实验五：路由器

学号：2112066

姓名：于成俊

专业：密码科学与技术

一、实验要求

简单路由器程序设计实验的具体要求为：

（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。

（5）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。

二、实验工具

（1）Npcap架构

- Npcap 是一种用于数据包捕获和网络分析的架构，用于 Windows 操作系统。它主要有两个重要的DLL文件：wpcap.dll和packet.dll。
  - wpcap.dll：这是libpcap库的Windows版本，Wireshark等网络分析工具会使用这个库来捕获Windows上的实时网络数据。
  - packet.dll：这是一个动态链接库，提供了一组低级函数，用于安装、启动和停止NPF设备驱动程序，从NPF驱动程序接收数据包，向NPF驱动程序发送数据包，以及获取可用网络适配器的列表。

这些文件为Npcap提供了核心功能，使其能够捕获和注入数据包，从而为网络分析和诊断工具提供支持。

所以我们要安装Npcap，并下载Npcap SDK。

（2）Npcap提供的函数

1.`pcap_findalldevs_ex`函数

Npcap 提供了`pcap_findalldevs_ex`函数来获取计算机上的网络接口设备的列表。它的函数原型如下：

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

- `source`：指定是本地适配器还是远程适配器，它告诉函数必须在哪里进行查找，并且使用与`pcap_open()`相同的语法。例如：
  - 本地适配器可以使用`'rpcap://'`
  - 远程适配器可以使用`'rpcap://host:port'`
  - 抓包文件可以使用`'file://c:/myfolder/'`
- `struct pcap_rmtauth *auth`：该参数为身份验证信息，可以为NULL，`struct pcap_rmtauth`的定义如下：

```
struct pcap_rmtauth {
    int type; //简要身份验证所需的类型。
    char *username; //用户名
    char *password; //密码
};
```

- `pcap_if_t **alldevs`：该参数用于存放获取的适配器数据。如果查找失败，`alldevs`的值为NULL
- `char *errbuf`：该参数存放查找失败的信息
- 函数的返回值为0表示查找成功，-1表示查找失败

2.`pcap_open`函数

Npcap提供了`pcap_open`函数来获取数据包捕获句柄以查看网络上的数据包，该函数实际调用的是`pcap_open_live`函数，其函数原型如下：

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

它接收五个参数：

- `const char *source`：网络设备的名字。
- `int snaplen`：定义了 pcap 捕获的最大字节数。
- `int flags`：指定是否将接口置于混杂模式；一般情况下，适配器只接收发给它自己的数据包，而那些在其他机器之间通讯的数据包，将会被丢弃。但混杂模式将会捕获所有的数据包（因为我们需要捕获其他适配器的数据包，所以需要打开这个开关，即设置为1）。
- `int read_timeout`：指定读取数据的超时时间，以毫秒为单位；设置为 0 说明没有超时（如果没有数据包到达，则永远不返回）；对应的还有 -1：读操作后立刻返回。
- `struct pcap_rmtauth *auth`：远程机器的登录信息。本地机器则为 NULL 。
- `char *errbuf`：用于存储错误信息字符串的缓冲区

该函数成功时返回 pcap_t *类型的handle，失败时返回 NULL。如果返回 NULL，则 errbuf 会填充适当的错误消息。

3. `pcap_next_ex` 函数

Npcap提供了`pcap_next_ex`函数用于数据报捕获，其函数原型如下：

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

它接收三个参数：

- `pcap_t* p`：已打开的捕捉实例的描述符。
- `struct pcap_pkthdr** pkt_header`：报文头
- `const u_char** pkt_data`：报文内容

4. `pcap_sendpacket`函数

Npcap提供了`pcap_sendpacket`函数用于发送数据包，其函数原型如下：

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

它接收三个参数：

- `pcap_t *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。
- `const u_char * 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。
- `int size`：要发送的长度

(3) 相关结构体

1. `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` 。

2.、 `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。
- broadaddr： 广播地址。
- dstaddr： P2P目的地址。

3. `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。

定义如下：

```
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、实验知识准备

（1）以太网帧

由于发送的数据包要封装在以太网帧中，我们需要了解相关的知识：

1. **以太网帧格式：**
   - 以太网帧通常包含以下字段：
     - **目的MAC地址（Destination MAC Address）：** 指定接收帧的设备的MAC地址。
     - **源MAC地址（Source MAC Address）：** 指定发送帧的设备的MAC地址。
     - **类型/长度字段（Type/Length）：** 指定以太网帧中数据的类型或表示数据字段的长度。当值大于等于0x0600时，表示数据类型；当值小于等于0x05DC时，表示数据长度。
     - **数据字段（Data）：** 包含传输的数据。
     - **校验和字段（FCS - Frame Check Sequence）：** 用于错误检测，通常是CRC-32校验。
2. **以太网帧的封装：**
   - 以太网帧可以封装各种网络层协议的数据，如IPv4、IPv6、ARP等。
   - 封装过程中，源和目的MAC地址、类型/长度字段会根据封装的网络层协议进行设置。
3. **MAC地址：**
   - MAC地址是网络适配器（网卡）的硬件地址，用于在局域网中唯一标识设备。
   - MAC地址是48位二进制数，通常以十六进制表示。
4. **广播和单播：**
   - 以太网帧可以是广播帧或单播帧。
   - 广播帧的目的MAC地址为全1（FF:FF:FF:FF:FF:FF），用于向网络中所有设备传输数据。
   - 单播帧的目的MAC地址是特定设备的MAC地址，用于直接发送数据到目标设备。

（2）ping原理

根据实验要求，做完我们的路由器，我需要进行ping通检测，所以我们要了解ping的相关知识。

Ping（Packet Internet Groper）是一种用于测试网络连接的常见工具，它基于 ICMP 协议（Internet Control Message Protocol）。以下是 Ping 的基本原理：

- **发起请求：** 使用 Ping 工具时，用户指定目标主机的 IP 地址或域名。Ping 发起一个 ICMP Echo 请求（ping请求）到目标主机。
- **ICMP Echo 请求：** Ping 发送一个 ICMP Echo 请求数据包到目标主机。该数据包包含一个特殊的 ICMP 报文，其中包含发起请求的主机的信息。
- **目标主机响应：** 目标主机收到 ICMP Echo 请求后，会生成一个 ICMP Echo 响应（ping响应）并将其发送回到发起请求的主机。
- **判断连通性：** 发起请求的主机通过收到响应和计算的 RTT 来判断与目标主机的连通性。如果目标主机响应正常，说明网络连接正常；如果没有响应，可能存在网络问题。

为了保证能ping通，我们需要正确转发ICMP报文。

（3）ICMP报文

Internet控制消息协议（ICMP，Internet Control Message Protocol）是用于在IP网络上发送错误消息和操作信息的协议。以下是 ICMP 报文的一些基本信息：

- **ICMP报文格式：** ICMP报文包括报文头部和数据部分。报文头部通常包含类型（Type）、代码（Code）、校验和（Checksum）等字段。
- **报文类型（Type）：** ICMP报文的类型指示了报文的目的和操作。一些常见的类型包括：
  - **Echo Request (8) 和 Echo Reply (0)：** 用于 Ping 测试，发起请求后接收响应。
  - **Destination Unreachable (3)：** 用于指示目标不可达的情况。
  - **Time Exceeded (11)：** 用于指示报文在网络中的传输时间超过了规定的时间。
- **报文代码（Code）：** 与类型字段一起，代码字段提供了更详细的信息，以指示报文的具体目的或错误类型。
- **校验和（Checksum）：** 用于检测 ICMP 报文的传输中是否发生了错误。接收方会检查校验和，如果发现错误，则可能丢弃该报文。
- **Echo Request和Echo Reply：** 用于 Ping 测试。发起方发送 Echo Request 报文，接收方收到后返回 Echo Reply 报文，从而可以测量往返时间（Round-Trip Time，RTT）。
- **Destination Unreachable：** 用于指示目标不可达的情况，可能是因为网络不可达、主机不可达等原因。
- **Time Exceeded：** 用于指示报文在网络中的传输时间超过了规定的时间，可能是由于路由器 TTL 减到零而导致的。

（4）IP数据包

ICMP消息被封装在IP数据包的数据部分中。所以我们还需要了解IP数据包的相关知识。

1. **IP数据包格式：**
   - IP数据包由首部和数据两部分组成。
   - IP首部包含了多个字段，包括版本号、头部长度、服务类型、总长度、标识、标志、片偏移、生存时间（TTL）、协议、首部校验和、源IP地址和目的IP地址。
   - 数据部分包含应用层传输的实际数据。
2. **IP版本：**
   - 目前主要使用的IP版本是IPv4和IPv6。
   - IPv4使用32位地址，而IPv6使用128位地址。

3. **IP地址：**
    - IP地址用于标识网络中的主机和设备。
    - IPv4地址通常表示为四个点分十进制数（例如，192.168.1.1）。
    - IPv6地址通常表示为一组八个十六进制块（例如，2001:0db8:85a3:0000:0000:8a2e:0370:7334）。
4. **生存时间（TTL）：**
    - TTL是一个8位字段，表示数据包在网络中可以经过的最大路由器数。
    - 每经过一个路由器，TTL减1。当TTL减到0时，数据包被丢弃，路由器可能发送ICMP Time Exceeded差错消息。

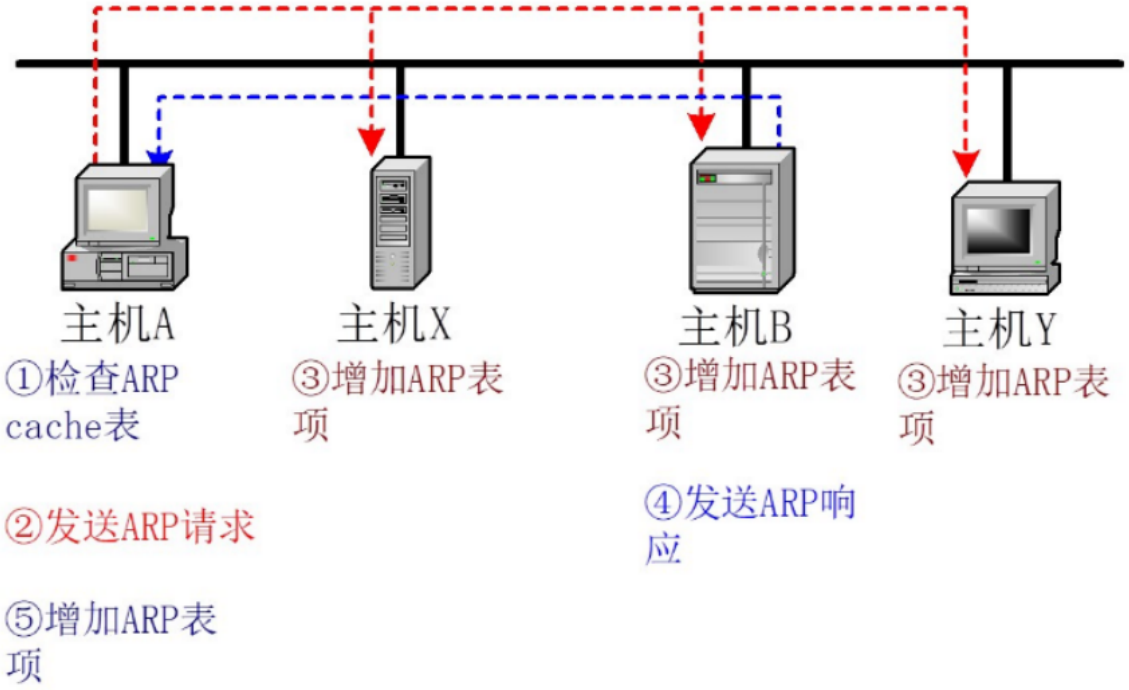
（5）ARP协议

由于我们需要获取本机的MAC地址和其他主机或路由器的MAC地址，所以我们需要用到ARP协议，下面是ARP协议比较重要的相关知识点：

地址解析协议（Address Resolution Protocol，ARP）是一种在IPv4网络中，将网络层IP地址映射为链路层物理地址（MAC地址）的协议。

1. **ARP的作用：**
    - ARP解决了将网络层IP地址映射到链路层物理地址的问题，以便在局域网中正确传递数据帧。
2. **ARP消息：**
    - ARP通过两种主要类型的消息实现地址解析：ARP请求和ARP响应。
    - ARP请求用于查询一个IP地址对应的MAC地址。
    - ARP响应用于回答ARP请求，提供IP地址和对应的MAC地址。
3. **ARP缓存（ARP Cache）：**
    - 主机和路由器维护一个ARP缓存表，其中存储了IP地址和相应的MAC地址的映射关系。
    - ARP缓存用于避免重复的ARP查询，提高网络性能。
4. **ARP协议的工作流程：**
    - 当主机A需要与主机B通信时，如果A不知道B的MAC地址，A会在局域网中广播一个ARP请求。
    - 主机B接收到ARP请求后，会发送一个ARP响应给主机A，包含自己的MAC地址。
    - 主机A收到ARP响应后，会将B的IP地址和MAC地址的映射关系存储在自己的ARP缓存中，以便将来的通信。
5. **ARP与IPv6：**
    - 在IPv6网络中，ARP被邻居发现协议（Neighbor Discovery Protocol，NDP）所取代，提供了类似ARP的功能。

ARP的完整工作过程图示如下：



（6）路由

1.路由选择

选择一条路径发送数据包的过程。

2.路由器

- 进行路由选择的计算机主机上运行合适的软件，可以作为路由器使用。
- 流行的操作系统都提供路由转发功能。

3.IP互联网

- IP互联网是由路由器将多个网络相互联接所组成的。
- 路由器自治：各个路由器独立地对待每个IP数据报。
- 路由器负责为每个IP数据报选择它认为的最佳路径。

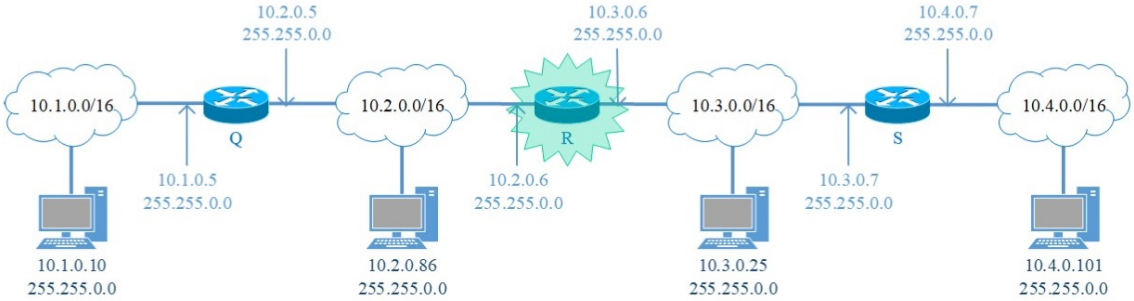
4.间接投递

互联网设备将IP数据报转发给与自己**直接**相连的另一个中间互联网设备的过程

5.直接投递

互联网设备将IP数据报转发给最终目的IP地址所属互联网设备的过程。

6.基本路由选择算法



路由器 R 的路由表

网络前缀 P	网络掩码 M	下一路由器 R
10.2.0.0	255.255.0.0	直接投递
10.3.0.0	255.255.0.0	直接投递
10.1.0.0	255.255.0.0	10.2.0.5
10.4.0.0	255.255.0.0	10.3.0.7

（7）tracert原理

如果还想通过tracert检测，则需要了解tracert相关的知识。

Tracert（Trace Route）是一个用于诊断网络问题的实用程序，它能够显示数据包从源主机传输到目标主机经过的路由器的路径。以下是Tracert的基本原理：

1. **TTL（Time to Live）字段：**
   - IP数据包头部中有一个TTL字段，它表示数据包在网络中可以经过的最大路由器数量。每经过一个路由器，TTL减1。
   - 当TTL减到0时，路由器会丢弃数据包，并向源主机发送ICMP Time Exceeded报文。
2. **Tracert的工作流程：**
   - Tracert向目标主机发送一系列的UDP数据包，每个数据包的目标端口号递增。
   - 初始TTL设为1，第一个数据包到达第一个路由器时，TTL减1，路由器将其丢弃并发送ICMP Time Exceeded报文给源主机。
   - 源主机收到ICMP Time Exceeded报文后，记录当前路由器的IP地址，并增加TTL，然后发送下一个UDP数据包。
   - 这个过程重复，每个数据包经过一个路由器，源主机记录下路由器的IP地址，直到达到目标主机。
3. **ICMP报文的利用：**
   - Tracert依赖于ICMP协议，特别是ICMP Time Exceeded报文，来确定数据包的路径。
   - 如果数据包成功到达目标主机，目标主机会发送ICMP Port Unreachable报文给源主机，表示目标端口不可达。
4. **显示结果：**
   - Tracert将每个路由器的IP地址显示出来，以及从源主机到每个路由器经过的时间（RTT，Round-Trip Time）。
   - 结果中的星号（*）表示在规定时间内未收到ICMP Time Exceeded或ICMP Port Unreachable报文，可能是由于防火墙过滤或路由器不响应ICMP引起的。

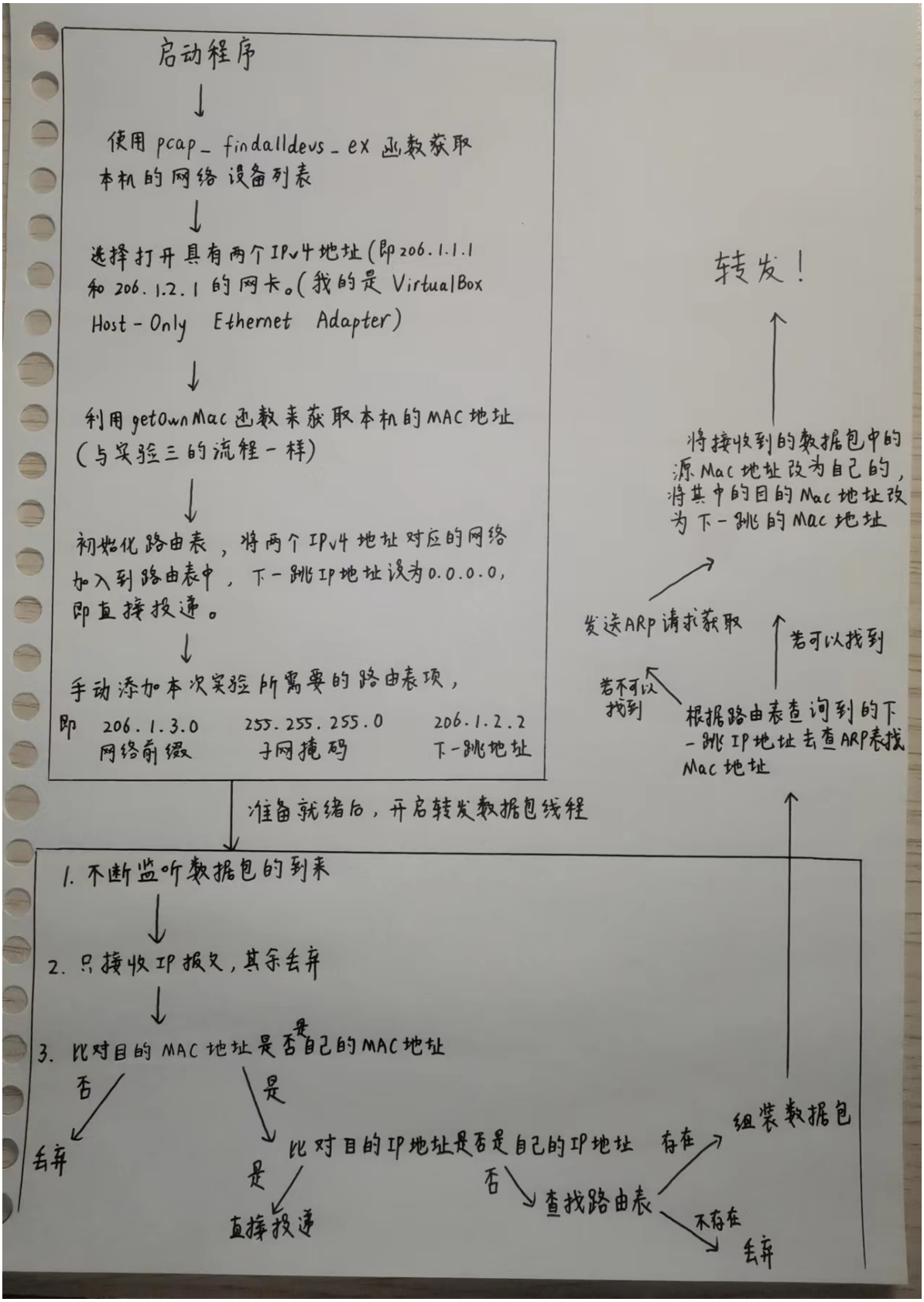
四、代码实现过程

（1）设计思路

掌握了相关的基本概念后，我们要先熟悉路由器转发数据包的流程：

1. **数据包的接收：**
    - 路由器的网络接口监听网络上的数据包。
    - 当一个数据包到达路由器的接口，路由器从链路层开始解析数据包。
2. **链路层处理：**
    - 路由器检查数据包的链路层帧头（Frame Header），获取源和目的 MAC 地址。
    - 如果目的 MAC 地址是路由器自身的 MAC 地址，说明这个数据包是发给路由器的。
3. **查找转发表：**
    - 路由器查找转发表（Routing Table）来确定如何转发数据包。
    - 转发表包含了网络地址（IP 地址）到出口接口的映射。
4. **目的 IP 地址检查：**
    - 路由器检查数据包的目的 IP 地址。
    - 如果目的 IP 地址在路由器的路由表中，路由器确定下一跳的地址。如果不在，则抛弃。
5. **选择下一跳：**
    - 路由器根据路由表中的信息选择下一跳的地址。
    - 如果下一跳是直接相连的网络，那么下一跳就是目的地的地址。
    - 如果下一跳是另一台路由器，路由器将数据包转发给该路由器。
6. **ARP 解析：**
    - 如果下一跳是直接相连的网络，路由器可能需要进行 ARP（Address Resolution Protocol）解析，以获取下一跳的 MAC 地址。
7. **封装数据包：**
    - 路由器创建一个新的链路层帧，更改源和目的 MAC 地址。将源MAC地址改为自身的MAC地址，将目的MAC地址改为下一跳的 MAC 地址。
    - 在新帧中封装原始数据包。
8. **转发数据包：**
    - 路由器将封装后的数据包发送到选择的出口接口，进入下一跳网络。
9. **接口处理：**
    - 接口驱动硬件将数据包发送到下一跳。

可以看出，本次实验是之前实验的结合。根据上面的流程，可以画出一个基本的流程图：



（2）关键结构体（定义在route.h文件中）

网络通信中的数据包都遵循特定的协议，为了更方便地处理网络协议中的数据包格式，即**解析数据包**和**封装数据包：**，我们需要根据数据包结构来自定义结构体。

1.帧首部结构体：

目的地址（6B）	源地址（6B）	长度/类型（2B）

```
// 帧首部结构体
typedef struct FrameHeader_t {
    // 目的地址
    BYTE DesMAC[6];
    // 源地址
    BYTE SrcMAC[6];
    // 帧类型
    WORD FrameType;
}FrameHeader_t;
```

2.ARP报文结构体：

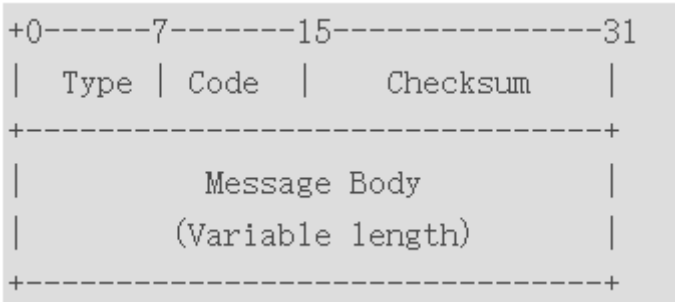

```
// ARP报文结构体
typedef struct ARPFrame_t {
    // 帧首部
    FrameHeader_t FrameHeader;
    // 硬件类型
    WORD HardwareType;
    // 协议类型
    WORD ProtocolType;
    // 硬件地址长度
    BYTE HLen;
    // 协议地址长度
    BYTE PLen;
    // 操作类型
```

```
    WORD Operation;
    // 发送方MAC地址(源MAC地址)
    BYTE SendMac[6];
    // 发送方IP地址(源IP地址)
    DWORD SendIP;
    // 接收方MAC地址(目的MAC地址)
    BYTE RecvMac[6];
    // 接收方IP地址(目的IP地址)
    DWORD RecvIP;
}ARPFrame_t;
```

3.IP报文头部结构体：


```
// IP报文头部结构体
typedef struct IPHeader_t
{
    BYTE Ver_HLen;          // 版本和首部长度（Version and Header Length）
    BYTE TOS;               // 服务类型（Type of Service）
    WORD TotalLen;          // 总长度（Total Length）
    WORD ID;                // 标识（Identification）
    WORD Flag_Segment;      // 标志和片偏移（Flags and Fragment Offset）
    BYTE TTL;               // 存活时间（Time to Live）
    BYTE Protocol;          // 协议类型（Protocol）
    WORD Checksum;          // 校验和（Header Checksum）
    ULONG SrcIP;            // 源IP地址（Source IP Address）
    ULONG DstIP;            // 目的IP地址（Destination IP Address）
} IPHeader_t;
```

4.ICMP头部结构体：




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

5.ICMP报文结构体

```
// ICMP报文结构体
typedef struct ICMP {
    // 帧首部
    FrameHeader_t FrameHeader;
    // IP首部
    IPHeader_t IPHeader;
    //ICMP首部
    ICMPHeader_t ICMPHeader;
    // 缓冲区，用于存储ICMP数据
    char buf[0x80];
}ICMP_t;
```

6.以1字节对齐

注意，我们需要使结构体中的成员变量按照1字节的倍数进行对齐，这意味着每个成员变量都会紧密相邻。我们使用pack（1）来实现，即按如下方式：

```
// 字节对齐方式
#pragma pack(1)
/*--------定义结构体----------*/
// 结束字节对齐方式
# pragma pack()
```

（3）关键类（定义在route.h文件中）

1.路由表项

- 每个路由表项包含了**子网掩码**、**目的网络**、**下一跳的IP地址**、**下一跳的MAC地址**、**序号**和**类型**。（类型有两种，一种是路由表初始化就有的，一种是后来手动添加的）
- 注意，每个路由表项还有一个指针`RouteItem* NextItem`，以便用链表来实现路由表

```
// 路由表项
class RouteItem{
public:
```

```
    // 掩码
    DWORD mask;
    // 目的网络
    DWORD dstnet;
    // 下一跳的IP地址
    DWORD nextIP;
    // 下一跳的MAC地址
    BYTE nextMAC[6];
    // 序号
    int number;
    // 类型0为直接相连,即直接投递的; 1为用户添加（直接相连 不可删除）
    int type;
    RouteItem* NextItem;
    RouteItem() {
        // 将其全部设置为零
        memset(this, 0, sizeof(*this));
    }
    // 打印掩码、目的网络、下一跳IP、类型
    void printitem() {
        // 打印的内容为：子网掩码、目的网络、下一跳IP和类型
        in_addr addr;
        cout << "路由表项" << left << setw(2) << number << ":  ";

        addr.s_addr = mask;
        cout << "子网掩码为:  " << left << setw(17) << inet_ntoa(addr);


        addr.s_addr = dstnet;
        cout << "目的网络为:  " << left << setw(17) << inet_ntoa(addr);

        addr.s_addr = nextIP;
        cout << "下一跳IP地址为:  " << left << setw(17) << inet_ntoa(addr);

        if (type == 0) {
            cout << "类型为:   直接相连" << endl;
        }
        else {
            cout << "类型为:   用户添加" << endl;
        }
    }
};
```

2.路由表

- 路由表有两个指针是`RouteItem* head, * tail`，分别是链表的头和尾，注意，它俩都是边界，即它俩不是具体的路由表项，是没有意义的。
- 路由表有一个初始化函数`void initialize() {}`，因为路由器有两个IP，即一开始，有两个网络与它直接相连，初始化的过程就是将目的网络为直接相连的这两个网络加入到路由表中。
- 路由表有一个添加路由表项的函数`void add(RouteItem* a ) {}`。当添加的时候，它是按照子网掩码大小来添加的，即按照网络前缀长度。这样，在进行匹配的时候，就符合最长匹配原则。并且，每次添加的时候，都会给每个路由表项重新编号。
- 此外，它还有打印函数、删除函数、查找函数。

```
// 路由表
class RouteTable
{
```

```
public:
    //head和tail都是边界，不是具体的路由表项
    RouteItem* head, * tail;
    // 目前存在的个数
    int num;
    // 路由表采用链表形式 并初始化直接跳转的网络
    RouteTable() {
        head = new RouteItem;
        tail = new RouteItem;
        head->NextItem = tail;
        tail->NextItem = NULL;
        num = 0;
    }
    //初始化函数
    void initialize() {
        // 本次实验初始一定只有两个网络
        if (strcmp(OwnIP_1, "") != 0) {
            RouteItem* temp = new RouteItem;
            // 本机网卡的IP和掩码进行按位与的结果为网络号
            temp->dstnet = (inet_addr(OwnIP_1)) & (inet_addr(OwnMASK));
            temp->mask = inet_addr(OwnMASK);
            temp->type = 0;
            // 将其初始化到链表中
            this->add(temp);
        }
        if (strcmp(OwnIP_2, "") != 0) {
            RouteItem* temp = new RouteItem;
            // 本机网卡的IP和掩码进行按位与的结果为网络号
            temp->dstnet = (inet_addr(OwnIP_2)) & (inet_addr(OwnMASK));
            temp->mask = inet_addr(OwnMASK);
            temp->type = 0;
            // 将其初始化到链表中
            this->add(temp);
        }
    }
    // 添加路由表项(不是直接投递的表项在直接投递的表项后面)
    void add(RouteItem* a) {
        RouteItem* temp = new RouteItem(*a);
        //按照掩码由长至短找到合适的位置
        if (num == 0) {
            temp->NextItem = head->NextItem;
            head->NextItem = temp;
        }
        else {
            // 方便找到插入的位置
            RouteItem* pointer;
            for (pointer = head->NextItem; pointer->NextItem != tail; pointer =
pointer->NextItem){

                if (temp->mask < pointer->mask && temp->mask >= pointer-
>NextItem->mask) {
                    break;
                }
            }
            // 插入到合适位置
            temp->NextItem = pointer->NextItem;
            pointer->NextItem = temp;
        }
```

```
        //设置编号
        RouteItem* pointer = head->NextItem;
        for (int i = 0; pointer != tail; pointer = pointer->NextItem, i++)
        {
            pointer->number = i;
        }
        num++;
    }
    //删除路由表项
    void remove(int number) {
        for (RouteItem* t = head; t->NextItem != tail; t = t->NextItem) {

            if (t->NextItem->number == number) {
                // 直接投递的路由表项(type=0)不可删除
                if (t->NextItem->type == 0) {
                    cout << "该项无法删除" << endl;
                    return;
                }
                else {
                    t->NextItem = t->NextItem->NextItem;
                    cout << "删除成功!" << endl;
                    return;
                }
            }
        }
        cout << "查无此项！" << endl;
    }
    //打印路由表:即打印（掩码、网络号、下一跳IP地址）
    void print() {
        RouteItem* pointer = head->NextItem;
        for (; pointer != tail; pointer = pointer->NextItem)
        {
            pointer->printitem();
        }
    }
    //查找 （最长前缀 返回下一跳的`ip`地址）
    DWORD lookup(DWORD ip) {
        RouteItem* t = head->NextItem;
        for (; t != tail; t = t->NextItem)
        {
            if ((t->mask & ip) == t->dstnet) {
                return t->nextIP;
            }
        }
        return -1;
    }
};
```

3.日志类

该日志类会将获取的IP与MAC地址的映射，以及转发和接收过程写入日志文件`Log.txt`

```
//日志类
class RouteLog {
public:
    // 索引
    int index;
```

```
    // 值为“ARP”或“IP”
    char type[5];
    //日志文件
    FILE* text_file;
    //行数
    int line;
    //初始化日志
    RouteLog() {
        // 初始化日志的参数
        line = 0;
        //以追加（"a"）和读取（"+"）的模式
        text_file = fopen("Log.txt", "a+");
    }
    // 将ARP响应得到的IP与MAC地址映射写入日志文件
    void write_ARPLog(ARPFrame_t* pkt) {
        fprintf(text_file, "ARP:");
        in_addr addr;
        addr.s_addr = pkt->SendIP;
        fprintf(text_file, "IP:  ");
        fprintf(text_file, "%s  ", inet_ntoa(addr));

        fprintf(text_file, "MAC:  ");
        for (int i = 0; i < 5; i++) {
            fprintf(text_file, "%02X-", pkt->SendMac[i]);
        }
        fprintf(text_file, "%02X\n", pkt->SendMac[5]);
    }
    //将转发过程和接收过程写入日志（op代表转发或接收）
    void write_route(const char* op, Data_t* pkt) {
        fprintf(text_file, "`IP`");
        fprintf(text_file, op);
        fprintf(text_file, ": ");
        in_addr addr;
        addr.s_addr = pkt->IPHeader.SrcIP;
        char* pchar = inet_ntoa(addr);
        fprintf(text_file, "源IP:  ");
        fprintf(text_file, "%s  ", pchar);
        fprintf(text_file, "目的IP:  ");
        addr.s_addr = pkt->IPHeader.DstIP;
        fprintf(text_file, "%s\n", pchar);
    }
    // 日志打印
    void print() {
        // 读取文件内容并输出到标准输出
        int ch;
        while ((ch = fgetc(text_file)) != EOF) {
            std::cout << static_cast<char>(ch);
        }
    }
    ~RouteLog() {
        fclose(text_file);
    }
};
```

4.ARP缓存表

- 我是用数组来实现的，最多可存储50个ARP表项，用来保存已经获取到的IP与MAC地址的映射。
- 它的插入函数`static void insert(DWORD ip, BYTE mac[6], pcap_t*& handle, RouteLog& WorkLog) {}`，为了使用方便，将获取IP与MAC地址的映射和插入到ARP缓存表结合到了一起，即在内部调用了`getOtherDeviceMAC`函数（稍后会介绍这个函数的实现），来获取IP与MAC地址的映射，之后再将其写入ARP缓存表中。

```
//ARP表项
class ARPItem {
public:
    // IP地址
    DWORD IP;
    // MAC地址
    BYTE MAC[6];
};
// ARP缓存表(存储已经得到的IP与MAC的映射关系)
class ARPtable {
public:
    static ARPItem arpitem[50];
    // 表项数量
    static int num;
    // 插入表项(自带发送ARP请求)
    static void insert(DWORD ip, BYTE mac[6], pcap_t*& handle, RouteLog&
WorkLog) {
        arpitem[num].IP = ip;
        getOtherDeviceMAC(ip, arpitem[num].MAC, handle, WorkLog);
        memcpy(mac, arpitem[num].MAC, 6);
        num++;
    }
    // 查找表项
    static int lookup(DWORD ip, BYTE mac[6]) {
        memset(mac, 0, 6);
        for (int i = 0; i < num; i++){
            if (ip == arpitem[i].IP){
                memcpy(mac, arpitem[i].MAC, 6);
                return 1;
            }
        }
        // 没找到则返回0
        return 0;
    }
};
ARPItem ARPtable::arpitem[50] = {};
// 初始化ARP表项数量
int ARPtable::num = 0;
```

（4）关键函数

1.用于对比两个MAC地址是否一样的函数

```
// 对比两个MAC地址是否相同,相同返回1,不同返回0
int compare(BYTE a[6], BYTE b[6])
{
    int result = 1;
    for (int i = 0; i < 6; i++)
    {
        if (a[i] != b[i])
            result = 0;
    }
    return result;
}
```

2.用于获取IP与MAC地址的映射关系

首先利用ARP协议获取本机mac地址，与ARP实验类似，请求本机网络接口上绑定的IP地址与MAC地址的对应关系：

1. 本地主机模拟一个远端主机，发送一个ARP请求报文，改请求报文请求本机网络接口上绑定的IP地址与MAC地址的对应关系
2. 在组装报文过程中，源MAC地址字段和源IP地址字段需要使用虚假的MAC地址和虚假的IP地址
3. 本次实验使用 `66-66-66-66-66-66` 作为源MAC地址，使用 `112.112.112.112` 作为源IP地址
4. 本地主机一旦获取该ARP请求，就会做出响应

```
// 根据IP获得本机的MAC地址
void getOwnMac(DWORD IP, pcap_t*& handle) {
    // 初始化 OwnMac 数组为 0
    memset(OwnMac, 0, sizeof(OwnMac));
    // 创建 ARP 帧结构体
    ARPFrame_t ARPFrame;
    // 设置 ARP 帧的目的地址为广播地址
    for (int i = 0; i < 6; i++) {
        ARPFrame.FrameHeader.DesMAC[i] = 0xff;
    }
    // 设置 ARP 帧的源 MAC 地址为虚拟 MAC 地址
    for (int i = 0; i < 6; i++) {
        ARPFrame.FrameHeader.SrcMAC[i] = 0x66;
    }
    // 设置 ARP 帧的帧类型为 ARP
    ARPFrame.FrameHeader.FrameType = htons(0x0806);
    // 设置 ARP 帧的硬件类型为以太网
    ARPFrame.HardwareType = htons(0x0001);
    // 设置 ARP 帧的协议类型为 IP
    ARPFrame.ProtocolType = htons(0x0800);
    // 设置 ARP 帧的硬件地址长度为 6
    ARPFrame.HLen = 6;
    // 设置 ARP 帧的协议地址长度为 4
    ARPFrame.PLen = 4;
    // 设置 ARP 帧的操作为 ARP 请求
    ARPFrame.Operation = htons(0x0001);
    // 设置 ARP 帧的发送方硬件地址为虚拟 MAC 地址
    for (int i = 0; i < 6; i++) {
        ARPFrame.SendMac[i] = 0x66;
    }
    // 设置 ARP 帧的发送方 IP 地址为虚拟 IP 地址
    ARPFrame.SendIP = inet_addr("112.112.112.112");
    // 设置 ARP 帧的接收方硬件地址为未知的 MAC 地址
```

```
    for (int i = 0; i < 6; i++) {
        ARPFrame.RecvMac[i] = 0x00;
    }
    // 设置 ARP 帧的接收方 IP 地址为传入的 IP 地址
    ARPFrame.RecvIP = IP;

    // 检查句柄是否为NULL
    if (handle == NULL) {
        cout << "网卡接口打开错误" << endl;
    }
    else {
        // 发送 ARP 帧
        if (pcap_sendpacket(handle, (u_char*)&ARPFrame, sizeof(ARPFrame_t)) ==
0) {
            ARPFrame_t* IPPacket;

            // 循环接收并处理 ARP 响应帧
            while (true) {
                pcap_pkthdr* pkt_header;
                const u_char* pkt_data;

                // 获取下一个数据包
                int rtn = pcap_next_ex(handle, &pkt_header, &pkt_data);

                // 如果成功获取数据包
                if (rtn == 1) {
                    // 将数据包的内容解析为 ARP 帧
                    IPPacket = (ARPFrame_t*)pkt_data;

                    // 检查帧类型是否为 ARP
                    if (ntohs(IPPacket->FrameHeader.FrameType) == 0x0806) {
                        // 检查是否为 ARP 响应
                        if (!compare(IPPacket->FrameHeader.SrcMAC,
ARPFrame.FrameHeader.SrcMAC) && compare(IPPacket->FrameHeader.DesMAC,
ARPFrame.FrameHeader.SrcMAC)) {
                            // 把获得的关系写入到日志表中
                            WorkLog.write_ARPLog(IPPacket);

                            // 将源 MAC 地址复制到 OwnMac 中
                            for (int i = 0; i < 6; i++) {
                                OwnMac[i] = IPPacket->FrameHeader.SrcMAC[i];
                            }
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

还有一个`void getOtherDeviceMAC(DWORD ip, BYTE mac[], pcap_t*& handle, RouteLog& WorkLog) {}`函数用来获取其他主机或路由器（不是本机的MAC地址），流程与`void getOwnMac(DWORD IP, pcap_t*& handle) {}`函数一样。**唯一的区别**在于要将源MAC地址设为本机网卡的真实MAC地址：

```
// 将APRFrame.FrameHeader.SrcMAC设置为本机网卡的MAC地址
for (int i = 0; i < 6; i++){
    ARPFrame.FrameHeader.SrcMAC[i] = OwnMac[i];
    ARPFrame.SendMac[i] = OwnMac[i];
}
```

3.计算校验和

这个函数用来计算IP头部的校验和，具体步骤如下：

- 将IP头部的所有16位字都视为16位无符号整数，进行累加。
- 如果累加的结果溢出，将溢出的部分加回到最低位。
- 取累加结果的反码，即将所有位取反。
- 将得到的反码作为IP头部的校验和。

```
// 计算 IP 头部校验和
void setchecksum_IP(IPHeader_t* temp)
{
    // 将原始校验和字段置为 0
    temp->Checksum = 0;

    // 初始化变量，用于存储校验和的中间结果
    unsigned int sum = 0;

    // 定义指针，指向数据结构 temp 中的 IP 首部
    WORD* t = (WORD*)temp;

    // 遍历 IP 首部的每两个字节
    for (int i = 0; i < sizeof(IPHeader_t) / 2; i++)
    {
        // 将每两个字节的值相加到 sum 中
        sum += t[i];

        // 包含原有的校验和相加
        // 如果 sum 超过 16 位，则将溢出的部分加回到 sum 中
        while (sum >= 0x10000)
        {
            int s = sum >> 16;
            sum -= 0x10000;
            sum += s;
        }
    }

    // 对 sum 取反，然后赋值给 IP 头部的校验和字段
    temp->Checksum = ~sum;
}
```

由于ICMP报文头部的校验和还需要覆盖数据部分，而本次实验只要求正确转发数据包即可，所以我并没有写计算ICMP报文头部校验和的函数。

4.检验校验和：

流程与计算校验和类似。

```
// 检查 IP 头部校验和
bool checkchecksum_IP(IPHeader_t* temp)
```

```
{
    // 初始化变量，用于存储校验和的中间结果
    unsigned int sum = 0;

    // 定义指针，指向数据结构 temp 中的 IP 首部
    WORD* t = (WORD*)temp;

    // 遍历 IP 首部的每两个字节
    for (int i = 0; i < sizeof(IPHeader_t) / 2; i++)
    {
        // 将每两个字节的值相加到 sum 中
        sum += t[i];

        // 包含原有的校验和相加
        // 如果 sum 超过 16 位，则将溢出的部分加回到 sum 中
        while (sum >= 0x10000)
        {
            int s = sum >> 16;
            sum -= 0x10000;
            sum += s;
        }
    }

    // 检查校验和是否等于 65535
    if (sum == 65535)
    {
        // 校验和全1代表正确，返回 true
        return true;
    }

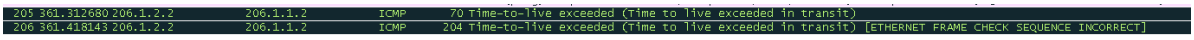
    // 校验和不等于 65535，代表错误，返回 false
    return false;
}
```

（5）转发线程

1.数据报转发函数

这里有一个坑，也就是转发数据报的时候，不能转发固定的长度。我在用Wireshark抓包发现，当length不对时，会出现`ETHERNET FRAME CHECK SEQUENCE INCORRECT`的错误。

如下图，长度为70的ICMP超时报文是正确的，而长度为204的ICMP超时报文是错误的。



所以，在用`pcap_next_ex`函数获取数据包时，需要根据结构体pkt_header的成员caplen的值，获取实际捕获的数据包的长度，从而进行转发。如下：

```
int rtn = pcap_next_ex(handle, &pkt_header, &pkt_data);
int packetLength = pkt_header->caplen;
```

然后，将`packetLength`作为参数传给`resend`函数，就不会再报错。

下面是resend函数的代码，核心就是将源MAC改为本机MAC，并将目的MAC改为下一跳MAC。

```
// 数据报转发(修改源MAC和目的MAC)
void resend(ICMP_t data, BYTE DstMAC[], pcap_t*& handle,int packetLength) {
```

```
    ICMP_t* temp = &data;
    // 将源MAC改为本机MAC
    memcpy(temp->FrameHeader.SrcMAC, temp->FrameHeader.DesMAC, 6);
    // 将目的MAC改为下一跳MAC
    memcpy(temp->FrameHeader.DesMAC, DstMAC, 6);
    // 发送数据报
    int rtn = pcap_sendpacket(handle, (const u_char*)temp, packetLength);
    //int rtn = pcap_sendpacket(handle, (const u_char*)temp, 74);
    if (rtn == 0) {
        // 将其写入日志
        WorkLog.write_route("转发", temp);
    }
}
```

2.发送ICMP超时报文函数

send函数的逻辑与resend函数是一样的，唯一的区别在于send函数只发送ICMP超时报文，从上面我们知道，长度为70的ICMP超时报文才是正确的，才会被主机识别，所以send函数是**以固定长度70来发送ICMP超时报文的。**

```
// 发送ICMP超时报文(修改源MAC和目的MAC)
void send(ICMP_t data, BYTE DstMAC[], pcap_t*& handle) {
    ICMP_t* temp = &data;
    // 将源MAC改为本机MAC
    memcpy(temp->FrameHeader.SrcMAC, OwnMac, 6);
    // 将目的MAC改为下一跳MAC
    memcpy(temp->FrameHeader.DesMAC, DstMAC, 6);
    // 发送数据报
    int rtn = pcap_sendpacket(handle, (const u_char*)temp, 70);
    if (rtn == 0) {
        // 将其写入日志
        WorkLog.write_route("发送", temp);
    }
}
```

3.转发线程函数

收到数据包后，处理流程如下：

- 1.判断数据包的目的MAC是否为自己本机的MAC地址，如果不是则抛弃。
- 2.判断是不是IP数据包，如果不是则抛弃。
- 3.查找路由表，如果没有找到路径则抛弃。
- 4.检查校验和，若不正确则抛弃。
- 5.检查是否是发给本机的数据包，如果不是则转发。
- 6.转发之前要将TTL减1，并重新计算校验和。（为了实现tracert）
- 7.如果TTL减1后为0，则说明，需要发送ICMP超时报文，所以不转发，构造一个ICMP超时报文，发送给源主机。
- 8.如果TTL减1后不为0，则需要转发。转发的时候，要修改帧首部的MAC地址。将源MAC地址改为本机MAC地址；将目的MAC地址改为下一跳MAC地址。

该函数包括非常重要的部分，即构造ICMP超时报文，由于较长，设计思路我会在**实验深入部分**来详细说明。详细过程如下：

```
// 转发线程函数
void Routeforward(RouteTable routetable, pcap_t*& handle) {
    while (true) {
```

```
        // 定义一个指向pcap_pkthdr结构的指针，用于存储数据包的头部信息
        pcap_pkthdr* pkt_header;
        // 定义一个指向u_char类型的指针，用于存储数据包的数据部分
        const u_char* pkt_data;
        // 无限循环，直到接收到数据包
        while (true) {
            // 使用pcap_next_ex函数从handle指定的数据源获取下一个数据包
            // 如果成功接收到数据包，pcap_next_ex函数会返回1，并且pkt_header和pkt_data会
被设置为指向数据包的头部信息和数据部分
            int rtn = pcap_next_ex(handle, &pkt_header, &pkt_data);
            // 如果接收到数据包，就跳出循环
            if (rtn) {
                break;
            }
        }
        int packetLength = pkt_header->caplen;
        // 将接收到的数据包的数据部分转换为ICMP_t类型的数据
        ICMP_t* data = (ICMP_t*)pkt_data;
        //比较数据包的目标MAC地址（data->FrameHeader.DesMAC）和本机的MAC地址（OwnMac）
        if (compare(data->FrameHeader.DesMAC, OwnMac)) {
            // 检查数据包的帧类型是否为0x0800（即IP协议）
            // ntohs函数用于将网络字节序转化为主机字节序
            if (ntohs(data->FrameHeader.FrameType) == 0x0800) {
                //检查IP数据报的上层协议是否为ICMP（Protocol字段是一个字节的，不存在字节序
的问题）
                if (data->IPHeader.Protocol == 1) {
                    // 如果是，将接收到的数据包写入日志
                    WorkLog.write_route("接收", data);
                    // 获取数据包的源IP地址和目标IP地址
                    DWORD SourceIP = data->IPHeader.SrcIP;
                    DWORD DestIP = data->IPHeader.DstIP;
                    // 在路由表中查找目标IP地址对应的下一跳IP地址
                    DWORD NextIP = routetable.lookup(DestIP);
                    // 如果在路由表中没有找到对应的表项，就丢弃这个数据包，并输出一条消息
                    if (NextIP == -1) {
                        cout << "未找到转发路径，已丢弃该数据包！" << endl;
                        continue;
                    }
                    // 首先，检查IP报头的校验和是否正确
                    // 如果校验和不正确，则直接丢弃数据包，不进行后续处理
                    if (checkchecksum_IP(&data->IPHeader)) {
                        // 检查数据包的目标IP地址是否为本机的IP地址
                        // 如果不是本机的IP地址，那么这个数据包需要转发
                        if (data->IPHeader.DstIP != inet_addr(OwnIP_1) && data-
>IPHeader.DstIP != inet_addr(OwnIP_2)) {
                            // 检查数据包是否为广播消息
                            // 如果不是广播消息，那么需要对数据包进行转发
                            int t1 = compare(data->FrameHeader.DesMAC,
broadcast);
                            int t2 = compare(data->FrameHeader.SrcMAC,
broadcast);
                            if (!t1 && !t2) {
                                // 如果TTL字段减1后的值为0，那么需要发送一个ICMP超时消息
                                if (data->IPHeader.TTL-1 == 0) {
                                    //构造ICMP超时报文
                                    ICMP_t* icmp_packet = new ICMP_t(*data);
                                    //清空
                                    memset(icmp_packet, 0, sizeof(ICMP_t));
```

```
                                    //设置帧头部
                                    icmp_packet->FrameHeader.FrameType =
htons(0x0800);
                                    //设置IP头部
                                    icmp_packet->IPHeader.Ver_HLen = 0b01000101;
                                    icmp_packet->IPHeader.TOS = 0;
                                    icmp_packet->IPHeader.TotalLen = htons(56);
                                    icmp_packet->IPHeader.Flag_Segment =
htons(0);
                                    icmp_packet->IPHeader.ID = 0;
                                    icmp_packet->IPHeader.TTL = 128;     //生存时间
                                    icmp_packet->IPHeader.Protocol = 1; // ICMP的
协议号
                                    icmp_packet->IPHeader.SrcIP =
inet_addr(OwnIP_2);
                                    icmp_packet->IPHeader.DstIP = data-
>IPHeader.SrcIP;
                                    //计算校验和
                                    setchecksum_IP(&icmp_packet->IPHeader);
                                    // 设置ICMP类型和代码
                                    icmp_packet->ICMPHeader.Type = 11; // ICMP超
时类型
                                    icmp_packet->ICMPHeader.Code = 0; // 超时的代
码
                                    icmp_packet->ICMPHeader.Checksum =
htons(0xf4ff);
                                    //将接收到的IP数据包的IP头部和数据部分的前64bit(就
是ICMP头部)放到数据部分
                                    memcpy(icmp_packet->buf, &data->IPHeader,
sizeof(data->IPHeader));
                                    memcpy(icmp_packet->buf + sizeof(data-
>IPHeader), &data->ICMPHeader, sizeof(data->ICMPHeader));
                                    //获取源IP和目的IP
                                    SourceIP = icmp_packet->IPHeader.SrcIP;
                                    DestIP = icmp_packet->IPHeader.DstIP;
                                    //查找路由表，找相关的路径
                                    NextIP = routetable.lookup(DestIP);
                                    //如果找到了
                                    if (NextIP != -1) {
                                        //用于存储获取到的MAC地址
                                        BYTE mac[6];
                                        if (NextIP == 0) {
                                            //发送ARP请求，寻找下一跳的MAC地址
                                            if (!ARPtable::lookup(DestIP, mac))
{
                                                //insert()函数中会发送ARP请求
                                                ARPtable::insert(DestIP, mac,
handle, WorkLog);
                                            }
                                            // 发送超时ICMP报文
                                            send(*icmp_packet, mac, handle);

                                        }
                                        else {
                                            //发送ARP请求，寻找下一跳的MAC地址
                                            if (!ARPtable::lookup(NextIP, mac))
{
```

```
                                        ARPtable::insert(NextIP, mac,
handle, WorkLog);
                                    }
                                    //计算IP头部校验和
                                    setchecksum_IP(&icmp_packet-
>IPHeader);
                                    // 发送超时ICMP报文
                                    send(*icmp_packet, mac, handle);
                                }
                                // 打印的内容为：`源IP 目的IP 下一跳IP`
                                in_addr addr;
                                cout << "------------------------------
------------------------------------------------------" << endl;
                                cout << "发送超时ICMP报文ing" << endl;
                                cout << "源IP:  ";
                                addr.s_addr = SourceIP;
                                char* pchar = inet_ntoa(addr);
                                printf("%s\t", pchar);
                                cout << endl;

                                cout << "目的IP:  ";
                                addr.s_addr = DestIP;
                                pchar = inet_ntoa(addr);
                                printf("%s\t", pchar);
                                cout << endl;

                                cout << "下一跳IP:  ";
                                addr.s_addr = NextIP;
                                pchar = inet_ntoa(addr);
                                printf("%s\t\t", pchar);
                                cout << endl;

                                cout << "------------------------------
------------------------------------------------------" << endl;
                            }
                        }
                        else {
                            // 将数据包的TTL字段减1
                            data->IPHeader.TTL -= 1;
                            //重新计算IP头部校验和
                            setchecksum_IP(&data->IPHeader);
                            //用于存储获取到的MAC地址
                            BYTE mac[6];
                            // 如果下一跳IP地址为0，那么这是一个直接投递的数据包
                            if (NextIP == 0) {
                                //获取MAC地址
                                if (!ARPtable::lookup(DestIP, mac)) {
                                    //insert()函数中会发送ARP请求
                                    ARPtable::insert(DestIP, mac,
handle, WorkLog);
                                }
                                // 转发数据包
                                resend(*data, mac, handle,
packetLength);
                            }
                            // 如果下一跳IP地址不为-1和0，那么这是一个非直接投递
的数据包
                            else if (NextIP != -1) {
```

```
                                    if (!ARPtable::lookup(NextIP, mac)) {
                                        ARPtable::insert(NextIP, mac,
handle, WorkLog);
                                    }
                                    // 转发数据包
                                    resend(*data, mac, handle,
packetLength);
                                }
                                // 打印的内容为：`源IP 目的IP 下一跳IP`
                                in_addr addr;
                                cout << "----------------------------------
--------------------------------------------------" << endl;
                                cout << "转发数据报ing" << endl;
                                cout << "源IP：  ";
                                addr.s_addr = SourceIP;
                                char* pchar = inet_ntoa(addr);
                                printf("%s\t", pchar);
                                cout << endl;

                                cout << "目的IP：  ";
                                addr.s_addr = DestIP;
                                pchar = inet_ntoa(addr);
                                printf("%s\t", pchar);
                                cout << endl;

                                cout << "下一跳IP：  ";
                                addr.s_addr = NextIP;
                                pchar = inet_ntoa(addr);
                                printf("%s\t\t", pchar);
                                cout << endl;

                                cout << "----------------------------------
--------------------------------------------------" << endl;

                            }
                        }
                    }

                }
                else {
                    cout << "校验和不正确，已丢弃该数据包" << endl;
                }

            }
        }
    }
}
```

(6) 主函数

在主函数中，首先获取设备列表，然后用户选择打开有两个IP地址的网卡，并利用ARP协议获取本机的MAC地址，接着根据这两个IP地址和子网掩码来初始化路由表，接着开启接收数据报的线程，满足要求的数据报就进行转发，最后在主程序中运行一个死循环，可以实现添加、删除、显示路由表项以及退出的操作。

```
int main(){
    //将const char* PCAP_SRC_IF_STRING 变为 char* pcap_src_if_string（接口字符串）
    // 解决使用函数pcap_findalldevs_ex()报错的问题
    char* pcap_src_if_string = new char[strlen(PCAP_SRC_IF_STRING)];
    strcpy(pcap_src_if_string, PCAP_SRC_IF_STRING);
    //打开的网络接口
    pcap_t* handle = NULL;
    // 获取网络设备
    find_alldevs(pcap_src_if_string, handle);
    //获取本机MAC地址
    cout << "本机MAC地址为：  ";
    getOwnMac(inet_addr(OwnIP_1), handle);
    printMAC(OwnMac);
    //路由表
    RouteTable routetable;
    //初始化路由表
    routetable.initialize();
    //启动接收线程
    thread RouteforwardThread(Routeforward, routetable, ref(handle));
    RouteforwardThread.detach();
    int operation;
    while (true)
    {
        // 进行简介
        cout <<
"================================================================================
=====" << endl;
        cout << "欢迎来到高级路由器，请选择你想要进行的操作：" << endl;
        cout << "1. 添加路由表项" << endl;
        cout << "2. 删除路由表项" << endl;
        cout << "3. 打印路由表：" << endl;
        cout << "4. 退出程序" << endl;
        cout <<
"================================================================================
=====" << endl;
        // 输入想要进行的操作
        cin >> operation;
        if (operation == 1)
        {
            RouteItem routeitem;
            char cin_mask[30];
            char cin_dstip[30];
            char cin_nextip[30];
            cout << "请输入网络掩码：" << endl;
            cin >> cin_mask;
            routeitem.mask = inet_addr(cin_mask);

            cout << "请输入目的网络`ip`地址：" << endl;
            cin >> cin_dstip;
            routeitem.dstnet = inet_addr(cin_dstip);

            cout << "请输入下一跳`ip`地址：" << endl;
            cin >> cin_nextip;
            routeitem.nextIP = inet_addr(cin_nextip);

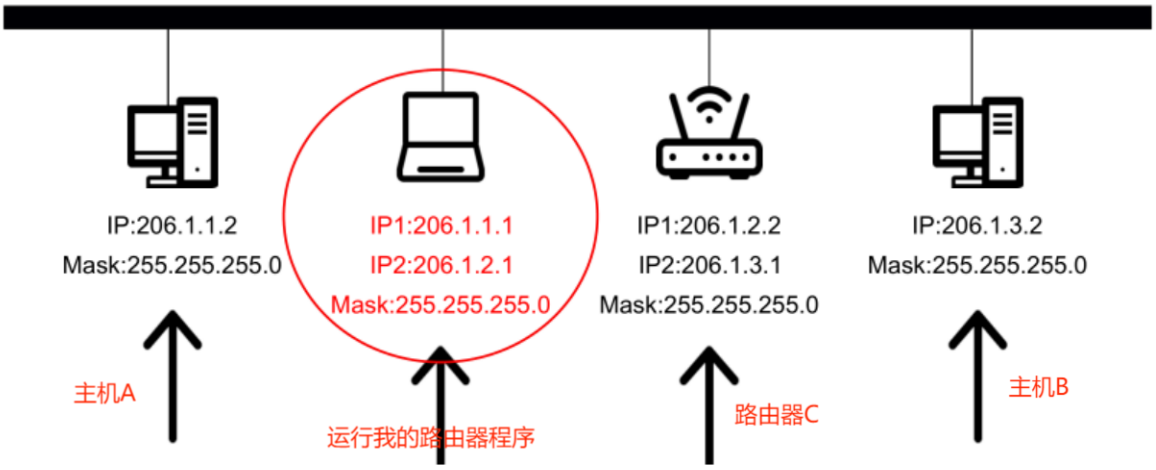
            // 手动添加的类型
            routeitem.type = 1;
```

```
            routetable.add(&routeitem);
        }
        else if (operation == 2)
        {
            cout << "请输入你想要删除的表项编号：" << endl;
            int number;
            cin >> number;
            routetable.remove(number);
        }
        else if (operation == 3)
        {
            routetable.print();
        }
        else if (operation == 4)
        {
            break;
        }
        else {
            cout << "请输入正确的操作号！" << endl;
        }
    }
    return 0;
}
```

五、实验验证过程

（1）网络拓扑结构

由于在虚拟机2上无法运行我写的路由器程序，所以我选择在主机上运行路由器程序，然后有两个虚拟机作为主机，有一个虚拟机作为路由器。网络拓扑如下：



（2）将网卡配置成双IP地址

我的是联想电脑，Windows11

- 右键点击**此电脑**—>点击**属性**—>点击网络和Internet—>点击高级网络设备—>点击以太网3—>编辑更多适配器选项—>点击Internet 协议版本4—>点击使用下面的IP地址—>点击高级
- 添加两个IPv4地址
    - 206.1.1.1　255.255.255.0
    - 206.1.2.2　255.255.255.0

如下图：

- 配置完成后，注意要一路点击确定返回，否则保存不上。

(3) 虚拟机配置

- 关闭防火墙：打开控制面板，进入“所有控制面板”选项—>选择Windows防火墙—>关闭防火墙（正常情况下，虚拟机都是关闭的）
- 虚拟机默认的IP地址都是实验要求的那样，所以不需要进行配置。
- 在充当路由器的虚拟机中，打开命令行输入`route add 206.1.1.0 mask 255.255.255.0 206.1.2.1`，来手动添加路由表项。
- 注意，每个虚拟机的网络适配器要调为桥接模式（右键虚拟机，点击设置，即可查看），如下：

（4）运行路由器程序

1.网络设备列表输出，选择打开双IP的网卡，即Device 9

```
Device 8: rpcap://\Device\NPF_{BA1DE218-79C5-4EE7-AA9D-8CF9DEF2FEFE}
   Description: Network adapter 'Microsoft Wi-Fi Direct Virtual Adapter' on local host
地址类型为IPv4 IP地址:  169.254.29.31
子网掩码: 255.255.0.0
地址类型为IPV6

Device 9: rpcap://\Device\NPF_{D6C6BB0D-8137-4E31-B164-54928D5A907A}
   Description: Network adapter 'VirtualBox Host-Only Ethernet Adapter' on local host
地址类型为IPv4 IP地址:  206.1.2.1
子网掩码: 255.255.255.0
地址类型为IPv4 IP地址:  206.1.1.1
子网掩码: 255.255.255.0
地址类型为IPV6

Device 10: rpcap://\Device\NPF_{4E76B6F8-28EE-16A8-F494-25D13FA82B68}
   Description: Network adapter 'Palantir Tunnel' on local host
地址类型为IPv4 IP地址:  198.18.0.1
子网掩码: 255.255.255.252
地址类型为IPV6
```

```
Device 14: rpcap://\Device\NPF_{911CDDC2-41DD-46C6-8DA6-D6D55C43A49F}
   Description: Network adapter 'OpenVPN Data Channel Offload' on local host
地址类型为IPv4 IP地址:  172.21.62.150
子网掩码: 255.255.254.0
地址类型为IPv4 IP地址:  169.254.211.203
子网掩码: 255.255.0.0
地址类型为IPV6

Device 15: rpcap://\Device\NPF_{799BD148-2436-4441-BAE7-762969BE800E}
   Description: Network adapter 'Realtek PCIe GbE Family Controller' on local host
地址类型为IPv4 IP地址:  169.254.159.213
子网掩码: 255.255.0.0
地址类型为IPV6

请选择你想打开的接口: `1 ~ 15`:
9
本机MAC地址为:  0A-00-27-00-00-16
```

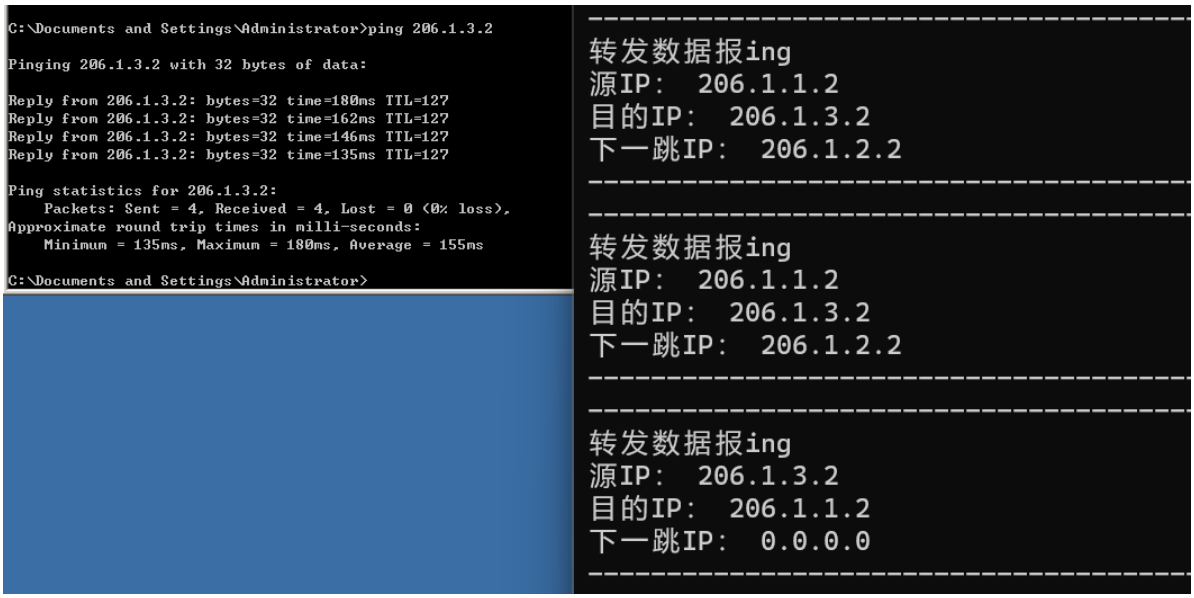
2.手动添加路由

输入1，然后输入网络掩码、目的网络IP地址、下一跳IP地址；接着可以输入3，来打印路由表，查看是否添加成功。注意，路由表中的前两项，是初始化路由表时就**自动添加**的。

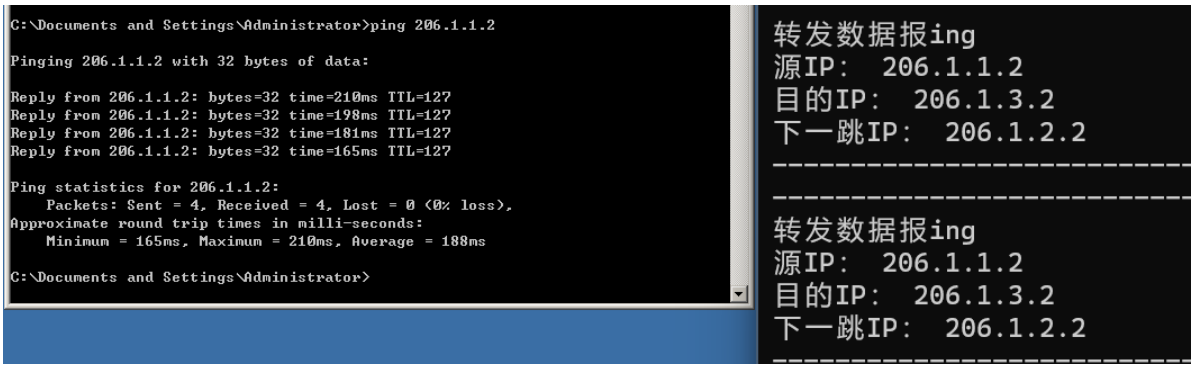
```
================================================================================
欢迎来到高级路由器，请选择你想要进行的操作:
1. 添加路由表项
2. 删除路由表项
3. 打印路由表:
4. 退出程序
================================================================================
1
请输入网络掩码:
255.255.255.0
请输入目的网络`ip`地址:
206.1.3.0
请输入下一跳`ip`地址:
206.1.2.2
================================================================================
欢迎来到高级路由器，请选择你想要进行的操作:
1. 添加路由表项
2. 删除路由表项
3. 打印路由表:
4. 退出程序
================================================================================
3
路由表项0 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.1.0        下一跳IP地址为:  0.0.0.0          类型为:   直接相连
路由表项1 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.2.0        下一跳IP地址为:  0.0.0.0          类型为:   直接相连
路由表项2 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.3.0        下一跳IP地址为:  206.1.2.2        类型为:   用户添加
================================================================================
```

3.进行ping和tracert检验

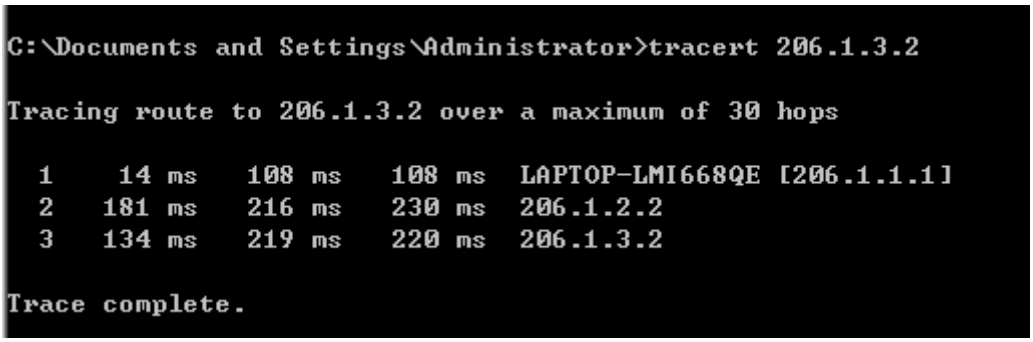
- 在主机A上，去ping主机B，还有相关的日志输出。



- 在主机B上，去ping主机A，还有相关的日志输出。



- 在主机A上，去tracert主机B，可以看出，路径正确



- 在主机B上，去tracert主机A，可以看出，路径也是正确的

4.日志输出文件

更详细的日志都输出在同目录下的Log.txt文件中：

```
Log.txt
文件    编辑    查看

ARP:IP:  206.1.2.1  MAC:  0A-00-27-00-00-16
ARP:IP:  206.1.2.1  MAC:  0A-00-27-00-00-17
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
ARP:IP:  206.1.2.2  MAC:  00-0C-29-A3-AD-E9
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`接收: 源IP:  206.1.1.2  目的IP:  206.1.1.2
`IP`转发: 源IP:  206.1.1.2  目的IP:  206.1.1.2
```

5.删除路由表项

- 删除默认添加的路由时，删除失败！

```
路由表项0 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.1.0        下一跳IP地址为:  0.0.0.0          类型为:   直
接相连
路由表项1 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.2.0        下一跳IP地址为:  0.0.0.0          类型为:   直
接相连
路由表项2 :  子网掩码为:  255.255.255.0    目的网络为:  206.1.3.0        下一跳IP地址为:  206.1.2.2        类型为:   用
户添加
=================================================================================
欢迎来到高级路由器，请选择你想要进行的操作：
1. 添加路由表项
2. 删除路由表项
3. 打印路由表：
4. 退出程序
=================================================================================
2
请输入你想要删除的表项编号：
0
该项无法删除
```

- 删除手动添加的路由，删除成功！

```
欢迎来到高级路由器，请选择你想要进行的操作：
1. 添加路由表项
2. 删除路由表项
3. 打印路由表：
4. 退出程序
================================================================================
2
请输入你想要删除的表项编号：
2
删除成功！
================================================================================
欢迎来到高级路由器，请选择你想要进行的操作：
1. 添加路由表项
2. 删除路由表项
3. 打印路由表：
4. 退出程序
================================================================================
3
路由表项0 ：  子网掩码为：  255.255.255.0    目的网络为：  206.1.1.0       下一跳IP地址为：  0.0.0.0         类型为：   直接相连
路由表项1 ：  子网掩码为：  255.255.255.0    目的网络为：  206.1.2.0       下一跳IP地址为：  0.0.0.0         类型为：   直接相连
================================================================================
```

6.退出路由器程序

```
================================================================================
欢迎来到高级路由器，请选择你想要进行的操作：
1. 添加路由表项
2. 删除路由表项
3. 打印路由表：
4. 退出程序
================================================================================
4

C:\Users\86180\source\repos\路由器\x64\Debug\路由器.exe (进程 26512)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

六、实验深入

（1）遇到的问题

完成程序后，进行ping通实验，发现可以ping通。然后又进行tracert实验，发现无法获取路径。后来发现是我转发数据包时，转发的数据包长度不对导致的。我之前的代码如下：

```
//int rtn = pcap_sendpacket(handle, (const u_char*)temp, 74);
```

只传输了74字节，一开始我以为，这是因为没有把ICMP报文头完整传输过去，所以，相当于源主机没有收到ICMP超时报文，从而出现无法显示路径的情况。于是我更改为：

```
//计算数据包长度
int packetLength = sizeof(FrameHeader_t) + sizeof(IPHeader_t) + sizeof(ICMP_t);
// 发送数据报
int rtn = pcap_sendpacket(handle, (const u_char*)temp, packetLength);
```

这样就将数据包完整的发送过去了，进行tracert，如下：

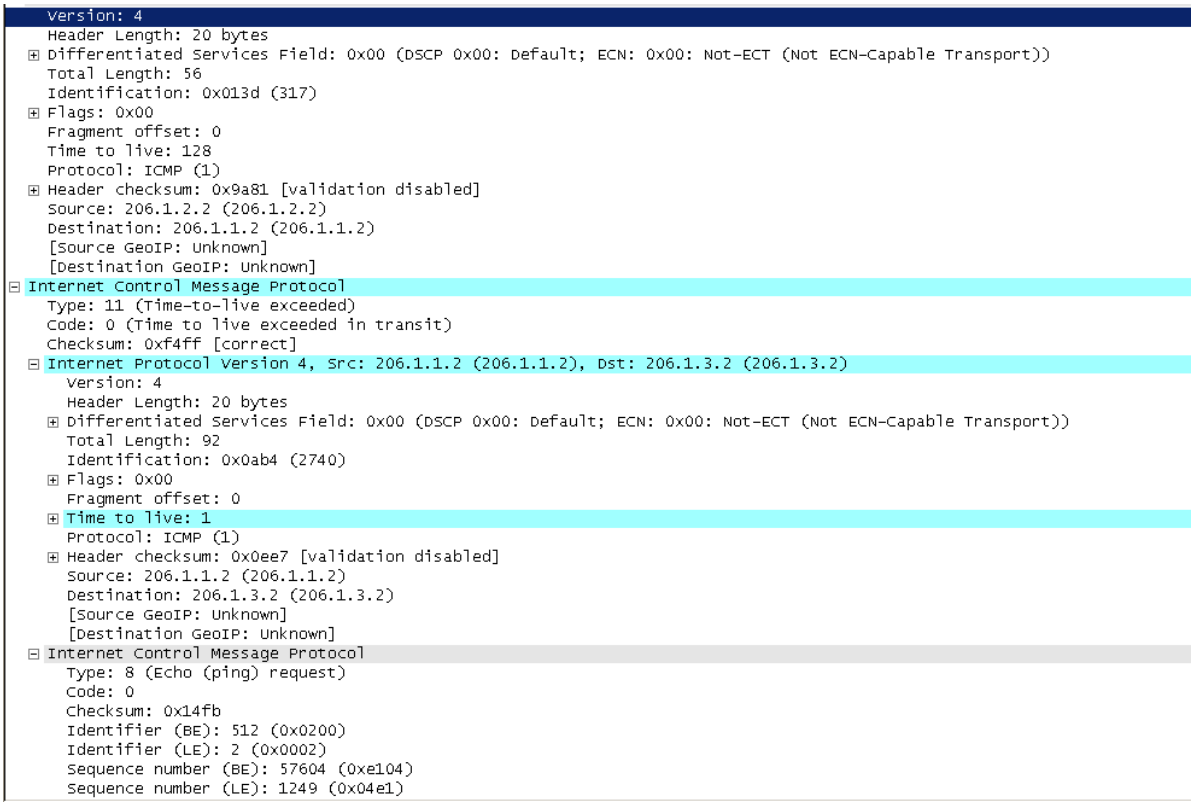
```
C:\Documents and Settings\Administrator>tracert 206.1.3.2

Tracing route to 206.1.3.2 over a maximum of 30 hops

  1     *        *        *     Request timed out.
  2   161 ms   220 ms   213 ms  206.1.2.2
  3   148 ms   218 ms   214 ms  206.1.3.2

Trace complete.
```

第一个，也就是运行路由器程序的本机却是Request timed out。通过在IP地址为206.1.1.2的虚拟机上（即发送tracert命令的主机）用**Wireshark**抓包，发现本机并没有发送ICMP超时报文。所以，我又尝试自己发送ICMP超时报文。通过用Wireshark抓到的ICMP超时报文，我仿照里面的内容进行设置。如下图：



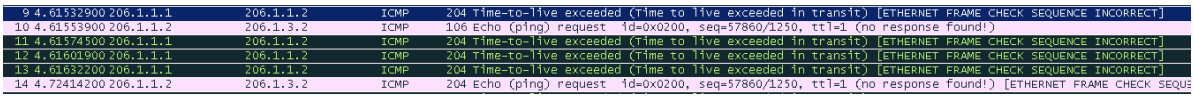
根据这个，我实现了如下代码：

```
// 如果TTL字段的值为0，那么需要发送一个ICMP超时消息
if (data->IPHeader.TTL == 0) {
                //构造ICMP超时报文
                ICMP_t* icmp_packet = new ICMP_t(*data);
                //清空
                memset(icmp_packet, 0, sizeof(ICMP_t));
                //设置帧头部
                icmp_packet->FrameHeader.FrameType = htons(0x0800);
                //设置IP头部
                icmp_packet->IPHeader.Ver_HLen = 0b01000101;
                icmp_packet->IPHeader.TOS = 0;
                icmp_packet->IPHeader.TotalLen =
htons(sizeof(IPHeader_t)+sizeof(ICMPHeader_t)+0x80);
                icmp_packet->IPHeader.Flag_Segment = htons(0);
                icmp_packet->IPHeader.ID = data->IPHeader.ID;
                icmp_packet->IPHeader.TTL = 127;     //生存时间
                icmp_packet->IPHeader.Protocol = 1; // ICMP的协议号
                icmp_packet->IPHeader.SrcIP = inet_addr(OwnIP_2);
                icmp_packet->IPHeader.DstIP = data->IPHeader.SrcIP;
                //计算校验和
                setchecksum_IP(&icmp_packet->IPHeader);
                // 设置ICMP类型和代码
                icmp_packet->ICMPHeader.Type = 11; // ICMP超时类型
                icmp_packet->ICMPHeader.Code = 0; // 超时的代码
                icmp_packet->ICMPHeader.Checksum = htons(0x040f);
                //将接收到的IP数据包的IP头部和数据部分的前64bit(就是ICMP头部)放到数据部分
```

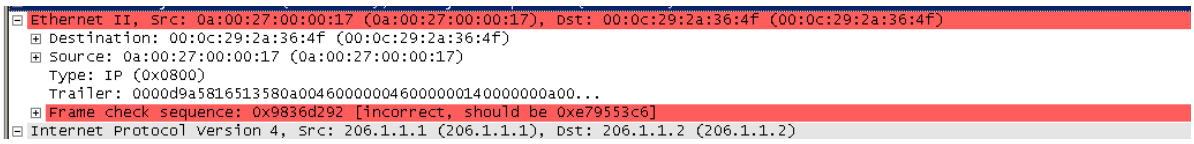
```
                memcpy(icmp_packet->buf, &data->IPHeader, sizeof(data-
>IPHeader));
                memcpy(icmp_packet->buf + sizeof(data->IPHeader), &data-
>ICMPHeader, sizeof(data->ICMPHeader));
                //获取源IP和目的IP
                SourceIP = icmp_packet->IPHeader.SrcIP;
                DestIP = icmp_packet->IPHeader.DstIP;
                //查找路由表，找相关的路径
                NextIP = routetable.lookup(DestIP);
                //如果找到了
                if (NextIP != -1) {
                                //用于存储获取到的MAC地址
                                BYTE mac[6];
                                if (NextIP == 0) {
                                    //发送ARP请求，寻找下一跳的MAC地址
                                    if (!ARPtable::lookup(DestIP, mac)) {
                                        //insert()函数中会发送ARP请求
                                        ARPtable::insert(DestIP, mac, handle,
WorkLog);
                                    }
                                    // 发送超时ICMP报文
                                    send(*icmp_packet, mac, handle);

                                }
                                else {
                                    //发送ARP请求，寻找下一跳的MAC地址
                                    if (!ARPtable::lookup(NextIP, mac)) {
                                        ARPtable::insert(NextIP, mac, handle,
WorkLog);
                                    }
                                    //计算IP头部校验和
                                    setchecksum_IP(&icmp_packet->IPHeader);
                                    // 发送超时ICMP报文
                                    send(*icmp_packet, mac, handle);
                                }
                }
}
```

但是依旧不行，不过有了很大的突破，那就是发送tracert命令的虚拟机收到了来自本机的ICMP超时报文！

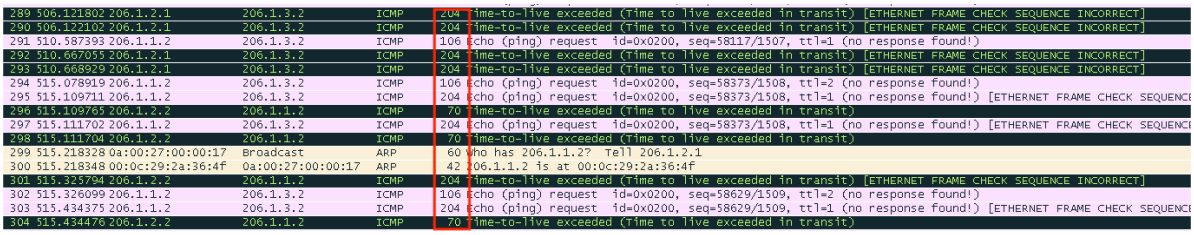


不过，所发的所有ICMP超时报文的（Frame check sequence）FCS码都不正确，如下图：



我又探究了很久，反复比对正确的ICMP超时报文格式和我自己构造的ICMP超时报文格式有什么区别？

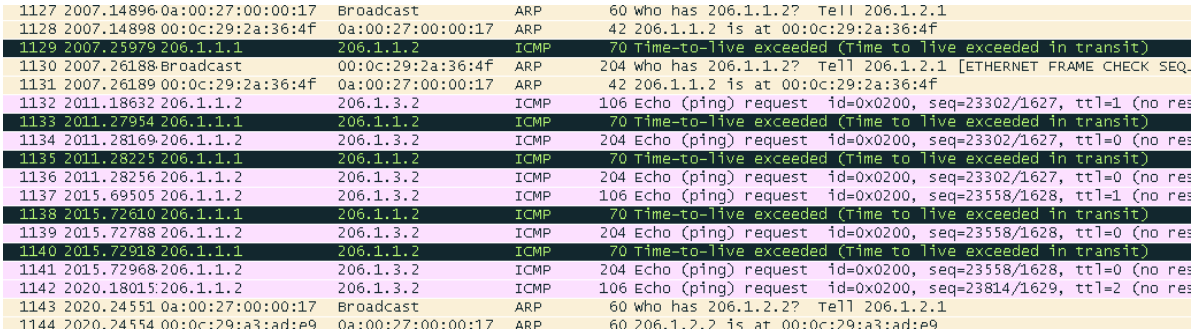
发现长度为70的都是正确的，长度为204的，即`sizeof(FrameHeader_t) + sizeof(IPHeader_t) + sizeof(ICMP_t)`的值，都是错误的。如下：



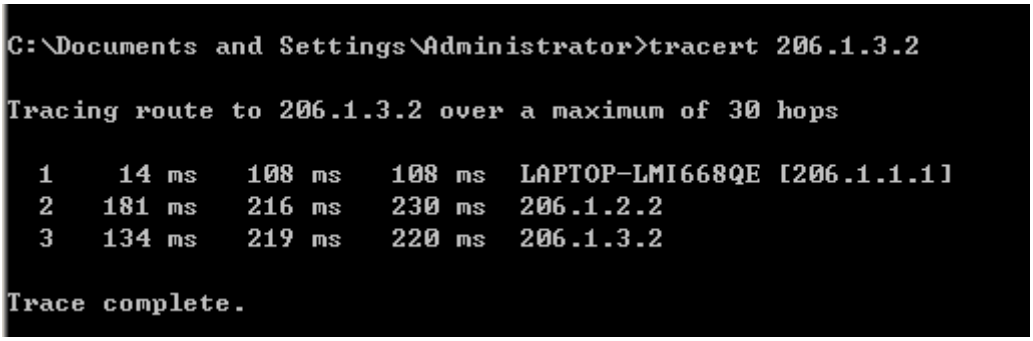
于是，我将代码改为了发送长度为70的ICMP报文，但是，主机依然不识别这个ICMP超时报文，于是我又详细比对了ICMP超时报文之间的区别，但并没有发现什么问题。后来查资料，了解到IP头部的ID要**唯一标识**，而我之前的理解是错误的，我将ID改为了接收数据包的ID，于是我将ID设置为0，如下：

```
icmp_packet->IPHeader.ID = 0;
```

构造的ICMP报文总算正确了，如下：



经过尝试，解决了问题，如下：



（2）原理探究

我是在主机运行的路由器程序，所以我需要在主机的某个网卡配置双IP。在本次实验中，我选择以太网3（VirtualBox Host-Only Ethernet Adapter）网卡。下面，我来探究下，为什么打开这个网卡，可以ping通虚拟机。

首先，我们先了解下虚拟机的相关网络概念：

- **虚拟网卡的角色**：

1. **在宿主机上**：当你创建虚拟机时，虚拟化软件会在宿主机上创建一个或多个虚拟网卡。这些虚拟网卡在操作系统中看起来就像真正的硬件设备，但实际上完全是由软件模拟的。
2. **在虚拟机内**：虚拟机自己也会有一张虚拟网卡，它代表虚拟机与外界（可能是宿主机或其他网络）的连接点。

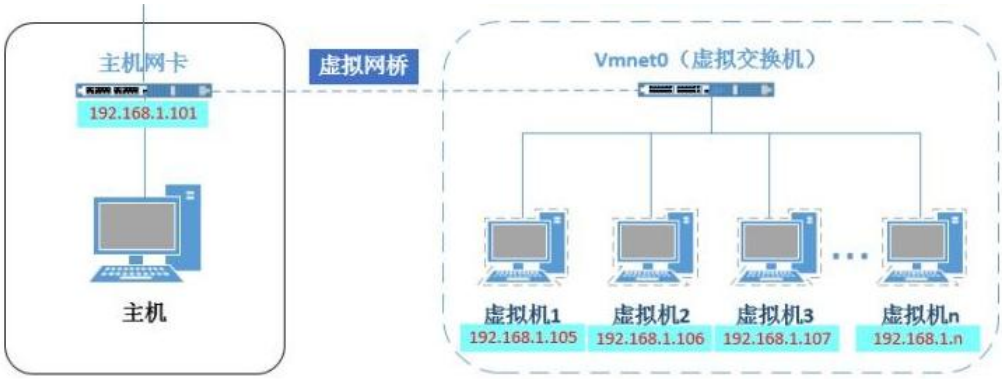
- **工作原理**：

1. **网络桥接模式**：虚拟化软件会将虚拟机的虚拟网卡与宿主机的物理网卡“桥接”在一起。这意味着虚拟机可以像宿主机上的任何其他物理设备一样访问网络。它会从路由器或DHCP服务器获取自己的IP地址。

2. **NAT（网络地址转换）模式**：虚拟机的所有网络流量都会通过宿主机的物理网卡传输，并使用宿主机的IP地址。这种方式下，虚拟机是不能被网络上的其他设备直接访问的，但它可以访问外部网络。
3. **主机模式**：虚拟机只能与宿主机通信，不能与外部网络通信。这通常用于测试和隔离环境。

然后，我们重点说一下桥接模式：

桥接模式是将主机网卡与虚拟机虚拟的网卡利用虚拟网桥进行通信。**在桥接的作用下，类似于把物理主机虚拟为一个交换机，所有设置桥接模式的虚拟机都将连接到这个交换机的一个接口上。同样物理主机也插在这个交换机中，所以桥接下的网卡与网卡都是交换模式的，可以相互访问而不干扰。**它的网络结构如下：



- 虚拟网桥会转发主机网卡接收到的广播和组播信息，以及目标为虚拟交换机网段的单播。所以，与虚拟交换机连接的虚拟网卡(如:eth0、eth1等)**接收到了以太网3发出的信息。**
- 桥接模式是通过虚拟网桥将主机上的网卡与虚拟交换机Vmnet0连接在一起，虚拟机上的虚拟网卡(并不是VMware Network Adapter VMnet1和VMware Network Adapter VMnet8)都连接在虚拟交换机Vmnet0上，所以桥接模式的虚拟机IP必须与主机在同一网段且子网掩码、网关与DNS也要与主机网卡一致。

七、实验总结

在本次实验中，几乎将本学期网技所学都应用了一遍，对整个课程有了更进一步的理解。并且，在调试bug的过程中，熟练掌握了Wireshark的使用，将各个报文格式又熟记于心。

github链接：https://github.com/happy206/Network-Technology-and-Applications