Linux模块编程

1.1 模块学习什么

1.认识什么是模块？跟我学习的驱动有什么关系？

2.熟悉模块的安装，卸载，查看

3.熟悉模块的基本框架

4.熟悉模块的编程方法

1.2 内核模块概述

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。

1.模块本身不被编译入内核映像，从而控制了内核的大小。

2.模块一旦被加载，它就和内核中的其他部分完全一样。

1.3 认识模块

我们在配置内核的时候，在配置菜单看到有一种选项可以选择为<M> <\*> < > 。

其中，<M>表示编译成模块，即modules，这个便于选项编译的代码不编译进内核zImage，而是编译成一个单独的文件，通常为后缀.ko（2.6以上内核版本是.ko，2.6内核之前是.o）文件，那么我们可以像软件一样选择安装和卸载这些.ko模块文件。

<\*>表示编译进内核，就是将这一段模块代码编译到了zImage镜像文件去了，在内核启动的时候自动安装执行我们的模块代码，这样做类似于一些安装系统的时候自带的驱动，在安装系统的时候就已经安装好了。

< > 表示没有选择，不做任何事情。

所以，我们所说的模块就是linux下的那些.ko文件。

1.3.1 tiny4412提供的模块测试程序

在我们学习的Tiny4412提供的 Linux3.5 源码中有一项模块测试选项，如下：

         Tiny4412module sample.

         Symbol:TINY4412\_HELLO\_MODULE [=m]

         Type: tristate

         Prompt:Tiny4412 module sample

         Definedat drivers/char/Kconfig:49

         Dependson: ARCH\_EXYNOS4 [=y]

Location:

          -> DeviceDrivers

                   ->Character devices

注意：因为有依赖项，所以它的依赖必须被选上，即ARCH\_EXYNOS4 [=y]。

 CONFIG\_ARCH\_EXYNOS4:

 Samsung EXYNOS4 SoCs based systems

 Symbol: ARCH\_EXYNOS4[=y]

 Type  :boolean

 Prompt: SAMSUNGEXYNOS4

  Defined at arch/arm/mach-exynos/Kconfig:14

  Depends on: ARCH\_EXYNOS [=y]

  Location:

     -> SystemType

       -> SAMSUNGEXYNOS SoCs Support

  Selects: HAVE\_SMP [=y] && MIGHT\_HAVE\_CACHE\_L2X0 [=y]

这个测试文件在drivers/char/目录下的tiny4412\_hello\_moduls.c。当把它选择为<M>之后，编译就会在同级目录下生成.ko文件。

编译命令：make modules

在 Linux 系统中，几乎所有驱动都可以编译模块的形式。

总之，模块代码我们可以在系统启动后再安装（zImage文件中并不包含该选项对应的C代码）。也可以编译到内核，像内核配置菜单中选择为y的选项，对应的C代码会被编译到zImage文件中。

这里记住一句话：linux下模块不一定是驱动，但是驱动肯是一种模块。

示例：编译成模块

1）在linux内核内核配置菜单makememuconfig上把模块测试选项为M，然后退出保存。

2）编译成模块，如下：

[root@localhost linux-3.5]# make modules

scripts/kconfig/conf--silentoldconfig Kconfig

 CHK     include/linux/version.h

 CHK    include/generated/utsrelease.h

make[1]: “include/generated/mach-types.h”是最新的。

 CALL    scripts/checksyscalls.sh

 CC [M]  crypto/ansi\_cprng.o

 CC [M] drivers/char/tiny4412\_hello\_module.o

 CC [M] drivers/scsi/scsi\_wait\_scan.o

 Building modules, stage 2.

 MODPOST 3 modules

 CC      crypto/ansi\_cprng.mod.o

 LD [M]  crypto/ansi\_cprng.ko

 CC     drivers/char/tiny4412\_hello\_module.mod.o

 LD [M] drivers/char/tiny4412\_hello\_module.ko

 CC     drivers/scsi/scsi\_wait\_scan.mod.o

 LD [M] drivers/scsi/scsi\_wait\_scan.ko

[root@localhost linux-3.5]#

以上就会在drivers/char/生成了tiny4412\_hello\_module.ko文件。

然后我们可以查看一下它的存在：

[root@localhost linux-3.5]# cd drivers/char/

[root@localhost char]# ls tiny4412\_hello\_module.ko -l

-rw-r--r--. 1root root 29062 9月 25 11:26 tiny4412\_hello\_module.ko

[root@localhost char]#

有了.ko文件，那么我们可在以外部独立的来加载到内核，怎么加载呢？又怎么卸载呢？

Linux系统提供专业的命令来完成这样的事情，下面来介绍这几个命令。

1.3.2 Linux模块安装，卸载，查看

Linux下模块提供有安装，卸载，查看安装了那个模块以及查看模块信息等命令。

1.3.1、模块相关命令

|  |  |
| --- | --- |
| insmod <file name.ko> | 安装模块进内核 |
| rmmod <file name> | 卸载指定的模块 |
| lsmod | 查看当前系统安装了哪些模块 |
| modinfo <file name> | 查看模块信息 |

1.insmod   -- 安装模块

示例：

[root@JUNJIA /home]# insmod tiny4412\_hello\_module.ko

[ 140.030000] Hello, Tiny4412 module is installed !

[root@JUNJIA /home]#

2.lsmod    -- 列出当前系统已经安装的模块及模块间的依赖关系（不包含被编译的内核的模块代码）

示例：

[root@JUNJIA /home]# lsmod

tiny4412\_hello\_module 773 0 - Live 0xbf000000

第1列：模块名

第2列：模块大小

第3列：被引用多少次（本模块的代码被多少个模块调用）

第4列：被哪个模块引用

说明：

lsmod 命令实际上读取并分析/proc/modules文件。

在安装模块的时候我们可能会遇到这样的问题，模块版本和要装的linux内核版本不同会出现问题。

比如我修改了编译模块的内核版本号，然后编译查看模块信息：

         [root@localhostlinux-3.5]# modinfo drivers/char/tiny4412\_hello\_module.ko

         filename:       drivers/char/tiny4412\_hello\_module.ko

         license:        GPL

         depends:

         intree:         Y

         vermagic:       3.5.0-FriendlyARM\_JUNJIASMP preempt mod\_unload ARMv7 p2v8

         [root@localhostlinux-3.5]#

安装会出现以下问题：

         [root@JUNJIA/home]# insmod tiny4412\_hello\_module.ko

         [3212.730000] tiny4412\_hello\_module: version magic '3.5.0-FriendlyARM\_JUNJIASMP preempt mod\_unload ARMv7 p2v8 ' should be '3.5.0-FriendlyARMSMP preempt mod\_unload ARMv7 p2v8 '

         insmod:can't insert 'tiny4412\_hello\_module.ko': invalid module format

         [root@JUNJIA/home]#

这个错误是说模块 版本号是3.5.0-FriendlyARM\_JUNJIA，而内核版本号是3.5.0-FriendlyARM，不匹配不能进行加载。那么我就只能

结论：模块版本和内核版本不同是不能安装模块的。

3.rmmod    -- 卸载模块（编译在内核的模块代码不可以移除）

示例：

[root@JUNJIA /home]# rmmodtiny4412\_hello\_module.ko

[root@JUNJIA /home]# lsmod

tiny4412\_hello\_module 773 0 - Live 0xbf000000 (O)    //发现没有卸载掉

[root@JUNJIA /home]# rmmodtiny4412\_hello\_module  //不加后缀

[  231.660000] Good-bye, Tiny4412 module was removed! //打印卸载函数里的信息

rmmod: module 'tiny4412\_hello\_module' not found//这个提示是busybos存在的BUG

[root@JUNJIA /home]#

注意：使用就一些的busybox制作的根文件系统的rmmod命令不能带扩展名，带了不能卸载；但是X86是可以带的，也可以不带。现在的新版本根文件系统制作工具busybox修复了这个功能，带.ko也能卸载。

有时候我们在卸载模块会遇到出错等问题，下面是一个常见的问题。

如果busybox下makemenuconfig配置如下：

                   Symbol:MODPROBE\_SMALL [=y]

                   Prompt:Simplified modutils

                   Definedat modutils/Config.in:15

                 Location:

                            ->Linux Module Utilities

模块安装后如果需要卸载，需要在lib下创建modules目录，然后再创建一个以内核版本名命名的子目录。

格式：mkdir/lib/modules/内核版本号

示例：

mkdir/lib/modules/3.5.0-FriendlyARM

内核版本号：(1)可以通过 uname -r 得到；(2)也可以通过查看内核配置，子版本是我们在配置内核的时候写上去的。如下：

         Symbol:LOCALVERSION [=-FriendlyARM]

           │ Type  : string

           │ Prompt: Local version - append to kernel release

           │   Defined at init/Kconfig:91

           │   Location:

           │     -> Generalsetup

那么内核版本号是内核主版本号（3.5.0）接以上的本地版本号（-FriendlyARM）构成。

我们可以有根治的方法，把Simplified modutils选项取消，然后选择出现的lsmod，insmod，rmmod命令，重新编译安装。

4.modinfo  -- 查看模块的基本信息

示例：

[root@localhost 桌面]# modinfo dm\_mod

         filename:     /lib/modules/2.6.32-279.el6.i686/kernel/drivers/md/dm-mod.ko

         license:        GPL

         author:         Joe Thornber<dm-devel@redhat.com>

         description:    device-mapper driver

         srcversion:     55E98DC47312D5D1A682B77

         depends:

         vermagic:       2.6.32-279.el6.i686 SMP mod\_unloadmodversions 686

         parm:           major:The major number of the devicemapper (uint)

说明：

filename：                          模块路径（相对的）

license：                             模块发布许可协议

author：                             模块作者

description：                     模块是功能描述

srcversion：                       源码版本 --- 不是由编程者决定，不可以控制

depends：                          依赖

vermagic：                        内核的版本魔数，简单就是一版本的ID,由内核源码决定，不可控制

parm：                                模块可以传递的参数介绍

使用这个命令可能遇到问题：

[root@JUNJIA /home]# modinfo tiny4412\_hello\_module.ko

modinfo: can't open'/lib/modules/3.5.0-FriendlyARM/modules.dep': No such file or directory

解决办法：

cd /lib/

mkdir modules

cd modules/

mkdir <linux version number>

cd <linux version number>

cp \*.ko ./

depmod

mv modules.dep.bb modules.dep

modinfo \*.ko

1.4模块程序框架

我们看一下tiny4412\_hello\_modules.c，内容如下：

         #include<linux/kernel.h>

         #include<linux/module.h>

         staticint \_\_init tiny4412\_hello\_module\_init(void)

         {

             printk("Hello, Tiny4412 module isinstalled !\n");

             return 0;

         }

         staticvoid \_\_exit tiny4412\_hello\_module\_cleanup(void)

         {

             printk("Good-bye, Tiny4412 module wasremoved!\n");

         }

         module\_init(tiny4412\_hello\_module\_init);

         module\_exit(tiny4412\_hello\_module\_cleanup);

         MODULE\_LICENSE("GPL");

以上是一个模块最基本的框架，我们来说明这个框架的组成都有哪些？

1.必须的头文件

#include <linux/kernel.h>

#include <linux/module.h>

包含了我们这个基本模块所用到的函数声明。

2.模块的初始化（加载）函数（必须）

以下模块初始化函数，insmod 模块时会执行，如果是编译到内核，在系统启动阶段会自动执行。典型的模块初始化函数形式入下：

static int \_\_init tiny4412\_hello\_module\_init(void)

         {

                 /\* 初始化代码 \*/

                   printk("Hello,Tiny4412 module is installed !\n");

                   return0;

         }

module\_init(tiny4412\_hello\_module\_init);

说明：

（1）声明static，限定它的作用域，防止与其他文件下的同名函数起冲突。

（2）\_\_init 声明它是一个初始化的函数，在insmod或当编译到内核，在系统启动阶段会自动执行的函数。         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）。

（3）参数是void类型，没有形参。

（4）返回值是int类型，通常正常执行返回0，出现错误返回错误码。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。

（5）做为外部模块的时候，模块必须以module\_init(函数名的形式被指定)。

（6）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。

3.模块卸载函数（必须）

当rmmod时，或则通过其他方式卸载模块时候会执行。

static void \_\_exittiny4412\_hello\_module\_cleanup(void)

{

                 /\* 释放代码 \*/

          printk("Good-bye, Tiny4412 module wasremoved!\n");

}

module\_exit(tiny4412\_hello\_module\_cleanup);

说明：

（1）声明static，限定它的作用域，防止与其他文件下的同名函数起冲突。

（2）\_\_exit声明它是一个卸载类型函数，在执行rmmod命令的时候执行。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

对于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")))

（3）没有形参，没有返回值。

（4）做为外部模块的时候，必须以“module\_exit(函数名)”的形式来指定。

（5）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：

1.若模块加载函数注册了 XXX，则模块卸载函数应该注销 XXX。

2.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。

3.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。

4.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。

4.module\_init和module\_exit

（1）module\_init 声明tiny4412\_hello\_module\_init是一个初始化函数。

（2）module\_exit 声明tiny4412\_hello\_module\_cleanup是一个卸载函数。

        module\_init(tiny4412\_hello\_module\_init);

module\_exit(tiny4412\_hello\_module\_cleanup);

这两句话的作用跟\_\_init，\_\_exit作用相同，这里是为了双重保险。通常\_\_init，\_\_exit这个编译到内核的时候使用，而module\_init，module\_exit这两编译成模块使用。

5.MODULE\_LICENSE("GPL");

声明该模块是基于GPL协议创建的，如果没有声明，当安装的时候提示该程序会污染我的内核。在Linux 2.6内核中，可接受的LICENSE包括“GPL”、“GPL v2”、“GPL and additional rights”、“Dual BSD/GPL”、“DualMPL/GPL”和“Proprietary”。

大多数情况下，内核模块应遵循 GPL 兼容许可权。

6.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。

7.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。

8.模块作者等信息声明（可选）

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驱动所支持的设备列表，如下所示：

 /\* 对应此驱动的设备表 \*/

 staticstruct usb\_device\_id skel\_table [] = {

        { USB\_DEVICE(USB\_SKEL\_VENDOR\_ID,

         USB\_SKEL\_PRODUCT\_ID)},

         { }/\* 表结束 \*/

};

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

小知识：

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。

1.5模块的编译方式

模块的编译通常有两种，一种是修改内核配置菜单，利用内核配置菜单编译；另一种是自己编写makefike文件，读取内核的makefile来编译。

第一种：修改内核配置菜单

1）把模块添加到内核源码目录；

2）然后做一个菜单，再把菜单配置为M选项；

3）再使用 make modules命令编译成.ko文件。

这种方式比较麻烦，繁琐。

第二种：自己编写makefile

独立编写一个Makefile来编译模块；这样模块源码可以放在任何地方，这种方法在开发中最常用。其实可以不需要Makefile,只需要一条命令就能搞定的。这条命令如下：

make -C 内核源码绝对路径M=模块源码文件所在的绝对路径 modules

但是写成makefile是为了方便我们执行多步操作。

示例：编写一个编译模块的makefile文件通常包含以下内容。

指定编译目标：obj-m:=xxx.o

ARM板的内核源码路径：/works/linux-3.5/

模块源码路径 ：/linux\_share/hello\_model\_single

命令：

make -C /root/work/linux-3.5/ M=/linux\_share/hello\_model\_singlemodules

这条命令完成的工作是进入到内核源码目录，读取内核源码目录的Makefile。（-C dir 是读入指定目录下的makefile），执行内核源码Makefile中的modules目标（这里我们自己知道了目标obj-m:=xxx.o），根据 modules 目标编译M所指向的文件路径下的 C文件。

为了更方便，一般我们写成Makefile文件。

单个文件单模块Makefile的模板：

# Makefile 2.6

#hello最终的模块名，单文件单模块时，这个名字就是源码文件名，hello.o对应于hello.c

obj-m :=hello.o

#KDIR是内核源码路径，当编译用于X86平台模块的时候使用X86上的内核源码，当编译ARM的模块时候使用自己配套的Linux内核源码

#KDIR  :=/lib/modules/$(shell uname-r)/build

KDIR   := /works/linux-3.5/

all:

         make -C $(KDIR) M=$(PWD) modules  #$(PWD) 是代表当前路径，也就是模块源码路径

clean:

         rm -f \*.ko \*.o \*.mod.o \*.mod.c\*.symvers \*.markers \*.unsigned \*.order \*~

一般只需根据自己的情况修改KDIR 和 obj-m := ? 内容就可以了。

         注意：KDIR所指向的内核源码一定要被成功编译过，没有清除工程才能编译模块。

1.6模块几种常见模型示例

1.6.1 单文件单模块示例

1.模块代码清单

/\* hello.c \*/

#include<linux/module.h>       /\* Needed byall modules \*/

#include <linux/init.h>         /\* Needed for the module-macros \*/

static int \_\_init hello\_init(void)

{

         printk(KERN\_DEBUG"Hello world, priority = 7\n");

         printk(KERN\_INFO"Hello world, priority = 6\n");

         printk("Helloworld, priority = DEFAULT\_MESSAGE\_LOGLEVEL\n");

         printk(KERN\_NOTICE"Hello world, priority = 5\n");

         printk(KERN\_WARNING"Hello world, priority = 4\n");

         printk(KERN\_ERR"Hello world, priority = 3\n");

         printk(KERN\_CRIT"Hello world, priority = 2\n");

         printk(KERN\_ALERT"Hello world, priority = 1\n");

         printk(KERN\_EMERG"Hello world, priority = 0\n");

  return 0;

}

static void  \_\_exit hello\_exit(void)

{

         printk(KERN\_DEBUG"Goodbye,cruel world!, priority = 7\n");

         printk(KERN\_INFO"Goodbye,cruel world!, priority = 6\n");

         printk("Goodbye,cruelworld!, priority = DEFAULT\_MESSAGE\_LOGLEVEL\n");

         printk(KERN\_NOTICE"Goodbye,cruel world!, priority = 5\n");

         printk(KERN\_WARNING"Goodbye,cruel world!, priority = 4\n");

         printk(KERN\_ERR"Goodbye,cruel world!, priority = 3\n");

         printk(KERN\_CRIT"Goodbye,cruel world!, priority = 2\n");

         printk(KERN\_ALERT"Goodbye,cruel world!, priority = 1\n");

         printk(KERN\_EMERG"Goodbye,cruel world!, priority = 0\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("DualBSD/GPL");

MODULE\_AUTHOR("BENSON");

MODULE\_DESCRIPTION("STUDY\_MODULE");

2.makefile模板

# Makefile 2.6

obj-m := hello.o

KDIR  :=/lib/modules/$(shelluname -r)/build

#KDIR   :=/root/work/linux-3.5/

all:

         make -C $(KDIR)M=$(PWD) modules

         @rm -f \*.o \*.mod.o\*.mod.c \*.symvers \*.markers \*.unsigned \*.order \*~

clean:

         rm -f \*.ko \*.o \*.mod.o\*.mod.c \*.symvers \*.markers \*.unsigned \*.order \*~

3.编译，安装模块

如果编译给X86用的，因为printk的打印优先级问题，不会都打印到终端。

我们可以在终端输入tail/var/log/messages可以查看到日志文件最后10行，注意查看动作快点，要不然日志更新之后就看不到了。

如果编译给开发版使用，因为没有设置优先级，可以全部从串口打印输出。

1.6.2多模块之间有依赖关系示例

1.示例代码清单

首先我编写两个模块，它们之间是有依赖的。代码如下：

Calculate.c文件代码清单：

#include <linux/init.h>

#include <linux/module.h>

static int add\_integar(int a,intb)

{

         returna+b;

}

static int sub\_integar(int a,intb)

{

         returna-b;

}

static int \_\_init sym\_init()

{

         return0;

}

static void \_\_exit sym\_exit() {}

module\_init(sym\_init);

module\_exit(sym\_exit);

MODULE\_LICENSE("GPL");

        //提供接口

EXPORT\_SYMBOL(add\_integar);

EXPORT\_SYMBOL(sub\_integar);

Hello.c代码清单：

#include <linux/module.h>

#include <linux/init.h>

//在其他地方定义，在这里声明之后可以在本文件使用

extern int add\_integar(int a,int b);

extern int sub\_integar(int a,int b);

static int \_\_init hello\_init(void)

{

         intres=add\_integar(1,2);

         printk(KERN\_EMERG"helloinit , res=%d\n",res);

         return0;

}

static void \_\_exit hello\_exit()

{

         intres=sub\_integar(2,1);

         printk(KERN\_EMERG"helloexit,res=%d\n",res);

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("GPL");

2.makefile模板

依赖关系的Makefile可以写成以下形式：

# hello.o对应hello.c ,calculate.o对应calculate.c,编译后会生成两个ko文件。

obj-m := hello.ocalculate.o

#KDIR := /lib/modules/$(shell uname -r)/build

KDIR :=/root/work/linux-3.5/

all:

         @make -C $(KDIR) M=$(PWD) modules

         @rm -f \*.o \*.mod.o \*.mod.c \*.symvers\*.markers \*.unsigned \*.order \*~ \*.bak

clean:

         rm -f \*.ko \*.o \*.mod.o \*.mod.c\*.symvers \*.markers \*.unsigned \*.order \*~ \*.bak

注意:hello.c，calculate.c 必须是以模块代码框架格式编写，不能普通c代码格式。

3.代码安装卸载操作分析

操作示例：

[root@JUNJIA /home]# ls

calculate.ko            hello.ko                  //把以上的两个.C文件编译成模块

[root@JUNJIA /home]# lsmod

[root@JUNJIA /home]# insmod hello.ko      //先安装hello.c

[ 7103.030000] hello: Unknown symboladd\_integar (err 0)

[ 7103.030000] hello: Unknown symbolsub\_integar (err 0)

insmod: can't insert 'hello.ko': unknown symbolin module or invalid parameter

提示错误，找不到add\_integar，sub\_integar符号。原因这两个函数具体实现代码不存在 hello.ko中，也不在当前的内核中，而是在calculate.c中。所以安装不成功。

那么怎么办呢？先安装 calculate.ko，这样add\_integar，sub\_integar这两个函数就在内核中了。

[root@JUNJIA /home]# insmod calculate.ko

再来安装hello.ko，在内核中就可以找到add\_integar，sub\_integar这两个函数，所以安装成功。

[root@JUNJIA /home]# insmod hello.ko

[ 7154.985000] hello init , res=3

如果先卸载calculate.ko提示失败，因为hello.ko还在使用这个模块中的函数。

[root@JUNJIA /home]# rmmod calculate

rmmod: remove 'calculate': Resource temporarily unavailable

查看，没有卸载成功。

[root@JUNJIA /home]# lsmod

hello 746 0 - Live 0xbf014000 (O)

calculate 951 1 hello, Live 0xbf010000 (O)

应该先卸载 hello.ko。

[root@JUNJIA /home]# rmmod hello

[ 7214.570000] hello exit,res=1

rmmod: module 'hello' not found   //这一句busybox的BUG，并非是代码的错误

[root@JUNJIA /home]# lsmod

calculate 951 0 - Live 0xbf010000 (O)

再卸载calculate.ko，成功！

[root@JUNJIA /home]# rmmod calculate

rmmod: module 'calculate' not found

[root@JUNJIA /home]# lsmod

结论：

1.一个模块可以提供函数给另外一个模块调用。

2.安装时候，先安装被调用的模块，再安装主调模块

3.卸载时候和安装顺序相反。

4.模块可以依赖多个模块。

如果修改代码，把 calculate.c中的以下两行注释掉：

//EXPORT\_SYMBOL(add\_integar);

//EXPORT\_SYMBOL(sub\_integar);

重新编译模块，重新安装：

[root@JUNJIA /home]# insmod calculate.ko

[root@JUNJIA /home]# lsmod

calculate 595 0 - Live 0xbf018000 (O)

[root@JUNJIA /home]# insmod hello.ko

[ 7766.860000] hello: Unknown symbol add\_integar (err 0)

[ 7766.860000] hello: Unknown symbol sub\_integar (err 0)

insmod: can't insert 'hello.ko': unknown symbol in module or invalidparameter

[root@JUNJIA /home]#

结论：如果一个模块的函数要给其他模块调用，需要本模块使用 EXPORT\_SYMBOL导出。

1.6.3多文件单模块示例

就是多个C文件编译成一个.ko文件。

这种情况，只能有一个C文件是以代码模块编程格式编写，其他的C文件只能是普通C代码文件。

1.示例代码

Calculate.c代码清单：

int add\_integar(int a,int b)

{

         returna+b;

}

int sub\_integar(int a,int b)

{

         returna-b;

}

Hello.c代码清单：

#include <linux/module.h>

#include <linux/init.h>

extern int add\_integar(int a,intb);

extern int sub\_integar(int a,intb);

static int \_\_init hello\_init(void)

{

         intres=add\_integar(1,2);

         printk(KERN\_EMERG"helloinit , res=%d\n",res);

         return0;

}

static void \_\_exit hello\_exit()

{

         intres=sub\_integar(2,1);

         printk(KERN\_EMERG"helloexit,res=%d\n",res);

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("GPL");

Makefile 模板和单文件单模块有点不一样。

2.多文件单模块Makefiel模板示例

# mulc就是最终模块的名字，自己定义，并且不能和任何一个c文件同名

obj-m := mulc.o

#以下的格式：

#模块名-objs =模块的源文件对就的.o文件列表，

#calculate.o hello.o分别对应calculate.c hello.c

mulc-objs = calculate.o hello.o

#linux内核源码路径

#KDIR  :=/lib/modules/$(shell uname -r)/build

KDIR := /root/work/linux-3.5/

all:

         @make-C $(KDIR) M=$(PWD) modules

         @rm-f \*.o \*.mod.o \*.mod.c \*.symvers \*.markers \*.unsigned \*.order \*~ \*.bak

clean:

         rm-f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers \*.markers \*.unsigned \*.order \*~ \*.bak

其实这种Makefile 模板通用性比较，也适用单文件单模块的编译。

1.7模块传递参数

模块安装时候可以给模块中的变量传递数值。安装后，模块中变量的值就是安装时所传入的值；如果没有传递的变量，则使用代码中默认值。

1.7.1、模块传递参数知识点

带参数的模块安装后会生成/sys/module/模块名/parameters/这样的一个目录。

例如如果安装了名字为  hello\_model\_param 的模块；则会生成/sys/module/hello\_model\_param/parameters/。

这个文件夹下会生成以参数为名字文件，其内容就是参数的值。可以使用cat 或echo操作这个文件，修改变量的值。但是一般情况下都不建议修改，所以这个权限的值一般设置只读的。

模块中如果变量要通过安装时传参数覆盖默认值，则需要使用以下宏修饰：

module\_param，module\_param\_array

（1）对于非数组类型变量使用module\_param。

语法：module\_param(变量名，变量类型，变量的访问权限)

变量的访问权限：变量在文件系统以一个文件形式呈现，这个文件权限属性就是“变量的访问权限”。

变量传递参数：insmod xx.ko 变量名=值

（2）对于数组类型变量使用module\_param\_array。

语法：module\_param\_array(数组名，数组元素类型，&num，变量的访问权限)

         num 是一个 char ,int,long 或uchar,uint,ulong类型的变量，用于保存传递给数组的元素个数。

数组传递参数： insmod xx.ko 数组名=元素0,元素1,元素2,  ……

1.7.2所支持的数据类型

module\_param，module\_param\_array所支持数据类型是限的：

type支持的基本类型有:

         bool   :布尔类型

         invbool:颠倒了值的bool类型;

         charp  :字符指针类型,内存为用户提供的字符串分配;(char \*)

         int    :整型

         long   :长整型

         short  :短整型

         uint   :无符号整型（unsigned int）

         ulong  :无符号长整型（unsigned long）

         ushort :无符号短整型（unsignedshort）

      指针类只能char\*,对应名字 charp .

1.7.3权限类型

         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。

module\_param(age, int,S\_IRUGO|S\_IWUSR);

         module\_param(name,charp ,S\_IRUGO);

         module\_param\_array(aa,int, &num, S\_IRUGO);

1.7.4 模块传递示例

示例代码：

#include <linux/module.h>

#include <linux/init.h>

static int num = 0;

static char \*name ="DAVID";

static int age  = 30;

static int aa[10] ={1,2,3,4,5,6,7,8,9,0};

//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。

 //S\_IWUSR给用户写权限

module\_param(age, int,S\_IRUGO|S\_IWUSR);

module\_param(name, charp,S\_IRUGO);

module\_param\_array(aa, int,&num, S\_IRUGO);

static int \_\_init hello\_init(void)

{

         inti = 0;

         printk(KERN\_EMERG"Name:%s\n",name);

         printk(KERN\_EMERG"Age:%d\n",age);

         for(i=0;i<10;i++)

                   printk(KERN\_EMERG"%d  ",aa[i]);

         printk(KERN\_EMERG"num:%d  \n",num);

         return0;

}

static void \_\_exithello\_exit(void)

{

         printk(KERN\_EMERG"Moduleexit!\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("GPL");

这个例子是单个文件单个模块方式。编译它生成hello\_model\_param.ko。

在开发板安装如下：

aa数组传递了3 个元素，输出num值就是3。

[root@JUNJIA /home]# insmodhello\_model\_param.ko aa=1,2,3

         [11294.715000] Name:DAVID

         [11294.715000] Age:30

         [11294.715000] 1

         [11294.715000] 2  [11294.715000] 3

         4 [11294.715000] 5

         6 [11294.715000] 7

         8 [11294.715000] 9

         0 [11294.715000] num:3

         [root@JUNJIA /home]#