# 前言

# 设备驱动程序简介

Linux系统的优点是，系统内部实现细节对所有人都是公开的。Linux内核由大量复杂的代码组成，设备驱动程序可以作为进入Linux内核世界大门的切入口。

  设备驱动程序在Linux内核中，是一个个独立的黑盒子，在调用内部接口时某个特定硬件做出响应，这些接口隐藏了设备的工作细节。用户的操作通过独立于特定驱动程序的一组标准化调用执行，将这些调用映射到作用于实际硬件的设备特有操作上，是设备驱动程序的任务。

  模块化的特点使得Linux驱动程序的编写非常方便简单

## 设备驱动程序的作用

设备驱动程序的作用在于提供机制，而不是提供策略。

  区分机制和策略是Unix设计背后隐含的最好思想之一。大部分编程问题都可以分成两部分：

机制：需要提供什么功能

策略：如何使用这些功能

我们应当尽可能做到让驱动程序不带策略。

  编写驱动程序时，特别注意：编写访问硬件的内核代码时，不要给用户强加任何特定策略。因为不同的用户有不同的需求，驱动程序应该处理如何使硬件可用的问题，而将怎样使用硬件的问题留给应用程序。

  驱动程序可以看作是应用程序和实际设备之间的一个软件层。

  驱动程序设计要综合考虑下面三个问题的因素：提供给用户尽量多的选项、编写驱动程序要占用的时间、尽量保持程序简单。

## 内核功能划分

Sdfg

根据内核完成任务的不同，内核功能分为如下几个部分：

### 进程管理

  负责创建和销毁进程，并处理它们和外部世界之间的连接。如进程之间的通信，控制进程如何共享CPU的调度器等。

### 内存管理

  内存是计算机的主要资源之一，用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源上为每个进程创建一个虚拟地址空间。

### 文件系统

  Unix中的每个对象几乎都可以当作文件来看待。内核可以在没有结构的硬件上构造结构化的文件系统。

### 设备控制

   几乎每一个系统操作最终都会映射到物理设备上。几乎所有设备操作都由驱动程序来完成。内核必须为系统中的每个外设嵌入相应的驱动程序。

### 网络功能

  大部分网络操作和具体进程无关，网络功能必须由操作系统管理。系统负责在应用程序和网络接口之间传递数据包，并根据网络活动控制程序的执行。所有的路由和地址解析都由内核处理。

### 可装载模块

  Linux有个很好的特性，内核提供的特性可在**运行时扩展，即系统启动并运行时，可以以模块的形式，向内核添加、移出功能**

作者：micro虾米  
链接：https://www.jianshu.com/p/f0990bb6abdc  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 设备和模块的分类

Fd

Linux系统将设备分为三种基本类型，字符设备模块、块设备模块、网络设备模块。

##### 字符设备

  字符设备是个能像字节流一样被访问的设备，类似文件。比如：字符终端(/dev/console)和串口(/dev/ttys0)

  设备文件和普通文件的唯一差别是，对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个个只能顺序访问的数据通道。除个别例外。

##### 块设备

  块设备是按照块进行读写的设备，一次只能传输一个或多个完整的块，一个块一般是512字节，或2的更高次幂字节的数据。  
  块设备上能够容纳文件系统。

##### 网络设备

  网络设备是一个能够和其他主机交换数据的设备，通常，接口是个硬件设备，也可能是个纯软件设备，比如回环接口(loopback)。  
  网络设备由内核中的网络子系统驱动，负责发送和接收数据包。

  除了设备驱动程序外，内核中其他一些功能，不管是硬件还是软件功能，都模块化了。如常见的文件系统

## 安全问题

D

系统中的所有安全检查都是由内核代码进行的，如果内核有安全漏洞，则整个系统就有安全漏洞。

  驱动程序编写者应当尽量避免在代码中实现安全策略。  
  驱动程序编写者应当避免由于自身原因引入安全方面的缺陷。  
  任何从用户进程得到的输入只有经过内核严格验证后才能使用。必须小心对待未初始化的内存，任何从内核中得到的内存，都必须在提供给用户进程或者设备之前清零或者以其他方式初始化，否则可能发生信息泄露。

  Linux内核也可编译为不支持模块方式，从而可以关闭任何模块相关的安全漏洞。此时，所有需要的驱动程序必须直接编译到内核中

作者：micro虾米  
链接：https://www.jianshu.com/p/f0990bb6abdc  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 版本编号

## 许可证条款

## 加入内核开发社团

## 本书概要

# 第二章 构造和运行模块

## 核心模块与应用程序的对比

### 用户空间和内核空间

  模块运行在内核空间，应用程序运行在用户空间。

  在Unix中，内核运行在最高级别，即超级用户态，这个级别可以进行所有的操作。而应用程序运行在最低级别，即用户态，这个级别处理器控制着对硬件的直接访问及对内存的非授权访问。

  内核空间和用户空间，不仅说明两种模式具有不同的优先级等级，还说明每个模式都有自己的内存映射，即自己的地址空间。

  当应用程序执行系统调用或者被硬件中断挂起时，Unix将执行模式从用户空间切换到内核空间。执行系统调用的内核代码运行在进程上下文中，代表调用进程执行操作。因此能够访问进程地址空间的所有数据。而处理器硬件中断的内核代码和进程时异步的，和任何一个进程无关。

  模块化代码在内核空间运行，用于扩展内核功能。驱动程序要执行两类任务，模块中的某些函数作为系统调用的一部分执行；其他函数则负责中断处理

作者：micro虾米  
链接：https://www.jianshu.com/p/53b69060a339  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### 内核中的并发

  内核编程和应用程序编程的区别在于**对并发的处理**。大部分应用程序，除了多线程应用程序外，通常都是顺序执行的。内核代码的运行环境更加复杂，即使是最简单的内核模块，都需要注意：同一时刻，可能会有很多事情发生。

内核编程必须考虑**并发**问题的原因：

1. Linux系统中通常正在运行多个并发进程，并且可能有多个进程同时使用驱动程序；
2. 大多数设备能够中断处理器，而中断处理程序异步运行，而且可能在驱动程序正试图处理其他任务时被调用；
3. 有些软件抽象（如：内核定时器）也是异步运行的；
4. Linux可能运行在对称多处理器系统（SMP），因此可能同时不止一个CPU运行驱动程序；
5. Linux2.6内核代码已经是可抢占的，即使在单处理器系统上也存在类似多处理器系统的并发问题。

  Linux内核代码（包括驱动程序）必须是**可重入**的，必须能够同时运行在多个上下文中。内核数据结构要保证多个线程分开执行，访问共享数据的代码必须避免破坏共享数据。驱动要能够处理并发问题，同时避免竞态。内核代码不能假定在给定代码段中能够独占处理器。

### 4. 当前进程

  虽然内核模块不像应用程序那样顺序地执行，然而内核执行的大多数操作还是和某个特定进程相关。内核代码可通过访问全局项current来获得当前进程。

  current在<asm/current.h>中定义，是一个指向struct task\_struct的指针。current指针指向当前正在运行的进程。可以通过访问struct task\_struct的某些成员来打印当前进程的进程ID和命令名：

printk(KERN\_INFO "The process is \"%s\" (pid %i)\n",

current->comm, current->pid);

  存储在current->comm成员中的命令名是当前进程所执行的程序文件的基本名称，裁剪在15个字符以内。

### 5. 其他细节

  应用程序在虚拟内存中布局，并具有一块很大的栈空间。栈用来保存函数调用历史及当前活动函数中的自动变量。而内核具有很小的栈，可能只有一个4096字节大小的页空间。驱动的函数必须和整个内核空间调用链一同共享这个栈。因此，不能声明大的自动变量，如果需要大的结构，应该在调用时动态分配该结构。

  在内核API中，具有双下划线（\_\_）的函数名称，通常是接口的底层组件，应谨慎使用。

  内核代码不能实现浮点数运算，内核代码中不需要浮点运算

作者：micro虾米  
链接：https://www.jianshu.com/p/53b69060a339  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### 核心模块与应用程序的对比

Gfh

内核模块和应用程序之间的不同之处：

1. 大多数中小规模的应用程序是从头到尾执行单个任务，而模块却只是预先注册自己以便服务于将来的某个请求，然后初始化函数立即结束。即模块初始化函数(hello\_init)的任务就是为以后调用模块函数预先做准备。模块的退出函数(hello\_exit)将在模块被卸载之前调用。
2. 这和事件驱动编程有点类似，但不是所有的应用程序都是事件驱动的，而每个内核模块都是这样的。事件驱动程序和内核模块之间的另一个区别是，应用程序在退出时，可以不管资源的释放或者其他的清除工作，但模块的退出函数必须撤销初始化函数所做的一切，保证没有多余内容残留在系统中。
3. 应用程序可以调用它并未定义的函数，因为链接过程能够解析外部引用，从而链接使用适当的函数库。而模块仅仅被链接到内核，因此模块能调用的函数仅仅是由内核导出的那些函数，而不存在任何可链接的函数库。因为没有任何函数库会和模块链接，因此源文件中不能包含通常应用程序的头文件。内核模块只能使用作为内核一部分的函数。和内核相关的大多数相关头文件保存在include/linux和include/asm目录中。
4. 应用程序和内核编程的处理错误的方式不同，应用程序的段错误可以使用调试器跟踪到源代码中的问题，而内核错误即使不影响系统，也会杀死当前进程。

模块卸载的好处，有助于缩短模块化驱动程序的开发周期

作者：micro虾米  
链接：https://www.jianshu.com/p/53b69060a339  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 编译和装载

### 1. 设置测试系统

第1步，要先从kernel.org的镜像网站上获取一个主线内核，并安装到自己的系统中，因为学习驱动程序的编写，最好使用标准内核。

第2步，必须在自己的系统中配置并构造好内核树，这样可以得到一个更加健壮的模块装载器，可以使内核的模块要和内核源码树中的目标文件连接。同时也需要这些目标文件存在于内核目录树中。这样，准备一个内核源代码树，构造一个新内核，并安装到自己的系统中，有利于开发工作的进行。

第3步，要决定在什么地方完成模块的开发、调试，内核代码中的错误可能导致用户进程甚至整个系统崩溃，这些错误通常不会制造更加严重的问题，但建议开发者应该在一个不包含任何敏感数据或者不执行重要服务的系统上完成内核的调试实验。

### 2. Hello World模块

   几乎所有编程学习都是以“Hello world”示例程序开始的，在这里的模块中也可以使用这个经典历程。如下代码段：

#include <linux/init.h>

#include <linux/module.h>

static int \_\_init hello\_init(void)

{

printk(KERN\_ALERT "Hello, world\n");

return 0;

}

static void \_\_exit hello\_exit(void)

{

printk(KERN\_ALERT "Goodbye, cruel world\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("Dual BSD/GPL");

module\_init() 宏表示模块被装载到内核时调用hello\_init函数  
module\_exit() 宏表示模块被移除内核时调用hello\_exit函数  
MODULE\_LICENSE() 宏用来告诉内核，该模块采用自由许可证；如果没有该声明，内核在装载该模块时会产生抱怨。

printk() 函数在Linux内核中定义，功能类似标准C库中的printf() 函数。是内核独有的打印输出函数，因为内核载运行的时不能依赖C库。模块中能够调用printk()函数，是因为在insmod插入模块之后，模块连接到内核，可以访问内核的公共符号。KERN\_ALERT定义了消息的优先级，需要在模块中显式地指定高优先级的原因是：具有默认优先级的消息可能不会输出在控制台上。

模块编译过程如下：

# make

make -C /lib/modules/4.15.0-55-generic/build M=/home/mcy/code/ldd3-demo/1\_module modules

make[1]: Entering directory '/usr/src/linux-headers-4.15.0-55-generic'

Makefile:976: "Cannot use CONFIG\_STACK\_VALIDATION=y, please install libelf-dev, libelf-devel or elfutils-libelf-devel"

CC [M] /home/mcy/code/ldd3-demo/1\_module/hello.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/mcy/code/ldd3-demo/1\_module/hello.mod.o

LD [M] /home/mcy/code/ldd3-demo/1\_module/hello.ko

make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-55-generic'

  可以通过insmod和rmmod命令来加载和卸载模块，注意：只有超级用户才有权加载和卸载模块。

# insmod hello.ko

Hello, world

# lsmod

Module Size Used by

hello 16384 0

# rmmod hello

Goodbye, cruel world

   能够编译模块的前提条件是，必须在模块的Makefile能够找到的路径，正确配置和构造了内核树。模块的编译参考下文中编译和装载部分。

   编写设备驱动程序并不困难，真正的困难在于理解设备并最大化其性能。

### 编译和装载

  本节将详细介绍如何将源代码编译成能够装载到内核中的可执行模块。

#### 3.1 编译模块

在构造内核模块之前，需要先满足一下条件：

1. 应确保具备了正确版本的编译器、模块工具和其他必要的工具；使用太旧、太新的工具都有可能出现未知问题；
2. 应该先配置并构造内核；最好运行和模块对应的内核；

之后，为模块创建Makefile很简单，只需要添加就可以了：

obj-m := hello.o

  由内核构造系统处理其余问题，上面的赋值语句说明有一个模块要从目标文件hello.o构造，而由该目标文件构造的模块名称为hello.ko。

  如果要构造的模块名称为module.ko，该模块由两个源文件file1.c和file2.c生成，则Makefile应该如下书写：

obj-m := module.o

module-objs := file1.o file2.o

  为了上述类型的Makefile文件正常工作，必须在大的内核构造系统中调用它，即在包含模块源代码和Makefile文件的目录中，用下面的命令：

make -C kernel\_path M=`pwd` modules

  kernel\_path为已经构造好的内核源代码的路径。该命令先改变目录到-C选项指定的目录（内核源代码目录），其中保存有内核的顶层Makefile文件。  
M=选项让该Makefile在构造modules目标之前返回到模块源代码目录。然后，modules目标指向obj-m变量中设定的模块，本例中为module.o。

  一般情况下，编译模块的文件为module.ko，在加载该模块文件时，会出现：

# insmod my\_module.ko

module: module is already loaded

  这是因为module.ko这个模块名字和系统中的模块名称冲突造成的，可以在Makefile中修改目标文件，避免使用module作为模块的命名。

  为了使内核树之外的模块构造更加方便，可以使用一下Makefile方法：

// Makefile

# 如果定义了KERNELRELEASE，则说明是从内核构造系统调用的

ifneq ($(KERNELRELEASE), )

obj-m := hello.o

# 否则，是直接从命令行调用的，这时需要调用内核构造系统

else

KERNELDIR ?= ......

PWD := $(shell pwd)

default:

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

clean:

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

endif

  在一个典型的构造过程中，Makefile被读取两次。当Makefile从命令行调用时，KERNELRELEASE变量还未设置，已安装的模块目录中存在一个符号链接，指向内核的构造树，以找到内核的源代码目录。在找到内核源代码树之后，该Makefile会调用default目标，这个目录使用之前描述过的方法，第二次运行make命令（$MAKE），以运行内核构造系统。在第二次读取该Makefile文件时，设置了obj-m，而内核的Makefile负责真正构造该模块。

  如上述Makefile所示，所有的Makefile都应该包含通常用来清除无用文件的目标、安装模块的目标等。

编译清理过程如下所示：

# make clean

make -C /lib/modules/4.15.0-55-generic/build M=/home/mcy/code/ldd3-demo/1\_module clean

make[1]: Entering directory '/usr/src/linux-headers-4.15.0-55-generic'

CLEAN /home/mcy/code/ldd3-demo/1\_module/.tmp\_versions

CLEAN /home/mcy/code/ldd3-demo/1\_module/Module.symvers

make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-55-generic'

#### 3.2 装载和卸载模块

   模块构造完成之后，下一步就是将模块装入内核，insmod和ld类似，insmod将模块的代码和数据装入内核，然后使用内核的符号表解析模块中任何未解析的符号。内核不会修改模块在磁盘的文件，而是修改内存中的副本。insmod可以接受一些命令行选项，并且可以在模块链接到内核之前给模块中的变量赋值（即模块传参，参考模块参数），模块可以在装载时进行配置，比编译时的配置更加灵活。

   insmod程序加载模块的过程解析，insmod依赖于定义在kernel/module.c中的系统调用，sys\_init\_module() 函数给模块分配内核内存以便装载模块，然后将模块正文复制到内存区域，并通过内核符号表解析模块中的内核引用，最后调用模块的初始化函数。sys\_前缀的函数用于系统调用，其他函数不能使用，方便使用grep搜索系统调用。

   modprobe工具和insmod类似，也用来将模块装载到内核中。modprobe和insmod的区别在于，modprobe会考虑要装载的模块是否引用了一些当前内核不存在的符号。如果有，modprobe会在当前模块搜索路径中查找定义了这些符号的其他模块。如果找到了这些模块，会同时将这些模块装载到内核。此时如果使用insmod将会失败，并在系统日志文件中记录“unresolved symbols”（未解析的符号）消息。

  rmmod工具从内核中移除模块。如果内核认为该模块正在使用状态，或者内核配置为禁止卸载模块，则无法移除该模块。

  lsmod工具列出当前装载到内核中的所有模块，还有其他模块是否在使用某个模块等信息。lsmod工具通过读取/proc/modules文件来获取这些信息。当前已装载模块信息也可以在/sys/module目录下找到。

# insmod hello.ko

Hello, world

# lsmod

Module Size Used by

hello 16384 0

# rmmod hello

Goodbye, cruel world

#### 3.3 版本依赖

   在缺少modversions的情况下，模块代码必须针对要链接的每个版本的内核重新编译。模块和特定内核版本定义的数据结构和函数原型紧密关联。

  在构造模块过程中，可以将模块和当前内核树中的vermagic.o文件链接，该目标文件包含了大量有关内核的信息，包括目标内核版本、编译器版本以及一些重要配置变量的设置。在试图装载模块时，这些信息用来检查模块和正在运行的内核的兼容性。如果有任何不匹配，就不会装载该模块，同时会有如下信息：

# insmod hello.ko

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

查看系统日志文件/var/log/messages，将看到导致模块装载失败的具体原因。

  如果要为某个特定的内核版本编译模块，则需要该特定版本对应的构造系统和源代码树。同时需要修改Makefile中的KERNELDIR变量来实现。如果打算编写一个能够和多个内核版本一起工作的模块，则必须使用宏以及#ifdef条件编译来构造并编译模块代码。

可以参考<linux/version.h>中的相关定义，该头文件已包含于<linux/module.h>头文件中，参考如下宏定义：

UTS\_RELEASE

&emsp;&emsp;扩展为一个描述内核版本的字符串，如：2.6.10

LINUX\_VERSION\_CODE

&emsp;&emsp;扩展为内核版本的二进制表示，版本发行号中的每一部分对应一个字节

KERNEL\_VERSION(major, minor, release)

&emsp;&emsp;该宏已组成版本号的三部分为参数，创建整数的版本号

  通过检查KERNEL\_VERSION和LINUX\_VERSION\_CODE宏而使用预处理条件，能够解决大部分基于内核版本的依赖性问题。最好的处理方法是，将所有相关的预处理条件语句几种存放在一个特定的头文件里。一般而言，依赖于特定版本或平台的代码应该隐藏在低层宏或者函数之后，高层函数可直接调用这些函数，而无需关注低层细节。这样的代码便于阅读，更为健壮。

## 内核符号表

### 1. 内核符号表

   在模块的装载中，insmod命令使用公共内核符号表来解析模块中未定义的符号。公共内核符号表中包含了所有的全局内核项的地址（即函数和变量的地址），这是实现模块化驱动程序所必需的。当模块被装入内核之后，它所导出的任何符号都会变成内核符号表的一部分。通常情况下，模块只需要实现自己的功能，不需要导出任何符号，但是如果其他模块需要从某个模块中获得某个符号，也可以导出该符号。

  新模块可以使用由我们的模块导出的符号，可以在其他模块上层叠新的模块。模块层叠技术在很多主流的内核源码中应用。如：每个USB输入设备模块层叠在usbcore和input模块之上。模块层叠技术在复杂的项目中非常有用。如果以驱动程序的形式实现一个新的软件抽象，则可以为硬件相关的实现提供一个“插头”。

   modprobe命令是处理层叠模块的一个实用工具。它除了和insmod类似的装入指定模块的功能之外，还同时装入指定模块所依赖的其他模块。modprode命令只能从标准的已安装模块目录中搜索需要装入的模块。

  通过层叠技术，可以将模块划分为多个层，通过简化每个层可缩短开发时间，和之前提到的机制和策略分离类似。

  Linux内核头文件提供一个方便的方法来管理符号对模块外部的可见性，减少可能的名字空间污染，即名称可能和内核其他地方定义的名称冲突，并适当隐藏信息。模块向其他模块导出符号，应该使用下面的宏：

EXPORT\_SYMBOL(name);

EXPORT\_SYMBOL\_GPL(name);

   这两个宏均用于将符号导出到模块外部。\_GPL版本使得要导出的模块只能被GPL许可证下的模块使用。**注意**：符号必须在模块文件的全局部分导出，不能在函数中导出。

### 2. 预备知识

   内核是一个特定的环境，对需要和它接口的代码有其自己的要求。有几个头文件是专门用于模块的，因此必须包含在每个可装载的模块中。所有的模块代码中都包含以下两个头文件：

#include <linux/module.h>

#include <linux/init.h>

// module.h 文件包含有可装载模块需要的大量符号和函数定义；

// init.h 头文件是用来初始化和清除函数；

  大部分在装载模块时向模块传递参数的模块要包含moduleparam.h头文件。模块应该指定代码所使用的许可证，需要包含：

MODULE\_LICENSE("GPL");

内核能识别的许可证有：  
  "GPL"  
  "GPLv2"  
  "GPL and additional rights"  
  "Dual BSD/GPL"  
  "Dual MPL/GPL"  
  "Proprietary"

  如果模块没有显式地标记为上述内核可识别的许可证，则会被假定是专有的，内核装载该模块会被“污染”。

可在模块中包含的其他描述性定义：

MODULE\_AUTHOR // 描述模块作者

MODULE\_DESCRIPTION // 说明模块用途的简短描述

MODULE\_VERSION // 代码修订号，有关版本字符串的信息

MODULE\_ALIAS // 模块的别名

MODULE\_DEVICE\_TABLE // 用来告诉用户空间模块所支持的设备

  以上宏定义，可以出现在模块源文件中函数以外的任何地方，但内核编码习惯是将这些声明放在文件的最后

## 初始化和关闭

#### 1. 初始化函数

   模块的初始化函数负责注册模块所提供的任何设施，即可以被应用程序访问的新功能，可能是一个完整的驱动程序或者仅仅是一个新的软件抽象。初始化函数的定义通常如下所示：

static int \_\_init initialization\_function(void)

{

// 初始化代码

return 0;

}

module\_init(initialization\_function);

   初始化函数被声明为static，因为初始化函数在特定文件之外没有其他意义。\_\_init标记表明该函数仅在初始化期间使用。在模块被装载之后，模块装载器就会将初始化函数扔掉，可将该函数占用的内存释放出来。

  注意：不要在结束初始化之后仍要使用的函数或数据结构上使用\_\_init和\_\_initdata标记。对于\_\_devinit和\_\_devinitdata，只有在内核未被配置为支持热插拔设备的情况下，才会被翻译为\_\_init和\_\_initdata。

  module\_init()宏的使用是强制性的，会在模块的目标代码中增加一个特殊的段，用于说明内核初始化函数所在的位置。如果没有这个定义，初始化函数永远不会被调用。

#### 2. 清除函数

   每个模块都需要一个清除函数，在模块被移除前注销接口并向系统中返回所有资源。该函数定义如下：

static void \_\_exit cleanup\_function(void)

{

// 清除代码

}

module\_exit(cleanup\_function);

  清除函数没有返回值，\_\_exit修饰词标记该代码仅用于模块卸载，编译器会把该函数放在特殊的ELF段中。如果模块被直接编译到内核中，或者内核配置不允许卸载模块，则被标记为\_\_exit的函数将被直接丢弃。所以被标记为\_\_exit的函数只能在模块被卸载或者系统关闭时被调用，其他任何用法都是错的。module\_exit()声明对于内核找到模块的清除函数是必需的。如果一个模块未定义清除函数，则内核不允许卸载该模块。

#### 3. 初始化过程中的错误处理

  在内核中注册设施时，注册可能会失败。即使最简单的动作，都需要内存分配，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，并确保所请求的操作已真正执行成功。如果在注册设施时遇到错误，首先要判断模块是否可以继续初始化，只要可能，模块应该继续向前并尽可能提供其功能。

  如果在发生了某个特定类型的错误之后无法继续装载模块，则要将出错之前的所有注册工作都撤销掉。即当模块的初始化出现错误之后，模块必须自行撤销已注册的设施。如果未能撤销已注册的设施，则内核会处于一种不稳定状态，这时，唯一有效的解决办法就是重新引导系统。所以必须在初始化过程出现错误时认真完成正确的工作。

  错误恢复的处理有时使用goto语句非常有效。正常情况下，很少使用goto，但是唯一在错误处理时却非常有效。内核经常使用goto来处理错误。如下例子所示：

int \_\_init my\_init\_function(void)

{

int err;

// 使用指针和名称注册

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; // 成功

fail\_those:

unregister\_that(ptr2, "skull");

fail\_that:

unregister\_that(ptr1, "skull");

fail\_this:

return err; // 返回错误

}

在出错的时候使用goto语句，将只撤销出错时刻以前所成功注册的那些设施。

  另一种方法是，记录任何成功注册的设施，在出错的时候调用模块的清除函数。清除函数将仅仅回滚已成功完成的步骤。这种方法需要更多的代码和CPU时间，因此在追求效率的代码中使用goto语句是最好的错误恢复机制。

  在Linux内核中错误编码是定义在<linux/errno.h>头文件中的负整数，如果不想使用其他函数返回的错误码，应该包含<linux/errno.h>头文件，以使用如：-ENODEV、-ENOMEM之类的符号值。每次返回核时的错误编码是个好习惯，因为用户程序可以通过perror()函数或类似途径将错误符号转换为有意义的字符串。

  模块的清除函数需要撤销初始化函数所注册的所有设施，并且习惯上以相反于注册的顺序撤销设施，如下所示：

void \_\_exit my\_cleanup\_function(void)

{

unregister\_those(ptr3, "skull");

unregister\_that(ptr2, "skull");

unregister\_this(ptr1, "skull");

return;

}

  如果初始化和清除工作涉及很多设施，则goto方法可能难以管理，因为所有用于清除设施的代码在初始化函数中给重复，同时一些标号交织在一起。

  每次发生错误时从初始化函数中调用清除函数，将减少代码的重复并且时代码更清晰、更有条理。清除函数必须在撤销每项设施的注册之前检查它的状态。如下示例：

struct something \*item1;

struct somethingelse \*item2;

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; // 返回成功

fail:

my\_cleanup();

return err; // 返回错误

}

  如上代码所示，根据调用的注册/分配函数的语义，可以使用或不使用外部标记来标记每个初始化步骤的成功。这种方式的初始化能很好地扩展到对大量设施的支持。**注意**：因为清除函数被非退出代码调用，因此不能将清除函数标记为\_\_exit；

#### 4. 模块装载竞争

  模块装载中也存在**竞态**。在模块注册完成之前，内核的某些部分可能会立即使用我们刚刚注册的任何设施，即在初始化函数还在运行的时候，内核就完全可能会调用我们的模块。因此，在首次注册完成之后，代码就应该准备好被内核的其他部分调用；在支持某个设施的所有内部初始化完成之前，不要注册任何设施。

  当模块初始化失败而内核的某些部分已经使用了模块所注册的某个设施时，此时根本不应该出现模块初始化失败，因为模块已经成功导出了可用的功能及符号。如果初始化一定要失败，则应该仔细处理内核其他部分正在进行的操作，并且要等待这些操作的完成。

作者：micro虾米  
链接：https://www.jianshu.com/p/72a68705e4c8  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 模块参数

### 模块参数

  由于系统不同，驱动程序需要的参数也许会变化，包括设备编号、控制参数等。为满足这些需求，内核允许对驱动程序指定参数，而这些参数可在装载驱动程序模块时改变。这些参数的值可以在运行insmod或modprob命令装载模块时赋值，而modprobe命令还可以从它的配置文件（/etc/modprob.conf）中读取参数值。

  例如：在前面“Hello world”模块的基础上进行扩展，添加两个参数：一个整数值howmany；另一个字符串whom；将向whom问候howmany次。可以使用命令：

insmod hellop.ko howmany=10 whom="Mom"

  驱动程序的实现方式中，参数必须使用module\_param()宏来声明，该宏需要三个参数：变量的名称、类型、及访问许可掩码，该宏必须放在驱动任何函数之外，通常在源文件的头部。

#include <moduleparam.h>

static char \*whom = "world";

static int howmany = 1;

module\_param(howmany, int, S\_IRUGO);

module\_param(whom, charp, S\_IRUGO);

内核支持的模块参数类型如下：  
bool  
invbool  
  布尔值（取true或false），关联变量应该是int型。invbool类型反转其值，即true变成false，而false变成true。  
charp  
  字符指针值。内核会为用户提供的字符串分配内存，并设置指针。  
int  
long  
short  
uint  
ulong  
ushort  
  具有不同长度的基本整数值。以u开头的类型用于无符号值。

  对于数组参数，在提供数组值时用逗号划分各数组成员。用以下宏：

module\_param\_array(name, type, num, perm);

name: 数组名称，即参数名称；

type: 数组元素的类型；

num: 是一个整型变量；

perm: 是常见的访问许可值；

  模块装载器会拒绝接受超过数组大小的值。所有的模块参数都应该给定一个默认值，insmod命令只会在用户明确设置了参数的值的情况下才会改变参数的值。模块可以根据默认值来判断是否是一个显式指定的参数。

module\_param()宏中的最后一个成员是访问许可值，这些宏定义在<linux/stat.h>中定义，该值用来控制谁能够访问sysfs中对模块参数的表述。如果perm=0，就不会有对应的sysfs入口项；否则，模块参数会在/sys/module中出现，并设置为给定的访问许可。

perm:

S\_IRUGO : 任何人都可以读取该值，但不能修改；

S\_IRUGO | S\_IWUSR : 允许root用户修改该参数；

  注意：如果一个参数通过sysfs而被修改，则如同模块修改了这个参数值一样，但是内核不会以任何方式通知模块。多数情况下，我们不应该让模块参数是可写的，除非打算检测这中修改并作出相应的改动。

  带参数的模块驱动程序如下所示：

#include <linux/init.h>

#include <linux/module.h>

static char \*whom = "world";

static int howmany = 1;

module\_param(whom, charp, S\_IRUGO);

module\_param(howmany, int, S\_IRUGO);

static int \_\_init hello\_init(void)

{

int i = 0;

printk(KERN\_ALERT "Hello, world\n");

printk(KERN\_INFO "whom: %s, howmany: %d\n", whom, howmany);

for (i = 0; i < howmany; i++) {

printk(KERN\_INFO "i= %d, whom: %s\n", whom);

}

return 0;

}

static void \_\_exit hello\_exit(void)

{

printk(KERN\_ALERT "Goodbye, cruel world\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("Dual BSD/GPL");

  默认加载模块运行如下：

# insmod my\_module.ko

Hello, world

whom: world, howmany: 1

i = 0, whom: world

  修改参数值加载模块运行如下：

# insmod my\_module.ko whom="Hello world, so happy!" howmany=3

Hello, world

whom: Hello, howmany: 3

i = 0, whom: Hello

i = 1, whom: Hello

i = 2, whom: Hello

作者：micro虾米  
链接：https://www.jianshu.com/p/80c5928085a9  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 在用户空间编写驱动程序

相对于内核空间编程，用户空间编程具有自己的一些优点。有时候编写一个用户空间驱动程序是替代内核空间驱动程序的一个好方法。

用户空间驱动程序的优点：

1. 可以和整个C库链接。
2. 可以使用通常的调试器调试驱动程序代码，而不用费力地调试正在运行的内核。
3. 如果用户空间驱动程序挂起，则简单地杀掉进程就可以。驱动程序带来的问题不会挂起整个系统，除非所驱动的硬件一斤共发生严重故障。
4. 和内核内存不同，用户内存可以换出。如果驱动程序很大但是不经常使用，则除了正在使用的情况之外，不会占用太多内存。
5. 良好设计的驱动程序仍然支持对设备的并发访问。
6. 如果读者必须写封闭源码的驱动程序，则用户空间驱动程序可更加容易地避免因为修改内核接口而导致的不明确的许可问题。

   通常，用户空间的驱动程序被实现为一个服务器进程，其任务是替代内核作为硬件控制的而唯一代理。客户应用程序可连接到该服务器并和设备执行实际的通信；这样，好的驱动程序进程可允许对设备的并发访问。

用户空间驱动程序最重要的缺点：

1. 中断在用户空间中不可用。
2. 只有通过mmap映射/dev/mem才能直接访问内存，但只有特权用户才可以执行该操作。
3. 只有在调用ioperm或iopl后才可以访问I/O端口。并不是所有平台都支持这两个系统调用，并且访问/dev/port可能很慢，只有特权用户才能引用这些系统调用和访问设备文件。
4. 响应时间很慢。因为在客户端和硬件之间传递数据和动作需要上下文切换。
5. 如果驱动程序被换出到磁盘，响应时间会更慢。由于用户空间程序一般需要链接多个库，因此通常需要占用多个内存页。
6. 用户空间中不能处理一些非常重要的设备，包括网络接口和块设备等。

   有一种情况适合在用户空间处理，就是准备处理一种新的、不常见的硬件时。在用户空间中我们可以研究如何管理这个硬件而不用担心挂起整个系统。一旦完成，就可以很容易将用户空间驱动程序封装到内核模块中。

## 快速参考

Gh

### 模块构造-快速参考

**insmod**  
**modprobe**  
**rmmod**  
  用来装载模块到正运行的内核和移除模块的用户空间工具；  
**#include <linux/init.h>**  
**module\_init(init\_function);**  
**module\_exit(cleanup\_function);**  
  用于指定模块的初始化和清除函数的宏；  
**\_\_init**  
**\_\_initdata**  
**\_\_exit**  
**\_\_exitdata**  
  仅用于模块初始化或清除阶段的函数（\_\_init 和 \_\_exit）和数据（\_\_initdata 和 \_\_exitdata）标记。标记为初始化的项目会在初始化结束后丢弃；而退出项目在内和未被配置为可卸载模块的情况下被丢弃。内核通过将相应的目标对象放置在可执行文件的特殊ELF段中而让这些标记起作用。  
**#include <linux/sched.h>**  
  最重要的头文件之一。该文件包含驱动程序使用的大部分内核API的定义，包括睡眠函数以及各种变量声明；  
**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**  
  由内核构造系统使用的Makefile符号，用来确定在当前目录中应构造哪些模块；  
**/sys/module**  
**/proc/modules**  
  /sys/module是sysfs目录层次结构中包含当前已装载模块信息的目录。/proc/modules是早期用法。只在单个文件中包括这些信息，其中包含了模块名称、每个模块使用的内存总量以及使用计数等。每一行之后还追加有额外的字符串，用来指定模块的当前活动标记。  
**vermagic.o**  
  内核源代码目录中的一个目标文件，描述了模块的构造环境；  
**#include <linux/module.h>**  
  必须的头文件，必须包含在模块源代码中；  
**#include <linux/version.h>**  
  包含所构造内核版本信息的头文件；  
**LINUX\_VERSION\_CODE**  
  整数宏，在处理版本依赖的预处理条件语句中非常有用；  
**EXPORT\_SYMBOL(symbol);**  
**EXPORT\_SYMBOL\_GPL(symbol);**  
  用来导出单个符号到内核的宏，第二个宏将导出符号的使用限于GPL许可证下的模块；  
**MODULE\_AUTHOR(author);**  
**MODULE\_DESCRIPTION(description);**  
**MODULE\_VERSION(version\_string);**  
**MODULE\_DEVICE\_TABLE(table\_info);**  
**MODULE\_ALIAS(alternate\_name);**  
  在目标文件中添加关于模块的文档信息；  
**#include <linux/moduleparam.h>**  
**module\_param(variable, type, perm);**  
  用来创建模块参数的宏，用户可在装载模块时调整这些参数的值。其中的类型可以是bool、charp、int、invbool、long、short、ushort、uint、ulong、intarray；  
**#include <linux/kernel.h>**  
**int printk(const char \*fmt, ...);**  
  函数printf的内核代码；

作者：micro虾米  
链接：https://www.jianshu.com/p/e93fa589d3c2  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# DOING第三章 字符设备驱动程序

本章的目标是编写一个模块化的字符设备驱动程序。在本书中，取设备驱动程序名称为：scull，即 Simple Character Utility for Loading Localities，区域装载的简单字符工具。scull是一个操作内存区域的字符设备驱动程序，这片内存区域相当于一个设备。

  scull的优点在于它不和硬件相关，而只是操作从内核中分配的一些内存。scull可以随便编译和运行，scull可以被移植到Linux支持的计算机平台上，scull还可以展示内核和字符设备驱动程序之间的接口并让用户运行某些测试例程。但是scull设备在实际上并不会操作外部设备。

### scull的设计

  编写设备驱动程序的第一步是定义驱动程序为用户程序提供的机制。由于scull是计算机内存的一部分，所以它可以随意被操作，可以是顺序或随机存取设备，也可以是一个或多个设备等。

### 主设备号和次设备号

  对字符设备的访问是通过文件系统内的设备名称进行的，即特殊文件、设备文件，或者被称为文件系统树的节点，通常位于/dev目录。

  字符设备驱动程序的设备文件可通过ls -l命令输出的第一列中的字符'c'来识别。块设备也在/dev下，由字符'b'标识。

  执行 ls -l 命令，在设备文件的最后修改日期前有两个数（用逗号分隔），对于普通文件代表文件的长度，而对于设备文件，就是相应设备的主设备号和次设备号。

  通常，主设备号标识设备对应的驱动程序。现代的Linux内核允许多个驱动程序共享主设备号，但大多数设备按照“一个主设备号对应一个驱动程序”的原则。

  次设备号由内核使用，用于确定设备文件所指的设备。

#### 设备编号的内部表达

  在内核中，dev\_t类型用来保存设备编号--包括主设备号和次设备号，包含在头文件<linux/types.h>中。在内核Linux2.6.0版本中，dev\_t是一个32位的数，其中的12位用来表示主设备号，其余20位用来表示次设备号，设备编号应该使用<linux/kdev\_t.h>中定义的宏。获取dev\_t的主设备号或次设备号，应使用：

MAJOR(dev\_t dev);

MINOR(dev\_t dev);

如果需要将主设备号和次设备号转换成dev\_t类型，则使用：

MKDEV(int major, int minor);

#### 分配和释放设备编号

  在建立一个字符设备之前，驱动程序需要获得一个或多个设备编号。

##### 静态申请设备编号

  如果提前确定所需要的设备编号，可以通过register\_chrdev\_region()函数注册设备编号：

#include <linux/fs.h>

int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name);

参数说明：  
first: 要分配的设备编号范围的起始值；  
count: 所请求的连续设备编号的个数；  
name: 是和该编号范围关联的设备名称，将出现在/proc/devices和sysfs中；  
返回值：  
分配成功时返回0；  
错误时返回一个负的错误码，并且不能使用所请求的编号区域；

##### 动态申请设备编号

  如果不知道设备将要使用的设备编号，可以通过alloc\_chrdev\_region()函数注册设备编号：

#include <linux/fs.h>

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name);

参数说明：  
dev: 用于输出参数，函数执行成功时，将保存已分配范围的第一个设备编号；  
firstminor: 要使用的被请求的第一个次设备号，通常是0；  
count: 所请求的连续设备编号的个数；  
name: 和该编号范围关联的设备名称，将出现在/proc/devices和sysfs中；

##### 释放设备编号

  在不使用申请的设备编号时，要释放这些设备编号，通常在模块的清除函数中释放申请的设备编号；设备编号的释放需要使用以下函数实现：

#include <linux/fs.h>

void unregister\_chrdev\_region(dev\_t first, unsigned int count);

  通过以上方法，位驱动程序的使用分配设备编号，但是在用户空间程序访问设备编号，驱动程序需要将设备编号和内部函数连接起来，而这些内部函数用来实现设备的操作。

#### 动态分配主设备号

  部分主设备号已经静态地分配给了常用设备，在内核的Documentation/devices.txt文件中可以查到设备清单，驱动程序申请设备编号可以由以下两个选择：

1. 简单选定一个尚未被使用的主设备号；此方法适用于只有我们自己使用驱动程序的情况；
2. 通过动态方式分配主设备号；此方法适用于驱动程序被广泛使用时的情况；此时选定的主设备号可能造成冲突和麻烦；

  对于一个新的驱动程序，建议不要随便选择一个当前未使用的设备号作为主设备号，应该使用动态分配机制获取主设备号。驱动程序应始终使用alloc\_chrdev\_region()函数，而不是register\_chrdev\_region()函数。

  动态分配主设备号的缺点：由于分配的主设备号不能保证始终一致，无法预先创建对应的设备节点。一旦分配了设备号，就可以从/proc/devices中读取到。

  为了能够加载使用动态主设备号的设备驱动程序，可以使用脚本代替insmod命令加载程序，该脚本在调用insmod之后，读取到/proc/devices以获取新分配的主设备号，然后创建对应的设备文件。

。。。。。。

  反复创建和删除/dev节点有些不必要，如果只是装载和卸载单个驱动程序，则可在第一次创建设备文件之后仅使用rmmod和insmod两个命令；因为动态设备号并不是随机生成的，如果不受其他动态模块影响，可以获取到相同的动态主设备号；但是这个技巧不适用于同时有多个驱动程序存在的场合。

  分配主设备号的**最佳方式是**：默认采用动态分配，同时保留在加载甚至是编译时指定主设备号的余地。scull驱动程序的实现方式：使用一个全局变量scull\_major保存所选择的设备号，一个全局变量scull\_minor保存次设备号。scull\_major默认值为SCULL\_MAJOR，该宏定义在scull.h中，SCULL\_MAJOR默认值为0，即采用动态分配设备号。这样既可以在编译前修改SCULL\_MAJOR宏定义，也可以通过insmod命令行指定scull\_major的值。

  在scull驱动中获取主设备号的代码，如下所示：

if (scull\_major) {

dev = MKDEV(scull\_major, scull\_minor);

result = register\_chrdev\_region(dev, scull\_nr, "scull");

} else {

result = alloc\_chrdev\_region(&dev, scull\_minor, scull\_nr, "scull");

scull\_major = MAJOR(dev);

}

if (result < 0) {

printk(KERN\_WARNING "scull: can't get major %d\n", scull\_major);

return result;

}

作者：micro虾米  
链接：https://www.jianshu.com/p/98dfb4067f88  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 一些重要的数据结构

### 重要的数据结构

  上一节中设备编号的注册仅仅是驱动程序代码必须完成的许多工作中的第一件事。大部分基本大的驱动程序操作涉及到三个重要的内核数据结构，分别是file\_operations、file、inode。

#### 文件操作file\_operations结构

  file\_operations结构用来将所有驱动程序操作连接到设备编号。该结构定义在<linux/fs.h>中，其中包含了一组函数指针。每个打开的文件，和一组函数关联；文件在内部由一个file结构表示，函数组包含在指向一个file\_operations结构的fops字段。这些操作主要用来实现系统调用，如：read、write等。可以认为文件是一个“对象”，而操作它的函数是“方法”。

  file\_operations结构或者指向这类结构的指针称为fops，这个结构中的每一个字段都必须指向驱动程序中实现特定操作的函数，对于不支持的操作，对应的字段可置为NULL值。

  \_\_user字符串，表明指针是一个用户空间地址，因此不能被直接引用，对编译来说，并没有任何效果。

。。。。。

  scull设备驱动程序所实现的只是几个重要的设备方法，如下初始化操作：

struct file\_operations fops = {

.owner = THIS\_MODULE,

.llseek = scull\_llseek,

.read = scull\_read,

.write = scull\_write,

.ioctl = scull\_ioctl,

.open = scull\_open,

.release = scull\_release,

};

  这个声明操作采用了标准C的标记化结构初始化语法。标记化的初始化方法允许对结构成员进行重新排序。某些场合下，将频繁被访问的成员放在相同的硬件缓存行上，将大大提高性能。

#### file结构

  file结构代表一个打开的文件，它并不仅限定于设备驱动程序，系统中每个打开的文件在内核空间都有一个对应的file结构。file结构由内核在open时创建，并传递给在该文件上进行操作的所有函数，直到最后的close函数。在文件的所有实例都被关闭之后，内核会释放这个file数据结构。定义在头文件<linux/fs.h>中，struct file是一个内核结构，不会出现在用户程序中。

  常用的结构成员如下所示：

mode\_t f\_mode;

文件模式。通过FMODE\_READ和FMODE\_WRITE位来标识文件是否可读或可写或可读写。由于内核在调用驱动程序的read和write前已经检查了访问权限，所以驱动程序不必为这两个方法检查权限。

loff\_t f\_ops;

当前的读/写位置。loff\_t是一个64位的数。用来修改文件位置。驱动程序可以读取这个值来获取文件中的当前位置。

unsigned int f\_flags;

文件标志。如：O\_RDONLY、O\_NONBLOCK、O\_SYNC。为了检查用户请求的是否是阻塞式操作。驱动程序需要检查O\_NONBLOCK标志。注意，检查读/写权限应该查看f\_mode而不是f\_flags。标志定义在头文件<linux/fcntl.h>中。

struct file\_operations \*f\_op;

与文件相关的操作。内核在执行open操作时对这个指针赋值，以后需要处理这些操作时就读取这个指针。

void \*private\_data;

open系统调用在调用驱动程序的open方法前将这个指针置为NULL。private\_data是跨系统调用时保存状态信息的非常有用的资源。

#### inode结构

  内核用inode结构在内部表示文件，和file结构不同，file结构表示打开的文件描述符。对单个文件，可能会由许多个表示打开的文件描述符的file结构，但它们都指向单个inode结构。

  inode结构中以下两个字段对编写驱动程序代码比较重要：

dev\_t i\_rdev;

// 对表示设备文件的inode结构，该字段包含了真正的设备编号；

struct cdev \\*i\_cdev;

// struct cdev是表示字符设备的内核的内部结构。当inode指向一个字符设备文件时，该字段包含了指向struct cdev结构的指针；

  从一个inode中获得主设备号和次设备号：

unsigned int imajor(struct inode \*inode);

unsigned int iminor(struct inode \*inode);

  为了防止因为内核版本升级引起的不兼容问题，应该使用上面的两个宏，而不是直接操作i\_rdev。

作者：micro虾米  
链接：https://www.jianshu.com/p/e54f1f69cced  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 字符设备的注册

Df

### 字符设备的注册

  内核内部使用struct cdev结构来表示字符设备，在内核调用设备的操作之前，必须分配注册一个或多个数据结构，参考上节重要的数据结构，因该包含头文件<linux/cdev.h>。

  分配和初始化struct cdev的方式有两种，

struct cdev \*my\_cdev = cdev\_alloc();

my\_cdev->ops = &my\_fops;

  初始化一分配到的cdev结构：

void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops);

  cdev结构的所有者字段，应该设置为THIS\_MODULE：

.owner = THIS\_MODULE

  cdev结构设置好之后，需要告诉内核该结构的信息：

int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

dev: 是cdev结构

num: 是该设备对应的第一个设备编号

count: 是应该和该设备关联的设备编号的数量，经常取count = 1

在某些情况下，会有多个设备编号对应于一个特定的设备；

  在使用cdev\_add()函数时应注意，这个调用可能会失败，如果返回一个负的错误码，该设备不会被添加到系统中。因此，在驱动程序还没有完全准备好处理设备上的操作时，就不能调用cdev\_add()函数。

  从系统中移除一个字符设备：

void cdev\_del(struct cdev \*dev);

  调用cdev\_del()函数之后，就不应该再访问cdev结构了。

#### scull中的设备注册

  在scull驱动中，通过struct scull\_dev结构来表示每个设备：

struct scull\_dev {

struct scull\_qset \*data; // 指向第一个量子集的指针

int quantum; // 当前量子的大小

int qset; // 当前数组的大小

unsigned long size; // 保存在其中的数据总量

unsigned int access\_key; // 由sculluid和sculloriv使用

struct semaphore sem; // 互斥信号量

struct cdev cdev; // 字符设备结构

}

  在scull驱动中，将struct cdev结构初始化并添加到系统中的操作如下：

static void scull\_setup\_cdev(struct scull\_dev \*dev, int index)

{

int err;

dev\_t devno = MKDEV(scull\_major, scull\_minor + index);

cdev\_init(&dev->cdev, &scull\_fops);

dev->cdev.owner = THIS\_MODULE;

dev->cdev.ops = &scull\_fops;

err = cdev\_add(&dev->cdev, devno, 1);

if (err)

printk(KERN\_NOTICE "Error %d adding scull: %d\n", err, index);

}

#### 早期的办法

  在内核Linux-2.6之前的版本中，字符设备驱动程序没有使用cdev接口，而是使用老的接口，在新的代码中不应该使用老的接口，因为这种机制会在将来的内核中消失。

  注册一个字符设备驱动程序的经典方式是：

int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);

major: 是设备的主设备号

name: 是驱动程序的名称（出现在/proc/devices中）

fops: 是默认的file\_operations结构

  register\_chrdev()函数，可实现静态、动态注册：

**动态注册**，即major = 0时，类似alloc\_chrdev\_region()函数，系统会动态为设备分配一个未被使用的主设备号，并将分配的主设备号作为返回值；

**静态注册**，即major不为0，而是由驱动开发者指定，类似register\_chrdev\_region()函数，系统会检测指定的主设备号是否已经被使用，如果没有被使用，则分配给该设备使用，如果已经被占用，就返回一个错误值。

  对register\_chrdev()函数的调用将为给定的主设备号注册0~255作为次设备号，并为每一个设备建立一个对应的默认cdev结构，不能使用大于255的主设备号和次设备号。

  注销一个已经注册的字符设备驱动程序，即将已经注册的设备从系统中移除的方式：

int unregister\_chrdev(unsigned int major, const char \*name);

// major和name必须和传递给register\_chrdev()函数的参数值保持一致，否则该调用会失败。

作者：micro虾米  
链接：https://www.jianshu.com/p/01ae7d527d97  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## open和release

## scull的内存使用

## read和write

## 试试新设备

## 快速参考

# 第四章 调试技术

## 内核中的调试支持

## 通过打印调试

## 通过查询调试

## 通过监视调试

## 调试系统故障

## 调试器和相关工具

# 第五章 并发和竞态

## scull的缺陷

## 并发及其管理

## 信号量和互斥体

## completiOn

## 自旋锁

## 锁陷阱

## 除了锁之外的办法

## 快速参考

# 第六章 高级字符驱动程序操作

## ioctl

## 阻塞型I/O

## poll和select

## 异步通知

## 定位设备

## 设备文件的访问控制

## 快速参考

# 第七章 时间、延迟及延缓操作

## 度量时间差

## 获取当前时间

## 延迟执行

## 内核定时器

## tasklet

## 工作队列

## 快速参考

# 第八章 分配内存

## kmalloc函数的内幕

## 后备高速缓存

## get—free—page和相关函数

## vmalloc及其辅助函数

## per-CPU变量

## 获取大的缓冲区

## 快速参考

# REF

Linux设备驱动程序学习----目录

https://www.jianshu.com/p/c71eef23873c