Linux 内核模块开发

Linux 内核模块简介

模块是可以在运行时加入内核的代码，Linux 内核支持很多种模块，驱动程 序就是其中最重要的一种，甚至文件系统也可以写成一个模块的形式，然后加入

内核中，每一个模块有编译好的目标代码组成，可以使用insmod 命令将模块加

入正在运行的内核，也可以使用rmmod 命令将一个未使用的模块从内核中删除， 试图删除一个正在使用的模块，将是不允许的。

Linux 内核的整体结构已经非常庞大，而其包含的组件也非常多。怎么将需要的部分加入内核？

方法一：把所有需要的功能都编译到Linux 内核。

方法二：将需要的功能编译成模块，然后动态加入内核中。

如果采用方法一，将会导致两个问题。 1）将会时生成的内核体积很大， 2）如果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。为了解决 此问题，Linux 提供了一种机制，这种机制被称为模块（Module）。

模块具有的特点：

1）模块本身不被编译入内核镜像，从而控制了内核镜像的大小。 2）模块一旦被加载，它就和内核中的其他部分完全一样。 3）内核模块可以在系统运行期间动态扩展系统功能而无须重新启动系统，更 无须为这些新增的功能重新编译一个新的系统内核镜像，节省了开发人员的时 间。

linux内核模块的组成

头文件

驱动模块会使用内核中的许多函数，所以需要包含必要的头文件。有两个头文件是所有驱动模块都必须包含的。这两个头文件：

```
#include<linux/module.h>
#include<linux/init.h>
```

其中module.h 文件包含了加载模块时需要使用的大量符号和函数定义，init.h 包含了模块加载函数和模块卸载函数的宏定义。

模块参数（可选）

模块参数是驱动模块加载时，需要传递给驱动模块的参数。如果一个驱动模块需要完成两个功能，那么就可以通过模块参数选择使用哪一种功能

模块加载函数

一般在这里实现初始化工作（如申请内存空间）

模块加载函数是模块加载时，需要执行的函数: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，完成模块的相关的初 始化工作。

模块卸载函数（必须）:

跟模块加载函数做相反的事情（如释放空间）

模块卸载函数是模块卸载时，需要执行的函数；当执行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。

模块许可申明（GPL）（必须）:

模块许可申明表示模块受内核支持的程度。有许可权的模块会更受到开发人员的重视。需要使用 MODULE_LICENSE 表示该模块的许可权限。内核可以识别的许可权限如下：

- MODULE_LICENSE("GPL"); //任一版本的GNU 公共许可权
- MODULE_LICENSE("GPL v2"); //GPL 版本2许可权
- MODULE_LICENSE("GPL and additional rights"); //GPL 及其附加许可权
- MODULE_LICENSE("Dual BSD/GPL"); //BSD/GPL 双重许可权
- MODULE_LICENSE("Dual MPL/GPL"); //MPL/GPL 双重许可权
- MODULE_LICENSE("Proprietary"); //专用许可权

如果一个驱动没有包含任何许可权，那么就会认为是不符合规范的。这时，内核加载这种模块时，会受到内核加载了一个非标准模块的警告。

不过对于任何一个驱动只要加了一个GPL 权限基本可以加载成功，GPL 权限表示通用公共许可证。GNU 通过公共许可证可以保证你有发布自由软件的自由；保证你能收到源码程序或者在你需要时能用到它；保证你能修改软件或将它的一部分用于新的自由软件。

模块编译

- 准备一个内核
- 保证此内核必须至少被编译过一次，保证此内核是在板子上运行的内核
- 编写Makefile

  Makefile解释

```
ROOTFS_DIR = /home/NFS
CROSS_COMPILE = arm-none-linux-gnueabi-
MODULE_NAME = hello
#APP_NAME = hello

ifeq ($(KERNELRELEASE),)
KERNEL_DIR =/home/linux/linux-ok210
CUR_DIR =$(shell pwd)

all :
        make -C $(KERNEL_DIR) M=$(CUR_DIR) modules
#       $(CROSS_COMPILE)gcc  $(APP_NAME).c -o  $(APP_NAME)
clean :
        make -C $(KERNEL_DIR) M=$(CUR_DIR) clean
        rm -rf $(APP_NAME)

install :
        cp -rvaf *ko  $(APP_NAME)  $(ROOTFS_DIR)/
else

obj-m = $(MODULE_NAME).o
```

模块操作方法

加载模块：insmod xxx.ko 卸载模块：rmmod xxx 出现如下问题： rmmod: chdir(/lib/modules): No such file or directory 解决方法： mkdir /lib/modules/2.6.35.5 查看当前状态的模块：lsmod

Linux 内核模块的文件格式

以内核模块形式存在的驱动程序，比如hello_world.ko，其在文件的数据组织形式上是ELF 格式，跟具体地说，内核模块是一中普通的可重定位的目标文件。通过file 命令可以查看hello_world.ko 文件，如下所示：

```
# file hello.ko
hello.ko: ELF 32-bit LSB  relocatable, ARM, EABI5 version 1 (SYSV),
BuildID[sha1]=0dabe01e4add1bfa27d561d6ba8f1e6335e21d5d, not stripped
```

上图描述的是ELF 目标文件的总体结构。图中省去了一些繁琐的结构，把最重要的结构提取出来。

- ELF Header 头位于文件的最前部。其包含了描述整个文件的基本属性，例如ELF 文件版本、目标机器型号、程序入口地址等。
- .text 表示代码段，存放文件的代码部分
- .data 表示数据段，存放已经初始化的数据等
- .Section Table 表描述了ELF 文件包含的所有段的信息，例如每个段的段名、段的长度、在文件中的便宜、读写权限及段的其他属性
- .symtab 表示符号表。符号表是一种映射函数到真实内存地址的数据结构。其就像一个字典，其记录了再编译阶段，无法确定地址的函数。该符号表将在模块文件加载阶段，有系统赋予真实的内存地址。

Linux 内核模块的加载函数

Linux 内核模块加载函数一般以__init 表示说明，典型的模块加载函数的形式如代码如下所示：

```
static int __init hello_init(void)
{
/*初始化代码*/
}
module_init(hello_init);
```

模块加载函数必须以"module_init(函数名)"的形式指定。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一种非常好的习惯，因为只有这样，用户程序才可以用perror 等方法把他们转换成有意义的错误信息字符串。在Linux 中，所有标示为init 的函数在链接的时候都放在.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个函数指针调用这些init 函数，并初始化完成后，释放init区段（包括.init.text、.initcall.init 等）。

Linux 内核模块的卸载函数

Linux 内核模块卸载函数一般以__exit 标识标识表明，典型的模块卸载函数的形式如下面代码所示：

```
static void __exit hello_exit(void)
{
/*释放代码*/
}
module_exit(hello_exit);
```

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
- 如模块加载函数动态申请了内存，则模块卸载函数应该释放内存
- 如模块加载函数申请了硬件资源（中断、DMA 通道、IO 内存）的占用，则模块卸载函数应该释放这些硬件资源
- 若模块加载函数开启了硬件，则模块卸载函数中一般要关闭之。

Linux 内核模块的参数

在用户态下编程可以通过main()的来传递命令行参数，而编写一个内核模块则通过

`module_param(参数名，参数类型，参数读/写权限)`

来实现内核模块参数的传递。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 *string ="this is a kernel module";
static int num =40000;
module_param(num,int,S_IRUGO);
module_param(string,charp,S_IRUGO);
```

在装载内核模块时，用户可以向内核模块传递参数，形式为"insmod (或modprobe) 模块名参数名=参数值"。module_param 宏是Linux 2.6 内核中新增的， 该宏被定义在include/linux/moduleparam.h 文件中，具体定义如下：

```
#define module_param(name, type, perm) \
module_param_named(name, name, type, perm)
```

其中使用了3 个参数:要传递的参数变量名, 变量的数据类型, 以及访问参数的权限。

最后的perm 字段是一个权限值,表示此参数在sysfs 文件系统中所对应的文件节点的属性。应当使用 中定义的值. 这个值控制谁可以存取这些模块参数在sysfs 中的表示.当perm 为0时，表示此参数不存在sysfs 文件系统下对应的文件节点。否则, 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, 带有给定的权限。 权限在include/linux/stat.h 中有定义，比如：

```
#define S_IRWXU 00700
#define S_IRUSR 00400
#define S_IWUSR 00200
#define S_IXUSR 00100
#define S_IRWXG 00070
#define S_IRGRP 00040
#define S_IWGRP 00020
#define S_IXGRP 00010
#define S_IRWXO 00007
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001
```

使用S_IRUGO 作为参数可以被所有人读取, 但是不能改变;

S_IRUGO|S_IWUSR 允许root 来改变参数.

注意, 如果一个参数被sysfs 修改,你的模块看到的参数值也改变了, 但是你的模块没有任何其他的通知. 你应当不 要使模块参数可写, 除非你准备好检测这个改变并且因而作出反应. 参数类型可以使byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool 或invbool（布尔的反）

Linux 内核模块导出符

Linux2.6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，这些宏放在include/linux/module.h 文件中

```
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EXPORT_SYMBOL_GPL只适用于GPL 许可权限的模块。下面代码给出了一个导出整数加、减法运算函数符号的内核模块的例子（这些导出符号毫无实际意义，仅仅是为了演示）。

```
//文件名为add_sub.c
#include<linux/init.h>
#include<linux/module.h>
MODULE_LICENSE("GPL");
int add_integar(int a,int b)
{
return a+b;
}
int sub_integar(int a,int b)
{
return a-b;
}
EXPORT_SYMBOL(add_integar);
EXPORT_SYMBOL(sub_integar);
```

看主程序hello.c如下：

```

#include<linux/init.h>
```

```c
#include<linux/module.h>
#include"add_sub.h"

static int a =3;
static int b =2;
static int c =1;
static char *str ="hello Allison";

/*1.实现模块加载函数,执行insmod xxx.ko就会执行此函数*/
static int __init hello_init(void)
{
    printk(KERN_INFO "%s()-%d:\n",__func__,__LINE__);
    if(c>0){
        printk(KERN_INFO "result =%d\n",add_int(a,b));
    }else{
        printk(KERN_INFO "result =%d\n",sub_int(a,b));
    }
    return 0;
}

/*2.实现模块卸载，执行rmmod xxx就会执行此函数*/
static void __exit hello_exit(void)
{
    printk(KERN_INFO "%s()-%d:\n",__func__,__LINE__);
}

/*param1:变量
  *param2:类型
  *param3:权限
  */
module_param(a,int,S_IRWXU);
module_param(b,int,S_IRWXU);
module_param(c,int,S_IRWXU);
module_param(str,charp,S_IRWXU);


/*3.申明加载函数与卸载函数*/
module_init(hello_init);
module_exit(hello_exit);

/*4.权限许可声明*/
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Allison");
MODULE_VERSION("V0.1");
MODULE_DESCRIPTION("this is a simple module");
```

Makefile 修改：

```
ROOTFS_DIR = /home/NFS
CROSS_COMPILE = arm-none-linux-gnueabi-
MODULE_NAME = hello
#APP_NAME = hello
```

```
ifeq ($(KERNELRELEASE),)
KERNEL_DIR =/home/linux/linux-ok210
CUR_DIR =$(shell pwd)

all :
        make -C $(KERNEL_DIR) M=$(CUR_DIR) modules
#       $(CROSS_COMPILE)gcc  $(APP_NAME).c -o  $(APP_NAME)
clean :
        make -C $(KERNEL_DIR) M=$(CUR_DIR) clean
        rm -rf $(APP_NAME)

install :
        cp -rvaf *ko  $(APP_NAME)  $(ROOTFS_DIR)/
else

obj-m += $(MODULE_NAME).o add_sub.o
obj-m += add_sub.o

endif
```

在开发板上,先执行

`$insmod add_sub.ko`

然后分别执行以下不同的传入参数值：

```
[root@S5PV210/]$insmod hello.ko c=0
[ 4605.345606] hello_init()-13:
[ 4605.347005] result =1
[root@S5PV210/]$rmmod hello
[ 4625.913662] hello_exit()-25:
rmmod: module 'hello' not found
[root@S5PV210/]$insmod hello.ko c=3
[ 4634.465337] hello_init()-13:
[ 4634.466735] result =5
```

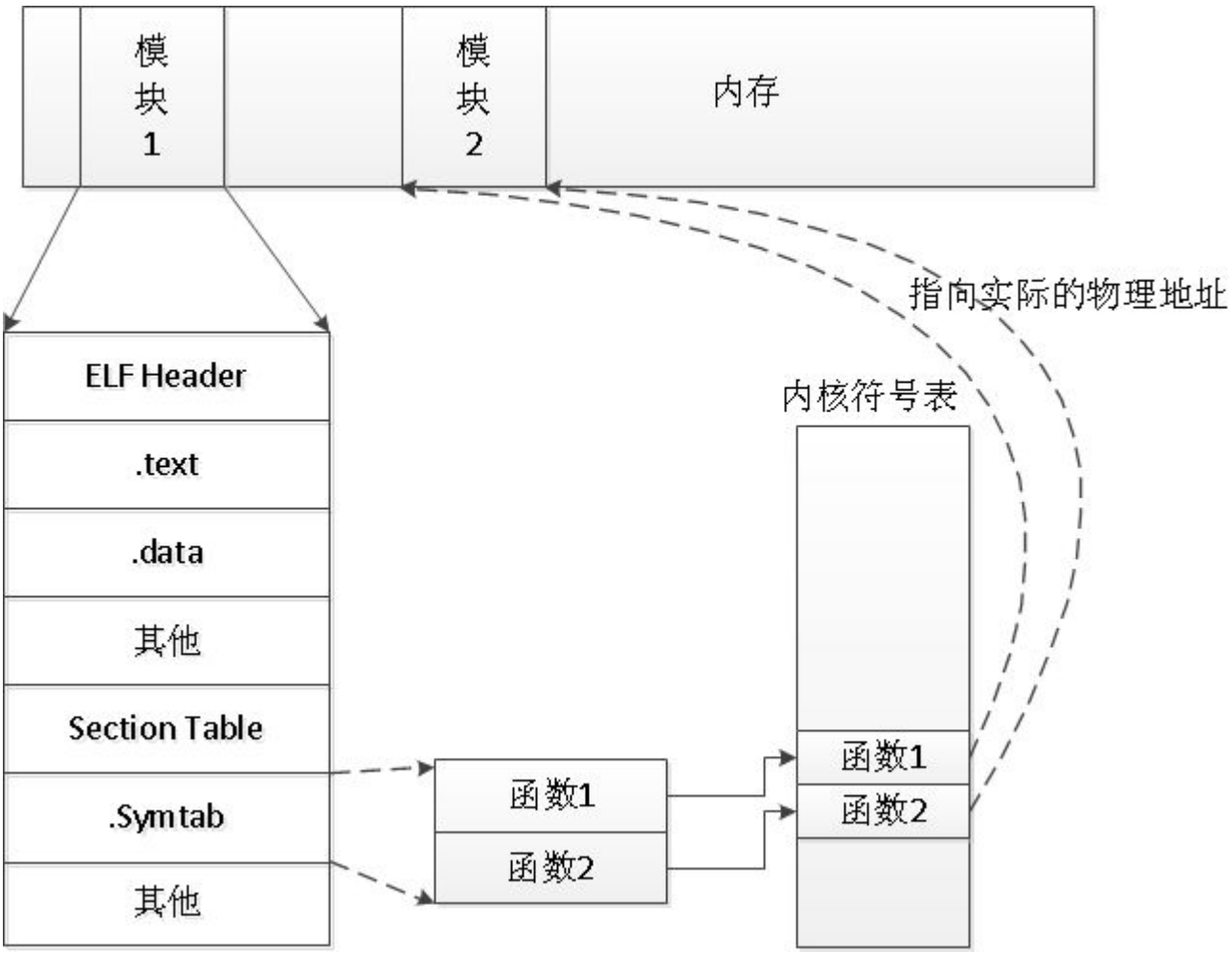
可以看到，不同的传入参数值，得到的结果是不一样的。

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(Allison);
MODULE_DESCRIPTION(A sample hello word module);
MODULE_VERSION(V0.1);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(hello);
```

Linux 模块之间的通信



模块2加载过程：

1. 使用insmod 2.ko 加载模块2
2. 内核为模块2分配空间，然后将模块的代码和数据转入分配内存中。
3. 内核发现符号表中(symtab)有函数1，函数2可以导出，于是将其内存地址记录在内核符号表中。

模块1在加载进内核时，系统会执行以下操作：

1. insmod 命令会为模块分配空间，然后将模块的代码和数据转入内存中
2. 内核在模块1的符号表（symtab）中发现一些未解释的函数。上图中这些未解释的好函数时“函数1”、“函数2”，这些函数位于模块2的代码中。所以模块会通过内核符号表，查到相应的函数，并将函数地址填到模块1的符号表中。通过模块1加载过程后，模块1就可以使用模块2提供的“函数1”和“函数2”了。