第 2 章 编写和运行模块

非常高兴现在终于可以开始编程了。本章将介绍模块编程和内核编程所需的所有必要的概念。我们将要不多的篇幅来编写和运行一个完整的模块。这种专业技术（expertise）是编写如何模块化设备驱动程序的基础。为了避免一下子给你很多概念，本章仅介绍模块，不介绍任何类别的设备。

这里介绍的所有内核内容（函数，变量，头文件和宏）也将在本章最后的参考部分再次介绍。

如果你已经座不住了，下面的代码是一个完整的“Hello, World”模块（这个模块事实上并没什么功能）。它可以在Linux 2.0 或以上版本上编译通过，但不能低于或等于 1.2，关于这一点本章将在稍后的部分解释*。

（代码）

函数 *printk* 是由 Linux 内核定义的，功能与 *printf* 相似；模块可以调用 printk，这是因为在 *insmod* 加载了模块后，模块就被连编到内核中了，也就可以调用内核的符号了。字符串`<1>`是消息的优先级。我之所以在模块中使用了高优先级是因为，如果你使用的是内核 2.0.x 和旧的 *klogd* 守护进程，默认优先级的消息可能不能显示在控制台上（关于这个问题，你可以暂且忽略，我们将在第 4 章，“调试技术”，的“Printk”小节中详细解释）。

通过执行 *insmod* 和 *rmmod* 命令，你可以试试这个模块，其过程如下面的屏幕输出所示。注意，只有超级用户才能加载和卸载模块。

（代码）

正如你所见，编写一个模块很容易。通过本章我们将深入探讨这个内容。

模块与应用程序

在深入探讨模块之前，很有必要先看一看内核模块与应用程序之间的区别。

一个应用从头到尾完成一个任务，而模块则是为以后处理某些请求而注册自己，完成这个任务后它的“主”函数就立即中止了。换句话说就是，*init_module()*（模块的入口点）的任务就是为以后调用模块的函数做准备；这就好比模块在说，“我在这，这是我能做的。”模块的第二个入口点，*cleanup_module*，仅当模块被下载前才被调用。它应该跟内核说，“我不在这了，别再让我做任何事了。”能够卸载也许是你最喜爱的模块化的特性之一，它可以让你减少开发时间；你无需每次都花很长的时间开机关机就可以测试你的设备驱动程序。

作为一个程序员，你一定知道一个应用程序可以调用应用程序本身没有定义的函数：前后的连编过程可以用相应的函数库解析那些外部引用。*printf* 就是这样一个函数，它定义在 libc 中。然而，内核要仅能连编到内核中，它能调用的仅是由内核开放出来的那些函数。例如，上面的 *helllo.c* 中的 *printk* 函数就是内核版的 *printf*，并由内核开放给模块给使用；除了没有浮点支持外，它和原函数几乎一模一样。

如图 2-1 所示，它勾画了为了在运行的内核中加入新函数，是如何调用函数以及如何使用函数指针的。

由于没有库连接到模块中，源码文件不应该模块任何常规头文件。与内核有关的所有内容都定义在目录*/usr/include/linux* 和*/usr/include/asm* 下的头文件中。在编译应用程序也会间接使

---

* 正如第 1 章，“Linux内核简介”，中所述，这个例子和本书中的所有其他例子都可以从O’Reilly的FTP站点上下载。

用这些头文件；其中的内核代码通过`#ifdef __KERNEL__`保护起来。这两个内核头文件目录通常都是到内核源码所在位置的符号连接。如果你根本就想要整个内核源码，你至少还要这两个目录的头文件。在比较新的内核中，你还可以在内核源码中发现 net 和 scsi 头文件目录，但很少有模块会需要这两个目录。

内核头文件的作用将稍后需要它们的地方再做介绍，

内核模块与应用程序的另一个区别是，你得小心“名字空间污染”问题。程序员在写小程序时，往往不注意程序的名字空间，但当这些小程序成为大程序的一部分时就会造成许多问题了。名字空间污染是指当存在很多函数和全局变量时，它们的名字已不再富有足够的意义来很容易的区分彼此的问题。不得不处理这种应用程序的程序员必须花很大的精力来单单记住这些“保留”名，并为新符号寻找新的唯一的名字。如果在写内核代码时出现这样的错误，这对我们来说是无法忍受的，因为即便最小的模块也要连编到整个内核中。防止名字空间污染的最佳方法是把所有你自己的符号都声明为 `static` 的，而且给所有的全局量加一个 well-defined 前缀。此外，你还可以通过声明一个符号表来避免使用 `static` 声明，这些内容将在本章的“注册符号表”小节中介绍。即便是模块内的私有符号也最好使用选定的前缀，这样有时会减轻调试的工作。通常，内核中使用的前缀都是小写的，今后我们将贯彻这一约定。

内核编程和应用程序编程的最后一个区别是如何处理失效：在应用程序开发期间，段违例是无害的，利用调试器可以轻松地跟踪到引起问题的错误之处，然而内核失效却是致命的，如果不是整个系统，至少对于当前进程是这样的。我们将在第 4 章“调试系统失效”小节中介绍如何跟踪内核错误。

用户空间和内核空间

本节的讨论概而言之就是，模块是在所谓的“内核空间”中运行的，而应用程序则是在“用户空间”中运行的。这些都是操作系统理论的最基本概念。

事实上，操作系统的作用就是给程序提供一个计算机硬件的一致的视图。此外，操作系统处理程序的独立操作，并防止对资源的未经授权的访问。当且仅当 CPU 可以实现防止系统软件免受应用软件干扰的保护机制，这些不同寻常的工作才有可能实现。

每种现代处理器都能实现这种功能。人们选择的方案是在 CPU 内部实现不同的操作模式（或级）。不同的级有不同的作用，而且某些操作不允许在最低级使用；程序代码仅能通过有限数目的“门”从一个级切换到另一个级。Unix 系统就是充分利用这一硬件特性设计而成的，但它只使用了两级（与此不同，例如，Intel 处理器就有四级）。在 Unix 系统中，内核在最高级执行（也称为“管理员态”），在这一级任何操作就可以，而应用程序则执行在最低级（所谓的“用户态”），在这一级处理器禁止对硬件的直接访问和对内存的未授权访问。

正如前面所述，在谈到软件时，我们通常称执行态为“内核空间”和“用户空间”，它们分别引用不同的内存映射，也就是程序代码使用不同的“地址空间”。

Unix 通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。执行系统调用的内核代码在进程的上下文上执行——它代表调用进程操作而且可以访问进程地址空间的数据。但与此不同，处理中断的代码相对进程而言是异步的，而且与任何一个进程都无关。

模块的功能就是扩展内核的功能；运行在内核中的模块化的代码。通常，一个设备驱动程序完成上面概括的两个任务：模块的某些函数做为系统调用执行，而某些函数则负责处理中断。

内核中的并发

内核编程新手首先要问的问题之一就是多任务是如何管理的。事实上，除了调度器之外，关于多任务并没有什么可以多说的，而且调度器也超出了程序员的一般活动范围。你可能会遇到这些任务，除了掌握如下这些原则外，模块编写者无需了解多任务。

与串行的应用程序不同，内核是异步工作的，代表进程执行系统调用。内核负责输入/输出以及系统内对每一个进程的资源管理。

内核（和模块）函数完全在一个线程中执行，除非它们要“睡眠”，否则通常都是在单个进程的上下文中执行——设备驱动程序应该能够通过交织不同任务的执行来支持并发。例如，设备可能由两个不同的进程同时读取。设备驱动程序串行地响应若干 read 调用，每一个都属于不同的进程。由于代码需要区别不同的数据流，内核（以及设备驱动程序）必须维护内部数据结构以区分不同的操作。这与一个学生学习交织在一起的若干门课程并非不无相似之处：每门课都有一个不同的笔记本。解决多个访问问题的另一个方法就是避免它，禁止对设备的并发访问，但这种怠惰的技术根本不值的讨论。

当内核代码运行时，上下文切换不可能无意间发生，所以设备驱动程序无需是可重入的，除非它自己会调用 *schedule*。必须等待数据的函数可以调用 *sleep_on*，这个函数接着又调用 *schedule*。不过你必须要小心，存在某些函数会无意导致睡眠，特别是任何对用户空间的访问。利用“天然非抢占”特性不是什么好的方法。我将在第 5 章，“字符设备驱动程序的扩展操作”的“编写可重入代码”小节中讲解可重入函数。

就对设备驱动程序的多个访问而言，有许多不同的途径来分离这些不同的访问，但都是依赖于任务相关的数据。这种数据可以是全局内核变量或是传给设备驱动程序函数的进程相关参数。最重要的用来跟踪进程的全局变量是`current`：一个指向`struct task_struct`结构的指针，在`<linux/sched.h>`中定义。current指针指向当前正在运行的用户进程。在系统调用执行期间，如*open*或*read*，当前进程就是调用这个调用的进程[*]。如果需要的话，内核代码就可以利用`current`使用进程相关信息。第 5 章“设备文件的访问控制”小节中就有使用这种技术的例子。

编译器就象外部引用 printk 一样处理 `current`。模块可以在任何需要的地方引用 `current`，*insmod* 会在加载时解析出所有对它的引用。例如，如下语句通过访问 `struct task_struct` 中的某些域打印当前进程的进程 ID 和命令名：

（代码）

存储在 `current->comm` 中的命令名是当前进程最后执行的可执行文件的基名。

编译和加载

本章的剩下部分将介绍编写虽然是无类别但很完整的模块。就是说，模块不属于任何第 1 章“设备和模块的类别”中罗列的类别中的任何一个。本章中出现的设备驱动程序称为*skull*，是“Simple Kernel Utility for Loading Localities”的缩写。去掉这个模块提供的范例函数，你可以重用这个模块，向内核加载你自己的本地代码。[*]

在我们介绍 *init_module* 和 *cleanup_module* 的作用之前，首先让我们写一个 *Makefile* 来编译内核可以加载的目标代码。

---

[*] 在版本 2.0 中，为了支持SMP，`current`是一个宏，扩展为`current_set[this_cpu]`。优化了对`current`访问的 2.1.37 则将它的值存放在堆栈中，也就去掉了全局符号。

[*] 我这里使用了“本地”，它是指个人对系统的修改，套用了Unix古老而优秀的传统*/usr/local*。

首先，在包含任何头文件前，我们需要在预处理器中定义符号__KERNEL__。这个符号用于选择使用头文件的哪一部分。由于*libc*包含了这些头文件*，应用程序最终也会包含内核头文件，但应用程序不需要内核原型。于是就用__KERNEL__符号和#ifdef将那些额外的去掉。将内核符号和宏开放给用户空间的程序会造成那个程序的名字空间污染。如果你正在为一台SMP（对称多处理器）机器编译，你还需要在包含内核头文件前定义__SMP__。这一要求似乎有点不那么方便，但一旦开放人员找到达成SMP透明的正确方法，它就会逐渐消失的。另一个很重要的符号就是 MODULE，必须在包含<linux/module.h>前定义这个符号。除非要把设备驱动程序编译到内核映象中去，MODULE 应该总是定义了的。由于本书所涉及的驱动程序都不是直接连编到内核中去的，它们都定义了这个符号。

由于头文件中的函数都是声明为inline的，模块编写者还必须给编译器指定*-O*选项。*gcc*只有打开优化选项后才能扩展内嵌函数，不过它能同时接受*-g*和*-O*选项，这样你就可以调试那些内嵌函数的代码了*。

最后，为了防止发生令人不愉快的错误，我建议你使用*-Wall*（全面报警）编译选项，并且还要修改源码去除所有编译器给出的警告，即便这样做会改变你已有的编程风格，你也要这么做。

所有我目前介绍的定义和选项都在 *make* 使用的 CFLAGS 变量中。

除了一个合适的 CFLAGS 变量外，将要编写的 *Makefile* 还需要一个将不同目标文件连接在一起的规则。这条规则仅当一个模块被分成若干个不同的源文件时才需要，这种并非很不常见。通过命令 ld -r 将模块连接在一起，这条命令虽然调用了连接器，但并没有连编操作。这是因为输出还是一个目标文件，它是输入文件的混合。-r 选项的意思是“可重定位”； 输出文件是可重定位的，这是因为它尚未嵌入绝对地址。

下面的 Makefile 实现了上述的所有功能，它能建立由两个源文件组成的模块。如果你的模块是由一个源文件组成的，只要跳过包含 ld -r 的那项就可以了。

（代码）

上面文件中那个复杂的 install 规则将模块安装到一个版本相关的目录中，稍后将做解释。Makefile 中的变量 VER 是从<linux/version.h>中截取的版本号。

模块编好了，接下来必须把它加载到内核中。正如我前面所说，insmod 就是完成这个工作的。这个程序有点象 ld，它要将模块中未解析的符号连编到正在运行的内核的符号表中。但与连接器不同，它并不修改磁盘文件，而是修改内存映象。insmod 有很多命令选项（如果想知道细节，可以看 man），可以在模块连编到内核前修改模块中的整数值和字符串值。因此，如果一个模块设计得体，可以在加载时对其进行配置；加载时配置要比编译时配置更灵活，但不幸的是，有时候仍然有人使用后者。加载时配置将在本章的后面“自动和手动配置”小节中讲解。

感兴趣的读者可能想知道内核是怎样支持 insmod 的：它依赖于 kernel/modulec.c 中定义的几个系统调用。sys_create_module 为装载模块分配内存（这些内存是由 vmalloc 分配的，见第 7 章“获取内存”中的“vmalloc 及其同胞”一节），为了连编模块，系统调用 get_kernel_syms 返回内核符号表，sys_init_module 将可重定位目标码复制到内核空间并调用模块的初始化函数。

如果你看过了内核源码，你就会发现系统调用的名字都有 sys_前缀。所有系统调用都是这样，其他函数并没有这个约定；当你在源码中查找系统调用时，知道这一点会对你有所帮助。

---

* 对于版本 5 和以往的版本来说是这样的。对于版本 6（*glibc*）来说可能会发生变化，但在我写这本书时讨论尚未结束。

* 不过你要注意，使用任何超过-O2 的优化选项后，编译器可能会把源码中未声明为inline的函数也按内嵌处理，这样做是非常危险的。对于内核代码来说，某些函数被调用时需要有一个标准的堆栈框架，这种优化就会导致问题出现。

版本相关性

要时刻牢记，对于你想连编的每一个不同版本的内核，你的模块都要相应地编译一次。每个模块都定义了一个称为 `kernel_version` 的符号，*insmod* 检查这个符号是否与当前内核版本号匹配。较新的内核已在`<linux/module.h>`中替你定义了这个符号（这也就是为什么 *hello.c* 中没有对它的声明）。这也意味着，如果你的模块是由多个源文件组成的，你只能有一个源文件包含了<linux/module.h>。与此相反，当你在 Linux 1.2 下编译时，必须在你的源码中定义 kernel_version。

如果版本不匹配，而你仍然想在不同版本的内核里加载你的模块，可以在 insmod 命令中指定-f（“强制”）选项完成，但这个操作不安全，可能会失败。而且很难事先说明要发生那种情况。由于符号不匹配，加载就会失败，此时你会得到一个错误信息。内核内部的变化也会造成加载失败。如何这种情况发生了，你可能会在系统运行时得到一个非常严重的错误，很可能造成系统 panic——出于这个缘由，注意版本失配。事实上，通过内核里的“版本机制”更完美地解决版本失配问题（稍后，第 11 章“Kerneld 和高级模块化”的“模块内版本控制”小节将介绍这一更先进的内容）。

如果你需要为某个特定的内核编译模块，你必须在上面的 Makefile 中包含相应内核的头文件（例如，通过声明不同的 INCLUDEPATH）。

为了处理加载时的版本相关性，insmod 安装特定的路径查询：如果不能在当前目录找到模块，就在版本相关的目录中查找，如果还失败就在/lib/modules/misc 中查找。上面那个 Makefile 中的 install 规则就遵循了这一约定。

写一个可以在从 1.2.13 到 2.0.x 的任一版本的内核上编译的内核是件复杂的任务。模块化接口已经做了修改，配置越来越容易。你可以看到上面的那个 hello.c 中，只要你只处理较新的内核就什么都不用声明。与此不同，可移植的接口如下所示：

（代码）

在 2.0 或更新的内核中，module.h 包含了 version.h，而且，如果没有定义__NO_VERSION__，module.h 还定义了 kernel_version。

如果你需要将多个源文件连接在一起组成一个模块，而又有多个文件都需要包含<linux/module.h>——比如你需要 module.h 里声明的宏，就可以使用符号__NO_VERSION__。在包含 module.h 前定义__NO_VERSION__就可以在你不想要自动声明字符串 kernerl_version 的源文件里防止它的发生（ld -r 会对一个符号的多处定义报警）。本书中的模块就使用__NO_VERSION__达成这一目的。

其他基于内核版本的相关性可以通过预处理的条件编译解决——version.h定义了整数宏 LINUX_VERSION_CODE。这个宏展开后是内核版本的二进制表示，一个字节代表版本发行号的一部分。例如，1.3.5 的编码是 66309（即，0x10305）。*利用这个信息，你可以轻松地判断你正处理的是哪个版本的内核。

当你检查某个版本时，使用十进制表示是不方便的。为了在一个源文件里支持多个内核版本，我将用下面的宏通过版本号的 3 个部分构建版本编码：

（代码）

---

* 这样就可以在稳定版本间存在 256 之多的开发用版本。

内核符号表

我们已经知道 insmod 是如何利用公开内核符号来解析未定义符号的了。这张表包含了实现模块化设备驱动程序所需的全局内核项－－函数和变量。可以从文件/proc/ksyms 中以文本的方式读取这个公开符号表

当你的模块被加载时，你声明的任何全局符号都成为内核符号表的一部分，你可以从文件/proc/ksyms 或命令 ksyms 的结果了解这一点。

新模块可以使用你开放出来的符号，而且你在其他模块之上堆叠新模块。在主流的内核源码中也使用了这种模块堆叠的方法：msdos 文件系统依赖于 fat 模块开放出来的符号，而 ppp 驱动程序则堆叠在报头压缩模块上。

在处理复杂对象时，模块堆叠非常有用。如果以设备驱动程序的形式实现一个新的抽象，它可以提供一个设备相关的插接口。比如，帧缓冲视频驱动程序可以将符号开放给下层 VGA 驱动程序使用。每个用户都加载帧缓冲视频驱动程序，然后在根据自己安装的设备加载相应的 VGA 模块。

分层次的模块化简化了每一层的任务，大大缩减了开发时间。这同我们第 1 章中讨论的机制与策略分离很相似。

注册符号表

另一种开放你的模块中的全局符号的方法是使用函数 register_symtab，这个函数是符号表管理的正式接口。这里所涉及的编程接口适用于内核 1.2.13 和 2.0。如果想详细了解 2.1 开发用内核所做的变动，请参见第 17 章“最新发展”。

正如函数 register_symtab 的名字所暗示，它用来在内核主符号表中注册符号表。这种方法要比通过静态和全局变量的方法清晰的多，这样程序员就可以把关于哪些开放给其他模块，哪些不开放的信息集中存放。这种方法比在源文件中到处堆放 static 声明要好的多。

如果模块在初始化过程中调用了 register_symtab，全局变量就不再是开放的了；只有那些显式罗列在符号表中的符号才开放给内核。

填写一个符号表是项挺复杂的工作，但内核开发人员已经写好了头文件简化这项工作。下面若干行代码演示了如何声明和开放一个符号表：

（代码）

有兴趣的读者可以看看<linux/symtab_begin.h>，但它可是内核中最难懂的头文件之一。事实上，仅想好好使用宏 X 的话，根本没必要读董它。

由于 register_symtab 是在模块加载到内核后被调用的，它可以覆盖模块静态或全局声明的符号。此时，register_symtab 用显式符号表替代模块默认开放的公共符号。

这种覆盖是可能的，因为 insmod 命令处理传递给系统调用 sys_init_module 的全局符号表，然后在调用 init_module 之前注册这个符号表。因此这之后的任何一次显式调用 register_symtab 都会替换相应模块的符号表。

如果你的模块不需要开放任何符号，而且你也不想把所有的东西都声明成 static 的，在 init_module 里加上下面一行语句就可以了。这次对 register_symtab 的调用通过注册一个空表覆盖了模块默认的符号表：

（代码）

如果源文件不想给堆叠在其上的模块提供什么接口，用上面那行语句隐藏所有的符号总是不错的。

当模块从内核卸载时，它所声明的所有公共符号也就自动从主符号表中注销了。不过是全局符号还是显式符号表，这一点都适用。

初始化和终止

正如前面已述，init_module 向内核注册模块所能提供的所有设施。这里我使用了“设施”，我的意思是指新功能，是一整个设备驱动程序或新软件抽象，是一个可以由应用程序使用的新功能。

通过调用内核函数完成新设施的注册。传递的参数通常为一个指向描述这个新设施的数据结构和要注册的设施名称。这个数据结构通常会包含一些指向模块函数的指针，这就是模块体内的函数是被调用的机制。

除了用来标别模块类别（如字符和块设备驱动程序）的“主”设施之外，模块还可以注册如下项目：

其他设备

由于这类设施仅仅用于总线型鼠标，这些设备曾一度称为鼠标设备。它们都是些不完整的设备，通常要比那些功能健全的设备简单。

串行端口

可以在运行时向系统里加入串口设备驱动程序；这也是支持 PCMCIA 调治解调器的机制。

行律

行律是处理终端数据流的软件层。模块可以注册新行律，以非标准方式处理终端事务。例如，模块 kmouse 就使用行律从串口鼠标中偷取数据。

终端设备驱动程序

终端设备驱动程序一组实现终端底层数据处理的函数。控制台和串口设备驱动程序为了创建终端设备，它们都要注册自己的驱动程序。而多端口串口则有自己的驱动程序。

/proc 文件

/proc 包含了用来访问内核信息的文件。由于它们也可以用来调试，第 4 章的“使用/proc 文件系统”将讲解/proc 文件。

二进制文件格式

对于每个可执行文件，内核扫描“二进制文件格式”列表并按相应的格式执行它。模块可以实现新的格式，Java 模块就是这样做的。

Exec 域

为了提供与其他流行 Unix 系统的兼容，必须修改内核的某些内部表格。一个“执行域”就是一组从其他操作系统约定到 Linux 系统的映射。例如，模块可以定义执行 SCO 二进制文件的执行域。

符号表

这个已在前面的“注册符号表”小节中介绍了。

上面这些项目都不是前一章所考虑的设备类型，而且都支持那些通常集成到驱动程序功能中的设施，如/proc 文件和行律。之所以鼠标和其他设备驱动程序都没有象“完整”字符设备那样管理，这主要是为了方便。过一会儿，当你读到第 3 章“字符设备”的“主从设备号”小节时，原因就明了了。

还可以将模块注册为某些驱动程序的附件，但这样做就太特殊了，这里就不作讨论了；它们都使用了“注册符号表”中讲到的堆叠技术。如果你想做更深一步的探究，你可以在内核源码中查查 register_symtab，并且找找不同驱动程序的入口点。大部分注册函数都是以 register_

开始的，这样你就可以用“register_”在/proc/ksyms 找找它们了。

init_module 中的错误处理

如果你注册时发生什么错误，你必须取消失败前所有已完成的注册。例如，如果系统没有足够内存分配新数据结构时，可能会发生错误。尽管这不太可能，但确实会发生，好的程序代码必须为处理这类事件做好准备。

Linux 不为每个模块保留它都注册了那些设施，因此当 init_module 在某处失败时，模块必须统统收回。如果你在注销你已经注册的设施时失败了，内核就进入一种不稳定状态：卸载模块后，由于它们看起来仍然是“忙”的，你再也不能注册那些设施了，而且你也无法注销它们了，因为你必须使用你注册时的那个指针，而你不太可能得到那个指针了。恢复这种情况非常复杂，通常，重新启动是最好的解决方法。

我建议你用 goto 语句处理错误恢复。我讨厌使用 goto，但以我个人来看，这是一个它有所做为的地方（而且，是唯一的地方）。在内核里，通常都会象这里处理错误那样使用 goto。

下面这段样例在成功和失败时都能正确执行：

（代码）

返回值（err）是一个错误编码。在 Linux 内核里，错误编码是一个负值，在<linux/errno.h>中定义。如果你不使用其他函数返回的错误编码而要生成自己的，你应该包含<linux/errno.h>，这样就可以使用诸如-ENODEV，-ENOMEM 之类的符号值。总是返回相应的错误编码是种非常好的习惯，因为这样一来用户程序就利用 perror 或相似的方法把它们转换成有意义的字符串了。

很明显，cleanup_module 要取消所有 init_module 中完成的注册。

（代码）

使用计数

为了确定模块是否可以安全地卸载，系统为每个模块保留了一个使用计数。由于模块忙的时候是不能卸载模块的，系统需要这些信息：当文件系统还被安装在系统上时就不能删除这个文件系统类型，而且你也不能在还有程序使用某个字符设备时就去掉它。

通过 3 个宏来维护使用计数：

MOD_INC_USE_COUNT

　　当前模块计数加 1。

MOD_DEC_USE_COUNT

　　计数减 1。

MOD_IN_USE

　　计数非 0 时返回真。

这些宏都定义在<linux/module.h>中，它们都操作不该有程序员直接访问的内部数据结构。事实上，在 2.1 开发版本中，模块管理已经做了非常大的修改，并在 2.1.18 中进行了彻底重写（预知乡情，参见第 17 章的“模块化”小节）。

注意，cleanup_module 中无需检查 MOD_IN_USE，因为内核在调用清除函数前就已经在系统调用 sys_delete_module（在 kernel/module.c 中定义）中调用完成了检查。

如果忘了更新使用计数，你就不能再卸载模块了。在开发期间这种情况很可能发生，所以你一定要牢记。例如，如果进程因你的驱动程序引用了 NULL 指针而终止，驱动程序就不可

能区关闭设备，使用计数也就无法回复到 0。一种可能的解决方法就是在调试期间完全不使用使用计数，将 MOD_INC_USE_COUNT 和 MOD_DEC_USE_COUNT 重新定义为空操作。另一个解决方法就是利用其他方法将计数强制复位为 0（在第 5 章的“使用 ioctl 参数”小节中介绍）。在编写成品模块时，决不能投机取巧。然而在调试时期，有时候忽略一些问题可以节省时间，是可以接受的。
使用计数的当前值可以在/proc/modules 中每一项的第 3 个域中找到。这个文件显式系统中当前共加载了那些模块，每一项对应一个模块。其中的域包括，模块名，模块使用的页面数和当前使用计数。这是一个/proc/modules 样例：
（代码）
(autoclean)标志表明模块由 kerneld 管理（见第 11 章）。较新的内核中又加入了一些新的标志，除了一件事外，/proc/modules 的基本结构完全相同：在内核 2.1.18 和更新的版本中，长度用字节计而不是页面计。

卸载

要卸载一个模块就要使用 rmmod 命令。由于无需连编，它的任务远比加载简单。这个命令调用系统调用 delete_module，如果使用计数为 0 它又调用模块的 cleanup_module。
cleanup_module 实现负责注销所有由模块已经注册了的项目。只有符号表是自动删除的。

使用资源

模块不使用资源是无法完成自己的任务的，这些资源包括内存，I/O 端口和中断，如果你要用 DMA 控制器的话，还得有 DMA 通道。
做为一个程序员，你一定已经习惯了内存分配管理，在这方面编写内核代码没什么区别。你的程序使用 kmalloc 分配内存，使用 kfree 释放内存。除了 kmalloc 多一个参数，优先级，外，它们和 malloc，free 很相似。很多情况下，用优先级 GFP_KERNEL 就可以了。缩写 GFP 代表“Get Free Page（获取空闲页面）。”
与此不同，获取 I/O 端口和中断乍听起来怪怪的，因为程序员一般同用显式的指令访问它们，不必让操作系统了解这些。“分配”端口和中断与分配内存不同，因为内存是从一个资源池中分配，并且每个地址的行为是一样的；I/O 端口都各有自己的作用，而且驱动程序需要在特定的端口上工作，而不能随便使用某个端口。

端口

对于大多数驱动程序而言，它们的典型工作就是读写端口。不管是初始化还是正常工作的时候，它们都是这样的。为了避免其他驱动程序的干扰，必须保证设备驱动程序以独占方式访问端口－－如果一个模块探测因自己的硬件而写某个端口，而恰巧这个端口又是属于另一个设备的，这之后一定会发生点怪事。
为了防止不同设备间的干扰，Linux 的开发者决定实现端口的请求/释放机制。然而，未授权的对端口的访问并不会产生类似于“段失效”那样的错误－－硬件无法支持端口注册。
从文件/proc/ioports 可以以文本方式获得已注册的端口信息，就象下面的样子：
（代码）

文件中的每一项是有驱动程序锁定的范围（以十六进制表示）。在这些被释放前，其他驱动程序不允许访问这些端口。
避免冲突有两个途径。首先，向系统增加新设备的用户检查/proc/ioports，然后在配置新设备使用空闲端口——这种方法假设设备可以通过跳线进行配置。然后，当软件驱动程序初始化自己时，它能自动探测新设备而对其他设备无害：驱动程序不会探测已由其他驱动程序使用的 I/O 端口。
事实上，基于 I/O 注册的冲突避免对于模块化驱动程序很合适，但对于连编到内核里的驱动程序来说却可能失败。尽管我们不涉及这种驱动程序，但还是很是必要注意到，对于一个在启动时初始化自己的驱动程序来说，由于它要使用之后会被注册的端口，很可能会造成对其他设备的误配置。虽然如此，还是没有办法让一个符合规范的驱动程序与已配置好的硬件交互，除非以前加载的驱动程序不注册它的端口。基于以上原因，探测 ISA 设备是件很危险的事，而且如果随正式 Linux 内核发行的驱动程序为了因与尚未加载的模块对应的设备交互，拒绝在模块加载时执行探测功能。
设备探测的问题是因为只有一种方法标别设备，即通过写目标端口然后再读的方法——处理器（而且是任何程序）只能查看数据线上的电子信号。驱动程序编写者知道一旦设备连接到某个特定的端口上，它就会响应相应的查询代码。但是如果另一个设备连到了端口上，程序仍然会写这个设备，但天知道它会怎么响应这个异常的探测操作。有时可以通过读外设的 BIOS，查看一个已知的字串来避免端口探测；已有若干 SCSI 设备使用了这种技术，但并不是每个设备都要有自己的 BIOS。
一个符合规范的驱动程序应该调用 check_region 查看是否某个端口区域已由其他驱动程序锁定，之后就用 request_region 将端口锁住，当驱动程序不再使用端口时调用 release_region 释放端口。这些函数的原型在<linux/ioports.h>中。
注册端口的典型顺序如下所示（函数 skull_probe_hw 包含了所有设备相关代码，这里没有出现）：
（代码）
在 cleanup_module 里释放端口：
（代码）
系统也使用了一套类似的请求/释放策略维护中断，但注册/注销中断比处理端口复杂，整个过程的详细解释将放在第 9 章“中断处理”中介绍。
与前面讲到的关于设施的注册/注销相似，对资源的请求/释放方法也适合使用已勾勒的基于 goto 的实现框架。
对于编写 PCI 设备驱动程序的人来说，不存在这里所讲的探测问题。我将在第 15 章“外部总线简介”中介绍。

ISA 内存

本节技术性很强，如果你对处理硬件问题不是很有把握，可以简单跳过这节。
在 Intel 平台上，ISA 槽上的目标设备可能会提供片上内存，范围在 640KB 到 1MB 之间（0xA0000 到 0xFFFFF）；这也是设备驱动程序可以使用的一类资源。
这种内存部件反映了 8086 处理器那个时代，当时 8086 的寻址只有一兆的大小。PC 设计人员决定，低端的 640KB 当做 RAM，而保留另外的 384KB 用于 ROM 和内存映射设备。今天，即便是最强力的个人电脑也还有这个在第一兆字节里的空洞。Linux 的 PC 版保留了这片内存，根本不考虑使用它。本节给出的代码可以让你访问这个区域的内存，但它仅限于 x86 平台，而且 Linux 内核要至少是 2.0.x 的，x 是多少都可以。2.1 版改变了物理内存的访

问方式，比如，640KB-1MB 这段范围内的 I/O 内存就不能再这样访问。访问 I/O 内存的正确方式是第 18 章“硬件管理”“低 1M 内的 ISA 内存”小节中的内容，这超出了本章的范围。尽管内核提供了端口和中断的请求/释放机制，当前它还是没能提供给 I/O 内存类似的机制，所以你得自己做了。如果我能理解 Linus 是如何看待 PC 体系结构的化，这里给的方法就不会变化了。

有时某个驱动程序需要在初始化时探测 ISA 内存；例如，我需要告诉视频截取器（frame grabber）在哪映射截取的图象。问题是，如果没有探测方法，我将无法辨别那段范围内哪块内存正在使用。人们需要能够辨别 3 种不同的情况：映射了 RAM，有 ROM（例如，VGA BIOS），或者那段区域空闲。

skull 样例给出一种处理这些内存的方法，但由于 skull 和物理设备无关，它打印完 640KB-1MB 这段内存区域的信息后就退出了。然而，有必要谈一谈用于分析内存的代码，因为它必须处理一些竞争条件。竞争条件就是这样一种情形，两个任务可以竞争同一个资源，而且未同步的操作可能会损坏系统。

尽管驱动程序编写者无需处理多任务，我们还是必须记住，中断可能在你的代码中间发生，而且中断处理函数可能会不提醒你就修改全局量。尽管内核提供了许多工具处理竞争条件，下面给得出的简单规则阐述了处理这个问题的方法；对这个问题的彻底对策将在第 9 章的“竞争条件”小节中给出。

- 如果仅仅是读取共享的量，而不是写，将其声明为 volatile，要求编译器不对其进行优化。这样，编译好的代码在每次源码读取它时读取这个量了。
- 如果代码需要检查和修改这个值，必须在操作期间关闭中断，这样可以防止其他进程在我们检查过这个值后，但恰恰又在我们修改这个量之前修改这个量。

我们建议采用如下关闭中断的顺序：

（代码）

这里 cli 代表“clear interrupt flag（清除中断标志）”。上面出现的函数都定义在<asm/system.h>中。

应该避免使用经典的 cli 和 sti 序列，因为有时你无法在关闭中断前断定中断是否打开了。如果此时调用 sti 就是产生很不规则的错误出现，很难追踪这样的错误。

由于那段内存只能通过写物理内存和读取检查才能标别，而且如果测试期间有中断的化，有可能会被其他程序修改，因此检查 RAM 段的代码同时利用了 volatile 声明和 cli。下面的这段代码并不是很简单，如果一个设备正在象它的内存写数据，而这段代码又在扫描这段区域，它就会误认为这段区域是空闲区。好在这样的情况很少发生。

在下面的源代码中，每个 printk 都带有一个 KERN_INFO 前缀。这个符号拼接在格式字串前面做消息的优先级，它定义在<linux/kernel.h>中。这个符号展开后与本章开始的 hello.c 中使用的<1>字串很相似。

（代码）

如果你在探测时注意恢复你所修改的字节，探测内存不会造成与其他设备的冲突。*

作为一个细心的读者，你可能会知道在 15MB-16MB 地址域内的 ISA 内存是怎么回事。很不幸，那是个更棘手的问题，我们将在第 8 章的“1M 以上的 ISA 内存”小节中讨论。

---

* 注意，由于某些设备可能将I/O寄存器映射到内存地址上，在某些情况下向内存写数据会带来一些副作用。出于这种考虑以及其他一些考虑，最好不要在产品型驱动程序中使用这里给出的代码。不过它还是可以简单的介绍模块本身，放在这里还是合适的。

自动和手动配置

根据系统的不同，驱动程序需要了解的若干参数也会随之变化。例如，设备必须了解硬件的I/O 地址或内存区域。
注意，本节所讨论的大部分问题并不适用于 PCI 设备（第 15 章介绍）。
根据设备的不同，除了 I/O 地址外，还有一些其他参数会影响系统的驱动程序的行为，如设备的品牌和发行号。驱动程序为了正确地工作有必要了解这些参数的具体值。用正确的数值设置驱动程序（即，配置它）是一项需要在初始化期间完成的复杂的任务。
基本说来，有两种方式可以获得这些正确的数值：或者是用户显式地给出它们，或者是驱动程序自己探测。无疑，自动探测是最好的驱动程序配置方法，而用户配置则是最好实现的；作为驱动程序编写者的一种权衡，他应该尽可能地实现自动配置，但又允许用户配置作为一种可选的方式替代自动配置。这种配置方法的另一个好处就是，在开发期间可以给定参数，从而不用自动探测，可以在以后实现它。
insmod 在加载时接受命令行中给定的整数和字串值，可以给参数赋值。这条命令可以修改在模块中定义的全局变量。例如，如果你的源码中包含了这些变量：
（代码）
那么你就可以使用如下命令加载模块：
（代码）
例子里使用了 printk，它可以显式，当 init_module 被调用时，赋值已经发生了。注意，insmod 可以给任何整型或字符指针变量赋值，不管它们是否是公共符号表中的一部分。但对于声明为数组的串是不能在加载时赋值的，因为它已经在编译时解析出来了，以后就不能修改了。
自动配置可以设计为按如下方式工作：“如果配置变量是默认值，就执行自动探测；否则，保留当前值。”为了让这种方法可以工作，“默认”值应该不是任何用户可以在加载时设定的值。
下面这段代码给出了 skull 是如何自动探测设备的端口地址的。在这个例子中，使用自动探测查找多个设备，而手动配置只限于一个设备。注意，函数 skull_detect 在上面已经给出了，而 skull_init_board 负责完成设备相关的初始化工作，这里没有给出。
（代码）
为了方便用户在 insmod 命令行中给出相应的参数，而且如果这些符号不会放到主符号表中的话，实际使用的驱动程序可以去掉配置变量的前缀（在本例中就是 skull_）。如果它们确实要放到主符号表中，好的办法就是声明两个符号：一个没有前缀，在加载时赋值，一个有前缀，用 register_symtab 放到符号表中。

在用户空间编写驱动程序

到现在为止，一个首次接触内核问题的 Unix 程序员困难会对编写模块非常紧张。写一个用户程序直接读写设备端口可能会更容易些。
事实上，从某些方面看采用用户空间编程更好，而且有时写一个所谓的“用户空间设备驱动程序”是对内核扩充的明智抉择。
用户空间驱动程序的优点可以总结如下：

- 可以连编完整的 C 库。驱动程序不必寻求许多外部程序的帮助就能维持许多外部任务（实现使用策略的工具程序通常和驱动程序一起发行）。
- 可以用传统的调试器调试驱动程序代码，不必费很大的力气去调试运行中的内核。

- 如果用户空间的驱动程序挂起了，你只要简单地把它杀掉就可以了。驱动程序的问题不太可能将整个系统挂起，除非被控制的硬件真的误操作了。
- 与内核内存不同，用户内存是可以换页的。驱动程序很大但不经常使用的设备除了正在使用时外，不会占用其他程序很多的 RAM。
- 一个细心设计的驱动程序同样可以对设备进行并发访问。

用户空间驱动程序的一个例子就是 X 服务器：它确切地了解它可以操作什么硬件，什么不能，并且给所有的 X 客户提供图形资源。库 libsvga 也是一个类似的程序。

通常，用户空间驱动程序的编写者都实现一个“服务器”进程，取代内核完成“负责硬件控制的唯一代理”的任务。客户应用为了最终完成同设备的通信，可以连接这些服务器；一个灵巧的进程可以允许对设备的并发访问。X 服务器就是这样工作的。

用户空间驱动程序的另一个例子是 gpm 鼠标服务器：它完成鼠标设备在不同客户间的仲裁，以便让多个鼠标敏感的应用同时运行在不同的虚终端上。

但有时用户空间驱动程序只授权一个程序访问设备。libsvga 是按这种方式工作的。它连编到应用程序中，在不必依赖集中式设施（如，服务器）的情况就扩展了应用程序的能力。这种方法由于避免了通信代价通常会有更好的性能，但它要求进程以高特权用户的身份运行。

用户空间驱动设备的方法也有很多的缺点。其中最重要的有：

- 在用户空间无法使用中断。除非你学习使用新的 vm86 系统调用而且还要忍受一点性能代价，否则没有方法可以解决这个问题。
- 只能通过 mmap 映射/dev/mem 才能直接访问内存，但只有特权用户才能这样做。
- 只能通过调用 ioperm 或 iopl 才能直接访问 I/O 端口，但只有特权用户才能这样做。
- 由于客户和硬件间传递信息或动作需要一次上下文切换，响应时间很慢。
- 更糟的是，如果驱动程序被换到磁盘后，响应时间会长的不可接受。使用系统调用 mlock 也许会有所助益，但由于用户空间程序要依赖于很多库，一般情况下你都锁住很多页面。
- 很多重要的设备不能在用户空间实现，它们包括网络接口和快设备，但不仅限于此。

正如你所见，用户空间驱动程序毕竟做不了太多的事。但有意义的应用还是存在的：例如，支持 SCSI 扫描仪设备的程序。扫描仪应用程序使用了“通用 SCSI”内核驱动程序，它向用户空间程序开发了低级 SCSI 函数，这样那些程序就可以控制自己的硬件了。

为了写一个用户空间驱动程序，了解一些硬件知识就足够了，而且没有必要去了解内核软件的细节。本书不再对用户级驱动程序进行进一步的讨论，而集中于内核代码。

但另一方面，当处理一些特殊设备时，你可能需要先在用户空间写驱动软件。这样，你在不会挂起整个系统的前提下了解如何控制你的硬件。一旦你完成后，可以很轻松地将这些代码封装到内核模块中。

快速索引

本节总结我们本章所经涉及的内核函数，变量，宏以及/proc 文件。就是说本节可以当做一个参考。如果某项属于某个头文件的话，每一项都会罗列在相关的头文件之后。此后每章的后面都会有类似的一节，总结响应章中出现的新符号。

__KERNEL__

MODULE

预处理符号，编译模块化内核代码时两者都要定义。

```
int init_module(void);
void cleanup_module(void);
```

模块的入口点，必须在模块源码中定义。

#include <linux/module.h>

所需的头文件。模块源码必须包含这个头文件。

MOD_INC_USE_COUNT

MOD_DEC_USE_COUNT

MOD_IN_USE

操作使用计数的宏。

/proc/modules

当前已加载的模块列表。每一项包含模块名，它们占用的内存大小以及使用计数。附加的字串是声明模块当前是否活动的标志。

int register_symtab(struct symbol_table *);

用于指定模块中公共符号的函数。在 2.1.18 以更新的内核中，没有了这个函数。见第 17 章的“模块化”小节。

int register_symtab_from(struct symbol_table *, long *);

内核 2.0 开发了这个函数而不是 register_symtab，register_symtab 只是一个预处理宏。你可以在/proc/ksyms 中找到 register_symtab_from，但源代码无处与这个函数打交道。

#include <linux/symtab_begin.h>

X(symbol),

#include <linux/symtab_end.h>

用于声明符号表的头文件和预处理宏，用在内核 1.2 和 2.0 中。在 2.1.1 中符号表接口变更了。

#include <linux/version.h>

所需的头文件。除非定义了__NO_VERSION__，否则<linux/module.h>包含了这个头文件。

LINUX_VERSION_CODE

整数宏，对处理版本相关性的#ifdef 很有用。

char kernel_version[] = UTS_RELEASE;

每个模块所需的变量。除非定义了__NO_VERSION__，否则<linux/module.h>定义了这个变量。

__NO_VERSION__

预处理宏。防止<linux/module.h>声明 kernel_version。

#include <linux/sched.h>

最重要的头文件之一。没有它你很难做什么事。

struct task_struct *current;

当前进程。

struct task_struct *current_set[];

Linux 2.0 支持对称多处理，它将 current 定义为一个宏，扩展为 current_set[this_cpu]。你可以在/proc/ksyms 中找到 current_set，但模块代码仍然使用 current。2.1 开发用内核引入一个更快的无需开放内核符号的访问 current 的方式。见第 17 章的“其他变化”小节。

current->pid

current->comm

当前进程的进程 ID 和命令名。

#include <linux/kernel.h>

int printk(const char *fmt, …);

在内核代码中使用的与 printf 相似的函数。

#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);

在内核代码中使用的与 malloc 和 free 相似的函数。优先级一般采用 GFP_KERNEL。

#include <linux/ioport.h>
int check_region(unsigned int from, unsigned int extent);
void reguest_region(unsigned int from, unsigned int extent, const char *name);
void release_region(unsigned int from, unsigned int extent);

用于申请和释放 I/O 端口的函数。版本 2.1.30 将 unsigned int 参数变为 unsigned long，但这个变化不会影响驱动程序代码。

#include <asm/system.h>

定义诸如 save_flags 和 restore_flags 之类访问机器寄存器的宏的头文件。

save_flags(long flags);
restore_flags(long flags);

预处理宏，可以允许临时修改处理标志。

cli();
sti();

关闭和打开中断。不应该使用 sti；而要使用 save_flags 和 restore_flags。

/proc/ksyms

公共内核符号表。

/proc/ioports

已安装设备的端口列表。