

第5章

Linux 文件系统与设备文件

本章导读

由于字符设备和块设备都良好地体现了“一切都是文件”的设计思想，掌握 Linux 文件系统、设备文件系统的知识就显得相当重要了。

首先，驱动最终通过与文件操作相关的系统调用或 C 库函数（本质也基于系统调用）被访问，而设备驱动的结构最终也是为了迎合提供给应用程序员的 API。

其次，驱动工程师在设备驱动中不可避免地会与设备文件系统打交道，包括从 Linux 2.4 内核的 devfs 文件系统到 Linux 2.6 以后的 udev。

5.1 节讲解了通过 Linux API 和 C 库函数在用户空间进行 Linux 文件操作的编程方法。

5.2 节分析了 Linux 文件系统的目录结构，简单介绍了 Linux 内核中文件系统的实现，并给出了文件系统与设备驱动的关系。

5.3 节和 5.4 节分别讲解 Linux 2.4 内核的 devfs 和 Linux 2.6 以后的内核所采用的 udev 设备文件系统，并分析了两者的区别。5.4 节还重点介绍了 Linux 的设备驱动模型、sysfs 以及 udev 的规则编写。

5.1 Linux 文件操作

5.1.1 文件操作系统调用

Linux 的文件操作系统调用（在 Windows 编程领域，习惯称操作系统提供的接口为 API）涉及创建、打开、读写和关闭文件。

1. 创建

```
int creat(const char *filename, mode_t mode);
```

参数 mode 指定新建文件的存取权限，它同 umask 一起决定文件的最终权限（mode&umask），其中，umask 代表了文件在创建时需要去掉的一些存取权限。umask 可通过系统调用 umask() 来改变：

```
int umask(int newmask);
```

该调用将 umask 设置为 newmask，然后返回旧的 umask，它只影响读、写和执行权限。

2. 打开

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

open() 函数有两个形式，其中 pathname 是我们要打开的文件名（包含路径名称，缺省是认为在当前路径下面），flags 可以是表 5.1 中的一个值或者是几个值的组合。

表 5.1 文件打开标志

标 志	含 义
O_RDONLY	以只读的方式打开文件
O_WRONLY	以只写的方式打开文件
O_RDWR	以读写的方式打开文件
O_APPEND	以追加的方式打开文件
O_CREAT	创建一个文件
O_EXEC	如果使用了 O_CREAT 而且文件已经存在，就会发生一个错误
O_NOBLOCK	以非阻塞的方式打开一个文件
O_TRUNC	如果文件已经存在，则删除文件的内容

O_RDONLY、O_WRONLY、O_RDWR 三个标志只能使用任意的一个。

如果使用了 O_CREATE 标志，则使用的函数是 int open(const char *pathname,int flags,mode_t mode)；这个时候我们还要指定 mode 标志，以表示文件的访问权限。mode 可以是表 5.2 中所列值的组合。

表 5.2 文件访问权限

标 志	含 义
S_IRUSR	用户可以读
S_IWUSR	用户可以写
S_IXUSR	用户可以执行
S_IRWXU	用户可以读、写、执行
S_IRGRP	组可以读
S_IWGRP	组可以写
S_IXGRP	组可以执行
S_IRWXG	组可以读、写、执行
S_IROTH	其他人可以读
S_IWOTH	其他人可以写
S_IXOTH	其他人可以执行
S_IRWXO	其他人可以读、写、执行
S_ISUID	设置用户执行 ID
S_ISGID	设置组的执行 ID

除了可以通过上述宏进行“或”逻辑产生标志以外，我们也可以自己用数字来表示，Linux 用 5 个数字来表示文件的各种权限：第一位表示设置用户 ID；第二位表示设置组 ID；第三位表示用户自己的权限位；第四位表示组的权限；最后一位表示其他人的权限。每个数字可以取 1（执行权限）、2（写权限）、4（读权限）、0（无）或者是这些值的和。例如，要创建一个用户可读、可写、可执行，但是组没有权限，其他人可以读、可以执行的文件，并设置用户 ID 位，那么应该使用的模式是 1（设置用户 ID）、0（不设置组 ID）、7（1+2+4，读、写、执行）、0（没有权限）、5（1+4，读、执行）即 10 705：

```
open("test", O_CREAT, 10 705);
```

上述语句等价于：

```
open("test", O_CREAT, S_IRWXU | S_IROTH | S_IXOTH | S_ISUID );
```

如果文件打开成功，open 函数会返回一个文件描述符，以后对该文件的所有操作就可以通过对这个文件描述符进行操作来实现。

3. 读写

在文件打开以后，我们才可对文件进行读写，Linux 中提供文件读写的系统调用是 read、write 函数：

```
int read(int fd, const void *buf, size_t length);
int write(int fd, const void *buf, size_t length);
```

其中，参数 buf 为指向缓冲区的指针，length 为缓冲区的大小（以字节为单位）。函数 read() 实现从文件描述符 fd 所指定的文件中读取 length 个字节到 buf 所指向的缓冲区中，返回值为实际读取的字节数。函数 write 实现把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中，返回值为实际写入的字节数。

以 O_CREAT 为标志的 open 实际上实现了文件创建的功能，因此，下面的函数等同于 creat() 函数：

```
int open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

4. 定位

对于随机文件，我们可以随机指定位置进行读写，使用如下函数进行定位：

```
int lseek(int fd, offset_t offset, int whence);
```

lseek() 将文件读写指针相对 whence 移动 offset 个字节。操作成功时，返回文件指针相对于文件头的位置。参数 whence 可使用下述值：

SEEK_SET：相对文件开头

SEEK_CUR：相对文件读写指针的当前位置

SEEK_END：相对文件末尾

`offset` 可取负值，例如下述调用可将文件指针相对当前位置向前移动 5 个字节：

```
lseek(fd, -5, SEEK_CUR);
```

由于 `lseek` 函数的返回值为文件指针相对于文件头的位置，因此下列调用的返回值就是文件的长度：

```
lseek(fd, 0, SEEK_END);
```

5. 关闭

当我们操作完成以后，要关闭文件，此时，只要调用 `close` 就可以了，其中 `fd` 是我们要关闭的文件描述符：

```
int close(int fd);
```

例程：编写一个程序，在当前目录下创建用户可读写文件 `hello.txt`，在其中写入“Hello, software weekly”，关闭该文件。再次打开该文件，读取其中的内容并输出在屏幕上。

解答如代码清单 5.1。

代码清单 5.1 Linux 文件操作用户空间编程（使用系统调用）

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <stdio.h>
5 #define LENGTH 100
6 main()
7 {
8     int fd, len;
9     char str[LENGTH];
10
11    fd = open("hello.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); /* 
12    创建并打开文件 */
13    if (fd) {
14        write(fd, "Hello World", strlen("Hello World")); /* 
15        写入字符串 */
16        close(fd);
17    }
18
19    fd = open("hello.txt", O_RDONLY);
20    len = read(fd, str, LENGTH); /* 读取文件内容 */
21    str[len] = '\0';
22    printf("%s\n", str);
23    close(fd);
24 }
```

编译并运行，执行结果为输出“Hello World”。

5.1.2 C 库文件操作

C 库函数的文件操作实际上独立于具体的操作系统平台，不管是在 DOS、Windows、Linux 还是在 VxWorks 中都是这些函数：

1. 创建和打开

```
FILE *fopen(const char *path, const char *mode);
```

fopen() 用于打开指定文件 filename，其中的 mode 为打开模式，C 库函数中支持的打开模式如表 5.3 所示。

表 5.3 C 库函数文件打开标志

标 志	含 义
r、rb	以只读方式打开
w、wb	以只写方式打开。如果文件不存在，则创建该文件，否则文件被截断
a、ab	以追加方式打开。如果文件不存在，则创建该文件
r+、r+b、rb+	以读写方式打开
w+、w+b、wh+	以读写方式打开。如果文件不存在时，创建新文件，否则文件被截断
a+、a+b、ab+	以读和追加方式打开。如果文件不存在，则创建新文件

其中，b 用于区分二进制文件和文本文件，这一点在 DOS、Windows 系统中是有区分的，但 Linux 不区分二进制文件和文本文件。

2. 读写

C 库函数支持以字符、字符串等为单位，支持按照某种格式进行文件的读写，这一组函数为：

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf (FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t n, FILE *stream);
```

fread() 实现从流 (stream) 中读取 n 个字段，每个字段为 size 字节，并将读取的字段放入 ptr 所指的字符数组中，返回实际已读取的字段数。当读取的字段数小于 num 时，可能是在函数调用时出现了错误，也可能是读到了文件的结尾。因此要通过调用 feof() 和 perror() 来判断。

write() 实现从缓冲区 ptr 所指的数组中把 n 个字段写到流 (stream) 中，每个字段长为 size 个字节，返回实际写入的字段数。

另外，C 库函数还提供了读写过程中的定位能力，这些函数包括：

```

int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
int fseek(FILE *stream, long offset, int whence);

```

3. 关闭

利用 C 库函数关闭文件依然很简单：

```
int fclose (FILE *stream);
```

例程：将第 5.1.1 节中的例程用 C 库函数来实现，如代码清单 5-2 所示。

代码清单 5.2 Linux 文件操作用户空间编程（使用 C 库函数）

```

1 #include <stdio.h>
2 #define LENGTH 100
3 main()
4 {
5     FILE *fd;
6     char str[LENGTH];
7
8     fd = fopen("hello.txt", "w+");      /* 创建并打开文件 */
9     if (fd) {
10         fputs("Hello World", fd);      /* 写入字符串 */
11         fclose(fd);
12     }
13
14     fd = fopen("hello.txt", "r");      /* 读取文件内容 */
15     fgets(str, LENGTH, fd);
16     printf("%s\n", str);
17     fclose(fd);
18 }
```

5.2 Linux 文件系统

5.2.1 Linux 文件系统目录结构

进入 Linux 根目录（即“/”，Linux 文件系统的入口，也是处于最高一级的目录），运行“ls -l”命令，看到 Linux 包含以下目录。

1. /bin

包含基本命令，如 ls、cp、mkdir 等，这个目录中的文件都是可执行的。

2. /sbin

包含系统命令，如 modprobe、hwclock、ifconfig 等，大多是涉及系统管理的命令，这个目录中的文件都是可执行的。

3. /dev

设备文件存储目录，应用程序通过对这些文件的读写和控制以访问实际的设备。

4. /etc

系统配置文件的所在地，一些服务器的配置文件也在这里，如用户账号及密码配置文件。`busybox` 的启动脚本也存放在该目录。

5. /lib

系统库文件存放目录等。

6. /mnt

`/mnt` 这个目录一般是用于存放挂载储存设备的挂载目录，比如含有 `cdrom` 等目录。可以参看 `/etc/fstab` 的定义。有时我们可以让系统开机自动挂载文件系统，并把挂载点放在这里。

7. /opt

`opt` 是“可选”的意思，有些软件包会被安装在这里。

8. /proc

操作系统运行时，进程及内核信息（比如 CPU、硬盘分区、内存信息等）存放在那里。`/proc` 目录为伪文件系统 `proc` 的挂载目录，`proc` 并不是真正的文件系统，它存在于内存之中。

9. /tmp

用户运行程序的时候，有时会产生临时文件，`/tmp` 用来存放临时文件。

10. /usr

这个是系统存放程序的目录，比如用户命令、用户库等。

11. /var

`var` 表示的是变化的意思，这个目录的内容经常变动，如 `/var` 的 `/var/log` 目录被用来存放系统日志。

12. /sys

Linux 2.6 以后的内核所支持的 `sysfs` 文件系统被映射在此目录上。Linux 设备驱动模型中的总线、驱动和设备都可以在 `sysfs` 文件系统中找到对应的节点。当内核检测到在系统中出现了新设备后，内核会在 `sysfs` 文件系统中为该新设备生成一项新的记录。

5.2.2 Linux 文件系统与设备驱动

图 5.1 所示为 Linux 中虚拟文件系统、磁盘/Flash 文件系统及一般的设备文件与设备驱动程序之间的关系。

应用程序和 VFS 之间的接口是系统调用，而 VFS 与文件系统以及设备文件之间的接口是 `file_operations` 结构体成员函数，这个结构体包含对文件进行打开、关闭、读写、控制的一系列成员函数，关系如图 5.2 所示。

由于字符设备的上层没有类似于磁盘的 ext2 等文件系统，所以字符设备的 `file_operations` 成员函数就直接由设备驱动提供了，在稍后的第 6 章，将会看到 `file_operations` 正是字符设备驱动的核心。块设备有两种访问方法，一种方法是不通过文件系统直接访问裸设备，在 Linux 内核实现了统一的 `def_blk_fops` 这一 `file_operations`，它的源代码位于 `fs/block_`

dev.c，所以当我们运行类似于“`dd if=/dev/sdb1 of=sdb1.img`”的命令把整个`/dev/sdb1`裸分区复制到`sdb1.img`的时候，内核走的是`def_blk_fops`这个`file_operations`；另外一种方法是通过文件系统来访问块设备，`file_operations`的实现则位于文件系统内，文件系统会把针对文件的读写转换为针对块设备原始扇区的读写。`ext2`、`fat`、`Btrfs`等文件系统中会实现针对VFS的`file_operations`成员函数，设备驱动层将看不到`file_operations`的存在。

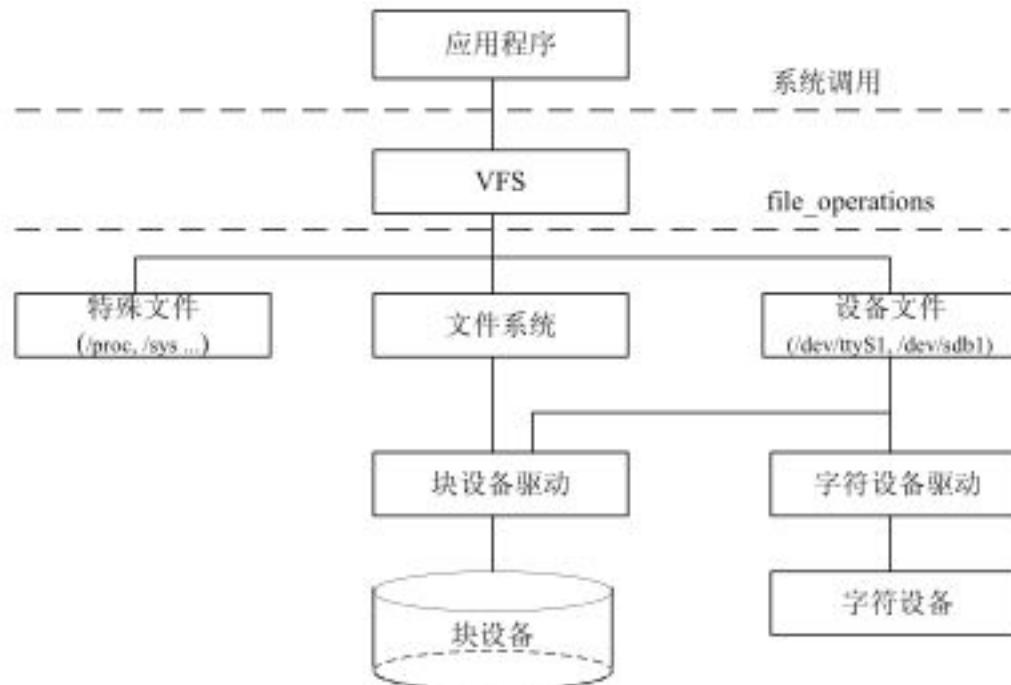


图 5.1 文件系统与设备驱动之间的关系

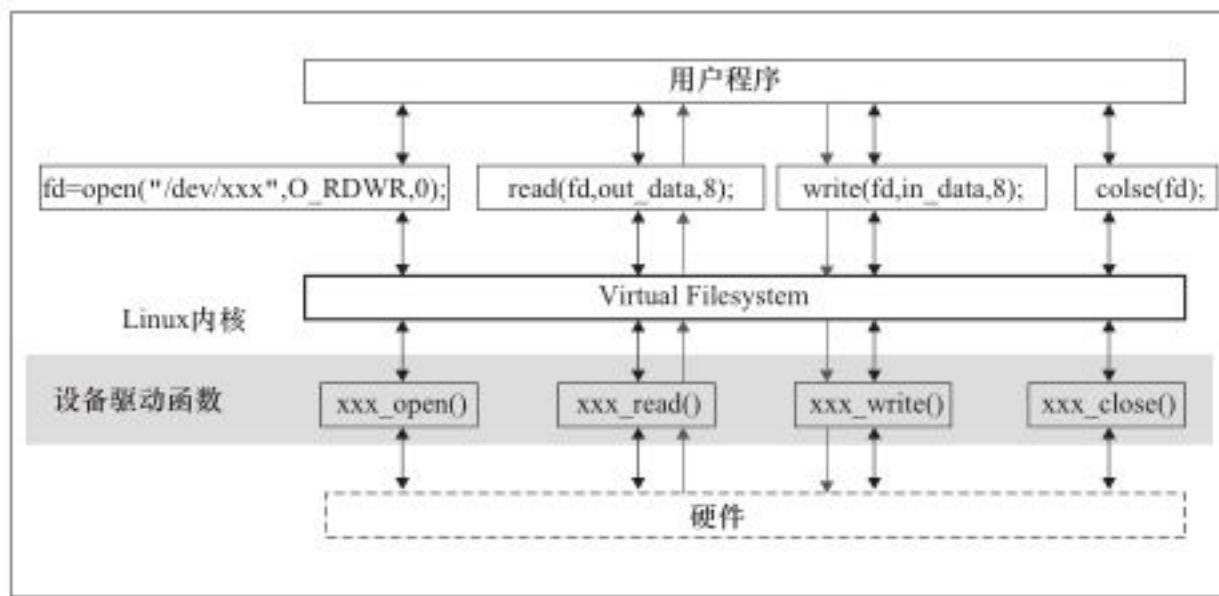


图 5.2 应用程序、VFS 与设备驱动

在设备驱动程序的设计中，一般而言，会关心`file`和`inode`这两个结构体。

1. file 结构体

`file`结构体代表一个打开的文件，系统中每个打开的文件在内核空间都有一个关联的

struct file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动源代码中，struct file 的指针通常被命名为 file 或 filp (即 file pointer)。代码清单 5.3 给出了文件结构体的定义。

代码清单 5.3 文件结构体

```

1 struct file {
2     union {
3         struct llist_node      fu_llist;
4         struct rcu_head        fu_rcuhead;
5     } f_u;
6     struct path             f_path;
7     #define f_dentry          f_path.dentry
8     struct inode            *f_inode;           /* cached value */
9     const struct file_operations*f_op;          /* 和文件关联的操作 */
10
11    /*
12     * Protects f_ep_links, f_flags,
13     * Must not be taken from IRQ context.
14     */
15    spinlock_t              f_lock;
16    atomic_long_t            f_count;
17    unsigned int              f_flags;           /* 文件标志，如 O_RDONLY、O_NONBLOCK、O_SYNC */
18    fmode_t                  f_mode;            /* 文件读 / 写模式，FMODE_READ 和 FMODE_WRITE */
19    struct mutex              f_pos_lock;
20    loff_t                   f_pos;              /* 当前读写位置 */
21    struct fown_struct       f_owner;
22    const struct cred        *f_cred;
23    struct file_ra_state f_ra;
24
25    u64                      f_version;
26    #ifdef CONFIG_SECURITY
27    void                     *f_security;
28    #endif
29    /* needed for tty driver, and maybe others */
30    void                     *private_data;        /* 文件私有数据 */
31
32    #ifdef CONFIG_EPOLL
33    /* Used by fs/eventpoll.c to link all the hooks to this file */
34    struct list_head          f_ep_links;
35    struct list_head          f_tfile_llink;
36    #endif                         /* #ifdef CONFIG_EPOLL */
37    struct address_space *f_mapping;
38 } __attribute__((aligned(4)));           /* lest something weird decides that 2 is OK */

```

文件读 / 写模式 mode、标志 f_flags 都是设备驱动关心的内容，而私有数据指针 private_data 在设备驱动中被广泛应用，大多被指向设备驱动自定义以用于描述设备的结构体。

下面的代码可用于判断以阻塞还是非阻塞方式打开设备文件：

```

if (file->f_flags & O_NONBLOCK)           /* 非阻塞 */
    pr_debug("open: non-blocking\n");
else                                     /* 阻塞 */
    pr_debug("open: blocking\n");

```

2. inode 结构体

VFS inode 包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是 Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁，inode 结构体的定义如代码清单 5.4 所示。

代码清单 5.4 inode 结构体

```

1 struct inode {
2     ...
3     umode_t i_mode;          /* inode 的权限 */
4     uid_t i_uid;             /* inode 拥有者的 id */
5     gid_t i_gid;             /* inode 所属的群组 id */
6     dev_t i_rdev;            /* 若是设备文件，此字段将记录设备的设备号 */
7     loff_t i_size;           /* inode 所代表的文件大小 */
8
9     struct timespec i_atime; /* inode 最近一次的存取时间 */
10    struct timespec i_mtime; /* inode 最近一次的修改时间 */
11    struct timespec i_ctime; /* inode 的产生时间 */
12
13    unsigned int      i_blkbits;
14    blkcnt_t         i_blocks; /* inode 所使用的 block 数，一个 block 为 512 字节 */
15    union {
16        struct pipe_inode_info *i_pipe;
17        struct block_device *i_bdev;
18                    /* 若是块设备，为其对应的 block_device 结构体指针 */
19        struct cdev *i_cdev;   /* 若是字符设备，为其对应的 cdev 结构体指针 */
20    }
21    ...
22 };

```

对于表示设备文件的 inode 结构，i_rdev 字段包含设备编号。Linux 内核设备编号分为主设备编号和次设备编号，前者为 dev_t 的高 12 位，后者为 dev_t 的低 20 位。下列操作用于从一个 inode 中获得主设备号和次设备号：

```

unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);

```

查看 /proc/devices 文件可以获知系统中注册的设备，第 1 列为主设备号，第 2 列为设备名，如：

```

Character devices:
  1 mem
  2 pty

```

```

3 ttyp
4 /dev/vc/0
4 tty
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
21 sg
29 fb
128 ptm
136 pts
171 ieee1394
180 usb
189 usb_device

Block devices:
1 ramdisk
2 fd
8 sd
9 md
22 ide1
...

```

查看 /dev 目录可以获知系统中包含的设备文件，日期的前两列给出了对应设备的主设备号和次设备号：

```

crw-rw---- 1 root      uuqp        4,   64 Jan 30 2003 /dev/ttys0
brw-rw---- 1 root      disk       8,     0 Jan 30 2003 /dev/sda

```

主设备号是与驱动对应的概念，同一类设备一般使用相同的主设备号，不同类的设备一般使用不同的主设备号（但是也不排除在同一主设备号下包含有一定差异的设备）。因为同一驱动可支持多个同类设备，因此用次设备号来描述使用该驱动的设备的序号，序号一般从 0 开始。

内核 Documents 目录下的 devices.txt 文件描述了 Linux 设备号的分配情况，它由 LANANA (the Linux Assigned Names and Numbers authority, 网址为 <http://www.lanana.org/>) 组织维护，Torben Mathiasen (device@lanana.org) 是其中的主要维护者。

5.3 devfs

devfs(设备文件系统)是由 Linux 2.4 内核引入的，引入时被许多工程师给予了高度评价，它的出现使得设备驱动程序能自主地管理自己的设备文件。具体来说，devfs 具有如下优点。

- 1) 可以通过程序在设备初始化时在 /dev 目录下创建设备文件，卸载设备时将它删除。

2) 设备驱动程序可以指定设备名、所有者和权限位，用户空间程序仍可以修改所有者和权限位。

3) 不再需要为设备驱动程序分配主设备号以及处理次设备号，在程序中可以直接给register_chrdev()传递0主设备号以获得可用的主设备号，并在devfs_register()中指定次设备号。

驱动程序应调用下面这些函数来进行设备文件的创建和撤销工作。

```
/* 创建设备目录 */
devfs_handle_t devfs_mk_dir(devfs_handle_t dir, const char *name, void *info);
/* 创建设备文件 */
devfs_handle_t devfs_register(devfs_handle_t dir, const char *name, unsigned
    int flags, unsigned int major, unsigned int minor, umode_t mode, void *ops,
    void *info);
/* 撤销设备文件 */
void devfs_unregister(devfs_handle_t de);
```

在Linux 2.4的设备驱动编程中，分别在模块加载、卸载函数中创建和撤销设备文件是被普遍采用并值得大力推荐的好方法。代码清单5.5给出了一个使用devfs的范例。

代码清单5.5 devfs的使用范例

```
1 static devfs_handle_t devfs_handle;
2 static int __init xxx_init(void)
3 {
4     int ret;
5     int i;
6     /* 在内核中注册设备 */
7     ret = register_chrdev(XXX_MAJOR, DEVICE_NAME, &xxx_fops);
8     if (ret < 0) {
9         printk(DEVICE_NAME " can't register major number\n");
10        return ret;
11    }
12    /* 创建设备文件 */
13    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
14        XXX_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &xxx_fops, NULL);
15    ...
16    printk(DEVICE_NAME " initialized\n");
17    return 0;
18 }
19
20 static void __exit xxx_exit(void)
21 {
22     devfs_unregister(devfs_handle);           /* 撤销设备文件 */
23     unregister_chrdev(XXX_MAJOR, DEVICE_NAME); /* 注销设备 */
24 }
25
26 module_init(xxx_init);
27 module_exit(xxx_exit);
```

代码中第 7 行和第 23 行分别用于注册和注销字符设备，使用的 register_chrdev() 和 unregister_chrdev() 在 Linux 2.6 以后的内核中仍被采用。第 13 和 22 行分别用于创建和删除 devfs 文件节点，这些 API 已经被删除了。

5.4 udev 用户空间设备管理

5.4.1 udev 与 devfs 的区别

尽管 devfs 有这样和那样的优点，但是，在 Linux 2.6 内核中，devfs 被认为是过时的方法，并最终被抛弃了，udev 取代了它。Linux VFS 内核维护者 Al Viro 指出了几点 udev 取代 devfs 的原因：

- 1) devfs 所做的工作被确信可以在用户态来完成。
- 2) devfs 被加入内核之时，大家期望它的质量可以迎头赶上。
- 3) 发现 devfs 有一些可修复和无法修复的 bug。
- 4) 对于可修复的 bug，几个月前就已经被修复了，其维护者认为一切良好。
- 5) 对于后者，在相当长的一段时间内没有改观。
- 6) devfs 的维护者和作者对它感到失望并且已经停止了对代码的维护工作。

Linux 内核的两位贡献者，Richard Gooch（devfs 的作者）和 Greg Kroah-Hartman（sysfs 的主要作者）就 devfs/udev 进行了激烈的争论：

Greg: Richard had stated that udev was a proper replacement for devfs.

Richard: Well, that's news to me!

Greg: devfs should be taken out because policy should exist in userspace and not in the kernel.

Richard: sysfs, developed in large part by Greg, also implemented policy in the kernel.

Greg: devfs was broken and unfixable

Richard: No proof. Never say never...

这段有趣的争论可意译如下：

Greg：Richard 已经指出，udev 是 devfs 合适的替代品。

Richard：哦，我怎么不知道？

Greg：devfs 应该下课，因为策略应该位于用户空间而不是内核空间。

Richard：哦，由 Greg 完成大部分工作的 sysfs 也在内核中实现了策略。

Greg：devfs 很蹩脚，也不稳定。

Richard：呵呵，没证据，别那么武断……

在 Richard Gooch 和 Greg Kroah-Hartman 的争论中，Greg Kroah-Hartman 使用的理论依据就在于 policy（策略）不能位于内核空间中。Linux 设计中强调的一个基本观点是机制和策略的分离。机制是做某样事情的固定步骤、方法，而策略就是每一个步骤所采取的不同方

式。机制是相对固定的，而每个步骤采用的策略是不固定的。机制是稳定的，而策略则是灵活的，因此，在Linux内核中，不应该实现策略。

比如Linux提供API可以让人把线程的优先级调高或者调低，或者调整调度策略为SCHED_FIFO什么的，但是Linux内核本身却不管谁高谁低。提供API属于机制，谁高谁低这属于策略，所以应该是应用程序自己去告诉内核要高或低，而内核不管这些杂事。同理，Greg Kroah-Hartman认为，属于策略的东西应该被移到用户空间中，谁爱给哪个设备创建什么名字或者想做更多的处理，谁自己去设定。内核只管把这些信息告诉用户就行了。这就是位于内核空间的devfs应该被位于用户空间的udev取代的原因，应该devfs管了一些不靠谱的事情。

下面举一个通俗的例子来理解udev设计的出发点。以谈恋爱为例，Greg Kroah-Hartman认为，可以让内核提供谈恋爱的机制，但是不能在内核空间中限制跟谁谈恋爱，不能把谈恋爱的策略放在内核空间。因为恋爱是自由的，用户应该可以在用户空间中实现“萝卜白菜，各有所爱”的理想，可以根据对方的外貌、籍贯、性格等自由选择。对应devfs而言，第1个相亲的女孩被命名为/dev/girl0，第2个相亲的女孩被命名为/dev/girl1，以此类推。而在用户空间实现的udev则可以使得用户实现这样的自由：不管你中意的女孩是第几个，只要它与你定义的规则符合，都命名为/dev/mygirl！

udev完全在用户态工作，利用设备加入或移除时内核所发送的热插拔事件(Hotplug Event)来工作。在热插拔时，设备的详细信息会由内核通过netlink套接字发送出来，发出的事情叫uevent。udev的设备命名策略、权限控制和事件处理都是在用户态下完成的，它利用从内核收到的信息来进行创建设备文件节点等工作。代码清单5.6给出了从内核通过netlink接收热插拔事件并冲刷掉的范例，udev采用了类似的做法。

代码清单5.6 netlink的使用范例

```

1 #include <linux/netlink.h>
2
3 static void die(char *s)
4 {
5     write(2, s, strlen(s));
6     exit(1);
7 }
8
9 int main(int argc, char *argv[])
10 {
11     struct sockaddr_nl nls;
12     struct pollfd pfd;
13     char buf[512];
14
15     //Open hotplug event netlink socket
16
17     memset(&nls, 0, sizeof(struct sockaddr_nl));
18     nls.nl_family = AF_NETLINK;

```

```

19     nls.nl_pid = getpid();
20     nls.nl_groups = -1;
21
22     pfd.events = POLLIN;
23     pfd.fd = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);
24     if (pfid.fd == -1)
25         die("Not root\n");
26
27     //Listen to netlink socket
28     if (bind(pfd.fd, (void *)&nls, sizeof(struct sockaddr_nl)))
29         die("Bind failed\n");
30     while (-1 != poll(&pfid, 1, -1)) {
31         int i, len = recv(pfd.fd, buf, sizeof(buf), MSG_DONTWAIT);
32         if (len == -1)
33             die("recv\n");
34
35         //Print the data to stdout.
36         i = 0;
37         while (i < len) {
38             printf("%s\n", buf + i);
39             i += strlen(buf + i) + 1;
40         }
41     }
42     die("poll\n");
43
44     //Dear gcc: shut up.
45     return 0;
46 }

```

编译上述程序并运行，把 Apple Facetime HD Camera USB 摄像头插入 Ubuntu，该程序会 dump 类似如下的信息：

```

ACTION=add
DEVLINKS=/dev/input/by-id/usb-Apple_Inc._FaceTime_HD_Camera__Built-in_
CC2B2F0TLDG6LL0-event-if00 /dev/input/by-path/pci-0000:00:0b.0-usb-0:1:1.0-event
DEVNAME=/dev/input/event6
DEVPATH=/devices/pci0000:00/0000:00:0b.0/usb1/1-1/1-1:1.0/input/input6/event6
ID_BUS=usb
ID_INPUT=1
ID_INPUT_KEY=1
ID_MODEL=FaceTime_HD_Camera__Built-in_
ID_MODEL_ENC=FaceTime\x20HD\x20Camera\x20\x28Built-in\x29
ID_MODEL_ID=8509
ID_PATH=pci-0000:00:0b.0-usb-0:1:1.0
ID_PATH_TAG=pci-

```

udev 就是采用这种方式接收 netlink 消息，并根据它的内容和用户设置给 udev 的规则做匹配来进行工作的。这里有一个问题，就是冷插拔的设备怎么办？冷插拔的设备在开机时就存在，在 udev 启动前已经被插入了。对于冷插拔的设备，Linux 内核提供了 sysfs 下面

一个 uevent 节点，可以往该节点写一个“add”，导致内核重新发送 netlink，之后 udev 就可以收到冷插拔的 netlink 消息了。我们还是运行代码清单 5.6 的程序，并手动往 /sys/module/psmouse/uevent 写一个“add”，上述程序会 dump 出来这样的信息：

```
ACTION=add
DEVPATH=/module/psmouse
SEQNUM=1682
SUBSYSTEM=module
UDEV_LOG=3
USEC_INITIALIZED=220903546792
```

devfs 与 udev 的另一个显著区别在于：采用 devfs，当一个并不存在的 /dev 节点被打开的时候，devfs 能自动加载对应的驱动，而 udev 则不这么做。这是因为 udev 的设计者认为 Linux 应该在设备被发现的时候加载驱动模块，而不是当它被访问的时候。udev 的设计者认为 devfs 所提供的打开 /dev 节点时自动加载驱动的功能对一个配置正确的计算机来说是多余的。系统中所有的设备都应该产生热插拔事件并加载恰当的驱动，而 udev 能注意到这点并且为它创建对应的设备节点。

5.4.2 sysfs 文件系统与 Linux 设备模型

Linux 2.6 以后的内核引入了 sysfs 文件系统，sysfs 被看成是与 proc、devfs 和 devtmpfs 同类别的文件系统，该文件系统是一个虚拟的文件系统，它可以产生一个包括所有系统硬件的层级视图，与提供进程和状态信息的 proc 文件系统十分类似。

sysfs 把连接在系统上的设备和总线组织成为一个分级的文件，它们可以由用户空间存取，向用户空间导出内核数据结构以及它们的属性。sysfs 的一个目的就是展示设备驱动模型中各组件的层次关系，其顶级目录包括 block、bus、dev、devices、class、fs、kernel、power 和 firmware 等。

block 目录包含所有的块设备；devices 目录包含系统所有的设备，并根据设备挂接的总线类型组织成层次结构；bus 目录包含系统中所有的总线类型；class 目录包含系统中的设备类型（如网卡设备、声卡设备、输入设备等）。在 /sys 目录下运行 tree 会得到一个相当长的树形目录，下面摘取一部分：

```
.
├── block
│   ├── loop0 -> ../devices/virtual/block/loop0
│   ├── loop1 -> ../devices/virtual/block/loop1
│   ├── loop2 -> ../devices/virtual/block/loop2
│   ├── loop3 -> ../devices/virtual/block/loop3
...
└── bus
    └── ac97
        ├── devices
        └── drivers
```

```
    |   |   ├── drivers_autoprobe
    |   |   ├── drivers_probe
    |   |   └── uevent
    |   └── acpi
    ...
    |   └── i2c
    |       ├── devices
    |       ├── drivers
    |       ├── drivers_autoprobe
    |       ├── drivers_probe
    |       └── uevent
    ...
    └── class
        ├── ata_device
        |   ├── dev1.0 -> ../../devices/pci0000:00/0000:00:01.1/ata1/link1/dev1.0/
        |   |   ata_device/dev1.0
        |   ├── dev1.1 -> ../../devices/pci0000:00/0000:00:01.1/ata1/link1/dev1.1/
        |   |   ata_device/dev1.1
        |   ├── dev2.0 -> ../../devices/pci0000:00/0000:00:01.1/ata2/link2/dev2.0/
        |   |   ata_device/dev2.0
        |   ├── dev2.1 -> ../../devices/pci0000:00/0000:00:01.1/ata2/link2/dev2.1/
        |   |   ata_device/dev2.1
    ...
    └── dev
        ├── block
        |   ├── 1:0 -> ../../devices/virtual/block/ram0
        |   ├── 1:1 -> ../../devices/virtual/block/ram1
        |   ├── 1:10 -> ../../devices/virtual/block/ram10
        |   ├── 11:0 -> ../../devices/pci0000:00/0000:00:01.1/ata2/host1/target1:0:0/
        |   |   1:0:0:0/block/sr0
    ...
        └── char
            ├── 10:1 -> ../../devices/virtual/misc/psaux
            ├── 10:184 -> ../../devices/virtual/misc/microcode
            ├── 10:200 -> ../../devices/virtual/misc/tun
            ├── 10:223 -> ../../devices/virtual/misc/uinput
    ...
    └── devices
        ├── breakpoint
        |   ├── power
        |   ├── subsystem -> ../../bus/event_source
        |   ├── type
        |   └── uevent
        ├── isa
        |   ├── power
        |   └── uevent
    ...
    └── firmware
    ...
    └── fs
        ├── cgroup
        ├── ecryptfs
        |   └── version
```

```

    ┌── ext4
    │   ├── features
    │   ├── sdal
    │   └── sdbl
    └── fuse
        └── connections
    ├── hypervisor
    └── kernel
        ├── debug
        │   ├── acpi
        └── bdi
    ...
    └── module
        ├── B250
        │   ├── parameters
        │   └── uevent
        ├── B250_core
        │   ├── parameters
        └── uevent
    ...
    └── power
        ├── disk
        ├── image_size
        ├── pm_async
        ├── reserved_size
        ├── resume
        ├── state
        ├── wake_lock
        └── wakeup_count

```

在 /sys/bus 的 pci 等子目录下，又会再分出 drivers 和 devices 目录，而 devices 目录中的文件是对 /sys/devices 目录中文件的符号链接。同样地，/sys/class 目录下也包含许多对 /sys/devices 下文件的链接。如图 5.3 所示，Linux 设备模型与设备、驱动、总线和类的现实状况是直接对应的，也正符合 Linux 2.6 以后内核的设备模型。

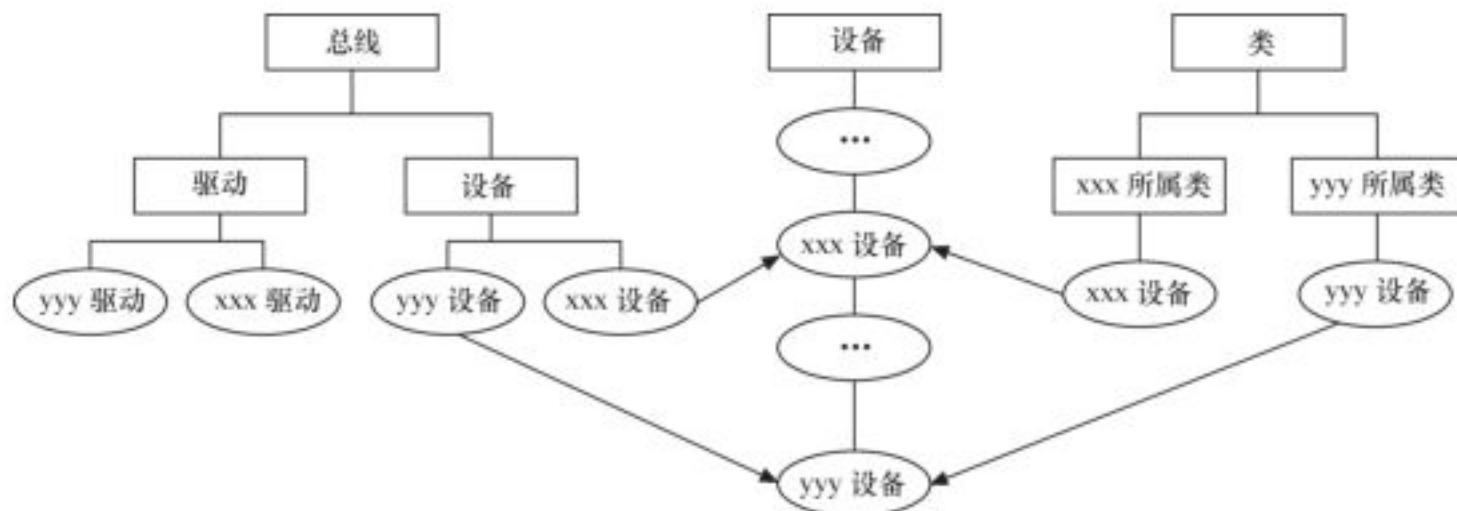


图 5.3 Linux 设备模型

随着技术的不断进步，系统的拓扑结构越来越复杂，对智能电源管理、热插拔以及即插即用的支持要求也越来越高，Linux 2.4 内核已经难以满足这些需求。为适应这种形势的需要，Linux 2.6 以后的内核开发了上述全新的设备、总线、类和驱动环环相扣的设备模型。图 5.4 形象地表示了 Linux 驱动模型中设备、总线和类之间的关系。

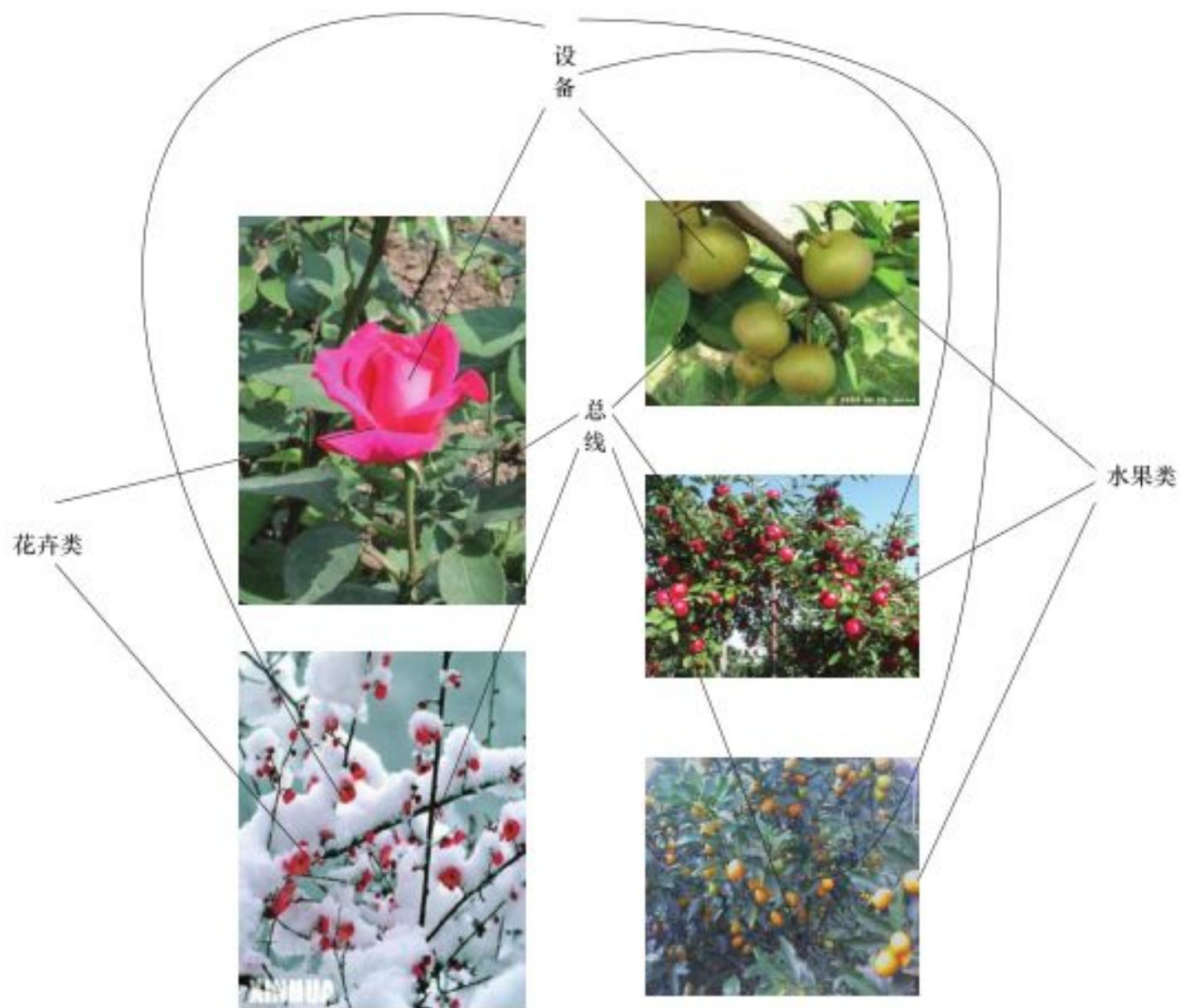


图 5.4 Linux 驱动模型中设备、总线和类的关系

大多数情况下，Linux 2.6 以后的内核中的设备驱动核心层代码作为“幕后大佬”可处理好这些关系，内核中的总线和其他内核子系统会完成与设备模型的交互，这使得驱动工程师在编写底层驱动的时候几乎不需要关心设备模型，只需要按照每个框架的要求，“填鸭式”地填充 `xxx_driver` 里面的各种回调函数，`xxx` 是总线的名字。

在 Linux 内核中，分别使用 `bus_type`、`device_driver` 和 `device` 来描述总线、驱动和设备，这 3 个结构体定义于 `include/linux/device.h` 头文件中，其定义如代码清单 5.7 所示。

代码清单 5.7 bus_type、device_driver 和 device 结构体

```

1 struct bus_type {
2     const char          *name;
3     const char          *dev_name;
4     struct device       *dev_root;
5     struct device_attribute *dev_attrs; /* use dev_groups instead */
6     const struct attribute_group **bus_groups;
7     const struct attribute_group **dev_groups;
8     const struct attribute_group **drv_groups;
9
10    int (*match)(struct device *dev, struct device_driver *drv);
11    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
12    int (*probe)(struct device *dev);
13    int (*remove)(struct device *dev);
14    void (*shutdown)(struct device *dev);
15
16    int (*online)(struct device *dev);
17    int (*offline)(struct device *dev);
18
19    int (*suspend)(struct device *dev, pm_message_t state);
20    int (*resume)(struct device *dev);
21
22    const struct dev_pm_ops *pm;
23
24    struct iommu_ops *iommu_ops;
25
26    struct subsys_private *p;
27    struct lock_class_key lock_key;
28 };
29
30 struct device_driver {
31     const char          *name;
32     struct bus_type      *bus;
33
34     struct module        *owner;
35     const char          *mod_name; /* used for built-in modules */
36
37     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
38
39     const struct of_device_id      *of_match_table;
40     const struct acpi_device_id    *acpi_match_table;
41
42     int (*probe)(struct device *dev);
43     int (*remove)(struct device *dev);
44     void (*shutdown)(struct device *dev);
45     int (*suspend)(struct device *dev, pm_message_t state);
46     int (*resume)(struct device *dev);
47     const struct attribute_group **groups;
48
49     const struct dev_pm_ops *pm;

```

```
50
51         struct driver_private *p;
52     };
53
54 struct device {
55     struct device     *parent;
56
57     struct device_private      *p;
58
59     struct kobject kobj;
60     const char          *init_name; /* initial name of the device */
61     const struct device_type *type;
62
63     struct mutex        mutex; /* mutex to synchronize calls to
64                           * its driver.
65                           */
66
67     struct bus_type      *bus;           /* type of bus device is on */
68     struct device_driver *driver;       /* which driver has allocated this
69                           * device */
70     void             *platform_data;   /* Platform specific data, device
71                           * core doesn't touch it */
72     struct dev_pm_info  power;
73     struct dev_pm_domain *pm_domain;
74
75 #ifdef CONFIG_PINCTRL
76     struct dev_pin_info *pins;
77 #endif
78
79 #ifdef CONFIG_NUMA
80     int             numa_node;        /* NUMA node this device is close to */
81 #endif
82     u64            *dma_mask;        /* dma mask (if dma'ble device) */
83     u64            coherent_dma_mask; /* Like dma_mask, but for
84                           * alloc_coherent mappings as
85                           * not all hardware supports
86                           * 64 bit addresses for consistent
87                           * allocations such descriptors. */
88
89     struct device_dma_parameters *dma_parms;
90
91     struct list_head dma_pools; /* dma pools (if dma'ble) */
92
93     struct dma_coherent_mem      *dma_mem; /* internal for coherent mem
94                           * override */
95 #ifdef CONFIG_DMA_CMA
96     struct cma *cma_area;        /* contiguous memory area for dma
97                           * allocations */
98 #endif
99 /* arch specific additions */
```

```

100 struct dev_archdata archdata;
101
102 struct device_node *of_node; /* associated device tree node */
103 struct acpi_dev_node acpi_node; /* associated ACPI device node */
104
105 dev_t devt; /* dev_t, creates the sysfs "dev" */
106 u32 id; /* device instance */
107
108 spinlock_t devres_lock;
109 struct list_head devres_head;
110
111 struct klist_node knode_class;
112 struct class *class;
113 const struct attribute_group **groups; /* optional groups */
114
115 void (*release)(struct device *dev);
116 struct iommu_group *iommu_group;
117
118 bool offline_disabled:1;
119 bool offline:1;
120 };

```

`device_driver` 和 `device` 分别表示驱动和设备，而这两者都必须依附于一种总线，因此都包含 `struct bus_type` 指针。在 Linux 内核中，设备和驱动是分开注册的，注册 1 个设备的时候，并不需要驱动已经存在，而 1 个驱动被注册的时候，也不需要对应的设备已经被注册。设备和驱动各自涌向内核，而每个设备和驱动涌入内核的时候，都会去寻找自己的另一半，而正是 `bus_type` 的 `match()` 成员函数将两者捆绑在一起。简单地说，设备和驱动就是红尘中漂浮的男女，而 `bus_type` 的 `match()` 则是牵引红线的月老，它可以识别什么设备与什么驱动是可配对的。一旦配对成功，`xxx_driver` 的 `probe()` 就被执行（`xxx` 是总线名，如 `platform`、`pci`、`i2c`、`spi`、`usb` 等）。

注意：总线、驱动和设备最终都会落实为 `sysfs` 中的 1 个目录，因为进一步追踪代码会发现，它们实际上都可以认为是 `kobject` 的派生类，`kobject` 可看作是所有总线、设备和驱动的抽象基类，1 个 `kobject` 对应 `sysfs` 中的 1 个目录。

总线、设备和驱动中的各个 `attribute` 则直接落实为 `sysfs` 中的 1 个文件，`attribute` 会伴随着 `show()` 和 `store()` 这两个函数，分别用于读写该 `attribute` 对应的 `sysfs` 文件，代码清单 5.8 给出了 `attribute`、`bus_attribute`、`driver_attribute` 和 `device_attribute` 这几个结构体的定义。

代码清单 5.8 `attribute`、`bus_attribute`、`driver_attribute` 和 `device_attribute` 结构体

```

1 struct attribute {
2     const char *name;
3     umode_t mode;
4 #ifdef CONFIG_DEBUG_LOCK_ALLOC
5     bool ignore_lockdep:1;

```

```

6         struct lock_class_key    *key;
7         struct lock_class_key    skey;
8 #endif
9 };
10
11 struct bus_attribute {
12     struct attribute        attr;
13     ssize_t (*show)(struct bus_type *bus, char *buf);
14     ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
15 };
16
17 struct driver_attribute {
18     struct attribute attr;
19     ssize_t (*show)(struct device_driver *driver, char *buf);
20     ssize_t (*store)(struct device_driver *driver, const char *buf,
21                      size_t count);
22 };
23
24 struct device_attribute {
25     struct attribute        attr;
26     ssize_t (*show)(struct device *dev, struct device_attribute *attr,
27                      char *buf);
28     ssize_t (*store)(struct device *dev, struct device_attribute *attr,
29                      const char *buf, size_t count);
30 };

```

事实上，sysfs 中的目录来源于 bus_type、device_driver、device，而目录中的文件则来源于 attribute。Linux 内核中也定义了一些快捷方式以方便 attribute 的创建工作。

```

#define DRIVER_ATTR(_name, _mode, _show, _store) \
struct driver_attribute driver_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define DRIVER_ATTR_RW(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RW(_name)
#define DRIVER_ATTR_RO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RO(_name)
#define DRIVER_ATTR_WO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_WO(_name)

#define DRIVER_ATTR(_name, _mode, _show, _store) \
    struct driver_attribute driver_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define DRIVER_ATTR_RW(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RW(_name)
#define DRIVER_ATTR_RO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RO(_name)
#define DRIVER_ATTR_WO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_WO(_name)

#define BUS_ATTR(_name, _mode, _show, _store) \
    struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define BUS_ATTR_RW(_name) \

```

```

    struct bus_attribute bus_attr_##_name = __ATTR_RW(_name)
#define BUS_ATTR_RO(_name) \
    struct bus_attribute bus_attr_##_name = __ATTR_RO(_name)

```

比如，我们在 drivers/base/bus.c 文件中可以找到这样的代码：

```

static BUS_ATTR(drivers_probe, S_IWUSR, NULL, store_drivers_probe);
static BUS_ATTR(drivers_autoprobe, S_IWUSR | S_IRUGO,
               show_drivers_autoprobe, store_drivers_autoprobe);
static BUS_ATTR(uevent, S_IWUSR, NULL, bus_uevent_store);

```

而在 /sys/bus/platform 等里面就可以找到对应的文件：

```

barry@barry-VirtualBox:/sys/bus/platform$ ls
devices  drivers  drivers_autoprobe  drivers_probe  uevent

```

代码清单 5.9 的脚本可以遍历整个 sysfs，并且 dump 出来总线、设备和驱动信息。

代码清单 5.9 遍历 sysfs

```

1 #!/bin/bash
2
3  # Populate block devices
4
5  for i in /sys/block/*/* /dev /sys/block/*/*/* /dev
6  do
7      if [ -f $i ]
8      then
9          MAJOR=$(sed 's/:.*//' < $i)
10         MINOR=$(sed 's/.*/::/' < $i)
11         DEVNAME=$(echo $i | sed -e 's@/dev@@' -e 's@.*@@')
12         echo /dev/$DEVNAME b $MAJOR $MINOR
13         #mknod /dev/$DEVNAME b $MAJOR $MINOR
14     fi
15 done
16
17 # Populate char devices
18
19 for i in /sys/bus/*/* /dev /sys/class/*/* /dev
20 do
21     if [ -f $i ]
22     then
23         MAJOR=$(sed 's/:.*//' < $i)
24         MINOR=$(sed 's/.*/::/' < $i)
25         DEVNAME=$(echo $i | sed -e 's@/dev@@' -e 's@.*@@')
26         echo /dev/$DEVNAME c $MAJOR $MINOR
27         #mknod /dev/$DEVNAME c $MAJOR $MINOR
28     fi
29 done

```

上述脚本遍历 sysfs，找出所有的设备，并分析出来设备名和主次设备号。如果我们把

27行前的“#”去掉，该脚本实际上还可以为整个系统中的设备建立 /dev/ 下面的节点。

5.4.3 udev 的组成

udev 目前和 systemd 项目合并在一起了，见位于 <https://lwn.net/Articles/490413/> 的文档《Udev and systemd to merge》，可以从 <http://cgit.freedesktop.org/systemd/>、<https://github.com/systemd/systemd> 等位置下载最新的代码。udev 在用户空间中执行，动态建立 / 删除设备文件，允许每个人都不用关心主 / 次设备号而提供 LSB (Linux 标准规范，Linux Standard Base) 名称，并且可以根据需要固定名称。udev 的工作过程如下。

- 1) 当内核检测到系统中出现了新设备后，内核会通过 netlink 套接字发送 uevent。
- 2) udev 获取内核发送的信息，进行规则的匹配。匹配的事物包括 SUBSYSTEM、ACTION、attribute、内核提供的名称（通过 KERNEL=）以及其他环境变量。

假设在 Linux 系统上插入一个 Kingston 的 U 盘，我们可以通过 udev 的工具“udevadm monitor --kernel --property --udev”捕获到的 uevent 包含的信息：

```

UDEV [6328.797974] add
/devices/pci0000:00/0000:00:0b.0/usb1/1-1/1-1:1.0/host7/target7:0:0/7:0:0:0/block/sdc (block)
ACTION=add
DEVLINKS=/dev/disk/by-id/usb-Kingston_DataTraveler_2.0_5B8212000047-0:0 /dev/
    disk/by-path/pci-0000:00:0b.0-usb-0:1:1.0-scsi-0:0:0:0
DEVNAME=/dev/sdc
DEVPATH=/devices/pci0000:00/0000:00:0b.0/usb1/1-1/1-1:1.0/host7/target7:0:0/7:0:0:0/
    block/sdc
DEVTYPE=disk
ID_BUS=usb
ID_INSTANCE=0:0
ID_MODEL=DataTraveler_2.0
ID_MODEL_ENC=DataTraveler\x202.0
ID_MODEL_ID=6545
ID_PART_TABLE_TYPE=dos
ID_PATH=pci-0000:00:0b.0-usb-0:1:1.0-scsi-0:0:0:0
ID_PATH_TAG=pci-0000_00_0b_0-usb-0_1_1_0-scsi-0_0_0_0
ID_REVISION=PMAP
ID_SERIAL=Kingston_DataTraveler_2.0_5B8212000047-0:0
ID_SERIAL_SHORT=5B8212000047
ID_TYPE=disk
ID_USB_DRIVER=usb-storage
ID_USB_INTERFACES=:080650:
ID_USB_INTERFACE_NUM=00
ID_VENDOR=Kingston
ID_VENDOR_ENC=Kingston
ID_VENDOR_ID=0930
MAJOR=8
MINOR=32
SEQNUM=2335
SUBSYSTEM=block

```

我们可以根据这些信息，创建一个规则，以便每次插入的时候，为该盘创建一个 /dev/kingstonUD 的符号链接，这个规则可以写成代码清单 5.10 的样子。

代码清单 5.10 设备命名规则范例

```
# Kingston USB mass storage
SUBSYSTEM=="block", ACTION=="add", KERNEL=="sd?", ENV{ID_TYPE}=="disk",
ENV{ID_VENDOR}=="Kingston", ENV{ID_USB_DRIVER}=="usb-storage", SYMLINK+="kingstonUD"
```

插入 kingston U 盘后，/dev/ 会自动创建一个符号链接：

```
root@barry-VirtualBox:/dev# ls -l kingstonUD
lrwxrwxrwx 1 root root 3 Jun 30 19:31 kingstonUD -> sdc
```

5.4.4 udev 规则文件

udev 的规则文件以行为单位，以 “#” 开头的行代表注释行。其余的每一行代表一个规则。每个规则分成一个或多个匹配部分和赋值部分。匹配部分用匹配专用的关键字来表示，相应的赋值部分用赋值专用的关键字来表示。匹配关键字包括：ACTION（行为）、KERNEL（匹配内核设备名）、BUS（匹配总线类型）、SUBSYSTEM（匹配子系统名）、ATTR（属性）等，赋值关键字包括：NAME（创建的设备文件名）、SYMLINK（符号创建链接名）、OWNER（设置设备的所有者）、GROUP（设置设备的组）、IMPORT（调用外部程序）、MODE（节点访问权限）等。

例如，如下规则：

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="08:00:27:35:be:ff",
ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="eth*", NAME="eth1"
```

其中的“匹配”部分包括 SUBSYSTEM、ACTION、ATTR、KERNEL 等，而“赋值”部分有一项，是 NAME。这个规则的意思是：当系统中出现的新硬件属于 net 子系统范畴，系统对该硬件采取的动作是“add”这个硬件，且这个硬件的“address”属性信息等于“08:00:27:35:be:ff”，“dev_id”属性等于“0x0”、“type”属性为 1 等，此时，对这个硬件在 udev 层次施行的动作是创建 /dev/eth1。

通过一个简单的例子可以看出 udev 和 devfs 在命名方面的差异。如果系统中有两个 USB 打印机，一个可能被称为 /dev/usb/lp0，另外一个便是 /dev/usb/lp1。但是到底哪个文件对应哪个打印机是无法确定的，lp0、lp1 和实际的设备没有一一对应的关系，映射关系会因设备发现的顺序、打印机本身关闭等而不确定。因此，理想的方式是两个打印机应该采用基于它们的序列号或者其他标识办法来进行确定的映射，devfs 无法做到这一点，udev 却可以做到。使用如下规则：

```
SUBSYSTEM=="usb", ATTR{serial}=="HXOLL0012202323480", NAME="lp_epson", SYMLINK+="printers/
epson_stylus"
```

该规则中的匹配项目有 SUBSYSTEM 和 ATTR，赋值项目为 NAME 和 SYMLINK，它意味着当一台 USB 打印机的序列号为“HXOLL0012202323480”时，创建 /dev/lp_epson 文件，并同时创建一个符号链接 /dev/printers/epson_styles。序列号为“HXOLL0012202323480”的 USB 打印机不管何时被插入，对应的设备名都是 /dev/lp_epson，而 devfs 显然无法实现设备的这种固定命名。

udev 规则的写法非常灵活，在匹配部分，可以通过“*”、“?”、[a ~ c]、[1 ~ 9] 等 shell 通配符来灵活匹配多个项目。* 类似于 shell 中的 * 通配符，代替任意长度的任意字符串，? 代替一个字符。此外，%k 就是 KERNEL，%n 则是设备的 KERNEL 序号（如存储设备的分区号）。

可以借助 udev 中的 udevadm info 工具查找规则文件能利用的内核信息和 sysfs 属性信息，如运行“udevadm info -a -p /sys/devices/platform/serial8250/tty/ttys0”命令将得到：

```
udevadm info -a -p /sys/devices/platform/serial8250/tty/ttys0

Udevadm info starts with the device specified by the devpath and then
walks up the chain of parent devices. It prints for every device
found, all possible attributes in the udev rules key format.
A rule to match, can be composed by the attributes of the device
and the attributes from one single parent device.

looking at device '/devices/platform/serial8250/tty/ttys0':
KERNEL=="ttys0"
SUBSYSTEM=="tty"
DRIVER=""
ATTR{irq}=="4"
ATTR{line}=="0"
ATTR{port}=="0x3F8"
ATTR{type}=="0"
ATTR{flags}=="0x10000040"
ATTR{iomem_base}=="0x0"
ATTR{custom_divisor}=="0"
ATTR{iomem_reg_shift}=="0"
ATTR{uartclk}=="1843200"
ATTR{xmit_fifo_size}=="0"
ATTR{close_delay}=="50"
ATTR{closing_wait}=="3000"
ATTR{io_type}=="0"

looking at parent device '/devices/platform/serial8250':
KERNELS=="serial8250"
SUBSYSTEMS=="platform"
DRIVERS=="serial8250"

looking at parent device '/devices/platform':
KERNELS=="platform"
SUBSYSTEMS=="platform"
```

```
DRIVERS=""
```

如果 /dev/ 下面的节点已经被创建，但是不知道它对应的 /sys 具体节点路径，可以采用 “udevadm info -a -p \$(udevadm info -q path -n /dev/<节点名>)” 命令反向分析，比如：

```
udevadm info -a -p $(udevadm info -q path -n /dev/sdb)
```

```
Udevadm info starts with the device specified by the devpath and then
walks up the chain of parent devices. It prints for every device
found, all possible attributes in the udev rules key format.
A rule to match, can be composed by the attributes of the device
and the attributes from one single parent device.
```

```
looking at device '/devices/pci0000:00/0000:00:0d.0/ata4/host3/target3:0:0:0:0/block/sdb':
  KERNEL=="sdb"
  SUBSYSTEM=="block"
  DRIVER=""
  ATTR{ro}=="0"
  ATTR{size}=="71692288"
  ATTR{stat}=="      584      1698      4669      7564          0          0          0
                0      7564      7564"
  ATTR{range}=="16"
  ATTR{discard_alignment}=="0"
  ATTR{events}==""
  ATTR{ext_range}=="256"
  ATTR{events_poll_msecs}=="-1"
  ATTR{alignment_offset}=="0"
  ATTR{inflight}=="          0          0"
  ATTR{removable}=="0"
  ATTR{capability}=="50"
  ATTR{events_async}==""

looking at parent device '/devices/pci0000:00/0000:00:0d.0/ata4/host3/
  target3:0:0:0:0:0':
  KERNELS=="3:0:0:0"
  SUBSYSTEMS=="scsi"
  DRIVERS=="sd"
  ATTRS{rev}=="1.0 "
  ATTRS{type}=="0"
  ATTRS{scsi_level}=="6"
  ATTRS{model}=="VBOX HARDDISK"
  ATTRS{state}=="running"
  ATTRS{queue_type}=="simple"
  ATTRS{iodone_cnt}=="0x299"
  ATTRS{iorequest_cnt}=="0x29a"
  ATTRS{queue_ramp_up_period}=="120000"
  ATTRS{timeout}=="30"
  ATTRS{evt_media_change}=="0"
  ATTRS{ioerr_cnt}=="0x7"
  ATTRS{queue_depth}=="31"
```

```

ATTRS{vendor}=="ATA      "
ATTRS{device_blocked}=="0"
ATTRS{iocounterbits}=="32"

looking at parent device '/devices/pci0000:00/0000:00:0d.0/ata4/host3/
    target3:0:0':
KERNELS=="target3:0:0"
SUBSYSTEMS=="scsi"
DRIVERS==""

looking at parent device '/devices/pci0000:00/0000:00:0d.0/ata4/host3':
KERNELS=="host3"
SUBSYSTEMS=="scsi"
DRIVERS==""

looking at parent device '/devices/pci0000:00/0000:00:0d.0/ata4':
KERNELS=="ata4"
SUBSYSTEMS==""
DRIVERS==""

looking at parent device '/devices/pci0000:00/0000:00:0d.0':
KERNELS=="0000:00:0d.0"
SUBSYSTEMS=="pci"
DRIVERS=="ahci"
ATTRS{irq}=="21"
ATTRS{subsystem_vendor}=="0x0000"
ATTRS{broken_parity_status}=="0"
ATTRS{class}=="0x010601"
ATTRS{consistent_dma_mask_bits}=="64"
ATTRS{dma_mask_bits}=="64"
ATTRS{local_cpus}=="ff"
ATTRS{device}=="0x2829"
ATTRS{enable}=="1"
ATTRS{msi_bus}==""
ATTRS{local_cpulist}=="0-7"
ATTRS{vendor}=="0x8086"
ATTRS{subsystem_device}=="0x0000"
ATTRS{d3cold_allowed}=="1"

looking at parent device '/devices/pci0000:00':
KERNELS=="pci0000:00"
SUBSYSTEMS==""
DRIVERS==""

```

在嵌入式系统中，也可以用 udev 的轻量级版本 mdev，mdev 集成于 busybox 中。在编译 busybox 的时候，选中 mdev 相关项目即可。

Android 也没有采用 udev，它采用的是 vold。vold 的机制和 udev 是一样的，理解了 udev，也就理解了 vold。Android 的源代码 NetlinkManager.cpp 同样是监听基于 netlink 的套接字，并解析收到的消息。

5.5 总结

Linux 用户空间的文件编程有两种方法，即通过 Linux API 和通过 C 库函数访问文件。用户空间看不到设备驱动，能看到的只有与设备对应的文件，因此文件编程也就是用户空间的设备编程。

Linux 按照功能对文件系统的目录结构进行了良好的规划。`/dev` 是设备文件的存放目录，`devfs` 和 `udev` 分别是 Linux 2.4 和 Linux 2.6 以后的内核生成设备文件节点的方法，前者运行于内核空间，后者运行于用户空间。

Linux 2.6 以后的内核通过一系列数据结构定义了设备模型，设备模型与 `sysfs` 文件系统中的目录和文件存在一种对应关系。设备和驱动分离，并通过总线进行匹配。

`udev` 可以利用内核通过 `netlink` 发出的 `uevent` 信息动态创建设备文件节点。