一、 模块化编程思想

linux 内核功能非常的强大，包含了很多的组件，而对于工程师而言，后期有两种方法将需要的功能包含进内核当中。

1、将所有的功能都直接编译进 Linux 内核；

优缺点：不会有版本不兼容的问题，不需要进行严格的版本检查；但镜像文件大，要在现有的内核中添加新的功能，则要编译整个内核。

2、将功能编译成模块，在需要的时候以模块的方式将某个功能注册到内核中；

优缺点：灵活，模块本身不编译进内核，从而控制了内核的大小；模块一旦被加载，将和其它部分完全一样；但兼容性不好（后期发布，模块必须添加到内核），会造成内存的利用率比较低。

二、 单模块编程

1、 命令

#insmod xxx.ko　　//将模块添加进内核

#rmmod xxx.ko　　//将模块从内核卸载

#lsmod xxx.ko　　//查看已安装到内核的模块

#modprobe xxx.ko　　//载入指定的个别模块，或是载入一组相依赖的模块。

modprobe 会根据 depmod 所产生的依赖关系，决定要载入哪些模块。若在载入 过程中发生错误，在 modprobe 会卸载整组的模块。依赖关系是通过读取 /lib/modules/2.6.xx/modules.dep 得到的。而该文件是通过 depmod 所建立。

#modinfo XXX.ko　　//查看模块信息。

#tree －a　　//查看当前目录的整个树结构

2、 加载模块函数模板

```
static int __init   XXX(void)
{
        xxxxxxxx
        return 0;
}
```

moudle_init(XXX);　　// 当执行#insmod 或者#modprobe 时调用，可以看作模块的入口；

当#insmod xxx.ko 时会调用 module_init 所修饰的 XXX 函数来完成一些初始化操作。

（ xxx 在写代码时要根据实际模块进行修改）

Linux 内核的模块加载函数一般用__init 标识声明，用于告诉编译器相关函数或变量仅用于初始化，初始化结束后就释放这段内存。

XXX 函数的返回值为一个整形的数，如果执行成功，则返回 0，初始化失败时则返回错误编码，Linux 内核当中的错误编码是负值，在<linux/errno.h>中有定义。

3、卸载模块函数模板

```
static void __exit XXX_exit(void)
{
        xxxxxxxxx
}
```

module_exit(xxx 函数);　　//执行#rmmod 时调用，指定卸载模块执行的函数，

在用 rmmod 或 modprobe 命令卸载模块时，该函数被执行。完成与加载相反的工作。

4、模块的开源许可和声明

模块的开源许可协议：MODULE_LICENSE("GPL");　//遵循 GPL 开源协议

模块的声明（非必须）：

```
MODULE_AUTHOR          // 声明作者
MODULE_DESCRIPTION     // 对模块简单的描述
MODULE_VERSION         // 声明模块的版本
MODULE_ALIAS           // 模块的别名
MODULE_DEVICE_TABLE // 告诉用户空间这个模块所支持的设备
```

注： MODULE_xxx 可以写在模块的任何地方（但必须在函数外面），习惯上写在模块的最后

5、Makefile 模板：

```
KERN_DIR = /root/linux-3.5
all:
    make -C $(KERN_DIR) M=`pwd` modules
clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order
obj-m += first_module.o
```

第一行： 指定内核源码路径， 需要内核源码路径的 makefile；(实际内核源码路径，可修改)；

第三行： 编译.c 文件生成.ko；

最后一行： 指定.c 文件名字；(实际用到的 c 文件名)；

#make 命令，生成.ko 文件，拷贝到文件系统执行

```
insmod xxx.ko       //加载模块，module_init 指定的函数就会被调用
rmmod xxx.ko        //卸载模块，module_exit 指定的函数就会被调用
lsmod               //查看当前装载的模块
```

注意：如果卸载提示错误没有/lib/modules 和 3.5.0-FriendlyARM，

执行：# mkdir /lib/modules -p

mkdir /lib/modules/3.5.0-FriendlyARM -p

6、模块的卸载函数和模块加载函数实现相反的功能，主要包括：

1） 若模块加载函数注册了 XXX，则模块卸载函数注销 XXX

2） 若模块加载函数动态分配了内存，则模块卸载函数释放这些内存

3） 若模块加载函数申请了硬件资源，则模块卸载函数释放这些硬件资源

4） 若模块加载函数开启了硬件资源，则模块卸载函数一定要关闭这些资源

三、多模块编程(模块间有依赖关系)

一个模块调用另一个模块里面的函数。

```
EXPORT_SYMBOL(hello1);          //声明 hello1 可以被外界模块调用
EXPORT_SYMBOL(hello2);          //声明 hello2 可以被外界模块调用
extern void hello1();
extern void hello2();           //声明外部函数
```

注意加载和卸载的顺序。

四、向模块传递参数

模块可以制定参数，在安装模块时向模块传递用户自定义的参数进行设置，这一过程通过 module_param()来实现，经过 module_param()声明过的变量如果在安装模块时没有设置参数的值，参

数保持原有定义的缺省值。

module_param(name,type,perm);　　//在加载模块时或者模块加载以后传递参数给模块

module_param(名字，　类型，　权限);

a、数据类型的取值情况：bool：布尔型；inbool：布尔反值；short: 短整型；ushort：无符号短整型；charp: 字符指针（相当于 char*,不超过 1024 字节的字符串）；int：整型；uint：无符号整型；long：长整型；ulong: 无符号长整型

b、perm 表示此参数在 sysfs 文件系统中所对应的文件节点的属性，其权限在 include/linux/stat.h 中有定义。它的取值可以用宏定义，也可以有数字法表示；

宏定义有：

#define S_IRUSR 00400　　//文件所有者可读

#define S_IWUSR 00200　　//文件所有者可写

#define S_IXUSR 00100　　//文件所有者可执行

#define S_IRGRP 00040　　//与文件所有者同组的用户可读

#define S_IWGRP 00020

#define S_IXGRP 00010

#define S_IROTH 00004　　//与文件所有者不同组的用户可读

#define S_IWOTH 00002

#define S_IXOTH 00001

除外，内核中还有以下定义：

#define S_IRWXUGO (S_IRWXU|S_IRWXG|S_IRWXO)

#define S_IALLUGO (S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)

#define S_IRUGO (S_IRUSR|S_IRGRP|S_IROTH)

#define S_IWUGO (S_IWUSR|S_IWGRP|S_IWOTH)

#define S_IXUGO (S_IXUSR|S_IXGRP|S_IXOTH)

数字法：1 表示执行权限，　2 表示写入权限，　4 表示读取权限。　一般用 8 进制表示即可；

例：　0664。从左向右看，第一位的 0 表示八进制的意思，第二位的 6 表示文件所有者的权限为可读可写，第三位的 6 表示文件同组用户的权限为可读可写，第四位的 4 表示文件其他用户的权限为只读。

例：int num;

```
char *p="hello";
module_param(num,int,S_IRUGO);
module_param(p,charp,S_IRUGO);
insmod first_module.ko num=5 p="hello" a=9,8,7,6,5,4,3
```

内核数组参数：

module_param_array(名字,类型，　数组元素个数指针,权限);　　//常用来传递数组

原型：module_param_array(name, type, nump, perm);

参数：name：模块参数的名称

type：　模块参数的数据类型

nump：　数组元素个数指针

perm：　模块参数的访问权限

例：staticintfish[10];

```
static int nr_fish;
module_param_array( fish, int, &nr_fish, 0664);
```

nr_fish:保存最终传递数组元素个数，不能大于 10 个

五、多文件编译为一个模块

多个*.c 文件编译生成一个模块驱动*.ko， 只需要一个文件当中进行模块初始化声明，其他文件当中只编写调用函数即可，关键点在于对几个文件编译的 Makefile 编写。

例：Makefile 编译两个文件 hellodrv0.c 和 hellodrv1.c， 模块初始化函数在 hellodrv1.c 当中，编译选项 obj-m := hellodrv.o 新指定的模块名称， hellodrv-objs = hellodrv0.o hellodrv1.o 说明这个模块包含的 obj 文件， 注意这里 obj-m 编译选项当中指定的名字不能和 objs 源文件当中的名字相同。

```
KERN_DIR = /xyd/linux-3.5

all:
    make -C $(KERN_DIR) M=`pwd` modules
clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order
obj-m := hellodrv.o
hellodrv-objs = hellodrv0.o hellodrv1.o
```

名字不能一样，否则会编译混淆产生错误

六、printk 函数的使用

printk 相当 printf 的孪生姐妹， printf 运行在用户态， printk 则在内核态被人们所熟知。printk 是在内核中运行的向控制台输出显示的函数， 可以指定输出的优先级。 printk 的输出一共有 8 种日志级别，如果没有指定，则默认为 DEFAULT_MESSAGE_LOGLEVEL（这个默认级别一般为<4>,即与 KERN_WARNING 在一个级别上）。

printk 的日志级别定义如下（ 在 include/linux/kernel.h 中） ：

```
#define KERN_EMERG      0/*紧急事件消息， 系统崩溃之前提示，表示系统不可用*/
#define KERN_ALERT      1/*报告消息， 表示必须立即采取措施*/
#define KERN_CRIT       2/*临界条件， 通常涉及严重的硬件或软件操作失败*/
#define KERN_ERR        3/*错误条件， 驱动程序常用 KERN_ERR 来报告硬件的错误*/
#define KERN_WARNING    4/*警告条件， 对可能出现问题的情况进行警告*/
#define KERN_NOTICE     5/*正常但又重要的条件， 用于提醒*/
#define KERN_INFO       6/*提示信息， 如驱动程序启动时， 打印硬件信息*/
#define KERN_DEBUG      7/*调试级别的消息*/
```

使用方法：

```
printk(KERN_INFO"this is printk!!\n");  等同于：printk(<4>"this is printk!!\n");
printk(KERN_EMERG "hello!!!!\n");       //如果不写默认级别为 4， 为 5 终端就不显示了
```

附：模块编译通用 Makefile 模板

使用说明：

1）直接 make 生成的模块和目录名相同，即目录名.ko；

2）也可以指定模块名： make mn=module_name，这样可以生成 module_name.ko；

编译 app 程序：app 程序文件存放在当前目录的 app 文件夹中，支持多文件编译成一个 app：

1）用法 1：make app 生成 “目录名_app” 的可执行文件；

2) 用法 2：make app an=an_name 生成“ an_name_app”的可执行文件；

使用前注意修改编译器，如果是 X86，则不用修改，也可以通过 make 传递参数如： make app cc=arm-linux-

```
#指定编译器
cc :=
ifeq ($(an),)
an=$(shell basename $(shell pwd))_app
endif

#获取 app 文件列表
APP_SRC :=$(wildcard ./app/*.c)
ifeq ($(mn),)
mn=$(shell basename $(shell pwd))
endif
MODULE_NAME=$(mn)
SRC :=$(wildcard *.c ./sub/*.c)
DIR :=$(notdir $(SRC))
OBJ :=$(patsubst %.c,%.o,$(DIR))
obj-m := $(MODULE_NAME).o
$(MODULE_NAME)-objs := $(OBJ)
obj-m := $(MODULE_NAME).o
$(MODULE_NAME)-objs := $(OBJ)

#内核源码路径
KDIR:=/lib/modules/$(shell uname -r)/build
#KDIR := /root/work/source/4412/linux_kernel/FriendlyARM/linux-3.5
all:
	@echo -ne "MODULE_NAME:$(MODULE_NAME)\nSRC=$(SRC)\nOBJ=$(OBJ)\n"
	 make -C $(KDIR) M=$(PWD) SRC=$(wildcard *.c ./sub/*.c) MODULE_NAME=$(mn)
DIR=$(notdir $(SRC)) OBJ=$(patsubst %.c,%.o,$(DIR)) modules
	@echo -ne "MODULE_NAME:$(MODULE_NAME)\nSRC=$(SRC)\nOBJ=$(OBJ)\n"
	@rm -rf $(mn).ko.unsigned ?.*.ko modules.order *.bak *.~ *.pre *.o .???* *.mod.o *~ *.mod.c
*.symvers *.markers .tmp_versions *.unsigned

#清除编译文件和 目标文件，备份文件等
.PHONY:clean
clean:
	@rm -rf $(an) ?*app $(mn).ko.unsigned *.ko ?.*.ko modules.order *.bak *.~ *.pre *.o .???*
*.mod.o *~ *.mod.c *.symvers *.markers .tmp_versions *.unsigned

#编译 app 文件
.PHONY:app
app:
	$(cc)gcc $(APP_SRC) -o $(an)
```