第十章　模块机制

如前所述，Linux 是一个整体式的内核（Monolithic Kernel）结构，也就是说，整个内核是一个单独的、非常大的程序，从实现机制来说，我们又把它划分为 5 个子系统（参见第一章内核结构），内核的各个子系统都提供了内部接口（函数和变量），这些函数和变量可供内核所有子系统调用和使用。

另一种是微内核结构，内核的功能块被划分成独立的模块，这些功能块之间有严格的通信机制，要给内核增加一个新成分，配置过程是相当费时的，例如，如果 NCR 810 SCSI 需要一个 SCSI 驱动程序，你不能把它直接加入内核，在用 NCR 810 之前，你就得进行配置并且建立新的内核。

Linux 的整体式结构决定了要给内核增加新的成分也是非常困难，因此 Linux 提供了一种全新的机制——可装入模块（Loadable Modules，以下简称模块），用户可以根据自己的需要，在不需要对内核进行重新编译的条件下，模块能被动态地插入到内核或从内核中移走。

10.1　概述

10.1.1　什么是模块

模块是内核的一部分（通常是设备驱动程序），但是并没有被编译到内核里面去。它们被分别编译并连接成一组目标文件，这些文件能被插入到正在运行的内核，或者从正在运行的内核中移走，进行这些操作可以使用 insmod（插入模块）或 rmmod（移走模块）命令，或者，在必要的时候，内核本身能请求内核守护进程(kerned)装入或卸下模块。这里列出在 Linux 内核源程序中所包括的一些模块。

- 文件系统：minix，xiafs，msdos，umsdos，sysv，isofs，hpfs,smbfs，ext3,nfs，proc 等。
- 大多数 SCSI 驱动程序：(如：aha1542，in2000)。
- 所有的 SCSI 高级驱动程序：disk，tape，cdrom，generic。
- 大多数以太网驱动程序：(非常多，不便于在这儿列出，请查看./Documentation/networking/net-modules.txt)。
- 大多数 CD-ROM 驱动程序：

```
aztcd:      Aztech,Orchid,Okano,Wearnes
cm206:      Philips/LMS CM206
```

```
gscd:       Goldstar GCDR-420
mcd, mcdx:  Mitsumi LU005, FX001
optcd:      Optics Storage Dolphin 8000AT
sjcd:       Sanyo CDR-H94A
sbpcd:      Matsushita/Panasonic CR52x, CR56x, CD200,
            Longshine LCS-7260, TEAC CD-55A
sonycd535:  Sony CDU-531/535, CDU-510/515
```

- 以及很多其他模块，诸如：

```
lp: 行式打印机
binfmt_elf: elf 装入程序
binfmt_java: java 装入程序
isp16: cd-rom 接口
serial: 串口（tty）
```

这里要说明的是，Linux 内核中的各种文件系统及设备驱动程序，既可以被编译成可安装模块，也可以被静态地编译进内核的映像中，这取决于内核编译之前的系统配置阶段用户的选择。通常，在系统的配置阶段，系统会给出 3 种选择（Y/M/N），“Y”表示要某种设备或功能，并将相应的代码静态地连接在内核映像中；“M”表示将代码编译成可安装模块，“N”表示不安装这种设备。

10.1.2 为什么要使用模块

按需动态装入模块是非常吸引人的，因为这样可以保证内核达到最小并且使得内核非常灵活，例如，当你可能偶尔使用 VFAT 文件系统，你只要安装(mount) VFAT，VFAT 文件系统就成为一个可装入模块，kerneld 通过自动装入 VFAT 文件系统建立你的 Linux 内核，当你卸下(unmount)VFAT 部分时，系统检测到你不再需要的 FAT 系统模块，该模块自动地从内核中被移走。按需动态装入模块还意味着，你会有更多的内存给用户程序。如前所述，内核所使用的内存是永远不会被换出的，因此，如果你有 100KB 不使用的驱动程序被编译进内核，那就意味着你在浪费 RAM。任何事情都是要付出代价的，内核模块的这种优势是以性能和内存的轻微损失为代价的。

一旦一个 Linux 内核模块被装入，那么它就像任何标准的内核代码一样成为内核的一部分，它和任何内核代码一样具有相同的权限和职责。像所有的内核代码或驱动程序一样，Linux 内核模块也能使内核崩溃。

10.1.3 Linux 内核模块的优缺点

利用内核模块的动态装载性具有如下优点：

- 将内核映像的尺寸保持在最小，并具有最大的灵活性；
- 便于检验新的内核代码，而不需重新编译内核并重新引导。

但是，内核模块的引入也带来了如下问题：

- 对系统性能和内存利用有负面影响；
- 装入的内核模块和其他内核部分一样，具有相同的访问权限，因此，差的内核模块会导致系统崩溃；
- 为了使内核模块访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改这些符号表；
- 有些模块要求利用其他模块的功能，因此，内核要维护模块之间的依赖性；
- 内核必须能够在卸载模块时通知模块，并且要释放分配给模块的内存和中断等资源；
- 内核版本和模块版本的不兼容，也可能导致系统崩溃，因此，严格的版本检查是必需的。

尽管内核模块的引入同时也带来不少问题，但是模块机制确实是扩充内核功能一种行之有效的方法，也是在内核级进行编程的有效途径。

10.2　实现机制

Linux 内核模块机制的实现与内核其他部分的关系并不是很大，相对来说也不是很复杂，但其设计思想是非常值得借鉴的。我们并不准备对其实现函数做一一介绍，在此仅介绍主要的数据结构及实现函数。

10.2.1　数据结构

1. 模块符号

如前所述，Linux 内核是一个整体结构，而模块是插入到内核中的插件。尽管内核不是一个可安装模块，但为了方便起见，Linux 把内核也看作一个模块。那么模块与模块之间如何进行交互呢，一种常用的方法就是共享变量和函数。但并不是模块中的每个变量和函数都能被共享，内核只把各个模块中主要的变量和函数放在一个特定的区段，这些变量和函数就统称为符号。到底哪些符号可以被共享？Linux 内核有自己的规定。对于内核模块，在 kernel/ksyms.c 中定义了从中可以“移出”的符号，例如进程管理子系统可以“移出”的符号定义如下：

```
/* process memory management */
EXPORT_SYMBOL (do_mmap_pgoff) ;
EXPORT_SYMBOL (do_munmap) ;
EXPORT_SYMBOL (do_brk) ;
EXPORT_SYMBOL (exit_mm) ;
EXPORT_SYMBOL (exit_files) ;
EXPORT_SYMBOL (exit_fs) ;
EXPORT_SYMBOL (exit_sighand) ;

    EXPORT_SYMBOL (complete_and_exit) ;
```

```
EXPORT_SYMBOL (__wake_up) ;
EXPORT_SYMBOL (__wake_up_sync) ;
EXPORT_SYMBOL (wake_up_process) ;
EXPORT_SYMBOL (sleep_on) ;
EXPORT_SYMBOL (sleep_on_timeout) ;
EXPORT_SYMBOL (interruptible_sleep_on) ;
EXPORT_SYMBOL (interruptible_sleep_on_timeout) ;
EXPORT_SYMBOL (schedule) ;
EXPORT_SYMBOL (schedule_timeout) ;
EXPORT_SYMBOL (jiffies) ;
EXPORT_SYMBOL (xtime) ;
EXPORT_SYMBOL (do_gettimeofday) ;
EXPORT_SYMBOL (do_settimeofday) ;
```

你可能对这些变量和函数已经很熟悉。其中宏定义 EXPORT_SYMBOL () 本身的含义是“ 移出符号 ”。为什么说是 “ 移出 ” 呢？因为这些符号本来是内核内部的符号，通过这个宏放在一个公开的地方，使得装入到内核中的其他模块可以引用它们。

实际上，仅仅知道这些符号的名字是不够的，还得知道它们在内核映像中的地址才有意义。因此，内核中定义了如下结构来描述模块的符号：

```
struct module_symbol
{
        unsigned long value; ／*符号在内核映像中的地址*/
        const char *name;   /*指向符号名的指针*/
};
```

从后面对 EXPORT_SYMBOL 宏的定义可以看出，连接程序（ld）在连接内核映像时将这个结构存放在一个叫做“ __ksymtab ”的区段中，而这个区段中所有的符号就组成了模块对外“ 移出 ” 的符号表，这些符号可供内核及已安装的模块来引用。而其他 “ 对内 ” 的符号则由连接程序自行生成，并仅供内部使用。

与 EXPORT_SYMBOL 相关的定义在 include/linux/module.h 中：

```
   #define __MODULE_STRING_1 (x)     #x
  #define __MODULE_STRING (x)       __MODULE_STRING_1 (x)

#define __EXPORT_SYMBOL (sym, str)                          \
const char __kstrtab_##sym[]                              \
__attribute__ ( (section (".kstrtab") ) )  = str;                \
const struct module_symbol __ksymtab_##sym                \
__attribute__ ( (section ("__ksymtab") ) )  =                    \
 {  (unsigned long) &sym, __kstrtab_##sym }

#if defined(MODVERSIONS) || !defined(CONFIG_MODVERSIONS)
#define EXPORT_SYMBOL(var)  __EXPORT_SYMBOL(var, __MODULE_STRING(var))
```

下面我们以 EXPORT_SYMBOL(schedule)为例，来看一下这个宏的结果是什么。

首先 EXPORT_SYMBOL(schedule)的定义成了__EXPORT_SYMBOL(schedule,“ sched- ule ”)。而__EXPORT_SYMBOL () 定义了两个语句，第 1 个语句定义了一个名为__kstrtab_ schedule 的字符串，将字符串的内容初始化为“ schedule ”，并将其置于内核映像中的.kstrtab 区段，注意这是一个专门存放符号名字符串的区段。第 2 个语句则定义了一个名为__kstrtab_ schedule 的 module_symbol 结构，将其初始化为 {&schedule，__kstrtab_ schedule} 结

构，并将其置于内核映像中的__ksymtab 区段。这样，module_symbol 结构中的域 value 的值就为 schedule 在内核映像中的地址，而指针 name 则指向字符串“ schedule ”。

2．模块引用(Module Reference)

模块引用是一个不太好理解的概念。 有些装入内核的模块必须依赖其他模块，例如，因为 VFAT 文件系统是 FAT 文件系统或多或少的扩充集，那么，VFAT 文件系统依赖（depend）于 FAT 文件系统，或者说，FAT 模块被 VFAT 模块引用，或换句话说，VFAT 为“ 父 ”模块，FAT 为“ 子 ”模块。其结构如下：

```
struct module_ref
{
         struct module *dep;          /* “ 父 ”模块指针*/
         struct module *ref;          /* “ 子 ”模块指针*/
         struct module_ref *next_ref;/*指向下一个子模块的指针*/
};
```

在这里“ dep ”指的是依赖，也就是引用，而“ ref ”指的是被引用。因为模块引用的关系可能延续下去，例如 A 引用 B，B 有引用 C，因此，模块的引用形成一个链表。

3．模块

模块的结构为 module，其定义如下：

```
   struct module_persist; /* 待决定 */

struct module
 {
        unsigned long size_of_struct;   /* 模块结构的大小，即 sizeof（module） */
        struct module *next;  /*指向下一个模块 */
        const char *name;   /*模块名，最长为 64 个字符*/
        unsigned long size;  /*以页为单位的模块大小*/

        union
           {
                 atomic_t usecount; /*使用计数，对其增减是原子操作*/
                 long pad;
         } uc;                      /* Needs to keep its size - so says rth */

        unsigned long flags;            /* 模块的标志 */

        unsigned nsyms;         /* 模块中符号的个数 */
        unsigned ndeps;        /* 模块依赖的个数 */
        struct module_symbol *syms; /* 指向模块的符号表,表的大小为 nsyms */

            struct module_ref deps; /*指向模块引用的数组，大小为 ndeps */
        struct module_ref *refs;
        int  (*init)  (void) ;       /* 指向模块的 init_module () 函数 */
        void  (*cleanup)  (void) ;  /* 指向模块的 cleanup_module () 函数 */
        const struct exception_table_entry *ex_table_start;
        const struct exception_table_entry *ex_table_end;
/*以下域是在以上基本域的基础上的一种扩展，因此是可选的。可以调用 mod_member_
```

present()函数来检查以下域的存在与否。*/

```
        const struct module_persist *persist_start;/*尚未定义*/
        const struct module_persist *persist_end;
        int  (*can_unload)  (void) ;
        int runsize                                  /*尚未使用*/
        const char *kallsyms_start;                  /*用于内核调试的所有符号 */
        const char *kallsyms_end;
        const char *archdata_start;                  /* 与体系结构相关的特定数据*/
        const char *archdata_end;
        const char *kernel_data;                     /*保留 */
};
```

其中，moudle 中的状态，即 flags 的取值定义如下：

```
  /* Bits of module.flags.  */

#define MOD_UNINITIALIZED      0 ／*模块还未初始化*/
#define MOD_RUNNING             1 ／*模块正在运行*/
#define MOD_DELETED             2 ／*卸载模块的过程已经启动*/
#define MOD_AUTOCLEAN          4 ／*安装模块时带有此标记，表示允许自动
卸载模块*/
#define MOD_VISITED             8 ／*模块被访问过*/
#define MOD_USED_ONCE           16 ／*模块已经使用过一次*/
#define MOD_JUST_FREED          32 ／*模块刚刚被释放*/
#define MOD_INITIALIZING        64 ／*正在进行模块的初始化*/－／
```

如前所述，虽然内核不是可安装模块，但它也有符号表，实际上这些符号表受到其他模块的频繁引用，将内核看作可安装模块大大简化了模块设计。因此，内核也有一个 module 结构，叫做 kernel_module，与 kernel_module 相关的定义在 kernel／module_c 中：

```
#if defined (CONFIG_MODULES)  || defined (CONFIG_KALLSYMS)

extern struct module_symbol __start___ksymtab[];
extern struct module_symbol __stop___ksymtab[];

extern const struct exception_table_entry __start___ex_table[];
extern const struct exception_table_entry __stop___ex_table[];

extern const char __start___kallsyms[] __attribute__  ( (weak) ) ;
extern const char __stop___kallsyms[] __attribute__  ( (weak) ) ;

struct module kernel_module =
{
        size_of_struct:         sizeof (struct module) ,
        name:                   "",
        uc:                     {ATOMIC_INIT (1) },
        flags:                  MOD_RUNNING,
        syms:                     __start___ksymtab,
        ex_table_start:         __start___ex_table,
        ex_table_end:           __stop___ex_table,
        kallsyms_start:         __start___kallsyms,
        kallsyms_end:           __stop___kallsyms,
};
```

首先要说明的是，内核对可安装模块的的支持是可选的。如果在编译内核代码之前的系

统配置阶段选择了可安装模块，就定义了编译提示 CONFIG_MODULES，使支持可安装模块的代码受到编译。同理，对用于内核调试的符号的支持也是可选的。

凡是在以上初始值未出现的域，其值均为 0 或 NULL。显然，内核没有 init_module()和 cleanup_module()函数，因为内核不是一个真正的可安装模块。同时，内核没有 deps 数组，开始时也没有 refs 链。可是，这个结构的指针 syms 指向__start___ksymtab，这就是内核符号表的起始地址。符号表的大小 nsyms 为 0，但是在系统能初始化时会在 init_module()函数中将其设置成正确的值。

在模块映像中也可以包含对异常的处理。发生于一些特殊地址上的异常，可以通过一种描述结构 exception_table_entry 规定对异常的反映和处理，这些结构在可执行映像连接时都被集中在一个数组中，内核的 exception_table_entry 结构数组就为__start___ex_table[]。当异常发生时，内核的异常响应处理程序就会先搜索这个数组，看看是否对所发生的异常规定了特殊的处理，相关内容请看第四章。

另外，从 kernel_module 开始，所有已安装模块的 module 结构都链在一起成为一条链，内核中的全局变量 module_list 就指向这条链：

```
struct module *module_list = &kernel_module;
```

10.2.2　实现机制的分析

当你新建立了最小内核（如何建立新内核，请看相关的 HOWTO），并且重新启动后，你可以利用实用程序“ insmod ”和“ rmmod ”，随意地给内核插入或从内核中移走模块。如果 kerneld 守护进程启动，则由 kerneld 自动完成模块的插拔。有关模块实现的源代码在/kernel/module.c 中，以下是对源代码中主要函数的分析。

1．启动时内核模块的初始化函数 init_modules()

当内核启动时，要进行很多初始化工作，其中，对模块的初始化是在 main.c 中调用 init_modules()函数完成的。实际上，当内核启动时唯一的模块就为内核本身，因此，初始化要做的唯一工作就是求出内核符号表中符号的个数：

```
*／
  * Called at boot time
  */

void __init init_modules (void)
{
         kernel_module.nsyms = __stop___ksymtab - __start___ksymtab;

        arch_init_modules (&kernel_module) ;
}
```

因为内核代码被编译以后，连接程序进行连接时内核符号的符号结构就“ 移出 ”到了 ksymtab 区段，__start___ksymtab 为第 1 个内核符号结构的地址，__stop___ksymtab 为最后一个内核符号结构的地址，因此二者之差为内核符号的个数。其中，arch_init_modules 是与体系结构相关的函数，对 i386 来说，arch_init_modules 在 include/i386/module.h 中

定义为：

```
#define arch_init_modules (x)     do { } while  (0)
```

可见，对 i386 来说，这个函数为空。

2．创建一个新模块

当用 insmod 给内核中插入一个模块时，意味着系统要创建一个新的模块，即为一个新的模块分配空间，函数 sys_create_module()完成此功能，该函数也是系统调用 screate_module () 在内核的实现函数，其代码如下：

```
/*
  * Allocate space for a module.
  */

asmlinkage unsigned long
sys_create_module (const char *name_user, size_t size)
{
         char *name;
         long namelen, error;
         struct module *mod;
         unsigned long flags;

         if  (!capable (CAP_SYS_MODULE) )
                 return -EPERM;
         lock_kernel () ;
         if  ( (namelen = get_mod_name (name_user, &name) )  < 0)  {
                error = namelen;
                 goto err0;
         }
        if  (size < sizeof (struct module) +namelen)  {
                 error = -EINVAL;
                goto err1;
         }
        if  (find_module (name)  != NULL)  {
                error = -EEXIST;
                goto err1;
         }
        if  ( (mod =  (struct module *) module_map (size) )  == NULL)  {
                error = -ENOMEM;
                 goto err1;
         }

         memset (mod, 0, sizeof (*mod) ) ;
         mod->size_of_struct = sizeof (*mod) ;
         mod->name =  (char *)  (mod + 1) ;
         mod->size = size;
        memcpy ( (char*)  (mod+1) , name, namelen+1) ;

         put_mod_name (name) ;

         spin_lock_irqsave (&modlist_lock, flags) ;
```

```
            mod->next = module_list;
            module_list = mod;      /* link it in */
           spin_unlock_irqrestore (&modlist_lock, flags) ;

            error =  (long)  mod;
            goto err0;
    err1:
            put_mod_name (name) ;
    err0:
            unlock_kernel () ;
            return error;
    }
```

下面对该函数中的主要语句给予解释。

- capable(CAP_SYS_MODULE)检查当前进程是否有创建模块的特权。
- 参数 size 表示模块的大小，它等于 module 结构的大小加上模块名的大小，再加上模块映像的大小，显然，size 不能小于后两项之和。
- get_mod_name（）函数获得模块名的长度。
- find_module（）函数检查是否存在同名的模块，因为模块名是模块的唯一标识。
- 调用 module_map（）分配空间，对 i386 来说，就是调用 vmalloc()函数从内核空间的非连续区分配空间。
- memset（）将分配给 module 结构的空间全部填充为 0，也就是说，把通过 module_map（）所分配空间的开头部分给了 module 结构；然后（module+1）表示从 mod 所指的地址加上一个 module 结构的大小，在此处放上模块的名字；最后，剩余的空间给模块映像。
- 新建 moudle 结构只填充了三个值，其余值有待于从用户空间传递过来。
- put_mod_name()释放局部变量 name 所占的空间。
- 将新创建的模块结构链入 module_list 链表的首部。

3．初始化一个模块

从上面可以看出，sys_create_module（）函数仅仅在内核为模块开辟了一块空间，但是模块的代码根本没有拷贝过来。实际上，模块的真正安装工作及其他的一些初始化工作由 sys_init_module()函数完成，该函数就是系统调用 init_module()在内核的实现代码，因为其代码比较长，为此对该函数的主要实现过程给予描述。

该函数的原型为：

```
asmlinkage long sys_init_module (const char *name_user, struct module *mod_user)
```

其中参数 name_user 为用户空间的模块名，mod_user 为指向用户空间欲安装模块的 module 结构。

该函数的主要操作描述如下。

- sys_create_module（）在内核空间创建了目标模块的 module 结构，但是这个结构还基本为空，其内容只能来自用户空间。因此，初始化函数就要把用户空间的 module 结构拷贝到内核中对应的 module 结构中。但是，由于内核版本在不断变化，因此用户空间 module 结构可能与内核中的 module 结构不完全一样，例如用户空间的 module 结构为 2.2 版，而内核空间的为 2.4 版，则二者的 module 结构就有所不同。为了防止二者的 module 结构在大小上

的不一致而造成麻烦，因此，首先要把用户空间的 module 结构中的 size_of_struct 域复制过来加以检查。从前面介绍的 module 结构可以看出，2.4 版中从 persist_start 开始的域是内核对 module 结构的扩充，用户空间的 module 结构中没有这些域，因此二者 module 结构大小的检查不包括扩充域。

- 通过了对结构大小的检查以后，先把内核中的 module 结构保存在堆栈中作为后备，然后就从用户空间拷贝其 module 结构。复制时是以内核中 module 结构的大小为准的，以免破坏内核中的内存空间。
- 复制过来以后，还要检查 module 结构中各个域的合理性。
- 最后，还要对模块名进行进一步的检查。虽然已经根据参数 name_user 从用户空间拷贝过来了模块名,但是这个模块名是否与用户空间 module 结构中所指示的模块名一致呢？显然，也可能存在不一致的可能，因此还要根据 module 结构的内容把模块映像中的模块名也复制过来，再与原来使用的模块名进行比较。
- 经过以上检查以后，可以从用户空间把模块的映像复制过来了。
- 但是把模块映像复制过来并不是万事大吉，模块之间的依赖关系还得进行修正，因为正在安装的模块可能要引用其他模块中的符号。虽然在用户空间已经完成了对这些符号的连接，但现在必须验证所依赖的模块在内核中还未被卸载。如果所依赖的模块已经不在内核中了，则对目标模块的安装就失败了。在这种情况下，应用程序（例如 insmod）有责任通过系统调用 delete_module()将已经创建的 module 结构从 moudle_list 中删除。
- 至此，模块的安装已经基本完成，但还有一件事要做，那就是启动待执行模块的 init_moudle（）函数，每个模块块必须有一个这样的函数，module 结构中的函数指针 init 就指向这个函数，内核可以通过这个函数访问模块中的变量和函数，或者说，init_moudle（）是模块的入口，就好像每个可执行程序的入口都是 main()一样。

4．卸载模块的函数 sys_delete_module()

卸载模块的系统调用为 delete_module()，其内核的实现函数为 sys_delete_module()，该函数的原型为：

```
asmlinkage long  sys_delete_module (const char *name_user)
```

与前面几个系统调用一样，只有特权用户才允许卸载模块。卸载模块的方式有两种，这取决于参数 name_user，name_user 是用户空间中的模块名。如果 name_user 非空，表示卸载一个指定的模块；如果为空，则卸载所有可以卸载的模块。

（1）卸载指定的模块

一个模块能否卸载，首先要看内核中是否还有其他模块依赖该模块，也就是该模块中的符号是否被引用，更具体地说，就是检查该模块的 refs 指针是否为空。此外，还要判断该模块是否在使用中，即__MOD_IN_USE（）宏的值是否为 0。只有未被依赖且未被使用的模块才可以卸载。

卸载模块时主要要调用目标模块的 cleanup_module()函数，该函数撤销模块在内核中的注册，使系统不再能引用该模块。

一个模块的拆除有可能使它所依赖的模块获得自由，也就是说，它所依赖的模块其 refs 队列变为空，一个 refs 队列为空的模块就是一个自由模块，它不再被任何模块所依赖。

(2) 卸载所有可以卸载的模块

如果参数 name_user 为空，则卸载同时满足以下条件的所有模块。

- 不再被任何模块所依赖。
- 允许自动卸载，即安装时带有 MOD_AUTOCLEAN 标志位。
- 已经安装但尚未被卸载，即处于运行状态。
- 尚未被开始卸载。
- 安装以后被引用过。
- 已不再使用。

以上介绍了 init_module ()、create_module ()、delete_module()三个系统调用在内核的实现机制，还有一个查询模块名的系统调用 query_module()。这几个系统调用是在实现 insmod 及 rmmod 实用程序的过程中被调用的，用户开发的程序一般不应该、也不必要直接调用这些系统调用。

5．装入内核模块 request_module()函数

在用户通过 insmod 安装模块的过程中，内核是被动地接受用户发出的安装请求。但是，在很多情况下，内核需要主动地启动某个模块的安装。例如，当内核从网络中接收到一个特殊的 packet 或报文时，而支持相应规程的模块尚未安装；又如，当内核检测到某种硬件时，而支持这种硬件的模块尚未安装等等，类似情况还有很。在这种情况下，内核就调用 request_module()主动地启动模块的安装。

request_module()函数在 kernel/kmod.c 中：

```
  /**
  * request_module - try to load a kernel module
     * @module_name: Name of module
*
     * Load a module using the user mode module loader. The function returns
     * zero on success or a negative errno code on failure. Note that a
     * successful module load does not mean the module did not then unload
     * and exit on an error of its own. Callers must check that the service
     * they requested is now available not blindly invoke it.
     *
     * If module auto-loading support is disabled then this function
     * becomes a no-operation.
     */
int request_module (const char * module_name)
{
           pid_t pid;
         int waitpid_result;
         sigset_t tmpsig;
        int i;
        static atomic_t kmod_concurrent = ATOMIC_INIT (0) ;
 #define MAX_KMOD_CONCURRENT 50  /* Completely arbitrary value - KAO */
         static int kmod_loop_msg;

         /* Don't allow request_module ()  before the root fs is mounted!  */
         if  ( ! current->fs->root )  {
```

```
                printk(KERN_ERR "request_module[%s]: Root fs not mounted\n", module_name);
            return -EPERM;
          }

        /* If modprobe needs a service that is in a module, we get a recursive
        * loop.  Limit the number of running kmod threads to max_threads/2 or
             * MAX_KMOD_CONCURRENT, whichever is the smaller.  A cleaner method
               * would be to run the parents of this process, counting how many times
        * kmod was invoked.  That would mean accessing the internals of the
             * process tables to get the command line, proc_pid_cmdline is static
              * and it is not worth changing the proc code just to handle this case.
              * KAO.
              */
              i = max_threads/2;
            if  (i > MAX_KMOD_CONCURRENT)
                    i = MAX_KMOD_CONCURRENT;
            atomic_inc (&kmod_concurrent) ;
            if  (atomic_read (&kmod_concurrent)  > i)  {
                    if  (kmod_loop_msg++ < 5)
                            printk (KERN_ERR
                        "kmod: runaway modprobe loop assumed and stopped\n") ;
                    atomic_dec (&kmod_concurrent) ;
                    return -ENOMEM;
            }

            pid = kernel_thread (exec_modprobe,  (void*)  module_name, 0) ;
            if  (pid < 0)  {
                   printk (KERN_ERR "request_module[%s]: fork failed, errno %d\n", module_name,
-pid) ;
                    atomic_dec (&kmod_concurrent) ;
                   return pid;
            }

            /* Block everything but SIGKILL/SIGSTOP */
            spin_lock_irq (&current->sigmask_lock) ;
            tmpsig = current->blocked;
            siginitsetinv (&current->blocked, sigmask (SIGKILL)  | sigmask (SIGS- TOP) ) ;
            recalc_sigpending (current) ;
            spin_unlock_irq (&current->sigmask_lock) ;

            waitpid_result = waitpid (pid, NULL, __WCLONE) ;
            atomic_dec (&kmod_concurrent) ;

            /* Allow signals again.. */
            spin_lock_irq (&current->sigmask_lock) ;
            current->blocked = tmpsig;
           recalc_sigpending (current) ;
            spin_unlock_irq (&current->sigmask_lock) ;

           if  (waitpid_result != pid)  {
     printk (KERN_ERR "request_module[%s]: waitpid (%d,...)  failed, errno %d\n",
                         module_name, pid, -waitpid_result) ;
```

```
        }
        return 0;
    }
```

对该函数的解释如下。

- 因为 request_module（）是在当前进程的上下文中执行的，因此首先检查当前进程所在的根文件系统是否已经安装。
- 对 request_module（）的调用有可能嵌套，因为在安装过程中可能会发现必须先安装另一个模块。例如，MS-DOS 模块需要另一个名为 fat 的模块，fat 模块包含基于文件分配表（FAT）的所有文件系统所通用的一些代码。因此，如果 fat 模块还不在系统中，那么在系统安装 MS-DOS 模块时，fat 模块也必须被动态链接到正在运行的内核中。因此，就要对嵌套深度加以限制，程序中设置了一个静态变量 kmod_concurrent，作为嵌套深度的计数器，并且还规定了嵌套深度的上限为 MAX_KMOD_CONCURRENT。不过，对嵌套深度的控制还要考虑到系统中对进程数量的限制，即 max_therads，因为在安装的过程中要创建临时的进程。
- 通过了这些检查以后，就调用 kernel_thread() 创建一个内核线程 exec_modprobe（）。exec_modprobe（）接受要安装的模块名作为参数，调用 execve() 系统调用执行外部程序/sbin/modprobe，然后， modprobe 程序真正地安装要安装的模块以及所依赖的任何模块。
- 创建内核线程成功以后，先把当前进程信号中除 SIGKILL 和 SIGSTOP 以外的所有信号都屏蔽掉，免得当前进程在等待模块安装的过程中受到干扰，然后就通过 waitpid（）使当前进程睡眠等待，直到 exec_modprobe（）内核线程完成模块安装后退出。当前进程被唤醒而从 waitpid（）返回时，又要恢复当前进程原有信号的设置。根据 waitpid（）的返回值可以判断 exec_modprobe () 操作的成功与否。如果失败，就通过 prink()在系统的运行日志“／var/log/message ”中记录一条出错信息。

10.3　模块的装入和卸载

10.3.1　实现机制

有两种装入模块的方法，第 1 种是用 insmod 命令人工把模块插入到内核，第 2 种是一种更灵活的方法，当需要时装入模块，这就是所谓的请求装入。

当内核发现需要一个模块时，例如，用户安装一个不在内核的文件系统时，内核将请求内核守护进程（kerneld）装入一个合适的模块。

内核守护进程（kerneld）是一个标准的用户进程，但它具有超级用户权限。kerneld 通常是在系统启动时就开始执行，它打开 IPC（Inter-Process Communication）到内核的通道，内核通过给 kerneld 发送消息请求执行各种任务。

kerneld 的主要功能是装入和卸载内核模块，但它也具有承担其他任务的能力，例如，当必要时，通过串行链路启动 PPP 链路，不需要时，则关闭它。kerneld 并不执行这些任务，

它通过运行诸如 insmod 这样的程序来做这些工作，kerneld 仅仅是内核的一个代理。

insmod 实用程序必须找到请求装入的内核模块，请求装入的内核模块通常保存在/lib/modules/kernel-version/目录下。内核模块被连接成目标文件，与系统中其他程序不同的是，这种目标文件是可重定位的（它们是 a.out 或 ELF 格式的目标文件）。insmods 实用程序位于／sbin 目录下，该程序执行以下操作。

(1) 从命令行中读取要装入的模块名。

(2) 确定模块代码所在的文件在系统目录树中的位置，即/lib/modules／kernel-version/目录。

(3) 计算存放模块代码、模块名和 module 结构所需要的内存区大小。调用 create_module( )系统调用，向它传递新模块的模块名和大小。

(4) 用 QM_MODULES 子命令反复调用 query_module( )系统调用来获得所有已安装模块的模块名。

(5) 用 QM_SYMBOL 子命令反复调用 query_module( )系统调用来获得内核符号表和所有已经安装到内核的模块的符号表。

(6) 使用内核符号表、模块符号表以及 create_module( )系统调用所返回的地址重新定位该模块文件中所包含的文件的代码。这就意味着用相应的逻辑地址偏移量来替换所有出现的外部符号和全局符号。

(7) 在用户态地址空间中分配一个内存区，并把 module 结构、模块名以及为正在运行的内核所重定位的模块代码的一个拷贝装载到这个内存区中。如果该模块定义了 init_module( )函数，那么 module 结构的 init 域就被设置成该模块的 init_module( )函数重新分配的地址。同理，如果模块定义了 cleanup_module( )函数，那么 cleanup 域就被设置成模块的 cleanup_module( )函数所重新分配的地址。

(8) 调用 init_module( )系统调用，向它传递上一步中所创建的用户态的内存区地址。

(9) 释放用户态内存区并结束。

为了取消模块的安装，用户需要调用/sbin/rmmod 实用程序，它执行以下操作：

(1) 从命令行中读取要卸载的模块的模块名。

(2) 使用 QM_MODULES 子命令调用 query_module( )系统调用来取得已经安装的模块的链表。

(3) 使用 QM_REFS 子命令多次调用 query_module( )系统调用来检索已安装的模块间的依赖关系。如果一个要卸载的模块上面还安装有某一模块，就结束。

(4) 调用 delete_module( )系统调用，向其传递模块名。

10.3.2 如何插入和卸载模块

如前所述，插入和卸载模块的实用程序为 insmod 和 rmmod，在此，我们将介绍在使用这些命令的过程中会遇到的问题，而并不详细介绍其用法，其更详细的使用请用 man 命令进行查看。

只有超级用户才能插入一个模块，其简单的命令如下：

```
insmod serial.o
```

其中，serial.o 为串口的驱动程序。

但是，这条命令执行以后可能会出现错误信息，诸如模块与内核版本不匹配、不认识的符号等。

例如，如果想插入 msdos.o，就可能出现如下信息：

```
msdos.o: unresolved symbol fat_date_UNIX2dos
msdos.o: unresolved symbol fat_add_cluster1
msdos.o: unresolved symbol fat_put_super
...
```

这是因为 msdos.o 引用的这些符号不是由内核“移出”的。为了证实这点，你可以查看/proc/ksyms，从中就可以发现内核移出的所有符号，但找不到“fat_date_unix2dos”符号。那么，怎样才能让这个符号出现在符号列表中呢？从这个符号可以看出，msdos.o 所依赖的模块为 fat.o，于是重新使用 insmod 命令把 fat.o 插入到内核，然后再查看/proc/modules，就会发现有两个模块被装入，并且一个模块依赖于另一个模块：

```
msdos                   5632   0  (unused)
fat                     30400   0 [msdos]
```

也许你要问，怎么才能知道所依赖的模块呢？除了从符号名判断外，更有效的方法是使用 depmod 和 modprobe 命令来代替 insmod 命令。

当错误信息为“kernel/module version mismatch”时，说明内核和模块的版本不匹配，这部分内容我们将在后面给予讨论。

通常情况下，当你插入模块时，还需要把参数传递给模块。例如，一个设备驱动程序想知道它所驱动的设备的 I/O 地址和 IRQ，或者一个网络驱动程序想知道你要它进行多少次的诊断跟踪。这里给出一个例子：

```
insmod ne.o  io=0x400  irq=10
```

这里装入的是 NE2000 类的以太网适配器驱动程序，并告诉它以太网适配器的 I/O 地址为 0x400，其所产生的中断为 IRQ 10。

对于可装入模块，并没有标准的参数形式，也几乎没有什么约定。每个模块的编写者可以决定 insmod 可以用什么样的参数。对于 Linux 内核现已支持的模块，Linux HOWTO 文档给出了每种驱动程序的参数信息。

另外，一个常见的错误是试图插入一个不是可装入模块的目标文件。例如，在内核配置阶段，你把 USB 核心模块静态地连接进基本内核中，因此，USB 核心模块就不是可装入模块。该模块的文件名是 usbcore.o，这看起来与可装入模块的文件名 usbcore.o 完全一样，但是你不能用 insmod 命令插入那个文件，否则，出现以下错误信息：

```
usbcore.o: couldn't find the kernel version this module was compiled for
```

这条消息告诉你，insmod 把 usbcore.o 当作一个合法的可装入模块来看待，并查找这个模块曾经被编译的内核版本，但没有找到。但我们知道，真正的原因是这个文件根本就不是一个可安装模块。

从内核卸载一个模块的命令为 rmmod，例如卸载 ne 模块的命令为：

```
rmmod  ne
```

10.4 内核版本

模块的编写与内核版本密切相关，在此，我们既要讨论内核版本与模块版本的兼容性问题，也要讨论内核从 2.0 到 2.2 及从 2.2 到 2.4 的内核 API 变化对编写模块的影响。

10.4.1 内核版本与模块版本的兼容性

可装入模块的编写者必须意识到，可装入模块既独立于内核又依赖于内核，所谓“独立”是因为它可以独立编译，所谓依赖是指它要调用内核或其他已装入模块的函数或变量，因此，内核版本的变化直接影响着曾经编写的模块是否能被新的内核认可。

例如，mydriver.o 是基于 Linux 2.2.1 内核编写和编译的，但是有人想把它装入到 Linux 2.2.2 的内核中，如果 mydriver.o 所调用的内核函数在 2.2.2 中有所变化，那么内核怎么知道内核版本与模块所调用函数的版本不一致呢？

为了解决这个问题，可装入模块的开发者就决定给模块也编以内核的版本号。在上面的例子中，mydriver.o 目标文件的.modinfo 特殊区段就含有“2.2.1”，因为 mydriver.o 的编译使用了来自 Linux 2.2.1 的头文件，因此，当把该驱动程序装入到 2.2.2 内核时，insmod 就会发现不匹配而失败，从而告诉你内核版本不匹配。

但是，Linux 2.2.1 和 2.2.2 之间的一点不兼容是否真的影响 mydriver.o 的执行呢？mydriver.o 仅仅调用了内核中几个函数、访问了几个数据结构，可以肯定地说，这些函数和数据结构并不一定随每个版本的稍微变化而变化。这种版本上的严格限制在一定程度上带来了不便，因为每当有版本变动时，就要重新编译（或从网上下载）许多可安装模块，而这种变动也许并不影响模块的运行，因此应当有其他的办法来解决这个问题。

办法之一，insmod 有一个-f 选项来强迫 insmod 忽略内核版本的不匹配，并把模块插入到任何版本中，但是，你仍然会得到一个版本不匹配的警告信息。

办法之二，就是将版本信息编码进符号名中，例如将版本号作为符号名的后缀。这样如果符号名相同而版本号不一致，insmod 就会认为是不同的符号而不予连接。但是，内核把是否将版本信息编码进符号名中是作为一个可选项 CONFIG_MODVERSIONS 来提供的。如果需要版本信息，就可以在编译内核代码前的系统配置阶段选择这个可选项；如果不需要，就可以在编译模块的源代码时加上-D CONFIG_MODVERSIONS 取消这个选项。

当以符号编码来编译内核或模块时，我们前面介绍的 EXPORT_SYMBOL（）宏定义的形式就有所不同，例如模块最常调用的内核函数 register_chrdev（），其函数名的宏定义的在 C 中为：

```
#define register_chrdev register_chrdev_Rc8dc8350
```

把符号 register_chrdev 定义为 register_chrdev 加上一个后缀，这个后缀就是 register_chrdev（）函数实际源代码的校验和，只要函数的源代码改动一个字符，这个校验和也会发生变化。因此，尽管你在源代码中读到的函数名为 register_chrdev，但 C 的预处理程序知道真正调用的是 register_chrdev_Rc8dc8350。

10.4.2　从版本 2.0 到 2.2 内核 API 的变化

与 Linux 2.0 相比，2.2 在性能、资源的利用、健壮性以及可扩展性等方面已经有了很大的改善，这些改善势必导致内核 API 的变化。所谓内核 API 就是指为进行内核扩展而提供的编程接口，内核的编程指编写驱动程序、文件系统及其他的内核代码。有关驱动程序的内容请参见下一章。

1．用户空间与内核空间之间数据的拷贝

早在 Linux 2.1.x 版本时，Liuns 就提出了一种有效的办法来改善内核空间与用户空间之间数据的拷贝。我们知道，内核空间与用户空间之间数据的拷贝要通过一个缓冲区，在以前的内核中，对这个缓冲区有效性的检查是通过 verify_area()函数的，如果这个缓冲区有效，则调用 memcpy_tofs()把数据从内核空间拷贝到用户空间。但是，verify_area()函数是低效的，因为它必须检查每一个页面，看其是否是一个有效的映射。

在 2.1.x（以及后来的版本）中，取消了对用户空间缓冲区每个页面的检查，取而代之的是用异常来处理非法的缓冲区。这就避免了在 SMP 上的竞争条件及有效性检查。verify_area()函数现在仅仅用来检查缓冲区的范围是否合法，这是一个快速的操作。

因此，如果你要把数据拷贝到用户空间，就使用 copy_to_user()函数，其用法如下：

```
if （ copy_to_user （ubuff, kbuff, length） ） return -EFAULT;
```

这里，ubuff 是用户空间的缓冲区，kbuff 是内核空间的缓冲区，而 length 是要拷贝的字节数。如果 copy_to_user()函数返回一个非 0 值，就意味着某些数据没有被拷贝（由于无效的缓冲区）。在这种情况下，返回-EFAULT 以表示缓冲区是无效的。类似地，从用户空间拷贝到内核空间的用法如下：

```
if （ copy_from_user （kbuff, ubuff, length） ） return -EFAULT;
```

注意，这两个函数都自动调用 verify_area()函数，你没必要自己调用它。

2．文件操作的方法

在内核 2.1.42 版本以后，增加了一个目录高速缓存（dcache）层，这个层加速了目录搜索操作（大约能提高 4 倍），但同时也需要改变文件操作接口。对驱动程序的编写者，这个变化相对比较简单：原来传递给 file_operations 某些方法的参数为 struct inode *，现在改为 struct dentry *。如果你的驱动程序要引用 inode，下面代码就足够了：

```
struct inode *inode = dentry->d_inode;
```

假定 dentry 是目录项的变量名。实际上，有些驱动程序就不涉及 inode，因此可忽略这一步。然而，你必须改变的是，重新声明 file_operations 中的函数。注意，某些方法还是把 inode 而不是 dentry 作为参数来传递。

有些方法甚至没有提供 dentry，仅仅提供了 struct file *，在这种情况下，你可以用下面的代码提取出 dentry：

```
struct dentry *dentry = file->f_dentry;
```

假定 file 是指向 file 指针的变量名。

下面是内核 2.2.x 文件操作的方法：

```
loff_t llseek （struct file *, loff_t, int）;
```

```
    ssize_t read (struct file *, char *, size_t, loff_t *) ;
    ssize_t write (struct file *, const char *, size_t, loff_t *) ;
    int readdir (struct file *, void *, filldir_t) ;
    unsigned int poll (struct file *, struct poll_table_struct *) ;
    int ioctl (struct inode *, struct file *, unsigned int, unsigned long) ;
    int mmap (struct file *, struct vm_area_struct *) ;
    int open (struct inode *, struct file *) ;
    int flush (struct file *) ;
    int release (struct inode *, struct file *) ;
    int fsync (struct file *, struct dentry *) ;
    int fasync (int, struct file *, int) ;
    int check_media_change (kdev_t dev) ;
    int revalidate (kdev_t dev) ;
    int lock (struct file *, int, struct file_lock *) ;
```

在你声明自己的 file_operations 结构时，应当确保把自己的方法放置在与上面一致的位置。不过，还有另外一种我们提到过的方法，其形式如下：

```
static struct file_operations mydev_fops = {
    open:    mydev_open,
    release: mydev_close,
    read:    mydev_read,
    write:   mydev_write,
};
```

gcc 编译程序能够把这些方法放在正确的位置，并把未定义的方法置为 NULL。

另外还值得注意的是，Linux 2.2 中引入了 pread() 和 pwrite()系统调用，这就允许进程可以从一个文件的指定位置进行读和写，这与另一个 lseek()系统调用类似但不完全相同。其不同之处是，pread() 和 pwrite()系统调用能对一个文件进行并发访问。为了对这些新的系统调用进行支持，在 read()和 write()方法中增加了第 4 个（或最后一个）参数，这个参数是指向 offset 的一个指针。作为驱动程序的编写者，你不必关心文件的位置，因此可以忽略这个参数。不过，为了正确性起见，你最好在你的驱动程序中避免使用新的系统调用，就像你不必支持 llseek()方法一样，你也可以在 read()和 write()方法的最顶行增加如下行来避免新旧系统调用的差异：

```
    if (ppos != &file->f_pos) return -ESPIPE
```

假定 ppos 是指向 offset 指针的变量名，file 是指向 struct file 结构的变量名。对于 read() 和 write()系统调用，传递给参数 offset 的为 file->f_pos 的地址，而对于 pread() 和 pwrite()系统调用，传递给 offset 的为 ppos 的地址，由此可以很容易地区别这种两种情况。

如果你确实关心文件的位置，那就必须使用和更新由 ppos 所指向的值，以跟踪进程正读在何处。

3．信号的处理

新增加的 signal_pending()函数时的信号的处理更加容易和健壮。2.0 版处理方式是：

```
    if (current->signal & ~current->blocked)
```

2.2 版是：

```
    if ( signal_pending (current) )
```

4．I/O 空间映射

任何系统都会有输入输出，因此就会涉及对外部设备的访问。在早期的计算机中，外设通常只有几个寄存器，通过这几个寄存器就可以完成对外设的所有操作。而现在的情况则大不一样，外设通常自带几 MB 的存储器，从自 PCI 总线出现以后，更是如此。所以，必须将外设卡上的存储器映射到内存空间，实际上是虚存空间（3GB 以上）。在以前的 Linux 内核版本中，这样的映射是通过 vremap（）建立的，现在改名为 ioremap()，以更确切地反映其真正的意图。

5．I/O 事件的多路技术

select() 和 poll()系统调用可以让一个进程同时处理多个文件描述符，也就是说可以使进程检测同时等待的多个 I/O 设备，当没有设备准备好时，select()阻塞，其中任一设备准备好时,select()就返回。在 Linux 2.0 中,驱动程序通过在 file_operations 结构中提供 select() 方法来支持这种技术，而在 Linux 2.2 中，驱动程序必须提供的是 poll()方法，这种方法具有更大的灵活性。

6．丢弃初始化函数和数据

当内核初始化全部完成以后，就可以丢弃以后不再需要的函数和数据，这意味着存放这些函数和数据的内存可以重新得到使用。但这仅仅应用在编译进内核的驱动程序，而不适合于可安装模块。

定义一个以后要丢弃的变量的形式为：

```
static int mydata __initdata = 0;
```

定义一个以后要丢弃的函数的形式为：

```
__initfunc (void myfunc (void) )
{
 }
```

__initdata 和 __initfunc 关键字把代码和数据放在一个特殊的“初始化”区段。较理想的做法应当是，尽可能地把更多的代码和数据放在初始化区段，当然，这里的代码和数据指的是初始化以后（当 init 进程启动时）不再使用的。

7．定时的设定

新增加了一些定时设定函数。Linux 2.0 设定定时是这样的：

```
current->timeout = jiffies + timeout;
schedule () ;
```

Linux 2.2 是：

```
timeout = schedule_timeout (timeout) ;
```

同理，如果你需要在一个等待队列上睡眠，但需要定时，Linux 2.0 操作是：

```
current->timeout = jiffies + timeout;
interruptible_sleep_on (&wait) ;
```

Linux 2.2 是：

```
timeout = interruptible_sleep_on_timeout (&wait, timeout) ;
```

注意，这些新函数返回的是剩余时间的多少。在某些情况下，这些函数在定时时间还没

到就返回。

8．向后兼容的宏

你可以把下面的代码包含进自己编写的代码中，这样就不必费神维护是为 Linux 2.2.x 还是为 Linux 2.0.x 所编译的驱动程序。

```
#include <linux/version.h>
#ifndef KERNEL_VERSION
#  define KERNEL_VERSION (a,b,c)   ( ( (a)  << 16)  +  ( (b)  << 8)  +  (c)
#endif
#if  (LINUX_VERSION_CODE < KERNEL_VERSION (2,1,0) )
#  include <linux/mm.h>
static inline unsigned long copy_to_user  (void *to, const void *from,
                     unsigned long n)
{
    if  ( !verify_area  (VERIFY_WRITE, to, n)  )  return n;
    memcpy_tofs  (to, from, n) ;
    return 0;
 }
 static inline unsigned long copy_from_user  (void *to, const void *from,
                       unsigned long n)
{
     if  ( !verify_area  (VERIFY_READ, from, n)  )  return n;
     memcpy_fromfs  (to, from, n) ;
     return 0;
 }
#  define __initdata
#  define __initfunc (func)  func
#else
#  include <asm/uaccess.h>
#endif
#ifndef signal_pending
#  define signal_pending (p)   (  (p) ->signal & ~ (p) ->blocked )
#endif
```

10.4.3　把内核 2.2 移植到内核 2.4

如果你曾对 Linux 2.0 版比较熟悉，现在要在内核 2.4 版下开发驱动程序，那么在了解了 2.0 到 2.2 内核 API 的变化后，还要了解 2.2 到 2.4 的变化。

1．使用设备文件系统（DevFS）

DevFS 设备文件系统是 Linux 2.4 一个全新的功能，它主要为了有效地管理/dev 目录而开发的。我能知道，UNIX/Linux 中所有的目录都是层次结构，唯独/dev 目录是一维结构（没有子目录），这就直接影响着访问的效率及管理的方便与否。另外，/dev 目录下的节点并不是按实际需要创建的，因此，该目录下存在大量实际不用的节点，但一般也不能轻易删除。

理想的／dev 目录应该是层次的、其规模是可伸缩的。DevFS 就是为达到此目的而设计

的。它在底层改写了用户与设备交互的方式和途径。它会给用户在两方面带来影响。首先，几乎所有的设备名称都做了改变，例如：“ /dev/hda ”是用户的硬盘，现在可能被定位于“ /dev/ide0/... ”。这一修改方案增大了设备可用的名字空间，且容许 USB 类和类似设备的系统集成。其次，不再需要用户自己创建设备节点。DevFS 的 /dev 目录最初是空的，里面特定的文件是在系统启动时、或是加载模块后驱动程序装入时建立的。当模块和驱动程序卸载时，文件就消失了。为保持和旧版本的兼容，可以使用一个用户空间守护程序“ devfsd ”，以使先前的设备名称能继续使用。目前，DevFS 的使用还只是一个实验性选项，由一个编译选项 CONFIG_DEVFS_FS 加以选择。

（1）注册和注销字符设备驱动程序

如前所述，一个新的文件系统要加入系统，必须进行注册。那么，一个新的驱动程序要加入系统，也必须进行注册。在下一章我们会看到，我们把设备大体分为字符设备和块设备。字符设备的注册和注销调用 register_chrdev()和 unregister_chrdev()函数。注册了设备驱动程序以后，驱动程序应该调用 devfs_register()登记设备的入口点，所谓设备的入口点就是设备所在的路径名；在注销设备驱动程序之前，应该调用 devfs_unregister()取消注册。

devfs_register()和 devfs_unregister() 函数原型为：

```
devfs_handle_t devfs_register (devfs_handle_t dir, const char *name,
    unsigned int flags,
    unsigned int major, unsigned int minor,
    umode_t mode, void *ops, void *info) ;

void devfs_unregister (devfs_handle_t de) ;
```

其中 devfs_handle_t 表示 DevFS 的句柄（一个结构类型），每个参数的含义如下。

dir：我们要创建的文件所在的 DevFS 的句柄。NULL 意味着这是 DevFS 的根，即/dev。

flags：设备文件系统的标志，缺省值为 DEVFS_FL_DEFAULT。

major：主设备号，普通文件不需要这一参数。

minor：次设备号，普通文件也不需要这一参数。

mode：缺省的文件模式（包括属性和许可权）。

ops：指向 file_operations 或 block_device_operations 结构的指针。

info：任意一个指针，这个指针将被写到 file 结构的 private_data 域。

例如，如果我们要注册的设备驱动程序叫做 DEVICE_NAME，其主设备号为 MAJOR_NR，次设备号为 MINOR_NR，缺省的文件操作为 device_fops，则该设备驱动程序的 init_module（）函数和 cleanup_module（）函数如下：

```
int init_module (void)
{
   int ret;

    if ( (ret = register_chrdev (MAJOR_NR, DEVICE_NAME, &device_fops) ) ==0)
        return ret;
}

void cleanup_module (void)
{
   unregister_chrdev (MAJOR_NR, DEVICE_NAME) ;
```

```
}
```

对以上代码进行改写以支持设备文件系统（假定设备入口点的名字为 DEVICE_ENTRY）。

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

devfs_handle_t devfs_handle;

int init_module (void)
{
  int ret;

 if  ( (ret = devfs_register_chrdev (MAJOR_NR, DEVICE_NAME, &device_fops) )  == 0)
      return ret;

 devfs_handle = devfs_register (NULL, DEVICE_ENTRY, DEVFS_FL_DEFAULT,
      MAJOR_NR, MINOR_NR, S_IFCHR | S_IRUGO | S_IWUSR,
      &device_fops, NULL) ;
}

void cleanup_module (void)
 {
   devfs_unregister_chrdev (MAJOR_NR, DEVICE_NAME) ;
   devfs_unregister (devfs_handle) ;
 }
```

(2) 在 DevFS 名字空间中创建一个目录

devfs_mk_dir()用来创建一个目录，这个函数返回 DevFS 的句柄，这个句柄用作 devfs_register 的参数 dir。

例如，为了在“ /dev/mydevice ”目录下创建一个设备设备入口点，则进行如下操作：

```
devfs_handle = devfs_mk_dir (NULL, "mydevice", NULL) ;
devfs_register (devfs_handle, DEVICE_ENTRY, DEVFS_FL_DEFAULT,
     MAJOR_NR, MINOR_NR, S_IFCHR | S_IRUGO | S_IWUSR,
     &device_fops, NULL) ;
```

(3) 注册一系列设备入口点

如果一个设备有几个从设备号，就说明同一个设备驱动程序控制了几个不同的设备，例如主 IDE 硬盘的主设备号为 3，但其每个分区都有一个从设备号，例如/dev/had2 的从设备号为 2。在 DevFS 下，每个从次设备号也有一个目录，例如/dev/ide0/,/dev/ide1/等，也就是说，每个从设备号都有一个设备入口点，于是就可以调用 devfs_register_series 来创建一系列的设备入口点。设备入口点的名字以 printf()函数中 format 参数的形式来创建。

注册 DEVICE_NR 设备入口点（从设备号从 MINOR_START 开始）的操作如下：

```
devfs_handle = devfs_mk_dir (NULL, "mydevice", NULL) ;

devfs_register_series (devfs_handle, "device%u", max_device, DEVFS_FL_ DEFAULT,
 MAJOR_NR, MINOR_START, S_IFCHR | S_IRUGO | S_IWUSR,
 &device_fops, NULL) ;
```

(4) 块设备

注册和注销块设备的函数为：

```
devfs_register_blkdev()
devfs_unregister_blkdev ()
```

3．使用/proc 文件系统

/proc 是一个特殊的文件系统，其安装点一般都固定为/proc。这个文件系统中所有的文件都是特殊文件，其内容不存在于任何设备上。每当创建一个进程时，系统就以其 pid 为文件名在这个目录下建立起一个特殊文件，使得通过这个文件就可以读／写相应进程的用户空间，而当进程退出时则将此文件删除。

/proc 文件系统中的目录项结构 dentry，在磁盘上没有对应结构，而以内存中的 proc_dir_entry 结构来代替，在 include/linux/proc_fs.h 中定义如下：

```
struct proc_dir_entry {
         unsigned short low_ino;
         unsigned short namelen;
         const char *name;
         mode_t mode;
         nlink_t nlink;
         uid_t uid;
         gid_t gid;
         unsigned long size;
         struct inode_operations * proc_iops;
         struct file_operations * proc_fops;
         get_info_t *get_info;
         struct module *owner;
         struct proc_dir_entry *next, *parent, *subdir;
         void *data;
         read_proc_t *read_proc;
         write_proc_t *write_proc;
        atomic_t count;         /* use count */
            int deleted;            /* delete flag */
         kdev_t  rdev;
 };
```

注册和注销/proc 文件系统的机制已经发生了变化。在 Linux 2.2 中，proc_dir_entry 结构是静态定义和初始化的，而在 Linux 2.4 中，这个数据结构被动态地创建。

（1）传送的数据小于一个页面的大小

当传送的数据小于一个页面大小时，/proc 文件系统的实现可以通过 proc_dir_entry 中的 read_proc 和 write_proc 方法来实现。

假定我们要注册的/proc 文件系统名为“ foo ”，在 Linux 2.2 中的代码如下。

foo_proc_entry 结构的初始化：

```
    struct proc_dir_entry foo_proc_entry = {
        namelen: 3,
        name : "foo",
        mode : S_IRUGO | S_IWUSR,
        read_proc : foo_read_proc,
        write_proc : foo_write_proc,
    };
```

proc 文件系统根节点，即目录项 proc_root 的初始化为：

```
    struct proc_dir_entry proc_root = {
          low_ino:        PROC_ROOT_INO,
          namelen:        5,
```

```
            name:           "/proc",
            mode:           S_IFDIR | S_IRUGO | S_IXUGO,
            nlink:          2,
            proc_iops:      &proc_root_inode_operations,
            proc_fops:      &proc_root_operations,
            parent:         &proc_root,
        };
```

注册：

```
      proc_register (&proc_root, &foo_proc_entry) ;
```

注销：

```
       proc_unreigster (&proc_root, foo_proc_entry.low_ino) ;
```

在 Linux 2.4 中。

注册：

```
    struct proc_dir_entry *ent;

    if  ( (ent = create_proc_entry ("foo", S_IRUGO | S_IWUSR, NULL) )  != NULL)  {
        ent->read_proc = foo_read_proc;
        ent->write_proc = foo_write_proc;
      }
```

注销：

```
    remove_proc_entry ("foo", NULL) ;
```

(2) 传送数据大于一个页面大小

当传送数据大于一个页面大小时，/proc 文件系统的实现应当通过完整的 file 结构来实现：

在 Linux 2.2 中。

相关数据结构为：

```
    struct file_operations foo_file_ops = {
     ......
    };

   struct inode_operations foo_inode_ops = {
      default_file_ops : &foo_file_ops;
   };

    struct proc_dir_entry foo_proc_entry = {
      namelen: 3,
      name : "foo",
      mode : S_IRUGO | S_IWUSR,
      ops : &foo_inode_ops,
   };
```

注册为：

```
    proc_register (&proc_root, &foo_proc_entry) ;
```

注销为：

```
    proc_unreigster (&proc_root, foo_proc_entry.low_ino) ;
```

在 Linux 2.4 中。

相关数据结构为：

```
    struct file_operations foo_file_ops = {
```

```
    ......
};

struct inode_operations foo_inode_ops = {
    ......
};
```

注册为：

```
struct proc_dir_entry *ent;

if ( (ent = create_proc_entry ("foo", S_IRUGO | S_IWUSR, NULL) ) != NULL) {
    ent->proc_iops = &foo_inode_ops;
    ent->proc_fops = &foo_file_ops;
}
```

注销为：

```
remove_proc_entry("foo", NULL);
```

3．块设备驱动程序

块设备驱动程序的界面有了很大的变化，新引入了 block_device_operations 结构，缓冲区高速缓存的接口也发生了变化。

(1) 设备注册

在 Linux 2.2 中，块设备与字符设备驱动程序的注册基本相同，都是通过 file_operations 结构进行的。在 Linux 2.4 中，引入了新结构 block_device_operations。例如，块设备的名字为 DEVICE_NAME，主设备号为 MAJOR_NR，则在 Linux 2.2 中如下所述。

数据结构为：

```
struct file_operations device_fops = {
   open : device_open,
   release : device_release,
   read : block_read,
   write : block_write,
   ioctl : device_ioctl,
   fsync : block_fsync,
};
```

注册为：

```
register_blkdev (MAJOR_NR, DEVICE_NAME, &device_fops) ;
```

在 Linux 2.4 中。

数据结构为：

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

struct block_device_operations device_fops = {
open : device_open,
release : device_release,
ioctl : device_ioctl,
};
```

注册为：

```
register_blkdev (MAJOR_NR, DEVICE_NAME, &device_fops) ;
```

(2) 缓冲区高速缓存接口

在块设备驱动程序中，有一个“请求函数”来处理缓冲区高速缓存的请求。在 Linux 2.2 中，请求函数的注册和定义如下。

函数原型为：

```
    void device_request (void) ;
```

注册为：

```
     blk_dev[MAJOR_NR].request_fn = &device_request;
```

请求函数的定义为：

```
    void device_request (void)
    {
     while  (1)  {
          INIT_REQUEST;

          ......

          switch  (CURRENT->cmd)  {
          case READ :
               // read
               break;
          case WRITE :
               // write
               break;
          default :
               end_request (0) ;
               continue;
          }

          end_request (1) ;
        }
    }
```

在 Linux 2.4 中。

函数原型为：

```
    int device_make_request (request_queue_t *q, int rw, struct buffer_head *sbh) ;
```

注册：

```
blk_queue_make_request (BLK_DEFAULT_QUEUE (MAJOR_NR) ,&device_make_request) ;
```

请求函数的定义为：

```
    int device_make_request (request_queue_t *q, int rw, struct buffer_head *sbh)
    {
       char *bdata;
        int ret = 0;

        ......

        bdata = bh_kmap (sbh) ;

        switch  (rw)  {
        case READ :
          // read
          break;
        case READA :
```

```
            // read ahead
            break;
          case WRITE :
            // write
            break;
          default :
            goto fail;
          }

          ret = 1;

        fail:
          sbh->b_end_io (sbh, ret) ;
          return 0;
        }
```

其中 request_queue_t 类型的定义请参见下一章，bh_kmap（）函数获得内核映射图。

4．PCI 设备驱动程序

Linux 2.4 包含了具有全部特征的资源管理子系统。它提供了“即插即用”功能， PCI 子系统也随之经历了改变。在 Linux 2.2 中，设备驱动程序搜索所驱动的设备，在 Linux 2.4 中，当驱动程序初始化时就注册设备的信息，当找到一个设备时 PCI 子系统就调用设备的初始化程序。

（1）驱动程序注册

假设要驱动的设备其商家 id 和设备 id 分配为 VENDOR_ID 和 DEVICE_ID，在 Linux 2.2 中，设备初始化函数如下：

```
struct pci_dev *pdev = NULL;

while  ( (pdev = pci_find_device (VENDOR_ID, DEVICE_ID, pdev) )  != NULL)  {
    // initialize each device
}
```

在 Linux 2.4 中。

数据结构为：

```
struct pci_device_id device_pci_tbl[] __initdata = {
    { VENDOR_ID, DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID },
    { 0, 0, 0, 0 },
};

int device_init_one (struct pci_dev *dev, const struct pci_device_id *ent) ;
void device_remove_one (struct pci_dev *pdev) ;

struct pci_driver device_driver = {
  name :      DEVICE_NAME,
  id_table :  device_pci_tbl,
  probe :          device_init_one,
  remove :         device_remove_one,
  suspend :        device_suspend,
  resume :         device_resume,
};
```

注册为：

```
if (pci_register_driver (&device_driver) <= 0)
    return -ENODEV;
```

注销为：

```
pci_unregister_driver (&device_driver) ;
```

5．文件系统的移植问题文件系统的移植问题，在此不赘述

6．下半部分（bottom half）处理程序、软中断（softirq）及 tasklets

为了处理硬件中断之外的中断，Linux 2.2 提供了下半部分。Linux 2.4 提供了两种新的机制：软中断及 tasklet。软中断在 SMP 上不是串行化执行，而是同一个处理程序可以在多个 CPU 上同时执行。为了提高 SMP 的性能，软中断机制现在主要用于网络子系统。对于 tasklet 来说，多个 tasklet 可以在多个 CPU 上执行，但一个 CPU 一次只能处理一个 tasklet。下半部分（bh）是由内核串行执行的，即使在 SMP 环境下，一个 CPU 也只能处理一个下半部分。因此，下半部分变得过时，一般情况下，使用 tasklet 就足够了。把下半部分移植到 tasklet 的具体内容请参看第三章的 3.5.6 节。

7．链表及等待队列

（1）通用双向链表

Linux 2.2 是以宏和内联函数的形式来定义通用链表的。Linux 2.2 主要在文件系统中使用了这种链表，而 Linux 2.4 使用得更加普遍（例如等待队列）。

在 include/linux/list.h 中定义的通用链表 list_head 如下：

```
struct list_head {
        struct list_head *next, *prev;
    };
```

如果我们定义一个整型数据的链表，则其定义如下：

```
struct foo_list {
    int data;
    struct list_head list;
};
```

然后，链表的头应该定义如下：

```
LIST_HEAD (foo_list_head) ;
```

或者为

```
struct list_head foo_list_head = LIST_HEAD_INIT (data_list_head) ;
```

或者为：

```
struct list_head foo_list_head;
INIT_LIST_HEAD (&foo_list_head) ;
```

现在，我们可以用 list_add()为链表增加一个节点，用 list_del()删除一个节点：

```
struct foo_list data;
list_add (&data.list, &foo_list_head) ;
list_del (&data.list) ;
```

使用 list_for_each()和 list_entry()来遍历链表：

```
struct list_head *head, *curr;
```

```
    struct foo_list *element;

    head = &foo_list_head;
    curr = head->next;

    list_for_each (curr, head)
        element = list_entry (curr, struct foo_list, list) ;
```

(2) 等待队列

Linux 2.2 以单链表实现了任务的等待队列，而 Linux 2.4 用通用双向链表实现了等待队列。

在 Linux 2.2 中。

队列的定义为：

```
    struct wait_queue *wq = NULL;
```

睡眠和唤醒为：

```
    interruptible_sleep_on (&wq) ;
    wake_up_interruptible (&wq) ;
```

在 Linux 2.4 中，等待队列的定义发生了变化，但实现函数还是一样的。

队列定义为：

```
    DECLARE_WAIT_QUEUE_HEAD (wq) ;
```

或者为：

```
    wait_queue_head_t wq;
    init_waitqueue_head (&wq) ;
```

10.5　编写内核模块

要写一个内核模块，你必须懂得 C 语言，并且你曾经写过一些程序当作进程而运行。但是，编写内核模块和编写一般的程序有很大的不同，首先，你要明白，内核模块一旦插入到内核，它将成为内核的一部分，这意味着仅仅无法控制的一个指针可能导致内核的崩溃；其次，你必须了解一些内核函数的使用方法和功能，尤其是和编写模块相关的函数；最后，要注意内核的版本号，如果你的模块是基于 2.0 版本编写的，你可能希望移植到 Linux 2.2；或者是基于 2.2 版本编写的，而希望移植到 2.4，这就需要知道这些版本的变化对编写模块程序的影响。

Linux 的内核是非常庞大的，但作为一个编程者，你应当至少读一些内核的源文件并弄懂它。然后，编写一个简单的程序，试试到底怎样编写一个内核模块程序。

10.5.1　简单内核模块的编写

一个内核模块应当至少有两个函数，第 1 个为 init_moudle，当模块被插入到内核时调用它；第 2 个为 cleanup_module，当模块从内核移走时调用它。init_module 的主要功能是在内核中注册一个处理某些事的处理程序。cleanup_module 函数的功能是取消 init_module 所做的事情。

下面看一个例子“Hello,world！”。

```
/ * hello.c
  * "Hello,world"   */

/*下面是必要的头文件*/

#include <linux/kernel.h>   /* 内核模块共享这个头文件 */
 #include <linux/module.h>   /* 这是一个模块 */

/* 处理 CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/*初始化模块 */
int init_module ()
{
  printk ("Hello, world - this is a simple module\n") ;

  /* 如果返回一个非 0，那就意味着 init_module 失败，不能装载该内核模块*/
  return 0;
}


/* 取消 init_module 所作的工作*/
void cleanup_module ()
{
  printk ("the module exits the kernel\n") ;
}
```

10.5.2 内核模块的 Makefiles 文件

内核模块不是独立的可执行文件，但在运行时其目标文件被连接到内核中，因此，编译内核模块时必须加 -c 标志，另外， 还得加确定的预定义符号。

__KERNEL__ -- 相当于告诉头文件，这个代码必须运行在内核模式下，而不是用户进程的一部分。

MODULE -- 这个标志告诉头文件，要给出适当的内核模块的定义。

LINUX -- ，从技术上讲，这个标志不是必要的。但是，如果你希望写一个比较正规的内核模块，在多个操作系统上能进行编译，这个标志将会使你感到方便。它可以允许你在独立于操作系统的部分进行常规的编译。

还有其他的一些标志是否被包含进去，这取决于编译模块时的选项。如果你不能明确内核怎样被编译，可以在 in/usr/include/linux/config.h 中查到。

__SMP__ -- ，对称多处理机。如果内核被编译成支持对称多处理机（即使它只不过运行在单个 CPU 上），这必须被定义。如果你要用对称多处理机，还有一些其他的事情必须做，在此不进行详细的讨论。

CONFIG_MODVERSIONS -- ，如果 CONFIG_MODVERSIONS 被激活，当编译内核模块时，你必须定义它，并且包含进 usr/include/linux/modversions.h 中，这也可以由代码本身来做。

Makefile 举例

```
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
         $ (CC)  $ (MODCFLAGS)  -c hello.c
         echo insmod hello.o to turn it on
         echo rmmod hello to turn it off
         echo
```

现在，你以 root 的身份对这个内核模块进行编译并连接后，形成一个目标文件 hello.o，然后用 insmod 把 hello 插入到内核，也可以用 rmmod 命令把 hello 从内核移走。如果你想知道结果如何，你可以查看/proc/modules 文件，从中会找到一个新加入的模块。

10.5.3　内核模块的多个文件

有时，可以从逻辑上把内核模块分成几个源文件，在这种情况下，需要做以下事情。

（1）除了一个源文件外，在其他所有的源文件中都要增加一行 #define __NO_VERSION__，这是比较重要的，因为 module.h 通常包括了对 kernel_version 的定义，kernel_version 是一个具有内核版本信息的全局变量，并且编译模块时要用到它。如果你需要 version.h，你就必须自己包含它，但如果你定义了__NO_VERSION__，module.h 就不会被包含进去。

（2）像通常那样编译所有的源文件。

（3）把所有的目标文件结合到一个单独文件中。在 x86 下，这样连接：

```
ld -m elf_i386 -r -o <name of module>.o <第 1 个源文件>.o <第 2 个源文件>.o
```

请看下面例子 start.c。

```
 /* start.c
 *
 * "Hello, world"
 * 这个文件包含了启动例程
 */
/*下面是必要的头文件 */

/* 内核模块的标准形式*/
#include <linux/kernel.h>
#include <linux/module.h>

/* 处理 CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 初始化模块 */
int init_module ()
{
```

```
  printk ("Hello, world - this is the kernel speaking\n") ;

  return 0;
}
```

另一个例子 stop.c。

```
/* stop.c */
/* 这个文件仅仅包含 stop 例程。*/

/* 必要的头文件*/

#include <linux/kernel.h>

#define __NO_VERSION__
#include <linux/module.h>
#include <linux/version.h>

#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

void cleanup_module ()
{
  printk ("Short is the life of a kernel module\n") ;
}
```

下面是多个文件的 Makefile。

```
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: start.o stop.o
         ld -m elf_i386 -r -o hello.o start.o stop.o

start.o: start.c /usr/include/linux/version.h
         $ (CC)  $ (MODCFLAGS)  -c start.c

stop.o:       stop.c /usr/include/linux/version.h
         $ (CC)  $ (MODCFLAGS)  -c stop.c
```

“ hello ”是模块名，它占用了一页（4KB）的内存，此时，没有其他内核模块依赖它。要从内核移走这个模块，敲入“ rmmod hello ”，注意，rmmod 命令需要的是模块名而不是文件名。其他实用程序的使用可参看相关的文挡。

关于模块编程更多的内容我们将在后续章节继续讨论。