第 3 章 字符设备驱动程序

本章的目标是编写一个完整的字符设备驱动程序。由于这类驱动程序适合于大多数简单的硬件设备，我们首先开放一个字符设备驱动程序。字符也相对比较好理解，比如说块设备驱动程序。我们的最终目标是写一个模块化的字符设备驱动程序，但本章我们不再讲述有关模块化的问题。

本章通篇都是从一个真实的设备驱动程序截取出的代码块：这个设备就是 scull，是“Simple Character Utility for Loading Localities”的缩写。尽管 scull 是一个设备，但它却是操作内存的字符设备。这种情况的一个副作用就是，只要涉及 scull，“设备”这个词就可以同“scull 使用的内存区”互换使用。

scull 的优点是，由于每台电脑都有内存，所以它与硬件无关。scull 用 kmalloc 分配内存，而且仅仅操作内存。任何人都可以编译和运行 scull，而且 scull 可以移植到所有 Linux 支持的平台上。但另一方面，除了演示内核于字符设备驱动程序间的交互过程，可以让用户运行某些测试例程外，scull 做不了“有用的”事。

scull 的设计

编写设备驱动程序的第一步就是定义驱动程序提供给用户程序的能力（“机制”）。由于我们的“设备”是电脑内存的一部分，我做什么都可以。它可以是顺便存取设备，也可以是随机存取设备，可以是一个设备，也可以是多个，等等。

为了是 scull 更有用，可以成为编写真实设备的驱动程序的模板，我将向你展示如何在电脑的内存之上实现若干设备抽象操作，每一种操作都有自己的特点。

scull 的源码实现如下设备。由模块实现的每一种设备都涉及一种类型：

scull0-3

4 个设备，共保护了 4 片内存区，都是全局性的和持久性的。“全局性”是指，如果打开设备多次，所有打开它的文件描述符共享其中的数据。“持久性”是指，如果设备关闭后再次打开，数据不丢失。由于可以使用常用命令访问这个设备，如 cp，cat 以及 shell I/O 重定向等，这个设备操作非常有趣；本章将深入探讨它的内部结构。

scullpipe0-3

4 个“fifo”设备，操作起来有点象管道。一个进程读取另一个进程写入的数据。如果有多个进程读同一个设备，他们彼此间竞争数据。通过 scullpipe 的内部结构可以了解阻塞型和非阻塞型读/写是如何实现的；没有中断也会出现这样的情况。尽管真实的驱动程序利用中断与它们的设备同步，但阻塞型和非阻塞型操作是非常重要的内容，从概念上讲与中断处理（第 9 章，中断处理，介绍）无关。

scullsingle

scullpriv

sculluid

scullwuid

这些设备与 scull0 相似，但在何时允许 open 操作时都不同方式的限制。第一个（scullsingle）只允许一次一个进程使用驱动程序，而 scullpriv 对每个虚拟控制台是私有的（每个设备对虚拟控制台是私有的）。sculluid 和 scullwuid 可以多次打开，但每次只能有一个用户；如果另

一个用户锁住了设备，前者返回-EBUSY，而后者则实现为阻塞型 open。通过这些可以展示如何实现不同的访问策略。
每一个 scull 设备都展示了驱动程序不同的功能，而且都不同的难度。本章主要讲解 scull0-3 的内部结构；第 5 章，字符设备驱动程序的扩展操作，将介绍更复杂的设备： “一个样例实现：scullpipe”介绍 scullpipe，“设备文件的访问控制”介绍其他设备。

主设备号和次设备号

通过访问文件系统的名字（或“节点”）访问字符设备，通常这些文件位于/dev 目录。设备文件是特殊文件，这一点可以通过 ls -l 输出的第一列中的“c”标明，它说明它们是字符节点。/dev 下还有块设备，但它们的第一列是“b”; 尽管如下介绍的某些内容也同样适用于块设备，现在我们只关注字符设备。如果你执行 ls 命令，在设备文件条目的最新修改日期前你会看到两个数（用逗号分隔），这个位置通常显示文件长度。这些数就是相应设备的主设备号和次设备号。下面的列表给出了我使用的系统上的一些设备。它们的主设备号是 10，1 和 4，而次设备号是 0，3，5，64-65 和 128-129。
（代码）
主设备号标识设备对应的驱动程序。例如，/dev/null 和/dev/zero 都有驱动程序 1 管理，而所有的 tty 和 pty 都由驱动程序 4 管理。内核利用主设备号将设备与相应的驱动程序对应起来。次设备号只由设备驱动程序使用；内核的其他部分不使用它，仅将它传递给驱动程序。一个驱动程序控制若干个设备并不为奇（如上面的例子所示）－－次顺便号提供了一种区分它们的方法。
向系统增加一个驱动程序意味着要赋予它一个主设备号。这一赋值过程应该在驱动程序（模块）的初始化过程中完成，它调用如下函数，这个函数定义在<linux/fs.h>：
（代码）
返回值是错误码。当出错时返回一个负值；成功时返回零或正值。参数 major 是所请求的主设备号，name 是你的设备的名字，它将在/proc/devices 中出现，fops 是一个指向跳转表的指针，利用这个跳转表完成对设备函数的调用，本章稍后将在“文件操作”一节中介绍这些函数。
主设备号是一个用来索引静态字符设备数组的整数。在 1.2.13 和早期的 2.x 内核中，这个数组有 64 项，而 2.0.6 到 2.1.11 的内核则升至 128。由于只有设备才处理次设备号，register_chrdev 不传递次设备号。
一旦设备已经注册到内核表中，无论何时操作与你的设备驱动程序的主设备号匹配的设备文件，内核都会通过在 fops 跳转表索引调用驱动程序中的正确函数。
接下来的问题就是如何给程序一个它们可以请求你的设备驱动程序的名字。这个名字必须插入到/dev 目录中，并与你的驱动程序的主设备号和次设备号相连。
在文件系统上创建一个设备节点的命令是 mknod，而且你必须是超级用户才能创建设备。除了要创建的节点名字外，该命令还带三个参数。例如，命令：
（代码）
创建一个字符设备（c），主设备号是 127，次设备号是 0。由于历史原因，次设备号应该在 0-255 范围内，有时它们存储在一个字节中。存在很多原因扩展可使用的次设备号的范围，但就现在而言，仍然有 8 位限制。

动态分配主设备号

某些主设备号已经静态地分配给了大部分公用设备。在内核源码树的Documentation/device.txt 文件中可以找到这些设备的列表。由于许多数字已经分配了，为新设备选择一个唯一的号码是很困难的－－不同的设备要不主设备号多得多。
很幸运（或是感谢某些人天才），你可以动态分配主设备号了。如果你调用 register_chrdev 时的 major 为零的话，这个函数就会选择一个空闲号码并做为返回值返回。主设备号总是正的，因此不会和错误码混淆。
我强烈推荐你不要随便选择一个一个当前不用的设备号做为主设备号，而使用动态分配机制获取你的主设备号。
动态分配的缺点是，由于分配给你的主设备号不能保证总是一样的，无法事先创建设备节点。然而这不是什么问题，这是因为一旦分配了设备号，你就可以从/proc/devices 读到。为了加载一个设备驱动程序，对 insmod 的调用被替换为一个简单的脚本，它通过/proc/devices 获得新分配的主设备号，并创建节点。
/proc/devices 一般如下所示：
（代码）
加载动态分配主设备号驱动程序的脚本可以利用象 awk 这类工具从/proc/devices 中获取信息，并在/dev 中创建文件。
下面这个脚本，scull_load，是 scull 发行中的一部分。使用以模块形式发行的驱动程序的用户可以在/etc/rc.d/rc.local 中调用这个脚本，或是在需要模块时手工调用。此外还有另一种方法：使用 kerneld。这个方法和其他模块的高级功能将在第 11 章“Kerneld 和高级模块化”中介绍。
（代码）
这个脚本同样可以适用于其他驱动程序，只要重新定义变量和调整 mknod 那几行就可以了。上面那个脚本创建 4 个设备，4 是 scull 源码中的默认值。
脚本的最后两行看起来有点怪怪的：为什么要改变设备的组和权限呢？原因是这样的，由 root 创建的节点自然也属于 root。默认权限位只允许 root 对其有写访问权，而其他只有读权限。正常情况下，设备节点需要不同的策略，因此需要进行某些修改。通常允许一组用户访问对设备，但实现细节却依赖于设备和系统管理员。安全是个大问题，这超出了本书的范围。scull_load 中的 chmod 和 chgrp 那两行仅仅是最为处理权限问题的一点提示。稍后，在第 5 章的“设备文件的访问控制”一节中将介绍 sculluid 源码，展示设备驱动程序如何实现自己的设备访问授权。
如果重复地创建和删除/dev 节点似乎有点过分的话，有一个解决的方法。如果你看了内核源码 fs/devices.c 的话，你可以看到动态设备号是从 127（或 63）之后开始的，你可以用 127 做为主设备号创建一个长命节点，同时可以避免在每次相关设备加载时调用脚本。如果你使用了几个动态设备，或是新版本的内核改变了动态分配的特性，这个技巧就不能用了。（如果内核发生了修改，基于内核内部结构编写的代码并不能保证继续可以工作。）不管怎样，由于开发期间模块要不断地加载和卸载，你会发现这一技术在开发期间还是很有用的。
就我看来，分配主设备号的最佳方式是，默认采用动态分配，同时留给你在加载时，甚至是编译时，指定主设备号的余地。使用我建议的代码将与自动端口探测的代码十分相似。scull 的实现使用了一个全局变量，scull_major，来保存所选择的设备号。该变量的默认值是 SCULL_MAJOR，在所发行的源码中为 0，即“选择动态分配”。用户可以使用这个默认值或选择某个特定的主设备号，既可以在编译前修改宏定义，也可以在 ins_mod 命令行中指定。

最后，通过使用 scull_load 脚本，用户可以在 scull_load 中命令行中将参数传递给 insmod。这里是我在 scull.c 中使用的获取主设备号的代码：
（代码）

从系统中删除设备驱动程序

当从系统中卸载一个模块时，应该释放主设备号。这一操作可以在 cleanup_module 中调用如下函数完成：
（代码）
参数是要释放的主设备号和相应的设备名。内核对这个名字和设备号对应的名字进行比较：如果不同，返回-ENINVAL。如果主设备号超出了所允许的范围或是并未分配给这个设备，内核一样返回-EINVAL。在cleanup_module中注销资源失败会有非常不号的后果。下次读取/proc/devices时，由于其中一个name字串仍然指向模块内存，而那片内存已经不存在了，系统将产生一次失效。这种失效称为Oops*，内核在访问无效地址时将打印这样的消息。
当你卸载驱动程序而又无法注销主设备号时，这种情况是无法恢复的，即便为此专门写一个“补救”模块也无济于事，因为 unregister_chrdev 中调用了 strcmp，而 strcmp 将使用未映射的 name 字串，当释放设备时就会使系统 Oops。无需说明，任何视图打开这个异常的设备号对应的设备的操作都会 Oops。
除了卸载模块，你还经常需要在卸载驱动程序时删除设备节点。如果设备节点是在加载时创建的，可以写一个简单的脚本在卸载时删除它们。对于我们的样例设备，脚本 scull_unload 完成这个工作。如果动态节点没有从/dev 中删除，就会有可能造成不可预期的错误：如果动态分配的主设备号相同，开发者计算机上的一个空闲/dev/framegrabber 就有可能在一个月后引用一个火警设备。“没有这个文件或目录”要比这个新设备所产生的后果要好得多。

dev_t 和 kdev_t

到目前为止，我们已经谈论了主设备号。现在是讨论次设备号和驱动程序如何使用次设备号来区分设备的时候了。
每次内核调用一个设备驱动程序时，它都告诉驱动程序它正在操作哪个设备。主设备号和次设备号合在一起构成一个数据类型并用来标别某个设备。设备号的组合（主设备号和次设备号合在一起）驻留在稍后介绍的“inode”结构的 i_rdev 域中。每个驱动程序接收一个指向 struct inode 的指针做为第一个参数。这个指针通常也称为 inode，函数可以通过查看 inode->i_rdev 分解出设备号。
历史上，Unix 使用 dev_t 保存设备号。dev_t 通常是<sys/types.h>中定义的一个 16 位整数。而现在有时需要超过 256 个次设备号，但是由于有许多应用（包括 C 库在内）都了解 dev_t 的内部结构，改变 dev_t 是很困难的，如果改变 dev_t 的内部结构就会造成这些应用无法运行。因此，dev_t 类型一直没有改变；它仍是一个 16 位整数，而且次设备号仍限制在 0-255 内。然而，在 Linux 内核内部却使用了一个新类型，kdev_t。对于每一个内核函数来说，这个新类型被设计为一个黑箱。它的想法是让用户程序不能了解 kdev_t。如果 kdev_t 一直是隐藏的，它可以在内核的不同版本间任意变化，而不必修改每个人的设备驱动程序。
有关 kdev_t 的信息被禁闭在<linux/kdev_t.h>中，其中大部分是注释。如果你对代码后的哲

* 怪里怪气的Linux爱好者将“Oops”这个词既当成名词也当成动词使用。

学感兴趣的话，这个头文件是一段很有指导性的代码。因为<linux/fs.h>已经包含了这个头文件，没有必要显式地包含这个文件。
不幸的是，kdev_t 类型是一个“现代”概念，在内核版本 1.2 中没有这个类型。在较新的内核中，所有的引用设备的内核变量和结构字段都是 kdev_t 的，但是在 1.2.13 中同样的变量却是 dev_t 的。如果你的驱动程序只使用它接收的结构字段，而不声明自己的变量的话，这不会有什么问题的。如果你需要声明自己的设备类型变量，为了可移植性你应该在你的头文件中加入如下几行：
（代码）
这段代码是样例源码中的 sysdep.h 头文件的一部分。我不会在源码中在引用 dev_t，但是要假设前一个条件语句已经执行了。
如下这些宏和函数是你可以对 kdev_t 执行的操作：
MAJOR(kdev_t dev);
从 kdev_t 结构中分解出主设备号。
MINOR(kdev_t dev);
分解出次设备号。
MKDEV(int ma, int mi);
通过主设备号和次设备号返回 kdev_t。
kdev_t_to_nr(kdev_t dev);
将 kdev_t 转换为一个整数（dev_t）。
to_kdev_t(int dev);
将一个整数转换为 kdev_t。注意，核心态中没有定义 dev_t，因此使用了 int。
与 Linux 1.2 相关的头文件定义了同样的操作 dev_t 的函数，但没有那两个转换函数，这也就是为什么上面那个条件代码简单地将它们定义返回它们的参数值。

文件操作

在接下来的几节中，我们将看看驱动程序能够对它管理的设备能够完成哪些不同的操作。在内核内部用一个 file 结构标别设备，而且内核使用 file_operations 结构访问驱动程序的函数。这一设计是我们所看到的 Linux 内核面向对象设计的第一个例证。我们将在以后看到更多的面向对象设计的例证。file_operations 结构是一个定义在<linux/fs.h>中的数指针表。结构 struct file 将在以后介绍。
我们已经 register_chrdev 调用中有一个参数是 fops，它是一个指向一组操作（open，read 等等）表的指针。这个表的每一个项都指向由驱动程序定义的处理相应请求的函数。对于你不支持的操作，该表可以包含 NULL 指针。对于不同函数的 NULL 指针，内核具体的处理行为是不同的，下一节将逐一介绍。
随着新功能不断加入内核，file_operations结构已逐渐变得越来越大（尽管从 1.2.0 到 2.0.x并没有增加新字段）。这种增长应该不会有什么副作用，因为在发现任何尺寸不匹配时，C编译器会将全局或静态struct变量中的未初始化字段填 0。新的字段都加到结构的末尾*，所以在编译时会插入一个NULL指针，系统会选择默认行为（记住，对于所有模块需要加载的新内核，都要重新编译一次模块）。
在 2.1 开发用内核中，有些与 fops 字段相关的函数原型发生了变化。这些变化将在第 17 章“近期发展”的“文件操作”一节中介绍。

* 例如，版本 2.1.31 增加一个称为lock的新字段。

纵览不同操作

下面的列表将介绍应用程序能够对设备调用的所有操作。这些操作通常称为“方法”，用面向对象的编程术语来说就是说明一个对象声明可以操作在自身的动作。
为了使这张列表可以用来当作索引，我尽量使它简洁，仅仅介绍每个操作的梗概以及当使用 NULL 时的内核默认行为。你可以在初次阅读时跳过这张列表，以后再来查阅。
在介绍完另一个重要数据结构（file）后，本章的其余部分将讲解最重要的一些操作并提供一些提示，告诫和真实的代码样例。由于我们尚不能深入探讨内存管理和异步触发机制，我们将在以后的章节中介绍这些更为复杂操作。
struct file_operations 中的操作按如下顺序出现，除非注明，它们的返回 0 时表示成功，发生错误时返回一个负的错误编码：
int (*lseek)(struct inode *, struct file *, off_t, int);
方法 lseek 用来修改一个文件的当前读写位置，并将新位置做为（正的）返回值返回。出错时返回一个负的返回值。如果驱动程序没有设置这个函数，相对与文件尾的定位操作失败，其他定位操作修改 file 结构（在“file 结构”中介绍）中的位置计数器，并成功返回。2.1.0 中该函数的原型发生了变化，第 17 章“原型变化”将讲解这些内容。
int (*read)(struct inode *, struct file *, char *, int);
用来从设备中读取数据。当其为 NULL 指针时将引起 read 系统调用返回-EINVAL（“非法参数”）。函数返回一个非负值表示成功的读取了多少字节。
int (*write)(struct inode *, struct file *, const char *, int);
向设备发送数据。如果没有这个函数，write 系统调用向调用程序返回一个-EINVAL。注意，版本 1.2 的头文件中没有 const 这个说明符。如果你自己在 write 方法中加入了 const，当与旧头文件编译时会产生一个警告。如果你没有包含 const，新版本的内核也会产生一个警告；在这两种情况你都可以简单地忽略这些警告。如果返回值非负，它就表示成功地写入的字节数。
int (*readdir)(struct inode *, struct file *, void *, filldir_t);
对于设备节点来说，这个字段应该为 NULL；它仅用于目录。
int (*select)(struct inode *, struct file *, int, select_table *);
select 一般用于程序询问设备是否可读和可写，或是否一个“异常”条件发生了。如果指针为 NULL，系统假设设备总是可读和可写的，而且没有异常需要处理。“异常”的具体含义是和设备相关的。在当前的 2.1 开发用内核中，select 的实现方法完全不同。（见第 17 章的“poll 方法”）。返回值告诉系统条件满足（1）或不满足（0）。
int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
系统调用 ioctl 提供一中调用设备相关命令的方法（如软盘的格式化一个磁道，这既不是读操作也不是写操作）。另外，内核还识别一部分 ioctl 命令，而不必调用 fops 表中的 ioctl。如果设备不提供 ioctl 入口点，对于任何内核没有定义的请求，ioctl 系统调用将返回-EINVAL。当调用成功时，返回给调用程序一个非负返回值。
int (*mmap)(struct inode *, struct file *, struct vm_area_struct *);
mmap 用来将设备内存映射到进程内存中。如果设备不支持这个方法，mmap 系统调用将返回-ENODEV。
int (*open)(struct inode *, struct file *);
尽管这总是操作在设备节点上的第一个操作，然而并不要求驱动程序一定要声明这个方法。如果该项为 NULL，设备的打开操作永远成功，但系统不会通知你的驱动程序。
void (*release)(struct inode *, struct file *);

当节点被关闭时调用这个操作。与 open 相仿，release 也可以没有。在 2.0 和更早的核心中，close 系统调用从不失败；这种情况在版本 2.1.31 中有所变化（见第 17 章）。

int (*fsync)(struct inode *, struct file *);

刷新设备。如果驱动程序不支持，fsync 系统调用返回-EINVAL。

int (*fasync)(struct inode *, struct file *, int);

这个操作用来通知设备它的 FASYNC 标志的变化。异步触发是比较先进的话题，将在第 5 章的“异步触发”一节中介绍。如果设备不支持异步触发，该字段可以是 NULL。

int (*check_media_change)(kdev_t dev);

check_media_change 只用于块设备，尤其是象软盘这类可移动介质。内核调用这个方法判断设备中的物理介质（如软盘）自最近一次操作以来发生了变化（返回 1）或是没有（0）。字符设备无需实现这个函数。

int (*revalidate)(kdev_t dev);

这是最后一项，与前面提到的那个方法一样，也只适用于块设备。revalidate 与缓冲区高速缓存有关。我们将在第 12 章“加载块设备驱动程序”的“可移动设备”中介绍 revalidate。

scull 驱动程序中适用的 file_operations 结构如下：

（代码）

在最新的开发用内核中，某些原型已经发生了变化。该列表是从 2.0.x 的头文件中提炼出来的，这里给出的原型对于大多数内核而言都是正确的。内核 2.1 引入的变化（以及为了使我们的模块可移植所进行的修改）在针对不同操作的每一节和第 17 章的“文件操作”中详细介绍。

file 结构

在<linux/fs.h>中定义的 struct file 是设备驱动程序所适用的又一个最重要的数据结构。注意，file 与用户程序中的 FILE 没有任何关联。FILE 是在 C 库中定义且从不出现在内核代码中。而 struct file 是一个内核结构，从不出现在用户程序中。

file 结构代表一个“打开的文件”。它有内核在 open 时创建而且在 close 前做为参数传递给如何操作在设备上的函数。在文件关闭后，内核释放这个数据结构。一个“打开的文件”与由 struct inode 表示的“磁盘文件”有所不同。

在内核源码中，指向 struct file 的指针通常称为 file 或 filp（“文件指针”）。为了与这个结构相混淆，我将一直称指针为 filp－flip 是一个指针（同样，它也是设备方法的参数之一），而 file 是结构本身。

struct file 中的最重要的字段罗列如下。与上节相似，这张列表在首次阅读时可以略过。在下一节中，我们将看到一些真正的 C 代码，我将讨论某些字段，到时你可以反过来查阅这张列表。

mode_t f_mode;

文件模式由 FMODE_READ 和 FMODE_WRITE 标别。你可能需要在你的 ioctl 函数中查看这个域来来检查读/写权限，但由于内核在调用你的驱动程序的 read 和 write 前已经检查了权限，你无需检查在这两个方法中检查权限。例如，一个不允许的写操作在驱动程序还不知道的情况下就被已经内核拒绝了。

loff_t f_ops;

当然读/写位置。loff_t 是一个 64 位数值（用 gcc 的术语就是 long long）。如果驱动程序需要知道这个值，可以直接读取这个字段。如果定义了 lseek 方法，应该更新 f_pos 的值。当传输数据时，read 和 write 也应该更新这个值。

unsigned short f_flags;
文件标志，如 O_RDONLY，O_NONBLOCK 和 O_SYNC。驱动程序为了支持非阻塞型操作需要检查这个标志，而其他标志很少用到。注意，检查读/写权限应该查看 f_mode 而不是 f_flags。所有这些标志都定义在<linux/fcntl.h>中。
struct inode *f_inode;
打开文件所对应的 i 节点。inode 指针是内核传递给所有文件操作的第一个参数，所以你一般不需要访问 file 结构的这个字段。在某些特殊情况下你只能访问 struct file 时，你可以通过这个字段找到相应的 i 节点。
struct file_operations *f_op;
与文件对应的操作。内核在完成 open 时对这个指针赋值，以后需要分派操作时就读这些数据。filp->f_op 中的值从不保存供以后引用；这也就是说你可以在需要的事后修改你的文件所对应的操作，下一次再操作那个打开文件的相应操作时就会调用新方法。例如，主设备号为 1 的设备（/dev/null，/dev/zero 等等）的 open 代码根据要打开的次设备号替换 filp->f_op 中的操作。这种技巧有助于在不增加系统调用负担的情况下方便识别主设备号相同的设备。能够替换文件操作的能力在面向对象编程技术中称为“方法重载”。
void *private_data;
系统调用 open 在调用驱动程序的 open 方法前将这个指针置为 NULL。驱动程序可以将这个字段用于任意目的或者忽略简单忽略这个字段。驱动程序可以用这个字段指向已分配的数据，但是一定要在内核释放 file 结构前的 release 方法中清除它。private_data 是跨系统调用保存状态信息的非常有用的资源，在我们的大部分样例都使用了这个资源。
实际的结构里还有其他一些字段，但它们对于驱动程序并不是特别有用。由于驱动程序从不填写 file 结构；它们只是简单地访问别处创建的结构，我们可以大胆地忽略这些字段。

Open 和 Close

现在让我们已经走马观花地看了一遍这些字段，下面我们将开始在实际的 scull 函数中使用这些字段。

Open 方法

open 方法是驱动程序用来为以后的操作完成初始化准备工作的。此外，open 还会增加设备计数，以便防止文件在关闭前模块被卸载出内核。
在大部分驱动程序中，open 完成如下工作：

- 检查设备相关错误（诸如设备未就绪或相似的硬件问题）。
- 如果是首次打开，初始化设备。
- 标别次设备号，如有必要更新 f_op 指针。
- 分配和填写要放在 filp->private_data 里的数据结构。
- 增加使用计数。

在 scull 中，上面的大部分操作都要依赖于被打开设备的次设备号。因此，首先要做的事就是标别要操作的是哪个设备。我们可以通过查看 inode->i_rdev 完成。
我们已经谈到内核是如何不使用次设备号的了，因此驱动程序可以随意使用次设备号。事实上，利用不同的次设备号访问不同的设备，或以不同的方式打开同一个设备。例如，/dev/ttyS0 和/dev/ttyS1 是两个不同的串口，而/dev/cua0 的物理设备与/dev/ttyS0 相同，仅仅是操作行为

不同。cua是“调出”设备；它们不是终端，而且它们也没有终端所需要的所有软件支持（即，它们没有加入行律*）。所有的串口设备都有许多不同的次设备号，这样驱动程序就区分它们了：ttyS与cua不一样。

驱动程序从来都不知道被打开的设备的名字，它仅仅知道设备号——而且用户可以按照自己的规范给用设备起别名，而完全不用原有的名字。如果你看看/dev 目录就会知道，你将发现对应相同主/次设备号的不同名字；设备只有一个而且是相同的，而且没有方法区分它们。例如，在许多系统中，/dev/psaux 和/dev/bmouseps2 都存在，而且它们有同样的设备号；它们可以互换使用。后者是“历史遗迹”，你的系统里可以没有。

scull驱动程序是这样使用次设备号的：最高 4 位标别设备类型个体（personality），如果该类型可以支持多实例（scull0-3 和scullpipe0-3），低 4 位可以供你标别这些设备。因此，scull0 的高 4 位与scullpipe0 不同，而scull0 的低 4 位与scull1 不同*。源码中定义了两个宏（TYPE 和NUM）从设备号中分解出这些位，我们马上就看到这些宏。

对于每一设备类型，scull 定义了一个相关的 file_operations 结构，并在 open 时替换 filp->f_op。下面的代码就是位切分和多 fops 是如何实现的：

（代码）

内核根据主设备号调用 open；scull 用上面给出的宏处理次设备号。接着用 TYPE 索引 scull_fop_array 数组，从中分解出被打开设备的方法集。

我在 scull 中所做的就是根据次设备号的类型给 filp->f_op 赋上正确的值。然后调用新的 fops 中定义的 open 方法。通常，驱动程序不必调用自己的 fops，它只有内核分配正确的驱动程序方法时调用。但当你的 open 方法不得不处理不同设备类型时，在根据被打开设备次设备号修改 fops 指针后就需要调用 fops->open 了。

scull_open 的实际代码如下。它使用了前面那段代码中定义的 TYPE 和 NUM 两个宏来切分次设备号：

（代码）

这里给一点解释。用来保存内存区的数据结构是 Scull_Dev，这里简要介绍一下。Scull_Dev 和 scull_trim（“Scull 的内存使用”一节中讨论）的内部结构这里并没有使用。全局变量 scull_nr_devs 和 scull_devices[]（全部小写）分别是可用设备数和指向 Scull_Dev 的指针数组。这段代码看起来工作很少，这是因为当调用 open 时它没做任何针对某个设备的处理。由于 scull0-3 设备被设计为全局的和永久性的，这段代码无需做什么。特别是，由于我们无法维护 scull 的打开计数，也就是模块的使用计数，因此没有类似于“首次打开时初始化设备”这类动作。

唯一实际操作在设备上的操作是，当设备写打开时将设备截断为长度 0。截断是 scull 设计的一部分：用一个较短的文件覆盖设备，以便缩小设备数据区，这与普通文件写打开截断为 0 很相似。

但“打开是截断”有一个严重的缺点：若干因为某些原因设备内存正在使用，释放这些内存会导致系统失效。尽管可能性不大，这种情况总会发生：如果 read 和 write 方法在数据传输时睡眠了，另一个进程可能写打开这个设备，这时麻烦就来了。处理竞争条件是一个相当高级的主题，我将在第 9 章的“竞争条件”中讲解。scull 处理这个问题的简单方法就是在内存还是使用时不释放内存，“Scull 的内存使用”一节中将说明。

以后我们看到其他 scull 个体（personality）时将会看到一个真正的初始化工作如何完成。

---

* “行律”是用来处理终端I/O策略的软件模块。

* 位切分一种典型的使用次设备号的方式。例如，IDE驱动程序就使用高 2 位表示磁盘号，低 6 位表示分区号。

release 方法

release 方法的作用正好与 open 相反。这个设备方法有时也称为 close。它应该：

- 使用计数减 1。
- 释放 open 分配在 filp->private_data 中的内存。
- 在最后一次关闭操作时关闭设备。

scull的基本模型无需进行关闭设备动作，所以所需代码是很少的*：

（代码）

使用计数减 1 是非常重要的，因为如果使用计数不归 0，内核是不会卸载模块的。

如果某个时刻一个从没被打开的文件被关闭了计数将如何保证一致呢？我们都知道，dup 和 fork 都会在不调用 open 的情况下，将一个打开文件复制为 2 个，但每一个都会在程序终止时关闭。例如，大多数程序从来不打开它们的 stdin 文件（或设备），但它们都会在终止关闭它。

答案很简单。如果 open 没有调用，release 也不会调。内核维护一个 file 结构被使用了多少次的使用计数。无论是 fork 还是 dup 都不创建新的数据结构；它们仅是增加已有结构的计数。

新的 struct file 仅由 open 创建。只有在该结构的计数归 0 时 close 系统调用才会执行 close 方法，这只有在删除这个结构时才会进行。close 方法与 close 系统调用间的关系保证了模块使用计数永远是一致的。

Scull 的内存使用

在介绍读写操作以前，我们最好先看看 scull 如何完成内存分配以及为什么要完成内存分配。为了全面理解代码我们需要知道“如何分配”，而“为什么”则反映了驱动程序编写者需要做出的选择，尽管 scull 绝不是一个典型设备，但同样需要。

本节只讲解 scull 中的内存分配策略，而不会讲解你写实际驱动程序时需要的硬件管理技巧。这些技巧将在第 8 章“硬件管理”和第 9 章中介绍。因此，如果你对针对内存操作的 scull 驱动程序的内部工作原理不感兴趣的话，你可以跳过这一节。

scull 使用的内存，这里也称为“设备”，是变长的。你写的越多，它就增长得越多；消减的过程只在用短文件覆盖设备时发生。

所选的实现 scull 的方法不是很聪明。实现较聪明的源码会更难读，而且本节的目的只是讲解 read 和 write，而不是内存管理。这也就是为什么虽然整个页面分配会更有效，但代码只使用了 kmalloc 和 kfree，而没有涉及整个页面的分配的操作。

而另一面，从理论和实际角度考虑，我又不想限制“设备”区的尺寸。理论上将，给所管理的数据项强加任何限制总是很糟糕的想法。从实际出发，为了测试系统在内存短缺时的性能，scull 可以帮助将系统的剩余内存用光。进行这样的测试有助于你理解系统的内部行为。你可以使用命令 cp /dev/zero /dev/scull 用光所有的物理内存，而且你也可以用工具 dd 选择复制到 scull 设备中多少数据。

在 scull 中，每个设备都是一组指针的链表，而每一个指针又指向一个 Scull_Dev 结构。每一个这样的结构通过一个中间级指针数组最多可引用 4,000,000 个字节。发行的源码中使用了一个有 1000 个指针的数组，每个指针指向 4000 个字节。我把每一个内存区称为一个“量

* 由于scull_open用不同的fops替换了filp->f_ops，不同种类的设备会使用不同的函数完成关闭操作，这一点

子”，数组（或它的长度）称为“量子集”。scull 设备和它的内存区如图 3-1 所示。
所选择的数字是这样的，向 scull 写一个字节就会消耗内存 8000 了字节：每个量子 4 个，量子集 4 个（在大多数平台上，一个指针是 4 个字节；当在 Alpha 平台编译时量子集本身就会耗费 8000 个字节，在 Alpha 平台上指针是 8 个字节）。但另一方面，如果你向 scull 写大量的数据，由于每 4MB 数据只对应一个表项，而且设备的最大尺寸只限于若干 MB，不可能超出计算机内存的大小，遍历这张链表的代价不是很大。
为量子和量子集选择合适的数值是一个策略问题，而非机制问题，而且最优数值依赖于如何使用设备。源码中为处理这些问题允许用户修改这些值：

- 在编译时，可以修改 scull.h 中的 SCULL_QUANTUM 和 SCULL_QSET。
- 在加载时，可以利用 insmod 修改 scull_quantum 和 scull_qset 整数值。
- 在运行时，用 ioctl 方法改变默认值和当前值。ioctl 将在第 5 章的“ioctl”一节中介绍。

使用宏和整数值进行编译时和加载时配置让人想起前面提到的如何选择主设备号。无论何时驱动程序需要一个随意的数值或这个数值与策略相关，我都使用这种技术。
留下来的唯一问题就是如何选择默认数值。尽管有时驱动程序编写者也需要事先调整配置参数，但他们在编写自己的模块时不会碰到同样的问题。在这个特殊的例子里，问题的关键在于寻找因未填满的量子和量子集导致的内存浪费和量子和量子集太小带来的分配、释放和指针连接等操作的代价之间的平衡。
此外，还必须考虑 kmalloc 的内部设计。现在我们还无法讲述太多的细节，只能简单规定“比 2 次幂稍小一点是最佳尺寸”比较好。kmalloc 的内部结构将在第 7 章“Getting Hold of Memory”的“The Real Story of kmalloc”一节中探讨。
默认数值的选择基于这样的假设，大部分程序员不会受限与 4MB 的物理内存，那样大的数据量有可能会写到 scull 中。一台内存很多的计算机的属主可能因测试向设备写数十 MB 的数据。因此，所选的默认值是为了优化中等规模的系统和大数据量的使用。
保存设备信息的数据机构如下：
（代码）
下面的代码给出了实际工作时是如何利用 Scull_Dev 保存数据的。其中给出的函数负责释放整个数据区，并且在文件写打开时由 scull_open 调用。如果当前设备内存正在使用，该函数就不释放这些内存（象“open 方法”中所说那样）；否则，它简单地遍历链表，释放所有找到的量子和量子集。
（代码）

读和写

读写 scull 设备也就意味着要完成内核空间和用户进程空间的数据传输。由于指针只能在当前地址空间操作，而驱动程序运行在内核空间，数据缓冲区则在用户空间，这一操作不能通过通常利用指针或 memcpy 完成。
由于驱动程序不过怎样都要在内核空间和用户缓冲区间复制数据，如果目标设备不是 RAM 而是扩展卡，也有同样的问题。事实上，设备驱动程序的主要作用就是管理设备（内核空间）和应用（用户空间）间的数据传输。
在 Linux 里，跨空间复制是通过定义在<asm/segment.h>里的特殊函数实现的。完成这种操作函数针对不同数据尺寸（char，short，int，long）进行了优化；它们中的大部分将在第 5 章的“使用 ioctl 参数”一节中介绍。

我们将在后面看到。

scull 中 read 和 write 的驱动程序代码需要完成到用户空间和来自用户空间的整个数据段的复制。下面这些提供这些功能，它们可以传输任意字节：

（代码）

这两个函数的名字可以追溯到第 1 版 Linux，那时唯一支持的体系结构是 i386，而且 C 代码中还可以窥见许多汇编码。在 Intel 平台上，Linux 通过 FS 段寄存器访问用户空间，到 Linux 2.0 时仍沿用了以前的名字。在 Linux 2.1 中它们改变了，但是 2.0 是本书的主要目标。详情可见第 17 章的“访问用户空间”。

尽管上面介绍的函数看起来很象正常的 memcpy 函数，但当在内核代码中访问用户空间时必须额外注意一些问题；正在被访问的用户页面现在可能不在内存中，而且页面失效处理函数有可能在传送页面的时候让进程进入睡眠状态。例如，必须从交换区读取页面时会发生这种情况。对驱动程序编写者来说，静效果就是对于任何访问用户空间的函数都必须是可重入的，而且能够与其他驱动程序函数并发执行。这就是为什么 scull 实现中不允许在 dev->usage 不为 0 时释放设备：read 和 write 方法在它们使用 memcpy 函数前先完成 usage 计数加 1。

现在谈谈实际的设备方法，读方法的任务是将数据从设备复制到用户空间（使用 memcpy_tofs），而写方法必须将数据从用户空间复制到设备（使用 memcpy_tofs）。每一个 read 或 write 系统调用请求传输一定量的字节，但驱动程序可以随意传送其中一部分数据——读与写的具体规则稍有不同。

如果有错误发生，read 和 write 都返回一个负值。返回给调用程序一个大于等于 0 的数值，告诉它成功传输了多少字节。如果某个数据成功地传输了，随后发生了错误，返回值必须是成功传输的字节计数，而错误直到下次函数被调用时才返回给程序。

read 的不同参数的相应功能如图 3-2 所示。

（图 3-2）

内核函数返回一个负值通知错误，该数的数值表示已经发生的错误种类（如第 2 章“编写和运行模块”的“init_module 中的错误处理”所述），运行在用户空间的程序访问变量 errno 获知发生什么错误。这两方面的不同行为，一方面坏是靠库规范强加的，另一方面是内核不处理 errno 的优点导致的。

现在再来谈谈可移植性，你一定会注意到 read 和 write 方法中参数 count 的类型总是 int，但在内核 2.1.0 中却变为 unsigned long。而且，由于 long 既可以表示 count 也可以表示负的错误编码，方法的返回值也从 int 变为了 long。

类型的修改是有益的：对于 count 来说，unsigned long 与 int 相比是一个更好的选择，它有更宽广的值域。这种选择非常好，Alpha 小组在 2.1 尚为发行时就修改了类型（主要是因为 GNU C 库在它们的系统调用定义中使用的是 unsigned long）。

尽管有好处，这一修改却带来了驱动程序代码的平台相关性。为了绕过这个问题，所有的 O’Reilly FTP 站点上的样例模块都使用了如下定义（来自 sysdep.h）：

（代码）

在宏计算后，read 和 write 的 count 计数就永远声明为 count_t，而返回值则是 read_write_t 的。我为没有使用 typedef，而是使用了宏定义，这是因为加入 typedef 后会引起更多的编译告警（见第 10 章“明智使用数据类型”中的“接口相关类型”一节）。另一方面，函数原型中出现大写的名字很难看，所以我使用标准 typedef 规范命名新“类型”。

第 17 章将更详细地介绍版本 2.1 的可移植性。

read 方法

调用程序对 read 返回值的解释如下：

- 如果返回值等于最为 count 参数传递给 read 系统调用的值，所请求的字节数传输就成功完成了。这是最好的情况。
- 如果返回值是正的，但是比 count 小，只有部分数据成功传送。这种情况因设备的不同可能有许多原因。大部分情况下，程序会重新读数据。例如，如果你用 fread 函数读数据，这个库库函数会不断调用系统调用直至所请求的数据传输完成。
- 如果返回值为 0，它表示已经到达了文件尾。
- 负值意味着发生了错误。值就是错误编码，错误编码在<linux/errno.h>中定义。

上面的表格中遗漏了一种情况，就是“没有数据，但以后会有”。在这种情况下，read 系统调用应该阻塞。我们将在第 5 章的“阻塞型 I/O”一节中处理阻塞出入。

scull 代码利用了这些规则。特别是，它利用了部分读规则。每一次调用 scull_read 只处理一个数据量子，而不必实现循环收集所有数据；这样一来代码就更短更易读了。如果读程序确实需要更多的数据，它可以重新调用这个调用。如果用标准库读取设备，应用程序都不会注意到数据传送的量子化过程。

如果当前的读位置超出了设备尺寸，scull 的 read 方法就返回 0 告知程序这里已经没有数据了（换句话说就是，我们已经到文件尾了）。如果进程 A 正在读设备，而此时进程 B 写打开这个设备，于是将设备截断为长度 0，这种情况就发生了。进程 A 突然发现自己超过了文件尾，并且在下次调用 read 时返回 0。

这里是 read 的代码：

（代码）

write 方法

与 read 相似，根据如下返回值规则，write 也可以传输少于请求的数据量：

- 如果返回值等于 count，则完成了请求数目的字节传送。
- 如果返回值是正的，但小于 count，只传输了部分数据。再说明一次，程序很可能会再次读取余下的部分。
- 如果值为 0，什么也没写。这个结果不是错误，而且也没有什么缘由需要返回一个错误编码。再说明一次，标准库会重复调用 write。以后的章节会介绍阻塞型 write，我们会对这种情形最更详尽的考察。
- 负值意味发生了错误；语义与 read 相同。

很不幸，有些错误程序在发生部分传输时会报错并异常退出。最显著的例子就是一个不算旧版本的 GNU 文件工具，它就有这样的错误。如果你的按照是 1995 年或更早的（例如，Slackware 2.3），你的 cp 在处理 scull 会失败。如果在 cp 写一块比一个量子大的数据时，你见到这样一条消息/dev/scull: no such file or directory，你就是用的这个版本的 cp。GNU dd 最初就被设计为拒绝读写部分块，而 cat 拒绝写部分块。因此，不应该用 cat 访问 scull，而 dd 则应该传递与 scull 的量子大小相同的块。注意，注意，这一缺陷在 scull 的实现中可以弥补，但我不想把代码搞的太复杂，能说明问题就行了。

与 scull 的 read 代码相同，write 代码每次只处理一个量子：

（代码）

试试新设备

一旦你了解了刚刚讲解的 4 个方法，驱动程序就可以编译和测试了；它保留你写的数据，直

至你用新覆盖它们。这个设备有点象长度只受物理 RAM 容量限制的数据缓冲区。你可以试试 cp，dd 以及输入/输出重定向等命令和机制来测试这个驱动程序。
根据你向 scull 里写了多少数据，用 free 命令可以看到空闲内存的缩减和扩增。
为了进一步证实读写确实是每次一个量子的，你可以在驱动程序的适当位置增加 printk，通过它来看看在程序读或写大数据块到底系统是如何工作的。另外，还可以用工具 strace 来监视程序调用的系统调用，以及它们的返回值。跟踪 cp 或 ls -l > /dev/scull0 会显示出量子化的读写。下一章将详细介绍监视（或跟踪）技术。

快速索引

本章介绍了下列符合头文件。struct file_operations 和 struct file 的字段列表这里没有给出。
#include <linux/fs.h>
“文件系统”头文件，它是编写设备驱动程序必须的头文件。所有重要的函数都在这里声明。
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
注册字符设备驱动程序。如果主设备不为 0，它就不加修改的使用；如果主设备号为 0，系统动态给这个设备赋一个新设备号。
int unregister_chrdev(unsigned int major, const char *name);
在卸载时注销驱动程序。major 和 name 字串都必须存放与注册驱动程序时相同的值。
kdev_t inode->i_rdev;
通过传递给每个设备方法的 inode 参数可以访问的当前设备的设备“号”。
int MAJOR(kdev_t dev);
int MINOR(kdev_t dev);
这两个宏从设备项中分解出主次设备号。
kdev_t MKDEV(int major, int minor);
这个宏由主次设备号构造 kdev_t。
#include <asm/segment.h>
2.0 及以上内核定义跨空间复制函数的头文件。这些函数是那些从用户段到内核段复制数据或反之的函数。版本 2.1 修改了这些函数以及头文件的名字（预知详情可见第 17 章的“访问用户空间”）。
void memcpy_fromfs(void *to, const void *from, unsigned long count);
void memcpy_tofs(void *to, const void *from, unsigned long count);
这些函数用来从用户空间到内核空间或反之复制字节数组。“FS”是用来在内核代码中访问用户空间的 i386 段寄存器。这些函数在版本 2.1 修改了。