

## 第13章 原始套接字

利用“原始套接字”(Raw Socket), 我们可访问位于基层的传输协议。本章专门讲解如何运用这种原始套接字, 来模拟 IP 的一些实用工具, 比如 Traceroute 和 Ping 程序等等。使用原始套接字, 亦可对 IP 头信息进行实际的操作。本章只关心 IP 协议; 至于如何针对其他协议使用原始套接字, 我们打算提及。而且, 大多数协议(除 ATM 以外)根本就不支持原始套接字。所有原始套接字都是使用 SOCK\_RAW 这个套接字类型来创建的, 而且目前只有 Winsock 2 提供了对它的支持。因此, 无论 Microsoft Windows CE 还是老版本的 Windows 95 (无 Winsock 2 升级) 均不能利用原始套接字的能力。

此外, 要想顺利使用原始套接字, 要求对基层的协议结构有一定程度的认识, 而那已超出了本书的范围。在这一章中, 我们打算讨论 Internet 控制消息协议 (ICMP)、Internet 组管理协议 (IGMP) 以及用户数据报协议 (UDP)。ICMP 会由 Ping 这个实用程序用到, 以便探测到某个主机的路由是否有效和畅通, 看看对方的机器是否会作出响应。对程序开发者来说, 经常都要用到一种程序化的方法, 以便判断一台机器是否“活动”, 网络数据能否抵达它。IP 多播通信利用 IGMP 将多播组成员信息通告给路由器。大多数 Win32 平台目前都增加了对 IGMP 第 2 版的支持。但在某些情况下, 我们也需要送出自己的 IGMP 数据包, 以便脱离组成员关系。至于 UDP 协议, 我们打算把它同 IP\_HDRINCL 这个套接字选项组合起来讨论。以它为例, 讲述如何发送自己的 IGMP 包。对这三种协议来说, 我们都只会讲解与本章示范代码及示范程序密切相关的那些部分。

### 13.1 原始套接字的创建

要想使用原始套接字, 第一步便是创建它。可用 socket 命令或 WSASocket 调用来做到这一点。注意在典型情况下, 在 Winsock 为 IP 列出的目录中, 并不存在 SOCK\_RAW 这一套接字类型。然而, 这并不能妨碍我们创建此种类型的套接字。它的意思只是说, 我们不能用一个 WSAPROTOCOL\_INFO 结构来创建一个原始套接字。请参考第 5 章, 那里详细讲述了如何用 WSAEnumProtocols 函数以及 WSAPROTOCOL\_INFO 结构来列举协议条目。注意在套接字的创建过程中, 必须自行设定 SOCK\_RAW 标志。下述代码片段解释了如何将 ICMP 作为一种基层 IP 协议, 来完成一个原始套接字的创建:

```
SOCKET s;  
  
s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);  
// Or  
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,  
    WSA_FLAG_OVERLAPPED);  
if (s == INVALID_SOCKET)  
{  
    // Socket creation failed  
}
```

由于原始套接字使人们能对基层传输机制加以控制, 所以有些人将其用于不法用途, 从

而造成了 Windows NT 下一个潜在的安全漏洞。因此，只有属于“管理员”(Administrators)组的成员，才有权创建类型为 SOCK\_RAW 的套接字。而 Windows 95 和 Windows 98 均未施加这方面的限制。

要想在 Windows NT 中绕过这一限制，可考虑禁止对原始套接字的安全检查。方法是在注册表创建如下变量，并将它的值设为 1 (DWORD 类型)：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet  
    \Services\Afd\Parameters\DisableRawSecurity
```

更改了注册表后，注意重新启动计算机。

在上述示范代码中，我们采用的是 ICMP 协议。但假如想使用 IGMP、UDP、IP 或者原始 IP，只需分别设置 IPPROTO\_IGMP、IPPROTO\_UDP、IPPROTO\_IP 或者 IPPROTO\_RAW 即可。然而，请注意其中存在的一处限制：在 Windows NT 4、Windows 98 以及 Windows 95 (安装 Winsock 2) 操作系统中，创建原始套接字时，只能使用 IGMP 和 ICMP。协议标志 IPPROTO\_UDP、IPPROTO\_IP 以及 IPPROTO\_RAW 均要求使用套接字选项 IP\_HDRINCL，而该选项在上述平台下都是不支持的。然而，Windows 2000 确实提供了对 IP\_HDRINCL 选项的支持，所以能够处理 IP 头 (IPPROTO\_RAW)、TCP 头 (IPPROTO\_TCP) 以及 UDP 头 (IPPROTO\_UDP)。

采用恰当的协议标志，完成了原始套接字的创建之后，接下来的事情便是在发送及接收调用中，使用对应的套接字句柄。创建原始套接字时，IP 头会包含在接收到的任何返回数据中，无论是否设定了 IP\_HDRINCL 选项。

## 13.2 Internet 控制消息协议

Internet 控制消息协议 (ICMP) 是便于不同主机间传递简短消息的一种机制。大多数 ICMP 消息都牵涉到主机间通信时发生的一些错误；而其他 ICMP 消息用于对主机进行查询。ICMP 协议采用的是 IP 定址机制，因为它本身就是封装在 IP 数据报内的一种协议。在图 13-1 中，我们展示了 ICMP 消息的各个字段。整条 ICMP 消息封装在一个 IP 头内。

8位ICMP类型	8位ICMP代码	16位ICMP检验和
ICMP具体内容(取决于类型和代码)		

图13-1 ICMP头

其中，第一个字段指定的是 ICMP 消息类型，可分为查询或错误两类。随后，“代码”字段进一步定义了查询或消息的类型。而“校验和”字段的长度为 16 位，是对 ICMP 头内容的一个补余求和。最后，ICMP 的实际内容要依赖于前面设定的 ICMP 类型及代码。在表 13-1 中，我们对那些类型及代码进行了详细总结。

若生成的是一条 ICMP 错误消息，那么在消息中，肯定包含了导致那个 ICMP 错误的 IP 头，以及 IP 数据报的头 8 个字节。这样一来，收到 ICMP 错误的主机便可将消息与一种特定的协议联系起来。并可根据实际情况，对那个错误作出处理。就我们目前的情况来说，Ping 需要依赖于回应请求及回应答复这两种 ICMP 查询，而不是仅仅依赖一条错误消息。生成 ICMP 消息的主机会通过 TCP 或 UDP，对问题作出响应；除此之外，ICMP 的用处并不大。在下一节里，

我们打算讨论如何随原始套接字一道，用 ICMP 协议来生成一个 Ping 请求，Ping 请求是用来回应请求和回应该消息的。

表13-1 ICMP消息类型

类型	查询 / 错误 ( 错误类型 )	代码	说 明
0	查询	0	回应该 ( Echo Reply )
3	错误：目标不可抵达	0	网络访问不到
		1	主机访问不到
		2	协议访问不到
		3	端口访问不到
		4	数据包需要分段 ( 分解 )，但却没有设置分段位
		5	源路由失效
		6	目标网络未知
		7	目标主机未知
		8	源主机独立 ( 作废 )
		9	目标网络管理性禁止
		10	目标主机管理性禁止
		11	针对特定的 TOS ( 服务类型 )，网络不可抵达
		12	针对特定的 TOS ( 服务类型 )，主机不可抵达
		13	由于过滤，通信被管理性地禁止
		14	违反主机优先级设定
		15	优先级失效
4	错误	0	源主机停止工作
5	错误：重定向	0	为网络重定向
		1	为主机重定向
		2	和 TOS 和网络重定向
		3	和 TOS 和主机重定向
8	查询	0	回应该 ( Echo Request )
9	查询	0	路由器广告
10	查询	0	路由器请求
11	错误：超时	0	传输过程中 TTL 的值变成 0
		1	重新组合时 TTL 的值变成 0
12	错误：参数问题	0	IP 头损坏
		1	必需的选项丢失
13	查询	0	时间戳请求
14	查询	0	时间戳答复
15	查询	0	信息请求
16	查询	0	信息答复
17	查询	0	地址掩码请求
18	查询	0	地址掩码答复

### 13.2.1 Ping 示例

我们经常用 Ping 来判断一个特定的主机是否处于活动状态，并且是否可以通过网络访问到。通过生成一个 ICMP “回应该” ( Echo Request )，并将其定向至打算查询的目标主机，便可知道自己是否能成功地访问到那台机器。当然，这样做并不能担保一个套接字客户机能与那个主机上的某个进程顺利地建立连接 ( 例如，远程服务器上的一个进程也许并没有进入监听模式 )。若 Ping 成功，那么它只能证明一件事情：远程主机的网络层可对网络事件作出响

应！概括地说，我们的这个Ping示例需要采取下面这些步骤：

- 1) 创建类型为SOCK\_RAW的一个套接字，同时设定协议 IPPROTO\_ICMP。
- 2) 创建并初始化ICMP头。
- 3) 调用sendto或WSASendto，将ICMP请求发给远程主机。
- 4) 调用recvfrom或WSARecvfrom，以接收任何ICMP响应。

发出ICMP回应请求以后，远程机器会拦截这个请求，然后生成一条回应答复消息，再通过网络传回给我们。假如出于某些方面的原因，不能抵达目标主机，就会生成对应的 ICMP错误消息（比如“目标主机访问不到”），由原先打算建立通信的那个路径上某处的一个路由器返回。假定与远程主机的物理性连接并不存在问题，但远程主机已经关机或没有设置对网络事件作出响应，便需由自己的程序来执行超时检定，来侦测出这样的情况。程序清单 13-1列出的Ping.c示例向大家清晰展示了如何创建一个特殊的套接字，以便实现 ICMP包的收发；以及如何通过IP\_OPTIONS套接字选项，来实现记录路由选项。

程序清单 13-1 Ping.c

```
// Module Name: Ping.c
//
// Command Line Options/Parameters:
//   Ping [host] [packet-size]
//
//   host          String name of host to ping
//   packet-size    Integer size of packet to send
//                  (smaller than 1024 bytes)
//
// #pragma pack(1)

#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define IP_RECORD_ROUTE 0x7
//
// IP header structure
//
typedef struct _iphdr
{
    unsigned int    h_len:4;           // Length of the header
    unsigned int    version:4;         // Version of IP
    unsigned char   tos;                // Type of service
    unsigned short  total_len;          // Total length of the packet
    unsigned short  ident;              // Unique identifier
    unsigned short  frag_and_flags;    // Flags
    unsigned char   ttl;                // Time to live
    unsigned char   proto;              // Protocol (TCP, UDP, etc.)
    unsigned short  checksum;           // IP checksum

    unsigned int    sourceIP;
    unsigned int    destIP;
} IpHeader;
```

```

#define ICMP_ECHO            8
#define ICMP_ECHOREPLY      0
#define ICMP_MIN            8 // Minimum 8-byte ICMP packet (header)

//
// ICMP header structure
//
typedef struct _icmphdr
{
    BYTE    i_type;
    BYTE    i_code;           // Type sub code
    USHORT  i_cksum;
    USHORT  i_id;
    USHORT  i_seq;
    // This is not the standard header, but we reserve space for time
    ULONG   timestamp;
} IcmpHeader;

//
// IP option header--use with socket option IP_OPTIONS
//
typedef struct _ipoptionhdr
{
    unsigned char    code;           // Option type
    unsigned char    len;           // Length of option hdr
    unsigned char    ptr;           // Offset into options
    unsigned long     addr[9];       // List of IP addrs
} IpOptionHeader;

#define DEF_PACKET_SIZE 32          // Default packet size
#define MAX_PACKET      1024       // Max ICMP packet size
#define MAX_IP_HDR_SIZE 60         // Max IP header size w/options

BOOL  bRecordRoute;
int    datasize;
char *lpdest;

//
// Function: usage
//
// Description:
//     Print usage information
//
void usage(char *progname)
{
    printf("usage: ping -r <host> [data size]\n");
    printf("        -r            record route\n");
    printf("        host          remote machine to Ping\n");
    printf("        datasize      can be up to 1 KB\n");
    ExitProcess(-1);
}

//
// Function: FillICMPData
//
// Description:

```

```
// Helper function to fill in various fields for our ICMP request
//
void FillICMPData(char *icmp_data, int datasize)
{
    IcmpHeader *icmp_hdr = NULL;
    char        *datapart = NULL;

    icmp_hdr = (IcmpHeader*)icmp_data;
    icmp_hdr->i_type = ICMP_ECHO;           // Request an ICMP echo
    icmp_hdr->i_code = 0;
    icmp_hdr->i_id = (USHORT)GetCurrentProcessId();
    icmp_hdr->i_cksum = 0;
    icmp_hdr->i_seq = 0;

    datapart = icmp_data + sizeof(IcmpHeader);
    //
    // Place some junk in the buffer
    //
    memset(datapart, 'E', datasize - sizeof(IcmpHeader));
}

//
// Function: checksum
//
// Description:
//   This function calculates the 16-bit one's complement sum
//   of the supplied buffer (ICMP) header
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
    return (USHORT)(~cksum);
}

//
// Function: DecodeIPOptions
//
// Description:
//   If the IP option header is present, find the IP options
//   within the IP header and print the record route option
//   values
//
void DecodeIPOptions(char *buf, int bytes)
{

```

```

IpOptionHeader *ipopt = NULL;
IN_ADDR        inaddr;
int            i;
HOSTENT        *host = NULL;

ipopt = (IpOptionHeader *) (buf + 20);

printf("RR:  ");
for(i = 0; i < (ipopt->ptr / 4) - 1; i++)
{
    inaddr.S_un.S_addr = ipopt->addr[i];
    if (i != 0)
        printf(" ");
    host = gethostbyaddr((char *)&inaddr.S_un.S_addr,
        sizeof(inaddr.S_un.S_addr), AF_INET);
    if (host)
        printf("(%-15s) %s\n", inet_ntoa(inaddr), host->h_name);
    else
        printf("(%-15s)\n", inet_ntoa(inaddr));
}
return;
}

//
// Function: DecodeICMPHeader
//
// Description:
//   The response is an IP packet. We must decode the IP header to
//   locate the ICMP data.
//
void DecodeICMPHeader(char *buf, int bytes,
    struct sockaddr_in *from)
{
    IpHeader        *iphdr = NULL;
    IcmpHeader       *icmphdr = NULL;
    unsigned short   iphdrlen;
    DWORD            tick;
    static int        icmpcount = 0;

    iphdr = (IpHeader *)buf;
    // Number of 32-bit words * 4 = bytes
    iphdrlen = iphdr->h_len * 4;
    tick = GetTickCount();

    if ((iphdrlen == MAX_IP_HDR_SIZE) && (!icmpcount))
        DecodeIPOptions(buf, bytes);

    if (bytes < iphdrlen + ICMP_MIN)
    {
        printf("Too few bytes from %s\n",
            inet_ntoa(from->sin_addr));
    }
    icmphdr = (IcmpHeader*)(buf + iphdrlen);

    if (icmphdr->i_type != ICMP_ECHOREPLY)

```

```

{
    printf("nonecho type %d recvd\n", icmphdr->i_type);
    return;
}
// Make sure this is an ICMP reply to something we sent!
//
if (icmphdr->i_id != (USHORT)GetCurrentProcessId())
{
    printf("someone else's packet!\n");
    return ;
}
printf("%d bytes from %s:", bytes, inet_ntoa(from->sin_addr));
printf(" icmp_seq = %d. ", icmphdr->i_seq);
printf(" time: %d ms", tick - icmphdr->timestamp);
printf("\n");

icmpcount++;
return;
}

void ValidateArgs(int argc, char **argv)
{
    int            i;

    bRecordRoute = FALSE;
    lpdest = NULL;
    datasize = DEF_PACKET_SIZE;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'r':           // Record route option
                    bRecordRoute = TRUE;
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
        else if (isdigit(argv[i][0]))
            datasize = atoi(argv[i]);
        else
            lpdest = argv[i];
    }
}

//
// Function: main
//
// Description:
//     Set up the ICMP raw socket, and create the ICMP header. Add
//     the appropriate IP option header, and start sending ICMP

```



```
// echo requests to the endpoint. For each send and receive,
// we set a timeout value so that we don't wait forever for a
// response in case the endpoint is not responding. When we
// receive a packet, decode it.
//
int main(int argc, char **argv)
{
    WSADATA          wsaData;
    SOCKET           sockRaw = INVALID_SOCKET;
    struct sockaddr_in dest,
                    from;
    int              bread,
                    fromlen = sizeof(from),
                    timeout = 1000,
                    ret;
    char             *icmp_data = NULL,
                    *recvbuf = NULL;
    unsigned int     addr = 0;
    USHORT           seq_no = 0;
    struct hostent   *hp = NULL;
    IpOptionHeader   ipopt;

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
    ValidateArgs(argc, argv);

    //
    // WSA_FLAG_OVERLAPPED flag is required for SO_RCVTIMEO,
    // SO_SNDTIMEO option. If NULL is used as last param for
    // WSASocket, all I/O on the socket is synchronous, the
    // internal user mode wait code never gets a chance to
    // execute, and therefore kernel-mode I/O blocks forever.
    // A socket created via the socket function has the over-
    // lapped I/O attribute set internally. But here we need
    // to use WSASocket to specify a raw socket.
    //
    // If you want to use timeout with a synchronous
    // nonoverlapped socket created by WSASocket with last
    // param set to NULL, you can set the timeout by using
    // the select function, or you can use WSAEventSelect and
    // set the timeout in the WSAWaitForMultipleEvents
    // function.
    //
    sockRaw = WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
                        WSA_FLAG_OVERLAPPED);
    if (sockRaw == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    if (bRecordRoute)
```

```

{
    // Setup the IP option header to go out on every ICMP packet
    //
    ZeroMemory(&iptopt, sizeof(iptopt));
    iptopt.code = IP_RECORD_ROUTE; // Record route option
    iptopt.ptr = 4;                // Point to the first addr offset
    iptopt.len = 39;               // Length of option header

    ret = setsockopt(sockRaw, IPPROTO_IP, IP_OPTIONS,
        (char *)&iptopt, sizeof(iptopt));
    if (ret == SOCKET_ERROR)
    {
        printf("setsockopt(IP_OPTIONS) failed: %d\n",
            WSAGetLastError());
    }
}

// Set the send/rcv timeout values
//
bread = setsockopt(sockRaw, SOL_SOCKET, SO_RCVTIMEO,
    (char *)&timeout, sizeof(timeout));
if (bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_RCVTIMEO) failed: %d\n",
        WSAGetLastError());
    return -1;
}

timeout = 1000;
bread = setsockopt(sockRaw, SOL_SOCKET, SO_SNDTIMEO,
    (char *)&timeout, sizeof(timeout));
if (bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_SNDTIMEO) failed: %d\n",
        WSAGetLastError());
    return -1;
}

memset(&dest, 0, sizeof(dest));
//
// Resolve the endpoint's name if necessary
//
dest.sin_family = AF_INET;
if ((dest.sin_addr.s_addr = inet_addr(lpdest)) == INADDR_NONE)
{
    if ((hp = gethostbyname(lpdest)) != NULL)
    {
        memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
        dest.sin_family = hp->h_addrtype;
        printf("dest.sin_addr = %s\n", inet_ntoa(dest.sin_addr));
    }
    else
    {
        printf("gethostbyname() failed: %d\n",
            WSAGetLastError());
        return -1;
    }
}
}

```

```
//
// Create the ICMP packet
//
datasize += sizeof(IcmpHeader);

icmp_data = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                      MAX_PACKET);
recvbuf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                    MAX_PACKET);
if (!icmp_data)
{
    printf("HeapAlloc() failed: %d\n", GetLastError());
    return -1;
}
memset(icmp_data, 0, MAX_PACKET);
FillICMPData(icmp_data, datasize);
//
// Start sending/receiving ICMP packets
//
while(1)
{
    static int nCount = 0;
    int      bwrote;

    if (nCount++ == 4)
        break;
    ((IcmpHeader*)icmp_data)->i_cksum = 0;
    ((IcmpHeader*)icmp_data)->timestamp = GetTickCount();
    ((IcmpHeader*)icmp_data)->i_seq = seq_no++;
    ((IcmpHeader*)icmp_data)->i_cksum =
        checksum((USHORT*)icmp_data, datasize);

    bwrote = sendto(sockRaw, icmp_data, datasize, 0,
                   (struct sockaddr*)&dest, sizeof(dest));
    if (bwrote == SOCKET_ERROR)
    {
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            printf("timed out\n");
            continue;
        }
        printf("sendto() failed: %d\n", WSAGetLastError());
        return -1;
    }
    if (bwrote < datasize)
    {
        printf("Wrote %d bytes\n", bwrote);
    }
    bread = recvfrom(sockRaw, recvbuf, MAX_PACKET, 0,
                    (struct sockaddr*)&from, &fromlen);
    if (bread == SOCKET_ERROR)
    {
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            printf("timed out\n");
            continue;
        }
    }
}
```

```
    }  
    printf("recvfrom() failed: %d\\n", WSAGetLastError());  
    return -1;  
}  
DecodeICMPHeader(recvbuf, bread, &from);  
  
Sleep(1000);  
}  
// Cleanup  
//  
if (sockRaw != INVALID_SOCKET)  
    closesocket(sockRaw);  
HeapFree(GetProcessHeap(), 0, recvbuf);  
HeapFree(GetProcessHeap(), 0, icmp_data);  
  
WSACleanup();  
return 0;  
}
```

对这个Ping示范程序来说,它最引人注目的一个特点便是采用了IP\_OPTIONS套接字选项。之所以要使用记录路由IP选项,是由于这样一来,当我们的ICMP包抵达路由器时,它的IP地址便能自动添加到IP选项头内——具体插入位置则取决于事先在IP选项头内设好的偏移量字段。每次遇到一个路由器向其中加入自己的IP地址时,这个偏移距离都会自动递增4。之所以是“4”,而不是其他数字,是由于对IPv4地址来说,它的长度正好是4个字节。本书不打算对IPv6的情况作深入探讨,目前也没有任何一种Windows平台提供了对它的支持。一旦收到回应答复,便可对选项头进行解析,列印出中途访问过的那些路由器的IP地址及主机名。大家可参考第9章的内容,了解还能使用其他什么类型的IP选项。

### 13.2.2 Traceroute示例

对IP网络来说,另一个非常有用的工具是Traceroute(路由追踪)。在Windows操作系统中,一般可以直接运行Tracert.exe,来调用这个工具。利用它,我们可侦测出为抵达网络内任何一个指定的主机,中途需经过哪些路由器,以及它们的IP地址是什么。当然,亦可利用Ping,使用IP选项头内的记录路由选项,侦测中途经过的路由器IP地址。然而,Ping最多只支持9次“跳跃”——这是在选项头内为地址分配的最大空间。一个IP数据报需要穿越多个物理性的网络时,每通过一个路由器,我们就说进行了一次“跳跃”。若网络较大(如Internet),穿过的路由器不止9个,便应换用Traceroute。

Traceroute的设计原理是令其向目的地发送一个UDP数据包,并重复递增IP的“存活时间”(TTL)值。最开始的时候,TTL等于1;也就是说,一旦它抵达路途中的第一个路由器,TTL首先会超时(变成0)。这样便会造成路由器生成一个ICMP“超时”数据包。随后,最初的TTL值递增1,以便UDP包能继续传到下一个路由器,而生成的ICMP超时包会自第一个路由器返回。只需将返回的每一条ICMP消息都收集下来,便能为中途经过的路由器IP地址勾勒出一个清晰的轮廓,直到最终抵达目标主机。一旦TTL的值递增得足够大,可以实际抵达目标位置,通常便会返回一条ICMP“端口访问不到”消息,因为在接收端主机上,并没有进程在等待这条消息。

之所以认为Traceroute是一个有用的工具,是由于它为我们提供了与中途经历的路由器有

关的大量信息。进行多播通信，或者在遇到路由问题的时候，这些资料便显得非常宝贵。对具体的应用程序来说，尽管需要执行 Traceroute的机会比Ping少得多，但对某些特定的任务而言，却恐怕只有Traceroute才能胜任。

有两个办法可用来实现 Traceroute程序。第一个办法，可使用 UDP包和发送数据报，连续递增更改TTL的值。TTL每一次“超时”，都会向我们返回一条ICMP消息。这种方法要求使用安装了UDP协议的一个套接字来发送数据，同时用安装了ICMP协议的另一个套接字来接收数据（读取返回的消息）。这个UDP套接字本身是一个极为普通的UDP套接字，如大家在第7章中所见。而ICMP套接字的类型比较特殊，是SOCK\_RAW（原始套接字），并设定了IPPROTO\_ICMP协议。UDP套接字的TTL值需要通过IP\_TTL套接字选项加以控制。此外，也可以创建一个UDP套接字，并使用IP\_HDRINCL选项（本章稍后还会详述），在IP头内对TTL进行人工设置。当然，这要求程序员进行更多的工作。

第二个办法，我们只需将ICMP数据包简单地发给目的地，同时连续递增更改TTL的值。在TTL“超时”的时候，这样做也会造成一条ICMP错误消息的返回。注意这种方法和Ping有着某些共通之处，它也只要求使用一个套接字（安装ICMP协议）。在本书配套光盘的示范代码文件夹下，大家可找到一个使用了ICMP数据包的Traceroute例子。本章正文并不打算列出它的完整源码，因为它的大部分代码都与前面的Ping例子相同。

### 13.3 Internet组管理协议

Internet组管理协议（IGMP）由IP多播通信专用，可对多播组的成员加入或脱离组进行管理。大家可参考第11章的内容，了解如何通过Winsock来加入及离开一个多播组。某个应用程序加入了一个多播组后，便会向本地网络的每个路由器都发出一条IGMP消息，使用特殊地址224.0.0.2，该地址固定对应于“所有路由器”组。根据这条IGMP消息，路由器便可知道以后发给指定多播地址的数据需要转发给这个新加入的成员。若没有这项能力，多播数据便和广播数据没什么区别了。

现在来澄清一些事实。IGMP协议共有两个版本：版本1（IGMPv1）和版本2（IGMPv2），分别由RFC 1112和RFC 2236文件加以定义。两者的主要区别在于：版本1没有提供专门的方法，让一个主机指示路由器停止转发送给一个多播组的数据。换言之，若某个主机决定脱离组成员关系，便没有办法将这种情况通报给路由器，而路由器当然会坚持不懈地向其转发数据，直至在其发出一个组查询后，发现对方没有“应答”。在协议第2版中，则增加了一条能够明确表明自己要“离开”的消息。这样一来，一旦打算放弃成员资格，主机便可向路由器发出通知，使其能够立即知道这一情况。除此以外，两个版本的头格式也存在一些不同。图13-2展示了版本1的头结构，图13-3则是版本的2头结构。由于长度仅为8字节，所以这两个头其实都非常简单。

4位IGMP版	4位IGMP类型	8位长度未用	16位IGMP检验和
32位多播地址(D类IP地址)			

图13-2 IGMPv1头格式

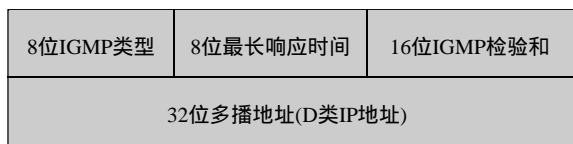


图13-3 IGMPv2头格式

其中，版本1的头含有两个4位字段。第一个字段指定IGMP版本，而第二个字段指定IGMP消息类型。在版本2中，单独一个8位字段代替了这两个字段。此外，版本1中未用的字段在新版本中派上了用场，用来指定最长的响应时间。只有在成员关系查询消息中，“最长响应时间”字段才有意义：它代表在发出一个响应报告之前，允许等候的最长时间；采用的单位是0.1秒。在其他所有消息中，发送都会将该字段设为0，而接收者也会忽略它。

对IGMPv1来说，版本字段必然设为1，而类型字段有两个可选的值，如表13-2所示。使用主机成员资格查询（0x1）消息，路由器可判断一个主机正在使用哪些多播组。在这种情况下，组地址字段的值应设为0。路由器会将查询数据包发给“所有主机”地址（224.0.0.1）。假如有主机仍是某个组的成员，那么各个主机都会将一个主机成员资格报告（IGMP消息类型0x2）反馈回“所有路由器”地址，同时指出自己加入的是哪个多播地址。假如一个主机是首次加入一个组，那么一条组成员资格报告消息也会发给“所有路由器”组。

如表13-3所示，IGMP的第2版加入了两种新的消息类型——版本2成员资格报告（0x16），以及离开组（0x17）消息。注意另两类消息在作用上与版本1是相同的（主机成员资格查询和成员资格报告），尽管为其分配的值不同。然而，请注意版本2的IGMP包已将版本及类型字段浓缩到单独一个字段里，而0x11展开成二进制，便是00010001。从表面看，完全相当于“版本等于1，类型等于1”！

表13-2 IGMP版本1的消息类型

类型	说 明
0x1	主机成员资格查询
0x2	主机成员资格报告

表13-3 IGMP版本2的消息类型

类型	说 明
0x11	成员资格查询
0x12	版本1成员资格报告
0x16	版本2成员资格报告
0x17	离开组

版本2的成员资格查询（0x11）与版本1的成员资格查询差别是极其微小的。前者不仅能进行版本1那样的标准查询，也能在一个网络上，查询在一个指定的组地址中，包括了哪些组成员。具体的做法是向“所有路由器”组发送一个成员资格查询包，同时在组地址字段中指定一个打算查询的目标组。版本2的成员资格报告则是最新设计的，因为它允许使用“最长超时”字段，对主机响应查询的时间作出限制。如超过时间仍未作出响应，便会禁止向其转发特定多播组的数据。

## 13.4 IP\_HDRINCL的使用

对原始套接字来说，它存在的一项限制在于，我们只能对已经定义好的协议进行操作，比如IGMP和ICMP等等。不能用IPPROTO\_UDP来创建一个新的原始套接字，也不能对UDP头进行操作；TCP也是一样的。要想对IP头进行操作，同时也能操作TCP或UDP头（或封装在IP内的其他任何协议），必须随原始套接字一道，使用IP\_HDRINCL这个套接字选项。利用该选项，我们除了能自行构建IP头，亦能构建其他协议头。

此外，假如想在应用程序中实现自己的协议方案，并将其封装到IP内，那么首先可以创建一个原始套接字，然后将协议类型设为IPPROTO\_RAW。这样一来，我们就可在IP头内人工设置协议字段，同时构建自己的定制协议头。在本小节内，我们打算讲述如何构建自己的UDP数据包，使大家对其中牵涉到的步骤有一个比较全面的了解。一旦理解了如何操作UDP头，再来创建自己的协议头，或对IP内封装的其他协议进行处理，便只是“小菜一碟”。

使用IP\_HDRINCL选项时，必须针对每一次发送调用，向IP头内自行填充内容。同时，还要填写封装在其中的其他任何协议头。IP头的结构已在第9章进行了讲述。大家可参考图9-3，以及专门讲述IP\_HDRINCL选项的那一小节。与IP相比，UDP头要简单得多。长度仅为8个字节，而且只包含了四个字段，如图13-4所示。其中，头两个字段分别对应源和目标端口号，长度各自为16位。第三个字段对应UDP长度；以字节为单位，指定UDP头以及数据的总长。第四个字段则是校验和，稍后还会对此详加解释。UDP包的最后一部分便是实际的数据。

16位源端口	16位目标端口
16位UDP长度	16位UDP校验和

图13-4 UDP头的格式

由于UDP是一种不能保证数据可靠传输的协议（即“不可靠”协议），所以校验和的计算是可选的。然而，为保持大家知识系统的完整性，在此还是要对其进行解释。与IP校验和不同，UDP校验和除覆盖了UDP头之外，还同时覆盖了实际的数据，此外还包括IP头的一部分。而IP校验和只针对IP头进行计算。要求用于计算UDP校验和的附加字段叫作“伪头”。一个伪头由下述项目构成：

- 32位源IP地址（IP头）。
- 32位目标IP地址（IP头）。
- 8位字段（零除外）。
- 8位协议。
- 16位UDP长度。

加入这些项目的是UDP头以及数据。校验和的计算方法和IP与ICMP的方法是相同的：得出16位1的求余总和。由于数据可能是个奇数，所以有时需要在数据末尾填充一个0字节，以便顺利计算出校验和。这个填充字段并不作为数据的一部分传递。在图13-5中，我们向大家演示了计算校验和所需的全部字段。头三个32位字构成了UDP伪头。紧接着的是UDP头及其数据。要注意的是，由于校验和是以16位值为基础计算出来的，所以或许需要用0字节对数据进行填充。





图13-5 UDP伪头

程序清单 13-2列出的示范程序作用很简单，从我们选择的任何 IP及端口，它向任何指定的目标 IP及端口地址发送一个 UDP数据包。第一步是创建一个原始套接字，然后设置 IP\_HDRINCL标志：

```
SOCKET  s;
BOOL    bOpt;

s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,
              WSA_FLAG_OVERLAPPED);
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
                sizeof(bOpt));
```

注意，我们现已创建了一个协议为 IPPROTO\_UDP的原始套接字。这种做法只适用于 Windows 2000，因为它同时也要求设置 IP\_HDRINCL选项。示范程序 Iphdrinc.c向大家阐释了如何利用原始套接字和 IP\_HDRINCL选项，操作传出去的数据包的 IP与UDP头。

程序清单 13-2 原始UDP示例

```
// Module Name: Iphdrinc.c
//
#pragma pack(1)

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>

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

#define MAX_MESSAGE      4068
#define MAX_PACKET      4096
//
// Set up some default values
//
#define DEFAULT_PORT     5150
#define DEFAULT_IP       "10.0.0.1"
#define DEFAULT_COUNT    5
#define DEFAULT_MESSAGE  "This is a test"

//
// Define the IP header. Make the version and length fields one
// character since we can't declare two 4-bit fields without
// the compiler aligning them on at least a 1-byte boundary.
```



```
//
typedef struct ip_hdr
{
    unsigned char ip_verlen;           // IP version & length
    unsigned char ip_tos;              // IP type of service
    unsigned short ip_totallength;     // Total length
    unsigned short ip_id;              // Unique identifier
    unsigned short ip_offset;          // Fragment offset field
    unsigned char ip_ttl;              // Time to live
    unsigned char ip_protocol;         // Protocol(TCP, UDP, etc.)
    unsigned short ip_checksum;        // IP checksum
    unsigned int ip_srcaddr;           // Source address
    unsigned int ip_destaddr;          // Destination address
} IP_HDR, *PIP_HDR, FAR* LPIP_HDR;
//
// Define the UDP header
//
typedef struct udp_hdr
{
    unsigned short src_portno;         // Source port number
    unsigned short dst_portno;         // Destination port number
    unsigned short udp_length;         // UDP packet length
    unsigned short udp_checksum;       // UDP checksum (optional)
} UDP_HDR, *PUDP_HDR;

//
// Global variables
//
unsigned long dwToIP,                 // IP to send to
dwFromIP;                             // IP to send from (spoof)
unsigned short iToPort,               // Port to send to
iFromPort;                             // Port to send from (spoof)
DWORD dwCount;                        // Number of times to send
char strMessage[MAX_MESSAGE];        // Message to send

//
// Description:
//   Print usage information and exit
//
void usage(char *programe)
{
    printf("usage: %s [-fp:int] [-fi:str] [-tp:int] [-ti:str]\n", programe);
    printf("    [-n:int] [-m:str]\n", programe);
    printf("    -fp:int   From (sender) port number\n");
    printf("    -fi:IP    From (sender) IP address\n");
    printf("    -tp:int   To (recipient) port number\n");
    printf("    -ti:IP    To (recipient) IP address\n");
    printf("    -n:int    Number of times to read message\n");
    printf("    -m:str    Size of buffer to read\n\n");
    ExitProcess(1);
}
//
// Function: ValidateArgs
//
// Description:
```

```

// Parse the command line arguments, and set some global flags to
// indicate the actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int                i;

    iToPort = DEFAULT_PORT;
    iFromPort = DEFAULT_PORT;
    dwToIP = inet_addr(DEFAULT_IP);
    dwFromIP = inet_addr(DEFAULT_IP);
    dwCount = DEFAULT_COUNT;
    strcpy(strMessage, DEFAULT_MESSAGE);

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'f':                // From address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
                            if (strlen(argv[i]) > 4)
                                iFromPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwFromIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 't':                // To address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
                            if (strlen(argv[i]) > 4)
                                iToPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwToIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 'n':                // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
            }
        }
    }
}

```

```

        case 'm':
            if (strlen(argv[i]) > 3)
                strcpy(strMessage, &argv[i][3]);
            break;
        default:
            usage(argv[0]);
            break;
    }
}
}
return;
}

//
// Function: checksum
//
// Description:
//   This function calculates the 16-bit one's complement sum
//   for the supplied buffer
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);

    return (USHORT)(~cksum);
}

//
// Function: main
//
// Description:
//   First parse command line arguments and load Winsock. Then
//   create the raw socket and set the IP_HDRINCL option.
//   Following this, assemble the IP and UDP packet headers by
//   assigning the correct values and calculating the checksums.
//   Then fill in the data and send to its destination.
//
int main(int argc, char **argv)
{
    WSADATA          wsd;
    SOCKET           s;
    BOOL             bOpt;
    struct sockaddr_in remote;    // IP addressing structures
    IP_HDR           ipHdr;
    UDP_HDR          udpHdr;
    int              ret;
    DWORD            i;

```

```

unsigned short    iTotalSize,    // Lots of sizes needed to fill
                  iUdpSize,      // the various headers with
                  iUdpChecksumSize,
                  iIPVersion,
                  iIPSize,
                  cksum = 0;
char              buf[MAX_PACKET],
                  *ptr = NULL;
IN_ADDR           addr;

// Parse command line arguments, and print them out
//
ValidateArgs(argc, argv);
addr.S_un.S_addr = dwFromIP;
printf("From IP: <%s>\n      Port: %d\n", inet_ntoa(addr),
        iFromPort);
addr.S_un.S_addr = dwToIP;
printf("To   IP: <%s>\n      Port: %d\n", inet_ntoa(addr),
        iToPort);
printf("Message: [%s]\n", strMessage);
printf("Count:   %d\n", dwCount);

if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    printf("WSAStartup() failed: %d\n", GetLastError());
    return -1;
}
// Creating a raw socket
//
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,0);
if (s == INVALID_SOCKET)
{
    printf("WSASocket() failed: %d\n", WSAGetLastError());
    return -1;
}

// Enable the IP header include option
//
bOpt = TRUE;
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
    sizeof(bOpt));
if (ret == SOCKET_ERROR)
{
    printf("setsockopt(IP_HDRINCL) failed: %d\n", WSAGetLastError());
    return -1;
}
// Initialize the IP header
//
iTotalSize = sizeof(ipHdr) + sizeof(udpHdr) + strlen(strMessage);

iIPVersion = 4;
iIPSize = sizeof(ipHdr) / sizeof(unsigned long);
//
// IP version goes in the high-order 4 bits of ip_verlen. The
// IP header length (in 32-bit words) goes in the lower 4 bits.
//

```

```

ipHdr.ip_verlen = (iIPVersion << 4) | iIPSize;
ipHdr.ip_tos = 0; // IP type of service
ipHdr.ip_totallength = htons(iTotalSize); // Total packet len
ipHdr.ip_id = 0; // Unique identifier: set to 0
ipHdr.ip_offset = 0; // Fragment offset field
ipHdr.ip_ttl = 128; // Time to live
ipHdr.ip_protocol = 0x11; // Protocol(UDP)
ipHdr.ip_checksum = 0; // IP checksum
ipHdr.ip_srcaddr = dwFromIP; // Source address
ipHdr.ip_destaddr = dwToIP; // Destination address
//
// Initialize the UDP header
//
iUdpSize = sizeof(udpHdr) + strlen(strMessage);

udpHdr.src_portno = htons(iFromPort);
udpHdr.dst_portno = htons(iToPort);
udpHdr.udp_length = htons(iUdpSize);
udpHdr.udp_checksum = 0;
//
// Build the UDP pseudo-header for calculating the UDP checksum.
// The pseudo-header consists of the 32-bit source IP address,
// the 32-bit destination IP address, a zero byte, the 8-bit
// IP protocol field, the 16-bit UDP length, and the UDP
// header itself along with its data (padded with a 0 if
// the data is odd length).
//
iUdpChecksumSize = 0;
ptr = buf;
ZeroMemory(buf, MAX_PACKET);

memcpy(ptr, &ipHdr.ip_srcaddr, sizeof(ipHdr.ip_srcaddr));
ptr += sizeof(ipHdr.ip_srcaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_srcaddr);

memcpy(ptr, &ipHdr.ip_destaddr, sizeof(ipHdr.ip_destaddr));
ptr += sizeof(ipHdr.ip_destaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_destaddr);

ptr++;
iUdpChecksumSize += 1;

memcpy(ptr, &ipHdr.ip_protocol, sizeof(ipHdr.ip_protocol));
ptr += sizeof(ipHdr.ip_protocol);
iUdpChecksumSize += sizeof(ipHdr.ip_protocol);

memcpy(ptr, &udpHdr.udp_length, sizeof(udpHdr.udp_length));
ptr += sizeof(udpHdr.udp_length);
iUdpChecksumSize += sizeof(udpHdr.udp_length);

memcpy(ptr, &udpHdr, sizeof(udpHdr));
ptr += sizeof(udpHdr);
iUdpChecksumSize += sizeof(udpHdr);

for(i = 0; i < strlen(strMessage); i++, ptr++)
    *ptr = strMessage[i];
iUdpChecksumSize += strlen(strMessage);

```

```
cksum = checksum((USHORT *)buf, iUdpChecksumSize);
udpHdr.udp_checksum = cksum;
//
// Now assemble the IP and UDP headers along with the data
// so we can send it
//
ZeroMemory(buf, MAX_PACKET);
ptr = buf;

memcpy(ptr, &ipHdr, sizeof(ipHdr)); ptr += sizeof(ipHdr);
memcpy(ptr, &udpHdr, sizeof(udpHdr)); ptr += sizeof(udpHdr);
memcpy(ptr, strMessage, strlen(strMessage));

// Apparently, this SOCKADDR_IN structure makes no difference.
// Whatever we put as the destination IP addr in the IP header
// is what goes. Specifying a different destination in remote
// will be ignored.
//
remote.sin_family = AF_INET;
remote.sin_port = htons(iToPort);
remote.sin_addr.s_addr = dwToIP;

for(i = 0; i < dwCount; i++)
{
    ret = sendto(s, buf, iTotalSize, 0, (SOCKADDR *)&remote,
        sizeof(remote));
    if (ret == SOCKET_ERROR)
    {
        printf("sendto() failed: %d\n", WSAGetLastError());
        break;
    }
    else
        printf("sent %d bytes\n", ret);
}
closesocket(s);
WSACleanup();

return 0;
}
```

建好套接字，并设置了IP\_HDRINCL选项之后，代码会开始IP头的填写。大家可注意到，代码将IP头声明为一个结构：IP\_HDR。注意头两个4位字段已合并成单独一个字段，因为编译程序只能在最小1个字节的边界上对齐字段。正是由于这个原因，代码必须将IP版本置于高4位。ip\_protocol字段的值设为0x11，它对应于UDP。代码也将ip\_srcaddr字段设为源IP地址（或者想让接收者认为的任何地址），并将ip\_destaddr字段设为接收者的IP地址。网络堆栈会计算出IP校验和，所以代码中不必另行设置。

下一步是对UDP头进行初始化。这个操作是非常简单的，因为字段的数量本来就不多。设置了源和目标端口号之后，同时还要设置UDP头的长度。校验和字段设为0。尽管在你编写的代码中，可能并不需要进行UDP校验和的计算，但这里的例子却那样做了，目的只是为了阐述如何用UDP“伪头”来做这件事情。为使计算简单一些，我们将所有必要的字段都复制到一个临时性的字符缓冲区内：buf。然后，将该缓冲区传递给我们的校验和计算函数，同时

指出缓冲区的长度，完成最终的计算。

发出数据报之前，要做的最后一件事情是将消息的各个部分“组装”到一个邻近的缓冲区内。只需简单地使用memcpy函数，便可将IP、UDP头以及数据复制到一个邻近的缓冲区内。接着调用sendto函数，将数据发送出去。注意在使用IP\_HDRINCL选项的时候，sendto中的to参数会被忽略。这是由于数据无论如何都会发送给我们在IP头内指定的那个主机！

下面来看看Iphdrinc.c程序实际运行时的情况。此时可使用第7章讲述的那个UDP接收者应用程序：Receiver.c。例如，可用下述参数来启动UDP接收者应用程序：

```
Receiver.exe -p:5150 -i:xxx.xxx.xxx.xxx -n:5 -b:1000
```

如果机器只安装了一个网络接口（即只对应一个IP地址），那么可考虑省略-i参数；否则的话，便请指定自己打算使用的那个接口的IP地址。在Windows 2000机器中，以下述形式启动Iphdrinc.c程序：

```
Iphdrinc.exe -fi:1.2.3.4 -fp:10 -ti:xxx.xxx.xxx.xxx -tp:5150 -n:5150
```

注意，用-ti参数设定的IP地址必须是接收者正在上面监听的同样的一个IP地址。随后，接收者应用程序应输出下述报告：

```
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
```

结束了对recvfrom的调用之后，Receiver.c除列印出收到的消息之外，还会显示出自SOCKADDR\_IN结构返回的地址信息，该结构已传递进入recvfrom。随后，可检验收到数据包的IP头是否与使用“Microsoft网络监视器”在网上捕捉数据包获得的结果相同。在表13-4中，我们列出了Iphdrinc示例可选的各个命令行参数。注意可自行设定源和目标的IP地址/端口编号。

表13-4 Iphdrinc.c参数

参 数	说 明
-fi:xxx.xxx.xxx.xxx	IP包的源IP地址
-fp:整数	IP包的源端口号
-ti:xxx.xxx.xxx.xxx	IP包的目标IP地址
-tp:整数	IP包的目标端口
-n:整数	要发送的UDP数据报的数量
-m:字符串	要送出的消息

## 13.5 小结

原始套接字是一种强有力的机制，可供我们对基层协议进行操作。本章向大家阐述了如何通过Winsock，利用原始套接字来创建ICMP和IGMP应用程序。但要注意的是，原始套接字亦适用于其他许多应用，本章不可能全部都能讲到。要想充分发挥原始套接字以及IP\_HDRINCL选项的功能，必须对IP协议以及IP内封装的其他所有协议有一个通盘、透彻的理解。