第 16 章　网络设备驱动程序

本章目标

- 了解以太网的发展历程
- 掌握以太网的连接方式、拓扑结构、传输介质和通信方式
- 掌握以太网的帧结构以及 TCP/IP 网络的结构
- 了解嵌入式以太网中常用的网络协议
- 了解嵌入式扩展以太网的常用芯片和它们各自的特点，掌握嵌入式系统中设计以太网接口的方法
- 掌握嵌入式 Linux 中以太网驱动程序的结构，理解 NE2000 兼容网卡的驱动程序
- 了解 socket 网络编程原理
- 能够利用套接字编写简单的网络应用程序

16.4 网卡驱动程序实例

下面以与 NE2000 兼容的网卡为例，具体介绍基于模块的网络驱动程序的设计过程。读者可以参考 linux/drivers/net/ne.c 和 linux/drivers/net/8390.c 两个文件的源代码。

NE2000 是以太网寄存器标准，使用非常广泛，几乎在所有的操作系统和网络协议栈中都支持 NE2000 的网络设备。

16.4.1 NE2000 的内核支持

在配置内核时，需要选择支持 NE2000 驱动程序，即 Network device support→Ethernet（10 or 100Mbit）→NE2000/NE1000 support（如图 16.9 所示），内核中的 Makefile 文件包括了对 linux/drivers/net/ne.c 和 linux/drivers/net/8390.c 两个文件的编译。

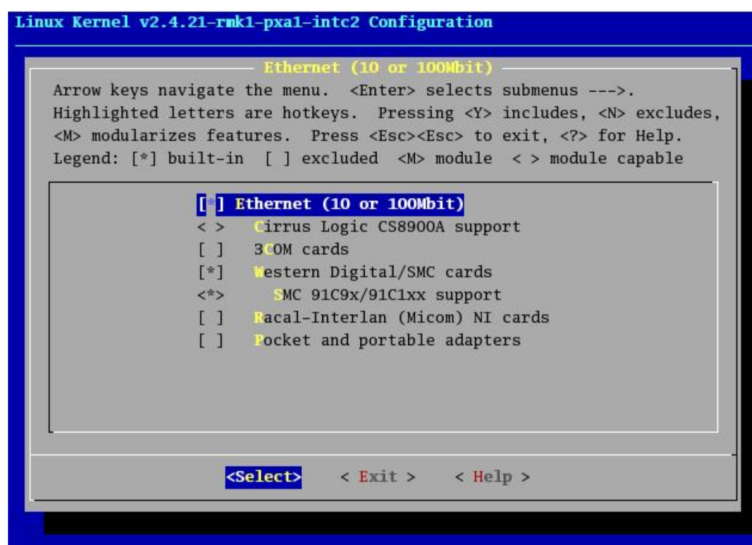

图 16.9 网络驱动程序的编译

16.4.2 网络接口的初始化

实现此功能是由 ne_probe()函数来完成的。前面已经提到过，在 init_module()函数中用它来初始化 init 函数指针。它主要对网卡进行检测，并且初始化系统中网络设备信息，用于后面的网络数据的发送和接收。具体过程及解释如下：

```
int __init ne_probe(struct net_device *dev)
{
unsigned int base_addr = dev->base_addr;
//初始化 dev-owner 成员
SET_MODULE_OWNER(dev);
//检测 dev->base_addr 是否合法，是则执行 ne-probe1()函数检测过程。不是，则需要自动检测
if (base_addr > 0x1ff)
return ne_probe1(dev, base_addr);
else if (base_addr != 0)
return -ENXIO;
```

```
    //如果有 ISAPnP 设备，则调用 ne_probe_isapnp()检测这种类型的网卡
    if (isapnp_present() && (ne_probe_isapnp(dev) == 0))
    return 0;
    …//省略
    return -ENODEV;
    }
```

其中，ne_probe_isapnp()和 ne_probe19()两个函数的区别在于检测中断号。PCI 方式只需指定 I/O 基地址就可以自动获得 IRQ，是由 BIOS 自动分配的；而 ISA 方式需要获得空闲的中断资源才能分配。

16.4.3　网络接口设备的打开和关闭

网络接口设备的打开就是激活网络接口，使它能接收来自网络的数据并且传递到网络协议栈的上面，也可以将数据发送到网络上。设备关闭就是停止操作。

在 NE2000 网络驱动程序中，网络设备打开由 dev_open()和 ne_open()完成，设备关闭由 dev_close()和 ne_close()完成。它们相应调用底层函数 ei_open()和 ei_close()来完成。其实现过程相对简单，不再赘述。

以 8390 为例，底层函数 ei_open()和 ei_close()函数的代码如下：

```
    int ei_open(struct net_device *dev)
    {
        unsigned long flags;
       struct ei_device *ei_local = (struct ei_device *) dev->priv;
       /* 仅当 ethdev_init()未被调用时使用 */
       if (ei_local == NULL)
       {
           printk(KERN_EMERG "%s: ei_open passed a non-existent device!\n",
dev->name);
           return -ENXIO;
       }
       /*时间检测*/
       if (dev->tx_timeout == NULL)
            dev->tx_timeout = ei_tx_timeout;
       if (dev->watchdog_timeo <= 0)
            dev->watchdog_timeo = TX_TIMEOUT;
            spin_lock_irqsave(&ei_local->page_lock, flags);
       NS8390_init(dev, 1);
       /* 设置标志位 */
       netif_start_queue(dev);
       spin_unlock_irqrestore(&ei_local->page_lock, flags);
       ei_local->irqlock = 0;
       return 0;
    }
    /*关闭网络设备*/
    int ei_close(struct net_device *dev)
    {
       struct ei_device *ei_local = (struct ei_device *) dev->priv;
       unsigned long flags;
       spin_lock_irqsave(&ei_local->page_lock, flags);
       NS8390_init(dev, 0);
           spin_unlock_irqrestore(&ei_local->page_lock, flags);
       netif_stop_queue(dev);
```

```
    return 0;
}
```

16.4.4 数据包发送和接收

在驱动程序层次上的发送和接收数据都是通过底层对硬件的读写来完成的。当网络上的数据到来时，将触发硬件中断，根据注册的中断向量表确定处理函数，进入中断向量处理程序，将数据送到上层协议进行处理。

对 NE2000 网卡的数据接收过程是由 ne_probe()函数中的中断处理函数 ei_interrupt 来完成的。在进入 ei_interrupt()之后再通过 ei_receive()从 8390 的接收缓冲区获得数据，并组合成 sk_buff 结构，再通过 netif_rx()函数将接收到的数据存放在系统的接收队列之中。

ei-interrupt()的函数原型如下：

```
void_ei_interrupt(int irq,void *dev_id, struct pt_regs *regs)
```

其中 irq 为中断号，dev_id 是产生中断的网络接口设备对应的结构指针，regs 表示当前的寄存器内容。

对 NE2000 网卡的数据发送是由 dev_dev_start_xmit 函数指针对应的 ei_start_xmit 函数它来完成数据包的发送。函数 ethdev_init()把 net_device 结构的 hard_start_xmit 指针初始化为 ei_start_xmit。

16.4.5 网络驱动程序的基本操作

网络设备做为一个对象，提供一些方法供系统访问。正是这些有统一接口的方法，屏蔽了硬件的具体细节，让系统对各种网络设备的访问都采用统一的形式，做到硬件无关性。

下面解释最基本的方法。

1．打开操作 open()

open()这个方法在网络设备驱动程序里是网络设备被激活的时候被调用（即设备状态由 down→up）。所以，实际上很多在 initialize 中的工作可以放到这里来做。比如资源的申请、硬件的激活。如果 dev→open 返回非 0（error），则硬件的状态还是 down。

open()方法另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 open 方法里要调用 MOD_INC_USE_COUNT 宏。

2．关闭操作 close()

close()操作完成和 open()操作相反的工作，它可以释放某些资源以减少系统的负担。close()是在设备状态由 up 转为 down 时被调用的。另外如果是作为模块装入的驱动程序，close()里应该调用 MOD_DEC_USE_COUNT，减少设备被引用的次数，以使驱动程序可以被卸载。另外，close()方法必须返回成功 0。

华清远见

华清远见嵌入式培训中心——http://www.farsight.com.cn

3．发送操作 hard_start_xmit()

所有的网络设备驱动程序都必须有这个发送方法。在系统调用驱动程序的 xmit 时，发送的数据放在一个 sk_buff 结构中。一般的驱动程序把数据传给硬件发出去。也有一些特殊的设备，比如 loopback，把数据组成一个接收数据再回送给系统，或者 dummy 设备直接丢弃数据。

如果发送成功，hard_start_xmit 方法里释放 sk_buff，返回 0（发送成功）。如果设备暂时无法处理，比如硬件忙，则返回 1。这时如果 dev→tbusy 置为非 0，则系统认为硬件忙，要等到 dev→tbusy 置 0 以后才会再次发送。tbusy 的置 0 任务一般由中断完成。

硬件在发送结束后产生中断，这时可以把 tbusy 置 0，然后用 mark_bh()调用通知系统可以再次发送。在发送不成功的情况下，也可以置 dev→tbusy 为 0，这样系统会不断尝试重发。

如果 hard_start_xmit 发送不成功，则不要释放 sk_buff。传送下来的 sk_buff 中的数据已经包含硬件需要的帧头。所以在发送方法里不需要再填充硬件帧头，数据可以直接提交给硬件发送。sk_buff 是被锁住的（locked），确保其他程序不会存取它。

4．接收操作 reception()

实际上，驱动程序并不存在一个接收方法，有数据收到应该是驱动程序来通知系统的。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 sk_buff（skb），从硬件读出数据放置到申请好的缓冲区里。

接下来，填充 sk_buff 中的一些信息。Skb→dev=dev，判断收到帧的协议类型，填入 skb→protocol（多协议的支持）。把指针 skb→mac.raw 指向硬件数据，然后丢弃硬件帧头(skb_pull)。还要设置 skb→pkt_type，标明第二层（链路层）数据类型，可以是以下类型。

- PACKET_BROADCAST：链路层广播。
- PACKET_MULTICAST：链路层组播。
- PACKET_SELF：发给自己的帧。
- PACKET_OTHERHOST：发给别人的帧（监听模式时会有这种帧）。

最后，程序调用 netif_rx()把数据传送给协议层，将 netif_rx()里的数据放入处理队列后返回。真正的处理是在中断返回以后，这样可以减少中断时间，调用 netif_rx()以后，驱动程序就不能再存取数据缓冲区 skb。

5．硬件帧头 hard_header

硬件一般都会在上层数据发送之前加上自己的硬件帧头，比如以太网（Ethernet）就有 14 字节的帧头。这个帧头是加在上层 ip、ipx 等数据包的前面的。驱动程序提供一个 hard_header 方法，协议层（ip、ipx、arp 等）在发送数据之前会调用这段程序。

硬件帧头的长度必须填在 dev→hard_header_len，这样协议层会在数据之前保留好硬件帧头的空间。这样 hard_header 程序只要调用 skb_push 然后正确填入硬件帧头就可以了。

在协议层调用 hard_header 时，传送的参数包括（2.0.xx）：数据的 sk_buff、device 指针、protocol、目的地址（daddr）、源地址（saddr）、数据长度（len）。

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

华清远见嵌入式培训中心——http://www.farsight.com.cn

其中，数据长度不使用 sk_buff 中的参数，因为调用 hard_header 时数据可能还没完全组织好。saddr 是 NULL 则使用缺省地址（default）。daddr 是 NULL 表明协议层不知道硬件目的地址。

如果 hard_header 完全填好了硬件帧头，则返回添加的字节数。如果硬件帧头中的信息还不完全，则返回负字节数，比如 daddr 为 NULL，但是帧头中需要目的硬件地址，典型的情况是以太网需要地址解析(arp)。当 hard_header 返回负数时，协议层会做进一步的 build header 工作。

目前，Linux 系统里就是通过 arp。如果 hard_header 返回正，dev→arp=1，表明不需要地址解析；返回负，dev→arp=0，需要地址解析。对 hard_header 的调用在每个协议层的处理程序里，如 ip_output。

6．地址解析 xarp

有些网络有硬件地址（比如 Ethernet），并且在发送硬件帧时需要知道目的硬件地址。这样就需要上层协议地址（ip、ipx）和硬件地址的对应。这个对应是通过地址解析完成的。

需要进行地址解析的设备在发送之前会调用驱动程序的 rebuild_header()方法。调用的主要参数包括指向硬件帧头的指针、协议层地址。如果驱动程序能够解析硬件地址，就返回 1，如果不能，返回 0。对 rebuild_header 的调用在 net/core/dev.c 的 do_dev_queue_xmit()里。

7．参数设置和统计数据

在驱动程序里还提供一些方法供系统对设备的参数进行设置和读取信息。一般只有超级用户（root）权限才能对设备参数进行设置。设置方法有：

```
dev->set_mac_address();
```

当用户调用 ioctl 类型为 SIOCSIFHWADDR 时，要设置这个设备的 mac 地址。对 mac 地址的设置一般没有太大意义的。

```
dev->set_config();
```

当用户调用 ioctl 时类型为 SIOCSIFMAP 时，系统会调用驱动程序的 set_config 方法。用户会传递一个 ifmap 结构，包含需要的 I/O、中断等参数。

```
dev->do_ioctl();
```

如果用户调用 ioctl 时类型在 SIOCDEVPRIVATE～SIOCDEVPRIVATE+15 之间，系统会调用驱动程序的这个方法。一般是设置设备的专用数据。

读取信息也是通过 ioctl 调用进行。除此之外驱动程序还可以提供一个 dev→get_stats 方法，返回一个 enet_statistics 结构，包含发送接收的统计信息。ioctl 的处理在 net/core/dev.c 的 dev_ioctl()和 dev_ifsioc()里。

8．网络驱动程序中用到的数据结构

最重要的是网络设备的数据结构。定义在 include/linux/netdevice.h 里。

```
struct device
```

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

华清远见嵌入式培训中心——http://www.farsight.com.cn

```
{
char *name;

unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
unsigned long base_addr;
unsigned char irq; /* 设备的 IRQ 号 */
/* 一些状态标志 */
volatile unsigned char start, /* 开始*/
interrupt; /* 中断*/
/* 在处理中断时 interrupt 设为 1，处理完清 0 */
unsigned long tbusy;
struct device *next;
/* 设备初始化仅执行一次*/
int (*init)(struct device *dev);
/* 一些硬件可以在一块板上支持多个接口，可能用到 if_port。 */
unsigned char if_port;
unsigned char dma; /* DMA 通道*/
struct enet_statistics* (*get_stats)(struct device *dev);
/* trans_start 记录最后一次成功发送的时间，可以用来确定硬件是否工作正常*/
unsigned long trans_start;
unsigned long last_rx;
/* flags 里面有很多内容，定义在 include/linux/if.h 里*/
unsigned short flags;
unsigned short family;
unsigned short metric;
unsigned short mtu;
/* type 标明物理硬件的类型，主要说明硬件是否需要 arp，定义在 include/linux/if_arp.h 里 */
unsigned short type;
/* 上层协议层根据 hard_header_len 在发送数据缓冲区前面预留硬件帧头空间*/
unsigned short hard_header_len;
/* priv 指向驱动程序自己定义的一些参数*/
/* 接口地址信息 */
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char pad; /* 使设备地址按 8 字节对齐 */
unsigned char dev_addr[MAX_ADDR_LEN]; /* hw 地址 */
unsigned char addr_len; /*硬件地址长度 */
unsigned long pa_addr; /* 协议地址 */
unsigned long pa_brdaddr; /*协议网卡地址 */
unsigned long pa_dstaddr; /* 协议 p-p 的目的地址 */
unsigned long pa_mask; /* 子网掩码 */
unsigned short pa_alen; /* 协议地址长度 */
struct dev_mc_list *mc_list; /* 多网卡列表*/
int mc_count;

struct ip_mc_list *ip_mc_list; /* IP 列表*/
__u32 tx_queue_len; /* 最大帧数*/
unsigned long pkt_queue; /* 包队列 */
struct device *slave; /* 从设备 */
struct net_alias_info *alias_info; /* 主设备信息 */
struct net_alias *my_alias;

/* 指向接口的缓冲 */
```

```
struct sk_buff_head buffs[DEV_NUMBUFFS];
/* 路由 */
int (*open)(struct device *dev);
int (*stop)(struct device *dev);
int (*hard_start_xmit) (struct sk_buff *skb,
struct device *dev);
int (*hard_header) (struct sk_buff *skb,
struct device *dev,
unsigned short type,
void *daddr,
void *saddr,
unsigned len);
int (*rebuild_header)(void *eth, struct device *dev,
unsigned long raddr, struct sk_buff *skb);
#define HAVE_MULTICAST
void (*set_multicast_list)(struct device *dev);
#define HAVE_SET_MAC_ADDR
int (*set_mac_address)(struct device *dev, void *addr);
#define HAVE_PRIVATE_IOCTL
int (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
int (*set_config)(struct device *dev, struct ifmap *map);
#define HAVE_HEADER_CACHE
void (*header_cache_bind)(struct hh_cache **hhp, struct device
*dev, unsigned short htype, __u32 daddr);
void (*header_cache_update)(struct hh_cache *hh, struct device
*dev, unsigned char * haddr);
#define HAVE_CHANGE_MTU
int (*change_mtu)(struct device *dev, int new_mtu);
struct iw_statistics* (*get_wireless_stats)(struct device *dev);
};
```

16.4.6 编写嵌入式 Linux 网络驱动程序中需要注意的问题

1．中断共享

嵌入式 Linux 系统运行几个设备共享同一个中断。需要共享的话，在申请的时候需指明共享方式。系统调用 request_irq()的定义如下：

```
int request_irq(unsigned int irq,
void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
unsigned long irqflags,
const char * devname,
void *dev_id);
```

如果共享中断，irqflags 设置 SA_SHIRQ 属性，这样就允许别的设备申请同一个中断。需要注意所有用到这个中断的设备在调用 request_irq()都必须设置这个属性。系统在回调每个中断处理程序时，可以用 dev_id 这个参数找到相应的设备。通常，dev_id 就设为 device 结构本身。系统处理共享中断是用各自的 dev_id 参数依次调用每一个中断处理程序。

2．硬件发送忙时的处理

主 CPU 的处理能力一般比网络发送要快，所以经常会遇到系统有数据要发，但上一包数

华清远见嵌入式培训中心——http://www.farsight.com.cn

据网络设备还没发送完。因为在 Linux 里网络设备驱动程序一般不做数据缓存，不能发送的数据都是通知系统发送不成功，所以必须要有一个机制在硬件不忙时及时通知系统接着发送下面的数据。

对发送忙的处理在前面设备的发送方法（hard_start_xmit）里已经描述过，即如果发送忙，置 tbusy 为 1。处理完发送数据后，在发送结束中断里清 tbusy，同时用 mark_bh()调用通知系统继续发送。

但在具体实现驱动程序时发现，这样的处理系统并不能及时地知道硬件已经空闲了，即在 mark_bh()以后，系统要等一段时间才会接着发送。造成发送效率很低。

3．流量控制（flow control）

网络数据的发送和接收都需要流量控制。这些控制是在系统里实现的，不需要驱动程序做工作。每个设备数据结构里都有一个参数 dev→tx_queue_len，这个参数标明发送时最多缓存的数据包。

在 Linux 系统里，以太网设备（10/100Mbps）tx_queue_len 一般设置为 100，串行线路（异步串口）为 10。实际上如果通过阅读源码可以知道，设置了 dev→tx_queue_len 并不是为缓存这些数据申请了空间。这个参数只是在收到协议层的数据包时判断发送队列里的数据是不是到了 tx_queue_len 的限度，以决定这一包数据加不加进发送队列。

发送时另一个方面的流控是更高层协议的发送窗口（TCP 协议里就有发送窗口）。达到了窗口大小，高层协议就不会再发送数据。接收流控也分两个层次。netif_rx()缓存的数据包有限制。另外高层协议也会有一个最大的等待处理的数据量。

注意　发送和接收流控处理在 net/core/dev.c 的 do_dev_queue_xmit()和 netif_rx()中。

4．调试

很多 Linux 的驱动程序都是编译进内核的，形成一个大的内核文件。但对调试来说，这是相当麻烦的。调试驱动程序可以用 module 方式加载。

支持模块方式的驱动程序必须提供两个函数：int init_module（void）和 void cleanup_module（void）。init_module()在加载此模块时调用，在这个函数里可以 register_netdev()注册设备。init_module()返回 0 表示成功，返回负表示失败。cleanup_module()在驱动程序被卸载时调用，清除占用的资源，调用 unregister_netdev()函数。

模块可以动态地加载、卸载。在 2.0.xx 版本里，还有 kerneld 自动加载模块，但是 2.2.xx 中已经取消了 kerneld。手工加载使用 insmod 命令，卸载用 rmmod 命令，查看内核中的模块用 lsmod 命令。

编译驱动程序用 gcc，主要命令行参数_DKERNEL _DMODULE，并且作为模块加载的驱动程序，只编译成 obj 形式（加-c 参数）。编译好的目标文件放在/lib/modules/2.x.xx/misc 下，在启动文件里用 insmod 加载。

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

16.5 网络驱动程序的测试

嵌入式 Linux 中的网络驱动程序没有对应的设备节点，读者可以通过观察/proc 系统中的/proc/net 信息来了解当前系统中的网络设备，如图 16.10 所示，列出了系统中的网络设备和数据包的收发状态。

```
[root@localhost root]# cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packe
ts errs drop fifo colls carrier compressed
    lo:      700      10    0    0    0     0          0         0      700
10    0    0    0    0       0          0
  eth0:        0       0    0    0    0     0          0         0     2052
 6    0    0    0    0       0          0
```

图 16.10　系统中的网路设备及其状态

16.5.1 嵌入式 Linux 的网络配置

嵌入式 Linux 的网络配置可以通过 ifconfig 和 route 两个命令完成，它们均可以在 busybox 包中取得。利用 ifconfig 命令配置 IP 地址的方法如下，如图 16.11 所示。

```
[root@localhost root]# ifconfig eth0 192.168.0.100
[root@localhost root]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:B5:2B:A4
          inet addr:192.168.0.100  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:2052 (2.0 Kb)
          Interrupt:10 Base address:0x10a4

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)

[root@localhost root]# _
```

图 16.11　ifconfig 配置 IP 地址

由图可见，“ifcongfig eth0 192.168.0.100”命令配置以太网设备（eth0）的 IP 地址为 192.168.0.100。配置完成后利用 ifconfig 命令即可看到，eth0 的 inet addr 已经变为 192.168.0.100 了。

ifconfig 命令还找到了另一个网络设备，即 lo（回环设备）。lo 设备是 Linux 系统中第一个网络设备，它在 Linux 网络中不可或缺，否则系统就无法接收来自本机的数据包。图 16.10 所示的回环设备的 inet addr 为 127.0.0.1。

实际上，常用的测试网络的方法是利用 ping 命令。当网络出现异常情况时，首先要做的就是看看是否能够 ping 通网络，具体过程如图 16.12 所示。

华清远见嵌入式培训中心——http://www.farsight.com.cn

```
[root@localhost root]# ping 192.168.0.100
PING 192.168.0.100 (192.168.0.100) 56(84) bytes of data.
64 bytes from 192.168.0.100: icmp_seq=1 ttl=64 time=2.63 ms
64 bytes from 192.168.0.100: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 192.168.0.100: icmp_seq=3 ttl=64 time=0.014 ms
64 bytes from 192.168.0.100: icmp_seq=4 ttl=64 time=0.015 ms
64 bytes from 192.168.0.100: icmp_seq=5 ttl=64 time=0.015 ms
64 bytes from 192.168.0.100: icmp_seq=6 ttl=64 time=0.014 ms
64 bytes from 192.168.0.100: icmp_seq=7 ttl=64 time=0.019 ms
64 bytes from 192.168.0.100: icmp_seq=8 ttl=64 time=0.015 ms

--- 192.168.0.100 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7007ms
rtt min/avg/max/mdev = 0.014/0.345/2.630/0.863 ms
[root@localhost root]# _
```

图 16.12　利用 ping 命令检测网络示意

默认情况下，ifconfig 会自动添加子网和路由（route）。但如果通过局域网连接到 internet，就需要管理员通过 route 命令配置网关等路由信息，在 PC 上，用户可以简单地执行 set up 或 netconfig 命令，就会跳出系统配置界面，然后可以根据个人需要对网络进行图形化配置，如图 16.13 所示。

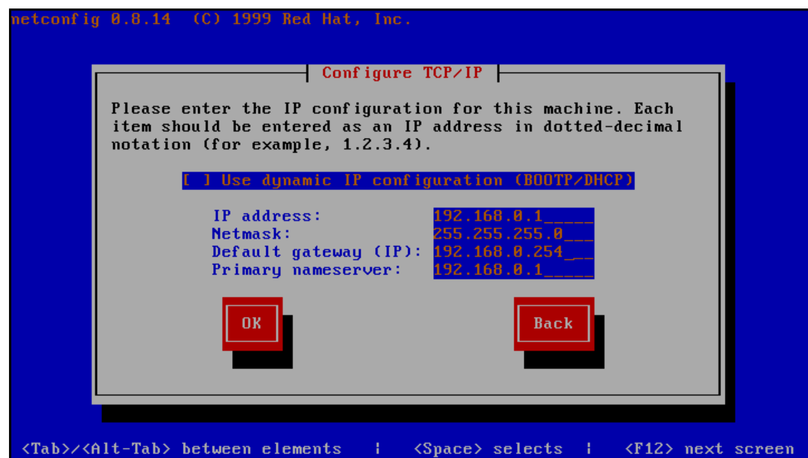

图 16.13　图形化配置系统网络

但是，嵌入式系统中移植的 Linux 中，往往并不支持图形化配置工具。不过不用担心，用户可以通过 route 命令设置这些网络信息。

```
#route add -net default gw 192.168.0.254 netmask 255.255.255.0
```

通过上面的命令，用户就可以设置默认路由，网关为 192.168.0.254，子网掩码为 255.255.255.0。现在，用户可以通过该路由访问网络了。

16.5.2　NFS 文件系统

NFS 是 Network File System 的缩写，即网络文件系统。它是由 SUN 公司开发，并在 1984 年推出的网络文件共享标准，主要为了实现在不同的系统间使用文件，因而它的通信协议和主机即 OS（操作系统）无关。

当用户想使用某个远程（remote）文件时，只要运行挂载命令 mount 就可以将远程的文件系统安装在自己的文件系统下。这样一来，对远程文件的操作和对本地文件的操作将没有

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

华清远见嵌入式培训中心——http://www.farsight.com.cn
什么区别，极大地方便了用户对远程文件的使用。

下面介绍如何配置 Linux 内核支持 NFS 网络文件系统。首先，在终端运行 setup 命令，选择 Networking support→Networking options→IP：kernel level autoconfiguration 配置内核支持 IP 层协议，如图 16.14 所示。

然后，配置内核的文件系统，过程如下，File systems→Network File Systems→Root file system on NFS，这就是 NFS 作为根文件系统的支持，如图 16.15 所示。

NFS 文件系统的使用使嵌入式系统的开发和调试工作变得非常方便。在 NFS 服务中，宿主机是被挂载端，为了远端客户机可以访问主机的文件，需要主机配置以下两方面内容。

华清远见

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

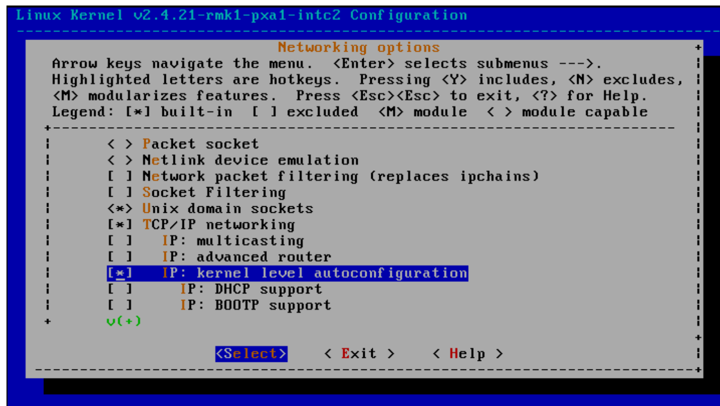

图 16.14 配置内核支持 IP 层协议

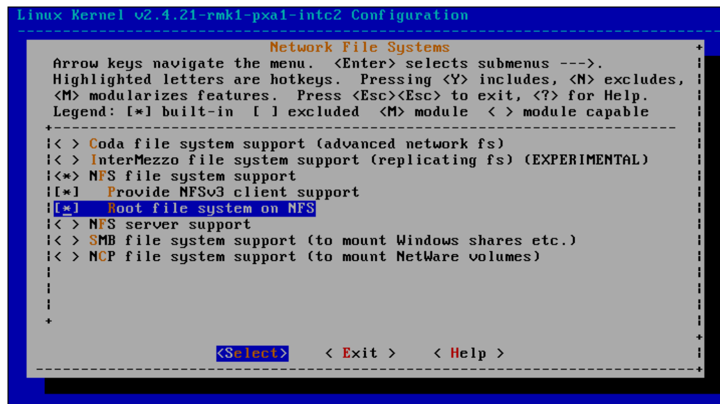

图 16.15 配置 NFS 根文件系统

1．打开主机的 NFS 服务，准许“指定用户”使用

首先，运行 setup 命令，选择 System services，将 nfs 一项选中（出现[*]表示选中），并去掉 ipchains 和 iptables 两项服务（即去掉它们前面的*号），然后退出，如图 16.16 和图 16.17 所示。

图 16.16 启动 NFS 系统服务

图 16.17 取消 iptables 服务和 ipchains 服务

“指定用户”是通过编辑文件“exports”完成的。首先在终端运行命令“vi /etc/exports”，在 exports 文件中加入：

```
/embedded_linux/nfs 192.168.0.50(rw,insecure,no_root_squash,no_all_squash)
```

然后，按<ESC>键，再输入“：wq”再回车，存储退出。其中，“/embedded_linux/nfs”是一个可以被 IP 地址是“192.168.0.50”计算机访问并可以被读/写的文件夹。

注意 可以通过更改这个 IP 值，让不同的计算机访问。

2．重新启动服务，使设置生效

在终端运行命令/etc/rc.d/init.d/nfs restart 即可使刚才的设置生效。

16.5.3 socket 编程

Linux 系统是通过提供套接字（socket）进行网络编程的。网络程序通过 socket 以及其他几个函数的配合，可以返回一个通信的文件描述符，系统可以将该文件描述符当成普通文件描述符来操作，这就是 Linux 设备无关性的好处所在。程序可以通过向描述符的读/写操作完成网络之间的数据通信。

1．socket()

```
int socket(int domain, int type, int protocol)
```

domain 说明网络程序所在的主机所采用的通信协议族，如 AF_Unix、AF_INET 等。AF_Unix 只能用于单一的 UNIX 系统进程间通信，而 AF_INET 是针对 Internet 使用的，可以在远程主机之间通信。

type 表示网络程序所采用的通信协议，如 SOCK_STREAM、SOCK_DGRAM 等。其中，SOCK_STREAM 表示使用 TCP 协议，系统提供有序、可靠、双向、面向连接的比特流。SOCK_DGRAM 表示使用 UDP 协议，此时系统仅提供定长、不可靠、无连接的通信。

华清远见嵌入式培训中心——http://www.farsight.com.cn

protocol 在指定了 type 参数之后，通常设置为 0 即可。

该函数为网络通信做最基本的准备工作，当成功时就返回文件描述符，否则返回-1，可以通过 errno 判断错误类型及原因。

2．bind()

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
```

- sockfd 是 socket 返回的文件描述符。
- addrlen 是 sockaddr 结构的长度。
- my_addr 是一个指向 sockaddr 的指针。

其中，sockaddr 结构体的定义如下：

```
struct sockaddr{
            unisgned short          as_family;
            char                            sa_data[14];
        };
```

但是，为了保证系统的兼容性，通常使用另外一个结构来代替，即 struct sockaddr_in()，其定义如下：

```
struct sockaddr_in{
            unisgned short          sin_family;
            unisgned short int          sin_port;
            struct  in_addr         sin_addr;
            unisgned char           sin_zero[8];
        };
```

3．listen()

```
int listen(int sockfd, int backlog)
```

- sockfd 是 bind 后的文件描述符。
- backlog 是请求排队的最大长度。当多个客户端程序和服务器相连时，该参数表示可以接受的排队长度。listen()函数将 bind()的文件描述符变为监听套接字，返回值与 bind 相同。

4．accept()

```
int accept(int sockfd, struct sockaddr *addr, int addrlen)
```

- sockfd 是 listen 后的文件描述符。
- addr 和 addrlen 由客户端的程序填写，服务器端仅传递指针即可。Bind、listen 和 accept 都是服务器端用的函数，当 accept 被调用时，服务器端会等待客户程序发出连接。当 accpet 成功时，就返回最后的服务器端文件描述符，这时，服务器端就可以向该描述符写信息了；当失败时，返回−1。

5．connect()

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
```

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

华清远见嵌入式培训中心——http://www.farsight.com.cn

- sockfd 是 socket 返回的文件描述符。
- serv_addr 保存了服务器端的连接信息，如 sin_addr 就是服务器端的地址。
- addrlen 是 sockaddr 结构的长度。

connect()函数是客户端程序用来和服务器端建立连接用的，成功时返回 0，否则返回-1。Sockfd 是同服务器端通信的文件描述符。

6．recv()

```
int recv（int sockfd，void * buf，int maxbuf，int options）
```

- sockfd 是 socket 返回的文件描述符。
- buf 为存放接收数据的缓冲区。
- maxbuf 表示缓冲区 buf 的大小。
- options 参数给出了一些选项，如 MSG_OOB、MSG_PEEK 等。recv()函数会回传收到的信息大小值，当出现错误传输时，会返回负值。

7．send()

```
int send（int sockfd，void *buffer，int msg_len，int options）
```

该函数中，sockfd、buffer 和 msg_len 与 recv()函数中的变量相同，仅是将要传输的信息先放入缓冲区 buffer。options 参数有 MSG_OOB、MSG_DONTROUTE 等选项，send()也会传回传输信息的大小值。

16.5.4 socket 编程实例

1．TCP 客户端

一个典型的客户端程序首先需要建立 socket 文件描述符，然后连接服务器，之后就可以读/写数据。该过程可以重复进行，直至读/写操作完全执行才关闭连接。

下面通过一个 TCP 客户端的实例程序来讲解 socket 编程的方法。在该程序中必须提供服务器端的主机名，并且要保证运行此程序前服务器已经正常工作。

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define RECVBUFSIZE 1024
int main(int argc, char *argv[])
{
 int sockfd;
 char buffer[RECVBUFSIZE];
 struct sockaddr_in server_addr;
 struct hostent *host;
```

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

```
    int portnumber,nbytes;

    if(argc!=3)
    {
     fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
     exit(1);
    }

    if((portnumber=atoi(argv[2]))<0)
    {
            fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
            exit(1);
    }

   /*客户程序开始建立 sockfd 描述符*/
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
    {
            fprintf(stderr,"Socket Error:%s\a\n",strerror(errno));
            exit(1);
    }

   /*客户程序填充服务端的资料*/
    bzero(&server_addr,sizeof(server_addr));
    server_addr.sin_family=AF_INET;
    server_addr.sin_port=htons(portnumber);
   server_addr.sin _ addr.s_addr = inet_addr(argv[1]);
   /*客户程序发起连接请求*/
    if(connect(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))
==-1)
   {
     fprintf(stderr,"Connect Error:%s\a\n",strerror(errno));
     exit(1);
   }
   /*连接成功*/
   if((nbytes=recv(sockfd,buffer, RECVBUFSIZE),0)==-1)
   {
            fprintf(stderr,"Read Error:%s\n",strerror(errno));
            exit(1);
    }
    buffer[nbytes]='\0';
    printf("I have received:%s\n",buffer);
   /*结束通信*/
    close(sockfd);
    exit(0);
   }
```

2. TCP 服务器端

TCP 服务器端的建立过程如下。

（1）通过 socket()函数建立套接口。

（2）通过 bind()函数绑定一个 IP 地址和端口地址，确定服务器的位置以便客户端访问。

（3）通过 listen()函数监听 listen 端口是否有新连接请求。

（4）通过 accept()函数接受新连接。

下面通过一个简单的服务器程序来理解 TCP 服务器编程的方法，该程序会向每个连接的客户端发送“Hello！”字符串。

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <errno.h>
    #include <string.h>
    #include <netdb.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <sys/socket.h>
    #define WAITBUF 10
    int main(int argc, char *argv[])
    {
     int sockfd,new_fd;
     struct sockaddr_in server_addr;
     struct sockaddr_in client_addr;
     int sin_size,portnumber;
     char hello[]="Hello!\n";
     if(argc!=2)
     {
      fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
      exit(1);
     }
     /*端口号不对，退出*/
     if((portnumber=atoi(argv[1]))<0)
     {
      fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
      exit(1);
     }
    /*服务器端开始建立 socket 描述符*/
     if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
     {
            fprintf(stderr,"Socket error:%s\n\a",strerror(errno));
            exit(1);
     }

    /*服务器端填充 sockaddr 结构*/
     bzero(&server_addr,sizeof(struct sockaddr_in));
     server_addr.sin_family=AF_INET;
    /*自动填充主机 IP*/
     server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
     server_addr.sin_port=htons(portnumber);
    /*捆绑 sockfd 描述符*/
     if(bind(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))
==-1)
     {
            fprintf(stderr,"Bind error:%s\n\a",strerror(errno));
            exit(1);
     }
     /*监听 sockfd 描述符*/
     if(listen(sockfd, WAITBUF)==-1)
```

```
    {
            fprintf(stderr,"Listen error:%s\n\a",strerror(errno));
            exit(1);
    }
    while(1)
    {
            /*服务器阻塞，直到客户程序建立连接*/
            sin_size=sizeof(struct sockaddr_in);
            if((new_fd=accept(sockfd,(struct sockaddr *)(&client_addr),&sin_size))
==-1)
            {
                    fprintf(stderr,"Accept error:%s\n\a",strerror(errno));
                    exit(1);
       }
       /*可以在这里加上自己的处理函数*/
           fprintf(stderr,"Server get connection from %s\n",
                                  inet_ntoa(client_addr.sin_addr));
           if(send(new_fd,hello,strlen(hello),0)==-1)
           {
            fprintf(stderr,"Write Error:%s\n",strerror(errno));
            exit(1);
           }
    /*这个通信已经结束*/
          close(new_fd);
    /*循环下一个*/
     }
    close(sockfd);
    exit(0);
   }
```

3．UDP 客户端

UDP 客户端必须知道一个服务器端绑定的端口和 IP 地址，以便发送数据。下面给出一个 UDP 客户端的程序实例。其中，程序需要一个服务器的主机名。

```
/*客户端程序 UDPClient.c，使用方法 UDPClient ServerIP ServerPort*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#define MAX_BUF_SIZE    1024

void udpc_requ(int sockfd,const struct sockaddr_in *addr,int len)
{
        char buffer[MAX_BUF_SIZE];
        int n;
        while(1)
        {       /*从键盘读入，写到服务端*/
                fgets(buffer,MAX_BUF_SIZE,stdin);
                sendto(sockfd,buffer,strlen(buffer),0,addr,len);
                bzero(buffer,MAX_BUF_SIZE);
                /*从网络上读，写到屏幕上*/
```

```
                n=recvfrom(sockfd,buffer,MAX_BUF_SIZE,0,NULL,NULL);
                buffer[n]=0;
                fputs(buffer,stdout);
        }
}

int main(int argc,char **argv)
{
        int sockfd,port;
        struct sockaddr_in     addr;

        if(argc!=3)
        {
                fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
                exit(1);
        }

        if((port=atoi(argv[2]))<0)
        {
                fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
                exit(1);
        }

        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        if(sockfd<0)
        {
                fprintf(stderr,"Socket  Error:%s\n",strerror(errno));
                exit(1);
        }
        /*填充服务器端的资料*/
        bzero(&addr,sizeof(struct sockaddr_in));
        addr.sin_family=AF_INET;
        addr.sin_port=htons(port);
        if(inet_aton(argv[1],&addr.sin_addr)<0)
        {
                fprintf(stderr,"Ip error:%s\n",strerror(errno));
                exit(1);
         }
         udpc_requ(sockfd,&addr,sizeof(struct sockaddr_in));
         close(sockfd);
}
```

4．UDP 服务器端

UDP 同样可以建立一个套接口并将其绑定到给定的地址，但是 UDP 的服务器端不负责监听和接受外来连接，客户也不必显示地连接到服务器。

下面通过一个 UDP 服务端的程序实例来进一步理解 UDP 的编程方法，该程序可以打印客户端发送的信息。

```
/*服务端程序 UDPServer.c*/
#include <sys/types.h>
#include <sys/socket.h>
```

华清远见嵌入式培训中心——http://www.farsight.com.cn

```
#include <netinet/in.h>
#include <stdio.h>
#include <errno.h>
#define SERVER_PORT     8888
#define MAX_MSG_SIZE    1024

void udps_respon(int sockfd)
{
        struct sockaddr_in addr;
        int     addrlen,n;
        char    msg[MAX_MSG_SIZE];

        while(1)
        {       /*等待数据请求*/
                n=recvfrom(sockfd,msg,MAX_MSG_SIZE,0,
                        (struct sockaddr*)&addr,&addrlen);
                msg[n]=0;
                /*显示服务器端已经收到了信息*/
                fprintf(stdout,"I have received %s",msg);
                /*数据回送*/
                sendto(sockfd,msg,n,0,(struct sockaddr*)&addr,addrlen);
        }
}

int main(void)
{
        int sockfd;
        struct sockaddr_in      addr;

        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        if(sockfd<0)
        {
        /*如果不能得到一个端口的文件描述符，返回错误信息*/
                fprintf(stderr,"Socket Error:%s\n",strerror(errno));
                exit(1);
        }
        bzero(&addr,sizeof(struct sockaddr_in));//清 0
        addr.sin_family=AF_INET;
        addr.sin_addr.s_addr=htonl(INADDR_ANY);
        addr.sin_port=htons(SERVER_PORT);
        if(bind(sockfd,(struct sockaddr *)&ddr,sizeof(struct sockaddr_in))<0)
        {
                fprintf(stderr,"Bind Error:%s\n",strerror(errno));
                exit(1);
        }
        udps_respon(sockfd);
        close(sockfd);
}
```

5．代理服务器

代理服务器是网络通信中常用的一种通信服务，它通过指定一个端口执行代理服务，以

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

华清远见嵌入式培训中心——http://www.farsight.com.cn
代理远端服务器的服务端口。

例如，当在主机名为“ProxyServer”的服务器 8000 端口运行代理服务来代理远端主机 TelnetServer 上的 Telnet 服务时，客户机就可以通过输入“Telnet Proxy Server 8000”使用 TelnetServer 的 Telnet 服务。

代理服务器源代码 proxy.c 如下：

```
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <netdb.h>
#define TCP_PROTO   "tcp"
int proxy_port;                           /*全局变量，指定代理服务器的端口*/
struct sockaddr_in hostaddr;              /*全局变量，远端主机地址*/
extern int errno;
extern char *sys_myerrlist[];
void parse_args (int argc, char **argv);    /*参数转换函数*/
void daemonize (int servfd);                /*创建守护进程函数*/
void do_proxy (int usersockfd);             /*代理处理函数*/
void errorout (char msg);                   /*错误输出函数*/
/*主函数*/
main (argc,argv)
int argc;
char **argv;
{
      int clilen;
      int childpid;
      int sockfd, newsockfd;
      struct sockaddr_in servaddr, cliaddr;
      /*把命令行参数转存到全局变量中*/
      parse_args(argc,argv);
      /*为侦听客户请求准备一个地址*/
      bzero((char *) &servaddr, sizeof(servaddr));
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
      servaddr.sin_port = proxy_port;
      /*得到一个端口的文件描述符*/
      if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
          fputs("failed to create server socket\r\n",stderr);
          exit(1);
      }
      /*绑定到前面的地址上*/
      if  (bind(sockfd,(struct sockaddr_in *) &servaddr,sizeof(servaddr)) < 0) {
         fputs("faild to bind server socket to specified port\r\n",stderr);
         exit(1);
      }
```

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org

```
/*准备接收*/
      listen(sockfd,5);
    /*把自身变为守护进程*/
    daemonize(sockfd);
    /*进入一个循环，并发处理连接请求*/
    while (1) {
         /*接受连接请求*/
         clilen = sizeof(cliaddr);
         newsockfd = accept(sockfd, (struct sockaddr_in *) &cliaddr, &clilen);
         if (newsockfd < 0 && errno == EINTR)
             continue;
           else if (newsockfd < 0)
             /*出错，kill 服务器*/
         errorout("failed to accept connection");
         /*产生一个子进程，进行连接处理*/
         if ((childpid = fork()) == 0) {
              close(sockfd);
              do_proxy(newsockfd);        /*真正的处理过程*/
              exit(0);
           }
         /*如果产生子进程失败，连接将被丢掉*/
          close(newsockfd);
          }
       }

/*****************************************************************
进行参数转换，把从命令行得到的参数值赋给全局变量
*****************************************************************/
void parse_args (argc,argv)
int argc;
char **argv;
{
    int i;
    struct hostent *hostp;
    struct servent *servp;
    unsigned long inaddr;
    struct {
        char proxy_port [16];
        char isolated_host [64];
        char service_name [32];
    } pargs;
/*输入不合规范*/
    if (argc < 4) {
           printf("usage: %s <proxy-port> <host> <service-name|port-number>\
r\n", argv[0]);
           exit(1);
    }
/*将输入参数先放到自定义的数据结构中*/
    strcpy(pargs.proxy_port,argv[1]);
    strcpy(pargs.isolated_host,argv[2]);
    strcpy(pargs.service_name,argv[3]);
/*检查端口号是否是数字，再赋给 proxy_port*/
    for (i = 0; i < strlen(pargs.proxy_port); i++)
         if (!isdigit(*(pargs.proxy_port + i)))
              break;
```

```
        if (i == strlen(pargs.proxy_port))
            proxy_port = htons(atoi(pargs.proxy_port));
        else {
            printf("%s: invalid proxy port\r\n",pargs.proxy_port);
            exit(0);
        }
    /*把远端服务器地址赋给 hostaddr*/
        bzero(&hostaddr,sizeof(hostaddr));
        hostaddr.sin_family = AF_INET;
    /*不管是主机名还是 IP 地址，都把它转换为 hostaddr 的地址*/
        if ((inaddr = inet_addr(pargs.isolated_host)) != INADDR_NONE)
            bcopy(&inaddr,&hostaddr.sin_addr,sizeof(inaddr));
        else if ((hostp = gethostbyname(pargs.isolated_host)) != NULL)
            bcopy(hostp->h_addr,&hostaddr.sin_addr,hostp->h_length);
        else {
            printf("%s: unknown host\r\n",pargs.isolated_host);
            exit(1);
        }
    /*不管是用数字表示端口还是用服务表示的端口，都把它转换后赋给 hostaddr.sin_port*/
        if ((servp = getservbyname(pargs.service_name,TCP_PROTO)) != NULL)
            hostaddr.sin_port = servp->s_port;
        else if (atoi(pargs.service_name) > 0)
            hostaddr.sin_port = htons(atoi(pargs.service_name));
        else {
            printf("%s: invalid/unknown service name or port number\r\n", pargs.
service_name);
            exit(1);
        }
    }

    /****************************************************************
    创建守护进程函数
    ****************************************************************/
    void daemonize (servfd)
    int servfd;
    {
        int childpid, fd, fdtablesize;
        /*忽略终端 I/O 读、写和 stop 信号*/
          signal(SIGTTOU,SIG_IGN);
          signal(SIGTTIN,SIG_IGN);
          signal(SIGTSTP,SIG_IGN);
        /*通过 fork 子进程，kill 父进程，把自身转入后台*/
        if ((childpid = fork()) < 0) {
            fputs("failed to fork first child\r\n",stderr);
            exit(1);
          }
        else if (childpid > 0)
          exit(0); /*若是父进程，结束*/
          /*设为会话组长，摆脱原终端*/
        setsid(0) ;

        /*释放控制终端*/
        if ((fd = open("/dev/tty",O_RDWR)) >= 0) {
            ioctl(fd,TIOCNOTTY,NULL);
```

```
        close(fd);
    }
    /*关闭除 servfd 外的所有文件描述符*/
    for (fd = 0, fdtablesize = getdtablesize(); fd < fdtablesize; fd++)
    if (fd != servfd)
       close(fd);
       /*改变工作目录到根目录*/
       chdir("/");
       /*重设文件掩码*/
       umask(0);
}
/*****************************************************
代理处理函数
*****************************************************************/
void do_proxy (usersockfd)
int usersockfd;
{
    int isosockfd;
    fd_set rdfdset;
    int connstat;
    int iolen;
    char buf[2048];
    /*作为一个客户端，新开一个端口以连接远端服务器*/
    if ((isosockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
    errorout("failed to create socket to host");
    /*发连接请求*/
    connstat = connect(isosockfd,(struct sockaddr *) &hostaddr, sizeof(
hostaddr));
     /*出错处理*/
    switch (connstat) {
    case 0:
    break;
    case ETIMEDOUT:
    case ECONNREFUSED:
    case ENETUNREACH:
    strcpy(buf,sys_myerrlist[errno]);
    strcat(buf,"\r\n");
    write(usersockfd,buf,strlen(buf));
    close(usersockfd);
    exit(1);
    break;
    default:
    errorout("failed to connect to host");
    }
    /*现在已经建立了连接，进入代理的数据反馈循环*/
    while (1) {
         /* 使用 select 进行并发处理 */
         FD_ZERO(&rdfdset);
         FD_SET(usersockfd,&rdfdset);
         FD_SET(isosockfd,&rdfdset);
    if (select(FD_SETSIZE,&rdfdset,NULL,NULL,NULL) < 0)
      errorout("select failed");
      /*客户端有数据发过来吗? */
    if (FD_ISSET(usersockfd,&rdfdset)) {
          /*小于或等于 0 意味着客户端已断*/
```

```
            if ((iolen = recv(usersockfd,buf,sizeof(buf)),0) <= 0)
              break;
              send(isosockfd,buf,iolen,0);
              /*把数据复制一份发给服务器端*/
          }
        /*远端服务器有数据发过来吗？*/
        if (FD_ISSET(isosockfd,&rdfdset)) {
            f ((iolen = recv(isosockfd,buf,sizeof(buf))) <= 0)
                break; /*接收数据长度小于或等于 0，则表明连接已断*/
                send(usersockfd,buf,iolen);
                /*把数据复制一份发给客户端*/
            }
          }
          close(isosockfd);
          close(usersockfd);
        }

    /****************************************************************
    出错处理函数
    ****************************************************************/
    void errorout (msg)
    char *msg;
    {
        FILE *console;
        console = fopen("/dev/console","a");
        fprintf(console,"proxyd: %s\r\n",msg);
        fclose(console);
        exit(1);
    }
```

当程序编译完成后，目录下会出现一个可执行文件，假设为 Proxy。用户只需要输入“#./proxy <端口> <代理服务器 IP> <代理服务器的 telnet 端口>”命令就可以通过代理服务器上网了。

16.6 知识索引

1．以太网

以太网（Ethernet）是由 Xerox 公司创建并与 Intel 和 DEC 公司联合开发的基带局域网规范。与 IEEE802.3 系列标准相类似，并不是一种具体的网络，而只是一种技术规范。

2．CSMA/CD

以太网采用带冲突检测的载波帧听多路访问（CSMA/CD）机制，其中的任意节点均可以看到在网络中发送的所有信息。因此，以太网也可说是一种广播网络。

华清远见嵌入式培训中心——http://www.farsight.com.cn

3．TCP/IP 分层

TCP/IP 网络协议分为网络接口层、网间层、传输层和应用层。

4．以太网接口控制器

以太网接口控制器主要由 MAC（Media Access Layer）和 PHY（Physical Layer）两部分组成。其中，MAC 层又叫媒体访问层，主要负责逻辑控制且容易集成在处理器内部。

5．Linux 系统的设备

Linux 系统的设备分为字符设备(char device)、块设备(block device)和网络设备(network device)3 种。网络设备在 Linux 里做专门的处理。Linux 的网络系统主要是基于 BSD UNIX 的 socket（套接字）机制。

6．Linux 的网络配置

Linux 的网络配置可以通过 ifconfig 和 route 两个命令完成，它们均可以在 busybox 包中取得。利用 ifconfig 命令可以配置系统的 IP 地址。

7．NFS

NFS 是 Network File System 的缩写，即网络文件系统。使用 NFS 文件系统可以使嵌入式系统的开发和调试工作变得非常方便。

8．套接字

Linux 系统是通过提供套接字（socket）进行网络编程的。网络程序通过 socket 以及其他几个函数的配合，可以返回一个通信的文件描述符，系统可以将该文件描述符当成普通文件描述符来操作，这就是 Linux 设备无关性的好处所在。程序可以通过向描述符的读/写操作完成网络之间的数据通信。

16.7　思考与练习

1．简述以太网的发展。
2．画出 TCP/IP 网络模型，说明各层次的作用。
3．什么是套接字？它有什么特性？
4．画出 socket 编程中数据流通信的服务器端与客户端通信流程。
5．画出代理服务器的程序流程图。

推荐《华清远见嵌入式培训中心》课程

华清远见嵌入式培训中心网址：　http://www.farsight.com.cn/

华清远见嵌入式培训中心——http://www.farsight.com.cn

嵌入式 linux 驱动开发初级班：http://www.farsight.com.cn/courses/TS-LinuxDriver.htm
嵌入式 linux 驱动开发高级班：http://www.farsight.com.cn/courses/TS-LinuxDriver2.htm

关于《华清远见嵌入式培训中心》

华清远见是一家以嵌入式技术培训为核心业务的高科技企业，主要从事高端嵌入式 linux、Wince、Vxworks、ARM、DSP、FPGA、symbian、cadence 等方面的技术培训。目前为中国软件行业协会嵌入式分会单位、清华大学合作培训机构、首家获得“高新技术企业认定证书”的 IT 培训机构，微软嵌入式合作伙伴，并于 2007 年先后成为 ARM 全球授权培训中心和 Symbian 授权培训中心。目前推出的主要业务有短期高端培训、长期就业培训（嵌入式学院）、企业内训以及项目开发 4 部分。作为嵌入式培训领域的专家，华清远见嵌入式培训中心一直致力于嵌入式技术的宣传与推广。通过公司的培训业务平台，华清远见曾为国内数十家大型企业（包括华为、摩托罗拉、三星、松下、NEC 等）成功实施过技术培训，并和多家企业达成了长期合作关系，每年有数千名技术人员受益于华清远见的技术研讨会、公开出版物、远程教学以及专题培训等，得到了业界的广泛认可。

华清远见

嵌入式学院（华清远见旗下品牌）——http://www.embedu.org