第十二章 加载块设备驱动程序

正如在第一章“Linux 核心简介”中“设备与模块的分类”中所概述的一样，Unix 的设备驱动程序并不仅限于字符设备。本章就来介绍一下第二大类的设备驱动程序——块设备驱动程序。所谓面向块的设备是指数据传输是以块为单位的（例如软盘和硬盘），这里硬件的块一般被称作“扇区（Sector）”。而名词“块”常用来指软件上的概念：驱动程序常常使用 1KB 大小的块，即使扇区大小为 512 字节。

在这一章，我们将来构造一个全特征的块设备驱动程序 *sbull*（Simple Block Utility for Loading Localities）。这个驱动程序与 *scull* 类似，也是使用计算机的内存作为硬件设备。换句话说，它是一个 RAM-disk 的驱动程序。*sbull* 可以在任何 Linux 计算机上执行（不过我只在有限的几个平台上作过测试）。

注册驱动程序

和字符设备驱动程序类似，核心里的块设备驱动程序也是由一个主设备号来标识。用来对其进行注册和取消注册的函数是：

```
int register_blkdev(unsigned int major, const char*name, struct file_operations *fops)
int unregister_blkdev(unsigned int major, const char*name);
```

参数的含义与字符设备驱动程序一样，对主设备号的动态赋值也类似。因此，一个 *sbull* 设备与 *scull* 一样将自己注册：

```
result=register_blkdev(sbull_major,“sbull”，$sbull_fops);
if(result<0){
        printk(KERN_WARNING“sbull:can’t get major %d\n”,sbull_major);
        return result;
        }
if (sbull_major==0)    sbull_major=result;         /*dynamic*/
major=sbull_major;                                 /*Use “major”later on to save typing*/
```

register_blkdev 的 fops 参数与我们在字符设备驱动程序中使用的类似，为 *read*，*write* 以及 *fsync* 的操作并不要求针对某个驱动程序。通用函数 b*lock_read*, *block_write* 及 *block_fsync* 被用来代替任何针对某个驱动程序的函数。另外，*check_media_change* 和 *revalidate* 对块设备驱动程序也有意义，二者都在 sbull_fops 中定义。

在 *sbull* 中使用的 fops 结构如下：

（代码 236）

通用的读写操作被用来获得较高的性能。通过数据缓冲获得加速，这在字符设备驱动程序重中是没有的。块设备驱动程序可以被缓冲是因为它们的数据服从于计算机的文件层次结构，任何应用程序都无法直接访问，而字符设备驱动程序则不是这样。

不过，当缓冲的高速缓存不能满足一个读请求或当一个待处理的写操作要刷新到物理磁盘上时，驱动程序必须被调用来进行真正的数据传送。fops 结构除了 *read* 和 *write* 外，并不带有入口点，因此，必须要一个额外的结构 blk_dev_struct 来发出对实际数据传送的请求。

这个结构在<linux/blkdev.h>定义，它有几个域，但只有第一个域需被驱动程序设置。下面是这个结构在核心 2.0 中的定义。

（代码 237）

当核心需要为 *sbull* 设备产生一个 I/O 操作时，它便调用函数 blk_dev[*sbull*_major].request_fn。

因此这个模块的初始化函数须设置这个域使其指向它自己的请求函数。这个结构中的其它域只供核心函数或宏进行内部使用；你不必在你的代码段中显式地使用它们。
一个块设备驱动程序模块与核心的关系见图 12-1。
除了 blk_dev 还有几个数组带有块设备驱动程序的信息。这些数组一般由主设备号（有时也用次设备号）进行索引。它们在 *drivers/block/ll_rw_block.c* 中被声明和描述。
int blk_size[][];
这个数组由主设备号和次设备号索引。它以 KB 为单位描述了每个设备的大小。如果 blk_size[major]是 NULL，则不对这个设备的大小进行检查（也就是说，核心可能要求数据传送通过 end_of_device）。
int blksize_size[][];
被每个设备所使用的块的大小，以字节为单位。与上一个数组类似，这个二维数组也是由主设备号和次设备号索引。如果 blksize_size[major]是一个空指针，那么便假设其块大小为 BLOCK_SIZE（目前是 1KB）。块大小必须是 2 的幂，因为核心使用移位操作将偏移量转换为块号。
int hardsect_size[][];
与其它的一样，这个数据结构也是由主设备号和次设备号索引。硬件扇区的缺省大小为 512 字节。直到包括 2.0.X 版本为止，可变扇区大小仍未真正支持，因为一些核心代码仍旧假设扇区大小为半 KB。不过很可能在 2.2 版本中会真正实现可变扇区大小。
int read_ahead[];
这个数组由主设备号索引，它定义了一个文件被顺序读取时，核心可以提前读取多少扇区。在进程请求数据之前将其读出可以改善系统的性能及总的吞吐率。慢速设备最好指定一个较大的提前读的值，而一个快速设备则可以在较小的提前读的值下工作的很好。这个提前读的值越大缓冲高速缓存则需要越多的内存。每个主设备号有一个提前读的值，它对所有次设备号有效。这个值可以通过驱动程序的 ioctl 方法来改变；硬盘驱动程序一般设为 8 个扇区，对应着 4KB。
sbull 设备允许在加载时设置这些值，它们作用于示例驱动程序的所有次设备号。在 *sbull* 中变量名和它们的缺省值为：
size=2048(KB)
由 *sbull* 生成的每个 ramdisk 占两兆字节，正如系统的缺省值。
hardsect=512(B)
sbull 扇区大小是常用的半 KB 值。改变 hardsect 的值是不允许的。
如前所述，其它的扇区大小并不被支持。如果你一定要改它，可以将 *sbull*/*sbull*.c 中的安全检查去掉。不过请做好发生严重的内存崩溃的危险的准备。除非在你尝试时，已经加上了对可变扇区大小的支持。
rahead=2(扇区)
因为 ramdisk 是一个快速设备，所以这个缺省提前读的值比较小。
sbull 设备也允许你选择一个设备个数进行安装。devs 是设备个数，缺省设为 2，表明缺省内存使用量为 4 兆——2 个大小为 2MB 的盘。
sbull 设备的 *init_module* 的实现如下（不含主设备号的注册和错误恢复）：
（代码 239）
相应的清除函数如下所示：
（代码 240）
这里，调用 *fsync_dev* 是必须的，用以清除核心保存在不同高速缓存中的对设备的所有引用。事实上，*fsync_dev* 是运行在 *block_fsync* 之后的引擎，它是块设备的 *fsync* “方法”。

头文件 blk.h

由于块设备驱动程序的绝大部分是设备无关的,核心的开发者通过把大部分相同的代码放在一个头文件<linux/blk.h>中，来试图简化驱动程序的代码。因此，每个块设备驱动程序都必须包含这个头文件，在<linux/blk.h>中定义的最重要的函数是 *end_request*，它被声明为 static（静态）的。让它成为静态的，使得不同驱动程序可有一个正确定义的 *end_request*，而不需要每个都写自己的实现。

在 Linux1.2 中，这个头文件应该用<linux/../../drivers/block/blk.h>来包含。原因在于当时还不支持自定义的块设备驱动程序，而这个头文件最初位于 drivers/block 源码目录下。

实际上，*blk.h* 相当不寻常，比如它定义了几个基于符号 MAJOR_NR 的符号，而 MAJOR_NR 必须由驱动程序在它包含这个头文件之前声明。这里，我们再次看到 *blk.h* 在设计时并没有真正考虑自定义驱动程序。

看看 *blk.h*，你会发现几个设备相关的符号是按照 MAJOR_NR 的值声明的，也就是说 MAJOR_NR 应该提前知道。然而，如果主设备号是动态赋值的，驱动程序无法预知其值，因此也就不能正确定义 MAJOR_NR。如果 MAJOR_NR 未定义，*blk.h* 就不能设定一些在 *end_request* 中使用的宏。因此，为了让自定义驱动程序从通用的 *end_request* 函数受益，从而避免重新实现它，驱动程序必须在包含 *blk.h* 之前定义 MAJOR_NR 和其它几个符号。

下面的列表描述了一些必须提前定义的<linux/blk.h>中的符号。列表结尾给出了 *sbull* 中使用的代码。

MAJOR_NR

> 这个符号用来访问一些数组，特别是 blk_dev 和 blksize_size。自定义驱动程序（如 *sbull*）不能给这个符号赋一个常量值，可以将其定义（#define）为一个存有主设备号的变量。对 *sbull* 而言，它是 sbull_major。

DEVICE_NAME

> 被生成的设备名。这个字符串用来从 *end_request* 中打印错误信息。

DEVICE_NR(kdev_t device)

> 这个符号用来从 kdev_t 设备号中抽取物理设备的序号。这个宏的值可以是 MINOR(device)或别的表达式。这要依据给设备或分区分配次设备号的常规方式而定。对同一个物理设备上的所有分区，这个宏应返回同一个设备号——也就是说，DEVICE_NR 表达的是磁盘号，而不是分区号。这个符号被用来声明 CURRENT_DEV，它在 *request_fn* 中用来确定被一个传送请求访问的硬件设备的次设备号，可分区设备将在后面“可分区设备”一节中介绍。

DEVICE_INTR

> 这个符号用来声明一个指向当前下半部处理程序的指针变量。宏 SET_INTR(intr)和 CLEAR_INTR 用来给这个变量赋值。当设备可以发出具有不同含义的中断时，使用多个处理程序是很方便的。这个主题将在后面“中断驱动的块设备驱动程序”一节中讨论。

TIMEOUT_VALUE

DEBICE_TIMEOUT

> TIMEOUT_VALUE 以记数的方式表达超时，这个超时的值与老计时器之一（特别地指计时器号 DEVICE_TIMEOUT）相关联。一个驱动程序可以在数据传送时间太长时，通过调用一个回调函数来检测错误条件。不过，由于老计时器由一个预赋值的计时器静态数组组成（见第六章“时间流”中“核心计时器”一节），一个自定义的驱动程序不能使用它们。我在 *sbull* 中对这两个符号都未定义，而是用一个新的计时器实现超时。

DEBICE_NO_RANDOM

在缺省情况下，函数 *end_request* 对系统熵值（即所有随机性的总量）有所贡献，这被 */dev/random* 所使用。如果一个设备不能对随机设备贡献显著的熵值，DEVICE_NO_RANDOM 应被定义。*/dev/random* 在第九章的“安装中断处理程序”中进行了介绍，SA_SAMPLE_RANDOM 也在那儿做了解释。

DEVICE_OFF(kdev_t device)

end_request 函数在结束时调用这个宏。例如在软盘驱动程序中，它调用一个函数，这个函数负责更新用来控制马达停转的一个计时器。如果设备没有被关掉，那么串 DEVICE_OFF 可以被定义为空。*sbull* 不使用 DEVICE_OFF。

DEVICE_ON(kdev_t device)

DEVICE_REQUEST

这些函数实际上并未在 Linux 的头文件中使用，所以驱动程序并不需要定义它们。大多数官方的 Linux 设备驱动程序声明这些符号并在内部使用它们，但我在 *sbull* 里并没有使用它。

sbull 驱动程序以如下的方式声明这些符号：

（代码 242）

头文件 *blk.h* 用上面列出的这些宏定义了一些可以由驱动程序使用的额外的宏，我将在后续章节里对之进行介绍。

处理请求

系统性能的方式排序。这些联结表中的请求被传递个驱动程序的请求函数，由它对链接表中的每个请求执行如下的任务：

- 检查当前请求的有效性。这个工作由在 *blk.h* 中定义的宏 INIT_REQUEST 完成。
- 进行实际的数据传送。用变量 CURRENT（实际上是个宏）可以获得发出请求的一些细节。CURRENT 是一个指向结构 request 的指针，我将在下节介绍这个结构的域。
- 清除当前的请求。这个操作由静态函数 *end_request* 完成，函数的代码在 *blk.h* 中。驱动程序向这个函数传递一个参数，即成功时为 1，失败时为 0。当 *end_request* 以参数 0 调用时，一个“I/O error”消息会被发给系统日志（通过 *printk*）。
- 循环回至开始，消化下一个请求。可以按照程序员的喜好使用一个 goto 或是一个 for(;;)，或者 while(1)。

实践中，请求函数的代码如下构造：

（代码 243）

尽管这段代码除了打印消息外什么都没有做，运行这个函数可以对数据传送的基本设计有一个很好的了解。到此为止，代码中唯一不清楚的地方是 CURRENT 的确切含义及它的域，这个我将在下一节介绍。

我的第一个 *sbull* 实现只包含了所示的空代码。我意在一个“不存在”的设备上构造一个文件系统，并使用它一会儿，只要数据仍在缓冲高速缓存中。在运行一个象这样罗嗦的请求函数时，看看系统日志能帮助你理解缓冲高速缓存是如何工作的。

在编译时，定义符号 *SBULL*_EMPTY_REQUEST，那么这个空且罗嗦的函数可以在 *sbull* 中运行。如果你想理解核心是如何处理不同块大小的，你可以在 *insmod* 命令行上实验 blksize=。这个空的请求函数通过打印每个请求的细节揭示了内部核心的工作情况。你或许也可以试试 hardsect=，但目前它被关闭了，因为比较危险。（见本章开始时的“注册驱动程序”）。

请求函数的代码并不显式地调用 return(),因为当列表中的待处理请求耗尽时，INIT-_REQUEST 会替你完成这个工作。

执行实际的数据传送

为了给 *sbull* 构造一个可以工作的数据传送，让我们先来看看核心是如何在结构 request 中描述一个请求的。这个结构在<linux/blkdev.h>中定义。通过访问 CURRENT 的域，驱动程序可以得到所有为在缓冲高速缓存的物理块设备之间传送数据所需要的信息。

CURRENT 是用来访问当前请求(即被首先服务的那个请求)。正如你可能猜到的，CURRENT 是 blk_dev[MAJOR_NR].current_request 的缩短形式。

下面这些当前请求的域包含了请求函数的有用信息：

kdev_t rq_dev;

请求所访问的设备。有本驱动程序所管理的所以设备均被使用同一个请求函数。一个请求函数处理所有的次设备号；rq_dev 可以被用来取得被操作的次设备。尽管 Linux1.2 称这个域为 dev，你仍然可以通过宏 CURRENT_DEV 来访问这个域。CURRENT_DEV 在我们所讨论的所有版本的核心中是可移植的。

int cmd;

这个域是 READ 或 WRITE。

unsigned long sector;

请求指向的第一个扇区。

unsigned long current_nr_sectors;

unsigned long nr_sectors;

当前请求的扇区数（大小）。驱动程序应该引用 current_nr_sectors,而应该忽略 nr_sectors（列在这里只是为了完整）。请看下一节“集簇请求”以获得更多的细节。

char *buffer

缓冲高速缓存中的域。如果 cmd==READ，就是写数据的位置；如果 cmd==WRITE，就是读数据的位置。

struct buffer_head *bh

这个结构描述了这个请求列表中的第一个缓冲区。我们将在“集簇请求”中用到这个域。在这个结构中还有其它的一些域，但它们基本上是核心内部使用的，驱动程序并不期望使用它们。

sbull 中可工作的请求函数的实现如下所示。在下面的代码中 sbull_devices 与 scull_device 类似。我们在第三章字符设备驱动程序的“打开方法”中介绍过 scull_devices。

（代码 245）

由于 *sbull* 只是个 RAM 盘，所以它的“数据传送”简化为一个 *memcpy* 调用。这个函数唯一“奇怪”的特征是条件语句中限制只能报告最多 5 个错误。这样做的目的是为了防止系统日志被太多的信息搞乱，因为 end_request(0)在请求失败时已打印了“I/O error”的消息。静态计数器是限制消息报告的标准做法，在核心中被多次用到。

集簇请求

上面请求函数中每次循环迭代都传送几个扇区——按照数据的使用，一般情况下，相当于一个块的“数据”量。例如，交换一次执行 PAGE_SIZE 大小的数据，而在 ext2 文件系统中就是传送 1KB 的块。

尽管在 I/O 中最方便的数据大小是一个块，但如果把相邻块的读或写集簇起来，你会获得很高的性能改善。在这个意义上，“相邻”指的是在硬盘上块的位置，而“连续”则指连续的内存区域。

将相邻块集簇有两个好处。首先，集簇加速了传送（例如，软盘驱动程序将相邻的块组合在一起，一次传送一个磁道的数据)。另外，它还能通过避免分配冗余的 request 结构来节省核心中的内存。

如果你愿意，也可以完全忽略集簇。上面给出的框架请求函数在没有集簇的情况下可以完全

正确地工作。不过，如果你想利用集簇，你需要更加仔细地研究 struct_request 的内部。
不幸的是，我所知道的所有的核心（至少到 2.1.51）都不能为自定义驱动程序进行集簇，而只对象 SCSI 和 IDE 这类内部驱动程序使用。如果你对核心的内部不感兴趣，你可以跳过本节的其余部分。不过，集簇将来还可能在模块中实现，它是通过减少相邻扇区的请求延迟来提高数据传送性能的一个有趣的途径。
在我描述驱动程序如何利用集簇请求之前，让我们先来看看当一个请求被排队时发生了什么。
当核心请求数据块传送时，它扫描目标设备的活动请求链表。当一个新块在盘上与一个已经被请求的块相邻时，它就被集簇到第一个块上。当前已存在的请求便被扩大了而不是增加一个新请求。
不幸的是，磁盘上相邻的两个数据缓冲区在内存中并不一定相邻。这个发现，外加上需要有效地管理缓冲高速缓存，导致创建一个 buffer_head 结构。一个 buffer_head 和一个数据缓冲相关联。
因此，一个“集簇”的请求，就是一个指向 buffer_head 的结构链表的 request_struct 结构。*end_request* 函数负责这个问题，这就是为什么前面给出的请求函数可以独立于集簇而工作。换句话说，*end_reques*t 要么清除当前请求并准备为下一个服务，要么准备处理同一个请求中的下一个缓冲区。因此，集簇对不关心它的设备驱动程序是透明的，上面的 *sbull* 函数就是一个例子。
一个驱动程序可能希望通过在它的 *request_fn* 函数中每次循环时处理整个缓冲区头链表的办法来从集簇中获益。为了做到这一点，驱动程序应该指向 CURRENT->current_nr_sectors（这个域我在上面的 *sbull_request* 中已经用过）和 CURRENT->nr_sectors，它包含了集簇在“当前”buffer_heads 列表中的相邻扇区的数目。
当前缓冲区头是 CURRENT->bh，而数据块是 CURRENT->bh->b_data。后一个指针为了象 *sbull* 一类忽略集簇的驱动程序缓冲在 CURRENT->buffer 中。
请求集簇在 *drivers/block/ll_rw_block.c* 的函数 *make_request* 中实现。不过，如上所说，集簇只对几个驱动程序有效（软驱，IDE，和 SCSI），以其主设备号为准。我曾通过以 major=34 装载 *sbull* 看到过集簇是如何工作的，因为 34 是 IDE3_MAJOR，而我的系统中没有第三个 IDE 控制器。
下面列表总结了当扫描一个集簇请求时应做的事项。bh 是被处理的缓冲区头——列表的第一项。对列表中的每个缓冲区头，驱动程序要完成下面一系列操作：

- 传送位于地址 bh->b_data，大小为 bh->b_size 字节数据块。数据传送的方向通常由 CURRENT->cmd 指出。
- 从列表中找出下一个缓冲区头：bh->b_request。接着通过将 b_request 置为 0，把刚传送过的缓冲区从列表中摘下。b_reqnext 指向你刚找出的新缓冲区。
- 通过调用 mark_buffer_uptodate(bh,1)，unlock_buffer(bh)，告诉核心你已完成对上个缓冲区的操作。这些调用保证缓冲高速缓存保持正确，不致有错误指向的指针。mark_buffer_uptodate 中参数“1”表示传送成功，若传送失败，则换为 0。
- 循环回到开始，传送下一个相邻块。

当你做完了集簇请求，CURRENT->bh 必须被更新以指向“已经被处理但未被解锁”的第一个缓冲区。如果列表中所有的缓冲区都已被处理和解锁，CURRENT->bh 可被置为 NULL。此时，驱动程序可以调用 *end_request*。如果 CURRENT->bh 是有效的，那么这个函数在转到下一个缓冲之前对其进行解锁——这是非集簇操作所发生的情况，此时由 *end_request* 照管所有的事情。如果指针为空，这个函数直接转到下一个请求。
全功能的集簇实现出现在 *driver/block/floppy.c*，而要求的所有操作出现在 *blk.h* 的 *end_request*

中。*floppy.c* 和 *blk.h* 都不容易理解，不过建议先从后者开始。

安装（Mounting）是如何工作的

块设备与字符设备及一般文件的不同在于它们可以被安装到计算机的文件系统上。这与一般的访问方式不同。一般的访问方式通过结构 file 进行，这个结构与特定的进程相关联，并且只在 *open* 到 *close* 之间存在。当一个文件系统被安装后，没有进程拥有一个 filp。
当核心把一个设备安装到文件系统上，它调用一般的 *open* 方法来访问驱动程序。然而，这种情况下 *open* 的参数 filp 是个虚的变量，几乎只是为了占个地方，它唯一有意义的域是 f_mode。其它域含任意值并不使用。f_mode 的值是告诉驱动程序设备是以只读（f_mode==FMODE_READ）还是读写（f_mode==（FMODE_READ|FMOD_WRITE））方式被安装。使用一个虚变量而不是 file 结构的原因是因为实际的结构 file 在进程结束时将被释放，而被安装的文件系统在 *mount* 命令完成后仍然存在。

在安装时，驱动程序唯一调用的是 *open* 方法。当磁盘被安装后，核心调用设备中的 *read* 和 *write* 方法（被映射到 request_fn）来管理文件系统中的文件。驱动程序并不知道 request_fn 服务的是一个进程（象 *fsck*）还是核心中的文件系统层。
至于 umount，它只是刷新缓冲高速缓存并调用驱动程序的 *release*（*close*）方法。由于没有有意义的 filp 可以传递给 fop->realse，核心使用 NULL。
因此，当你实现 *release* 时，你应将驱动程序设为能处理为 NULL 的 filp 指针。不然，如果你用了 filp，你可能运行 *mkfs* 和 *fsck*，它们都使用 filp 来访问设备，你也可能 mount 这个设备，但 umount 将无法运行，原因就是 NULL 指针。
由于一个块设备驱动程序的 release 实现不能用 filp->private_data 来访问设备信息，它采用 inode->i_rdev 来区分设备。这里是 *release* 的 *sbull* 实现：
（代码 249）
其它的驱动程序函数并不关心 filp 问题，因为它们与安装的文件系统无关。例如，一个显示地 *open* 这个设备的进程只发出 *ioctl*。

ioctl 方法

如字符设备一样，块设备也可以通过 *ioctl* 系统调用进行操作。两者之间相对不一样的地方在于块设备驱动程序有大量驱动程序都要支持的 *ioctl* 命令。
块设备驱动程序经常要处理的命令如下所示，它们在<linux/fs.h>中被声明。

BLKGETSIZE

获取当前设备的大小，以扇区数表示。由系统调用传递的 数值 arg 是一个指向 long 数值的指针，用来将大小拷贝到一个用户空间的变量中。这个 *ioctl* 命令可以被 *mkfs* 用来获知产生的文件系统的大小。

BLKFLSBUF

字面上的意思是“刷新缓冲区”。这个命令的实现对每个设备都是一样的，我们将在后面整个 *ioctl* 方法的示例代码中给出来。

BLKRAGET

用来为设备取得当前提前读的值。当前数值应该用在参数 arg 中传递给 *ioctl* 的指针写进一个 long 类型的用户空间变量。

BLKRASET

设置提前读的值。用户进程在 arg 中传递这个新值。

BLKRRPART

重读分区表。这个命令只对可分区设备有意义，将在后面“可分区设备”中介绍。

BLKROSET
BLKROGET
这些命令用来改变和检查设备的只读标志。因为代码是设备无关的，它们由宏RO_IOCTLS（kdev_tdev,unsigned long where）来实现。这个宏在 *blk.h* 中定义。
HDIO_GETGEO
在<linux/hdreg.h>中定义，用来获得磁盘的几何参数。这个参数应被写入用户空间的结构 hd_geometry 中，它也在 *hdreg.h* 中定义。*sbull* 显示了这个命令的一般实现。
HDIO_GETGEO 是<linux/hdreg.h>中定义的一系列 HDIO 命令中最常用的一个。感兴趣的读者可以查看 *ide.c* 和 *hd.c* 以获得这些命令的更多信息。
这里列出的这些命令的一个主要缺点是它们是以“老”方法定义的（是第五章“增强的字符设备驱动程序操作”中“选择 *ioctl* 命令”一节），因此无法使用位域的宏来减化代码——每个命令要实现它自己的 *verify_area*。不过，如果一个驱动程序需要定义它自己的命令来利用设备的一些特殊特点，你可以自由地使用“新”方法来定义命令。
sbull 设备只支持上面的通用命令，因为实现设备特定的命令与实现字符设备驱动程序的命令没有什么不同。*sbull* 的 *ioctl* 实现如下所示，它将有助于你理解上面列出的命令。
（代码 250）
（代码 251）
函数开始的 PDEBUG 语句被留出，这样当你编译这个模块时，你可以打开调试（debugging）来看看设备上调用了哪个 *ioctl* 命令。
例如，对于显示的 *ioctl* 命令，你可以在 *sbull* 上使用 *fdisk*。下面是在我自己系统上的一个示例执行过程：
（代码 252 1#）
在会话过程中下面的消息出现在我的系统日志中：
（代码 252 2#）
第一个 ioctl 是 HDIO_GETGEO，它在 *fdisk* 启动时被调用；第二个是 BLKRRPART。对后一个命令的 *sbull* 实现仅仅是调用一下 revalidate 函数，它则在打印输出中打印最后的消息（见本章后面的“revalidate”）。

可拆卸的设备

在我们讨论字符设备驱动程序时，我们忽略了 fops 结构中的最后两个文件操作，因为它们只是为可拆卸块设备而设的。现在是看看它们的时候了。*sbull* 并不真是可拆卸的，但它假装是，因此它实现了这些方法。
我所说的操作是 *check_media_change* 和 *revalidate*。前者用来发现设备自上次访问以来是否改变过，后者则在磁盘变动之后重新初始化驱动程序的状态。
至于 *sbull*，与设备相联的数据区在使用计数下降为零后半分钟要释放。待这个设备处于未安装状态（或关闭状态）足够长的时间以模拟一次磁盘的改变，下一次对设备的访问分配一个新的内存区域。
这一类的“时间到期”通过一个核心计数器来实现。

check_media_change

这个检查函数接收到 kev_t 做为一个确定设备的参数。如果介质被改变了返回值为 1，否则为 0。如果一块设备驱动程序不支持可拆卸设备，可以通过置 fops->check_media_change 为 NULL 来避免这个声明函数。
有趣的是要注意，当一个设备是可拆卸的，但却无法判断它是否改变了，这时，返回 1 是个安全选择。事实上，IDE 驱动程序在处理可拆卸磁盘时就是这么做的。

sbull 的实现是这样的，当由于计数器超时，设备已经从内存中删除时就返回 1，如果数据仍然有效则返回 0。如果设置了调试，它同时向系统日志打印一条消息，这样用户就可以检查核心什么时候调用了这个方法。

（代码 253 1#）

revalidate

这个有效化函数是在检测到一个磁盘的改变时被调用。它也被在核心的 2.1 版中实现的各种 *stat* 系统调用。返回值目前不做使用；为安全起见，返回 0 表示成功，出错时返回一个负的错误代码。

revalidate 执行的动作是设备特定的，但 revalidate 通常更新一些内部状态信息以反映新的设备。

在 *sbull* 中，*revalidate* 方法在没有一个有效区域的情况下试图分配一块新的数据区域。

（代码 253　2#）

（代码 254　1#）

特别注意

当可拆卸设备已经打开时，驱动程序也应该检查是否有磁盘的改变；在 *mount* 时核心自动调用它的 *check_disk_change* 函数，但在 *open* 时，并不这样做。

不过，有些程序直接访问磁盘数据而不安装这个设备，*fsck*，*mcopy* 和 *fdisk* 都是这类程序的例子。如果驱动程序在内存中保存可拆卸设备的状态信息，它应在设备第一次打开时调用 *check_disk_change* 函数。这个核心函数还要依赖驱动程序方法（*check_media_change* 和 *revalidate*），因此在 *open* 里不须实现任何特别的东西。

这里是 *open* 的 *sbull* 实现，它关注了发生磁盘改变的情况：

（代码 254　2#）

在驱动程序中不需对磁盘的改变做任何别的。如果一个磁盘被改变了，而它的打开计数大于零，那么数据会被破坏。防止这种情况发生的唯一方法是让利用在物理上支持的设备使用计数控制门锁。*open* 和 *close* 可以在合适的时候关闭或打开锁。

可分区设备

如果你想用 *fdisk* 生成分区，你会发现它们有一些问题。*fdisk* 程序称这些分区为*/dev/sbull01*，*/dev/sbull02* 以 此类推，但文件系统上并不存在这些名字。的确，基本的 *sbull* 设备是一个字节阵列，不存在提供访问数据区域的子区域的入口点，因此想对 *sbull* 进行分区是行不通的。

为了能对设备分区，我们必须给每个物理设备分配几个次设备号。一个数字用来访问整个设备（如*/dev/hda*），其它的用来访问不同的分区（如*/dev/hda1*）。由于 *fdisk* 产生分区名的办法是在全盘设备名后加一个数字后缀，我们将在后面的块设备驱动程序中遵循同样的命名规则。

在本节中我将要介绍的设备叫 *spull*，因此它是一个“简单的可分区工具（Simple Partitionable Utility）”。这个设备位于 *spull* 目录，完全与 *sbull* 无关，尽管它们共享很多代码。

在字符设备驱动程序 *scull* 中，不同的次设备号可以实现不同的行为，因此一个驱动程序可以显示几种不同的实现。而按照次设备号区分块设备是不可行的，这就是为什么 *sbull* 和 *spull* 被分离开。这种无能为力是块设备驱动程序的一个基本特征，因为几个数据结构和宏只是作为主设备号的函数定义的。

关于移植，需要注意的是可分区模块不能被加载到核心的 1.2 版，因为符号 *resetup_one_dev*（在本节后面介绍）没有被引出到模块。在对 SCSI 盘的支持模块化之前，没有人会考虑可分区的模块。

我要介绍的设备结点被称做 pd，表示“可分区磁盘（partitionable disk）”。四个完整的设备（又称“单元”）被称做*/dev/pd*a 直到*/dev/pdd*；每个设备最多支持 15 个分区。次设备号有下面的含义：低四位表示分区号（0 为完整的设备），高四位表示单元号。这个规则在源文件中由下面的宏表达：

（代码 255）

普通硬盘

每个可分区设备需要知道它是如何分区的。这个信息可以从分区表中得到。初始化进程的一部分包括解码分区表，并更新内部数据结构以反映分区信息。

这个解码并不容易。不过幸运的是，核心提供可被所有块设备驱动程序使用的“普通硬盘”支持，它显著地减少了处理分区驱动程序的代码。这个普通支持的另一个好处是驱动程序的作者不必理解分区是如何完成的，而不需要修改驱动程序的代码就可以在核心中支持新的分区方式。

想要支持分区的块设备驱动程序要包含<linux/genhd.h>，并声明结构 gendisk。所有这样的结构被组织在一个链表中，它的头是全局指针 gendisk_head。

在我们进行下一步之前，让我们先看看结构 gendisk 的域。你为了利用普通设备支持就需要理解它们。

int major

确定这个结构所指的设备驱动程序的主设备号。

const char*major_name

属于这个主设备号的设备的基本名。每个设备名是通过在这个名字后为每个单元加一个字母并为每个分区加一个数字得到。例如，“hd”是用来构成*/dev/hda1* 和*/dev/hda3* 的基本名。基本名最多 5 个字符长，因为 *add_partition* 在一个 8 字节的缓冲区中构造全名，它要附加上一个确定单元的字母，分区号和一个终止符‘\0’。*spull* 所用的名字是 pd（“可分区磁盘（partitionable disk）”）。

int minor_shift

从设备的次设备号中获取驱动器号要进行移位的次数。在 *spull* 中这个数是 4。这个域中的值应与宏 DEVICE_NR(device)中的定义一致（见本章前面的“头文件 *blk.h*”）。*spull* 中的宏扩展为 device>>4。

int max_p

分区的最大数目。在我们的例子中，max_p1 是 16，或更一般地，是<<minor_shift。

int max_nr

单元的最大数目。在 *spull* 中，这个数字是 4。单元最大数目在移位 minor_shift 次后的结果应匹配次设备号的可能的范围，目前是 0-255。IDE 驱动程序可以同时支持很多驱动器和每一个驱动器很多分区，因为它注册了几个主设备号，从而绕过了次设备号范围小的问题。

void(*init)（struct gendisk*）

驱动程序的初始化函数，它在初始化设备后和分区检查执行前被调用。我将在下面介绍这个函数更多的细节。

struct hd_struct *part

设备的解码后的分区表。驱动程序用这一项确定通过每个次设备号哪些范围的磁盘扇区是可以访问的。大多数驱动程序实现 max_nr<<minor_shift 个结构的静态数值，并负责数组的分配和释放。在核心解码分区表之前驱动程序应将数组初始化为零。

int *sizes

这个域指向一个整数数组。这个数组保持着与 blk_size 同样的信息。驱动程序负责分配

和释放该数据区域。注意设备的分区检查把这个指针拷贝到 blk_size，因此处理可分区设备的驱动程序不必分配这后一个数组。

int nr_real

存在的真实设备（单元）的个数。这个数字必须小于等于 max_nr。

void *real_devices

这个指针被那些需要保存一些额外私有信息的驱动程序内部使用（这与 filp->private_data 类似）。

void struct gendisk *next

在普通硬盘列表中的一根链。

分区检查的设计最适合那些直接链入核心映象的驱动程序，因此我将从介绍核心代码的基本结构开始。以后我将介绍 *spull* 模块处理它的分区的方法。

核心中的分区检测

在引导时，*init/main.c* 调用了各种各样的初始化函数。其中一个是 *start_kernel*，它通过调用 *device_setup* 来初始化所有的驱动程序。这个函数又调用 *blk_dev_init*，接着检查所有注册的普通硬盘的分区信息。任何一个块设备驱动程序，如果它找到至少一个它的设备，就将这个驱动程序的 genhd 结构注册到核心列表中，这样它的分区便可以被正确地检测出来。

因此，一个可分区的驱动程序应该声明它自己的结构 genhd。这个结构看起来如下：

（代码 258）

于是，在这个驱动程序的初始化函数中，这个结构被排队在可分区设备的主列表中。

被链入核心的驱动程序的初始化函数与 *init_module* 等价，即使它被调用的方式不同。这个函数一定包含如下两行，它们用来将结构排队：

```
my_gendisk.next=gendisk_head;
gendisk_head=my_gendisk;
```

通过将结构插入链表，这简单的两行便是驱动程序入口点为所有的分区正确地识别和配置所需要的所有内容。

额外的设置通过 *my_geninit* 完成。在上面的例子中，这个函数填充“单元数”域来反映计算机系统的实际硬件设置。在 *my_geninit* 结束后，*gendisk.c* 为所有的盘（单元）执行实际的分区检测。你可以看到系统启动时被检测的分区，因为 *gendisk.c* 在系统控制台上打印分区检查 Partition check:，后面跟随它在可得的普通硬盘上找到的所有分区。

你可以修改前面的代码，推迟 my_sizes 和 my_partitions 的分配直到 *my_geninit* 函数。这可以节省少量的核心内存，因为这些数组可以小到 nr_real<<minor_shift，而竟态数组则必须为 max_nr<<minor_shift 字节长。不过，典型的数值是每个物理单元节省几百个字节。

模块中的分区检测

一个模块化的驱动程序和链接到核心的驱动程序的区别在于它无法受益于集体中的初始化。相反，它需要处理它自己的设置。由于没有为模块的两步初始化，所以 spull 的 gendisk 结构在它的 init 函数指针中有一个 NULL 指针：

（代码 259 1#）

同时也不必在普通硬盘的全局链表里注册 gendisk 结构。

通过引出函数 *resetup_one_dev*，文件 *gendisk.c* 被准备用来处理象模块需要一类“晚的”初始化。*resetup_one_dev* 为单个物理设备扫描分区。其原型是：

boid resetup_one_dev(struct gendisk *dev,int drive);

从这个函数名字你可以看出来它是要改变一个设备的设置信息。这个函数被设计为由 *ioctl* 里 BLKRRPART 实现调用，但他也可以被用来完成一个模块的初始设置。

当一个模块被初始化后，它应该为每个它将要访问的物理设备调用 *resetup_one_dev*，从而将

分区信息贮存 my_gendisk->part 中。分区信息会被设备的 *request_fn* 函数使用。
在 *spull* 中，*init_module* 函数除了通常的指令外还包含了下面的代码。它分配分区检测所需的数组并初始化数组中完整磁盘的项目。
（代码 259 2#）
（代码 260 1#）
有趣的是注意到 *resetup_one_dev* 通过重复调用下面函数打印分区信息：
printk（“%s：”，disk_name（hd，minor，buf））；
这就是为什么 *spull* 要打印一个引导串。它意味着要为塞进系统日志的信息增加一些上下文。当一个可分区的模块被卸载时，驱动程序应该通过为每个支持的主/次对调用 *fsync_dev* 来安排所有的分区刷新。而且，如果结构 gendisk 被插在全局链表中，它应该被删除——注意 *spull* 并未自己插入它，原因上面提到过。
spull 的清除函数是：
（代码 260 2#）
（代码 261）

使用 *Initrd* 进行分区检测

如果你想从一个设备上安装你的根文件系统，而这个设备的驱动程序只有模块化的形式，你就必须使用由现代 Linux 核心提供的 *Initrd* 工具。我不想在这里介绍 *Initrd*，这一小节是针对那些了解 *Initrd* 并想知道它是如何影响块设备驱动程序的读者的。
当你用 *Initrd* 引导一个核心时，它会在安装真正的根文件系统之前建立一个暂时的运行环境。模块通常是从被用作临时根文件系统的 ramdisk 中装载。
由于 *Initrd* 进程是在所有其它引导时初始化完成之后才开始运行（但在真正的根文件系统安装之前），因此在装载一个普通模块和在 *Initrd* ramdisk 中的模块没有区别。如果一个驱动程序可以正确地装载并以模块的形式被使用，那么所有含有 *Initrd* 的 Linux 发布都可以将这个驱动程序包含在安装盘中而不需要你研究修改核心源码。

spull 的设备方法

除了初始化和清除工作，可分区设备和不可分区设备还有其它的不同。根本上说，这些区别来自一个事实，即如果一个磁盘是可分区的，那么同一个物理设备可以通过不同的次设备号进行访问。从次设备号到磁盘上物理位置的映射由 *resetup_one_dev* 存贮在数组 gendisk->part 中。下面的代码只包含了 *spull* 与 *sbull* 不同的部分，因为绝大部分代码是完全一样的。
首先，*open* 和 *close* 必须掌握每个设备的使用记数情况。由于使用记数是关于物理设备（单元）的，下面的赋值被用于 dev 变量：
spull_Dev *dev=spull_devicex+DEVICE_NR(inode->i_rdev)；
这里用到的宏 DEVICE_NR 必须包含<linus/blk.h>之前被声明。
尽管几乎所有的物理方法可以工作于物理设备，*ioctl* 应该访问每个分区的特定信息。例如，应该告诉 *mkfs* 每个分区的大小，而不是完整设备的大小。下面是 *iotcl* 的 BLKGETSIZE 命令如何因一个设备一个次设备号变为一个设备多个次设备号而受到影响的。正如你所期望的，spull=gendisk->part 被用来做为分区大小的来源。
（代码 262　#1）
另一个对可分区设备不同的 ioctl 命令是 BLKRRPART。对可分区设备来说重读分区表是有意义的，它等价于在发生磁盘改变后对磁盘的重有效化（revalidating）：
（代码 262　#2）
函数 *spull_revalidate* 接着调用 *resetup_one_dev* 来重建分区表。不过，它首先需要清掉所有以前的信息——不然的话，尾分区还会出现在分区表的后部，如果新分区表含有少于以前的分区数。

（代码 262　#3）
但是 *sbull* 和 *spull* 的主要区别在于请求函数。在 *spull* 中，请求函数必须使用分区信息以在不同次设备号之间正确地传送数据。
spull_gendisk->part 中的信息在物理设备上为每个分区定位。part[minor]->nr_sects 是分区的大小，part[minor]->start_sect 是距磁盘起始位置的偏移。请求函数最终转回到完整磁盘的实现。
下面是在 *spull_request* 中的相关行：
（代码 263#4）
（代码 263#1）
扇区数乘以扇区大小 512（这里直接编写在 *spull* 中的）得到以字节为单位的分区的大小。

中断驱动的块设备驱动程序

当一个驱动程序控制一个实际的硬件设备时，操作一般是中断驱动的。使用中断依靠在 I/O 操作时释放处理器，从而提高系统性能。为了让中断驱动 I/O 能够工作，被控制的设备必须能异步地传送数据并产生中断。
当驱动程序是中断驱动时，请求函数派生出一个数据传送并立即返回，并不调用 *end_request*。不过，核心并不认为请求已经被完成了，直到 *end_request* 被调用时。因此，当设备发出信号表明数据传送已经完成时，上半部或下半部中断处理程序调用 *end_request*。
若不使用系统微处理器，*sbull* 和 *spull* 都不能传送数据；不过 *spull* 装备了伪装中断驱动操作的能力，这通过在装载时指定 irq=1 实现。当 irq 为零时，驱动程序使用一个核心计时器来推迟当前请求的完成。延迟的长度是 irq 的值：值越大，延迟越长。
中断驱动设备的请求函数告诉硬件执行传送并返回。*spull* 函数执行通常的错误检查，并调用 *memcpy* 传送数据（这个任务在实际驱动程序中是异步执行的）。它将确认应答延迟至中断时。
（代码 263　#2）
（代码 264　#1）
当设备正在处理当前请求时，新来的请求可以积累起来，但如果驱动程序正在处理一个请求，核心并不为之调用请求函数。这就是为什么这里显示的函数并不检查双重调用。
在上一个数据传送完成后，设置下一次是中断处理程序的责任。为避免代码重复，处理程序通常调用请求函数，因此请求函数应能在中断时运行（见第 6 章“任务队列的本质”）。
在我们的示例模块中，中断处理程序的角色是通过计时器超时时调用的函数完成的。那个函数调用 *end_request* 并通过调用请求函数来调度下一次数据传送。
（代码 264 2#）
注意这个中断处理程序调用请求函数来调度下一次操作。这意味着在这种情况下，请求函数必须能在中断时进行。
如果你试图以中断驱动风格运行 *spull* 模块，你会明显地注意到增加的延迟。这个设备几乎象它以前一样快，因为缓冲高速缓存避免了内存和物理设备之间的绝大多数数据传送。如果你想感受一下一个慢设备是如何运行的，你可以在加载 *spull* 时为 irq 指定一个大点儿的值。

快速参考

下面总结的是在写一个块设备驱动程序时最重要的函数和宏。不过，为了节省空间，我没有列出结构 request 和 genhd 的域，另外我也省略了预先定义的 *ioctl* 命令。
int register_blkdev(unsigned int major,const char *name,struct file_operations *fops);
int unregister_blkdev(unsigned int major,const char *name);

这两个函数负责在 *init_module* 中的设备注册和 *cleanup_module* 中的设备撤除。

struct blk_dev_struct blk_dev[MAX_BLKDEV];

这个数组用做在核心和驱动程序间请求的传递。blk_dev[major].request_fn 应该在加载时被赋值以指向“当前请求的请求函数”。

int read_ahead[][];

每个主设备号的提前读的值。数值为 8 对硬盘一类的设备比较合理；对慢速介质这个值应该大一点儿。

int read_ahead[];

int blksize_size[][];

int blk_size[][];

这些二维数组由主设备号和次设备号索引。驱动程序负责分配和释放与主设备号关联的矩阵中的行。这些数组分别表示以字节为单位的设备块的大小（通常为 1KB），以 KB 为单位的每个次设备的大小（并非以块为单位），以字节为单位的硬件扇区的大小。

当前，不支持 512 以外的扇区大小，尽管代码中有一个钩子函数（hook）。

MAJOR_NR

DEVICE_NAME

DEVICE_NR(kdev_t device)

DEVICE_INTR

#include <linux/blk.h>

这些宏必须在驱动程序包含头文件之前定义，因为绝大多数头文件要用到它们。

struct request *CURRENT

这个宏指向当前的请求。这个请求结构描述一个要被传送的数据块，被当前驱动程序的 *request_fn* 使用。

#include <linux/gendisk.h>

struct genhd;

普通硬盘使得 Linux 轻松地支持可分区设备。

void resetup_one_dev(struct gendisk *genhd,int ddrive);

这个函数扫描磁盘的分区表，并重写 genhd->part 以反映新的分区。