Router 实验设计报告

中国科学院大学
页
2022 年 4 月 30 日

一、路由器转发实验

1. 实验内容

（1） 在给定框架的基础上实现具有 IP 地址查找、 IP 数据包转发，处理 ARP 请求和应答、ARP 缓存管理发送和发送 ICMP 报文功能的路由器

（2） 运行给定网络拓扑(router_topo.py)在单路由器网络上完成 ping 测试。

（3） 构造一个包含多个路由器节点组成的网络，在多路由网络上完成 ping 测试和 traceroute 测试

2. 实验流程

1）搭建实验环境：

①在主机上安装 arptables iptables，用于禁止每个节点的相应功能，

```
sudo apt install arptables iptables
```

②安装 traceroute，用于输出路径上节点 ip 信息。

```
sudo apt install traceroute
```

2）在理解已有的代码框架的基础上补充完成以下函数，实现对应的功能：

① arp.c：

（1） arp_send_request：发送 ARP 请求；

（2） arp_send_reply：进行ARP 应答；

（3） handle_arp_packet：处理 ARP 数据包

②arpcache.c：

（1） arpcache_lookup：查找 ARP cache 中是否有需要的 IP 和 mac 地址映射；

（2） arpcache_append_packet：查询 ARP cache 失败时，将包挂起并发送 ARP 请求；

（3） arpcache_insert：插入 IP 到 mac 的映射到 cache，并发送等待该映射的数据包；

（4） arpcache_sweep：老化删除已经不具备时效性的表项，处理未收到应答的包。

③icmp.c：

（1） icmp_send_packet：发送 ICMP 数据包。

④ip.c：

（1） handle_ip_packet：处理 IP 数据包。

⑤ip_base.c：

（1） longest_prefix_match：路由表最长前缀匹配；

（2） ip_send_packet：发送 IP 数据包。

3）make 生成 router，修改并运行给定网络拓扑(router_topo.py)在单路由器网络上完成 ping 测试。给定的网络拓扑如下图 1：

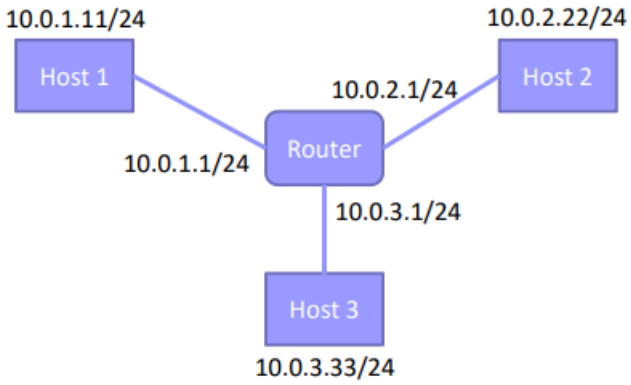

图 1

4）构建包含 2 个 router 、2 个 host 的网络拓扑结构，在多路由网络上完成 ping 测试和 traceroute 测试。构建拓扑结构如下图 2 所示：

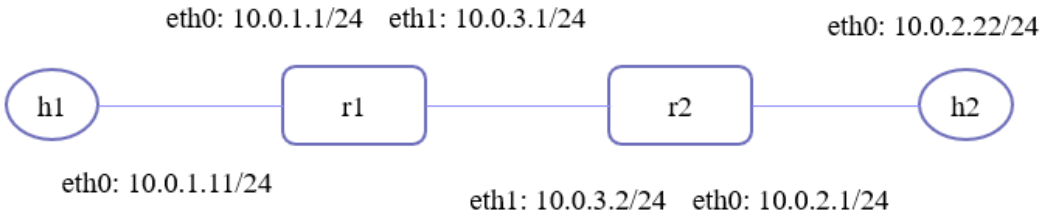

图 2

5）攥写实验报告。

3. 实验设计

路由器读取所有接口的信息后初始化路由表和 arp-cache，然后开始处理从接口文件描符得到的数据包，路由器会读这个数据包的头部，来判断这是一个 IP 数据包还是一个 ARP 数据包，然后调用 handle_ip_packet 函数处理 ip 数据包，调用 handle_arp_packet 函数处理 arp 数据包。handle_ip_packet 函数和 handle_arp_packet 函数以及它们需要调用的函数即为我们需要实现的实验部分，下文按照函数调用出现先后顺序介绍具体函数实现。

1） **handle_ip_packet** 函数（ip,c）

首先提取目的 IP 地址，如果目的 IP 地址等于接口 IP 地址，进一步检查如果报文是 ICMP

echo-request 报文， 则发送 ICMP echo-reply 报文；否则需要转发这个报文，先用最长前缀匹配查找路由表，如果查找成功且生存期没有耗尽，则重新计算校验和并转发数据包，否则调用icmp_send_packet 函数向源主机报错。

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stderr, "TODO: handle ip packet.\n");
    struct iphdr *IP_head = packet_to_ip_hdr(packet);
    u32 dest_ip = ntohl(IP_head->daddr);

    if (dest_ip == iface->ip)
    {
        struct icmphdr *Icmp_head = (struct icmphdr *)(packet +
ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
        if (IP_head->protocol == IPPROTO_ICMP && Icmp_head->type ==
ICMP_ECHOREQUEST)
        {
            icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
        }
        free(packet);
    }
    else
    {
        rt_entry_t *rt = longest_prefix_match(dest_ip);

        if (rt)
        { //路由表查找成功

            if (--IP_head->ttl <= 0)
            {
                icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED,
ICMP_EXC_TTL);
                free(packet);
            }
            else
            {
                IP_head->checksum = ip_checksum(IP_head);
                ip_send_packet(packet, len);
            }
        }
        else
        {
            icmp_send_packet(packet, len, ICMP_DEST_UNREACH,
ICMP_NET_UNREACH);
```

```
        }
    }
}
```

2） **icmp_send_packet 函数**（icmp.c）

主要任务是按照格式填充 ICMP 包。首先读取数据包头部查看具体的 type，若发送的是 reply，则是在回复 ping ，Rest of ICMP Header 拷贝 Ping 包中的相应字段，否则当 type=ICMP_TIME_EXCEEDED 时，说明生存期耗尽；当 type=ICMP_DEST_UNREACH 时，说明查找路由表时失败。然后根据不同的 type 和输入的参数按照 icmp 报文格式配置 icmp 数据包。 当 type 不是 reply 时，Rest of ICMP Header 前 4 字节设置为 0，接着拷贝收到数据包的 IP 头部和随后的 8 字节。

```
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    // fprintf(stderr, "TODO: malloc and send icmp packet.\n");
    struct iphdr *ihr = packet_to_ip_hdr(in_pkt); // in_pkt 的IP 首部
    int pkt_len = 0;

    if (type == ICMP_ECHOREPLY)
        pkt_len = len;
    else
        pkt_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + IP_HDR_SIZE(ihr) +
ICMP_HDR_SIZE + 8;

    char *packet = (char*)malloc(pkt_len * sizeof(char));
    struct ether_header *eh = (struct ether_header*)packet;
    eh->ether_type = htons(ETH_P_IP);

    struct iphdr *packet_IPhead = packet_to_ip_hdr(packet);
    rt_entry_t *rt = longest_prefix_match(ntohl(ihr->saddr));
    ip_init_hdr(packet_IPhead, rt->iface->ip, ntohl(ihr->saddr), pkt_len -
ETHER_HDR_SIZE, 1);
    struct icmphdr *Icmp_head = (struct icmphdr*)(packet + ETHER_HDR_SIZE
+ IP_BASE_HDR_SIZE);
    Icmp_head->type = type;
    Icmp_head->code = code;

    int Rest_begin = ETHER_HDR_SIZE + IP_HDR_SIZE(packet_IPhead) + 4;

    if (type == ICMP_ECHOREPLY)
        memcpy(packet + Rest_begin, in_pkt + Rest_begin, pkt_len -
Rest_begin);
    else {
        memset(packet + Rest_begin, 0, 4);
```

```
        memcpy(packet + Rest_begin + 4, in_pkt + ETHER_HDR_SIZE,
IP_HDR_SIZE(ihr) + 8);
    }

    Icmp_head->checksum = icmp_checksum(Icmp_head, pkt_len -
ETHER_HDR_SIZE - IP_BASE_HDR_SIZE);
    ip_send_packet(packet, pkt_len);
}
```

3） **ip_send_packet 函数**（ip_base.c）

用于发送路由器依据 IP 相关协议生成的报文，如果网关不为 0，该路由器任何端口 IP 都与目的 IP 不在同一网段，向默认网关发包。否则两台主机在同一网段，直接向目的 ip 发地址发包。

```
void ip_send_packet(char *packet, int len)
{
    // fprintf(stderr, "TODO: send ip packet.\n");
    struct iphdr *pkt_IPhead = packet_to_ip_hdr(packet);
    u32 dst_ip = ntohl(pkt_IPhead->daddr);
    rt_entry_t *rt = longest_prefix_match(dst_ip);

    if (rt->gw) // 该路由器任何端口 IP 都与目的 IP 不在同一网段
        iface_send_packet_by_arp(rt->iface, rt->gw, packet, len);
    else
        iface_send_packet_by_arp(rt->iface, dst_ip, packet, len);
}
```

4） **longest_prefix_match 函数**（ip_base.c）

因为子网掩码的缘故，我们匹配路由表在查找时需顺序遍历路由表，返回匹配的网络号最长的表项。

```
rt_entry_t *longest_prefix_match(u32 dst)
{
    // fprintf(stderr, "TODO: longest prefix match for the packet.\n");
    rt_entry_t *rt_entry = NULL, *rt_longest = NULL;
    list_for_each_entry(rt_entry, &rtable, list){
        if((rt_entry->dest & rt_entry->mask) == (dst & rt_entry->mask)) {
            if(!rt_longest || rt_longest->mask < rt_entry->mask)
                rt_longest = rt_entry;
        }
    }
    return rt_longest;
}
```

iface_send_packet_by_arp 函数在arpcache中查找dst_ip的mac地址。 如果找到，填充以太网头，通过iface_send_packet发送此报文，否则，将此报文挂起到arpcache中，发送arp请

求。其中需要使用管理arpcache的函数 arpcache_lookup 和arpcache_append_packet。

5） **arpcache_lookup 函数** （arpcache.c）

遍历表，查找是否有一个具有相同 IP 和 mac 地址的表项，如果查找到了返回 1，否则返回 0。

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
    pthread_mutex_lock(&arpcache.lock);
    for(int i=0;i<MAX_ARP_SIZE;i++){
        if(arpcache.entries[i].valid && arpcache.entries[i].ip4 == ip4){
            memcpy(mac,arpcache.entries[i].mac,ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return 0;
}
```

6） **arpcache_append_packet 函数**（arpcache.c）

当查询 ARP cache 失败时调用将要发送的包挂起，并且发送 ARP 请求。在arp-cache待处理数据包的列表中查找，如果已经有一个具有相同IP地址和iface的条目(这意味着对应的arp请求已经发出)，只需将这个数据包附加到该条目的尾部(该条目可能包含多个数据包);否则，malloc指定IP地址和iface的新表项，附加报文，发送arp请求。

```
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int
len)
{
    // fprintf(stderr, "TODO: append the ip address if lookup failed, and
send arp request if necessary.\n");
    struct arp_req *entry = NULL;
    struct cached_pkt *cached_packet = (struct
cached_pkt*)malloc(sizeof(struct cached_pkt));
    cached_packet->packet = packet;
    cached_packet->len = len;

    int found=0;

    pthread_mutex_lock(&arpcache.lock);

    list_for_each_entry(entry, &(arpcache.req_list), list){
        if (entry->ip4 == ip4 && entry->iface == iface){
            found = 1;
```

```
            break;
        }
    }

    if (found)
        list_add_tail(&(cached_packet->list), &(entry->cached_packets));
    else{
        struct arp_req *req = (struct arp_req*)malloc(sizeof(struct
arp_req));
        req->iface = iface;
        req->ip4 = ip4;
        req->sent = time(NULL);
        req->retries = 1;
        init_list_head(&(req->cached_packets));
        list_add_tail(&(cached_packet->list), &(req->cached_packets));
        list_add_tail(&(req->list), &(arpcache.req_list));
        arp_send_request(iface, ip4);
    }

    pthread_mutex_unlock(&arpcache.lock);
}
```

到此 icmp 数据包的处理完毕，接下来回头看如何利用 handle_arp_packet 函数处理 ip 数据包。

7） **handle_arp_packet 函数**（arp.c）

当目的 IP 地址等于接口 IP 地址，如果报文时 Request 报文，回复自己的 mac 地址给源主机，如果时 Reply 报文说明自己的请求得到回应将信息源主机信息添加到自己的 arp-cache 中。 否则，丢弃这个报文。

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stderr, "TODO: process arp packet: arp request & arp
reply.\n");

    struct ether_arp * ether_arp_pkt = (struct ether_arp *)(packet +
ETHER_HDR_SIZE);


     // 判断目的地址是否为本端口
    if (ntohl(ether_arp_pkt->arp_tpa) == iface->ip){
        // 请求报文得到回应，要保存到ARP 缓存中
        if (ntohs(ether_arp_pkt->arp_op) == ARPOP_REPLY){
```

```
            arpcache_insert(ntohl(ether_arp_pkt->arp_spa),
ether_arp_pkt->arp_sha);
        }
        else if (ntohs(ether_arp_pkt->arp_op) == ARPOP_REQUEST){
            arp_send_reply(iface, ether_arp_pkt);
        }
    }
}
```

8） **arpcache_insert 函数**（arpcache.c）

在 arp-cache 中插入 IP->mac 映射，若找到 ip 项与给定 ip 相同，则更新，否则寻找一个空的项填入，如果缓存已满（32），随机选一个替换出去如果有等待这个映射的数据包，填充每个数据包的以太头，然后发送出去。insert 和 seweep 修改 arp 表时需要加锁互斥访问。

```
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: insert ip->mac entry, and send all the pending
packets.\n");
    pthread_mutex_lock(&arpcache.lock);


    int index = rand() % 32;// 随机生成的0～31 的整数值,没找到无效项则对这一
项覆盖

    for (int i=0; i<MAX_ARP_SIZE; i++){

    // valid==0 说明表项无效，ip 只对应一个 mac

        if (!arpcache.entries[i].valid || arpcache.entries[i].ip4 == ip4){
            index = i;
            break;
        }
    }

    arpcache.entries[index].ip4 = ip4;
    memcpy(arpcache.entries[index].mac, mac, ETH_ALEN);
    arpcache.entries[index].added = time(NULL);
    arpcache.entries[index].valid = 1;

    struct arp_req *entry = NULL;
    struct arp_req *entry_next = NULL;

    list_for_each_entry_safe(entry, entry_next, &(arpcache.req_list),
list) {
        if (entry->ip4 == ip4){
```

```
            struct cached_pkt *pkt = NULL;
            struct cached_pkt *pkt_next;

            list_for_each_entry_safe(pkt, pkt_next,
&(entry->cached_packets), list){
                memcpy(pkt->packet, mac, ETH_ALEN);
                iface_send_packet(entry->iface, pkt->packet, pkt->len);
                free(pkt);
            }

            list_delete_entry(&(entry->list));
            free(entry);
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
}
```

9）**arp_send_request 和 arp_send_reply 函数**（arp.c）

按照 ARP 报文格式填充数据包并发送即可。其中当目的 MAC 地址不可知时，发送 ARP 请求 MAC 地址写 FF: FF: FF: FF: FF: FF 广播请求。

4. 实验结果及分析

（1）单路由器网络上完成 ping 测试

脚本 router_topo.py 增加测试语句如下所示：

```
    net.start()
    r1.cmd('./router &')

    print(h1.cmd('ping -c 2 10.0.1.1'))
    print(h1.cmd('ping -c 2 10.0.2.22'))
    print(h1.cmd('ping -c 2 10.0.3.33'))
    print(h1.cmd('ping -c 2 10.0.3.11'))
    print(h1.cmd('ping -c 2 10.0.4.1'))

    #CLI(net)
    net.stop()
```

得到 h1 ping r1, h2, h3 能够 ping 通，结果如下图 3 所示，图 4 为 h1 ping 10.0.3.11（主机不可达）和 10.0.4.1（网络不可达）的结果，无法 ping 通。

```
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 字节，来自 10.0.1.1: icmp_seq=1 ttl=64 时间=1025 毫秒
64 字节，来自 10.0.1.1: icmp_seq=2 ttl=64 时间=0.142 毫秒

--- 10.0.1.1 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1024 毫秒
rtt min/avg/max/mdev = 0.142/512.332/1024.523/512.190 ms， 管道 2

PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 字节，来自 10.0.2.22: icmp_seq=1 ttl=63 时间=0.118 毫秒
64 字节，来自 10.0.2.22: icmp_seq=2 ttl=63 时间=0.133 毫秒

--- 10.0.2.22 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1027 毫秒
rtt min/avg/max/mdev = 0.118/0.125/0.133/0.007 ms

PING 10.0.3.33 (10.0.3.33) 56(84) bytes of data.
64 字节，来自 10.0.3.33: icmp_seq=1 ttl=63 时间=0.097 毫秒
64 字节，来自 10.0.3.33: icmp_seq=2 ttl=63 时间=0.065 毫秒

--- 10.0.3.33 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1022 毫秒
rtt min/avg/max/mdev = 0.065/0.081/0.097/0.016 ms
```

图 3

```
PING 10.0.3.11 (10.0.3.11) 56(84) bytes of data.
来自 10.0.1.1 icmp_seq=1 目标主机不可达
来自 10.0.1.1 icmp_seq=2 目标主机不可达

--- 10.0.3.11 ping 统计 ---
已发送 2 个包， 已接收 0 个包，+2 错误，100% 包丢失，耗时 1018 毫秒
 管道 2

PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
来自 10.0.1.1 icmp_seq=1 目标网络不可达
来自 10.0.1.1 icmp_seq=2 目标网络不可达

--- 10.0.4.1 ping 统计 ---
已发送 2 个包， 已接收 0 个包，+2 错误，100% 包丢失，耗时 1013 毫秒
```

图 4

（2）在多路由网络上完成 ping 测试和 traceroute 测试

h1 结点 ping r1，r2，h2 的结果和 h1 结点运行 traceroute 的结果如图 5 所示。结果显示连通性正常。另外 traceroute 能够正确输出路径上每个节点的 IP 信息，对应图 2 拓扑中 h1->r1(eth0)->r1(eth1)->r2(eth1)->r2(eth0)->h2 的线路信息，符合预期结果。

```
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 字节，来自 10.0.1.1: icmp_seq=1 ttl=64 时间=1030 毫秒
64 字节，来自 10.0.1.1: icmp_seq=2 ttl=64 时间=0.112 毫秒

--- 10.0.1.1 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1030 毫秒
rtt min/avg/max/mdev = 0.112/514.863/1029.614/514.751 ms， 管道 2

PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 字节，来自 10.0.2.22: icmp_seq=1 ttl=62 时间=0.501 毫秒
64 字节，来自 10.0.2.22: icmp_seq=2 ttl=62 时间=1.03 毫秒

--- 10.0.2.22 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1014 毫秒
rtt min/avg/max/mdev = 0.501/0.763/1.026/0.262 ms

PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
64 字节，来自 10.0.1.11: icmp_seq=1 ttl=62 时间=0.549 毫秒
64 字节，来自 10.0.1.11: icmp_seq=2 ttl=62 时间=0.470 毫秒

--- 10.0.1.11 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% 包丢失，耗时 1016 毫秒
rtt min/avg/max/mdev = 0.470/0.509/0.549/0.039 ms

traceroute to 10.0.2.22 (10.0.2.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.056 ms  0.030 ms  0.023 ms
 2  10.0.3.2 (10.0.3.2)  0.103 ms  0.101 ms  0.099 ms
 3  10.0.2.22 (10.0.2.22)  0.121 ms  0.117 ms  0.111 ms
```

图 5

5. 实验总结

通过本次实验具体实现路由器，切身理解图 6 中 TCP 协议网络层的 IP、ARP 和 ICMP 的具体工作流程和它们的作用，对数据包的具体转发过程有了十分清晰深刻的认识。另外，在自己构建拓扑的时候 ping 失败过，trouceroute 也打出了如图 7 所示奇怪的结果，探索一下发现改变 addLink 的连接顺序会改变路由器的端口（如 eth0,eth1）连接顺序，因此并不是能随意连接的，同时也再次体会到了 python 简洁的同时要默默遵守一些类似语法块时由缩进决定的特别性质。

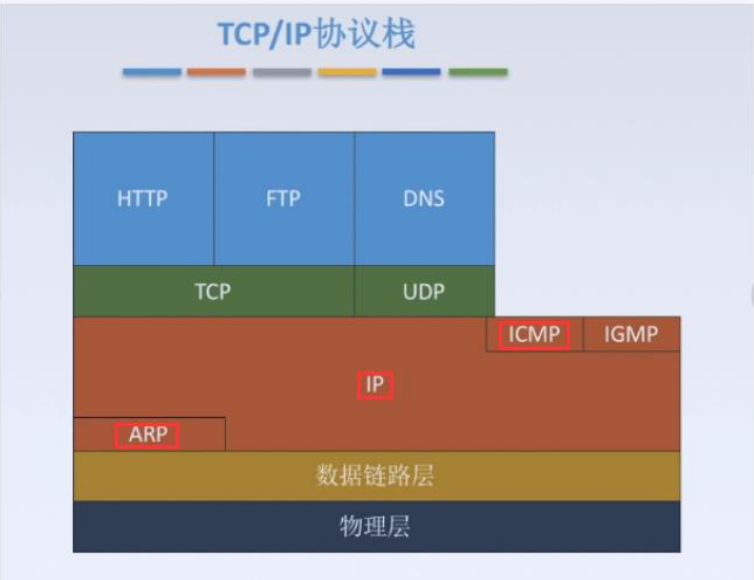

图 6

```
PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
来自 10.0.2.22 icmp_seq=1 目标主机不可达
来自 10.0.2.22 icmp_seq=2 目标主机不可达

--- 10.0.1.11 ping 统计 ---
已发送 2 个包， 已接收 0 个包，+2 错误，100% 包丢失，耗时 1013 毫秒
 管道 2

traceroute to 10.0.2.22 (10.0.2.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.354 ms  0.080 ms  0.026 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * 10.0.1.1 (10.0.1.1)  3960.210 ms !H  3960.171 ms !H
```

图 7

参考资料：

【1】 一个完整的网络数据包转发过程。

【2】 彻底搞懂系列之：ARP 协议。