深入分析 Linux 内核源码

第五章进程调度

第六章 Linux 内存管理

第十二章 网络

第十三章 启动系统

附录：

前　言

Linux 内核全部源代码是一个庞大的世界，大约有 200 多万行，占 60MB 左右的空间。因此，如何在这庞大而复杂的世界中抓住主要内容，如何找到进入 Linux 内部的突破口，又如何能把 Linux 的源代码变为自己的需要，这就是本书要探讨的内容。

首先，本书的第一章领你走入 Linux 的大门，让你对 Linux 内核的结构有一个整体的了解。然后，第二章介绍了分析 Linux 源代码应具备的基本硬件知识，这是继续向 Linux 内核迈进的必备条件。中断作为操作系统中发生最频繁的一个活动，本书用一章的内容详细描述了中断在操作系统中的具体实现机制。

大家知道，操作系统中最核心的内容就是进程管理、内存管理和文件管理。本书用大量的篇幅描述了这三部分内容，尤其对最复杂的虚拟内存管理进行了详细的分析，其中对内

存初始化部分的详细描述将对嵌入式系统的开发者有所帮助。

在对 Linux 内核有一定了解后，读者可能希望能够利用内核函数进行内核级程序的开发，例如开发一个设备驱动程序。Linux 的模块机制就是支持一般用户进行内核级编程。另外，读者在进行内核级编程时还可以快速查阅本书附录部分提供的 Linux 内核 API 函数。

网络也是 Linux 中最复杂的部分之一，这部分内容足可以写一本书。本书仅以面向对象的思想为核心，分别对网络部分中的四个主要对象：协议、套接字、套接字缓冲区及网络设备接口进行了分析。有了对这四个对象的分析，再结合文件系统、设备驱动程序的内容，读者就可以具体分析自己感兴趣的相关内容。

Linux 在不断地发展，本书锁定版本为 Linux2.4.16。尽管本书力图反映 Linux 内核较本质的东西，但由于我们的知识有限，对有些问题的理解难免有偏差，甚至有不少“bug”，希望读者能尽可能多地发现它，以共同对本书进行改进和完善。

在本书的编写的过程中，作者查阅了大量的资料，也阅读了大量的源代码，但本书中反映的内容也仅仅是主要内容。因为一本书的组织形成是一种线性结构，而知识本身的组织结构是一种树型结构，甚至是多线索的网状结构，因此，在本书的编写过程中，作者深感书的表现能力非常有限，一本书根本无法囊括全部。在参考书目中，我们将给出主要的参考书及主要网站的相关内容。

本书的第一版是《Linux 操作系统内核分析》该书曾被中科院指定为考博参考书，在第一版的编写过程中，康华、季进宝、陈轶飞、张波、张蕾及胡清俊参预了编写。第一般出版后得到了很多读者的充分肯定和赞扬。在本次改版的过程中，依然保留了第一版的风格，但加深了对进程管理、内存管理及文件管理的剖析。

本书在 2002 年出版后，很多读者来信给予肯定，但是因为针对的是 2.4 版内核，出版社不再给予出版。应不少读者的要求，本书的内容放在内核之旅网站，欢迎读者阅读并讨论。

作者　　陈莉君

1.1 GNU 与 Linux 的成长

GNU 是自由软件之父 Richard Stallman 在 1984 年组织开发的一个完全基于自由软件的软件体系，与此相应的有一分通用公共许可证(General Public License，简称 GPL)。Linux 以及与她有关的大量软件是在 GPL 的推动下开发和发布的。

自由软件之父 Stallman 像一个神态庄严的传教士一样喋喋不休地到处传播自由软件的福音，阐述他创立 GNU 的梦想：“自由的思想，而不是免费的午餐”。这位自由软件的“顶级神甫”为自己的梦想付出了大半生的努力，他不但自己创作了许多自由软件如 GCC 和 GDB，在他的倡导下，目前人们熟悉的一些软件如 BIND、Perl、Apache、TCP/IP 等都成了自由软件的经典之作。

如果说 Stallman 创立并推动了自由软件的发展，那么，Linus 毫不犹豫奉献给 GNU 的 Linux，则把自由软件的发展带入到一个全新的境界。

实际上，Linus 是一个理想主义者，但他又非常脚踏实际。当 Linux 的第一个“产品”版 Linux1.0 问世的时候，是按完全自由扩散版权进行扩散的。他要求 Linux 内核的所有源代码必须公开，而且任何人均不得从 Linux 交易中获利。他这种纯粹的自由软件的理想实际上妨碍了 Linux 的扩散和发展，因为这限制了 Linux 以磁盘拷贝或者 CD-ROM 等媒体形式发行的可能，也限制了一些商业公司参与 Linux 的进一步开发并提供技术支持的良好愿望。于是 Linus 决定转向 GPL 版权，这一版权除了规定自由软件的各项许可权之外，还允许用户出售自己的程序拷贝。

这一版权上的转变对 Linux 的进一步发展可谓至关重要。从此以后，便有很多家技术力量雄厚又善于市场运做的商业软件公司，加入到了原先完全由业余爱好者和网络黑客所参与的这场自由软件运动，开发出了多种 Linux 的发行版本，磨光了纯粹自由软件许多不平的棱角，增加了更易于用户使用的图形用户界面和众多的软件开发工具，这极大地拓展了 Linux 的全球用户基础。

Linux 内核的功能以及它和 GPL 的结合，使许多软件开发人员相信这是有前途的项目，开始参加内核的开发工作。并将 GNU 项目的 C 库、gcc、Emacs、bash 等很快移植到 Linux 内核上来。可以说，Linux 项目一开始就和 GNU 项目紧密结合在一起，系统的许多重要组成部分直接来自 GNU 项目。Linux 操作系统的另一些重要组成部分则来自加利福尼亚大学 Berkeley 分校的 BSD Unix 和麻省理工学院的 X Windows 系统项目。这些都是经过长期考验的成果。

正是 Linux 内核与 GNU 项目、BSD Unix 以及 MIT 的 X11 的结合，才使整个 Linux 操作系统得以很快形成，而且建立在稳固的基础上。

当 Linux 走向成熟时，一些人开始建立软件包来简化新用户安装和使用 Linux。这些软件包称为 Linux 发布或 Linux 发行版本。发行 Linux 不是某个个人或组织的事。任何人都可以将 Linux 内核和操作系统其它组成部分组合在一起进行分布。在早期众多的 Linux 发行版本中，最有影响的要数 Slackware 发布。当时它是最容易安装的 Linux 发行版本，在推广 Linux 的应用中，起了很大的作用。Linux 文档项目（LDP）是围绕 Slackware 发布写成的。目前，Red Hat 发行版本的安装更容易，应用软件更多，已成为最流行的 Linux 发行版本；而 Caldera 则致力于 Linux 的商业应用，它的发展速度也很快。这两个发行版本也有相应的成套资料。在中文的 Linux 发行版本方面，国内已经有众多的 Linux 厂商，如红旗 Linux，BluePoint Linux，中软 Linux 等。每种发行版本有各自的优点和弱点，但它们使用的**内核**和开发工具则是一致的。

2.1 i386 的寄存器

80386 作为 80X86 系列中的一员，必须保证向后兼容，也就是说，既要支持 16 位的处理器，也要支持 32 位的处理器。在 8086 中，所有的寄存器都是 16 位的，下面我们来看一下 80386 中寄存器有何变化：

- 把 16 位的通用寄存器、标志寄存器以及指令指针寄存器扩充为 32 位的寄存器
- 段寄存器仍然为 16 位。
- 增加 4 个 32 位的控制寄存器
- 增加 4 个系统地址寄存器

- 增加 8 个调式寄存器
- 增加 2 个测试寄存器

3．1．1 中断向量

Intel x86 系列微机共支持 256 种向量中断，为使处理器较容易地识别每种中断源，将它们从 0 到 256 编号，即赋以一个中断类型码 n，Intel 把这个 8 位的无符号整数叫做一个向量，因此，也叫**中断向量**。所有 256 种中断可分为两大类：异常和中断。异常又分为**故障（Fault）和陷阱(Trap)**，它们的共同特点是既不使用中断控制器，又不能被屏蔽。中断又分为外部**可屏蔽中断**（INTR）和外部**非屏蔽中断**（NMI），所有 I/O 设备产生的中断请求（IRQ）均引起屏蔽中断，而紧急的事件（如硬件故障）引起的故障产生非屏蔽中断。

非屏蔽中断的向量和异常的向量是固定的，而屏蔽中断的向量可以通过对中断控制器的编程来改变。Linux 对 256 个向量的分配如下：

- 从 0~31 的向量对应于异常和非屏蔽中断。
- 从 32~47 的向量（即由 I/O 设备引起的中断）分配给屏蔽中断。
- 剩余的从 48~255 的向量用来标识软中断。Linux 只用了其中的一个（即 128 或 0x80 向量）用来实现系统调用。当用户态下的进程执行一条 int 0x80 汇编指令时，CPU 就切换到内核态，并开始执行 system_call( )内核函数。

4.2 Linux 中的进程概述

Linux 中的每个进程由一个 task_struct 数据结构来描述，在 Linux 中，**任务**(task)、和**进程**(process)是两个相同的术语，task_struct 其实就是通常所说的“进程控制块”即 PCB。task_struct 容纳了一个进程的所有信息，是系统对进程进行控制的唯一手段，也是最有效的手段。

在 Linux2.4 中，Linux 为每个新创建的进程动态地分配一个 task_struct 结构。系统所允许的最大进程数是由机器所拥有的物理内存的大小决定的，例如，在 IA32 的体系结构中，一个 512M 内存的机器，其最大进程数可以达到 32K，这是对旧内核（2.2 以前）版本的极大改进[1]。

Linux 支持多处理机（SMP），所以系统中允许有多个 CPU， Linux 作为多处理机操作系统时系统中允许的最大 CPU 个数为 32。很显然，Linux 作为单机操作系统时，系统中只有一个 CPU，本书主要讨论单处理机的情况。

和其他操作系统类似，Linux 也支持两种进程：普通进程和实时进程。实时进程具有一定程度上的紧迫性，要求对外部事件做出非常快的响应；而普通进程则没有这种限制。所以，调度程序要区分对待这两种进程，通常，实时进程要比普通进程优先运行。这两种进程的区分也反映在 task_struct 数据结构中了。

总之，包含进程所有信息的 task_struct 数据结构是比较庞大的，但是该数据结构本身并不复杂，我们将它的所有域按其功能可做如下划分：

- **进程状态（State）**
- **进程调度信息（Scheduling Information）**
- **各种标识符（Identifiers）**
- **进程通信有关信息（IPC：Inter_Process Communication）**
- **时间和定时器信息（Times and Timers）**
- **进程链接信息（Links）**
- **文件系统信息（File System）**
- **虚拟内存信息（Virtual Memory）**
- **页面管理信息（page）**
- **对称多处理器（SMP）信息**
- **和处理器相关的环境（上下文）信息（Processor Specific Context）**

---

[1] 在 Linux2.2 及以前的版本中，用一个 task 数组来管理系统中所有进程的 task_struct 结构，因此，系统中进程的最大个数受数组大小的限制。

- **其它信息**

下面我们对 task_struct 结构进行具体描述。

5.1 Linux 时间系统

计算机是以严格精确的时间进行数值运算和和数据处理的，最基本的时间单元是时钟周期，例如取指令、执行指令、存取内存等，但是我们不讨论这些纯硬件的东西，这里要讨论的是操作系统建立的时间系统，这个时间系统是整个操作系统活动的动力。

时间系统是计算机系统非常重要的组成部分，特别是对于 Unix 类分时系统尤为重要。时间系统通常又被简称为时钟，它的主要任务是维持系统时间并且防止某个进程独占 CPU 及其他资源，也就是驱动进程的调度。本节将详细讲述时钟的来源、在 Linux 中的实现及其重要作用，使读者消除对时钟的神秘感。

6.1 Linux 的内存管理概述

Linux 是为多用户多任务设计的操作系统, 所以存储资源要被多个进程有效共享；且由于程序规模的不断膨胀，要求的内存空间比从前大得多。 Linux 内存管理的设计充分利用了计算机系统所提供的虚拟存储技术，真正实现了虚拟存储器管理。

第二章介绍的 Intel386 的段机制和页机制是 Linux 实现虚拟存储管理的一种硬件平台。实际上， Linux2.0 以上的版本不仅仅可以运行在 Intel 系列个人计算机上，还可以运行在 Apple 系列、DEC Alpha 系列、MIPS 和 Motorola 68k 等系列上， 这些平台都支持虚拟存储器管理，我们之所以选择 Intel386，是因为它具有代表性和普遍性。

Linux 的内存管理主要体现在对虚拟内存的管理。我们可以把 Linux 虚拟内存管理功能概括为以下几点：

- 大地址空间
- 进程保护
- 内存映射
- 公平的物理内存分配
- 共享虚拟内存

关于这些功能的实现，我们将会陆续介绍。

7.1 管道

在进程之间通信的最简单的方法是通过一个文件，其中有一个进程写文件，而另一个进程从文件中读，这种方法比较简单，其优点体现在：

- 只要进程对该文件具有访问权限，那么，两个进程间就可以进行通信。
- 进程之间传递的数据量可以非常大。

尽管如此，使用文件进行进程间通信也有两大缺点：

- 空间的浪费。写进程只有确保把新数据加到文件的尾部，才能使读进程读到数据，对长时间存在的进程来说，这就可能使文件变得非常大。
- 时间的浪费。如果读进程读数据比写进程写数据快，那么，就可能出现读进程不断地读文件尾部，使读进程做很多无用功。

要克服以上缺点而又使进程间的通信相对简单，管道是一种较好的选择。

所谓管道，是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称 pipe 文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，可从管道中接收数据。由于发送进程和接收进程是利用管道进行通信的，故又称管道通信。这种方式首创于 Unix 系统，因它能传送大量的数据，且很有效，故很多操作系统都引入了这种通信方式，Linux 也不例外。

为了协调双方的通信，管道通信机制必须提供以下三方面的协调能力：

• 互斥。当一个进程正在对 pipe 进行读/写操作时，另一个进程必须等待。

• 同步。当写(输入)进程把一定数量(如 4KB)数据写入 pipe 后，便去睡眠等待，直到读(输出)进程取走数据后，再把它唤醒。当读进程读到一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将它唤醒。

• 对方是否存在。只有确定对方已存在时，方能进行通信。

8.2 VFS 中的数据结构

虚拟文件系统所隐含的主要思想在于引入了一个通用的文件模型，这个模型能够表示所有支持的文件系统。该模型严格遵守传统 Unix 文件系统提供的文件模型。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux 的编码并未采用面向对象的程序设计语言（比如 C++)。因此对象作为数据结构来实现：数据结构中指向函数的域就对应于对象的方法。

通用文件模型由下列对象类型组成：

- **超级块（superblock）对象**：存放系统中已安装文件系统的有关信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的**文件系统控制块**，也就是说，每个文件系统都有一个超级块对象.
- **索引节点（inode）对象**：存放关于具体文件的一般信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的**文件控制块(FCB)**，也就是说，每个文件都有一个索引节点对象。每个索引节点对象都有一个索引节点号，这个号唯一地标识某个文件系统中的指定文件。
- **目录项（dentry）对象**：存放目录项与对应文件进行链接的信息。VFS 把每个目录看作一个由若干子目录和文件组成的常规文件。例如，在查找 路径名/tmp/test 时 ， 内核为 根目录“/ ”创建一个目录项对象, 为根目录下的 tmp 项创建一个第二级目录项对象，为 /tmp 目录下的 test 项创建一个第三级目录项对象。
- **文件(file)对象**：存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内存中。

下面我们讨论超级块、索引节点、目录项及文件的数据结构，它们的共同特点有两个：

- 充分考虑到对多种具体文件系统的兼容性
- 是“虚”的，也就是说只能存在于内存

这正体现了 VFS 的特点，在下面的描述中，读者也许能体会到这一点。

9.2.4 组描述符

块组中，紧跟在超级块后面的是组描述符表，其每一项称为组描述符，是一个叫 ext2_group_desc 的数据结构，共 32 字节。它是用来描述某个块组的整体信息的。

```
struct ext2_group_desc
{
    __u32   bg_block_bitmap;      /* 组中块位图所在的块号 */
    __u32   bg_inode_bitmap;        /* 组中索引节点位图所在块的块号 */
    __u32   bg_inode_table;     /*组中索引节点表的首块号 */
    __u16   bg_free_blocks_count;   /* 组中空闲块数 */
    __u16   bg_free_inodes_count;   /* 组中空闲索引节点数 */
    __u16   bg_used_dirs_count; /*组中分配给目录的节点数 */
    __u16   bg_pad;                   /*填充，对齐到字*/
    __u32 [3]  bg_reserved;            /*用 null 填充 12 个字节*/
}
```

每个块组都有一个相应的组描述符来描述它，所有的组描述符形成一个组描述符表，组描述符表可能占多个数据块。组描述符就相当于每个块组的超级块，一旦某个组描述符遭到破坏，整个块组将无法使用，所以组描述符表也像超级块那样，在每个块组中进行备份，

以防遭到破坏。组描述符表所占的块和普通的数据块一样，在使用时被调入块高速缓存。

10.1.1 什么是模块

模块是内核的一部分（通常是设备驱动程序），但是并没有被编译到内核里面去。它们被分别编译并连接成一组目标文件，这些文件能被插入到正在运行的内核，或者从正在运行的内核中移走，进行这些操作可以使用 insmod(插入模块)或 rmmod(移走模块)命令，或者，在必要的时候，内核本身能请求内核守护进程(kerned)装入或卸下模块。这里列出在 Linux 内核源程序中所包括的一些模块：

- 文件系统: minix, xiafs, msdos, umsdos, sysv, isofs, hpfs, smbfs, ext3,nfs，proc 等
- 大多数 SCSI 驱动程序: (如： aha1542, in2000)
- 所有的 SCSI 高级驱动程序: disk, tape, cdrom, generic.
- 大多数以太网驱动程序: (非常多，不便于在这儿列出，请查看 ./Documentation/networking/net-modules.txt)
- 大多数 CD-ROM 驱动程序:

  aztcd: Aztech,Orchid,Okano,Wearnes
  cm206: Philips/LMS CM206
  gscd: Goldstar GCDR-420
  mcd, mcdx: Mitsumi LU005, FX001
  optcd: Optics Storage Dolphin 8000AT
  sjcd: Sanyo CDR-H94A
  sbpcd: Matsushita/Panasonic CR52x, CR56x, CD200, Longshine LCS-7260, TEAC CD-55A
  sonycd535: Sony CDU-531/535, CDU-510/515

- 以及很多其它模块, 诸如:

  lp: 行式打印机
  binfmt_elf: elf 装入程序
  binfmt_java: java 装入程序
  isp16: cd-rom 接口
  serial: 串口(tty)

这里要说明的是，Linux 内核中的各种文件系统及设备驱动程序，既可以被编译成可安装模块，也可以被静态地编译进内核的映象中，这取决于内核编译之前的系统配置阶段用户的选择。通常，在系统的配置阶段，系统会给出三种选择（Y/M/N），“Y”表示要某种设备或功能，并将相应的代码静态地连接在内核映像中；“M”表示将代码编译成可安装模块，“N”表示不安装这种设备。

11.1 概述

在 Linux 中输入/输出设备被分为三类：块设备，字符设备和网络设备。这种分类的使用方法，可以将控制不同输入/输出设备的驱动程序和其它操作系统软件成分分离开来。例如文件系统仅仅控制抽象的块设备，而将与设备有关的部分留给低层软件，即驱动程序。字符设备指那些无需缓冲区可以直接读写的设备，如系统的串口设备/dev/cua0 和/dev/cua1。块设备则仅能以块为单位进行读写的设备，如软盘，硬盘，光盘等，典型块的大小为 512 或 1024 字节。从名称使人想到，字符设备在单个字符的基础上接收和发送数据。为了改进传送数据的速度和效率，块设备在整个数据缓冲区填满时才一起传送数据。网络设备可以通过 BSD 套接口访问数据，关于这方面的内容我们将在第十二章中进行讨论。

在 Linux 中，对每一个设备的描述是通过主设备号和从设备号，其中主设备号描述控制这个设备的驱动程序，也就是说驱动程序和主设备号是一一对应的，从设备号是用来区分同一个驱动程序控制的不同设备。例如主 IDE 硬盘的每个分区的从设备号都不相同，/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主、从设备号将包含在系统调用中的设备特殊文件映射到设备的管理程序，以及大量系统表格中，如字

符设备表—chrdevs。 块（磁盘）设备和字符设备的设备特殊文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但 Linux 寻找和初始化网络设备时才建立这种文件

12.1 概述

Linux 优秀的网络功能和它严密科学的设计思想是分不开的。在分析 Linux 网络内容之前，我们先大体上了解一下网络部分的设计思想及其特点，这对于我们后面的分析很有帮助：

1. Linux 的网络部分沿用了传统的层次结构。网络数据从用户进程传输到网络设备要经过四个层次，如图图 12.1 所示：

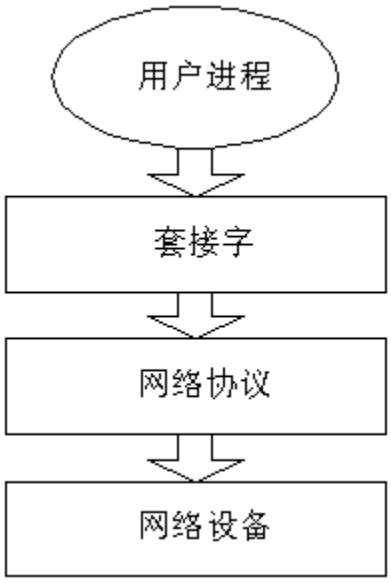

图 12.1 Linux 网络层次模型

每个层次的内部，还可以再细分为很多层次。数据的传输过程只能依照层次的划分，自顶向下进行，不能跨越其中的某个或某些层次，这就使得网络传输只能有一条而且是唯一的一条路径，这样做的目的就是为了提高整个网络的可靠性和准确性。

2．Linux 对以上网络层次的实现采用了面向对象的设计方法，层次模型中的各个层次被抽象为对象，这些对象是：

网络协议（protoal）

网络协议是一种网络语言，它规定了通信双方之间交换信息的一种规范，它是网络传输的基础。

套接字（socket）

一个套接字就是网络中的一个连接，它向用户提供了文件的 I/O，并和网络协议紧密地联系在一起，体现了网络和文件系统、进程管理之间的关系，它是网络传输的入口。

设备接口（device and interface）

网络设备接口控制着网络数据由软件——硬件——软件的过程，体现了网络和设备的关系，它是网络传输的桥梁。

网络缓冲区（network buffers）

网络中的缓冲器叫做套接字缓冲区（sk_buff）。它是一块保存网络数据的内存区域，体现了网络和内存管理之间的关系，它是网络传输的灵魂。

这四个对象之间的关系请看图 12.2：

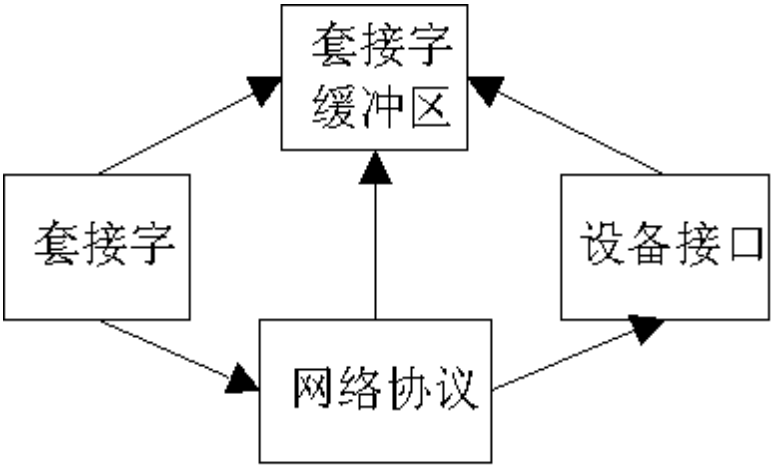

图 12.2 Linux 的网络对象及其之间的关系

从上图我们可以看出：这四个对象之间的关系是非常紧密的，其中套接字缓冲区的作用非常重要，它和其他三个对象均有关系。本章下面的部分将对这四个对象及其之间的关系做详细的介绍。

Linux 网络部分为了提高它整体上的兼容性,每一个核心对象都包含了很多种类，为了便于对网络内核的分析，每一个对象我们只选择最常用的一种详细说明，其他种类从略。

13.1 初始化流程

每一个操作系统都要有自己的初始化程序，Linux 也不例外。那么，怎样初始化？我们首先看一下初始化的流程。

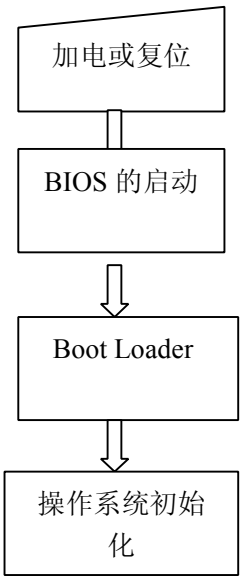

图 13.1 初始化流程

图13.1中的加电或复位这一项代表操作者按下电源开关或复位按钮那一瞬间计算机完成的工作。BIOS 的启动是紧跟其后的基于硬件的操作，它的主要作用就是完成硬件的初始化，稍后还要对 BIOS 进行详细的描述。 BIOS 启动完成后，Boot Loader 将读操作系统代码，然后由操作系统来完成初始化剩下的所有工作。

1. 驱动程序的基本函数

类别	函数名	功能	函数形成	参数	描述
驱动程序入口和出口点	module_init	驱动程序初始化入口点	module_init (x)	x 为启动时或插入模块时要运行的函数	如果在启动时就确认把这个驱动程序 插入内核或以静态形成链接，则 module_init 将其初始化例程加入到"__initcall.int"代码段，否则将用 init_module 封装其初始化例程，以便该驱动程序作为模块来使用。
	module_exit	驱动程序退出出口点	module_exit (x)	x 为驱动程序被卸载时要运行的函数	当驱动程序是一个模块，用 rmmod 卸载一个模块时 module_exit （ ） 将用 cleanup_module （ ） 封装 clean-up 代码。如果驱动程序是静态地链接进内核，则 module_exit（）函数不起任何作用。
原子和指针操作	atomic_read	读取原子变量	atomic_read (v)	v 为指向 atomic_t 类型的指针	原子地读取 v 的值。注意要保证 atomic 的有用范围只有 24 位。
	atomic_set	设置原子变量	atomic_set (v, i)	v 为指向 atomic_t 类型的指针，i 为待设置的值	原子地把 v 的值设置为 i。注意要保证 atomic 的有用范围只有 24 位。

	atomic_add	把整数增加到原子变量	void atomic_add (int i, atomic_t * v)	i 为要增加的值，v 为指向 atomic_t 类型的指针。	原子地把 i 增加到 v。注意要保证 atomic 的有用范围只有 24 位。
	atomic_sub	减原子变量的值	void atomic_sub (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针。	原子地从 v 减取 i。注意要保证 atomic 的有用范围只有 24 位。
	atomic_sub_and_test	从变量中减去值，并测试结果	int atomic_sub_and_test (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针。	原子地从 v 减取 i 的值，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位。
	atomic_inc	增加原子变量的值	void atomic_inc (atomic_t * v)	v 为指向 atomic_t 类型的指针。	原子地从 v 减取 1。注意要保证 atomic 的有用范围只有 24 位。

	atomic_dec	减取原子变量的值	void atomic_dec (atomic_t * v)	v 为指向 atomic_t 类型的指针。	原子地给 v 增加 1。注意要保证 atomic 的有用范围只有 24 位。
	atomic_dec_and_test	减少和测试	int atomic_dec_and_test (atomic_t * v)	v 为指向 atomic_t 类型的指针。	原子地给 v 减取 1，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位。
	atomic_inc_and_test	增加和测试	int atomic_ inc _and_test (atomic_t * v)	v 为指向 atomic_t 类型的指针。	原子地给 v 增加 1，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位。
	atomic_add_negative	如果结果为负数，增加并测试	int atomic_add_negative (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针。	原子地给 v 增加 i，如果结果为负数，则返回真，如果结果大于等于 0，则返回假。注意要保证 atomic 的有用范围只有 24 位。
	et_unaligned	从非对齐位置获取值	get_unaligned (ptr)	ptr 指向获取的值	这个宏应该用来访问大于单个字节的值，该值所处的位置不在按字节对齐的位置，例如从非 u16 对齐的位置检索一个 u16 的值。注意，在某些体系结构中，非对齐访问要化费较高的代价。

	put_unaligned	把值放在一个非对齐位置	put_unaligned (val, ptr)	val为要放置的值，ptr指向要放置的位置	这个宏用来把大于单个字节的值放置在不按字节对齐的位置，例如把一个u16值写到一个非u16对齐的位置。注意事项同上。
延时、调度及定时器例程	schedule_timeout	睡眠到定时时间到	signed long schedule_timeout (signed long timeout)	timeout为以jiffies为单位的到期时间	使当前进程睡眠，直到所设定的时间到期。如果当前进程的状态没有进行专门的设置，则睡眠时间一到该例程就立即返回。如果当前进程的状态设置为： ① TASK_UNINTERRUPTIBLE：则睡眠到期该例程返回0, TASK_INTERRUPTIBLE：如果当前进程接收到一个信号，则该例程就返回，返回值取决于剩余到期时间。 当该例程返回时，要确保当前进程处于TASK_RUNNING状态。

2．双向循环链表的操作

函数名	功能	函数形成	参数	描述
list_add	增加一个新元素	void list_add (struct list_head * new, struct list_head * head)	new为要增加的新元素，head为增加以后的链表头	在指定的头元素后插入一个新元素，用于栈的操作。
list_add_tail	增加一个新元素	void list_add_tail (struct list_head * new, struct list_head * head);	new为要增加的新元素，head为增加以前的链表头	在指定的头元素之前插入一个新元素，用于队列的操作。
list_del	从链表中删除一个元素	void list_del (struct list_head * entry);	entry为要从链表中删除的元素	

list_del_init	从链表删除一个元素，并重新初始化链表	void list_del_init (struct list_head * entry)	entry 为要从链表中删除的元素	
list_empty	测试一个链表是否为空	int list_empty (struct list_head * head)	head 为要测试的链表	
list_splice	把两个链表合并在一起	void list_splice (struct list_head * list, struct list_head * head)	list 为新加入的链表，head 为第一个链表	
list_entry	获得链表中元素的结构	list_entry (ptr, type, member)	ptr 为指向 list_head 的指针，type 为一个结构体，而 member 为结构 type 中的一个域，其类型为 list_head。	
list_for_each	扫描链表	list_for_each (pos, head)	pos 为指向 list_head 的指针，用于循环计数，head 为链表头。	

3．基本 C 库函数

当编写驱动程序时，一般情况下不能使用 C 标准库的函数。Linux 内核也提供了与标准库函数功能相同的一些函数，但二者还是稍有差别。

类别	函数名	功能	函数形成	参数	描述
字符串转换	simple_strtol	把一个字符串转换为一个有符号长整数	long simple_strtol (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数。	

	simple_strtoll	把一个字符串转换为一个有符号长长整数	long long simple_strtoll (const char * cp, char ** endp, unsigned int base)	cp指向字符串的开始，endp为指向要分析的字符串末尾处的位置，base为要用的基数。	
	simple_strtoul	把一个字符串转换为一个无符号长整数	long long simple_strtoul (const char * cp, char ** endp, unsigned int base)	cp指向字符串的开始，endp为指向要分析的字符串末尾处的位置，base为要用的基数。	

	simple_strtoull	把一个字符串转换为一个无符号长长整数	long long simple_strtoull (const char * cp, char ** endp, unsigned int base)	cp指向字符串的开始，endp为指向要分析的字符串末尾处的位置，base为要用的基数。	
	vsnprintf	格式化一个字符串，并把它放在缓存中。	int vsnprintf (char * buf, size_t size, const char * fmt, va_list args)	buf为存放结果的缓冲区，size为缓冲区的大小，fmt为要使用的格式化字符串，args为格式化字符串的参数。	

	snprintf	格式化一个字符串，并把它放在缓存中。	int snprintf (char * buf, size_t size, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数。	
	vsprintf	格式化一个字符串，并把它放在缓存中。	int vsprintf (char * buf, const char * fmt, va_list args)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为要使用的格式化字符串，args 为格式化字符串的参数。	

	sprintf	格式化一个字符串，并把它放在缓存中。	int sprintf (char * buf, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数。	
字符串操作	strcpy	拷贝一个以 NUL 结束的字符串	char * strcpy (char * dest, const char * src)	dest 为目的字符串的位置，src 为源字符串的位置。	
	strncpy	拷贝一个定长的、以 NUL 结束的字符串	char * strncpy (char * dest, const char * src, size_t count)	dest 为目的字符串的位置，src 为源字符串的位置，count 为要拷贝的最大字节数	与用户空间的 strncpy 不同，这个函数并不用 NUL 填充缓冲区，如果与源串超过 count，则结果以非 NUL 结束

	strcat	把一个以NUL结束的字符串添加到另一个串的末尾	char * strcat (char * dest, const char * src)	dest 为要添加的字符串，src 为源字符串。	
	strncat	把一个定长的、以NUL结束的字符串添加到另一个串的末尾	char * strncat (char * dest, const char * src, size_t count)	dest 为要添加的字符串，src 为源字符串，count 为要拷贝的最大字节数	注意，与 strncpy,形成对照，strncat 正常结束。
	strchr	在一个字符串中查找第一次出现的某个字符	char * strchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符。	

	strrchr	在一个字符串中查找最后一次出现的某个字符	char * strrchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符。	
	strlen	给出一个字符串的长度	size_t strlen (const char * s)	s 为给定的字符串	
	strnlen	给出给定长度字符串的长度	size_t strnlen (const char * s, size_t count)	s 为给定的字符串	
	strpbrk	在一个字符串中查找第一次出现的一组字符	char * strpbrk (const char * cs, const char * ct)	cs 为被搜索的字符串，ct 为待搜索的一组字符	
	strtok	把一个字符串分割为子串	char * strtok (char * s, const char * ct)	s 为被搜索的字符串，ct 为待搜索的子串	注意，一般不提倡用这个函数，而应当用 strsep

	memset	用给定的值填充内存区	void * memset (void * s, int c, size_t count)	s 为指向内存区起始的指针，c 为要填充的内容，count 为内存区的大小	I/O 空间的访问不能使用 memset，而应当使用 memset_io。
	bcopy	把内存的一个区域拷贝到另一个区域	char * bcopy (const char * src, char * dest, int count)	src 为源字符串，dest 为目的字符串，而 count 为内存区的大小	注意，这个函数的功能与 memcpy 相同，这是从 BSD 遗留下来的，对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio
	memcpy	把内存的一个区域拷贝到另一个区域	void * memcpy (void * dest, const void * src, size_t count)	dest 为目的字符串，Src 为源字符串，而 count 为内存区的大小	对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio
	memmove	把内存的一个区域拷贝到另一个区域	void * memmove (void * dest, const void * src, size_t count)	dest 为目的字符串，Src 为源字符串，而 count 为内存区的大小	memcpy 和 memmove 处理重叠的区域，而该函数不处理。

	memcmp	比较内存的两个区域	int memcmp (const void * cs, const void * ct, size_t count)	cs 为一个内存区，ct 为另一个内存区，而 count 为内存区的大小	
	memscan	在一个内存区中查找一个字符	void * memscan (void * addr, int c, size_t size)	addr 为内存区，c 为要搜索的字符，而 size 为内存区的大小	返回 c 第一次出现的地址，如果没有找到 c，则向该内存区传递一个字节。
	strstr	在以 NUL 结束的串中查找第一个出现的子串	char * strstr (const char * s1, const char * s2)	s1 为被搜索的串，s2 为待搜索的串。	
	memchr	在一个内存区中查找一个字符	void * memchr (const void * s, int c, size_t n)	s 为内存区，为待搜索的字符，n 为内存的大小	返回 c 第一次出现的位置，如果没有找到 c，则返回空。
位操作	set_bit	在位图中原子地设置某一位	void set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	这个函数是原子操作，如果不需要原子操作，则调用__set_bit 函数，nr 可以任意大，位图的大小不限于一个字。

	__set_bit	在位图中设置某一位	void __set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	
	clear_bit	在位图中清某一位	void clear_bit (int nr, volatile void * addr)	nr 为要清的位，addr 为位图的起始地址	该函数是原子操作，但不具有加锁功能，如果要用于加锁目的，应当调用 smp_mb__before_clear_bit 或 smp_mb__after_clear_bit 函数，以确保任何改变在其他的处理器上是可见的。
	__change_bit	在位图中改变某一位	void __change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址。	与 change_bit 不同，该函数是非原子操作。
	change_bit	在位图中改变某一位	void change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址。	
	test_and_set_bit	设置某一位并返回该位原来的值	int test_and_set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址。	该函数是原子操作
	__test_and_set_bit	设置某一位并返回该位原来的值	int __test_and_set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址。	该函数是非原子操作，如果这个操作的两个实例发生竞争，则一个成功而另一个失败，因此应当用一个锁来保护对某一位的多个访问。

	test_and_clear_bit	清某一位，并返回原来的值	int test_and_clear_bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址。	该函数是原子操作
	__test_and_clear_bit	清某一位，并返回原来的值	int __test_and_clear_bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址。	该函数为非原子操作
	test_and_change_bit	改变某一位并返回该位的新值	int test_and_change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址。	该函数为原子操作
	test_bit	确定某位是否被设置	int test_bit (int nr, const volatile void * addr)	nr 为要测试的第几位，addr 为位图的起始地址。	
	find_first_zero_bit	在内存区中查找第一个值为 0 的位	int find_first_zero_bit (void * addr, unsigned size)	addr 为内存区的起始地址，size 为要查找的最大长度	返回第一个位为 0 的位号

	find_next_zero_bit	在内存区中查找第一个值为0的位	int find_next_zero_bit (void * addr, int size, int offset)	addr 为内存区的起始地址，size 为要查找的最大长度，offset 开始搜索的起始位号。	
	ffz	在字中查找第一个0	unsigned long ffz (unsigned long word);	word 为要搜索的字。	
	ffs	查找第一个已设置的位	int ffs (int x)	x 为要搜索的字。	这个函数的定义方式与 Libc 中的一样。
	hweight32	返回一个N位字的加权平衡值	hweight32 (x)	x 为要加权的字	一个数的加权平衡是这个数所有位的总和。

4．Linux 内存管理中 Slab 缓冲区

函数名	功能	函数形成	参数	描述

kmem_cache_create	创建一个缓冲区	kmem_cache_t * kmem_cache_create (const char * name, size_t size, size_t offset, unsigned long flags, void (*ctor) (void*, kmem_cache_t *, unsigned long), void (*dtor) (void*, kmem_cache_t *, unsigned long));	Name 为在/proc/slabinfo 中标识这个缓冲区的名字；size 为在这个缓冲区中创建对象的大小；offset 为页中的位移量；flags 为 Slab 标志；ctor 和 dtor 分别为构造和析构对象的函数。	成功则返回指向所创建缓冲区的指针，失败则返回空。不能在一个中断内调用该函数，但该函数的执行过程可以被中断。当通过该缓冲区分配新的页面时 ctor 运行，当页面被还回之前 dtor 运行。
kmem_cache_shrink	缩小一个缓冲区	int kmem_cache_shrink (kmem_cache_t * cachep)	Cachep 为要缩小的缓冲区	为缓冲区释放尽可能多的 Slab。为了有助于调试，返回 0 意味着释放所有的 Slab。
kmem_cache_destroy	删除一个缓冲区	int kmem_cache_destroy (kmem_cache_t * cachep);	cachep 为要删除的缓冲区	从 Slab 缓冲区删除 kmem_cache_t 对象，成功则返回 0。 这个函数应该在卸载模块时调用。调用者必须确保在 kmem_cache_destroy 执行期间没有其他对象再从该缓冲区分配内存。
kmem_cache_alloc	分配一个对象	void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);	cachep 为要删除的缓冲区，flags 请参见 kmalloc()	从这个缓冲区分配一个对象。只有当该缓冲区没有可用对象时，才用到标志 flags。

kmalloc	分配内存	void * kmalloc (size_t size, int flags)	size 为所请求内存的字节数，flags 为要分配的内存类型	kmalloc 是在内核中分配内存常用的一个函数。flags 参数的取值如下： GFP_USER – 代表用户分配内存，可以睡眠。 GFP_KERNEL – 分配内核中的内存，可以睡眠 GFP_ATOMIC – 分配但不睡眠，在中断处理程序内部使用。 另外，设置 GFP_DMA 标志表示所分配的内存必须适合 DMA，例如在 i386 平台上，就意味着必须从低 16MB 分配内存。
kmem_cache_free	释放一个对象	void kmem_cache_free (kmem_cache_t * cachep, void * objp)	cachep 为曾分配的缓冲区，objp 为曾分配的对象。	释放一个从这个缓冲区中曾分配的对象
kfree	释放以前分配的内存	void kfree (const void * objp)	objp 为由 kmalloc（）返回的指针	

5．Linux 中的 VFS

类别	函数名	功能	函数形成	参数	描述

目录项缓存	d_invalidate	使一个目录项无效	int d_invalidate (struct dentry * dentry)	dentry 为要无效的目录项	如果通过这个目录项能够到达其他的目录项，就不能删除这个目录项，并返回-EBUSY。如果该函数操作成功，则返回 0。
	d_find_alias	找到索引节点一个散列的别名	struct dentry * d_find_alias (struct inode *	inode 为要讨论的索引节点	如果 inode 有一个散列的别名，就获取对这个别名，并返回它，否则返回空。注意，如果 inode 是一个目录，就只能有一个别名，如果它没有子目录，就不能进行散列。
	prune_dcache	裁减目录项缓存	void prune_dcache (int count)	count 为要释放的目录项的一个域	缩小目录项缓存。当需要更多的内存，或者仅仅需要卸载某个安装点（在这个安装点上所有的目录项都不使用），则调用该函数。 如果所有的目录项都在使用，则该函数可能失败。
	shrink_dcache_sb	为一个超级块而缩小目录项缓存	void shrink_dcache_sb (struct super_block * sb)	sb 为超级块	为一个指定的超级块缩小目录缓存。在卸载一个文件系统是调用该函数释放目录缓存。
	have_submounts	检查父目录或子目录是否包含安装点	int have_submounts (struct dentry * parent)	parent 为要检查的目录项	如果 parent 或它的子目录包含一个安装点，则该函数返回真。
	shrink_dcache_parent	裁减目录项缓存	void shrink_dcache_parent (struct dentry * parent)	parent 为要裁减目录项的父目录项	裁减目录项缓存以删除父目录项不用的子目录项。

	d_alloc	分配一个目录项	struct dentry * d_alloc (struct dentry * parent, const struct qstr * name)	parent 为要分配目录项的父目录项，name 为指向 qstr 结构的指针。	分配一个目录项。如果没有足够可用的内存，则返回 NULL。成功则返回目录项。
	d_instantiate	为一个目录项填充索引节点信息	void d_instantiate (struct dentry * entry, struct inode * inode)	entry 为要完成的目录项，inode 为这个目录项的 inode。	在目录项中填充索引节点的信息。注意，这假定 inode 的 count 域已由调用者增加，以表示 inode 正在由该目录项缓存使用。
	d_alloc_root	分配根目录项	struct dentry * d_alloc_root (struct inode * root_inode)	root_inode 为要给根分配的 inode。	为给定的 inode 分配一个根("/") 目录项，该 in ode 被实例化并返回。如果没有足够的内存或传递的 inode 参数为空，则返回空。
	d_lookup	查找一个目录项	struct dentry * d_lookup (struct dentry * parent, struct qstr * name)	parent 为父目录项，name 为要查找的目录项名字的 qstr 结构。	为 name 搜索父目录项的子目录项。如果该目录项找到，则它的引用计数加 1，并返回所找到的目录项。调用者在完成了对该目录项的使用后，必须调用 d_put 释放它。
	d_validate	验证由不安全源所提供的目录项	int d_validate (struct dentry * dentry, struct dentry * dparent)	dentry 是 dparent 有效的子目录项，dparent 是父目录项（已知有效）	一个非安全源向我们发送了一个 dentry，在这里，我们要验证它并调用 dget。该函数由 ncpfs 用在 readdir 的实现。如果 dentry 无效，则返回 0。
	d_delete	删除一个目录项	void d_delete (struct dentry * dentry)	dentry 为要删除的目录项	如果可能，把该目录项转换为一个负的目录项，否则从哈希队列中移走它以便以后的删除。

	d_rehash	给哈希表增加一个目录项	void d_rehash (struct dentry * entry)	dentry 为要增加的目录项	根据目录项的名字向哈希表增加一个目录项
	d_move	移动一个目录项	void d_move (struct dentry * dentry, struct dentry * target)	dentry 为要移动的目录项，target 为新目录项	更新目录项缓存以反映一个文件名的移动。目录项缓存中负的目录项不应当以这种方式移动。
	__d_path	返回一个目录项的路径	char * __d_path (struct dentry * dentry, struct vfsmount * vfsmnt, struct dentry * root, struct vfsmount * rootmnt, char * buffer, int buflen)	dentry 为要处理的目录项，vfsmnt 为目录项所属的安装点，root 为根目录项，rootmnt 为根目录项所属的安装点，buffer 为返回值所在处，buflen 为 buffer 的长度。	把一个目录项转化为一个字符串路径名。如果一个目录项已被删除，串“(deleted)”被追加到路径名，注意这有点含糊不清。返回值放在 buffer 中。 "buflen" 应该为页大小的整数倍。调用者应该保持 dcache_lock 锁。
	is_subdir	新目录项是否是父目录项的子目录	int is_subdir (struct dentry * new_dentry, struct dentry * old_dentry)	new_dentry 为新目录项，old_dentry 为旧目录项。	如果新目录项是父目录的子目录项（任何路径上），就返回 1，否则返回 0。
	find_inode_number	检查给定名字的目录项是否存在	ino_t find_inode_number (struct dentry * dir, struct qstr * name)	dir 为要检查的目录，name 为要查找的名字。	对于给定的名字，检查这个目录项是否存在，如果该目录项有一个 inode，则返回其索引节点号，否则返回 0。

	d_drop	删除一个目录项	void d_drop (struct dentry * dentry)	dentry 为要删除的目录项	d_drop 从父目录项哈希表中解除目录项的哈希连接，以便通过 VFS 的查找再也找不到它。注意这个函数与 d_delete 的区别，d_delete 尽可能地把目录项表记为负的，查找时会得到一个负的目录项，而 d_drop 会使查找失败。
	d_add	向哈希队列增加目录项	void d_add (struct dentry * entry, struct inode * inode)	dentry 为要增加的目录项，inode 为与目录项对应的索引节点。	该函数将把目录项加到哈希队列，并初始化 inode。这个目录项实际上已在 d_alloc（）函中得到填充。
	dget	获得目录项的一个引用	struct dentry * dget (struct dentry * dentry)	dentry 为要获得引用的目录项	给定一个目录项或空指针，如果合适就增加引用 count 的值。当一个目录项有引用时（count 不为 0），就不能删除这个目录项。引用计数为 0 的目录项永远也不会调用 dget。
	d_unhashed	检查目录项是否被散列	int d_unhashed (struct dentry * dentry)	dentry 为要检查的目录项	如果通过参数传递过来的目录项没有用哈希函数散列过，则返回真。
索引节点处理	__mark_inode_dirty	使索引节点“脏”	void __mark_inode_dirty (struct inode * inode, int flags)	inode 为要标记的索引节点，flags 为标志，应当为 I_DIRTY_SYNC	这是一个内部函数，调用者应当调用 mark_inode_dirty 或 mark_inode_dirty_sync。
	write_inode_now	向磁盘写一个索引节点	void write_inode_now (struct inode * inode, int sync)	inode 为要写到磁盘的索引节点，sync 表示是否需要同步。	如果索引节点为脏，该函数立即把它写到给磁盘。主要由 knfsd 来使用。

	clear_inode	清除一个索引节点	void clear_inode (struct inode * inode)	inode 为要写清除的索引节点	由文件系统来调用该函数，告诉我们该索引节点不再有用。
	invalidate_inodes	丢弃一个设备上的索引节点	int invalidate_inodes (struct super_block * sb);	sb 为超级块	对于给定的超级块，丢弃所有的索引节点。如果丢弃失败，说明还有索引节点处于忙状态，则返回一个非 0 值。如果丢弃成功，则超级块中所有的节点都被丢弃。
	get_empty_inode	获得一个索引节点	struct inode * get_empty_inode (void)	无	这个函数的调用发生在诸如网络层想获得一个无索引节点号的索引节点，或者文件系统分配一个新的、无填充信息的索引节点。 成功则返回一个指向 inode 的指针，失败则返回一个 NULL 指针。返回的索引节点不在任何超级块链表中。
	iunique	获得一个唯一的索引节点号	ino_t iunique (struct super_block * sb, ino_t max_reserved)	sb 为超级块，max_reserved 为最大保留索引节点号	对于给定的超级块，获得该系统上一个唯一的索引节点号。这一般用在索引节点编号不固定的文件系统中。返回的节点号大于保留的界限但是唯一。 注意，如果一个文件系统有大量的索引节点，则这个函数会很慢。
	insert_inode_hash	把索引节点插入到哈希表	void insert_inode_hash (struct inode * inode)	inode 为要插入的索引节点	把一个索引节点插入到索引节点的哈希表中，如果该节点没有超级块，则把它加到一个单独匿名的链中。

	remove_inode_hash	从哈希表中删除一个索引节点	void remove_inode_hash (struct inode * inode)	inode 为要删除的索引节点	从超级块或匿名哈希表中删除一个索引节点
	iput	释放一个索引节点	void iput (struct inode * inode)	inode 为要释放的索引节点	如果索引节点的引用计数变为 0，则释放该索引节点，并且可以撤销它。
	bmap	在一个文件中找到一个块号	int bmap (struct inode * inode, int block)	inode 为文件的索引节点，block 为要找的块。	返回设备上的块号，例如，寻找索引节点 1 的块 4，则该函数将返回相对于磁盘起始位置的盘块号。
	update_atime	更新访问时间	void update_atime (struct inode * inode)	inode 为要访问的索引节点	更新索引节点的访问时间，并把该节点标记为写回。这个函数自动处理只读文件系统、介质、“noatime”标志以及具有“noatime”标记者的索引节点。
	make_bad_inode	由于 I/O 错误把一个索引节点标记为坏	void make_bad_inode (struct inode * inode)	inode 为要标记为坏的索引节点	由于介质或远程网络失败而造成不能读一个索引节点时，该函数把该节点标记为“坏”，并引起从这点开始的 I/O 操作失败。
	is_bad_inode	是否是一个错误的 inode	int is_bad_inode (struct inode * inode)	inode 为要测试的索引节点	如果要测试的节点已标记为坏，则返回真。
注册以及超	register_filesystem	注册一个新的文件系统	int register_filesystem (struct file_system_type * fs)	fs 为指向文件系统结构的指针	把参数传递过来的文件系统加到文件系统的链表中。成功则返回 0，失败则返回一个负的错误码。

级块	unregister_filesystem	注销一个文件系统	int unregister_filesystem (struct file_system_type * fs)	fs 为指向文件系统结构的指针	把曾经注册到内核中的文件系统删除。如果没有找到个文件系统，则返回一个错误码，成功则返回 0。 这个函数所返回的 file_system_type 结构被释放或重用。
	get_super	获得一个设备的超级块	struct super_block * get_super (kdev_t dev)	dev 为要获得超级块的设备	扫描超级块链表，查找在给定设备上安装的文件系统的超级块。如果没有找到，则返回空。

6．Linux 的连网

	函数名	功能	函数形成	参数	描述
套接字缓冲区函数	skb_queue_empty	检查队列是否为空	int skb_queue_empty (struct sk_buff_head * list)	list 为队列头	如果队列为空返回真，否则返回假
	skb_get	引用缓冲区	struct sk_buff * skb_get (struct sk_buff * skb)	skb 为要引用的缓冲区	对套接字缓冲区再引用一次，返回指向缓冲区的指针
	kfree_skb	释放一个 sk_buff	void kfree_skb (struct sk_buff * skb)	sk 为要释放的缓冲区	删除对一个缓冲区的引用，如果其引用计数变为 0，则释放它
	skb_cloned	缓冲区是否是克隆的	int skb_cloned (struct sk_buff * skb)	skb 为要检查的缓冲区	如果以 skb_clone 标志来产生缓冲区，并且是缓冲区多个共享拷贝中的一个，则返回真。克隆的缓冲区具有共享数据，因此在正常情况下不必对其进行写。
	skb_shared	缓冲区是否是共享的	int skb_shared (struct sk_buff * skb)	skb 为要检查的缓冲区	如果有多于一个的人引用这个缓冲区就返回真。

	skb_share_check	检查缓冲区是否共享的，如果是就克隆它	struct sk_buff * skb_share_check (struct sk_buff * skb, int pri)	skb 为要检查的缓冲区，pri 为内存分配的优先级	如果缓冲区是共享的，就克隆这个缓冲区，并把原来缓冲区的引用计数减 1，返回新克隆的缓冲区。如果不是共享的，则返回原来的缓冲区。当从中断状态或全局锁调用该函数时，pri 必须是 GFP_ATOMIC。 内存分配失败则返回 NULL。
	skb_unshare	产生一个共享缓冲区的拷贝	struct sk_buff * skb_unshare (struct sk_buff * skb, int pri);	skb 为要检查的缓冲区，pri 为内存分配的优先级	如果套接字缓冲区是克隆的，那么这个函数就创建一个新的数据拷贝，并把原来缓冲区的引用计数减 1，返回引用计数为 1 的新拷贝。如果不是克隆的，就返回原缓冲区。当从中断状态或全局锁调用该函数时，pri 必须是 GFP_ATOMIC。 内存分配失败则返回 NULL。
	skb_queue_len	获得队列的长度	__u32 skb_queue_len (struct sk_buff_head * list_)	list_为测量的链表	返回&sk_buff 队列的指针。
	__skb_queue_head	在链表首部对一个缓冲区排队	void __skb_queue_head (struct sk_buff_head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区。	在链表首部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中。

	skb_queue_head	在链表首部对一个缓冲区排队	void skb_queue_head (struct sk_buff_head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区。	在链表首部对一个缓冲区进行排队。这个函持有锁，因此可以安全地使用。一个缓冲区不能同时放在两个链表中。
	__skb_queue_tail	在链表尾部对一个缓冲区排队	void __skb_queue_tail (struct sk_buff_head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区。	在链表尾部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中。
	skb_queue_tail	在链表尾部对一个缓冲区排队	void skb_queue_tail (struct sk_buff_head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区。	在链表尾部对一个缓冲区进行排队。这个函持有锁，因此可以安全地使用。一个缓冲区不能同时放在两个链表中。
	__skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * __skb_dequeue (struct sk_buff_head * list)	list 为要操作的队列	删除链表首部。这个函数不持有任何锁，因此使用时应当持有适当的锁。如果队链表为空则返回 NULL，成功则返回首部元素。
	skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * skb_dequeue (struct sk_buff_head * list)	list 为要操作的队列	删除链表首部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素。
	skb_insert	插入一个缓冲区	void skb_insert (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中。

	skb_append	追加一个缓冲区	void skb_append (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中。
	skb_unlink	从链表删除一个缓冲区	void **skb_unlink** (struct sk_buff * *skb*);	*Skb* 为要删除的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。
	_skb_dequeue_tail	从队尾删除	struct sk_buff * **__skb_dequeue_tail** (struct sk_buff_head * *list*)	List 为要操作的链表	从链表尾部删除。这个函数不持有任何锁，因此必须持以合适的锁来使用。如果链表为空，则返回 NULL，成功则返首部元素。
	skb_dequeue_tail	从队头删除	struct sk_buff * **skb_dequeue_tail** (struct sk_buff_head * *list*)	List 为要操作的链表	删除链表尾部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素。
	skb_put	把数据加到缓冲区	unsigned char * **skb_put** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充缓冲区所使用的数据区。如果扩充后超过缓冲区总长度，内核会产生警告。函数返回的指针指向所扩充数据的第一个字节。

	skb_push	把数据加到缓冲区的开始	unsigned char * **skb_push** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充在缓冲区的开始处缓冲区所使用的数据区。如果扩充后超过缓冲区首部空间的总长度，内核会产生警告。函数返回的指针指向所扩充数据的第一个字节。
	skb_pull	从缓冲区的开始删除数据	unsigned char * **skb_pull** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要删除的数据长度	这个函数从链表开始处删除数据，把腾出的内存归还给首部空间。把指向下一个缓冲区的指针返回。
	skb_headroom	缓冲区首部空闲空间的字节数	int **skb_headroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 首部空闲空间的字节数
	skb_tailroom	缓冲区尾部的空闲字节数	int **skb_tailroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 尾部空闲空间的字节数
	skb_reserve	调整头部的空间	void **skb_reserve** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要改变的缓冲区，len 为要删除的字节数	通过减少尾部空间，增加一个空&sk_buff 的首部空间。这仅仅适用于空缓冲区。
	skb_trim	从缓冲区删除尾部	void **skb_trim** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要改变的缓冲区，len 为新的长度	通过从尾部删除数据，剪切缓冲区的长度。如果缓冲区已经处于指定的长度，则不用改变。

	skb_orphan	使一个缓冲区成为孤儿	void **skb_orphan** (struct sk_buff * *skb*);	skb 是要成为孤儿的缓冲区	如果一个缓冲区当前有一个拥有者，我们就调用拥有者的析构函数，使 skb 没有拥有者。该缓冲区继续存在，但以前的拥有者不再对其“负责”。
	skb_queue_purge	使一个链表空	void **skb_queue_purge** (struct sk_buff_head * *list*)	list 为要腾空的链表	删除在 &sk_buff 链表上的所有缓冲区。这个函数持有链表锁，并且是原子的。
	__skb_queue_purge	使一个链表空	void **__skb_queue_purge** (struct sk_buff_head * *list*);	list 为要腾空的链表	删除在 &sk_buff 链表上的所有缓冲区。这个函数不持有链表锁，调用者必须持有相关的锁来使用它。
	dev_alloc_skb	为发送分配一个 skbuff	struct sk_buff * **dev_alloc_skb** (unsigned int *length*)	Length 为要分配的长度	分配一个新的 &sk_buff，并赋予它一个引用计数。这个缓冲区有未确定的头空间。用户应该分配自己需要的头空间。 如果没有空闲内存，则返回 NULL。尽管这个函数是分配内存，但也可以从中断来调用。

	skb_cow	当需要时拷贝skb的首部	struct sk_buff * **skb_cow** (struct sk_buff * *skb*, unsigned int *headroo*m)	Skb为要拷贝的缓冲区，headroom为需要的头空间	如果传递过来的缓冲区缺乏足够的头空间或是克隆的，则该缓冲区被拷贝，并且附加的头空间变为可用。如果没有空闲的内存，则返回空。如果缓冲区拷贝成功，则返回新的缓冲区，否则返回已存在的缓冲区。
	skb_over_panic	私有函数	void **skb_over_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb为缓冲区，sz为大小，here为地址。	用户不可调用。
	skb_under_panic	私有函数	void **skb_under_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb为缓冲区，sz为大小，here为地址。	用户不可调用。
	alloc_skb	分配一个网络缓冲区	struct sk_buff * **alloc_skb** (unsigned int *size*, int *gfp_mas*k)	size为要分配的大小，gfp_mask为分配掩码	分配一个新的&sk_buff。返回的缓冲区没有size大小的头空间和尾空间。新缓冲区的引用计数为1。返回值为一缓冲区，如果失败则返回空。从中断分配缓冲区，掩码只能使用GFP_ATOMIC的gfp_mask。
	__kfree_skb	私有函数	void **__kfree_skb** (struct sk_buff * *skb*)	skb为缓冲区	释放一个sk_buff。释放与该缓冲区相关的所有事情，清除状态。这是一个内部使用的函数，用户应当调用kfree_skb。

	skb_clone	复制一个 sk_buff	struct sk_buff * **skb_clone** (struct sk_buff * *skb*, int *gfp_mask*)	skb 为要克隆的缓冲区，gfp_mask 为分配掩码。	复制一个&sk_buff。新缓冲区不是由套接字拥有。两个拷贝共享相同的数据包而不是结构。新缓冲区的引用计数为 1。如果分配失败，函数返回 NULL，否则返回新的缓冲区。如果从中断调用这个函数，掩码只能使用 GFP_ATOMIC 的 gfp_mask。
	skb_copy	创建一个 sk_buff 的私有拷贝	struct sk_buff * **skb_copy** (const struct sk_buff * *skb*, int *gfp_mask*)	skb 为要拷贝的缓冲区，gfp_mask 为分配优先级。	既拷贝&sk_buff 也拷贝其数据。该函数用在调用者希望修改数据并需要数据的私有拷贝来进行改变时。失败返回 NULL，成功返回指向缓冲区的指针。 返回的缓冲区其引用计数为 1。如果从中断调用，则必须传递的优先级为 GFP_ATOMIC。

	skb_copy_expand	拷贝并扩展sk_buff	struct sk_buff * **skb_copy_expand** (const struct sk_buff * *skb*, int *newheadroom*, int *newtailroom*, int *gfp_mask*);	skb为要拷贝的缓冲区，newheadroom为头部的新空闲字节数，newtailroom为尾部的新空闲字节数。	既拷贝&sk_buff也拷贝其数据，同时分配额外的空间。当调用者希望修改数据并需要对私有数据进行改变，以及给新的域更多的空间时调用该函数。失败返回NULL，成功返回指向缓冲区的指针。 返回的缓冲区其引用计数为1。如果从中断调用，则必须传递的优先级为GFP_ATOMIC。

7．网络设备支持

	函数名	功能	函数形成	参数	描述
驱动程序的支持	init_etherdev	注册以太网设备	truct net_device * **init_etherdev** (struct net_device * *dev*, int *sizeof_priv*)	dev为要填充的以太网设备结构，或者要分配一个新的结构时为NULL，sizeof_priv是为这个以太网设备要分配的额外私有结构的大小。	用以太网的通用值填充这个结构的域。如果传递过来的dev为NULL，则构造一个新的结构，包括大小为sizeof_priv的私有数据区。强制将这个私有数据区在32字节（不是位）上对齐。

	dev_add_pack	增加数据包处理程序	void **dev_add_pack** (struct packet_type *pt)	pt为数据包类型	把一个协议处理程序加到网络栈，把参数传递来的&packet_type链接到内核链表中。
	dev_remove_pack	删除数据包处理程序	void **dev_remove_pack** (struct packet_type *pt)	pt为数据包类型	删除由dev_add_pack曾加到内核的协议处理程序。把&packet_type从内核链表中删除，一旦该函数返回，这个结构还能再用。
	__dev_get_by_name	根据名字找设备	struct net_device * **__dev_get_by_name** (const char * *name*);	name为要查找的名字	根据名字找到一个接口。必须在RTNL信号量或dev_base_lock锁的支持下调用。如果找到这个名字，则返回指向设备的指针，如果没有找到，则返回NULL。引用计数器并没有增加，因此调用者必须小心地持有锁。
	dev_get_by_name	根据名字找设备	struct net_device * **dev_get_by_name** (const char * *name*)	name为要查找的名字	根据名字找到一个接口。这个函数可以在任何上下文中调用并持有自己的锁。返回句柄的引用计数增加，调用者必须在其不使用时调用dev_put释放它，如果没有匹配的名字，则返回NULL。
	dev_get	测试设备是否存在	int **dev_get** (const char * *name*)	name为要测试的名字	测试名字是否存在。如果找到则返回真。为了确保在测试期间名字不被分配或删除，调用者必须持有rtnl信号量。这个函数主要用来与原来的驱动程序保持兼容。

	__dev_get_by_index	根据索引找设备	struct net_device * **__dev_get_by_index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。该设备的引用计数没有增加，因此调用者必须小心地关注加锁，调用者必须持有 RTNL 信号量或 dev_base_lock 锁。
	dev_get_by_index	根据名字找设备	struct net_device * **dev_get_by_index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。所返回设备的引用计数加 1，因此，在用户调用 dev_put 释放设备之前，返回指针是安全的。
	dev_alloc_name	为设备分配一个名字	int **dev_alloc_name** (struct net_device * *dev*, const char * *name*)	dev 为设备，name 为格式化字符串。	传递过来一个格式化字符串，例如 ltd，该函数试图找到一个合适的 id。设备较多时这是很低效的。调用者必须在分配名字和增加设备时持有 dev_base 或 rtnl 锁，以避免重复。返回所分配的单元号或出错返回一个复数。

	dev_alloc	分配一个网络设备和名字	struct net_device * **dev_alloc** (const char * *name*, int * *err*)	name 为格式化字符串,err 为指向错误的指针	传递过来一个格式化字符串，例如 ltd，函数给该名字分配一个网络设备和空间。如果没有可用内存，则返回 NULL。如果分配成功，则名字被分配，指向设备的指针被返回。如果名字分配失败，则返回 NULL，错误的原因放在 err 指向的变量中返回。调用者必须在做这一切时持有 dev_base 或 RTNL 锁，以避免重复分配名字。
	netdev_state_change	设备改变状态	void **netdev_state_change** (struct net_device * *dev*)	name 为引起通告的设备	当一个设备状态改变时调用该函数。
	dev_load	装入一个网络模块	void **dev_load** (const char * *name*)	name 为接口的名字	如果网络接口不存在，并且进程具有合适的权限，则这个函数装入该模块。如果在内核中模块的装入是不可用的，则装入操作就变为空操作。
	dev_open	为使用而准备一个接口	int **dev_open** (struct net_device * *dev*)	device 为要打开的设备	以从低层到上层的过程获得一个设备。设备的私有打开函数被调用，然后多点传送链表被装入，最后设备被移到上层，并把 NETDEV_UP 信号发送给网络设备的 notifier chain。 在一个活动的接口调用该函数只能是个空操作。失败则返回一个负的错误代码。

	dev_close	关闭一个接口	int **dev_close** (struct net_device * *dev*)	dev 为要关闭的设备	这个函数把活动的设备移到关闭状态。向网络设备的 notifier chain 发送一个 NETDEV_GOING_DOWN。然后把设备变为不活动状态，并最终向 notifier chain 发 NETDEV_DOWN 信号。
	register_netdevice_notifier	注册一个网络通告程序块	int **register_netdevice_notifier** (struct notifier_block * *nb*)	nb 为通告程序	当网络设备的事件发生时，注册一个要调用的通告程序。作为参数传递来的通告程序被连接到内核结构，在其被注销前不能重新使用它。失败则返回一个负的错误码。
	unregister_netdevice_notifier	注销一个网络通告块	int **unregister_netdevice_notifier** (struct notifier_block * *nb*)	nb 为通告程序	取消由 register_netdevice_notifer 曾注册的一个通告程序。把这个通告程序从内核结构中解除，然后还可以重新使用它。失败则返回一个负的错误码。
	dev_queue_xmit	传送一个缓冲区	int **dev_queue_xmit** (struct sk_buff * *skb*)	skb 为要传送的缓冲区	为了把缓冲区传送到一个网络设备，对缓冲区进行排队。调用者必须在调用这个函数前设置设备和优先级，并建立缓冲区。该函数也可以从中断中调用。失败返回一个负的错误码。成功并不保证帧被传送，因为也可能由于拥塞或流量调整而撤销这个帧。

	netif_rx	把缓冲区传递到网络协议层	void **netif_rx** (struct sk_buff * *skb*)	skb 为要传送的缓冲区	这个函数从设备驱动程序接受一个数据包，并为上层协议的处理对其进行排队。该函数总能执行成功。在处理期间，可能因为拥塞控制而取消这个缓冲区。
	net_call_rx_atomic		void net_call_rx_atomic(void(fn) (void))	fn 为要调用的函数	使一个函数的调用就协议层而言是原子的。
	register_gifconf	注册一个 SIOCGIF 处理程序	int **register_gifconf** (unsigned int *family*, gifconf_func_t * *gifconf*)	family 为地址组，gifconf 为处理程序	注册由地址转储例程决定的协议。当另一个处理程序替代了由参数传递过来的处理程序时，才能释放或重用后者。
	netdev_set_master	建立主/从对	int **netdev_set_master** (struct net_device * *slave*, struct net_device * *master*)	slave 为从设备，master 为主设备。	改变从设备的主设备。传递 NULL 以中断连接。调用者必须持有 RTNL 信号量。失败返回一个负错误码。成功则调整引用计数，RTM_NEWLINK 发送给路由套接字，并且返回 0。
	dev_set_allmulti	更新设备上多个计数	void **dev_set_allmulti** (struct net_device * *dev*, int *inc*)	dev 为设备，inc 为修改者。	把接收的所有多点传送帧增加到设备或从设备删除。当设备上的引用计数依然大于 0 时，接口保持着对所有接口的监听。一旦引用计数变为 0，设备回转到正常的过滤操作。负的 inc 值用来在释放所有多点传送需要的某个资源时减少其引用计数。

	dev_ioctl	网络设备的ioctl	int **dev_ioctl** (unsigned int *cm*d, void * *ar*g)	cmd为要发出的命令，arg为用户空间指向ifreq结构的指针。	向设备发布ioctl函数。这通常由用户空间的系统调用接口调用，但有时也用作其他目的。返回值为一个正数，则表示从系统调用返回，为负数，则表示出错。
	dev_new_index	分配一个索引	int **dev_new_index** (*voi*d)	无	为新的设备号返回一个合适而唯一的值。调用者必须持有rtnl信号量以确保它返回唯一的值。
	netdev_finish_unregister	完成注册	int **netdev_finish_unregister** (struct net_device * *de*v)	dev为设备。	撤销或释放一个僵死的设备。成功返回0。
	unregister_netdevice	从内核删除设备	int **unregister_netdevice** (struct net_device * *de*v)	dev为设备。	这个函数关闭设备接口并将其从内核表删除。成功返回0，失败则返回一个负数。
8390网卡	ei_open	打开/初始化网板	int **ei_open** (struct net_device * *de*v)	dev为要初始化的网络设备。	尽管很多注册的设备在每次启动时仅仅需要设置一次，但这个函数在每次打开设备时还彻底重新设置每件事。
	ei_close	关闭网络设备	int **ei_close** (struct net_device * *de*v)	dev为要关闭的网络设备。	ei_open的相反操作，在仅仅在完成“ifconfig<devname> down”时使用
	ei_interrupt	处理来自8390的中断	void **ei_interrupt** (int *ir*q, void * *dev_i*d, struct pt_regs * *reg*s);	irq为中断号，dev_id为指向net_devicede指针，regs没有使用。	处理以太网接口中断。我们通过网卡指定的函数从8390取回数据包，并在网络栈触发它们。如果需要，我们也处理传输的完成并激活传输路径。我们也根据需要更新计数器并处理其他的事务。

	ethdev_init	初始化 8390 设备结构的其余部分	int **ethdev_init** (struct net_device * *dev*)	dev 为要初始化的网络设备结构。	初始化 8390 设备结构的其余部分。不要用__init（），因为这也由基于 8390 的模块驱动程序使用。
	NS8390_init	初始化 8390 硬件	void **NS8390_init** (struct net_device * *dev*, int *startp*)	dev 为要初始化的设备，startp 为布尔值，非 0 启动芯片处理。	必须持以锁才能调用该函数

8．模块支持

	函数名	功能	函数形成	参数	描述
模块装入	request_module	试图装入一个内核模块	int **request_module** (const char * *module_name*)	module_name 为模块名	使用用户态模块装入程序装入一个模块。成功返回 0，失败返回一个负数。注意，一个成功的装入并不意味着这个模块在自己出错时就能卸载和退出。调用者必须检查他们所提出的请求是可用的，而不是盲目地调用。如果自动装入模块的功能被启用，那么这个函数就不起作用。

	call_usermodehelper	启动一个用户态的应用程序	int call_usermodehelper (char * path, char ** argv, char ** envp);	path 为应用程序的路径名，argv 为以空字符结束的参数列表，envp 为以空字符结束的环境列表。	运行用户空间的一个应用程序。该应用程序被异步启动。它作为 keventd 的子进程来运行，并具有 root 的全部权能。Keventd 在退出时默默地获得子进程。 必须从进程的上下文中调用该函数，成功返回 0，失败返回一个负数。
内部模块支持	inter_module_register	注册一组新的内部模块数据	void inter_module_register (const char * im_name, struct module * owner, const void * userdata)	im_name 为确定数据的任意字符串，必须唯一，owner 为正在注册数据的模块，通常用 THIS_MODULE，userdata 指向要注册的任意用户数据。	检查 im_name 还没有被注册，如果已注册就发出“抱怨”。对新数据，则把它追加到 inter_module_entry 链表。
	inter_module_unregister	注销一组内部模块数据	void inter_module_unregister (const char * im_name)	im_name 为确定数据的任意字符串，必须唯一。	检查 im_name 已经注册，如果没有注册就发出“抱怨”。对现有的数据，则把它从 inter_module_entry 链表中删除。

	inter_module_get	从另一模块返回任意的用户数据	const void * inter_module_get (const char * im_name)	im_name 为确定数据的任意字符串，必须唯一。	如果 im_name 还没有注册，则返回 NULL。增加模块拥有者的引用计数，如果失败则返回 NULL，否则返回用户数据。
	inter_module_get_request	内部模块自动调用 request_module	const void * inter_module_get_request (const char * im_name, const char * modname)	im__name 为确定数据的任意字符串，必须唯一；modname 为期望注册 m_name 的模块。	如果 inter_module_get 失败，调用 request_module，然后重试。
	inter_module_put	释放来自另一个模块的数据	void inter_module_put (const char * im_name)	im__name 为确定数据的任意字符串，必须唯一。	如果 im_name 还没有被注册，则“抱怨”，否则减少模块拥有者的引用计数。

9．硬件接口

	函数名	功能	函数形成	参数	描述
硬件处理	Disable_irq_nosync	不用等待使一个 irq 无效	void inline **disable_irq_nosync** (unsigned int *irq*)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效。与 disable_irq 不同，这个函数并不确保 IRQ 处理程序的现有实例在退出前已经完成。可以从 IRQ 的上下文中调用该函数。

	Disable_irq	等待完成使一个 irq 无效	void **disable_irq** (unsigned int *irq*)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效。 这个函数要等待任何挂起的处理程序在退出之前已经完成。如果你在使用这个函数，同时还持有 IRQ 处理程序可能需要的一个资源，那么，你就可能死锁。要小心地从 IRQ 的上下文中调用这个函数。
	Enable_irq	启用 irq 的	void **enable_irq** (unsigned int *irq*)	irq 为中断号	重新启用这条 IRQ 线上的中断处理。在 IRQ 的上下文中调用这个函数。
	Probe_irq_mask	扫描中断线的位图	unsigned int **probe_irq_mask** (unsigned long *val*)	val 为要考虑的中断掩码	扫描 ISA 总线的中断线，并返回活跃中断的位图。然后把中断探测的逻辑状态返回给它以前的值。

MTRR处理	Mtrr_add	增加一种内存区类型	int **mtrr_add** (unsigned long *base*, unsigned long *size*, unsigned int *type*, char *increment*)	base为内存区的物理基地址，size为内存区大小，type为MTRR期望的类型，increment为布尔值，如果为真，则增加该内存区的引用计数。	内存区类型寄存器控制着较新的Intel处理器或非Intel处理器上的高速缓存。这个函数可以增加请求MTRR的驱动程序。每个处理器实现的详细资料和硬件细节都对调用者隐藏。如果不能增加内存区，则可能因为所有的区都在使用，或CPU就根本不支持，于是返回一个负数。成功则返回一个寄存器号，但应当仅仅当作一个cookie来对待。 可用的类型为： MTRR_TYPE_UNCACHEABLE：无高速缓存 MTRR_TYPE_WRITEBACK：随时以猝发方式写回 MTRR_TYPE_WRCOMB：立即写回，但允许猝发
	Mtrr_del	删除一个内存区类型	int **mtrr_del** (int *reg*, unsigned long *base*, unsigned long *size*);	reg为由mtrr_add返回的寄存器，base为物理基地址，size为内存区大小。	如果提供了寄存器reg，则base和size都可忽略。这就是驱动程序如何调用寄存器。如果引用计数降到0，则释放该寄存器，该内存区退回到缺省状态。成功则返回寄存器，失败则返回一个负数。

PCI 支持库	pci_find_slot	从一个给定的 PCI 插槽定位 PCI	struct pci_dev * **pci_find_slot** (unsigned int *bus*, unsigned int *devfn*)	bus 为所找 PCI 设备所驻留的 PCI 总线的成员，devfn 为 PCI 插槽的成员。	给定一个 PCI 总线和插槽号，所找的 PCI 设备位于 PCI 设备的系统全局链表中。如果设备被找到，则返回一个指向它的数据结构，否则返回空。
	pci_find_device	根据 PCI 标识号开始或继续搜索一个设备	struct pci_dev * **pci_find_device** (unsigned int *vendor*, unsigned int *device*, const struct pci_dev * *from*)	vendor 为要匹配的 PCI 商家 id，或要与所有商家 id 匹配的 PCI_ANY_ID，device 为要匹配的 PCI 设备 id，或要与所有商家 id 匹配的 PCI_ANY_ID，from 为以前搜索中找到的 PCI 设备，或对于一个新的搜索来说为空。	循环搜索已知 PCI 设备的链表。如果找到与 vendor 和 device 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL。 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索。
	pci_find_class	根据类别开始或继续搜索一个设备	struct pci_dev * **pci_find_class** (unsigned int *class*, const struct pci_dev * *from*)	class：根据类别名称搜索 PCI 设备； Previous：在搜索着找到的 PCI 设备，对于新的搜索则为 NULL。	循环搜索已知 PCI 设备的链表。如果找到与 class 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL。 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索。

	pci_find_capability	查询设备的权能	int pci_find_capability (struct pci_dev * dev, int cap)	dev 为要查询的 PCI 设备，cap 为权能取值。	断定一个设备是否支持给定 PCI 权能。返回在设备 PCI 配置空间内所请求权能结构的地址，如果设备不支持这种权能，则返回 0。
	pci_find_parent_resource	返回给定区父总线的资源区	struct resource * **pci_find_parent_resource** (const struct pci_dev * *dev*, struct resource * *res*)	dev 为设备结构，该结构包括要搜索的资源，res 为要搜索的子资源记录。	对于给定设备的给定资源区，返回给定区所包含的父总线的资源区。
	pci_set_power_state	设置一个设备电源管理的状态。	int **pci_set_power_state** (struct pci_dev * *dev*, int *new_state*)	dev 为 PCI 设备，new_state 为新的电源管理声明 (0 == D0, 3 == D3 等)。	设置设备的电源管理状态。对于从状态 D3 的转换，并不像想象的那么简单，因为很多设备在唤醒期间忘了它们的配置空间。返回原先的电源状态。
	pci_save_state	保存设备在挂起之前 PCI 的配置空间	int pci_save_state (struct pci_dev * dev, u32 * buffer)	dev 为我们正在处理的 PCI 设备，buffer 为持有配置空间的上下文。	缓冲区必须足够大，以保持整个 PCI2.2 的配置空间 (>= 64 bytes)。
	pci_restore_state	恢复 PCI 设备保存的状态	int pci_restore_state (struct pci_dev * dev, u32 * buffer)	dev 为我们正在处理的 PCI 设备，buffer 为保存的配置空间。	
	pci_enable_device	驱动程序使用设备前进行初始化	int pci_enable_device (struct pci_dev * dev)	dev 为要初始化的 PCI 设备。	驱动程序使用设备前对设备进行初始化。请求低级代码启用 I/O 和内存。如果设备被挂起，则唤醒它。小心，这个函数可能失败。

	pci_disable_device	使用PCI设备之后使其无效	void pci_disable_device (struct pci_dev * dev)	dev为使无效的PCI设备	向系统发送信号，以表明系统不再使用PCI设备。这仅仅包括使PCI总线控制（如果激活）无效。
	pci_enable_wake	当设备被挂起时启用设备产生PME#	int pci_enable_wake (struct pci_dev * dev, u32 state, int enable)	dev为对其实施操作的PCI设备，state为设备的当前状态，enable为启用或禁用“产生”的标志。	当系统被挂起时，在设备的PM能力中设置位以产生PME#。如果设备没有PM能力，则返回-EIO。如果设备支持它，则返回-EINVAL，但不能产生唤醒事件。如果操作成功，则返回0。
	pci_release_regions	释放保留的PCI I/O和内存资源	void pci_release_regions (struct pci_dev * pdev)	pdev为PCI设备，其资源以前曾由pci_request_regions保留。	释放所有的PCI I/O和以前对pci_request_regions成功调用而使用的内存。只有在PCI区的所有使用都停止后才调用这个函数。
	pci_request_regions	保留PCI I/O和内存资源	int pci_request_regions (struct pci_dev * pdev, char * res_name)	pdev为PCI设备，它的资源要被保留，res_name为与资源相关的名字。	把所有与PCI设备pdev相关联的PCI区进行标记，设备pdev是由属主res_name保留的。除非这次调用成功返回，否则不要访问PCI内的任何地址。 成功返回0，出错返回EBUSY，失败时也打印警告信息。

	pci_register_driver	注册一个 PCI 设备	int pci_register_driver (struct pci_driver * drv)	drv 为要注册的驱动程序结构。	把驱动程序结构增加到已注册驱动程序链表，返回驱动程序注册期间所声明的 PCI 设备号。即使返回值为 0，驱动程序仍然是已注册。
	pci_unregister_driver	注销一个 PCI 设备	void pci_unregister_driver (struct pci_driver * drv)	drv 为要注销的驱动程序结构。	从已注册的 PCI 驱动程序链表中删除驱动程序结构，对每个驱动程序所驱动的设备，通过调用驱动程序的删除函数，给它一个清理的机会，把把这些设备标记为无驱动程序的。
	pci_insert_device	插入一个热插拔设备	void pci_insert_device (struct pci_dev * dev, struct pci_bus * bus)	dev 为要插入的设备，bus 为 PCI 总线，设备就插入到该总线。	把一个新设备插入到设备列表，并向用户空间（/sbin/hotplug）发出通知。
	pci_remove_device	删除一个热插拔设备	void pci_remove_device (struct pci_dev * dev)	dev 为要删除的设备	把一个新设备从设备列表删除，并向用户空间（/sbin/hotplug）发出通知。
	pci_dev_driver	获得一个设备的 pci_driver	struct pci_driver * pci_dev_driver (const struct pci_dev * dev)	dev 为要查询的设备	返回合适的 pci_driver 结构，如果一个设备没有注册的驱动程序，则返回 NULL。
	pci_set_master	为设备 dev 启用总线控制	void pci_set_master (struct pci_dev * dev)	dev 为要启用的设备	启用设备上的总线控制，并调用 pcibios_set_master 对特定的体系结构进行设置。

	pci_setup_device	填充一个设备的类和映射信息	int pci_setup_device (struct pci_dev * dev)	dev 为要填充的设备结构	用有关设备的商家、类型、内存及 IO 空间地址，IRO 线等初始化设备结构。在 PCI 子系统初始化时调用该函数。成功返回 0，设备类型未知返回-1

10 块设备

函数名	功能	函数形式	参数	描述	其他
blk_cleanup_queue	当不再需要一个请求队列时，释放一个 request_queue_t	void blk_cleanup_queue (request_queue_t * q);	q 为要释放的请求队列。	blk_cleanup_queue 与 blk_init_queue 是成对出现的。应该在释放请求队列时调用该函数；典型的情况是块设备正被注销时调用。该函数目前的主要任务是释放分配到队列中所有的 struct request 结构。	低级驱动程序有希望首先完成任何重要的请求…

blk_queue_head active	指明请求队列的头是否可以是活跃的。	void blk_queue_head active (request_queue_t * q, int active)	q 为这次申请的队列，active 为一个标志，表示队列头在哪儿是活跃的。	块设备驱动程序可以选定把当前活动请求留在请求队列，只有在请求完成时才移走它。 队列处理例程为安全起见把这种情况假定为缺省值，并在请求被撤销时，将不再在合并或重新组织请求时包括请求队列的头。 如果驱动程序在处理请求之前从队列移走请求，它就可以在合并和重新安排中包含队列头。这可以通过以 active 标志为 0 来调用 blk_queue_head active。 如果一个驱动程序一次处理多个请求，它必须从请求队列移走他们（或至少一个）。 当一个队列被插入，则假定该队列头为不活跃的。	

blk_queue_make_request	为设备定义一个交替的make_request 函数。	void blk_queue_make_request (request_queue_t * q, make_request_fn * mfn)	q 为受影响设备的请求队列，mfn 为交替函数。	把 buffer_heads 结构传递到设备驱动程序的常用方式为让驱动程序把请求收集到请求队列，然后让驱动程序准备就绪时把请求从那个队列移走。这种方式对很多块设备驱动程序很有效。但是，有些块设备（如虚拟设备 md 或 lvm）并不是这样，而是把请求直接传递给驱动程序，这可以通过调用 blk_queue_make_request（）函数来达到。	按以上方式操作的驱动程序必须能够恰当地处理在“高内存”的缓冲区，这是通过调用 bh_kmap 获得一个内核映射，或通过调用 create_bounce 在常规内存创建一个缓冲区。
blk_init_queue	为块设备的使用准备一个请求队列。	void blk_init_queue (request_queue_t * q, request_fn_proc * rfn)	q 为要初始化的请求队列，rfn 为处理请求所调用的函数。	如果一个块设备希望使用标准的请求处理例程，就调用该函数。当请求队列上有待处理的请求时，调用 rfn 函数。	blk_init_queue 的反操作函数为 blk_cleanup_queue ，当撤销块设备时调用后者（例如在模块卸载时）。

generic_make_request	形成块设备的 I/O 请求。	void generic_make_request (int rw, struct buffer_head * bh)	rw 为 I/O 操作的类型，即 READ、WRITE 或 READA，bh 是内存和磁盘上的缓冲区首部。	READ 和 WRITE 的含义很明确，READA 为预读。该函数不返回任何状态。请求的成功与失败，以及操作的完成是由 bh->b_end_io 递送的。	
submit_bh	类似于上一个函数	void submit_bh (int rw, struct buffer_head * bh)	rw 为 I/O 操作的类型，即 READ、WRITE 或 READA，bh 为描述 I/O 的 buffer_head	该函数与与 generic_make_request 的目的非常类似，但 submit_bh 做更多的事情。	
ll_rw_block	对块设备的低级访问	void ll_rw_block (int rw, int nr, struct buffer_head * * bhs)	rw 为 READ、WRITE 或 READA，nr 为数组中 buffer_heads 的个数，bhs 为指向 buffer_heads 的数组。	对普通文件的读/写和对块设备的读/写，都是通过调用该函数完成的。	所有的缓冲区必须是针对同一设备的。

11. USB 设备

函数名	功能	函数形成	参数	描述
usb_register	注册一个USB设备	Int usb_register (struct usb_driver * new_driver)	new_driver为驱动程序的USB操作	注册一个具有USB核心的USB驱动程序。只要增加一个新的驱动程序，就要扫描一系列独立的接口，并允许把新的驱动程序与任何可识别的设备相关联，成功则返回0，失败则返回一个负数。
usb_scan_devices	扫描所有未申明的USB接口	Usb_scan_devices (void)	无	扫描所有未申明的USB接口，并通过“probe”函数向它们提供所有已注册的USB驱动程序。这个函数将在usb_register（）调用后自动地被调用。
usb_deregister	注销一个USB驱动程序	Usb_deregister (struct usb_driver * driver)	Driver为要注销的驱动程序的USB操作。	从USB内部的驱动程序链表中取消指定的驱动程序
usb_alloc_bus	创建一个新的USB宿主控制器结构	Struct usb_bus * usb_alloc_bus (struct usb_operations * op)	op为指向struct usb_operations的指针，这是一个总线结构	创建一个USB宿主控制器总线结构，并初始化所有必要的内部对象（仅仅由USB宿主控制器使用）。如果没有可用内存，则返回NULL。
usb_free_bus	释放由总线结构所使用的内存	Void usb_free_bus (struct usb_bus * bus)	无	（仅仅由USB宿主控制器驱动程序使用）
usb_register_bus	注册具有usb核心的USB宿主控制器	Void usb_register_bus (struct usb_bus * bus);	Bus指向要注册的总线	仅仅由USB宿主控制器驱动程序使用

1．参考文献

陈莉君 Linux 操作系统内核分析 人民邮电出版社 2000.3
陈莉君等译 深入理解 Linux 内核 中国电力出版社 2001.10
陈莉君等译 Linux 内核设计与实现 机械工业出版社 2003.11
毛德操，胡希明 Linux 内核源代码情景分析 浙江大学出版社 2001.9
田云等 保护模式下 80386 及其编程 清华大学出版社 1993.12
艾德才等 80486 / 80386 系统原理与接口大全 清华大学出版社 1995.8
王鹏等译 操作系统设计与实现 电子工业出版社 1998.8
李善平等 Linux 操作系统实验教程 机械工业出版社 1999.10
ALESSANDRO RUBINI 著 LISOlEG 等译 Linux 设备驱动程序 中国电力出版社 2000.4

2．在线文档

（1） Linux 源代码的获取：

站点为 http://www.kernel.org/ ，在这里可以找到各种源代码版本及补丁。

(2) Linux 源代码超文本交叉检索工具

国外网站：http://lxr.linux.no/ ，国内镜象网站为：http://www2.linuxforum.net/lxr/http/source

（3）Linux 内核文档项目（LDP）

站点为：http://www.linuxdoc.org ，该主页中还包括了有用的链接、指南、FAQ 及 HOWTO。

（4）GCC 对 C 语言的扩展

站点为：http://developer.apple.com/techpubs/macosx/DeveloperTools/Compiler/Compiler.1d.html，该主页描述了标准 C 中所没有而 GCC 对 C 的扩展功能。

（5）Linux 的汇编

站点为：http://www.tldp.org/HOWTO/Assembly-HOWTO。

（6）Linux 开发论坛

新闻组为：comp.os.linux.development.system。专门讨论 Linux 内核的开发问题。

（7）Linux 内核邮件列表

邮件列表为：linux-kernel@vger.rutger.edu。这份邮件列表的内容非常丰富，可以从中找到 Linux 内核当前开发版的最新内容。

（8）Linux 的内存管理

网站为：http://linux-mm.org/，该主页描述有关内存管理的各种信息。

（9）Linux 虚拟文件系统

网站为：http://www.coda.cs.cmu.edu/doc/talks/linuxvfs/，其中对 Linux 的虚拟文件系统进行了描述。

（10）Linux 内核文档与源码分析

中文网站：http://www2.linuxforum.net/ker_plan/index/main.htm，这是国内 Linux 内核爱好者的论坛。

（11）Linux 内核可装入模块编程

在 http://www.lcic.org/pics/books 上是 Linux 内核模块编程指南的在线文档，适合于模块编程的初学者。

另一网站 http://blacksun.box.sk/lkm.html 上的文档 ，是给黑客及系统管理员的的权威文档。

可以说，Linux 的在线文档数以万计，在此仅列举了与 Linux 内核相关的主要网站。