Linux 驱动开发庖丁解牛之二

——模块编程

dreamice
e-mail:dreamice.jiang@gmail.com

本文是建立在前面的开发环境已经成功建立的基础之上的。如果没有建立好，请参照《Linux 驱动开发庖丁解牛之一——开发环境的建立》。

已经有很多文档讲述模块编程，个人觉得《The Linux kernel module programming guide》是最详尽的。本文不再立足于从理论上去阐述模块编程的相关知识，而着重从实践的基础上去掌握模块编程，领悟模块编程的实质。当然，具备足够的理论知识才能从实践出发，所以，本文档尽量配合《Linux Device Driver》第三版的第二章，以及讲述模块编程最完善的文档《The Linux kernel module programming guide》。下面，我们从实践开始出发吧。

1. 人之初（hello world）

```
/*
 *  hello.c - The first kernel module programming
 */
#include <linux/module.h>      /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_ALERT */
#include <linux/init.h>

MODULE_LICENCE(“Dual BSD/GPL”);

static int hello_init(void)
{
        printk(KERN_INFO "Hello world\n");

        /*
         * A non 0 return means init_module failed; module can't be loaded.
         */
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_INFO "Goodbye world\n");
}

module_init(hello_init);
```

```
module_exit(hello_exit);

MODULE_AUTHOR("dreamice, jyjiang2005@gmail.com");
MODULE_DESCRIPTION("The first module program");
MODULE_VERSION("V1.0");
MODULE_ALIAS("Chinese: ren zhi chu");
```

Makefile:

```
obj-m := hello.o
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
	$(RM) *.o *.ko *.mod.c Module.symvers
```

现在，我们一步一步来解析这个最简单的 hello world 模块程序。

1．/linux/module.h 这个是必须的。这个头文件包含了对模块结构的定义以及相关信息。

2．module_init 和 module_exit 这两个函数是必须的。module_init 就好比应用程序的 main 函数，没有 main 函数，应用程序将不知道从哪里开始执行。

3．关于 printk，在 ldd3 的第四章有详细的说明，这个可以说是内核调试的一个基本手段。

4．MODULE_LICENCE，MODULE_AUTHOR，MODULE_DESCRIPTION，MODULE_VERSION，MODULE_ALIA，分别是模块许可证，模块作者、描述、版本以及别名的描述，除了许可证这个比较正式以外（遵循 GPL），其它几个主要是用作开发者的一些控制和描述信息，使用比较灵活。

5．Makefile，很特别，不同于一般的应用程序的 Makefile。首先，模块编译的目标必须以 obj-m 这样的形式指出；其次，模块的编译必须指定内核源代码的路径——这也是模块运行在内核空间的一个原因。模块有多个源文件生成的情况可如下编写：

```
obj-m := module.o
module-objs := file1.o file2.o
```

6．module_exit 为模块推出执行清理的函数。如果模块加载后不允许卸载，那么这个函数就不用实现了。

2. 模块常见错误

处理错误常常是程序员比较头痛的事情，这里我把比较常见的一些模块编译错误罗列一下，可能不是很全面。

1． Invalid module format

这个通常是版本不一致问题导致的。比如说，你在 2.4 内核上编译的模块，如果运行到 2.6 版的内核，有可能就会出现这样的提示。

另外可以尝试一下，把一个普通文件改名为 hello.ko，执行 insmod hello.ko，也会报

这个错误。

2． Unresolved symbol……

这个错误常常是你引用的某个函数可能出了问题。如引用内核模块并没有导出的函数。在这里，顺便把模块符号导出描述一下：

EXPROT_SYMBOL(name)

EXPROT_SYMBOL_GPL(name)

如果一个模块希望另一个模块引用自己的函数，那么必须使用以上两个函数导出符号，否则，其它模块是不能引用的。就会报这个错误了。

3． 这个错误不是编译问题的错误：

模块运行于内核空间，所以，不能引用标准库函数，也不能处理浮点数。如果不加注意，可能导致一些无法预知的错误。

4． 由于内核版本升级，导致一些结构体的改变，如果在低版本的内核上编写模块，到高版本编译，就可能出现“no such member……”类似的错误。

Linux 驱动程序的开发者无非处于两种情况：

1． 对一个全新的硬件编写驱动程序。这种情况需要对该版本内核的相关部分有充分了解。程序员的精力既要考虑模块功能的设计实现，又要考虑内核的接口功能等等。

2． 移植一个驱动程序。往往从低版本到高版本或者高版本到低版本的移植。这种情况下，需要对两个版本的内核有一个充分的认识，就是可能有些结构体，或者函数接口变化的问题。程序员往往不需要花太多的精力在模块功能的实现上，而更多的精力在版本差异上。

3. 模块层叠技术

关于模块层叠技术很多书上介绍的比较简略。模块层叠技术，主要指模块的依赖关系，如模块 A 的实现依赖于 B，如果 B 并没有加载，那么，当 insmod A 的时候，将无法成功。这个时候，就必须使用 modprobe A，把相关的模块统统加载到内核，即 B 也得到了加载。我们尽量不要使用 modprobe –r 来移出一个模块，因为这将导致相关联的模块都被移出。

4．模块实现的内核代码分析

1．数据结构

模块相关的数据结构存放在 include/linux/module.h

```
struct module
{
    enum module_state state;

    /* Member of list of modules */
    struct list_head list;
```

```
/* Unique handle for this module */
char name[MODULE_NAME_LEN];

/* Sysfs stuff. */
struct module_kobject mkobj;
struct module_param_attrs *param_attrs;
struct module_attribute *modinfo_attrs;
const char *version;
const char *srcversion;

/* Exported symbols */
const struct kernel_symbol *syms;
unsigned int num_syms;
const unsigned long *crcs;

/* GPL-only exported symbols. */
const struct kernel_symbol *gpl_syms;
unsigned int num_gpl_syms;
const unsigned long *gpl_crcs;

/* unused exported symbols. */
const struct kernel_symbol *unused_syms;
unsigned int num_unused_syms;
const unsigned long *unused_crcs;
/* GPL-only, unused exported symbols. */
const struct kernel_symbol *unused_gpl_syms;
unsigned int num_unused_gpl_syms;
const unsigned long *unused_gpl_crcs;

/* symbols that will be GPL-only in the near future. */
const struct kernel_symbol *gpl_future_syms;
unsigned int num_gpl_future_syms;
const unsigned long *gpl_future_crcs;

/* Exception table */
unsigned int num_exentries;
const struct exception_table_entry *extable;

/* Startup function. */
int (*init)(void);

/* If this is non-NULL, vfree after init() returns */
void *module_init;
```

```
        /* Here is the actual code + data, vfree'd on unload. */
        void *module_core;

        /* Here are the sizes of the init and core sections */
        unsigned long init_size, core_size;

        /* The size of the executable code in each section.    */
        unsigned long init_text_size, core_text_size;

        /* The handle returned from unwind_add_table. */
        void *unwind_info;

        /* Arch-specific module values */
        struct mod_arch_specific arch;

        /* Am I unsafe to unload? */
        int unsafe;

        /* Am I GPL-compatible */
        int license_gplok;

#ifdef CONFIG_MODULE_UNLOAD
        /* Reference counts */
        struct module_ref ref[NR_CPUS];

        /* What modules depend on me? */
        struct list_head modules_which_use_me;

        /* Who is waiting for us to be unloaded */
        struct task_struct *waiter;

        /* Destruction function. */
        void (*exit)(void);
#endif

#ifdef CONFIG_KALLSYMS
        /* We keep the symbol and string tables for kallsyms. */
        Elf_Sym *symtab;
        unsigned long num_symtab;
        char *strtab;

        /* Section attributes */
        struct module_sect_attrs *sect_attrs;
```

```
#endif

        /* Per-cpu data. */
        void *percpu;

        /* The command line arguments (may be mangled).    People like
           keeping pointers to this stuff */
        char *args;
};
```

在内核中，每一个内核模块都由这样一个 module 对象来描述。所有的 module 对象由一个链表链接在一起，其中每个对象的 next 域都指向链表的下一个元素。

State 表示 module 当前的状态,主要包括一下几种状态：

MODULE_STATE_LIVE

MODULE_STATE_COMING

MODULE_STATE_GOING

其中，加载后的模块的状态为 MODULE_STATE_LIVE。

name 保存 module 的名字；

param_attrs 指向 module 可传递的参数的名称及属性；

init 和 exit 这两个函数指针，可以看作是 hello world 对应的 init 和 exit 函数，即模块初始化和模块退出所调用的函数；

struct list_head modules_which_use_me 这个成员是一个链表，指示了所有依赖于该模块的模块。

下面我们继续看看模块的加载和退出函数。

操作系统在初始化时，调用 static LIST_HEAD(modules)建立了一个空链表，之后，每装入一个内核模块，即创建一个 struct module 结构，并把它链入 modules 这个全局的链表中。

从操作系统内核的角度来说，它提供的用户服务，都是通过系统调用来实现的。实际上，我们在调用 module_init 和 module_exit 时，都是首先需要通过这两个系统调用来实现的：sys_init_module(), sys_delete_module()。我使用的是比较新的 2.6.25 的内核源码。在内核源码：arch/x86/kernel/ syscall_table_32.S,我们看到这两个系统调用的函数：

```
        ……
        .long sys_sigprocmask
        .long sys_ni_syscall        /* old "create_module" */
        .long sys_init_module
        .long sys_delete_module
        .long sys_ni_syscall        /* 130:    old "get_kernel_syms" *
        …….
```

在 kernel/module.c 中，我们来看这两个函数的具体实现：

```
        /* This is where the real work happens */
/* umod 指向用户空间中该内核模块 image 所在的位置。Image 以 elf 的可执行文件格式保存，image 的最前部是 elf_ehdr 类型结构，长度由 len 指示。Uargs 指向来自用户空间的参数。*/
asmlinkage long
sys_init_module(void __user *umod,
        unsigned long len,
```

```
        const char __user *uargs)
{
    struct module *mod;
    int ret = 0;

    /* Must have permission */
    /*验证是否有权限装入内核模块*/
    if (!capable(CAP_SYS_MODULE))
        return -EPERM;

    /* Only one module load at a time, please */
    /*一个时候，只能有一个内核模块在加载中…这可能也是为了保持内核模块链表的独占式访问*/
    if (mutex_lock_interruptible(&module_mutex) != 0)
        return -EINTR;

    /* Do all the hard work */
    /*真正的内核模块加载的实现函数，包括物理内存的分配，image 的检查等等，这个函数实现比较庞大复杂，这里不作细致分析了…这里之后，模块的 state 也变成了 MODULE_STATE_COMING */
    mod = load_module(umod, len, uargs);
    if (IS_ERR(mod)) {
        mutex_unlock(&module_mutex);
        return PTR_ERR(mod);
    }

    /* Drop lock so they can recurse */
    mutex_unlock(&module_mutex);

    blocking_notifier_call_chain(&module_notify_list,
            MODULE_STATE_COMING, mod);

    /* Start the module */
    if (mod->init != NULL)
        ret = mod->init(); /*执行模块的初始化工作…*/
    if (ret < 0) {/*初始化失败*/
        /* Init routine failed: abort.    Try to protect us from
                buggy refcounters. */
        /*模块加载不成功，设置成 MODULE_STATE_GOING ，因为稍后将作清除处理*/
        mod->state = MODULE_STATE_GOING;
        synchronize_sched();
        module_put(mod);
        mutex_lock(&module_mutex);
        free_module(mod);
```

```
        mutex_unlock(&module_mutex);
        wake_up(&module_wq);
        return ret;
    }
    if (ret > 0) {
        printk(KERN_WARNING "%s: '%s'->init suspiciously returned %d, "
                            "it should follow 0/-E convention\n"
                    KERN_WARNING "%s: loading module anyway...\n",
                    __func__, mod->name, ret,
                    __func__);
        dump_stack();
    }

    /* Now it's a first class citizen!    Wake up anyone waiting for it. */
    mod->state = MODULE_STATE_LIVE; /*模块加载成功后，状态变成_LIVE 了*/
    wake_up(&module_wq);

    mutex_lock(&module_mutex);
    /* Drop initial reference. */
    module_put(mod);
    unwind_remove_table(mod->unwind_info, 1);
    module_free(mod, mod->module_init);
    mod->module_init = NULL;
    mod->init_size = 0;
    mod->init_text_size = 0;
    mutex_unlock(&module_mutex);

    return 0;
}

/* name_user  是要删除的模块名称，前面的__user 说明这是一个用户空间的名称*/
asmlinkage long
sys_delete_module(const char __user *name_user, unsigned int flags)
{
    struct module *mod;
    char name[MODULE_NAME_LEN];
    int ret, forced = 0;

    /*权限检查*/
    if (!capable(CAP_SYS_MODULE))
        return -EPERM;

    /*把模块名称从用户空间传递到内核空间*/
    if (strncpy_from_user(name, name_user, MODULE_NAME_LEN-1) < 0)
```

```
        return -EFAULT;
name[MODULE_NAME_LEN-1] = '\0';

if (mutex_lock_interruptible(&module_mutex) != 0)
        return -EINTR;

/*查找要删除的 module*/
mod = find_module(name);
if (!mod) {
        ret = -ENOENT;
        goto out;
}

/*如果还有其它模块正在使用该模块，则不能把该模块删除*/
if (!list_empty(&mod->modules_which_use_me)) {
        /* Other modules depend on us: get rid of them first. */
        ret = -EWOULDBLOCK;
        goto out;
}

/* Doing init or already dying? */
if (mod->state != MODULE_STATE_LIVE) {
        /* FIXME: if (force), slam module count and wake up
                        waiter --RR */
        DEBUGP("%s already dying\n", mod->name);
        ret = -EBUSY;
        goto out;
}

/* If it has an init func, it must have an exit func to unload */
if (mod->init && !mod->exit) {
        forced = try_force_unload(flags);
        if (!forced) {
                /* This module can't be removed */
                ret = -EBUSY;
                goto out;
        }
}

/* Set this up before setting mod->state */
mod->waiter = current;

/* Stop the machine so refcounts can't move and disable module. */
ret = try_stop_module(mod, flags, &forced);
```

```
    if (ret != 0)
        goto out;

    /* Never wait if forced. */
    if (!forced && module_refcount(mod) != 0)
        wait_for_zero_refcount(mod);

    /* Final destruction now noone is using it. */
    if (mod->exit != NULL) {
        mutex_unlock(&module_mutex);
        mod->exit(); /*  这里相当于执行模块的 exit 函数，执行后续的清理工作……*/
        mutex_lock(&module_mutex);
    }
    /* Store the name of the last unloaded module for diagnostic purposes */
    strlcpy(last_unloaded_module, mod->name, sizeof(last_unloaded_module));
    free_module(mod);

 out:
    mutex_unlock(&module_mutex);
    return ret;
}
```

以上就是模块实现的内核源码分析，虽然分析不够详尽，但对于编写内核模块来说，理清这么一条线，将有助于更加深入的理解模块的执行及在内核中的情况。

5. 后记

虽然费了很大力气写完了这篇模块编程，但总感觉欠完善，这也跟自己对内核以及驱动程序的经验和整体把握能力有关。我希望它虽然存在很多缺点，但能起到一个引线的作用，作一个总结和分析，同时是对自己阅读学习的总结，也希望对它人有所帮助，作为去深入细化研究的一个参考吧。

敬请批评指正！