LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 5 章　Linux 文件系统与设备文件系统

由于字符设备和块设备都很好地体现了“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非常重要。

首先，设备驱动最终通过操作系统的文件系统调用或 C 库函数（本质也基于系统调用）被访问。

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，如 Linux 2.4 内核的 devfs 文件系统和 Linux 2.6 内核的基于 sysfs 的 udev 文件系统。

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 内核所采用的 udev 设备文件系统，并分析了两者的区别。

5.1 Linux 文件操作

5.1.1 文件操作的相关系统调用

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，如下所示：

```
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，默认时认为在当前路径下面），flags 可以是如表 5.1 所示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 这 3 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，用来表示文件的访问权限。mode 可以是如表 5.2 所示值的组合。

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

续表

标　志	含　义
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	设置用户的执行 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 总共用 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；第五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，如果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 10705，如下所示：

```
open("test", O_CREAT, 10705);
```

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 系统中提供文件读写的系统

调用是 read、write 函数，如下所示：

```
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 实现将把 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 函数实际上实现了文件创建的功能，因此，下面的函数等同 creat()函数：

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

4．定位

对于随机文件，我们可以随机地指定位置读写，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 可以使用如下值。

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

当操作完成以后，就要关闭文件了，只要调用 close 函数就可以了，其中 fd 是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，如代码清单 5.1 所示。

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    {
15      write(fd, "Hello World", strlen("Hello World")); /*写入字符串 */
16
17      close(fd);
18    }
19
20    fd = open("hello.txt", O_RDWR);
21    len = read(fd, str, LENGTH); /* 读取文件内容 */
22    str[len] = '\0';
23    printf("%s\n", str);
24    close(fd);
25  }
```

编译并运行，执行结果为输出“Hello World”。

5.1.2 C 库函数的文件操作

C 库函数的文件操作实际上是独立于具体的操作系统平台的，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()实现打开指定文件 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，则创建新文件，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 系统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 stream 中读取 *n* 个字段，每个字段为 size 个字节，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。在读取的字段数小于 num 时，可能是在函数调用时出现错误，也可能是读到文件的结尾。所以要通过调用 feof()和 ferror()来判断。

write()实现从缓冲区 ptr 所指的数组中把 *n* 个字段写到 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是很简单的操作，如下所示：

```
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范例（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    {
11      fputs("Hello World", fd); /* 写入字符串 */
12      fclose(fd);
13    }
14
15    fd = fopen("hello.txt", "r");
16    fgets(str, LENGTH, fd); /* 读取文件内容 */
17    printf("%s\n", str);
18    fclose(fd);
19  }
```

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，可以看到 Linux 系统包含以下目录。

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．/boot

Linux 系统的内核及引导系统程序所需要的文件，如 vmlinuz、initrd.img 文件都位于这个目录中。

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就可以访问实际的设备。

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。

5．/home

普通用户的家目录。

6．/lib

库文件存放目录。

7．/lost+found

在 Ext2 或 Ext3 文件系统中，当系统意外崩溃或机器意外关机时会产生一些文件碎片放在这里。

8．/mnt

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

9．/opt

opt 是“可选”的意思，有些软件包会被安装在这里，比如在 Fedora Core 5.0 中的 OpenOffice 就是安装在这里，用户自己编译的软件包也可以安装在这个目录中。

10．/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）存放在这里。/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。

11．/root

Linux 超级权限用户 root 的家目录。

12．/sbin

存放可执行文件，大多是涉及系统管理的命令，是超级权限用户 root 的可执行命令存放地，普通用户无权限执行这个目录下的命令，这个目录和/usr/sbin;/usr/X11R6/sbin 或/usr/local/sbin 目录是相似的。

13．/tmp

有时用户运行程序的时候会产生临时文件，/tmp 用来存放临时文件。

14．/usr

这个是系统存放程序的目录，比如命令、帮助文件等，它包含很多文件和目录，Linux 发行版提供的软件包大多被安装在这里。

15．/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。

16．/sys

Linux 2.6 内核所支持的 sysfs 文件系统被映射在此目录。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。

17．/initrd

若在启动过程中使用了 initrd 映像作为临时根文件系统，则在执行完其上的/linuxrc 挂接真正的根文件系统后，原来的初始 RAM 文件系统被映射到/initrd 目录。

5.2.2 Linux 文件系统与设备驱动

图 5.1 所示为 Linux 系统中虚拟文件系统、磁盘文件（存放于 RamDisk、Flash、ROM、SD 卡、U 盘等文件系统中的文件也属于磁盘文件）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 与磁盘文件系统以及普通设备之间的接口是 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。

由于字符设备的上层没有磁盘文件系统，所以字符设备的 file_operations 成员函数就直接由设备驱动提供了，file_operations 正是字符设备驱动的核心。

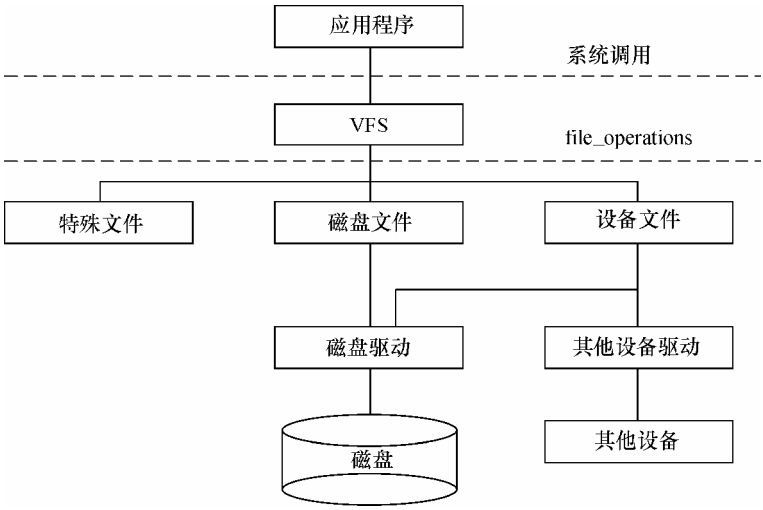

图 5.1　文件系统与设备驱动

而对于块存储设备而言，ext2、fat、jffs2 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。磁盘文件系统和设备驱动会将对磁盘上文件的访问最终转换成对磁盘上柱面和扇区的访问。

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

1．file 结构体

文件结构体代表一个打开的文件（设备对应于设备文件），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  {
3    union
4    {
5      struct list_head fu_list;
6      struct rcu_head fu_rcuhead;
7    } f_u;
8    struct dentry *f_dentry; /*与文件关联的目录入口(dentry)结构*/
9    struct vfsmount *f_vfsmnt;
10   struct file_operations *f_op; /* 和文件关联的操作*/
11   atomic_t f_count;
12   unsigned int f_flags; /*文件标志，如 O_RDONLY、O_NONBLOCK、O_SYNC*/
13   mode_t f_mode; /*文件读/写模式，FMODE_READ 和 FMODE_WRITE*/
14   loff_t f_pos; /* 当前读写位置*/
15   struct fown_struct f_owner;
16   unsigned int f_uid, f_gid;
17   struct file_ra_state f_ra;
```

```
18
19   unsigned long f_version;
20   void *f_security;
21
22   /* tty 驱动需要，其他的驱动可能需要 */
23   void *private_data; /*文件私有数据*/
24
25   #ifdef CONFIG_EPOLL
26     /* 被 fs/eventpoll.c 使用以便连接所有这个文件的钩子(hooks) */
27     struct list_head f_ep_links;
28     spinlock_t f_ep_lock;
29   #endif /* #ifdef CONFIG_EPOLL */
30   struct address_space *f_mapping;
31 };
```

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用于描述设备的结构体。

驱动程序中经常会使用如下类似的代码来检测用户打开文件的读写方式。

```
if (file->f_mode & FMODE_WRITE) //用户要求可写
{
}
if (file->f_mode & FMODE_READ) //用户要求可读
{
}
```

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    ...
4    umode_t i_mode; /* inode 的权限 */
5    uid_t i_uid; /* inode 拥有者的 id */
```

```
    6    gid_t i_gid; /* inode 所属的群组 id */
    7    dev_t i_rdev; /* 若是设备文件，此字段将记录设备的设备号 */
    8    loff_t i_size; /* inode 所代表的文件大小 */
    9
    10   struct timespec i_atime; /* inode 最近一次的存取时间 */
    11   struct timespec i_mtime; /* inode 最近一次的修改时间 */
    12   struct timespec i_ctime; /* inode 的产生时间 */
    13
    14   unsigned long i_blksize; /* inode 在做 I/O 时的区块大小 */
    15   unsigned long i_blocks; /* inode 所使用的 block 数，一个 block 为 512
byte*/
    16
    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 };
```

对于表示设备文件的 inode 结构，i_rdev 字段包含设备编号。Linux 2.6 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，如下所示：

```
    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，如下所示：

```
crw-rw----    1 root     uucp       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：http://www.lanana.org/）组织维护，Torben Mathiasen 是其中的主要维护者。需要注意的是，LANANA 给出的设备号标准并不是硬性规定，在具体的设备驱动程序中，尽管一般会遵循 LANANA，但是也可以有例外。

5.3 devfs 设备文件系统

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它自己的设备文件。具体来说，devfs 具有如下优点。

- 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。
- 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。
- 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 主设备号以动态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删除工作。

```
    /*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，分别在模块加载和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 的例子。

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     {
10      printk(DEVICE_NAME " can't register major number\n");
11      return ret;
12    }
13    /*创建设备文件*/
14    devfs_handle         =devfs_register(NULL,         DEVICE_NAME,
DEVFS_FL_DEFAULT,
15    XXX_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &xxx_fops, NULL);
16    ...
17    printk(DEVICE_NAME " initialized\n");
18    return 0;
19  }
20
21  static void _ _exit xxx_exit(void)
22  {
23    devfs_unregister(devfs_handle); /*撤销设备文件*/
24    unregister_chrdev(XXX_MAJOR, DEVICE_NAME); /*注销设备*/
25  }
26
27  module_init(xxx_init);
28  module_exit(xxx_exit);
```

代码中第 7 行和第 24 行分别用于注册和注销字符设备，使用的 register_chrdev()和 unregister_ chrdev()在 Linux2.6 内核中虽然仍然被支持，但是是过时的做法。第 14 行和第 23 行分别用于创建和删除 devfs 文件节点。

5.4 udev 设备文件系统

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，udev 取代了它。Linux VFS 内核维护者 Al Viro 指出了 udev 取代 devfs 的几点原因：

- devfs 所做的工作被确信可以在用户态来完成。
- 一些 bug 相当长的时间内未被修复。
- devfs 的维护者和作者停止了对代码的维护工作。

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输出到位于/sys 的 sysfs 文件系统。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 sysfs 中的信息来进行创建设备文件节点等工作。

由于 udev 根据系统中硬件设备的状态动态更新设备文件，进行设备文件的创建和删除等，因此，在使用 udev 后，/dev 目录下就会只包含系统中真正存在的设备了。

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 则不能。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于一个配置正确的计算机是多余的。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

1．sysfs 文件系统

Linux 2.6 内核引入了 sysfs 文件系统，sysfs 被看成是与 proc、devfs 和 devpty 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、device、bus、drivers、class、power 和 firmware。

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，drivers 目录包括内核中所有已注册的设备驱动程序，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 目录运行 tree 会得到一个相当长的树型目录，下面摘取一部分：

```
    |-- block
    |   |-- fd0
    |   |-- md0
    |   |-- ram0
    |   |-- ram1
    |   |-- ...
    |-- bus
    |   |-- eisa
    |   |   |-- devices
    |   |   '-- drivers
    |   |-- ide
    |   |-- ieee1394
    |   |-- pci
    |   |   |-- devices
    |               |               |               |--     0000:00:00.0
-> ../../../devices/pci0000:00/0000:00:00.0
    |               |               |               |--     0000:00:01.0
-> ../../../devices/pci0000:00/0000:00:01.0
    |               |               |               |--     0000:00:07.0
-> ../../../devices/pci0000:00/0000:00:07.0
    |   |   '-- drivers
    |   |       |-- PCI_IDE
    |   |       |   |-- bind
    |   |       |   |-- new_id
    |   |       |   '-- unbind
    |   |       '-- pcnet32
    |            |                                  |--    0000:00:11.0
-> ../../../../devices/pci0000:00/0000:00:11.0
    |   |          |-- bind
    |   |          |-- new_id
    |   |          '-- unbind
    |   |-- platform
    |   |-- pnp
    |   '-- usb
    |       |-- devices
    |       '-- drivers
    |           |-- hub
    |           |-- usb
    |           |-- usb-storage
    |           '-- usbfs
    |-- class
    |   |-- graphics
    |   |-- hwmon
    |   |-- ieee1394
    |   |-- ieee1394_host
    |   |-- ieee1394_node
    |   |-- ieee1394_protocol
    |   |-- input
    |   |-- mem
    |   |-- misc
    |   |-- net
    |   |-- pci_bus
    |   |   |-- 0000:00
    |   |   |   |-- bridge -> ../../../devices/pci0000:00
    |   |   |   |-- cpuaffinity
    |   |   |   '-- uevent
    |   |   '-- 0000:01
```

```
    |   |      |-- bridge -> ../../../devices/pci0000:00/0000:00:01.0
    |   |      |-- cpuaffinity
    |   |      '-- uevent
    |   |-- scsi_device
    |   |-- scsi_generic
    |   |-- scsi_host
    |   |-- tty
    |   |-- usb
    |   |-- usb_device
    |   |-- usb_host
    |   '-- vc
    |-- devices
    |   |-- pci0000:00
    |   |   |-- 0000:00:00.0
    |   |   |-- 0000:00:07.0
    |   |   |-- 0000:00:07.1
    |   |   |-- 0000:00:07.2
    |   |   |-- 0000:00:07.3
    |   |-- platform
    |   |   |-- floppy.0
    |   |   |-- host0
    |   |   |-- i8042
    |   |-- pnp0
    |   '-- system
    |-- firmware
    |-- kernel
    |   '-- hotplug_seqnum
    |-- module
    |   |-- apm
    |   |-- autofs
    |   |-- cdrom
    |   |-- eisa_bus
    |   |-- i8042
    |   |-- ide_cd
    |   |-- ide_scsi
    |   |-- ieee1394
    |   |-- md_mod
    |   |-- ohci1394
    |   |-- parport
    |   |-- parport_pc
    |   |-- usb_storage
    |   |-- usbcore
    |   |   |-- parameters
    |   |   |-- refcnt
    |   |   '-- sections
    |   |-- virtual_root
    |   |   '-- parameters
    |   |       '-- force_probe
    |   '-- vmhgfs
    |       |-- refcnt
    |       '-- sections
    |           '-- _ _versions
    '-- power
        '-- state
```

在/sys/bus 的 pci 等子目录下，又会在分出 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 目录下包含许多对

/sys/devices 下文件的链接。如图 5.2 所示，这与设备、驱动、总线和类的现实状况是直接对应的，也正符合 Linux 2.6 内核的设备模型。

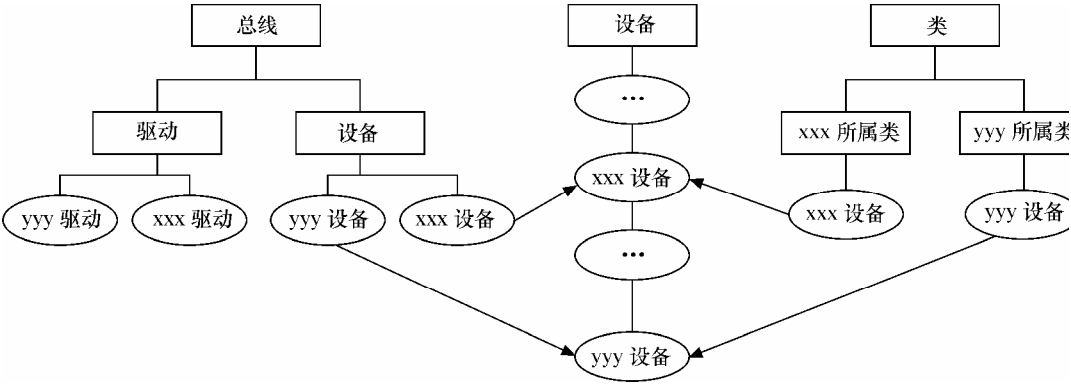

图 5.2　Linux 设备模型

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 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 5.3 也形象地表示了 Linux 驱动模型中设备、总线和类之间的关系。

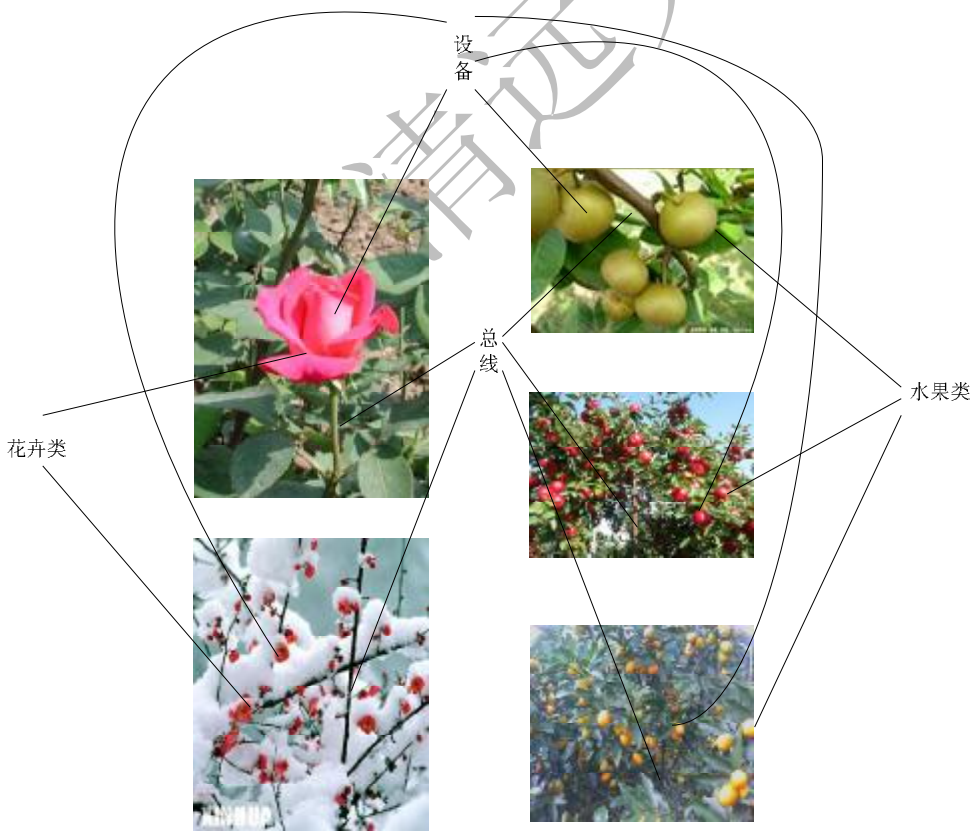

图 5.3　Linux 驱动模型中设备、总线和类之间的关系

大多数情况下，Linux 2.6 内核中的设备模型代码会处理好这些关系，内核中的总

线级和其他内核子系统会完成与设备模型的交互，这使得驱动工程师几乎不需要关心设备模型。但是，理解 Linux 设备模型的实现机制对驱动工程师仍然是大有裨益的，具体而言，内核将借助下文将介绍的 kobject、kset、subsystem、bus_type、device、device_driver、class、class_device、class_interface 等重量级数据结构来完成设备模型的架构。

2．kobject 内核对象

kobject 是 Linux 2.6 引入的设备管理机制，在内核中由 kobject 结构体表示，这个数据结构使所有设备在底层都具有统一的接口。kobject 提供了基本的对象管理能力，是构成 Linux 2.6 设备模型的核心结构，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。 kobject 结构体的定义如代码清单 5.6 所示。

代码清单 5.6 kobject 结构体

```
1  struct kobject
2  {
3    char *k_name;
4    char name[KOBJ_NAME_LEN]; //对象名称
5    struct kref kref; //对象引用计数
6    struct list_head entry; //用于挂接该 kobject 对象到 kset 链表
7    struct kobject *parent; //指向父对象的指针
8    struct kset *kset; //所属 kset 的指针
9    struct kobj_type *ktype; //指向对象类型描述符的指针
10   struct dentry *dentry; //sysfs 文件系统中与该对象对应的文件节点入口
11 };
```

内核通过 kobject 的 kref 成员实现对象引用计数管理，且提供两个函数 kobject_get()、kobject_put()分别用于增加和减少引用计数，当引用计数为 0 时，所有该对象使用的资源将被释放。

kobject 的 ktype 成员是一个指向 kobj_type 结构的指针，表示该对象的类型。如代码清单 5.7 所示，kobj_type 数据结构包含 3 个成员：用于释放 kobject 占用的资源的 release()函数、指向 sysfs 操作的 sysfs_ops 指针和 sysfs 文件系统默认属性列表。

代码清单 5.7 kobj_type 结构体

```
1  struct kobj_type
2  {
3   void (*release)(struct kobject *);//release 函数
4   struct sysfs_ops    * sysfs_ops;//属性操作
5   struct attribute    ** default_attrs;//默认属性
6  };
```

kobj_type 结构体种的 sysfs_ops 包括 store()和 show()两个成员函数，用于实现属性的读写，代码清单 5.8 给出了 sysfs_ops 结构体的定义。当从用户空间读取属性时，show()函数将被调用，该函数将指定属性值存入 buffer 中返回给用户，而 store()函数用于存储用户通过 buffer 传入的属性值。和 kobject 不同的是，属性在 sysfs 中呈现为一个文件，而 kobject 则呈现为 sysfs 中的目录。

代码清单 5.8 sysfs_ops 结构体

```
1 struct sysfs_ops
2 {
3   ssize_t (*show)(struct kobject *, struct attribute *,char *);
```

```
    4   ssize_t (*store)(struct kobject *,struct attribute *,const char *,
size_t);
    5 };
```

Linux 内核中提供一系列操作 kobject 的函数：

```
    void kobject_init(struct kobject * kobj);
```

该函数用于初始化 kobject，它设置 kobject 引用计数为 1，entry 域指向自身，其所属 kset 引用计数加 1。

```
    int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

该函数用于设置指定 kobject 的名称。

```
    void    kobject_cleanup(struct    kobject    *    kobj)    和    void
kobject_release(struct kref *kref);
```

该函数用于清除 kobject，当其引用计数为 0 时，释放对象占用的资源。

```
    struct kobject *kobject_get(struct kobject *kobj);
```

该函数用于将 kobj 对象的引用计数加 1，同时返回该对象的指针。

```
    void kobject_put(struct kobject * kobj);
```

该函数用于将 kobj 对象的引用计数减 1，如果引用计数降为 0，则调用 kobject_release()释放该 kobject 对象。

```
    int kobject_add(struct kobject * kobj);
```

该函数用于将 kobject 对象加入 Linux 设备层次，它会挂接该 kobject 对象到 kset 的 list 链中，增加父目录各级 kobject 的引用计数，在其 parent 指向的目录下创建文件节点，并启动该类型内核对象的 hotplug 函数。

```
    int kobject_register(struct kobject * kobj);
```

该函数用于注册 kobject，它会先调用 kobject_init()初始化 kobj，再调用 kobject_add()完成该内核对象的添加。

```
    void kobject_del(struct kobject * kobj);
```

这个函数是 kobject_add()的反函数，它从 Linux 设备层次(hierarchy)中删除 kobject 对象。

```
    void kobject_unregister(struct kobject * kobj);
```

这个函数是 kobject_register()的反函数，用于注销 kobject。与 kobject_register()相反，它首先调用 kobject_del()从设备层次中删除该对象，再调用 kobject_put()减少该对象的引用计数，如果引用计数降为 0，则释放该 kobject 对象。

3．kset 内核对象集合

kobject 通常通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合，在内核中用 kset 数据结构表示，其定义如代码清单 5.9 所示。

代码清单 5.9　kset 结构体

```
1 struct kset
2 {
3   struct subsystem    * subsys; //所在的 subsystem 的指针
4   struct kobj_type    * ktype; //指向该 kset 对象类型描述符的指针
```

```
    5   struct list_head    list; //用于连接该 kset 中所有 kobject 的链表头
    6   spinlock_t      list_lock;
    7   struct kobject      kobj; //嵌入的 kobject
    8   struct kset_uevent_ops  * uevent_ops;//事件操作集
    9 };
```

包含在 kset 中的所有 kobject 被组织成一个双向循环链表，list 即是该链表的头。ktype 成员指向一个 kobj_type 结构，被该 kset 中的所有 kobject 共享，表示这些对象的类型。kset 数据结构还内嵌了一个 kobject 对象（kobj 成员表示），所有属于这个 kset 的 kobject 对象的 parent 均指向这个内嵌的对象。此外，kset 还依赖于 kobj 维护引用计数， kset 的引用计数实际上就是内嵌的 kobject 对象的引用计数。

与 kobject 相似，Linux 提供一系列函数操作 kset。kset_init()完成指定 kset 的初始化，kset_get()和 kset_put()分别增加和减少 kset 对象的引用计数，kset_add()和 kset_del()函数分别实现将指定 keset 对象加入设备层次和从其中删除，kset_register()函数完成 kset 的注册，kset_unregister()函数则完成 kset 的注销。

kobject 被创建或删除时会产生事件（event），kobject 所属的 kset 将有机会过滤事件或为用户空间添加信息。每个 kset 能支持一些特定的事件变量，在热插拔事件发生时，kset 的成员函数可以设置一些事件变量，这些变量将被导出到用户空间。kset 的 uevent_ops 成员是执行该 kset 事件操作集 kset_uevent_ops 的指针，kset_uevent_ops 的定义如代码清单 5.10 所示。

代码清单 5.10 kset_uevent_ops 结构体

```
    1 struct kset_uevent_ops
    2 {
    3   int (*filter)(struct kset *kset, struct kobject *kobj);//事件过滤
    4   const char *(*name)(struct kset *kset, struct kobject *kobj);
    5   int (*uevent)(struct kset *kset, struct kobject *kobj, char **envp,
    6           int num_envp, char *buffer, int buffer_size);//环境变量设
置
    7 };
```

filter()函数用于过滤掉不需要导出到用户空间的事件，uevent()函数用于导出一些环境变量给用户的热插拔处理程序，各类设备导出的环境变量如下。

- PCI 设备：ACTION（“add”/“remove”）、PCI_ CLASS（16 进制的 PCI 类、子类、接口，如 c0310）、PCI_ ID（Vendor:Device，如 0123:4567）、PCI_SUBSYS_ID（子系统 Vendor: 子系统 Device，如 89ab:cdef）、PCI_ SLOT_ NAME（Bus:Slot.Func，如 00:07.2）。
- USB 设备：ACTION（“add”/“remove”）、DEVPATH（/sys/DEVPATH）、PRODUCT（idVendor/ idProduct/bcdDevice，如 46d/c281/108）、TYPE（bDeviceClass/bDeviceSubClass/bDeviceProtocol，如 9/0/0）、INTERFACE（bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol，如 3/1/1），如果内核配置了 usbfs 文件系统，还会导出 DEVFS（USB 驱动列表的位置，如/proc/bus/usb）和

DEVICE（USB 设备节点路径）。

- 网络设备：ACTION（“register”/“unregister”）、INTERFACE（接口名，如“eth0”）。
- 输入设备：ACTION（“add”/“remove”）、PRODUCT（idbus/idvendor/idproduct/idversion，如 1/46d/c281/108）、NAME（设备名，如“ALCOR STRONG MAN KBD HUB”）、PHYS（设备物理地址 ID，如 usb-00:07.2-2.3/input0）、EV（来自 evbit，如 120002）、KEY（来自 evbit，如 e080ffdf 1dfffff ffffffff fffffe）、LED（来自 ledbit，如 7）。
- IEEE1394 设备：ACTION（“add”/“remove”）、VENDOR_ID（24 位的 vendor id，如 123456）、GUID（64 位 ID）、SPECIFIER_ID（24 位协议标准拥有者 ID）、VERSION（协议标准版本 ID）。

用户空间的热插拔脚本根据传入给它的参数（如 PCI 的参数为热插拔程序路径、“pci”和 0）以及内核导出的环境变量采取相应的行动，如下面的脚本程序会在 PRODUCT 为“82d/100/0”的 USB 设备被插入时加载 visor 模块，在被拔出时卸载 visor 模块，如下所示。

```
if [ "$1"="usb" ]; then
    if [ "$PRODUCT"="82d/100/0" ]; then
        if[ "$ACTION" = "add" ]; then
            /sbin/modprobe visor
        else
            /sbin/rmmod visor
        fi
    fi
fi
```

4．subsystem 内核对象子系统

subsystem 是一系列 kset 的集合，它描述系统中某一类设备子系统，如 block_subsys 表示所有的块设备，对应于 sysfs 文件系统中的 block 目录。devices_subsys 对应于 sysfs 中的 devices 目录，描述系统中所有的设备。subsystem 由 struct subsystem 数据结构描述，其定义如代码清单 5.11 所示。

代码清单 5.11　subsystem 结构体

```
1 struct subsystem
2 {
3   struct kset kset; //内嵌的 kset 对象
4   struct rw_semaphore rwsem; //互斥访问信号量
5 };
```

每个 kset 必须属于某个 subsystem，通过设置 kset 结构中的 subsys 域指向指定的 subsystem 可以将一个 kset 加入到该 subsystem。所有挂接到同一 subsystem 的 kset 共享同一个 rwsem 信号量，用于同步访问 kset 中的链表。

内核也提供了一组类似于 kobect 和 kset 的操作 subsystem 的操作，如下所示：

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

5．Linux 设备模型组件

系统中的任一设备在设备模型中都由一个 device 对象描述，其对应的数据结构 struct device 定义如代码清单 5.12 所示。

代码清单 5.12　device 结构体

```
1  struct device
2  {
3   struct klist        klist_children; // 设备列表中的孩子列表
4   struct klist_node   knode_parent;// 兄弟节点
5   struct klist_node   knode_driver; // 驱动结点
6   struct klist_node   knode_bus; // 总线结点
7   struct device   * parent; // 指向父设备
8
9   struct kobject kobj;   // 内嵌一个 kobject 对象
10  char    bus_id[BUS_ID_SIZE];// 总线上的位置
11  struct device_attribute uevent_attr;
12
13  struct semaphore    sem;
14
15  struct bus_type * bus;  // 总线
16  struct device_driver *driver;// 使用的驱动
17  void        *driver_data;   // 驱动私有数据
18  void        *platform_data; //平台特定的数据
19  void        *firmware_data; //固件特定的数据(如 ACPI、BIOS 数据)
20  struct dev_pm_info  power;
21
22  u64     *dma_mask;  //dma 掩码
23  u64     coherent_dma_mask;
24  struct list_head    dma_pools;  // DMA 缓冲池
25  struct dma_coherent_mem *dma_mem;
26
27  void    (*release)(struct device * dev); // 释放设备方法
28 };
```

device 结构体用于描述设备相关的信息设备之间的层次关系，以及设备与总线、

驱动的关系。

内核提供了相应的函数用于操作 device 对象。其中 device_register()函数将一个新的 device 对象插入设备模型，并自动在/sys/devices 下创建一个对应的目录。device_unregister()完成相反的操作，注销设备对象。get_device()和 put_device()分别增加与减少设备对象的引用计数。通常 device 结构体不单独使用，而是包含在更大的结构体中，比如描述 PCI 设备的 struct pci_dev，其中的 dev 域就是一个 device 对象。

系统中的每个驱动程序由一个 device_driver 对象描述，对应的数据结构的定义如代码清单 5.13 所示。

代码清单 5.13　device_driver 结构体

```
1  struct device_driver
2  {
3   const char      * name;   //设备驱动程序的名称
4   struct bus_type     * bus; //总线
5
6   struct completion   unloaded;
7   struct kobject      kobj;//内嵌的 kobject 对象
8   struct klist        klist_devices;
9   struct klist_node   knode_bus;
10
11  struct module       * owner;
12
13  int (*probe)    (struct device * dev); //指向设备探测函数
14  int (*remove)   (struct device * dev); //指向设备移除函数
15  void    (*shutdown) (struct device * dev);
16  int (*suspend)  (struct device * dev, pm_message_t state);
17  int (*resume)   (struct device * dev);
18 };
```

与 device 结构体类似，device_driver 对象依靠内嵌的 kobject 对象实现引用计数管理和层次结构组织。内核提供类似的函数用于操作 device_driver 对象。如 get_driver()增加引用计数，driver_register()用于向设备模型插入新的 driver 对象，同时在 sysfs 文件系统中创建对应的目录。device_driver()结构体还包括几个函数，用于处理探测、移除和电源管理事件。

系统中总线由 struct bus_type 描述，其定义如代码清单 5.14 所示。

代码清单 5.14　bus_type 结构体

```
1  struct bus_type
2  {
3   const char      * name;//总线类型的名称
4
5   struct subsystem    subsys;//与该总线相关的 subsystem
```

```
6   struct kset     drivers;//所有与该总线相关的驱动程序集合
7   struct kset     devices;//所有挂接在该总线上的设备集合
8   struct klist        klist_devices;
9   struct klist        klist_drivers;
10
11  struct bus_attribute    * bus_attrs;//总线属性
12  struct device_attribute * dev_attrs;//设备属性
13  struct driver_attribute * drv_attrs; //驱动程序属性
14
15  int     (*match)(struct device * dev, struct device_driver * drv);
16  int     (*uevent)(struct device *dev, char **envp,
17              int num_envp, char *buffer, int buffer_size);//事件
18  int     (*probe)(struct device * dev);
19  int     (*remove)(struct device * dev);
20  void        (*shutdown)(struct device * dev);
21  int     (*suspend)(struct device * dev, pm_message_t state);
22  int     (*resume)(struct device * dev);
23 };
```

每个 bus_type 对象都内嵌一个 subsystem 对象，bus_subsys 对象管理系统中所有总线类型的 subsystem 对象。每个 bus_type 对象都对应/sys/bus 目录下的一个子目录，如 PCI 总线类型对应于/sys/bus/pci。在每个这样的目录下都存在两个子目录：devices 和 drivers（分别对应于 bus_type 结构中的 devices 和 drivers 域）。其中 devices 子目录描述连接在该总线上的所有设备，而 drivers 目录则描述与该总线关联的所有驱动程序。与 device_driver 对象类似，bus_type 结构还包含几个函数处理热插拔、即插即拔和电源管理事件的函数。

系统中的设备类由 struct class 描述，表示某一类设备。所有的 class 对象都属于 class_subsys 子系统，对应于 sysfs 文件系统中的/sys/class 目录。代码清单 5.15 给出了 class 结构体的定义。

代码清单 5.15　class 结构体

```
1  struct class
2  {
3   const char      * name;  //类名
4   struct module       * owner;
5
6   struct subsystem    subsys; //对应的 subsystem
7   struct list_head    children; //class_device 链表
8   struct list_head    interfaces; //class_interface 链表
9   struct semaphore    sem;    //children 和 interfaces 链表锁
10
11  struct class_attribute      * class_attrs;//类属性
```

```
12  struct class_device_attribute  * class_dev_attrs;//类设备属性
13
14  int (*uevent)(struct class_device *dev, char **envp,
15            int num_envp, char *buffer, int buffer_size);//事件
16
17  void    (*release)(struct class_device *dev);
18  void    (*class_release)(struct class *class);
19 };
```

每个 class 对象包括一个 class_device 链表，每个 class_device 对象表示一个逻辑设备，并通过 struct class_device 中的 dev 成员（一个指向 struct device 的指针）关联一个物理设备。这样，一个逻辑设备总是对应于一个物理设备，但是一个物理设备却可能对应于多个逻辑设备，代码清单 5.16 给出了 class_device 的定义。此外，class 结构中还包括用于处理热插拔、即插即拔和电源管理事件的函数，这与 bus_type 对象相似。

代码清单 5.16　class_device 结构体

```
1  struct class_device
2  {
3   struct list_head    node;
4
5   struct kobject      kobj;//内嵌的 kobject
6   struct class        * class;    //所属的类
7   dev_t           devt;       // dev_t
8   struct class_device_attribute *devt_attr;
9   struct class_device_attribute uevent_attr;
10  struct device       * dev;  //如果存在，创建到/sys/devices 相应入口的
符号链接
11  void            * class_data;   //私有数据
12  struct class_device *parent;    //父设备
13
14  void    (*release)(struct class_device *dev);
15  int (*uevent)(struct class_device *dev, char **envp,
16            int num_envp, char *buffer, int buffer_size);
17  char    class_id[BUS_ID_SIZE]; //类标识
18 };
```

下面两个函数用于注册和注销 class。

```
int class_register(struct class * cls);
void class_unregister(struct class * cls);
```

下面两个函数用于注册和注销 class_device。

```
int class_device_register(struct class_device *class_dev);
void class_device_unregister(struct class_device *class_dev);
```

除了 class、class_device 结构体外，还存在一个 class_interface 结构体，当设备加入或离开类时，将引发 class_interface 中的成员函数被调用，class_interface 的定义如代码清单 5.17 所示。

代码清单 5.17　class_interface 结构体

```
1  struct class_interface
```

```
2 {
3   struct list_head    node;
4   struct class        *class;//对应的 class
5   int (*add)(struct class_device *, struct class_interface *);//设
备加入时触发
6   void (*remove)(struct class_device *, struct class_interface *);//
设备移出时触发
7 };
```

下面两个函数用于注册和注销 class_interface。

```
int class_interface_register(struct class_interface *class_intf);
void class_interface_unregister(struct class_interface *class_intf);
```

6．属性

在 bus、device、driver 和 class 层次上都分别定义了其属性结构体，包括 bus_attribute、driver_attribute、class_attribute、class_device_attribute，这几个结构体的定义在本质是完全相同的，如代码清单 5.18 所示。

代码清单 5.18　bus_attribute/driver_attribute/class_attribute/class_device_attribute 结构体

```
1  /* 总线属性 */
2  struct bus_attribute
3  {
4   struct attribute    attr;
5   ssize_t (*show)(struct bus_type *, char * buf);
6   ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
7  };
8  /* 驱动属性 */
9  struct driver_attribute
10 {
11  struct attribute    attr;
12  ssize_t (*show)(struct device_driver *, char * buf);
13  ssize_t (*store)(struct device_driver *, const char * buf, size_t
count);
14 };
15 /* 类属性 */
16 struct class_attribute
17 {
18  struct attribute    attr;
19  ssize_t (*show)(struct class *, char * buf);
20  ssize_t (*store)(struct class *, const char * buf, size_t count);
21 };
22 /* 类设备属性 */
23 struct class_device_attribute
24 {
25  struct attribute    attr;
26  ssize_t (*show)(struct class_device *, char * buf);
27  ssize_t (*store)(struct class_device *, const char * buf, size_t
count);
28 };
```

下面一组宏分别用于创建和初始化 bus_attribute、driver_attribute、class_attribute、class_device_ attribute。

```
BUS_ATTR(_name,_mode,_show,_store)
DRIVER_ATTR(_name,_mode,_show,_store)
CLASS_ATTR(_name,_mode,_show,_store)
CLASS_DEVICE_ATTR(_name,_mode,_show,_store)
```

下面一组函数分别用于添加和删除 bus、driver、class、class_device 属性。

```
int bus_create_file(struct bus_type * bus, struct bus_attribute * attr);
void bus_remove_file(struct bus_type * bus, struct bus_attribute *
attr);

int device_create_file(struct device * dev, struct device_attribute *
attr);
void device_remove_file(struct device * dev, struct device_attribute
* attr);

int class_create_file(struct class * cls, const struct class_attribute
* attr);
void class_remove_file(struct class * cls, const struct class_attribute
* attr);

int class_device_create_file(struct class_device * class_dev,
```

```
                const struct class_device_attribute * attr);
void class_device_remove_file(struct class_device * class_dev,
                const struct class_device_attribute * attr);
```

xxx_create_file()函数中会调用 sysfs_create_file()，而 xxx_remove_file()函数中会调用 sysfs_ remove_file()函数，xxx_create_file()会创建对应的 sysfs 文件节点，而 xxx_remove_file()会删除对应的 xxx 文件节点。

5.4.3 udev 的组成

udev 的主页位于 http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html，上面包含了 udev 的详细介绍，从 http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/ 上可以下载最新的 udev 包。udev 的设计目标如下。

- 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。
- 提供 LSB 标准名称。
- 如果需要，可提供固定的名称。

为了提供这些功能，udev 以 3 个分割的子计划发展： namedev、libsysfs 和 udev。namedev 为设备命名子系统，libsysfs 提供访问 sysfs 文件系统从中获取信息的标准接口，udev 提供/dev 设备节点文件的动态创建和删除策略。udev 程序承担与 namedev 和 libsysfs 库交互的任务，当/sbin /hotplug 程序被内核调用时，udev 将被运行。udev 的工作过程如下。

（1） 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新的记录并导出一些设备特定的信息及所发生的事件。

（2） udev 获取内核导出的信息，它调用 namedev 决定应该给该设备指定的名称，如果是新插入设备，udev 将调用 libsysfs 决定应该为该设备的设备文件指定的主/次设备号，并用分析获得的设备名称和主/次设备号创建/dev 中的设备文件；如果是设备移除，则之前已经被创建的/dev 文件将被删除。

在 namedev 中使用 5 个步骤来决定指定设备的命名。

（1）标签（label）/序号（serial）：这一步检查设备是否有惟一的识别记号，例如 USB 设备有惟一的 USB 序号，SCSI 有惟一的 UUID。如果 namedev 找到与这种惟一编号相对应的规则，它将使用该规则提供的名称。

（2）设备总线号：这一步会检查总线设备编号，对于不可热插拔的环境，这一步足以辨别设备。例如，PCI 总线编号在系统的使用期间内很少变更。如果 namedev 找到相对应的规则，规则中的名称就会被使用。

（3）总线上的拓扑：当设备在总线上的位置匹配用户指定的规则时，就会使用该规则指定的名称。

（4）替换名称：当内核提供的名称匹配指定的替代字符串时，就会使用替代字符串指定的名称。

（5）内核提供的名称：如果以前的几个步骤都没有被提供，缺省的内核将被指定给该设备。

代码清单 5.19 给出了一个 namedev 命名规则的例子，第 2、4 行定义的是符合第 1 步的规则，第 6、8 行定义的是符合第 2 步的规则，第 11、14 行定义的是符合第 3 步的规则，第 16 行定义的是符合第 4 步的规则。

代码清单 5.19　namedev 命名规则

```
1  # USB Epson printer to be called lp_epson
2  LABEL, BUS="usb", serial="HXOLL0012202323480", NAME="lp_epson"
```

```
3  # USB HP printer to be called lp_hp,
4  LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"
5  # sound card with PCI bus id 00:0b.0 to be the first sound card
6  NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"
7  # sound card with PCI bus id 00:07.1 to be the second sound card
8  NUMBER, BUS="pci", id="00:07.1", NAME="dsp1"
9  # USB mouse plugged into the third port of the first hub to be
10 # called mouse0
11 TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"
12 # USB tablet plugged into the second port of the second hub to be
13 # called mouse1
14 TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"
15 # ttyUSB1 should always be called visor
16 REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

5.4.4 udev 规则文件

udev 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（用于匹配行为）、KERNEL（用于匹配内核设备名）、BUS（用于匹配总线类型）、SYSFS（用于匹配从 sysfs 得到的信息，比如 label、vendor、USB 序列号）、SUBSYSTEM（匹配子系统名）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）等。

例如如下规则：

```
    SUBSYSTEM=="net", ACTION=="add", SYSFS{address}=="00:0d:87:f6:59:f3",
IMPORT="/sbin/ rename_netiface %k eth0"
```

其中的匹配部分有 3 项，分别是 SUBSYSTEM、ACTION 和 SYSFS。而赋值部分有一项，是 IMPORT。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加入这个硬件，且这个硬件在 sysfs 文件系统中的“address”信息等于“00:0d:87:f6:59:f3”时，对这个硬件在 udev 层次施行的动作是调用外部程序/sbin/rename_netiface，并给该程序传递两个参数，一个是“%k”，代表内核对该新设备定义的名称，另一个是“eth0”。

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，一个被称为/dev/usb/lp0，则另外一个便是/dev/usb/lp1。但是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，映射关系会因为设备发现的顺序、打印机本身关闭等原因而不确定。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信息的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：

```
    BUS="usb",        SYSFS{serial}="HXOLL0012202323480",        NAME="lp_epson",
SYMLINK="printers/ epson _stylus"
```

该规则中的匹配项目有 BUS 和 SYSFS，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，[x-y]是访问定义。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 udevinfo 工具查找规则文件可以利用的信息，如运行“udevinfo -a -p /sys/ block/sda”命令将得到以下信息：

```
    Udevinfo 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 '/block/sda':
        KERNEL=="sda"
        SUBSYSTEM=="block"
        DRIVER==""
        ATTR{stat}==""    1689     3169    85746    24000     2017     2095
32896    47292
        0    23188    71292"
        ATTR{size}=="6291456"
        ATTR{removable}=="0"
        ATTR{range}=="16"
        ATTR{dev}=="8:0"

      looking               at               parent               device
'/devices/platform/host0/target0:0:0/0:0:0:0':
        KERNELS=="0:0:0:0"
        SUBSYSTEMS=="scsi"
        DRIVERS=="sd"
        ATTRS{ioerr_cnt}=="0x5"
        ATTRS{iodone_cnt}=="0xe86"
        ATTRS{iorequest_cnt}=="0xe86"
        ATTRS{iocounterbits}=="32"
        ATTRS{timeout}=="30"
        ATTRS{state}=="running"
        ATTRS{rev}=="1.0 "
        ATTRS{model}=="VMware Virtual S"
        ATTRS{vendor}=="VMware, "
        ATTRS{scsi_level}=="3"
        ATTRS{type}=="0"
        ATTRS{queue_type}=="none"
        ATTRS{queue_depth}=="3"
        ATTRS{device_blocked}=="0"

      looking at parent device '/devices/platform/host0/target0:0:0':
        KERNELS=="target0:0:0"
        SUBSYSTEMS==""
        DRIVERS==""

      looking at parent device '/devices/platform/host0':
        KERNELS=="host0"
```

```
    SUBSYSTEMS==""
    DRIVERS==""

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

5.4.5 创建和配置 udev

udev 包括 udev、udevcontrol、udevd、udevsend、udevmonitor、udevsettle、udevstart、udevinfo、udevtest 和 udevtrigger 等处于用户空间的程序。在嵌入式系统中，只需要 udevd 和 udevstart 就能使 udev 工作。我们可以按照下面的步骤来生成和配置 udev。

（1）从 http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/下载 udev 程序，如笔者下载的是 udev-114.tar.gz。

（2）运行“tar zxvf udev-114.tar.gz”命令解压缩 udev 程序包。

（3）运行 make 编译 udev，当前目录下会生成 test-udev、udevcontrol、udevd、udevinfo、udevmonitor、udevsettle、udevstart、udevtest 和 udevtrigger 共 9 个工具程序。

（4）把第（3）步生成的工具程序复制到/sbin 目录，同时把解压缩 udev-114.tar.gz 后获得的 etc 目录下的 udev 目录复制到系统的/etc 下，etc/udev 下包含 udev.conf 配置文件、rules.d 目录（该目录中的文件给出了设备规则）等。

（5）编写完成启动、停止、重新启动等工作的 udev 脚本，代码清单 5.20 给出了一个范例。

代码清单 5.20 udev 运行脚本范例

```
1  #!/bin/sh -e
2  # udev 初始脚本
3
4  # 检查包是否已经被安装
5  [ -x /sbin/udevd ] || exit 0
6
7  case "$1" in
8     start)
9   # 需要 2.6.15 中引入的 uevent 的支持，如果系统中没有，使用静态/dev
10  if [ ! -f /sys/class/mem/null/uevent ]; then
11     if mountpoint -q /dev; then
12      umount -l /dev/.static/dev
13      umount -l /dev
14     fi
15     exit 1
16  fi
17
18  if ! mountpoint -q /dev; then
19     # initramfs 没有挂载/dev，因此，这里需要完成
20     mount -n --bind /dev /etc/udev
21     mount -n -t tmpfs -o mode=0755 udev /dev
22     mkdir -m 0700 -p /dev/.static/dev
23     mount -n --move /etc/udev /dev/.static/dev
24  fi
25
26  # 复制默认的设备树
```

```
27  cp -a -f /lib/udev/devices/* /dev
28
29  # 现在全部通过 netlink 进行，所以之前的热插拔程序不再使用
30  if [ -e /proc/sys/kernel/hotplug ]; then
31      echo "" > /proc/sys/kernel/hotplug
32  fi
33
34  # 开始 udevd
35  log_begin_msg "Starting kernel event manager..."
36 if start-stop-daemon --start --quiet --exec /sbin/udevd -- --daemon;
then
37      log_end_msg 0
38  else
39      log_end_msg $?
40  fi
41
42  # 打开 udevtrigger 的事务日志
43  /sbin/udevmonitor -e >/dev/.udev.log &
44  UDEV_MONITOR_PID=$!
45
46  # 弥补丢失的事件
47  log_begin_msg "Loading hardware drivers..."
48  /sbin/udevtrigger
49  if /sbin/udevsettle; then
50      log_end_msg 0
51  else
52      log_end_msg $?
53  fi
54
55  # 杀死 udevmonitor
56  kill $UDEV_MONITOR_PID
57  ;;
58    stop)
59  log_begin_msg "Stopping kernel event manager..."
60  if start-stop-daemon --stop --quiet --oknodo --exec /sbin/udevd;
then
61      log_end_msg 0
62  else
63      log_end_msg $?
64  fi
```

```
    65  umount -l /dev/.static/dev
    66  umount -l /dev
    67  ;;
    68    restart)
    69  cp -au /lib/udev/devices/* /dev
    70
    71  log_begin_msg "Loading additional hardware drivers..."
    72  /sbin/udevtrigger
    73  if /sbin/udevsettle; then
    74     log_end_msg 0
    75  else
    76     log_end_msg $?
    77  fi
    78  ;;
    79    reload|force-reload)
    80  log_begin_msg "Reloading kernel event manager..."
    81  if start-stop-daemon --stop --signal 1 --exec /sbin/udevd; then
    82     log_end_msg 0
    83  else
    84     log_end_msg $?
    85  fi
    86  ;;
    87 *)
    88  echo                   "Usage:                   /etc/init.d/udev
{start|stop|restart|reload|force-reload}"
    89  exit 1
    90  ;;
    91 esac
    92
    93 exit 0
```

如果已经知道了设备对应的 sysfs 路径，则可以使用 udevtest 工具测试 udev 对该设备所采取的行动，这可以帮助进行 udev 规则的调试。如运行“udevtest /sys/class/tty/ttys0”命令的结果为：

```
    This program is for debugging only, it does not run any program,
    specified by a RUN key. It may show incorrect results, because
    some values may be different, or not available at a simulation run.

    parse_file: reading '/etc/udev/rules.d/05-udev-early.rules' as rules
file
    parse_file: reading '/etc/udev/rules.d/60-persistent-input.rules' as
```

```
rules file
    parse_file: reading '/etc/udev/rules.d/60-persistent-storage.rules'
as rules file
    parse_file: reading '/etc/udev/rules.d/95-udev-late.rules' as rules
file
    main: looking at device '/class/tty/ttys0' from subsystem 'tty'
    udev_rules_get_name: no node name set, will use kernel name ''
    udev_db_get_device: found a symlink as db file
    udev_device_event: device '/class/tty/ttys0' already in database,
cleanup
    udev_node_add: creating device node '/dev/ttys0', major=3, minor=48,
mode=0660, uid=0, gid=0
    main: run: 'socket:/org/kernel/udev/monitor'
```

从中可以看出 udev 将为与/sys/class/tty/ttys0 对应的设备使用内核中的名称通过 udev_node_ add 创建/dev/ttys0 设备文件，主、次设备号分别为 3 和 48。

5.5 总结

Linux 系统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，能看到的只有设备对应的文件，因此文件编程即是用户空间的设备编程。

Linux 系统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 设备和 Linux 2.6 设备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 设备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，udev 可以利用 sysfs 中记录的信息定义规则并提取主次设备号动态创建/dev 设备文件节点。

- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

  http://www.farsight.com.cn/courses/TS-LinuxApp.htm

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm