简单内核模块的测试

第4章

- 内核模块编程的第一步
- 内核模块程序的准备
- 内核模块程序的组成
- 用于模块编译的Makefile
- 模块参数的说明
- 内核消息的输出
- 内核与模块

也许有时误认为内核模块和设备驱动程序是一回事，但其实是完全不同的概念。设备驱动程序是用于驱动硬件的程序库，而内核模块是内核运行中可动态加载的内核程序库。Linux与其他系统不同，可以提供很多功能（如，文件系统、设备驱动程序、通信程序等），其中设备驱动程序就是实现驱动硬件功能的。要编写设备驱动程序至少要掌握内核模块加载函数insmod( )和卸载函数rmmod()。

尤其是内核模块具有程序特征，而且可以向内核动态加载和卸载。所以，与一般程序不同的源程序形式，对2.4内核版和2.6内核版略有区别。同样的，相应的Makefile也有所不同。

本章基于内核版本2.4与版本2.6，将介绍内核模块程序所需的基础知识，并了解内核模块的加载和卸载过程。

2.4内核
2.4内核的模块形式与编译方法
MODULE_PARM宏定义
2.6内核
2.6内核的模块形式与编译方法
MODULE_PARAM宏定义

4.1　内核模块编程的第一步

需要指出的是，并不是为了刻意追随通常语言类的书以输出“Hello world”的惯例。但本章还是以输出“Hello world”作为引例，来介绍编写简单的设备驱动程序的方法。

首先让我们看一下输出“Hello world”的一般程序。

```
#include <stdio.h>
void main(int argc,char **argv)
{
   printf("Hello world\n");
}
```

这个例子虽然显得过于简单，但至少具备了能够编译通过的起码的条件。通过本例，将了解编写C语言程序必须要知道的头文件和编译链接所需的main函数的声明方法，以及用于输出结果的最基本的printf()函数的使用方法。

作者在本书中将尽可能以简单的例子作为引例来讲解设备驱动程序的编写方法。下面首先将介绍，为了编写并测试内核模块化的简单设备驱动程序而应掌握的基本内容。

- 程序中应包含的头文件、基本组成函数、宏定义；
- C函数库中的类似printf( )的字符串输出函数printfk( )；
- 设备驱动程序模块的编译方法；
- 用于内核模块管理的utility的使用方法。

其中除内核模块管理utility之外的其他项的使用方法，在内核版本2.4和2.6中略有不同。所以将按其版本号举例说明具体使用方法。

首先以输出“Hello world”的内核模块作为例子。进入内核模块说明之前请记住如同一般C语言程序有开始和结束，内核模块也有加载到内核和从内核卸载的过程。详细说明请看以下内容。

4.1.1　“Hello world”的内核模块（内核版本2.4）

下面的例子是在内核版本2.4中运行，所以首先应启动内核版本2.4.21。对此已在第1章中作了说明。

1. 实例

[实例4-1]　内核模块代码2.4(base/test.c)

```
01 #define MODULE
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04
05 int init_module(void)
06 {
07 printk("Hello, world\n");
08 return 0;
09 }
```

```
10
11 void cleanup_module(void)
12 {
13 printk("Goodbye world\n");
14 }
```

本例可以用gcc加上相应的选项后以命令行形式编译。但为了减少失误以及方便以后再用，还是作Makefile。

[实例4-2]内核模块代码2.4(/base/Makefile)

```
1 KERNELDIR = /lib/modules/$(shell uname -r)/build
2 CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include -O
3
4 all: test.o
5
6 clean:
7  rm -rf *.o
```

注意

类似rm –rf *.o的Makefile 执行命令之前必须用<TAB>键缩进，如果用空格健来缩进的话，Make就不能处理对应的命令。

2. 执行方法

为了编译内核模块执行如下命令的话，将生成test.o文件。这就是内核模块文件。

```
[root@]#make
```

要使该内核模块能够在内核中运行，应加载到内核。

```
[root@]#insmod test.o
```

但是，如果上述命令在xWindow终端或者在别的计算机上用telnet来执行的话不会输出“Hello world”。若要看运行结果则执行如下命令。

```
[root@]#dmesg
```

这个命令用来输出内核的输出信息。如果输出信息末尾若有“Hello world”字符串，则说明运行正常。

用lsmod命令可以确认内核中是否包含了test.o模块，也可以用如下命令卸载该模块。

```
[root@]#rmmod test
```

4.1.2 “Hello world”的内核模块（内核版本2.6）

要想测试如下所示的设备驱动程序模块，必须以版本2.6.4来启动系统。有关版本号的确认方法请参照第1章说明。

1. 实例

[实例4-3]　模块代码2.6(/base/test.c)

```
01 #include <linux/init.h>
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04
05 static int hello_init(void)
06 {
07      printk("Hello, world \n");
08      return 0;
09 }
10

11 static void hello_exit(void)
12 {
13     printk("Goodbye, world\n");
14 }
15
16 module_init(hello_init);
17 module_exit(hello_exit);
18
19 MODULE_LICENSE("Dual BSD/GPL");
```

编译内核版本2.6上运行的模块的Makefile与内核版本2.4有所不同。

[实例4-4]　Makefile 2.6（/base/Makefile）

```
01 obj-m  := test.o
02
03 KDIR   := /lib/modules/$(shell uname -r)/build
04 PWD    := $(shell pwd)
05
06 default:
07       $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
08 clean:
09      rm -rf *.ko
10      rm -rf *.mod.*
11      rm -rf .*.cmd
12      rm -rf *.o
```

2. 执行方法

如同内核版本2.4上的内核模块编译，编译内核模块应执行如下命令，随之生成test.ko文件。

```
[root@]# make
```

若要执行该内核模块，用如下命令将其加载到内核中。

```
[root@]# insmod test.ko
```

从内核中卸载时要使用如下命令。

```
[root@]#rmmod test.ko
```

4.1.3 另一种形式的内核版本2.4的模块程序

前面列举了用于内核版本2.4的模块形式的例子，但内核版本2.6.X上所用的模块仍然可以在内核版本2.4上使用。这是因为内核版本2.4以后开始支持内核版本2.6.X上所用的模块形式。内核版本2.4上可以使用的内核版本2.6形式的模块如下。

[实例4-5] 内核版本2.6模块结构

```
01 define MODULE
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04
05 static int hello_init(void)
06 {
07      printk("Hello, world \n");
08      return 0;
09 }
10

11 static void hello_exit(void)
12 {
13     printk("Goodbye, world\n");
14 }
15
16 module_init(hello_init);
17 module_exit(hello_exit);
```

为了保持说明上的一贯性，在本书中假定支持前面所述的内核版本2.4的模块形式。

4.2 内核模块程序的准备

所谓模块的概念是在Linux中作者最欣赏的创意。通过模块方式一举解除了众多Linux爱好者的痛苦。另外，模块作为设备驱动程序开发者而言也是必须要掌握的一项基本要领。在本节将了解到模块的实现原理和内核相关的符号表以及模块操作实用程序(utility)。

过去开始制作内核时，一定是把制作好的设备驱动程序加到内核源程序后才可以进行编译和测试。换言之，内核程序里包含了设备驱动程序的程序。当然，这种方式现在仍在使用，但随之而来的是，要开发设备驱动程序就需要很长的内核编译过程，而且若要修改驱动程序，就得重新启动系统，甚至因为驱动程序的错误导致内核中途停止运行。

Linux内核开发组考虑到这种低效率的弊端，提出了模块的概念。这一概念的核心是，在内核启动之后能够动态地加载或卸载设备驱动程序。

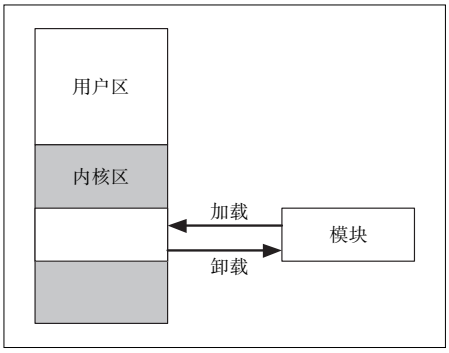
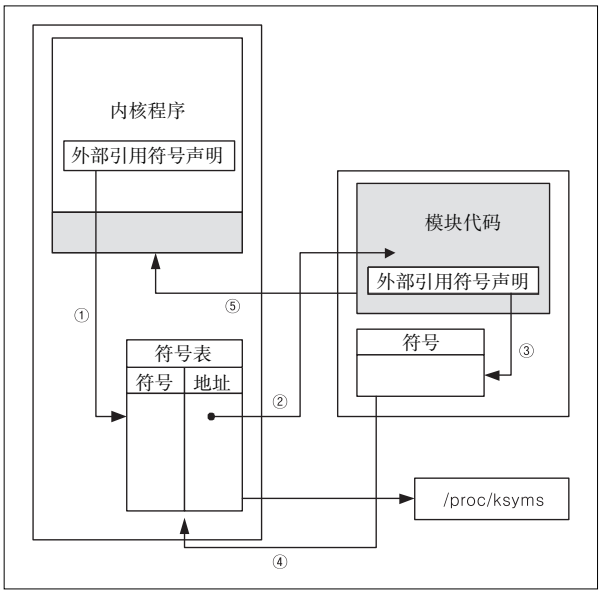


图4-1　模块概念

4.2.1　模块实现原理

内核也是作为程序经过编译、链接后生成可执行代码。Linux内核开发组也认识到这一特性，将链接功能体现在内核之中。即类似于设备驱动程序的内核函数库作成目标（object）代码形式，一旦有装载请求，就可以通过系统调用，内核将动态链接相应的目标代码。这一概念类似于动态函数库。但是因为内核已经是链接结束后的状态，所以也就不可能提供像链接符号表的实地址信息。正是因为不可能自行编译处理，所以在内核中追加了符号表功能。所谓符号表是在内核内部函数或变量中可供外部引用的函数符号表。利用这一符号表，将目标形式的内核模块例行程序与所引用的内部函数或变量实现动态链接。



①把内核中声明的外部引用符号登录到符号表中
②模块加载到内核时根据内核内的符号得到引用地址
③模块中外部引用的符号声明通过目标代码和符号表来管理
④模块中声明为外部引用符号登录到内核内部的符号表中
⑤声明模块代码中未声明的地址即可登录到内核之中

图4-2　模块动态链接示意图

4.2.2 内核提供的符号表：/proc/ksyms

在Linux启动后，以“/proc/ksyms”文件形式对外提供内核所用的符号表。可以用cat命令查看其内容，例如：

```
[root@]#cat /proc/ksyms
⋮
#c0134a50 register_chrdev_R1a5f156e
#c0134ae0 unregister_chrdev_Rc192d491
⋮
```

如图4-3所示，第一项表示内核内部的符号地址，第二项中的字符串表示符号名和符号版本信息。图中的例子表示，register_chrdev( )函数和unregister_chrdev( )函数被赋予的地址分别为0xc0134a50和0xc0134ae0。

函数名之后的数字部分_R1a5f156e和_Rc192d491，表示函数符号的版本信息。内核函数或变量有可能设计成按其内核版本作相应的操作或变成另外形态的变量。为了防止因版本差异所引起的误操作，应确保同一环境下的内核版本应进行编译后动态链接。

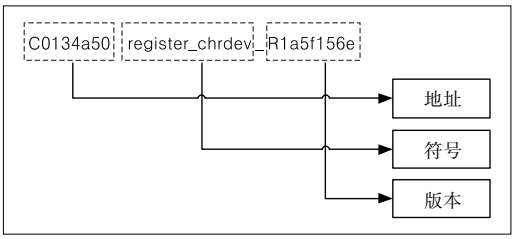


图4-3 ksyms的含义

4.2.3 模块应用程序

insmod是用于通过内核符号表系统把设计成模块的内核目标代码链接到内核的外部命令。该命令将设计成模块目标代码形式的设备驱动程序模块加载到内核时要求进行内存分配，并引用符号表把函数和变量的实际值分配到目标代码后再装载到内核中。

除此之外，还有如下用于模块化的设备驱动程序的加载或卸载的应用程序。

- insmod 加载模块到内核；
- rmmod 从内核卸载模块；
- lsmod 查看内核中加载的模块目录；
- depmod 生成模块间依赖性信息；
- modprobe 加载或卸载模块。

modprobe是用于内核模块自动安装功能的很有用的辅助程序。如果需要，特定模块通过modprobe可以加载或卸载指定模块。Modprobe程序通过引用module.dep文件，也可以登录使相应模块启动所必要的另外模块。

本书中结合实例详细介绍insmod、rmmod、lsmod命令。

由于内核版本2.6中与模块关联的内容有所变化，所以如果使用版本2.4中的模块应用程序的话将发生错误。因此要按下述网址，下载module-init-tools-0.9.11a.tar.gz以上版本后再安装。

http://www.kernel.org/pub/linux/kernel/people/rusty/modules/module-init-tools-0.9.11a.tar.gz

[root@]# in smod[-fkmpsxXv][-o module_name]object_file[symbol=value...].

[选项]

- -f　即使当前运行中的内核和编辑模块的内核版本不同，仍加载模块。
- -k　模块上高定auto-clean变量。由于kerneld，模块停止使用一定时间（1分钟）时，清除模块。
- -m　处于内核Panic状态时，输出加载（loadmap），使调试模块更容易。
- -o　不从源代码对象文件的基本名称中导出模块名，直接指定模块名称。
- -p　检查模块是否成功加载到内核当中。
- -s　输出syslog。
- -v　输出运行过程中生成的所有信息。

安装的模块实用程序版本或实用程序的种类不同，很可能不执行选项。Module-init-tools-0.9.11a中只执行-p、-s、-f选项。

1. rmmod

是清除加载到内核中的模块的命令。与insmod不同的是，它指定设备驱动程序模块名称。Init-tools-0.9.11a指定对象文件后，自动提取并清除模块名。旧版本必须指定模块名。

```
rmmod[-f][-w][-s][-v][modulename]
[选项]
• -v-verbose　输出运行过程中生成的所有信息。
• -f-force　不能清除模块的状态下强制清除模块。此时，不能保障内核的稳定性。
• -w--wait　等到可清除模块的状态后，再清除模块。
• -s 输出syslog。
```

2. lsmod

显示加载到内核中的模块状态。下面给出了运行实例。

```
[root@]# lsmod
Module            Size     Used by
Test              1344        0
Autofs           15328        0
parport_pc       20104        0
parport          40768        1       parport_pc
```

第1行表示加载的模块名。第2行表示模块占据内核的记忆空间大小。第3行表示是否正在运行。第4行表示参照相应模块的模块名。

4.3 内核模块程序的组成

到此为止，介绍了简单模块程序的编写、编译、加载、卸载方法。本节结合前面章节列举的模块代码的具体内容，逐一介绍模块程序的组成要点。

4.3.1 声明头文件

为编写模块代码，首先声明内核代码中的头文件。gcc原则上把“/usr/include/”头文件作为默认的头文件目录。而模块程序引用内核程序的头文件所在目录，并在Makefile里作相应设定。编写模块程序时可以不必关心头文件的目录，但在内核版本2.4和2.6种，头文件内容和顺序有所区别。

在2.4内核中声明为如表4-1所示。必要时可以包含其他头文件。而在2.6内核中以#include <linux/init.h>来代替2.4内核中的#define MODULE。

表4-1 不同版本的头文件

2.4内核	2.6内核
#define MODULE	#include <linux/init.h>
#include <linux/module.h>	#include <linux/module.h>
#include <linux/kernel.h>	#include <linux/kernel.h>

4.3.2 模块初始化函数和删除函数的声明

由于模块是经过编译生成目标代码，所以需要用模块应用程序加载、链接过程。一旦模块链接到内核，内核为了初始化设备驱动程序，将调用特殊函数。在2.4内核中函数名是固定的，而在2.6内核中通过宏来定义多种函数名。从模块卸载时为了安全卸载设备驱动程序，调用内部结束处理函数。这个函数的定义方式在版本2.4与版本2.6中有区别的。

在2.4内核中，模块加载时调用init_module( )函数，而卸载模块时调用cleanuo_module( )函数。在版本2.6中，可以任意指定函数名。但是要用宏定义来声明是加载还是卸载函数。在下一个例子中，module_init是加载模块时引用的宏定义，而module_exit宏指定从内核卸载模块时调用的函数。

表4-2 不同版本中模块相关函数的声明

2.4内核	2.6内核
int init_module(void);	module_init(hello_init);
void cleanup_module(void);	module_exit(hello_exit);

4.3.3 2.6内核的权限登记

2.4内核中不一定要在设备驱动程序模块上登记LICENSE。2.6内核可以自动登记模块的LICENSE，但会发生错误。因此，以下列语句表示模块的LICENSE。

```
MODULE_LICENSE("Dual BSD/GPL");
```

作为MODULE_LICENSE的字符串和其意义可参考[表4-3]，不涉及企业密码保护时，可以使用“Dual BSD/GPL”。省略及“Proprietary”状态表示不能使用几个API，但是不会出现太大的问题。

表4-3　MODULE_LICENSE中使用的字符串和意义

LICENSE	名称
GPL	GNU Public License v2 or later
GPL v2	GNU Public License v2
GPL and additional rights	GNU Public License v2 rights and more
Dual BSD/GPL	GNU Public License v2 or BSD license choice
Dual MPL/GPL	GNU Public License v2 or Mozilla license choice
Proprietary	Non free products

4.4　用于模块编译的Makefile

模块必须由程序执行编译，而Linux中利用make实用程序编译模块。本节介绍利用make实用程序编译设备驱动程序模块的Makefile的编写方法。对于2.4内核，编译设备驱动程序模块时，直接执行命令行中的编译命令。

```
[root@ ]#  cc      -D__KERNEL__-DMODULE  -Wall  -O2  -I/usr/src/linux/
       include -c hello.c -o hello.o
```

但是2.6内核版本的处理方法较为简单。因此利用Makefile编译模块。

内核还提供了编译外部目录中模块的方法。前面介绍的实例就是典型的形态，在这里熟悉Makefile后，以相同的方式使用该命令。

4.4.1　2.4内核中编译外部模块的Makefile

2.4内核中用于编译模块的Makefile命令非常简单。下面再次给出了前面的实例。

1. 实例

[实例4-6]　Makefile分析

```
01  KERNELDIR = /lib/modules/$(shell uname -r)/build
02  CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include -O
03
04  all: test.o
05
06  clean:
07      rm -rf *.o
```

2. 源代码解析

（1）第1行

指定编译的模块将要引用的内核源代码的目录。模块与内核有着紧密的联系，为了防止因

版本出错，要统一运行模块的内核源代码。此时可以直接指定源代码，也可以间接指定其代码。该结构对于编译内核后利用make modules_install命令配置模块的情况较为有效，此时编译的内核应处于运行状态。

“uname-r”命令输出当前运行中的内核版本信息。/lib/modules/内核版本/build文件以符号连接在与当前运行中的内核源代码所匹配目录上。内核源代码为/usr/src/linux，内核版本为2.4.18时，/lib/modules/2.4.18/build文件以符号连接在/usr/src/linux上。

（2）第2行

编译模块时指定的编译选项。

- D_ _KERNEL _ _　激活内核有关的内容
- DMODULE　激活与模块有关的内容
- O　优化
- I$(KERNELDIR)/include　模块源代码中引用的内核头文件位置

（3）第4行

创建test.o的Makefile语句。利用多个对象创建test.O对象时定义下列内容。

```
all:   test.o
test.o:   test1.o test2.o test3.o
ld -r test1.o test2.o test3.o -o test.o
```

（4）第6行

清除所有编译生成的文件。

```
[root@ ]# make clean
```

4.4.2 2.6内核中编译外部模块的Makefile

1. 实例

[实例4-7] Makefile分析

```
01  obj-m    := test.o
02
03  KDIR     := /lib/modules/$(shell uname -r)/build
04  PWD      := $(shell pwd)
05
06  default:
07      $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
08  clean:
09      rm -rf *.ko
10      rm -rf *.mod.*
11      rm -rf .*.cmd
12      rm -rf *.o
```

2. 源代码解析

（1）第1行

定义生成模块的名称。没有特殊约定时，test.c将会成为编译成test.O的源代码文件。

```
obj-m := test.o
module-objs := test1.o test2.o test3.o
```

（2）第3行

指定内核源代码的位置。

（3）第4行

指定编译对象模块源代码所在位置的当前目录。

（4）第6行

指定编译模块的命令。

（5）第8行

利用编译结果清除所有的生成文件。

```
[root@ ]# make clean
```

编译结果生成了很多文件，2.6内核中生成的模块实际名称为test.ko，也可以使用test.o。

4.5　模块参数的说明

对于从未接触内核或设备驱动程序的人来说运行硬件的boot初始化脚本几乎是未知的黑箱空间。该初始化脚本处理几个模块相关的内容，其中之一就是处理运行模块时使用的控制值。本节详细介绍了内核加载模块后处理相应参数的方法。

设备驱动程序通常使用固定的内部变量值，而由源代码指定该变量。根据具体情况，还可在运行设备驱动程序之前修改参量。代表性的例子为设定不支持即插即用（PnP）的网卡的输入输出地址或IRQ值（最近基本使用PCI卡，所以很少见）。另外，视频设置也是一种很好的实例。该类参数属于初始化变量，可在加载或卸载模块时进行修改。通常利用boot loader的选项功能或模块的实用程序insmod命令调整为内核命令模式。设备驱动程序的初始化参数中外部能够修改的称做模块变量。设备驱动程序创建为模块后，利用下列insmod命令可以指定变量。

```
insmod ./test.ko onevalue=0x27 twostring="Oh my god!!"
```

由insmod命令指定模块变量，而模块插入内核时处理该变量。使用特定宏定义输入输出端口和IRQ号经常使用的模块变量。该宏函数根据变量提供类型信息，因此内核可以检查加载模块时写入的数据。对于正数，该值应是10进制、8进制或16进制中的一种。

为了使用insmod( )函数在以模块形式加入的对象变量上代入数据，把10进制指定17，8进制指定021，而16进制指定为0x11形态。传送值为陈列结构时，利用规定顺序和逗号（，）区分。可以省略各个要素，从而进入下一步骤。对于引号(”)开始的字符串，可以处理C语言中使用的转义序列（escape sequence）。shell提示器（prompter）中shell的注释会先处理引号，因此需要搜索。

下面通过实例帮助读者进一步了解定义变量的方法和使用该变量的方法。

4.5.1　2.4内核模块变量的实例

以下为2.4内核中使用模块变量的例子。编译该实例的文档与前面介绍的Makefile的内容

相同。实例中介绍了利用模块实用程序insmod命令控制设备驱动程序内部定义的一个正数和一个字符串的定义方法。

1. 实例

[实例4-8] 2.4内核模块变量实例（2.6/param/test.c）

```
01  #define MODULE
02  #include <linux/module.h>
03  #include <linux/kernel.h>
04
05  static int onevalue = 1;
06  static char *twostring = NULL;
07
08  MODULE_PARM(onevalue, "i");
09  MODULE_PARM(twostring, "s");
10
11  int init_module(void)
12  {
13      printk("Hello, world [onevalue=%d:twostring=%s]\n", onevalue,
        twostring);
14      return 0;
15  }
16
17  void cleanup_module(void)
18  {
19      printk("Goodbye world\n");
20  }
21
22  MODULE_AUTHOR("You Young-chang frog@falinux.com");
23  MODULE_DESCRIPTION("Module Parameter Test Module");
```

2. 源代码解析

（1）[test.c]8~9行

初始化模块时设定正数变量onevalue和字符串变量twostring。为实现MODULE_PARM宏的外部引用定义符号，MODULE_PARM宏的语法如下。

```
MODULE_PARM（变量名，变量类型）扩展
[变量类型]
• b:byte
• h:short
• i(I小写字母):int
• l(L小写字母):long
• s:string
```

MODULE_PARM宏不能定义变量，因此事先定义将使用的变量。MODULE_ PARM(onevalue,“i”);为正数，前面定义为int onevalue = 1;形态。另外，由接受字符串的char*twostring定义的字符串地址变量不分配内存，因此要特别留意。

（2）[test.c]13行

输出模块变量的设定值。

变量或常数上使用的static

定义设备驱动程序的变量或常数时使用static关键字。编译语法中在变量或常数上定义static时，不能从外部连接，只能在相同的源代码中调用或者使用。Linux内核不能检索所有的变量状态，为使源代码的变量和函数不与其他源代码冲突，只能采取上述调用方式。不会复用定义的函数或变量时可以不使用static关键字。此时static的意义受到全局变量或函数的限制。因为函数内部变量上使用static关键字，就会起到全局变量的效果。

3. 实施方法

为了测试实例利用make命令进行编译。然后内核上加载编译的模块，并传递模块变量，该命令如下。

```
[root@ ]# insmod test.o onevalue=0x27 twostring="Oh my godrmmod test"
```

实施结果如下。

```
[root@ ]# insmod test.o onevalue=0x27 twostring="Oh my godrmmod test"
[root@ ]# rmmod test
[root@ ]# dmesg
    ⋮
    ⋮
Hello, world [onevalue=39:twostring=Oh my godrmmod test]
Goodbye world
```

获得其他模块信息的方法

开发设备驱动程序时，通常会添加设备驱动程序设计人和设备驱动程序有关的说明。为了在设备驱动程序上包含此类信息，提供MODULE_AUTHOR和MODULE_DESCRIPTION宏。

4.5.2　2.6内核模块变量的使用实例

2.6内核中可以使用2.4内核所使用的宏。此外，支持更加完善的宏。支持模块变量的使用属性并进一步扩大了变量类型。下面给出了2.6内核中使用模块变量的实例。

1. 实例

[实例4-9]　2.6内核模块变量实例（2.6/param/test.c）

```
01  #include <linux/init.h>
02  #include <linux/module.h>
03  #include <linux/kernel.h>
```

```
04  #include <linux/moduleparam.h>
05
06  static int onevalue = 1;
07  static char *twostring = NULL;
08
09  module_param(onevalue, int, 0);
10  module_param(twostring, charp, 0);
11
12  static int hello_init(void)
13  {
14      printk("Hello, world [onevalue=%d:twostring=%s]\n", onevalue,
            twostring);
15      return 0;
16  }
17
18  static void hello_exit(void)
19  {
20      printk("Goodbye, world\n");
21  }
22
23  module_init(hello_init);
24  module_exit(hello_exit);
25
26  MODULE_AUTHOR("You Young-chang frog@falinux.com");
27  MODULE_DESCRIPTION("Module Parameter Test Module");
28  MODULE_LICENSE("Dual BSD/GPL");
```

2. 源代码解析

（1）[test.c]4行

与2.4内核不同，2.6内核中使用模块变量时应包含linux/ moduleparam.h文件。

（2）[test.c]6~10行

初始化模块时设定正数变量onevalue和字符串变量twostring。为实现MODULE_PARM宏的外部引用定义符号。2.6内核中由module_param宏代替2.4内核中使用过的MODULE_PARM宏。其使用方法与2.4内核基本相同。

```
module_param（变量名，变量类型、使用属性）
[变量类型]
• short: short
• ushort: unsigned short
• int: int
• uint: unsigned int
• long: long
• ulong: unsigned long
• charp: char*
• bool: int
• invbool: int
• intarray: int*
[使用属性]
```

为了测试实例利用make命令进行编译。然后利用下列命令，在内核上加载编译的模块，并传递模块变量。

3. 节点属性

除了特殊情况，通常把文件节点属性指定为0。该参数用来设置用户的访问权限，普通用户能在内核上加载模块的几率很少。如果一定要设置节点，可用文件权限（Permission）等概念指定0644等八进制数。

```
[root@ ]# insmod test.ko onevalue=0x27 twostring="Oh my godrmmod test"
```

实施结果如下：

```
[root@ ]# insmod test.ko onevalue=0x27 twostring="Oh my godrmmod test"
[root@ ]# rmmod test.ko
[root@ ]# dmesg
    ⋮
    ⋮
Hello, world [onevalue=39:twostring=Oh my godrmmod test]
Goodbye world
```

4.6　内核消息的输出

普通的应用程序中利用printf( )函数输出message，而利用scanf( )函数接收用户的输入。内核不接收用户的输入没有对应scanf( )的函数，但是使用printk( )函数输出运行状态。本节介绍用于输出内核信息的printk( )函数的使用方法以及printk( )函数在内核内部的处理方式。

4.6.1　printk( )函数

printk( )函数是创建并测试设备驱动程序时使用的最有效的调试工具。调试内核和设备驱动程序的方法较多，但是考虑到printk( )函数的方便性，实际调试过程中基本使用printk( )函数。该函数的使用方法与printf( )函数基本相似，但具有输出和管理内核消息的特性。

- 管理消息的记录级别（Log Level）；
- 原型队列结构的管理；
- 指定多个输出设备。

1. 记录级数（Log Level）

printk( )函数向外部输出内核的状态，即内核消息。该内核消息为了表示多数系统的运行状态，主要输出路径或错误以及其他系统的变化。由管理员记录或保存内核消息，而大容量系统生成的消息量也惊人，因此管理员分类管理内核消息。此时根据内核消息的重要程度定义级数，并表现在内核消息的头位置，而这便称为记录级别。

记录级别在printk( )函数上传送的字符串头文字上使用“<1>”等数字标记级别。该级别的有效范围为输出“\n”对应字符。但是表示级别时并不提倡使用“<1>”等形态。执行printk( )函数时，虽然将字符串起始位置“<”和“>”之间的数字分为级别，但比起直接使

用数字，使用linux/kernel.h文档上定义的声明更加有效。

表4-4 可使用的常数定义语句和意义

常数定义语句	意义
#define KERN_EMERG	“<0>” /*系统不运行*/
#define KERN_ALERT	“<1>” /*总是输出*/
#define KERN_CRIT	“<2>” /*关键信息*/
#define KERN_ERR	“<3>” /*error信息*/
#define KERN_WARNING	“<4>” /*提示信息*/
#define KERN_NOTICE	“<5>” /*正常的信息*/
#define KERN_INFO	“<6>” /*系统信息*/
#define KERN_DEBUG	“<7>” /*调试信息*/

作为例子，查看下列常数语句。

```
printk(KERN_INFO "system ok\n");
```

该定义句与下面两个语句意义相同。

```
printk("<6>" "system ok\n");
printk("<6>system ok\n");
```

不标记级别时，默认为KERN_WARNING等Level，下一个语句的处理结果相同。

```
printk(KERN_WARNING "system ok\n");
printk("<4>" "system ok\n");
printk("<4>system ok\n");
printk("system ok\n");
```

2. 原型队列结构的管理

内核消息并不直接输出到控制台设备上，而是保存在内核内部原型队列结构的日志缓存空间上。该日志缓存的大小指定为CONFIG_LOG_BUF_SHIFT值，表示为2的倍数。该值定义在include/config/log/buf/shift.h上，而默认值为14，因此日志缓存的容量应是16384字节。

保存内核消息的日志缓存被连接在输出的控制台设备和记录数据的日志程序上。因此，由printk( )函数保持到日志缓冲上的速度比转移数据的速度要快，一旦超出日志缓存大小，就会损失最早的数据。

3. 日志缓存和控制台

Linux系统以多种方式输入命令。可以利用监控器（monitor）和键盘，也可以采取telnet等方式登录其他系统完成输入。X系统可以打开多个虚拟终端窗口。各个终端在任何情况下执行相同的权限和功能，因此内核无法判断接收内核消息的设备。接收内核消息的设备统称为控制台。

对于PC机中运行的Linux系统，VGA测试画面便是控制台。X系统中利用特殊命令指定终端后方可成为控制台。启动设备后，对于直接在X系统上运行的Linux而言，按下CTRL+ALT+F1弹出的画面便是控制台。嵌入式系统把串行接口作为控制台。

4.6.2　管理内核内存的daemon

内核消息通过控制台告知管理员，同时在特殊的守护进程（daemon）中记录和管理此消息。

- klogd　记录和管理内核发生的消息；
- syslogd　记录并管理内核发生的消息和应用程序中请求的系统信息。

这两个守护进程都在Linux系统中运行时，把所有的内核消息都记录到/var/log/messages文件上。若发生过多的消息无法输出全部内容时，结束klogd守护进程，利用-c选项重新设置，对于更加详细的内容请参考llogd有关的帮助。当内核发生fault时，控制台的级别自动设定为KERN_DEVUG。

4.6.3　dmesg命令

使用dmesg命令可以查看日志缓存中记录的内容。该命令显示原型队列缓存中记录的内容。日志缓存上记录的内容都输出到控制台上，且由内核的守护进程将消息记录在/var/log/messages上，原形队列的结构特性最近发生的消息仍留在日志缓存上。

dmesg命令利用此类特性输出最近发生的内核消息。

4.6.4　/proc/kmsg

控制台转换为终端就可以直接输出内核的messages。但是该方法有一定的难度，而且还要补充相关的程序。最简单的方法是输出/proc/kmsg文件。每次发生内核messages就立即输出，并执行下列命令查看结果。

```
cat/proc/kmsg
```

中止输出时，按下CTRL+C键结束程序。这样对话框即使不是控制台，也可以轻松查看内核的messages。

4.6.5　printk( )函数的注意事项

创建并测试设备驱动程序时，有可能过多地使用printk( )函数。但是，printk( )函数的执行时间较长，应尽量细化使用过程。另外，在时间处理相关程序中使用printk( )函数时，有可能延迟时间而出错。

printk( )函数输出时必须匹配各行的字符。因此测试设备驱动程序的过程中为了查看状态必须使用“\n”字符。设备驱动程序的初学者经常忽略这一点，导致调试过程混乱，应特别留意。

4.7　内核与模块

就算把设备驱动程序创建为模块，也没有必要理解模块有关的所有内容，因此根据读者需求可以跳过将要介绍的内容。但是，若能理解内核处理模块的方法及特性，将有助于理解

内核、模块以及设备驱动程序。读者可以简单阅读本节的内容，巩固原有知识。

4.7.1 创建为模块的原因

Linux内核不是很大。但这仅仅是相对的概念，仅表示比其他操作系统小，但是与嵌入式操作系统相比还是非常庞大的。其实Linux的所谓小是指比起庞大的功能而言的。

Linux内核支持庞大的功能。不仅支持i386，而且支持当前的大部分32位处理器。因此，不仅支持多种平台的设备驱动程序，还体现所有的文件系统及网络相关功能。另外，其结构优化合理。但是，一个系统不可能使用所有的功能，为了在系统中生成优化的内核，可以设置内核编译选项。这也是公开源代码的Linux内核的一大优点。利用内核编译选项可以优化Linux内核，但面向不确定用户的二进位形态发行版还是要包含支持设备驱动程序的所有功能。这是内核增大的主要原因。

但是，自从使用模块扩大内核功能后，渐渐解决了此问题。发行版只包含相同进程的基本功能，其他发行为模块。另外，新功能也不必经过内核编译过程，因此功能的扩展能力较强。总之，使用模块的原因在于方便Linux内核的扩展及系统的优化。

Linux内核的源代码是公开的，但是多数企业将Linux内核作为嵌入式系统或以提高利润为目的应用此内核，而这些企业通常不会公开他们开发的技术。Linux内核具有GPL许可证，应公开代码，这些企业的做法不符合Linux的宗旨。但是Linux内核还有LGPL许可证。

4.7.2 内核内部的模块管理

软件开发过程中定义了模块的意义，表示体现特定功能的独立要素。Linux内核的模块也表示体现某种功能的内核部分，表示ELF结构文件。

Linux内核为了模块特别定义了struct module结构体，且以列表形式管理模块。该结构体定义在include/linux/module.h上，其结构如下。

```
01  struct module
02  {
03      enum module_state state;      //模块状态
04      struct list_head list;
                              //内核内部管理模块的列表结构
05      char name[MODULE_NAME_LEN];   //不重复的模块名
06
07      const struct kernel_symbol *syms;
                              //支持外部引用的模块符号有关变量
08      unsigned int num_syms;
09      const unsigned long *crcs;
10
11      const struct kernel_symbol *gpl_syms;
                        //GPL Licence受限模块的支持外部引用的
12      unsigned int num_gpl_syms;    //模块符号有关变量
13      const unsigned long *gpl_crcs;
14
15      unsigned int num_exentries;          //其他表
16      const struct exception_table_entry *extable;
```

```
17
18      int (*init)(void);      //加载模块时初始化调用函数指针
19      void *module_init;          //初始化模块时分配的数据地址
20      void *module_core;          //模块程序核心上分配的内存地址
21      unsigned long init_size, core_size;
                                    //模块数据的大小和core大小
22      unsigned long init_text_size, core_text_size;
                                    //执行代码的大小
23      struct mod_arch_specific arch;       //体系结构模块值
24      int unsafe;                     //卸载模块后是否稳定
25      int license_gplok;      //表示模块是否具有GPL许可
26  #ifdef CONFIG_MODULE_UNLOAD
27      struct module_ref ref[NR_CPUS];      //模块引用计数
28      struct list_head modules_which_use_me;
                        //管理其他模块的引用的结构体变量
29      struct task_struct *waiter;
                        //为清除模块等待中的task管理结构体变量
30      void (*exit)(void);     //卸载模块时结束调用的函数指针
31  #endif
32  #ifdef CONFIG_KALLSYMS
33      Elf_Sym *symtab;
34      unsigned long num_symtab;
35      char *strtab;
36  #endif
37      void *percpu;
38      char *args;      //内核命令行要素
39  };
```

1. 模块加载过程

内核上加载模块时使用insmod命令。modprode执行insmod命令内核上加载模块，内核中的request_module也是最终执行insmod命令。该insmod命令执行下列过程。

（1）模块名包含“.o”或“.ko”时，可以从文件名获得模块名。否则直接默认为模块名，从/lib/modules/的子目录中查找对应模块的文件名，并读取该文件。

（2）求出保存文件代码、模块名及module结构体的内存空间。

（3）利用该信息调用create_module( )函数。该函数检查是否具有处理模块的权限，并利用find_module( )函数确认是否为加载的模块。若没有加载到内核上，调用vmalloc( )函数分配新模块的内存空间。分配的内存空间中初始化module结构体内容，并在其后复制模块名。然后加载到管理模块的内核模块列表中。最后返回模块分配到的内存起始地址。

（4）利用query_module( )函数求出内核符号表和内核中加载的其他模块符号表。

此时在query_module( )上指定 QM_MODULES、QM_INFO、QM_SYMBOL值获得必要的信息。其中，QM_MODULES用于获取内核中包含的模块名，QM_INFO用于获取各个模块的起始地址和大小。通过前面求出的模块信息，利用QM_SYMBOL求得实际内核符号表和加载到内核上的其他模块的符号表。

（5）利用内核符号表、模块符号表以及加载到内核上的当前模块的内存起始地址，重新

分配模块的程序代码地址。此时转换为模块引用的外部函数或变量的外部符号和全域符号所对应的逻辑地址offset。

（6）用户模式地址空间上分配内存空间，并在此复制模块结构体的内容和模块名以及重新分配的模块代码。

（7）分配模块结构体的init域函数地址和exit域函数地址。

（8）利用用户模式地址空间上分配的内存地址调用init_module( )函数。该函数执行类似于create_module( )函数的操作。先检查是否具有处理模块的权限，然后利用find_module( )函数或create_module( )函数查找位置，覆盖用户模式中设定的模块结构体内容。接着检查module结构体上的地址是否正确。然后在分配给模块的内存上复制剩余的内容。最后利用init域上定义的地址，调用包含在模块上的模块初始化函数。释放用户模式内存，并结束函数。

2. 从内核卸载模块的过程

内核中卸载模块时执行rmmod( )函数，该过程如下。

（1）模块名包含“.o”或“.ko”时，可以从文件名获得模块名。否则直接作为模块名。

（2）利用query_module( )函数求出模块有关信息，内核符号表和内核中加载的其他模块符号表。此时使用QM_MODULES、QM_SYMBOL值。若rmmod命令选项上设置–r，则利用QM_REFS获得其他模块信息。从而获得将要卸载的模块目录。

（3）利用将要卸载的模块目录调用delete_module( )函数。该函数利用系统求得内核符号表和加载到内核上的其他模块的符号表。此时检查是否具有卸载模块的权限，并利用find_module( )函数确认是否为加载的模块。还要检查该模块是否被其他模块引用。若没被引用，再检查是否在使用。当确认不使用时，调用module结构体的exit域上定义的函数。此时从内核的模块管理列表中卸载相应模块，并利用vfree( )函数释放内存。

（4）卸载列表中所有模块后，结束函数。

4.7.3 内核模式和内核内存地址空间

模块加载到内核上，作出内核的一部分运行。这表示模块定义的函数以内核模式运行，且模块内函数中引用的内存空间也就是内核空间。因此，内核中执行的函数可以完成其他内核源代码所做的所有事情。利用该特性可以开发内核模式中测试的程序。还可以实现硬件控制程序及内核内部的搜索功能。但是模块以内核模式在内核内存空间运行，因此不能执行强行中断及无限循环的延迟，不能阻止引用执行应用程序所获得的错误地址空间。

4.7.4 单向的符号引用

内核以结束编译的状态执行，因此使用固定的内部数据结构和变量以及函数。相反，模块最后才会被编译，从而插入内核中。因此模块不能修改运行中内核的数据结构。另外，不能清除已经存在的全局变量。考虑到这些特性，要根据已经定义的结构体形态创建模块。模块本身形成的结构体只能在模块中使用。编译的内核无法判断模块中加入的函数。因此，内

核只能使用静态包含的函数。相反，模块不仅使用自身的函数和变量，还可以使用内核提供的函数和变量（只限于允许外部引用的函数和变量）。内核不能引用模块中追加的函数，因此只能指定地址指针结构的域，并利用专门的登记函数引用模块提供的函数。这种特性称做单向符号引用。

4.7.5　模块和模块的引用

模块可以引用其他模块的函数或变量。因此创建具有复合功能的内核模块时，按照功能分类，并且形成层级。第19章模块之间的交互中介绍了详细内容。