

第 4 章

Linux 内核模块

本章导读

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了展现。4.1 ~ 4.2 节与 4.3 ~ 4.8 节是整体与部分的关系。

4.9 节说明了独立存在的 Linux 内核模块的 Makefile 文件编写方法和模块的编译方法。

4.10 节讨论了使用模块“绕开” GPL 的问题。

4.1 Linux 内核模块简介

Linux 内核的整体架构本就非常庞大，其包含的组件也非常多。而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 内核中。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。

有没有另一种机制可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，其对应的代码被动态地加载到内核中呢？

Linux 提供了这样的机制，这种机制被称为模块（Module）。模块具有这样的特点。

- 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。

为了使读者初步建立对模块的感性认识，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  /*
2  * a simple kernel module: hello
3  *
4  * Copyright (C) 2014 Barry Song (baohua@kernel.org)
5  *
```

```

6  * Licensed under GPLv2 or later.
7  */
8
9 #include <linux/init.h>
10 #include <linux/module.h>
11
12 static int __init hello_init(void)
13 {
14     printk(KERN_INFO "Hello World enter\n");
15     return 0;
16 }
17 module_init(hello_init);
18
19 static void __exit hello_exit(void)
20 {
21     printk(KERN_INFO "Hello World exit\n");
22 }
23 module_exit(hello_exit);
24
25 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
26 MODULE_LICENSE("GPL v2");
27 MODULE_DESCRIPTION("A simple Hello World Module");
28 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 GPL v2 许可权限的声明以及一些描述信息，位于本书配套 Ubuntu 的 /home/baohua/develop/training/kernel/drivers/hello 目录。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() 而不是用户空间的 printf()，printk() 的用法和 printf() 基本相似，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 驱动的调试章节中将会详细讲解。

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

Module	Size	Used by
hello	9 472	0
nls_iso8859_1	12 032	1
nls_cp437	13 696	1
vfat	18 816	1
fat	57 376	1 vfat
...		

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

```
$ cat /proc/modules
hello 12393 0 - Live 0xe67a2000 (OF)
nls_utf8 12493 1 - Live 0xe678e000
isofs 39596 1 - Live 0xe677f000
vboxsf 42561 2 - Live 0xe6767000 (OF)
...
```

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，该目录下又有一个 refcnt 文件和一个 sections 目录，在 /sys/module/hello 目录下运行 “tree -a” 可得到如下目录树：

```
root@barry-VirtualBox:/sys/module/hello# tree -a
.
├── coresize
├── holders
├── initsize
├── initstate
├── notes
│   └── .note.gnu.build-id
├── refcnt
└── sections
    ├── .exit.text
    ├── .gnu.linkonce.this_module
    ├── .init.text
    ├── .note.gnu.build-id
    ├── .rodata.str1.1
    ├── .strtab
    └── .symtab
├── srcversion
├── taint
└── uevent

3 directories, 15 files
```

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。模块之间的依赖关系存放在根文件系统的 /lib/modules/<kernel-version>/modules.dep 文件中，实际上是在整体编译内核的时候由 depmod 工具生成的，它的格式非常简单：

```
kernel/lib/cpu-notifier-error-inject.ko: kernel/lib/notifier-error-inject.ko
kernel/lib/pm-notifier-error-inject.ko: kernel/lib/notifier-error-inject.ko
kernel/lib/lru_cache.ko:
kernel/lib/cordic.ko:
kernel/lib/rbtree_test.ko:
kernel/lib/interval_tree_test.ko:
updates/dkms/vboxvideo.ko: kernel/drivers/gpu/drm/drm.ko
```

使用 modinfo <模块名> 命令可以获得模块的信息，包括模块作者、模块的说明、模块

所支持的参数以及 vermagic:

```
# modinfo hello.ko
filename:      /home/baohua/develop/training/kernel/drivers/hello/hello.ko
alias:         a simplest module
description:   A simple Hello World Module
license:       GPL v2
author:        Barry Song <21cnbao@gmail.com>
depends:
vermagic:      4.0.0-rc1 SMP mod_unload 686
```

4.2 Linux 内核模块程序结构

一个 Linux 内核模块主要由如下几个部分组成。

(1) 模块加载函数

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。

(2) 模块卸载函数

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，完成与模块卸载函数相反的功能。

(3) 模块许可证声明

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 内核模块领域，可接受的 LICENSE 包括 “GPL”、“GPL v2”、“GPL and additional rights”、“Dual BSD/GPL”、“Dual MPL/GPL” 和 “Proprietary”（关于模块是否可以采用非 GPL 许可权，如 “Proprietary”，这个在学术界和法律界都有争议）。

大多数情况下，内核模块应遵循 GPL 兼容许可权。Linux 内核模块最常见的是以 MODULE_LICENSE(“GPL v2”) 语句声明模块采用 GPL v2。

(4) 模块参数 (可选)

模块参数是模块被加载的时候可以传递给它的值，它本身对应模块内部的全局变量。

(5) 模块导出符号 (可选)

内核模块可以导出的符号 (symbol，对应于函数或变量)，若导出，其他模块则可以使用本模块中的变量或函数。

(6) 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，错误编码是一个接近于 0 的负值，在 `<linux/errno.h>` 中定义，包含 `-ENODEV`、`-ENOMEM` 之类的符号值。总是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 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下列代码：

```
request_module(module_name);
```

灵活地加载其他内核模块。

在 Linux 中，所有标识为 `__init` 的函数如果直接编译进入内核，成为内核镜像的一部分，在连接的时候都会放在 `.init.text` 这个区段内。

```
#define __init      __attribute__((__section__(".init.text")))
```

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）的内存。

除了函数以外，数据也可以被定义为 `__initdata`，对于只是初始化阶段需要的数据，内核在初始化完后，也可以释放它们占用的内存。例如，下面的代码将 `hello_data` 定义为 `__initdata`：

```

static int hello_data __initdata = 1;

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, world %d\n", hello_data);
    return 0;
}
module_init(hello_init);

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, world\n");
}
module_exit(hello_exit);

```

4.4 模块卸载函数

Linux 内核模块加载函数一般以 `__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，而不返回任何值，且必须以“`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。

我们用 `__exit` 来修饰模块卸载函数，可以告诉内核如果相关的模块被直接编译进内核（即 `built-in`），则 `cleanup_function()` 函数会被省略，直接不链进最后的镜像。既然模块被内置了，就不可能卸载它了，卸载函数也就没有存在的必要了。除了函数以外，只是退出阶段采用的数据也可以用 `__exitdata` 来形容。

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 1 个整型参数和 1 个字符指针参数：

```

static char *book_name = "dissecting Linux Device Driver";
module_param(book_name, charp, S_IRUGO);

static int book_num = 4000;
module_param(book_num, int, S_IRUGO);

```

在装载内核模块时，用户可以向模块传递参数，形式为“`insmod (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缺省值。如果模块被内置，就无法 `insmod` 了，但是 `bootloader` 可以通过在 `bootargs` 里设置“`模块名.参数名 = 值`”的形式给该内置的模块传递参数。

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。

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)`”。

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，其中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。

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，位于本书配套源代码/kernel/drivers/param目录下），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
4 static char *book_name = "dissecting Linux Device Driver";
5 module_param(book_name, charp, S_IRUGO);
6
7 static int book_num = 4000;
8 module_param(book_num, int, S_IRUGO);
9
10 static int __init book_init(void)
11 {
12     printk(KERN_INFO "book name:%s\n", book_name);
13     printk(KERN_INFO "book num:%d\n", book_num);
14     return 0;
15 }
16 module_init(book_init);
17
18 static void __exit book_exit(void)
19 {
20     printk(KERN_INFO "book module exit\n ");
21 }
22 module_exit(book_exit);
23
24 MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
25 MODULE_LICENSE("GPL v2");
26 MODULE_DESCRIPTION("A simple Module for testing module params");
27 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：

```
# 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' book_num=5000”命令时，输出的是用户传递的参数：

```
# 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
```

另外，在`/sys`目录下，也可以看到`book`模块的参数：

```
barry@barry-VirtualBox:/sys/module/book/parameters$ tree
.
├── book_name
└── book_num
```

并且我们可以通过“`cat book_name`”和“`cat book_num`”查看它们的值。

4.6 导出符号

Linux 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中：

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

导出的符号可以被其他模块使用，只需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。

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
4 int add_integer(int a, int b)
5 {
6     return a + b;
7 }
8 EXPORT_SYMBOL_GPL(add_integer);
9
10 int sub_integer(int a, int b)
11 {
12     return a - b;
13 }
14 EXPORT_SYMBOL_GPL(sub_integer);
15
16 MODULE_LICENSE("GPL v2");
```

从“`/proc/kallsyms`”文件中找出`add_integer`、`sub_integer`的相关信息：

```
# grep integer /proc/kallsyms
e679402c r __ksymtab_sub_integer [export_symb]
```

```
e679403c r __kstrtab_sub_integer [export_symb]
e6794038 r __kcrcatab_sub_integer [export_symb]
e6794024 r __ksymtab_add_integer [export_symb]
e6794048 r __kstrtab_add_integer [export_symb]
e6794034 r __kcrcatab_add_integer [export_symb]
e6793000 t add_integer [export_symb]
e6793010 t sub_integer [export_symb]
```

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，以表明该驱动模块所支持的设备，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 /* table of devices that work with this driver */
2 static struct usb_device_id skel_table [] = {
3 { USB_DEVICE(USB_SKEL_VENDOR_ID,
4     USB_SKEL_PRODUCT_ID) },
5 { } /* terminating entry */
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 以后的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()` 和 `module_put()` 的引入、使用与 Linux 2.6 以后的内核下的设备模型密切相关。Linux 2.6 以后的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管理模块的使用计数。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 以后的内核下，对于设备驱动而言，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管理模块，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：

```
KVERS = $(shell uname -r)

# Kernel modules
obj-m += hello.o

# Specify flags for the module compilation.
#EXTRA_CFLAGS=-g -O0

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

该 Makefile 文件应该与源代码 `hello.c` 位于同一目录，开启其中的 `EXTRA_CFLAGS=-g -O0`，可以得到包含调试信息的 `hello.ko` 模块。运行 `make` 命令得到的模块可直接在 PC 上运行。

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
modulename-objs := file1.o file2.o
```

4.10 使用模块“绕开”GPL

Linux内核有两种导出符号的方法给模块使用，一种方法是EXPORT_SYMBOL()，另外一种是EXPORT_SYMBOL_GPL()。这一点和模块A导出符号给模块B用是一致的。

内核的Documentation/DocBook/kernel-hacking.tmpl明确表明“the symbols exported by EXPORT_SYMBOL_GPL() can only be seen by modules with a MODULE_LICENSE() that specifies a GPL compatible license.”由此可见内核用EXPORT_SYMBOL_GPL()导出的符号是不可以被非GPL模块引用的。

由于相当多的内核符号都是以EXPORT_SYMBOL_GPL()导出的，所以历史上曾经有一些公司把内核的EXPORT_SYMBOL_GPL()直接改为EXPORT_SYMBOL()，然后将修改后的内核以GPL形式发布。这样修改内核之后，模块不再使用内核的EXPORT_SYMBOL_GPL()符号，因此模块不再需要GPL。对此Linus的回复是：“I think both them said that anybody who were to change a xyz_GPL to the non-GPL one in order to use it with a non-GPL module would almost immediately fall under the “willful infringement” thing, and that it would make it MUCH easier to get triple damages and/or injunctions, since they clearly knew about it”。因此，这种做法可能构成“蓄意侵权(willful infringement)”。

另外一种做法是写一个wrapper内核模块（这个模块遵循GPL），把EXPORT_SYMBOL_GPL()导出的符号封装一次后再以EXPORT_SYMBOL()形式导出，而其他的模块不直接调用内核而是调用wrapper函数，如图4.1所示。这种做法也具有争议。

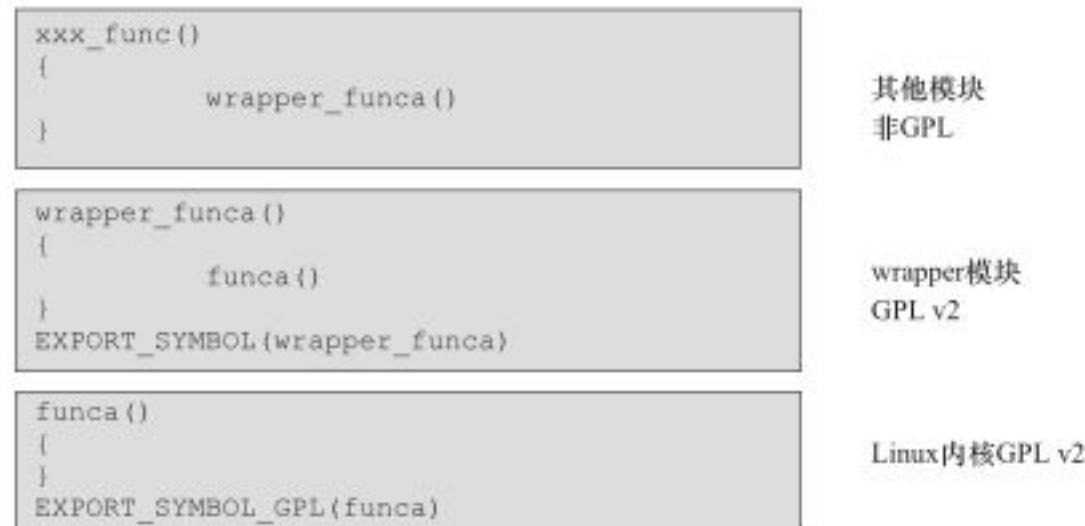


图4.1 将EXPORT_SYMBOL_GPL重新以EXPORT_SYMBOL导出

一般认为，保守的做法是 Linux 内核不能使用非 GPL 许可权。

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存在，因此，掌握这一章的内容是编写任何设备驱动的必需。