《LINUX 设备驱动开发详解》

作者：华清远见

第 4 章　Linux 内核模块

本章简介

Linux 设备驱动会以内核模块的形式出现，因此，学会编写 Linux 内核模块编程是学习 Linux 设备驱动的先决条件。

4.1～4.2 节讲解了 Linux 内核模块的概念和结构，4.3～4.8 节对 Linux 内核模块的各个组成部分进行详细讲解，4.1～4.2 节与 4.3～4.8 节是整体与部分的关系。

4.9 节讲解了独立存在的 Linux 内核模块的 Makefile 文件编写方法和模块的编译方法。

4.1 Linux 内核模块简介

Linux 内核的整体结构非常庞大，其包含的组件也非常多。我们怎样把需要的部分都包含在内核中呢？

一种方法是把所有需要的功能都编译到 Linux 内核。这会导致两个问题，一是生成的内核会很大，二是如果我们要在现有的内核中新增或删除功能，将不得不重新编译内核。

有没有一种机制使得编译出的内核本身并不需要包含所有功能，而在这些功能需要被使用的时候，其对应的代码可被动态地加载到内核中呢？

Linux 提供了这样的一种机制，这种机制被称为模块（Module），可以实现以上效果。模块具有以下特点。

- 模块本身不被编译入内核映像，从而控制了内核的大小。
- 模块一旦被加载，它就和内核中的其他部分完全一样。

为了使读者对模块建立初步的感性认识，我们先来看一个最简单的内核模块“Hello World”，如代码清单 4.1 所示。

代码清单 4.1　一个最简单的 Linux 内核模块

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4  static int hello_init(void)
5  {
6  printk(KERN_ALERT " Hello World enter\n");
7  return 0;
8  }
9  static void hello_exit(void)
10 {
11 printk(KERN_ALERT " Hello World exit\n ");
12 }
13 module_init(hello_init);
14 module_exit(hello_exit);
15
16 MODULE_AUTHOR("Song Baohua");
17 MODULE_DESCRIPTION("A simple Hello World Module");
18 MODULE_ALIAS("a simplest module");
```

这个最简单的内核模块只包含内核模块加载函数、卸载函数和对 Dual BSD/GPL 许可权限的声明以及一些描述信息。编译它会产生 hello.ko 目标文件，通过“insmod ./hello.ko”命令可以加载它，通过“rmmod hello”命令可以卸载它，加载时输出“Hello World enter”，卸载时输出“Hello World exit”。

内核模块中用于输出的函数是内核空间的 printk()而非用户空间的 printf()，printk()的用法和 printf()相似，但前者可定义输出级别。printk()可作为一种最基本的内核调试手段。

在 Linux 系统中，使用 lsmod 命令可以获得系统中加载了的所有模块以及模块间的依赖关系，例如：

```
[root@localhost driver_study]# lsmod
Module                  Size  Used by
hello                   1568   0
ohci1394                32716  0
ide_scsi               16708   0
ide_cd                  39392  0
cdrom                   36960  1 ide_cd
```

lsmod 命令实际上读取并分析/proc/modules 文件，与上述 lsmod 命令结果对应的/proc/modules 文件如下：

```
[root@localhost driver_study]# cat /proc/modules
hello 1568 0 - Live 0xc8859000
ohci1394 32716 0 - Live 0xc88c8000
ieee1394 94420 1 ohci1394, Live 0xc8840000
ide_scsi 16708 0 - Live 0xc883a000
ide_cd 39392 0 - Live 0xc882f000
cdrom 36960 1 ide_cd, Live 0xc8876000
```

内核中已加载模块的信息也存在于/sys/module 目录下，加载 hello.ko 后，内核中将包含/sys/module/hello 目录，该目录下又包含一个 refcnt 文件和一个 sections 目录，在/sys/module/hello 目录下运行“tree –a”得到如下目录树：

```
[root@localhost hello]# tree -a
.
|-- refcnt
'-- sections
     |-- .bss
     |-- .data
     |-- .gnu.linkonce.this_module
     |-- .rodata
     |-- .rodata.str1.1
     |-- .strtab
     |-- .symtab
     |-- .text
     '-- __versions
```

modprobe 命令比 insmod 命令要强大，它在加载某模块时会同时加载该模块所依赖的其他模块。使用 modprobe 命令加载的模块若以“modprobe -r filename”的方式卸载将同时卸载其依赖的模块。

使用 modinfo <模块名>命令可以获得模块的信息，包括模块的作者、模块的说明、模块所支持的参数以及 vermagic，如下所示：

```
[root@localhost driver_study]# modinfo hello.ko
filename:      hello.ko
license:       Dual BSD/GPL
author:        Song Baohua
description:    A simple Hello World Module
alias:          a simplest module
vermagic:      2.6.15.5 686 gcc-3.2
depends:
```

4.2 Linux 内核模块的程序结构

一个 Linux 内核模块主要由以下几个部分组成。

- 模块加载函数（必须）。

当通过 insmod 或 modprobe 命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块的相关初始化工作。

- 模块卸载函数（必须）。

当通过 rmmod 命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块加载函数相反的功能。

- 模块许可证声明（必须）。

模块许可证（LICENSE）声明描述内核模块的许可权限，如果不声明 LICENSE，模块被加载时，将收到内核被污染 （kernel tainted）的警告。

在 Linux 2.6 内核中，可接受的 LICENSE 包括“GPL”、“GPL v2”、“GPL and additional rights”、“Dual BSD/GPL”、“Dual MPL/GPL”和“Proprietary”。

大多数情况下，内核模块应遵循 GPL 兼容许可权。Linux 2.6 内核模块最常见的是以 MODULE_LICENSE("Dual BSD/GPL")语句声明模块采用 BSD/GPL 双 LICENSE。

- 模块参数（可选）。

模块参数是模块被加载的时候可以被传递给它的值，它本身对应模块内部的全局变量。

- 模块导出符号（可选）。

内核模块可以导出符号（symbol，对应于函数或变量），这样其他模块可以使用本模块中的变量或函数。

- 模块作者等信息声明（可选）。

4.3 模块加载函数

Linux 内核模块加载函数一般以_ _init 标识声明，典型的模块加载函数的形式如代码清单 4.2 所示。

代码清单 4.2 内核模块加载函数

```
1  static int _ _init initialization_function(void)
2  {
3  /* 初始化代码 */
4  }
5  module_init(initialization_function);
```

模块加载函数必须以“module_init(函数名)”的形式被指定。它返回整型值，若初始化成功，应返回 0。而在初始化失败时，应该返回错误编码。在 Linux 内核里，错误编码是一个负值，在<linux/errno.h>中定义，包含-ENODEV、-ENOMEM 之类的符号值。返回相应的错误编码是种非常好的习惯，因为只有这样，用户程序才可以利用 perror 等方法把它们转换成有意义的错误信息字符串。

在 Linux 2.6 内核中，可以使用 request_module(const char *fmt, …)函数加载内核模块，驱动开发人员可以通过调用

```
request_module(module_name);
```

或

```
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

来加载其他内核模块。

在 Linux 内核中，所有标识为_ _init 的函数在连接的时候都放在.init.text 这个区段内，此外，所有的_ _init 函数在区段.initcall.init 中还保存了一份函数指针，在初始化时内核会通过这些函数指针调用这些_ _init 函数，并在初始化完成后释放 init 区段（包括.init.text，.initcall.init 等）。

4.4 模块卸载函数

Linux 内核模块卸载函数一般以_ _exit 标识声明，典型的模块卸载函数的形式如代码清单 4.3 所示。

代码清单 4.3 内核模块卸载函数

```
1  static void _ _exit cleanup_function(void)
2  {
3  /* 释放代码 */
4  }
5  module_exit(cleanup_function);
```

模块卸载函数在模块卸载的时候执行，不返回任何值，必须以“module_exit(函数名)”的形式来指定。通常来说，模块卸载函数要完成与模块加载函数相反的功能，如下所示。

- 若模块加载函数注册了 XXX，则模块卸载函数应该注销 XXX。
- 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。
- 若模块加载函数申请了硬件资源（中断、DMA 通道、I/O 端口和 I/O 内存等）的占用，则模块卸载函数应释放这些硬件资源。
- 若模块加载函数开启了硬件，则卸载函数中一般要关闭硬件。

和_ _init 一样，_ _exit 也可以使对应函数在运行完成后自动回收内存。实际上，_ _init 和_ _exit 都是宏，其定义分别为：

```
#define _ _init _ _attribute_ _ ((_ _section_ _ (".init.text")))
```

和

```
#ifdef MODULE
#define _ _exit _ _attribute_ _ ((_ _section_ _(".exit.text")))
#else
#define _ _exit _ _attribute_used_ _ _ _attribute_ _ ((_ _section_ _(".exit.text")))
#endif
```

数据也可以被定义为_initdata 和_exitdata，这两个宏分别为：

```
#define _ _initdata_ _attribute_ _ ((_ _section_ _ (".init.data")))
```

和

```
#define _ _exitdata_ _attribute_ _ ((_ _section_ _(".exit.data")))
```

4.5 模块参数

我们可以用“module_param(参数名,参数类型,参数读/写权限)”为模块定义一个参数，例如下列代码定义了一个整型参数和一个字符指针参数：

```
static char *book_name = "深入浅出 Linux 设备驱动";
static int num = 4000;
module_param(num, int, S_IRUGO);
module_param(book_name, charp, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为“insmode（或 modprobe）模块名 参数名=参数值”，如果不传递，参数将使用模块内定义的默认值。

参数类型可以是 byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool 或 invbool（布尔的反），在模块被编译时会将 module_param 中声明的类型与变量定义的类型进行比较，判断是否一致。

模块被加载后，在/sys/module/目录下将出现以此模块名命名的目录。当“参数读/写权限”为 0 时，表示此参数不存在 sysfs 文件系统下对应的文件节点，如果此模块存在“参数读/写权限”不为 0 的命令行参数，在此模块的目录下还将出现 parameters 目录，包含一系列以参数名命名的文件节点，这些文件的权限值就是传入 module_param()的“参数读/写权限”，而文件的内容为参数的值。

除此之外，模块也可以拥有参数数组，形式为“module_param_array(数组名,数组类型,数组长,参数读/写权限)”。从 2.6.0～2.6.10 版本，需将数组长变量名赋给“数组长”，从 2.6.10 版本开始，需将数组长变量的指针赋给“数组长”，当不需要保存实际输入的数组元素个数时，可以设置“数组长”为 NULL。

运行 insmod 或 modprobe 命令时，应使用逗号分隔输入的数组元素。

现在我们定义一个包含两个参数的模块（如代码清单 4.4 所示），并观察模块加载时被传递参数和不传

递参数时的输出。

代码清单 4.4　带参数的内核模块

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4
5  static char *book_name = "dissecting Linux Device Driver";
6  static int num = 4000;
7
8  static int book_init(void)
9  {
10  printk(KERN_INFO " book name:%s\n",book_name);
11  printk(KERN_INFO " book num:%d\n",num);
12  return 0;
13 }
14 static void book_exit(void)
15 {
16  printk(KERN_ALERT " Book module exit\n ");
17 }
18 module_init(book_init);
19 module_exit(book_exit);
20 module_param(num, int, S_IRUGO);
21 module_param(book_name, charp, S_IRUGO);
22
23 MODULE_AUTHOR("Song Baohua, author@linuxdriver.cn");
24 MODULE_DESCRIPTION("A simple Module for testing module params");
25 MODULE_VERSION("V1.0");
```

对上述模块运行“insmod book.ko”命令加载，相应输出都为模块内的默认值，通过查看“/var/log/messages”日志文件可以看到内核的输出，如下所示：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:03:10 localhost kernel:  <6> book name:dissecting Linux Device Driver
Jul  2 01:03:10 localhost kernel:  book num:4000
```

当用户运行“insmod book.ko book_name='GoodBook' num=5000”命令时，输出的是用户传递的参数，如下所示：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:06:21 localhost kernel:  <6> book name:GoodBook
Jul  2 01:06:21 localhost kernel:  book num:5000
```

4.6 导出符号

Linux 2.6 的“/proc/kallsyms”文件对应着内核符号表，它记录了符号以及符号所在的内存地址。

模块可以使用如下宏导出符号到内核符号表：

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出的符号将可以被其他模块使用，使用前声明一下即可。EXPORT_SYMBOL_GPL()只适用于包含 GPL 许可权的模块。代码清单 4.5 给出了一个导出整数加、减运算函数符号的内核模块的例子（这些导出符号没有实际意义，只是为了演示）。

代码清单 4.5　内核模块中的符号导出

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4
```

```
5  int add_integar(int a,int b)
6  {
7   return a+b;
8  }
9
10 int sub_integar(int a,int b)
11 {
12  return a-b;
13 }
14
15 EXPORT_SYMBOL(add_integar);
16 EXPORT_SYMBOL(sub_integar);
```

从“/proc/kallsyms”文件中找出 add_integar、sub_integar 相关信息：

```
[root@localhost driver_study]# cat /proc/kallsyms | grep integar
c886f050 r _ _kcrctab_add_integar       [export]
c886f058 r _ _kstrtab_add_integar       [export]
c886f070 r _ _ksymtab_add_integar       [export]
c886f054 r _ _kcrctab_sub_integar       [export]
c886f064 r _ _kstrtab_sub_integar       [export]
c886f078 r _ _ksymtab_sub_integar       [export]
c886f000 T add_integar  [export]
c886f00b T sub_integar  [export]
13db98c9 a _ _crc_sub_integar    [export]
e1626dee a _ _crc_add_integar    [export]
```

4.7 模块声明与描述

在 Linux 内核模块中，我们可以用 MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_ VERSION、MODULE_DEVICE_TABLE、MODULE_ALIAS 分别声明模块的作者、描述、版本、设备表和别名，例如：

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

对于 USB、PCI 等设备驱动，通常会创建一个 MODULE_DEVICE_TABLE，如代码清单 4.6 所示。

代码清单 4.6　驱动所支持的设备列表

```
1 /* 对应此驱动的设备表 */
2 static struct usb_device_id skel_table [] = {
3 { USB_DEVICE(USB_SKEL_VENDOR_ID,
4   USB_SKEL_PRODUCT_ID) },
5  { } /* 表结束 */
6 };
7
8 MODULE_DEVICE_TABLE (usb, skel_table);
```

此时，并不需要读者理解 MODULE_DEVICE_TABLE 的作用，后续相关章节会有详细介绍。

4.8 模块的使用计数

Linux 2.4 内核中，模块自身通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT 宏来管理自己

被使用的计数。

Linux 2.6 内核提供了模块计数管理接口 try_module_get(&module)和 module_put (&module)，从而取代 Linux 2.4 内核中的模块使用计数管理宏。模块的使用计数一般不必由模块自身管理，而且模块计数管理还考虑了 SMP 与 PREEMPT 机制的影响。

```
int try_module_get(struct module *module);
```

该函数用于增加模块使用计数；若返回为 0，表示调用失败，希望使用的模块没有被加载或正在被卸载中。

```
void module_put(struct module *module);
```

该函数用于减少模块使用计数。

try_module_get ()与 module_put()的引入与使用与 Linux 2.6 内核下的设备模型密切相关。Linux 2.6 内核为不同类型的设备定义了 struct module *owner 域，用来指向管理此设备的模块。当开始使用某个设备时，内核使用 try_module_get(dev->owner)去增加管理此设备的 owner 模块的使用计数；当不再使用此设备时，内核使用 module_put(dev->owner)减少对管理此设备的 owner 模块的使用计数。这样，当设备在使用时，管理此设备的模块将不能被卸载。只有当设备不再被使用时，模块才允许被卸载。

在 Linux 2.6 内核下，对于设备驱动工程师而言，很少需要亲自调用 try_module_get()与 module_put()，因为此时开发人员所写的驱动通常为支持某具体设备的 owner 模块，对此设备 owner 模块的计数管理由内核里更底层的代码（如总线驱动或是此类设备共用的核心模块）来实现，从而简化了设备驱动的开发。

4.9 模块的编译

我们可以为代码清单 4.1 的模板编写一个简单的 Makefile，如下所示：

```
obj-m := hello.o
```

并使用如下命令编译 Hello World 模块，如下所示：

```
make -C /usr/src/linux-2.6.15.5/ M=/driver_study/ modules
```

如果当前处于模块所在的目录，则以下命令与上述命令同等：

```
make –C /usr/src/linux-2.6.15.5 M=$(pwd) modules
```

其中-C 后指定的是 Linux 内核源代码的目录，而 M=后指定的是 hello.c 和 Makefile 所在的目录，编译结果如下所示：

```
[root@localhost driver_study]# make -C /usr/src/linux-2.6.15.5/ M=/driver_study/ modules
make: Entering directory '/usr/src/linux-2.6.15.5'
  CC [M]  /driver_study/hello.o
/driver_study/hello.c:18:35: warning: no newline at end of file
  Building modules, stage 2.
  MODPOST
  CC      /driver_study/hello.mod.o
  LD [M]  /driver_study/hello.ko
make: Leaving directory '/usr/src/linux-2.6.15.5'
```

从中可以看出，编译过程中经历了这样的步骤：先进入 Linux 内核所在的目录，并编译出 hello.o 文件，运行 MODPOST 会生成临时的 hello.mod.c 文件，而后根据此文件编译出 hello.mod.o，之后连接 hello.o 和 hello.mod.o 文件得到模块目标文件 hello.ko，最后离开 Linux 内核所在的目录。

中间生成的 hello.mod.c 文件的源代码如代码清单 4.7 所示。

代码清单 4.7　模块编译时生成的.mod.c 文件

```
1   #include <linux/module.h>
2   #include <linux/vermagic.h>
3   #include <linux/compiler.h>
4
```

```
5   MODULE_INFO(vermagic, VERMAGIC_STRING);
6
7   struct module __this_module
8   __attribute__((section(".gnu.linkonce.this_module"))) = {
9    .name = KBUILD_MODNAME,
10   .init = init_module,
11  #ifdef CONFIG_MODULE_UNLOAD
12   .exit = cleanup_module,
13  #endif
14  };
15
16  static const char __module_depends[]
17  __attribute_used__ __
18  __attribute__((section(".modinfo"))) =
19  "depends=";
```

hello.mod.o 产生了 ELF（Linux 所采用的可执行/可连接的文件格式）的两个节，即 modinfo 和.gun.linkonce.this_module。

如果一个模块包括多个.c 文件（如 file1.c、file2.c），则应该以如下方式编写 Makefile：

```
obj-m := modulename.o
module-objs := file1.o file2.o
```

4.10 模块与 GPL

对于自己编写的驱动等内核代码，如果不编译为模块则无法绕开 GPL，编译为模块后企业在产品中使用模块，则公司对外不再需要提供对应的源代码，为了使公司产品所使用的 Linux 操作系统支持模块，需要完成如下工作。

- 在内核编译时应该选上“Enable loadable module support”，嵌入式产品一般不需要动态卸载模块，所以“可以卸载模块”不用选，当然选了也没关系，如图 4.1 所示。

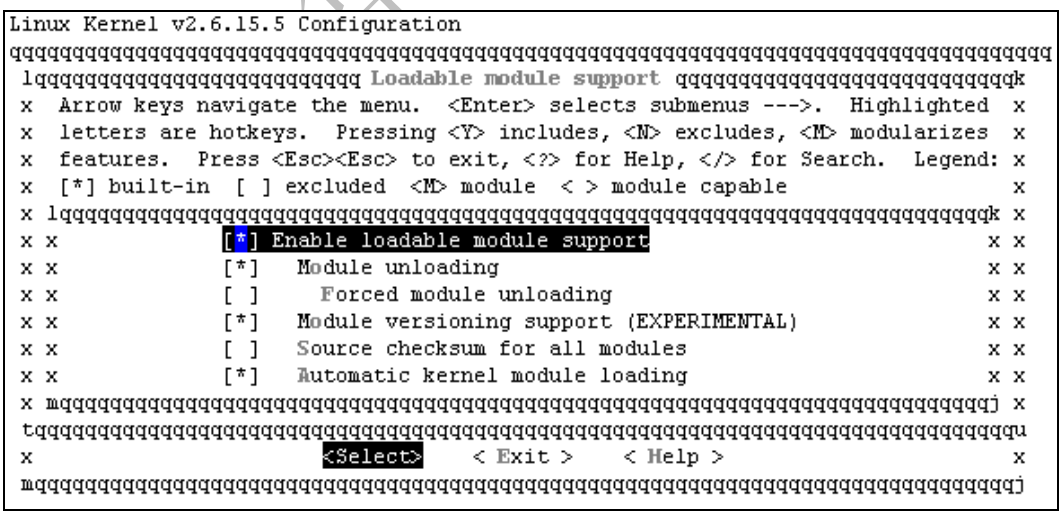

图 4.1 内核中支持模块的编译选项

如果有项目被选择“M”，则编译时除了 make bzImage 以外，也要 make modules。

- 将我们编译的内核模块.ko 文件放置在目标文件系统的相关目录中。
- 产品的文件系统中应该包含了支持新内核的 insmod、lsmod、rmmod 等工具，由于嵌入式产品中一般不需要建立模块间依赖关系，所以 modprobe 可以不要，一般也不需要卸载模块，所以 rmmod 也可以不要。

- 在使用中用户可使用 insmod 命令手动加载模块，如 insmod xxx.ko。
- 但是一般而言，产品在启动过程中应该加载模块，在嵌入式 Linux 的启动过程中，加载企业自己的模块的最简单的方法是修改启动过程的 rc 脚本，增加 insmod /.../xxx.ko 这样的命令。如某设备正在使用的 Linux 系统中包含如下 rc 脚本：

```
mount /proc
mount /var
mount /dev/pts
mkdir /var/log
mkdir /var/run
mkdir /var/ftp
mkdir -p /var/spool/cron
mkdir /var/config
...
insmod /usr/lib/company_driver.ko 2> /dev/null
/usr/bin/userprocess
/var/config/rc
```

4.11 总结

本章主要讲解了 Linux 内核模块的概念和基本的编程方法。内核模块由加载/卸载函数、功能函数以及一系列声明组成，它可以被传入参数，也可以导出符号供其他模块使用。

由于 Linux 设备驱动以内核模块的形式而存在，因此，掌握这一章的内容是编写任何类型设备驱动所必须的。在具体的设备驱动开发中，将驱动编译为模块也有很强的工程意义，因为如果将正在开发中的驱动直接编译入内核，而开发过程中会不断修改驱动的代码，则需要不断地编译内核并重启 Linux，但是如果编译为模块，则只需要 rmmod 并 insmod 即可，开发效率大为提高。

联系方式