计算机网络实验报告

实验四

静态路由编程实现

学号：1412200065

姓名：刘博

时间：2016.4.18

1. 实验目的：

本实验主要目的设计和实现一个简单的静态路由机制，用以取代 Linux 资深通过 ip forwarding 实现的静态路由方式，进而加深对二三层协议衔接即静态路由的理解。

2. 数据结构说明：

以太网帧头部：

```
typedef struct ethernet {
        unsigned char dst[6];
        unsigned char src[6];
        unsigned char type[2];
} eth_head;
```

Dst：目的 mac 地址；
Src：源 mac 地址；
Type：上层协议类型；

Ip 头部：

```
typedef struct IP {
        unsigned char IHL : 4;                  //Internet Head Length
        unsigned char version : 4;              //IP Version
        unsigned char ECN : 2;
        unsigned char DS : 6;                   //Determine Service
        unsigned short length;                  //Length of datagram
        unsigned short label;
        unsigned short offset : 13;
        unsigned char tag : 3;
        unsigned char live;
        unsigned char protocol;
        unsigned short check_sum;
        unsigned char ip_src[4];                //Source Address
        unsigned char ip_dst[4];                //Destination Address
} ip_head;
```

IHL：网络头部长度；
Version：ip 版本号；
Live：ttl 数据报生存期；
Protocol：上层协议类型；
Check_sum：校验和；
Ip_src：源 ip 地址；
Ip_dst：目的 ip 地址；

ARP 头部：

```c
typedef struct ARP {
        unsigned char mac_target[6];
        unsigned char mac_source[6];
        unsigned short ethertype;
        unsigned short hw_type;
        unsigned short proto_type;
        unsigned char mac_addr_len;
        unsigned char ip_addr_len;
        unsigned short op;
        unsigned char mac_sender[6];
        unsigned char ip_sender[4];
        unsigned char mac_receiver[6];
        unsigned char ip_receiver[4];
        unsigned char padding[18];
} arp_head;
```

Mac_target：目标 mac 地址；

Mac_source：源 mac 地址；

Hw_type：硬件类型（网卡编号）；

Proto_type：协议类型；

Mac_addr_len：mac 地址长度；

Ip_addr_len：ip 地址长度；

Op：操作类型；

Mac_sender：mac 发送者地址；

Ip_sender：ip 发送者地址；

Mac_receiver：mac 接受者地址；

Ip_receiver：ip 接受者地址；

Padding：附加段；

路由表项：

```c
typedef struct route_item {
        unsigned char destination[16];
        unsigned char gateway[16];
        unsigned char netmask[16];
        unsigned char interface[16];
} route_item;
```

Destination：目的 ip 地址；

Gateway：网关地址；

Netmask：网络掩码；

Interface：网络接口（网卡名）；

Arp 表项：

```c
typedef struct arp_item {
        unsigned char ip_addr[16];
        unsigned char mac_addr[18];
} arp_item;
```

Ip_addr：ip 地址；

Mac_addr：mac 地址；

Device 表项：

```
typedef struct device_item {
        unsigned char interface[16];
        unsigned char ip_addr[16];
        unsigned char mac_addr[18];
} device_item;
```

Interface：网卡接口（网卡名）；
Ip_addr：网卡 ip 地址；
Mac_addr：网卡 mac 地址；

3. 配置文件说明：

ip 配置：

router1：

ifconfig eth0 192.168.0.1 netmask 255.255.255.0
ifconfig eth1 192.168.1.1 netmask 255.255.255.0

router2：

ifconfig eth0 192.168.1.2 netmask 255.255.255.0
ifconfig eth1 192.168.2.1 netmask 255.255.255.0

pc1：

ifconfig eth0 192.168.0.2 netmask 255.255.255.0

pc2：

ifconfig eth0 192.168.2.2 netmask 255.255.255.0

route 配置：

router1：

ip route add 192.168.2.0/24 via 192.168.1.2
echo 0 > /proc/sys/net/ipv4/ip_forward

router2：

ip route add 192.168.0.0/24 via 192.168.1.1
echo 1 > /proc/sys/net/ipv4/ip_forward

pc1：

route add default gw 192.168.0.1

pc2：

route add default gw 192.168.2.1

configuration 配置：
route_table.txt：
192.168.1.0 * 255.255.255.0 eth1
192.168.0.0 * 255.255.255.0 eth0

192.168.2.0 192.168.1.2 255.255.255.0 eth1

device_table.txt：
eth0 192.168.0.1 00:0c:29:77:9e:74
eth1 192.168.1.1 00:0c:29:77:9e:7e

4. 程序设计思路及运行流程：

设计思路：
设计思路基于静态路由流程进行设计，首先需要理解静态路由的转发流程，然后就可以进行设计；
首先路由器进行初始化，直接将路由表和设备表进行读取和赋值：

```
read_settings();
```

```
void read_settings(void) {
        FILE *fp = fopen("route_table.txt","r");
        while(!feof(fp))
        {
                route_item *p = (route_item *)route_info + route_item_index;
                fscanf(fp,"%s %s %s %s",p->destination,p->gateway,p->netmask,p-
>interface);
                route_item_index++;
        }
        route_item_index--;
        //route item index for all route items
        fclose(fp);
        fp = fopen("device_table.txt","r");
        while(!feof(fp))
        {
                device_item *p = (device_item *)device + device_item_index;
                fscanf(fp,"%s %s %s",p->interface,p->ip_addr,p->mac_addr);
                strcpy(arp_table[arp_item_index].ip_addr,p->ip_addr);
                strcpy(arp_table[arp_item_index].mac_addr,p->mac_addr);
                device_item_index++;
                arp_item_index++;
        }
        device_item_index--;
        //device item index for all device items
        fclose(fp);
}
```

Read_settings 函数进行初始化，包括路由表，设备表，arp 表的初始化；
初始化完成后，创建套接字，监听路由器的所有接口：

```
if((sock_fd = socket(PF_PACKET,SOCK_RAW,htons(ETH_P_ALL))) < 0)
{
        printf("error create raw socket\n");
        return -1;
}
```

每当收到一个数据包是，路由器需要根据其以太网的地址来确定该包是否属于 arp 包，如果是 arp 包，则进行 arp 表的更新，如果不是，则进行 ip 判断，如果是 ip 包，则进行 ip 转发处理，否则丢弃该包；
下面分别介绍 arp 和 ip 处理：

1. arp 处理：

若接收到的是 arp 包，路由器首先检查该 arp 中的对应规则是否在自己的 arp 表中存

在，如果是，则不作进一步处理，如果不是，则将新的规则加入自己当前的 arp 表项中；

```
//hit arp_packet, reflash the arp_buffer
if(*(unsigned short *)(eth->type) == htons(ETHER_ARP))
{
        make_up_arp((arp_head *)buffer);
        continue;
}
```

（抓到了 arp 数据包）

```
void make_up_arp(arp_head *arp) {
        if(arp->op == htons(ARP_REQUEST) || arp->op == htons(ARP_REPLY))
        {
                char *ip_temp = inet_ntoa(*(struct in_addr *)&arp->ip_sender);
                unsigned char mac_temp[18] = {0};
                unsigned char *mac = arp->mac_sender;
                sprintf(mac_temp,"%02x:%02x:%02x:%02x:%02x:%02x",mac[0],mac
[1],mac[2],mac[3],mac[4],mac[5]);
                if(arp_local(ip_temp) == -1)
                        add_arp(ip_temp,mac_temp);
        }
}
```

（读取 arp 包并进行判断）

```
int arp_local(unsigned char *arp_temp) {
        int i;
        for(i = 0; i < arp_item_index; i++)
        {
                if(strcmp(arp_table[i].ip_addr,arp_temp) == 0)
                        return i;
        }
        return -1;
}

void add_arp(unsigned char *new_ip, unsigned char *new_mac) {
        if(arp_item_index < MAX_ARP_SIZE)
        {
                strcpy(arp_table[arp_item_index].ip_addr,new_ip);
                strcpy(arp_table[arp_item_index].mac_addr,new_mac);
                arp_item_index++;
        }
        else
        {
                printf("no enough space to restore arp");
        }
        return;
}
```

（判断 arp 是否存在，如果不是，则加入当前的 arp 表中）

2. 不是 arp 包和 ip 包：丢弃！

```
//hit neither arp nor ip
if(*(unsigned short *)(eth->type) != htons(ETHER_IP))
        continue;
```

3. Ip 包处理：

当路由器抓获一个 ip 包时，路由器进行 ip 地址读取：

```
//hit ip protocal
ip = (void *)((unsigned char *)eth + sizeof(eth_head));

sprintf(ip_src,"%d.%d.%d.%d",ip->ip_src[0],ip->ip_src[1],ip->ip_src[2],ip->ip_src[3]);
sprintf(ip_dst,"%d.%d.%d.%d",ip->ip_dst[0],ip->ip_dst[1],ip->ip_dst[2],ip->ip_dst[3]);
```

根据 ip 地址，判断该包是否是发往本路由器的，如果是发往自身的，则不进行转发，直接回复：

```
//check whether to be forward
if(!need_forward(ip->ip_dst,ip->ip_src))
{
        printf("don't need forward!\n");
        continue;
}
```

（判断是否属于本路由器，如果是，则不再进行转发）

```
bool need_forward(unsigned char *src, unsigned char *dst) {
        int i;
        for(i = 0; i < device_item_index; i++)
        {
                unsigned int addr;
                inet_aton(device[i].ip_addr,(struct in_addr *)&addr);
                //if the dst or the src ip belongs to router,no need to forward
                if(*(unsigned int *)src == addr || *(unsigned int *)dst == addr)
                return false;
        }
        return true;
}
```

如果该 ip 包不是发往本路由器的，那么就需要进行转发，首先根据 route 表找到当前目标 ip 地址的下一跳的 ip 地址：

```
//read route table and find ip route item index(find next ip)
int route_temp_index = 0;
route_temp_index = find_ip(ip);

char temp_ip[18] = {0};
char *next_ip = (void *)temp_ip;
if(*route_info[route_temp_index].gateway != '*')
        strcpy(next_ip,route_info[route_temp_index].gateway);
else
        strcpy(next_ip,ip_dst);
//printf("next ip = %s\n",route_info[route_temp_index].gateway);

if(route_temp_index == -1)
        continue;
```

（如果是 gateway 是*表示网关为同一个子网，可以直接发送，无需地址转换）

如果 route 表找不到该目的 ip 地址，则显示无法连接（unreachable）

```
int find_ip(ip_head *ip) {
        unsigned int addr;
        unsigned int mask;
        int i;
        for(i = 0; i < route_item_index; i++)
        {
                inet_aton(route_info[i].destination,(struct in_addr *)&addr);
                inet_aton(route_info[i].netmask,(struct in_addr *)&mask);
                if(addr== ((*(unsigned int *)ip->ip_dst) & mask))
                {
#ifdef DEBUG
                        printf("find ip hit route table : %d\n",i);
#endif
                        return i;
                }
        }
        printf("no hit on ip\n");
        return -1;
}
```

（注意此处需要用到掩码）

找到下一跳的 ip 地址后，根据 arp 表找到下一跳的网卡接口地址：

```
                //read device table and find device interface for mac
                int device_temp_index = 0;
                device_temp_index = find_device(route_info
[route_temp_index].interface);
                if(device_temp_index == -1)
                        return -1;
```

（找到对应的 device 接口）

```
int find_device(unsigned char *interface) {
        int i;
        for(i = 0; i < device_item_index; i++)
        {
                if(strcmp(interface, device[i].interface) == 0)
                {
#ifdef DEBUG
                        printf("hit device on device : %s\n",device
[i].interface);
#endif
                        return i;
                }
        }
        printf("no hit on device\n");
        return -1;
}
```

找到网卡接口后，新的以太网帧（转发包）的源地址就确定了，接下来需要确定目的地址：

根据目的 ip 地址进行 arp 转换寻找 mac 地址：

```
//read arp table and find arp item index(find mac for next ip)
int arp_temp_index = 0;
arp_temp_index = find_arp(next_ip);
```

（寻找当前 arp 表中是否存在该 ip 地址）

```
int find_arp(char *next_ip) {
        int i;
        for(i = 0; i < arp_item_index; i++)
        {
                if(strcmp(next_ip,arp_table[i].ip_addr) == 0)
                {
#ifdef DEBUG
                        printf("find mac hit arp table : %d\n",i);
#endif
                        return i;
                }
        }
        printf("no hit on mac\n");
        return -1;
}
```

如果当前 arp 表中没有对应的 ip 地址存在，则进行广播发送 arp 包询问：

```
                while(arp_temp_index == -1)
                {
                        new_arp(device[device_temp_index].mac_addr,device
[device_temp_index].ip_addr,next_ip,route_info[route_temp_index].interface);
                        arp_temp_index = find_arp(next_ip);
                }
```

其中 new_arp（）函数就是用来发送 arp 询问请求的：

```
void new_arp(unsigned char *mac_addr,unsigned char *src_ip,unsigned char
*dst_ip,unsigned char *interface) {
        int sock_fd;
        arp_head arp;
        struct in_addr sender,receiver;
        struct sockaddr_ll sl;

        if((sock_fd = socket(AF_PACKET,SOCK_RAW,htons(ETH_P_ARP))) < 0)
        {
                printf("error create arp packet!\n");
                return;
        }

        //fill arp packet
        memset(&arp,0, sizeof(arp_head));

        //ether addr
        copy_mac("ff:ff:ff:ff:ff:ff",arp.mac_target);
        copy_mac(mac_addr,arp.mac_source);

        arp.ethertype = htons(ETHER_ARP);
        arp.hw_type = htons(0x1);;
        arp.proto_type = htons(ETHER_IP);
        arp.mac_addr_len = 6;
        arp.ip_addr_len = 4;
        arp.op = htons(ARP_REQUEST);
```

```
        //arp addr
        copy_mac(mac_addr,arp.mac_sender);
        inet_aton(src_ip,&sender);
        memcpy(&arp.ip_sender,&sender,sizeof(sender));
        inet_aton(dst_ip,&receiver);
        memcpy(&arp.ip_receiver,&receiver,sizeof(receiver));

        //create struct sockaddr for sendto
        struct sockaddr_ll addr;
        memset(&addr,0,sizeof(addr));
        addr.sll_family = AF_PACKET;
        addr.sll_protocol = htons(ETH_P_ARP);
        struct ifreq req;
        strcpy(req.ifr_name,interface);
        int s;
        if((s = socket(AF_PACKET,SOCK_DGRAM,0)) < 0)
                printf("socket AF_INET error\n");
        ioctl(s,SIOCGIFINDEX,&req);
        close(s);
        memset(&sl,0,sizeof(sl));
        sl.sll_family = AF_PACKET;
        sl.sll_ifindex = req.ifr_ifindex;

        //send
        int len = sendto(sock_fd, &arp, sizeof(arp), 0, (struct sockaddr *)&sl,
sizeof(sl));
        int n_read;
        unsigned char buffer[100];

        //receive
        n_read = recvfrom(sock_fd, buffer, 2048, 0, NULL, NULL);
        make_up_arp((arp_head *)buffer);
}
```

首先发送 arp 请求的地址是广播地址（ff：ff：ff：ff：ff：ff），然后将对应的以太网头部协议，地址，网卡设备号依次填好，用 sendto（）函数进行发送；收到的包进行 arp 分析；并将其加入到 arp 表中；

确定了下一跳的 mac 地址，就可以将以太网帧 mac 地址进行修改，成为转发包：

```
//change the head of ethernet
copy_mac(device[device_temp_index].mac_addr,eth->src);
copy_mac(arp_table[arp_temp_index].mac_addr,eth->dst);
```

最后进行发送整个以太网帧即可：

```
                struct sockaddr_ll addr;
                bzero(&addr,sizeof(addr));
                addr.sll_family = PF_PACKET;
                struct ifreq req;
                strcpy(req.ifr_name, route_info[route_temp_index].interface);
                ioctl(sock_fd,SIOCGIFINDEX,&req);
                addr.sll_ifindex = req.ifr_ifindex;

                //send message
                if(sendto(sock_fd,buffer,n_read,0,(struct sockaddr *)
(&addr),sizeof(addr)) < 0)
                {
                        printf("send error\n");
                }
```

*归纳来讲整个运行的流程如下：

1. 路由程序启动
2. 读取配置文件并初始化数据结构
3. 监听所有的网口
4. 捕获来自网口的数据包
5. 判断数据包类型
    ARP报文，将发送方的信息登记入ARP table，转到步骤3
    IP报文，转到步骤6
    其他，丢弃该数据包，转到步骤3
6. 判断是否需要转发，如果不需要，转到步骤3
7. 在路由表中查询下一跳的IP地址，如果无此表项，则网络不可达，丢弃该数据包，转到步骤3
8. 根据下一跳的发送接口查询设备表，获得发送网卡的MAC地址，无此表项则程序出错
9. 根据下一跳IP地址查询ARP表
    无此表项则发送ARP请求，并将ARP信息登记入ARP表，转到步骤9
    有此表项则获取下一跳MAC地址，转到步骤10
10. 将数据包中以太层的源MAC地址和目的MAC地址改为路由器发送设备的MAC地址和下一跳的MAC地址
11. 将IP报头的ttl减一
12. 重新计算IP报头的checksum
13. 发送该数据包
14. 转发完成，转到步骤3

其中计算 checksum 以及 ttl 改变如下：

```
//change ttl and checksum
ip->check_sum = 0;
int n = (*(unsigned char *)ip)&0x0f;
ip->live--;
ip->check_sum = in_cksum((unsigned short*)ip,n*4);
```

至此路由转发功能完成；

5. 运行结果截图：

打开 router1 的 ip_forward 时：

```
root@ubuntu:/home/kirito# cat /proc/sys/net/ipv4/ip_forward
1
```

（pc1 ping pc2）

```
root@ubuntu:/home/kirito# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=62 time=1093 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=62 time=83.4 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=62 time=1.75 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=62 time=1.12 ms
64 bytes from 192.168.2.2: icmp_seq=5 ttl=62 time=1.14 ms
64 bytes from 192.168.2.2: icmp_seq=6 ttl=62 time=3.38 ms
64 bytes from 192.168.2.2: icmp_seq=7 ttl=62 time=1.26 ms
64 bytes from 192.168.2.2: icmp_seq=8 ttl=62 time=1.95 ms
64 bytes from 192.168.2.2: icmp_seq=9 ttl=62 time=1.88 ms
64 bytes from 192.168.2.2: icmp_seq=10 ttl=62 time=1.68 ms
^C
--- 192.168.2.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9022ms
rtt min/avg/max/mdev = 1.127/119.108/1093.389/325.673 ms, pipe 2
root@ubuntu:/home/kirito#
```

Router1 上对 eth0 进行 wireshark 抓包：

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
2	1.009847000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
3	1.092468000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
4	1.092691000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
5	2.010561000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
6	2.011737000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
7	3.012540000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi

```
▶Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▼Ethernet II, Src: Vmware_25:5c:1d (00:0c:29:25:5c:1d), Dst: Vmware_77:9e:74 (00:0c:29:77
  ▶Destination: Vmware_77:9e:74 (00:0c:29:77:9e:74)
  ▶Source: Vmware_25:5c:1d (00:0c:29:25:5c:1d)
   Type: IP (0x0800)
▼Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.2.2 (192.168.2
```

（可以看到当打开路由转发功能时，对 eth0 而言，其以太网源地址是 pc1 的 mac 地址，目的地址是 eth0 的 mac 地址）

Router1 上对 eth1 进行 wireshark 抓包：

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
2	0.001460000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
3	1.002337000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
4	1.003448000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
5	2.004474000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
6	2.005860000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
7	3.006027000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi

```
▶Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▼Ethernet II, Src: Vmware_77:9e:7e (00:0c:29:77:9e:7e), Dst: Vmware_6b:19:62 (00:0c:29:6b
  ▶Destination: Vmware_6b:19:62 (00:0c:29:6b:19:62)
  ▶Source: Vmware_77:9e:7e (00:0c:29:77:9e:7e)
   Type: IP (0x0800)
▼Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.2.2 (192.168.2
```

（可以看到当打开路由转发功能时，对 eth1 而言，其以太网源地址是 router1 的 eth1 地址，目的地址是 pc2 的 mac 地址）

此时 pc1 和 pc2 可以互相 ping 通；

下面关闭 router1 的 ip_forward：

```
root@ubuntu:/home/kirito# cat /proc/sys/net/ipv4/ip_forward
0
```

打开编写好的程序（router）：

（pc1 ping pc2）

```
root@ubuntu:/home/kirito# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=62 time=5.49 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=62 time=1.23 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=62 time=2.20 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=62 time=2.44 ms
64 bytes from 192.168.2.2: icmp_seq=5 ttl=62 time=2.12 ms
64 bytes from 192.168.2.2: icmp_seq=6 ttl=62 time=1.23 ms
64 bytes from 192.168.2.2: icmp_seq=7 ttl=62 time=2.39 ms
64 bytes from 192.168.2.2: icmp_seq=8 ttl=62 time=2.21 ms
64 bytes from 192.168.2.2: icmp_seq=9 ttl=62 time=1.98 ms
64 bytes from 192.168.2.2: icmp_seq=10 ttl=62 time=2.24 ms
64 bytes from 192.168.2.2: icmp_seq=11 ttl=62 time=2.44 ms
64 bytes from 192.168.2.2: icmp_seq=12 ttl=62 time=2.39 ms
64 bytes from 192.168.2.2: icmp_seq=13 ttl=62 time=2.14 ms
64 bytes from 192.168.2.2: icmp_seq=14 ttl=62 time=2.17 ms
^C
--- 192.168.2.2 ping statistics ---
15 packets transmitted, 14 received, 6% packet loss, time 14027ms
rtt min/avg/max/mdev = 1.237/2.337/5.490/0.952 ms
root@ubuntu:/home/kirito#
```

程序显示界面：

```
root@ubuntu:/home/kirito/Desktop/source# ./router
no hit on mac
eth->src = 00:0c:29:77:9e:7e
eth->dst = 00:0c:29:6b:19:62
from ip = 192.168.0.2
forward to ip = 192.168.1.2,mac = 00:0c:29:77:9e:7e via eth1
no hit on mac
eth->src = 00:0c:29:77:9e:74
eth->dst = 00:0c:29:25:5c:1d
from ip = 192.168.2.2
forward to ip = 192.168.0.2,mac = 00:0c:29:77:9e:74 via eth0
eth->src = 00:0c:29:77:9e:7e
eth->dst = 00:0c:29:6b:19:62
from ip = 192.168.0.2
forward to ip = 192.168.1.2,mac = 00:0c:29:77:9e:7e via eth1
eth->src = 00:0c:29:77:9e:74
eth->dst = 00:0c:29:25:5c:1d
from ip = 192.168.2.2
forward to ip = 192.168.0.2,mac = 00:0c:29:77:9e:74 via eth0
eth->src = 00:0c:29:77:9e:7e
eth->dst = 00:0c:29:6b:19:62
from ip = 192.168.0.2
forward to ip = 192.168.1.2,mac = 00:0c:29:77:9e:7e via eth1
eth->src = 00:0c:29:77:9e:74
```

Router1 上对 eth0 进行 wireshark 抓包：

No.	Time	Source	Destination	Protocol	Lengtł	Info
1	0.000000000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
2	0.003101000	Vmware_77:9e:74	Broadcast	ARP	60	Who has
3	0.003731000	Vmware_25:5c:1d	Vmware_77:9e:74	ARP	60	192.168.
4	0.004773000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
5	1.002377000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
6	1.003273000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
7	2.004494000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi

```
▶Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▼Ethernet II, Src: Vmware_25:5c:1d (00:0c:29:25:5c:1d), Dst: Vmware_77:9e:74 (00:0c:29:77
 ▶Destination: Vmware_77:9e:74 (00:0c:29:77:9e:74)
 ▶Source: Vmware_25:5c:1d (00:0c:29:25:5c:1d)
  Type: IP (0x0800)
▼Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.2.2 (192.168.2
```

（可以看到当关闭路由转发功能，打开程序时，对 eth0 而言，其以太网源地址是 pc1 的 mac 地址，目的地址是 eth0 的 mac 地址，并且有 arp 包进行询问，可见路由转发正确）

Router1 上对 eth1 进行 wireshark 抓包：

1	0.000000000	Vmware_77:9e:7e	Broadcast	ARP	60	Who has
2	0.000794000	Vmware_6b:19:62	Vmware_77:9e:7e	ARP	60	192.168.
3	0.001042000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
4	0.002981000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
5	1.001726000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi
6	1.002634000	192.168.2.2	192.168.0.2	ICMP	98	Echo (pi
7	2.004769000	192.168.0.2	192.168.2.2	ICMP	98	Echo (pi

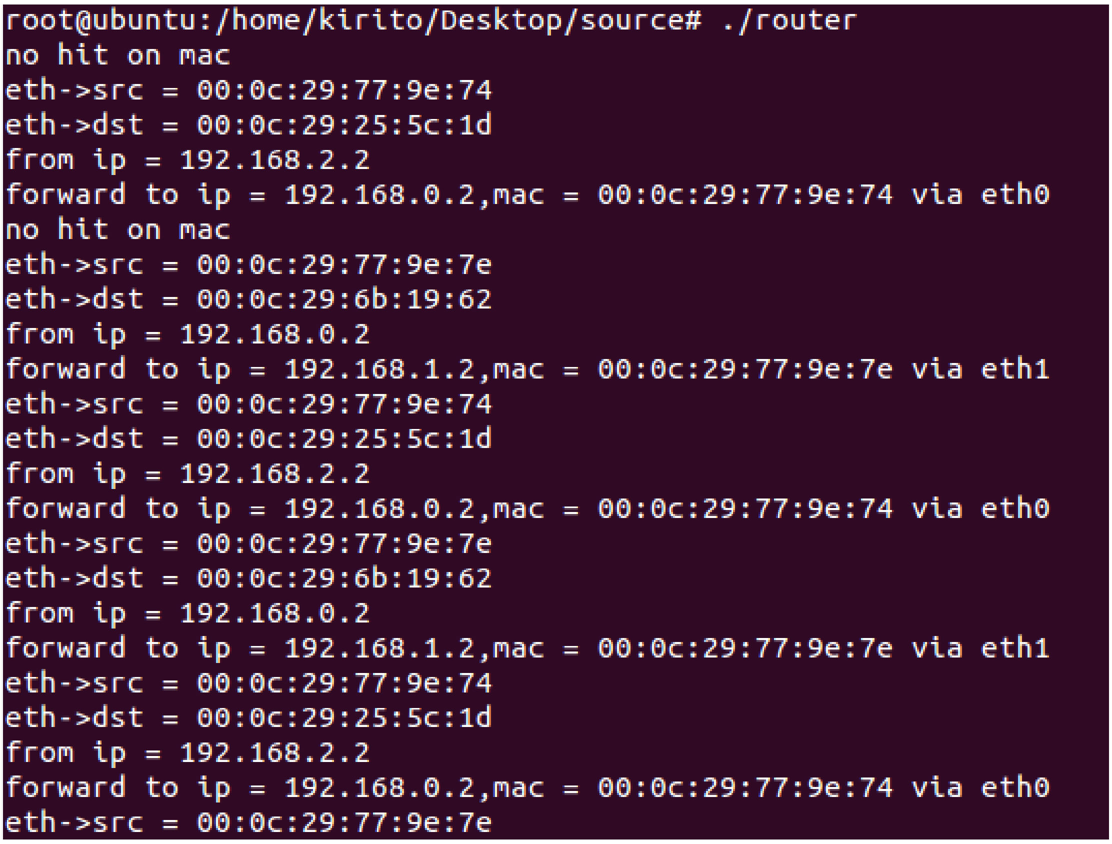
```
▶Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▼Ethernet II, Src: Vmware_77:9e:7e (00:0c:29:77:9e:7e), Dst: Vmware_6b:19:62 (00:0c:29:6b
 ▶Destination: Vmware_6b:19:62 (00:0c:29:6b:19:62)
 ▶Source: Vmware_77:9e:7e (00:0c:29:77:9e:7e)
  Type: IP (0x0800)
▼Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.2.2 (192.168.2
```

（可以看到当关闭路由转发功能，打开程序时，对 eth1 而言，其以太网源地址是 router1 的 eth1 地址，目的地址是 pc2 的 mac 地址，并且首先由 arp 包进行询问 mac 地址，可见路由转发正确）

反向（pc2 ping pc1）

```
kirito@ubuntu:~$ ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=62 time=4.40 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=62 time=2.17 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=62 time=2.34 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=62 time=1.65 ms
64 bytes from 192.168.0.2: icmp_seq=5 ttl=62 time=1.50 ms
64 bytes from 192.168.0.2: icmp_seq=6 ttl=62 time=1.18 ms
64 bytes from 192.168.0.2: icmp_seq=7 ttl=62 time=2.54 ms
64 bytes from 192.168.0.2: icmp_seq=8 ttl=62 time=2.44 ms
64 bytes from 192.168.0.2: icmp_seq=9 ttl=62 time=2.26 ms
^C
--- 192.168.0.2 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8016ms
rtt min/avg/max/mdev = 1.189/2.279/4.406/0.872 ms
kirito@ubuntu:~$
```

程序显示如下：



```
root@ubuntu:/home/kirito/Desktop/source# ./router
no hit on mac
eth->src = 00:0c:29:77:9e:74
eth->dst = 00:0c:29:25:5c:1d
from ip = 192.168.2.2
forward to ip = 192.168.0.2,mac = 00:0c:29:77:9e:74 via eth0
no hit on mac
eth->src = 00:0c:29:77:9e:7e
eth->dst = 00:0c:29:6b:19:62
from ip = 192.168.0.2
forward to ip = 192.168.1.2,mac = 00:0c:29:77:9e:7e via eth1
eth->src = 00:0c:29:77:9e:74
eth->dst = 00:0c:29:25:5c:1d
from ip = 192.168.2.2
forward to ip = 192.168.0.2,mac = 00:0c:29:77:9e:74 via eth0
eth->src = 00:0c:29:77:9e:7e
eth->dst = 00:0c:29:6b:19:62
from ip = 192.168.0.2
forward to ip = 192.168.1.2,mac = 00:0c:29:77:9e:7e via eth1
eth->src = 00:0c:29:77:9e:74
eth->dst = 00:0c:29:25:5c:1d
from ip = 192.168.2.2
forward to ip = 192.168.0.2,mac = 00:0c:29:77:9e:74 via eth0
eth->src = 00:0c:29:77:9e:7e
```

可见反向任然能够 ping 通，可见路由转发没有问题；

6. 相关参考资料：

（百度，google 等）；

7. 对比样例程序：

本次实验无任何样例程序；

8. 代码个人创新及思考：

所有代码均为原创，纯手打，首先该代码很好地模仿了路由功能，其次还可以在不同时候识别不同的网关形式（*）；

思考：

1. 开始时使用 ping 后发现没有显示，用 wireshark 抓包后发现对于 request 和 reply 包的网卡 mac 地址不同，由于没有填写正确的 mac 地址导致 ping 收不到回复包，所以 ping 一直没有显示，后来改成正确的路由地址后正确；
2. 开始时初始化设备时，没有将设备的 ip 和 mac 直接填写到 arp 表中，导致 arp 表项不全，同时无法询问（wireshark 上显示自己问自己，死循环），所以导致转发无法实现。
3. Ping 其实是一个两端互发的过程，所以路由表对于两侧的路由规则都必须了解，否则会只有 request 没有 reply（亲身体会）