第6单元　嗅探器

6.1　知识讲解

6.1.1　原始套接字

Linux网络协议栈中的套接字主要分为三类：

- 流式套接字：工作在传输层TCP协议之上
- 数据报套接字：工作在传输层UDP协议之上
- 原始套接字：工作在网络层或物理层之上

1. 原始套接字的特点

1) 读写ICMP/IGMP报文

对于ICMP、IGMP等封装在IP数据包内部但又在传输层以下的报文，内核总会为协议类型匹配的原始套接字传递一份拷贝。通过这种方式，可以在用户空间处理这些报文，而无需内核参与，从而减轻系统内核的负担。

2) 读写部分IP报文

内核对于无法识别协议类型的IP报文，会首先查找有没有合适的原始套接字，如果有这样的套接字，内核会将该报文复制一份给它们，否则直接丢弃该报文。

3) 直接通过物理层捕获数据包

对于使用TCP、UDP等传输层协议的IP报文，内核可以识别，会交给相应的功能模块进行处理，而不会传递给原始套接字，除非创建套接字时指定了PF_PACKET协议族(socket函数的第一个参数)和ETH_P_IP或ETH_P_ALL通信协议(socket函数的第三个参数)。这样的原始套接字将直接通过物理层捕获数据包。

4) 自己构造IP包头

原始套接字可以自己构造IP包头，但必须先通过setsockopt函数为该套接字设置IP_HDRINCL选项，以告诉内核IP包头由自己填充而不由内核填充。

5) 需要超级用户权限

创建原始套接字需要超级用户权限。

2. 原始套接字的相关操作

1) 创建原始套接字

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

第一个参数domain表示协议族，一般情况下用PF_INET或AF_INET表示互联网协议族，二者可以认为是等价的。严格来讲PF_INET是协议族(Protocol Family)，而AF_INET则是地址族(Address Family)。按照最初的设计，一个协议族可以包含多个地址族，但迄今为止每个协议族下面仅有一个地址族。在有关套接字的头文件中有如下宏定义：

```
#define PF_INET AF_INET
```

这两个宏在数值上相等，功能上亦无差别。另外，如果需要直接通过物理层捕获数据包，可将domain参数设置为PF_PACKET。

第二个参数type表示套接字类型。SOCK_RAW表示原始套接字。

第三个参数protocol表示通信协议。如果想通过原始套接字捕获TCP或UDP包，需要将此参数设置为ETH_P_IP或ETH_P_ALL。出于兼容性的考虑，最好先通过htons函数将这两个宏转换为网络字节序短整型。

创建原始套接字需要超级用户权限，否则socket函数会返回-1，同时将errno设置为EACCES。

2) 绑定和连接

通常情况下，原始套接字并不需要绑定操作。如果绑定也只是绑定IP地址而不会涉及端口号：

- 绑定到某个特定IP地址的原始套接字：
  - 接收数据时，只能接收以此IP为目的地址的数据包
  - 发送数据时，数据包的源地址会被内核设置为此IP
- 不绑定任何特定IP地址的原始套接字：
  - 接收数据时，可接收以任意IP为目的地址的数据包
  - 发送数据时，数据包的源地址会被设置为接口主IP

不同主机上的原始套接字可以通过connect函数彼此连接，同样只连接IP地址而不涉及端口号。连接成功以后可以通过write或send函数发送数据，read或recv函数接收数据，而发送的目的和接收的源即为连接的对方主机。

3) 读写原始套接字

通过原始套接字发送数据，默认情况下，由内核负责对IP包头的填充，但如果通过setsockopt函数为该套接字设置了IP_HDRINCL选项，则必须由用户程序填充IP包头，内核只负责计算并填充IP包头的校验和。当数据大于链路的最大传输单元(Maximum Transmission Unit, MTU)时，内核将自动对数据包进行分片。

读写原始套接字和读写普通套接字并没有显著的差别：

- sendto/recvfrom：向/从任意主机发送/接收数据包
- send/recv：向/从已连接(connect)的对方主机发送/接收数据包
- write/read：与send/recv等价

3. 数据包的处理

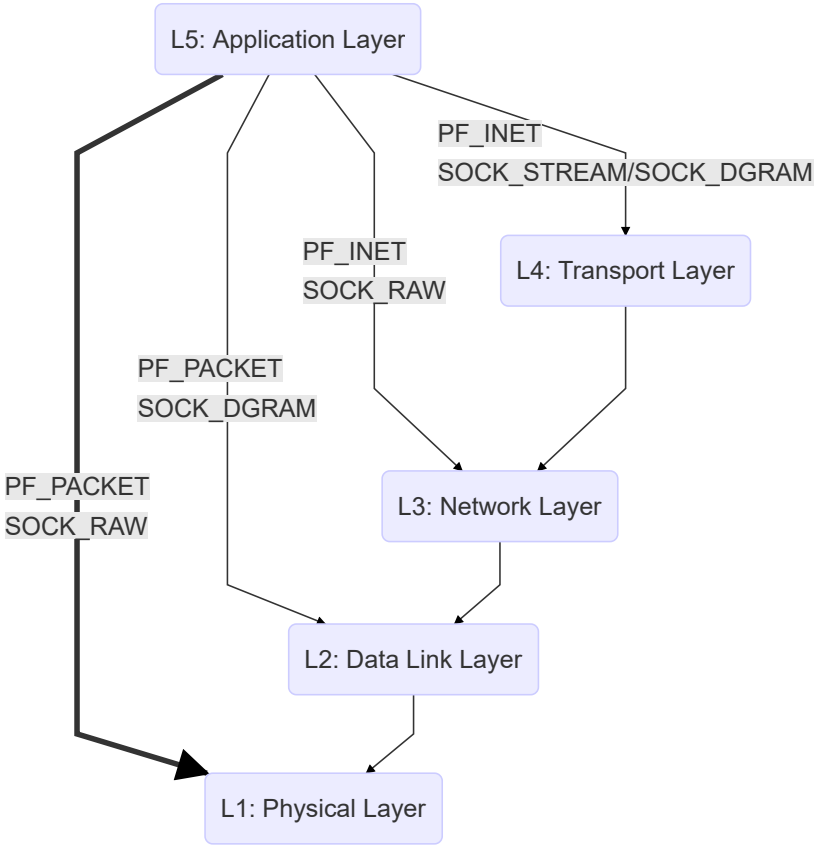
当内核收到一个无法识别协议类型的IP报文时，会检查系统中所有进程的所有原始套接字，根据以下原则决定是否将该IP报文拷贝给相应的套接字：

- 套接字的通信协议(socket函数的第三个参数)与该IP报文的通信协议完全匹配
- 套接字的绑定地址(调用过bind函数)与该IP报文的目的地址完全匹配
- 套接字的连接地址(调用过connect函数)与该IP报文的源地址完全匹配

如果一个原始套接字的通信协议(socket函数的第三个参数)为0，并且没有绑定(bind)和连接(connect)，那么该套接字将能捕获到内核收到的除TCP和UDP以外的所有IP报文。

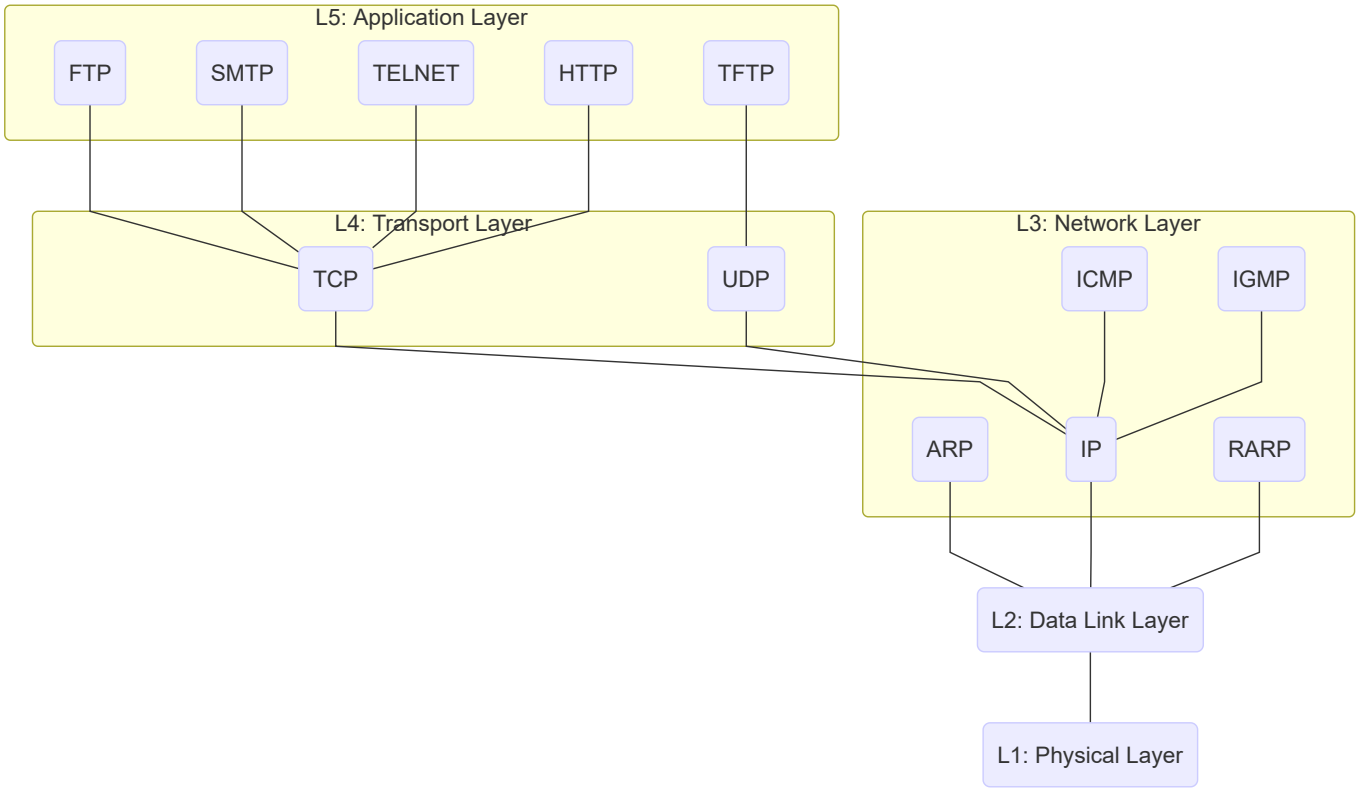
如果希望原始套接字也能捕获TCP和UDP数据包，可在创建该套接字时指定PF_PACKET协议族(socket函数的第一个参数)和ETH_P_IP或ETH_P_ALL通信协议(socket函数的第三个参数)。这样的原始套接字将直接通过物理层捕获数据包。

协议族和套接字类型与网络协议栈的关系如下图所示：



6.1.2 TCP/IP协议栈

TCP/IP协议栈模型中的常见协议及其位置如下图所示：

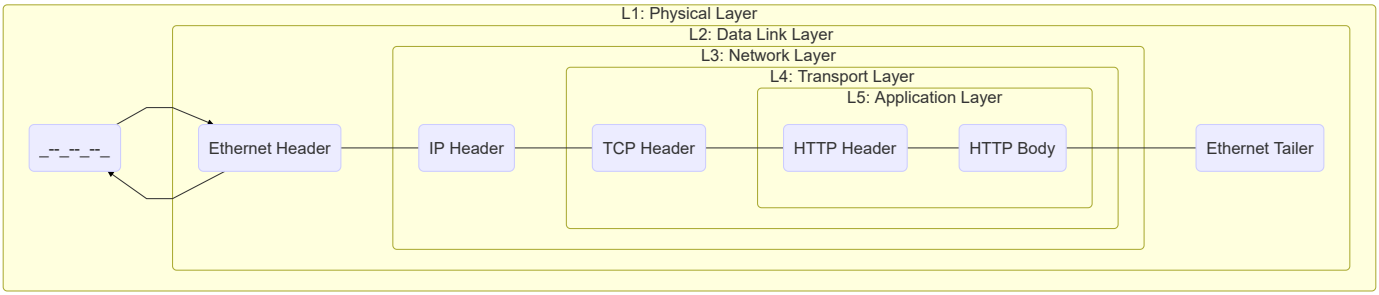


6.1.3 数据封装与解析

当应用程序通过协议栈向网络发送数据时，应用层的数据要依次经历传输层、网络层、数据链路层，最后进入物理层。每一层都要为数据添加该层的头部(有的层还有尾部)信息，以实现层次化控制。这个过程称为数据封装。

当网络接口设备，如网卡，从网络接收数据时，物理层的数据要依次经历数据链路层、网络层、传输层，最后进入应用层。每一层都要对该层的头部(有的层还有尾部)信息进行解析，最后得到用户数据。这个过程称为数据解析。

以HTTP报文为例，数据包的层次化结构与网络协议栈的关系如下图所示：



嗅探器(Sniffer)的工作原理就是利用原始套接字捕获流经主机网卡的数据包，并按上图所示封装结构，解析每一层的头部(尾部)信息，并将解析结果显示出来。

6.2 实训案例

6.2.1 基于Raw Socket的网络嗅探器

应用有关原始套接字的知识，编写一个网络嗅探器，捕获网络数据包并分析其基本信息，例如IP地址、端口号、协议类型、物理地址等。

实现简单的过滤器功能，捕获指定的数据包，例如捕获指定IP地址、指定协议的数据包。

6.2.2 程序清单

1. 包结构

```
// packet.h
// 包结构

#pragma once

#include <stdint.h>

#pragma pack(1)
//
// +-----------------+-----------------------------------------+
// | Ethernet Header |              Payload Data               |
// +-----------------+-----------------------------------------+
// |<-------------------- Ethernet Packet -------------------->|
//
// 以太网头
typedef struct tag_EthernetHeader {
    uint8_t  dstMacAddr[6]; // 目的MAC地址
    uint8_t  srcMacAddr[6]; // 源MAC地址
    uint16_t frameType;     // 帧类型
}   ETHERNET_HEADER;

// 以太网包
typedef struct tag_EthernetPacket {
    ETHERNET_HEADER ethernetHeader; // 以太网头
    uint8_t         payloadData[0]; // 载荷数据
}   ETHERNET_PACKET;
//
// +-----------------+------------+----------------------------+
// | Ethernet Header | ARP Header |        Payload Data        |
// +-----------------+------------+----------------------------+
//                   |<-------------- ARP Packet -------------->|
//
// ARP头
typedef struct tag_ArpHeader {
    uint16_t hardwareType;       // 硬件类型
    uint16_t protocolType;       // 协议类型
    uint8_t  hardwareAddrLen;    // 硬件地址长度
    uint8_t  protocolAddrLen;    // 协议地址长度
    uint16_t operationCode;      // 操作码
    uint8_t  srcHardwareAddr[6]; // 源硬件地址
    uint32_t srcProtocolAddr;    // 源协议地址
    uint8_t  dstHardwareAddr[6]; // 目的硬件地址
    uint32_t dstProtocolAddr;    // 目的协议地址
}   ARP_HEADER;

// ARP包
typedef struct tag_ArpPacket {
    ARP_HEADER arpHeader;      // ARP头
    uint8_t    payloadData[0]; // 载荷数据
}   ARP_PACKET;
//
// +-----------------+------------+----------------------------+
// | Ethernet Header | IP Header  |        Payload Data        |
// +-----------------+------------+----------------------------+
//                   |<-------------- IP Packet -------------->|
//
// IP头
typedef struct tag_IpHeader {
    uint8_t  verAndHeaderLen; // 版本(4位)和包头长度(4位)
    uint8_t  typeOfService;   // 服务类型
    uint16_t packetLen;       // 包长度
    uint16_t id;              // 标识符
    uint16_t flagsAndOffset;  // 标志(3位)和偏移(13位)
    uint8_t  ttl;             // 生存时间
    uint8_t  protocol;        // 上层协议
    uint16_t checksum;        // 校验和
    uint32_t srcIpAddr;       // 源IP地址
    uint32_t dstIpAddr;       // 目的IP地址
}   IP_HEADER;
```

```
// IP包
typedef struct tag_IpPacket {
    IP_HEADER ipHeader;       // IP头
    uint8_t   payloadData[0]; // 载荷数据
}   IP_PACKET;
//
// +-----------------+------------+-------------+--------------+
// | Ethernet Header | IP Header  | ICMP Header | Payload Data |
// +-----------------+------------+-------------+--------------+
//                               |<------ ICMP Packet ------->|
//
// ICMP头
typedef struct tag_IcmpHeader {
    uint8_t  type;     // 类型
    uint8_t  code;     // 代码
    uint16_t checksum; // 校验和
    uint16_t id;       // 标识符
    uint16_t seqNumb;  // 序号
}   ICMP_HEADER;

// ICMP包
typedef struct tag_IcmpPacket {
    ICMP_HEADER icmpHeader;     // ICMP头
    uint8_t     payloadData[0]; // 载荷数据
}   ICMP_PACKET;
//
// +-----------------+------------+-------------+--------------+
// | Ethernet Header | IP Header  | TCP Header  | Payload Data |
// +-----------------+------------+-------------+--------------+
//                               |<------- TCP Packet ------->|
//
// TCP头
typedef struct tag_TcpHeader {
    uint16_t srcPort;           // 源端口
    uint16_t dstPort;           // 目的端口
    uint32_t seqNumb;           // 序号
    uint32_t ackNumb;           // 确认序号
    uint16_t headerLenAndFlags; // 包头长度(4位)和标志(6位)
    uint16_t winSize;           // 窗口大小
    uint16_t checksum;          // 校验和
    uint16_t urp;               // 紧急指针
}   TCP_HEADER;

// TCP包
typedef struct tag_TcpPacket {
    TCP_HEADER tcpHeader;      // TCP头
    uint8_t    payloadData[0]; // 载荷数据
}   TCP_PACKET;
//
// +-----------------+------------+-------------+--------------+
// | Ethernet Header | IP Header  | UDP Header  | Payload Data |
// +-----------------+------------+-------------+--------------+
//                               |<------- UDP Packet ------->|
//
// UDP头
typedef struct tag_UdpHeader {
    uint16_t srcPort;   // 源端口
    uint16_t dstPort;   // 目的端口
    uint16_t packetLen; // 包长度
    uint16_t checksum;  // 校验和
}   UDP_HEADER;

// UDP包
typedef struct tag_UdpPacket {
    UDP_HEADER udpHeader;      // UDP头
    uint8_t    payloadData[0]; // 载荷数据
}   UDP_PACKET;

#pragma pack()
```

2. 声明RawSocket类

```
// rawsocket.h
// 声明RawSocket类

#pragma once

#include <sys/socket.h>

// 原始套接字
class RawSocket {
public:
    // 构造函数
    RawSocket(void);
    // 析构函数
    ~RawSocket(void);

    // 创建套接字
    int create(int protocol);
    // 设置网卡混杂模式
    int promisc(const char* nic) const;
    // 接收数据包
    ssize_t recv(void* buf, size_t len) const;

private:
    int sockfd; // 套接字文件描述符
};
```

3. 实现RawSocket类

```cpp
// rawsocket.cpp
// 实现RawSocket类

#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <string.h>
#include <stdio.h>

#include "rawsocket.h"

// 构造函数
RawSocket::RawSocket(void) : sockfd(-1) {}

// 析构函数
RawSocket::~RawSocket(void) {
    if(sockfd != -1)
        close(sockfd);
}

// 创建套接字
int RawSocket::create(int protocol) {
    if(sockfd == -1)
        if((sockfd = socket(PF_PACKET, SOCK_RAW, protocol)) == -1) {
            perror("socket");
            return -1;
        }

    return 0;
}

// 设置网卡混杂模式
int RawSocket::promisc(const char* nic) const {
    struct ifreq ifr;
    strncpy(ifr.ifr_name, nic, IFNAMSIZ);
    if(ioctl(sockfd, SIOCGIFFLAGS, &ifr) == -1) {
        perror("ioctl");
        return -1;
    }
    //
    // IFF_PROMISC：混杂模式，即全接收模式。在该模式下，以太网卡将接
    // 收所有的包，而不管这些包是发给谁的。缺省情况下，以太网卡会启用
    // 硬件过滤，只接收广播包及发给本网卡的包。嗅探器通过将以太网卡设
    // 置为全接收模式，接收所有到达本网卡的包，以窃听本子网内部的秘密
    //
    ifr.ifr_flags |= IFF_PROMISC;
    if(ioctl(sockfd, SIOCSIFFLAGS, &ifr) == -1) {
        perror("ioctl");
        return -1;
    }

    return 0;
}

// 接收数据包
ssize_t RawSocket::recv(void* buf, size_t len) const {
    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);

    ssize_t rlen = recvfrom(sockfd, buf, len, 0,
        (struct sockaddr*)&addr, &addrlen);
    if(rlen == -1) {
        perror("recvfrom");
        return -1;
    }
```

```
    return rlen;
}
```

4. 声明Sniffer类

```
// sniffer.h
// 声明Sniffer类

#pragma once

#include <stdint.h>

#include "rawsocket.h"

// 嗅探器
class Sniffer : public RawSocket {
public:
    // 过滤器
    typedef struct tag_filter
    {
        in_addr_t srcIpAddr; // 源IP地址
        in_addr_t dstIpAddr; // 目的IP地址
        uint32_t  protocol;  // 协议
    }   FILTER;

    // 设置x右数第i(基幺)位
    static uint32_t set(uint32_t x, int i);
    // 获取x右数第i(基幺)位
    static uint32_t get(uint32_t x, int i);

    // 构造函数
    Sniffer(FILTER filter);
    // 析构函数
    ~Sniffer(void);

    // 初始化
    int init(int protocol, const char* nic);
    // 嗅探
    void sniff(void);

private:
    // 打印内存
    void mem(const void* buf, size_t len) const;
    // 打印MAC地址
    void mac(const uint8_t* addr) const;
    // 打印IP地址
    void ip(uint32_t addr) const;

    // 解析ICMP包
    void icmp(const void* packet, size_t len) const;
    // 解析TCP包
    void tcp(const void* packet, size_t len) const;
    // 解析UDP包
    void udp(const void* packet, size_t len) const;

    // 解析ARP包
    void arp(const void* packet, size_t len) const;
    // 解析IP包
    void ip(const void* packet, size_t len) const;

    // 解析以太网包
    void ethernet(const void* packet, size_t len) const;

    static const size_t MAX_PACKET; // 最大分组长度

    FILTER   filter; // 过滤器
    uint8_t* packet; // 分组
};
```

5. 实现Sniffer类

```cpp
// sniffer.cpp
// 实现Sniffer类

#include <arpa/inet.h>
#include <iostream>
#include <iomanip>
using namespace std;

#include "sniffer.h"
#include "packet.h"

const size_t Sniffer::MAX_PACKET = 2048; // 最大分组长度

// 设置x右数第i(基幺)位
uint32_t Sniffer::set(uint32_t x, int i) {
    return x | 1 << (i - 1);
}

// 获取x右数第i(基幺)位
uint32_t Sniffer::get(uint32_t x, int i) {
    return x & 1 << (i - 1);
}

// 构造函数
Sniffer::Sniffer(FILTER filter) :
    filter(filter), packet(new uint8_t[MAX_PACKET]) {}

// 析构函数
Sniffer::~Sniffer(void) {
    delete[] packet;
}

// 初始化
int Sniffer::init(int protocol, const char* nic) {
    // 创建套接字
    if (create(protocol) == -1)
        return -1;

    // 设置网卡混杂模式
    if (promisc(nic) == -1)
        return -1;

    return 0;
}

// 嗅探
void Sniffer::sniff(void) {
    for (;;) {
        // 接收数据包
        ssize_t len = recv(packet, MAX_PACKET);
        if (len > 0)
            // 解析以太网包
            ethernet(packet, len);
    }
}

// 打印内存
void Sniffer::mem(const void* buf, size_t len) const {
    cout << "OFFSET  01-02-03-04-05-06-07-08  09-10-11-12-13-14-15-16  "
        "---- ASCII -----" << endl;

    uint8_t (*p)[16] = (uint8_t (*)[16])buf;
    size_t lines = (len + 15) / 16;

    // 逐行打印
    for (size_t i = 0; i < lines; ++i) {
        // 按六位16进制格式打印偏移地址
        cout << hex << setfill('0');
        cout << setw(6) << i * 16 << "  ";
```

```
        // 逐字节打印16进制部分
        for (size_t j = 0; j < 16; ++j) {
            if (i * 16 + j >= len)
                cout << "   ";
            else
                cout << setw(2) << (unsigned int)p[i][j] << ' ';

            if (j == 7)
                cout << ' ';
        }

        cout << ' ';

        // 逐字节打印ASCII码部分
        for (size_t j = 0; j < 16; ++j) {
            if (i * 16 + j >= len)
                break;

            if (' ' <= p[i][j] && p[i][j] <= '~')
                cout << (char)p[i][j];
            else
                cout << '.';
        }

        cout << endl;
    }
}

// 打印MAC地址
void Sniffer::mac(const uint8_t* addr) const {
    cout << hex << noshowbase << setfill('0');
    for (int i = 0; i < 6; ++i)
        cout << setw(2) << (unsigned int)addr[i] << (i < 5 ? ':' : '\n');
}

// 打印IP地址
void Sniffer::ip(uint32_t addr) const {
    struct in_addr in = {addr};
    cout << inet_ntoa(in) << endl;
}

// 解析ICMP包
void Sniffer::icmp(const void* packet, size_t len) const {
    cout << "********************************* ICMP "
        "*********************************" << endl;
    mem(packet, len);
    cout << "----------------------------------------"
        "----------------------------------" << endl;

    const ICMP_PACKET* icmpPacket = (const ICMP_PACKET*)packet;

    // 类型
    cout << "                                  Type: " <<
        dec << (unsigned int)icmpPacket->icmpHeader.type << endl;
    // 代码
    cout << "                                  Code: " <<
        dec << (unsigned int)icmpPacket->icmpHeader.code << endl;
    // 校验和
    cout << "                              Checksum: 0x" <<
        hex << ntohs(icmpPacket->icmpHeader.checksum) << endl;
    // 标识符
    cout << "                                    ID: " <<
        dec << ntohs(icmpPacket->icmpHeader.id) << endl;
    // 序号
    cout << "                       Sequence Number: " <<
        dec << ntohs(icmpPacket->icmpHeader.seqNumb) << endl;
}

// 解析TCP包
```

```cpp
void Sniffer::tcp(const void* packet, size_t len) const {
    cout << "********************************** TCP "
        "***********************************" << endl;
    mem(packet, len);
    cout << "---------------------------------------"
        "----------------------------------" << endl;

    const TCP_PACKET* tcpPacket = (const TCP_PACKET*)packet;

    // 源端口
    cout << "                          Source Port: " <<
        dec << ntohs(tcpPacket->tcpHeader.srcPort) << endl;
    // 目的端口
    cout << "                     Destination Port: " <<
        dec << ntohs(tcpPacket->tcpHeader.dstPort) << endl;
    // 序号
    cout << "                      Sequence Number: " <<
        dec << ntohl(tcpPacket->tcpHeader.seqNumb) << endl;
    // 确认序号
    cout << "     Acknowledgement Sequence Number: " <<
        dec << ntohl(tcpPacket->tcpHeader.ackNumb) << endl;
    // 包头长度(4位)和标志(6位)
    uint16_t headerLenAndFlags = ntohs(
        tcpPacket->tcpHeader.headerLenAndFlags);
    cout << "                        Header Length: " <<
        dec << 4 * (headerLenAndFlags >> 12) << endl;
    cout << "                                Flags: 0x" <<
        hex << (headerLenAndFlags & 0x3f) << endl;
    // 窗口大小
    cout << "                          Window Size: " <<
        dec << ntohs(tcpPacket->tcpHeader.winSize) << endl;
    // 校验和
    cout << "                            Checksum: 0x" <<
        hex << ntohs(tcpPacket->tcpHeader.checksum) << endl;
    // 紧急指针
    cout << "                      Urgent Pointer: " <<
        dec << ntohs(tcpPacket->tcpHeader.urp) << endl;
}

// 解析UDP包
void Sniffer::udp(const void* packet, size_t len) const {
    cout << "********************************** UDP "
        "***********************************" << endl;
    mem(packet, len);
    cout << "---------------------------------------"
        "----------------------------------" << endl;

    const UDP_PACKET* udpPacket = (const UDP_PACKET*)packet;

    // 源端口
    cout << "                          Source Port: " <<
        dec << ntohs(udpPacket->udpHeader.srcPort) << endl;
    // 目的端口
    cout << "                     Destination Port: " <<
        dec << ntohs(udpPacket->udpHeader.dstPort) << endl;
    // 包长度
    cout << "                        Packet Length: " <<
        dec << ntohs(udpPacket->udpHeader.packetLen) << endl;
    // 校验和
    cout << "                            Checksum: 0x" <<
        hex << ntohs(udpPacket->udpHeader.checksum) << endl;
}

// 解析ARP包
void Sniffer::arp(const void* packet, size_t len) const {
    cout << "********************************** ARP "
        "***********************************" << endl;
    mem(packet, len);
    cout << "---------------------------------------"
        "----------------------------------" << endl;
```

```
    const ARP_PACKET* arpPacket = (const ARP_PACKET*)packet;

    // 硬件类型
    cout << "                        Hardware Type: 0x" <<
        hex << ntohs(arpPacket->arpHeader.hardwareType) << endl;
    // 协议类型
    cout << "                        Protocol Type: 0x" <<
        hex << ntohs(arpPacket->arpHeader.protocolType) << endl;
    // 硬件地址长度
    cout << "             Hardware Address Length: " <<
        dec << (unsigned int)arpPacket->arpHeader.hardwareAddrLen <<
        endl;
    // 协议地址长度
    cout << "             Protocol Address Length: " <<
        dec << (unsigned int)arpPacket->arpHeader.protocolAddrLen <<
        endl;
    // 操作码
    cout << "                      Operation Code: 0x" <<
        hex << ntohs(arpPacket->arpHeader.operationCode) << endl;
    // 源硬件地址
    cout << "             Source Hardware Address: ";
    mac(arpPacket->arpHeader.srcHardwareAddr);
    // 源协议地址
    cout << "             Source Protocol Address: ";
    ip(arpPacket->arpHeader.srcProtocolAddr);
    // 目的硬件地址
    cout << "        Destination Hardware Address: ";
    mac(arpPacket->arpHeader.dstHardwareAddr);
    // 目的协议地址
    cout << "        Destination Protocol Address: ";
    ip(arpPacket->arpHeader.dstProtocolAddr);
}

// 解析IP包
void Sniffer::ip(const void* packet, size_t len) const {
    cout << "********************************** IP "
        "**********************************" << endl;
    mem(packet, len);
    cout << "---------------------------------------"
        "-----------------------------------" << endl;

    const IP_PACKET* ipPacket = (const IP_PACKET*)packet;

    // 版本(4位)和包头长度(4位)
    cout << "                               Version: IPv" <<
        dec << (unsigned int)(ipPacket->ipHeader.verAndHeaderLen >>
        4) << endl;
    cout << "                         Header Length: " <<
        dec << 4 * (unsigned int)(ipPacket->ipHeader.verAndHeaderLen &
        0xf) << endl;
    // 服务类型
    cout << "                     Type of Service: 0x" <<
        hex << (unsigned int)ipPacket->ipHeader.typeOfService << endl;
    // 包长度
    cout << "                         Packet Length: " <<
        dec << ntohs(ipPacket->ipHeader.packetLen) << endl;
    // 标识符
    cout << "                                    ID: " <<
        dec << ntohs(ipPacket->ipHeader.id) << endl;
    // 标志(3位)和偏移(13位)
    uint16_t flagsAndOffset = ntohs(ipPacket->ipHeader.flagsAndOffset);
    cout << "                                 Flags: 0x" <<
        hex << (flagsAndOffset >> 13) << endl;
    cout << "                                Offset: " <<
        dec << (flagsAndOffset & 0x1fff) << endl;
    // 生存时间
    cout << "                                   TTL: " <<
        dec << (unsigned int)ipPacket->ipHeader.ttl << endl;
    // 上层协议
```

```
    cout << "                            Protocol: " <<
        dec << (unsigned int)ipPacket->ipHeader.protocol << endl;
    // 校验和
    cout << "                            Checksum: 0x" <<
        hex << ntohs(ipPacket->ipHeader.checksum) << endl;
    // 源IP地址
    cout << "                   Source IP Address: ";
    ip(ipPacket->ipHeader.srcIpAddr);
    // 目的IP地址
    cout << "              Destination IP Address: ";
    ip(ipPacket->ipHeader.dstIpAddr);

    // 若过滤源IP地址，但源IP地址不符合过滤条件
    if (filter.srcIpAddr && filter.srcIpAddr !=
        ipPacket->ipHeader.srcIpAddr)
        return;

    // 若过滤目的IP地址，但目的IP地址不符合过滤条件
    if (filter.dstIpAddr && filter.dstIpAddr !=
        ipPacket->ipHeader.dstIpAddr)
        return;

    switch (ipPacket->ipHeader.protocol) {
        case 1: // ICMP
            if (get(filter.protocol, 2)) // 若过滤ICMP协议
                icmp(ipPacket->payloadData,
                    len - sizeof(ipPacket->ipHeader));
            break;

        case 6: // TCP
            if (get(filter.protocol, 3)) // 若过滤TCP协议
                tcp(ipPacket->payloadData,
                    len - sizeof(ipPacket->ipHeader));
            break;

        case 17: // UDP
            if (get(filter.protocol, 4)) // 若过滤UDP协议
                udp(ipPacket->payloadData,
                    len - sizeof(ipPacket->ipHeader));
            break;
    }
}

// 解析以太网包
void Sniffer::ethernet(const void* packet, size_t len) const {
    cout << "******************************** ETHERNET "
        "********************************" << endl;
    mem(packet, len);
    cout << "----------------------------------------"
        "--------------------------------" << endl;

    const ETHERNET_PACKET* ethernetPacket =
        (const ETHERNET_PACKET*)packet;

    // 目的MAC地址
    cout << "             Destination MAC Address: ";
    mac(ethernetPacket->ethernetHeader.dstMacAddr);
    // 源MAC地址
    cout << "                  Source MAC Address: ";
    mac(ethernetPacket->ethernetHeader.srcMacAddr);
    // 帧类型
    uint16_t frameType = ntohs(ethernetPacket->ethernetHeader.frameType);
    cout << "                          Frame Type: 0x" <<
        hex << frameType << endl;

    switch (frameType) {
        case 0x806: // ARP包
            if (get(filter.protocol, 1)) // 若过滤ARP协议
                arp(ethernetPacket->payloadData,
                    len - sizeof(ethernetPacket->ethernetHeader));
```

```
            break;

        case 0x800: // IP包
            if (filter.protocol >> 1) // 若过滤IP协议
                ip(ethernetPacket->payloadData,
                    len - sizeof(ethernetPacket->ethernetHeader));
            break;
    }
}
```

6. 测试Sniffer类

```cpp
// sniffer_test.cpp
// 测试Sniffer类

#include <unistd.h>
#include <strings.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;

#include "sniffer.h"

int main(int argc, char* argv[]) {
    Sniffer::FILTER filter; // 过滤器
    bzero(&filter, sizeof(filter));

    // 解析命令行参数
    char opt, nic[IFNAMSIZ+1] = "lo";
    while ((opt = getopt(argc, argv, "n:s:d:aitu")) != -1)
        switch (opt) {
            case 'n': // 网卡名
                strncpy(nic, optarg, IFNAMSIZ);
                break;

            case 's': // 过滤源IP地址
                filter.srcIpAddr = inet_addr(optarg);
                break;

            case 'd': // 过滤目的IP地址
                filter.dstIpAddr = inet_addr(optarg);
                break;

            case 'a': // 过滤ARP协议
                filter.protocol = Sniffer::set(filter.protocol, 1);
                break;

            case 'i': // 过滤ICMP协议
                filter.protocol = Sniffer::set(filter.protocol, 2);
                break;

            case 't': // 过滤TCP协议
                filter.protocol = Sniffer::set(filter.protocol, 3);
                break;

            case 'u': // 过滤UDP协议
                filter.protocol = Sniffer::set(filter.protocol, 4);
                break;

            default:
                cerr << "Usage: " << argv[0] <<
                    "-n <nic> -s <src_ip> -d <dst_ip> -a|i|t|u" << endl;
                return EXIT_FAILURE;
        }

    // 未设过滤协议则捕获一切协议
    if (! filter.protocol)
        filter.protocol = -1;

    Sniffer sniffer(filter); // 嗅探器
    if (sniffer.init(htons(ETH_P_ALL), nic) == -1)
        return EXIT_FAILURE;

    sniffer.sniff(); // 嗅探

    return EXIT_SUCCESS;
}
```

7. 测试Sniffer类构建脚本

```
# sniffer_test.mak
# 测试Sniffer类构建脚本

PROJ   = sniffer_test
OBJS   = sniffer_test.o sniffer.o rawsocket.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.

$(PROJ): $(OBJS)
        $(LINK) $^ -o $@

.cpp.o:
        $(CXX) $(CFLAGS) $^

clean:
        $(RM) $(PROJ) $(OBJS)
```

6.3 扩展提高

6.3.1 通过libpcap库捕获数据包

1. libpcap简介

对数据包的捕获也可以借助Linux下的libpcap开发包来实现。该开发包由加州大学伯克利分校的Van Jacobson、Craig Leres和Steven McCanne合作开发，它提供了丰富的API函数，可以帮助程序员快速开发数据包捕获软件。

libpcap的官方网址：http://www.tcpdump.org

libpcap具有如下特点：

- 对数据包捕获功能进行封装，易于使用
- 可以设置各种复杂的组合过滤规则，捕获用户感兴趣的数据包
- 过滤模块在内核层次实现，效率非常高

很多著名的网络抓包工具，如tcpdump，和网络入侵检测工具，如snort，都是基于libpcap开发的。

2. libpcap的功能

1) 捕获数据包

使用libpcap可以方便、高效地捕获网络数据包。

2) 过滤数据包

libpcap可以在内核层次对数据包进行过滤，不仅效率高而且过滤规则非常详细，可以进行各种复杂的组合，实现强大的过滤功能。

3) 分析数据包

libpcap在捕获数据包的同时还提供了一些辅助信息，如捕获时间、数据包长度等，可以帮助开发者更好地分析数据包。

4) 存储数据包

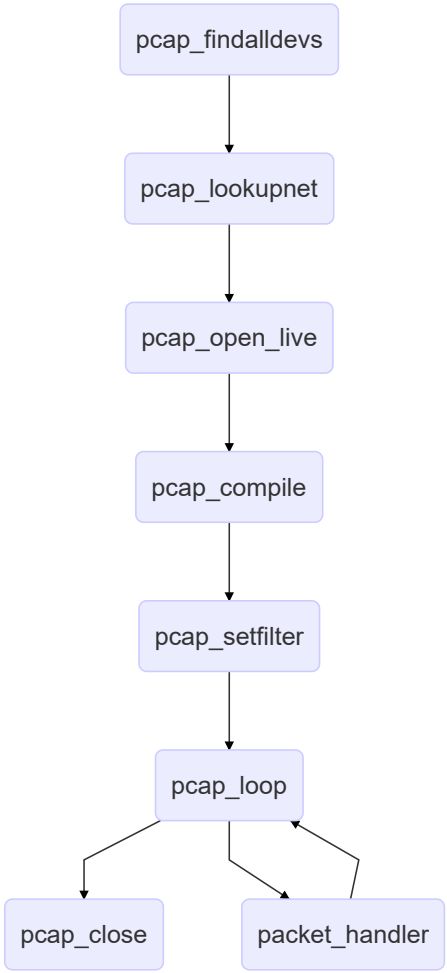
libpcap可将捕获到的数据包存储到本地，甚至可以离线方式对本地文件中的数据包进行分析。

3. libpcap捕获分析数据包的步骤

使用libpcap对网络数据包进行捕获和分析的步骤如下：

- 获取设备列表：调用pcap_findalldevs函数
- 获取网络地址和子网掩码：调用pcap_lookupnet函数
- 打开指定的设备：调用pcap_open_live函数
- 编译过滤规则：调用pcap_compile函数
- 设置过滤规则：调用pcap_setfilter函数
- 捕获分析数据包：调用pcap_loop函数
- 关闭设备：调用pcap_close函数

如下图所示：



4. libpcap捕获分析数据包的关键程序

1) 获取设备列表

```
if (pcap_findalldevs(&devices, errbuf) == -1) {
    fprintf(stderr, "pcap_findalldevs: %s\n", errbuf);
    return -1;
}
```

2) 获取网络地址和子网掩码

```
if (pcap_lookupnet(devname, &net_ip, &net_mask, errbuf) == -1) {
    fprintf(stderr, "pcap_lookupnet: %s\n", errbuf);
    return -1;
}
```

3) 打开指定的设备

```
if ((dev_handle_pcap = pcap_open_live(devname, BUFSIZ, 1, 0, errbuf)) == NULL) {
    fprintf(stderr, "pcap_open_live: %s\n", errbuf);
    return -1;
}
```

4) 编译过滤规则

```
if (pcap_compile(dev_handle_pcap, &bpf_filter, bpf_filter_string, 0, net_ip) == -1) {
    fprintf(stderr, "pcap_compile: %s\n", bpf_filter_string);
    return -1;
}
```

5) 设置过滤规则

```
if (pcap_setfilter(dev_handle_pcap, &bpf_filter) == -1) {
    fprintf(stderr, "pcap_setfilter: %s\n", bpf_filter_string);
    return -1;
}
```

6) 过滤表达式

过滤表达式相当于一种微型语言，亦有其特定的语法：

- 过滤表达式支持逻辑运算，如and、or、not等，也可以通过小括号人为地规定优先级
- 基于协议的过滤可以使用诸如ip、arp、rarp、tcp和udp等协议限定符
- 基于MAC地址的过滤使用ether限定符：
  - 过滤源MAC地址：ether src 00:E0:4C:E0:38:88
  - 过滤目的MAC地址：ether dst 00:E0:4C:E0:38:88
  - 同时过滤源和目的MAC地址：ether host 00:E0:4C:E0:38:88
- 基于IP地址的过滤使用host限定符：
  - 过滤源IP地址：src host 192.168.1.27
  - 过滤目的IP地址：dst host 192.168.1.27
  - 同时过滤源和目的IP地址：host 192.168.1.27
- 基于端口的过滤使用port限定符：
  - 过滤特定端口：port 80

例如：捕获ARP和ICMP包。

```
arp or icmp
```

例如：捕获以192.168.1.27为源或目的地址的80端口TCP包。

```
(ip and tcp) and (host 192.168.1.27) and (port 80)
```

例如：捕获在主机192.168.1.27和192.168.1.28之间传递的所有UDP包。

```
(ip and udp) and ((src host 192.168.1.27 and dst host 192.168.1.28) or
                  (src host 192.168.1.28 and dst host 192.168.1.27))
```

例如：捕获从MAC地址00:E0:4C:E0:38:88发往MAC地址00:50:56:C0:D2:F6的所有ARP包。

```
arp and (ether src 00:E0:4C:E0:38:88 and ether dst 00:50:56:C0:D2:F6)
```

7) 捕获分析数据包

```
pcap_loop(dev_handle_pcap, -1, packet_handler, NULL);
```

pcap_loop函数每捕获到一个数据包，就会将其作为参数传递给程序设计者自己定义的packet_handler函数，该函数负责对数据包的具体处理，其原型如下：

```
void packet_handler(u_char* argument, const struct pcap_pkthdr* packet_header,
    const u_char* packet_content);
```

其中，packet_header参数为libpcap在捕获数据包时附加的辅助信息，如时间戳、包长度等，packet_content参数为捕获到的数据包内容。

8) 关闭设备

```
pcap_close(dev_handle_pcap);
```

5. libpcap注意事项

使用libpcap库需要添加-lpcap链接选项。

libpcap只能捕获数据包而不能发送数据包。Linux下的另一个开发工具libnet可以填充和发送数据包。libpcap的Windows版本winpcap既可捕获亦可发送数据包。

6.3.2　通过tcpdump命令捕获数据包

1. tcpdump简介

tcpdump是一款功能十分强大的网络数据包捕获分析工具：

- 支持针对协议栈层次、协议种类、主机、网络和端口的数据包过滤
- 支持基于正则表达式形式的过滤策略描述
- 主要用于对网络的分析、维护、统计和检测，如定位网络瓶颈、跟踪流量变化等
- 在源代码公开的同时提供应用编程接口(API)，支持二次开发
- 大多数Linux系统都缺省提供对tcpdump的集成，无需单独安装
- tcpdump需要网卡工作于混杂模式，必须获得root权限才能运行

2. tcpdump命令

tcpdump命令的语法如下：

```
tcpdump [-aAbdDefhHIJKlLnNOpqStuUvxX#][-B size][-c count]
        [-C file_size][-E algo:secret][-F file][-G seconds]
        [-i interface][-j tstamptype][-m module][-M secret][--number]
        [-Q in|out|inout][-r file][-s snaplen][--time-stamp-precision precision]
        [--immediate-mode][-T type][--version][-V file][-w file][-W filecount]
        [-y datalinktype][-z postrotate-command][-Z user][expression]
```

3. tcpdump选项

tcpdump命令的选项众多，其中部分常用选项如下表所示：

选项	含义
-a	将网络地址和广播地址转换成名字
-dd	将匹配的数据包以C语言代码的格式输出
-ddd	将匹配的数据包以十进制形式输出
-e	输出数据链路层包头
-f	将外部互联网地址以数字形式输出

-l	对标准输出进行缓冲，可在捕获的同时查看数据包
-n	不把网络地址转换为主机名
-N	不输出主机名中的域名后缀，如tcpdump.org只输出tcpdump
-O	不运行数据包匹配模板的优化器
-p	不将网卡设置为混杂模式
-q	快速输出，只输出较少的协议信息
-S	将TCP序号以绝对值而非相对值的形式输出
-t	不在输出的每一行打印时间戳
-u	输出未解码的NFS句柄
-v	输出详细信息，如IP包头中的服务类型、生存时间(TTL)等
-vv	输出更详细的信息
-vvv	输出最详细的信息
-c count	指定数据包监听上限，达到此数量即退出tcpdump
-C file_size	指定数据包文件上限，达到此字节数即不再写文件
-E algo:secret	用指定的算法解密IPSec ESP数据包
-F file	从指定的文件中读取用于过滤数据包的正则表达式，忽略命令行中的正则表达式
-i interface	指定监听的网卡
-m module	打开指定的SMI MIB组件(需要系统支持libsmi)
-r file	从指定的文件中读取数据包
-s snaplen	从每个数据包中读取最开始的snaplen字节，而非默认的68字节
-T type	将捕获的数据包解析为指定的类型，如rpc、rtp、rtcp、vat、wb等
-w file	将捕获的数据包写入指定的文件

4. tcpdump的正则表达式
tcpdump通过正则表达式过滤数据包。如果没有指定正则表达式，则捕获全部数据包，否则只捕获那些满足正则表达式规则的数据包。

tcpdump的正则表达式包含三种主要关键字：

- 类型关键字：host、net、port，缺省为host。例如：
  - tcpdump host 192.168.1.27
  - 捕获IP地址为192.168.1.27的主机收发的所有数据包
- 方向关键字：src、dst、src or dst、src and dst。缺省为src or dst。例如：
  - tcpdump dst net 192.168.1.1
  - 捕获目标网络地址为192.168.1.1的所有数据包
- 协议关键字：ether、fddi、tr、ip、ip6、arp、rarp、decnet、tcp、udp。例如：
  - tcpdump udp
  - 捕获所有UDP协议的数据包

tcpdump的正则表达式还包含其它一些关键字：

- gateway、broadcast
- less、greater

- and(&&)、or(||)、not(!)

通过这些关键字的灵活组合，可以表达各种复杂的过滤条件。

5. tcpdump简单示例

例如：捕获ARP和UDP包。

```
$ tcpdump arp or udp
```

例如：捕获除主机alice之外的所有IP包。

```
$ tcpdump ip host not alice
```

例如：捕获主机192.168.1.27收发的所有TELNET包。

```
$ tcpdump tcp and host 192.168.1.27 and port 23
```

例如：捕获以192.168.1.27为源地址，以192.168.1.28为目的地址的80端口TCP包。

```
$ tcpdump tcp and src host 192.168.1.27 and dst host 192.168.1.28 and port 80
```

例如：捕获在主机192.168.1.27和192.168.1.28之间传递的所有TCP包。

```
$ tcpdump \(ip and tcp\) and \(\(src host 192.168.1.27 and dst host 192.168.1.28\) or
                               \(src host 192.168.1.28 and dst host 192.168.1.27\)\)
```

6. tcpdump注意事项

tcpdump的正则表达式中如果包含括号，前面必须加上反斜杠“\”。

达内集团◇C++教研部◇闵卫