

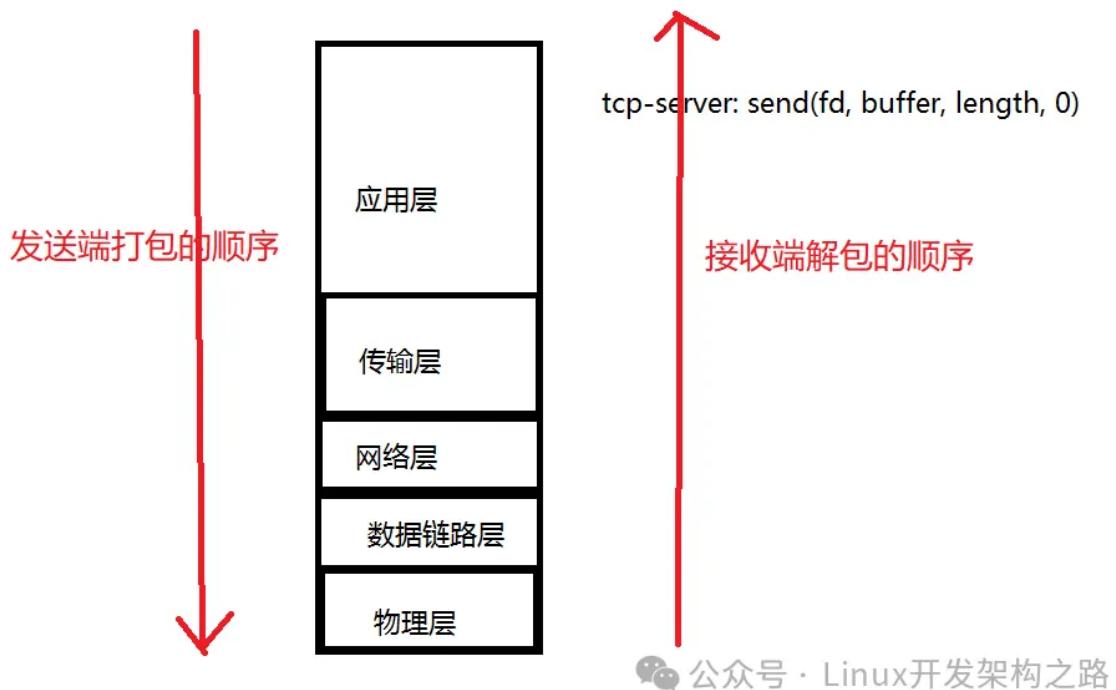
# C++实现一个用户协议栈

## 协议栈

协议栈，指的是TCP/IP协议栈。linux系统中，协议栈是内核实现的。

协议，是通信双方对包格式的一种约定。

为什么是栈呢？因为对于包的组织，类似于栈的数据结构。发送端组织包的顺序是应用层->传输层->网络层->数据链路层，之后通过网卡将数字信号转换成光电信号，发送给接收端；接收端的网卡将光电信号转换成数字信号，解包的顺序是数据链路层->网络层->传输层->应用层。



如何拿到最原始的数据？

raw socket，socket的第二个参数，可以设置SOCK\_STREAM，SOCK\_DGRAM. SOCK\_RAW就可以拿到以太网数据。tcpdump、wireshark就是利用这种方法。

- netmap
- dpdk

# 网卡的作用

---

Client发送数据给server，数据首先到达网卡，经过两步到达应用程序

1. 将数据从网卡的内存copy到内核协议栈，内核协议栈对数据包进行解析
2. 应用程序通过调用recv函数，将数据从内核copy到用户空间，得到应用层的数据包

网卡的作用，接受的时候，是将光电信号转换成数字信号；发送的时候，将数字信号转换成光电信号

## 什么是用户态协议栈

---

就是将协议栈，做到应用程序。为什么要这么做呢？减少了一次数据copy的过程，绕过内核，数据可以直接从网卡copy到应用程序，对于性能会有很大的提升。



公众号 · Linux开发架构之路

## 为什么要有用户态协议栈呢？

是为了解决 C10M 的问题。

之前说过 C10K 的问题，使用epoll可以解决 C10K 的问题。现在epoll已经可以支持两三百万的并发了。

## 什么是 C10M 问题

现10M(即1千万)的并发连接挑战意味着什么：(网上找的)

- 1) 1千万的并发连接数；
- 2) 100万个连接/秒：每个连接以这个速率持续约10秒；

- 3) 10GB/秒的连接：快速连接到互联网；
- 4) 1千万个数据包/秒：据估计目前的服务器每秒处理50K数据包，以后会更多；
- 5) 10微秒的延迟：可扩展服务器也许可以处理这个规模(但延迟可能会飙升)；
- 6) 10微秒的抖动：限制最大延迟；
- 7) 并发10核技术：软件应支持更多核的服务器(通常情况下，软件能轻松扩展到四核，服务器可以扩展到更多核，因此需要重写软件，以支持更多核的服务器).

我们来计算一下，单机承载1000万连接，需要的硬件资源：

内存：1个连接，大概需要4k recvbuffer，4k sendbuffer，一共需要10M  
 $* 8k = 80G$

CPU：10M 除以 50K = 200核

只是支持这么多连接，还没有做其他事情，就需要这么多的资源，如果在加上其他的限制，加上业务的处理，资源肯定会更多。使用用户态协议栈，可以减少一次数据的copy，可以节省很大一部分资源。

要实现用户态协议栈，很关键的一个问题，是网络数据怎么才能绕过内核，直接到达用户空间？netmap、dpdk为用户态协议栈的实现，提供了可能。

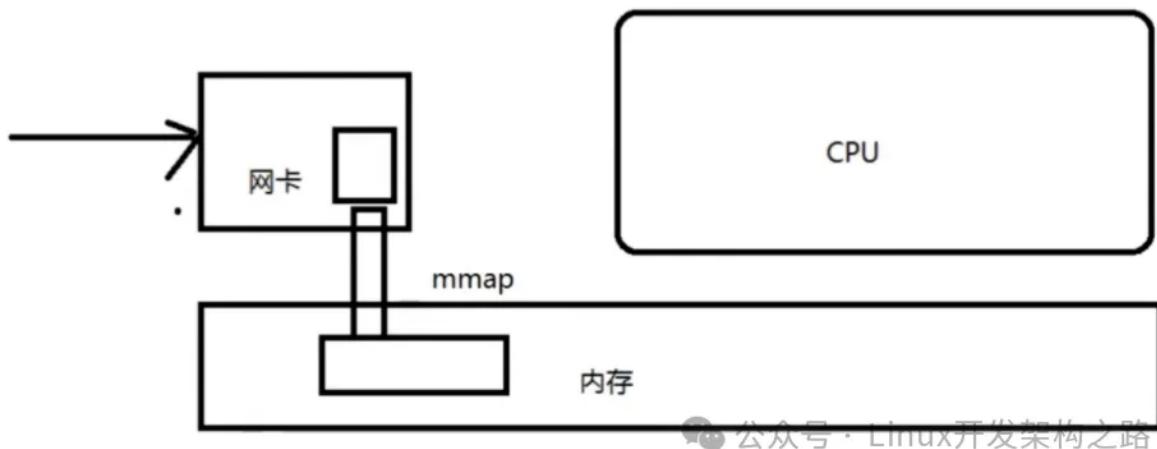
这次我们使用了netmap实现用户态协议栈，后面会介绍dpdk。

## netmap原理

---

netmap主要利用了mmap，将网卡中数据，直接映射到内存。netmap直接接管网卡数据，可以绕过内核协议栈。我们直接在应用程序中实现协议栈，对协议进行解析，就可以获取到网络数据了。

**零拷贝**，使用的是mmap方式，本质是DMA的方式，不需要CPU参与。普通copy，从磁盘copy数据到内存，需要CPU的move指令。sendfile使用的是mmap方式。**零拷贝主要是说CPU有没有参与，而不是说有没有copy。**是由主板上的DMA芯片将外设的数据copy到内存。

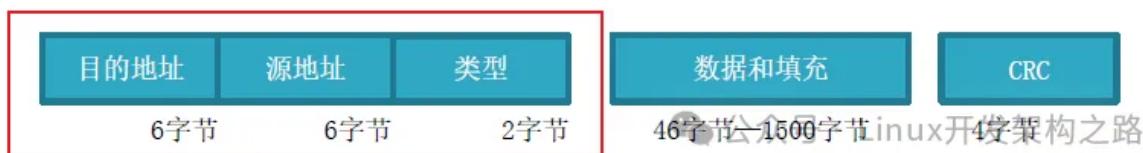


netmap可以在github上下载，按照上面的readme编译安装，使用比较方便。

<https://github.com/luigirizzo/netmap>

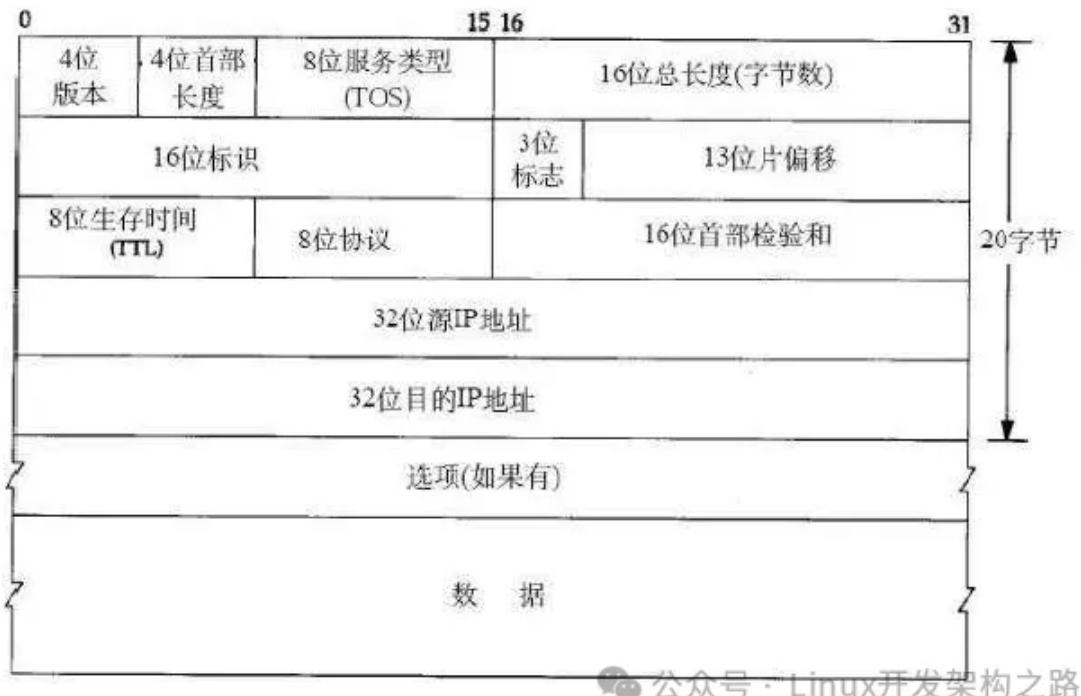
## 利用netmap实现用户态协议栈

### 以太网头定义



```
typedef struct _ethhdr {
    unsigned char h_dst[ETH_ADDR_LENGTH];
    unsigned char h_src[ETH_ADDR_LENGTH];
    unsigned short h_proto;
} ethhdr;
```

### IP头定义



公众号 · Linux开发架构之路

```
typedef struct _iphdr {
    unsigned char hdrlen:4, // ip头长度, 最大15*4=60字节
                  version:4;
    unsigned char tos;
    unsigned short length;
    unsigned short id;
    unsigned short flag_offset;
    unsigned char ttl; // time to live ping的ttl就是ip头里面的ttl
    unsigned char type;
    unsigned short check;
    unsigned int sip;
    unsigned int dip;
} iphdr;
```

## UDP 头定义



公众号 · Linux开发架构之路

UDP比TCP要简单很多，没有序列号，无法保证消息必达；也做不了重传；没有拥塞控制。

```
typedef struct _udphdr {  
  
    unsigned short sport;  
    unsigned short dport;  
    unsigned short length;  
    unsigned short check;  
  
} udphdr;
```

## UDP包定义

UDP包组成：以太网头 + IP头 + UDP头 + UDP数据。

以太网头、IP头、UDP头都已经定义好了，这里有一个问题，UDP数据怎么定义？

用指针是不合适的，这里引入了**零长数组**，也叫柔性数组。

```
typedef struct _udppkt {  
  
    ethhdr eh; // 14  
    iphdr ip; // 20  
    udphdr udp; // 8  
  
    unsigned char data[0];  
  
} udppkt;
```

柔性数组的好处，是不占空间，只是占了一个位置，作用相当于标签。

### 零长数组使用条件：

1. 不关心长度，可以通过某种方法计算出它的长度，比如通过udp头的length能够计算出来用户数据的长度；
2. 它的内存是提前分配好的，不会越界。

因为4字节对齐的关系，`sizeof(udppkt)`的结果是44，所以需要使用单字节对齐，结果就是42。

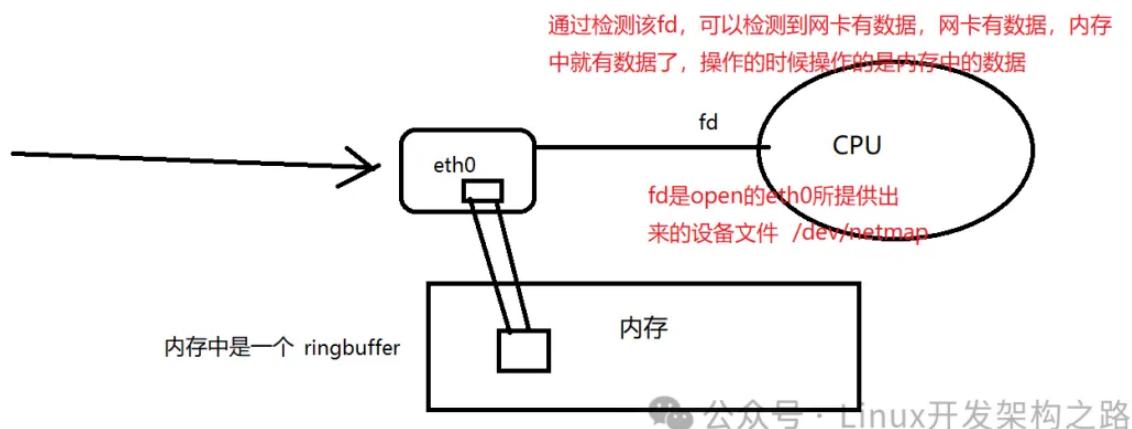
```
#pragma pack(1)
```

## 使用netmap处理UDP包

nm\_open主要做了两个事情：

1. 把网卡的内存映射到内存；
2. 把fd指向eth0对应的设备文件；

通过检测fd，就可以判断网卡是否有数据，有数据就可以直接操作内存。



如果来了一个数据，如果数据很多，网卡需要进行模拟信号和数字信号转换，DMA也需要不断把数据映射到内存，怎么把

多个数据包给组织起来？比如一下来100个包，怎么把这100个包组织起来？使用ringbuff。

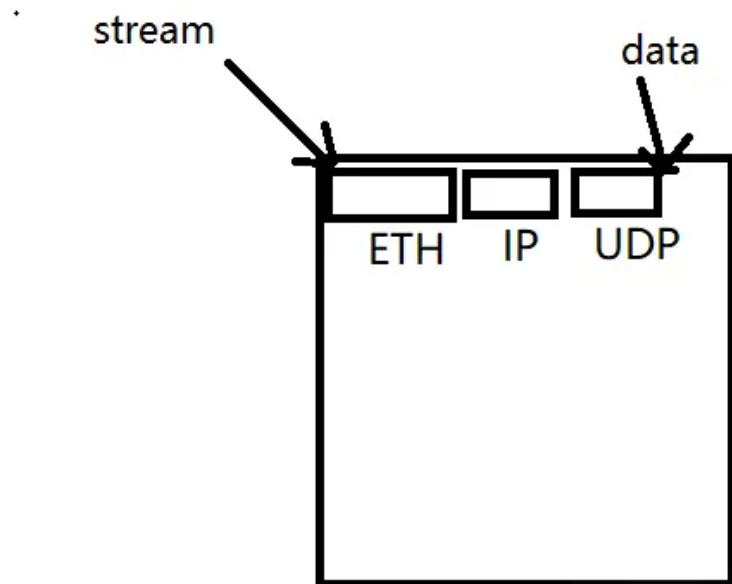
对于大量数据，从网卡将数据取到内存中，CPU有两种做法：

1. 轮询
2. 事件

对于大量数据，使用轮询方式比较好。这就是网络这一层，从网卡里面取数据的两种方法。事件的方式针对稀疏型的数据。

nm\_nextpkt是操作内存，取出来的是一个完整的包。

之后通过去掉以太网头，IP头，UDP头，得到用户数据。



公众号 · Linux开发架构之路

```

int main() {
    struct nm_pkthdr h; // ringbuff的指针
    struct nm_desc *nmr = nm_open("netmap:eth1", NULL, 0,
NULL);

    if (nmr == NULL) return -1;

    struct pollfd pfd = {0};
    pfd.fd = nmr->fd;
    pfd.events = POLLIN;

    while (1) {

        int ret = poll(&pfd, 1, -1);
        if (ret < 0) continue;

        if (pfd.events & POLLIN) {

            unsigned char *stream = nm_nextpkt(nmr, &h);
// ringbuff

            ethhdr *eh = (ethhdr*)stream;
            if (ntohs(eh->h_proto) == PROTO_IP) {

                udppkt *udp = (udppkt*)stream;

```

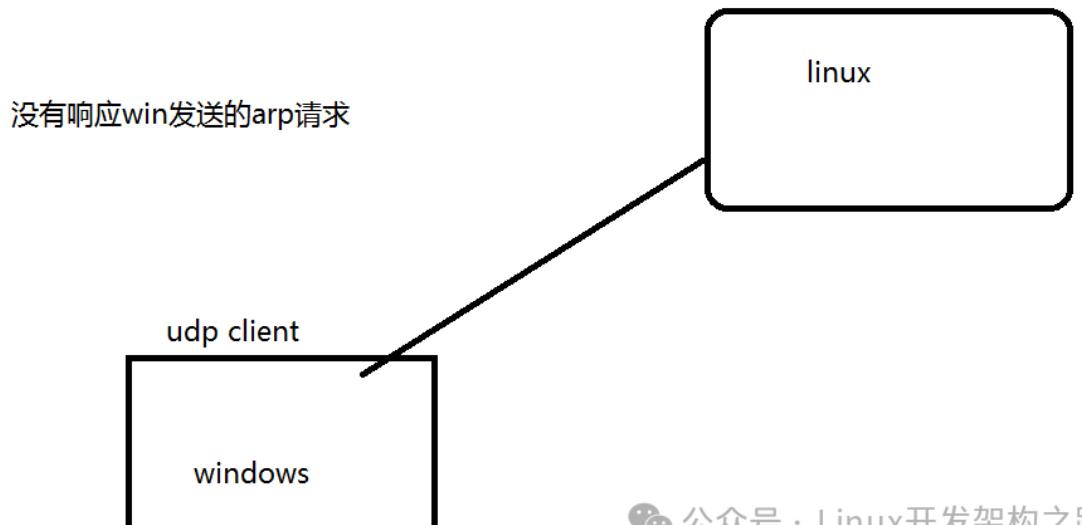
```
    if (udp->ip.type == PROTO_UDP) {  
  
        int udplen = ntohs(udp->udp.length);  
  
        udp->data[udplen - 8] = '\0';  
  
        printf("udp --> %s\n", udp->data);  
  
    }  
  
}  
  
}  
  
}
```

程序编译运行后，是可以成功打印udp数据的。

运行的时候，注意要使用两块网卡，不然eth0的网卡被我们的程序接管了，ssh就无法登陆了。

但是会有一个问题，程序运行一段时间后，无法处理udp包了。

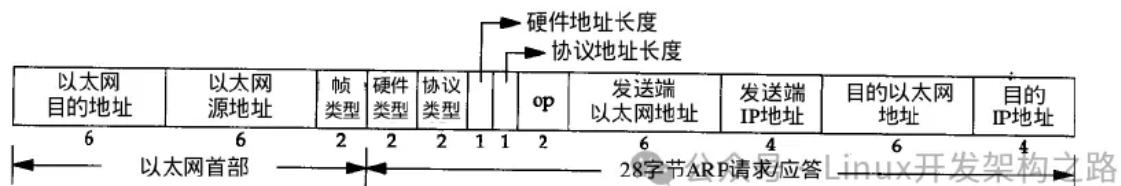
原因是机器会维护一个arp表，就是ip和mac地址的对应关系表，arp表会老化，过期了之后，如果要向目标机器发送消息，会重新发送arp请求，由于我们的程序接管了机器的网卡，但是却没有处理arp请求，导致对端的arp表中没有我们这台机器的数据。



公众号 · Linux开发架构之路

## arp协议实现

### arp头定义



```

typedef struct _aprhdr {
    unsigned short h_type;
    unsigned short protocol;
    unsigned char h_addr_len;
    unsigned char p_addr_len;
    unsigned short oper;
    unsigned char smac[ETH_ADDR_LENGTH];
    unsigned int sip;
    unsigned char dmac[ETH_ADDR_LENGTH];
    unsigned int dip;
} arphdr;

```

# arp包定义

```
typedef struct _arppkt {  
  
    ethhdr eh;  
    arphdr arp;  
  
} arppkt;
```

## 处理arp包

解析arp包，将源ip、源mac和目的ip、目的mac互换，并且将oper设置为2(arp 请求是1，回复是2)，就可以了。

```
int main() {  
    struct nm_pkthdr h; // ringbuff的指针  
    struct nm_desc *nmr = nm_open("netmap:eth1", NULL, 0,  
    NULL);  
  
    if (nmr == NULL) return -1;  
  
    struct pollfd pfd = {0};  
    pfd.fd = nmr->fd;  
    pfd.events = POLLIN;  
  
    while (1) {  
  
        int ret = poll(&pfd, 1, -1);  
        if (ret < 0) continue;  
  
        if (pfd.events & POLLIN) {  
  
            unsigned char *stream = nm_nextpkt(nmr, &h);  
            // ringbuff  
  
            ethhdr *eh = (ethhdr*)stream;  
            if (ntohs(eh->h_proto) == PROTO_IP) {  
  
                udppkt *udp = (udppkt*)stream;
```

```

        if (udp->ip.type == PROTO_UDP) {

            int udpLen = ntohs(udp->udp.length);

            udp->data[udpLen - 8] = '\0';

            printf("udp --> %s\n", udp->data);

        }

    } else if (ntohs(eh->h_proto) == PROTO_ARP) {

        arppkt *arp = (arppkt *)stream;

        arppkt arp_rt;

        if (arp->arp.dip ==
inet_addr("10.36.121.51")) { // 去掉if判断，就可以做arp攻击

            echo_arp_pkt(arp, &arp_rt,
"fa:16:3e:72:4c:ca");

            nm_inject(nmr, &arp_rt,
sizeof(arp_rt));

            printf("arp ret\n");

        }

    }

}

```

## arp攻击

客户端发送请求的时候，如果发现目的地址不在arp表当中，会自动发送一个arp请求。

arp请求是向局域网内所有的机器发送请求，要获取目标ip对应的机器，也就是mac地址。如果对于任何arp请求，你会进行回复，就会造成arp欺骗，也就是arp攻击。

在上面的代码中，如果不对arp包中的目的ip进行判断，就可以实现arp攻击。

```
if (arp->arp.dip == inet_addr("10.36.121.51")) { // 去掉if  
判断，就可以做arp攻击
```

## 完整代码

```
#include <stdio.h>  
  
#include <sys/poll.h>  
#include <arpa/inet.h>  
  
#define NETMAP_WITH_LIBS  
  
#include <net/netmap_user.h>  
  
#pragma pack(1) // 单字节对齐  
  
#define ETH_ADDR_LENGTH 6  
  
#define PROTO_IP 0x0800  
  
#define PROTO_ARP 0x0806  
  
#define PROTO_UDP 17  
#define PROTO_ICMP 1  
  
  
typedef struct _ethhdr {  
    unsigned char h_dst[ETH_ADDR_LENGTH];  
    unsigned char h_src[ETH_ADDR_LENGTH];  
    unsigned short h_proto;
```

```
    } ethhdr;

typedef struct _iphdr {
    unsigned char hdrlen:4, // ip头长度, 最大15*4=60字节
                version:4;
    unsigned char tos;
    unsigned short length;
    unsigned short id;
    unsigned short flag_offset;//这里貌似有问题, 应该使用位域
    unsigned char ttl; // time to live ping的ttl就是ip头里面的ttl
    unsigned char type;
    unsigned short check;
    unsigned int sip;
    unsigned int dip;
} iphdr;

typedef struct _aprhdr {
    unsigned short h_type;
    unsigned short protocol;
    unsigned char h_addr_len;
    unsigned char p_addr_len;
    unsigned short oper;
    unsigned char smac[ETH_ADDR_LENGTH];
    unsigned int sip;
    unsigned char dmac[ETH_ADDR_LENGTH];
    unsigned int dip;
} arphdr;

typedef struct _udphdr {
    unsigned short sport;
```

```
    unsigned short dport;
    unsigned short length;
    unsigned short check;
}

} udphdr;

typedef struct _icmphdr {

    unsigned char type;
    unsigned char code;
    unsigned short check;
    unsigned short id;
    unsigned short seq;
    unsigned char mask[32];

} icmphdr;

typedef struct _udppkt {

    ethhdr eh; // 14
    iphdr ip; // 20
    udphdr udp; // 8

    unsigned char data[0];

} udppkt;

typedef struct _arppkt {

    ethhdr eh;
    arphdr arp;

}

} arppkt;

typedef struct _icmppkt {
```

```
    ethhdr eh;
    iphdr ip;
    icmpphdr icmp;

} icmpppkt;

int str2mac(char *mac, char *str) {

    char *p = str;
    unsigned char value = 0x0;
    int i = 0;

    while (p != '\0') {

        if (*p == ':') {
            mac[i++] = value;
            value = 0x0;
        } else {

            unsigned char temp = *p;
            if (temp <= '9' && temp >= '0') {
                temp -= '0';
            } else if (temp <= 'f' && temp >= 'a') {
                temp -= 'a';
                temp += 10;
            } else if (temp <= 'F' && temp >= 'A') {
                temp -= 'A';
                temp += 10;
            } else {
                break;
            }
            value <<= 4;
            value |= temp;
        }
        p++;
    }

    mac[i] = value;

    return 0;
}
```

```
}

void echo_arp_pkt(arppkt *arp, arppkt *arp_rt, char *mac)
{
    memcpy(arp_rt, arp, sizeof(arppkt));

    memcpy(arp_rt->eh.h_dst, arp->eh.h_src,
ETH_ADDR_LENGTH);
    str2mac(arp_rt->eh.h_src, mac);
    arp_rt->eh.h_proto = arp->eh.h_proto;

    arp_rt->arp.h_addr_len = 6;
    arp_rt->arp.p_addr_len = 4;
    arp_rt->arp.oper = htons(2);

    str2mac(arp_rt->arp.smac, mac);
    arp_rt->arp.sip = arp->arp.dip;

    memcpy(arp_rt->arp.dmac, arp->arp.smac,
ETH_ADDR_LENGTH);
    arp_rt->arp.dip = arp->arp.sip;

}

unsigned short in_cksum(unsigned short *addr, int len)
{
    register int nleft = len;
    register unsigned short *w = addr;
    register int sum = 0;
    unsigned short answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
```

```
        *(u_char *)&answer) = *(u_char *)w ;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;

    return (answer);

}

void echo_icmp_pkt(icmppkt *icmp, icmppkt *icmp_rt) {

    memcpy(icmp_rt, icmp, sizeof(icmppkt));

    memcpy(icmp_rt->eh.h_dst, icmp->eh.h_src,
ETH_ADDR_LENGTH);
    memcpy(icmp_rt->eh.h_src, icmp_rt->eh.h_dst,
ETH_ADDR_LENGTH);
    icmp_rt->eh.h_proto = icmp->eh.h_proto;

    icmp_rt->icmp.type = 0x0; //
    icmp_rt->icmp.code = 0x0; //
    icmp_rt->icmp.check = 0x0;

    icmp_rt->ip.sip = icmp->ip.dip;
    icmp_rt->ip.dip = icmp->ip.sip;

    icmp_rt->icmp.check = in_cksum((unsigned
short*)&icmp_rt->icmp, sizeof(icmphdr));
}

// insmod netmap.ko
// gcc -o net_family_o net_family_0.c
int main() {
    struct nm_pkthdr h; // ringbuff的指针
    struct nm_desc *nmr = nm_open("netmap:eth1", NULL, 0,
NULL);
```

```
if (nmr == NULL) return -1;

struct pollfd pfd = {0};
pfd.fd = nmr->fd;
pfd.events = POLLIN;

while (1) {

    int ret = poll(&pfd, 1, -1);
    if (ret < 0) continue;

    if (pfd.events & POLLIN) {

        unsigned char *stream = nm_nextpkt(nmr, &h);
// ringbuff

        ethhdr *eh = (ethhdr*)stream;
        if (ntohs(eh->h_proto) == PROTO_IP) {

            udppkt *udp = (udppkt*)stream;

            if (udp->ip.type == PROTO_UDP) {

                int udplen = ntohs(udp->udp.length);

                udp->data[udplen - 8] = '\0';

                printf("udp --> %s\n", udp->data);

            } else if (udp->ip.type == PROTO_ICMP) {

                icmppkt *icmp = (icmppkt*)stream;

                icmpkt icmp_rt;

                echo_icmp_pkt(icmp, &icmp_rt);

                nm_inject(nmr, &icmp_rt,
sizeof(icmp_rt));

```

```
        }

    } else if (ntohs(eh->h_proto) == PROTO_ARP) {

        arppkt *arp = (arppkt *)stream;

        arppkt arp_rt;

        if (arp->arp.dip ==
inet_addr("10.36.121.51")) { // 去掉if判断，就可以做arp攻击

            echo_arp_pkt(arp, &arp_rt,
"fa:16:3e:72:4c:ca");

            nm_inject(nmr, &arp_rt,
sizeof(arp_rt));

            printf("arp ret\n");

        }

    }

}

}

}
```

## ARP协议详解 ----- 一看就懂

### 什么是ARP?

ARP协议是“Address Resolution Protocol”（地址解析协议）的缩写。

# ARP的作用

在以太网环境中，数据的传输所依赖的是MAC地址而非IP地址，而将已知IP地址转换为MAC地址的工作是由ARP协议来完成的。

在局域网中，网络中实际传输的是“帧”，帧里面有目标主机的MAC地址的。在以太网中，一个主机和另一个主机进行直接通信，必须要知道目标主机的MAC地址。但这个目标MAC地址是如何获得的呢？它就是通过地址解析协议获得的。

所谓“地址解析”就是主机在发送帧前将目标IP地址转换成目标MAC地址的过程。ARP协议的基本功能就是通过目标设备的IP地址，查询目标设备的MAC地址，以保证通信的顺利进行。

## ARP工作流程

假设主机A和B在同一个网段，主机A要向主机B发送信息，具体的地址解析过程如下：

(1) 主机A首先查看自己的ARP表，确定其中是否包含有主机B对应的ARP表项。如果找到了对应的MAC地址，则主机A直接利用ARP表中的MAC地址，对IP数据包进行帧封装，并将数据包发送给主机B。

(2) 如果主机A在ARP表中找不到对应的MAC地址，则将缓存该数据报文，然后以广播方式发送一个ARP请求报文。

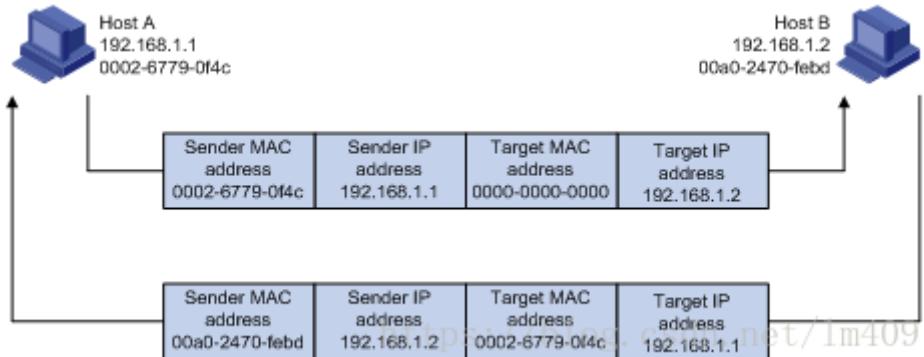
ARP请求报文中的发送端IP地址和发送端MAC地址为主机A的IP地址和MAC地址，目标IP地址和目标MAC地址为主机B的IP地址和全0的MAC地址。

由于ARP请求报文以广播方式发送，该网段上的所有主机都可以接收到该请求，但只有被请求的主机（即主机B）会对该请求进行处理。

(3) 主机B比较自己IP地址和ARP请求报文中的目标IP地址，当两者相同时进行如下处理：将ARP请求报文中的发送端（即主机A）的IP地址和MAC地址存入自己的ARP表中。

之后以单播方式发送ARP响应报文给主机A，其中包含了自身的MAC地址。

(4) 主机A收到ARP响应报文后，将主机B的MAC地址加入到自己的ARP表中以用于后续报文的转发，同时将IP数据包进行封装后发送出去。



## 抓包分析

### 圈中一对ARP请求和ARP应答

23624 182.743495	Shenzhen_0c:8d:62	IntelCor_27:54:e3	ARP	42 192.168.1.1 is at 8c:f2:28:0c:8d:62
24570 194.768930	HuaweiTe_32:76:3f	Broadcast	ARP	42 Who has 192.168.1.1? Tell 192.168.1.106
24584 194.958545	IntelCor_27:54:e3	Broadcast	ARP	42 Who has 192.168.1.1? Tell 192.168.1.101
24585 194.968182	Shenzhen_0c:8d:62	IntelCor_27:54:e3	ARP	42 192.168.1.1 is at 8c:f2:28:0c:8d:62
24904 200.241679	IntelCor_27:54:e3	Shenzhen_0c:8d:62	ARP	42 Who has 192.168.1.1? Tell 192.168.1.101
24905 200.243998	Shenzhen_0c:8d:62	IntelCor_27:54:e3	ARP	42 192.168.1.1 is at 8c:f2:28:0c:8d:62
24953 200.960923	IntelCor_27:54:e3	Broadcast	ARP	42 Who has 192.168.1.1? Tell 192.168.1.101
24954 200.968793	Shenzhen_0c:8d:62	IntelCor_27:54:e3	ARP	42 192.168.1.1 is at 8c:f2:28:0c:8d:62
28861 215.556062	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.106? Tell 192.168.1.1
29316 216.580107	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.106? Tell 192.168.1.1
29387 217.604196	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.106? Tell 192.168.1.1
29584 219.549536	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.116? Tell 192.168.1.1
30473 224.772077	HuaweiTe_32:76:3f	Broadcast	ARP	42 Who has 192.168.1.1? Tell 192.168.1.106
30503 225.591226	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.106? Tell 192.168.1.1
31006 231.530282	Shenzhen_0c:8d:62	Broadcast	ARP	42 Who has 192.168.1.116? Tell 192.168.1.1

## ARP请求报文

```

Ethernet II, Src: IntelCor_27:54:e3 (94:65:9c:27:54:e3), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  Destination: Broadcast (ff:ff:ff:ff:ff:ff)      为获得某个IP地址的MAC地址, 先进行广播
  Source: IntelCor_27:54:e3 (94:65:9c:27:54:e3)
  Type: ARP (0x0806)

Address Resolution Protocol (request)
  Hardware type: Ethernet (1)
  Protocol type: IPv4 (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)                         发送端
  Sender MAC address: IntelCor_27:54:e3 (94:65:9c:27:54:e3)
  Sender IP address: 192.168.1.101
  Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00) ← 广播时全0, 未填充
  Target IP address: 192.168.1.1               目标端
                                                因为此时还不知道目的MAC地址

```

## ARP应答报文

```

Ethernet II, Src: Shenzhen_0c:8d:62 (8c:f2:28:0c:8d:62), Dst: IntelCor_27:54:e3 (94:65:9c:27:54:e3)
  Destination: IntelCor_27:54:e3 (94:65:9c:27:54:e3) ← ARP请求中的源地址变为ARP应答中的目的地址
  Source: Shenzhen_0c:8d:62 (8c:f2:28:0c:8d:62)
  Type: ARP (0x0806)

Address Resolution Protocol (reply)
  Hardware type: Ethernet (1)
  Protocol type: IPv4 (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: reply (2)                           回答时这两个字段相同
  Sender MAC address: Shenzhen_0c:8d:62 (8c:f2:28:0c:8d:62)
  Sender IP address: 192.168.1.1
  Target MAC address: IntelCor_27:54:e3 (94:65:9c:27:54:e3)
  Target IP address: 192.168.1.101

```

# ARP表

设备通过ARP解析到目的MAC地址后，将会在自己的ARP表中增加IP地址到MAC地址的映射表项，以用于后续到同一目的地报文的转发。

C:\Users\Administrator>arp -a		
接口:	Internet 地址	物理地址
219.223.164.23 --- 0xe	219.223.164.1	08-81-f4-96-29-07
	219.223.164.6	08-81-f4-96-29-07
	219.223.167.255	ff-ff-ff-ff-ff-ff
	224.0.0.22	01-00-5e-00-00-16
	224.0.0.252	01-00-5e-00-00-fc
	255.255.255.255	ff-ff-ff-ff-ff-ff

## 动态ARP表

动态ARP表项由ARP协议通过ARP报文自动生成和维护，可以被老化，可以被新的ARP报文更新，可以被静态ARP表项覆盖。当到达老化时间、接口down时会删除相应的动态ARP表项。

## 静态ARP表

静态ARP表项通过手工配置和维护，不会被老化，不会被动态ARP表项覆盖。

配置静态ARP表项可以增加通信的安全性。静态ARP表项可以限制和指定IP地址的设备通信时只使用指定的MAC地址，此时攻击报文无法修改此表项的IP地址和MAC地址的映射关系，从而保护了本设备和指定设备间的正常通信。

## 免费ARP

免费ARP指主机发送ARP查找自己的IP地址，通常发生在系统引导期间进行接口配置时。与标准ARP的区别就是免费ARP分组的目的IP地址字段封装的是自己的IP地址，即向所在网络请求自己的MAC地址。

## 免费ARP的作用有：

1. 一个主机可以通过它来确定另一个主机是否设置了相同的 IP地址。

正常情况下发送免费ARP请求不会收到ARP应答，如果收到了一个ARP应答，则说明网络中存在与本机相同的IP地址的主机，发生了地址冲突。

## 2. 更新其他主机高速缓存中旧的硬件地址信息。

如果发送免费ARP的主机正好改变了硬件地址，如更换了接口卡。

其他主机接收到这个ARP请求的时候，发现自己的ARP高速缓存表中存在对应的IP地址，但是MAC地址不匹配，那么就需要利用接收的ARP请求来更新本地的ARP高速缓存表项。

## 3. 网关利用免费ARP防止ARP攻击

有些网关设备在一定的时间间隔内向网络主动发送免费ARP报文，让网络内的其他主机更新ARP表项中的网关MAC地址信息，以达到防止或缓解ARP攻击的效果。

## 4. 利用免费ARP进行ARP攻击

ARP协议并不只在发送了ARP请求才接收ARP应答，计算机只要接收到ARP应答数据包，就会使用应答中的IP和MAC地址对本地的ARP缓存进行更新。

主机可以构造虚假的免费ARP应答，将ARP的源MAC地址设为错误的MAC地址，并把这个虚假的免费ARP应答发送到网络中，那么所有接收到这个免费ARP应答的主机都会更新本地ARP表中相应IP地址对应的MAC地址。更新成功后，这些主机的数据报文就会被转发到错误的MAC地址，从而实现了ARP欺骗的攻击。

# 代理ARP

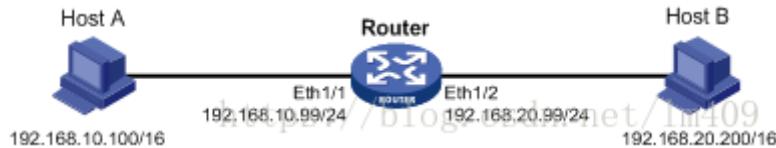
代理ARP就是通过使用一个主机(通常为router)，来作为指定的设备使用自己的 MAC 地址来对另一设备的ARP请求作出应答。

## 为什么需要代理ARP？

先要了解，路由器的重要功能之一就是把局域网的广播包限制在该网内，阻止其扩散，否则会造成网络风暴。

ARP请求是个广播包，它询问的对象如果在同一个局域网内，就会收到应答。但是如果询问的对象不在同一个局域网该如何处理？路由器就提供了代理ARP为这个问题提供了解决方案。

# 工作过程



两台主机A和B处于同一网段但不同的广播段（不在同一物理网络上）时，主机A发送ARP请求主机B的MAC地址时，因为路由器不转发广播包的原因，ARP请求只能到达路由器。如果路由器启用了代理ARP功能，并知道主机B属于它连接的网络，那么路由器就用自己接口的MAC地址代替主机B的MAC地址来对主机A进行ARP应答。主机A接收ARP应答，但并不知道代理ARP的存在。

## 代理ARP的优缺点

### 优点：

代理ARP能在不影响路由表的情况下添加一个新的Router，使子网对该主机变得透明化。一般代理ARP应该使用在主机没有配置默认网关或没有任何路由策略的网络上。

### 缺点：

从工作工程可以看到，这其实是一种ARP欺骗。而且，通过两个物理网络之间的路由器的代理ARP功能其实互相隐藏了物理网络，这导致无法对网络拓扑进行网络概括。此外，代理ARP增加了使用它的那段网络的ARP流量，主机需要更大的ARP缓存空间，也不会为不使用ARP进行地址解析的网络工作。

## ARP攻击

ARP协议的基本功能就是通过目标设备的IP地址，查询目标设备的MAC地址，以保证通信的进行。

基于ARP协议的这一工作特性，黑客向对方计算机不断发送有欺诈性质的ARP数据包，数据包内包含有与当前设备重复的Mac地址，使对方在回应报文时，由于简单的地址重复错误而导致不能进行正常的网络通信。

一般情况下，受到ARP攻击的计算机会出现两种现象：

1. 不断弹出“本机的XXX段硬件地址与网络中的XXX段地址冲突”的对话框。

## 2. 计算机不能正常上网，出现网络中断的症状。

因为这种攻击是利用ARP请求报文进行“欺骗”的，所以防火墙会误以为是正常的请求数据包，不予拦截。因此普通的防火墙很难抵挡这种攻击。

# C语言结构体位域及其存储

## 前言

在一些应用中，比如网络协议，经常会涉及对数据的某些比特位进行操作，尽管可以使用位的相关运算，但是C语言提供了位域用以支持对一个字节的某几个位进行访问，操作起来也更加方便。本文关注于说明C语言中位域的使用及其在内存中的排列规则，尤其在大小端平台下位域存储的差异。

## 位域的定义与引用

位域不同于一般的结构体成员，它以位为单位来定义成员的长度，因此在结构体中定义位域时，必须要指明位域成员所需要占用的二进制位数。一个简单的定义位域的示例如下：

```
struct Foo {
    int a: 5;      // 数据类型名    变量名: 二进制位数
    int b: 3;
    int c: 2;
};
```

由于位域本质上是一种特殊的结构体成员，因此一般结构体成员的引用方法同样适用于位域成员。不过，需要特别注意的是，位域成员存储是以二进制位作为单位的，而内存的最小寻址单元是字节，所以不能直接引用位域成员的地址。

## 匿名位域

在使用位域时，如果有需要可以选择跳过某些位不使用，其方法是在结构体类型中定义位域成员时，只指定成员占用的二进制位数，而不定义成员名。由于被跳过的这些位域成员没有名字，因此在程序中也无法进行引用。

```

struct Foo {
    int a: 3;
    int b: 4;
    int : 5;      // 定义匿名位域，选择性跳过部分位不使用
    int c: 3;
};

```

特别的，我们可以根据需要定义二进制位数为0的匿名位域成员，目的是指定某些位域从一个新的内存单元开始存放。

## 位域的存储

结构体中位域在内存中的存储遵循以下规则：

若相邻的位域成员的类型相同，且其占用二进制位数未超过该类型可容纳的范围，则后面的成员紧接着前一个成员进行存储；

若相邻的位域成员的类型相同，但其占用二进制位数超过该类型可容纳的范围，则后面的成员从该类型占用空间之后的内存单元开始存储；

若相邻的位域成员的类型不同，则取决于编译器的实现。对于GCC编译器会尽量利用空闲的位对数据进行存储；

若位域之间定义有匿名位域成员，则匿名位域成员指定的空闲位不用于后续成员的数据存储；

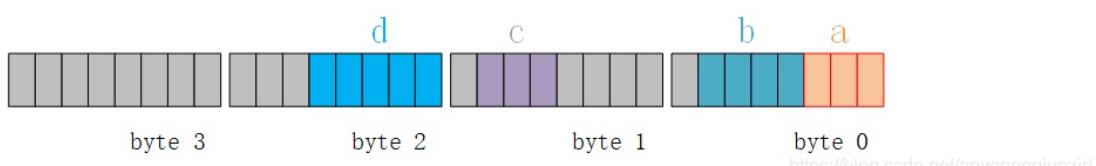
## 说明示例1

考虑如下结构体，该结构体内的定义几乎使用到了上面提到的所有规则：

```

struct Foo {
    unsigned short a: 3;
    unsigned short b: 4;
    unsigned short : 5;      // 定义无名位域，选择性跳过部分位不使用
    unsigned short c: 3;
    unsigned short d: 5; // 前一个字节剩余空间不足以存放下一个成员，跳过剩余的位
};

```



- 对于成员a、b、c再加上匿名成员占用的总内存空间并未超过**unsigned short**类型空间的大小，因此在**unsigned short**类型可容纳的范围内，这些成员可以紧挨着存放；

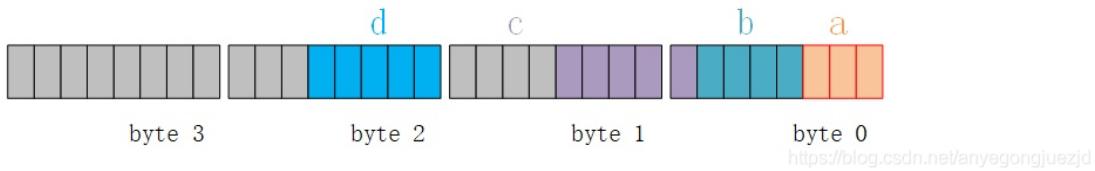
- 当后续存放成员d时，前一个unsigned short类型数据剩余的空间已不足以容纳d，因此选择下一个内存单元进行存放。

特别地，如果定义上述结构体中匿名成员占用的位数为0，那么对于第一个unsigned short类型数据在除被a、b占用区域的其它剩余空间都将不会被使用，则成员c需要从byte2开始进行存储。

## 说明示例2

现在考虑相邻位域成员类型不同的情况，定义如下结构体并画出其在内存中的布局如下（这个结果基于gcc编译器）：

```
struct Foo {  
    unsigned char a: 3;  
    unsigned char b: 4;  
    unsigned short c: 5;  
    unsigned short d: 5;  
};
```



GCC编译器会尽可能地利用空闲的内存位进行位域成员的存放，这里有一点需要注意的是，尽管对于第一个unsigned char类型可容纳的单字节范围在存放完成员a和b后，剩余的一位已不足以存放成员c，但是GCC编译器仍然将这一位分配给了c。

## 位域与大小端

系统的大小端差异会同时牵涉到字节序和比特序问题，对于结构体位域这种会涉及比特位层面数据的操作，几乎需要时刻考虑平台大小端的差异。结构体内位域成员在大小端系统上的内存分配规则如下：

- 无论是大端或小端模式，位域的存储都是由内存低地址向高地址分配，即从低地址字节的低位bit开始向高地址字节的高位bit分配空间；
- 位域成员在已分配的内存区域内，按照机器定义的比特序对数据的各个bit位进行排列。即在小端模式中，位域成员的最低有效位存放在内存低bit位，最高有效位存放在内存高bit位；大端模式则相反。

## 程序示例

为了说明上述的规则，参考如下代码：

```
struct Foo2 {
    unsigned short a: 1;
    unsigned short b: 3;
    unsigned short c: 4;
    unsigned short d: 4;
    unsigned short e: 4;
};

void test_foo2(void)
{
    union {
        struct Foo2 foo;
        unsigned short s_data;
    }val;

    val.foo.a = 1;
    val.foo.b = 3;
    val.foo.c = 5;
    val.foo.d = 7;
    val.foo.e = 15;

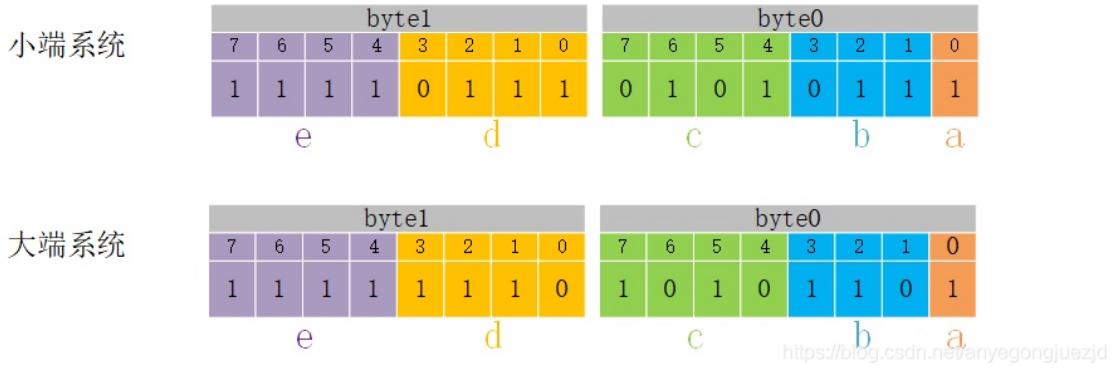
    printf("val is 0x%x.\n", val.s_data);

    return;
}
```

运行上述程序：

- 在小端设备上的输出的结果为：**0xf757**；
- 换成大端设备，程序的运行结果为：**0xb57f**。

我们画出程序所定义结构体位域成员在内存中的存储布局如下所示（因为平时多数都是在使用小端机器，在此有意将字节的顺序和字节内的位顺序进行颠倒，方便对比）：



为了更便于理解位域在内存中的排列规则，建议将位域成员内存空间的分配和解析分开来看：

1. 第一步先考虑内存空间的分配，从上图中可以看到，不论大小端都是从内存地址的低位开始；
2. 当位域成员占用空间确定之后，考虑于位域成员数据位的排布，可以看到小端系统从低bit位开始存放数据，这是符合我们预期的，而大端设备则恰恰相反，大端系统从高bit位开始存放数据，因此在大端设备中，我们需要转换下思维从内存的高位开始解析数据位。

## C/C++位域详解整理

### 一、定义：

有些信息在存储时，并不需要占用一个完整的字节，而只需要占一个或几个二进制位。例如在存放一个开关量时，只有0和1两种状态，只需要用一位二进制位即可。为了节省存储空间，并且为了让处理变得更便捷，C语言又提供了一种数据结构，称为“位域”或者“位段”。所谓位域，就是把一个字节中的二进制位划分为不同的区域，并说明每个区域的位数。每个域都有一个域名，允许程序中按照域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

### 二、结构：

位域的结构与结构体相似

```
struct <位域结构名>
{
    ...
    <类型说明符> <位域名> : <位域长度> // 位域列表
    ...
};
```

ex:

```
struct Demo
{
    int a : 8;
    int b : 2;
    int c : 4;
};
```

而位域变量的说明也与结构体变量的说明方式相同，可以采用先定义后说明、同时定义说明、直接说明这三种方式，例如：

方式1：

```
struct Demo
{
    int a : 8;
    int b : 2;
    int c : 4;
}demo1;
```

方式2：

```
struct Demo
{
    int a : 8;
    int b : 2;
    int c : 4;
};
struct Demo demo1;
```

方式3：

```
typedef struct Demo
{
    int a : 8;
```

```
int b : 2;
int c : 4;
}De;
De demo1;
```

这三种方式中的demo1均为Demo的变量，且占两个字节，其中位域a占8bit，位域b占2bit，位域c占4bit。

### 三、几点注意事项：

1. 一个位域必须存储在同一个字节中，不能跨字节存储。如一个字节所剩空间不能存储下一个位域的时候，应从下一个字节开始存储。也可以有意使某个位域从下一单元开始，如：

```
struct Demo
{
    int a : 4;
    int : 0; //空域
    int b : 6; //从第二个字节开始存放
};
```

在这个位域定义中，a占第一个字节的4bit，这个字节的另4bit填0表示不使用，b从第二个字节开始，占6bit。

2. **由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说位域的不能超过8bit。**
3. 位域可以无位域名，这时它只用作填充或调整位置。无名的位域是不能使用的。例如：

```
struct Demo
{
    int a : 4;
    int : 2; // 这2bit不能使用
    int b : 2;
};
```

总结：从以上三点来看，位域的本质就是一种结构类型，不过其成员是按二进制位分配的。

## 四、位域的使用：

位域的使用与结构体相同，其一般形式为：<位域变量名>.<位域名>，位域允许各种格式的输出。

```
struct Demo
{
    unsigned int a : 1;
    unsigned int b : 4;
    unsigned int c : 3;
}demo1,*pdemo;

void main()
{
    demo1.a = 1;
    demo1.b = 15;
    demo1.c = 7;
    printf("demo1.a = %d, demo1.b = %d, demo1.c =
%d\n",demo1.a,demo1.b,demo1.c);

    pdemo = &demo1;
    pdemo->a = 0;
    pdemo->b |= 1;
    pdemo->c |= 3;
    printf("pdemo->a = %d, pdemo->b = %d, pdemo->c =
%d\n",pdemo->a,pdemo->b,pdemo->c);
}
```

上例程序中定义了位域结构Demo，其中有三个位域a,b,c。说明了该位域结构的两个变量demo1和指向Demo类型的指针变量pdemo，这表示位域也是可以使用指针的。

同时也给三个位域进行了赋值（注意赋值不能超过该位域的允许范围）。

上例也表示了位域也可以进行复合的位运算。

## 五、含位域结构体的sizeof

位域结构的成员不能单独被取sizeof值

C99规定int、unsigned int、bool可以做位域的类型，但编译器几乎都对此做了扩展，允许其他类型的存在。

使用位域的主要目的在于压缩内存，大致规则如下：

1. 如果相邻位域字段的类型相同，且其位宽之和小于类型的sizeof大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；
2. 如果相邻位域字段的类型相同，且其位宽之和大于类型的sizeof大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；
3. 如果相邻位域字段的类型不同，则各个编译器的具体实现有所差异，VC6不压缩，而Dev C++压缩；
4. 如果位域字段之间插着非位域字段，则不进行压缩；
5. 整个位域结构体的总体大小为最宽的基本类型成员大小的整数倍。

例一：

```
struct Demo
{
    char c1 : 3;
    char c2 : 4;
    char c3 : 5;
};

sizeof(Demo) = 2;
```

位域类型为char，第一个字节仅仅能存储下c1和c2，所以c3只能存储到第二个字节中去。

例二：

```
struct Demo
{
    char c1 : 3;
    short s2 : 4;
    char c3 : 5;
};
```

在VC中： sizeof(Demo) = 6。

因为字节对齐，所以c1要对齐s2，即c1要占用2个Byte，s2要占用2个Byte，而c3占用1个Byte。

又因为整个结构大小最宽的类型为short，所以要是short的整数倍，所以还需要填充1个Byte，所以sizeof(Demo) = 6。

在Dev C++中， sizeof(Demo) = 2；

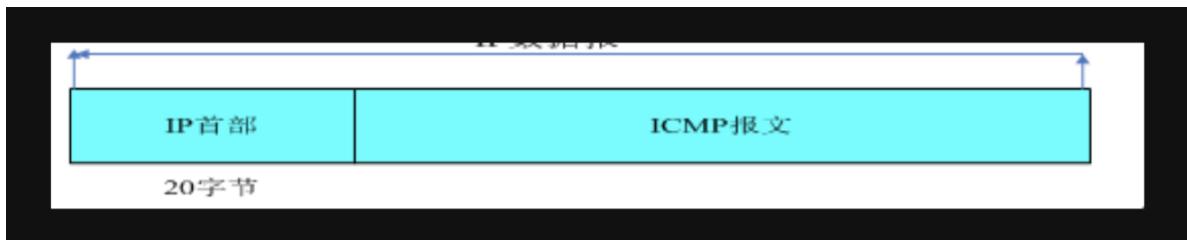
例三：

```
struct Demo
{
    char c1 : 3;
    char c2;
    char c3 : 5;
};

sizeof(Demo) = 3;
```

c1占1个bit, c2占1个bit, c3占一个bit。

## 深入理解ICMP协议



### 概述：

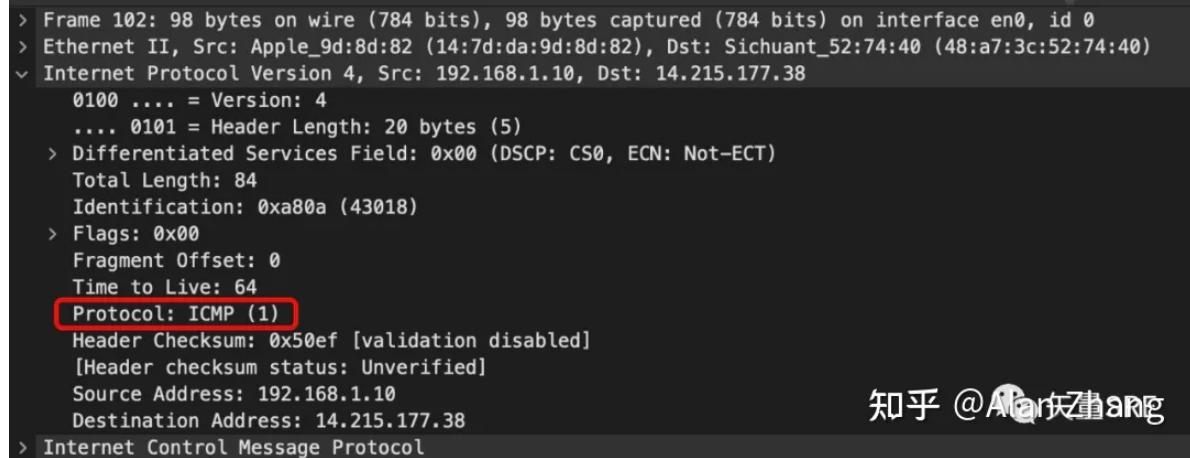
1. ICMP允许主机或路由报告差错情况和提供有关异常情况。ICMP是因特网的标准协议，但ICMP不是高层协议，而是IP层的协议。通常ICMP报文被IP层或更高层协议（TCP或UDP）使用。一些ICMP报文把差错报文返回给用户进程。
2. ICMP报文作为IP层数据报的数据，加上数据报的首部，组成数据报发送出去。
3. ICMP报文的种类有两种，即ICMP差错报告报文和ICMP询问报文。

## ICMP协议

ICMP是 Internet Control Message Protocol 的缩写，即互联网控制消息协议。它是互联网协议族的核心协议之一。它用于 TCP/IP 网络中发送控制消息，提供可能发生在通信环境中的各种问题反馈，通过这些信息，使网络管理者可以对所发生的问题作出诊断，然后采取适当的措施解决问题。

虽然 ICMP 是网络层协议，但是它不像 IP 协议和 ARP 协议一样直接传递给数据链路层，而是先封装成 IP 数据包然后再传递给数据链路层。所以在 IP 数据包中如果协议类型字段的值是 1 的话，就表示 IP 数据是 ICMP 报文。IP 数据包就是靠这个协议类型字段来区分不同的数据包的。如下图

WireShark 抓包所示：



Frame 102: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface en0, id 0  
> Ethernet II, Src: Apple\_9d:8d:82 (14:7d:da:9d:8d:82), Dst: Sichuan\_52:74:40 (48:a7:3c:52:74:40)  
Internet Protocol Version 4, Src: 192.168.1.10, Dst: 14.215.177.38  
0100 .... = Version: 4  
.... 0101 = Header Length: 20 bytes (5)  
> Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)  
Total Length: 84  
Identification: 0xa80a (43018)  
> Flags: 0x00  
Fragment Offset: 0  
Time to Live: 64  
Protocol: ICMP (1) (1)  
Header Checksum: 0x50ef [validation disabled]  
[Header checksum status: Unverified]  
Source Address: 192.168.1.10  
Destination Address: 14.215.177.38  
> Internet Control Message Protocol

知乎 @AlanZhang

在 IP 通信中如果某个包因为未知原因没有到达目的地址，那么这个具体的原因就是由 ICMP 负责告知。而 ICMP 协议的类型定义中就清楚的描述了各种报文的含义。

## ICMP协议类型

ICMP协议的类型分为两大类，**查询报文**和**差错报文**。如下表：

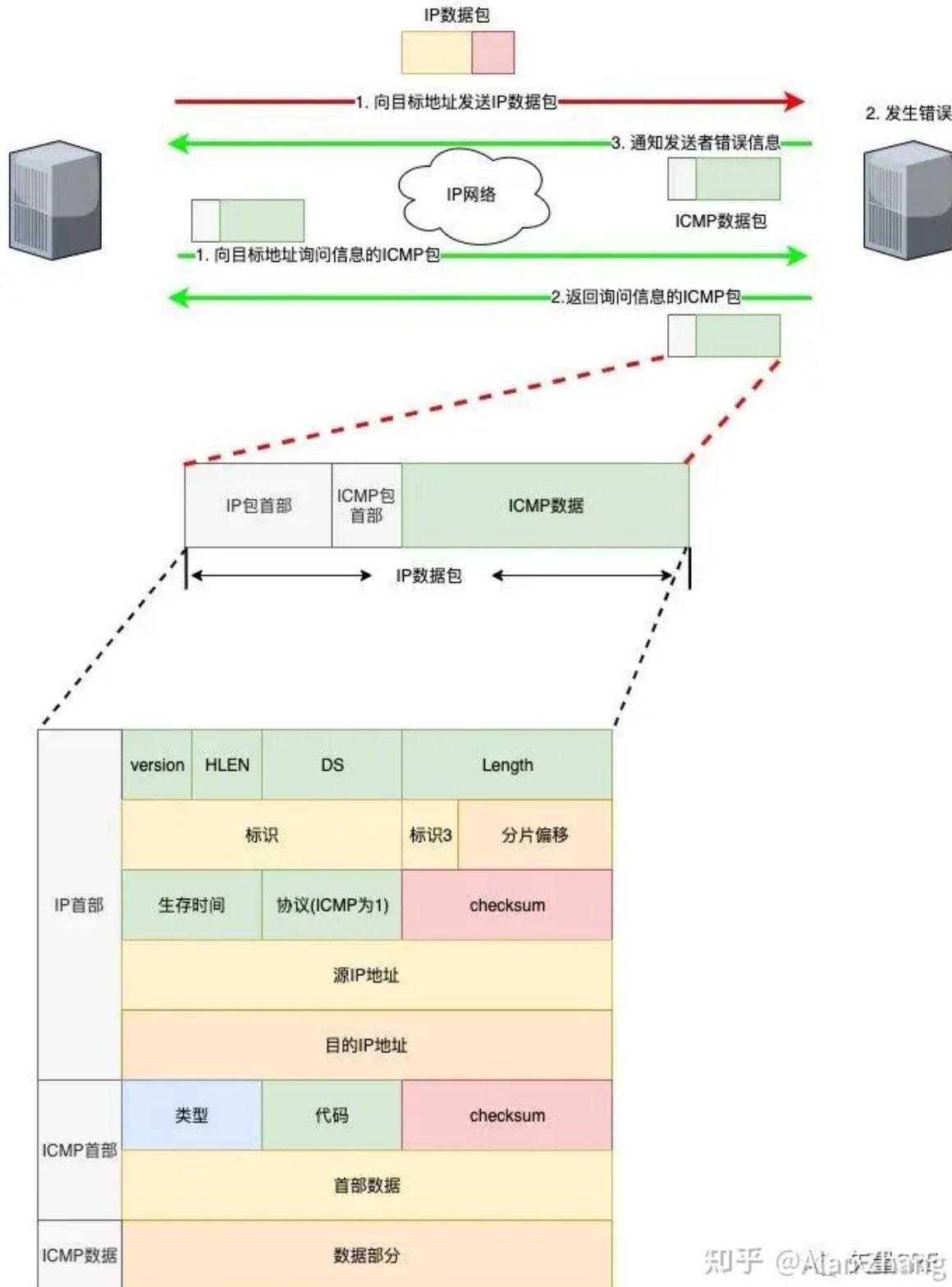
种类	类型	代码	报文含义
查询报文	0	0	回显应答
查询报文	8	0	回显请求
差错报文	3	0	网络不可达
差错报文	3	1	主机不可达
差错报文	3	2	协议不可达
差错报文	3	3	端口不可达
差错报文	3	4	需要进行分片但设置不分片比特
差错报文	3	5	源站选路失败
差错报文	3	6	目的网络未知
差错报文	3	7	目的主机未知
差错报文	3	8	源主机被隔离
差错报文	3	9	目的网络被强制禁止
差错报文	3	10	目的主机被强制禁止
差错报文	3	11	由于服务类型TOS, 网络不可达
差错报文	3	12	由于服务类型TOS, 主机不可达
差错报文	3	13	由于过滤, 通信被强制禁止
差错报文	3	14	知乎 @Alan_Zhang 手机越权

差错报文	3	15	优先中止生效
差错报文	4	0	源端被关闭
差错报文	5	0	对网络重定向
差错报文	5	1	对主机重定向
差错报文	5	2	对服务类型和网络重定向
差错报文	5	3	对服务类型和主机重定向
差错报文	9	0	路由器通告
差错报文	10	0	路由器请求
差错报文	11	0	传输期间生存时间为0
差错报文	11	1	在数据报组装期间生存时间为0
差错报文	12	0	坏的IP首部
差错报文	12	1	知乎 @Alan_Zhang 缺少必需的选项

是不是看起来有点多啊？计算机网络真的是博大精深啊。而平时我们经常使用的少之又少。

# ICMP报文格式

从 ICMP 的报文格式来说，ICMP 是 IP 的上层协议。但是 ICMP 是分担了 IP 的一部分功能。所以，他也被认为是与 IP 同层的协议。下面一起来看一下 ICMP 数据包格式和报文内容吧。具体参考下图：

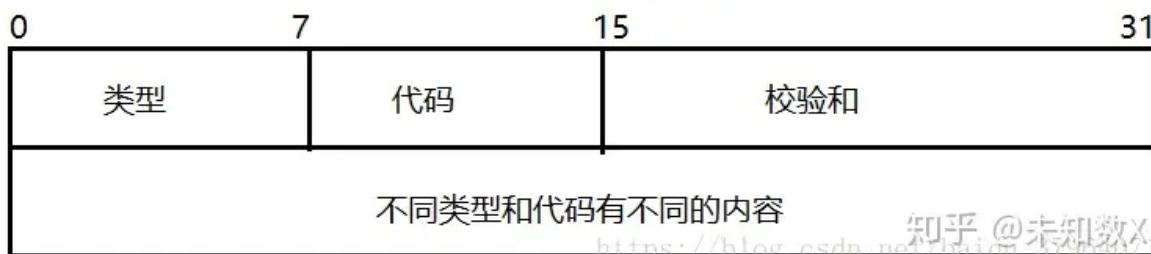


可以看到，我们一层层剥开 ICMP 的外壳，查看其本质。方便我们更好的理解 ICMP 协议。

# ICMP的报文格式

ICMP报文包含在IP数据报中，IP报头在ICMP报文的最前面。一个ICMP报文包括IP报头（至少20字节）、ICMP报头（至少八字节）和ICMP报文（属于ICMP报文的数据部分）。当IP报头中的协议字段值为1时，就说明这是一个ICMP报文。ICMP报头如下图所示。

如下图：



## 二.ICMP报文的格式



1. 类型：占8位
2. 代码：占8位
3. 检验和：占16位

说明：ICMP所有报文的前4个字节都是一样的，但是剩下的其他字节则互不相同。

4. 其它字段都ICMP报文类型不同而不同。

1> ICMP报文的前4个字节是统一的格式，共有三个字段：即类型，代码和检验和。

2> 8位类型和8位代码字段一起决定了ICMP报文的类型。

类型8，代码0：表示回显请求(ping请求)。

类型0，代码0：表示回显应答(ping应答)

类型11，代码0：超时

3>16位的检验和字段：包括数据在内的整个ICMP数据包的检验和；其计算方法和IP头部检验和的计算方法一样的。

ICMP报文具体分为查询报文和差错报文(对ICMP差错报文有时需要做特殊处理，因此要对其进行区分。如：对ICMP差错报文进行响应时，永远不会生成另一份ICMP差错报文，否则会出现死循环)

# ICMP协议实现--Ping命令

通过上面的叙述，我们初步了解了 ICMP 协议的内容，那么下面我们来看一下ICMP 的具体实现与应用吧，首先我们来了解**查询报文**的应用--ping，下面我们将通过 ping 同一个子网的主机，来看一下整个过程是怎么样的。

那么比较完整的流程是：

## 1. 向目的服务器发送回显请求

首先，向目的服务器上执行ping命令，主机会构建一个 ICMP 回显请求消息数据包（类型是8，代码是0），在这个回显请求数据包中，除了类型和代码字段，还被追加了标识符和序号字段。标识符和序号字段分别是16位的字段。ping命令在发送回显请求数据包时，会将进程号填写在标识符里。对于序号，每送出一个数据包数值就增加1。而且，回显请求的选项数据部分用来装任意数据。这个任意数据用来调整ping的交互数据包的大小。如下图在192.168.1.10上执行ping 192.168.1.1的命令：

ping命令执行的时候，他的进程号是43991：

```
AlandeMacBook-Pro:~ alan$ ps -ef | grep ping
501 43991 43640 0 11:34下午 ttys000 0:00.00 ping 192.168.1.1
501 43993 43950 0 11:35下午 ttys001 0:00.01 grep ping
AlandeMacBook-Pro:~ alan$
```

知乎 @Alan夜笙歌

通过WireShark抓包可以看出，上图中的192.168.1.10主机会构建一个ICMP回显请求消息数据包。

No.	Time	Source	Destination	Protocol	Length	Info
3	0.286331	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=11/2816, ttl=64 (reply in 4)
4	0.295416	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=11/2816, ttl=64 (request in 3)
45	1.291745	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=12/3072, ttl=64 (reply in 46)
46	1.294671	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=12/3072, ttl=64 (request in 45)
47	2.293409	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=13/3328, ttl=64 (reply in 49)
49	2.303695	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=13/3328, ttl=64 (request in 47)
50	3.297810	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=14/3584, ttl=64 (reply in 51)
51	3.307123	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=14/3584, ttl=64 (request in 50)
62	4.300205	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=15/3840, ttl=64 (reply in 64)
64	4.306910	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=15/3840, ttl=64 (request in 62)
65	5.305554	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=16/4096, ttl=64 (reply in 66)
66	5.314873	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=16/4096, ttl=64 (request in 65)
67	6.309316	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=17/4352, ttl=64 (reply in 68)
68	6.319583	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=17/4352, ttl=64 (request in 67)
69	7.311921	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=18/4608, ttl=64 (reply in 70)
70	7.321809	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=18/4608, ttl=64 (request in 69)

知乎 @Alan夜笙歌

上图中我们可以看到ICMP的Type(协议类型)是8，Code(代码)是0，Identifier(ping进程号)是43991，同时还有Sequence Number发送序号11，以及发送时间。

## 2. 目的服务器发送回显应答

当192.168.1.10送到回显请求数据包后，192.168.1.1就会向发送方192.168.1.10发送回显应答（类型是0，代码是0），这个ICMP回显应答数据包在IP层来看，与被送来的回显请求数据包基本上一样。不同的只有源、目标IP地址字段被交换了，Type类型字段里填入了表示回显应答的0。通过WireShark抓包可以看出，如下图：

No.	Time	Source	Destination	Protocol	Length	Info
3	0.286331	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=11/2816, ttl=64 (reply in 4)
4	0.295416	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=11/2816, ttl=64 (request in 3)
45	1.291745	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=12/3872, ttl=64 (reply in 46)
46	1.294671	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=12/3872, ttl=64 (request in 45)
47	2.293489	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=13/3328, ttl=64 (reply in 49)
49	2.303695	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=13/3328, ttl=64 (request in 47)
50	3.297810	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=14/3584, ttl=64 (reply in 51)
51	3.307123	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=14/3584, ttl=64 (request in 50)
62	4.300285	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=15/3840, ttl=64 (reply in 64)
64	4.306918	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=15/3840, ttl=64 (request in 62)
65	5.305554	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=16/4096, ttl=64 (reply in 66)
66	5.314873	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=16/4096, ttl=64 (request in 65)
67	6.309316	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=17/4352, ttl=64 (reply in 68)
68	6.319583	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=17/4352, ttl=64 (request in 67)
69	7.311921	192.168.1.10	192.168.1.1	ICMP	98	Echo (ping) request id=0xd7ab, seq=18/4608, ttl=64 (reply in 70)
70	7.321889	192.168.1.1	192.168.1.10	ICMP	98	Echo (ping) reply id=0xd7ab, seq=18/4608, ttl=64 (request in 69)

> Frame 4: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface en0, id 0  
 > Ethernet II, Src: Sichuant\_52:74:40 (48:a7:3c:52:74:40), Dst: Apple\_9d:8d:82 (14:7:da:9d:8d:82)  
 > Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.1.10  
 > Internet Control Message Protocol  
 Type: 0 (Echo (ping) reply)  
 Code: 0  
 Checksum: 0x6f55 [correct]  
 [Checksum Status: Good]  
 Identifier (BE): 55211 (0xd7ab)  
 Identifier (LE): 43991 (0xabd7)  
 Sequence Number (BE): 11 (0x000b)  
 Sequence Number (LE): 2816 (0xb00b)  
 [Request frame: 3]  
 [Response time: 9.005 ms]  
 Timestamp from icmp data: May 1, 2021 23:35:08.456752000 CST  
 [Timestamp from icmp data (relative): 0.009260000 seconds]  
 > Data (48 bytes)

知乎 @Alandzhang

### 3. 源服务器显示相关数据

如果源服务器可以接收到回显应答数据包，那我们就认为192.168.1.1是正常工作着的。进一步，记住发送回显请求数据包的时间，与接收到回显应答数据包的时间差，就能计算出数据包一去一回所需要的时间。这个时候ping命令就会将目的服务器的IP地址，数据大小，往返花费的时间打印到屏幕上。如下图：

```
AlandeMacBook-Pro:~ alan$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=10.978 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=10.285 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=9.654 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=10.133 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=2.904 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=10.501 ms
```

知乎 @Alandzhang

## ICMP协议实现--traceroute命令

traceroute命令是一款充分利用ICMP差错报文类型的应用，其主要用作追踪路由信息，下面我们将逐个查看其工作原理。

我们通过执行 traceroute 192.168.1.1，来分析一下他的原理，它的原理就是利用 IP 包的 TTL 从 1 开始按照顺序递增的同时发送 UDP 包，强制接收 ICMP 超时消息的方法。

首先 traceroute 会将 IP 包的 TTL 设置为 1，然后发送 UDP 包，他会填入一个端口号作为 UDP 目标端口号（默认是：33434-33534）。如下图：

No.	Time	Source	Destination	Protocol	Length	Info
17	3.619160	192.168.1.10	192.168.1.1	UDP	66	48596 → 33435 Len=24
18	3.630642	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)
19	3.634969	192.168.1.10	192.168.1.1	UDP	66	48596 → 33436 Len=24
20	3.638519	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)
21	3.638972	192.168.1.10	192.168.1.1	UDP	66	48596 → 33437 Len=24
22	3.641892	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)

```
> Frame 17: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface en0, id 0
> Ethernet II, Src: Apple_9d:8d:82 (14:7d:da:9d:8d:82), Dst: Sichuant_52:74:40 (48:a7:3c:52:74:40)
└ Internet Protocol Version 4, Src: 192.168.1.10, Dst: 192.168.1.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        Total Length: 52
        Identification: 0xbdd5 (48597)
        Flags: 0x00
        Fragment Offset: 0
    < Time to Live: 1
        > [Expert Info (Note/Sequence): "Time To Live" only 1]
            Protocol: UDP (17)
            Header Checksum: 0x7888 [validation disabled]
            [Header checksum status: Unverified]
            Source Address: 192.168.1.10
            Destination Address: 192.168.1.1
    > User Datagram Protocol, Src Port: 48596, Dst Port: 33435
    > Data (24 bytes)
```

知乎 @AlanZhang

当目的主机收到 UDP 包后，会返回 ICMP 差错报文消息（类型 3，代码 3）。参照上面的表，该报文类型是端口不可达，说明发送方发出的 UDP 包到达了目的主机。如下图：

No.	Time	Source	Destination	Protocol	Length	Info
17	3.619160	192.168.1.10	192.168.1.1	UDP	66	48596 → 33435 Len=24
18	3.630642	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)
19	3.634969	192.168.1.10	192.168.1.1	UDP	66	48596 → 33436 Len=24
20	3.638519	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)
21	3.638972	192.168.1.10	192.168.1.1	UDP	66	48596 → 33437 Len=24
22	3.641892	192.168.1.1	192.168.1.10	ICMP	94	Destination unreachable (Port unreachable)

```
> Frame 18: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface en0, id 0
> Ethernet II, Src: Sichuant_52:74:40 (48:a7:3c:52:74:40), Dst: Apple_9d:8d:82 (14:7d:da:9d:8d:82)
> Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.1.10
└ Internet Control Message Protocol
    Type: 3 (Destination unreachable)
    Code: 3 (Port unreachable)
    Checksum: 0x808a [correct]
    [Checksum Status: Good]
    Unused: 00000000
    > Internet Protocol Version 4, Src: 192.168.1.10, Dst: 192.168.1.1
    > User Datagram Protocol, Src Port: 48596, Dst Port: 33435
    > Data (24 bytes)
```

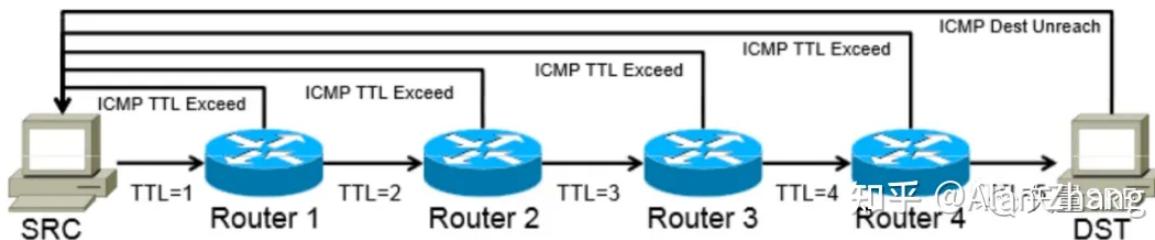
知乎 @AlanZhang

这样的过程，traceroute 就可以拿到了所有的路由器 IP，这样子就可以看到从源主机到目的主机过程中的所有路由信息。

当然实际情况有的路由器禁用 ICMP，那么他就根本不会返回这个 ICMP 差错报文，所以是看不到中间经过的路由IP的。

**Tips:** traceroute 在类 Unix/Linux 系统中默认使用的是 UDP 协议，也可以通过参数修改为使用 ICMP 协议；windows 操作系统中只使用 ICMP 协议。

说了这么多，我们还是直接来看一张图吧，基本可以描述清楚 traceroute 的整个过程。如下图：



## ICMP作为IP的上层协议在工作

ICMP 的内容是放在 IP 数据包的数据部分里来互相交流的。也就是，从 ICMP 的报文格式来说，ICMP 是 IP 的上层协议。但是，正如 RFC 所记载的，ICMP 是分担了 IP 的一部分功能。所以，被认为是与 IP 同层的协议。看一下 RFC 规定的数据包格式和报文内容吧。

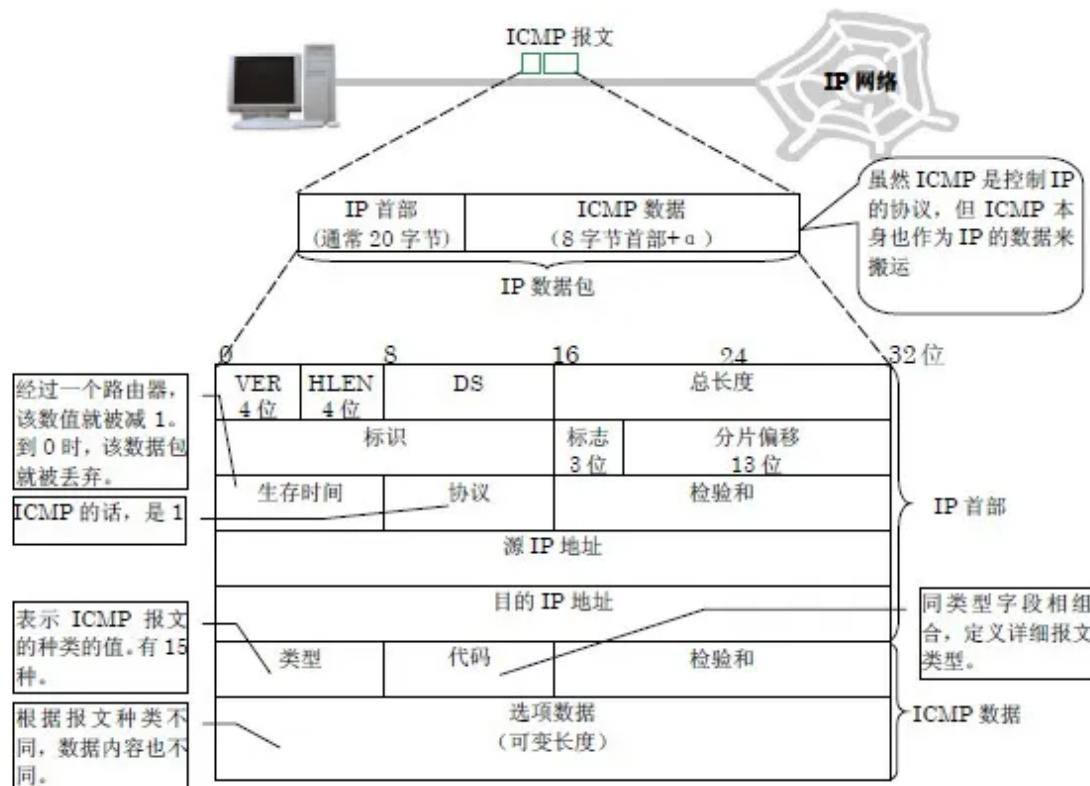


图 3 理解 ICMP 报文的格式

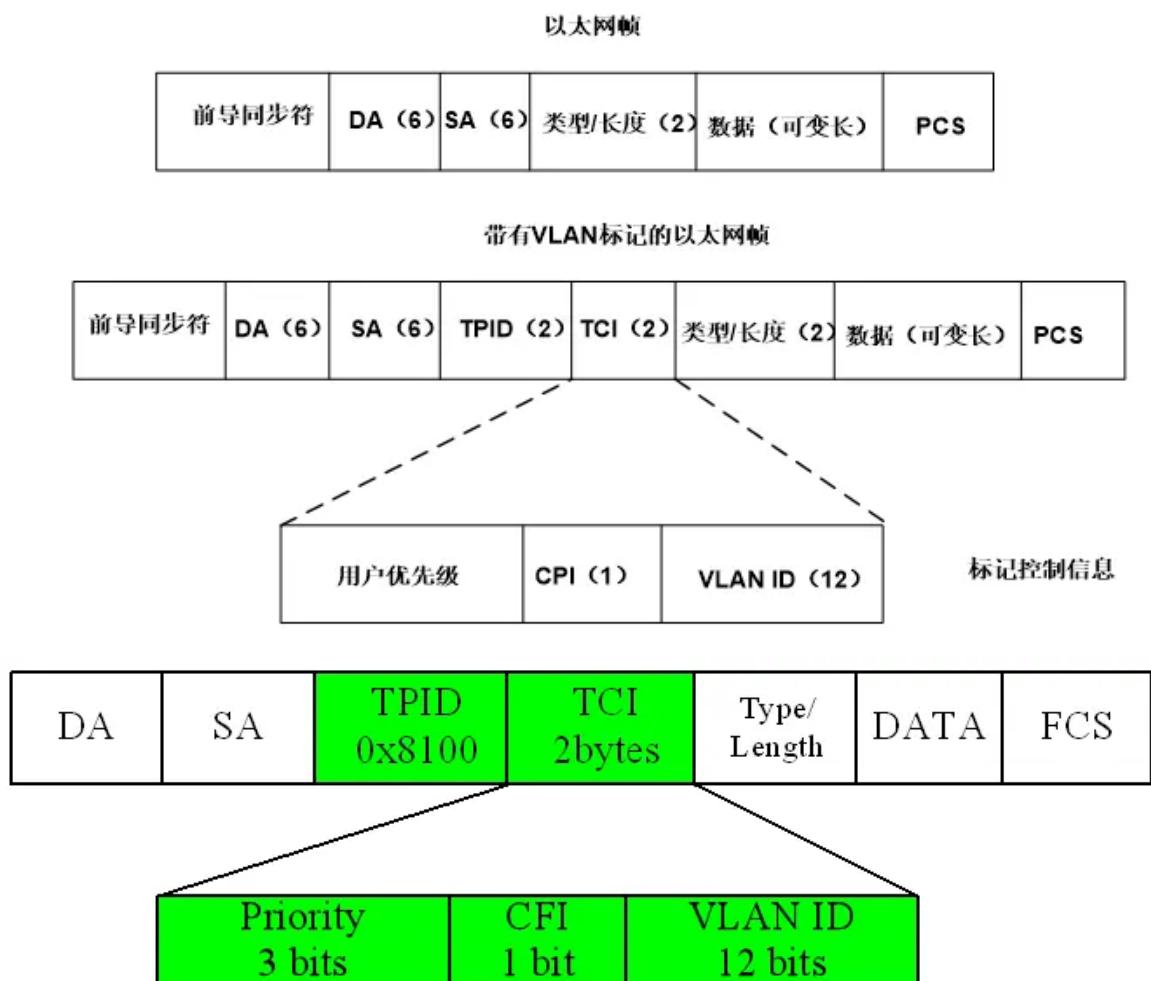
知乎 @内核补给站

# 总结

以上内容就是关于ICMP协议的全部内容啦，相信你看完本篇文章就可以深入理解每一次ping, traceroute背后的工作原理啦。

## TCP/IP协议专栏——MAC地址详解

### 以太帧格式



- 前导同步符：共8字节，由7个字节的前同步信号和一个分界符开始字节构成。
- DA/SA：共6个字节，前3个字节代表供应商代码，后3个字节代表厂商序列号。
- TPID：标记协议标识符，2个字节，值位16进制的8100。
- TCI：标记控制信息，2个字节，包括3个Bits的用户优先级 (IEEE802.1P), 1Bits的规范格式标识符 (CFI), 用于标识MAC地址是否规范，12Bits的VID, 表示该帧所属的VLAN。
- 类型/长度：为类型时，表示是Ethernet II帧结构，类型取值大于1536，为长度时表示IEEE802.3帧结构。

- 数据区：为第三层协议，不支持VLAN标记时，长度为46 - 1514字节，支持VLAN标记时，由于增加了4个字节，长度为50-1518字节。
- FCS：是帧校验。

## MAC地址简介

1. MAC 地址是硬件标识（Media Access Control Address）。
2. MAC地址长度为：6字节：48bit，
3. 通常用12位16进制数表示，每两个数之间用冒号隔开。如：  
00:D0:D0:C0:3F:A0就是一个MAC地址：
4. 前6位16进制数00:D0:D0代表网络硬件制造商的编号，由IEEE分配；
5. 后6位16进制数C0:3F:A0代表该制造商所制造的某个网络产品（如网卡）的系列号。
6. 每个网络制造商必须确保它所制造的每个以太网设备都具有相同的前六位以及不同的后六位。

## MAC表分类

1. MAC表包括软件表和硬件表。
2. MAC地址表的表项由MAC地址和VLAN ID唯一标识。
3. 只要MAC地址和VLAN ID部分相同，就认为是同一个表项。
4. 交换机维护着一张MAC地址表，MAC地址和交换机的端口一一对应

## MAC模块性能指标—主要性能指标

1. 学习速率：例如1500个/秒；
2. MAC表容量：即MAC表最多支持写入多少条MAC
3. 老化时间精度：若老化时间为Tage，则要求在1 ~ 2Tage内老化掉无效MAC地址
4. 冲突链长度：MAC表为HASH表，并且HASH冲突链是有限的。若HASH链过短，则即使MAC表未满，也可能出现MAC地址无法添加到MAC表中。不同设备冲突链长度不同。

## C语言中#pragma pack(1)的用法

---

# 一：何时使用

#pragma pack(1)的用法大多是用在结构体中

## 二. 为什么使用#pragma pack(1)

结构体的字节对齐方式在不同的编译器中不同，会存在数据冗余，以下举个例子

```
struct example
{
    char header_start;
    double data_type;
};
```

现有的结构体，就会按照结构体成员中最大的数据类型对齐，例子当中就是double型，按照8个字节进行对齐。那么此时sizeof(example)就是16，存在7个空字节，因为其中char只占一个字节

如果加上#pragma pack(1)，那么example按1个字节对齐方式对齐，此时sizeof(example)就是等于9

## 三. 注意点

这种方法的使用一定要是成对使用，如下面例子

```
#pragma pack(1)
struct example
{
    char header_start;
    double data_type;
};
#pragma pack()
```

我们一定要在结构体末尾加上#pragma pack()进行取消自定义字节对齐的命令，如果不取消，可能会导致整个程序存在问题。因为会影响到其他的结构体对齐方式

## 四.#pragma pack()的一些用法

- #pragma pack(show) //显示当前内存对齐的字节数，编辑器默认8字节对齐
- #pragma pack(n) //设置编辑器按照n个字节对齐，n可以取值1,2,4,8,16
- #pragma pack(push) //将当前的对齐字节数压入栈顶，不改变对齐字节数
- #pragma pack(push,n) //将当前的对齐字节数压入栈顶，并按照n字节对齐
- #pragma pack(pop) //弹出栈顶对齐字节数，不改变对齐字节数
- #pragma pack(pop,n) //弹出栈顶并直接丢弃，按照n字节对齐

## 五.题目

牛牛需要建立一个结构体Nowcoder，该结构体包括三个成员，其类型分别是int、double和char。假设牛牛想让这个结构体所占据的内存最小，请问你该怎么建立该结构体？（输入三个相应类型的变量用于初始化结构体）

### 示例1

输入：1,1.000,a

返回值：13

### 代码

```
/**  
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的  
 * 值即可  
 *  
 * @param n int整型  
 * @param d double浮点型  
 * @param c char字符型  
 * @return int整型  
 */  
  
#pragma pack(1)
```

```

struct Nowcoder{
    char c;
    int n;
    double d;
};

int smaller_space(int n, double d, char c ) {
    // write code here
    struct Nowcoder temp;
    temp.c=c;
    temp.n=n;
    temp.d=d;
    return sizeof(temp);
}

```

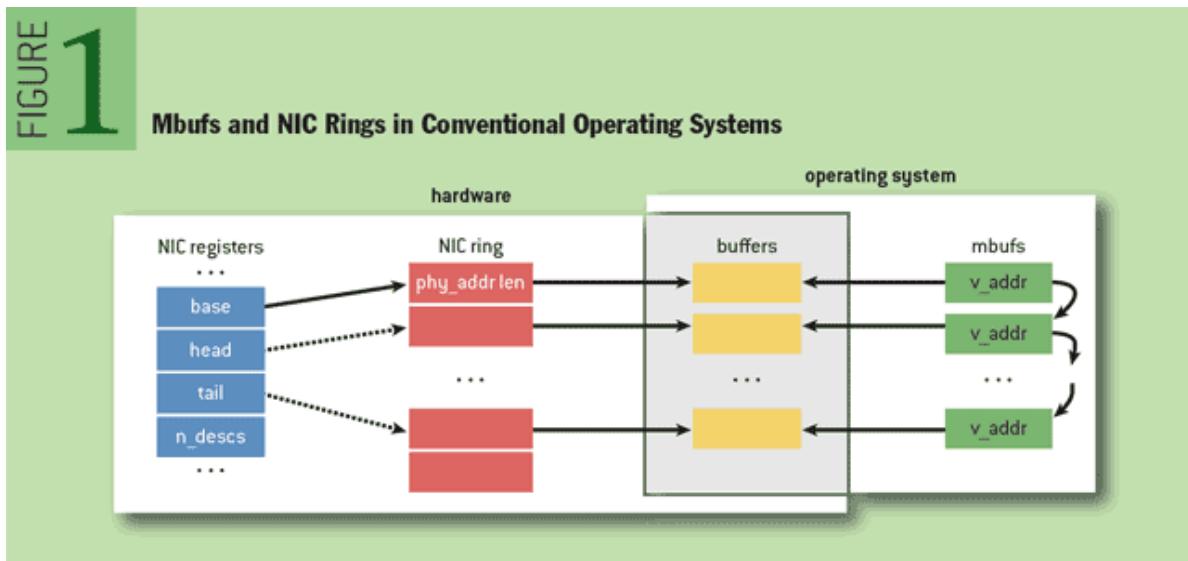
## netmap 介绍

`netmap` 是一个高效的收发报文的 I/O 框架，已经集成在 FreeBSD 的内部了。

### 一、架构

现在的网卡都使用多个 `buffer` 来发送和接收 packet，并有一个叫 `NIC ring` 的环形数组。

`NIC ring` 是静态分配的，它的槽指向 `mbufs` 链的部分缓冲区。

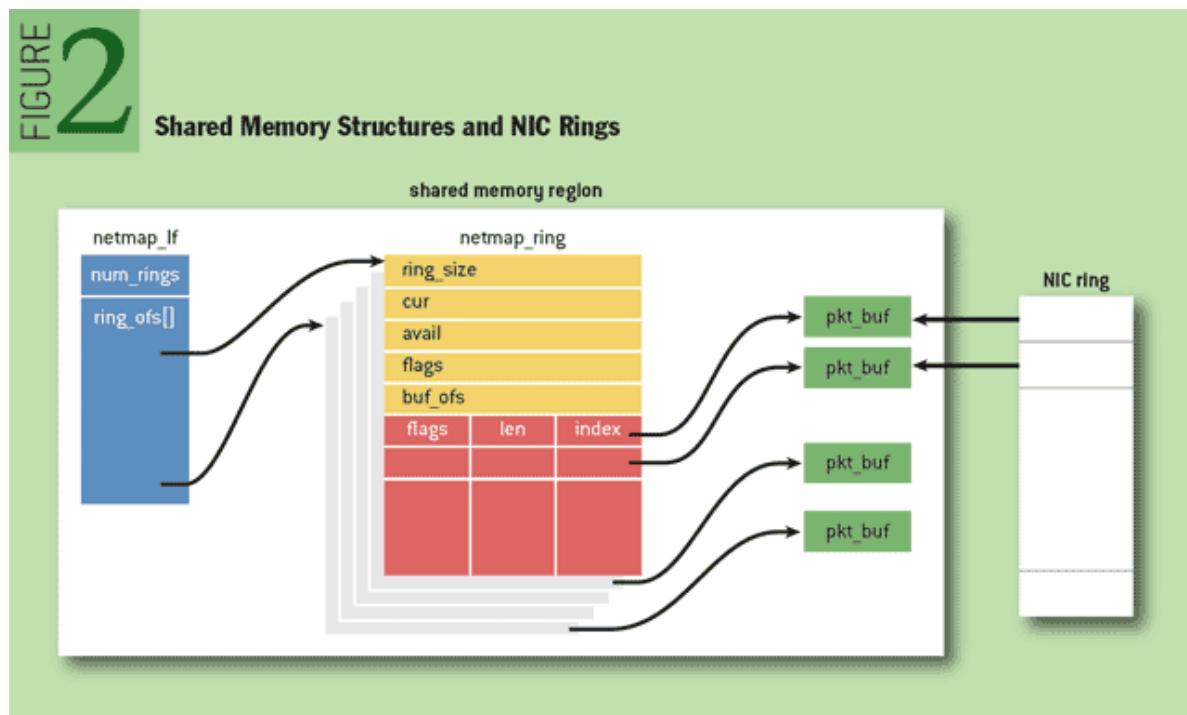


netmap 内存映射网卡的packet buffer到用户态，实现了自己的发送和接收报文的circular ring来对应网卡的 ring，使用 netmap 时，程序运行在用户态，即使出了问题也不会 crash 操作系统。

下图显示了一个接口可以有多个 netmap ring。

将文件描述符绑定到 NIC 时，应用程序可以选择将所有 ring 或仅一个 ring 附加到文件描述符。

- 使用所有 ring，相同的代码可以用于单队列或多队列 NIC。
- 使用一个 ring，可以通过每个 ring 一个进程/CPU core 来构建高性能系统，从而在系统中并行。



netmap 使用 poll 等待网卡的文件描述符可接收或可发送。

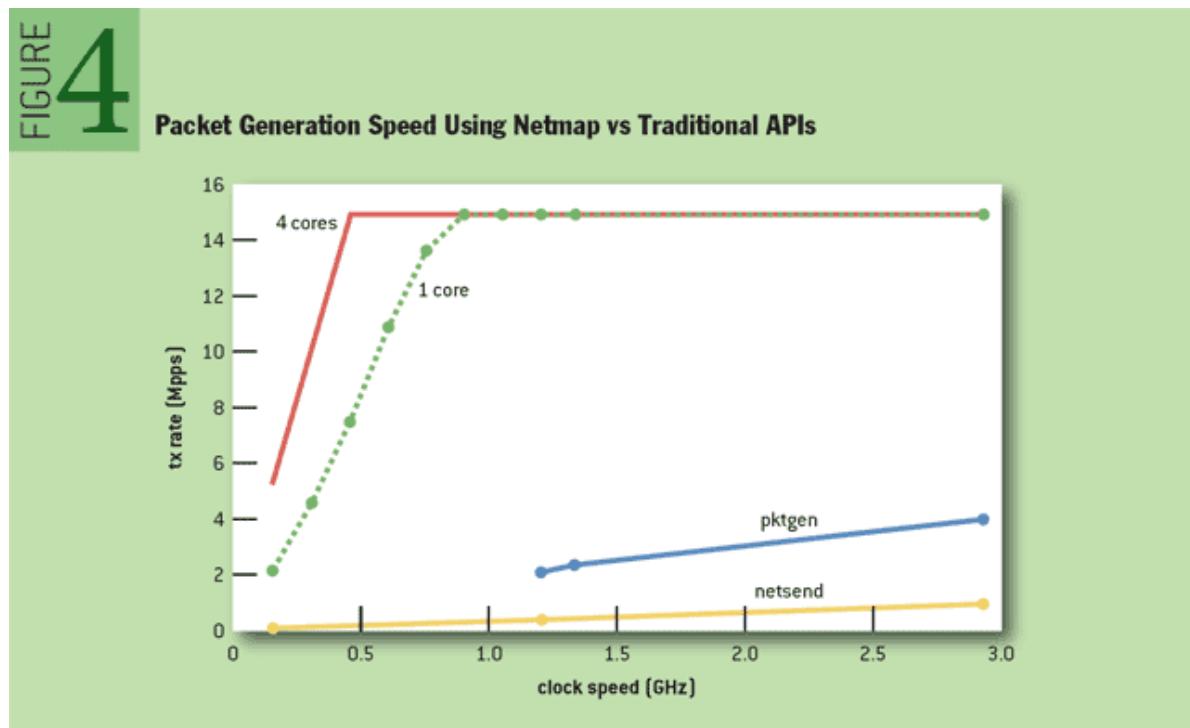
netmap 会建立一个字符设备 `/dev/netmap`，然后通过 `nm_open` 来注册网卡为 netmap 模式。

注意：这里顺便提一下，网卡进入 `netmap` 模式后，`ifconfig` 是看不到网卡统计信息变化的，`wireshark` 也抓不到报文，因为协议栈被旁路了。

内存映射的区域里面，有网卡的收发队列，这样可以通过将接收缓冲区的地址写在发送的 ring 里面实现零拷贝（Zero-copy）。

## 二、性能

netmap 官网说在 10GigE 上测试，发包速率可以达到 14.88Mpps，收包的速率和发包相近。同时还支持多网卡队列。



我在网络设备 Intel 82599EB 10-Gigabit 上测试，发包速率和 netmap 官网的速度差不多，可以达到 1400W pps，不过收包要比这个低很多。。。不知道是不是我测的方式有问题。

## 三、例子

这是官方的例子，不过已经很老了（也就是不能直接用了^\_^），但是还是可以大概说明使用过程的：

```
struct netmap_if *nifp;
struct nmreq req;
int i, len;
char *buf;

fd = open("/dev/netmap", 0); // 打开字符设备
strcpy(req.nr_name, "eth0");
ioctl(fd, NIOCREG, &req); // 注册网卡
mem = mmap(NULL, req.nr_memszie, PROT_READ | PROT_WRITE, 0,
fd, 0);
nifp = NETMAP_IF(mem, req.nr_offset);
```

```
for (;;) {
    struct pollfd x[1];
    struct netmap_ring *ring = NETMAP_RX_RING(nifp, 0);

    x[0].fd = fd;
    x[0].events = POLLIN;
    poll(x, 1, 1000);
    for (; ring->avail > 0 ; ring->avail--) {
        i = ring->cur;
        buf = NETMAP_BUF(ring, i);
        use_data(buf, ring->slot[i].len);
        ring->cur = NETMAP_NEXT(ring, i);
    }
}
```

# 用户空间协议栈设计和netmap综合指南，将网络效率提升到新高度

**简介：**这篇综合指南将深入探讨用户空间协议栈设计和netmap技术，以提高网络效率。我们将详细介绍用户空间协议栈的工作原理、优点和挑战，并提供一系列优化策略。同时，我们将重点介绍netmap技术，它是一个高性能数据包I/O框架，可以极大地提升网络吞吐量和响应速度。通过结合用户空间协议栈设计和netmap技术，读者将了解如何最大限度地提高网络连接的速度和效率。本指南适用于网络开发人员、系统管理员和对网络性能优化感兴趣的读者。无论你是初学者还是有经验的专业人士，我们相信这篇文章将为你带来全面的见解和实用的指导，帮助你将网络效率提升到一个新的高度。

## 一、协议概念

### 1.1、七层网络模型和五层网络模型

## 七层网络模型

应用层

表示层

会话层

传输层

网络层

数据链路层

物理层

# 五层网络模型

应用层

传输层

网络层

数据链路层

物理层

- **应用层：**

最接近用户的一层，为用户程序提供网络服务。主要协议有HTTP、FTP、TFTP、SMTP、DNS、POP3、DHCP等。

- **表示层：**数据的表示、安全、压缩。管理数据的解密和加密。
- **会话层：**负责在网络中的两个节点之间的建立、维持和终止通信。
- **传输层：**

模型中最重要的一层，负责传输协议的流控和差错校验。数据包离开网卡后进入的就是传输层；主要协议有：TCP、UDP等。

- **网络层：**将网络地址翻译成对应的物理地址。主要协议有：ICMP、IP等。
- **数据链路层：**

建立逻辑连接、进行硬件地址寻址、差错校验等功能，解决两台相连主机之间的通信问题。主要协议有SLIP、以太网协议/MAC帧协议、ARP和RARP等。

- **物理层：**

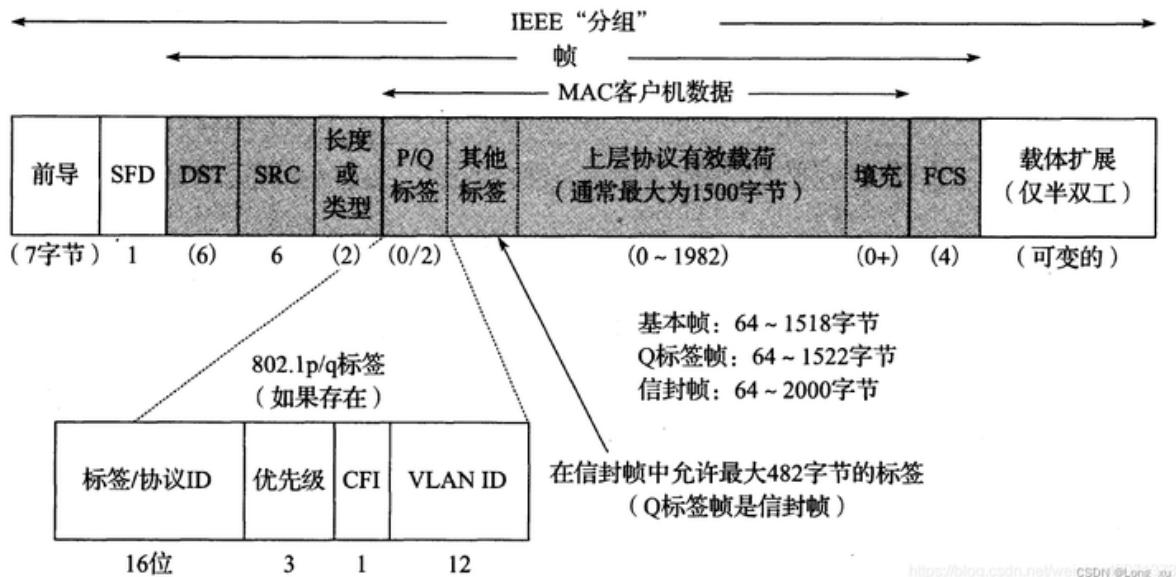
模型的最低层，建立、维护、断开物理连接，传输比特流。常见的物理媒介有光纤、电缆、中继器等。主要协议有RS232等。

## 1.2、以太网

以太网不是一种网络，而是一种局域网技术，它既有数据链路层内容，也有一些物理层内容。局域网技术除了以太网外，还有令牌环网、无线LAN/WAN等。

**以太网的网线必须是双绞线，以太网中的所有主机共享一个通信通道；**当局域网中一台主机发送数据后，该局域网的所有设备都会收到该数据。因为共用一个通信通道，因此同一时刻只允许一台主机发送数据；如果同一时刻不只有一个主机发送数据，为避免干扰，该主机会执行碰撞避免算法（等待一段时间后再进行数据重发）。

以太网帧格式如下：



源地址和目的地址是指网卡MAC地址，长度是48 bit（6字节）。帧协议类型字段有三种，分别对应IP协议、ARP协议和RARP协议。帧末尾是CRC校验码。

定义一个以太网头结构体示例代码：

```
#define ETHER_ADDR_LEN      6
struct etherhdr {
    unsigned char dst_mac[ETHER_ADDR_LEN];
    unsigned char src_mac[ETHER_ADDR_LEN];
    unsigned short protocol;
};
```

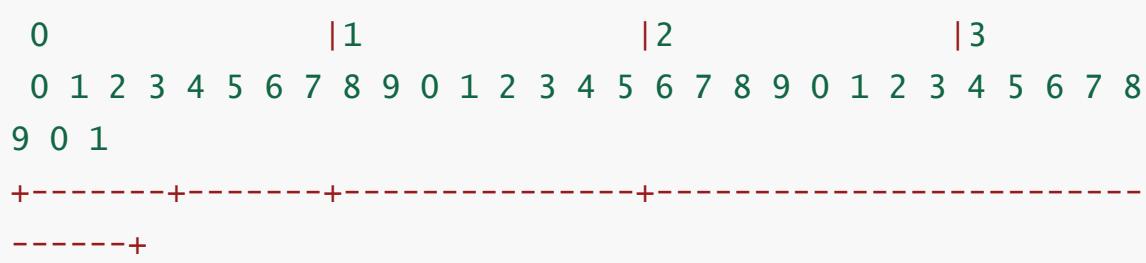
输出它的大小：

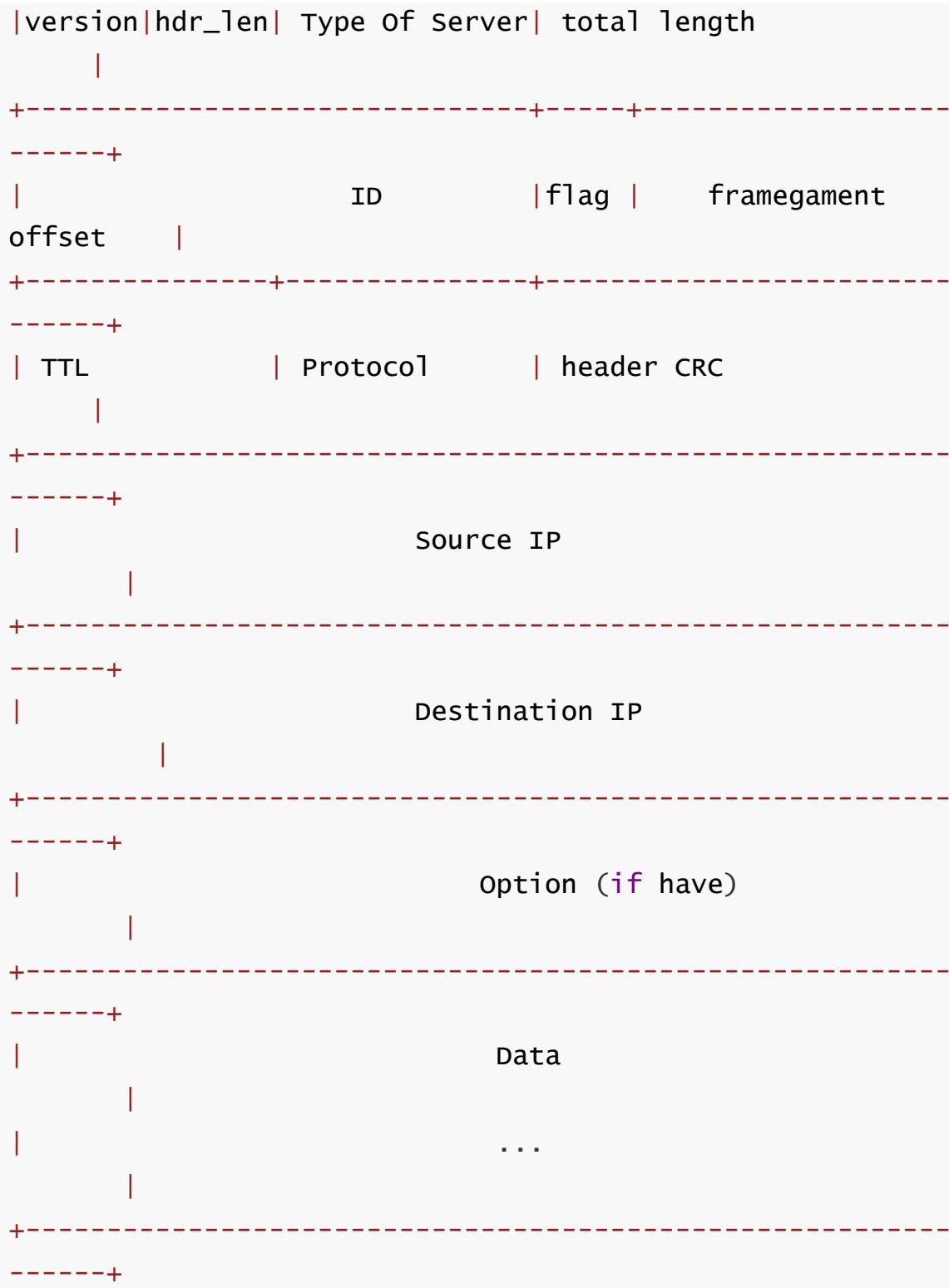
```
sizeof(struct etherhdr) = 14
```

### 1.3、IP协议

IP协议全称Internet Protocol，即网际互连协议，存在于网络层，负责数据在网络中传输。

IP协议格式如下：





定义一个IP协议头结构体示例代码：

```
struct iphdr {
    unsigned char version : 4,
    hdrlen : 4;

    unsigned char tos;
```

```
unsigned short totlen;

unsigned short id;
unsigned short flag : 3,
    offset : 13;
unsigned char ttl;

unsigned char protocol;

unsigned short check;

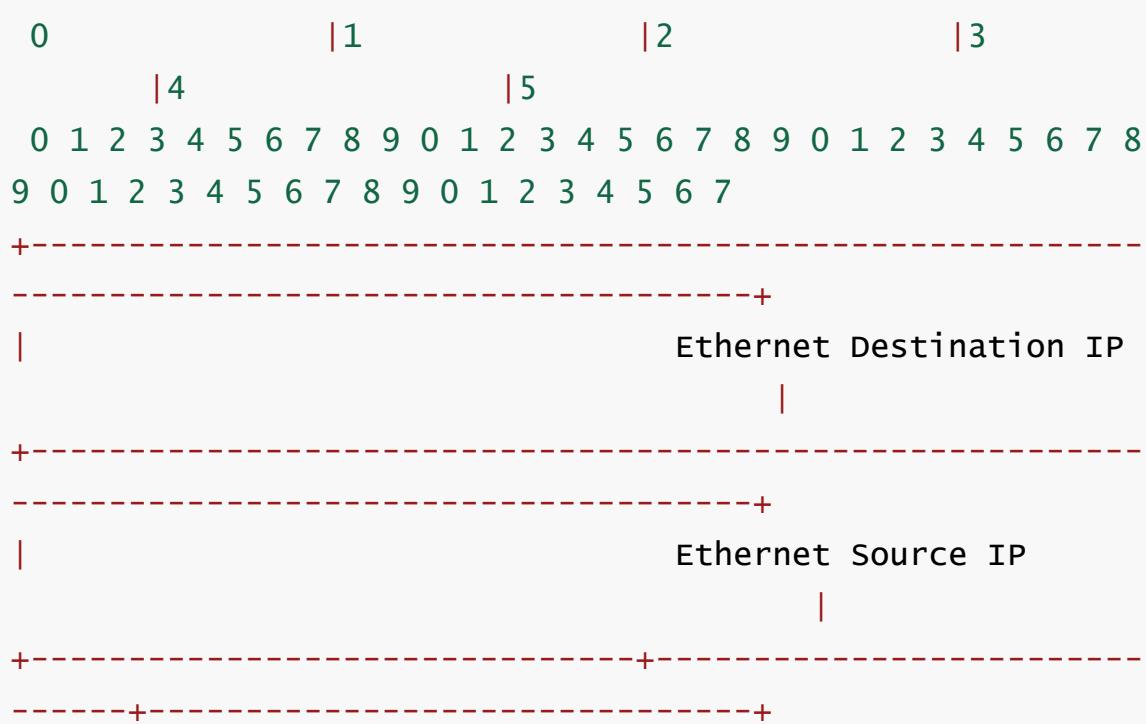
unsigned int sip;
unsigned int dip;
};
```

## 1.4、ARP协议

ARP协议全称Address Resolution Protocol，即地址解析协议，是根据IP地址获取MAC地址的一个TCP/IP协议。

**ARP协议的作用：**在同一个局域网中要给对方发消息，就必须得知道对方的MAC地址，而实际大部分情况下只知道对方的IP地址，因此需要**通过ARP协议来根据IP地址来获取目标主机的MAC地址。**

ARP的数据格式如下：





可以看出，ARP是MAC帧协议的上层协议，前3个字段和最后一个字段对应的就是以太网头部。由于ARP数据包的长度不足46字节，因此ARP数据包在封装成为MAC帧时还需要补上18字节的填充字段。

定义一个arp协议头结构体示例代码：

```
struct arphdr{  
    unsigned short h_type;  
    unsigned short h_proto;  
    unsigned char   h_addrlen;  
    unsigned char   protolen;  
    unsigned short  oper;  
    unsigned char   smac[ETH_ALEN];  
    unsigned int    sip;  
    unsigned char   dmac[ETH_ALEN];  
    unsigned int    dip;  
    // pad  
};
```

#### 1.4.1、ARP攻击原理

**arp攻击得到主要目的是使网络无法正常通信。** 向局域网中的所有主机发送ARP应答，其中包含网关IP地址和虚假的MAC地址。局域网中的主机收到ARP应答跟新ARP表后，再发送数据时，就会发送到虚假的MAC地址导致通信故障，就无法和网关正常通信，导致无法访问互联网。

#### 1.4.2、ARP欺骗原理

**ARP欺骗并不会使网络无法正常通信，而是通过冒充网关或其他主机** 使得到达网关或主机的数据流量通过攻击主机进行转发。

比如冒充网关：ARP欺骗发送arp应答给局域网中其他的主机，其中包含网关的IP地址和进行ARP欺骗的主机MAC地址;并且也发送了ARP应答给网关，其中包含局域网中所有主机的IP地址和进行arp欺骗的主机MAC地址。当局域网中主机和网关收到ARP应答跟新ARP表后，主机和网关之间的流量就需要通过攻击主机进行转发。

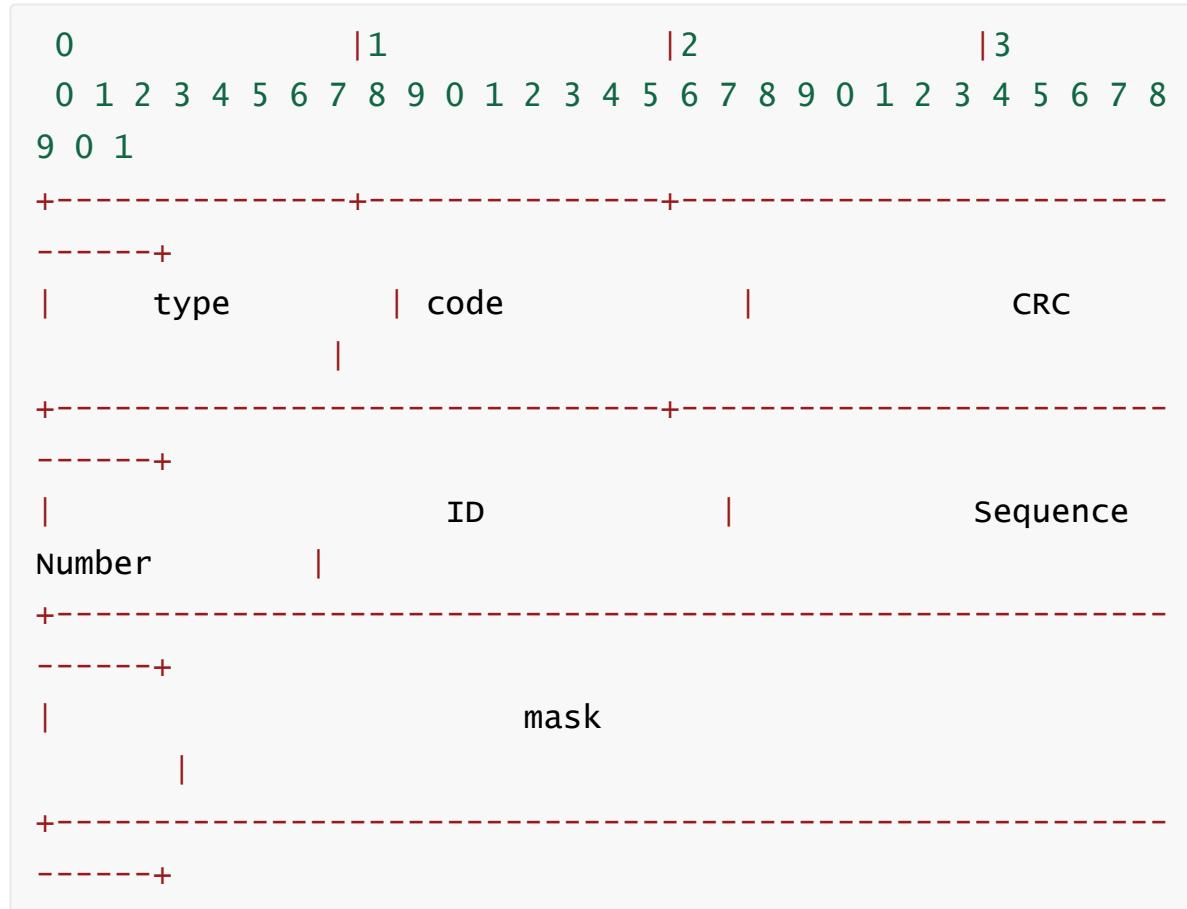
冒充主机的过程和冒充网关相同。

### 1.5、ICMP协议

ICMP全称Internet Control Message Protocol，即互联网控制消息协议，位于 IP 报文的数据段。虽然ICMP是网络层协议，但它不直接传递数据到数据链路层，而是封装成IP数据包再传递到数据链路层，IP数据包中的协议类型字段为1就表示ICMP报文。ICMP协议的类型主要有两类：查询报文和差

错报文。

ICMP报文格式如下：



标识	含义
type	类型，0 代表应答 ICMP 报文、8 代表请求 ICMP 报文。
code	代码，type= 8 && code= 0 表示回显请求 (ping 请求)；type= 0 && code= 0 表示回显应答 (ping 应答)；type = 11 && code = 0 表示超时。
CRC	校验和，包括数据在内的整个 ICMP 数据报的检验和
ID	标识符，将发送进程的 ID 号放置在标识符字段，这样即使在主机上运行了多个 ping 程序，ping 程序也可以识别出返回的信息。
Sequence Number	序列号，从 0 开始，每发送一次心得回显请求就加 1。
mask	子网掩码

定义一个ICMP协议头结构体示例代码：

```
// ICMP
struct icmpphdr {
    unsigned char type;
    unsigned char code;
    unsigned short check;
    unsigned short identifier;
    unsigned short sep;
    unsigned char data[32];
};
```

### ICMP的应用：

- ping命令。向目的服务器发送回显请求，目的服务器发送回显应答；计算发送回显请求数据包的时间与接收到回显应答数据包的时间差，就是数据包一去一回所需要的时间。
- traceroute命令。traceroute命令利用 ICMP 差错报文类型，用作追踪路由信息。前提条件是路由器没有禁用 ICMP。

## 1.6、MTU概念

MTU，全称Maximum Transmission Unit，即最大传输单元。说明一次数据帧可以发送或接收的最大数据量；以字节为单位，一般是1500，不同网络类型的MTU不同。

1. 如果一次发送要发送的数据超过MTU，需要在IP层对数据进行分片。数据分片和组装在IP层，因为不同网络的MTU不同，不仅源主机可能需要对数据进行分片，数据传输过程中的路由器也可能对数据分片。
2. 以太网规定数据的最小长度为46字节，如果发送数据小于46字节，需要填充，比如ARP数据包就需要填充才能发送。
3. 对于UDP，是定长的8字节报头，如果IP报头没有携带可选项字段，那么UDP一次携带的数据最大为 $1500 - 20 - 8 = 1472$ 字节，如果超出这个大小就需要在IP层进行分片。分片带来的后果是增加UDP的丢包率。
4. 分片也会增加TCP的丢包率，不过TCP有重传机制；因此需要尽可能避免分片，降低TCP重传次数。

## 1.7、MSS概念

MSS，全称Maximum Segment Size，即最大报文段大小。表示TCP传往另一端的最大块数据的长度。

当一个连接建立时，连接的双方都要提供各自的MSS。通过协商确定MSS的值（双方MSS的最小值）以避免TCP分片。如果没有分段发生，MSS越大越好。

## 1.8、TTL概念

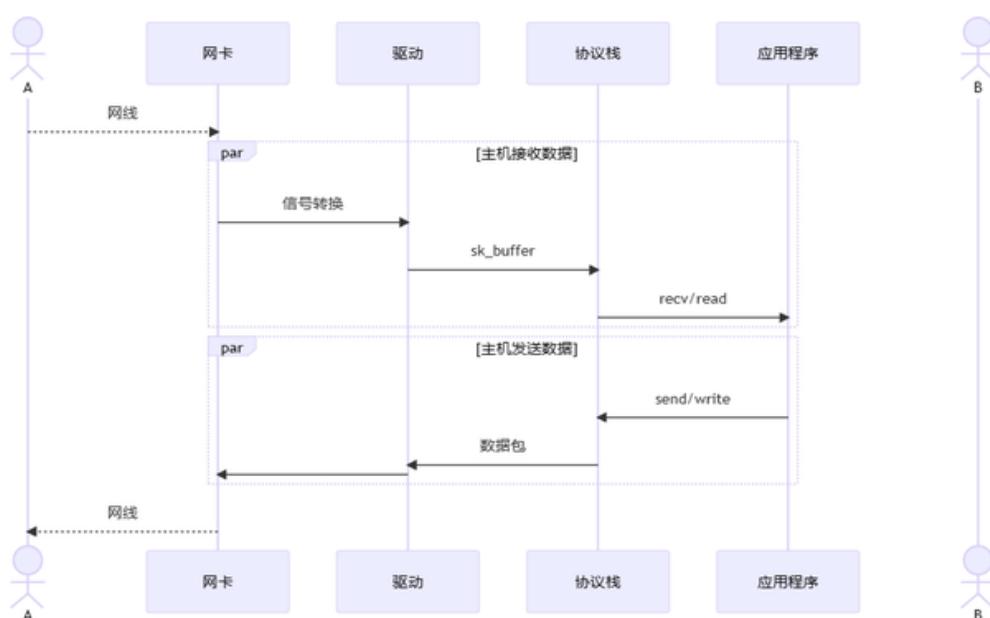
TTL，全称Time To Live，即存活时间；指一个数据包可传递的最长距离（跃点数）。

当一个数据包经过一个路由器时，TTL减一；当TTL=0时路由器就会取消数据包的转发。

我们知道网络是有环存在的，设计TTL的目的是防止数据包因为不正确的路由表等原因造成无线循环而无法送达导致耗尽网络资源。

## 二、数据传输框图

网络上所有的数据传输都要经过网卡，网卡将模拟信号转换为数字信号，也就是将物理层信号转换为数据链路层信号。



## 注意：

- send()返回成功不代表发送成功，send()只是把数据拷贝到写缓冲区，真正发送数据由协议栈完成。如果客户端宕机而服务端一直执行send()，那么在一段时间后send()会返回-1；因为写缓冲区中的数据没有发送出去导致写缓冲区爆满。
- 协议栈就是数据根据七层网络模型，自顶向下一层一层的协议头包住数据；接收端也是根据七层网络模型，自底向上一层层的解析协议。
- 驱动如何把数据传递到协议栈？

在Linux kernel有一个sk\_buffer结构，sk\_buffer将驱动获取的数据通过sk\_buffer传递到协议栈中。关于整个过程的执行流程可以参考这篇文章：[链接](#)。

## 三、校验和 checksum 的计算方法

- (1) 先将需要计算checksum数据中的checksum字段设为0；
- (2) 将checksum的数据按2 byte(16 bit)划分，如果最后有单个byte的数据，则在其后面补1 byte的0构成2 byte；
- (3) 将所有的2 byte(16 bit)值累加，得到一个4 byte (32 bit) 的值；
- (4) 将得到的4 byte (32 bit) 的值的高16bit与低16bit相加得到一个新的4 byte (32 bit) 值；若新值大于0xFFFF，再将新值的高16bit与低16bit相加。
- (5) 将上一步计算所得的值按位取反，即得到checksum值，保存到checksum字段即可。

示例代码：

```
unsigned short in_cksum(unsigned short *addr, int len)
{
    register int nleft = len;
    register unsigned short *w = addr;
    register int sum = 0; //32bit
    unsigned short answer = 0; //16bit

    while (nleft > 1)
    {
        sum += *w++; //16bit为一组累加
    }
}
```

```

    nleft -= 2;
}

if (nleft == 1)//存在单个byte情况
{
    *(u_char*)&answer) = *(u_char*)w;
    sum += answer;
}
sum = (sum >> 16) + (sum & 0xffff); // 高16bit与低16bit
相加
sum += (sum >> 16); //防止值大于0xffff

//结果
answer = ~sum;

return (answer);
}

```

## 补：C++中的register变量

用register说明的局部变量称为寄存器变量，该变量将可能以寄存器作为存储空间。

register说明仅能建议（而非强制）系统使用寄存器，这是因为寄存器虽然存取速度快，但个数有限，当寄存器不够用时，该变量仍按auto变量处理。

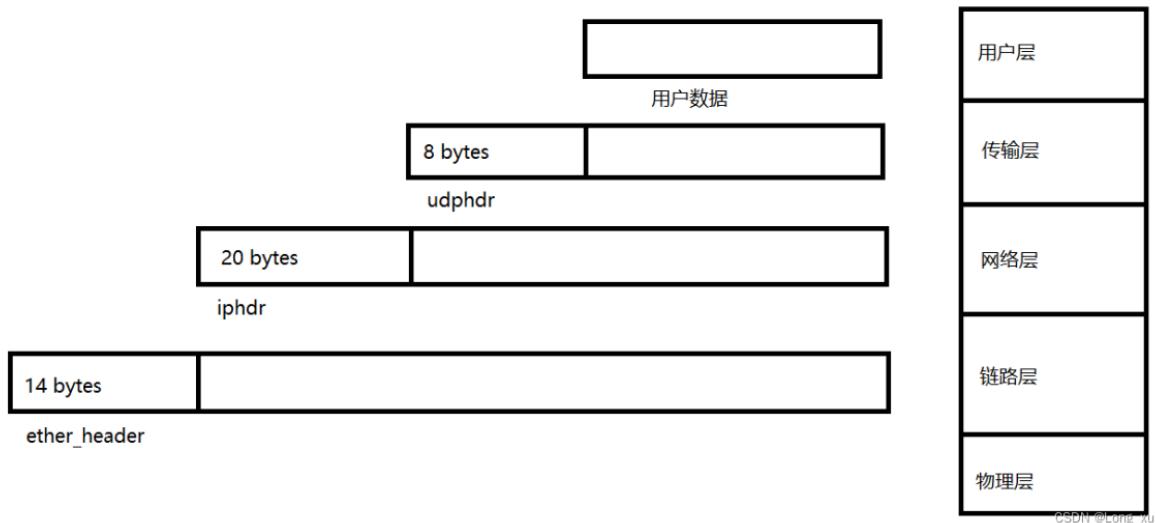
一般在短时间内被频繁访问的变量置于寄存器中可提高效率。不过并不建议经常使用register变量，理由如下。

1. 在许多情况下使用register变量效果并不明显；
2. 有些版本的C++编译系统具有对局部变量按某种策略自动决定可否占用可用寄存器的功能，效果比程序员决定可能要好一些；
3. 局部变量存于寄存器时它将没有内存地址，可能影响与寻址有关的操作，如寻址运算符&的操作。

# 四、协议栈设计--netmap

要实现一个协议栈，那么就需要获得原始的协议数据。

## Udp数据帧



## 4.1、获取原始协议数据的方法

(1) raw socket, 即原始套接字，可以接收本机网卡的数据帧或数据包。有四种方式创建这类socket。

目标	实现		
发送 接收 IP数 据包	socket(PF_INET,SOCK_RAW,IPPROTO_TCP \\\n  IPPROTO_UDP \\\n  IPPROTO_ICMP) ;	IPPROTO_UDP \\	IPPROTO_ICMP)
发送 接收 以太 网数 据帧	socket(PF_PACKET,SOCK_RAW,htons(ETH_P_IP \\\n  ETH_P_ARP \\\n  ETH_P_ALL));	ETH_P_ARP \\	ETH_P_ALL));
发送 接收 以太 网数 据帧 (不 包括 以太 网头 部)	socket(PF_PACKET,SOCK_DGRAM,htons(ETH_P_IP \\	ETH_P_ARP \\	ETH_P_ALL));

(2) 旁路。[netmap](#)、dpdk等

(3) hook。bpf、ebpf等

## 4.2、零长数组

零长数组，顾名思义，就是长度为零的数组。一般在GNU C中使用，其他编译器使用可能会报错或警告。

零长度数组的一个特点是它不占用内存存储空间。如下示例：

```
#include <stdio.h>

char test[0];

int main()
{
    printf("size = %ld\n", sizeof(test));
    return 0;
}

// 输出为 0
```

在结构体中使用，它同样也不占内存：

```
#include <stdio.h>

struct test{
    int len;
    int ch[0];
};

int main(void)
{
    printf("size of = %ld\n", sizeof(struct test));
    return 0;
}
```

零长数组的使用：内存已经分配，但数据长度不确定，需要计算出数据长度的，就可以使用零长数组。零长数组在内存池中使用比较多。

使用示例：

```
#include <stdio.h>
```

```
struct test{
    int len;
    char ch[0];
};

int main(void)
{
    struct test *buf;
    buf = (struct test *)malloc(sizeof(struct test)+ 16);
    memset(buf,0,sizeof(struct test)+ 16);

    strcpy(buf->ch, "hello world\n");
    puts(buf->ch);

    free(buf);
    return 0;
}
```

## 4.3、修改ens33为eth0

(1) 打开/etc/default/grub

```
sudo nano /etc/default/grub
```

(2) 找到GRUB\_CMDLINE\_LINUX="" 改为

**GRUB\_CMDLINE\_LINUX="net.ifnames=0 biosdevname=0"**

(3) 写入配置

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

(4) 重启系统

```
reboot
```

## 4.4、netmap下载安装

以ubuntu为例。

(1) 切换到根目录：

```
cd /
```

(2) 切换到root权限:

```
sudo su
```

(3) 在根目录clone netmap:

```
git clone https://github.com/luigirizzo/netmap.git
正克隆到 'netmap' ...
remote: Enumerating objects: 28670, done.
remote: Counting objects: 100% (978/978), done.
remote: Compressing objects: 100% (397/397), done.
remote: Total 28670 (delta 603), reused 867 (delta 533),
pack-reused 27692
接收对象中: 100% (28670/28670), 10.13 MiB | 2.72 MiB/s, 完成.
处理 delta 中: 100% (18306/18306), 完成.
```

(4) 安装编译环境:

```
apt-get install build-essential
```

build-essential的作用:

想要安装开发工具软件包，以拥有 sudo 权限用户身份或者 root 身份运行下面的命令：

```
sudo apt update
sudo apt install build-essential
```

这个命令将会安装一系列软件包，包括gcc, g++, 和make。

你可能还想安装关于如何使用 GNU/Linux 开发的手册。

```
sudo apt-get install manpages-dev
```

通过运行下面的命令，打印 GCC 版本，来验证 GCC 编译器是否被成功地安装。

```
gcc --version
```

## (5) 进入netmap/LINUX 目录:

```
cd /netmap/LINUX/
```

## (6) 执行配置:

```
./configure
```

此过程会下载一些东西，然后提示耐心等待一段时间，过程有点久，请耐心等待，如下。

```
apps                      dedup vale-ctl nmreplay tlem lb bridge pkt-gen
native drivers             mlx5 i40e ice ixgbe igb virtio_net.c

Contents of the drivers.mak file:
mlx5@conf := CONFIG_MLX5_CORE_EN
mlx5@src := tar xf /netmap/LINUX/ext-drivers/mlnx-en-5.3-1.0.0.1-ubuntu18.04-x86_64.tgz && tar xf mlnx-en-5.3-1.0.0.1-ubuntu18.04-x86_64/src/MLNX_EN_SRC-5.3-1.0.0.1.tgz && tar xf MLNX_EN_SRC-5.3-1.0.0.1/SOURCES/mlnx-en_5.3.orig.tar.gz && ln -s mlnx-en-5.3 mlx5
mlx5@patch := patches/mellanox--mlx5--5.3
i40e@conf := CONFIG_I40E
i40e@src := tar xf /netmap/LINUX/ext-drivers/i40e-2.4.6.tar.gz && ln -s i40e-2.4.6/src i40e
i40e@patch := patches/intel--i40e--2.4.6
ice@conf := CONFIG_ICE
ice@src := tar xf /netmap/LINUX/ext-drivers/ice-1.7.16.tar.gz && ln -s ice-1.7.16/src ice
ice@patch := patches/intel--ice--1.7.16
ixgbevf@conf := CONFIG_IXGBEVF
ixgbevf@src := tar xf /netmap/LINUX/ext-drivers/ixgbevf-4.3.2.tar.gz && ln -s ixgbevf-4.3.2/src ixgbevf
ixgbevf@patch := patches/intel--ixgbevf--4.3.2
ixgbe@conf := CONFIG_IXGBE
ixgbe@src := tar xf /netmap/LINUX/ext-drivers/ixgbe-5.3.8.tar.gz && ln -s ixgbe-5.3.8/src ixgbe
ixgbe@patch := patches/intel--ixgbe--5.3.8
igb@conf := CONFIG_IGB
igb@src := tar xf /netmap/LINUX/ext-drivers/igb-5.3.5.20.tar.gz && ln -s igb-5.3.5.20/src igb
igb@patch := patches/intel--igb--5.3.5.20
virtio_net.c@conf := CONFIG_VIRTIO_NET
virtio_net.c@src := mkdir -p virtio_net.c && cp /netmap/LINUX/ext-drivers/virtio_net.c virtio_net.c
virtio_net.c@patch := patches/custom--virtio_net.c--4.9
CSDN @Long_xu
```

```

cyb@DESKTOP-U0IAA21:~/UserProtocolStack/netmap/LINUX$ ./configure
*****
*** NOTE ****
*** Running some preliminary tests to customize the build environment.
*****
*** NOTE ****
*** We are trying to download the original sources for driver
*** mlx5 using the following command:
*** test -e /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz || wget https://content.mellanox.com/ofed/MLNX_E_N-5.8-3.0.7.0/MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz -P /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers
*** If this fails, please download the above file and put it
*** in /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/, then run configure again.
*****
--2024-04-16 21:26:14-- https://content.mellanox.com/ofed/MLNX_EN-5.8-3.0.7.0/MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz
Resolving content.mellanox.com (content.mellanox.com)... 197.178.241.102
Connecting to content.mellanox.com (content.mellanox.com)|197.178.241.102|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7767490 (7.4M) [application/x-tar]
Saving to: '/home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz'

MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz 100%[=====] 7.41M 2.11MB/s in 4.5s

2024-04-16 21:26:19 (1.66 MB/s) - '/home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/MLNX_EN_SRC-debian-5.8-3.0.7.0.tgz' saved [7767490/7767490]

*****
*** NOTE ****
*** We are trying to download the original sources for driver
*** i40e using the following command:
*** test -e /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/i40e-2.19.3.tar.gz || wget https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz -P /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/
*** If this fails, please download the above file and put it
*** in /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/, then run configure again.
*****
--2024-04-16 21:26:19-- https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz
Resolving sourceforge.net (sourceforge.net)... 172.64.150.145, 104.18.37.111, 2606:4700:4400::ac40:9691, ...
Connecting to sourceforge.net (sourceforge.net)|172.64.150.145|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz/ [following]
--2024-04-16 21:26:20-- https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz/
Reusing existing connection to sourceforge.net:443.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz/download [following]
--2024-04-16 21:26:20-- https://sourceforge.net/projects/e1000/files/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz/download
Reusing existing connection to sourceforge.net:443.
HTTP request sent, awaiting response... 302 Found
Location: https://downloads.sourceforge.net/project/e1000/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz?ts=gAAAAABmHnx-6tDFGWLhgEk0Mld82A65Ga_c1SHFKC9GNLNPmQTbSVUD9-rAezsfqvndSLGr3sOoUyobSsiMV-zcq6GLhv3Iw%3D&use_mirror=nchc&r= [following]
--2024-04-16 21:26:20-- https://downloads.sourceforge.net/project/e1000/i40e%20stable/2.19.3/i40e-2.19.3.tar.gz?ts=gAAAAABmHnx-6tDFGWLhgEk0Mld82A65Ga_c1SHFKC9GNLNPmQTbSVUD9-rAezsfqvndSLGr3sOoUyobSsiMV-zcq6GLhv3Iw%3D&use_mirror=nchc&r=
Resolving downloads.sourceforge.net (downloads.sourceforge.net)... 204.68.111.105
Connecting to downloads.sourceforge.net (downloads.sourceforge.net)|204.68.111.105|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 686306 (670K) [application/x-gzip]
Saving to: '/home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/i40e-2.19.3.tar.gz'

i40e-2.19.3.tar.gz 100%[=====] 1.67M 8.82KB/s in 62s

```

## (7) 编译和安装:

```
make && make install
```

此过程也需要耐心等待一段时间，过程有点久。

```

.....
##install -D -m 644 ice.7.gz //usr/share/man/man7/ice.7.gz
/sbin/depmod -e -F /boot/System.map-4.15.0-142-generic -a
4.15.0-142-generic
Updating initramfs...
update-initramfs -u
update-initramfs: Generating /boot/initrd.img-4.15.0-142-
generic
make[1]: Leaving directory '/netmap/LINUX/ice-1.7.16/src'
```

```
make -C ixgbe install INSTALL_MOD_PATH= CFLAGS_EXTRA="-Wno-unused-but-set-variable -Wno-attributes -Wno-maybe-uninitialized -Wno-unused-variable -Wno-unused-label -I/netmap/LINUX -I/netmap/LINUX -I/netmap/LINUX/..sys -I/netmap/LINUX/..sys/dev -DCONFIG_NETMAP -Wno-unused-but-set-variable -g -DCONFIG_NETMAP_NULL -DCONFIG_NETMAP_PTNETMAP -DCONFIG_NETMAP_GENERIC -DCONFIG_NETMAP_MONITOR -DCONFIG_NETMAP_PIPE -DCONFIG_NETMAP_VALE" NETMAP_DRIVER_SUFFIX=KSRC=/lib/modules/4.15.0-142-generic/build KBUILD_EXTRA_SYMBOLS=/netmap/LINUX/Module.symvers
make[1]: Entering directory '/netmap/LINUX/ixgbe-5.3.8/src'
make[2]: Entering directory '/usr/src/linux-headers-4.15.0-142-generic'
        Building modules, stage 2.
        MODPOST 1 modules
make[2]: Leaving directory '/usr/src/linux-headers-4.15.0-142-generic'
Copying manpages...
Installing modules...
make[2]: Entering directory '/usr/src/linux-headers-4.15.0-142-generic'
        INSTALL /netmap/LINUX/ixgbe-5.3.8/src/ixgbe.ko
At main.c:160:
- SSL error:02001002:system library:fopen:No such file or directory: bss_file.c:175
- SSL error:2006D080:BIO routines:BIO_new_file:no such file: bss_file.c:178
sign-file: certs/signing_key.pem: No such file or directory
        DEPMOD 4.15.0-142-generic
make[2]: Leaving directory '/usr/src/linux-headers-4.15.0-142-generic'
Running depmod...
make[1]: Leaving directory '/netmap/LINUX/ixgbe-5.3.8/src'
```

```
make -C igb install INSTALL_MOD_PATH= CFLAGS_EXTRA="-DDISABLE_PACKET_SPLIT -fno-pie -I/netmap/LINUX -I/netmap/LINUX -I/netmap/LINUX/..sys -I/netmap/LINUX/..sys/dev -DCONFIG_NETMAP -Wno-unused-but-set-variable -g -DCONFIG_NETMAP_NULL -DCONFIG_NETMAP_PTNETMAP -DCONFIG_NETMAP_GENERIC -DCONFIG_NETMAP_MONITOR -DCONFIG_NETMAP_PIPE -DCONFIG_NETMAP_VALE" NETMAP_DRIVER_SUFFIX=KSRC=/lib/modules/4.15.0-142-generic/build KBUILD_EXTRA_SYMBOLS=/netmap/LINUX/Module.symvers
make[1]: Entering directory '/netmap/LINUX/igb-5.3.5.20/src'
make[2]: Entering directory '/usr/src/linux-headers-4.15.0-142-generic'
Building modules, stage 2.
MODPOST 1 modules
make[2]: Leaving directory '/usr/src/linux-headers-4.15.0-142-generic'
Copying manpages...
Installing modules...
make[2]: Entering directory '/usr/src/linux-headers-4.15.0-142-generic'
    INSTALL /netmap/LINUX/igb-5.3.5.20/src/igb.ko
At main.c:160:
- SSL error:02001002:system library:fopen:No such file or
directory: bss_file.c:175
- SSL error:2006D080:BIO routines:BIO_new_file:no such
file: bss_file.c:178
sign-file: certs/signing_key.pem: No such file or
directory
DEPMOD 4.15.0-142-generic
make[2]: Leaving directory '/usr/src/linux-headers-4.15.0-142-generic'
Running depmod...
make[1]: Leaving directory '/netmap/LINUX/igb-5.3.5.20/src'
```

```
make -C virtio_net.c install INSTALL_MOD_PATH=
EXTRA_CFLAGS="-I/netmap/LINUX -I/netmap/LINUX -
I/netmap/LINUX/..sys -I/netmap/LINUX/..sys/dev -
DCONFIG_NETMAP -Wno-unused-but-set-variable -g -
DCONFIG_NETMAP_NULL -DCONFIG_NETMAP_PTNETMAP -
DCONFIG_NETMAP_GENERIC -DCONFIG_NETMAP_MONITOR -
DCONFIG_NETMAP_PIPE -DCONFIG_NETMAP_VALE"
NETMAP_DRIVER_SUFFIX= KSRC=/lib/modules/4.15.0-142-
generic/build
make[1]: Entering directory '/netmap/LINUX/virtio_net.c'
make -C "/lib/modules/4.15.0-142-generic/build"
M=/netmap/LINUX/virtio_net.c modules_install
make[2]: Entering directory '/usr/src/linux-headers-
4.15.0-142-generic'
    INSTALL /netmap/LINUX/virtio_net.c/virtio_net.ko
At main.c:160:
- SSL error:02001002:system library:fopen:No such file or
directory: bss_file.c:175
- SSL error:2006D080:BIO routines:BIO_new_file:no such
file: bss_file.c:178
sign-file: certs/signing_key.pem: No such file or
directory
    DEPMOD 4.15.0-142-generic
make[2]: Leaving directory '/usr/src/linux-headers-4.15.0-
142-generic'
make[1]: Leaving directory '/netmap/LINUX/virtio_net.c'
make -C build-apps/dedup install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
apps/dedup'
install -D dedup //usr/local/bin/dedup
install -D -m 644 /netmap/LINUX/..apps/lb/lb.8
//usr/local/share/man/man8/lb.8
make[1]: Leaving directory '/netmap/LINUX/build-
apps/dedup'
make -C build-apps/vale-ctl install
SRCDIR=/netmap/LINUX/.. BUILDDIR=/netmap/LINUX DESTDIR=""
PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
apps/vale-ctl'
```

```
install -D vale-ctl //usr/local/bin/vale-ctl
install -D -m 644 /netmap/LINUX/../../apps/vale-ctl/vale-
ctl.4 //usr/local/share/man/man4/vale-ctl.4
make[1]: Leaving directory '/netmap/LINUX/build-apps/vale-
ctl'
make -C build-apps/nmreplay install
SRCDIR=/netmap/LINUX/.. BUILDDIR=/netmap/LINUX DESTDIR=""
PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
apps/nmreplay'
install -D nmreplay //usr/local/bin/nmreplay
install -D -m 644
/netmap/LINUX/../../apps/nmreplay/nmreplay.8
//usr/local/share/man/man8/nmreplay.8
make[1]: Leaving directory '/netmap/LINUX/build-
apps/nmreplay'
make -C build-apps/tlem install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
apps/tlem'
install -D tlem //usr/local/bin/tlem
install -D -m 644 /netmap/LINUX/../../apps/tlem/tlem.8
//usr/local/share/man/man8/tlem.8
make[1]: Leaving directory '/netmap/LINUX/build-apps/tlem'
make -C build-apps/lb install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-apps/lb'
install -D lb //usr/local/bin/lb
install -D -m 644 /netmap/LINUX/../../apps/lb/lb.8
//usr/local/share/man/man8/lb.8
make[1]: Leaving directory '/netmap/LINUX/build-apps/lb'
make -C build-apps/bridge install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
apps/bridge'
install -D bridge //usr/local/bin/bridge
install -D -m 644 /netmap/LINUX/../../apps/bridge/bridge.8
//usr/local/share/man/man8/bridge.8
install -D bridge-b //usr/local/bin/bridge-b
```

```
install -D -m 644 /netmap/LINUX/./apps/bridge/bridge.8
//usr/local/share/man/man8/bridge.8
make[1]: Leaving directory '/netmap/LINUX/build-
apps/bridge'
make -C build-apps/pkt-gen install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-apps/pkt-
gen'
install -D pkt-gen //usr/local/bin/pkt-gen
install -D -m 644 /netmap/LINUX/./apps/pkt-gen/pkt-gen.8
//usr/local/share/man/man8/pkt-gen.8
install -D pkt-gen-b //usr/local/bin/pkt-gen-b
install -D -m 644 /netmap/LINUX/./apps/pkt-gen/pkt-gen.8
//usr/local/share/man/man8/pkt-gen.8
make[1]: Leaving directory '/netmap/LINUX/build-apps/pkt-
gen'
install -m 0644 -D /netmap/LINUX/./sys/net/netmap.h
//usr/local/include/net/netmap.h
install -m 0644 -D /netmap/LINUX/./sys/net/netmap_user.h
//usr/local/include/net/netmap_user.h
install -m 0644 -D /netmap/LINUX/./sys/net/netmap_virt.h
//usr/local/include/net/netmap_virt.h
install -m 0644 -D
/netmap/LINUX/./sys/net/netmap_legacy.h
//usr/local/include/net/netmap_legacy.h
install -m 0644 -D /netmap/LINUX/./libnetmap/libnetmap.h
//usr/local/include/libnetmap.h
install -D -m 644 /netmap/LINUX/./share/man/man4/netmap.4
//usr/local/share/man/man4/netmap.4
install -D -m 644 /netmap/LINUX/./share/man/man4/vale.4
//usr/local/share/man/man4/vale.4
install -D -m 644 /netmap/LINUX/./share/man/man4/ptnet.4
//usr/local/share/man/man4/ptnet.4
make -C build-libnetmap install SRCDIR=/netmap/LINUX/..
BUILDDIR=/netmap/LINUX DESTDIR="" PREFIX="/usr/local"
make[1]: Entering directory '/netmap/LINUX/build-
libnetmap'
install -D libnetmap.a //usr/local/lib/libnetmap.a
make[1]: Leaving directory '/netmap/LINUX/build-libnetmap'
```

```

cyb@DESKTOP-U0IAA21:~/UserProtocolStack/netmap/LINUX$ make && make install
cp -Rp /lib/modules/5.15.146.1-microsoft-standard-WSL2/build/drivers/net/vmxnet3 vmxnet3
patch --quiet --force -p1 < patches/vanilla--vmxnet3--40d00--60600;
mv vmxnet3/Makefile vmxnet3/orig.mak || mv vmxnet3/Kbuild vmxnet3/orig.mak; cp drv-subdir.mak vmxnet3/Makefile
touch get-vmxnet3
test -e /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/ixgbefv-4.15.1.tar.gz || wget https://sourceforge.net/projects/e1000/files/ixgbefv%20stable/4.15.1/ixgbefv-4.15.1.tar.gz -P /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/
tar xf /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/ixgbefv-4.15.1.tar.gz && ln -s ixgbefv-4.15.1/src ixgbefv
patch --quiet --force -p1 < patches/intel--ixgbefv-4.15.1;
/home/cyb/UserProtocolStack/netmap/LINUX/intel-fix.sh ixgbefv
patching file common.mk
Hunk #1 succeeded at 199 (offset 24 lines).
touch get-ixgbefv
cp -Rp /lib/modules/5.15.146.1-microsoft-standard-WSL2/build/drivers/net/ethernet/intel/igc igc
patch --quiet --force -p1 < patches/vanilla--igc--50600--99999;
mv igc/Makefile igc/orig.mak || mv igc/Kbuild igc/orig.mak; cp drv-subdir.mak igc/Makefile
touch get-igc
test -e /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/igb-5.10.2.tar.gz || wget https://sourceforge.net/projects/e1000/files/igb%20stable/5.10.2/igb-5.10.2.tar.gz -P /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/
tar xf /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/igb-5.10.2.tar.gz && ln -s igb-5.10.2/src igb
patch --quiet --force -p1 < patches/intel--igb-5.10.2;
/home/cyb/UserProtocolStack/netmap/LINUX/intel-fix.sh igb
patching file common.mk
Hunk #1 succeeded at 199 (offset 24 lines).
touch get-igb
cp -Rp /lib/modules/5.15.146.1-microsoft-standard-WSL2/build/drivers/net/ethernet/intel/e1000 e1000
patch --quiet --force -p1 < patches/vanilla--e1000--20620--99999;
mv e1000/Makefile e1000/orig.mak || mv e1000/Kbuild e1000/orig.mak; cp drv-subdir.mak e1000/Makefile
touch get-e1000
cp -Rp /lib/modules/5.15.146.1-microsoft-standard-WSL2/build/drivers/net/veth.c veth.c
patch --quiet --force -p1 < patches/vanilla--veth.c--41400--60700;
touch get-veth.c
cp -Rp /lib/modules/5.15.146.1-microsoft-standard-WSL2/build/drivers/net/ethernet/nvidia/forcedeth.c forcedeth.c
patch --quiet --force -p1 < patches/vanilla--forcedeth.c--20626--99999;
touch get-forcedeth.c
test -e /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/virtio_net.c || wget https://raw.githubusercontent.com/torvalds/linux/v4.9/drivers/net/virtio_net.c -P /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/
mkdir -p virtio_net.c && cp /home/cyb/UserProtocolStack/netmap/LINUX/ext-drivers/virtio_net.c virtio_net.c/
patch --quiet --force -p1 < patches/custom--virtio_net.c--4.9;
touch get-virtio_net.c
make -C /lib/modules/5.15.146.1-microsoft-standard-WSL2/build M=/home/cyb/UserProtocolStack/netmap/LINUX EXTRA_CFLAGS=' -I /home/cyb/UserProtocolStack/n etmap/LINUX -I /home/cyb/UserProtocolStack/netmap/LINUX -I /home/cyb/UserProtocolStack/netmap/LINUX .. /sys /dev -D CONFIG_NETMAP -Wno-unused-but-set-variable -Wno-attributes -Wno-packed-not-aligned -Wno-stringop-truncation -Wno-missing-attributes -Wno-format-truncation -Wno-maybe-uninitialized -Wno-unused-variable -Wno-unused-label -Wno-implicit-fallthrough -Wno-missing-prototypes -g -D CONFIG_NETMAP_N ULL -D CONFIG_NETMAP_PTNETMAP -D CONFIG_NETMAP_GENERIC -D CONFIG_NETMAP_MONITOR -D CONFIG_NETMAP_PIPE -D CONFIG_NETMAP_VALE' modules CONFIG_NETMAP=m CONFIG_G_FORCEDETH=m CONFIG_VETH=m CONFIG_E1000=m CONFIG_IGC=m CONFIG_VMXNET3=m O_DRIVERS="e1000/ forcedeth.o igc/ veth.o vmxnet3/" NETMAP_DRIVER_SUFFIX=
make[1]: Entering directory '/home/cyb/WSL2-Linux-Kernel-linux-msft-wsl-5.15.146.1'
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000_main.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000_hw.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000_ethtool.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000_param.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_main.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_mac.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_i225.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_base.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_nvmm.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_phys.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_diag.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_ethtool.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_ptp.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_dump.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_tsn.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc_xdp.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/vmxnet3Drv.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/vmxnet3/vmxnet3_ethtool.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/vmxnets/vmxnet3.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_mem2.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_mbq.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_legacy.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_bdg.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_kloop.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_vale.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_ofloadings.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_pipe.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_monitor.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_generic.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_null.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_common.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_linux.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap_ptnet.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/netmap.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/forcedeth.o
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/veth.o
MODPOST /home/cyb/UserProtocolStack/netmap/LINUX/module.symvers
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000/e1000.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/e1000/e1000.ko
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/forcedeth.mod.o
LD [M] /home/cyb/UserProtocolStack/netmap/LINUX/forcedeth.ko
CC [M] /home/cyb/UserProtocolStack/netmap/LINUX/igc/igc.mod.o

```

## (8) 使用netmap:

```
insmod netmap.ko
```

每次使用前都要执行insmod netmap.ko，它在/netmap/LINUX/路径下。

## (9) 检查netmap是否insmod成功:

```
ls /dev/netmap -l
```

出现如下表示成功：

```
crw----- 1 root root 10, 54 8月 31 12:53 /dev/netmap
```

## (10) 编译运行自己的代码

```
# 头文件 #include<net/netmap_user.h> 在 /netmap/sys/目录下  
# 和/usr/local/include/net/目录下  
gcc -o testcode testcode.c -I /netmap/sys/
```

## netmap 安装过程

1、uname -r：仅仅查看内核版本

```
cyb@DESKTOP-UOIAA21:~/UserProtocolStack/netmap/LINUX$ uname -r  
5.15.146.1-microsoft-standard-WSL2
```

2、uname -a：

```
cyb@DESKTOP-UOIAA21:~/UserProtocolStack/netmap/LINUX$ uname -a  
Linux DESKTOP-UOIAA21 5.15.146.1-microsoft-standard-WSL2 #1 SMP Thu Jan 11 04:09:03 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
```

3、查看内核版本号：cat /proc/version

```
cyb@DESKTOP-UOIAA21:~/UserProtocolStack/netmap/LINUX$ cat /proc/version  
Linux version 5.15.146.1-microsoft-standard-WSL2 (root@65c757a075e2) (gcc (GCC) 11.2.0, GNU ld (GNU Binutils) 2.37) #1 SMP Thu Jan 11 04:09:03 UTC 2024
```

在windows中使用 wsl -l -v 查看所安装的WSL系统版本

在Linux系统里，要进行系统调用开发，往往是需要安装内核的headers，才能找到所需要的C语言headers文件。

一般来说，安装内核headers，在ubuntu里只需要执行：sudo apt-get install linux-headers-\$(uname -r)，但是在WSL里，这样子是不行的，因为WSL安装的是微软特供版，需要用对应版本的headers，apt-get安装不到。

照着网上使劲折腾的时候，发现我一直升级不成功，最终才发现原来我一直在使用在WSL 1，而不是最新的WSL2。

在Windows的powershell或者cmd里，执行命令：wsl -l -v，可以看到如下内容：

```
C:\Users\shahuwang\workspace\media>wsl -l -v  
NAME STATE VERSION  
* Ubuntu-20.04 Running 2  
docker-desktop-data Running 2  
docker-desktop Running 2 知乎 @王瑞期
```

这表明我在wsl里安装的三个系统，都是使用的wsl2，如果有一个的version=1，就表示这个使用的是wsl 1，可以用如下的命令进行转换：

```
wsl --set-version ubuntu-20.04 2
```

## 安装netmap，报错：

```
cyb@DESKTOP-UOIAA21:~/UserProtocolStack/netmap/LINUX$ ./configure
*****
*** Cannot find kernel directory.
*** We need at least the kernel headers to compile the netmap kernel module.
*** If your kernel headers are not in the standard place, please provide the
*** correct path using the
*** --kernel-dir=/path/to/kernel/dir
*** option.
*** Otherwise, check that the 'build' symlink in
***      /lib/modules/$(uname -r)
*** is not broken.
*** Current configuration values:
*** kernel directory      /lib/modules/5.15.146.1-microsoft-standard-WSL2/build
*** kernel sources        [/usr/lib/modules/5.15.146.1-microsoft-standard-WSL2/build]
*** linux version         [/usr/lib/modules/5.15.146.1-microsoft-standard-WSL2/build]
*** module file           netmap.ko
*** subsystems            null ptnetmap generic monitor pipe vale
*** apps                  dedup vale-ctl nmreplay titem lb bridge pkt-gen
*** native drivers         mlx5 vmxnet3 i40e ice ixgbevf ixgbe igc igb e1000e e1000 veth.c forcedeth.c virtio_net.c r8169.c stmmac
*****
*****
```

解决办法：按照这篇教程：WSL升级到最新版本Linux内核headers的方法（[WSL升级到最新版本Linux内核headers的方法 - 知乎\(zhihu.com\)](#)）安装linux内核。之后成功

报错：CONFIG\_X86\_X32 enabled but no binutils support

解决方法：按照这篇教程安装：[arch/x86/Makefile:142: CONFIG\\_X86\\_X32 enabled but no binutils support-CSDN博客](#)

如何还不能解决，那么找到wsl2的内核的.config文件，注释掉：  
CONFIG\_X86\_X32

## 4.5、协议栈实现代码示例

示例简单实现了arp、icmp、udp的协议栈；其他协议的实现类似。

```
//需要开启netmap的宏
#define NETMAP_WITH_LIBS

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <net/netmap_user.h>
#include <sys/poll.h>
#include <arpa/inet.h>
```

```
#pragma pack(1) //设置一字节对齐方式

#define ETH_ALEN          6
#define PROTO_IP           0x0800 //      IP 协议
#define PROTO_ARP          0x0806

#define PROTOCOL_UDP       17
#define PROTO_ICMP          1
#define PROTO_IGMP          2

#define ICMP_TYPE_ANS       0
#define ICMP_TYPE_REQ        8

#define ETHER_ADDR_LEN       6

#define MY_IP                "192.168.7.146"
#define MY_MAC               "00:0c:29:39:a8:c4"
// ether
struct etherhdr {
    unsigned char dst_mac[ETHER_ADDR_LEN];
    unsigned char src_mac[ETHER_ADDR_LEN];
    unsigned short protocol;
};

// IP
struct iphdr {
    unsigned char version : 4,
                  hdrlen : 4;
    unsigned char tos;
    unsigned short totlen;
    unsigned short id;
    unsigned short flag : 3,
                  offset : 13;
    unsigned char ttl;
    unsigned char protocol;
    unsigned short check;
    unsigned int   sip;
    unsigned int   dip;
};
```

```
// UDP
struct udphdr {
    unsigned short sport;
    unsigned short dport;
    unsigned short length;
    unsigned short check;
};

struct udppkt {
    struct etherhdr eth;
    struct iphdr ip;
    struct udphdr udp;
    unsigned char payload[0];// 零长数组
};

// ARP
struct arphdr{
    unsigned short h_type;
    unsigned short h_proto;
    unsigned char h_addrlen;
    unsigned char protolen;
    unsigned short oper;
    unsigned char smac[ETH_ALEN];
    unsigned int sip;
    unsigned char dmac[ETH_ALEN];
    unsigned int dip;
};

struct arppkt {
    struct etherhdr eth;
    struct arphdr arp;
};

// ICMP
struct icmpHdr {
    unsigned char type;
    unsigned char code;
    unsigned short check;
    unsigned short identifier;
}
```

```

    unsigned short sep;
    unsigned char    data[32];
};

struct icmppkt{
    struct etherhdr eth;
    struct iphdr ip;
    struct icmphdr icmp;
};

void echo_udp_pkt(struct udppkt *udp,struct udppkt
*udp_rt)
{
    memcpy(udp_rt, udp, sizeof(struct udppkt));
    memcpy(udp_rt->eth.dst_mac, udp->eth.src_mac,
ETH_ALEN);
    memcpy(udp_rt->eth.src_mac, udp->eth.dst_mac,
ETH_ALEN);

    udp_rt->ip.sip = udp->ip.dip;
    udp_rt->ip.dip = udp->ip.sip;

    udp_rt->udp.sport = udp->udp.dport;
    udp_rt->udp.dport = udp->udp.sport;
}

unsigned short in_cksum(unsigned short *addr,int len)
{
    register int nleft = len;
    register unsigned short *w = addr;
    register int sum = 0;//32bit
    unsigned short answer = 0;//16bit

    while (nleft > 1)
    {
        sum += *w++;//16bit为一组累加
        nleft -= 2;
    }
}

```

```
    if (nleft == 1)//存在单个byte情况
    {
        *(u_char*)&answer) = *(u_char*)w;
        sum += answer;
    }
    sum = (sum >> 16) + (sum & 0xffff); // 高16bit与低16bit
相加
    sum += (sum >> 16); //防止值大于0xffff

    //结果
    answer = ~sum;

    return (answer);
}

void echo_icmp_pkt(struct icmppkt *icmp, struct icmppkt
*icmp_rt)
{
    memcpy(icmp_rt, icmp, sizeof(struct icmppkt));

    memcpy(icmp_rt->eth.dst_mac, icmp->eth.src_mac,
ETH_ALEN);
    memcpy(icmp_rt->eth.src_mac, icmp->eth.dst_mac,
ETH_ALEN);

    icmp_rt->icmp.type = ICMP_TYPE_ANS;
    icmp_rt->icmp.code = 0;
    icmp_rt->icmp.check = 0;

    icmp_rt->ip.sip = icmp->ip.dip;
    icmp_rt->ip.dip = icmp->ip.sip;

    icmp_rt->icmp.check = in_cksum((unsigned
short*)&icmp_rt->icmp, sizeof(struct icmp));
}

int str2mac(char *mac, char *str) {
    char *p = str;
    unsigned char value = 0x0;
```

```

int i = 0;
while (p != '\0') {
    if (*p == ':') {
        mac[i++] = value;
        value = 0x0;
    }
    else {
        unsigned char temp = *p;
        if (temp <= '9' && temp >= '0') {
            temp -= '0';
        }
        else if (temp <= 'f' && temp >= 'a') {
            temp -= 'a';
            temp += 10;
        }
        else if (temp <= 'F' && temp >= 'A') {
            temp -= 'A';
            temp += 10;
        }
        else {
            break;
        }
        value <<= 4;
        value |= temp;
    }
    p++;
}

mac[i] = value;

return 0;
}

void echo_arp_pkt(struct arppkt *arp, struct arppkt
*arp_rt, char *hmac) {
    memcpy(arp_rt, arp, sizeof(struct arppkt));

    memcpy(arp_rt->eth.dst_mac, arp->eth.src_mac,
ETH_ALEN);
    str2mac(arp_rt->eth.src_mac, hmac);
}

```

```
arp_rt->eth.protocol = arp->eth.protocol;

arp_rt->arp.h_addrlen = 6;
arp_rt->arp.protolen = 4;
arp_rt->arp.oper = htons(2);

str2mac(arp_rt->arp.smac, hmac);
arp_rt->arp.sip = arp->arp.dip;

memcpy(arp_rt->arp.dmac, arp->arp.smac, ETH_ALEN);
arp_rt->arp.dip = arp->arp.sip;
}

// netmap

int main()
{
printf("length = %ld\n", sizeof(struct etherhdr));

    struct pollfd pfd = {0}; // poll
    struct nm_pkthdr h;
    struct etherhdr *eh;
    // 打开/dev/netmap, 映射网卡数据到内存空间
    struct nm_desc *nmr = nm_open("netmap:eth0",
NULL, 0, NULL);

    if (nmr == NULL)
    {
        printf("netmap open fail!\n");
        return -1;
    }

    pfd.fd = nmr->fd; // 指向/dev/netmap
    pfd.events = POLLIN; // 监听读事件

    while (1)
    {
        int ret = poll(&pfd, 1, -1);
        if (ret < 0)
            continue;
        else
            process_packet(nmr, h); // 处理接收到的数据包
    }
}
```

```

if (pfds.events & POLLIN) //操作内存
{
    unsigned char *stream = nm_nextpkt(nmr, &h); //从环形队列中取出一个数据包
    eh = (struct etherhdr *)stream;
    //将网络数据转换为本地字节序
    if ( ntohs(eh->protocol) == PROTO_IP)
    {
        struct udppkt *pkt = (struct udppkt
*)stream;
        if (pkt->ip.protocol == PROTOCOL_UDP)
        {
            struct in_addr addr;
            addr.s_addr = pkt->ip.sip;
            // udp包length字段表示的是整个UDP包的总长度
            // (包含udp的头长度)
            int length = ntohs(pkt->udp.length);
            printf("%s:%d:length:%d, ip
length:%d\n",
                   inet_ntoa(addr),
                   pkt->udp.sport,
                   length,
                   ntohs(pkt->ip.totlen));
            pkt->payload[length - 8] = '\0';
            printf("pkt: %s\n", pkt->payload);

            struct udppkt udp_rt;
            echo_udp_pkt(pkt, &udp_rt);
            nm_inject(nmr, &udp_rt, sizeof(struct
udppkt));
        }
    }
    else if (pkt->ip.protocol == PROTO_ICMP)
    {
        struct icmppkt *icmp = (struct
icmppkt*)stream;
        printf("icmp-----> %d,%x\n",
               icmp->icmp.type, icmp->icmp.check);
    }
}

```

```

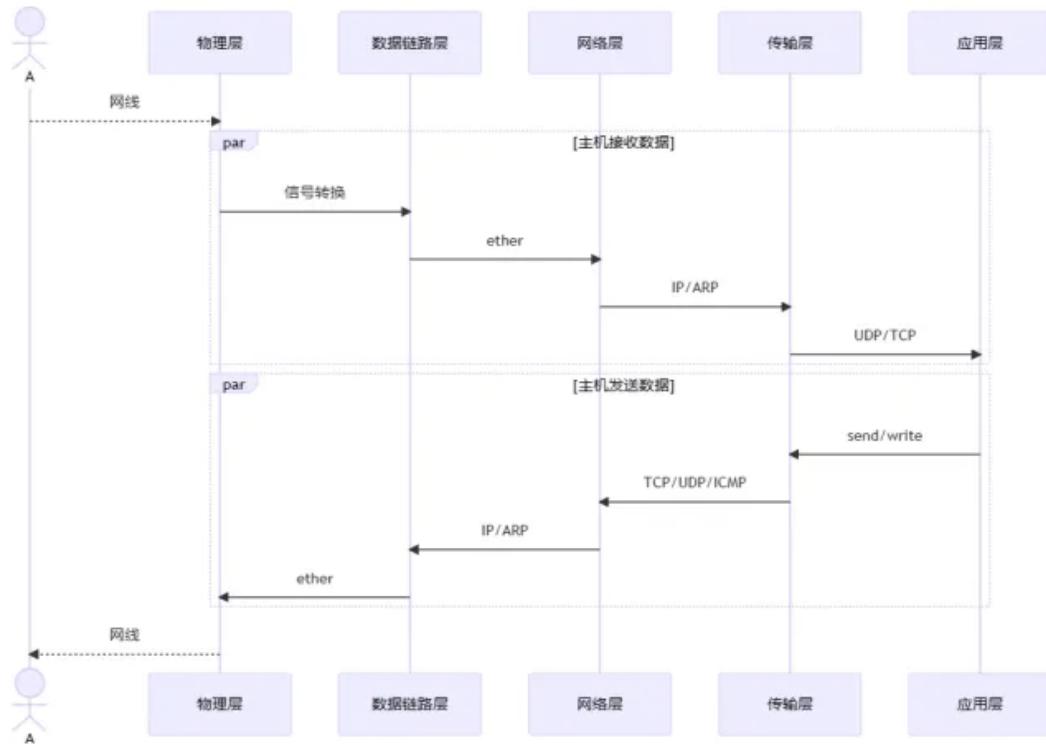
        if (icmp->icmp.type ==
ICMP_TYPE_REQ)//0 代表应答 ICMP 报文、8 代表请求 ICMP 报文。
{
    struct icmppkt icmp_rt = {0};
    echo_icmp_pkt(icmp, &icmp_rt);
    nm_inject(nmr, &icmp_rt,
sizeof(struct icmppkt));
}
}
else if (pkt->ip.protocol == PROTO_IGMP)
{
    printf("PROTO_IGMP packet\n");
}
else
{
    printf("other ip packet\n");
}
}
else if (ntohs(eh->protocol) == PROTO_ARP)
{
    struct arppkt *arp = (struct
arppkt*)stream;
    struct arppkt arp_rt;
    if (arp->arp.dip == inet_addr(MY_IP))
    {
        echo_arp_pkt(arp, &arp_rt, MY_MAC);
        nm_inject(nmr, &arp_rt, sizeof(struct
arppkt));
    }
}
}

return 0;
}

```

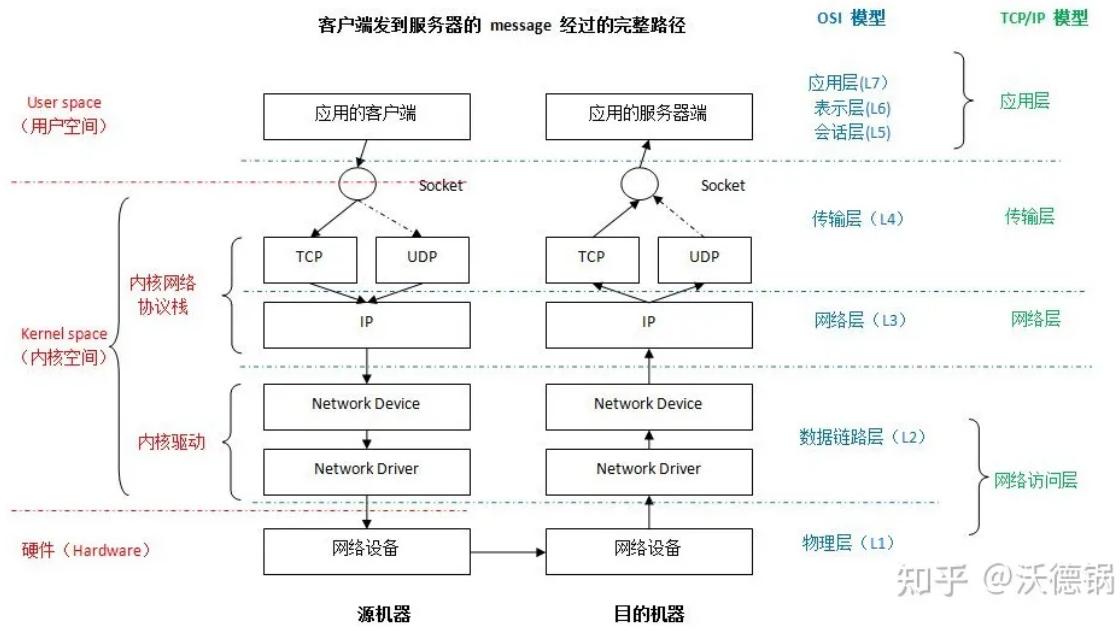
# 总结

要实现一个协议栈，需要清楚七层网络模型，熟悉协议标准；获得协议的原始数据包后需要一层层的拨开解析；发送数据之前需要将协议一层层的往下包装。



# 深度解析Linux网络流程及sk\_buff数据结构

## 1. Linux 网络



## 1.1 发送端

### 1.1.1 应用层

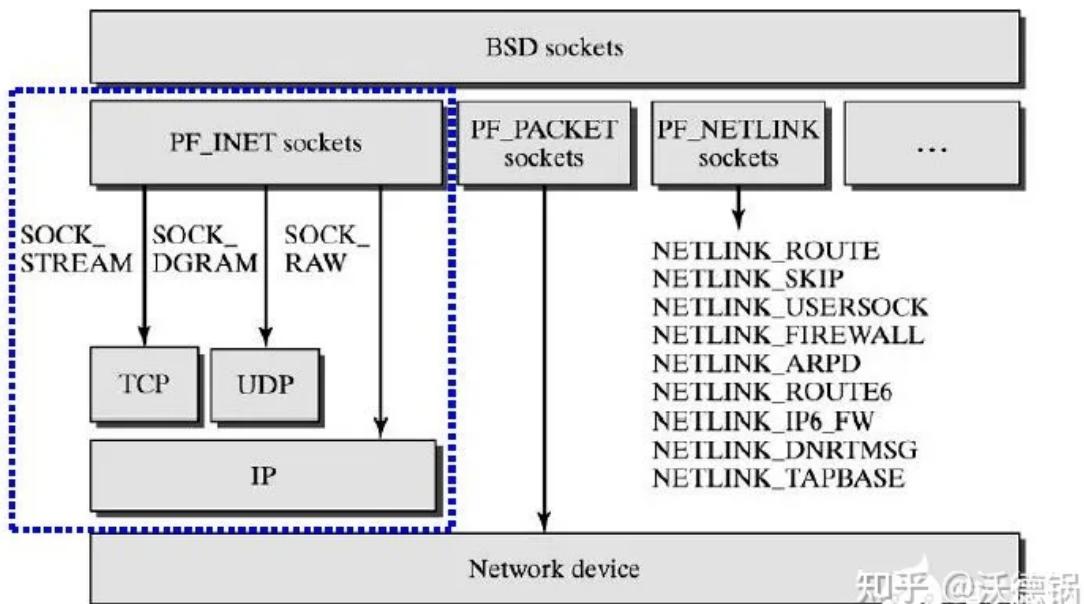
#### (1) Socket

应用层的各种网络应用程序基本上都是通过 Linux Socket 编程接口来和内核空间的网络协议栈通信的。Linux Socket 是从 BSD Socket 发展而来的，它是 Linux 操作系统的重要组成部分之一，它是网络应用程序的基础。从层次上来说，它位于应用层，是操作系统为程序员提供的 API，通过它，应用程序可以访问传输层协议。

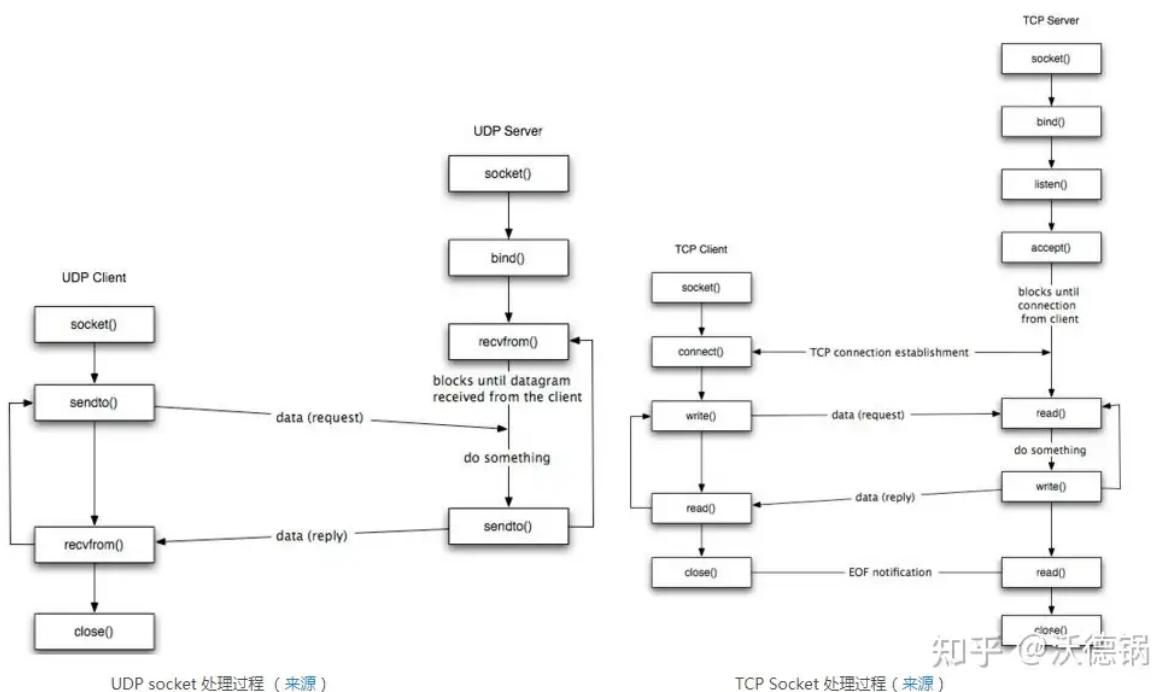
socket 位于传输层协议之上，屏蔽了不同网络协议之间的差异

socket 是网络编程的入口，它提供了大量的系统调用，构成了网络程序的主体

在Linux系统中，socket 属于文件系统的一部分，网络通信可以被看作是对文件的读取，使得我们对网络的控制和对文件的控制一样方便。



知乎 @沃德锅



## (2) 应用层处理流程

网络应用调用Socket API `socket (int family, int type, int protocol)` 创建一个 socket，该调用最终会调用 Linux system call `socket()`，并最终调用 Linux Kernel 的 `sock_create()` 方法。该方法返回被创建好了的那个 socket 的 file descriptor。对于每一个 user space 网络应用创建的 socket，在内核中都有一个对应的 `struct socket` 和 `struct sock`。其中，`struct sock` 有三个队列 (queue)，分别是 rx, tx 和 err，在 sock 结构被初始化的时候，这些缓冲队列也被初始化完成；在收据收发过程中，每个 queue 中保存要发送或者接受的每个 packet 对应的 Linux 网络栈 `sk_buffer` 数据结构的实例 skb。

对于 TCP socket 来说，应用调用 `connect()` API，使得客户端和服务器端通过该 socket 建立一个虚拟连接。在此过程中，TCP 协议栈通过三次握手会建立 TCP 连接。默认地，该 API 会等到 TCP 握手完成连接建立后才返回。在建立连接的过程中的一个重要步骤是，确定双方使用的 Maximum Segement Size (MSS)。因为 UDP 是面向无连接的协议，因此它是不需要该步骤的。

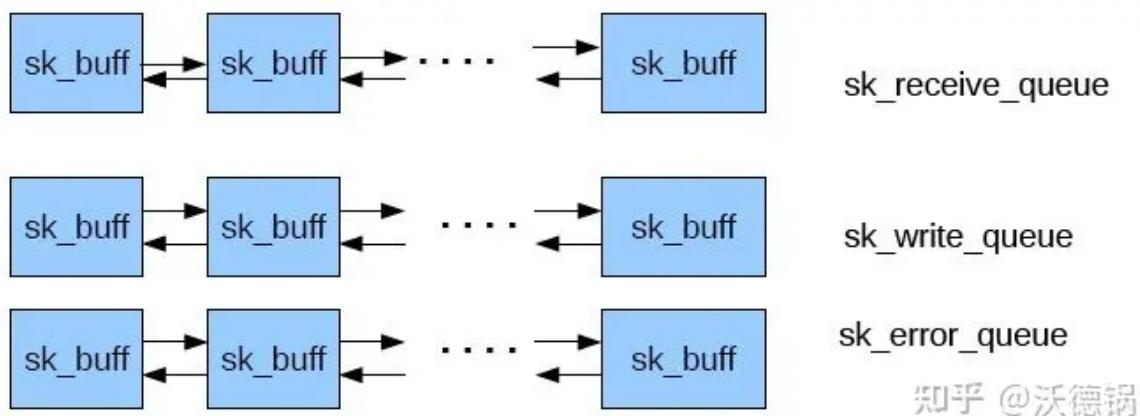
应用调用 Linux Socket 的 `send` 或者 `write` API 来发出一个 message 给接收端

`sock_sendmsg` 被调用，它使用 `socket descriptor` 获取 `sock struct`，创建 message header 和 socket control message

`_sock_sendmsg` 被调用，根据 socket 的协议类型，调用相应协议的发送函数。

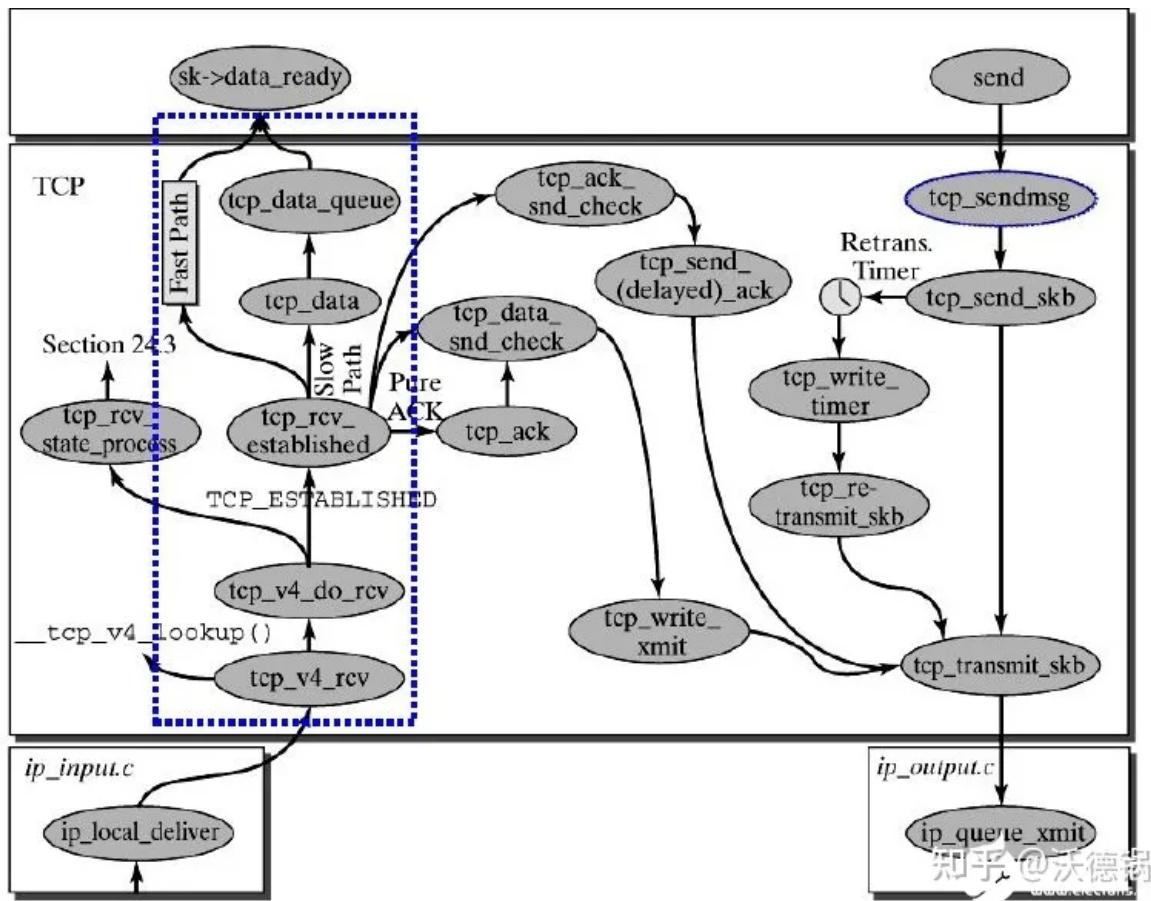
对于 TCP，调用 `tcp_sendmsg` 函数。

对于 UDP 来说，userspace 应用可以调用 `send()`/`sendto()`/`sendmsg()` 三个 system call 中的任意一个来发送 UDP message，它们最终都会调用内核中的 `udp_sendmsg()` 函数。



### 1.1.2 传输层

传输层的最终目的是向它的用户提供高效的、可靠的和成本有效的数据传输服务，主要功能包括（1）构造 TCP segment （2）计算 checksum （3）发送回复（ACK）包 （4）滑动窗口（sliding window）等保证可靠性的操作。TCP 协议栈的大致处理过程如下图所示：



TCP 栈简要过程：

`tcp_sendmsg` 函数会首先检查已经建立的 TCP connection 的状态，然后获取该连接的 MSS，开始 segment 发送流程。

构造 TCP 段的 payload：它在内核空间中创建该 packet 的 sk\_buffer 数据结构的实例 skb，从 userspace buffer 中拷贝 packet 的数据到 skb 的 buffer。

构造 TCP header。

计算 TCP 校验和 (checksum) 和 顺序号 (sequence number)。

TCP 校验和是一个端到端的校验和，由发送端计算，然后由接收端验证。其目的是为了发现TCP首部和数据在发送端到接收端之间发生的任何改动。如果接收方检测到校验和有差错，则TCP段会被直接丢弃。TCP校验和覆盖 TCP 首部和 TCP 数据。

TCP的校验和是必需的

发到 IP 层处理：调用 IP handler 句柄 `ip_queue_xmit`，将 skb 传入 IP 处理流程。

UDP 栈简要过程：

UDP 将 message 封装成 UDP 数据报

调用 `ip_append_data()` 方法将 packet 送到 IP 层进行处理。

### 1.1.3 IP 网络层 - 添加header 和 checksum, 路由处理, IP fragmentation

网络层的任务就是选择合适的网间路由和交换结点，确保数据及时传送。

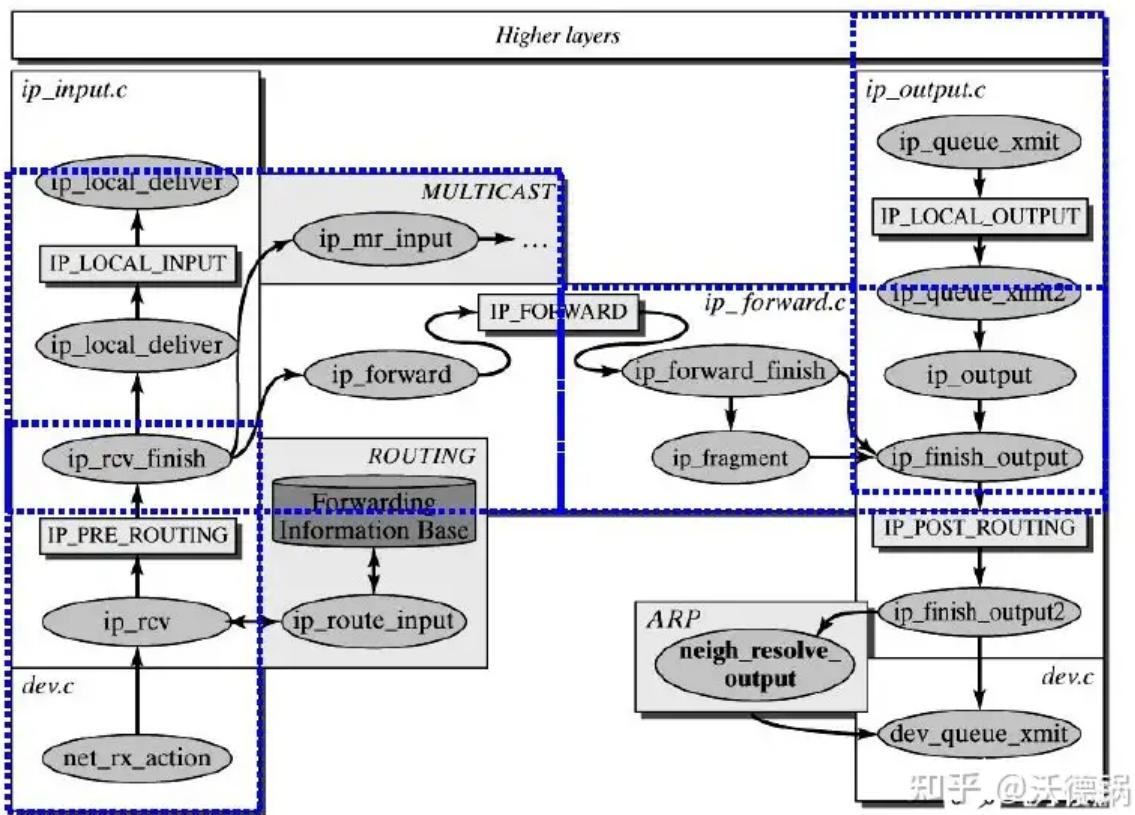
网络层将数据链路层提供的帧组成数据包，包中封装有网络层包头，其中含有逻辑地址信息--源站点和目的站点地址的网络地址。其主要任务包括

(1) 路由处理，即选择下一跳 (2) 添加 IP header (3) 计算 IP header checksum，用于检测 IP 报文头部在传播过程中是否出错 (4) 可能的话，进行 IP 分片 (5) 处理完毕，获取下一跳的 MAC 地址，设置链路层报文头，然后转入链路层处理。

IP 头：

版本	头长度	业务类型(TOS)	总长度 (total length)				
标识符 (Identification)		flags 分段偏移量 (Fragment Offset)					
TTL	协议 (Protocol)	包头校验 (Header Checksum)					
源地址 (Source Address)							
目的地址(Destination Address)							
选项(Options)			填充(Padding)				

IP 栈基本处理过程如下图所示：



首先，`ip_queue_xmit(skb)`会检查 `skb->dst` 路由信息。如果没有，比如套接字的第一个包，就使用 `ip_route_output()` 选择一个路由。

接着，填充IP包的各个字段，比如版本、包头长度、TOS等。

中间的一些分片等，可参阅相关文档。基本思想是，当报文的长度大于 mtu，gso的长度不为0就会调用 `ip_fragment` 进行分片，否则就会调用 `ip_finish_output2` 把数据发送出去。`ip_fragment` 函数中，会检查 IP\_DF 标志位，如果待分片IP数据包禁止分片，则调用 `icmp_send()` 向发送方发送一个原因为需要分片而设置了不分片标志的目的不可达ICMP报文，并丢弃报文，即设置IP状态为分片失败，释放skb，返回消息过长错误码。

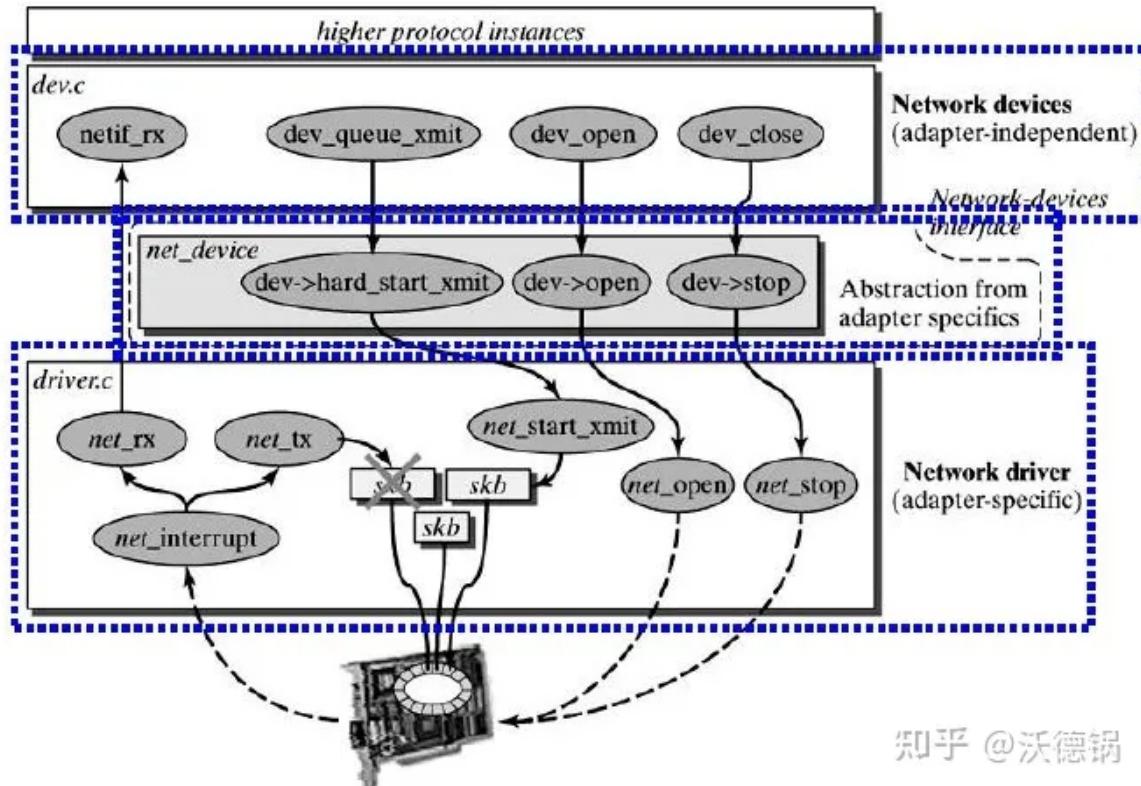
接下来就用 `ip_finish_output2` 设置链路层报文头了。如果，链路层报头缓存有（即hh不为空），那就拷贝到skb里。如果没，那么就调用 `neigh_resolve_output`，使用 ARP 获取。

#### 1.1.4 数据链路层

功能上，在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧（Frame）在信道上无差错的传输，并进行各电路上的动作系列。数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检

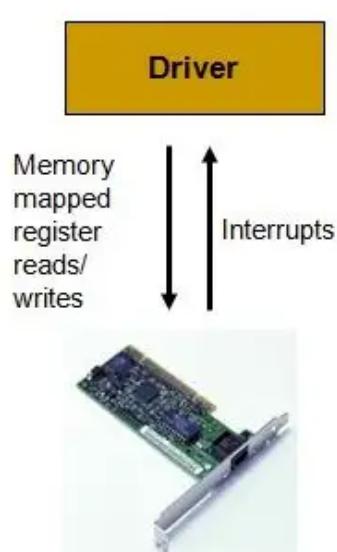
错、重发等。在这一层，数据的单位称为帧（frame）。数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。

实现上，Linux 提供了一个 Network device 的抽象层，其实现在 `linux/net/core/dev.c`。具体的物理网络设备在设备驱动中 (`driver.c`) 需要实现其中的虚函数。Network Device 抽象层调用具体网络设备的函数。



知乎 @沃德锅

### 1.1.5 物理层 - 物理层封装和发送



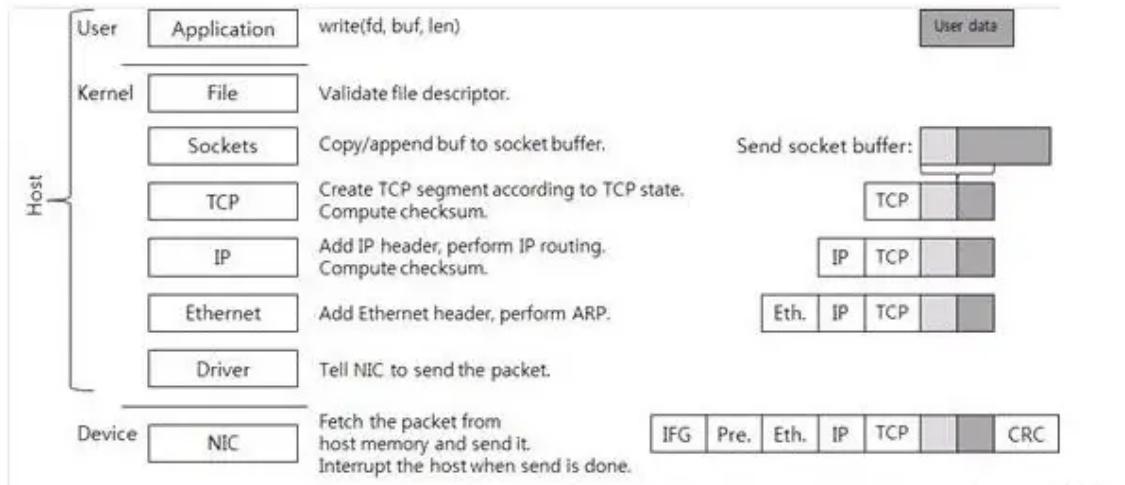
- Driver talks to the device:
  - Writing commands to memory-mapped *control/status registers*
  - Setting aside buffers for packet transmission/reception
  - Describing these buffers in *descriptor rings*
- Device talks to driver:
  - Generating *interrupts* (both on send and receive)
  - Placing values in control status registers
  - DMA'ing packets to/from available buffers
  - Updating status in descriptor rings

知乎 @沃德锅

物理层在收到发送请求之后，通过 DMA 将该主存中的数据拷贝至内部 RAM (buffer) 之中。在数据拷贝中，同时加入符合以太网协议的相关 header, IFG、前导符和CRC。对于以太网网络，物理层发送采用 CSMA/CD,即在发送过程中侦听链路冲突。

一旦网卡完成报文发送，将产生中断通知 CPU，然后驱动层中的中断处理程序就可以删除保存的 skb 了。

### 1.1.6 简单总结



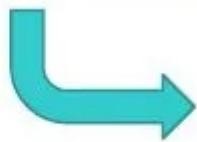
知乎 @欣德纳

## 1.2 接收端

### 1.2.1 物理层和数据链路层

- Network card
  - receives a frame

issues an  
*interrupt*



- Driver

- handles the *interrupt*
  - Frame → RAM
  - Allocates `sk_buff` (called `skb`)
  - Frame → `skb`

- Driver (cont.)

- calls device independent `core/dev.c:netif_rx(skb)`
  - puts `skb` into CPU queue
  - issues a "soft" interrupt

- CPU

- calls `core/dev.c:net_rx_action()`

- removes `skb` from CPU queue
- passes to network layer e.g. ip/arp
- In this case: IPv4 `ipv4/ip_input.c:ip_recv()`

知乎 @沃德锅

## 简要过程：

一个 package 到达机器的物理网络适配器，当它接收到数据帧时，就会触发一个中断，并将通过 DMA 传送到位于 linux kernel 内存中的 rx\_ring。

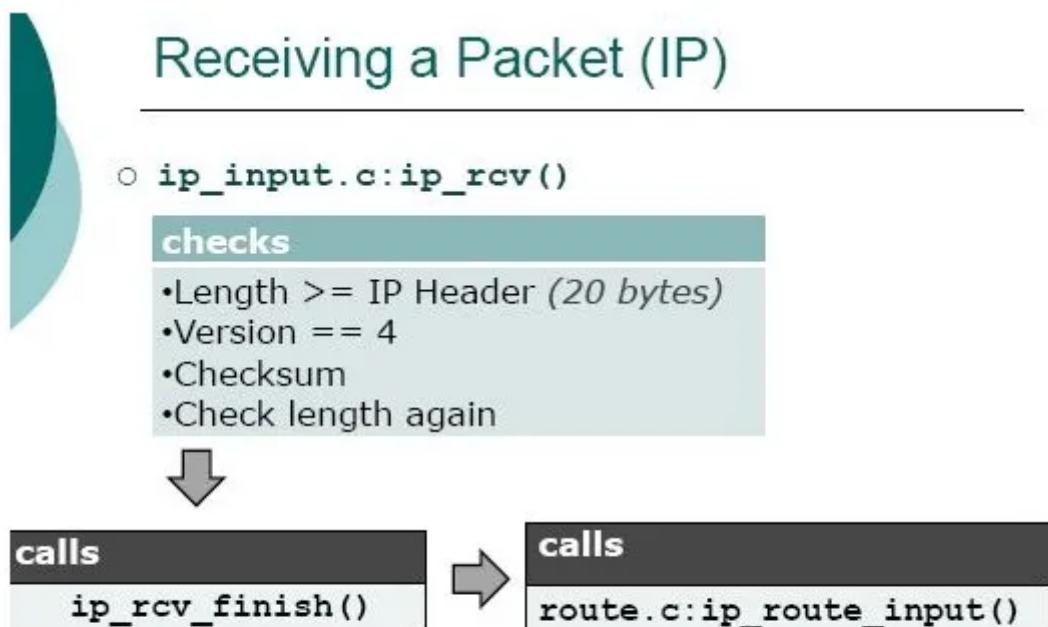
终端处理程序经过简单处理后，发出一个软中断 (`NET_RX_SOFTIRQ`)，通知内核接收到新的数据帧。

内核 2.5 中引入一组新的 API 来处理接收的数据帧，即 NAPI。所以，驱动有两种方式通知内核：(1) 通过以前的函数 `netif_rx`；(2) 通过 NAPI 机制。该中断处理程序调用 Network device 的 `netif_rx_schedule` 函数，进入软中断处理流程，再调用 `net_rx_action` 函数。

该函数关闭中断，获取每个 Network device 的 rx\_ring 中的所有 package，最终 pacakage 从 rx\_ring 中被删除，进入 netif\_receive\_skb 处理流程。

netif\_receive\_skb 是链路层接收数据报的最后一站。它根据注册在全局数组 ptype\_all 和 ptype\_base 里的网络层数 据报类型，把数据报递交给不同的网络层协议的接收函数(INET域中主要是 ip\_rcv 和 arp\_rcv)。该函数主要就是调用第三层协议的接收函数处理该skb包，进入第三层网络层处理。

## 1.2.2 网络层



## Receiving a Packet (routing)

- ipv4/route.c:ip\_route\_input()

Destination == me?	
YES	ip_input.c:ip_local_deliver()
NO	Calls ip_route_input_slow()

- ipv4/route.c:ip\_route\_input\_slow()

Can forward?	
• Forwarding enabled?	
• Know route?	
NO	Sends ICMP

IP 层的入口函数在 `ip_rcv` 函数。该函数首先会做包括 package checksum 在内的各种检查，如果需要的话会做 IP defragment (将多个分片合并)，然后 packet 调用已经注册的 `Pre-routing netfilter hook`，完成后最终到达 `ip_rcv_finish` 函数。

`ip_rcv_finish` 函数会调用 `ip_router_input` 函数，进入路由处理环节。它首先会调用 `ip_route_input` 来更新路由，然后查找 route，决定该 package 将会被发到本机还是会被转发还是丢弃：

如果是发到本机的话，调用 `ip_local_deliver` 函数，可能会做 `defragment` (合并多个 IP packet)，然后调用 `ip_local_deliver` 函数。该函数根据 package 的下一个处理层的 protocol number，调用下一层接口，包括 `tcp_v4_rcv` (TCP)，`udp_rcv` (UDP)，`icmp_rcv` (ICMP)，`igmp_rcv` (IGMP)。对于 TCP 来说，函数 `tcp_v4_rcv` 函数会被调用，从而处理流程进入 TCP 栈。

如果需要转发 (forward)，则进入转发流程。该流程需要处理 TTL，再调用 `dst_input` 函数。该函数会 (1) 处理 `Netfilter Hook` (2) 执行 IP fragmentation (3) 调用 `dev_queue_xmit`，进入链路层处理流程。

## Forwarding a Packet

- Forwarding is per-device basis
  - Receiving device!
- Enable/Disable forwarding in Linux:
  - Kernel
  - /proc file system ↔ Kernel
  - read/write normally (in most cases)

```
* /proc/sys/net/ipv4/conf/<device>/forwarding  
* /proc/sys/net/ipv4/conf/default/forwarding  
* /proc/sys/net/ipv4/ip_forwarding
```

## Forwarding a Packet (cont.)

- `ipv4/ip_forward.c:ip_forward()`

### IP TTL > 1

YES	Decreases TTL
NO	Sends ICMP

- `core/dev.c:dev_queue_xmit()`
- Default queue: priority FIFO
  - `sched/sch_generic.c:pfifo_fast_enqueue()`
- Others: FIFO, Stochastic Fair Queuing, etc. 知乎 @沃德锅

### 1.2.3 传输层 (TCP/UDP)

传输层 TCP 处理入口在 `tcp_v4_rcv` 函数（位于 `linux/net/ipv4/tcp_ipv4.c` 文件中），它会做 TCP header 检查等处理。

调用 `_tcp_v4_lookup`，查找该 package 的 open socket。如果找不到，该 package 会被丢弃。接下来检查 socket 和 connection 的状态。

如果socket 和 connection 一切正常，调用 `tcp_prequeue` 使 package 从内核进入 user space，放进 socket 的 receive queue。然后 socket 会被唤醒，调用 system call，并最终调用 `tcp_recvmsg` 函数去从 socket receive queue 中获取 segment。

### 1.2.4 接收端 - 应用层

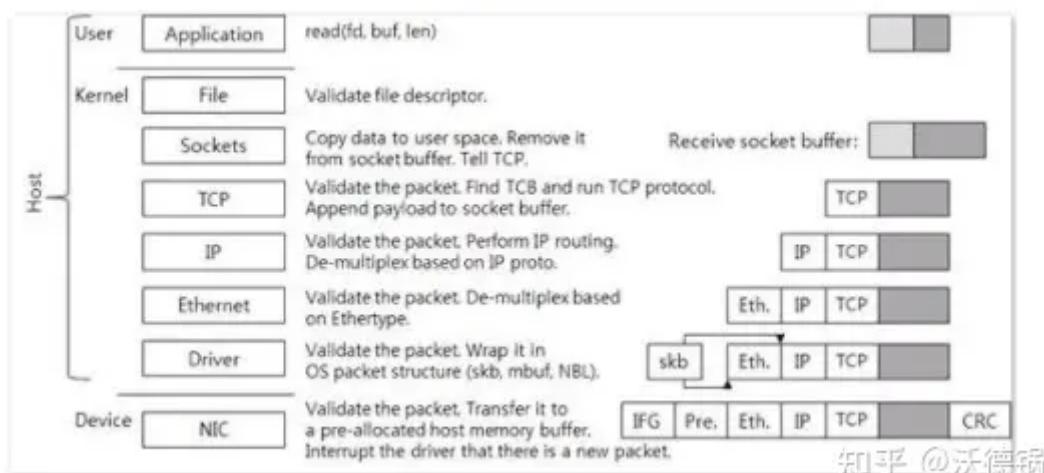
每当用户应用调用 `read` 或者 `recvfrom` 时，该调用会被映射为/`net/socket.c` 中的 `sys_recv` 系统调用，并被转化为 `sys_recvfrom` 调用，然后调用 `sock_recmsg` 函数。

对于 INET 类型的 socket，`/net/ipv4/af_inet.c` 中的 `inet_recvmsg` 方法会被调用，它会调用相关协议的数据接收方法。

对 TCP 来说，调用 `tcp_recvmsg`。该函数从 socket buffer 中拷贝数据到 user buffer。

对 UDP 来说，从 user space 中可以调用三个 system call `recv()`/`recvfrom()`/`recvmsg()` 中的任意一个来接收 UDP package，这些系统调用最终都会调用内核中的 `udp_recvmsg` 方法。

### 1.2.5 报文接收过程简单总结



## 2. Linux `sk_buff` struct 数据结构和队列 (Queue)

### 2.1 `sk_buff`

(本章节摘选自 // ams <http://ekharkernel.blogspot.com/2014/08/what-is-skb-in-linux-kernel-what-are.html>)

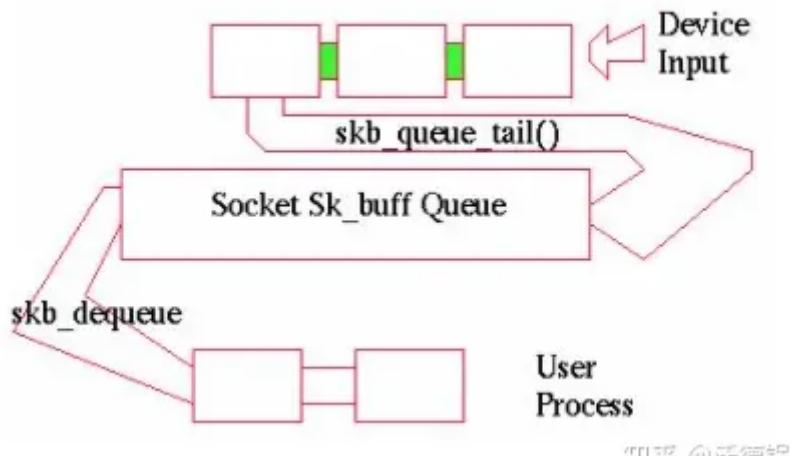
#### 2.1.1 `sk_buff` 是什么

当网络包被内核处理时，底层协议的数据被传送更高层，当数据传送时过程反过来。由不同协议产生的数据(包括头和负载)不断往下层传递直到它们最终被发送。因为这些操作的速度对于网络层的表现至关重要，内核使用一个特定的结构叫 `sk_buff`，其定义文件在 `skbuff.h`。Socket buffer 被用来在网络实现层交换数据而不用拷贝来或去数据包 - 这显著获得速度收益。

`sk_buff` 是 Linux 网络的一个核心数据结构，其定义文件在 `skbuff.h`。

socket kernel buffer (`skb`) 是 Linux 内核网络栈 (L2 到 L4) 处理网络包 (packets) 所使用的 buffer，它的类型是 `sk_buffer`。简单来说，一个 `skb` 表示 Linux 网络栈中的一个 packet；TCP 分段和 IP 分组生产的多个 `skb` 被一个 `skb list` 形式来保存。

`struct sock` 有三个 `skb` 队列 (`sk_buffer queue`)，分别是 `rx`, `tx` 和 `err`。



知乎 @沃德锐

它的主要结构成员：

```
struct sk_buff {  
    /* These two members must be first. */  
    //packet 可以存在于 list 或者 queue 中，这两个成员用于链表处  
   理  
    struct sk_buff      *next;  
    struct sk_buff      *prev;  
    struct sk_buff_head *list; //该 packet 所在的 list  
    ...  
    struct sock         *sk;      //跟该 skb 相关联的 socket
```

```

    struct timeval          stamp; // packet 发送或者接收的时
间, 主要用于 packet sniffers
    struct net_device      *dev;   //这三个成员跟踪该 packet 相关
的 devices, 比如接收它的设备等
    struct net_device      *input_dev;
    struct net_device      *real_dev;

    union {                //指向各协议层 header 结构
        struct tcphdr      *th;
        struct udphdr      *uh;
        struct icmphdr     *icmph;
        struct igmphdr     *igmph;
        struct iphdr       *ipiph;
        struct ipv6hdr     *ipv6h;
        unsigned char       *raw;
    } h;

    union {
        struct iphdr       *iph;
        struct ipv6hdr     *ipv6h;
        struct arphdr      *arph;
        unsigned char       *raw;
    } nh;

    union {
        unsigned char       *raw;
    } mac;

    struct dst_entry      *dst; //指向该 packet 的路由目的结
构, 告诉我们它会被如何路由到目的地
    char                 cb[40]; //SKB control block, 用于各协
议层保存私有信息, 比如 TCP 的顺序号和帧的重发状态
    unsigned int          len, //packet 的长度
                          data_len,
                          mac_len,      /// MAC header 长度
                          csum;         # packet 的 checksum, 用于计
算保存在 protocol header 中的校验和。发送时, 当 checksum
offloading 时, 不设置; 接收时, 可以由device计算

```

`unsigned char local_df, //用于 IPV4 在已经做了分片的情况下再分片，比如 IPSEC 情况下。`

`cloned:1, //在 skb 被 cloned 时设置，此时，skb 各成员是自己的，但是数据是shared的`

`nohdr:1, //用于支持 TSO`

`pkt_type, //packet 类型`

`ip_summed; //网卡能支持的校验和计算的类型，NONE`

`表示不支持，HW 表示支持，`

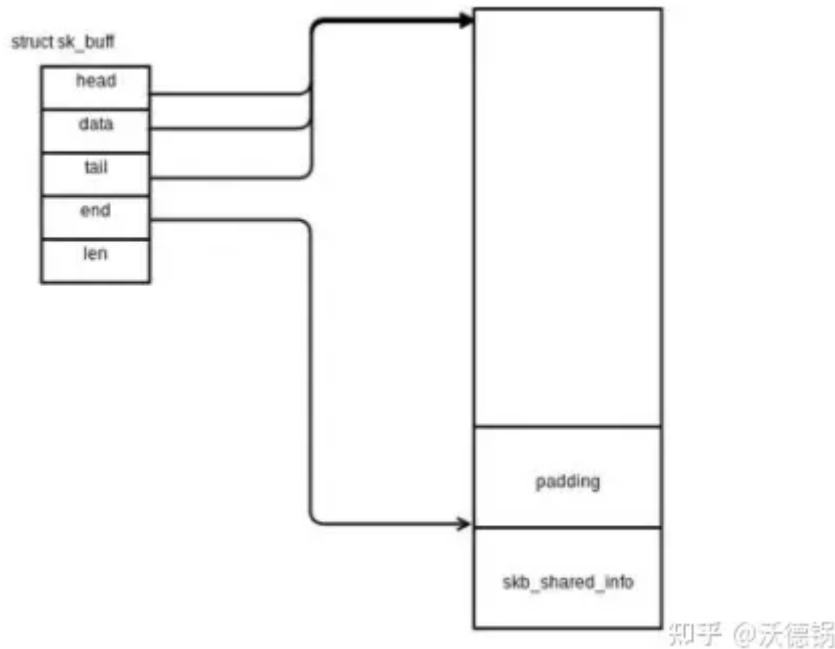
`__u32 priority; //用于 QoS`

`unsigned short protocol, //接收 packet 的协议`

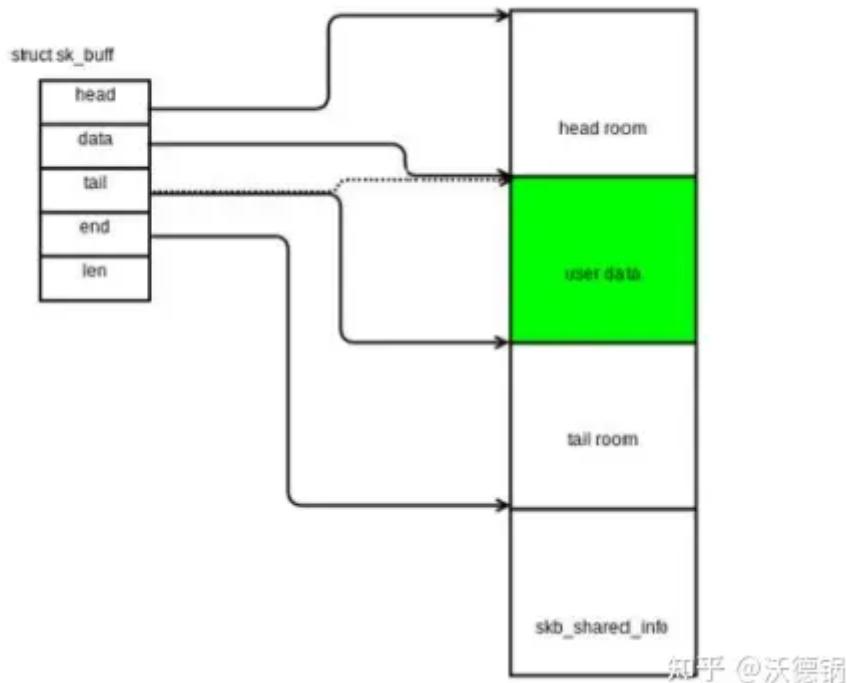
`security;`

## 2.1.2 skb 的主要操作

(1) 分配 `skb = alloc_skb(len, GFP_KERNEL)`

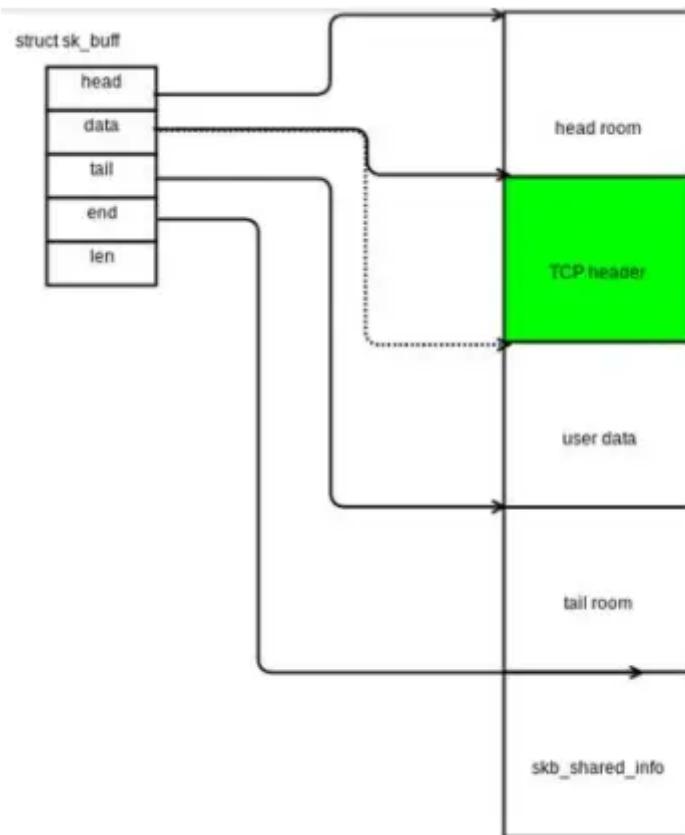


(2) 添加 payload (`skb_put(skb, user_data_len)`)



知乎 @沃德钢

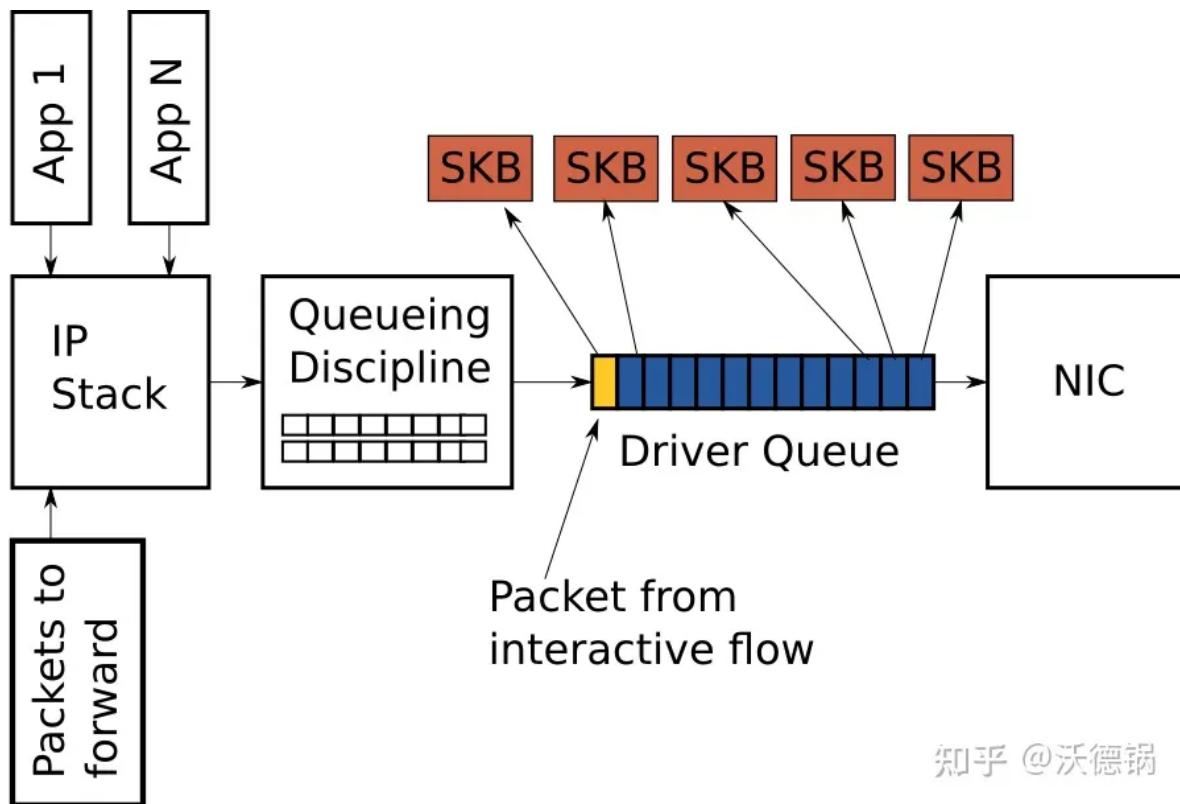
(3) 使用 `skb->push` 添加 protocol header, 或者 `skb->pull` 删除 header



知乎 @沃德钢

## 2.2 Linux 网络栈使用的驱动队列 (driver queue)

### 2.2.1 队列



知乎 @沃德锅

在 IP 栈和 NIC 驱动之间，存在一个 driver queue (驱动队列)。典型地，它被实现为 FIFO ring buffer，简单地可以认为它是固定大小的。这个队列不包含 packet data，相反，它只是保存 socket kernel buffer (skb) 的指针，而 skb 的使用如上节所述是贯穿内核网络栈处理过程的始终的。

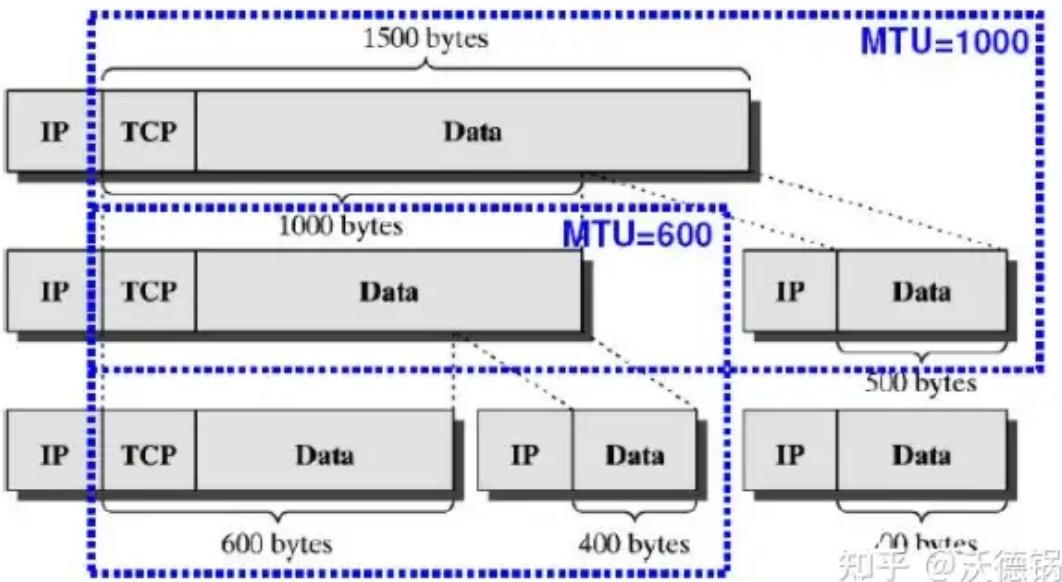
该队列的输入时 IP 栈处理完毕的 packets。这些 packets 要么是本机的应用产生的，要么是进入本机又要被路由出去的。被 IP 栈加入队列的 packets 会被网络设备驱动 (hardware driver) 取出并且通过一个数据通道 (data bus) 发到 NIC 硬件设备并传输出去。

在不使用 TSO/GSO 的情况下，IP 栈发到该队列的 packets 的长度必须小于 MTU。

### 2.2.2 skb 大小 - 默认最大大小为 NIC MTU

绝大多数的网卡都有一个固定的最大传输单元 (maximum transmission unit, MTU) 属性，它是该网络设备能够传输的最大帧 (frame) 的大小。对以太网来说，默认值为 1500 bytes，但是有些以太网络可以支持巨帧 (jumbo frame)，最大能到 9000 bytes。在 IP 网络栈内，MTU 表示能发给 NIC 的最大 packet 的大小。比如，如果一个应用向一个 TCP socket 写入了 2000 bytes 数据，那么 IP 栈需要创建两个 IP packets 来保持每个 packet 的大小等于或者小于 1500 bytes。可见，对于大数据传输，相对较小的 MTU 会导致产生大量的小网络包 (small packets) 并被传入 driver queue。这成为 IP 分片 (IP fragmentation)。

下图表示 payload 为 1500 bytes 的 IP 包，在 MTU 为 1000 和 600 时候的分片情况：



知乎 @沃德锅



