# 网络技术与应用课程实验报告

## 实验名称：简化路由器程序设计

学号：2211489 姓名：冯佳明 专业：物联网工程

一、实验要求：简单路由器程序设计

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. 需要给出方式路由表的建立和形成
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
5. 完成的程序须通过现场测试，并讲解和报告自己的设计思路、开发和实现过程、测试方法和过程

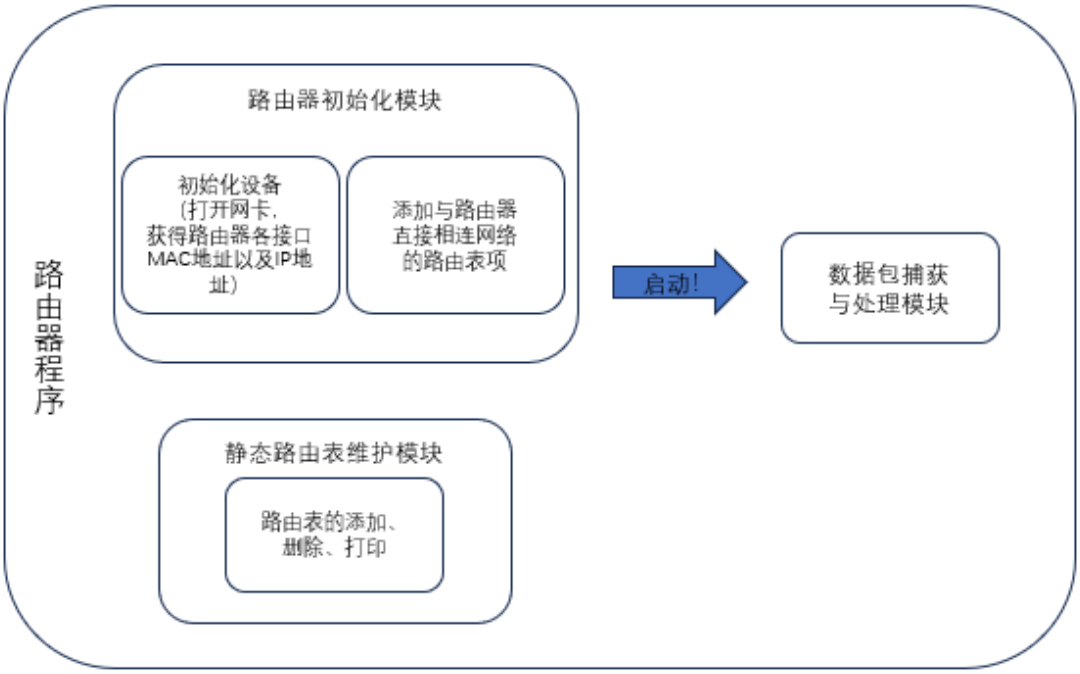
二、设计思路

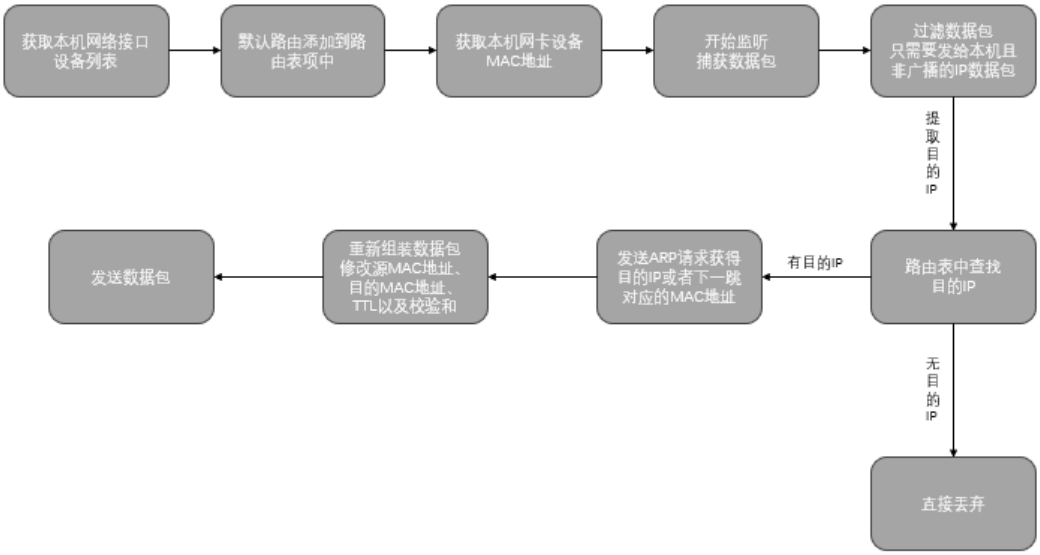
本次实验利用 Npcap 编写一个简单的路由器程序，在之前的实验基础上，实现了以下功能：

- 静态路由表维护：程序提供静态路由的添加、删除和打印的功能
- IP 数据包处理：包括 IP 数据包的接收、IP 数据包的选路和 IP 数据包的发送等工作
- ARP 请求与解析：在将一个 IP 数据包发送到下一跳之前，路由处理程序会发送 ARP 请求获得下一跳的物理 MAC 地址，并通过解析 ARP 响应报文提取
- 处理 IP 数据包的 TTL 值：IP 数据包中的 TTL 控制数据包在网络中的停留时间，因此数据包经过时，路由器程序需要判断 TTL 的值，抛弃 TTL 小于等于 0 的数据包，并将需要转发的数据包的 TTL 减 1.
- 重新计算 IP 数据包的头部校验和：路由器程序需要对 TTL 进行处理，发送的 IP 数据包与接收时的 IP 数据包头部会存在差异，因此需要重新计算 IP 数据包的头部校验和。

按照简单路由程序的实现要求，本程序可以分为路由器初始化、静态路由表维护以及数据包捕获与处理三大模块。

- 初始化模块负责初始化设备，包括打开网卡、获得路由器各接口的 MAC 地址以及 IP 地址，添加与路由器直接相连网络的路由表项，并启动相应的数据包捕获与处理模块
- 静态路由表维护模块负责完成路由表的添加、删除以及打印
- 数据包捕获与处理模块负责捕获流经本路由器的数据包并按照路由协议进行处理




三、实验步骤

（一） 报文结构设计

对于以太网帧首部、IP 数据报首部、IP 数据报报文、ARP 帧的设计，采用前几次实验的设计格式；

对于路由表，定义 router_table 结构体

1. 以太网帧首部

```
18  //定义帧首部
19  typedef struct FrameHeader_t {
20      BYTE DesMAC[6];      //目的地址
21      BYTE SrcMAC[6];      //源地址
22      WORD FrameType;      //帧类型
23  }FrameHeader_t;
```

2. IP 数据报首部

```
25  //定义IP首部
26  typedef struct IPHeader_t {
27      BYTE Ver_HLen;       //IP版本和头部长度
28      BYTE TOS;            //服务类型
29      WORD TotalLen;       //总长度
30      WORD ID;             //标识
31      WORD Flag_Segment;   //片偏移
32      BYTE TTL;            //生存时间
33      BYTE Protocol;       //协议
34      WORD Checksum;       //首部校验和
35      ULONG SrcIP;         //源IP
36      ULONG DstIP;         //目的IP
37  }IPHeader_t;
```

3. IP 数据报报文

```
39  //定义包含帧首部和IP首部的数据包
40  typedef struct Data_t {
41      FrameHeader_t FrameHeader;
42      IPHeader_t IPHeader;
43  }Data_t;
```

4. ARP 帧

```
45  //定义ARP帧
46  typedef struct ARPFrame_t {
47      FrameHeader_t FrameHeader;
48      WORD HardwareType;   //硬件类型
49      WORD ProtocolType;   //协议类型
50      BYTE HLen;           //硬件地址长度
51      BYTE PLen;           //协议地址长度
52      WORD Operation;      //操作
53      BYTE SendHa[6];      //源MAC地址
54      DWORD SendIP;        //源IP地址
55      BYTE RecvHa[6];      //目的MAC地址
56      DWORD RecvIP;        //目的IP地址
57  }ARPFrame_t;
```

5. 路由表结构

```
54    //定义路由表
55    typedef struct router_table {
56        ULONG netmask;          //网络掩码
57        ULONG desnet;           //目的网络
58        ULONG nexthop;          //下一站路由
59    }router_table;
```

（二） 路由操作函数设计

1. 路由匹配函数：实现最长匹配的路由查找功能。

使用 for 循环遍历路由表，使用子网掩码和目的 IP 地址进行与运算，运算结果与路由表中的目的网络进行比较，如果匹配，则表明目的 IP 属于该项的网络。

选择所有满足匹配的路由项中，目的网络地址最长的一项，返回下一跳的 IP 地址。如果没有找到匹配项，返回 0xffffffff。

```
// 路由选择函数，最长匹配，返回值为下一跳的IP地址
ULONG search(router_table* t, int tLength, ULONG DesIP)
{
    ULONG best_desnet = 0;  //最优匹配的目的网络
    int best = -1;
    for (int i = 0; i < tLength; i++)
    {
        if ((t[i].netmask & DesIP) == t[i].desnet)
        {
            if (t[i].desnet >= best_desnet)//最长匹配
            {
                best_desnet = t[i].desnet;
                best = i;
            }
        }
    }

    if (best == -1)
        return 0xffffffff;
    else
        return t[best].nexthop;
}
```

2. 增加路由表项函数：向路由表中添加新的路由项。

路由表会按前缀长度排序，前缀长度长的排在前面。因此，引入 getPrefixLength 函数用于计算子网掩码的前缀长度。

additem 函数像路由表中添加新的路由项，首先会检查路由表是否已满，如果路由表已满，则不能添加。然后检查路由表中是否存在相同的路由项，如果存在，则不能添加。

接下来增加路由表项，首先计算新路由项的前缀长度，然后通过循环遍历当前路由表，比较已存在项和新项的前缀长度。前缀长度较长的路由项排在前面，找到合适的位置后，将新项插入找到的位置并更新路由表的长度。

```cpp
// 计算子网掩码的前缀长度
int getPrefixLength(int netmask)
{
    int length = 0;
    while (netmask)
    {
        netmask &= (netmask - 1);
        length++;
    }
    return length;
}
```

```cpp
// 向路由表中添加项（插入时按前缀长度排序）
bool additem(router_table* t, int& tLength, router_table item)
{
    if (tLength == ROUTER_TABLE_SIZE)  // 路由表满，不能添加
    {
        cout << "添加失败！路由表项已满，请先删除不常用的表项！" << endl;
        return false;
    }

    // 检查路由表中是否已存在相同的路由项
    for (int i = 0; i < tLength; i++) {
        if (t[i].desnet == item.desnet && t[i].netmask == item.netmask && t[i].nexthop == item.nexthop)
        {
            cout << "添加失败！已存在相同路由表项！" << endl;
            return false;  // 如果已存在完全相同的路由项，返回false
        }
    }

    // 获取新路由项的前缀长度
    int itemPrefixLength = getPrefixLength(item.netmask);

    // 将新项插入到路由表中，并保持路由表按前缀长度排序
    int insertIndex = tLength;
    for (int i = 0; i < tLength; i++) {
        // 通过比较前缀长度来决定插入位置，前缀长度长的排在前面
        int existingPrefixLength = getPrefixLength(t[i].netmask);
        if (existingPrefixLength < itemPrefixLength) {
            insertIndex = i;
            break;
        }
    }

    // 将元素插入到找到的合适位置
    for (int i = tLength; i > insertIndex; i--) {
        t[i] = t[i - 1];  // 向后移动元素，为新项腾出空间
    }

    t[insertIndex] = item;  // 将新项放到正确的位置
    tLength++;  // 更新路由表的长度
    cout << "添加成功！" << endl;

    return true;
}
```

3. 删除路由表项函数：

在开始删除操作前，首先检查路由表是否为空，如果路由表为空，则无法删除。

此外，如果删除的内容是默认路由表项（即第一条路由和第二条路由），则输出警告，提示无法删除。

遍历路由表，找到与给的 index 相匹配的路由项，进行删除操作并更新路由表长度。如果没有找到，则删除失败。

```
//从路由表中删除项
bool deleteitem(router_table* t, int& tLength, int index)
{
    if (tLength == 0)   //路由表空，不能删除
    {
        cout << "[警告] 删除失败！当前路由表为空表" << endl;
        return false;

    }

    if (index == 0 || index == 1)
    {
        cout << "[警告] 删除失败！不能删除默认路由" << endl;
        return false;
    }

    for (int i = 0; i < tLength; i++)
    {
        if (i == index)   //删除以index索引的表项
        {
            for (; i < tLength - 1; i++)
                t[i] = t[i + 1];
            tLength = tLength - 1;
            cout << "[删除] 删除成功！当前路由表为：" << endl;
            print_rt(t, tLength);

            return true;
        }
    }

    cout << "[警告] 删除失败！当前路由表中不存在该项！" << endl;
    return false;   //路由表中不存在该项则不能删除
}
```

4. 设置校验和函数

在计算开始之前，先清空旧的校验和，然后以 16 位为一组进行累加，每次累加都检查是否有溢出，如果发现存在溢出，就进行回卷。计算完成后对最终计算结果按位取反并存储。

```
// 设置校验和
void setchecksum(Data_t* temp)
{
    temp->IPHeader.Checksum = 0;
    unsigned int sum = 0;
    WORD* t = (WORD*)&temp->IPHeader;    //每16位为一组
    for (int i = 0; i < sizeof(IPHeader_t) / 2; i++)
    {
        sum += t[i];
        while (sum >= 0x10000)       //如果溢出，则进行回卷
        {
            int s = sum >> 16;
            sum -= 0x10000;
            sum += s;
        }
    }
    temp->IPHeader.Checksum = ~sum;//结果取反
}
```

（三） 程序主体函数设计

1. 获取本机网络接口信息：

调用 in_dev 函数，获取本机所有的网络接口设备，并输出相关信息

```
pcap_if_t* alldevs = nullptr;    //指向设备链表首部的指针
pcap_if_t* d = nullptr;
pcap_addr_t* a = nullptr;        //表示接口地址指针


int num = 0;

// 初始化
num = in_dev(alldevs, d, a);
```

```cpp
int in_dev(pcap_if_t*& alldevs, pcap_if_t*& d, pcap_addr_t*& a)
{
    char errbuf[PCAP_ERRBUF_SIZE];  //错误信息缓冲区

    //获取当前网卡列表
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, //获取本机的接口设备
        NULL, //无需认证
        &alldevs, //指向设备列表首部
        errbuf //出错信息保存缓冲区
    ) == -1)
    {
        cout << "获取本机网卡列表时出错:" << errbuf << endl;
        return 0;
    }

    int num = 0;

    // 打印网卡的列表
    for (d = alldevs; d != NULL; d = d->next)
    {
        num++;
        cout << num << ". " << d->name << "->" << d->description << ";" << endl;

        for (a = d->addresses; a != NULL; a = a->next)
        {
            if (a->addr->sa_family == AF_INET)
            {
                cout << "IP地址：";
                printIP((((sockaddr_in*)a->addr)->sin_addr).s_addr);
                cout << "    ";
                cout << "子网掩码：";
                printIP((((sockaddr_in*)a->netmask)->sin_addr).s_addr);
                cout << "    ";
                cout << "广播地址：";
                printIP((((sockaddr_in*)a->broadaddr)->sin_addr).s_addr);
                cout << endl << endl;
            }
        }
    }

    return num;
}
```

2. 选择一个网络接口设备打开，打印其详细信息，并设置为本机的默认路由

```
//选择网卡
cout << "请选择设备（1-" << num << "):" << endl;
int dev_select_num = 0;
cin >> dev_select_num;

while (dev_select_num < 1 || dev_select_num > num)
{
    cout << "字符非法，请重新输入（1-" << num << "):" << endl;
    cin >> dev_select_num;
    if (dev_select_num >= 1 && dev_select_num <= num)
    {
        break;
    }
}

//转到选择的设备
d = alldevs;
for (int i = 0; i < dev_select_num - 1; i++)
{
    d = d->next;
}

//打印选择的设备的详细信息
cout << "您选择的设备信息为：" << d->name << ";" << endl;
cout << "描述信息：" << d->description << endl;
```

```
//打印选择网卡的IP、子网掩码、广播地址
for (a = d->addresses; a != NULL; a = a->next)
{
    if (a->addr->sa_family == AF_INET)
    {
        cout << "IP地址：";
        printIP((((sockaddr_in*)a->addr)->sin_addr).s_addr);
        local_ip = inet_addr(inet_ntoa(((struct sockaddr_in*)(a->addr))->sin_addr));

        cout << "   ";
        cout << "子网掩码：";
        printIP((((sockaddr_in*)a->netmask)->sin_addr).s_addr);
        cout << "   ";
        cout << "广播地址：";
        printIP((((sockaddr_in*)a->broadaddr)->sin_addr).s_addr);
        cout << endl;

        ULONG NetMask, DesNet, NextHop;
        DesNet = (((sockaddr_in*)a->addr)->sin_addr).s_addr;
        NetMask = (((sockaddr_in*)a->netmask)->sin_addr).s_addr;
        DesNet = DesNet & NetMask;
        NextHop = 0;
        router_table temp;
        temp.netmask = NetMask;
        temp.desnet = DesNet;
        temp.nexthop = NextHop;

        // 本机信息作为默认路由
        additem(rt, rt_length, temp);
    }
}

char errbuf[PCAP_ERRBUF_SIZE];  //错误信息缓冲区

//打开网络接口
pcap_t* handle = pcap_open_live(d->name, BUFSIZ, 1, 1000, errbuf);
if (pcap_open == NULL)
{
    cout << "打开设备" << dev_select_num << "的网络接口失败：" << errbuf << endl;
    return 1;
}
```

3. 调用 set_filter 函数设置“ip or arp”过滤器

```
// 设置过滤器
int set_filter(pcap_t* handle, pcap_if_t* d)
{
    u_int net_mask;
    char packet_filter[] = "ip or arp";
    struct bpf_program fcode;

    net_mask = ((sockaddr_in*)(d->addresses->netmask))->sin_addr.S_un.S_addr;

    if (pcap_compile(handle, &fcode, packet_filter, 1, net_mask) < 0)
    {
        cout << " [警告] 编译过滤器失败！" << endl;
        return 0;
    }

    if (pcap_setfilter(handle, &fcode) < 0)
    {
        cout << " [警告] 设置过滤器时出错！" << endl;
        return 0;
    }

    return 1;
}
```

4. 发送 ARP 请求获取本机 MAC 地址，这里封装了函数 sene_arp_req 函数用于发送 ARP 请求，get_mac 函数用于从 ARP 响应包中提取 MAC 地址

对于获取本机 MAC 地址，使用虚拟的 IP 和 MAC 地址发送 ARP 请求，然后等待 ARP 响应包，并从中提取 MAC 地址

```
// 发送ARP请求获取本机MAC地址
BYTE scrMAC[6] = { 0x66, 0x66, 0x66, 0x66, 0x66, 0x66 };
ULONG virIP = inet_addr("112.112.112.112");
send_arp_req(handle, scrMAC, virIP, local_ip);
get_mac(handle, local_ip, virIP, local_mac);

cout << " 本机IP: ";
printIP(local_ip);
cout << "    本机MAC: ";
for (int i = 0; i < 6; i++)
{
    cout << hex << (int)local_mac[i];
    if (i != 5)cout << "-";
    else cout << endl;
}
```

```cpp
// 发送ARP请求
int send_arp_req(pcap_t* handle, BYTE* srcMAC, ULONG scrIP, ULONG targetIP)
{
    ARPFrame_t ARPFrame;

    for (int i = 0; i < 6; i++)
    {
        ARPFrame.FrameHeader.DesMAC[i] = 0xff; //目的Mac地址设置为广播地址
        ARPFrame.FrameHeader.SrcMAC[i] = srcMAC[i]; //源MAC地址
        ARPFrame.SendHa[i] = srcMAC[i];
        ARPFrame.RecvHa[i] = 0x00; //目的MAC地址设置为0
    }

    ARPFrame.FrameHeader.FrameType = htons(0x0806); // 帧类型为ARP
    ARPFrame.HardwareType = htons(0x0001);          // 硬件类型为以太网
    ARPFrame.ProtocolType = htons(0x0800);          // 协议类型为IP
    ARPFrame.HLen = 6;       // 硬件地址长度为6
    ARPFrame.PLen = 4;       //协议地址长度为4
    ARPFrame.Operation = htons(0x0001);     // 操作为ARP请求
    ARPFrame.SendIP = scrIP;
    ARPFrame.RecvIP = targetIP;

    int result = pcap_sendpacket(handle, (u_char*)&ARPFrame, sizeof(ARPFrame_t));

    return result;
}
```

```cpp
// 从ARP包中解析MAC地址
int get_mac(pcap_t* p, ULONG targetIP, ULONG scrIP, BYTE* mac)
{
    pcap_pkthdr* pkt_header = new pcap_pkthdr[1500];
    const u_char* pkt_data;
    int flag = 0;
    ARPFrame_t* ARPFrame;

    while (!flag)
    {
        int res = pcap_next_ex(p, &pkt_header, &pkt_data);
        if (res == 0)
        {
            continue;
        }

        if (res == 1)
        {
            ARPFrame = (ARPFrame_t*)pkt_data;
            if (ARPFrame->SendIP == targetIP && ARPFrame->RecvIP == scrIP)
            {
                for (int i = 0; i < 6; i++) {
                    mac[i] = ARPFrame->SendHa[i];
                }
                flag = 1;
            }
        }
    }

    return 1;
}
```

5. 至此，做完了所有准备工作，正式进入程序的核心部分

6. 路由表项管理

用户选择是否要修改路由表，如果不做更改，则直接进入捕获数据包部分；如果更改，可以选择增加、删除路由表项，打印当前路由表。

```
LOOP:

    ULONG DesNet, NetMask, NextHop;
    char* desnet = new char[20];
    char* netmask = new char[20];
    char* nexthop = new char[20];

    int op1, fin = 1;
    cout << "是否要修改路由表:" << endl;
    cout << "  1.是\n  2.否" << endl;
    cin >> op1;

    if (op1 == 2)
    {
        fin = 0;
        cout << "当前路由表为：" << endl;
        print_rt(rt, rt_length);
    }
```

```
    while (fin)
    {
        int op2 = 0;
        cout << "请选择你要进行的操作:" << endl;
        cout << "  1.增加路由表项\n  2.删除路由表项\n  3.打印当前路由表" << endl;

        cin >> op2;

        if (op2 == 1)
        {
            cout << " [增加] 请输入目的网络号:";
            cin >> desnet;
            cout << " [增加] 请输入子网掩码:";
            cin >> netmask;
            cout << " [增加] 请输入下一跳步:";
            cin >> nexthop;

            DesNet = inet_addr(desnet);
            NetMask = inet_addr(netmask);
            NextHop = inet_addr(nexthop);

            router_table addRoute;
            addRoute.desnet = DesNet;
            addRoute.netmask = NetMask;
            addRoute.nexthop = NextHop;

            //addRoute(desnet, netmask, nexthop);
            additem(rt, rt_length, addRoute);

        }
```

```
        if (op2 == 2)
        {
            int num = 0;
            cout << " [删除] 请输入要删除的路由项索引：";
            cin >> num;

            if (num == 0 || num == 1)
            {
                cout << " [警告] 不能删除默认路由！" << endl;
            }
            else
                deleteitem(rt, rt_length, num);

        }

        if (op2 == 3)
        {
            print_rt(rt, rt_length);
        }

        else if (op2 < 0 || op2 > 4)
        {
            cout << " [警告] 输入非法！请重新输入" << endl;
        }

        int op3;
        cout << "是否要修改路由表:" << endl;
        cout << "  1.是\n  2.否" << endl;
        cin >> op3;

        if (op3 == 2)
        {
            fin = 0;
            cout << "当前路由表为：" << endl;
            print_rt(rt, rt_length);
            break;
        }

    }
```

7. 数据包捕获并分析

（1） 捕获数据包并存储

```
int ret = pcap_next_ex(handle, &pkt_header, &pkt_data);  // 获取数据包

if (ret)
{
    IPPacket = (Data_t*)pkt_data;

    // 存储
    ULONG Len = pkt_header->len + sizeof(FrameHeader_t);
    u_char* send_packet = new u_char[Len];
    //memcpy(send_packet, pkt_data, Len);
    for (int i = 0; i < Len; i++)
    {
        send_packet[i] = pkt_data[i];
    }
```

（2） 提取数据包的以太网帧和 IP 数据包信息

```
WORD FrameType = IPPacket->FrameHeader.FrameType;
WORD RecvChecksum = IPPacket->IPHeader.Checksum;
```

（3） 对捕获到的数据包进行分析

检查数据包的目的 IP、目的 MAC 与本机是否一致

```
// 检查数据包的目的IP与本机IP是否一致，一致1，不一致0
bool ip_compare = 1;
for (int i = 0; i < 6; i++)
{
    if (local_ip != IPPacket->IPHeader.DstIP)
    {
        ip_compare = 0;
    }
}

// 检查数据包的目的MAC与本机MAC是否一致，一致1，不一致0
bool mac_compare = 1;
for (int i = 0; i < 6; i++)
{
    if (local_mac[i] != IPPacket->FrameHeader.DesMAC[i])
    {
        mac_compare = 0;
    }
}

// 检查是否是IPV4，是1，不是0
bool is_ipv4 = (FrameType == 0x0800);
```

（4） 判断是否需要转发：

本机是一个路由器，需要转发时如果数据包是 IPv4 类型，目标 IP 不是本机 IP，但目标 MAC 是本机 MAC，说明这个数据包时主机 A 发给主机 B 或者为主机 B 发给主机 A 的数据包，这时候需要对数据包进行转发。

判断需要转发后，首先调用 print_ip_packet 函数，打印数据包的相关信息；

调用 search 函数进行选路，结果存入 nextIP，如果 nextIP 为 0xffffffff，证明目标不可达，输出警告提示并中止转发

```
// 如果目的IP不是本机IP，目的MAC地址是本机MAC--转发
if (is_ipv4 && !ip_compare && mac_compare)
{
    print_ip_packet(IPPacket);

    // 选路
    nextIP = search(rt, rt_length, IPPacket->IPHeader.DstIP);

    if (nextIP == 0)
    {
        nextIP = IPPacket->IPHeader.DstIP;
    }
    else if (nextIP == 0xffffffff)
    {
        cout << " [警告] 不可达。无法转发数据包，请重试！ " << endl;
        Count = 8;
    }
```

8. 路由转发

（1） 调用 send_arp_req 函数发送 ARP 请求，获取下一跳 MAC 地址，其中，源 IP 为主机 IP，源 MAC 为主机 MAC，目的 IP 为下一跳 IP，目的 MAC 为广播地址

（2） 调用 get_mac 提取 ARP 响应包的 MAC 地址，并输出

```
// 发送ARP请求，获取下一跳MAC
send_arp_req(handle, local_mac, local_ip, nextIP);
get_mac(handle, nextIP, local_ip, nextMac);

cout << " 下一跳IP：";
printIP(nextIP);
cout << "     下一跳MAC：";
for (int i = 0; i < 6; i++)
{
    cout << hex << (int)nextMac[i];
    if (i != 5)cout << "-";
    else cout << endl;
}
```

（3） 更改数据包，将目标 MAC 改为刚刚获取的下一跳 MAC，将 TTL-1，并重新计算校验和

```
// 更改IP数据包的目的MAC地址
Data_t* temp_packet;
temp_packet = (Data_t*)send_packet;
for (int i = 0; i < 6; i++)
{
    temp_packet->FrameHeader.DesMAC[i] = nextMac[i];
}

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

temp_packet->IPHeader.Checksum = 0;  // 清零校验和
setchecksum(temp_packet);

// 将修改后的TTL同步
memcpy(temp_packet + sizeof(FrameHeader_t) + offsetof(IPHeader_t, TTL), &temp_packet->IPHeader.TTL, sizeof(temp_packet->IPHeader.TTL));

// 同步校验和
memcpy(temp_packet + sizeof(FrameHeader_t) + offsetof(IPHeader_t, Checksum), &temp_packet->IPHeader.Checksum, sizeof(temp_packet->IPHeader.Checksum));
```

（4） 发送修改后的数据包

```
// 发送
if (!pcap_sendpacket(handle, send_packet, Len))
{
    Data_t* t;
    t = (Data_t*)send_packet;
    cout << " [转发] ";
    print_ip_packet(t);

    Count++;

    cout << endl;
}

if (Count == 8)
    break;
```

（5） 在代码中，使用了 Count 用来判断数据包转发次数，因为执行一次 ping 命令会发送 4 个 ICMP 报文到目标主机，同时目标主机会回复 4 个报文，因此完成 8 次转发，说明完成了一次 ping 命令

四、实验结果

1. 在 VMWare 上运行三台虚拟机，全部使用 VMnet8（NAT 模式），并配置 IP 地址如下表所示

| 主机 | IP 地址 | 子网掩码 | 默认路由 |
|---|---|---|---|
| XP 1 | 206.1.1.2 | 255.255.255.0 | 206.1.1.1 |
| XP 3 | 206.1.3.2 | 255.255.255.0 | 206.1.3.1 |

| 路由器 B | IP 地址 | 子网掩码 |
|---|---|---|
| XP 2 | 206.1.2.2 | 255.255.255.0 |
| XP 2 | 206.1.3.1 | 255.255.255.0 |

2. 使用本机物理机作为路由器 A 进行本次实验，修改 VMnet8 的网卡配置如下表所示

| 路由器 A | IP 地址 | 子网掩码 |
|---|---|---|
| 本机 | 206.1.1.1 | 255.255.255.0 |
| 本机 | 206.1.2.1 | 255.255.255.0 |

3. 在路由器 B 所在主机 Windows XP2 中配置路由项

```
C:\Documents and Settings\2>route add 206.1.1.0 mask 255.255.255.0 206.1.2.1
```

4. 在路由器 A 不运行路由转发程序时，从主机 A ping 主机 B，测试网络连通性：运行结果如下图所示，可以看到显示超时，无法 ping 通

```
C:\Documents and Settings\1>ping 206.1.3.2

Pinging 206.1.3.2 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 206.1.3.2:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\Documents and Settings\1>
```

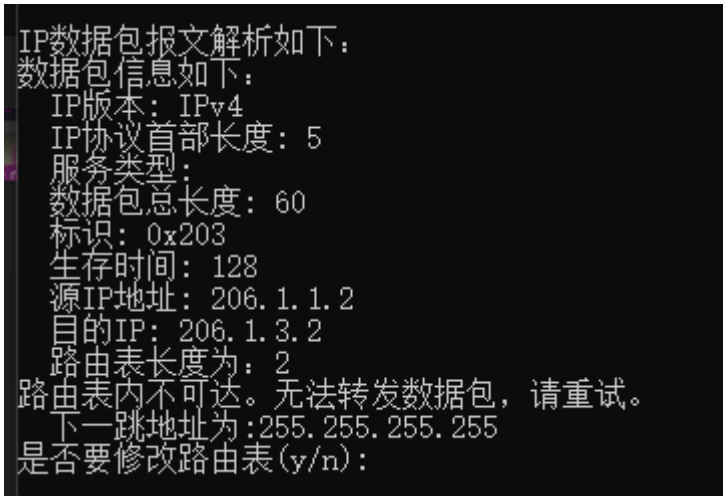
5. 在路由器 A 所在主机本机物理机上运行路由转发程序

（1） 选择 VMnet8 所在的设备 5 打开，运行效果如下图所示，可以看到输出了设备的信息，并且添加了默认路由

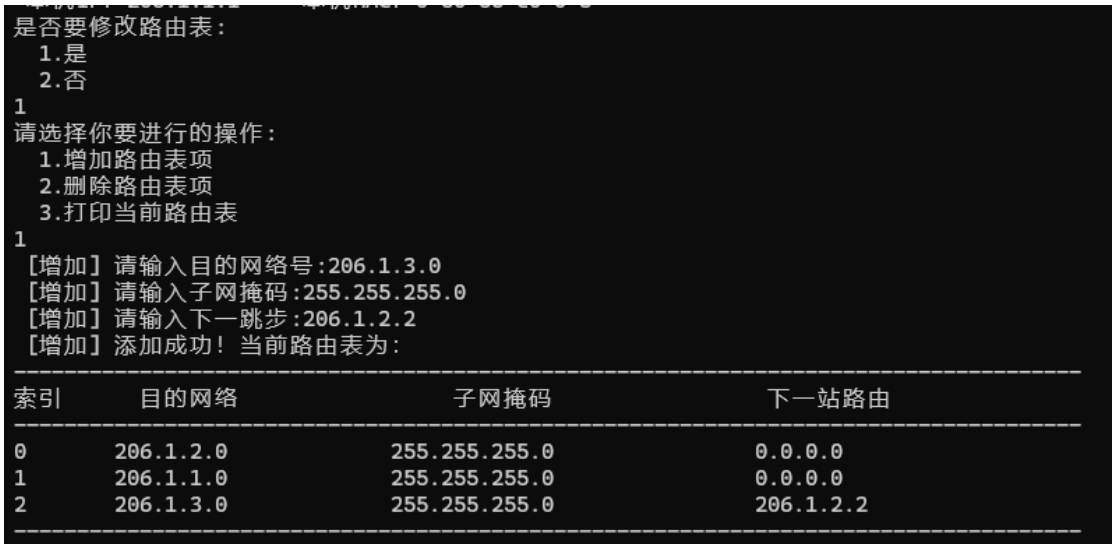
```
D:\Code\C++\Wangji\router\

1. rpcap://\Device\NPF_{BDFF5B12-1655-415A-B84A-BFE3C34E3C25}->Network adapter 'WAN Miniport (Network Monitor)' on local host;
2. rpcap://\Device\NPF_{9775282B-D5FB-487C-B563-0952FE67DBB4}->Network adapter 'WAN Miniport (IPv6)' on local host;
3. rpcap://\Device\NPF_{987AB693-8FC6-4F9A-9E66-89EA6DA760B5}->Network adapter 'WAN Miniport (IP)' on local host;
4. rpcap://\Device\NPF_{93F890BF-9B73-406B-9F66-730414C3049F}->Network adapter 'Intel(R) Wi-Fi 6E AX211 160MHz' on local host;
IP地址: 10.136.39.18    子网掩码: 255.255.128.0    广播地址: 10.136.127.255

5. rpcap://\Device\NPF_{B754A8F1-CBF1-41A7-9F07-91B44E924B41}->Network adapter 'VMware Virtual Ethernet Adapter for VMnet8 #2' on local host;
IP地址: 206.1.2.1    子网掩码: 255.255.255.0    广播地址: 206.1.2.255

IP地址: 206.1.1.1    子网掩码: 255.255.255.0    广播地址: 206.1.1.255

6. rpcap://\Device\NPF_{A9C32118-3F34-486B-8CA9-3EA2E9AD2745}->Network adapter 'VMware Virtual Ethernet Adapter for VMnet1 #2' on local host;
IP地址: 192.168.163.1    子网掩码: 255.255.255.0    广播地址: 192.168.163.255

7. rpcap://\Device\NPF_{634059BE-0654-4F70-BFE6-9156F8412245}->Network adapter 'Microsoft Wi-Fi Direct Virtual Adapter #2' on local host;
IP地址: 169.254.225.156    子网掩码: 255.255.0.0    广播地址: 169.254.255.255

8. rpcap://\Device\NPF_{11D97A15-FAE2-41CC-9CB9-D86F476DCD3D}->Network adapter 'Microsoft Wi-Fi Direct Virtual Adapter' on local host;
IP地址: 169.254.249.252    子网掩码: 255.255.0.0    广播地址: 169.254.255.255

9. rpcap://\Device\NPF_Loopback->Network adapter 'Adapter for loopback traffic capture' on local host;
IP地址: 127.0.0.1    子网掩码: 255.0.0.0    广播地址: 0.0.0.0

5 选择设备 (1-9):
5
您选择的设备信息为: rpcap://\Device\NPF_{B754A8F1-CBF1-41A7-9F07-91B44E924B41};
描述信息: Network adapter 'VMware Virtual Ethernet Adapter for VMnet8 #2' on local host
IP地址: 206.1.2.1    子网掩码: 255.255.255.0    广播地址: 206.1.2.255
 [增加] 添加成功! 当前路由表为:
------------------------------------------------------------------------------
索引      目的网络                子网掩码                下一站路由
------------------------------------------------------------------------------
0       206.1.2.0             255.255.255.0               0.0.0.0
------------------------------------------------------------------------------

IP地址: 206.1.1.1    子网掩码: 255.255.255.0    广播地址: 206.1.1.255
 [增加] 添加成功! 当前路由表为:
------------------------------------------------------------------------------
索引      目的网络                子网掩码                下一站路由
------------------------------------------------------------------------------
0       206.1.2.0             255.255.255.0               0.0.0.0
1       206.1.1.0             255.255.255.0               0.0.0.0
------------------------------------------------------------------------------

 本机IP: 206.1.1.1     本机MAC: 0-50-56-c0-0-8
是否要修改路由表:
  1.是
  2.否
```

（2） 不对路由表项进行操作，直接从主机 A ping 主机 B：

主机 A ping 不通主机 B，同时路由程序的输出如下图所示




（3） 增加路由表项




（4） 删除路由表项：运行效果如下图所示，可以看到无法对默认路由（索引 0 和索引 1）进行删除，可以对非默认路由表项进行删除（索引 2）

```
是否要修改路由表:
  1.是
  2.否
1
请选择你要进行的操作:
  1.增加路由表项
  2.删除路由表项
  3.打印当前路由表
2
[删除] 请输入要删除的路由项索引: 0
[警告] 不能删除默认路由!
是否要修改路由表:
  1.是
  2.否
1
请选择你要进行的操作:
  1.增加路由表项
  2.删除路由表项
  3.打印当前路由表
2
[删除] 请输入要删除的路由项索引: 1
[警告] 不能删除默认路由!
是否要修改路由表:
  1.是
  2.否
1
请选择你要进行的操作:
  1.增加路由表项
  2.删除路由表项
  3.打印当前路由表
2
[删除] 请输入要删除的路由项索引: 2
[删除] 删除成功! 当前路由表为:
------------------------------------------------------------------------------
索引      目的网络                子网掩码                下一站路由
------------------------------------------------------------------------------
0       206.1.2.0             255.255.255.0             0.0.0.0
1       206.1.1.0             255.255.255.0             0.0.0.0
------------------------------------------------------------------------------
```

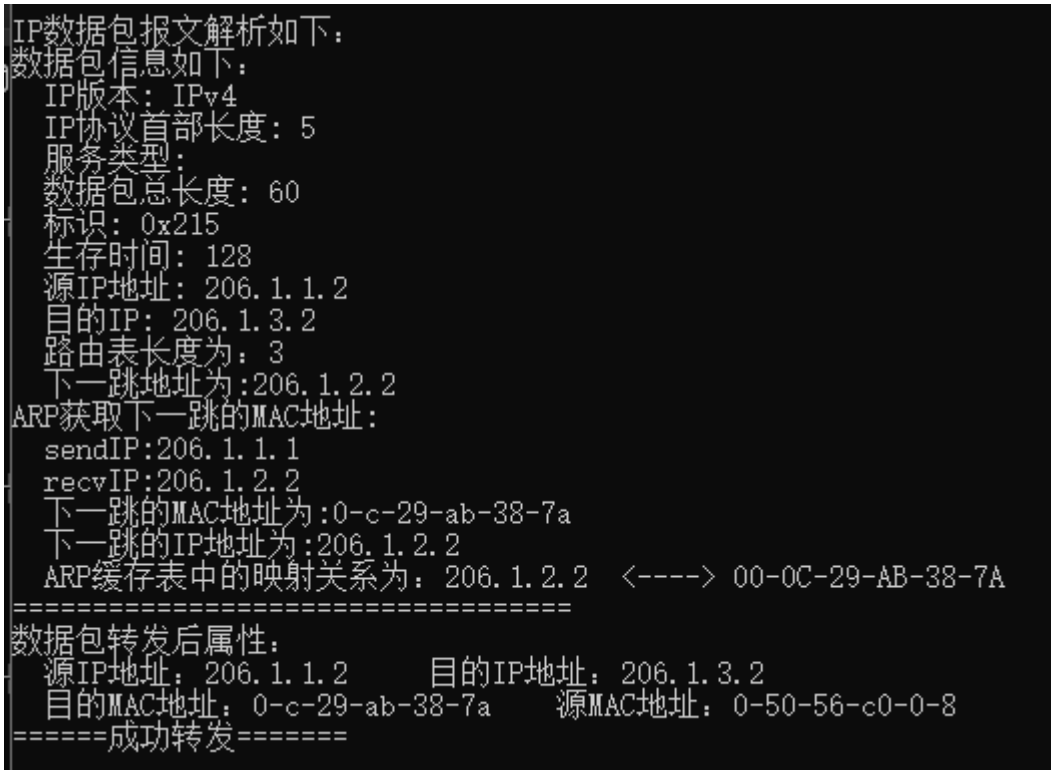
6. 在路由 A 存在转发表项时，从主机 A ping 主机 B，运行结果如下图所示，可以看到 ping 命令执行成功

```
C:\Documents and Settings\1>ping 206.1.3.2

Pinging 206.1.3.2 with 32 bytes of data:

Reply from 206.1.3.2: bytes=32 time=1261ms TTL=126
Reply from 206.1.3.2: bytes=32 time=1430ms TTL=126
Reply from 206.1.3.2: bytes=32 time=1706ms TTL=126
Reply from 206.1.3.2: bytes=32 time=2002ms TTL=126

Ping statistics for 206.1.3.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1261ms, Maximum = 2002ms, Average = 1599ms

C:\Documents and Settings\1>
```

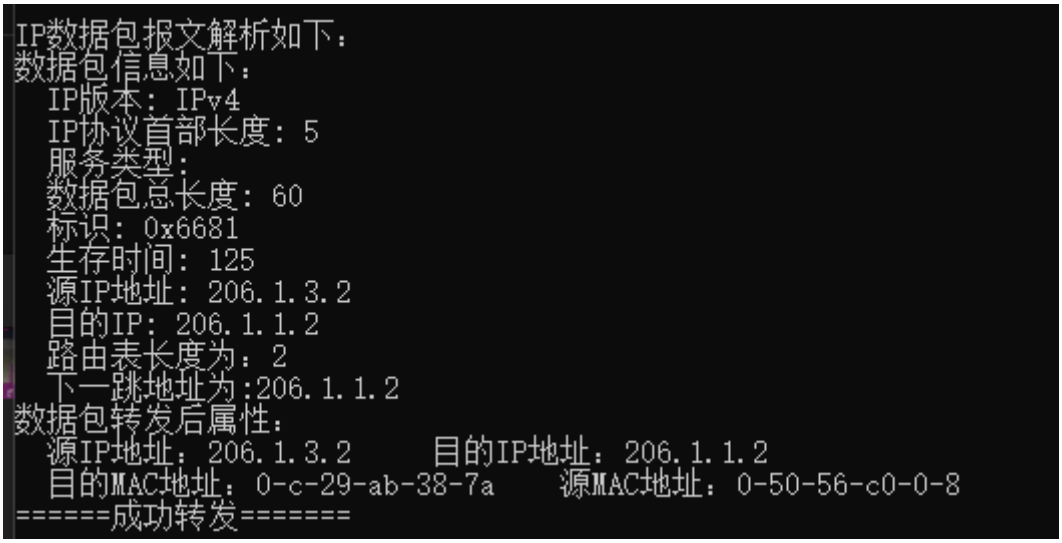
查看路由程序的日志输出：

先是主机 A 到主机 B 的数据包：源 IP 为主机 A 的 IP 地址，目的 IP 为主机 B 的 IP 地址，按照路由规则，下一跳的 IP 地址为 206.1.2.2，发送 ARP 请求获取下一跳的 MAC 地址，将获取到的 MAC 地址设为数据包的目的 MAC 地址然后转发出

去；




再是主机 B 回复主机 A 的数据包：源 IP 为主机 B 的 IP 地址，目的 IP 为主机 A 的 IP 地址，按照路由规则，下一跳的 IP 地址为 206.1.1.2，发送 ARP 请求获取下一跳的 MAC 地址，将获取到的 MAC 地址设为数据包的目的 MAC 地址然后转发出去；



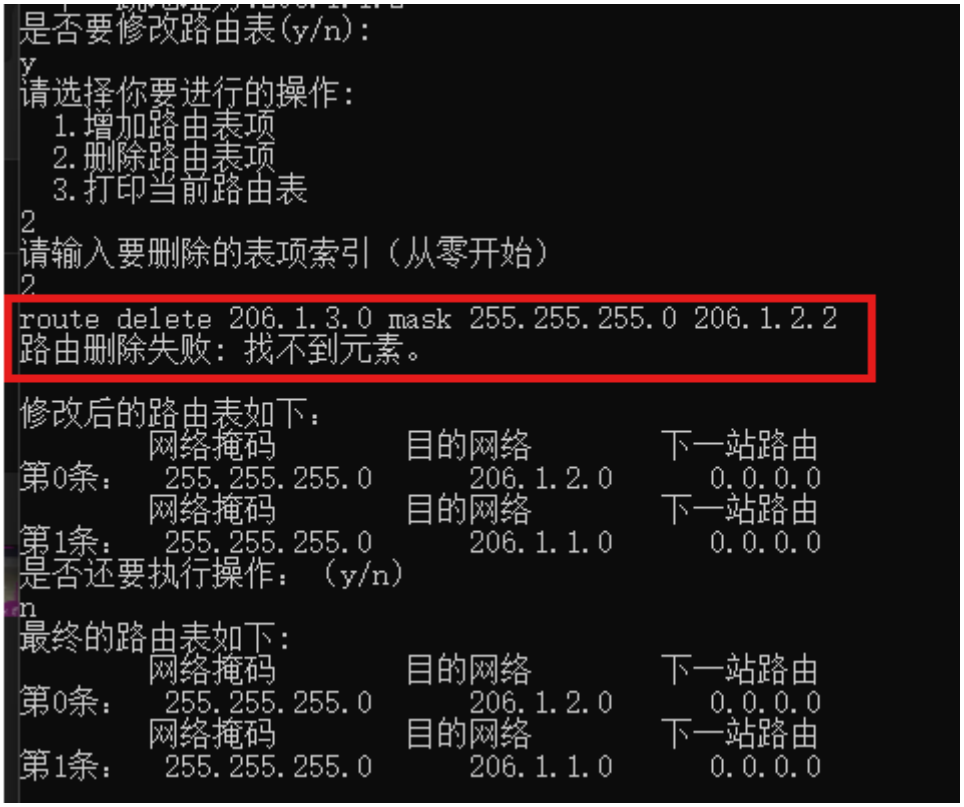

后续的数据包转发日志与上述类似，在此不再赘述

7. 此时使用主机 ping 主机 A 也可以 ping 通，路由程序分析也与之前类似，不再赘述

五、遇到的问题

在执行完路由转发后，对此前添加的路由表项进行删除，虽然打印的路由表已经显示没有索引为 2 的项，但是实际上并没有实际完成删除，在这个时候从主机 A ping 主机 B，依旧可以 ping 通。

这个 bug 与删除路由表项以及捕获处理路由表项部分的代码均有关联




当执行删除操作时，会在路由表中找到该索引对应的表项，将索引项后的每一项向前移动一位，并且将路由表的长度减一，所以打印出来的路由表项为删除之后的正确路由表

实际上，如果函数对数据包捕获分析部分代码无误的话，删除这部分不进行更改也可以。

下面对于数据包捕获分析部分进行分析

由于程序的设计是一条线顺着执行下来的，在每次执行完一轮数据包收发后，会打印是否要进行修改，这时候如果选择不修改，路由程序会在主机没有任何操作的形况下，打印一轮数据包后继续输出是否要进行操作；而即使选择修改，也只是假象上的修改完成，因为程序的跳转逻辑出现问题，修改并未实际同步，程序始终按照第一次修改完成时设置的路由表进行收包、转发。

所以出现这个 bug 的本质原因还是 while 循环和 goto 语句出现的逻辑跳转出现错误。

通过输出调试信息和对程序进行单步调试，成功解决了该 bug。

六、思考与改进

使用路由程序进行转发时，可以看到主机 A 控制台显示回复 time 在几百到几千毫秒之间。我在路由 A 所在的物理主机上添加了路由转发命令，并开启路由转发服务

后，再次执行该命令，主机 A 控制台显示回复 time 只需要几十毫秒，这样巨大差距产生的原因，可能是在每次捕获到数据报文后，都会执行一次 ARP 请求与解析，事实上，只需要在初次进行通讯时进行 ARP 的请求与解析，然后将其存入 ARP 缓存表中，后续再次进行通讯时，只需要查询 ARP 缓存，无需花费大量时间发送、解析 ARP 请求。