嗅探器的设计与实现

一、实验目的

1. 设计与实现带GUI的嗅探器
2. 实现NPS功能
3. 完善NPA功能

二、实验环境

IDE：Qt Creator 4.13.1

第三方库：pcap

Github：https://github.com/wjialei/HSniffer.git

三、实验内容

1. 搭建GUI界面

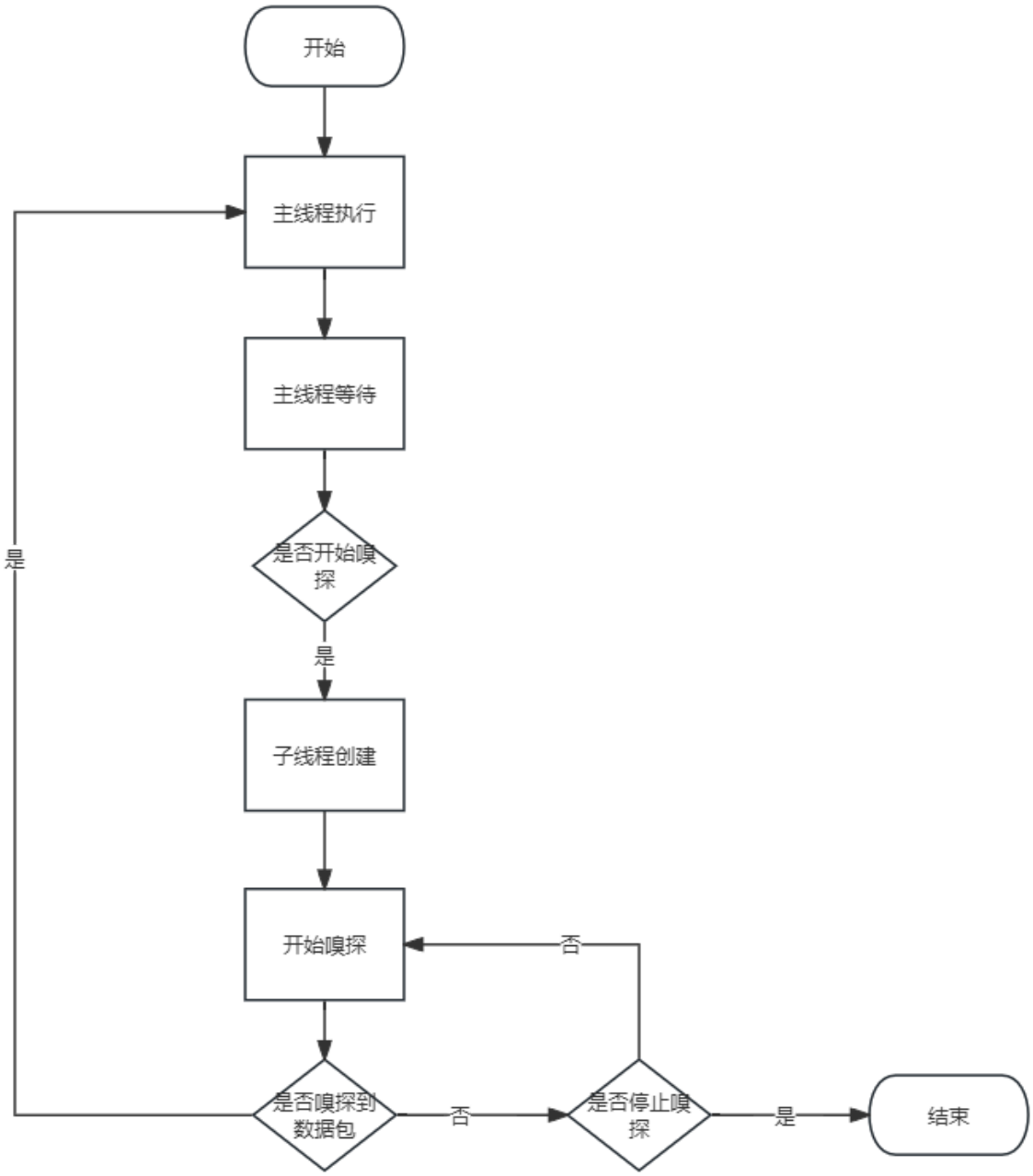
本实验所设计GUI界面如下，在本界面中可以选择机器上的不同网卡进行嗅探，嗅探所获得数据会在下面三个展示区展示。

2. 设计抓包流程

本实验中，采用了QT的多线程技术、信号和槽技术来完成主界面的渲染和子线程的嗅探。

1. 程序开始运行时，会创建主线程，在主线程中获取到网卡信息以及主界面的初次渲染；
2. 用户点击“开始捕获”按钮，主线程会创建一个子线程，同时发送相关参数信息给子线程；
3. 子线程启动后，会根据从主线程接收到参数进行嗅探；
4. 子线程嗅探到数据包，将数据包解析并格式化后发送回主线程；

5. 主线程接收到数据包后，对界面进行再一次渲染；
6. 用户点击“停止捕获”按钮，子线程停止嗅探。



3. 参数传递方式

a. 信号与槽

本实验中，主线程给子线程传递选定的嗅探网卡、过滤规则，子线程给主线程传递数据包简要信息等使用了QT中的信号与槽技术，举例如下。

```
// 绑定主线程的网卡编号信号和抓包线程的网卡编号接收槽
connect(this, &MainWindow::sendAdapterIndex, ct,
&CapThread::recAdapterIndex);

qRegisterMetaType<PacketTableItem>("PacketTableItem");

// 绑定子线程的发送信息信号和主线程的信息接收槽
connect(ct, &CapThread::sendMsgtoMain, this, &MainWindow::recMsgfromCap);

```

```
ui->adaptersComboBox->addItem(tr("请选择一个网卡接口"));
int ret = getAllAdapters();
if(ret == false) {
    QMessageBox::warning(this, tr("Sniffer"), tr("无法获取网卡接口"),
QMessageBox::Ok);
} else {
    for(pcap_if_t* ptr = allAdapters; ptr!=NULL; ptr=ptr->next) {
        ui->adaptersComboBox->addItem(QString("%1").arg(ptr->description));
    }
}

// 点击按钮发送网卡编号和过滤规则给子线程
void MainWindow::on_startCapButton_clicked()
{
    int adapterIndex = ui->adaptersComboBox->currentIndex();
    string rule = ui->ruleLineEdit->text().toStdString();
    emit sendAdapterIndex(adapterIndex, rule);
    capFlag = true;

    ct->start();
    ui->finishCapButton->setEnabled(true);
    ui->startCapButton->setEnabled(false);

}
```

b. 全局变量

本实验中，网络数据包具体内容的传递，以及是否停止嗅探的信号是使用全局变量传递的。将类中的变量设置为“extern”类型，使得使用该类的所有其他类都可以进行修改。举例如下。

```
extern bool capFlag; // 该变量位于CapThread中

// 点击主界面按钮可以修改该变量的值
void MainWindow::on_finishCapButton_clicked()
{
    capFlag = false;
    ui->finishCapButton->setEnabled(false);
    ui->startCapButton->setEnabled(true);
}
```

四、关键代码分析

1. 网络数据包捕获

a. 获取网卡信息

通过调用pcap库中的“pcap_findalldevs_ex”函数，获取到本机器上的网卡信息，并存放在“allAdapters”中。

```
pcap_if_t* allAdapters = nullptr;

bool getAllAdapters() {
```

```
    if(allAdapters) {
        freeAdapters();
        cntAdapters = 0;
    }
    allAdapters = nullptr;
    memset(errbuf, 0x00, sizeof(errbuf));
    int ret = pcap_findalldevs_ex(PCAP_SRC_IF_STRING, nullptr, &allAdapters,
errbuf);
    if (ret == -1)
    {
        return false;
    }
    pcap_if_t *ptr = allAdapters;
    while(ptr) {
        ptr = ptr->next;
        cntAdapters++;
    }
    return true;
}
```

b. 嗅探器配置

通过调用pcap中的“pcap_lookupnet”、“ pcap_open”完成嗅探器的配置，调用“pcap_compile”、“pcap_setfilter”完成过滤规则的配置。

```
pcap_t* sniff = nullptr;

bool Sniff(int num, string filter_pattern) {
    if(num < 0) return false;
    pcap_if_t* adapter = allAdapters;
    for(int i=0; i<num-1&&adapter; i++) {
        adapter = adapter->next;
    }
    if(adapter == nullptr) return false;
    const char* dev = adapter->name;
    if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
             cout << "get netmask flase" << endl;
             net = 0;
             mask = 0;
    }
    sniff = pcap_open(adapter->name, MAXDATAFRAME,
PCAP_OPENFLAG_PROMISCUOUS, 1000, nullptr, errbuf);
    if(sniff == nullptr) {
        return false;
    }
    filter_exp = filter_pattern.c_str();
    if(pcap_compile(sniff, &fp, filter_exp, 0, net) == -1) {
        cout << "filter compile false" << endl;
        return false;
    }
    if (pcap_setfilter(sniff, &fp) == -1) {
        cout << "filter set false" << endl;
        return false;
    }
```

```
    if(pcap_datalink(sniff) != DLT_EN10MB) return false;
    return sniff!=nullptr;
}
```

c. 数据包捕获

通过调用pcap中的“pcap_next_ex”完成数据包的捕获。

```
bool getDataPacket() {
    packData = nullptr;
    int r= pcap_next_ex(sniff, &packHeader, &packData);
    return r;
}
```

2. 链路层数据分析

a. MAC信息

根据链路层数据头定义，将原始数据包中的MAC相关数据转为16进制字符串，保存起来。

```
struct ethernet_info {
    string mac_dest;
    string mac_src;
    string mac_type;
    string mac_content;
};

bool parseEthernetProtocol(data_packet* dp, const u_char* d) {
    struct ethernet_header {
        u_char ether_dhost[6];
        u_char ether_shost[6];
        u_short ether_type;
    };
    ethernet_header* etherH = (ethernet_header*)d;

    dp->ethernet_header = new ethernet_info();
#define NTOHS(A) ((((A)&0xFF00)>>8) | (((A)&0x00FF)<<8));
    etherH->ether_type = NTOHS(etherH->ether_type);
    dp->ethernet_header->mac_type = "0x" + dataToString(etherH->ether_type,
16);

    // dest mac address
    memset(buf, 0x00, sizeof (buf));
    sprintf(buf, " %02X:%02X:%02X:%02X:%02X:%02X ",
            etherH->ether_dhost[0],
            etherH->ether_dhost[1],
            etherH->ether_dhost[2],
            etherH->ether_dhost[3],
            etherH->ether_dhost[4],
            etherH->ether_dhost[5]);
    dp->ethernet_header->mac_dest = buf;

    // source mac address
```

```
    memset(buf, 0x00, sizeof (buf));
    sprintf(buf, " %02X:%02X:%02X:%02X:%02X:%02X ",
            etherH->ether_shost[0],
            etherH->ether_shost[1],
            etherH->ether_shost[2],
            etherH->ether_shost[3],
            etherH->ether_shost[4],
            etherH->ether_shost[5]);
    dp->ethernet_header->mac_src = buf;

    dp->ethernet_header->mac_content += "0x";
    for(int i=0; i<6; i++) {
        string s = dataToString(etherH->ether_dhost[i], 16);
        dp->ethernet_header->mac_content += string("0", 2-s.size()) + s + "
";
    }
    for(int i=0; i<6; i++) {
        string s = dataToString(etherH->ether_shost[i], 16);
        dp->ethernet_header->mac_content += string("0", 2-s.size()) + s + "
";
    }
    string s = dataToString(etherH->ether_type, 16);
    dp->ethernet_header->mac_content += string("0", 4-s.size()) + s;

    return true;
}
```

b. ARP协议

根据ARP协议头定义，将原始数据包中的ARP协议数据转为16进制字符串，保存起来。

```
struct arp_info {
    string arp_htype;
    string arp_ptype;
    string arp_hsize;
    string arp_psize;
    string arp_opcode;
    string arp_src;
    string arp_sip;
    string arp_dest;
    string arp_dip;
};

bool parseArpProtocol(data_packet* dp, const u_char* d) {
    struct arp_header {
        u_short arp_htype;
        u_short arp_ptype;
        u_char arp_hsize;
        u_char arp_psize;
        u_short arp_opcode;
        u_char arp_src[6];
        long arp_sip;
        u_char arp_dest[6];
        long arp_dip;
    };
```

```
    arp_header* arpH = (arp_header*)(d + 14);

    dp->arp_header = new arp_info();
    dp->arp_header->arp_htype = "0x" + dataToString(MY_NTOHS(arpH->arp_htype), 16);
    dp->arp_header->arp_ptype = "0x" + dataToString(MY_NTOHS(arpH->arp_ptype), 16);
    dp->arp_header->arp_hsize = "0x" + dataToString(arpH->arp_hsize, 16);
    dp->arp_header->arp_psize = "0x" + dataToString(arpH->arp_psize, 16);
    dp->arp_header->arp_opcode = "0x" + dataToString(MY_NTOHS(arpH->arp_opcode), 16);
    dp->arp_header->arp_sip = iptos(arpH->arp_sip);
    dp->arp_header->arp_dip = iptos(arpH->arp_dip);

    // dest mac address
    memset(buf, 0x00, sizeof (buf));
    sprintf(buf, " %02X:%02X:%02X:%02X:%02X:%02X ",
            arpH->arp_dest[0],
            arpH->arp_dest[1],
            arpH->arp_dest[2],
            arpH->arp_dest[3],
            arpH->arp_dest[4],
            arpH->arp_dest[5]);
    dp->arp_header->arp_dest = buf;

    // source mac address
    memset(buf, 0x00, sizeof (buf));
    sprintf(buf, " %02X:%02X:%02X:%02X:%02X:%02X ",
            arpH->arp_src[0],
            arpH->arp_src[1],
            arpH->arp_src[2],
            arpH->arp_src[3],
            arpH->arp_src[4],
            arpH->arp_src[5]);
    dp->arp_header->arp_src = buf;

    return true;
}
```

3. 网络层数据分析

a. IP协议

根据IP协议头定义，将原始数据包中的IP协议数据转为16进制字符串，保存起来。

```
struct ip_info {
    string ip_version;
    string ip_headLen;
    string ip_diffserv;
    string ip_totalLen;
    string ip_identification;
    string ip_flag_offset;
    string ip_ttl;
    string ip_protocol;
    string ip_checkSum;
```

```
    string ip_src;
    string ip_dest;
    string ip_content[4];
};

bool parseNetworkProtocol(data_packet* dp, const u_char* d) {
    struct ip_header {
        u_char ip_version_headerLen;
        u_char ip_service;
        u_short ip_totalLen;
        u_short ip_identification;
        u_short ip_flag_offset;
        u_char ip_ttl;
        u_char ip_protocol;
        u_short ip_checkSum;
        long ip_src;
        long ip_dest;
    };
    ip_header* ipH = (ip_header*)(d + 14);

    dp->ip_header = new ip_info();
    if((ipH->ip_version_headerLen&(0x40)) == 0x40) {
        dp->ip_header->ip_version = "ipv4";
    } else if((ipH->ip_version_headerLen&(0x60)) == 0x60) {
        dp->ip_header->ip_version = "ipv6";
    } else {
        return false;
    }
    char len = ipH->ip_version_headerLen & 0x0f;
    if(len < 0x05) return false;
    dp->ip_header->ip_headLen = "0x" + dataToString(len, 16);
    dp->ip_header->ip_diffserv = "0x" + dataToString(ipH->ip_service, 16);
    dp->ip_header->ip_totalLen = "0x" + dataToString(MY_NTOHS(ipH->ip_totalLen), 16);
    dp->ip_header->ip_identification = "0x" + dataToString(MY_NTOHS(ipH->ip_identification), 16);
    dp->ip_header->ip_flag_offset = "0x" + dataToString(MY_NTOHS(ipH->ip_flag_offset), 16);
    dp->ip_header->ip_ttl = "0x" + dataToString(ipH->ip_ttl, 16);
    dp->ip_header->ip_protocol = dataToString(ipH->ip_protocol, 10);
    dp->ip_header->ip_checkSum = "0x" + dataToString(ipH->ip_checkSum, 16);
    dp->ip_header->ip_src = iptos(ipH->ip_src);
    dp->ip_header->ip_dest = iptos(ipH->ip_dest);
    dp->ip_header->ip_content[0] = "0x" + FUN(dataToString(ipH->ip_version_headerLen,16),2) + " " + FUN(dataToString(ipH->ip_service,16),2) + " " + FUN(dataToString(ipH->ip_totalLen,16),4);
    dp->ip_header->ip_content[1] = "0x" + FUN(dataToString(ipH->ip_identification,16),4) + " " + FUN(dataToString(ipH->ip_flag_offset,16),4);
    dp->ip_header->ip_content[2] = "0x" + FUN(dataToString(ipH->ip_ttl,16),2) + " " + FUN(dataToString(ipH->ip_protocol,16),2) + " " + FUN(dataToString(ipH->ip_checkSum,16),4);

    return true;
}
```

4. 传输层数据分析

a. TCP协议

根据TCP协议头定义，将原始数据包中的TCP协议数据转为16进制字符串，保存起来。

```
struct tcp_info {
    string tcp_sport;
    string tcp_dport;
    string tcp_seqNum;
    string tcp_ackNum;
    string tcp_offset_res_flag;
    string tcp_windowSize;
    string tcp_checkSum;
    string tcp_urgentPoint;
    string tcp_content[5];
};

bool parseTcpProtocol(data_packet* dp, const u_char* d) {
    struct tcp_header {
        u_short tcp_sport;
        u_short tcp_dport;
        u_int tcp_seqNum;
        u_int tcp_ackNum;
        u_short tcp_off_res_flag;
        u_short tcp_winSize;
        u_short tcp_checkSum;
        u_short tcp_urgentPoint;
    };
    tcp_header* tcpH = (tcp_header*)(d + 14 + 20);

    dp->tcp_header = new tcp_info();
    dp->tcp_header->tcp_sport = "0x" + dataToString(MY_NTOHS(tcpH->tcp_sport), 16)+ "(" + to_string(MY_NTOHS(tcpH->tcp_sport)) + ")";
    dp->tcp_header->tcp_dport = "0x" + dataToString(MY_NTOHS(tcpH->tcp_dport), 16) + "(" + to_string(MY_NTOHS(tcpH->tcp_dport)) + ")";
    dp->tcp_header->tcp_seqNum = "0x" + dataToString(MY_NTOHL(tcpH->tcp_seqNum), 16) + "(" + to_string(MY_NTOHL(tcpH->tcp_seqNum)) + ")";
    dp->tcp_header->tcp_ackNum = "0x" + dataToString(MY_NTOHS(tcpH->tcp_ackNum), 16) + "(" + to_string(MY_NTOHS(tcpH->tcp_ackNum)) + ")";
    dp->tcp_header->tcp_offset_res_flag = "0x" + dataToString(MY_NTOHS(tcpH->tcp_off_res_flag), 16);
    dp->tcp_header->tcp_windowSize = "0x" + dataToString(MY_NTOHS(tcpH->tcp_winSize), 16);
    dp->tcp_header->tcp_checkSum = "0x" + dataToString(MY_NTOHS(tcpH->tcp_checkSum), 16);
    dp->tcp_header->tcp_urgentPoint = "0x" + dataToString(MY_NTOHS(tcpH->tcp_urgentPoint), 16);
    dp->tcp_header->tcp_content[0] = "0x" + FUN(dataToString(MY_NTOHS(tcpH->tcp_sport), 16), 4) + " " + FUN(dataToString(MY_NTOHS(tcpH->tcp_dport), 16), 4);
    dp->tcp_header->tcp_content[1] = "0x" + FUN(dataToString(MY_NTOHL(tcpH->tcp_seqNum), 16), 8);
    dp->tcp_header->tcp_content[2] = "0x" + FUN(dataToString(MY_NTOHL(tcpH->tcp_ackNum), 16), 8);
```

```
    dp->tcp_header->tcp_content[3] = "0x" + FUN(dataToString(MY_NTOHS(tcpH->tcp_off_res_flag), 16), 4) + " " + FUN(dataToString(MY_NTOHS(tcpH->tcp_winSize), 16), 4);
    dp->tcp_header->tcp_content[4] = "0x" + FUN(dataToString(MY_NTOHS(tcpH->tcp_checkSum), 16), 4) + " " + FUN(dataToString(MY_NTOHS(tcpH->tcp_urgentPoint), 16), 4);

    return true;
}
```

b. UDP协议

根据UDP协议头定义，将原始数据包中的UDP协议数据转为16进制字符串，保存起来。

```
struct udp_info {
    string udp_sport;
    string udp_dport;
    string udp_len;
    string udp_checkSum;
    string udp_content[2];
};


bool parseUdpProtocol(data_packet* dp, const u_char* d) {
    struct udp_header {
        u_short udp_sport;
        u_short udp_dport;
        u_short udp_length;
        u_short udp_checkSum;
    };
    udp_header* udpH = (udp_header*)(d + 14 + 20);

    dp->udp_header = new udp_info();
    dp->udp_header->udp_sport = "0x" + dataToString(udpH->udp_sport,16) + "(" + to_string(udpH->udp_sport) + ")";
    dp->udp_header->udp_dport = "0x" +dataToString(udpH->udp_dport,16) + "(" + to_string(udpH->udp_dport) + ")";
    dp->udp_header->udp_len = "0x" +dataToString(udpH->udp_length,16);
    dp->udp_header->udp_checkSum = "0x" +dataToString(udpH->udp_checkSum,16);
    dp->udp_header->udp_content[0] = "0x" + FUN(dataToString(udpH->udp_sport, 16), 4) + " " + FUN(dataToString(udpH->udp_dport, 16), 4);
    dp->udp_header->udp_content[1] = "0x" + FUN(dataToString(udpH->udp_length, 16), 4) + " " + FUN(dataToString(udpH->udp_checkSum, 16), 4);

    return true;
}
```

c. ICMP协议

根据ICMP协议头定义，将原始数据包中的ICMP协议数据转为16进制字符串，保存起来。

```
struct icmp_info {
    string icmp_type;
    string icmp_code;
    string icmp_checkSum;
    string icmp_identification;
    string icmp_seq;
    string icmp_initTime;
    string icmp_recvTime;
    string icmp_sendTime;
};


bool parseIcmpProtocol(data_packet* dp, const u_char* d) {
    struct icmp_header {
        u_char icmp_type;
        u_char icmp_code;
        u_short icmp_checkSum;
        u_short icmp_identification;
        u_short icmp_seq;
        u_int icmp_initTime;
        u_short icmp_recvTime;
        u_short icmp_sendTime;
    };
    icmp_header* icmpH = (icmp_header*)(d + 14 +20);

    dp->icmp_header = new icmp_info();
    dp->icmp_header->icmp_type = "0x" + dataToString(icmpH->icmp_type, 16);
    dp->icmp_header->icmp_code = "0x" + dataToString(icmpH->icmp_code, 16);
    dp->icmp_header->icmp_checkSum = "0x" + dataToString(MY_NTOHS(icmpH->icmp_checkSum), 16);
    dp->icmp_header->icmp_identification = "0x" + dataToString(MY_NTOHS(icmpH->icmp_identification), 16);
    dp->icmp_header->icmp_seq = "0x" + dataToString(MY_NTOHS(icmpH->icmp_seq), 16);
    dp->icmp_header->icmp_initTime = "0x" + dataToString(MY_NTOHL(icmpH->icmp_initTime), 16);
    dp->icmp_header->icmp_recvTime = "0x" + dataToString(MY_NTOHS(icmpH->icmp_recvTime), 16);
    dp->icmp_header->icmp_sendTime = "0x" + dataToString(MY_NTOHS(icmpH->icmp_sendTime), 16);

    return true;
}
```

五、实验感想

1. 不足之处

a. 解析的协议较少

本实验所设计的嗅探器，目前只对ARP、IP、TCP、UDP、ICMP协议做了具体的解析，对于更多的应用层协议以及IPv6协议尚未做解析。

b.界面设计过于简陋

本实验设计的嗅探器的GUI界面，仅仅是对所捕获的数据包的简要信息、十六进制信息、解析信息做了简单的的展示，与如wireshark等工具的界面差距极大。

2. 改进方向

a. 添加更多的协议解析方法

1. 目前最高解析到了传输层，还可以添加应用层协议的解析；
2. 还可以添加IPv6协议的解析。

b. 优化GUI界面设计

1. 进一步美化数据简要信息显示表格；
2. 设计新的交互逻辑，使得十六进制数据可以和解析数据选中时匹配高亮。

c. 添加数据包保存功能

1. 添加将保存捕获到的数据包到文件中的功能。