

**网络编程技术**

**课程项目报告**

**班 级 9191062301**

**学生姓名 孙傲歆**

**学 号 919106840333**

**题 目 发包器与抓包器的实现**

**指导教师 苏铓**

**2021年10月**

**摘 要**

本课程项目包括两个部分：**一是发包器，即协议编辑器**；**二是抓包器，即网络嗅探器。**

发包器基于python+scapy进行开发设计，基于Tkinter的Python GUI界面设计。实现了HTTP、DNS、TCP、UDP、ARP、IP、ICMP、MAC协议的编辑与发送，并且支持生成协议字段的默认值，支持用户输入协议字段值，发包前对协议字段的合理性进行检查，发包前自动计算并显示校验和，支持单次发包、多次发包，支持连续发包，连续发包时可随时停止和计算并显示数据包发送速度的功能。

抓包器同样基于python+scapy进行开发设计。并基于TKinter进行GUI界面设计。项目支持连续抓包，并可以随时停止、继续。还可以将已经抓取到的数据进行存储，并读取本地的相关存储文件。项目支持对每一个数据包的读取分析，可以获取数据包应用层、传输层、网络层、链路层的相关协议信息，并可读取十六进制数据。同时还可以编写BPF语句进行过滤。

本报告分为**项目开发基础技术**、**系统分析**、**系统开发**、**系统测试**、**特殊问题及解决方案**、**总结**六个部分。是一个从浅入深、层层递进的关系。其中项目开发基础技术主要介绍项目基本内容及网络协议相关内容；系统分析主要介绍功能实现方案以及系统的详细设计；系统开发则介绍开发环境与重要代码；特殊问题及解决方案和总结两个部分则是对系统开发过程的一个反思与体会。

项目的两个部分都已完成并可成功运行。项目不仅可以在传统的windows操作系统下运行，还能在Ubuntu、centos等linux操作系统上运行。

**关键字：**协议编辑器； 网络嗅探器； python； scapy

# **目 录**

[**摘 要** 2](#_Toc85704767)

[**目 录** 3](#_Toc85704768)

[第一章 项目开发基础技术 4](#_Toc85704769)

[1.1 项目内容介绍 4](#_Toc85704770)

[1.2 网络协议的基本格式 5](#_Toc85704771)

[**1.2.1 应用层协议** 5](#_Toc85704772)

[**1.2.2 传输层协议** 7](#_Toc85704773)

[**1.2.3 网络层协议** 7](#_Toc85704774)

[**1.2.4 链路层协议** 9](#_Toc85704775)

[第二章 系统分析 9](#_Toc85704776)

[2.1 系统功能介绍及实现方案 9](#_Toc85704777)

[**2.1.1 发包器功能介绍及实现方案** 9](#_Toc85704778)

[**2.1.2 抓包器功能介绍及实现方案** 10](#_Toc85704779)

[2.2 系统详细设计 11](#_Toc85704780)

[**2.2.1 发包起详细设计** 11](#_Toc85704781)

[**2.2.2 抓包器详细设计** 12](#_Toc85704782)

[第三章 系统开发 13](#_Toc85704783)

[3.1 开发环境 13](#_Toc85704784)

[3.2 代码实现 13](#_Toc85704785)

[**3.2.1 发包器代码实现** 13](#_Toc85704786)

[**3.2.2 抓包器代码实现** 16](#_Toc85704787)

[第四章 系统测试 20](#_Toc85704788)

[4.1 环境测试 20](#_Toc85704789)

[4.2 功能测试 21](#_Toc85704790)

[4.3 测试结果 24](#_Toc85704791)

[第五章 特殊问题及解决方案 24](#_Toc85704792)

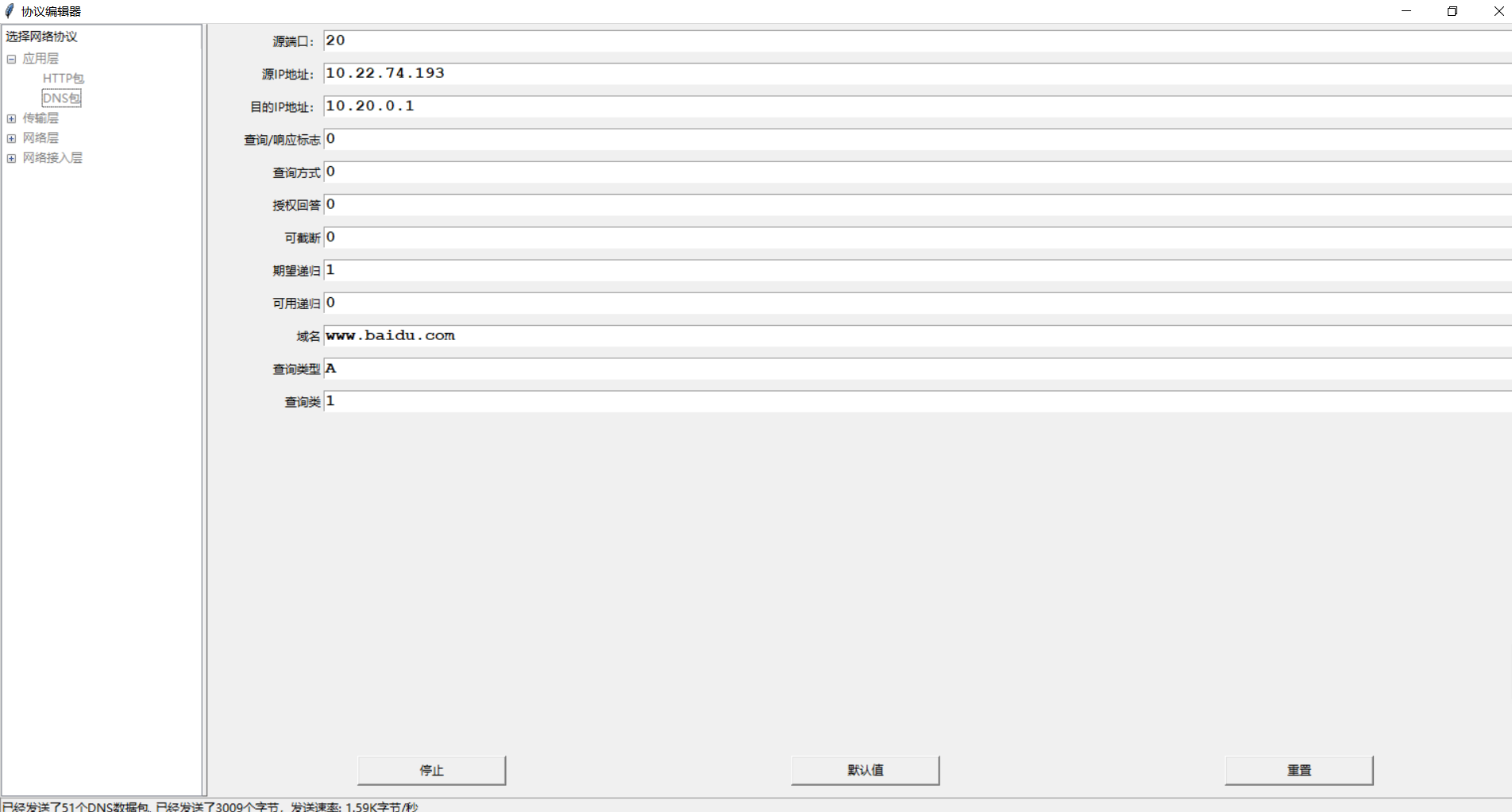
[第六章 总 结 25](#_Toc85704793)

第一章 项目开发基础技术

1.1 项目内容介绍

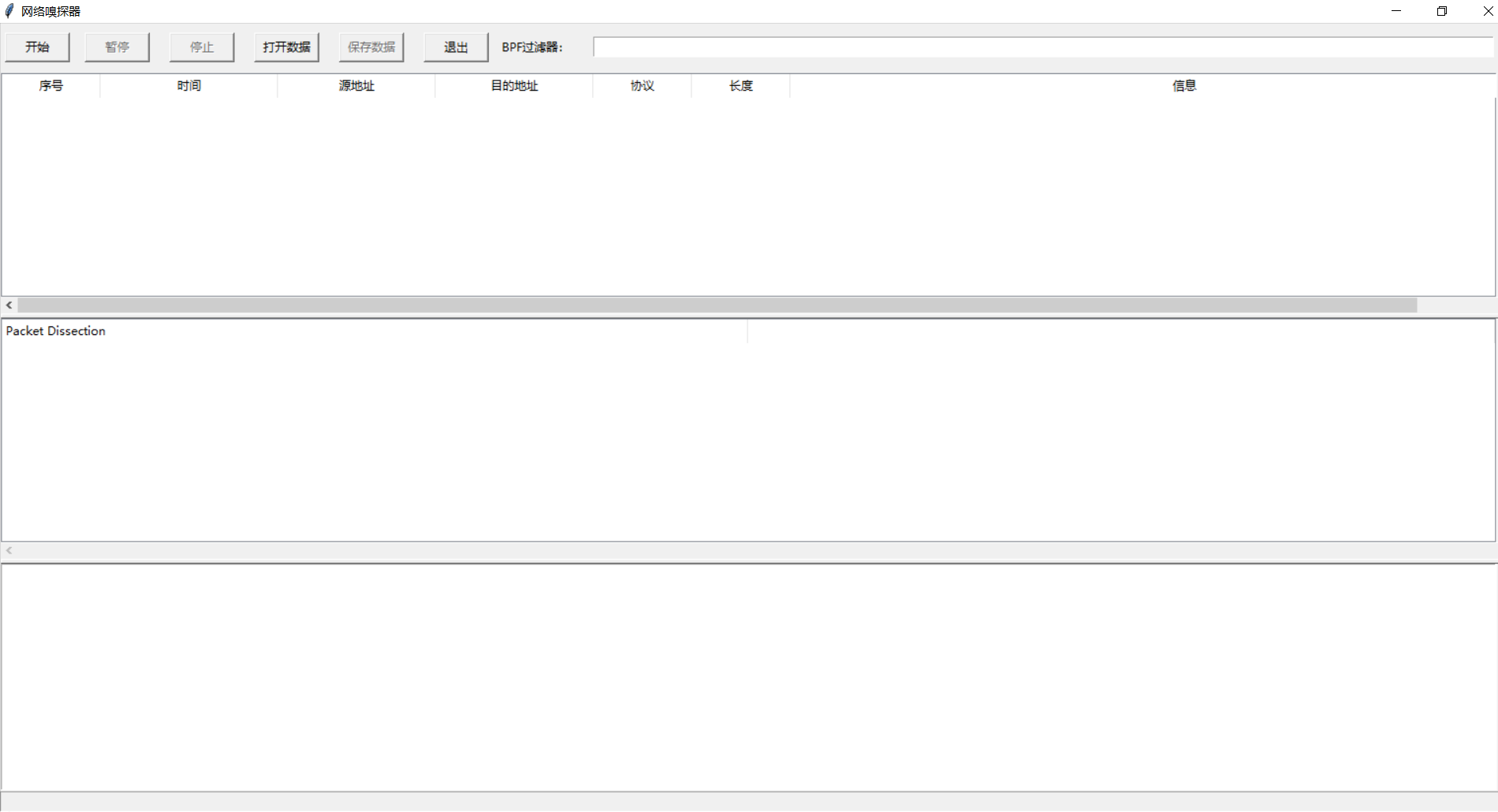
本项目使用python设计一个简单的发包工具与抓包工具。其中使用到了python自带的GUI界面标准库tkinter以及第三方库scapy。

发包器代码量为800行左右，其主要功能包括：（1）各种协议的编辑及发送（2）生成协议字段的默认值（3）数据包发送速率的计算（4）发包可随时开始随时停止



**图1:发包器界面（windows运行）**

抓包器代码量为300行左右，其主要功能包括：（1）主机网卡数据包的抓取（2）单个数据包每层具体协议查看（3）数据包数据部分的十六进制显示（4）信息的存储与读取（5）BPF过滤



**图2:抓包器界面（windows运行）**

1.2 网络协议的基本格式

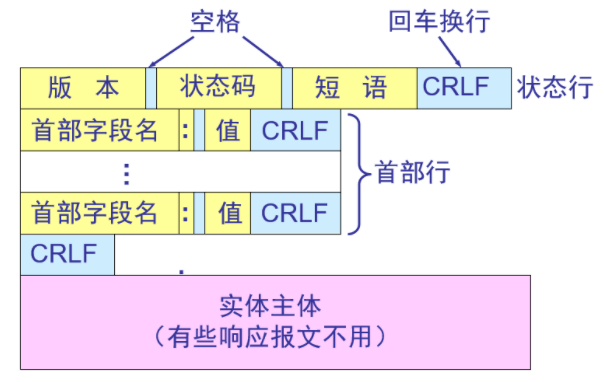
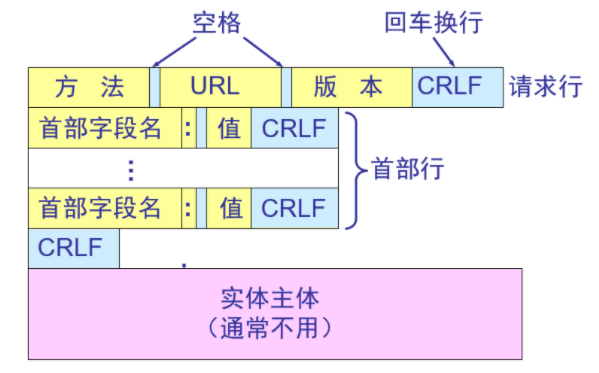
项目支持编辑的协议包括：HTTP、DNS、TCP、UDP、ICMP、IP、ARP、MAC。在开发项目之前，我们必须要了解不同网络协议的基本格式。

**1.2.1 应用层协议**

**1.HTTP协议**

超文本传输协议（Hyper Text Transfer Protocol，HTTP）是一个简单的请求-响应协议，它通常运行在TCP之上。它指定了客户端可能发送给服务器什么样的消息以及得到什么样的响应。请求和响应消息的头以ASCII形式给出；而消息内容则具有一个类似MIME的格式。这个简单模型是早期Web成功的有功之臣，因为它使开发和部署非常地直截了当。

HTTP协议不同于其他网络协议，其有两类报文：请求报文和响应报文，由于 HTTP是面向正文的(text-oriented)，因此在报文中的每一个字段都是一些 ASCII码串，因而每个字段的长度都是不确定的。这两种报文的结构基本上是相同的，都是由三部分组成：开始行、首部行、实体主体。具体HTTP报文格式如下图所示：

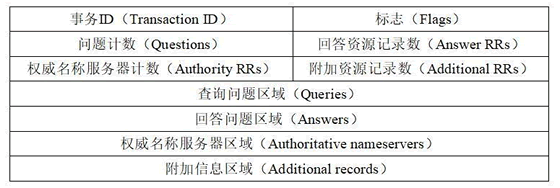


**图3:HTTP报文格式（左为请求报文，右为响应报文）**

**2.DNS协议**

域名解析（Domain name resolution，DNS）是把域名指向网站空间IP，让人们通过注册的域名可以方便地访问到网站的一种服务。IP地址是网络上标识站点的数字地址，为了方便记忆，采用域名来代替IP地址标识站点地址。域名解析就是域名到IP地址的转换过程。域名的解析工作由DNS服务器完成。

整个DNS协议主要分为3部分内容，即基础结构部分、问题部分、资源记录部分。



**图4:DNS报文格式**

对于发包器而言，较为重要的位**基础结构部分**和**问题部分**。

基础结构部分又包含6个部分，在这里我们仅介绍6个部分中的**标志**字段。标志字段又分为若干个字段，如下图所示：



**图5:标志字段**

标志字段中每个字段的含义如下：

**·QR（Response）：**查询请求/响应的标志信息。查询请求时，值为 0；响应时，值为 1。

**·Opcode：**操作码。其中，0 表示标准查询；1 表示反向查询；2 表示服务器状态请求。

**·AA（Authoritative）：**授权应答，该字段在响应报文中有效。值为 1 时，表示名称服务器是权威服务器；值为 0 时，表示不是权威服务器。

**·TC（Truncated）：**表示是否被截断。值为 1 时，表示响应已超过 512 字节并已被截断，只返回前 512 个字节。

**·RD（Recursion Desired）：**期望递归。该字段能在一个查询中设置，并在响应中返回。该标志告诉名称服务器必须处理这个查询，这种方式被称为一个递归查询。如果该位为 0，且被请求的名称服务器没有一个授权回答，它将返回一个能解答该查询的其他名称服务器列表。这种方式被称为迭代查询。

**·RA（Recursion Available）：**可用递归。该字段只出现在响应报文中。当值为 1 时，表示服务器支持递归查询。

**·Z：**保留字段，在所有的请求和应答报文中，它的值必须为 0。

**·rcode（Reply code）：**返回码字段，表示响应的差错状态。当值为 0 时，表示没有错误；当值为 1 时，表示报文格式错误（Format error），服务器不能理解请求的报文；当值为 2 时，表示域名服务器失败（Server failure），因为服务器的原因导致没办法处理这个请求；当值为 3 时，表示名字错误（Name Error），只有对授权域名解析服务器有意义，指出解析的域名不存在；当值为 4 时，表示查询类型不支持（Not Implemented），即域名服务器不支持查询类型；当值为 5 时，表示拒绝（Refused），一般是服务器由于设置的策略拒绝给出应答，如服务器不希望对某些请求者给出应答。

问题部分指的是报文格式中查询问题区域（Queries）部分。该部分是用来显示 DNS 查询请求的问题，通常只有一个问题。该部分包含正在进行的查询信息，包含查询名（被查询主机名字）、查询类型、查询类。问题部分格式如下图所示：



**图6:问题部分**

该部分中每个字段含义如下：

**·查询名：**一般为要查询的域名，有时也会是 IP 地址，用于反向查询。

**·查询类型：**DNS 查询请求的资源类型。通常查询类型为 A 类型，表示由域名获取对应的 IP 地址。

**·查询类：**地址类型，通常为互联网地址，值为 1。

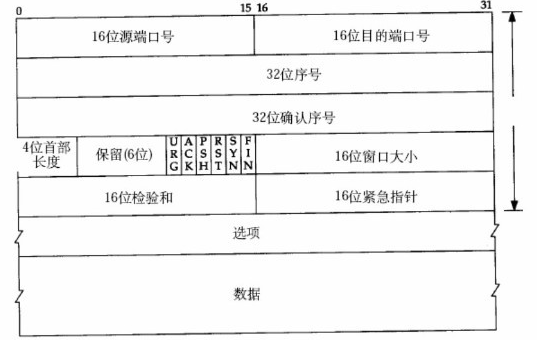
**1.2.2 传输层协议**

**1.TCP协议**

传输控制协议（TCP，Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793 [1] 定义。

TCP旨在适应支持多网络应用的分层协议层次结构。 连接到不同但互连的计算机通信网络的主计算机中的成对进程之间依靠TCP提供可靠的通信服务。TCP假设它可以从较低级别的协议获得简单的，可能不可靠的数据报服务。 原则上，TCP应该能够在从硬线连接到分组交换或电路交换网络的各种通信系统之上操作。

TCP报文的各个数据段比较容易理解，下图为TCP报文的格式：

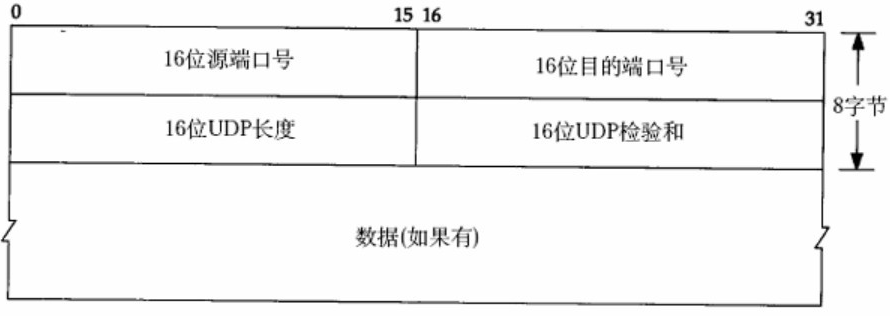


**图7:TCP报文格式**

**2.UDP协议**

Internet 协议集支持一个无连接的传输协议，该协议称为用户数据报协议（UDP，User Datagram Protocol）。UDP 为应用程序提供了一种无需建立连接就可以发送封装的 IP 数据包的方法。RFC 768 [1] 描述了 UDP。

Internet 的传输层有两个主要协议，互为补充。无连接的是 UDP，它除了给应用程序发送数据包功能并允许它们在所需的层次上架构自己的协议之外，几乎没有做什么特别的事情。面向连接的是 TCP，该协议几乎做了所有的事情。下图为UDP协议的报文格式：



**图8:UDP报文格式**

**1.2.3 网络层协议**

**1.IP协议**

IP是Internet Protocol（网际互连协议）的缩写，是TCP/IP体系中的网络层协议。设计IP的目的是提高网络的可扩展性：一是解决互联网问题，实现大规模、异构网络的互联互通；二是分割顶层网络应用和底层网络技术之间的耦合关系，以利于两者的独立发展。根据端到端的设计原则，IP只为主机提供一种无连接、不可靠的、尽力而为的数据包传输服务。下图为具体的IP报文格式：



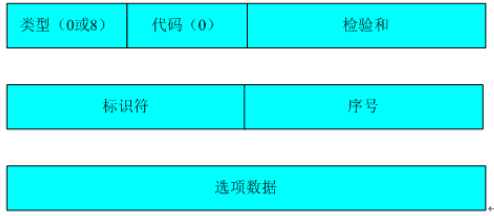
**图9:IP报文格式**

值得一提的是，IP协议包括IPv4和IPv6两个版本。本项目中发包器默认只能发送IPv4协议，而抓包器则可以任意抓取IPv4或IPv6协议。

**2.ICMP协议**

ICMP（Internet Control Message Protocol）Internet控制报文协议。它是TCP/IP协议簇的一个子协议，用于在IP主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

ICMP使用IP的基本支持，就像它是一个更高级别的协议，但是，ICMP实际上是IP的一个组成部分，必须由每个IP模块实现。下图为ICMP报文格式：

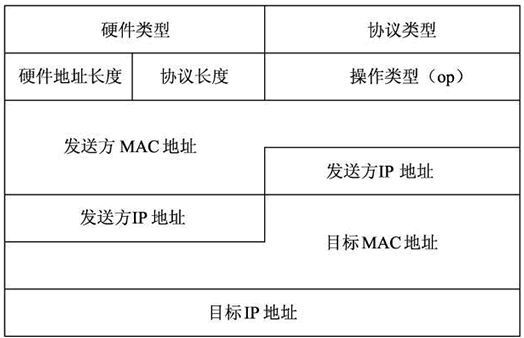


**图10:ICMP报文格式**

**3.ARP协议**

地址解析协议，即ARP（Address Resolution Protocol），是根据IP地址获取物理地址的一个TCP/IP协议。主机发送信息时将包含目标IP地址的ARP请求广播到局域网络上的所有主机，并接收返回消息，以此确定目标的物理地址；收到返回消息后将该IP地址和物理地址存入本机ARP缓存中并保留一定时间，下次请求时直接查询ARP缓存以节约资源。

ARP命令可用于查询本机ARP缓存中IP地址和MAC地址的对应关系、添加或删除静态对应关系等。下图为ARP报文格式：



**图11:ARP报文格式**

**1.2.4 链路层协议**

**1.** **MAC帧**

MAC帧是数据帧的一种。而所谓数据帧，就是数据链路层的协议数据单元，它包括三部分：帧头，数据部分，帧尾。其中，帧头和帧尾包含一些必要的控制信息，比如同步信息、地址信息、差错控制信息等；数据部分则包含网络层传下来的数据，比如ip数据包。

MAC帧的帧头包括三个字段。前两个字段分别为6字节长的目的地址字段和源地址字段，目的地址字段包含目的MAC地址信息，源地址字段包含源MAC地址信息。第三个字段为2字节的类型字段，里面包含的信息用来标志上一层使用的是什么协议，以便接收端把收到的MAC帧的数据部分上交给上一层的这个协议。下图为具体的MAC帧头部：



**图12：MAC帧头部**

第二章 系统分析

2.1 系统功能介绍及实现方案

**2.1.1 发包器功能介绍及实现方案**

**功能1：各种协议的编辑与发送**

对于每一种协议数据包，都主要分为三个模块：（1）协议编辑器模块（2）协议默认值获取模块（3）协议打包模块。而对于协议包的发送，系统则提供统一的发送函数。也就是说，每一种协议数据包均打包为相同或类似的格式，然后调用发送函数进行发送。这样设计可以使系统更加清晰、简洁。

**功能2：生成协议字段的默认值**

想要获取协议字段的默认值，首先要获取当前主机的默认网关。将目的ip地址设置为本地的默认网关ip地址。对于其他字段的值，我们可以使用python第三方库scapy中定义的协议函数，来获取默认的协议包。例如，default\_ip\_packet=IP()，这样便获取了一个默认的ip包，然后获取该ip包的各个字段值作为默认值加入编辑框中即可。

**功能3：数据包发送速率的计算**

对于每一数据包，在发送开始之前我们可以使用len()函数获取数据包的长度packet\_size（单位为bytes），使用datatime.now()函数获取当前时间begin\_time。开始发包后，记录总发包数n，每发一个数据包n+1，并记录发完数据包的时间end\_time。然后运用以下公式便可计算出发包速率：

total\_bytes = packet\_size \* n

total\_time = (end\_time - begin\_time).total\_seconds()

bytes\_per\_second = total\_bytes / total\_time/1024

**功能4：发包可随时开始随时停止**

通过创建一个辅助性的线程事件来实现发包的停止。即使用while循环，判断该线程事件是否被唤醒，如果未被唤醒则一直进行发包操作，当点击“停止”按钮时，立刻唤醒该线程事件，程序跳出循环，发包结束。

**2.1.2 抓包器功能介绍及实现方案**

**功能1：主机网卡数据包的抓取**

直接调用scapy库中的sniff函数即可实现数据包的抓取。sniff函数原型为：sniff(count=0,store=1,offline=None,prn=None,filter=None,L2socket=None,timeout=None,opened\_socket=None,stop\_filter=None,iface=None)

各参数含义如下:

**count:**抓取报的数量，设置为0时则一直捕获

**store:**保存抓取的数据包或者丢弃，1保存，0丢弃

**offline:**从pcap文件中读取数据包，而不进行嗅探，默认为None

**prn:**为每个数据包定义一个回调函数

**filter:**过滤规则，可以在里面定义winreshark里面的过滤语法

**L2socket:**使用给定的L2socket

**timeout:**在给定的事件后停止嗅探，默认为None

**opened\_socket:**对指定的对象使用.recv进行读取

**stop\_filter:**定义一个函数，决定在抓到指定的数据之后停止

**iface:**指定抓包的网卡,不指定则代表所有网卡

**功能2：单个数据包每层具体协议查看**

每抓取一个数据包，程序便会回调一次解析函数，对数据包进行解析。并显示出协议的序号、时间、源地址、目的地址、协议、长度、信息。若点击某个数据包，则会在协议解析区解析此数据包。程序会先清除解析区现有内容。按照协议层次显示数据包，并根据新插入数据项的长度动态调整协议解析区的宽度。其次程序根据协议的不同类型自动计算校验和，并判断校验和是否正确。

**功能3：数据包数据部分的十六进制显示**

同上，若点击某个数据包，程序会在hexdump区显示此数据包的十六进制内容。实现也比较简单，直接调用scapy包的hexdump()函数即可完成解析。

**功能4：信息的存储与读取**

信息的存储只需调用tkinter包中的asksaveasfilename ()函数，并将保存的文件名后缀改为.pcap即可。

读取文件时，先调用tkinter包中的askopenfilename()函数读取文件，然后再次调用sniff函数，并将函数的参数offline设置为获取的文件，即可完成文件的读取。

**功能5：BPF过滤**

在sniff函数中设置filter=fitler\_entry.get()，即获取文本输入框中的内容。若BPF过滤栏为空，则不设置任何过滤条件；若BPF过滤栏中有内容，则将其赋值给filter参数，实现过滤。（BPF过滤栏中输入的内容必须符合BPF过滤的语法规则，否则会报错）

**功能6：抓包可随时停止和暂停**

与发包器一样，通过创建两个线程事件的来完成此项功能。暂停功能与发包器的停止功能实现方式一致。而发包器的停止功能，则是通过sniff函数中的stop\_filter参数，当停止线程事件被唤醒时，进行回调，终止发包。

2.2 系统详细设计

**2.2.1 发包起详细设计**

**主要全局变量：**

|  |  |
| --- | --- |
| **变量名** | **含义** |
| main\_panedwindow | 左右分隔窗体 |
| protocol\_editor\_panedwindow | 协议编辑区窗体 |
| protocols\_tree | 协议导航树 |
| default\_gateway | 当前网卡的默认网关 |
| stop\_sending | 用来终止数据包发送线程的线程事件 |

**主要自定义函数及功能：**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | **输入** | **输出** | **功能描述** |
| create\_protocols\_tree | None | 协议导航树 | 创建协议导航树并作为参数输出 |
| toggle\_protocols\_tree\_state | None | None | 使协议导航树失效 |
| on\_click\_protocols\_tree | TreeView单击事件 | None | 协议导航树单击事件响应函数 |
| create\_protocol\_editor | root: 协议编辑区；  field\_names: 协议字段名列表 | 协议字段编辑框列表 | 创建协议字段编辑区 |
| clear\_protocol\_editor | entries: 协议字段编辑框列表 | None | 清空协议编辑器的当前值 |
| create\_bottom\_buttons | root: 编辑编辑区 | 发送按钮和清空按钮 | 创建发送按钮和重置按钮 |
| create\_xxx\_sender | None | None | 创建相应的协议编辑器，函数名中xxx为具体的协议名，如：ip、dns、mac |
| create\_default\_xxx\_packet | entries: 协议字段编辑框列表 | None | 在协议字段编辑框中填入默认的相应协议包的字段值，函数名中xxx含义同上 |
| send\_xxx\_packet | send\_packet\_button:包发送按钮  entries:协议字段编辑框列表 | None | 发送相应协议的数据包，函数名中xxx含义同上 |
| send\_packet | packet\_to\_send: 待发送的数据包 | None | 用于发送数据包的线程函数，持续发送数据包 |

**2.2.2 抓包器详细设计**

**主要全局变量：**

|  |  |
| --- | --- |
| **变量名** | **含义** |
| stop\_sniff\_event | 用来终止数据包抓取线程的线程事件 |
| pause\_sniff\_event | 用来暂停数据包抓取线程的线程事件 |
| sniff\_count | 抓取总数 |
| sniff\_array | 所有抓取到的报文 |

**主要自定义函数及功能：**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | **输入** | **输出** | **功能描述** |
| timestamp2time | Timestamp:抓取包的时间 | 格式化后的时间字符串 | 时间戳转为格式化的时间字符串 |
| on\_click\_packet\_list\_tree | event: TreeView单击事件 | None | 数据包列表单击事件响应函数，在数据包列表单击某数据包时，在协议解析区解析此数据包，并在hexdump区显示此数据包的十六进制内容 |
| packet\_producer | None | None | 生产者函数，抓取数据包 |
| packet\_consumer | pkt:抓取到的数据包 | None | 消费者函数，用于packet\_producer中sniff的回调，解析数据包 |
| save\_captured\_data\_to\_file | None | None | 将抓到的数据包保存为pcap格式的文件 |
| readPcap | None | None | 读取本地的pcap文件，并显示 |
| start\_capture | None | None | 开始按钮单击响应函数，如果是停止后再次开始捕获，会提示用户保存已经捕获的数据 |
| pause\_capture | None | None | 暂停按钮单击响应函数 |
| stop\_capture | None | None | 停止按钮单击响应函数 |
| quit\_program | None | None | 退出按钮单击响应函数,退出程序前要提示用户保存已经捕获的数据 |

第三章 系统开发

3.1 开发环境

语言：python

系统：win10，ubuntu20.04.3

集成开发环境：Visual Studio Code

3.2 代码实现

由于项目代码量较大，这里我们仅展示部分重要代码。

**3.2.1 发包器代码实现**

**1.创建协议导航树**

def create\_protocols\_tree():

    protocols\_tree.heading('#0', text='选择网络协议', anchor='w')

    # 应用层

    applicatoin\_layer\_tree\_entry = protocols\_tree.insert("", 0, "应用层", text="应用层")  # ""表示父节点是根

    http\_packet\_tree\_entry = protocols\_tree.insert(applicatoin\_layer\_tree\_entry, 0, "HTTP包", text="HTTP包")

    dns\_packet\_tree\_entry = protocols\_tree.insert(applicatoin\_layer\_tree\_entry, 1, "DNS包", text="DNS包")

    # 传输层

    transfer\_layer\_tree\_entry = protocols\_tree.insert("", 1, "传输层", text="传输层")

    tcp\_packet\_tree\_entry = protocols\_tree.insert(transfer\_layer\_tree\_entry, 0, "TCP包", text="TCP包")

    upd\_packet\_tree\_entry = protocols\_tree.insert(transfer\_layer\_tree\_entry, 1, "UDP包", text="UDP包")

    # 网络层

    ip\_layer\_tree\_entry = protocols\_tree.insert("", 2, "网络层", text="网络层")

    ip\_packet\_tree\_entry = protocols\_tree.insert(ip\_layer\_tree\_entry, 0, "IP包", text="IP包")

    icmp\_packet\_tree\_entry = protocols\_tree.insert(ip\_layer\_tree\_entry, 1, "ICMP包", text="ICMP包")

    arp\_packet\_tree\_entry = protocols\_tree.insert(ip\_layer\_tree\_entry, 2, "ARP包", text="ARP包")

    # 网络接入层

    ether\_layer\_tree\_entry = protocols\_tree.insert("", 3, "网络接入层", text="网络接入层")

    mac\_frame\_tree\_entry = protocols\_tree.insert(ether\_layer\_tree\_entry, 1, "MAC帧", text="MAC帧")

    protocols\_tree.bind('<<TreeviewSelect>>', on\_click\_protocols\_tree)

    style = Style(tk)

    disabled\_bg = style.lookup("TEntry", "fieldbackground", ("disabled",))

    style.map("Treeview",

              fieldbackground=[("disabled", disabled\_bg)],

              foreground=[("disabled", "gray")],

              background=[("disabled", disabled\_bg)])

    protocols\_tree.pack()

    return protocols\_tree

**2.创建数据包编辑器（以ICMP包为例）**

def create\_icmp\_sender():

    # ICMP帧编辑区

    icmp\_fields = '类型：', '代码：', '首部校验和：', '标识符：', '序列号：','源IP地址：','目的IP地址:'

    entries = create\_protocol\_editor(protocol\_editor\_panedwindow, icmp\_fields)

    send\_packet\_button, reset\_button, default\_packet\_button = create\_bottom\_buttons(protocol\_editor\_panedwindow)

    # 为"回车键"的Press事件编写事件响应代码，发送ICMP包

    tk.bind('<Return>', (lambda event: send\_icmp\_packet(entries, send\_packet\_button)))  # <Return>代表回车键

    # 为"发送"按钮的单击事件编写事件响应代码，发送ICMP包

    send\_packet\_button.bind('<Button-1>', (

    lambda event: send\_icmp\_packet(entries, send\_packet\_button)))  # <Button-1>代表鼠标左键单击

    # 为"清空"按钮的单击事件编写事件响应代码，清空协议字段编辑框

    reset\_button.bind('<Button-1>', (lambda event: clear\_protocol\_editor(entries)))

    # 为"默认值"按钮的单击事件编写事件响应代码，在协议字段编辑框填入ICMP包字段的默认值

    default\_packet\_button.bind('<Button-1>', (lambda event: create\_default\_icmp\_packet(entries)))

**3.获取数据包默认值（以ICMP包为例）**

def create\_default\_icmp\_packet(entries):

    clear\_protocol\_editor(entries)#清空协议字段编辑框列表

    default\_icmp\_packet = ICMP()

    default\_ip\_packet= IP()

    entries[0].insert(0, int(default\_icmp\_packet.type))

    entries[1].insert(0, int(default\_icmp\_packet.code))

    entries[2]['state'] = NORMAL # 可操作

    entries[2].insert(0, "单机发送时自动计算")

    entries[2]['state'] = DISABLED  # 不可操作

    entries[3].insert(0, hex(default\_icmp\_packet.id))

    entries[4].insert(0, hex(default\_icmp\_packet.seq))

    # 目标IP地址设成本地默认网关

    entries[6].insert(0, default\_gateway)

    default\_ip\_packet = IP(dst=entries[6].get())

    entries[5].insert(0, default\_ip\_packet.src)

**4.数据包打包及发送（以ICMP包为例）**

def send\_icmp\_packet(entries, send\_packet\_button):

    if send\_packet\_button['text'] == '发送':

        icmp\_type = int(entries[0].get())

        icmp\_code = int(entries[1].get())

        icmp\_id = int(entries[3].get(),16)

        icmp\_seq = int(entries[4].get(),16)

        ip\_src = entries[5].get()

        ip\_dst = entries[6].get()

        ip=IP(src=ip\_src, dst=ip\_dst)

        icmp= ICMP(type=icmp\_type,code=icmp\_code,id=icmp\_id,seq=icmp\_seq)

        raw\_packet=raw(ip/icmp)

        packet\_to\_send=IP(raw\_packet)

        packet\_to\_send.show()

        entries[2]['state'] = NORMAL  # 重新激活

        entries[2].delete(0, END)

        entries[2].insert(0, hex(packet\_to\_send.chksum))

        entries[2]['state'] = DISABLED  # 不可操作

        # 开一个线程用于连续发送数据包

        t = threading.Thread(target=send\_packet, args=(packet\_to\_send,))

        t.setDaemon(True)

        t.start()

        # 使协议导航树不可用

        toggle\_protocols\_tree\_state()

        send\_packet\_button['text'] = '停止'

    else:

        # 终止数据包发送线程

        stop\_sending.set()

        # 恢复协议导航树可用

        toggle\_protocols\_tree\_state()

        send\_packet\_button['text'] = '发送'

**5.数据包发送**

def send\_packet(packet\_to\_send):

    n = 0# 对发送的数据包次数进行计数，用于计算发送速度

    stop\_sending.clear()

    packet\_size = len(packet\_to\_send)# 待发送数据包的长度（用于计算发送速度）

    # 推导数据包的协议类型

    proto\_names = ['DNS','HTTP','TCP', 'UDP', 'ICMP', 'IP', 'ARP', 'MAC','Ether', 'Unknown']

    packet\_proto = ''

    for pn in proto\_names:

        if pn in packet\_to\_send:

            packet\_proto = pn

            break

    # 开始发送时间点

    begin\_time = datetime.now()

    while not stop\_sending.is\_set():

        if isinstance(packet\_to\_send, Ether):

            sendp(packet\_to\_send, verbose=0)

        else:

            send(packet\_to\_send, verbose=0)

        n += 1

        end\_time = datetime.now()

        total\_bytes = packet\_size \* n

        total\_time = (end\_time - begin\_time).total\_seconds()

        if total\_time == 0:

            total\_time = 2.23E-308  # 当begin\_time和end\_time相等时，将total\_time设为IEEE 745标准中规定的最小浮点数

        bytes\_per\_second = total\_bytes / total\_time / 1024

        status\_bar.set('已经发送了%d个%s数据包, 已经发送了%d个字节，发送速率: %0.2fK字节/秒',n, packet\_proto, total\_bytes, bytes\_per\_second)

**3.2.2 抓包器代码实现**

**1.抓取数据包**

def packet\_producer():

    sniff(prn=lambda pkt: packet\_consumer(pkt), stop\_filter=lambda pkt: stop\_sniff\_event.is\_set(),

          filter=fitler\_entry.get())

**2.解析数据包**

def packet\_consumer(pkt):

    global sniff\_count

    global sniff\_array

    if not pause\_sniff\_event.is\_set():

        sniff\_count = sniff\_count + 1

        sniff\_array.append(pkt)

        packet\_time = timestamp2time(pkt.time)

        # 推导数据包的协议类型

        proto\_names = ['Raw','DNS','TCP', 'UDP', 'ICMP', 'IPv6', 'IP', 'ARP', 'Ether', 'Unknown']

        proto = ''

        for pn in proto\_names:

            if pn in pkt:

                proto = pn

                break

        if proto == 'ARP' or proto == 'Ether':

            src = pkt.src

            dst = pkt.dst

        else:

            if 'IPv6' in pkt:

                src = pkt[IPv6].src

                dst = pkt[IPv6].dst

            elif 'IP' in pkt:

                src = pkt[IP].src

                dst = pkt[IP].dst

        length = len(pkt)

        info = pkt.summary()

        packet\_list\_tree.insert("", 'end', sniff\_count, text=sniff\_count, values=(sniff\_count, packet\_time, src, dst, proto, length, info))

        packet\_list\_tree.update\_idletasks()

**3.协议解析区解析数据包，并在hexdump区显示数据包的十六进制内容**

def on\_click\_packet\_list\_tree(event):

    global sniff\_array

    selected\_item = event.widget.selection()  # event.widget获取Treeview对象，调用selection获取选择对象名称

    # 清空packet\_dissect\_tree上现有的内容

    packet\_dissect\_tree.delete(\*packet\_dissect\_tree.get\_children())

    # 设置协议解析区的宽度

    packet\_dissect\_tree.column('Dissect', width=packet\_list\_frame.winfo\_width())

    packet = sniff\_array[int(selected\_item[0]) - 1]

    # 按照协议层次显示数据包

    lines = (packet.show(dump=True)).split('\n')

    last\_tree\_entry = None

    for line in lines:

        if line.startswith('#'):

            line = line.strip('# ')

            last\_tree\_entry = packet\_dissect\_tree.insert('', 'end', text=line)

        else:

            packet\_dissect\_tree.insert(last\_tree\_entry, 'end', text=line)

        col\_width = font.Font().measure(line)

        # 根据新插入数据项的长度动态调整协议解析区的宽度

        if packet\_dissect\_tree.column('Dissect', width=None) < col\_width:

            packet\_dissect\_tree.column('Dissect', width=col\_width)

    #计算正确的校验和

    packetCheckSum = Ether(raw(packet))

    # 校验和检查结果

    isIPChkSum = 'Error'

    isTCPChkSum = 'Error'

    isUDPChkSum = 'Error'

    # 检查数据包的校验和

    if 'IP' in packet:

        # 检查IP校验和

        if packetCheckSum[IP].chksum == packet[IP].chksum:

            isIPChkSum = 'OK'

        else:

            isIPChkSum = 'Error'

    if 'TCP' in packet:

        # 检查TCP校验和

        if packetCheckSum[TCP].chksum == packet[TCP].chksum:

            isTCPChkSum = 'OK'

        else:

            isTCPChkSum = 'Error'

    elif 'UDP' in packet:

        # 检查UDP校验和

        if packetCheckSum[UDP].chksum == packet[UDP].chksum:

            isUDPChkSum = 'OK'

        else:

            isUDPChkSum = 'Error'

    elif 'ICMP' in packet:

        #ICMP校验和

        if packetCheckSum[ICMP].chksum == packet[ICMP].chksum:

            isICMPChkSum='OK'

        else:

            isICMPChkSum='Error'

        # 插入校验和显示区

    if 'IP' in packet or 'IPv6' in packet:

        last\_tree\_entry = packet\_dissect\_tree.insert('', 'end', text='校验和')

        packet\_dissect\_tree.insert(last\_tree\_entry, 'end', text='IP校验和:' + isIPChkSum)

    if 'TCP' in packet:

        packet\_dissect\_tree.insert(last\_tree\_entry, 'end', text='TCP校验和:' + isTCPChkSum)

    elif 'UDP' in packet:

        packet\_dissect\_tree.insert(last\_tree\_entry, 'end', text='UDP校验和:' + isUDPChkSum)

    elif 'ICMP' in packet:

        packet\_dissect\_tree.insert(last\_tree\_entry, 'end', text='ICMP校验和:' + isICMPChkSum)

    # 在hexdump区显示此数据包的十六进制内容

    hexdump\_scrolledtext['state'] = 'normal'

    hexdump\_scrolledtext.delete(1.0, END)

    hexdump\_scrolledtext.insert(END, hexdump(packet, dump=True))

    hexdump\_scrolledtext['state'] = 'disabled'

**4.读取文件**

def readPcap():

    filename = askopenfilename(filetypes=[('PCAP Files', '\*.pcap')], title="打开pcap文件")

    if filename != '':

        global sniff\_count

        global sniff\_array

        # 如果是停止状态再打开，提示保存pcap文件

        if sniff\_count != 0:

            save\_captured\_data\_to\_file()

            sniff\_count = 0

            sniff\_array = []

            packet\_list\_tree.delete(\*packet\_list\_tree.get\_children())

            packet\_dissect\_tree.delete(\*packet\_dissect\_tree.get\_children())

            stop\_sniff\_event.clear()

            pause\_sniff\_event.clear()

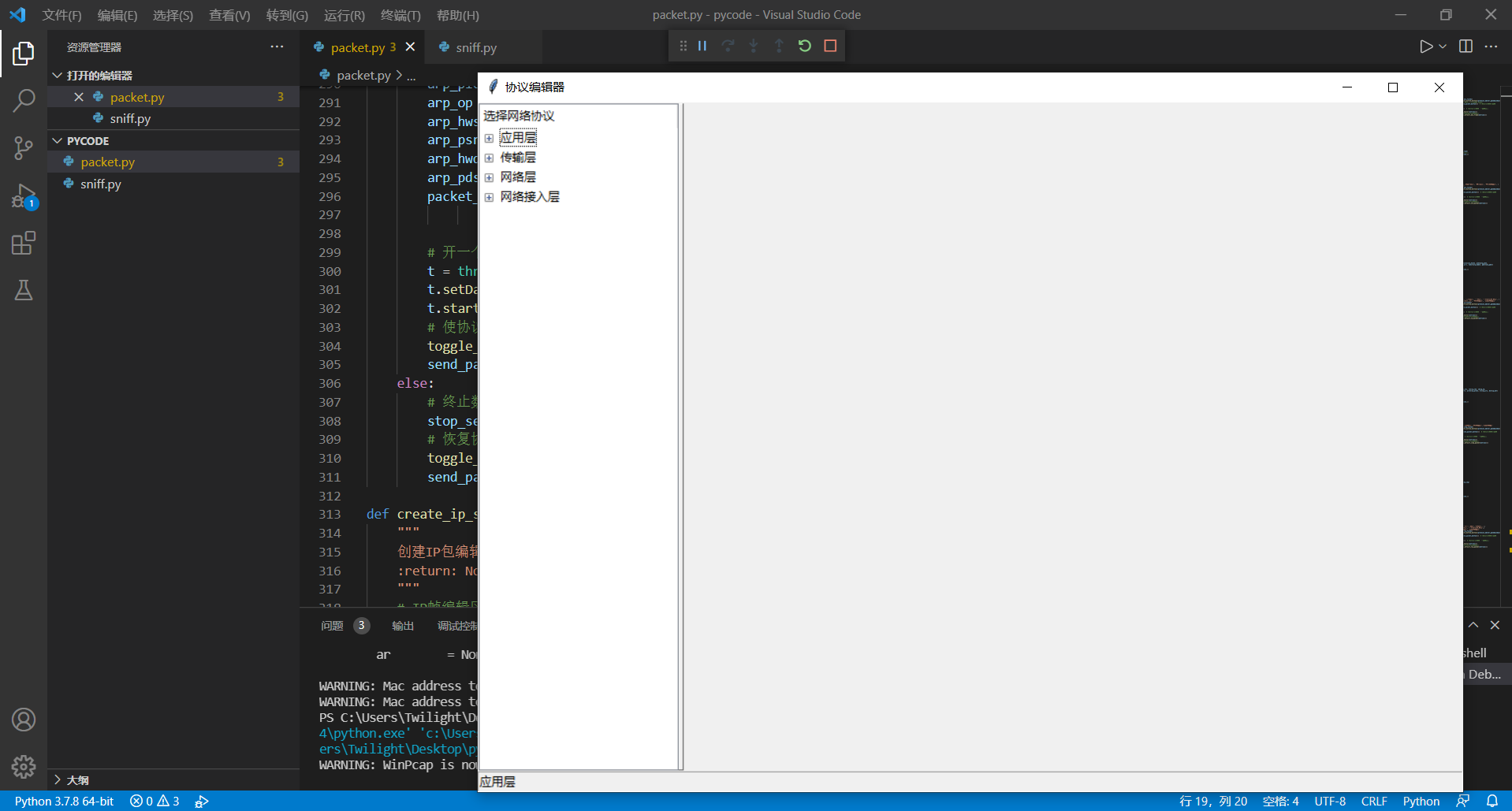
        sniff(prn=lambda pkt: packet\_consumer(pkt), stop\_filter=lambda pkt: stop\_sniff\_event.is\_set(),

          filter=fitler\_entry.get(), offline=filename)

第四章 系统测试

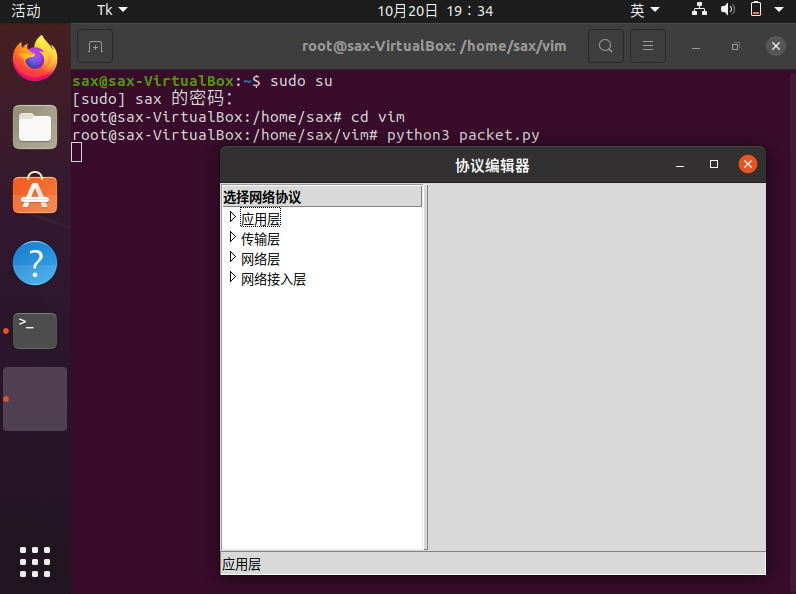
4.1 环境测试

Windows环境下运行：在Visual Studio Code中打开项目文件，键盘输入Ctrl+F5即可运行。运行效果如下：



**图13:windows环境下运行发包器**

Linux环境下运行：在虚拟机中运行Ubuntu操作系统，Ctrl+Alt+T打开终端。键入sudo su输入密码获取管理员权限，进入项目所在的文件夹并输入python3+项目名即可运行。 运行效果如下：



**图14:Linux环境下运行发包器**

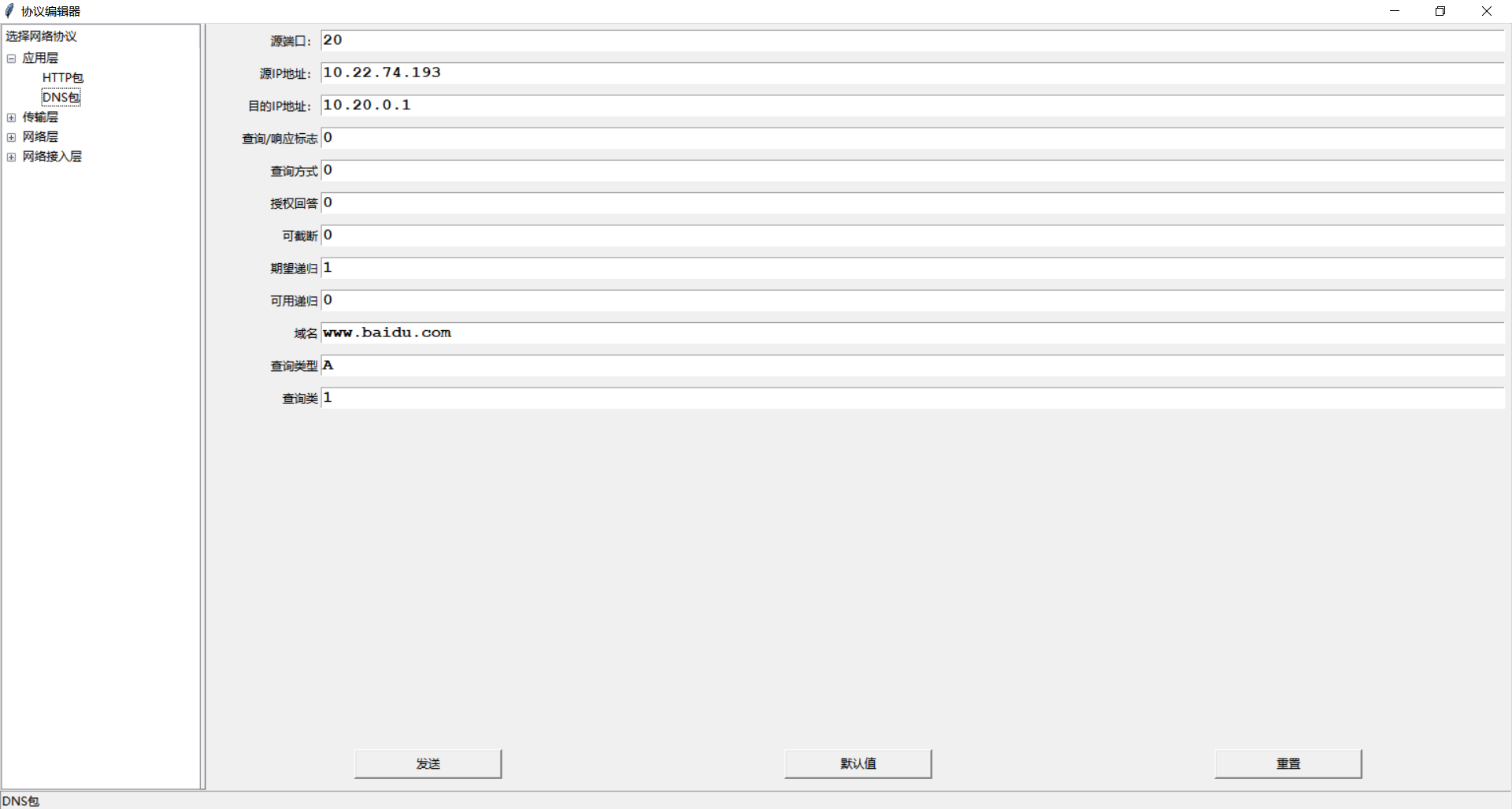
4.2 功能测试

为了便于测试以下测试结果均在windows操作系统下运行。

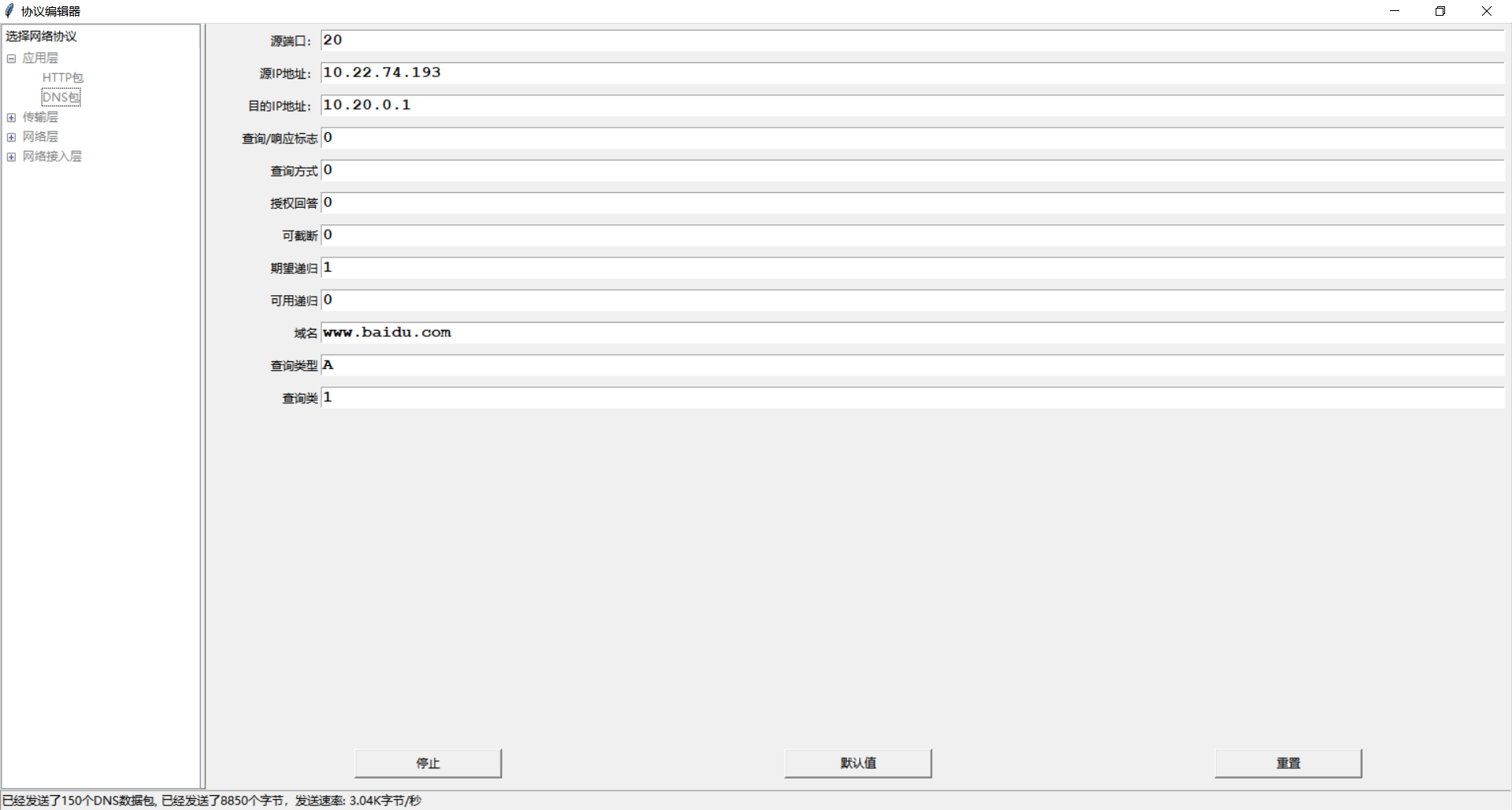
**1.启动发包器，在协议选择栏选择想要发送的数据包（以发DNS包为例）**



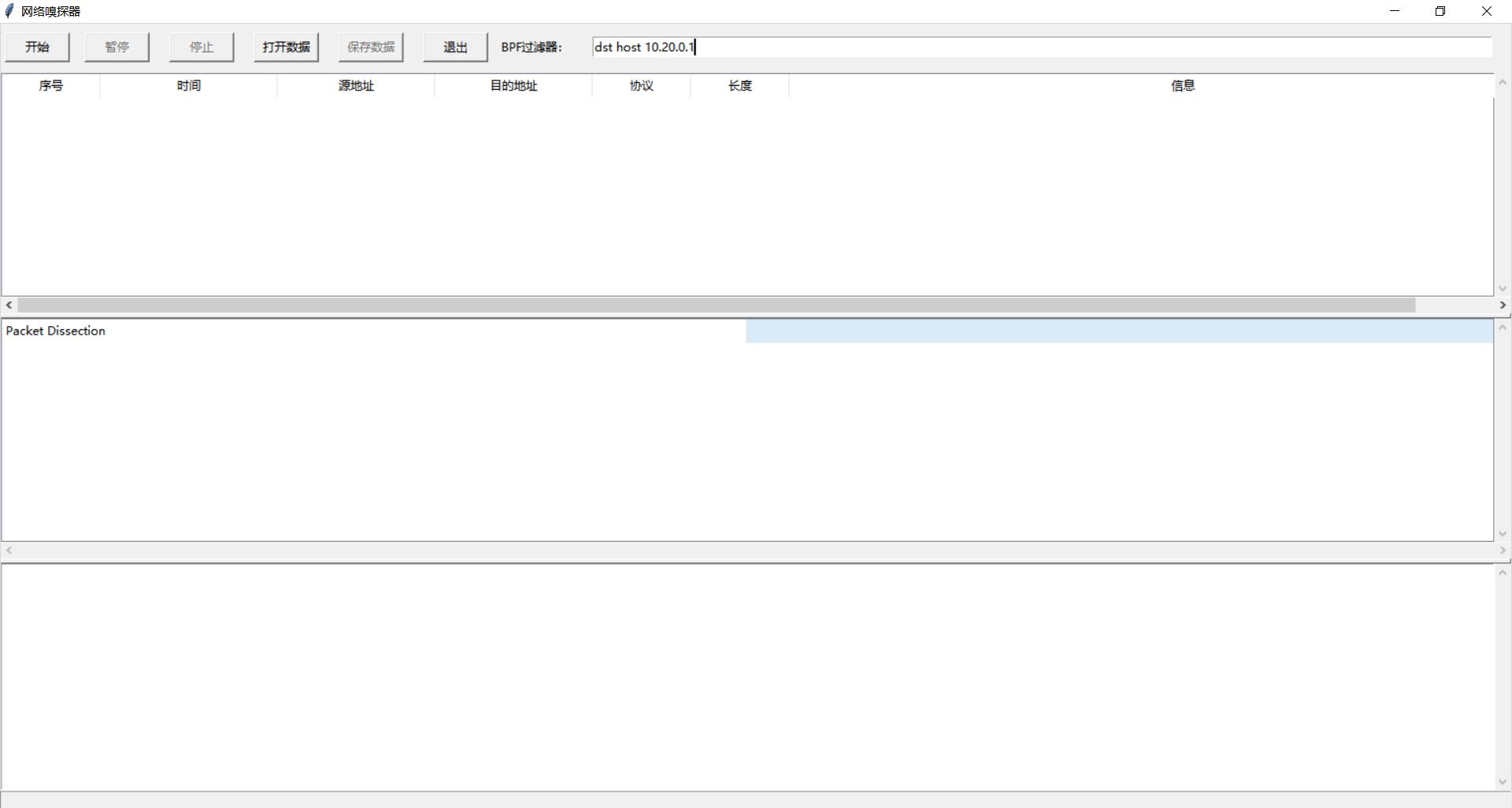
**2.可选择自定义数据或者获取默认这（这里我们选择获取默认值）**



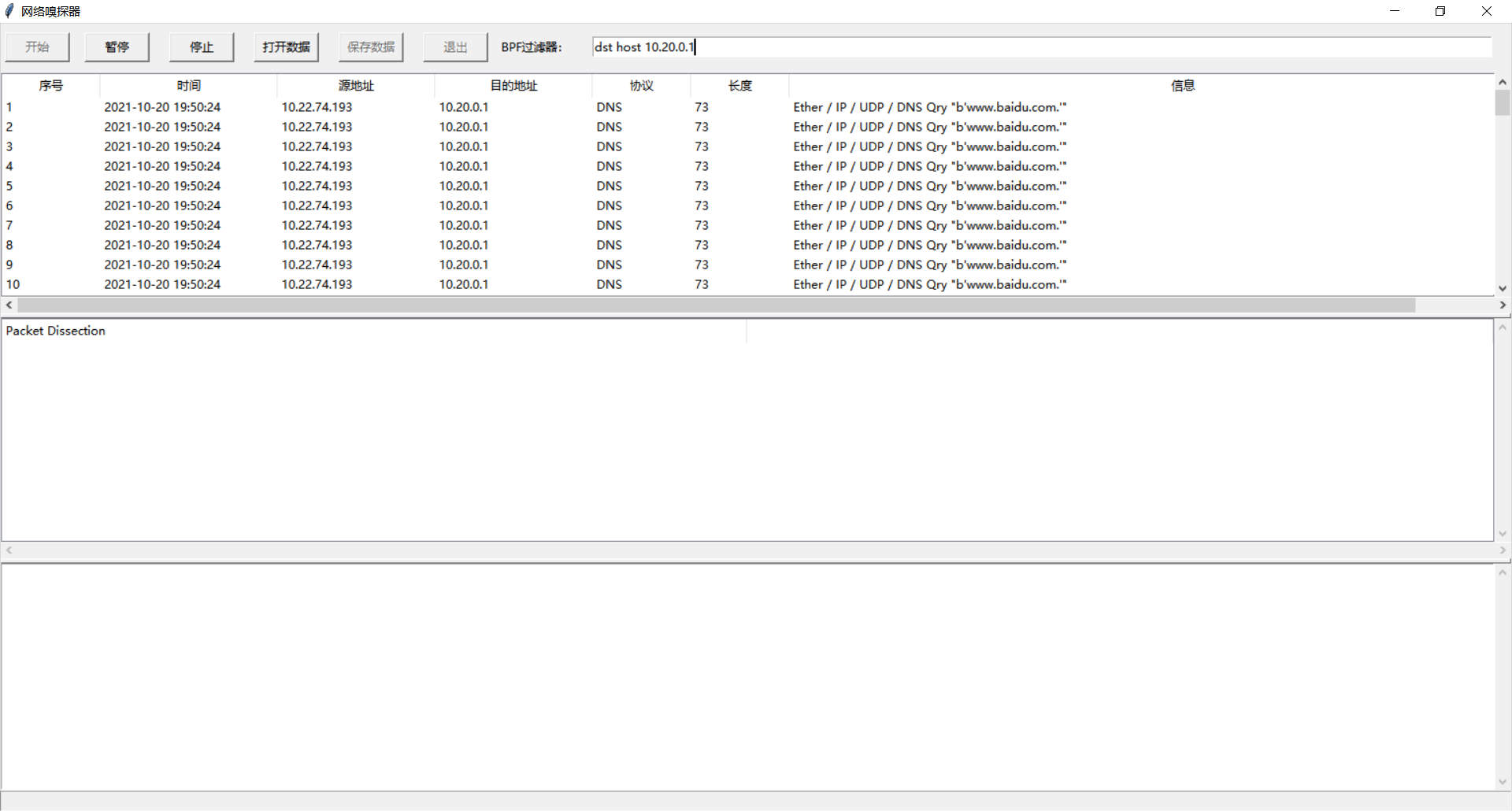
**3.点击发送**



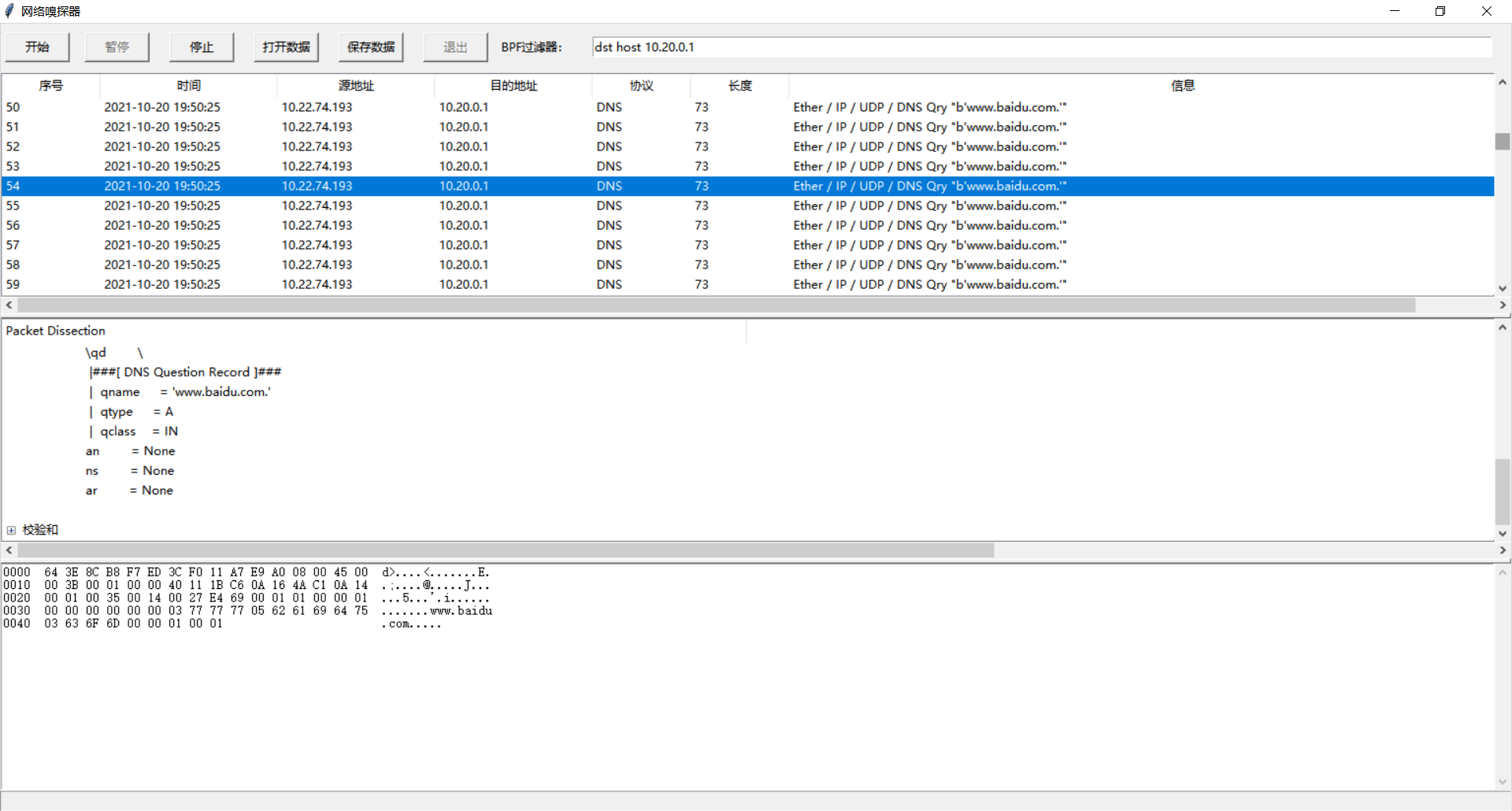
**4.启动抓包器，在BPF过滤器中输入过滤条件**



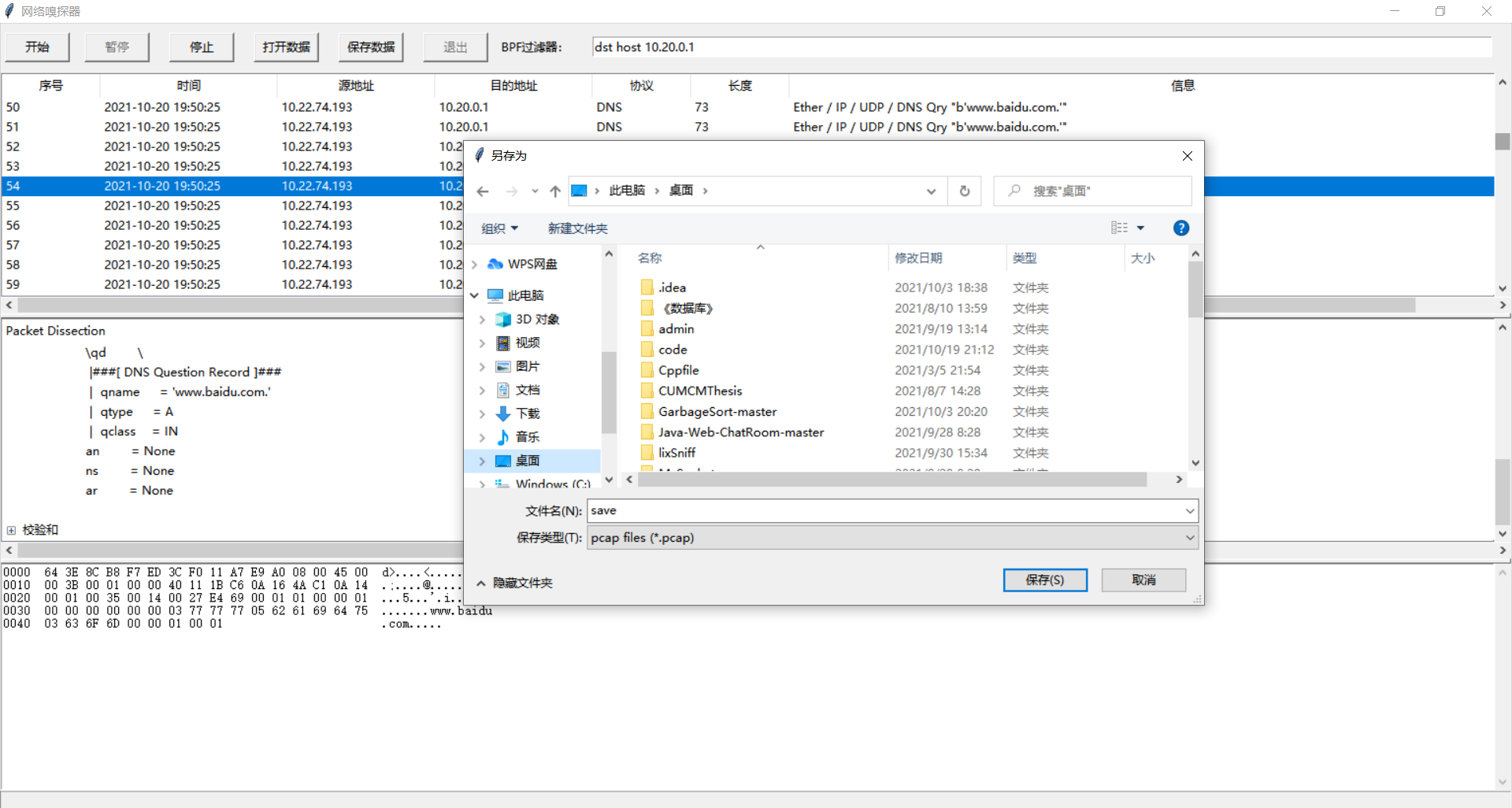
**5.点击开始，抓包器开始捕获数据包**



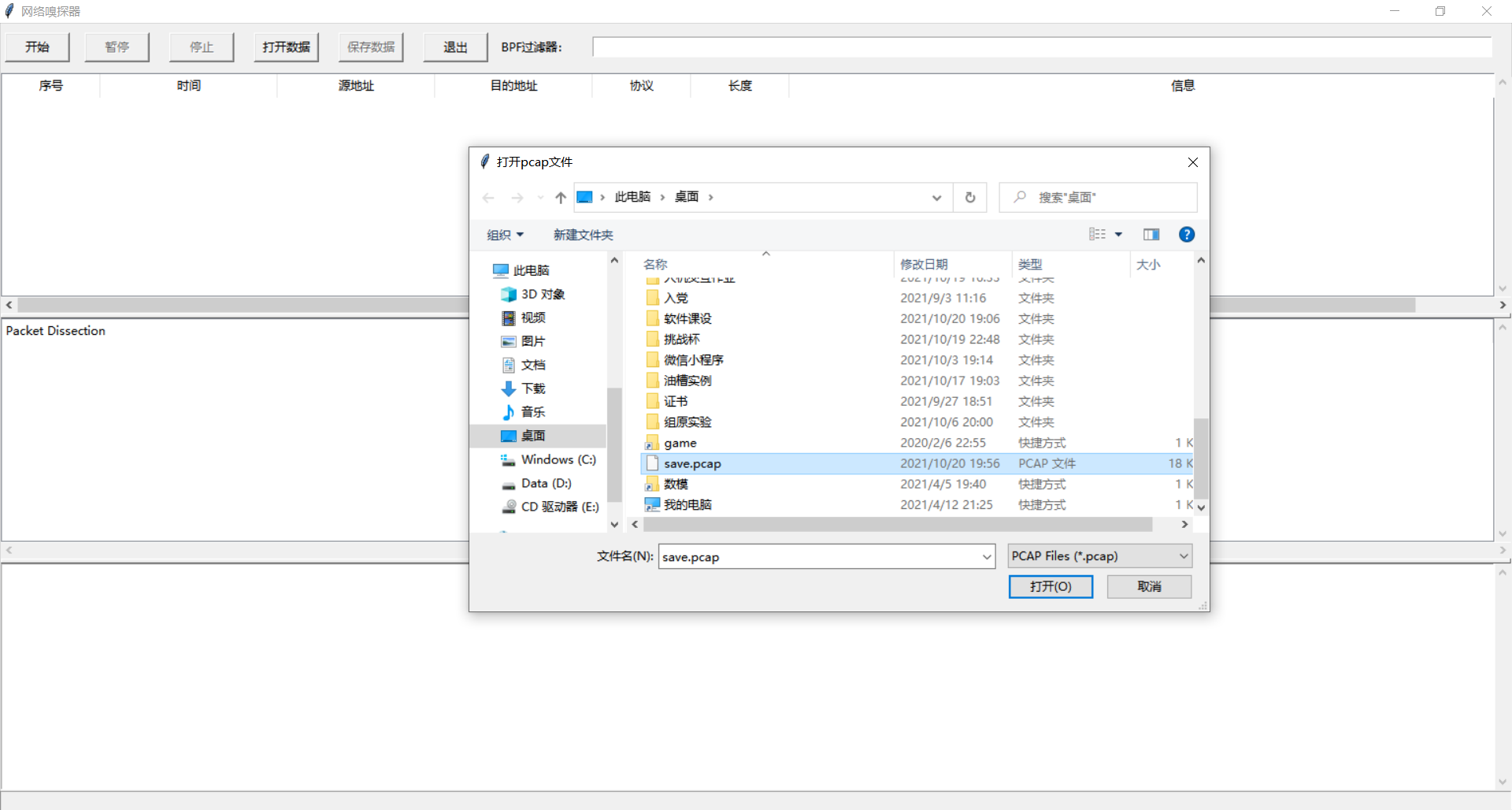
**6.点击停止，抓包器停止抓取。再点击某一具体数据包，可查看其详细信息**

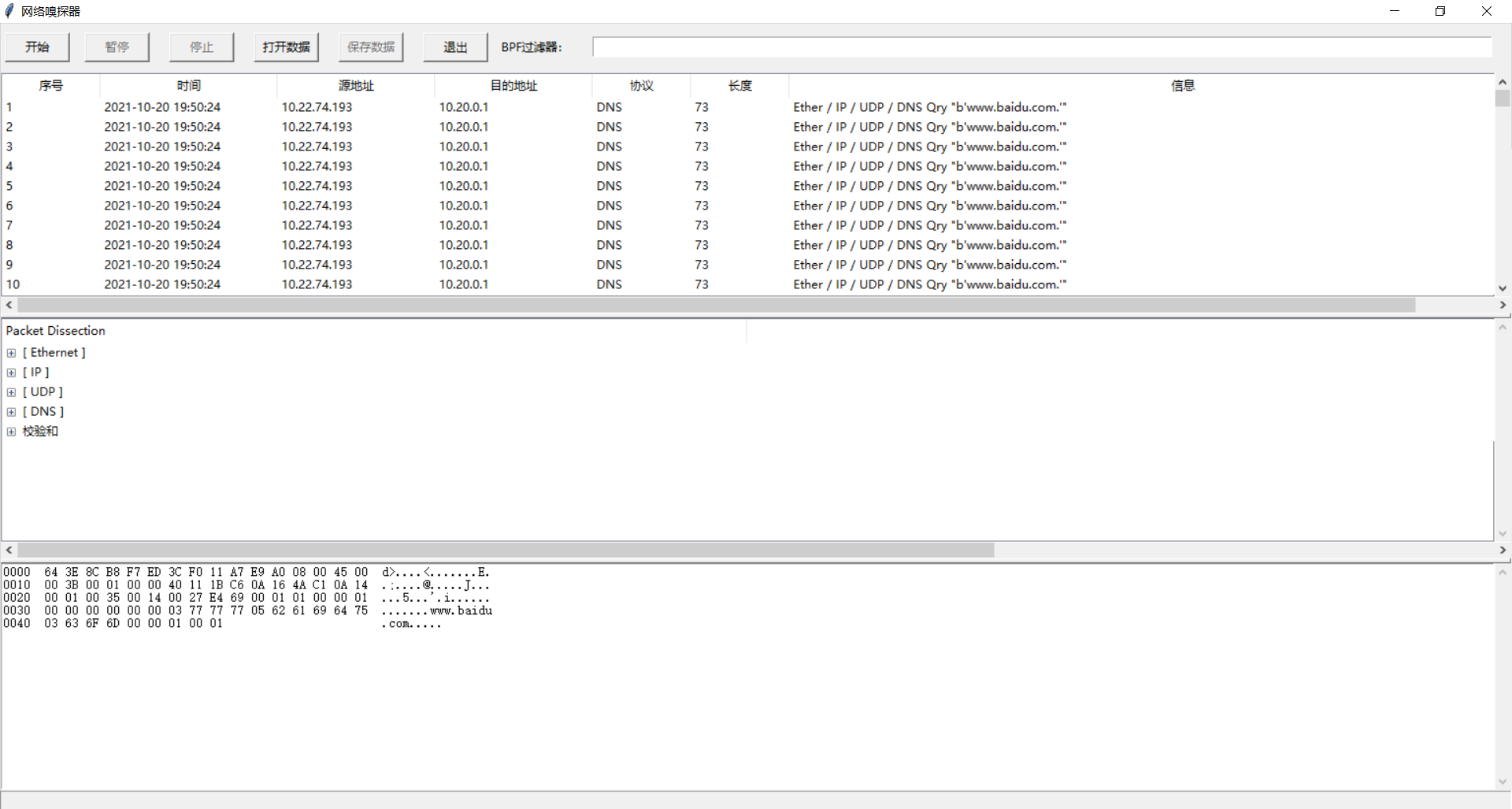


**7.点击保存数据，可以在本地保存抓取到的数据**



**8.点击打开数据，可以打开本地存储的数据**





4.3 测试结果

测试成功，发包器可以成功发送数据包，而抓包器也可以成功捕获发送的数据包。

但值得一提的是，项目也有一些不足之处，比如发包器只能计算并显示发包速度，而不能控制发包速度，导致发包和获取同时进行时会变得卡顿，这一现象在linux虚拟机上尤其严重。其次抓包器的过滤系统的设计也存在一些缺陷，该过滤器要求使用者必须熟悉BPF过滤语法，否则无法使用。

第五章 特殊问题及解决方案

**问题1**

**问题描述：**原有的获取默认网关代码在linux环境下报错，错误信息为数组越界，通过查找相关资料并没有获取到错误的原因。

**解决方案：**在windows环境下获取默认网关的代码为：default\_gateway = [a for a in os.popen('route print').readlines() if ' 0.0.0.0 ' in a][0].split()[-3]。而在linux环境下获取默认网关的代码修改为：default\_gateway=get\_ default\_gateway()。其中

get\_ default\_gateway()函数定义如下：

def get\_default\_gateway():

with open("/proc/net/route") as fh:

for line in fh:

fields = line.strip().split()

if fields[1] != '00000000' or not int(fields[3], 16) & 2:

continue

return socket.inet\_ntoa(struct.pack("<L", int(fields[2], 16)))

**结果：**代码在linux环境下也可正常运行，且可成功获取默认网关

**问题2:**

**问题描述：**在linux系统下运行发包器后，点击发包无响应。运行抓包器也无法捕获任何数据包。系统报错，错误信息为：PermissionError:[Errno 1] Operation not permitted

**解决方案：**在终端中输入sudo su，输入账号密码，获取管理员权限，再次运行代码。

**结果：**两个项目均可正常运行。

**问题3:**

**问题描述：**直接使用sniff函数读取pcap文件，在linux环境下运行无误，但在windows环境下报错，报错信息为：tcpdump is not available

**解决方案：**改写windows环境下的代码，直接使用packets =rdpcap(filename)读取文件，然后使用for循环遍历packets包，每读取一个包便调用一次

packet\_consumer函数进行解析。

**结果：**windows环境下也可读取文件，但使用for循环每次调用消费者函数，其效率十分之低，若pcap文件过大，读取文件时间会有数秒钟。

第六章 总 结

本项目包括一个发包器（协议编辑器）和一个抓包器（网络嗅探器）。项目可在windows系统和Linux系统上运行，且项目运行无误，无严重bug。

本次网络编程的大作业是我第一次使用python开发GUI项目，以前使用python主要是用于数据分析和数学建模。所以有许多不熟悉的地方，或多或少地学习借鉴了网上的资料。

不得不说，python这个语言虽没有C++或是JAVA那样有着悠久的历史。但其功能强大，代码简单明了，对于具体的开发带来了极大的便利。比如本项目中，抓包器的实现，仅仅使用了300行左右的代码就编写出了一个带有图形界面且具有完备功能的项目。而若使用C++的winpcap构建网络嗅探器，则十分繁琐，且GUI界面的构建也比较麻烦。

此次网络编程项目的编写，让我充分了解到网络编程的乐趣，它不像面向过程的程序设计那样枯燥乏味，也不像面向对象的程序设计那样复杂繁琐。以前我总是觉得网络这个东西是高不可攀的，没有办法去改变或者获取网络上的这些数据，但经过几周的学习，我充分认识到了网络编程的奇妙。原来网络并不是看不见摸不着的东西，网络上的各种数据都以数据包的形式在主机与主机之间来回传递。

总而言之，这门课程让我受益匪浅，不仅重温了上学期计算机网络课程中的相关知识，并对这些知识进行巩固加强，还让我知道了网络编程的各种方法，并且提高了代码编写的水平。