计算机网络体系结构实验报告

选课序号 42 专业 网络工程 班级 2班 学号 2220193626 学生姓名 李子毅 课程名称计算机网络体系结构实验 实验时间 2021.7.21

实验名称 数据包分析实验 实验成绩

1. 实验内容：通过编程编写一个程序来实现类似wireshark的各项功能，实现对数据包协议的分析，对数据包的捕获、分析、保存、发送。
2. 实验目的：通过本次实验提升对于网络协议的理解，提升文献的查找能力和代码的编写能力，熟悉数据包分析的实现过程以及多线程并发操作的处理逻辑。

三、实验要求：

（1）掌据原始套接字（raw socket）或Winpcap的使用方法；掌握数据包捕获和数据包分析的相关知识；

（2）实现数据包的捕获、分析，并保存捕获的数据包；

（3）根据捕获的信息进行统计分析，实时流量监测等；

（4）改写数据包内容并发送；

（5）交互界面良好；

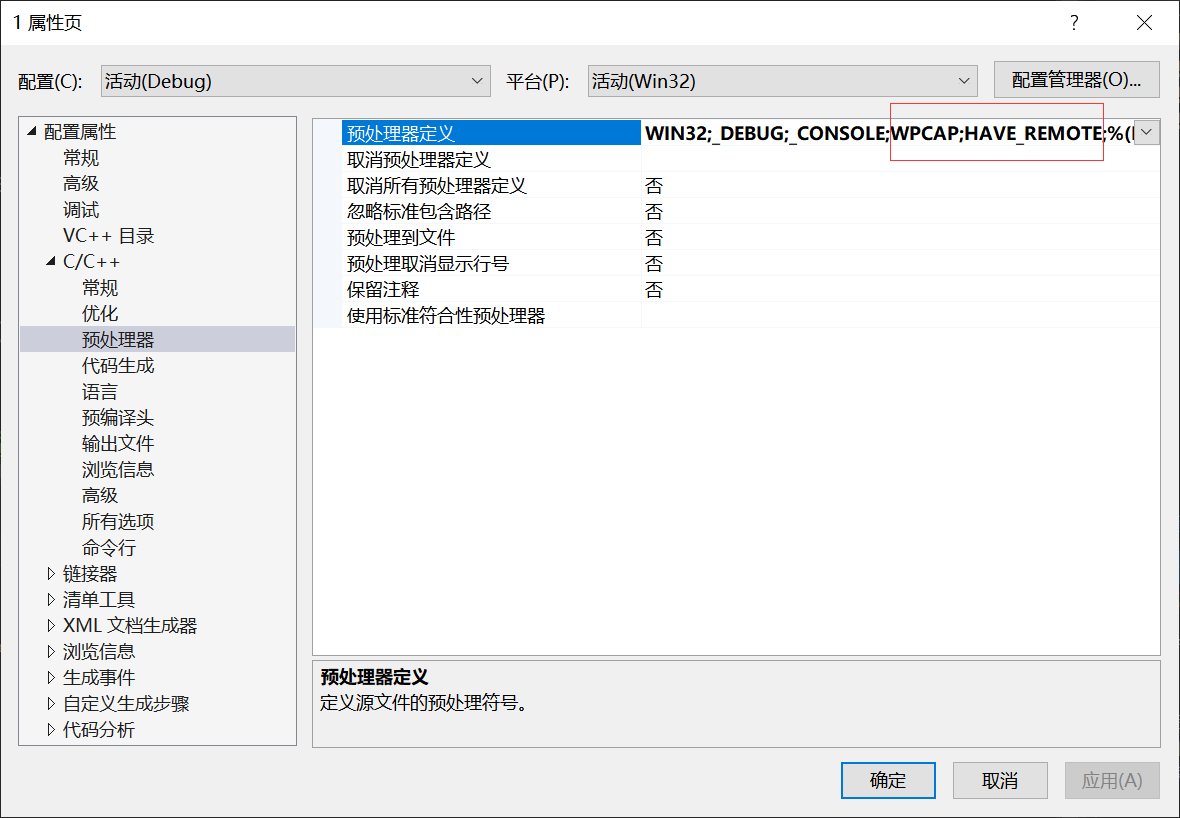
（6）提高综合运用文献的能力。

四、系统环境、开发工具简介（写清楚开发环境配置步骤，若有需要，可截图说明）

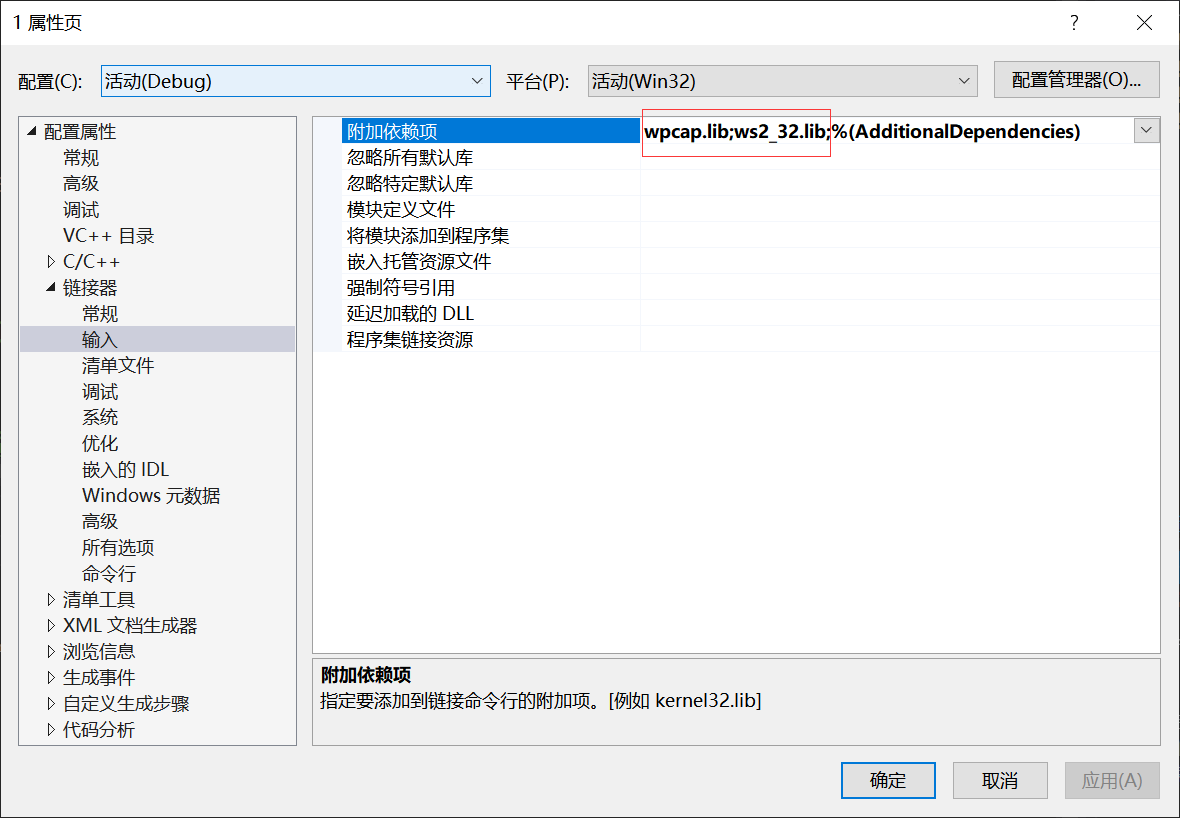
1.系统环境和开发工具简介：  
选择的系统环境是windows 10，选择的开发工具visual studio 2019，这是一个由微软公司开发的适合简单工程的IDE，采用的数据包有winpcap和libnet包配置过程如下：

A.首先从官网下载winpcap包以及winpcap安装程序

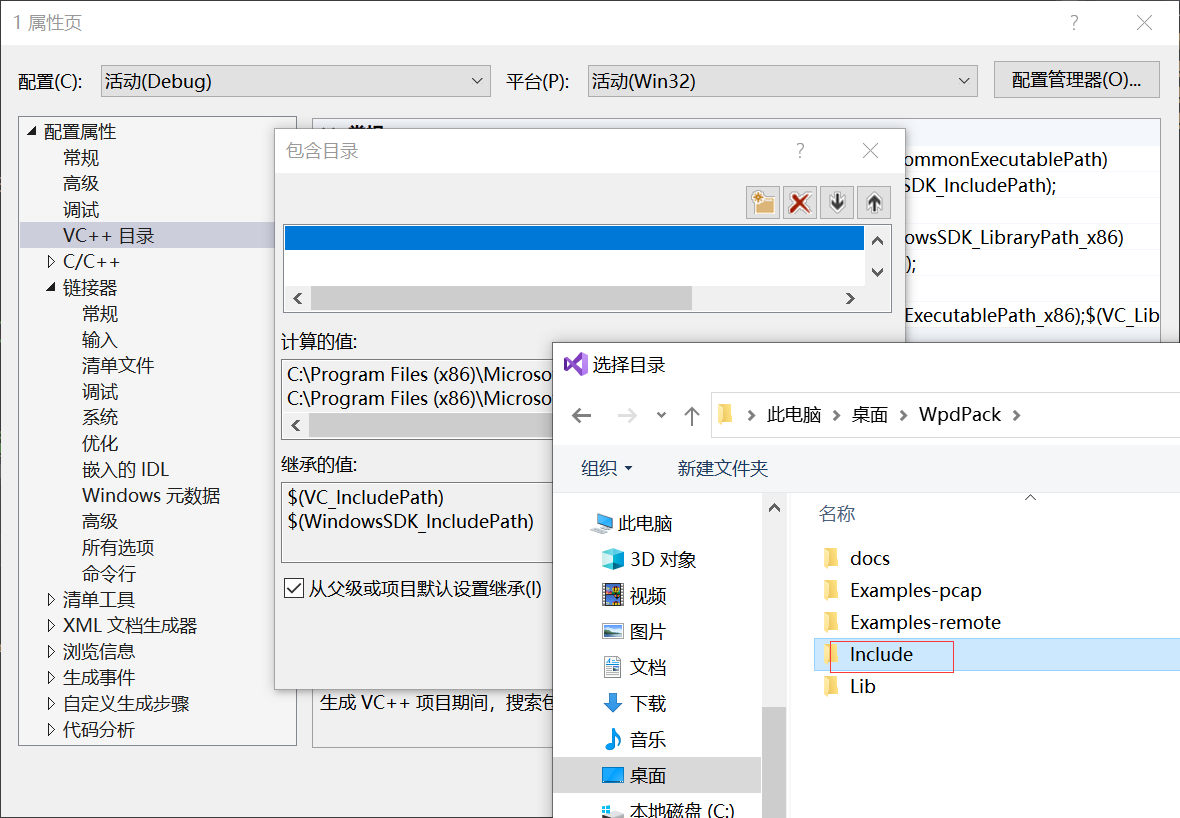
B.在项目属性中添加宏定义WPCAP和HAVE\_REMOTE：



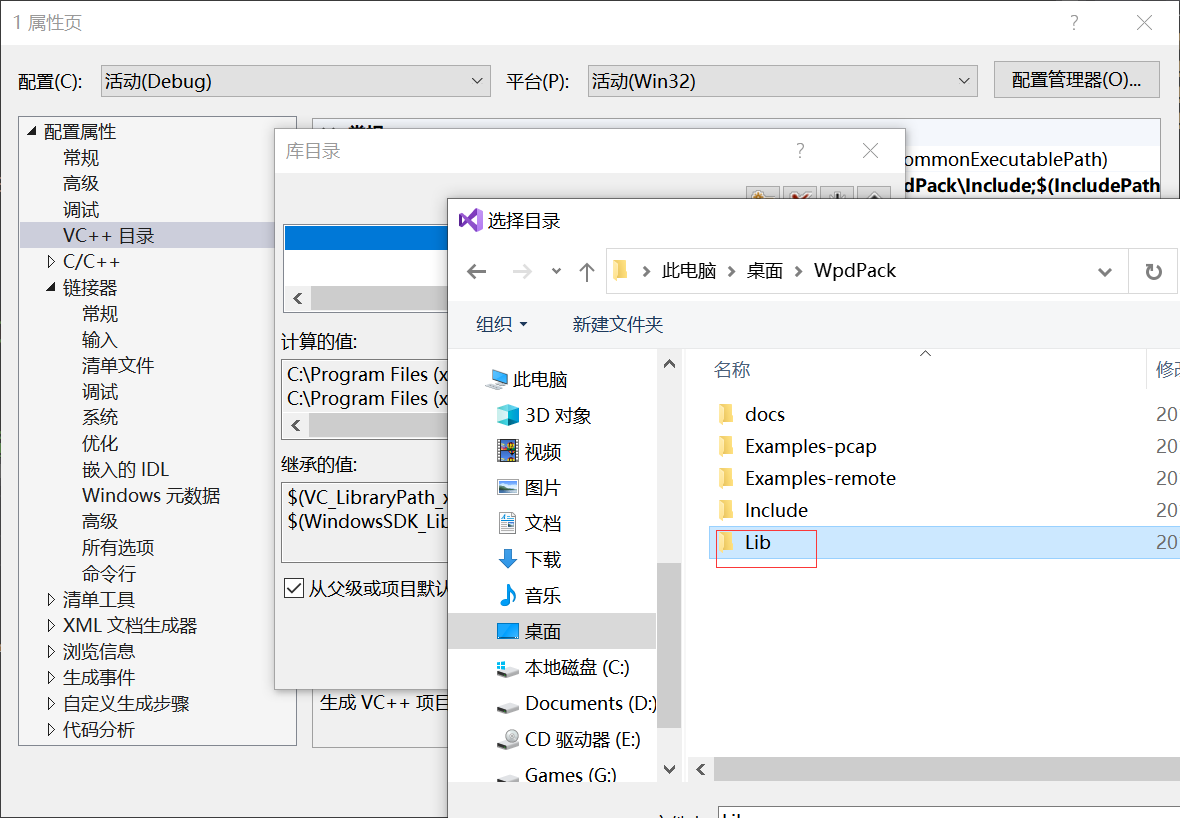
1. 添加两个附加库wpcap.lib以及ws2\_32.lib

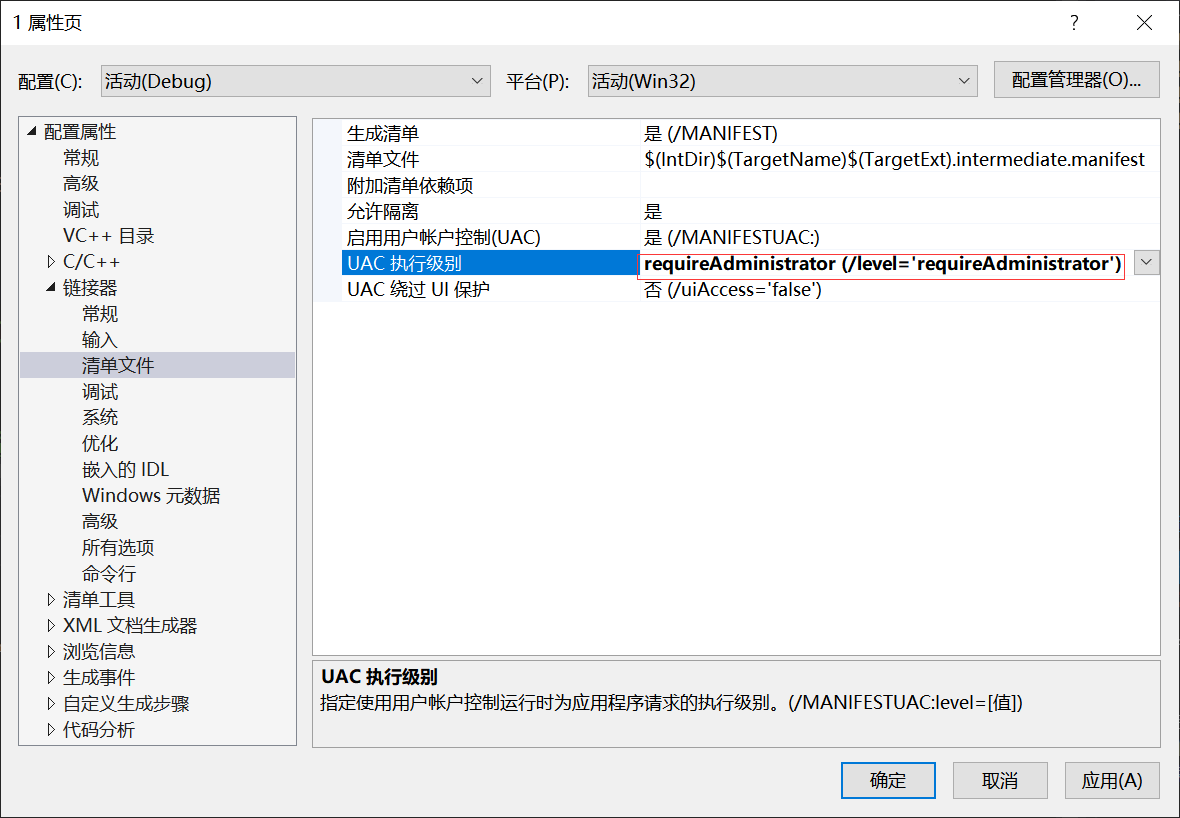


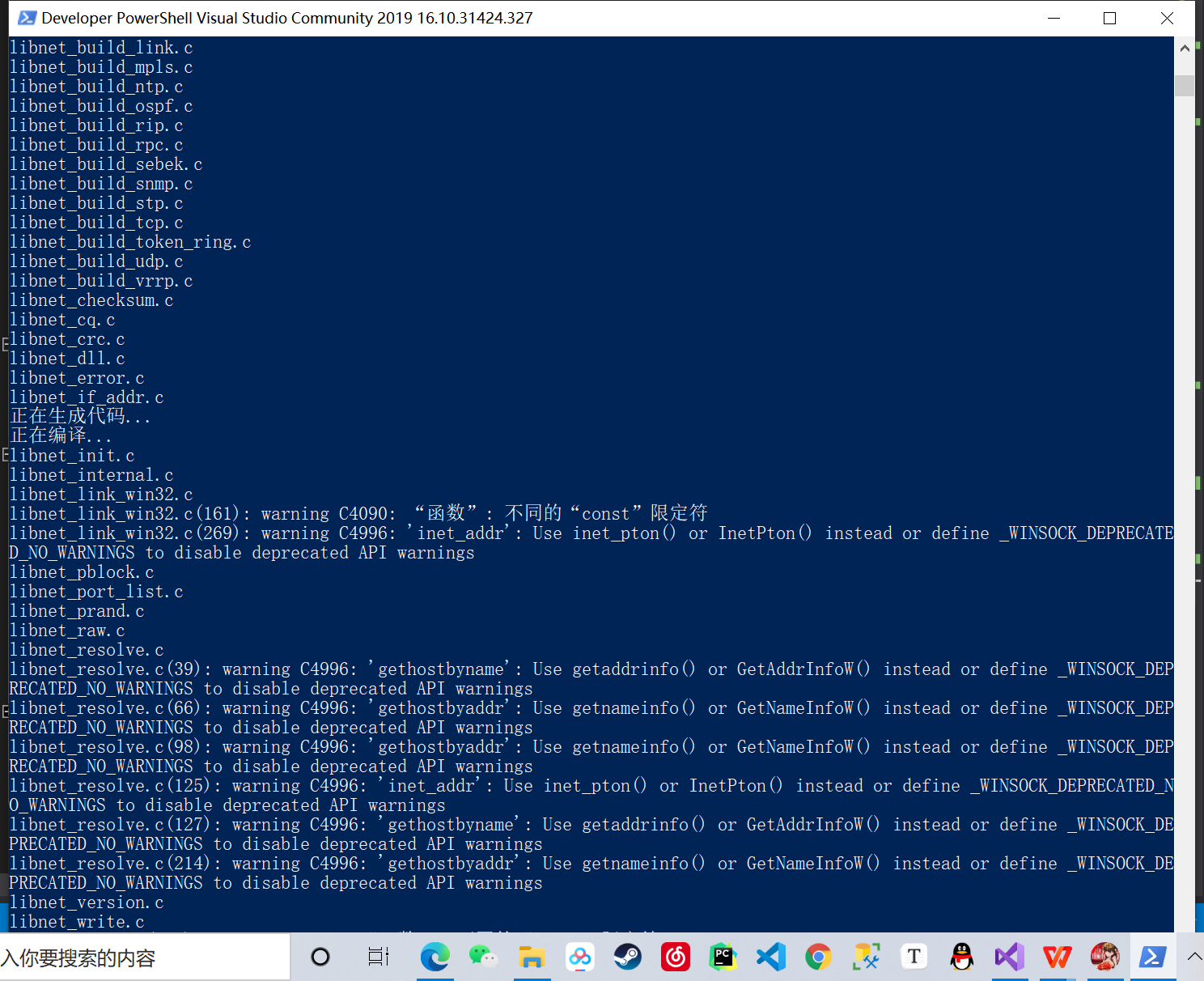
1. 添加包含路径：



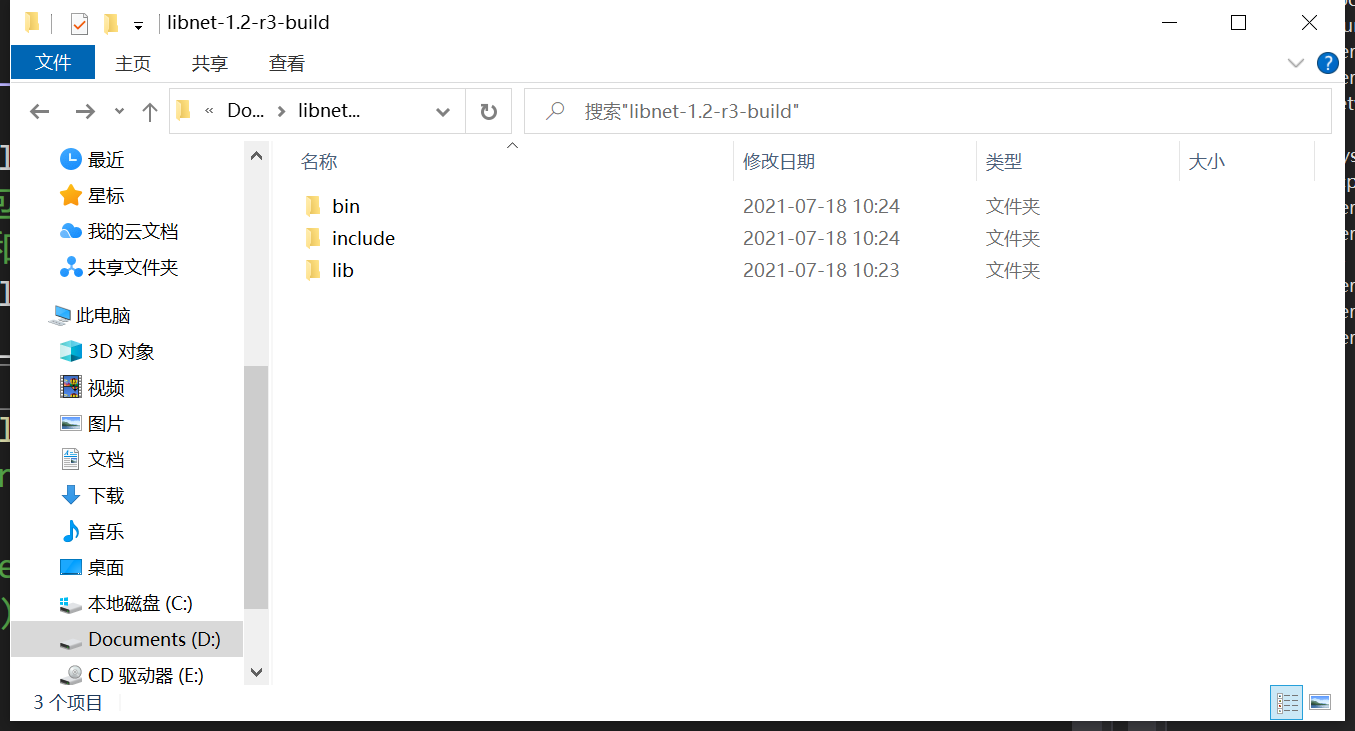
1. 添加库路径：



1. 设置UAC，以便得到管理员权限，此步骤用于方便：  
   
2. 在winpcap的安装根目录下解压从github下载的最新版libnet开发包，然后利用vs的命令行转移到含有msvc.dat的文件下，并编译运行，



1. 然后创建编译好的文件夹，将dll和lib以及include文件都加入，这样这个库也就可以用了，接着便是采用同样的手法在项目中加入这个库，不再展示：



1. 实验原理

1.数据传输的原理

数据在网络上是以很小的称为帧（Frame）的单位传输的，帧由几部分组成，不同的部分执行不同的功能。帧通过特定的称为网络驱动程序的软件进行成型，然后通过网卡发送到网线上，通过网线到达它们的目的机器，在目的机器的一端执行相反的过程。接收端机器的以太网卡捕获到这些帧，并告诉操作系统帧已到达，然后对其进行存储。

每一个在局域网（LAN）上的工作站都有其硬件地址(MAC地址)，这些地址惟一地表示了网络上的机器（这一点与Internet地址系统比较相似）。当用户发送一个数据包时，这些数据包就会发送到LAN上所有可用的机器。

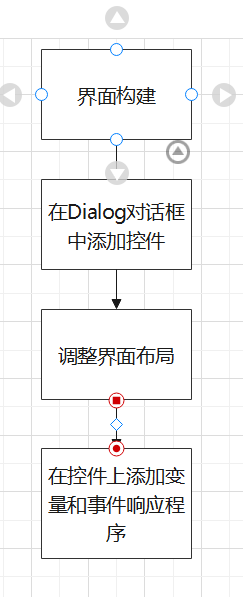
如果使用Hub/即基于共享网络的情况下，网络上所有的机器都可以“听”到通过的流量，但对不属于自己的数据包则不予响应（换句话说，工作站A不会捕获属于工作站B的数据，而是简单地忽略这些数据）。如果某个工作站的网络接口处于混杂模式，那么它就可以捕获网络上所有的数据包和帧。

但是现代网络常常采用交换机作为网络连接设备枢纽，在通常情况下，交换机不会让网络中每一台主机侦听到其他主机的通讯，因此Sniffer技术在这时必须结合网络端口镜像技术进行配合。而衍生的安全技术则通过ARP欺骗来变相达到交换网络中的侦听。

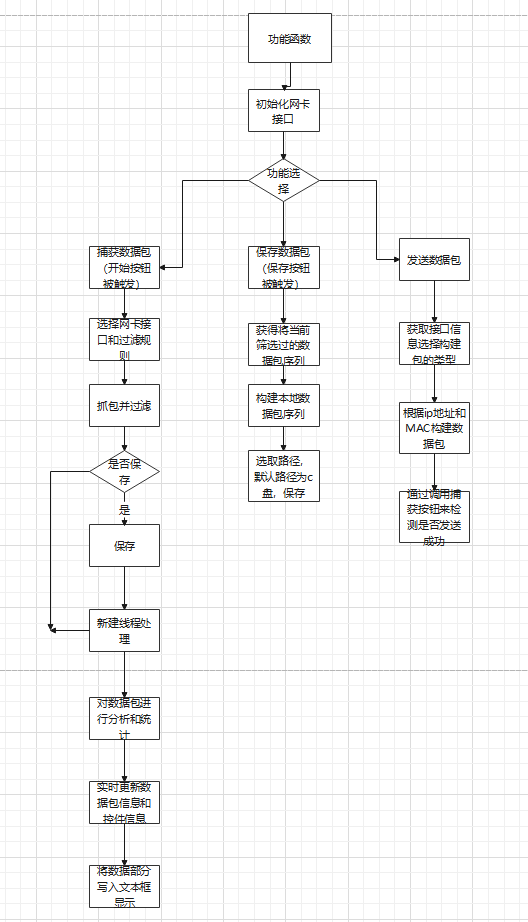
2.Sniffer(嗅探器)的原理

Sniffer程序是一种利用以太网的特性把网络适配卡（NIC，一般为以太网卡）置为杂乱（promiscuous）模式状态的工具，一旦网卡设置为这种模式，它就能接收传输在网络上的每一个信息包。程序设计工作在网络环境中的底层，它会拦截所有的正在网络上传送的数据，并且通过相应的软件处理，可以实时分析这些数据的内容，进而分析所处的网络状态和整体布局。

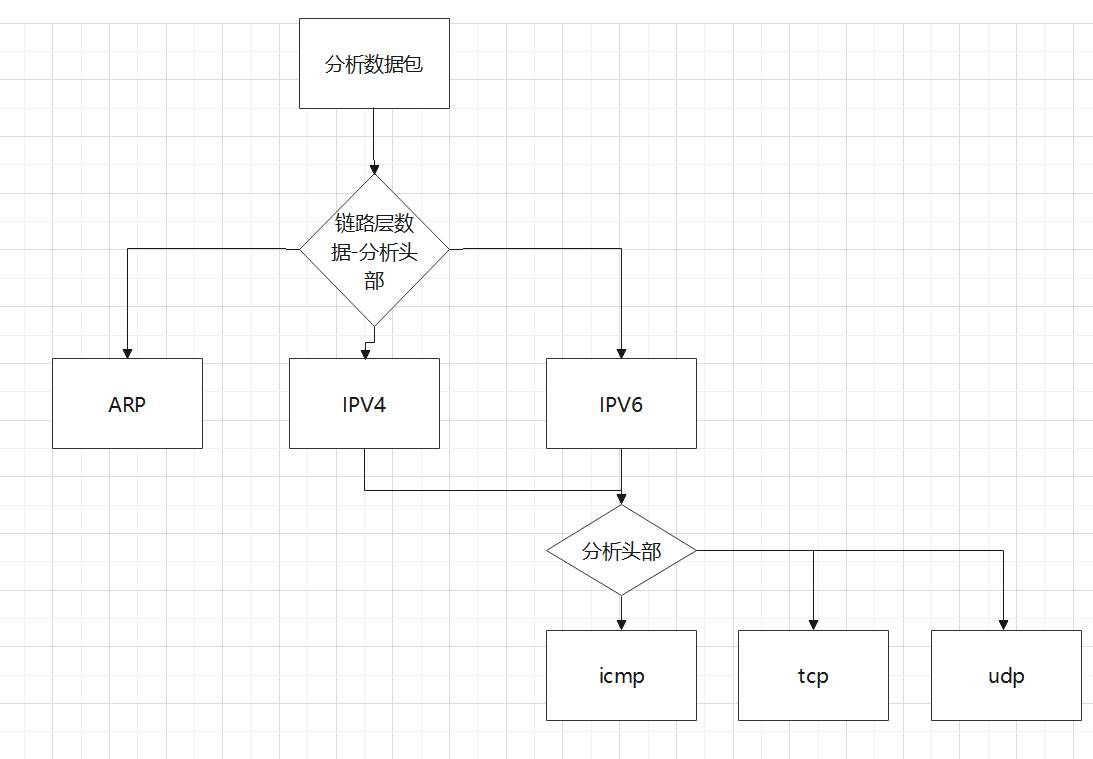
1.程序设计思想（数据流图、算法、程序流程图等）  
采用winpcap作为底层后端抓包逻辑构建的工具，然后MFC作为前端来展示，初始的界面构建流程如下：



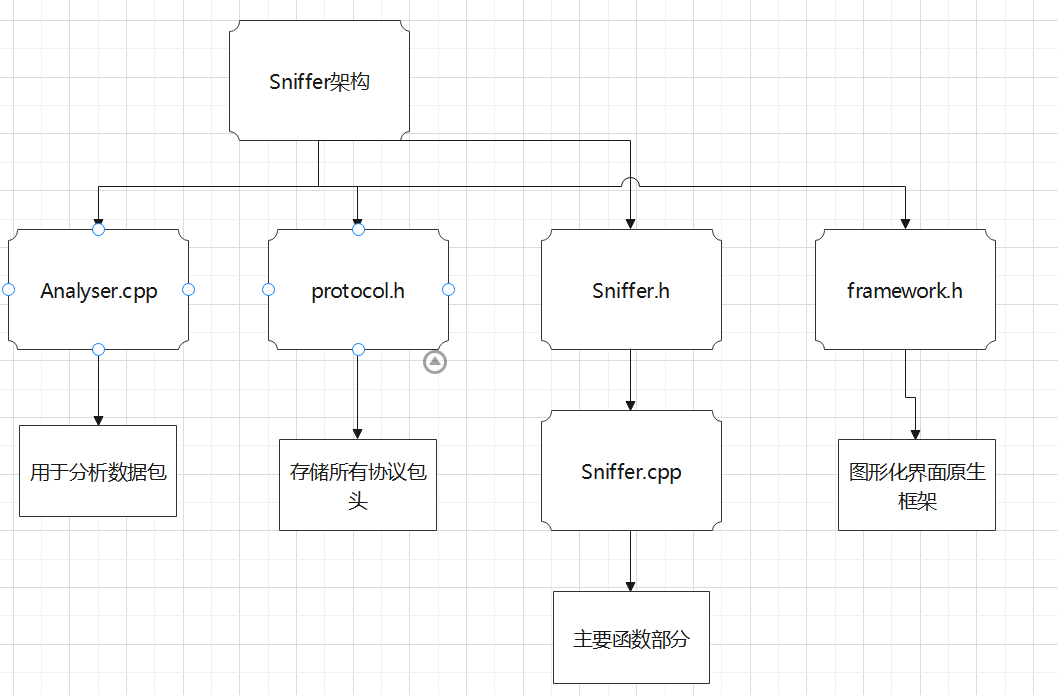
这就是前端界面的构建流程，接下来是后端的几个功能函数的介绍：



对不同包的数据结构的解析比较简单，在官方文档中可以看到几个定义，其余的自己想要的数据包只需要照着来做即可，这里不再展示，最后我展示一下分析数据报的流程：



2.主要的程序模块



捕捉函数：

//捕捉函数

int CSnifferDlg::Sniffer\_startCap()

{

int if\_index, filter\_index, count;

u\_int netmask;//获取子网掩码

struct bpf\_program fcode;//用于储存过滤规则，仅仅用于pcap\_compile

char packet\_filter[] = "";

//首先初始化

Sniffer\_initCap();

//获得接口和过滤器索引,用于找到指定的包

if\_index = this->m\_ComboBox.GetCurSel();

filter\_index = this->m\_ComboBoxRule.GetCurSel();

//错误处理

if(0 == if\_index || CB\_ERR == if\_index)

{

MessageBox(\_T("请选择一个合适的网卡接口"));

return -1;

}

if (CB\_ERR == filter\_index)

{

MessageBox(\_T("过滤器选择错误"));

return -1;

}

/\*获得选中的网卡接口\*/

d = alldevs;

for (count = 0; count < if\_index - 1; count++)

{

d = d->next;

}

/\* 打开适配器进行抓包 \*/

if ((adhandle = pcap\_open(d->name, // 设备名

65536, // 要捕捉的数据包的部分

// 65535保证能捕获到不同数据链路层上的每个数据包的全部内容

PCAP\_OPENFLAG\_PROMISCUOUS, // 混杂模式

1000, // 读取超时时间

NULL, // 远程机器验证

errbuf // 错误缓冲池

)) == NULL)

{

MessageBox(\_T("无法打开接口" + CString(d->description)));

pcap\_freealldevs(alldevs);//释放内存

return -1;

}

/\* 检查数据链路层，为了简单，我们只考虑以太网 \*/

if (pcap\_datalink(adhandle) != DLT\_EN10MB)

{

MessageBox(\_T("该网络并不是以太网" ));

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

return -1;

}

if (d->addresses != NULL)

/\* 获得接口第一个地址的掩码 \*/

netmask = ((struct sockaddr\_in\*)(d->addresses->netmask))->sin\_addr.S\_un.S\_addr;

else

/\* 如果接口没有地址，那么假设一个C类的掩码 \*/

netmask = 0xffffff;

//编译过滤器

if (filter\_index == 0)//如果不过滤

{

if (pcap\_compile(adhandle, &fcode, packet\_filter, 1, netmask) < 0)

{

MessageBox(\_T("无法编译过滤器"));

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

return -1;

}

}

else

{

CString str;

char\* filter;//一个过滤器

int len, x;

this->m\_ComboBoxRule.GetLBText(filter\_index, str);//获取过滤器标号中的内容

len = str.GetLength() + 1;

filter = (char\*)malloc(len);//为过滤器分配内容

for (x = 0; x < len; x++)

{

filter[x] = str.GetAt(x);

}

//用过滤器过滤数据包

if (pcap\_compile(adhandle, &fcode, filter, 1, netmask) < 0)

{

MessageBox(\_T("语法错误，无法编译过滤器"));

pcap\_freealldevs(alldevs);

return -1;

}

}

//设置过滤器

if (pcap\_setfilter(adhandle, &fcode) < 0)

{

MessageBox(\_T("过滤器设置有误"));

/\* 释放设备列表 \*/

pcap\_freealldevs(alldevs);

return -1;

}

/\*设置过滤包存储路径\*/

CFileFind file;

char thistime[30];

struct tm\* ltime;

memset(filepath, 0, 512);

memset(filename, 0, 64);

if (!file.FindFile(\_T("保存数据")))

{

CreateDirectory(\_T("保存数据"), NULL);

}

time\_t nowtime;//存储当前时间

time(&nowtime);

ltime = localtime(&nowtime);

strftime(thistime, sizeof(thistime), "%Y%m%d %H%M%S", ltime);

strcpy(filepath, "SavedData\\");

strcat(filename, thistime);

strcat(filename, ".pcap");

strcat(filepath, filename);

dumpfile = pcap\_dump\_open(adhandle, filepath);

if (dumpfile == NULL)

{

MessageBox(\_T("文件创建错误！"));

return -1;

}

pcap\_freealldevs(alldevs);

//接收数据，新建线程处理

LPDWORD threadCap = NULL;

m\_ThreadHandle = CreateThread(NULL, 0, Sniffer\_CapThread, this, 0, threadCap);

if (m\_ThreadHandle == NULL)

{

int code = GetLastError();

CString str;

str.Format(\_T("创建线程错误，代码为%d."), code);

MessageBox(str);

return -1;

}

return 1;

创建线程：

//线程函数(copy)

DWORD WINAPI Sniffer\_CapThread(LPVOID lpParameter)

{

int res, nItem;

struct tm\* ltime;

CString timestr, buf, srcMac, destMac;

time\_t local\_tv\_sec;

struct pcap\_pkthdr\* header; //数据包头

const u\_char\* pkt\_data = NULL, \* pData = NULL; //网络中收到的字节流数据

u\_char\* ppkt\_data;

CSnifferDlg\* pthis = (CSnifferDlg\*)lpParameter;

if (NULL == pthis->m\_ThreadHandle)

{

MessageBox(NULL, \_T("线程句柄错误"), \_T("提示"), MB\_OK);

return -1;

}

//读取一个数据包

while ((res = pcap\_next\_ex(pthis->adhandle, &header, &pkt\_data)) >= 0)

{

if (res == 0) //超时

continue;

struct datapkt\* data = (struct datapkt\*)malloc(sizeof(struct datapkt));

memset(data, 0, sizeof(struct datapkt));

if (NULL == data)

{

MessageBox(NULL, \_T("空间已满， 无法接收新的数据包"), \_T("Error"), MB\_OK);

return -1;

}

//分析出错或所接收数据包不在处理范围内

if (analyze\_frame(pkt\_data, data, &(pthis->npacket)) < 0)

continue;

//将数据包保存到打开的文件中

if (pthis->dumpfile != NULL)

{

pcap\_dump((unsigned char\*)pthis->dumpfile, header, pkt\_data);

}

//更新各类数据包计数

pthis->Sniffer\_updatePacket();

//将本地化后的数据装入一个链表中，以便后来使用

ppkt\_data = (u\_char\*)malloc(header->len);

memcpy(ppkt\_data, pkt\_data, header->len);

pthis->m\_localDataList.AddTail(data);

pthis->m\_netDataList.AddTail(ppkt\_data);

/\*预处理，获得时间、长度\*/

data->len = header->len; //链路中收到的数据长度

local\_tv\_sec = header->ts.tv\_sec;

ltime = localtime(&local\_tv\_sec);

data->time[0] = ltime->tm\_year + 1900;

data->time[1] = ltime->tm\_mon + 1;

data->time[2] = ltime->tm\_mday;

data->time[3] = ltime->tm\_hour;

data->time[4] = ltime->tm\_min;

data->time[5] = ltime->tm\_sec;

/\*为新接收到的数据包在listControl中新建一个item\*/

buf.Format(\_T("%d"), pthis->npkt);

nItem = pthis->m\_listCtrl.InsertItem(pthis->npkt, buf);

/\*显示时间戳\*/

timestr.Format(\_T("%d/%d/%d %d:%d:%d"), data->time[0],

data->time[1], data->time[2], data->time[3], data->time[4], data->time[5]);

pthis->m\_listCtrl.SetItemText(nItem, 1, timestr);

/\*显示长度\*/

buf.Empty();

buf.Format(\_T("%d"), data->len);

pthis->m\_listCtrl.SetItemText(nItem, 2, buf);

/\*显示源MAC\*/

buf.Empty();

buf.Format(\_T("%02X-%02X-%02X-%02X-%02X-%02X"), data->ethh->src[0], data->ethh->src[1],

data->ethh->src[2], data->ethh->src[3], data->ethh->src[4], data->ethh->src[5]);

pthis->m\_listCtrl.SetItemText(nItem, 3, buf);

/\*显示目的MAC\*/

buf.Empty();

buf.Format(\_T("%02X-%02X-%02X-%02X-%02X-%02X"), data->ethh->dest[0], data->ethh->dest[1],

data->ethh->dest[2], data->ethh->dest[3], data->ethh->dest[4], data->ethh->dest[5]);

pthis->m\_listCtrl.SetItemText(nItem, 4, buf);

/\*获得协议\*/

pthis->m\_listCtrl.SetItemText(nItem, 5, CString(data->pktType));

/\*获得源IP\*/

buf.Empty();

if (0x0806 == data->ethh->type)

{

buf.Format(\_T("%d.%d.%d.%d"), data->arph->ar\_srcip[0],

data->arph->ar\_srcip[1], data->arph->ar\_srcip[2], data->arph->ar\_srcip[3]);

}

else if (0x0800 == data->ethh->type) {

struct in\_addr in;

in.S\_un.S\_addr = data->iph->saddr;

buf = CString(inet\_ntoa(in));

}

else if (0x86dd == data->ethh->type) {

int n;

for (n = 0; n < 8; n++)

{

if (n <= 6)

buf.AppendFormat(\_T("%02x:"), data->iph6->saddr[n]);

else

buf.AppendFormat(\_T("%02x"), data->iph6->saddr[n]);

}

}

pthis->m\_listCtrl.SetItemText(nItem, 6, buf);

/\*获得目的IP\*/

buf.Empty();

if (0x0806 == data->ethh->type)

{

buf.Format(\_T("%d.%d.%d.%d"), data->arph->ar\_destip[0],

data->arph->ar\_destip[1], data->arph->ar\_destip[2], data->arph->ar\_destip[3]);

}

else if (0x0800 == data->ethh->type) {

struct in\_addr in;

in.S\_un.S\_addr = data->iph->daddr;

buf = CString(inet\_ntoa(in));

}

else if (0x86dd == data->ethh->type) {

int n;

for (n = 0; n < 8; n++)

{

if (n <= 6)

buf.AppendFormat(\_T("%02x:"), data->iph6->daddr[n]);

else

buf.AppendFormat(\_T("%02x"), data->iph6->daddr[n]);

}

}

pthis->m\_listCtrl.SetItemText(nItem, 7, buf);

/\*对包计数\*/

pthis->npkt++;

/\*对流量计数\*/

pthis->BYTE += (sizeof(header->len));

}

return 1;

}

保存函数：

//保存函数

int CSnifferDlg::Sniffer\_saveFile()

{

//用于

CFileFind find;

//如果没找到文件

if (find.FindFile(CString(filepath))==NULL)

{

MessageBox(\_T("保存文件出错"));

return -1;

}

//打开文件对话框

//保存格式设置为.pcap。以便可以使用wireshark打开

CFileDialog FileDlg(FALSE, \_T(".pcap"), NULL, OFN\_HIDEREADONLY | OFN\_OVERWRITEPROMPT);

//设置初始位置为C://

FileDlg.m\_ofn.lpstrInitialDir = \_T("c:\\");

if (FileDlg.DoModal() == IDOK)

{

CopyFile(CString(filepath), FileDlg.GetPathName(), TRUE);

}

return 1;

}

发送函数：

//发送函数

int CSnifferDlg::Sniffer\_sendPacket()

{

int index;

index = m\_ComboBoxPk.GetCurSel();//获得选取的序号

unsigned char sendbuf[42]; //arp包结构大小，42个字节

unsigned char sendbuf\_tcp[1460];

char mac[6] = { 0x00,0x11,0x22,0x33,0x44,0x55 };

char ip[4] = { 0x01,0x02,0x03,0x04 };

ethhdr eh;//定义各种需要的协议头

arphdr ah;

tcphdr th;

udphdr uh;

iphdr ih;

Psdhdr psh;

if (index == 1)

{

//赋值MAC地址

memset(eh.dest, 0xff, 6); //以太网首部目的MAC地址，全为广播地址

memcpy(eh.src, mac, 6); //以太网首部源MAC地址

memcpy(ah.ar\_srcmac, mac, 6); //ARP字段源MAC地址

memset(ah.ar\_destmac, 0xff, 6); //ARP字段目的MAC地址

memcpy(ah.ar\_srcip, ip, 4); //ARP字段源IP地址

memset(ah.ar\_destip, 0x05, 4); //ARP字段目的IP地址

eh.type = htons(ETH\_ARP); //htons：将主机的无符号短整形数转换成网络字节顺序

ah.ar\_hrd = htons(ARP\_HARDWARE);

ah.ar\_pro = htons(ETH\_IP);

ah.ar\_hln = 6;

ah.ar\_pln = 4;

ah.ar\_op = htons(ARP\_REQUEST);

//构造一个ARP请求

memset(sendbuf, 0, sizeof(sendbuf)); //ARP清零

memcpy(sendbuf, &eh, sizeof(eh));

memcpy(sendbuf + sizeof(eh), &ah, sizeof(ah));

//如果发送成功

if (pcap\_sendpacket(adhandle, sendbuf, 42) == 0)

{

printf("\nPacketSend succeed\n");

return 1;

}

else

{

printf("PacketSendPacket in getmine Error: %d\n", GetLastError());

return -1;

}

}

else

{

memset(eh.dest, 0xff, 6); //以太网首部目的MAC地址，全为广播地址

memcpy(eh.src, mac, 6); //以太网首部源MAC地址

eh.type = htons(ETH\_IP); //htons：将主机的无符号短整形数转换成网络字节顺序

//ip头部部分

ih.ihl = (4 << 4 | sizeof(ih) / sizeof(unsigned int));

ih.tos = 0;

ih.tlen = htons((unsigned short)(sizeof(iphdr) + sizeof(tcphdr)));

ih.id = 1;

ih.frag\_off = 0x40;

ih.ttl = 20;

ih.proto = PROTO\_TCP;

ih.check = 0;

ih.saddr = inet\_addr(ip);

ih.daddr = inet\_addr("5.5.5.5");

//tcp伪首部

psh.saddr = ih.saddr;

psh.daddr = ih.daddr;

psh.mbz = 0;

psh.ptcl = ih.proto;

psh.plen = htons(sizeof(tcphdr));

//tcp

th.sport = htons(20);//源端口

th.dport = htons(20);//目的端口

th.seq = htonl(1);//序列号

th.ack\_seq = 1;

th.syn = (u\_char)1;//标志

th.window = htons(400);//窗口大小

th.check = 0;//校验和暂时设为0

th.urg = 0;//偏移

//填充发送缓冲区

memset(sendbuf\_tcp, 0, sizeof(sendbuf\_tcp));

memcpy(sendbuf\_tcp, (void\*)&eh, sizeof(eh));

memcpy(sendbuf\_tcp + sizeof(eh), (void\*)&ih, sizeof(ih));

memcpy(sendbuf\_tcp + sizeof(eh) + sizeof(ih), (void\*)&th, sizeof(tcphdr));

//发送

if (pcap\_sendpacket(adhandle, sendbuf\_tcp, 1460) == 0)

{

return 1;

}

else

{

return -1;

}

}

}

协议的结构体定义：

#ifndef PROTOCOL\_H

#define PROTOCOL\_H

#define PROTO\_ICMP 1

#define PROTO\_TCP 6

#define PROTO\_UDP 17

#define LITTLE\_ENDIAN 1234

#define BIG\_ENDIAN 4321

//Mac帧头 占14个字节

typedef struct ethhdr

{

u\_char dest[6]; //6个字节 目标地址

u\_char src[6]; //6个字节 源地址

u\_short type; //2个字节 类型

};

//ARP头

typedef struct arphdr

{

u\_short ar\_hrd; //硬件类型

u\_short ar\_pro; //协议类型

u\_char ar\_hln; //硬件地址长度

u\_char ar\_pln; //协议地址长度

u\_short ar\_op; //操作码，1为请求 2为回复

u\_char ar\_srcmac[6]; //发送方MAC

u\_char ar\_srcip[4]; //发送方IP

u\_char ar\_destmac[6]; //接收方MAC

u\_char ar\_destip[4]; //接收方IP

};

//定义IP头

typedef struct iphdr

{

#if defined(LITTLE\_ENDIAN)

u\_char ihl : 4;

u\_char version : 4;

#elif defined(BIG\_ENDIAN)

u\_char version : 4;

u\_char ihl : 4;

#endif

u\_char tos; //TOS 服务类型

u\_short tlen; //包总长 u\_short占两个字节

u\_short id; //标识

u\_short frag\_off; //片位移

u\_char ttl; //生存时间

u\_char proto; //协议

u\_short check; //校验和

u\_int saddr; //源地址

u\_int daddr; //目的地址

u\_int op\_pad; //选项等

};

//定义TCP头

typedef struct tcphdr

{

u\_short sport; //源端口地址 16位

u\_short dport; //目的端口地址 16位

u\_int seq; //序列号 32位

u\_int ack\_seq; //确认序列号

#if defined(LITTLE\_ENDIAN)

u\_short res1 : 4,

doff : 4,

fin : 1,

syn : 1,

rst : 1,

psh : 1,

ack : 1,

urg : 1,

ece : 1,

cwr : 1;

#elif defined(BIG\_ENDIAN)

u\_short doff : 4,

res1 : 4,

cwr : 1,

ece : 1,

urg : 1,

ack : 1,

psh : 1,

rst : 1,

syn : 1,

fin : 1;

#endif

u\_short window; //窗口大小 16位

u\_short check; //校验和 16位

u\_short urg\_ptr; //紧急指针 16位

u\_int opt; //选项

};

//定义UDP头

typedef struct udphdr

{

u\_short sport; //源端口 16位

u\_short dport; //目的端口 16位

u\_short len; //数据报长度 16位

u\_short check; //校验和 16位

};

//定义ICMP

typedef struct icmphdr

{

u\_char type; //8位 类型

u\_char code; //8位 代码

u\_char seq; //序列号 8位

u\_char chksum; //8位校验和

};

//定义IPv6

typedef struct iphdr6

{

//#if defined(BIG\_ENDIAN)

u\_int version : 4, //版本

flowtype : 8, //流类型

flowid : 20; //流标签

u\_short plen; //有效载荷长度

u\_char nh; //下一个头部

u\_char hlim; //跳限制

u\_short saddr[8]; //源地址

u\_short daddr[8]; //目的地址

};

//定义ICMPv6

typedef struct icmphdr6

{

u\_char type; //8位 类型

u\_char code; //8位 代码

u\_char seq; //序列号 8位

u\_char chksum; //8位校验和

u\_char op\_type; //选项：类型

u\_char op\_len; //选项：长度

u\_char op\_ethaddr[6]; //选项：链路层地址

};

//对各种包进行计数

typedef struct pktcount

{

int n\_ip;

int n\_ip6;

int n\_arp;

int n\_tcp;

int n\_udp;

int n\_icmp;

int n\_icmp6;

int n\_http;

int n\_other;

int n\_sum;

};

//

//要保存的数据结构

typedef struct datapkt

{

char pktType[8]; //包类型

int time[6]; //时间

int len; //长度

struct ethhdr\* ethh; //链路层包头

struct arphdr\* arph; //ARP包头

struct iphdr\* iph; //IP包头

struct iphdr6\* iph6; //IPV6

struct icmphdr\* icmph; //ICMP包头

struct icmphdr6\* icmph6; //ICMPv6包头

struct udphdr\* udph; //UDP包头

struct tcphdr\* tcph; //TCP包头

void\* apph; //应用层包头

};

//定义ARP包,总长度42字节

struct ArpPacket

{

ethhdr ed;

arphdr ah;

};

//定义伪首部部分

//定义伪头部

struct Psdhdr {

unsigned long saddr;

unsigned long daddr;

char mbz;

char ptcl;

unsigned short plen;

};

#endif

3.程序实现功能

（1）选取网卡，接收不同种类的数据包

（2）对抓取的数据包进行逐一褪去包头的分析，然后以树形结构展示

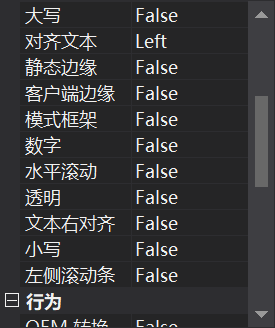
（3）对开始捕获后抓取的不同类型数据包及通过网卡的流量进行分析

（4）发送两种不同种类的数据包

（5）将读取的数据包序列保存，并可以采用wireshark打开。

4.程序调试过程

1. 实验中出现的错误以及解决方式
2. this指针的用法未知，不知道怎么编写事件处理程序，解决方式：查阅相关资料，了解到this指针用于指向这个类本身的成员变量，由于界面也是一个类，所以可以通过this指向其中的某些内容来获取某一个控件的状态。
3. 不清楚pcap\_compile以及pcap\_setfilter之间的区别，解决方式：经过查阅资料了解到，pcap\_compile()用于把一个字符串应用于一个过滤程序。
4. 多线程的创建不了解，因为我没有系统学过操作系统相关的知识，所以查找了一些多线程的参考程序，了解到可以使用CreateThread去实现，当然也需要一个线程指针，这个指针的创建可以采用线程函数创建，返回值为DWORD。
5. 如何保存文件，首先需要有一个文件的位置，然后用一个内置的CFilefind类，它自带的成员函数有一个可以用于找到对应的文件位置
6. Error: no data exchange control with ID 0x%04X.\n ", nIDC，这个错误困扰了我非常长的时间，因为这个错误是可以通过编译的，只是会出现asserttion，解决方式是：首先调试程序，然后查看堆栈，发现错误出现的地方，然后记录nIDC的值，然后再去头文件查找定义，发现是nIDC为1000的变量出错了，于是我再去找这个变量，发现是我以前在图形化拖拽界面中误删了一个空间，但是这个空间相关的内容还在所致。
7. 程序的独立性不强，解决方式：对需要的各个部件采用分别设计的原则，设计成不同的类，并且设计多个函数采用extern的方式互相使用。
8. 程序的界面不美观，很多控件无法使用。解决方式：尝试不同控件的UI展示，然后在VS中进行改动，这样交互界面就更加良好。

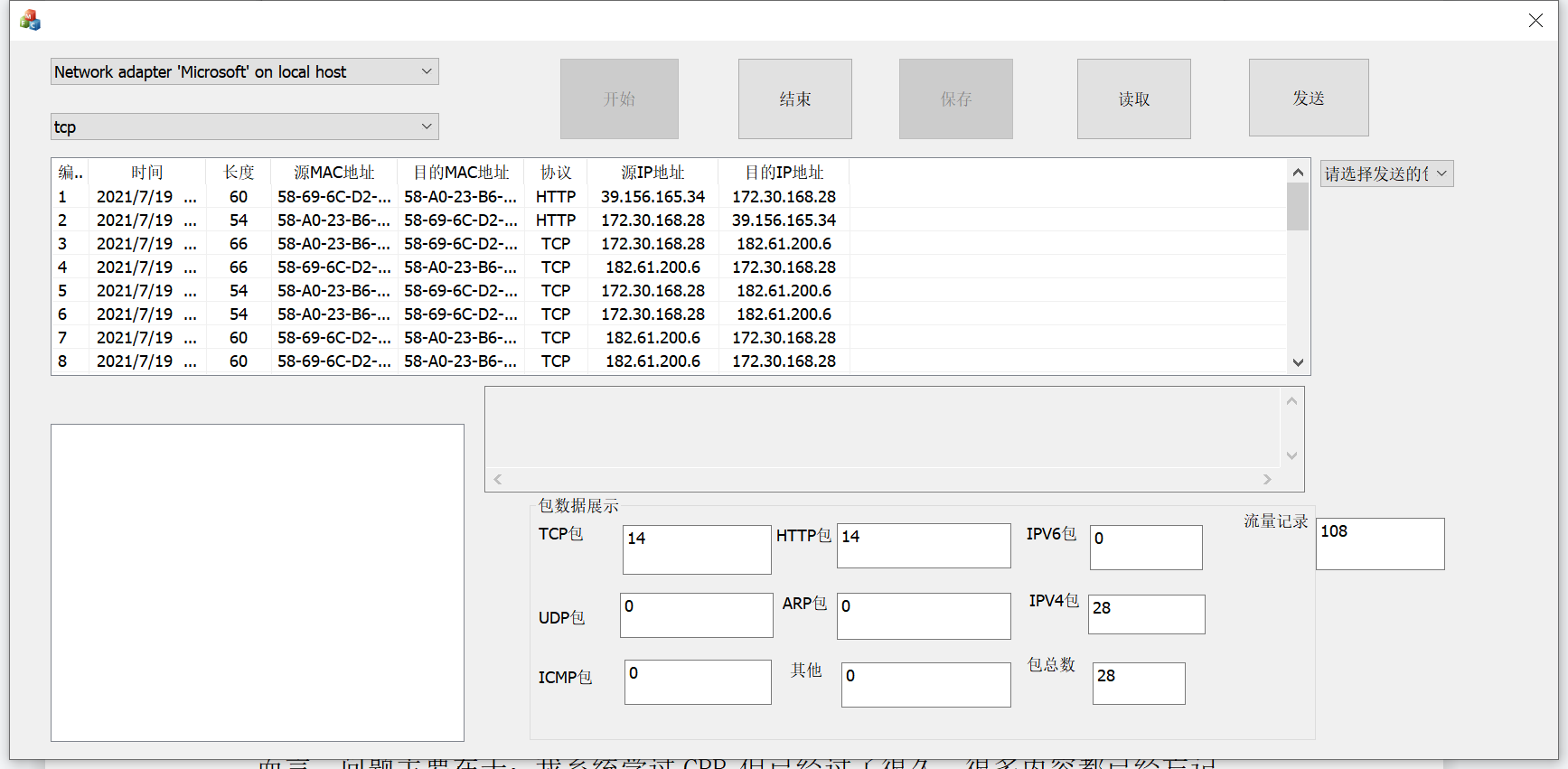


1. 如何实现改写发送包的功能，这里我只是做了一个简单的尝试，我通过鼠标操作拿到一个一个数据包的源IP和目的IP，然后将他们交换，再加入其他数据，通过**libnet**这一非常热门的数据包构建神器来整合成一个数据包并发送，这样就实现了数据包的发送和改写，当然，为了使更具有实用价值，应该可以加几个文本框来让用户自己输入一些数据来发送，这才是真正意义上实现了改写，不过原理是相同的，不过最终发现libnet在win平台还是不太好用，所以最后搁置了
2. 发送数据包的功能中不知道如何将结构体定义转化为可以发送的字节流，解决方式为利用一个叫htons的函数，可以把ip和mac地址转化为可以被识别的字节流，通过这一思想我实现了arp包和tcp包的发送，当然，这个ip和mac地址是虚构的，但由于流经网卡，也是可以抓取到的。

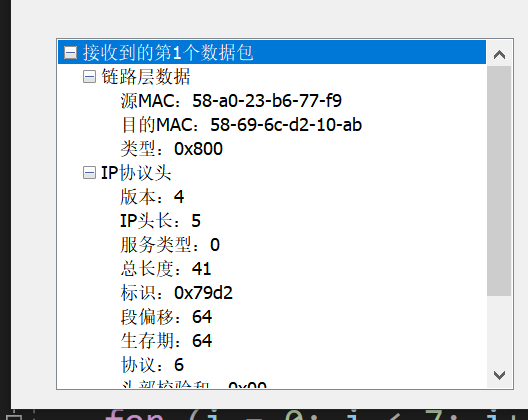
七、程序使用说明、运行结果

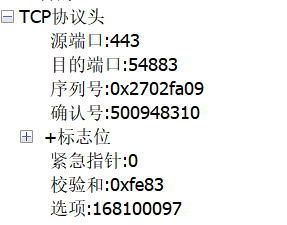
1. 界面如下，首先选取网卡接口和过滤规则，点击开始按钮：



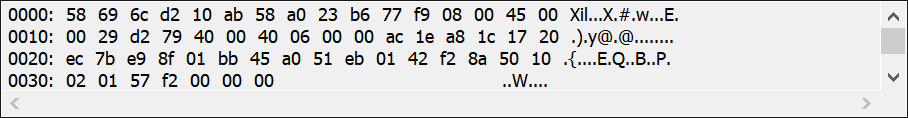
2.选取网卡和报文之后，我选择TCP过滤器，可以看到抓取到了使用TCP协议的报文

3.在左下角树形结构可以展开看报文的各个部分的结构

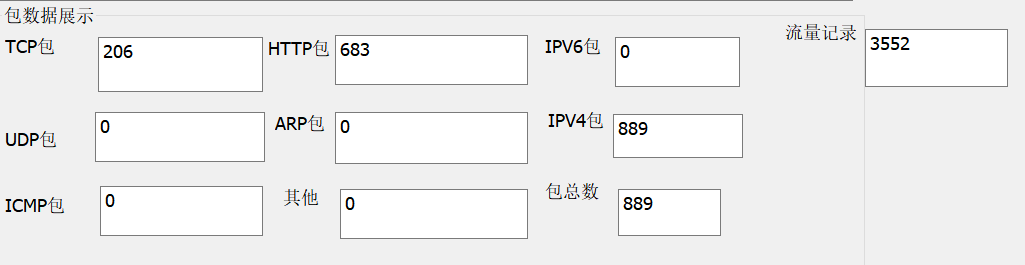




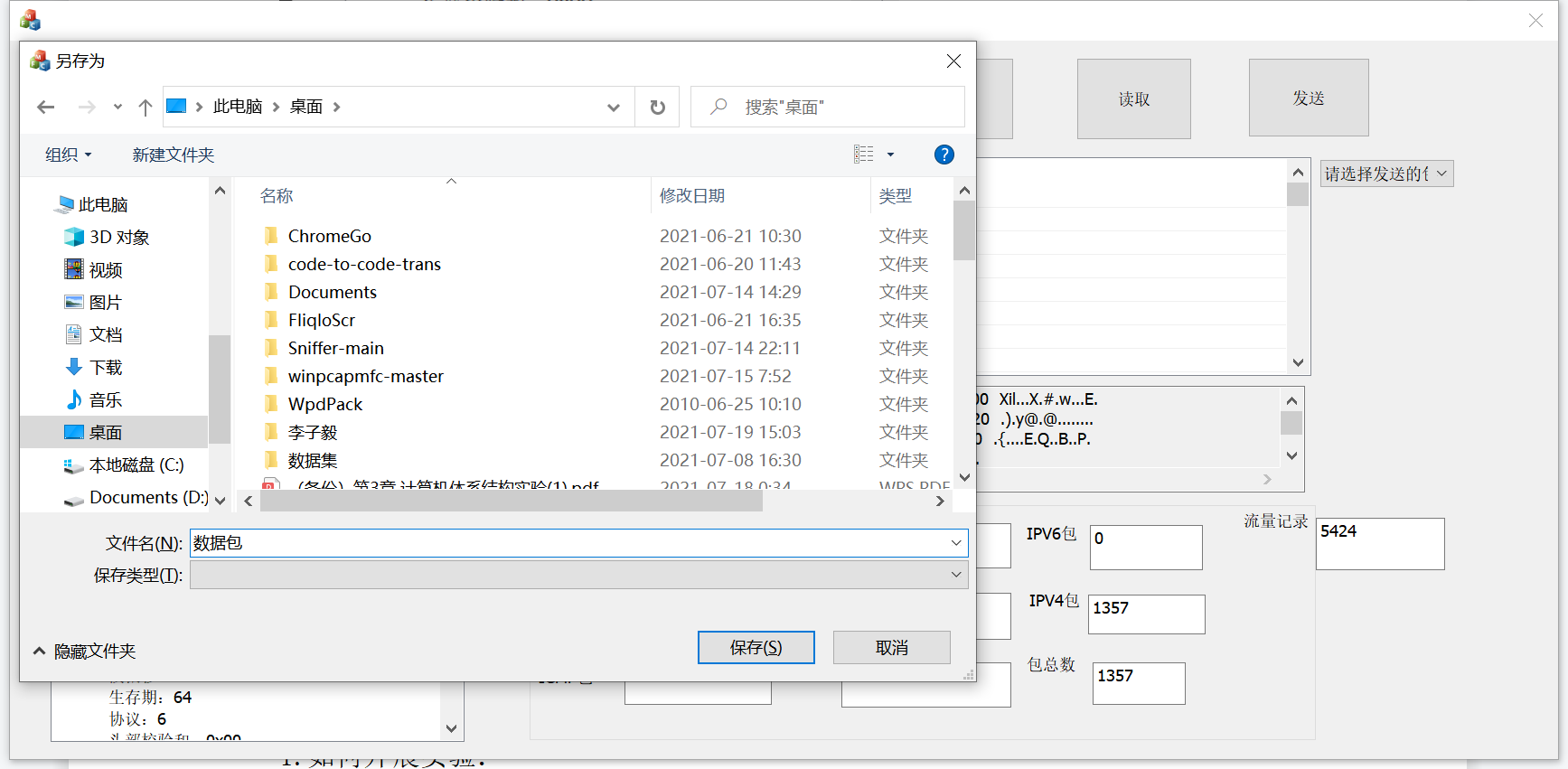
1. 右边的数据部分可以查看数据部分的内容：



1. 可以查看对于包数据的统计结果：



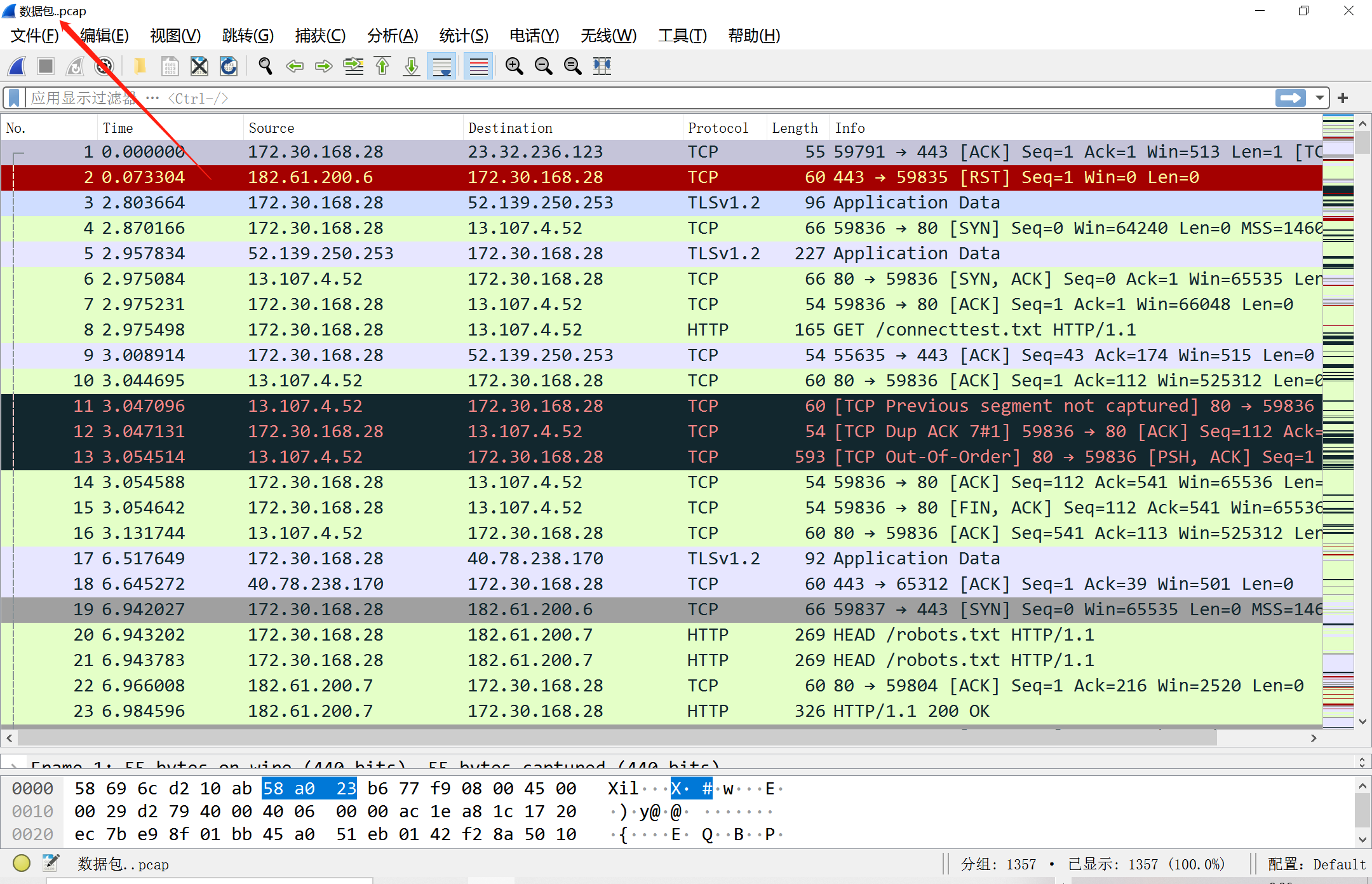
1. 点击结束之后，当前界面的数据已经暂时存入内存，现在测试保存功能，选取保存路径后点击保存按钮。



这里我选择存到桌面，名字为数据包，点击确定，可以看到桌面已经有了数据包

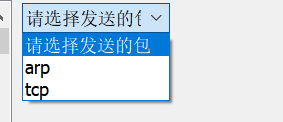


现在打开wireshark来打开这个刚刚保存的数据包：

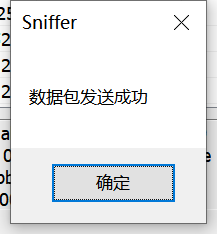


可以看到是成功的。

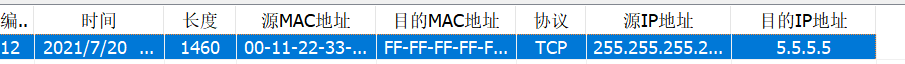
1. 现在测试发送功能，首先选取要发送包的类型，目前提供tcp和arp两种类型，



这里选取tcp，点击发送，可以看到提示发送成功



然后抓取tcp数据包，由于我设置的是一组特殊的MAC地址00-11-22-33和FF-FF-FF-FF，所以抓取后易于发现，然后我再打开抓包功能进行抓包。



可以看到抓取成功，发现功能是成功的。

八、实验总结

1.如何开展实验：

(1)

首先拿到这个题目时我询问了老师是否可以使用其他的编程语言来写，老师说最好使用C语言来写，但C语言确实在制作图形化界面上有一定的难度，所以我想到了使用C++语言和C语言来结合完成这次任务，其中C++语言主要是想用QT这一库来构建GUI界面。然后网络编程部分就用推荐的WINPCAP来实现，对于我而言，问题主要在于：我系统学过CPP,但已经过了很久，很多内容都已经忘记，尤其面向对象的思想要拾起来并不容易，还有就是QT是一款非常成熟的库，使用其他可能需要系统的学习，学习的成本也很大，所以这项工作并不容易，不过我打算先从自己可以入手的方向入手，先着手工作再逐渐温习知识。

最终选择了通过MFC来构建GUI界面，它与QT类似，配置环境更简单，采用拖拽的形式构建图形化界面，然后在图形化界面上添加事件处理程序，这样就与我用winpcap制作的后端相连了。

1. 实验收获

通过本次实验，我深刻理解了一个数据包的接受和发送过程，实现了自顶向下的网络体系结构知识，在构建协议头的过程中又重温了各个协议的数据报的首部组成，并且自己用代码搭建了这些不同报文的结构体，另外还在这段时间学习了c++和一些库，对于线程和回调函数等知识有了一些了解并及时记录了下来，提升了自己的编程水平，收益良多，最后感谢白老师这段时间的帮助和指导。

九、参考文献

《winpcap中文技术文档》

《RFC协议》

《计算机网络：自顶向下》

《计算机网络》