第1章 内核模块

模块最大的好处是可以动态扩展应用程序的功能而无须重新编译链接生成一个新的应用程序映像，这种广义上的模块概念其实并非 Linux 系统所特有，在微软的 Windows 系统上动态链接库 DLL（Dynamic Link Library）便是模块概念的一个典型应用场景，对应到 Linux 系统上这种模块以所谓的共享库 so（shared object）文件的形式存在[1]。

本章要讨论的主题——Linux 内核模块，在概念及原理方面与上面提到的 DLL 和 so 模块类似，但又有其独特的一面，内核模块可以在系统运行期间动态扩展系统功能而无须重新启动系统[2]，更无须为这些新增的功能重新编译一个新的系统内核映像。内核模块的这个特性为内核开发者开发验证新的功能提供了极大的便利，因为像 Linux 这么庞大的系统，编译一个新内核并重新启动将浪费开发者大量的时间。

虽然设备驱动程序并不一定要以内核模块的形式存在，并且内核模块也不一定就代表着一个设备驱动程序，但是内核模块的这种特性似乎注定是为设备驱动程序而生。Linux 系统下的设备驱动程序员在开发一个新的设备驱动的过程中，使用的最多的工具之一是 insmod，这是一个简单的向系统动态加载内核模块的命令。很难想象，如果没有 insmod 这样的机制，在 Linux 底下调试一个设备驱动会是怎样的一件让人痛苦崩溃的事情！笔者相信，任何一个在 Linux 上面有过实际的驱动程序开发经历的人都会有类似的感受。

Linux 系统虽然为内核模块机制提供了完善的支持，使得其下的内核模块是如此强大，然而现实中事情往往并非如预想的那样一帆风顺，如果对其幕后的机制不甚了解，在实际的开发过程之中，除了驱动程序自身要实现的功能可能会遇到麻烦以外，在使用 Linux 中的内核模块机制本身时，也会遇到各种各样的问题，比如在用 insmod 命令加载一个模块时，就很可能会碰到类似下面的错误信息：

```
root@AMDLinuxFGL:/# insmod demodev.ko
insmod: error inserting 'demodev.ko': -1 Invalid module format
```

如果 dmesg 一下，就会看到内核针对上述错误打印出的出错信息如下所示：

```
demodev: version magic '3.1.7 SMP mod_unload modversions' should be '3.1.7 SMP mod_unload '
```

[1] 在软件工程中，模块这一术语在不同的上下文环境中有不同的语义。本书中提到的模块特指某种动态或者静态链接库。因为静态库在原理上和动态库有很大的区别，所以本章提到的模块，背后都暗含着动态链接的思想，更具体地，就是以.ko 形式存在的所谓内核模块。

[2] 如果因为动态加载的模块自身的原因导致系统崩溃，则是另一回事了。

直觉上，这应该不是在驱动程序自身要实现的功能上出现了问题，问题应该出在驱动程序所在的模块在加载时与系统中内核模块框架互动的环节中。很明显，Linux 内核设计中为模块这种机制提供了完善的支持，以内核模块形式存在的设备驱动程序也必然要遵循这种框架下的规则才能正常工作，也许绝大多数情况下模块都会工作得很好，然而诸如上面提到的这类模块相关的错误也绝非罕见。

既然规则不由我们定义，那么了解并遵守规则就成了避免或者解决这类问题的唯一途径。一个成熟的 Linux 设备驱动程序开发者应该能很快确定这些错误的原因并给出相应的解决方案，而新手在这类错误面前更多的感觉则可能是迷惘和不知所措。因此，无论是出于现实工作的需要，还是为了满足自己的好奇心，Linux 下的设备驱动程序员都有必要花上足够多的时间来了解隐藏在内核模块背后的技术细节，而这也正是本章要深入探讨内核模块机制的目的。本章将重点关注并讨论如下的问题：

- 内核和模块导出符号机制。
- 模块的加载过程。
- 模块如何使用内核或者其他模块导出的函数与变量。
- 模块的参数传递机制。
- 模块之间的依赖关系。
- 模块中的版本控制机制。

1.1 内核模块的文件格式

以内核模块形式存在的驱动程序 ,比如 demodev.ko ,其在文件的数据组织形式上是 ELF(Executable and Linkable Format) 格式，更具体地，内核模块是一种普通的可重定位目标文件。用 file 命令查看 demodev.ko 文件，可以得到类似如下的输出：

dennis@AMDLinuxFGL:/$ file demodev.ko

demodev.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

ELF 是 Linux 下非常重要的一种文件格式，常见的可执行程序都是以 ELF 的形式存在。本书不会详细讨论 ELF 格式的技术细节，但是为了让读者能更好地理解后续的模块加载、导出符号和模块参数等相关主题，在这里我们结合 Linux 源代码中定义的 ELF 相关数据结构（基于 x86-64 位体系架构），给出 ELF 格式的一个比较详细的结构图，如图 1-1 所示（这张图在后续的小节中会被多次引用）。

图 1-1 中忽略了驱动程序模块 ELF 文件中不会用到的 Program header table。从图 1-1 可以看到，静态的 ELF 文件视图[3]总体上可分为三大部分 头部的 ELF header、中间的 Section 和尾部的 Section header table。

[3] 在说 ELF 文件视图时，它总是静态的。加上“静态的”是为了强调和“动态的”执行期 ELF 内存视图的对比。当然，ELF 文件被加载时总是会先被放到某段内存区域中，此时称为“静态的”内存视图；而在后续的模块加载处理过程中，其在内存中搬移的视图则称为“动态的”内存视图，简称内存视图。

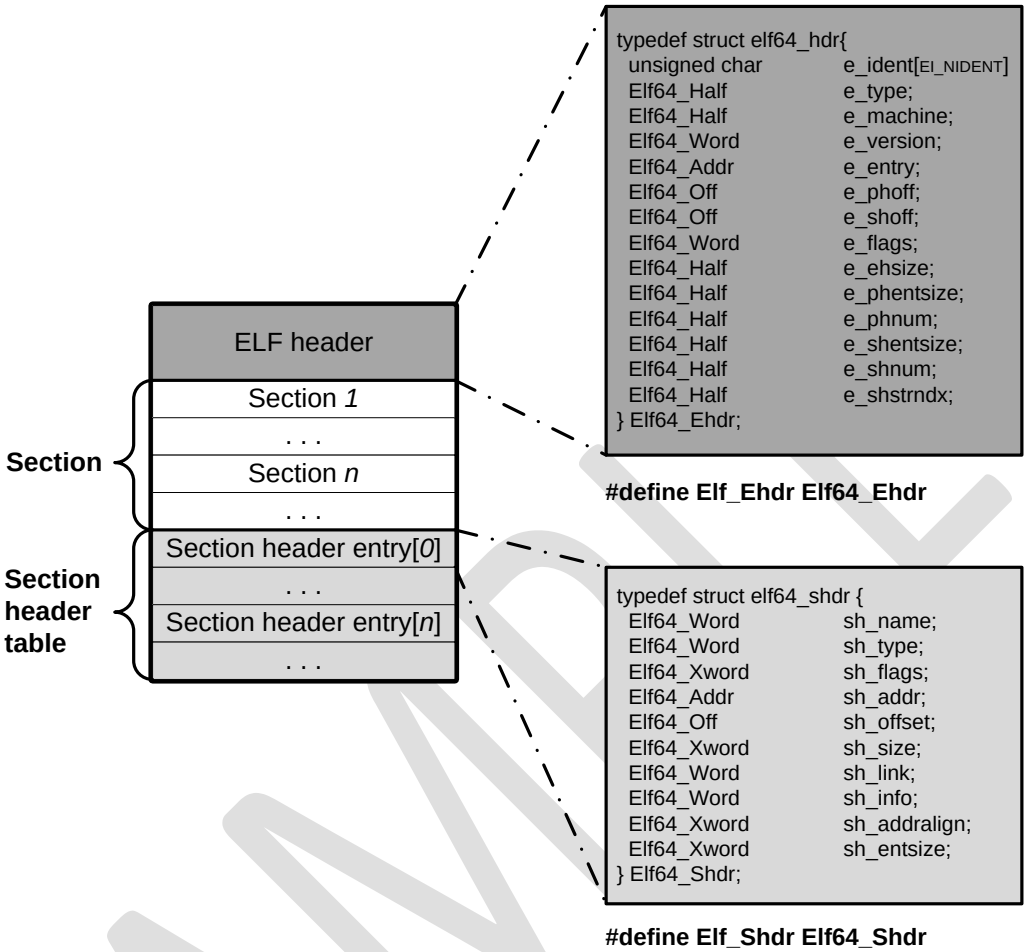

图 1-1 ELF 文件视图格式

❍ ELF header 部分

其大小是 64 字节[4]，位于文件头部。对于驱动模块文件而言，其中一些比较重要的数据成员如下：

e_type

表明文件类型，对于驱动模块，这个值是 1，也就是说驱动模块是一个可定位的 ELF 文件(relocatable file)。

e_shoff

表明 Section header table 部分在文件中的偏移量。

[4] 32 位系统中 ELF header 的大小为 52 字节。

e_shentsize

表明 Section header table 部分中每一个 entry[5]的大小（以字节计）。

e_shnum

表明 Section header table 中有多少个 entry。因此，Section header table 的大小便为 e_shentsize×e_shnum 个字节。

e_shstrndx

与 Section header entry 中的 sh_name 一起用来指明对应的 section 的 name。

❍ Section 部分

ELF 文件的主体，位于文件视图中间部分的一个连续区域中。但是当模块被内核加载时，会根据各自属性被重新分配到新的内存区域（有些 section 也可能只是起辅助作用，因而在运行时并不占用实际的内存空间）。

❍ Section header table 部分

该部分位于文件视图的末尾，由若干个（具体个数由 ELF header 中的 e_shnum 变量指定）Section header entry 组成，每个 entry 具有同样的数据结构类型。对于设备驱动模块而言，一些比较重要的数据成员如下：

sh_type

用来标识该 entry 所对应 section 的类型和语义，对于内核模块文件而言，常见的类型包括 SHT_PROGBITS、SHT_RELA、SHT_NOTE、SHT_SYMTAB 和 SHT_STRTAB。

sh_addr

这个值用来表示该 entry 所对应的 section 在内存中的实际地址。在静态的文件视图中，该值为 0，当模块被内核加载时，加载器会用该 section 在内存中的实际地址来改写 sh_addr（如果该 section 在运行期间不占用内存空间，那么该值为 0）。

sh_offset

表明对应的 section 在文件视图中的偏移量。

sh_size

[5] 在本章中，entry 具有特别的含义，特指 Section header table 中的一个 entry。在本章接下来的代码表示中会用下面一种简化的表达方式：entry[i]表示 Section header table 中第 i 个 entry。

表明对应的 section 在文件视图中的大小（以字节计）。类型为 SHT_NOBITS 的 section 例外，这种 section 在文件视图中不占有空间。

sh_entsize

用于一种特定的 section，该 section 的内容是一组固定大小的 entry 所构成的表，如符号表，此种情况下 sh_entsize 用来表示表中 entry 的大小。

以上简单介绍了内核模块所属 ELF 文件的一些主要数据成员，显然设备驱动程序并不会使用到这些数据，它们是给内核模块加载器在加载模块时使用的，这里只是为了给后续的模块加载过程的讨论做一个简单的技术铺垫（如果读者对 ELF 文件的技术细节感兴趣，这里推荐一个非常实用的在 Linux 环境下读取 ELF 文件信息的工具——readelf）。接下来在进行模块加载这个沉重的话题讨论前，先来看一个有趣的东西：模块是如何向外界导出符号信息的。

1.2 EXPORT_SYMBOL 的内核实现

看过 Linux 内核源码的读者应该知道，源码中充斥着像 EXPORT_SYMBOL 这样的宏，在我们自己的设备驱动程序中也经常会发现它的身影。大部分时间里，我们只知道它用来向外界导出一个符号，仅此而已。我们对这些宏是如此习惯，以至于常常忽略其存在的意义，更不用说去仔细探究其背后的实现原理了。然而这些不起眼的宏却有着大用场，如果没有它们，我们的驱动程序甚至连 printk 这样常见的内核函数都不能使用。

本节描述 EXPORT_SYMBOL、EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE 宏定义导出符号的内核机制。之所以把导出符号作为独立的一节，是因为在模块加载的过程中会使用本节描述到的机制，而且导出符号这一特性在 Linux 系统中对模块的存在具有重要意义。读者应该结合本节和模块加载部分的相关内容来理解如何导出符号和使用导出的符号这一重要的内核机制。

如果没有独立存在的内核模块，Linux 内核作为单一的内核映像，向外导出符号就失去了意义。对于静态编译链接而成的内核映像而言，所有的符号引用都将在静态链接阶段完成。然而，内核模块的出现让事情发生了变化：内核模块不可避免地要使用到内核提供的基础设施（以调用内核函数的形式发生），作为独立编译链接的内核模块，必须要解决这种静态链接无法完成的符号引用问题（在内核模块所在的 ELF 文件中，这种引用被称为“未解决的引用”[6]）。处理“未解决引用”问题的本质是在模块加载期间找到当前“未解决的引用”符号在内存中的实际目标地址。

内核和内核模块通过符号表的形式向外部世界导出符号的相关信息，这种导出符号的方式在代码层面以 EXPORT_SYMBOL 宏定义的形式存在。从全局来看，EXPORT_SYMBOL 这类宏功能的完整实现需要经过三个部分来完成：EXPORT_SYMBOL 宏定义部分，链接脚本链接器部分和使用导出符号部

[6] 在模块的链接阶段，链接器有可能无法找到一些符号(比如模块中调用到的 printk 函数)的目标代码，此时它在符号表中将该符号标记为未定义的(UNDEF)符号，也就是此处所说的“未解决的引用”符号。在模块的加载阶段，由内核模块加载器负责解析这些“未解决的引用”符号的目标地址。

分。本节讲述前两个部分，第三部分的描述将延后到模块加载的相关段落。

下面通过这些宏定义来仔细考察代码背后的技术细节。

```
<include/linux/module.h>

#define __EXPORT_SYMBOL(sym, sec)                       \
        extern typeof(sym) sym;                         \
        __CRC_SYMBOL(sym, sec)                          \
        static const char __kstrtab_##sym[]             \
        __attribute__((section("__ksymtab_strings"), aligned(1))) \
        = MODULE_SYMBOL_PREFIX #sym;                            \
        static const struct kernel_symbol __ksymtab_##sym       \
        __used                                          \
        __attribute__((section("___ksymtab" sec "+" #sym), unused)) \
        = { (unsigned long)&sym, __kstrtab_##sym }

#define EXPORT_SYMBOL(sym)                              \
        __EXPORT_SYMBOL(sym, "")

#define EXPORT_SYMBOL_GPL(sym)                                  \
        __EXPORT_SYMBOL(sym, "_gpl")

#define EXPORT_SYMBOL_GPL_FUTURE(sym)                           \
        __EXPORT_SYMBOL(sym, "_gpl_future")
```

以上为来自 Linux 源码树中的 EXPORT_SYMBOL 等相关宏的定义细节。其中的__CRC_SYMBOL 用来作为版本控制信息使用，在本章后续的“模块的版本控制”一节中将予以讨论。在接下来的分析中，为了使读者更清楚其中的实现细节，我会对内核中的源码稍作改写，这种改写并不会改变原来代码的本质，而只是为了让代码看起来更加清楚简洁。此外，为叙述简单起见，我将用 EXPORT_SYMBOL(my_exp_function)作为一个具体的例子，即向模块的外部导出一个名为 my_exp_function 的函数，这个导出函数的例子同样也用在 EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE 中。

从前面的源代码可以看出，每个 EXPORT_SYMBOL 宏实际上定义了两个变量：

static const char ***__kstrtab_my_exp_function** = "my_exp_function";

static const struct kernel_symbol **__ksymtab_my_exp_function** =

{ (unsigned long)& my_exp_function, __kstrtab_my_exp_function };

第一个变量是个简单的 char 型指针，用来表示导出的符号名称；第二个变量类型是 struct kernel_symbol 数据结构，用来表示一个内核符号，struct kernel_symbol 的定义为：

<include/linux/module.h>

```
struct kernel_symbol
{
    unsigned long value;
    const char *name;
};
```

其中，value 是该符号的目标地址，name 是符号名。所以，单由该数据结构可以知道，用 EXPORT_SYMBOL(my_exp_function)来导出符号“my_exp_function”，实际上是要通过 struct kernel_symbol 的一个对象告诉外部世界关于这个符号的两点信息：符号名称和目标地址[7]。

可见，用 EXPORT_SYMBOL 等宏来导出一个符号，本质上就是基于该符号产生一个 struct kernel_symbol 类型的对象，该对象将被放在一个指定的 section 中，而符号名则放到另一个指定的 section 中。

以 EXPORT_SYMBOL(my_exp_function)为例，展开后的符号名“**my_exp_function**”会被放置在一个名为“__ksymtab_strings”的 section 中，而 struct kernel_symbol 的对象 **__ksymtab_my_exp_function** 会放置在一个名为“__ksymtab+my_exp_function”的 section 中（对于 EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE 而言，其 struct kernel_symbol 对象所在的 section 名称则分别为“__ksymtab_gpl+my_exp_function”和“__ksymtab_gpl_future+ my_exp_function”）。

对这些 section 的使用需要经过一个重要的中间环节，即链接脚本与链接器部分。链接脚本告诉链接器把所有目标文件中的名为“__ksymtab+*”的 section 放置在最终内核（或者是内核模块）映像文件的名为“__ksymtab”的 section 中（对于目标文件中的名为“__ksymtab_gpl+*”、“__ksymtab_gpl_future+*”、“__kcrctab+*”、“__kcrctab_gpl+*”和“__kcrctab_gpl_future+*”的 section 都同样处理）。下面是 Linux 内核映像在链接阶段所依据的链接脚本的相关片段：

<arch/x86/kernel/vmlinux.lds>

[7] 对于由内核模块导出的符号而言，由于在静态链接时无法确定该符号在内存中的最终地址，因此这个地址信息要一直等到模块被成功加载进系统后才有效。在模块加载的过程中，由内核模块加载器负责修改该成员以反映出符号在内存中的最终目标地址，这也就是所谓的“重定位”过程。

```
__ksymtab : AT(ADDR(__ksymtab) - 0xffffffff80000000)
{ __start___ksymtab = .; *(SORT(___ksymtab+*))  __stop___ksymtab = .; }

__ksymtab_gpl : AT(ADDR(__ksymtab_gpl) - 0xffffffff80000000)
{ __start___ksymtab_gpl = .; *(SORT(___ksymtab_gpl+*))  __stop___ksymtab_gpl = .; }

__ksymtab_gpl_future : AT(ADDR(__ksymtab_gpl_future) - 0xffffffff80000000)
{ __start___ksymtab_gpl_future = .;
*(SORT(___ksymtab_gpl_future+*))  __stop___ksymtab_gpl_future = .; }

__kcrctab : AT(ADDR(__kcrctab) - 0xffffffff80000000)
{ __start___kcrctab = .; *(SORT(___kcrctab+*))  __stop___kcrctab = .; }

__kcrctab_gpl : AT(ADDR(__kcrctab_gpl) - 0xffffffff80000000)
{ __start___kcrctab_gpl = .; *(SORT(___kcrctab_gpl+*))  __stop___kcrctab_gpl = .; }

__kcrctab_gpl_future : AT(ADDR(__kcrctab_gpl_future) - 0xffffffff80000000)
{ __start___kcrctab_gpl_future = .;
*(SORT(___kcrctab_gpl_future+*))  __stop___kcrctab_gpl_future = .; }

__ksymtab_strings : AT(ADDR(__ksymtab_strings) - 0xffffffff80000000)
{ *(__ksymtab_strings) }
```

脚本的主要作用是将目标文件中所有的诸如“__ksymtab+*”的 section 合并到一个新的名为“__ksymtab”的 section 中。以上是 Linux 内核的链接脚本，那么对于模块而言，它的链接脚本在什么地方呢？答案是：

```
<scripts/module-common.lds>
SECTIONS {
    /DISCARD/ : { *(.discard) }
    __ksymtab           : { *(SORT(___ksymtab+*)) }
    __ksymtab_gpl       : { *(SORT(___ksymtab_gpl+*)) }
```

```
    __ksymtab_gpl_future    : { *(SORT(___ksymtab_gpl_future+*)) }
    __kcrctab               : { *(SORT(___kcrctab+*)) }
    __kcrctab_gpl           : { *(SORT(___kcrctab_gpl+*)) }
    __kcrctab_gpl_future    : { *(SORT(___kcrctab_gpl_future+*)) }
}
```

在我们的示例源码中，曾用 EXPORT_SYMBOL(my_exp_function)向外界导出了一个符号，根据前面对于 EXPORT_SYMBOL 宏的描述，在目标文件中应该会产生一个名为“__ksymtab+ my_exp_function”的 section，读者也许想急于验证这一点，那么可以用 objdump 工具来完成，但是这里的目标文件不应该是 demodev.ko，因为这个最终的内核模块映像已经过了模块链接脚本的处理，最终只会有“__ksymtab” section 存在，而不会出现“__ksymtab+ my_exp_function” section。我们应该用“objdump -x demodev.o”命令，这样才可以看到“__ksymtab+ my_exp_function” section。

这里之所以要把所有向外界导出的符号统一放到这些特殊的 section 里面，是为了在加载其他模块时用来查找那些“未解决的引用”符号，在稍后的“模块的加载过程”一节中可看到这种用途。注意这里由链接脚本定义的几个变量：__start___ksymtab、__stop___ksymtab、__start___ksymtab_gpl、__stop___ksymtab_gpl、__start___ksymtab_gpl_future、__stop___ksymtab_gpl_future，它们会在对内核导出的符号表进行查找时被用到。

内核源码中为使用这些由链接器产生的变量作了如下的声明：

<kernel/module.c>

```
extern const struct kernel_symbol __start___ksymtab[];
extern const struct kernel_symbol __stop___ksymtab[];
extern const struct kernel_symbol __start___ksymtab_gpl[];
extern const struct kernel_symbol __stop___ksymtab_gpl[];
extern const struct kernel_symbol __start___ksymtab_gpl_future[];
extern const struct kernel_symbol __stop___ksymtab_gpl_future[];
extern const unsigned long __start___kcrctab[];
extern const unsigned long __start___kcrctab_gpl[];
extern const unsigned long __start___kcrctab_gpl_future[];
```

如此，内核代码便可以直接使用这些变量而不会引起编译错误。内核模块加载器在处理模块中“未解决的引用”的符号时，会使用到这里定义的变量。读者在前面模块的链接脚本中也许注意到，模块除

了重新组织了一下 section 的布局之外，并没有如内核那样产生出类似__start___ksymtab 这样的变量，这使得对模块导出符号的查找和对内核导出符号的查找有了不同的处理方式，本章后续的章节会详细讨论这个问题。

1.3 模块的加载过程

在用户空间，用 insmod 这样的命令来向内核空间安装一个内核模块，本节将详细讨论模块加载时的内核行为。当调用“insmod demodev.ko”来安装 demodev.ko 这样的内核模块时，insmod 会首先利用文件系统的接口将其数据读取到用户空间的一段内存中，然后通过系统调用 sys_init_module 让内核去处理模块加载的整个过程。

1.3.1 sys_init_module（第一部分）

sys_init_module 的函数原型为：

long sys_init_module(void __user *umod, unsigned long len, const char __user *uargs);

其中，第一参数 umod 是指向用户空间 demodev.ko 文件映像数据的内存地址，第二参数 len 是该文件的数据大小，第三参数 uargs 是传给模块的参数在用户空间下的内存地址。

在 sys_init_module 函数中，加载模块的任务主要是通过调用 load_module 函数来完成的，该函数的定义为：

<kernel/module.c>

```
static struct module *load_module(void __user *umod,
                                  unsigned long len,
                                  const char __user *uargs)
```

所有参数同 sys_init_module 函数中的完全一样，实际上在 sys_init_module 函数的一开始便会调用该函数，调用时传入的实参完全来自于 sys_init_module 函数，没有经过任何的处理或者修改。

为了更清楚地解释模块加载时的内核行为，我们把 sys_init_module 分为两个部分：第一部分是调用 load_module，完成模块加载最核心的任务；第二部分是在模块被成功加载到系统之后的后续处理。我们将在讨论完 load_module 部分之后再继续讨论 sys_init_module 的第二部分。不过，在继续 load_module 话题之前，先要看一个内核中非常重要的数据结构——struct module。

1.3.2 struct module

load_module 函数的返回值是一个 struct module 类型的指针，struct module 是内核用来管理系统中加载的模块时使用的一个非常重要的数据结构，一个 struct module 对象代表着现实中一个内核模块在 Linux 系统中的抽象，该结构的定义如下（删除了一些 trace 和 unused symbol 相关的部分）：

<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_attribute    *modinfo_attrs;
        const char *version;
        const char *srcversion;
        struct kobject *holders_dir;

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

        /* Kernel parameters. */
        struct kernel_param *kp;
        unsigned int num_kp;

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

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

    /* Exception table */
    unsigned int num_exentries;
    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 int init_size, core_size;

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

    /* Size of RO sections of the module (text+rodata) */
    unsigned int init_ro_size, core_ro_size;

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

    unsigned int taints;  /* same bits as kernel:tainted */
```

```
#ifdef CONFIG_KALLSYMS
        /*
         * We keep the symbol and string tables for kallsyms.
         * The core_* fields below are temporary, loader-only (they
         * could really be discarded after module init).
         */
        Elf_Sym *symtab, *core_symtab;
        unsigned int num_symtab, core_num_syms;
        char *strtab, *core_strtab;

        /* Section attributes */
        struct module_sect_attrs *sect_attrs;

        /* Notes attributes */
        struct module_notes_attrs *notes_attrs;
#endif
/* The command line arguments (may be mangled).   People like
keeping pointers to this stuff */
        char *args;
#ifdef CONFIG_SMP
        /* Per-cpu data. */
        void __percpu *percpu;
        unsigned int percpu_size;
#endif

#ifdef CONFIG_MODULE_UNLOAD
        /* What modules depend on me? */
        struct list_head source_list;
        /* What modules do I depend on? */
        struct list_head target_list;
```

```
    /* Who is waiting for us to be unloaded */
    struct task_struct *waiter;
    /* Destruction function. */
    void (*exit)(void);

    struct module_ref {
        unsigned int incs;
        unsigned int decs;
    } __percpu *refptr;
#endif

#ifdef CONFIG_CONSTRUCTORS
    /* Constructor functions. */
    ctor_fn_t *ctors;
    unsigned int num_ctors;
#endif
};
```

我们很快就会在后续的模块加载部分看到使用这些成员的具体代码，现在先把一些重要的成员变量简单描述如下：

enum module_state state

用于记录模块加载过程中不同阶段的状态，module_state 的定义如下：

```
enum module_state
{
    //模块被成功加载进系统时的状态
    MODULE_STATE_LIVE,
    //模块正在加载中
    MODULE_STATE_COMING,
    //模块正在卸载中
    MODULE_STATE_GOING,
```

};

struct list_head list

用来将模块链接到系统维护的内核模块链表中，内核用一个链表来管理系统中所有被成功加载的模块。

char name[MODULE_NAME_LEN]

模块名称。

const struct kernel_symbol *syms

内核模块导出的符号所在起始地址。

const unsigned long *crcs

内核模块导出符号的校验码所在起始地址。

struct kernel_param *kp

内核模块参数所在的起始地址。

int (*init)(void)

指向内核模块初始化函数的指针，在内核模块源码中由 module_init 宏指定。

struct list_head source_list
struct list_head target_list

用来在内核模块间建立依赖关系。

1.3.3 load_module

作为内核模块加载器中最核心的函数，load_module 负责最艰苦的模块加载全过程。我们将仔细讨论该函数，因为除了可以了解内核模块加载的幕后机制之外，还能了解到一些非常有趣的特性，诸如内核模块如何调用内核代码导出的函数，被加载的模块如何向系统中其他的模块导出自己的符号，以及模块如何接收外部的参数等。在介绍这部分内容时，如果完全按照内核代码的顺序依序进行的话，逻辑上可能会显得比较凌乱。所以此处文字组织的基本思路是：将 load_module 函数按照各主要功能分成若干部分，各部分在下文中的出现顺序尽可能维持在代码中的出现顺序；如果某些功能之间存在着某种依赖关系，比如有 A 和 B 两个功能，A 功能的叙述需要用到 B 功能中提供的机制，则先介绍 B 功能；独立于功能模块之外的一些基础设施，比如某些功能性函数，则尽量往前放。且处于篇幅的考虑，我会将一些不太常见也不太重要的内容忽略掉。

❍ 模块 ELF 静态的内存视图

如图 1-2 所示，用户空间程序 insmod 首先通过文件系统接口读取内核模块 demodev.ko 的文件数据，将其放在一块用户空间的存储区域中（图中 void *umod 所示）。然后通过系统调用 sys_init_module 进入到内核态，同时将 umod 指针作为参数传递过去（同时传入的还有 umod 所指向的空间大小 len 和存放有模块参数的地址空间指针 uargs）。

sys_init_module 函数几乎在一开始就会调用 load_module 函数，后者将在内核空间利用 vmalloc[8] 分配一块大小同样为 len 的地址空间（如图 1-2 中 Elf_Ehdr *hdr 所示。然后通过调用 copy_from_user 函数将用户空间的文件数据复制到内核空间中，从而在内核空间构造出 demodev.ko 的一个 ELF 静态的内存视图，随后模块加载器会利用读入的 ELF 文件头的数据进行格式检查，以防止用户加载不符合既定格式的内核模块。load_module 函数接下来的操作都将以此视图为基础，为使叙述简单起见，我将该视图称为 HDR 视图（图 1-2 下方点画线椭圆部分所覆盖的区域）。HDR 视图所占用的内存空间将在 load_module 结束时通过 vfree 予以释放。

[8] 内核中使用 vmalloc 函数分配内存空间的一个典型案例。

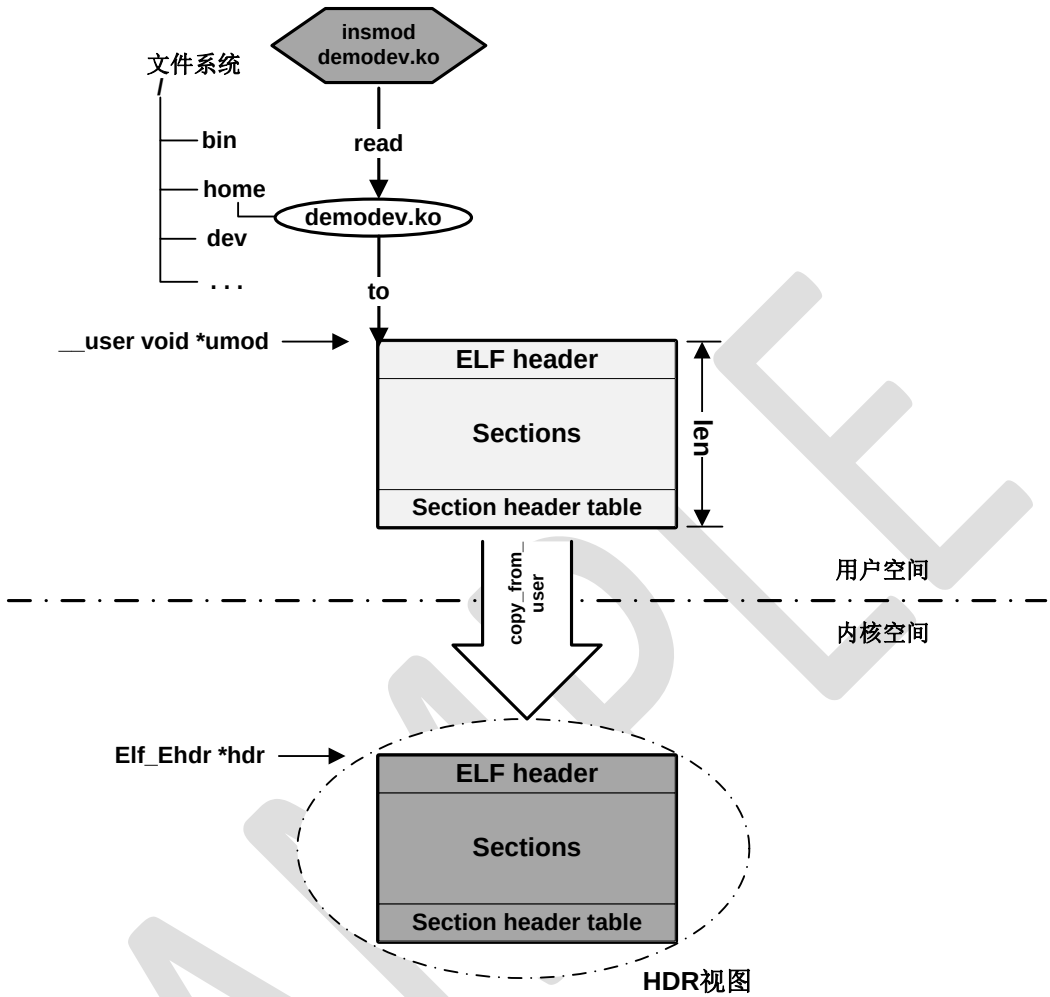

图 1-2 insmod 构造的 ELF 静态内存视图

❍ 字符串表（String Table）

字符串表是 ELF 文件中的一个 section，用来保存 ELF 文件中各个 section 的名称或符号名，这些名称以字符串的形式存在。图 1-3 给出了一个具体的字符串表实例。

由图 1-3 可见，字符串表中各个字符串的构成和 C 语言中的字符串完全一样，都以'\0'作为一个字符串的结束标记。由 index 指向的字符串是从字符串表第 index 个字符开始，直到遇到一个'\0'标记，如果 index 处恰好是'\0'，那么 index 指向的就是个空串（null string）。

在驱动模块所在的 ELF 文件中，一般有两个这样的字符串表 section，一个用来保存各 section 名称的字符串，另一个用来保存符号表中每个符号名称的字符串。虽然同样都是字符串表 section，但是得到这两个 section 的基地址的方法并不一样。

index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

index	string
0	null string
1	name.
7	Variable
11	able
16	able
24	null string

图 1-3 字符串表

存放各 section 名称字符串表的 section 基地址为 char *secstrings = (char *)hdr + entry[hdr->e_shstrndx].sh_offset。而获得符号名称字符串表 section 的基地址则有点绕：首先要遍历 Section header table 中所有的 entry ,去找一个 entry[i].sh_type = SHT_SYMTAB 的 entry，SHT_SYMTAB 表明这个 entry 所对应的 section 是一符号表。这种情况下，entry[i].sh_link 是符号名称字符串表 section 在 Section header table 中的索引值，换句话说，符号名称字符串表所在 section 的基地址为 char *strtab = (char *)hdr + entry[entry[i].sh_link]. sh_offset。

如此，若想获得某一 section 的名称（假设该 section 在 Section header table 中的索引值是 i），那么用 secstrings + entry[i].sh_name 即可。

至此，load_module 函数通过以上计算获得了 section 名称字符串表的基地址 secstrings 和符号名称字符串表的基地址 strtab，内核加载器将这些信息记录在一个类型为 struct load_info 的对象实例中，以留待后续加载过程中使用。

❍ HDR 视图的第一次改写

在获得了 section 名称字符串表的基地址 secstrings 和符号名称字符串表的基地址 strtab 之后，函数开始第一次遍历 Section header table 中的所有 entry，将每个 entry 中的 sh_addr 改写为：entry[i].sh_addr = (size_t)hdr + entry[i].sh_offset，这样 entry[i].sh_addr 将指向该 entry 所对应的 section 在 HDR 视图中的实际存储地址。

在遍历过程中，如果发现 CONFIG_MODULE_UNLOAD 宏没有定义[9]，表明系统不支持动态卸载一个模块，这样，对于名称为“.exit”的 section，将来就没有必要把它加载到内存中，内核代码于是清除对应 entry 中 sh_flags 里面的 SHF_ALLOC 标志位，同时被清除这个标志位的 section 还有

[9] CONFIG_MODULE_UNLOAD 宏用来表明 Linux 内核是否支持 module 的 unload 特性 ,即是否支持使用 rmmod 工具从系统中卸载一个模块。

"__versions"和".modinfo"，SHF_ALLOC 标志位的清除意味着此 entry 对应的 section 将不占有最终的内存空间，本章后续的内容会继续这里的讨论。该过程发生在 rewrite_section_headers()函数的实现中。

相对于刚复制到内核空间的 HDR 视图，HDR 视图的第一次改写只是在自身基础上修改了 Section header table 中的某些 entry 的部分成员，其他方面没有任何变化。接下来在“HDR 视图的第二次改写”一节中将会看到经这里改写后的 HDR 视图会再次被改写，在那里，HDR 视图中的绝大部分 section 会被搬移到一个新的内存空间中，那也是它们最终的内存位置。

- find_sec 函数

内核用 find_sec 来寻找某一 section 在 Section header table 中的索引值，其函数原型为：

static unsigned int find_sec(const struct load_info *info, const char *name);

其中第一个参数的数据类型为

<kernel/module.c>

```
struct load_info {
    Elf_Ehdr *hdr;
    unsigned long len;
    Elf_Shdr *sechdrs;
    char *secstrings, *strtab;
    unsigned long *strmap;
    unsigned long symoffs, stroffs;
    struct _ddebug *debug;
    unsigned int num_debug;
    struct {
        unsigned int sym, str, mod, vers, info, pcpu;
    } index;
};
```

我们在稍早前的讨论中实际上已经接触到了该结构体的用法，它用来记录模块加载过程中所提取到的一些信息，这些信息在内核模块整个加载过程中随时可能会被用到，在后续的讨论中，为简化起见，我们用 ld_info 来指代它的对象实例。find_sec()的第二个参数是要查找的 section 名称，所以这个函数调用的前提是我们已经知道模块文件中某些 section 的名称。

函数如果成功，将返回该 section 在 Section header table 中对应 entry 的索引值，如果没有找到，

则返回 0。函数的具体实现过程非常简单：遍历 Section header table 中所有的 entry（忽略没有 SHF_ALLOC 标志的 section，因为这样的 section 在模块的执行期间并不占优实际内存空间），对每一个 entry，先找到其所对应的 section name，然后和传入的 name 参数进行比较，如果相等，说明找到了对应的 section，返回该 section 在 Section header table 中的索引值。

在对 HDR 视图进行第一次改写之后，内核通过调用 find_sec，分别查找以下名称的 section：，“__versions”、“.modinfo”、“.gnu.linkonce.this_module”和".data..percpu"。查找的索引值分别保存在变量 ld_info->index.vers 、 ld_info->index.info 、 ld_info->index.mod 中和 ld_info->index.pcpu，以备将来使用。

❍ struct module 类型变量 mod 初始化

1.3.2 节中提到了 struct module 是内核用来表示一个模块的非常重要的数据结构。在 load_module 函数中定义有一个 struct module 类型的变量 mod，该变量的初始化是通过模块 ELF 文件中一个名为“.gnu.linkonce.this_module”的 section 来完成的。

ELF 文件中出现的这个 section 其实是模块的编译工具链完成的，并非出自设备驱动程序员的手笔。如果我们仔细看一下编译后的模块所在的目录，就会发现一个扩展名为“.mod.c”的文件，打开该文件，会发现有如下定义：

```
struct module __this_module
    __attribute__((section(".gnu.linkonce.this_module"))) = {
    .name = KBUILD_MODNAME,
    .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
    .exit = cleanup_module,
#endif
    .arch = MODULE_ARCH_INIT,
};
```

其中的__attribute__((section(".gnu.linkonce.this_module")))部分很清楚地揭示了内核模块 ELF 文件中“.gnu.linkonce.this_module”section 出现的根源。

这段定义还有一个比较有趣的地方在于对 struct module 结构体中的 init 和 exit 成员变量的初始化：

```
.init = init_module
.exit = cleanup_module
```

直觉告诉我们，这里的 init 和 exit 应该指向我们的驱动程序源码中实现的模块初始化和退出函数，然而经过和实际的设备驱动程序源码对比，结果也许有点让人失望，在我们驱动程序的源码中定义的初始化和退出函数并不是 init_module 和 cleanup_module。这其实是拜 module_init 和 module_exit

宏所赐，几乎在每一个内核模块源码中都会出现它们的身影[10]，它们利用了 gcc 提供的别名技术（__attribute__(alias)）。

在 MODULE 宏被定义的前提下[11]，也就是针对独立的内核模块而言，module_init 宏定义如下：

```
<include/linux/init.h>
#define module_init(initfn)                    \
    static inline initcall_t __inittest(void)  \
    { return initfn; }                         \
    int init_module(void) __attribute__((alias(#initfn)));
```

该宏定义的核心是最后一句，它将 init_module 函数的别名设定为 initfn，而后者正是我们在设备驱动程序中实现的模块初始化函数，对模块退出函数.exit 的初始化亦同样如此。总之，模块的构造工具链为我们安插了一个“.gnu.linkonce.this_module”section，并初始化了其中的一些成员。在模块加载过程中，load_module 函数将利用这个 section 中的数据来初始化 mod 中的部分成员变量。

模块被加载到内存中之后，内核通过 find_sec 函数查找到“.gnu.linkonce.this_module”section 在 Section header table 中所对应的索引值 modindex，这样通过下面这行简单的代码就得到了“.gnu.linkonce.this_module”section 在内存中的实际地址。

```
mod = (void *)sechdrs[modindex].sh_addr;
```

于是，在第一次改写的 HDR 视图的基础上，mod 指针指向了实际的 struct module 所在的内存地址。接下来我们会看到，在 HDR 视图第二次被改写后，mod 指针将会重新指向“.gnu.linkonce.this_module”section 在内存中的最终地址。

❍ HDR 视图的第二次改写

在这次改写中，HDR 视图中绝大多数的 section 会被搬移到新的内存空间中，之后会根据这些 section 新的内存地址再次改写图 1-2 中的 HDR 视图，使其中 Section header table 中各 entry 的 sh_addr 指向新的也是最终的内存地址。

在为那些需要移动的 section 分配新的内存空间地址之前，内核模块加载器需要决定出 HDR 视图中哪些 section 需要移动，如果移动的话要移动到什么位置。内核代码中 layout_sections 函数用来做这件事，在 layout_sections 函数中，内核会遍历 HDR 视图中的每一个 section，对每一个标记有

10 对内核模块而言，这并非必要，视模块具体需求而定。

11 如果宏 MODULE 没有定义，那么意味着当前的内核模块将和 Linux 内核源码一起编译形成最终的内核映像，这种情况下，module_init 宏退化为一个__initcall，它所包含的函数将在 Linux 系统启动的初始化阶段被调用。

SHF_ALLOC[12]的 section，将其划分到两大类型区域当中：CORE 和 INIT。

为了完成这种分类，layout_sections 函数首先为标记了 SHF_ALLOC 的 section 定义了四种类型：code、read-only data、read-write data 和 small data。任何一个标记了 SHF_ALLOC 的 section 必定属于这四类中的一类。之后，对应每一个分类，函数都会遍历 Section header table 中的所有项 ,将 section name 不是以".init"开始的 section 划归为 CORE 区域 ,并且修改 HDR 视图中 Section header table 中对应 entry 的 sh_entsize，用以记录当前 section 起始地址在 CORE 区域中的偏移量：

entry[i]->sh_entsize = get_offset(mod, &mod->core_size, s, i);

同时用 struct module 对象 mod 中的成员变量 core_size 记录下到当前正在操作的 section 为止的 CORE 区域大小。

mod->core_size += entry[i].sh_size;

对于 CORE 区域中的 code section 和 RO(Read-Only) section 的大小，内核分别用 struct module 结构中的成员变量 core_text_size 和 core_ro_size 来记录。

对于 INIT 区域的分类，和 CORE 区域的划分基本一样，不同的地方在于属于 INIT 区域的 section，其 name 必须以".init"开始，内核用 struct module 结构中的成员变量 init_size 来记录当前 INIT 区域的大小。

mod->init_size += entry[i].sh_size;

对于 INIT 区域中的 code section 和 RO(Read-Only) section 的大小，内核分别用 struct module 结构中的成员变量 init_text_size 和 init_ro_size 来记录。

layout_sections()函数只是对模块中的 section 进行分类和取舍等工作，并没有真正把分类过的 section 进行搬移。

在对 section 进行搬移之前，接下来会有个对符号表的处理，内核代码中通过调用 layout_symtab 函数来完成。Linux 的内核源码中根据是否启用了内核配置选项 CONFIG_KALLSYMS 给出了 layout_symtab 函数的两种不同的定义。

如果没有启用 CONFIG_KALLSYMS，那么 layout_symtab 函数就是个空函数，不做任何事情。CONFIG_KALLSYMS 是一个决定内核映像中是否保留所有符号的配置选项 ,在内核配置文件 Kconfig 中，可以看到如下说明：

Say Y here to let the kernel print out symbolic crash information and symbolic stack backtraces.

12 SHF_ALLOC 标记表示该 section 在模块运行过程中，需要占用内存空间。根据 ELF spec(Portable Formats Specification, Version 1.1):SHF_ALLOC—The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file, this attribute is off for those sections.

This increases the size of the kernel somewhat, as all symbols have to be loaded into the kernel image.

简言之，这是个为了方便系统调试而增加的选项，启用它的代价就是导致最终内核映像文件变大（当然占用的系统内存也会相应增加）。

在启用了 CONFIG_KALLSYMS 选项的 Linux 源码树基础上编译内核模块，会导致内核模块也会将其中的符号保留在内存中[13]。由于在内核模块的 ELF 文件中，符号表所在的 section 没有 SHF_ALLOC 标志，所以上面提到的 layout_sections 函数既不会把符号表 section 划归到 CORE 区域中，也不会把符号表划分到 INIT 区域中，这也是为什么内核要通过另外一个函数 layout_symtab 来对符号表进行分类这种特殊处理。

在对内核模块 ELF 文件中的 section 进行了 CORE 和 INIT 区域的划分之后，内核调用 move_module 函数为 CORE 和 INIT 区域分配对应的内存空间并做真正的搬移动作。move_module 的实现相对比较简单，它通过 vmalloc_exec 来做内存分配的工作，分配后的基地址分别记录在 mod->module_core 和 mod->module_init 中，然后把对应的 section 数据搬移到其在 CORE 和 INIT 区域所在内存空间的最终位置上。显然，在把各 section 搬移到其新的内存地址之后，内核需要改写 HDR 视图中的 Section header table 中对应 entry 的 sh_addr，以使其指向新的地址。

注意，由于此时“.gnu.linkonce.this_module”section 是一个带有 SHF_ALLOC 标志的可写数据 section，也会被搬移到 CORE 区域所在内存空间中，所以必须更新 mod 变量使之指向新的内存地址：

```
mod = (void *)info->sechdrs[info->index.mod].sh_addr;
```

这里之所以要对 HDR 视图中的某些 section 做这样的搬移，是因为在模块加载过程结束时，系统会释放掉 HDR 视图所在的内存区域，不仅如此，在模块初始化工作完成后，INIT 区域所在的内存空间也会被释放掉。由此可见，当一个模块被成功加载进系统，初始化工作完成之后，最终留下的仅仅是 CORE 区域中的内容，因此划分到 CORE 区域中的数据应是模块在系统中整个生存期间会使用到的数据。

如此处理之后，我们在图 1-2 的基础上得到了图 1-4：

[13] layout_symtab 函数不会把内核模块中的所有符号都保留，它只保留那些所谓的“core symbols”，具体代码参见 is_core_symbol 函数。此处的“保留”系指符号最终将被放在 CORE 区域中，它在模块的整个生命周期中一直存在。

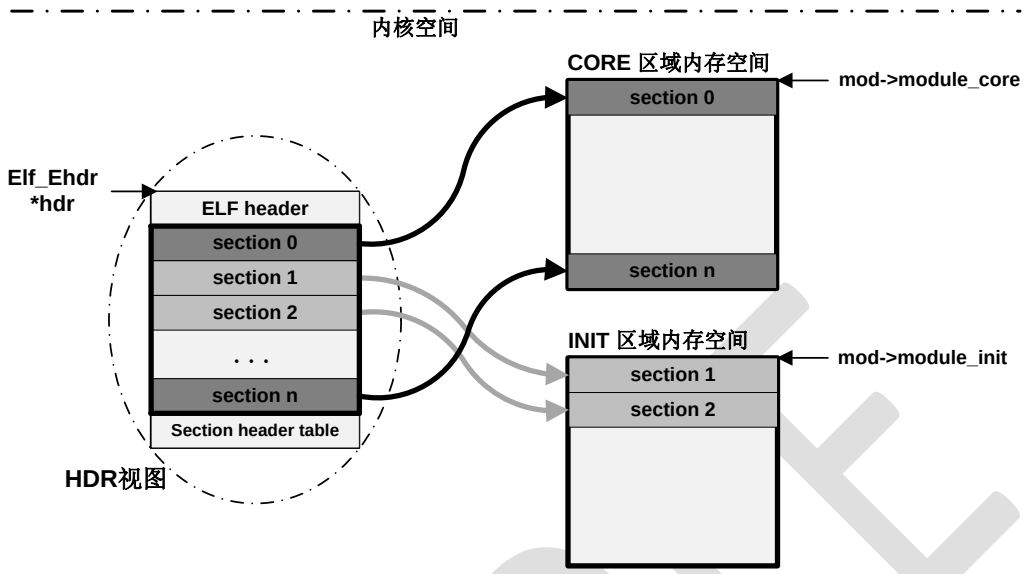

图 1-4 模块加载时的 section 搬移

❍ 模块导出的符号

我们知道模块不仅可以使用内核或者其他模块导出的符号，而且可以向外部导出自己的符号，模块导出符号使用的宏和内核导出符号所使用的完全一样：EXPORT_SYMBOL、EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_FUTURE。由 1.2 节对这些宏定义代码分析可知，内核模块会把导出的符号分别放到“__ksymtab”、“__ksymtab_gpl”和“__ksymtab_gpl_future”section 中。

如果一个内核模块向外界导出了自己的符号，那么将由模块的编译工具链根据模块的链接脚本来生成这些导出符号的 section，而且这些 section 都带有 SHF_ALLOC 标志，所以在模块加载过程中会被搬移到 CORE 区域中。如果模块没有向外界导出任何符号，那么在模块的 ELF 文件中，自然也不会生成这些 section。

显然，内核需要对模块导出的符号进行管理，以便在处理其他模块中那些“未解决的引用”符号时能够找到这些符号[14]。内核对模块导出的符号的管理使用到了 struct module 结构中如下的成员变量：

```
<include/linux/module.h>
struct module
{
    ...
    /* Exported symbols */
```

[14] 在本章第 1.2 节，我们曾提到模块的链接脚本与内核的链接脚本在导出符号上的区别：模块的链接脚本没有定义__start___ksymtab 这样的变量，所以接下来我们将会看到内核如何管理模块导出的符号。

```
        const struct kernel_symbol *syms;
        const unsigned long *crcs;
        unsigned int num_syms;
        …
        /* GPL-only exported symbols. */
        unsigned int num_gpl_syms;
        const struct kernel_symbol *gpl_syms;
        const unsigned long *gpl_crcs;
        …
        /* symbols that will be GPL-only in the near future. */
        const struct kernel_symbol *gpl_future_syms;
        const unsigned long *gpl_future_crcs;
        unsigned int num_gpl_future_syms;
        …
}
```

在把 HDR 视图中的 section 搬移到最终的 CORE 和 INIT 区域之后，内核通过对 HDR 视图中 Section header table 的查找，获得“__ksymtab”、“__ksymtab_gpl”和“__ksymtab_gpl_future” section 在 CORE 区域中的地址，将其记录在 mod->syms、mod->gpl_syms 和 mod->gpl_future_syms 中，该任务由 find_module_sections 函数来完成，其相关代码片段如下：

```
<kernel/module.c>
static void find_module_sections(struct module *mod, struct load_info *info)
{
        …
        mod->syms = section_objs(info, "__ksymtab",
                                 sizeof(*mod->syms), &mod->num_syms);
        …
        mod->gpl_syms = section_objs(info, "__ksymtab_gpl",
                                     sizeof(*mod->gpl_syms),
                                     &mod->num_gpl_syms);
```

```
    ...
    mod->gpl_future_syms = section_objs(info,
                                        "__ksymtab_gpl_future",
                                        sizeof(*mod->gpl_future_syms),
                                        &mod->num_gpl_future_syms);
    ...
}
```

section_objs()函数在其内部调用 find_sec 来查找相应的 section，并通过最后一个参数返回该符号表中符号的个数。

如此，内核通过这些变量将可得到一个模块导出符号的所有信息，如图 1-5 所示。读者将在接下来的“find_symbol 函数”部分中看到这些变量的具体用途。

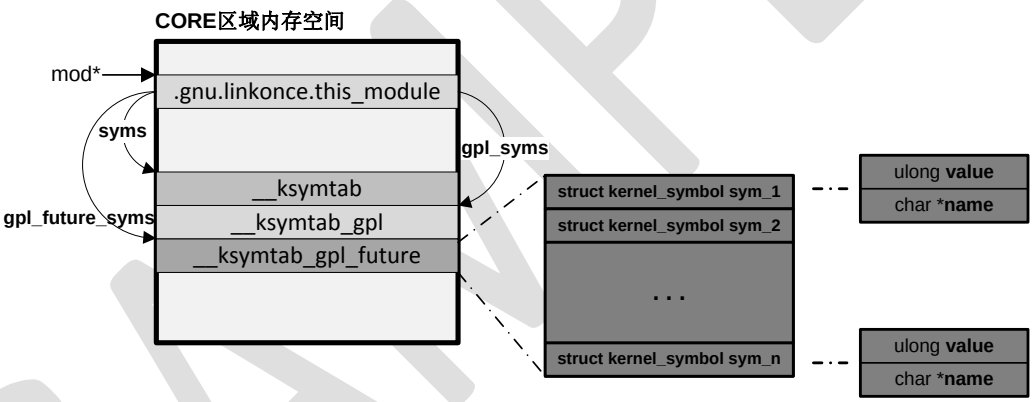

图 1-5 内核模块导出的符号

❍ find_symbol 函数

在模块加载过程中，find_symbol 是个非常重要的函数，顾名思义，它用来查找一个符号。当内核模块加载器发现模块中有一个“未解决的引用”符号的时候，这个函数就有机会被使用到。该函数的原型如下：

```
const struct kernel_symbol *find_symbol(const char *name,
                                        struct module **owner,
                                        const unsigned long **crc,
                                        bool gplok,
                                        bool warn);
```

其中，第一个参数作为函数输入，表示要查找的符号名；第二个参数用以表明符号可能隶属的模块，作为返回值；第三个参数表示符号的 crc 校验码，作为返回值。第四个参数作为函数输入，表示要查找的符号是否遵循 GPL 协议。

在深入到这个函数内部之前，有必要先介绍几个数据结构，这几个数据结构将在 find_symbol 函数的调用链中被用到：

<include/linux/module.h>

```
struct symsearch {
    const struct kernel_symbol *start, *stop;
    const unsigned long *crcs;
    enum {
        NOT_GPL_ONLY,
        GPL_ONLY,
        WILL_BE_GPL_ONLY,
    } licence;
    bool unused;
};
```

struct symsearch 用来对应要查找的每一个符号表 section，换句话说，对要查找的每个符号表 section，内核代码都要为之产生一个 struct symsearch 类型的实例。结构体中的成员变量 start 和 stop 分别指向对应 section 的开始和结束地址，bool 型的 unused 成员用来表示内核是否配置了 CONFIG_UNUSED_SYMBOLS 选项，不过这个选项是“非主流”的，长远看这个选项最终可能会消失，因此本书只在这里提一下，在后续的章节中将忽略所有该选项被启用时才起作用的代码。另一个比较重要的成员是 enum 型的 licence，GPL_ONLY 表示符号只提供给满足 GPL 协议的模块使用，NOT_GPL_ONLY 表示不一定要只给满足 GPL 协议的模块使用，WILL_BE_GPL_ONLY 表示将来只提供给满足 GPL 协议的模块使用。再提醒一下，NOT_GPL_ONLY 符号由 EXPORT_SYMBOL 负责导出，GPL_ONLY 符号由 EXPORT_SYMBOL_GPL 负责导出，WILL_BE_GPL_ONLY 符号由 EXPORT_SYMBOL_GPL_FUTURE 负责导出。

<kernel/module.c>

```
struct find_symbol_arg {
    /* Input */
    const char *name;
```

```
    bool gplok;
    bool warn;

    /* Output */
    struct module *owner;
    const unsigned long *crc;
    const struct kernel_symbol *sym;
};
```

find_symbol_arg 用做查找符号的标识参数，可以看到其大部分数据成员与 find_symbol 函数原型中的参数完全一致，在 find_symbol 函数调用链中，一个该类型的变量 fsa 用来在调用链中传递信息。

以下仔细分析 find_symbol 的功能，其源代码如下：

<kernel/module.c>

```
const struct kernel_symbol *find_symbol(const char *name,
                                        struct module **owner,
                                        const unsigned long **crc,
                                        bool gplok,
                                        bool warn)
{
    struct find_symbol_arg fsa;

    fsa.name = name;
    fsa.gplok = gplok;
    fsa.warn = warn;

    if (each_symbol_section(find_symbol_in_section, &fsa)) {
        if (owner)
            *owner = fsa.owner;
        if (crc)
            *crc = fsa.crc;
        return fsa.sym;
```

```
    }
    return NULL;
}
```

函数的核心是调用 each_symbol_section()函数进，后者是用来进行符号查找的主要函数，为节约篇幅起见，这里不再摘录其源代码，而是直接讲述其主要功能框架。

总体上，each_symbol_section 函数可以分成两个部分：第一部分是在内核导出的符号表中查找对应的符号，如果找到，就通过 fsa 返回该符号的信息，否则，再进行第二部分的查找；第二部分是在系统中已加载的模块（系统中所有已成功加载的模块都以链表的形式保存在一个全局变量 modules 中）的导出符号表中查找对应的符号，如果找到就通过 fsa 返回该符号的信息，否则函数返回 false。图 1-6 展示了 find_symbol 在查找一个符号时可能的搜索路径：

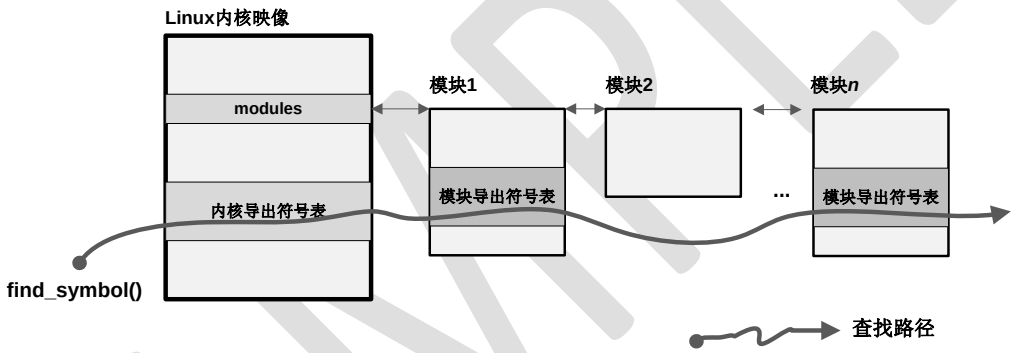

图 1-6 find_symbol 查找符号时的搜索路径

第一部分在对内核符号表进行查找时，首先构造一个 struct symsearch 类型的数组 arr。

```
static const struct symsearch    arr[] = {
        { __start___ksymtab, __stop___ksymtab, __start___kcrctab,
          NOT_GPL_ONLY, false },
        { __start___ksymtab_gpl, __stop___ksymtab_gpl,
          __start___kcrctab_gpl,
          GPL_ONLY, false },
        { __start___ksymtab_gpl_future, __stop___ksymtab_gpl_future,
          __start___kcrctab_gpl_future,
          WILL_BE_GPL_ONLY, false },
```

```
};
```

注意这里的__start___ksymtab、__start___kcrctab 和__stop___ksymtab 等变量已经在前面的“EXPORT_SYMBOL 的内核实现”一节中交代过，它们在内核的链接脚本中定义由链接器负责产生，由内核源码负责声明，现在到了使用它们的时候了。

函数首先查询内核的导出符号表，通过调用函数 each_symbol_in_section，后者的核心代码段如下（经过适当改写）：

<kernel/module.c>

```
static bool each_symbol_in_section(const struct symsearch *arr, struct module *owner,
                                   void *data)
{
    unsigned int i, j;

    for (j = 0; j < ARRAY_SIZE(arr); j++) {
        for (i = 0; i < arr[j].stop - arr[j].start; i++)
            if (find_symbol_in_section(&arr[j], owner, i, data))
                return true;
    }

    return false;
}
```

为了在内核的导出符号表中查找某一指定的符号名，each_symbol_in_section 函数使用了两层 for 循环：外层 j 引导的 for 循环用来遍历符号可能所在的内核导出符号表中的各 section；内层 i 引导的 for 循环用来遍历外层 for 循环所指定的 section 中的每个 struct kernel_symbol 类型的元素。对于每个 kernel_symbol 对象，都会调用 find_symbol_in_section 函数进行查找。

为了清楚地理解内核加载模块时如何处理“未解决的引用”符号，有必要仔细分析一下 find_symbol_in_section 函数的主要功能。因为对 Linux 下的设备驱动程序员而言，几乎每天都在和这个功能打交道，清楚地理解其内核机制，将来一旦在加载模块时出现相关问题，也可以将其快速定位并最终解决。另外，对于带有“_GPL”后缀的符号名，在写驱动程序的内核模块时常常会遇到，然而其背后到底蕴涵着怎样的设计理念呢？通过分析 find_symbol_in_section 函数，也可以得到所需的答案。

find_symbol_in_section 函数的完整源代码如下：

<kernel/module.c>

```
static bool find_symbol_in_section(const struct symsearch *syms,
                                   struct module *owner,
                                   void *data)
{
    struct find_symbol_arg *fsa = data;
    struct kernel_symbol *sym;

    sym = bsearch(fsa->name, syms->start, syms->stop - syms->start,
                  sizeof(struct kernel_symbol), cmp_name);

    if (sym != NULL && check_symbol(syms, owner, sym - syms->start, data))
        return true;

    return false;
}
```

函数首先调用 bsearch()函数在当前的符号表 section 中进行二分查找，因为符号表所在的 section 实际上就是由 struct kernel_symbol 类型的对象所构成的数组，所以这个函数应当很好理解，如果找到了指定的符号，bsearch()函数返回那个符号的对象 sym，否则返回 NULL。假设函数在当前 section 中成功找到了指定的符号名，那么也不代表这个导出的符号一定可以被模块所使用。所以内核紧接着调用 check_symbol()函数对找到的符号进行检查，以确定是否可被当前正被加载的模块所使用。check_symbol()函数的实现如下：

<kernel/module.c>

```
static bool check_symbol(const struct symsearch *syms,
                         struct module *owner,
                         unsigned int symnum, void *data)
{
    struct find_symbol_arg *fsa = data;
    if (!fsa->gplok) {
        if (syms->licence == GPL_ONLY)
```

```
                return false;
            if (syms->licence == WILL_BE_GPL_ONLY && fsa->warn) {
                printk(KERN_WARNING "Symbol %s is being used "
                        "by a non-GPL module, which will not "
                        "be allowed in the future\n", fsa->name);
                printk(KERN_WARNING "Please see the file "
                        "Documentation/feature-removal-schedule.txt "
                        "in the kernel source tree for more details.\n");
            }
        }
        fsa->owner = owner;
        fsa->crc = symversion(syms->crcs, symnum);
        fsa->sym = &syms->start[symnum];
        return true;
}
```

我们现在讨论的上下文依然是在内核导出的符号表中进行查找，函数中 syms->licence 的值是在 each_symbol_section 函数中通过 arr 数组静态构造的，其值可为 NOT_GPL_ONLY、GPL_ONLY 和 WILL_BE_GPL_ONLY，具体的值由当初符号被导出时使用的 EXPORT_*宏决定。

而 fsa->gplok 和 fsa->warn 的设定最早是在 find_symbol 函数中 ,是通过后者的函数参数传入的。fsa->warn 主要用来控制警告信息的输出。fsa->gplok 用来表示当前正加载的模块是不是满足 GPL 协议（GPL module 或 non-GPL module），fsa->gplok = true 表明这是个 GPL module，否则就是 non-GPL module。内核判断一个模块是否 GPL 兼容，要使用到本章后面的“模块的信息”部分中的内容。

对于一个 non-GPL module 而言，它不能使用内核导出的属于 GPL_ONLY 的那些符号，所以即便该模块中一个“未解决的引用”符号名匹配上内核导出的一个属于 GPL_ONLY 的符号，也不能认为查找成功。但是如果要查找的符号匹配上一个属于 WILL_BE_GPL_ONLY 的符号，因为这个导出的符号“将来要成为 GPL_ONLY”,所以即使现在还不是 GPL_ONLY ,查找姑且算是成功的 ,不过即便如此，因为内核对模块将来能否成功使用该符号没有给出明确的承诺，所以此时它的处理方式是：如果 fsa->warn 为真，就通过 printk 给出一个警告信息。对于一个 GPL module 而言，一切好说，可以使用内核导出的所有符号。

函数如果成功查找到符号 ,利用传进来的 data 指针(其实就是一个 struct find_symbol_arg 类型的指针变量 fsa)将符号相关信息传给上层调用的函数，对于内核导出的符号而言，它不属于某一特定的模块，所以这种情形下 owner 的值为 NULL。函数的最后有一个对 symversion()函数的调用，在启动了模块版本控制的情形下(CONFIG_MODVERSIONS=y)，该函数会返回当前符号在 crc section 中对应的 crc 所在的地址。

至此，find_symbol 的第一部分，即在内核导出的符号表中查找指定的符号已经结束。如果指定的符号没能在内核导出的符号表中完成解析，那么将进入 find_symbol 函数的第二部分，否则符号查找的工作将成功结束。

下面开始介绍 find_symbol 的第二部分，也即在系统已经加载的模块所导出的符号表中查找符号。内核为达成此目的，需要在加载一个内核模块时完成下面两件事：

第一 ,模块成功加载进系统之后 ,需要将表示该模块的 struct module 类型变量 mod 加入到 modules 中，后者是内核维护的一个全局的链表变量，用来记录系统中所有已加载的模块。

<kernel/module.c>

```
static LIST_HEAD(modules)；
list_add_rcu(&mod->list, &modules)；
```

第二 ,模块导出的符号信息记录在 mod 的相关成员变量中 ,这个过程的详细描述参见本章前面的“模块导出的符号”部分。

在系统所有已加载的模块导出的符号中查找某一指定符号，其核心代码片段如下：

<kernel/module.c>

```
bool each_symbol_section(bool (*fn)(const struct symsearch *arr,
                         struct module *owner,
                         void *data),
                 void *data)
{
    ...
    list_for_each_entry_rcu(mod, &modules, list) {
        struct symsearch arr[] = {
            { mod->syms, mod->syms + mod->num_syms, mod->crcs,
              NOT_GPL_ONLY, false },
            { mod->gpl_syms, mod->gpl_syms + mod->num_gpl_syms,
              mod->gpl_crcs,
```

```
                GPL_ONLY, false },
            { mod->gpl_future_syms,
                mod->gpl_future_syms + mod->num_gpl_future_syms,
                mod->gpl_future_crcs,
                WILL_BE_GPL_ONLY, false },
        };

        if (each_symbol_in_section(arr, ARRAY_SIZE(arr), mod, fn, data))
            return true;
}
```

相对于 find_symbol 的第一部分（在内核导出的符号表中查找某一符号），第二部分唯一的区别在于构造的 arr 数组，它使用了模块加载过程中动态生成的 mod->syms、mod->gpl_syms 和 mod->gpl_future_syms 等变量，这种差异体现在前文已经提及的模块与内核链接脚本的不同。函数在全局链表 modules 中遍历所有已加载的内核模块，对其中的每一模块都构造一个新的 arr 数组，然后在其中查找特定的符号。

❍ 对“未解决的引用”符号（unresolved symbol）的处理

前文中已多次提到内核模块 ELF 文件中的“未解决的引用”符号，所谓的“未解决的引用”符号，就是模块的编译工具链在对模块进行链接生成最终的.ko 文件时，对于模块中调用的一些函数，最简单的比如 printk 函数，链接工具无法在该模块的所有目标文件中找到这个函数的具体实现（因为这个函数是在 Linux 的内核源代码中实现的，其指令码存在于编译内核生成的目标文件中，模块的链接工具显然不会也不应该去查找内核的目标文件），所以就会将这个符号标记为“未解决的引用”，对它的处理将一直延续到内核模块被加载时（处理的核心是在内核或者是其他内核模块导出的符号中找到这个“未解决的引用”符号[15]，继而找到该符号所在的内存地址，从而最终形成正确的函数调用）。

Linux 内核中，一个名为 simplify_symbols 的函数用来实现这一功能，这是个很有意思的函数，我们不妨仔细看一下它的代码。

<kernel/module.c>

```
/* Change all symbols so that st_value encodes the pointer directly. */
static int simplify_symbols(struct module *mod, const struct load_info *info)
```

[15] 读者可在 Linux 环境下通过 nm 命令来查看一个模块中出现的所有“未解决的引用”符号，比如：

dennis@AMDLinuxFGL:~$ nm demodev.ko

该命令的输出中，所有前面带一“U”标志的符号均为“未解决的引用”符号。

```
{
    Elf_Shdr *symsec = &info->sechdrs[info->index.sym];
    Elf_Sym *sym = (void *)symsec->sh_addr;
    unsigned long secbase;
    unsigned int i;
    int ret = 0;
    const struct kernel_symbol *ksym;

    for (i = 1; i < symsec->sh_size / sizeof(Elf_Sym); i++) {
        const char *name = info->strtab + sym[i].st_name;

        switch (sym[i].st_shndx) {
        case SHN_COMMON:
            /* We compiled with -fno-common.  These are not
               supposed to happen.  */
            DEBUGP("Common symbol: %s\n", name);
            printk("%s: please compile with -fno-common\n",
                   mod->name);
            ret = -ENOEXEC;
            break;

        case SHN_ABS:
            /* Don't need to do anything */
            DEBUGP("Absolute symbol: 0x%08lx\n",
                   (long)sym[i].st_value);
            break;

        case SHN_UNDEF:
```

```
                ksym = resolve_symbol_wait(mod, info, name);
                /* Ok if resolved.  */
                if (ksym && !IS_ERR(ksym)) {
                        sym[i].st_value = ksym->value;
                        break;
                }

                /* Ok if weak.  */
                if (!ksym && ELF_ST_BIND(sym[i].st_info) == STB_WEAK)
                        break;

                printk(KERN_WARNING "%s: Unknown symbol %s (err %li)\n",
                       mod->name, name, PTR_ERR(ksym));
                ret = PTR_ERR(ksym) ?: -ENOENT;
                break;

        default:
                /* Divert to percpu allocation if a percpu var. */
                if (sym[i].st_shndx == info->index.pcpu)
                        secbase = (unsigned long)mod_percpu(mod);
                else
                        secbase = info->sechdrs[sym[i].st_shndx].sh_addr;
                sym[i].st_value += secbase;
                break;
        }
    }

    return ret;
```

```
}
```

简言之，在加载模块的过程中，simplify_symbols 函数用来为当前正在加载的模块中所有“未解决的引用”符号产生正确的目标地址。对这段代码的透彻理解需要读者具备 ELF 文件格式规范的相关概念，我们不可能在本书中全面介绍 ELF 文件格式，但是为了让读者能理解上面的代码，还是从代码的角度出发，将其中所涉及的一些有关 ELF 文件的概念予以简单介绍。

代码中的 Elf_Sym 定义的是符号表中的元素，具体定义如下：

```
<include/linux/elf.h>
typedef struct elf64_sym {
    Elf64_Word      st_name;    /* Symbol name, index in string tbl */
    unsigned char   st_info;    /* Type and binding attributes */
    unsigned char   st_other;   /* No defined meaning, 0 */
    Elf64_Half      st_shndx;   /* Associated section index */
    Elf64_Addr      st_value;   /* Value of the symbol */
    Elf64_Xword     st_size;    /* Associated symbol size */
} Elf64_Sym;

#ifdef CONFIG_64BIT
#define Elf_Sym     Elf64_Sym
#else
#define Elf_Sym     Elf32_Sym
#endif
```

其中，st_name 是符号名在符号名称字符串表中的索引值，详见本章前面的“字符串表（String Table）”部分。st_value 是符号所在的内存地址。simplify_symbols 函数的主要功能就是在加载模块时重新为符号生成正确的 st_value 值。st_shndx 是该符号所在的 section 在 Section header table 中的索引值，但是该值还有一些特殊的定义，详见 ELF 文件格式文档。对于符号表，它是 ELF 文件中的一个 section，这个 section 就是由一系列 struct Elf_Sym 型元素所构成的一个数组，每个元素代码一个符号。

在对符号表的概念有了基本了解之后，回过头来看看 simplify_symbols 函数的代码实现。函数首先通过一个 for 循环遍历符号表中的所有符号，对于每一个符号都会根据该符号的 st_shndx 值分情况进行处理。前面刚刚提到 st_shndx，通常情况下，该值表示符号所在 section 的索引值，为方便叙

述，我们称这种符号为一般符号。对于一般符号来说，它的 st_value 在 ELF 文件中的值是从其所在 section 起始处算起的一个偏移量，代码中在 switch 的 default 分支下进行处理：先得到符号所在 section 的最终内存地址，然后加上它在 section 中的偏移量，这样就得到了符号的最终内存地址。

除了一般符号，还有些符号的 st_shndx 具有特殊的含义，典型的如 SHN_ABS 和 SHN_UNDEF，前者表明该符号具有绝对地址，因此 simplify_symbols 函数无须对这种情况予以任何处理，后者表明该符号是一“undefined symbol”，其实就是我们一直说的“未解决的引用”符号。这种情况下 simplify_symbols 函数会调用 resolve_symbol 函数来处理该未定义符号，后者会调用 find_symbol 函数去查找该符号（详细的查找过程见本章前面的“find_symbol 函数”部分），如果找到了，就把它在内存中的实际地址赋值给 st_value。

如此，经过 simplify_symbols 函数的调用之后，如不出意外，内核模块符号表中的所有符号就都有了正确的 st_value 值，也即都有了正确的内存地址。

到目前为止，一切关于符号相关的处理貌似都很完美，然而情况真是如此吗？如果当前正在加载的模块中一个“未解决的引用”符号是由系统中别的已加载的内核模块导出的，情况会怎样呢？如果读者的探索精神足够强烈，想想那些由内核模块导出的符号吧[16]，由前面的内容，虽然“__ksymtab”、“__ksymtab_gpl”和“__ksymtab_gpl_future” section 都被搬移到了最终的内存地址处，而且这些地址也被表示模块的 mod 变量记录在案，但是这些 section 中的内容呢？

回头看看图 1-5“内核模块导出的符号”，每个“__ksymtab”、“__ksymtab_gpl”和“__ksymtab_gpl_future” section 都是由 struct kernel_symbol 类型的元素所构成的数组。到目前为止，如果仔细考察每个元素的话，会发现其中的 value 成员依然是内核模块在静态编译时产生的地址。换句话说，根本不是这些符号在模块被加载进系统之后在内存中的实际地址。这显然不是我们想要的效果：想想本节前半部分提到的对模块中“未解决的引用”符号的处理，如果在别的模块中找到的符号其内存地址只是当初该模块在静态链接时写入的地址，那么对该符号的引用必然导致错误的内存访问，这是个很严重的问题。而 Linux 内核对这一问题的处理便引出了本章接下来的一个话题——重定位。

- 重定位（relocation）

重定位主要用来解决静态链接时的符号引用与动态加载时实际符号地址不一致的问题，上节结束部分提到的模块导出的符号地址，就是一个典型的需要重定位的例子。仔细讨论重定位的内容并不是件轻松的事情，因为重定位的任务包含很多方面的内容，尤其是跟体系架构相关的一些微妙晦涩的技术细节。考虑到本书的主题定位，也许在这里详细讲述重定位的技术细节并不是件很有价值的事情：篇幅把握得不够理想，很可能就冲淡了本章关于内核模块加载这一主线。消耗大量的篇幅和读者大量的时间，所涉及的主题在现实中却难有用武之地。但是重定位毕竟是内核加载过程中一个很重要的步骤，仔细权衡之下，笔者决定采取一个相对折中的方案，在讨论重定位时就事论事。本节就以上节末尾提

16 为什么只关注那些由模块导出的符号呢？因为对于内核导出的符号而言，所有符号的实际链接地址都会被解决，除非内核被设计成可重定位的。而内核模块则不同，它本身就是可重定位的。

出的问题来展开重定位的话题。

如果模块有用 EXPORT_SYMBOL 导出的符号，那么模块的编译工具链会为这个模块的 ELF 文件生成一个独立的特殊 section：“.rela__ksymtab”，它专门用于对“__ksymtab”section 的重定位，称为 relocation section。这个 section 是由下面的数据结构元素形成的一个数组。

<include/linux/elf.h>

```
typedef struct elf64_rela {
  Elf64_Addr      r_offset;     /* Location at which to apply the action */
  Elf64_Xword     r_info;       /* index and type of relocation */
  Elf64_Sxword    r_addend;     /* Constant addend used to compute value */
} Elf64_Rela;
```

先来看看 Linux 源码中用于模块加载时重定位的代码:

<kernel/module.c>

```
static int apply_relocations(struct module *mod, const struct load_info *info)
{
    unsigned int i;
    int err = 0;

    /* Now do relocations. */
    for (i = 1; i < info->hdr->e_shnum; i++) {
        unsigned int infosec = info->sechdrs[i].sh_info;
        /* Not a valid relocation section? */
        if (infosec >= info->hdr->e_shnum)
            continue;
        /* Don't bother with non-allocated sections */
        if (!(info->sechdrs[infosec].sh_flags & SHF_ALLOC))
            continue;
        if (info->sechdrs[i].sh_type == SHT_REL)
            err = apply_relocate(info->sechdrs, info->strtab,
                                 info->index.sym, i, mod);
```

```
        else if (info->sechdrs[i].sh_type == SHT_RELA)
            err = apply_relocate_add(info->sechdrs, info->strtab,
                                     info->index.sym, i, mod);
        if (err < 0)
            break;
    }
    return err;
}
```

代码用一个 for 循环来遍历 HDR 视图中 Section header table 中所有的 entry。对于一个重定位的 section，其 entry 中的 sh_type 的值或为 SHT_REL 或为 SHT_RELA，分别对应两种不同的重定位方式，我们以第二种类型 SHT_RELA(x86-64 使用的重定位方式)为例：在遍历的过程中，如果发现了一个 sh_type = SHT_RELA 的 section，系统就调用 apply_relocate_add()函数来执行重定位，后者是个体系结构相关的函数，在 x86-64 架构上的具体实现为：

<arch/x86/kernel/module.c>

```
int apply_relocate_add(Elf64_Shdr *sechdrs,
                       const char *strtab,
                       unsigned int symindex,
                       unsigned int relsec,
                       struct module *me)
{
    unsigned int i;
    Elf64_Rela *rel = (void *)sechdrs[relsec].sh_addr;
    Elf64_Sym *sym;
    void *loc;
    u64 val;

    for (i = 0; i < sechdrs[relsec].sh_size / sizeof(*rel); i++) {
        /* This is where to make the change */
        loc = (void *)sechdrs[sechdrs[relsec].sh_info].sh_addr
            + rel[i].r_offset;
```

```
/* This is the symbol it is referring to.  Note that all
   undefined symbols have been resolved.  */
sym = (Elf64_Sym *)sechdrs[symindex].sh_addr
      + ELF64_R_SYM(rel[i].r_info);

val = sym->st_value + rel[i].r_addend;
switch (ELF64_R_TYPE(rel[i].r_info)) {
case R_X86_64_NONE:
      break;
case R_X86_64_64:
      *(u64 *)loc = val;
      break;
case R_X86_64_32:
      *(u32 *)loc = val;
      if (val != *(u32 *)loc)
            goto overflow;
      break;
case R_X86_64_32S:
      *(s32 *)loc = val;
      if ((s64)val != *(s32 *)loc)
            goto overflow;
      break;
case R_X86_64_PC32:
      val -= (u64)loc;
      *(u32 *)loc = val;
      break;
default:
```

```
                printk(KERN_ERR "module %s: Unknown rela relocation: %llu\n",
                        me->name, ELF64_R_TYPE(rel[i].r_info));
                return -ENOEXEC;
            }
        }
        return 0;

overflow:
        printk(KERN_ERR "overflow in relocation type %d val %Lx\n",
                (int)ELF64_R_TYPE(rel[i].r_info), val);
        printk(KERN_ERR "`%s' likely not compiled with -mcmodel=kernel\n",
                me->name);
        return -ENOEXEC;
}
```

函数的核心是用一个 for 循环来遍历当前 relocation section 中所有数组元素(类型为 Elf64_Rela)，每一个元素 rel[i]对应一次重定位的过程 --

首先根据 rel[i]的 r_offset 成员获得需要修改地址值的指针 loc：

```
void    *loc = (void *)sechdrs[sechdrs[relsec].sh_info].sh_addr+ rel[i].r_offset;
```

relocation section header 中的 sh_info 是需要重定位的 section 在 Section header table 中的索引值。

然后找到需要重定位的符号在符号表中的地址 sym：

```
Elf64_Sym * sym = (Elf64_Sym *)sechdrs[symindex].sh_addr+ ELF64_R_SYM(rel[i].r_info);
```

接着得到当前重定位符号的最终目标地址：

```
u64   val = sym->st_value + rel[i].r_addend;
```

最后自然是用符号最终的目标地址 val 来修改 loc 指向的数值即可，由后续的 switch 语句来完成，本质上就是:

```
*loc = val;
```

修改模块导出的符号在内存空间中最终的地址值只是“重定位”的一个典型应用，但是“重定位”的内涵并不仅限于此。对于内核模块而言，对“未解决引用”的符号处理和“重定位”是相辅相成的，两者相互补充相互利用。

❍ 模块参数

内核模块在用 insmod 命令加载时，可以通过诸如以下的命令向模块传递一些参数。

insmod demodev.ko dolphin=10 bobcat=5

其中 dolphin=10 和 bobcat=5 就是向模块传递的参数，dolphin 和 bobcat 是参数名，10 和 5 是具体的参数值。当然为了能正确接收外部的参数，内核模块本身在源代码中必须用 module_param 宏声明模块可以接收的参数。在上面的例子中，模块应该使用诸如 module_param(dolphin, int, 0) 来声明一个模块参数，例如下面的代码片段：

<demodev.c>

```
#include <linux/module.h>
#include <linux/kernel.h>

int dolphin;
int bobcat;
module_param(dolphin, int, 0);
module_param(bobcat, int, 0);

static int demodev_init(void)
{
        printk("dolphin=%d,bobcat=%d\n", dolphin, bobcat);
        return 0;
}
static void demodev_exit(void)
{
        printk("+demodev_exit!\n");
}
module_init(demodev_init);
module_exit(demodev_exit);
```

从本章稍后的讨论中可以得知，内核模块加载器对模块参数的构造（初始化）过程发生在对模块初始化函数 demodev_init 的调用之前，所以在 demodev_init 函数被调用时，已经可以得到从命令行传过来的实际参数。

Linux 源码中 module_param 宏相关的完整定义如下：

<include/linux/moduleparam.h>

```
#define __module_param_call(prefix, name, ops, arg, isbool, perm)      \
      /* Default value instead of permissions? */                      \
      static int __param_perm_check_##name __attribute__((unused)) =\
      BUILD_BUG_ON_ZERO((perm) < 0 || (perm) > 0777 || ((perm) & 2))      \
      + BUILD_BUG_ON_ZERO(sizeof(""prefix) > MAX_PARAM_PREFIX_LEN);\
      static const char __param_str_##name[] = prefix #name;              \
      static struct kernel_param    __moduleparam_const    __param_##name \
      __used                                                    \
     __attribute__ ((unused,__section__ ("__param"),aligned(sizeof(void *)))) \
      = { __param_str_##name, ops, perm, isbool ? KPARAM_ISBOOL : 0,      \
            { arg } }

#define module_param_cb(name, ops, arg, perm)                             \
      __module_param_call(MODULE_PARAM_PREFIX,                            \
                  name, ops, arg, __same_type((arg), bool *), perm)

#define __MODULE_PARM_TYPE(name, _type)                                   \
  __MODULE_INFO(parmtype, name##type, #name ":" _type)

#define module_param_named(name, value, type, perm)                       \
      param_check_##type(name, &(value));                                 \
      module_param_cb(name, &param_ops_##type, &value, perm);              \
      __MODULE_PARM_TYPE(name, #type)

#define module_param(name, type, perm)                                    \
      module_param_named(name, name, type, perm)
```

基本上，上述的宏系列会在一个名为“__param”的 section[17]中定义一些变量。这段宏的定义细究起来有点晦涩。这里不妨以本节开始的例子来展开该宏（去除了一些跟调试相关的微末细节），以便使读者对 module_param 宏建立一个具体的印象，module_param(dolphin, int, 0)展开后如下：

```
param_check_int(dolphin, &( dolphin));
static int __param_perm_check_dolphin   __attribute__((unused)) =      \
static const char __param_str_dolphin[] = " dolphin ";          \
static struct kernel_param   __moduleparam_const   __param_ dolphin \
__used                                                  \
__attribute__ ((unused,__section__ ("__param"),aligned(sizeof(void *)))) \
= { __param_str_ dolphin, &param_ops_int, 0, 0,    \
    { &dolphin } }
```

可见 module_param(dolphin, int, 0)在“__param” section 中定义了一个类型为 struct kernel_param 的静态常量。struct kernel_param 的定义如下：

```
<include/linux/moduleparam.h>
struct kernel_param {
    const char *name;
    const struct kernel_param_ops *ops;
    u16 perm;
    u16 flags;
    union {
        void *arg;
        const struct kparam_string *str;
        const struct kparam_array *arr;
    };
};
```

[17] 如同模块导出的符号所在的 section 一样，模块参数所在的 section 也是一个 optional 的 section：如果模块没有用 module_param 相关宏声明任何参数变量，那么最终的 ELF 文件将不会生成对应的 section。

其中 name 为参数名，perm 为对 sysfs 文件系统中模块参数的访问许可，定义在结构体 struct kernel_param_ops 对象 ops 中的成员函数 set 和 get 用来在模块 mod 的 args 成员和模块的参数 section 间拷贝数据，最后的 union 为指向参数的指针。

__used 和 unused 主要用来避免编译器产生警告信息，因为此处声明的__param_dolphin 变量在模块源码的其他部分并不会被使用。

param_check_int 宏用来检测变量 dolphin 在 module_param 宏之前是否定义，因为 struct kernel_param 中的 union 联合体只是用来放置模块使用的参数所在地址，如果之前该参数没有定义，就不可能生成&dolphin 的值。所以我们的内核模块示例源代码 demodev.c 在用 module_param 声明模块参数之前，要首先定义出这些参数。

int dolphin; //首先定义

module_param(dolphin, int, 0); //然后再声明模块参数

如果在设备驱动程序中忘了先定义参数变量 dolphin，只有 module_param(dolphin, int, 0)来声明模块参数，将会得到如下编译错误：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# make
make   -C /lib/modules/3.1.6/build   M=/home/dennis/Linux/book/chap01 modules
make[1]: Entering directory `/home/dennis/Linux/kernel/linux-3.1.6'
  CC [M]   /home/dennis/Linux/book/chap01/demodev.o
/home/dennis/Linux/book/chap01/demodev.c: In function '__check_dolphin':
/home/dennis/Linux/book/chap01/demodev.c:5: error: 'dolphin' undeclared (first use in this function)
/home/dennis/Linux/book/chap01/demodev.c:5: note: each undeclared identifier is reported only once for each function it appears in
/home/dennis/Linux/book/chap01/demodev.c: At top level:
/home/dennis/Linux/book/chap01/demodev.c:5: error: 'dolphin' undeclared here (not in a function)
/home/dennis/Linux/book/chap01/demodev.c:5: confused by earlier errors, bailing out
make[2]: *** [/home/dennis/Linux/book/chap01/demodev.o] Error 1
make[1]: *** [_module_/home/dennis/Linux/book/chap01] Error 2
make[1]: Leaving directory `/home/dennis/Linux/kernel/linux-3.1.6'
make: *** [default] Error 2
```

在上面的宏展开的实例中，可以看到指向参数的指针值被设定为“{ &dolphin }”，在模块静态链接期间，&dolphin 指令不可能生成其最终的运行期地址，因此模块参数所在的“__param” section

需要有一个对应的 relocation section ".rela__param"，用来完成对参数指针的重定位，这样才能把命令行中的参数值正确复制到模块的 "__param" section 中。

除了 module_param 之外，Linux 系统下还有另外两个宏 module_param_array 和 module_param_string，分别用来设定数组型和字符串型参数，本书不再赘述。

下面讨论 "insmod demodev.ko dolphin=10 bobcat=5" 中携带的参数值如何为模块所用，读者大约可以猜想出命令行中的参数值会被复制到模块的参数中，这样模块在开始使用参数 dolphin 之前，其值已经被 insmod 命令行中的实际值所改写。图 1-7 展示了命令行参数传递到模块的 "__param" section 的全过程。

在实际的内核源代码中，sys_init_module 函数的最后一个参数 const char __user *uargs 清楚地表明这是由用户空间传递过来的放置模块参数的内存地址，在 insmod 一个模块时所携带的参数将以字符串的形式向内核空间传递。然后在 load_module 函数中，通过 strndup_user 的调用将用户空间的模块参数复制到内核空间。

```
args = strndup_user(uargs, ~0UL >> 1);
```

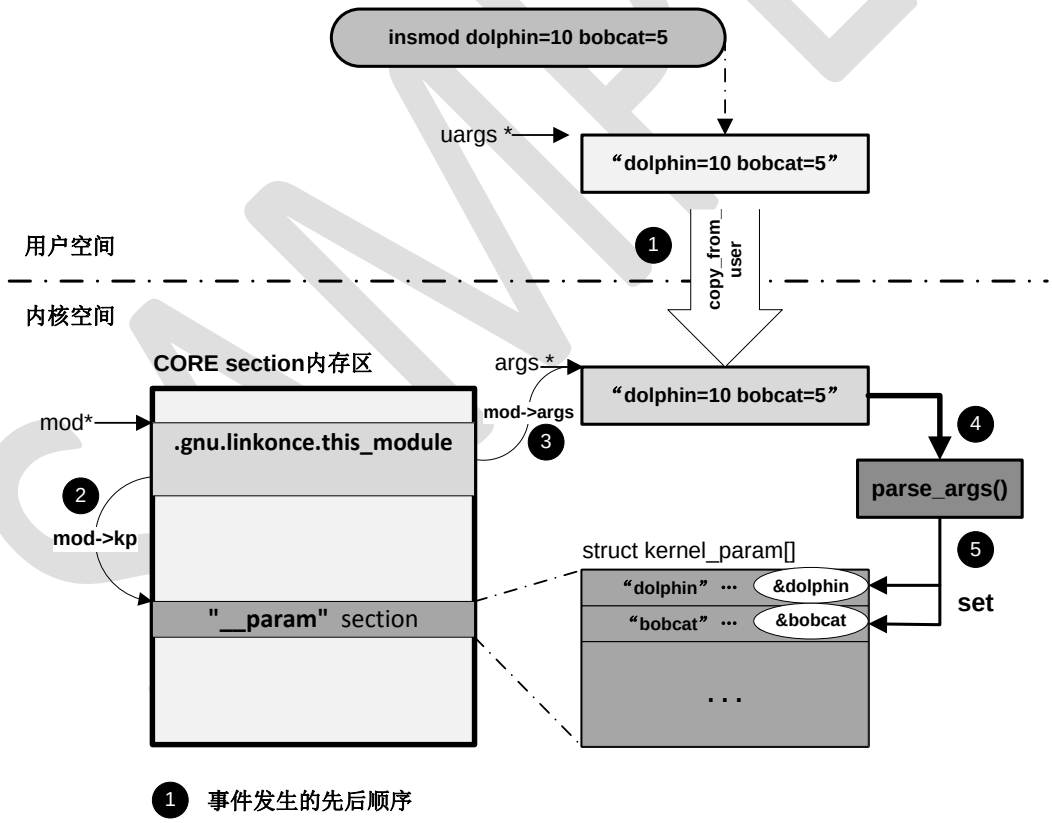

图 1-7 内核模块参数传递过程示意图

strndup_user 函数内部会调用 kmalloc 为在内核空间保存模块参数字符串分配一段内存区域，然后

通过 copy_from_user 将模块参数从用户空间复制到内核空间。

接着，在 HDR 视图的 section 被搬移到 CORE 和 INIT section 之后，load_module 通过下面的 section_objs 函数调用取得“__param” section 在内存空间的最终地址，并记录在 struct module 的 struct kernel_param *kp 成员变量中。

mod->kp = section_objs(info, "__param", sizeof(*mod->kp), &mod->num_kp);

mod->num_kp 为“__param” section 中 struct kernel_param 对象的个数。此后 mod->args 参数也将指向内核空间中保留参数字符串的内存区域。

最后调用 parse_args 函数将 mod->args 中的参数值复制到“__param” section 中对应的参数：

<kernel/module.c>

```
static struct module *load_module(void __user *umod,
                                  unsigned long len,
                                  const char __user *uargs)
{
    ...
    parse_args(mod->name, mod->args, mod->kp, mod->num_kp, NULL);
    ...
}
```

parse_args 函数的主要流程是，针对命令行中出现的每一个参数，用其参数名与“__param” section 中出现的每一个 struct kernel_param 对象的 name 成员进行匹配，如果匹配成功即认为找到了对应的参数，然后通过 struct kernel_param 中的 set 函数指针将参数值复制到 struct kernel_param 中 arg、str 或者 arr 所指向的地址空间。对于本节开始的例子，set 指针指向 param_set_int 函数，后者在 Linux 内核源码中由 STANDARD_PARAM_DEF 宏（kernel/params.c）负责定义，展开后的 param_set_int 如下：

```
int param_set_int(const char *val, struct kernel_param *kp)
{
    long l;
    int ret;

    ret = strict_strtol (val, 0, &l);
    if (ret < 0 || ((int)l != l))
        return ret < 0 ? ret : -EINVAL;
```

```
    *((int *)kp->arg) = l;   //将命令行中的参数值复制到“__param” section 的 kp 中
    return 0;
}
```

param_set_int 函数调用之后，模块“__param” section 中“dolphin” 和“bobcat” 所对应的 struct kernel_param 对象中的 arg 成员将被赋值为 10 和 5，也就是在 insmod 命令行中传入的模块实际参数值。这之后，内核模块将可以通过自身的 dolphin 和 bobcat 变量引用到这些传入的参数值。

现在我们可以看到，在把命令行的参数值复制到模块的参数这个过程中，module_param 宏所定义的“__param” section 起了桥梁的作用，透过“__param” section，内核可以找到模块中定义的参数所在的内存地址，继而可以用命令行中的参数值改写之。因为内核模块加载器在解析命令行参数时，对命令行中参数的构成格式有严格的要求，在参数名与参数值之间只能用“=”，且不能有空格，如果把前面的命令行写成如下形式（“dolphin=” 和“10” 之间有个空格）：

insmod demodev.ko dolphin= 10 bobcat=5

那么将会得到如下信息：

insmod: error inserting 'demodev.ko': -1 Invalid parameters

dmesg 中针对这个错误的输出如下：

[262282.558333] demodev: `' invalid for parameter `dolphin'

至此，我们已经理解了内核模块的参数机制，包括模块源码中如何声明参数，内核模块加载时如何把命令行中的参数复制到模块中等。

❍ 模块间的依赖关系

实际运行的系统中并不是只加载一个模块，模块可以随时添加进系统，也可以随时被卸载。这些内核模块之间并不是完全相对独立的，比如当一个模块引用到另一个模块中导出的符号时，这两个模块间就建立了依赖关系。因此依赖关系只存在于模块与模块之间，模块与内核之间不构成依赖关系，因为在模块生存期间我们不可能去卸载内核，当然内核也不可能引用到模块导出的符号。内核必须能跟踪模块间的这种依赖关系，只有这样，如果卸载一个模块时由于存在依赖关系而有可能影响到系统的稳定时，内核才可能采取必要的措施防止这种情况发生。

内核用 struct module 数据结构中定义的如下成员变量来跟踪模块间的这种依赖关系。

```
<include/linux/module.h>

#ifdef CONFIG_MODULE_UNLOAD
    /* What modules depend on me? */
    struct list_head source_list;
    /* What modules do I depend on? */
```

```
    struct list_head target_list;


#endif
```

其中的 struct list_head source_list 和 struct list_head target_list 用来构建有依赖关系模块的链表，对该链表的使用要结合数据结构 struct module_use：

```
<include/linux/module.h>

struct module_use {
    struct list_head source_list;
    struct list_head target_list;
    struct module *source, *target;
};
```

显然，模块间的这种依赖关系只有当模块能够被卸载时才有可能出现问题，对一个没有启用 CONFIG_MODULE_UNLOAD 宏的系统而言，因为模块被禁止卸载，因此内核无须对依赖关系做出处理。

模块依赖关系的建立最早发生在当前模块对象 mod 被加载时，模块加载函数调用 resolve_symbol 函数来解决其中一些“未解决的引用”符号。如果成功地在其他模块导出的符号中找到了指定的符号，那么 resolve_symbol 函数会将导出这一“未解决的引用”符号的模块记录在一个变量 struct module *owner 中，然后调用 ref_module(mod, owner)在模块 mod 和 owner 之间建立依赖关系。ref_module 函数在做一些必要的安全性检查之后调用 add_module_usage(mod, owner)在 mod 和 owner 模块间建立依赖关系，add_module_usage 函数的定义如下：

```
<kernel/module.c>

static int add_module_usage(struct module *a, struct module *b)
{
    struct module_use *use;


    DEBUGP("Allocating new usage for %s.\n", a->name);
    use = kmalloc(sizeof(*use), GFP_ATOMIC);
    if (!use) {
        printk(KERN_WARNING "%s: out of memory loading\n", a->name);
        return -ENOMEM;
    }
```

```
    use->source = a;
    use->target = b;
    list_add(&use->source_list, &b->source_list);
    list_add(&use->target_list, &a->target_list);
    return 0;
}
```

函数首先调用 kmalloc 分配一 struct module_use 型内存空间 use，然后将 use 中的 source 指向 mod 模块，target 指向 owner 模块，同时将 use 的 target_list 加入 mod 中的 target_list 指向的双向链表，将 use 的 source_list 加入 owner 中的 source_list 指向的双向链表。图 1-8 展示了通过 struct module_use 对象在三个模块间建立依赖关系的场景：

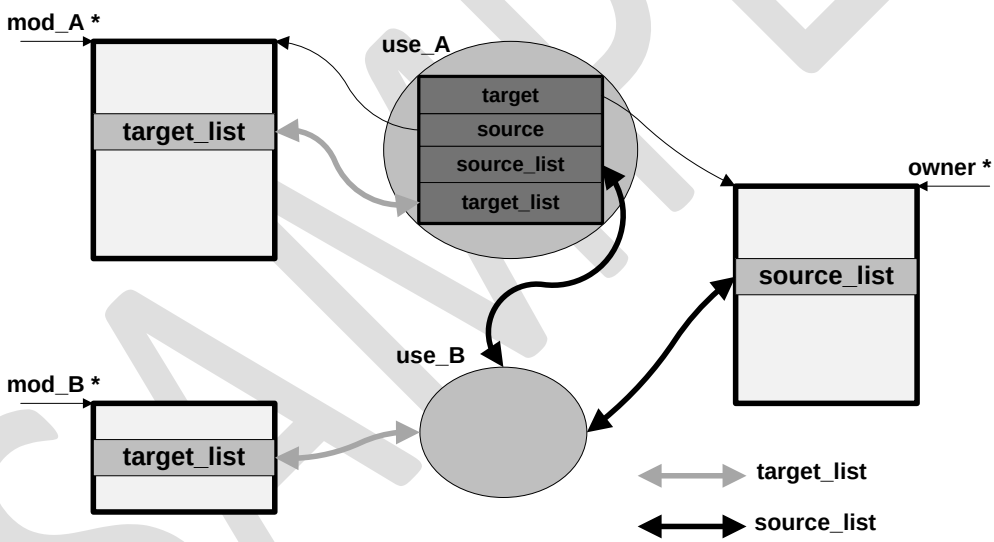

图 1-8 mod_A 和 mod_B 模块依赖于 owner 模块

图 1-8 中，模块 mod_A 和 mod_B 均依赖于模块 owner，mod_A 和 mod_B 之间则没有依赖关系。owner 模块先加入系统，它导出一个函数为模块 mod_A 和 mod_B 所使用。图中显示 mod_A 先于 mod_B 加入系统，在 mod_A 加入系统时，模块加载器创建了 use_A 对象在 mod_A 和 owner 间建立依赖关系。随后 mod_B 加入了系统，因为它和 owner 模块间存在依赖关系，加载器同样创建了 use_B 对象在 mod_B 和 owner 间建立关联。从图中可以看到，use_A 对象中的 source_list 成员已不再指向 owner->source_list，而是指向了 use_B->source_list，后者则和 owner->source_list 建

立了直接的链接关系。

如此，mod_A 和 mod_B 模块可以通过遍历其 target_list 成员知道所依赖的所有模块，而 owner 模块则可以通过遍历其 source_list 成员知道所有依赖于自己的模块。

当从系统中卸载一个模块时，系统必须确保没有其他模块依赖于该模块，根据上面的讨论，只要模块结构中的 source_list 是一空链表，就表明没有其他模块依赖于它。相关代码在模块卸载部分讨论。

❍ 模块的版本控制

版本控制主要用来解决内核模块和内核之间的接口一致性问题。所谓内核模块和内核之间的接口，简单地说是指由内核导出并被内核模块调用的那些符号。产生这种问题的根源在于内核模块和内核作为独立实体各自分开编译。想象一下，一个在 Linux 2.6.18 内核源码树基础上编译出来的内核模块能否成功加载到内核版本号为 2.6.35 的 Linux 系统中？如果模块使用的一个接口在 2.6.35 版本中已被改变或者废弃，那么前者将会因为无法找到一个未经定义的符号而导致加载失败，后者虽然有可能加载成功，但因为使用到了一个内核已经废弃的接口而需承担相应的风险。

因此，内核和模块之间必须协商出一种机制确保上述问题不会出现。Linux 系统对此的解决方案是使用接口的校验和，也叫接口 CRC 校验码。这种方法的基本思想非常简单，根据函数的参数生成一个大小为 4 字节的 CRC 校验码，当双方校验码相等时视为相同接口，否则为不同接口。

为了确保这种机制能够正常工作，内核必须首先启用 CONFIG_MODVERSIONS 这个宏，在此基础上内核模块在编译时也必须启用 CONFIG_MODVERSIONS，否则模块将会因为出现无法解决的未定义符号错误导致加载失败。显然这是个需要双方共同协作才能解决的问题。如果内核编译时没有启用 CONFIG_MODVERSIONS，那么系统将不会启用本节所说的方案，即使被加载的模块是在另一个启用 CONFIG_MODVERSIONS 的内核源码树的基础上编译出来的[18]。

下面首先从内核的角度来看看 CONFIG_MODVERSIONS 宏对内核导出符号产生的影响。在前面“EXPORT_SYMBOL 的内核实现”一节中我们看到了内核用于导出符号的 EXPORT_SYMBOL 相关宏的定义，其中出现了一个__CRC_SYMBOL 宏，如果在内核编译时启用了 CONFIG_MODVERSIONS，即对内核启用了版本控制特性，那么__CRC_SYMBOL 的定义为：

```
#define __CRC_SYMBOL(sym, sec)                                  \
    extern void *__crc_##sym __attribute__((weak));             \
    static const unsigned long __kcrctab_##sym                  \
    __used                                                      \
    __attribute__((section("___kcrctab" sec "+" #sym), unused))  \
    = (unsigned long) &__crc_##sym;
```

[18] 这种情形下模块加载器依赖 vermagic 对模块与内核的接口一致性进行检查，本章后续部分会对此进行讨论。

如此，对于 EXPORT_SYMBOL(my_exp_function)的例子，将会在原来的基础上新增如下的定义：

```
extern void * __crc_ my_exp_function;
static const unsigned long __kcrctab_my_exp_function        \
__attribute__((section("___kcrctab" sec "+" #sym), unused))
= (unsigned long) &__crc_my_exp_function;
```

__kcrctab_my_exp_function 用来保存__crc_my_exp_function 变量地址并将其放在一个名为“__kcrctab+*”的 section 中（对于 EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE，其所在的 section 的名称分别为“__kcrctab_gpl+*”和“__kcrctab_gpl_future+*”）。

由此可见，如果内核编译时启用了 CONFIG_MODVERSIONS 宏，那么对于每一个导出的符号，都会生成一个对应的 CRC 校验码。反之，如果内核编译时没有启用 CONFIG_MODVERSIONS 宏，那么系统将不会为导出的符号产生 CRC 校验码。“__kcrctab+*”等 section 存在的意义在于当符号查找时，可以通过该 section 找到对应符号的校验码，以此来判断模块所使用的接口是否和当前正运行的内核提供的接口相匹配。

CONFIG_MODVERSIONS 宏对内核的另一个影响存在于加载内核模块的过程中。前面在“对‘未解决的引用’符号（unresolved symbol）的处理”部分中提到，对于模块中出现的未定义的符号，内核会调用 resolve_symbol 函数予以解决，在 resolve_symbol 函数的内部会调用 find_symbol 来查找该符号。如果成功查找到，则函数接下来会调用 check_version 对这种接口进行校验码的验证。在没有启用 CONFIG_MODVERSIONS 的系统中，这个函数直接返回 1，因此没有启用 CONFIG_MODVERSIONS 的内核不会进行接口一致性的检验。

接下来看一下 CONFIG_MODVERSIONS 对内核模块的影响。

首先，前面提到的 CONFIG_MODVERSIONS 宏对内核导出符号的影响同样适用于模块导出的符号。

其次，启用 CONFIG_MODVERSIONS 会导致模块的编译工具链在最终模块的 ELF 文件中产生一个名为“__versions”的 section[19]，打开模块源码所在目录下的.mod.c 文件，会发现类似如下代码：

```
static const struct modversion_info ____versions[]
__used
__attribute__((section("__versions"))) = {
        { 0x372ac21f, "module_layout" },
        { 0x3ec8886f, "param_ops_int" },
        { 0x810b3618, "param_ops_string" },
        { 0x27e1a049, "printk" },
```

[19] “__versions”section 是由 scripts/mod/modpost.c 生成的，为叙述简单起见，本书将其统称为工具链。

```
        { 0xb4390f9a, "mcount" },
};
```

代码定义了一个类型为 struct modversion_info 的数组____versions，放在“__versions”section 中。struct modversion_info 的定义如下：

```
<include/linux/module.h>

struct modversion_info
{
    unsigned long crc;
    char name[MODULE_NAME_LEN];
};
```

可见当编译内核模块时，若对应的内核配置文件中启用了 CONFIG_MODVERSIONS，则模块最终的 ELF 文件中会生成一个名为“__versions”的 section ,该 section 会将模块中的所有“未解决的引用”符号名和对应的校验码放入其间。在前面的那个.mod.c 的示例中，printk 作为模块的一个“未解决的引用”符号被放在了“__versions”section 中，工具链为其产生的 CRC 校验码为 0x27e1a049。

在分析完启用 CONFIG_MODVERSIONS 宏对内核和内核模块双方的影响后，再回过头来看一看当模块加载时处理“未解决的引用”符号时是如何对接口一致性进行验证的。这种验证是通过在 resolve_symbol 函数里调用 check_version 函数完成的，check_version 函数的完整定义如下：

```
<kernel/module.c>

#ifdef CONFIG_MODVERSIONS
static int check_version(Elf_Shdr *sechdrs,
                         unsigned int versindex,
                         const char *symname,
                         struct module *mod,
                         const unsigned long *crc,
                         const struct module *crc_owner)
{
    unsigned int i, num_versions;
    struct modversion_info *versions;
```

```
    /* Exporting module didn't supply crcs?  OK, we're already tainted. */
    if (!crc)
        return 1;

    /* No versions at all?  modprobe --force does this. */
    if (versindex == 0)
        return try_to_force_load(mod, symname) == 0;

    versions = (void *) sechdrs[versindex].sh_addr;
    num_versions = sechdrs[versindex].sh_size
        / sizeof(struct modversion_info);

    for (i = 0; i < num_versions; i++) {
        if (strcmp(versions[i].name, symname) != 0)
            continue;

        if (versions[i].crc == maybe_relocated(*crc, crc_owner))
            return 1;
        DEBUGP("Found checksum %lX vs module %lX\n",
               maybe_relocated(*crc, crc_owner), versions[i].crc);
        goto bad_version;
    }

    printk(KERN_WARNING "%s: no symbol version for %s\n",
           mod->name, symname);
    return 0;

bad_version:
    printk("%s: disagrees about version of symbol %s\n",
           mod->name, symname);
    return 0;
```

```
}
#else
static inline int check_version(Elf_Shdr *sechdrs,
                unsigned int versindex,
                const char *symname,
                struct module *mod,
                const unsigned long *crc,
                const struct module *crc_owner)
{
    return 1;
}
#endif
```

在定义了 CONFIG_MODVERSIONS 的前提下，check_version 用一个 for 循环在“__versions”section 中进行遍历，对每一个 struct modversion_info 元素和找到的符号名 symname 进行匹配，如果匹配成功，再进行接口的校验码比较，如果校验码相等，说明模块所使用的接口和内核导出的接口是一致的，否则产生版本不匹配的错误。

关于校验码的计算[20]，内核源码树中提供了一个产生 CRC 校验码的工具 genksyms，它位于 Linux 内核源码的 scripts/genksyms 目录下。内核模块编译工具链通过它来生成导出符号的 CRC 校验码。比如下面的 demodev.h 文件，其中导出了一个函数符号 my_exp_function：

```
<demodev.h>
int my_exp_function(int a, int b);
EXPORT_SYMBOL(my_exp_function);
```

那么使用如下命令就可以看到 genksyms 为 my_exp_function 生成的 CRC 校验和：

```
root@AMDLinuxFGL:/home/dennis/book# gcc -E demodev.h -D__GENKSYMS__
-D__KERNEL__ | ./genksyms > demodev.crc
```

打开 demodev.crc 文件，可以看到类似下面的内容：

```
__crc_my_exp_function = 0x48a0010f;
```

genksyms 只为由 EXPORT_SYMBOL 系列的宏导出符号生成 CRC 校验码。对于内核和内核模块而

[20] 关于校验码生成算法，本书不作详细讨论，读者可以简单认为：$V_{CRC}=f$(导出函数名，函数的参数表)。

言，它们在各自编译链接阶段均独立使用 genksyms 来为导出的符号和那些“未解决的引用”符号生成校验和。

图 1-9 展示了接口校验码在验证接口有效性时的一个具体例子，图中的模块 A 使用到了内核导出的一个函数 demofunc，该模块当前的.ko 文件最初是在内核源码树的 2.6.18 版本上编译而成的，现在要在运行内核 B（版本号为 2.6.35）的 Linux 系统上加载该模块。现在的问题是，内核导出的函数 demofunc 在从版本 2.6.18 到 2.6.35 的演变过程中发生了改变，新版本内核源码中该函数原型较旧版本多出了一个参数。如果内核在加载模块时没有版本控制机制，那么在运行新版本的 Linux 系统中加载该模块时会发生什么呢？答案是结果不确定，内核也许会正常运行，也许会崩溃，但无论如何这是个潜在的危险行为。

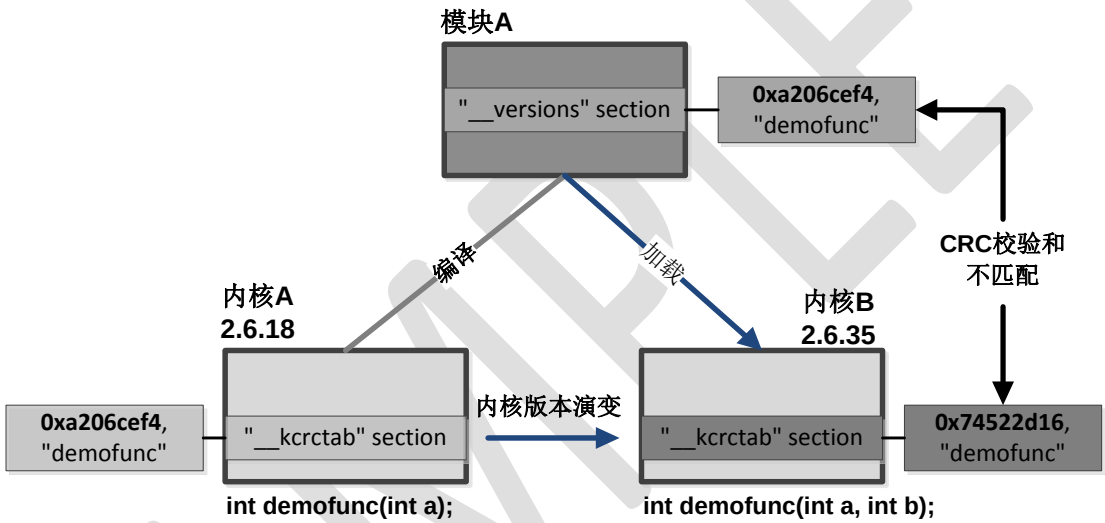

图 1-9 新旧内核导出的接口 CRC 不匹配

如果内核 A 和 B 在当初配置时都启用了版本控制机制，那么它们除了导出 demofunc 这个符号外，还会分别为这个接口生成对应的 CRC 校验码。当图中的模块 A 在内核 A 源码树的基础上进行编译时，模块的构造工具链也会负责为模块中这个“未解决的引用”符号 demofunc 生成 CRC 校验码，因为模块编链工具链和内核工具链使用的 CRC 校验码生成工具是完全一样的，所以它们都会获得针对 demofunc 函数接口的一个 CRC 校验码，图中的值为 0xa206cef4。

而对于图中的内核 B 而言，虽然使用了同样的 CRC 校验码生成工具，但因为函数多出了一个参数，所以内核 B 得到的 demofunc 函数接口校验码为 0x74522d16。这样，当将图中的模块 A 向内核 B 加载时，会因为两者的 CRC 校验码不匹配而导致无法加载该模块，从而避免可能造成内核不稳定的危险行为。

基于上面的讨论，我们建议在对内核进行配置时启用 CONFIG_MODVERSIONS 选项，这样编译出的内核在模块加载时就会启动版本控制机制。如果模块在一个没有启用 CONFIG_MODVERSIONS 宏的内核源码树基础上进行编译链接，模块的构造工具将不会为模块中导出的符号和那些“未解决的引用”符号生成 CRC 校验码，如果将这样的模块安装到一个启用了 CONFIG_MODVERSIONS 的

Linux 系统中，加载该模块就会在版本控制环节出现问题，这种情况下即使强行加载也会导致内核的污染。

最后要强调的是，对于 CONFIG_MODVERSIONS 机制，模块的编译工具链只对导出的符号产生 CRC 校验码，最明显的，内核在编译过程中所有导出的符号都会被记录到一个名为“Module.symvers”的文件中，下面是该文件的一部分：

root@AMDLinuxFGL:/home/dennis/Linux/kernel/linux-3.1.6# cat Module.symvers | grep printk

0xc60f75ec	*__ftrace_vprintk*	*vmlinux*	*EXPORT_SYMBOL_GPL*
0x85133f5c	*netdev_printk*	*vmlinux*	*EXPORT_SYMBOL*
…			
0x27e1a049	*printk*	*vmlinux*	*EXPORT_SYMBOL*
0xbf8ba54a	*vprintk*	*vmlinux*	*EXPORT_SYMBOL*
0x72741f25	*trace_vbprintk*	*vmlinux*	*EXPORT_SYMBOL_GPL*

其中每行第二列就是导出的符号，第一列是该导出符号的 CRC 校验码，第三列是导出该符号的模块，第四列是导出符号的类型。

如果一个独立编译的内核模块，比如 demodev.ko，引用到了内核导出的符号，比如 printk，那么在 demodev.ko 的编译链接过程中，工具链会到内核源码所在的目录下查找 Module.symvers 文件，将得到的 printk 的 CRC 校验码 0x27e1a049 记录到 demodev.ko 的“__versions”section 中，换句话说，工具链不会在使用导出符号的地方重新为之生成一个 CRC 校验码。在 demodev 模块成功编译后，读者可以在其源码所在的目录下发现一个名为“demodev.mod.c”的文件，在该文件中可以发现类似下面的内容：

```
<demodev.mod.c>

…

static const struct modversion_info ____versions[]

__used

__attribute__((section("__versions"))) = {

        { 0x372ac21f, "module_layout" },

        { 0x3ec8886f, "param_ops_int" },

        { 0x810b3618, "param_ops_string" },

        { 0x27e1a049, "printk" },
```

```
        { 0xb4390f9a, "mcount" },
};
...
```

从中可以看到 printk 函数的校验码和内核源码树中 Module.symvers 中的完全一样。

从这里还可以引申出一个有趣的问题，如果有两个模块 A 和 B，A 导出的一个符号“a_sym”为 B 所用，因为 A 和 B 都是各自独立编译链接，意味着 A 导出的符号及其 CRC 校验码将放在 A 所在目录的 Module.symvers 文件中，这样在编译链接 B 模块时将会产生一个 WARNING，大意是"a_sym" [/home/dennis/Linux/B.ko] undefined!，即便在 B 模块源码中加入 extern … a_sym 这样的声明也无法消除这个 WARNING，产生该 WARNING 的原因是 B 模块只能找到 Linux 内核导出符号所在的 Module.symvers 文件，而无法找到 A 模块产生的 Module.symvers 文件，所以在模块编译阶段无法确定最终在模块加载时是否能找到这个“a_sym”符号，因此只是简单地给了个 WARNING。从表象上看，这个 WARNING 尚不是致命的，因为还有模块加载器这最后一道防线，正如前面所讨论的，它会到内核及所有加载进系统的内核模块所导出的符号表中查找“a_sym”，假如在 B 模块加载前 A 模块已经被加载进系统，那么有理由相信模块加载器是可以帮 B 模块找到“a_sym”符号的，但是遗憾的是模块 B 通常都不可能被加载成功，dmesg 对此给出的提示信息是大约是“B: no symbol version for a_sym”云云。所以此时可去查看 B 模块的.mod.c 文件，在它的 versions[]里一定不会有“a_sym”的身影，因为工具链无法获得它的 CRC 校验码。知道了原因问题就好解决了，最简单的 把 A 模块 Module.symvers 文件的内容添加到 Linux 内核源码树中的 Module.symvers 文件中，这样就可以消除 WARNING 而且 B 模块也可以成功加载。如果此时再看 B 模块的.mod.c 文件，就会发现符号“a_sym”及其 CRC 校验码已经出现在 versions[]中，并且 modinfo 显示 B 模块有了个依赖关系 depends：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/framewk/device#modinfo B.ko
filename:       B.ko
description:    A simple kernel module
author:         dennis chen @ AMDLinuxFGL
license:        GPL
srcversion:     5D9E027831864213C7A6B2D
depends:        A
vermagic:       3.1.6 SMP mod_unload modversions
```

❍ 模块的信息（modinfo）

模块的最终 ELF 文件中都会有一个名为“.modinfo”的 section，这个 section 以文本的形式保留着模块的一些相关信息。在 Linux 环境下可以用 modinfo 工具来查看一个模块存储的信息，比如下面

modinfo 输出的 demodev.ko 模块的信息：

root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# modinfo demodev.ko

filename: *demodev.ko*

description: *A simple kernel module as an illustration*

author: *dennis chen @ AMDLinuxFGL*

license: *GPL*

srcversion: *D34E5D0B9552602BF236A4A*

depends:

vermagic: *3.1.6 SMP mod_unload modversions*

parm: *dolphin:int*

parm: *demodev_str:string*

模块的源码中用 MODULE_INFO 宏来向该 section 添加模块信息 ,MODULE_INFO 宏的相关定义如下：

```
<include/linux/module.h>

#ifdef MODULE
#define ___module_cat(a,b) __mod_ ## a ## b
#define __module_cat(a,b) ___module_cat(a,b)
#define __MODULE_INFO(tag, name, info)                                  \
static const char __module_cat(name,__LINE__)[]                         \
  __used  __attribute__((section(".modinfo"),unused ,  aligned(1)))   \
   = __stringify(tag) "=" info
#else   /* !MODULE */
/* This struct is here for syntactic coherency, it is not used */
#define __MODULE_INFO(tag, name, info)                                  \
   struct __module_cat(name,__LINE__) {}
#endif
/* Generic info of form tag = "info" */
#define MODULE_INFO(tag, info) __MODULE_INFO(tag, tag, info)
```

在 MODULE 没有定义的情况下 ,MODULE_INFO 等同空定义。而对于内核模块而言 ,MODULE_INFO

在“.modinfo”section 中定义了一个类似"tag=info"的字符串，内核中通过调用 get_modinfo 函数来获得 tag 所在字符串的值 info。内核模块开发者可以通过 MODULE_INFO 宏来向模块添加一些相关信息，此外，模块的编译工具链会自动在模块中添加如下两个 info: MODULE_INFO(vermagic, VERMAGIC_STRING)和 MODULE_INFO(srcversion, "0E163D6E46F5467DAFBB8FA")，读者可以在.mod.c 文件中看到这两处声明

在模块加载的早期阶段，加载器需要调用 check_modinfo 函数来获得“.modinfo”section 中的相关信息以便确定当前要加载的模块是否满足某些特定的要求，这些信息包括：

1） 模块的 vermagic

内核和内核模块的 vermagic 都是通过 MODULE_INFO 定义的一个 VERMAGIC_STRING 字符串，后者实际上是一个生成字符串的宏，该宏会根据不同的内核配置信息生成不同的字符串。模块加载过程中会检查模块中的 vermagic 是否和当前运行的内核定义的 vermagic 一致，如果不一致加载将失败，dmesg 命令会发现类似下面的错误信息：

demodev: version magic '3.1.6 SMP mod_unload modversions' should be '3.1.6 SMP mod_unload '

上面的信息输出对应着 load_module 函数调用链中如下的代码片段：

<kernel/module.c>

```
static int check_modinfo(struct module *mod, struct load_info *info)
{
    const char *modmagic = get_modinfo(info, "vermagic");
    int err;

    /* This is allowed: modprobe --force will invalidate it. */
    if (!modmagic) {
        err = try_to_force_load(mod, "bad vermagic");
        if (err)
            return err;
    } else if (!same_magic(modmagic, vermagic, info->index.vers)) {
        printk(KERN_ERR "%s: version magic '%s' should be '%s'\n",
                mod->name, modmagic, vermagic);
        return -ENOEXEC;
    }
    ...
```

```
    /* Set up license info based on the info section */
    set_license(mod, get_modinfo(info, "license"));
}
```

该代码片段首先在当前模块的“.modinfo”section 中查找 vermagic 对应的字符串 modmagic，如果找到则和当前内核的 vermagic 字符串进行比较，以确定两者是否匹配，不匹配的话模块将加载失败。读者可以通过如下命令查看一个模块的“.modinfo”section 的内容：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# readelf -p .modinfo demodev.ko

String dump of section '.modinfo':
[     0]  description=A simple kernel module as an illustration
[    36]  author=dennis chen @ AMDLinuxFGL
[    57]  license=GPL
[    63]  parmtype=demodev_str:string
[    7f]  parmtype=dolphin:int
[    94]  srcversion=D34E5D0B9552602BF236A4A
[    b7]  depends=
[    c0]  vermagic=3.1.6 SMP mod_unload modversions
```

所以对于 demodev.ko 这个模块而言，“.modinfo”section 中 vermagic 对应的字符串就为"3.1.6 SMP mod_unload modversions"，打开模块源码目录下的 demodev.mod.c 文件，可以看到下列信息：

```
MODULE_INFO(vermagic, VERMAGIC_STRING);
```

内核模块中用来产生 vermagic 的 MODULE_INFO 是通过 scripts/mod/modpost.c 文件自动生成的，内核模块开发者无须在源码中显式添加这一信息。

前面已经提到，VERMAGIC_STRING 实际上是由内核源码树的相关配置信息所组合而成的一个字符串，如下所示：

```
<include/linux/vermagic.h>

#define VERMAGIC_STRING                         \
    UTS_RELEASE " "                             \
    MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT             \
```

```
MODULE_VERMAGIC_MODULE_UNLOAD MODULE_VERMAGIC_MODVERSIONS\
MODULE_ARCH_VERMAGIC
```

读者很容易将 VERMAGIC_STRING 所组合的信息与"3.1.6 SMP mod_unload modversions "这样的具体字符串对应起来。因为不同的内核源码树的配置信息不一定相同，所以从这个角度而言，vermagic 也可以看做是模块版本控制的一部分。

check_modinfo()函数最后的 set_license 用来处理模块 license 相关的问题，这恰好是接下来要讨论的话题。

2） 模块的 license

模块的 license 在模块源码中以 MODULE_LICENSE 宏引出，该宏的主体就是 MODULE_INFO：

```
#define MODULE_LICENSE(_license)    MODULE_INFO(license, _license)
```

内核模块加载过程中，会调用 license_is_gpl_compatible 来确定模块的 license 是否与 GPL 兼容：

```
<include/linux/license.h>

static inline int license_is_gpl_compatible(const char *license)
{
    return (strcmp(license, "GPL") == 0
        || strcmp(license, "GPL v2") == 0
        || strcmp(license, "GPL and additional rights") == 0
        || strcmp(license, "Dual BSD/GPL") == 0
        || strcmp(license, "Dual MIT/GPL") == 0
        || strcmp(license, "Dual MPL/GPL") == 0);
}
```

非以上形式的 license 被认为与 GPL 不兼容，这样的模块在加载进系统后会导致内核被污染，内核用 mod 对象的 unsigned int taints 成员记录一个模块是否会污染内核。在前面刚提到的 check_modinfo()函数中我们看到它最后调用了 set_license()函数，后者用来对模块的 license 做检查，set_license 的代码为：

```
<kernel/module.c>

static void set_license(struct module *mod, const char *license)
{
    if (!license)
        license = "unspecified";
```

```
    if (!license_is_gpl_compatible(license)) {
        if (!test_taint(TAINT_PROPRIETARY_MODULE))
            printk(KERN_WARNING "%s: module license '%s' taints "
                "kernel.\n", mod->name, license);
        add_taint_module(mod, TAINT_PROPRIETARY_MODULE);
    }
}
```

函数调用 license_is_gpl_compatible()来判断模块是否 GPL 兼容，如果不是，将把该模块记录成会污染内核的模块：

```
mod->taints |= (1U << 0);
```

这样，以后就可以通过 mod->taints 来判断要加载的模块是否 GPL 兼容。而对于运行中的系统是否被污染，内核用一个 unsigned long 型全局变量 tainted_mask 来表示，在系统因故障挂起时，tainted_mask 会影响系统调试信息的输出，以告之内核是否曾被污染过。

另外，non-GPL 的模块无法使用内核或其他内核模块用 EXPORT_SYMBOL_GPL 导出的符号，在加载这样的模块时将出现"Unknown symbol in module"类似的错误信息，我们在本章的前面的内容中已经详细讨论过相关细节。

1.3.4 sys_init_module（第二部分）

load_module 函数完成模块加载几乎所有的艰苦工作之后，重新返回到 sys_init_module,后者在 load_module 的基础上所做的事情相对就比较简单了，主要的工作包括：

- 调用模块的初始化函数

本章“struct module 类型变量 mod 初始化”部分中已经提到了 mod 中 init 函数指针是如何指向模块源码中的初始化函数的，现在由 sys_init_module 负责调用它，相关代码如下：

<kernel/module.c>

```
sys_init_module( void __user * umod, unsigned long  len,  const char __user *  uargs)
{
    ...
    if (mod->init != NULL)
        ret = do_one_initcall(mod->init);
    if (ret < 0) {
        /* Init routine failed: abort.  Try to protect us from
```

```
                buggy refcounters. */
        mod->state = MODULE_STATE_GOING;
        synchronize_sched();
        module_put(mod);
        blocking_notifier_call_chain(&module_notify_list,
                                     MODULE_STATE_GOING, mod);
        free_module(mod);
        wake_up(&module_wq);
        return ret;
    }
    …
}
```

从以上代码可以看出，模块初始化函数对模块而言并不是必须的。如果模块提供了初始化函数，那么它将在 do_one_initcall 函数内部被调用。如果模块初始化函数被成功调用，那么模块就算是被加载进了系统，因此需要更新模块的状态为 MODULE_STATE_LIVE：

```
mod->state = MODULE_STATE_LIVE;
```

原则上，如果内核模块提供了初始化函数 init，那么也必须提供对应的退出函数 exit。

❍ 释放 INIT section 所占用的空间

模块一旦被成功加载，HDR 视图和 INIT 区域所占的内存空间将不再会被用到，因此需要释放它们。在 sys_init_module 函数中，释放 INIT 区域所在内存空间由函数 module_free 完成，后者通过调用 vfree 来释放 INIT 区域所在的内存空间（mod->module_init）：

```
module_free(mod, mod->module_init);
```

而 HDR 视图所占空间的释放则发生得更早，它实际上出现在 load_module 函数的最后部分：

```
<kernel/module.c>

static noinline struct module *load_module(void __user *umod,
                                           unsigned long len,
                                           const char __user *uargs)
{
    struct load_info info = { NULL, };
    …
```

```
    vfree(info.hdr);
    ...
}
```

模块成功加载进系统之后，正在运行的内核和系统中所有加载的模块关系如图 1-10 所示。

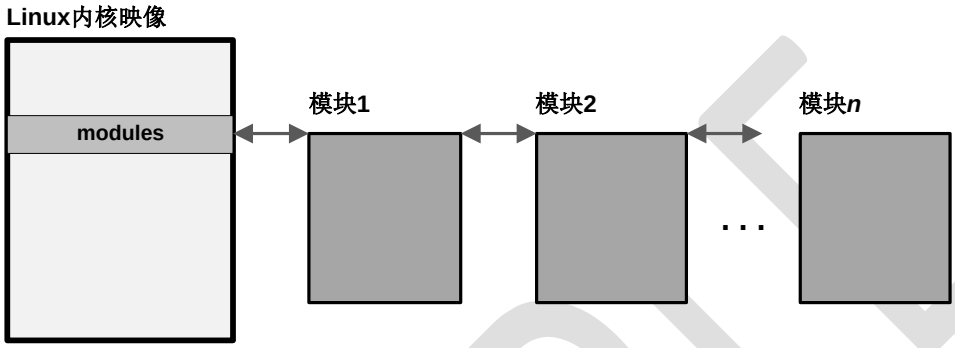

图 1-10 内核与内核模块

内核用一全局变量 modules 链表记录系统中所有已加载的模块。

❍ 呼叫模块通知链

Linux 内核提供了一个很有趣的特性，也就是所谓的通知链（notifier call chain），这个特性不只是在模块的加载过程中会使用到，在内核系统的其他组件中也常常会使用到。通过通知链，模块或者其他的内核组件可以向其感兴趣的一些内核事件进行注册，当该事件发生时，这些模块或者组件当初注册时的回调函数将会被调用。内核模块机制中实现的模块通知链 module_notify_list 就是内核中众多通知链中的一条。通知链背后的实现机制其实很简单，通过链表的形式，内核将那些注册进来的关注同类事件的结点构成一个链表，当某一特定的内核事件发生时，事件所属的内核组件负责遍历该通知链上的所有结点，调用结点上的回调函数。所有的通知链头部都是一个 struct blocking_notifier_head 类型的变量，该类型的定义为：

```
<include/linux/notifier.h>

struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block __rcu *head;
};
```

这里以内核模块的通知链 module_notify_list 为例，来讨论一下通知链的实现原理[21]。

[21] 内核模块中的通知链机制目前主要用于 kprobe。

module_notify_list 为一全局变量，用来管理所有对内核模块事件感兴趣的通知结点，其定义为：

<kernel/module.c>

```
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

上述声明其实是定义并初始化了一个类型为 struct blocking_notifier_head 的对象 module_notify_list。如果一个内核模块想了解当前系统中所有跟模块相关的事件，可以调用 register_module_notifier 向内核注册一个结点对象，该结点对象中包含有一个回调函数。当 register_module_notifier 函数成功向系统注册了一个回调结点之后，系统中所有那些模块相关的事件发生时都会调用到这个回调函数。为了具体了解幕后的机制，下面先来看看 register_module_notifier 函数在内核源码中的实现：

<kernel/module.c>

```
int register_module_notifier(struct notifier_block * nb)
{
    return blocking_notifier_chain_register(&module_notify_list, nb);
}
```

函数的参数是个 struct notifier_block 型指针，代表一个通知结点对象。struct notifier_block 的定义为：

<include/linux/notifier.h>

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block __rcu *next;
    int priority;
};
```

其中，notifier_call 就是所谓的通知结点中的回调函数，next 用来构成通知链，priority 代表一个通知结点优先级，用来决定通知结点在通知链中的先后顺序，数值越大代表优先级越高。

blocking_notifier_chain_register 最终通过调用 notifier_chain_register 函数将一个通知结点加入 module_notify_list 管理的链表。在向一个通知链中加入新结点时，系统会把各结点的 priority 作为一个排序关键字进行简单排序，其结果是越高优先级的结点越靠近头结点，当有事件发生时，它将优先被通知到。图 1-11 展示了多个模块在同一条通知链上调用了 register_module_notifier，由此形成的该调用链结构示意图：

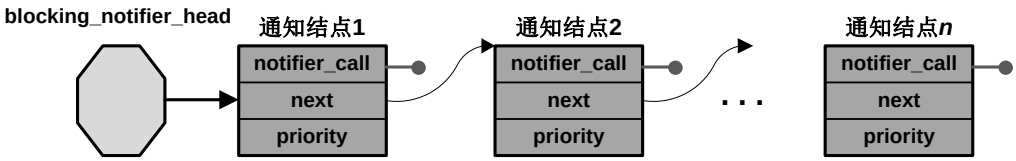

图 1-11 内核中的一条通知链构成示意图

与 register_module_notifier 相反，如果要从一条通知链中注销一个通知结点，那么应该使用 unregister_module_notifier 函数，该函数的原型为：

int unregister_module_notifier(struct notifier_block * nb)；

以上讨论了如何向/从系统中的一条通知链注册/注销一个通知结点，接下来看看当某个特定的内核事件发生时，通知链上各结点的回调函数如何被触发。以内核模块加载过程为例，sys_init_module 函数在调用完 load_module 之后，会通过 blocking_notifier_call_chain 函数来通知调用链 module_notify_list 上的各结点，例如，如果 load_module 函数成功返回，表明模块加载的大部分工作已经完成，此时 sys_init_module 会通过调用 blocking_notifier_call_chai()函数来通知 module_notify_list 上的结点，一个模块正在被加入系统（MODULE_STATE_COMING）：

```
<kernel/module.c>

sys_init_module( void __user * umod, unsigned long  len,  const char __user *  uargs)
{
    ...
    mod = load_module(umod, len, uargs);
    if (IS_ERR(mod))
        return PTR_ERR(mod);

    blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod);
    ...
}
```

上面代码段中的 blocking_notifier_call_chain 函数将使通知链 module_notify_list 上的各结点的回调函数均被调用，其实现原理可以简单概括为遍历 module_notify_list 上的各结点，依次调用各结点上的 notifier_call 函数。读者从源码中也可以发现，对应模块加载的其他阶段（MODULE_STATE_LIVE 和 MODULE_STATE_GOING），内核模块加载器都会调用 blocking_notifier_call_chain 函数来通知 module_notify_list 上的各个通知结点。

最后用一个实际的例子来加深读者对模块通知链的理解，下面所列内核模块 modnoti.ko 的源码展示了如何利用模块通知链来监听系统中与模块相关的事件：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>

static struct notifier_block *pnb = NULL;

static char *mstate[] = {"LIVE", "COMING", "GOING"};

//通知结点对象 pnb 上的回调函数
int get_notify(struct notifier_block *p, unsigned long v, void *m)
{
    printk("module <%s> is %s, p->priority=%d\n", ((struct module *)m)->name, mstate[v],
        p->priority);
    return 0;
}

static int hello_init(void)
{
    //分配一个 struct notifier_block 通知结点对象
    pnb = kzalloc(sizeof(struct notifier_block), GFP_KERNEL);
    if(!pnb)
        return -1;
    //通知结点上的回调函数
    pnb->notifier_call = get_notify;
    pnb->priority = 10;
    register_module_notifier(pnb);
    printk("A listening module is coming...\n");
    return 0;
}
```

```
static void hello_exit(void)
{
    unregister_module_notifier(pnb);
    kfree(pnb);
    printk("A listening module is going\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE(“GPL”);
```

当通过“insmod modnoti.ko”把该内核模块加入系统后，在 dmesg 的输出中可以发现如下的输出信息：

root@AMDLinuxFGL:/home/dennis/book/gene-module# dmesg

[230330.738545] A listening module is coming...

[230330.738555] module <modnoti> is LIVE, p->priority=10

因为 modnoti.ko 中向通知链 module_notify_list 注册一个结点的动作发生在模块的初始化函数中，所以虽然 sys_init_module 函数在模块初始化函数前调用了 blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod)，但是此时 modnoti 模块的通知结点还没有出现在 module_notify_list 通知链中，因而 modnoti 只能接收到发生在模块初始化函数之后的 blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_LIVE, mod)的通知消息。

在成功加载完 modnoti.ko 模块后，再通过“insmod demodev.ko”来加载另一个内核模块，此时 dmesg 的输出又多出了如下两行：

[230357.414452] module <demodev> is COMING, p->priority=10

[230357.414467] module <demodev> is LIVE, p->priority=10

对比 sys_init_module 函数源码，读者应该很容易理解为何出现上述两行输出。

1.3.5 模块的卸载

相对于模块的加载，从系统中卸载一个模块的任务相对要轻松许多。将一个模块从系统中卸载，使用 rmmod 命令，比如“rmmod demodev”。rmmod 通过系统调用 sys_delete_module 来完成卸载工作，该函数原型如下：

long sys_delete_module(const char __user * name_user, unsigned int flags);

❍ find_module 函数

sys_delete_module 函数首先将来自用户空间的欲卸载模块名用 strncpy_from_user 函数复制到内核空间：

```
if (strncpy_from_user(name, name_user, MODULE_NAME_LEN-1) < 0)
        return -EFAULT;
```

然后调用 find_module 函数在内核维护的模块链表 modules 中利用 name 来查找要卸载的模块。find_module 函数的定义如下：

<kernel/module.c>

```
/* Search for module by name: must hold module_mutex. */
struct module *find_module(const char *name)
{
    struct module *mod;

    list_for_each_entry(mod, &modules, list) {
        if (strcmp(mod->name, name) == 0)
            return mod;
    }
    return NULL;
}
```

函数通过 list_for_each_entry 在 modules 链表中遍历每一个模块 mod ,通过前面的讨论我们知道，全局变量 modules 用来管理系统中所有已加载的模块形成的链表。如果查找到指定的模块，则函数返回该模块的 mod 结构，否则返回 NULL。

❍ 检查模块依赖关系

如果 sys_delete_module 函数成功查找到了要卸载的模块，那么接下来就要检查是否有别的模块依赖于当前要卸载的模块，出于系统稳定性的考虑，一个有依赖关系的模块不应该从系统中卸载掉。在前面“模块间的依赖关系”部分中已经讨论了内核如何跟踪系统中各模块之间的依赖关系，这里只需检查要卸载模块的 source_list 链表是否为空，即可判断这种依赖关系，相关代码段为：

```
if (!list_empty(&mod->source_list)) {
        /* Other modules depend on us: get rid of them first. */
```

```
        ret = -EWOULDBLOCK;
        goto out;
}
```

在本章稍早前的“模块间的依赖关系”部分我们已经讨论过，如果一个模块对象 mod 的成员 source_list 是一空链表，即意味着系统中没有别的模块依赖于 mod，此时就可以继续模块后续的卸载行为，否则模块将不会被成功卸载(返回错误码 EWOULDBLOCK 到用户空间的 rmmod)。

- free_module 函数

如果一切正常，sys_delete_module 函数最后会调用 free_module 函数来做模块卸载末期的一些清理工作，包括更新模块的状态为 MODULE_STATE_GOING，将卸载的模块从 modules 链表中移除，将模块占用的 CORE 区域空间释放，释放模块从用户空间接收的参数所占的空间等，函数的实现相对比较简单，这里就不再仔细讨论了。

1.4 本章小结

本章详细讨论了作为设备驱动程序的重要存在形式——内核模块幕后的技术细节，内核模块可以在系统运行期间动态扩展系统的功能，这是其最大的优势。在用户空间，加载和卸载模块使用的是一组称为 mod utils 的工具包，其中包括最基本的 insmod 和 rmmod 工具。

内核模块在文件格式上是一种可重定位的 ELF 文件，由 Linux 系统中的内核模块加载器负责加载和卸载。模块可以调用内核源码或者其他模块实现的函数，这需要模块的构造工具链和内核模块加载器共同协作才可以完成。为此，内核和内核模块通过 EXPORT_SYMBOL 宏向外导出符号，这些导出的符号可以被其他模块所使用，它们被放在一个特殊的 section 中，内核和内核模块拥有各自的 section 用来保存导出符号的信息。系统中所有成功加载的模块都以链表的形式存放在内核的一个全局变量 modules 中。

模块在编译时需要指定一个内核源码树，这种指定不是出于链接的需要，而是模块需要内核源码头文件中的一些定义，包括以头文件形式出现的内核配置信息。因此，一个.ko 文件总是基于某一特定内核源码树所构成，如果要在一个运行不同内核版本的系统中加载该.ko 文件，有可能会引起一些潜在问题。例如，随着内核版本的演进，老的.ko 文件所调用的一些内核函数在新版本的内核中可能消失或者改变了。为了防止这一问题的发生，内核引入了版本控制机制。如果内核模块以开放源码的形式向外发布，则版本不一致并不会成为一个问题，用户可以在新版本的内核上重新编译构造新的.ko 文件。