上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

《计算机网络》课程设计

简易网络嗅探器实现

学年学期:	2025-2026 学年第一学期
姓　　名:	马悦钊
专　　业:	信息安全
学　　院:	计算机学院
指导老师:	许可
完成时间:	2025 年 11 月 28 日

摘　要

本文介绍了一款基于 Python 开发的网络嗅探器工具，可实时时捕获和分析网络数据包，支持 TCP、UDP、ICMP 等多协议解析，具备数据包过滤、混杂模式及 IP 分片重组功能。工具采用 Tkinter 构建图形用户界面，实现了网卡选择、捕获控制、数据筛选及详情详情展示等功能，通过 Scapy 库处理数据包，提供 PCAP 文件的保存与加载功能。测试表明，该工具能有效完成网络监控与故障排查，适用于网络管理及协议学习场景。

关键词：网络嗅探器；Python；Sniffer；Scapy；数据包捕获；协议解析；图形图形界面

目 录

第 1 章　网络嗅探简介

1.1　概念原理

网络嗅探（Network Sniffing）是一种监视网络流量的技术，通过捕获网络中的数据包并进行分析，以了解网络通信情况。它可以帮助网络管理员诊断网络问题、检测网络攻击和优化网络性能。

网络嗅探器工作在 OSI 模型的数据链路层，通过将网络接口设置为混杂模式（Promiscuous Mode），可以捕获经过该接口的所有数据包，而不仅仅是发送给自身的数据包。捕获到的数据包会被按照不同的协议进行解析，提取出源地址、目的地址、协议类型、数据内容等信息。

1.2　应用场景

- 网络故障排查：通过分析数据包，定位网络连接问题、协议错误等
- 网络安全监控：检测异常流量、网络攻击（如 ARP 欺骗、端口扫描等）
- 网络性能分析：统计网络流量、分析带宽使用情况，优化网络资源分配
- 协议分析与学习：深入了解各种网络协议的工作原理和数据格式

1.3　注意事项

- 权限要求：运行网络嗅探器需要管理员/root 权限
- 网络安全：在使用混杂模式时，请确保符合相关法律法规，仅用于合法的网络分析
- 性能影响：长时间捕获大量数据包可能会影响系统性能
- 兼容性：在不同操作系统上可能需要调整部分配置

第 2 章　项目基本信息

2.1　项目简介

本项目是一个基于 Python Scapy 库开发的网络嗅探器工具，能够实时捕获和分析网络数据包，支持多种协议解析、数据包过滤、混杂模式和分片重组等功能。该工具提供了直观的图形界面，便于网络分析和故障排查。项目已发布在 Github-NIS3364 Network Sniffer.

2.2　开发环境

- Python 3.10 或更高版本
- 依赖库：
  - scapy >= 2.4.5
  - psutil >= 5.9.0
  - tkinter (Python 标准库)

2.3　项目结构

```
sniffer/
├── src/                    # 源代码目录
│   ├── main.py             # 程序入口文件
│   ├── sniffer.py          # 核心嗅探功能实现
│   └── process_packet.py   # 数据包处理模块
├── test/                   # 测试脚本目录
│   ├── test.pcap           # 测试记录
│   ├── test.py             # 功能测试脚本
│   └── hex2png.py          # 十六进制数据转换为PNG图片脚本
├── ico/                    # 图标资源目录
│   └── Sniffer.ico         # 应用程序图标
├── requirements.txt        # 项目依赖文件
└── README.md               # 项目说明文档
```

图 2.1　项目结构

第 3 章　项目功能展示

3.1　主要功能列表

1. **实时数据包捕获**：支持在不同网卡上实时捕获网络数据包
2. **多协议解析**：支持 TCP、UDP、ICMP、IPv4、IPv6、ARP 等常见网络协议的解析
3. **数据包过滤**：支持源/目标 IP、源/目标端口等条件过滤数据包
4. **混杂模式支持**：可开启混杂模式捕获网络中所有数据包
5. **分片重组**：重组 IP 分片数据包，还原完整数据内容
6. **PCAP 包导出与分析**：支持 PCAP 格式数据包的导出及导入分析

3.2　运行展示

3.2.1　选择网卡接口

在顶部功能区中，可以从下拉列表中选择要监听的网络接口。

图 3.1　选择网络接口

3.2.2　步骤 2：设置过滤条件

在顶部功能区域选中协议类型进行过滤。

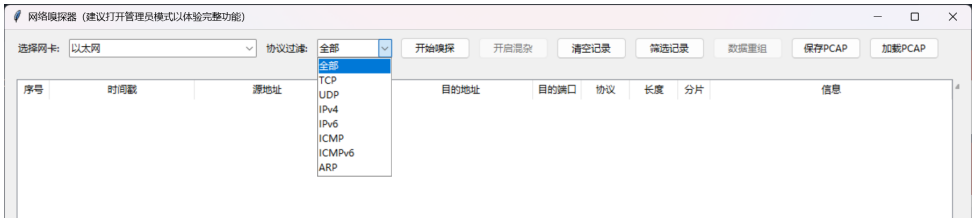


图 3.2　设置过滤条件

3.2.3　配置捕获选项

在以管理员模式运行时，可以点击按钮开启/关闭混杂模式（非管理员模式下该按钮禁用）。

图 3.3　配置捕获选项

3.2.4　控制捕获过程

点击控制按钮开始/暂停捕获。也可以随时点击清空按钮清除当前捕获的数据包。

3.2.5　筛选数据包

停止捕获后，可以根据 IP 地址、端口号等条件对捕获到的数据包进行筛选。

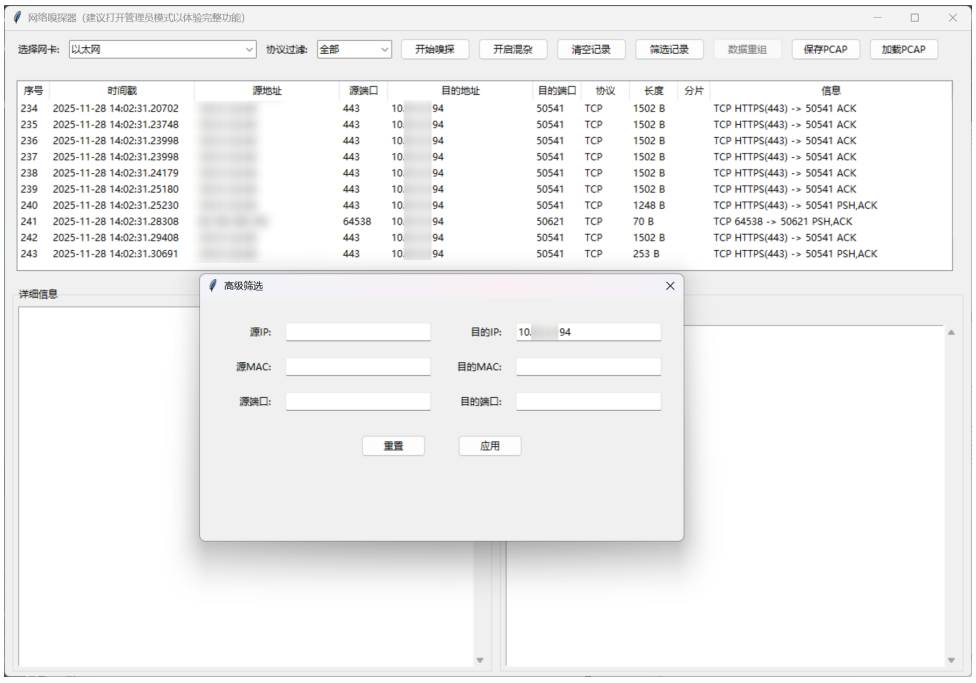


图 3.4　筛选数据包

3.2.6　查看数据包详情

在列表中选择数据包，在下方查看详细信息：左侧部分是对数据包的各层协议分析，依赖 scapy 对数据包各字段的提取实现；右侧为提取到的数据部分，可以选择以十六进制/ASCII/UTF-8 编码展示。

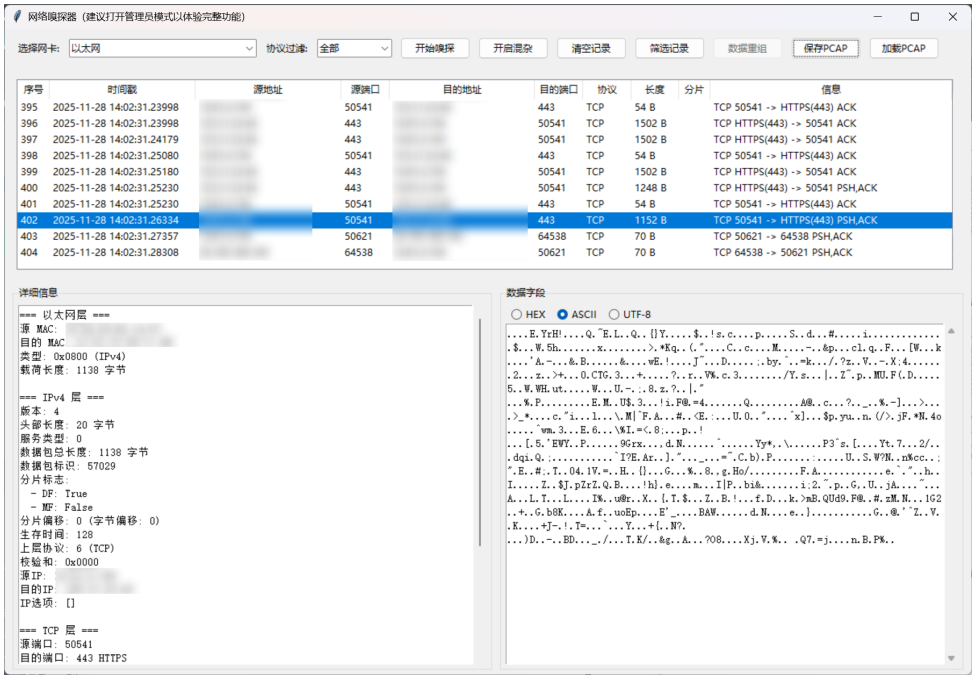


图 3.5　查看数据包详情

对于分片数据包，可以点击顶部功能区的重组按钮，查看完整数据内容。

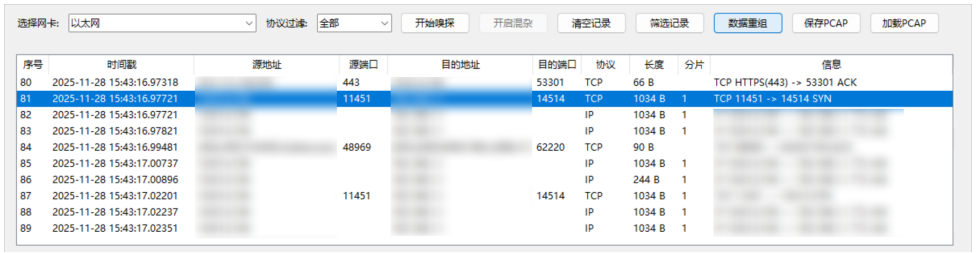


图 3.6　重组数据包

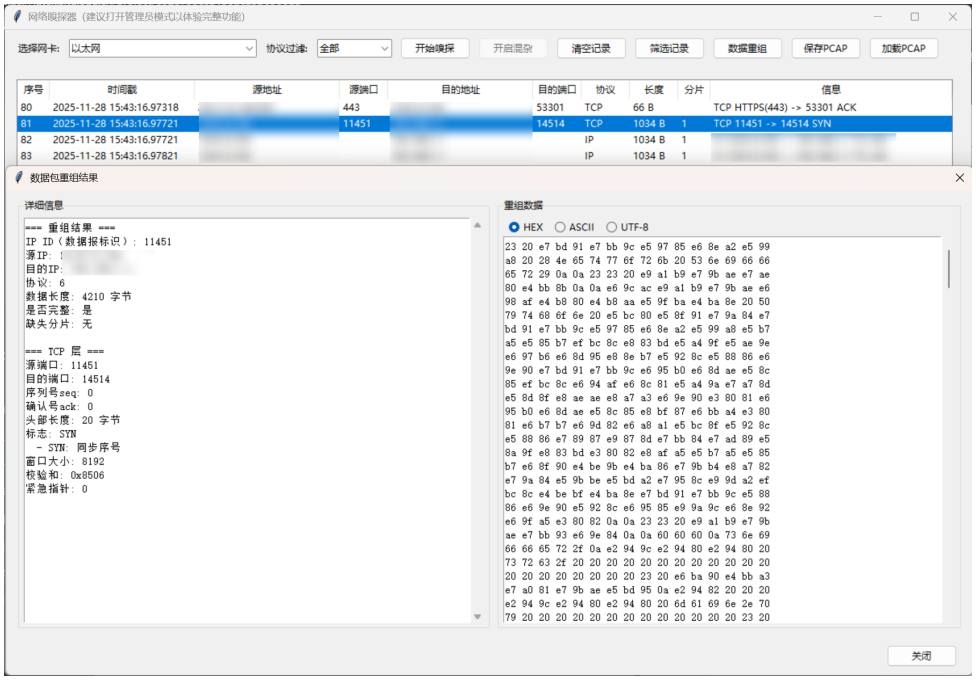


图 3.7　重组数据包展示

3.2.7　导出为 PCAP 文件及从 PCAP 文件导入

点击顶部功能区的“保存 PCAP”或“加载 PCAP”按钮，可以调用系统窗口将捕获的数据包保存为 PCAP 格式，或从 PCAP 格式文件中加载数据包并进行分析。

3.3　功能测试

项目提供了测试脚本 `test/test.py`，用于验证嗅探器的功能：

1. **分片发送长报文**：测试嗅探器的分片重组功能
2. **发送图像文件**：测试文件内容传输的捕获
3. **多协议测试**：测试 TCP、UDP、ICMP、ARP 等协议的解析

在启动嗅探器后，运行测试脚本，可以使用筛选功能辅助观察：由于测试脚本使用以太网卡向 IP 192.168.1.1 发送测试数据，因此设置筛选选项：dst_ip=192.168.1.1。

3.3.1　分片重组测试

首先测试分片数据包重组功能：在主界面中选中“分组”列标识为“1”的数据包，点击右上角重组按钮，打开重组窗口：

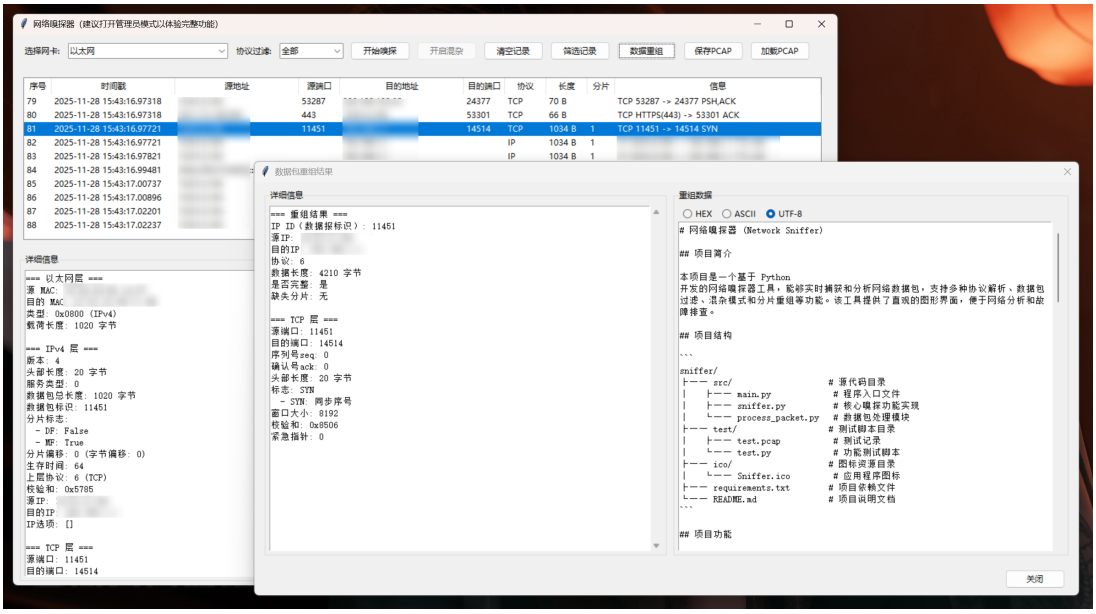


图 3.8　分片重组测试

可以观察到数据包成功重组，左侧显示重组结果，包含 IP ID，源 IP、目标 IP、数据总长度、是否完整及缺失分片等字段；同时显示其上层协议的相关信息。右侧是重组数据的展示，使用 UTF-8 解码后，我们成功得到测试脚本中发送的 `README.md` 的内容。

3.3.2　文件重组测试

测试脚本中还发送了一个 PNG 文件，这里将截获到的数据包重组后，使用 ASCII 解码，发现数据包头包含“PNG”字段。再使用 `test/HEX2PNG.py` 脚本将重组后的十六进制数据转换为 PNG 格式，我们成功得到测试脚本中发送的 `ico/Sniffer.png` 的图片，这也是软件打包为二进制文件时使用的 ico。

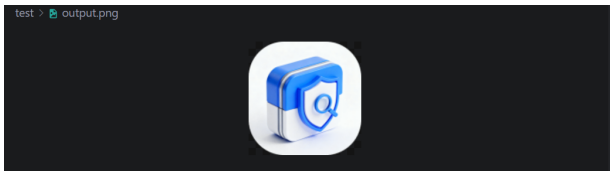


图 3.9　重组后的图标

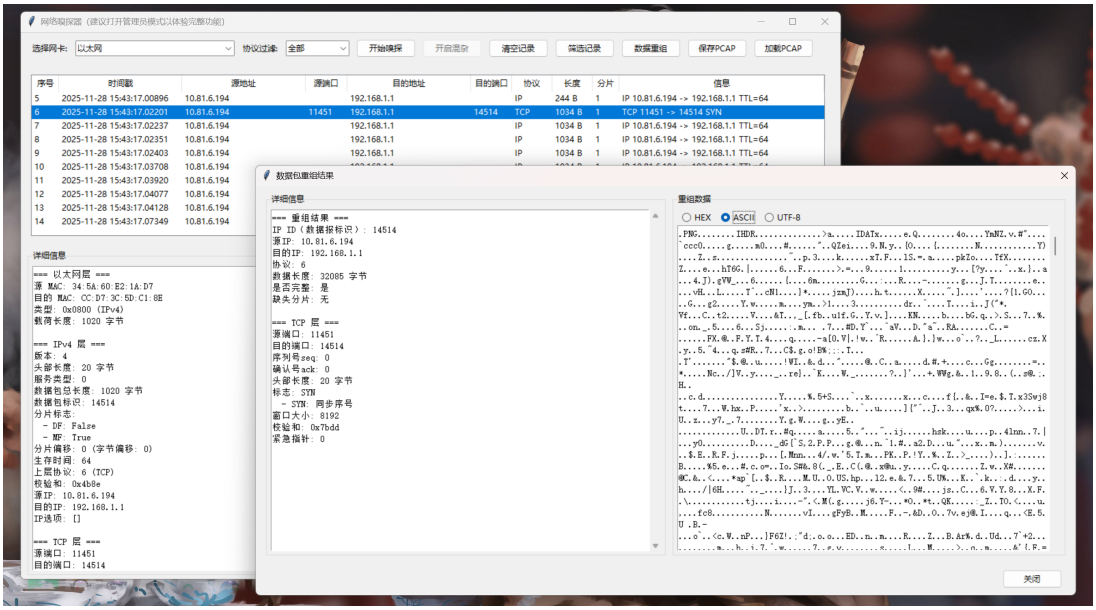


图 3.10　文件重组测试

3.3.3　多协议测试

最后进行多协议解析测试。测试脚本中分别以 ICMP、UDP、TCP、ARP 发送测试请求，可以观察到网络嗅探器均能正确解析并数据显示。

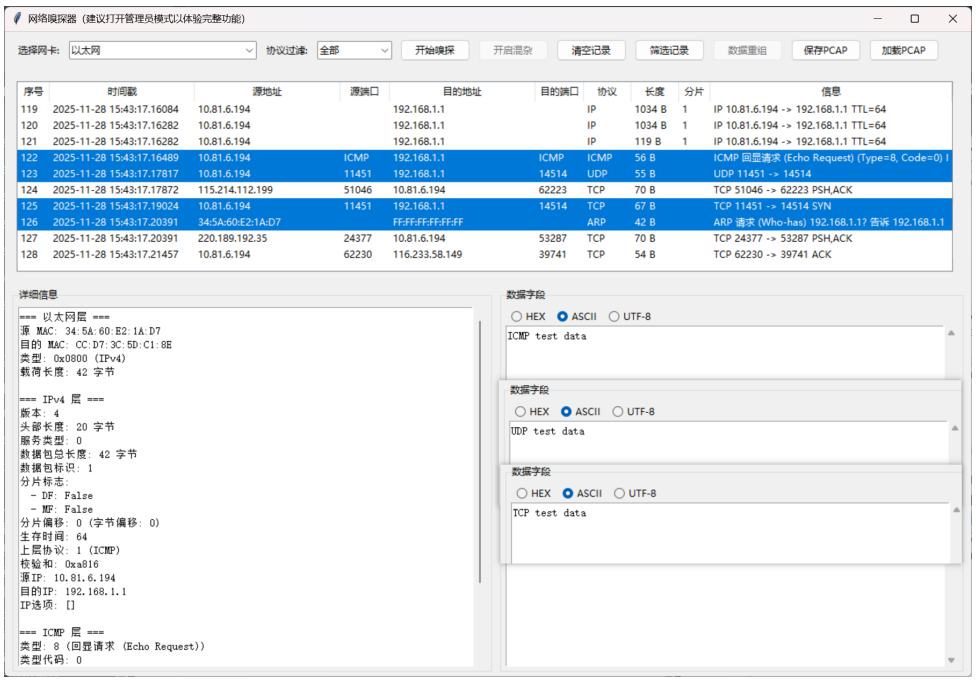


图 3.11　多协议测试

第 4 章　项目实现

4.1　整体架构设计

本项目采用模块化设计，主要分为以下几个模块：

1. **主程序模块**（main.py）：程序入口，负责初始化界面和检查权限
2. **嗅探器核心模块**（sniffer.py）：实现数据包捕获、界面交互等核心功能
3. **数据包处理模块**（process_packet.py）：负责解析不同协议的数据包

4.2　核心功能实现

4.2.1　主程序模块（main.py）

主程序模块负责初始化应用程序，检查运行权限，并创建主窗口。

```
import os
import tkinter as tk
from sniffer import NetworkSniffer

def main():
    """主函数"""
    # 检查是否以管理员/root权限运行
    if os.name == "nt":  # Windows
        try:
            import ctypes
            is_admin = ctypes.windll.shell32.IsUserAnAdmin() != 0
        except:
            pass
    else:  # Unix/Linux
        is_admin = (os.geteuid() == 0)

    root = tk.Tk()
    app = NetworkSniffer(root, is_admin)

    # 设置程序退出处理
    def on_closing():
```

```
22            if app.is_sniffing:
23                app._pause_sniffing()
24            root.destroy()
25
26        root.protocol("WM_DELETE_WINDOW", on_closing)
27        root.mainloop()
28
29    if __name__ == "__main__":
30        main()
```

4.2.2　嗅探器核心模块（sniffer.py）

该模块实现了网络嗅探器的核心功能，包括基于 Tkinter 的界面创建、基于 psutil 的网络接口读取、基于 scapy 的数据包捕获、PCAP 文件处理以及状态控制等。

4.2.2.1　初始化与界面创建

```
1   class NetworkSniffer:
2       """网络嗅探器主类"""
3
4       def __init__(self, root: tk.Tk, is_admin):
5           """初始化嗅探器"""
6           self.root = root
7           if is_admin:
8               self.root.title("网络嗅探器（管理员模式）")
9           else:
10              self.root.title("网络嗅探器（建议打开管理员模式以体验完整功
        能）")
11          self.root.geometry("1200x800")
12          self.root.minsize(1200, 800)
13          self.fonts = ("SimHei", 10)
14
15          # 程序状态变量
16          self.is_sniffing = False
17          self.sniff_thread = None
18          self.packet_list: List[Dict[str, Any]] = []
19          self.original_packet_list: List[Dict[str, Any]] = []
20          self.selected_packet_index = -1
```

```
21          self.protocol_filters = {}
22          self.current_interface = None
23          self.packets_to_save = []
24          self.is_loading_from_pcap = False
25          self.is_filtered = False
26          self.is_promiscuous = False
27          self.is_admin = is_admin
28
29          # 初始化界面
30          self._create_widgets()
31          self._layout_widgets()
32          self._setup_events()
33
34          # 加载网络接口
35          self.load_network_interfaces()
```

4.2.2.2　数据包捕获功能

```
1   def _start_sniffing(self):
2       """开始嗅探数据包"""
3       # 省略相关处理过程
4       self.sniff_thread = threading.Thread(target=self.
       _sniff_packets, args=(interface, protocol_filter), daemon=True)
5       self.sniff_thread.start()
6   def _sniff_packets(self, interface: str, protocol_filter: str):
7       """数据包嗅探函数"""
8       def packet_callback(packet):
9           if not self.is_sniffing:
10              return False
11
12          self._process_packet(packet)
13
14      # 构建过滤器表达式
15      filter_expr = ""
16      if protocol_filter != "全部":
17          if protocol_filter == "IPv4":
18              filter_expr = "ip"
19          elif protocol_filter == "IPv6":
```

```
20              filter_expr = "ip6"
21          else:
22              filter_expr = protocol_filter.lower()
23
24      try:
25          scapy.sniff(iface=interface, prn=packet_callback, store=
    False, stop_filter=lambda _: not self.is_sniffing, filter=
    filter_expr, promisc=self.is_promiscuous)  # 根据标志启用/禁用混杂
    模式
26      except Exception as e:
27          self.root.after(0, lambda: messagebox.showerror("错误", f"
    嗅探失败: {str(e)}"))
28          self.root.after(0, self._pause_sniffing)
```

4.2.2.3　PCAP 文件处理

```
1   def save_to_pcap(self):
2       """保存为PCAP文件"""
3       try:
4           # 打开文件对话框
5           file_path = filedialog.asksaveasfilename(
6               defaultextension=".pcap",
7               filetypes=[("PCAP Files", "*.pcap"), ("All Files", "*"
    )]
8           )
9
10          if file_path:
11              # 保存数据包
12              wrpcap(file_path, self.packets_to_save)
13              messagebox.showinfo("成功", f"数据包已保存到 {file_path}"
    )
14      except Exception as e:
15          messagebox.showerror("错误", f"保存失败: {str(e)}")
16
17  def load_from_pcap(self):
18      """从PCAP文件加载数据包（使用线程避免GUI卡死）"""
19      try:
20          # 打开文件对话框
```

```
21          file_path = filedialog.askopenfilename(
22              filetypes=[("PCAP Files", "*.pcap"), ("All Files", "*"
    )]
23          )
24
25          if file_path:
26              # 启动加载线程
27              load_thread = threading.Thread(target=
    _load_packets_thread, daemon=True)
28              load_thread.start()
29
30      except Exception as e:
31          messagebox.showerror("错误", f"加载失败: {str(e)}")
32          # 重置加载标志
33          self.is_loading_from_pcap = False
```

4.2.3　数据包处理模块（process_packet.py）

该模块负责解析不同协议的数据包，基于 python scapy 库提供的接口提取各种关键信息，处理后添加到 `packet_info` 字典中，返回给 `NetworkSniffer` 类实例进行处理并展示。

4.2.3.1　IP 协议处理

```
1  def process_ip(packet: scapy.Packet, packet_info: Dict[str, Any]):
2      """处理IP层"""
3      if IP not in packet:
4          return
5
6      ip_packet = packet[IP]
7      # 获取协议类型名称
8      IP_proto_names = {
9          1: "ICMP",
10         2: "IGMP",
11         6: "TCP",
12         17: "UDP",
13         # 其他协议...
14     }
```

```
15      IP_proto_name = IP_proto_names.get(ip_packet.proto, f"Protocol
        {ip_packet.proto}")
16
17      # 检查是否是分片数据包
18      is_fragment = (not ip_packet.flags.DF) and (ip_packet.flags.MF
        or ip_packet.frag > 0)
19      if is_fragment:
20          packet_info["is_fragment"] = "1"
21
22      packet_info["protocol"] = "IP"
23      packet_info["info"] = f"IP {ip_packet.src} -> {ip_packet.dst}
       TTL={ip_packet.ttl}"
24
25      packet_info["src"] = ip_packet.src
26      packet_info["dst"] = ip_packet.dst
27
28      # 详细信息
29      packet_info["detail"] += f"=== IPv4 层 ===\n"
30      packet_info["detail"] += f"版本: {ip_packet.version}\n"
31      packet_info["detail"] += f"头部长度: {ip_packet.ihl * 4} 字节\n"
32      packet_info["detail"] += f"服务类型: {ip_packet.tos}\n"
33      packet_info["detail"] += f"数据包总长度: {ip_packet.len} 字节\n"
34      packet_info["detail"] += f"数据包标识: {ip_packet.id}\n"
35      # 其他详细信息提取
36
37      # 尝试提取应用层数据
38      if packet.haslayer(scapy.Raw):
39          packet_info["data"] = packet[scapy.Raw].load
```

4.2.3.2　其他协议处理

类似地，模块中还实现了对 TCP、UDP、ICMP、ARP 等协议的解析处理函数，这里不再一一展示。

4.2.3.3　分片重组处理

IP 数据包分片的重组处理主要依赖于两个函数，`reassemble_packet` 实现重组功能，并返回给 `NetworkSniffer` 类的成员函数进行处理并展示。

```
def reassemble_packet(Sniffer):
    """重组分片数据包"""

    # 获取当前选中的IP数据包信息，收集具有相同IP ID、源IP、目的IP和协议的所有分片及其对应的packet_info加入fragments_with_info，并根据偏移量对其进行排序

    # 尝试重组数据包
    try:
        reassembled_data = b""

        for _, fragment, _, packet_info in fragments_with_info:
            # 优先使用packet_info中已解析的数据
            if packet_info["data"]:
                fragment_data = packet_info["data"]
            else:
                # 如果没有预解析数据，直接从IP包中提取载荷
                ip_header_length = fragment.ihl * 4
                fragment_data = bytes(fragment)[ip_header_length:]
            # 将当前分片数据添加到重组数据中
            reassembled_data += fragment_data

        # 检查分片完整性、从偏移量0开始的连续性、中间分片的连续性、是否有最后一个分片（没有MF标志）

        reassembled_detail = "=== 重组结果 ===\n"
        reassembled_detail += f"IP ID（数据报标识）: {ip_id}\n"
        reassembled_detail += f"源IP: {src_ip}\n"
        reassembled_detail += f"目的IP: {dst_ip}\n"
        reassembled_detail += f"协议: {proto}\n"
        reassembled_detail += f"数据长度: {max_offset} 字节\n"
        reassembled_detail += f"是否完整: {'是' if is_complete else '否'}\n"
        reassembled_detail += f"缺失分片: {', '.join(missing_fragments) if not is_complete else '无'}\n\n"
        # 其余详细信息
        return reassembled_detail, reassembled_data

```

```
34          except Exception as e:
35              messagebox.showerror("重组错误", f"重组过程中发生错误: {str(e)}
    ")
36              return None, None
```

第 5 章　人工智能协作

在本次编程实践中，我借助了 TRAE IDE 3.0.1 的帮助，使用大模型 Agent 与对话流协助完成工作，切身体会到了 Vibe Coding 的魅力。

5.1　模型列表

在项目开发流程中，我使用 TRAE IDE 的 Agent 并令其自动选择模型。其内置的模型如下：

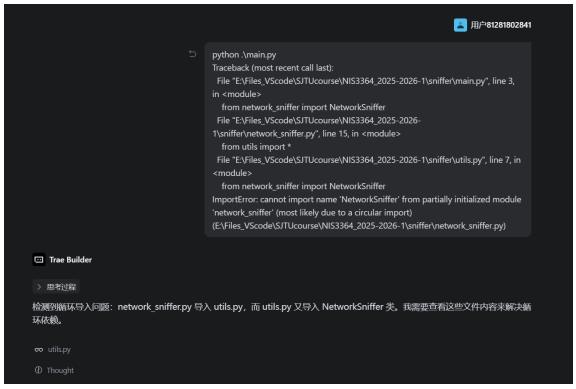
- Doubao-Seed-Code
- Kimi-K2-0905
- GLM-4.6
- MiniMax-M2
- DeepSeek-V3.1-Terminus
- Qwen-3-Coder

5.2　模型交互

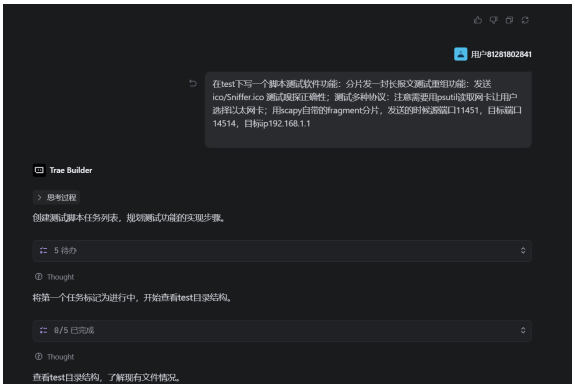
首先，我使用以下 Prompt 令大模型为我生成一个基本的项目框架：

图 5.1　基本任务描述

接着，我阅读并修改代码，完善详细功能，在运行时遇到报错或需要对 UI 界面进行修改时询问大模型。最后借助大模型的帮助为整个项目生成一份 README 文档。



(a) 循环导入 DEBUG



(b) 生成调试脚本

(c) 重组功能 DEBUG

(d) 顶部 UI 修改

(e) 重组 UI 修改

(f) 生成 README

图 5.2　各功能模块调试与修改结果

第 6 章　总结

6.1　项目成果

本项目成功实现了一个功能完善的网络嗅探器，具备以下特点：

1. 支持多种网络协议的解析，包括 TCP、UDP、ICMP、ICMPv6、IPv4、IPv6、ARP 等
2. 提供直观友好的图形界面，方便用户操作和数据分析
3. 实现了数据包过滤、分片重组、混杂模式等高级功能
4. 具备良好的可扩展性，便于后续添加新的协议解析和功能扩展

6.2　项目不足与改进方向

由于时间有限，本项目还存在许多不足之处，如：对于一些不常见的网络协议支持不够完善；在处理大量数据包时，即便使用多线程并对数据包进行分批处理，界面可能出现卡顿现象；数据包分析功能相对基础，缺乏深度分析能力等等。

如若后续有更多时间，我会考虑从以下方面对项目进行进一步的优化与改进，实现一个更先进、更完善的网络嗅探器。

1. 增加更多协议的解析支持，如 HTTP、FTP、DNS 等应用层协议
2. 优化数据包处理和界面刷新机制，提高程序性能
3. 增加更强大的数据分析功能，如流量统计图表、异常检测等
4. 完善跨平台兼容性，确保在不同操作系统上都能稳定运行

6.3　经验与体会

通过本项目的开发，深入理解了网络协议的工作原理和数据包结构，掌握了网络嗅探的核心技术。同时，在项目开发过程中，也锻炼了问题分析和解决能力，学会了如何将复杂的功能分解为模块化的组件，提高代码的可维护性和可扩展性。

这次编程实践也让我切身体会到人工智能的发展迅速与实用价值。借助 TRAE IDE 中的多模型 Agent 协作，从项目框架搭建、代码调试到 UI 界面迭代与文档生成，AI 工具有效降低了开发门槛、提升了问题解决效率，让我能够更聚焦于核心功能的逻辑实现与性能优化。这种人机协同的开发模式不仅是技术实践的助力，更启发了我对未来软件

开发流程的思考——合理运用 AI 工具赋能编程工作，既能弥补个体经验的局限，也能为创新实践注入更多可能性，这将成为我后续学习与开发中持续探索的方向。