第五次实验报告

简单路由器程序的设计

郭裕彬 2114052 物联网工程

实验要求

- 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。
- 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。
- 需要给出路由表的手工插入、删除方法。
- 需要给出路由器的工作日志，显示数据报获取和转发过程。
- 完成的程序须通过现场测试，并在班（或小组）中展示和报告自己的设计思路、开发和实现过程、测试方法和过程。

程序设计

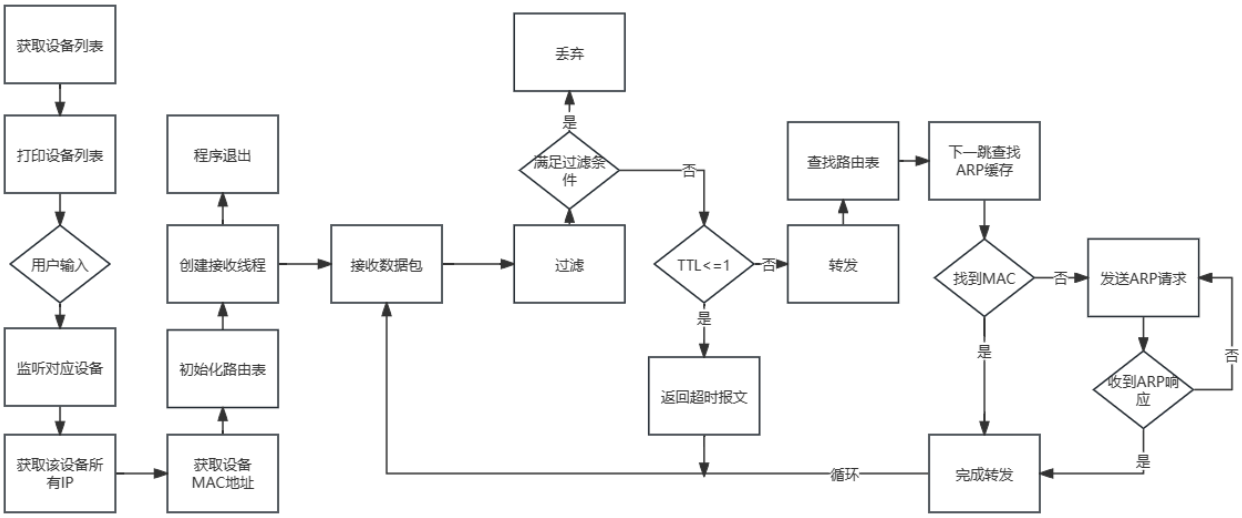
获取网卡列表和对应信息，打开网卡开始监听后获取网卡MAC地址，初始化路由表。

路由器仅需要转发目的MAC地址为自己而目的IP非本机IP的数据包，从监听的网卡中筛选出这些数据包后，首先判断是否存在ttl=1的报文，如果有需要修改其IP头、ICMP报文部分的信息并原路发回修改后的超时报文；否则提取数据包的目的IP地址，通过路由表进行路由选择，选择成功后记录下一跳地址，失败则直接丢弃。

通过下一跳地址查询ARP缓存表，如果未找到该地址对应的物理地址，则需要发送ARP请求报文，得到响应后获得物理地址，修改包中的ttl和校验和，封装成数据帧并通过对应的接口发出数据包。

路由器在命令行界面会输出路由的报文的简单信息，如时间、目的MAC、目的IP、源MAC、源IP等；同时后台会将报文的详细解析信息存入硬盘中的日志文件中。

路由器主函数中会循环接收外部输入的命令，完成路由表的插入、删除、打印、ARP表打印等操作。



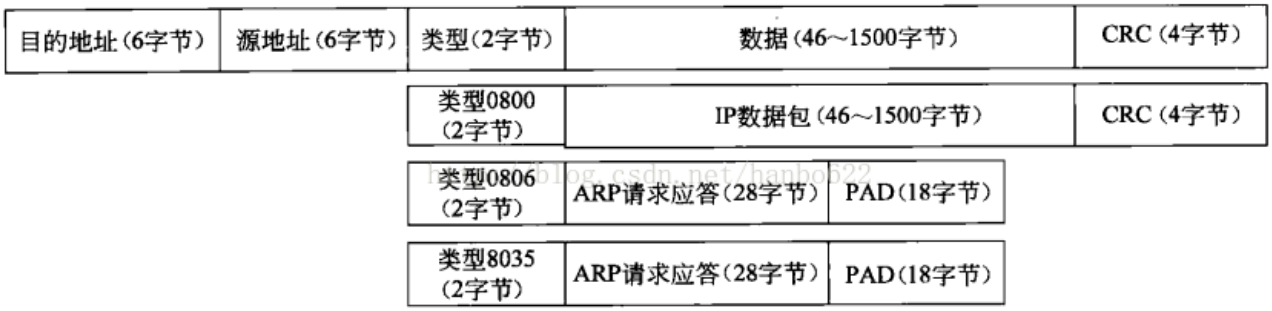
代码实现

数据结构

本次编程使用npcap 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。npcap捕获到的数据包保存在 一个无结构的缓冲区中，我们通过将缓冲区的数据套用到自定义的结构体中，可以简化读取其中数据的流程。此外，在定义这些包数据结构时，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。定义完成后，再使用 #pragma pack(0) 恢复默认对齐方式。

对于路由表和ARP缓存表，使用类来进行封装，并附加对表结构的处理函数；

以太帧首部



以太帧首部为上图中目的地址、源地址和类型三个字段，其中类型字段决定了路由器和接收方按照什么方式来处理数据部分的内容。

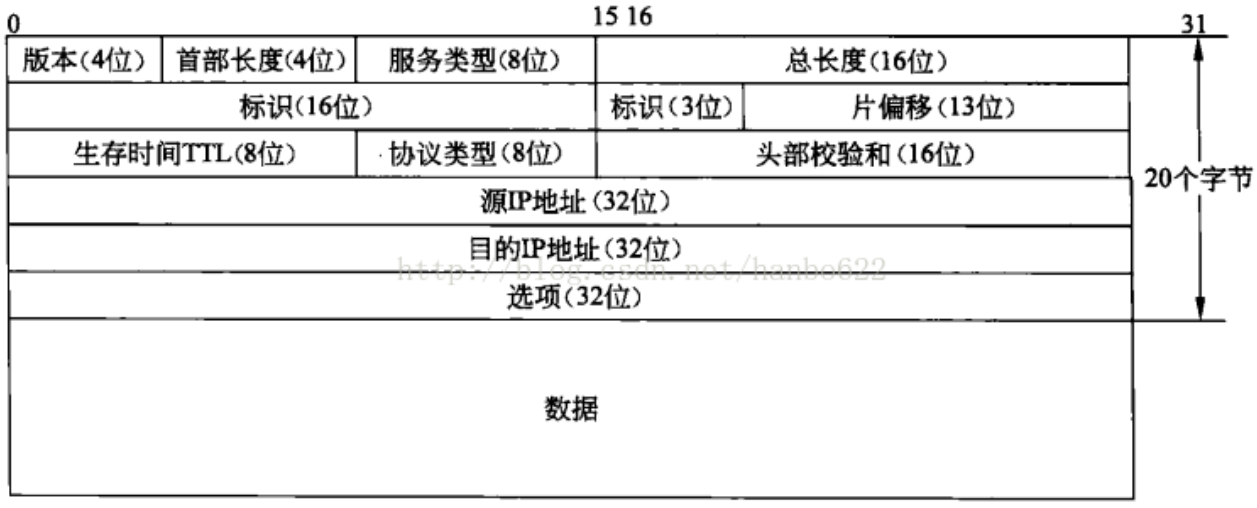
```
//以太帧首部-14字节
typedef struct ehter_header {
    u_char    ether_dhost[6];    //目的MAC地址
    u_char    ether_shost[6];    //源MAC地址
    u_short   ether_type;        //帧类型
}ether_header;
```

ARP帧

ARP报文部分承接以太网帧首部，在以太帧首部类型字段0X0806之后的结构如上图，本次实验沿用之前实验设计，将以太帧首部也放入到ARP帧内容中进行处理。

```
//ARP帧内容-14+28字节
typedef struct ARP_frame {
    ether_header header;     //以太帧首部
    u_short hardware;        //硬件类型
    u_short protocol;        //协议类型
    u_char hardware_size;    //硬件地址长度
    u_char protocol_size;    //协议地址长度
    u_short opcode;          //操作码
    u_char sender_mac[6];    //源MAC地址
    ip_address sender_ip;    //源IP地址
    u_char target_mac[6];    //目的MAC地址
    ip_address target_ip;    //目的IP地址
}ARP_frame;
```

IP数据包首部

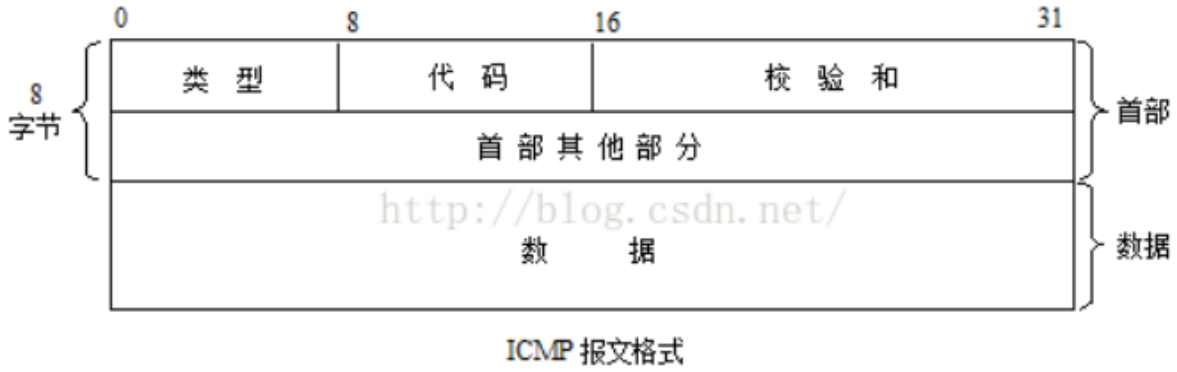


IPv4数据包首部的范围为上图右侧所示20字节（不包括选项）。

```
//4byte ip地址
typedef struct ip_address {
    u_long ip;
}ip_address;
//IPv4首部
typedef struct ip_header {
```

```
    u_char  ver_ihl; // 版本号(4 bits) + 首部长度(4 bits)
    u_char  tos;     // 服务类型
    u_short tlen;    // 总长度
    u_short identification; // 标识
    u_short flags_fo; // 标志(3 bits) + 片偏移(13 bits)
    u_char  ttl;      // 生存时间
    u_char  proto;    // 协议类型
    u_short crc;      // 头部校验和
    ip_address  saddr; // 源IP地址
    ip_address  daddr; // 目的IP地址
}ip_header;
```

ICMP首部



ICMP 报文格式

首部其他部分在一般查询报文中为2字节的标识符和2字节的序号，我们按照这种结构来定义ICMP首部结构体，但由于我们只需要在ICMP差错报文中的超时报文部分对ICMP报文范围内的数据进行修改，因此“首部其他部分”只要按照协议设计赋值为全0。

```
//ICMP报文
typedef struct icmp_header {
    u_char type;    //类型
    u_char code;    //代码
    u_short crc;    //校验和
    u_short identification;//标识符
    u_short seq;    //seq序号
}icmp_header;
```

数据包结构

沿用之前的实验设计，将以太帧首部和IP数据包首部两个数据结构合并为一个数据结构，方便统一处理。

```
//数据包结构体
typedef struct data_packet {
    ether_header etherHeader;   //以太帧首部
    ip_header ipHeader;         //ip首部
}data_packet;
```

路由表

1. 路由表表项

   表项记录了目的网络、子网掩码和下一跳地址等信息，同时记录了该表项下一跳地址对应的接口，使用一个flag来记录表项属性是直连还是静态。

```
//路由表表项
class router_table_entry {
public:
    u_long target_net;      //目的网络
    u_long mask;            //掩码
    u_long next_jump;       //下一跳地址
    u_int dev_ip_index;     //对应接口
    u_int flag;             //标记表项属性
    router_table_entry() {
        memset(this, 0, sizeof(*this));
    }
```

2. 路由表

路由表使用vector来存储路由表项，函数的具体实现在之后说明。

```
//路由表结构
class router_table {
public:
    //使用vector来存储表项
    vector<router_table_entry> routerTable;
    router_table();
    //插入路由表表项
    void router_table_add(router_table_entry entry);
    //删除路由表表项
    void router_table_delete(int i);
    //打印路由表
    void print();
    //查找路由表
    u_long router_table_search(u_long ip);
};
```

ARP缓存表

1. ARP缓存表表项

在实际情况下，ARP缓存表的表项需要定义生存时间，以避免经过长时间后IP与MAC和实际情况不对应的问题。本次实验中路由比较简单，没有设置这一属性。

```
//ARP缓存表表项
class arp_table_entry {
public:
    u_long ip;
    u_char mac[6];
    arp_table_entry() {
        memset(this, 0, sizeof(*this));
    }
};
```

2. ARP缓存表

同样是使用vector来保存ARP缓存表表项，此外由于查找时IP地址与表项中的信息是精确匹配的，因此不需要额外的排序实现。

```
//ARP缓存表
class arp_table {
public:
    vector<arp_table_entry> arpTable;
    //打印ARP缓存表
    void print();
    //插入缓存表表项
    void arp_table_add(u_long ip);
    //查找缓存表
    bool arp_table_search(u_long ip, u_char m[6]);
};
```

函数实现

路由表相关函数

1. 路由表插入

   通过自定义比较函数来完成在路由表插入时候的排序，以便在匹配路由表时只要找到第一个符合项就能返回，提高查找效率。

```
bool cmp(const router_table_entry& a, const
router_table_entry& b) {
    //网络地址相同比较掩码中1的位数（大小）
    if (a.target_net == b.target_net) {
        //掩码更大的在前
        return a.mask > b.mask;
    }
    //否则网络地址大的在前
    else return a.target_net > b.target_net;
}

//in class router_table
//插入路由表表项
void router_table_add(router_table_entry entry) {
    routerTable.push_back(entry);
    sort(routerTable.begin(), routerTable.end(), cmp);
    return;
```

2. 路由表项删除

对于直接投递的项不允许删除

```
//删除路由表表项
    void router_table_delete(int i) {
        if (i > routerTable.size()) {
            cout << "错误的编号" << endl;
            return;
        }
        if (routerTable[i - 1].flag & FLAG_DIR) {
            cout << "直接投递项不可删除" << endl;
            return;
        }
        routerTable.erase(routerTable.begin() + i - 1);
        return;
    }
```

3. 路由表项查找

将传入的IP地址与表项的掩码相与，得到的结果若与该项的网络地址相同，由于插入时进行了排序，说明找到了最长匹配的表项，返回该项记录的下一跳地址。

```
    //查找路由表
    u_long router_table_search(u_long ip) {
        for (int i = 0; i < routerTable.size(); i++) {
            if (routerTable[i].target_net == (ip &
routerTable[i].mask)) {
                return routerTable[i].next_jump;
            }
        }
        return -1;
    }
```

ARP缓存表相关函数

1. ARP表项插入

ARP表项插入是被动的，只有在检测到ARP缓存表中不存在对应要投递的IP的MAC项时才会执行此动作。

```
```

```
//in class arp_table
void arp_table_add(u_long ip) {
    arp_table_entry e;
    e.ip = ip;
    u_char temp[6];
    //构造ARP请求包
    ARP_frame ARPframe;
    ARP_frame* RecFrame;
    struct pcap_pkthdr* arppkt_header;
    const u_char* arppkt_data;
    for (int i = 0; i < 6; i++)
    {
        //以太帧源MAC地址为本机物理地址
        ARPframe.header.ether_shost[i] = dev_mac[i];
        //以太帧目的MAC地址为ff:ff:ff:ff:ff:ff的广播地址
        ARPframe.header.ether_dhost[i] = 0xff;
        ARPframe.sender_mac[i] = dev_mac[i];
        ARPframe.target_mac[i] = 0x00;
    }
    //源IP地址为本网卡设备IP地址
    strncpy((char*)&ARPframe.sender_ip.ip, (char*)(d-
>addresses->addr->sa_data + 2), 4);
    u_char addr = 0;
    //目的IP地址为传入的请求IP
    ARPframe.target_ip.ip = ip;
    ARPframe.header.ether_type = htons(0x0806);
    ARPframe.hardware = htons(0x0001);
    ARPframe.protocol = htons(0x0800);
    ARPframe.hardware_size = 6;
    ARPframe.protocol_size = 4;
    ARPframe.opcode = htons(0x0001);//1表示请求包
    //打印和写入信息
    time(&rawtime);
    ptminfo = localtime(&rawtime);
    printf("[%02d:%02d:%02d][发送]:ARP request dest desIP=",
ptminfo->tm_hour, ptminfo->tm_min, ptminfo->tm_sec);
    print_ip(ip);
    printf("\n");
    write_packet(fp, (data_packet*)&ARPframe, 1);
```

```
    //发送ARP请求
    pcap_sendpacket(adhandle, (u_char*)&ARPframe,
sizeof(ARPframe));
    //循环捕获ARP应答
    while (1)
    {
        int rtn = pcap_next_ex(adhandle, &arppkt_header,
&arppkt_data);
        switch (rtn)
        {
        case -1:
            cout << "捕获错误" << endl;
            break;
        case 0:
            cout << "未捕获到数据报" << endl;
            break;
        default:
            //收到了数据包
            RecFrame = (ARP_frame*)arppkt_data;
            if (ntohs(RecFrame->header.ether_type) ==
0x0806) {
                if ((ntohs(RecFrame->protocol) == 0x0800)
                    && (ntohs(RecFrame->opcode) == 0x0002)
                    && (RecFrame->target_ip.ip ==
ARPframe.sender_ip.ip)
                    && (RecFrame->sender_ip.ip ==
ARPframe.target_ip.ip)) {
                    //收到的是ARP响应包且响应包源IP是请求包目的IP,
响应包目的IP是本机IP
                    //读取并保存该包中存储的源MAC地址
                    for (int i = 0; i < 6; i++) {
                        e.mac[i] = RecFrame->sender_mac[i];
                    }
                    //打印和写入信息
                    time(&rawtime);
                    ptminfo = localtime(&rawtime);
                    printf("[%02d:%02d:%02d][接收]:ARP reply
desIP:", ptminfo->tm_hour, ptminfo->tm_min, ptminfo-
>tm_sec);
```

```
                    print_ip(ARPframe.target_ip.ip);
                    printf("  desMAC:");
                    print_mac(RecFrame->sender_mac);
                    printf("\n");
                    write_packet(fp, (data_packet*)RecFrame,
0);
                    goto FINISH;
                }
            }
        }
    }
    FINISH:
    //把新建立的ARP表项插入缓存表中
    arpTable.push_back(e);
}
```

2. ARP查询

本查询比较简单，遍历ARP缓存表，只要找到了表项中的IP地址与传入的IP地址相同的项，就把该项的MAC地址保存到传入的地址中。

```
//in class arp_table
bool arp_table_search(u_long ip, u_char m[6]) {
    for (int i = 0; i < arpTable.size(); i++) {
        if (arpTable[i].ip == ip) {
            memcpy(m, arpTable[i].mac, 6);
            return 1;
        }
    }
    return 0;
}
```

校验和相关函数

1. 校验和计算

IP数据包首部和ICMP报文首部中都存在校验和，用于保证对应部分内容的完整性，这两部分的校验和计算逻辑一致，只是计算范围分别是20字节和8字节，本次实验沿用了之前使用过的校验和计算函数，仅展示IP数据包中校验和的计算函数

```
```

```
void calChecksum(data_packet* pkt) {
    //计算校验和
    pkt->ipHeader.crc = 0;
    u_long sum = 0;
    u_short* pointer = (u_short*)&pkt->ipHeader;
    //对ip首部字段按16位求和
    for (int i = 0; i < 10; i++)
    {
        u_short temp = pointer[i];
        sum += temp;
    }
    //超出16位
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    //取反
    pkt->ipHeader.crc = (u_short)~sum;
}
```

2. 校验和验证函数

   校验和验证有多种方法，本函数使用的方法是：保存收到数据包的校验和，后将该部分置位为0，重新计算一遍校验和，如果与原本的校验和相等，则说明校验通过

```
bool checkChecksum(data_packet* pkt) {
    //检验校验和
    u_int sum = pkt->ipHeader.crc;
    pkt->ipHeader.crc = 0;
    calChecksum(pkt);
    //与原本校验和一致则通过
    if ((u_short)sum == pkt->ipHeader.crc)
        return true;
    else return false;
}
```

转发线程函数

该线程将自程序开始时一直运行，直至用户退出程序。

在while(1)循环中，接收到数据包后按照程序设计中约定的范围过滤，进而判断是否要返回超时报文、包校验和是否错误、是否是广播包、是否在路由表找到下一跳的表项，通过这些验证仍在往下执行，则意味着是正常的包，等待转发。根据查询路由表得到的表项属性，分辨是直接投递还是有下一跳地址的情况，并进行对应的ARP缓存表查询，ARP缓存中未找到则进行一次ARP请求。等待响应返回或直接找到表项后，修改数据包以太帧的两个地址和IP数据包部分的TTL，重新计算IP数据包部分的校验和，将包发出。

```
DWORD WINAPI forwardThread(LPVOID lparam) {
    while (1) {
        //循环直至接收到数据包
        while (1) {
            if (pcap_next_ex(adhandle, &pkt_header, &pkt_data))
                break;
        }
        ether_header* eheader = (ether_header*)pkt_data;
        if (memcmp(&(eheader->ether_dhost), &dev_mac, 6) == 0) {
            //包的以太帧目的MAC地址是本机
            if (ntohs(eheader->ether_type) == 0x0800) {
                //包的以太帧类型表明是IP数据包
                data_packet* data = (data_packet*)pkt_data;
                //打印和写入相关信息
                time(&rawtime);
                ptminfo = localtime(&rawtime);
                printf("[%02d:%02d:%02d][接收]:IP数据包 sourceIP:",
ptminfo->tm_hour, ptminfo->tm_min, ptminfo->tm_sec);
                print_ip(data->ipHeader.saddr.ip);
                printf("  targetIP:");
                print_ip(data->ipHeader.daddr.ip);
                printf("  sourceMAC:");
                print_mac(data->etherHeader.ether_shost);
                printf("  targetMAC:");
                print_mac(data->etherHeader.ether_dhost);
                printf("\n");
                write_packet(fp,data, 0);

                //如果接收到的IP数据包的TTL<=1，说明要返回超时报文
                if (data->ipHeader.ttl <= 1) {
                    ICMP_packet_send(11, 0);
                    continue;
```

```
                }

                //如果接收到的IP数据包的校验和错误，丢弃
                if (!checkChecksum(data)) {
                    cout << "校验和错误" << endl;
                    continue;
                }

                if (data->ipHeader.daddr.ip != dev_ip[0][1] &&
data->ipHeader.daddr.ip != dev_ip[0][0]) {
                    //包的目的IP不是本机接口中任意一个IP
                    if (not_broadcast(data)) {
                        //不是广播包
                        //
                        //查找路由表中目的IP的下一跳地址，未找到则忽略
                        u_long tip =
routerTable.router_table_search(data->ipHeader.daddr.ip);
                        if (tip == -1) {
                            cout << "目标ip未找到" << endl;
                            continue;
                        }

                        u_char* mac = new u_char[6];
                        if (tip == 0) {
                            //下一跳地址为0表明是直接投递
                            if (!arpTable.arp_table_search(data-
>ipHeader.daddr.ip, mac)) {
                                //在ARP缓存表中没有找到包目的IP对应项
                                arpTable.arp_table_add(data-
>ipHeader.daddr.ip);

                            }
                            arpTable.arp_table_search(data-
>ipHeader.daddr.ip, mac);
                        }
                        else {
                            //下一跳地址有意义
                            if (!arpTable.arp_table_search(tip,
mac)) {
```

```
                                //在ARP缓存表中没有找到下一跳地址的对应项
                                arpTable.arp_table_add(tip);
                            }
                            arpTable.arp_table_search(tip, mac);
                        }
                        //修改包的以太帧的源MAC地址为本网卡设备MAC地址，目的
MAC地址为查询ARP缓存表得到的物理地址
                        memcpy(data->etherHeader.ether_shost, data-
>etherHeader.ether_dhost, 6);
                        memcpy(data->etherHeader.ether_dhost, mac,
6);
                        //包的TTL减一
                        data->ipHeader.ttl -= 1;
                        //重新计算校验和
                        calChecksum(data);
                        int len = ntohs(data->ipHeader.tlen);
                        data_packet* curdata =
(data_packet*)malloc(len + 14);
                        memcpy(curdata, data, len + 14);
                        if (pcap_sendpacket(adhandle,
(u_char*)data, len+14) == 0){
                            //成功发送，打印和写入相关信息
                            time(&rawtime);
                            ptminfo = localtime(&rawtime);
                            write_packet(fp, curdata, 1);
                            printf("[%02d:%02d:%02d][发送]:IP数据包
sourceIP:", ptminfo->tm_hour, ptminfo->tm_min, ptminfo->tm_sec);
                            print_ip(curdata->ipHeader.saddr.ip);
                            printf("  targetIP:");
                            print_ip(curdata->ipHeader.daddr.ip);
                            printf("  sourceMAC:");
                            print_mac(curdata-
>etherHeader.ether_shost);
                            printf("  targetMAC:");
                            print_mac(curdata-
>etherHeader.ether_dhost);
                            printf("\n");
                        }
                    }
```

```
                }
            }
        }
    }
}
```

超时报文处理函数

在循环线程中受到TTL<=1的报文时需要进行超时返回，调用下述函数。

构造数据包，以太帧部分交换源目的地址，IP数据包部分修改TTL，ICMP部分type为11，code为0，之后两个字节如数据结构部分所述为全0，数据部分为发生差错的IP数据包首部和数据区的8个字节。

```
void ICMP_packet_send(u_char type, u_char code) {
    u_char* ICMPbuf = new u_char[70];
    //构造以太帧部分
    memcpy(((ether_header*)ICMPbuf)->ether_shost,
((ehter_header*)pkt_data)->ether_dhost, 6);
    memcpy(((ether_header*)ICMPbuf)->ether_dhost,
((ehter_header*)pkt_data)->ether_shost, 6);
    ((ether_header*)ICMPbuf)->ether_type = htons(0x0800);
    //构造IP数据包部分
    ip_header* icmp_ip_header = (ip_header*)(ICMPbuf + 14);
    ip_header* pkt_ip_header = (ip_header*)(pkt_data + 14);
    icmp_ip_header->ver_ihl = pkt_ip_header->ver_ihl;
    icmp_ip_header->tos = pkt_ip_header->tos;
    icmp_ip_header->tlen = htons(56);
    icmp_ip_header->flags_fo = pkt_ip_header->flags_fo;
    icmp_ip_header->ttl = 64;        //TTL更新为64
    icmp_ip_header->proto = 1;
    icmp_ip_header->saddr.ip = dev_ip[0][1];
    icmp_ip_header->daddr = pkt_ip_header->saddr;
    calChecksum((data_packet*)ICMPbuf);
    //构造ICMP部分
    icmp_header* icmp_icmp_header = (icmp_header*)(ICMPbuf + 34);
    icmp_icmp_header->type = type;
    icmp_icmp_header->code = code;
```

```
    icmp_icmp_header->identification = 0;
    icmp_icmp_header->seq = 0;
    calChecksum_icmp(icmp_icmp_header);
    //超时报告报文数据部分为发生差错的IP数据包首部和数据区的8个字节
    memcpy((u_char*)(ICMPbuf + 42), (ip_header*)(pkt_data + 14),
20);
    memcpy((u_char*)(ICMPbuf + 62), (u_char*)(pkt_data + 34), 8);
    //返回ICMP超时报告报文
    pcap_sendpacket(adhandle, (u_char*)ICMPbuf, 70);
}
```

日志写入函数

使用全局的文件指针`FILE* fp`来完成日志文件的写入。根据读取包中的一些区分字段来进行不同内容的文件写入。

```
fp = fopen("log.txt", "a+");
void write_packet(FILE* fp, data_packet* data, bool type) {
    if (type == 1)
        fprintf(fp, "[%02d:%02d:%02d][发送]:", ptminfo->tm_hour,
ptminfo->tm_min, ptminfo->tm_sec);
    else
        fprintf(fp, "[%02d:%02d:%02d][接收]:", ptminfo->tm_hour,
ptminfo->tm_min, ptminfo->tm_sec);

    u_short ethernet_type = ntohs(data->etherHeader.ether_type);
    if (ethernet_type == 0x0800) {
        in_addr addr;
        addr.s_addr = data->ipHeader.saddr.ip;
        char* temp = inet_ntoa(addr);
        fprintf(fp, "sourceIP:%s\t", temp);
        addr.s_addr = data->ipHeader.daddr.ip;
        temp = inet_ntoa(addr);
        fprintf(fp, "targetIP:%s\n", temp);
        u_char* m = data->etherHeader.ether_shost;
        fprintf(fp, "
sourceMAC:%02x:%02x:%02x:%02x:%02x:%02x\t", m[0], m[1], m[2], m[3],
m[4], m[5]);
```

```
        m = data->etherHeader.ether_shost;
        fprintf(fp, "targetMAC:%02x:%02x:%02x:%02x:%02x:%02x\n",
m[0], m[1], m[2], m[3], m[4], m[5]);
        fprintf(fp, "++++++++++++++++++++IP数据报
++++++++++++++++++++\n");
        fprintf(fp, "IP version:%d\n", data->ipHeader.ver_ihl >>
4);
        fprintf(fp, "首部长度:%d\n", (data->ipHeader.ver_ihl & 0xF) *
4);
        fprintf(fp, "服务类型%d\n", data->ipHeader.tos);
        fprintf(fp, "总长度:%d\n", data->ipHeader.tlen);
        fprintf(fp, "标识:0x%x\n", ntohs(data-
>ipHeader.identification));
        fprintf(fp, "标志:0x%x\n", ntohs(data->ipHeader.flags_fo) >>
13);
        fprintf(fp, "片偏移:%d\n", ntohs(data->ipHeader.flags_fo) &
0x1FFF);
        fprintf(fp, "ttl:%d\n", data->ipHeader.ttl);
        fprintf(fp, "头部校验和:0x%x\n", ntohs(data->ipHeader.crc));
        switch (data->ipHeader.proto) {
        case 1:
            fprintf(fp, "协议类型:ICMP\n");
            fprintf(fp, "
+++++++++++++++ICMP+++++++++++++++\n");
            icmp_header* icmpHeader = (icmp_header*)(data + 34);
            switch (ntohs(icmpHeader->type)) {
            case 0:
                if (ntohs(icmpHeader->code) == 0) {
                    fprintf(fp, "类型:Ping应答\n");
                }
                break;
            case 8:
                if (ntohs(icmpHeader->code) == 0) {
                    fprintf(fp, "类型:Ping请求\n");
                }
                break;
            case 11:
                if (ntohs(icmpHeader->code) == 0) {
                    fprintf(fp, "类型:超时，(traceroute)\n");
```

```
                }
                else if (ntohs(icmpHeader->code) == 1) {
                    fprintf(fp, "类型:超时，数据包组装\n");
                }
                break;
            default:
                break;
            }
            break;
        }
        fprintf(fp, "++++++++++++++++++++解析完成
++++++++++++++++++++\n\n");
    }
    else if (ethernet_type == 0x0806) {
        ARP_frame* arpPacket = (ARP_frame*)(data);
        u_char* m = data->etherHeader.ether_shost;
        fprintf(fp, "
sourceMAC:%02x:%02x:%02x:%02x:%02x:%02x\t", m[0], m[1], m[2], m[3],
m[4], m[5]);
        m = data->etherHeader.ether_shost;
        fprintf(fp, "targetMAC:%02x:%02x:%02x:%02x:%02x:%02x\n",
m[0], m[1], m[2], m[3], m[4], m[5]);
        fprintf(fp, "++++++++++++++++++++ARP数据报
++++++++++++++++++++\n");
        fprintf(fp, "硬件类型:%d\n", ntohs(arpPacket->hardware));
        fprintf(fp, "协议类型:0x%x\n", ntohs(arpPacket->protocol));
        switch (ntohs(arpPacket->opcode)) {
        case 1:
            fprintf(fp, "操作类型:ARP请求\n");
            break;
        case 2:
            fprintf(fp, "操作类型:应答报文\n");
            break;
        }
        m = arpPacket->sender_mac;
        fprintf(fp, "sourceMAC:%02x:%02x:%02x:%02x:%02x:%02x\n",
m[0], m[1], m[2], m[3], m[4], m[5]);
        in_addr addr;
        addr.s_addr = arpPacket->sender_ip.ip;
```

```
        char* temp = inet_ntoa(addr);
        fprintf(fp, "sourceIP:%s\n", temp);
        m = arpPacket->target_mac;
        fprintf(fp, "targetMAC:%02x:%02x:%02x:%02x:%02x:%02x\n",
m[0], m[1], m[2], m[3], m[4], m[5]);
        addr.s_addr = arpPacket->target_ip.ip;
        temp = inet_ntoa(addr);
        fprintf(fp, "targetIP:%s\n", temp);
        fprintf(fp, "++++++++++++++++++++解析完成
++++++++++++++++++++\n");
    }
}
```

主函数执行流程

1. 全局变量

```
PIP_ADAPTER_ADDRESSES pAddresses;

pcap_t* adhandle;
pcap_if_t* d;
struct pcap_pkthdr* pkt_header;
const u_char* pkt_data;
u_char dev_mac[6];

HANDLE hThread;
DWORD dwThreadId;

u_long dev_ip[2][10];
FILE* fp = nullptr;
time_t rawtime;
struct tm* ptminfo;
```

2. 获取设备列表

```
pcap_if_t* alldevs;
int inum;
int i = 0;
char errbuf[PCAP_ERRBUF_SIZE];

//获取设备列表
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING,
    NULL, &alldevs, errbuf) == -1)
{
    fprintf(stderr, "在检测设备时出现错误: %s\n", errbuf);
    exit(1);
}
```

3. 打印设备列表，并根据用户选择打开设备监听

```
//打印列表
for (d = alldevs; d; d = d->next)
{
    printf("%d. %s", ++i, d->name);
    if (d->description)
        printf(" (%s)\n", d->description);
    else
        printf(" 无可用设备! \n");
}
if (i == 0)  return -1;
printf("请输入设备编号 (1-%d):", i);
scanf("%d", &inum);
if (inum < 1 || inum > i)
{
    printf("\n超出可用范围\n");
    pcap_freealldevs(alldevs);
    return -1;
}
//跳转至想要监听的设备
for (d = alldevs, i = 0; i < inum - 1; d = d->next, i++);
//打开设备
if ((adhandle = pcap_open(d->name, // name of the device
    65536, // portion of the packet to capture
           // 65536 guarantees that the whole packet will
```

```
            // be captured on all the link layers
    PCAP_OPENFLAG_PROMISCUOUS, // promiscuous mode
    1000, // read timeout
    NULL, // authentication on the remote machine
    errbuf // error buffer
)) == NULL)
{
    fprintf(stderr,
        "\n%s设备不支持！%\n",
        d->name);
    pcap_freealldevs(alldevs);
    return -1;
}
printf("\n正在 %s 上监听...\n", d->description);
```

4. 获取绑定在该设备上的所有IP地址

   因为路由器设备使用的是一个网卡绑定多个IP地址作为多个接口存在，路由器程序需要了解到绑定的所有IP和对应的子网掩码。

```
int dev_ip_index = 0;
for (pcap_addr* a = d->addresses; a; a = a->next) {
    if (a->addr->sa_family == AF_INET) {
        dev_ip[0][dev_ip_index] = (((struct sockaddr_in*)a-
>addr)->sin_addr.s_addr);
        dev_ip[1][dev_ip_index] = (((struct sockaddr_in*)a-
>netmask)->sin_addr.s_addr);

        dev_ip_index++;
    }
}
```

5. 获取监听的设备的MAC地址

   本过程通过调用包含在`iphlpapi.h`的`GetAdaptersAddresses`函数来定位到选择的适配器，使用的虚拟环境只有一个网卡设备，故直接通过该函数定位到网卡之后就可以使用指针来得到其物理地址。

```
 //获取网卡设备MAC地址
pAddresses = nullptr;
ULONG outbuflen = 0;
GetAdaptersAddresses(AF_UNSPEC, 0, NULL, pAddresses,
&outbuflen);
pAddresses = (IP_ADAPTER_ADDRESSES*)malloc(outbuflen);
GetAdaptersAddresses(AF_INET, NULL, NULL, pAddresses,
&outbuflen);
//记录其物理地址
for (int i = 0; i < 6; i++)
    dev_mac[i] = pAddresses->PhysicalAddress[i];
```

6. 初始化路由表，添加直接投递项

通过之前步骤得到的绑定在该设备上的所有IP地址，添加路由表直接投递项。

```
//初始化路由表，添加D项
for (int i = 0; i < dev_ip_index; i++) {
    router_table_entry t;
    t.target_net = dev_ip[0][i] & dev_ip[1][i];
    t.mask = dev_ip[1][i];
    t.next_jump = 0;
    t.dev_ip_index = i;
    t.flag = FLAG_DIR;
    routerTable.router_table_add(t);
}
```

7. 创建转发线程，并在主函数中使用无限循环接收用户输入的命令。

```
hThread = CreateThread(NULL, NULL, forwardThread,
LPVOID(&routerTable), 0, &dwThreadId);
int operation;
while (1)
{
    printf("输入后续操作\n1.打印路由表\n2.添加路由表项\n3.删除路由表
项\n4.打印arp缓存\n");
    cin >> operation;
    if (operation == 1) {
        routerTable.print();
```

```
    }
    else if (operation == 2) {
        router_table_entry t;
        char userin_net[20];
        char userin_mask[20];
        char userin_nj[20];
        printf("输入目的网络: ");
        cin >> userin_net;
        printf("输入掩码: ");
        cin >> userin_mask;
        printf("输入下一跳: ");
        cin >> userin_nj;
        t.mask = inet_addr(userin_mask);
        t.target_net = inet_addr(userin_net);
        t.next_jump = inet_addr(userin_nj);
        for (int i = 0; i < dev_ip_index; i++) {
            if ((dev_ip[0][i] & dev_ip[1][i]) == (t.mask &
t.target_net)) {
                t.dev_ip_index = i;
                break;
            }
        }
        routerTable.router_table_add(t);
    }
    else if (operation == 3) {
        printf("输入要删除的表项编号: ");
        int index;
        cin >> index;
        routerTable.router_table_delete(index);
    }
    else if (operation == 0) {
        break;
    }
    else if (operation == 4) {
        arpTable.print();
    }
    else {
        printf("无效操作码\n");
    }
```

```
}
```

程序测试

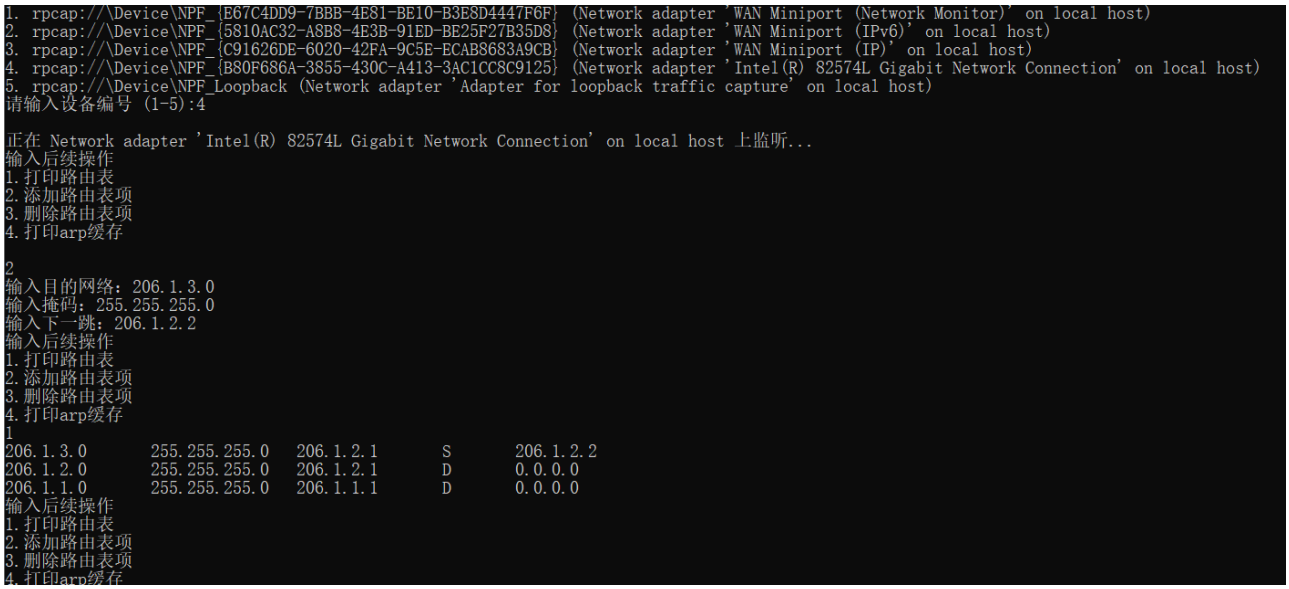
环境准备

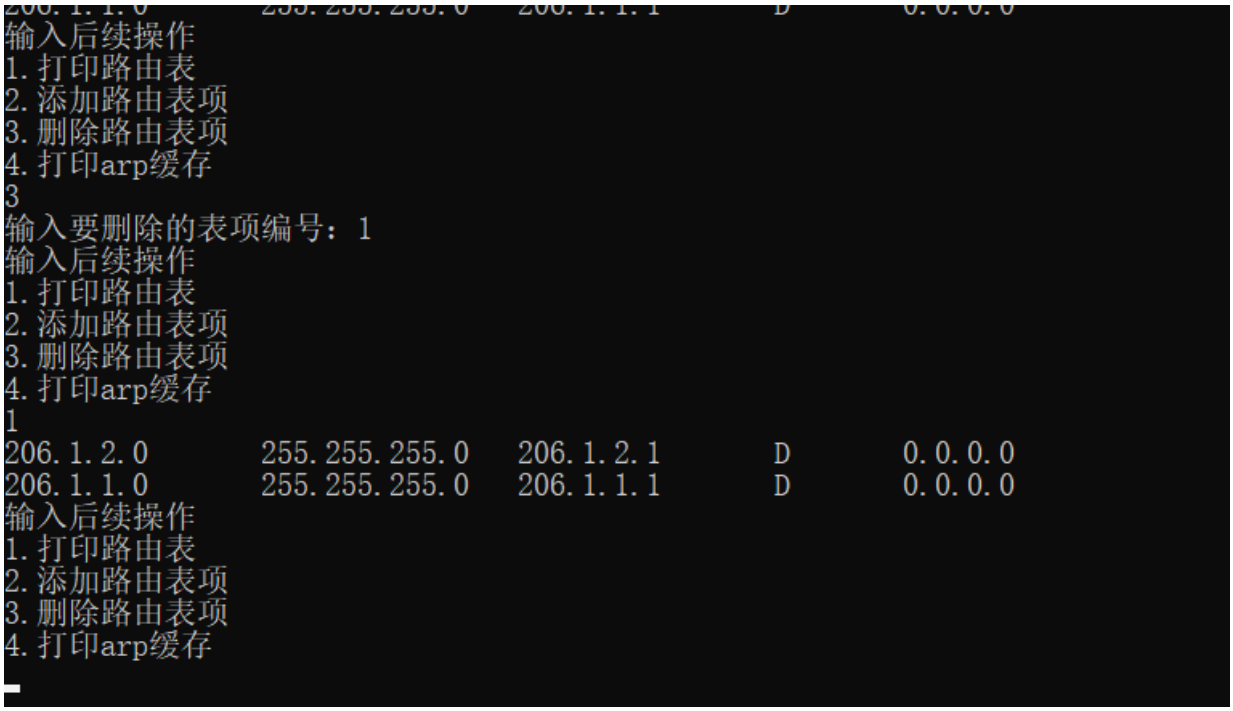
使用部署在VMware上的四台搭载了Windows server 2019系统的虚拟机完成路由程序的测试：

PC0	PC1	PC2	PC3
206.1.1.2	206.1.1.1 206.1.2.1	206.1.2.2 206.1.3.1	206.1.3.2

其中，PC0和PC3作为两端的主机，PC1中放置编写好的路由器程序，PC2开启路由转发服务充当路由器。

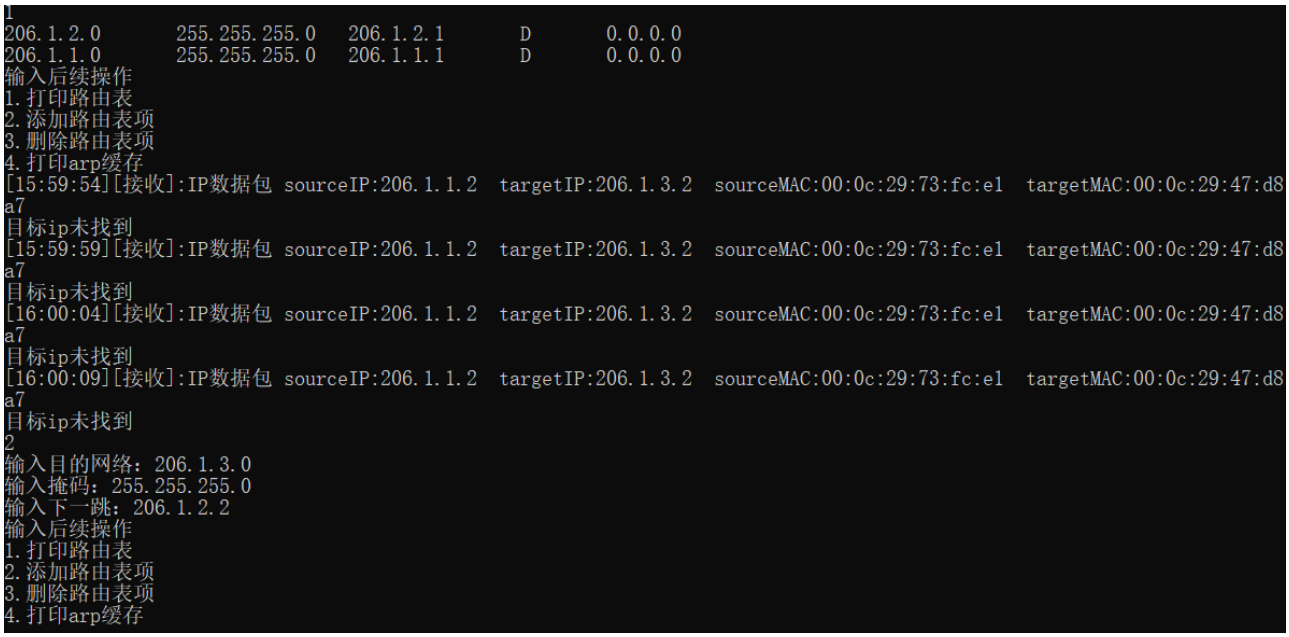
在PC1上运行路由器程序，可以在如下的命令行界面中添加、删除路由表项，输入2选择添加路由表项，依次输入目的网络、子网掩码和下一跳地址，即可插入路由表；删除路由表时，只允许删除非直连项，输入对应索引后即可删除。



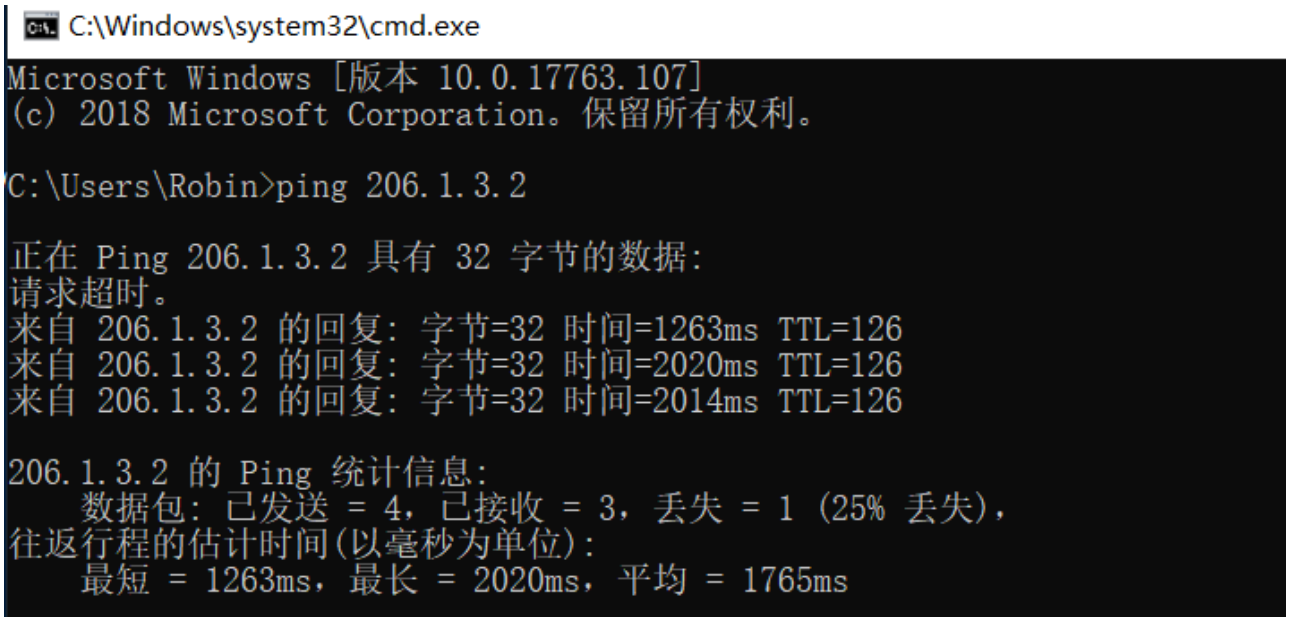


重新添加完路由表项后，即可开始尝试连通性测试。

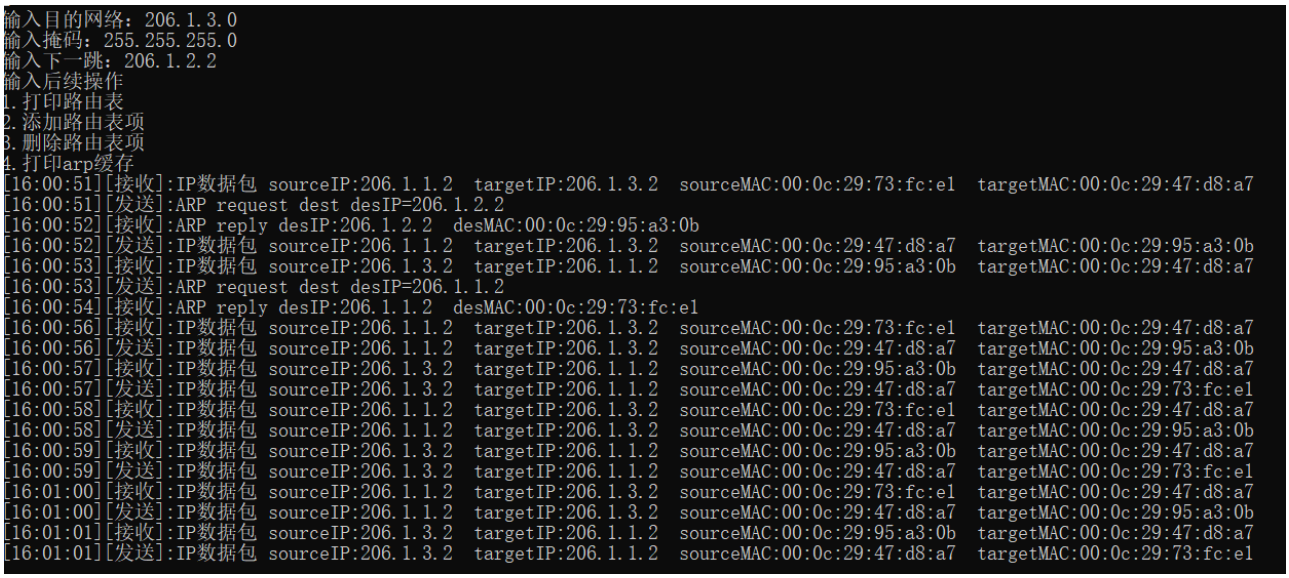
Ping



使用PC0去Ping PC3，得到结果如下：



其中，第一次超时是路由器进行ARP请求等待时间过长导致的，路由器程序界面的信息打印如下：



此外，日志文件中也有更为详细的记录：

```
log - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
ttl:128
头部校验和:0x68b1
协议类型:ICMP
        ++++++++++++++++ICMP++++++++++++++++
++++++++++++++++++++解析完成++++++++++++++++++++

[16:00:51][接收]:sourceIP:206.1.1.2        targetIP:206.1.3.2
            sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:73:fc:e1
++++++++++++++++++++IP数据报++++++++++++++++++++
IP version:4
首部长度:20
服务类型0
总长度:15360
标识:0x320a
标志:0x0
片偏移:0
ttl:128
头部校验和:0x68b0
协议类型:ICMP
        ++++++++++++++++ICMP++++++++++++++++
++++++++++++++++++++解析完成++++++++++++++++++++

[16:00:51][发送]:              sourceMAC:00:0c:29:47:d8:a7      targetMAC:00:0c:29:47:d8:a7
++++++++++++++++++++ARP数据报++++++++++++++++++++
硬件类型:1
协议类型:0x800
操作类型:ARP请求
sourceMAC:00:0c:29:47:d8:a7
```

tracert

使用PC0 tracert PC3得到如下结果，说明能够正常执行tracert：

```
C:\Users\Robin>tracert 206.1.3.2

通过最多 30 个跃点跟踪到 206.1.3.2 的路由

  1   940 ms  1008 ms  1009 ms  WIN-L1N1J62LN64 [206.1.1.1]
  2  2013 ms  2014 ms  2015 ms  206.1.2.2
  3  2018 ms  2036 ms  2019 ms  206.1.3.2

跟踪完成。
```

对应的路由器程序中的信息打印：

```
16:04:16][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:16][发送]:IP数据包 sourceIP:206.1.1.1  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:17][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:17][发送]:IP数据包 sourceIP:206.1.1.1  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:18][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:18][发送]:IP数据包 sourceIP:206.1.1.1  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:19][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.1.1  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:19][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.1.1  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:19][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:19][发送]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:95:a3:0b
16:04:20][接收]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:95:a3:0b  targetMAC:00:0c:29:47:d8:a7
16:04:20][发送]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:21][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:21][发送]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:95:a3:0b
16:04:22][接收]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:95:a3:0b  targetMAC:00:0c:29:47:d8:a7
16:04:22][发送]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:23][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:23][发送]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:95:a3:0b
16:04:24][接收]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:95:a3:0b  targetMAC:00:0c:29:47:d8:a7
16:04:24][发送]:IP数据包 sourceIP:206.1.2.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:25][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:25][发送]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:95:a3:0b
16:04:26][接收]:IP数据包 sourceIP:206.1.3.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:95:a3:0b  targetMAC:00:0c:29:47:d8:a7
16:04:26][发送]:IP数据包 sourceIP:206.1.3.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:73:fc:e1
16:04:27][接收]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:73:fc:e1  targetMAC:00:0c:29:47:d8:a7
16:04:27][发送]:IP数据包 sourceIP:206.1.1.2  targetIP:206.1.3.2  sourceMAC:00:0c:29:47:d8:a7  targetMAC:00:0c:29:95:a3:0b
16:04:28][接收]:IP数据包 sourceIP:206.1.3.2  targetIP:206.1.1.2  sourceMAC:00:0c:29:95:a3:0b  targetMAC:00:0c:29:47:d8:a7
```

仓库链接

Network-Technology-and-Application/lab5 at main · shockstove/Network-Technology-and-Application (github.com)