第 11 章 kerneld 和高级模块化

在本书的第二部分，我们要讨论的话题比到目前为止我们所接触过的话题都更为高级。我们将再次从模块化讲起，第二章“*编写和运行模块*”中对模块化的介绍只是其中的一部分；*modules* 包(它们的最新版本被称作 *modutils*)支持一些更高级的特性，它们比前面讨论的安装和运行一个基本的驱动程序所需的特性要更为复杂。

本章将讨论 *kerneld* 程序，模块中的版本支持(一种便利性，它使你在升级内核时不必重新编译你的各个模块)以及在卸载和重新装载一个模块时对数据持久性的支持。最后这项功能只有 2.0.0 版或更新版本的 *modules* 包才提供。

按需加载模块

为了方便用户加载和卸载模块，并且避免把不再使用的驱动程序继续保留在核心中浪费内核存储空间，Linux 提供了对模块的自动加载和卸载的支持。(在 1.2 版以前不提供这种支持)要利用这个特性，在编译内核前进行的配置中你必须打开对 *kerneld* 的支持。需要时可以请求附加模块的能力对于使用堆叠式模块的驱动程序尤其有用。

隐藏在*kerneld*之后的思想很简单，但却很有效。当内核试图访问不可用资源时，它会通知用户程序而不仅仅是返回一个错误。如果守护进程成功地获得该资源，内核将继续工作；否则它将返回错误。实际上申请任何一种资源的时候都可以使用这种办法：诸如字符设备和块设备驱动程序，行律和网络协议等等。

用于获得按需装载能力的机制是使用一个修改过的消息队列，利用它在内核空间和用户空间之间相互传递文本信息。要让按需装载能正确工作，必须正确地配置用户级守护进程，并且内核代码必须做好准备，等待所需的模块。

可以从按需装载中受益的驱动程序的一个典型例子是通用帧捕获者(frame-grabber)驱动程序。它能支持几种不同的外设，但却表现出相同的外部行为。发布中将包括它所支持的所有设备卡的代码，但是在运行时只有正在被使用的那个特定设备卡的代码才真正需要。这样开发者就能够把具体实现划分为一个定义软件接口的通用模块和一系列用于低层操作的与硬件相关的模块。在通用模块检测到系统中安装的捕获者的类型后，它就能够为该捕获者申请正确的模块。

用户级方面

*kerneld*程序生存在用户空间，负责处理来自内核的对新模块的请求。它通过创建自己的消息队列和内核相连，然后进入睡眠，等待请求。

请求一个模块时，守护进程从内核中接收一个字符串并试图解析它。这个字符串可能是

下面两种形式之一：

- 目标文件的名字，就象*insmod*命令的典型参数一样。**floppy**是这种名字的一个例子；在这种情况下，守护进程将查找文件*floppy.o*并装载它。

- 更一般的标志符，比如**block-major-2**，它用来指明主设备号为2的块设备－也就是软盘驱动程序。这种类型的字符串是最常见的，因为内核通常只知道资源的数字标志符。例如，当你试图使用一个块设备时，内核只知道它的主设备号；仅仅就为了能通过名字来请求每个块设备而为它们实现各自不同的钩子函数很浪费。

显然，后一种情况时，必须有某种方法把模块的"id"映射成它的实际名字。这种关联并不由*kerneld*本身完成而是由*kerneld*调用*modprobe*来完成。在*depmod*命令的帮助下，由*modprobe*来处理模块装载的细节；*kerneld*本身只负责与内核的通讯并生成外部任务。所有这些程序都在*modules*包中一起发布。*depmod*是一个能产生类似Makefile那样的模块依赖信息的工具，而*modprobe*是能替代*insmod*用来正确装载模块堆栈的程序。例如*ppp*模块堆叠在*slhc*模块(Serial Line Header Compression)之上(换句话说，可以使用*slhc*模块中的符号)。除非已经装载了*slhc*，否则命令*insmod ppp*就会失败；另一方面，假如在安装好模块之后会调用命令*depmod -a*来创建依赖规则，命令*modprobe ppp*就能成功。

*insmod*和*modprobe*间的另一个差别是后者不会在当前目录中查找模块，它只在*/lib/modules*下的缺省目录中查找。这是因为该程序是一个系统实用例程，而不是一个交互工具；你可以通过在*/etc/modules.conf*中指定你自己的目录，来把它们加入缺省目录集。

/etc/modules.conf 是一个用于定制 *modules* 包的文本文件。它负责把象 **block-major-2** 这样的名字关联到 **floppy**。注意，2.0 前的版本的 *modules* 包查找的是另一个文件 */etc/conf.modules*；出于兼容的考虑，仍支持这种文件名，但提倡更为标准的名字 *modules.conf*。

modules.conf 的语法在 *depmod* 和 *modprobe* 命令的 man 页中有很好的描述；然而，我觉得有必要在这里提及一些重要命令的意思。我用下面几行作为例子：

```
#sample line for /etc/modules.conf
keep
path[misc]=~rubini/driverBook/src/*
option short irq=1
alias eth0 ne
```

上面显示的第一行是注释；**path[misc]**指出在哪查找各种模块－而 **keep** 指出应把用户路径加到缺省路径中，而不是替换缺省路径。**Option** 制导(directive)指出在装载 short 模块时总是设定 **irq=1**，**alias** 行则指出当需要装载 **eth0** 时，相关的文件是 *ne.o*(ne2000 接口的驱动程序)。象 **alias block-major-2 floppy** 这样的行并不真正需要，因为 *modprobe* 已经知道的所有设备的官方主设备号，并且这些“可预见”的 **alias** 命令在程序中预定义过了。

那么，按需装载模块的正确安装，就是在文件*/etc/modules.conf* 中加入这么几行，因为 *kerneld* 是依靠 *modprobe* 来进行实际的装载操作。

内核级方面

请求加载模块和卸载模块，内核代码可以使用**<linux/kerneld.h>**中定义的函数。这些函数都定义成内联函数，实际上又将参数传递给了 *kerneld_send*。*kerneld_send* 函数是用来与 *kerneld* 通讯的一个灵活的引擎，它存在于文件 *ipc/msg.c* 中，如果你感兴趣的话，可以到那里浏览它。

这里，我不准备探讨 *kerneld_send* 的细节，因为在头文件**<linux/kerneld.h>**中定义的下列一些调用，足够你用来实现按需装载：

int request_module(const char *name)

需要加载模块的时候可以调用该函数。参数 **name** 或者是模块的文件名，或者是在用户空间解析的 id 类型字符串。在装载成功完成(或失败)后该函数返回。*request_module* 只能在进程上下文中被调用，因为当前进程将进入睡眠，等待模块被加载。任何一个按需加载的模块，在使用计数降为 0 时，都将自动卸载。

int release_module(const char *name，int waitflag)

请求立即卸载一个模块。如果 **waitflag** 不为 0，意味着函数在返回前必须等待卸载结束。如果 **waitflag** 为 0，函数可以在中断时间内调用－如果值得这么做的话。

int delayed_release_module(const char *name)

请求延迟的模块卸载。该函数总是立即返回。它的效果就是模块 **name** 在使用计数降为 0 就卸载，即使该模块并不是由 *kerneld* 加载的。

int cancel_release_module(const char *name)

该函数取消 *delayed_release_module* 的作用，它不阻止按需装载模块的自动卸载，最少当前的实现是这样的。一般不需要该函数，在这里提到它主要是出于完整性的考虑。

如果在内核空间检测到了错误，*kerneld_send* 的返回值，包括所有列出的这些函数的返回值都会是负的。如果内核中一切运行正常，返回值被置为执行这些操作的用户空间程序的退出值。成功时的退出值为 0，出错时为 1 到 255 间的一个数值。

有关 *kerneld_send* 的一个好消息就是即使在内核配置成不提供对 *kerneld* 的支持时，该函数仍然存在(并向模块开放)。因而，模块编写者总是可以调用上面显示的这些函数，但此时只返回**-ENOSYS**。当然，不能运行在 1.2 版的内核上，因为所有这些机制都是到 1.3.57 版才引入的。

现在，让我们实际地试着使用这些按需加载函数。为此目的，我们使用两个模块，分别叫作 *master* 和 *slave*，O'Reilly 的 FTP 站点上的 *misc-modules* 目录下以源文件的形式发布。我们还将使用 *slaveD.o* 来测试延迟卸载，并且使用 *slaveH.o* 来测试手工加载以及自动卸载模块。

为了不安装模块也可以运行测试代码，我在自己的*/etc/modules.conf* 文件中加入了如下

一些行：

```
keep
path[misc]=~rubini/driverBook/src/misc-modules
```

slave 模块只是一个空文件，而 *master* 模块看起来象下面这样：

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

int init_module(void)
{
int r[3];   /* 结果 */

r[0]=request_module("slave");
r[1]=request_module("slaveD");
r[2]=request_module("unexists");
printk("master: loading results are %i,%i,%i\n",r[0],r[1],r[2]);
return 0; /* 成功 */
}

void cleanup_module(void)
{
int r[4];/* results */

r[0]=release_module("slave",1/* wait */);
r[1]=release_module("slaveH",1 /* wait */);
r[2]=delayed_release_module("salveD");
r[3]=release_module("unexists",1 /* wait */);
printk("master: unloading results are %i,%i,%i,%i\n",r[0],r[1],r[2],r[3]);
}
```

在装载时，*master* 模块试着载入两个模块和一个并不存在的模块。除非你改变了终端的日志级别(loglevel)，否则 *printk* 消息将出现在终端上。下面是系统被配置成支持 *kerneld* 而该守护进程又是活动的时候，装载 *mater* 模块时的结果：

```
morgana.root# depmod -a
morgana.root# insmod master
master: loading results are 0,0,255
morgana.root# cat /proc/modules
slaveD          1          0 (autoclean)
slave           1          0 (autoclean)
master          1          0
isofs           5          1 (autoclean)
```

从 *request_module* 返回的值以及*/proc/modules* 文件（在第二章的“初始化和终止”一节中描述）均显示 slave 模块已经正确装载。另一方面，装载 unexists 的返回值 255 意味着用户程序失败，退出码是 255（或－1，因为它的长度为一个字节）。

我们简要的看看在卸载时会发生些什么，但在此之前先让我们手工加载 *slaveH*：

```
morgana.root# insmod slaveH
morgana.root# cat /proc/modules
slaveH          1          0
slaveD          1          0 (autoclean)
slave           1          0 (autoclean)
master          1          0
isofs           5          0 (autoclean)
morgana.root# rmmod master
master: unloading results are 0,0,0,255
morgana.root# cat /proc/modules
slaveD          1          0 (autoclean)
isofs           5          1 (autoclean)
morgana.root# sleep60;cat /proc/modules
isofs           5          1 (autoclean)
```

结果显示，除了 **unexists** 的卸载，一切都很正常，并且 *slaveD* 在一段时间过后也会被卸载。

尽管提供了各种例程，你还会发现，大部分时间 *request_module* 函数都能满足你的需要，而不要求你处理模块卸载；实际上，对不使用的模块，缺省地会自动进行卸载。绝大大部分时候，你甚至不必检查函数的返回值，因为只需要模块提供的一些函数。下面的实现比检查 *request_module* 的返回值更方便：

```
if ( (ptr = look_for_feature()) == NULL )     /*  是否没有该特性  */
request_module(modname);                  /*  试图装载它  */
if ( (ptr = look_for_feature()) == NULL )     /*  是否仍没有该特性  */
      return -ENODEV;                         /*  出错  */
```

模块中的版本控制

关于模块的一个主要问题是它们的版本相关性，在第二章的“版本相关性”一节中我曾经介绍过。针对每个要使用的版本的不同头文件都需要重新编译一次模块，当你运行好几个定制的模块时，这是件非常痛苦的事情。如果你运行的是以二进制形式发布的商业模块时，甚至连重新编译也是不可能的。

幸运的，内核开发者找到了一个变通的办法来处理版本问题。他们的想法是，只有改变了内核提供的软件接口，一个模块才不能兼容不同的内核版本。然后，软件接口可以由函数原型以及函数调用涉及到的所有数据结构的确切定义所表示。最后，可以使用一个CRC算法

把所有关于软件接口的信息映射到一个单一的 32 位数值[*]上去。

版本相关性的问题通过在每个由内核导出的符号的名字后面附加一个该符号相关信息的校验和来得到处理。解析头文件，就可以从中取出这些信息。这种便捷性是可选的，在编译的时候可以启动它们。

例如，当启动版本支持时，符号 **printk** 以类似 **printk_R12345678** 的形式向模块开放，这里 **12345678** 时函数使用的软件接口的检验和的 16 进制表示。加载一个模块到内核时，仅当加到内核中每个符号上的检验和匹配加到模块中相同符号上的校验和时，*insmod*(或 *modprobe*)才能够完成它的任务。

让我们来看看内核和模块都启动了版本支持的时候，会发生些什么:

- 内核本身并不修改符号。进程以通常的方式与内核链接，而且 *vmlinux* 文件的符号表看起来也和以前一样。
- 公共符号表使用版本名字创建，如*/proc/ksyms* 文件所示。
- 模块必须使用合并后的名字编译，这些名字在目标文件中是以未定义符号出现的。
- 装载程序用模块中未定义符号匹配内核中的公共符号，因此也要使用版本信息。

然而，上述情况只有当内核和模块都创建成支持版本化时才有效。如果有任何一方使用了原来的符号名，*insmod* 都会放弃版本信息，并试着用第二章的“版本相关性”一节中描述的方式来匹配模块声明的内核版本号和内核提供的版本号。

在模块中使用版本支持

当内核已经准备(可选的)输出版本化的符号时，模块源代码只需准备好支持该选项。可以在两处加入版本控制：在 *Makefile* 中或在源代码本身。因为 *modules* 包文档描述了在 *Makefile* 中如何做，我将向你显示在 C 源代码中如何做。用于演示 *kerneld* 如何工作的 *master* 模块能够支持版本化符号。如果用于编译模块的内核利用了版本支持的话，这种能力会自动启动。

用于合并符号名字的主要工具是头文件<linux/modversions.h>，它包括了所有公共内核符号的预处理定义。在包含这个头文件后，不管模块何时使用了内核符号，编译器都将看到合并了的版本。*modversions.h* 中的定义只有预先定义过 **MODVERSIONS** 才有效。

如果内核已经启动了版本支持，为了在模块中也启动它，我们必须保证在 **<linux/autoconf.h>**中已经定义过 **CONFIG_MODVERSIONS**。那个头文件控制着在当前内核中(编译时)启动了哪些特性。每个 **CONFIG_**宏定义声明相应选项的状态是否要激活。

---

[*] 实际上，CRC算法检测不到SMP和非SMP模块间的不兼容性，因为许多接口函数都是内联(**inline**)的，它们在SMP和非SMP机器上时不同编译的，即使它们对应了同样的检验和。你必须非常小心地避免混淆SMP模块和常规的模块。

这样，*master.c* 的初始化部分包含如下部分：

```
#include <linux/autoconf.h> /*  检索 CONFIG_*宏  */
#if defined(CONFIG_MODVERSION) && !defined(MODVERSIONS)
#     define MODVERSIONS    /*  强迫打开它  */
#endif

#ifdef MODVERSIONS
#     include <linux/modversions.h>
#endif
```

在版本化的内核上编译这个文件时，目标文件的符号表会引用版本化符号，这些版本化符号匹配内核本身开放的那些符号。下面的屏幕快照显示了 *master.o* 中储存的符号名字。在 *nm* 的输出中，"T"代表“文本(text)”，"D"代表“数据(data)”，"U"代表“未定义(undefined)”。最后一个标记表示目标文件引用了但没有声明的符号。

```
morgana% nm master.o
000000b0  T cleanup_module
00000000  T init_module
00000000  D kernel_version
          U kerneld_send_R7d428f45
          U printk_Rad1148ba
morgana% egrep 'printk|kerneld_send' /proc/ksyms
00131b40 kerneld_send_R7d428f45
0011234c printk_Rad1148ba
```

因为加到 *master.o* 中符号名上的校验和包含了与 *printk* 和 *kerneld_send* 相关的整个接口，模块与大部分内核版本都兼容。然而，如果与其中任一函数有关的数据结构被改变了，*insmod* 将因为模块与内核的不兼容而拒绝装载它。

开放版本化符号

以前的讨论中未涉及的情况是，当其它模块使用一个模块开放的符号时，会发生写什么。如果依赖版本信息来获得模块的可移植性，那么我们希望能把 CRC 校验码加到我们自己的符号上去。这个问题比仅仅链接到内核技巧性更高一些，因为我们需要将合并后的符号名向其它模块开放；我们需要一种办法来生成校验和。

分析头文件和生成校验和的任务是由随 *modules* 包一起发行的一个工具 *genksyms* 来做的。这个程序在自身的标准输入上接受 C 预处理器的输出，并在标准输出上打印出一个新的头文件。这个输出文件定义了原来那个源文件开放出来的每个符号的带检验和的版本。*genksyms* 的输出通常以后缀.*ver* 保存；下面我将遵循同样的惯例。

为了显示如何开放符号，我生成了两个名为 *export.c* 和 *import.c* 的虚构的模块。*export*

开放了一个名为 *export_function* 的简单函数，并且该函数会被第二个模块 *import.c* 使用。这个函数接收两个整数参数并返回它们的和－我们对这个函数并不感兴趣，而是对链接过程更感兴趣。

misc-modules 目录下的 *Makefile* 文件有从 *export.c* 生成 *export.ver* 文件的规则，因此 *export_function* 的检验和符号可以被 *import* 模块使用：

```
ifdef MODVERSIONS
export.o import.o: export.ver
endif

export.ver: export.c
	$(CC) -I$(INCLUDEDIR) -E -D__GENKSYMS__ $^|genksyms > $@
```

这几行演示了如何生成 *export.ver*，并且只有定义过了 **MODVERSIONS** 才会把它加到两个目标文件的依赖关系中去。如果内核启动了版本支持，还要添加几行到 *Makefile* 中负责定义 **MODVERSIONS**，但并不值得在这里展示它们。

然后，源文件必须为每个可能的预处理流程声明正确的预处理符号：不论是给 *genksyms* 的输入和真正编译过程，不论是启动还是关闭了版本支持。进一步，*export.c* 应当能够象 *master.c* 那样自动检测内核中的版本支持。下面几行向你显示了如何成功地做到这一点：

```
#ifndef EXPORT_SYMTAB
#	define EXPORT_SYMTAB	/* 需要这个定义是因为我们要开放符号*/
#endif

#include <linux/autoconf.h> /* 检索 CONFIG_* 宏 */
#if defined(CONFIG_MODVERSIONS)&& !defined(MODVERSIONS)
#	define MODVERSIONS
#endif

/*
 * 将内核符号和我们的符号的版本化定义包含进来，*除非*我们正在
 * 生成校验和(定义了*__GENKSYMS__)
 */
#if defined(MODVERSIONS) && !defined(__GENKSYMS__)
#	include <linux/modversions.h>
#	include "export.ver"	/* 为了包含 CRC，重定义了 "export_function" */
#endif
```

这些代码，虽然令人讨厌，但好处是：可以让 *Makefile* 处于一个干净的状态。另一方面，由 *make* 来传递正确的标志，涉及到为各种情况编写冗长的命令行，因此我就不在这里做了。

简单的 *import* 模块通过传递数字 2 和 2 作为参数，来调用 *export_function*；期望的结果

是 4。下面的例子显示 *import* 确实链接到了 *export* 的版本化符号，并且调用了函数。版本化符号出现在*/proc/ksyms* 文件中。

```
morgana.root# insmod export
morgana.root# grep export /proc/ksyms
0202d024        export_function_R2eb14c1e   (export)
morgana.root# insmod import
import my mate tells that 2+2 = 4
morgana.root# cat /proc/modules
import                  1               0
export                  3       [import]        0
```

跨过卸载/装载的持久存储

一旦我们装备上了 *kerneld* 和版本支持，就会发现使用模块比使用链进内核的驱动程序更方便。模块化只有一个问题：如果一个驱动程序由 *kerneld* 载入，然后被配置（通过 *ioctl* 或者其它方法），那么下次将该驱动程序载入内核时又必须重新配置它。而启动时的配置信息则可以在*/etc/modules.conf* 文件中一劳永逸的指定，因此当要多次使用按需装载时，运行时的配置变得容易丧失。用户会可能会失望地发现刚离开休息一会设备的配置信息就已经丢失了。我们需要的是一种可以在模块卸载后持久地保存相关信息的技术。

实际上，*modules* 包从 2.0.0 版开始(*modules-2.2.0*)提供这种能力。

真正的代码还没有集成进官方的内核，但很可能会被 Linus 的源码所接受。目前，为了启动对持久存储的支持，你需要使用 *modules* 包中发布的一个补丁；这个补丁在**<linux/kerneld.h>**中添加了几行代码。

实际上，隐藏在模块信息的持久存储之后的想法很直接：与用户空间相互传输信息，内核代码可以与转载和卸载模块使用同一个 *kerneld* 引擎。然后，守护进程使用一个通用数据库来管理信息存储。

在用户空间而不是在内核空间实现持久存储的原因是为了简化代码。尽管可以设计出仅与内核空间有关的实现，从内核空间访问一个数据库文件需要将库代码在不可交换的内核空间中复制，而在用户空间，库代码的使用则没有任何开销。

Kerneld 中提议的实现使用了 *gdbm* 库来实现数据库。也可以选择使用盘上数据库。如果使用了该数据库，就可以获得跨过系统启动的持久性存储；如果没有使用该数据库，你只能在 *kerneld* 进程的生存期内获得持久性。

下面这些函数是在头文件**<linux/kerneld.h>**中定义的，用于获得持久存储特性：

```
int set_persist(char *key, void *value, size_t length);
int get_persist(char *key, void *value, size_t length);
```

这些函数的参数是一个文本关键字（**key**）和数据项本身－该关键字唯一地标记数据库中的一个数据项，而数据项则呈现为一个指针和长度的熟悉形式。参数 **key** 在整个系统内都必须唯一。这样就允许每个模块通过在关键字前加上模块名字将自己的关键字分离出来，但是它也允许不同的模块共享配置变量，如果因为什么原因需要这么做的话。

可能的返回值和其它调用 *kerneld_send* 的函数是一样的：0 表示成功，负数通知一个内核空间的错误，而正数用于通知一个用户空间的错误。通常可以忽略返回值，因为如果有错的话，*get_persist* 不会修改 **value** 的值，而如果 *set_persist* 不能保存这个值的话，也不会做任何事情。

新近的 *kerneld* 守护进程开始支持这种新特性，所以模块也可以选择在内核中不对 *kerneld.h* 做修补而将 *set_persist* 和 *get_persist* 的定义包含进来。但要注意向前兼容。建议使用在 *modules* 中发布的补丁；在被官方的内核源代码中包含前持久存储的内部实现可能会有变化。

我们已经看到，使用持久存储的主要原因是避免每次把模块载入一个运行内核时又要重新配置它。这对按需装载的模块的运行时配置尤其重要；这对装载时配置也是一个有意义的选项，因为更新*/etc/modules.conf* 对普通用户来说有些复杂。

持久存储另一个可能的用处是跟踪系统的硬件配置以避免不必要的探测。探测硬件是一种冒险的操作。它可能会错误地配置了其它的硬件，特别是对 ISA 设备，因为 ISA 不象 PCI 那样提供了一种通用的方法来扫描系统总线。（第 15 章“*外设总线概貌*”详细讨论了该问题。）

下面的例子代码显示了一个名为 *psm*（Persistent Storage Module）的假想模块是如何避免不必要的探测的。为简化讨论，这个例子程序最多支持一个设备。

```
int psm_base = 0; /*  基本的 I/O 端口，在装载时可以设定  */

int init_module(void)
{
    if (psm_base==0){ /*  在装载时没有设定  */
        get_persist("psm_base", &psm_base, sizeof(int));
        if (psm_check_hw(psm_base)!=0)
            psm_base=0; /*  旧的数值不再有效：探测  */
    }
    else
        if (psm_check_hw(psm_base)!=0)
            return -ENODEV; /*  没有任何地方指明基地址  */

    if (psm_base==0)
        psm_base=psm_probe(); /*  返回基本端口，或者，如果没能找到就返回 0 */
    if (psm_base==0)
        return -ENODEV;/*  没有找到任何设备  */
```

```
        set_persist("psm_base", &psm_base, sizeof(int));/* 找到：保存它 */
    }
```

只有在装载时没有指定基本端口，并且以前的端口不再有效的时候，这些代码才探测硬件。如果找到一个设备，基本端口被保存起来，留作后用。

当驱动程序要支持多个设备时，检测新增加的硬件这个问题的一个可能的办法就是定义一个 **psm_newhw** 变量，如果添加了新设备到系统，那么用户可以在装载时对该变量进行设置。如果这样实现的话，那么当存在新设备时，用户必须使用 **insmod psm_newhw=1** 命令。如果 **psm_newhw** 不为 0，*init_module* 试着探测新设备，而在通常情况下它使用的是保存的信息。一个设备的基地址中的改变在上面给出的代码中已经处理过了，而不需要用户在装载时进行干预。

快速参考

本章引入下面一些内核符号：

/etc/modules.conf

modprobe 和 *depmod* 程序的配置文件。它用于配置按需加载，在这两个程序的 man 页中有描述。

#include <linux/kerneld.h>
int request_module(const char *name);
int release_module(const char *name, int waitflag);
int delayed_release_module(const char *name);
int cancel_release_module(const char *name);

这些函数通过 *kerneld* 守护进程进行模块的按需加载。

#include <linux/autoconf.h>
CONFIG_MODVERSIONS

只有当前内核被编译成支持版本化符号时这个宏才会被定义。

#ifdef MODVERSIONS
#include <linux/modversions.h>

这个头文件只有在 **CONFIG_MODVERSIONS** 有效时才存在，它包含了内核开放的所有符号的版本化名字。

EXPORT_SYMTAB

如果使用了版本支持并且你的模块使用了 *register_symtab* 来开放它自己的符号，必须定义这个宏。

__GENKSYMS__

当 *genksyms* 读入预处理文件来生成新的版本代码时，*make* 定义了这个宏。当生成新的

检验和时，该宏用于有条件的防止包含**<linux/modversions.h>**头文件。

int get_persist(char *key, void *value, size_t length);

int get_persist(char *key, void *value, size_t length);

对模块数据的永久性存储的支持依赖于这两个函数，它们在头文件**<linux/kerneld.h>**中定义。