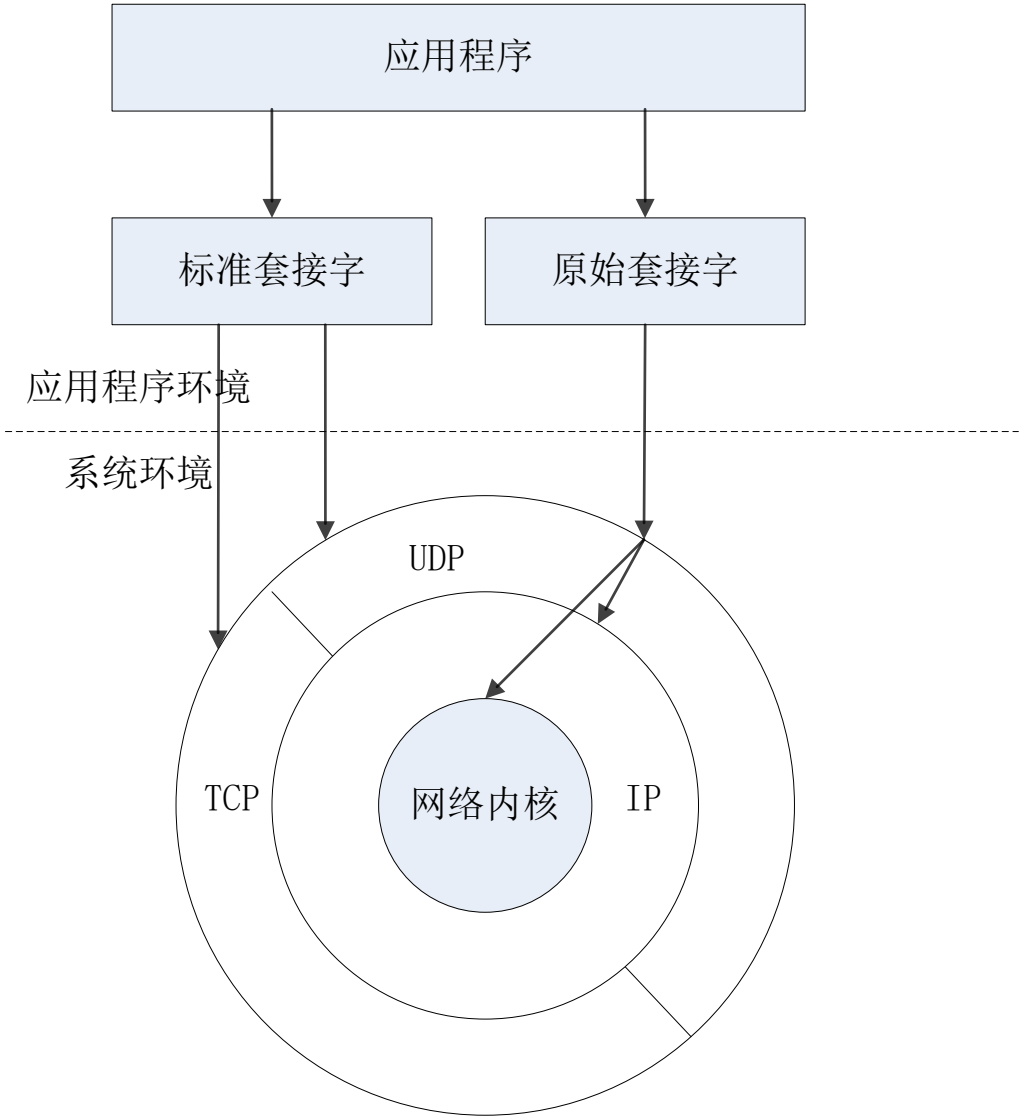
网络程序设计

原始套接字

原始套接字的能力

- Question?
  - 怎样发送一个自定义的IP数据包？
  - 怎样接收ICMP协议承载的差错报文？
  - 怎样使主机捕获网络中其它主机间的报文？
  - 怎样伪装本地的IP地址？
  - 怎样模拟TCP或UDP协议的行为实现对协议的灵活操控？

原始套接字的能力



原始套接字的能力

- 原始套接字提供普通TCP和UDP套接字不提供的以下三种能力
  - 读写ICMPv4、IGMPv4和ICMPv6等分组。
  - 读写内核不处理其协议字段的IPv4数据报。
  - 控制IPv4头部。

原始套接字的能力

- 原始套接字的适用场合
  - （1）特殊用途的探测应用
  - （2）基于数据包捕获的应用
  - （3）特殊用途的传输应用

原始套接字编程模型

发送数据过程：

①socket初始化

②创建套接字，指定使用原始套接字进行通信，指定IP头选项；(两种选择：I.仅构造IP数据II.构造IP首部和IP数据)

③指定目的地址和通信端口；

④填充头和数据；

⑤发送数据；

⑥关闭套接字；

⑦结束对windows sockets dll的使用。

原始套接字编程模型

接收数据过程：

①socket初始化

②创建套接字，指定使用原始套接字进行通信；

③设置接收选项；

④接收数据；

⑤过滤数据；

⑥关闭套接字；

⑦结束对windows sockets dll的使用。

网络接口提交给原始套接字的数据并不一定是网卡接收到的所有数据，如果希望得到特定类型的数据报，在步骤3，应用程序可能需要对套接字的接收进行控制，设定接收选项。

问题1：原始套接字与流式套接字和数据报套接字在编程过程中增加了哪些附加操作？

问题2：应用程序能够接收到哪些数据？

原始套接字创建——1）创建函数

SOCK_RAW

SOCKET socket(int *af*, int *type*, int *protocol*)

协议	值	含义
IPPROTO_IP	0	IP协议
IPPROTO_ICMP	1	ICMP协议
IPPROTO_IGMP	2	IGMP协议
BTHPROTO_RFCOMM	3	蓝牙通信协议
IPPROTO_IPV4	4	IPv4协议
IPPROTO_TCP	6	TCP协议
IPPROTO_UDP	17	UDP协议
IPPROTO_ IPV6	41	IPv6协议
IPPROTO_ ICMPV6	58	ICMPv6协议
IPPROTO_ RAW	255	原始IP包

– 创建者的权限——administrator

原始套接字创建——2）IP首部控制

- IP_HDRINCL选项

函数定义：**int setsockopt(SOCKET** *s*, **int** *level*, **int** *optname*, **const char*** *optval*, **int** *optlen* **);**

输入参数：

s：套接字描述符；

level：选项定义的层次，如**IPPROTO_IP**；

optname：指定套接字选项的名字

optval：指向请求选项数据缓冲区的指针；

optlen：选项数据**optval**缓冲区的长度。

返回值：

正确：**0**；错误：**SOCKET_ERROR**（**WSAGetLastError**）

功能：为套接字相关的选项设置当前值。

说明：套接字选项有两种类型：设置或禁止特征

原始套接字创建——2）IP首部控制

- 说明

  套接字选项有两种类型

  - 设置或禁止特征/行为的布尔选项
    - 设置：**Optval**指向一个非零整数
    - 禁止：**optval**指向一个等于零的整数
    - **Optlen**等于整型数的长度
  - 要求整数值或结构的选项
    - **Optval**指向一个包含选项要求值的整数或结构
    - **Optlen**为此整数或结构的长度

原始套接字创建——2）IP首部控制

- 举例：
  - 布尔选项设置：

```
BOOL blnFlag=TRUE;
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char *)&blnFlag, sizeof(blnFlag);
```

  - 整数值选项设置：

例**1**：int ttl = 7 ; // TTL value.

```
setsockopt(sock, IPPROTO_IP, IP_TTL, (char *)&ttl, sizeof(ttl))
```

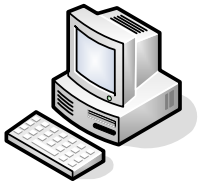
例**2**：nTimeOver=1000;

```
setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO, (char*)&nTimeOver,sizeof(nTimeOver))
```

原始套接字创建——3）端点地址关联

- Bind
  - 功能：指定从这个原始套接字发送的所有数据报的源IP地址
  - 作用范围：原始套接字不存在端口号概念，bind函数仅仅设置本地地址
- Connect
  - 功能：指定从这个原始套接字发送的所有数据报的目的IP地址
  - 作用范围：原始套接字不存在端口号概念，connect函数仅设置远端IP地址

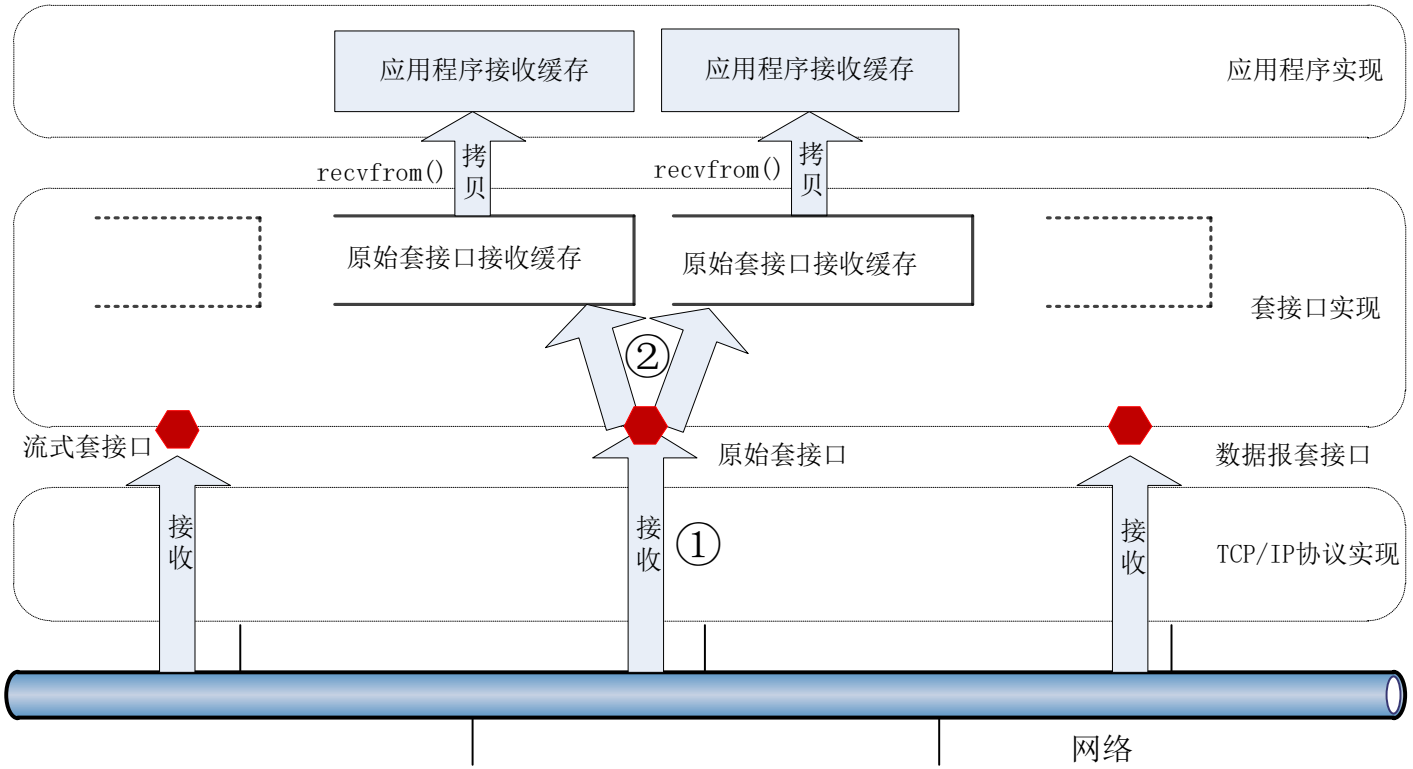
原始套接字输入



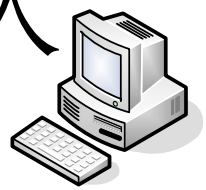
- 接收到的内容
  - 包括IP头部在内的完整数据报（IPv4）

原始套接字输入

- 接收到的类型



原始套接字输入



- 接收到的类型

内核把哪些接收到的IP数据报传递到原始套接口？

- ≠UDP分组和TCP分组
- ≠片段分组
- =内核处理后的ICMP、IGMP分组
- =不认识其协议字段的所有IP数据报

一个原始套接口能够收到哪些数据？

- SOCKET（协议）：匹配数据报的协议字段
- Bind：匹配数据报的宿IP地址
- Connect：匹配数据报的源IP地址

原始套接字输入

- 既然原始套接字能够接收到的数据报文是有限制的，那么如何在第一个层次上扩展原始套接口被复制的数据种类呢？

套接字控制命令SIO_RCVALL

使用方法

1.创建原始套接字，由于IPv6尚未实现SIO_RCVALL，因而套接字地址簇必须是AF_INET，协议必须是IPPROTO_IP；
2.将套接字绑定到指定的本地接口；
3.调用WSAIoctl为套接字设置SIO_RCVALL I/O控制命令；
4.调用接收函数，捕获IP数据报。

```
int
WSAAPI
WSAIoctl(
    _In_ SOCKET s,
    _In_ DWORD dwIoControlCode,
    _In_reads_bytes_opt_(cbInBuffer) LPVOID lpvInBuffer,
    _In_ DWORD cbInBuffer,
    _Out_writes_bytes_to_opt_(cbOutBuffer, *lpcbBytesReturned) LPVOID lpvOutBuffer,
    _In_ DWORD cbOutBuffer,
    _Out_ LPDWORD lpcbBytesReturned,
    _Inout_opt_ LPWSAOVERLAPPED lpOverlapped,
    _In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
```

SIO_RCVALL控制码的使用

lpvInBuffer

A pointer to the input buffer that should contain the option value. The possible values for the **SIO_RCVALL** IOCTL option are specified in the **RCVALL_VALUE** enumeration defined in the *Mstcpip.h* header file.

The possible values for **SIO_RCVALL** are as follows:

Value	Meaning
RCVALL_OFF	Disable this option so a socket does not receive all IPv4 or IPv6 packets passing through a network interface.
RCVALL_ON	Enable this option so a socket receives all IPv4 or IPv6 packets passing through a network interface. This option enables promiscuous mode on the network interface card (NIC), if the NIC supports promiscuous mode. On a LAN segment with a network hub, a NIC that supports promiscuous mode will capture all IPv4 or IPv6 traffic on the LAN, including traffic between other computers on the same LAN segment. All of the captured packets (IPv4 or IPv6, depending on the socket) will be delivered to the raw socket. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) on the interface. Netmon uses the same mode for the network interface, but does not use this option to capture traffic.

RCVALL_SOCKETLEVELONLY	This feature is not currently implemented, so setting this option does not have any affect.
RCVALL_IPLEVEL	Enable this option so an IPv4 or IPv6 socket receives all packets at the IP level passing through a network interface. This option does not enable promiscuous mode on the network interface card. This option only affects packet processing at the IP level. The NIC still receives only packets directed to its configured unicast and multicast addresses. However, a socket with this option enabled will receive not only packets directed to specific IP addresses, but will receive all the IPv4 or IPv6 packets the NIC receives. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) received on the interface.

cbInBuffer

The size, in bytes, of the input buffer. This parameter should be equal to or greater than the size of the **RCVALL_VALUE** enumeration value pointed to by the *lpvInBuffer* parameter.

lpvOutBuffer

A pointer to the output buffer. This parameter is unused for this operation.

cbOutBuffer

The size, in bytes, of the output buffer. This parameter is unused for this operation.

lpcbBytesReturned

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer. This parameter is unused for this operation.

```
DWORD Optval = RCVALL_ON;
DWORD dwBytesReturned = 0;
int iResult;

iResult = WSAIoctl(SnifferSocket, SIO_RCVALL, &Optval, sizeof(Optval),NULL, NULL, &dwBytesReturned, NULL, NULL);
if (iResult == SOCKET_ERROR) {
  printf("WSAIoctl failed with error: %ld\n", WSAGetLastError());
  closesocket(SnifferSocket);
  WSACleanup();
}
```

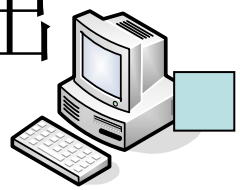
//dwBytesReturned虽然对SIO_RCVALL这个控制命令没用，但必须要有，否则WSAIoctl会执行错误

原始套接字输入

- 使用原始套接字接收数据是在无连接的方式下进行数据接收，原始套接字很可能接收到很多非预期的数据报

考虑数据过滤能力

原始套接字输出



- 发送数据的目标
  - 连接模式：在connect中指明
  - 非连接模式：在sendto中指明
- 发送数据的内容

a）IPv4标志字段可置0，告知内核设置
b）IPv4头部校验和字段置0，总是由内核计算并存储
c）IPv4选项字段可选

原始套接字输出

发送数据——sendto

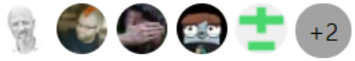
int sendto(SOCKET *s*, const char FAR* *buf*, int *len*, int *flags*, const struct sockaddr FAR* *to*, int *tolen*)

Buf的内容：

增加了与原始套接字定义相关的协议首部，如IP头、UDP头、TCP头、ICMP头等。

- *如果指明了IP_HDRINCL选项，则包括IP头；*
- *否则协议头部信息与socket函数protocol 字段相符。*

IPPROTO_IP socket options

03/23/2021 • 10 minutes to read • 

The following tables describe **IPPROTO_IP** socket options that apply to sockets created for the IPv4 address family (AF_INET). See the **getsockopt** and **setsockopt** function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the **WSAEnumProtocols**, **WSCEnumProtocols**, or **WSCEnumProtocols32** function.

Some socket options require more explanation than these tables can convey; such options contain links to additional pages.

Options

Option	Get	Set	Optval type	Description
IP_HDRINCL	yes	yes	DWORD (boolean)	When set to **TRUE**, indicates the application provides the IP header. Applies only to SOCK_RAW sockets. The TCP/IP service provider may set the ID field, if the value supplied by the application is zero. The IP_HDRINCL option is applied only to the SOCK_RAW type of protocol. A TCP/IP service provider that supports SOCK_RAW should also support IP_HDRINCL.

IPPROTO_IP级别的更多选项

Windows对原始套接字的限制

- 在Windows7、Windows Vista、Windows XP SP2和Windows XP SP3中，通过原始套接字发送数据的能力受到了诸多限制，这些限制主要包括：
  - TCP的数据不能通过原始套接字发送；
  - 如果源IP地址不正确，则UDP的数据不能通过原始套接字发送。所谓不正确是指为数据报设置的源地址不属于本机有效的IP地址，这样限制了恶意代码通过伪造源IP地址进行拒绝服务攻击的能力，同时还限制了发送IP欺骗数据包的能力；
  - 当设置协议类型为IPPROTO_TCP时，不允许将原始套接字绑定到本地地址（当协议类型为其它协议时，绑定是允许的）。
- Windows在Server版的操作系统中对原始套接字支持较好，上述限制不适用于Windows Server 2003、Windows Server 2008、以及Windows SP2以前的操作系统。

使用原始套接字实现ping

ICMP数据报头部

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

#pragma comment(lib,"Ws2_32.lib")

// IP 头部 -- RFC 791
typedef struct tagIPHDR
{
    u_char  VIHL;           // Version and IHL
    u_char  TOS;            // Type Of Service
    short   TotLen;         // Total Length
    short   ID;             // Identification
    short   FlagOff;        // Flags and Fragment Offset
    u_char  TTL;            // Time To Live
    u_char  Protocol;       // Protocol
    u_short Checksum;       // Checksum
    struct  in_addr iaSrc;  // Internet Address - Source
    struct  in_addr iaDst;  // Internet Address - Destination
}IPHDR, * PIPHDR;

// ICMP 头部 - RFC 792
typedef struct tagICMPHDR
{
    u_char  Type;           // Type
    u_char  Code;           // Code
    u_short Checksum;       // Checksum
    u_short ID;             // Identification
    u_short Seq;            // Sequence
}ICMPHDR, * PICMPHDR;
```

```
#define REQ_DATASIZE 32     // Echo Request Data size
#define ICMP_ECHOREPLY  0  //ICMP类型码0: 回声应答
#define ICMP_ECHOREQ    8  //ICMP类型码8: 回声请求

// ICMP Echo 请求报文
typedef struct tagECHOREQUEST
{
    ICMPHDR icmpHdr;
    DWORD   dwTime;
    char    cData[REQ_DATASIZE];
}ECHOREQUEST, * PECHOREQUEST;

// ICMP Echo 应答报文
typedef struct tagECHOREPLY
{
    IPHDR   ipHdr;
    ECHOREQUEST echoRequest;
    char    cFiller[256];
}ECHOREPLY, * PECHOREPLY;


void Ping(LPCSTR pstrHost);
int SendEchoRequest(SOCKET s, LPSOCKADDR_IN lpstToAddr);
u_short in_cksum(u_short* addr, int len);
int WaitForEchoReply(SOCKET s);
DWORD RecvEchoReply(SOCKET s, LPSOCKADDR_IN lpsaFrom, u_char* pTTL);
```

```c
int main(int argc, char** argv)
{
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD(2, 2);
    int nRet;

    //检查输入合法性
    if (argc != 2){
        fprintf(stderr, "\nUsage: ping hostname\n");
        return 1;
    }
    //初始化套接字
    nRet = WSAStartup(wVersionRequested, &wsaData);
    if (nRet){
        fprintf(stderr, "\nError initializing WinSock\n");
        return 1;
    }
    // 核对套接字版本
    if (wsaData.wVersion != wVersionRequested){
        fprintf(stderr, "\nWinSock version not supported\n");
        return 1;
    }
    //调用Ping函数完成具体功能
    Ping(argv[1]);
    // 释放套接字
    WSACleanup();
}
```

```
void Ping(LPCSTR pstrHost) {
    SOCKET    rawSocket;
    int iResult;
    struct addrinfo* pHostAddrInfo = NULL, hints;
    DWORD     dwTimeSent;
    DWORD     dwElapsed;
    u_char    cTTL;
    int       nLoop;
    int       nRet;

    //创建原始套接字
    rawSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (rawSocket == SOCKET_ERROR)
    {
        printf("套接字创建错误，错误码: %d\n",  WSAGetLastError());
        return;
    }
    //查找目标主机地址信息
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_RAW;
    hints.ai_protocol = IPPROTO_ICMP;
    iResult = getaddrinfo(pstrHost, 0, &hints, &pHostAddrInfo);
    if (iResult != 0) {
        printf("getaddrinfo执行错误，错误码: %d\n", iResult);
        return;
    }
    //输出提示信息
    char strIP[16]; ;
    printf("\nPinging %s [%s] with %d bytes of data:\n",
        pstrHost,
        inet_ntop(AF_INET, &(((sockaddr_in*)(pHostAddrInfo->ai_addr))->sin_addr), strIP, 16),
        REQ_DATASIZE);
```

```
    //多次Ping操作
    for (nLoop = 0; nLoop < 4; nLoop++)
    {
        // 发送 ICMP echo 请求报文
        SendEchoRequest(rawSocket, (sockaddr_in*)(pHostAddrInfo->ai_addr));

        // 使用select模型来等待ICMP回声应答,选select模型是因为我们需要超时判断
        nRet = WaitForEchoReply(rawSocket);
        if (nRet == SOCKET_ERROR)
        {
            printf("select()执行出错，错误码:%d\n", WSAGetLastError());
            break;
        }
        if (!nRet)
        {
            printf("\nTimeOut");
            break;
        }

        // Receive reply
        dwTimeSent = RecvEchoReply(rawSocket, (sockaddr_in*)(pHostAddrInfo->ai_addr), &cTTL);

        // Calculate elapsed time
        dwElapsed = GetTickCount() - dwTimeSent;
        printf("\nReply from: %s: bytes=%d time=%ldms TTL=%d",
            inet_ntop(AF_INET, &(((sockaddr_in*)(pHostAddrInfo->ai_addr))->sin_addr), strIP, 16),
            REQ_DATASIZE,
            dwElapsed,
            cTTL);
    }
    printf("\n");
    freeaddrinfo(pHostAddrInfo);
    nRet = closesocket(rawSocket);
    if (nRet == SOCKET_ERROR)
        printf("closesocket()执行出错，错误码: %d\n", WSAGetLastError());
}
```

```
//构造并发送ICMP ECHO请求数据包
int SendEchoRequest(SOCKET s, LPSOCKADDR_IN lpstToAddr)
{
    static ECHOREQUEST echoReq;
    static u_short nId = 1;
    static u_short nSeq = 1;
    int nRet;

    // 填充ICMP请求报头
    echoReq.icmpHdr.Type = ICMP_ECHOREQ;
    echoReq.icmpHdr.Code = 0;
    echoReq.icmpHdr.Checksum = 0;
    echoReq.icmpHdr.ID = nId++;
    echoReq.icmpHdr.Seq = nSeq++;

    // 填充数据
    for (nRet = 0; nRet < REQ_DATASIZE; nRet++)
        echoReq.cData[nRet] = ' ' + nRet;

    // 存储时间戳
    echoReq.dwTime = GetTickCount();

    //计算校验和字段
    echoReq.icmpHdr.Checksum = in_cksum((u_short*)&echoReq, sizeof(ECHOREQUEST));

    //发送回声请求
    nRet = sendto(s,                        /* socket */
        (LPSTR)&echoReq,            /* buffer */
        sizeof(ECHOREQUEST),
        0,                          /* flags */
        (LPSOCKADDR)lpstToAddr, /* destination */
        sizeof(SOCKADDR_IN));   /* address length */

    if (nRet == SOCKET_ERROR)
        printf("sendto()出错，错误码:%d\n", WSAGetLastError());
    return (nRet);
}
```

```c
//等待回声应答的到来
int WaitForEchoReply(SOCKET s)
{
    struct timeval Timeout;
    fd_set readfds;
    //设置读等待套接字组
    readfds.fd_count = 1; //套接字数量
    readfds.fd_array[0] = s;
    Timeout.tv_sec = 5;  //秒
    Timeout.tv_usec = 0;  //微秒
    //等待套接字上的网络事件,返回发生网络事件的套接字数量的总和，超时返回0，失败返回SOCKET_ERROR
    return(select(1, &readfds, NULL, NULL, &Timeout));
}
```

```
//接收回声应答报文
DWORD RecvEchoReply(SOCKET s, LPSOCKADDR_IN lpsaFrom, u_char* pTTL)
{
    ECHOREPLY echoReply;
    int nRet;
    int nAddrLen = sizeof(struct sockaddr_in);

    // Receive the echo reply
    nRet = recvfrom(s,                  // socket
        (LPSTR)&echoReply,  // buffer
        sizeof(ECHOREPLY),  // size of buffer
        0,                  // flags
        (LPSOCKADDR)lpsaFrom,   // From address
        &nAddrLen);         // pointer to address len

// Check return value
    if (nRet == SOCKET_ERROR)
        printf("recvfrom()出错，错误码：%d\n", WSAGetLastError());

    // return time sent and IP TTL
    *pTTL = echoReply.ipHdr.TTL;
    return(echoReply.echoRequest.dwTime);
}
```

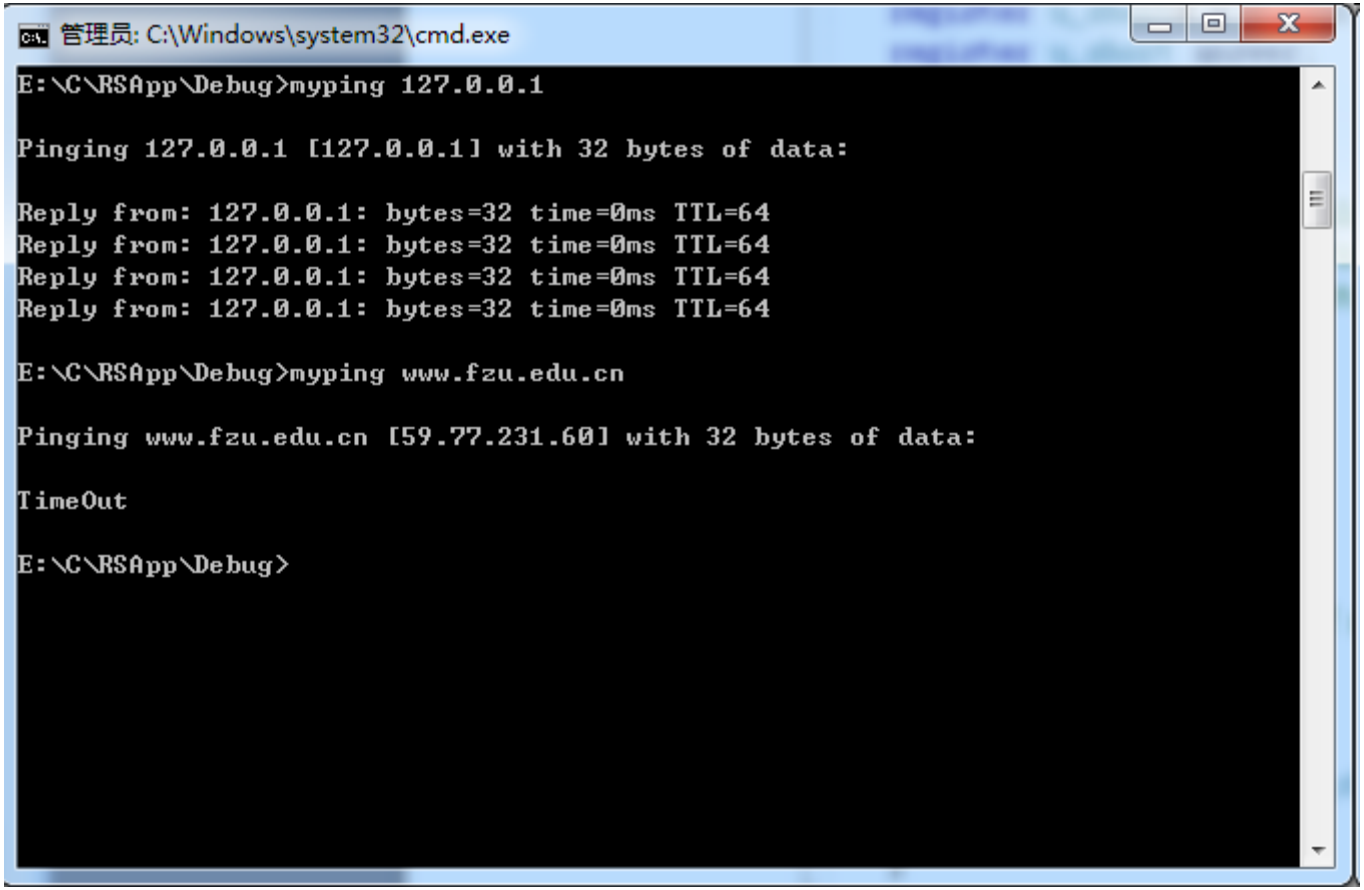
```c
//计算校验和
u_short in_cksum(u_short* addr, int len)
{
    register int nleft = len;
    register u_short* w = addr;
    register u_short answer;
    register int sum = 0;

    /*
     *  Our algorithm is simple, using a 32 bit accumulator (sum),
     *  we add sequential 16 bit words to it, and at the end, fold
     *  back all the carry bits from the top 16 bits into the lower
     *  16 bits.
     */
    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    /* mop up an odd byte, if necessary */
    if (nleft == 1) {
        u_short u = 0;

        *(u_char*)(&u) = *(u_char*)w;
        sum += u;
    }

    /*
     * add back carry outs from top 16 bits to low 16 bits
     */
    sum = (sum >> 16) + (sum & 0xffff); /* add hi 16 to low 16 */
    sum += (sum >> 16);         /* add carry */
    answer = ~sum;              /* truncate to 16 bits */
    return (answer);
}
```


管理员: C:\Windows\system32\cmd.exe
E:\C\RSApp\Debug>myping 127.0.0.1
Pinging 127.0.0.1 [127.0.0.1] with 32 bytes of data:
Reply from: 127.0.0.1: bytes=32 time=0ms TTL=64
Reply from: 127.0.0.1: bytes=32 time=0ms TTL=64
Reply from: 127.0.0.1: bytes=32 time=0ms TTL=64
Reply from: 127.0.0.1: bytes=32 time=0ms TTL=64
E:\C\RSApp\Debug>myping www.fzu.edu.cn
Pinging www.fzu.edu.cn [59.77.231.60] with 32 bytes of data:
TimeOut
E:\C\RSApp\Debug>

数据包捕获

```
#include <stdio.h>
#include <tchar.h>
#include <winsock2.h>
#include <mstcpip.h>
#include <ws2tcpip.h>

#pragma comment(lib,"ws2_32.lib")

//定义默认的缓冲区长度和端口号
#define DEFAULT_BUFLEN 65535
#define DEFAULT_NAMELEN 512

int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA wsaData;
    SOCKET SnifferSocket = INVALID_SOCKET;
    char recvbuf[DEFAULT_BUFLEN];
    int iResult;
    int recvbuflen = DEFAULT_BUFLEN;
    struct addrinfo* local = NULL, *ptrAddrTemp=NULL,hints;
    char HostName[DEFAULT_NAMELEN];
    struct in_addr addr;
    struct sockaddr_in RemoteAddr;
    int addrlen = sizeof(struct sockaddr_in);
    int in = 0, i = 0;
    DWORD Optval = RCVALL_ON;
    DWORD dwBytesReturned = 0;
    char strIP[16];
```

```
// 初始化套接字
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}

//创建原始套接字
printf("\n创建原始套接字...");
SnifferSocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
if (SnifferSocket == INVALID_SOCKET) {
    printf("socket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

//获取本机名称
memset(HostName, 0, DEFAULT_NAMELEN);
iResult = gethostname(HostName, sizeof(HostName));
if (iResult == SOCKET_ERROR) {
    printf("gethostname failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
```

```
//获取本机可用IP
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_RAW;
hints.ai_protocol = IPPROTO_IP;
iResult = getaddrinfo(HostName, 0, &hints, &local);
if (iResult != 0) {
    printf("getaddrinfo执行错误，错误码：%d\n", iResult);
    return 1;
}
printf("\n本机可用的IP地址为：\n");
if (local == NULL)
{
    printf("gethostbyname failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
ptrAddrTemp = local;
while (ptrAddrTemp != NULL) {
    addr = ((sockaddr_in*)(ptrAddrTemp->ai_addr))->sin_addr;
    printf("\tIP Address #%d: %s\n", i, inet_ntop(AF_INET, &addr, strIP, 16));
    i++;
    ptrAddrTemp = ptrAddrTemp->ai_next;
}

printf("\n请选择捕获数据待使用的接口号：");
scanf_s("%d", &in);

i = 0;
ptrAddrTemp = local;
while (ptrAddrTemp != NULL && i != in) {
    i++;
    ptrAddrTemp = ptrAddrTemp->ai_next;
}
```

```
    //绑定本地地址
    iResult = bind(SnifferSocket, ptrAddrTemp->ai_addr, ptrAddrTemp->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %ld\n", WSAGetLastError());
        closesocket(SnifferSocket);
        WSACleanup();
        return 1;
    }
    printf(" \n成功绑定套接字和#%d号接口地址", in);
    freeaddrinfo(local);

    //设置套接字接收命令
    //dwBytesReturned虽然对SIO_RCVALL这个控制命令没用，但必须要有，否则WSAIoctl会执行错误
    iResult = WSAIoctl(SnifferSocket, SIO_RCVALL, &Optval, sizeof(Optval),NULL, NULL, &dwBytesReturned, NULL, NULL);
    if (iResult == SOCKET_ERROR) {
        printf("WSAIoctl failed with error: %ld\n", WSAGetLastError());
        closesocket(SnifferSocket);
        WSACleanup();
    }

    //开始接收数据
    printf(" \n开始接收数据");
    do
    {
        //接收数据
        iResult = recvfrom(SnifferSocket, recvbuf, DEFAULT_BUFLEN, 0, (struct sockaddr*) & RemoteAddr, &addrlen);
        if (iResult > 0)
            printf("\n接收到来自%s的数据包，长度为%d.", inet_ntop(AF_INET, &(RemoteAddr.sin_addr), strIP, 16), iResult);
        else
            printf("recvfrom failed with error: %ld\n", WSAGetLastError());
    } while (iResult > 0);
    return 0;
}
```


Microsoft Visual Studio 调试控制台
创建原始套接字...
本机可用的IP地址为：
IP Address #0: 192.168.1.2
IP Address #1: 192.168.230.1
IP Address #2: 169.254.191.61
请选择捕获数据待使用的接口号：0
成功绑定套接字和#0号接口地址
开始接收数据
接收到来自192.168.1.2的数据包，长度为1225.
接收到来自192.168.1.2的数据包，长度为40.
接收到来自192.168.1.2的数据包，长度为444.
接收到来自192.168.1.2的数据包，长度为40.
接收到来自192.168.1.2的数据包，长度为52.
接收到来自192.168.1.2的数据包，长度为40.
接收到来自192.168.1.2的数据包，长度为121.
接收到来自192.168.1.2的数据包，长度为61.
接收到来自192.168.1.2的数据包，长度为40.
E:\C\RSApp\Debug\MySniff.exe (进程 7516)已退出，返回代码为：-1073741510。
按任意键关闭此窗口...