lab2 数据包捕获与分析

物联网工程___2111194___胡博程

一、实验要求

摘自学院网站

数据包捕获与分析编程实验要求如下：

1. 了解NPcap的架构。
2. 学习NPcap的设备列表获取方法、网卡设备打开方法，以及数据包捕获方法。
3. 通过NPcap编程，实现本机的数据包捕获，显示捕获数据帧的源MAC地址和目的MAC地址，以及类型/长度字段的值。
4. 捕获的数据报不要求硬盘存储，但应以简单明了的方式在屏幕上显示。必显字段包括源MAC地址、目的MAC地址和类型/长度字段的值。
5. 编写的程序应结构清晰，具有较好的可读性。

二、实验准备

1. 下载Npcap的SDK、sourcecode、DebugSymbols和可执行文件
2. 在vscode中进行环境搭建
    - 更改工作区中settings.json配置好code Runner插件，此处需要在编译时手动连接wpcap.lib、packet.lib和lws2_32.lib，使得程序正确编译和运行。

```
{
    "code-runner.executorMap": {
        "c": "cd $dir && gcc $fileName -o
${fileDirname}\\output\\$fileNameWithoutExt -
L$workspaceRoot\\Network_technology\\SDK\\Lib\\x64 -lwpcap -lPacket &&
cd $dir\\bin\\ && $fileNameWithoutExt",
        "cpp": "cd $dir && chcp 65001 && g++ $fileName -o
$fileNameWithoutExt -lws2_32 -
L$workspaceRoot\\Network_technology\\SDK\\Lib\\x64 -lwpcap -lPacket  &&
./$fileNameWithoutExt"
    },
    "files.associations": {
        "iostream": "cpp",
        "ostream": "cpp",
        "array": "cpp",
        "atomic": "cpp",
        "*.tcc": "cpp",
        "bitset": "cpp",
        "cctype": "cpp",
        "chrono": "cpp",
        "clocale": "cpp",
        "cmath": "cpp",
```

```
        "cstdarg": "cpp",
        "cstddef": "cpp",
        "cstdint": "cpp",
        "cstdio": "cpp",
        "cstdlib": "cpp",
        "cstring": "cpp",
        "ctime": "cpp",
        "cwchar": "cpp",
        "cwctype": "cpp",
        "deque": "cpp",
        "unordered_map": "cpp",
        "vector": "cpp",
        "exception": "cpp",
        "algorithm": "cpp",
        "map": "cpp",
        "memory": "cpp",
        "memory_resource": "cpp",
        "optional": "cpp",
        "ratio": "cpp",
        "set": "cpp",
        "string": "cpp",
        "string_view": "cpp",
        "system_error": "cpp",
        "tuple": "cpp",
        "type_traits": "cpp",
        "utility": "cpp",
        "fstream": "cpp",
        "initializer_list": "cpp",
        "iosfwd": "cpp",
        "istream": "cpp",
        "limits": "cpp",
        "new": "cpp",
        "sstream": "cpp",
        "stdexcept": "cpp",
        "streambuf": "cpp",
        "thread": "cpp",
        "cinttypes": "cpp",
        "typeinfo": "cpp",
        "codecvt": "cpp"
    }
}
```

- 修改c_cpp_properties.json

```
......
"includePath": [
        "${workspaceFolder}\\**",
        "${workspaceFolder}\\Network_technology\\SDK\\Include\\**"
        //"${workspaceFolder}\\Network technology\\SDK\\Include",
      ],
......
"compilerArgs": [
        "-L${workspaceFolder}\\Network_technology\\SDK\\Lib\\x64",
        "-lwpcap",
        "-lPacket"
      ]
.....
```

includePath定义了头文件的搜索路径，将我们的pcap相关源码路径放进去使得我们能够将需要的头文件`include`进文件，**compilerArgs**定义了传递给编译器的参数，告诉编译器在链接时在哪里查找库文件，动态链接`wpcap`和`Packet`库。

3. 测试环境无误
4. 了解以太帧与IP数据包的结构

三、实验原理

1、Npcap

Npcap 是 WinPcap 的继任者，后者多年来已经成为 Windows 下的数据包捕获标准库。

使用Npcap一般遵循下面的步骤：

安装: 首先，您需要从其官方网站下载并安装 Npcap。

选择适配器: 使用 `pcap_findalldevs` 函数检索可用的网络适配器列表。

打开适配器: 使用 `pcap_open_live` 打开一个网络适配器以进行数据包捕获。

设置过滤器 (可选): 如果您只对某些特定类型的流量感兴趣，可以使用 `pcap_compile` 和 `pcap_setfilter` 函数设置一个过滤器。

开始捕获: 使用 `pcap_loop` 或 `pcap_next` 等函数开始捕获数据包。

处理数据包: 当捕获到数据包时，您可以分析这些数据包，从中提取有用的信息，或者将它们保存到文件中。

关闭适配器: 使用 `pcap_close` 关闭打开的适配器。

清理: 如果使用了 `pcap_findalldevs`，记得使用 `pcap_freealldevs` 释放资源。

2、函数原型解释

（1）pcap_findalldevs

函数原型:

```
int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
```

函数功能：用于检索可用的网络适配器列表。

参数:

- `alldevsp`: 是一个指向pcap_if_t指针的指针。当函数成功返回时，它将指向一个pcap_if_t结构链表，描述了机器上的所有网络适配器。
- `errbuf`: 是一个字符串缓冲区，用于存储可能发生的错误消息。

返回值:

- 返回 `0`：表示成功检索到网络适配器列表。
- 返回 `-1`：表示在检索过程中出现错误。在这种情况下，`errbuf` 参数将填充有关该错误的描述。

（2）pcap_open_live

函数原型：

```
pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms,
char *errbuf);
```

函数功能：打开一个物理网络适配器，以便捕获流经该适配器的数据包。

参数：

- `device`：是一个字符串，指定要打开的网络适配器的名称。
- `snaplen`：指定了要捕获的每个数据包的最大字节数。
- `promisc`：设置为1时，适配器将被设置为混杂模式，这意味着它会捕获所有经过的数据包，而不仅仅是发送给该适配器的数据包。
- `to_ms`：读取超时，以毫秒为单位。
- `errbuf`：是一个字符串缓冲区，用于存储可能发生的错误消息。

返回值：

- `pcap_t *`类型
- 返回非 `NULL` 值：表示一个指向打开的适配器的 `pcap_t` 结构的指针，这表明成功打开了网络适配器。
- 返回 `NULL`：表示在尝试打开网络适配器时出现错误。在这种情况下，`errbuf` 参数将填充有关该错误的描述。

（3）pcap_loop

函数原型：

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

函数功能：捕获数据包，并为每个捕获的数据包调用指定的回调函数。

参数：

- `p`：是一个指向pcap_t结构的指针，该结构描述了一个打开的数据包捕获会话。
- `cnt`：指定要捕获的数据包数量。设置为0表示无限制。
- `callback`：当捕获到数据包时要调用的回调函数的指针。
- `user`：传递给回调函数的一个指针，该指针指向用户定义的数据。

返回值：

- 返回 `0`：表示成功捕获了指定数量的数据包（由 `cnt` 参数指定）。
- 返回 `-1`：表示在捕获过程中出现错误。您可以使用 `pcap_geterr` 函数获取有关该错误的描述。
- 返回 `-2`：表示数据包捕获被 `pcap_breakloop` 函数中断。

（4）pcap_close

函数原型：

```
void pcap_close(pcap_t *p);
```

函数功能：关闭一个数据包捕获会话。

参数:

- `p`: 是一个指向pcap_t结构的指针，该结构描述了一个打开的数据包捕获会话。

（5）pcap_freealldevs

函数原型:

```
void pcap_freealldevs(pcap_if_t *alldevs);
```

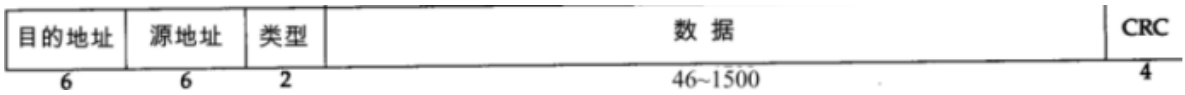
函数功能：释放通过`pcap_findalldevs`函数获取的网络适配器列表。

参数:

- `alldevs`: 是一个指向pcap_if_t结构链表的指针，描述了机器上的所有网络适配器。

3、以太帧与IP数据包的结构

（1）以太帧结构



字段	含义
前同步码	用来使接收端的适配器在接收 MAC 帧时能够迅速调整时钟频率，使它和发送端的频率相同。前同步码为 7 个字节，1 和 0 交替。
帧开始定界符	帧的起始符，为 1 个字节。前 6 位 1 和 0 交替，最后的两个连续的 1 表示告诉接收端适配器：“帧信息要来了，准备接收”。
目的地址	接收帧的网络适配器的物理地址（MAC 地址），为 6 个字节（48 比特）。作用是当网卡接收到一个数据帧时，首先会检查该帧的目的地址，是否与当前适配器的物理地址相同，如果相同，就会进一步处理；如果不同，则直接丢弃。
源地址	发送帧的网络适配器的物理地址（MAC 地址），为 6 个字节（48 比特）。
类型	上层协议的类型。由于上层协议众多，所以在处理数据的时候必须设置该字段，标识数据交付哪个协议处理。例如，字段为 0x0800 时，表示将数据交付给 IP 协议。
数据	也称为效载荷，表示交付给上层的数据。以太网帧数据长度最小为 46 字节，最大为 1500 字节。如果不足 46 字节时，会填充到最小长度。最大值也叫最大传输单元（MTU）。 在 Linux 中，使用 ifconfig 命令可以查看该值，通常为 1500。
帧检验序列 FCS	检测该帧是否出现差错，占 4 个字节（32 比特）。发送方计算帧的循环冗余码校验（CRC）值，把这个值写到帧里。接收方计算机重新计算 CRC，与 FCS 字段的值进行比较。如果两个值不相同，则表示传输过程中发生了数据丢失或改变。这时，就需要重新传输这一帧。

wireshark捕获数据包直观图

```
▾ Ethernet II, Src: IETF-VRRP-VRID_08 (00:00:5e:00:01:08), Dst: IntelCor_6e:3b:dd (2c:8d:b1:6e:3b:dd)
  ▾ Destination: IntelCor_6e:3b:dd (2c:8d:b1:6e:3b:dd)
      Address: IntelCor_6e:3b:dd (2c:8d:b1:6e:3b:dd)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: IETF-VRRP-VRID_08 (00:00:5e:00:01:08)
      Address: IETF-VRRP-VRID_08 (00:00:5e:00:01:08)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
```

（2）ip数据包首部结构

IP数据报格式图解

wireshark捕获数据包直观图

```
Internet Protocol Version 4, Src: 39.156.132.17, Dst: 10.136.63.16
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      0000 00.. = Differentiated Services Codepoint: Default (0)
      .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    Total Length: 52
    Identification: 0x0000 (0)
  Flags: 0x40, Don't fragment
      0... .... = Reserved bit: Not set
      .1.. .... = Don't fragment: Set
      ..0. .... = More fragments: Not set
    Fragment Offset: 0
    Time to Live: 50
    Protocol: TCP (6)
    Header Checksum: 0x537f [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 39.156.132.17
    Destination Address: 10.136.63.16
```

4、关于首部校验

IP数据包的首部校验码（通常称为IP 校验和）是一个用于检测IP首部是否被破坏的简单的错误检测机制。它不涉及数据部分，仅涉及IP 首部。

使用方法

1. **发送时的计算**：当主机或路由器发送一个IP 数据包时，它会计算首部的校验和，并将其放在IP 首部的校验和字段中。
2. **接收时的验证**：当主机或路由器接收一个IP 数据包时，它会计算首部的校验和（方法与发送时相同）。如果计算出的结果是0xFFFF，那么首部很可能是完整的，没有损坏。如果结果不是0xFFFF，那么数据包的首部可能已经被损坏，该数据包将被丢弃。

计算方法

1. **初始化校验和为0**：开始时，将校验和设置为0。
2. **分组处理**：将IP 首部的数据视为一个一个的16位整数序列进行处理。如果首部字节总数不是偶数，则添加填充（通常为0）使其长度成为偶数。
3. **累加**：将所有的16位整数加在一起。如果任何加法的结果大于16位，那么将溢出的部分加回到结果的低16位上。
4. **取反**：将最后的结果取反，即将所有的位进行翻转。

5、网络序与主机序

不同的CPU有不同的字节序类型 这些字节序是指整数在内存中保存的顺序 这个叫做主机序， 分为**大端序**和**小端序**

- **大端序**：高位字节排放在内存的低地址端，低位字节排放在内存的高地址端
- **小端序**：低位字节排放在内存的低地址端，高位字节排放在内存的高地址端

- 例如：0x12 34 56 78在内存中的存放形式为：
  - 内存　　低地址 ----------> 高地址
  - Big-Endian:　　0x12 | 0x34 | 0x56 | 0x78
  - Little-Endian:　　0x78 | 0x56 | 0x34 | 0x12

网络字节顺序是TCP/IP中规定好的一种数据表示格式，它与具体的CPU类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释。网络字节顺序采用big endian排序方式。

在Npcap中，网络序和主机序之间的转换使用了一组标准的函数。

以下是这些函数：

1. **htons()**:
   - **原型**: `uint16_t htons(uint16_t hostshort);`
   - **描述**: 将一个16位数从主机字节序转换为网络字节序。
2. **htonl()**:
   - **原型**: `uint32_t htonl(uint32_t hostlong);`
   - **描述**: 将一个32位数从主机字节序转换为网络字节序。
3. **ntohs()**:
   - **原型**: `uint16_t ntohs(uint16_t netshort);`
   - **描述**: 将一个16位数从网络字节序转换为主机字节序。
4. **ntohl()**:
   - **原型**: `uint32_t ntohl(uint32_t netlong);`
   - **描述**: 将一个32位数从网络字节序转换为主机字节序。

注意：并不是所有的网络数据包字段都要进行字节序转换，对于8字节及以下的的字段并不需要这个步骤

6、补充说明

区分两种结构体`pcap_if_t`与`pcap_t`，前者是网络适配器列表的指针，描述网络接口，包含了接口的名称、描述、地址等信息，但不涉及任何与捕获会话相关的操作。后者准确来说是适配器句柄，代表一个活跃的数据包捕获会话与`pcap_if_t`不同，为函数`pcap_open_live`的返回值

四、实验过程

1、封装结构体用于储存以太帧与IP数据包

具体代码如下

```
struct ip_header // IP首部
{
```

```
    uint8_t Ver_HLen;       // 8位版本+首部长度
    uint8_t TOS;            // 8位服务类型
    uint16_t TotalLen;      // 16位总长度
    uint16_t ID;            // 16位标识
    uint16_t Flag_Segment; // 16位标志+片偏移
    uint8_t TTL;            // 8位生存时间
    uint8_t Protocol;       // 8位协议
    uint16_t Checksum;      // 16位首部校验和
    uint32_t SrcIP;         // 32位源IP地址
    uint32_t DstIP;         // 32位目的IP地址
    uint16_t cal_checksum();
    void showIPPacket();
};

struct ether_header
{
    uint8_t Ether_Dhost[6]; // 目的地址
    uint8_t Ether_Shost[6]; // 源地址
    uint16_t Ether_Type;    // 以太网类型
    void showEtherPacket(ip_header *ip_protocol);
};
```

2、打印设备列表并选择一个网卡设备

```
    pcap_if_t *alldevs, *device;   // 网络适配器列表的指针：pcap_if_t，pcap_if_t结构体链表，描述网络接口
                                   // 包含了接口的名称、描述、地址等信息，但不涉及任何与捕获会话相关的操作。
    pcap_t *adhandle;              // 适配器句柄，代表一个活跃的数据包捕获会话与pcap_if_t不同
    char errbuf[PCAP_ERRBUF_SIZE]; // 用于储存错误消息的缓冲区

    /*
    pcap_if_t 是用于识别和选择网络接口的。
    pcap_t 是用于处理实际的数据包捕获的。*/

    // 获取设备列表
    if (pcap_findalldevs(&alldevs, errbuf) == -1) // 错误消息会存在errbuf中，返回0成功，-1失败
    {
        std::cerr << "Error in pcap_findalldevs: " << errbuf << std::endl;
        return 1;
    }

    // 显示设备列表
    std::cout << "Available devices are:\n";
    int i = 0;
    for (device = alldevs; device; device = device->next)
    {
        std::cout << ++i << ". " << device->name << " ";
        // 打印有名称的设备
        if (device->description)
            std::cout << "(" << device->description << ")\n";
        else
            std::cout << "(No description available)\n";
    }
```

```
    if (i == 0)
    {
        std::cout << "\nNo interfaces found! Make sure pcap is installed.\n";
        return 1;
    }

    // 选择一个设备
    int inum;
    std::cout << "Enter the interface number (1-" << i << "):";
    std::cin >> inum;

    if (inum < 1 || inum > i)
    {
        std::cout << "\nInterface number out of range.\n";
        pcap_freealldevs(alldevs);
        return 1;
    }

    // 跳转到选定的适配器
    for (device = alldevs, i = 0; i < inum - 1; device = device->next, i++)
        ;
```

使用 `pcap_findalldevs` 函数来获取系统上所有可用的网络适配器列表，并将这些适配器的信息存储在 `alldevs` 指针所指向的链表中。随后通过遍历 `alldevs` 链表，代码列出所有可用的网络适配器。输入想要监听的设备号通过 `alldevs` 链表的 `next` 成员找到相应的设备。

3、打开设备并开始捕获数据包

```
// 打开设备
    // 设备名称、捕获长度、混杂模式（接收所有经过的数据包）、读超时、错误缓冲区
    if ((adhandle = pcap_open_live(device->name, 65536, 1, 1000, errbuf)) ==
NULL)
    {
        std::cerr << "\nUnable to open the " << device->name << " is not
supported by pcap\n";
        pcap_freealldevs(alldevs);
        return 1;
    }

    std::cout << "\nListening on " << device->description << "...\n";

    // 释放设备列表
    pcap_freealldevs(alldevs);

    // 开始捕获
    int packetNum = 0;
    cout << "你想一共捕获多少个数据包";
    cin >> packetNum;
    pcap_loop(adhandle, packetNum, packet_handler, NULL);
    // packet_handler 是一个回调函数，每当捕获到一个数据包时，这个函数都会被调用
```

打开一个网络适配器：使用 `pcap_open_live()` 函数尝试打开一个网络适配器（设备）以捕获数据包。

- `device->name`：要打开的网络适配器的名称。
- `65536`：捕获的每个数据包的最大长度。

- `1`: 设置适配器为混杂模式，这意味着它会捕获所有流过的数据包，而不仅仅是那些目标为该适配器的数据包。
- `1000`: 读取超时，单位为毫秒。
- `errbuf`: 存储可能的错误消息的缓冲区。
- 如果打开设备失败（返回`NULL`），则输出错误消息，释放之前获取的设备列表，并退出程序。
- 如果成功打开设备，输出一个消息告诉用户正在监听哪个设备。

释放设备列表： 使用`pcap_freealldevs()`函数释放之前通过`pcap_findalldevs()`获取的设备列表。这是一个好习惯，以确保不浪费内存。

开始捕获：

- 询问用户想要捕获多少个数据包并将该数字存储在`packetNum`变量中。
- 使用`pcap_loop`函数开始捕获数据包。
  - `adhandle`: 表示打开的设备的句柄。
  - `packetNum`: 要捕获的数据包数量。
  - `packet_handler`: 每当捕获到一个数据包时要调用的回调函数。
  - `NULL`: 传递给回调函数的用户定义数据。

（1）回调函数packet_handler

```
void analysis_Ethernet(u_char *user_data, const struct pcap_pkthdr *pkInfo,
const u_char *packet)
{
    ether_header *ethernet_protocol = (struct ether_header *)packet; // 获取以太网
帧头部
    ip_header *ip_protocol = (struct ip_header *)(packet + 14);      // 获取以太网
帧头部
    ethernet_protocol->showEtherPacket(ip_protocol);
}

void packet_handler(u_char *user_data, const struct pcap_pkthdr *pkthdr, const
u_char *packet)
{
    // 参数解释
    //   user_data传递用户定义的数据到回调函数中，通常用不到
    //   pkInfo保存了此数据包的时间信息和长度信息
    //   packet是数据包的内容
    static int packet_count = 0;

    packet_count++;

    cout << "Now Packet Number: " << dec << packet_count << endl;

    analysis_Ethernet(user_data, pkthdr, packet);
}
```

packet_handler函数： 这是每当捕获到一个数据包时都会被调用的回调函数。

- 函数首先增加`packet_count`以跟踪捕获的数据包数量，并输出当前的数据包编号。
- 然后，它调用`analysis_Ethernet()`函数来分析和显示数据包的内容。

analysis_Ethernet函数： 这个函数分析和显示以太网帧的内容。

- 使用指针类型转换从原始数据包中获取以太网帧的头部。

- 同样地，也获取IP头部（通常位于以太网帧头部之后的14字节位置）。
- 调用`showEtherPacket()`方法来显示以太网帧的内容。此方法传入了一个`ip_header *`类型的变量`ip_protocol`，用于找到以太帧的IP数据包部分，对这个部分进行分析。

（2）数据包分析输出函数showEtherPacket，showIPPacke

```
void ether_header::showEtherPacket(ip_header *ip_protocol)
{
    uint16_t type = ntohs(this->Ether_Type);
    cout << "<------------------开始解析以太网帧数据包------------------> " <<
endl;
    cout << "目的MAC地址：" << hex << (int)this->Ether_Dhost[0] << ":" <<
(int)this->Ether_Dhost[1] << ":" << (int)this->Ether_Dhost[2] << ":" <<
(int)this->Ether_Dhost[3] << ":" << (int)this->Ether_Dhost[4] << ":" <<
(int)this->Ether_Dhost[5] << endl;
    cout << "源MAC地址：" << hex << (int)this->Ether_Shost[0] << ":" <<
(int)this->Ether_Shost[1] << ":" << (int)this->Ether_Shost[2] << ":" <<
(int)this->Ether_Shost[3] << ":" << (int)this->Ether_Shost[4] << ":" <<
(int)this->Ether_Shost[5] << endl;
    cout << "以太网类型：" << setfill('0') << setw(4) << hex << type << endl;
    switch (type)
    {
    case 0x0800:
        cout << "以太网类型：IPv4" << endl;
        ip_protocol->showIPPacket();
        break;
    case 0x0806:
        cout << "以太网类型：ARP" << endl;
        break;
    case 0x8035:
        cout << "以太网类型：RARP" << endl;
        break;
    case 0x86DD:
        cout << "以太网类型：IPv6" << endl;
        break;
    default:
        cout << "以太网类型：其他" << endl;
        break;
    }
    cout << "<------------------完成以太网帧数据包解析------------------> " << endl
         << endl;
}
```

从以太帧中抽取目的MAC地址、源MAC地址和type，根据type字段对下层的协议进行分类，如果是IPV4协议——对应`type`字段为`0x8000`，使用传进来的`ip_protocol`执行`showIPPacket()`函数对IP数据包进行分析

`showIPPacket()`代码如下

```
void ip_header::showIPPacket()
{
    cout << "<---------------开始解析IP层数据包---------------> " << endl;
    cout << "版本号：" << (this->Ver_HLen >> 4) << endl;
    cout << "首部长度：" << (this->Ver_HLen & 0x0F) * 4 << endl;
    cout << "服务类型：" << (int)this->TOS << endl; // 3位的优先级子字段
    // 优先级子字段用于指定数据包的优先级
```

```
    switch ((int)this->TOS >> 5)
    {
    case 0:
        cout << "优先级：Routine,数据包不需要特殊处理。" << endl;
        break;
    case 1:
        cout << "优先级：Priority,数据包不需要特殊处理。" << endl;
        break;
    case 2:
        cout << "优先级：Immediate,数据包需要立即处理。" << endl;
        break;
    case 3:
        cout << "优先级：Flash,数据包需要快速处理。" << endl;
        break;
    case 4:
        cout << "优先级：Flash Override,数据包需要立即处理，并覆盖其他数据包的处理。" <<
endl;
        break;
    case 5:
        cout << "优先级：CRITIC/ECP,数据包是关键数据或网络控制数据，需要最高优先级处理。"
<< endl;
        break;
    case 6:
        cout << "优先级：Internetwork Control,数据包是网络控制数据，需要高优先级处理。"
<< endl;
        break;
    case 7:
        cout << "优先级：Network Control,数据包是网络控制数据，需要最高优先级处理。" <<
endl;
        break;
    }
    cout << "总长度：" << ntohs(this->TotalLen) << endl;
    cout << "标识：" << ntohs(this->ID) << endl;
    cout << "标志：" << bitset<3>(ntohs(this->Flag_Segment) >> 13) << endl;
    cout << "对应标志：保留位、不分片（DF）位和更多分片（MF）位。" << endl;
    cout << "片偏移：" << (ntohs(this->Flag_Segment) & 0x1FFF) << endl;
    cout << "生存时间：" << (int)this->TTL << endl;
    cout << "协议编号：" << (int)this->Protocol << endl;
    switch ((int)this->Protocol)
    {
    case 1:
        cout << "协议类型：ICMP" << endl;
        break;
    case 2:
        cout << "协议类型：IGMP" << endl;
        break;
    case 6:
        cout << "协议类型：TCP" << endl;
        break;
    case 17:
        cout << "协议类型：UDP" << endl;
        break;
    case 58:
        cout << "协议类型：ICMPv6" << endl;
        break;
    case 89:
        cout << "协议类型：OSPF" << endl;
        break;
```

```
    case 132:
        cout << "协议类型：SCTP" << endl;
        break;
    case 255:
        cout << "协议类型：RAW" << endl;
        break;
    }
    cout << "首部校验和：" << ntohs(this->Checksum) << endl;
    cout << "验证首部校验和：" << this->cal_checksum() << endl;

    // inet_ntoa() 函数被用于将IP数据包的源IP地址转换为点分十进制的字符串表示
    cout << "源IP地址：" << inet_ntoa(*(struct in_addr *)&this->SrcIP) << endl;
    cout << "目的IP地址：" << inet_ntoa(*(struct in_addr *)&this->DstIP) << endl;
    cout << "<---------------完成IP层数据包解析----------------> " << endl;
}
```

根据各字段抽取版本号，首部长度、服务类型等字段，进而对于服务类型和协议编号进行更细致的划分，此外还根据数据包自带的校验和和依据算法计算得到的校验和进行了比对，检验了数据包的正确性

（3）计算校验和函数cal_checksum

```
uint16_t ip_header::cal_checksum()
{
    uint32_t sum = 0;
    uint16_t *ptr = reinterpret_cast<uint16_t *>(this); // 转成指向16位无符号整数指针，十六位十六位处理
    // 将校验和字段置为0，因为checksum本来也是16位的
    this->Checksum = 0;
    // 将IP首部中的每16位相加
    for (int i = 0; i < sizeof(ip_header) / 2; ++i)
    {
        sum += ntohs(ptr[i]); // noths函数将一个无符号短整型数从网络字节顺序转换为主机字节顺序
    }
    // 把高16位和低16位相加
    while (sum >> 16)
    {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 返回校验和
    return static_cast<uint16_t>(~sum);
}
```

算法解释：

```
uint32_t sum = 0;
```

定义一个 32 位的无符号整数变量 `sum`，用于存储校验和的中间结果。

```
uint16_t *ptr = reinterpret_cast<uint16_t *>(this);
```

定义一个指向当前 IP 首部的指针 `ptr`，并将其强制转换为一个指向 16 位无符号整数的指针。这样可以方便地对 IP 首部中的每个 16 位进行处理。

```
this->Checksum = 0;
```

将 IP 首部中的校验和字段 `Checksum` 置为 0，以便重新计算校验和。

```
for (int i = 0; i < sizeof(ip_header) / 2; ++i)
{
    sum += ntohs(ptr[i]);
}
```

对 IP 首部中的每个 16 位进行处理，将它们的值加到 `sum` 变量中。这里使用了 `ntohs()` 函数将网络字节序转换为主机字节序。

```
while (sum >> 16)
{
    sum = (sum & 0xFFFF) + (sum >> 16);
}
```

将 `sum` 变量的高 16 位和低 16 位相加，直到高 16 位为 0。这是为了将可能存在的进位加到低 16 位中。

```
return static_cast<uint16_t>(~sum);
```

将 `sum` 变量按位取反，并将结果转换为 16 位无符号整数，作为函数的返回值。这是因为 IP 首部的校验和是取反后的值。

3、最后关闭数据包捕获会话。

```
    // 关闭句柄
    pcap_close(adhandle);

    return 0;
```

五、实验结果

```
Active code page: 65001
Available devices are:
1. \Device\NPF_{39A679BB-9F6D-4AA1-BE2F-BE55BF3440F8} (WAN Miniport (Network
Monitor))
2. \Device\NPF_{8B8F25B8-C216-4F59-AE6D-7BA33A00C88C} (WAN Miniport (IPv6))
3. \Device\NPF_{5D6EC51C-060D-4D4D-BAB7-FC1234F6C63E} (WAN Miniport (IP))
4. \Device\NPF_{C3023BEB-A2C5-4967-8181-21B143524782} (Bluetooth Device
(Personal Area Network))
5. \Device\NPF_{E8B84E9D-8C25-4A16-96E5-791599AB0B16} (Intel(R) Wi-Fi 6 AX201
160MHz)
6. \Device\NPF_{E9FC2A0D-5687-43EF-9B34-B23425FE9914} (VMware Virtual Ethernet
Adapter for VMnet8)
7. \Device\NPF_{3DEC8C05-FBF3-4F08-A26F-89EBDED92931} (VMware Virtual Ethernet
Adapter for VMnet1)
8. \Device\NPF_{18AD10F2-88C3-42F7-BD5A-19D38B7DBD61} (Microsoft Wi-Fi Direct
Virtual Adapter #2)
9. \Device\NPF_{13B0D18E-5EEB-4D44-8617-702BB48EFE62} (Microsoft Wi-Fi Direct
Virtual Adapter)
10. \Device\NPF_Loopback (Adapter for loopback traffic capture)
```

```
11. \Device\NPF_{C48D3ECF-3FC3-499B-BBBC-37A10818A872} (Realtek PCIe GbE Family Controller)
Enter the interface number (1-11):5

Listening on Intel(R) Wi-Fi 6 AX201 160MHz...
你想一共捕获多少个数据包5
Now Packet Number: 1
<------------------开始解析以太网帧数据包------------------>
目的MAC地址：1:0:5e:7f:ff:fa
源MAC地址：5c:c3:36:2e:d6:e2
以太网类型：0800
以太网类型：IPv4
<---------------开始解析IP层数据包---------------->
版本号：4
首部长度：14
服务类型：0
优先级：Routine,数据包不需要特殊处理。
总长度：14d
标识：ab4
标志：010
对应标志：保留位、不分片（DF）位和更多分片（MF）位。
片偏移：0
生存时间：4
协议编号：11
协议类型：UDP
首部校验和：ccfe
验证首部校验和：ccfe
源IP地址：10.136.163.107
目的IP地址：239.255.255.250
<---------------完成IP层数据包解析---------------->
<------------------完成以太网帧数据包解析------------------>

Now Packet Number: 2
<------------------开始解析以太网帧数据包------------------>
目的MAC地址：1:0:5e:7f:ff:fa
源MAC地址：5c:c3:36:2e:d6:e2
以太网类型：0800
以太网类型：IPv4
<---------------开始解析IP层数据包---------------->
版本号：4
首部长度：14
服务类型：0
优先级：Routine,数据包不需要特殊处理。
总长度：156
标识：ab5
标志：010
对应标志：保留位、不分片（DF）位和更多分片（MF）位。
片偏移：0
生存时间：4
协议编号：11
协议类型：UDP
首部校验和：ccf4
验证首部校验和：ccf4
源IP地址：10.136.163.107
目的IP地址：239.255.255.250
<---------------完成IP层数据包解析---------------->
<------------------完成以太网帧数据包解析------------------>
......
```

六、需要注意的地方与困难解决

1、**注意ether_header与ip_header结构体中成员的声明顺序**

在`analysis_Ethernet`函数中我们使用强制类型转换将packet转换成struct ether_header *类型，使得packet中的数据**按顺序填充**到ether_header结构体的成员变量之中，需要保证两结构体中成员变量声明顺序与实际上的以太网帧与IP数据包头部的各字段顺序是相同的（例如以太帧中最高6位是目的地址，再6位是源地址，那么在ether_header结构体中就要先声明目的地址的变量再声明源地址的变量）

2、重新编译程序的时候保证程序对应可执行文件没有运行，不然会出现权限问题，无法对正在运行的程序做覆盖更改

3、需要选择对应与本机电脑架构的动态链接库，X86和X64是不同的