**内核编译与安装**

从www.kernel.org上下载源码包\*.tar.bz2

解压源码包

进入源码目录

执行make menconfig基于图形菜单配置内核，如果不想配置可使用默认命令make def config

执行make命令编译内核

执行make modules编译内核模块

执行make modules\_install安装内核模块

执行make install安装内核

执行reboot重启系统

**Hello World驱动模块**

示例代码：

#include <linux/init.h>

#include <linux/module.h>

static int test\_init(void){

    printk(KERN\_ALERT "Hello World\n");

    return 0;

}

static void  test\_exit(void){

    printk(KERN\_ALERT "Goodbye World\n");

}

// 宏，用于声明模块的加载函数

module\_init(test\_init);

// 宏，用于声明模块的释放函数

module\_exit(test\_exit);

// 代码规范声明，这些规范定义了模块在传播中的版权问题

MODULE\_LICENSE("GPL");

makefile文件：

# 当我们执行make时，当前makefile实际被载入了两次

# 第一次 KERNELRELEASE 未赋值，进入 ifeq

# 执行 all 命令，all 执行 make -C 转到内核目录执行内核的 makefile，并携带 -M 当前目录路径过去

# 内核执行内核的 makefile，并定义了KERNELRELEASE，并载入当前的 makefile

# 第二次进入 makefile，KERNELRELEASE 以赋值，进入 else

# obj-m 被定义

ifeq ($(KERNELRELEASE),)

# 当前路径

CURRENT\_PATH:=$(shell pwd)

# 当前内核版本

LINUX\_KERNEL:=$(shell uname -r)

# 当前内核路径（注：不是内核源码路径）

LINUX\_KERNEL\_PATH:=/lib/modules/$(LINUX\_KERNEL)/build

# 编译

all:

    make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

    rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

else

# 模块依赖，内核 makefile 会为 testmodule.o 自动添加 testmodule.c 依赖

obj-m += testmodule.o

endif

当执行make时会在当前目录下生成testmodule.ko内核模块

**模块操作**

安装内核模块：sudo insmod testmodule.ko

卸载内核模块：sudo rmmod testmodule

列出以加载的模块：lsmod

查看内核模块的输出信息：dmesg | tail -100

**模块参数**

#include <linux/init.h>

#include <linux/module.h>

// 模块参数变量

static char \* moduleparam1 = 0;

// 指定模块需要的参数，多参数则使用多行如下代码

// 参数名：moduleparam1

// 参数类型：charp（指针类型），也可以是其他类型 byte, int, long 等，但不支持浮动数

// 访问权限：S\_IRUGO（读），S\_IRUGP读权限，S\_IWUSR写权限，权限之间可以通过"|"进行连接

module\_param(moduleparam1, charp, S\_IRUGO);

static int test\_init(void){

    printk(KERN\_ALERT "Hello World\n");

    if(moduleparam1 != 0){

        // 输出模块参数

        printk(KERN\_ALERT "%s\n", moduleparam1);

    }

    return 0;

}

static void  test\_exit(void){

    printk(KERN\_ALERT "Goodbye World\n");

}

// 宏，用于声明模块的加载函数

module\_init(test\_init);

// 宏，用于声明模块的释放函数

module\_exit(test\_exit);

// 代码规范声明，这些规范定义了模块在传播中的版权问题

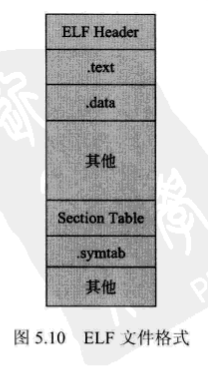
MODULE\_LICENSE("GPL");

在加载模块时指定参数（moduleparam1为我们的参数名）

sudo insmod testmodule.ko moduleparam1="abc"

**模块文件格式**

模块使用ELF文件格式

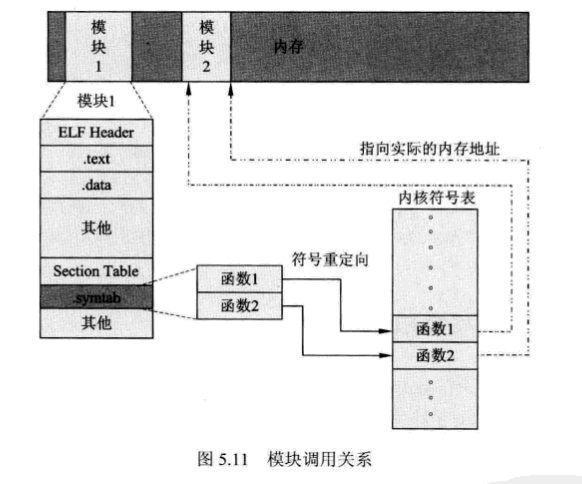


Text：代码段

Data：数据段，存放以初始化的数据

Symtab：符号表

**模块之间的通信**

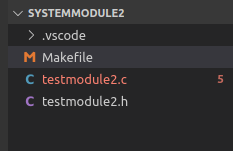


模块通信的过程：

1. 内核装载模块2，发现模块2的符号表的函数1、2可以导出，所以将函数1、2的地址存入内核符号表
2. 内核装载模块1，发现模块1的符号表函数1、2未解析，内核查找内核符号表函数1、2的地址，将地址填入模块1的符号表中

模块通信实例：

1. 新建项目SystemModule2，与原先项目SystemModule目录同在一个目录上
2. SystemModule2代码结构



1. Testmodule2.h代码，定义函数

#ifndef Testmodule\_H

#define Testmodule\_H

long add\_integer(long a, long b);

long sub\_integer(long a, long b);

#endif

1. Testmodule2.c代码

#include <linux/init.h>

#include <linux/module.h>

#include "./testmodule2.h"

// 定义add\_integer

long add\_integer(long a, long b){

    return a+b;

}

// 定义sub\_integer

long sub\_integer(long a, long b){

    return a-b;

}

// 导出add\_integer函数

EXPORT\_SYMBOL(add\_integer);

// 导出sub\_integer函数

EXPORT\_SYMBOL(sub\_integer);

static int test\_init(void){

    printk(KERN\_ALERT "Hello World\n");

    return 0;

}

static void  test\_exit(void){

    printk(KERN\_ALERT "Goodbye World\n");

}

// 宏，用于声明模块的加载函数

module\_init(test\_init);

// 宏，用于声明模块的释放函数

module\_exit(test\_exit);

// 代码规范声明，这些规范定义了模块在传播中的版权问题

MODULE\_LICENSE("GPL");

1. SystemModule2的filemake（与之前的filemake差不多，只是在all命令中不清理中间文件）

ifeq ($(KERNELRELEASE),)

# 当前路径

CURRENT\_PATH:=$(shell pwd)

# 当前内核版本

LINUX\_KERNEL:=$(shell uname -r)

# 当前内核路径（注：不是内核源码路径）

LINUX\_KERNEL\_PATH:=/lib/modules/$(LINUX\_KERNEL)/build

# 编译

all:

    make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

else

# 模块依赖，内核 makefile 会为 testmodule.o 自动添加 testmodule.c 依赖

obj-m += testmodule2.o

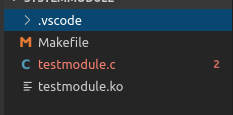
endif

1. 执行make时SystemModule2会产生一个符号表文件Module.symvers，其定义的模块的导出函数

0x00000000 sub\_integer /home/dabuside/Project/SystemModule2/testmodule2 EXPORT\_SYMBOL

0x00000000 add\_integer /home/dabuside/Project/SystemModule2/testmodule2 EXPORT\_SYMBOL

1. 转到SystemModule项目，代码结构如下



1. Testmudule.c

#include <linux/init.h>

#include <linux/module.h>

// 导入SystemModule2模块的头文件

#include "../SystemModule2/testmodule2.h"

// 模块参数变量

static long a = 0;

static long b = 0;

module\_param(a, long, S\_IRUGO);

module\_param(b, long, S\_IRUGO);

static int test\_init(void)

{

printk(KERN\_ALERT "Hello World\n");

    // 使用SystemModule2模块的函数

    printk(KERN\_ALERT "a+b=%ld\n", add\_integer(a, b));

printk(KERN\_ALERT "a-b=%ld\n", sub\_integer(a, b));

    return 0;

}

static void  test\_exit(void){

    printk(KERN\_ALERT "Goodbye World\n");

}

// 宏，用于声明模块的加载函数

module\_init(test\_init);

// 宏，用于声明模块的释放函数

module\_exit(test\_exit);

// 代码规范声明，这些规范定义了模块在传播中的版权问题

MODULE\_LICENSE("GPL");

1. SystemModule的makefile文件，这里使用的SystemModule2的符号表KBUILD\_EXTRA\_SYMBOLS

ifeq ($(KERNELRELEASE),)

# 当前路径

CURRENT\_PATH:=$(shell pwd)

# 当前内核版本

LINUX\_KERNEL:=$(shell uname -r)

# 当前内核路径（注：不是内核源码路径）

LINUX\_KERNEL\_PATH:=/lib/modules/$(LINUX\_KERNEL)/build

# 编译

all:

    make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

    rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

else

# 模块依赖，内核 makefile 会为 testmodule.o 自动添加 testmodule.c 依赖

obj-m += testmodule.o

# 模块依赖的符号表

KBUILD\_EXTRA\_SYMBOLS=$(obj)/../SystemModule2/Module.symvers

endif

1. 执行各个模块的make
2. 使用 insmod 命令先安装testmodule2模块，在安装testmodule模块
3. 使用dmesg命令查看模块输出

**将模块加入内核**

如果希望模块随系统一起启动，需要将模块静态编译进内核

将模块静态编译进内核需要配置Kconfig，这里就不做介绍了