

**计算机网络原理**

**课程设计报告**

**题目：**协议分析软件开发及网络协议分析

**学 院 计算机科学学院**

**专 业 计算机科学与技术**

**班 级 1802班**

**姓 名 段星雨**

**学 号**  **41809010218**

**指导教师 王明明**

**2021年7月7日**

目录

[一、任务概述 1](#_Toc76655097)

[1.1 设计目的 1](#_Toc76655098)

[1.2 项目任务和要求 1](#_Toc76655099)

[1.3 参考资料 1](#_Toc76655100)

[二、开发环境 1](#_Toc76655101)

[2.1 开发机环境 1](#_Toc76655102)

[2.2 编程环境 1](#_Toc76655103)

[三、项目需求分析 2](#_Toc76655104)

[3.1 需求分析 2](#_Toc76655105)

[3.2 需求实现功能 2](#_Toc76655106)

[3.3 功能图 2](#_Toc76655107)

[四、项目设计和实现 3](#_Toc76655108)

[4.1 项目流程图 3](#_Toc76655109)

[4.2 项目结构 3](#_Toc76655110)

[4.3 程序数据流转图 4](#_Toc76655111)

[4.3.1 实时抓包和离线数据解析 4](#_Toc76655112)

[4.3.2 网速监控 5](#_Toc76655113)

[4.3.3 获取设备信息 5](#_Toc76655114)

[4.4 项目重点逻辑解析 5](#_Toc76655115)

[4.3.1 install.bat 5](#_Toc76655116)

[4.3.2 go.mod 6](#_Toc76655117)

[4.3.3 conf.yaml 7](#_Toc76655118)

[4.3.4 main方法 7](#_Toc76655119)

[4.3.5 ttype.go 8](#_Toc76655120)

[4.3.6 pcap handler创建 9](#_Toc76655121)

[4.3.7 协议分析示例 9](#_Toc76655122)

[4.3.8 网速监控 10](#_Toc76655123)

[五、程序的运行和测试结果 11](#_Toc76655124)

[5.1 运行方法 11](#_Toc76655125)

[5.2 运行结果 11](#_Toc76655126)

[5.2.1 获取设备信息 run.exe -dn 12](#_Toc76655127)

[5.2.2 实时抓包 run.exe -catch 12](#_Toc76655128)

[5.2.2 pacp文件分析 run.exe -offline 14](#_Toc76655129)

[5.2.3 网速监控 run.exe -speed 16](#_Toc76655130)

[六、设计中的问题及心得 16](#_Toc76655131)

[七、附录 16](#_Toc76655132)

[7.1 程序清单 16](#_Toc76655133)

[7.1.1 install.bat 16](#_Toc76655134)

[7.1.2 conf.yaml 17](#_Toc76655135)

[7.1.3 main.go 17](#_Toc76655136)

[7.1.4 service 19](#_Toc76655137)

[7.1.5 handler 27](#_Toc76655138)

[7.1.6 ttype/ttype.go 31](#_Toc76655139)

[7.1.6 go.mod 34](#_Toc76655140)

# 一、任务概述

## 1.1 设计目的

深入理解和掌握计算机网络的基本理论及工作原理，熟悉计算机网络和互联网的组成，培养对计算机网络系统的认知、设计与应用开发能力，同时培养运用计算机网络技术解决实际问题的能力。

## 1.2 项目任务和要求

网络协议分层结构，网络协议工作过程，协议数据单元结构，协议分析，协议设计，协议分析软件开发。通过课程设计理解网络协议的层次结构及网络协议的工作过程，掌握网络协议的分析方法及网络协议的设计与应用。

## 1.3 参考资料

Gopacket github 源码

<https://github.com/google/gopacket>

Gopacket 官方Go doc文档

<https://pkg.go.dev/github.com/google/gopacket>

# 二、开发环境

## 2.1 开发机环境

操作系统：Windows10 20H2

CPU：AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

RAM：16GB

## 2.2 编程环境

语言：golang

语言版本：1.15.6 windows/amd64

IDE：Goland 2021.1

抓包软件：Npcap version 1.10, based on libpcap version 1.9.1

# 三、项目需求分析

## 3.1 需求分析

现阶段已存在的对网络协议的分析的软件较少，主要有两者。一者是wireshark，是将网卡变为混乱模式，然后对抓取指定的数据包，对不同layer分析。另一者是charles，主要使用代理模式，作为应用层的http协议进行抓取分析。两者的劣势也很明显wireshark可以很好的对任何层分析，但是抓取数据量太多，而且并不会对高级层协议多包重组，charles则是只能对http协议分析，又因为使用了代理模式，需要配置ca证书，抓取成功率也降低了。

所有现在打算开发一款软件可以类似于wireshark，并且可以将结果美化为html和json格式，方便用户使用。

## 3.2 需求实现功能

考虑协议较多，本软件只对ethernet，ip，tcp，udp，dns协议进行解析，具体功能如下：

a. 获取当前机器的所有网卡

b. 实时解析某一网口的包信息，保存pacp文件，将数据转化为json和html

c. 离线解析pcap文件，将数据转化为json和html

d. 实时网速解析插件，可以实时获取网络速度

## 3.3 功能图

协议分析软件

网卡分析

网络包分析

网速分析

网卡信息展示

离线数据分析

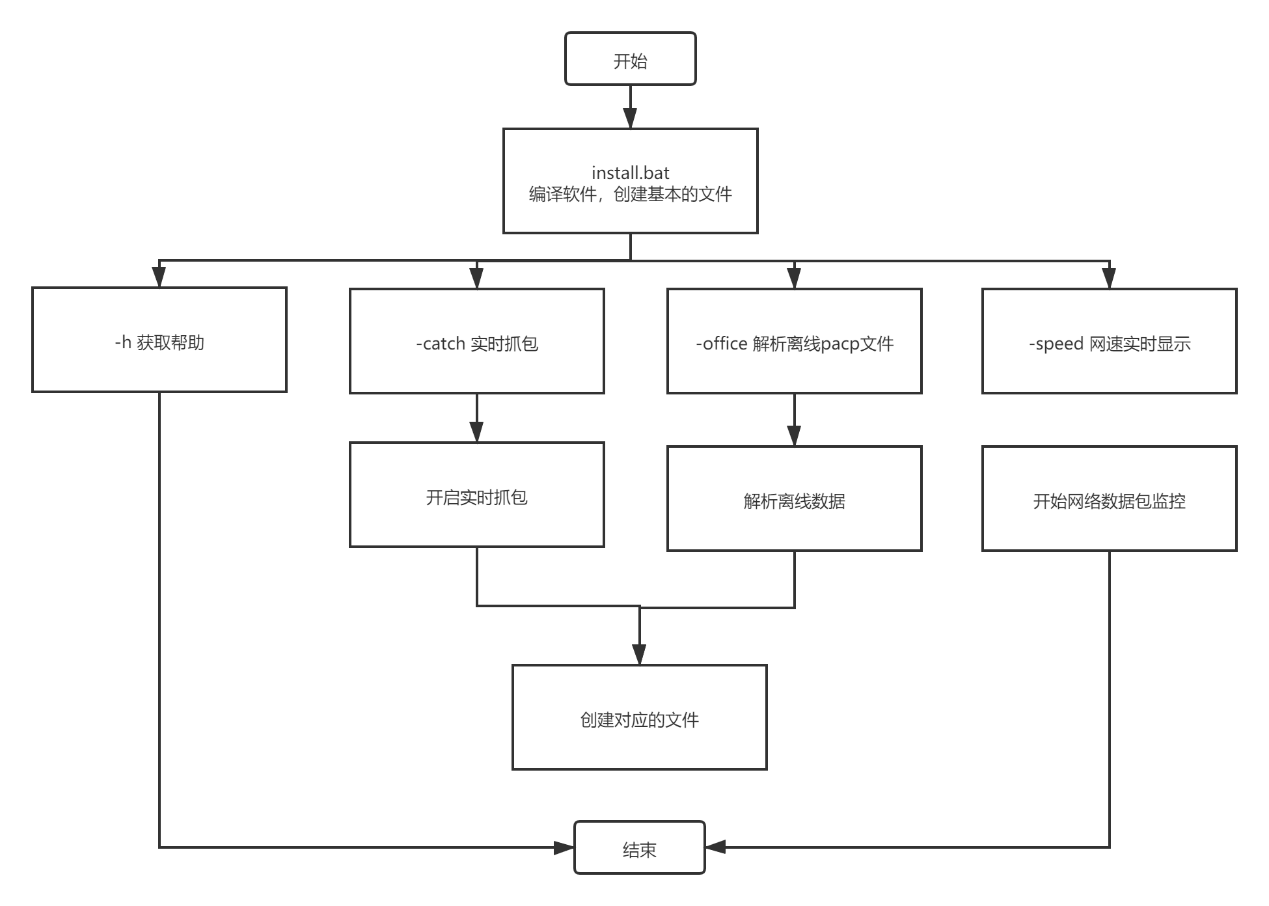
实时数据分析

网速实时解析

# 四、项目设计和实现

项目源码已上传至github：<https://github.com/971181317/net-toll>

## 4.1 项目流程图



## 4.2 项目结构

│ conf.yaml 配置yaml文件

│ go.mod go语言依赖文件

│ go.sum 由go mod tidy指令拉包后生成

│ install.bat 编译bat脚本

│ main.go main函数

│ run.exe 由install.bat生成的可执行文件

│

├─.idea goland相关配置

│ ......

│

├─handler

│ catch.go 抓包和包解析相关handler

│ device.go 设备相关handler

│ speed.go 网速相关handler

│

├─lib 资源文件夹，储存示例和前端资源

│ bootstrap.min.css bootstrap资源

│ bootstrap.min.js bootstrap资源

│ jquery-1.11.0.min.js jquery

│ test.html 示例文件

│ test.pcap 示例文件

│

├─offline\_data

│ ├─1625621786 以时间戳为名称的文件夹，里面包含最终的结果

│ │ pocket data.html

│ │ pocket data.json

│ │ pocket data.pcap

│

├─service

│ catch.go 抓包和包解析相关handler

│ device.go 设备相关handler

│ speed.go 网速相关handler

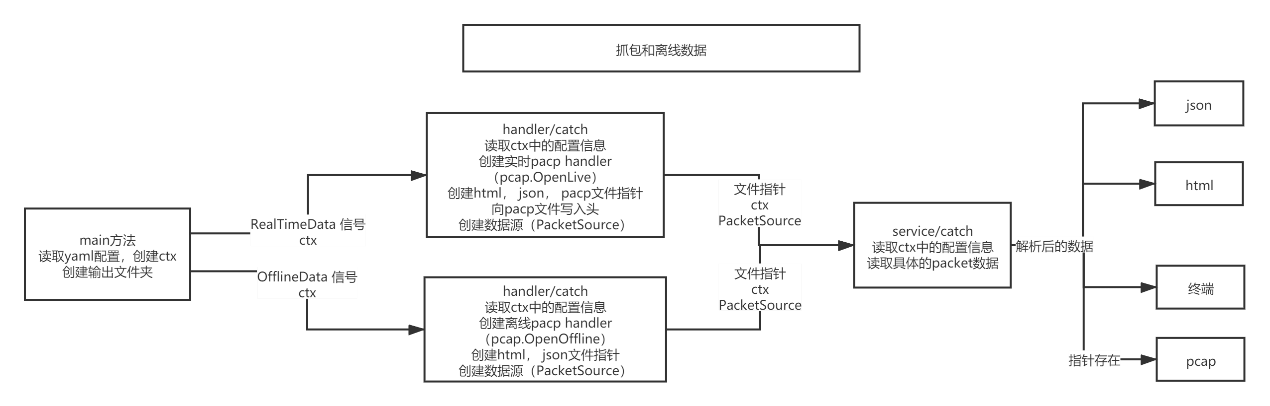
│

└─ttype

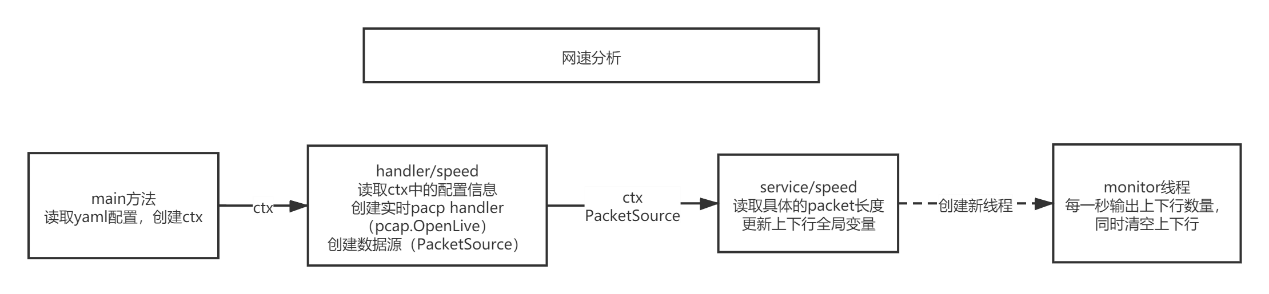
ttype.go 定义程序中使用的相关类型，包括context，html文件内容等

## 4.3 程序数据流转图

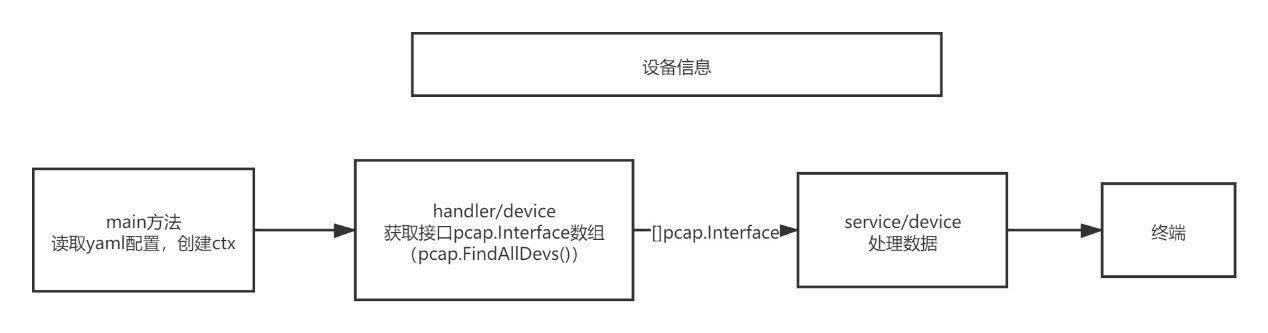
### 4.3.1 实时抓包和离线数据解析



### 4.3.2 网速监控



### 4.3.3 获取设备信息



## 4.4 项目重点逻辑解析

### 4.3.1 install.bat

本文件用于编译go语言生成可执行程序，并且会判断资源文件。因为cmd编码问题，修改编码后编译程序utf-8，之后控制台编码不统一，暂时使用英文输出

@echo off

rem 判断go环境是否存在

if not exist %GOPATH% echo go env is not exist && goto end

rem 判断资源文件，不影响编译

if not exist ./conf.yaml echo 'conf.yaml' is not exist

if not exist ./lib echo 'lib' dir is not exist

if not exist ./offline\_data mkdir offline\_data && echo offline\_data dir is create

echo Compiling............

rem go拉包

go mod tidy

rem 开始编译

if exist ./main.go go build -o run.exe

if exist ./run.exe (goto success) else (goto false)

:success

echo Compilation is complete

echo Please use the run.exe file in the directory to run

goto end

:false

echo Compilation Error

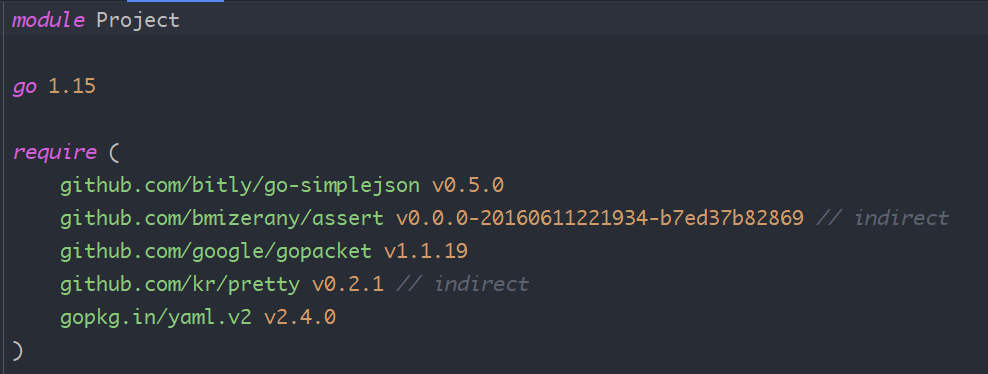
goto end

:end

echo install program exit

### 4.3.2 go.mod

go语言依赖项目



主要依赖为

simplejson：封装json内容

gopacket：go原生使用libpacp api

yaml.v2：解析yaml文件使用

### 4.3.3 conf.yaml

该文件是软件的配置文件，可以定义设备名称，捕获字节数，混杂模式，珠宝数量，抓包间隔

# wifi

DeviceName: \Device\NPF\_{E01139AE-3D36-44EB-8F8A-712BBB555F29}

# 以太网

#DeviceName: \Device\NPF\_{B275C137-2F67-4C36-A2FF-21047C0BBB28}

# 捕获一个数据包的多少个节

SnapshotLen: 1024

# 设置网卡是否工作在混杂模式

Promiscuous: false

#Filter: "tcp"

OfflineFilePath: offline\_data/1625621786/pocket data.pcap

LinkType: 1

# 抓包数量

PacketCount: 100

# 抓包间隔，-1为持续

TimeoutSecond: -1

### 4.3.4 main方法

在main.go中，用于处理控制台参数，并指向对应的方法

func main() {  
 fmt.Println(pcap.Version())  
 InitConf()  
 //读取控制台参数  
 args := os.Args  
 //处理无参数情况  
 if len(args) == 1 {  
 fmt.Print("-h 获取帮助")  
 } else {  
 switch args[1] {  
 case "-h":  
 fmt.Println("-dn 获取设备信息")  
 fmt.Println("-catch 实时抓包，保存内容到filePath")  
 fmt.Println("-offline 从filePath中获取解析离线数据")  
 fmt.Println("-speed 监控实时网速")  
 case "-dn":  
 GetDeviceNameHandel()  
 case "-catch":  
 InitFile()  
 PacketHandler(RealTimeData, &confCtx)  
 //分析离线数据  
 case "-offline":  
 InitFile()  
 PacketHandler(OfflineData, &confCtx)  
 case "-speed":  
 SpeedHandler(&confCtx)  
 default:  
 fmt.Print("错误参数, -h获取帮助")  
 }  
 }  
}

### 4.3.5 ttype.go

数据类型定义，其中包含ctx定义，datasource定义，html文件内容。

ctx定义，继承了context.Context，包含了struct tag，方便解析yaml

// ConfigCtx 配置

type ConfigCtx struct {

context.Context

DeviceName string `yaml:"DeviceName"`

SnapshotLen int32 `yaml:"SnapshotLen"` //捕获一个数据包的多少个字节

Promiscuous bool `yaml:"Promiscuous"` //设置网卡是否工作在混杂模式

Filter string `yaml:"Filter"`

OfflineFilePath string `yaml:"OfflineFilePath"`

LinkType layers.LinkType `yaml:"LinkType"`

PacketCount int32 `yaml:"PacketCount"`

TimeoutSecond int64 `yaml:"TimeoutSecond"` //设置抓到包返回的超时

Timeout time.Duration

RealTimeFilePath string

JsonFilePath string

HtmlFilePath string

DataSource DataSource

}

DataSource类型定义，用于识别离线数据和实时数据

// DataSource 数据源

type DataSource uint8

const (

OfflineData DataSource = 1 //离线数据

RealTimeData DataSource = 2 //实时数据

)

### 4.3.6 pcap handler创建

handler/catch.go，抓包动作的创建handler的情况

// PacketHandler 开始抓取

func PacketHandler(dataSource DataSource, ctx \*ConfigCtx) {

if dataSource == OfflineData {

handle, err = pcap.OpenOffline(ctx.OfflineFilePath)

} else if dataSource == RealTimeData {

handle, err = pcap.OpenLive(ctx.DeviceName, ctx.SnapshotLen, ctx.Promiscuous, ctx.Timeout)

}

if err != nil {

fmt.Printf("开启抓包错误 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

//关闭handle

defer handle.Close()

//过滤

if ctx.Filter != "" {

err = handle.SetBPFFilter(ctx.Filter)

if err != nil {

fmt.Printf("过滤设置错误 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

}

…………..

}

### 4.3.7 协议分析示例

Service/catch.go

// analysisData 分析数据

func analysisData(packet gopacket.Packet, htmlF \*os.File, json \*sj.Json) {

……..

// 以太网协议

ethernetLayer := packet.Layer(layers.LayerTypeEthernet)

if ethernetLayer != nil {

ethernetLayerJson := map[string]interface{}{}

fmt.Println("以太网：")

ethernetPacket, \_ := ethernetLayer.(\*layers.Ethernet)

fmt.Println("- 源 MAC: ", ethernetPacket.SrcMAC.String())

fmt.Println("- 目标 MAC: ", ethernetPacket.DstMAC.String())

fmt.Println("- 以太网类型: ", ethernetPacket.EthernetType)

fmt.Println()

ethernetLayerJson["SrcMAC"] = ethernetPacket.SrcMAC.String()

ethernetLayerJson["DstMAC"] = ethernetPacket.DstMAC.String()

ethernetLayerJson["EthernetType"] = ethernetPacket.EthernetType

layersJson["ethernet"] = ethernetLayerJson

//写详细数据到html

fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")

fmt.Fprintln(htmlF, "<h2 class=\"text-center\">Ethernet</h2>")

fmt.Fprintf(htmlF, "<p>源 MAC&nbsp;:&nbsp;&nbsp;%s<p>\n", ethernetPacket.SrcMAC.String())

fmt.Fprintf(htmlF, "<p>目标 MAC&nbsp;:&nbsp;&nbsp; %s<p>\n", ethernetPacket.DstMAC.String())

fmt.Fprintf(htmlF, "<p>以太网类型&nbsp;:&nbsp;&nbsp;%s<p>\n", ethernetPacket.EthernetType)

fmt.Fprintln(htmlF, "</div></div>")

}

…….

}

### 4.3.8 网速监控

Service/speed.go

var (

downStreamDataSize = 0 // 单位时间内下行的总字节数

upStreamDataSize = 0 // 单位时间内上行的总字节数

)

func SpeedService(packetSource \*gopacket.PacketSource, macAddr string, ctx \*ttype.ConfigCtx) {

// 开启子线程，每一秒计算一次该秒内的数据包大小平均值，并将下载、上传总量置零

go monitor()

// 开始抓包

for packet := range packetSource.Packets() {

// 只获取以太网帧

ethernetLayer := packet.Layer(layers.LayerTypeEthernet)

if ethernetLayer != nil {

ethernet := ethernetLayer.(\*layers.Ethernet)

// 如果封包的目的MAC是本机则表示是下行的数据包，否则为上行

if ethernet.DstMAC.String() == macAddr {

downStreamDataSize += len(packet.Data()) // 统计下行封包总大小

} else {

upStreamDataSize += len(packet.Data()) // 统计上行封包总大小

}

}

}

}

// 每一秒计算一次该秒内的数据包大小平均值，并将下载、上传总量置零

func monitor() {

for {

fmt.Print(fmt.Sprintf("\rDown:%.2fkb/s \t Up:%.2fkb/s", float32(downStreamDataSize)/1024/1, float32(upStreamDataSize)/1024/1))

downStreamDataSize = 0

upStreamDataSize = 0

time.Sleep(1 \* time.Second)

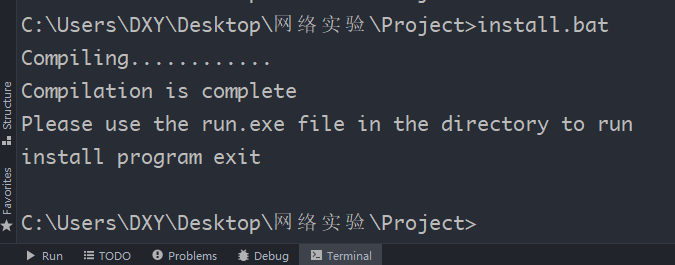
}

}

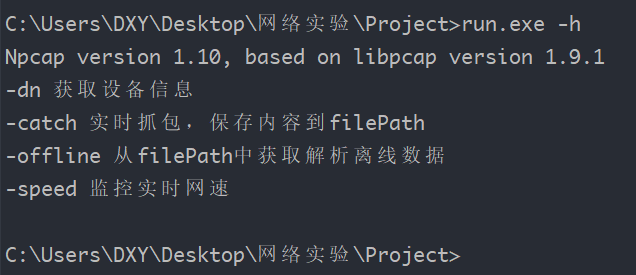
# 五、程序的运行和测试结果

## 5.1 运行方法

a.运行根目录下的install.bat，之后会在根目录生成run.exe的可执行文件



b. 运行 run.exe -h 显示出提示



## 5.2 运行结果

### 5.2.1 获取设备信息 run.exe -dn

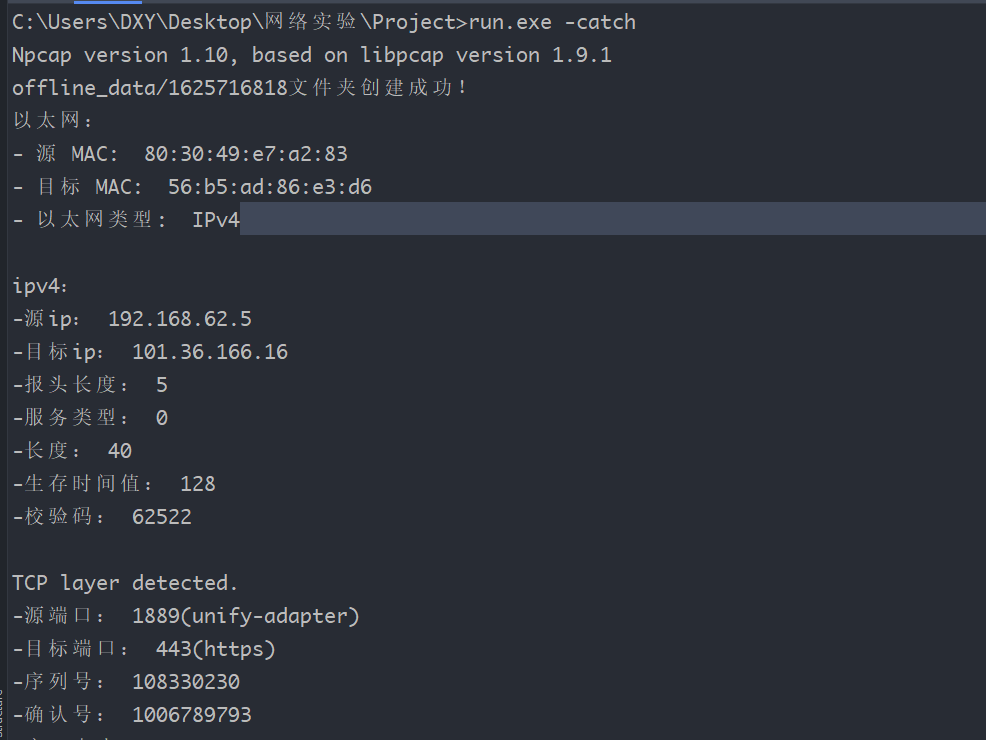


### 5.2.2 实时抓包 run.exe -catch

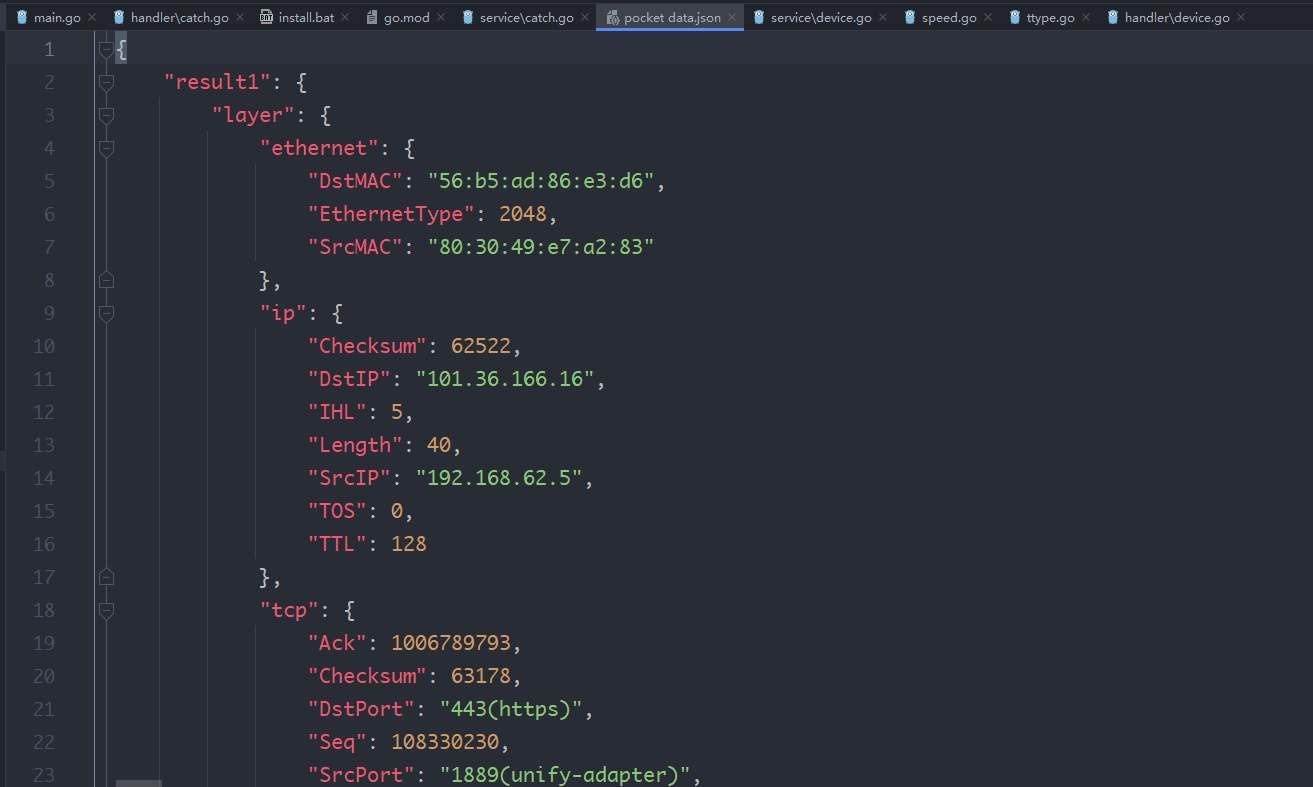
yaml配置



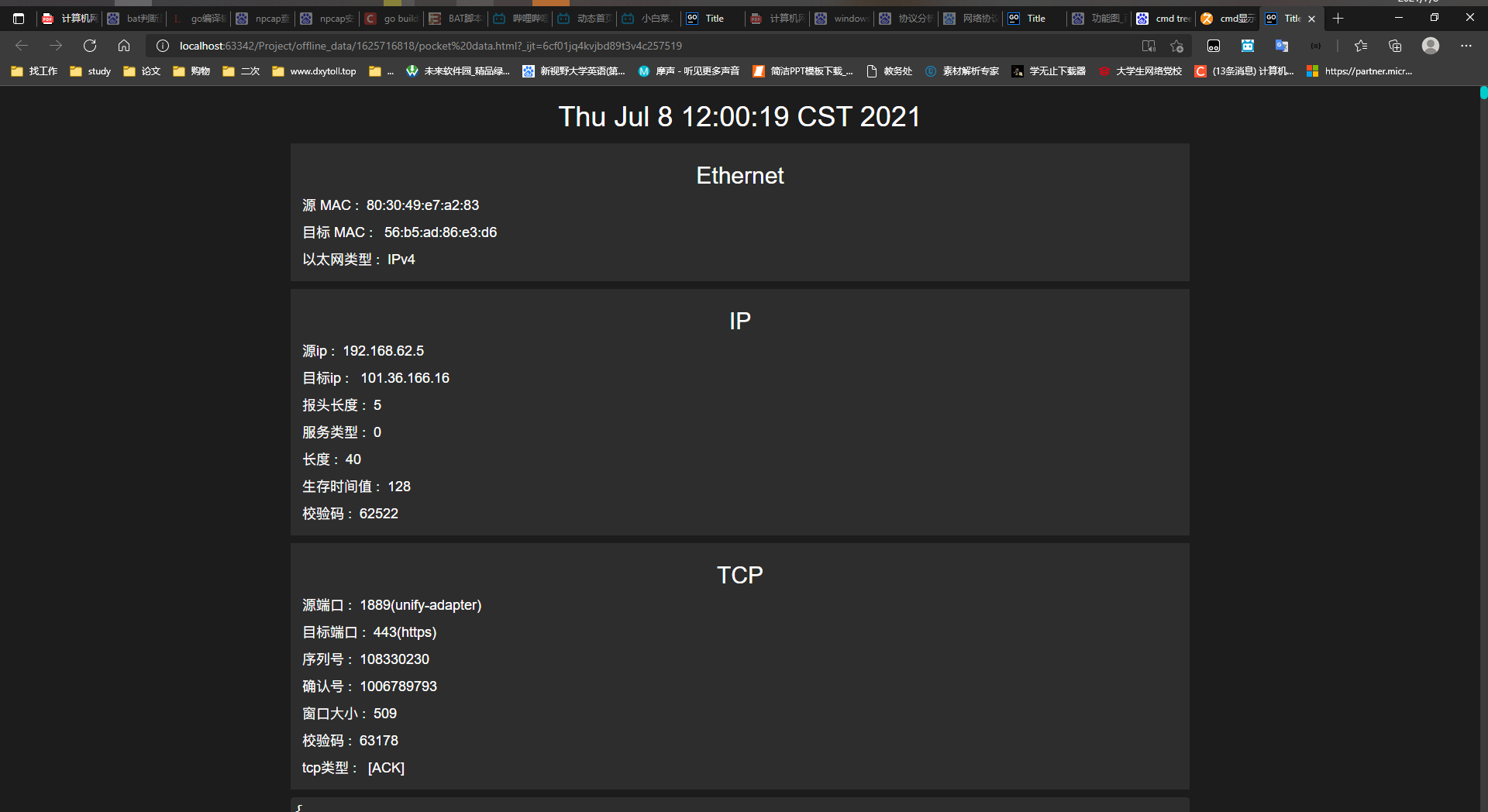
控制台输出：



生成的json文件



生成的html文件

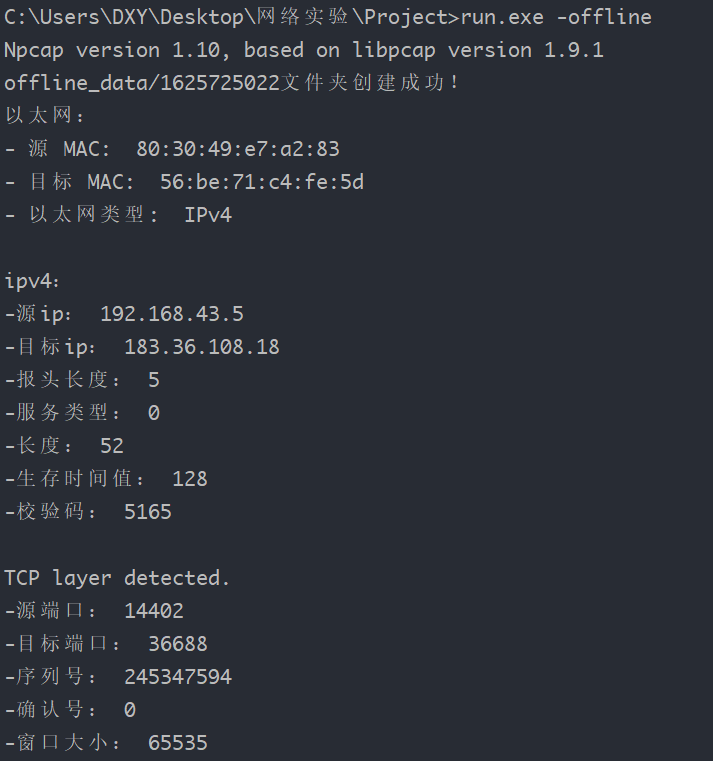


### 5.2.2 pacp文件分析 run.exe -offline

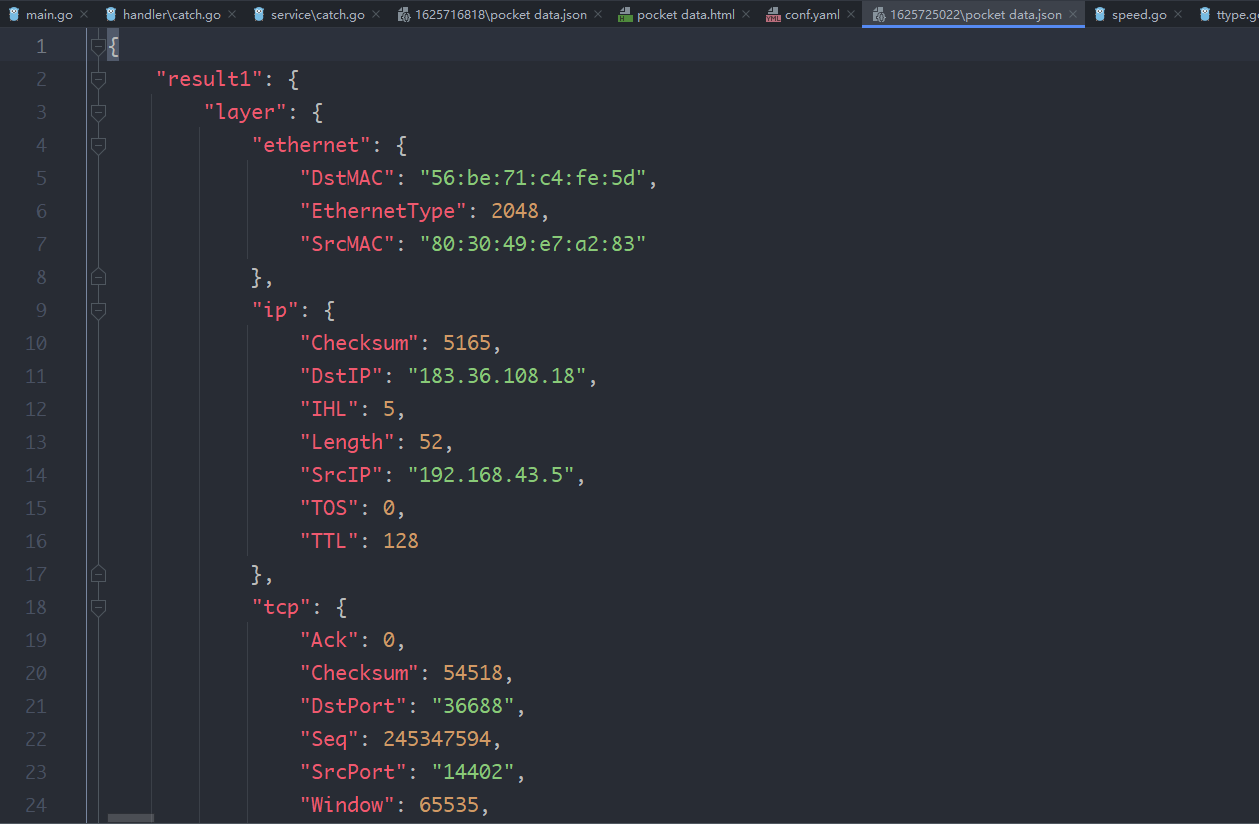
yaml配置



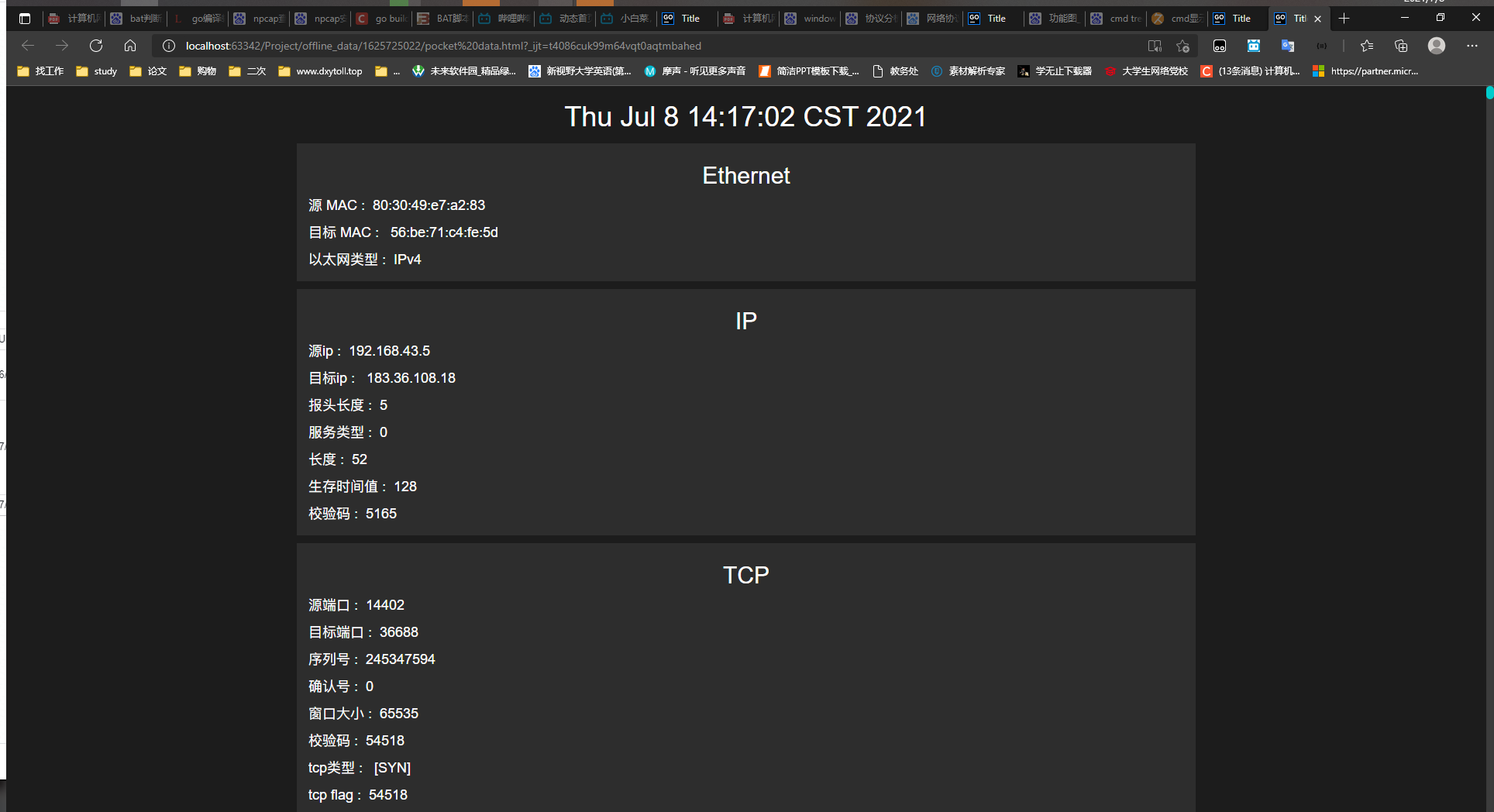
控制台输出：



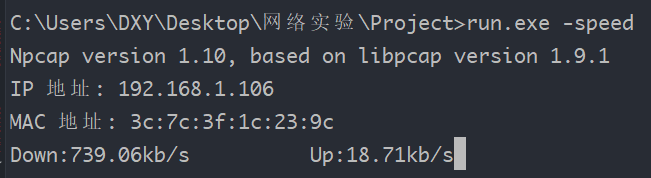
生成的json文件



生成的html文件



### 5.2.3 网速监控 run.exe -speed



# 六、设计中的问题及心得

a. gopacket相关的中文文档基本没有，有的文档只是对godoc做了简单的翻译，为了可以完成实验，最后选择了读英文的godoc，并且翻阅gopacket的源码才完成了本次的课设。

b. 因为go语言没有现成易用的可视化框架，所以最后选择输出为html页面进行可视化的操作。

c. 理论上程序是支持跨平台运行的

# 七、附录

## 7.1 程序清单

### 7.1.1 install.bat

@echo off  
  
rem 判断go环境是否存在  
  
if not exist %GOPATH% echo go env is not exist && goto end  
  
rem 判断资源文件，不影响编译  
if not exist ./conf.yaml echo 'conf.yaml' is not exist  
if not exist ./lib echo 'lib' dir is not exist  
if not exist ./offline\_data mkdir offline\_data && echo offline\_data dir is create  
  
echo Compiling............  
  
rem go拉包  
go mod tidy  
  
rem 开始编译  
if exist ./main.go go build -o run.exe  
  
if exist ./run.exe (goto success) else (goto false)  
  
:success  
 echo Compilation is complete  
 echo Please use the run.exe file in the directory to run  
 goto end  
  
:false  
 echo Compilation Error  
 goto end  
  
:end  
 echo install program exit

### 7.1.2 conf.yaml

# wifi  
#DeviceName: \Device\NPF\_{E01139AE-3D36-44EB-8F8A-712BBB555F29}  
# 以太网  
DeviceName: \Device\NPF\_{B275C137-2F67-4C36-A2FF-21047C0BBB28}  
# 捕获一个数据包的多少个节  
SnapshotLen: 1024  
# 设置网卡是否工作在混杂模式  
Promiscuous: false  
#Filter: "tcp"  
OfflineFilePath: offline\_data/1625621786/pocket data.pcap  
LinkType: 1  
# 抓包数量  
PacketCount: 100  
# 抓包间隔，-1为持续  
TimeoutSecond: -1

### 7.1.3 main.go

package main

import (

. "Project/handler"

. "Project/ttype"

"fmt"

"github.com/google/gopacket/pcap"

"gopkg.in/yaml.v2"

"io/ioutil"

"os"

"strconv"

"time"

)

var

(

confCtx = ConfigCtx{}

err error

)

func main() {

fmt.Println(pcap.Version())

InitConf()

//读取控制台参数

args := os.Args

//处理无参数情况

if len(args) == 1 {

fmt.Print("-h 获取帮助")

} else {

switch args[1] {

case "-h":

fmt.Println("-dn 获取设备信息")

fmt.Println("-catch 实时抓包，保存内容到filePath")

fmt.Println("-offline 从filePath中获取解析离线数据")

fmt.Println("-speed 监控实时网速")

case "-dn":

GetDeviceNameHandel()

case "-catch":

InitFile()

PacketHandler(RealTimeData, &confCtx)

//分析离线数据

case "-offline":

InitFile()

PacketHandler(OfflineData, &confCtx)

case "-speed":

SpeedHandler(&confCtx)

default:

fmt.Print("错误参数, -h获取帮助")

}

}

}

func InitConf() {

//解析配置文件

conf, err := ioutil.ReadFile("conf.yaml")

if err != nil {

fmt.Print(err)

}

yaml.Unmarshal(conf, &confCtx)

confCtx.Timeout = time.Duration(confCtx.TimeoutSecond) \* time.Second

}

func InitFile() {

dir := "offline\_data/" + strconv.FormatInt(time.Now().Unix(), 10)

err = os.Mkdir(dir, os.ModePerm)

if err != nil {

fmt.Println(dir+"文件夹创建失败：", err.Error())

} else {

fmt.Println(dir + "文件夹创建成功！")

}

filePath := fmt.Sprintf("%s/pocket data", dir)

confCtx.RealTimeFilePath = filePath + ".pcap"

confCtx.JsonFilePath = filePath + ".json"

confCtx.HtmlFilePath = filePath + ".html"

}

### 7.1.4 service

#### 7.1.4.1 service/speed.go

package service  
  
import (  
 "Project/ttype"  
 "fmt"  
 "github.com/google/gopacket"  
 "github.com/google/gopacket/layers"  
 "time"  
)  
  
var (  
 downStreamDataSize = 0 // 单位时间内下行的总字节数  
 upStreamDataSize = 0 // 单位时间内上行的总字节数  
)  
  
func SpeedService(packetSource \*gopacket.PacketSource, macAddr string, ctx \*ttype.ConfigCtx) {  
 // 开启子线程，每一秒计算一次该秒内的数据包大小平均值，并将下载、上传总量置零  
 go monitor()  
  
 // 开始抓包  
 for packet := range packetSource.Packets() {  
 // 只获取以太网帧  
 ethernetLayer := packet.Layer(layers.LayerTypeEthernet)  
 if ethernetLayer != nil {  
 ethernet := ethernetLayer.(\*layers.Ethernet)  
 // 如果封包的目的MAC是本机则表示是下行的数据包，否则为上行  
 if ethernet.DstMAC.String() == macAddr {  
 downStreamDataSize += len(packet.Data()) // 统计下行封包总大小  
 } else {  
 upStreamDataSize += len(packet.Data()) // 统计上行封包总大小  
 }  
 }  
 }  
}  
  
// 每一秒计算一次该秒内的数据包大小平均值，并将下载、上传总量置零  
func monitor() {  
 for {  
 fmt.Print(fmt.Sprintf("\rDown:%.2fkb/s \t Up:%.2fkb/s", float32(downStreamDataSize)/1024/1, float32(upStreamDataSize)/1024/1))  
 downStreamDataSize = 0  
 upStreamDataSize = 0  
 time.Sleep(1 \* time.Second)  
 }  
}

#### 7.1.4.2 service/catch.go

package service  
  
import (  
 "Project/ttype"  
 "bytes"  
 "encoding/json"  
 "fmt"  
 sj "github.com/bitly/go-simplejson"  
 "github.com/google/gopacket"  
 "github.com/google/gopacket/layers"  
 "github.com/google/gopacket/pcapgo"  
 "io"  
 "os"  
 "strconv"  
 "time"  
)  
  
var packetCount int32 = 0  
  
// PacketService 解析数据包Handle，可提供写文件操作  
func PacketService(packetSource \*gopacket.PacketSource, w \*pcapgo.Writer, jsonF \*os.File, htmlF \*os.File, ctx \*ttype.ConfigCtx) {  
 //创建json  
 resJson := sj.New()  
 //html写入头  
 io.WriteString(htmlF, ttype.HtmlPre)  
  
 for packet := range packetSource.Packets() {  
 packetJson := sj.New()  
 curTimeStr := time.Now().Format(time.UnixDate)  
  
 if htmlF != nil {  
 //将格式化的HTML写入html文件  
 fmt.Fprintf(htmlF, "<div class=\"row text-center\"><h1>%s</h1></div>\n", curTimeStr)  
 }  
  
 analysisData(packet, htmlF, packetJson)  
 packetJson.Set("time", curTimeStr)  
  
 //有文件指针写文件  
 if w != nil {  
 w.WritePacket(packet.Metadata().CaptureInfo, packet.Data())  
 }  
  
 packetCount++  
 resJson.Set(fmt.Sprintf("result%d", packetCount), packetJson)  
 if packetCount > ctx.PacketCount {  
 break  
 }  
  
 //写html文件  
 packetJsonStr, \_ := packetJson.MarshalJSON()  
 if htmlF != nil {  
 //将格式化的HTML写入html文件  
 fmt.Fprintf(htmlF, "<div class=\"row\"><pre class=\"block\">%s</pre></div>\n", packetJsonStr)  
 }  
 }  
  
 //json字符串  
 jsonStr, \_ := resJson.MarshalJSON()  
  
 //写json文件  
 if jsonF != nil {  
 //格式化的json  
 var formatResJson bytes.Buffer  
 err := json.Indent(&formatResJson, []byte(jsonStr), "", "\t")  
 if err != nil {  
 fmt.Println("json格式化失败")  
 }  
 //将格式化的json写入文件  
 io.WriteString(jsonF, formatResJson.String())  
 }  
  
 //html写入尾  
 io.WriteString(htmlF, ttype.HtmlEnd)  
}  
  
// analysisData 分析数据  
func analysisData(packet gopacket.Packet, htmlF \*os.File, json \*sj.Json) {  
 layersJson := map[string]interface{}{}  
 // 以太网协议  
 ethernetLayer := packet.Layer(layers.LayerTypeEthernet)  
 if ethernetLayer != nil {  
 ethernetLayerJson := map[string]interface{}{}  
 fmt.Println("以太网：")  
 ethernetPacket, \_ := ethernetLayer.(\*layers.Ethernet)  
 fmt.Println("- 源 MAC: ", ethernetPacket.SrcMAC.String())  
 fmt.Println("- 目标 MAC: ", ethernetPacket.DstMAC.String())  
 fmt.Println("- 以太网类型: ", ethernetPacket.EthernetType)  
 fmt.Println()  
  
 ethernetLayerJson["SrcMAC"] = ethernetPacket.SrcMAC.String()  
 ethernetLayerJson["DstMAC"] = ethernetPacket.DstMAC.String()  
 ethernetLayerJson["EthernetType"] = ethernetPacket.EthernetType  
 layersJson["ethernet"] = ethernetLayerJson  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">Ethernet</h2>")  
 fmt.Fprintf(htmlF, "<p>源 MAC&nbsp;:&nbsp;&nbsp;%s<p>\n", ethernetPacket.SrcMAC.String())  
 fmt.Fprintf(htmlF, "<p>目标 MAC&nbsp;:&nbsp;&nbsp; %s<p>\n", ethernetPacket.DstMAC.String())  
 fmt.Fprintf(htmlF, "<p>以太网类型&nbsp;:&nbsp;&nbsp;%s<p>\n", ethernetPacket.EthernetType)  
 fmt.Fprintln(htmlF, "</div></div>")  
 }  
  
 // ip协议  
 ipLayer := packet.Layer(layers.LayerTypeIPv4)  
 if ipLayer != nil {  
 ipLayerJson := map[string]interface{}{}  
 fmt.Println("ipv4：")  
 ip, \_ := ipLayer.(\*layers.IPv4)  
 fmt.Println("-源ip：", ip.SrcIP.String())  
 fmt.Println("-目标ip：", ip.DstIP.String())  
 fmt.Println("-报头长度：", ip.IHL)  
 fmt.Println("-服务类型：", ip.TOS)  
 fmt.Println("-长度：", ip.Length)  
 fmt.Println("-生存时间值：", ip.TTL)  
 fmt.Println("-校验码：", ip.Checksum)  
  
 ipLayerJson["SrcIP"] = ip.SrcIP.String()  
 ipLayerJson["DstIP"] = ip.DstIP.String()  
 ipLayerJson["IHL"] = ip.IHL  
 ipLayerJson["TOS"] = ip.TOS  
 ipLayerJson["Length"] = ip.Length  
 ipLayerJson["TTL"] = ip.TTL  
 ipLayerJson["Checksum"] = ip.Checksum  
 layersJson["ip"] = ipLayerJson  
 fmt.Println()  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">IP</h2>")  
 fmt.Fprintf(htmlF, "<p>源ip&nbsp;:&nbsp;&nbsp;%s<p>\n", ip.SrcIP.String())  
 fmt.Fprintf(htmlF, "<p>目标ip&nbsp;:&nbsp;&nbsp; %s<p>\n", ip.DstIP.String())  
 fmt.Fprintf(htmlF, "<p>报头长度&nbsp;:&nbsp;&nbsp;%d<p>\n", ip.IHL)  
 fmt.Fprintf(htmlF, "<p>服务类型&nbsp;:&nbsp;&nbsp;%d<p>\n", ip.TOS)  
 fmt.Fprintf(htmlF, "<p>长度&nbsp;:&nbsp;&nbsp;%d<p>\n", ip.Length)  
 fmt.Fprintf(htmlF, "<p>生存时间值&nbsp;:&nbsp;&nbsp;%d<p>\n", ip.TTL)  
 fmt.Fprintf(htmlF, "<p>校验码&nbsp;:&nbsp;&nbsp;%d<p>\n", ip.Checksum)  
 fmt.Fprintln(htmlF, "</div></div>")  
 }  
  
 // tcp协议  
 tcpLayer := packet.Layer(layers.LayerTypeTCP)  
 if tcpLayer != nil {  
 tcpLayerJson := map[string]interface{}{}  
 fmt.Println("TCP layer detected.")  
 tcp, \_ := tcpLayer.(\*layers.TCP)  
 fmt.Println("-源端口：", tcp.SrcPort)  
 fmt.Println("-目标端口：", tcp.DstPort)  
 fmt.Println("-序列号：", tcp.Seq)  
 fmt.Println("-确认号：", tcp.Ack)  
 fmt.Println("-窗口大小：", tcp.Window)  
 fmt.Println("-校验码：", tcp.Checksum)  
 fmt.Println("-tcp类型：")  
 var tcpOptionsJsonArr []interface{}  
 for i := 0; i < len(tcp.Options); i++ {  
 \_m := map[string]interface{}{}  
 fmt.Printf("---type：%s, 长度：%d\n", tcp.Options[i].OptionType.String(), tcp.Options[i].OptionLength)  
 \_m["OptionType"] = tcp.Options[i].OptionType.String()  
 \_m["OptionLength"] = tcp.Options[i].OptionLength  
 tcpOptionsJsonArr = append(tcpOptionsJsonArr, \_m)  
 }  
 var tcpFlagJsonArr []string  
 fmt.Println("-tcp flag：")  
 if tcp.FIN {  
 fmt.Println("---FIN")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "FIN")  
 }  
 if tcp.SYN {  
 fmt.Println("---SYN")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "SYN")  
 }  
 if tcp.RST {  
 fmt.Println("---RST")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "RST")  
 }  
 if tcp.PSH {  
 fmt.Println("---PSH")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "PSH")  
 }  
 if tcp.ACK {  
 fmt.Println("---ACK")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "ACK")  
 }  
 if tcp.URG {  
 fmt.Println("---URG")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "URG")  
 }  
 if tcp.ECE {  
 fmt.Println("---ECE")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "ECE")  
 }  
 if tcp.CWR {  
 fmt.Println("---CWR")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "CWR")  
 }  
 if tcp.NS {  
 fmt.Println("---NS")  
 tcpFlagJsonArr = append(tcpFlagJsonArr, "NS")  
 }  
  
 tcpLayerJson["SrcPort"] = tcp.SrcPort.String()  
 tcpLayerJson["DstPort"] = tcp.DstPort.String()  
 tcpLayerJson["Seq"] = tcp.Seq  
 tcpLayerJson["Ack"] = tcp.Ack  
 tcpLayerJson["Window"] = tcp.Window  
 tcpLayerJson["Checksum"] = tcp.Checksum  
 tcpLayerJson["tcp\_flag"] = tcpFlagJsonArr  
 tcpLayerJson["option"] = tcpOptionsJsonArr  
 layersJson["tcp"] = tcpLayerJson  
 fmt.Println()  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">TCP</h2>")  
 fmt.Fprintf(htmlF, "<p>源端口&nbsp;:&nbsp;&nbsp;%s<p>\n", tcp.SrcPort.String())  
 fmt.Fprintf(htmlF, "<p>目标端口&nbsp;:&nbsp;&nbsp;%s<p>\n", tcp.DstPort.String())  
 fmt.Fprintf(htmlF, "<p>序列号&nbsp;:&nbsp;&nbsp;%d<p>\n", tcp.Seq)  
 fmt.Fprintf(htmlF, "<p>确认号&nbsp;:&nbsp;&nbsp;%d<p>\n", tcp.Ack)  
 fmt.Fprintf(htmlF, "<p>窗口大小&nbsp;:&nbsp;&nbsp;%d<p>\n", tcp.Window)  
 fmt.Fprintf(htmlF, "<p>校验码&nbsp;:&nbsp;&nbsp;%d<p>\n", tcp.Checksum)  
 fmt.Fprintln(htmlF, "<p>tcp类型&nbsp;:&nbsp;&nbsp;", tcpFlagJsonArr, "<p>")  
 for idx, opts := range tcpOptionsJsonArr {  
 if idx == 0 {  
 fmt.Fprintf(htmlF, "<p>tcp flag&nbsp;:&nbsp;&nbsp;%d<p>\n", tcp.Checksum)  
 }  
 fmt.Fprintf(htmlF, "<p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;%s&nbsp;:&nbsp;&nbsp;%d<p>\n",  
 opts.(map[string]interface{})["OptionType"],  
 opts.(map[string]interface{})["OptionLength"])  
 }  
 fmt.Fprintln(htmlF, "</div></div>")  
 }  
  
 // udp  
 udpLayer := packet.Layer(layers.LayerTypeUDP)  
 if udpLayer != nil {  
 udpLayerJson := map[string]interface{}{}  
 fmt.Println("udp：")  
 udp, \_ := udpLayer.(\*layers.UDP)  
  
 fmt.Println("-源端口：", udp.SrcPort.String())  
 fmt.Println("-目标端口：", udp.DstPort.String())  
 fmt.Println("-校验码：", udp.Checksum)  
  
 udpLayerJson["SrcPort"] = udp.SrcPort.String()  
 udpLayerJson["DstPort"] = udp.DstPort.String()  
 udpLayerJson["Checksum"] = udp.Checksum  
 layersJson["udp"] = udpLayerJson  
 fmt.Println()  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">UDP</h2>")  
 fmt.Fprintf(htmlF, "<p>源端口&nbsp;:&nbsp;&nbsp;%s<p>\n", udp.SrcPort.String())  
 fmt.Fprintf(htmlF, "<p>目标端口&nbsp;:&nbsp;&nbsp;%s<p>\n", udp.DstPort.String())  
 fmt.Fprintf(htmlF, "<p>校验码&nbsp;:&nbsp;&nbsp;%d<p>\n", udp.Checksum)  
 fmt.Fprintln(htmlF, "</div></div>")  
 }  
  
 // dns  
 dnsLayer := packet.Layer(layers.LayerTypeDNS)  
 if dnsLayer != nil {  
 dnsLayerJson := map[string]interface{}{}  
 fmt.Println("dns：")  
 dns, \_ := dnsLayer.(\*layers.DNS)  
  
 fmt.Println("-应答码：", dns.ResponseCode.String())  
 fmt.Println("-是否期望递归：", dns.RD)  
 fmt.Println("-是否支持递归：", dns.RA)  
 fmt.Println("-截断：", dns.TC)  
 fmt.Println("-问题记录数：", dns.QDCount)  
 fmt.Println("-回答记录数：", dns.ANCount)  
 fmt.Println("-授权记录数：", dns.NSCount)  
 fmt.Println("-附加记录数：", dns.ARCount)  
  
 dnsLayerJson["ResponseCode"] = dns.ResponseCode.String()  
 dnsLayerJson["RD"] = dns.RD  
 dnsLayerJson["RA"] = dns.RA  
 dnsLayerJson["TC"] = dns.TC  
 dnsLayerJson["QDCount"] = dns.QDCount  
 dnsLayerJson["ANCount"] = dns.ANCount  
 dnsLayerJson["NSCount"] = dns.NSCount  
 dnsLayerJson["ARCount"] = dns.ARCount  
 layersJson["dns"] = dnsLayerJson  
 fmt.Println()  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">DNS</h2>")  
 fmt.Fprintf(htmlF, "<p>应答码&nbsp;:&nbsp;&nbsp;%s<p>\n", dns.ResponseCode.String())  
 fmt.Fprintf(htmlF, "<p>是否期望递归&nbsp;:&nbsp;&nbsp; %s<p>\n", strconv.FormatBool(dns.RD))  
 fmt.Fprintf(htmlF, "<p>是否支持递归&nbsp;:&nbsp;&nbsp;%s<p>\n", strconv.FormatBool(dns.TC))  
 fmt.Fprintf(htmlF, "<p>截断&nbsp;:&nbsp;&nbsp;%s<p>\n", strconv.FormatBool(dns.RA))  
 fmt.Fprintf(htmlF, "<p>问题记录数&nbsp;:&nbsp;&nbsp;%d<p>\n", dns.QDCount)  
 fmt.Fprintf(htmlF, "<p>回答记录数&nbsp;:&nbsp;&nbsp; %d<p>\n", dns.ANCount)  
 fmt.Fprintf(htmlF, "<p>授权记录数&nbsp;:&nbsp;&nbsp;%d<p>\n", dns.NSCount)  
 fmt.Fprintf(htmlF, "<p>附加记录数&nbsp;:&nbsp;&nbsp;%d<p>\n", dns.ARCount)  
 fmt.Fprintln(htmlF, "</div></div>")  
 }  
  
 //引用层解析报文  
 applicationLayer := packet.ApplicationLayer()  
 if applicationLayer != nil {  
 fmt.Println("应用层报文：")  
 payload := applicationLayer.Payload()  
  
 fmt.Println(payload)  
  
 //写详细数据到html  
 fmt.Fprintln(htmlF, "<div class=\"row\"><div class=\"block\">")  
 fmt.Fprintln(htmlF, "<h2 class=\"text-center\">Application</h2>")  
 fmt.Fprintln(htmlF, "<p>应用层报文&nbsp;:&nbsp;&nbsp;", payload, "<p>")  
 fmt.Fprintln(htmlF, "</div></div>")  
  
 str := "["  
 for \_, p := range payload {  
 str += strconv.Itoa(int(p)) + " "  
 }  
 str += "]"  
 json.Set("applicationLayerPayLoad", str)  
 }  
  
 // 打印所有层  
 fmt.Println("所有层:")  
 for \_, layer := range packet.Layers() {  
 fmt.Println("- ", layer.LayerType())  
 }  
  
 json.Set("layer", layersJson)  
}

#### 7.1.4.3 service/device.go

package service

import (

"fmt"

"github.com/google/gopacket/pcap"

)

// GetDeviceService 处理数据信息，并打印

func GetDeviceService(devices []pcap.Interface) {

// 打印信息

for \_, device := range devices {

fmt.Println("\n名称: ", device.Name)

fmt.Println("描述: ", device.Description)

for \_, address := range device.Addresses {

fmt.Println("- IP 地址: ", address.IP)

fmt.Println("- 子网掩码: ", address.Netmask)

}

}

}

### 7.1.5 handler

#### 7.1.5.1 handler /speed.go

package handler

import (

"Project/service"

"Project/ttype"

"errors"

"fmt"

"github.com/google/gopacket"

"github.com/google/gopacket/pcap"

"net"

"os"

)

func SpeedHandler(ctx \*ttype.ConfigCtx) {

devices := GetDeviceArrHandel()

// 根据网卡名称从所有网卡中取到精确的网卡

var device pcap.Interface

for \_, d := range devices {

if d.Name == ctx.DeviceName {

device = d

}

}

// 根据网卡的ipv4地址获取网卡的mac地址，用于后面判断数据包的方向

macAddr, err := findMacAddrByIp(findDeviceIpv4(device))

if err != nil {

fmt.Printf("获取mac地址出错 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

fmt.Printf("IP 地址: %s\n", findDeviceIpv4(device))

fmt.Printf("MAC 地址: %s\n", macAddr)

// 获取网卡handler，可用于读取或写入数据包

handle, err = pcap.OpenLive(ctx.DeviceName, ctx.SnapshotLen, ctx.Promiscuous, ctx.Timeout)

if err != nil {

fmt.Printf("开启抓包错误 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

defer handle.Close()

packetSource := gopacket.NewPacketSource(handle, handle.LinkType())

service.SpeedService(packetSource, macAddr, ctx)

}

// 获取网卡的IPv4地址

func findDeviceIpv4(device pcap.Interface) string {

for \_, addr := range device.Addresses {

if ipv4 := addr.IP.To4(); ipv4 != nil {

return ipv4.String()

}

}

panic("硬件没有ipv4")

}

// 根据网卡的IPv4地址获取MAC地址

// 有此方法是因为gopacket内部未封装获取MAC地址的方法，所以这里通过找到IPv4地址相同的网卡来寻找MAC地址

func findMacAddrByIp(ip string) (string, error) {

interfaces, err := net.Interfaces()

if err != nil {

panic(interfaces)

}

for \_, i := range interfaces {

addrs, err := i.Addrs()

if err != nil {

panic(err)

}

for \_, addr := range addrs {

if a, ok := addr.(\*net.IPNet); ok {

if ip == a.IP.String() {

return i.HardwareAddr.String(), nil

}

}

}

}

return "", errors.New(fmt.Sprintf("寻找mac err: %s", ip))

}

#### 7.1.5.2 handler /catch.go

package handler

import (

. "Project/service"

. "Project/ttype"

"fmt"

"github.com/google/gopacket"

"github.com/google/gopacket/pcap"

"github.com/google/gopacket/pcapgo"

"os"

)

var (

err error

handle \*pcap.Handle

)

// PacketHandler 开始抓取

func PacketHandler(dataSource DataSource, ctx \*ConfigCtx) {

if dataSource == OfflineData {

handle, err = pcap.OpenOffline(ctx.OfflineFilePath)

} else if dataSource == RealTimeData {

handle, err = pcap.OpenLive(ctx.DeviceName, ctx.SnapshotLen, ctx.Promiscuous, ctx.Timeout)

}

if err != nil {

fmt.Printf("开启抓包错误 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

//关闭handle

defer handle.Close()

//过滤

if ctx.Filter != "" {

err = handle.SetBPFFilter(ctx.Filter)

if err != nil {

fmt.Printf("过滤设置错误 %s: %v", ctx.DeviceName, err)

os.Exit(1)

}

}

//pcap文件指针

var w \*pcapgo.Writer = nil

//json

var jsonF \*os.File = nil

//html

var htmlF \*os.File = nil

//创建json文件

jsonF, err = os.Create(ctx.JsonFilePath) //创建文件

if err != nil {

fmt.Println("json文件创建失败")

}

//创建html文件

htmlF, err = os.Create(ctx.HtmlFilePath) //创建文件

if err != nil {

fmt.Println("html文件创建失败")

}

// 打开输出pcap文件并写入头文件

if dataSource == RealTimeData {

handle, err = pcap.OpenLive(ctx.DeviceName, ctx.SnapshotLen, ctx.Promiscuous, ctx.Timeout)

f, \_ := os.Create(ctx.RealTimeFilePath)

w = pcapgo.NewWriter(f)

w.WriteFileHeader(uint32(ctx.SnapshotLen), ctx.LinkType)

defer f.Close()

}

// 开始解析包

packetSource := gopacket.NewPacketSource(handle, handle.LinkType())

PacketService(packetSource, w, jsonF, htmlF, ctx)

}

#### 7.1.5.3 handler /device.go

package handler

import (

"Project/service"

"fmt"

"github.com/google/gopacket/pcap"

"os"

)

// GetDeviceNameHandel 获取设备并打印

func GetDeviceNameHandel() {

devices := GetDeviceArrHandel()

service.GetDeviceService(devices)

}

func GetDeviceArrHandel() []pcap.Interface {

// 寻找所有的设备

devices, err := pcap.FindAllDevs()

if err != nil {

fmt.Println("寻找设备出错")

os.Exit(1)

}

return devices

}

### 7.1.6 ttype/ttype.go

package ttype

import (

"context"

"github.com/google/gopacket/layers"

"time"

)

// DataSource 数据源

type DataSource uint8

const (

OfflineData DataSource = 1 //离线数据

RealTimeData DataSource = 2 //实时数据

)

// ConfigCtx 配置

type ConfigCtx struct {

context.Context

DeviceName string `yaml:"DeviceName"`

SnapshotLen int32 `yaml:"SnapshotLen"` //捕获一个数据包的多少个字节

Promiscuous bool `yaml:"Promiscuous"` //设置网卡是否工作在混杂模式

Filter string `yaml:"Filter"`

OfflineFilePath string `yaml:"OfflineFilePath"`

LinkType layers.LinkType `yaml:"LinkType"`

PacketCount int32 `yaml:"PacketCount"`

TimeoutSecond int64 `yaml:"TimeoutSecond"` //设置抓到包返回的超时

Timeout time.Duration

RealTimeFilePath string

JsonFilePath string

HtmlFilePath string

DataSource DataSource

}

const HtmlPre = `<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<script type="text/javascript" src="../../lib/jquery-1.11.0.min.js"></script>

<link href="../../lib/bootstrap.min.css" rel="stylesheet">

<script type="text/javascript" src="../../lib/bootstrap.min.js"></script>

<style>

body {

background: #1c1c1c;

}

.block {

font-size: 18px;

color: white;

padding: 5px;

margin: 5px;

background: #2d2d2d;

border: none;

}

.key {

color: rgb(255,121,170);

font-weight: bold;

}

.null {

color: #f1592a;

font-weight: bold;

}

.string {

color: rgb(241,250,118);

font-weight: bold;

}

.number {

color: rgb(156,147,249);

font-weight: bold;

}

.boolean {

color: rgb(156,147,249);

font-weight: bold;

}

h1 {

color: white;

}

body::-webkit-scrollbar-track {

-webkit-box-shadow: inset 0 0 6px rgba(0, 0, 0, 0.1);

background-color: #3d3d3d;

border-radius: 10px;

}

body::-webkit-scrollbar {

width: 10px;

background-color: #3d3d3d;

}

body::-webkit-scrollbar-thumb {

border-radius: 10px;

background-color: rgb(0, 204, 204);

}

.block::-webkit-scrollbar-track {

-webkit-box-shadow: inset 0 0 6px rgba(0, 0, 0, 0.1);

background-color: #3d3d3d;

border-radius: 10px;

}

.block::-webkit-scrollbar {

width: 10px;

background-color: #3d3d3d;

}

.block::-webkit-scrollbar-thumb {

border-radius: 10px;

background-color: rgb(0, 204, 204);

}

p {

margin-left: 10px;

}

</style>

</head>

<body>

<div class="container">`

const HtmlEnd = `</div>

<script>

function JsonFormat() {

let json = $(this).text()

json = JSON.stringify(JSON.parse(json), undefined, 2);

json = json.replace(/&/g, '&').replace(/</g, '<').replace(/>/g, '>');

let res = json.replace(/("(\\u[a-zA-Z0-9]{4}|\\[^u]|[^\\"])\*"(\s\*:)?|\b(true|false|null)\b|-?\d+(?:\.\d\*)?(?:[eE][+\-]?\d+)?)/g, function (match) {

var cls = 'number';

if (/^"/.test(match)) {

if (/:$/.test(match)) {

cls = 'key';

} else {

cls = 'string';

}

} else if (/true|false/.test(match)) {

cls = 'boolean';

} else if (/null/.test(match)) {

cls = 'null';

}

return '<span class="' + cls + '">' + match + '</span>';

});

$(this).html(res)

}

$('pre').each(JsonFormat);

</script>

</body>

</html>`

### 7.1.6 go.mod

module Project

go 1.15

require (

github.com/bitly/go-simplejson v0.5.0

github.com/bmizerany/assert v0.0.0-20160611221934-b7ed37b82869 // indirect

github.com/google/gopacket v1.1.19

github.com/kr/pretty v0.2.1 // indirect

gopkg.in/yaml.v2 v2.4.0

)