“黑色经典”系列之《嵌入式 Linux 应用程序开发详解》

第 10 章 嵌入式 Linux 网络编程

本章目标

本章将介绍嵌入式 Linux 网络编程的基础知识。由于网络在嵌入式中的应用非常广泛，基本上常见的应用都会与网络有关，因此，掌握这一部分的内容是非常重要的。经过本章的学习，读者将会掌握以下内容。

- 掌握 TCP/IP 协议的基触知识
- 掌握嵌入式 Linux 基础网络编程
- 掌握嵌入式 Linux 高级网络编程
- 分析理解 Ping 源代码
- 能够独立编写客户端、服务器端的通信程序
- 能够独立编写 NTP 协议实现程序

10.1 TCP/IP 协议概述

10.1.1 OSI 参考模型及 TCP/IP 参考模型

读者一定都听说过著名的 OSI 协议参考模型，它是基于国际标准化组织（ISO）的建议发展起来的，从上到下共分为 7 层：应用层、表示层、会话层、传输层、网络层、数据链路层及物理层。这个 7 层的协议模型虽然规定得非常细致和完善，但在实际中却得不到广泛的应用，其重要的原因之一就在于它过于复杂。但它仍是此后很多协议模型的基础，这种分层架构的思想在很多领域都得到了广泛的应用。

与此相区别的 TCP/IP 协议模型从一开始就遵循简单明确的设计思路，它将 TCP/IP 的 7 层协议模型简化为 4 层，从而更有利于实现和使用。TCP/IP 的协议参考模型和 OSI 协议参考模型的对应关系如下图 10.1 所示。

下面分别对者 TCP/IP 的 4 层模型进行简要介绍。

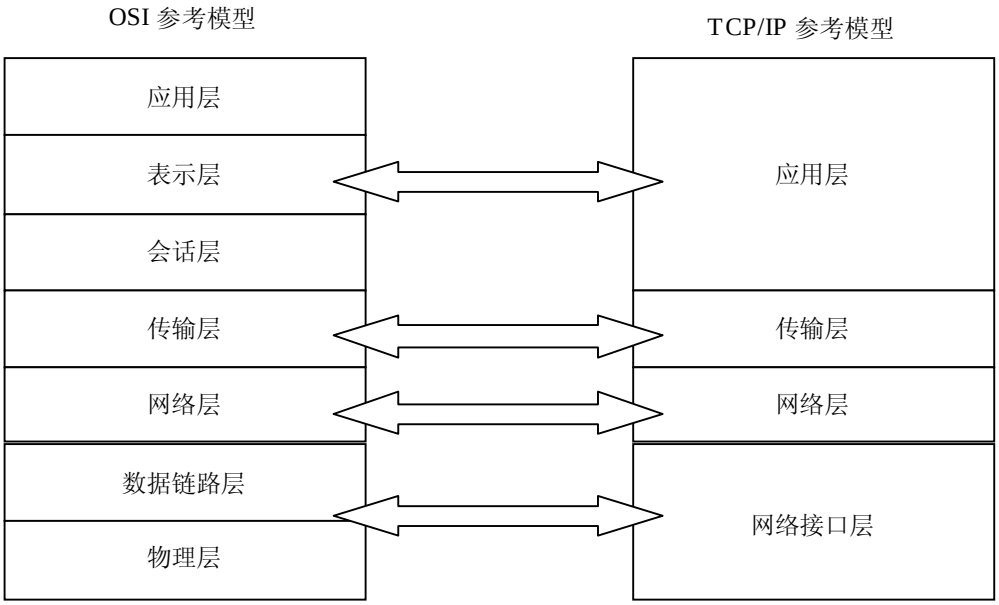

图 10.1 OSI 模型和 TCP/IP 参考模型对应关系

- 网络接口层：负责将二进制流转换为数据帧，并进行数据帧的发送和接收。要注意的是数据帧是独立的网络信息传输单元。
- 网络层：负责将数据帧封装成 IP 数据报，并运行必要的路由算法。
- 传输层：负责端对端之间的通信会话连接与建立。传输协议的选择根据数据传输方式而定。
- 应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。

10.1.2 TCP/IP 协议族

虽然 TCP/IP 名称只包含了两个协议，但实际上，TCP/IP 是一个庞大的协议族，它包括了各个层次上的众多协议，图 10.2 列举了各层中一些重要的协议，并给出了各个协议在不同层次中所处的位置如下。

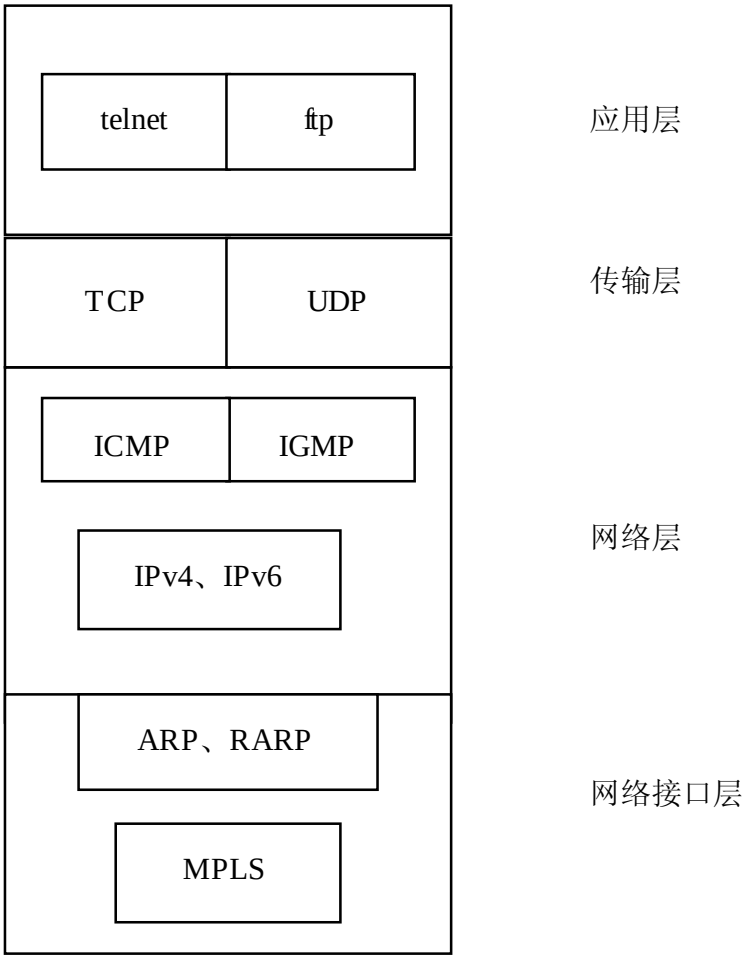

图 10.2 TCP/IP 协议族

- ARP：用于获得同一物理网络中的硬件主机地址。
- MPLS：多协议标签协议，是很有发展前景的下一代网络协议。
- IP：负责在主机和网络之间寻址和路由数据包。
- ICMP：用于发送报告有关数据包的传送错误的协议。
- IGMP：被 IP 主机用来向本地多路广播路由器报告主机组成员的协议。
- TCP：为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。
- UDP：提供了无连接通信，且不对传送包进行可靠的保证。适合于一次传输少量数据，

可靠性则由应用层来负责。

10.1.3 TCP 和 UDP

在此主要介绍在网络编程中涉及到的传输层 TCP 和 UDP 协议。

1．TCP

（1）概述

同其他任何协议栈一样，TCP 向相邻的高层提供服务。因为 TCP 的上一层就是应用层，因此，TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递。应用程序通过编程调用 TCP 并使用 TCP 服务，提供需要准备发送的数据，用来区分接收数据应用的目的地址和端口号。

通常应用程序通过打开一个 socket 来使用 TCP 服务，TCP 管理到其他 socket 的数据传递。可以说，通过 IP 的源/目的可以惟一地区分网络中两个设备的关联，通过 socket 的源/目的可以惟一地区分网络中两个应用程序的关联。

（2）三次握手协议

TCP 对话通过三次握手来初始化的。三次握手的目的是使数据段的发送和接收同步，告诉其他主机其一次可接收的数据量，并建立虚连接。

下面描述了这三次握手的简单过程。

- 初始化主机通过一个同步标志置位的数据段发出会话请求。
- 接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发送的数据段的起始字节的顺序号、应答并带有将收到的下一个数据段的字节顺序号。
- 请求主机再回送一个数据段，并带有确认顺序号和确认号。

图 10.3 就是这个流程的简单示意图。

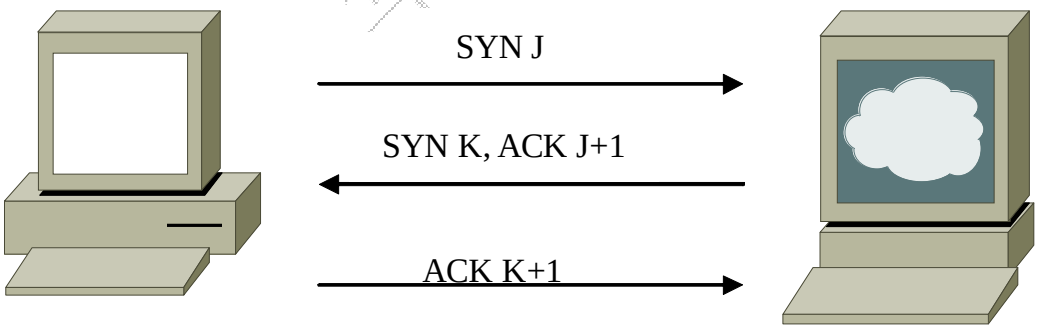

图 10.3　TCP 三次握手协议

TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体向回发送一个数据报，其中包含有一个确认序号，它意思是希望收到的下一个数据报的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据报。

（3）TCP 数据报头

图 10.4 给出了 TCP 数据报头的格式。

TCP 数据报头的含义如下所示。

- 源端口、目的端口：16 位长。标识出远端和本地的端口号。

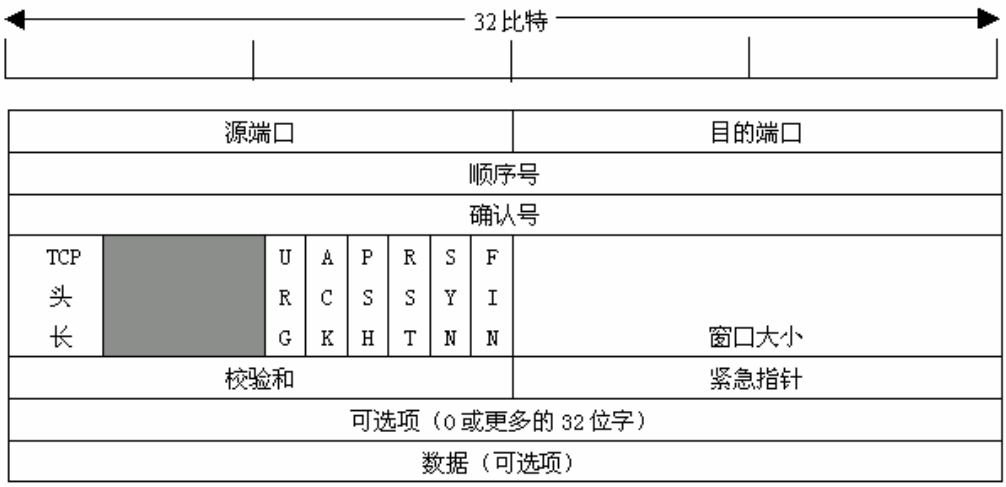

图 10.4　TCP 数据报头的格式

- 序号：32 位长。标识发送的数据报的顺序。
- 确认号：32 位长。希望收到的下一个数据报的序列号。
- TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。接收方因此请求数据报一到便可送往应用程序而不必等到缓冲区装满时才传送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误的连接。还可以用于拒绝非法的数据报或拒绝连接请求。
- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。
- 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字。包括最大 TCP 载荷，窗口比例、选择重发数据报等选项。

2．UDP

（1）概述

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可同时作为应用的客户或服务器方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。

UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是在网络质量越来越高的今天，UDP 的应用得到了大大的增强。它比 TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议。

（2）UDP 数据包头

UDP 数据包头如下图 10.5 所示。

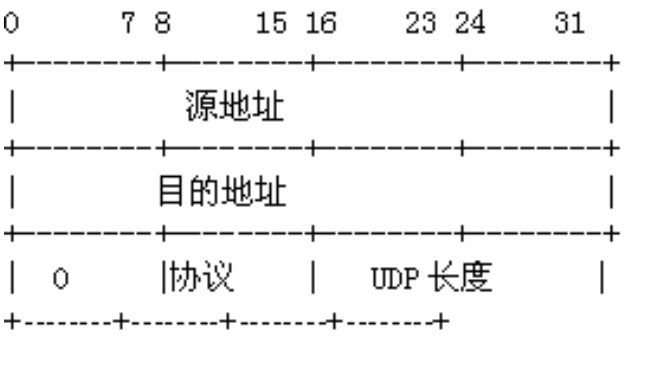

图 10.5 UDP 数据包头

- 源地址、目的地址：16 位长。标识出远端和本地的端口号。
- 数据报的长度是指包括报头和数据部分在内的总的字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。

3．协议的选择

协议的选择应该考虑到以下 3 个方面。

（1）对数据可靠性的要求

对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。

（2）应用的实时性

由于 TCP 协议在传送过程中要进行三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VOIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。

（3）网络的可靠性

由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此，在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要再采用 TCP 协议，选择 UDP 协议来减少网络负荷。

10.2 网络基础编程

10.2.1 socket 概述

1．socket 定义

在 Linux 中的网络编程是通过 socket 接口来进行的。人们常说的 socket 接口是一种特殊的 I/O，它也是一种文件描述符。每一个 socket 都用一个半相关描述{协议，本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议，本地地址、本地端口、远程地址、远程端口}。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的。

2．socket 类型

常见的 socket 有 3 种类型如下。

（1）流式 socket（SOCK_STREAM）

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺序性。

（2）数据报 socket（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，并且不保证是可靠、无差错的。它使用数据报协议 UDP。

（3）原始 socket

原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

10.2.2 地址及顺序处理

1．地址结构相关处理

（1）数据结构介绍

下面首先介绍两个重要的数据类型：sockaddr 和 sockaddr_in，这两个结构类型都是用来保存 socket 信息的，如下所示：

```
struct sockaddr {
      unsigned short sa_family; /*地址族*/
      char sa_data[14]; /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};
struct sockaddr_in {
      short int sa_family; /*地址族*/
      unsigned short int sin_port; /*端口号*/
      struct in_addr sin_addr; /*IP 地址*/
```

```
        unsigned char sin_zero[8]; /*填充 0 以保持与 struct sockaddr 同样大小*/
    };
```

这两个数据类型是等效的，可以相互转化，通常 sockaddr_in 数据类型使用更为方便。在建立 socketadd 或 sockaddr_in 后，就可以对该 socket 进行适当的操作了。

（2）结构字段

表 10.1 列出了该结构 sa_family 字段可选的常见值。

表 10.1

结构定义头文件	#include <netinet/in.h>
Sa_family	AF_INET：IPv4 协议
	AF_INET6：IPv6 协议
	AF_LOCAL：UNIX 域协议
	AF_LINK：链路地址协议
	AF_KEY：密钥套接字（socket）

对了解 sockaddr_in 其他字段的含义非常清楚，具体的设置涉及到其他函数，在后面会有详细讲解。

2．数据存储优先顺序

（1）函数说明

计算机数据存储有两种字节优先顺序：高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输，因此在有些情况下，需要对这两个字节存储优先顺序进行相互转化。这里用到了四个函数：htons、ntohs、htonl、ntohl。这四个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host，n 代表 network，s 代表 short，l 代表 long。通常 16 位的 IP 端口号用 s 代表，而 IP 地址用 l 来代表。

（2）函数格式说明

表 10.2 列出了这 4 个函数的语法格式。

表 10.2　　htons 等函数语法要点

所需头文件	#include <netinet/in.h>
函数原型	uint16_t htons(unit16_t host16bit) uint32_t htonl(unit32_t host32bit) uint16_t ntohs(unit16_t net16bit) uint32_t ntohs(unit32_t net32bit)
函数传入值	host16bit：主机字节序的 16bit 数据
	host32bit：主机字节序的 32bit 数据
	net16bit：网络字节序的 16bit 数据
	net32bit：网络字节序的 32bit 数据

函数返回值	成功：返回要转换的字节序
	出错：–1

注意 调用该函数只是使其得到相应的字节序，用户不需清楚该系统的主机字节序和网络字节序是否真正相等。如果是相同不需要转换的话，该系统的这些函数会定义成空宏。

3．地址格式转化

（1）函数说明

通常用户在表达地址时采用的是点分十进制表示的数值（或者是以冒号分开的十进制 IPv6 地址），而在通常使用的 socket 编程中所使用的则是二进制值，这就需要将这两个数值进行转换。这里在 IPv4 中用到的函数有 inet_aton、inet_addr 和 inet_ntoa，而 IPv4 和 IPv6 兼容的函数有 inet_pton 和 inet_ntop。由于 IPv6 是下一代互联网的标准协议，因此，本书讲解的函数都能够同时兼容 IPv4 和 IPv6，但在具体举例时仍以 IPv4 为例。

这里 inet_pton 函数是将点分十进制地址映射为二进制地址，而 inet_ntop 是将二进制地址映射为点分十进制地址。

（2）函数格式

表 10.3 列出了 inet_pton 函数的语法要点。

表 10.3　　inet_pton 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_pton(int family,const char *strptr, void *addrptr)	
函数传入值	family	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
	strptr：要转化的值	
	addrptr：转化后的地址	
函数返回值	成功：0	
	出错：–1	

表 10.4 列出了 inet_ntop 函数的语法要点。

表 10.4　　inet_ntop 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_ntop(int family, void *addrptr, char *strptr, size_t len)	
函数传入值	family	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
	addrptr：转化后的地址	
	strptr：要转化的值	

	Len：转化后值的大小
函数返回值	成功：0
	出错：−1

4．名字地址转化

（1）函数说明

通常，人们在使用过程中都不愿意记忆冗长的 IP 地址，尤其到 IPv6 时，地址长度多达 128 位，那时就更加不可能一次次记忆那么长的 IP 地址了。因此，使用主机名将会是很好的选择。在 Linux 中，同样有一些函数可以实现主机名和地址的转化，最为常见的有 gethostbyname、gethostbyaddr、getaddrinfo 等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 gethostbyname 是将主机名转化为 IP 地址，gethostbyaddr 则是逆操作，是将 IP 地址转化为主机名，另外 getaddrinfo 还能实现自动识别 IPv4 地址和 IPv6 地址。

gethostbyname 和 gethostbyaddr 都涉及到一个 hostent 的结构体，如下所示：

```
Struct hostent{
      char *h_name;/*正式主机名*/
      char **h_aliases;/*主机别名*/
      int h_addrtype;/*地址类型*/
      int h_length;/*地址长度*/
      char **h_addr_list;/*指向 IPv4 或 IPv6 的地址指针数组*/
}
```

调用该函数后就能返回 hostent 结构体的相关信息。

getaddrinfo 函数涉及到一个 addrinfo 的结构体，如下所示：

```
struct addrinfo{
      int ai_flags;/*AI_PASSIVE,AI_CANONNAME;*/
      int ai_family;/*地址族*/
      int ai_socktype;/*socket 类型*/
      int ai_protocol;/*协议类型*/
      size_t ai_addrlen;/*地址长度*/
      char *ai_canoname;/*主机名*/
      struct sockaddr *ai_addr;/*socket 结构体*/
      struct addrinfo *ai_next;/*下一个指针链表*/
}
```

hostent 结构体而言，addrinfo 结构体包含更多的信息。

（2）函数格式

表 10.5 列出了 gethostbyname 函数的语法要点。

表 10.5　　gethostbyname 函数语法要点

所需头文件	#include <netdb.h>
函数原型	Struct hostent *gethostbyname(const char *hostname)
函数传入值	Hostname：主机名
函数返回值	成功：hostent 类型指针
	出错：−1

调用该函数时可以首先对 addrinfo 结构体中的 h_addrtype 和 h_length 进行设置，若为 IPv4 可设置为 AF_INET 和 4；若为 IPv6 可设置为 AF_INET6 和 16；若不设置则默认为 IPv4 地址类型。

表 10.6 列出了 getaddrinfo 函数的语法要点。

表 10.6　　getaddrinfo 函数语法要点

所需头文件	#include <netdb.h>
函数原型	Int getaddrinfo(const char *hostname,const char *service,const struct addrinfo *hints,struct addrinfo **result)
函数传入值	Hostname：主机名
	service：服务名或十进制的串口号字符串
	hints：服务线索
	result：返回结果
函数返回值	成功：0
	出错：−1

在调用之前，首先要对 hints 服务线索进行设置。它是一个 addrinfo 结构体，表 10.7 列举了该结构体常见的选项值。

表 10.7　　addrinfo 结构体常见选项值

结构体头文件	#include <netdb.h>	
ai_flags	AI_PASSIVE：该套接口是用作被动地打开	
	AI_CANONNAME：通知 getaddrinfo 函数返回主机的名字	
family	AF_INET：IPv4 协议	
	AF_INET6：IPv6 协议	
	AF_UNSPE：IPv4 或 IPv6 均可	
ai_socktype	SOCK_STREAM：字节流套接字 socket（TCP）	
	SOCK_DGRAM：数据报套接字 socket（UDP）	
ai_protocol	IPPROTO_IP：IP 协议	
	IPPROTO_IPV4：IPv4 协议	4
	IPPROTO_IPV6：IPv6 协议	
	IPPROTO_UDP：UDP	
	IPPROTO_TCP：TCP	

（1）通常服务器端在调用 getaddrinfo 之前，ai_flags 设置 AI_PASSIVE，用于 bind 函数（用于端口和地址的绑定后面会讲到），主机名 nodename 通常会设置为 NULL。

（2）客户端调用 getaddrinfo 时，ai_flags 一般不设置 AI_PASSIVE，但是主机名 nodename 和服务名 servname（端口）则应该不为空。

注意

（3）即使不设置 ai_flags 为 AI_PASSIVE，取出的地址也并非不可以被 bind，很多程序中 ai_flags 直接设置为 0，即 3 个标志位都不设置，这种情况下只要 hostname 和 servname 设置的没有问题就可以正确 bind。

（3）使用实例

下面的实例给出了 getaddrinfo 函数用法的示例，在后面小节中会给出 gethostbyname 函数用法的例子。

```
/*getaddrinfo.c*/
#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>
int main()
{
      struct addrinfo hints,*res=NULL;
      int rc;
      memset(&hints,0,sizeof(hints));
/*设置 addrinfo 结构体中各参数*/
      hints.ai_family=PF_UNSPEC;
      hints.ai_socktype=SOCK_DGRAM;
      hints.ai_protocol=IPPROTO_UDP;
/*调用 getaddinfo 函数*/
      rc=getaddrinfo("127.0.0.1","123",&hints,&res);
      if (rc != 0) {
            perror("getaddrinfo");
            exit(1);
      }
      else
            printf("getaddrinfo success\n");
}
```

运行结果如下所示：

```
[root@(none) tmp]# getaddrinfo success
```

10.2.3 socket 基础编程

（1）函数说明

进行 socket 编程的基本函数有 socket、bind、listen、accept、send、sendto、recv、recvfrom 这几个，其中对于客户端和服务器端以及 TCP 和 UDP 的操作流程都有所区别，这里先对每个函数进行一定的说明，再给出不同情况下使用的流程图。

- socket：该函数用于建立一个 socket 连接，可指定 socket 类型等信息。在建立了 socket 连接之后，可对 socketadd 或 sockaddr_in 进行初始化，以保存所建立的 socket 信息。
- bind：该函数是用于将本地 IP 地址绑定端口号的，若绑定其他地址则不能成功。另外，它主要用于 TCP 的连接，而在 UDP 的连接中则无必要。
- connect：该函数在 TCP 中是用于 bind 的之后的 client 端，用于与服务器端建立连接，而在 UDP 中由于没有了 bind 函数，因此用 connect 有点类似 bind 函数的作用。
- send 和 recv：这两个函数用于接收和发送数据，可以用在 TCP 中，也可以用在 UDP 中。当用在 UDP 时，可以在 connect 函数建立连接之后再用。
- sendto 和 recvfrom：这两个函数的作用与 send 和 recv 函数类型，也可以用在 TCP 和 UDP 中。当用在 TCP 时，后面的几个与地址有关参数不起作用，函数作用等同于 send 和 recv；当用在 UDP 时，可以用在之前没有使用 connect 的情况时，这两个函数可以自动寻找制定地址并进行连接。

服务器端和客户端使用 TCP 协议的流程图如图 10.6 所示。

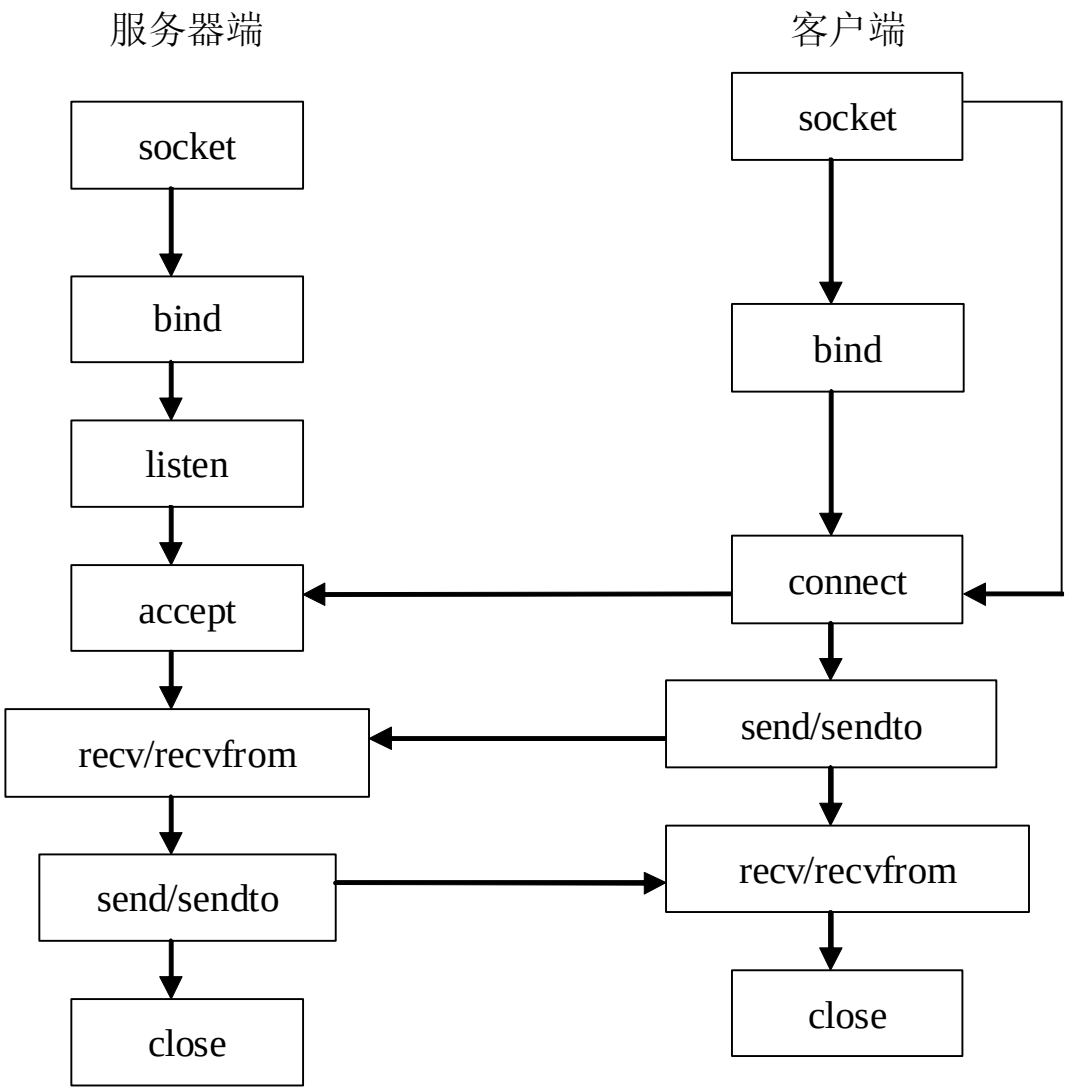

图 10.6　使用 TCP 协议 socket 编程流程图

服务器端和客户端使用 UDP 协议的流程图如图 10.7 所示。

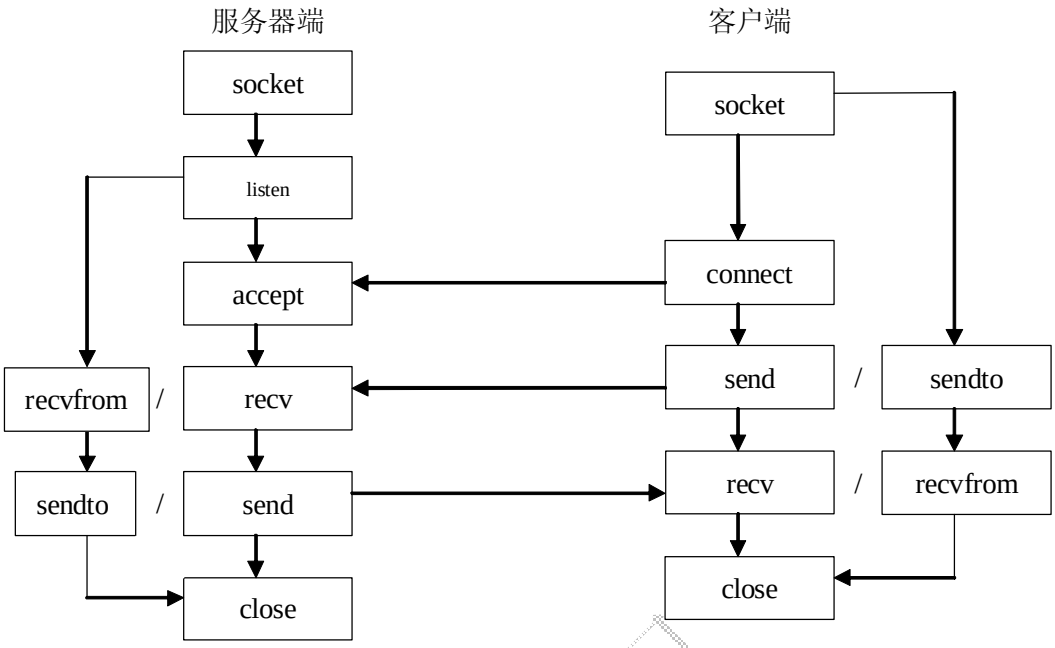

图 10.7　使用 UDP 协议 socket 编程流程图

（2）函数格式

表 10.8 列出了 socket 函数的语法要点。

表 10.8　　socket 函数语法要点

所需头文件	#include <sys/socket.h>	
函数原型	int socket(int family, int type, int protocol)	
函数传入值	family： 协议族	AF_INET：IPv4 协议
		AF_INET6：IPv6 协议
		AF_LOCAL：UNIX 域协议
		AF_ROUTE：路由套接字（socket）
		AF_KEY：密钥套接字（socket）
	type： 套接字类型	SOCK_STREAM：字节流套接字 socket
		SOCK_DGRAM：数据报套接字 socket
		SOCK_RAW：原始套接字 socket
	protoco：0（原始套接字除外）	
函数返回值	成功：非负套接字描述符	
	出错：−1	

表 10.9 列出了 bind 函数的语法要点。

表 10.9　　bind 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int bind(int sockfd,struct sockaddr *my_addr, int addrlen)
函数传入值	socktd：套接字描述符
	my_addr：本地地址
	addrlen：地址长度
函数返回值	成功：0
	出错：−1

端口号和地址在 my_addr 中给出了，若不指定地址，则内核随意分配一个临时端口给该应用程序。

表 10.10 列出了 listen 函数的语法要点。

表 10.10 listen 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int listen(int sockfd， int backlog)
函数传入值	socktd：套接字描述符
	Backlog：请求队列中允许的最大请求数，大多数系统缺省值为 20
函数返回值	成功：0
	出错：−1

表 10.11 列出了 accept 函数的语法要点。

表 10.11 accept 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
函数传入值	socktd：套接字描述符
	addr：客户端地址
	addrlen：地址长度
函数返回值	成功：0
	出错：−1

表 10.12 列出了 connect 函数的语法要点。

表 10.12 connect 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
函数传入值	socktd：套接字描述符
	serv_addr：服务器端地址
	addrlen：地址长度
函数返回值	成功：0

	出错：−1

表 10.13 列出了 send 函数的语法要点。

表 10.13　　send 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int send(int sockfd, const void *msg, int len, int flags)
函数传入值	socktd：套接字描述符
	msg：指向要发送数据的指针
	len：数据长度
	flags：一般为 0
函数返回值	成功：发送的字节数
	出错：−1

表 10.14 列出了 recv 函数的语法要点。

表 10.14　　recv 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recv(int sockfd,void *buf,int len,unsigned int flags)

续表

函数传入值	socktd：套接字描述符
	buf：存放接收数据的缓冲区
	len：数据长度
	flags：一般为 0
函数返回值	成功：接收的字节数
	出错：−1

表 10.15 列出了 sendto 函数的语法要点。

表 10.15　　sendto 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int sendto(int sockfd, const void *msg,int len,unsigned int flags,const struct sockaddr *to, int tolen)
函数传入值	socktd：套接字描述符
	msg：指向要发送数据的指针
	len：数据长度
	flags：一般为 0
	to：目地机的 IP 地址和端口号信息

	tolen：地址长度
函数返回值	成功：发送的字节数
	出错：–1

表 10.16 列出了 recvfrom 函数的语法要点。

表 10.16 recvfrom 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct sockaddr *from,int *fromlen)
函数传入值	socktd：套接字描述符
	buf：存放接收数据的缓冲区
	len：数据长度
	flags：一般为 0
	from：源机的 IP 地址和端口号信息
	tolen：地址长度
函数返回值	成功：接收的字节数
	出错：–1

（3）使用实例

该实例分为客户端和服务器端，其中服务器端首先建立起 socket，然后调用本地端口的绑定，接着就开始与客户端建立联系，并接收客户端发送的消息。客户端则在建立 socket 之后调用 connect 函数来建立连接。

源代码如下所示：

```
/*server.c*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#define SERVPORT 3333
#define BACKLOG 10
#define MAX_CONNECTED_NO 10
#define MAXDATASIZE 5
```

```
    int main()
    {
          struct sockaddr_in server_sockaddr,client_sockaddr;
          int sin_size,recvbytes;
          int sockfd,client_fd;
          char buf[MAXDATASIZE];
    /*建立 socket 连接*/
          if((sockfd = socket(AF_INET,SOCK_STREAM,0))== -1){
                perror("socket");
                exit(1);
          }
          printf("socket success!,sockfd=%d\n",sockfd);
    /*设置 sockaddr_in 结构体中相关参数*/
          server_sockaddr.sin_family=AF_INET;
          server_sockaddr.sin_port=htons(SERVPORT);
          server_sockaddr.sin_addr.s_addr=INADDR_ANY;
          bzero(&(server_sockaddr.sin_zero),8);
    /*绑定函数 bind*/
          if(bind(sockfd,(struct sockaddr *)&server_sockaddr,sizeof(struct
sockaddr))== -1){
                perror("bind");
                exit(1);
          }
          printf("bind success!\n");
    /*调用 listen 函数*/
          if(listen(sockfd,BACKLOG)== -1){
                perror("listen");
                exit(1);
          }
          printf("listening....\n");
    /*调用 accept 函数，等待客户端的连接*/
          if((client_fd=accept(sockfd,(struct sockaddr *)&client_sockaddr,&sin_
size))== -1){
                perror("accept");
                exit(1);
          }
    /*调用 recv 函数接收客户端的请求*/
          if((recvbytes=recv(client_fd,buf,MAXDATASIZE,0))== -1){
                perror("recv");
```

```
            exit(1);
        }
        printf("received a connection :%s\n",buf);
        close(sockfd);
}

/*client.c*/
#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>
#define SERVPORT 3333
#define MAXDATASIZE 100
main(int argc,char *argv[]){
        int sockfd,sendbytes;
        char buf[MAXDATASIZE];
        struct hostent *host;
        struct sockaddr_in serv_addr;
        if(argc < 2){
            fprintf(stderr,"Please enter the server's hostname!\n");
            exit(1);
      }
/*地址解析函数*/
        if((host=gethostbyname(argv[1]))==NULL){
            perror("gethostbyname");
            exit(1);
        }
/*创建 socket*/
        if((sockfd=socket(AF_INET,SOCK_STREAM,0))== -1){
            perror("socket");
            exit(1);
        }
/*设置 sockaddr_in 结构体中相关参数*/
        serv_addr.sin_family=AF_INET;
        serv_addr.sin_port=htons(SERVPORT);
```

```
        serv_addr.sin_addr=*((struct in_addr *)host->h_addr);
        bzero(&(serv_addr.sin_zero),8);
    /*调用 connect 函数主动发起对服务器端的连接*/
        if(connect(sockfd,(struct sockaddr *)&serv_addr,\
              sizeof(struct sockaddr))== -1){
              perror("connect");
              exit(1);
        }
    /*发送消息给服务器端*/
        if((sendbytes=send(sockfd,"hello",5,0))== -1){
              perror("send");
              exit(1);
        }
    close(sockfd);
    }
```

在运行时需要先启动服务器端，再启动客户端。这里可以把服务器端下载到开发板上，客户端在宿主机上运行，然后配置双方的 IP 地址，确保在双方可以通信（如使用 ping 命令验证）的情况下运行该程序即可。

```
[root@(none) tmp]# ./server
socket success!,sockfd=3
bind success!
listening....
received a connection :hello
[root@www yul]# ./client 59.64.128.1
```

10.3 网络高级编程

在实际情况中，人们往往遇到多个客户端连接服务器端的情况。由于之前介绍的如 connet、recv、send 都是阻塞性函数，若资源没有准备好，则调用该函数的进程将进入睡眠状态，这样就无法处理 I/O 多路复用的情况了。本节给出了两种解决 I/O 多路复用的解决方法，这两个函数都是之前学过的 fcntl 和 select（请读者先复习第 6 章中的相关内容）。可以看到，由于在 Linux 中把 socket 也作为一种特殊文件描述符，这给用户的处理带来了很大的方便。

1．fcntl

函数 fcntl 针对 socket 编程提供了如下的编程特性。

- 非阻塞 I/O：可将 cmd 设置为 F_SETFL，将 lock 设置为 O_NONBLOCK。
- 信号驱动 I/O：可将 cmd 设置为 F_SETFL，将 lock 设置为 O_ASYNC。

下面是用 fcntl 设置为非阻塞 I/O 的使用实例：

```
/*fcntl.c*/
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <fcntl.h>
#define SERVPORT 3333
#define BACKLOG 10
#define MAX_CONNECTED_NO 10
#define MAXDATASIZE 100
int main()
{
      struct sockaddr_in server_sockaddr,client_sockaddr;
      int sin_size,recvbytes,flags;
      int sockfd,client_fd;
      char buf[MAXDATASIZE];
      if((sockfd = socket(AF_INET,SOCK_STREAM,0))== -1){
            perror("socket");
            exit(1);
      }
      printf("socket success!,sockfd=%d\n",sockfd);
      server_sockaddr.sin_family=AF_INET;
      server_sockaddr.sin_port=htons(SERVPORT);
      server_sockaddr.sin_addr.s_addr=INADDR_ANY;
      bzero(&(server_sockaddr.sin_zero),8);
      if(bind(sockfd,(struct sockaddr *)&server_sockaddr,sizeof(struct
sockaddr))== -1){
            perror("bind");
            exit(1);
```

```
        }
        printf("bind success!\n");
        if(listen(sockfd,BACKLOG)== -1){
              perror("listen");
              exit(1);
        }
        printf("listening....\n");
    /*调用 fcntl 函数设置非阻塞参数*/
        if((flags=fcntl( sockfd, F_SETFL, 0))<0)
                    perror("fcntl F_SETFL");
              flag |= O_NONBLOCK;
              if(fcntl(fd,F_SETEL,flags)<0)
                    perror("fcntl");
        while(1){
              sin_size=sizeof(struct sockaddr_in);
              if((client_fd=accept(sockfd,(structsockaddr*)&client_sockaddr,
&sin_size))== -1){
                    perror("accept");
                    exit(1);
              }
              if((recvbytes=recv(client_fd,buf,MAXDATASIZE,0))== -1){
                    perror("recv");
                    exit(1);
              }
              if(read(client_fd,buf,MAXDATASIZE)<0){
                    perror("read");
                    exit(1);
              }
              printf("received a connection :%s",buf);
        close(client_fd);
        exit(1);
        }/*while*/
    }
```

运行该程序，结果如下所示：

```
    [root@(none) tmp]]# ./fcntl
    socket success!,sockfd=3
    bind success!
    listening....
```

```
accept: Resource temporarily unavailable
```

可以看到，当 accept 的资源不可用时，程序就会自动返回。

2．select

使用 fcntl 函数虽然可以实现非阻塞 I/O 或信号驱动 I/O，但在实际使用时往往会对资源是否准备完毕进行循环测试，这样就大大增加了不必要的 CPU 资源。在这里可以使用 select 函数来解决这个问题，同时，使用 select 函数还可以设置等待的时间，可以说功能更加强大。下面是使用 select 函数的服务器端源代码：

```
/*select_socket.c*/
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#define SERVPORT 3333
#define BACKLOG 10
#define MAX_CONNECTED_NO 10
#define MAXDATASIZE 100
int main()
{
      struct sockaddr_in server_sockaddr,client_sockaddr;
      int sin_size,recvbytes;
      fd_set readfd;
      fd_set writefd;
      int sockfd,client_fd;
      char buf[MAXDATASIZE];
      if((sockfd = socket(AF_INET,SOCK_STREAM,0))== -1){
            perror("socket");
            exit(1);
      }
      printf("socket success!,sockfd=%d\n",sockfd);
```

```
        server_sockaddr.sin_family=AF_INET;
        server_sockaddr.sin_port=htons(SERVPORT);
        server_sockaddr.sin_addr.s_addr=INADDR_ANY;
        bzero(&(server_sockaddr.sin_zero),8);
        if(bind(sockfd,(struct   sockaddr   *)&server_sockaddr,sizeof(struct
sockaddr))== -1){
              perror("bind");
              exit(1);
        }
        printf("bind success!\n");
        if(listen(sockfd,BACKLOG)== -1){
              perror("listen");
              exit(1);
        }
        printf("listening....\n");
   /*将调用 socket 函数的描述符作为文件描述符*/
        FD_ZERO(&readfd);
        FD_SET(sockfd,&readfd);
        while(1){
        sin_size=sizeof(struct sockaddr_in);
   /*调用 select 函数*/
        if(select(MAX_CONNECTED_NO,&readfd,NULL,NULL,(struct timeval *)0)>0){
              if(FD_ISSET(sockfd,&readfd)>0){
                    if((client_fd=accept(sockfd,(struct sockaddr *)&client_
sockaddr,&sin_size))== -1){
                          perror("accept");
                          exit(1);
                    }
                    if((recvbytes=recv(client_fd,buf,MAXDATASIZE,0))== -1){
                          perror("recv");
                          exit(1);
                    }
                    if(read(client_fd,buf,MAXDATASIZE)<0){
                          perror("read");
                          exit(1);
                    }
                    printf("received a connection :%s",buf);
              }/*if*/
              close(client_fd);
```

```
            }/*select*/
        }/*while*/
    }
```

运行该程序时，可以先启动服务器端，再反复运行客户端程序即可，服务器端运行结果如下所示：

```
[root@(none) tmp]# ./server2
socket success!,sockfd=3
bind success!
listening....
received a connection :hello
received a connection :hello
```

10.4 ping 源码分析

10.4.1 ping 简介

Ping 是网络中应用非常广泛的一个软件，它是基于 ICMP 协议的。下面首先对 ICMP 协议做一简单介绍。

ICMP 是 IP 层的一个协议，它是用来探测主机、路由维护、路由选择和流量控制的。ICMP 报文的最终报宿不是报宿计算机上的一个用户进程，而是那个计算机上的 IP 层软件。也就是说，当一个带有错误信息的 ICMP 报文到达时，IP 软件模块就处理本身问题，而不把这个 ICMP 报文传送给应用程序。

ICMP 报文类型有：回送(ECHO)回答（0）；报宿不可到达（3）；报源断开（4）；重定向（改变路由）（5）；回送（ECHO）请求（8）；数据报超时（11）；数据报参数问题（12）；时间印迹请求（13）；时间印迹回答（14）；信息请求（15）；信息回答（16）；地址掩码请求（17）；地址掩码回答（18）。

虽然每种报文都有不同的格式，但它们开始都有下面三段：

- 一个 8 位整数报文 TYPE（类型）段；
- 一个 8 位 CODE（代码码）段，提供更多的报文类型信息；
- 一个 16 位 CHECKSUM（校验和）段；

此外，报告差错的 ICMP 报文还包含产生问题数据报的网际报头及前 64 位数据。一个 ICMP 回送请求与回送回答报文的格式如表 10.17 所示。

表 10.17 ICMP 回送请求与回送回答报文格式

类型	CODE	校验和[CHECKSUM]
标识符		序列号

数据

10.4.2 ping 源码分析

下面的 ping.c 源码是在 busybox 里实现的源码。在这个完整的 ping.c 代码中有较多选项的部分代码，因此，这里先分析除去选项部分代码的函数实现部分流程，接下来再给出完整的 ping 代码分析。这样，读者就可以看到一个完整协议实现应该考虑到的各个部分 。

1．Ping 代码主体流程

Ping.c 主体流程图如下图 10.8 所示。另外，由于 ping 是 IP 层的协议，因此在建立 socket 时需要使用 SOCK_RAW 选项。在循环等待回应信息处，用户可以指定“-f”洪泛选项，这时就会使用 select 函数来指定在一定的时间内进行回应。

2．主要选项说明

Ping 函数主要有以下几个选项：

- d：调试选项（F_SO_DEBUG）
- f：洪泛选项（F_FLOOD）
- i：等待选项（F_INTERVAL）
- r：路由选项（F_RROUTE）
- l：广播选项（MULTICAST_NOLOOP）

对于这些选项，尤其是路由选项、广播选项和洪泛选项都会有不同的实现代码。

另外，ping 函数可以接受用户使用的 SIGINT 和 SIGALARM 信号来结束程序，它们分别指向了不同的结束代码，请读者阅读下面相关代码。

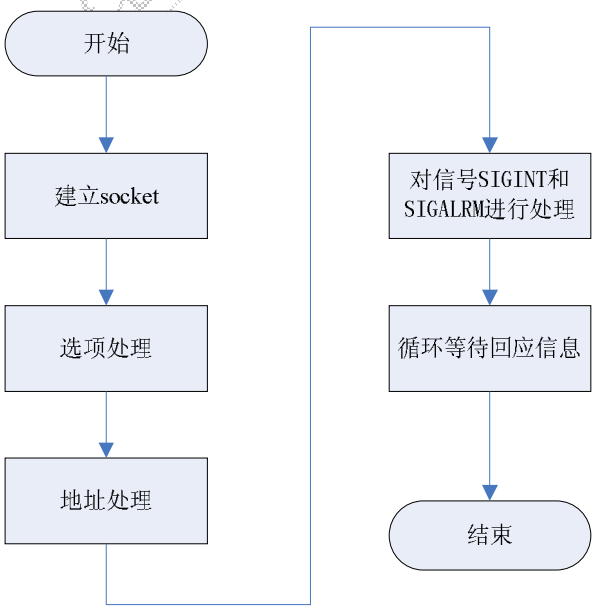

图 10.8 ping 主体流程图

3．源代码及注释

（1）主体代码

ping 代码的主体部分可以四部分，首先是一些头函数及宏定义：

```
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/time.h>
#include <sys/signal.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <getopt.h>
#include <resolv.h>
#define F_FLOOD 0x001
#define F_INTERVAL  0x002
#define F_NUMERIC   0x004
#define F_PINGFILLED    0x008
#define F_QUIET 0x010
#define F_RROUTE    0x020
#define F_SO_DEBUG  0x040
#define F_SO_DONTROUTE  0x080
#define F_VERBOSE   0x100

/* 多播选项 */
int moptions;
#define MULTICAST_NOLOOP    0x001
#define MULTICAST_TTL       0x002
#define MULTICAST_IF        0x004
…
```

接下来的第 2 部分是建立 socket 并处理选项：

```
Int main(int argc, char *argv[])
{
      struct timeval timeout;
      struct hostent *hp;
      struct sockaddr_in *to;
      struct protoent *proto;
      struct in_addr ifaddr;
      int i;
      int ch, fdmask, hold, packlen, preload;
      u_char *datap, *packet;
      char *target, hnamebuf[MAXHOSTNAMELEN];
      u_char ttl, loop;
      int am_i_root;
…
      static char *null = NULL;

      /*__environ = &null;*/
      am_i_root = (getuid()==0);

      /*
      *建立 socket 连接，并且测试是否是 root 用户
      */
      if ((s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
            if (errno==EPERM) {
                  fprintf(stderr, "ping: ping must run as root\n");
            }
            else perror("ping: socket");
            exit(2);
      }
…
      preload = 0;
      datap = &outpack[8 + sizeof(struct timeval)];
      while ((ch = getopt(argc, argv, "I:LRc:dfh:i:l:np:qrs:t:v")) != EOF)
            switch(ch) {
            case 'c':
                  npackets = atoi(optarg);
                  if (npackets <= 0) {
```

```
                        (void)fprintf(stderr,
                            "ping: bad number of packets to transmit.\n");
                        exit(2);
                    }
                    break;
    /*调用选项*/
                case 'd':
                    options |= F_SO_DEBUG;
                    break;
    /*flood 选项*/
                case 'f':
                    if (!am_i_root) {
                        (void)fprintf(stderr,
                            "ping: %s\n", strerror(EPERM));
                        exit(2);
                    }
                    options |= F_FLOOD;
                    setbuf(stdout, NULL);
                    break;
    /*等待选项*/
                case 'i':       /* wait between sending packets */
                    interval = atoi(optarg);
                    if (interval <= 0) {
                        (void)fprintf(stderr,
                            "ping: bad timing interval.\n");
                    exit(2);
                }
                options |= F_INTERVAL;
                break;
            case 'l':
                if (!am_i_root) {
                    (void)fprintf(stderr,
                            "ping: %s\n", strerror(EPERM));
                        exit(2);
                    }
                    preload = atoi(optarg);
                    if (preload < 0) {
                        (void)fprintf(stderr,
                            "ping: bad preload value.\n");
```

```
                    exit(2);
                }
                break;
    …
            default:
                usage();
            }
        argc -= optind;
        argv += optind;

        if (argc != 1)
            usage();
        target = *argv;
```

接下来的第 3 部分是用于获取地址，这里主要使用了 inet_aton 函数，将点分十进制地址转化为二进制地址。当然，作为完整的 ping 程序有较完善的出错处理：

```
        memset(&whereto, 0, sizeof(struct sockaddr));
        to = (struct sockaddr_in *)&whereto;
        to->sin_family = AF_INET;
    /*地址转换函数*/
        if (inet_aton(target, &to->sin_addr)) {
            hostname = target;
        }
        else {
    #if 0
            char * addr = resolve_name(target, 0);
            if (!addr) {
                (void)fprintf(stderr,
                    "ping: unknown host %s\n", target);
                exit(2);
          }
            to->sin_addr.s_addr = inet_addr(addr);
            hostname = target;
    #else
    /*调用 gethostbyname 识别主机名*/
            hp = gethostbyname(target);
            if (!hp) {
                (void)fprintf(stderr,
                    "ping: unknown host %s\n", target);
```

```
                exit(2);
            }
            to->sin_family = hp->h_addrtype;
            if (hp->h_length > (int)sizeof(to->sin_addr)) {
                hp->h_length = sizeof(to->sin_addr);
            }
            memcpy(&to->sin_addr, hp->h_addr, hp->h_length);
             (void)strncpy(hnamebuf, hp->h_name, sizeof(hnamebuf) - 1);
            hostname = hnamebuf;
#endif
        }
```

接下来的一部分主要是对各个选项（如路由、多播）的处理，这里就不做介绍了。再接下来是 ping 函数的最主要部分，就是接收无限循环回应信息，这里主要用到了函数 recvfrom。另外，对用户中断信息也有相应的处理，如下所示：

```
        if (to->sin_family == AF_INET)
            (void)printf("PING %s (%s): %d data bytes\n", hostname,
                inet_ntoa(*(struct in_addr *)&to->sin_addr.s_addr),
                datalen);
        else
            (void)printf("PING %s: %d data bytes\n", hostname, datalen);
/*若程序接收到 SIGINT 或 SIGALRM 信号，调用相关的函数*/
        (void)signal(SIGINT, finish);
        (void)signal(SIGALRM, catcher);
…
/*循环等待客户端的回应信息*/
        for (;;) {
            struct sockaddr_in from;
            register int cc;
            int fromlen;

            if (options & F_FLOOD) {
/*形成 ICMP 回应数据包，在后面会有讲解*/
                pinger();
/*设定等待实践*/
                timeout.tv_sec = 0;
                timeout.tv_usec = 10000;
                fdmask = 1 << s;
/*调用 select 函数*/
```

```
                if (select(s + 1, (fd_set *)&fdmask, (fd_set *)NULL,
                    (fd_set *)NULL, &timeout) < 1)
                    continue;
            }
            fromlen = sizeof(from);
/*接收客户端信息*/
            if ((cc = recvfrom(s, (char *)packet, packlen, 0,
                (struct sockaddr *)&from, &fromlen)) < 0) {
                if (errno == EINTR)
                    continue;
                perror("ping: recvfrom");
                continue;
            }
            pr_pack((char *)packet, cc, &from);
            if (npackets && nreceived >= npackets)
                break;
        }
        finish(0);
        /* NOTREACHED */
        return 0;
}
```

（2）其他函数

下面的函数也是 ping 程序中用到的重要函数。首先 catcher 函数是用户在发送 SIGINT 时调用的函数，在该函数中又调用了 SIGALARM 信号的处理来结束程序。

```
static void
catcher(int ignore)
{
        int waittime;

        (void)ignore;
        pinger();
/*调用 catcher 函数*/
        (void)signal(SIGALRM, catcher);
        if (!npackets || ntransmitted < npackets)
            alarm((u_int)interval);
        else {
            if (nreceived) {
                waittime = 2 * tmax / 1000;
```

```
                    if (!waittime)
                          waittime = 1;
                    if (waittime > MAXWAIT)
                          waittime = MAXWAIT;
            }   else
                    waittime = MAXWAIT;
    /*调用 finish 函数，并设定一定的等待实践*/
                (void)signal(SIGALRM, finish);
                (void)alarm((u_int)waittime);
          }
    }
```

Pinger 函数也是一个非常重要的函数，用于形成 ICMP 回应数据包，其中 ID 是该进程的 ID，数据段中的前 8 字节用于存放时间间隔，从而可以计算 ping 程序从对端返回的往返时延差，这里的数据校验用到了后面定义的 in_cksum 函数。其代码如下所示：

```
    static void
    pinger(void)
    {
          register struct icmphdr *icp;
          register int cc;
          int i;

    /*形成 icmp 信息包，填写 icmphdr 结构体中的各项数据*/
          icp = (struct icmphdr *)outpack;
          icp->icmp_type = ICMP_ECHO;
          icp->icmp_code = 0;
          icp->icmp_cksum = 0;
          icp->icmp_seq = ntransmitted++;
          icp->icmp_id = ident;         /* ID */

          CLR(icp->icmp_seq % mx_dup_ck);

    /*设定等待实践*/
          if (timing)
                (void)gettimeofday((struct timeval *)&outpack[8],
                   (struct timezone *)NULL);

          cc = datalen + 8;         /* skips ICMP portion */
```

```
        /* compute ICMP checksum here */
        icp->icmp_cksum = in_cksum((u_short *)icp, cc);

        i = sendto(s, (char *)outpack, cc, 0, &whereto,
             sizeof(struct sockaddr));

        if (i < 0 || i != cc)  {
             if (i < 0)
                  perror("ping: sendto");
             (void)printf("ping: wrote %s %d chars, ret=%d\n",
                  hostname, cc, i);
        }
        if (!(options & F_QUIET) && options & F_FLOOD)
             (void)write(STDOUT_FILENO, &DOT, 1);
   }
```

pr_pack 是数据包显示函数，分别打印出 IP 数据包部分和 ICMP 回应信息。在规范的程序中通常将数据的显示部分独立出来，这样就可以很好地加强程序的逻辑性和结构性。

```
   void
   pr_pack(char *buf, int cc, struct sockaddr_in *from)
   {
        register struct icmphdr *icp;
        register int i;
        register u_char *cp,*dp;
   /*#if 0*/
        register u_long l;
        register int j;
        static int old_rrlen;
        static char old_rr[MAX_IPOPTLEN];
   /*#endif*/
        struct iphdr *ip;
        struct timeval tv, *tp;
        long triptime = 0;
        int hlen, dupflag;

        (void)gettimeofday(&tv, (struct timezone *)NULL);

        /* 检查 IP 数据包头信息 */
        ip = (struct iphdr *)buf;
```

```
        hlen = ip->ip_hl << 2;
        if (cc < datalen + ICMP_MINLEN) {
              if (options & F_VERBOSE)
                    (void)fprintf(stderr,
                      "ping: packet too short (%d bytes) from %s\n", cc,
                      inet_ntoa(*(struct in_addr *)&from->sin_addr.s_addr));
              return;
        }

        /* ICMP 部分显示 */
        cc -= hlen;
        icp = (struct icmphdr *)(buf + hlen);
        if (icp->icmp_type == ICMP_ECHOREPLY) {
              if (icp->icmp_id != ident)
                    return;           /* 'Twas not our ECHO */
              ++nreceived;
              if (timing) {
#ifndef icmp_data
                    tp = (struct timeval *)(icp + 1);
#else
                    tp = (struct timeval *)icp->icmp_data;
#endif
                    tvsub(&tv, tp);
                    triptime = tv.tv_sec * 10000 + (tv.tv_usec / 100);
                    tsum += triptime;
                    if (triptime < tmin)
                          tmin = triptime;
                    if (triptime > tmax)
                          tmax = triptime;
              }

              if (TST(icp->icmp_seq % mx_dup_ck)) {
                    ++nrepeats;
                    --nreceived;
                    dupflag = 1;
              } else {
                    SET(icp->icmp_seq % mx_dup_ck);
                    dupflag = 0;
              }
```

```
            if (options & F_QUIET)
                  return;

            if (options & F_FLOOD)
                  (void)write(STDOUT_FILENO, &BSPACE, 1);
            else {
                  (void)printf("%d bytes from %s: icmp_seq=%u", cc,
                    inet_ntoa(*(struct in_addr *)&from->sin_addr.s_addr),
                    icp->icmp_seq);
                  (void)printf(" ttl=%d", ip->ip_ttl);
                  if (timing)
                        (void)printf(" time=%ld.%ld ms", triptime/10,
                                    triptime%10);
                  if (dupflag)
                        (void)printf(" (DUP!)");
                  /* check the data */
#ifndef icmp_data
                  cp = ((u_char*)(icp + 1) + 8);
#else
                  cp = (u_char*)icp->icmp_data + 8;
#endif
                  dp = &outpack[8 + sizeof(struct timeval)];
                  for (i = 8; i < datalen; ++i, ++cp, ++dp) {
                        if (*cp != *dp) {
      (void)printf("\nwrong data byte #%d should be 0x%x but was 0x%x",
         i, *dp, *cp);
                              cp = (u_char*)(icp + 1);
                              for (i = 8; i < datalen; ++i, ++cp) {
                                    if ((i % 32) == 8)
                                          (void)printf("\n\t");
                                    (void)printf("%x ", *cp);
                              }
                              break;
                        }
                  }
            }
      } else {
            /* We've got something other than an ECHOREPLY */
```

```
            if (!(options & F_VERBOSE))
                  return;
            (void)printf("%d bytes from %s: ", cc,
            pr_addr(from->sin_addr.s_addr));
            pr_icmph(icp);
      }

/*#if 0*/
      /*显示其他 IP 选项 */
      cp = (u_char *)buf + sizeof(struct iphdr);

      for (; hlen > (int)sizeof(struct iphdr); --hlen, ++cp)
            switch (*cp) {
            case IPOPT_EOL:
                  hlen = 0;
                  break;
            case IPOPT_LSRR:
                  (void)printf("\nLSRR: ");
                  hlen -= 2;
                  j = *++cp;
                  ++cp;
                  if (j > IPOPT_MINOFF)
                        for (;;) {
                              l = *++cp;
                              l = (l<<8) + *++cp;
                              l = (l<<8) + *++cp;
                              l = (l<<8) + *++cp;
                              if (l == 0)
                                    (void)printf("\t0.0.0.0");
                        else
                              (void)printf("\t%s", pr_addr(ntohl(l)));
                        hlen -= 4;
                        j -= 4;
                        if (j <= IPOPT_MINOFF)
                              break;
                        (void)putchar('\n');
                  }
                  break;
            case IPOPT_RR:
```

```
                j = *++cp;     /* get length */
                i = *++cp;     /* and pointer */
                hlen -= 2;
                if (i > j)
                     i = j;
                i -= IPOPT_MINOFF;
                if (i <= 0)
                     continue;
                if (i == old_rrlen
                   && cp == (u_char *)buf + sizeof(struct iphdr) + 2
                   && !memcmp((char *)cp, old_rr, i)
                   && !(options & F_FLOOD)) {
                     (void)printf("\t(same route)");
                     i = ((i + 3) / 4) * 4;
                     hlen -= i;
                     cp += i;
                     break;
                }
                old_rrlen = i;
                memcpy(old_rr, cp, i);
                (void)printf("\nRR: ");
                for (;;) {
                     l = *++cp;
                     l = (l<<8) + *++cp;
                     l = (l<<8) + *++cp;
                     l = (l<<8) + *++cp;
                     if (l == 0)
                          (void)printf("\t0.0.0.0");
                     else
                          (void)printf("\t%s", pr_addr(ntohl(l)));
                     hlen -= 4;
                     i -= 4;
                     if (i <= 0)
                          break;
                     (void)putchar('\n');
                }
                break;
           case IPOPT_NOP:
                (void)printf("\nNOP");
```

```
                    break;
              default:
                    (void)printf("\nunknown option %x", *cp);
                    break;
              }
/*#endif*/
       if (!(options & F_FLOOD)) {
              (void)putchar('\n');
              (void)fflush(stdout);
       }
}
```

in_cksum 是数据校验程序，如下所示：

```
static int
in_cksum(u_short *addr, int len)
{
       register int nleft = len;
       register u_short *w = addr;
       register int sum = 0;
       u_short answer = 0;

       /*这里的算法很简单，就采用 32bit 的加法*/
       while (nleft > 1)  {
              sum += *w++;
              nleft -= 2;
       }

       if (nleft == 1) {
              *(u_char *)(&answer) = *(u_char *)w ;
              sum += answer;
       }

       /*把高 16bit 加到低 16bit 上去*/
       sum = (sum >> 16) + (sum & 0xffff);
       sum += (sum >> 16);
       answer = ~sum;
       return(answer);
}
```

Finish 程序是 ping 程序的结束程序，主要是打印出来一些统计信息，如下所示：

```
    static void
    finish(int ignore)
    {
          (void)ignore;
          (void)signal(SIGINT, SIG_IGN);
          (void)putchar('\n');
          (void)fflush(stdout);
          (void)printf("--- %s ping statistics ---\n", hostname);
          (void)printf("%ld packets transmitted, ", ntransmitted);
          (void)printf("%ld packets received, ", nreceived);
          if (nrepeats)
               (void)printf("+%ld duplicates, ", nrepeats);
          if (ntransmitted)
               if (nreceived > ntransmitted)
                    (void)printf("-- somebody's printing up packets!");
               else
                    (void)printf("%d%% packet loss",
                       (int) (((ntransmitted - nreceived) * 100) /
                       ntransmitted));
          (void)putchar('\n');
          if (nreceived && timing)
               (void)printf("round-trip min/avg/max = %ld.%ld/%lu.%ld/%ld.%ld
ms\n",
                    tmin/10, tmin%10,
                    (tsum / (nreceived + nrepeats))/10,
                    (tsum / (nreceived + nrepeats))%10,
                    tmax/10, tmax%10);

          if (nreceived==0) exit(1);
          exit(0);
    }

    #ifdef notdef
    static char *ttab[] = {
          "Echo Reply",     /* ip + seq + udata */
          "Dest Unreachable",   /* net, host, proto, port, frag, sr + IP */
          "Source Quench",  /* IP */
```

```
        "Redirect",       /* redirect 类型, gateway, + IP  */
        "Echo",
        "Time Exceeded",  /*传输超时*/
        "Parameter Problem",  /* IP 参数问题 */
        "Timestamp",      /* id + seq + three timestamps */
        "Timestamp Reply",    /* " */
        "Info Request",       /* id + sq */
        "Info Reply"      /* " */
};
#endif
```

pr_icmph 函数是用于打印 ICMP 的回应信息，如下所示：

```
static void
pr_icmph(struct icmphdr *icp)
{
      switch(icp->icmp_type) {
/*ICMP 回应*/
      case ICMP_ECHOREPLY:
            (void)printf("Echo Reply\n");
            /* XXX ID + Seq + Data */
            break;
/*ICMP 终点不可达*/
      case ICMP_DEST_UNREACH:
            switch(icp->icmp_code) {
            case ICMP_NET_UNREACH:
                  (void)printf("Destination Net Unreachable\n");
                  break;
            case ICMP_HOST_UNREACH:
                  (void)printf("Destination Host Unreachable\n");
                  break;
            case ICMP_PROT_UNREACH:
                  (void)printf("Destination Protocol Unreachable\n");
                  break;
            …
            default:
                  (void)printf("Dest Unreachable, Unknown Code: %d\n",
                     icp->icmp_code);
                  break;
            }
```

```
            /* Print returned IP header information */
#ifndef icmp_data
            pr_retip((struct iphdr *)(icp + 1));
#else
            pr_retip((struct iphdr *)icp->icmp_data);
#endif
            break;
            …
            default:
                  (void)printf("Redirect, Bad Code: %d", icp->icmp_code);
                  break;
            }
            (void)printf("(New addr: %s)\n",
                        inet_ntoa(icp->icmp_gwaddr));
#ifndef icmp_data
            pr_retip((struct iphdr *)(icp + 1));
#else
            pr_retip((struct iphdr *)icp->icmp_data);
#endif
            break;
      case ICMP_ECHO:
            (void)printf("Echo Request\n");
            /* XXX ID + Seq + Data */
            break;
      case ICMP_TIME_EXCEEDED:
            switch(icp->icmp_code) {
            case ICMP_EXC_TTL:
                  (void)printf("Time to live exceeded\n");
                  break;
            case ICMP_EXC_FRAGTIME:
                  (void)printf("Frag reassembly time exceeded\n");
                  break;
            default:
                  (void)printf("Time exceeded, Bad Code: %d\n",
                     icp->icmp_code);
                  break;
            }
      …
      default:
```

```
                (void)printf("Bad ICMP type: %d\n", icp->icmp_type);
        }
    }
```

pr_iph 函数是用于打印 IP 数据包头选项，如下所示：

```
    static void
    pr_iph(struct iphdr *ip)
    {
        int hlen;
        u_char *cp;

        hlen = ip->ip_hl << 2;
        cp = (u_char *)ip + 20;       /* point to options */
        (void)printf("Vr HL TOS  Len  ID Flg  off TTL Pro cks  Src  Dst Data\n");
        (void)printf(" %1x  %1x  %02x %04x %04x",
           ip->ip_v, ip->ip_hl, ip->ip_tos, ip->ip_len, ip->ip_id);
        (void)printf("   %1x %04x", ((ip->ip_off) & 0xe000) >> 13,
           (ip->ip_off) & 0x1fff);
        (void)printf("  %02x  %02x %04x", ip->ip_ttl, ip->ip_p, ip->ip_sum);
        (void)printf(" %s ", inet_ntoa(*((struct in_addr *) &ip->ip_src)));
        (void)printf(" %s ", inet_ntoa(*((struct in_addr *) &ip->ip_dst)));
        /* dump and option bytes */
        while (hlen-- > 20) {
            (void)printf("%02x", *cp++);
        }
        (void)putchar('\n');
    }
```

pr_addr 是用于将 ascii 主机地址转换为十进制点分形式并打印出来，这里使用的函数是 inet_ntoa，如下所示：

```
    static char *
    pr_addr(u_long l)
    {
        struct hostent *hp;
        static char buf[256];

        if ((options & F_NUMERIC) ||
           !(hp = gethostbyaddr((char *)&l, 4, AF_INET)))
             (void)sprintf(buf, /*sizeof(buf),*/ "%s",inet_ntoa(*(struct
```

```
in_addr *)&l));
          else
               (void)sprintf(buf, /*sizeof(buf),*/ "%s (%s)", hp->h_name,inet_
ntoa(*(struct in_addr *)&l));
          return(buf);
     }
```

Usage 函数是用于显示帮助信息，如下所示：

```
     static void
     usage(void)
     {
          (void)fprintf(stderr,
             "usage: ping [-LRdfnqrv] [-c count] [-i wait] [-l preload]\n\t[-p
pattern] [-s packetsize] [-t ttl] [-I interface address] host\n");
          exit(2);
     }
```

10.5 实验内容——NTP 协议实现

1．实验目的

通过实现 NTP 协议的练习，进一步掌握 Linux 下网络编程，并且提高协议的分析与实现能力，为参与完成综合性项目打下良好的基础。

2．实验内容

Network Time Protocol（NTP）协议是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟，GPS 等）做同步化，它可以提供高精确度的时间校正（LAN 上与标准间差小于 1 毫秒，WAN 上几十毫秒），且可用加密确认的方式来防止恶毒的协议攻击。

NTP 提供准确时间，首先要有准确的时间来源，这一时间应该是国际标准时间 UTC。NTP 获得 UTC 的时间来源可以是原子钟、天文台、卫星，也可以从 Internet 上获取。这样就有了准确而可靠的时间源。时间是按 NTP 服务器的等级传播。按照距离外部 UTC 源的远近将所有服务器归入不同的 Stratun（层）中。Stratum-1 在顶层，有外部 UTC 接入，而 Stratum-2 则从 Stratum-1 获取时间，Stratum-3 从 Stratum-2 获取时间，以此类推，但 Stratum 层的总数限制在 15 以内。所有这些服务器在逻辑上形成阶梯式的架构相互连接，而 Stratum-1 的时间服务器是整个系统的基础。

进行网络协议实现时最重要的是了解协议数据格式。NTP 数据包有 48 个字节，其中 NTP 包头 16 字节，时间戳 32 个字节。其协议格式如图 10.9 所示。

2	5	8	16	24	32bit
LI	VN	Mode	Stratum	Poll	Precision
Root Delay					
Root Dispersion					
Reference Identifier					
Reference timestamp (64)					
Originate Timestamp (64)					
Receive Timestamp (64)					
Transmit Timestamp (64)					
Key Identifier (optional) (32)					
Message digest (optional) (128)					

图 10.9　NTP 协议数据格式

其协议字段的含义如下所示。

- LI：跳跃指示器，警告在当月最后一天的最终时刻插入的迫近闰秒（闰秒）。
- VN：版本号。
- Mode：模式。该字段包括以下值：0—预留；1—对称行为；3—客户机；4—服务器；5—广播；6—NTP 控制信息。
- Stratum：对本地时钟级别的整体识别。
- Poll：有符号整数表示连续信息间的最大间隔。
- Precision：有符号整数表示本地时钟精确度。
- Root Delay：有符号固定点序号表示主要参考源的总延迟，很短时间内的位 15 到 16 间的分段点。
- Root Dispersion：无符号固定点序号表示相对于主要参考源的正常差错，很短时间内的位 15 到 16 间的分段点。
- Reference Identifier：识别特殊参考源。
- Originate Timestamp：这是向服务器请求分离客户机的时间，采用 64 位时标格式。
- Receive Timestamp：这是向服务器请求到达客户机的时间，采用 64 位时标格式。
- Transmit Timestamp：这是向客户机答复分离服务器的时间，采用 64 位时标格式。
- Authenticator（Optional）：当实现了 NTP 认证模式时，主要标识符和信息数字域就包括已定义的信息认证代码（MAC）信息。

由于 NTP 协议中涉及到比较多的时间相关的操作，为了简化实现过程，本实验仅要求实现 NTP 协议客户端部分的网络通信模块，也就是构造 NTP 协议字段进行发送和接收，最后与时间相关的操作不需进行处理。

3．实验步骤

（1）画出流程图

简易 NTP 客户端实现流程图如图 10.10 所示。

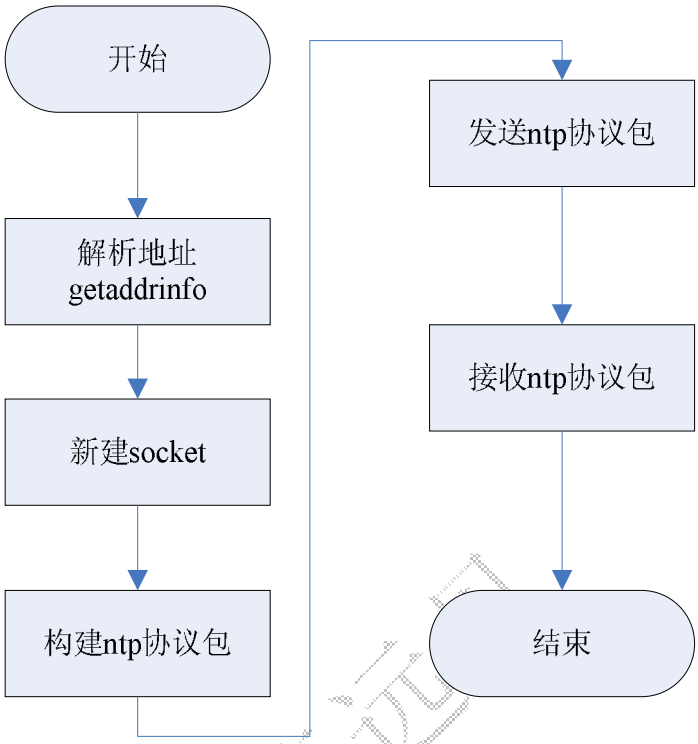

图 10.10　简易 NTP 客户端流程图

（2）编写程序

具体代码如下：

```
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
struct NTPPacket
{
```

```
    char Leap_Ver_Mode;
    /*client=0*/
    char Startum;
    char Poll;
    char Precision;
    double RootDelay;
    double Dispersion;
    char RefIdentifier[4];
    char RefTimeStamp[8];
    char OriTimeStamp[8];
    char RecvTimeStamp[8];
    char TransTimeStamp[8];
    };

    #define NTPPORT       123
    #define TIMEPORT      37
    #define NTPV1   "NTP/V1"
    #define NTPV2   "NTP/V2"
    #define NTPV3   "NTP/V3"
    #define NTPV4   "NTP/V4"
    #define TIME    "TIME/UDP"
    double SecondBef1970;
    struct sockaddr_in sin;
    struct addrinfo       hints, *res=NULL;
    int rc,sk;
    char Protocol[32];
    /*构建 NTP 协议包*/
    int ConstructPacket(char *Packet)
    {
    char Version=1;
    long SecondFrom1900;
    long Zero=0;
    int Port;
    time_t timer;
    strcpy(Protocol,NTPV1);
    /*判断协议版本*/
    if(strcmp(Protocol,NTPV1)||strcmp(Protocol,NTPV2)||strcmp(Protocol,NTPV3)
||strcmp(Protocol,NTPV4))
    {
```

```
        Port=NTPPORT;
        Version=Protocol[6]-0x30;
        Packet[0]=(Version<<3)|3;      //LI--Version--Mode
        Packet[1]=0;          //Startum
        Packet[2]=0;          //Poll interval
        Packet[3]=0;          //Precision
/*包括 Root delay、Root disperse 和 Ref Indentifier */
        memset(&Packet[4],0,12);
/*包括 Ref timestamp、Ori timastamp 和 Receive Timestamp */
        memset(&Packet[16],0,24);
        time(&timer);
        SecondFrom1900=SecondBef1970+(long)timer;
        SecondFrom1900=htonl(SecondFrom1900);
        memcpy(&Packet[40],&SecondFrom1900,4);
        memcpy(&Packet[44],&Zero,4);
        return 48;
        }
else   //time/udp
{
        Port=TIMEPORT;
        memset(Packet,0,4);
        return 4;
        }
return 0;
}

/*计算从 1900 年到现在一共有多少秒*/
long GetSecondFrom1900(int End)
{
int Ordinal=0;
int Run=0;
long Result;
int i;
        for(i=1900;i<End;i++)
        {
                if(((i%4==0)&&(i%100!=0))||(i%400==0)) Run++;
                else Ordinal++;
        }
        Result=(Run*366+Ordinal*365)*24*3600;
```

```
        return Result;
    }

    /*获取 NTP 时间*/
    long GetNtpTime(int sk,struct addrinfo *res)
    {
    char Content[256];
    int  PacketLen;
    fd_set PendingData;
    struct timeval BlockTime;
    int  FromLen;
    int  Count=0;
    int  result,i;
    int  re;
    struct NTPPacket RetTime;
        PacketLen=ConstructPacket(Content);
        if(!PacketLen)
            return 0;
    /*客户端给服务器端发送 NTP 协议数据包*/
        if((result=sendto(sk,Content,PacketLen,0,res->ai_addr,res->ai_addrl-
en))<0)
            perror("sendto");
        else
            printf("sendto success result=%d \n",result);
        for(i=0;i<5;i++)
        {
            printf("in for\n");
    /*调用 select 函数，并设定超时时间为 1s*/
            FD_ZERO(&PendingData);
            FD_SET(sk, &PendingData);
            BlockTime.tv_sec=1;
            BlockTime.tv_usec=0;
            if(select(sk+1,&PendingData,NULL,NULL,&BlockTime)>0)
            {
                FromLen=sizeof(sin);
    /*接收服务器端的信息*/
                if((Count=recvfrom(sk,Content,256,0,res->ai_addr,&(res->
ai_addrlen)))<0)
                    perror("recvfrom");
```

```
                    else
                          printf("recvfrom success,Count=%d \n",Count);
                    if(Protocol==TIME)
                    {
                          memcpy(RetTime.TransTimeStamp,Content,4);
                          return 1;
                    }
                    else if(Count>=48&&Protocol!=TIME)
                    {
                          RetTime.Leap_Ver_Mode=Content[0];
                          RetTime.Startum=Content[1];
                          RetTime.Poll=Content[2];
                          RetTime.Precision=Content[3];
                          memcpy((void *)&RetTime.RootDelay,&Content[4],4);
                          memcpy((void *)&RetTime.Dispersion,&Content[8],4);
                          memcpy((void *)RetTime.RefIdentifier,&Content[12],4);
                          memcpy((void *)RetTime.RefTimeStamp,&Content[16],8);
                          memcpy((void *)RetTime.OriTimeStamp,&Content[24],8);
                          memcpy((void *)RetTime.RecvTimeStamp,&Content[32],8);
                          memcpy((void *)RetTime.TransTimeStamp,&Content[40],8);
                          return 1;
                    }
              }
        }
        close(sk);
        return 0;
}

int main()
{
        memset(&hints,0,sizeof(hints));
        hints.ai_family=PF_UNSPEC;
        hints.ai_socktype=SOCK_DGRAM;
        hints.ai_protocol=IPPROTO_UDP;
/*调用 getaddrinfo 函数，获取地址信息*/
        rc=getaddrinfo("200.205.253.254","123",&hints,&res);
        if (rc != 0) {
              perror("getaddrinfo");
              return;
```

```
        }
        sk = socket(res->ai_family, res->ai_socktype,res->ai_protocol);
        if (sk <0 ) {
              perror("socket");
        }
        else
        {
              printf("socket success!\n");
        }
    /*调用取得 NTP 时间函数*/
        GetNtpTime(sk,res);
    }
```

本 章 小 结

本章首先概括地讲解了 OSI 分层结构以及 TCP/IP 协议各层的主要功能，介绍了常见的 TCP/IP 协议族，并且重点讲解了网络编程中需要用到的 TCP 和 UDP 协议，为嵌入式 Linux 的网络编程打下良好的基础。

接着本章介绍了 socket 的定义及其类型，并逐个介绍常见的 socket 基础函数，包括地址处理函数、数据存储转换函数，这些函数都是最为常用的函数，要在理解概念的基础上熟练掌握。

接下来介绍的是网络编程中的基础函数，这也是最为常见的几个函数，这里要注意 TCP 和 UDP 在处理过程中的不同。同时，本章还介绍了较为高级的网络编程，包括调用 fcntl 和 select 函数，这两个函数在之前都已经讲解过，但在这里会有特殊的用途。

最后，本章以 ping 函数为例，讲解了常见协议的实现过程，读者可以看到一个成熟的协议是如何实现的。

本章的实验安排了实现一个较为简单的 NTP 客户端程序，主要实现了其中数据收发的主要功能，至于其他时间调整相关的功能在这里就不详细介绍了。

思考与练习

实现一个小型模拟的路由器，就是接收从某个 IP 地址的连接，再把该请求转发到另一个 IP 地址的主机上去。