第二章 建立和运行模块

2.1hello world模块

许多编程书籍从一个 "hello world" 例子开始, 作为一个展示可能的最简单的程序的方法. 本书涉及的是内核模块而不是程序; 因此, 对无耐心的读者, 下面的代码是一个完整的 "hello world"模块:

```c
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello, world\n");
        return 0;
}
static void hello_exit(void)
{

        printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

这个模块定义了两个函数，一个在模块被加载到内核时被调用（`hello_init`），另一个在模块被移除时被调用（`hello_exit`）。`module_init`和`module_exit`行使用特殊的内核宏来指示这两个函数的角色。另一个特殊的宏（`MODULE_LICENSE`）被用来告诉内核这个模块拥有一个免费的许可证；如果没有这样的声明，当模块被加载时，内核会发出警告。根据你的系统用来传递消息行的机制，你看到的输出可能会有所不同。特别是，前面的屏幕截图是从文本控制台获取的；如果你在窗口系统下运行的终端模拟器中运行 insmod 和 rmmod，你的屏幕上可能什么都看不到。消息会发送到系统日志文件之一，比如 /var/log/messages（实际的文件名在不同的 Linux 发行版之间可能会有所不同）。用于传递内核消息的机制将在第四章中进行描述。

如你所见，编写一个模块并不像你可能预期的那样困难——至少，只要模块不需要做任何有价值的事情。难的部分是理解你的设备以及如何最大化性能。我们在本章中深入讨论模块化，并将设备特定的问题留到后面的章节。

- `printk`函数在Linux内核中定义，并对模块可用；它的行为类似于标准C库函数printf。内核需要自己的打印函数，因为它自己运行，不需要C库的帮助。模块可以调用printk，因为在insmod加载它之后，模块被链接到内核，并可以访问内核的公共符号（函数和变量，如下一节详细说明）。字符串KERN_ALERT是消息的优先级。我们在这个模块中指定了一个高优先级，因为默认优先级的消息可能不会出现在任何有用的地方，这取决于你运行的内核版本，klogd守护进程的版本，以及你的配置。你现在可以忽略这个问题；我们在第4章中解释它。

- `printk` 是 Linux 内核中的一个函数，它的功能类似于标准 C 库中的 `printf` 函数，用于在内核日志中输出调试信息。由于内核运行在一个独立的环境中，没有 C 库的支持，因此需要自己的打印函数。

- `printk` 可以接受与 `printf` 相同的格式化字符串，但是它还有一些额外的功能。例如，它可以接受一个优先级参数（如 `KERN_ALERT`），这个参数决定了消息在系统日志中的重要性。这对于调试和系统故障排查非常有用。

- `dmesg`（display message或driver message）是一个在Unix和Unix-like操作系统中的命令行工具，用于输出内核的消息。当系统启动时，内核会产生大量的消息，这些消息包含了系统硬件和启动过程的信息，这些信息被保存在一个环形缓冲区中。

- `dmesg` 命令可以用来查看这个环形缓冲区中的消息，这对于诊断和解决系统启动问题，或者了解系统硬件配置非常有用。

- 例如，如果你想查看系统启动过程中的所有消息，你可以在终端中输入 `dmesg` 命令。如果你想查看关于特定硬件（比如USB）的消息，你可以使用 `dmesg | grep usb` 命令。

- `printk` 是 Linux 内核中的日志系统，它的输出通常会出现在以下几个地方：

1. 控制台：如果内核参数 `console` 设置了相应的设备（如 `console=ttyS0` 或 `console=tty0`），`printk` 的输出会被发送到这个设备。
2. `/proc/kmsg`：这是一个只读文件，包含了内核日志消息。通常，系统日志守护进程（如 `syslogd` 或 `rsyslogd`）会读取这个文件并将消息写入到 `/var/log/messages` 或其他日志文件。
3. `dmesg` 命令：`dmesg` 命令可以用来查看内核启动以来的消息。这些消息实际上是从 `/dev/kmsg` 设备读取的。

- 请注意，`printk` 的输出可能会受到内核日志级别（`/proc/sys/kernel/printk`）的影响。如果 `printk` 的日志级别低于内核日志级别，那么 `printk` 的输出可能不会被显示。
  printk函数的用法-CSDN博客
  (https://blog.csdn.net/wwwlyj123321/article/details/88422640)

2.2内核vs应用程序

在我们深入讨论之前，有必要强调一下内核模块和应用程序之间的各种差异。虽然大多数小型和中型应用程序从头到尾执行单一任务，但每个内核模块只是注册自己以便为未来的请求服务，其初始化函数立即结束。换句话说，模块的初始化函数的任务是为模块的函数的后续调用做准备；就好像模块在说，“我在这里，我能做这些事情。”模块的退出函数（在示例中是hello_exit）在模块卸载之前被调用。它应该告诉内核，“我不在那里了，不要再让我做任何事情。”这种编程方式类似于事件驱动编程，但是并非所有的应用程序都是事件驱动的，而每一个内核模块都是。

事件驱动应用程序和内核代码之间的另一个主要区别在于退出函数：虽然一个终止的应用程序可以在释放资源或避免清理时懒散，但模块的退出函数必须仔细地撤销初始化函数建立的所有内容，否则这些片段会一直留在系统中，直到系统重启。顺便说一下，卸载模块的能力是你最会欣赏的模块化特性之一，因为它有助于缩短开发时间；你可以测试你的新驱动程序的连续版本，而不需要每次都经历漫长的关机/重启周期。

在编程中，应用程序可以调用它没有定义的函数，这些函数通常定义在库中。例如，printf函数就是在libc库中定义的。当你在代码中调用printf函数时，编译器并不直接知道这个函数的具体实现，它只知道这个函数的声明。然后在链接阶段，链接器会在库中查找这个函数的具体实现，并将其链接到你的程序中，这样你的程序就可以使用这个函数了。

然而，对于内核模块来说，情况就不同了。内核模块只能链接到内核，也就是说，它只能调用内核导出的函数，不能调用其他库中的函数。这是因为内核模块是在内核空间运行的，而不是在用户空间，所以它不能直接访问用户空间的库。

例如，我们在hello.c模块中使用的printk函数，这是一个在内核中定义并导出给模块使用的函数，它的功能和printf类似，但有一些小的差异，最主要的差异就是它不支持浮点数。这是因为内核空间为了保持简洁和高效，通常不包含处理浮点数的代码。

图2-1展示了如何在模块中使用函数调用和函数指针来为正在运行的内核添加新功能。

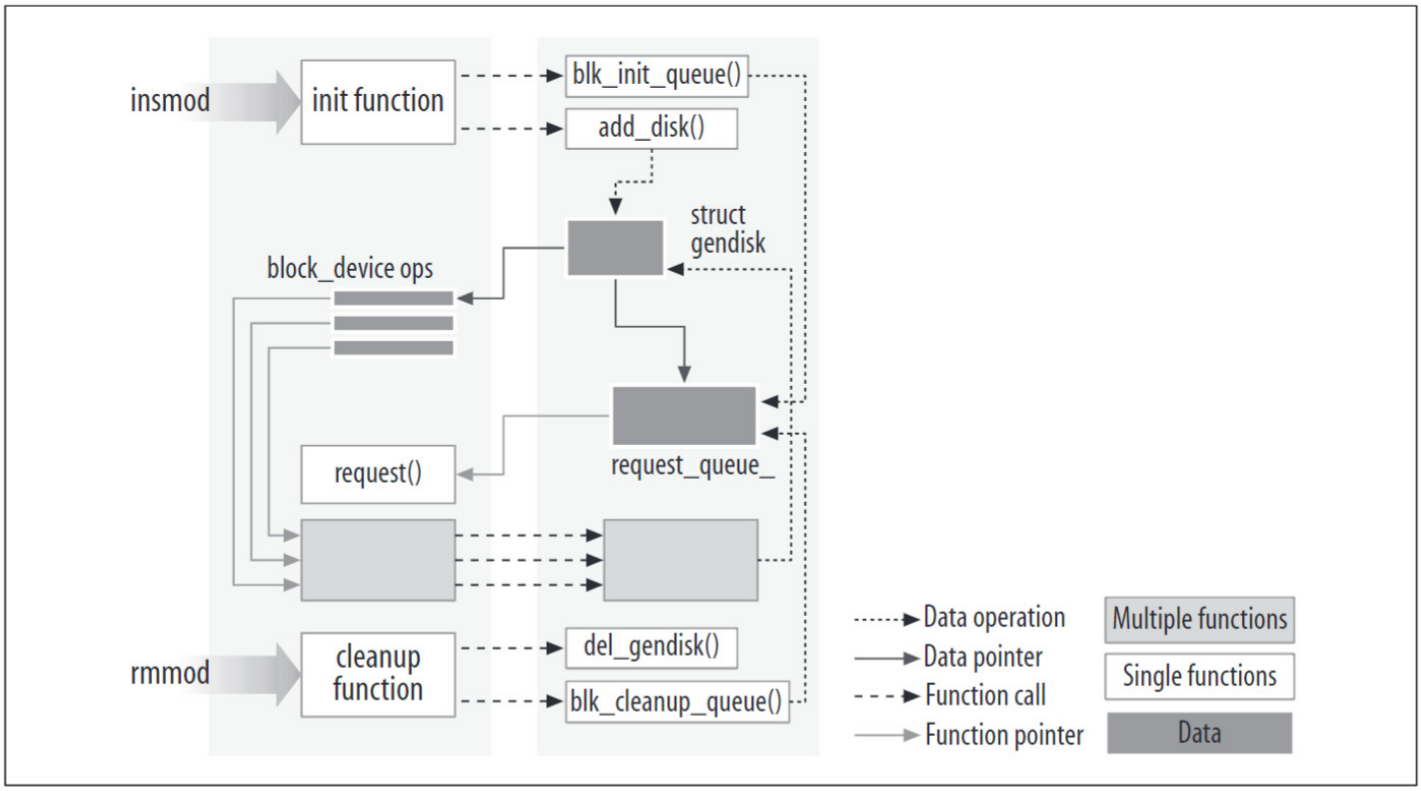

Figure 2-1. Linking a module to the kernel

因为没有库链接到模块，源文件永远不应该包含通常的头文件，<stdarg.h>和非常特殊的情况是唯一的例外。只有实际上是内核本身的一部分的函数才能在内核模块中使用。与内核相关的任何事情都在你设置和配置的内核源代码树中的头文件中声明；大多数相关的头文件位于include/linux和include/asm，但是其他的include子目录已经被添加以承载与特定内核子系统相关的材料。

本书将在需要每个头文件时介绍各个内核头文件的角色。

内核编程和应用程序编程之间的另一个重要区别在于每个环境如何处理错误：虽然在应用程序开发过程中分段错误是无害的，而且总是可以使用调试器来追踪源代码中的问题，但是内核错误至少会杀死当前的进程，如果不是整个系统的话。我们将在第4章中看到如何追踪内核错误。

2.3用户空间和内核空间

模块运行在内核空间，而应用程序运行在用户空间。这个概念是操作系统理论的基础。

操作系统的实际作用是为程序提供对计算机硬件的一致视图。此外，操作系统必须考虑程序的独立操作和防止未经授权的资源访问。只有当CPU强制保护系统软件免受应用程序的影响时，这个复杂的任务才可能实现。每个现代处理器都能强制执行这种行为。选择的方法是在CPU本身实现不同的操作模式（或级别）。这些级别有不同的角色，一些操作在较低级别是不允许的；程序代码只能通过有限数量的门从一个级别切换到另一个级别。Unix系统设计为利用这种硬件特性，使用两个这样的级别。所有当前的处理器至少有两个保护级别，一些像x86家族的处理器有更多的级别；当存在多个级别时，使用最高和最低级别。在Unix下，内

核在最高级别（也称为监督器模式）执行，其中允许所有操作，而应用程序在最低级别（所谓的用户模式）执行，处理器在这里规定了对硬件的直接访问和对内存的未经授权的访问。

我们通常将执行模式称为内核空间和用户空间。这些术语不仅包括两种模式中固有的不同权限级别，而且还包括每种模式都可以有自己的内存映射——自己的地址空间。

每当应用程序发出系统调用或被硬件中断挂起时，Unix都会将执行从用户空间转移到内核空间。执行系统调用的内核代码是在进程的上下文中工作的——它代表调用进程操作，并能够访问进程地址空间中的数据。另一方面，处理中断的代码与进程异步，并且与任何特定的进程无关。

模块的作用是扩展内核功能；模块化的代码运行在内核空间。通常，驱动程序会执行前面概述的两项任务：模块中的一些函数作为系统调用的一部分执行，一些则负责处理中断。

- a. 内核空间：这是操作系统内核的运行环境，具有最高的权限级别。在这个空间中，代码可以直接访问硬件资源，执行任何CPU指令，并且可以访问整个物理内存。操作系统的核心组件，如调度器、内存管理器、设备驱动等，都在内核空间运行。
- b. 用户空间：这是应用程序的运行环境，权限级别较低。用户空间的代码不能直接访问硬件或执行某些CPU指令，也不能直接访问其他进程的内存。如果需要访问硬件或其他资源，必须通过系统调用请求内核提供服务。

2.4内核的并发(concurrency)

内核编程与常规应用程序编程的一个主要区别是并发性问题。大多数应用程序（多线程应用程序是个例外）通常是顺序运行的，从头到尾，无需担心其他可能改变其环境的事情。但是，内核代码并不在这样简单的环境中运行，即使是最简单的内核模块也必须考虑到许多事情可能同时发生。

内核编程中的并发性来源有几个。自然，Linux系统运行多个进程，其中可能有多个进程试图同时使用你的驱动程序。大多数设备都能中断处理器；中断处理程序异步运行，可能在你的驱动程序试图做其他事情的同时被调用。几个软件抽象（比如在第7章中介绍的内核定时器）也是异步运行的。此外，Linux可以在对称多处理器（symmetric multiprocessor ,SMP）系统上运行，这意味着你的驱动程序可能在多个CPU上同时执行。最后，在2.6版本中，内核代码已经变得可抢占；这个改变使得即使是单处理器系统也有许多与多处理器系统相同的并发性问题。

- 当一个设备需要CPU的注意时，它会发送一个中断信号。这个中断信号会导致CPU暂停当前的任务，转而去执行一个特定的函数，这个函数就是中断处理程序。这个过程是异步的，也就是说，它可以在任何时间发生，不论CPU当前在做什么。
- 内核提供了一些软件抽象，比如内核定时器，这些抽象也是异步运行的。内核定时器可以在一定的时间后触发一个函数的执行。这个函数的执行是异步的，也就是说，它不会阻塞其他的任务，而是在适当的时间被调度执行。

Linux内核代码，包括驱动程序代码，必须能够在多个上下文中同时运行，这就是所谓的"可重入"。这意味着，如果一个函数在执行过程中被中断，并且这个中断处理程序又调用了这个函数，那么这个函数应该能够正确处理这种情况。

为了实现这一点，数据结构必须被设计得足够精细，以保证多个执行线程的独立性。同时，代码在访问共享数据时，必须采取防止数据损坏的方式，比如使用锁或者其他同步机制。

编写能够处理并发并避免竞态条件（执行顺序不幸导致不良行为的情况）的代码需要深思熟虑，可能会有些棘手。正确管理并发是编写正确的内核代码所必需的；因此，本书中的每一个示例驱动程序都是考虑到并发性编写的。我们会在遇到它们时解释使用的技术；第5章也专门讨论了这个问题和内核提供的用于并发管理的原语。

- 竞态条件是指，当多个线程访问和修改同一数据时，由于线程执行顺序的不确定性，可能会导致程序结果的不确定性。

驱动程序员常犯的一个错误是认为，只要代码不进入睡眠状态，就不需要考虑并发问题。但实际上，即使在非抢占式内核中，这个假设在多处理器系统上也是不成立的。在2.6版本的Linux内核中，内核代码几乎不能假设它可以在一段代码执行期间一直占有处理器。如果你的代码没有考虑并发，那么可能会导致非常难以调试的问题。

2.5当前进程

虽然内核模块的执行顺序并不像应用程序那样一步接一步，但是内核执行的大部分操作都是为特定的进程服务的。内核代码可以通过访问在<asm/current.h>中定义的全局变量current来获取当前进程的信息，这个变量是一个指向struct task_struct的指针，这个结构体在<linux/sched.h>中定义。current指针指向的就是当前正在执行的进程。

当执行系统调用，如open或read时，当前进程就是发起这个调用的进程。如果需要，内核代码可以通过使用current来获取和使用进程特定的信息。这种技术的示例将在第6章中展示。

实际上，current并不是真正意义上的全局变量。为了支持多处理器系统（SMP），内核开发人员开发了一种机制，这种机制可以在相关的CPU上找到当前的进程。这种机制必须要快，因为对current的引用是非常频繁的。这个机制通常是依赖于特定架构的，它在内核栈上隐藏了一个指向task_struct结构的指针。但是，这个实现的细节对其他内核子系统是隐藏的，设备驱动程序只需要包含<linux/sched.h>，然后就可以引用当前进程了。

```c
printk(KERN_INFO "The process is \"%s\" (pid %i)\n",
current->comm, current->pid);
```

2.6一些其他细节

内核编程与用户空间编程在许多方面都有所不同。我们会在整本书中逐一指出这些差异，但有一些基本问题，虽然不需要单独的章节来讨论，但值得一提。所以，当你深入研究内核时，应该记住以下几点。

应用程序在虚拟内存中布局，有一个非常大的栈区域。栈用于保存函数调用历史和所有当前活动函数创建的自动变量。相反，内核的栈非常小，可能只有一个4096字节的页面那么大。你的函数必须与整个内核空间调用链共享这个栈。因此，声明大的自动变量永远不是一个好主意；如果你需要更大的结构，应该在调用时动态分配它们。

通常，当你查看内核API时，会遇到以双下划线（__）开头的函数名。这样标记的函数通常是接口的低级组件，应谨慎使用。基本上，双下划线对程序员说："如果你调用这个函数，一定要知道你在做什么。"

- 在C语言和一些其他编程语言中，以双下划线（__）开头的函数名通常表示这是一个低级别或内部的函数。这种命名约定通常用于操作系统内核、库或其他低级编程环境中，以区分高级或公开的API和低级或内部的函数。

- 在Linux内核编程中，以双下划线开头的函数通常表示这是一个只应在特定上下文或条件下使用的函数。这些函数可能没有进行错误检查或其他保护措施，因此在调用它们时需要特别小心。基本上，双下划线告诉程序员："如果你要调用这个函数，你需要确保你知道你在做什么。"
  内核代码不能进行浮点运算。启用浮点运算将需要内核在每次进入和退出内核空间时保存和恢复浮点处理器的状态，至少在某些架构上是这样。鉴于内核代码实际上不需要浮点运算，额外的开销是不值得的。

2.7编译模块

我们首先需要了解模块的构建过程。模块的构建过程与用户空间应用程序的构建过程有很大的不同；内核是一个大型的、独立的程序，对其组件的组装方式有详细和明确的要求。构建过程也与以前版本的内核的操作方式不同；新的构建系统使用起来更简单，产生的结果更正确，但它看起来与以前的方式非常不同。内核构建系统是一个复杂的系统，我们只看其中的一小部分。内核源代码中的Documentation/kbuild目录中的文件是任何希望理解表面下真正发生的所有事情的人必读的。

在构建内核模块之前，你必须满足一些先决条件。首先，你需要确保你有足够新的版本的编译器、模块工具和其他必要的工具。内核文档目录中的Documentation/Changes文件总是列出了所需的工具版本；在进一步操作之前，你应该查阅它。试图用错误的工具版本来构建内核（及其模块）可能会导致无尽的微妙、困难的问题。注意，偶尔，编译器的版本过新可能和版本过旧一样有问题；内核源代码对编译器做了很多假设，新版本有时会暂时破坏一些东西。

如果你还没有内核树，或者还没有配置和构建那个内核，现在是时候去做了。如果你的文件系统上没有这个树，你就不能为2.6内核构建可加载模块。实际运行你正在为其构建的内核也是有帮助的（虽然不是必需的）。

一旦你设置好所有的东西，为你的模块创建一个makefile就很简单了。实际上，对于本章前面展示的“hello world”示例，一行代码就足够了：

```c
obj-m := hello.o
```

熟悉make，但不熟悉2.6内核构建系统的读者可能会想知道这个makefile是如何工作的。毕竟，上面的这行代码并不是传统makefile的样子。答案当然是，内核构建系统处理了其余的部分。上面的赋值（利用了GNU make提供的扩展语法）声明了有一个模块需要从hello.o这个对象文件构建。从对象文件构建后，生成的模块被命名为hello.ko。
`obj-m`是一个特殊的变量，用于指定要构建的模块。
我们要构建一个名为`hello`的模块，源代码文件是`hello.c`（编译后生成`hello.o`）。内核构建系统会处理剩下的部分，包括编译源代码、链接对象文件，最后生成`hello.ko`模块。注意这里不是makefile本身的语法,需要有内核构建系统的扩展。

- 内核构建系统是Linux内核源代码中的一部分，它负责编译和链接内核以及内核模块。它是一个基于Makefile的复杂系统，包含了大量的Makefile和脚本，用于处理内核和内核模块的构建过程。
- 内核构建系统的核心是顶级Makefile，位于内核源代码的根目录。此外，每个源代码目录都有自己的Makefile，用于编译该目录下的源代码。这些Makefile都被顶级Makefile包含，形成一个复杂的构建系统。
- 内核构建系统的主要任务包括：

1. 处理内核配置：内核构建系统允许用户选择要包含在内核中的特性和驱动程序。
2. 编译内核和内核模块：内核构建系统会编译内核源代码和选定的内核模块。
3. 链接内核：内核构建系统会链接编译后的对象文件，生成内核映像。
4. 安装内核和内核模块：内核构建系统可以将编译和链接后的内核和内核模块安装到适当的位置。

反之, 如果你有一个模块名为 module.ko, 是来自 2 个源文件(姑且称之为, file1.c 和 file2.c), 正确的书写应当是:

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

像上面显示的那样的makefile必须在更大的内核构建系统的上下文中被调用才能工作。如果你的内核源代码树位于，比如说，你的 `~/kernel-2.6` 目录，那么构建你的模块所需的make命令（在包含模块源代码和makefile的目录中输入）将是：

```c
make -C ~/kernel-2.6 M=`pwd` modules
```

这个命令首先改变它的目录到-C选项提供的目录（也就是你的内核源代码目录）。在那里，它找到内核的顶级makefile。M=选项使得makefile在尝试构建模块目标之前回到你的模块源代码目录。这个目标，反过来，引用了在obj-m变量中找到的模块列表，我们在我们的例子中将它设置为module.o。
这个命令是用来构建Linux内核模块的。让我们逐个解析这个命令的各个部分：

- `make`：这是GNU make工具，用于控制可执行文件的编译。
- `-C ~/kernel-2.6`： `-C` 选项告诉make改变到 `~/kernel-2.6` 目录。这个目录应该包含内核的顶级Makefile。
- `M=$(pwd)`： `M=` 是一个特殊的选项，用于指定模块的源代码目录。 `$(pwd)` 是一个shell命令，返回当前工作目录的路径。这个选项告诉make在构建模块之前先返回到当前目录。
- `modules`：这是要构建的目标。在这个上下文中， `modules` 目标表示要构建所有在Makefile中指定的模块。

所以，整个命令的意思是：“在 `~/kernel-2.6` 目录下运行make，构建在当前目录的Makefile中指定的所有模块。”

输入前面的make命令可能会让人感到疲倦，所以内核开发者们开发了一种makefile习语，使得在内核树外部构建模块的人的生活变得更加容易。技巧是按照以下方式编写你的makefile：

```c
# 如果定义了KERNELRELEASE，我们就是从内核构建系统调用的，可以使用它的语言。

ifneq ($(KERNELRELEASE),)

    obj-m := hello.o

# 否则我们是直接从命令行调用的；调用内核构建系统。

else

    KERNELDIR ?= /lib/modules/$(shell uname -r)/build

    PWD  := $(shell pwd)

default:

    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

再次，我们看到了扩展的GNU make语法在行动。在典型的构建中，这个makefile会被读取两次。当makefile从命令行调用时，它注意到KERNELRELEASE变量没有被设置。它通过利用安装模块目录中的符号链接build指向内核构建树的事实，找到内核源代码目录。如果你实际上并没有运行你正在为其构建的内核，你可以在命令行上提供一个KERNELDIR=选项，设置KERNELDIR环境变量，或者重写makefile中设置KERNELDIR的那一行。一旦找到了内核源代码树，makefile就调用default:目标，运行第二个make命令（在makefile中参数化为$(MAKE)）来调用内核构建系统，如前所述。在第二次读取时，makefile设置obj-m，内核makefiles负责实际构建模块。

- `?=` 操作符用于条件赋值。这意味着只有当变量之前没有被赋值时，才会对其进行赋值。
- `$(shell pwd)`：这是在Makefile中使用的语法。`$(shell command)`是GNU make的一个功能，它允许你在Makefile中执行shell命令并获取其输出。在这个例子中，`pwd`命令被执行，它返回当前工作目录的路径。

- `$(PWD)`：这是一个环境变量，它在大多数Unix-like系统（包括Linux）中都可用。在shell脚本或者Makefile中，你可以使用`$(PWD)`来获取当前工作目录的路径。在Makefile中，你也可以使用`$(PWD)`来获取Makefile所在的目录的路径。

- `` `pwd` ``：这是在shell脚本中使用的语法。在反引号`` ` ``中的命令会被执行，并且会被替换为命令的输出。在这个例子中，`pwd`命令被执行，它返回当前工作目录的路径。

- `uname -r`是一个Unix命令，用于打印系统的内核版本。在这个上下文中，`uname -r`命令被执行，并且返回的内核版本被用于构造内核源代码的路径。

- 需要注意的是，`default:`并不是一个特殊的目标名。在没有指定目标的情况下，`make`实际上会执行Makefile中的第一个目标。在许多Makefile中，你会看到名为`all:`的目标被放在第一个，作为默认目标。在这个Makefile中，`default:`目标恰好是第一个（也是唯一的）目标，所以它会被作为默认目标。

2.8加载和卸载模块

模块构建完成后，如何将其加载到内核中?这个过程主要由`insmod`程序完成。`insmod`是一个命令行工具，用于将模块插入到内核中。它将模块的代码和数据加载到内核的内存空间中。内核会将模块中的未解析符号链接到内核的符号表中。这是一个动态链接过程，类似于链接器`ld`的工作方式。内核并不修改模块的磁盘文件，而是修改内存中的副本,这是因为内核操作的是内存中的数据，而不是磁盘文件。这样可以提高效率，并且避免修改原始的模块文件。`insmod`可以在将模块链接到当前内核之前为模块中的参数赋值：这意味着你可以在加载模块时对其进行配置，这为用户提供了比编译时配置更大的灵活性。尽管有时仍会使用编译时配置。内核通过在`kernel/module.c`中定义的一个系统调用`sys_init_module`来支持`insmod`。`sys_init_module`函数会分配内核内存来保存一个模块，然后将模块文本复制到那个内存区域，通过内核符号表解析模块中的内核引用，并调用模块的初始化函数来启动所有操作。

系统调用的名称都带有`sys_`前缀：这是一个命名约定，所有的系统调用都遵循这个约定。这对所有系统调用都是如此，而其他函数则不是,这对于在源代码中搜索系统调用时非常有用。

`modprobe`是一个值得简要提及的工具。`modprobe`和`insmod`一样，都是用来将模块加载到内核中的。它们的不同之处在于，`modprobe`会检查要加载的模块是否引用了当前内核中未定义的任何符号。如果找到了这样的引用，`modprobe`会在当前模块搜索路径中查找定义了相关符号的其他模块。当`modprobe`找到这些模块（被加载的模块需要它们）时，它也会将它们加载到内核中。如果在这种情况下使用`insmod`，命令会失败，并在系统日志文件中留下“未解析的符号”消息。

如前所述，可以使用`rmmod`工具从内核中移除模块。请注意，如果内核认为模块仍在使用中（例如，某个程序仍然打开了由模块导出的设备的文件），或者内核已被配置为不允许移除

模块，那么模块移除将失败。可以配置内核允许“强制”移除模块，即使它们看起来正在忙碌。然而，如果你达到了需要使用这个选项的地步，那么可能已经出现了严重的问题，重启可能是更好的解决方案。

`lsmod`程序会生成一个当前加载到内核中的模块列表。它还提供了一些其他信息，如使用特定模块的其他模块等。`lsmod`通过读取`/proc/modules`虚拟文件来工作。当前加载的模块信息也可以在`/sys/module`下的sysfs虚拟文件系统中找到。

- `sysfs`是一个伪文件系统，它提供了一种从用户空间访问内核对象的方式，这些对象反映了设备模型的各个方面。`sysfs`在`/sys`目录下挂载，并为内核数据结构提供了一种组织方式。例如，你可以在`/sys/devices`下找到系统中所有设备的列表，或者在`/sys/class`下找到按类别（如网络设备、声音设备等）组织的设备列表。`sysfs`也用于一些用户空间工具，如`udev`，它用于管理设备节点，并可以用于获取设备的属性，如设备的制造商、型号等信息。

2.9版本依赖

请记住，你的模块代码必须为它链接的每个内核版本重新编译，至少在没有modversions的情况下是这样的，这里没有涵盖modversions，因为它们更多的是为发行版制作者而不是开发者服务的。模块与特定内核版本中定义的数据结构和函数原型紧密相关；从一个内核版本到下一个内核版本，模块看到的接口可能会发生显著变化。当然，这对开发内核尤其如此。

在构建内核模块时，内核不仅仅假设给定的模块已经针对正确的内核版本进行了构建。在构建过程中，会有一个步骤是将你的模块链接到当前内核树中的一个文件，这个文件被称为`vermagic.o`。

`vermagic.o`是一个对象文件，它包含了大量关于模块构建目标的内核的信息。这些信息包括目标内核的版本、用于编译模块的编译器的版本，以及一些重要的配置变量的设置。

这些信息在加载模块时非常重要，因为它们可以用来检查模块是否与正在运行的内核版本兼容。如果这些信息不匹配，模块将无法加载。这是一种保护机制，防止加载与当前内核版本不兼容的模块，这可能会导致系统不稳定或其他问题。当试图加载一个模块时，可以测试这些信息与正在运行的内核的兼容性。如果事情不匹配，模块就不会被加载；相反，你会看到类似以下的内容：

```c
# insmod hello.ko

Error inserting './hello.ko': -1 Invalid module format
```

查看系统日志文件（/var/log/messages或你的系统配置使用的其他文件）将揭示导致模块加载失败的具体问题。
Linux系统本身和大部分服务器程序的日志文件默认放在/var/log/下
日志文件	存放内容
/var/log/message	内核消息及各种应用程序的公共日志信息,包括启动,IO错误,网络错误
/var/log/cron	crond周期性计划任务产生的时间信息
/var/log/dmesg	引导过程中各种时间信息
/var/log/lastlog	每个用户最近登陆事件
/var/log/wtmp	每个用户登陆注销及系统启动事件
/var/log/btmp	失败的登陆尝试及验证事件
/var/log/secure	用户认证相关安全事件信息

如果你需要为特定的内核版本编译一个模块，你需要使用该特定版本的构建系统和源代码树。只需简单地更改前面示例中makefile的KERNELDIR变量即可。

内核接口在版本之间经常发生变化。如果你正在编写一个需要与多个内核版本（特别是必须跨主要版本）一起工作的模块，你可能需要使用宏和 `#ifdef` 构造来正确地构建你的代码。这本书的这个版本只关注一个主要的内核版本，所以你在我们的示例代码中不常看到版本测试。但是，偶尔还是需要它们的。在这种情况下，你需要使用在 `linux/version.h` 中找到的定义。

以下是一些与内核版本相关的宏：

- `UTS_RELEASE`：此宏展开为描述此内核树版本的字符串。例如，"2.6.10"。
- `LINUX_VERSION_CODE`：此宏展开为内核版本的二进制表示，每个版本发布号的一部分占一个字节。例如，2.6.10的代码是132618（即，0x02060a）。有了这个信息，你可以（几乎）轻松地确定你正在处理的内核版本。
- `KERNEL_VERSION(major,minor,release)`：这是用于从构成版本号的各个数字构建整数版本代码的宏。例如，`KERNEL_VERSION(2,6,10)` 展开为132618。当你需要比较当前版本和已知的检查点时，这个宏非常有用。

大多数基于内核版本的依赖性可以通过利用 `KERNEL_VERSION` 和 `LINUX_VERSION_CODE` 的预处理器条件来解决。然而，版本依赖性不应该用复杂的 `#ifdef` 条件语句来混乱驱动程序代码；处理不兼容性的最好方法是将它们限制在特定的头文件中。作为一般规则，明确依赖于版本（或平台）的代码应该隐藏在低级宏或函数后面。高级代码可以直接调用这些函数，而不用关心低级别的细节。这样编写的代码往往更易于阅读，也更稳健。

2.10平台依赖性

每种计算机平台都有其特性，内核设计者可以自由地利用所有这些特性来在目标对象文件中实现更好的性能。

与必须将其代码与预编译的库链接并坚持参数传递约定的应用程序开发者不同，内核开发者可以将一些处理器寄存器分配给特定的角色，他们已经这样做了。此外，内核代码可以针对CPU家族中的特定处理器进行优化，以从目标平台中获得最大的效益：与通常以二进制格式分发的应用程序不同，可以为特定的计算机集进行内核的自定义编译。

例如，IA32（x86）架构已经被细分为几种不同的处理器类型。尽管按照现代的标准，旧的80386处理器的指令集相当有限，但它仍然得到支持（至少现在还是）。这种架构中的更现代的处理器引入了许多新的功能，包括进入内核的更快的指令、处理器间锁定、复制数据等。当在正确的模式下操作时，新的处理器还可以使用36位（或更大）的物理地址，使它们能够寻址超过4GB的物理内存。其他处理器家族也有类似的改进。根据各种配置选项，内核可以被构建为使用这些额外的功能。

显然，如果一个模块要与给定的内核一起工作，它必须与构建该内核时对目标处理器的理解相同。再次，`vermagic.o`对象起到了作用。当加载一个模块时，内核会检查模块的处理器特定配置选项，并确保它们与正在运行的内核匹配。如果模块是用不同的选项编译的，它将不会被加载。

如果你打算为通用分发编写一个驱动程序，你可能会想知道你如何可能支持所有这些不同的变体。最好的答案，当然，是以GPL兼容的许可证发布你的驱动程序，并将其贡献给主线内核。如果不能做到这一点，以源代码形式分发你的驱动程序和一套在用户系统上编译它的脚本可能是最好的答案。一些供应商已经发布了工具来简化这个任务。如果你必须以二进制形式分发你的驱动程序，你需要查看你的目标分发版提供的不同内核，并为每一个提供一个模块版本。一定要考虑到自分发版制作以来可能已经发布的任何勘误内核。然后，我们在第1章的“许可条款”部分讨论过，还需要考虑许可问题。作为一般规则，以源代码形式分发东西是在世界上前进的更容易的方式。

2.11内核符号表

我们已经看到`insmod`如何将未定义的符号解析到公共内核符号表中。该表包含了实现模块化驱动程序所需的全局内核项（函数和变量）的地址。当一个模块被加载时，模块导出的任何符号都会成为内核符号表的一部分。在通常情况下，一个模块可以实现其自身的功能，而无需导出任何符号。然而，当其他模块可能从中受益时，你需要导出符号。

新的模块可以使用你的模块导出的符号，你可以在其他模块之上堆叠新的模块。模块堆叠也在主流内核源码中实现：msdos文件系统依赖于fat模块导出的符号，每个输入USB设备模块都堆叠在usbcore和输入模块之上。

模块堆叠在复杂的项目中很有用。如果一个新的抽象以设备驱动程序的形式实现，它可能会为硬件特定的实现提供一个插口。例如，video-for-linux驱动程序集被分割成一个通用模块，该模块导出由特定硬件的低级设备驱动程序使用的符号。根据你的设置，你加载通用的视频模块和你安装硬件的特定模块。并行端口和各种可附加设备的支持也是以同样的方式处

理的，USB内核子系统也是如此。并行端口子系统中的堆叠在图2-2中显示；箭头显示了模块之间以及与内核编程接口的通信。

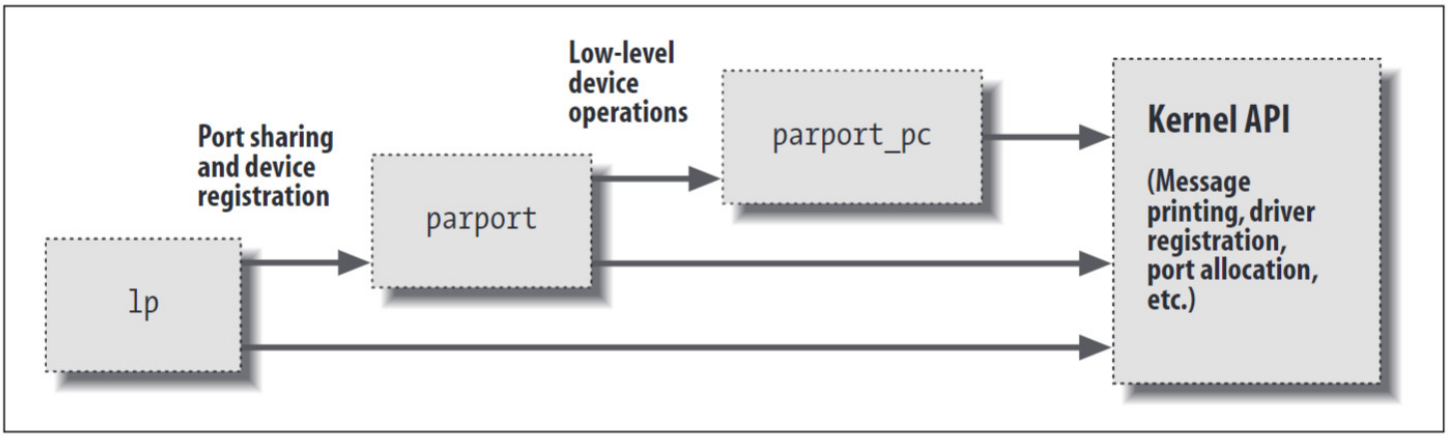

Figure 2-2. Stacking of parallel port driver modules

- 模块堆叠是Linux内核中的一个概念，它允许一个内核模块依赖于另一个内核模块的功能。这意味着一个模块可以使用另一个模块导出的符号（函数或变量）。这种依赖关系允许模块在其他模块之上“堆叠”，形成一个模块层次结构。
- 模块堆叠允许开发者在多个模块之间共享和重用代码。例如，一个通用的USB核心模块可以提供所有USB设备共享的基本功能，而特定的USB设备驱动模块则可以在此基础上添加特定设备的功能。通过模块堆叠，开发者可以将复杂的功能分解为一系列更小、更易于管理和维护的模块。每个模块都可以专注于实现一个特定的任务或功能。由于模块堆叠允许模块之间的依赖关系，因此可以在运行时动态地加载或卸载模块，以添加或删除功能，而无需重新编译或重启内核。模块堆叠允许开发者为特定的硬件设备创建特定的驱动模块，这些驱动模块可以在通用的硬件抽象模块之上堆叠。这使得内核可以在不同的硬件平台上运行，同时提供一致的编程接口。

在使用堆叠模块时，了解 `modprobe` 工具会很有帮助。正如我们之前描述的， `modprobe` 的功能与 `insmod` 大致相同，但它还会加载你想要加载的模块所需的任何其他模块。因此，有时一个 `modprobe` 命令可以替代多次调用 `insmod` （尽管当从当前目录加载你自己的模块时，你仍然需要 `insmod` ，因为 `modprobe` 只在标准的已安装模块目录中查找）。

使用堆叠将模块分割成多个层可以通过简化每一层来帮助减少开发时间。这类似于我们在第1章中讨论的机制和策略之间的分离。

- 在内核开发中，机制和策略的分离是一种设计原则，它鼓励将实现特定功能（机制）的代码与决定如何使用该功能（策略）的代码分开。这种分离可以提高代码的模块化程度，使得代码更易于理解、维护和重用。

- 模块堆叠在实现这种分离上起到了关键作用。具体来说，你可以创建一个提供特定机制（例如，访问硬件设备）的基础模块，然后在这个基础模块之上堆叠其他模块，这些模块实现了使用基础模块提供的机制的各种策略。

- 例如，你可能有一个通用的USB核心模块，它提供了访问USB设备的基本机制。然后，你可以为每种类型的USB设备创建一个单独的驱动模块，这些驱动模块实现了如何使用USB核心模块提供的机制来与特定类型的设备进行交互的策略。这样，USB核心模块和设备驱动模块就可以独立地开发和更新，而不会相互干扰。

Linux内核头文件提供了一种方便的方式来管理你的符号的可见性，从而减少命名空间污染（用可能与内核其他地方定义的名称冲突的名称填充命名空间）并促进适当的信息隐藏。如果你的模块需要为其他模块导出符号，应使用以下宏。

```c
EXPORT_SYMBOL(name);

EXPORT_SYMBOL_GPL(name);
```

在Linux内核模块编程中，`EXPORT_SYMBOL`和`EXPORT_SYMBOL_GPL`是两个用于导出模块符号的宏。这些符号可以是函数或变量，导出后，其他模块就可以使用这些符号。

`EXPORT_SYMBOL(name);`这个宏使得名为`name`的符号在模块外部可用，也就是说，其他模块可以链接到这个符号。

`EXPORT_SYMBOL_GPL(name);`这个宏的功能与`EXPORT_SYMBOL`相同，都是使得名为`name`的符号在模块外部可用。但是，`EXPORT_SYMBOL_GPL`只允许GPL许可的模块链接到这个符号。这是一种保护GPL许可的机制，确保只有同样遵循GPL许可的模块才能使用这个符号。

这些宏必须在模块文件的全局部分，即任何函数之外使用，因为这些宏会扩展为一个特殊的变量声明，这个变量需要在全局范围内可访问。这个特殊的变量会被存储在模块可执行文件的一个特殊部分，也就是“ELF段”。当内核加载模块时，会查找这个“ELF段”，以找到模块导出的所有符号。
（对此感兴趣的读者可以查看`<linux/module.h>`以获取详细信息，尽管不需要这些详细信息就可以使事情正常工作。）

- ELF（可执行和可链接格式）是一种常见的二进制文件格式，用于可执行文件、目标代码、共享库和内核模块等。ELF文件由一系列的段（segments）组成，每个段包含一种类型的数据。

- 在ELF文件中，段是文件的主要组织单位。每个段都包含一种类型的信息，例如程序代码、初始化数据、未初始化数据、符号表、重定位信息等。段的内容和布局取决于它的类型和用途。

- 在Linux内核模块编程中，当一个模块导出一个符号时，这个符号会被存储在一个特殊的ELF段中。当内核加载模块时，它会查找这个段，以找到模块导出的所有符号。

- 你可以使用`readelf`命令行工具查看ELF文件的段信息。例如，`readelf -S module.ko`命令会显示名为`module.ko`的内核模块文件的所有段的信息。

2.12预备知识

在我们查看实际的模块代码之前，我们需要先看一些需要出现在你的模块源文件中的其他东西。内核是一个独特的环境，它对与之接口的代码施加了自己的要求。

大多数内核代码最终都会包含相当多的头文件，以获取函数、数据类型和变量的定义。我们会在遇到这些文件时进行检查，但有一些是特定于模块的，必须出现在每个可加载的模块中。因此，几乎所有的模块代码都有以下内容：

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

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

`module.h`包含了许多可加载模块所需的符号和函数的定义。你需要`init.h`来指定你的初始化和清理函数，就像我们在上面的“hello world”示例中看到的那样，我们将在下一节中重新讨论这个问题。大多数模块还包括`moduleparam.h`，以便在加载时向模块传递参数；我们将很快讨论这个问题。

虽然不是严格必要的，但你的模块确实应该指定适用于其代码的许可证。做到这一点只是包含一个`MODULE_LICENSE`行的问题：

```c
MODULE_LICENSE("GPL");
```

内核认可的特定许可证包括“GPL”（适用于GNU通用公共许可证的任何版本）、“GPL v2”（仅适用于GPL版本2）、“GPL和附加权利”、“双BSD/GPL”、“双MPL/GPL”和“专有”。除非你的模块明确标记为内核认可的自由许可证，否则它被认为是专有的，当模块加载时，内核会被“污染”。正如我们在第1章的“许可条款”一节中提到的，内核开发者倾向于对在加载专有模块后遇到问题的用户提供帮助。

模块中可以包含的其他描述性定义包括`MODULE_AUTHOR`（声明谁写了模块）、`MODULE_DESCRIPTION`（模块做什么的人类可读声明）、`MODULE_VERSION`（用于代码

修订号；请参阅 <linux/module.h> 中的注释，了解在创建版本字符串时使用的约定）、MODULE_ALIAS（此模块可以被知道的另一个名称）和 MODULE_DEVICE_TABLE（告诉用户空间模块支持哪些设备）。我们将在第11章讨论 MODULE_ALIAS，在第12章讨论 MODULE_DEVICE_TABLE。

各种 MODULE_ 声明可以出现在源文件的函数外的任何地方。然而，内核代码中的一个相对较新的约定是将这些声明放在文件的末尾。
例如:

```c
#include <linux/module.h>
#include <linux/init.h>

static int __init my_module_init(void)
{
    printk(KERN_INFO "My module loaded\n");
    return 0;
}

static void __exit my_module_exit(void)
{
    printk(KERN_INFO "My module unloaded\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple example Linux module.");
MODULE_VERSION("0.01");
```

2.13初始化和关停

如前所述，模块初始化函数注册模块提供的任何设施。设施，我们指的是新的功能，无论是整个驱动程序还是新的软件抽象，都可以被应用程序访问。初始化函数的实际定义总是像这样:

```c
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

初始化函数应声明为静态的，因为它们不应在特定文件之外可见；然而，没有硬性规定，除非明确要求，否则没有函数会导出到内核的其余部分。定义中的`__init`标记可能看起来有点奇怪；它是对内核的一个提示，指定的函数只在初始化时使用。模块加载器在模块加载后丢弃初始化函数，使其内存可用于其他用途。有一个类似的标签（`__initdata`）用于只在初始化期间使用的数据。使用`__init`和`__initdata`是可选的，但它是值得的。只要确保不要在初始化完成后使用它们的任何函数（或数据结构）。你也可能在内核源码中遇到`__devinit`和`__devinitdata`；这些只有在内核没有为热插拔设备配置时，才会转换为`__init`和`__initdata`。我们将在第14章中查看热插拔支持。

使用`module_init`是必须的。这个宏在模块的对象代码中添加了一个特殊的部分，说明模块的初始化函数在哪里可以找到。没有这个定义，你的初始化函数永远不会被调用。

模块可以注册许多不同类型的设施，包括不同类型的设备、文件系统、加密转换等等。对于每个设施，都有一个特定的内核函数来完成这个注册。传递给内核注册函数的参数通常是指向描述新设施和正在注册的设施名称的数据结构的指针。数据结构通常包含指向模块函数的指针，这就是模块主体中的函数如何被调用的。

可以注册的项目超出了第1章中提到的设备类型列表。它们包括，但不限于，串行端口、杂项设备、sysfs条目、/proc文件、可执行域和行规则。许多可注册的项目支持的函数并不直接与硬件相关，但仍然在“软件抽象”领域。这些项目可以被注册，因为它们无论如何都被集成到驱动程序的功能中（例如/proc文件和行规则）。

- **sysfs条目**：sysfs是一个虚拟文件系统，它提供了一种方式来访问和改变内核对象的属性。这些对象包括设备、模块等。sysfs条目就是这些对象在sysfs文件系统中的表示。
- **/proc文件**：/proc文件系统是一个伪文件系统，它用于访问内核和进程的状态信息。它包含了一系列的文件和目录，这些文件和目录提供了一种方式来读取和改变内核参数。
- **可执行域**：在Linux内核中，可执行域是一个抽象，它定义了一种特定类型的二进制文件的执行方式。例如，Linux内核有一个用于执行ELF二进制文件的可执行域，还有一个用于执行脚本文件的可执行域。

- **行规则**：在Linux内核中，行规则是一种抽象，它定义了如何处理串行线路上的字符。例如，一个行规则可能会定义如何处理回车和换行字符，或者如何处理特殊的控制字符。

还有其他设施可以作为某些驱动程序的附加组件进行注册，但它们的使用如此特定，以至于没有必要讨论它们；它们使用堆叠技术，如“内核符号表”一节中所述。如果你想进一步探索，你可以在内核源码中搜索 `EXPORT_SYMBOL`，并找到不同驱动程序提供的入口点。大多数注册函数的前缀都是 `register_`，所以另一种可能的查找方式是在内核源码中搜索 `register_`。

2.13.1清理函数

每个非平凡的模块也需要一个清理函数，该函数在模块被移除之前注销接口并将所有资源返回给系统。这个函数定义如下：

```c
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
module_exit(cleanup_function);
```

清理函数没有值返回，所以它被声明为void。 `__exit` 修饰符将代码标记为仅用于模块卸载（通过使编译器将其放在一个特殊的ELF段中）。如果你的模块直接构建到内核中，或者你的内核配置为不允许卸载模块，那么标记为 `__exit` 的函数将被简单地丢弃。因此，一个标记为 `__exit` 的函数只能在模块卸载或系统关闭时被调用；任何其他用途都是错误的。再次， `module_exit` 声明是必要的，以便内核找到你的清理函数。

2.13.2初始化中的错误处理

当你向内核注册设施时，你必须始终记住的一件事是，注册可能会失败。即使是最简单的操作通常也需要内存分配，而所需的内存可能无法获得。因此，模块代码必须始终检查返回值，并确保请求的操作实际上已经成功。

当你注册实用程序时出现任何错误，首要任务是决定模块是否可以继续初始化自己。通常，模块可以在注册失败后继续运行，如果必要的话，可以降低功能。只要可能，你的模块应该

继续前进，并在事情失败后提供它能提供的能力。

如果事实证明你的模块在某种特定类型的失败后无法加载，你必须撤销在失败之前执行的任何注册活动。Linux不保留每个模块已注册的设施的注册表，所以如果在某个点初始化失败，模块必须自己退出所有事情。如果你没有注销你获得的东西，内核就会处于不稳定的状态；它包含指向不再存在的代码的内部指针。在这种情况下，通常唯一的补救方法是重启系统。当初始化错误发生时，你真的想要小心翼翼地做正确的事情。

错误恢复有时最好用goto语句来处理。我们通常不喜欢使用goto，但在我们看来，这是它有用的一种情况。在错误情况下小心使用goto可以消除大量复杂的、高度缩进的、“结构化”的逻辑。因此，在内核中，goto经常被用来处理错误.

以下示例代码（使用虚构的注册和注销函数）如果在任何点初始化失败都会正确地执行：

```c
int __init my_init_function(void)

{

    int err;

    /* registration takes a pointer and a name */

    err = register_this(ptr1, "skull");

    if (err) goto fail_this;

    err = register_that(ptr2, "skull");

    if (err) goto fail_that;

    err = register_those(ptr3, "skull");

    if (err) goto fail_those;

    return 0; /* success */

  fail_those: unregister_that(ptr2, "skull");

  fail_that: unregister_this(ptr1, "skull");

  fail_this: return err; /* propagate the error */

}
```

这段代码试图注册三个（虚构的）设施。如果失败，将使用goto语句只注销在事情变糟之前成功注册的设施。

另一个选项，不需要复杂的goto语句，就是跟踪已经成功注册的内容，并在出现任何错误时调用你的模块的清理函数。清理函数只撤销已经成功完成的步骤。然而，这种替代方案需要更多的代码和更多的CPU时间，所以在快速路径中，你仍然会将goto作为最好的错误恢复工具。

`my_init_function`的返回值`err`是一个错误代码。在Linux内核中，错误代码是负数，属于在`<linux/errno.h>`中定义的集合。如果你想生成你自己的错误代码，而不是返回你从其他函数得到的错误代码，你应该包含`<linux/errno.h>`，以便使用象`-ENODEV`、`-ENOMEM`等符号值。始终返回适当的错误代码是一种好的做法，因为用户程序可以使用`perror`或类似的方法将它们转换为有意义的字符串。

显然，模块清理函数必须撤销初始化函数执行的任何注册，而且通常（但通常不是强制的）按照注册它们的相反顺序注销设施：

```c
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

如果你的初始化和清理比处理几个项目更复杂，那么goto方法可能变得难以管理，因为所有的清理代码必须在初始化函数中重复，其中混杂着几个标签。因此，有时候，代码的不同布局可能更成功。

这段代码的主要目的是为了最小化代码重复并保持一切流畅。当发生错误时，从初始化中调用清理函数。然后，清理函数必须检查每个项目的状态，然后撤销其注册。

在最简单的形式中，代码如下所示：

```c
struct something *item1;
struct somethingelse *item2;
int stuff_ok;
void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff( );
    return;
}
int __init my_init(void)
{
    int err = -ENOMEM;
    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item1 || !item2)
        goto fail;
```

```
    err = register_stuff(item1, item2);
    if (!err)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* success */
  fail:
    my_cleanup( );
    return err;
}
```

如代码所示，你可能需要或不需要外部标志来标记初始化步骤的成功，这取决于你调用的注册/分配函数的语义。无论是否需要标志，这种初始化方式都能很好地扩展到大量的项目，通常比之前显示的技术更好。但是，请注意，当清理函数被非退出代码调用时，不能标记为__exit，就像前面的例子一样。

- `my_init` 函数是初始化函数，它试图分配两个项目并注册它们。如果在任何步骤中出现错误（例如，如果任何一个项目无法分配或注册失败），它将跳转到标签 `fail`，在那里它会调用 `my_cleanup` 函数并返回错误。

2.13.3模块加载竞争

模块加载中的一个重要问题：竞态条件。如果你在编写初始化函数时不小心，可能会产生一些可能影响整个系统稳定性的情况。我们将在后面的内容中讨论竞态条件，现在先简单说两点。

首先，你应该始终记住，内核的其他部分可以在你注册的任何设施注册完成后立即使用它。换句话说，内核可能在你的初始化函数仍在运行时调用你的模块。因此，你的代码必须准备好在完成第一次注册后立即被调用。在完成支持该设施所需的所有内部初始化之前，不要注册任何设施。

其次，你还必须考虑如果你的初始化函数决定失败，但内核的某部分已经在使用你的模块注册的设施会发生什么。如果你的模块可能出现这种情况，你应该认真考虑不让初始化失败。毕竟，模块显然已经成功地导出了一些有用的东西。如果初始化必须失败，它必须小心地避开内核其他地方正在进行的任何可能的操作，直到这些操作完成.

2.14模块参数

驱动程序需要知道的一些参数可能会因系统而异。这些可以从要使用的设备号（我们将在下一章中看到）到驱动程序应如何操作的许多方面。例如，SCSI适配器的驱动程序通常有控制标记命令队列使用的选项，而集成设备电子（IDE）驱动程序允许用户控制DMA操作。如果你的驱动程序控制的是旧硬件，它可能还需要明确地告诉它在哪里找到该硬件的I/O端口或I/O内存地址。内核通过使驱动程序可以指定在加载驱动程序的模块时可能更改的参数来支持这些需求。

这些参数值可以在加载时由insmod或modprobe分配；后者还可以从其配置文件（/etc/modprobe.conf）读取参数分配。这些命令接受在命令行上指定几种类型的值。作为演示这种能力的一种方式，想象一下对本章开头显示的“hello world”模块（称为hellop）的急需增强。我们添加了两个参数：一个名为howmany的整数值和一个名为whom的字符字符串。然后，我们的功能更强大的模块在加载时，不仅一次，而是howmany次向whom问好。然后，可以使用如下命令行加载这样的模块：

```c
insmod hellop howmany=10 whom="Mom"
```

以这种方式加载后，hellop会说10次“Hello, Mom”。

在insmod可以更改模块参数之前，模块必须使它们可用。参数是通过module_param宏声明的，该宏在moduleparam.h中定义。module_param接受三个参数：变量的名称，它的类型，以及用于伴随的sysfs条目的权限掩码。这个宏应该放在任何函数之外，通常在源文件的头部附近找到。所以hellop会声明它的参数并使它们对insmod可用，如下所示：

```c
static char *whom = "world";

static int howmany = 1;

module_param(howmany, int, S_IRUGO);

module_param(whom, charp, S_IRUGO);
```

支持许多类型的模块参数：

- bool / invbool：布尔值（真或假）（相关变量应为int类型）。invbool类型会反转值，因此真值变为假，反之亦然。
- charp：字符指针值。为用户提供的字符串分配内存，并相应地设置指针。
- int / long / short / uint / ulong / ushort：各种长度的基本整数值。以u开头的版本用于无符号值。

这样，当使用insmod命令加载模块时，可以通过命令行参数来设置这些变量的值。

模块加载器还支持数组参数，其中值作为逗号分隔的列表提供。要声明一个数组参数，使用：

```c
module_param_array(name,type,nump,perm);
```

其中，name是你的数组（和参数）的名称，type是数组元素的类型，nump是一个整数变量，perm是通常的权限值。如果在加载时设置了数组参数，nump被设置为提供的值的数量。模块加载器拒绝接受超过数组容量的值。

如果你真的需要一个在上面的列表中没有出现的类型，模块代码中有钩子允许你定义它们；查看moduleparam.h以获取如何做到这一点的详细信息。所有模块参数都应该给一个默认值；insmod只有在用户明确告知时才更改值。模块可以通过测试参数与其默认值来检查显式参数。

最后一个module_param字段是一个权限值；你应该使用在<linux/stat.h>中找到的定义。这个值控制谁可以访问sysfs中模块参数的表示。如果perm设置为0，那么根本没有sysfs条目；否则，它会在/sys/module下以给定的权限集出现。对于一个可以被世界读取但不能被更改的参数，使用S_IRUGO；S_IRUGO|S_IWUSR允许root更改参数。注意，如果一个参数

通过sysfs被更改，你的模块看到的那个参数的值会改变，但你的模块不会以任何其他方式得到通知。除非你准备好检测更改并相应地做出反应，否则你可能不应该使模块参数可写。

2.15在用户空间中做的事情

对于第一次处理内核问题的Unix程序员来说，编写一个模块可能会感到紧张。编写一个直接读写设备端口的用户程序可能会更容易。

确实，有一些支持用户空间编程的论点，有时编写所谓的用户空间设备驱动程序是对内核黑客的明智选择。在这一节中，我们讨论了你可能为什么要在用户空间中编写驱动程序的一些原因。然而，这本书是关于内核空间驱动程序的，所以我们不会超出这个介绍性的讨论。

用户空间驱动程序的优点包括：

- 可以链接完整的C库。驱动程序可以执行许多异乎寻常的任务，而无需求助于外部程序（通常与驱动程序一起分发的实现使用策略的实用程序）。
- 程序员可以在驱动程序代码上运行常规的调试器，而无需通过扭曲来调试正在运行的内核。
- 如果用户空间驱动程序挂起，你可以简单地杀掉它。除非被控制的硬件真的出现问题，否则驱动程序的问题不太可能挂起整个系统。
- 用户内存是可交换的，与内核内存不同。一个不常用的设备有一个巨大的驱动程序，除非它实际上在使用，否则不会占用其他程序可能使用的RAM。
- 一个设计良好的驱动程序程序仍然可以，像内核空间驱动程序一样，允许并发访问设备。
- 如果你必须编写一个闭源驱动程序，用户空间选项使你更容易避免模糊的许可证情况和与改变内核接口的问题。

例如，USB驱动程序可以为用户空间编写；参见libusb.sourceforge.net的（still young）libusb项目和内核源代码中的“gadgetfs”。另一个例子是X服务器：它知道硬件能做什么，不能做什么，并且它向所有X客户端提供图形资源。然而，请注意，有一个缓慢但稳定的趋势，即向基于帧缓冲的图形环境漂移，在这种环境中，X服务器仅作为一个基于实际内核空间设备驱动程序进行实际图形操作的服务器。

通常，用户空间驱动程序的编写者实现一个服务器进程，接管内核的任务，成为负责硬件控制的唯一代理。然后，客户端应用程序可以连接到服务器来执行与设备的实际通信；因此，一个智能的驱动程序进程可以允许并发访问设备。这就是X服务器的工作方式。

通常，用户空间驱动程序的编写者实现一个服务器进程，接管内核的任务，成为负责硬件控制的唯一代理。然后，客户端应用程序可以连接到服务器来执行与设备的实际通信；因此，一个智能的驱动程序进程可以允许并发访问设备。这就是X服务器的工作方式。

但是，设备驱动的用户空间方法有许多缺点。最重要的是：

- 用户空间中没有中断。在一些平台上，有一些解决这个限制的方法，比如IA32架构上的vm86系统调用。
- 只有通过映射/dev/mem才能直接访问内存，而且只有特权用户才能做到这一点。
- 只有在调用ioperm或iopl后才能访问I/O端口。此外，并非所有平台都支持这些系统调用，而且访问/dev/port可能会太慢而无法有效。这两个系统调用和设备文件都是特权用户的专属。
- 响应时间更慢，因为需要上下文切换来在客户端和硬件之间传输信息或操作。
- 更糟糕的是，如果驱动程序已经被交换到磁盘，响应时间就会无法接受地长。使用mlock系统调用可能会有所帮助，但通常你需要锁定许多内存页面，因为一个用户空间程序依赖于大量的库代码。mlock也只限于特权用户。
- 最重要的设备不能在用户空间中处理，包括但不限于网络接口和块设备。

如你所见，用户空间驱动程序毕竟不能做那么多事情。然而，仍然存在一些有趣的应用：例如，对SCSI扫描设备的支持（由SANE包实现）和CD写入器（由cdrecord和其他工具实现）。在这两种情况下，用户级设备驱动程序都依赖于“SCSI通用”内核驱动程序，该驱动程序将低级SCSI功能导出到用户空间程序，以便它们可以驱动自己的硬件。

当你开始处理新的和不寻常的硬件时，在用户空间工作可能是有意义的。这样，你可以学习管理你的硬件，而不用冒挂起整个系统的风险。一旦你做到了这一点，将软件封装在一个内核模块中应该是一个无痛的操作。

2.16参考

这一部分总结了我们在本章中涉及的内核函数、变量、宏和/proc文件。它的目的是作为一个参考。每个项目在相关的头文件后面列出（如果有的话）。几乎每一章的最后都有一个类似的部分，总结了在该章中引入的新符号。本节中的条目通常按照它们在章节中引入的顺序出现：

```
insmod modprobe rmmod
```

- 用户空间的实用程序，将模块加载到正在运行的内核并移除它们。

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

module_init(init_function);

module_exit(cleanup_function);
```

- 宏，用于指定模块的初始化和清理函数。

`__init __initdata __exit __exitdata` 标记函数（**init**和exit）和数据（**initdata**和exitdata）只在模块初始化或清理时使用。标记为初始化的项目在初始化完成后可能被丢

弃；如果未在内核中配置模块卸载，退出项目可能被丢弃。这些标记通过使相关对象放置在可执行文件的一个特殊ELF段中来工作。

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

- 最重要的头文件之一。这个文件包含了驱动程序使用的大部分内核API的定义，包括用于睡眠的函数和许多变量声明。

```
struct task_struct *current;
```

- 当前进程。

```
current->pid
```

```
current->comm
```

- 当前进程的进程ID和命令名。
- `obj-m` 内核构建系统使用的makefile符号，用于确定当前目录中应构建哪些模块。
- /sys/module /proc/modules /sys/module是一个sysfs目录层次结构，包含当前加载的模块的信息。/proc/modules是该信息的旧版，单文件版本。条目包含模块名、每个模块占用的内存量和使用计数。每行后面附加了额外的字符串，以指定当前对模块活动的标志。
- `vermagic.o` 来自内核源目录的一个对象文件，描述了模块构建的环境。

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

- 必需的头文件。模块源码必须包含它。

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

- 一个包含正在构建的内核版本信息的头文件。
- `LINUX_VERSION_CODE` 整数宏，用于#ifdef版本依赖。

```
EXPORT_SYMBOL (symbol);
```

```
EXPORT_SYMBOL_GPL (symbol);
```

- 用于向内核导出符号的宏。第二种形式限制了导出符号的使用，只能用于GPL许可的模块。

```c
MODULE_AUTHOR(author);

MODULE_DESCRIPTION(description);

MODULE_VERSION(version_string);

MODULE_DEVICE_TABLE(table_info);

MODULE_ALIAS(alternate_name);
```

- 在对象文件中放置模块的文档。

```
MODULE_LICENSE(license);
```

- 声明控制此模块的许可证。

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

module_param(variable, type, perm);
```

- 创建一个模块参数的宏，用户在加载模块时（或对于内置代码，在启动时）可以调整该参数。类型可以是bool、charp、int、invbool、long、short、ushort、uint、ulong或intarray。

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

int printk(const char * fmt, ...);
```

- 对于内核代码的printf的类似物。