第一章　概论

1.1　网络的历史

所谓计算机网络就是通过通信线路互相连接的计算机的集合。它是由计算机及外围设备，数据通讯和中断设备等构成的一个群体。目前，计算机网络大部分都是多台计算机之间能互连，通信，达到资源共享目的的网络系统，它是电子计算机及其应用技术与通讯技术日益发展且两者密切结合的产物。

计算机的通信通常有两种方式：

1．通过双绞线，同轴电缆，电话线或光缆等有形传输介质而互相实现通信；

2．通过激光，微波，地球卫星等无形介质实现无线通讯。

后者是今后发展的主要方向，因为 90 年代以后，出现了各种各样的微型电脑（笔记本电脑），无线网络在以后一定会进一步发展。

计算机网络的研制开始于 60 年代中期，至今以有 20 多年的历史。其网络技术发展及应用已经十分普及，已经渗透到各个领域，并正在日益显示着它对信息化社会所带来的影响和深远的意义。

在网络发展上，最早出现的是分布在很大的地理范围内的远程网络（Wide Area Network，WAN），例如美国国防部高级研究计划局首先研制的 ARPA 网，它从 1969 年建立，至今已经发展成为跨越几大洲的巨型网络。

70 年代中期由于微型计算机的出现和微处理器的出现，以及短程通讯技术的迅猛发展，两者相辅相成，又促进以微机为基础的各种局域网络（Local Area Network，LAN）的飞快发展，1975 年美国 Xerox 公司首先推出了 Ethernet，与此时英国剑桥大学研制成剑桥环网，他们是 LAN 的代表。

LAN 与 WAN 有所区别，其特点为：

- 有限的地理范围，通常网内的计算机限于一栋大楼，楼群或一个企业及单位。
- 较高的通讯速率，大多在每秒 1-100M bps ，而 WAN 大多在几十 Kbps。
- 通讯介质多样。
- 通常为一个部门所拥有。

特别是 80 年代以来，以微机为基础，LAN 技术有了极其迅速的发展。

90 年代计算机网络化大趋势尤为明显。具称 1978 年全世界约有 700 万人每天使用计算机，而到 1998 年上升到 5000 万人，目前全世界已经拥有超过一亿台的计算机，预计每天上机人数可达 2 亿以上。计算机的性能价格比以每年 25%的速度在提高。微机的应用已经渗透到国民经济的各个部门，乃至家庭和个人。这标志着正步入信息时代，世界范围内的社会信息数据正在每年增长 40%到 45%的年增长率在增加，这就是迫切实现网络化的动力源泉。据称，约有 65%的计算机要联网或已经联网，以求彼此通信，达到资源共享的目

标。

90 年代计算机网络化更加向深度和广度方向发展。人们要求网络传输的内容范围增加，诸如数据之外，还需传输声音，图形，图象和文字，这就是以网络为基础的多媒体技术，使网络的应用广度更加扩大，并最终为信息化社会的实现所必须的网络连接奠定基础。

当前国际 LAN 的市场上，两雄称霸，龙争虎斗的局面，将可能持续相当长一段时间。

正如大家知道的那样，80 年代后期美国 Novell 公司先是以“一花独秀，压倒群芳”之势占据了国际 LAN 市场 60%以上，一路领先，扶摇直上，尤其是 NetWare 386 V3.11 版推出后，受到普遍的注目；随后，国际上的软件公司龙头老大 Microsoft 公司先后推出了 LAN Manager V1.0（即 LAN 3+ Open）、LAN Manager V2.0 和 V2.1，后来居上，成为世界 LAN 的两大支柱之一。1992 年 10 月 Microsoft 又抢先发布了 LAN Manager V2.2，以更加领先于 Novell 的 NetWare 386 V3.11，但后者立即随后推出了 NetWare 4.0。可见“龙争虎斗”，瓜分市场的情景。

Novel LAN 采取了“将网络协议软件与网络操作系统 NetWare 紧密结合起来”的设计构想，可达到节省开销，提高运行效率之目标。Novell LAN 最大的特点是与其底层的网卡的无关性，即是说 NetWare 可以虚拟的在所有流行的 LAN 上面运行，使它成为一个理想的开发网络应用软件的平台，吸引了广大用户软件人员为之开发越来越多的网络应用软件。反过来又推动其发展，同时 Novell LAN 采取了开放协议技术（OPT），允许各种网络协议紧密结合，进而在 NetWare 386 V3.11 版中采用了 NLM 模块的组合技术，可以实现异机种联网的难题。此外，Novell LAN 不需专用服务器，占用工作站内存最小，使用方便，功能强，效率高，兼容性强，可靠性高，保密性强，容错性好。尤其在 NetWare 386 V3.11 版中实现了服务器软件的“分布式结构策略”、“横向信息共享”、“报文传送”技术、增添了“TCP/IP 栈”、实现了“SNA 协议”和“开放式数据链路接口”等一系列新技术，使 Novell LAN 更深入人心，扩大了市场。

与此同时，Microsoft 公司的 LAN Manager V2.1 和 V2.2 版除了具备 Novel LAN 一些通常的优点之外，还采用了“客户机／服务器”（Client/Server）的先进内网络体系结构，以及基于多用户，多任务并发操作系统 OS/2 作为服务器的强大功能，并以 OS/2，Unix，VMS 和 Windows NT 作为开发平台，更便于异类机种联网和异网互连。由于 LAN Manager 与 Windows 紧密结合，使它有更好的性能价格比。

在网络化技术迅速发展的今天，使用性强的 TCP/IP 协议立下了汗马功劳。起先，TCP/IP（Transmission Control Protocol/Internet Protocol）是美国国防部于 70 年代提出的重大决策之一，将网络（当时主要是中大型机连成的网络）互连起来，并按 TCP/IP 协议实现异网之间“数据通讯和资源共享”，接着美国国防部高级计划局（DARPA）于 70 年代末提出了一系列的国际互连（Internet）技术。使得在科学研究，军事和社会生活迫切需要的大范围实现资源共享和交换信息得以实现。

TCP/IP 协议的基本思想是通过网间连接器（Gateway）将各种不同的网络连接起来，在各个网络的低层协议之上构造一个虚拟的大网，是用户与其他网的通讯就像与本网的主机通讯一样方便。

国际标准化组织 (ISO) 对网络标准提出了 OSI/RM（开放系统互连七层协议的参考模型），这七层自低向高分别为物理层，数据链路层，网络层，传输层，会话层，表示层和应用层。而自此之前，DARPA 提出的 TCP/IP 仅仅提供了高四层标准，对低三层没有定义，

因此 TCP/IP 在设计时必须要解决与低层的接口问题。

1.2　OSI 模型

OSI 模型是国际互连网标准化组织（International Standards Organizations ISO）所定义的，为了使网络的各个层次有标准。这个模型一般被称为“ISO OSI（Open System Interconnection）Reference Model”。虽然迄今为止没有哪种网络结构是完全按照这种模型来实现的，但它是一个得到公认的网络体系结构的模型。

OSI 模型拥有 8 个层次：

1．Physical 物理层

它在物理线路上传输 bit 信息，处理与物理介质有关的机械的，电气的，功能的和规程的特性。它是硬件连接的接口。

2．Data Link 数据链路层

它负责实现通信信道的无差错传输，提供数据成帧，差错控制，流量控制和链路控制等功能。

3．NetWork 网络层

负责将数据正确迅速的从源点主机传送到目的点主机，其功能主要有寻址以及与相关的流量控制和拥塞控制等。

物理层，数据链路层和网络层构成了通信子网层。通讯子网层与硬件的关系密切，它为网络的上层（资源子网）提供通讯服务。

4．Transport 传输层

为上层处理过程掩盖下层结构的细节，保证把会话层的信息有效的传到另一方的会话层。

5．Session 会话层

它提供服务请求者和提供者之间的通讯，用以实现两端主机之间的会话管理，传输同步和活动管理等。

6．Presentation 表示层

它的主要功能是实现信息转换，包括信息压缩，加密，代码转换及上述操作的逆操作等。

7．Application 应用层

它为用户提供常用的应用，如电子邮件，文件传输，Web 浏览等等。

需要注意的是 OSI 模型并不是一个网络结构，因为它并没有定义每个层所拥有的具体的服务和协议，它只是告诉我们每一个层应该做什么工作。但是，ISO 为所有的层次提供了标准，每个标准都有其自己的内部标准定义。

下面我们来看看 OSI 模型的层次图（图 1-1）：

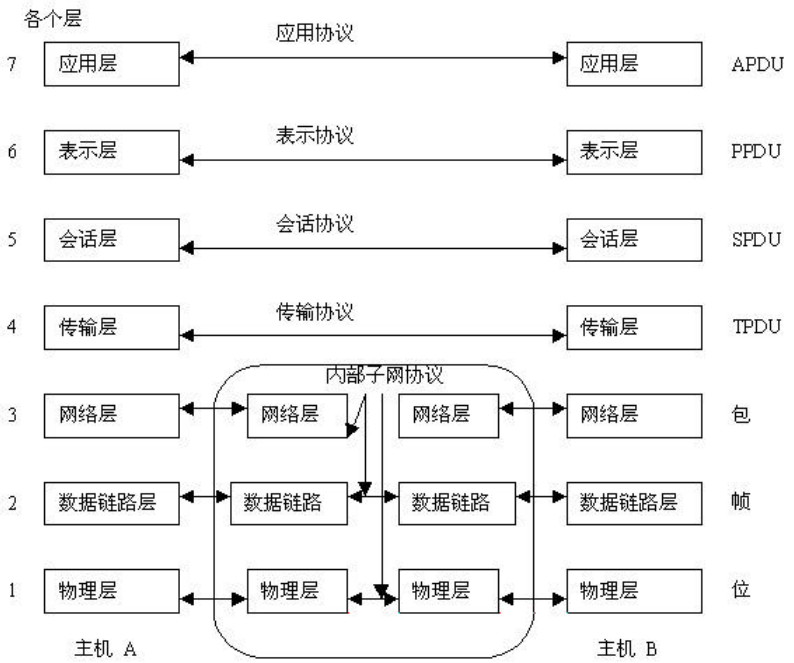

图 1-1 OSI 模型的层次图

1.3 Internet 体系模型

Internet 网是由许多子网通过网关互连组成的一个网格集合。网关是一个执行网络间转发功能的系统，被网关连接的子网有一个共同特点，它们都使用 TCP/IP 通信协议。Internet 是建立在 TCP/IP 基础上，因此采用了 TCP/IP 的网络体系结构。TCP/IP 的网络体系结构如表 1-1 所示：

表 1-1 TCP/IP 的网络体系结构

SMTP	DNS	HTTP	FTP	TELNET
TCP	UDP	NVP		
ICMP				
IP		ARP	RARP	
以太网	PDN	其他		
电话线	同轴电缆	光缆		

在 TCP/IP 网络体系结构中，第一层和第二层是 TCP/IP 的基础，其中 PDN 为公共数据网。第三层是网络层，它包含四个协议：IP，ICMP，ARP 和反向 ARP。第四层是传输层，在网络上的计算机间建立端到端的连接和服务，它包含 TCP，UDP 和 NVP 等协议。最高层包含了 FTP，TELNET，SMTP，DNS，HTTP 等协议。

网络层的主要功能由互连网协议（IP）提供，它提供端到端的分组分发，表示网络号

及主机结点的地址，数据分块和重组；并为相互独立的局域网建立互连网络的服务。

要想网络连入到 Internet，必须获得全世界统一的 IP 地址。IP 地址为 32 位，由 4 个十进制数组成，每个数值的范围为 0~255，中间用“.”隔开。每个 IP 地址定义网络 ID 和网络工作站 ID。网络 ID 标识在同一物理网络中的系统；网络工作站 ID 标识网络上的工作站，服务器或路由选择器，每个网络工作站地址对网络 ID 必须唯一。Internet IP 地址有三种基本类型：

- A 类地址

其 W 的高端位为 0，允许有 126 个 A 类地址，分配给拥有大量主机的网络。

- B 类地址

由 W.X 表示网络 ID，其高端前二位为二进制的 10，它用于分配中等规模的网络，可有 16384 个 B 类地址。

- C 类地址

其高端前三位为二进制 110，允许大约 200 万个 C 类地址，每个网络只有 254 个主机，用于小型的局域网。

其格式表示如表 1-2：

表 1-2　IP 网络地址的格式

类型	IP 地址	网络地址	主机 ID
A	W.X.Y.Z	W	X.Y.Z
B	W.X.Y.Z	W.X	Y.Z
C	W.X.Y.Z	W.X.Y	Z

1.4　客户/服务器模型

主机结构的计算机系统是企业最早采用的计算机系统，它运行 Unix 操作系统或其他多用户的操作系统。在多用户操作系统的支持下，各个用户通过终端设备来访问计算机系统，资源共享，数据的安全保密，通讯等等全部由计算机提供。系统的管理任务仅仅局限在单一计算机平台上，管理与维护比较简单。

但是，主机系统的灵活性比较差，系统的更新换代需要功能更加强大的计算机设备。系统可用性较差，如果没有采用特殊的容错设施，主机一旦出现故障，就可以引起整个系统的瘫痪。

客户机／服务器的体系结构如图 1-2 所示。

在客户机／服务器体系结构中至少有两台以上的计算机，这些计算机是由网络连接在一起，实现资源与数据共享。计算机之间通过传输介质连接起来，在它们之间形成通路。计算机之间必须按照协议互相通讯，协议（Protocol）是一组使计算机互相了解的规则与标准，是计算机通讯语言。网络中的设备只有按照规定的协议来通讯的，而让执行不同协议的计算机互相通讯也是一件复杂的事情。所以国际标准组织指定了开放系统互连（OSI）协议，描述了计算机网络各结点之间的数据传送所需求的服务框架，称为计算机网络协议参考模型。许多计算机网络厂家都以自己的技术支持某种协议，以此来开发计算机的网络

产品。

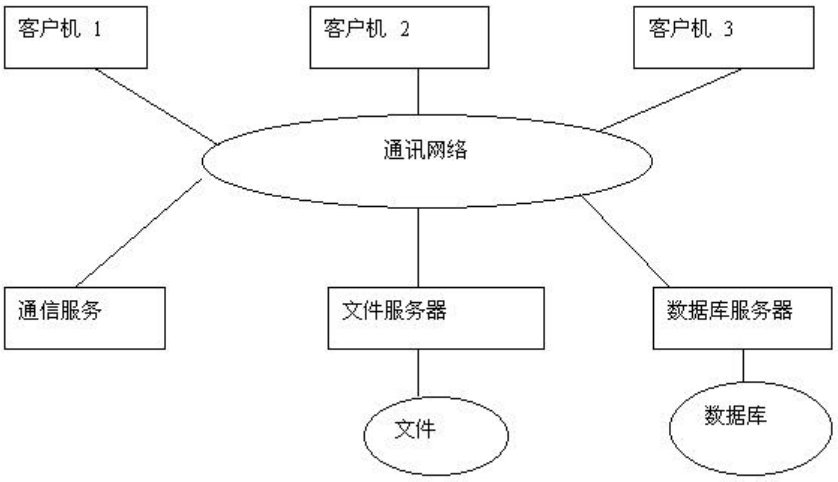

图 1- 2 客户机／服务器的体系结构

网络计算环境中的资源可以为各个结点上的计算机共享，从服务的观点上来看，网络中的计算机可扮演不同的角色：有的计算机只是执行＂服务请求＂任务，是一个客户机的角色，有的计算机用语完成指定的＂服务功能＂，是服务的提供者，起着服务器的角色。

在网络化的计算机环境中，为计算机提供网络服务与网络管理是网络操作系统（NOS）的基本功能。网络操作系统协调资源共享，对服务请求执行管理。最通用的网络服务是文件服务，打印服务，信息服务，应用服务与数据库服务等。

- 文件服务

文件服务可以有效的存储，恢复与移动数据文件，它要执行数据的读，写，访问控制以及数据的管理操作。文件服务可以帮助用户很快的将数据文件由一个地方转移到另外一个地方。网络的文件服务可实现计算机之间的文件传送，文件转储，文件更新以及文件归档等。

- 打印服务

打印服务用于控制与管理网络打印机与传真设备的网络服务，实现打印机硬件资源共享。

- 信息服务

信息服务可动态的处理网络个结点计算机用户之间，应用程序之间的通信，网络的信息服务为计算机网络目标之间提供了通信工具，并对分散的目标进行管理与操作。信息服务可以实现工作组的应用，进行工作流程管理，决定工作流程路径，转移策略，处理分布的商业事物等。信息服务可在用户之间传递信息与文件资料，可建立集成电子邮件系统等。

- 应用服务

网络应用服务用语协调网络间的硬件和软件资源，建立一个最适合的平台来运行应用软件。

- 数据库服务

网络的数据库服务体统了共享数据的存储，查询，管理和恢复等多方面的服务。在数据服务中，客户机的任务是接受用户的服务请求，并将这些请求按一定格式发送到服务器，

客户机还对服务器返回的响应数据进行处理，并按规定形式呈现给用户。数据库服务器用来分析用户请求，实施对数据库的访问与控制，并将处理结果返回给客户端。此时网络上传输的只是请求与少量的查询结果，其网络通信负担比基于文件系统的 LAN 网少的多。

在 1985 年后形成的客户机／服务器计算模式，一般是针对一个企业的全部活动，按照企业的业务模型由系统分析员建立整个企业的信息系统框架。再设计基于客户端／服务器模型的。再设计系统结构时，首先要考虑以下几点：

- 需要多少资源并将他们设计为服务器。
- 有多少客户站点，他们要完成什么子任务。
- 明确每个子业务和其他业务有什么关系，需要传递什么信息。

子业务由各站点开发的客户应用程序实现，程序开发的着眼点是如何实现本系统的子任务。客户端程序通常由应用程序员利用常规的开发工具来完成。

服务器站点只开发服务器程序，该应用程序主要考虑如何发挥本站点资源的功能，如何提供更方便的服务。这些程序一般由软硬件制造商提供开发工具并带有大量实用程序，尽量减少应用时的开发。

关于客户机／服务器系统开发变化如表 1-3：

表 1-3　客户机／服务器系统开发变化

	目前的 C/S 系统	下一步的 C/S 系统	将来的 C/S 系统
客户机	主要用 CASE 工具和程序设计语言开发	主要用软部件开发	主要用软部件来
服务器	主要用程序设计语言开发	主要用程序设计语言开发	主要用软部件或够家开发（基于分布）

1.4　UNIX 的历史

“One half of the world must sweat and goarn that
the other half may dream.”

----Henry Wadworth Longfellow

1.4.1　Unix 诞生前的故事

我们先谈谈 UNIX 的创世之初，有两点需要牢牢把握：

1．虽然 UNIX 的许多部分和其实现过程是创造性的，但其几个重要的思想都可以追溯到早期的操作系统发展。

2．如果不是 Ken Thompson，如果不是他心灵手巧，擅长摆弄当时那些身边触手可及的工具，UNIX 是不可能被写出来的。那是 1968 年，Ken Thompson 和同在贝尔实验室计算机研究小组的同事们一起进行关于 MULTICS 项目的研究工作。MULTICS 是一个误入歧途而又辉煌灿烂的计算系统。她提供了非常复杂的功能，同时消耗大量的计算资源。她太大而且太慢，研究人员们不得不一开始就缩减其初始设，进行简化实现。尽管如此，几个可工作的 MULTICS 实现还是完成了，提供了非常好的计算环境。在贝尔实验室的那个是

在一台模拟 GE635 的 GE645 上完成的。系统提供分时服务，但她主要是面向批处理的，其环境笨拙且不友好。Ken 和他的伙伴们（特别是 Dennis Ritchie 和 Joseph Ossanna）不想放弃 MULTICS 提供的舒适环境，于是他们开始向 AT&T 的管理部门游说，希望能获得一个交互式平台，诸如 DEC-10，并在其上建造他们自己的操作系统。DEC-10 是 DEC 公司（Digtal Equipment Corp.）推出的一系列机种的一种。该机有一个非常灵活的交互式分时系统。很不幸，与那个时代的许多分时平台一样，DEC-10 非常昂贵。

我们应该庆幸，Ken 的请求被拒绝了。这样的情性又发生了几次，这对 Ken 来说是太不幸了。由于 MULTICS 的失败，AT&T 管理当局被 Ken 的计划打动，他们也没有兴趣来投资另一个仅仅是在不同的硬件上设计一个看起来与 MULTICS 一样的操作系统。

与此同时，Ken 对一个成为星际旅行的游戏非常有兴趣。该程序模拟太阳系的几个主要的星体和一艘可在不同对方着陆的飞船。Ken 将其安装在 GE 系统上，GE 系统忽快忽慢的响应时间使 Ken 大为失望。而且根据后来 Dennis 的说法，在 GE 系统上运行一次该游戏需要 75 美元，太贵了。Ken 和 Dennis 后来找到了现在非常有名的‘ little-used PDP-7 sitting in a corner ”，他们用 GE 系统生成了可在该机器运行的程序代码。

1.4.2 UNIX 的诞生

有了星际旅行，Ken 有了正当的理由去实现他曾在 MULTICS 计划中设计和模拟的理论上的文件系统。很自然，一台有用的机器需要的不仅仅是一个文件系统。Ken 和他的朋友还完成了第一个命令解释器（Shell）和一些简单的文件处理工具。开始时，他们用 GE 系统来为 PDP-7 进行交叉编译。很快，他们写好了汇编器（assembler），系统已经开始自支持了。这时的系统已经有点象 UNIX 了（如用 fork()来支持多任务）。文件系统与现在的文件系统相对相似。它使用 i-节点，而且有特殊的文件类型来支持目录和设备。那台 PDP-7 可同时支持二个用户。

MULTICS 其实是代表‘ MULTiplexed Information and Computing System ”。1970 年，Brian Kernighan 开玩笑称 Ken 的系统为“ UNICS ”，代表“ UNiplexed Information and Computinig System ”，毕竟与 Ken 的系统相比，MULTICS 过于庞大了。（某些人称 MULTICS 代表‘ Many Unnecessarily Large Tables In Core Simultaneously ”而 UNIX 则是裁剪了的 MULTICS。）不久，UNICS 变成了 UNIX 而且被流传下来。

计算机研究小组并不对 PDP-7 十分满意。其一是它是借来的一台机器，更主要的是它能力有限，不太可能提供计算服务。于是小组再次提交申请，这回是一台 PDP-11/20 来研究文字处理。该申请与前一次的显著的区别是 PDP-10 的价格只是 DEC-10 的凤毛麟角。由于这次的申请十分具体——一个文字处理系统，AT&T 的管理当局宽宏大量为他们购买了 PDP-11。1970 年 UNIX 被移植到 PDP-11/20 上。那可不是一件轻而易举的事，整个系统全是用汇编写的啊！小组又将汇编写的 roff（又称为 runoff，troff 的前身）从 PDP-7 移植到 PDP-11 上。再加上一个编辑器就足以称为一个文字处理系统了。与此同时，贝尔实验室的专利局正在寻找一个文字处理系统。他们选择了计算机研究小组的基于 UNIX 系统的 PDP-11/20。贝尔实验室专利局成了 UNIX 的首家商业用户。这第一个系统有几点是很值得注意的。跑 UNIX 的 PDP-11/20 没有存储保护。它仅有一个 0.5Mb 的磁盘。它同时支持三个用户，分别完成编辑，排版，再加上计算机研究小组进行进一步的 UNIX 开发。该系统的手册被标为“ First Edition ”，日期为 1971 年 11 月。

第二版于 1972 年发行，增添了管道的功能。该版本还加上了除汇编之外的编程语言支持。特别值得一提的是 Ken 曾试图用 NB 语言来重写核心。NB 是由 B 语言（由 Ken 和 Dennis 设计）修改而来的。B 语言的前身是 BCPL，BCPL(Basic CPL)是 Martin Richards 于 1967 年在剑桥设计的。CPL（Combined Programming Language）则是 1963 年伦敦大学和剑桥大学的合作项目。而 CPL 则颇受 Algol60（1960 设计）的设计思想影响。

所有这些语言在控制结构上都和 C 语言相似，不过 B 和 BCPL 都是“无类型”的语言（尽管有点用词不当），它们只支持按“字”来访问内存。NB 演化为 C，而 C 则很快称为新的工具和应用的首选语言。

参与 MULTICS（MULTICS 用 PL/I 书写）的经验告诉 Ken 和 Dennis，用高级语言来写系统是合算的。由此，他们一直试图完成它。1973 年，C 语言加入了结构和全局变量。与此同时，Ken 和 Dennis 成功地用 C 重写了 UNIX 核心。Shell 也被重写了。这增加系统的健壮性，也使编程和调试变得容易了很多。那时，大约有 25 个 UNIX 系统。在贝尔实验室内部成立了 UNIX 系统小组来进行内部维护工作。几家大学都和贝尔实验室签定协议，获得了第四版的拷贝。协议主要是不泄露源码，在那时还没有许可证这回事。Ken 自己录制磁带，不收任何费用。第一卷磁带由在纽约的哥伦比亚大学获得。

1974 年，Ken 和 Dennis 在 Communications of the ACM 上发表了论文介绍 UNIX 系统。那时，Communications 是计算机科学的主要刊物，那篇文章在学术界引起了广泛的兴 趣。第五版正式以“仅用于教育目的”的方式向各大学提供。价格也只是名义上够磁带和手册的费用。第五版在许多大学用作教学。这时 Ken 和 Dennis 仍在积极地投入 UNIX 的研究；然而，他们继续避免提供支持的承诺。他们的小组被称为“Research”(或在贝尔实验室内部称为“1127”)。他们的机器被命名为 research。你可以通过 uucp 向他们发送 bug 报告，打电话询问他们，甚至进他们的办公室和他们一起讨论 UNIX 的问题。通常他们总能在其后的若干天内解决 bug。与 research 的在贝尔实验室的另一个小组被称为 PWB，Programmer's Workbench。由 Rudd Canaday 领导的 PWB 小组支持一个用于大型软件开发的 UNIX 版本。PWB 试图向那些并不对 UNIX 研究感兴趣的用户通过服务。他们做了大量的工作来强化了 UNIX 的核心，包括支持更多的用户。PWB 的两个非常有用的计划分别是 SCCS（源码控制系统）和 RJE（使用 UNIX 作为实验室其它主机的前段）。PWB 最终注册为 PWB/UNIX1.0。UNIX 替代了越来越多的 PDP-11 上的 DEC 公司的操作系统。尽管 UNIX 不被支持，但她的魅力远胜于她的问题而吸引了许多的用户。除了系统本身的许多优点外，源码是可以获得的，而且系统从整体上也是易于理解的。进行修改和扩充很容易。这使得 UNIX 与其同类的其它操作系统大不一样。

1975 年，第六版 UNIX 系统发行了。这是第一个在贝尔实验室外广为流传的 UNIX 系统。AT&T（通过 West Electric Co.）开始向商业和政府用户提供许可证。Mike Lesk 发行了他的可移植 C 语言库。该库提供了可在任何支持 C 语言的机器上进行 I/O 的库例程。这是用 C 书写可移植代码的重要的一步。Dennis 后来重写了该库并称其为标准 I/O 库（即所谓 stdio）。UNIX 用户们首次在纽约市进行会晤，有纽约城市大学的 Mel Ferentz 作东。当时有 40 人参加。从此以后该会议每两年举行一次，会议是极不正式的。如果你想进行演讲，你就举手，并且讲就行了。这些会议是极好的交流 bugs 报告，修改和软件的方式。每个人都带上两卷磁带参加会议，一卷是给别人的，一卷是用来录制新东西的。

1977 年，Interactive Systems 公司称为首家向最终用户出售 UNIX 的公司。UNIX 终于

成了产品。在同一时期有三个小组将 UNIX 移植到不同的机器上。Steve Johnson 和 Dennis Ritchie 将 UNIX 移植到一台 Interdata 8/32 机器上。澳大利亚的 Wollongong 大学的 Richard Miller 和同事们将 UNIX 移植到一台 Interdata 7/31 上。Tom Lyon 和其在普林斯顿（Princeton）的助手们完成了到 VM/370 的移植。每次移植都干的十分漂亮。具体点，所有这三台机器都与 PDP-11 有显著的差异。事实上，这正是问题之所在。许多操作系统都没有被设计为能在多种机器上跑。类似地，许多机器又为了某种特定的操作系统而设计。例如，如果硬件能完成进程之间的保护，操作系统利用这功能就很有意义了。

UNIX 很快被移植到其它类型的 PDP-11 上。每个都有些很有趣的功能且不断地加大了 UNIX 可支持硬件的复杂度（这些功能包括浮点处理器，可写微码，内存管理和保护，分离的命令和数据空间等等）。然而，PDP-11 系列很明显地都是基于 16 位地址空间的，所有的程序都实现于 64Kb 的大小。很滑稽的是这到促进了小程序的编写。有了支持合作进程的管道以及 exec()之后，通过它们将几个小的应用连接一个大的应用。这是 UNIX 编程的一个特点，也许我们要感谢 PDP-11 有限的地址空间。UNIX 被移植到 IBM 的 Series1 小型机上（尽管有人认为这好比是将物质与反物质结合在一起）。Series1 有与 PDP-11 相同的字大小，但它的字节是颠倒的。因此当系统初次启动时它打印出来的是“NUXI”而不是：“UNIX”。从那时起，“NUXI”问题就成了字节顺序问题的代名词。

1977 年，加利福尼亚伯克利分校（the University of California, Berkeley）的计算机科学系开始发行他们的 Pascal 解释器。其中还包括了一些新的设备驱动程序，对核心的修改，ex 编辑器，和一个比 V6 的 Shell 更好用的 Shell（“Pascal Shell”）。这就是所谓的 1BSD（1st Berkeley Software Distribution）。

1.4.3 1979 – UNIX 第七版

1979 年 UNIX 的第七版发行了。Version 7 包括了一个完整的 K&R C 编译器，它首次包括了强制类型转换，联合和类型定义。系统还提供了一个更为复杂的 Shell（称为“sh”或“Bourne shell”，取自它的作者之一，Stephen Bourne）。系统支持更大的文件。由于不懈的努力移植的结果，核心更加鲁棒，系统有了更多的外设驱动程序。

第七版的程序员手册以达到了大约 400 页（仍然可以很合适地装在一卷里）。UNIX 的其它读物则成为了第二和第三卷，大约各有 400 页。

在贝尔实验室，John Reiser 和 Tom London 将 V7 UNIX 移植到了 VAX 机上。这次移植称为 UNIX32V。在某种程度上，VAX 是一个大一点的 PDP-11，按这样的理解移植工作相对容易些。为了让 UNIX 快速移植和跑得快点，VAX 上的特殊硬件功能（换页）被忽略了。虽然如此，由于 VAX 比 PDP-11 有了相当大的地址空间（4Gb），不带换页功能的 UNIX 仍旧在实验室里广为流传，且用了好一段时间。伯克利也获得了该版本并作为进一步研究的基础。

Whitesmith 是第一个商业 C 编译器供应商。不幸的是由于在许可证问题上不够明确，C 编译器的库函数不得不故意使用不兼容的函数名和参数规范。之后，C 语言的用户接口（函数名）被裁决为不能拥有版权，现在 Whitesmith 的 C 与 UNIX 兼容了。

1.4.4　UNIX 仅仅是历史吗?

UNIX 仅仅是历史吗? 不，UNIX 就在这。IDC（International Data Corporation）报导，1985 年 UNIX 的市场大约价值$3.6 billion。全世界大约有 6%的预算是花在计算机上的。根据 1987 年 12 月发行的 UNIX WORLD，该年度有大约$5.5 billion 花在 UNIX 系统上，其中 10%是花在人员方面。IDC 估计该年度全世界有大约 8%的预算是用于计算机的。Novon 研究组宣称 1987 年间有大约 300,300 套 UNIX 系统出售。在使用的 UNIX 系统达 750,000 套。估计有 4.5 billion 的 UNIX 用户，而且用户花在 UNIX 上的机时高于 DOS 的。

预计 1990 年将销售的 UNIX 系统达 450,000 套，大部分是商业用途。到 1991 年, UNIX 市场将占整个计算机市场的 20%，而且还将不断地持续增涨。很清楚，UNIX 是成功的一例。Dennis 和 Ken 曾说："UNIX 的成功并不是过分依赖于新的创意，更重要的是她是从一组丰富的概念中精选并充分发掘的产物。"这可能不是人们问 UNIX 为什么如此成功所期望得到的答案。不管怎样，不断增涨的 UNIX 发行数目和 UNIX 持续的健康发展是惊人的。

1.5　Linux 的发展

在迅猛发展的国际互联网上，有这样一群人，他们是一支由编程高手，业余计算机玩家，黑客们组成的奇怪队伍，完全独立地开发出在功能上毫不逊色于微软的商业操作系统的一个全新的免费 UNIX 操作系统——Linux（发音为 Li-nucks），成为网络上一支不可小视的力量，以不到四年的微薄资格就成为微软的一个强劲对手。据很不精确的统计，全世界使用 Linux 操作系统的人已经有数百万之多，而且绝大多数是在网络上使用的。而在中国，随着 Internet 大潮的卷入，一批主要以高等院校的学生和 ISP（Internet Service Provider）的技术人员组成的 Linux 爱好者队伍也已经蓬蓬勃勃地成长起来，可以说在中国，随着网络的不断普及，免费而性能优异的 Linux 操作系统必将发挥出越来越大的作用。

Linux 是什么？按照 Linux 开发者的说法，Linux 是一个遵循 POSIX 标准的免费操作系统，具有 BSD 和 SYSV 的扩展特性（表明其在外表和性能上同常见的 UNIX 非常相象，但是所有系统核心代码已经全部被重新编写了）。它的版权所有者是芬兰籍的 Linus B. Torvalds 先生（Linus.Torvalds@Helsinki.FI）和其他开发人员，并且遵循 GPL 声明（GNU General Public License）。

Linux 可以在基于 Intel 386，486，Pentium，PentiumPro，Pentium MMX，PentiumII 型处理器以及 Cyrix，AMD 的兼容芯片（如 6x86，K6 等芯片）的个人计算机上运行，它可以将一台普通的个人电脑立刻变成一台功能强劲的 UNIX 工作站，在 Linux 上可以运行大多数 UNIX 程序：TEX，X Window 系统，GNU 的 C/C++编译器。它让用户端坐家中就可以享受 UNIX 的全部威力。如今有越来越多的商业公司采用 Linux 作为操作系统，例如科学工作者使用 Linux 来进行分布式计算，ISP 使用 Linux 配置 Intranet 服务器，电话拨号服务器等网络服务器，CERN（西欧核子中心）采用 Linux 做物理数据处理，美国 98 年 1 月最卖座的影片《泰坦尼克号》的片中计算机动画的设计工作就是在 Linux 平台下进行的。更有趣的是去年 InfoWorld 把年度最佳技术支持奖颁给了 Linux，给批评自由软件没有良好服务的人好好地上了一课。越来越多的商业软件公司宣布支持 Linux。在国外的大学中很

多教授用 Linux 来讲授操作系统原理和设计。当然对于大多数用户来说最重要的一点是，现在我们可以在自己家中的计算机上进行 UNIX 编程，享受阅读操作系统的全部源代码的乐趣了！

1.5.1 Linux 的发展历史

如果以人类的年龄来算的话，Linux 还是一个没有上学的七岁小娃娃。1991 年 8 月一位来自芬兰赫尔辛基大学的年轻人 Linus Benedict Torvalds，对外发布了一套全新的操作系统。事情的缘起是这样的：为了实习使用著名的计算机科学家 Andrew S. Tanenbaum 开发的 Minix（一套功能简单，简单易懂的 UNIX 操作系统，可以在 8086 上运行，后来也支持 80386，在一些 PC 机平台上非常流行），Linus 购买了一台 486 微机，但是他发现 Minix 的功能还很不完善，于是他决心自己写一个保护模式下的操作系统，这就是 Linux 的原型。最开始的 Linux 是用汇编语言编写的。主要工作是用来处理 80386 保护模式

1991 年 10 月 5 日，Linus 发布了 Linux 的第一个“正式”版本： 0.02 版，现在 Linus 可以运行 bash（GNU 的一个 UNIX shell 程序），GCC（GNU 的 C 编译器），它几乎还是什么事情也做不了，但是它被设计成一个黑客的操作系统，主要的注意力被集中在系统核心的开发工作上了，没有人去注意用户支持，文档工作，版本发布等等其他东西。

最开始的 Linux 版本被放置到一个 FTP 服务器上供大家自由下载，FTP 服务器的管理员认为这是 Linus 的 Minix，因而就建了一个 Linux 目录来存放这些文件，于是 Linux 这个名字就传开了，如今已经成了约定俗成的名称了。

然后这个娃娃操作系统就以两个星期出一次新的修正版本的速度迅速成长，在版本 0.03 之后 Linus 将版本号迅速提高到 0.10，这时候更多的人开始在这个系统上工作。在几次修正之后 Linus 将版本号提高到 0.95，这表明他希望这个系统迅速成为一个“正式”的操作系统，这时候是 1992 年，但是直到一年半之后，Linux 的系统核心版本仍然是 0.99.p114，已经非常接近 1.0 了。

Linux 终于在 1994 年的 3 月 14 日发布了它的第一个正式版本 1.0 版，而 Linux 的讨论区也从原来的 comp.os.minix 中独立成为 alt.os.linux，后来又更名为 comp.os.Linux。这是 USENET 上有名的投票表决之一，有好几万用户参加了投票。后来由于使用者越来越多，讨论区也越来越拥挤又不得不再细分成 comp.os.linux.*，如今已经有十几个讨论组了，这还不把专门为 Redhat Linux 和 Debian Linux 设的讨论组计算在内。这个讨论组也是 USENET 上最热闹的讨论组之一，每天都有数以万计的文章发表。

目前 Linux 已经是一个完整的类 UNIX 操作系统了。其最新的稳定核心版本号为 2.2.11。

Linux 的吉祥物，是一只可爱的小企鹅（起因是因为 Linus 是芬兰人，因而挑选企鹅作为吉祥物）。

说到这里，就不得不说一下同 Linux 密切相关的 GNU 了，如果没有 GNU，Linux 也许不会发展得这么快，可是如果没有 Linux，GNU 也不会有如今这么巨大的影响力。

1.5.2 什么叫 GNU？

GNU 就是 GNU's Not Unix 的缩写，GNU 的创始人 Stallman 认为 UNIX 虽然不是最

好的操作系统，但是至少不会太差，而他自信有能力把 UNIX 不足的地方加以改进，使它成为一个优良的操作系统，就是名为 GNU 的一个同 UNIX 兼容的操作系统，并且开发这个系统的目的就是为了让所有计算机用户都可以自由地获得这个系统。任何人都可以免费地获得这个系统的源代码，并且可以相互自由拷贝。因而在使用 GNU 软件的时候我们可以理直气壮地说我们使用的是正版软件。当然 GNU 也是有自己的版权声明的，就是它有名的 Copyleft（相对于版权的英文 Copyright），就是用户获得 GNU 软件后可以自由使用和修改，但是用户在散布 GNU 软件时，必须让下一个用户有获得源代码的权利并且必须告知他这一点。这一条看似古怪的规定是为了防止有些别有用心的人或公司将 GNU 软件稍加修改就去申请版权，说成是自己的产品。其目的就是要让 GNU 永远是免费和公开的。

GNU 是谁发起的？ GNU 是由自由软件基金会 （Free Software Foundation，FSF）的董事长 Richard M. Stallman（RMS）于 1984 年发起的，如今已经有十几年的历史了。Stallman 本来是在美国麻省理工学院的人工智能实验室从事研究工作的研究员，同时也是世界上可数的几个顶尖程序员之一，他的最著名的作品也是 GNU 的第一个软件就是 GNU Emacs，UNIX 平台上的一个编辑器。这个软件推出后受到广大 UNIX 用户 的热烈欢迎，由于它同时提供源代码，大家都热心地替它排除错误，增加功能，它的功能越来越强大，终于成为 UNIX 平台上最好的编辑器，上至 CRAY 超级计算机，下至最普遍的 PC 机，从 DOS 到 Windows，从 VMS 到 UNIX 都可以使用这个 Emacs。受到这个软件成功的鼓励，Stallman 成立了自由软件基金会，以推广 GNU 计划。基金会成立之后，主要靠一些厂家的捐献和出售 GNU 程序的使用手册，以及拷贝 GNU 软件的电脑磁带和光盘来维持，不过许多硬件厂家开始基金会提供高性能的工作站，这其中包括 HP 和 SONY，AT&T 这样的国际性大公司。

1.5.3　Linux 的特色

Linux 具有以下的特色：

1．多工系统——同时执行多个进程。

2．多人使用——同一部机器可供多人同时使用。

3．须在 386 protected mode 下执行。

4．采用保护模式的方式执行各个进程，所以个别的进程失控不会造成系统死机。

5．Linux 在磁盘上只读取程序中实际用到的部份（动态联结 dynamic linking）。

6．各程序可使用 copy-on-write pages 上的资料，意即多个程序可以使用同一块内存区。最初几个程序共用一块内存区域，但当某个程序尝试写入这段内存时，该 page（4KB）就被拷贝一份到别的地方，以後该程序的那 4KB 就指向新的 page。如此一来可增加速度并减少内存的使用。

7．Linux 可使用虚拟内存，但须在硬盘上规划一块区域作置换用的 partition。

8．Linux 符合 POSIX 定义，原代码与 System V、及一部份的 BSD 和 SVR4 完全兼容。

9．透过 iBCS2 模拟可执行大部份 SCO UNIX、SVR3、SVR4 的程序。

10．所有的原代码都是可免费获得的，包括所有的核心程序、驱动程序、发展工具程序、使用者的程序。目前尚有些商用程序提供给 Linux 的使用者使用，但并无附上原代码。

11．支持多国语言键盘且易新增。

12．多重虚拟的 consoles——可使用热键作更换。

13．支持数种常见的文件系统 minix-1、Xenix、System V filesystems, DOS，FAT, OS/2 的 HPFS（read-only）。本身支持两种 file system：EXT2 and XIAFS，且文件名称长度可至 256 个字。

14．“UMSDOS（Unix-like MSDOS）”可在 DOS partition 中安装 Linux。

15．支持的 CD-ROM 文件系统，可读取各种标准 CD-ROM 格式，如 ISO 9660。

16．TCP/IP 网络，包含 ftp ，telnet ，NFS 等。

1.5.4 硬件需求

Linux 对硬件并不挑剔，可以在很多机器上运行，只是效率可能会差很多。

1．最少的设备需求

386SX、2 MB RAM、1.44 MB or 1.2 MB 软驱、支持的 video card，以上这些仅可供你测试 Linux 是否可在此部机器上执行。若有 5 MB 至 10 MB 的硬盘空间，则可安装一些公用程序、shells、系统管理程序等。

2．较佳的设备需求

若你要去执行一些较需计算的程序，如 gcc,X,Tex 等，那则需要比 386SX 更快的 CPU，否则你就要多点耐心了。至少你将需要 4MB RAM，若要执行 X-Window 或让多人同时使用，则至少将需 8MB RAM 才足够。

假若使用较少的内存，虽然还是能执行，因它将使用虚拟内存（那需用到硬盘），但其速度之慢会让人情绪不好... 较多的内存对 DOS 而言虽无太大的帮助，但对 Linux 可就有其相当的价值的。

至於硬盘的容量需求则要看你要存储多少东西而定。一般需要 10MB 的空间来装一些公用程序、shells、系统管理程序等。一个较完备的系统则需要 Slackware，MCC，Debian 或 Linux/PRO，及其他共享软件，这些东西需要 60MB 至 200MB 的空间才够。

1.5.5 Linux 可用的软件

大部分常用的 Unix 工具和程序已经移植到 Linux 上了，包含大部分的 GNU 程序和许多 X client。其实移植这些软件到 Linux 上是很容易的事，大部分的程序原代码在 Linux 上重新编译时都不须修改或是只要修改一些即可，因为 Linux 几乎完全符合 POSIX 的标准。可惜的是目前 Linux 上供一般 user 用的套装软件并不很多，以下将列出已知可在 Linux 上使用的软件：

- 基本的 Unix 命令。ls，tr，sed，awk 等一般 Unix 都有的命令。
- 软件发展工具。gcc，gdb，make，bison，flex，perl，rcs，cvs，gprof。
- X-Window 环境。X11R5（XFree 2.1.1），X11R6（XFree 3.1）。
- 文字编辑器。GNU Emacs，Lucid Emacs，MicroEmacs，jove，epoch，elvis（GNU vi），vim，vile，joe，pico，jed。
- Shells。Bash（h-compatible），zsh（与 ksh 相容），pdksh，tcsh，csh，rc，ash。
- 通讯程序。Taylor（BNU 兼容）UUCP，kermit，szrz，minicom，pcomm，xcomm，term，Seyon。
- News 和 mail。C-news，innd，trn，nn，tin，smail，elm，mh，pine。

- 文字处理排版。Tex，groff，doc，ez。
- PostScript 软件。Ghostscript, GhostView（X-Window）。
- WWW。NCSA Mosaic，Netscape。
- GAME。Nethack，一些 Mud 和 X-Window 上的 game。
- 套装软件。AUIS，the Andrew User Interface System。

以上这些软件程序当然也都是免费的。

1.5.6 为什么选择 Linux ？

下面是一些选择 Linux 的原因：

- Linux 是“免费”的，上面又有那么多“免费”的软件，为什么不用？
- 瘟都死实在太不稳定了，受不了，换个平台吧。
- 我想学习 Unix，可是钱包里钞票不多，先从 Linux 开始吧。
- 我想学习操作系统，哪里有开放原代码的 OS？而且还要很活跃，有前途。
- 我对网络并行计算有兴趣，基于 Linux 的并行计算不但费用低廉而且功能强大有潜力，重要的是有源码。
- 我是（或想成为）一名 Hacker,Linux 当然是最好的工具之一。
- Linux 这么热，潜在的商业价值不可限量，尽早转移以便在未来有较好的一席之地。
- 惊奇地发现 Linux 性能相当的好，稳定性也很好，用它替换商业操作系统真是明智的选择。
- Oracle,Infomix,Sysbase,IBM 都支持 Linux 了，用它来做数据库平台也挺不错。
- 烦了一次又一次去买许可证（奸商经常设这样的陷阱）,Linux 遵循公共版权许可证(GPL)正合我意。
- Linux 太适合 Internet/Intranet，它本身就是通过网络来协同开发的，网络时代为什么不用 Linux？
- 采用 Linux 可以极大地降低拥有者总成本（TCO）。
- 等待商业操作系统补丁的耐心是有限度的，更受不了总被商家牵着鼻子走，开放原代码的 Linux 使我至少有一定的控制权。
- 开放原代码使我可以按照自己的需要添加或删除某些功能，用户可定制性真是太好了！！
- 利用开放原代码的 Linux 还可以来开发路由器，嵌入式系统，网络计算机，个人数字助理等等，GNU 真是巨大的知识宝库，何乐而不用？（中国的 IT 业者真该仔细考虑这个问题）
- 我崇尚自由软件的精神，自由程序员是我的梦想，愿意为之贡献自己的力量！！
- 不为什么……

1.6 Linux 和 Unix 的发展

很多年中，贝尔实验室一直是开发 Unix 的中心机构，1990 年，AT&T 更新组建了一个机构，称为 Unix 系统实验室，称为 USL，来控管这项工作，1993 年 6 月，AT&T 将 USL

卖给了 NOVELL 公司，1993 年 10 月，NOVELL 公司将“Unix”改为 X/open,它是一个国际标准化组织.

现在 Unix 有很多版本，但是它们都有两个显著的特点:多任务多用户的分时系统。多用户指在同一时刻可以支持多个用户，多任务指在同一时刻可以执行多道程序。

Unix 的一个重要分支来源于加利福尼亚大学的贝克利分校（Berkeley）。最初，Berkeley Unix 基于 AT&T Unix,但最新的版本设计的程序要比 AT&T System V 灵活的多。Berkeley Unix 的正规名称是 BSD，是 Berkeley Software Distribution 的词头缩写。

虽然 Unix 有多种版本（表 1-4），担实际上它们或基于 BSD，或基于 System V，或者基于二者之上。

表 1-4 Unix 的各种版本

Unix 的名称	公司或组织，机构名称
386BSD	internet 免费提供
AIX	IBM
A/UX	Apple
BSD	加利福尼亚大学的贝克利分校
BSD-LITE	加利福尼亚大学的贝克利分校
Goherent	BSDI
Dynix	Scquent
FreeBSD	internet 免费提供
HP-UX	HP
Hurd(GNN)	FSF
Interactive	Graphics
Linux	internet 免费提供
Mach	Carnegie-Mellon
Minix	AndyTanenbaum
MKSToolkit	MorticeKer
NetNSD	internet 免费提供
Nextstep	Next
OSF/1	DEC
SCOUnix	SarctaCruzOperation
Solaris	SunMicrosystem
SunOs	SunMicrosystem
SystemVUnix	pc 机上的各种版本
Unicos	CrayResearch
Unixware	Novell

第二章 UNIX/Linux 模型

2.1 UNIX/Linux 基本结构

图 2-1 绘出了 UNIX 系统的高层次的体系结构。图中心的硬件部分向操作系统提供基本服务。操作系统直接与硬件交互，向程序提供公共服务，并使他们同硬件特性隔离。当我们把整个系统看成层的集合时，通常将操作系统成为系统内核，或简称内核，此时强调的是它同用户程序的隔离。因为程序是不依赖于其下面的硬件的，所以，如果程序对硬件没做什么假定的话，就容易把它们在不同硬件上运行的 UNIX 系统之间迁移。比如，那些假定了机器字长的程序比起没假定机器字长的程序来就较难于搬到其它机器上。

外层的程序，诸如 shell 及编辑程序（vi），是通过引用一组明确定义的系统调用而与内核交互的。这些系统调用通知内核为调用程序做各种操作，并在内核与调用程序之间交换数据。图中出现的一些程序属于标准的系统配置，就是大家所知道的命令。但是由名为 a.out 的程序所指示的用户私用程序也可以存在于这一层。此处的 a.out 是被 C 编译程序产生的可执行文件的标准名字。其它应用程序能在较低的程序层次之上构筑而成，因此它们存在于本图的最外层。比如，标准的 C 编译程序 cc 就处在本图的最外层；它引用 C 预处理程序、两次编译程序、汇编程序及装入程序（称为连接—编译程序），这些都是彼此分开的低层程序。虽然该图对应用程序只描绘了两个级别的层次，但用户能够对层次进行扩从，直到级别的数目适合于自己的需要。确实，为 UNIX 系统所偏爱的程序设计风格鼓励把现存程序组合起来去完成一个任务。

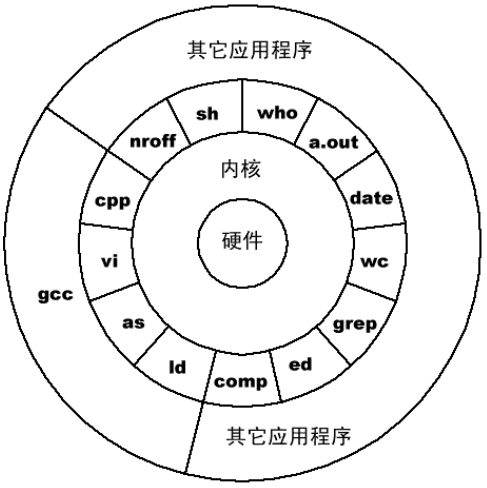

图 2- 1 UNIX 系统的高层次的体系结构

一大批提供了对系统的高层次看法的应用子程序及应用程序，诸如 shell、编辑程序、SCCS（Source Code Control System）及文档准备程序包等，都逐渐变成了“UNIX 系统”这一名称的同义语。然而，它们最终都使用由内核提供的底层服务，并通过系统调用（System Call）的集合利用这些服务。系统调用的集合及其实现系统调用的内部算法形成了内核的

主体。简言之，内核提供了 UNIX/Linux 系统全部应用程序所依赖的服务，并且内核的定义了这些服务。下面我们将进一步介绍内核，对内核的体系结构提出一个总的看法，勾画出它的基本概念和结构，这将帮助读者更好的学习以后的内容。

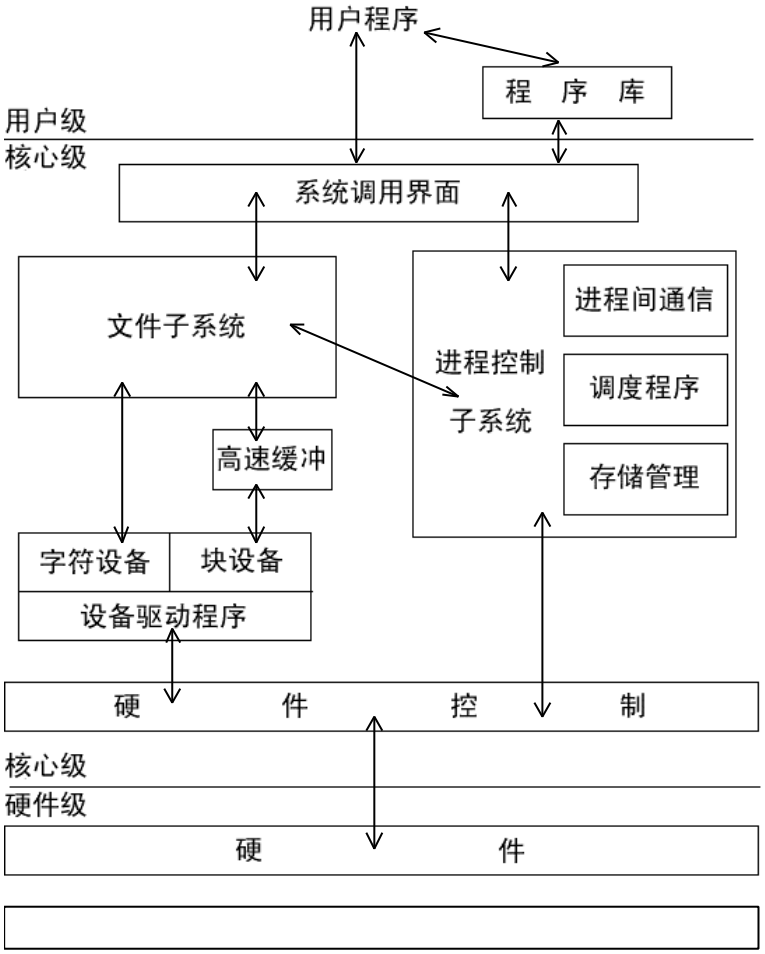

图 2-2 Unix 系统内核结构

图 2-2 给出了内核的框图，示出了各种模块及他们之间的相互关系，特别的，它示出了内核的两个主要成分：左边的文件子系统和右边的进程控制子系统。虽然，在实际上，由于某些模块同其它模块的内部操作进行交互而使内核偏离该模型，但该图仍可以作为观察内核的一个有用的逻辑观点。

在图 2-2 中我们看到了三个层次：用户、内核及硬件。系统调用与库接口体现了图 2-1 中描绘的用户程序与内核间的边界。系统调用看起来象 C 程序中普通的函数调用，而库把这些函数调用映射成进入操作系统所需要的源语。然而，汇编语言程序可以不经过系统调用库而直接引用系统调用。程序常常使用像标准 I/O 库这样一些其它的库程序以提供对系统调用的更高级的使用。由于在编译期间把这些库连接到程序上，因此，以这里的观点来说，这些库是用户程序的一部分。

图 2-2 把系统调用的集合分成与文件子系统交互作用的部分及与进程控制子系统交互作用的部分。文件子系统管理文件，其中包括分配文件空间，管理空闲空间，控制对文件的存取，以及为用户检索数据。进程通过一个特定的系统调用集合，比如通过系统调用 open,close,read,write,stat,chown 以及 chmod 等与文件子系统交互。

文件子系统使用一个缓冲机制存取文件数据，缓冲机制调节在核心与二级存储设备之

间的数据流。缓冲机制同块 I/O 设备驱动程序交互作用，以便启动往核心去的数据传送及从核心的来的数据传送。设备驱动程序是用来控制外围设备操作的核心模块。块 I/O 设备是随机存取存储设备，或者说，它们的设备驱动程序似的它们的设备驱动程序使得它们对于系统的其它部分来说好像是随机存取存储设备。例如，一个磁带驱动程序可以允许核心把一个磁带装置作为一个随机存取存储设备看待。文件子系统和可以在没有缓冲机制干预的情况下直接与“原始”I/O 设备驱动程序交互作用。原始设备，有时也被成为字符设备，包括所有非块设备的设备。

进程控制子系统负责进程同步、进程间通讯，存储管理及进程调度。当要执行一个文件而把该文件装入存储器中时，文件子系统与进程控制子系统交互：进程子系统在执行可执行文件之前，把它们读到内存中。输入输出存储管理模块控制存储分配。在任何时刻，只要系统没有足够的屋里存储供所有进程使用，核心就在内存与二级存储之间对进程进行交换，以便所有的进程都得到公平的执行机会。

调度程序模块把 CPU 分配给进程。该模块调度各进程依次运行，直到它们因等待资源而自愿放弃 CPU，或者知道它们最近一次的运行时间超出一个时间量，从而核心抢占它们。于是调度程序选择最高优先权的合格进程投入运行；当原来的进程成为最高优先权的合格进程时，还会再次投入运行。进程间通信有几种形式，从时间的异步软中断信号到进程间消息的同步传输，等等。本书中主要的讲的网络通信，也是进程间通信的一种。

最后，硬件控制负责处理中断及与及其与机器通信。象磁盘或终端这样的设备可以在一个进程正在执行时中断 CPU。如果出现这种情况，在对中断服务完毕之后核心可以恢复被中断了的进程的执行。中断不是由特殊的进程服务的，而是由核心中的特殊函数服务的。这些特殊函数是在当前运行的进程上下文中被调用的。

2.2 输入和输出

输入和输出是交互式的操作系统的一个重要的组成部分。在 UNIX/Linux 中，采用了以抽象文件为基础的输入/输出系统，减少了系统对硬件的依赖性，简化了输入/输出的操作，同时又增加了代码的灵活性。但是，由于使用了抽象的概念，所以在理解和掌握上有一定的难度，需要认真的体会。下面，我们就简要介绍一下 UNIX 的文件系统。

2.2.1 UNIX/Linux 文件系统简介

UNIX 的文件系统有如下的特点：

- 层次结构
- 对文件数据的一致对待
- 建立与删除文件的能力
- 文件的动态增长
- 文件数据的权限保护
- 把外围设备作为文件看待

文件系统被组织成树状，称为目录树。目录树有一个成为根（root）的节点（记做“/”）。文件系统结构中的每个非树节点都是文件的一个目录（directory），树的叶节点上的文件既可以是目录，也可以是正规文件（regular files），还可以是特殊设备文件（special device files）。文件名由路径名（path name）给出，路径名描述了怎样在一个文件系统树中确定一个文件的位置。路径名是一个分量名序列，各分量名之间用“/”隔开。分量是一个字符序列，它致命一个北唯一的包含在前级（目录）分量中的文件名。一个完整的路径名由一个斜杠字符开始，并且指明一个文件，这个文件可以从文件系统的根开始，沿着该路径名的后继分

量名所在的那个分支游历文件树而找到。

在 UNIX/Linux 系统中，程序不了解内核按怎样的内部格式存贮文件，而把数据作为无格式的字节流看待。程序可以按他们自己的意愿去解释字节流，但这种解释与操作系统如何存储数据无关。因此，对文件中数据进行存取的语法是由系统定义的，并且对所有的程序都是同样的。但是，数据的语义是由程序自己定义的。比如，正文格式化程序 troff 希望在正文的每一行尾部着到换行符，而系统记帐程序则希望找到定长记录。两个程序都使用相同的系统服务，以存取文件中作为字节流存在的数据，而在程序内部，它们通过分析把字节流解释成适当的格式。如果哪一个程序发现格式是错误的，则由它自己负责采取适当的行动。

从这方面说，目录也像正规文件。系统把目录中的数据作为字节流看待。但是由于该数据中包含许多以预定格式记录的目录中的文件名，所以操作系统以及诸如 ls 这样的程序就能够在目录中发现文件。

> i 节点(inode)：
>
> Linux 缺省使用一种叫 EXT 2 的文件系统，在这种文件系统中，每个文件在它所在的目录中都有一个对应的 inode，其中保存了文件的文件名，长度，存取权限等信息。可以这样认为：目录就是由 inode 所组成的特殊文件。

对一个文件的存取权限由与文件相联系的 access permissions 所控制。存取权限能够分别对文件所有者，同组用户及其它人这三类用户独立的建立许可权，以控制读写及执行的许可权。如果目录存取权限允许的话，则用户可以创建文件。新创建的文件是文件系统目录结构的树叶节点。

对于用户来说，UNIX 系统把设备看成文件。以特殊设备文件标名的设备，占据着文件系统目录结构中的节点位置。程序存取正规文件时使用什么语法，他们在存取设备时也使用什么语法。读写设备的语义在很大程度上与读写正规文件时相同。设备保护方式与正规文件的保护方式相同：都是通过适当建立它们的（文件）存取许可权实现的。由于设备名看起来象正规文件名，并且对于设备和正规文件能执行相同的操作，所以大多数程序在其内部不必知道它们所操纵的文件的类型。

2.2.2 流和标准 I/O 库

UNIX/Linux 内核为我们提供了一系列用于访问文件系统（包括其它 I/O 设备）的系统调用，如 open,close 等，通过这些系统调用我们可以实现全部的 I/O 功能。但由这些系统调用组成的 I/O 系统也存在使用不便，缺乏灵活性等的缺点。

为了提高 I/O 系统的模块性和灵活性，Ritchie 提出了流的概念。“流”是在内核空间中的流驱动程序与用户空间中的进程之间的一种全双工处理和数据传输通路。在内核中，流通过流首、驱动程序以及它们之间的零个或多个模块组成。流首是流最靠近用户进程的那一端。由流上用户进程发出的所有系统调用都由流首处理。

流驱动程序可以是提供外部 I/O 设备服务的一种设备驱动程序；也可以是一种软件驱动程序，通常这种驱动程序称为伪设备驱动程序。流驱动程序主要处理内核与设备间的数据传输。除了进行流机制使用的数据结构与该设备理解的数据结构间的转换之外，它很少或根本不处理别的数据。

在流首和驱动程序之间可以插入一个或多个模块，以便在流首和驱动程序间传递消息时对其进行中间处理。流模块由用户进程在流中动态的互联。创建这种连接不需要内核编程、汇编或连接编辑。

流使用队列结构，以保持与压入的模块或打开的流设备有关的信息。队列总是成对分

配，一个用于读另一个用于写。每一个驱动程序、模块和流首都各有一个队列对。只要打开流或者把模块压入到流中，就分配队列对。

数据以消息的形式在驱动程序和流首之间以及在模块间传递。消息是一组数据结构，它们用于在用户进程、模块和驱动程序间传递数据、状态和控制信息。从流首向驱动程序，或者从继承向设备传递的消息称之为“顺流”传播（也称之为“写侧”）。类似的，消息以另一方向传递，即从设备向进程或从驱动程序向流首方向传递，称之为“逆流”传播（也称之为“读侧”）。

一个流消息由一个或多个消息块构成。每一个“块”是由首部、数据块和数据缓冲区组成的三元组，流首在用户进程的数据空间和流内核数据空间之间传输数据。用户进程发送给驱动程序的数据被打包成流消息，然后顺流传递。当包含数据的消息经由逆流到达流首时，此消息由流首处理，它把数据复制到用户缓冲区中。

在流内部，消息由类型指示符区分。逆流发送的某些消息类型可能导致流首执行特定的动作，如，送一个信号给用户进程。其它消息类型主要在流内部传递信息，用户进程不会直接见到这些消息。

流的概念已经被 UNIX/Linux 系统所广泛使用。如进程通信中的管道就是用流来实现的。

Ritchie 还为 C 开发了一个基于流的 I/O 库，称为标准 I/O 库。这个库具有有效的、功能强大的和可移植的文件访问性能。组成库的例行程序提供了一个用户不可见的自动缓冲机构，从而使得访问文件的次数和调用系统调用的次数最小化，取得了较高的效率。这个库的使用范围较广，因为它提供了许多比系统调用 read 和 write 更强的性能，如格式输出和数据转换等。标准 I/O 例行库还是可移植的，它们不受任何 UNIX 的特殊性的限制，并且已经成为与 UNIX 无关的 C 语言 ANSI 标准部分。任何 C 编译程序都提供对标准 I/O 库全部例行程序的访问，而不管其操作系统是什么。

输入输出（文件系统及其操作）是 UNIX/Linux 程序设计中的基础和重要组成，但是由于在大部分 C 语言教材中对此都有比较详细的介绍，故请对这个问题有兴趣的读者自行参阅其它资料，这里不再赘述。

2.3 进程

在多道程序工作的环境下，操作系统必须能够实现资源的共享和程序的并发执行，从而使程序的执行出现了并行、动态和相互制约的新特征。为了能反映程序活动的这些新特点，UNIX 引入了进程（process）这个概念。UNIX 的进程是一个正在执行的程序的映象。这里需要注意的是程序和进程的区别。一个程序是一个可执行的文件，而一个进程则是一个执行中的程序实例。在 UNIX/Linux 系统中可以同时执行多个进程（这一特征有时称为多任务设计），对进程数目无逻辑上的限制，并且系统中可以同时存在一个程序的多个实例。各种系统调用允许进程创建新进程、终止进程、对进程执行的阶段进行同步及控制对各种事件的反映。在进程使用系统调用的条件下，进程便相互独立的执行了。

进程是 UNIX/Linux 程序设计中最重要的部分，在后面的章节中我们将对进程作详细的介绍。

第三章　进程控制

3.1　进程的建立与运行

3.1.1　进程的概念

在 UNIX 中，进程是正在执行的程序。它相当于 Windows 环境内的任务这一概念。每个进程包括程序代码和数据。其中数据包含程序变量数据、外部数据和程序堆栈等。

系统的命令解释程序 shell 为了执行一条命令，就要建立一个新的进程并运行它，例如：

```
$cat file1
```

该命令就会使 shell 专门建立一个进程来运行 cat 命令。

再看一个复杂一些的命令：

```
$ls | wc –ll
```

这个命令就会使 shell 建立两个进程，以并发运行命令 ls 和 wc,把目录列表命令 ls 的输出通过管道送至字计数命令 wc。

因为一个进程对应于一个程序的执行，所以绝对不要把进程与程序这两个概念相混淆。进程是动态的概念，而程序为静态的概念。实际上，多个进程可以并发执行同一个程序，对于公用的实用程序就常常是这样。例如，几个用户可以同时运行一个编辑程序，每个用户对此程序的执行均作为一个单独的进程。

在 UNIX 中，一个进程又可以启动另一个进程，这就给 UNIX 的进程环境提供了一个象文件系统目录树那样的层次结构。进程树的顶端是一个控制进程，它是一个名为 init 的程序的执行，该进程是所有用户进程的祖先。

Linux 同样向程序员提供一些进程控制方面的系统调用，其中最重要的有以下几个：

1．fork()。它通过复制调用进程来建立新的进程，它是最基本的进程建立操作。

2．exec。它包括一系列的系统调用，其中每个系统调用都完成相同的功能，即通过用一个新的程序覆盖原内存空间，来实现进程的转变。各种 exec 系统调用之间的区别仅在于它们的参数构造不同。

3．wait()。它提供了初级的进程同步措施，它能使一个进程等待，直到另一个进程结束为止。

4．exit()。这个系统调用常用来终止一个进程的运行。

在下面，我们将对 Linux 的进程进行详细的讨论，并要对以上系统调用作出详细的介绍。

3.1.2　进程的建立

系统调用 fork()是建立进程的最基本操作,它是把 Linux 变换为多任务系统的基础。fork()在 Linux 系统库 unistd.h 中的函数声明如下：

```
pid_t fork(void);
```

如果 fork()调用成功，就会使内核建立一个新的进程，所建的新进程是调用 fork()的进程的副本。也就是说，新的进程运行与其创建者一样的程序，其中的变量具有与创建进程那变量相同的值。但是这两个进程间还是有差距的，我们在下面将详细的讨论。

新建立的进程被成为子进程（child process），那个调用 fork()建立此新进程的进程被称为父进程（parent process）。以后，父进程与子进程就并发执行，它们都从 fork()调用后的那句语句开始执行。

有些读者可能习惯于纯串行的程序设计环境，一开始对 fork()调用的理解可能会有一些困难。图 3-1 给出了 fork()调用的情况，有助于对 fork()调用的理解。图中给出了三个语句，先是调用 printf()，随后调用 fork()，然后又调用 printf()。

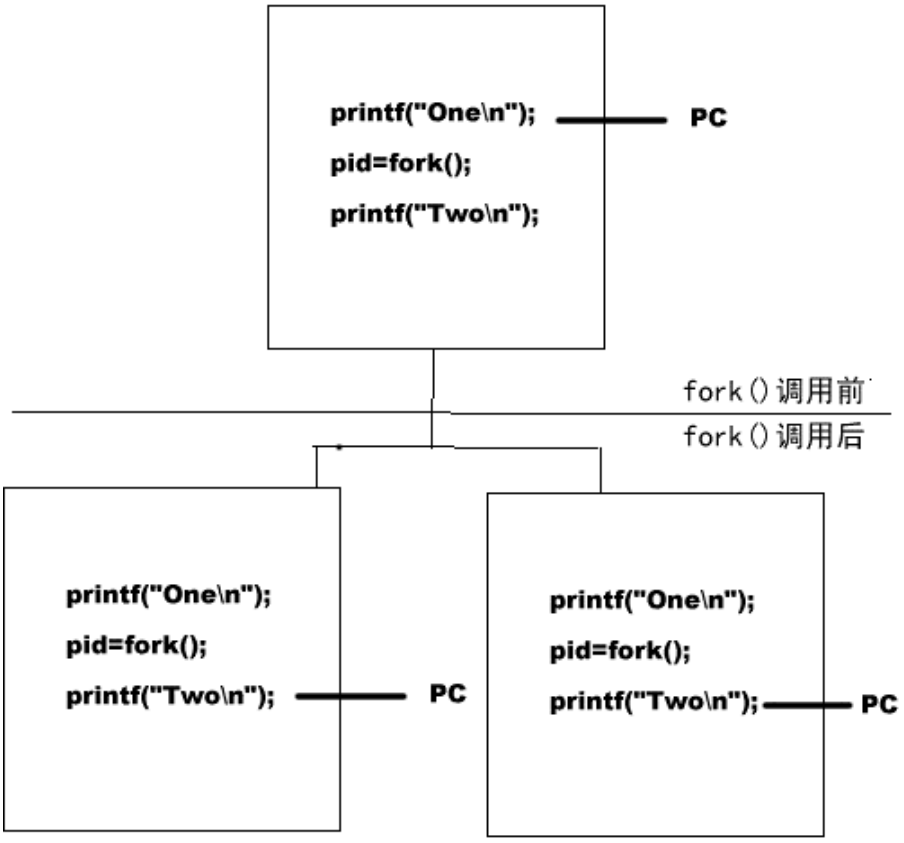

图 3- 1 fork()调用执行示意图

如图 3-1，它分为 fork()调用前和调用后两部分。调用前的那一部分给出了进程 A 调用 fork()的情况。PC（程序计数器）指向当前执行的语句。这时它指向第一个 printf 语句。调用后那一部分给出了调用 fork()以后的情况。这时进程 A 和 B 一起运行，进程 A 是父进程，进程 B 是子进程，它是进程 A 的副本，执行与 A 一样的程序。两个 PC 都指向第二个 printf 语句，即 fork()调用之后的语句。也就是说，A 和 B 都从程序的相同点开始执行。

系统调用 fork()没有参数，它返回一个 pid_t 类型的值 pid。pid 被用来区分父进程和子进程。在父进程中，pid 被置为一个非 0 的正整数；在子进程中，pid 被置为 0。根据 fork()在父进程和子进程中的返回值不同，程序员可以据此为两个进程指定不同的工作。

在父进程中，pid 中返回的数是子进程的进程标识符。这个数用于在系统中表示一个进程，就像用户标识符标识一个用户那样。因为所有的进程都是通过 fork()调用形成的，所以每个 UNIX 进程都有自己的进程标识符，而且它是唯一的。

下面请大家看一个程序，从中可以看到系统调用 fork()的作用，以及进程标识符的使用情况：

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

main()
{
    pid_t pid;
    printf("Now only one process\n");
    printf("Calling fork…\n");
    pid=fork();
    if (!pid)
        printf("I'm the child\n");
    else if (pid>0)
        printf("I'm the parent, child has pid %d\n",pid);
    else
        print ("Fork fail!\n");
}
```

fork 调用后面的条件语句有三个分支：第一个分支对应于 pid 的值为零，它给出了子进程的工作；第二个分支对应于 pid 之值为正数，它给出了父进程的工作。第三个分支对应于 pid 之值为负数（实际为-1），它给出了 fork 建立子进程失败时所作的工作。当系统那进程总数已达到系统规定的最大数，或者是用户可建立的进程数已达到系统规定的最大数时，这时再调用 fork，则会导致失败，并在 errno 中含有出错代码 EAGAIN。我们还应该注意到。上述两个进程间没有同步措施，所以父进程和子进程的输出内容有可能会叠加在一起。

从上面的讨论可以直到，fork()调用是一个非常有用的系统调用。如果把它隔离起来单独看的话，其似乎是空洞无意义的。但是，当它与其它的 Linux 功能结合起来时，就显现出了它的价值。例如，可以用 Linux 提供的进程间通信机构（如信号和管道等），使父进程与子进程协作完成彼此有关的不同任务。经常与 fork()配合使用的另一个系统调用是 exec，我们即将在下面讨论它。

3.1.3 进程的运行

1．系统调用 exec 系列

如果 fork()是程序员唯一可使用的建立进程的手段，那么 Linux 的性能会受很大影响。因为 fork()只能建立相同程序的副本。幸运的是，Linux 还提供了系统调用 exec 系列，它可以用于新程序的运行。exec 系列中的系统调用都完成相同的功能，它们把一个新程序装入调用进程的内存空间，来改变调用进程的执行代码，从而形成新进程。如果 exec 调用成功，调用进程将被覆盖，然后从新程序的入口开始执行。这样就产生了一个新的进程，但是它的进程标识符与调用进程相同。这就是说，exec 没有建立一个与调用进程并发的新进程，而是用新进程取代了原来的进程。所以，对 exec 调用成功后，没有任何数据返回，这与 fork()不同。下面给出了 exec 系列调用在 Linux 系统库中 unistd.h 中的函数声明：

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int    execle( const char *path, const char *arg , ..., char* const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

为了使事情简单明了，我们将着重讨论 exec 系列中的一个系统调用，即 execl()。execl()调用的参数均为字符型指针，第一个参数 path 给出了被执行的程序所在的文件名，它必须是一个有效的路径名，文件本身也必须含有一个真正的可执行程序。但是不能用 exec()l 来运行一个 shell 命令组成的文件。系统只要检查文件的开头两个字节，就可以知到该文件是否为程序文件（程序文件的开头两个字节是系统规定的专用值）。第二个以及用省略号表示的其它参数一起组成了该程序执行时的参数表。按照 Linux 的惯例，参数表的第一项是不带路径的程序文件名。被调用的程序可以访问这个参数表，它们相当于 shell 下的命令行参数。实际上，shell 本身对命令的调用也是用 exec 调用来实现的。由于参数的个数是任意的，所以必须用一个 null 指针来标记参数表的结尾。下面给出一个使用 execl 调用来运行目录列表程序 ls 的例子：

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

main()
{
    printf("Executing ls\n");
    execl("/bin/ls","ls","-l",NULL);

    /* 如果 execl 返回，说明调用失败 */
    perror("execl failed to run ls");
    exit(1);
}
```

我们用图 3-2 来表示该程序的工作情况。调用前那一部分给出了 execl()即将执行之前时的进程情况，调用后那一部分给出了被改变进程的情况，它现在运行 ls 程序。程序计数器 PC 指向 ls 的第一行，表明 execl()导致从新程序的入口开始执行。

请注意，程序在 execl()调用后紧跟着一个对库例行程序 perror()的无条件调用。这是因为，如果调用程序还存在，并且 execl()调用返回，那么肯定是 execl()调用出错了。这时，execl()和其它 exec 调用总是返回-1。这也就是说，只要 execl()和其它 exec 调用成功，就肯定清除了调用程序而代之以新的程序。

exec 系列的其它系统调用给程序员提供使用 exec 功能的灵活性，它们能适用于多种形式的参数表。execv()只有两个参数：第一个参数指向被执行的程序文件的路径名，第二个参数 argv 是一个字符型指针的数组，如下所示：

```
char *argv []
```

这个数组中的第一个元素指向被执行程序的文件名（不含路径），剩下的元素指向程序所用的参数。因为该参数表的长度是不确定的，所以要用 null 指针作结尾。

下面给出一个用 execv()运行 ls 命令的例子：

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

main()
{
    char* av[]={"ls","-l",NULL};
    execv("/bin/ls",av);
    perror("execv failed");
```

```
    exit(1);
}
```

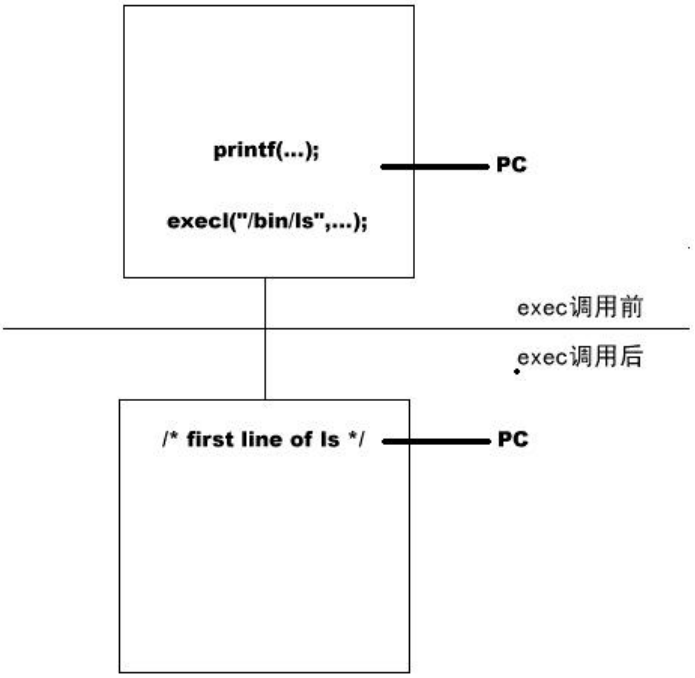

图 3-2 exec()**调用执行示意图**

系统调用 execlp()**和** execvp()**分别类似于系统调用** execl()**和** execv()**，它们的主要区别是：**execlp()**和** execvp()**的第一个参数指向的是一个简单的文件名，而不是一个路径名。它们通过检索** shell **环境变量** PATH**指出的目录，来得到该文件名的路径前缀部分。例如，可以在** shell **中用下述命令序列来设置环境变量** PATH**：**

```
$PATH=/bin;/usr/bin;/sbin
$export PATH
```

这就使 execlp()**和** execvp()**首先在目录**/bin,**然后在目录**/usr/bin**，最后在目录**/sbin **中搜索程序文件。另外，**execlp **和** execvp **还可以用于运行** shell **程序，而不只是普通的程序。**

2**．对** exec **传送变量的访问**

任何被 exec **调用所执行的程序，都可以访问** exec **调用中的参数。这些参数是调用** exec **的程序传送给它的。我们可以通过定义程序** main()**函数的参数来使用这些参数，方法如下：**

```
main( int argc, char* argv[] );
```

这对于大多数人来说应该是熟悉的，这种方法就是 C **语言程序访问命令行参数的方法。这也显示了** shell **本身就是使用** exec **启动进程的。**

以上说明的 main()**函数中，**argc **是参数计数器，**argv **指向参数数组本身。所以，用** execvp() **执行一个程序，如下所示：**

```
chat* argin[]={“command”, “with”, “argument”, NULL};
```

当 prog **程序启动后，它取得的** argc **和** argv **之值如下：**

```
argc=3;
argv[0]=”command”;
argv[1]=”with”;
argv[2]=”argument”;
```

```
    argv[3]=NULL;
```

为了进一步说明这种参数传递技术，请考虑下列程序 showarg：

```
#include <stdio.h>

main(int argc,char* argv[])
{
    while(--argc>0)
    {
        printf("%s ",*(++argv));
        printf("\n");]
}
```

这个程序的工作是把它的参数（除第一个参数外）的值送标准输出。如果用如下程序段来调用 showarg **的话，则其** argc **参数为** 3,**输出结果为：**”hello world”**。**

```
char* argin[]={"showarg", "hello", "world", NULL};
execvp(argin[0],argin);
```

3．exec **和** fork()**的联用**

系统调用 exec **和** fork()**联合起来为程序员提供了强有力的功能。我们可以先用** fork()**建立子进程，然后在子进程中使用** exec,**这样就实现了父进程运行一个与其不同的子进程，并且父进程不会被覆盖。**

下面我们给出一个 exec **和** fork()**联用的例子，从中我们可以清楚的了解这两个系统调用联用的细节。其程序清单如下：**

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

main()
{
        int pid;
        /* fork 子进程 */
        pid=fork();
        switch(pid) {
        case -1:
                perror("fork failed");
                exit(1);
        case 0:
                execl("/bin/ls","ls","-l","--color",NULL);
                perror("execl failed");
                exit(1);
        default:
                wait(NULL);
                printf("ls completed\n");
                exit(0);
        }
}
```

在程序中，在调用 fork()**建立一个子进程之后，马上调用了** wait()，**使父进程在子进程**

结束之前，一直处于睡眠状态。所以，wait()向程序员提供了一种实现进程之间同步的简单方法，我们将在下面对它作出更详细的讨论。

为了说明得更清楚一些，我们用图 3-3 来解释程序的工作。图 3-3 分为 fork()调用前、fork()调用后和 exec 调用后三个部分。

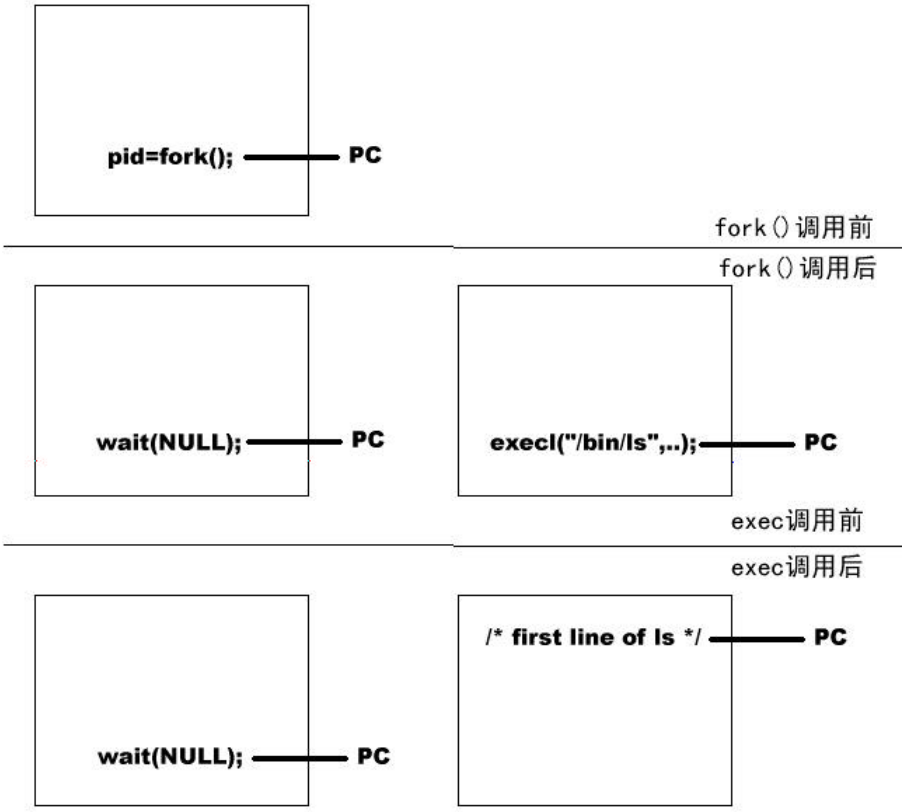

图 3-3 exec()和 fork()的联用

在 fork()调用前，只有一个进程 A，PC 指向将要执行的下一个语句。fork()调用后，就有了进程 A 和进程 B。A 是父进程，它正在执行系统调用 wait()，使进程 A 睡眠，直至进程 B 结束。同时，B 正在用 exec 装入命令 ls。exec 调用后，进程 B 的程序被 ls 的代码取代，这时执行 ls 命令的代码。进程 B 的 PC 指向 ls 的第一个语句。由于 A 正在等待 B 的结束，所以它的 PC 所指位置未变。

现在我们应该了解命令解释程序 shell 的工作概况。当 shell 从命令行接受到以正常方式（即前台运行）执行一个命令或程序的要求时，它就按上述方法调用 fork()、exec 和 wait()，以实现命令或程序的执行。当要求在后台执行一个命令或程序时，shell 就省略对 wait 的调用，使得 shell 和命令进程并发运行。

为了帮助读者进一步熟悉和掌握 fork()和 exec 的使用，我们再来看一个名为 docommand 的程序，这个程序仿真 Linux 库调用 system()，它可以在程序中执行一个 shell 命令。docommand 的主题是对 fork()和 exec 的调用。程序清单如下：

```
int docommand(char* command)
{
    int pid;
    switch(pid=fork())
    {
```

```
        case -1:
            return -1;
        case 0:
            execl("/bin/sh","sh","-c",command,NULL);
            exit(127);
        default:
            wait(NULL);
        }
        return 0;
    }
```

docommand 并没有通过 exec 去直接执行指定的命令，而是通过 exec 去执行 shell（即/bin/sh），并由 shell 再执行指定的命令。这是一种非常巧妙的方法，它使得 docommand 能使用 shell 提供的一系列特性（如文件名扩展等）。在引用 shell 中使用的参数-c，表示从下一个参数中取得命令名，而不是从标准输入上取得。

3.1.4 数据和文件描述符的继承

1．fork()、文件和数据

用系统 fork()建立的子进程几乎与其父进程完全一样。子进程中的所有变量均保持它们在父进程中之值（fork()的返回值除外）。因为子进程可用的数据是父进程可用数据的拷贝，并且其占用不同的内存地址空间，所以必须要确保以后一个进程中变量数据的变化，不能影响到其它进程中的变量。这一点非常重要。

另外，在父进程中已打开的文件，在子进程中也已被打开，子进程支持这些文件的文件描述符。但是，通过 fork()调用后，被打开的文件与父进程和子进程存在着密切的联系，这是以为子进程与父进程公用这些文件的文件指针。这就有可能发生下列情况：由于文件指针由系统保存，所以程序中没有保存它的值，从而当子进程移动文件指针时，也等于移动了父进程的文件指针。这就可能会产生意想不到到结果。

为了说明上述情况，我们给出一个实例程序 proc_file。在这个程序中使用了两个预定义的函数 failure()和 printpos()。failure()用来完成简单的出错处理，它只是调用 perror()来显示出错信息。其实现如下：

```
failure( char* s)
{
    perror(s);
    exit(1);
}
```

printpos()实现显示一个文件的文件指针之值，其实现如下：

```
printpos( char* string, int fildes)
{
    long pos;
    if ((pos=lseek(fildes,0L,1)<0L)
        failure("lseek failed");
    printf("%s: %ld \n",string,pos);
}
```

另外我们还假定文件 data 已经存在，并且它的长度不小于 20 个字符。下面给出程序 proc_file 的清单：

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

failure( char* s)
{
        perror(s);
        exit(1);
}

printpos( char* string, int fildes)
{
        long pos;
        if ((pos=lseek(fildes,0L,1))<0L)
              failure("lseek failed");
        printf("%s: %ld \n",string,pos);
}

main()
{
 int fd; /* 文件描述符 */
 int pid;/* 进程标识符 */
 char buf[10]; /* 数据缓冲区 */
 /* 打开文件 */
 if ((fd=open("data",O_RDONLY))<0)
        failure("open failed");
 read(fd,buf,10); /* advance file pointer */
 printpos("Before fork",fd);
 /* fork 新进程 */
 if ((pid=fork())<0)
        failure("fork failed");
 else if (!pid)
        /* 子进程 */
        printpos("Child before read",fd);
        read(fd,buf,10);
        printpos("child after read",fd);
 } else {
        /* 父进程 */
        /* 等待子进程运行结束 */
        wait(NULL);
        printpos("parent after wait",fd);
 }
}
```

该程序运行结果如下：

```
Before fork: 10
Child before read: 10
child after read: 20
parent after wait: 20
```

这充分证明了文件指针为两个进程共用这一事实。

2．exec()和打开文件

当一个程序调用 exec 执行新程序时，在程序中已被打开的文件，其在新程序中仍保持打开。这就是说，已打开文件描述符能通过 exec 被传送给新程序，并且这些文件的指针也不会被 exec 调用改变。

这儿，我们要介绍一个与文件有关的执行关闭位（close-on-exec），该位被设置的话，则调用 exec 时会关闭相应的文件。该位的默认值为非设置。例行程序 fcntl 能用于对这一标志位的操作，下面的程序段给出了设置“执行关闭”位的方法。

```
#include <fcntl.h>
    …
    …
    …
int fd;
fd=open("file",O_RDONLY);
    …
    …
fcntl(fd,F_SETFD,1);
```

如果已经设置了执行关闭位，我们可以用下面的语句来撤销“执行关闭“位的设置，并取得它的返回值：

```
res=fcntl(fd,F_SETFD,0);
```

如果文件描述符所对应的文件的“执行关闭位”已经被设置，则 res 为 1,否则 res 之值为 0。

3.2 进程的控制操作

3.2.1 进程的终止

系统调用 exit()实现进程的终止。exit()在 Linux 系统函数库 stdlib.h 中的函数声明如下：

```
void exit(int status);
```

exit()只有一个参数 status，称作进程的退出状态，父进程可以使用它的低 8 位。exit()的返回值通常用于指出进程所完成任务的成败。如果成功，则返回 0；如果出错，则返回非 0 值。

exit()除了停止进程的运行外，它还有一些其它作用，其中最重要的是，它将关闭所有已打开的文件。如果父进程因执行了 wait()调用而处于睡眠状态，那么子进程执行 exit()会重新启动父进程运行。另外，exit()还将完成一些系统内部的清除工作，例如缓冲区的清除工作等。

除了使用 exit()来终止进程外，当进程运行完其程序到达 main()函数末时，进程会自动终止。当进程在 main()函数内执行一个 return 语句时，它也会终止。

在 Linux 中还有一个用于终止进程的系统调用_exit()。它在 Linux 系统函数库 unistd.h 中被声明：

```
void _exit(int status)
```

其使用方法与 exit()完全相同，但是它执行终止进程的动作而没有系统内部的清除工作。因此，只有那些对系统内部了解比较深的程序员才使用它。

3.2.2 进程的同步

系统调用 wait()是实现进程同步的简单手段，它在 Linux 系统函数库 sys/wait.h 中的函数声明如下：

```
pid_t wait(int *status)
```

我们在前面已经看到了，当子进程执行时，wait()可以暂停父进程的执行，使起等待。一旦子进程执行完，等待的父进程就会重新执行。如果有多个子进程在执行，那么父进程中的 wait()在第一个子进程结束时返回，恢复父进程执行。

通常情况下，父进程调用 fork()后要调用 wait()。例如：

```
pid=fork();
if (!pid){
    /* 子进程 */
} else {
    /* 父进程 */
    wait(NULL);
}
```

当希望子进程通过 exec 运行一个完全不同的进程时，就要进程 fork()和 wait()的联用。wait()的返回值通常是结束的那个子进程的进程标识符。如果 wait()返回-1，表示没有子进程结束，这时 errno 中含有出错代码 ECHILD。

wait()有一个参数，它可以是一个指向整型数的指针，也可以是一个 null 指针。如果参数用了 null 指针，wait 就忽略它。如果参数是一个有效的指针，那么 wait 返回时，该指针就指向子进程退出时的状态信息。通常，该信息就是子进程通过 exit 传送出来的出口信息。下面的程序 status 就给出了这种情况下，wait 的使用方法。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

main()
{
int pid,status,exit_status;

if ((pid=fork()) <0)
{
    perror("fork failed");
    exit(1);
}

if (!pid)
    /* 子进程 */
    sleep(4);
    exit(5); /* 使用非零值退出，以便主进程观察 */
```

```
}

/* 父进程 */
if (wait(&status) <0) {
        perror("wait failed");
        exit(1);
}

/* 将 status 与 0xFF(255)与来测试低 8 位 */
if (status & 0xFF)
        printf("Somne low-roderbits not zero\n");
else {
        exit_status=status >> 8;
        exit_status &=0xFF;
        printf("Exit status from %d was %d\n", pid,exit_status);
}
exit(0);
}
```

虽然这个过程看起来有一点复杂，但是其含义十分清除：通过 exit 返回给父进程之值存放在 exit_status 的低位中，为了使其有意义，exit_status 的高 8 位必须为 0（注意，在整型量中，低 8 位在前，高 8 位在后）。因此从 wait()返回后，就可以用按位与操作进行测试，如果它们不为 0，表示该子进程是被另一个进程用一种称为信号的通信机构停止的，而不是通过 exit()结束的。

3.2.3 进程终止的特殊情况

我们在前面讨论了用 wait()和 exit()联用来等待子进程终止的情况。但是，还有两种进程终止情况值得讨论。这两种情况为：

1．子进程终止时，父进程并不正在执行 wait()调用。

2．当子进程尚未终止时，父进程却终止了。

在第一种情况中，要终止的进程就处于一种过渡状态（称为 zombie），处于这种状态的进程不使用任何内核资源，但是要占用内核中的进程处理表那的一项。当其父进程执行 wait()等待子进程时，它会进入睡眠状态，然后把这种处于过渡状态的进程从系统内删除，父进程仍将能得到该子进程的结束状态。

在第二种情况中，一般允许父进程结束，并把它的子进程（包括处于过渡状态的进程）交归系统的初始化进程所属。

3.2.4 进程控制的实例

在这一部分，我们将利用前面介绍的进程控制的知识，来构造一个简单的命令处理程序，取名为 smallsh。这样作有两个目的：第一，可以巩固和发展我们在这一章中介绍的概念；第二，它展示了标准的 Linux 系统程序也没有什么特别的东西。特别是，它表明了 shell 也只是一个在用户注册时调用的普通程序。

smallsh 的基本功能是：它能在前台或后台接收命令并执行它们。它还能处理由若干个命令组成的命令行。它还具有文件名扩展和 I/O 重定向等功能。

smallsh 的基本逻辑如下：

```
while (EOF not typed) {
        从用户终端取得命令行
        执行命令行
}
```

我们把取命令行内容用一个函数来完成，并取名为 userin。userin 能显示提示符，然后等待用户从键盘输入一命令行信息。它接收到的输入内容应存入程序的一个缓冲区中。

我们可以忽略到一些初始化工作，但是，userin 的基本步骤是：首先显示提示符，提示符的具体内容由用户通过参数传送给函数；然后每次从键盘读一个字符，当遇到换行符或文件结束符（用 EOF 符号表示）时，就结束。

我们用的基本输入例程是 getchar，它实际上是标准 I/O 库中的一个宏（macro），它从程序的标准输入读入一个字符，userin 把每个读入的字符都存入字符型数组 inpbuf 中，当它结束时，userin 就返回读入字符的个数或 EOF(表示文件结尾)。注意，换行符也要存入 inpbuf，而不能丢弃。

函数 userin 的代码如下：

```
#include "smallsh.h"

/*  程序缓冲区和指针  */
static char inpbuf[MAXBUF],tokbuf[2*MAXBUF],
        *ptr =inpbuf,*tok=tokbuf;
/* userin()函数  */
int userin(chat* p)
{
 int c,count;

 ptr=inpbuf;
 tok=tokbuf;
 /*  显示提示  */
 printf("%s ",p);
 for (count=0;;) {
 if ((c=getchar())==EOF)
        return(EOF);
 if (count<MAXBUF)
        inpbuf[count++]=c;
 if (c =='\n' && count <MAXBUF) {
        inpbuf[count]='\0';
        return(count);
 }
 /*  如果行过长重新输入  */
 if (c=='\n') {
        printf("smallsh:input line too long\n");
        count=0;
        printf("%s ",p);
 }
 }
```

```
}
```

头文件 smallsh.h 中含有一些有用的定义，该文件的内容如下所示：

```
#include <stdio.h>

#define EOL 1 /* 行结束 */
#define ARG 2
#define AMPERSAND 3
#define SEMICOLON 4
#define MAXARG 512 /* 命令行参数个数的最大值 */
#define MAXBUF 512 /* 输入行的最大长度 */
#define FOREGROUND 0
#define BACKGROUND 1
```

上述文件中定义的内容，有一些未被 userin 引用，我们将在后面的例程中引用它们。smallsh.h 文件中还蕴涵了标准头文件 stdio.h，它为我们提供了 getchar 和 EOF 的定义。

接下来我们看一下 gettok，它从 userin 构造的命令行缓冲区中分析出命令名和参数。gettok 的调用方法为：

```
toktype=gettok(&tptr);
```

toktype 是一个整型变量，它的值指出分析出内容之类型。它的取值范围可以从 smallsh.h 中得到，包括 EOL，SEMICOLON 等。tptr 是一个字符型指针，gettok 调用后，该指针指向实际的析出内容。由于 gettok 要为分析出的内容分配存贮区，所以我们必须传送 tptr 的地址，而不是它的值。

下面给出 gettok 的程序。由于它引用了字符指针 tok 和 ptr，所以它必须与 userin 放在同一个源文件中。现在可以知道在 userin 的开头初始化 tok 和 ptr 的原因了。

```
gettok(char* output)
{
    int type;

    outptr=tok;
    /* 首先去除空白字符 */
    for (;*ptr==''||*ptr=='\t';ptr++);
    *tok++=*ptr;
    switch(*ptr++) {
        case '\n':
        type=EOL;break;
        case '&':
        type=AMPERSAND;break;
        case ';':
        type=SEMICOLON;break;
        default:
        type=ARG;
        while (inarg(*ptr))
            *tok++=*ptr++;
    }
    *tok++='\0';
```

```
        return (type);
    }
```

例行程序 inarg **用于确定一个字符是否可以作为参数的组成符。我们只要检查这个字符是否是** smallsh **的特殊字符。** inarg **的程序如下：**

```
    static char special[]={' ','\t','*',';','\n','\0'};

    inarg(char c)
    {
        char *wrk;
        for (wrk=special;*wrk!='\0';wrk++)
            if (c==*wrk)
                return(0);
        return(1);
    }
```

在上面我们已经介绍了完成实际工作的几个函数。我们下面将介绍使用这些完成实际工作的函数的例行程序。

例行程序 procline **使用函数** gettok()**分析命令行，在处理过程中构造一张参数表。当它遇到换行符或分号时，它就调用例行程序** runcommand **来执行被分析的命令行。它假定已经用** userin **读入了一个输入行。下面给出例行程序** procline **的代码：**

```
    #include "smallsh.h"

    procline()
    {
     char * arg[MAXARG+1];
     int toktype;
     int narg;
     int type;

     for(narg=0;;) {
     switch(toktype=gettok(&arg[narg])) {
     case ARG:
        if (narg<MAXARG)
            narg++;
        break:
     case EOL:
     case SEMICOLON:
     case AMPERSAND:
        type=(toktype==AMPERSAND)?
        BACKGROUND:FOREGROUND;
        if (narg!=0) {
            arg[narg]=NULL;
            runcommand(arg,type);
        }
        if (toktype==EOL)
```

```
                return;
            narg=0;
            break;
        }
        }
    }
```

下一步是说明 runcommand 例行程序，它实现启动命令进程。runcommand 在本质上是前面介绍的例行程序 docommand 的改进。它设有一个整型参数 where，如果 where 之值被设置为 BACKGROUND（在 smallsh.h 中被定义），那末将忽略 wait()调用，并且 runcommand 只显示进程标识符后就返回。下面给出 runcommand 的代码：

```
#include "smallsh.h"

runcommand(char** cline,int where)
{
 int pid,exitstat,ret;

 if((pid=fork())<0) {
        perror("fork fail");
        return(-1);
 }
 if (!pid) { /*  子进程代码  */
        execvp(*cline,cline);
        perror(*cline);
        exit(127);
 }
 /*  父进程代码  */
 /*  后台进程代码  */
 if (where==BACKGROUND) {
        printf("[process id %d]\n",pid);
        return(0);
 }
 /*  前台进程代码  */
 while ((ret=wait(&exitstat))!=pid && ret !=-1) ;
 return (ret==-1?-1:exitstat);
}
```

在这里例行程序中，用下面的复杂循环代替了 docommand 中简单的 wait()调用：

```
while (ret=wait(&exitstat))!=pid && ret!=-1);
```

这就可以保证，只有当最后被启动的子进程结束时，runcommand 才结束，这就避免了后台命令中途结束带来的问题。如果觉得这看起来还不太清楚，那么请记住，wait()返回的是第一个结束的子进程的进程标识符，而不是最后一个被启动的子进程的进程标识符。

runcommand 还使用了 execvp()系统调用，这意味着按当前环境变量 path 中的目录来搜索命令中表明的程序文件。

最后一步是写出 main()函数，它把上面介绍的各个部分联系到一起，其程序如下：

```
#include "smallsh.h"
```

```
char *prompt="command>";

main()
{
 while (userin(prompt)!=EOF)
       procline();
}
```

3.3 进程的属性

每个 Linux 进程都具有一些属性，这些属性可以帮助系统控制和调度进程的运行，以及维持文件系统的安全等。我们已经接触过一个进程属性，它就是进程标识符，用于在系统内标识一个进程。另外还有一些来自环境的属性，它们确定了进程的文件系统特权。我们在本节中还要介绍其它一些重要的进程属性。

3.3.1 进程标识符

系统给每个进程定义了一个标识该进程的非负正数，称作进程标识符。当某一进程终止后，其标识符可以重新用作另一进程的标识符。不过，在任何时刻，一个标识符所代表的进程是唯一的。系统把标识符 0 和 1 保留给系统的两个重要进程。进程 0 是调度进程，它按一定的原则把处理机分配给进程使用。进程 1 是初始化进程，它是程序/sbin/init 的执行。进程 1 是 UNIX 系统那其它进程的祖先，并且是进程结构的最终控制者。

利用系统调用 getpid 可以得到程序本身的进程标识符，其用法如下：

```
pid=getpid();
```

利用系统调用 getppid 可以得到调用进程的父进程的标识符，其用法如下：

```
ppid=getppid();
```

下面给出一个例子，其中的例行程序 gentemp 使用 getpid 产生一个唯一的临时文件名，该文件名的形式为：

```
/tmp/tmp<pid>.<no>
```

每对 getemp()调用一次，文件名的后缀 no 就增 1，文件名中的 pid 为用 getpid 取到的进程标识符。该例行程序还调用 access 来检查该文件是否已经存在，更增加了可靠性。

例行程序 gentemp 的代码如下所示：

```
#include <strings.h>
#include <unistd.h>

static int num=0;
static char namebuf[20];
static char prefix[]="/tmp/tmp";

char* gentemp()
{
      int length,pid;

      /*  获得进程标识符  */
```

```
    pid=getpid();

    strcpy(namebuf,prefix);
    length=strlen(namebuf);
    /* 在文件名中增加 pid 部分 */
    itoa(pid,&namebuf[length]);

    strcat(namebuf,".");
    length=strlen(namebuf);
    do{
        /* 增加后缀 number */
        itoa(num++,&namebuf[length]);
    } while (access(namebuf,0)!=-1);
    return namebuf;
}
```

在 gentemp 中调用了例行程序 itoa()，这个例行程序把一个整数转换成其对应的 ASCII 字符串。下面给出 itoa()的程序清单。请注意其中的第二个 for 循环体内的第一个语句，它实现把一个数转换成其对应的 ASCII 字符。

```
/* itoa 把整型转换成字符串 */
itoa( int i,char* string)
{
    int power, j;

    j=i;
    for (power=1;j>=10;j/=10)
        power*=10;
    for (;power>0;power/=10) {
        *string++='0'+i/power;
        i%=power;
    }
    *string='\0';
}
```

3.3.2 进程的组标识符

Linux 把进程分属一些组，用进程的组标识符来知识进程所属组。进程最初是通过 fork()和 exec 调用来继承其进程组标识符。但是，进程可以使用系统调用 setpgrp()，自己形成一个新的组。setpgrp()在 Linux 系统函数库 unistd.h 中的函数声明如下：

```
int setpgrp(void);
```

setpgrp()的返回值 newpg 是新的进程组标识符，它就是调用进程的进程标识符。这时，调用进程就成为这个新组的进程组首（process group leader）。它所建立的所有进程，将继承 newpg 中的进程组标识符。

一个进程可以用系统调用 getpgrp()来获得其当前的进程组标识符，getpgrp()在 Linux 系统函数库 unistd.h 中的函数声明如下：

```
int setpgrp(void);
```

函数的返回值就是进程组的标识符。

进程组对于进程间的通信机构——信号来说，是非常有用的。我们将在下一章内讨论它。现在，我们讨论进程组的另一个应用。当某个用户退出系统时，则相应的 shell 进程所启动的全部进程都要被强行终止。系统是根据进程的组标识符来选定应该终止的进程的。如果一个进程具有跟其祖先 shell 进程相同的组标识符，那末它的生命期将可超出用户的注册期。这对于需要长时间运行的后台任务是十分有用的。

下面给出一个改变进程的组标识符的例子，它的效果相当于使用“不中止”程序 nohup 的效果。

```
main()
{
    int newpgid;

    /* 改变进程组 */
    newpgid=setpgrp();

    /* 程序体 */
    ……
    ……
}
```

3.3.3 进程环境

进程的环境是一个以 NULL 字符结尾的字符串之集合。在程序中可以用一个以 NULL 结尾的字符型指针数组来表示它。系统规定，环境中每个字符串形式如下：

```
name=something
```

Linux 系统提供了 environ 指针，通过它我们可以在程序中访问其环境内容。

在使用 environ 指针前，应该首先声明它：

```
extern char **environ;
```

下面的这段代码（showenv.c）演示了如何通过 environ 指针访问环境变量：

```
extern char** environ;

main()
{
    char** env=environ;

    while (*env) {
        printf(%s\n",*env++);
    }
    return;
}
```

下面是这个程序运行后的结果：

```
HOME=/home/roy
USER=roy
LOGNAME=roy
PATH=/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin
```

```
MAIL=/var/spool/mail/roy
SHELL=/bin/tcsh
SSH_CLIENT=192.168.35.72 1145 22
SSH_TTY=/dev/pts/0
TERM=ansi
HOSTTYPE=i486-linux
VENDOR=intel
OSTYPE=linux
MACHTYPE=i486
SHLVL=1
PWD=/home/roy/test
GROUP=roy
HOST=bbs
HOSTNAME=bbs
```

以上的结果是运行该程序的 shell 进程环境，其中包括了像 HOME 和 PATH这些被 shell 使用的重要变量。

从这个例子中可以看到，一个进程的初始环境与用 fork()或 exec 建立它的父进程之环境相同。由于环境可以通过 fork()或者 exec 被传送，所以其信息被半永久性的保存。对于新建立的进程来说，可以重新指定新的环境。

如果要为进程指定新的环境，则需要使用 exec 系列中的两种系统调用：execle()和 execve()。它们在 Linux 系统函数库 unistd.h 中的函数声明如下：

```
int execle( const char *path, const char *arg , ..., char * const envp[]);
int execve (const char *filename, char *const argv [], char*
    const envp[]);
```

它们的调用方法分别类似于 execl()和 execv()，所不同的是它们增加了一个参数 envp，这是一个以 NULL 指针结束的字符数组，它指出了新进程的环境。下面的程序演示了 execve()的用法，它用 execve()把新的环境传送给上面的程序程序 showenv：

```
#include <unistd.h>

main()
{
char *argv[]={"showenv", NULL},
      *envp[]={"foo=bar", "bar=foo", NULL};

execve("./showenv",argv,envp);
perror("exeve failed.");
return;
}
```

程序执行结果如下：

```
foo=bar
bar=foo
```

最后我们利用 environ 指针构造一个函数 findenv()，其程序如下：

```
extern char** environ;
```

```
char* findenv(char* name)
{
int len;
char **p;

for(p=environ;*p;p++)
{
      if((len=pcmp(name,*p))>=0 &&
            *(*(p+1))=='=')
                  return *(p+l+1);
}
return NULL;
}

int pcmp(char* s1, char* s2)
{
int i=0;

while(*s1) {
      i++;
      if (*s1++!=*s2++)
            return -1;
}
return i;
}
```

findenv()根据参数给出的字符串 name，扫描环境内容，找出“ name=string”这种形式的字符串。如果成功，findenv()就返回一个指向这个字符串中”string”部分的指针。如果不成功，就返回一个 NULL 指针。

在 Linux 的系统函数库 stdlib.h 中提供了一个系统调用 getenv()，它完成与 findenv()同样的工作。另外还有一个与 getenv()相配对的系统调用 putenv()，它用于改变和扩充环境，其使用方法为：

```
putenv(“newvariable=value”);
```

如果调用成功，其就返回零。需要注意的是，它只能改变调用进程的环境，而父进程的环境并不随之改变。

3.3.4 进程的当前目录

每个进程都有一个当前目录。一个进程的当前目录最初为其父进程的当前目录，可见当前目录的初始值是通过 fork()和 exec 传送下去的。我们必须认识到，当前目录是进程的一个属性。如果子进程通过 chdir()改变了它的当前目录，那么其父进程的当前目录并没有因此而改变。鉴于此原因，系统的 cd 命令（改变当前目录命令）实际上是一个 shell 自身的内部命令，其代码在 shell 内部，而没有单独的程序文件。只有这样，才能改变相应 shell 进程的当前目录。否则的话，只能改变 cd 程序所运行进程自己的当前目录。当初刚把多任务处理加入 UNIX 时，cd 命令是作为一个普通程序来实现的，没有考虑到上述情况，因而引起了一些混乱。

类似的，每个进程还有一个根目录，它与绝对路径名的检索起点有关。与当前目录一样，进程的根目录的初始值为其父进程的根目录。可以用系统调用 chroot()来改变进程的根目录，但是这不会改变其父进程的根目录。

3.3.5 进程的有效标识符

每个进程都有一个实际用户标识符和一个实际组标识符，它们永远是启动该进程之用户的用户标识符和组标识符。

进程的有效用户标识符和有效组标识符也许更重要些，它们被用来确定一个用户能否访问某个确定的文件。在通常情况下，它们与实际用户标识符和实际组标识符是一致的。但是，一个进程或其祖先进程可以设置程序文件的置用户标识符权限或置组标识符权限。这样，当通过 exec 调用执行该程序时，其进程的有效用户标识符就取自该文件的文件主的有效用户标识符，而不是启动该进程的用户的有效用户标识符。

有几个系统调用可以用来得到进程的用户标识符和组标识符，详见下列程序：

```
#include <unistd.h>
#include <sys/types.h>

uid_t uid,euid;
gid_t gid,egid;
….
….
/* 取进程的实际用户标识符 */
uid=getuid();

/* 取进程的有效用户标识符 */
euid=geteuid();

/* 取进程的实际组标识符 */
gid=getgid();

/* 取进程的有效组标识符 */
egid=getegid();
```

另外，还有两个系统调用可以用来设置进程的有效用户标识符和有效组标识符，它们的使用格式如下：

```
#include <sys/types.h>
#include <unistd.h>

uid newuid;
pid newgid;
int status;

/* 设定进程的有效用户标识符 */
status=setuid(newuid);

/* 设定进程的有效组标识符 */
```

```
status=getgid(newgid);
```

不是超级用户所引用的进程，只能把它的有效用户表示符和有效组标识符重新设置成其实际用户标识符和实际组标识符。超级用户所引用的进程就可以自由进行其有效用户标识符和有效组标识符的设置。这两个调用的返回值为零，表示调用成功完成；返回值为-1，则表示调用失败。

通过这两个系统调用，进程可以改变自己的标识符，进而改变自己的权限（因为 Linux 中权限是通过标识符来判断的）。比如一个 root 建立的进程可以用这种方法放弃一部分的 root 权限而只保留工作所需的权限。这样可以提高系统的安全性。但是需要注意的是，一旦 root 进程通过这种方式放弃了 root 特权，将无法再通过 setuid()调用的方式重新获得 root 权，因为一个非 root 标识符的进程是无法设定 root 标识符的。这时可以使用 Linux 的另外两个系统调用 seteuid()和 setegid()。其调用方式和前两个完全相同。但是它们是根据进程程序文件的标识符来判断设定的。因此，一个 root 的程序文件在任何时候都可以将自己重新 seteuid()为 root。

3.3.6 进程的资源

Linux 提供了几个系统调用来限制一个进程对资源的使用。它们是 getrlimit()，setrlimit()和 getrusage()。它们的的函数声明如下：

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrlimit (int resource, struct rlimit *rlim);
int getrusage (int who, struct rusage *usage);
int setrlimit (int resource, const struct rlimit *rlim);
```

其中，getrlimit 和 setrlimit 分别被用来取得和设定进程对资源的的限制。它们的参数是相同的，第一个参数 resource 指定了调用操作的资源类型，可以指定的几种资源类型见表 3-1：

表 3-1 resource 参数的取值及其含义

RLIMIT_CPU	CPU 时间，以秒为单位。
RLIMIT_FSIZE	文件的最大尺寸，以字节为单位。
RLIMIT_DATA	数据区的最大尺寸，以字节为单位。
RLIMIT_STACK	堆栈区的最大尺寸，以字节为单位。
RLIMIT_CORE	最大的核心文件尺寸，以字节为单位。
RLIMIT_RSS	resident set 的最大尺寸。
RLIMIT_NPROC	最大的进程数目
RLIMIT_NOFILE	最多能打开的文件数目。
RLIMIT_MEMLOCK	最大的内存地址空间

第二个参数 rlim 用于取得/设定具体的限制。struct rlimit 的定义如下：

```
struct rlimit
{
    int    rlim_cur;
    int    rlim_max;
};
```

rlim_cur 是目前所使用的资源数，rlim_max 是限制数。如果想取消某个资源的限制，可以把 RLIM_INFINITY 赋给 rlim 参数。

只有超级用户可以取消或者放大对资源的限制。普通用户只能缩小对资源的限制。
如果调用成功，函数返回 0，否则返回-1。
系统调用 getrusage()返回当前的资源使用情况。其有两个参数：
第一个参数 who 指定了查看的对象，可以是：

RUSAGE_SELF　查看进程自身的资源使用状况。
RUSAGE_CHILDREN　查看进程的子进程的资源使用状况。

第二个参数 usage 用于接收资源的使用状况，rusage 结构的定义如下：

```
struct rusage
{
    struct timeval ru_utime; /* 使用的用户时间 */
    struct timeval ru_stime; /* 使用的系统时间 */
    long  ru_maxrss;          /* 最大的保留集合尺寸 */
    long  ru_ixrss;       /* 内部共享内存尺寸*/
    long  ru_idrss;       /* 内部非共享数据尺寸 */
    long  ru_isrss;       /* 内部非共享栈尺寸 */
    long  ru_minflt;          /* 重复声明页 */
    long  ru_majflt;          /* 错误调用页数 */
    long  ru_nswap;       /* 交换区 */
    long  ru_inblock;         /* 阻塞的输入操作数 */
    long  ru_oublock;         /* 阻塞的输出操作数 */
    long  ru_msgsnd;          /* 发送的消息 */
    long  ru_msgrcv;          /* 接受的消息 */
    long  ru_nsignals;        /* 接受的信号 */
    long  ru_nvcsw;       /* 志愿上下文开关 */
    long  ru_nivcsw;          /* 非志愿上下文开关 */
};
```

函数调用成功，返回 0，否则返回-1。

3.3.7 进程的优先级

系统以整型变量 nice 为基础，来决定一个特定进程可得到的 CPU 时间的比例。nice 之值从 0 至其最大值。我们把 nice 值称为进程的优先数。进程的优先数越大，其优先权就越低。普通进程可以使用系统调用 nice()来降低它的优先权，以把更多的资源分给其它进程。具体的做法是给系统调用 nice 的参数定一个正数，nice()调用将其加到当前的 nice 值上。例如：

```
#include <unistd.h>

nice(5);
```

这就使当前的优先数增加了 5,显然，其对应进程的优先权降低了。
超级用户可以用系统调用 nice()增加优先权，这时只需给 nice()一个负值的参数，如：

```
nice(-1);
```

3.4 守护进程

3.4.1 简介

守护进程是一种后台运行并且独立于所有终端控制之外的进程。UNIX/Linux 系统通常有许多的守护进程，它们执行着各种系统服务和管理的任务。

为什么需要有独立于终端之外的进程呢？首先，处于安全性的考虑我们不希望这些进程在执行中的信息在任何一个终端上显示。其次，我们也不希望这些进程被终端所产生的中断信号所打断。最后，虽然我们可以通过&将程序转为后台执行，我们有时也会需要程序能够自动将其转入后台执行。因此，我们需要守护进程。

3.4.2 守护进程的启动

要启动一个守护进程，可以采取以下的几种方式：

1．在系统期间通过系统的初始化脚本启动守护进程。这些脚本通常在目录 etc/rc.d 下，通过它们所启动的守护进程具有超级用户的权限。系统的一些基本服务程序通常都是通过这种方式启动的。

2．很多网络服务程序是由 inetd 守护程序启动的。在后面的章节中我们还会讲到它。它监听各种网络请求，如 telnet、ftp 等，在请求到达时启动相应的服务器程序（telnet server、ftp server 等）。

3．由 cron 定时启动的处理程序。这些程序在运行时实际上也是一个守护进程。

4．由 at 启动的处理程序。

5．守护程序也可以从终端启动，通常这种方式只用于守护进程的测试，或者是重起因某种原因而停止的进程。

6．在终端上用 nohup 启动的进程。用这种方法可以把所有的程序都变为守护进程，但在本节中我们不予讨论。

3.4.3 守护进程的错误输出

守护进程不属于任何的终端，所以当需要输出某些信息时，它无法像通常程序那样将信息直接输出到标准输出和标准错误输出中。这就需要某些特殊的机制来处理它的输出。为了解决这个问题，Linux 系统提供了 syslog()系统调用。通过它，守护进程可以向系统的 log 文件写入信息。它在 Linux 系统函数库 syslog.h 中的定义如下：

```
void syslog( int priority, char *format, ...);
```

该调用有两个参数：

priority 参数指明了进程要写入信息的等级和用途，可以的取值如表 3-2 所示：

表 3-2 priority 等级取值及其含义

等级	值	描述
LOG_EMERG	0	系统崩溃（最高优先级）
LOG_ALERT	1	必须立即处理的动作
LOG_CRIT	2	危急的情况
LOG_ERR	3	错误
LOG_WARNING	4	警告
LOG_NOTICE	5	正常但是值得注意的情况（缺省）
LOG_INFO	6	信息

LOG_DEBUG	7	调试信息（最低优先级）

如果等级没有被指定，就自动取缺省值 LOG_NOTICE。

表 3-3 是用途的类型：

表 3-3 priority 用途的取值及其含义

用途	描述
LOG_AUTH	安全/管理信息
LOG_AUTHPRIV	安全/管理信息（私人）
LOG_CRON	cron 守护进程
LOG_DAEMON	系统守护进程
LOG_FTP	ftp 守护进程
LOG_KERN	内核守护进程
LOG_LOCAL0	local use
LOG_LOCAL1	local use
LOG_LOCAL2	local use
LOG_LOCAL3	local use
LOG_LOCAL4	local use
LOG_LOCAL5	local use
LOG_LOCAL6	local use
LOG_LOCAL7	local use
LOG_LPR	行打印机系统
LOG_MAIL	mail 系统
LOG_NEWS	network news 系统
LOG_SYSLOG	syslogd 进程产生的信息
LOG_USER	随机用户信息（缺省）
LOG_UUCP	UUCP 系统

如果没有指定用途，缺省的 LOG_USER 就自动被指定。

syslog()调用后面的参数用法和 printf()类似，message 是一个格式串，指定了记录输出的格式。需要注意的是在这个串的最后需要指定一个%m，其对应着 errno 错误码。

下面是一个例子：

```
syslog(LOG_INFO|LOG_LOCAL2,"rename(%s,%s): %m",file1,file2);
```

在 etc/syslog.conf 中指定了各种信息存放的位置。例如，在 syslog.conf 中下面的一项：

```
local7.debug        /var/log/temp/log
```

表示系统将所有 LOG_DEBUG|LOG_LOCAL7 的信息都储存到/var/log/temp/log 中，这样可以方便信息的分类整理。

在一个进程使用 syslog()的时候，应该先使用 openlog()打开系统记录：

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);
```

参数 ident 是一个字符串，它将被加在所有用 syslog()写入的信息前。通常这个参数是程序的名字。

参数 options 可以是表 3-4 这些参数或是它们的或（|）的结果：

表 3-4 option 的取值及其含义

参数	描述
LOG_CONS	如果不能写入 log 信息，则直接将其发往主控台
LOG_NDELAY	直接建立与 syslogd 进程的连接而不是打开 log 文件
LOG_PERROR	将信息写入 log 的的同时也发送到标准错误输出
LOG_PID	在每个信息中加入 pid 信息。

参数 facility 指定了 syslog()调用的缺省用途值。

在使用完 log 之后，可以使用系统调用 closelog()来关闭它：

```
void closelog(void);
```

3.4.4 守护进程的建立

在介绍守护进程的建立之前，首先来看一下下面的这个例程 daemon_init() ，它演示了建立一个守护进程的全部过程：

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <syslog.h>

#define MAXFD 64

void daemon_init(const char *pname, int facility)
{
    int i:
    pid_t pid;

    /* fork,终止父进程 */
    if (pid=fork())
        exit(0);
    /* 第一子进程 */
    setsid();

    signal(SIGHUP,SIG_IGN);
    /* fork,终止第一子进程 */
    if (pid=fork())
        exit(0);
    /* 第二子进程 */
    daemon_proc=1;
    /* 将工作目录设定为"/" */
    chdir("/");
    /* 清除文件掩码 */
    umask(0);
    /* 关闭所有文件句柄 */
    for (i=0;i<MAXFD;i++)
    {
        close(i);
    }
    /* 打开 log */
    openlog(pname,LOG_PID,facility);
}
```

看过了上面的程序，下面我们就来讲讲建立一个守护进程需要进行哪些操作：

1．fork

首先需要 fork 一个子进程并将父进程关闭。如果进程是作为一个 shell 命令在命令行上前台启动的，当父进程终止时，shell 就认为该命令已经结束。这样子进程就自动称为了后台进程。而且，子进程从父进程那里继承了组标识符同时又拥有了自己的进程标识符，这样保证了子进程不会是一个进程组的首进程。这一点是下一步 setsid 所必须的。

2．setsid

setsid()调用创建了一个新的进程组，调用进程成为了该进程组的首进程。这样，就使该进程脱离了原来的终端，成为了独立于终端外的进程。

3．忽略 SIGHUP 信号，重新 fork

这样使进程不在是进程组的首进程，可以防止在某些情况下进程意外的打开终端而重新与终端发生联系。

4．改变工作目录，清除文件掩码

改变工作目录主要是为了切断进程与原有文件系统的联系。并且保证无论从什么地方启动进程都能正常的工作。清除文件掩码是为了消除进程自身掩码对其创建文件的影响。

5．关闭全部已打开的文件句柄

这是为了防止子进程继承了在父进程中打开的文件而使这些文件始终保持打开从而产生某些冲突。

6．打开 log 系统

以上就是建立一个守护进程的基本步骤。当然，一个实际的守护进程要比这个例子复杂许多，但是万变不离其宗，原理都是相同的。通过上面几步，我们可以正确的建立自己的守护进程。

3.5 本章小结

进程是 UNIX/Linux 系统的基本核心概念，具有十分重要的地位和作用。

在本章中，我们详细介绍了进程的建立和使用以及相关的问题。熟练的掌握这些知识，是 Linux 编程所必须的。希望读者能用心体会。

第四章　进程间通信

网络程序设计中通常包括两个或更多的需要互相对话（interprocess communications）的进程，因此进程通信的方法在网络程序设计中是极为重要的。网络程序设计在这个方面不同于一般程序设计通常所使用的方法。一个传统的程序可以通过全局变量或函数调用和不同的模块(甚至同一机器上的其它应用程序)对话，但是在网络上却不行。

网络程序设计的一个重要的目标是保证进程间不互相干涉，否则系统可能被挂起或自锁，因此，进程间必须使用简洁有效的方法进行通信，在此方面，Linux 具有非常显著的兼容性。因为 Linux 的许多基本性能如管道，队列等都非常适合网络。

在这一章中，我们将详细的介绍各种进程间通信的方法及其使用方式。

4.1　进程间通信的一些基本概念

下面是一些在学习进程间通信时会遇到的基本概念：

- 进程阻塞

当一个进程在执行某些操作的条件得不到满足时，就自动放弃 CPU 资源而进入休眠状态，以等待条件的满足。当操作条件满足时，系统就将控制权返还给该进程继续进行未完的操作。

- 共享资源

因为计算机的内存、存储器等资源是有限的，无法为每一个进程都分配一份单独的资源。所以系统将这些资源在各个进程间协调使用，称为共享资源。

- 锁定

当某个进程在使用共享资源使用，可能需要防止别的进程对该资源的使用。比如，一个进程在对某个文件进行读操作时，如果别的进程也在此时向文件中写入了内容，就可能导致进程读入错误的数据。为此，Linux 提供一些方法来保证共享资源在被某个进程使用时，别的进程无法使用。这就叫做共享资源的锁定。

4.2　信号

信号是 UNIX 系统所使用的进程通信方法中，最古老的一种。系统使用它来通知一个或多个进程异步事件的发生，比如键盘上某个键被按下，或者计时器到达了某个特定的事件。系统也用信号来处理某种严重的错误。比如，一个进程试图向一块不存在的虚拟内存写入数据，或者某个进程试图执行一条非法指令。

信号不但能从内核发往一个进程，也能从一个进程发往另一个进程。例如，用户在后台启动了一个要运行较长时间的程序，如果想中断其执行，可以用 kill 命令把 SIGTERM 信号发送给这个进程，SIGTERM 将终止此进程的执行。

信号还提供了向 UNIX 系统进程传送软中断的简单方法。信号可以中断一个进程，而不管它正在作什么工作。由于信号的特点，所以不用它来作进程间的直接数据传送，而把它用作对非正常情况的处理。

由于信号本身不能直接携带信息，这就限制了它作为一项通用的进程通信机制。但是，

每种信号都有其特定的含义，并由其名字所指示。在 Linux 系统库 bits/signum.h 中对这些信号名作了定义，每个名字代表一个正整数。例如：

```
#define SIGHUP          1       /* Hangup (POSIX).  */
```

定义了信号 SIGHUP。

Linux 提供的大多数信号类型是供内核使用的，只有少数的几种信号可以用作在进程之间传送。下面给出常用的信号和它们的意义：

- SIGHUP

当终止一个终端时，内核就把这一种信号发送给该终端所控制的所有进程。通常情况下，一个进程组的控制终端是该用户拥有的终端，但不完全是如此。当进程组的首进程结束时，就会向该进程组的所有进程发送这种信号。这就可以保证当一个用户退出使用时，其后台进程被终止，除非有其它方面的安排。

- SIGINT

当一个用户按了中断键（一般为 Ctrl+C）后，内核就向与该终端有关联的所有进程发送这种信号。它提供了中止运行程序的简便方法。

- SIGQUIT

这种信号与 SIGINT 非常相似，当用户按了退出键时（为 ASCII 码 FS，通常为 Ctrl+\），内核就发送出这种信号。SIGQUIT 将形成 POSIX 标准所描述的非正常终止。我们称这种 UNIX 实现的实际操作为核心转贮（core dump），并用信息“ Quit (core dump) ”指出这一操作的发生。这时，该进程的映象被转贮到一个磁盘文件中，供调试之用。

- SIGILL

当一个进程企图执行一条非法指令时，内核就发出这种信号。例如，在没有相应硬件支撑的条件下，企图执行一条浮点指令时，则会引起这种信号的发生。SIGILL 和 SIGQUIT 一样，也形成非正常终止。

- SIGTRAP

这是一种由调试程序使用的专用信号。由于他的专用行和特殊性，我们不再对它作进一步的讨论。SIGTRAP 也形成非正常终止。

- SIGFPE

当产生浮点错误时（比如溢出），内核就发出这种信号，它导致非正常终止。

- SIGKILL

这是一个相当特殊的信号，它从一个进程发送到另一个进程，使接收到该信号的进程终止。内核偶尔也会发出这种信号。SIGKILL 的特点是，它不能被忽略和捕捉，只能通过用户定义的相应中断处理程序而处理该信号。因为其它的所有信号都能被忽略和捕捉，所以只有这种信号能绝对保证终止一个进程。

- SIGALRM

当一个定时器到时的时候，内核就向进程发送这个信号。定时器是由改进程自己用系统调用 alarm()设定的。

- SIGTERM

这种信号是由系统提供给普通程序使用的，按照规定，它被用来终止一个进程。

- SIGSTOP

这个信号使进程暂时中止运行，系统将控制权转回正在等待运行的下一个进程。

- SIGUSR1 和 SIGUSR2

和 SIGTERM 一样，这两种信号不是内核发送的，可以用于用户所希望的任何目的。

- SIGCHLD

子进程结束信号。UNIX 中用它来实现系统调用 exit()和 wait()。执行 exit()时，就向子

进程的父进程发送 SIGCHLD 信号，如果这时父进程政在执行 wait()，则它被唤醒；如果这时候父进程不是执行 wait()，则此父进程不会捕捉 SIGCHLD 信号，因此该信号不起作用，子进程进入过渡状态（如果父进程忽略 SIGCHLD，子进程就结束而不会进入过渡状态）。这个机制对大多数 UNIX 程序员来说是相当重要的。

对于大多数情况来说，当进程接收到一个信号时，它就被正常终止，相当于进程执行了一个临时加入的 exit()调用。在这种情况下，父进程能从进程返回的退出状态中了解可能发生的事情，退出状态的低 8 位含有信号的号码，其高 8 位为 0。

信号 SIGQUIT、SIGILL、SIGTRAP、SIGSYS 和 SIGFPE 会导致一个非正常终止，它们将发生核心转贮，即把进程的内存映象写入进程当前目录的 core 文件之中。core 文件中以二进制的形式记录了终止时程序中全部变量之值、硬件寄存器之值和内核中的控制信息。非正常终止进程的退出状态除了其低端第 7 位被置位外，其它均与通过信号正常终止时一样。

Linux 的调试程序 gdb 知道 core 文件的格式，可以用它们来观察进程在转贮点上的状态。这样，就可以用 gdb 正确的定出发生问题的位置。

这里再介绍一下系统调用 abort()，它在 Linux 系统库 stdlib.h 中定义：

```
void abort(void);
```

abort()向调用进程发送一个信号，产生一个非正常终止，即核心转贮。由于它能够使一个进程在出错时记录进程的当前状态，所以可以用它来作为调试的辅助手段。这也说明了进程可以向自己发送信号这一事实。

4.2.1 信号的处理

几乎所有的信号都将终止接收到该信号的进程。对于一些简单的程序，这完全能满足要求。用户按了中断或者退出键，就可以停止一个有问题的程序的运行。但是在大型的程序中，一些意料之外的信号会导致大问题。例如，正当在对一个重要的数据库进行修改期间，由于不小心碰到了中断键，而使程序被意外的终止，从而产生严重的后果。

UNIX 的系统调用 signal()用于接收一个指定类型的信号，并可以指定相应的方法。这就是说，signal()能够将指定的处理函数与信号向关联。它在 Linux 系统库 signal.h 中的函数声明如下：

```
int signal (int sig, __sighandler_t handler);
```

Signal()有两个参数：

第一个参数 sig 指明了所要处理的信号类型，它可以取除了 SIGKILL 和 SIGSTOP 外的任何一种信号。参数 handler 描述了与信号关联的动作，它可以取以下三种值：

- 一个返回值为整数的函数地址。

此函数必须在 signal()被调用前声明，handler 中为这个函数的名字。当接收到一个类型为 sig 的信号时，就执行 handler 所指定的函数。这个函数应有如下形式的定义：

```
int func(int sig);
```

sig 是传递给它的唯一参数。执行了 signal()调用后，进程只要接收到类型为 sig 的信号，不管其正在执行程序的哪一部分，就立即执行 func()函数。当 func()函数执行结束后，控制权返回进程被中断的那一点继续执行。

- SIG_IGN

这个符号表示忽略信号。执行了相应的 signal()调用好，进程会忽略类型为 sig 的信号。

- SIG_DFL

这个符号表示恢复系统对信号的默认处理。

函数如果执行成功，就返回信号在此次 signal()调用之前的关联。

如果函数执行失败，就返回 SIG_ERR。通常这种情况只有当 sig 参数不是有效的信号时才会发生。函数不对 handler 的有效性进行检查。

下面我们来看几个例子。

首先，下面的这段代码则将使进程忽略 SIGINT 信号：

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

int main(void)
{
        signal(SIGINT,SIG_IGN); /*告诉进程将 SIGINT 信号忽略*/
        printf("xixi\n");
        sleep(10); /*系统函数 sleep()使进程休眠指定的时间（以秒为单位）*/
        printf("end\n");
        return;
}
```

如果在程序中需要重新恢复系统对信号的缺省处理，就使用下面的语句：

```
signal(SIGINT,SIG_DFL);
```

在 Linux 程序中常常利用 SIG_IGN 和 SIG_DFL 屏蔽 SIGINT 和 SIGQUIT 来保证执行重要任务的程序不会被意外的中止。

在 shell 中也是利用这一技术来确保用户按中断键时，不中断后台程序的运行。因为被一个进程忽略的信号，在进程执行 exec()调用后，仍然被忽略，所以 shell 能够调用 signal() 来保证 SIGQUIT 和 SIGINT 被忽略，然后用 exec 执行新程序。但是要注意到，在父进程中设定的信号和函数的关联关系会被 exec()调用自动用 SIG_DFL 恢复成系统的缺省动作，这是因为在 exec 的子进程中没有父进程的函数映象。

再让我们来看看下面这段捕捉 SIGINT 的代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

int catch(int sig);

int main(void)
{
        signal(SIGINT,catch); /* 将 SIGINT 信号与 catch 函数关联 */
        printf("xixi\n");
        sleep(10);
        printf("end\n");
        return;
}

int catch(int sig)
{
```

```
        printf("Catch succeed!\n");
        return 1;
}
```

当程序运行时我们按下中断键（Ctrl+C），进程被中断，函数 catch 就被执行。它执行完毕后，进程回到中断点继续执行。

如果我们希望一个进程被信号终止前能够完成一些处理工作，如删除工作中使用的临时文件等，就可以设计一个信号处理函数来完成工作。比如可以有这样的一个处理函数：

```
int catch(int sig)
{
        printf("Catch succeed!\n");
        exit(1);
}
```

这个函数在最后调用 exit()函数来使进程结束运行。这样就保留了信号原有的中断进程的功能。

这里需要指出一点，当程序把一个信号处理函数与 SGINT 和 SIGQUIT 联系起来后，如果该程序在后台执行，那么由于 shell 的作用，会使得 SIGINT 和 SIGQUIT 被忽略。这样后台程序就不会被 SIGINT 和 SIGOUT 所中止。

前面曾经提过，signal()调用返回原先与指定信号相关联的处理函数，这样，我们就可以保存和恢复原来对指定信号的处理动作。下面的代码说明这一技术：

```
int (*oldptr)(),newcatch();
/*  设定 SIGINT 的关联，同时保存原来的关联*/
oldptr=singal(SIGINT,newcatch);
/*  工作代码段  */
…..

/*  恢复原来的关联  */
signal(SIGINT,oldptr);
```

4.2.2 信号与系统调用的关系

当一个进程正在执行一个系统调用时，如果向该进程发送一个信号，那么对于大多数系统调用来说，这个信号在系统调用完成之前将不起作用，因为这些系统调用不能被信号打断。但是有少数几个系统调用能被信号打断，例如：wait(),pause()以及对慢速设备（终端、打印机等）的 read()、write()、open()等。如果一个系统调用被打断，它就返回-1，并将 errno 设为 EINTR。可以用下列代码来处理这种情况：

```
        if (wirte(tfd,buf,SIZE)<0) {
            if (errno==EINTR) {
                warn("Write interrupted.");
                …
                …
            }
        }
```

4.2.3　信号的复位

在 Linux 中，当一个信号的信号处理函数执行时，如果进程又接收到了该信号，该信号会自动被储存而不会中断信号处理函数的执行，直到信号处理函数执行完毕再重新调用相应的处理函数。下面的程序演示了这一点：

```
#include <signal.h>

    int interrupt()
    {
        printf("Interrupt called\n");
        sleep(3);
        printf("Interrupt Func Ended.\n");
    }

    main()
    {
        signal(SIGINT,interrupt);
        printf("Interrupt set for SIGINT\n");
        sleep(10);
        printf("Program NORMAL ended.\n");
        return;
    }
```

执行它，结果如下：

```
        Interrupt set for SIGINT
        <ctrl+c>
        Interrupt called
        <ctrl+c>
        Func Ended
        Interrupt called
        Func Ended
        Program NORMAL ended.
```

但是如果在信号处理函数执行时进程收到了其它类型的信号，该函数的执行就会被中断：

```
#include <signal.h>

    int interrupt()
    {
        printf("Interrupt called\n");
        sleep(3);
        printf("Interrupt Func Ended.\n");
    }

    int catchquit()
    {
```

```
        printf("Quit called\n");
        sleep(3);
        printf("Quit ended.\n");
}
main()
{
        signal(SIGINT,interrupt);
        signal(SIGQUIT,catchquit);
        printf("Interrupt set for SIGINT\n");
        sleep(10);
        printf("Program NORMAL ended.\n");
        return;
}
```

执行这个程序的结果如下：

```
Interrupt set for SIGINT
<ctrl+c>
Interrupt called
<ctrl+\>
Quit called
Quit ended.
Interrupt Func Ended.
Program NORMAL ended.
```

还要注意的是，在 Linux **系统中同种信号是不能积累的。比如我们执行上面的代码：**

```
Interrupt set for SIGINT
<ctrl+c>
Interrupt called
<ctrl+c><ctrl+c><ctrl+c>
Func Ended
Interrupt called
Func Ended
Program NORMAL ended.
```

而且如果两个信号同时产生，系统并不保证进程接收它们的次序。以上的两个缺点影响了信号作为进程通信手段的可靠性，因为一个进程不能保证它发出的信号不被丢失。

当某个信号未被处理的时候，如果对该信号执行 signal **调用，那么该信号将被注销。**

4.2.4 在进程间发送信号

一个进程通过对 signal()**的调用来处理其它进程发送来的信号。同时，一个进程也可以向其它的进程发送信号。这一操作是由系统调用** kill()**来完成的。**kill()**在** linux **系统库** signal.h **中的函数声明如下：**

```
int kill(pid_t pid, int sig);
```

参数 pid **指定了信号发送的对象进程：它可以是某个进程的进程标识符**(pid)**，也可以是以下的值：**

如果 pid **为零，则信号被发送到当前进程所在的进程组的所有进程；**

如果 pid 为-1，则信号按进程标识符从高到低的顺序发送给全部的进程（这个过程受到当前进程本身权限的限制，请看后面的解释）；

如果 pid 小于-1,则信号被发送给标识符为 pid 绝对值的进程组里的所有进程。

需要说明的是，一个进程并不是向任何进程均能发送信号的，这里有一个限制，就是普通用户的进程只能向具有与其相同的用户标识符的进程发送信号。也就是说，一个用户的进程不能向另一个用户的进程发送信号。只有 root 用户的进程能够给任何线程发送信号。

参数 sig 指定发送的信号类型。它可以是任何有效的信号。

由于调用 kill()的进程需要直到信号发往的进程的标识符，所以这种信号的发送通常只在关系密切的进程之间进行，比如父子进程之间。

下面是一个使用 kill()调用发送信号的例子。这个程序建立两个进程，并通过向对方发送信号 SIGUSR1 来实现它们之间的同步。这两个进程都处于一个死循环中，在接收对方发送的信号之前，都处于暂停等待中。这是通过系统调用 pause()来实现的，它能够使一个程序暂停，直至一个信号到达，然后进程输出信息，并用 kill 发送一个信号给对方。当用户按了中断键，这两个进程都将终止。

```
#include <signal.h>

int ntimes=0;

main()

{

    int pid,ppid;
    int p_action(), c_action();

    /*  设定父进程的 SIGUSR1 */
    signal(SIGUSR1,p_action);

    switch(pid=fork()) {
        case -1:   /*fork 失败*/
            perror("synchro");
            exit(1);
        case 0:    /*子进程模块*/
        /*  设定子进程的 SIGUSR1 */
        signal(SIGUSR1,c_action);

        /*  获得父进程的标识符  */
        ppid=getppid();

        for(;;) {
            sleep(1);
            kill(ppid,SIGUSR1);
            pause();
        }
```

```
            /*死循环*/
            break;

            default:   /*父进程模块*/

            for (;;) {
                pause();
                sleep(1);
                kill(pid,SIGUSR1);
            }
            /*死循环*/
        }
}

p_action()
{
    printf("Patent caught signal #%d\n",++ntimes);
}

c_action()
{
    printf("Child caught signal #%d\n",++ntimes);
}
```

程序运行结果如下：

```
Patent caught signal #1
Child caught signal #1
Patent caught signal #2
Child caught signal #2
Patent caught signal #3
Child caught signal #3
Patent caught signal #4
Child caught signal #4
<ctrl+c>
```

这里顺便介绍一下 kill **命令，它是一个对系统调用** kill()**的命令层接口。**kill **命令用于向一个运行进程发送信号，它发送的信号默认为** SIGTERM**，但是也可以指定为其它信号。我们可以直接用信号的号码来指定** kill **命令所发送信号之类型，也可以用符号名指定。比如可以用下面的命令来完成向进程标识符为** 1234 **的进程发送** SIGINT **信号：**

```
kill –s SIGINT 1234
```

4.2.5 系统调用 alarm()和 pause()

1．**系统调用** alarm()

alarm()**是一个简单而有用的系统调用，它可以建立一个进程的报警时钟，在时钟定时**

器到时的时候，用信号向程序报告。alarm()系统调用在 Linux 系统函数库 unistd.h 中的函数声明如下：

unsigned int alarm(unsigned int seconds);

函数唯一的参数是 seconds，其以秒为单位给出了定时器的时间。当时间到达的时候，就向系统发送一个 SIGARLM 信号。例如：

alarm(60);

这一调用实现在 60 秒后发一个 SIGALRM 信号。alarm 不会象 sleep 那样暂停调用进程的执行，它能立即返回，并使进程继续执行，直至指定的延迟时间到达发出 SIGALRM 信号。事实上，一个由 alarm()调用设置好的报警时钟，在通过 exec()调用后，仍将继续有效。但是，它在 fork()调用后中，在子进程中失效。

如果要使设置的报警时钟失效，只需要调用参数为零的 alarm()：

alarm(0)

alarm()调用也不能积累。如果调用 alarm 两次，则第二次调用就取代第一次调用。但是，alarm 的返回值柜橱了前一次设定的报警时钟的剩余时间。

当需要对某项工作设置时间限制时，可以使用 alarm()调用来实现。其基本方法为：先调用 alarm()按时间限制值设置报警时钟，然后进程作某一工作。如果进程在规定时间以内完成这一工作，就再调用 alarm(0)使报警时钟失效。如果在规定时间内未能完成这一工作，进程就会被报警时钟的 SIGALRM 信号中断，然后对它进行校正。

下面这个程序使用上述方法来强制用户作出回答。在其中包括一个 quickreply()函数，它有一个参数 prompt，它是一个指向提示字符串的指针。quickreply 的返回值也是一个指针。它指向含有输入行信息的字符串。这个例行程序在试作五次之后，如果仍未得到输入信息，就返回一个 null 指针。每当 quickreply 要提醒用户时，它就向终端发送 ASCII 码 007，这会使终端响铃。

quickreply 调用了标准 I/O 库中的例行程序 gets()。gets()把标准输入上的下一行信息存入一个字符型数组，它返回一个指向该数组的指针。当到达文件末或出错时，gets 则返回一个 null 指针。函数 catch 是信号 SIGALRM 的关联函数，它完成对此信号的处理。catch 设置了一个 timed_out 标志，在 quickreply 中对这个标志进行检查，看它是否超过了规定的时限。

```
#include <stdio.h>
#include <signal.h>

#define TIMEOUT 5
#define MAXTRIES 5
#define LINESIZE 100
#define BELL     '\007'
#define TRUE 1
#define FALSE 0

/* 判断超时是否已经发生的标志 */
static int time_out;

static char inputline[LINESIZE];
char* quickreply (char* prompt);
```

```
main()
{
    printf("%s\n",quickreply("Input"));
}

char* quickreply (char* prompt)
{
    int (*was)(),catch(),ntries;
    char* answer;

    /* 设定捕捉 SIGALRM 的的关联并保存原有关联 */
    was=signal(SIGALRM,catch);

    for (ntries=0;ntries<MAXTRIES;ntries++)
    {
        time_out=FALSE;
        printf("\n%s>",prompt);

        /* 设定定时器 */
        alarm(TIMEOUT);

        /* 获取输入 */
        answer=gets(inputline);

        /* 关闭定时器 */
        alarm(0);

        if (!time_out)
            break;
    }

    /* 恢复原有的 SIGALRM 关联 */
    signal(SIGALRM,was);

    return (time_out?((char*) 0):answer);
}
/* SIGALRM 信号处理函数 */
catch()
{
    /* 设定超时标志 */
    time_out=TRUE;

    /* 响铃警告 */
    putchar(BELL);
```

```
}
```

2．**系统调用** pause()

系统调用 pause()**能使调用进程暂停执行，直至接收到某种信号为止。**pause()**在** Linux **系统函数库** unistd.h **中的函数声明如下：**

```
int pause(void);
```

该调用没有任何的参数。它的返回始终是 -1 **，此时** errno **被设置为** ERESTARTNOHAND**。**

下面这个程序为了在规定时间显示一个消息，使用了 alarm **和** pause**。对它的调用方法如下：**

```
$tml minutes message-text &
```

第一个参数为时间数，第二个参数为显示的消息。

```
#include <stdio.h>
#include <signal.h>

#define TRUE 1
#define FALSE 0
#define BELLS "\007\007\007"

int alarm_flag=FALSE;

/* SIGALRM 处理函数 */
setflag()
{
    alarm_flag=TRUE;
}

main(int argc,char* argv[])
{
    int nsecs;
    int i;

    if (argc<2)
    {
        fprintf(stderr,"Usage:tml #minutes message\n");
        exit(1);
    }
    if ((nsecs=atoi(argv[1])*60)<=0)
    {
        fprintf(stderr,"Invalid time\n");
        exit(2);
    }

    /* 设定 SIGALRM 的关联动作 */
    signal(SIGALRM,setflag);
```

```
    /* 设定定时器 */
    alarm(nsecs);

    /*使用 pause()调用等待信号*/
    pause();

    if (alarm_flag)
    {
        printf(BELLS);
        for (i=2;i<argc;i++)
        {
            printf("%s\n",argv[i]);
        }
    }
    exit(0);
}
```

4.2.6 系统调用 setjmp()和 longjmp()

有时候，当接收到一个信号时，希望能跳回程序中以前的一个位置执行。例如，在有的程序内，当用户按了中断键，则程序跳回到显示主菜单执行。我们可以用库系统调用 setjmp()和 longjmp()来完成这项工作。setjmp()能保存程序中的当前位置（是通过保存堆栈环境实现的），longjmp()能把控制转回到被保存的位置。在某种意义上，longjmp()是远程跳转，而不是局部区域内的跳转。我们必须注意到，由于堆栈已经回到被保存位置这一点，所以 longjmp()从来不返回。然而，与其对应的 setjmp()是要返回的。

setjmp()和 longjmp()在 setjmp.h 中的定义分别如下：

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

setjmp()只有一个参数 env，用来保存程序当前位置的堆栈环境。而 longjmp()有两个参数：

参数 env 是由 setjmp()所保存的堆栈环境。

参数 val 设置 setjmp()的返回值。longjmp()本身是没有返回的，但其执行后跳转到保存 env 参数的 setjmp()调用，并由 setjmp()调用返回，就好像程序刚刚执行完 setjmp()一样，此时 setjmp()的返回值就是 val。但是要注意的是，longjmp()调用不能使 setjmp()调用返回 0，如果 val 为 0，则 setjmp()的返回为 1。

下面的例子演示了 setjmp()和 longjmp()的使用：

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

jmp_buf position;

main()
```

```
{

    int goback();
        …
        …
    /*  保存当前的堆栈环境  */
    setjmp(position);

    signal(SIGINT,goback);

    domenu();
        …
        …
}

goback()
{

    fprintf(stderr,"\nInterrupted\n");
    /*  跳转回被保存的断点  */
    longjmp(position,1);

}
```

4.3　管道

简单的说，管道就是将一个程序的输出和另外一个程序的输入连接起来的单向通道。它是 UNIX/Linux 系统的各种进程通信方法中，最古老而应用最为广泛的一种（特别是在 shell 中）。

```
#ls -l|more
```

在上面的例子中我们建立了这样的一个管道：获取 ls -l 的输出，再将其作为 more 命令的输入。形象的说，就是数据沿着管道从管道的左边流到了管道的右边。

这个例子并不复杂，只要是对 Linux/UNIX 比较熟悉的人都曾经使用过类似的命令。但是，在简单的命令底下，Linux/UNIX 内核究竟都做了些什么呢？

当进程创建一个管道的时候，系统内核同时为该进程设立了一对文件句柄（一个流），一个用来从该管道获取数据（read），另一个则用来做向管道的输出（write）。

图 4-1 显示了进程和管道间的相互作用。

从图 4-1 中可以清楚的看出进程和管道是如何通过句柄进行数据交换的。进程通过句柄 fd[0]向管道写入（write）数据，同时通过 fd[1]从管道读出（read）数据。到这里有人也许会想起 UNIX 的文件处理。事实上，在 Linux 系统内核里，每个管道都是用一个 inode 节点来表示的。（当然，你是不会看到这个节点的，它只存在于系统的内核中。）理解了这一点，我们就可以容易的掌握接下来要讲的管道的 I/O 处理了。

不过，到目前为止，我们所建立的管道模型还没有任何的实际意义。因为这个管道只被用来同单个进程通信。建立一个同自己通信的管道有什么用处呢？为了解决这个问题，

我们在主进程中利用 fork()函数创建一个自身的自进程。大家也许还记得，fork()的子进程自动继承了父进程打开的文件句柄。利用继承的句柄，就可以实现父/子间的通信了。这个关系可以用图 4-2 来表示：

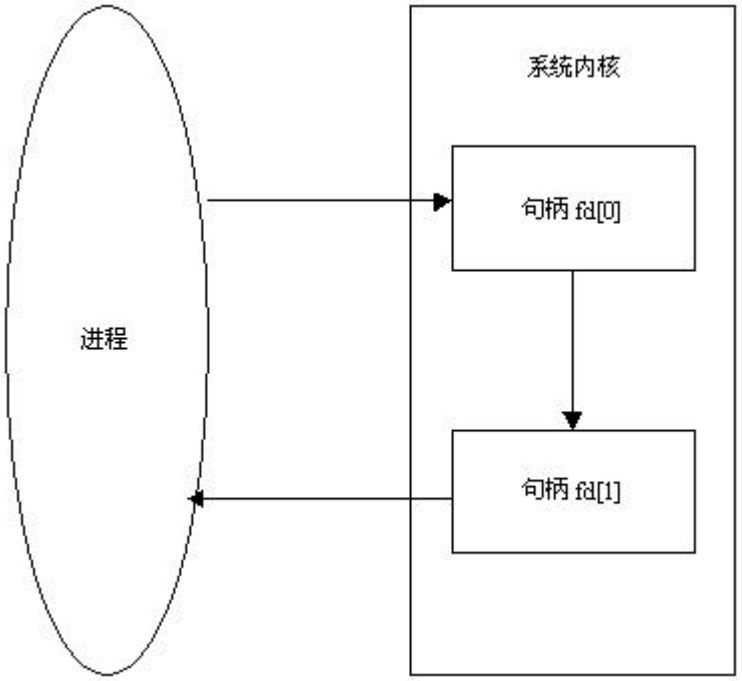

图 4-1 进程和管道间的相互作用

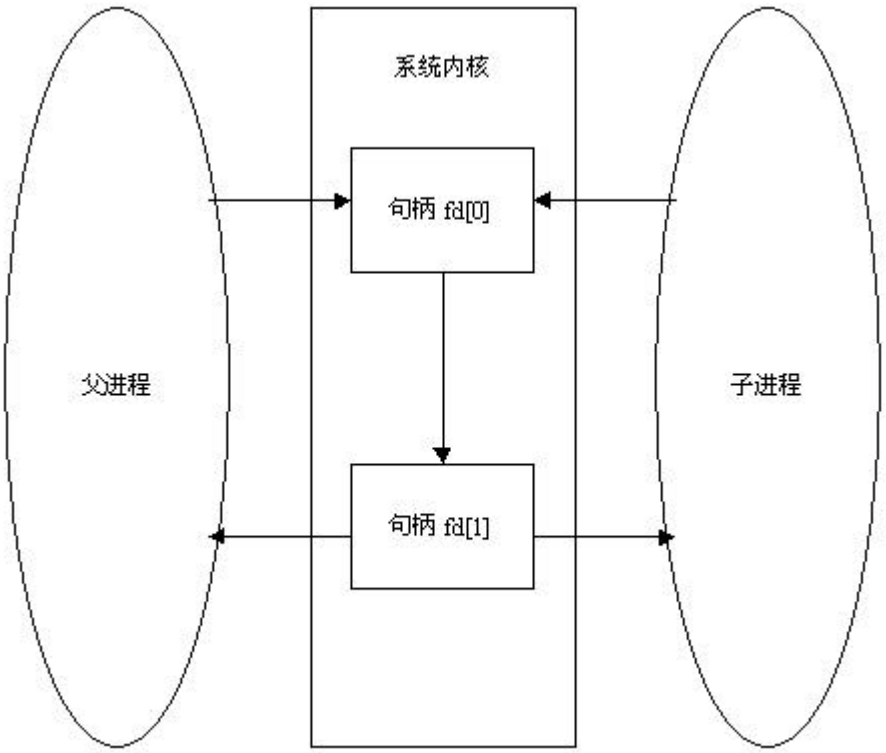

图 4-2 fork()函数调用后的管道

现在，父子两个进程同时拥有对同一个管道的读写句柄。因为管道必须是单向的（因为它没有提供锁定的保护机制），所以我们必须决定数据的流动方向（从父到子，还是从子到父？），然后在每个进程中关闭不需要的句柄。假设我们需要管道从子进程向父进程传送数据，关闭了相应句柄后的管道可以用图 4-3 来表示。

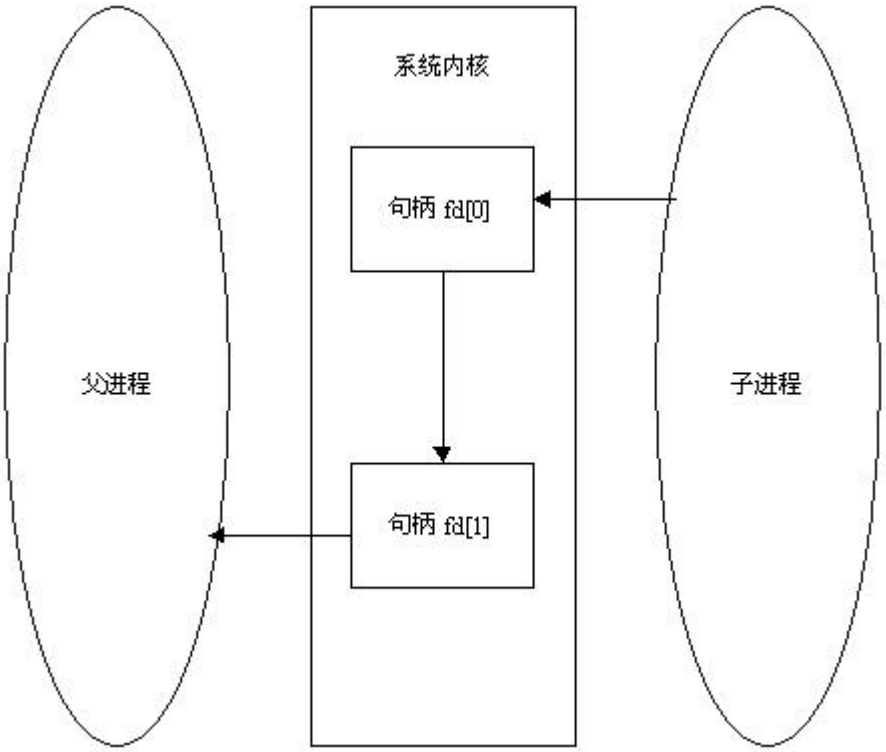

图 4-3　关闭了相应句柄后的管道

这样，一个完整的管道就被建立了。下面我们就可以使用 read()和 write()函数来对它进行读写操作了。关于这两个函数的具体使用在 UNIX 的文件函数中有介绍，读者可以自行参阅相应的资料。

4.3.1　用 C 来建立、使用管道

以上我们介绍了管道的概念和它在 Linux 系统中的模型。下面，我们就将开始用 C 来建立自己的管道。

1．Pipe()函数

在 C 程序中，我们使用系统函数 pipe()来建立管道。它只有一个参数：一个有两个成员的整型数组，用于存放 pipe()函数新建立的管道句柄。其函数原型如下：

系统调用： pipe();
函数声明： int pipe(int fd[2]);
返回值： 0 on success
-1 on error: errno = EMFILE (no free descriptors)
EMFILE (system file table is full)
EFAULT (fd array is not valid)

注意: fd[0] **用来从管道中读**, fd[1] **用来向管道中写**

数组中的第一个元素（fd[0]）是从管道中读出数据的句柄，第二个元素（fd[1]）是向管道写入数据的句柄。也即是说，fd[1]的写入由 fd[0]读出。

在建立了管道之后，我们使用 fork()函数建立一个子线程：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>


main()
{
int         fd[2];
pid_t       childpid;

pipe(fd);

if((childpid = fork()) == -1)
{
perror("fork");
exit(1);
}

…
…
}
```

接下来，我们假设需要管道中数据的流动是从子进程到父进程。这样父进程就需要关闭（close()）写管道的句柄（fd[1]），而子进程需要关闭读管道的进程（fd[0]）。

注意：因为父子进程同时拥有读写句柄，为了避免不必要的麻烦，我们在程序中务必要记住关闭不需要的句柄！

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
 {
 int         fd[2];
 pid_t       childpid;

 pipe(fd);

 if((childpid = fork()) == -1)
 {
        perror(“fork”);
        exit(1);
```

```
}

if(childpid == 0)
{
      /*  子进程关闭管道的读句柄  */
       close(fd[0]);
}
else
{
      /*  父进程关闭管道的写句柄  */
       close(fd[1]);
}                          .

……
……

}
```

管道建立之后，我们就可以像操作普通文件一样对其进行操作：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
int        fd[2],  nbytes;
pid_t      childpid;
char       string[] = "Hello, world!\n";
char       readbuffer[80];

pipe(fd);
if((childpid = fork()) == -1)
{
      perror("fork");
      exit(1);
      }

if(childpid == 0)
{
      /*  子进程关闭管道的读句柄  */
     close(fd[0]);
      /*  通过写句柄向管道写入信息  */
       write(fd[1], string, strlen(string));
_exit(0);
  }
```

```
    else
  {
    /* 父进程关闭管道的写句柄 */
    close(fd[1]);
    /* 通过读句柄从管道读出信息 */
    nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
    printf("Received string: %s", readbuffer);
      }

    return(0);

  }
```

2．使用 dup()函数

有时候我们需要将子进程当中的管道的句柄定向到标准 I/O（stdin/stdout）上去。这样，在子进程中使用 exec()函数调用外部程序时，这个外部程序就会将管道作为它的输入/输出。这个过程可以用系统函数 dup()来实现。其函数原型如下：

系统调用： dup();
函数声明： int dup(int oldfd);
返回值： new descriptor on success
-1 on error: errno = EBADF (oldfd is not a valid descriptor)
EBADF (newfd is out of range)
EMFILE (too many descriptors for the process)

注意: 旧句柄没有被关闭，新旧两个句柄可以互换使用

虽然原句柄和新句柄是可以互换使用的，但为了避免混淆，我们通常会将原句柄关闭（close）。同时要注意，在 dup()函数中我们无法指定重定向的新句柄，系统将自动使用未被使用的最小的文件句柄（记住，句柄是一个整型量）作为重定向的新句柄。请看下面的例子：

```
        ……
        ……
  pipe(fd);

  childpid = fork();
  if(childpid == 0)
        {

  /* 关闭子进程的文件句柄 0(stdin) */
  close(0);

  /* 将管道的读句柄定义到 stdin */
      dup(fd[0]);
      execlp("sort", "sort", NULL);
              ……
       }
```

在上例中巧妙的利用了 dup()函数的特性。因为文件句柄 0（stdin）被关闭了，对 dup 函数的调用就将管道读句柄 fd[0]定向到了 stdin（因为句柄 0 是最小的未用句柄）。然后我们调用 execlp 函数，用外部过程 sort 覆盖了子进程的代码。因为它继承了子进程的基本输

入/输出，所以它就将管道作为了它的输入。现在，我们在父进程里向管道写入的任何数据都将自动被 sort 接受进行排序……

3．使用 dup2()函数

在 Linux 系统中还有一个系统函数 dup2()。单从函数名上我们也可以判断出它和 dup()函数的渊源。下面是它的原型：

系统调用：　dup2();
函数声明：　int dup2(int oldfd, int newfd);
返回值：　new descriptor on success
-1 on error: errno = EBADF (oldfd is not a valid descriptor)
EBADF (newfd is out of range)
EMFILE(too many descriptors for the process)
注意: 旧句柄将被 dup2()自动关闭

显然，原来的 close 以及 dup 这一套调用现在全部由 dup2()来完成。这样不仅简便了程序，更重要的是，它保证了操作的独立性和完整性，不会被外来的信号所中断。在原来的 dup()调用中，我们必须先调用 close()函数。假设此时恰好一个信号使接下来的 dup()调用不能立即执行，这就会引发错误（进程没有了 stdin）。使用 dup2()就不会有这样的危险。下面的例子演示了 dup2()函数的使用：

```
……

pipe(fd);

childpid = fork();

if(childpid == 0)
{
        /*  将管道的读入端定向到 stdin */
        dup2(0, fd[0]);
        execlp("sort", "sort", NULL);
        ……
}
```

4．使用 popen()/pclose()函数

看了 dup2()函数，一定有人会想，既然能把 close 和 dup 合成一个函数，那么有没有把 fork、exec 和 dup()结合的函数呢？答案是肯定的。它就是 linux 的系统函数 popen()：

库函数：　popen();
函数声明：　FILE *popen (char *command, char *type);
返回值：　new file stream on success
NULL on unsuccessful fork() or pipe() call
NOTES：　creates a pipe, and performs fork/exec operations using "command"

popen()函数首先调用 pipe()函数建立一个管道，然后它用 fork()函数建立一个子进程，运行一个 shell 环境，然后在这个 shell 环境中运行"command"参数指定的程序。数据在管道中流向由"type"参数控制。这个参数可以是"r"或者"w"，分别代表读和写。需要注意的是，"r"和"w"两个参数不能同时使用！在 Linux 系统中，popen 函数将只使用"type"参数中第一个字符，也就是说，使用"rw"和"r"作为"type"参数的效果是一样的，管道将只打开成读状态。

使用 popen 打开的管道必须用 pclose()函数来关闭。还记得 fopen 和 fclose 的配对使用吗？这里再次显示了管道和文件的相似性。

库函数：　pclose();
函数声明：　int pclose(FILE *stream);

返回值： exit status of wait4() call
-1 if "stream" is not valid, or if wait4() fails
NOTES: waits on the pipe process to terminate, then closes the stream.

下面是一个使用 popen/pclose 的例子：

```
#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
int    cntr;
FILE *pipe_fp;
char *strings[MAXSTRS] = { "roy", "zixia", "gouki","supper", "mmwan"};

            /*  用 popen 建立管道  */
if (( pipe_fp = popen("sort", "w")) == NULL)
{
      perror("popen");
exit(1);
}

           /* Processing loop */
for(cntr=0; cntr<MAXSTRS; cntr++)
{
     fputs(strings[cntr], pipe_fp);
     fputc('\n', pipe_fp);
}

/*  关闭管道  */
pclose(pipe_fp);
return(0);
}
```

使用 popen()函数除了节省源代码之外，它还有一个优点：你可以在"command"中使用任意合法的 shell 指令，包括重定向和管道！下面的几个例子都是合法的 popen 调用：

```
popen("ls ~roy", "r");
popen("sort > /tmp/zixia", "w");
popen("sort | uniq | more", "w");
```

下面是一个稍微复杂一点的例子，在里面建立了两个管道：

```
#include <stdio.h>

int main(void)
{
            FILE *pipein_fp, *pipeout_fp;
            char readbuf[80];
```

```
        /*  用 popen 建立一个通向"ls:的读管道  */
        if (( pipein_fp = popen("ls", "r")) == NULL)
          {
                perror("popen");
                 exit(1);
          }

        /*  用 popen 建立一个通向"sort"的写管道  */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
          {
                perror("popen");
                exit(1);
           }

        /*  进程循环  */
         while(fgets(readbuf, 80, pipein_fp))
                fputs(readbuf,  pipeout_fp);

        /*  关闭打开的管道  */
         pclose(pipein_fp);
         pclose(pipeout_fp);

         return(0);
    }
```

最后，为了更好的理解管道，我们给出一个 popen()**和** fopen()**混合使用的例子，请读者与上例对照，自行分析管道与文件处理的异同：**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE  *pipe_fp,  *infile;
        char  readbuf[80];

       if( argc != 3) {
                fprintf(stderr,  "USAGE:   popen3  [command]  [filename]\n");
                 exit(1);
         }

        /*  打开输入文件  */
       if (( infile = fopen(argv[2], "rt")) == NULL)
         {
                perror("fopen");
                exit(1);
         }
```

```
        /*  建立写管道 */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
          {
                perror("popen");
                exit(1);
          }

        /* Processing loop */
          do
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;

                fputs(readbuf, pipe_fp);
          } while(!feof(infile));

          fclose(infile);
          pclose(pipe_fp);

          return(0);
}
```

4.3.2 需要注意的问题

以下是一些在管道的使用中需要注意的问题：

1．pipe()**的调用必须在** fork()**之前；**

2．**及时关闭不需要的管道句柄；**

3．**使用** dup()**之前确定定向的目标是最小的文件句柄；**

4．**管道只能实现父子进程间的通信，如果两个进程之间没有** fork()**关系，就必须考虑其他的进程通信方法。**

4.4 有名管道

为了解决管道不能提供非父/子关系进程间通信的缺陷，在管道的基础上发展了有名管道（FIFOs**）的概念。我们知道，尽管管道在** Linux **系统内部是以文件节点（**inode**）的形式存在的，但是由于其对外的不可见性（“无名”性），我们无法创建新的句柄对其进行访问。而有名管道在** Linux **系统中以一种特殊的设备文件的形式存在于文件系统中。这样它不仅具有了管道的通信功能，也具有了普通文件的优点（可以同时被多个进程共享，可以长期存在等等），有效的解决了管道通信的缺点。**

4.4.1 有名管道的创建

因为有名管道是存在于文件系统中的文件节点，所以我们可以用建立文件节点的方式

来建立有名管道。在 shell 中我们可以用下面的命令：

```
#mknod sampleFIFO p
#mkfifo – m 0666 sampleFIFO
```

以上的两个命令是等价的，它们都会在当前的文件系统中建立一个名字为 samlpeFIFO 的有名管道。不过，在细节上他们还是有差别的。mkfifo 命令可以用“ -m ”选项指定所建立的有名管道的存取权限，而 mknod 则需要之后使用 chmod 来改变有名管道的存取权限。

通过文件列表信息中的 p 指示符我们可以迅速的辨认出有名管道。例如：

```
#ls -l
prw-r--r--    1 root       root                0 May 14 16:25 sampleFIFO|
```

在 C 中我们通过系统函数 mknod 来建立有名管道：

```
        库函数：  mknod();
      函数声明：  int mknod( char *pathname, mode_t mode, dev_t dev);
        返回值：  0 on success,
                  -1 on error: errno = EFAULT (pathname invalid)
                                       EACCES (permission denied)
                                       ENAMETOOLONG (pathname too long)
                                       ENOENT (invalid pathname)
                                       ENOTDIR (invalid pathname)
                                      (see man page for mknod for others)
          NOTES:  Creates a filesystem node (file, device file, or FIFO)
```

下面是个简单的例子：

```
mknod(“/tmp/sampleFIFO”,s_IFIFO|0666,0)
```

这条语句在文件系统中建立了一个名为”/tmp/sampleFIFO”的有名管道，其读写权限是 0666（当然，最终的权限还和你的 umask 值有关）。mknod 的第三个参数在创建有名管道时被忽略，一般都填零。

4.4.2　有名管道的 I/O 使用

有名管道和管道的操作是相同的，只是要注意，在引用已经存在的有名管道时，首先要用系统中的文件函数来打开它，才能接下来进行其他的操作。例如，我们可以用操作文件流的 fopen()和 fclose()来打开一个有名管道。下面是一个 server 方的例子：

```
/* fifoserver.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>

#define FIFO_FILE   "sampleFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

      /* Create the FIFO if it does not exist */
```

```
        umask(0);

        /*在文件系统中创建有名管道*/
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                /*打开有名管道*/
                fp = fopen(FIFO_FILE, "r");

                /*从有名管道中读取数据*/
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);

                /*关闭有名管道*/
                fclose(fp);
        }
        return(0);
}
```

因为有名管道自动支持进程阻塞，所以我们可以让这个 server **在后台运行：**

```
#fifoserver &
```

然后运行下面的 client **程序：**

```
#include <stdio.h>
#include <stdlib.h>

#define  FIFO_FILE          "sampleFIFO"

int main(int argc, char *argv[])
{
        FILE  *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }
          /*打开有名管道*/
        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }
          /*向有名管道中写入数据*/
        fputs(argv[1], fp);
          /*关闭有名管道*/
```

```
        fclose(fp);
        return(0);
}
```

由于有名管道的自动阻塞特性，当上面的 server 打开一个有名管道准备读入时，server 进程就会被阻塞以等待其他进程（在这里是我们的 client 进程）在有名管道中写入数据。反之亦然。不过，如果需要，我们也可以在打开一个有名管道时使用 O_NONBLOCK 标志来关闭它的自动阻塞特性。

4.4.3　未提到的关于有名管道的一些注意

首先，有名管道必须同时有读/写两个进程端。如果一个进程试图向一个没有读入端进程的有名管道写入数据，一个 SIGPIPE 信号就会产生。这在涉及多个进程的有名管道通信中是很有用的。

其次，关于管道操作的独立性。一个“独立”的操作意味着，这个操作不会因为任何原因而被中断。比如，在 POSIX 标准中，头文件/usr/include/posix1_lim.h 中定义了在一次独立的管道读/写操作中最大传输的数据量(buffer size)：

```
#define  _POSIX_PIPE_BUF          512
```

也即是说，在一次独立的管道读/写操作中最多只能传送 512 个字节的数据，当数据量超过这个上限时操作就只能被分成多次独立的读/写操作。在 Linux 系统中，头文件“linux/limits.h”中定义了类似的限制：

```
#define  PIPE_BUF          4096
```

可以看出，和 POSIX 标准比，上限被大大增加了。这在涉及多进程的有名管道操作中是非常重要的。如果在某个进程的一次写操作中传输的数据量超过了独立读/写操作的数据量上限，这个操作就有可能被别的进程的写操作打断。也就是说，别的进程把数据插入了该进程写入管道的数据序列中从而造成混乱。这是在有名管道应用中需要特别注意的。

4.5　文件和记录锁定

共享资源的保护问题是多进程操作系统中一个非常重要的问题。在这一节中，我们将会讲述一些保护文件这种使用频率最高的共享资源的方法。在正式开始之前，让我们先来看一个例子。

4.5.1　实例程序及其说明

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define SEQFILE "./sequo"
#define MAXBUF 100
```

```
main()
{
        int fd,i,n,pid,seqno;
        char buff[MAXBUF+1];

        pid=getpid();
        if ((fd=open("data",O_RDWR|O_CREAT) )<0)
        {
                perror("Can't open");
                exit(1);
        }
        for(i=0;i<5;i++)
        {
                my_lock(fd);
                lseek(fd,01,0);

                if ((n=read(fd,buff,MAXBUF))<=0)
                {
                        perror("read error");
                        exit(1);
                }
                buff[n]=0;
                if (!(n=sscanf(buff,"%d\n",&seqno)))
                {
                        perror("sscanf error");
                        exit(1);
                }
                printf("pid=%d,seq#=%d\n",pid,seqno);
                seqno++;
                sprintf(buff,"%03d\n",seqno);
                n=strlen(buff);
                lseek(fd,01,0);
                if( write(fd,buff,n)!=n)
                {
                        perror("write error");
                        exit(1);
                }
                my_unlock();
        }
        close(fd);
}

my_lock(int fd)
{
```

```
        return;
}

my_unlock(int fd)
{
        return;
}
```

在上面实例程序中文件锁定和文件解锁并不执行任何上锁和解锁操作，因此该程序的工作将简化为打开序号文件，把序号文件中的序号加 1 后写回源文件，以及关闭序号文件等。

如果我们把序号文件"seqno"的内容初始化为 1，并且运行一次该程序，我们将得到如下结果：

```
pid=5657,seq#=1
pid=5657,seq#=2
pid=5657,seq#=3
pid=5657,seq#=4
pid=5657,seq#=5
```

如果把序号文件再一次初始化为 1 并且用如下命令运行两次该程序：a.out&a.out &，我们会见到这样的结果：

```
pid=5708,seq#=1
pid=5709,seq#=1
pid=5709,seq#=2
pid=5709,seq#=3
pid=5709,seq#=4
pid=5709,seq#=5
pid=5708,seq#=2
pid=5708,seq#=3
pid=5708,seq#=4
pid=5708,seq#=5
```

分析上面第二个运行结果，我们不难看出：序号文件“ seqno ”是该程序所产生的两个进程同时要求访问的共享资源。第一个进程读入序号文件输出 1，并准备增值后写回原文件时，第二个进程开始被系统操作调度，它读入“ seqno ”文件，然后把序号一直修改到 5 后退出，直到此时，第一个进程才重新获得 CPU，它把增值后的 2 写回原文件，继续以下操作，依次输出（这里需要说明的是，为了节省篇幅，我们在程序中只作了 5 次循环。在此程序实际运行中，由于 5 次循环占用时间太短，所以一般不会发生文中叙述的两个进程的变换。读者在试验此程序时，可以将循环次数加大，如 500 次，就可以观察到文中所叙述的现象了）。

第二个结果并不是我们所期望的输出，究其原因，主要是由于在共享资源的访问中没有健全互斥性措施，从而引起了访问和操作的混乱。因此我们要求进程在对共享资源访问前必须进行锁定以避免其它进程对它的操作，当然在该进程访问完共享资源后也要进行解锁操作，以使其它进程能有机会占用共享资源。文件和记录锁定就是 Linux 操作系统为共享资源提供的互斥性保障。

4.5.2 锁定中的几个概念

文件锁定的是整个文件，而记录锁定只锁定文件的某一特定部分。UNIX 的记录指的是从文件的某一相对位置开始的一段连续的字节流，它不同于其它以强制性记录结构阻止文件的操作系统，因此，UNIX 记录锁更恰当的称呼应该是范围锁，它是对文件某个范围的锁定。

文件和记录锁定可分为咨询式锁定和强制锁定两种。当正在运行的某一进程对它将要访问的某一文件进行了咨询式锁定后，其它想要访问该文件的进程将被操作系统告知共享文件已经上了锁，但这并不阻止它们对锁定文件的操作。只要有对锁定文件的存取权，这些进程便可忽视咨询式锁定而去写赏了锁的文件。强制锁定的含义则要严格多了，当某一共享文件被强制后，操作系统将会对每个读写文件的请求进行核查，只有在确证该请求不会干扰上了锁的文件时，才允许对应的操作。System V 和 BSD 都提供了咨询式锁定方式。这两种锁定方式都被 Linux 支持。

4.5.3 System V 的咨询锁定

System V 的锁函数 lockf()具有如下的形式：

```
#include <unistd.h>

int lockf(int fd, int function, long size);
```

参数 fd 是在文件打开操作中获得的文件描述符；

参数 function 可以取如下的参数值：

F_ULOCK	为一个先前锁定的区域解锁
F_LOCK	锁定一个区域
F_TLOCK	测试并锁定一个区域
F_TEST	测试一个区域是否已经上锁。

参数 size 指明了从文件当前位置开始的一段连续锁定区域的长度，当 size 为 0 时，锁定记录将由当前位置一直扩展到文件尾。

函数 lockf()既可以用来上锁有可以用来测试是否已经赏了锁。如果 lockf 的参数 function 为 F_LOCK 指定文件的对应区域已被其它进程锁定，那么 lockf 的调用进程将被阻塞直到该区域解锁。上述情况我们称为阻塞。如果在调用 lockf()时把参数设为 F_TLOVK，那么当被测试的区域上了锁时，lockf 便会立即返回-1，出错返回码 errno 将为 EAGAIN,它是一个非阻塞调用。

```
if(!lockf(fd,F_TEST,size))
{
    rc==lockf(fd,F_LOCK,size);
        …
        …
}
```

上面这段代码看上去好像是非阻塞调用，但是如果当运行此代码段的进程在测试到对应文件没有被锁定时，又有另一个进程被操作系统调度占有 CPD，它将同样测试出文件未被锁定，然后对共享文件上锁。当后继进程在对锁文件操作时，再一次被操作系统调度的第一个进程，其锁定文件的操作将仍然是一个阻塞性调用。因此为了实现非阻塞调用，我

们必须使用 F_TLOCK 参数的 lockf()调用。

有个锁函数 lockf()之后，我们便可以完善前面的上锁 my_lock()和解锁 my_unlock()函数，防止共享文件访问中的混乱情况。下面的上锁函数采用的是阻塞调用。

```
#include <unistd.h>

my_lock(int fd)
{
    /* 将文件指针移回文件头 */
    lseek(fd,0L,0);
    /* 锁定整个文件 */
    if (lockf(fd,F_LOCK,0L)==-1)
    {
        perror("can't F_LOCK");
        exit(1);
    }
}

my_unlock(int fd)
{
    lseek(fd,0L,0);
    if(lockf(fd,F_ULOCK,0L)==-1)
    {
        perror("can't F_UNLOCK");
        exit(1);
    }
}
```

4.5.4　BSD 的咨询式锁定

4.3 BSD UNIX 操作系统提供了如下形式的调用来锁定和解锁一个文件：

```
#include <sys/file.h>

int flock(int fd, int operation);
```

调用 flock 有两个参数：

参数 fd 是一个已打开文件的文件描述符；

参数 operation 可设定为下述各值：

LOCK_SH　共享锁

LOCK_EX　互斥锁

LOCK_UN　解锁

LOCK_NB　当文件已被锁定时不阻塞

4.3BSD UNIX 使用 flock()来请求对指定文件的咨询式锁定和解锁。4.3BSD 的咨询锁有共享锁和互斥锁两种。在任一给定时刻，多个进程可以用于属于同一文件的共享锁，但是某共享文件不能同时具有多个互斥锁或存在共享锁和互斥锁共存的情况。如果锁定成功，flock 将返回零，否则返回-1。

flock()允许的锁操作有下列几种：

LOCK_SH	阻塞性共享锁
LOCK_EX	阻塞性互斥锁
LOCK_SH | LOCK_NB	非阻塞性共享锁
LOCK_EX | LOCK_NB	非阻塞性互斥锁
LOCK_UN	解锁

当 flock()采用非阻塞锁定操作时，对已锁定文件的锁定将使该调用失败返回，其出错码为 EWOULDBLOCK。

下面，我们将使用 BSD 的文件机制，构造 4.5.1 节中缺省的上锁和解锁函数。其中上锁函数采取的是互斥锁。

```
#include <sys/file.h>

my_flock(int fd)
{

    if (flock(fd,LOCK_EX))==-1)
    {
        perror("can LOCK_EX");
        exit(1);
    }
}

my_unload(fd)
{
    if (flock(fd,LOCK_UN)==-1)
    {
        perror("can't LOCK_UN");
        exit(1);
    }
}
```

有了这个上锁函数和 my_lock()和解锁函数 my_unlock()之后，让我们再回过头来看看前面的 a.out&命令的运行结果：

```
pid=5894,seq#=1
pid=5894,seq#=2
pid=5894,seq#=3
pid=5894,seq#=4
pid=5894,seq#=5
pid=5895,seq#=6
pid=5895,seq#=7
pid=5895,seq#=8
pid=5895,seq#=9
pid=5895,seq#=10
```

该结果完全体现了对共享序号文件“ seqno”的理想共享操作。

4.5.5　前面两种锁定方式的比较

由于 Linux 支持上面的两种锁定方式，所以可以根据不同的实际情况选用不同的锁定方式。以上的两种锁定方式有以下的不同：

1．System V的锁定方式是记录锁定，可以指定锁定的范围。而 BSD 的锁定方式是文件锁定，只能指定锁定文件。

2．System V 的锁定是每个进程所独有的，可以用于父子进程间的共享锁定。而 BSD 的锁定方式是可以继承的，父子进程间使用的是同一锁定的，所以不能用于父子进程间的文件共享锁定。

4.5.6　Linux 的其它上锁技术

创建和使用一个辅助文件以表示进程对共享文件的锁操作是 Linux 其它上锁技术的基本点。如果辅助文件存在，资源便被其它进程锁定了，否则，进程就可以创建辅助文件以对资源上锁。

创建辅助文件的一个比较直观的想法是先测试一下辅助文件是否已经存在，若不存在便创建之：

```
if((fd=open(file,0))<0)
    fd=creat(file,0644);

.....
```

然而由于上述方法采用了两个独立的系统调用。在这两个调用之间，其它进程也有可能被调度占用 CPU，测试和创建相同的文件。当先前被打断的进程再依次运行后，它所发出的 creat 调用即使文件已经存在也不会出错。

为了正确使用辅助文件，实现文件的锁操作，我们采用的第一个技巧用到这样一个事实：如果文件的新链接名已经存在，系统调用 link()便会出错，在全程变量 errno 中返回 EEXIST。link()调用的形式如下：

```
#include <unistd.h>

int link(char* existingpath, char* newpath);
```

两个参数分别是原文件所在的路径和新建链接的路径。

我们的技巧是创建一独特临时文件，它的名字由进程号得来。一旦创建了该文件，我们便用辅助锁文件名形成到该临时文件的另一个链接。如果链接成功，进程便把文件锁定了。这时有两个路径指向锁定文件（基于进程号的临时文件和锁文件）。然后我们用 unlink()系统调用，把临时文件删除，只剩下一个指向该文件的链接。当需要解除锁定时，我们就用 unlink()删除解除对该文件的链接。

```
#include <sys/errno.h>

#define LOCKFILE "seqno.lock"

extern int errno;
```

```
my_lock(int fd)
{
    int temfd;
    char tempfile[30];

    sprintf(tempfile,"LCK%d",getpid());
    /* 建立一个临时文件，然后关闭之 */
    if ((temfd=creat(tempfile,0444))<0)
    {
        perror("can't creat temp file");
        exit(1);
    }
    close(temfd);

    /* 现在试图用辅助锁文件名建立对 */
    /* 临时文件的链接。这可以测试锁 */
    /* 锁文件是否已经存在 */
    while (link(tempfile,LOCKFILE)<0) {
        if（errno!=EEXIST)
        {
            perror("link error");
            exit(1);
        }
        sleep(1);
    }
    if(unlink(tempfile)<0)
    {
        perror("unlink error");
        exit(1);
    }
}

my_unlock(int fd)
{
    if (unlink(LOCKFILE)<0)
    {
        perror("unlink error");
        exit(1);
    }
}
```

第二个技巧要用到这样一个事实，如果文件已经存在且不允许写，那么对该文件调用 creat()**便会错误返回，**errno **置为** EACCES**。我们解决的办法是创建一所有写许可都被禁止的临时锁文件。如果调用** creat()**成功，调用进程便知道它给该文件上锁了，它便可以安全修改序号文件。解锁操作便是把临时锁文件移开。调用者没有必要对未链接的文件有读或**

写的权限。该技术有个疏漏之处，就是如果进程有超级用户特权，就不会被错误返回。这是因为超级用户可以写任何文件。

```
#include <sys/errno.h>

#define LOCKFILE "seqno.lock"
#define TEMPLOCK "temp.lock"

my_lock(int fd)
{
    int tempfd;

    /* 试图创建一个全部写权限都被关闭的 */
    /* 临时文件。如果该文件已经存在，则 */
    /*   creat()调用失败    */
    while ((tempfd=creat(TEMPLOCK,0))<0)
    {
        if(errno!=EACCES)
        {
            perror("creat error");
            exit(1);
        }
        sleep(1);
    }
    close(temfd);
}

my_unlock(int fd)
{
    if(unlink(TEMPLOCK)<0)
    {
        perror("unlink error");
        exit(1);
    }
}
```

第三个即使是使用 Linux **所提供的** open()**调用的可选项。头文件** fcntl.h **中定义了系统调用** open()**的可选项。如果选项中既设置了** O_CREAT **又设置了** O_EXCL **的话，如果文件已经存在，**open()**调用便会失败，否则话便创建该文件。**

```
#include <fcntl.h>
#include <sys/errno.h>

#define LOCKFILE "sqgno.lock"
#define PERMS 0666

extern int errno;
```

```
my_lock(int fd)
{
    int tempfd;

    while((tempfd=open(LOCKFILE,O_RDWR|O_CREAT|O_EXCL,PERMS))<0)
    {
        if (errno!=EXIST)
        {
            perror("open error");
            exit(1);
        }
        sleep(1);
    }
    close(tempfd);
}

my_unlock(fd)
{
    if(unlink(LOCKFILE)<0)
    {
        perror("unlink error");
        exit(1);
    }
}
```

对于以上三个技术要注意如下几点：

1．这些技术比实际的文件锁定系统调用要花更长的时间。这是因为要用到多个系统调用以及多个文件系统操作。

2．除了包含共享资源的文件外，还要用到一个辅助锁文件。在第一个例子中，我们既需要序号文件又需要辅助锁文件。

3．系统崩溃后辅助文件会存在，需要设计相应的方法处理之。

4．系统调用 link **不能为不同逻辑文件系统上的文件建立链接，因此临时文件一般不放在**/tmp **文件系统中。**

5．如果进程以超级用户特权竞争资源的话，第二种方法就无效。

6．当辅助锁为另一进程占有时，需要锁的进程便不知何时能再检测一下，在我们的例子中是等待 1 **秒钟。理想的情况是锁释放时应通知需要锁的进程。**

7．占有锁的进程可以不解锁便终止。

4.6 System V IPC

AT&T **在** UNIX System V **中引入了几种新的进程通讯方式，即消息队列（**Message Queues**），信号量（**semaphores**）和共享内存（**shared memory**），统称为** System V IPC**。在** Linux **系统编程中，它们有着广泛的应用。**

System V IPC **的一个显著的特点，是它的具体实例在内核中是以对象的形式出现的，**

我们称之为 IPC 对象。每个 IPC 对象在系统内核中都有一个唯一的标识符。通过标识符内核可以正确的引用指定的 IPC 对象.。需要注意的是，标识符的唯一性只在每一类的 IPC 对象内成立。比如说，一个消息队列和一个信号量的标识符可能是相同的，但绝对不会出现两个有相同标识符的消息队列。

标识符只在内核中使用，IPC 对象在程序中是通过关键字（key）来访问的。和 IPC 对象标识符一样，关键字也必须是唯一的。而且，要访问同一个 IPC 对象，Server 和 Client 必须使用同一个关键字。因此，如何构造新的关键字使之不和已有的关键字冲突，并保证 Server 和 Client 使用的关键字是相同的，是建立 IPC 对象时首先要解决的一个问题。

通常，我们使用系统函数 ftok()来生成关键字。

库函数： ftok();
函数声明： key_t ftok (char *pathname, char proj);
返回值： new IPC key value if successful
-1 if unsuccessful
errno set to return of stat() call

ftok()函数通过混合 pathname 所指文件的 inode 和 minor device 值以及 proj 的值来产生关键字。这样并不能完全保证关键字的唯一性，不过程序可以检测关键字的冲突并通过更换 pathname 和 proj 的组合来产生新的关键字。下面是一段使用 ftok 的代码：

```
key_t     mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在这段代码中，ftok 函数混合文件/tmp/myapp 和字符 a 来产生关键字 mykey。下面是更常用的一段代码：

```
key_t     mykey;
mykey = ftok(".", 'a');
```

只要我们保证 server 和 client 从同一个目录运行，我们就可以保证它们使用上面的代码产生的关键字是相同的。

获得了关键字以后，就可以通过它来建立或引用具体 IPC 对象了。

在我们进入到具体的 IPC 对象前，先看看几个和 System V IPC 有关的命令行指令：

4.6.1 ipcs 命令

ipcs 命令在终端显示系统内核的 IPC 对象状况。
ipcs –q 只显示消息队列
ipcs –m 只显示共享内存
ipcs –s 只显示信号量
下面是在某个 Linux 系统上使用 ipcs 命令的显示情况：

```
[roy@bbs ~]$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000a0a 644        roy        600        1024       2
0x00000af8 775        roy        600        1024       2

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
```

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

[roy@bbs ~]$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

[roy@bbs ~]$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

[roy@bbs ~]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000a0a 644        roy        600        1024       2
0x00000af8 775        roy        600        1024       2
```

4.6.2 ipcrm 命令

使用 ipcrm 命令强制系统删除已存在的 IPC 对象。

它的命令格式如下：

ipcrm <msg | sem | shm> <IPC ID>

ipcrm 后面的参数指定要删除的 IPC 对象类型，分别为消息队列（msg）、信号量（sem）和共享内存（shm）。然后需要给出要删除对象的标识符。标识符可以通过 ipcs 命令来取得。

通过正确的使用以上的两个命令可以帮助我们有效的解决 IPC 对象使用中的问题。

在下面的章节中我们将开始讲述具体的每一种 IPC 对象。

4.7 消息队列（Message Queues）

顾名思义，消息队列就是在系统内核中保存的一个用来保存消息的队列。但这个队列并不是简单的进行“先入先出”的操作，我们可以控制消息用更为灵活的方式流动。

4.7.1 有关的数据结构

在介绍消息队列的使用前，让我们先熟悉一下在后面会碰到的几个和消息队列有关的数据结构：

1．ipc_perm

系统使用 ipc_perm 结构来保存每个 IPC 对象权限信息。在 Linux 的库文件 linux/ipc.h 中，它是这样定义的：

struct ipc_perm

```
{
   key_t    key;
   ushort uid;    /* owner euid and egid */
   ushort gid;
   ushort cuid;  /* creator euid and egid */
   ushort cgid;
   ushort mode;  /* access modes see mode flags below */
   ushort seq;    /* slot usage sequence number */
};
```

结构里的前几个成员的含义是明显的，分别是 IPC 对象的关键字，uid 和 gid。然后是 IPC 对象的创建者的 uid 和 gid。接下来的是 IPC 对象的存取权限。最后一个成员也许有点难于理解，不过不要担心，这是系统保存的 IPC 对象的使用频率信息，我们完全可以不去理会它。

2．msgbuf

消息队列最大的灵活性在于，我们可以自己定义传递给队列的消息的数据类型的。不过这个类型并不是随便定义的，msgbuf 结构给了我们一个这类数据类型的基本结构定义。在 Linux 的系统库 linux/msg.h 中，它是这样定义的：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
        long mtype;          /* type of message */
        char mtext[1];       /* message text */
};
```

它有两个成员：

mtype 是一个正的长整型量，通过它来区分不同的消息数据类型。

mtext 是消息数据的内容。

通过设定 mtype 值，我们可以进行单个消息队列的多向通讯。如下图，client 可以给它向 server 发送的信息赋于一个特定的 mtype 值，而 server 向 client 的信息则用另一个 mtype 值来标志。这样，通过 mtype 值就可以区分这两向不同的数据。

利用相同的原理，可以实现更复杂的例子。

需要注意的是，虽然消息的内容 mtext 在 msgbuf 中只是一个字符数组，但事实上，在我们定义的结构中，和它对应的部分可以是任意的数据类型，甚至是多个数据类型的集合。比如我们可以定义这样的一个消息类型：

```
struct my_msgbuf {
    long      mtype;            /* Message type */
    long      request_id;       /* Request identifier */
    struct   client info;     /* Client information structure */
};
```

在这里，与 mtext 对应的是两个数据类型，其中一个还是 struct 类型。由此可见消息队列在传送消息上的灵活性。

不过，虽然没有类型上的限制，但 Linux 系统还是对消息类型的最大长度做出了限制。在 Linux 的库文件 linux/msg.h 中定义了每个 msgbuf 结构的最大长度：

```
#define MSGMAX   4056    /* <= 4056 */    /* max size of message (bytes) */
```

也即是说，包括 mtype 所占用的 4 个字节，每个 msgbuf 结构最多只能只能占用 4056 字节的空间

3．msg

消息队列在系统内核中是以消息链表的形式出现的。而完成消息链表每个节点结构定义的就是 msg **结构。它在** Linux **的系统库** linux/msg.h **中的定义是这样的：**

```
/* one msg structure for each message */
struct msg {
        struct msg *msg_next;    /* next message on queue */
        long    msg_type;
        char *msg_spot;          /* message text address */
        time_t msg_stime;        /* msgsnd time */
        short msg_ts;            /* message text size */
};
```

msg_next **成员是指向消息链表中下一个节点的指针，依靠它对整个消息链表进行访问。**

msg_type **和** msgbuf **中** mtype **成员的意义是一样的。**

msg_spot **成员指针指出了消息内容（就是** msgbuf **结构中的** mtext**）在内存中的位置。**

msg_ts **成员指出了消息内容的长度。**

4．msgqid_ds

msgqid_ds **结构被系统内核用来保存消息队列对象有关数据。内核中存在的每个消息队列对象系统都保存一个** msgqid_ds **结构的数据存放该对象的各种信息。在** Linux **的库文件** linux/msg.h **中，它的定义是这样的：**

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* first message on queue */
        struct msg *msg_last;           /* last message in queue */
        __kernel_time_t msg_stime;      /* last msgsnd time */
        __kernel_time_t msg_rtime;      /* last msgrcv time */
        __kernel_time_t msg_ctime;      /* last change time */
        struct wait_queue *wwait;
        struct wait_queue *rwait;
        unsigned short msg_cbytes;      /* current number of bytes on queue */
        unsigned short msg_qnum;        /* number of messages in queue */
        unsigned short msg_qbytes;      /* max number of bytes on queue */
        __kernel_ipc_pid_t msg_lspid;   /* pid of last msgsnd */
        __kernel_ipc_pid_t msg_lrpid;   /* last receive pid */
};
```

其中，

msg_perm **成员保存了消息队列的存取权限以及其他一些信息（见上面关于** ipc_perm **结构的介绍）。**

msg_first **成员指针保存了消息队列（链表）中第一个成员的地址。**

msg_last **成员指针保存了消息队列中最后一个成员的地址。**

msg_stime **成员保存了最近一次队列接受消息的时间。**

msg_rtime **成员保存了最近一次从队列中取出消息的时间。**

msg_ctime **成员保存了最近一次队列发生改动的时间（见后面的章节）。**

wwait 和 rwait 是指向系统内部等待队列的指针。

msg_cbytes 成员保存着队列总共占用内存的字节数。

msg_qnum 成员保存着队列里保存的消息数目。

msg_qbytes 成员保存着队列所占用内存的最大字节数。

msg_lspid 成员保存着最近一次向队列发送消息的进程的 pid。

msg_lrpid 成员保存着最近一次从队列中取出消息的进程的 pid。

4.7.2 有关的函数

介绍完了有关的结构之后，我们来看看处理消息队列所用到的函数：

1．msgget()

msgget()函数被用来创建新的消息队列或获取已有的消息队列。其函数定义如下：

系统调用： msgget()

函数声明： int msgget (key_t key, int msgflg)

返回值： message queue identifier on success
-1 on error: errno = EACCESS (permission denied)
EEXIST (Queue exists, cannot create)
EIDRM (Queue is marked for deletion)
ENOENT (Queue does not exist)
ENOMEM (Not enough memory to create queue)
ENOSPC (Maximum queue limit exceeded)

msgget()函数的第一个参数是消息队列对象的关键字(key)，函数将它与已有的消息队列对象的关键字进行比较来判断消息队列对象是否已经创建。而函数进行的具体操作是由第二个参数，msgflg 控制的。它可以取下面的几个值：

IPC_CREAT ：

如果消息队列对象不存在，则创建之，否则则进行打开操作;

IPC_EXCL：

和 IPC_CREAT 一起使用（用"|"连接），如果消息对象不存在则创建之，否则产生一个错误并返回。

如果单独使用 IPC_CREAT 标志，msgget()函数要么返回一个已经存在的消息队列对象的标识符，要么返回一个新建立的消息队列对象的标识符。如果将 IPC_CREAT 和 IPC_EXCL 标志一起使用，msgget()将返回一个新建的消息对象的标识符，或者返回-1 如果消息队列对象已存在。IPC_EXCL 标志本身并没有太大的意义，但和 IPC_CREAT 标志一起使用可以用来保证所得的消息队列对象是新创建的而不是打开的已有的对象。

除了以上的两个标志以外，在 msgflg 标志中还可以有存取权限控制符。这种控制符的意义和文件系统中的权限控制符是类似的。

最后，我们将使用 msgget()函数建立一个更加简便的封装函数来作为本节的例子：

```
int open_queue( key_t keyval )
{
        int         qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }
```

```
        return(qid);
}
```

这个简单的例子中唯一需要注意的一点就是在 msgflg **中加入了存取权限控制符** 0660。**其余的部分请读者自行分析。**

2．msgsnd()

从函数名就可以看出，msgsnd()**函数是用来向消息队列发送消息的。在** linux/msg.h **它的函数定义是这样的：**

系统调用： msgsnd()

函数声明： int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg)

返回值： 0 on success

-1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was asserted)
EACCES (permission denied, no write permission)
EFAULT (msgp address isn't accessable – invalid)
EIDRM (The message queue has been removed)
EINTR (Received a signal while waiting to write)
EINVAL (Invalid message queue identifier, nonpositive message type, or invalid message size)
ENOMEM (Not enough memory to copy message buffer)

传给 msgsnd()**函数的第一个参数** msqid **是消息队列对象的标识符（由** msgget()**函数得到），第二个参数** msgp **指向要发送的消息所在的内存，第三个参数** msgsz **是要发送信息的长度（字节数），可以用以下的公式计算：**

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个参数是控制函数行为的标志，可以取以下的值：

0**，忽略标志位；**

IPC_NOWAIT**，如果消息队列已满，消息将不被写入队列，控制权返回调用函数的线程。如果不指定这个参数，线程将被阻塞直到消息被可以被写入。**

这里我们将创建一个封装函数来演示 msgsnd()**函数的使用：**

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;

      /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }
        return(result);
}
```

利用这节和上节中我们创建的两个函数，我们已经可以写出一个很简单的消息发送程序：

```
main()
{
        int     qid;
        key_t   msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
```

```
        int     request;        /* Work request number */
        double  salary;         /* Employee's salary */
    } msg;

    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');

    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
        perror("open_queue");
        exit(1);
    }

    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */

    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
        perror("send_message");
        exit(1);
    }
}
```

在程序中我们首先使用 ftok **函数产生关键字，再调用我们的封装函数** open_queue()**得到消息队列的标识符。最后再用** send_message()**函数将消息发送到消息队列中。**

3．msgrcv()

和 msgsnd()**函数对应，**msgrcv()**函数被用来从消息队列中取出消息。它在** linux/msg.h **中的定义是这样的：**

系统调用： msgrcv()

函数声明： int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype,
int msgflg)

返回值： Number of bytes copied into message buffer
-1 on error: errno = E2BIG (Message length is greater than msgsz,
no MSG_NOERROR)
EACCES (No read permission)
EFAULT (Address pointed to by msgp is invalid)
EIDRM (Queue was removed during retrieval)
EINTR (Interrupted by arriving signal)
EINVAL (msgqid invalid, or msgsz less than 0)
ENOMSG (IPC_NOWAIT asserted, and no message
exists in the queue to satisfy the request)

函数的前三个参数和 msgsnd()**函数中对应的参数的含义是相同的。第四个参数** mtype

指定了函数从队列中所取的消息的类型。函数将从队列中搜索类型与之匹配的消息并将之返回。不过这里有一个例外。如果 mtype 的值是零的话，函数将不做类型检查而自动返回队列中的最旧的消息。

第五个参数依然是是控制函数行为的标志，取值可以是：

0,表示忽略；

IPC_NOWAIT，如果消息队列为空，则返回一个 ENOMSG，并将控制权交回调用函数的进程。如果不指定这个参数，那么进程将被阻塞直到函数可以从队列中得到符合条件的消息为止。如果一个 client 正在等待消息的时候队列被删除，EIDRM 就会被返回。如果进程在阻塞等待过程中收到了系统的中断信号，EINTR 就会被返回。

MSG_NOERROR，如果函数取得的消息长度大于 msgsz，将只返回 msgsz 长度的信息，剩下的部分被丢弃了。如果不指定这个参数，E2BIG 将被返回，而消息则留在队列中不被取出。

当消息从队列内取出后，相应的消息就从队列中删除了。

和上节一样，我们将开发一个 msgrcv()的封装函数 read_message()：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

      /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

利用上面提到的 msgrcv()对消息长度的处理，我们可以使用下面的方法来检查队列内是存在符合条件的信息：

```
int peek_message( int qid, long type )
{
        int     result, length;

        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

这里我们将 msgp 和 msgsz 分别设为 NULL 和零。然后检查函数的返回值，如果是 E2BIG 则说明存在符合指定类型的消息。一个要注意的地方是 IPC_NOWAIT 的使用，它防止了阻

塞的发生。

4．msgctl()

通过 msgctl()函数，我们可以直接控制消息队列的行为。它在系统库 linux/msg.h 中的定义是这样的：

系统调用：　　msgctl()

函数声明：　　int msgctl (int msgqid, int cmd, struct msqid_ds *buf)

返回值：　　0 on success

-1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
EFAULT (Address pointed to by buf is invalid with IPC_SET and IPC_STAT commands)
EIDRM (Queue was removed during retrieval)
EINVAL (msgqid invalid, or msgsz less than 0)
EPERM (IPC_SET or IPC_RMID command was issued, but calling process does not have write (alter) access to the queue)

函数的第一个参数 msgqid 是消息队列对象的标识符。

第二个参数是函数要对消息队列进行的操作，它可以是：

IPC_STAT

取出系统保存的消息队列的 msqid_ds 数据，并将其存入参数 buf 指向的 msqid_ds 结构中。

IPC_SET

设定消息队列的 msqid_ds 数据中的 msg_perm 成员。设定的值由 buf 指向的 msqid_ds 结构给出。

IPC_EMID

将队列从系统内核中删除。

这三个命令的功能都是明显的，所以就不多解释了。唯一需要强调的是在 IPC_STAT 命令中队列的 msqid_ds 数据中唯一能被设定的只有 msg_perm 成员，其是 ipc_perm 类型的数据。而 ipc_perm 中能被修改的只有 mode,pid 和 uid 成员。其他的都是只能由系统来设定的。

最后我们将使用 msgctl()函数来开发几个封装函数作为本节的例子：

IPC_STAT 的例子：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }
        return(0);
}
```

IPC_SET 的例子：

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

       /* Retrieve a current copy of the internal data structure */
        get_queue_ds( qid, &tmpbuf);
```

```
        /* Change the permissions using an old trick */
          sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

        /* Update the internal data structure */
          if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
          {
                  return(-1);
          }
          return(0);
}
```

IPC_RMID 的例子：

```
int remove_queue( int qid )
{
          if( msgctl( qid, IPC_RMID, 0) == -1)
          {
                  return(-1);
          }

          return(0);
}
```

4.7.3 消息队列实例——msgtool，一个交互式的消息队列使用工具

没有人能够拒绝现成的准确技术信息所提供的迅捷与方便。这些材料为我们学习和探索新的领域提供了了非常好的机制。同样，将技术信息应用于现实的领域中也能大大的加快我们学习的过程。

直到目前，我们所接触的有关消息队列的实例只有几个简单的封装函数。虽然它们也很有用，但是还不够深入。因此，我们下面将提供一个将消息队列应用于实际的例子——命令行程序 msgtool。使用它我们可以在命令行上提供消息队列的功能。

1．背景知识

msgtool 工具通过命令行参数来决定它的行为，这样它可以被方便的应用于 shell 脚本中。msgtool 提供了和消息队列有关的全部功能，包括创建、删除、更改消息队列以及收发消息等。不过，我们提供的这个版本只接收字符数组类型的数据，接收其它类型数据的功能请读者自行完成。

2．msgtool 的命令行语法

发送消息：

msgtool s (type) "text"

取得消息：

msgtool r (type)

修改权限：

msgtool m (mode)

删除消息队列：

msgtool d

3．msgtool 使用的例子

msgtool　s　　1 test
msgtool　s　　5 test
msgtool　s　　1 "This is a test"
msgtool　r　　1
msgtool　d
msgtool　m　　660

4．msgtool 的源码

下面是 msgtool 程序的源码。它必须在一个支持 System V IPC 的 Linux 内核上编译。请在编译前确认您所使用的内核在编译时打开了“支持 System V IPC”选项。

顺便说一句，msgtool 在执行任何命令时，只要消息队列不存在，它就自动创建一个。

注意：由于在 msgtool 中使用 ftok()来创建关键字，所以如果你在执行 msgtool 的 shell 脚本中改变了工作目录，msgtool 有可能不会工作。这个问题请读者自行解决。

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);


int main(int argc, char *argv[])
{
        key_t key;
        int     msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

       /* Create unique key via call to ftok() */
        key = ftok(".", 'm');
```

```
        /* Open the queue - create if necessary */
          if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                    perror("msgget");
                    exit(1);
          }

          switch(tolower(argv[1][0]))
          {
                    case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                                  atol(argv[2]), argv[3]);
                              break;
                    case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                              break;
                    case 'd': remove_queue(msgqueue_id);
                              break;
                    case 'm': change_queue_mode(msgqueue_id, argv[2]);
                              break;
                     default: usage();
          }
          return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
          printf("Sending a message ...\n");
          qbuf->mtype = type;
          strcpy(qbuf->mtext, text);

          if((msgsnd(qid, (struct msgbuf *)qbuf,
                    strlen(qbuf->mtext)+1, 0)) ==-1)
          {
                    perror("msgsnd");
                    exit(1);
          }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Read a message from the queue */
          printf("Reading a message ...\n");
          qbuf->mtype = type;
          msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
```

```
            printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
          /* Remove the queue */
            msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
            struct msqid_ds myqueue_ds;

          /* Get current info */
            msgctl(qid, IPC_STAT, &myqueue_ds);

          /* Convert and load the mode */
            sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

          /* Update the mode */
            msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
{
            fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
            fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
            fprintf(stderr, "                    (r)ecv <type>\n");
            fprintf(stderr, "                     (d)elete\n");
            fprintf(stderr, "                    (m)ode <octal mode>\n");
            exit(1);
}
```

4.8　信号量(Semaphores)

信号量简单的说就是用来控制多个进程对共享资源使用的计数器。它经常被用作一种锁定保护机制，当某个进程在对资源进行操作时阻止其它进程对该资源的访问。需要注意的是，System V **中的信号量对象实际上是信号量的集合（**set**），它可以包含多个信号量，控制多个共享资源。**

4.8.1 有关的数据结构

和消息队列一样，我们在介绍它的使用前将首先介绍一些有关的数据结构：

1．sem

前面提到，信号量对象实际是多个信号量的集合。在 Linux 系统中，这种集合是以数组的形式实现的。数组的每个成员都是一个单独的信号量，它们在系统中是以 sem 结构的形式储存的。Sem 结构在 Linux 系统库 linux/sem.h 中的定义是这样的：

```
/* One semaphore structure for each semaphore in the system. */
    struct sem {
            short     sempid;         /* pid of last operation */
            ushort    semval;         /* current value */
            ushort    semncnt;        /* num procs awaiting increase in semval */
            ushort    semzcnt;        /* num procs awaiting semval = 0 */
    };
```

其中，

sem_pid 成员保存了最近一次操作信号量的进程的 pid 。

sem_semval 成员保存着信号量的计数值。

sem_semncnt 成员保存着等待使用资源的进程数目。

sem_semzcnt 成员保存等待资源完全空闲的的进程数目。

2．semun

semun 联合在 senctl()函数中使用，提供 senctl()操作所需要的信息。它在 Linux 系统 linux/sem.h 中的定义是这样的：

```
/*  arg for semctl system calls. */
    union semun {
            int val;                /* value for SETVAL */
            struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
            ushort *array;          /* array for GETALL & SETALL */
            struct seminfo *__buf;  /* buffer for IPC_INFO */
            void *__pad;
    };
```

前三个参数在对 senctl()函数介绍中会讲到，这里暂时先不管它们。后两个参数是 Linux 系统所独有的，只在系统的内核中使用，我们就不多介绍了。

3．sembuf

sembuf 结构被 semop()函数(后面回讲到)用来定义对信号量对象的基本操作。它在 linux/sem.h 中是这样定义的：

```
/* semop system calls takes an array of these. */
struct sembuf {
        unsigned short  sem_num;        /* semaphore index in array */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

其中，

sem_num 成员为接受操作的信号量在信号量数组中的序号（数组下标）。

sem_op 成员定义了进行的操作(可以是正、负和零)。

sem_flg 是控制操作行为的标志。

如果 sem_op 是负值，就从指定的信号量中减去相应的值。这对应着获取信号量所监控的资源的操作。如果没有在 sem_flg 指定 IPC_NOWAIT 标志，那么，如果现有的信号量数值小于 sem_op 的绝对值（表示现有的资源少于要获取的资源），调用 semop()函数的进程就回被阻塞直到信号量的数值大于 sem_op 的绝对值（表示有足够的资源被释放）。

如果 sem_op 是正值，就在指定的信号量中加上相应的值。这对应着释放信号量所监控的资源的操作。

如果 sem_op 是零，那么调用 semop()函数的进程就会被阻塞直到对应的信号量值为零。这种操作的实质就是等待信号量所监控的资源被全部使用。利用这种操作可以动态监控资源的使用并调整资源的分配，避免不必要的等待。

4．semid_qs

和 msgqid_ds 类似，semid_qs 结构被系统用来储存每个信号量对象的有关信息。它在 Linux 系统库 linux/sem.h 中是这样定义的：

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
        struct ipc_perm sem_perm;               /* permissions .. see ipc.h */
        __kernel_time_t sem_otime;              /* last semop time */
        __kernel_time_t sem_ctime;              /* last change time */
        struct sem      *sem_base;              /* ptr to first semaphore in array */
        struct sem_queue *sem_pending;          /* pending operations to be processed */
        struct sem_queue **sem_pending_last;    /* last pending operation */
        struct sem_undo *undo;                  /* undo requests on this array *
/
        unsigned short  sem_nsems;              /* no. of semaphores in array */
};
```

其中，

sem_perm 成员保存了信号量对象的存取权限以及其他一些信息（见上面关于 ipc_perm 结构的介绍）。

sem_otime 成员保存了最近一次 semop()操作的时间。

sem_ctime 成员保存了信号量对象最近一次改动发生的时间。

sem_base 指针保存着信号量数组的起始地址。

sem_pending 指针保存着还没有进行的操作。

sem_pending_last 指针保存着最后一个还没有进行的操作。

sem_undo 成员保存了 undo 请求的数目。

sem_nsems 成员保存了信号量数组的成员数目。

4.8.2　有关的函数

介绍完有关的数据结构，接下来我们将介绍使用信号量要用到的函数：

1．semget()

使用 semget()函数来建立新的信号量对象或者获取已有对象的标识符。它在 linux/sem.h 中的函数声明是这样的：

系统调用：　semget()

函数声明：　int semget (key_t key, int nsems, int semflg);

返回值： semaphore set IPC identifier on success
-1 on error: errno = EACCESS (permission denied)
EEXIST (set exists, cannot create (IPC_EXCL))
EIDRM (set is marked for deletion)
ENOENT (set does not exist, no IPC_CREAT was used)
ENOMEM (Not enough memory to create new set)
ENOSPC (Maximum set limit exceeded)

函数接受三个参数。其中第一个参数 key **和第三个参数** semflg **和前面讲过的** msgget() **函数中的两个参数是对应的，作用和取值的意义也相同，读者可以参看** msgget()**的有关介绍。函数的第二个参数** nsems **是信号量对象所特有的，它指定了新生成的信号量对象中信号量的数目，也就是信号量数组成员的个数。在** linux/sem.h **定义了它的上限：**

```
#define SEMMSL  32                  /* <= 512 max num of semaphores per id */
```

如果函数执行的是打开而不是创建操作，则这个参数被忽略。

下面我们将创建一个封装函数作为本节例子：

```
int open_semaphore_set( key_t keyval, int numsems )
{
        int         sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }
        return(sid);
}
```

程序的分析请读者自行完成。

2．semop()

使用这个函数来改变信号量对象中各个信号量的状态。它在 Linux **系统库** linux/sem.h **中的函数声明如下：**

系统调用： semop()
函数声明： int semop (int semid, struct sembuf *sops, unsigned nsops);
返回值： 0 on success (all operations performed)
-1 on error: errno = E2BIG (nsops greater than max number of ops allowed atomically)
EACCESS (permission denied)
EAGAIN (IPC_NOWAIT asserted, operation could not go through)
EFAULT (invalid address pointed to by sops argument)
EIDRM (semaphore set was removed)
EINTR (Signal received while sleeping)
EINVAL (set doesn't exist, or semid is invalid)
ENOMEM (SEM_UNDO asserted, not enough memory to create the undo structure necessary)
ERANGE (semaphore value out of range)

函数的第一个参数 semid **是要操作的信号量对象的标识符。第二个参数** sops **是** sembuf **的数组，它定义了** semop()**函数所要进行的操作序列。第三个参数** nsops **保存着** sops **数组的长度，也即** semop()**函数将进行的操作个数。**

在前面对 sembuf 结构的介绍中我们已经介绍了 semop()的各种基本操作。下面我们将结合例子对这些操作作进一步的介绍。首先假设我们已经通过 semget()函数得到了一个只包含一个信号量的信号量对象，它监控着某台最多能处理 10 份作业的打印机的使用。我们下面的操作都将是只针对这个信号量的。

假设我们要向打印机交付一份作业。可以定义下面的 sembuf 变量来完成这个操作：

```
struct sembuf   sem_get = { 0, -1, IPC_NOWAIT };
```

它告诉系统，将信号量对象中序号为零的信号量（第一个信号量）减一。IPC_NOWAIT 标志的定义告诉系统，如果打印机的作业量（10 份）已满，则不阻塞进程而是直接将控制权返回进程并返回失败信息。

定义完操作后，我们使用下面的代码来执行它：

```
if((semop(sid, &sem_get, 1) == -1)
        perror("semop");
```

作业打印完成后，我们使用下面的 sembuf 变量来定义一个释放资源的操作：

```
struct sembuf sem_release= { 0, 1, IPC_NOWAIT };
```

它告诉系统将信号量对象中序号为零的对象加一。

然后用下面的代码来完成这个操作：

```
semop(sid,&sem_release,1);
```

这样，我们就完成了一个完整的作业打印操作。

3．semctl()函数

和消息队列的 msgctl()函数类似，semctl()函数被用来直接对信号量对象进行控制。它在 linux/sem.h 中的函数声明如下：

系统调用： semctl()

函数声明： int semctl (int semid, int semnum, int cmd, union semun arg);

返回值： positive integer on success

-1 on error: errno = EACCESS (permission denied)

EFAULT (invalid address pointed to by arg argument)

EIDRM (semaphore set was removed)

EINVAL (set doesn't exist, or semid is invalid)

EPERM (EUID has no privileges for cmd in arg)

ERANGE (semaphore value out of range)

比较一下这两个函数的参数我们回发现一些细微的差别。首先，因为信号量对象事实上是多个信息量的集合而非单一的个体，所以在进行操作时，不仅需要指定对象的标识符，还需要用信号量在集合中的序号来指定具体的信号量个体。

两个函数都有 cmd 参数，指定了函数进行的具体操作。不过，和 msgctl()函数相比，semctl()函数可以进行的操作要多得多：

IPC_STAT　取得信号量对象的 semid_ds 结构信息，并将其储存在 arg 参数中 buf 指针所指内存中返回。

IPC_SET　用 arg 参数中 buf 的数据来设定信号量对象的的 semid_ds 结构信息。和消息队列对象一样，能被这个函数设定的只有少数几个参数。

IPC_RMID　从内存中删除信号量对象。

GETALL　取得信号量对象中所有信号量的值，并储存在 arg 参数中的 array 数组中返回。

GETNCNT　返回正在等待使用某个信号量所控制的资源的进程数目。

GETPID　返回最近一个对某个信号量调用 semop()函数的进程的 pid。

GETVAL　返回对象那某个信号量的数值。

GETZCNT　返回正在等待某个信号量所控制资源被全部使用的进程数目。

SETALL 用 arg 参数中 array 数组的值来设定对象内各个信号量的值。

SETVAL 用 arg 参数中 val 成员的值来设定对象内某个信号量的值。

函数的第四个参数 arg 提供了操作所需要的其它信息。它的各个成员的意义在前面已经有过介绍，这里不再赘述。需要强调的是它和 msgctl()中的参数不一样，是一个普通的变量而不是指针，初学者常常在这个问题上犯错误。

下面举几个使用 semctl()的例子。

```
int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

上面的代码返回信号量对象中某个信号量的值。注意这里 semctl()函数的最后一个参数取的是零，这是因为执行 GETVAL 命令时这个参数被自动忽略了。

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
}
```

上面的代码用 initval 参数来设定信号量对象中某个信号量的值。

最后用一个例子来强调一个极易被忽视的错误。

在消息队列和信号量对象中，都有 IPC_STAT 和 IPC_SET 的操作。但是由于传递参数的类型不同，造成了它们在使用上的差别。在 msgctl()函数中，IPC_STAT 操作只是简单的将内核内 msgqid_ds 结构的地址赋予 buf 参数(是一个指针)。而在 semctl()函数中，IPC_STAT 操作是将 semid_ds 的内容拷贝到 arg 参数的 buf 成员指针所指的内存中。所以，下面的代码会产生错误，而 msgctl()函数的类似代码却不会：

```
void getmode(int sid)
{
        int rc;
        union semun semopts;

        /*下面的语句会产生错误*/
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Pemission Mode were %o\n", semopts.buf->sem_perm.mode);
        return;
    }
```

为什么呢？因为实现没有给 buf 指针分配内存，其指向是不确定的。这种“不定向”的指针是 C 程序中最危险的陷阱之一。改正这个错误，只需要提前给 buf 指针准备一块内存。下面是修改过的代码：

```
void getmode(int sid)
{
    int rc;
    union semun semopts;
    struct semid_ds mysemds;
    /*给 buf 指针准备一块内存*/
    semopts.buf = &mysemds;

    /*现在 OK 了*/
    if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
    {
        perror("semctl");
        exit(1);
    }

    printf("Permission Mode were %o\n", semopts.buf->sem_perm.mode);
    return;
  }
```

4.8.3　信号量的实例——semtool，交互式的信号量使用工具

1．背景知识

semtool 工具通过命令行参数来决定它的行为，这样它可以被方便的应用于 shell 脚本中。semtool 提供了和信号量有关的全部功能，包括创建信号量、操作、删除信号量对象以及更改信号量权限等。使用它我们可以在命令行上控制资源的共享。

2．semtool 的命令行语法

建立信号量对象：

semtool c (number of semaphores in set)

锁定信号量：

semtool l (semaphore number to lock)

解开信号量的锁定

semtool u (semaphore number to unlock)

改变信号量的权限

semtool m (mode)

删除信号量对象

semtool d

3．semtool 的使用举例

```
semtool   c  5
semtool   l
semtool   u
semtool   m  660
semtool   d
```

4．semtool 的源码

semtool 程序的源码如下：

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int     semset_id;

        if(argc == 1)
                usage();

       /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,   atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          locksem(semset_id, atoi(argv[2]));
                          break;
                case 'u': if(argc != 3)
                                usage();
```

```
                                opensem(&semset_id, key);
                                unlocksem(semset_id, atoi(argv[2]));
                                break;
                        case 'd': opensem(&semset_id, key);
                                removesem(semset_id);
                                break;
                        case 'm': opensem(&semset_id, key);
                                changemode(semset_id, argv[2]);
                                break;
                         default: usage();

                }

                return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

                if((*sid = semget(key, 0, 0666)) == -1)
                {
                        printf("Semaphore set does not exist!\n");
                        exit(1);
                }

}

void createsem(int *sid, key_t key, int members)
{
                int cntr;
                union semun semopts;

                if(members > SEMMSL) {
                        printf("Sorry, max number of semaphores in a set is %d\n",
                                SEMMSL);
                        exit(1);
                }

                printf("Attempting to create new semaphore set with %d members\n",
                                                members);

                if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                                == -1)
```

```
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
        }

        semopts.val = SEM_RESOURCE_MAX;

      /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}

void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

      /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }

        sem_lock.sem_num = member;

        if((semop(sid, &sem_lock, 1)) == -1)
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");

        dispval(sid, member);
}

void unlocksem(int sid, int member)
{
```

```
        struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
        int semval;

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

      /* Is the semaphore set locked? */
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }

        sem_unlock.sem_num = member;

      /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");

        dispval(sid, member);
}

void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}

unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;

        semopts.buf = &mysemds;

      /* Return number of members in the semaphore set */
```

```
        return(semopts.buf->sem_nsems);
}

int getval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
}

void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;

       /* Get current values for internal data structure */
        semopts.buf = &mysemds;

        rc = semctl(sid, 0, IPC_STAT, semopts);

        if (rc == -1) {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

       /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);

       /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");

}

void dispval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
```

```
        printf("semval for member %d is %d\n", member, semval);
}

void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:   semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "                  (l)ock <sem #>\n");
        fprintf(stderr, "                  (u)nlock <sem #>\n");
        fprintf(stderr, "                   (d)elete\n");
        fprintf(stderr, "                  (m)ode <mode>\n");
        exit(1);
}
```

4.9　共享内存(Shared Memory)

共享内存，简单的说就是被多个进程共享的内存。它在各种进程通信方法中是最快的，因为它是将信息直接映射到内存中，省去了其它 IPC **方法的中间步骤。**

4.9.1　有关的数据结构

下面我们来介绍几个和共享内存有关的数据结构：

1．shmid_ds

和前面介绍的两个 IPC **对象一样，共享内存也有一个给系统内存用来保存相关信息的结构，就是** shmid_ds**。它在** linux/shm.h **中的定义是这样的：**

```
struct shmid_ds {
        struct ipc_perm         shm_perm;       /* operation perms */
        int                     shm_segsz;      /* size of segment (bytes) */
        __kernel_time_t         shm_atime;      /* last attach time */
        __kernel_time_t         shm_dtime;      /* last detach time */
        __kernel_time_t         shm_ctime;      /* last change time */
        __kernel_ipc_pid_t      shm_cpid;       /* pid of creator */
        __kernel_ipc_pid_t      shm_lpid;       /* pid of last operator */
        unsigned short          shm_nattch;     /* no. of current attaches */
        unsigned short          shm_unused;     /* compatibility */
        void                    *shm_unused2;   /* ditto - used by DIPC */
        void                    *shm_unused3;   /* unused */
};
```

其中，

shm_perm **成员储存了共享内存对象的存取权限及其它一些信息。**

shm_perm **成员定义了共享的内存大小（以字节为单位）。**

shm_atime **成员保存了最近一次进程连接共享内存的时间。**

shm_dtime **成员保存了最近一次进程断开与共享内存的连接的时间。**

shm_ctime 成员保存了最近一次 shmid_ds 结构内容改变的时间。

shm_cpid 成员保存了创建共享内存的进程的 pid。

shm_lpid 成员保存了最近一次连接共享内存的进程的 pid。

shm_nattch 成员保存了与共享内存连接的进程数目。

剩下的三个成员被内核保留使用，这里就不介绍了。

4.9.2 有关的函数

接下来我们介绍和共享内存有关的函数：

1．sys_shmget()函数

使用 shmget()函数来创建新的或取得已有的共享内存。它在 Linux 系统库 linux/shm.h 中的定义是这样的：

系统调用： shmget()

函数声明： int shmget (key_t key, int size, int shmflg);

返回值： shared memory segment identifier on success
-1 on error: errno = EINVAL (Invalid segment size specified)
EEXIST (Segment exists, cannot create)
EIDRM (Segment is marked for deletion, or was removed)
ENOENT (Segment does not exist)
EACCES (Permission denied)
ENOMEM (Not enough memory to create segment)

和前面两个 IPC 对象的对应函数一样，shmget()函数的第一个参数 key 是共享内存的关键字；第二个参数 size 是创建的共享内存的大小，以字节为单位。第三个参数 shmflg 是控制函数行为的标志量，其取值的含义和作用和 msgget()及 semget()函数的对应参数都是相同的，这里不再赘述。

如果操作成功，函数返回共享内存的标识符。

下面的代码示范了 shmget()函数的使用：

```
int open_shm( key_t keyval, int segsize )
{
        int         shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

2．shmat()函数

当一个进程使用 shmget()函数得到了共享内存的标识符之后，就可以使用 shmat()函数来将共享内存映射到进程自己的内存空间内。shmat()函数在 Linux 系统函数库 linux/shm.h 中的函数声明如下：

系统调用： shmat()

函数声明： int shmat (int shmid, char *shmaddr, int shmflg);

返回值： address at which segment was attached to the process, or
-1 on error: errno = EINVAL (Invalid IPC ID value or attach address

passed)
ENOMEM (Not enough memory to attach segment)
EACCES (Permission denied)

第一个参数是共享内存的标识符。

第二个参数 shmaddr **指定了共享内存映射的地址。因为这样必须要预先分配内存，十分不便，所以我们在使用时常常将这个参数置零，这样系统会自动为映射分配一块未使用的内存。如果指定了地址，可以给第三个参数** shmflg **指定** SHM_RND **标志来强迫将内存大小设定为页面的尺寸。**

如果指定了 SHM_RDONLY **参数，共享内存将被映射成只读。**

映射成功后，函数返回指向映射内存的指针。

下面的这段代码演示了 shmat()**函数的使用：**

```
char *attach_segment( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

得到了映射内存的指针之后，我们就可以像读写普通内存一样对共享内存进行读写了。

3．shmctl()**函数**

和前两个 IPC **对象一样，共享内存也有一个直接对其进行操作的函数，就是** shmctl() **函数。它在** Linux **系统函数库** linux/shm.h **中的函数声明是这样的：**

系统调用： shmctl()
函数声明： int shmctl (int shmqid, int cmd, struct shmid_ds *buf);
返回值： 0 on success
-1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
EFAULT (Address pointed to by buf is invalid with IPC_SET and IPC_STAT commands)
EIDRM (Segment was removed during retrieval)
EINVAL (shmqid invalid)
EPERM (IPC_SET or IPC_RMID command was issued, but calling process does not have write (alter) access to the segment)

这个函数和 msgget()**函数十分相似，用法也相同。它支持的操作有：**

IPC_STAT　**获得共享内存的信息。**

IPC_SET　**设定共享内存的信息。**

IPC_RMID　**删除共享内存。**

需要说明的是，当执行 IPC_RMID **操作时，系统并不是立即将其删除，而只是将其标为待删，然后等待与其连接的进程断开连接。只有当所有的连接都断开以后系统才执行真正的删除操作。当然，如果执行** IPC_RMID **的时候没有任何的连接，删除将是立即的。**

4．shmdt()**函数**

当一个进程不再需要某个共享内存的映射时，就应该使用 shmdt()**函数断开映射。它在** linux/shm.h **中的函数声明如下：**

系统调用： shmdt()
函数声明： int shmdt (char *shmaddr);
返回值： -1 on error: errno = EINVAL (Invalid attach address passed)

shmdt()**函数唯一的参数是共享内存映射的指针。怎么样，是不是想起了** malloc()/free() **函数呢？**

4.9.3 共享内存应用举例——shmtool,交互式的共享内存使用工具

1．背景知识

shmtool 工具通过命令行参数来决定它的行为，这样它可以被方便的应用于 shell 脚本中。shmtool 提供了和共享内存有关的全部功能，包括创建、删除共享内存以及对其的读写等。和前面的例子一样，在任何操作中，只要共享内存不存在，它就自动被创建。

2．shmtool 的命令行语法

向共享内存写入字符串：

```
shmtool w "text"
```

从共享内存中读出字符串：

```
shmtool r
```

改变共享内存的权限：

```
shmtool m (mode)
```

删除共享内存：

```
shmtool d
```

3．共享内存使用举例

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

4．shmtool 的源码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

main(int argc, char *argv[])
{
        key_t key;
        int    shmid, cntr;
        char   *segptr;

        if(argc == 1)
                usage();

       /* Create unique key via call to ftok() */
        key = ftok(".", 'S');

       /* Open the shared memory segment - create if necessary */
        if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
```

```
                printf("Shared memory segment exists - opening as client\n");

            /* Segment probably already exists - try as a client */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creating new shared memory segment\n");
        }

    /* Attach (map) the shared memory segment into the current process */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'w': writeshm(shmid, segptr, argv[2]);
                                break;
                case 'r': readshm(shmid, segptr);
                                break;
                case 'd': removeshm(shmid);
                                break;
              case 'm': changemode(shmid, argv[2]);
                                break;
                 default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
        strcpy(segptr, text);
        printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
```

```
        printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

      /* Get current values for internal data structure */
        shmctl(shmid, IPC_STAT, &myshmds);

       /* Display old permissions */
        printf("Old permissions were: %o\n", myshmds.shm_perm.mode);

      /* Convert and load the mode */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);

       /* Update the mode */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - A utility for tinkering with shared memory\n");
        fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
        fprintf(stderr, "                (r)ead\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "               (m)ode change <octal mode>\n");
        exit(1);
}
```

4.9.4 共享内存与信号量的结合使用

在前面的介绍中可以看出，共享内存的使用非常的简便，只要取得映射的指针就可以直接存取，省去了其它 IPC 方式的传递过程，因此效率极高。不过，因此也产生了一些问题。如:一个进程修改一个共享内存单元，另一个进程在读该共享内存单元时可能有第三个进程立即修改该单元，从而会影响程序的正确性。同时，由于分时系统对各进程是分时间

片处理的，可能会引起不同的正确性问题。为了避免这些问题，在使用中常常将共享内存和信号量结合使用，利用信号量的保护机制来防止内存的不正确共享。

我们通过实例程序 shmcopy 来进一步说明共享存贮器的实际使用情况。shmcopy 的功能很简单：把其从标准输入读到的内容送到标准输出。每次调用 shmcopy 就形成两个进程——读进程和写进程。它们共享两个缓冲区，这两个缓冲区作为共享内存段来实现。当读进程把数据读入第一个缓冲区期间，写进程就把第二个缓冲区内容写出去，反之亦然。由于读和写是并发进行的，所以数据吞吐量增加了。这种方法可以用在磁盘高速缓冲程序中。

为了使两个进程同步，防止读进程装满缓冲区之前，写进程就把该缓冲区内容写出来，我们使用了两个信号量。几乎所有的共享存贮器程序都要使用信号量来实现同步。共享存贮器机构本身没有提供同步功能。

在程序 shmcopy 中使用了标题文件 share_ex.h，该文件的内容如下：

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

/* 共享内存的关键字 */
#define SHMKEY1_t (key_t)0x10
#define SHMKEY2 (key_t)0x15
/* 信号量的关键字 */
#define SEMKEY (key_t)0x20

/* 读写缓冲区的大小 */
#define SIZ 5*BUFSIZ

struct databuf{
    int d_nread;
    char d_buf[SIZ];
};
```

其中，符号常数 BUFSIZ 在头文件 stdio.h 中被定义，它是系统的磁盘块长度。结构 databuf 将强加给每个共享存贮器段，其中的 d_nread 域用于存放读进程所读的字符数，可以通过共享存贮器段把这个数传送给写进程。d_buf 域用于存放共享信息。

下面的清单中有三个例行程序，getseg()和 getsem()分别实现对两个共享存贮器段和信号量组的初始化。remove 用于在进程结束端删除各种 IPC 目标。

```
#include "share_ex.h"

#define IFLAGS (IPC_CREAT | IPC_EXCL)
#define ERR ((struct databuf*) -1)

static int shmid1,shmid2,semid;

fatal(char *mes)
```

```
{
    perror(mes);
    exit(1);
}

getseg(struct databuf** p1,struct databuf** p2)
{
    /* 创建共享内存 */
    if ((shmid1=shmget(SHMKEY1,sizeof(struct databuf),0600|IFLAGS
        )) < 0)
        fatal("shmget");

    if ((shmid2=shmget(SHMKEY2,sizeof(struct databuf),0600|IFLAGS
        )) < 0)
        fatal("shmget");

    /* 建立与共享内存的连接 */
    if ((*p1=(struct databuf*)(shmat(shmid1,0,0)))==ERR)
        fatal("shmat");

    if ((*p2=(struct databuf*)(shmat(shmid1,0,0)))==ERR)
        fatal("shmat");

}

int getsem()
{
    /* 建立信号量对象 */
    if ((semid=semget(SEMKEY,2,0600|IFLAGS))<0)
        fatal("semget");

    /* 初始化信号量对象 */
    if (semctl(semid,0,SETVAL,0)<0)
        fatal("semctl");

    if (semctl(semid,1,SETVAL,0)<0)
        fatal("semctl");

    return (semid);
}

remove()
{
    if(shmctl(shmid1,IPC_RMID,NULL)<0)
```

```
        fatal("shmctl");

    if(shmctl(shmid2,IPC_RMID,NULL)<0)
        fatal("shmctl");
    if(shmctl(semid,IPC_RMID,NULL)<0)
        fatal("semctl");
}
```

请注意以上程序中调用 shmat 把共享存贮器段连入进程的地址空间的方法。这些例行程序中用 fatal 来处理出错，它知识调用了 perror()，然后再调用 exit()。

shmcopy 的 main 函数很简单，它现实调用初始化例行程序，然后建立读进程（父进程）和写进程（子进程）。main 函数的清单如下：

```
#include "share_ex.h"

main()
{

    int semid,pid;
    struct datebuf *buf1,*buf2;

    /* 初始化信号量对象 */
    semid=getsem();

    /* 创建并连接共享内存 */
    getseg(&buf1,&buf2);

    switch(pid=fork()) {
        case -1:
            fatal("fork");
        case 0:
            writer(semid,buf1,buf2);
            remove();
            break;
        default:
            reader(semid,buf1,buf2);
            break;
    }

    exit(0);
}
```

请注意，程序结束是写进程调用了 remove()。另外，main()函数在调用 fork()前建立了 IPC 对象，这样，标识共享存贮器段的地址在两个进程中都是有意义的。

例行程序 reader 从标准输入读取内容，它从 semid 中得到信号量描述符，从 buf1 和 buf2 中得到两个共享存贮器段的地址。它的清单如下：

```
#include "share_ex.h"

struct sembuf p1={ 0,-1, 0 },
              p2={1,-1,0 },
              v1={0, 1,0 },
              v2={1, 1,0};
reader(int semid,struct databuf *buf1,struct databuf *buf2)
{
    for(;;) {

        /*  读入 buf1 缓冲区  */
        buf1->dnread=read(0,buf1->d_buf,SIZ);

        /*  同步  */
        semop(semid,&v1,1);
        semop(semid,&p2,1);

        /*  防止 writer 进程休眠  */
        if (buf1->d_nread<=0)
            return;

        buf2->d_nread(0,buf2-<d_buf,SIZ);

        semop(semid,&v2,1);
        semop(semid,&p1,1);

        if(buf2->d_nread<=0)
            return;
        }
}
```

这儿的 sembuf **类型结构为一个拥有两个信号量的信号量组定义了** P、V **操作。**P、V **操作在这里不是用于锁定代码的临界区，而用于读进程和写进程同步。**reader **用** v1 **来发生一个读操作已完成的信号，并调用带** p2 **的** semop **来等待** writer **的写操作已完成信号。**

shmcrop **调用的最后一个例行程序是** writer()**，其清单如下：**

```
#include "share_ex.h"

extern struct sembuf p1,p2;
extern struct sembuf v1,v2;

writer(int semid,struct databuf buf1,struct databuf buf2)
{
    for(;;) {
        semop(semid,&p1,1);
```

```
        if (buf1->d_nread<=0)
            return;

        write(1,buf1->d_buf,buf1->d_nread);

        semop(semid,&v1,1);
        semop(semid,&p2,1);

        if (buf2->d_nread<=0)
            return;

        write(1,buf2->d_buf,buf2->d_nread);
        semop(semid,&v2,1);

}
```

这儿要注意 reader()**和** writer **对信号量组使用的一致性。**writer()**用** v2 **来发出一个写操作已完成的信号，并调用** p1 **来等待** semop **的读操作已完成信号。另外，必须注意下列情况：**buf1->d_nread **和** buf2->d_nread **之值是由读进程设置的。**

shmcopy **作为命令的使用方法如下：**

shmcopy <input file >output file

第五章　通信协议简介

5.1　引言

通信协议用于协调不同网络设备之间的信息交换，它们建立了设备之间互相识别的有价值的信息机制。当今在通信界有许多可采用的协议，如 XNS、SNA、TCP/IP 等。这些协议具有处理各种不同数据通信类型类型的几种基本结构。

在小型工作群和办公部门，局域网正在成为企业极计算的主要平台，过去只有简单的二十个用户的网络现已发展成能包含几千个用户的全企业网络，并可将许多不同的办公部门连在一起。

为了配合这种新要求，LAN 协议变得越来越有力和灵活。本章将介绍几种最广泛使用的协议，比较详细的讨论它们的结构和经过网络进行通信的方法。

5.2　XNS（Xerox Network Systems）概述

Xerox Network Systems（XNS）由 Xerox 公司的 Palo Alto 研究中心（Xerox PARC）的研究人员开发。起初被设计成连接用于 Xerox 环境的计算机。XNS 是许多局域网结构的基础设计，对 Novel 和 3com 等公司后来的设计有很大影响。

作为多重 LAN 结构基础的 XNS，其主要特点是设计清楚而简单，实现起来相对方便。

XNS 是最早一种分层网络结构，比 OSI 参考模型出现得更早，对后来的 7 层 OSI 具有一定的启发作用。

XNS 结构是为它自己的用户阻止专用而设计的，主要服务于 PARC 的研究人员。用于网络上的电子邮件、电子资料交换和资料的远程打印等项业务。这意味着各个站点比较靠近。由于存在着距离较近的有利条件，因此设备之间的通信可设计得比较快而可靠。

在 XNS 结构的情况下，可以互联几种类型的设备，特别是工作站和路由器。局域网络内部的工作站互相连接，各个局域网络又可互联，形成成为网中网的较大系统。路由器通常放在各个局域网络的交界处，用于在子网间路上选择数据报文。

5.2.1　XNS 分层结构

XNS 结构由五层组成。它和 OSI 模型相似，较高层使用较低层提供的服务，完成较低层的任何请求。这种协议提供了一个服务范围，包括报文传递、请求和回答包交换，以及排序的包和字节流。层的编号是从 0 到 4。第 0 层是与网络通信媒体相互作用的最低层，第 4 层包括网络应用和服务。

1．第四层

第四层是 XNS 的最高层。与 OSI 模型的应用层（第 7 层）相对应。在这一层规定了两个协议和服务。

- 交换服务（Clearing house）。

执行分布的名字服务，提供给 XNS 连接的计算机一种使资源和计算机与名字相联系或相结合的方法。

- 网关存取协议（GAP）。

它提供使 XNS 系统连接到非 XNS 系统的网关服务。文件、打印和报文服务通常归第四层

2．第三层

XNS 的第三层与 OSI 模型的对话层和表示层（第 5 层和第 6 层）相一致。该层所规定的协议称为信使，信使控制存取网路或网际上的远程过程调用。远程过程调用（RPC）是类似本地过程调用的网络扩展。它们能让应用程序存取文件，并能远程现实和打印网络上的其它资源。

3．第二层

XNS 的第二层与 OSI 模型的传输层（第 4 层）相一致。该层除包含被认为是传输层协议的协议外，还有以下几个协议：

- 排序包协议（SPP）。

它是一种全双工的传输级协议。这个协议考虑到了两个站点之间的可靠数据交换。协调发送方和接收方之间的同步并在它们之间建立连接。在这个处理期间家里排序好，标记在它们之间建立的包，在包级的基础上完成这种排序。

- 包交换协议（PEP）。

它是一种请求和响应协议。如果在规定的时间后没有接收到响应，则重传包。它使用在面向事务的应用程序中，因为这种应用程序要求简单而快速的信息交换协议。

- 路由选择信息协议（RIP）。

类似于其它的 RIP 实现，在距离向量算法的基础上提供路由选择表的动态修改。应注意这个规范没有规定特定的路由选择算法，但是大多数厂家正在实现它们，它的修改时间间隔是 60 秒。

- 差错协议。

允许一个网络站点通知另一个网络站点被接收的包已经发生差错。

- 回送协议（Echo）。

这个协议考虑到了网络通路的测试或记录的来回旅程时间。

4．第一层

XNS 的第一层符合 OSI 模型的网络层（第 3 层）。第一层本质上执行路由器的功能。在源和目的地包地址的基础上经过网络路由选择包。用于实现这种服务的协议被称为网际数据报文协议（IDP）。这个协议的目的是寻址、路由选择和传递标准网际包。它为在网际上传输和接收数据提供传递系统。这个过程传递无连接型服务。

5．第零层

第零层与 OSI 模型的物理和数据链路层（第 1 层和第 2 层）相一致，Xerox 称这层的协议为传输媒体协议。和 OSI 模型的情况一样，第零层负责为经过物理通信媒体传输和接收数据。这层支持的协议有 Ethernet、RS-232-C 和 RS-449。此外，X.25 被认为是传输媒体协议。

XNS 网际可以由几个不同的相连网络组成。由称为路由器的计算机将这些网络连在一起。路由器能在连接的网络之间选择路由传输数据包。每个网络可以在理论上有无数的被连接的工作站，每个站点可以运行多重应用程序。

为了正确识别通信终点，XNS 使用以下三个编号进行网际寻址：

- 网络号。它是 32 位的编码，用于识别网络（子网）。
- 主机（可以是网络站点）号。正确识别网络接口卡（NIC），以及安装 NIC 主机的 48 位地址。这种 48 位地址是世界上唯一的地址，不能配置和修改。
- 插座（socker）号。插座是网络中通信的最终电，它允许工作站中的网络协议了解数据的目的地。插座号通常指端口号，但它的含义大体上是端口号、主机和网络号的组

合。它是一个应用程序用于识别通信终点的 16 位值。

虽然在整个网际上主机号是唯一的，似乎可以不用网络号，但是由于网络号通常比主机上，如果以来网络号进行路由选择，则能减少路由选择表的规模，而且能以较快的速率交换包，因此网络号是必要的。使用较小的表还可以减少修改，维持路由选择表所需要的开销。

5.3 IPX/SPX 协议概述

Novell Netwate 协议是以 XNS 协议为基础的，也是分层结构。甚至寻址方式也和 XNS 非常一致。Netware 协议主要包括 IPX、SPX、RIP、NCP 和 NetBIOS 仿真。

和 Xerox 的网际数据报文协议与排序包协议相对应的 Novell 实现称为网际包交换（IPX）和排序包交换（SPX）。SPX 是一个可靠的，面向连接的协议。IPX 是不可靠的数据报文协议。

为了查询、维护工作站和服务器上的路由选择表，Netware 使用了路由选择协议（RIP）的修改的版本。

Novell 采用了 XNS 的网络寻址结构。由网络编号，主机编号和主机上的插座编号提供了完全的地址。网络编号长度是 32 位，主机编号是 48 位、插座编号是 16 位。

5.3.1 网际包交换（IPX）

网际包交换（IPX）是 Novell 的网络分层协议。IPX 是从 XNS 的网际数据报文协议推导而来的，具有相同的包结构。

IPX 提供给工作站和服务器非连接型的，不可靠的数据报文服务。为了使包传递到目的地，IPX 竭尽全力，但不要求确认信息，IPX 依赖于高层协议（例如 SPX 或 NCP）提供可靠的排序数据流服务。

IPX 由帧头和数据部分组成，帧头 30 字节长。由于 IPX 不为包的分段提供任何设施，因此 IPX 实现方法必须确保它们所发送的包是足够的小，能在它们需要经过的任何网络上传输。IPX 要求所有链路能处理 576 字节长的 IPX 包，因此不能发送大于 576 字节的包。但如果链路能处理大于 576 字节的包，则可使用较大的包。IPX 包结构如表 5-1 所示：

表 5-1 IPX 包结构

校验和	2 字节
长度	2 字节
传输控制	1 字节
包类型	1 字节
目的地网络	4 字节
目的地主机	6 字节
目的地插座	2 字节
源网络	4 字节
源节点	6 字节
源插座	2 字节
数据	多达 546 字节

下面讨论某些个别字段

- 校验和。

这个字段必须包含值 FFFFH（H 为十六进制标志）。校验和可看成是最高档的奇偶校验。其目的是确保所发送的位与所接收的位相同。在传输期间不变动包中的位。

- 长度。

包含整个 IPX 数据报文，帧头和所包括的数据的字节数。包的最小长度是 30 字节（帧头），最大长度是 576 字节（如果包通过网际路由器）。

- 传输控制。

用于确保数据报文绝不无限的循环，该字段起初由发送站点设置为“0”，它技术条约的次数（经过路由器的数量）。当计数值等于 16 时，放弃该数据报文。

- 包类型。

使 IPX 能向相应的较高层协议传递包。该字段用于只是数据字段中的数据类型，由于 IPX 是以 XNS 的 IDP 协议的推导结果，因此它遵循 Xerox 给出的类型。IPX 规定了以下的四种包类型：

表 5-2 IPX 包类型

未知的包类型	0（hex） 类型
包交换协议	4（hex） 类型
排序包类型	5（hex） 类型
Netware 核心协议	B（hex） 类型

网际地址由网络地址，主机地址和插座地址组成，插座地址识别主机（工作站）上的通信进程。IPX 支持每个站点多达 50 个插座，0 的网络地址表示本地网络。

为了提供重要的网络带宽服务，Novell 规定了几个著名的插座地址：

表 5-3 几个常用插座及其含义

文件服务器	
0451H	Netware 核心协议（NCP）
路由器固定插座	
0452H	服务公告协议（SAP）
0453H	路由选择信息协议（RIP）
工作站插座	
4000H-6000H	用于工作站与文件服务器交互作用和其它网络通信的动态分配的插座
0455H	NetBIOS
0456H	诊断包

- 数据。

包含经过网络发送的信息。应用程序可以使用 API 插座上放置数据，并使用 IPX 将它发送到远程计算机，较高层协议（例如 SPX、NCP、RIP 和 SAP）的包被放入 IPX 包的数据部分。在数据部分填充较高层协议包的这种处理被称为封装。

IPX 与数据链路层连接，为了通过网络传输，数据链路层将数据编制成适于跨网传输的帧。IPX 提供给数据链路层的目的地主机地址取决于被连到本地网络，还是被连到远程网络。

如果目的地站点处于本地网络内，则将这个站点地址交付给带有 IPX 包的数据链路层，以便直接传递到目的地站点，如果目的地站点和源站点不在同一网络内，则还需做额外的工作。

在这种情况下，IPX 发送路由选择请求包，使用路由选择信息协议（RIP）来确定发送包的最佳路由。然后由对方返回一个确认信息。包括能向目的地网络转送包的路由器的主机地址。在具备这种信息的情况下，本地和远程计算机的源和目的地地址被传递到链路层，以便能立即传递到这个站点（路由器）。

当路由器接收包时，它确定是否将包连接到储存在 IPX 包那的目的地网络，如果需要连到目的地网络，则直接传递包，否则路由器咨询它的路由选择表，以确定下一个路由器向哪里传送包。IPX 包的源和目的地址始终不便。在目的地站点上，IPX 将包传递到适当

的插座。

5.3.2 排序包交换（SPX）

排序包交换（SPX）是 Novell 的传输层协议。它起源于 Xerox 的排序包协议（SPP）。

SPX 在网络站点之间提供可靠的面向连接的虚拟电路服务。SPX 利用 IPX 的数据报文服务提供排序数据流。它通过实现一个特定的系统，保证每个被发送的包已被确认来完成上述功能。此外它还在网络站点之间提供流量控制，并确保没有重份被传递到远程进程。

为了降低网络上的拥塞状况，SPX 减少了不必要的重传发生次数。重传多半发生在发送站点超时等待确认的情况下，SPX 使用了一个特定的算法，能准确估算重传时间，从而减少了试重传次数。

为了装载连接控制信息，SPX 向 IPX 包帧头部分增加 12 字节，使原 30 字节的帧头变成 42 字节的组合帧头。SPX 的帧头格式如下：

表 5-4 SPX 帧头格式

连接控制	1 字节
数据流类型	1 字节
源连接 ID	2 字节
目的地连接 ID	2 字节
顺序编号	2 字节
确认编号	2 字节
分配编号	2 字节
数据	多达 534 字节

- 连接控制。它是控制链路上数据流的一组标记。它可以表明报文结束，并请求确认。
- 数据流类型。它使较高层协议能赋予包中数据含义。该字段中信息的意义取决于使用这个字段的协议。
- 源连接 ID 和目的地连接 ID，能唯一识别同层进程之间的连接，通过使用连接 ID，一个 IPX 插座可多路转换多重连接。
- 排序编号。用于实现 SPX 的流量控制，包排序和抑制复制的包。
- 确认编号。由目的地使用。用于通知发送方它所等待的下一个顺序编号。
- 分配编号。用于通知发送方接收方有多少连接可使用的缓冲器，此外这个编号还帮助实现连接的流量控制，使发送方和接收方同步。

5.4 Net BIOS 概述

网络基本输入/输出系统（Net BIOS）是高级应用程序接口，被设计成程序员使用 IBM PC 网络能建立网络应用程序。它是由 Sytek 公司开发的，起初在 IBM PC 网络适配器卡上实现。1984 年被 IBM 引入。Microsoft 使用它来配合它的 MS Net 产品。后来 IBM 提供了仿真程序，使 Net BIOS 能与 NIC 配合，同 Token-Ring 网络一起工作。

当今几乎所有的网络公司，包括 IBM、Novell、Microsoft 和 3Com，都支持 Net BIOS 接口。特别是 IBM 的 LAN Server 操作系统和 Microsoft 的 LAN Manager。Novell 公司通过使用 IPX 协议栈提供 Net BIOS 支持。3Com 的实现是使用它的 XNS 协议栈。无论怎样实现，支持网络分布应用程序的 Net BIOS 接口都保持了一致性。

目前已经使用 Net BIOS 接口编写了大量的网络应用程序。由于它的流行性，估计将继续是一种开发分布应用程序的有力工具。

Net BIOS 不是一个协议，更确切的说它是一个接口。它使网络应用程序和一组命令相

结合，从而建立通信对话、发送、接收数据和命名网络对象（目标）。正如从 OSI 模型上所看到的，Net BIOS 提供了对话层接口。在这层上 Net BIOS 能提供可靠的，面向连接的数据传输流，并识别网络上站点的名字系统。此外 Net BIOS 还提供了非连接型数据报文服务。但 Net BIOS 不提供路由选择服务，因而使构造网络变得很困难。

为了分配和管理网络上站点的名字，Net BIOS 提供了一组命令，名字长 16 字符，一个名字表内可以有 1-254 个名字。一个被称为 Net BIOS_NAME_NUMBER_1 的名字总存在于名字表内，该名字的前 10 个字节是“0“，后 6 个字节（48 位）是网络接口卡的地址。Net BIOS 使得网络上两个命名的对象能建立连接。

Net BIOS 名字可以是某一对象的唯一名字或小组名字。前者在整个网络是唯一的，不能被复制。小组名字可由网络上较多的站点使用。

当应用程序希望同远程应用程序建立联系时，使用远程站点的名字启动调用，然后使用建立的对话在应用程序之间交换数据。

5.5 Apple Talk 概述

Apple Computer 于 1983-1984 年开始设计一组称为 Apple Talk 的通信协议。目的是将它们的新型 Macintosh 个人计算机系统（包括打印机、打印服务器、文件服务器、路由器和网关）连到其它厂家生产的计算机系统。

起初 Apple Talk 的主要应用是将图形计算机连到新型 Apple Laser Writer 激光打印机。从那时起在先进的软件和硬件配合下，Apple Talk 所提供的较灵活的基本网络支持发回了重要作用，使桌面排版系统得到很大改进。

每台 Macintosh 和 Laser Writer（激光打印机）都具有适于 Apple Talk 联网结构的内部硬件支持。此外，系统软件还包括网络继承化支持。

Apple Talk 被设计成能以多种方式使外部设备联到 Machitosh 层。它是一个完全开放型的的，可扩展的网络结构，支持各种新型的物理网络计数和新型协议栈。将本地网络联入互联网可实现广域的大量计算机和外部设备的连接。

此外，Apple Talk 还被设计成支持同级间联网。在每个网络节点上实现这种服务，以分布方式操作。

Apple Talk 网络设计吸收了 Macintosh 的总体设计思想。与网络互相作用的各个用户是尽可能透明的，以便使标准操作能使用网际上的多种共享资源。尤其重要的是用户能在它们的台式机上安装卷，通过使用标准选择等操作能利用文件。

网络节点的安装被设计的相当灵活，用户只需连接物理链路，大部分配置由系统软件自动管理。

该公司于 1989 年 6 月引入了 Apple Talk Phase 2（第 2 阶段），作为现有的 Apple Talk 的向下兼容扩展。通过启动网络上的大量节点（工作站、打印机、服务器），Apple Talk Phase 2 提供了较大的全企业级网络支持。原有的 Apple Talk 最多支持一个网络上的 254 个节点，而 Phase 2 能启动与一个网络相关联的多个子网，每个子网最多包括 253 个节点。

Apple Talk Phase 2 还提供对 Local Talk, Ether Talk 和 Token Talk 的支持（Local Talk 是 Apple 的物理层和数据链路层规范，Ether Talk 和 Token Talk 分别是 Ethernet 和 Token Ring 的 Apple 实现）。Apple Talk 网际路由器被修改得能启动多达 8 个 Apple Talk 网络的连接（以任何 Local Talk, Ether Talk 和 Token Talk 的混合方式）。

开发者们已经使 Apple Talk 能支持多种类型的分布应用程序，支持 Apple MAC、IBM PS/2 和 PC 兼容机，运行 UNIX 的工作站和 Apple IIGS。这些应用程序包括文件共享、打印支持和电子报文。此外这些应用程序也可使用在 Apple Talk 网络连接的 DEC 小型机上。

在 Linux 内核中也有对 Apple Talk 的支持。

5.6 TCP/IP 概述

最近几年来分布处理的发展趋势已经从原来的主要依赖大中型机，转变成由 LAN 连接的 PC 机。这些 PC 机网络往往由许多不同类型的计算机和其它多售主设备组成。这种情况给一度曾支配大型主机通信的专有通信协议带来了严重冲击。专有协议只允许同一厂家的计算机系统之间互相通信。例如，DEC 的计算机只能和 DECnet 通信；IBM 主机只能与 SNA 通信；Apple 计算机只能与 Apple Talk 通信。

当今的分布式的 LAN 需要把一个组织的大型机、小型机、企业级 LAN 和公共数据网连在一起。无疑这将对开发互联多种不兼容系统的非专有协议是一个很大的促进。

当前出现的主要非专有网际互联产品是 TCP/IP 和 OSI 协议。OSI 七层协议已经在国际上通用。TCP/IP 许多年来就一直被人们所采用，而且越来越成熟，已称为被广泛接收的协议。据不完全同级，目前世界上已有大约 40 多个国家使用这种协议连接了 5000 多个网络，几十万台主机。这充分反映了 TCP/IP 的成熟和广泛的可用性。大多数类型的计算机环境都有 TCP/IP 产品，它提供了文件传输、电子邮件、终端仿真、传输服务和网络管理。

从长远的观点看，OSI 标准最终会超过 TCP/IP，提供较完善的目录和网络管理服务。但是 OSI 承诺的网际互联设备目前还只限于 LAN 标准的 IEEE802 系列。虽然 OSI 的报文处理服务（MHS）或 X.400 电子邮件；生产自动协议；文件传送、存储和管理（FTAM）以及虚拟终端（VT）能以各种不同形式采用，但不同厂家的产品是不能互相操作的。这种情况有利于 TCP/IP 的流行。

5.6.1 TCP/IP 结构模型

TCP/IP 实际上一个一起工作的通信家族，为网际数据通信提供通路。为讨论方便可将 TCP/IP 协议组大体上分为三部分：

1．Internet 协议（IP）

2．传输控制协议（TCP）和用户数据报文协议（UDP）

3．处于 TCP 和 UDP 之上的一组协议专门开发的应用程序。它们包括：TELNET，文件传送协议（FTP），域名服务（DNS）和简单的邮件传送程序（SMTP）等许多协议。

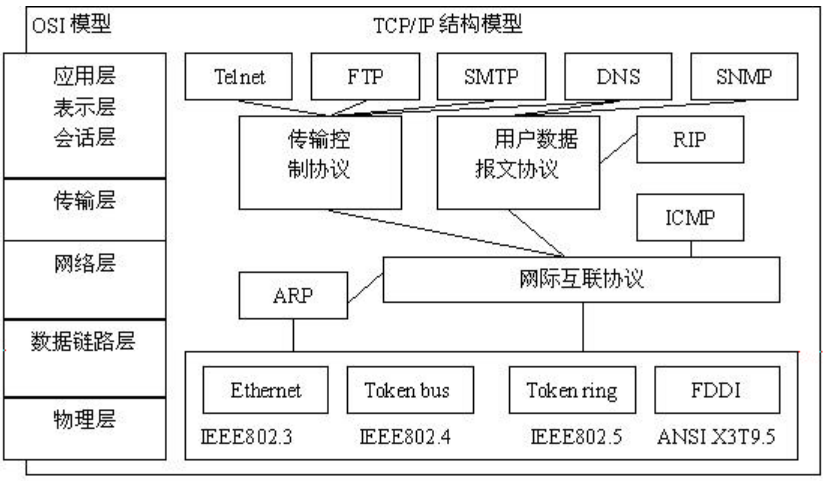

图 5-1 TCP/IP 和 OSI 网络模型的分层图

以下分别简介这三部分：

第一部分也称为网络层。包括 Internet 协议（IP）、网际控制报文协议（ICMP）和地址识别协议（ARP）.

- Internet 协议（IP）。

该协议被设计成互联分组交换通信网，以形成一个网际通信环境。它负责在源主机和目的地主机之间传输来自其较高层软件的称为数据报文的数据块，它在源和目的地之间提供非连接型传递服务。

- 网际控制报文协议（ICMP）。

它实际上不是 IP 层部分，但直接同 IP 层一起工作，报告网络上的某些出错情况。允许网际路由器传输差错信息或测试报文。

- 地址识别协议（ARP）。

ARP 实际上不是网络层部分，它处于 IP 和数据链路层之间，它是在 32 位 IP 地址和 48 位局域网地址之间执行翻译的协议。

第二部分是传输层协议。包括传输控制协议和用户数据报文协议。

- 传输控制协议（TCP）。

由于 IP 提供非连接型传递服务，因此 TCP 应为应用程序存取网络创造了条件，使用可靠的面向连接的传输层服务。该协议为建立网际上用户进程之间的对话负责。此外，还确保两个以上进程之间的可靠通信。它所提供的功能如下。

1．监听输入对话建立请求。

2．请求另一网络站点对话

3．可靠的发送和接收数据。

4．适度的关闭对话。

- 用户数据报文协议（UDP）。UDP 提供不可靠的非连接型传输层服务，它允许在源和目的地站点之间传送数据，而不必在传送数据之前建立对话。此外，该协议还不使用 TCP 使用的端对端差错校验。当使用 UDP 时，传输层功能全都发回，而开销却比较低。它主要用于那些不要求 TCP 协议的非连接型的应用程序。例如，名字服务和网络管理。

最后是应用程序部分。这部分包括 Telnet，文件传送协议（FTP 和 TFTP），简单的文件传送协议（SMTP）和域名服务（DNS）等协议。

TCP/IP 使用了主干网络，能连接各种主机和 LAN 的多级分层结构，局部用户能方便的联网，不致影响到整个网络系统。此外这种结构还有利于局部用户控制操作和管理。

TCP/IP 具有两个主要功能。第一是 IP 在网络之间（有时在个别网络内部）提供路由选择。第二是 TCP 将 TP 传递的数据传送的接收主机那的适当的处理部件。

5.6.2 Internet 协议（IP）

IP 的主要目的是提供子网的互联，形成较大的网际，使不同的子网之间能传输数据。网际是由许多自治系统组成的，每个系统是一个中央管理的网络或是一系列的子网，每个自治系统提供用于连到其它自治系统的网关。IP 规定如何连接子网和互联设备如何工作。

IP 规定包如何从一个子网路由选择到另一个子网。自治系统中的每个节点具有唯一的 IP 地址。IP 使用本身的帧头和检查来确保数据报文的正确传送。由于有维持当前路由选择表的路由选择修改信息，从而帮助了这个过程的顺利完成。路由选择表列出了子网上各种不同节点之间的通路和通路开销，如果个别节点之间有较多的通路，则可选择最经济的一条。如果数据包过大，使目的地不能接收，则将它分成较小的段。当从 LAN 向 WAN 传输数据时，包的分段是特别重要的。例如 Token-Ring LAN 能支持 4500 字节的包，而 X.25 分组网通常只支持 128 字节的包，因此必须进行分段。

归纳起来 IP 主要有以下四个主要功能：

（1）数据传送

（2）寻址

（3）路由选择

（4）数据报文的分段

1．数据传送的基本特点（无连接的最佳努力服务）

IP 层使用于经过网际传递数据的通路进入传递系统。当人们一听到 IP 这个名字时自然会联想起将许多子网连在一起的通称为路由器的设备。IP 的确是执行路由器相关的任务。但正如上面所提到的，IP 的功能不只是这些，它还完成许多其它工作。IP 协议运行在连接子网的所有参与网络的站点机上，以使各个站点能将它们的数据包递交给路由器或传送给同一网络上的其它设备。IP 协议处于数据链路层和传输层之间。

IP 的主要目的是为数据输入/输出网络提供基本算法，为高层协议提供无连接的传送服务。这意味这在 IP 将数据递交给接收站点以前不在传输站点和接收站点之间建立对话（虚拟链路）。它只是封装和传递数据，但不向发送者或接收者报告包的状态，不处理所遇到的故障。这意味这如果数据链路故障或遭遇可恢复的错误时，IP 不予通知和处理。它将报文和错误一起传出去，由高层协议（TCP）负责执行消除差错。换句话说，TCP 可能反复传输和发送数据。

IP 将正确格式化的数据包递交给目的地站点，不期待状态相应。由于 IP 是无连接的协议，因此它可能接收和传递发送给它的错误序列的数据。此外它还可能发送复份的数据。提供消除差错的过程是高层协议的责任。IP 是网络传递系统的一部分。它接受并格式化数据，以便传输到数据链路层。此外 IP 还检索来自数据链路的数据，并将它送给请求的高层。IP 传送的信息单元被称为“数据报文”，这种数据报文可经过告诉网络（Ethernet Token Ring 或 FDDI）传送，当经过这类网络传送时，数据报文被组装称为包。

IP 协议不注意包内的数据类型，它所知道的一切是必须将某些称为 IP 帧头的控制协议加到高层协议（TCP 或者 UDP）所接受的数据上，并试图把它传递给网络或者网际上的某

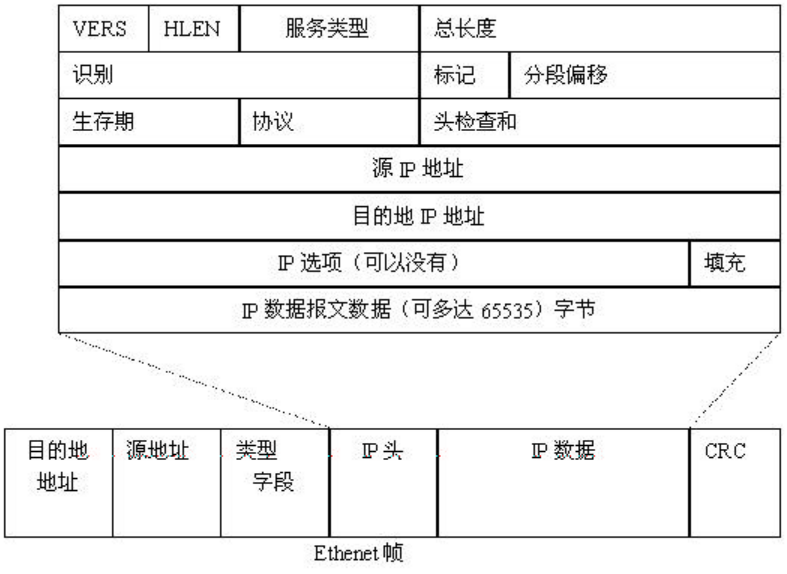

图 5-2　封装在 Ethernet 帧中的 IP 头

些节点。

IP 协议向主机和路由器提供应如何处理被传输或被接受的包的机制。为了了解 IP 的功能，请观察它向包增加的控制信息（IP 头）。见图 5-2。该图表示 IP 头北包装在 Ethenet 帧

内。从中可以看到 IP 帧头在包中的位置。

图 5-2 的上面部分是 IP 帧头。这时一个 IP 数据报文的标准包头。下面我们将通过观察 IP 数据报文中的帧头信息来研究 IP 数据报文传递的功能。各字段定义如下：

- VERS 是 4 位，规定网络站点所实现的 IP 当前版本。
- HLEN 是 IP 头的长度，共 4 位。在实际使用中，并不是必须使用 IP 头的所有位，所以需要该字段来指明 IP 头的长度。以 32 位表示字，IP 头的长度以字为增量变化，最短的 IP 头是 20 字节（不包括数据和选项），因此这个字段的值是 5（20 字节=160 位；160 位/32 位=5），也就是 5 个字。如果这个字段的值变成了 6，就等于增加了 32 位（一个字）。
- 服务类型 8 位，它可以细分为如下的形式：

优先权	D	T	R	未使用

优先权字段 3 位，可以有 0-7 的值（0 为正常值，7 为网络控制）。它允许传输站点的应用程序向 IP 层发送数据报文的优先权。该字段与 D（时延）、T（吞吐量）和 R（可靠性相结合，这些位向路由器表明应采取哪个路由。这个字段被称为 Type of Service（TOS）服务类型标识符。

D 位—当设置为 1 时请求低时延

T 位—请求高吞吐量时置 1。

R 位—请求高可靠性时置 1。

例如，如果去目的地有两个以上的路由，路由器将读这个字段，以选择一个正确的路由。由应用程序（即 Telnet 或 Ftp）设置 TOS 字段，路由器只读这种字段，不负责设置。在读信息的基础上，路由器将选择数据报文的最佳路由。在网络上传输包以前，由运行在主机上的 TCP/IP 应用程序设置这些位。它不要求路由器维持许多路由选择表。

- 总长度（16 位）。
- 这是以字节度量的数据报文长度。IP 数据报文的数据区可以有 65535 字节长（包括头和数据部分）。

2．分段包

有时会出现从一个网络传出的包大得不能传入另外网络的情况。例如考虑从 TokenRing 网络（典型情况是支持 4472 字节的最大传输包）向 Ethernet LAN（只支持 1518 字节的最大传输包）传输帧。TCP/IP 路由器必须能将较大的包破碎成较小的包。TCP 将建立适于连接的包大小，但如果两个通信站点被多种类型的媒体分开，那么将怎样支持不同的传输包大小呢？将包分裂成适合于 LAN 传输或异机种 WAN 路由选择是 IP 层完成的另一任务。使用下面的字段完成这方面的工作。

- 识别、标志和分段偏移（分段控制，共 16 位）

这些概念表明如何分段被传送的太大的数据报文。TCP/IP 几乎可以运行在任何数据链路上，当向不同的网络发送数据时，可以同时发送的数据的最大规模（包大小）在那些网络上可以发生变化。Ethernet 的最大包长度为 1518 字节（包括所有的帧头）,Token-Ring 是 17800 字节（16Mbps）和 4472 字节（4Mbps），FDDI 考虑到 4472 字节数据规模。任何一个网络都可能通过最大的帧，IP 考虑了它能接受多大的包，可以满足所有这些网络之间数据交换的需要。

每个被分裂的数据报文的 IP 头几乎是相等的。它识别哪些数据报文属于一个小组，确保数据报文不适配。接受 IP 会使用这个字段和源 IP 地址来识别哪些数据报文应归属在一起，使用标记完成如下任务：

（1）标志出是否出现了较多的分段。

（2）是否将一份数据报文分段。

如果所经过的网络使用不同的帧长度，那么将包分段就是特别重要的。了解网桥的读

者知道，网桥没有这种能力，如果网桥接受了太大的包（传送网络不能传递），正如 IEEE802 标准所提到的，它将这个包丢弃。一旦建立对话，大多数协议具有处理最大包长度的能力，因此每个站点可以处理包的分段，不影响网桥操作。

总长度和分段偏移字段使 IP 能重新构造数据报文并将它传递到高层软件。总长度指出原始包的总长度，偏移字段向正在组装包的节点指出该包偏离的开始端，此时数据处于分段，以重新构造包.

- 生存时间（TTL）（8 位）。

在包的传递过程中可能会出现错误情况，引起包在网际的路由器之间不断循环。为防止此类事件发生，因而引入了 TTL。由包的发源地设置生存事件的起始值。生存事件是一个由路由器使用的字段，确保包不会无限的循环。在发送站点设置这个字段，然后随着数据报文通过每个路由器而减一。当把这个字段的值减到 0 时，路由器将废弃这个包，并通知数据报文的发源地，它不能转送这个包

- 协议字段。

该字段用于指出哪个较高级协议发送了帧，哪个接受协议应得到这个帧。有许多协议可以处于 IP 的上面。就在 IP 上的协议而言，对 IP 并不是特定的。当前对通用的传输实现是 TCP 和 UDP。该字段的目的是使 IP 知道如何正确的将包传递到它上面的正确的机构，如果将协议字段设置成 TCP，则将包处理得适合 TCP 要求，以便进一步进行帧处理。UDP 的情况也一样。

- 校验和。

这是一个 16 位的循环冗余检验，目的是确保帧头的完整性。利用 IP 数据段中的数据产生 CRC（循环冗余检验）数，由发送站点放入这个字段。当接收站点读数据时，它将计算 CRC。如果两个 CRC 不匹配，则表示帧头有错位，将废弃包。随着数据报文被每个路由器接收，每个路由器将重新计算校验和，这是因为数据报文由所穿过的每个路由器改变 TTL。

- IP 选项字段。

选项类别确定数据是正常数据还是用作网络控制的数据。在选项类别内包含了多种选项编号。“0”的选项类别代表数据报文或网络控制包、类别“0”内的好书表示必须使用严格的源路由选择（如由源主机所规定的）。在那种情况下，包所经过的每个网关向包增加它的 IP 地址，以便识别。选项类别“2”中的编号 4 用于规定计时打印包在去目的地的途中所执行的所有暂停。通过记录平均时延和节点的处理时间可测量总体网络性能。

- IP 源和目的地地址字段。

这些字段指出包将被传递到的最终目的地 IP 地址和起始发送这个包的站点的 IP 地址（各 32 位）。这些地址将分辨 IP 网际上的所有主机。IP 地址是非常重要的，以下详细讨论。

3．IP 寻址

Internet 地址由位于斯坦福研究所（SRI）的 Internet Network Information Center（Internet NIC）Registration Service 发放。Inter NIC 可提供给你一个相当大的主要网络地址，以识别你的网络上的每个端点。

如果你有自己的网络，可以建立自己的网络信息中心，除非你计划连到 Internet 不需要和 Internet 的 NIC 接洽。但即使你不连接 Internet，NIC 仍将帮助你建立你自己的网络地址方案。如果有一天你希望同其它 TCP/IP 网络连接（例如同 Internet 连接）最好使用下面的约定。

你大概已经看到了 32 位 IP 地址。它使用了带点的十进制数，例如 128.101.4.9。它的二进制等效是：1000 0000 0110 0100 0000 0100 0000 1001。显然表示成点的十进制数方便得多。

Internet 地址可分成 5 类：

表 5-5 IP 地址分类

地址类型	第一个字节的十进制值
A	000-127
B	128-191
C	192-233
D	224-239
E	240-255

A 类网络地址有 128 个（支持 127）个网络，占有最左边的一个字节（8 位）。高位（0）表示识别这种地址的类型。因此这个位不能用作地址位，剩下右边的 31 位提供2^{31}个端点的寻址。这些大致中大约 1/3 已经被分配，想得到这类地址是很困难的。

B 类地址使用左边两个 8 位用来网络寻址。两个高位（10）用于识别这种地址的类型，其余的 14 位用作网络地址，右边的两个字节（16 位）用作网络节点，大约已经分配了 5000 个 B 类地址。

C 类地址是最常见的 Internet 地址。三个高位（110）用于地址类型识别，左边三个字节的其余 21 位用于寻址。C 类地址支持4×10^{6}个网络，每个网络可多达 256 端点，到目前为止已经使用了2×10^{6}个 C 类地址。

D 类地址是相当新的。它的识别头是 1110，用于组播，例如用于路由器修改。

E 类地址为时延保留，其识别头是 1111。

网络软件和路由器使用子网掩码来识别报文临时呆在网络内部，还是被路由选择到其它地方。在一个字段内“1”出现表明一个字段保换所有或部分网络地址。“0”表明主机地址为止。例如，最常用的 C 类地址使用前三个 8 位来识别网络，最后一个 8 位识别主机。因此子网掩码是 255.255.255.0。

此外还可以使用掩码建立子网，让我们来假定用户申请 B 类地址，Inter NIC 提供给用户 179.143.XXX.XXX。如果他 11 个 LAN，每个 LAN 有 89 个工作站，那么他可以把所有的工作站连入具有这种 B 类地址的 Internet。（Internet 地址正在变得缺少，按目前的使用增长率，到 2002 年将全部被使用，但目前正在考虑机种地址扩充方案）。

建立一个 255.255.255.0 的掩码在第三单元那对每个 LAN 分配一固定的数，比如 1-11，最后的单元用于每个 LAN 上的工作站（按 TCP/IP 的说法是主机）地址，因此你的地址将有形式：179.143.（1-11）.（1-254）。在最后的两个 8 位字组那怎样选择地址完全由你决定，不一定是顺序的。

IP 数据报文是无连接的，按 TCP/IP 的说法是不可靠的。但由于 TCP 协议提供了可靠的（确认的）面向连接的服务，因此不确认的传输是可接受的。

无连接服务的缺点是明显的，如果数据报文被破坏，或者由于缓冲器太小不能存放它们，将会引起数据报文被放弃。此外，链路也可能故障，主机和网关可能拥塞，以及由于不正确的实现建立了错误的路由或失效的帧头。

由于存在上述这些可能性，因此总是实现称为 Internet Control Message Protocal（ICMP）的第二协议。ICMP 可以告诉发源主机问题，并期待它解决。

4．网际控制报文协议（ICMP）

网络层上另一个重要协议是网际控制报文协议（ICMP）。IP 需要它帮助传输差错和控制报文。ICMP 报文在不同的 IP 模块间交换。一种报文是回应请求，用于测试目的地是否可达。此外回应请求报文还跟踪相应时间，以便确定线路的平均时延，进一步同应用程序

的时延阈进行比较。例如，如果线路时延太长，则基于主机的应用程序可以暂停。

当网关接收它们不能转送的包时，便发送一个不可到达目的地的报文，这类报文能指出网络或主机是不能到达的，或个别较高层协议或端口是不可到达的。如果源主机表示不需要分段包，网关还可以回送报文，指出若不分段数据就不能转送。如果源规定的路由故障，则可发送这类报文，说明目的地网络或主机是不可到达的。

如果由于某种原因目的地是不可到达的，则 IP 模块将通知 TCP 模块，然后 TCP 模块通知 Telnet 虚拟终端服务，最后 Telnet 将在屏幕上显示一个报文，通知用户这个报文是不可传递的。

另一类 ICMP 报文被称为“源断开”报文。它是一种拥塞控制方法。如果在 IP 网关接收了较多的包，超出控制能力，则可以靠 ICMP 进行摆脱，典型做法是经过缓冲器析溢出，然后网关发出一个“源断开”报文，以命令发送模块降低发送率。这些报文只是在短期生效。接收到“源断开”报文后，暂时降低了传输率。然后自动逐渐恢复一直到达原来的传输率为止。当接到新的“源断开”报文时，再次降低传输率。“源断开”报文增强了基于窗口的流量控制能力。

ICMP 报文的另一种类型仍旧是路由选择变化请求，它有几种不同的请求内容。例如，大，当网关得知所选择的网关不是去目的地的最佳网关时，便向源主机发送一个“重定向数据请求”。这些报文也用来规定重定向某些类型的服务。当网关接到一个标有生存期限的包时，便发送一个“长路由通知”。当 IP 模块遇到生存期满的包时，也通知源主机。

第四类 ICMP 报文用于计时打印请求和确认。这些报文用于估算网络上的平均往复时延，以便确定特定的程序所使用的最佳传输率和路由（传输设备）。为了估算这种往复时延使用了四种计时打印：

（1）发送方计时打印请求包。

（2）当接收方接收到包时打印包。

（3）当它发送回答报文时再次打印。

（4）当源发方接收到回答报文时打印回答报文。

计时打印（2）和（3）提供了处理这种信息需要多长时间的基本估算，计时打印（1）和（4）提供了包在网络上传输所需时间总量。反复计算多次打印可产生平均时延。

5.6.3 传输控制协议（TCP）

原始的 TCP/IP 主机是经过电话线连接的。70 年代早期通信工具和当今的通信设备截然不同，线路的噪声非常大，不能处理数据，因此 TCP 协议具有严格的内装差错检验算法确保数据的完整性。下面几段文章解释了 TCP 协议的结构严格性。

TCP 是重要的传输层协议，它和 UDP 不同，传输层软件 TCP 的目的是允许数据同网络上的另外站点进行可靠的交换。它能提供端口编号的译码，以识别主机的应用程序，而且完成数据的可靠传输。

为了说明 TCP 结构的严格性，让我们先来打个比方。假设你正在向某人讲述一个故事。如果那个人只是站在那里不应答你，你将不能辩明他是否了解你所讲述的内容。如果那个人用点头之类的方式应答，那么你将知道你所讲的话他是了解的，因此能继续同他交谈。与此相似，TCP 协议使用顺序编号和确认信息同网络上另外的站点交谈。使用顺序编号来确定包内数据的排序并发现故障的包。因为网际上不同的包不一定会以发送它们的顺序到达（例如，路由器废弃一系列传输包中的某一个），所以要对包中的数据进行排序，以确保与发送的顺序相同。此外，接收站点还可能接收两个同样的包，为了进行可靠类型的通信，使用带有确认信息的顺序编号，这种处理被称为全双工。连接的每一端都必须考虑到另一端的需要而维持它自己的顺序编号。

TCP 是面向字节的顺序协议，这意味着包内的每个字节被分配一个顺序编号，并分配给每包一个顺序编号。分配给包那每个字节的顺序编号可以合理的重复。

TCP 的顺序编号方法与它的产生年代有关，那时的通信手段落后，不是所有的网络都采用这种办法（使用一个独立的传输层软件）。例如：NetWare 是依靠网络层软件传输数据 NetWare 控制协议提供包的顺序编号。

为可靠的完成数据传输任务，TCP 将报文或数据分成可管理的长度并加上 TCP 头。图 5-3 表示一个 TCP 头。它描述了 TCP 头中一些主要的字段：

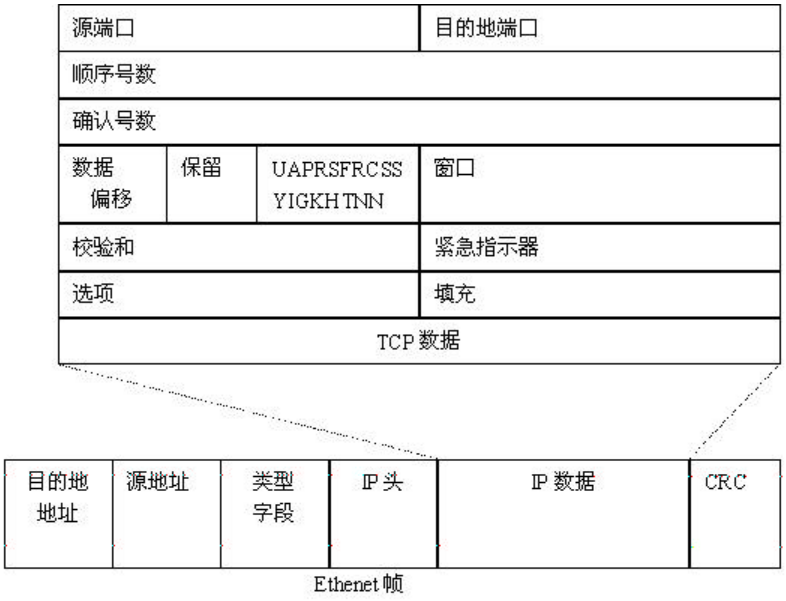

图 5-3 TCP 头信息

- 源端口（16 位）。源发站点的端口编号。
- 目的地端口（16 位）。接收站点的端口编号。
- 顺序号数（32 位）。分配给 TCP 包的编号。

除设置 SYN 位指出包的开始字节编号外，如果设置了这个位，顺序编号是最初的顺序号数（ISN），第一个数据字节是 ISN+1。

- 确认编号数（32 位）。

目的地站点向源站点发送的编号。对以前所接收的包（或许多包）表明确认。该序号指出目的地站点希望接收下一个顺序编号。一旦建立了连接就始终设置这个字段。

- 数据偏移（4 位）。

指出 TCP 头的长度（即 TCP 头中的 32 位字的数）。它表明数据开始和 TCP 头结束。对于正常的 20 字节的头，这个字段设置成 0101。

- 保留位（6 位）。

为未来使用而保留。必须设置为 0。

- 控制位（6 位）。

用作个别控制位，见表 5-6。

表 5-6 控制位的取值及其含义

URG	紧急指示字段
ACK	如果设置，该包包含确认。
PSH	推入功能
RST	恢复连接。用于这种情况：一个功能是不接收连接请求
SYN	用于建立序号（同步序号）。

FIN	数据不在从连接的发送点进入，结束总报文。

- 窗口（16 位）。

窗口字段也称接收窗口大小，表示在 TCP 连接上准备由主机接收的的 8 位字节的数目。

- CRC 校验和（16 位）。

一个差错检验数，用于确定被接收的数据报文在传输期间是否被讹误。包括 TCP 头和所有数据。

- 紧急指示字段（16 位）。

它指出了紧接紧急数据的字节的顺序编号。

- 选项。

长度变量，它考虑到 TCP 使用的各种选项：

（1）选项表的结束

（2）无操作

（3）最大分段长度

TCP 提供的主要服务有：

（1）建立、维持和终结两个进程之间的连接。

（2）可靠的包传递（经过确认过程）。

（3）编序包（可靠的数据传送）。

（4）控制差错的机制。

（5）通过使用端口，允许在个别的源和目的地主机内部实现和不同进程多重连接的能力。

（6）使用全双工操作的数据交换。

5.6.4 用户数据报文协议

UDP 也是 TCP/IP 的传输层协议，它是无连接的，不可靠的传输服务。当接收数据时它不向发送方提供确认信息，它不提供输入包的顺序，如果出现丢失包或重份包的情况，也不会向发送方发出差错报文。这一点很象 IP 协议。UDP 的主要作用是分配和管理端口编号，以正确无误的识别运行在网络站点上的个别应用程序。由于它执行功能时具有低的开销，因而执行速度比 TCP 快。它多半用于不需要可靠传输的应用程序，例如网络管理域，域名服务器等。

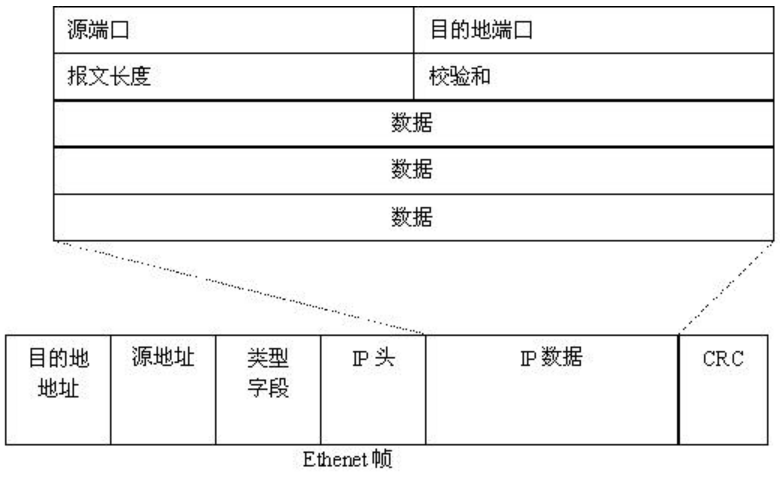

图 5-4 UDP 头

任何与 UDP 相配合作为传输层服务的应用程序必须提供确认和顺序系统，以确保包是以发送它们的同样顺序到达。也就是说，使用 UDP 的应用程序必须提供这类服务。（参见后面章节中关于 tftp 协议的介绍）

图 5-4 表示一个一个 UDP 头，应用数据被封装在 UDP 头那。传输层具有它自己的，与所有其它层不相关的帧头。然后 UDP 头及其数据被封装在 IP 头内，由 IP 协议将这个数据报文发送到数据链路层，依次下去，数据链路层又使用它的帧头包装这个报文，最后将数据送到物理层实际传输。

当接收包时，数据链路层将把地址解释为它自己的，剥去它的帧头，将包传递给 IP 层，IP 层将根据 IP 头上的正确 IP 地址接受包。剥去它的头，最后将包交给 UDP 软件，UDP 接受包，而且必须按 UDP 头上的端口编号进行译码。

5.7 小结

本章介绍了几种 LAN 协议。其中，着重讨论了 TCP/IP 协议。

TCP/IP 现在正在被越来越多的团体用户所接收。目前 TCP/IP 已经演变成最成熟的联网环境之一。事实上最近几年来 TCP/IP 和 UNIX 已经提供了多厂家产品互联性的模式，TCP/IP 的最大价值是它的不同平台之间提供互联性的能力。

在 OSI 真正履行自己的承诺以前，不太完善的通信协议结构，如 SNA、DECnet 和 TCP/IP 将继续流行。

第六章 Berkeley套接字

6.1 引言

网络程序设计全靠套接字接受和发送信息，尽管套接字这个词好象显得有些神秘，但其实这个概念极易理解。

这章主要讲述Sockets API（Application Program Interface），以及一些其他的细节（比如Socket的历史、数据中的常用结构等），通过这些介绍，使读者慢慢掌握Linux下的Socket编程。

6.2 概述

在开始介绍有关编程的知识之前，首先让我们来了解一些与socket有关的背景知识。

6.2.1 Socket的历史

在80年代早期，远景研究规划局（Advanced Research Projects Agency, ARPA）资助了佳利福尼亚大学伯克利分校的一个研究组，让他们将TCP/IP软件移植到UNIX操作系统中，并将结果提供给其他网点。作为项目的一部分，设计者们创建了一个接口，应用进程使用这个接口可以方便的进行通信。他们决定，只要有可能就使用以有的系统调用，对那些不能方便的容入已有的函数集的情况，就再增加新的系统调用以支持TCP/IP功能。

这样做的结果就出现了插口接口（Berkeley Socket），这个系统被称为Berkeley UNIX或BSD UNIX。(TCP/IP首次出现在BSD 4.1版本release 4.1 of Berkeley Software Distribution)。

由许多计算机厂商，都采用了Berkeley UNIX，于是许多机器上都可以使用Socket了。这样，Socket接口就被广泛使用，到现在已经成为事实上的标准。（图6-1）

6.2.2 Socket的功能

Socket的英文原意就是“孔”或“插座”，现在，作为BSD UNIX的进程通讯机制，取其后一种意义。日常生活中常见的插座，有的是信号插座，有的是电源插座，有的可以接受信号（或能量），有的可以发送信号（或能量）。假如电话线与电话机之间安放一个插座（相当于二者之间的接口，这一部分装置物理上是存在的）则Socket非常相似于电话插座。

将电话系统与面向连接的Socket机制相比，有着惊人相似的地方。以一个国家级的电话网为例。电话的通话双方相当于相互通信的两个进程；通话双方所在的地区（享有一个全局唯一的区号）相当于一个网络，区号是它的网络地址；区内的一个单位的交换机相当于一台主机，主机分配给每个用户的局内号码相当于Socket号（下面将谈到）。

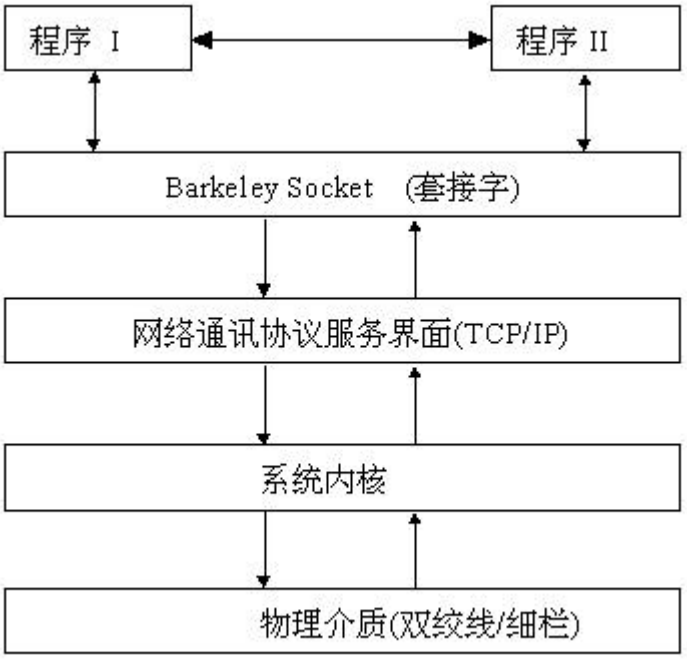

图 6-1　socket 接口示意图

任何用户在通话之前，首先要占有一部电话机，相当于申请一个 Socket 号；同时要知道对方的电话号码，相当于对方有一个 Socket。然后向对方拨号呼叫，相当于发出连接请求（假如对方不在同一区内，还要拨对方区号，相当于给出网络地址）。对方假如在场并空闲（相当于通信的另一主机开机且可以接受连接请求），拿起电话话筒，双方就可以正式通话，相当于连接成功。双方通话的过程，是向电话机发出信号和从电话机接受信号的过程，相当于向 Socket 发送数据和从 Socket 接受数据。通话结束后，一方挂起电话机，相当于关闭 Socket，撤消连接。

在电话系统中，一般用户只能感受到本地电话机和对方电话号码的存在，建立通话的过程、话音传输的过程以及整个电话系统的技术细节对它都是透明的，这也与 Socket 机制非常相似。Socket 利用网间网通信设施实现进程通信，但它对通信设施的细节毫不关心，只要通信设施能提供足够的通信能力，它就满足了。

至此，我们对 Socket 进行了直观的描述。抽象出来，Socket 实质上提供了进程通信的端点。进程通信之前，双方首先必须各自创建一个端点，否则是没有办法建立联系并相互通信的。正如打电话之前，双方必须各自拥有一台电话机一样。

每一个 Socket 都用一个半相关描述：

{协议，本地地址，本地端口}

一个完整的 Socket 则用一个相关描述

{协议，本地地址，本地端口，远程地址，远程端口}

每一个 Socket 有一个本地的唯一 Socket 号，由操作系统分配。

最重要的是，Socket 是面向客户－服务器模型而设计的，针对客户和服务器程序提供不同的 Socket 系统调用。客户随机申请一个 Socket 号（相当于一个想打电话的人可以在

任何一台入网的电话上拨叫呼叫）；服务器拥有全局公认的 Socket，任何客户都可以向它发出连接请求和信息请求（相当于一个被呼叫的电话拥有一个呼叫方知道的电话号码）。

Socket 利用客户—服务器模式巧妙的解决了进程之间建立通信连接的问题。服务器 Socket 为全局所公认非常重要。两个完全随机的用户进程之间，因为没有任何一方的 Socket 是固定的，就像打电话却不知道别人的电话号码，要通话是不可能的。

6.2.3 套接字的三种类型

套接字有三种类型：流式套接字（SOCK_STREAM），数据报套接字（SOCK_DGRAM）及原始套接字。

1．流式套接字（SOCK_STREAM）

流式的套接字可以提供可靠的、面向连接的通讯流。如果你通过流式套接字发送了顺序的数据：“1”、“２”。那么数据到达远程时候的顺序也是“１”、“２”。

流式套接字可以做什么呢？你听说过 Telnet 应用程序吗？听过？哦，最常用的 BBS 服务，以及系统的远程登陆都是通过 Telnet 协议连接的。Telnet 就是一个流式连接。你是否希望你在 Telnet 应用程序上输入的字符（或汉字）在到达远程应用程序的时候是以你输入的顺序到达的？答案应该是肯定的吧。还有 WWW 浏览器，它使用的 HTTP 协议也是通过流式套接字来获取网页的。事实上，如果你 Telnet 到一个 Web Site 的 80 端口上，然后输入 “GET 网页路径名”然后按两下回车（或者是两下 Ctrl+回车）然后你就得到了“网页路径名”所代表的网页！

流式套接字是怎样保证这种应用层次上的数据传输质量呢？它使用了 TCP（The Transmission Control Protocol）协议（可以参考 RFC-793 来得到 TCP 的细节）。TCP 保证了你的数据传输是正确的，并且是顺序的。TCP 是经常出现的 TCP/IP 中的前半部分。IP 代表 Internet Protocol（因特网协议，参考 RFC-791）IP 只处理网络路由。

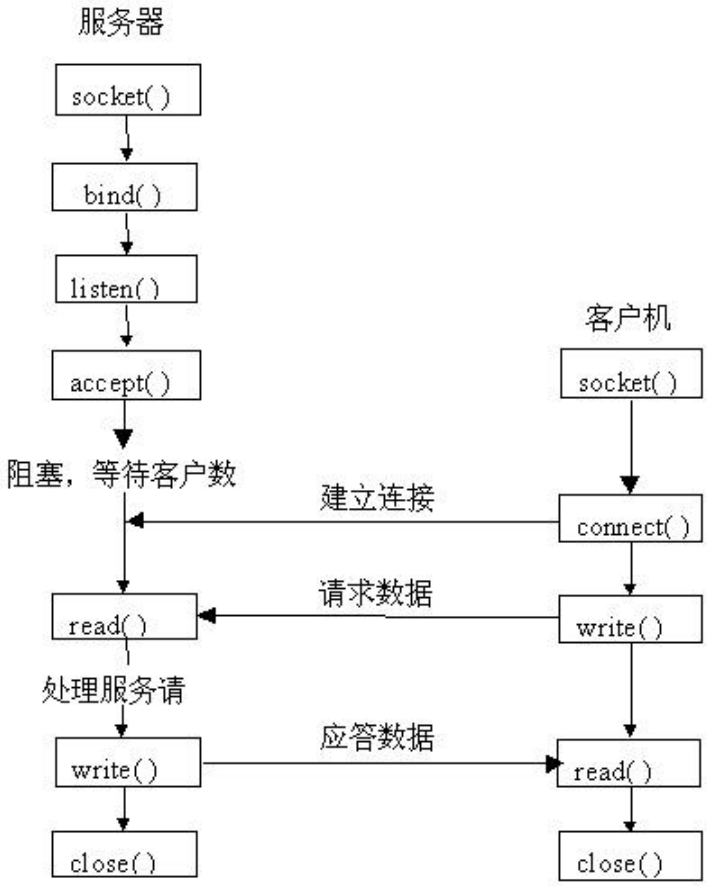

图 6-2　面向连接的 socket 的工作流程

2．数据报套接字（SOCK_DGRAM）

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠，无差错。原始套接字允许对低层协议如 IP 或 ICMP 直接访问，主要用于新的网络协议实现的测试等。

数据报套接字（Datagram Sockets**）怎样呢？为什么它叫做“无连接”？应该怎样处理它们呢？为什么它们是不可靠的？好的，这里有一些事实：**

- **如果你发送了一个数据报，它可能不会到达。**
- **它可能会以不同的顺序到达。**
- **如果它到达了，它包含的数据中可能存在错误。**

数据报套接字也使用 IP，但是它不使用 TCP，它使用使用者数据报协议 UDP（User Datagram Protocol **可以参考** RFC 768**）**

为什么说它们是“无连接”的呢？因为它（UDP**）不像流式套接字那样维护一个打开**

的连接，你只需要把数据打成一个包，把远程的 IP 贴上去，然后把这个包发送出去。这个过程是不需要建立连接的。UDP 的应用例子有： tftp, bootp 等。

那么，数据包既然会丢失，怎样能保证程序能够正常工作呢？事实上，每个使用 UDP 的程序都要有自己的对数据进行确认的协议。比如， TFTP 协议定义了对于每一个发送出去的数据包，远程在接受到之后都要回送一个数据包告诉本地程序：“我已经拿到了！”(一个 “ACK” 包)。如果数据包发的送者在 5 秒内没有的得到回应，它就会重新发送这个数据包直到数据包接受者回送了 “ACK” 信号。这些知识对编写一个使用 UDP 协议的程序员来说是非常必要的。

无连接服务器一般都是面向事务处理的，一个请求一个应答就完成了客户程序与服务程序之间的相互作用。若使用无连接的套接字编程，程序的流程可以用图 6-3 表示。

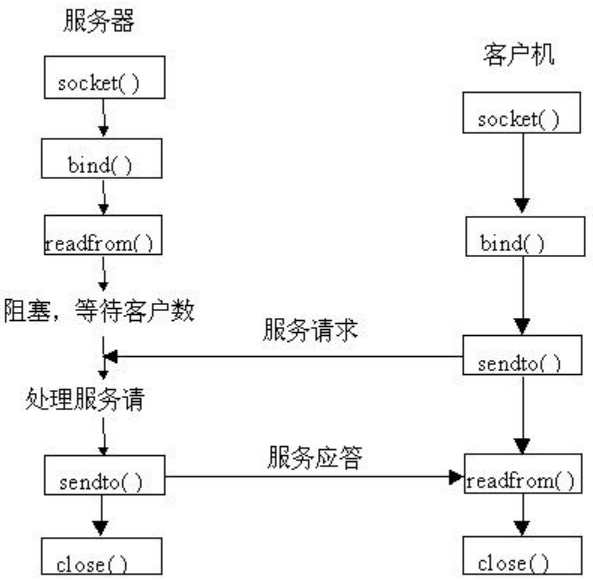

图 6-3 无连接的 socket 工作流程

面向连接服务器处理的请求往往比较复杂，不是一来一去的请求应答所能解决的，而且往往是并发服务器。使用面向连接的套接字编程,可以通过图 6-2 来表示。

套接字工作过程如下:服务器首先启动，通过调用 socket()建立一个套接字，然后调用 bind()将该套接字和本地网络地址联系在一起，再调用 listen()使套接字做好侦听的准备，并规定它的请求队列的长度,之后就调用 accept()来接收连接。客户在建立套接字后就可调用 connect()和服务器建立连接。连接一旦建立，客户机和服务器之间就可以通过调用 read()和 write()来发送和接收数据。最后，待数据传送结束后，双方调用 close()关闭套接字。

3．原始套接字

原始套接字主要用于一些协议的开发，可以进行比较底层的操作。它功能强大，但是没有上面介绍的两种套接字使用方便，一般的程序也涉及不到原始套接字。

6.3　Linux 支配的网络协议

网络协议是系统进行系统与系统间通讯的的接口。在 Linux 系统上，TCP/IP（Transmission Control / Internet Protocol）是最常见的。 TCP/IP 是一个网络协议协议族，我们将在下面进行详细介绍。

6.3.1　什么是 TCP/IP?

用简单的话来讲， TCP/IP 是一个网络协议族的名字，协议是所有软件产品必须遵守的、能够保证各种软件产品能够正确通讯的规则。协议还定义了每一部分数据块怎样管理所传输的数据。

精确一点说，一个协议定义了两个应用程序或是计算机之间能够进行互相通讯，对于其中的每一个（应用程序或计算机）都保证使用同样的标准。TCP/IP 代表传输控制协议/网络协议（注意：它们是两个不同的协议 !），它是做为软件的网络组成部件而设计的。每个 TCP/IP 的协议都有他专门的工作，比如万维网（WWW），发送电子邮件（E-mail），传输文件（Ftp），提供远程登陆服务等。

TCP/IP 协议可以根据提供的不同的服务分为几组：

1．控制数据的协议

TCP（传输控制协议 Transmission Control Protocol）以连接为基础，也就是说两台电脑必须先建立一个连接，然后才能传输数据。事实上，发送和接受的电脑必须一直互相通讯和联系。

UDP（使用者数据报协议 User Datagram Protocol）它是一个无连接服务，数据可以直接发送而不必在两台电脑之间建立一个网络连接。它和有连接的 TCP 相比，占用带宽少，但是你不知道你的数据是否真正到达了你的客户端，而客户端收到的数据也不知道是否还是原来的发送顺序。

2．数据路由协议

路由协议分析数据包的地址并且决定传输数据到目的电脑最佳路线。他们也可以把大的数据分成几部分，并且在目的地再把他们组合起来。

IP（因特网协议 Internet Protocol）处理实际上传输数据。

ICMP（因特网控制信息协议 Internet Control Message Protocol）处理 IP 的状态信息，比如能影响路由决策的数据错误或改变。

RIP（路由信息协议 Routing Information Protocol）它是几个决定信息传输的最佳路由路线协议中的一个。

OSPF（Open Shortest Path First）一个用来决定路由的协议。网络地址协议决定了命名电脑地址的方法：使用一个唯一的数字和一个字母名字。

ARP（地址决定协议 Address Resolution Protocol）确定网络上一台电脑的数字地址。

DNS（域名系统 Domain Name System）从机器的名字确定一个机器的数字地址。

RARP（反向地址决定协议 Reverse Address Resolution Protocol）确定网络上一台计算机的地址，和 ARP（地址决定协议 Address Resolution Protocol）正好相反。

3．用户服务

BOOTP（启动协议 Boot Protocol） 由网络服务器上取得启动信息，然后将本地的网

络计算机启动。

FTP（文件传输协议 File Transfer Protocol）通过国际互连网从一台计算机上传输一个或多个文件到另外一台计算机。

TELNET（远程登陆）允许一个远程登陆，使用者可以从网络上的一台机器通过 TELNET 连线到另一台机器，就像使用者直接在本地操作一样。

EGP（外部网关协议 Exterior Gateway Protocol）为外部网络传输路由信息。

GGP（网关到网关协议 Gateway-to-Gateway Protocol）在网关和网关之间传输路由协议。

IGP（内部网关协议 Interior Gateway Protocol）在内部网络传输路由信息。

3．其他协议(也为网络提供了重要的服务)

NFS（网络文件系统 Network File System）允许将一台机器的目录被另一台机器上的用户 安装（Mount）到自己的机器上，就像是对本地文件系统进行操作一样进行各式各样的操作。

NIS（网络信息服务 Network Information Service）对整个网络用户的用户名、密码进行统一管理，简化在 NIS 服务下整个网络登陆的用户名／密码检查。

RPC（远程过程调用 Remote Procedure Call）通过它可以允许远程的应用程序通过简单的、有效的手段联系本地的应用程序，反之也是。

SMTP（简单邮件传输协议 Simple Mail Transfer Protocol）一个专门为电子邮件在多台机器中传输的协议，平时发邮件的 SMTP 服务器提供的必然服务。

SNMP（简单网络管理协议 Simple Network Management Protocol）这是一项为超级用户准备的服务，超级用户可以通过它来进行简单的网络管理。

6.4 套接字地址

好了，关于 socket 的背景知识我们已经讲得够多了，下面，就让我们正式开始揭开 socket 的神秘面纱吧！

6.4.1 什么是 Socket？

大家经常谈论 “ Socket ”（套接字），那么一个套接字究竟是什么呢？

一个套接字可以这样来解释：它是通过标准的 UNIX 文件描述符和其他的程序通讯的一个方法。

6.4.2 Socket 描述符

使用 UNIX 的黑客高手有这么一句话：“ 恩，在 UNIX 系统中，任何东西都是一个文件。” 这句话描述了这样一个事实：在 UNIX 系统中，任何对 I/O 的操作，都是通过读或写一个文件描述符来实现的。

一个文件描述符只是一个简单的整形数值，代表一个被打开的文件（这里的文件是广义的文件，并不只代表不同的磁盘文件，它可以代表一个网络上的连接，一个先进先出队列，一个终端显示屏幕，以及其他的一切）。在 UNIX 系统中任何东西都是一个文件 !! 所以如果你想通过 Internet 和另外一个程序通讯的话，你将会是通过一个文件来描述符实现的。你最好相信这一点。

好的，你已经相信 Socket 是一个文件描述符了，那么我们应该怎样才能得到这个代表网络连接的文件描述符呢？你现在一定非常在意这个问题。是这样的：你首先调用系统函数 socket()，它返回一个套接字（Socket）描述符，然后你就可以通过对这个套接字描述符进行一些操作：系统函数 send() 和 recv()（你可以使用“man”命令来查找系统帮助：man send, man recv）。

你会想：“套接字描述符是一个文件描述符，为什么不能用对文件操作的 write() 和 read() 来进行套接字通讯呢？”事实上， write() 和 read() 是可以对套接字描述符进行操作的，但是，通过使用 send() 和 recv() 函数，你可以对网络数据的传输进行更好的控制！

6.4.3　一个套接字是怎样在网络上传输数据的？

我们已经谈过了网络协议层，那么我们还应该继续多了解一些东西：物理网络上的数据是怎样传送的。

我们可以认为是这样的：

数据被分成一个一个的包（Packet），包的数据头（或数据尾）被第一层协议（比如 TFTP 协议） 加上第一层协议数据；然后整个包（包括内部加入的 TFTP 信息头）被下层协议再次包装（比如 UDP），再这之后数据包会再次被下层协议包装（比如 IP 协议），最后是被最底层的硬件层（物理层）包装上最后一层信息（Ethernet 信息头）。

当接受端的计算机接收到这个包后，硬件首先剥去数据包中的 Ethernet 信息头，然后内核在剥去 IP 和 UDP 信息头，最后把数据包提交给 TFTP 应用程序，由 TFTP 剥去 TFTP 信息头，最后得到了原始数据。

下面我们再大致回顾一下著名的网络层次模型。

通过这个网络模型，你可以写套接字的应用程序而不必在乎事实上数据在物理层中的传输方法（无论是以太网，还是并口、AUI 或是其他的什么方法）。

因为已经有程序在底层为你处理了这些问题了。下面是 OSI 模型，你可以记住它来应付一些测验。

- 应用层
- 表示层
- 会话层
- 传输层
- 网络层
- 数据链路层
- 物理层

物理层就是硬件层（比如并口，以太网）。应用程序层离物理层很远很远，以至于它可以不受物理层的影响。

上面这个模型是最一般的模型，但是在 Linux 中，真正用到的模型是下面这样子的：

- 应用层（Telnet，Ftp，等等）
- 主机间对话层（TCP 和 UDP）
- 网络层（IP 和路由）
- 网络底层（相当于 OSI 模型中网络、数据链路和物理层）

现在，你大概已经明白各个协议层是怎样对原始数据进行包装和解包的了吧。看见对

于每一个数据包有多少项工作需要做了吗？对！你对每一个数据包都需要自己用“cat”命令来查看协议信息头！

开个玩笑。对流式套接字你所需要做的只是调用 send() 函数来发送数据。而对于数据报套接字，你需要自己加个信息头，然后调用 sendto() 函数把数据发送出去。Linux 系统内核中已经建立了 Transport Layer 和 Internet Layer。硬件负责 NetworkAccess Layer。简单而有效，不是吗？

6.5 套接字的一些基本知识

好的，从现在开始，我们应该谈些和程序有关的事情了。

6.5.1 基本结构

首先，我想介绍一些使用套接字编程中常见的网络数据结构对大家会很有帮助。

1．struct sockaddr

这个结构用来存储套接字地址。

数据定义：

```
struct sockaddr {
unsigned short sa_family;          /* address族, AF_xxx */
char sa_data[14];                  /* 14 bytes的协议地址 */
};
```

sa_family 一般来说，都是“AFINET”。

sa_data 包含了一些远程电脑的地址、端口和套接字的数目，它里面的数据是杂溶在一切的。

为了处理 struct sockaddr， 程序员建立了另外一个相似的结构 struct sockaddr_in：

```
struct sockaddr_in (“in” 代表 “Internet”)
struct sockaddr_in {
    short int sin_family;              /* Internet地址族 */
    unsigned short int sin_port;       /* 端口号 */
    struct in_addr sin_addr;           /* Internet地址 */
    unsigned char sin_zero[8];         /* 添0（和struct sockaddr一样大小）*/
};
```

这个结构提供了方便的手段来访问 socket address（struct sockaddr）结构中的每一个元素。注意 sin_zero[8] 是为了是两个结构在内存中具有相同的尺寸，使用 sockaddr_in 的时候要把 sin_zero 全部设成零值（使用 bzero()或 memset()函数）。而且，有一点很重要，就是一个指向 struct sockaddr_in 的指针可以声明指向一个 sturct sockaddr 的结构。所以虽然 socket() 函数需要一个 structaddr * ，你也可以给他一个 sockaddr_in * 。注意在 struct sockaddr_in 中，sin_family 相当于 在 struct sockaddr 中的 sa_family，需要设成“AF_INET”。最后一定要保证 sin_port 和 sin_addr 必须是网络字节顺序（见下节）！

2．struct in_addr

其定义如下：

```
/* 因特网地址 (a structure for historical reasons) */
```

```
struct in_addr {
        unsigned long s_addr;
};
```

如果你声明了一个“ina”作为一个 struct sockaddr_in 的结构，那么“ina.sin_addr.s_addr”就是 4 个字节的 IP 地址（按网络字节顺序排放）。需要注意的是，即使你的系统仍然使用联合而不是结构来表示 struct in_addr，你仍然可以用上面的方法得到 4 个字节的 IP 地址（一些 #defines 帮了你的忙）。

6.5.2　基本转换函数

在前面提到了网络字节顺序。那么什么是网络字节顺序，它有什么特殊性，又如何将我们通常使用的数据转换成这种格式呢？

1．网络字节顺序

因为每一个机器内部对变量的字节存储顺序不同（有的系统是高位在前，底位在后，而有的系统是底位在前，高位在后），而网络传输的数据大家是一定要统一顺序的。所以对与内部字节表示顺序和网络字节顺序不同的机器，就一定要对数据进行转换（比如 IP 地址的表示，端口号的表示）。但是内部字节顺序和网络字节顺序相同的机器该怎么办呢？是这样的：它们也要调用转换函数，但是真正转换还是不转换是由系统函数自己来决定的。

2．有关的转化函数

我们通常使用的有两种数据类型：短型（两个字节）和长型（四个字节）。下面介绍的这些转换函数对于这两类的无符号整型变量都可以进行正确的转换。

如果你想将一个短型数据从主机字节顺序转换到网络字节顺序的话，有这样一个函数：它是以“h”开头的（代表“主机”）；紧跟着它的是“to”，代表“转换到”；然后是“n”代表“网络”；最后是“s”，代表“短型数据”。H-to-n-s，就是 htons() 函数（可以使用 Host to Network Short 来助记）

很简单吧……我没有理解的时候觉得这个函数不好记呢……

你可以使用“n”,“h”,“to”,“s”,“l”的任意组合……当然，你要在可能的情况下进行组合。比如，系统是没有 stolh() 函数的（Short to Long Host?）。

下面给出套接字字节转换程序的列表：

- htons()——“Host to Network Short” 主机字节顺序转换为网络字节顺序（对无符号短型进行操作 4 bytes）
- htonl()——“Host to Network Long” 主机字节顺序转换为网络字节顺序（对无符号长型进行操作 8 bytes）
- ntohs()——“Network to Host Short “ 网络字节顺序转换为主机字节顺序（对无符号短型进行操作 4 bytes）
- ntohl()——“Network to Host Long “ 网络字节顺序转换为主机字节顺序（对无符号长型进行操作 8 bytes）

注意：现在你可能认为自己已经精通于这几个函数的用处了……你可能会想：“恩……在我的 68000 机器内部，字节的表示顺序已经是网络字节顺序了，那么我的程序里就不必调用 htonl() 来转换我的 IP 地址了”。是的，你可能是对的。但是假如你把你的程序移植到一个内部字节顺序和网络字节顺序相反的机器上，你的程序就会运行不正常！所以，一定要记住：在你把数据发送到 Internet 之前，一定要把它的字

节顺序从主机字节顺序转换到网络字节顺序！

在 struct sockaddr_in 中的 sin_addr 和 sin_port 他们的字节顺序都是网络字节顺序，而 sin_family 却不是网络字节顺序的。为什么呢？

这个是因为 sin_addr 和 sin_port 是从 IP 和 UDP 协议层取出来的数据，而在 IP 和 UDP 协议层，是直接和网络相关的，所以，它们必须使用网络字节顺序。然而， sin_family 域只是内核用来判断 struct sockaddr_in 是存储的什么类型的数据，并且， sin_family 永远也不会被发送到网络上，所以可以使用主机字节顺序来存储。

3．IP 地址转换

很幸运， Linux 系统提供和很多用于转换 IP 地址的函数，使你不必自己再写出一段费力不讨好的子程序来吃力的变换 IP。

首先，让我假设你有一个 struct sockaddr_in ina，并且你的 IP 是 166.111.69.52 ，你想把你的 IP 存储到 ina 中。你可以使用的函数： inet_addr() ，它能够把一个用数字和点表示 IP 地址的字符串转换成一个无符号长整型。你可以像下面这样使用它：

ina.sin_addr.s_addr = inet_addr（“166.111.69.52”）;

注意：

- inet_addr() 返回的地址已经是网络字节顺序了，你没有必要再去调用 htonl() 函数，是不是很方便呢？
- 上面的用法并不是一个很好的习惯，因为上面的代码没有进行错误检查。如果 inet_addr() 函数执行错误，它将会返回 –1……等等！二进制的无符号整数值 –1 相当于什么？相当于 255.255.255.255 !! 一个广播用的 IP 地址！没有办法，你只能在你自己的程序里进行对症下药的错误检查了。

好，现在我们已经可以把字符串的 IP 地址转换成长整型了。那么还有没有其他的方法呢？如果你有一个 struct in_addr 并且你想把它代表的 IP 地址打印出来（按照 数字.数字.数字.数字的格式）……

这里，你可以使用函数 inet_ntoa()（“ ntoa ”代表“ Network to ASCII ”）：

printf（“%s”, inet_ntoa(ina.sin_addr)）;

这段代码将会把 struct in_addr 里面存储的网络地址以 数字.数字.数字.数字 的格式显示出来。

注意：

- inet_ntoa() 使用 struct in_addr 作为一个参数，不是一个长整型值。
- inet_ntoa() 返回一个字符指针，它指向一个定义在函数 inet_ntoa() 中的 static 类型字符串。所以每次你调用 inet_ntoa()，都会改变最后一次调用 inet_ntoa() 函数时得到的结果。

比如：

```
char *a1, a2;
a1 = inet_ntoa(ina1.sin_addr); /* this is 166.111.69.52 */
a2 = inet_ntoa(ina2.sin_addr); /* this is 166.111.69.53 */
printf(“ address 1: %s\n”,a1);
printf(“ address 2: %s\n”,a2);
```

将会显示出：

```
address 1: 166.111.69.53
address 2: 166.111.69.53
```

如果你想把结果保存下来，那么你可以在每次调用 inet_ntoa() 后调用 strcpy() 将结果存到另外一个你自己的字符串中。

在后面，将会介绍怎样把域名转换为 IP。

6.6　基本套接字调用

Linux 支持伯克利（BSD）风格的套接字编程．它同时支持面向连接和不连接类型的套接字。

在面向连接的通讯中服务器和客户机在交换数据之前先要建立一个连接．再不连接通讯中数据被作为信息的一部分被交换．无论那一种方式，服务器总是最先启动，把自己绑定（Banding）在一个套接字上，然后侦听信息．服务器究竟怎样试图去侦听就得依靠你编程所设定的连接的类型了。

你需要了解的一些系统调用：

- socket()
- bind()
- connect()
- listen()
- accept()
- send()
- recv()
- sendto()
- recvfrom()
- close()
- shutdown()
- setsockopt()
- getsockopt()
- getpeername()
- getsockname()
- gethostbyname()
- gethostbyaddr()
- getprotobyname()
- fcntl()

我们将在以下详细介绍这些系统调用。

6.6.1　socket() 函数

取得套接字描述符！（记得我们以前说过的吗？它其实就是一个文件描述符）

socket 函数的定义是下面这样子的：

```
#include <sys/types.h>
#include <sys/socket.h>

int socket（int domain , int type , int protocol）;
```

你是否对 int domain 和 int type、int protocol 有些疑惑呢？调用 socket()的参数是什么呢？

首先，domain 需要被设置为 “ AF_INET ”，就像上面的 struct sockaddr_in。然后，type 参数告诉内核这个 socket 是什么类型，“ SOCK_STREAM ”或是“ SOCK_DGRAM ”。最后，只需要把 protocol 设置为 0 。

注意：事实上，domain 参数可以取除了“ AF_INET ”外的很多值，types 参数也可以取除了“ SOCK_STREAM ”或“ SOCK_DGRAM ”的另外类型。具体可以参考 socket 的 man pages（帮助页）。

套接字创建时没有指定名字．客户机用套接字的名字读写它。这就是下面的绑定函数所要做之事．

socket()函数只是简单的返回一个你以后可以使用的套接字描述符。如果发生错误，socket()函数返回 –1 。全局变量 errno 将被设置为错误代码。（可以参考 perror() 的 man pages）

6.6.2 bind() 函数

bind()函数可以帮助你指定一个套接字使用的端口。

当你使用 socket() 函数得到一个套接字描述符，你也许需要将 socket 绑定上一个你的机器上的端口。

- 当你需要进行端口监听 listen()操作，等待接受一个连入请求的时候，一般都需要经过这一步。比如网络泥巴（ＭＵＤ），Telnet a.b.c.d 4000。
- 如果你只是想进行连接一台服务器，也就是进行 connect() 操作的时候，这一步并不是必须的。

bind()的系统调用声明如下：

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (int sockfd , struct sockaddr *my_addr , int addrlen) ;
```

参数说明：

- sockfd 是由 socket()函数返回的套接字描述符。
- my_addr 是一个指向 struct sockaddr 的指针，包含有关你的地址的信息：名称、端口和 IP 地址。
- addrlen 可以设置为 sizeof(struct sockaddr)。

好，下面我们看一段程序：

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

#define MYPORT 4000
main()
{
    int sockfd ;
```

```
struct sockaddr_in my_addr ;

sockfd = socket(AF_INET, SOCK_STREAM, 0);        /* 在你自己的程序中 */
/* 要进行错误检查！! */
my_addr.sin_family = AF_INET ;                   /* 主机字节顺序 */
my_addr.sin_port = htons （MYPORT）;             /* 网络字节顺序，短整型 */
my_addr.sin_addr.s_addr = inet_addr(“166.111.69.52”) ;
bzero(&(my_addr.sin_zero), 8);                   /* 将整个结构剩余*/
/* 部分数据设为 0 */
/* 不要忘记在你自己的程序中加入判断 bind 错误的代码 !! */
bind (sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)）;
    ……
    ……
```

这里有一些值得注意的代码段：

- my_addr.sin_port **是网络字节顺序。**
- my_addr.sin_addr.s_addr **也是网络字节顺序。**
- **代码段包含的头文件，在不同的系统中可能有一点小小的区别。（不过在** Linux **中是如此）如果并非如此，你可以查一查** man pages **来获取帮助。**

最后，bind()**可以在程序中自动获取你自己的** IP **地址和端口。**

代码如下：

```
my_addr.sin_port = 0 ;                           /* 随机选择一个端口 */
my_addr.sin_addr.s_addr = INADDR_ANY ;           /* 使用自己的地址 */
```

如上，通过设置 my_addr.sin_port **为** 0**，**bind()**可以知道你要它帮你选择合适的端口；通过设置** my_addr.sin_addr.s_addr **为** INADDR_ANY**，**bind()**知道你要它将** s_addr **填充为运行这个进程的机器的** IP**。这一切都可以要求** bind()**来自动的帮助你完成。**

如果你注意到了一些细节的话，你可能会发现我并没有将 INADDR_ANY **转换为网络字节顺序！是这样的，**INADDR_ANY **的值为** 0**，**0 **就是** 0**，无论用什么顺序排列位的顺序，它都是不变的。**

有读者会想了，因为我用的 INADDR_ANY **是一个**#define**，那么如果将我的程序移植到另外一个系统，假如那里的** INADDR_ANY **是这样定义的：**#define INADDR_ANY 100**，那么我的程序不是就会不运行了吗？那么下面这段代码就** OK **了**

```
my_addr.sin_port = htons(0);                     /* 随机选择一个未用的端口 */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY) ;    /* 使用自己的IP地址 */
```

现在我们已经是这么的严谨，对于任何数值的 INADDR_ANY **调用** bind **的时候就都不会有麻烦了。**

当 bind()**函数调用错误的时候，它也是返回**–1 **作为错误发生的标志。**errn **的值为错误代码。**

另外一件必须指出的事情是：当你调用 bind()**的时候，不要把端口数设置的过小！小于** 1024 **的所有端口都是保留下来作为系统使用端口的，没有** root **权利无法使用。你可以使用** 1024 **以上的任何端口，一直到** 65535 **：你所可能使用的最大的端口号（当然，你还**

要保证你所希望使用的端口没有被其他程序所使用）。

最后注意有关 bind()的是：有时候你并不一定要调用 bind()来建立网络连接。比如你只是想连接到一个远程主机上面进行通讯，你并不在乎你究竟是用的自己机器上的哪个端口进行通讯（比如 Telnet），那么你可以简单的直接调用 connect()函数，connect()将自动寻找出本地机器上的一个未使用的端口，然后调用 bind()来将其 socket 绑定到那个端口上。

6.6.3 connect()函数

让我们花一点时间来假设你是一个 Telnet 应用程序。你的使用者命令你建立一个套接字描述符。你遵从命令，调用了 socket()。然后，使用者告诉你连接到 “166.111.69.52”的 23 端口（标准的 Telnet 端口）……你应该怎么做呢？

你很幸运：Telnet 应用程序，你现在正在阅读的就是套接字的进行网络连接部分：connect()。

connect() 函数的定义是这样的：

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);
```

connect()的三个参数意义如下：

- sockfd ：套接字文件描述符，由 socket()函数返回的。
- serv_addr 是一个存储远程计算机的 IP 地址和端口信息的结构。
- addrlen 应该是 sizeof(struct sockaddr)。

下面让我们来看看下面的程序片段：

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

#define DEST_IP “166.111.69.52”
#define DEST_PORT 23

main()
{
    int sockfd ;
    /* 将用来存储远程信息 */
    struct sockaddr_in dest_addr ;
    /* 注意在你自己的程序中进行错误检查!! */
    sockfd = socket（AF_INET, SOCK_STREAM, 0）;
    /* 主机字节顺序 */
    dest_addr.sin_family = AF_INET ;
    /* 网络字节顺序，短整型 */
    dest_addr.sin_port = htons （DEST_PORT）;
```

```
dest_addr.sin_addr.s_addr = inet_addr（DEST_IP）;
/* 将剩下的结构中的空间置 0 */
bzero(&(dest_addr.sin_zero), 8）;

/* 不要忘记在你的代码中对 connect()进行错误检查!! */
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    ……
    ……
```

再次强调，一定要检测 connect()的返回值：如果发生了错误（比如无法连接到远程主机，或是远程主机的指定端口无法进行连接等）它将会返回错误值 –1 。 全局变量 errno 将会存储错误代码。

另外，注意我们没有调用 bind()函数。基本上，我们并不在乎我们本地用什么端口来通讯，是不是？我们在乎的是我们连到哪台主机上的哪个端口上。Linux 内核自动为我们选择了一个没有被使用的本地端口。

在面向连接的协议的程序中,服务器执行以下函数：

- 调用 socket()函数创建一个套接字。
- 调用 bind()函数把自己绑定在一个地址上。
- 调用 listen()函数侦听连接。
- 调用 accept()函数接受所有引入的请求。
- 调用 recv()函数获取引入的信息然后调用 send()回答。

6.6.4 listen() 函数

listen()函数是等待别人连接，进行系统侦听请求的函数。当有人连接你的时候，你有两步需要做：通过 listen()函数等待连接请求，然后使用 accept()函数来处理。（accept()函数在下面介绍）。

listen()函数调用是非常简单的。函数声明如下：

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

listen()函数的参数意义如下：

- sockfd 是一个套接字描述符，由 socket()系统调用获得。
- backlog 是未经过处理的连接请求队列可以容纳的最大数目。

backlog 具体一些是什么意思呢？每一个连入请求都要进入一个连入请求队列，等待 listen 的程序调用 accept()（accept()函数下面有介绍）函数来接受这个连接。当系统还没有调用 accept()函数的时候，如果有很多连接，那么本地能够等待的最大数目就是 backlog 的数值。你可以将其设成 5 到 10 之间的数值（推荐）。

像上面的所有函数一样， listen()如果返回 –1 ，那么说明在 listen()的执行过程中发生了错误。全局变量 errno 中存储了错误代码。

那么我们需要指定本地端口了，因为我们是等待别人的连接。所以，在 listen()函数调用之前，我们需要使用 bind() 函数来指定使用本地的哪一个端口数值。

如果你想在一个端口上接受外来的连接请求的话，那么函数的调用顺序为：

```
socket() ;
bind() ;
listen() ;
/* 在这里调用 accept()函数 */
    ……
```

下面将不给出例程，因为 listen()是非常容易理解的。下面的 accept()函数说明中的例程中，有 listen()的使用。

6.6.5 accept()函数

函数 accept()有一些难懂。当调用它的时候，大致过程是下面这样的：

- 有人从很远很远的地方尝试调用 connect()来连接你的机器上的某个端口（当然是你已经在 listen()的）。
- 他的连接将被 listen 加入等待队列等待 accept()函数的调用（加入等待队列的最多数目由调用 listen()函数的第二个参数 backlog 来决定）。
- 你调用 accept()函数，告诉他你准备连接。
- accept()函数将回返回一个新的套接字描述符，这个描述符就代表了这个连接！

好，这时候你有了两个套接字描述符，返回给你的那个就是和远程计算机的连接，而第一个套接字描述符仍然在你的机器上原来的那个端口上 listen()。

这时候你所得到的那个新的套接字描述符就可以进行 send()操作和 recv()操作了。

下面是 accept()函数的声明：

```
#include <sys/socket.h>

int accept(int sockfd, void *addr, int *addrlen);
```

accept()函数的参数意义如下：

- sockfd 是正在 listen() 的一个套接字描述符。
- addr 一般是一个指向 struct sockaddr_in 结构的指针；里面存储着远程连接过来的计算机的信息（比如远程计算机的 IP 地址和端口）。
- addrlen 是一个本地的整型数值，在它的地址传给 accept() 前它的值应该是 sizeof(struct sockaddr_in)；accept()不会在 addr 中存储多余 addrlen bytes 大小的数据。如果 accept()函数在 addr 中存储的数据量不足 addrlen，则 accept()函数会改变 addrlen 的值来反应这个情况。

读者现在应该想到：如果调用 accept()失败的话，accept()函数会返回 –1 来表明调用失败，同时全局变量 errno 将会存储错误代码。

下面我们来看一段程序片段：

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

/* 用户连接的端口号 */
```

```
#define MYPORT 4000

/*  等待队列中可以存储多少个未经过 accept()处理的连接  */
#define BACKLOG 10
main()
{
        /*  用来监听网络连接的套接字 sock_fd，用户连入的套接字使用 new_fd */
    int sockfd, new_fd ;
    /*  本地的地址信息  */
    struct sockaddr_in my_addr ;

    /*  连接者的地址信息  */
    struct sockaddr_in their_addr ;

    int sin_size;
    /*  记得在自己的程序中这部分要进行错误检查！  */
    sockfd = socket(AF_INET, SOCK_STREAM, 0) ;
    /*  主机字节顺序  */
    my_addr.sin_family = AF_INET ;

    /*  网络字节顺序，短整型  */
    my_addr.sin_port = htons(MYPORT) ;

    /*  自动赋值为自己的 IP */
    my_addr.sin_addr.s_addr = INADDR_ANY ;

    /*  将结构中未使用部分全部清零  */
    bzero(&(my_addr.sin_zero), 8) ;

    /*  不要忘记在你自己的程序中下面的程序调用需要进行错误检测！！*/
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    listen(sockfd, BACKLOG);
    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, &their_addr, &sin_size);
            ……
            ……
```

注意：我们使用了套接字描述符 new_fd 用来进行所有的 send() 和 recv()调用。如果你只想获得一个单独的连接，那么你可以将原来的 sock_fd 关掉（调用 close()），这样的话就可以阻止以后的连接了。

在面向连接的通信中客户机要做如下一些事：

- 调用 socket()函数创建一个套接字。
- 调用 connect()函数试图连接服务。
- 如果连接成功调用 write()函数请求数据，调用 read()函数接收引入的应答。

6.6.6 send()、recv()函数

这两个函数是最基本的，通过连接的套接字流进行通讯的函数。

如果你想使用无连接的使用者数据报的话，请参考下面的 sendto() 和 recvfrom() 函数。

send() 函数的声明：

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sockfd, const void *msg, int len, int flags);
```

send 的参数含义如下：

- sockfd 是代表你与远程程序连接的套接字描述符。
- msg 是一个指针，指向你想发送的信息的地址。
- len 是你想发送信息的长度。
- flags 发送标记。一般都设为 0（你可以查看 send 的 man pages 来获得其他的参数值并且明白各个参数所代表的含义）。

下面看看有关 send()函数的代码片段：

```
char *msg = “Hello! World!”;
int len, bytes_sent;
    ……
    ……
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
    ……
    ……
    ……
```

send()函数在调用后会返回它真正发送数据的长度。

注意：send() 所发送的数据可能少于你给它的参数所指定的长度！

因为如果你给 send()的参数中包含的数据的长度远远大于 send()所能一次发送的数据，则 send()函数只发送它所能发送的最大数据长度，然后它相信你会把剩下的数据再次调用它来进行第二次发送。

所以，记住如果 send()函数的返回值小于 len 的话，则你需要再次发送剩下的数据。幸运的是，如果包足够小（小于 1K），那么 send()一般都会一次发送光的。

像上面的函数一样，send()函数如果发生错误，则返回 –1 ，错误代码存储在全局变量 errno 中。

下面我们来看看 recv()函数。

函数 recv()调用在许多方面都和 send()很相似，下面是 recv()函数的声明：

```
#include <sys/types.h>
#include <sys/socket.h>
```

int recv(int sockfd, void *buf, int len, unsigned int flags）;

recv()的参数含义如下：

- sockfd 是你要读取数据的套接字描述符。
- buf 是一个指针，指向你能存储数据的内存缓存区域。
- len 是缓存区的最大尺寸。
- flags 是 recv() 函数的一个标志，一般都为 0（具体的其他数值和含义请参考 recv() 的 man pages）。

recv() 返回它所真正收到的数据的长度。（也就是存到 buf 中数据的长度）。如果返回 –1 则代表发生了错误（比如网络以外中断、对方关闭了套接字连接等），全局变量 errno 里面存储了错误代码。

很简单，不是吗？现在你已经可以使用套接字连接进行网络发送数据和接受数据了！Ya! 你现在已经成为了一个 Linux 下的网络程序员了！

6.6.7 sendto() 和 recvfrom() 函数

这两个函数是进行无连接的 UDP 通讯时使用的。使用这两个函数，则数据会在没有建立过任何连接的网络上传输。因为数据报套接字无法对远程主机进行连接，想想我们在发送数据前需要知道些什么呢？

对了！是远程主机的 IP 地址和端口！

下面是 sendto()函数和 recvfrom()函数的声明：

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto（int sockfd, const void *msg, int len, unsigned int flags,
const struct sockaddr *to, int tolen）;
```

和你所看到的一样，这个函数和 send()函数基本一致。

- sockfd 是代表你与远程程序连接的套接字描述符。
- msg 是一个指针，指向你想发送的信息的地址。
- len 是你想发送信息的长度。
- flags 发送标记。一般都设为 0。（你可以查看 send 的 man pages 来获得其他的参数值并且明白各个参数所代表的含义）
- to 是一个指向 struct sockaddr 结构的指针，里面包含了远程主机的 IP 地址和端口数据。
- tolen 只是指出了 struct sockaddr 在内存中的大小 sizeof(struct sockaddr)。

和 send()一样，sendto()返回它所真正发送的字节数（当然也和 send()一样，它所真正发送的字节数可能小于你所给它的数据的字节数）。 当它发生错误的时候，也是返回 –1 ，同时全局变量 errno 存储了错误代码。

同样的，recv()函数和 recvfrom()函数也基本一致。

recvfrom()的声明为：

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

int recvfrom(int sockfd, void *buf, int len, unsigned int flags
struct sockaddr *from, int *fromlen);
```

其参数含义如下：

- sockfd 是你要读取数据的套接字描述符。
- buf 是一个指针，指向你能存储数据的内存缓存区域。
- len 是缓存区的最大尺寸。
- flags 是 recv() 函数的一个标志，一般都为 0（具体的其他数值和含义请参考 recv() 的 man pages）。
- from 是一个本地指针，指向一个 struct sockaddr 的结构（里面存有源 IP 地址和端口数）.
- fromlen 是一个指向一个 int 型数据的指针，它的大小应该是 sizeof（struct sockaddr）. 当函数返回的时候，formlen 指向的数据是 form 指向的 struct sockaddr 的实际大小.

recvfrom() 返回它接收到的字节数，如果发生了错误，它就返回–1 ，全局变量 errno 存储了错误代码.

如果一个信息大得缓冲区都放不下，那么附加信息将被砍掉。该调用可以立即返回,也可以永久的等待。这取决于你把 flags 设置成什么类型。你甚至可以设置超时(timeout)值。在说明书(man pages)中可以找到 recvfrom 的更多信息。

注意：如果你使用 cnnect()连接到了一个数据报套接字的服务器程序上，那么你就可以使用 send() 和 recv() 函数来传输你的数据. 不要以为你在使用一个流式的套接字，你所使用的仍然是一个使用者数据报的套接字，只不过套接字界面在 send() 和 recv()的时候自动帮助你加上了目标地址，目标端口的信息.

6.6.8 close()和 shutdown()函数

程序进行网络传输完毕后，你需要关闭这个套接字描述符所表示的连接。实现这个非常简单，只需要使用标准的关闭文件的函数：close()。

使用方法：

```
close(sockfd);
```

执行 close()之后，套接字将不会在允许进行读操作和写操作。任何有关对套接字描述符进行读和写的操作都会接收到一个错误。

如果你想对网络套接字的关闭进行进一步的操作的话，你可以使用函数 shutdown()。它允许你进行单向的关闭操作，或是全部禁止掉。

shutdown()的声明为：

```
#include <sys/socket.h>
int shutdown（int sockfd, int how）;
```

它的参数含义如下：

- sockfd 是一个你所想关闭的套接字描述符.
- how 可以取下面的值。0 表示不允许以后数据的接收操；1 表示不允许以后数据

的发送操作；2 表示和 close()一样，不允许以后的任何操作（包括接收，发送数据）

shutdown() 如果执行成功将返回 0，如果在调用过程中发生了错误，它将返回–1，全局变量 errno 中存储了错误代码．

如果你在一个未连接的数据报套接字上使用 shutdown()函数（还记得可以对数据报套接字 UDP 进行 connect()操作吗？），它将什么也不做．

6.6.9 setsockopt() 和 getsockopt() 函数

Linux 所提供的 socket 库含有一个错误（bug）。此错误表现为你不能为一个套接字重新启用同一个端口号，即使在你正常关闭该套接字以后。例如，比方说，你编写一个服务器在一个套接字上等待的程序．服务器打开套接字并在其上侦听是没有问题的。无论如何，总有一些原因（不管是正常还是非正常的结束程序）使你的程序需要重新启动。然而重启动后你就不能把它绑定在原来那个端口上了。从 bind()系统调用返回的错误代码总是报告说你试图连接的端口已经被别的进程所绑定。

问题就是 Linux 内核在一个绑定套接字的进程结束后从不把端口标记为未用。在大多数 Linux/UNIX 系统中，端口可以被一个进程重复使用，甚至可以被其它进程使用。

在 Linux 中绕开这个问题的办法是，当套接字已经打开但尚未有连接的时候用 setsockopt()系统调用在其上设定选项（options）。setsockopt()调用设置选项而 getsockopt()从给定的套接字取得选项。

这里是这些调用的语法：

```
#include<sys/types.h>
#include<sys/socket.h>

int getsockopt(int sockfd, int level, int name, char *value, int *optlen);
int setsockopt(int sockfd, int level, int name, char *value, int *optlen);
```

下面是两个调用的参数说明：

- sockfd 必须是一个已打开的套接字。
- level 是函数所使用的协议标准（protocol level）（TCP/IP 协议使用 IPPROTO_TCP，套接字标准的选项实用 SOL_SOCKET）。
- name 选项在套接字说明书中（man page）有详细说明。
- value 指向为 getsockopt()函数所获取的值，setsockopt()函数所设置的值的地址。
- optlen 指针指向一个整数，该整数包含参数以字节计算的长度。

现在我们再回到 Linux 的错误上来．当你打开一个套接字时必须同时用下面的代码段来调用 setsockopt()函数：

```
/* 设定参数数值 */
opt = 1; len = sizeof(opt);
/* 设置套接字属性 */
setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&opt,&len);
```

setsockopt()函数还有很多其他用法，请参考帮助页（man pages）．

6.6.10 getpeername()函数

这个函数可以取得一个已经连接上的套接字的远程信息（比如 IP 地址和端口），告诉

你在远程和你连接的究竟是谁.

它的声明为：

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

下面是参数说明：

- sockfd 是你想取得远程信息的那个套接字描述符。
- addr 是一个指向 struct sockaddr （或是 struct sockaddr_in）的指针。
- addrlen 是一个指向 int 的指针，应该赋于 sizeof(struct sockaddr)的大小。

如果在函数执行过程中出现了错误，函数将返回 –1 ，并且错误代码储存在全局变量 errno 中。

当你拥有了远程连接用户的 IP 地址，你就可以使用 inet_ntoa() 或 gethostbyaddr()来输出信息或是做进一步的处理。

6.6.11 gethostname()函数

gethostname()函数可以取得本地主机的信息．它比 getpeername()要容易使用一些。

它返回正在执行它的计算机的名字。返回的这个名字可以被 gethostbyname()函数使用，由此可以得到本地主机的 IP 地址。

下面是它的声明：

```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
```

参数说明如下：

- hostname 是一个指向字符数组的指针，当函数返回的时候，它里面的数据就是本地的主机的名字.
- size 是 hostname 指向的数组的长度.

函数如果成功执行，它返回 0，如果出现错误，则返回–1，全局变量 errno 中存储着错误代码。

6.7 DNS 的操作

6.7.1 理解 DNS

你应该知道 DNS 吧？DNS 是“ Domain Name Service ”(域名服务）的缩写。有了它，你就可以通过一个可读性非常强的因特网名字得到这个名字所代表的 IP 地址。转换为 IP 地址后，你就可以使用标准的套接字函数（bind()，connect()，sendto()，或是其他任何需要使用的函数)。

在这里，如果你输入命令：

```
$ telnet bbs.tsinghua.edu.cn
```

Telnet 可以知道它需要连往 202.112.58.200。这就是通过 DNS 来实现的。

6.7.2 和 DNS 有关的函数和结构

DNS 是怎样工作的呢？你可以使用 gethostbyname()函数。

它的声明如下：

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

正如你所看见的，它返回了一个指向 struct hostent 的指针．Struct hostent 是这样定义的：

```
struct hostent {
        char *h_name;
        char **h_aliases;
        int h_addrtype;
        int h_length;
        char **h_addr_list;
};
#define h_addr h_addr_list[0]
```

下面是上面各个域代表含义的解释：

- h_name 是这个主机的正式名称。
- h_aliases 是一个以 NULL（空字符）结尾的数组，里面存储了主机的备用名称。
- h_addrtype 是返回地址的类型，一般来说是“ AF_INET ”。
- h_length 是地址的字节长度。
- h_addr_list 是一个以 0 结尾的数组，存储了主机的网络地址。

注意：网络地址是以网络字节顺序存储的。

- h_addr - h_addr_list 数组的第一个成员．

gethostbyname() 返回的指针指向结构 struct hostent ，如果发生错误，它将会返回 NULL（但是 errno 并不代表错误代码，h_errno 中存储的才识错误代码。参考下面的 herror()函数）。

应该如何使用这个函数呢？它看起来有一点点吓人。相信我，它使用起来远远要比它看起来容易。

6.7.3　DNS 例程

下面我们来看一段例程：

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <errno.h>
        #include <netdb.h>
        #include <sys/types.h>
        #include <netinet/in.h>

        int
        main (int argc, char *argv[])
{
struct hostent *h;
/*  检测命令行中的参数是否存在  */
```

```
if (argc != 2)

    /* 如果没有参数，给出使用方法 */
    fprintf (stderr "usage: getip address\n");

    /* 然后退出 */
    exit(1);
}

/* 取得主机信息 */
if ((h=gethostbyname(argv[1])) == NULL)
{
    /* 如果 gethostbyname 失败，则给出错误信息 */
    herror("gethostbyname");
    /* 然后退出 */
    exit(1);
}

/* 列印程序取得的信息 */
printf(" Host name : %s\n", h->h_name);
printf("IP Address : %s\n", inet_ntoa (*((struct in_addr *)h->h_addr)));
/* 返回 */
return 0;
}
```

使用 gethostbyname()函数，你不能使用 perror()来输出错误信息（因为错误代码存储在 h_errno 中而不是 errno 中。所以，你需要调用 herror()函数。

上面的程序是不是很神奇呢？你简单的传给 gethostbyname() 一个机器名（" bbs.tsinghua.edu.cn "），然后就从返回的结构 struct hostent 中得到了 IP 等其他信息．

程序中输出 IP 地址的程序需要解释一下：

h->h_addr 是一个 char*，但是 inet_ntoa()函数需要传递的是一个 struct in_addr 结构。所以上面将 h->h_addr 强制转换为 struct in_addr*，然后通过它得到了所有数据。

6.8 套接字的 Client/Server 结构实现的例子

现在是一个服务器／客户端的世界．几乎网络上的所有工作都是由客户端向服务器端发送请求来实现的．比如 Telnet ，当你向一个远程主机的 23 端口发出连接请求的时候，远程主机上的服务程序（Telnetd）就会接受这个远程连接请求。允许你进行 login 操作。等等。

服务器和客户机之间可以使用任何方式通讯，包括 SOCK_STREAM, SOCK_DGRAM，或是其他任何方式（只要他们使用相同的方法）．

一些服务器／客户机的例子是 telnet/telnetd，ftp/ftpd，bootp/bootpd。每次你使用 ftp，你同时都使用了远程主机上的 ftpd 服务。一般来说，服务器上有一个程序等待连接。当接收到一个连接的时候，服务器程序调用系统函数 fork()来得到一个子进程，专门处理这个连接的操作。

下面我们来看一个简单的流服务器：

6.8.1　简单的流服务器

这个服务器所有的工作就是给远程的终端发送一个字符串：“ Hello,World! ”你所需要做的就是在命令行上启动这个服务器，然后在另外一台机器上使用 telnet 连接到这台我们自己写的服务器上：

```
$ telnet remotehostname 4000
```

remotehostname 就是你运行我们自己写的服务器的那台机器名。

服务器代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
```

```
/* 服务器要监听的本地端口 */
#define MYPORT 4000

/* 能够同时接受多少没有 accept 的连接 */
#define BACKLOG 10

main()
{
    /* 在 sock_fd 上进行监听，new_fd 接受新的连接 */
    int sock_fd, new_fd ;
    /* 自己的地址信息 */
    struct sockaddr_in my_addr;

    /* 连接者的地址信息*/
    struct sockaddr_in their_addr;

    int sin_size;
/* 这里就是我们一直强调的错误检查．如果调用 socket() 出错，则返回 */ if ((sockfd =
```

```
    socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
    /* 输出错误提示并退出 */
    perror("socket");
    exit(1);
    }

    /* 主机字节顺序 */
    my_addr.sin_family = AF_INET;

    /* 网络字节顺序，短整型 */
    my_addr.sin_port = htons(MYPORT);

    /* 将运行程序机器的 IP 填充入 s_addr */
    my_addr.sin_addr.s_addr = INADDR_ANY;

    /* 将此结构的其余空间清零 */
    bzero(&(my_addr.sin_zero), 8);
/* 这里是我们一直强调的错误检查！！ */ if (bind(sockfd, (struct sockaddr *)&my_addr,
    sizeof(struct sockaddr)) == -1)
    {
    /* 如果调用 bind()失败，则给出错误提示，退出 */
    perror("bind");
    exit(1);
    }

    /* 这里是我们一直强调的错误检查！！ */
    if (listen(sockfd, BACKLOG) == -1)
    {
    /* 如果调用 listen 失败，则给出错误提示，退出 */
    perror("listen");
    exit(1);
    }

    while(1)
    {
    /* 这里是主 accept()循环 */
    sin_size = sizeof(struct sockaddr_in);

      /* 这里是我们一直强调的错误检查！！ */
```

```
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
            {
        /* 如果调用 accept()出现错误，则给出错误提示，进入下一个循环 */
      perror("accept");
      continue;
            }
      /* 服务器给出出现连接的信息 */
      printf（"server: got connection from %s\n", inet_ntoa(their_addr.sin_addr)）;
/* 这里将建立一个子进程来和刚刚建立的套接字进行通讯 */
      if (!fork())
      /* 这里是子进程 */
      /* 这里就是我们说的错误检查！ */
      if (send(new_fd, " Hello, world!\n", 14, 0) == -1)
                  {
      /* 如果错误，则给出错误提示，然后关闭这个新连接，退出 */
      perror("send");
      close(new_fd);
      exit(0);
                  }
      /* 关闭 new_fd 代表的这个套接字连接 */
      close(new_fd);
            }
      }
      /* 等待所有的子进程都退出 */
      while(waitpid(-1,NULL,WNOHANG) > 0);
}
```

为了更清楚的描述这个套接字服务器的运行过程，我把所有的代码都写在了这个大大的 main()主函数中。如果你觉得分成几个子程序会清楚一些，你可以自己式着把这个程序改成几个小函数。

你可以使用下面这个套接字客户端来得到 "Hello, World!"这个字符串。

6.8.2　简单的流式套接字客户端程序

这个程序比起服务器端程序要简单一些。它所做的工作就是 connect()到服务器的 4000 端口，然后把服务器发送的字符串给显示出来。

客户端程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
```

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

/* 服务器程序监听的端口号 */
#define PORT 4000

/* 我们一次所能够接收的最大字节数 */
#define MAXDATASIZE 100

int
main(int argc, char *argv[])
{
    /* 套接字描述符 */
    int sockfd, numbytes;
    char buf[MAXDATASIZE];

    struct hostent *he;
    /* 连接者的主机信息 */
    struct sockaddr_in their_addr;

    /* 检查参数信息 */
    if (argc != 2)
    {
    /* 如果没有参数，则给出使用方法后退出 */
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
    }

    /* 取得主机信息 */
    if ((he=gethostbyname(argv[1])) == NULL)
    /* 如果 gethostbyname()发生错误，则显示错误信息并退出 */
        herror("gethostbyname");
        exit(1);
    }
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    /* 如果 socket()调用出现错误则显示错误信息并退出 */
    perror("socket");
    exit(1);
    }
```

```
    /* 主机字节顺序 */
    their_addr.sin_family = AF_INET;

    /* 网络字节顺序，短整型 */
    their_addr.sin_port = htons(PORT);

    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    /* 将结构剩下的部分清零*/
    bzero(&(their_addr.sin_zero), 8);

    if（connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1）
    {
    /* 如果 connect()建立连接错误，则显示出错误信息，退出 */
    perror(“connect”);
    exit(1);
    }

    if（(numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1）
    {
    /* 如果接收数据错误，则显示错误信息并退出 */
    perror(“recv”);
    exit(1);
    }

    buf[numbytes] = ‘\0’;
    printf(“Received: %s”,buf);
    close(sockfd);
    return 0;
}
```

注意：显然，你必须在运行 client **之前先启动** server**。否则** client **的执行会出错（显示**“ Connection refused”**）。**

6.8.3　数据报套接字例程（DatagramSockets）

在这里我不对数据报做过多的描述，下面你将看见另外一对例程（使用数据报）：talker.c **和** listener.c**。**

listener **在一台机器上作为服务器程序运行，它监听端口** 5000 **．**

talker **发送** UDP **数据包到服务器的** 5000 **端口，传送使用者的数据。**

下面是 listener.c **的源码：**

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

```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

/* 要连接到的端口号 */
#define MYPORT 5000

/* 能够接收的最长数据 */
#define MAXBUFLEN 100

main()
{
int sockfd;
    /* 本机的地址信息 */
    struct sockaddr_in my_addr;

    /* 连接这的地址信息 */
    struct sockaddr_in their_addr;
    int addr_len, numbytes;
    char buf[MAXBUFLEN];

    /* 取得一个套接字描述符 */
    if（(sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1）
    {
    /* 如果取得套接字描述符失败，则给出错误信息，退出 */
    perror(“socket”);
    exit(1);
    }
    /* 主机字节顺序 */
    my_addr.sin_family = AF_INET;

    /* 网络字节顺序，短整型 */
    my_addr.sin_port = htons(MYPORT);

    /* 自动设置为自己的 IP */
    my_addr.sin_addr.s_addr = INADDR_ANY;
```

```
    /* 将结构的其余空间清零 */
    bzero(&(my_addr.sin_zero), 8);

      /* 绑定端口 */
      if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
    /* 如果绑定端口出错，则显示错误信息然后退出 */
    perror("bind");
    exit(1);
    }

    addr_len = sizeof(struct sockaddr);
/* 接收数据 */
    if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN, 0,
      (struct sockaddr *)&their_addr, &addr_len)) == -1)
    {
    /* 如果 recvfrom()调用出错，则显示错误信息后退出 */
    perror("recvfrom");
    exit(1);
    }

    /* 显示接收到的数据 */
    printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
    printf("packet is %d bytes long\n",numbytes);
    buf[numbytes] = '\0';
    printf("packet contains \"%s\"\n",buf);

    /* 关闭套接字连接 */
    close(sockfd);
}
```

注意我们调用 socket()**函数的时候使用的是** SOCK_DGRAM **为参数。而且，我们并不需要** listen()**或是** accept()**。这是因为我们使用了无连接的使用者数据报套接字！**

下面的是 talker.c **的源码：**

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
```

```
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

/* 要连接的端口 */
#define MYPORT 5000

int main(int argc, char *argv[])
{
int sockfd;
    /* 连接者的地址信息 */
    struct sockaddr_in their_addr;
    struct hostent *he;
    int numbytes;

    if (argc != 3)
    {
      /* 检测是否有所须参数，如没有，则显示使用方法后退出 */
      fprintf(stderr,“usage: talker hostname message\n”);
      exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL)
    {
      /* 取得主机的信息，如果失败则显示错误信息后退出 */
      herror(“gethostbyname”);
      exit(1);
    }

    if（(sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
    /* 申请一个数据报套接字描述符，失败则退出 */
    perror (“socket”);
    exit(1);
    }

    /* 主机字节顺序 */
    their_addr.sin_family = AF_INET;

    /* 网络字节顺序，短整型 */
```

```
    their_addr.sin_port = htons(MYPORT);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    /*  将结构中未用的部分清零  */
    bzero(&(their_addr.sin_zero), 8);
    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
       (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
    {
    /*  把信息发送到指定的主机指定端口，如出错则提示退出  */
    perror("recvfrom");
    exit(1);
    }

    printf("sent %d bytes to %s\n",numbytes,inet_ntoa(their_addr.sin_addr));
    /*  关闭套接字描述符后退出  */
    close(sockfd);
    return 0;
}
```

上面这两个程序，你需要在一台主机上首先运行 listener**，然后在另外一台主机上运行** talker**。现在看到它们之间的通讯了吗？**

最后，我们要注意一点：使用连接的数据报套接字。因为我们在讲使用数据报，所以我们需要了解它。如果我们的 talker **程序使用了** connect()**函数来连接** listener **的地址，那么** talker **程序就能够使用** sent()**和** recv()**来处理数据了。因为** talker **程序在** connect()**函数中已经知道了远程主机的地址和端口号.**

6.9　保留端口

6.9.1　简介

大多数网络应用程序使用两个协议：传输控制协议（TCP**）和用户数据包协议（**UDP**）。他们都使用一个端口号以识别应用程序。端口号为主机上所运行之程序所用，这样就可以通过号码象名字一样来跟踪每个应用程序。端口号让操作系统更容易的知道有多少个应用程序在使用系统，以及哪些服务有效。**

理论上，端口号可由每台主机上的管理员自由的分配。但为了更好的通信通常采用一些约定的协议。这些协议使能通过端口号识别一个系统向另一个系统所请求的服务的类型。基于如此理由，大多数系统维护一个包含端口号及它们所提供哪些服务的文件。

端口号被从 1 **开始分配。通常端口号超出** 255 **的部分被本地主机保留为私有用途。**1 **到** 255 **之间的号码被用于远程应用程序所请求的进程和网络服务。每个网络通信循环地进出主计算机的** TCP **应用层。它被两个所连接的号码唯一地识别。这两个号码合起来叫做套接字.组成套接字的这两个号码就是机器的** IP **地址和** TCP **软件所使用的端口号。**

因为网络通讯至少包括两台机器，所以在发送和接收的机器上都存在一个套接字。由于每台机器的 IP **地址是唯一的。端口号在每台机器中也是唯一的，所以套接字在网络中应**

该是唯一的。这样的设置能使网络中的两个应用程序完全的基于套接字互相对话。

发送和接收的机器维护一个端口表，它列出了所有激活的端口号。两台机器都包括一个进程叫做绑定，这是每个任务的入口，不过在两台机器上恰恰相反。换句话说，如果一台机器的源端口号是 23 而目的端口号被设置成 25，那么另一台机器的源端口号设置成 25 目的端口号设置成 23。

6.9.2 保留端口

系统留有 1024 个保留端口。这些端口是留给系统使用的，在系统中，只有具有 Root 权利的人才可以使用 1024 以下的端口（包括 1024）

这里是 RedHat 6.0 中 /etc/services 文件：

```
[root@bbs /etc]# cat /etc/services
# /etc/services:
# $Id: services,v 1.4 1997/05/20 19:41:21 tobias Exp $
#
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, most entries here have two entries
# even if the protocol doesn‘t support UDP operations.
# Updated from RFC 1700, “Assigned Numbers” (October 1994).  Not all ports
# are included, only the more common ones.
tcpmux          1/tcp                           # TCP port service multiplexer
ztelnet         2/tcp
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                          # message send protocol
msp             18/udp                          # message send protocol
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
```

```
fsp             21/udp          fspd
ssh             22/tcp                  # SSH Remote Login Protocol
ssh             22/udp                  # SSH Remote Login Protocol
telnet          23/tcp
#stelnet                30/tcp
# 24 - private
smtp            25/tcp          mail
# 26 - unassigned
time            37/tcp          timserver
time            37/udp          timserver
rlp             39/udp          resource        # resource location
nameserver      42/tcp          name            # IEN 116
whois           43/tcp          nicname
re-mail-ck      50/tcp                  # Remote Mail Checking Protocol
re-mail-ck      50/udp                  # Remote Mail Checking Protocol
domain          53/tcp          nameserver      # name-domain server
domain          53/udp          nameserver
mtp             57/tcp                  # deprecated
bootps          67/tcp                  # BOOTP server
bootps          67/udp
bootpc          68/tcp                  # BOOTP client
bootpc          68/udp
tftp            69/udp
gopher          70/tcp                  # Internet Gopher
gopher          70/udp
rje             77/tcp          netrjs
finger          79/tcp
www             80/tcp          http            # WorldWideWeb HTTP
www             80/udp                  # HyperText Transfer Protocol
link            87/tcp          ttylink
kerberos        88/tcp          kerberos5 krb5  # Kerberos v5
kerberos        88/udp          kerberos5 krb5  # Kerberos v5
supdup          95/tcp
# 100 - reserved
hostnames       101/tcp         hostname        # usually from sri-nic
iso-tsap        102/tcp         tsap            # part of ISODE.
csnet-ns        105/tcp         cso-ns          # also used by CSO name server
csnet-ns        105/udp         cso-ns
# unfortunately the poppassd (Eudora) uses a port which has already
```

```
# been assigned to a different service. We list the poppassd as an
# alias here. This should work for programs asking for this service.
# (due to a bug in inetd the 3com-tsmux line is disabled)
#3com-tsmux      106/tcp         poppassd
#3com-tsmux      106/udp         poppassd
rtelnet          107/tcp                 # Remote Telnet
rtelnet          107/udp
pop-2            109/tcp         postoffice      # POP version 2
pop-2            109/udp
pop-3            110/tcp                 # POP version 3
pop-3            110/udp
sunrpc           111/tcp         portmapper      # RPC 4.0 portmapper TCP
sunrpc           111/udp         portmapper      # RPC 4.0 portmapper UDP
#by zixia RPC            111/tcp         portmapper      # RPC 4.0 portmapper TCP
#RPC              111/udp         portmapper      # RPC 4.0 portmapper UDP
auth             113/tcp         authentication tap ident
sftp             115/tcp
uucp-path        117/tcp
nntp             119/tcp         readnews untp   # USENET News Transfer Protocol
ntp              123/tcp
ntp              123/udp                 # Network Time Protocol
netbios-ns       137/tcp                 # NETBIOS Name Service
netbios-ns       137/udp
netbios-dgm      138/tcp                 # NETBIOS Datagram Service
netbios-dgm      138/udp
netbios-ssn      139/tcp                 # NETBIOS session service
netbios-ssn      139/udp
imap2            143/tcp         imap            # Interim Mail Access Proto v2
imap2            143/udp         imap
snmp             161/udp                 # Simple Net Mgmt Proto
snmp-trap        162/udp         snmptrap        # Traps for SNMP
cmip-man         163/tcp                 # ISO mgmt over IP (CMOT)
cmip-man         163/udp
cmip-agent       164/tcp
cmip-agent       164/udp
xdmcp            177/tcp                 # X Display Mgr. Control Proto
xdmcp            177/udp
nextstep         178/tcp         NeXTStep NextStep       # NeXTStep window
nextstep         178/udp         NeXTStep NextStep       # server
bgp              179/tcp                 # Border Gateway Proto.
```

```
bgp             179/udp
prospero        191/tcp                 # Cliff Neuman's Prospero
prospero        191/udp
irc             194/tcp                 # Internet Relay Chat
irc             194/udp
smux            199/tcp                 # SNMP UNIX Multiplexer
smux            199/udp
at-rtmp         201/tcp                 # AppleTalk routing
at-rtmp         201/udp
at-nbp          202/tcp                 # AppleTalk name binding
at-nbp          202/udp
at-echo         204/tcp                 # AppleTalk echo
at-echo         204/udp
at-zis          206/tcp                 # AppleTalk zone information
at-zis          206/udp
qmtp            209/tcp                 # The Quick Mail Transfer Protocol
qmtp            209/udp                 # The Quick Mail Transfer Protocol
z3950           210/tcp         wais    # NISO Z39.50 database
z3950           210/udp         wais
ipx             213/tcp                 # IPX
ipx             213/udp
imap3           220/tcp                 # Interactive Mail Access
imap3           220/udp                 # Protocol v3
rpc2portmap     369/tcp
rpc2portmap     369/udp                 # Coda portmapper
codaauth2       370/tcp
codaauth2       370/udp                 # Coda authentication server
ulistserv       372/tcp                 # UNIX Listserv
ulistserv       372/udp
https           443/tcp                 # MCom
https           443/udp                 # MCom
snpp            444/tcp                 # Simple Network Paging Protocol
snpp            444/udp                 # Simple Network Paging Protocol
saft            487/tcp                 # Simple Asynchronous File Transfer
saft            487/udp                 # Simple Asynchronous File Transfer
npmp-local      610/tcp         dqs313_qmaster  # npmp-local / DQS
npmp-local      610/udp         dqs313_qmaster  # npmp-local / DQS
npmp-gui        611/tcp         dqs313_execd    # npmp-gui / DQS
npmp-gui        611/udp         dqs313_execd    # npmp-gui / DQS
hmmp-ind        612/tcp         dqs313_intercell# HMMP Indication / DQS
```

```
hmmp-ind        612/udp         dqs313_intercell# HMMP Indication / DQS
#
# UNIX specific services
#
exec            512/tcp
biff            512/udp         comsat
login           513/tcp
who             513/udp         whod
shell           514/tcp         cmd             # no passwords used
syslog          514/udp
printer         515/tcp         spooler         # line printer spooler
talk            517/udp
ntalk           518/udp
route           520/udp         router routed   # RIP
timed           525/udp         timeserver
tempo           526/tcp         newdate
courier         530/tcp         rpc
conference      531/tcp         chat
netnews         532/tcp         readnews
netwall         533/udp                         # -for emergency broadcasts
uucp            540/tcp         uucpd           # uucp daemon
afpovertcp      548/tcp                         # AFP over TCP
afpovertcp      548/udp                         # AFP over TCP
remotefs        556/tcp         rfs_server rfs  # Brunhoff remote filesystem
klogin          543/tcp                         # Kerberized 'rlogin' (v5)
kshell          544/tcp         krcmd           # Kerberized 'rsh' (v5)
kerberos-adm    749/tcp                         # Kerberos 'kadmin' (v5)
#
webster         765/tcp                         # Network dictionary
webster         765/udp
#
# From "Assigned Numbers":
#
#> The Registered Ports are not controlled by the IANA and on most systems
#> can be used by ordinary user processes or programs executed by ordinary
#> users.
#
#> Ports are used in the TCP [45,106] to name the ends of logical
#> connections which carry long term conversations.  For the purpose of
```

```
#> providing services to unknown callers, a service contact port is
#> defined.  This list specifies the port used by the server process as its
#> contact port.  While the IANA can not control uses of these ports it
#> does register or list uses of these ports as a convienence to the
#> community.
#
ingreslock         1524/tcp
ingreslock         1524/udp
prospero-np        1525/tcp                    # Prospero non-privileged
prospero-np        1525/udp
datametrics        1645/tcp        old-radius  # datametrics / old radius entry
datametrics        1645/udp        old-radius  # datametrics / old radius entry
sa-msg-port        1646/tcp        old-radacct # sa-msg-port / old radacct entry
sa-msg-port        1646/udp        old-radacct # sa-msg-port / old radacct entry
radius             1812/tcp                    # Radius
radius             1812/udp                    # Radius
radacct            1813/tcp                    # Radius Accounting
radacct            1813/udp                    # Radius Accounting
cvspserver         2401/tcp                    # CVS client/server operations
cvspserver         2401/udp                    # CVS client/server operations
venus              2430/tcp                    # codacon port
venus              2430/udp                    # Venus callback/wbc interface
venus-se           2431/tcp                    # tcp side effects
venus-se           2431/udp                    # udp sftp side effect
codasrv            2432/tcp                    # not used
codasrv            2432/udp                    # server port
codasrv-se         2433/tcp                    # tcp side effects
codasrv-se         2433/udp                    # udp sftp side effect
mysql              3306/tcp                    # MySQL
mysql              3306/udp                    # MySQL
rfe                5002/tcp                    # Radio Free Ethernet
rfe                5002/udp                    # Actually uses UDP only
cfengine           5308/tcp                    # CFengine
cfengine           5308/udp                    # CFengine
bbs                7000/tcp                    # BBS service
#
#
```

```
# Kerberos (Project Athena/MIT) services
# Note that these are for Kerberos v4, and are unofficial.  Sites running
# v4 should uncomment these and comment out the v5 entries above.
```

```
#
kerberos4       750/udp         kerberos-iv kdc # Kerberos (server) udp
kerberos4       750/tcp         kerberos-iv kdc # Kerberos (server) tcp
kerberos_master 751/udp                         # Kerberos authentication
kerberos_master 751/tcp                         # Kerberos authentication
passwd_server   752/udp                         # Kerberos passwd server
krb_prop        754/tcp                         # Kerberos slave propagation
krbupdate       760/tcp         kreg            # Kerberos registration
kpasswd         761/tcp         kpwd            # Kerberos "passwd"
kpop            1109/tcp                        # Pop with Kerberos
knetd           2053/tcp                        # Kerberos de-multiplexor
zephyr-srv      2102/udp                        # Zephyr server
zephyr-clt      2103/udp                        # Zephyr serv-hm connection
zephyr-hm       2104/udp                        # Zephyr hostmanager
eklogin         2105/tcp                        # Kerberos encrypted rlogin
#
# Unofficial but necessary (for NetBSD) services
#
supfilesrv      871/tcp                         # SUP server
supfiledbg      1127/tcp                        # SUP debugging
#
# Datagram Delivery Protocol services
#
rtmp            1/ddp                           # Routing Table Maintenance Protocol
nbp             2/ddp                           # Name Binding Protocol
echo            4/ddp                           # AppleTalk Echo Protocol
zip             6/ddp                           # Zone Information Protocol
#
# Services added for the Debian GNU/Linux distribution
poppassd        106/tcp                         # Eudora
poppassd        106/udp                         # Eudora
mailq           174/tcp                         # Mailer transport queue for Zmailer
mailq           174/tcp                         # Mailer transport queue for Zmailer
ssmtp           465/tcp                         # SMTP over SSL
gdomap          538/tcp                         # GNUstep distributed objects
gdomap          538/udp                         # GNUstep distributed objects
snews           563/tcp                         # NNTP over SSL
ssl-ldap        636/tcp                         # LDAP over SSL
omirr           808/tcp         omirrd          # online mirror
```

```
omirr           808/udp         omirrd          # online mirror
rsync           873/tcp                         # rsync
rsync           873/udp                         # rsync
simap           993/tcp                         # IMAP over SSL
spop3           995/tcp                         # POP-3 over SSL
socks           1080/tcp                        # socks proxy server
socks           1080/udp                        # socks proxy server
rmtcfg          1236/tcp                        # Gracilis Packeten remote config
server
xtel            1313/tcp                        # french minitel
support         1529/tcp                        # GNATS
cfinger         2003/tcp                        # GNU Finger
ninstall        2150/tcp                        # ninstall service
ninstall        2150/udp                        # ninstall service
afbackup        2988/tcp                        # Afbackup system
afbackup        2988/udp                        # Afbackup system
icp             3130/tcp                        # Internet Cache Protocol (Squid)
icp             3130/udp                        # Internet Cache Protocol (Squid)
postgres        5432/tcp                        # POSTGRES
postgres        5432/udp                        # POSTGRES
fax             4557/tcp                        # FAX transmission service
(old)
hylafax         4559/tcp                        # HylaFAX client-server protocol
(new)
noclog          5354/tcp                        # noclogd with TCP (nocol)
noclog          5354/udp                        # noclogd with UDP (nocol)
hostmon         5355/tcp                        # hostmon uses TCP (nocol)
hostmon         5355/udp                        # hostmon uses TCP (nocol)
ircd            6667/tcp                        # Internet Relay Chat
ircd            6667/udp                        # Internet Relay Chat
webcache        8080/tcp                        # WWW caching service
webcache        8080/udp                        # WWW caching service
tproxy          8081/tcp                        # Transparent Proxy
tproxy          8081/udp                        # Transparent Proxy
mandelspawn     9359/udp        mandelbrot      # network mandelbrot
amanda          10080/udp                       # amanda backup services
kamanda         10081/tcp                       # amanda backup services (Kerberos)
kamanda         10081/udp                       # amanda backup services (Kerberos)
amandaidx       10082/tcp                       # amanda backup services
amidxtape       10083/tcp                       # amanda backup services
```

```
isdnlog         20011/tcp                       # isdn logging system
isdnlog         20011/udp                       # isdn logging system
vboxd           20012/tcp                       # voice box system
vboxd           20012/udp                       # voice box system
binkp           24554/tcp                       # Binkley
binkp           24554/udp                       # Binkley
asp             27374/tcp                       # Address Search Protocol
asp             27374/udp                       # Address Search Protocol
tfido           60177/tcp                       # Ifmail
tfido           60177/udp                       # Ifmail
fido            60179/tcp                       # Ifmail
fido            60179/udp                       # Ifmail
```

```
# Local services
linuxconf 98/tcp
swat            901/tcp                         # Add swat service used via inetd
[root@bbs /etc]#
```

下面，我们以以 Web Server **的端口** 80 **做例子来看看这份单子说明了些什么？：**

它在 services **文件中的那一行是这样的：**

```
www             80/tcp          http            # WorldWideWeb HTTP
```

大家可以看到这一行分 3 **部分：**

- www **代表** HTTP **协议的端口名（也就是缺省的** Web Browser **连接服务器时的端口）。**
- 80/tcp **这一部分是用 "／" 号分开的，前半部分表示的是端口号（这里的** HTTP **协议的端口是** 80**），后半部分表示是一个** TCP **连接（也就是有连接的套接字，相对应的是** UDP**）。**
- http **代表是** HTTP **协议。**
- # WorldWideWeb HTTP **最后大家看到的这个是以“＃”号打头的，是一些注释。**

我们所在意的其实只有 HTTP **和** 80 **．通过这个规律，大家可以看到这个** RedHat 6.0 **自己所定义的保留端口（其中包括一些大于** 1024 **的端口）**

这个文件只是定义了每个服务所使用的端口和它的别名。假如你运行

```
$telnet 127.0.0.1 www
```

那么你就连接到了本地的 Web **服务器上（当然，前提是你已经启动了这个** Web **服务器）。**

自己编程的时候应该尽量避免自己的服务器所使用的端口和系统的 Services **文件中已经声明的端口重叠。避免的方法除了参考系统的** Services **文件以外，你还可以直接对系统进行** telnet **来进行测试。**

比如你的程序想使用 4000 **端口进行监听网络连接，你为了确定是否已经有程序使用了** 4000 **端口，可以像下面这样操作：**

```
$telnet 127.0.0.1 4000
```

如果系统给出了错误信息：

```
[root@bbs /etc]# telnet 127.0.0.1 4000
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
```

那么说明系统中没有程序使用 4000 端口，你可以放心的使用了。

技巧：如果你自己写了一个 Server 和 Client ，但是 Client 却无法连上 Server 而你又不知道究竟是哪个有问题的时候，你可以使用系统的工具 telnet 来帮助你。如果你的 Server 监听的端口是 4000 ，那么可以直接使用 telnet 去连接 4000 端口。如果使用 telnet 连接正常，那么你就可以确定你的 Server 运行正常.

6.10 五种 I/O 模式

下面我们简单的介绍一个各种 I/O 操作模式。在 Linux/UNIX 下，有下面这五种 I/O 操作方式：

- 阻塞 I/O
- 非阻塞 I/O
- I/O 多路复用
- 信号驱动 I/O（SIGIO）
- 异步 I/O

这章讲述了一些 I/O 的细节，你可以在第一次阅读的时候跳过这部分，然后在第二次阅读本书的时候再来读这一节。

一般来说，程序进行输入操作有两步：

1．等待有数据可以读

2．将数据从系统内核中拷贝到程序的数据区。

对于一个对套接字的输入操作，第一步一般来说是等待数据从网络上传到本地。当数据包到达的时候，数据将会从网络层拷贝到内核的缓存中；第二步是从内核中把数据拷贝到程序的数据区中。

6.10.1　阻塞 I/O 模式

阻塞 I/O 模式是最普遍使用的 I/O 模式。大部分程序使用的都是阻塞模式的 I/O 。缺省的，一个套接字建立后所处于的模式就是阻塞 I/O 模式。

对于一个 UDP 套接字来说，数据就绪的标志比较简单：

- 已经收到了一整个数据报
- 没有收到。

而 TCP 这个概念就比较复杂，需要附加一些其他的变量。

在图 6-4 中，一个进程调用 recvfrom ，然后系统调用并不返回知道有数据报到达本地系统，然后系统将数据拷贝到进程的缓存中。（如果系统调用收到一个中断信号，则它的调用会被中断）

我们称这个进程在调用 recvfrom 一直到从 recvfrom 返回这段时间是阻塞的。当 recvfrom 正常返回时，我们的进程继续它的操作。

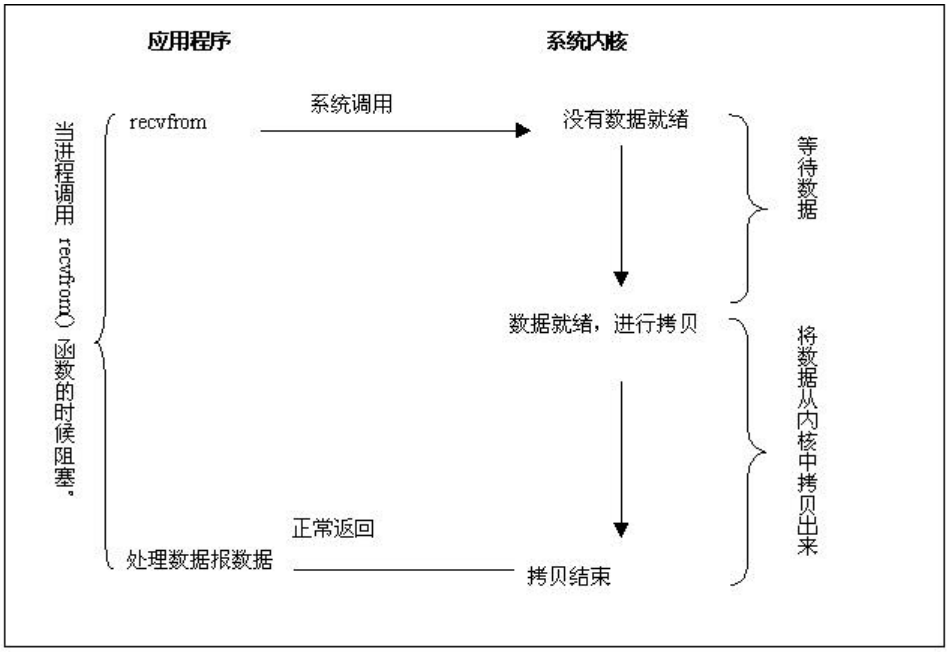

图 6-4　tcp 连接的简单示例

6.10.2　非阻塞模式 I/O

当我们将一个套接字设置为非阻塞模式，我们相当于告诉了系统内核：“当我请求的 I/O 操作不能够马上完成，你想让我的进程进行休眠等待的时候，不要这么做，请马上返回一个错误给我。”

我们可以参照图 6-5 来描述非阻塞模式 I/O 。

我们开始对 recvfrom 的三次调用，因为系统还没有接收到网络数据，所以内核马上返回一个 EWOULDBLOCK 的错误。第四次我们调用 recvfrom 函数，一个数据报已经到达了，内核将它拷贝到我们的应用程序的缓冲区中，然后 recvfrom 正常返回，我们就可以对接收到的数据进行处理了。

当一个应用程序使用了非阻塞模式的套接字，它需要使用一个循环来不听的测试是否一个文件描述符有数据可读（称做 polling）。应用程序不停的 polling 内核来检查是否 I/O 操作已经就绪。这将是一个极浪费 CPU 资源的操作。这种模式使用中不是很普遍。

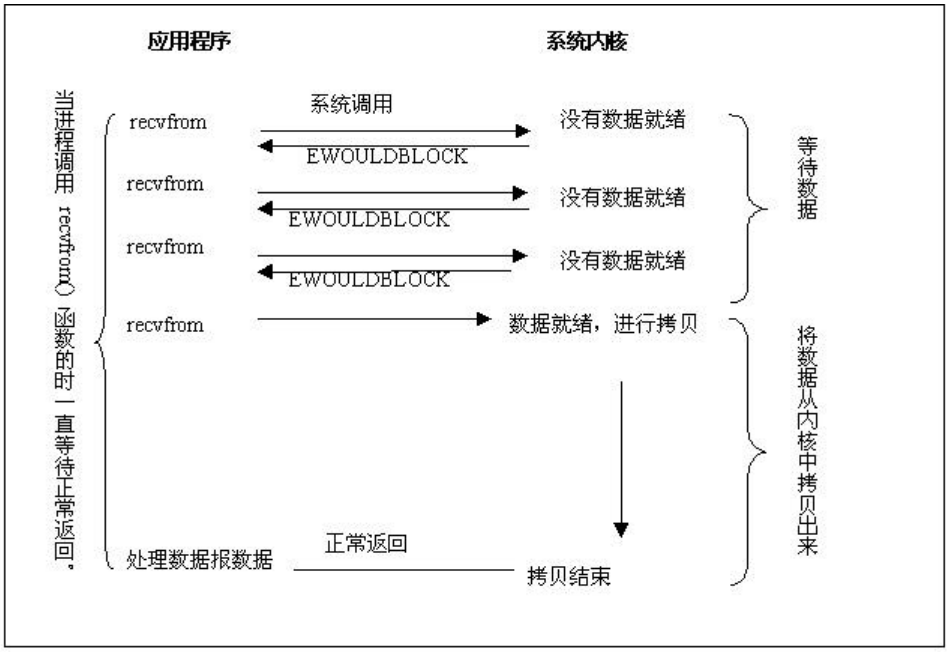

图 6-5　非阻塞模式 I/O

6.10.3　I/O 多路复用

在使用 I/O 多路技术的时候，我们调用 select()函数和 poll()函数，在调用它们的时候阻塞，而不是我们来调用 recvfrom（或 recv）的时候阻塞。图 6-6 说明了它的工作方式。

当我们调用 select 函数阻塞的时候，select 函数等待数据报套接字进入读就绪状态。当 select 函数返回的时候，也就是套接字可以读取数据的时候。这时候我们就可以调用 recvfrom 函数来将数据拷贝到我们的程序缓冲区中。

和阻塞模式相比较，select()和 poll()并没有什么高级的地方，而且，在阻塞模式下只需要调用一个函数：读取或发送，在使用了多路复用技术后，我们需要调用两个函数了：先调用 select()函数或 poll()函数，然后才能进行真正的读写。

多路复用的高级之处在于，它能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

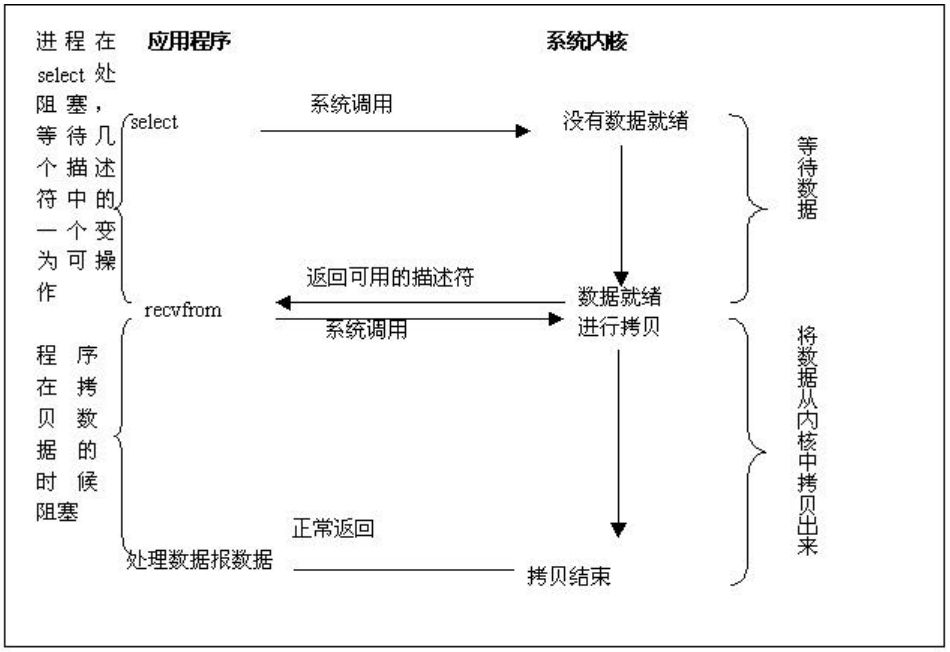

图 6-6 I/O 多路复用

假设我们运行一个网络客户端程序，要同时处理套接字传来的网络数据又要处理本地的标准输入输出。在我们的程序处于阻塞状态等待标准输入的数据的时候，假如服务器端的程序被 kill（或是自己 Down 掉了），那么服务器程端的 TCP 协议会给客户端（我们这端）的 TCP 协议发送一个 FIN 数据代表终止连接。但是我们的程序阻塞在等待标准输入的数据上，在它读取套接字数据之前（也许是很长一段时间），它不会看见结束标志．我们就不能够使用阻塞模式的套接字。

IO 多路技术一般在下面这些情况中被使用：

- 当一个客户端需要同时处理多个文件描述符的输入输出操作的时候（一般来说是标准的输入输出和网络套接字）， I/O 多路复用技术将会有机会得到使用。
- 当程序需要同时进行多个套接字的操作的时候。
- 如果一个 TCP 服务器程序同时处理正在侦听网络连接的套接字和已经连接好的套接字。
- 如果一个服务器程序同时使用 TCP 和 UDP 协议。
- 如果一个服务器同时使用多种服务并且每种服务可能使用不同的协议（比如 inetd 就是这样的）。

I/O 多路服用技术并不只局限与网络程序应用上。几乎所有的程序都可以找到应用 I/O 多路复用的地方。

6.10.4 信号驱动 I/O 模式

我们可以使用信号，让内核在文件描述符就绪的时候使用 SIGIO 信号来通知我们。我们将这种模式称为信号驱动 I/O 模式。

使用这种模式，我们首先需要允许套接字使用信号驱动 I/O ，还要安装一个 SIGIO 的

处理函数。在这种模式下，系统调用将会立即返回，然后我们的程序可以继续做其他的事情。当数据就绪的时候，系统会向我们的进程发送一个 SIGIO 信号。这样我们就可以在 SIGIO 信号的处理函数中进行 I/O 操作（或是我们在函数中通知主函数有数据可读）。

我们现在还不必对 SIGIO 信号处理函数做过多的了解（在下一章中我们会介绍信号的有关内容）。对于信号驱动 I/O 模式，它的先进之处在于它在等待数据的时候不会阻塞，程序可以做自己的事情。当有数据到达的时候，系统内核会向程序发送一个 SIGIO 信号进行通知，这样我们的程序就可以获得更大的灵活性，因为我们不必为等待数据进行额外的编码。

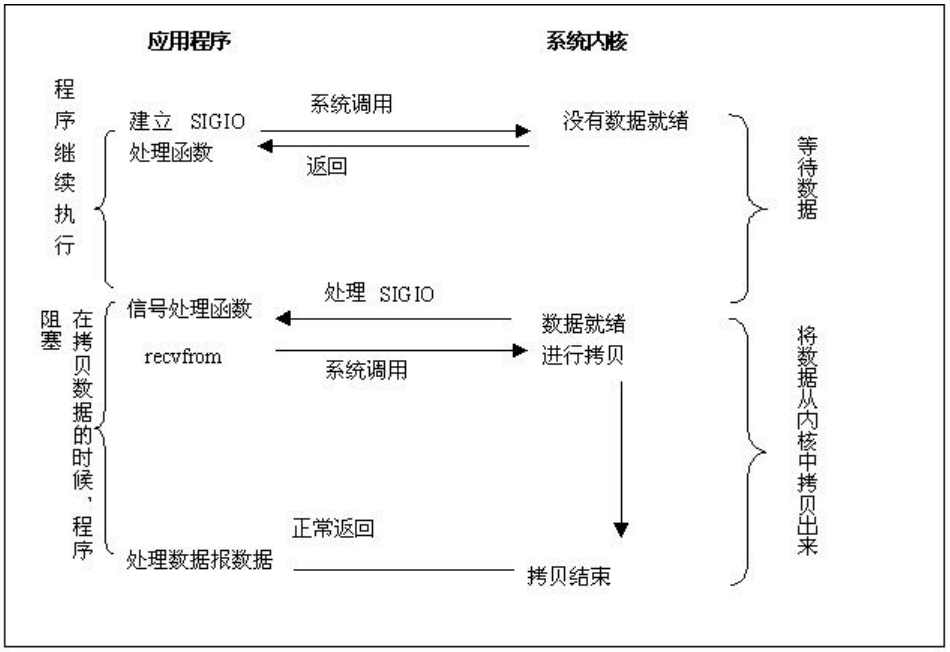

图 6-7 信号驱动 I/O

信号 I/O 可以使内核在某个文件描述符发生改变的时候发信号通知我们的程序。异步 I/O 可以提高我们程序进行 I/O 读写的效率。通过使用它，当我们的程序进行 I/O 操作的时候，内核可以在初始化 I/O 操作后立即返回，在进行 I/O 操作的同时，我们的程序可以做自己的事情，直到 I/O 操作结束，系统内核给我们的程序发消息通知。

基于 Berkeley 接口的 Socket 信号驱动 I/O 使用信号 SIGIO。有的系统 SIGPOLL 信号，它也是相当于 SIGIO 的。

为了在一个套接字上使用信号驱动 I/O 操作，下面这三步是所必须的。

（1）一个和 SIGIO 信号的处理函数必须设定。

（2）套接字的拥有者必须被设定。一般来说是使用 fcntl 函数的 F_SETOWN 参数来进行设定拥有者。

（3）套接字必须被允许使用异步 I/O。一般是通过调用 fcntl 函数的 F_SETFL 命令，O_ASYNC 为参数来实现。

注意：我们在设置套接字的属主之前必须将 SIGIO 的信号处理函数设好，SIGIO 的缺省动作是被忽略。因此我们如果以相反的顺序调用这两个函数调用，那么在 fcntl 函数调用之后，signal 函数调用之前就

有一小段时间程序可能接收到 SIGIO 信号。那样的话，信号将会被丢弃。在 SVR4 系统中，SIGIO 在 <sys/signal.h> 头文件中被定义为 SIGPOLL，而 SIGPOLL 信号的缺省动作是终止这个进程。所以我们一定要保证这两个函数的调用顺序：先调用 signal 设置好 SIGIO 信号处理函数，然后在使用 fcntl 函数设置套接字的属主。

虽然设定套接字为异步 I/O 非常简单，但是使用起来困难的部分是怎样在程序中断定产生 SIGIO 信号发送给套接字属主的时候，程序处在什么状态。

1．UDP 套接字的 SIGIO 信号

在 UDP 协议上使用异步 I/O 非常简单．这个信号将会在这个时候产生：

- 套接字收到了一个数据报的数据包。
- 套接字发生了异步错误。

当我们在使用 UDP 套接字异步 I/O 的时候，我们使用 recvfrom()函数来读取数据报数据或是异步 I/O 错误信息。

2．TCP 套接字的 SIGIO 信号

不幸的是，异步 I/O 几乎对 TCP 套接字而言没有什么作用。因为对于一个 TCP 套接字来说， SIGIO 信号发生的几率太高了，所以 SIGIO 信号并不能告诉我们究竟发生了什么事情。在 TCP 连接中， SIGIO 信号将会在这个时候产生：

- 在一个监听某个端口的套接字上成功的建立了一个新连接。
- 一个断线的请求被成功的初始化。
- 一个断线的请求成功的结束。
- 套接字的某一个通道（发送通道或是接收通道）被关闭。
- 套接字接收到新数据。
- 套接字将数据发送出去。
- 发生了一个异步 I/O 的错误。

举例来说，如果一个正在进行读写操作的 TCP 套接字处于信号驱动 I/O 状态下，那么每当新数据到达本地的时候，将会产生一个 SIGIO 信号，每当本地套接字发出的数据被远程确认后，也会产生一个 SIGIO 信号。对于我们的程序来讲，是无法区分这两个 SIGIO 有什么区别的。在这种情况下使用 SIGIO，TCP 套接字应当被设置为无阻塞模式来阻止一个阻塞的 read 和 write（recv 和 send）操作。我们可以考虑在一个只进行监听网络连接操作的套接字上使用异步 I/O，这样当有一个新的连接的时候，SIGIO 信号将会产生。

一个对信号驱动 I/O 比较实用的方面是 NTP（网络时间协议 Network Time Protocol）服务器，它使用 UDP。这个服务器的主循环用来接收从客户端发送过来的数据报数据包，然后再发送请求。对于这个服务器来说，记录下收到每一个数据包的具体时间是很重要的。因为那将是返回给客户端的值，客户端要使用这个数据来计算数据报在网络上来回所花费的时间。图 6-8 表示了怎样建立这样的一个 UDP 服务器。

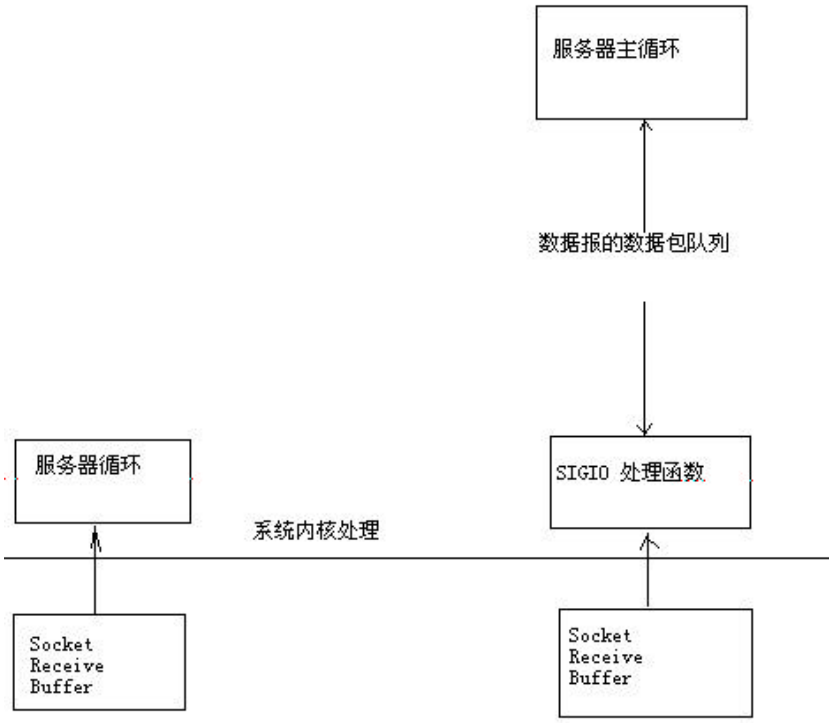

图 6-8　NTP 服务器

大多数的 UDP 服务都被设计成图左边的模式。但是 NTP 服务器使用的是图右边的技术。当有一个新的数据报到达的时候，SIGIO 的处理函数会取出它放入一个程序等待读取的队列，主程序会从这个队列中读取数据。虽然这样会增加程序代码的长度，但是它能够获取数据包到达服务器程序的准确时间.

6.10.5　异步 I/O 模式

当我们运行在异步 I/O 模式下时，我们如果想进行 I/O 操作，只需要告诉内核我们要进行 I/O 操作，然后内核会马上返回。具体的 I/O 和数据的拷贝全部由内核来完成，我们的程序可以继续向下执行。当内核完成所有的 I/O 操作和数据拷贝后，内核将通知我们的程序。

异步 I/O 和 信号驱动 I/O 的区别是：

- 信号驱动 I/O 模式下，内核在操作可以被操作的时候通知给我们的应用程序发送 SIGIO 消息。
- 异步 I/O 模式下，内核在所有的操作都已经被内核操作结束之后才会通知我们的应用程序。

如下图，当我们进行一个 IO 操作的时候，我们传递给内核我们的文件描述符，我们的缓存区指针和缓存区的大小，一个偏移量 offset，以及在内核结束所有操作后和我们联系的方法。这种调用也是立即返回的，我们的程序不需要阻塞住来等待数据的就绪。我们可以要求系统内核在所有的操作结束后（包括从网络上读取信息，然后拷贝到我们提供给内核的缓存区中）给我们发一个消息。

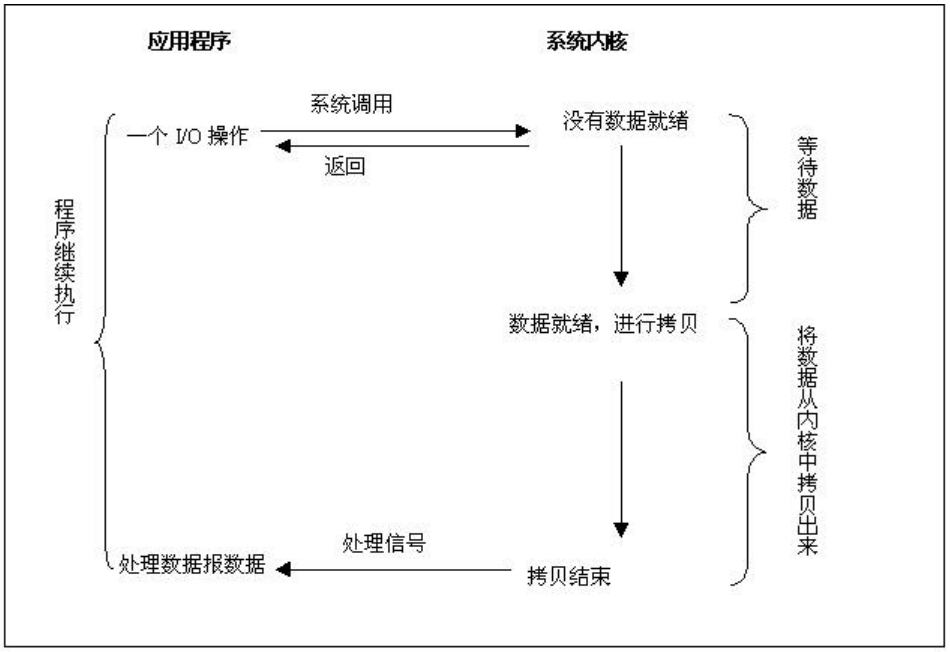

图 6-9 异步 I/O

6.10.6 几种 I/O 模式的比较

下面这个表格对这几种 I/O 模式进行了对比。

表 6-1 几种 I/O 模式的对比

阻塞模式	非阻塞模式	I/O 多路复用	信号驱动 I/O	异步 I/O	
初始化	检查	检查		初始化	等待数据
	检查	阻塞			
	检查				
	检查				
	检查				
	检查				
	检查				
	检查	就绪	信号通知		将数据从内核拷贝给用户程序
		初始化	初始化		
结束	结束	结束	结束	信号通知	

我们可以从中清楚的看出各个模式的差别，自己的程序可以挑选合适的模式来使用。

6.10.7 fcntl()函数

阻塞．你应该明白它的意思。简单的说，阻塞就是＂睡眠＂的同义词．你也许注意到

你运行上面的 listener 的时候，它只不过是简单的在那里等待接收数据。它调用 recvfrom()函数，但是那个时候（listener 调用 recvfrom()函数的时候），它并没有数据可以接收．所以 recvfrom()函数阻塞在那里（也就是程序停在 recvfrom()函数处睡大觉）直到有数据传过来。

很多函数都可以阻塞。像 accept()函数是阻塞的，所有以 recv 开头的函数也都是阻塞的。它们这样做的原因是他们需要这样做。

当你一开始建立一个套接字描述符的时候，系统内核就被设置为阻塞状态。如果你不想你的套接字描述符是处于阻塞状态的，那么你可以使用函数 fcntl()。

fcntl()函数声明如下：

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

int fcntl (int fd, int cmd, long arg）;
```

下面我们看一段程序片段：

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

sockfd = socket（AF_INET, SOCK_STREAM, 0）;
fcntl（sockfd, F_SETFL, O_NONBLOCK）;
……
……
```

这样将一个套接字设置为无阻塞模式后，你可以对套接字描述符进行有效的“检测”.如果你尝试从一个没有接收到任何数据的无阻塞模式的套接字描述符那里读取数据，那么读取函数会马上返回–1 代表发生错误，全局变量 errno 中的值为 EWOULDBLOCK。

一般来说，这种无阻塞模式在某些情况下不是一个好的选择。假如你的程序一直没有接收到传过来的数据，那么你的程序就会进行不停的循环来检查是否有数据到来，浪费了大量的 CPU 时间，而这些 CPU 时间本来可以做其他事情的。

另外一个比较好的检测套接字描述符的方法是调用 select()函数。

6.10.8　套接字选择项 select()函数

这个技术有一点点奇怪但是它对我们的程序确是非常有用的。

假想一下下面的情况：

你写的服务器程序想监听客户端的连接，但是你同时又想从你以前已经建立过的连接中来读取数据。你可能会说：“没有问题，我不就是需要使用一个 accept()函数和一对儿recv()函数吗？”。不要这么着急，你要想想，当你调用 accept()函数阻塞的时候，你还能调用 recv()函数吗？“使用非阻塞套接字！”你可能会这么说。是的，你可以。但是如果你又不想浪费宝贵的 CPU 时间，该怎么办呢？

Select()函数可以帮助你同时监视许多套接字。它会告诉你哪一个套接字已经可以读取数据，哪个套接字已经可以写入数据，甚至你可以知道哪个套接字出现了错误，如果你想知道的话。

下面是 select()函数的声明：

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

下面是 select()函数的参数说明：

- numfds 是 readfds，writefds，exceptfds 中 fd 集合中文件描述符中最大的数字加上 1。
- readfds 中的 fd 集合将由 select 来监视是否可以读取。
- writefds 中的 fds 集合将由 select 来监视是否可以写入。
- exceptfds 中的 fds 集合将由 select 来监视是否有例外发生。

如果你想知道是是否可以从标准输入和一些套接字（sockfd）中读取数据，你就可以把文件描述符和 sockfd 加入 readfds 中。numfds 的数值设成 readfds 中文件描述符中最大的那个加上一，也就是 sockfd+1（因为标准输入的文件描述符的值为 0 ，所以其他任何的文件描述符都会比标准输入的文件描述符大）。

当 select()函数返回的时候，readfds 将会被修改用来告诉你哪一个文件描述符你可以用来读取数据。使用 FD_ISSET() 宏，你可以选出 select()函数执行的结果。

在进行更深的操作前，我们来看一看怎样处理这些 fd_sets。下面这些宏可以是专门进行这类操作的：

- FD_ZERO（fd_set *set）将一个文件描述符集合清零
- FD_SET（int fd, fd_set *set）将文件描述符 fd 加入集合 set 中。
- FD_CLR（int fd, fd_set *set）将文件描述符 fd 从集合 set 中删除．
- FD_ISSET（int fd, fd_set *set）测试文件描述符 fd 是否存在于文件描述符 set 中．

那么，struct timeval 是什么呢？是这样的，一般来说，如果没有任何文件描述符满足你的要求，你的程序是不想永远等下去的．也许每隔 1 分钟你就想在屏幕上输出信息：“hello!”。这个代表时间的结构将允许你定义一个超时。在调用 select()函数中，如果时间超过 timeval 参数所代表的时间长度，而还没有文件描述符满足你的要求，那么 select()函数将回返回，允许你进行下面的操作。

这个 timeval 结构定义如下：

```
struct timeval
{
    int tv_sec ;        /* 秒数 */
    int tv_usec ;       /* 微秒 */
} ;
```

只需要将 tv_sec 设置为你想等待的秒数，然后设置 tv_usec 为想等待的微秒数（真正的时间就是 tv_sec 所表示的秒数加上 tv_usec 所表示的微秒数）．注意，是微秒（百万分之一）而不是毫秒．一秒有 1,000 毫秒，一毫秒有 1,000 微秒。所以，一秒有 1,000,000 微秒．

当 select()函数返回的时候，timeval 中的时间将会被设置为执行为 select()后还剩下的时间。

现在，我们拥有了一个以微秒为单位的记时器！但是因为 Linux 和 UNIX 一样，最小的时间片是 100 微秒，所以不管你将 tv_usec 设置的多小，实质上记时器的最小单位是 100 微秒．

另外需要注意的是：

- 如果你将 struct timeval 设置为 0，则 select()函数将会立即返回，同时返回在你的集合中的文件描述符的状态。
- 如果你将 timeout 这个参数设置为 NULL，则 select()函数进入阻塞状态，除了等待到文件描述符的状态变化，否则 select()函数不会返回。

下面这段代码演示了从标准输入等待输入等待 2.5 秒．

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

/* 标准输入的文件描述符数值 */
#define STDIN 0

main()
{
    struct timeval tv;
    fd_set readfds;

    /* 设置等待时间为 2 秒零 500,000 微秒 */
    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    /* 因为我们只想等待输入，所以将 writefds 和 execeptfds 设为 NULL */
    /* 程序将会在这里等待 2 秒零 500,000 微秒，除非在这段时间中标准输入有操作 */
    select(STDIN+1, &readfds, NULL, NULL, &tv);
    /* 测试 STDIN 是否在 readfds 集合中 */
    if (FD_ISSET(STDIN, &readfds))
    {
    /* 在，则在标准输入有输入 */
    printf(" A key was pressed!\n");
    }
    else
    {
    /* 不在，则在标准输入没有任何输入 */
```

```
        printf("Timed out.\n");
        }
}
```

在标准输入上，你需要输入回车后终端才会将输入的信息传给你的程序。所以如果你没有输入回车的话，程序会一直等待到超时。

对 select()函数需要注意的最后一点：如果你的套接字描述符正在通过 listen()函数侦听等待一个外来的网络连接，则你可以使用 select()函数（将套接字描述符加入 readfds 集合中）来测试是否存在一个未经处理的新连接。

上面是对 select()函数的一些简单介绍。

6.11 带外数据

许多传输层都支持带外数据（Out-Of-Band data），有时候也称为快速数据（Expedited Data）．之所以有带外数据的概念，是因为有时候在一个网络连接的终端想“快速”的告诉网络另一边的终端一些信息．这个“快速”的意思是我们的“提示”信息会在正常的网络数据（有时候称为带内数据 In-Band data）之前到达网络另一边的终端．这说明，带外数据拥有比一般数据高的优先级．但是不要以为带外数据是通过两条套接字连接来实现的．事实上，带外数据也是通过以有的连接来传输。

不幸的是，几乎每个传输层都有不同的带外数据的处理方法。我们下面研究的是 TCP 模型的带外数据，提供一个小小的例子来看看它是怎样处理套接字的带外数据，及调用套接字 API 的方法。

流套接字的抽象中包括了带外数据这一概念，带外数据是相连的每一对流套接字间一个逻辑上独立的传输通道。带外数据是独立于普通数据传送给用户的，这一抽象要求带外数据设备必须支持每一时刻至少一个带外数据消息被可靠地传送。这一消息可能包含至少一个字节；并且在任何时刻仅有一个带外数据信息等候发送。对于仅支持带内数据的通讯协议来说（例如紧急数据是与普通数据在同一序列中发送的），系统通常把紧急数据从普通数据中分离出来单独存放。这就允许用户可以在顺序接收紧急数据和非顺序接收紧急数据之间作出选择（非顺序接收时可以省去缓存重叠数据的麻烦）。在这种情况下，用户也可以“偷看一眼”紧急数据。

某一个应用程序也可能喜欢线内处理紧急数据，即把其作为普通数据流的一部分。这可以靠设置套接字选项中的 SO_OOBINLINE 来实现。在这种情况下，应用程序可能希望确定未读数据中的哪一些是“紧急”的（“紧急”这一术语通常应用于线内带外数据）。为了达到这个目的，在 Sockets 的实现中就要在数据流保留一个逻辑记号来指出带外数据从哪一点开始发送．

select()函数可以用于处理对带外数据到来的通知。

6.11.1 TCP 的带外数据

TCP 上没有真正意义上的“带外数据”。TCP 是由一种叫做“紧急模式”的方法来传输带外数据的。假设一个进程向一个 TCP 套接字写入了 N 个字节的数据，数据被 TCP 套接字的发送缓冲区缓存，等待被发送到网络上面．我们在图 6-10 可以看见数据的排列。

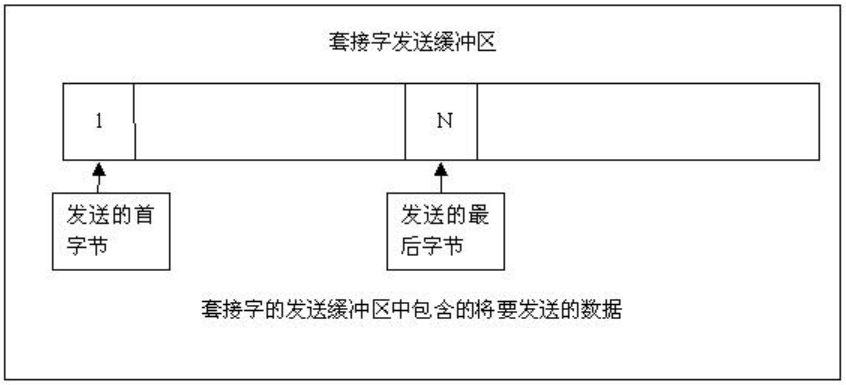

图 6-10　TCP **数据的排列**

现在进程使用以 MSG_OOB **为参数的** send()**函数写入一个单字节的"带外数据"，包含一个** ASCII **字符"**a**"：**

send（fd, “a”, 1, MSG_OOB）;

TCP **将数据放在下一个可用的发送缓冲区中，并设置这个连接的"紧急指针"（**urgent pointer**）指向下一个可用的缓冲区空间．图** 6-11 **表示了我们描述的这个状态，并将带外数据（**Out-Of-Band**）表示为"**OOB**"。**

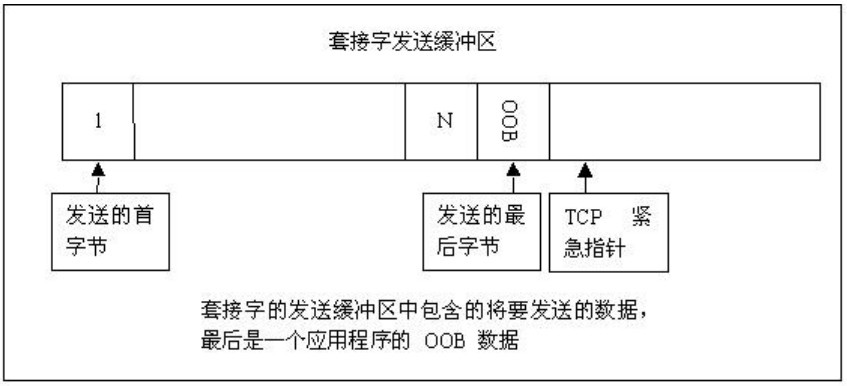

图 6-11　ODB **数据**

TCP **的紧急指针的指向的位置是在程序发送的** OOB **数据的后面。**

由图 6-11 **所表示的** TCP **套接字的状态，得知下一个将要发送的数据是** TCP **的** URG **（**Urgent pointer**）标志，发送完** URG **标志，**TCP **才会发送下面的带外数据的那个字节。但是** TCP **所一次发送的数据中，可能只包含了** TCP **的** URG **标志，却没有包含我们所发送的** OOB **数据．是否会发生这种情况而取决于** TCP **将要发送的数据队列中，在** OOB **数据之前的数据的多少。如果在一次发送中，**OOB **前的数据已经占满了名额，则** TCP **只会发送** URG **标志，不会发送** OOB **数据**

这是一个 TCP **紧急数据状态的重要特性：**TCP **的信息头指出发送者进入了紧急模式(比方说，在紧急偏移处设置了** URG **标志)，但是紧急偏移处的数据并没有必要一定要发送出去．事实上，如果一个** TCP **套接字的流传送停止后（可能是接收方的套接字的接收缓冲区没有空余空间)，为了发送带外数据，系统会发送不包含数据的** TCP **数据包，里面标明这是一个带外数据．这也是我们使用带外数据的一个有利点：**TCP **连接就算是在不能向对方**

发送数据的时候，也可以发送出一个带外数据的信号。

如果我们像下面这样发送一个多字节的带外数据：

```
send(fd, “abc”, 3, MSG_OOB);
```

在这个例子中， TCP 的紧急指针指向了数据最后一位的后面， 所以只有最后一位数据（“ c ”）才被系统认为是“ 带外数据 ”。

我们上面大致了解了发送方是怎样发送“ 带外数据 ”的了，下面我们来看一看接收方是怎样接收“ 带外数据 ”的。

1．当 TCP 收到一个包含 URG 标志的数据段时，TCP 会检查“ 紧急指针 ”来验证是否紧急指针所指的数据已经到达本地。也就是说，无论这次是否是 TCP 紧急模式从发送方到接收方的第一次传输带外数据。一般来说，TCP 传输数据会分成许多小的数据包来传输（每个包的到达时间也不同）。可能有好几个数据包中都包含紧急指针，但是这几个包中的紧急指针都是指向同一个位置的，也就是说多个紧急指针指向一个数据。需要注意的是，对于这一个带外数据，虽然有多个指针指向它，但是只有第一个紧急指针会通知程序注意。

2．接收进程收到另外一个带外数据的通知的条件是：有另外一个带外数据的指针到达．注意这里是“ 另外一个带外数据 ”的指针，不是上面的“ 一个带外数据 ”的另外一个指针。首先， SIGURG 信号回发送给套接字的属主，这个取决于是否已经使用 fcntl()函数或 ioctl()函数设定套接字的属主和这个程序对 SIGURG 信号的具体操作函数。其次，如果一个程序正阻塞与对这个套接字描述符的 select()函数的调用中，则 select()函数会产生一个例外，然后返回。

注意：进程收到带外数据的通知的时候，并不会在乎带外数据的真正数据是否到达。

3．当紧急指针所指的真正的带外数据通过 TCP 网络到达接收端的时候，数据或者被放入带外数据缓冲区或是只是简单的和普通的网络数据混合在一起。在缺省的条件下，SO_OOBINLINE 套接字选项是不会被设置的，所以这个单字节的带外数据并没有被防在套接字的接收缓存区中，而是被放入属于这个套接字的一个单独的带外数据缓存区中。如果这个进程想读取这个带外数据的具体内容的话，唯一的办法就是调用 recv，recvfrom，或是 recvmsg 函数，并且一定要指定 MSG_OOB 标志。

4．如果一个进程将套接字设置为 SO_OOBINLINE 属性，则由紧急指针所指的，代表带外数据的那个字节将回被放在正常套接字缓冲区中的最左边．在这种情况下，进程不能指定 MSG_OOB 来读取这个一个字节的带外数据，但是它可以知道带外数据的到达时间：通过检查套接字的带外数据标记．

有可能发生的一些错误：

5．如果当连接没有发送带外数据的时候进程来读取带外数据（比如说，通过 MSG_OOB 参数来接收函数），则 EINVAL 将会被返回。

6．当真正的带外数据到达之前的时候，进程被通知（SIGURG 或是 select 函数）有带外数据到达（也就是说带外数据的通知信号已经到达），如果进程尝试读取带外数据，则返回 EWOULDFBLOCK ．进程所能做的只是去读取套接字的接收缓存区．（也许，由于缓存区的数据以满，带外数据的那个字节信息无法传输过来，这样的话也许你需要清理一下接收缓存区来给带外数据空出一些空间）

7．如果进程尝试多次读取同一个带外数据，则 EINVAL 将会被返回。

8．如果进程将套接字属性设置为 SO_OOBINLINE ，然后尝试通过指定 MSG_OOB

标志来读取带外数据，则 EINVAL 将会被返回。

下面我们将前面的套接字例程做一些变动来测试带外数据的发送与接收.

6.11.2　OOB 传输套接字例程（服务器代码 Server.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

/* 服务器要监听的本地端口 */
#define MYPORT 4000

/* 能够同时接受多少没有 accept 的连接 */
#define BACKLOG 10

void
sig_urg（int signo）;

main()
{
    /* 在 sock_fd 上进行监听，new_fd 接受新的连接 */
    int sock_fd, new_fd ;
      /* 用于存储以前系统缺省的 SIGURL 处理器的变量 */ void * old_sig_urg_handle ;
    /* 自己的地址信息 */
    struct sockaddr_in my_addr;

    /* 连接者的地址信息*/
    struct sockaddr_in their_addr;

    int sin_size;
    int n ;
    char buff[100] ;

      /* 这里就是我们一直强调的错误检查．如果调用 socket() 出错，则返回 */
      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
```

```
/* 输出错误提示并退出 */
perror("socket");
exit(1);
}

/* 主机字节顺序 */
my_addr.sin_family = AF_INET;

/* 网络字节顺序，短整型 */
my_addr.sin_port = htons(MYPORT);

/* 将运行程序机器的 IP 填充入 s_addr */
my_addr.sin_addr.s_addr = INADDR_ANY;

/* 将此结构的其余空间清零 */
bzero(&(my_addr.sin_zero), 8);
/* 这里是我们一直强调的错误检查!! */ if (bind(sockfd, (struct sockaddr *)&my_addr,
sizeof(struct sockaddr)) == -1)
{
/* 如果调用 bind()失败，则给出错误提示，退出 */
perror("bind");
exit(1);
}

/* 这里是我们一直强调的错误检查!! */
if (listen(sockfd, BACKLOG) == -1)
{
/* 如果调用 listen 失败，则给出错误提示，退出 */
perror("listen");
exit(1);
}

/* 设置 SIGURG 的处理函数  sig_urg */
old_sig_urg_handle = signal（SIGURG, sig_urg）;

/* 更改 connfd 的属主 */
fcntl（sockfd, F_SETOWN, getpid()）;

while(1)
{
```

```
    /* 这里是主 accept()循环 */
    sin_size = sizeof(struct sockaddr_in);

      /* 这里是我们一直强调的错误检查!! */
      if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
        {
      /* 如果调用 accept()出现错误，则给出错误提示，进入下一个循环 */
    perror("accept");
    continue;
        }

    /* 服务器给出出现连接的信息 */
    printf（"server: got connection from %s\n", inet_ntoa(their_addr.sin_addr)）;
/* 这里将建立一个子进程来和刚刚建立的套接字进行通讯 */
    if (!fork())
    /* 这里是子进程 */
    while（1）
              {
      if（(n = recv（new_fd, buff, sizeof（buff）–1）) == 0）
                  {
    printf（"received EOF\n"）;
    break ;
                  }
    buff[n] = 0 ;
    printf（"Recv %d bytes: %s\n", n, buff）;
              }
    /* 关闭 new_fd 代表的这个套接字连接 */
    close(new_fd);
        }
    }
    /* 等待所有的子进程都退出 */
    while(waitpid(-1,NULL,WNOHANG) > 0);

    /* 恢复系统以前对 SIGURG 的处理器 */
    signal（SIGURG, old_sig_urg_handle）;
}

void
sig_urg（int signo）
{
```

```
    int n;
    char buff[100] ;

    printf（“SIGURG received\n”）;
    n = recv（new_fd, buff, sizeof（buff）– 1, MSG_OOB）;
    buff [ n ] = 0 ;
    printf（“recv %d OOB byte: %s\n” , n, buff）;
}
```

6.11.3 OOB传输套接字例程（客户端代码Client.c）

下面是客户端程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

/* 服务器程序监听的端口号 */
#define PORT 4000

/* 我们一次所能够接收的最大字节数 */
#define MAXDATASIZE 100

int
main(int argc, char *argv[])
{
    /* 套接字描述符 */
    int sockfd, numbytes;
    char buf[MAXDATASIZE];

    struct hostent *he;
    /* 连接者的主机信息 */
    struct sockaddr_in their_addr;

    /* 检查参数信息 */
    if (argc != 2)
    {
```

```
/* 如果没有参数，则给出使用方法后退出 */
fprintf(stderr,“usage: client hostname\n”);
exit(1);
}

/* 取得主机信息 */
if ((he=gethostbyname(argv[1])) == NULL)
/* 如果 gethostbyname()发生错误，则显示错误信息并退出 */
    herror(“gethostbyname”);
    exit(1);
}
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
/* 如果 socket()调用出现错误则显示错误信息并退出 */
perror(“socket”);
exit(1);
}

/* 主机字节顺序 */
their_addr.sin_family = AF_INET;

/* 网络字节顺序，短整型 */
their_addr.sin_port = htons(PORT);

their_addr.sin_addr = *((struct in_addr *)he->h_addr);
/* 将结构剩下的部分清零*/
bzero(&(their_addr.sin_zero), 8);

if（connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1）
{
/* 如果 connect()建立连接错误，则显示出错误信息，退出 */
perror(“connect”);
exit(1);
}

/* 这里就是我们说的错误检查！ */
if (send(new_fd, “123”, 3, 0) == -1)
{
/* 如果错误，则给出错误提示，然后关闭这个新连接，退出 */
perror(“send”);
close(new_fd);
```

```
        exit(0);
        }
printf（“Send 3 byte of normal data\n”）;
        /* 睡眠 1 秒 */
        sleep（1）;

          if (send(new_fd, “4”, 1, MSG_OOB）== -1)
        {
        perror(“send”);
        close(new_fd);
        exit(0);
        }
        printf（“Send 1 byte of OOB data\n”）;
        sleep（1）;

          if (send(new_fd, “56”, 2, 0) == -1)
        {
        perror(“send”);
        close(new_fd);
        exit(0);
        }
printf（“Send 2 bytes of normal data\n”）;
sleep（1）;
if (send(new_fd, “7”, 1, MSG_OOB）== -1)
        {
        perror(“send”);
        close(new_fd);
        exit(0);
        }
printf（“Send 1 byte of OOB data\n”）;
sleep（1）;
if (send(new_fd, “89”, 2, MSG_OOB）== -1)
        {
        perror(“send”);
        close(new_fd);
        exit(0);
        }
        printf（“Send 2 bytes of normal data\n”）;
        sleep（1）;
```

```
    close(sockfd);
    return 0;
}
```

6.11.4　编译例子

注意：你显然需要在运行 client **之前启动** server**．否则** client **会执行出错（显示**“Connection refused”**）．**

当只有一个连接的时候（因为这个服务器是多进程的，所以如果有多个连接同时存在可能会导致屏幕输出混乱），可以得到下面的结果：（注意是使用我们下面的客户程序来连接的，并且假设你运行我们的服务器程序是在本地机器上面）

```
root@bbs# gcc – o server server.c
root@bbs# gcc – o client client.c
root@bbs# ./server
root@bbs# ./client 127.0.0.1
Send 3 bytes of normal data          <- Client输出
Recv 3 bytes: 123                    <- Server输出
Send 1 byte of OOB data              <- Client输出
SIGURG received                      <- Server输出
Recv 1 OOB byte: 4                   <- Server输出
Send 2 bytes of normal data          <- Client输出
Recv 2 bytes: 56                     <- Server输出
Send 1 byte of OOB data              <- Client输出
SIGURG Received                      <- Server输出
Recv 1 OOB byte: 7                   <- Server输出
received EOF                         <- Server输出
```

这个结果正是我们想要的。每一个客户端发送的带外数据都导致服务器端产生了 SIGURG **信号，服务器端收到** SIGURG **信号后，就去读取带外数据了。**

6.12　使用 Inetd（Internet 超级服务器）

6.12.1　简介

利用 inetd **来做网路程序设计是个既简单又稳定的设计方法，您不需要考虑到复杂的** socket programming**。您的设计工作几乎在设计好通讯协定後就完成了，所需要的技巧，仅为简单的文字分析技巧。**

6.12.2　一个简单的 inetd 使用的服务器程序 hello inet service

首先，我们先来撰写一个称为 hello **的服务程序。**

hello.c

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

```
void main(void)
{
/* 作为一般的程序，这个 printf 将输出到标准输出 */
printf（“Welcome!\n Hello! World! \n”）;
}
```

这个程序很简单，不是吗?

编译。

```
root@bbs$ gcc -o hello hello.c
```

好了，现在我们已经拥有这个程序的可执行版本了。如果你直接执行这个程序的话，会出现下面这样的结果：

```
root@bbs#./hello
Welcome!
Hello! World!
root@bbs#
```

啊，程序将输出写出来了，我们成功了！注意！我们需要的是进行网络上的传输操作，现在我们做到的只能够在本地给你显示一些字符而已，我们需要的是能够传输到网络的另一端。

6.12.3 /etc/services 和 /etc/inetd.conf 文件

我们必须通过设置系统的两个文件：/etc/services 和/etc/inetd.conf 来对系统进行配置，从而将我们的 hello 程序变成网络可访问的。

下面我们来看看如何设定/etc/services 及/etc/inetd.conf。

在我们更改系统的/etc/services 文件前我们先来做一个测试，以便帮助各位读者能够对它的作用有更清楚的理解。

我们在本地机器上输入下面的命令：

```
root@bbs# telnet localhost hello
hello: bad port number
```

注意系统给出的错误信息：“ hello: bad port number”。因为 telnet 命令的第二个参数应该是想登陆系统的端口，我们给出的 hello，系统不知道 hello 是什么端口，所以它说：“ 错误的端口数字 ”。

那么下面我们这样做：

在/etc/services 中加入以下这一行

```
hello           20001/tcp
```

其意义为 hello 这项服务是在 port 20001、是一个 TCP 连接。

当我们进行到这一步的时候，你可以再试试进行刚才给出“ hello: bad port number ”错误信息的操作：

```
root@bbs# telnet lcoalhost hello
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
```

信息变了： Unable to connect to remote host: Connection refused.

这说明系统已经知道了 hello 代表的是哪个端口（以为我们上面在 /etc/services 文件中指定了 hello 是一个 tcp 连接，在 20001 端口），但是系统无法和 hello 端口建立连接，因为没有任何程序在监听 20001 端口来等待连接。

OK，现在我们已经告诉了系统我们的 hello 程序使用什么端口了，可是当我们连接 hello 的端口的时候系统还没有将我们的程序执行．下面：

在 /etc/inetd.conf 中加入以下这一行 ：

goodie stream tcp nowait root /full_goodie_path_name/goodie

各个参数的意义为

<service_name><sock_type><proto><flags><user><server_path><args>

- service_name 是需要在系统服务中存在的名称。
- sock_type 有很多种，大多用的是 stream/dgram。
- proto 一般用 tcp/udp。
- flags 有 wait/nowait。
- user 是您指定该程序要以那一个使用者来启动，这个例子中用的是 root，如果有安全性的考量，应该要改用 nobody。一般来说，建议您用低权限的使用者，除非必要，不开放 root 使用权。
- server_path 及 args，这是您的服务程序的位置及您所想加入的参数。

接下来重新启动 inetd

root@bbs# killall inetd

root@bbs# inetd

root@bbs#

这样我们便建立起一个 port 20001 的 hello service。

现在我们来检验一下 goodie 是否可以执行:

telnet localhost 20001

或

telnet your_host_name 20001

或

telnet locahost hello

执行结果如下：

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^）'.

Welcome!

Hello! World!

root@bbs#

Yahoo!! 我们现在连接成功了!! 原来一个简单的，只能显示两行欢迎信息的，没有涉及到任何网络连接的小程序，现在可以进行网络服务了！很神奇吧！

6.12.4　一个复杂一些的 inetd 服务器程序

很简单不是吗? 信不信由您，telnet/pop3/imap/ftp 都是靠这种方式建立起来的服务。 当

然， telnet/pop3/imap/ftp 各项服务都有复杂的命令处理过程，我们现在这个程序只能显示欢迎信息，但是至少我们已经可以让它做网络上的一个服务程序了。下面我们现在来建立一点小小的“网络协定”，这个协定使我们可以输入“exit”时，离开程序，而其他的指令都是输出与输入相同的字串。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
 {
    /* 网络接受缓存区 */
    char buf[1024];

    /* 是否接收到了 exit 字符串的标志 */
    int ok;

    /* 输出欢迎信息 */
    printf("Welcome! This is hello service!\n");

    /* 因为 Linux/UNIX 系统具有缓存作用， fflush 函数将缓存中的数据立即送出，
    防止网络连线的另外一边无法接收到少量的数据 */
    fflush(stdout);
    /* 初始化 OK, 设置为没有接收到 exit */
    ok=0;

    do
/* 如果标准输入没有数据输入，则程序在此处循环等待 */
       while (fgets(buf,1023,stdin)==NULL);
/* 检查当前的输入是否为 "exit" */ if (strncasecmp(buf,"exit",4)==0)
         {
    /* 设置标志位 */
    ok=1;
         }

    /* 将接收到的字符串原样送出 */
    printf(buf);
/* 将缓存区中的数据立即发送 */ fflush(stdout);
    } while（!ok）;
}
```

因为 inetd 将网络的输入作为程序的标准输入，而将程序的输出作为程序的网络输出，

所以程序中的 stdin 相当于对网络套接字进行读而 stdout 相当于是对网络套接字进行 send 操作。

执行：

telnet localhost hello

或

telnet your_host_name 20001

运行结果如下：

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^）'.
Welcome! This is hello service!

输入"help"

help
help

输入"exit"

exit
exit
Connection closed by foreign host.

6.12.5　一个更加复杂的 inetd 服务器程序

我们现在已经可以简单的处理网络远程发送过来的命令了，而我们程序所做的处理只是对 stdin 和 stdout 进行操作！下面，我们将设计一个稍微复杂一点点的通讯协定，比较通用于一般用途。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* 所支持的命令的数组 */
char *cmds[]={ "help", "say", "hello", "bye", "exit", NULL};
/* 判断一个字符串是命令数组中的第几个命令，如果不存在则返回 – 1 */
int getcmd(char *cmd)
{
    int n=0;
    while (cmds[n]!=NULL)
    {
    if (strncasecmp(cmd,cmds[n],strlen(cmds[n]))==0)
    return n;
    n++;
}
  return -1;
```

```
}

   /* 主函数 */
   void main(void)
{
   /* 接收的命令的缓存区 */
   char buf[1024];

   /* 是否为退出命令的标志 */
   int ok;

   /* 输出欢迎信息 */
   printf("Welcome! This is hello service!\n");

   /* 清除标准输出的缓存 */
   fflush(stdout);

   /* 初始设置 ok 为没有收到退出命令 */
   ok=0;

   /* 程序主循环体 */
   do
   /* 如果程序没有接收到输入则循环等待 */
   while (fgets(buf,1023,stdin)==NULL) ;

   /* 收到数据后进行命令判断 */
   switch (getcmd(buf))
   case -1: printf("Unknown command!\n"); break;
   case  0: printf("How may I help you, sir?\n"); break;
   case  1: printf("I will say %s",&buf[3]); break;
   case  2: printf("How're you doing today?\n"); break;
   case  3: printf("Si ya, mate!\n");  ok=1; break;
   case  4: printf("Go ahead!\n"); ok=1; break;

     }

   /* 清空输出缓冲区 */
   fflush(stdout);
   } while （!ok）;
}
```

运行：

```
telnet localhost hello
```

或

```
telnet your_host_name 2001
```

试试看输入“help”、“say”、“hello”、“bye”、“exit”等等指令，及其它一些不在命令列中的指令。

好了，现在我们知道了，Inetd 就是将我们写的使用标准输出和标准输入的程序转变成网络程序，这样可以大大的简化我们的编程，避免了和什么 socket()、recv()、send() 函数打交道。

6.12.6　程序必须遵守的安全性准则

注意：在设计 inetd 服务程序时，要特别注意 buffer overflow（缓存区溢出）的问题，也就是以下这种状况:

```
char buffer_overflow[64];
fscanf(stdin,“%s”,buffer_overflow);
```

历来几乎所有的安全漏洞都是由此而来的。

你一定不可这样用，不论任何理由，类同的用法也不可以。黑客高手可以透过将您的 buffer 塞爆，然後塞进他自己的程序进来执行。

6.12.7　小结

通过 Linux 系统提供的 inetd 服务，我们可以方便的编写网络程序而从来不用去在乎那些看起来高深难懂的套接字函数．你所需要做的只是写一个普通的读写标准输入输出的程序，然后去配置一下系统 inetd 的配置文件：/etc/services 文件和 /etc/inetd.conf 文件。

6.13　本章总结

BSD UNIX 引入了作为一种机制的套接字抽象，它允许应用程序于操作系统的协议软件接口。由于许多厂商采纳了套接字，套接字接口已经成了一种事实上的标准。

一个程序调用 socket 函数创建一个套接字描述符。Socket 调用的参数指明了所使用的协议和所要求的服务器．所有的 TCP/IP 协议都是 Internet 协议族的一部分．系统为套接字创建了一个内部的数据结构，并把协议族域填上，系统还使用服务类型参数来选择某个指定的协议（常常是 UDP 或 TCP）。

其他的系统调用允许应用程序指明一个本地地址（bind），强迫套接字进入被动模式以便为某个服务器使用(listen),或强迫插口进入主动模式以便为某个客户机使用(connect).服务器可以进一步使用 accept 调用以获得入连接请求（accept），客户机和服务器都可以发送或接收数据（read 或 write)。最后，在结束某个插口后，客户机和服务器都可以撤消该接口（close)。

套接字有五种 I/O 模式：阻塞模式／非阻塞模式／IO 多路复用／信号驱动 IO／异步 IO。

带外数据是一种可以快速的通知网络的另一端计算机信息的一种方法．带外数据甚至可以只告诉远程计算机它的存在而不必将它的具体数据传输过去．带外数据并不是建立两

个连接来传送数据（至少在 TCP 中不是这样），它是将所谓的＂带外数据＂影射到已经存在的套接字连接中。

很少有人想写网络应用程序，因此进程的细节最好留给那些想写的人．实践和查阅大量的例子程序是开始写网络代码的最好的方法．但是要掌握这门技术却要花许多年时间。

第七章　网络安全性

7.1　网络安全简介

连接网络的主机，特别是连接因特网的主机，比没有连接入网络的主机会暴露出更多的安全问题。安全问题。网络安全性高可以降低连接网络的风险，但就其性质而言，网络访问和计算机安全性是矛盾的。网络是一条数据高速公路，它专门用来增加对计算机系统的访问，而安全性却需要控制访问。提供网络安全性是在公开访问与控制访问之间的一种折中。

我们可以将网络想象成高速公路，它就像高速公路一样为所有的访问者——无论是受欢应的访问者还是不受欢应的黑客（hacker，在网络上非法入侵别人机器的人）——提供相等的访问权。通常，我们是通过锁门来为财产提供安全性的，而不是封锁接到。同样，网络安全性一般是指对单台主机提供何时的安全性，而不是直接在网络上提供安全性。

在很多小城镇中，人们相互之间都互相认识，因而房门往往是不锁的。但在大城市中，房门安装了笨重的门闩和链条。在短短的十几年中，因特网已从一个只有数十个用户的“小镇”发展到具有数百万用户的“大城市”。就像大城市使邻居之间变得陌生一样，因特网的飞速发展减少了网络邻居之间的信任度。对计算机安全性要求的增长是一个负效应，但这种发展并不是一件坏事情。一个大城市可以提供更大的选择余地和更多的服务，同样，扩展后的网络也可以提供日益增加的服务。对大多数人来说，安全只是访问网络时需要考虑的一小部分。

随着网络的发展，其越来越社会化，网络的非法入侵事件也有增无减。但是，这些入侵的十几程度常常被大大的夸大了。对侵入迹象的反应过度会阻碍对网络的正常利用，因此一定要对症下药。有关网络安全性的最好建议是尊重常规，在 RFC1233 中很好的阐述了这一原则：

“尊重常规是用来确定安全策略的最何时的准则。精心设计的安全性方案和机制，诚然是令人佩服的，也确实可以充分发挥作用，但应兼顾控制的简单性，即实施其方案时，在经济和时间上的投资也必须予以充分考虑”。

7.1.1　网络安全的重要性

1996 年初，五角大楼宣布其计算机系统在 1995 年遭到 215 万次非法入侵。更令人不安的是大多数非法入侵未被察觉。这些非法入侵给国家安全带来的影响程度还未确定，但是已发现的非法入侵多数是针对计算机系统所存放的敏感信息，其中有三分之二的非法入侵是成功的，黑客盗窃、修改或破坏了系统上的数据。

同时，我们也应该以严肃的态度来考虑企业间谍无处不在这个问题。有很多公司并没有充分的准备来对付非法入侵者，甚至没有意识到它们的存在，有些公司还没有看到这些威胁对它们的影响。事实上，计算机在线社会是人类生存的真实社会的仿制品，在电子空间中有许多不道德的人不断侵犯各种计算机系统上的安全系统，还有一些并非不道德的人出于某种心理在电子空间徘徊并和用户开一些玩笑来嘲笑用户的安全性。

事实上，网络从其出现的一开始就受到黑客们的攻击，下面是历史上著名的几次网络事故：

- 1988 年 9 月，著名的蠕虫程序（worm）被传播到因特网上，造成因特网的瘫痪，使许多网络不能连接到因特网上。
- 1991 年 5 月，在 Biscay 海湾发生了一起由于网络被非法入侵而造成的沉船事故。这是由于欧洲气象预报中心的计算机系统被网络黑客侵入，造成气象预报卫星不能正常工作，导致一场暴风雨的预报失误而酿成的悲剧。
- 1993 年 6 月，美国一家医院连接到网络上的一些化验报告，其数据被黑客侵入后将阴性改为阳性，许多被化验者因此误认为自己患上了癌症。

在过去的十几年中，网络黑客们一直在通过计算机的漏洞对计算机系统进行攻击，而且这种攻击方法变得越来越复杂。

1988 年，大部分入侵者的方法仅是靠猜测口令，利用系统的配置不当、以及系统上软件本身的漏洞。到了 1998 年，这些方法仍被使用，但又增加了新的手段。有些入侵者甚至通过读取操作系统源代码的方法来获取系统的漏洞，并以此展开攻击。一些网络黑客编写的攻击站点的工具软件，在因特网上也可以容易的得到，这就给网络安全带来了更严峻的挑战。

正是由于这些在线犯罪，FBI 的国家计算机犯罪小组建议采用防火墙作为防止计算机犯罪的措施。虽然 NCCS 还有其它安全措施，但是普遍认为采用防火墙是防止 Intranet 被入侵的最好方法。

其他被推荐的措施有：

- 对非法访问的登录进行横幅警告。显示在登录屏幕上，警告那些非法用户，它们正在进入一个受保护的系统。
- 键盘级监控。这个安全措施十几上是捕捉在某台工作站上的键盘敲击并记录在文件中。这些文件经管理员查看后确定是否有非正常方式，如从一台工作站上出现多次登录失败等。
- 捕捉、跟踪来自电话公司的服务。这种服务允许用户的公司截取进来的电话并跟踪到出发点，并找出和排除非法的远程访问。
- 呼叫者 ID。这种服务（现在家庭用户中也很流行）识别进入系统的号码和姓名，能够在一开始就识别并排除非法的远程访问。
- 电话截取。这种服务（与呼叫者 ID 有关）允许某些电话号码被阻止接通到电话系统。例如。假设调制解调器（远程访问系统）接收到一个来自某大学宿舍的一个数字电话，可以截取这个电话号码（可以自己打个电话到学校机关）。
- 数据加密措施。可以用加密和解密软件来“搅乱”数据，使多数黑客没有兴趣登录到系统。
- 防火墙。所有这些安全措施中，防火墙是最具有防护性的。因为它可以根据需要设计并安装在系统中的最重要部位。如果将防火墙与其他一些安全措施相结合，会得到一个对公司的数据和 Intranet 最佳的安全措施。

7.1.2 信息系统安全的脆弱性

黑客攻击网络已有十几年的历史。它们的攻击方法包括：猎取访问路线、猎取口令、强行闯入、改变与建立 UAF（用户授权文件）记录，偷取额外特权、引入“特洛伊木马”软件掩盖其真实企图、引入命令过程或程序“蠕虫”把自己几声在特权用户上、清理磁盘、使用一个节点作为网关（代理）到其他节点上以及通过隐蔽信道进行非法活动等。

黑客使用很多工具进行攻击，例如 Rootkit 工具、特洛伊木马和轨迹跟踪等。SATAN 工具专门用来查找和分析一个网络安全的薄弱环节。Aita Vista 工具可以用来查找一个网络的脆弱部分或者黑客敏感的信息。

在因特网上有名的黑客网址有：

www.cert.org
www.10pht.com
www.root.org/warez.html
www.2600.com
www.microagwny.com/home/claw/hackers.html
www.halcyon.com/yakboy/42ff.html
www.netwalk.com/silicon/void-f.html

黑客在网络上经常采用的攻击手段是：利用 UNIX 操作系统提供的 telnet、ftp 和 remote exec 守护进程等缺省帐户进行攻击。另外，黑客还可以采用 UNIX 操作系统提供的命令 finger 与 rusers 等收集的信息不断提高自己的攻击能力；利用 sendmail 漏洞，采用 debug 工具、wizard 工具、pipe 机制、假名及 Ident 守护进程进行攻击；采用 ftp 的匿名访问进行攻击；利用 NFS 的漏洞进行攻击；利用 CGI 程序的漏洞进行攻击；通过 rsh、rlogin、rexec 守护进程以及 X Windows 等方法进行攻击。

网络黑客之所以能够得逞，是因为信息系统本身存在一些安全方面的脆弱性。

下面是信息系统在安全方面存在的一些问题：

1．操作系统安全的脆弱性

- 操作系统的体系结构造成操作系统本身的不安全，这是计算机系统不安全的根本原因。操作系统的程序可以动态链接，包括 I/O 的驱动程序与系统服务，都可以用打补丁的方式进行动态链接。Linux 操作系统的许多版本升级开发都是采用打补丁的方式进行的。这种方法厂商可以使用，黑客同样也可以使用。这种动态链接是计算机病毒产生的好环境。一个靠升级与打补丁开发的操作系统不可能从根本上解决安全问题，但操作系统支持程序动态链接与数据动态交换又是现代系统继承和系统扩展的必备功能，因此这是互相矛盾的。
- 操作系统支持在网络上传输文件，包括可执行的映象文件，即在网络上加载程序。这样也破坏了系统的安全性。
- 操作系统不安全的另一个原因在于它可以创建进程，甚至支持在网络的节点上创建和激活远程进程，更重要的是被创建的进程可以继承创建进程的权限。这一点与上一点（可以在网络上加载程序）结合起来就构成了可以在远端服务器上安装“间谍”软件的条件。若再加上把这种间谍软件以补丁的方式“打”在一个合法的用户上，尤其“打”在一个特权用户上，间谍软件就可以使系统进程与作业的监视程序都检测不到它的存在。
- 操作系统通常都提供守护进程。这种软件实质上是一些系统简称，它们总在等待一些条件的出现。一旦有满足需要的条件出现，程序便继续运行下去。这样的软件都是黑客可以利用的。这里应该说明的是：关键不在于有没有守护进程，而在于这种守护进程在 UNIX、Windows NT 操作系统上是否具有与操作系统核心层软件同等的权力。
- 操作系统提供远程过程调用(RPC)服务，RPC 服务本身也存在一些可以被非法用户利用的漏洞。
- 操作系统提供 debug（调试器）程序。许多研制系统软件的人员，它们的基本技能就是开发补丁程序和系统调试器。掌握了这两种技术，它们就有条件从事黑客可以从事的事情。
- 操作系统安排的无口令入口是为系统开发人员提供的便捷入口，但它也是黑客的通道。另外，有些操作系统还有隐蔽的通道。

2．计算机网络安全的脆弱性

Iternet/Intranet 使用的 TCP/IP 协议以及 FTP、E-mail、RPC 和 NFS 等都包含许多不安全的因素，存在许多漏洞。

许多人都知道，1988 年 Robert Morries 在 VAX 机上用 C 语言编写了一个通过 GUESS 软件，根据搜索的用户名字来猜测机器密码口令的程序。结果是该程序自 1988 年 11 月开始在网络上传播依赖，几乎每年都给因特网上的系统造成一亿美元的损失。

黑客通常采用源端口、源路由、SOCKS、TCP 序列预测或者使用远程过程访问(RPC)进行直接扫描等方法对防火墙进行攻击。

3．数据库管理系统安全的脆弱性

数据库管理系统的安全必须与操作系统的安全相匹配。由于数据库的安全管理同样是建立在分级管理的基础之上的，因此 DBMS 的安全也是脆弱的。

4．缺少安全管理

世界上现有的信息系统绝大多数都缺少安全管理员，缺少信息系统安全管理的技术规范，缺少定期的安全测试与检查，更缺少安全监控。

我国许多企业的信息系统已经使用了许多年，但计算机的系统管理员与用户的注册还出于缺省配置的状态。

从某种意义上讲，缺少安全管理是造成系统不安全的最直接因素。因此，如果要提高整个系统的安全性，首要的任务就是找一位专门管理系统的人员，由他来维护和监督系统的安全。

7.2 Linux 网络不安全的因素

大部分计算机安全问题是由于管理不当，而不是由于系统软件的漏洞。严谨的管理将会大大减少系统被非法入侵的可能性。

下面是一些常见的安全漏洞。

1．特权软件的安全漏洞

- IFS

一种攻击的方法是通过 IFS(Input Files Separator，输入字段分隔符) shell 变量来实现的。该变量用于决定传给 shell 字符串的分隔符。例如，一个程序调用函数 system()或 popen()执行一个 shell 命令，那么该命令首先由 shell 来分析，如果执行的用户可以控制 IFS 环境变量，就可能会导致不可预测的结果。

这是一个典型例子：如果程序执行如下的代码：

```
system("/bin/ls –l");
```

如果 IFS 变量被设置为包含"/"字符，而一个恶意的程序被命名为 bin 并且放在用户的 path 变量那，则该命令会被解释成 bin ls –l，它执行程序 bin 并带有两个参数 ls 和-l。为了防止这种情况的发生，一个程序应该尽量不要使用 system()、popen()、execlp()、和 execvp()这些依赖外部环境的函数来运行其他程序，而应该使用 execle()这样自行传递环境变量的函数。

下面是一个设置 IFS 变量的命令行指令：

```
%setenv IFS /        #C shell

$IFS=/               #B shell
$export $IFS
HOME
```

另一个攻击方法是通过使用 HOME 环境变量。

通常，csh 和 ksh 在路径名称中用字符"~"来代替 HOME 变量。因此，如果一个入侵者能改变 HOME 变量的值，就能利用一个使用字符"~"作为 HOME 命令的 shell 文件来达到

目的。

例如，如果一个 shell 文件用~/。rhosts 或者$HOME/。rhosts 指向用户指定的文件，就有可能被人通过在执行命令前重新设置 HOME 环境变量的方法来进行破坏活动。

- PATH

使用 PATH 攻击方法的特征是，利用 PATH 环境变量中文件路径的值和顺序。不合理的路径顺序会导致意外的结果——如果执行的命令不是以绝对路径的方式执行。例如，用下面的方式指定 PATH：

PATH=。/：/usr/bin：/bin：/sbin

如果有人在当前路径创建了一个名为 ls 的文件，它就会比/bin/ls 先执行。如果该文件包括如下内容：

```
#! /bin/sh
/bin/cp /bin/sh /tmp/。secret 2>/dev/null
rm –f $0
exec /bin/ls “$1”
```

该代码会暗中创建一个/bin/sh 的拷贝，它在执行时会以执行该文件的用户的身份被执行。此外，它消除证据，使用户在执行该命令时感觉不到是在执行另外一个程序。

- 缓冲区溢出

由于不好的编程习惯，也会导致软件本身的安全漏洞。在这个方面最典型的例子是 Morris 蠕虫攻击程序。该漏洞是由于系统调用 gets()(系统调用 fgets()没有这个问题)时不检查参数的长度而造成的。这使得在用户的控制下缓冲区会溢出。因此，在正常情况下，程序在遇到缓冲区溢出时会停止运行。而如果用户对操作系统很熟悉，就可以使程序在遇到缓冲区溢出时转而运行另一个程序。

设想如下情况，一个特权程序（suid）在遇到缓冲区溢出后转而执行一个 shell，那么这时的普通用户就变成得来 root 用户。因此，缓冲区溢出问题是非常严重的。据统计，有 70%以上的成功的非法入侵是通过缓冲区溢出来实现的。另外还有几个系统调用也存在同样的漏洞，如 scanf()、sscanf()、fscanf()和 sprintf()。

- umask 值

一个程序员经常犯的错误是，umask（缺省的文件保护掩码）的设置不正确。许多程序没有检查 umask(的值，而且经常忘记指定新建文件的保护掩码值。程序创建了一个新文件，却忘记改变其保护模式而使之安全。黑客们经常可以利用这一点，更改可写的文件从而获取特权。

因此，在创建一个新文件之前，一定要先用 Linux 提供的系统调用 umask()设定一个正确的 umask 值。

- 状态返回值

另一个程序员经常犯的错误是程序不检查每个系统调用的返回值。在程序设计时程序员往往会认为某些系统调用总是正确执行的。但是，如果一个黑客可以控制他的运行环境，那么他就可以设法使一个在程序中被认为是永远正确的系统调用产生错误。这会使用户的程序产生不可预测的结果。

- 捕捉信号

一般，程序员编写的程序不捕捉它可以接收的信号，这一点也常常被黑客们所利用。

例如，一个黑客将其 umask 设置为一个适当的值，然后向一个没有正确捕捉信号的特权程序发送 SIGQUIT 信号——该信号导致该程序产生 core 文件。此时，该 core 文件的所有者是执行该程序的 UID，但是它的保护掩码是 umask 设定的，黑客可以读取这个文件。黑客就可以进一步利用 gdb 等工具来读这个 core 文件，从而可能获取系统上的重要信息。

- 队列边界检查

这个漏洞与前面讲述的“缓冲区溢出”问题是密切相关的。

典型的情况是，程序中使用了固定长度的数组变量，而在执行时却不检查其中的数据是否超过数组边界，从而导致程序出现不可预测的结果。

2．特洛伊木马程序

特洛伊木马程序与一般用户想要执行的程序从外观上（如文件名）看很相似，例如编辑器、登录程序或者游戏程序。这种程序与一般用户想要执行的程序表面上很相似，但是却完成其他的操作，例如删除文件、窃取密码和格式化磁盘等。等到用户发现，却为时已晚。

特洛伊木马可以出现在很多地方。它们可以出现在被编译过的程序中，也可以出现在又系统管理员执行的系统命令文件中。有的特洛伊木马程序还可作为消息（例如电子邮件或发给终端的消息）的一部分发送。一些邮件头（mail headers）允许用户退到外壳(shell)并执行命令，该特性能在邮件被阅读的时候激活。给终端发送特定的消息能在终端上存储一个命令序列，然后该命令序列被执行，就好像在终端上直接从键盘输入一样。编辑器初始化文件（如 vi 对应的.exrc 文件）也是经常出现特洛伊木马的地方。

特洛伊木马程序非常普遍。黑客们经常以多种方式改变系统，以便在最初的攻击活动被发现后，还可以进入系统。这也使一个被攻破的系统很难恢复。因此为了找出特洛伊木马，有必要搜索整个系统。

3．网络监听及数据截取

计算机网络安全中的一个重大问题是计算机之间传输的数据可以很容易的被截取。在过去大型主机的时代，这不成为威胁，因为在那种系统上数据的传输处于系统的控制之下。但由于异种机的互联，敏感数据的传输会处于系统的控制之外，有许多现成的软件可以监视网络上传输的数据。尤其脆弱的是总线型网络（例如以太网），这种网络上发送给每个特定机器的数据都可以被网络上的任何机器截取到。

这意味着任何数据都可以被截取并用于不同的目的，不仅仅包括敏感数据，还有信息交换（例如登录顺序，包括口令）。数据截取并不一定要从网络本身截取。通过在网络软件上或应用程序上安装特洛伊木马，就能截取数据并保存到磁盘上以备后用。

4．软件之间的相互作用和设置

计算机安全受到威胁的根本原因是计算机系统的运行的软件日益增长的复杂性。任何一个人都不可能编写整个系统，因此也无法预测系统那每个部分之间的相互作用。一个例子是/bin/login 的一个问题，它接收其他一些程序的非法参数，从而可以使普通用户成为超级用户。

5．研究源代码的漏洞

许多入侵者是通过研究一些程序的源代码而成功的攻击系统的。通常，这种源代码可以免费下载得到，这使得许多人可以研究它，并找出里面的潜在漏洞。过去这种攻击方法还很少见，而如今却非常普遍。从某种意义上讲，这也是一种好事。因为它可以促进程序员改正软件错误，也使人们了解一些软件编程的方法。

7.3 Linux 程序员安全

Linux 系统为程序员提供了许多子程序，这些子程序可存取各种安全属性。有些是信息子程序，返回文件属性、实际的和有效的 UID、GID 等信息。有些子程序可改变文件属、UID、GID 等；有些处理口令文件和小组文件，还有些完成加密和解密。

本节主要讨论有关系统子程序，标准 C 函数库库子程序的安全，如何写安全的 C 程序

并从 root 的角度介绍程序设计（仅能被 root 调用的子程序）

7.3.1 系统子程序

下面是和网络安全有关的一些系统子程序。

1．I/O 子程序

（1）creat()：建立一个新文件或重写一个暂存文件。

需要两个参数：文件名和存取许可值（8 进方式）。如：

```
/* 建立存取许可方式为 0666 的文件 */
creat("/usr/pat/read_write"，0666)
```

调用此子程序的进程必须要有建立的文件的所在目录的写和执行许可，置给 creat()的许可方式变量将被 umask()设置的文件建立屏蔽值所修改，新文件的所有者和小组由有效的 UID 和 GID 决定。

返回值为新建文件的文件描述符。

（2）fstat()：见后面的 stat()

（3）open()：在 C 程序内部打开文件。

需要两个参数：文件路径名和打开方式(I，O，I&O)。

如果调用此子程序的进程没有对于要打开的文件的正确存取许可（包括文件路径上所有目录分量的搜索许可），将会引起执行失败。如果此子程序被调用去打开不存在的文件，除非设置了 O_CREAT 标志，调用将不成功。此时，新文件的存取许可作为第三个参数（可被用户的 umask 修改）。

当文件被进程打开后再改变该文件或该文件所在目录的存取许可，不影响对该文件的 I/O 操作。

（4）read()：从已由 open()打开并用作输入的文件中读信息。

它并不关心该文件的存取许可。一旦文件作为输入打开，即可从该文件中读取信息。

（5）write()：输出信息到已由 open()打开并用作输出的文件中。

同 read()一样，它也不关心该文件的存取许可。

2．进程控制

（1）exec()族：包括 execl()，execv()，execle()，execve()，execlp()和 execvp()。

可将一可执行模快拷贝到调用进程占有的存贮空间。正被调用进程执行的程序将不复存在，新程序取代其位置。

这是 UNIX 系统中一个程序被执行的唯一方式：用将执行的程序复盖原有的程序。

安全注意事项：

- 实际的和有效的 UID 和 GID 被传递给由 exec()调入的不具有 SUID 和 SGID 许可的程序。
- 如果由 exec()调入的程序有 SUID 和 SGID 许可，则有效的 UID 和 GID 将设置给该程序的所有者或小组。
- 文件建立屏蔽值将传递给新程序。
- 除设了对 exec()关闭标志的文件外，所有打开的文件都传递给新程序。用 fcntl()子程序可设置对 exec()的关闭标志。

（2）fork()：用来建立新进程。其建立的子进程是与调用 fork()的进程（父进程）完全相同的拷贝（除了进程号外）

安全注意事项：

- 子进程将继承父进程的实际和有效的 UID 和 GID。
- 子进程继承文件方式建立屏蔽值。

- 所有打开的文件传给子进程。

（3）signal()：允许进程处理可能发生的意外事件和中断。

需要两个参数：信号编号和信号发生时要调用的子程序。

信号编号定义在 signal.h 中。

信号发生时要调用的子程序可由用户编写，也可用系统给的值，如：SIG_IGN，则信号将被忽略，SIG_DFL 则信号将按系统的缺省方式处理。

如许多与安全有关的程序禁止终端发中断信息（BREAK 和 DELETE），以免自己被用户终端终止运行。

有些信号使 UNIX 系统的产生进程的核心转储（进程接收到信号时所占内存的内容，有时含有重要信息），此系统子程序可用于禁止核心转储。

3．文件属性

（1）access()：检测指定文件的存取能力是否符合指定的存取类型。

需要两个参数：文件名和要检测的存取类型（整数）。

存取类型定义如表 7-1：

表 7-1 acess()存取类型定义

0	检查文件是否存在
1	检查是否可执行（搜索）
2	检查是否可写
3	检查是否可写和执行
4	检查是否可读
5	检查是否可读和执行
6	检查是否可读可写可执行

这些数字的意义和 chmod 命令中规定许可方式的数字意义相同。

此子程序使用实际的 UID 和 GID 检测文件的存取能力(一般有效的 UID 和 GID 用于检查文件存取能力)。

返回值为 0 表示许可，-1 表示不许可。

（2）chmod()：将指定文件或目录的存取许可方式改成新的许可方式。

需要两个参数：文件名和新的存取许可方式。

（3） chown()：同时改变指定文件的所有者和小组的 UID 和 GID。

注意，chown()函数与 shell 中的 chown 命令的功能不同。

由于此子程序可以同时改变文件的所有者和组，故必须要取消所操作文件的 SUID 和 SGID 许可，以防止用户建立 SUID 和 SGID 程序，然后运行 chown()调用去获得别人的权限。

（4）stat()：返回文件的状态（属性）。

需要两个参数：文件路径名和一个结构指针，该结构指针被用来指向状态信息的存放的位置。

结构定义见表 7-2：

表 7- 2 结构定义

st_mode	文件类型和存取许可方式
st_ino	Inode 节点号
st_dev	文件所在设备的 ID
st_rdev	特别文件的 ID
st_nlink	文件链接数
st_uid	文件所有者的 UID
st_gid	文件组的 GID

st_size	按字节计数的文件大小
st_atime	最后存取时间（读）
st_mtime	最后修改时间（写）和最后状态的改变
st_ctime	最后的状态修改时间

返回值为 0 表示成功，1 表示失败

（5）umask()：将调用进程及其子进程的文件建立屏蔽值设置为指定的存取许可。

需要一个参数：新的文件建立屏值。

4．UID 和 GID 的处理

（1）getuid()：返回进程的实际 UID。

（2）getgid()：返回进程的实际 GID。

以上两个子程序可用于确定是谁在运行进程。

（3）geteuid()：返回进程的有效 UID。

（4）getegid()：返回进程的有效 GID。

以上两个子程序在一个程序不得不确定它是否在运行某用户而不是运行它本来的用户的 SUID 程序时非常有用，可调用它们来检查确认本程序的确是以该用户的 SUID 许可在运行。

（5）setuid()：用于改变有效的 UID。

对于一般用户，此子程序仅对要在有效和实际的 UID 之间变换的 SUID 程序才有用(从原有效 UID 变换为实际 UID)，以保护进程不受到安全危害。实际上该进程不再是 SUID 方式运行。

（6）setgid()：用于改变有效的 GID。

7.3.2 标准 C 函数库

下面是一些经常使用的与安全有关的标准 C 函数。

1．标准 I/O

这些函数被用来进行基本的 I/O 操作。

（1）fopen()：打开一个文件供读或写，安全方面的考虑同 open()一样。

（2）fread()，getc()，fgetc()，gets()，scanf()和 fscanf()：从已由 fopen()打开供读的文件中读取信息。它们并不关心文件的存取许可。这一点同 read()系统调用非常的类似，请读者注意。

（3）fwrite()、put()、fputc()、puts、fputs()、printf()、fprintf()：写信息到已由 fopen()打开供写的文件中。

它们也不关心文件的存取许可。同 write()。

（4）getpass()：从终端上读至多 8 个字符长的口令，不回显用户输入的字符。

需要一个参数：提示信息。

该子程序将提示信息显示在终端上，禁止字符回显功能，从/dev/tty 读取口令，然后再恢复字符回显功能，返回刚敲入的口令的指针。

（5）popen()：将在“运行 shell”那节中介绍。

2．/etc/passwd 处理

有一组子程序可对/etc/passwd 文件进行方便的存取，可对文件读取到入口项或写新的入口项或更新等等。

（1）getpwuid()：从/etc/passwd 文件中获取指定的 UID 的入口项。

（2）getpwnam()：对于指定的登录名，在/etc/passwd 文件检索入口项。

以上两个子程序返回一指向 passwd 结构的指针，该结构定义在/usr/include/pwd。h 中，

定义如下：

```
struct passwd
    char  *  pw_name;          /*  登录名  */
    char  *  pw_passwd;        /*  加密后的口令  */
     uid_t    pw_uid;          /*  UID  */
     gid_t    pw_gid;          /*  GID  */
    char  *  pw_age;           /*  代理信息  */
    char  *  pw_comment;       /*  注释  */
     char * pw_gecos;
    char  *  pw_dir;           /*  主目录  */
    char  *  pw_shell;         /*  使用的 shell */
};
```

（3）getpwent()，setpwent()，endpwent()：对口令文件作后续处理。

首次调用 getpwent()，打开/etc/passwd 并返回指向文件中第一个入口项的指针，保持调用之间文件的打开状态。

再调用 getpwent()可顺序地返回口令文件中的各入口项。

调用 setpwent()把口令文件的指针重新置为文件的开始处。

使用完口令文件后调用 endpwent()关闭口令文件。

（4）putpwent()：修改或增加/etc/passwd 文件中的入口项。

此子程序将入口项写到一个指定的文件中，一般是一个临时文件，直接写口令文件是很危险的。最好在执行前做文件封锁，使两个程序不能同时写一个文件。算法如下：

- 建立一个独立的临时文件，即/etc/passnnn，nnn 是 PID 号。
- 建立新产生的临时文件和标准临时文件/etc/ptmp 的链，若建链失败，则为有人正在使用/etc/ptmp，等待直到/etc/ptmp 可用为止或退出。
- 将/etc/passwd 拷贝到/etc/ptmp，可对此文件做任何修改。
- 将/etc/passwd 移到备份文件/etc/opasswd。
- 建立/etc/ptmp 和/etc/passwd 的链。
- 断开/etc/passnnn 与/etc/ptmp 的链。

注意：临时文件应建立在/etc 目录，才能保证文件处于同一文件系统中，建链才能成功，且临时文件不会不安全。此外，若新文件已存在，即便建链的是 root 用户，也将失败，从而保证了一旦临时文件成功地建链后没有人能再插进来干扰。当然，使用临时文件的程序应确保清除所有临时文件，正确地捕捉信号。

3．/etc/group 的处理

有一组类似于前面的子程序处理/etc/group 的信息，使用时必须用#include 语句将/usr/include/grp.h 文件加入到自己的程序中。该文件定义了 group 结构，将由 getgrnam()，getgrgid()，getgrent()返回 group 结构指针。

（1）getgrnam()：在/etc/group 文件中搜索指定的小组名，然后返回指向小组入口项的指针。

（2）getgrgid()：类似于前一子程序，不同的是搜索指定的 GID。

（3）getgrent()：返回 group 文件中的下一个入口项。

（4）setgrent()：将 group 文件的文件指针恢复到文件的起点。

（5）endgrent()：用于完成工作后，关闭 group 文件。

（6）getuid()：返回调用进程的实际 UID。

（7）getpruid()：以 getuid()返回的实际 UID 为参数，确定与实际 UID 相应的登录名，

或指定一 UID 为参数。

（8）getlogin()：返回在终端上登录的用户的指针。

系统依次检查 STDIN，STDOUT，STDERR 是否与终端相联，与终端相联的标准输入用于确定终端名，终端名用于查找列于/etc/utmp 文件中的用户，该文件由 login 维护，由 who 程序用来确认用户。

（9）cuserid()：首先调用 getlogin()，若 getlogin()返回 NULL 指针，再调用 etpwuid(getuid())。

以下为 shell 命令：

（10）logname：列出登录进终端的用户名。

（11）who am i：显示出运行这条命令的用户的登录名。

（12）id：显示实际的 UID 和 GID（若有效的 UID 和 GID 和实际的不同时也显示有效的（UID 和 GID）和相应的登录名）。

4．加密子程序

1977 年 1 月，NBS 宣布一个用于美国联邦政府 ADP 系统的网络的标准加密法：数据加密标准即 DES 用于非机密应用方面。DES 一次处理 64BITS 的块，56 位的加密键。

（1）setkey()，encrypt()：提供用户对 DES 的存取。

此两子程序都取 64BITS 长的字符数组，数组中的每个元素代表一个位，为 0 或 1。setkey()设置将按 DES 处理的加密键，忽略每第 8 位构成一个 56 位的加密键。encrypt()然后加密或解密给定的 64BITS 长的一块，加密或解密取决于该子程序的第二个变元，0：加密 1：解密。

（2）crypt()：是 UNIX 系统中的口令加密程序，也被/usr/lib/makekey 命令调用。

crypt()子程序与 crypt 命令无关，它与/usr/lib/makekey 一样取 8 个字符长的关键词，2 个 salt 字符。关键词送给 setkey()，salt 字符用于混合 encrypt() 中的 DES 算法，最终调用 encrypt()重复 25 次加密一个相同的字符串。

返回加密后的字符串指针。

5．运行 shell

下面这两个函数被用来在程序中运行 shell 命令：

（1）system()：运行/bin/sh 执行其参数指定的命令，当指定命令完成时返回。

（2）popen()：类似于 system()，不同的是命令运行时，其标准输入或输出联到由 popen()返回的文件指针。

二者都调用 fork()，exec()，popen()还调用 pipe()，完成各自的工作，因而 fork()和 exec()的安全方面的考虑开始起作用。

7.3.3 书写安全的 C 程序

一般有两方面的安全问题，在写程序时必须考虑：

1．确保自己建立的任何临时文件不含有机密数据，如果有机密数据，设置临时文件仅对自己可读/写。确保建立临时文件的目录仅对自己可写。

2．确保自己要运行的任何命令(通过 system()，popen()，execlp()，execvp()运行的命令)的确是自己要运行的命令，而不是其它什么命令，尤其是自己的程序为 SUID 或 SGID 许可时要小心。

第一方面比较简单，在程序开始前调用 umask(077)。若要使文件对其他人可读，可再调 chmod()，也可用下述语名建立一个"不可见"的临时文件。

```
creat("/tmp/xxx"，0);
file=open("/tmp/xxx"，O_RDWR);
```

unlink("/tmp/xxx");

文件/tmp/xxx 建立后，打开，然后断开链，但是分配给该文件的存储器并未删除，直到最终指向该文件的文件通道被关闭时才被删除。打开该文件的进程和它的任何子进程都可存取这个临时文件，而其它进程不能存取该文件，因为它在/tmp 中的目录项已被 unlink()删除。

第二方面比较复杂而微妙，由于 system()，popen()，execlp()，execvp()执行时，若不给出执行命令的全路径，就能"骗"用户的程序去执行不同的命令。因为系统子程序是根据 PATH 变量确定哪种顺序搜索哪些目录，以寻找指定的命令，这称为 SUID 陷井。最安全的办法是在调用 system()前将有效 UID 改变成实际 UID，另一种比较好的方法是以全路径名命令作为参数。execl()，execv()，execle()，execve()都要求全路径名作为参数。有关 SUID 陷井的另一方式是在程序中设置 PATH，由于 system()和 popen()都启动 shell，故可使用 shell 句法。如：

system("PATH=/bin：/usr/bin cd");

这样允许用户运行系统命令而不必知道要执行的命令在哪个目录中，但这种方法不能用于 execlp()，execvp()中，因为它们不能启动 shell 执行调用序列传递的命令字符串。

再强调一次：在通过自己的程序运行另一个程序前，应将有效 UID 改为实际的 UID，等另一个程序退出后，再将有效 UID 改回原来的有效 UID

7.3.4 SUID/SGID 程序指导准则

以下是书写 SUID 和 SFID 程序时应该注意的安全准则：

1．不要写 SUID/SGID 程序，大多数时候无此必要。

2．设置 SGID 许可，不要设置 SUID 许可。应独自建立一个新的小组。

3．不要用 exec()执行任何程序。记住 exec()也被 system()和 popen()调用。

- 若要调用 exec()（或 system()，popen()），应事先用 setgid()（getgid()）将有效 GID 置加实际 GID。
- 若不能用 setgid()，则调用 system()或 popen()时，应设置 IFS：

  popen("IFS=\t\n;export IFS;/bin/ls"，"r");
- 使用要执行的命令的全路径名。
- 若不能使用全路径名，则应在命令前先设置 PATH：

  popen("IFS=\t\n;export IFS;PATH=/bin：/usr/bin;/bin/ls"，"r");
- 不要将用户规定的参数传给 system()或 popen();若无法避免则应检查变元字符串中是否有特殊的 shell 字符。
- 若用户有个大程序，调用 exec()执行许多其它程序，这种情况下不要将大程序设置为 SGID 许可。可以写一个（或多个）更小，更简单的 SGID 程序执行必须具有 SGID 许可的任务，然后由大程序执行这些小 SGID 程序。

4．若用户必须使用 SUID 而不是 SGID，以相同的顺序记住(2)，(3)项内容，并相应调整。不要设置 root 的 SUID 许可。选一个其它户头。

5．若用户想给予其他人执行自己的 shell 程序的许可，但又不想让他们能读该程序，可将程序设置为仅执行许可，并只能通过自己的 shell 程序来运行。

6．编译，安装 SUID/SGID 程序时应按下面的方法：

（1）确保所有的 SUID（SGID）程序是对于小组和其他用户都是不可写的，存取权限的限制低于 4755（2755）将带来麻烦。只能更严格。4111（2111）将使其他人无法寻找程序中的安全漏洞。

（2）警惕外来的编码和 make/install 方法。

某些 make/install 方法不加选择地建立 SUID/SGID 程序。这会极大的威胁系统的安全，因此，在安装外来的编码时，应该：

- 检查违背上述指导原则的 SUID/SGID 许可的编码。
- 检查 makefile 文件中可能建立 SUID/SGID 文件的命令。

7.3.5 root 程序的设计

有若干个子程序可以从有效 UID 为 0 的进程中调用。许多前面提到的子程序，当从 root 进程中调用时，将完成和原来不同的处理。主要是忽略了许可权限的检查。由 root 用户运行的程序当然是 root 进程(SUID 除外)，因有效 UID 用于确定文件的存取权限，所以从具有 root 的程序中，调用 fork()产生的进程，也是 root 进程。

1．setuid()：从 root 进程调用 setuid()时，其处理有所不同，setuid()将把有效的和实际的 UID 都置为指定的值。这个值可以是任何整型数。而对非 root 进程则仅能以实际 UID 或本进程原来有效的 UID 为变量值调用 setuid()。

2．setgid()：在系统进程中调用 setgid()时，与 setuid()类似，将实际和有效的 GID 都改变成其参数指定的值。

调用以上两个子程序时，应当注意下面几点：

- 调用一次 setuid()（setgid()）将同时设置有效和实际 UID（GID），独立分别设置有效或实际 UID（GID）固然很好，但无法做到这点。
- setuid()（setgid()）可将有效和实际 UID（GID）设置成任何整型数，其数值不必一定与/etc/passwd(/etc/group)中用户(小组)相关联。一旦程序以一个用户的 UID 了 setuid()，该程序就不再做为 root 运行，也不可能再获 root 特权。

3．chown()：当 root 进程运行 chown()时，chown()将不删除文件的 SUID 和/或 SGID 许可，但当非 root 进程运行 chown()时，chown()将取消文件的 SUID 和/或 SGID 许可。

4．chroot()：改变进程对根目录的概念，调用 chroot()后，进程就不能把当前工作目录改变到新的根目录以上的任一目录，所有以/开始的路径搜索，都从新的根目录开始。

5．mknod()：用于建立一个文件，类似于 creat()，差别是 mknod()不返回所打开文件的文件描述符，并且能建立任何类型的文件（普通文件，特殊文件，目录文件）。若从非 root 进程调用 mknod()将执行失败，只有建立 FIFO 特别文件（有名管道文件）时例外，其它任何情况下，必须从 root 进程调用 mknod()。由 creat()仅能建立普通文件，mknod()是建立目录文件的唯一途径，因而仅有 root 能建立目录，这就是为什么 mkdir 命令具有 SUID 许可并属 root 所有。一般不从程序中调用 mknod()。通常用/etc/mknod 命令建立特别设备文件而这些文件一般不能在使用着时建立和删除，mkdir 命令用于建立目录。当用 mknod()建立特别文件时，应当注意确从所建的特别文件不允许存取内存，磁盘，终端和其它设备。

6．unlink()：用于删除文件。参数是要删除文件的路径名指针。当指定了目录时，必须从 root 进程调用 unlink()，这是必须从 root 进程调用 unlink()的唯一情况，这就是为什么 rmdir 命令具有 root 的 SGID 许可的原因。

7．mount()，umount()：由 root 进程调用，分别用于安装和拆卸文件系统。这两个子程序也被 mount 和 umount 命令调用，其参数基本和命令的参数相同。调用 mount()，需要给出一个特别文件和一个目录的指针，特别文件上的文件系统就将安装在该目录下，调用时还要给出一个标识选项，指定被安装的文件系统要被读/写(0)还是仅读(1)。umount()的参数是要一个要拆卸的特别文件的指针。

7.4 小结

在本章中我们简要介绍了有关网络安全性的一些知识。网络安全性是一个很大的主题，我们在这里只能介绍一些基本的知识，如果想对此作更加深入的了解，请参阅有关网络安全性的其它资料。

第十章　远程命令执行

10.1　引言

所谓远程命令的执行，是指在本地主机上的一个活动进程能让一个在远程主机上的程序被执行。4.3 BSD 提供了 rsh 程序，执行它可以激活另外一个系统上的程序，rsh 程序具有如下的功能：一个进程可以通过远程输入输出命令去激活另外一个系统上的程序（进程）。

我们希望在通常的 Linux 环境下，就能够直接把本地的数据写到远程系统的输入输出通道中，并且能够直接从远程系统的标准输入通道中读取数据，另外，我们还希望能直接从远程系统的标准出错通道中读取出错信息。我们必须有效的把标准输入和标准出错这两类不同的信息流区分开来，否则我们就无法辨认哪个是标准的输入数据，哪个是标准的出错信息。此外，我们还希望能够向远程进程发信号，此外作为控制远程进程执行的方法，该控制进程必须在远程主机上运行，因为它必须在该主机上执行 kill 系统调用向远程进程发信号。

这些要求的关系如图 10-1：

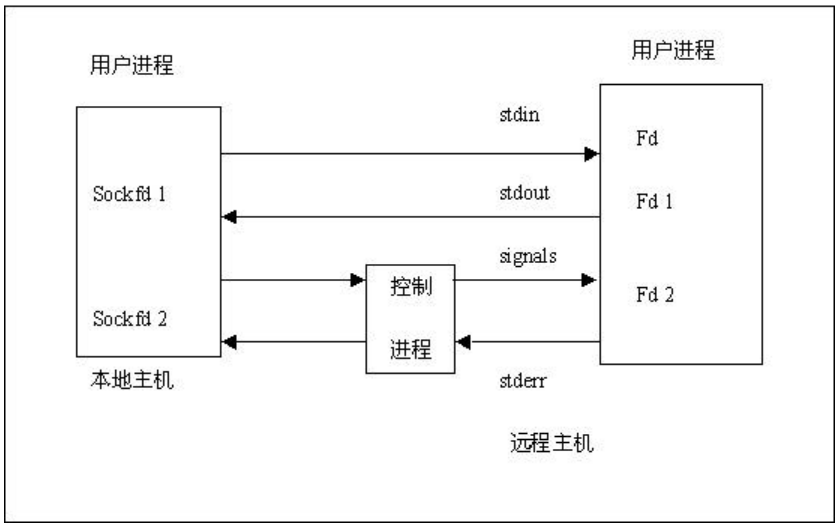

图 10-1　远程命令执行示意

现在我们要讨论两个在远程系统执行中执行这类进程的函数：rcmd 和 rexec，这两者的区别与用来确认函数调用者在远程系统上执行特定进程的许可权的方法有关，并且这两个函数是通过激活远程系统上的不同服务器来建立本地进程和远程进程间的通信。

它们的区别如表 10-1。

表 10-1　rcmd 和 rexec 函数的比较

本地主机上的函数	远程主机上服务器	区　　别
Rcmd()	Rshd	调用者必须具有超级用户的权限以便可以在本地主机上登记一个保留端口。不需要登录名和口令。

Rexec()	Rexecd	为了在远程主机上检验，调用者必须将登录名和口令（登录时键入的口令字符串）传输给服务器。

10.2 rcmd 函数和 rshd 服务器

rcmd 客户函数以及它对应的 rshd 服务器是 4.3 BSD 系统的关键，许多 4.3 BSD 的“r”命令都 调用了 rcmd 函数，如：rlogin 、rcp 、rsh rdist 以及 rdump。

rcmd 函数的原型是：

```
int rcmd (char * * ahost , int remport ,char * cliuname ,
char * servuame ,char * cmd ,int * sockfd2);
```

参数 ahost 是指向远程主机名字的地址的指针，主机名字可以通过函数 gethostbyname 查找。由于这是一个指向字符串的指针，因而主机名可以返回给调用者。

rcmd 函数得到一个保留的 TCP 端口，这就意味着调用进程必须拥有超级用户的特权，远程系统上 rcmd 连接的 TCP 端由 remport 参数指明，rcmd 连接到的进程了解 rcmd 和 rshd 使用的协议。

在调用 rcmd 函数之前的一段典型代码是：

```
# include <netdb.h>
struct server * sp;
if ( (sp=getservbyname ( “shell”, “tcp”) ) = = NULL)
{
    fprint ( stderr , “shell/tcp: unknown service \n”);
    exit (1);
}
```

remport 参数必须按照网络字节顺序，安排这由库函数 getservbyname 处理。

cliuname 和 servuname 参数分别指明用户名和服务器名，这些登录名由远程系统上的服务器用来确认用户。

cmd 串含有在远程主机上执行的命令串，rshd 执行此命令串：

```
shell – c cmd
```

这里，shell 是远程系统口令文件 servuname 项中的外壳域，此域通常指明三个普通的 UNIX 外壳之一。Bourneshell ，Cshell 或者 KornShell。因为该命令串由一外壳执行，它可以含有特殊外壳变形字符，由远程主机上的外壳来解释。

最后一个参数 sockfd2 是指向一整数的指针。若此指针不是 NULL，则 rcmd 函数打开用进程与控制进程之间的第二套接字，然后通过此指针返回给调用者。若此指针为 NULL，则远程进程的标准出错复制到标准输出（sockfd1），并且无法向远程进程发信号。这个第二套接字通过 rcmd 被连接到保留的 TCP 端口，我们把这个 TCP 端口称为辅助端口。

rcmd 函数返回值是套接字描述符 sockfd1，如果出错，则返回-1。

rcmd 函数使用的应用协议见表 10-2。

表 10-2 rcmd 使用的协议

客户－rcmd( )	服务器－rshd
使用一个保留端口建立一个套接字。连接到服务器。	接收连接并得到客户的地址，如果用户不连接到保留口，则终止。
如果辅助端口由调用者请求，则使用一个保留端口建立一个另套接字。写 ASCII 串指明辅助端口号。仅写含有终止空字节的空串。	读辅助端口号。如果非零且不在保留端口范围之内，则终止；如果非零，则使用一个保留端口建立一个套接字并连接到客户的辅

	助端口。
如果需要辅助端口，接收来自服务器的地址，且如果服务器为起辅助连接结束未连接一个保留端口，返回-1 给调用者。写三个 ASCII 串给服务器 cliuname、servuname 和 cmd。	读三个 ASCII 串：客户用户名、服务器用户名和命令串，确认用户。如果成功，给客户应答一个值为二进制数字 0 的字节。如果不成功，则写一个值为二进制数字 1 的字节，后跟 ASCII 出错信息及换行，并终止。
从服务器读确认状态。如果出错，则读出错误信息并将它输出到标准出错，然后给调用者返回-1；如果成功，则给调用者返回套接字描述符。	服务器建立进程结构：如果需要辅助端口，建立控制进程，为客户激活外壳来执行客户命令。

下面是 rcmd 函数的源程序主体，滤去了部分出错处理及部分变量说明：

```
/*返回套接字描述符 sockfd1*/
rcmd (ahost , rport , cliuname , servuname , cmd , sockfd1)
char    * *  ahost ;
/*  主机名地址指针*/
u_short rport;
/*  服务器端口*/
char   * cliuname；
/*  客户系统中的用户名*/
char   * servuname;
/*  服务器系统中的用户名*/
char   *cmd;
/*  服务器上要执行的命令串*/
  int     *  fd2ptr;
  /*  指向第二个套接字描述符*/


  {
      int            sockfd1,lport;
      char           c;
      struct sockaddr_in serv_addr,serv2_addr;
      struct hostent      *  hp;
      fe_set                 readfds;

       hp=gethostbyname (*ahost);
       *ahost=hp->h_name;
       for(;;)
       {
       sockfd1=rresvport(&lport);
       fcnt1( sockfd1,F_SETOWN,getpid( ) );
       /*  填充服务器套接字地址并与之连接*/
       bzero ( ( char * ) &serv_addr,sizeof(serv_addr) );
       serv_addr.sin_family = hp ->h_addrtype;
       hcopy ( hp->h_addr_list[0],
       ( caddr_t)&serv_addr.sin_addr,hp ->h_length);
       serv_addr.sin_port = rport;
```

```
        if ( connect (sockfd1, (struct sockaddr * )&serv_addr,
        sizeof (serv_addr) )   >=0 )
        break;                              /*  OK，继续下一步  */
        close (sockfd1);
        /*出错处理（省略）*/

        if ( hp ->h_addr_list[1] !=NULL)
        {                                                    /*如果主机另有地址*/
              hp ->h_addr_list ++;
              bcopy (hp ->h_addr_list[0],
              (caddr_t)&serv_addr.sin_addr,hp ->h_length) ;
              fprintf (stderr , "Trying %s …\n",
              inet_ntoa (serv_addr.sin_addr) );
              continue;
        }
}       /*  end of for */

if ( fd2ptr = = ( int ) 0 )      /*  调用者不使用第二通道*/
{
        write (sockfd1," ",1);
        lport = 0;
}
else
{
        lport - -;
           socktemp = rresvport (&lport);
        listen (socktemp,1)
        /*  将含有端口号的 ASCII 串发往服务器*/
        sprintf (num, "%d",lport);
        write (sockfd1,num,strlen (num) +1);
        FD_ZERO (&readfds)
        FD_SET (sockfd1,&readfds);
        FD_SET (socktemp,&readfds);
        Select (32,&readfds, (fd_set *) 0,(fd_set *)0,
           (struct timeval *)0;
        FD_ISSET( socktemp.&readfds) ;
        /*  服务器在第二套接字上与客户连接  */
        len=sizeof (serv2_addr);
        sockfd2 =accept (socktemp,& serv2_addr,&len);
        close (socktemp);
        fd2ptr = sockfd2;                   /  返回给调用者  */
        serv2_addr.sin_port =ntohs ( (u_short) serv2_addr.sin_port);
)
```

```
    write  (sockfd1,cliuname,     strlen(cliuname) +1 );
    write  (sockfd1,servuname,   strlen(servuname) +1 );
    write  ( sockfd1,cmd ,         strlen(cdm) +1 ) ;
    read ( sockfd1,&c,1);

    if (c ! = 0)
    /*  未收到应答()，表明服务器出错，从服务器读出错误信息并送往标准出错*/
      while ( read (sockfd1,&c,1) = = 1)
    {
    write (2,&c,1) ;
    if ( c= = '\n');
      break;
    }
}
/* end of rcmd*/
```

进程控制是由服务器进程 rshd **加以实施的，下面我们先给出** rshd **源程序的大体框架，然后再讨论。这里滤去了若干变量说明和出错处理。**

```
    int one =1;
    /* main 函数*/
    main ( )

    {
        struct sockaddr_in    cli_addr;
        struct  linger              linger;

        openlog ("rsh,LOG_PID|LOG_ODELAY,LOG_DAEMON);
        addrlen = sizeof (cli_addr);
        getpeername (0,(struct sockaddr *)&cli_addr,&addrlen) ;
        setsockpt (0,SOL_SOCKET,SO_KEEPALIVE,( char *)*one,
        sizeof ( one ) );
        linger.1_onoff = 1;
        linger.1_linger = 60;
        setsockopt (0,SOL_SOCKET,SO_LINGER,(chat *)&linger,
          sizeof( linger) );
        Do (&cli_addr);     /*  Do 函数不返回*/
    }

    /* do 函数：*/
    do (cli_addrp)
    Struct sockaddr_in * cli _addrp;     /*  client's internet address*/
    {
        signal (SIGINT ,SIG_DFL):
        signal (SIGQUIT,SLG_DFL);
        signal (SIGTERM,SIG_DFL);
```

```
if ( cli_addrp ->sin_family ! =AF_INET)
        exit (1);
cli_addrp ->sin_port = ntohs ( (u)short )cli_addrp ->sin_port};
if ( cli_addrp ->sin_port >=IPPORT_RESERVED || cli_addrp ->sin_port<
IPPORT_RESERVED/2)
exit( 1 );
alarm (60);
clisecport = 0;
for (;;)
{
    if ( cc =read( 0,&c,1)!=1)
      {
          shutdown (0,2);
          exit(1);
      }
    if ( c= = 0 )
        break;
    clisecport=(clisecport * 10)+(c-'0');
}
alarm (0);
if ( clisecport ! =0)
    {
        if (clisecport >=IPPORT_RESERVED)
            exit(1);
        oursecport = IPPCRT_RESERVED – 1;
        if ( ( sockfd2=rresvport (&oursecport) )<0)
            exit (1);
        cli_addrp ->sin_port=htons ( u_short)clisecport};
        if ( connect (sockfd2,( struct sockaddr *)cli_addrp, sizeof (* cli_addrp) ) <0)
            exit(1);
    }
hp= gethostbyname ( ( char* )&cli_addrp ->sin_addr,sizeof (struct in_addr),
  cli_addrp->sin_family);
setpwent( );
if (chdir(pwd->pw_dir) <0)
  chdir("/");
if (pwd ->pw_passwd !=NULL& *pwd->pw_passwd !='\0' &&
  ruserok (hostname,pwd ->pa_uid= =0,cliuname,servuname)<0)
    exit(1);
if (pwd->pw_iud !=0&&access("/etc/nologin",F_OK) = =0 )
    exit (1);
/* 向用户回写空字节，表明成功*/
if (write (2," ",1) !=1)
  exit (1);
```

```
    if (clisecport )
    {
        if ( pipe (pipefd)<0)
            exit (1);
        if ( childpid =fork( ) ) = = -1}
            exit (1);
        if (pipefd[0]>sockfd2)
            maxfdpl=pipefd[ 0 ];
        else
            maxfdpl = sockfd2;
        maxfdpl ++;
    if (childpid !=0)
    {
        close (0);
        close (1);
        close (2);
        close (pipefd[1]);
        FD_ZERO(&readfrom);
        FD_SET(sockfd2,&readfrom);
        FD_SET(pipefd[0],&readfrom);
        ioctl( pipefd[0],FIONBIO,(char *)&one);
        do {
            Ready=readfrom;
            if (select (maxfdpl,&ready,(fd_set *)0, (fd_set *)0,(struct timeval *)0)<0)
                break;
            if (FD-ISSET(sockfd2,&ready) )
            if (read(sockfd2,&sigval,1)<=0)
                FD-CLR(sockfd2,&readfrom};
            else
                killpg (childpid,sigval);
            if( FD_ISSET(pipefd[0],&ready );
            {
                cc=read(pipefd[0],buf,sizeof ( buf );
                if cc<=0 )
                {
                    shutdown (sockfd2,2);
                    FD_CLR( pipefd[0] ,&readfrom );
                }
                else write (sockfd2 ,buf, cc );
             )while (FD_ISSET(sockfd2 ,&readfrom ) ||
            FD_ISSET (pipefd[0],&readfrom) );
            exit( 0 );
    }
    setpgrp (0, getpid( ) );
```

```
close ( sockfd2 );
close (pipefd[ 0 ]);
dup2 (pipefd [ 1 ],2);
close (pipefdp[ 1 ]);
}
if ( * pwd ->pw_shell = =‘\0’)
   pwd ->pw_shell =“/bin/sh”;
setgid ( ( gid_t)pwd->pwd_gid);
initgroups ( pwd ->pw_name,pwd->pw_gid);
setuid ( (uid_t)pwd ->pw_uid);
ebviron =env_prts;
strncat (env_home,pwd->pw_dir,sizeof (env_home) – 6 );
strncat (env_shell,pwd ->pw_shell,sizeof(env_shell) – 7);
strncat (env_user ,pwd->name,sizeof(env_user) – 6 );
if ( ( cp = rindex)pwd->pw_shell, ‘/’ ) )!=NULL )
   cp ++;
   else cp =pwd ->pw_shell;
   execl (pwd->pw_shell,cp, “-c”,cmdbuf , (char *)0 );
   exit (1);
}
```

注意，如果 chdir **失败，对指定服务器用户名的主目录的修改会返回一个错误。此外成为控制进程的循环使用的是** killpg **系统调用给一个指定进程组发信号，这是因为** 4.3 BSD **不支持** kill **系统调用给进程组发信号的选择。**

当 rcmd **连接** rshd **服务器的时候，**4.3 BSD **系统上的** inetd **超级服务器在该端口上侦听连接请求，通过** fork **和** exec **激活** rshd **服务器，并由** rshd **进程执行外壳程序，然后它又复制并执行用户命令，这是在远程系统上两次** fork **和三次** exec **的最小情况，是不可避免的。**

在 rshd **源代码口应该注意：**

- **子进程建立自身，作为一进程组的头，也即它执行的** shell **和** shell **的任何子程序均属于同一进程组。这主要是为了满足信号处理的需要。控制进程（**rshd **父进程）执行** killpg **系统调用将接收的来自客户的任何信号发给整个进程组，而不仅是启动的** shell **进程。**
- **子进程为由它激活的** shell **建立一个环境表，这与** login **程序调用用户的** shell **前所做的工作类似，至少在激活用户的** shell **时，应该设置** HOME**、**USER**、和** PATH**变量。**
- **本地系统上的客户进程无法得到远程系统上的** shell **的** exit **状态。**

10.3 rexec 函数和 rexecd 服务器

rexec **函数和** rcmd **函数类似，但有一个大的区别，那就是调用** rexec **的进程不需要超级用户权限，因为不需要一个保留端口。在** roxec **和其服务器** roxecd **之间使用的确认方案是从客户传递到服务器的用户的文本口令，在服务器加密后与服务器系统中的口令文件中的加密文件进行比较。**

但这样一来就使得用户的口令明文在网络上传输，而且为了将口令明文传送给 roxec **函数，调用者必须将口令放在源文件中，这是安全问题中的一大漏洞，因此基本上不实用。**

roxec **的调用参数和** rcmd **相似：**

```
int rexec (char * * ahost , int remport ,char * servuname,
```

char * password , char * cmd , int * sockfd2);

ahost remport , servuname ,cmd ,和 sockfd2 参数与 rcmd 函数的参数意义相同，口令及服务器用户名用于在服务器系统上确认调用者。

rexec 函数 与 rexecd 服务器的源代码与前面的 rcmd 和 rshd 的源代码相似，它们的控制进程相同，唯一的区别与远程主机上的用户确认相关，因此这里不再给出它们的源代码了。

第十一章 远程注册

11.1 简介

用户在使用计算机的时候，往往需要在一台计算机上登录进入另外一台计算机中，以使用另外一台计算机的资源。在一个计算机上注册登录进入另外一台计算机，称为远程登录，它是网络应用很重要的一个部分。4.3 BSD 提供了两个远程注册实用程序，一个是 rlogin，该程序假设远程服务器也是一个 Linux 系统，另外一个 实用程序是 Telnet，它是 TCP /IP 支持的标准的 INTERNET 的一个实用程序，与主机操作系统无关，本书将主要讨论 rlogin。

远程登录与多用户、多终端系统有一定的相似之处，因此用户和容易将它们混淆起来。远程登录的实质是杂本地仿真一个终端，调用远程系统上的 shell 进程处理本地命令。我们将在以后的文章中进行讨论其具体的过程。

伪终端是远程登录中非常重要的一个部分，它对于本地用户来说就相当于一个终端，用户对其操作就相当于在远程系统上进行操作。伪终端涉及到许多的东西，比较复杂，应对其有充分的了解。伪终端仿真程序只是远程登录程序的有个部分，一个有机的组成，只负责仿真远程终端，而远程登录的控制部分由远程登录的客户程序和服务器程序完成，这个我们应该充分认识到。

远程登录程序由客户程序和服务器程序组成，它们分别负责本地系统及远程系统中网络通信，将请求及应答在两个系统之间传送。

11.2 终端行律和伪终端

终端是一个全双工的设备，具有单独的输入路径和输出路径。终端行律（terminal line discipline ）在内核中处于用户进程和设备驱动程序之间，它能够完成下面所例举的几个功能：

- 回显键入的字符。
- 把键入的字符组成行，以便进程能够整行的从终端读入。
- 编辑输入的行。Linux 允许擦除以前键入的字符，也允许擦除一个整行，从新的一行开始输入。
- 当某些终端控制键被按下的时候产生信号，比如：SIGINT 和 SIGQUIT。
- 处理流控制字符，如按下 Ctr—S 键，屏幕输出暂停，按下 Ctr—Q 键后又重新开始输出。
- 允许键入文件结束字符。
- 进行字符转换。例如，当一个进程写入一个新行开始符，行律都把它转换成为一个回车符和一个换行符。又如，如果终端不处理 tab 键，则将 tab 字符转换成若干字符。

终端处理的部分难点在于许多不同的设备连接在一根异步串行线上。不仅交互式终端是如此，打印机、调制解调器、绘图仪等等设备也是如此。即使终端用于交互式输入，不同的程序也会以不同的方式访问终端，有些处理终端的行，有些处理全屏幕编辑，而有些则禁止回显。

但对于某些程序来说，它们截取了标准输入和标准输出，对其进行另外的处理，如记录所有的输入和输出。但是这类程序有些缺点，如果得不到系统提示符或者不能 tty ,vi 等，这是因为，模拟的标准输入和标准输出在系统看来，不是一个终端设备。如果在其中加入

行律，则上面所说的所有问题就不存在，按照上面的规律，我们可以知道，事实上，这也就是伪终端所要做的事情。

伪终端（pseudo terminal）是一对设备，一半称为主设备，一半称为从设备，进程打开一对终端设备就同时得到两个文件描述符，我们把伪终端缩写成 pty。

伪终端如图 11-1 所示：

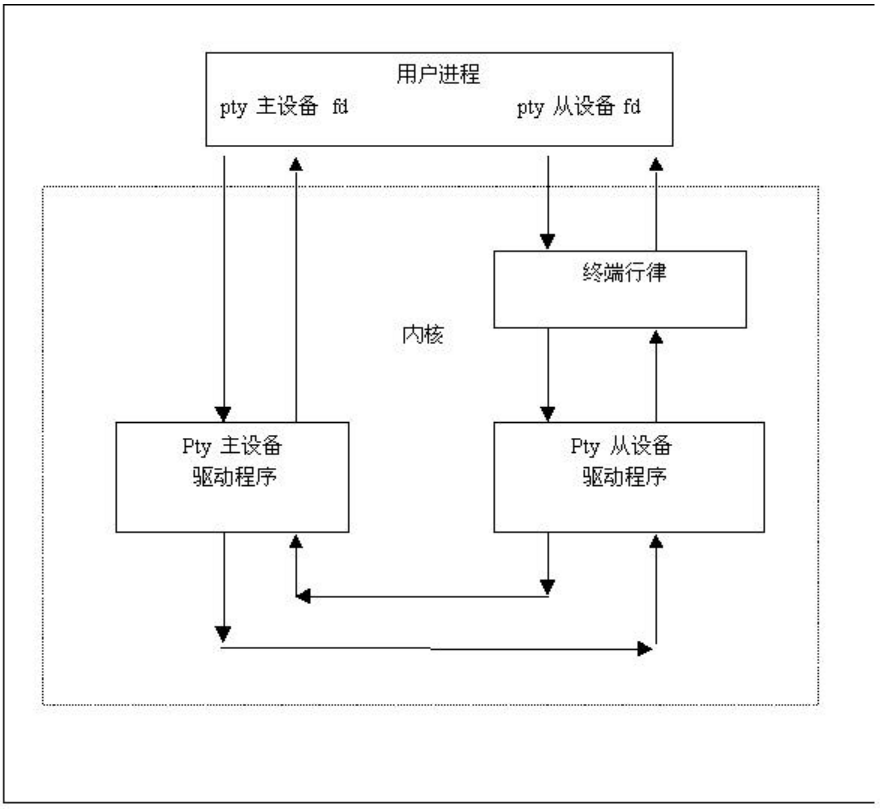

图 11-1 伪终端

写进主 pty 的所有数据看上去像是来自从 pty，而写进从 pty 的所有数据看起来则像是来自主 pty。下面的两个函数运行于 4.3 BSD，pty_master 打开主 pty，而 pty_slave 打开从 pty：

```
static char pty_name[ 12 ];
   /* "/dev/[pt]tyXY" = 10 chars + null byte */
   int pty_master( )

{
int i ,master_fd;
char * ptr ;
struct stat statbuff;
   static char ptychar ="pqrs";                              /* X */
   static char hexdigit []="0123456789abcdef";                    /* Y * /

/*  打开主 pty "/dev/pty[pqrs][0-9a-d]" */
for (ptr=ptychar; * pty!=0;pty++)
   {
strcpy (pty_name, "dev/ptyXY");
      pty_name[8]= ptr ;    / X */
```

```
    pty_name[9]='0';        /* Y */
  /*  如果不存在名字 "dev/pty0"，则退出 */
  if stat( pty_name,&statbuff)<0}
  break;
  for (i= 0;i<10;i++)
    {
pty_name[9]=hexdigit[i];                    /*0-15->0-9a-f */
if ( ( master_fd=fopen(pty_name,O_RDWR) )>=0)
return ( master_fd);
    }
  }
return (-1);
  }                        /* end of pty_master */
int pty_slave (master_fd)
int master)fd;

{
int slave_fd;
pty_name[5]='t';                    /* 将"/dev/ptyXY"改为 "/dev/ttyXY" */
if ( ( slave_fd-fopen(pty_name,O_RDWR) )<0)
    {
close (master_fd);
return (-1);
    }
}
```

System V Release 3.2 **也支持伪终端，在库** / usr / lib / libpty **中有三个函数可以用来建立伪终端，并且由函数** grantpt **激活的程序** / usr / lib / pt_chomod **可改变从设备的权限和属主** (Owner)**，这是一个设置用户号的根程序，需要超级用户的特权来改变从设备的属主。下面是函数** pty_master **和** pty _slave**：**

```
#define     PTY_MASTER     "dev/ptmx"
int pty_masster( )

{
int master_fd;
/*打开 pty "/dev/ptmx" */
if ( ( master_fd= open (PTY_MASTER,O_RDWR) )
  return (-1);
  return (master_fd);
)

int pty_slave(master_fd)
int master_fd;
{
int slave_fd;
```

```
char    *    slavename;
int          grantpt( );   /* libpt.a */
int          nlockpt( );   /*libpt.a */
char    *    ptsname ( ); /* libpt.a */

if ( grantpt(master_fd)<0)
      close (masetr_fd);
if ( unlockpt(master_fd)<0
{
close ( master_fd);
return (-1);
}
slavename=ptsname(master_fd);
if (slavename = =NULL)

{
      close (master_fd);
      return (-1);
}
slave_fd = open (slavenam,O_RDWR);
if (slave_fd <0)

{
       close(master_fd);
       return (-1);
}
/*  降将伪终端硬件仿真模块 ptem 和标  */
/*  将终端行律 ldterm 压入从设备流  */
if )ioctl (slave_fd,I_PUSH,“ptem”)<0}

{
       close (master_fd);
       return (-1);
}
if (ioctl(slave_fd,I_PUSH,“idterm”)<0)
{
       close(master_fd);
       return(-1);
}
return (slave_fd);
}
```

组成终端行律的模块加在从设备流上而非主设备流上,这是为了让从设备看起来更像一

个终端。伪终端的典型用法是由一个进程打开一个 pty，然后调用 fork 建立自己的一个子进程，然后由子进程打开从 pty 并加载另一个程序，并把从 pty 作为标准输入，标准输出和标准错误传给新进程，该进程便如同连上了一个终端设备。

11.3 终端方式字和控制终端

有些程序中有两个行律模块，都试图处理特殊字符，一般情况下我们只需要一个行律模块来解释键入的字符。在实际终端上我们需要行律模块以传递所有的东西，而 pty 需要行律模块以便像正常终端一样运行，这就需要把伪终端的方式字初始化为终端方式字并把它的行律模块置为 cooked 方式以便传递所有的字符。

4.3 BSD 的终端具有如下三种方式：

- cooked 方式。在该方式下输入被加工成行，所有特殊字符都要进行处理，一般的交互式终端都处于这种方式。
- raw 方式。该方式允许进程接收所有输入的字符，而系统不对这些字符进行解释，如 vi 就使用这种方式。但是，如果进程设置了该方式，而在终止的时候没有设置回到 cooked 方式，终端便一直处于 raw 方式，这样该终端便可能将禁止回显的字符回显出来，只有在键入换行符的时候才能终止一行。
- cbreak 方式。该方式介于 cooked 方式和 raw 方式之间，在 cbreak 方式下，进程一次从终端读入一个字符而不是一行字符，信号键仍然起作用，但是可编辑的特点就没有了。

不需要回显字符并要字符不带加工的传输的时候，就应该设置终端为 raw 方式。4.3 BSD 中该函数为：

```
   static   struct sgttyd       tty_mode;
int tty_raw(fd)
int fd;                    /*  终端设备  */
{
   struct sgttyb temp_mode;
   if (ioctl (fd,TIOCGETP,      (char *)&temp_mode)<0)
   return (-1);
   tty_mode =temp_mode ;
   temp_mode.sg_flags | =RAW ;      /*   打开 RAW 模式  */
   temp_mode.sg_flags & = ~ ECHO;            /*关闭 ECHO*/
   if (ioctl (fd,TIOCSETP,(char *)*temp_mode)<0)
      return (-1);
   return (0);
}
   /*  恢复终端模式为调用 tty_mode 之前模式*/
   int tty_reset (fd)
   int fd;
   {
      if (ioctl (fd,TIOCSETP,(char * )&tty_mode) <0)
      return (-1); return (0);
   }
```

另外，我们需要把 pty 初始化为某中已知的状态。当从设备打开的时候，我们不能确定终端行律处于哪重状态，为了设置这个状态，我们需要把标准输入上的终端行律方式字

记录下来，然后把 pty 的方式字设置为 raw 方式。在 4.3 BSD 中，把终端方式字记录下来需要记录四个结构和两个整数。

读取以及设置方式字的函数 tty_getmode 和 tty_ setmode 如下：

```
static struct sttyd          tty_sgttyb;
static struct tchars         tty_tchars;
static struct ltchars        tty_ltchars;
static struct winsize        tty_winsize;
static int                   tty _localmode;
static int                   tty_ldisc;

int tty_getmode(oldfd)
int oldfd;
{
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_sgttyb)<0)
      return (-1);
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_tchars)<0)
      return (-1);
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_ltchars)<0)
      return (-1);
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_localmode)<0)
      return (-1);
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_ldisc)<0)
      return (-1);
   if (ioctl (oldfd,TIOCGETP,(char * )&ttya_winsize)<0)
      return (-1);
   return (0);
}

int tty_setmode(newfd)
int newfd;
{
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_sgttyb)<0)
      return (-1);
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_tchars)<0)
      return (-1);
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_ltchars)<0)
      return (-1);
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_localmode)<0)
      return (-1);
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_ldisc)<0)
      return (-1);
   if (ioctl(newfd,TIOCGETP, ( char *)&tty_winsize)<0)
      return (-1);
   return (0);
```

```
}
```

System V没有像 4.3 BSD 一样提供三个指定的方式字，而可以禁止和允许诸如信号产生、组成行、编辑行、和回显。

以下 System V 函数把终端设置为 raw 方式：

```
static struct termio        tty_mode;
int tty_raw(fd)
int fd;
{
      struct termio tmp_mode;
      if (ioctl (fd ,TCGETA,(char * )&tmp_mode )<0)
            return (-1);
      tty_mode = tmp_mode;
      tmp_mode.c_iflag =0;
      tmp_mode.c_oflag & = ~OPOST;
      tmp_mode.c_lflag & = ~(ISIG|TCANON|ECHO|XCASE);
      tmp_mode.c_cflag & =~(CSIZE|PARENB);
      tmp_mode.c_cflag 1 =cs8
      tmp_mode.c_cc [VMIN] =1;
      tmp_mode.c_cc[VTIME] =1;
      if (ioctl (fd,TCSETA,(char * )& tmp_mode)<0)
         return (-1);
         return (0);
}
```

在 System V 中要记录终端的方式只需要把结构 termio 记下便可，如下列程序所示：

```
if ( (fd= open("dev/tty",O_RDWR) ) > =0)
   {
      if (ioctl (fd,TIOCNOTTY,( char * )0)<0)
      err_sys("ioctl TIOCNOTTY error");
      close (fd);
   }
if ( (slave_fd =pty_slave(master_fd) ) <0)
      err _sys ("can't open pty slave");
close (master_fd);
if ( ( tty_setmode(slave_fd)<0)
      err_sys("can't tty mode of pty slave");
```

在 System V 中，为了使子进程脱离其控制终端而把从 pty 作为其控制终端，必须：

- 调用 setpgrp 使子进程成为进程组头，与它的控制终端脱离。
- 打开从 pty 使之成为它的控制终端。

实现上述两点的代码段如下：

```
setpgrp ( );
if ( ( slave_fd =pty_slave(master_fd)) <0)
      err_sys("can't open pty slave");
close (master _fd);
if (tty_setmode(slave_fd)<0)
```

```
err_sys("can't set tty mode of pty slave");
```

11.4 rlogin 概述

有了前面的介绍，现在就可以讨论实际的远程登录进程了。

在 rlogin 客户进程中，我们把客户系统的终端行律设置为 raw 方式，以便把键入的所有数据传送给远程系统，在远程系统上运行的 vi 进程需要客户系统的终端行律处于 raw 方式。在 Linux 系统中，由客户系统键入的字符一般是由服务员系统来回显。

如果远程系统处于 raw 方式（例如在远程系统中运行 vi），那么由远程系统上的进程本身（如 vi）来完成回显工作。不管远程系统是怎样完成回显工作的，用户在终端上键入的每个字符都由客户系统经过网络传送到远程系统，然后再传送回来，回显在本地终端上。

注意，在客户系统上，rlogin 客户进程调用 fork 函数建立子进程，然后父、子进程分别处理两个方向上的数据流动，而在远程系统上只用一个服务员进程处理两个方向上的数据流动。4.3 BSD 中 rlogin 是用 select 系统调用来完成输入流的两路复用。再者，客户进程的父子进程间存在着某种形式的信息交换。

客户之所以以两个进程运行，是因为不终止子进程的情况下可能终止父进程，在终止了父进程后，用户还可以在客户系统上键入其他命令而让远程系统的输入继续显示在本地终端上。

11.5 窗口环境

Linux 用一个数据文件来记录终端的特点，在 4.3 BSD 以及早期的 System V中都使用文件 termcap，而最近版本的 System V 则使用文件 terminfo。这些文件包含有终端屏幕的大小，典型情况下终端屏幕为 24 行 80 列。但这些文件存在一个问题，那就是终端窗口的尺寸不会改变。以当前的技术情况，可以提供各种各样的方式来动态的改变终端窗口的大小。这种情况下，使用整个屏幕的软件如全屏幕编辑软件 vi，便应知道窗口的改变以重画整个屏幕。

支持可变大小多窗口的终端设备可以改变窗口的大小，这种设备不局限于位图（bitmap）显示器，如 4.3 BSD 的 window。程序可以在标准 ASCII 终端上进行多窗口操作。为了支持窗口环境，必须保留窗口的当前尺寸并且使得进程能够读到窗口大小，设置窗口大小，知道窗口大小的改变。4.3 BSD 中，可以用 iocl 调用得到储存窗口大小，如下：

```
#include <ioctl.h>
int ioctl (int fd,TIOCGWINSZ,struct winsize * winptr);
int ioctl (int fd,TIOCSWINSZ,struct winsize * winptr);
struct winsize
{
  unsigned  short     ws_row;        /*每行字符树*/
  unsigned  short     ws_col;        /*每列字符数*/
  unsigned  short     ws_xpixel;     /*水平，象素*/
  unsigned  short     ws_ypixel;     /*垂直，象素*/
}
```

内核为每个终端和伪终端维护一 winsize 结构，但不用它来做任何的事情，内核所要做的是为激活进程提供一个跟踪窗口大小的途径。当一个窗口大小改变的时候，内核产生一个 SIGWINCH 信号，该信号被发送给与此终端相关的终端进程组。如 4.3 BSD 的 vi 捕获了信号 SIGWINCH，那么 vi 便应该知道该终端窗口的大小以便把较长的行分成两行并应该

知道窗口的底行在哪里。不管窗口的大小什么时候改变，vi 都会捕获到该信号以便重画屏幕。当使用窗口环境的时候，一般都是用伪终端为每个激活的窗口提供一个注册 shell。

现在来考虑改变窗口大小对远程登录的影响，主要问题在于客户系统上改变窗口大下必须通过服务员系统。客户和服务员应采取以下步骤：

- 当用户改变窗口大小的时候，将一个特殊字节串传送给 layers 进程，该进程用于处理终端的多路复用，通过一个 RS－232 与终端通信，它也与使用伪终端在窗口下运行的进程进行通信，该进程与住 pty 相连。
- layers 进程接收到该字节串的时候就对从 pty 调用 ioctl，这样信号 SIGWINCH 便发送给了从 pty 进程组中的进程以及 rlogin 父进程。
- rlogin 父进程捕获到 SIGWINCH 信号，便调用 ioctl 得到新的窗口的尺寸，这个新的窗口尺寸又通过网络发送给 rlogin 服务员进程。
- rlogin 接收到新的窗口尺寸便对其主 pty 调用 ioctl，内核便把信号 SIGWINCH 发送给从 pty 进程组中的进程。
- 该进程捕获到信号 SIGWINCH，然后重画屏幕。

这样，本地系统和远程系统的窗口大小都改变了。

11.6 流控制与伪终端方式字

大多数终端行律都用于交互式环境，两个方向都没有字符缓冲，这样做使得进程能够成块的读写数据，而设备驱动程序则尽量的接收并尽快的把数据发送给终端。终端输出的速度通常受到连接设备速度的限制，如果显示在终端上的输出太快而无法看清，为暂停输出，拥护可以按称为暂停键的特殊字符键，需要继续读下去的时候，键下一开始键，于是终端行律便恢复输出。暂停键一般都是 CTL－S，而开始键一般都是 CTL－Q。除了输出放在缓冲区外，终端输入也由行律放在一个缓冲区中，这样在进程准备读字符前便可键入字符。

每个终端和伪终端都有一个输入队列和一个输出队列，如果行律想回显输入字符，只需要把输入字符传到输出队列。

一般来说，键入中断键和停止键都会清除输入队列和输出队列，并终止当前运行的进程。当终端行律处于 raw 方式的时候，这两个键不再特殊，也就是说，键入它们不会清除队列。

在一个远程登录系统下，远程系统的行律处理中断键：

- 远程行律把字符放在其输出队列上等着 rlogin 通过伪终端来读取它们。
- 远程系统把准备发送本地系统的数据放在缓冲区中。
- 本地系统的网络缓冲区也包含字符，等待着被 rlogin 客户进程取走。
- 本地系统的终端行律也包含要在终端上显示的字符，这个时候因为终端一般慢于网络或者产生输出的远程进程。

再者，用户在本地系统上键入中断键的时候，远程行律便进行中断处理，而用户当然希望来不及输出的东西被清除掉，而不愿意坐在终端前等着一行行的输出。

流控制最好由客户系统来处理。若是由远程系统处理的话，该字符将被传到远程系统，由远程系统的行律模块来禁止输出，但是在此之前，已经传回本地系统的字符还是会依次显示在屏幕上。在客户系统上完成流控制的问题在于无论远程系统什么时候处于 raw 方式，暂停和开始字符都得不到解释。远程系统处于 raw 方式的时候，客户还要把开始和暂停字符送给远程系统去解释，但如果远程系统不处于 raw 方式，我们就可以让本地系统去处理流控制，远程系统的行律应以某种方式通知 rlogind 开始和暂停符何时被禁止何时被允许。

如一个进程设置 TIOCFLUSH 调用 ioctl ,便会把终端行律中的输入和输出列消除光。如

果远程系统上的一个进程调用 ioctl，除了让远程系统的行律模块清除其输出缓冲外，我们还希望把网络上用于输出到本地终端上的缓冲清除掉，这是远程系统行律模块应该知道的情况，也就是 rlogind 应该知道的情况。

为了处理这些情况，4.3 BSD 伪终端设备驱动程序支持一个可选的包方式，该方式对伪终端主设备调用 ioctl，在 ioctl 中需要设置 TIOCPKT 并带一个非零参数。由伪终端从设备上的行律模块通知伪终端主设备，在它的行律模块中出现了某一事件。在该方式秒，主 pty 的每个 read 调用返回如下字节：

- 单字节 0 后面跟来自从 pty 的实际数据，0 是一个标志字节，用于指示缓冲区的剩余部分是一般的数据。
- 单字节非零值，该字节是一个控制字节，用于表明从 pty 上出现了某一事件，文件<ioctl.h>包含了该字节的常数定义。

TIOCPKT _FLUSHREAD 表明终端输入队列被清除

TIOCPKT _FLUSHWRITE 表明终端输出队列别清除

TIOCPKT_STOP 表明终端输出被终止了

TIOCPKT _START 表明终端输出重新开始

TIOCPKT _DOSTOP 表明从 pty 已经发生了变化以致于终端停止符是 CTL—S，开始符是 CTL—Q，且终端不处于 raw 方式。

TIOCPKT_NOSTOP 表明从 pty 已经发生了变化以至于终端停止键不 CTL—S 或者开始符不是 CTL—Q，或者终端处于 raw 方式。

在这种方式下，从主 pty 读的进程便能够在调用 read 之前区分一般的数据和控制信息。rlogind 服务员只对以下的三个控制信息感兴趣：

- TIOCPKT_FLUSHWRITE
- TIOCPKT_NOSTOP
- TIOCPKT_DOSTOP

当服务员从主 pty 中读到以上三个字节中的任意一个的时候，就把带外信号（out-of-band）发送给 rlogin。rlogin 子进程从网络中读数据，它准备接收带外信号（信号 SIGURG）并执行相应的动作。

1．TIOCPKT_FLUSHWRITE

由于服务员终端输出队列被清除，客户也应该尽量把来不及输出的清除掉。它先为标准输出（本地终端）设置 TIOCFLUSH 调用 ioctl 清除终端行律中的输出缓冲，然后从网络中读，直到读到带外字节并把数据丢掉。以这种方式，网络缓冲中的任何数据也被丢掉了。只要接收者读到了带外数据，它便扔掉所有的带内数据并把带带数据先发送过来。

2．TIOCPKT_NOSTOP

在这种情况下，从 pty 不再把 CTL—S 和 CTL—Q 作为停止符和开始符，或者从 pty 处于 raw 方式。在任何一种情况下，rlogin 不再完成流控制并且把所有的字符传给服务员进程，在远程系统上启动 vi 便属于这种情况。

3．TIOCPKT_DOSTOP

这种情况下，从 pty 不属于 raw 方式并停止开始符为 CTL—S 和 CTL—Q，这样就允许 rlogin 处理流控制。为此客户把本地终端的行律模块置为 cbreak 方式（而非 raw 方式）以使客户能进行流控制。用户终止 vi 时就是如此。

综上所述，远程登录系统实际上是一个远程回显，本地流控制且带有输出清除的设施。

- 远程回显是因为客户要求服务员完成回显工作。
- 终端输出的流控制是在客户系统上完成的，只要远程系统上的停止和开始符为 CTL—S 和 CTL—Q。

- 需要清除的时候，客户把来不及输出的东西清除掉。除此之外，我们还加入了把窗口变化传给服务员这一远程登录设施。

11.7 rlogin 客户程序

前面介绍了登程登录的客户，下面是其主要执行函数的框架：

```
struct tchars notc = {-1,-1,-1,.-1,-1,-1};
struct lchars noltc = {-1,-1,-1,.-1,-1,-1};
do (oldsigmask)
int oldsigmask;
{
    struct sgtyb sb;
    ioctl(0,TIOGETP,(char * )&sb);
    defflags = sb.sg_flags;
    tabflag =defflags & TBDELAY;
    defflags =ECHO|CRMOD;
    deferase =sb.sg_erase;
    defkill =sb.sg_kill;
    ioctl (0,TIOCLGET, (char * )*deflflags);
    ioctl (0,TIOCGETC,(char * )&deftc);
    notc.t_startc =deftc.t_startc;
    notc.t_stopc =deftc.f_stopc;
    ioctl (0,TIOCGETC,(char *)&defltc);
    signal (SIGINT,SIG_IGN);
    setsignal (SIGHUP,exit);
    setsignal (SIGQUIT,exit);
    if ( ( childpid = fork() ) <0)
        done (1);
    if ( childpid = = 0)        /*  子进程= = reader */
    {
        tty_mode(1);
        if ( reader(oldsigmask) = = 0)
           exit ( 0);
        sleep (1);
        exit (3);
    }
    /*  父进程 = = writer */
    signal(SIGURG,sigurg_parent);
    signal (SIGUSR1,sigusr1_parent);
    sigsetmask( oldsigmask);
    signal (SIGCHLD,sigckd_parent);
    writer( );
    done (0);
}
```

窗口变化信息是由客户以带内的方式传递给服务员，而伪终端控制信息一使带外数据

发送的。

11.8 rlogin 服务器

下面是服务员进程的主要函数代码。

```
do(cli_addrp)
struct sockaddr_in    *    cli_addrp;
{
      int I ,masterfd,slavefd,childpid;
      register struct hostent    *  hp;
      struct hostent hostent;
      char    c;

      alarm (60);
      read(0,&c,1);
      if ( c!=0)
            exit (1);
      alarm (0);
      cli_addrp->sin_port=ntohs( (u_short)cli_addrp->sin_port);
      hp=gethostbynaddr (&cli_addrp->sin_addr,
      sizeof(struct in_addr),cli_addrp->sin_family);
      if ( hp = = NULL )
      {
            hp = &hostent;
            hp->h_name –inet_ntoa(cli_addrp->sin_addr);
      }
      if (cli_addrp->sin_family !=AF_INET    ||
         cli_addrp->sin_port    >=IPPORT_RESERVED ||
            cli_addrp->sin_port    <=IPPORT_RESERVED/2)
                  fatal(0,“permission denied”);
      write (0,“ ”,1);
      for(c=‘p’;c<‘s’,c ++);
   }
   struct stat            statbuff;
   line= “/dev/ptyXY”;
   line[8]=c;
   line[9]=‘0’;
   if (stat(line,&statbuff)<0)
   break;
   for (i=0,i<16;i++)
   {
      line[9] =“0123456789abcdef”[i];
      if ((masterfd = open (line,O_RDWR))>0)
            goto gotpty;
```

```
            }
    )
gotpty:
ioctl (masterfd,TIOCSWISZ,&swin);
line[5]='t';
if ((slavefd =open (line ,O_RDWR))<0)
   fatalperror (o,line);
if (fchmod (slavefd,0))
   fatalperror(0,line);
signal (SIGHUP,SIG_IGN);
vhangup( );
signal (SIGHUP,SIG_IGN);
if (( slavefd =open (ling,O_RDWR))<0)
      fatalperror(0,line);
setup_term(masterfd);
if ( ( childpid = fork( )<0)
   fataperror(0,“ ”);
if childpid = = 0)
/*  子进程  */
{
      close (0);
      close (slavefd,0);
      dup2 (slavefd,0);
      dup2 (slavefd,1);
      dup2 (slavefd,2);
      close (slavefd);
      execl (“/bin/login”,“login”,“-r”,hp->h_name,(char * )0);
      fataperror(2,“/bin/login”);
}
else    /*  父进程  */
{
      close (slavefd);
      ioctl (0,FIONBIO,&one);
      ioctl (masterfd,FIONBIO,&one);
      ioctl (masterfd,TIOCPKT,&one);
      signal(SIGTSTP,SIG_IGN);
      signal (SIGCHLD,cleanup);
      setpgrp (0,0);
      protocol (0,masterfd);
      signal (SIGHLD,SIG_IGN);
      cleanup( );
}
```

当客户通过 TCP **端口发送来一个登录请求的时候，**inted **就激活远程登录服务员** rlogind**，由它激活标准** Linux login **程序。如果通过了检查，则由** login **进程激活用户的注**

册 shell。

代码中函数 cleanup 调用了函数 logout 和 logwtmp，第一个函数把文件 /etc/utmp 的用户入口删除，该文件为每个当前注册进入系统的用户保持一个入口项。函数 logwtmp 在文件 /usr/adm/wtmp 中增加了一个入口项，每当用户注册进入或者退出登录的时候，都在该文件中加入一个入口项。这两个文件一般都由程序 /bin/login 和/etc/init 来维持。事实上，程序 login 只完成其中一部分，这里文件终止项都由 rlogind 程序来处理。

第十二章　远程过程调用

12.1　引言

本章我们对一些技术和机制进行讨论，这些技术和机制有助于程序员使用客户机-服务器范例。我们一般性的介绍远程过程调用 RPC（Remote Procedure Call）的概念，还将描述一个工具（rpcgen），它可以为一个使用远程调用的程序生成它所需要的大部分 C 代码。最后我们给出一个完整的可以运行的例子，通过这个例子来说明 rpcgen 这个工具是如何生成一个使用远程过程调用的客户机和服务器的。

所谓过程调用，是指将控制从一个过程 A 传递到另一个过程 B，返回时过程 B 将控制进程交给调用过成 A。在目前大多数系统中，调用者和被调用者都在给定主机系统中的一个进程中，他们是在产生程序执行文件时由链接器连接起来的，这类过程调用成为本地过程调用。而远程过程调用（RPC）指的是由本地系统上的进程激活远程系统上的进程，我们将此称为过程调用是因为它对程序员来说表现为常规过程调用。我们用“请求”代表客户调用远程过程，“应答”代表远程过程将其结果返回给客户。

对于编写远程调用程序来说，首先应熟悉与远程调用有关的各个系统调用，这些系统调用在前面已经使用过很多次，但这里还要再提出一次，因为这是网络通信软件设计的基础。

远程过程调用和远程命令执行很容易混淆起来，其实它们之间的区别还是很明显的。远程命令执行是由远程主机开始一个命令程序的执行，而远程过程调用是由本地主机开始一个程序的执行，在程序执行的中间调用远程主机提供的过程。处理远程过程调用的进程有两个，一个是本地客户进程，另一个是远程服务器。对本地进程来说，远程过程调用表现为对客户进程的过程控制，然后由客户进程生成一个消息，通过网络系统调用发往远程服务器。网络信息中包括过程调用所需的参数，远程服务器接到信息后调用相应过程，然后将结果通过网络发回客户进程，再由客户进程将结果返回给调用进程。因此，远程系统调用对调用者表现为本地过程调用，但实际上是调用了远程系统上的过程。

远程系统调用 RPC 使用了若干种不同的传送协议，不了解它们，在使用不同传送协议的系统之间，就无法实现远程过程调用。这里介绍了三种传送协议：Sun RPC,Xerox Courier 和 Apollo RPC。

对于不同的传送协议，应有不同的通信处理方法，因此对于上面三种传送协议要提供三种不同的程序。

12.2　远程过程调用模型

我们在描述客户机／服务器程序时，不能只分别查看客户机和服务器的各个构建（Component）的结构，我们在构建一个客户机／服务器软件时，我们不能一次关注于其中的一个构建。我们必须考虑整个系统是如何交互的，以及各个构建之间应该如何交互。

为了帮助程序员理解客户机／服务器的交互，研究人员已经为构建分布程序设计出一套概念性的框架。该框架被称为远程过程调用模型（Remote Procedure Call Model）或 RPC 模型，它把我们所熟悉的来自传统程序的概念作为分布式应用的基础。

远程调用模型主要来自于传统语言中的过程调用机制。过程调用提供了一个强有力的抽象，它允许程序员将一个程序划分成一些小的，可管理的，易于理解的片段。过程特别有用，因为它具有一个能给出程序执行的概念性模型的简单明了的实现。下图说明了这个

概念。

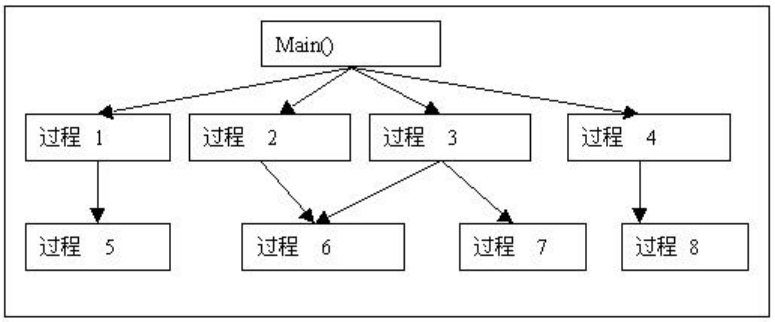

图 12-1 过程概念

过程概念：一个传统的程序由一个或多个过程所组成。它们往往按照一种调用等级来安排。从过程 n 到过程 m 的箭头代表由从 n 到 m 的调用。

远程过程调用模型使用了和传统过程一样的抽象。只是，它允许一个过程的边界跨越两台计算机。图 12-2 表示了远程调用过程模型如何将一个程序划分成两片，每片在一个单独的计算机上执行。当然，远程过程调用不能有一台计算机传递到另一台上。在程序可以使用远程过程调用之前，必须加入允许程序与远程过程通信的协议软件。

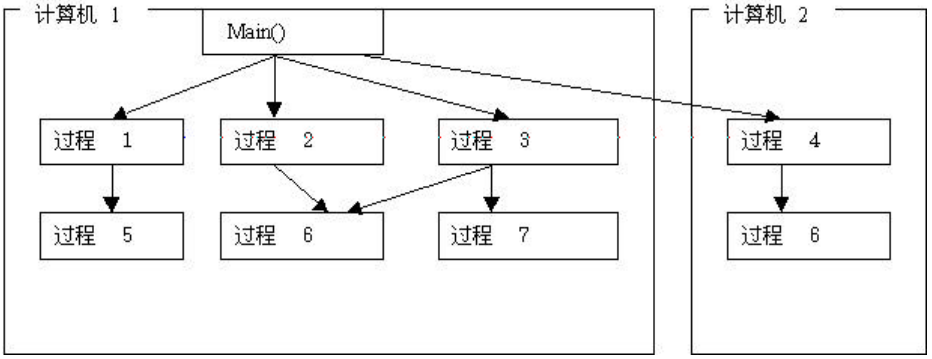

图 12- 2 分布式程序

图 12-2 是一个分布式程序，它说明了怎样从一个调用本地过程的程序扩展成使用远程过程调用的程序。划分发生在主程序和过程四之间。实现远程过程调用需要有一个通讯协议。

12.3 传统过程调用和远程过程调用的比较

一般程序的过程模型提供了一个程序执行的概念性解释，它可以直接扩展到远程过程调用上。让我们考虑一下内存中那些以编译好的程序代码在执行中的顺序，这将有助于帮助我们理解这个概念。图 12-3 展示了控制流从主程序传到两个过程，然后返回。

计算机从一个主程序 main 开始执行，它将一直执行下去，直到遇到第一个过程调用。这个调用使执行转入到某个指定的过程代码并继续执行。如果它遇到了另一个调用，计算机便转入到第二个调用。

程序继续在所有调用的过程中执行，直到它遇到了 return 语句。这个返回使程序恢复到紧挨着最后一个调用之后的那个位置。任意的时候，只有一个执行在继续。因此，在计算机执行对一个过程的调用时，另外一个过程必须暂时停止。计算机挂起调用进程，在调用过程中，将这个过程的所有变量的值又可以使用了。一个被调用的进程可以进一步使用过程调用，因为计算机记住了调用的次序，而且总是返回最近执行的调用者。

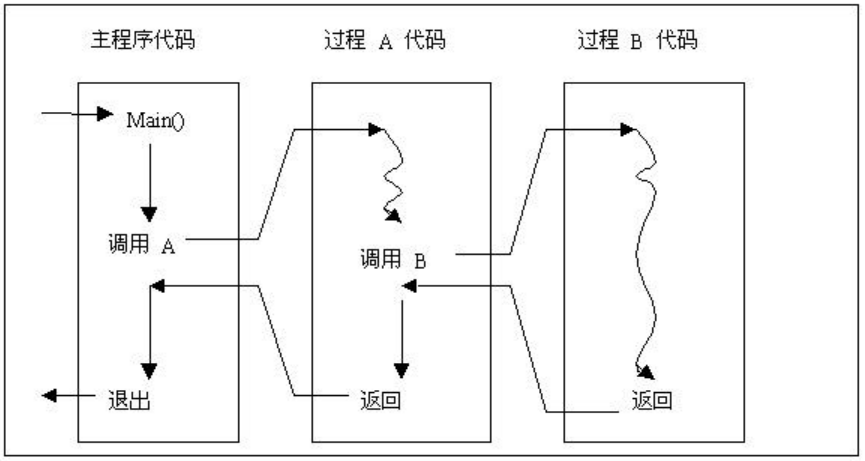

图 12-3　普通程序的执行过程

下面我们来考虑远程过程。传统过程调用可以帮助我们来理解它。我们现在不考虑一个客户机程序和服务器程序交换消息，而是想象每个服务器实现了一个（远程的）过程，而且，客户机和服务器之间的交互对应于过程的调用和返回。由客户机发送给服务器的请求对应与一个远程过程调用，而由服务器返回的信息，对应与一个过程的返回指令执行。图 12-4 展示了这个比喻。

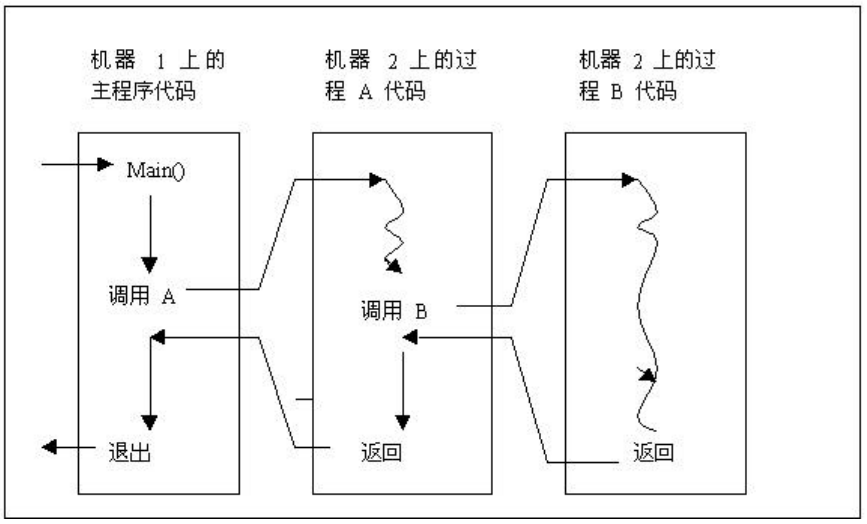

图 12-4　远程过程调用的执行模型

图 12-4 给出了远程过程调用的执行模型。单个执行顺序是建立在分布式环境下的。服务器和服务器间的尖头代表一个远程过程调用时，控制权是如何由一个客户机传递给一个服务器的，以及它又是如何在服务器响应后返回给客户机。

远程过程调用概念提供了一个强有力的类比方法，它允许程序员以一种他所熟悉的环境来思考客户机和服务器的交互。如同一个传统的过程调用，一个远程过程调用把控制权传递给被调用的过程。也像传统的过程调用一样，在调用进行中，系统把调用过程的执行挂起，而只允许被调用过程执行。

当一个远程程序发出一个响应的时候，对应与在传统过程调用执行一个 return。控制权返回给被调用者，被调用过程停止执行。嵌套的过程调用的想法也可应用到远程过程调用。一个远程过程也许要调用另一个远程过程。如上图所展示的，嵌套的远程过程调用对

应与这种情况，既一个服务器变成了另一个服务器的客户机。

当然，远程过程调用和客户机／服务器之间交互的细节不是一个这个类比所能够说明白的，例如：一个传统的过程会一直保持在完全不活动的状态，直到有调用它的行为。而远程过程调用，服务器在没有请求的时候必须在那里一直等待接受请求。更大的不同来自与数据流传向远方的方式。传统的过程调用只能接受几个参数，返回也仅仅可以返回少数几个结果。而远程过程调用可以接受或返回任意数量的数据。

如果本地调用和远程过程调用的行为是一致的，这样当然是理想的。但是，许多实际的约束却使他们不能这样。

- 首先，网络的延时会使一个远程过程调用的开销远远比本地调用要大。
- 其次，传统的过程调用因为被调用过程和调用过程运行在同一块内存空间上，可以在过程间传递指针。而远程过程调用不能够将指针作为参数，因为远程过程与调用者运行在完全不同的地址空间中。
- 再次，因为一个远程调用不能共乡调用者的环境，所以它就无法直接访问调用者的 I/O 描述符或操作系统功能。例如远程过程调用不能在其调用者的标准错误记录文件中直接写入错误信息。

12.4 远程过程调用的定义

Sun Microsystem 公司定义了一个特定形式的远程过程调用，它称为 Sun RPC，开放网络计算 (Open Network Computing，ONC) RPC，或简称 RPC。ONC 远程过程定义也被业界所广泛接受。它已经被用来做了许多网络软件的应用机制，其中包括网络文件系统（NFC Network File System）。

ONC RPC 定义了调用者（客户机）发出的调用服务器中的某个远程过程的格式，参数的格式，以及被调用过程返回给调用者的结果的格式。它允许调用程序使用 UDP 或 TCP 来装载报文，它利用 XDR 来表示过程的参数以及 RPC 报文首部中的其他条目。最后，除了协议说明之外，ONC RPC 还包括一个编译系统（rpcgen），它帮助程序员自动建立分布式程序。

12.5 远程过程调用的有关问题

为了使 RPC 的使用和调用过程一样，应该先解决一些问题，下面分别予以讨论。

1．参数传递

在客户进程和服务器之间的参数传递是可见的，这样，如果参数通过指针引用就会出现问题，这是因为服务器无法访问客户系统上的内存单元。一个典型的解决方法是 RPC 协议只允许客户进程传递值参，对于每个远程进程，可以定义一定的格式，如预先定义输入参数以及返回值的格式是什么。

2．传送协议

有些 RPC 实现仅使用单个传送层协议，有些允许选择，本书中使用的是下面一些协议：

- Sun RPC（UDP,TCP）
- Xerox Courier（SPP）
- Apollo RPC（UDP,DDS）

其中 Sun RPC 可用于面向连接的或非连接的协议，Xerox courier 仅用于面向连接的协议，Apollo RPC 仅用于非连接协议。

使用非连接协议的时候，client stub 要考虑到丢失信息的问题，而面向连接的协议则无须考虑这个问题，但另一方面，使用面向连接的协议的开销要大的多。

3．数据表示

对于一个本地进程，客户进程和服务器在同一个系统上执行，因此不存在数据不兼容的问题，但对于远程过程调用 RPC，如果客户进程和服务器位于不同体系结构的系统上，则必须有数据转换。

本书讨论的所有实现均通过给 RPC 实现支持的数据类型定义一个或者多个标准格式来处理这个问题。我们必须考虑所有的数据项，而不仅仅是二进制，客户在使用 ASCII 的系统上而服务器在使用 EBCDIC 的系统上也应该是可行的。我们可以参考一下本书中的其它例子是如何处理的。

- TCP/IP 协议组在各种协议头中的所有 16 位和 32 位域使用高字节在后的字节顺序，诸如 32 位的网络和主机 ID 以及 16 位的 UDP 端口号域等。
- XNS 协议组也使用 16 位和 32 位域的高位字节在后的字节顺序。
- 4.3 BSD 打印机假脱机系统将其二进制研制在单字节内，不存在字节序的问题，对打印机名、文件名、文件大小和登录名等使用 ASCII。
- 远程登录客户和服务器在处理窗口大小的协议中的 16 位窗口大小域使用高字节在后的字节序。
- 4.3 BSD rmt 服务器在除“返回状态”命令外的所有命令中使用 ASCII 串，这个特殊命令在不同 UNIX 系统间也不能移植，应完全避免使用。该服务器对读写到磁带的数据没有限制。

12.5.1　远程过程调用传送协议

现在我们比较一下本书中将要讨论的三个 RPC：

- Sun RPC。它使用的数据表示标准是 XDR，XDR 使用高字节在后的字节序，任何域的最小尺寸是 32 位。例如，当一个 VAX 客户进程要将一个 16 位整数传递给同样运行在 VAX 上的服务器时，该 16 位数首先由客户转换成 32 位高字节在后的整数，然后由服务器转换回低字节在后的 16 位整数。
- Xerox Courier。它使用的标准是高位字节在后的字节序，任何域的最小尺寸是 16 位。字符数据编码成 16 位的 Xerox NS 字符集，该字符集对于常规字符使用 ASCII，对于其它的专用字符集如希腊语使用转义符。例如，当给某一打印机发送数学文本的时候使用此字符集。
- Apollo RPC。Apollo NDR 并不只是使用单个网络标准，而是支持多个格式。如果用户的内部格式是其支持的格式之一，它允许发送者使用其自己的内部格式，然后，如果与发送者的格式不同，接受者将其转换成自己的格式。

这三种 RPC 实现所支持的数据类型和格式如表 12-1。

表 12-1　三种 RPC 实现所支持的数据类型和格式

数据类型	Sun RPC	Xerox Courier	Apollo NDR
8 位逻辑	支持	支持	支持
16 位逻辑			
32 位逻辑			
8 位带符号整数	支持	支持	支持
8 位无符号整数			支持
16 位带符号整数		支持	支持
16 位无符号整数		支持	支持
32 位带符号整数	支持	支持	支持
32 位无符号整数	支持	支持	支持

64 位带符号整数 64 位无符号整数	支持 支持		支持 支持
字节序	高字节在后	高字节在后	高字节在后 低字节在后
带符号整数格式	补码	补码	补码
32 位浮点数 64 位浮点数	支持 支持		支持 支持
浮点数格式		IEEE	IEEE,VAX IBM 或 Cray
字符类型	ASCII	16 位 NS	ASCII 或 EBCDIC
枚举 结构(记录) 固定长一维数组 变长一维数组 固定长多维数组 变长多维数组 联合 固定长隐含数组 变长隐含数组	支持 支持 支持 支持 支持 支持 支持	支持 支持 支持 支持 支持 支持	支持 支持 支持 支持 支持 支持 支持 支持 支持

注：空表示不支持

所有这三种均使用隐含类型，也就是说，通过网络传送的只是变量的值而不是变量的类型。而成为 ASNI 的 ISO 数据表示技术使用的是显式类型，在发送每个域值的同时也发送消息中每个数据域的类型，

例如，当 Courier 发送一个 32 位 整数值时，不管是作为一个过程参数还是作为一个过程的结果，只有该 32 位值在网络上传送。而 ASNI 则会传送一个 8 位字节，指明下一个值是一个整数，其后所跟的另一个字节说明该整数域的长度（字节数），然后再跟一个，两个，甚至五个字节的实际整数值。

12.5.2 Sun RPC

这里描述的版本是“RPCSRC3.9”，实际的 RPC 协议是 Version 2。Sun RPC 由以下几个部分组成：

- Rpcgen，该部分是定义远程过程接口、产生 client stub 和 sever stub 的编译器。
- XDR（外部数据表示），是使在不同系统间可移植的方式编码数据的标准方法。

12.5.3 Xerox Courier

Courier 既是协议又是一个说明语言,它主要用于 UNIX 和其他 Xerox 系统通知如 Xerox 打印服务器或 Xerox 文件服务器。

现在使用 Courier 给出前一节中的客户和服务器程序，我们必须编写三段代码：

- RPC 说明，用 Courier 语言
- 客户程序，用 C 语言
- 服务器程序，用 C 语言

Courier 远程过程以三种方法返回：

- 常规返回
- 远程过程拒绝执行的拒绝返回
- 远程过程产生的中止返回

在上面的例子中是常规返回，拒绝返回和中止返回可由客户程序处理，如果客户程序不处理这些错误，则会终止客户程序。

我们现在讨论使用 Courier 执行一远程过程的步骤。支持 courier RPC 的任何 UNIX 系统必须运行一个守护进程以在一公认的 SPP 端口（端口 5）上监听客户进程的连接请求，在编译和连接了服务器程序之后，我们应当告知 Courier 守护进程某服务器可以被激活，为此，我们必须在文件 /etc/Courierservices 中增加一项以指明所执行文件的程序号、版本号和路径名。

Xnscourierd 守护进程通常是在系统启动时启动的，其执行步骤如下：

- 客户调用 CourierOpen 打开与指定主机的连接，由此建立起与远程主机上端口 5 的 spp 连接，在此端口上 xnscourierd 进程监听客户请求。
- 守护进程产生一个子进程来处理该连接，子进程从连接上读出前几个字节确认是一个 Courier 客户后，等待其发出 RPC 请求，指明被调用的程序，版本和过程。
- 客户调用 Bindate 函数的时候，client stub 向连接写入合适的程序号、版本号和过程号。
- 子进程从连接读出这个信息，然后读文件 / etc / Courierservices 以确定执行哪个服务器程序，接着由子进程执行相应的服务器程序。注意，子进程已经从连接读取了过程号，在调用服务器程序时必须将此过程号传递给服务器程序，这个过程号是以 exec 命令行中的形式传递给服务器程序的，由服务器程序中的 server stub 确定要调用哪两个远程过程。
- 启动 dateserver 程序，由 server stub 的 main 函数处理其命令行参数以确定调用哪个远程过程，然后调用 BinDate 函数，函数返回时，server stub 将结果转换成 Courier 标准形式并将它们送回客户。
- 接着客户进程调用 StrDate 函数。Client stub 通过连接发送程序号、版本号和过程号，dateserver 进程中的 server stub 职别到程序和版本没有改变，就调用 StrDate 函数，调用前 server stub 从连接读取参数并将它们从 Courier 格式转化成函数要求的格式；若是一不同程序或不同版本的请求，server stub 就 必须执行对应的程序来处理这一新的请求。
- 函数返回 server stub 的时候，它转换返回值并将它们发送回客户，最后由客户调用 Courier Close 关闭 SPP 连接，服务器进程正常终止。

Courier 为客户和服务器之间的网络连接使用的是 XNS 顺序分组协议（SPP），因为这是一个面向连接的协议，故对客户和服务器之间交换的数据量没有限制。Courier 使用 SPP 头中的消息结束标志以及数据流类型域。

12.5.4　Apollo RPC

网络计算体系机构（NCA）是 Apollo 的远程计算系统结构，尽管此体系结构已公开发表，但称之为 NCS （Network Computing System ）的 Apollo 仍是一特权产品。NCS 分为以下几部分：

- NIDL（网络接口定义语言。有一个编译器将此语言转化为 server stub 和 client stub。
- NDR （网络数据表示），它定义了传递所支持的数据类型所使用的标准形式。
- 运行时间库。

NCA RPC 允许任意数目的参数和返回值，我们可以指定每个参数是输入（从客户传递给服务器），输出（从服务器返回给客户）或既是输入又是输出。

Apollo RPC 运行于非连接的传递协议，通常使用的是 UDP 或 DDS （Appolo 的特权网络协议）。如果网络消息太长而不能放入一个网络分组中时，Apollo 实现要对网络消息进行分段和组装，UDP 的上限是 64K 字节，而 Xerox IDP 的上限是 546 字节。

实现使用的是完全的套接字地址，而不只是某个协议的端口号，并且标识特定服务器的客户句柄含有服务器的整个套接字地址，也就是说能够以与协议无关的方式向客户进程

写，因为完全的套接字地址是每个调用的句柄的一部分，客户在将其句柄加载到特定服务器之前不需要知道正在使用什么传送协议，即便句柄加载的一个服务器，句柄对客户进程来说也是一个非透明的结构。

12.6 stub 过程简介

在我们将一个传统程序改写成 RPC 程序时，我们为了实现 RPC 而加入程序的附加过程被称为 stub 过程。理解 stub 过程的最简单的方法是设想一个传统的程序，它被分割成两个程序，一个已有的过程被转移到一个远程机器中。在远程过程（服务器）一端，stub 过程取代了调用者。这 stub 实现了远程过程调用所需要的所有的通信。因为 stub 与原来的调用使用了一样的接口，因此，增加这些 stub 既不要求更改原来的调用过程，也不要求更改原来的被调用过程。图 12-5 展示了 stub 的概念，它说明了 stub 这个过程是怎样分离出本地过程和远程部分的。

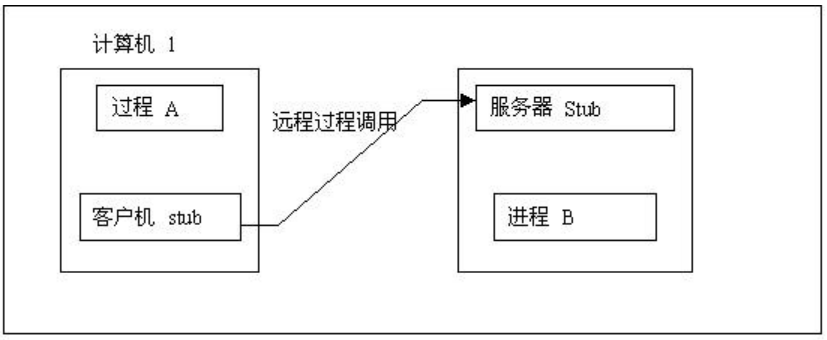

图 12-5 stib 过程

加入到程序中的 stub 过程实现了一个远程过程调用。因为 stub 与原来的调用使用了一样的接口，因此，增加这些 stub 既不要求更改原来的调用过程，也不要求更改原来的被调用过程。

12.7 rpcgen 简介

因为实现一个 RPC 服务器所需要的多数代码是不变的，例如，如果远程过程过程号与客户机端 stub 过程之间的映射被保存在一个数据结构中，那么所有的服务器都可以使用相同的分派进程。与此类似，所有的服务器都可以使用相同的代码来向端口映射器注册。

为了避免不必要的编程，ONC RPC 的实现包含一个工具，它自动生成实现一个分布市程序所需要的大部分代码。这个工具叫做 rpcgen，它读取一个规格说明文件作为输入，生成 C 的源程序作为输出。规格说明文件包含常量，全局数据类型，全局数据，以及远程（过程包括过程参数和结果类型）的声明。Rpcgen 产生的代码包含了实现客户机和服务器（它提供指明的远程过程调用）程序所需要的大部分源代码。具体的说，rpcgen 为客户机端和服务器端生成 stub 过程，它包括参数整理，RPC 报文，把调用分配到正确的过程，发送应答，在参数和结果的外部表示和本地数据表示之间进行转换。Rpcgen 的输出在与一个应用程序和程序员编写少数文件相结合后，便产生了完整的客户机和服务器程序。

RPC 是一个使用性很广的概念。一个程序远可以使用 RPC 来帮助指明或实现一个分布式程序。在使用 ONC RPC 时，程序员可以选择各种方式，如，在白手起家构建代码时按照 RPC 的规格说明进行，使用在 RPC 库中所找到的过程，或者使用一个称为 rpcgen 的程

序自动生成工具。

在下面的几节中，我们将通过一个例子来讲解 RPC 的具体使用。

12.8　分布式程序生成的例子

我们下面举一个例子来看看 rpcgen 是如何工作的。

下面的程序是一个简单的查找字典的程序。该字典提供四个基本操作：

- 初始化（initialize），它初始化数据库（既清除以前存储的所有信息）；
- 插入（insert），它插入一条新的信息；
- 删除（delete），它删除一个条目；
- 查找（look up），它寻找某个条目。

我们假设数据库中的每一个条目都是单个的单词，接着使用数据库来检查新单词，以便知道没个单词是否在字典中。输入的指令我们定义为每行包含一个单字母的命令，后面跟着一个单词参数。表 12-2 为我们命令参数的说明表：

表 12-2　命令参数说明

单字母命令	参数	含义
I	无	通过删除所有单词来初始化数据库
I	Word	把 word 插入到数据库中
D	Word	把 word 从数据库中删除
L	Word	在数据库中查找 word
Q	无	退出

我们可以通过 I 命令先初始化数据库，然后使用 i 命令来插入几个单词，通过 l 命令来查找数据库中是否存在。

12.8.1　我们如何能够构造出一个分布式应用程序

作为一个程序员，我们可以按照下面的顺序来编写分布式应用程序：

●构建一个解决该问题的常规应用程序

●选择一组过程，以便将这些过程转移到一个远程机器中，通过这个方法将程序分解。

●为远程过程程序编写一个 rpcgen 规格说明，包括远程过程调用的名字极其编号，还有对其参数的声明。选择一个远程程序号和一个版本号（通常为 1 ）。

●运行 rpcgen 来检查该规格说明，如何合法，便生成四个源代码文件，这些文件将在客户机和服务器程序中使用。

●为客户机端和服务器端编写 stub 接口例程。

●编译并连接客户机程序。它主要由四个主要文件构成：最初的应用程序（远程过程被从中删除了的那个），客户机端的 stub （由 rpcgen 生成），客户机端的接口 stub 以及 XDR 过程（由 rpcgen 生成）。当所有的文件都被编译和连接到一起后，最终的可执行的程序就是客户机。

●便宜并链接服务器程序。它由四个重要文件构成：由最初的应用程序得来的过程，他们现在构成了远程程序，服务器端的 stub （由 rpcgen 生成），服务器端的借口 stub 以及 XDR 过程（rpcgen 生成的）。当所有的这些文件被编译和链接到一起后，最终的可执行程序就是服务器。

●在远程机器上启动服务器，在本地机器上启动客户机。

下面我们将详细的解释每一个步骤。

1．构建一个常规的应用程序

要构建这个字典应用例子，我们首先来写出实现这个字典的常规版本。

```
/* dict.c 包含 main，initw，netxin，insertw，deletew，lookupw 函数 */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* 每一个输入的命令的最大长度 */
#define MAXWORD 50
/* 整个字典中可以包含的最多字数 */
#define DICTSIZ 100
/* 字典的数据存取数组 */
char dict[DICTSIZ][MAXWORD+1] ;
/* 字典中字母数 */
int    nwords = 0 ;
int nextin ( char* cmd, char* word) ;
int initw() ;
int insertw ( const char* word ) ;
int deletew ( const char* word ) ;
int lookupw ( const char* word ) ;
/* 主函数,进行各种操作 */
int main ( int argc, char* argv[] )
{
    /* 从命令行取得命令的变量 */
    char word [ MAXWORD + 1 ] ;
    char cmd ;

    /* 输入单词的长度 */
    int    wrdlen ;
    while ( 1 )
    {
        wrdlen = nextin ( &cmd, word ) ;
        if ( wrdlen < 0 )
        {
            exit ( 0 ) ;
        }
        switch ( cmd )
        {
            case 'I' :
            /* 初始化 */
                initw () ;
                printf ( "Dictionary initalized to empty. \n" ) ;
                break ;
            case 'i' :
            /* 插入操作 */
                insertw ( word ) ;
```

```
                printf ( "%s inserted. \n", word ) ;
                break ;
            case 'd' :
            /*  删除操作  */
                if ( deletew(word) )
                {
                    printf ( "%s deleted. \n", word ) ;
                }else
                {
                    printf ( "%s not found. \n", word ) ;
                }
                break ;
            case 'l' :
            /*  查找  */
                if ( lookupw(word) )
                {
                    printf ( "%s was found. \n", word ) ;
                }else
                {
                    printf ( "%s was not found. \n", word ) ;
                }
                break ;
            case 'q' :
            /*  退出  */
                printf ( "program quits \n" ) ;
                exit ( 0 ) ;
            default :
            /*  无法识别的命令  */
                printf ( "command %c invalid. \n", cmd ) ;
                break ;
        } /* switch 结束  */
    }
}
/*********************************************************************
    nextin 函数: 读取输入的命令和命令后面跟的单词参数          *
 *********************************************************************/

int nextin ( char* cmd, char* word )
{
    int    i, ch ;

    ch = getc ( stdin ) ;
    while ( isspace(ch) )
    {
```

```
            ch = getc ( stdin ) ;
        }
        if ( ch == EOF )
        {
            return -1 ;
        }
        *cmd = (char) ch ;
        ch = getc ( stdin ) ;
        while ( isspace(ch) )
        {
            ch = getc ( stdin ) ;
        }
        if ( EOF == ch )
        {
            return -1 ;
        }
        if ( '\n' == ch )
        {
            return 0 ;
        }
        i = 0 ;
        while ( !isspace(ch) )
        {
            if ( MAXWORD < ++i )
            {
                printf ( "error: word too long\n" ) ;
                exit ( 1 ) ;
            }
            *word++ = ch ;
            ch = getc ( stdin ) ;
        }
        *word++ = 0 ;
        return i ;
}

/*************************************************************
        initw    初始化字典，清空所有数据                    *
 *************************************************************/
int initw()
{
        nwords = 0 ;
        return 1 ;
}
```

```
/*************************************************************
      insertw    将一个单词插入字典                              *
 *************************************************************/
int insertw ( const char* word )
{
      strcpy ( dict[nwords], word ) ;
      nwords ++ ;
      return nwords ;
}

/*************************************************************
      insertw    在字典中删除一个单词                            *
 *************************************************************/
int deletew ( const char* word )
{
      int    i ;

      for ( i=0 ; i< nwords ; i++ )
      {
            if ( 0==strcmp(word, dict[i]) )
            {
                  nwords -- ;
                  strcpy ( dict[i], dict[nwords] ) ;
                  return 1 ;
            }
      }
      return 0 ;
}

/*************************************************************
      insertw    在字典查找一个单词                              *
 *************************************************************/
int lookupw ( const char* word )
{
      int    I ;

      for ( I=0 ; nwords > I ;I++ )
      {
            if ( 0 == strcmp(word, dict[I]) )
            {
                  return 1 ;
            }
      }
      return 0 ;
```

}

为了使应用程序简单并且便于理解，文件 dict.c 中的这个常规程序使用了二维数组来存储单词。一个全局变量 nwords 记录了任何时候字典中的单词数量。著称许包含一个循环，在每次循环中读取并处理输入文件总的一行。它调用过程 nextin 从下一输入行中读取一个命令（可能还包含一个单词），接着使用一个 C 语言的 switch 语句在六种可能的情况中选择其一。这些情况对应与五个有效命令再加上一个处理非法输入的默认情况。

主程序对每中情况都调用一个过程来处理细节。例如，对应于一个插入命令 I，它调用过程 insertw。过程 insertw 在数组的末尾插入一个新的单词，并将 nwords 加 1。

其他过程也按照我们所期望的那样进行操作。过程 deletew 寻找所要删除的单词，如果找到了这个单词，便用字典中的最后一个单词取代它，然后将 nwords 减 1。而 lookupw 循序查找数组，以便确定字典中是否有该指定的单词。若有则返回 1，否则返回 0。

为产生应用程序的二进制文件，我们可以调用 C 编译器。在 Linux 中可以这样：

```
gcc –o dict dict.c
```

从文件 dict.c 中的源程序产生一个命为 dict 的可执行文件。'

2．将程序划分成两部分

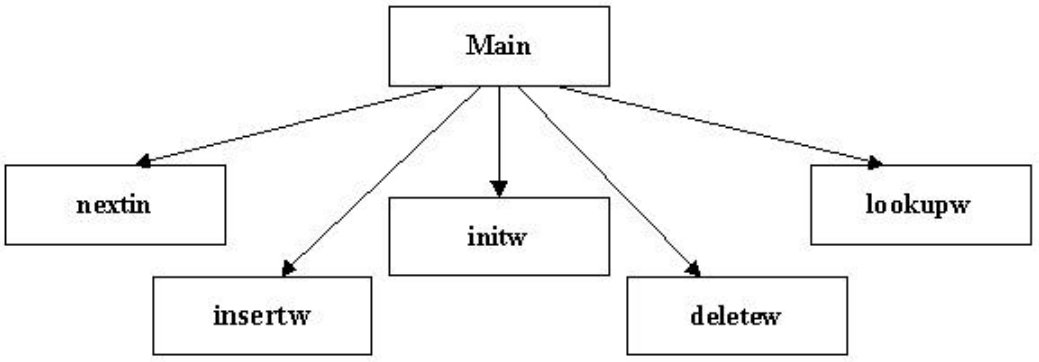

图 12-6 dict 应用过程的组织情况

常规程序一旦构建完成并经过测试，就可以将它划分成本地构建和远程构建了。在我们开始划分程序之前，我们必须有一个程序过程调用的概念模型。图 12-6 展示了我们的 dict 应用过程的组织情况。

在考虑哪个过程可以转移到远程机器上时，程序员必须考虑每个过程所需要的设施。例如：过程 nextin 在每次调用的时候要从标准输入读取下一行的命令，并对它进行分析。因为它需要访问标准输入的文件描述符，所以 nextin 必须被放在主程序中。

注意：执行 I/0 或者访问文件描述符的过程不能轻易的转移到远程机器中。

我们还必须考虑每个过程所要访问的数据所处的位置。例如，过程 lookupw 需要访问全部单词数据库。如果执行 lookupw 的机器不同于字典所处的机器，对 lookupw 的 RPC 调用就必须将整个字典作为参数来传递。

这种将巨大的数据结构作为参数传递的远程过程调用的效率是非常低下的，因为 RPC 必须为每个远程过程调用对整个数据结构进行读取和编码。一般来说，执行过程的机器应当与放置过程所要访问数据的机器是同一个。将巨大的数据结构传递给远程过程的效率是非常低的。

我们在考虑了最初的字典应用程序以及每个过程所访问的数据之后，很明显应当把过程 insertw，deletew，initw，lookupw 和字典本身放到同一台机器中。

假设我们决定将字典的存储以及相关的过程转移到一台远程机器中。为理解这么做的后果，我们要在穿件一个分布式程序和数据结构的形象，图 12-7 说明了数据及对它的访问

过程转移到一台远程机器后，这个字典应用程序的新结构：

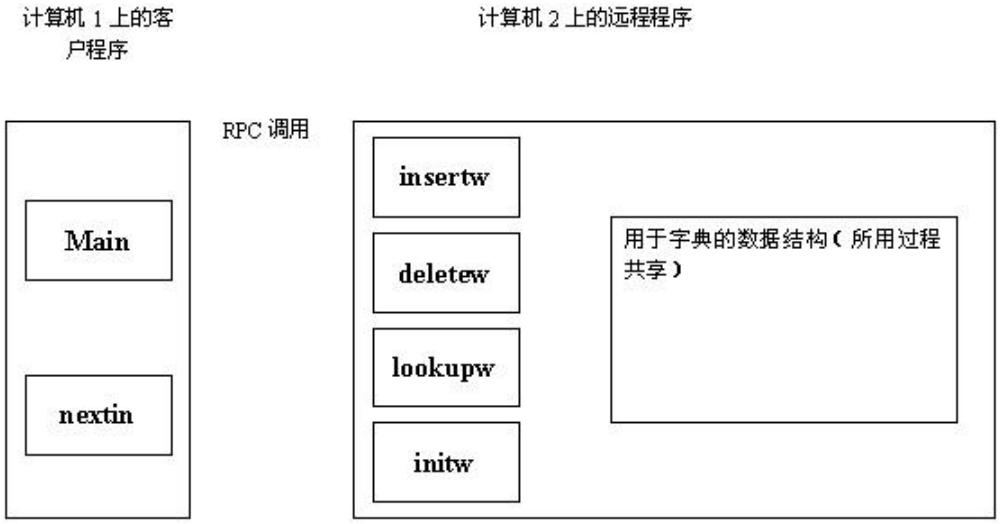

图 12-7　dict 远程程序的结构

图 12-7 可以帮助我们思考对程序的划分（划分为本地构建和远程构建）。我们必须要考虑每个过程是否要访问数据以及它所需要的服务，还必须考虑每个远程过程所要求的参数以及在网上传送这些信息所带来的开销。最后，该图还能帮助我们了解网络延时将如何旖旎感想程序的性能。

下面我们将源程序划分为两个构件。我们需要明确的划分出每个构件所使用的常量和数据结构，将每个构建放置到一个单独的文件当中。在本字典的例子中，划分是简洁明了的，因为最初的源文件可以在过程 nextin 和 initw 之间进行划分。文件 dict1.c 含有主程序和过程 nextin：

```
/* dict1.c  包含 main 和 nextin 过程 */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* 每一个输入的命令的最大长度 */
#define MAXWORD 50
int nextin ( char* cmd, char* word) ;
int initw() ;
int insertw ( const char* word ) ;
int deletew ( const char* word ) ;
int lookupw ( const char* word ) ;
/* 主函数,进行各种操作 */
int main ( int argc, char* argv[] )
{
    /* 从命令行取得命令的变量 */
    char word [ MAXWORD + 1 ] ;
    char cmd ;
    /* 输入单词的长度 */
```

```
int    wrdlen ;

while ( 1 )
{
      wrdlen = nextin ( &cmd, word ) ;
      if ( wrdlen < 0 )
      {
            exit ( 0 ) ;
      }
      switch ( cmd )
      {
            case 'I' :
            /*  初始化  */
                  initw () ;
                  printf ( "Dictionary initalized to empty. \n" ) ;
                  break ;
            case 'i' :
            /*  插入操作  */
                  insertw ( word ) ;
                  printf ( "%s inserted. \n", word ) ;
                  break ;
            case 'd' :
            /*  删除操作  */
                  if ( deletew(word) )
                  {
                        printf ( "%s deleted. \n", word ) ;
                  }else
                  {
                        printf ( "%s not found. \n", word ) ;
                  }
                  break ;
            case 'l' :
            /*  查找  */
                  if ( lookupw(word) )
                  {
                        printf ( "%s was found. \n", word ) ;
                  }else
                  {
                        printf ( "%s was not found. \n", word ) ;
                  }
                  break ;
            case 'q' :
            /*  退出  */
                  printf ( "program quits \n" ) ;
```

```
                exit ( 0 ) ;
            default :
            /*  无法识别的命令  */
                printf ( "command %c invalid. \n", cmd ) ;
                break ;
        } /* switch 结束  */
    }
}

/*************************************************************************
    nextin 函数:  读取输入的命令和命令后面跟的单词参数          *
 *************************************************************************/
int nextin ( char* cmd, char* word )
{
    int    i, ch ;

    ch = getc ( stdin ) ;
    while ( isspace(ch) )
    {
        ch = getc ( stdin ) ;
    }
    if ( ch == EOF )
    {
        return -1 ;
    }
    *cmd = (char) ch ;
    ch = getc ( stdin ) ;
    while ( isspace(ch) )
    {
        ch = getc ( stdin ) ;
    }
    if ( EOF == ch )
    {
        return -1 ;
    }
    if ( '\n' == ch )
    {
        return 0 ;
    }
    i = 0 ;
    while ( !isspace(ch) )
    {
        if ( MAXWORD < ++i )
        {
```

```
                printf ( “error: word too long\n” ) ;
                exit ( 1 ) ;
            }
            *word++ = ch ;
            ch = getc ( stdin ) ;
        }
        *word++ = 0 ;
        return i ;
}
```

文件 dict2.c **包含了来自最初的应用程序中的一些过程，他们将成为远程程序中的一部分。另外，它还要包含对各个过程要共享的全局数据的声明。在这里，文件还没有成为真正的远程程序，我们只是将这个程序简单的分为两部分。**

```
/* dict2.c 包含 initw, insertw, deletew, lookupw 过程 */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* 每一个输入的命令的最大长度 */
#define MAXWORD 50
/* 整个字典中可以包含的最多字数 */
#define DICTSIZ 100
/* 字典的数据存取数组 */
char dict[DICTSIZ][MAXWORD+1] ;
/* 字典中字母数 */
int    nwords = 0 ;
int    nextin ( char* cmd, char* word) ;
int    initw() ;
int    insertw ( const char* word ) ;
int    deletew ( const char* word ) ;
int    lookupw ( const char* word ) ;
/************************************************************
      initw    初始化字典，清空所有数据                    *
 ************************************************************/
int    initw()
{
      nwords = 0 ;
      return 1 ;
}
/************************************************************
      insertw    将一个单词插入字典                        *
 ************************************************************/
int    insertw ( const char* word )
{
```

```
    strcpy ( dict[nwords], word ) ;
    nwords ++ ;
    return nwords ;
}
/************************************************************
    deletew    在字典中删除一个单词                          *
 ************************************************************/
int   deletew ( const char* word )
{
    int   i ;

    for ( i=0 ; i< nwords ; i++ )
    {
        if ( 0==strcmp(word, dict[i]) )
        {
            nwords -- ;
            strcpy ( dict[i], dict[nwords] ) ;
            return 1 ;
        }
    }
    return 0 ;
}
/************************************************************
    lookupw    在字典查找一个单词                            *
 ************************************************************/

int   lookupw ( const char* word )
{
    int   i ;

    for ( i=0 ; nwords > i ;i++ )
    {
        if ( 0 == strcmp(word, dict[i]) )
        {
            return 1 ;
        }
    }
    return 0 ;
}
```

注意，对符号常量 MAXWORD **的定义在两个构件中都出现了，因为他们都要声明用于存储变量。然而，只有** dict2.c **中才含有用与存储字典的数据结构的声明，因为只有远程过程调用才包含字典的数据结构。**

从实际的观点看，将应用程序分为两个文件使得分别编译客户机和服务器成为可能。编译器检查诸如符号常量之类的问题，而链接程序将检查所有的数据结构同引用它们的过

程结合到了一起。在 Linux 下：

gcc –c dict1.c

gcc –c dict2.c

这将产生两个目标文件（并不是完整的程序）。这些构建必须连接到一起以产生一个可执行的程序。但是，这并不是编译他们的直接原因：我们使用编译器来检查这两个文件是否语法正确。

在考虑让编译器检查代码这种方法时，我们要记住，大多数分布式程序要比我们的简单例子复杂的多。一次编译可能会发现一大堆问题，他们会转移程序员的注意力。在插入其他代码之前抓住这些问题可以使我们以后调试省力的多。

3．创建一个 rpcgen 规格说明

我们现在已经为我们的字典程序选择了一种结构，下面我们将准备一个 rpc 规格说明。从本质上说，一个 rpcgen 规格说明文件包含了对一个远程程序的声明以及它所使用的数据结构。

该规格说明文件包含常量，类型定义，以及对客户机和服务器程序的声明。更加准确的说，这个规格文件包括：

- 声明在客户机或服务器（远程程序）中所使用的常量。
- 声明所使用的数据类型（特别是对远程过程的参数）。
- 声明远程程序，每个程序中所包含的过程，以及它们的参数的类型。

我们知道 RPC 使用一些数字来表示远程程序以及在这些程序中的远程过程。在规格说明文件中的程序声明定义了诸如程序的 RPC 号，版本号，以及分配给程序中的过程的编号等细节。

所有这些声明必须用 RPC 编程语言来给出，而不是用 C。尽管他们的区别是微小的，但它们可能会成为障碍。例如，RPC 用关键字 string 代表以 null 结尾的字符串，而 C 却用 char* 来表示。因此，即使是一个有经验的程序员，要产生一个正确的规范说明，可能也需要多次反复。

文件 rdict.x 说明了一个 rpcgen 规格说明。它包含了对字典程序的 RPC 版的声明。

```
/* rdict.x RPC 规格说明 */
const      MAXWORD = 50 ;
const      DICTSIZ = 100 ;

/*********************************************************
     RDICTPROG 远程过程,包括 insert, delete 和 lookup    *
 *********************************************************/
/*  远程过程的名字 */
program RDICTPROG
{
     /*  远程过程的版本 */
     version RDICTVERS
     {
     /*  第一个过程 INITW */
     int INITW ( void ) = 1 ;
     /*  第二个过程 INSERTW */
     int INSERTW ( string ) = 2 ;
     /*  第三个过程 DELETEW */
```

```
    int DELETEW ( string ) = 3 ;
    /*  第四个过程 LOOKUPW */
    int LOOKUPW ( string ) = 4 ;
    /*  定义了版本号  */
    }= 1 ;
/*  远程过程的数字 （ 必须是唯一的 ）*/
}= 0x30090949 ;
```

一个 rpcgen **规格说明文件并没有囊括最初的程序中所能找到的所有声明.它仅仅定义了那些在客户机和服务器之间要共享的常量和数据类型，或者是那些需要指明的参数。**

这个规格说明的例子是由定义常量 MAXWORD **和** DICTSIZE **开始。在最初的应用中，这两个常量都是用** C **的预处理语句** #define **定义的符号常量。** RPC **不使用** C **的符号常量声明。取而代之的是，它要求符号常量用关键字** const **声明，赋值时使用等号（**=**）。**

按照约定，规格说明文件使用大写的名字来定义过程和程序。正如我们下面将看到的，这些名字会成为可以在 C **程序中使用的符号常量。在这里，并不是绝对的有要求要大写，但是这样做可以有助于避免冲突。**

4．**运行** rpcgen

在完成了规格说明后，我们可以运行 rpcgen **来检查语法错误，如果没有发生错误，系统将会产生三个文件。**

```
    [zixia@bbs tmp]$ ls
    rdict.x
    [zixia@bbs tmp]$rpcgen rdict.x
    [zixia@bbs tmp]$ ls
    rdict.h    rdict.x    rdict_clnt.c    rdict_svc.c
```

系统生成的文件为 rdict.h rdict_clnt.c rdict_svc.c **。**

下面我们先来看看 rdict.h **文件：**

```
/*  Please do not edit this file.
    It was generated using rpcgen.    */
#ifndef _RDICT_H_RPCGEN
#define _RDICT_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern “C” {
#endif

#define MAXWORD 50
#define DICTSIZ 100
#define RDICTPROG 0x30090949
#define RDICTVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define INITW 1
extern    int * initw_1(void *, CLIENT *);
```

```
extern    int * initw_1_svc(void *, struct svc_req *);
#define INSERTW 2
extern    int * insertw_1(char **, CLIENT *);
extern    int * insertw_1_svc(char **, struct svc_req *);
#define DELETEW 3
extern    int * deletew_1(char **, CLIENT *);
extern    int * deletew_1_svc(char **, struct svc_req *);
#define LOOKUPW 4
extern    int * lookupw_1(char **, CLIENT *);
extern    int * lookupw_1_svc(char **, struct svc_req *);
extern int rdictprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define INITW 1
extern    int * initw_1();
extern    int * initw_1_svc();
#define INSERTW 2
extern    int * insertw_1();
extern    int * insertw_1_svc();
#define DELETEW 3
extern    int * deletew_1();
extern    int * deletew_1_svc();
#define LOOKUPW 4
extern    int * lookupw_1();
extern    int * lookupw_1_svc();
extern int rdictprog_1_freeresult ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif
#endif /* !_RDICT_H_RPCGEN */
```

它包含了在规格说明文件中所声明的所有常量和数据类型 C **的合法声明。另外，**rpcgen **还增加了对远程过程的定义。在这个例子代码中，**rpcgen **定义大写的** INSERTW **为** 2**，因为规格说明中声明过程** INSERTW **为远程程序的第二个过程。**

这里需要解释一下的是 rdict.h **中的外部过程声明。这个被声明的过程构成了客户机端的** stub **的接口部分。过程名取自业已声明过的过程名，只是将其全部换为小写，并附加上一个下画线和程序的版本号。例如，我们的规格说明文件的离子声明：远程程序包含有过程** DELETEW**。于是** rdict.h **含有一个对过程** deletew_1 **的外部声明。为了理解为什么** rpcgen **要声明这些接口例程，我们可以回顾** stub **接口的部分目的：它允许** rpcgen **选择自己的调用约定，而又可以让最初的调用过程保持不变。**

作为接口 stub **命名的一个离子，让我们考虑过程** insertw**。最初的过程将成为服务器的一部分，而且将保持不变。这样，服务器将具有一个名为** insertw **的过程，它和最初的应用程序具有相同的参数。为避免命名冲突，服务器必须为接口** stub **过程使用不同的名字。**Rpcgen

让服务器端的通讯 stub 调用一个叫做 insertw_1 的接口 stub 过程。该调用使用 rpcgen 所选择的参数，而且它允许程序员设计 insertw_1 以便能用正确的参数调用 insertw。

对于我们这个字典程序的例子，rpcgen 产生了文件 rdict_clnt.c，这是一个源程序，它将成为这个分布式客户机端的通信 stub。

下面我们来看看 rpcgen 生成的客户机代码：

```
/*  Please do not edit this file.
    It was generated using rpcgen. */
#include <memory.h> /* for memset */
#include "rdict.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *initw_1(void *argp, CLIENT *clnt)
{
    static int clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, INITW,
        (xdrproc_t) xdr_void, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *
insertw_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, INSERTW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *deletew_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
```

```
    if (clnt_call (clnt, DELETEW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *lookupw_1(char **argp, CLIENT *clnt)
{
    static int clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, LOOKUPW,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

该文件为远程程序中的每一个过程准备好了一个通信 stub 过程。像在服务器中一样，选择过程名字的时候有意避免了冲突。

现在我们来看最后一个文件：rdict_svc.c**。它含有服务器所需要的代码。该文件包含服务器在开始时要执行的主程序。它包括获得一个协议端口号，向端口影射器注册 RPC 程序，接着便等待接收 RPC 调用。它将每个调用分派给合适的服务器端的 stub 接口。当被调用的过程响应时，服务器创建一个 RPC 应答并将它发回客户机。**

```
/*  Please do not edit this file.
    It was generated using rpcgen. */
#include “rdict.h”
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void rdictprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
```

```
union {
char *insertw_1_arg;
char *deletew_1_arg;
char *lookupw_1_arg;
} argument;
char *result;
xdrproc_t _xdr_argument, _xdr_result;
char *(*local)(char *, struct svc_req *);

switch (rqstp->rq_proc) {
case NULLPROC:
(void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
return;
case INITW:
_xdr_argument = (xdrproc_t) xdr_void;
_xdr_result = (xdrproc_t) xdr_int;
local = (char *()(char *, struct svc_req *)) initw_1_svc;
break;
case INSERTW:
_xdr_argument = (xdrproc_t) xdr_wrapstring;
_xdr_result = (xdrproc_t) xdr_int;
local = (char *()(char *, struct svc_req *)) insertw_1_svc;
break;
case DELETEW:
_xdr_argument = (xdrproc_t) xdr_wrapstring;
_xdr_result = (xdrproc_t) xdr_int;
local = (char *()(char *, struct svc_req *)) deletew_1_svc;
break;
case LOOKUPW:
_xdr_argument = (xdrproc_t) xdr_wrapstring;
_xdr_result = (xdrproc_t) xdr_int;
local = (char *()(char *, struct svc_req *)) lookupw_1_svc;
break;
default:
svcerr_noproc (transp);
return;
}
memset ((char *)&argument, 0, sizeof (argument));
   if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
svcerr_decode (transp);
return;
}
result = (*local)((char *)&argument, rqstp);
   if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
```

```
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "unable to free arguments");
    exit (1);
    }
    return;
}

int main (int argc, char **argv)
{
    register SVCXPRT *transp;
    pmap_unset (RDICTPROG, RDICTVERS);
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, RDICTPROG, RDICTVERS, rdictprog_1, IPPROTO_UDP)) {
    fprintf (stderr, "unable to register (RDICTPROG, RDICTVERS, udp).");
    exit(1);
    }
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, RDICTPROG, RDICTVERS, rdictprog_1, IPPROTO_TCP)) {
    fprintf (stderr, "unable to register (RDICTPROG, RDICTVERS, tcp).");
    exit(1);
    }

    svc_run ();
    fprintf (stderr, "svc_run returned");
    exit (1);
    /* NOTREACHED */
}
```

该文件一旦被生成，就可以被编译成为目标代码的形式。在 Linux **系统下，编译它们的命令是：**

```
gcc –c rdict_clnt.c
gcc –c rdict_svc.c
```

每个命令用一个 C **源文件产生一个相应的目标文件。目标文件名用后缀“** .o **”替代“** .c **”后缀。例如，**rdict_clnt.c **被编译后的目标文件为** rdict_clnt.o**。**

5**．编写** stub **接口过程**

rpcgen 产生的文件并没有构成一个完整的程序。它还要求我们必须编写客户机端和服务器端的接口例程。在远程程序中的每一个远程过程都必须存放在一个接口过程中。

在客户机端，原来的主应用程序控制着处理的进行。它调用接口过程，而它所使用的过程名和参数类型与最初的那个程序（非分布式版）调用原过程所于使用的完全一样，在分布式版本中，原来这些过程业已成为远程的了。每个接口过程都必须要将它的参数转换为 rpcgen 所使用的形式，还必须接着调用响应的客户机端的通信过程。例如，因为最初的程序含有一个叫做 insertw 的过程，它使用一个指向字符串的指针作为参数，客户机端接口必须也含有这样一个过程。该接口过程调用 insertw_1，它是 rpcgen 生成的客户机端的通信 stub。

常规的过程参数与通信 stub 所使用的参数间的主要不同在于，rpcgen 生成的所有过程的参数都使用间接方式。例如，如果最初的过程有一个整形参数，那么，通信 stub 中该过程的响应的参数必须是一个指向整数的指针。在字典程序中，多数过程要求一个字符串参数，它在 c 中用一个字符指针（char*）来声明。在响应的通信 stub 中，都要求它们的参数是一个指向字符指针的指针（char **）。

文件 rdict_cif.c 展示了接口例程如何将参数转换为 rpcgen 产生的代码所期望的形式。对程序中的每个远程过程，文件中都包含了一个客户机端口的接口过程。

```
/* rdict_cif.c    包含 initw, insertw, deletew, lookupw 过程  */
#include <stdio.h>
#include <rpc/rpc.h>

#include “rdict.h”
/* Client : 客户读    腌    衿鞣谋淞?*/
extern    CLIENT    *handle  ;

/*  为返回数据作临时存储  */
static    int        *ret  ;
/*************************************************************
      initw    客户端的过程接口,调用 initw_1                    *
 *************************************************************/

int initw()
{
      ret = initw_1 ( 0, handle ) ;
      return ret==0?0:*ret ;
}

/*************************************************************
      insertw    客户端的过程接口,调用 insertw_1                *
 *************************************************************/
int insertw ( const char* word )
{
      char **arg ;
      arg = &word ;
      ret = insertw_1 ( arg, handle ) ;
```

```
    return ret==0?0:*ret ;
}

/*************************************************************
    deletew    客户端的过程接口,调用 insertw_1                *
 *************************************************************/
int deletew ( const char* word )
{
    char **arg ;
    arg = (char**)&word ;

    ret = deletew_1 ( arg, handle ) ;
    return ret==0?0:*ret ;
}

/*************************************************************
    lookupw    客户端过程接口,调用 insertw_1                  *
 *************************************************************/
int lookupw ( const char* word )
{
    char **arg ;
    arg = &word ;
    ret = insertw_1 ( arg, handle ) ;
    return ret==0?0:*ret ;
}
```

下面我们在来看看服务器端的接口例程：

在服务器端，接口例程接受来自 rpcgen **所产生的通信** stub **的调用，并将控制权传递给实现这个指定调用的过程。如同客户机端那样，服务器端接口例程必须吧参数由** rpcgen **所选择的类型变为被调用过程所使用的类型。在大多数情况下，这种差异在于一种间接方式：** rpcgen **传递的是指向一个对象的指针，而不是对象本身。为了转换一个参数，接口过程只需要使用** C **的见解运算符（*）。文件** rdict_sif.c **展示了这个概念。它包含了字典程序的服务器端的接口例程。**

```
/* rdict_sif.c 包含 init_1, insert_1, delete_1, lookup_1 过程 */
#include <rpc/rpc.h>
#include “rdict.h”

static      int    retcode ;

/*************************************************************
    insertw_1_svc    服务器端接口: insertw 过程                *
 *************************************************************/
int *insertw_1_svc ( char **w, struct svc_req *rqstp )
{
    retcode = insertw ( *w ) ;
```

```
    return &retcode ;
}

/*************************************************************
    initw_1    服务器端接口: initw 过程                    *
 *************************************************************/
int *initw_1_svc ( void *p, struct svc_req *rqstp )
{
    retcode = initw () ;
    return &retcode ;
}

/*************************************************************
    deletew_1    服务器端接口: deletew 过程                  *
 *************************************************************/
int *deletew_1_svc ( char **w, struct svc_req *rqstp )
{
    retcode = deletew ( *w ) ;
    return &retcode ;
}

/*************************************************************
    lookupw_1    服务器端接口: lookupw 过程                  *
 *************************************************************/
int *lookupw_1_svc ( char **w, struct svc_req *rqstp )
{
    retcode = lookupw ( *w ) ;
    return &retcode ;
}
```

6．编译并链接客户机程序

在客户机借口例程编写完成并放到一个源文件后，它们就可以被编译了。例如：文件 rdict_cif.c 中包含了字典例子中所有的接口例程。在 Linux 系统下，这样编译：

```
gcc –c rdict_cif.c
```

编译器产生输出文件 rdict_cif.o。为了完成客户机，程序员需要在最初的主程序中加入一点新的细节，因为新版本使用了 RPC，程序需要针对 RPC 声明的 C 的 include 文件。它还需要包含文件 rdict.h，这是因为该文件包含了客户机和服务器都要使用的常量定义。

客户机程序还需要声明并初始化一个句柄（handle），RPC 通信例程用该句柄和服务器通信。多数客户机使用已定义的类型 CLIENT 声明这个句柄，并且通过调用 RPC 例程库 clnt_create 来初始化这个句柄。文件 rdict.c 展示了这些必要的代码：

```
/* rdict.c    包含 main 和 nextin 过程  */
#include <rpc/rpc.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>

#include "rdict.h"
/* 每一个输入的命令的最大长度 */
#define MAXWORD 50

/* 远程主机的名字 */
#define RMACHINE "localhost"

/* 远程过程的句柄 */
CLIENT  *handle ;
int nextin ( char* cmd, char* word) ;
int initw() ;
int insertw ( const char* word ) ;
int deletew ( const char* word ) ;
int lookupw ( const char* word ) ;

/* 主函数,进行各种操作 */
int
main ( int argc, char* argv[] )
{
    /* 从命令行取得命令的变量 */
    char word [ MAXWORD + 1 ] ;
    char cmd ;
    /* 输入单词的长度 */
    int   wrdlen ;

    handle = clnt_create ( RMACHINE, RDICTPROG, RDICTVERS, "tcp" ) ;
    if ( 0 == handle )
    {
        printf ( "Could not contact remote program. \n" ) ;
        exit ( 1 ) ;
    }
    while ( 1 )
    {
        wrdlen = nextin ( &cmd, word ) ;
        if ( wrdlen < 0 )
        {
            exit ( 0 ) ;
        }
        switch ( cmd )
        {
            case 'I' :
            /* 初始化 */
```

```
                initw () ;
                printf ( "Dictionary initalized to empty. \n" ) ;
                break ;
            case 'i' :
            /* 插入操作 */
                insertw ( word ) ;
                printf ( "%s inserted. \n", word ) ;
                break ;
            case 'd' :
            /* 删除操作 */
                if ( deletew(word) )
                {
                    printf ( "%s deleted. \n", word ) ;
                }else
                {
                    printf ( "%s not found. \n", word ) ;
                }
                break ;
            case 'l' :
            /* 查找 */
                if ( lookupw(word) )
                {
                    printf ( "%s was found. \n", word ) ;
                }else
                {
                    printf ( "%s was not found. \n", word ) ;
                }
                break ;
            case 'q' :
            /* 退出 */
                printf ( "program quits \n" ) ;
                exit ( 0 ) ;
            default :
            /* 无法识别的命令 */
                printf ( "command %c invalid. \n", cmd ) ;
                break ;
        } /* switch 结束 */
    }
    clnt_destroy ( handle ) ;
}

/*********************************************************************
    nextin 函数: 读取输入的命令和命令后面跟的单词参数          *
 *********************************************************************/
```

```
int nextin ( char* cmd, char* word )
{
    int    i, ch ;

    ch = getc ( stdin ) ;
    while ( isspace(ch) )
    {
        ch = getc ( stdin ) ;
    }
    if ( ch == EOF )
    {
        return -1 ;
    }
    *cmd = (char) ch ;
    ch = getc ( stdin ) ;
    while ( isspace(ch) )
    {
        ch = getc ( stdin ) ;
    }
    if ( EOF == ch )
    {
        return -1 ;
    }
    if ( '\n' == ch )
    {
        return 0 ;
    }
    i = 0 ;
    while ( !isspace(ch) )
    {
        if ( MAXWORD < ++i )
        {
            printf ( "error: word too long\n" ) ;
            exit ( 1 ) ;
        }
        *word++ = ch ;
        ch = getc ( stdin ) ;
    }
    *word++ = 0 ;
    return i ;
}
```

比较 rdict.c **和** dict1.c**，可以从中看出，我们只加入了很少的代码。这个例子代码使用符号常量** RMACHINE **来指定远程机器的名字。为了使测试更加容易，**RMACHINE **业已定义为** localhost**，这意味着客户机和服务器将在同一台机器上运行。当然，对分布式程序的**

测试一旦完成，程序员将改变定义，使它成为服务器的永久性位置。

Clnt_create 企图向某个指定的远程机器建立连接。如果连接的企图失败，clnt_create 返回值 NULL，这使应用程序向用户报告差错。如果 clnt_create 报告了一个差错，我们的程序将退出。在实际中，客户机可以重试，或者维护一个机器列表，试着对表中的各个机器进行连接。

就像其他的 C 源文件，rdict.c 可以这样编译：

```
gcc –c rdict.c
```

在 rdict.c 的目标程序被编译出来以后，构成客户机的所有文件可以被链接成为一个可执行的程序。我们使用下面的命令：

```
gcc –o rdict rdict.o rdict_clnt.o rdict_cif.o
```

或

```
gcc –o rdict rdict.c rdict_clnt.c rdict_cif.c
```

7．编译和链接服务器程序

rpcgen 所生成的输出包含了一个服务器所需要的大多数代码。程序员提供了来年各个附加的文件：服务器接口例程（rdict_sif.c）和远程过程本身。对我们的字典例子来说，远程过程的最终版本出现在文件 rdict_srp.c 中。这些过程的代码来自最初的应用。

```
/* rdict_srp.c 包含 initw, insertw, deletew, lookupw 过程 */
#include <rpc/rpc.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include “rdict.h”
/* 字典的数据存取数组 */
char dict[DICTSIZ][MAXWORD+1] ;
/* 字典中字母数 */
int    nwords = 0 ;

int    nextin ( char* cmd, char* word) ;
int    initw() ;
int    insertw ( const char* word ) ;
int    deletew ( const char* word ) ;
int    lookupw ( const char* word ) ;

/*************************************************************
      initw    初始化字典，清空所有数据                          *
 *************************************************************/
int    initw()
{
      nwords = 0 ;
      return 1 ;
}
```

```
/*************************************************************
      insertw    将一个单词插入字典                              *
 *************************************************************/
int    insertw ( const char* word )
{
      strcpy ( dict[nwords], word ) ;
      nwords ++ ;
      return nwords ;
}

/*************************************************************
      deletew    在字典中删除一个单词                            *
 *************************************************************/
int    deletew ( const char* word )
{
      int    i ;

      for ( i=0 ; i< nwords ; i++ )
      {
            if ( 0==strcmp(word, dict[i]) )
            {
                  nwords -- ;
                  strcpy ( dict[i], dict[nwords] ) ;
                  return 1 ;
            }
      }
      return 0 ;
}

/*************************************************************
      lookupw    在字典查找一个单词                              *
 *************************************************************/
int    lookupw ( const char* word )
{
      int    i ;

      for ( i=0 ; nwords > i ;i++ )
      {
            if ( 0 == strcmp(word, dict[i]) )
            {
                  return 1 ;
            }
      }
      return 0 ;
```

```
}
```

我们这样来编译它：

```
gcc –c rdict_srp.c
```

然后，我们可以将所有的目标程序链接成一个可执行文件：

```
gcc –o rdictd rdict_svc.o rdict_sif.o rdict_srp.o
```

或

```
gcc –o rdictd rdict_svc.c rdict_sif.c rdict_srp.c
```

运行该命令时，它将可执行代码写入文件 rdictd 中。

8．启动服务器和执行客户机

我们现在开始对程序进行测试。正如我们上面说的，我们在同一台机器上面运行服务器和客户机。

在运行我们的 rdict 客户机之前，我们必须要先运行 rdictd 服务器。否则你将看到下面的错误信息：

```
Could not contact remote program.
```

我们要启动 rdictd 服务器程序：

```
./rdictd &
```

然后我们在启动 rdict 客户机就可以交互的进行客户机的操作。

12.9　小结

用 rpcgen 构造一个分布式程序由八个步骤组成。程序员由一个解决该问题的常规应用程序开始，决定如何将程序划分为本地构建和远程执行构建，将应用划分成两个物理的部分，创建一个描述远程过程的规格说明文件，运行 rpcgen 产生所需要的文件。接着程序员编写客户机端和服务器端的接口例程，并将它们与 rpcgen 所产生的代码相结合。最后，程序员编译并链接客户机端和服务器端的文件，以便产生可执行的客户机和服务器程序。

尽管 rpcgen 免除了 RPC 所要求的大部分编写代码的工作，但编写一个分布式程序还是要认真思考的。在考虑如何将程序划分成本地构建和远程构建时，程序员必须检查被每个程序片段要访问的数据，以便使数据的移动最小。程序远还必须要考虑到每个远程过程所引入的延时，以及各个程序片段将如何访问 I/O 设施。

本章所给的字典应用的例子展示了将一个简单的应用转变为一个分布式的程序要花费很多的精力。更复杂的应用程序要求更加复杂的规格说明和接口过程。

第十三章　远程磁带的访问

13.1　简介

远程介质访问主要是指远程磁盘和远程磁带的访问，这里主要以远程磁带的访问为例来讲述远程介质的访问（简称远程带驱的访问）。

大多数的计算机用户都知道做备份的重要性。一旦存储着大量信息的系统由于发生了故障而造成数据丢失，其损失将是十分巨大的，或者是几天，或者是几个月，甚至是几年的工作成果，转眼之间就全部化为乌有。由于系统错误而丢失珍贵数据的风险永远不可接受，因此，养成做备分的习惯对于计算机用户来说是十分重要的，但做备分的时候经常会发现外存不够用，如在一个多用户系统中，做一次备分将用掉几百兆、几 G 的空间，这就不是仅仅使用软盘或者硬盘所能满足的，而必须使用另外一种外存—磁带。虽然磁带的存取速度比软、硬盘慢，在访问方式上只能顺序存取，不如软、硬盘方便，但它有一个最大的好处，即存储容量大、成本低。随着技术的进步，现在已经出现了光盘，其容量更大，速度更快，相信在不久的将来可以取代磁带作为常用备份介质。

一般而言，使用几个小的备份比用一个大的备份要好一些。小的备份查找速度快而且易于使用，用户也不会忽视它们；更重要的是，若一个大型的备份（通常在一个磁带或者磁盘中容纳不下，而由访问裸设备的 cpio 命令像 DOS 中的 backup 命令那样顺序的“倒”出来后放在若干磁盘或者磁带中）中的某一环节由于某种原因损坏了，那么剩余的部分也就不能读取了。这样，使用几个小的备份就明显的改善了备份过程的可靠性，并且在发生意外的时候可以减少损失。因此，我们建议用户在做备份的时候尽可能的多做一些小的备份。

一般的，在一个局部网络如 Ethernet 中，许多工作站都没有配备磁带驱动器，甚至在整个网络中也没有配置带驱，这就使得我们只能通过 LAN 将数据在远程系统的带驱上备份到磁带上。本节将要讨论的 4.3 BSD rmt 协议就是用于这个目的，该协议由 4.3 BSD 中的 rdump 和 rrestore 命令使用。

要使用磁带设备来备份系统，就应该首先了解磁带设备是如何存取的，应该采用什么系统调用，按什么数据结构存取等等。目前用户使用磁盘较为频繁，相对来说使用磁带就少的多了，这也是客观上造成了用户对于磁带设备理解甚少的缘故。因此，熟悉磁带设备的存取方法是设计程序要做的第一步。

Rmt 协议是远程带驱访问所遵循的协议，它定义了远程带驱访问所需信息量的格式。该协议实际上描述了 Linux 系统上的远程带驱的访问过程。在 Linux 系统中提供了一个称为 rmt 调度进程的远程磁带服务器，它通常存放在文件 /etc/rmt 中，由 rshd 或者 rexecd 启动，它遵循 rmt 协议，我们自己编写的客户程序要与 rmt 调度进程通信，因此也应该遵循这个协议。

要访问远程带驱，用户只需要将必要的参数传递给远程系统上的 rmt 调度进程，具体的访问过程由远程 rmt 进程执行，本地进程只需要关心数据通信就可以了。

在了解远程带驱的访问过程后，我们就可以着手建立程序框架，编写具体的应用程序。客户程序由于只关心数据的发送和接收，因此比较简单，只要清楚了 rmt 协议，编写应用程序应该没有问题。

我们后面将主要介绍 rmt 服务器程序。

13.2 Linux 磁带驱动器的处理

在 Linux 系统中的术语“介质”代表磁盘和磁带。在磁带中可建立文件系统，并且可以像磁盘那样安装，它们唯一的区别在于磁带驱动器的设备文件名。

某些磁带部件是不可安装的，只能用于裸设备，生产厂商在用户指南中已经说明了磁带驱动器的有关信息，用户必须根据其中定义的存取方法来存取。通常盒式磁带驱动器是可安装的，而大型 9 道卷轴磁带是不可安装的。

Linux 系统给出了用户进程和磁带驱动器之间的一个简单接口函数，即 read()/write() 系统调用。使用 read()从磁带驱动器中读一个整块，该调用返回读到的实际字节数；使用 write 系统调用写一个指定大小的块到磁带驱动器。用户应该保证有足够大小的读缓冲区来处理磁带上的最大块，否则会出错。在关闭磁带驱动器的时候，如果最后的操作是一个 write，则向磁带写两个文件结束标记。

4.3 BSD 中的磁带驱动器是通过名字来访问的。如将/dev/rmt8 作为参数传递给 open 系统调用，指定一个磁带驱动器。在名字 /dev/nrmt8 中，前缀“n”指明磁带在关闭的时候不能倒带。

Linux 提供的这种磁带读取方法使得用户可以很方便的对磁带设备进行读写，其具体的实现由 Linux 系统去做，用户不需要了解磁带设备的各种控制。这也充分体现了 Linux 的优点，用户只需要在 Linux 系统提供的平台上应用就可以了，不必要陷入繁琐的硬件控制。

4.3 BSD 另外还为磁带设备提供了 ioctl 命令，它允许对磁带进行一些控制属性的操作：写文件结束标记，指定记录或者文件数的空间，使磁带驱动器脱线等等。通常情况下 System V 也为盒式磁带提供了一个 cpio 版本，它是专门为磁带操作而开发的，不能用于磁盘。除此之外用户还可以使用 tar 命令(tape archive，磁带归档)。Tar 命令最初就是用于支持大型磁带的，但现在也可以用于软盘和小型磁带。

13.3 rmt 协议

4.3 BSD 提供了一个称为 rmt 调度进程的远程磁带服务器，程序放在 /etc/rmt 中，由远程 shell 服务器 rshd 启动，也可由 rexecd 服务器启动，这两个服务器之间的区别就在于确认方式的不同。

客户与服务器通过 TCP 连接通信的时候，必须遵守一个应用协议，该协议主要由 ASCII 串组成，以换行符结束。

客户发出请求后，服务器根据情况给予相应的应答。

应答有两种：

Aretval \n（A 返回值\n）和 Eerrornum\nerrorstring\n（E 错误号\n 错误信息\n）

第一种应答是 ASCII 字符“A”后跟一个 ASCII 数码，再紧跟一个换行符，表示该请求已经成功完成。第二种应答是在出现错误的时候由服务器产生的，它由 ASCII 字符“E”后跟一 ASCII 出错号（Linux 错误号）及换行符，再加上一个错误信息串及一个换行符组成。

客户的所有请求也都以一个 ASCII 字符开始，其格式有如下几种：

1．Opathname\nmode\n (O 路径名\n 模式\n)

打开指定路径名为 pathname 的设备，模式 mode 是一个十进数值，由它说明 open 系统调用的方式，通常 mode 值为 0 表示读，为 1 表示写，为 2 表示读和写。打开成功的时候，应答为：

A0\n

2．Cpathname\n（C 路径名\n）

关闭当前已经打开的设备。路径名 pathname 参数虽然要求提供，但在服务器中被忽略，并不使用。关闭成功的时候，应答为：

A0\n

3．Loffset\nwhence\n（L 偏移量\n 偏移起始\n）

如果设备支持随机访问，那么在当前已经打开的设备上执行一次 lseek 操作的结果将使读写指针指向要读写的位置。Lseek 操作成功的时候，返回的 retval 值是一个由 lseek 返回的长整型值。

4．Wcount\ndata（W 计数值\n 数据）

表明将要对当前打开的设备写入 count 个字节，要写入的数据与客户在此命令和换行符后面立即发送。在执行 write 系统调用成功以后，retval 为该系统调用的返回值，它应与指定的 count 值相等。

5．Rcount\n（R 计数值\n）

表明将从当前打开的设备中读取不多于 count 字节的数据。如果 read 系统调用成功，应答为：

Acount\ndata

6．Ioperation\ncount\n（I 操作\n 计数值\n）

表明要在磁带驱动器上执行一次 ioctl 操作。操作 operation 所允许的值可以在头文件<sys/mtio.h>中找到。如表 13-1 所示。Operation 同时还可以有其它一些值，但它们与硬件无关。该操作的返回值 retval 在 4.3 BSD 中没有定义。

表 13-1 /etc/rmt 的 ioctl 操作

操 作	描 述
0	写 count 个 EOF 标志
1	向后覆盖 count 个 EOF 标志
2	向前覆盖 count 个 EOF 标志
3	向后覆盖 count 个记录
4	向前覆盖 count 个记录
5	卷带（忽略 count）
6	卷带（忽略 count）并脱离驱动器

7．s\n（S\n）

返回当前打开设备的状态。用户应该尽量避免使用此要求，这是因为该请求返回的值是含有硬件相关信息的二进制结构。

在通常的操作中使用到的命令仅仅有 open，close，read 和 write，除了 I 和 S 以外，该协议并没有假定远程设备就是磁带设备，因此该协议也可以用于其它的远程设备访问。

13.4 rmt 服务器设计分析

前面已经介绍了 rmt 协议，它定义了各种请求以及应答报文格式，因此 rmt 服务器可以根据各种请求分别予以处理。

从总体上看，服务器程序可以看成一个无限循环，在循环过程中检测是否有请求到来。如有，则根据不同的请求类型（由请求的第一个 ASCII 字符如 O，L，C 等判别）给予相应的处理，这可以用 C 语言的 Switch 分支语句实现。下面将介绍各种请求的处理过程。

1．open 请求

该请求的格式是：

Opathname\nmode\n（O 路径名\n 模式\n）

由于是请求打开一个磁带设备，因此，如果原来已经打开了一个设备，这个时候就必须首先关闭这个设备。磁带描述符 Tapefd 大于或者等于 0 的时候表示已经打开了一个设备，用 close 系统调用关闭之：

```
close (Tapefd);
```

这个时候可‘以用 open 系统调用打开请求的磁带设备：

```
Tapefd =open(pathname,atoi(mode));
```

若打开设备失败（Tapefd<0）,则应答一个 Linux 错误号 errno（由 Linux 给出）以及相应的错误信息：

```
ResponErr(errno);
```

若成功(Tapefd)>=0)，则应答 A0\n：

```
ResponVal(oL);
```

这里，ResponErr(ErrNo)函数可以简单实现如下：

```
char ErrMsg[100];
sprintf (ErrMsg，“E%%d\n”，Errno,ErrString[errno]);
write (sockfd2,ErrMsg,strlen(ErrMsg) );
```

ResponVal （longval）函数也可以简单的实现如下：

```
char AckMsg[10];
sprintf(AckMsg,“ A%ld\n” longval);
write (sockfd2,AckMsg,strlen(AckMsg));
```

2．close 请求

该请求的格式为：

Cpathname\n（C 路径名\n）

它请求关闭磁带设备，因此只需要将当前已经打开的磁带设备关闭就可以了，对于传送过来的 pathname 参数，虽然在请求报文中予以要求，但是实际上并没有使用。

关闭磁带设备使用 close 系统调用：

```
close(Tapefd)
```

若关闭调用失败（系统调用返回值<0），则应答一个 Linux 错误信号 errno 以及错误信息：

```
ResponErr(errno);
```

若成功(返回值>=0)，则应答 A0\n：

```
ResponVal(oL)
```

ResponErr 函数和 Respon Val 函数定义同上面一样。最后将 Tapefd 置成-1，表示没有磁带设备打开，此时对磁带设备的操作将导致错误：

```
Tapefd=-1;
```

3．lseek 请求

该请求的格式是：

Loffer\nwhence\n（L 偏移量\n 偏移起始\n）

如果设备支持随机访问，则该请求用于定位该设备读写指针,读写设备就从该指针指向的位置开始。在 Linux 系统中把各种外围设备都当成一个文件来处理，磁带设备也不例外，Tapefd 就是其文件描述符，该请求可以用 lseek 系统调用来完成：

```
lretval=lseek (Tapefd ,atoi(offset) ,atoi (whence));
```

根据其返回值 lretval (长整形)来判别该请求是否成功完成。若 lretval <0，表明失败，则应答一个 Linux 错误号 errno 以及错误信息：

ResponErr(errno)；

若返回值 lretval >=0，表明成功，则应答返回值 lretval ：

ResponVal (lretval);

4．Write 请求

该请求的格式是：

Wcount\ndata（W 计数值\n 数据）

Write 请求从 L 请求所定位的设备位置开始向当前打开的设备写入 count 个字节的数据 data，data 在换行符之后立即发送。使用 Write 系统调用写设备：

```
N=atoi(count);
Record =checkbuf(record,n);
For (I=0;I+=cc)
   Cc= read(sockfd1,&record[I],n-I);
If (( retval =write (Tapefd,record,n))<0)
ResponErr(errno);
Else
Respon Val ((long)retval);
```

首先，将字节数存入变量 n 中，然后分配一段缓冲区给 record，用于存放要接受的数据 data，然后从套接字中逐字节读入数据。在读数据结束后，调用 write 系统调用向磁带设备写，根据实际写入的字节数，给予不同的应答。若返回值<0，表明写入错误，应答一个 Linux 错误号以及错误：

```
ResponErr(errno);
```

否则表明写入了若干字节，应答写入的字节数：

```
Respon Val (retval);
```

在上面的程序段中有两点要注意。一个是读数据的时候是反复读，直到读到 n 字节，在 for 循环语句中，cc 为每次读到的字节数，循环变量每次加 cc，若没有读到数据，循环变量实际上没有变，继续读，这只是一种程序设计技巧。第二点就是分配缓冲区空间，必须保证有足够的空间存储数据，若原有缓冲区空间够大，就不必分配新的缓冲区，因为一个服务器一次只处理一个请求，所以只需要一个缓冲区；若原有缓冲空间不够，则必须分配一个新的缓冲区，同时将原有的缓冲区释放掉。

5．Read 请求

该请求的格式是：

Rcount \n（R 计数值\n）

服务器接到该请求后，从当前打开的设备读数据， 其字节数不超过 count 个。由于还要将读到的数据应答给请求方，因此需要一个缓冲区。在读完数据后，将数据发往请求方：

```
n=atoi (count);
record =check(record,n);
if (retval =read(Tapefd,record,n)<0)
ResponErr(errno);
Else
{
Respon Val ((long) retval );
}
```

与 write 请求类似，read 请求先将计数 count 保留在变量 n 中，然后分配读缓冲区给 record，用系统调用 read 从磁带设备中读 n 个字节到缓冲区 record 中。如果读失败，则应

答一个 Linux 错误信号 ：

ResponErr(errno);

否则先应答实际读到的字节数：

Respon Val((long)retval);

然后将读到的数据发往请求方：

write (sockfd2,record,retval);

由请求方接收这些数据。

6．Ioctl 请求

该请求的格式为：

Ioperation \ncount\n

(I 操作\n 计数值\n)

仅仅使用前面提供的几种操作还不足以控制真个磁带设备，可能还需要使用 Linux 系统提供的系统调用 ioctl 开进一步控制磁带设备。 Ioctl 允许的几种操作值在前面已经介绍过了，这里就不再重复。服务器接到该请求后，执行 ioctl，然后根据情况给予不同应答：

```
struct mtop mtop;
mtop.mt_op=atoi(operation);
mtop.mt_count=atoi(count);
if (ioctl(Tapefd,MTIOCTOP,(char * )&mtop<0)
ResponErr(errno);
else
Respon Val(( long )mtop.mp_cout);
```

mtop 是 Linux 为 ioctl 调用提供的一个数据结构，请求报文提供的参数就是通过 mtop 传递 ioctl 调用的。若该调用失败，则应答一个 Linux 错误号，否则应答 ioctl 系统调用的结果。

7．Statu 请求

该请求的格式是：

s\n（S\n）

服务器接收到该请求后，通过系统调用 ioctl 来获得当前打开设备的状态。若调用失败，则应答一个 Linux 错误号，否则应答状态信息的长度以及状态信息：

```
struct mtget        mtget;‘
if (ioctl(Tapefd,MTIOCGET,(char * )&mtget)<0)
ResponErr( errno);
Else
{
Respon Val(( long)sizeof(mtget);
Write (sockfd2,(char * )&mtget,sizeof(mtget));
)
```

前面对于 W 请求的解释说明是在 write 系统调用返回后才给客户应答的，我们可以修改 rmt 服务器，使得在调用 write 系统调用向磁带驱动器写数据块之前给客户以应答。这样服务器就必须记住 write 系统调用的结果以便发回个给客户的下一应答，这是因为写出错的时候还必须给客户一个信号。这也意味着客户在写完最后一个数据块后必须执行某一命令，如 C 命令来关闭设备，以使得它可以得到最后的一个 write 系统调用的结果。

这些程序都是类似的，并不难理解，相信读者可以编写出自己的网络通信程序。

第十四章　WWW 上 HTTP 协议

14.1　引言

Internet 是在 ARPAnet（美国国防部高级研究工程局网络）的基础上发展起来的，ARPAnet 建立于 60 年代末期，在刚开始的十几年中它主 要服务于科研教育部门，到 90 年代初期，随着 WWW 的发展，Internet 逐渐走向民用，由于 WWW 通过良好的界面大大简化了 Internet 操作的难度，使得用户的数量急剧增加，许多政府机构、商业公司意识到 Internet 具有巨大的潜力，于是纷纷大量加入 Internet，这样 Internet 上的站点数量大大增长，网络上的信息五花八门、十分丰富。

如今 Internet 已经深入到人们生活的各个部分，通过 WWW 浏览、电子邮件等方式，人们可以及时的获得自己所需的信息，Internet 大大方便了信息的传播，给人们带来一个全新的通讯方式，可以说 Internet 是继电报、电话发明以来人类通讯方式的又一次革命。

WWW 的飞速发展和广泛应用得宜于其提供的大量服务，这些服务为 人们的信息交流带来了极大的便利。WWW 的含义是（World Wide Web，环球信息网）,是一个基于超文本方式的信息查询方式。WWW 是由欧洲粒子物理研究中心（CERN）研制的。通过超文本方式将 Internet 上不同地址的信息有机的组织在一起，WWW 提供了一个友好的界面，大大方便了人们的信息浏览，而且 WWW 方式仍然可以提供传统的 Internet 服务，如 FTP、Gopher、News、E-Mail 等。

14.2　HTTP 客户请求

WWW 访问是标准的客户端／服务器模式。下面我们分别阐述一下它们。

14.2.1　客户端

在一个使用者的观点来看，Web 上面有丰富的图象文字多媒体信息。一般来说简称为“ 页面 ”。每个页面都包含指向其他页面的“ 连接 ”,“ 连接 ” 指向的页面可以存储与世界的任意的地方。使用者可以通过点击这个连接，浏览连接所指向的页面。而新的页面又存在许多连接，使用者又可以继续点击下去。包含这种连接的页面称为“ 超文本 ”(hypertext)。

用户浏览页面所使用的程序成为‘ 浏览器 ”（Browser）。现在 Netscape 和 Internet Explorer 是最常用的两种浏览器。一个浏览器发出一个取得页面的请求，然后解释页面中存在的控制命令，将页面显示出来。一个页面包含一个“ 标题 ” 和一些信息。页面中可以连接到其他页面的字符串被称为“ 超文本链接 ”。它一般都包含下划线。当我们把光标（或是鼠标箭头）移动到超文本链接上并且按“ 回车 ”（或是点击鼠标左键）的时候，浏览器就会去取代表这个超文本链接的页面。

14.2.2　服务器端

每一个 Web 服务器上面都运行着一个侦听 TCP 的 80 端口的进程来等待来自客户端的 HTTP 请求（一般来说，都是 Web Browser 发出这个请求）。当一个连接发生的时候，客户

端发送一个请求，当服务器收到这个请求后，服务器将会将客户端所请求的数据返回客户端。在这之后，这个连接就结束了。定义了请求和回应规则的协议称为"HTTP"。我们在下面将会学习有关它的一些知识。图 14-1 大致表示出了一个 HTTP 请求的过程。

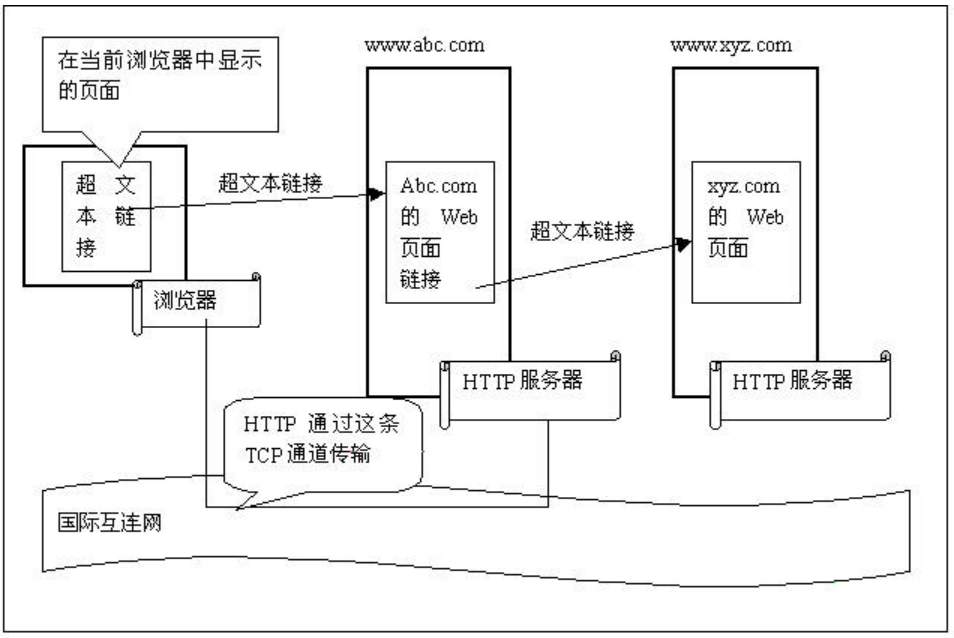

图 14-1　HTTP 请求的过程

14.2.3　Web 请求简介

在这个例子中，使用者点击有超文本链接的文字或图象，然后浏览器就将超文本链接所代表的 URL (Uniform Resource Locator)显示出来。

URL 由下面这三部分构成：(以HTTP://www.aka.citf.net/index.html 为例)

- 最前面的是协议的名字，比如"HTTP://"。
- 后面跟着的是存放页面的服务器名字（也可以是 IP），我们这里的服务器名字为"www.aka.citf.net "。
- 后面跟着的为页面在服务器上面存放的绝对路径。上例中的页面路径为/index.html。
- 最后还可能跟着提供服务的服务器端口。（可以省略，因为每一个协议都有自己的标准端口。因此只有一个服务器没有使用标准的端口提供服务，我们才需要自己指定这个端口。

一般来说，一个浏览器被使用者选中了一个超文本链接后，发生了如下几步操作：

1．浏览器中的某个 URL 被使用者选中。

2．浏览器使用 DNS 查询 URL 中域名所代表的 IP。

3．DNS 发回一个回应，通知浏览器所查询域名所代表的 IP。

4．浏览器通过 TCP 协议连接到 Web 服务器的 80 端口（如果使用缺省不指定特殊端口的话)。

5．浏览器向 Web 服务器发送一个 GET HTTP://aka.citf.net/index.html请求。

6．Web 服务器将 index.html 发送回来。

7．浏览器结束这个 TCP 连接。

8．浏览器将 index.html 这个页面显示出来。

9．浏览器将 index.html 中包含的图象，声音等也下载，显示，播放。

许多浏览器对于它所进行的每一部分工作都会显示在它窗口下面的状态条中。这样，当浏览速度非常慢的时候，你就可以知道究竟是哪一部分出了问题：或者是 DNS 解析不出 IP，还是连接不上 Web 服务器。

值得一提的是页面中的多媒体文件，比如图片。浏览器对于页面中的每一个图片都会建立一个新的 TCP 连接来进行获取操作，如果页面中有许许多多的图象，那么浏览器就会建立许多 TCP 连接来取得这些图象。

HTTP 是一种 ASCII 的协议，就像 SMTP 一样。这样，我们就可以非常简单的使用一个 Telnet 来和 Web 服务器进行交互：连接到服务器的 80 端口。在下面我们看一个简单的例子（下面各行以 C: 开头的为我们客户端的输入，而以 S: 开头的是 Web 服务器的回应）。

```
[zixia@bbs zixia]$ telnet localhost 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^）'.
C: GET / HTTP/1.0
S: HTTP/1.1 200 OK
S: Date: Sat, 24 Jul 1999 14:24:27 GMT
S: Server: Apache/1.3.6 (UNIX)  (Red Hat/Linux) mod_perl/1.19
S: Last-Modified: Wed, 07 Apr 1999 21:17:54 GMT
S: ETag: "d5802-799-370bcb82"
S: Accept-Ranges: bytes
S: Content-Length: 1945
S: Connection: close
S: Content-Type: text/html
S:
S: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
S: <HTML>
S:   <HEAD>
S:   <TITLE>Test Page for Red Hat Linux's Apache Installation</TITLE>
S:   </HEAD>
S: <!-- Background white, links blue (unvisited), navy (visited), red (active) -->
S:   <BODY
S:     BGCOLOR="#FFFFFF"
S:     TEXT="#000000"
S:     LINK="#0000FF"
S:     VLINK="#000080"
S:     ALINK="#FF0000"
```

S:　>
S:　<H1 ALIGN="CENTER">It Worked!</H1>
S:　<P>
S:　If you can see this, it means that the installation of the
S:　<A
S:　HREF="http://www.apache.org/"
S:　>Apache</A>
S:　software on this Red Hat Linux system was
S: successful. You may now add content to
S:　this directory and replace this page.
S:　</P>
S:　<HR WIDTH="50%" SIZE="8">
S:　<BLOCKQUOTE>
S:　If you are seeing this instead of the content you expected, please
S:　<STRONG>contact the administrator of the site involved.</STRONG>　If
S:　you send mail about this to the authors of the Apache software or Red
S:　Hat Software, who almost
S:　certainly have nothing to do with this site, your message will be
S:　<STRONG><BIG>ignored</BIG></STRONG>.
S:　</BLOCKQUOTE>
S:　<HR WIDTH="50%" SIZE="8">
S:　<P>
S:　The Apache
S:　<A
S:　HREF="manual/index.html"
S:　>documentation</A>
S:　has been included with this distribution. <p>For documentation
S:　and information on Red Hat
S:　Linux, please visit the web site of Red
S:　Hat Software. The manual for Red Hat Linux is available <a
S:　href="http://www.redhat.com/manual">here.
S:　</P>
S:　<P>
S:　You are free to use the image below on an Apache-powered web
S:　server.　Thanks for using Apache!
S:　</P>
S:　<P ALIGN="CENTER">
S:　<IMG SRC="/icons/apache_pb.gif"
S:　ALT="[Powered
S: by Apache]">

S: </P>
S: You are free to use the image below on a Red Hat Linux-powered web
S: server. Thanks for using Red Hat Linux!
S:
S: <img src="poweredby.gif" alt="[Powered
S: by Red Hat Linux]">
S:
S: </BODY>
S: </HTML>
Connection closed by foreign host.
[zixia@bbs zixia]$ _

我们这样使用的时候，记得在 GET 和 URL 中间一定要有空格，在 URL 和 HTTP/1.0 中间也要留有空格。而且最后一定要按两下回车，也就是要多出一个空行来。（这些都是 HTTP 协议所规定的格式）

注：上面的信息可能由于各种原因（比如你的机器的 WWW ROOT 目录不同）而不同，但是取得页面的方法是一样的。

这个例子中，是我们自己连接的 Web 服务器（而不是 Web 浏览器），我们首先连接到了 Web 服务器的 80 端口，然后发送一条请求某个页面的命令（ GET … ），并在后面附上了协议的版本号（ HTTP/1.0 ）。然后服务器的应答发送给我们一条消息：200 OK。200 代表一切正常，没有发现错误。这条应答下面的信息是符合 RFC 822 的说明 MIME 的信息。下面跟着一行空白，代表 HTTP 信息头结束，空白下面的就是你所请求的页面数据了。Web 服务器如果想发送一个图片，那么 MIME 的信息头可能是下面这样：

Content-Type: Image/GIF

通过这个方法，使用 MIME 类型可以允许通过 HTTP 协议传送不同的文件类型。

HTTP 浏览器可以同时支持其他协议，比如 FTP，GOPHER 等协议。但是在浏览器上支持他们可能会使浏览器代码体积变的蠢笨。HTTP Proxy 可以解决这个问题。现在来说，大多数使用者只是在通过 HTTP Proxy 传递 HTTP 中转指令（比如教育网内，出国要收费，所以大多数时候人们都是通过 HTTP Proxy 来浏览国外的 Web 站点）。但是 Proxy 有另外一个用处就是它可以帮助 Web 浏览器来传输 FTP，Gopher 站点，而浏览器并不用再去支持 FTP，Gopher 协议（图 14-2）。

这样，我们就可以使用浏览器来浏览一般情况下浏览器所不支持的协议站点。浏览器所能够浏览的范围通过 Proxy 的支持变的更加宽广了。当然，这样的 Proxy 需要支持各种各样的协议。

HTTP 的 Proxy 服务器还支持缓存。它的意思是如果你曾经通过 Proxy 传输过一个页面（或是一个多媒体文件），则 Proxy 会将数据存放在它的缓存区中。当你以后在去取相同的页面（或是多媒体文件）的时候，Proxy 服务器就会在它的缓存区中发现这个文件，可以直接将缓存中的数据发送给你。这样就节省了在去其他服务器取页面（或多媒体文件）的时间和带宽。

代理服务器还可以做访问限制。这点特性可以方便的控制一个局域网内部的访问范围。比如一个学校内部通过一个 Proxy 上网，那么学校就可以设置 Proxy 不允许访问一些

站点（比如HTTP://www.playboy.com ）。这样当一个浏览器向 Proxy 发送一个取得信息的请求，Proxy 服务器可以自己做出判断，从而决定是否允许这个请求。

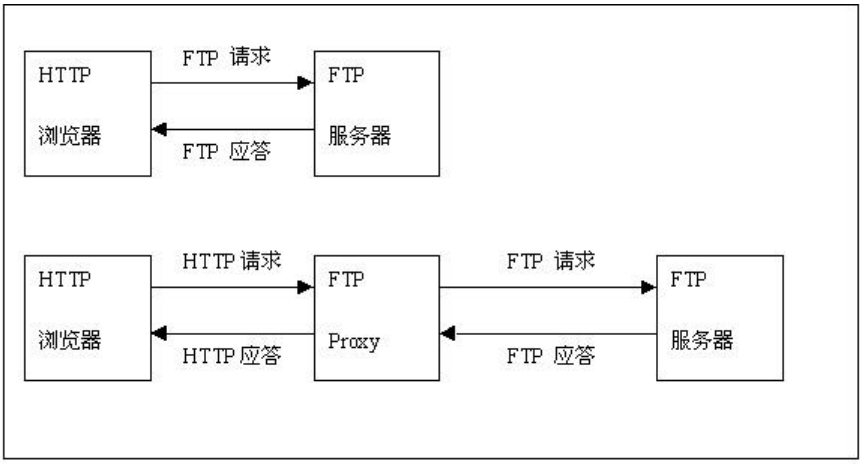

图 14-2　HTTP Proxy 服务器

14.2.4　HTTP– HyperText Transfer Protocol 超文本传输协议

标准的 Web 传输协议是 HTTP 协议（HyperText Transfer Protocol）。它定义每一次交互过程都是一个字符请求，然后紧跟着由 RFC 822 MIME 的应答。虽然使用 TCP 连接进行传输数据非常常用，但是 HTTP 协议也可以使用 TCP 协议以外的连接。

HTTP 还在发展之中，现在正在使用的有好几种 HTTP 协议，而且还有其他的版本正在开发之中。下面我们将要谈论的 HTTP 协议，是那些最基本的，并且已经基本决定下来不再做变更的一些命令（需要注意的是各个版本之间可能会有一些很小的区别）。

HTTP 协议清楚的分为两部分：从浏览器向服务器发送的请求集合和服务器送往浏览器的应答集合。下面我们分别讨论他们：

所有新版本的 HTTP 协议支持两种不同的请求格式：

简单请求方式和完全请求方式。

- 简单请求方式：它发送的请求数据的字符串只包含一个 GET 行，指明要取的页面地址。没有注明 HTTP 协议的版本号。它所获得的服务器的回应数据只包含它所请求页面的数据，没有任何 HTTP 信息头，也没有 MIME 信息。想对它进行测试的话，你可以试试 telnet zixia.citf.net 80 然后输入 GET /index.html 但是并不在 GET 的末尾输入 HTTP 协议的版本号。返回的页面将不包含任何的 HTTP 信息头和 MIME 信息说明，直接就是你所期望的页面的数据。
- 完全请求方式：它在发送请求页面的 HTTP 信息的时候，在 GET 命令行的末尾要添加上 HTTP 协议的版本号。请求信息可以包含很多行的数据，最后跟着的是一个空行，代表 HTTP 请求数据头的结束。完全请求方式的第一行是包含命令的一行（这一行中的命令以 "GET" 最常见，但是它也可以是其他命令），后面跟着你想得到的页面的地址，最后是 HTTP 协议的版本号。在后面的行中是在 RFC 822 中定义过的其他信息。

虽然 HTTP 协议是由使用 WEB 而被设计出来的，但是现在它已经发展到了很多领域，所以它在定义的时候就为将来的使用多定义了一些命令。在完全请求方式中，第一行的第

一个单词被称为"方法"（Method 或 Command），它用来定义对 Web 的访问方法。当你需要获取一个页面的时候，你可以根据需要使用各种 HTTP 协议的"方法"。下面给出几个"方法"的列表，注意他们是区分大小写的。

表 14-1 HTTP 方法

方法	方法的描述
GET	发出读取一个 Web 页面的请求
HEAD	发出读取一个 Web 页面头部的信息
PUT	发出存储一个 Web 页面信息的指令
POST	为一个已经被命名的文件添加信息
DELETE	删除一个 Web 页面
LINK	连接上两个存在的资源
UNLINK	将已经连上资源的连接终止

- GET 方法

GET 方法向服务器发出获取一个页面的请求（或是将“页面”换成“实体”，这样可以代表更一般的情况），要得到经过 MIME 编码的数据。如过在 GET 命令后面加上一个“If-Modified-Since”信息头，则服务器只发送在指定的日期后有改动过的数据。使用这种机制，浏览器可以查询是否服务器上面存放的页面比自己的缓存区中存放的页面新，从而决定是否去从新下载。

- HEAD 方法

HEAD 方法只向服务器请求信息头，并不请求真正的数据。这个帮法可以帮助浏览器得到页面的一些信息（比如最后更新日期），收集服务器页面上的索引，或者只是用来测试是否存在这个页面。

- PUT 方法

PUT 方法和 GET 方法正好相反，它不是读取一个页面，而是写入一个页面。这个方法可以帮助你在远程机器上面建立页面。它的后面跟着真正的数据信息（允许进行 MIME 编码），而这些信息将被服务器写入磁盘（如果允许的话）。PUT 的 MIME 编码和 PUT 一样。

- POST 方法

POST 方法和 PUT 方法有些相仿。但是不同的是，POST 方法是将信息"添加"到以有页面后（可以作为参数），而 PUT 是"替换"或"新生成"一个页面。将一条信息 POST 到新闻组或向 WEB 的讨论区添加一篇文章都是 POST 方法的例子。

- DELETE 方法

也许你已经猜到了…… DELETE 方法删除一个页面。像 PUT 一样，它需要身份认证和服务器对这个身份的授权。使用这个命令要特别注意，没有任何迹象会告诉你 DELETE 操作成功。即使远程的 Web 服务器已经同意你进行 DELETE 操作，也不一定会成功。因为文件可能拥有 Web 服务器无法删除的模式。

- LINK 和 UNLINK 方法

它们允许你在从一个页面连接到另外一个页面。

每一个请求都得到一个包含状态信息的回应，也许还有附加的信息。（比如：页面的

全部或是它的一部分）状态信息可能是 200 代码（代表一切正常，未发生错误），或是其他的错误代码（比如 304：未更改；400：无效请求；403：禁止访问）。

HTTP 的标准描述信息头和信息体的详细定义可以参考 RFC 822 MIME 信息。

14.3　Web 编程

我们现在已经了解一个访问 WWW 的浏览器是如何于一个 Web 服务器进行交互的了。那么我们就可以借助以前我们将的套接字编程，自己来写一个简单的 WWW 浏览器程序。

浏览器程序介绍：

- 使用方法：www_client Web_Server_Host
- 显示：它只能将 HTML 原码显示出来，所以实际上，它应该叫做 WWW 超文本原码浏览器。

www_client.c 源码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/*  建立一个 HTTP TCP 连接的辅助函数  */
int
htconnect (char *domain,int port )
  {
   int white_sock;
   struct hostent * site;
   struct sockaddr_in me;

   site = gethostbyname(domain);
   if (site==NULL)
     {
          return -2;
     }

   white_sock = socket (AF_INET, SOCK_STREAM, 0 ) ;
   if (white_sock<0)
     {
     return -1;
      }
```

```
    memset(&me,0,sizeof(struct sockaddr_in));
    memcpy(&me.sin_addr,site->h_addr_list[0],site->h_length);
    me.sin_family = AF_INET;
    me.sin_port = htons(port);

    return
    (connect (white_sock,(struct sockaddr*)&me,sizeof(struct sockaddr))<0 ) ?
    1 : white_sock ;
    }

/*  发送 HTTP 信息头的辅助函数  */
int htsend(int sock,char *fmt,...)
   {
      char BUF[1024];
      va_list argptr;

      va_start(argptr,fmt);
      vsprintf(BUF,fmt,argptr);
      va_end(argptr);
      return send(sock,BUF,strlen(BUF),0) ;
   }

   void main(int argc,char **argv)
   {
      int black_sock;
      char bugs_bunny[3];
      if (argc<2)
      {
         printf ( "Usage:\n\twww_client host\n" ) ;
         return ;
       }
      black_sock = htconnect(argv[1],80);
      if (black_sock<0)
      {
            printf ( "Socket Connect Error!\n" ) ;
            return ;
      }
     htsend(black_sock,"GET / HTTP/1.0%c",10);
     htsend(black_sock,"Host: %s%c",argv[1],10);
```

```
    htsend(black_sock,"%c",10);
    while (read(black_sock,bugs_bunny,1)>0)
    {
        printf("%c",bugs_bunny[0]);
     }
     close(black_sock);
  }
```

下面我们来编译这个程序：

```
root@bbs#gcc –owww_client www_client.c
root@bbs#
```

我们来执行它：

```
root@bbs#www_client localhost
```

在我的 RedHat 6.0 **的机器上，它将会连接到自己的** 80 **端口，而缺省的** RedHat **的** Web **页面是** Apache Web **服务器的测试页，在我的机器上面显示结果大致如下：**

```
root@bbs# ./www_client localhost
HTTP/1.1 200 OK
Date:       Fri, 23 Jul 1999 20:22:56 GMT
Server: Apache/1.3.6 (UNIX)  (Red Hat/Linux) mod_perl/1.19
Last-Modified: Wed, 07 Apr 1999 21:17:54 GMT
ETag: "d5802-799-370bcb82"
Accept-Ranges: bytes
Content-Length: 1945
Connection: close
Content-Type: text/html

  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
  <HTML>
  <HEAD>
  <TITLE>Test Page for Red Hat Linux's Apache Installation</TITLE>
  </HEAD>
  <!-- Background white, links blue (unvisited), navy (visited), red (active) -->
  <BODY
BGCOLOR="#FFFFFF"
TEXT="#000000"
LINK="#0000FF"
VLINK="#000080"
ALINK="#FF0000"
 >
<H1 ALIGN="CENTER">It Worked!</H1>
<P>
```

If you can see this, it means that the installation of the
<A
HREF="http://www.apache.org/"
>Apache</A>
software on this Red Hat Linux system was successful. You may now add content to
this directory and replace this page.
</P>
<HR WIDTH="50%" SIZE="8">
<BLOCKQUOTE>
If you are seeing this instead of the content you expected, please
<STRONG>contact the administrator of the site involved.</STRONG> If
you send mail about this to the authors of the Apache software or Red
Hat Software, who almost
certainly have nothing to do with this site, your message will be
<STRONG><BIG>ignored</BIG></STRONG>.
</BLOCKQUOTE>
<HR WIDTH="50%" SIZE="8">
<P>
The Apache
<A
HREF="manual/index.html"
>documentation</A>
has been included with this distribution. <p>For documentation
and information on Red Hat
Linux, please visit the web site of Red
Hat Software. The manual for Red Hat Linux is available <a
href="http://www.redhat.com/manual">here.
</P>
<P>
You are free to use the image below on an Apache-powered web
server. Thanks for using Apache!
</P>
<P ALIGN="CENTER">
<IMG SRC="/icons/apache_pb.gif" ALT="[Powered by Apache]">
</P>
You are free to use the image below on a Red Hat Linux-powered web
server. Thanks for using Red Hat Linux!
<p align="center">

```
<a href="http://www.redhat.com/"><img src="poweredby.gif" alt="[ Powered
   by Red Hat Linux ]"></a>
   </p>
   </BODY>
</HTML>
root@bbs#
```

好了，我们现在已经可以取得一台服务器的首页了。如果你有兴趣，可以对 HTML **超文本进行分析，然后在屏幕上像** Netscape **或** Internet Expoler **那样画出页面来。**

14.4　小结

在这章中，我们简单的介绍了 Web **的发展，**Web **浏览器和服务器的行为，以及** HTTP **协议。最后我们自己写了一个可以取得服务器首页的小程序，来模拟** Web **浏览器的简单行为。**

附录 A 有关网络通信的服务和网络库函数

accept 系统调用

用法：

retcode = accept (socket, addr, addrlen) ;

说明：

服务器调用 socket 创建一个套接字，用 bind 指定一个本地 IP 地址和协议端口号，然后用 listen 使套接字处于被动状态，并设置连接请求队列的长度。accept 从队列中取走下一个连接请求（或一直在那里等待下一个连接请求的到达），为请求创建一个新套接字，并返回新的套接字描述符。 accept 只用于流套接字（如 TCP 套接字）。

参数：

- socket 类型为 int,，含义是由 socket 函数创建的一个套接字描述符。
- addr 类型为&sockaddr，含义是一个地址结构的指针。accept 在该结构中填入远程机器的 IP 号和协议端口号。
- addrlen 类型为&int 含义是一个整数指针，初始指定为 sockaddr 参数的大小，当调用返回十，指定为存储在 addr 中的字节数。

返回码：

accept 成功时返回一个非负套接字描述符。发生错误的时候返回 –1。

当发生错误时，全局变量 errno 含有如下值之一：

- EBADF 错误原因是第一个参数未指定合法的描述符。
- ENOTSOCK 错误原因是第一个参数未指定一个套接字描述符
- EOPNOTSUPP 错误原因是套接字类型不是 SOCK_STREAM。
- EFAULT 错误原因是第二个参数中的指针非法
- EWOULDBLOCK 错误原因是套接字被标记为非阻塞的，且没有正等待连接的联入请求。

bind 系统调用

用法:

retcode = bind (socket, localaddr, addrlen) ;

说明：

bind 为一个套接字指明一个本地 IP 和协议端口号。 bind 主要由服务器使用，它需要指定一个知名协议口。

参数：

- socket 类型为 int，含义是由 socket 调用创建一个套接字描述符
- localaddr 类型为&sockaddr，含义是一个地址结构，指定一个 IP 地址和协议端口

号

- addrlen 类型为 int，含义是地址结构的字节数大小。

返回码：

bind 若成功则返回 0 ，返回 –1 表示发生了错误。当错误发生时，全局变量 errno 含有下面的错误代码：

- EBADF　错误原因是第一个参数未指定合法的描述符。
- ENOTSOCK　错误原因是第一个参数未指定一个套接字描述符
- EADDRNOTAVAIL　错误原因是指明的地址不可用（比如一个 IP 地址与本地接口不匹配）
- EADDRINUSE　错误原因是指明的地址正在使用（如另外一个进程已经分配了协议端口）
- EINVAL　错误原因是套接字已经帮定到一个地址上
- EACCES　错误原因是不允许应用程序指明的地址
- EFAULT　错误原因是参数 localaddr 中的指针非法

---

close 系统调用

用法：

retcode = close (socket) ;

说明：

应用程序使用完一个套接字后调用 close 。close 文明的中止通信，并删除套接字。任何正在套接字上等待被读取的数据都将被丢弃。

实际上，Linux 实现了引用计数器制，它允许多个进程共享一个套接字。如果 n 个进程共享一个套接字，引用计数器将为 n 。close 每被一个进程调用一次，就将引用计数器减 1。一旦引用计数器减到 0 ，套接字将被释放。

参数：

- socket 类型为 int，含义是将被关闭的套接字描述符

返回值：

close 若成果就返回 0 ，如果返回 –1 则代表有错误 fasheng.dang 错误发生时，全局变量 errno 将含有以下值：

- EBADF　错误原因是参数 socket 未指定一个合法的描述符

---

connect 系统调用

用法：

retcode = connect (socket, addr, addrlen) ;

说明：

connect 允许调用者为先前创建的套接字指明远程端点的地址。如果套接字使用了 TCP ，connect 就使用三方握手建立一个连接；如果套接字使用 UDP ，connect 仅指明远程端点，但不向它传送任何数据报。

参数：

- socket 类型为 int，含义是一个套接字的描述符
- addr 类型为&sockaddr_in，含义是远程机器端点地址
- addrlen 类型为 int，含义是第二个参数的长度

返回值：

connect 若成功就返回 0 ，返回 –1 代表发生了错误。当错误发生时，全局变量 errno 含有下面的值：

- EBADF 错误原因是参数 socket 未指定一个合法的描述符
- ENOTSOCK 错误原因是参数 socket 未指定一个套接字描述符
- EAFNOSUPPORT 错误原因是远程端点指定的地址族不能与这种类型的套接字一起使用
- EADDRNOTAVAIL 错误原因是指定的地址不可用
- EISCONN 错误原因是套接字已被连接
- ETIMEDOUT 错误原因是（只用于 TCP）协议因未成功建立一个连接而超时
- ECONNREFUSED 错误原因是（只用于 TCP）连接被远程机器拒绝
- ENETUNREACH 错误原因是（只用于 TCP）网络当前不可到达
- EADDRINUSE 错误原因是指定的地址正在使用
- EINPROGRESS 错误原因是（只用于 TCP）套接字是非阻塞的，且一个连接尝试将北阻塞
- EALREADY 错误原因是（只用于 TCP）套接字是非阻塞的，且调用将等待前一个连接尝试完成

---

fork 系统调用

用法：

```
retcode = fork () ;
```

说明：

虽然 fork 并不于套接字通信直接相关，但是由于服务器使用它创建并发的进程，因此它很重要。 fork 创建一个新进程，执行与原进程相同的代码。两个进程共享在调用 fork 时已打开的所有套接字和文件描述符。两个进程有不同的进程标识符和不同的父进程标识符。

参数：

- 没有任何参数

返回值：

如果成功，fork 给子进程返回 0 ，给父进程返回子进程的进程标识符（非零）。如果发生了错误，它将返回 –1 。全局变量可能是下面的值：

- EAGAIN 错误原因是已经达到了系统限制的进程总数，或已经达到了对每个用户的进程限制。
- ENOMEM 错误原因是系统没有足够的内存用于新进程。

---

gethostbyaddr 库函数

用法：

retcode = gethostbyaddr (addr, len, type) ;

说明：

gethostbyaddr **搜索关于某个给定** IP **地址的主机信息。**

参数：

- addr **类型为**&char**，含义是指向一个数组的指针，该数组含有一个主机地址（如** IP **地址）**
- len **类型为** int**，含义是一个整数，它给出地址长度（**IP **地址长度是** 4**）**
- type **类型为** int**，含义是一个整数，它给出地址类型（**IP **地址类型为** AF_INET**）**

返回值：

gethostbyaddr **如果成功，返回一个** hostent **结构的指针。如果发生错误，返回** 0 **。**

hostent **结构声明如下：**

```
struct hostent
{                               /*  一个主机项  */
    char h_name ;               /*  正式主机名  */
    char h_aliases[] ;          /*  其它别名列表  */
    int  h_addrtype ;           /*  主机地址类型  */
    int  h_length ;             /*  主机地址长度  */
    char **h_addr_list          /*  主机地址列表  */.
} ;
```

当发生错误时，全局变量 h_errno **中含有下列值之一：**

- HOST_NOT_FOUND　**错误原因是不知道所指定的名字**
- TRY_AGAIN　**错误原因是暂时错误：本地服务器现在不能于授权机构联系**
- NO_RECOVERY　**错误原因是发生了无法恢复的错误。**
- NO_ADDRESS　**错误原因是指明的名字有效，但是它无法于某个** IP **地址对应**

---

gethostbyname **库调用**

用法：

retcode = gethostbyname (name) ;

说明：

gethostbyname **将一个主机名映射为一个** IP **地址。**

参数：

- name **类型为**&char**，含义是一个含有主机名字符串的地址**

返回值：

gethostbyname **如果成功就返回一个** hostent **结构的指针，如发生错误则返回** 0 **。**hostent **结构声明为：**

```
struct hostent
{                               /*  一个主机项  */
```

```
    char h_name ;           /* 正式主机名 */
    char h_aliases[] ;      /* 其它别名列表 */
    int  h_addrtype ;       /* 主机地址类型 */
    int  h_length ;         /* 主机地址长度 */
    char **h_addr_list      /* 主机地址列表 */.
} ;
```

当发生错误时，全局变量 h_errno **含有下列值之一：**

- HOST_NOT_FOUND **错误原因是不知道所指定的名字**
- TRY_AGAIN **错误原因是暂时错误：本地服务器**
- NO_RECOVERY **错误原因是发生了无法恢复的错误。**
- NO_ADDRESS **错误原因是指明的名字有效，但是它无法于某个** IP **地址对应**

---

gethostid **系统调用**

用法：

```
hostid = gethostid () ;
```

说明：

应用程序调用 gethostid **以获取指派给本地机器的唯一的** 32 **位的主机标识符。通常，主机标识符是机器的主** IP **地址。**

参数：

- **没有任何参数**

返回值：

gethostid **返回一个含有主机标识符的长整数。**

---

gethostname **系统调用**

用法：

```
retcode = gethostname ( name, namelen ) ;
```

说明：

gethostname **用文本字符串的形式返回本地机器的主名字。**

参数：

- name **类型为**&char**，含义是放置名字的字符数组的地址**
- namelen **类型为** int**，含义是名字数组的长度（至少应该为** 65**）**

返回值：

若 gethostname **成功则返回** 0 **，若发生错误则返回** –1 **。当发生错误十，全局变量** errno **含有以下值：**

- EFAULT **错误原因是** name **或** namelen **参数不正确**

---

getpeername **系统调用**

用法：

retcode = getpeername (socket, remaddr, addrlen) ;

说明：

应用程序使用 getpeername **获取以建立连接的套接字的远程端点的地址。通常，客户机调用** connect **时设置了远程端点的地址，所以它知道远程地址。但是，使用** accept **获得连接的服务器，可能需要查询套接字来找出远程地址。**

参数：

- socket **类型为** int**，含义是一个由** socket **函数创建的套接字描述符**
- remaddr **类型为**&sockaddr**，含义是一个含有对端地址的** sockaddr **结构的指针。**
- addrlen **类型为**&int**，含义是一个整数指针，调用前，该函数含有第二个参数的长度，调用后该整数含有远程端点地址的实际长度。**

返回值：

getpeername **如果成功则返回** 0 **，如果发生错误则返回** –1 **。当发生错误的时候，全局变量** errno **含有如下值之一：**

- EBADF　**错误原因是第一个参数未指定一个合法的描述符**
- ENOTSOCK　**错误原因是第一个参数未指定一个套接字描述符**
- ENOTCONN　**错误原因是套接字不是一个已经连接的套接字**
- ENOBUFS　**错误原因是系统没有足够的资源完成操作**
- EFAULT　**错误原因是** remaddr **参数指针无效**

---

getprotobyname **系统调用**

用法：

retcode = getprotobyname (name) ;

说明：

应用程序调用 getprotobyname **，以便根据协议名找到该协议正式的整数值。**

参数：

- name **类型为**&char**，含义是一个含有协议名字的字符串地址**

返回值：

getprotobyname **如果成功则返回** protoent **类型的指针。如果发生错误则返回** 0 **。结构** protoent **声明如下：**

```
struct protoent
{                           /*  协议描述项  */
    char p_name ;           /*  协议的正式名  */
    char **p_aliases ;      /*  协议的别名列表  */
    int  p_proto ;          /*  正式协议号  */
}
```

---

getservbyname **库调用**

用法：

retcode = getservbyname (name, proto) ;

说明：

getservbyname **根据给出的服务名，从网络服务库中获取该服务的有关信息。客户机和服务器都调用** getservbyname **将服务名映射为协议端口号。**

参数：

- name **类型为**&char**，含义是一个含有服务名的字符串指针。**
- proto **类型为**&char**，含义是一个含有所用协议名的字符串指针。**

返回值：

getservbyname **如果成功则返回一个** servent **结构的指针，如果发生错误则返回一个空指针。**servent **结构声明如下：**

```
struct servent
{                           /*  服务项  */
    char s_name ;           /*  正式服务名  */
    char **s_aliases ;      /*  其它别名列表  */
    int    s_port ;         /*  该服务使用的端口  */
    char s_proto ;          /*  服务所使用的协议  */
} ;
```

---

getsockname **系统调用**

用法：

retcode = getsockname (socket, name, namelen) ;

说明：

getsockname **获得指定套接字的本地地址。**

参数：

- socket **类型为** int**，含义是一个由** socket **创建的描述符**
- name **类型为**&sockaddr**，含义是一个含有** IP **地址和套接字协议端口号的结构的指针**
- namelen **类型为**&int**，含义是** name **结构中的字节数，返回时为结构的大小**

返回值：

getsockname **如果成功则返回** 0 **，如果发生错误则返回** –1 **。一旦发生错误，全局变量** errno **中含有如下值之一：**

- EBADF **错误原因是第一个参数没有指定一个合法的描述符**
- ENOTSOCK **错误原因是第一个参数没有指定一个套接字描述符**
- ENOBUFS **错误原因是系统中没有足够的缓冲区空间可用**
- EFAULT **错误原因是** name **或** namelen **的地址不正确**

---

getsockopt **系统调用**

用法：

retcode = getsockopt (socket, level, opt, optval, optlen) ;

说明：

getsockopt **允许一个应用获得某个套接字的参数值或该套接字所使用的协议。**

参数：

- socket **类型为** int**，含义是一个套接字描述符**
- level **类型为** int**，含义是一个整数，它标识某个协议级**
- opt **类型为** int**，含义是一个整数它标识某个选项**
- optval **类型为**&char**，含义是存放返回值的缓冲区地址**
- optlen **类型为**&int**，含义是缓冲区大小，返回时为所发现的值的长度。**

用于套接字的查究级选项包括：

- SO_DEBUG **调试信息的状态**
- SO_REUSEADDR **允许本地地址重用**
- SO_KEEPALIVE **连接保持状态**
- SO_DONTROUTE **忽略出报文的选路**
- SO_LINGER **如果存在数据，延迟关闭**
- SO_BROADCAST **允许传输广播报文**
- SO_OOBINLINE **在带内接受带外数据**
- SO_SNDBUF **输出缓冲区大小**
- SO_RCVBUF **输入缓冲区大小**
- SO_TYPE **套接字的类型**
- SO_ERROR **获取并清楚套接字的上一次出错**

返回值：

getsockopt **如果成功则返回** 0 **，如果发生错误则返回** –1 **，并且** errno **全局变量中含有如下值之一：**

- EBADF　**错误原因是第一个参数未指定一个合法的描述符**
- ENOTSOCK　**错误原因是第一个参数未指定一个套接字描述符**
- ENOPROTOOPT　**错误原因是** opt **不正确**
- EFAULT　**错误原因是** optval **或** optlen **的地址不正确**

---

gettimeofday **系统调用**

用法：

retcode = gettimeofday (tm, tmzone) ;

说明：

gettimeofday **从系统中提取当前时间和日期，以及有关本地时区的信息。**

参数：

- tm **类型为**&struct timeval**，含义是一个** timeval **结构的地址**
- tmzone **类型为**&struct timezone**，含义是一个** timezone **结构的地址**

两个结构声明如下：

```
struct timeval
{                           /* 存储时间的结构 */
    long tv_sec ;           /* 自 1/1/70 以来的秒数 */
    long tv_usec ;          /* 超过 tv_sec 的毫秒数 */
} ;

struct timezone
{                               /* timezone 信息结构 */
    int   tz_minuteswest ;      /* 格林尼治以西的分钟数 */
    int   tz_dsttime ;          /* 所用的校正的类型 */
} ;
```

返回值：

gettimeofday **如果成功则返回** 0 **，如果发生错误则返回** –1 **。一旦发生错误，全局变量** errno **将含有如下的值：**

- EFAULT

tm **或** tmzone **参数含有不正确的地址**

---

listen **系统调用**

用法：

```
retcode = listen ( socket, queuelen ) ;
```

说明：

服务器使用 listen **是套接字处于被动状态（准备接受联入请求）。在服务器处理某个请求时，协议软件应将后续收到的请求排队，**listen **也设置排队的连接请求数目。**listen **只用于** TCP **套接字。**

参数：

- socket **类型为** int**，含义是一个由** socket **调用创建的套接字描述符**
- queuelen **类型为** int**，含义是入连接请求的队列大小**

返回值：

listen **若成功则返回** 0 **，若发生错误则返回** –1 **。一旦出错，全局变量** errno **含有如下值之一：**

- EBADF **错误原因是第一个参数未指定一个合法的描述符**
- ENOTSOCK **错误原因是第一个参数未指定一个套接字描述符**
- EOPNOTSUPP **错误原因是套接字类型不支持** listen

---

read **系统调用**

用法：

```
retcode = read ( socket, buff, buflen ) ;
```

说明：

客户机或服务器使用 read 从套接字获取输入。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- buff 类型为&char，含义是一个存放输入字符的数组的指针
- buflen 类型为 int，含义是一个整数，它指明 buff 数组中的字节数

返回值：

read 如果检测到套接字上遇到文件结束就返回 0 ，若它获得了输入就返回当前读取的字节数，如果发生了错误就返回 –1 。一旦出错，全局变量 errno 中含有如下值之一：

- EBADF　错误原因是第一个参数未指定合法的描述符
- EFAULT　错误原因是地址 buff 不合法
- EIO　错误原因是在读数据时 I/O 发生错误
- EINTR　错误原因是某个信号中断了操作
- EWOULDBLOCK 指定的是非阻塞的 I/O ，但是套接字没有数据。

---

recv 系统调用

用法：

retcode = recv (socket, buffer, length, flags) ;

说明：

recv 从套接字读取下一个入报文。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- buffer 类型为&char，含义是存放报文的缓冲区的地址
- length 类型为 int，含义是缓冲区的长度
- flags 类型为 int，含义是控制位，它指明是否接受带外数据和是否预览报文

返回值：

recv 如果成功则返回报文中的字节数，如果发生错误则返回 –1 。一旦出错，全局变量 errno 中含有如下值之一：

- EBADF　错误原因是第一个参数没有指定合法的描述符
- ENOTSOCK　错误原因是第一个参数不是一个套接字描述符
- EWOULDBLOCK　错误原因是套接字没有数据，但是它已经被指定为非阻塞 I/O
- EINTR　错误原因是在读操作时被信号所中断
- EFAULT　错误原因是参数 buffer 不正确

---

recvfrom 系统调用

用法：

retcode = recvfrom (socket, buffer, length, flags, from , fromlen) ;

说明：

recvfrom 从套接字获取下一个报文，并记录发送者的地址（允许调用者发送应答）。

参数：

- socket **类型为** int，**含义是一个由** socket **函数创建的套接字描述符**
- buffer **类型为**&char，**含义是存放报文的缓冲区地址**
- length **类型为** int，**含义是缓冲区的长度**
- flags **类型为** int，**含义是控制位，指明是否接受带外数据和是否预览报文**
- from **类型为**&sockaddr，**含义是存放发送方地址结构的地址**
- fromlen **类型为**&int，**含义是缓冲区的长度，返回时为发送者地址的大小**

返回值：

recvfrom **如果成功便返回报文中的字节数，如果发生错误则返回** -1 **。一旦出错，全局变量** errno **将含有下面值之一：**

- EBADF **错误原因是第一个参数没有指定合法的描述符**
- ENOTSOCK **错误原因是第一个参数没有指定一个套接字描述符**
- EWOULDBLOCK **错误原因是套接字没有数据，但是已经被设定为非阻塞** I/O **。**
- EINTR **错误原因是在读操作进行时，被信号所中断**
- EFAULT **错误原因是参数** buffer **不正确。**

---

recvmsg **系统调用**

用法：

retcode = recvmsg (socket, msg, flags) ;

说明：

recvmsg **返回套接字上到达的下一个报文。它将报文放入一个结构，该结构包括首部和数据。**

参数：

- socket **类型为** int，**含义是一个由** socket **函数创建的套接字描述符**
- msg **类型为**&struct msghdr，**含义是一个报文结构的指针**
- flags **类型为** int，**含义是控制位，它指明是否接受带外数据和是否预览报文**

报文用 msghdr **传递结构，其格式如下：**

```
struct msghdr
{
    caddr_t   msg_name ;          /* 可选地址 */
    int       msg_namelen ;       /* 地址大小 */
    struct iovec   msg_iov ;      /* 散列/紧凑数组 */
    int   msg_iovlen              /* msg_iov 中的元素字节数 */
    caddr_t   msg_accrights ;     /* 发送/接受权限 */
    int       msg_accrghtslen ;   /* 特权字段的长度 */
} ;
```

返回码：

recvmsg **如果成功便返回报文中的字节数。如果发生错误则返回** –1 **。出错后，全局变量** errno **中含有如下值之一：**

- EBADF　错误原因是第一个参数没有指定合法的描述符
- ENOTSOCK　错误原因是第一个参数没有指定一个套接字描述符
- EWOULDBLOCK　错误原因是套接字没有数据，但已被指定为非阻塞 I/O 。
- EINTR　错误原因是在读操作可以传递数据前被信号中断
- EFAULT　错误原因是参数 msg 不正确。

---

select 系统调用

用法：

retcode = select (numfds, refds, wrfds, exfds, time) ;

说明：

select 提供异步 I/O ，它允许单进程等待一个指定文件描述符集合中的任意一个描述符最先就绪。调用者也可以指定最大等待时间。

参数：

- numfds 类型为 int，含义是集合中文件描述符的数目
- refds 类型为&fd_set，含义是用作输入的文件描述符的集合
- wrfds 类型为&fd_set，含义是用作输出的文件描述符的集合
- exfds 类型为&fd_set，含义是用作异常的文件描述符的集合
- time 类型为&struct timeval，含义是最大等待时间

涉及描述符的参数由整数组成，而整数的第 i 比特与描述符 i 相对应。宏 FD_CLR 和 FD_SET 清楚或设置各个比特位。

返回值：

select 如果成功则返回就绪的文件描述符数，若时间限制已到则返回 0 ，如果发生错误则返回 –1 。一旦出错全局变量　errno 中含有下面的值之一：

- EBADF　错误原因是某个描述符集合指定了一个非法的描述符。
- EINTR　错误原因是在等待超时或任何一个北选择的描述符准备就绪以前，被信号中断
- EINVAL　错误原因是时间限制值不正确

---

send 系统调用

用法：

retcode = send (socket, msg, msglen, flags) ;

说明：

应用程序调用 send 将一个报文传送到另一个机器。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- msg 类型为&char，含义是报文的指针
- msglen 类型为 int，含义是报文的字节长度
- flags 类型为 int，含义是控制位，指定是否接受带外数据和是否预览报文

返回值：

send 若成功就返回以发送的字节数。否则返回 –1 。一旦出错，全局变量 errno 含有如下值之一：

- EBADF 错误原因是第一个参数未指定合法的描述符
- ENOTSOCK 错误原因是第一个参数未指定一个套接字描述符
- EFAULT 错误原因是参数 msg 不正确
- EMSGSIZE 错误原因是报文对套接字而言太大了
- EWOULDBLOCK 错误原因是套接字没有数据，但是已经北指定为非阻塞 I/O
- ENOBUFS 错误原因是系统没有足够的资源完成操作

---

sendmsg 系统调用

用法：

retcode = sendmsg (socket, msg, flags) ;

说明：

sendmsg 从 msghdr 结构中提取出一个报文并发送。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- msg 类型为&struct msghdr，含义是报文结构的指针
- flags 类型为 int，含义是控制位，指定是否接受带外数据和是否预览报文

返回值：

sendmsg 若成功就返回以发送的字节数。否则返回 –1 。一旦出错，全局变量 errno 含有如下值之一：

- EBADF 错误原因是第一个参数未指定合法的描述符
- ENOTSOCK 错误原因是第一个参数未指定一个套接字描述符
- EFAULT 错误原因是参数 msg 不正确
- EMSGSIZE 错误原因是报文对套接字而言太大了
- EWOULDBLOCK 错误原因是套接字没有数据，但是已经北指定为非阻塞 I/O
- ENOBUFS 错误原因是系统没有足够的资源完成操作

---

sendto 系统调用

用法：

retcode = sendto (socket, msg, msglen, flags, to, tolen) ;

说明：

sendto 从一个结构中获取目的地址，然后发送报文。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- msg 类型为&char，含义是报文的指针
- msglen 类型为 int，含义是报文的字节长度

- flags 类型为 int，含义是控制位，指定是否接受带外数据和是否预览报文
- to 类型为&sockaddr，含义是地址结构的指针
- tolen 类型为&int，含义是地址的字节长度的指针

返回值：

sendto 若成功就返回以发送的字节数。否则返回 –1 。一旦出错，全局变量 errno 含有如下值之一：

- EBADF　错误原因是第一个参数未指定合法的描述符
- ENOTSOCK　错误原因是第一个参数未指定一个套接字描述符
- EFAULT　错误原因是参数 msg 不正确
- EMSGSIZE　错误原因是报文对套接字而言太大了
- EWOULDBLOCK　错误原因是套接字没有数据，但是已经北指定为非阻塞 I/O
- ENOBUFS　错误原因是系统没有足够的资源完成操作

---

sethostid 系统调用

用法：

(void) sethostid (hostid) ;

说明：

系统管理员在系统启动时运行一个有特权的程序，该程序调用 sethostid 为本地机器指派唯一的 32 比特主机标识符。通常，主机标识符是机器的 IP 地址。

参数：

- hostid 类型为 int，含义是被保存作为主机标识符的值

返回值：

- 运行它的程序必须有 root 权限。否则 sethostid 不会改变主机的标识符。

---

setsockopt 系统调用

用法：

retcode = setsockopt (socket, level, opt, optval, optlen) ;

说明：

getsockopt 允许一个应用改变某个套接字的参数值或该套接字所使用的协议。

参数：

- socket 类型为 int，含义是一个套接字描述符
- level 类型为 int，含义是一个整数，它标识某个协议级
- opt 类型为 int，含义是一个整数它标识某个选项
- optval 类型为&char，含义是存放返回值的缓冲区地址
- optlen 类型为&int，含义是缓冲区大小，返回时为所发现的值的长度。

用于套接字的查究级选项包括：

- SO_DEBUG 调试信息的状态
- SO_REUSEADDR 允许本地地址重用

- SO_KEEPALIVE 连接保持状态
- SO_DONTROUTE 忽略出报文的选路
- SO_LINGER 如果存在数据，延迟关闭
- SO_BROADCAST 允许传输广播报文
- SO_OOBINLINE 在带内接受带外数据
- SO_SNDBUF 输出缓冲区大小
- SO_RCVBUF 输入缓冲区大小
- SO_TYPE 套接字的类型
- SO_ERROR 获取并清楚套接字的上一次出错

返回值：

setsockopt 如果成功则返回 0 ，如果发生错误则返回 –1 ，并且 errno 全局变量中含有如下值之一：

- EBADF 错误原因是第一个参数未指定一个合法的描述符
- ENOTSOCK 错误原因是第一个参数未指定一个套接字描述符
- ENOPROTOOPT 错误原因是 opt 不正确
- EFAULT 错误原因是 optval 或 optlen 的地址不正确

---

shutdown 系统调用

用法：

retcode = shutdown (socket, direction) ;

说明：

shutdown 函数用于全双工的套接字，并且用于部分关闭连接。

参数：

- socket 类型为 int，含义是一个由 socket 函数创建的套接字描述符
- direction 类型为 int，含义是 shutdown 需要的方向： 0 表示终止进一步输入，1 表示终止进一步输出，2 表示终止输入输出。

返回值：

shutdown 调用若操作成功则返回 0 ，若发生错误则返回 –1 。一旦出错，全局变量 errno 中含有一个指出错误原因的代码：

- EBADF 错误原因是第一个参数未指明一个合法的描述符
- ENOTSOCK 错误原因是第一个参数未指明一个套接字描述符
- ENOTCONN 错误原因是指定的套接字当前未连接

---

socket 系统调用

用法：

retcode = socket (family, type, protocol) ;

说明：

socket 函数创建一个用于网络通信的套接字，并返回该套接字的整数描述符。

参数：

- family **类型为** int**，含义是协议或地址族(** **对于** TCP/IP **为** PF_INET **，也可使用** AF_INET)
- type **类型为** int**，含义是服务的类型（对于** TCP **为** SOCK_STREAM**，对于** UDP **为** SOCK_DGRAM**）**
- protocol **类型为** int**，含义是使用的协议号，或是用** 0 **指定** family **和** type **的默认协议号**

返回值：

- EPROTONOSUPPORT　**错误原因是参数中的错误：申请的服务或指定的协议无效**
- EMFILE　**错误原因是应用程序的描述符表已经满**
- ENFILE　**错误原因是内部的系统文件表已满**
- ENOBUFS　**错误原因是系统没有可用的缓冲空间**

---

write **系统调用**

用法：

retcode = write (socket, buff, buflen) ;

说明：

write **允许一个应用程序给远方的客户端发送信息。**

参数：

- socket **类型为** int**，含义是一个由** socket **函数创建的套接字描述符**
- buff **类型为**&char**，含义是一个存放输入字符的数组的指针**
- buflen **类型为** int**，含义是一个整数，它指明** buff **数组中的字节数**

返回值：

write **若它成功就返回发送的字节数，如果发生了错误就返回** –1 **。一旦出错，全局变量** errno **中含有如下值之一：**

- EBADF　**错误原因是第一个参数未指定合法的描述符**
- EPIPE　**错误原因是试图向一个未连接的流套接字上写**
- EFBIG　**错误原因是写入的数据超过了系统容量**
- EFAULT　**错误原因是地址** buff **不合法**
- EINVAL　**错误原因是套接字的指针无效**
- EIO　**错误原因是在读数据时** I/O **发生错误**
- EWOULDBLOCK **指定的是非阻塞的** I/O **，但是套接字没有数据。**

附录 B Vi 使用简介

Vi 是 Unix 世界里极为普遍的全屏幕文本编辑器，几乎可以说任何一台 Unix 机器都会提供这套软件。Linux 当然也有，它的 vi 其实是 elvis（版权问题），不过它们都差不多。熟悉 DOS 下的文本处理后，也许会感到 vi 并不好用；Unix 上也已经发展出许多更新、更好用的文本编辑器，但是并不一定每一台 Unix 机器上都会安装这些额外的软件。所以，学习 vi 的基本操作还是有好处，让你在各个不同的机器上得心应手。

B.1 Vi 基本观念

Unix 提供一系列的 ex 编辑器，包括 ex, edit 和 vi。相对於全屏幕编辑器，现在可能很难想像如何使用 ex, edit 这种行列编辑器（有人用过 DOS 3.3 版以前所附的 EDLIN 吗？）。Vi 的原意是“ Visual”，它是一个立即反应的编辑程式，也就是说可以立刻看到操作结果。

也由於 vi 是全屏幕编辑器，所以它必须控制整个终端屏幕哪里该显示些什么。而终端的种类有许多种，特性又不尽相同，所以 vi 有必要知道现在所使用的是哪一种终端机。这是藉由 TERM 这个环境变数来设定，设定环境变数方面请查看所使用 shell 的说明。（除非执行 vi 的时候回应 unknow terminal type，否则可以不用设定。）

只要简单的执行 vi 就可以进入 vi 的编辑环境。在实际操作之前先对它有个概略的了解会比较好。Vi 有两种模式，输入模式以及命令模式。输入模式即是用来输入文字数据，而命令模式则是用来下达一些编排文件、存文件、以及离开 vi 等等的操作命令。当执行 vi 后，会先进入命令模式，此时输入的任何字符都视为命令。

B.1.1 进入与离开

要进入 vi 可以直接在系统提示字符下输入 vi ＜文件名称＞，vi 可以自动帮你调入所要编辑的文件或是开启一个新文件。进入 vi 后屏幕左方会出现波浪符号，凡是列首有该符号就代表此列目前是空的。要离开 vi 可以在命令模式下输入 :q, :wq 命令则是存文件后再离开（注意冒号）。要切换到命令模式下则是用 [ESC] 键，如果不晓得现在是处于什么模式，可以多按几次 [ESC]，系统会发出哔哔声以确定进入命令模式。

B.1.2 Vi 输入模式

要如何输入数据呢？有好几个命令可以进入输入模式：

- 新增（append）

从光标所在位置后面开始新增数据，光标后的数据随新增数据向后移动。A 从光标所在列最后面的地方开始新增数据。

- 插入（insert）

I 从光标所在位置前面开始插入数据，光标后的数据随新增数据向后移动。I 从光标所在列的第一个非空白字符前面开始插入数据。

- 开始（open）

o 在光标所在列下新增一列并进入输入模式。

O 在光标所在列上方新增一列并进入输入模式。

也许文字叙述看起来有点复杂，但是只要实际操作一下马上可以了解这些操作方式。实践很重要，尤其是电脑方面的东西随时可以尝试及验证结果。极力建议实际去使用它而

不要只是猛 K文件，才有事半功倍的效用。

B.2 Vi 基本编辑

配合一般键盘上的功能键，像是方向键、[Insert] 、[Delete] 等等，现在你应该已经可以利用 vi 来处理文字数据了。当然 vi 还提供其他许许多多功能让文字的处理更形方便，有兴趣的看倌请继续。

在继续下去之前先来点 BCC 吧。电脑有许多厂牌，不同的硬体及操作系统。PC 也不是仅仅只有 IBM PC 及其兼容品而已。事实上，包括键盘，终端等等往往都有不同的规格。这代表什么？

在文本编辑软件上会遇这样的问题，某些电脑的键盘上没有特定的几个功能键！那麼不就有某些功能不能用了？这个问题在 Unix 系统上也一样，几乎各大电脑厂商都有自己的 Unix 系统，而 vi 的操作方法也会随之有点出入。我们固然可以用 PC 的键盘来说明 vi 的操作，但是还是得提一下这个问题。

B.2.1 删除与修改

何谓编辑？在这里我们认为是文字的新增修改以及删除，甚至包括文字区块的搬移、复制等等。这里先介绍 vi 的如何做删除与修改。（注意：在 vi 的原始观念里，输入跟编辑是两码子事。编辑是在命令模式下操作的，先利用命令移动光标来定位要进行编辑的地方，然后才下命令做编辑。）

x 删除光标所在字符。

dd 删除光标所在的列。

r 修改光标所在字符，r 后接著要修正的字符。

R 进入取代状态，新增数据会覆盖原先数据，直到按 [ESC] 回到命令模式下为止。

s 删除光标所在字符，并进入输入模式。

S 删除光标所在的列，并进入输入模式。

其实呢，在 PC 上根本没有这麼麻烦！输入跟编辑都可以在输入模式下完成。例如要删除字符，直接按 [Delete] 不就得了。而插入状态与取代状态可以直接用 [Insert] 切换，犯不著用什么命令模式的编辑命令。不过就如前面所提到的，这些命令几乎是每台终端都能用，而不是仅仅在 PC 上。

在命令模式下移动光标的基本命令是 h, j, k, l。想来各位现在也应该能猜到只要直接用 PC 的方向键就可以了，而且无论在命令模式或输入模式下都可以。多容易不是。

当然 PC 键盘也有不足之处。有个很好用的命令 u 可以恢复被删除的数据，而 U 命令则可以恢复光标所在列的所有改变。这与某些电脑上的 [Undo] 按键功能相同。

B.3 Vi 进阶应用

相信现在对於 vi 应该已经有相当的认识。处理文字也不会有什么麻烦才对。如果有兴趣善用 vi 的其它功能进一步简化操作过程，不妨继续看下去。

B.3.1 移动光标

由於许多编辑工作是由光标来定位，所以 vi 提供许多移动光标的方式，这个我们列几张简表来说明（这些当然是命令模式下的命令）：

命令	说明	功能键
0	移动到光标所在列的最前面	[Home]
$	移动到光标所在列的最后面	[End]

[CTRL][d]	向下半页	
[CTRL][f]	向下一页	[PageDown]
[CTRL][u]	向上半页	
[CTRL][b]	向上一页	[PageUp]

命令	说明
H	移动到视窗的第一列
M	移动到视窗的中间列
L	移动到视窗的最后列
b	移动到下个字的第一个字母
w	移动到上个字的第一个字母
e	移动到下个字的最后一个字母
^	移动到光标所在列的第一个非空白字符

命令	说明
n-	减号移动到上一列的第一个非空白字符
	前面加上数字可以指定移动到以上 n 列
n+	加号移动到下一列的第一个非空白字符
	前面加上数字可以指定移动到以下 n 列
nG	直接用数字 n 加上大写 G 移动到第 n 列

命令	说明
fx	往右移动到 x 字符上
Fx	往左移动到 x 字符上
tx	往右移动到 x 字符前
Tx	往左移动到 x 字符前
;	配合 f&t 使用，重复一次
,	配合 f&t 使用，反方向重复一次
/string	往右移动到有 string 的地方
?string	往左移动到有 string 的地方
n	配合 /&? 使用，重复一次
N	配合 /&? 使用，反方向重复一次

命令	说明	备注
n(	左括号移动到句子的最前面	句子是以
	前面加上数字可以指定往前移动 n 个句子	! . ? 三种符号来界定
n)	右括号移动到下个句子的最前面	
	前面加上数字可以指定往后移动 n 个句子	
n{	左括弧移动到段落的最前面	段落是以
	前面加上数字可以指定往前移动 n 个段落	段落间的空白列界定
n}	右括弧移动到下个段落的最前面	

	前面加上数字可以指定往后移动 n 个段落	

不要尝试背诵这些命令，否则后果自行负责。它们看起来又多又杂乱，事实上这是文字叙述本身的障碍。再强调一次，实际去使用它只要几次就可以不经大脑直接下达这些奇奇怪怪的命令，远比死记活背搞得模模糊糊强多了。

B.3.2 进阶编辑命令

这些编辑命令非常有弹性，基本上可以说是由命令与范围所构成。例如 dw 是由删除指令 d 与范围 w 所组成，代表删除一个字 d（elete） w（ord）。

命令列表如下：

d 删除（delete）

y 复制（yank）

p 放置（put）

c 修改（change）

范围可以是下列几个：

e 光标所在位置到该字的最后一个字母

w 光标所在位置到下个字的第一个字母

b 光标所在位置到上个字的第一个字母

$ 光标所在位置到该列的最后一个字母

0 光标所在位置到该列的第一个字母

) 光标所在位置到下个句子的第一个字母

(光标所在位置到该句子的第一个字母

) 光标所在位置到该段落的最后一个字母

{ 光标所在位置到该段落的第一个字母

说实在的，组合这些命令来编辑文件有一点点艺术气息。不管怎麼样，它们提供更多编辑文字的能力。值得注意的一点是删除与复制都会将指定范围的内容放到暂存区里，然后就可以用命令 p 贴到其它地方去，这是 vi 用来处理区段拷贝与搬移的办法。

某些 vi 版本，例如 Linux 所用的 elvis 可以大幅简化这一坨命令。如果稍微观察一下这些编辑命令就会发现问题其实是定范围的方式有点杂，实际上只有四个命令罢了。命令 v 非常好用，只要按下 v 键，光标所在的位置就会反白，然后就可以移动光标来设定范围，接著再直接下命令进行编辑即可。

对於整列操作，vi 另外提供了更方便的编辑命令。前面曾经提到过删除整列文字的指令 dd 就是其中一个；cc 可以修改整列文字；而 yy 则是复制整列文字；命令 D 则可以删除光标到该列结束为止所有的文字。

B.3.3 文件命令

文件命令多以“:”开头，跟编辑命令有点区别。例如前面提到结束编辑的命令就是:q。现在就简单说明一下作为本篇的结尾：

:q 结束编辑(quit)如果不想存文件而要放弃编辑过的文件则用:q! 强制离开。

:w 存文件(write)

其后可加所要存文件的文件名。

可以将文件命令合在一起，例如:wq 即存文件后离开。

zz 功能与 :wq 相同。

另外值得一提的是 vi 的部份存文件功能。可以用 :n,mw filename 将第 n 行到第 m 行

的文字存放的所指定的 filename 里去哩。时代在变，世界在变，vi 也在变，不过大致上就这么多了。

附录 C　Linux 下 C 语言使用与调试简介

C.1　C 语言编程

Linux 的发行版中包含了很多软件开发工具。它们中的很多是用于 C 和 C++应用程序开发的。本文介绍了在 Linux 下能用于 C 应用程序开发和调试的工具。本文的主旨是介绍如何在 Linux 下使用 C 编译器和其他 C 编程工具，而非 C 语言编程的教程。在本文中你将学到以下知识：

- 什么是 C
- GNU C 编译器
- 用 gdb 来调试 GCC 应用程序

你也能看到随 Linux 发行的其他有用的 C 编程工具。这些工具包括源程序美化程序（pretty print programs），附加的调试工具，函数原型自动生成工具（automatic function prototypers）。

注意：源程序美化程序（pretty print programs）自动帮你格式化源代码产生始终如一的缩进格式。

C.2　什么是 C?

C 是一种在 UNIX 操作系统的早期就被广泛使用的通用编程语言。它最早是由贝尔实验室的 Dennis Ritchie 为了 UNIX 的辅助开发而写的，开始时 UNIX 是用汇编语言和一种叫 B 的语言编写的。从那时候起，C 就成为世界上使用最广泛计算机语言。

C 能在编程领域里得到如此广泛支持的原因有以下一些：

- 它是一种非常通用的语言。几乎你所能想到的任何一种计算机上都有至少一种能用的 C 编译器。并且它的语法和函数库在不同的平台上都是统一的，这个特性对开发者来说很有吸引力。
- 用 C 写的程序执行速度很快。
- C 是所有版本的 UNIX 上的系统语言。

C 在过去的二十年中有了很大的发展。在 80 年代末期美国国家标准协会（American National Standards Institute）发布了一个被称为 ANSI C的 C 语言标准。这更加保证了将来在不同平台上的 C 的一致性。在 80 年代还出现了一种 C 的面向对象的扩展称为 C++。

Linux 上可用的 C 编译器是 GNU C编译器，它建立在自由软件基金会的编程许可证的基础上，因此可以自由发布。你能在 Linux 的发行光盘上找到它。

C.3　GNU C 编译器

随 Slackware Linux 发行的 GNU C 编译器（GCC）是一个全功能的 ANSI C兼容编译器。如果你熟悉其他操作系统或硬件平台上的一种 C 编译器，你将能很快地掌握 GCC。本节将介绍如何使用 GCC 和一些 GCC 编译器最常用的选项。

C.3.1　使用 GCC

通常后跟一些选项和文件名来使用 GCC 编译器。gcc 命令的基本用法如下：

```
gcc [options] [filenames]
```

命令行选项指定的操作将在命令行上每个给出的文件上执行。下一小节将叙述一些你会最常用到的选项。

C.3.2　GCC 选项

GCC 有超过 100 个的编译选项可用。这些选项中的许多你可能永远都不会用到，但一些主要的选项将会频繁用到。很多的 GCC 选项包括一个以上的字符，因此你必须为每个选项指定各自的连字符，并且就象大多数 Linux 命令一样你不能在一个单独的连字符后跟一组选项。例如，下面的两个命令是不同的：

```
gcc -p -g test.c
gcc -pg test.c
```

第一条命令告诉 GCC 编译 test.c 时为 prof 命令建立剖析（profile）信息并且把调试信息加入到可执行的文件里。第二条命令只告诉 GCC 为 gprof 命令建立剖析信息。

当你不用任何选项编译一个程序时，GCC 将会建立（假定编译成功）一个名为 a.out 的可执行文件。例如，下面的命令将在当前目录下产生一个叫 a.out 的文件：

```
gcc test.c
```

注意：当你使用-o 选项时，-o 后面必须跟一个文件名。

你能用-o 编译选项来为将产生的可执行文件指定一个文件名来代替 a.out。例如，将一个叫 count.c 的 C 程序编译为名叫 count 的可执行文件，你将输入下面的命令：

```
gcc -o count count.c
```

GCC 同样有指定编译器处理多少的编译选项。-c 选项告诉 GCC 仅把源代码编译为目标代码而跳过汇编和连接的步骤。这个选项使用的非常频繁因为它使得编译多个 C 程序时速度更快并且更易于管理。缺省时 GCC 建立的目标代码文件有一个.o 的扩展名。

S 编译选项告诉 GCC 在为 C 代码产生了汇编语言文件后停止编译。GCC 产生的汇编语言文件的缺省扩展名是.s。-E 选项指示编译器仅对输入文件进行预处理。当这个选项被使用时，预处理器的输出被送到标准输出而不是储存在文件里。

C.3.3　优化选项

当你用 GCC 编译 C 代码时，它会试着用最少的时间完成编译并且使编译后的代码易于调试，易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 GCC 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是-O 和 -O2 选项。

O 选项告诉 GCC 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。-O2 选项告诉 GCC 产生尽可能小和尽可能快的代码。-O2 选项将使编译的速度比使用 -O 时慢。但通常产生的代码执行速度会更快。

除了 -O 和 -O2 优化选项外，还有一些低级选项用于产生更快的代码。这些选项非常的特殊，而且最好只有当你完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用。这些选项的详细描述，请参考 GCC 的指南页，在命令行上键入 man gcc。

C.3.4　调试和剖析选项

GCC 支持数种调试和剖析选项。在这些选项里你会最常用到的是 -g 和 -pg 选项。

g 选项告诉 GCC 产生能被 GNU 调试器使用的调试信息以便调试你的程序。GCC 提供了一个很多其他 C 编译器里没有的特性，在 GCC 里你能使 -g 和 -O (产生优化代码)联用。

这一点非常有用因为你能在与最终产品尽可能相近的情况下调试你的代码。在你同时使用这两个选项时你必须清楚你所写的某些代码已经在优化时被 GCC 作了改动。关于调试 C 程序的更多信息请看下一节“用 gdb 调试 C 程序”。-pg 选项告诉 GCC 在你的程序里加入额外的代码，执行时，产生 gprof 用的剖析信息以显示你的程序的耗时情况。关于 gprof 的更多信息请参考 “gprof” 一节。

C.3.5 用 gdb 调试 GCC 程序

Linux 包含了一个叫 gdb 的 GNU 调试程序。gdb 是一个用来调试 C 和 C++ 程序的强力调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。以下是 gdb 所提供的一些功能：

- 它使你能监视你程序中变量的值。
- 它使你能设置断点以使程序在指定的代码行上停止执行。
- 它使你能一行行的执行你的代码。

在命令行上键入 gdb 并按回车键就可以运行 gdb 了，如果一切正常的话，gdb 将被启动并且你将在屏幕上看到类似的内容：

GDB is free software and you are welcome to distribute copies of it under certain conditions; type “show copying” to see the conditions. There is absolutely no warranty for GDB; type “show warranty” for details.
GDB 4.14 (i486-slakware-linux), Copyright 1995 Free Software Foundation, Inc.
(gdb)

当你启动 gdb 后，你能在命令行上指定很多的选项。你也可以以下面的方式来运行 gdb：
gdb <fname>

当你用这种方式运行 gdb，你能直接指定想要调试的程序。这将告诉 gdb 装入名为 fname 的可执行文件。你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件，或者与一个正在运行的程序相连。你可以参考 gdb 指南页或在命令行上键入 gdb –h 得到一个有关这些选项的说明的简单列表。

1．为调试编译代码（Compiling Code for Debugging）

为了使 gdb 正常工作，你必须使你的程序在编译时包含调试信息。调试信息包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。

在编译时用 –g 选项打开调试选项。

2．gdb 基本命令

gdb 支持很多的命令使你能实现不同的功能。这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令，表 C-1 列出了你在用 gdb 调试时会用到的一些命令。想了解 gdb 的详细使用请参考 gdb 的指南页。

表 C-1 gdb 的常用命令

命 令	描 述
File	装入想要调试的可执行文件。
Kill	终止正在调试的程序。
List	列出产生执行文件的源代码的一部分。
Next	执行一行源代码但不进入函数内部。
Step	执行一行源代码而且进入函数内部。
Run	执行当前被调试的程序
Quit	终止 gdb

Watch	使你能监视一个变量的值而不管它何时被改变。
Break	在代码里设置断点，这将使程序执行到这里时被挂起。
Make	使你能不退出 gdb 就可以重新产生可执行文件。
Shell	使你能不离开 gdb 就执行 UNIX shell 命令。

gdb 支持很多与 UNIX shell 程序一样的命令编辑特征。你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令。你也能用光标键上下翻动历史命令。

3．gdb 应用举例

本节用一个实例教你一步步的用 gdb 调试程序。被调试的程序相当的简单，但它展示了 gdb 的典型应用。

下面列出了将被调试的程序。这个程序被称为 greeting，它显示一个简单的问候，再用反序将它列出。

```
#include    <stdio.h>
main ()
{
      char my_string[] = “hello there”;
      my_print (my_string);
         my_print2 (my_string);
}

void my_print (char *string)
{
      printf (“The string is %s\n”, string);
}

void my_print2 (char *string)
{
      char *string2;
      int size, i;
      size = strlen (string);
      string2 = (char *) malloc (size + 1);
      for (i = 0; i < size; i++)
      {
         string2[size－  i] = string[i];
      }

      string2[size+1] = ‘\0’;
      printf (“The string printed backward is %s\n”, string2);
}
```

用下面的命令编译它：

```
gcc –ggdb – o  test  test.c
```

这个程序执行时显示如下结果：

```
The string is hello there
The string printed backward is
```

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。我们所设想的

输出应该是：

The string printed backward is ereht olleh

由于某些原因，my_print2 函数没有正常工作。让我们用 gdb 看看问题究竟出在哪儿，先键入如下命令：

gdb greeting

注意： 记得在编译 greeting 程序时把调试选项打开。

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb，你可以在 gdb 提示符下用 file 命令来载入它：

(gdb) file greeting

这个命令将载入 greeting 可执行文件就象你在 gdb 命令行里装入它一样。这时你能用 gdb 的 run 命令来运行 greeting 了。当它在 gdb 里被运行后结果大约会象这样：

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样。问题是，为什么反序打印没有工作? 为了找出症结所在，我们可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下键入 list 命令三次，列出源代码：

```
(gdb) list
(gdb) list
(gdb) list
```

技巧：在 gdb 提示符下按回车健将重复上一个命令。

第一次键入 list 命令的输出如下：

```
1   #include   <stdio.h>
2
3   main ()
4   {
5   char my_string[] = “hello there”;
6
7   my_print (my_string);
8   my_print2 (my_string);
9   }
10
```

如果按下回车，gdb 将再执行一次 list 命令，给出下列输出：

```
11  my_print (char *string)
12  {
13  printf (“The string is %s\n”, string);
14  }
15
16  my_print2 (char *string)
17  {
18  char *string2;
19  int size, i;
```

```
20
```

再按一次回车将列出 greeting 程序的剩余部分：

```
21    size = strlen (string);
22    string2 = (char *) malloc (size + 1);
23    for (i = 0; i < size; i++)
24    string2[size - i] = string[i];
25    string2[size+1] = '\0';
26    printf ("The string printed backward is %s\n", string2);
27    }
```

根据列出的源程序，你能看到要设断点的地方在第 24 行，在 gdb 命令行提示符下键入如下命令设置断点：

```
(gdb) break 24
```

gdb 将作出如下的响应：

```
Breakpoint 1 at 0x139: file greeting.c, line 24
(gdb)
```

现在再键入 run 命令，将产生如下的输出：

```
Starting program: /root/greeting
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24
24    string2[size-i]=string[i]
```

你能通过设置一个观察 string2[size - i] 变量的值的观察点来看出错误是怎样产生的，做法是键入：

```
(gdb) watch string2[size - i]
```

gdb 将作出如下回应：

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步的执行 for 循环了：

```
(gdb) next
```

经过第一次循环后，gdb 告诉我们 string2[size - i] 的值是 'h'。gdb 用如下的显示来告诉你这个信息：

```
Watchpoint 2, string2[size - i]
Old value = 0 '\000'
New value = 104 'h'
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23
23 for (i=0; i<size; i++)
```

这个值正是期望的。后来的数次循环的结果都是正确的。当 i=10 时，表达式 string2[size - i] 的值等于 'e'，size - i 的值等于 1，最后一个字符已经拷到新串里了。

如果你再把循环执行下去，你会看到已经没有值分配给 string2[0] 了，而它是新串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空(null)字符。所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出了。

现在找出了问题出在哪里，修正这个错误是很容易的。你得把代码里写入 string2 的第一个字符的的偏移量改为 size-1 而不是 size。这是因为 string2 的大小为 12，但起始偏移量是 0，串内的字符从偏移量 0 到 偏移量 10，偏移量 11 为空字符保留。

为了使代码正常工作有很多种修改办法。一种是另设一个比串的实际大小小 1 的变量。这是这种解决办法的代码：

```
#include   <stdio.h>
main ()
{
      char my_string[] = “hello there”;
      my_print (my_string);
      my_print2 (my_string);
}

my_print (char *string)
{
      printf (“The string is %s\n”, string);
}

my_print2 (char *string)
{
      char *string2;
      int size, size2, i;
      size = strlen (string);
      size2 = size -1;
      string2 = (char *) malloc (size + 1);
      for (i = 0; i < size; i++)
      string2[size2 - i] = string[i];
      string2[size] = ‘\0’;
      printf (“The string printed backward is %s\n”, string2);
}
```

C.4 另外的 C 编程工具

Slackware Linux 的发行版中还包括一些我们尚未提到的 C 开发工具。本节将介绍这些工具和它们的典型用法。

C.4.1 Xxgdb

xxgdb 是 gdb 的一个基于 X Window 系统的图形界面。xxgdb 包括了命令行版的 gdb 上的所有特性。xxgdb 使你能通过按按钮来执行常用的命令。设置了断点的地方也用图形来显示。

你能在一个 Xterm 窗口里键入下面的命令来运行它：

```
xxgdb
```

你能用 gdb 里任何有效的命令行选项来初始化 xxgdb。此外 xxgdb 也有一些特有的命令行选项，表 C-2 列出了这些选项。

表 C-2 xxgdb 的命令行选项

db_name	指定所用调试器的名字，缺省是 gdb。
db_prompt	指定调试器提示符，缺省为 gdb。
Gdbinit	指定初始化 gdb 的命令文件的文件名，缺省为 .gdbinit。

Nx	告诉 xxgdb 不执行 .gdbinit 文件。
Bigicon	使用大图标。

C.4.2　Calls

你可以在 sunsite.unc.edu FTP 站点用下面的路径：

/pub/Linux/devel/ lang/c/calls.tar.Z

来取得 calls，一些旧版本的 Linux CD-ROM 发行版里也附带有。因为它是一个有用的工具，我们在这里也介绍一下。如果你觉得有用的话，从 BBS，FTP，或另一张 CD-ROM 上弄一个拷贝。calls 调用 GCC 的预处理器来处理给出的源程序文件，然后输出这些文件的里的函数调用树图。

注意，在你的系统上安装 calls，以超级用户身份登录后执行下面的步骤：

1．解压和 untar 文件。

2．cd 进入 calls untar 后建立的子目录。

3．把名叫 calls 的文件移动到 /usr/bin 目录。

4．把名叫 calls.1 的文件移动到目录 /usr/man/man1。

5．删除 /tmp/calls 目录。

这些步骤将把 calls 程序和它的指南页安装载你的系统上。

当 calls 打印出调用跟踪结果时，它在函数后面用中括号给出了函数所在文件的文件名：

```
main [test.c]
```

如果函数并不是向 calls 给出的文件里的，calls 不知道所调用的函数来自哪里，则只显示函数的名字：

```
printf
```

calls 不对递归和静态函数输出。递归函数显示成下面的样子：

```
fact <<< recursive in factorial.c >>>
```

静态函数象这样显示：

```
total [static in calculate.c]
```

作为一个例子，假设用 calls 处理下面的程序：

```
#include <stdio.h>
main ()
{
    char my_string[] = “hello there”;
    my_print (my_string);
    my_print2(my_string);
}

my_print (char *string)
{
    printf (“The string is %s\n”, string);
}

my_print2 (char *string)
{
    char *string2;
```

```
    int size, size2, i;
    size = strlen (string);
    size2 = size -1;
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
    {
       string2[size2 - i] = string[i];
    }

    string2[size] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
```

将产生如下的输出：

```
1    main [test.c]
2    my_print [test.c]
3    printf
4    my_print2 [test.c]
5    strlen
6    malloc
7    printf
```

calls **有很多命令行选项来设置不同的输出格式，有关这些选项的更多信息请参考** calls **的指南页。方法是在命令行上键入** calls -h**。**

C.4.3 cproto

cproto **读入** C **源程序文件并自动为每个函数产生原型申明。用** cproto **可以在写程序时为你节省大量用来定义函数原型的时间。**

如果你让 cproto **处理下面的代码：**

```
#include   <stdio.h>
main ()
{
    char my_string[] = "hello there";
    my_print (my_string);
    my_print2(my_string);
}

my_print (char *string)
{
    printf ("The string is %s\n", *string);
}

my_print2 (char *string)
{
    char *string2;
    int size, size2, i;
```

```
    size = strlen (string);
    size2 = size -1;
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
    {
       string2[size2 - i] = string[i];
    }

    string2[size] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
```

你将得到下面的输出：

```
/* test.c */
int main(void);
int my_print(char *string);
int my_print2(char *string);
```

这个输出可以重定向到一个定义函数原型的包含文件里。

C.4.4　Indent

indent **实用程序是** Linux **里包含的另一个编程实用工具。这个工具简单的说就为你的代码产生美观的缩进的格式。**indent **也有很多选项来指定如何格式化你的源代码。这些选项的更多信息请看** indent **的指南页，在命令行上键入** indent -h**。**

下面的例子是 indent **的缺省输出：**

运行 indent **以前的** C **代码：**

```
#include   <stdio.h>

main () {
        char my_string[] = "hello there";
        my_print (my_string);
        my_print2(my_string); }
my_print (char *string)
{

    printf      ("The string is %s\n", *string);
}

my_print2               (char  *string)  {
    char *string2;
    int size, size2, i;
    size = strlen (string);
    size2 = size -1;

       string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
```

```
        string2[size2 - i] = string[i];
    string2[size] = '\0';

    printf ("The string printed backward is %s\n", string2);

}
```

运行 indent **后的** C **代码：**

```
#include   <stdio.h>
main ()
{
    char my_string[] = "hello there";
        my_print (my_string);
    my_print2 (my_string);
}

my_print (char *string)
{
    printf ("The string is %s\n", *string);
}

my_print2 (char *string)
{
    char *string2;
    int size, size2, i;
        size = strlen (string);
    size2 = size -1;
    string2 = (char *) malloc (size + 1);
    for (i = 0; i < size; i++)
    {
        string2[size2 - i] = string[i];
    }

    string2[size] = '\0';
    printf ("The string printed backward is %s\n", string2);
}
```

indent **并不改变代码的实质内容，而只是改变代码的外观。使它变得更可读，这永远是一件好事。**

C.4.5 Gprof

gprof **是安装在你的** Linux **系统的** /usr/bin **目录下的一个程序。它使你能剖析你的程序从而知道程序的哪一个部分在执行时最费时间。**

gprof **将告诉你程序里每个函数被调用的次数和每个函数执行时所占时间的百分比。你**

如果想提高你的程序性能的话这些信息非常有用。

为了在你的程序上使用 gprof，你必须在编译程序时加上 -pg 选项。这将使程序在每次执行时产生一个叫 gmon.out 的文件。gprof 用这个文件产生剖析信息。

在你运行了你的程序并产生了 gmon.out 文件后你能用下面的命令获得剖析信息：

```
gprof <program_name>
```

参数 program_name 是产生 gmon.out 文件的程序的名字。

技巧：gprof 产生的剖析数据很大， 如果你想检查这些数据的话最好把输出重定向到一个文件里。

C.4.6　f2c 和 p2c

f2c 和 p2c 是两个源代码转换程序。f2c 把 FORTRAN 代码转换为 C 代码，p2c 把 Pascal 代码转换为 C 代码。当你安装 GCC 时这两个程序都会被安装上去。

如果你有一些用 FORTRAN 或 Pascal 写的代码要用 C 重写的话，f2c 和 p2c 对你非常有用。这两个程序产生的 C 代码一般不用修改就直接能被 GCC 编译。

如果要转换的 FORTRAN 或 Pascal 程序比较小的话可以直接使用 f2c 或 p2c 不用加任何选项。如果要转换的程序比较庞大，包含很多文件的话你可能要用到一些命令行选项。

注意：f2c 要求被转换的程序的扩展名为.f 或.F。

在一个 FORTRAN 程序上使用 f2c，输入下面的命令：

```
f2c my_fortranprog.f
```

要把一个 Pascal 程序装换为 C 程序，输入下面的命令：

```
p2c my_pascalprogram.pas
```

这两个程序产生的 C 源代码的文件名都和原来的文件名相同，但扩展名由 .f 或 .pas 变为 .c.