs表示。

### （2）数据结构blk\_major\_name

为了记录目前已经分配了哪些块设备的主设备号，内核使用散列表major\_names来表示，由结构体blk\_major\_name定义（block/genhd.c）

block/genhd.c

static struct blk\_major\_name {

struct blk\_major\_name \*next;

int major;

char name[16];

} \*major\_names[BLKDEV\_MAJOR\_HASH\_SIZE];

#define BLKDEV\_MAJOR\_HASH\_SIZE 255

表示最多支持255个主设备号散列表，当主设备号超过255时，通过major\_to\_index接口转化到对应的散列表，次设备号则通过分区来表示。

### （3）数据结构kobj\_map

同字符设备一样，见上节

### （4）接口register\_blkdev

如果major=0，则动态申请，从空闲处申请相应主设备号；如果非0，则先判断相应major链表是否存在，如果已经被使用则返回错误，否则添加。整个散列表的管理结构如图所示：



### （5）接口add\_disk

添加块设备的分区信息到内核列表。调用blk\_register\_region将块设备添加到上面kobj\_map散列表里，整个散列表的管理结构如图所示：



该接口还会调用register\_disk—> device\_add向内核注册一个块设备。

本文从总体框架上描述了linux设备驱动情况，每个部分都需要深入研究才能真正掌握，待后续慢慢补充。

# 附录A

参考网址：

<http://www.wowotech.net/sort/device_model/page/1>

<https://blog.csdn.net/andylauren/article/details/51803331>