第 1 章 Linux 内核简介

世界各地都有人在钻研 Linux 内核，大多是在写设备驱动程序。尽管每个驱动程序都不一样，而且你还要知道自己设备的特殊性，但是这些设备驱动程序的许多原则和基本技术技巧都是一样的。通过本书，可以学会写自己的设备驱动程序，并且可以钻研内核的相关部分。本书涉及到的是设备无关编程技巧，不会将例子跟特殊设备绑定在一起。

本章没有实际编写代码。但我要介绍一些关于 Linux 内核的背景概念，这样到我们稍后开始介绍实际编程时，就很顺利了。

当你学习编写驱动程序的时候，你也会发现很多关于 Linux 内核的知识，这对理解你机器怎么工作很有帮助，并且还可以知道为什么你的机器没有希望的那么快，或者为什么不按照你象要它做的那样做。我们会逐渐介绍一些新概念，先从简单的驱动程序开始，每介绍一些新概念都会看到相关例子代码，这些代码都不需要特殊硬件。

驱动程序作者的作用

作为一个程序员，你可以选择自己的驱动程序，在编程所需时间和结果的灵活性之间做个可以接受的权衡。尽管说驱动程序的灵活性看起来有那么点怪，我喜欢这个词是因为它强调了设备驱动程序提供的是机制，而不是策略。

机制和策略之间的差别是 Unix 设计背后最好的点子之一。实际编程中遇到的大多数问题都可以被划分成两个部分：“需要作什么”（机制）和“这个程序怎么用”（策略）。如果这两个主题是由程序不同部分来承担的，或者是由不同的程序组合一起承担的，那么这个软件包就很容易开发，也很适合特殊需求。

举个例子，Unix 的图形显示管理在 X 服务器和窗口管理器之间划了一道线，X 服务器了解硬件并给用户程序提供唯一的接口，而窗口管理器实现特殊的策略并不需要知道硬件的任何信息。人们可以在不同硬件上使用同样的窗口管理器，并且不同用户在同一台工作站上可以使用不同的设置。另一个例子是 TCP/IP 的网络分层结构：操作系统提供抽象的套接字操作，是设备无关的，不同服务器主管这个服务。另外，ftpd 服务器提供文件传输机制，而用户可以使用任何客户端程序；命令行的客户端和图形化界面的客户端都存在，并且谁都可以为传输文件写一个新的用户界面。

只要涉及到驱动程序，就会运用这样的功能划分。软盘驱动程序是设备无关的——这不仅表现在磁盘是一个连续读写的字节数组上。如何使用设备是应用程序要做的事：tar 要连续地写数据，而 mkfs 则为要安装的设备做准备工作，mcopy 依赖于设备上存在的特殊数据结构。在写驱动程序时，程序员应该特别留心这样的基本问题：我们要写内核代码访问硬件，但由于不同用户有不同需要，我们不能强迫用户采用什么样的特定策略。设备驱动程序应该仅仅处理硬件，将如何使用硬件的问题留给应用程序。如果在提供获得硬件能力的同时没有增加限制，我们就说驱动程序是灵活的。不过，有时必须要作一些策略决策。

可以从不同侧面来看你的驱动程序：它是位于应用层和实际设备之间的软件。驱动程序的程序员可以选择这个设备应该怎样实现：不同的驱动程序可以提供不同的能力，甚至相同的设备也可以提供不同能力。实际驱动程序设计应该是在众多需求之间的一个平衡。例如，不同程序可以同时使用同一个设备，而驱动程序的开发者可以完全自由地决定如何处理同步机制。你可以实现到设备上的内存映射，而完成独立于硬件的具体能力，或者你可以提供给用

户函数库，帮助应用程序的程序员在可用原语的基础上实现新策略，或者诸如此类的方法。一个很重要需要考虑的问题就是，如何在提供给用户尽可能多的选项，平衡你需要编写所花费的时间，以及为使错误尽可能少而保持代码简单之间的平衡。

如果即为同步又为异步操作设计驱动程序，如果允许同时打开多次，并且如果能够发掘所有硬件功能，而不用增加软件层“去简化事情”——例如将二进制数据转换成文本或者策略相关的操作——那就很容易编写而且很好维护了。达成“策略无关”实际上是软件设计的共同目标。

实际上，大多数设备驱动程序是和用户程序一起发布的，这些程序可以帮助完成对目标设备的配置和访问。这些程序可以是从简单的配置程序到完整的图形应用。通常还要提供一个客户端库文件。

本书讨论范围是内核，所以我们不考虑策略问题，也不考虑应用程序或支持库。有时，我们会讨论不同策略，以及如何支持这些策略，但我们不会深入到使用一定策略或设备编程需要的细节问题。不过你应该可以理解，用户程序是一个软件包的内核，就算策略无关的软件包也会和配置文件一起发布，这些文件提供了基本机制上的缺省行为。

划分内核

在 Unix 系统中，若干并发进程会参加不同的任务。每个进程都要求获得系统资源，可以是计算、内存、网络连接或别的资源。内核是一整块可执行代码，用它来负责处理所有这样的请求。尽管在不同的内核任务之间的区别不是总能清楚地标识出来，内核的作用还是可以被划分的。如图 1-1 所示，划分为如下这些部分：

进程管理

内核负责创建和终止进程，并且处理它们和外部世界的联系（输入和输出）。对整个系统功能来讲，不同进程之间的通信（通过信号，管道，进程间通信原语）是基本的，这也是由内核来处理的。另外，调度器，可能是整个操作系统中最关键的例程，是进程管理中的一部分。更广广义的说，内核的进程管理活动实现了在一个 CPU 上多个进程的抽象概念。

内存管理

计算机内存是主要资源，而使用内存的策略是影响整个系统性能的关键。内核为每个进程在有限可利用的资源上建立了虚拟地址空间。内核不同部分通过一组函数与内存管理子系统交互，这些包括从简单的 malloc/free 到更稀奇古怪的功能。

（图 1-1）

文件系统

Unix 系统是建立在文件系统这个概念上的；Unix 里几乎所有东西都可以看作文件。内核在非结构的硬件上建立了结构化的文件系统，这个抽象的文件被系统广泛应用。另外，Linux 支持多文件系统类型，即，物理介质上对数据的不同组织方法。

设备控制

几乎每种系统操作最后都要映射到物理设备上。除了处理器，内存和少数其他实体外，几乎所有设备的控制操作都由设备相关的代码来实现。这些代码就是设备驱动程序。内核必须为

每个外部设备嵌入设备驱动程序，从硬盘驱动器到键盘和磁带。内核的这方面功能就是本书的着眼点。

网络

网络必须由操作系统来管理，由于大多数网络操作不是针对于进程的：接收数据包是异步事件。数据包必须在进程处理它们以前就被收集，确认和分发。系统通过程序和网络接口发送数据包，并且应该可以正确地让程序睡眠，并唤醒等待网络数据的进程。另外，所有路由和地址解析问题是在内核里实现的。

在本书结尾部分的第 16 章“内核源码的物理布局”里，您可以看到 Linux 内核的路标，但现在这里的话应该足够了。

Linux 的一个很好的特征就是，它可以在运行的时候扩展内核代码，也就是说在系统运行的时候你可以增加系统的功能。

每个可以增加到内核中的代码称为一个模块。Linux 内核支持相当多的模块的类型（或“类”），但不仅仅只局限于设备驱动程序。每个模块由目标代码组成（没有连接成完整的可执行文件），通过 insmod 程序它们可以动态连接到运行着的内核中，而通过 rmmod 程序就可以去除这些模块。

在图 1-1 中，你可以标别出处理不同任务的不同模块类别——根据模块提供的功能，每个模块属于一个特定的类。

设备类和模块

在 3 类设备中，Unix 看待设备的方式有所区别，每种方式是为了不同的任务。Linux 可以以模块的形式加载每种设备类型，因此允许用户在最新版本的内核上实验新硬件，跟随内核的开发过程。

一考虑到模块，每个模块通常只实现一个驱动程序，因此是可以分类的。例如，字符设备模块，或块设备模块。将模块分成不同的类型或类并不是固定不变的；程序员可以选择在单独一整块代码中创建一个模块实现不同的驱动程序。不过好的程序员会为他们实现的每一个新功能创建不同模块。

现在回到驱动程序，有如下三种类型：

字符设备

可以象文件一样访问字符设备，字符设备驱动程序负责实现这些行为。这样的驱动程序通常会实现 open，close，read 和 write 系统调用。系统控制台和并口就是字符设备的例子，它们可以很好地用流概念描述。通过文件系统节点可以访问字符设备，例如/dev/tty1 和/dev/lp1。在字符设备和普通文件系统间的唯一区别是：普通文件允许在其上来回读写，而大多数字符设备仅仅是数据通道，只能顺序读写。当然，也存在这样的字符设备，看起来象个数据区，可以来回读取其中的数据。

块设备

块设备是文件系统的宿主，如磁盘。在大多数 Unix 系统中，只能将块设备看作多个块进行访问为，一个块设备通常是 1K 字节数据。Linux 允许你象字符设备那样读取块设备——允许一次传输任意数目的字节。结果是，块设备和字符设备只在内核内部的管理上有所区别，因此也就是在内核/驱动程序间的软件接口上有所区别。就象字符设备一样，每个块设备也

通过文件系统节点来读写数据，它们之间的不同对用户来说是透明的。块设备驱动程序和内核的接口和字符设备驱动程序的接口是一样的，它也通过一个传统的面向块的接口与内核通信，但这个接口对用户来说时不可见的。

网络接口

任何网络事务处理都是通过接口实现的，即，可以和其他宿主交换数据的设备。通常，接口是一个硬件设备，但也可以象 loopback（回路）接口一样是软件工具。网络接口是由内核网络子系统驱动的，它负责发送和接收数据包，而且无需了解每次事务是如何映射到实际被发送的数据包。尽管“telnet”和“ftp”连接都是面向流的，它们使用同样的设备进行传输；但设备并没有看到任何流，仅看到数据报。

由于不是面向流的设备，所以网络接口不能象/dev/tty1 那样简单地映射到文件系统的节点上。Unix 调用这些接口的方式是给它们分配一个独立的名字（如 eth0）。这样的名字在文件系统中并没有对应项。内核和网络设备驱动程序之间的通信与字符设备驱动程序和块设备驱动程序与内核间的通信是完全不一样的。内核不再调用 read，write，它调用与数据包传送相关的函数。

事实上，Linux中还有一类“设备驱动程序模块”：SCSI*设备驱动程序。尽管每个连接到SCSI总线上的外设在/dev目录中不是字符设备就是块设备，但软件的内部组织并不完全同。

正如网络接口给网络子系统提供硬件相关的功能一样，SCSI 控制器提供给 SCSI 子系统如何访问实际接口电缆。SCSI 是计算机和外设之间的通信协议，每种 SCSI 设备都响应相同的协议，与计算机插的是哪种控制板没有关系。因此，Linux 内核嵌入一个所谓 SCSI“实现”（即，文件操作到 SCSI 通信协议的映射）。驱动程序编写人员必须在 SCSI 抽象层和物理电缆之间实现这种映射。这种映射依赖于 SCSI 控制器，却与 SCSI 电缆上连接的设备无关。

除了设备驱动程序，还有一些别的模块化加载到核心中的驱动程序，可以是软件，也可以时硬件。并非实现驱动程序的模块中最重要的一类是文件系统。文件系统类型是与为了表示目录和文件等实体的信息的组织方式相关的。因此这种实体不是“设备驱动程序”，其中并没有确定某种设备与信息组织的方式有关；由于文件系统将原始数据组织为高层信息，它实际上是软件驱动程序。

如果你考虑到 Unix 系统对底层文件系统的依赖程度，你就可以意识到软件概念对系统操作的重要性。文件系统信息解码的能力位于内核分层的最底层，而且是最重要的；甚至如果给自己的 CD-ROM 写一个新的块设备驱动程序时，要是不能对其上数据上运行 ls 或 cp，那个驱动程序就根本没有什么用处。Linux 支持文件系统模块，它的软件接口声明了可以操作在文件系统节点，目录，文件和超级块上的操作。因此，接口与实际数据传出传入磁盘是完全独立的，这是由块设备驱动程序完成的。对程序员来讲，由于正式内核中已经包含大部分重要文件系统类型的代码，编写一个文件系统模块是很不寻常的要求。

安全问题

最近讨论安全问题很是时髦，而大多数程序员都考虑过系统的安全问题，所以，为防止产生误解，一开始我就要谈论这个问题。

安全性有两方面。一个问题是由于用户对程序的误操作或发掘出错误造成的；另一个问题是由程序员实现的（错误）功能造成的。显然，程序员比普通用户拥有更多的权利。换句话说

---

* SCSI时Small Computer System Interface（小型计算机系统接口）的缩写；它时工作站市场上形成的标准，也广泛应用在PC领域。

就是，以 root 权限运行从朋友那拿来的程序比给他/她一个 root 外壳要危险得多。尽管访问编译器本质上不是安全性漏洞，但当所编译的代码执行时，还是会出现漏洞；要小心处理模块，因为内核模块可以做任何事。模块比超级用户的外壳的威力还要强大，它的特权是由 CPU 确认的。

所有系统中的安全性检查都是内核代码完成的。如果内核有安全性漏洞，系统就有漏洞。在内核正式发布版本中，只有 root 可以加载模块；系统调用 create_module 检查调用进程的用户 ID。因此在正式内核中，只有超级用户，或是成功地称为 root 的闯入者可以利用特权代码的威力。

幸亏在编写设备驱动程序或别的模块时，很少需要考虑安全性问题，因为访问块设备的进程已经受到更通用的块设备技术的严格控制了。例如，对于块设备来说，安全是由文件系统节点的权限和 mount 命令处理的，因此，在实际的块设备驱动程序中通常没有什么好检查的。尽管如此，从收到第三方软件开始，尤其是当软件涉及到内核时，要格外小心；由于每个人都可以访问源代码，都可以改写和重新编译这些东西。如果可以信任发布版中已经编译好的内核，就要避免编译来源不可靠的内核——如果你不能以 root 身份运行编译好的二进制代码，那最好不要运行编译好的内核。例如，有敌意改动过的内核可能允许任何人加载模块，因此通过 create_module 就可以打开一个不希望出现的后门。

如果确实在模块相关部分考虑到安全性问题，我敦促你看一看 securelevel 内核变量是怎样使用的。当我写这些的时候，Linux 社团里正在讨论控制 securelevel 变量来防止模块加载和卸载。很有趣，你可以注意到最近的内核支持在内核编译时可以删除对模块的支持，这样就关闭了所有安全性有关的漏洞。

版本编号

做为开始研究编程前的最后一点，我很愿意就 Linux 非同寻常的版本编号这个问题加以说明，同时说明本书使用哪个版本的内核。

首先要注意，Linux 系统使用的每个软件包都有它自己的发行号，它们之间有一些内部的依赖关系：你需要在特定版本的软件包上使用特定版本的另一个软件包。Linux 发布版的开发者通常要处理大量软件包的匹配问题，而使用者从预先打包的发布版本上安装软件，无需考虑版本问题。而另一方面，那些更换和升级系统软件的人都要自己处理版本问题。所幸的是，一些最新发布的版本允许单个软件包的升级，它是通过检查软件包间的依赖性实现的，这可大大简化了使用者为维护系统软件一致所要做的事情。

在本书中，我假定你有 2.6.3 版或更新的 gcc 编译器，1.3.57 版或更新版本的模块工具，以及最新的程序开发用 GNU 工具（最重要的是 gmake）。这些要求不是那么严格，由于几乎所有的 Linux 安装版上都配有 GNU 工具，这些版本都相对较旧（此外，内核 2.0 版及其后继版本不能用比版本 2.6 还旧的 gcc 编译）。注意，最近的内核包含一个称为 Documentation/Changes 的文件，它罗列了所有编译和运行这个内核版本所需要的软件。1.2 的源码中没有这个文件。

只要涉及到内核部分，我将集中介绍 2.0.X 和 1.2.13 版本，编写适用于这两个版本的代码。偶数内核版本（如 1.2.x 和 2.0.x）是稳定版本，专门用于发布的版本。而相反，奇数版本是开发中的一个快照，相当短暂；最新的版本代表最新的发展状况，但可能过几天就过时了。这里没有别的通用原因来解释为什么要运行 1.3 和 2.1 核心，除非他们是最新的版本。不过有时你会选择运行一个开发用内核，或时由于它有一些在稳定版本中没有的特性，而你正好需要这些特性，或者很简单，你就是自己改动了这些版本的一些特性，并且你没有时间更新你的补丁。不过还是要注意，实验用内核没有什么保证，如果由于在非当前奇数版本的内核

中的臭虫导致你丢失数据，没人帮得了你。不过，本书支持直到 2.1.43 版的开发用内核，最后一章介绍如何编写可以区分 2.0 和 2.1.x 之间不同之处的驱动程序。
至于 1.2.13，尽管相当旧，我仍觉得这是一个很重要的版本。尽管在某些平台上 2.0.x 比 1.2.13 要快得多，但 1.2.13 非常小，对想使用旧硬件的人来说是个很好的选择。基于 386 处理器，带小 RAM 的低价系统非常适合于做嵌入式系统或自动化控制器，用 1.2.13 比 2.0.x 也许会更快，由于 1.2.13 是以前 1.2.x 版本的错误修订版，我不想考虑更早的 1.2 内核。
无论什么时候，只要 1.2.13 和 2.0 或最新的 2.1 版不兼容，我都会告诉你。
不管怎样，我的主要目标版本是 Linux 2.0，本书介绍的一些特征在旧版本中是没有的。大多数样例模块在很多内核版本中都是可以编译和运行的；特别是它们都已经在 2.0.30 版本上测试过了，而且大多数也都经过了 1.2.13 版本的测试。有时，我的例子不支持 1.2，但只在本书的第二部分中出现这样的情况，这部分对设计来讲更深入，并且可以不用考虑旧版本。
由于 Linux 已不再只是“PC 兼容机的 Unix 变体”了，它的另一个特点是，它是平台独立的操作系统。事实上，除了 x86 以外，它也成功地用于 Axp-Alpha，Sparc 处理器，Mips Rx000 和其他一些平台。本书也尽力达到平台无关性，而且所有的例子代码都在 PC，Alpha 平台和 Sparc 机器上测试过。由于代码在 32 位和 64 位处理机（Alpha）上测试过，它们应该可以在其他平台上编译和运行。正如你所预料的，依赖特殊硬件的编码不能适用于所有的平台，但在源代码中都有所说明。

许可证术语

Linux 是按 GNU“General Public License”(GPL)分布许可证的。这是由自由软件基金会为 GNU 计划设计的文档。GPL 允许任何人重新发布，出售 GPL 的产品，只要允许接收者从源码重新建立二进制文件的精确副本。另外，任何从 GPL 产生的软件产品也必须按 GPL 发布。这种许可的主要目的是通过允许每个人随意修改程序来推广知识；同时，向公众出售软件的人仍可以做他们的工作。尽管这是个很简单的目标，还是有一些正在进行的有关 GPL 及其使用的讨论。如果你想读到这些许可证，你可以在你系统的若干个地方找到它们，包括目录 /usr/src/linux，有一个称为 COPING 文件。
当涉及到第三方和定制的模块时，它们不属于内核，因此你无需限制它们使用 GPL 许可证。模块通过明确的接口使用内核，它不是内核的一部分，这种关系和用户程序通过系统调用使用内核很相似。
简而言之，如果你的代码深入内核内部而你又想发布代码，你必须使用 GPL。尽管自己修改自己使用不是非要使用 GPL，如果你要发布代码，就必须在发布中包含源代码——人们获得你的软件包，并且可以随意修改它，重建二进制文件。换句话说，如果你写了一个模块，你就可以按照二进制格式发布你的模块。然而，由于模块通常要针对要连接的内核重新编译（在第 2 章“编写和运行模块”中的“版本相关性”一节中，第 11 章“Kerneld 和高级模块化”的“模块中的版本控制”一节中都有所介绍），这也不总是可行的。对发布模块的二进制代码一般障碍是，模块包含了定义或声明在内核头文件中的代码；不过这个障碍并不能成立，因为头文件是内核公共接口的一部分，因此它不受许可证制约。
至于谈到本书，大部分源码都是可以重新发布的，或者以源码形式，或者以二进制代码形式，不论 O’Reilly 还是我都不对任何基于此的工作有任何许可证权。所有程序都可以通过 FTP 从 ftp://ftp.ora.com/pub/examples/linux/drivers/下载，而且同一个目录下的 LICENCE 文件有具体的许可证说明。
当例子中包含了部分内核代码时，GPL 就适用了：与源码一同发行的文本很清楚地说明了这一点。这仅对某些源文件是有效的，对本书主题而言，这是次要的。

全书概貌

从此开始，我们进入内核编程的世界。第 2 章介绍模块化，解释了这门技艺的秘密，并给出了运行模块的代码。第 3 章，字符设备驱动程序，讨论字符设备驱动程序并且给出了基于内存的设备驱动程序的完整代码，可以按你的喜好进行读写。使用内存做为设备的硬件基础，可以使任何人运行例子代码，而无需增加特殊硬件。

调式技术对程序员来讲是至关重要的，这些内容在第 4 章“调试技术”中介绍。这样，运用我们新的调试技巧，我们将面对字符设备驱动程序高级功能，如阻塞型操作，select 的使用以及非常常用的 ioctl 调用；这是都是第 5 章“字符设备驱动程序的扩展操作”的主题。

在涉及硬件管理之前，我们先解剖几个内核软件接口：第 6 章“时间流”，讲解内核是如何管理时间的，第 7 章“获取内存”，讲解内存分配。

接下来我们着重于硬件：第 8 章“硬件管理”，介绍 I/O 端口的管理和设备中的内存缓冲区管理；之后在第 9 章“中断处理”介绍中断处理。遗憾的是，由于需要某些硬件支持来测试中断的软件接口，不是每个人能运行本章给出的样例代码。我已经尽我全力保持所需的硬件支持减少到最小，但你还得亲自动手用烙铁做你的硬件“设备”。这个设备仅仅是一个加到并口上的跳线，所以我希望这不是问题。

第 10 章“合理使用数据类型”又提供一些有关编写内核软件和一致性问题的建议。

在本书的第二部分，我们更加雄心勃勃；因此从第 11 章开始，我们重新讨论模块化，更加深入讨论这个问题。

第 12 章“加载块设备驱动程序”介绍了如何实现块设备驱动程序，强调和字符设备驱动程序的区别。接下来，第 13 章“Mmap 和 DMA”讲解了我们原先在内存管理中留下来的问题：mmap 和 DMA。到此为止，关于字符设备和块设备驱动程序的所有问题我们都介绍过了。

接下来介绍第三类设备驱动程序：第 14 章“网络设备驱动程序”讨论一些关于网络接口的细节，剖析了样例网络设备驱动程序的代码。

有些设备驱动程序的功能直接依赖于外设所在的接口总线，所以第 15 章“外设总线概貌”介绍了现在经常用到的总线实现的主要功能，着重介绍内核支持的 PCI 总线。

最后，第 16 章是内核源代码的一次检阅：对那些想理解全部设计的人来讲，这是一个起点，但他们可能会被 Linux 浩如烟海的代码吓倒。

在 Linux2.0 版发布后不久，2.1 开发树开始引入不兼容性；这是在第一个月中引入的最重要的内容。第 17 章“近期发展”，它几乎可以看作是附录，它收集所有在 2.1.43 版本发布之前不兼容的东西，并且提供了解决这些兼容性问题的方法。在这章的最后，你可以编写出一个设备驱动程序，它能够在 1.2.13 版本上编译，运行，也可以在所有 2.0 和 2.1.43 版本之间的内核上编译，运行。2.2 很有希望会和 2.1.43 非常相似，你的软件需要为此做好准备。