实验报告：简单路由器程序的设计

计算机学院 2010519 卢麒萱

一、实验内容说明

本次实验为设计性实验：简单路由器程序的设计，其具体要求如下：

1. 设计和实现一个路由器程序，要求完成的路由器程序能和现有的路由器产品（如思科路由器、华为路由器、微软的路由器等）进行协同工作。
2. 程序可以仅实现`IP`数据报的获取、选路、投递等路由器要求的基本功能。可以忽略分片处理、选项处理、动态路由表生成等功能。
3. 需要给出路由表的手工插入、删除方法。
4. 需要给出路由器的工作日志，显示数据报获取和转发过程。

二、实验准备

在使用`Winpcap`之前，首先需要安装`WinPcap`驱动程序和`DLL`程序。可以从https://www.winpcap.org/install/default.htm网站下载适合自己电脑的版本并安装。然后需要从https://www.winpcap.org/devel.htm网站下载开发者工具包（`Developer's pack`），该工具包中包含了开发基于`WinPcap`所需要的库文件、包含文件等，需要解压到自己的电脑中，记录下解压后文件夹的目录。

下面就`WinPcap`编程中可能会用到的函数以及数据结构做简要介绍。

1、`pcap_findalldevs_ex()`函数

获取网络接口设备（网卡）列表，其原型如下：

```
int pcap_findalldevs_ex
(
    char *  source,
    struct pcap_rmtauth *  auth,
    pcap_if_t **   alldevs,
    char *  errbuf
)
```

参数说明：

- `source`：适配器文件所在位置，指定从哪里获取网络接口列表，取值有：
    1. `'rpcap://'`：表示本地的适配器，此时可以用宏定义`PCAP_SRC_IF_STRING`。
    2. `'rpcap://hostname:prot'`：表示主机名称为`hostname`并且端口号为`port`，如本地的`hostname`为`localhost`,端口号一般为`80`。
    3. `'file://c:/myfolder/'`：指定路径。
- `auth`：指向`pcap_rmtauth`结构体,当连接到远程`host`时，需要它保存一些信息。对于本地连接时没有意义，一般取`NULL`。
- `alldevs`：利用结构体`pcap_if`存储适配器信息，并保存在链表结构的`alldevs`中。
- `errbuf`：保存错误信息的缓冲区。

`return`：该函数调用成功时返回0，这时`alldevs`参数指向网络接口链表的第一个元素；调用失败时返回-1，具体的错误信息保存在`errbuf`中。

2、`pcap_freealldevs()`函数

释放设备列表，其原型如下：

```
void pcap_freealldevs(pcap_if_t *alldevsp);
```

参数说明：

- `alldevsp`：指向需要释放的设备链表的第一个元素，通常由`pcap_findalldevs_ex()`函数返回。

3、`pcap_open()`函数

打开某一网络接口设备，其原型如下：

```
pcap_t* pcap_open(
    const char *source,
    int snaplen,
    int flags,
    int read_timeout,
    struct pcap_rmtauth *auth,
    char *errbuf
);
```

参数说明：

- `source`：我们所要打开的网络接口卡的名字。当我们获取所有的设备之后，这个`source`就是`d->name`。如果是文件的话，就是文件的名字，也是`d->name`。
- `snaplen`：我们抓取的数据包的最大长度，100就是抓取整个数据包的前100bytes，65536最大，把整个包都包括了。
- `flags`：设置网络设备打开的状态，最常用的是`PCAP_OPENFLAG_PROMISCUOUS`，表示这个网络设备以混杂模式打开，可以捕捉局域网中所有数据包。
- `read_timeout`：设置延迟时间(`milliseconds`)。捕捉数据包的时候，延迟一定的时间，然后再调用内核中的程序，这样效率较高。0表示没有延迟，没有包到达的时候永不返回。-1表示立即返回。
- `auth`：远程机器的登录信息。本地机器则为`NULL`。
- `errbuf`：存储出错信息的缓冲区。

`return`：返回这个打开设备的描述符，如果出错，返回`NULL`，具体的错误信息保存在`errbuf`中。

4、`pcap_sendpacket()`函数

```
int pcap_sendpacket(
    pcap_t *p,
    u_char buf,
    int size
);
```

参数说明:

- `p`：指定`pcap_sendpacket()`函数通过哪块接口网卡发送数据包。该参数为一个指向`pcap_t`结构的指针，通常是调用`pcap_open()`函数成功后返回的值。
- `buf`：指向需要发送的数据包，该数据包应该包括各层的头部信息。值得注意的是，以太网帧的`CRC`校验和字段不应该包含在`buf`中，`WinPcap`在发送过程中会自动为其增加校验和。
- `size`：指定发送数据包的大小。

`return`：发送成功时，`pcap_sendpacket()`函数返回0，否则返回-1。

5、`pcap_next_ex()`函数

不使用回调函数捕获网络数据包，其函数原型如下：

```
int pcap_next_ex(
    pcap_t* p,
    struct pcap_pkthdr **pkt_header,
    u_char ** pkt_data
);
```

参数说明：

- `p`：指定捕获哪块网卡上的网络数据包。为一个指向`pcap_t`结构的指针，通常是调用`pcap_open()`函数成功后返回的值。
- `pkt_header`：报文头。在`pcap_next_ex()`函数调用成功后，该参数指向的`pcap_pkthdr`结构保存所捕获的网络数据包的一些基本信息。
- `pkt_data`：报文内容。在`pcap_next_ex()`函数调用成功后，指向捕获到的网络数据包。

`return`：返回1表示该函数正确捕获到一个数据包，这时，`pkt_header`保存捕获数据包的一些基本信息，`pkt_data`指向捕获数据包的完整数据；返回0表示该函数在指定时间范围（`read_timeout`）内没有捕获到任何网络数据包；返回-1表示该函数在调用过程中发生错误；返回-2表示获取到离线记录文件的最后一个报文。

6、`pcap_compile()`函数

将str参数指定的字符串编译到过滤程序中。

```
int pcap_compile(
    pcap_t *p,
    struct bpf_program *fp,
    char *str,
    int optimize,
    bpf_u_int32 netmask
);
```

参数说明：

- `p`：表示`pcap`会话句柄。
- `fp`：一个`bpf_program`结构的指针，在`pcap_compile()`函数中被赋值，存放编译以后的规则。
- `str`：规则表达式格式的过滤规则（`filter`）。
- `optimize`：控制结果代码的优化。
- `netmask`：监听接口的网络掩码。

`return`：返回-1表示操作失败，其他值表示成功。

7、`pcap_setfilter()`函数

```
int pcap_setfilter(
    pcap_t *p,
    struct bpf_program *fp
);
```

- `p`：表示`pcap`的会话句柄。
- `fp`：表示经过编译后的过滤规则。

`return`：返回-1表示操作失败，其他值表示成功。

8、`pcap_if_t`结构

`alldevs`指向的网络接口链表中元素的类型，其定义如下：

```
Typedef struct pcap_if pcap_if_t;
struct pcap_if {
    struct pcap_if *next;
    char *name;
    char *description;
    struct pcap_addr *addresse;
    u_int flags;
};
```

参数说明：

- `next`：指向链表中的下一个元素。最后一个元素的`next`为`NULL`。
- `name`：指向该网卡名称。
- `description`：该网卡描述内容。
- `address`：指向包含这块网卡拥有的所有`IP`地址的地址链表。
- `flags`：标识该网络接口卡是不是一块回送网卡，是的话为`PCAP_IF_LOOKBACK`。

9、`pcap_addr_t`结构

网络接口链表中元素的`addresse`属性的类型，其定义如下：

```
Typedef struct pcap_addr pcap_addr_t;
struct pcap_addr {
      struct pcap_addr *next;
      struct sockaddr *addr;
      struct sockaddr *netmask;
      struct sockaddr *broadaddr;
      struct sockaddr *dstaddr;
};
```

参数说明：

- `next`：指向下一个元素的指针。
- `addr`：`IP`地址。
- `netmask`：网络掩码。
- `addr`：指向该网卡名称。
- `netmask`：该网卡描述内容。
- `broadaddr`：广播地址。
- `dstaddr`：P2P目的地址。
- `flags`：标识该网络接口卡是不是一块回送网卡，是的话为`PCAP_IF_LOOKBACK`。

10、`sockaddr`结构、`sockaddr_in`结构和`in_addr`结构

`sockaddr`结构是上述提到的`pcap_addr_t`结构中的数据类型，`sockaddr_in`和`sockaddr`是并列的结构，指向`sockaddr_in`的结构体的指针也可以指向`sockaddr`的结构体，并代替它，即可以使用`sockaddr_in`建立所需要的信息，然后进行类型的强制转换。`in_addr`结构是`sockaddr_in`结构中`sin_addr`属性的数据类型。

```
struct sockaddr {
```

```
        u_short sa_family;
        char    sa_data[14];
};

struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char    sin_zero[8];
};

struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr
#define s_host  S_un.S_un_b.s_b2
#define s_net   S_un.S_un_b.s_b1
#define s_imp   S_un.S_un_w.s_w2
#define s_impno S_un.S_un_b.s_b4
#define s_lh    S_un.S_un_b.s_b3
};
```

- `sockaddr` 是内核用来储存地址的结构。`sa_family` 指向一个地址族，本实验中用来判断该地址是否为 `IP` 地址。该结构用不同的 `unsigned short` 数来表示不同的地址协议，`AF_INET` 被定义为 2，代表 `TCP/IP` 协议族。`sa_data` 数组存储地址，本实验中不使用。
- `sockaddr_in` 是一个指向 `socket` 地址和网络类型的结构。`sin_family` 指代协议族，在 `socket` 编程中只能是 `AF_INET`，`sin_port` 存储端口号（使用网络字节顺序），`sin_addr` 存储 `IP` 地址，使用 `in_addr` 这个数据结构，`sin_zero` 是为了让 `sockaddr` 与 `sockaddr_in` 两个数据结构保持大小相同而保留的空字节。
- `in_addr` 是一个用来储存 `IP` 地址的结构，本实验中使用该结构中定义的联合中的一个 `unsigned long` 变量 `s_addr` 来实际存储 `IP` 地址。

三、实验步骤

1、项目设计思路

简单路由处理软件可以仅接收需要转发的 `IP` 数据报，这些 `IP` 数据报的共同特点是目的 `MAC` 地址指向本机但目的 `IP` 地址不属于本机的 `IP` 地址。由于 `WinPcap` 提供的包过滤机制很高，因此，可以利用 `WinPcap` 的包过滤机制筛选出需要处理的 `IP` 数据报提交给简单路由处理程序。

按照 `IP` 路由选择算法，在利用 `WinPcap` 获取到需要转发的 `IP` 数据报后，路由处理软件首先需要提取该报文的目的 `IP` 地址，并通过路由表为其进行路由选择。如果路由选择成功，则记录需要投递到的下一路由地址；如果不成功，则简单地将该报文抛弃。

在将路由选择成功发 `IP` 数据报发送到相应的接口之前，首先需要利用 `ARP` 获取下一站路由器接口的 `MAC` 地址。一旦得到下一站路由器的 `MAC` 地址，就可以把 `IP` 数据报封装成数据帧并通过相应的接口发送出去。

2、具体流程及关键代码分析

1、创建基于对话框的 `MFC` 程序

1. 启动 `Visual Studio 2010`；

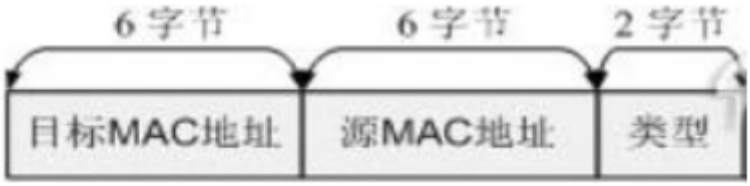
2. 新建一个基于对话框的`MFC`项目；
3. 项目->属性->`VC++`目录->包含目录，添加`WpdPack`目录下的`Include`目录；
4. 项目->属性->`VC++`目录->库目录，添加`WpdPack`目录下的`lib`目录；
5. 项目->属性->`C/C++`->预处理器->预处理器定义，添加`HAVE_REMOTE;WPCAP;`；
6. 项目->属性->链接器->输入->附加依赖项，添加`Packet.lib;wpcap.lib`；
7. 绘制对话框界面，添加相应控件；

2、相关数据结构定义

网络中传输的数据包是经过封装的，每一次封装都会增加相应的首部。由于`WinPcap`在数据链路层捕获数据包，因此，在以太网中传送的数据都包含以太网帧头信息。同时，由于我们捕获到的数据包保存在一个无结构的缓冲区中，因此，在实际编程过程中通常需要定义一些有关首部的数据结构。通过将这些结构赋予存放数据包的无结构缓冲区，简化数据的提取过程。值得注意的是，`WinPcap`捕获到的数据包在缓冲区中是连续存放的，但通常`VC++`默认`IDE`的设置并不是字节对齐的。因此，定义这些包首部数据结构时，需要使用`#pragma pack(1)`语句通知生成程序按照字节对齐方式生成下面的数据结构。在这些数据结构定义完成后，使用`#pragma pack()`恢复默认对齐方式。

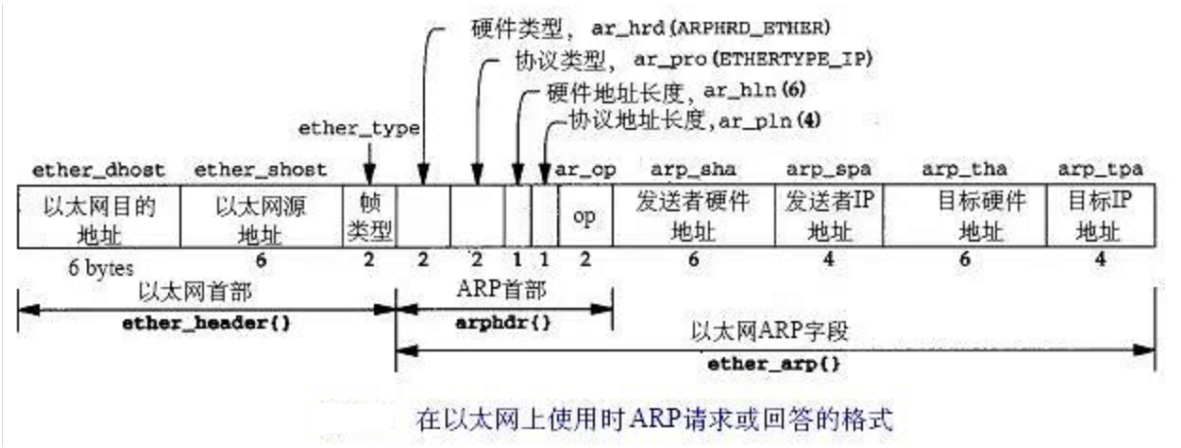
```
#pragma pack(1)//进入字节对齐方式
.....//数据结构定义
#pragma pack()//恢复默认对齐方式
```

1、帧头部




```
//帧头部结构体 共14字节
typedef struct FrameHeader_t {
    UCHAR DesMAC[6]; // 目的MAC地址 6字节
    UCHAR SrcMAC[6]; // 源MAC地址 6字节
    USHORT FrameType; // 上一层协议类型 2字节
} FrameHeader_t;
```

2、`ARP`帧



根据这个结构图我们可以作如下定义：

```
typedef struct ARPFrame_t { // ARP帧
    FrameHeader_t FrameHeader; // 帧首部
    WORD HardwareType; // 硬件类型
    WORD ProtocolType; // 协议类型
    BYTE HLen; // 硬件地址长度
    BYTE PLen; // 协议地址长度
    WORD Operation; // 操作值
    UCHAR SendHa[6]; // 源MAC地址
    ULONG SendIP; // 源IP地址
    UCHAR RecvHa[6]; // 目的MAC地址
    ULONG RecvIP; // 目的IP地址
} ARPFrame_t;
```

3、IP首部

IP协议首部的结构图为：

根据IP协议首部的结构可以作如下定义，其中，源IP地址和目的IP地址的数据类型均为上面定义的ip_address类型。

```
typedef struct IPHeader_t { // IP首部
    BYTE Ver_HLen; // 版本+头部长度
    BYTE TOS; // 服务类型
    WORD TotalLen; // 总长度
    WORD ID; // 标识
    WORD Flag_Segment; // 标志+片偏移
    BYTE TTL; // 生存时间
    BYTE Protocol; // 协议
    WORD Checksum; // 头部校验和
    ULONG SrcIP; // 源IP地址
    ULONG DstIP; // 目的IP地址
} IPHeader_t;
```

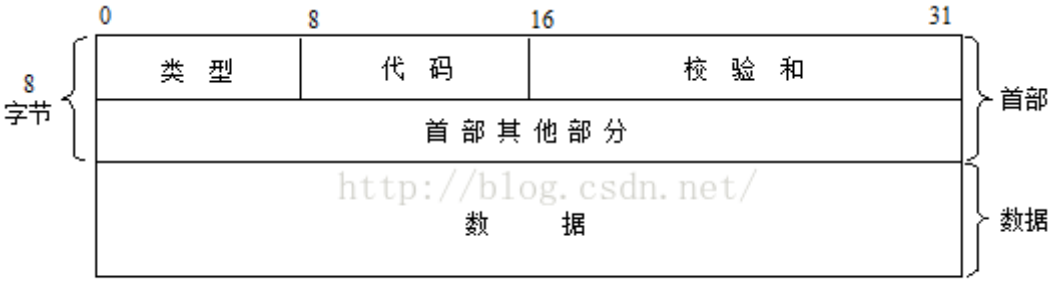
4、包含帧首部和IP首部的数据包

将上面定义的`FrameHeader_t`和`IPHeader_t`两个结构合起来即可得到。

```
typedef struct IPFrame_t { // IP帧
    FrameHeader_t FrameHeader; // 帧首部
    IPHeader_t IPHeader; // IP首部
} IPFrame_t;
```

5、`ICMP`首部

`ICMP`协议首部的结构图为：



ICMP 报文格式

根据`IP`协议首部的结构可以作如下定义：

```
typedef struct ICMPHeader_t { // ICMP首部
    BYTE Type; // 类型
    BYTE Code; // 代码
    WORD Checksum; // 校验和
    WORD Id; // 标识
    WORD Sequence; // 序列号
} ICMPHeader_t;
```

6、网络地址

我们知道，一个网络可以由IP地址与子网掩码相与得到，因此作如下定义：

```
typedef struct ip_t { // 网络地址
    ULONG IPAddr; // IP地址
    ULONG IPMask; // 子网掩码
} ip_t;
```

7、接口信息

以下数据结构定义了一个接口信息，包括：设备名、设备描述、`MAC`地址、`IP`地址列表和`pcap`句柄。由于一块网卡上可能绑定了不止一个`IP`，因此需要用一个列表来保存它的`IP`信息。

```
typedef struct  IfInfo_t { // 接口信息
    char* DeviceName; // 设备名
    CString Description; // 设备描述
    UCHAR MACAddr[6]; // MAC地址
    CArray <ip_t,ip_t&> ip; // IP地址列表
    pcap_t *adhandle; // pcap句柄
} IfInfo_t;
```

8、发送数据包结构

```
typedef struct SendPacket_t { // 发送数据包结构
    int len; // 长度
    BYTE PktData[2000]; // 数据缓存
    ULONG TargetIP; // 目的IP地址
    UINT_PTR n_mTimer; // 定时器
    UINT IfNo; // 接口序号
} SendPacket_t;
```

9、路由表项结构

路由表由一个个路由表项构成，每个路由表项包括了子网掩码、目的地址和下一跳步这三个信息，同时还记录了接口序号。

```
typedef struct RouteTable_t { // 路由表项结构
    ULONG Mask; // 子网掩码
    ULONG DstIP; // 目的地址
    ULONG NextHop; // 下一跳步
    UINT IfNo; // 接口序号
} RouteTable_t;
```

10、`IP-MAC`地址映射结构

路由器中保存了一个`IP-MAC`地址映射表，它保存了`IP`地址和`MAC`地址的映射关系，因为在数据包转发的过程中，要通过`IP`地址获得`MAC`地址，修改数据包首部之后再进行转发。

```
typedef struct IP_MAC_t { // IP-MAC地址映射结构
    ULONG IPAddr; // IP地址
    UCHAR MACAddr[6]; // MAC地址
} IP_MAC_t;
```

3、全局变量

全局变量包括：接口信息数组、接口个数、定时器个数，同时还有`CList`类型的发送数据包缓存队列、`IP-MAC`地址映射列表以及路由表。其中，发送数据包缓存队列用来保存因没有获得`MAC`地址而暂时发送不出去的数据包；`IP-MAC`地址映射列表保存`IP`和`MAC`地址的映射信息，路由器将从中得到`MAC`地址并进行转发；路由表就是一个路由表项的列表。同时，还要申请一个互斥锁，以保证多个线程对统一资源的互斥访问。

```
// ------------------全局变量------------------
IfInfo_t IfInfo[MAX_IF]; // 接口信息数组
int IfCount; // 接口个数
UINT_PTR TimerCount; // 定时器个数

CList<SendPacket_t, SendPacket_t&> SP; // 发送数据包缓存队列
CList<IP_MAC_t, IP_MAC_t&> IP_MAC; // IP-MAC地址映射列表
CList<RouteTable_t, RouteTable_t&> RouteTable; // 路由表

CMutex mMutex(0,0,0); //互斥
```

4、获取本机设备列表

在初始化对话框函数`OnInitDialog()`中通过调用`pcap_findalldevs_ex()`函数获取本机的设备列表，如果获取失败，弹出对话框打印失败信息。然后将设备列表显示在`listbox`控件上。

```
// 获得本机的设备列表
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL /*无需认证*/, &m_alldevs,
errbuf) == -1)
{
    // 错误，返回错误信息
    sprintf_s(strbuf, "pcap_findalldevs_ex错误: %s", errbuf);
    PostMessage(WM_QUIT, 0, 0);
}

for(m_selectdevs = m_alldevs; m_selectdevs != NULL; m_selectdevs= m_selectdevs-
>next) // 显示接口列表
    m_dev.AddString(CString(m_selectdevs->name)); // 利用d->name获取该网络接口设备的
名字

    m_dev.SetCurSel(0);
    m_dev.SetHorizontalExtent(1600);
    m_log.SetHorizontalExtent(1600);
```

5、用户选择并打开设备

首先获取`listbox`被选中的行的数目，然后遍历获取到被选中的接口，通过接口的`addresses`信息分别获取到其`IP`地址和掩码地址，这里要先进行类型转换，转换为`struct sockaddr_in`类型的指针名，然后通过其`sin_addr.s_addr`属性获取地址。同时，由于网卡上可能绑定了不止一个`IP`，因此要用一个循环获取到这块网卡上绑定的所有`IP`地址信息。在获取到`IP`地址信息后，检查它的`IP`地址的个数，由于本次实验要求路由器必须至少有两个`IP`，因此若不符合路由器`IP`地址数目要求则弹出对话框打印错误信息。接下来，使用`pcap_open`函数打开设备，如果其返回值为`NULL`，表示打开错误，输出错误信息，直接释放设备列表，返回；否则，设备已正确打开，可以进行数据包的发送和捕获了。

```
// 选择接口
int N = m_dev.GetCurSel(); // 获取listbox被选中的行的数目
m_selectdevs = m_alldevs;
while(N--)
    m_selectdevs = m_selectdevs->next;

IfInfo[0].DeviceName = m_selectdevs->name;
IfInfo[0].Description = m_selectdevs->description;
for(a = m_selectdevs->addresses; a; a = a->next)
{
    if (a->addr->sa_family == AF_INET)
    {
        ipaddr.IPAddr = (((struct sockaddr_in *)a->addr)->sin_addr.s_addr);
        ipaddr.IPMask = (((struct sockaddr_in *)a->netmask)->sin_addr.s_addr);
        IfInfo[0].ip.Add(ipaddr);
        j++;
    }
}

// 不符合路由器IP地址数目要求
if (j < 2)
{
    MessageBox(L"该路由程序要求本地主机至少应具有2个IP地址");
    GetDlgItem(IDC_BUTTON_START)->EnableWindow(TRUE);
    m_dev.EnableWindow(TRUE);
    return;
}
```

```
// 打开接口
if ( (IfInfo[i].adhandle = pcap_open(IfInfo[i].DeviceName, // 设备名
                                     65536,// 最大包长度
                                     PCAP_OPENFLAG_PROMISCUOUS,// 混杂模式
                                     1000,// 超时时间
                                     NULL,// 远程认证
                                     errbuf// 错误缓存
                                        ) ) == NULL)
{
    // 错误，显示错误信息
    sprintf_s(strbuf, "接口未能打开。WinPcap不支持%s。", IfInfo[i].DeviceName);
    // 释放设备列表
    pcap_freealldevs(m_alldevs);
    return;
}
```

6、获取网卡MAC地址

首先开启数据包捕获线程，然后将列表中网卡硬件地址清0，以便为后续的赋值做准备，然后为得到真实网卡地址，使用虚假的`MAC`地址和`IP`地址向本机发送`ARP`请求，此处设置虚假的`MAC`地址为66.66.66.66.66.66，设置虚假的`IP`地址为112.112.112.112，然后发送`ARP`数据包请求对应的`MAC`地址。

```
// 开启数据包捕获线程，获取本地接口的MAC地址，线程数目为网卡个数
CWinThread* pthread;
for (i = 0; i < IfCount; i++)
{
    pthread = AfxBeginThread(CaptureLocalARP, &IfInfo[i],
THREAD_PRIORITY_NORMAL);
    if(!pthread)
    {
        MessageBox(L"创建数据包捕获线程失败！");
        PostMessage(WM_QUIT, 0, 0);
    }
}
// 将列表中网卡硬件地址清0
for (i = 0; i < IfCount; i++)
{
    setMAC(IfInfo[i].MACAddr, 0);
}

// 为得到真实网卡地址，使用虚假的MAC地址和IP地址向本机发送ARP请求
setMAC(srcMAC, 66); // 设置虚假的MAC地址
srcIP = inet_addr("112.112.112.112"); // 设置虚假的IP地址
for (i = 0; i < IfCount; i++){
    ARPRequest(IfInfo[i].adhandle, srcMAC, srcIP, IfInfo[i].ip[0].IPAddr);
}
```

在数据包捕获线程中通过`pcap_next_ex()`函数不停地捕获数据包，捕获的时候需要设置过滤条件为`ARP`包、响应包且源`IP`地址为该网卡的`IP`地址（作为参数传入），当捕获到这样一个数据包就从中获得`MAC`地址并且给传入的参数赋值，然后`return 0`，该线程结束。

```
// 获取本地接口MAC地址线程
UINT CaptureLocalARP(PVOID pParam)
{
```

```
    int res;
    struct pcap_pkthdr *header;
    const u_char *pkt_data;
    IfInfo_t *pIfInfo;
    ARPFrame_t *ARPFrame;
    CString DisplayStr;

    pIfInfo = (IfInfo_t *)pParam;

    while (true)
    {
        res = pcap_next_ex(pIfInfo->adhandle, &header, &pkt_data);
        // 超时
        if (res == 0)
            continue;
        if (res > 0)
        {
            ARPFrame = (ARPFrame_t *) (pkt_data);
            // 得到本接口的MAC地址
            if ((ARPFrame->FrameHeader.FrameType == htons(0x0806))
                && (ARPFrame->Operation == htons(0x0002))
                && (ARPFrame->SendIP == pIfInfo->ip[0].IPAddr))
            {
                cpyMAC(pIfInfo->MACAddr, ARPFrame->SendHa);
                return 0;
            }
        }
    }
}
```

通过`ARPRequest()`函数发送`ARP`请求，该函数传入的参数有设备列表的句柄、源`MAC`地址、源`IP`地址以及目的`IP`地址。首先构造`FrameHeader`，它的目的MAC地址设置为全广播地址，源`MAC`地址设置为66:66:66:66:66:66，然后填充`ARP`帧，设置其硬件类型、保护协议等参数，其源`IP`和目的`IP`均为传进来的参数，分别为112.112.112.112和本机网卡上的`IP`。然后通过`pcap_sendpacket()`函数发送数据包。

```
// 发送ARP请求
void ARPRequest(pcap_t *adhandle, UCHAR *srcMAC, ULONG srcIP, ULONG targetIP)
{
    ARPFrame_t ARPFrame;
    int i;

    for (i = 0; i < 6; i++)
    {
        ARPFrame.FrameHeader.DesMAC[i] = 255;
        ARPFrame.FrameHeader.SrcMAC[i] = srcMAC[i];
        ARPFrame.SendHa[i] = srcMAC[i];
        ARPFrame.RecvHa[i] = 0;
    }

    ARPFrame.FrameHeader.FrameType = htons(0x0806);
    ARPFrame.HardwareType = htons(0x0001);
    ARPFrame.ProtocolType = htons(0x0800);
    ARPFrame.HLen = 6;
    ARPFrame.PLen = 4;
    ARPFrame.Operation = htons(0x0001);
```

```
    ARPFrame.SendIP = srcIP;
    ARPFrame.RecvIP = targetIP;

    pcap_sendpacket(adhandle, (u_char *) &ARPFrame, sizeof(ARPFrame_t));
}
```

7、初始化路由表

由于我们已经获得了网卡上的所有IP地址以及掩码地址，此时可以实现直接投递的路由表项的添加。对于网卡上的每一个IP地址，目标网络为IP地址与掩码地址相与，下一跳地址为0，代表直接投递，子网掩码即为网卡上的掩码地址，然后将这条路由表项加入到路由表中，并且在listbox控件中显示。

```
// 初始化路由表并显示
RouteTable_t rt;
for (i = 0; i < IfCount; i++)
{
    for (j = 0; j < IfInfo[i].ip.GetSize(); j++)
    {
        rt.IfNo = i;
        rt.DstIP = IfInfo[i].ip[j].IPAddr  & IfInfo[i].ip[j].IPMask;
        rt.Mask  = IfInfo[i].ip[j].IPMask;
        rt.NextHop = 0; // 直接投递
        RouteTable.AddTail(rt);
        m_routeTable.InsertString(-1, IPntoa(rt.Mask) + " -- " +
IPntoa(rt.DstIP) + " -- " + IPntoa(rt.NextHop) + "  (直接投递)");
    }
}
```

8、设置过滤规则

该路由器在运行过程中只需要接收两种数据包，分别为ARP相应帧，它是路由器先发送ARP数据包请求与IP地址相对应的MAC地址，当该IP地址对应的主机收到这个请求包，会发送响应包，其中包括了自己的MAC地址，解析这个ARP响应包即可获得想要的MAC地址；以及需要路由的帧，具体表现为在两个子网中的主机若想要通信，则必须经过路由器的转发。主机A发送数据包的IP地址为主机B的IP地址，MAC地址为路由器的MAC地址，当路由器收到这样一个数据包之后，就查找主机B的MAC地址并转发数据包。因此我们要设置过滤规则将上述两种数据包过滤出来。其具体的逻辑为：ARP数据包且为响应包，或者目的IP不是自己网卡上任何一个IP的数据包。然后通过pcap_compile()函数把刚才设置的过滤字符串编译进过滤信息。

```
// 设置过滤规则:仅仅接收arp响应帧和需要路由的帧
CString Filter, Filter0, Filter1;
Filter1 = "(";
for (i = 0; i < IfCount; i++)
{
    Filter0 += L"(ether dst " + MACntoa(IfInfo[i].MACAddr) + L")";
    for (j = 0; j < IfInfo[i].ip.GetSize(); j++){
        Filter1 += L"(ip dst host " + IPntoa(IfInfo[i].ip[j].IPAddr) + L")";
        if (((j == (IfInfo[i].ip.GetSize() - 1))) && (i == (IfCount - 1)))
            Filter1 += ")";
        else
            Filter1 += " or ";
    }
    if (i != (IfCount-1))
        Filter0 += " or ";
}
```

```
Filter = Filter0 + L" and ((arp and (ether[21]=0x2)) or (not" + Filter1 + L"))";

for(int i=0;i<Filter.GetLength();i++)
{
    strbuf[i]=char(Filter[i]);
}
strbuf[Filter.GetLength()]='\0';

for (i = 0; i < IfCount; i++)
{
    if ((pcap_compile(IfInfo[i].adhandle , &fcode, strbuf, 1,
IfInfo[i].ip[0].IPMask) <0 )||(pcap_setfilter(IfInfo[i].adhandle, &fcode)<0))
    {
        MessageBox(Filter+L"过滤规则编译不成功，请检查书写的规则语法是否正确！设置过滤器错
误！");
        PostMessage(WM_QUIT,0,0);
    }
}
```

9、释放设备列表

这时，我们就不需要设备列表了，将其释放。

```
// 不再需要该设备列表,释放之
pcap_freealldevs(m_alldevs);
```

10、开启捕获数据包线程

接下来，通过`Capture()`线程捕获数据包，若线程初始化失败，则弹出提示框。

```
// 开始捕获数据包
pthread = AfxBeginThread(Capture, &IfInfo[i], THREAD_PRIORITY_NORMAL);
if(!pthread)
{
    MessageBox(L"创建数据包捕获线程失败！");
    PostMessage(WM_QUIT, 0, 0);
}
```

在数据包捕获线程中，在循环中调用`pcap_next_ex()`函数捕获数据包，然后分析捕获到的数据包，如果它的帧类型是0806，代表它是`ARP`数据包，转到`ARP`数据包处理函数；如果它的帧类型是0800，代表它是`IP`数据包，转到`IP`数据包处理函数。此处使用前面定义的各种数据结构来承接捕获到的数据包，会方便很多。

```
// 数据包捕获线程
UINT Capture(PVOID pParam)
{
    int res;
    IfInfo_t *pIfInfo;
    struct pcap_pkthdr *header;
    const u_char *pkt_data;

    pIfInfo = (IfInfo_t *)pParam;

    // 开始正式接收并处理帧
    while (true)
    {
```

```
        res = pcap_next_ex( pIfInfo->adhandle, &header, &pkt_data);

        if (res == 1)
        {
            FrameHeader_t *fh;
            fh = (FrameHeader_t *)pkt_data;
            switch (ntohs(fh->FrameType))
            {
                case 0x0806:
                    ARPFrame_t *ARPf;
                    ARPf = (ARPFrame_t *)pkt_data;
                    // ARP包，转到ARP包处理函数
                    ARPPacketProc(header, pkt_data);
                    break;

                case 0x0800:
                    IPFrame_t *IPf;
                    IPf = (IPFrame_t*) pkt_data;
                    // IP包，转到IP包处理函数
                    IPPacketProc(pIfInfo, header, pkt_data);
                    break;
                default:
                    break;
            }
        }
        else if (res == 0)  // 超时
        {
            continue;
        }
        else
        {
            AfxMessageBox(L"pcap_next_ex函数出错!");
        }
    }
    return 0;
}
```

11、ARP数据包处理

当收到一个ARP数据包后，调用该ARP数据包处理函数，首先判断该ARP帧的操作数，如果是2，代表其是ARP响应包，将收到该数据包的信息打印到日志里，然后查询IP-MAC地址映射表，若IP-MAC地址映射表中已经存在这个对应关系，输出日志信息后直接返回，倘若没有这个对应关系，将其加入进去。然后由于新获得了一个对应关系，需要检查缓冲区中有无可以发送的数据包，在对缓冲区进行操作之前，需要先加锁，然后遍历转发缓冲区，倘若找到了数据包的目的IP地址为刚加进来的映射中的IP地址，则修改数据包的目的MAC地址为刚才获得的MAC地址，并将其转发。当缓冲区里没有可以转发的数据包，解锁。

```
// 处理ARP数据包
void ARPPacketProc(struct pcap_pkthdr *header, const u_char *pkt_data)
{
    bool flag;
    ARPFrame_t *ARPf;
    IPFrame_t *IPf;
    SendPacket_t sPacket;
    POSITION pos, CurrentPos;
    IP_MAC_t ip_mac;
```

```
    UCHAR macAddr[6];

    ARPf = (ARPFrame_t *)pkt_data;

    if (ARPf->Operation == ntohs(0x0002))
    {
        pDlg->m_log.InsertString(-1, L"收到ARP响应包");
        pDlg->m_log.InsertString(-1, (L"   ARP "+(IPntoa(ARPf->SendIP))+L" -- "
                +MACntoa(ARPf->SendHa)));
        // IP-MAC地址映射表中已经存在该对应关系
        if (IPLookup(ARPf->SendIP, macAddr))
        {
            pDlg->m_log.InsertString(-1, L"    该对应关系已经存在于IP—MAC地址映射表
中");
            return;
        }
        else
        {
            ip_mac.IPAddr = ARPf->SendIP;
            memcpy(ip_mac.MACAddr, ARPf->SendHa, 6);
            // 将IP-MAC映射关系存入表中
            IP_MAC.AddHead(ip_mac);
            // 日志输出信息
            pDlg->m_log.InsertString(-1,L"   将该对应关系存入IP—MAC地址映射表中");
        }

        mMutex.Lock(INFINITE);
        do{
            // 查看是否能转发缓存中的IP数据报
            flag = false;
            // 没有需要处理的内容
            if (SP.IsEmpty())
                break;
            // 遍历转发缓存区
            pos = SP.GetHeadPosition();
            for (int i=0; i < SP.GetCount(); i++)
            {
                CurrentPos = pos;
                sPacket = SP.GetNext(pos);

                if (sPacket.TargetIP == ARPf->SendIP)
                {
                    IPf = (IPFrame_t *)sPacket.PktData;
                    cpyMAC(IPf->FrameHeader.DesMAC, ARPf->SendHa);

                    for(int t = 0; t < 6; t++)
                    {
                        IPf->FrameHeader.SrcMAC[t] =
IfInfo[sPacket.IfNo].MACAddr[t];
                    }
                    // 发送IP数据包
                    pcap_sendpacket(IfInfo[sPacket.IfNo].adhandle, (u_char *)
sPacket.PktData, sPacket.len);

                    SP.RemoveAt(CurrentPos);

                    // 日志输出信息
```

```
                    pDlg->m_log.InsertString(-1, L"    转发缓存区中目的地址是该MAC地址
的IP数据包");
                    pDlg->m_log.InsertString(-1, (L"      发送IP数据
包："+IPntoa(IPf->IPHeader.SrcIP) + L"->"
                        + IPntoa(IPf->IPHeader.DstIP) + "  " + MACntoa(IPf-
>FrameHeader.SrcMAC )
                        +L"->"+MACntoa(IPf->FrameHeader.DesMAC)));

                    flag = true;
                    break;
                }
            }
        } while(flag);

        mMutex.Unlock();
    }
}
```

我们通过 `IPLookup()` 函数来查询 `IP-MAC` 映射表，具体功能为通过给定的 `IP` 在 `IP-MAC` 映射表中查找对应的 `MAC` 地址。通过遍历即可获得。

```
// 查询IP-MAC映射表
bool IPLookup(ULONG ipaddr, UCHAR *p)
{
    IP_MAC_t ip_mac;
    POSITION pos;

    if (IP_MAC.IsEmpty())
        return false;

    pos = IP_MAC.GetHeadPosition();
    for (int i = 0; i < IP_MAC.GetCount(); i++)
    {
        ip_mac = IP_MAC.GetNext(pos);
        if (ipaddr == ip_mac.IPAddr)
        {
            for (int j = 0; j < 6; j++)
            {
                p[j] = ip_mac.MACAddr[j];
            }
            return true;
        }
    }
    return false;
}
```

12、`IP` 数据包处理

当收到一个 `IP` 数据包后，调用该 `IP` 数据包处理函数。首先检查是否超时，即检查 `IP` 数据包首部的 `TTL`，若超时，则发送类型值为11的 `ICMP` 超时数据包，并直接返回。然后进行校验和检验，如果检验结果不正确，在日志里打印错误信息，同样直接返回。接下来进行路由查询，调用 `RouteLookup()` 函数在路由表中查找下一跳地址，如果未找到对应信息，则发送类型值为3的 `ICMP` 目的不可达数据包。若找到了下一跳地址，则需要进行数据包的转发。首先修改数据包的源 `MAC` 地址，然后重新计算校验和。接下来，在 `IP-MAC` 地址映射表中查找目的 `IP` 对应的 `MAC` 地址，若找到，则转发该数据包，若没找到，

则将该数据包加入缓冲区，同时发送ARP请求包，请求这个IP对应的MAC地址。在加入缓冲区前仍然要加锁，同时开启一个定时器，若超时或者缓冲区已满，则直接丢弃该数据包。

```
// 处理IP数据包
void IPPacketProc(IfInfo_t *pIfInfo, struct pcap_pkthdr *header, const u_char
*pkt_data)
{
    IPFrame_t *IPf;
    SendPacket_t sPacket;

    IPf = (IPFrame_t *) pkt_data;

    pDlg->m_log.InsertString(-1, (L"收到IP数据包:" + IPntoa(IPf->IPHeader.SrcIP) +
L"->"
        + IPntoa(IPf->IPHeader.DstIP)));

    // ICMP超时
    if (IPf->IPHeader.TTL <= 1)
    {
        ICMPPacketProc(pIfInfo, 11, 0, pkt_data);
        return;
    }

    IPHeader_t *IpHeader = &(IPf->IPHeader);
    // ICMP差错
    if (IsChecksumRight((char *)IpHeader) == 0)
    {
        // 日志输出信息
        pDlg->m_log.InsertString(-1, L"   IP数据包包头校验和错误，丢弃数据包");
        return;
    }

    DWORD nextHop; // 经过路由选择算法得到的下一站目的IP地址
    UINT ifNo; // 下一跳的接口序号
    // 路由查询
    if((nextHop = RouteLookup(ifNo, IPf->IPHeader.DstIP, &RouteTable)) == -1)
    {
        // ICMP目的不可达
        ICMPPacketProc(pIfInfo, 3, 0, pkt_data);
        return;
    }
    else
    {
        sPacket.IfNo = ifNo;
        sPacket.TargetIP = nextHop;

        cpyMAC(IPf->FrameHeader.SrcMAC, IfInfo[sPacket.IfNo].MACAddr);

        // TTL减1
        IPf->IPHeader.TTL -= 1;

        unsigned short check_buff[sizeof(IPHeader_t)];
        // 设IP头中的校验和为0
        IPf->IPHeader.Checksum = 0;

        memset(check_buff, 0, sizeof(IPHeader_t));
        IPHeader_t * ip_header = &(IPf->IPHeader);
```

```
        memcpy(check_buff, ip_header, sizeof(IPHeader_t));

        // 计算IP头部校验和
        IPf->IPHeader.Checksum = ChecksumCompute(check_buff,
sizeof(IPHeader_t));

        // IP-MAC地址映射表中存在该映射关系
        if (IPLookup(sPacket.TargetIP, IPf->FrameHeader.DesMAC))
        {
            memcpy(sPacket.PktData, pkt_data, header->len);
            sPacket.len = header->len;
            if(pcap_sendpacket(IfInfo[sPacket.IfNo].adhandle, (u_char *)
sPacket.PktData, sPacket.len) != 0)
            {
                // 错误处理
                AfxMessageBox(L"发送IP数据包时出错!");
                return;
            }

            // 日志输出信息
            pDlg->m_log.InsertString(-1,L"   转发IP数据包：");
            pDlg->m_log.InsertString(-1,(L"   "+IPntoa(IPf->IPHeader.SrcIP) +
L"->"
                + IPntoa(IPf->IPHeader.DstIP) + "  " + MACntoa(IPf-
>FrameHeader.SrcMAC )
                + L"->" + MACntoa(IPf->FrameHeader.DesMAC)));
        }
        // IP-MAC地址映射表中不存在该映射关系
        else
        {
            if (SP.GetCount() < 65530)      // 存入缓存队列
            {
                sPacket.len = header->len;
                // 将需要转发的数据报存入缓存区
                memcpy(sPacket.PktData, pkt_data, header->len);

                // 在某一时刻只允许一个线程维护链表
                mMutex.Lock(INFINITE);

                sPacket.n_mTimer = TimerCount;
                if (TimerCount++ > 65533)
                {
                    TimerCount = 1;
                }
                pDlg->SetTimer(sPacket.n_mTimer, 10000, NULL);
                SP.AddTail(sPacket);

                mMutex.Unlock();

                // 日志输出信息
                pDlg->m_log.InsertString(-1,L"   缺少目的MAC地址，将IP数据包存入转发
缓冲区");
                pDlg->m_log.InsertString(-1, (L"   存入转发缓冲区的数据包
为："+IPntoa(IPf->IPHeader.SrcIP)
                    + L"->" + IPntoa(IPf->IPHeader.DstIP) + L"  " + MACntoa(IPf-
>FrameHeader.SrcMAC)
                    + L"->xx:xx:xx:xx:xx:xx"));
                pDlg->m_log.InsertString(-1, L"   发送ARP请求");
```

```
                // 发送ARP请求
                ARPRequest(IfInfo[sPacket.IfNo].adhandle,
IfInfo[sPacket.IfNo].MACAddr,
                    IfInfo[sPacket.IfNo].ip[1].IPAddr, sPacket.TargetIP);
            }
            else // 如缓存队列太长，抛弃该报
            {
                // 日志输出信息
                pDlg->m_log.InsertString(-1, L"    转发缓冲区溢出，丢弃IP数据包");
                pDlg->m_log.InsertString(-1, (L"    丢弃的IP数据包为：" +
IPntoa(IPf->IPHeader.SrcIP) + L"->"
                    + IPntoa(IPf->IPHeader.DstIP) + L"  " + MACntoa(IPf-
>FrameHeader.SrcMAC)
                    + L"->xx:xx:xx:xx:xx:xx"));
            }
        }
    }
}
```

在IP数据包转发时，需要查询路由表来获得下一跳地址。由于我们要进行最长匹配，因此设置一个变量记录以匹配到的掩码地址，然后遍历路由表，找到目的IP所在的网络且掩码最长的路由表项，获得其下一跳地址。若为直接投递，则将目的IP地址设置为原来的IP地址即可。

```
// 查询路由表
DWORD RouteLookup(UINT &ifNO, DWORD desIP, CList <RouteTable_t, RouteTable_t&>
*routeTable)
{
    // desIP为网络序
    DWORD MaxMask = 0; // 获得最大的子网掩码的地址，没有获得时初始化为-1
    int Index = -1; // 获得最大的子网掩码的地址对应的路由表索引，以便获得下一站路由器的地址

    POSITION pos;
    RouteTable_t rt;
    DWORD tmp;

    pos = routeTable->GetHeadPosition();
    for (int i=0; i < routeTable->GetCount(); i++)
    {
        rt = routeTable->GetNext(pos);
        if ((desIP & rt.Mask) == rt.DstIP)
        {
            Index = i;

            if(rt.Mask >= MaxMask)
            {
                MaxMask = rt.Mask;
                ifNO = rt.IfNo;

                if (rt.NextHop == 0)    // 直接投递
                {
                    tmp = desIP;
                }
                else
                {
                    tmp = rt.NextHop;
                }
```

```
            }
        }
    }

    if(Index == -1)             // 目的不可达
    {
        return -1;
    }
    else        // 找到了下一跳地址
    {
        return tmp;
    }
}
```

13、发送 ICMP 数据包

若发生超时或者目的不可达，则应发送 ICMP 数据包，首先填充帧首部，将源 MAC 地址和目的 MAC 地址互换，并且设置帧类型为 IP 帧。然后填充 IP 首部，修改源 IP 地址为本机网卡上的 IP 地址。然后填充 ICMP 首部，根据传入的参数设置 Type 和 code，然后计算校验和并填充。最后填充数据并发送数据包、打印日志信息。

```
// 发送ICMP数据包
void ICMPPacketProc(IfInfo_t *pIfInfo, BYTE type, BYTE code, const u_char
*pkt_data)
{
    u_char * ICMPBuf = new u_char[70];

    // 填充帧首部
    memcpy(((FrameHeader_t *)ICMPBuf)->DesMAC, ((FrameHeader_t *)pkt_data)-
>SrcMAC, 6);
    memcpy(((FrameHeader_t *)ICMPBuf)->SrcMAC, ((FrameHeader_t *)pkt_data)-
>DesMAC, 6);
    ((FrameHeader_t *)ICMPBuf)->FrameType = htons(0x0800);

    // 填充IP首部
    ((IPHeader_t *)(ICMPBuf+14))->Ver_HLen = ((IPHeader_t *)(pkt_data+14))-
>Ver_HLen;
    ((IPHeader_t *)(ICMPBuf+14))->TOS = ((IPHeader_t *)(pkt_data+14))->TOS;
    ((IPHeader_t *)(ICMPBuf+14))->TotalLen = htons(56);
    ((IPHeader_t *)(ICMPBuf+14))->ID = ((IPHeader_t *)(pkt_data+14))->ID;
    ((IPHeader_t *)(ICMPBuf+14))->Flag_Segment = ((IPHeader_t *)(pkt_data+14))-
>Flag_Segment;
    ((IPHeader_t *)(ICMPBuf+14))->TTL = 64;
    ((IPHeader_t *)(ICMPBuf+14))->Protocol = 1;
    //((IPHeader_t *)(ICMPBuf+14))->SrcIP = ((IPHeader_t *)(pkt_data+14))-
>DstIP;
    ((IPHeader_t *)(ICMPBuf+14))->SrcIP = pIfInfo->ip[1].IPAddr;
    ((IPHeader_t *)(ICMPBuf+14))->DstIP = ((IPHeader_t *)(pkt_data+14))->SrcIP;
    ((IPHeader_t *)(ICMPBuf+14))->Checksum = htons(ChecksumCompute((unsigned
short *)(ICMPBuf+14),20));

    // 填充ICMP首部
    ((ICMPHeader_t *)(ICMPBuf+34))->Type = type;
    ((ICMPHeader_t *)(ICMPBuf+34))->Code = code;
    ((ICMPHeader_t *)(ICMPBuf+34))->Id = 0;
    ((ICMPHeader_t *)(ICMPBuf+34))->Sequence = 0;
```

```
    ((ICMPHeader_t *)(ICMPBuf+34))->Checksum = htons(ChecksumCompute((unsigned
short *)(ICMPBuf+34),8));

    // 填充数据
    memcpy((u_char *)(ICMPBuf+42),(IPHeader_t *)(pkt_data+14),20);
    memcpy((u_char *)(ICMPBuf+62),(u_char *)(pkt_data+34),8);

    // 发送数据包
    pcap_sendpacket(pIfInfo->adhandle, (u_char *)ICMPBuf, 70 );

    // 日志输出信息
    if (type == 11)
    {
        pDlg->m_log.InsertString(-1, L"    发送ICMP超时数据包：");
    }
    if (type == 3)
    {
        pDlg->m_log.InsertString(-1, L"    发送ICMP目的不可达数据包：");
    }
    pDlg->m_log.InsertString(-1, (L"    ICMP ->" + IPntoa(((IPHeader_t *)
(ICMPBuf+14))->DstIP)
        + L"-" + MACntoa(((FrameHeader_t *)ICMPBuf)->DesMAC)));

    delete [] ICMPBuf;
}
```

14、校验和计算与检验

在手动构建数据包时，需要计算校验和。具体方法为将数据包首部的数据每16位相加，若有进位则加到最低位，最后取反，得到校验和。检验校验和的过程类似，同样先计算校验和，并与数据包首部的校验和进行比较，如果相同则可以认为数据包传输正确，反之则认为数据包传输过程中丢失了数据。

```
// 计算校验和
unsigned short ChecksumCompute(unsigned short * buffer,int size)
{
    // 32位，延迟进位
    unsigned long cksum = 0;
    while (size > 1)
    {
        cksum += * buffer++;
        // 16位相加
        size -= sizeof(unsigned short);
    }
    if(size)
    {
        // 最后可能有单独8位
        cksum += *(unsigned char *)buffer;
    }
    // 将高16位进位加至低16位
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
    // 取反
    return (unsigned short)(~cksum);
}

// 判断IP数据包头部校验和是否正确
int IsChecksumRight(char * buffer)
```

```
{
    // 获得IP头内容
    IPHeader_t * ip_header = (IPHeader_t *)buffer;
    // 备份原来的校验和
    unsigned short checksumBuf = ip_header->Checksum;
    unsigned short check_buff[sizeof(IPHeader_t)];
    // 设IP头中的校验和为0
    ip_header->Checksum = 0;

    memset(check_buff, 0, sizeof(IPHeader_t));
    memcpy(check_buff, ip_header, sizeof(IPHeader_t));

    // 计算IP头部校验和
    ip_header->Checksum = ChecksumCompute(check_buff, sizeof(IPHeader_t));

    // 与备份的校验和进行比较
    if (ip_header->Checksum == checksumBuf)
        return 1;
    else
        return 0;
}
```

15、添加路由表项

在程序中输入掩码地址、目的网络、下一跳地址后，点击添加，首先，会记录下输入框中的内容，然后检查合法性。如果路由表中已经包含了一条掩码地址相同、目的网络号相同但是下一跳地址不相同的路由表项，表示输入错误，弹出提示框。然后检验下一跳地址的网络是否是该网卡所处的网络，如果不是，同样表示输入错误，弹出提示框。如果通过了合法性检查，把该条路由表项添加到路由表，并且在路由表窗口中显示该路由表项。

```
void Ctask3_2Dlg::OnBnClickedButtonAdd()
{
    // TODO: 在此添加控件通知处理程序代码
    bool flag;
    int i, j;
    DWORD ipaddr;
    RouteTable_t rt;

    m_next.GetAddress(ipaddr);
    ipaddr = htonl(ipaddr);

    // 检查合法性
    DWORD ipaddr1;
    DWORD ipaddr2;
    DWORD ipaddr3;
    POSITION pos, CurrentPos;
    // 记录子网掩码
    m_mask.GetAddress(ipaddr1);
    // 记录目的IP
    m_dest.GetAddress(ipaddr2);
    // 记录下一跳
    m_next.GetAddress(ipaddr3);
    pos = RouteTable.GetHeadPosition();
    for (i = 0; i < RouteTable.GetCount(); i++)
    {
        CurrentPos = pos;
        rt = RouteTable.GetNext(pos);
```

```
        if ((rt.Mask == htonl(ipaddr1)) && ((rt.Mask & rt.DstIP) == (rt.Mask &
htonl(ipaddr2))) && (htonl(ipaddr3) != rt.NextHop))
        {
            MessageBox(L"该路由无法添加，请重新输入！");
            return;
        }
    }

    flag = false;
    for (i = 0; i < IfCount; i++)
    {
        for (j = 0; j < IfInfo[i].ip.GetSize(); j++)
        {
            if (((IfInfo[i].ip[j].IPAddr) & (IfInfo[i].ip[j].IPMask)) ==
((IfInfo[i].ip[j].IPMask) & ipaddr))
            {
                rt.IfNo = i;
                // 记录子网掩码
                m_mask.GetAddress(ipaddr);
                rt.Mask = htonl(ipaddr);
                // 记录目的IP
                m_dest.GetAddress(ipaddr);
                rt.DstIP = htonl(ipaddr);
                // 记录下一跳
                m_next.GetAddress(ipaddr);
                rt.NextHop = htonl(ipaddr);
                // 把该条路由表项添加到路由表
                RouteTable.AddTail(rt);
                // 在路由表窗口中显示该路由表项
                m_routeTable.InsertString(-1, IPntoa(rt.Mask) + " -- "
                    + IPntoa(rt.DstIP) + " -- " + IPntoa(rt.NextHop));
                flag = true;
            }
        }
    }
    if (!flag)
    {
        MessageBox(L"输入错误，请重新输入！");
    }
}
```

16、删除路由表项

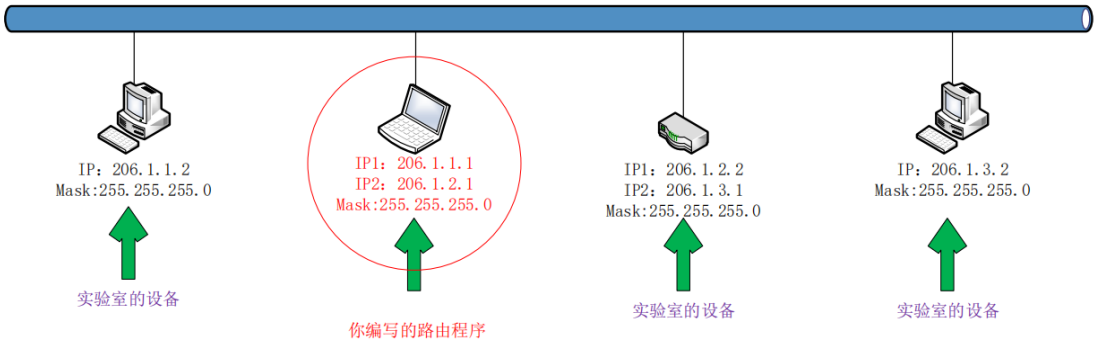
首先选中需要删除的路由表项，然后分别获得子网掩码选项、目的地址选项以及下一跳地址选项，将其从路由表窗口中删除，然后遍历路由表，把需要删除的路由表项从路由表中删除，若下一跳地址为0，表示为直接连接路由，不允许删除，弹出提示框。

```
void Ctask3_2Dlg::OnBnClickedButtonDelete()
{
    // TODO: 在此添加控件通知处理程序代码
    int i;
    char str[100], ipaddr[20];
    ULONG mask, destination, nexthop;
    RouteTable_t rt;
    POSITION pos, CurrentPos;

    str[0] = NULL;
```

```
    ipaddr[0] = NULL;
    if ((i = m_routeTable.GetCurSel()) == LB_ERR)
    {
        return;
    }

    CString STR;
    m_routeTable.GetText(i, STR);

    for(int i = 0; i < STR.GetLength(); i++)
        str[i] = STR[i];
    str[STR.GetLength()] = '\0';

    // 取得子网掩码选项
    strncat_s(ipaddr, str, 15);
    mask = inet_addr(ipaddr);
    // 取得目的地址选项
    ipaddr[0] = 0;
    strncat_s(ipaddr, &str[19], 15);
    destination = inet_addr(ipaddr);
    // 取得下一跳地址选项
    ipaddr[0] = 0;
    strncat_s(ipaddr, &str[38], 15);
    nexthop = inet_addr(ipaddr);

    if (nexthop == 0)
    {
        MessageBox(L"直接连接路由，不允许删除！");
        return;
    }

    // 把该路由表项从路由表窗口中删除
    m_routeTable.DeleteString(i);

    // 路由表中没有需要处理的内容，则返回
    if (RouteTable.IsEmpty())
    {
        return;
    }

    // 遍历路由表,把需要删除的路由表项从路由表中删除
    pos = RouteTable.GetHeadPosition();
    for (i=0; i<RouteTable.GetCount(); i++)
    {
        CurrentPos = pos;
        rt = RouteTable.GetNext(pos);

        if ((rt.Mask == mask) && (rt.DstIP == destination) && (rt.NextHop ==
nexthop))
        {
            RouteTable.RemoveAt(CurrentPos);
            return;
        }
    }
}
```

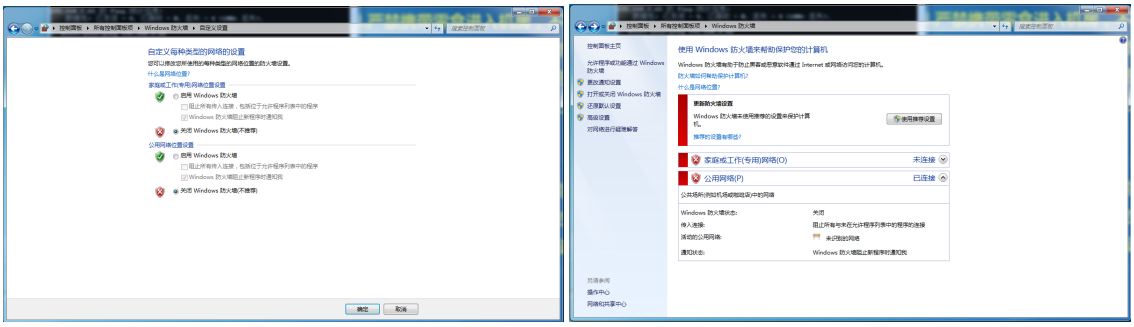
四、实验结果

为了进行结果检验，首先需要按照如下拓扑图进行四台测试电脑的配置。



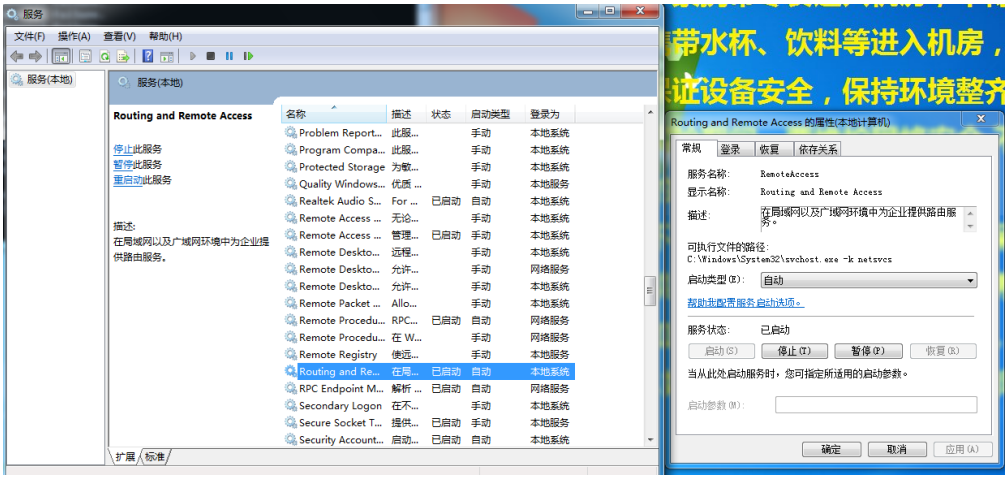
1、关闭防火墙

若希望两台主机可以互相`ping`通，首先要关闭每台计算机上的防火墙，关闭方式为：打卡控制面板，进入“所有控制面板”选项->选择”Windows防火墙“->选择“打开或关闭Windows防火墙”->选择“关闭Windows防火墙（不推荐）”->点击“确定”。如下所示即为正确关闭了防火墙。



2、开启`Routing and Remote Access`服务

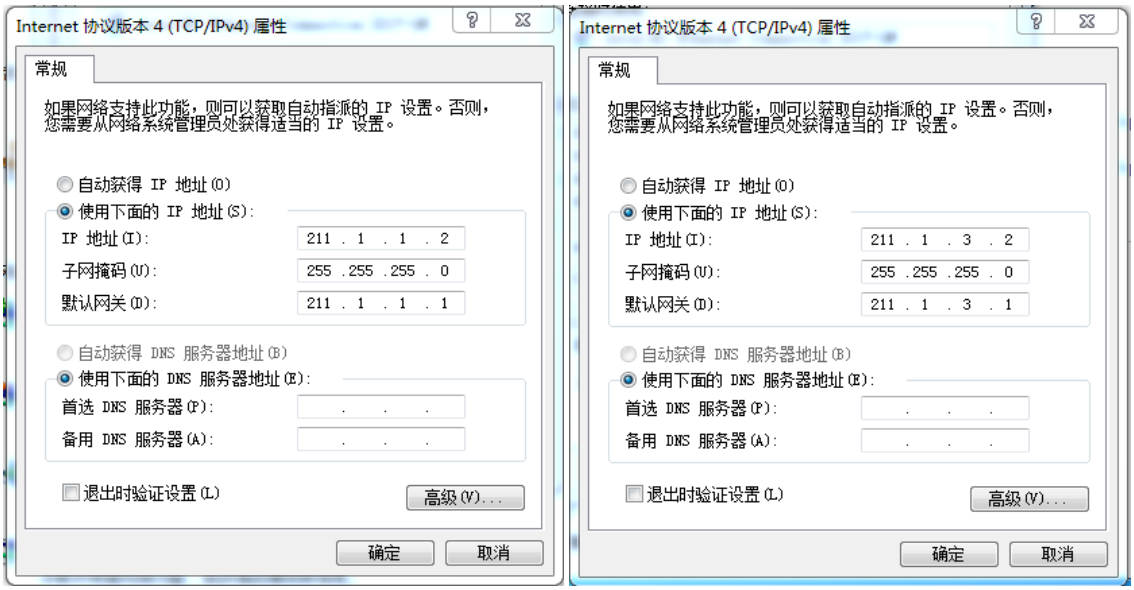
对于第三台电脑，打开“服务”，找到`Routing and Remote Access`，双击进入，在启动类型中选择“手动”，再次进入，单击“启动”即可。



3、配置互联网中主机的`IP`地址和默认路由

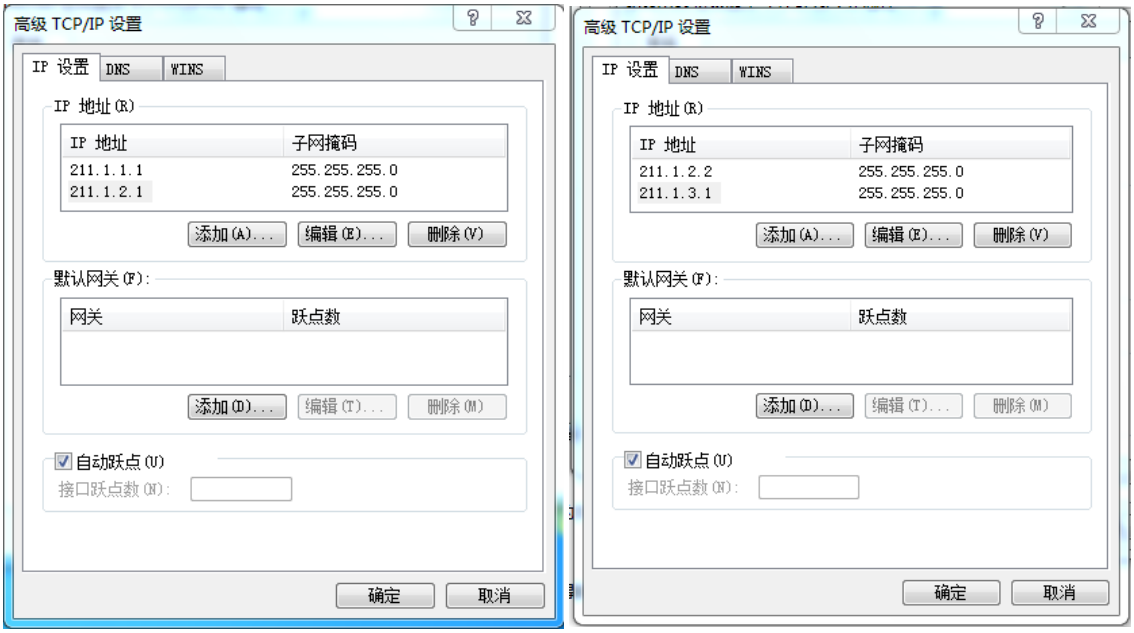
依次点击”计算机“->“网络”->”网络和共享中心“->“本地连接”->”属性“->”Internet协议版本4（TCP/IPv4）“->”属性“，在“Internet协议版本4（TCP/IPv4）属性”对话框中选择”使用下面的`IP`地址（S）“，在“`IP`地址（I）“中输入211.1.1.2，在”子网掩码（U）“中输入255.255.255.0，在”默认网关（D）“中输入211.1.1.1，点击”确定“。此时完成了一个计算机的`IP`地址的配置，接下来，对应地配置另外一台计算

机，它的IP地址、子网掩码、默认网关分别为211.1.3.2，255.255.255.0，211.1.3.1，如下图所示。



4、配置路由设备的IP地址

依次点击"计算机"->"网络"->"网络和共享中心"->"本地连接"->"属性"->"Internet协议版本4（TCP/IPv4）"->"属性"，在"Internet协议版本4（TCP/IPv4）属性"对话框中点击"高级"，分别加入两个IP地址，分别为211.1.1.1和211.1.2.1，它们的子网掩码都是255.255.255.0，点击"确定"。此时完成了一个路由器的IP地址的配置，接下来，对应地配置另外一个路由器，它的两个IP地址分别为211.1.2.2和211.1.3.1，子网掩码都是255.255.255.0，如下图所示。

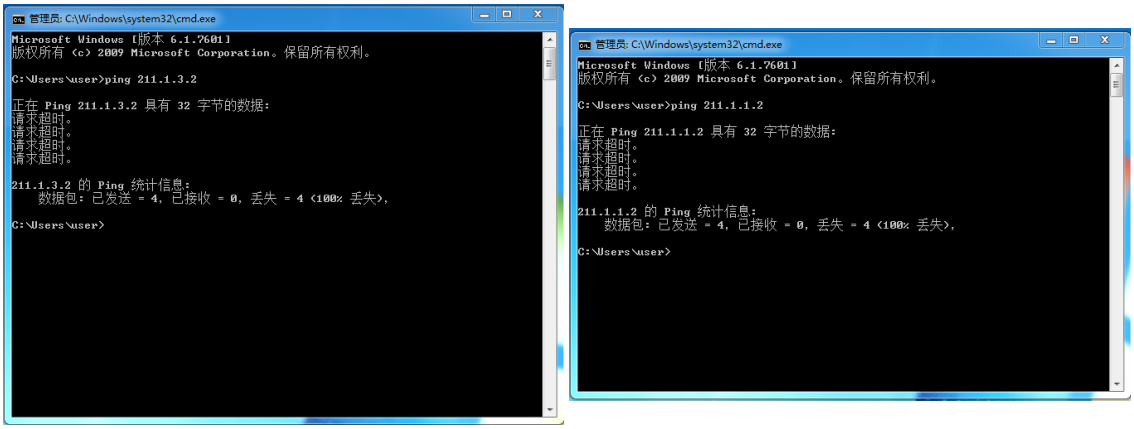


5、利用命令行程序配置路由设备的静态路由

对于第三台电脑，打开命令行，输入"ipconfig"，可以看到我们刚才配置的IP地址，接下来需要配置路由表，输入"route add 211.1.1.0 mask 255.255.255.0 211.1.2.1"，代表我们增加了一条路由表项，其目的网络为211.1.1.0，掩码为255.255.255.0，下一路由器地址为211.1.2.1，如下图所示。

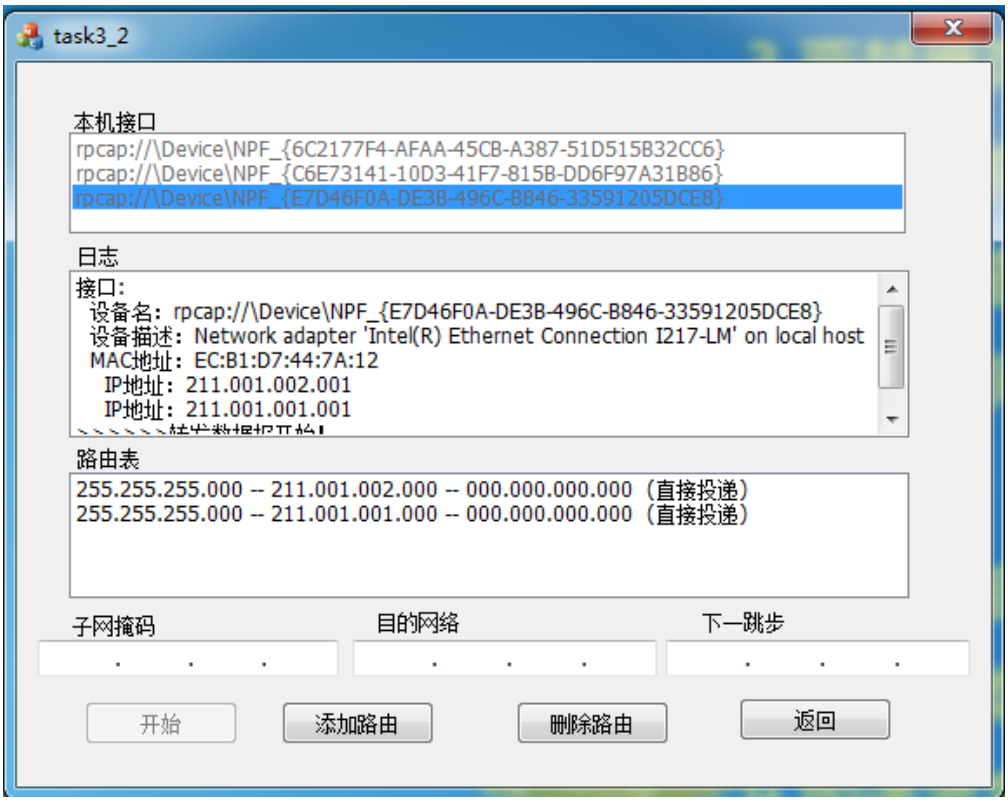
6、测试网络连通性

在启动我们的路由器之前，先测一下两台主机的连通性，在主机1的命令行中输入ping 211.1.3.2，可以看到ping不通，在主机2的命令行中输入ping 211.1.1.2，同样发现ping不通。

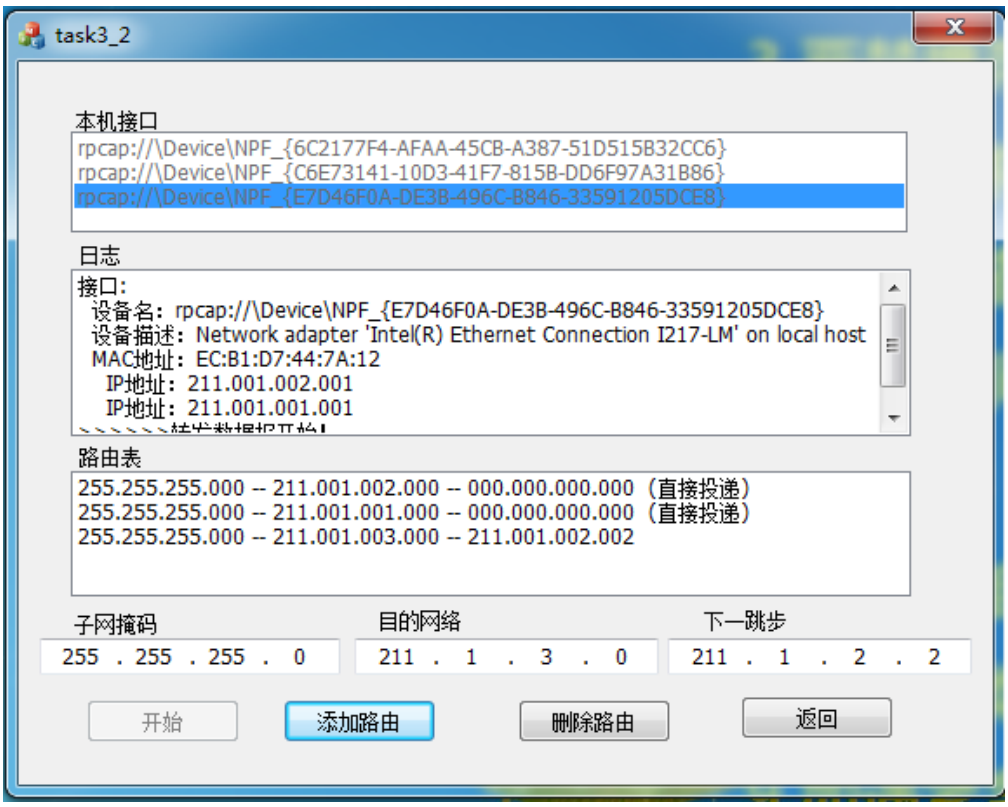


7、启动路由器

在第二台电脑上启动我们的路由器，选择一块网卡，点击开始，可以看到程序正确获得了这块网卡上的`MAC`地址和两个`IP`地址。

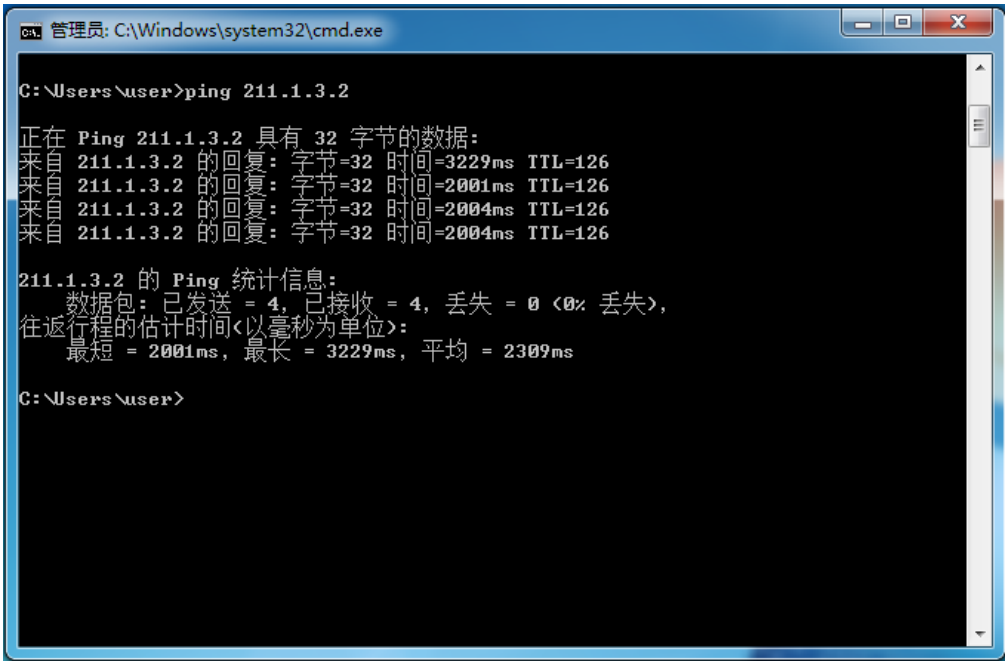


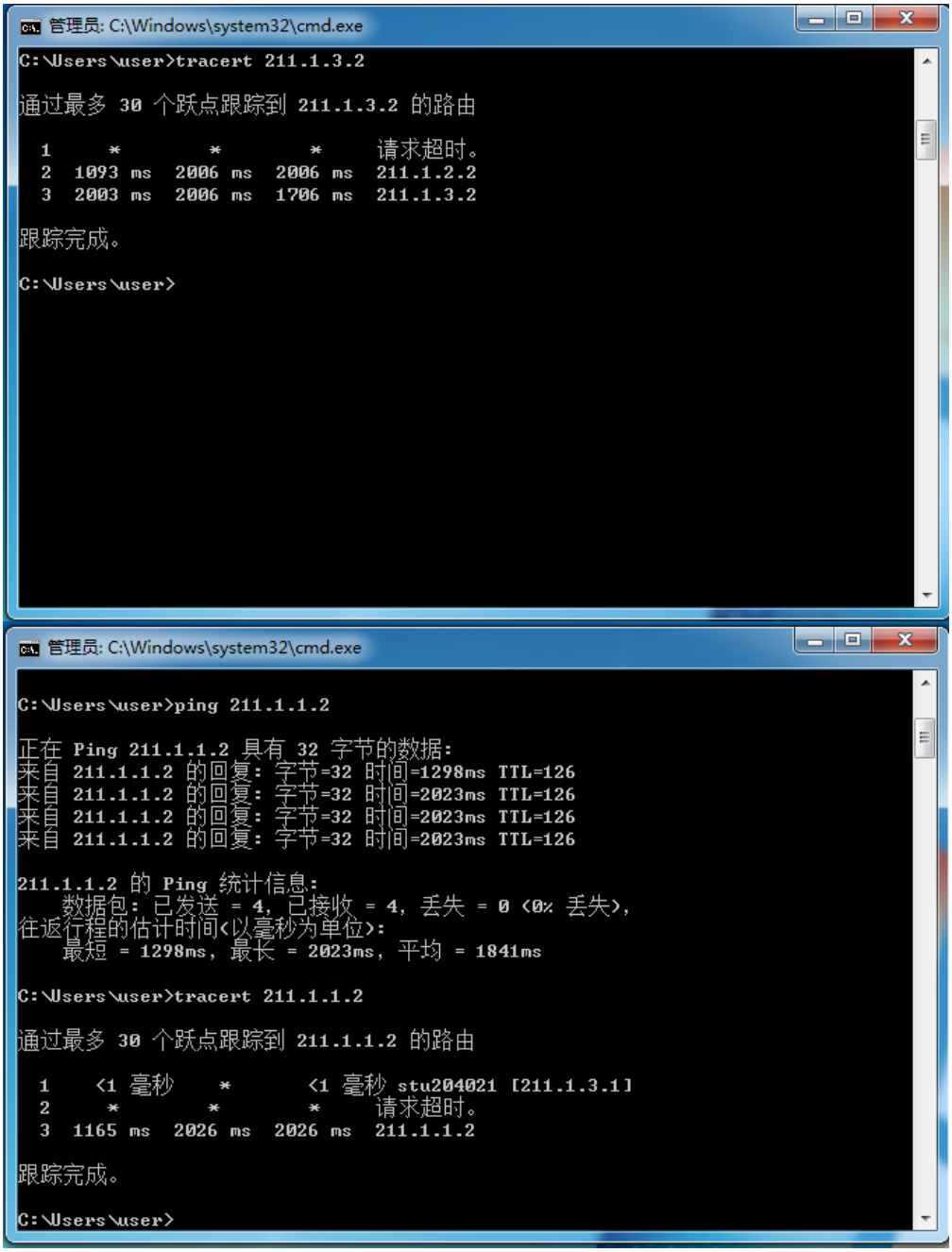
然后添加一条掩码地址为255.255.255.0、目的网络为211.1.3.0、下一跳地址为211.1.2.2的路由表项。



8、再次测试网络连通性

此时，再次在主机1的命令行中输入ping 211.1.3.2，可以看到ping成功，输入tracert 211.1.3.2，观察输出信息；在主机2的命令行中输入ping 211.1.1.2，同样发现ping成功，输入tracert 211.1.1.2，观察输出信息。





遗憾的是，在`tracert`命令中无法显示路由程序所在路由器的`IP`信息，笔者仍将继续探索该问题，找到问题根源所在。

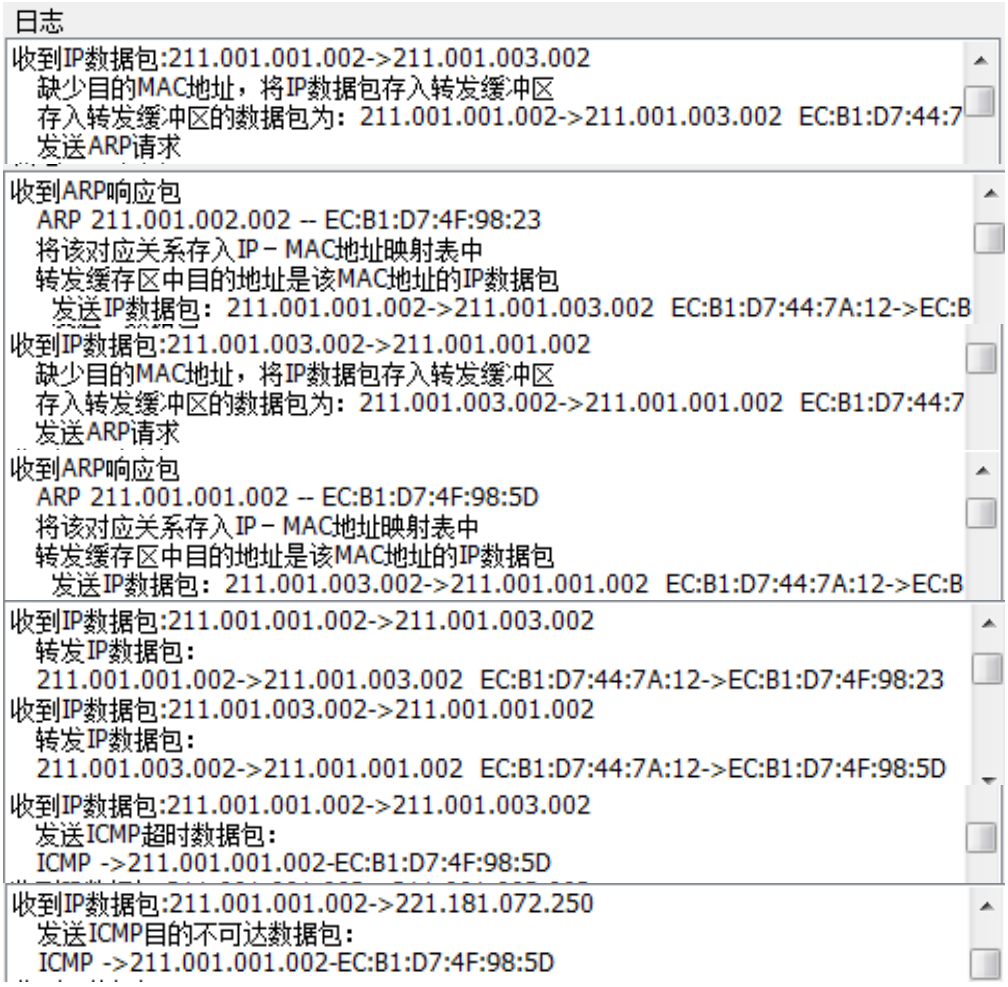
9、路由器输出日志信息

通过观察日志信息，我们可以看到该路由器工作的流程。首先，211.1.1.2给211.1.3.2发送一个数据包，由于这是第一次发送，路由器缺少211.1.3.2的`MAC`地址，因此将这个`IP`数据包存入转发缓冲区，同时发送`ARP`数据包请求211.1.3.2的MAC地址。然后路由器收到`ARP`响应包，获得了211.1.3.2的`MAC`地址，并将这对`IP-MAC`映射关系存入`IP-MAC`地址映射表中，然后将缓冲区中刚才存入的211.1.1.2->211.1.3.2的`IP`数据包转发，在转发之前路由器查询了路由表找到下一跳地址。

然后路由器收到了211.1.3.2给211.1.1.2回复的数据包。同样的，路由器中没有211.1.1.2的`MAC`地址，便先将该数据包存入转发缓冲区中，然后发送`ARP`数据包请求211.1.1.2的`MAC`地址。然后路由器收到`ARP`响应包，将211.1.1.2和其`MAC`地址的对应关系存入`IP-MAC`地址映射表中，然后将缓冲区中刚才存入的211.1.3.2->211.1.1.2的`IP`数据包转发。

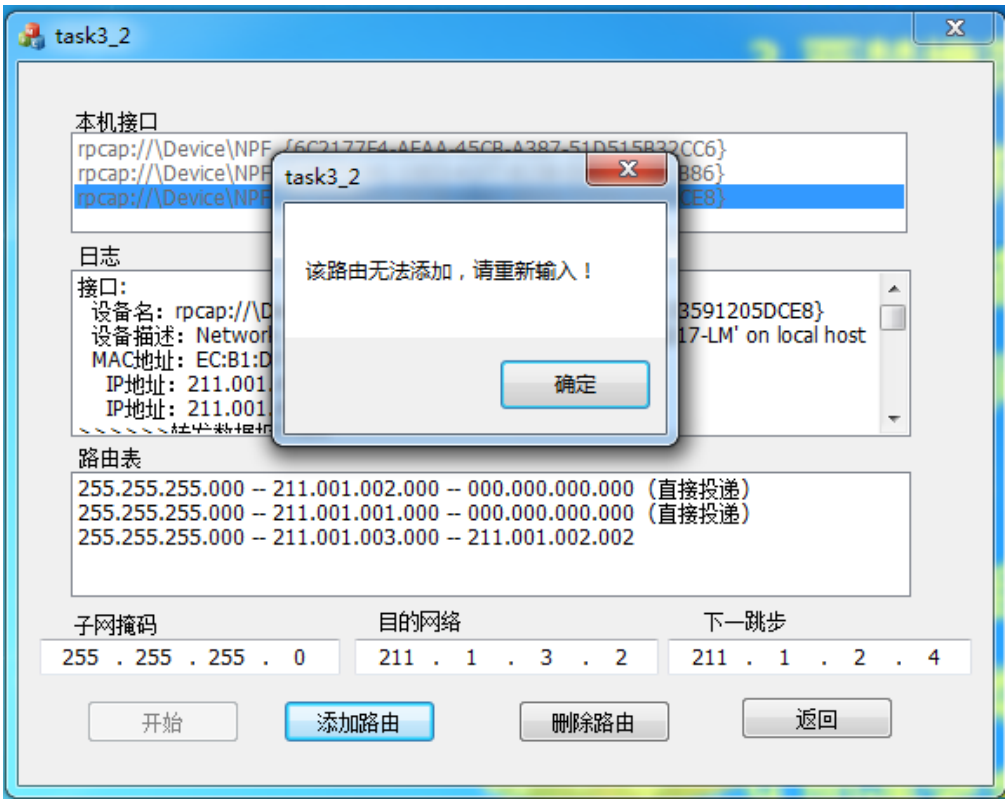
然后路由器再次收到了211.1.1.2给211.1.3.2发送的数据包，这时，由于路由器中已经存了211.1.3.2的`IP`与`MAC`地址的对应关系，通过查`IP-MAC`地址映射表即可获得，因此在获得路由表中的下一跳地址后直接转发。对于211.1.3.2给211.1.1.2发送的数据包同样直接查`IP-MAC`地址映射表即可获得211.1.1.2的`MAC`地址，并完成转发。

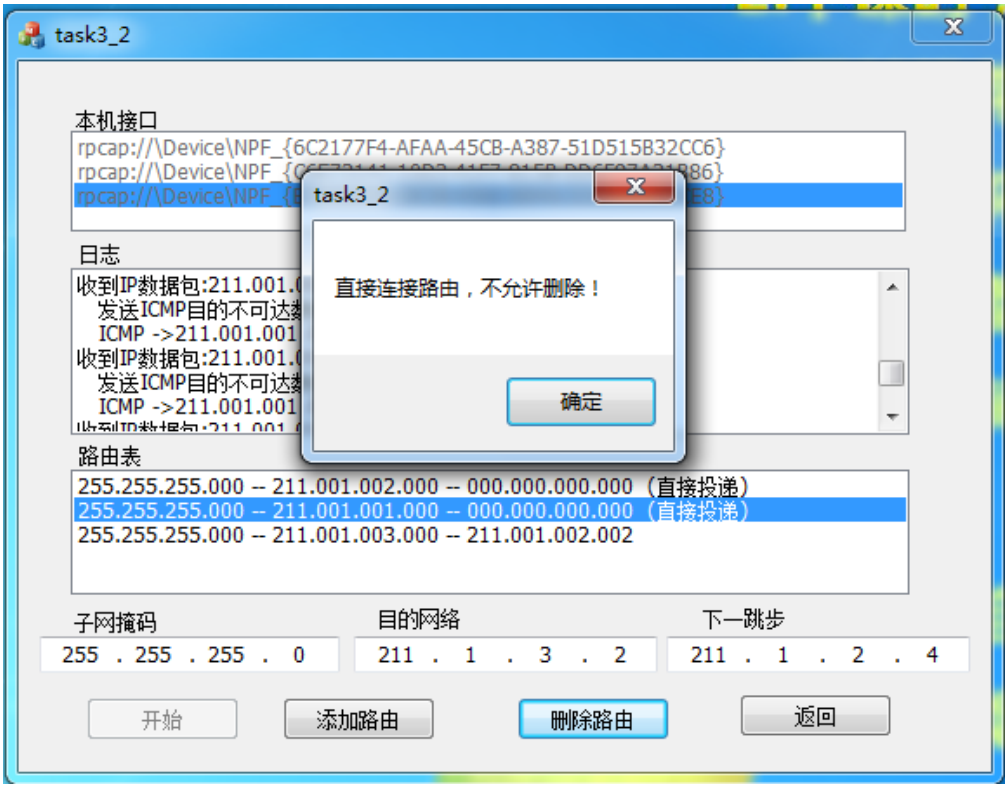
同时，可以看到在运行`tracert`命令时路由器会发送`ICMP`超时数据包；若收到的数据包的目的`IP`地址不可达则会发送`ICMP`目的不可达数据包。



10、添加、删除路由表项

在添加路由表的时候会进行简单的合法性检查，同样，在删除路由表的时候也会进行简单的合法性检查。




task3_2
本机接口
rpcap://\Device\NPF_{6C2177F4-AFAA-45CB-A387-51D515B32CC6}
日志
收到IP数据包:211.001.
发送ICMP目的不可达
ICMP ->211.001.001
收到IP数据包:211.001.
发送ICMP目的不可达
ICMP ->211.001.001
路由表
255.255.255.000 -- 211.001.002.000 -- 000.000.000.000 （直接投递）
255.255.255.000 -- 211.001.001.000 -- 000.000.000.000 （直接投递）
255.255.255.000 -- 211.001.003.000 -- 211.001.002.002
子网掩码
255 . 255 . 255 . 0
目的网络
211 . 1 . 3 . 2
下一跳步
211 . 1 . 2 . 4
开始
添加路由
删除路由
返回
task3_2
直接连接路由，不允许删除！
确定