Broadview
www.broadview.com.cn

深入Linux设备驱动程序内核机制

Internals of Linux Device Driver

◎ 陈学松 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深入 Linux 设备驱动程序内核机制

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

这是一本系统阐述 Linux 设备驱动程序技术内幕的专业书籍，它的侧重点不是讨论如何在 Linux 系统下编写设备驱动程序，而是要告诉读者隐藏在这些设备驱动程序背后的那些内核机制及原理。作者通过对 Linux 内核源码抽丝剥茧般的解读，再辅之以精心设计的大量图片，使读者在阅读完本书后对驱动程序前台所展现出来的那些行为特点变得豁然开朗。

本书涵盖了编写设备驱动程序所需要的几乎所有的内核设施，比如内核模块、中断处理、互斥与同步、内存分配、延迟操作、时间管理，以及新设备驱动模型等内容。为了避免读者迷失在某一技术细节的讨论当中，本书在一个比较高的层面上进行展开，以一种先框架再细节的结构安排极大地简化了读者的阅读与学习。

本书不仅适合那些在 Linux 系统下从事设备驱动程序开发的专业技术人员阅读，也同样适合有志于从事 Linux 设备驱动程序开发或对 Linux 设备驱动程序及 Linux 内核感兴趣的在校学生等阅读。对于没有任何 Linux 设备驱动程序开发经验的初学者，建议先阅读那些讨论“如何”在 Linux 系统下编写设备驱动程序的入门书籍，然后再阅读本书来理解“为什么”要以这样或者那样的方式来编写设备驱动程序。

图书在版编目（CIP）数据

深入 Linux 设备驱动程序内核机制 / 陈学松著. —北京：电子工业出版社，2012.1
ISBN 978-7-121-15052-4

Ⅰ. ①深… Ⅱ. ①陈… Ⅲ. ①Linux 操作系统－程序设计 Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字（2011）第 231765 号

策划编辑：张春雨
责任编辑：白 涛
印 刷：
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：33.75 字数：856 千字
印 次：2012 年 1 月第 1 次印刷
印 数：3000 册 定价：98.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推 荐 序

这不是一本单纯的关于 Linux 设备驱动程序入门的书。它是给有一定的 Linux 设备驱动程序编写经验并且对众多 Linux 底层设备驱动内幕机制感兴趣的读者量身定制的。与市面上已经出版的 Linux 相关方面的图书的不同之处在于，本书并不着重于全面描述 Linux 内核，也不只是简单地告诉你如何去写一个 Linux 下的设备驱动程序。它是从设备驱动程序的视角出发，深入到 Linux 内核去剖析那些和驱动程序实现机制密切相关的技术内幕。比如让你理解为什么在这个地方驱动程序应该使用 work queue 而不是 tasklet，为什么在中断处理例程里应该使用 spin_lock 而不是 mutex_lock……因为只有当你对驱动程序中使用的各种内核实现有了清晰的认识，你才能在日常的工作当中随心所欲地驾驭它们，写出更高性能更安全的代码。知其然，更知其所以然，对于沉迷于技术领域的人而言，这种不断探索的好奇心是对技术工作能长期保持热情的一个基本特质。相对于市面上已经出版的相关书籍而言，本书具有以下两个鲜明的特色：

细节揭秘

目前市场上已经出版的 Linux 内核和驱动程序方面的书籍，大体上可分为两种。一种是侧重于内核本身，鉴于目前 Linux 的内核源码已经十分庞大，这些讲解内核的书有些本身非常全面，作者的写作态度也非常严谨，比如 *Deep Understanding Linux Kernel*，还有新近出版的 *Professional Linux Kernel Architecture*，后者几乎涵盖了新版 Linux 内核中绝大部分重要的构件，但也正因如此，这样的书籍就不可能在与驱动程序相关的机制上留下太多笔墨。另外还有一种是专门讲解 Linux 驱动方面的书籍，典型的有 *Linux Device Driver* 和 *Essential Linux Device Driver*。这些书着重于介绍 Linux 驱动的基本概念和架构，但是对于想了解更多幕后的技术细节的读者来说，《深入 Linux 设备驱动程序内核机制》一书可提供更详细的资源和帮助。通常当你想深入理解一些一般书籍没有描述的机制时，你可能会采用在线搜索或查看源码的方式，但有时这不仅费时也未必能得到满意的答案。本书提供了另一途径让你更系统、有效地理解这些内核机制。我相信对于广大忙于在校学习、职场深造或课题攻关的读者来说，本书可提供很多有益的帮助。

图片说理

这本书另外一个很大的特点是，作者大量使用其精心设计的图片来帮助你清晰地理解一些复杂的概念、流程和架构。这在中文版原创的图书中是很难能可贵的，相对而言外文书在这方面做得就要好很多。形象直观的图片胜过大量的文字，也能节省读者大量的时间。可以看到，本书的作者在这一方面做了很大的努力去加以完善，在我看来，这是一个非常好的尝试。本书作者当前正在 AMD 上海研发中心从事 Linux 显卡驱动等系统软件方面的研发工作，能在繁忙的工作之余，通过对自己学习和实践经验的总结写下这样一本书，对增进国内读者的 Linux 系统开发能力将起到很大的作用。我相信，如果作者有足够的时间与精力的话，这本书还可以进一步完善，包括在某些技术方面可以有更精细的描述。

AMD 图形软件架构师　PMTS　俞辉

2011 年 8 月 24 日于加拿大

前　言

在 Linux 庞大的源码树中，设备驱动程序部分的代码已经占了相当大的比例。现实的工作中，大量的采用 Linux 系统的平台需要设备驱动程序才能把 Linux 的内核真正运行起来，同时通过编写 Linux 设备驱动程序，使得我们经由亲手编写具有特权等级的代码来一探 Linux 内核幕后的秘密成为可能。所以，无论是从日常工作的需要还是只为单纯满足对 Linux 内核机制好奇心的角度来说，学习并掌握 Linux 设备驱动程序的编写都是非常必要的，同时也是一件非常有趣且有意义的事情。

初衷与定位

这本书并不仅仅是单纯地讨论如何在 Linux 系统下编写一个设备驱动程序，因为关于这方面的内容，市面上已经有大量类似的图书可供参考。本书的总体思想是从内核的角度来看设备驱动程序，从设备驱动程序的角度深入到内核中，比如通过对 spin_lock 以及 spin_lock_irq 等内核源码的分析，来告诉你在什么场合下应该使用 spin_lock，什么场合下又应该选择 spin_lock_irq。还有，比如我们几乎每天都会在设备驱动程序所代表的内核模块中使用 MODULE_LICENSE("GPL")这样的声明，这个声明是如此地平凡，以至于我们常常忽略它存在的价值。但是在某个夜深人静的夜晚，感觉长夜漫漫无心睡眠时，在你内心深处的某个地方是否会想过，这个声明对一个内核模块而言，它到底意味着什么，如果没有它，加载这样一个模块对系统又会造成什么样的影响，如此等等，读者都可以在阅读本书的过程中找到答案。

很显然，只有当你清楚地理解了一个东西的内在机制，你才能更好地去使用它们，如果不幸在使用过程中出现问题，也才可以快速将其定位并最终予以解决。台湾著名技术作家侯捷曾引林雨堂先生在《朱门》中的一句话，“只用一样东西，不明白它的道理，实在不高明”，来描述他当时写作时的心境，其实这句话也同样适合我用来阐明写作本书的初衷之一。

但是这并不意味着只有 Linux 系统下设备驱动程序的编写老手才适合阅读本书，因为我在本书写作过程中，一般会先给出一个总体的框架，然后在此基础上对 Linux 提供给设备驱动程序使用的每一个常见而重要的内核设施进行细致地分析，同时辅之以验证性质的代码来使得这种略嫌抽象的讨论具体化，以激发读者对技术探索的兴趣。所以即便是入门级的读者，也可以通过阅读本书来加深对 Linux 下编写设备驱动程序的理解。

另外要说的是，读者不应该寄希望于阅读两三本书就可以掌握 Linux 下设备驱动程序编写的精髓，所有的书籍只能在大体上给你一个参考借鉴的作用，真正的理解还要靠读者自己去努力，诚所谓“纸上得来终觉浅，绝知此事要躬行”。

编排与范围

在本书的结构编排上，我努力使各章节独立起来，但是少量的向前或者向后引用还是必不可少的。但是总体上，我将最基本的篇章尽量放到前面，一些加强型或者高级点的话题尽量放到后边。在描述驱动程序内核机制方面，为了避免单纯的代码解释所带来的抽象感，我会使用具体的例子来将所能看到的驱动程序的前台行为和它的幕后机制串联起来，以帮助读者建立起全面立体的设备驱动程序架构蓝图。不过 Linux 对某些特性的支持因为考虑到各种平台和性能等诸多因素，其实现很可能会有多种不同的方法，比如从内核态驱动程序向用户空间导出信息的文件系统方面，就至少有 proc 和 sysfs 两种形式。因此本书在描述具体的例子时，一定是遵循其中的某种实现，在诸多实现机制的选择上，本书会从实用性和实时性角度出发，采用内核中最新引入或者是最有发展前景的实现，对于某些即将过时的实现机制（因为兼容或者代码维护工作量的关系，一些老的机制可能依然残留在新版的内核代码中），除非出于技术细节的对比或者从增加知识面的角度考虑会有所涉及，否则将不会作为本书的主线。

在代码的引用上，为了突出功能主线部分和削减本书的篇幅，我会删除代码中用来增加调试信息、性能增强及防御性代码这些部分。对于系统体系架构相关的代码，我主要以 x86 与 ARM 平台为主，因为这两者是当前最流行的两种处理器架构。关于本书所参考的 Linux 内核源码的版本，在本书刚开始写作时参考的是 2.6.35 的版本，在写作的中后期，已经将内核版本更新到了 2.6.39，在本书的修订阶段，我已经努力将之前完成的内容更新到了 2.6.39。当然，因为作者时间精力所限，加之 Linux 内核本身就博大精深，内核版本也一直在不断更新变化中，所以书中肯定还会有这样那样潜在的错误，希望读者朋友们能不吝批评指正，以使我们得以共同提高。

创作历程

我有幸自参加工作以来，在 Linux 下从事设备驱动程序相关的开发工作已经有 9 年多的时间，这期间在 Linux 上所接触的平台既有 x86，也有 ARM，甚至包括少量的 PowerPC。在我看来，学习某一操作系统下的设备驱动程序的编写，主要包含两个方面：一个是该操作系统本身对设备驱动程序框架的支持，也可以称之为设备驱动模型，另一个则是对要驱动的硬件的理解。对于后者，设备驱动程序开发者将要面对各种各样的硬件设备，了解它们的最好也最直接的方法当然是这样硬件的 datasheet。前者则主要和操作系统息息相关，比如在 Linux 系统下开发设备驱动程序，必然要熟练掌握 Linux 为设备驱动的编写所提供的

各种内核实施及相关的各种数据结构，本书的内容主要就是探讨 Linux 内核为设备驱动程序编写所提供的所有这些设施的幕后技术。

本书最早的写作酝酿大约在 2010 年 10 月份前后，在此之前，或者是出于自己对以往积累的技术总结的需要，或者是出于将自己的一些技术心得与同行分享的目的，总之，我陆陆续续在一些论坛上发表了若干剖析 Linux 设备驱动程序内核机制的帖子，这些帖子最终使我萌发了用一本书来总结自己以往的 Linux 设备驱动程序开发经验的想法。我把最初的大约一章半的稿子发给了电子工业出版社，很快就得到了策划编辑张春雨先生的肯定，接下来也很顺利地通过了选题的论证，这之后就是一段极其漫长且非常辛苦的写作过程。时间是最大的挑战，由于白天需要工作，写作的时间只能是留给夜晚或者周末，在写作最紧张的时刻，经常要写到凌晨 2 点多。除了时间上的困难之外，如何将一个技术点用最透彻最简洁的语言描述清楚，如何对 Linux 内核中纷繁复杂的内容进行取舍，这些也都是非常耗费精力的事情。技术本身的理解也许并不困难，难在如何去把你心中掌握的东西清晰准确地以文字的方式表达出来，这不同于论坛的发帖，可以非常自由甚至随心所欲，写书的话，必须考虑它的完整性、逻辑性以及可读性，同时还要考虑将来潜在的读者群。尤其是如果你想认认真真写一本书的话，有时候甚至需要反复推敲一个技术点的表达方式。在写作灵感枯竭的时候，看着时间飞快掠过，而眼前的文档却没有留下几行字，那种强烈的挫折感与沮丧感真得会让人动摇自己的信念：自己是否还能坚持下去？！所以当这本书即将出版时，我还很有些恍惚，不敢相信自己居然磕磕绊绊地最终完成了这些书稿。

意见反馈

读者如果在阅读本书的过程中有任何意见或者建议，欢迎通过下面的 E-mail 与我取得联系：ricard_chen@yahoo.com。

关于本书使用到的源代码，读者可在 www.embexperts.com 网站上下载。另外，关于本书后续的一些勘误、某些技术细节方面的讨论也会在该网站相应的版面上进行。

致谢

首先，我要感谢我的家人，如前所述，写书占去了我大量的业余时间，我的父母和怀孕的妻子在此期间承担了几乎所有的家务劳动，替我捣腾出不少的写作时间，感谢她们！我的宝贝女儿在今年 8 月 15 日健康出生，成为我的家庭中新的一员，这本书也正好可以作为父亲的见面礼送给她——可爱的萌萌同学。

其次是电子工业出版社的张春雨与白涛编辑，从选题的论证到文字编辑，他们都付出了极其辛苦的劳动并且提出了很多有益的建议，那些逝去的不堪回首岁月里满眼尘封的 E-mail 见证了这一点！当然，还要感谢我现在所效力的 AMD 公司，因为它使得我不必为生活所

迫去写一本书，对技术的热情与兴趣才是我最终得以坚持下来的最大因素。

最后，在本书的审核方面，AMD 的 PMTS 及显卡驱动软件架构师俞辉在百忙中为本书作序并审核了部分章节，AMD 上海研发中心 Linux Graphic Base Driver 团队的 Lisa Wu 及研发经理刘刚也为本书的写作提供了支持，诺基亚与西门子的研发经理胡兵全审核了本书第 1 章及第 12 章，EMC 的 PE Thomas 审核了本书第 3 章及第 4 章。Marvell 的资深软件工程师 James Lai 亦审核了本书部分章节并有宝贵意见，在此一并表示感谢！

陈学松

2011 年 8 月 29 日于上海

目　录

第 2 章

字符设备驱动程序

现实世界中存在着大量的设备，这些设备在电气特性和 I/O 方式上都各不相同。为了简化设备驱动程序员的工作，Linux 系统从这些各异的设备中提取出了共性的特征，将其划分为三大类：字符设备、块设备和网络设备。内核针对每一类设备都提供了对应的驱动模型框架，包括基本的内核设施和文件系统接口。这样设备驱动程序员在写某类设备驱动程序时，就有一套完整的驱动模型框架可以使用，从而可将大量的精力放在设备本身的操作上。图 2-1 展示了一个粗略的 Linux 设备驱动程序结构图：

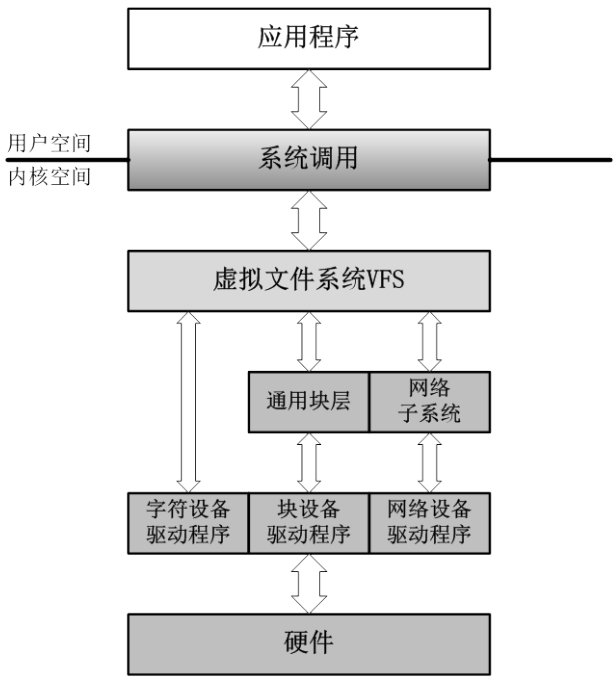

图 2-1　Linux 设备驱动程序结构图

其中，字符设备驱动程序是这三类设备驱动程序中最常见，也是相对比较容易理解的一种，现实中的大部分硬件都可由字符设备驱动程序来控制。这类硬件的特征是，在 I/O 传输过程中以字符为单位，这种字符流的传输速率通常都比较缓慢（因而其内核设施中不提供缓存机制），常见的如键盘、鼠标及打印机等设备。

本章将详细讨论构成字符设备驱动程序的内核设施的幕后机制，此外还将讨论应用程序如何与字符设备驱动程序进行交互，也即应用程序如何使用字符设备驱动程序提供的服务，这将涉及字符设备的文件系统接口等相关内容。

字符设备驱动程序所提供的功能是以设备文件[1]的形式提供给用户空间程序使用，本章将首先讨论应用程序与设备文件，然后再深入探讨字符设备驱动程序的内核机制。

2.1 应用程序与设备驱动程序互动实例

在深入讨论字符设备驱动程序之前，我们先用一个实际的例子来展示应用程序如何与字符设备驱动程序进行交互。这个例子中，我们首先给出一个简单的字符设备驱动程序的内核模块，接着通过 insmod 工具将这个内核模块加入到系统中，之后通过 mknod 来创建一个设备文件节点（在这个例子中我们将手动创建设备文件节点，不过后面会讨论设备节点的自动生成机制），最后再编写一个小的应用程序，用该应用程序来调用前面设备驱动程序所提供的服务。因为这里只是展示应用程序与设备驱动程序相互交互的环节，所以无论是设备驱动程序还是应用程序，都尽量保持简单且与具体硬件设备无关。在本章后续的小节中将仔细讨论这个例子中所有关键环节的幕后技术细节。

字符设备驱动程序源码

<demo_chr_dev.c>

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>

static struct cdev chr_dev; //定义一个字符设备对象
static dev_t ndev; //字符设备节点的设备号

static int chr_open(struct inode *nd, struct file *filp)
{
        int major = MAJOR(nd->i_rdev);
        int minor = MINOR(nd->i_rdev);
        printk("chr_open, major=%d, minor=%d\n", major, minor);
        return 0;
}

static ssize_t chr_read(struct file *f, char __user *u, size_t sz, loff_t *off)
```

[1] 本书中“设备文件”、“设备文件节点”和“设备节点”表述的是同一个意思。

```
{
        printk("In the chr_read() function!\n");
        return 0;
}

//字符设备驱动程序中非常关键的一个数据结构 struct file_operations
struct file_operations chr_ops =
{
        .owner = THIS_MODULE,
        .open = chr_open,
        .read = chr_read,
};

//模块的初始化函数
static int demo_init(void)
{
        int ret;
        cdev_init(&chr_dev, &chr_ops); //初始化字符设备对象
        ret = alloc_chrdev_region(&ndev, 0, 1, "chr_dev"); //分配设备号
        if(ret < 0)
                return ret;
        printk("demo_init():major=%d, minor=%d\n", MAJOR(ndev), MINOR(ndev));
        ret = cdev_add(&chr_dev, ndev, 1);//将字符设备对象 chr_dev 注册进系统
        if(ret < 0)
                return ret;

        return 0;
}

static void demo_exit(void)
{
        printk("Removing chr_dev module...\n");
        cdev_del(&chr_dev); //将字符设备对象 chr_dev 从系统中注销掉
        unregister_chrdev_region(ndev, 1); //释放分配的设备号
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dennis @AMDLinuxFGL");
MODULE_DESCRIPTION("A char device driver as an example");
```

以上就是一个字符设备驱动程序的源码，虽然极其简单，以至于没有做任何有实质意义的事情，但是它展示了字符设备驱动程序的典型框架结构，字符设备驱动程序中绝大多数的关键元素都出现在了上面这个示例程序中，它们将成为本章后续讨论的核心。

读者可以参照下面这个简单的 Makefile 文件来编译上述的模块：

```
obj-m := demo_chr_dev.o
KERNELDIR := /lib/modules/$(shell uname -r) /build[2]
PWD := $(shell pwd)

default:
	$(MAKE)    -C $(KERNELDIR) M=$(PWD)    modules
clean:
	rm -f *.o *.ko *.mod.c
```

如果一切顺利，将得到一个名为 demo_chr_dev.ko 的内核模块。

应用程序源码

<main.c>

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define CHR_DEV_NAME "/dev/chr_dev"

int main()
{
	int ret;
	char buf[32];
	int fd = open(CHR_DEV_NAME, O_RDONLY|O_NDELAY);
	if(fd < 0)
	{
		printf("open file %s failed!\n", CHR_DEV_NAME);
		return -1;
	}
	read(fd, buf, 32);
	close(fd);

	return 0;
}
```

读者可以用 gcc 来生成该应用程序的可执行文件 main：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# gcc main.c -o main
```

应用程序主要是用 open 打开一个设备文件节点，然后在打开的设备文件描述符 fd 上调用

[2] 读者可能需要根据自己系统中实际的内核源码路径来修改这里的 KERNELDIR 值，以消除可能出现的编译错误。

read 函数。调用 read 函数时，除了 fd 必须使用外，其他两个参数完全是为了满足 read 函数调用的需要，设备驱动程序中的 chr_read 不会用到这些参数。

这个简单的例子将展示应用程序如何通过文件系统调用，穿越到内核空间，呼叫到设备驱动程序实现的各种接口函数。本章稍后将和读者一道去探讨这个示例程序背后所包含的技术细节，等到对字符设备驱动程序的各种内核设施及文件系统的接口有了深入的理解，相信在实际的工作中一定可以自由地驾驭它们，即便遇到问题也可以快速定位和解决。

示例操作步骤

现在用 insmod 把 demo_chr_dev.ko 加入到系统：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/gene-module# insmod demo_chr_dev.ko
```

dmesg 针对这个 insmod 的输出信息为：

```
[19611.946440] demo_init():major=248, minor=0
```

通过上面 dmesg 的输出信息，我们知道 alloc_chrdev_region 函数给内核模块 demo_chr_dev.ko 分配的主设备号为 248，次设备号为 0。根据这个设备号信息用 mknod 命令在系统的/dev 目录下为该模块生成一个新的设备文件节点：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# mknod /dev/chr_dev c 248 0
```

如果一切正常，那么在/dev 目录下就会产生一个新的设备文件节点“/dev/chr_dev”，可以用 ls 命令来仔细观察一下它：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# ls -l /dev/chr_dev
crw-r--r-- 1 root root 248, 0 2011-05-11 21:41 /dev/chr_dev
```

上面 ls 命令的输出反映出了设备节点“/dev/chr_dev”的如下一些关键信息：

“crw-r--r--”中的字符“c”表明这是个字符设备文件，248 是该设备节点的主设备号，次设备号则是 0，这跟我们的预期是完全一致的。

有了对应的设备文件之后，现在可以运行我们的应用程序了：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# ./main
```

查看 dmesg 对此的输出信息：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# dmesg -c
[20340.589750] chr_open, major=248, minor=0
[20340.589760] In the chr_read() function!
```

对比前面内核模块 demo_chr_dev.ko 的源码，读者应该知道上述两行的输出分别来自内核模块中的 chr_open 和 chr_read 函数，虽然在这个示例程序中它们几乎没做任何事情，但是我们见证了应用程序成功调用到了设备驱动程序实现的函数，这正是我们所预期的目标。

2.2 struct file_operations

在开始讨论字符设备驱动程序内核机制前，有必要先交代一下 struct file_operations 数据结构，其定义如下：

```
<include/linux/fs.h>
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
                                           unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
};
```

可以看到，struct file_operations 的成员变量几乎全是函数指针，因为本书的后续章节会陆续讨论到这个结构体中绝大多数成员的实现，所以这里不再解释其各自的用途。读者也许很快会发现，现实中字符设备驱动程序的编写，其实基本上是围绕着如何实现 struct file_operations 中的那些函数指针成员而展开的。通过内核文件系统组件在其间的穿针引

线，应用程序中对文件类函数的调用，比如 read()等，将最终被转接到 struct file_operations 中对应函数指针的具体实现上。

该结构中唯一非函数指针类成员 owner，表示当前 struct file_operations 对象所属的内核模块，几乎所有的设备驱动程序都会用 THIS_MODULE 宏给 owner 赋值，该宏的定义为：

<include/linux/module.h>

```
#define THIS_MODULE (&__this_module)
```

__this_module 是内核模块的编译工具链为当前模块产生的 struct module 类型对象，所以 THIS_MODULE 实际上是当前内核模块对象的指针，file_operations 中的 owner 成员可以避免当 file_operations 中的函数正在被调用时，其所属的模块被从系统中卸载掉。如果一个设备驱动程序不是以模块的形式存在，而是被编译进内核，那么 THIS_MODULE 将被赋值为空指针，没有任何作用。

2.3　字符设备的内核抽象

顾名思义，字符设备驱动程序管理的核心对象是字符设备。从字符设备驱动程序的设计框架角度出发，内核为字符设备抽象出了一个具体的数据结构 struct cdev，其定义如下：

<include/linux/cdev.h>

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

在本章后续的内容中将陆续看到它们的实际用法，这里只把这些成员的作用简单描述如下：

struct kobject kobj

内嵌的内核对象，其用途将在“Linux 设备驱动模型”一章中讨论。

struct module *owner

字符设备驱动程序所在的内核模块对象指针。

const struct file_operations *ops

字符设备驱动程序中一个极其关键的数据结构，在应用程序通过文件系统接口呼叫到

设备驱动程序中实现的文件操作类函数的过程中，ops 指针起着桥梁纽带的作用。

struct list_head list

用来将系统中的字符设备形成链表。

dev_t dev

字符设备的设备号，由主设备号和次设备号构成。

unsigned int count

隶属于同一主设备号的次设备号的个数，用于表示由当前设备驱动程序控制的实际同类设备的数量。

设备驱动程序中可以用两种方式来产生 struct cdev 对象。一是静态定义的方式，比如在前面的那个示例程序中，通过下列代码静态定义了一个 struct cdev 对象：

```
static struct cdev chr_dev;
```

另一种是在程序的执行期通过动态分配的方式产生，比如：

```
static struct cdev *p = kmalloc(sizeof(struct cdev), GFP_KERNEL);
```

其实 Linux 内核源码中提供了一个函数 cdev_alloc，专门用于动态分配 struct cdev 对象。cdev_alloc 不仅会为 struct cdev 对象分配内存空间，还会对该对象进行必要的初始化：

<fs/char_dev.c>

```
struct cdev *cdev_alloc(void)
{
        struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
        if (p) {
                INIT_LIST_HEAD(&p->list);
                kobject_init(&p->kobj, &ktype_cdev_dynamic);
        }
        return p;
}
```

需要注意的是，内核引入 struct cdev 数据结构作为字符设备的抽象，仅仅是为了满足系统对字符设备驱动程序框架结构设计的需要，现实中一个具体的字符硬件设备的数据结构的抽象往往要复杂得多，在这种情况下 struct cdev 常常作为一种内嵌的成员变量出现在实际设备的数据机构中，比如：

```
struct my_keypad_dev{
        //硬件相关的成员变量
        int a;
        int b;
```

```
        int c;
        …
        //内嵌的 struct cdev 数据结构
        struct cdev cdev;
};
```

在这样的情况下，如果要动态分配一个 struct real_char_dev 对象，cdev_alloc 函数显然就无能为力了，此时只能使用下面的方法：

```
static struct real_char_dev *p = kzalloc(sizeof(struct real_char_dev), GFP_KERNEL);
```

前面讨论了如何分配一个 struct cdev 对象，接下来的一个话题是如何初始化一个 cdev 对象，内核为此提供的函数是 cdev_init：

<fs/char_dev.c>

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
        memset(cdev, 0, sizeof *cdev);
        INIT_LIST_HEAD(&cdev->list);
        kobject_init(&cdev->kobj, &ktype_cdev_default);
        cdev->ops = fops;
}
```

函数的代码非常直白，不再赘述。一个 struct cdev 对象在被最终加入系统前，都应该被初始化，无论是直接通过 cdev_init 或者是其他途径。理由很简单，这是 Linux 系统中字符设备驱动程序框架设计的需要。

照理在谈完 cdev 对象的分配和初始化之后，下面应该讨论如何将一个 cdev 对象加入到系统了，但是由于这个过程需要用到设备号相关的技术点，所以暂且先来探讨设备号的问题。

2.4　设备号的构成与分配

本节开始讨论设备号相关的问题，不过设备号对于设备驱动程序而言究竟意味着什么，换句话说，它在内核中起着怎样的作用，本节暂不讨论，这里只关心它在内核中是如何分配和管理的。

2.4.1　设备号的构成

Linux 系统中一个设备号由主设备号和次设备号构成，Linux 内核用主设备号来定位对应的设备驱动程序，而次设备号则由驱动程序使用，用来标识它所管理的若干同类设备。因此，从这个角度而言，设备号作为一种系统资源，必须仔细加以管理，以防止因设备号与驱动程序错误的对应关系所带来的混乱。

Linux 用 dev_t 类型变量来标识一个设备号，这是个 32 位的无符号整数：

<include/linux/types.h>

```
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t        dev_t;
```

图 2-2 显示了 2.6.39 版本内核中设备号的构成：

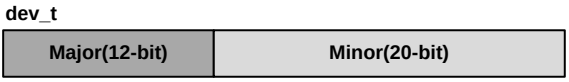

图 2-2 Linux 的设备号的构成

在这一内核版本中，dev_t 的低 20 位用来表示次设备号，高 12 位用来表示主设备号。随着内核版本的演变，上述的主次设备号的构成也许会发生改变，所以设备驱动程序开发者应该避免直接使用主次设备号所占有的位宽来获得对应的主设备号或次设备号。为了保证在主次设备号位宽发生改变时，现有的程序依然可以正常工作，内核提供了如下几个宏供设备驱动程序操作设备号时使用：

<include/linux/kdev_t.h>

```
#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

MAJOR 宏用来从一个 dev_t 类型的设备号中提取出主设备号，MINOR 宏则用来提取设备号中的次设备号。MKDEV 则是将主设备号 ma 和次设备号 mi 合成一个 dev_t 类型的设备号。在上述宏定义中，MINORBITS 宏在 2.6.39 版本中定义的值是 20，如果之后的内核对主次设备号所占用的位宽重新进行调整，例如将 MINORBITS 改成 12，只要设备驱动程序坚持使用 MAJOR、MINOR 和 MKDEV 来操作设备号，那么这部分代码应该无须修改就可以在新内核中运行。

2.4.2 设备号的分配与管理

在内核源码中，涉及设备号分配与管理的函数主要有以下两个：

- register_chrdev_region 函数

该函数的代码实现如下：

<fs/char_dev.c>

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
{
        struct char_device_struct *cd;
```

```
        dev_t to = from + count;
        dev_t n, next;

        for (n = from; n < to; n = next) {
                next = MKDEV(MAJOR(n)+1, 0);
                if (next > to)
                        next = to;
                cd = __register_chrdev_region(MAJOR(n), MINOR(n),
                                next - n, name);
                if (IS_ERR(cd))
                        goto fail;
        }
        return 0;
fail:
        to = n;
        for (n = from; n < to; n = next) {
                next = MKDEV(MAJOR(n)+1, 0);
                kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
        }
        return PTR_ERR(cd);
}
```

该函数的第一参数 from 表示的是一个设备号，第二参数 count 是连续设备编号的个数，代表当前驱动程序所管理的同类设备的个数，第三参数 name 表示设备或者驱动的名称。register_chrdev_region 的核心功能体现在内部调用的__register_chrdev_region 函数中，在讨论这个函数之前，先要看一个全局性的指针数组 chrdevs，它是内核用于设备号分配与管理的核心元素，其定义如下：

<fs/char_dev.c>

```
static struct char_device_struct[3] {
        struct char_device_struct *next;
        unsigned int major;
        unsigned int baseminor;
        int minorct;
        char name[64];
        struct cdev *cdev;          /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE[4]];
```

这个数组中的每一项都是一个指向 struct char_device_struct 类型的指针。系统刚开始运行时，该数组的初始状态如图 2-3 所示：

---

[3] 这个结构体中的成员变量 cdev 在设备号管理模块中没有任何用处，至少在目前看来是这样。如果不是保留用于将来的某种扩展，那么可以预见，在不久的将来这个成员最终会被清除掉，正如源码注释中所说的那样，“will die”。

[4] 在 2.6.39 版本的内核源码中，CHRDEV_MAJOR_HASH_SIZE 定义的值为 255。

现在回过头来看看 register_chrdev_region 函数，这个函数要完成的主要功能是将当前设备驱动程序要使用的设备号记录到 chrdevs 数组中，有了这种对设备号使用情况的跟踪，系统就可以避免不同的设备驱动程序使用同一个设备号的情形出现。这意味着当设备驱动程序调用这个函数时，事先已经明确知道它所要使用的设备号，之所以调用这个函数，是要将所使用的设备号纳入到内核的设备号管理体系中，防止别的驱动程序错误使用到。当然如果它试图使用的设备号已经被之前某个驱动程序使用了，调用将不会成功，register_chrdev_region 函数将会返回一个负的错误码告知调用者，如果调用成功，函数返回 0。

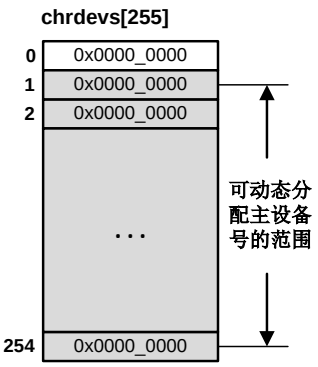

图 2-3　初始状态的 chrdevs 数组结构

上述这些设备号功能的实现其实最终发生在 register_chrdev_region 函数内部所调用的__register_chrdev_region 函数中，它会首先分配一个 struct char_device_struct 类型的对象 cd，然后对其进行一些初始化：

<fs/char_dev.c>

```
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                         int minorct, const char *name)
{
        cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);
        …
        cd->major = major;
        cd->baseminor = baseminor;
        cd->minorct = minorct;
        strlcpy(cd->name, name, sizeof(cd->name));
```

这个过程完成之后，它开始搜索 chrdevs 数组，搜索是以哈希表的形式进行的，为此必须首先获取一个散列关键值，正如读者所预料的那样，它用主设备号来生成这个关键值：

```
i = major_to_index(major);
```

这是个非常简单的获得散列关键值的方法，i = major % 255。此后函数将对 chrdevs[i]元素管理的链表进行扫描，如果 chrdevs[i]上已经有了链表节点，表明之前有别的设备驱动程序

使用的主设备号散列到了 chrdevs[i]上，为此函数需要相应的逻辑确保当前正在操作的设备号不会与这些已经在使用的设备号发生冲突，如果有冲突，函数将返回错误码，表明本次调用没有成功。如果本次调用使用的设备号与 chrdevs[i]上已有的设备号没有发生冲突，先前分配的 struct char_device_struct 对象 cd 将加入到 chrdevs[i]领衔的链表中成为一个新的节点。没有必要再仔细分析__register_chrdev_region 函数中的相关代码了，接下来以一个具体的例子来了解这一过程。

在 chrdevs 数组尚处于初始状态的情形下，假设现在有一个设备驱动程序要使用的主设备号是 257，次设备号分别是 0、1、2 和 3（意味着该驱动程序将管理四个同类型的设备）。它对 register_chrdev_region 函数的调用如下：

```
int ret = register_chrdev_region(MKDEV(257, 0), 4, "demodev");
```

上述对 register_chrdev_region 函数的调用完毕后，chrdevs 数组的状态将变成图 2-4 所示（图中假设新分配的 struct char_device_struct 节点的基地址为 0xC8000004，这些节点基地址数值只是用来使读者有个直观的概念，并非代表系统中实际分配的地址值）：

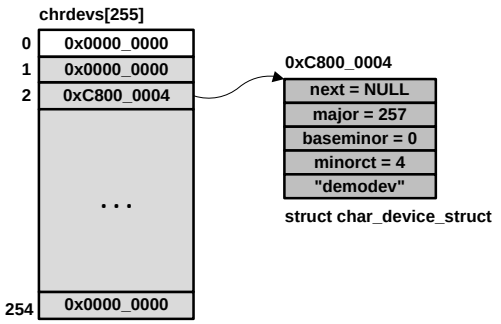

图 2-4　主设备号 257 注册后的 chrdevs 数组状态

现在假设有另一个设备驱动程序使用的主设备号为 2，次设备号为 0，当它调用 register_chrdev_region(MKDEV(2, 0), 1, "augdev")来向系统注册设备号时，因为 2 % 255 = 2，所以也将索引到 chrdevs 数组的第 2 项。虽然数组的第 2 项中已经有"demodev"设备在使用，但是因为这次注册的设备号是 MKDEV(2, 0)，与设备"demodev"的设备号 MKDEV(257, 0)并不冲突，所以注册总会成功。因为 Linux 在将设备"augdev"对应的 struct char_device_struct 对象节点加入到哈希表中时，采用了插入排序，这导致同一哈希列表将按照 major 的大小递增排列，因此此时的 chrdevs 数组状态如图 2-5 所示：

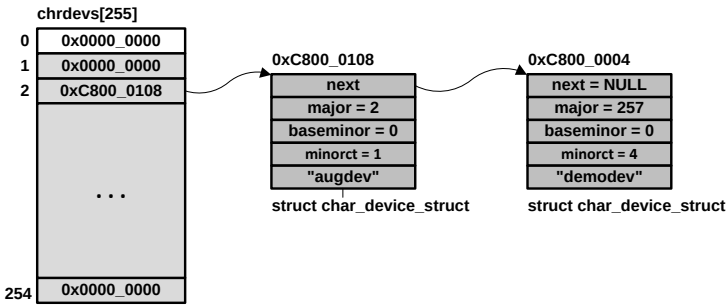

图 2-5　主设备号 2 加入后的 chrdevs 数组状态

一个有趣的事实是，在图 2-5 的基础上，假设有另一个设备驱动程序调用 register_chrdev_region 函数向系统注册，主设备号也为 257，那么只要其次设备号所在的范围[baseminor, baseminor + minorct]不与设备"demodev"的次设备号范围发生重叠，系统依然会生成一个新的 struct char_device_struct 节点并加入到对应的哈希链表中。在主设备号相同的情况下，如果次设备号的范围有重叠，则意味着有设备号的冲突，这将导致对 register_chrdev_region 函数的调用失败。对主设备号相同的若干 struct char_device_struct 对象，当系统将其加入链表时，将根据其 baseminor 成员的大小进行递增排序。

- alloc_chrdev_region 函数

该函数由系统协助分配设备号，分配的主设备号范围将在 1~254 之间，其定义如下：

<fs/char_dev.c>

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
                        const char *name)
{
        struct char_device_struct *cd;
        cd = __register_chrdev_region(0, baseminor, count, name);
        if (IS_ERR(cd))
                return PTR_ERR(cd);
        *dev = MKDEV(cd->major, cd->baseminor);
        return 0;
}
```

这个函数的核心调用也是__register_chrdev_region，相对于 register_chrdev_region，alloc_chrdev_region 在调用__register_chrdev_region 时，第一个参数为 0，这将导致__register_chrdev_region 执行下面的逻辑：

<fs/char_dev.c>

```
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                         int minorct, const char *name)
```

```
{
        …
        if (major == 0) {
                for (i = ARRAY_SIZE(chrdevs)-1; i > 0; i--) {
                        if (chrdevs[i] == NULL)
                                break;
                }
                if (i == 0) {
                        ret = -EBUSY;
                        goto out;
                }
                major = i;
                ret = major;
        }
        …
}
```

上述代码片段的实现原理非常简单，它在 for 循环中从 chrdevs 数组的最后一项（也就是第 254 项）依次向前扫描，如果发现该数组中的某项，比如第 i 项，对应的数值为 NULL，那么就把该项对应的索引值 i 作为分配的主设备号返回给驱动程序，同时生成一个 struct char_device_struct 节点，并将其加入到 chrdevs[i]对应的哈希链表中。如果从第 254 项一直到第 1 项，这其中所有的项对应的指针都不为 NULL，那么函数失败并返回一非 0 值，表明动态分配设备号失败。如果分配成功，所分配的主设备号将记录在 struct char_device_struct 对象 cd 中，并将该对象返回给 alloc_chrdev_region 函数，后者通过下面的代码将新分配的设备号返回给函数的调用者：

```
*dev = MKDEV(cd->major, cd->baseminor);
```

设备号作为一种系统资源，当所对应的设备驱动程序被卸载时，很显然要把其所占用的设备号归还给系统，以便分配给其他内核模块使用。不管是用 register_chrdev_region 还是 alloc_chrdev_region 注册或者分配的设备号，在 Linux 中都由下面的函数负责释放：

<fs/char_dev.c>

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

函数在 chrdevs 数组中查找参数 from 和 count 所对应的 struct char_device_struct 对象节点，找到以后将其从链表中删除并释放该节点所占用的内存，从而将对应的设备号释放以供其他设备驱动模块使用。

以上讨论了内核中用于设备号分配与管理的技术细节，焦点是 register_chrdev_region 和 alloc_chrdev_region 两个函数，除了 alloc_chrdev_region 还具有让系统协助分配一个主设备号的功能外，它们最主要的作用其实都是通过 chrdevs 数组来跟踪系统中设备号的使用情况，以防止实际使用中出现设备号冲突的情况。这是内核提供给设备驱动程序使用的一种预防性措施，并没有必然的理由说设备驱动程序一定要使用这两个函数，如果可以确定设

备驱动程序将要使用的设备号不会与系统中已有的设备号发生冲突，完全可以绕开它们。但很明显这是一种非常糟糕的习惯，如果某些设备驱动程序没有使用系统提供的 register_chrdev_region 或者 alloc_chrdev_region 函数，那么系统将失去一个对设备号使用情况进行跟踪的措施。既然内核在设备驱动程序的框架设计中定义了这种规则，作为设备驱动程序的实际开发者，没有理由不去遵循这些规则。

2.5 字符设备的注册

前面已经讨论了字符设备对象的分配、初始化及设备号等概念，在一个字符设备初始化阶段完成之后，就可以把它加入到系统中，这样别的模块才可以使用它。把一个字符设备加入到系统中所需调用的函数为 cdev_add，它在 Linux 源码中的实现如下：

<fs/char_dev.c>

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
        p->dev = dev;
        p->count = count;
        return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

其中，参数 p 为要加入系统的字符设备对象的指针，dev 为该设备的设备号，count 表示从次设备号开始连续的设备数量。

cdev_add 的核心功能通过 kobj_map 函数来实现，后者通过操作一个全局变量 cdev_map 来把设备（*p）加入到其中的哈希链表中。cdev_map 的定义如下：

<fs/char_dev.c>

```
static struct kobj_map *cdev_map;
```

这是一个 struct kobj_map 指针类型的全局变量，在 Linux 系统启动期间由 chrdev_init 函数负责初始化。struct kobj_map 的定义如下：

<drivers/base/map.c>

```
struct kobj_map {
        struct probe {
                struct probe *next;
                dev_t dev;
                unsigned long range;
                struct module *owner;
                kobj_probe_t *get;
                int (*lock)(dev_t, void *);
                void *data;
        } *probes[255];
```

```
    struct mutex *lock;
};
```

kobj_map 函数中哈希表的实现原理和前面注册分配设备号中的几乎完全一样，通过要加入系统的设备的主设备号 major（major=MAJOR(dev)）来获得 probes 数组的索引值 i（i = major % 255），然后把一个类型为 struct probe 的节点对象加入到 probes[i]所管理的链表中，如图 2-6 所示。其中 struct probe 所在的矩形块中的深色部分是我们重点关注的内容，记录了当前正在加入系统的字符设备对象的有关信息。其中，dev 是它的设备号，range 是从次设备号开始连续的设备数量，data 是一 void *变量，指向当前正要加入系统的设备对象指针 p。图 2-6 展示了两个满足主设备号 major % 255 = 2 的字符设备通过调用 cdev_add 之后，cdev_map 所展现出来的数据结构状态。

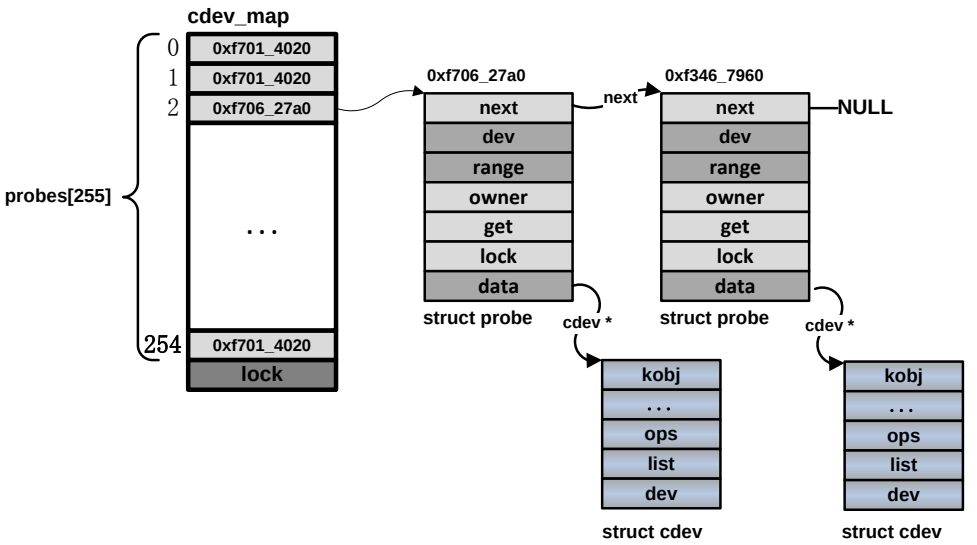

图 2-6　通过 cdev_add 向系统中加入设备

所以，简单地说，设备驱动程序通过调用 cdev_add 把它所管理的设备对象的指针嵌入到一个类型为 struct probe 的节点之中，然后再把该节点加入到 cdev_map 所实现的哈希链表中。

对系统而言，当设备驱动程序成功调用了 cdev_add 之后，就意味着一个字符设备对象已经加入到了系统，在需要的时候，系统就可以找到它。对用户态的程序而言，cdev_add 调用之后，就已经可以通过文件系统的接口呼叫到我们的驱动程序，本章稍后将会详细描述这一过程。

不过在开始文件系统如何通过 cdev_map 来使用驱动程序提供的服务这个话题之前，我们要来看看与 cdev_add 相对应的另一个函数 cdev_del。其实光通过这个函数名，读者想必也想到这个函数的作用了：在 cdev_add 中我们动态分配了 struct probe 类型的节点，那么当对应的设备从系统中移除时，显然需要将它们从链表中删除并释放节点所占用的内存空间。在

cdev_map 所管理的链表中查找对应的设备节点时使用了设备号。cdev_del 函数的实现如下：

<fs/char_dev.c>

```
void cdev_del(struct cdev *p)
{
        cdev_unmap(p->dev, p->count);
        kobject_put(&p->kobj);
}
```

对于以内核模块形式存在的驱动程序，作为通用的规则，模块的卸载函数应负责调用这个函数来将所管理的设备对象从系统中移除。

2.6　设备文件节点的生成

在 Linux 系统下，设备文件是种特殊的文件类型，其存在的主要意义是沟通用户空间程序和内核空间驱动程序。换句话说，用户空间的应用程序要想使用驱动程序提供的服务，需要经过设备文件来达成。当然，如果你的驱动程序只是为内核中的其他模块提供服务，则没有必要生成对应的设备文件。

按照通用的规则，Linux系统所有的设备文件都位于/dev目录下。/dev目录在Linux系统中算是一个比较特殊的目录，在Linux系统早期还不支持动态生成设备节点时，/dev目录就是挂载的根文件系统下的/dev，对这个目录下所有文件的操作使用的是根文件系统提供的接口。比如，如果Linux系统挂载的根文件系统是ext3，那么对/dev目录下所有目录/文件的操作都将使用ext3 文件系统的接口。随着后来Linux内核的演进，开始支持动态设备节点的生成[5]，使得系统在启动过程中会自动生成各个设备节点，这就使得/dev目录不必要作为一个非易失的文件系统的形式存在。因此，当前的Linux内核在挂载完根文件系统之后，会在这个根文件系统的/dev目录上重新挂载一个新的文件系统devtmpfs，后者是个基于系统RAM的文件系统实现。当然，对动态设备节点生成的支持并不意味着一定要将根文件系统中的/dev目录重新挂载到一个新的文件系统上，事实上动态生成设备节点技术的重点并不在文件系统上面。

动态设备节点的特性需要其他相关技术的支持，在后续的章节中会详细描述这些特性。目前先假定设备节点是通过 Linux 系统下的 mknod 命令静态创建。为方便叙述，下面用一个具体的例子来描述设备文件产生过程中的一些关键要素，这个例子的任务很简单：在一个 ext3 类型的根文件系统中的/dev 目录下用 mknod 命令来创建一个新的设备文件节点 demodev，对应的驱动程序使用的设备主设备号为 2，次设备号是 0，命令形式为：

```
root@LinuxDev:/home/dennis# mknod /dev/demodev c 2 0
```

上述命令成功执行后，将会在/dev 目录下生成一个名为 demodev 的字符设备节点。如果用 strace 工具来跟踪一下上面的命令，会发现如下输出（删去了若干不相关部分）：

```
root@LinuxDev:/home/dennis# strace mknod /dev/demodev c 2 0
execve("/bin/mknod", ["mknod", "/dev/demodev", "c", "30","0"], [/* 36 vars */]) = 0
…
mknod("/dev/demodev", S_IFCHR|0666, makedev(30,0)) = 0
```

[5] 这里动态生成设备节点的说法是相对于使用 mknod 命令生成设备节点而言的，前者直接通过文件系统接口来生成对应的设备节点。

…

可见 Linux 下的 mknod 命令最终是通过调用 mknod 函数来实现的，调用时的重要参数有两个，一是设备文件名（"/dev/demodev"），二是设备号（makedev(30,0)）。设备文件名主要在用户空间使用（比如用户空间程序调用 open 函数时），而内核空间则使用 inode 来表示相应的文件。本书只关注内核空间的操作，对于前面的 mknod 命令，它将通过系统调用 sys_mknod 进入内核空间，这个系统调用的原型是：

```
<include/linux/syscalls.h>
long sys_mknod(const char __user *filename, int mode, unsigned dev);
```

注意sys_mknod的最后一个参数dev，它是由用户空间的mknod命令构造出的设备号。sys_mknod系统调用将通过/dev目录上挂载的文件系统接口来为/dev/demodev生成一个新的inode[6]，设备号将被记录到这个新的inode对象上。

图 2-7 展示了通过 ext3 文件系统在/dev 目录下生成一个新的设备节点/dev/demodev 的主要流程。

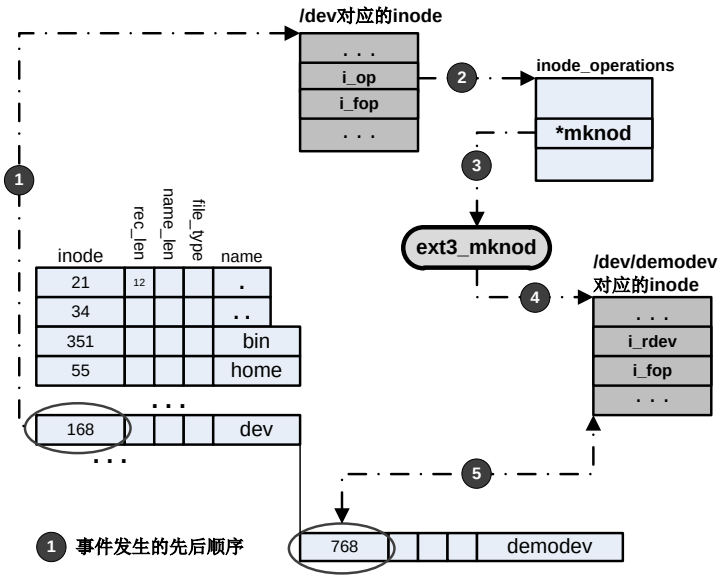

图 2-7 ext3 文件系统 mknod 的主要流程

完整了解设备节点产生的整个过程需要知晓 VFS 和特定文件系统的技术细节。然而从驱动程序员的角度来说，没有必要知道文件系统相关的所有细节，只需关注文件系统和驱动程序间是如何建立上关联的就足够了。

6 对于实际的文件系统，比如 ext3 文件系统，产生一个 node 的过程因为同时要涉及底层存储设备的操作，因而会变得很复杂。

sys_mknod 首先在根文件系统 ext3 的根目录“/”下寻找 dev 目录所对应的 inode，图中对应的 inode 编号为 168，ext3 文件系统的实现会通过某种映射机制，通过 inode 编号最终得到该 inode 在内存中的实际地址（图中由标号 1 的线段表示）。接下来会通过 dev 的 inode 结构中的 i_op 成员指针所指向的 ext3_dir_inode_operations（这是个 struct inode_operations 类型的指针），来调用该对象中的 mknod 方法，这将导致 ext3_mknod 函数被调用。

ext3_mknod 函数的主要作用是生成一个新的 inode（用来在内核空间表示 demodev 设备文件节点，demodev 设备节点文件与新生成的 inode 之间的关联在图 2-7 中由标号 5 的线段表示）。在 ext3_mknod 中会调用一个和设备驱动程序关系密切的 init_special_inode 函数，其定义如下：

<fs/inode.c>

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
        inode->i_mode = mode;
        if (S_ISCHR(mode)) {
                inode->i_fop = &def_chr_fops;
                inode->i_rdev = rdev;
        } else if (S_ISBLK(mode)) {
                inode->i_fop = &def_blk_fops;
                inode->i_rdev = rdev;
        } else if (S_ISFIFO(mode))
                inode->i_fop = &def_fifo_fops;
        else if (S_ISSOCK(mode))
                inode->i_fop = &bad_sock_fops;
        else
                printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
                                " inode %s:%lu\n", mode, inode->i_sb->s_id,
                                inode->i_ino);
}
```

这个函数最主要的功能便是为新生成的 inode 初始化其中的 i_fop 和 i_rdev 成员。设备文件节点 inode 中的 i_rdev 成员用来表示该 inode 所对应设备的设备号，通过参数 rdev 为其赋值。设备号在由 sys_mknod 发起的整个内核调用链中进行传递，最早来自于用户空间的 mknod 命令行参数。

i_fop 成员的初始化根据是字符设备还是块设备而有不同的赋值。对于字符设备，fop 指向 def_chr_fops，后者主要定义了一个 open 操作：

<fs/char_dev.c>

```
const struct file_operations def_chr_fops = {
        .open = chrdev_open,
        …
};
```

相对于字符设备，块设备的 def_blk_fops 的定义则要有点复杂：

<fs/block_dev.c>

```
const struct file_operations def_blk_fops = {
        .open           = blkdev_open,
        .release        = blkdev_close,
        .llseek         = block_llseek,
        .read           = do_sync_read,
        .write          = do_sync_write,
        .aio_read       = generic_file_aio_read,
        .aio_write      = blkdev_aio_write,
        .mmap           = generic_file_mmap,
        .fsync          = blkdev_fsync,
        .unlocked_ioctl     = block_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl       = compat_blkdev_ioctl,
#endif
        .splice_read= generic_file_splice_read,
        .splice_write       = generic_file_splice_write,
};
```

关于块设备，将在本书第 11 章“块设备驱动程序”中详细讨论，这里依然把考察的重点放在字符设备上。字符设备 inode 中的 i_fop 指向 def_chr_fops。至此，设备节点的所有相关铺垫工作都已经结束，接下来可以看看打开一个设备文件到底意味着什么。

2.7 字符设备文件的打开操作

作为例子，这里假定前面对应于/dev/demodev 设备节点的驱动程序在自己的代码里实现了如下的 struct file_operations 对象 fops：

```
static struct file_operations fops = {
        .open = demoopen,
        .read = demoread,
        .write = demowrite,
        .ioctl = demoioctl,
};
```

用户空间 open 函数的原型为：

```
int open(const char *filename, int flags, mode_t mode);
```

这个函数如果成功，将返回一个文件描述符，否则返回-1。函数的第一个参数 filename 表示要打开的文件名，第二个参数 flags 用于指定文件的打开或者创建模式，本书在后续“字符设备的高级操作”一章中会讨论其中一些常见取值对驱动程序的影响，最后一个参数

mode 只在创建一个新文件时才使用，用于指定新建文件的访问权限，比如可读、可写及可执行等权限。

位于内核空间的驱动程序中 open 函数的原型为：

```
<include/linux/fs.h>
struct file_operations {
        …
        int (*open) (struct inode *, struct file *);
        …
};
```

两者相比差异很大。接下来我们将描述从用户态的 open 是如何一步一步调用到驱动程序提供的 open 函数（在我们的例子中，它的具体实现是 demoopen）的。如同设备文件节点的生成一样，透彻了解这里的每一个步骤也需要掌握全面的 Linux 下文件系统的技术细节。从设备驱动程序员的角度，我们依然将重点放在两者如何建立联系的关键点上。

用户程序调用 open 函数返回的文件描述符，本文用 fd 表示，这是个 int 型的变量，会被用户程序后续的 read、write 和 ioctl 等函数所使用。同时可以看到，在驱动程序中的 demodev_read、demodev_write 和 demodev_ioctl 等函数其第一个参数都是 struct file *filp。显然内核需要在打开设备文件时为 fd 与 filp 建立某种联系，其次是为 filp 与驱动程序中的 fops 建立关联。

用户空间程序调用 open 函数，将发起一个系统调用，通过 sys_open 函数进入内核空间，其中一系列关键的函数调用关系如图 2-8 所示：

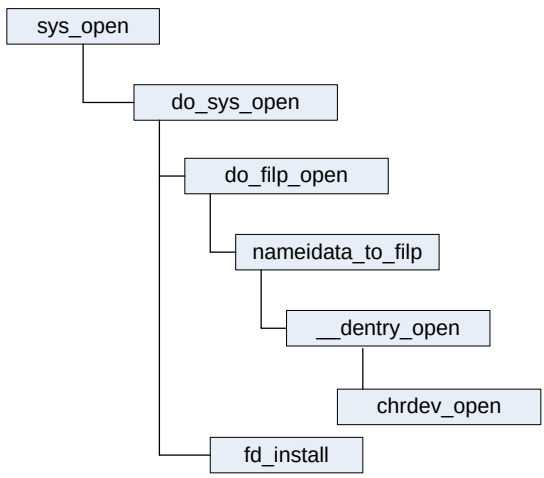

图 2-8　sys_open 到 chrdev_open 调用流程

do_sys_open函数首先通过get_unused_fd_flags为本次的open操作分配一个未使用过的文件

描述符fd[7]：

<fs/open.c>

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
        …
        fd = get_unused_fd_flags(flags);
        …
}
```

get_unused_fd_flags 实际上是封装了 alloc_fd 的一个宏，真正分配 fd 的操作发生在 alloc_fd 函数中，后者会涉及大量文件系统方面的细节，这不是本书的主题。读者这里只需知道 alloc_fd 将会为本次的 open 操作分配一个新的 fd。

do_sys_open 随后调用 do_filp_open 函数，后者会首先查找"/dev/demodev"设备文件所对应的 inode。在 Linux 文件系统中，每个文件都有一个 inode 与之对应。从文件名查找对应的 inode 这一过程，同样会涉及大量文件系统方面的细节。

do_filp_open 在成功查找到"/dev/demodev"设备文件对应的 inode 之后，接着会调用函数 get_empty_filp，后者会为每个打开的文件分配一个新的 struct file 类型的内存空间（本书将把指向该结构体对象的内存指针简写为 filp）：

<fs/namei.c>

```
struct file *do_filp_open(int dfd, const char *pathname,
                const struct open_flags *op, int flags)
{
        struct nameidata nd;
        struct file *filp;

        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
        …
        return filp;
}
```

内核用 struct file 对象来描述进程打开的每一个文件的视图，即使是打开同一文件，内核也会为之生成一个新的 struct file 对象，用来表示当前操作的文件的相关信息，其定义为：

<include/linux/fs.h>

```
struct file {
```

[7] 为了跟踪对文件的读写等操作，内核对于每一次打开的文件都会分配一个文件描述符 fd 和一个 struct file 类型的实例 filp，这个二元组(fd, filp)会被后续的 read、write 等操作使用以向内核记录本次读写操作的信息。从这个角度而言，fd 实际上相当于一个文件可能出现的多种视图的一个索引。作为一种系统资源，一个进程可以分配多少个 fd 决定了一个进程可以 open 多少个文件。

```
        union {
                struct list_head    fu_list;
                struct rcu_head     fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry   f_path.dentry
#define f_vfsmnt   f_path.mnt
        const struct file_operations    *f_op;
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head    f_ep_links;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
};
```

这个结构中与设备驱动程序关系最密切的是 f_op、f_flags、f_count 和 private_data 成员。f_op 指针的类型是 struct file_operations，恰好我们的字符设备驱动程序中也需要实现一个该类型的对象，马上我们将看到这两者之间是如何建立联系的。f_flags 用于记录当前文件被 open 时所指定的打开模式，这个成员将会影响后续的 read/write 等函数的行为模式。成员 f_count 用于对 struct file 对象的使用计数，当 close 一个文件时，只有 struct file 对象中 f_count 成员为 0 才真正执行关闭操作。private_data 常被用来记录设备驱动程序自身定义的数据，因为 filp 指针会在驱动程序实现的 file_operations 对象其他成员函数之间传递，所以可以通过 filp 中的 private_data 成员在某一个特定文件视图的基础上共享数据。

进程为文件操作维护一个文件描述符表（current->files->fdt），正如在本节开始部分看到的那样，对设备文件的打开，最终会得到一个文件描述符 fd，然后用该描述符 fd 作为进程维护的文件描述符表（指向 struct file *类型数组）的索引值，将之前新分配的 struct file 空间地址赋值给它：

```
current->files->fdt->pfd[fd] = filp;
```

这样，用户空间程序在后续的 read、write、ioctl 等函数调用中利用 fd 就可以找到对应的 filp，如图 2-9 所示：

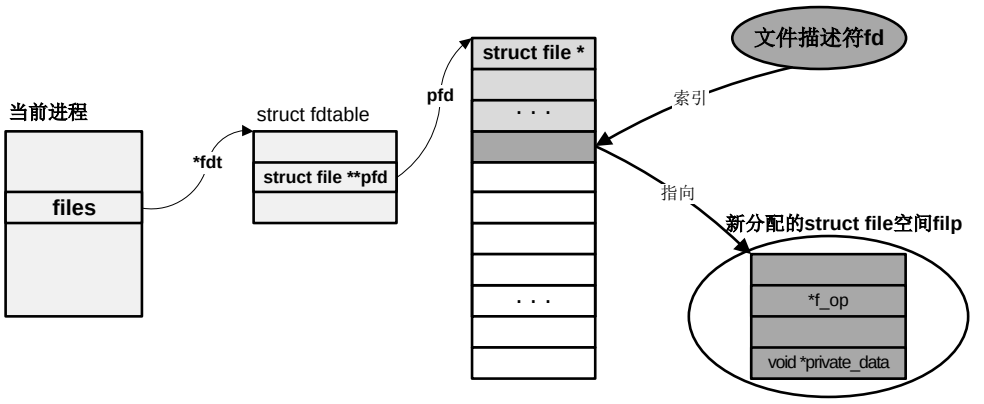

图 2-9　fd 与 filp 的关联

在 do_sys_open 的后半部分，会调用__dentry_open 函数将"/dev/demodev"对应节点的 inode 中的 i_fop 赋值给 filp->f_op，然后调用 i_fop 中的 open 函数：

<fs/open.c>

```
static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
                                  struct file *f,
                                  int (*open)(struct inode *, struct file *),
                                  const struct cred *cred)
{
        struct inode *inode;
        …
        f->f_op = fops_get(inode->i_fop);
        …
        if (!open && f->f_op)
                open = f->f_op->open;
        if (open) {
                error = open(inode, f);
                …
        }
        …
}
```

__dentry_open 函数当初在 nameidata_to_filp 中被调用时，第四个实参是 NULL，所以在__dentry_open 中，open = f->f_op->open。在上节设备文件节点的生成中，我们知道 inode->i_fop = &def_chr_fops，这样 filp->f_op = &def_chr_fops。接下来会利用 filp 中的这个新的 f_op 作调用：filp->f_op->open(inode, filp)，于是 chrdev_open 函数将被调用到。该函数非常重要，为了突出其主线，下面先将它改写成以下简单几行：

<fs/char_dev.c>

```
static int chrdev_open(struct inode *inode, struct file *filp)
{
        int ret = 0, idx;

        struct kobject *kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
        struct cdev *new = container_of(kobj, struct cdev, kobj);
        inode->i_cdev = new;
        list_add(&inode->i_devices, &new->list);
        filp->f_op = new->ops;
        if (filp->f_op->open) {
                ret = filp->f_op->open(inode,filp);
        }
        return ret;
}
```

函数首先通过 kobj_lookup 在 cdev_map 中用 inode->i_rdev 来查找设备号所对应的设备 new，这里展示了设备号的作用。成功查找到设备后，通过 filp->f_op = new->ops 这行代码将设备对象 new 中的 ops 指针（前面曾讨论过，驱动程序通过调用 cdev_init 将其实现的 file_operations 对象的指针赋值给设备对象 cdev 的 ops 成员）赋值给 filp 对象中的 f_op 成员，此处展示了如何将驱动程序中实现的 struct file_operations 与 filp 关联起来，从此图 2-9 中的 filp->f_op 将指向驱动程序中实现的 struct file_operations 对象。

接下来函数会检查驱动程序中是否实现了 open 函数（if (filp->f_op->open)），如果实现了，就调用设备驱动程序中实现的 open 函数。打开一个字符设备节点的大体流程如图 2-10 所示：

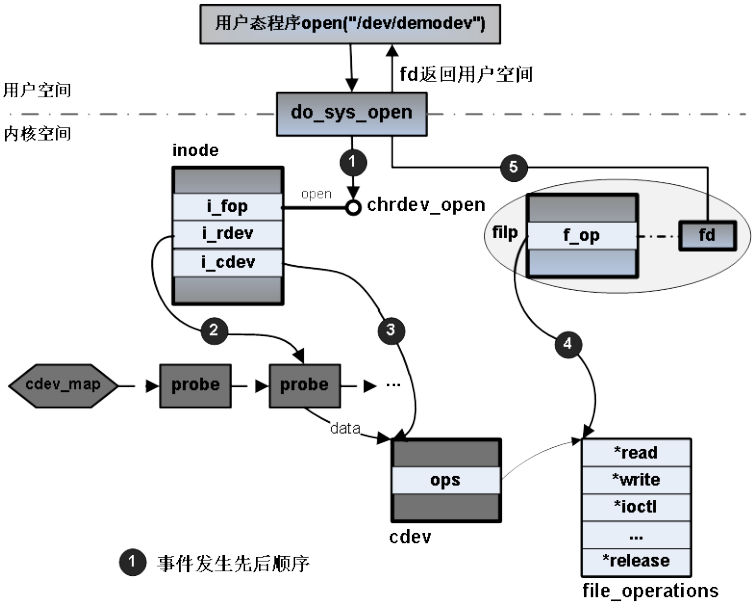

图 2-10　开一个字符设备节点的功能流程

图中，当应用程序打开一个设备文件时，将通过系统调用 sys_open 进入内核空间。在内核空间将主要由 do_sys_open 函数负责发起整个设备文件打开操作，它首先要获得该设备文件所对应的 inode，然后调用其中的 i_fop 函数，对字符设备节点的 inode 而言，i_fop 函数就是 chrdev_open（图中标号 1 的线段），后者通过 inode 中的 i_rdev 成员在 cdev_map 中查找该设备文件所对应的设备对象 cdev（图中标号 2 的线段），在成功找到了该设备对象之后，将 inode 的 i_cdev 成员指向该字符设备对象（图中标号 3 的线段），这样下次再对该设备文件节点进行打开操作时，就可以直接通过 i_cdev 成员得到设备节点所对应的字符设备对象，而无须再通过 cdev_map 进行查找。内核在每次打开一个设备文件时，都会产生一个整型的文件描述符 fd 和一个新的 struct file 对象 filp 来跟踪对该文件的这一次操作，在打开设备文件时，内核会将 filp 和 fd 关联起来，同时会将 cdev 中的 ops 赋值给 filp->f_op（图中标号 4 的线段）。最后，sys_open 系统调用将设备文件描述符 fd 返回到用户空间，如此在用户空间对后续的文件操作 read、write 和 ioctl 等函数的调用，将会通过该 fd 获得文件所对应的 filp，根据 filp 中的 f_op 就可以调用到该文件所对应的设备驱动上实现的函数。

通过以上过程，我们看到了设备号在其中的重要作用。当设备驱动程序通过 cdev_add 把一个字符设备对象加入到系统时，需要一个设备号来标记该对象在 cdev_map 中的位置信息。当我们在用户空间通过 mknod 来生成一个设备文件节点时，也需要在命令行中提供设备号的信息，内核会将该设备号信息记录到设备文件节点所对应 inode 的 i_rdev 成员中。当我们的应用程序打开一个设备文件时，系统将会根据设备文件对应的 inode->i_rdev 信息在 cdev_map 中寻找设备。所以在这个过程中务必要保证设备文件节点的 inode->i_rdev 数据和设备驱动程序使用的设备号完全一致，否则就会发生严重问题。对应到现实世界的操作，那就是在用 mknod 生成设备节点时所提供的设备号信息一定要与设备驱动程序中分配使用的设备号一致。

在上述 open 一个设备文件的基础上，接下来不妨看看它的相反操作 close。有了前面对 open 操作技术细节讨论所打下的良好基础，现在理解起 close 并不困难，在此读者也正好可以看看用户空间 open 函数返回的文件描述符 fd 如何被 close 等函数使用。

用户空间 close 函数的原型为：

```
int close(unsigned int fd);
```

针对 close 的系统调用函数为 sys_close，这里将其核心代码重新整理如下：

<fs/open.c>

```
int sys_close(unsigned int fd)
{
        struct file * filp;
        struct files_struct *files = current->files;
        struct fdtable *fdt;
        int retval;
```

```
        …
        fdt = files_fdtable(files);
        …
        filp = fdt->fd[fd];
        …
        retval = filp_close(filp, files);
        …
        return retval;
}
```

从 fd 得到 filp 这段代码，请读者参考本章 2-9。接下来调用 filp_close 函数，close 函数的大部分秘密都隐藏在其中，有必要看看其主要代码片段：

<fs/open.c>

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval = 0;

        if (!file_count(filp)) {
                printk(KERN_ERR "VFS: Close: file count is 0\n");
                return 0;
        }

        if (filp->f_op && filp->f_op->flush)
                retval = filp->f_op->flush(filp, id);
                …
        fput(filp);
        return retval;
}
```

if (!file_count(filp))用来判断 filp 中的 f_count 成员是否为 0，如果针对同一个设备文件 close 的次数多于 open 次数，就会出现这种情况，此时函数直接返回 0，因为实质性的工作都被前面的 close 做完了。接下来的情况有点意思，如果设备驱动程序定义了 flush 函数，那么在 release 函数被调用前，会首先调用 flush，这是为了确保在把文件关闭前缓存在系统中的数据被真正写回到硬件中。字符设备很少会出现这种情况，因为这种设备的慢速 I/O 特性决定了它无须使用这种缓冲机制来提升系统性能，但是块设备就不一样了，比如 SCSI 硬盘会和系统进行大量数据的传输，为此内核为块设备驱动程序设计了高速缓存机制，这种情况下为了保证文件数据的完整性，必须在文件关闭前将高速缓存中的数据写回到磁盘中。不过这是后话了，块设备驱动程序的这种机制将在“块设备驱动程序”一章中讨论。

函数的最后调用 fput，貌似很简单的一个函数，其实内涵却很丰富：

<fs/file_table.c>

```
void fput(struct file *file)
{
```

```
        if (atomic_long_dec_and_test(&file->f_count))
            __fput(file);
}
```

函数中的那个 atomic_long_dec_and_test 是个体系架构相关的原子测试操作，就是说，如果 file->f_count 的值为 1，那么它将返回 true，这意味着可以真正关闭当前的文件了，所以__fput 将被调用，并最终完成文件关闭的任务，它的一些关键调用节点如下所示：

<fs/file_table.c>

```
static void __fput(struct file *file)
{
        …
        if (unlikely(file->f_flags & FASYNC)) {
            if (file->f_op && file->f_op->fasync)
                file->f_op->fasync(-1, file, 0);
        }
        if (file->f_op && file->f_op->release)
            file->f_op->release(inode, file);
        …
        fops_put(file->f_op);
        file_free(file);
}
```

注意上面的 FASYNC 标志位，在本书后面的章节会讨论到 file_operations 中的一些常用的函数实现。然后函数调用到了设备驱动程序中提供的 release 函数，接下来是一些系统资源的释放。可见，对于应用程序的一个 close 调用，并非必然对应着 release 函数的调用，只有在当前文件的所有副本都关闭之后，release 函数才会被调用。

2.8 本章小结

本章描述了字符设备驱动程序内核框架的技术细节。基本上可以看到，字符设备驱动内核框架的展开是按照两条线进行的：一条是设备与系统的关系，一个字符设备对象 cdev 通过 cdev_add 加入到系统中（由 cdev_map 所管理的哈希链表），此时设备号作为哈希索引值；另一条是设备与文件系统的关系，设备通过设备号以设备文件的形式向用户空间宣示其存在。这两条线间的联系通过文件系统接口去打开一个字符设备文件而建立：

- mknod 命令将为字符设备创建一个设备节点，mknod 的系统调用将会为此设备节点产生一个 inode，mknod 命令行中给出的设备号将被记录到 inode->i_rdev 中，同时 inode 的 i_fop 会将 open 成员指向 chrdev_open 函数。
- 当用户空间 open 一个设备文件时，open 函数通过系统进入内核空间。在内核空间，首先找到该设备节点所对应的 inode，然后调用 inode->i_fop->open()，我们知道这将导致

chrdev_open 函数被调用。同时，open 的系统调用还将产生一个(fd, filp)二元组来标识本次的文件打开操作，这个二元组是一一对应的关系。

- chrdev_open 通过 inode->i_rdev 在 cdev_map 中查找 inode 对应的字符设备，cdev_map 中记录着所有通过 cdev_add 加入系统的字符设备。
- 当在 cdev_map 中成功查找到该字符设备时，chrdev_open 将 inode->i_cdev 指向找到的字符设备对象，同时将 cdev->ops 赋值给 filp->f_op。
- 字符设备驱动程序负责实现 struct file_operations 对象，在字符设备对象初始化时 cdev_init 函数负责将字符设备对象 cdev->ops 指向该 file_operations 对象。
- 用户空间对字符设备的后续操作，比如 read、write 和 ioctl 等，将通过 open 函数返回的 fd 找到对应的 filp，然后调用 filp->f_op 中实现的各类字符设备操作函数。

以上就是内核为字符设备驱动程序设计的大体框架，从中可以看到设备号在沟通用户空间的设备文件与内核中的设备对象之间所起的重要作用。

另外，对于字符设备驱动程序本身而言，核心的工作是实现 struct file_operations 对象中的各类函数，file_operations 结构中虽然定义了众多的函数指针，但是现实中设备驱动程序并不需要为它的每一个函数指针都提供相应的实现。本书后面的“字符设备的高级操作”一章会详细讨论其中一些重要函数的作用和实现原理。

第 8 章

时间管理

设备驱动程序需要对时间进行操作，典型的可以分为两大类：延时与定时。前者是在两个连续的动作 A 与 B 之间插入一段时间空白，也即在动作 A 执行后需要等待若干时间才能执行动作 B，至于在这段时间空白内，当前处理器也许是进入忙等待状态，也许是切换到一个新进程上。后者是在一个指定的时间点到达后执行某些动作，轮询是其最典型的应用。

本章将讨论这两类时间上的操作的技术细节，设备驱动程序员在掌握了这些幕后的技术之后可以更好地理解设备驱动是如何对时间进行掌控的，当程序中需要对时间进行管理时选择最合适的解决方案。

8.1 jiffies

内核源代码中几乎到处充斥着 jiffies 这样的变量，作为设备驱动程序员对此想必也一定不会陌生，在某些书中它被形象地称为“时钟滴答”。内核源码中针对 32 位和 64 位系统分别定义了 jiffies 和 jiffies_64：

```
<include/linux/jiffies.h>
#define __jiffy_data    __attribute__((section(".data")))
extern u64 __jiffy_data jiffies_64;
extern unsigned long volatile __jiffy_data jiffies;
```

其中的__jiffy_data 表明这两个变量将出现在内核最终映像的".data"区中，另外在头文件中在一个变量的声明前使用了“extern”关键字，提示了这个变量可能定义在某个别的文件中，事实上它们出现在内核的链接脚本文件 vmlinux.lds 中。除了数据位宽不一样外，上述两个变量在原理上是一样的。为了叙述的方便，下面只提 jiffies，本节稍后会给出两者在操作上的一些细微的区别。

通常jiffies在Linux系统启动引导阶段被初始化为 0，当系统完成了对时钟中断[1]的初始化之

[1] 基于 x86 体系架构的 Linux 系统中产生时钟中断的硬件典型的有可编程中断计数器 PIT（Programmable Interrupt Timer）8253 和高级可编程中断控制器 APIC（Advanced Programmable Interrupt Controller），后者的分辨率及稳定性都要比前者好得多，用来实现高分辨率的时间源。

后，在每个时钟中断（“时钟滴答”）处理例程中该值都会被加 1，如图 8-1 所示：

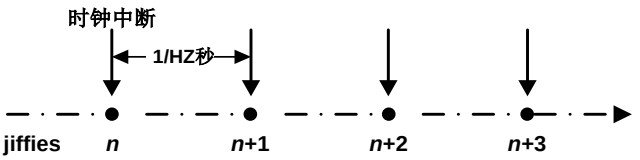

图 8-1　每隔 1/HZ 秒 jiffies 的值增 1

因此该值储存了系统自最近一次启动以来的时钟滴答数。在形式上，它跟我们日常所熟悉的时分秒这样的时间概念有很大的不同，不过对于设备驱动程序而言，出于时间管理的需要，使用 jiffies 就已经足够，因为它甚少用这种时间形式与应用程序进行沟通。除了时钟中断处理例程中对 jiffies 进行更新外，其他任何模块（驱动程序当然也不例外）都只是读取该值以获得当前时钟计数。

在实际使用 jiffies 时，还需要了解 Linux 内核中另一个与时钟中断息息相关的宏 HZ，它用来表示系统中时钟中断发生的频率：

```
<include/asm-generic/param.h>
#ifdef __KERNEL__
# define HZ         CONFIG_HZ       /* Internal kernel timer frequency */
# define USER_HZ       100          /* some user interfaces are */
# define CLOCKS_PER_SEC (USER_HZ)          /* in "ticks" like times() */
#endif
```

从上述的定义可以看出，内核提供了在配置阶段通过 CONFIG_HZ 修改 HZ 数值的可能性，但绝大多数情况下都没有必要修改它，使用内核默认的值 1000 就足够了。事实上 CONFIG_HZ 并未出现在内核的配置菜单选项中，而是就在内核源码根目录下的.config 文件中。HZ 值为 1000 意味着系统 1 秒内要发生 1000 次时钟中断，也就是说每隔 1 毫秒，jiffies 的值就会增加 1。所以，如果驱动程序使用 jiffies 来对时间进行度量的话，其精度只能局限在毫秒级别上，更高精度的时间管理单纯使用 jiffies 无法满足要求。

相对于jiffies而言，jiffies_64 是个 64 位的变量（即便是在 32 位的体系架构上也是一样，此时它是一个unsigned long long型的变量，通过组合两个unsigned long型变量得到），在 64 位平台上，它们其实是同一个变量，而在 32 位平台上，jiffies和jiffies_64 的低 32 位是重合的。之所以引入jiffies_64，是考虑到了 32 位变量jiffies的溢出问题，在HZ=1000 的情况下，大约 50 天就会导致jiffies溢出。对于驱动程序中的时间度量而言，这并不是个大问题（但是在作时间比较的时候仍然需要小心处理），不过现实中显然要考虑某些系统[2]有需要知道自系统最近一次运行以来真正的时钟滴答数的需求，因此Linux内核中同时引入了jiffies_64 来

[2] 比如对某些特殊的服务器而言，它们启动一次运行的时间也许足够久，以至于久到发生了 jiffies 的溢出。

记录系统的时钟计数。为了保证jiffies和jiffies_64 两个变量无论在 32 位还是 64 位平台上在记录时钟滴答数时的一致性，内核通过链接脚本做了些手脚，有兴趣的读者可以阅读一下链接内核时所使用的链接脚本vmlinux.lds，可发现类似下面的内容：

```
#ifdef CONFIG_X86_32
OUTPUT_ARCH(i386)
jiffies = jiffies_64;
#else
OUTPUT_ARCH(i386:x86-64)
jiffies_64 = jiffies;
#endif
```

如果要在 32 位系统上读取 jiffies_64 的值，必须使用 get_jiffies_64 函数，因为在直接读取 jiffies_64 的高 32 位或者低 32 位时，对应的低 32 位或者高 32 位可能已经发生更新。get_jiffies_64 函数使用顺序锁的方式来保证对 jiffies_64 变量读取操作的原子性：

<kernel/time.c>

```
#if (BITS_PER_LONG < 64)
u64 get_jiffies_64(void)
{
        unsigned long seq;
        u64 ret;

        do {
                seq = read_seqbegin(&xtime_lock);
                ret = jiffies_64;
        } while (read_seqretry(&xtime_lock, seq));
        return ret;
}
#endif
```

从设备驱动程序的角度出发，仅仅使用 jiffies 变量就已足够满足所有基于 jiffies 的时间度量任务，所以本章接下来的部分将主要以 jiffies 为讨论对象。

由于 jiffies 在内核源码中作为一个全局性变量被导出，所以如果驱动程序中需要读取当前的 jiffies 值，只需在源码中包含头文件 linux/jiffies.h 即可，比如下面的代码片段：

```
#include <linux/jiffies.h>
unsigned long    j , timestamp_1, timestamp_2;

j = jiffies; //读取当前的时钟计数值
timestamp_1 = jiffies + 2 * HZ; //timestamp_1 为未来的 2 秒
timestamp_2 = jiffies + 3 * HZ /1000; //timestamp_2 为未来的 3 毫秒
```

Linux 设备驱动程序中使用 jiffies 的几个常用的场景分别有时间比较、时间转换以及设置定时器（timer）时对未来时间的设定，本节先讨论前两个话题，后一个话题将在稍后的系统

定时器一节中予以讨论。

8.1.1　时间比较

设备驱动程序有时候需要对程序执行过程中的两个时间点进行比较，以确定时间点之间的先后次序。因为 jiffies 在每次的时钟中断处理例程中都会被更新，因此可以通过两次时间点所对应的 jiffies 值来进行判断。如果没有前面提到的 jiffies 值溢出的问题，那么这种判断的逻辑非常简单，但是因为溢出的可能性是存在的，所以程序应该谨慎处理。好在 Linux 内核为此提供了一组用以判断时间点先后顺序的宏，通过特定的技巧非常安全地处理了 jiffies 值溢出的情况，程序中可以放心使用。这组宏为：

time_after(a, b)

如果时间点 a 在时间点 b 之后，该宏返回 true。

time_before(a, b)

如果时间点 a 在时间点 b 之前，该宏返回 true。

time_after_eq(a, b)

该宏类似于 time_after，但是在 a 和 b 两个时间点相等时，该宏也返回 true。

time_before_eq(a, b)

该宏类似于 time_before，但是在 a 和 b 两个时间点相等时，该宏也返回 true。

time_in_range(a, b, c)

该宏用来检查时间点 a 是否包含在时间间隔[b, c]内，因为检查包含边界，所以当 a 等于 b 或者 c 时，该宏也会返回 true。

在使用以上宏时，参数 a 和 b 都应该是 unsigned long 型变量。如果是针对 jiffies_64 类型来作这种时间顺序的判断，那么除了 time_in_range 宏之外，只需在宏的后面加上后缀 64 即可，例如 time_after64，不过设备驱动程序中使用 64 位的情形极其罕见。

因此，设备驱动程序中不应该直接使用 jiffies 的值来作时间点先后顺序的比较，而应该使用内核提供的上述宏来完成。下面给出一个具体的例子，假设驱动程序的某个函数 demo_function 需要调用比如 do_time_task 函数来完成一个任务，但是对任务完成的时间有特定的要求（比如要在 2 毫秒之内完成），如果在规定的时间内没有完成，就需要调用 task_timeout 函数来处理，否则 demo_function 函数就算顺利完成。如果使用下面的代码将是不安全的：

```
int demo_function()
```

```
{
    unsigned long timeout = jiffies + 2 * HZ / 1000;  //设定超时的时间为 2 毫秒
    do_time_task();                  //调用 do_time_task 来完成某一任务
    if (timeout < jiffies)           //根据当前最新的 jiffies 值来判断是否超时
        return task_timeout();       //do_time_task()完成时间超过了 2 毫秒，调用超时处理函数
    return 0;
}
```

因为存在 jiffies 溢出环绕的可能性，所以上述的 if 语句中 timeout < jiffies 条件在超时的情况下也可能返回 false（比如计算出的 timeout 值本身已经接近 jiffies 溢出环绕的临界点时）。正确的代码应该是：

```
int demo_function()
{
    unsigned long timeout = jiffies + 2 * HZ / 1000;  //设定超时的时间为 2 毫秒
    do_time_task();                  //调用 do_time_task 来完成某一任务
    if (time_after(jiffies, timeout)) //根据当前最新的 jiffies 值来判断是否超时
        return task_timeout();       //do_time_task()完成时间超过了 2 毫秒，调用超时处理函数
    return 0;
}
```

8.1.2 时间转换

有时候，设备驱动程序可能需要将用 jiffies 表达的时间间隔转化成毫秒 ms 或者是微秒 us 的形式，这种情况大多出现在需要将时钟滴答这种形式转化成人类易于理解的 ms 或者是 us 这样的时间形式下，比如为了在驱动程序中打印出一次 DMA 传输所花费的时间，在 DMA 传输开始前记录 start_jiffies = jiffies，然后进行 DMA 传输，在 DMA 传输结束后记录下 end_jiffies = jiffies，这样本次的 DMA 传输所花费的时间将为 time = jiffies_to_msecs(end_jiffies - start_jiffies)，这里 time 的单位将会是 ms。

Linux 内核源码为此提供了一组相关的转换函数：

```
<include/linux/jiffies.h>
unsigned int jiffies_to_msecs(const unsigned long j);
unsigned int jiffies_to_usecs(const unsigned long j);
unsigned long msecs_to_jiffies(const unsigned int m);
unsigned long usecs_to_jiffies(const unsigned int u);
```

从这些函数的命名上已经可以很清楚地知道其各自的功能，此处不再赘述。时间转换的另一种情形发生在用户态程序和设备驱动程序的交互上，应用程序员更多地使用秒以及毫秒等时间形式。此种情形下，内核定义了 struct timeval 和 struct timespec 两种数据结构：

```
<include/linux/time.h>
struct timespec {
```

```
    __kernel_time_t   tv_sec;                 /* seconds */
    long              tv_nsec;                      /* nanoseconds */
};

struct timeval {
    __kernel_time_t           tv_sec;         /* seconds */
    __kernel_suseconds_t      tv_usec;        /* microseconds */
};
```

可见 timespec 用秒和纳秒来描述时间，而 timeval 则采用秒和毫秒的形式。内核同样提供了 jiffies 变量和这两个数据结构的实例间相互转换的函数：

```
<include/linux/jiffies.h>
unsigned long timespec_to_jiffies(const struct timespec *value);
void jiffies_to_timespec(const unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(const struct timeval *value);
void jiffies_to_timeval(const unsigned long jiffies, struct timeval *value);
```

这些函数的用法也是很明显的。

到目前为止，已经讨论了基于 jiffies 的时间度量的方法，鉴于 jiffies 自身精度的局限性，如果需要使用更高精度的时间度量方法，也许要借助于某些体系架构特定的计时寄存器，例如很有名的时间戳计数器 TSC（Time Stamp Counter），但是使用这些寄存器的程序代码将失去在不同平台之间的可移植性。因为用 jiffies 来衡量一个时间间隔在绝大多数的情况下已经足以满足需要，所以本书将不再讨论其他的时间度量方法。

8.2　延时操作

设备驱动程序中延时操作的常见应用场景是，当 CPU 通过外部设备的寄存器对设备发出指令时，外设执行相应的动作，在该动作完成之后通过更新比如状态寄存器来告之本次操作的执行结果，CPU 需要读取该状态寄存器的值来获得设备的执行结果。因为 CPU 的速度很快，而外部设备可能需要一定的时间才能完成本次操作，如果 CPU 在写完寄存器后直接读取设备的状态寄存器，那么很有可能得到错误的结果（设备尚未完成指定的操作，因而还没有更新状态寄存器），所以 CPU 在对外设发出操作指令后，需要延时一段时间以等待设备操作的完成。这种情形在设备驱动程序中一个简单而典型的代码执行序列应该是：

```
001     write_command_reg(…);      //CPU 写外设的寄存器以发起一个操作指令
002     delay(…);  //延时操作，等待设备操作完成。它的实现机制是本节要讨论的主题
003     read_status_reg(…);    //CPU 读外设的寄存器以获得设备执行结果
```

上述序列的第 2 行执行的就是一个延时操作，因为它的存在使得第 3 行的指令向后推迟了

一段时间才被执行到，这使得外部设备有足够的时间[3]来完成当前的操作并将执行结果更新到状态寄存器中。下面讨论设备驱动程序如何实现delay函数以实现对应的时间延迟。

从实现延时精度的角度出发，可以将延迟函数分成两大类：一类是基于时钟滴答 jiffies 实现的延迟，因为这类延迟的时间粒度一般在毫秒 ms 级别，所以被称为“长延时”；另一类的延时精度已经超越了时钟滴答的边界，比如微秒 us 和纳秒 ns 级的延迟，显然单纯依靠 jiffies 已经无法满足要求，此时需要有另外的实现机制，也就是所谓的“短延时”函数。下面先从长延时开始讨论。

8.2.1 长延时

基于时钟滴答 jiffies 的长延时函数有几种实现方法，主要围绕在延迟实现过程中是否让出处理器来展开，在具体的实现上分为“忙等待”和“让出处理器”两大类。

❍ 忙等待

用忙等待来实现延时是最简单的。虽然因为忙等待的关系其执行效率饱受诟病，但是不得不承认在设备驱动程序早期的开发调试阶段这是一种很简便的手法来判断设备是否能像预期的那样工作。

最简单的忙等待实现是用 while 或者其他的什么循环，比如：

```
unsigned long t = 0xFFFFFF;
while(t--);
```

这是种相当粗糙的实现延时操作的原始方法，现实当中的程序员也许迫切想获得某种延迟效果而又不想稍微用点脑力去寻求更好的解决方案时，上面的代码就会出现。因为它的缺点是如此明显，所以它显然不应该出现在最终发布出去的软件版本中。

其实完全可以利用 jiffies 来实现一种比较理想的忙等待延时策略，而且相对前面的那种忙等待，这种方法对延时的长短也有很好的控制，比如为了实现 1 s 的等待，可以使用下面的方法：

```
unsigned long j = jiffies + HZ;
while(time_before(jiffies, j))
        cpu_relax();
```

上面的这段延时代码看起来已经很像那么回事了，除了利用time_before和jiffies来实现 1 s 延时的控制，还在while循环体中加入了对cpu_relax函数的调用。显然cpu_relax的实现不会

[3] 设备驱动程序员也许要根据外部设备的 data sheet 等操作规范文档来评估一个操作需要花费多少时间，在获得这个数据后才可能精确地设定后续 delay 操作需要延迟的时间宽度。

导致当前代码让出处理器，否则就不能称为忙等待了[4]。cpu_relax是个平台相关的函数，在x86 架构上，其核心指令是NOP，也就是空指令：

<arch/x86/include/asm/processor.h>

```
/* REP NOP (PAUSE) is a good thing to insert into busy-wait loops. */
static inline void rep_nop(void)
{
        asm volatile("rep; nop" ::: "memory");
}

static inline void cpu_relax(void)
{
        rep_nop();
}
```

其他平台上，比如 ARM，cpu_relax 的实现可能是一个内存屏障类的函数调用：

<arch/arm/include/asm/processor.h>

```
#if __LINUX_ARM_ARCH__ == 6
#define cpu_relax()                     smp_mb()
#else
#define cpu_relax()                     barrier()
#endif
```

无论如何，这段代码实现背后的原理还是非常直白的，其缺陷也同样很直白：

（1）基本上在这 1 s 的延迟时间段内进入忙等待的 CPU 做不了任何事情，对于当前的高速 CPU 而言，这种资源的浪费是很可观的。而且对于单 CPU 系统来说，如果不幸在进入这段忙等待的代码前关闭了 CPU 的中断，那么 jiffies 的值将不会被更新，while 循环的条件将一直满足，这种情况下除了重启系统似乎没有更好的解决方法。

（2）对于一个可抢占式的系统而言，上面的忙等待代码有可能在某次中断的过程中被抢占，比如，进程 A 在等待外设的数据时进入了睡眠状态，此时调度器调度进程 B 到当前的 CPU 上运行，假设进程 B 恰好执行的就是上述的忙等待代码，如果在延迟时间段尚未结束时，进程 A 因外设数据的就绪被唤醒并且调度优先级高于进程 B，那么 A 将被重新调度到当前处理器上运行，如果 A 再次被调度后连续运行的时间超过了 1 s，那么当 B 被再次调度运行时，while 中的条件显然已经不再满足，此时延时的目的虽然是达到了，但是延迟的时间并不是当初设定的 1 s，而可能是比如 1.5 s 等。

---

[4] 至于这段忙等待代码在执行过程中是否会让出当前处理器，要看内核是否配置启用了可抢占性。在 1 s 的时间里，这个 while 循环体在执行过程中会出现大量的时钟中断，如果内核不可抢占，那么运行在内核态的这个忙等待代码不会产生新的调度点，因此所占有的处理器不会被强制剥夺。但对于可抢占式内核而言，如果有更高优先级的任务就绪，则它可能会被调度出处理器。

上述的问题 2 对于驱动程序来说并不是什么大的问题，即便是在后面讨论的一些改进型的延迟实现中也同样存在。如果没有 CPU 资源的浪费，那么即便延迟函数造成了预设延迟时间段的延伸，对设备驱动的性能而言也不会有实质性的影响，而对这个问题的根本性解决也许要涉及对调度器的改进，这对于延时精度本来就要求不高的长延迟函数而言，没有充足的理由。而对问题 1 的改进则导致了“让出处理器”解决方案的出现。

让出处理器

在忙等待的实现中，处于忙等待中的代码一直占用处理器会导致系统性能降低，于是一种改进的方案是：当代码进入到 while 循环时，不再调用 cpu_relax()函数而是调用 schedule()调度函数以让出处理器，这样就解决了忙等待一直浪费处理器的弊端。比如下面的代码：

```
unsigned long j = jiffies + HZ;
while(time_before(jiffies, j))
        schedule();
```

这种方法相对于前面的忙等待来说，解决了无端占用 CPU 的问题，但还不是最佳的解决方案，因为主动调用 schedule()函数的进程虽然可以让出处理器，但依然在当前 CPU 的运行队列中。这使得在空闲的系统中无法进入 idle 状态，因为即便 CPU 的运行队列中只有当前一个进程，它也会陷入让出处理器之后马上又被调度运行，然后再让出处理器这样的怪圈中。无法进入 idle 状态对系统的电源管理模块来说不是件好事情，因为有些智能化的电源管理模块可以根据当前 CPU 的负载情况来决定是否改变 CPU 的运行频率，频率的改变与 CPU 供电电压的改变是息息相关的。CPU 的 idle 状态可以被电源管理模块所利用，如果后者发现 CPU 进入了 idle 状态，可以降低 CPU 的核心频率从而降低整机的能耗，这在以 ARM 为主的嵌入式平台上尤为常见。

当然，相对于一直占用 CPU 资源的忙等待，这种方法提升了 CPU 资源的利用率，使得当前进程在延迟等待期间 CPU 可以运行其他的进程。另外要提的一点是，进程在调用 schedule()函数让出处理器后，并不能保证可以很快再次获得处理器，其再次获得处理器的时间间隔取决于当前处理器运行队列中进程的数量，以及这些进程与延迟等待进程的调度优先级。换言之，问题 2 中阐述的预设延迟时间段延伸的问题依然存在。

上面提到的采用 schedule()函数的解决方法之所以还不是最佳的，其根本原因在于调用 schedule()函数的进程依然处于 CPU 的运行队列中。为了解决这个问题，此时应该能想到内核提供的另外一种可供设备驱动程序使用的调度类的基础设施：schedule_timeout。所以，如果一个延迟 1 s 的函数可以用下面的这样一个简单的代码段来实现：

```
delay_1s()
{
```

```
        set_current_state(TASK_UNINTERRUPTIBLE)[5];
        schedule_timeout(jiffies + HZ);
}
```

上面这段代码之所以可以解决直接采用 schedule()函数所带来的负面问题，主要在于在调用 schedule_timeout 之前先调用了 set_current_state 宏将当前进程的状态设置为 TASK_UNINTERRUPTIBLE，这样当随后的 schedule_timeout 函数被调用时，后者的内部实现中调用了 schedule()函数，因为当前进程之前的状态已经被设置为 TASK_UNINTERRUPTIBLE，所以在 schedule 函数中当前进程将会被移出处理器的运行队列，因此也就解决了采用直接调用 schedule 函数那种方案所带来的不利影响。也许有读者会问，在早先的那个直接调用 schedule 函数的方案前调用一下 set_current_state(TASK_UNINTERRUPTIBLE)不也可以把当前进程移出运行队列吗？比如：

```
unsigned long j = jiffies + HZ;
while(time_before(jiffies, j)){
        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule();
}
```

上面的代码实现非常糟糕，已经不只是仅对性能有影响那么简单了，通过在 schedule()函数前使用 set_current_state(TASK_UNINTERRUPTIBLE)，固然可以使当前进程从处理器的运行队列中移开，但它此后将再也没有机会被调度，因为不会再有别的代码去更改其状态使其可以再次进入运行队列，这与使用等待队列和调用 schedule_timeout 完全不同。如果使用了等待队列，那么我们知道等待队列中的睡眠进程有被唤醒一说，关于这点，本书前面的章节已经讨论过。如果使用 schedule_timeout，当前进程从运行队列移走之后将被记录到定时器的数据节点中，当定时器的时间到期后，将会唤醒该进程使其重新进入运行队列。这里不妨通过内核中 schedule_timeout 实现的一些关键代码，来看一看此处蕴藏的秘密：

<kernel/timer.c>

```
signed long __sched schedule_timeout(signed long timeout)
{
        struct timer_list timer;
        unsigned long expire;
        …
        expire = timeout + jiffies;

        setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
        __mod_timer(&timer, expire, false, TIMER_NOT_PINNED);
        schedule();
```

[5] 此处当然可以使用 TASK_INTERRUPTIBLE 等进程状态标志，不过需要小心处理在延迟时间到达之前进程被信号等中断的可能。为了便于叙述，接下来的文字中统一使用 TASK_UNINTERRUPTIBLE 标志。

```
        del_singleshot_timer_sync(&timer);

        /* Remove the timer from the object tracker */
        destroy_timer_on_stack(&timer);
        …
}
```

函数的主要职能是在调用 schedule()函数前实现了一个定时器，关于定时器，本章后续的内容将很快讨论到它，此处只要记住当 expire 指定的时钟滴答到期时，setup_timer_on_stack 函数调用中的第二个参数 process_timeout 会被调用到，同时注意到指向当前进程的 current 指针作为第三个实参传给了 setup_timer_on_stack 函数，这样在 process_timeout 函数被调用时将会获得当前进程的指针，我们很容易猜出 process_timeout 函数要做的事情，当定时器到期时，它被调用以唤醒 current 进程：

<kernel/timer.c>

```
static void process_timeout(unsigned long __data)
{
        wake_up_process((struct task_struct *)__data);
}
```

它是如此简单明了，所以不需要在这上面浪费过多的文字。

现在我们已经看到，使用 schedule_timeout 可以确保在指定的延迟时间到期时进程可以重新获得调度的机会，因为结合了对 set_current_state(TASK_UNINTERRUPTIBLE)的一起使用，使得进程在指定的延时时间段内不会出现在运行队列中，这就很好地解决了单纯调用 schedule 函数所带来的问题。需要提醒一下的是，当定时器到期时，虽然 process_timeout 将进程重新放入了处理器的运行队列，但是它何时被调度依然无法给出精确的时间点（这取决于调度器）。换言之，预定的延迟时间段的延伸在这里同样是个无法回避的问题。如果在调用 schedule_timeout 前没有使用 set_current_state 来将当前进程的状态改为 TASK_UNINTERRUPTIBLE，那么效果等同于直接调用 schedule 函数，除了已经讨论过的不利因素外，使用 schedule_timeout 并不会带来额外的麻烦。从内核的角度，试图唤醒一个已经处于运行队列中的进程并不会造成多少困惑和工作量，好奇的读者可以看看 wake_up_process 的代码实现在这种情况下是如何处理的。

事实上，Linux 内核还提供了一个基于上述 schedule_timeout 版本的实现毫秒级睡眠的函数 msleep：

<kernel/timer.c>

```
void msleep(unsigned int msecs)
{
        unsigned long timeout = msecs_to_jiffies(msecs) + 1;

        while (timeout)
```

```
        timeout = schedule_timeout_uninterruptible(timeout);
}
```

schedule_timeout_uninterruptible 的内部实现其实就使用了 __set_current_state 和 schedule_timeout：

```
signed long __sched schedule_timeout_uninterruptible(signed long timeout)
{
    __set_current_state(TASK_UNINTERRUPTIBLE);
    return schedule_timeout(timeout);
}
```

注意到__set_current_state(TASK_UNINTERRUPTIBLE)将当前进程的状态设置为不可中断的 TASK_UNINTERRUPTIBLE，这样将可以确保进程将至少休眠用参数 msecs 指定的时间。内核中还提供了 msleep 的一个变体 msleep_interruptible：

<kernel/timer.c>

```
unsigned long msleep_interruptible(unsigned int msecs)
{
    unsigned long timeout = msecs_to_jiffies(msecs) + 1;

    while (timeout && !signal_pending(current))
        timeout = schedule_timeout_interruptible(timeout);
    return jiffies_to_msecs(timeout);
}
```

相对于 msleep 函数，msleep_interruptible 函数内部通过 schedule_timeout_interruptible 在当前进程睡眠之前设置其状态为 TASK_INTERRUPTIBLE，这样睡眠的进程将处于“可中断的睡眠”这样的状态，如果在 msecs 指定的延迟时间到期之前，进程因为接收到了信号而被唤醒，while 循环中的 signal_pending(current)将返回 true，那么 schedule_timeout_interruptible 将返回原先指定的休眠时间 msecs 的剩余时间值，这意味着 msleep_interruptible 将无法保证进程一定会在指定的延迟时间过后醒来。通常情况下 msleep_interruptible 都会返回 0，意味着进程完整地休眠了 msecs 指定的时间值。

所以在“让出处理器”实现长延时的方案中，直接调用内核提供的 msleep 类的函数是最简单最方便的一种方式了。

现在可以简单总结一下“让出处理器”这种延时方案的实现了，通过对直接单纯调用 schedule 方案的改进，我们获得了一个相对比较理想的解决方案：在等待延迟的时间段内，进程并不会占用处理器，因而也就不会影响处理器进入 idle 状态，相对于忙等待而言，这无疑是个很大的优势。说它相对比较理想，是因为它依然存在着延时精度的问题，不过这毕竟是系统内核的限制，不是我们的设备驱动程序做不到。另外要强调的是，到目前为止所有延迟的实现都是基于时钟滴答，因为它的实现机制导致了它在度量粒度上的固有限制（所以被称为“长延时”），所以如果需要实现粒度更细的延时，比如微秒甚至纳秒级，就需

要采用其他的方法了，这正是下一节“短延时”所要讨论的问题。

8.2.2 短延时

有时候设备驱动程序需要更短的延时，比如微秒甚至纳秒级的延迟，这种量级的时延有时候被通俗地称为“短延时”，形象地说明了在延迟粒度上与长延时的区别。基于下面的两个事实，这种量级延迟的实现已经不可能也不必要像前面讨论“长延时”那样直接用时钟滴答 jiffies 来实现：

（1）假设在通常的 HZ=1000 的系统上，1 秒钟内 jiffies 增加的数量为 HZ，也就是说 jiffies 的值每隔 1 毫秒才会增加 1，这意味着根据前后时间点的 jiffies 值来度量时间宽度的话，其分辨率最多也只能达到毫秒级。所以在这种情况下，要实现微秒级的延迟，单纯通过 jiffies 的差值已经不可能完成。

（2）在微秒级的水平上，必须要考虑到进程切换所带来的时间开销，因为进程切换所耗费的时间大约就在几个微秒到上百个微秒之间[6]。这种情况下，如果像“长延时”中“让出处理器”那样实现延迟，很有可能在进程切换的时候延迟的时间就到了。所以“短延时”一般都是基于忙等待来实现。

Linux 内核提供了毫秒、微秒和纳秒级的延迟实现：

```
<include/linux/delay.h>
void mdelay(unsigned long msecs);
void udelay(unsigned long usecs);
void ndelay(unsigned long nsecs);
```

这些延迟的实现都是基于忙等待，其中最基础的一个宏是 udelay，另外两个宏都是通过宏 udelay 来实现自身功能。udelay 的实现方法与体系架构相关。在讨论上述函数的实现时，一个很重要的变量是 loops_per_jiffy，用来表示在一个完整的 jiffies 时间段内运行一个内部循环的次数，只要知道了该值就能计算出比如 1 us 循环的次数，这样通过在一个循环中对该循环次数递减就可以在 1 us 的时间到期时退出循环，从而实现 1 us 的延时，这就是短延时函数实现的基本原理。当然如果特定的体系架构上有更精确的硬件计数器，比如 TSC，则利用它们来实现短延时要更精确、更容易。当然这样的实现方式是以牺牲跨平台的可移植性为代价的。Linux 内核有几种方法可以获得该 loops_per_jiffy 值，因为和驱动程序关系不大，所以此处不再详细讨论。

[6] 这里只是给出了大约的量级，精确测量 Linux 内核中发生一次进程切换的开销不是一件简单的事情。

8.3　内核定时器

内核定时器是设备驱动程序中经常要用到的另一个重要的内核设施。如果驱动程序希望在将来某个可度量的时间点到期后，由内核安排执行某项任务（此处的任务通常是驱动程序自身定义的某个函数，接下来的叙述中称之为定时器函数），便可以使用定时器来完成。

设备驱动程序中对内核定时器的一个典型使用场景是用它来实现轮询机制，因为定时器函数自身可以重新启用它所在的定时器，所以在一个时间段到期后，定时器函数被调用，在函数内部因为又重新启用了该定时器，这样便形成了一个不断循环的定时器函数被系统调用的模式。此种情形下，如果设备驱动程序需要周期性地检查设备的某种操作状态，便可以在定时器函数中来完成。

驱动程序等内核模块如果要使用定时器，首先应该定义一个定时器类型的变量。struct timer_list 是内核提供的一个用来表示定时器的数据结构，其定义如下（删去了一些用于调试及统计信息的成员）：

```
<include/linux/timer.h>
struct timer_list {
        /*
         * All fields that change during normal runtime grouped to the
         * same cacheline
         */
        struct list_head entry;
        unsigned long expires;
        struct tvec_base *base;

        void (*function)(unsigned long);
        unsigned long data;
        int slack;
};
```

其中在驱动程序中常用的是以下三个成员：

unsigned long expires

指定定时器的到期时间。

void (*function)(unsigned long)

定时器函数。当 expires 中指定的时间到期时，该函数将被触发。

unsigned long data

定时器对象中携带的数据。通常的用途是，当定时器函数被调用时，内核将把该成员

作为实际参数传递给定时器函数。之所以要这样做，是因为定时器函数将在中断上下文中执行，而非当前进程的地址空间中。

其他的一些成员将主要由内核使用，用以实现定时器的内核机制，在后面会看到这些成员的用法。

为了让读者对驱动程序使用内核定时器有个直观的印象，接下来将先给出一段示例代码，然后再对其中一些关键函数的使用及其内核实现机制进行分节讨论。下面的代码展示了一个设备驱动程序通过使用内核定时器来轮询设备状态。

```
struct device_regs *devreg = NULL; //定义一个用于表示设备寄存器的结构体指针
struct timer_list    demo_timer; //定义一个内核定时器对象

//
//定义定时器函数，当定时器对象 demo_timer 中 expires 成员指定的时间到期后，该函数将
//被调用
//
static void demo_timer_func (unsigned long data)
{
        //在定时器函数中重新启动定时器以实现轮询的目的
        demo_timer.expires = jiffies + HZ;
        add_timer(&demo_timer);

        //定时器函数将 data 参数通过类型转换获得设备寄存器的结构体指针
        struct device_regs *preg = (struct device_regs *) data;
        //定时器函数此后将会读取设备状态
        …
}

//
//用于打开设备的函数实现
//
static int demo_dev_open(…)
{
        …
        //分配设备寄存器结构体的指针变量，最好放在模块初始化函数中…
        devreg = kmalloc(sizeof(struct device_regs), GFP_KERNEL);
        …
        init_timer(&demo_timer); //调用内核函数 init_timer 来初始化定时器对象 demo_timer
        demo_timer.expires = jiffies + HZ; //设定定时器到期时间点，从现在开始的 1 秒钟
        demo_timer.data = (unsigned long) devreg; //将设备寄存器指针地址作为参数
        demo_timer.function = &demo_timer_func;
        add_timer(&demo_timer);
        …
}
```

```
//
//用于关闭设备的函数实现
//
static int demo_dev_ release(…)
{
        …
        del_timer_sync(&demo_timer); //删除定时器对象
        …
}
```

8.3.1　init_timer

在前面的示例代码中，demo_dev_open 函数在对定时器对象 demo_timer 的 expires、data 和 function 成员赋值前，调用了 init_timer 函数（内核源码中以宏定义的形式出现）。init_timer 函数内部会调用__init_timer，其定义如下（去除了 struct timer_list 中一些调试相关成员的代码）：

<kernel/timer.c>

```
static void __init_timer(struct timer_list *timer,
                         const char *name,
                         struct lock_class_key *key)
{
        timer->entry.next = NULL;
        timer->base = __raw_get_cpu_var(tvec_bases);
        timer->slack = -1;
}
```

可见 init_timer 函数主要初始化定时器对象中与内核实现相关的成员，所以设备驱动程序在开始使用定时器对象前，应该调用 init_timer，这样从内核层面出发，后续对定时器的一些操作才会被内核所支持，下面在讨论 add_timer 函数时会看到这一点。

8.3.2　add_timer

当程序定义了一个定时器对象，并且通过 init_timer 函数及相应代码对该定时器对象中的 expires、data 和 function 等成员初始化之后，程序需要调用 add_timer 将该定时器对象加入到系统中，这样定时器才会在 expires 表示的时间点到期后被触发。可以想见，add_timer 函数的内部实现将不再独立，它必然会和内核中关于定时器的基础架构发生关联。

内核自身对于定时器的管理与操作设计有一个非常完整的框架，详细讨论这些技术细节需要相当的篇幅，其中大量的内容属于内核实现的范畴。因此我们决定将后续的讨论限定在设备驱动程序员需要关注的范围之内，也即在更广的范围内我们给出定时器内核实现原理

的大体架构，在更细分的范围我们重点讨论与驱动程序中对定时器的使用等密切相关的部分。这样的安排相信对于设备驱动程序员而言是合理的：在了解了基本原理的前提下，通过对内核如何组织和调用到期的定时器函数的讨论，现实中我们将知道如何更安全更有效地使用定时器，这也是写作本书的主要目的。

接下来将首先讨论内核如何管理系统中的定时器，然后会看到定时器函数如何在指定的时间到期后被调用，最后会讨论 add_timer 函数是如何将一个定时器对象加入到系统中的。

内核中定义了一个数据结构 struct tvec_base 来管理系统中添加的所有定时器，其定义如下：

<kernel/timer.c>

```
struct tvec_base {
        spinlock_t lock;
        struct timer_list *running_timer;
        unsigned long timer_jiffies;
        unsigned long next_timer;
        struct tvec_root tv1;
        struct tvec tv2;
        struct tvec tv3;
        struct tvec tv4;
        struct tvec tv5;
} ____cacheline_aligned;
```

其中的 tv1、tv2、tv3、tv4 和 tv5 被内核用来对系统中注册的定时器进行散列式的管理，后面会看到其用法。内核为系统中的每个 CPU 都定义了一个 struct tvec_base 类型的变量 tvec_bases：

<kernel/timer.c>

```
static DEFINE_PER_CPU(struct tvec_base *, tvec_bases) = &boot_tvec_bases;
```

tvec_bases 用来将系统中加入的每个定时器组织管理起来。用简单的单一链表结构当然也可以实现这一目标，然而系统会在每个时钟中断中去扫描该链表并要分辨出哪些定时器已经到期或者是即将到期，所以必须使得这一任务的执行效率非常高以消耗极小的 CPU 资源。因此内核采用了上述 struct tvec_base 结构来组织链表，读者可以简单地认为它是基于哈希表的一个实现。每当设备驱动程序通过 add_timer 向系统添加一个定时器对象时，系统都会对该定时器对象的到期时间 expires 进行分类，根据到期时间的长短将当前定时器对象放到 struct tvec_base 对象的成员 tv1、tv2、tv3、tv4 和 tv5 领衔的定时器链表中。比如，其中的 tv1 中定时器的到期时间范围是 0~255 个时钟周期，前面已经看到了 struct tvec_base 结构的定义，它的成员 tv1 其实也是个数组，大小是 256，分别对应 expires 为 0~255 个 jiffies 的定时器，如果有多个到期时间相同的定时器，则它们将会以双链表的形式链接到同一数组项中。其他的成员 tv2、tv3、tv4 和 tv5 用来存放到期时间更久的定时器，除此之外与 tv1 的原理是一样的。图 8-2 为向系统中添加一个定时器对象的示意图。

至此程序只是完成了向系统添加一个定时器对象的工作，接下来讨论添加的定时器对象在指定的时间到期时如何被触发，也就是定时器对象中的定时器函数何时被调用的问题。

我们知道，Linux 内核一秒中都会发生很多次的时间中断，在每个时钟中断处理函数中，严格地说是时钟中断处理的下半部也就是 softirq 部分，会对 tvec_bases 管理的定时器队列进行扫描，以确定当前队列中有哪些定时器已经到期。

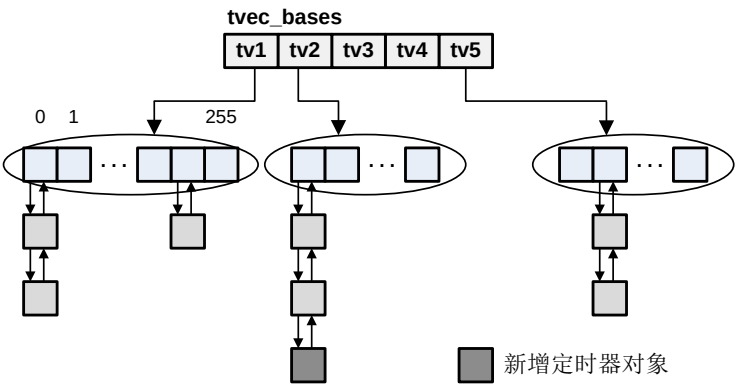

图 8-2　通过 add_timer 向系统新增一个定时器对象

Linux 内核中时钟中断的 softirq 为 TIMER_SOFTIRQ，对应的软中断处理函数的安装发生在系统初始化阶段的 init_timers 函数中：

<kernel/timer.c>

```
void __init init_timers(void)
{
        …
        open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

所以，当时钟中断的 softirq 被调度执行时，它将运行对应的 run_timer_softirq 函数。在每个时钟中断处理的上半部分，都会调用 run_local_timers 函数，后者则通过使用 raise_softirq 函数来触发时钟中断的 softirq 部分：

<kernel/timer.c>

```
void run_local_timers(void)
{
        hrtimer_run_queues();
        raise_softirq(TIMER_SOFTIRQ);
        softlockup_tick();
}
```

所以，当时钟中断的 softirq 部分被调度执行时，run_timer_softirq 会负责扫描 tvec_bases 所在的定时器管理队列，找到已经到期的函数，然后调用到期定时器对象节点上的定时器函数：

<kernel/timer.c>

```
static void run_timer_softirq(struct softirq_action *h)
{
        struct tvec_base *base = __get_cpu_var(tvec_bases);
        …
        if (time_after_eq(jiffies, base->timer_jiffies))
```

```
        __run_timers(base);
}
```

run_timer_softirq 对于那些到期的定时器队列调用__run_timers 函数进一步处理，后者的部分核心代码如下：

<kernel/timer.c>

```
static inline void __run_timers(struct tvec_base *base)
{
    struct timer_list *timer;

    spin_lock_irq(&base->lock);
    while (time_after_eq(jiffies, base->timer_jiffies)) {
        …
        while (!list_empty(head)) {
            void (*fn)(unsigned long);
            unsigned long data;

            timer = list_first_entry(head, struct timer_list,entry);
            fn = timer->function;
            data = timer->data;
            …
            detach_timer(timer, 1);

            spin_unlock_irq(&base->lock);
            call_timer_fn(timer, fn, data);
            spin_lock_irq(&base->lock);
        }
    }
}
```

函数的总体思想是，对 tvec_bases 管理的定时器队列进行扫描，如果发现有定时器到期（代码中用 time_after_eq 来进行判断），则调用该定时器对象的 fn 函数（fn = timer->function，fn(data)），这个过程发生在 call_timer_fn 函数中。读者需要注意，在调用 call_timer_fn 前__run_timers 调用了 detach_timer(timer, 1)，该函数会把当前正在处理的定时器对象从 tvec_bases 中删除，所以当一个定时器对象中的定时函数被调用时，该定时器对象已经从系统的定时器队列中删除了，所以如果要让该定时器对象在以后能继续被系统所调用，则需要再次调用 add_timer 或者是 mod_timer 来将该定时器对象重新加入到系统中去，这是设备驱动程序用定时器来实现轮询机制的基本原理。

通过上面的讨论可以知道，由于内核对系统中的定时器队列的扫描发生在时钟中断的softirq部分，鉴于softirq的实现机制[7]，在某些情况下可能会导致当一个定时器对象中的定时器函

[7] 关于 softirq 的实现原理，可以参考“中断处理”一章。

数被调用时，实际的jiffies值已经超出了当时安装定时器时预设的jiffies值，换句话说，使用定时器也同样存在着实际到期时间点延伸的问题，如果使用当中对定时精度有严格的要求，那么也许要考虑在现有的通用内核上加入某些实时性的扩展。

8.3.3 del_timer 和 del_timer_sync

同 add_timer 函数相反，del_timer 类的函数负责从系统的定时器管理队列中摘除一个定时器对象。del_timer 和 del_timer_sync 的函数原型为：

<kernel/timer.c>

```
int del_timer(struct timer_list *timer)；
int del_timer_sync(struct timer_list *timer)；
```

del_timer 与 del_timer_sync 函数只在 SMP 系统上才有所区别，在单处理器系统中，del_timer_sync 等同于 del_timer。

对于 del_timer 函数要摘除的定时器对象 timer，函数会首先判断该对象是否是一个 pending 的定时器，一个处于 pending 状态的定时器是处在处理器的定时器管理队列中正等待被调度执行的定时器对象。如果一个要被 del_timer 函数删除的 timer 对象已经被调度执行（内核源码称这种定时器状态为 inactive），函数将直接返回 0，否则函数将通过 detach_timer 将该定时器对象从队列中删除。在多处理器的 SMP 系统中，del_timer_sync 函数要完成的任务除了同 del_timer 一样从定时器队列中删除一个定时器对象外，还会确保当函数返回时系统中没有任何处理器正在执行定时器对象上的定时器函数，而如果只是调用 del_timer，那么当函数返回时，被删除的定时器对象的定时器函数可能正在其他处理器上运行。

8.4 本章小结

本章讨论了设备驱动程序中可能用到的与时间度量和定时相关的话题。这类时间管理相关的任务从总体上可以分成两大类，一类是延迟当前处理器的执行，另一类是设定一个延迟时间点，当该延迟时间点到期后执行特定的动作。

对于延迟的实现，又可以分为“忙等待”和“让出处理器”两种方式，前者是让当前的处理器进入到一个不断的循环中以实现延迟当前的执行，因为处理器在这种循环中无法进行其他任务的处理，所以这种方式会浪费 CPU 的资源。因此为了改善这种处理器浪费的现象，又有了所谓“让出处理器”的延迟方式，相对于“忙等待”，前者会在当前进程的延迟期间让出处理器，这样处理器就可以用来执行别的进程，从而提高其利用率。但如果程序需要的延迟时间非常短，比如只在微秒甚至纳秒级，这种情况如果采用“让出处理器”的方式来实现，那么由于进程切换的时间开销大约也是在微秒这个级别上，所以极有可能当前需要延迟执行进程刚被切换出处理器，延迟的时间就已经到了，此时又需要重新将该进程切

换至处理器，如此效率反而不高。

定时器在设备驱动程序中最常见也最典型的使用场景是用来实现轮询机制，内核为定时器机制设计了一套完整的机制，对于设备驱动程序而言，只需定义一个定时器对象并指定其到期时间及实现一个定时器函数，然后通过 add_timer 或者 mod_timer 将该定时器对象加到系统中即可。当一个定时器对象到期被执行时，内核会将其从系统的定时器管理队列中摘除下来，所以为了实现轮询，驱动程序需要在定时器函数中重新将该定时器对象加入到管理队列中。因为定时器的实现机制是基于系统中的硬件时钟中断，因为受硬件时钟精度以及时钟中断 softirq 固有的实现特性，定时器的精度并非完美，但是对于绝大多数的设备驱动而言已经足够了。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

- 穿针引线，将Linux设备驱动程序从台前到幕后融会贯通
- 条分缕析，剖析Linux设备驱动程序使用到的每一个重要的内核设施
- 高屋建瓴，多层次立体化揭示Linux设备驱动程序的框架结构
- 化繁为简，简单的示例源码具体验证内核背后的运作机制

上架建议：Linux

ISBN 978-7-121-15052-4

9 787121 150524 >

定价：98.00元

策划编辑：张春雨
责任编辑：白　涛
封面设计：侯士卿