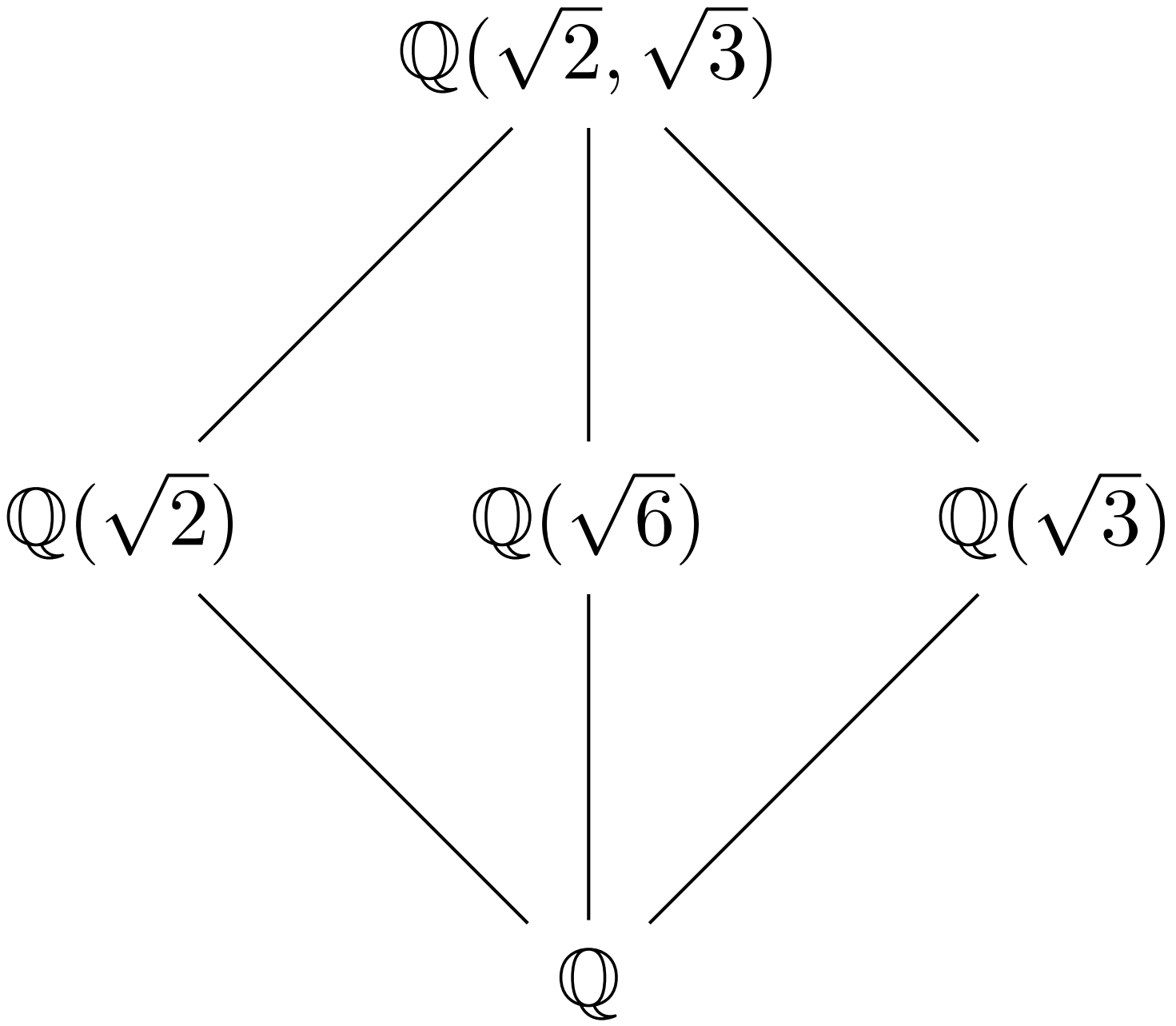

Linux **设备驱动开发教程**

作者：左元

目录

第一章 简介

Linux 内核是一种复杂、轻便、模块化并被广泛使用的软件。大约 80% 的服务器和全世界一半以上设备的嵌入式系统上运行着 Linux 内核。设备驱动程序在整个 Linux 系统中起着至关重要的作用。由于 Linux 已成为非常流行的操作系统。

设备驱动程序通过内核在用户空间和设备之间建立连接。

Linux 起源于芬兰的莱纳斯 · 托瓦尔兹（Linus Torvalds）在 1991 年凭个人爱好开创的一个项目。这个项目不断发展，至今全球有 1000 多名贡献者。现在，Linux 已经成为嵌入式系统和服务器的必选。内核作为操作系统的核心，其开发不是一件容易的事。

和其他操作系统相比，Linux 拥有更多的优点。

- 免费。
- 丰富的文档和社区支持。
- 跨平台移植。
- 源代码开放。
- 许多免费的开源软件。

本教程尽可能做到通用，但是仍然有些特殊的模块，比如设备树，目前在 x86 上没有完整实现。那么话题将专门针对 ARM 处理器，以及所有完全支持设备树的处理器。为什么选这两种架构？因为它们在桌面和服务器（x86）以及嵌入式系统（ARM）上得到广泛应用。

1.1 环境配置

在 Ubuntu 下，安装如下包。

安装一些包

```
$ sudo apt-get update
$ sudo apt-get install gawk wget git diffstat
unzip texinfo \
gcc-multilib build-essential chrpath socat
libsdl1.2-dev \
xterm ncurses-dev lzop
```

安装针对 ARM 体系结构的交叉编译器。

安装交叉编译器

```
$ sudo apt-get install gcc-arm-linux-gnueabihf
```

1.1.1 获取源码

在早期内核（2003 年前）中，使用奇偶数对版本进行编号：奇数是稳定版，偶数是不稳定版。随着 2.6 版本的发布，版本编号方案切换为 X.Y.Z 格式。

- X：代表实际的内核版本，也被称为主版本号，当有向后不兼容的 API 更改时，它会递增。
- Y：代表修订版本号，也被称作次版本号，在向后兼容的基础上增加新的功能后，它会递增。
- Z：代表补丁，表示与错误修订相关的版本。

这就是所谓的语义版本编号方案，这种方案一直持续到 2.6.39 版本；当 Linus Torvalds 决定将版本升级到 3.0 时，意味着语义版本编号在 2011 年正式结束，然后采用的是 X.Y 版本编号方案。

升级到 3.20 版时，Linus 认为不能再增加 Y，决定改用随意版本编号方案：当 Y 值增加到手脚并用也数不过来时就递增 X。这就是版本直接从 3.20 变化到 4.0 的原因。

现在内核使用的 X.Y 随意版本编号方案，这与语义版本编号无关。

> ! Linus：仁慈的独裁者。

源代码的组织

必须使用 Linus Torvald 的 Github 仓库。

下载源码

```
1 $ git clone https://github.com/torvalds/linux
2 $ git checkout 版本号 # 例如：git checkout v4.1
3 $ ls
```

内核中各文件夹的含义：

- arch/：Linux 内核是一个快速增长的工程，支持越来越多的体系结构。这意味着，内核尽可能通用。与体系结构相关的代码被分离出来，并放入此目录中。该目录包含与处理器相关的子目录，例如 alpha/、arm/、mips/、blackfin/等。
- block/：该目录包含块存储设备代码，实际上也就是 I/O 调度算法。
- crypto/：该目录包含密码 API 和加密算法代码。
- Documentation/：这应该是最受欢迎的目录。它包含不同内核框架和子系统所使用 API 的描述。在论坛发起提问之前，应该先看这里。
- drivers/：这是最重的目录，不断增加的设备驱动程序都被合并到这个目录，不同的子目录中包含不同的设备驱动程序。
- fs/：该目录包含内核支持的不同文件系统的实现，诸如 NTFS、FAT、ETX{2,3,4}、sysfs、procfs、NFS 等。
- include/：该目录包含内核头文件。
- init/：该目录包含初始化和启动代码。
- ipc/：该目录包含进程间通信（IPC）机制的实现，如消息队列、信号量和共享内存。
- kernel/：该目录包含基本内核中与体系结构无关的部分。
- lib/：该目录包含库函数和一些辅助函数，分别是通用内核对象（kobject）处理程序和循环冗余码（CRC）计算函数等。
- mm/：该目录包含内存管理相关代码。
- net/：该目录包含网络（无论什么类型的网络）协议相关代码。
- scripts/：该目录包含在内核开发过程中使用的脚本和工具，还有其他有用的工具。

- security/：该目录包含安全框架相关代码。
- sound/：该目录包含音频子系统代码。
- usr/：该目录目前包含了 initramfs 的实现。

内核必须保持它的可移植性。任何体系结构特定的代码都应该位于 arch 目录中。当然，与用户空间 API 相关的内核代码不会改变（系统调用、/proc、/sys），因为它会破坏现有的程序。

1.1.2 内核配置

Linux 内核是一个基于 makefile 的工程，有 1000 多个选项和驱动程序。配置内核可以使用基于 ncurse 的接口命令 `make menuconfig`，也可以使用基于 X 的接口命令 make xconfig。一旦选择，所有选项会被存储到源代码根目录下的 `.config` 文件中。

大多情况下不需要从头开始配置。每个 arch 目录下面都有默认的配置文件可用，可以把它们用作配置起点：

列出配置文件

```
1 $ ls arch/<you_arch>/configs/
```

对于基于 ARM 的 CPU，这些配置文件位于 `arch/arm/configs/`；对于基于 V3S 处理器的 Atguigu 派，默认的配置文件位于 `arch/arm/configs/atguigupi_defconfig`；类似地，对于 x86 处理器，可以在 `arch/x86/configs/` 找到配置文件，仅有两个默认配置文件：`i386_defconfig` 和 `x86_64_defconfig`，它们分别对应于 32 位和 64 位版本。

对 x86 系统，内核配置非常简单：

内核配置

```
1 $ make x86_64_defconfig
2 $ make zImage -j16
3 $ make modules
4 $ makeINSTALL_MOD_PATH </where/to/install>
5 $ modules_install
```

对于基于 V3S 的开发板 AtguiguPI：

可以先执行 `ARCH=arm make atguigupi_defconfig`，然后执行 `ARCH=arm make menuconfig`。前一个命令把默认的内核选项存储到 `.config` 文件中；后一个命令则根据需求来更新、增加或者删除选项。

1.1.3 构建自己的内核

构建自己的内核需要指定相关的体系结构和编译器。

交叉编译

```
1 $ ARCH=arm make atguigupi_defconfig
2 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make zImage -j16
```

内核构建完成后，会在`arch/arm/boot/`下生成一个单独的二进制映像文件。使用下列命令构建模块：

构建模块

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make modules
```

可以通过下列命令安装编译好的模块：

安装模块

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make modules_install
```

`modules_install`目标需要指定一个环境变量`INSTALL_MOD_PATH`，指出模块安装的目录。如果没有设置，则所有的模块将会被安装到`/lib/modules/$(KERNELRELEASE)/kernel/`目录下，具体细节将会在第 2 章讨论。

V3S 处理器支持设备树，设备树是一些文件，可以用来描述硬件（相关细节会在第 6 章介绍）。无论如何，运行下列命令可以编译所有 ARCH 设备树：

编译所有设备树

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make dtbs
```

然而，dtbs 选项不一定适用于所有支持设备树的平台。要构建一个单独的 DTB，应该执行下列命令：

单独编译 *DTB* 文件

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make atguigupi.dtb
```

1.2 内核代码编写风格

深入学习本节之前应该先参考一下内核编码风格手册，它位于内核源代码树的`Documentation/CodingStyle`目录下。编码风格是应该遵循的一套规则，如果想要内核开发人员接受其补丁就应该遵守这一规则。其中一些规

则涉及缩进、程序流程、命名约定等。

常见的规则如下。

- 始终使用 8 个字符的制表符缩进，每一行不能超过 80 个字符。如果缩进妨碍函数书写，那只能说明嵌套层次太多了。
- 每一个不被导出的函数或变量都必须声明为静态的（static）。
- 在带括号表达式的内部两端不要添加空格。
  `s = sizeof(struct file);` 是可以接受的，
  而 `s = sizeof( struct file );` 是不被接受的。
- 禁止使用 `typedef` 。
- 请使用 `/* this */` 注释风格，不要使用 `// this` 。
- 宏定义应该大写，但函数宏可以小写。
- 不要试图用注释去解释一段难以阅读的代码。应该重写代码，而不是添加注释。

1.3 内核结构分配和初始化

内核总是为其数据结构和函数提供两种可能的分配机制。

下面是其中的一些数据结构。

- 工作队列。
- 列表。
- 等待队列。
- Tasklet。
- 定时器。
- 完成量。
- 互斥锁。
- 自旋锁。

动态初始化器是通过宏定义实现的，因此全用大写：

- `INIT_LIST_HEAD()`
- `DECLARE_WAIT_QUEUE_HEAD()`
- `DECLARE_TASKLET()`
- 等等

表示框架设备的数据结构总是动态分配的，每个都有其自己的分配和释放 API。框架设备类型如下。

- 网络设备。
- 输入设备。
- 字符设备。
- IIO 设备。
- 类设备。
- 帧缓冲。
- 调节器。
- PWM 设备。
- RTC。

静态对象在整个驱动程序范围内都是可见的，并且通过该驱动程序管理的每个设备也是可见的。而动态分配对象则只对实际使用该模块特定实例的设备可见。

1.4 类、对象、面向对象的编程

内核通过类和设备实现面向对象的编程。内核子系统被抽象成类，有多少子系统，`/sys/class/`下几乎就有多少个目录。`struct kobject`结构是整个实现的核心，它包含一个引用计数器，以便于内核统计有多少用户使用了这个对象。每个对象都有一个父对象，在`sysfs`（加载之后）中会有一项。

属于给定子系统的每个设备都有一个指向`operations(ops)`结构的指针，该结构提供一组可以在此设备上执行的操作。

第二章 设备驱动程序基础

驱动程序是专用于控制和管理特定硬件设备的软件，因此也被称作设备驱动程序。从操作系统的角度来看，它可以位于内核空间（以特权模式运行），也可以位于用户空间（具有较低的权限）。本教程仅涉及内核空间驱动程序，特别是 Linux 内核驱动程序。我们给出的定义是，设备驱动程序把硬件功能提供给用户程序。

本章涉及以下主题。

- 模块构建过程及其加载和卸载。
- 驱动程序框架以及调试消息管理。
- 驱动程序中的错误处理。

2.1 内核空间和用户空间

内核空间和用户空间的概念有点抽象，主要涉及内存和访问权限，如图2.1所示。可以这样认为：内核是有特权的，而用户应用程序则是受限制的。这是现代 CPU 的一项功能，它可以运行在特权模式或非特权模式。学习第 11 章之后，这个概念会更加清晰。

图2.1说明内核空间和用户空间的分离，并强调了系统调用代表它们之间的桥梁（将在本章后面讨论）。每个空间的描述如下。

- 内核空间：内核驻留和运行的地址空间。内核内存（或内核空间）是由内核拥有的内存范围，受访问标志保护，防止任何用户应用程序有意或无意间与内核搞混。另一方面，内核可以访问整个系统内存，因为它在系统上以更高的优先级运行。在内核模式下，CPU 可以访问整个内存（内核空间和用户空间）。
- 用户空间：正常程序（如 vim 等）被限制运行的地址（位置）空间。可以将其视为沙盒或监狱，以便用户程序不能混用其他程序拥有的内存或任何其他资源。在用户模式下，CPU 只能访问标有用户空间访问权限的内存。用户应用程序运行到内核空间的唯一方法是通过系统调用，其中一些调用是 read、write、open、close 和 mmap 等。用户空间代码以较低的优先级运行。当进程执行系统调用时，软件中断被发送到内核，这将打开特权模式，以便该进程可以在内核空间中运行。系统调用返回时，内核关闭特权模式，进程再次受限。

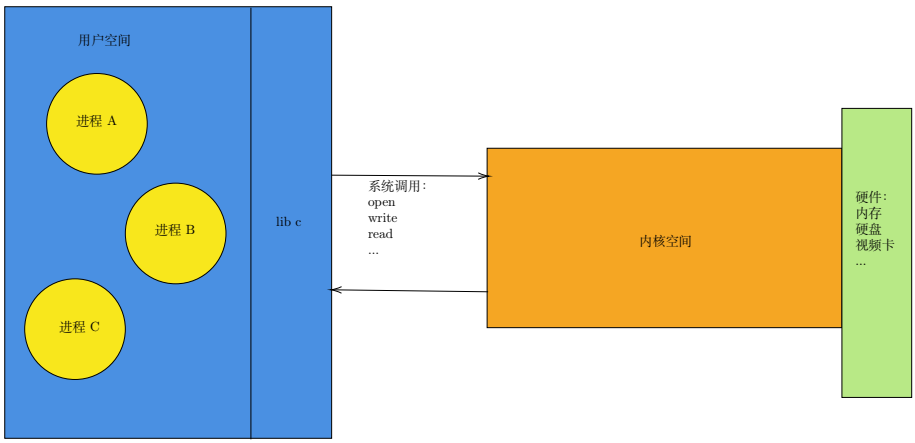

图 2.1: 内核空间和用户空间示意图

2.1.1 模块的概念

模块之于 Linux 内核就像插件（组件）之于用户软件（如 Firefox），模块动态扩展了内核功能，甚至不需要重新启动计算机就可以使用。大多数情况下，内核模块是即插即用的。一旦插入，就可以使用了。为了支持模块，构建内核时必须启用下面的选项：

支持模块

```
1 CONFIG_MODULES=y
```

2.1.2 模块依赖

Linux 内核中的模块可以提供函数或变量，用 EXPORT_SYMBOL 宏导出它们即可供其他模块使用，这些被称作符号。模块 B 对模块 A 的依赖是指模块 B 使用从模块 A 导出的符号。

在内核构建过程中运行 depmod 工具可以生成模块依赖文件。它读取 `/lib/modules/<kernel_release>/` 中的每个模块来确定它应该导出哪些符号以及它需要什么符号。该处理的结果写入文件 modules.dep 及其二进制版本 modules.dep.bin。它是一种模块索引。

2.1.3 模块的加载和卸载

模块要运行，应该先把它加载到内核，可以用 insmod 或 modprobe 来实现，前者需要指定模块路径作为参数，这是开发期间的首选；后者更智能化，是生产系统中的首选。

1. 手动加载。

   手动加载需要用户的干预，该用户应该拥有 root 访问权限。实现这一点的两种经典方法如下。

   在开发过程中，通常使用 insmod 来加载模块，并且应该给出所加载模块的路径：

   insmod 加载模块

   ```
   1         $ insmod /path/to/mydrv.ko
   ```

   这种模块加载形式低级，但它是其他模块加载方法的基础，也是本教程中将要使用的方法。相反，系统管理员或在生产系统中则常用 modprobe。modprobe 更智能，它在加载指定的模块之前解析文件 modules.dep，以便首先加载依赖关系。它会自动处理模块依赖关系，就像包管理器所做的那样：

   modprobe 加载模块

   ```
   1         $ modprobe mydrv
   ```

   能否使用 modprobe 取决于 depmod 是否知道模块的安装。

   能否使用 *modprobe*

   ```
   1         $ /etc/modules-load.d/<filename>.conf
   ```

   如果要在启动的时候加载一些模块，则只需创建文件 `/etc/modules-load.d/<filename>.conf`，并添加应该加载的模块名称（每行一个）。

`<filename>` 应该是有意义的名称，人们通常使用模块：`/etc/modules-load.d/modules.conf`。
当然也可以根据需要创建多个 `.conf` 文件。
下面是一个 `/etc/modules-load.d/mymodules.conf` 文件中的内容：

配置文件示例

```
#this line is a comment
uio
iwlwifi
```

2. 自动加载
depmod 实用程序的作用不只是构建 modules.dep 和 modules.dep.bin 文件。内核开发人员实际编写驱动程序时已经确切知道该驱动程序将要支持的硬件。他们把驱动程序支持的所有设备的产品和厂商 ID 提供给该驱动程序。depmod 还处理模块文件以提取和收集该信息，并在/lib/modules/<kernel_release>/modules.alias 中生成 modules.alias 文件，该文件将设备映射到其对应的驱动程序。
下面的内容摘自 modules.alias：

modules.alias

```
alias usb:v0403pFF1Cd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pFF18d*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFFd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFEd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFDd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFCd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0D8Cp0103d*dc*dsc*dp*ic*isc*ip*in* snd_usb_audio
alias usb:v*p*d*dc*dsc*dp*ic01isc03ip*in* snd_usb_audio
alias usb:v200Cp100Bd*dc*dsc*dp*ic*isc*ip*in* snd_usb_au
```

在这一步，需要一个用户空间热插拔代理（或设备管理器），通常是 udev（或 mdev），它将在内核中注册，以便在出现新设备时得到通知。
通知由内核发布，它将设备描述（pid、vid、类、设备类、设备子类、接口以及可标识设备的所有其他信息）发送到热插拔守护进程，守护进程再调用 modprobe，并向其传递设备描述信息。接下来，modprobe 解析 modules.alias 文件，匹配与该设备相关的驱动程序。在加载模块之前，modprobe 会在 module.dep 中查找与其有依赖关系的模块。如果发现，则在相关模块加载之前先加载所有依赖模块；否则，直接加载该模块。

3. 模块卸载
常用的模块卸载命令是 rmmod，人们更喜欢用这个来卸载 insmod 命令加载的模块。使用该命令时，应该把要卸载的模块名作为参数向其传递。
模块卸载是内核的一项功能，该功能的启用或禁用由 CONFIG_MODULE_UNLOAD 配置选项的值决定。没有这个选项，就不能卸载任何模块。以下设置将启用模块卸载功能：

模块卸载配置

```
CONFIG_MODULE_UNLOAD=y
```

在运行时，如果模块卸载会导致其他不良影响，则即使有人要求卸载，内核也将阻止这样做。这是因为内核通过引用计数记录模块的使用次数，这样它就知道模块是否在用。如果内核认为删除一个模块是不安全的，就不会删除它。然而，以下设置可以改变这种行为：

强制卸载模块配置

```
MODULE_FORCE_UNLOAD=y
```

上面的选项应该在内核配置中设置，以强制卸载模块：

强制卸载模块

```
rmmod -f mymodule
```

而另一个更高级的模块卸载命令是`modeprobe -r`，它会自动卸载未使用的相关依赖模块：

自动卸载相关依赖

```
modeprobe -r mymodule
```

这对于开发者来说是一个非常有用的选择。用下列命令可以检查模块是否已加载：

列出模块

```
lsmod
```

2.2 驱动程序框架

helloworld.c

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4
5  static int __init helloworld_init(void) {
6      pr_info("Hello world!\n");
7      return 0;
8  }
9
10 static void __exit helloworld_exit(void) {
11     pr_info("End of the world\n");
12 }
13
14 module_init(helloworld_init);
15 module_exit(helloworld_exit);
16 MODULE_AUTHOR("Yuan Zuo <zuoyuantc@gmail.com>");
17 MODULE_LICENSE("GPL");
```

2.2.1 模块的入口点和出口点

内核驱动程序都有入口点和出口点：前者对应于模块加载时调用的函数（modprobe 和 insmod），后者是模块卸载时执行的函数（在执行 rmmod 或 modprobe -r 时）。

main() 函数是用 C/C++ 编写的每个用户空间程序的入口点，当这个函数返回时，程序将退出。而对于内核模块，情况就不一样了：入口点可以随意命名，它也不像用户空间程序那样在 main() 返回时退出，其出点在另一个函数中定义。开发人员要做的就是通知内核把哪些函数作为入口点或出口点来执行。实际函数 hellowolrd_init 和 hellowolrd_exit 可以被命名成任何名字。实际上，唯一必须要做的是把它们作为参数提供给 module_init() 和 module_exit() 宏，将它们标识为相应的加载和删除函数。

综上所述，module_init() 用于声明模块加载（使用 insmod 或 modprobe）时应该调用的函数。初始化函数中要完成的操作是定义模块的行为。module_exit() 用于声明模块卸载（使用 rmmod）时应该调用的函数。

> ! 在模块加载或卸载后，init 函数或 exit 函数立即运行一次。

`__init`和`__exit`属性

`__init` 和 `__exit` 实际上是在 include/linux/init.h 中定义的内核宏，如下所示：

宏定义

```
1  #define __init __section(.init.text)
```

```
2 #define __exit __section(.exit.text)
```

`__init` 关键字告诉链接器将该代码放在内核对象文件的专用部分。这部分事先为内核所知，它在模块加载和 init 函数执行后被释放。这仅适用于内置驱动程序，而不适用于可加载模块。内核在启动过程中第一次运行驱动程序的初始化函数。

由于驱动程序不能卸载，因此在下次重启之前不会再调用其 init 函数，没有必要在 init 函数内记录引用次数。对于 `__exit` 关键字也是如此，在将模块静态编译到内核或未启用模块卸载功能时，其相应的代码会被忽略，因为在这两种情况下都不会调用 exit 函数。`__exit` 对可加载模块没有影响。

我们花一点时间进一步了解这些属性的工作方式，这涉及被称作可执行和可链接格式（ELF）的目标文件。ELF 目标文件由不同的命名部分组成，其中一些部分是必需的，它们成为 ELF 标准的基础，但也可以根据自己的需要构建任一部分，并由特殊程序使用。内核就是这样做。执行 `objdump -h module.ko` 即可打印出指定内核模块 module.ko 的不同组成部分。

打印内容只有少部分属于 ELF 标准。

- .text：包含程序代码，也称为代码。
- .data：包含初始化数据，也称为数据段。
- .rodata：用于只读数据。
- .comment：注释。
- 未初始化的数据段，也称为由符号开始的块（block started by symbol，bss）。

其他部分是根据内核的需要添加的。本章较重要的部分是.modeinfo 和.init.text，前者存储有关模块的信息，后者存储以 __init 宏为前缀的代码。

链接器（Linux 系统上的 ld）是 binutils 的一部分，负责将符号（数据、代码等）放置到生成的二进制文件中的适当部分，以便在程序执行时可以被加载器处理。二进制文件中的这些部分可以自定义、更改它们的默认位置，甚至可以通过提供链接器脚本 [称为链接器定义文件（LDF）或链接器定义脚本（LDS）] 来添加其他部分。要实现这些操作只需通过编译器指令把符号的位置告知链接器即可，GNU C 编译器为此提供了一些属性。Linux 内核提供了一个自定义 LDS 文件，它位于 arch/<arch>/kernel/vmlinux.lds.S 中。对于要放置在内核 LDS 文件所映射的专用部分中的符号，使用 __init 和 __exit 进行标记。

总之，__init 和 __exit 是 Linux 指令（实际上是宏），它们使用 C 编译器属性指定符号的位置。这些指令指示编译器将以它们为前缀的代码分别放在.init.text 和.exit.text 部分，虽然内核可以访问不同的对象部分。

2.2.2 模块信息

即使不读代码，也应该能够收集到关于给定模块的一些信息（如作者、参数描述、许可）。内核模块使用其.modinfo 部分来存储关于模块的信息，所有 MODULE_* 宏都用参数传递的值更新这部分的内容。其中一些宏是 MODULE_DESCRIPTION()、MODULE_AUTHOR() 和 MODULE_LICENSE()。内核提供的在模块信息部分添加条目的真正底层宏是 MODULE_INFO(tag,info)，它添加的一般信息形式是 tag=info。这意味着驱动程序作者可以自由添加其想要的任何形式信息，例如：

作者信息

```
1 MODULE_INFO(my_field_name, "What eeasy value");
```

在给定模块上执行`objdump -d -j .modinfo`命令可以转储内核模块.modeinfo 部分的内容，如图 2-3 所示。

第三章 设备树的概念

设备树（DT，Derive Tree）是易于阅读的硬件描述文件，它采用 JSON 式的格式化风格，在这种简单的树形结构中，设备表示为带有属性的节点。属性可以是空（只有键，用来描述布尔值），也可以是键值对，其中的值可以包含任意的字节流。本章简单地介绍 DT，每个内核子系统或框架都有自己的 DT 绑定。讲解有关话题时将包括具体的绑定。DT 源于 OF，这是计算机公司公认的标准，其主要目的是定义计算机固件系统的接口。

本章涉及以下主题。

- 命名约定，以及别名和标签。
- 描述数据类型及其 API。
- 管理寻址方案和访问设备资源。
- 实现 OF 匹配风格，提供应用程序的相关数据。

3.1 设备树机制

将选项 `CONFIG_OF` 设置为 Y 即可在内核中启用 DT。要在驱动程序中调用 DT API，必须添加以下头文件：