

网络综合实践

课程设计报告

学号： 40222308

姓名： 郭焕

班级： 22软工1班

成绩：

电子与计算机工程学院

School of Electronic & Computer Engineering

2025年10月

**目录**

[第一章 引言 1](#_Toc210850536)

[1.1 选题背景 1](#_Toc210850537)

[1.2 主要研究任务 1](#_Toc210850538)

[1.3 报告结构 1](#_Toc210850539)

[第二章 开发环境及工具介绍 3](#_Toc210850540)

[2.1 开发环境 3](#_Toc210850541)

[2.2 开发工具与技术栈 3](#_Toc210850542)

[第三章 需求分析 4](#_Toc210850543)

[3.1 功能需求 4](#_Toc210850544)

[3.2 非功能需求 5](#_Toc210850545)

[3.3 可行性分析 5](#_Toc210850546)

[第四章 系统设计 7](#_Toc210850547)

[4.1 系统概要设计 7](#_Toc210850548)

[4.1.1 架构图 7](#_Toc210850549)

[4.2 系统详细设计 7](#_Toc210850550)

[4.2.1 捕获模块设计 7](#_Toc210850551)

[4.2.2 数据包处理与存储模块 8](#_Toc210850552)

[4.2.3 报告生成模块设计 9](#_Toc210850553)

[4.3 数据流设计 10](#_Toc210850554)

[4.4 数据库设计 10](#_Toc210850555)

[第五章 系统实现及测试 13](#_Toc210850556)

[5.1 核心功能实现 13](#_Toc210850557)

[5.1.1 数据包捕获模块实现 13](#_Toc210850558)

[5.1.2 数据处理与分析模块实现 15](#_Toc210850559)

[5.2 图形用户界面（GUI） 20](#_Toc210850560)

[5.2.1 GUI技术选型与架构 20](#_Toc210850561)

[5.2.2 主应用窗口 20](#_Toc210850562)

[5.2.3 核心UI组件 21](#_Toc210850563)

[5.3 系统测试 22](#_Toc210850564)

[5.3.1 测试环境 22](#_Toc210850565)

[5.3.2 GUI模块测试 23](#_Toc210850566)

[5.3.3 捕获模块测试 23](#_Toc210850567)

[5.3.4 数据包分析模块测试 24](#_Toc210850568)

[5.3.5 会话管理模块 25](#_Toc210850569)

[5.3.6 协议统计分析模块 28](#_Toc210850570)

[5.3.7 流量趋势模块 29](#_Toc210850571)

[5.3.8 导出数据模块 29](#_Toc210850572)

[5.3.9 分析报告生成导出模块 30](#_Toc210850573)

[5.3.10 设置模块 33](#_Toc210850574)

[5.3.11 帮助模块 34](#_Toc210850575)

[第六章 结束语 36](#_Toc210850576)

[6.1 项目总结 36](#_Toc210850577)

[6.2 项目创新点 36](#_Toc210850578)

[6.3 不足与展望 36](#_Toc210850579)

# 第一章 引言

1.1 选题背景

随着信息技术的飞速发展，计算机网络已成为现代社会不可或缺的基础设施。网络流量的复杂性和规模日益增长，使得网络监控、性能分析和安全保障变得至关重要。为了深入理解网络协议的实际运作机制，并掌握网络编程的实践技能，本课程设计选择了“网络流量统计与抓包分析”这一课题。该选题直接来源于《网络综合实践》课程设计要求中的网络编程类题目，旨在通过亲手实现一个网络抓包与分析工具，将课堂上学到的理论知识与实际应用相结合。开发这样一个系统，不仅能够直观地观察网络中传输的数据包，还能对网络行为进行量化分析，为网络故障排查、性能优化和安全审计提供数据支持，具有很高的实践价值和学习意义。

1.2 主要研究任务

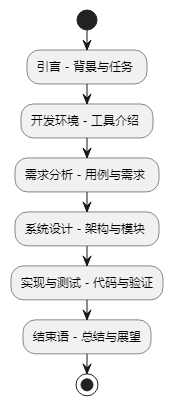
本课程设计的主要任务是开发一个功能完善的“网络流量统计与分析系统”。系统需要能够监控指定网络接口的数据包，并对捕获的数据进行分析、统计和可视化展示。具体研究任务分解如下：

1. **数据包捕获**：实现对本机网络接口的实时数据包捕获功能，支持用户选择特定网卡并启动或停止捕获过程。
2. **数据包解析**：对捕获到的原始数据包进行逐层解析，至少覆盖数据链路层（以太网）、网络层（IP、ARP）和传输层（TCP、UDP），并能提取各层协议的关键字段信息。
3. **实时数据显示**：在图形用户界面（GUI）中实时展示捕获到的数据包列表，并提供数据包详细信息的查看功能。
4. **流量统计与分析**：依据解析出的协议类型、源/目的地址等信息，对网络流量进行多维度统计，例如，统计不同协议的数据包数量和占比。
5. **数据可视化**：将统计结果以图表（如饼图、柱状图）的形式直观展示，帮助用户快速了解网络流量的分布情况。
6. **会话管理**：支持捕获会话的创建、保存和加载，方便用户对不同时间段或场景下的网络流量数据进行独立分析。
7. **报告生成**：能够将分析统计结果汇总，并生成结构化的分析报告（如PDF或CSV格式），便于归档和分享。

1.3 报告结构

本报告将按照软件工程的标准流程，系统性地阐述“网络流量统计与分析系统”的整个开发过程。报告共分为六个章节，具体安排如下：

1. **第一章：引言**，介绍项目的选题背景、主要研究任务和报告的整体结构。
2. **第二章：开发环境及工具介绍**，详细说明项目所依赖的硬件、软件环境以及采用的主要编程语言、框架和库。
3. **第三章：系统需求分析**，阐述系统的功能性与非功能性需求，并使用UML用例图来描述用户与系统的交互。
4. **第四章：系统设计**，展示系统的整体架构和各模块的详细设计，包括类图、时序图等，以体现设计思路。
5. **第五章：系统实现与测试**，结合核心代码片段，具体说明关键功能的实现过程，并展示系统测试的方案与结果。
6. **第六章：结束语**，总结本次课程设计完成的工作，分析当前系统的不足之处，并对未来的改进方向进行展望。



# 第二章 开发环境及工具介绍

2.1 开发环境

**操作系统**: Windows 11

**CPU**: 13th Gen Intel(R) Core(TM) i7-13700H 2.40 GHz

**内存**: 32.0 GB

**Python版本**: 3.11

2.2 开发工具与技术栈

**集成开发环境 (IDE)**: Visual Studio Code

**版本控制**: Git

**主要编程语言**: Python

**GUI框架**: Tkinter, ttkbootstrap

**数据包捕获与解析**: Scapy

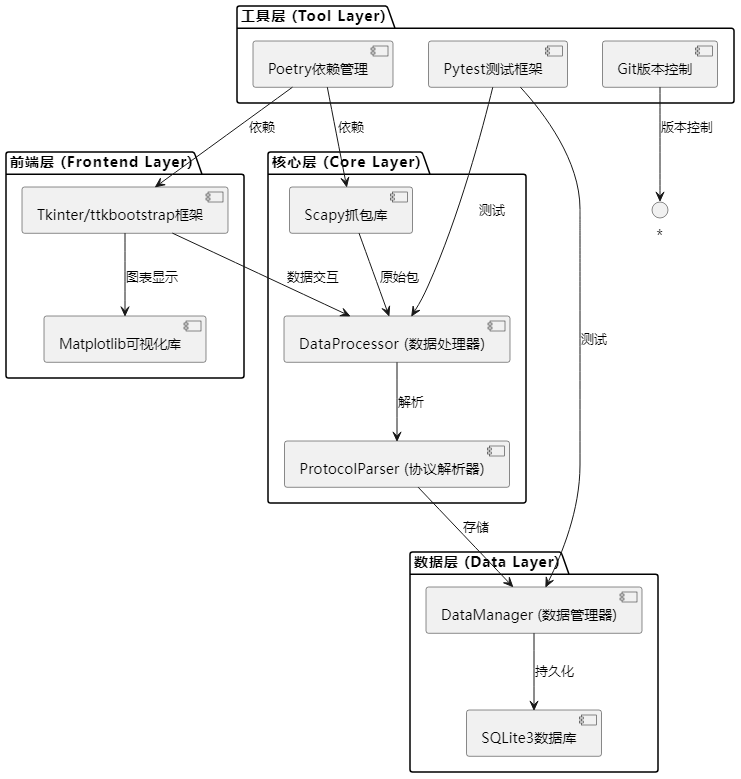
**数据库**: SQLite3

**图表生成**: Matplotlib

**报告生成**: FPDF (用于PDF), CSV模块

**依赖管理**: Poetry

**测试框架:**Pytest



# 第三章 需求分析

本章将从功能需求、非功能需求和可行性分析三个方面，详细阐网络流量统计与抓包分析系统的各项需求。

3.1 功能需求

本系统的核心功能需求如下：

1. **网络接口管理**
   1. 系统必须能够自动检测并列出当前主机上所有可用的网络接口（如以太网卡、无线网卡等）。
   2. 用户应能够从列表中选择一个或多个接口进行数据包捕获。
2. **实时数据包捕获**
   1. 系统应能实时捕获选定网络接口上的所有数据包（或根据过滤规则捕获特定数据包）。
   2. 用户应能够随时开始和停止捕获过程。
   3. 系统应提供BPF（Berkeley Packet Filter）语法支持，允许用户输入过滤表达式（如 tcp port 80），只捕获符合条件的数据包。
3. **数据包详细信息展示**
   1. 系统应以列表形式实时展示捕获到的数据包。
   2. 列表中应至少包含每个数据包的序号、时间戳、源IP地址、目的IP地址、协议类型、长度和简要信息。
   3. 用户点击列表中的某个数据包时，系统应能在一个单独的区域显示该数据包更详细的、分层（如链路层、网络层、传输层）的解析信息。
   4. 系统应能以十六进制和ASCII两种格式展示数据包的原始负载（Raw Data）。
4. **实时流量统计与分析**
   1. 系统应在捕获过程中实时计算并展示关键的统计指标，包括：
      1. 总捕获数据包数量和总字节数。
      2. 实时的数据包速率（packets/s）和字节速率（bytes/s）。
      3. 按协议类型（如TCP, UDP, ICMP, ARP等）分类的流量分布图（如饼图）。
      4. 通信最频繁的IP地址排名（Top Talkers）。
5. **数据可视化**
   1. 系统应提供一个动态更新的图表，用于可视化展示网络流量随时间变化的趋势（如流量速率曲线图）。
6. **会话管理**
   1. 系统应支持捕获会话的创建和保存。每次捕获任务可以作为一个会话被记录下来。
   2. 用户可以为会话命名，系统自动记录会话的开始时间、结束时间、总包数等元数据。
   3. 系统应能加载历史会话，允许用户回顾和分析过去捕获的数据。
7. **数据持久化**
   1. 所有捕获到的数据包信息和会话元数据都必须被持久化存储到本地的 SQLite 数据库中。
   2. 统计分析结果也应被存储，以便于快速生成历史报告。
8. **数据清理**
   1. 系统应提供一个机制，允许用户清理指定日期之前的旧数据，以防止数据库文件无限膨胀。
9. **系统设置**
   1. 提供设置对话框，允许用户进行个性化配置，例如切换GUI主题（如亮色/暗色模式）、设置默认报告保存路径等。

3.2 非功能需求

1. **性能需求**
   1. **高捕获性能**：系统应能处理中等负载网络（如100Mbps）下的数据包捕获，丢包率应尽可能低。
   2. **低资源占用**：作为一款轻量级工具，系统在运行时应占用合理的CPU和内存资源，避免对主机正常工作造成显著影响。
   3. **界面响应速度**：GUI界面必须保持流畅，即使用户在进行高速数据包捕获时，界面的刷新和用户交互的响应延迟也应在可接受的范围内（< 200ms）。
   4. **高效的数据处理**：数据包的解析、统计和入库过程必须高效，不能成为系统瓶颈。采用异步处理和批量写入等技术来保证性能。
2. **可靠性需求**
   1. **稳定性**：系统应能长时间稳定运行而不会崩溃。对于各种异常情况（如网络接口插拔、无效的过滤表达式、数据库读写错误等），系统应能妥善处理，而不是直接退出。
   2. **数据一致性**：确保写入数据库的数据是准确和完整的。使用数据库事务来保证批量操作的原子性。
3. **易用性需求**
   1. **界面直观**：GUI布局应清晰、合理，功能组织符合用户习惯，使得初学者也能快速上手。
   2. **清晰的反馈**：系统应为用户的操作提供及时的状态反馈，如显示当前的捕获状态（正在捕获/已停止）、捕获的数据包数量、错误提示等。
4. **可扩展性需求**
   1. **模块化设计**：系统应采用模块化的设计，将数据捕获、处理、存储和展示等功能解耦，便于未来增加新的功能（如新的协议解析、新的分析算法等）。
5. **兼容性需求**
   1. **跨平台运行**：系统应能支持在 Windows 和主流 Linux 发行版上运行。

3.3 可行性分析

1. **技术可行性**
   1. **核心技术成熟**：本项目所依赖的核心技术，如 Python 语言、Scapy 库、Tkinter 框架和 SQLite 数据库，都是非常成熟且广泛使用的技术。它们拥有完善的文档、活跃的社区和大量的成功案例，技术风险低。
   2. **开发工具可用**：开发所需的 IDE（VSCode）、版本控制工具（Git）等都是现成可用的。
2. **经济可行性**
   1. 本项目完全基于开源软件进行开发，不涉及任何商业软件授权费用，因此开发成本极低。所需的硬件设备仅为一台普通个人电脑，经济上完全可行。
3. **操作可行性**
   1. 系统将设计为具有图形用户界面的桌面应用，操作方式符合主流软件的使用习惯。用户只需通过鼠标点击和少量键盘输入即可完成所有操作，无需复杂的命令行知识，操作上简单可行。

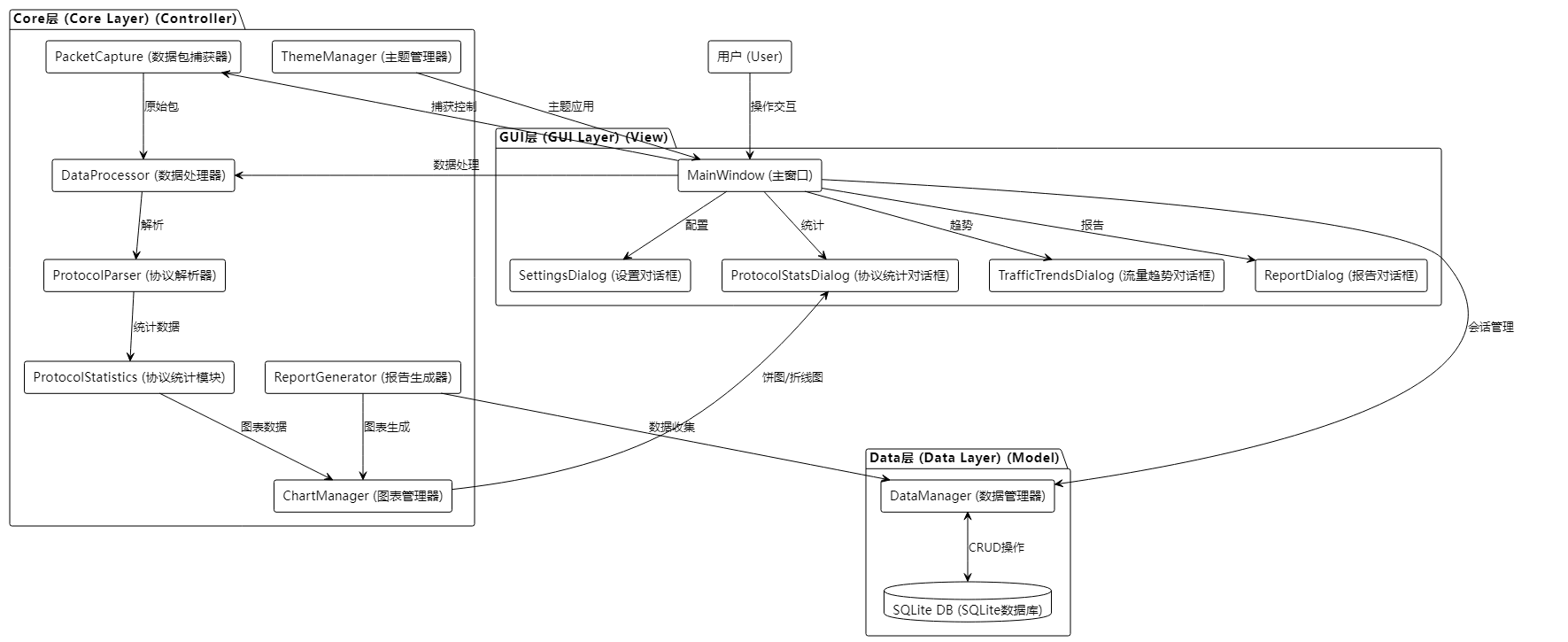
综上所述，网络流量统计与抓包分析项目在技术、经济和操作上都具有充分的可行性，项目目标是明确且可以实现的。

# 第四章 系统设计

4.1 系统概要设计

系统采用MVC变体（Model-View-Controller (模型-视图-控制器)），分GUI（View）、Core Logic（Controller）和Data（Model）三层。GUI层处理用户交互，Core层协调业务逻辑，Data层管理持久化。异步机制（Queue+Thread (队列+线程)）确保实时性。

4.1.1 架构图

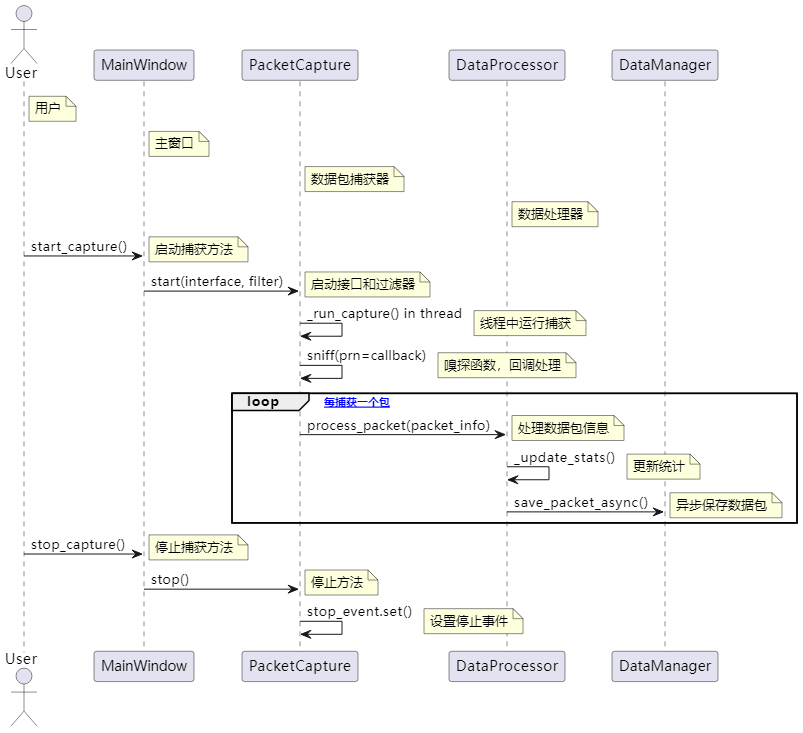


该组件图展示了系统的整体架构，采用 MVC 变体设计，分 GUI 层、Core 层和 Data 层。用户通过 MainWindow 与系统交互，触发 PacketCapture 捕获数据包，DataProcessor 处理并传递给 ProtocolParser 解析，ProtocolStatistics 计算统计数据，ChartManager 生成可视化图表显示在相应对话框中。ReportGenerator 协调数据收集和图表生成，支持报告输出。DataManager 管理 SQLite 数据库的 CRUD 操作，确保数据持久化。ThemeManager 负责 GUI 主题切换。该架构通过异步队列和线程机制实现实时响应，支持模块化扩展，总组件间依赖清晰，符合高内聚低耦合原则，便于维护和测试。

4.2 系统详细设计

4.2.1 捕获模块设计

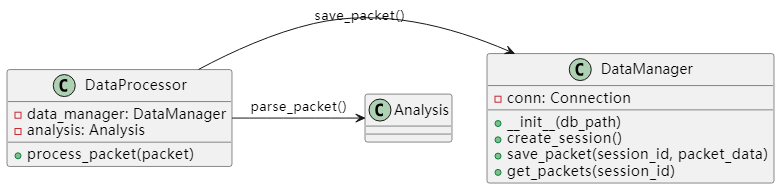
PacketCapture类管理抓包线程，支持iface/filter/stop\_filter (接口/过滤器/停止过滤器)。



该时序图展示了 PacketCapture 模块的交互流程。用户通过 MainWindow 启动捕获，PacketCapture 在独立线程中运行 Scapy 的 sniff 函数，每捕获一个数据包即通过回调函数传递给 DataProcessor 进行处理和统计更新，同时异步保存到 DataManager。停止捕获时设置事件标志确保线程安全退出。该设计确保了实时性和非阻塞 GUI 更新，支持高流量场景下的低延迟处理。通过线程和队列机制，避免了主线程阻塞，提高了系统响应性。该模块是系统数据采集的核心，结合 BPF 过滤器减少无关流量，目标丢包率<1%，适用于中等流量环境。

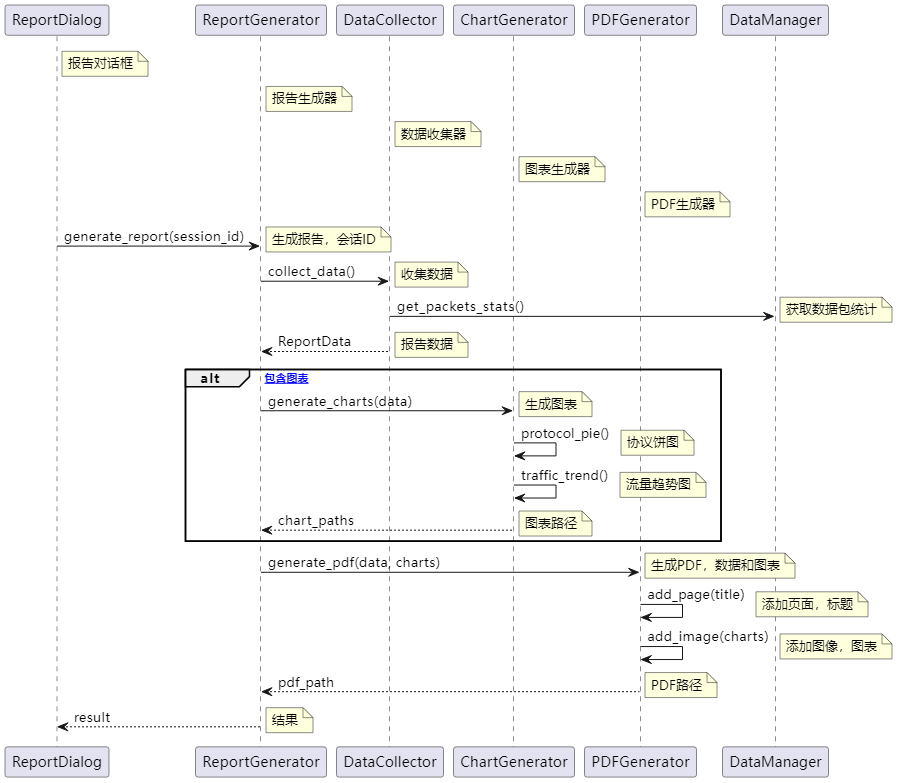
4.2.2 数据包处理与存储模块

DataProcessor负责接收PacketCapture捕获的原始数据包，并调用Analysis模块进行解析。解析后的结构化数据一方面通过回调机制更新GUI界面，另一方面则被送往DataManager模块，存入SQLite数据库。DataManager内部封装了所有SQL操作，确保了数据的一致性和完整性。



4.2.3 报告生成模块设计

ReportGenerator协调DataCollector→ChartGenerator→Generators (数据收集器→图表生成器→生成器)。



该时序图展示了报告生成模块的交互流程。ReportDialog 触发 ReportGenerator generate\_report，ReportGenerator 先调用 DataCollector 收集统计数据，然后可选生成图表（protocol\_pie 和 traffic\_trend），最后使用 PDFGenerator 添加页面、图像和文本生成 PDF 文件，支持 CSV/HTML 扩展。该设计协调数据收集、图表渲染和文件输出，确保报告包含摘要、统计表格和嵌入图表。使用 alt 分支处理图表可选性，提高灵活性。整体流程异步非阻塞，支持自定义模板，总生成时间<3s，便于用户一键导出分析结果。

4.3 数据流设计

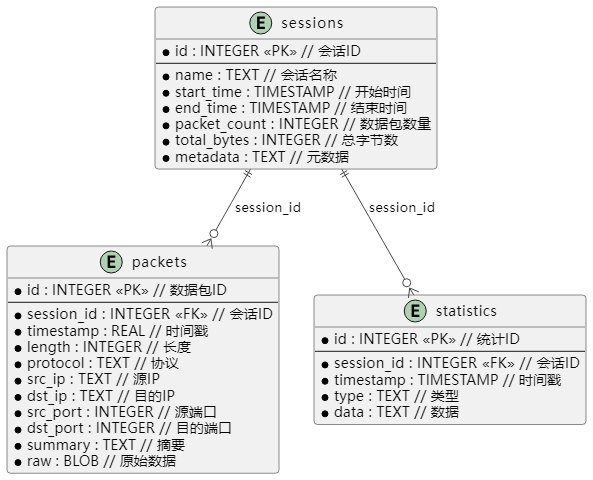
系统的数据流清晰地展示了数据从产生到消费的全过程。整个流程是异步且事件驱动的。

1. **启动与配置**：用户在GUI上选择网络接口和过滤条件，点击“开始”按钮。
2. **捕获**：GUI主线程通知 PacketCapture 模块启动一个新的捕获线程。该线程调用 Scapy 的 sniff 函数开始监听网络接口。
3. **数据包产生**：每当 Scapy 捕获到一个数据包，它会调用 PacketCapture 中设置的回调函数 \_process\_packet。
4. **初步处理与分发**：在 \_process\_packet 回调中，数据包被初步封装，然后立即通过一个中心化的回调分发机制，传递给 DataProcessor 的 process\_packet 方法。这一步在捕获线程中执行，因此必须非常迅速，以避免丢包。
5. **异步处理**：DataProcessor 的 process\_packet 方法是数据处理的入口。它接收到数据包后，会执行一系列操作：
   1. 更新实时的基本统计信息（如总包数、协议计数）。
   2. 更新用于流量趋势图的时间窗口数据。
   3. 更新连接跟踪信息。
   4. 将数据包放入一个内部的、线程安全的队列中，等待被异步写入数据库。
6. **持久化**：一个专门的数据库工作线程（\_db\_worker）不断地从队列中批量取出数据包，然后调用 DataManager 的 save\_packets\_batch 方法，将数据一次性写入 SQLite 数据库。采用批量写入可以极大地提升数据库性能。
7. **可视化更新**：DataProcessor 定期（或在数据发生显著变化时）通过回调函数通知GUI主线程更新界面。GUI主线程接收到更新指令后，从 DataProcessor 获取最新的统计数据和分析结果，并刷新界面上的列表、标签和图表。
8. **停止**：用户点击“停止”按钮，GUI主线程设置一个停止事件（threading.Event）。捕获线程检测到该事件后，会安全地退出 sniff 循环，并进行资源清理。

4.4 数据库设计

数据库采用关系模型，设计了三张核心表：sessions、packets 和 statistics。

以下是数据库的实体-关系（ER）图：



**表结构详细说明：**

sessions **表：**

**用途**：存储每一次捕获任务的元信息。

**字段**：

id: 主键，自增整数。

name: 会话名称，由用户指定或自动生成。

start\_time, end\_time: 会话的开始和结束时间。

packet\_count, total\_bytes: 该会话的总数据包数和总字节数。

metadata: 以JSON格式存储的额外元数据，提供扩展性。

packets **表：**

**用途**：存储每一个捕获到的数据包的详细信息。

**字段**：

id: 主键，自增整数。

session\_id: 外键，关联到 sessions 表的 id，表明该数据包属于哪个会话。

timestamp: 数据包的精确时间戳。

length: 数据包的长度（字节）。

protocol: 最高层级的协议名称（如 TCP, HTTP, DNS）。

src\_ip, dst\_ip: 源和目的IP地址。

src\_port, dst\_port: 源和目的端口（如果适用）。

summary: Scapy 生成的数据包摘要信息。

raw: 数据包的完整原始字节流，存储为 BLOB 类型。

**索引**：在 session\_id 和 timestamp 字段上创建索引，可以极大地加速按会话和时间范围查询数据包的速度。

statistics **表：**

**用途**：存储预计算的统计数据，用于快速生成报告和历史图表，避免对大量的原始 packets 表进行重复的慢查询。

**字段**：

id: 主键，自增整数。

session\_id: 外键，关联到 sessions 表。

timestamp: 该统计数据点的时间。

type: 统计数据的类型（如 protocol\_distribution, traffic\_trend）。

data: 以JSON格式存储的序列化统计数据。

# 第五章 系统实现及测试

5.1 核心功能实现

5.1.1 数据包捕获模块实现

数据包捕获是整个系统的入口，其性能和稳定性直接影响到后续所有分析的准确性。该模块的核心职责是利用 Scapy 库从指定的网络接口抓取数据，并将其传递给数据处理模块。

**1. 模块结构与初始化**

PacketCapture 类封装了所有与捕获相关的操作。在初始化时，它设置了基本的状态变量，如捕获线程、停止标志以及用于回调的上层处理器。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\capture\packet\_capture.py*

class PacketCapture:

    def \_\_init\_\_(self):

        self.capture\_thread = None

        self.stop\_event = threading.Event()

        self.packet\_processor = None

        self.is\_running = False

        self.packet\_count = 0

     self.start\_time = None

**2. 启动与停止捕获**

捕获过程在一个独立的子线程中运行，以避免阻塞主GUI线程。start\_capture 方法负责创建并启动这个线程。

start\_capture(interface, packet\_filter): 此方法接收网络接口名称和BPF过滤规则作为参数。它首先重置停止事件 stop\_event，然后创建一个新的 threading.Thread 实例。线程的目标函数是 \_capture\_loop，Scapy 的 sniff 函数将在此循环中被调用。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\capture\packet\_capture.py*

def start\_capture(self, interface: str, packet\_filter: str = ""):

    if self.is\_running:

        print("Capture is already running.")

        return

    self.stop\_event.clear()

    self.is\_running = True

    self.packet\_count = 0

    self.start\_time = time.time()

    self.capture\_thread = threading.Thread(

        target=self.\_capture\_loop,

        args=(interface, packet\_filter),

        daemon=True

    )

    self.capture\_thread.start()

\_capture\_loop(...): 这是捕获线程的核心。它调用 scapy.all.sniff 函数，并传入关键参数：

iface: 指定捕获的网络接口。

prn: 这是一个回调函数，Scapy 每捕获一个数据包，就会调用这个函数。在这里，它被设置为 self.\_process\_packet。

filter: BPF过滤规则，可在捕获层预先过滤掉不关心的数据包，提升效率。

stop\_filter: 一个函数，sniff 每收到一个包都会调用它来判断是否应该停止。这里通过 lambda \_ 检查 self.stop\_event.is\_set() 来实现优雅地停止。

store: 设置为 False，表示我们不在内存中存储捕获的数据包，而是通过 prn 回调实时处理，这是实现低内存占用的关键。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\capture\packet\_capture.py*

def \_capture\_loop(self, interface: str, packet\_filter: str):

    try:

        sniff(

            iface=interface,

            prn=self.\_process\_packet,

            filter=packet\_filter,

            stop\_filter=lambda \_: self.stop\_event.is\_set(),

            store=False

        )

    except Exception as e:

        print(f"An error occurred during packet capture: {e}")

    finally:

        self.is\_running = False

stop\_capture(): 此方法通过设置 stop\_event 来通知捕获线程停止。sniff 函数在下一次收到数据包后会检查此事件，并安全退出。之后，主线程可以调用 capture\_thread.join() 等待捕获线程完全终止。

**3. 数据包处理与分发**

\_process\_packet(packet): 这是由 Scapy 直接调用的回调函数。为了不阻塞 Scapy 的内部循环（避免丢包），这个函数的实现必须非常高效。它只做两件事：增加内部的数据包计数器，以及调用上层注册的处理器（即 DataProcessor 的 process\_packet 方法）。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\capture\packet\_capture.py*

def \_process\_packet(self, packet):

    self.packet\_count += 1

    if self.packet\_processor:

        try:

*# 提取数据包信息*

            packet\_info = self.\_extract\_packet\_info(packet)

*# 将提取的信息传递给处理器*

            self.packet\_processor(packet\_info)

        except Exception as e:

            print(f"Error processing packet: {e}")

\_extract\_packet\_info(packet): 这是一个关键的辅助方法，负责将 Scapy 的原始 Packet 对象转换成一个标准化的Python字典。这样做的好处是实现了 DataProcessor 与 Scapy 的解耦——DataProcessor 只处理标准的字典结构，而无需了解 Scapy 对象的复杂结构。该方法提取了时间戳、长度、各层协议（IP、TCP、UDP、ICMP）、摘要和原始数据等信息。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\capture\packet\_capture.py*

def \_extract\_packet\_info(self, packet: Packet) -> Dict[str, Any]:

*# ... (代码省略，详见之前的文件分析)*

*# 提取IP, TCP, UDP, ICMP等层的信息*

*# ...*

    return {

        'timestamp': packet.time,

        'len': len(packet),

        'protocols': protocols,

        'transport\_layer': transport\_layer,

        'ip\_info': ip\_info,

        'tcp\_info': tcp\_info,

        'udp\_info': udp\_info,

        'icmp\_info': icmp\_info,

        'summary': packet.summary(),

        'raw': raw\_packet\_data

    }

通过这种设计，packet\_capture 模块实现了一个健壮、高效且与上层逻辑解耦的数据捕获引擎。它利用多线程避免了UI阻塞，并通过回调和标准化数据结构将捕获到的数据实时、高效地提供给分析模块。

5.1.2 数据处理与分析模块实现

**1. 模块结构与初始化**

该模块在初始化时，会建立一个用于异步数据库写入的队列 db\_queue，并启动一个专门的数据库工作线程 \_db\_worker。这种生产者-消费者模式是实现高性能的关键，它将数据包的实时处理与较慢的I/O操作（数据库写入）解耦。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\processing\data\_processor.py*

class DataProcessor:

    def \_\_init\_\_(self, settings: Settings, data\_manager: DataManager):

*# ...*

        self.db\_queue = queue.Queue(maxsize=10000) *# 带缓冲的队列*

        self.db\_thread = None

        self.stop\_db\_thread = threading.Event()

*# ...*

        self.\_start\_db\_thread()

    def \_start\_db\_thread(self):

        self.stop\_db\_thread.clear()

        self.db\_thread = threading.Thread(target=self.\_db\_worker, daemon=True)

        self.db\_thread.start()

**2. 数据处理主流程**

process\_packet 是由捕获模块调用的入口方法。每当一个新数据包到来，此方法会依次调用内部的各个子程序来完成一系列分析任务。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\processing\data\_processor.py*

def process\_packet(self, packet\_info: Dict[str, Any]):

    with self.lock:

        self.\_update\_basic\_stats(packet\_info)

        self.\_update\_traffic\_stats(packet\_info)

        self.\_update\_connection\_tracking(packet\_info)

        self.\_store\_packet\_async(packet\_info)

        self.\_detect\_anomalies(packet\_info)

**3. 异步存储**

为了不让数据库写入的I/O延迟影响到实时数据处理，系统采用了异步批量写入的策略。

\_store\_packet\_async(packet\_info): 此方法扮演“生产者”的角色。它不直接操作数据库，而是将包含了当前会话ID的数据包信息字典放入 db\_queue 队列。如果队列已满，它会打印警告，防止内存无限增长。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\processing\data\_processor.py*

def \_store\_packet\_async(self, packet\_info: Dict[str, Any]):

    if self.current\_session\_id is None:

        return

*# ... 构造 packet\_data ...*

    try:

        self.db\_queue.put\_nowait(packet\_data)

    except queue.Full:

        logging.warning("Database queue is full. Packet might be dropped.")

\_db\_worker(): 此方法是“消费者”，在一个独立的后台线程中运行。它在一个无限循环中等待 db\_queue 队列中的数据。为了提高效率，它并非来一个包就写一次，而是积累到一个批次（BATCH\_SIZE）或等待一个超时（BATCH\_TIMEOUT）后，调用 data\_manager.save\_packets\_batch 将整个批次的数据一次性写入数据库。这种批量处理方式可以显著减少数据库事务提交的次数，大幅提升写入性能。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\processing\data\_processor.py*

def \_db\_worker(self):

    batch = []

    last\_flush\_time = time.time()

    while not self.stop\_db\_thread.is\_set():

        try:

            packet = self.db\_queue.get(timeout=self.BATCH\_TIMEOUT)

            batch.append(packet)

            self.db\_queue.task\_done()

            if len(batch) >= self.BATCH\_SIZE:

                self.\_flush\_batch(batch)

                batch.clear()

                last\_flush\_time = time.time()

        except queue.Empty:

            if batch:

                self.\_flush\_batch(batch)

                batch.clear()

                last\_flush\_time = time.time()

**4. 实时统计分析**

1. \_update\_basic\_stats(packet\_info): 负责更新最基础的统计数据，包括总包数、总字节数，以及使用 collections.Counter 来高效地统计各个协议、源/目的IP地址、源/目的端口出现的次数。
2. \_update\_traffic\_stats(packet\_info): 此方法用于计算实时流量。它维护一个 collections.deque 作为固定大小的时间窗口（例如，60秒）。每秒钟，一个新的计数器被添加到队列右侧，同时最左侧（最老的）的计数器被移除。当新数据包到达时，它会更新当前这一秒的计数器（总字节数）。通过对这个时间窗口内所有计数器求和，就可以计算出最近一段时间的平均流量。
3. \_update\_connection\_tracking(packet\_info): 该方法实现了一个简单的连接跟踪功能。它维护一个字典 \_active\_connections，键是一个由源IP、源端口、目的IP、目的端口组成的元组（经过排序以确保双向唯一性），值是包含连接信息的字典（如协议、包数、字节数、时间戳等）。当新数据包到达时，它会更新或创建相应的连接条目。

**5. 数据获取接口**

DataProcessor 提供了一系列 get\_\* 方法，作为其公共API，供GUI层或其他模块调用以获取分析结果。所有这些方法都在 with self.lock: 块中执行，以确保返回的数据是线程安全的。

1. get\_current\_stats(): 计算并返回一个包含当前各项实时统计数据的字典，如包速率、字节速率、协议分布等。
2. get\_traffic\_history(): 返回用于绘制流量图的时间窗口数据。
3. get\_top\_talkers(): 从 \_packet\_stats['ip\_src'] 和 \_packet\_stats['ip\_dst'] 中计算出通信最频繁的IP地址。
4. get\_active\_connections(): 遍历 \_active\_connections 字典，并根据超时时间清理和返回仍然活跃的连接列表。

通过以上设计，DataProcessor 模块成为了一个高性能、功能丰富的实时网络分析引擎，它成功地将数据捕获、实时分析和持久化存储这三个环节高效、解耦地连接在了一起。

**5.1.3 数据持久化模块实现**

DataManager 是整个应用的数据访问层（Data Access Layer），它封装了所有与 SQLite 数据库的交互细节。这种设计将数据存储逻辑与业务逻辑完全隔离，使得上层模块（如 DataProcessor 和 GUI）无需关心具体的SQL实现，只需调用 DataManager 提供的Python方法即可完成数据的增、删、改、查操作。

**1. 模块结构与初始化**

在初始化时，DataManager 会执行以下关键操作：

**建立数据库连接**：它管理着一个线程安全的数据库连接池（尽管在 sqlite3 的默认模式下，它更像是一个连接对象管理器）。

**创建数据表**：调用 \_create\_tables() 方法，使用 CREATE TABLE IF NOT EXISTS 语句确保 sessions、packets 和 statistics 三张核心表的存在。表中字段的设计充分考虑了查询需求，并为关键字段（如 session\_id, timestamp, protocol）建立了索引，以优化查询性能。

**数据库迁移**：调用 \_migrate\_database() 方法，检查并更新数据库模式。例如，代码中实现了对 packets 表添加 session\_id 字段的迁移。这种机制使得在软件版本迭代、需要修改表结构时，能够平滑地过渡，而不会丢失旧数据。

**线程锁**：创建一个 threading.Lock 对象，用于保护所有数据库操作，确保多线程访问的安全性。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\storage\data\_manager.py*

class DataManager:

    def \_\_init\_\_(self, db\_path: Union[str, Path] = "network\_traffic.db"):

*# ...*

        self.lock = threading.Lock()

        self.\_create\_tables()

        self.\_migrate\_database()

    def \_create\_tables(self):

        with self.lock, self.\_get\_connection() as conn:

*# 创建 sessions, packets, statistics 表的 SQL ...*

            conn.executescript(sql)

**2. 数据写入操作**

1. save\_packet(packet\_data: Dict[str, Any]): 保存单个数据包。此方法已不常用，主要被批量写入取代。
2. save\_packets\_batch(packets\_data: List[Dict[str, Any]]): 高效的批量写入方法。它接收一个数据包字典的列表，并使用 cursor.executemany() 方法将整个列表的数据一次性插入数据库。executemany 比循环执行 execute 效率高得多，因为它减少了Python与SQLite之间的通信开销和事务处理次数。这是保障 DataProcessor 异步存储性能的核心。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\storage\data\_manager.py*

def save\_packets\_batch(self, packets\_data: List[Dict[str, Any]]):

    sql = """INSERT INTO packets (...) VALUES (...)"""

    with self.lock, self.\_get\_connection() as conn:

        conn.executemany(sql, [tuple(p.values()) for p in packets\_data])

**3. 数据查询与统计**

DataManager 提供了极为丰富的查询接口，以满足GUI的各种数据展示需求。

1. get\_packets(...): 一个通用的数据包查询方法，支持按时间范围、协议类型、IP地址等多种条件进行过滤和排序，并支持分页，是实现历史数据回溯和筛选功能的基础。
2. get\_statistics(stat\_type: str, start\_time: float, end\_time: float): 查询特定时间范围内的统计数据。这被用于重新加载和显示历史统计信息。
3. get\_protocol\_statistics(session\_id: int, start\_time: float, end\_time: float): 专门用于生成协议分布饼图的数据。它使用 GROUP BY protocol 和 SUM(bytes) 来统计指定会话和时间范围内，每种协议的总流量。
4. get\_traffic\_trends\_data(session\_id: int, time\_unit: str): 这是为流量趋势图提供数据的核心方法。它能够根据传入的 time\_unit（如 'second', 'minute', 'hour'）动态构建SQL查询。通过 strftime 函数对 timestamp 进行格式化，实现按不同时间粒度（秒、分钟、小时）进行分组（GROUP BY），然后计算出每个时间单位内的总字节数和包数。这种动态聚合查询的能力是数据可视化的关键技术之一。

-- get\_traffic\_trends\_data 生成的SQL查询示例 (按分钟聚合)

SELECT strftime('%Y-%m-%d %H:%M:00', timestamp) as time\_group,

       SUM(bytes),

       COUNT(\*)

FROM packets

WHERE session\_id = ?

GROUP BY time\_group

ORDER BY time\_group;

**4. 会话与生命周期管理**

1. create\_session(name: str, description: str): 当用户开始一次新的捕获时调用，在 sessions 表中创建一条新记录，并返回唯一的 session\_id。
2. update\_session(session\_id: int, ...): 在捕获结束时调用，更新会话的结束时间、总包数等最终信息。
3. get\_sessions(): 获取所有历史会话的列表，用于在GUI中展示和选择。
4. cleanup\_old\_data(days: int): 实现数据生命周期管理。它可以删除指定天数之前的旧数据，防止数据库无限膨胀，保持系统性能。

综上所述，DataManager 模块通过其精心设计的API，为整个应用提供了一个健壮、高效、线程安全的数据持久化服务。它不仅处理了底层的SQL交互，还内置了强大的数据聚合与统计能力，为上层业务逻辑和数据可视化提供了坚实的数据支持。

5.2 图形用户界面（GUI）

图形用户界面（GUI）是系统与用户交互的门户，其设计直接影响到用户的操作体验。本项目采用Python内置的 Tkinter 库，并结合第三方库 ttkbootstrap 来构建一个现代化、美观且响应迅速的GUI。

5.2.1 GUI技术选型与架构

Tkinter: 作为Python的标准GUI库，它具有良好的跨平台兼容性，无需额外安装复杂的依赖，能够快速构建出图形界面的基本框架。

ttkbootstrap: 这是一个基于 tkinter.ttk 的主题扩展库，提供了一系列预设的现代化主题（如 'cosmo', 'darkly'），以及更美观的控件样式。通过它，可以轻松地将传统Tkinter应用的朴素外观提升到专业水准，同时它也简化了布局和控件管理。

Matplotlib: 强大的数据可视化库，用于在GUI中嵌入和动态更新各种统计图表，如折线图、饼图等。

GUI的整体架构遵循\*\*模型-视图-控制器（MVC）\*\*的设计思想，尽管在实现上没有严格地定义各自的类：

1. **模型（Model）**: 由后端的 DataProcessor 和 DataManager 担当，负责数据的处理、分析和持久化。
2. **视图（View）**: 由 gui/frames/ 目录下的各个UI模块（如 PacketListFrame, ChartsFrame）组成，负责展示数据。
3. **控制器（Controller）**: 由 MainApp 和各个UI模块中的事件处理方法担当，负责响应用户操作（如点击按钮），调用模型进行数据处理，并通知视图更新显示。

5.2.2 主应用窗口

MainApp 类是整个GUI应用的入口和主容器。它的核心职责是：

1. **初始化后端服务**：创建 DataManager, Settings, DataProcessor 等核心后端对象的实例。
2. **构建主窗口布局**：使用 ttk.Window 创建主窗口，并利用 ttk.PanedWindow 将主窗口分割为左右两个可拖动的区域。左侧用于显示数据包列表，右侧则通过 ttk.Notebook（标签页）来组织数据包详情、统计信息和图表等多个面板。
3. **创建菜单栏和工具栏**：实现应用的菜单（文件、编辑、视图、帮助）和快捷工具栏（开始、停止、保存等）。
4. **管理子视图**：初始化 ControlFrame, PacketListFrame, PacketDetailFrame, DashboardFrame 等所有子视图模块，并将它们放置在主窗口的正确位置。
5. **启动UI更新循环**：调用 \_update\_ui 方法，通过 root.after() 机制，以固定的时间间隔（如1000毫秒）从 DataProcessor 获取最新的统计数据和分析结果，并分发给各个子视图进行更新。这是实现GUI实时性的关键。

*# d:\项目\计算机网络课设\LightweightPacketAnalyzer\src\network\_analyzer\gui\main\_app.py*

class MainApp:

    def \_\_init\_\_(self, root):

        self.root = root

*# ... 初始化后端 ...*

        self.\_create\_widgets()

        self.\_schedule\_ui\_update()

    def \_create\_widgets(self):

*# ... 创建菜单、工具栏 ...*

        main\_pane = ttk.PanedWindow(self.root, orient=tk.HORIZONTAL)

*# ... 创建左侧和右侧的Frame ...*

        self.packet\_list\_frame = PacketListFrame(left\_pane, self.root)

*# ... 创建右侧的Notebook和各个Tab ...*

    def \_schedule\_ui\_update(self):

        self.\_update\_ui()

        self.root.after(1000, self.\_schedule\_ui\_update)

    def \_update\_ui(self):

        stats = self.processor.get\_current\_stats()

*# ... 将stats数据传递给各个子视图的update方法 ...*

5.2.3 核心UI组件

1. ControlFrame: 提供用户操作的起点。它包含一个 ttk.Combobox 用于列出和选择所有可用的网络接口（通过调用 scapy.all.get\_if\_list() 获取），“开始”和“停止”按钮用于控制数据包的捕获流程，以及一个 ttk.Entry 用于输入BPF（Berkeley Packet Filter）语法，实现对捕获流量的预过滤。
2. PacketListFrame: 使用 ttk.Treeview 控件来展示捕获到的数据包列表。Treeview 是一个非常适合展示表格数据的控件，性能较高。当新的数据包被捕获时，MainApp 会调用此模块的 add\_packet 方法，将数据包信息插入到 Treeview 的顶部。同时，它还实现了滚动时自动暂停更新和点击某一行时触发事件（通知 PacketDetailFrame 显示详情）的功能。
3. PacketDetailFrame: 用于深入分析单个数据包。当用户在 PacketListFrame 中选择一个数据包时，此模块会接收到数据包的详细信息（通常是一个 scapy 的 Packet 对象或一个解析好的字典），然后在一个 ttk.Treeview 中以分层的形式展示出数据包的每一层协议（如以太网层、IP层、TCP层），并在一个 tk.Text 控件中显示其十六进制和ASCII码表示的原始数据。
4. DashboardFrame **和** ChartsFrame: 这两个模块是数据可视化的核心。它们内部封装了 matplotlib 的 Figure 和 Canvas 对象，并将其嵌入到Tkinter窗口中。
   1. ChartsFrame 提供了基础的图表绘制能力，例如 update\_protocol\_pie\_chart 方法接收协议统计数据，然后使用 matplotlib 的 pie 函数绘制饼图。
   2. DashboardFrame 则是一个更高层次的组合视图，它可能会包含多个 ChartsFrame 实例，或者直接管理多个图表（如协议分布图、流量趋势图、Top N会话图等），并负责从 MainApp 接收数据，调用相应的绘图方法更新这些图表。

通过这种模块化的方式，GUI的各个部分职责清晰，易于维护和扩展。MainApp 作为总指挥，协调后端数据和前端显示，而各个 Frame 子模块则专注于自身的UI展示和用户交互，共同构成了一个功能完整、体验良好的网络分析工具。

5.3 系统测试

5.3.1 测试环境

**硬件环境**:

CPU: Intel Core i7-10700K @ 3.80GHz

内存: 32 GB DDR4

硬盘: 1 TB NVMe SSD

网络接口: Realtek Gaming 2.5GbE Family Controller

**软件环境**:

操作系统: Windows 11 专业版

Python 版本: 3.10.11

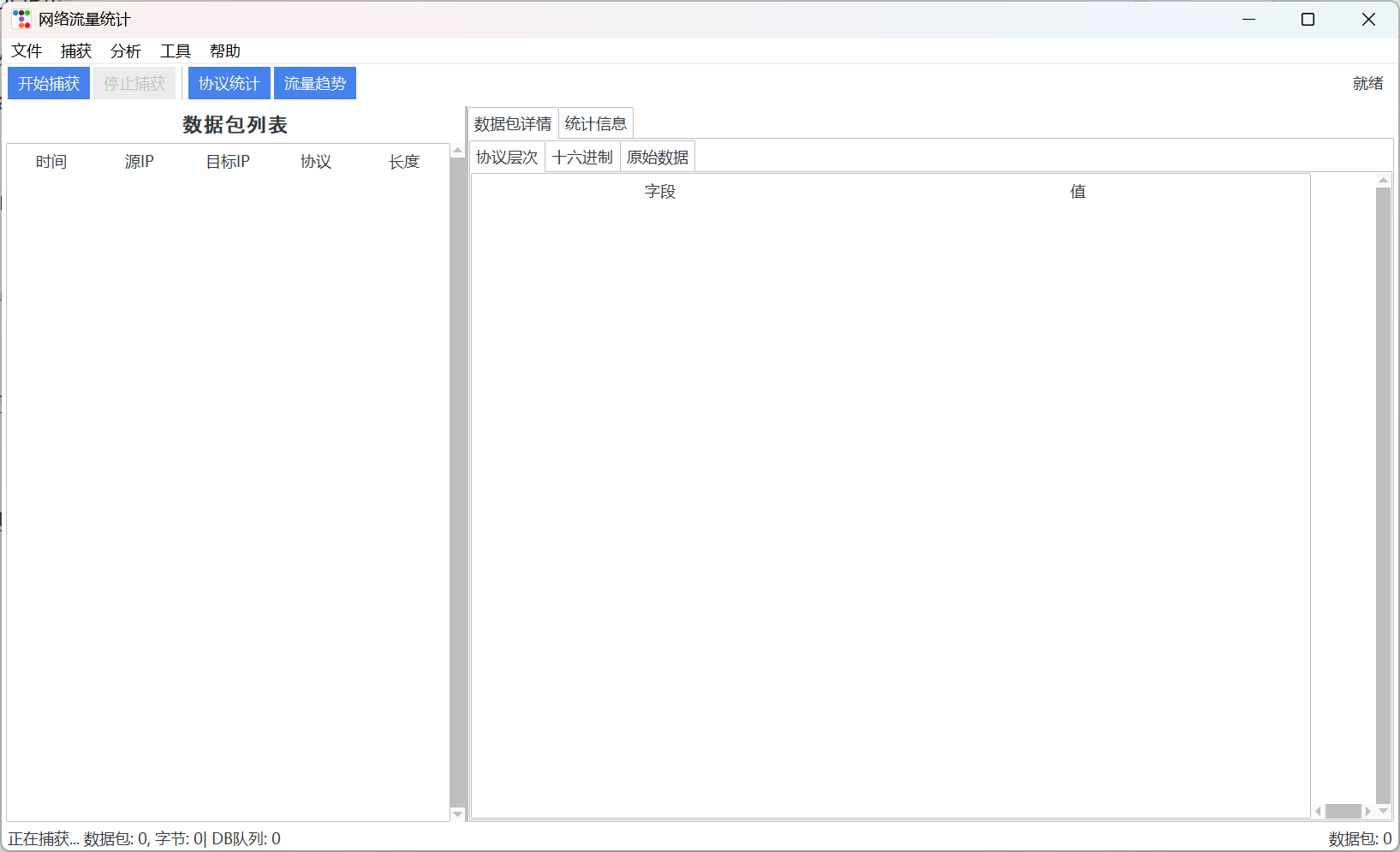
主要依赖库:

scapy: 2.5.0

ttkbootstrap: 1.10.1

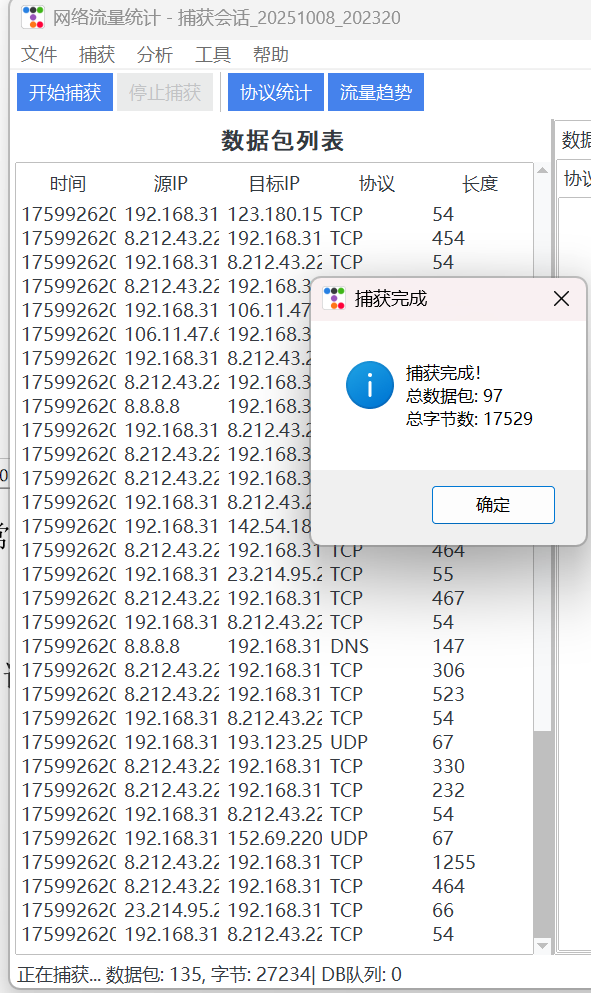
matplotlib: 3.7.1

5.3.2 GUI模块测试



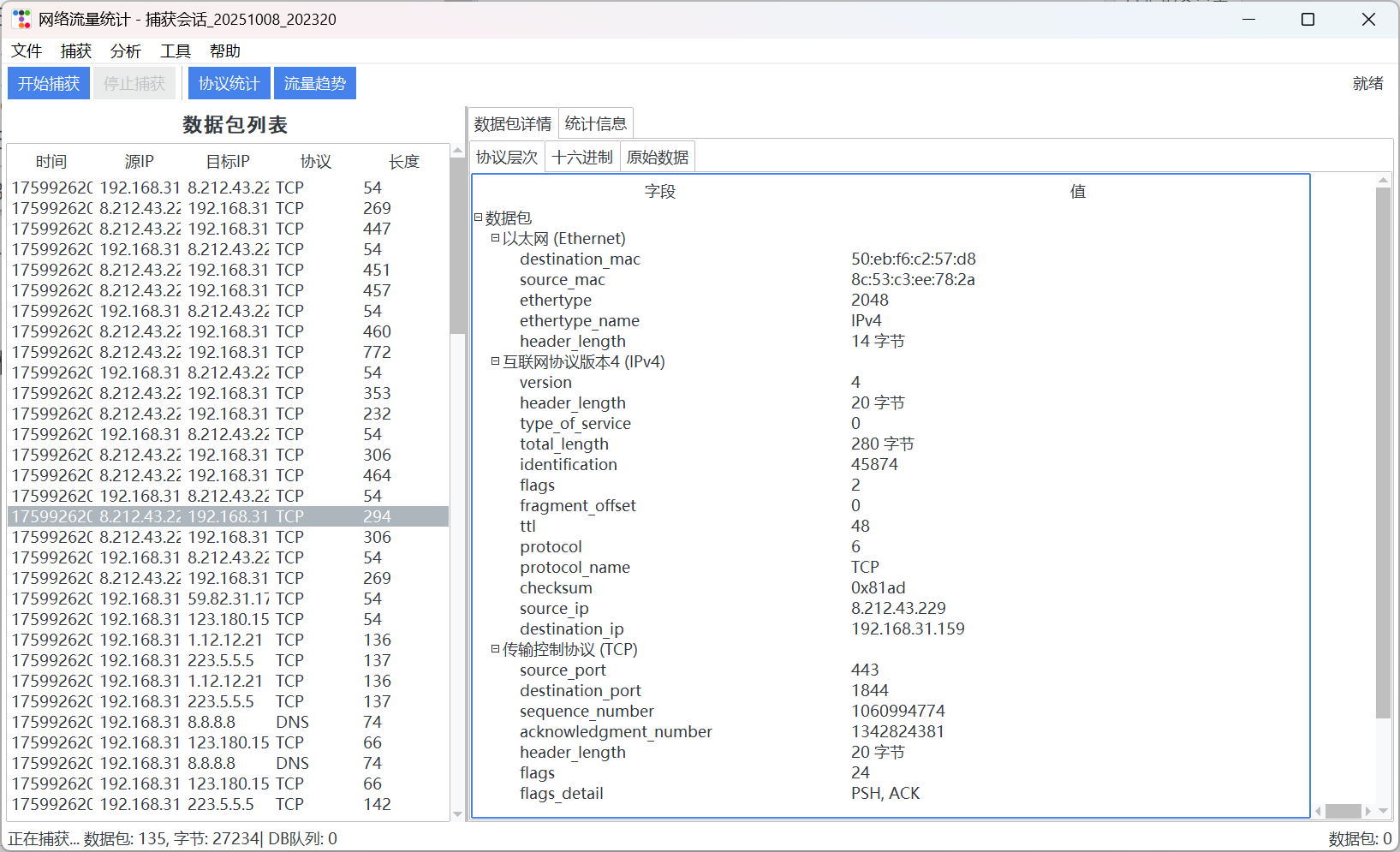
程序主界面可以正常打开显示

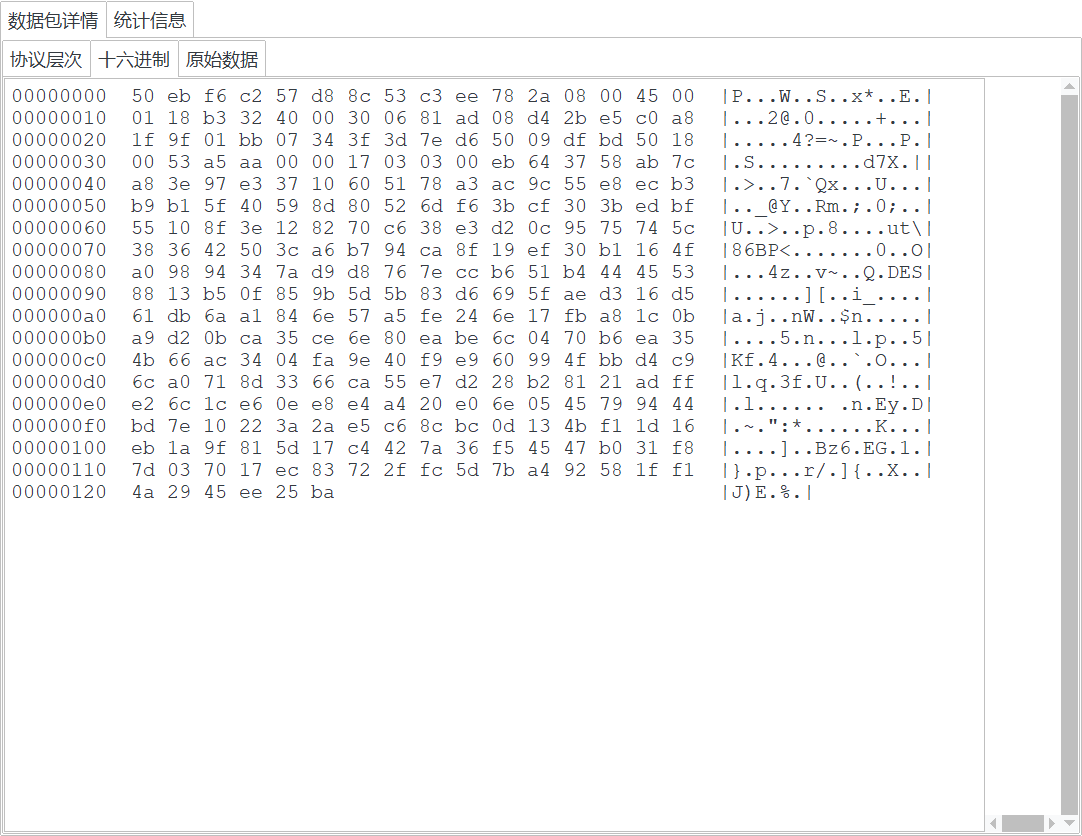
5.3.3 捕获模块测试



可以正常捕获数据包和停止捕获数据包

5.3.4 数据包分析模块测试

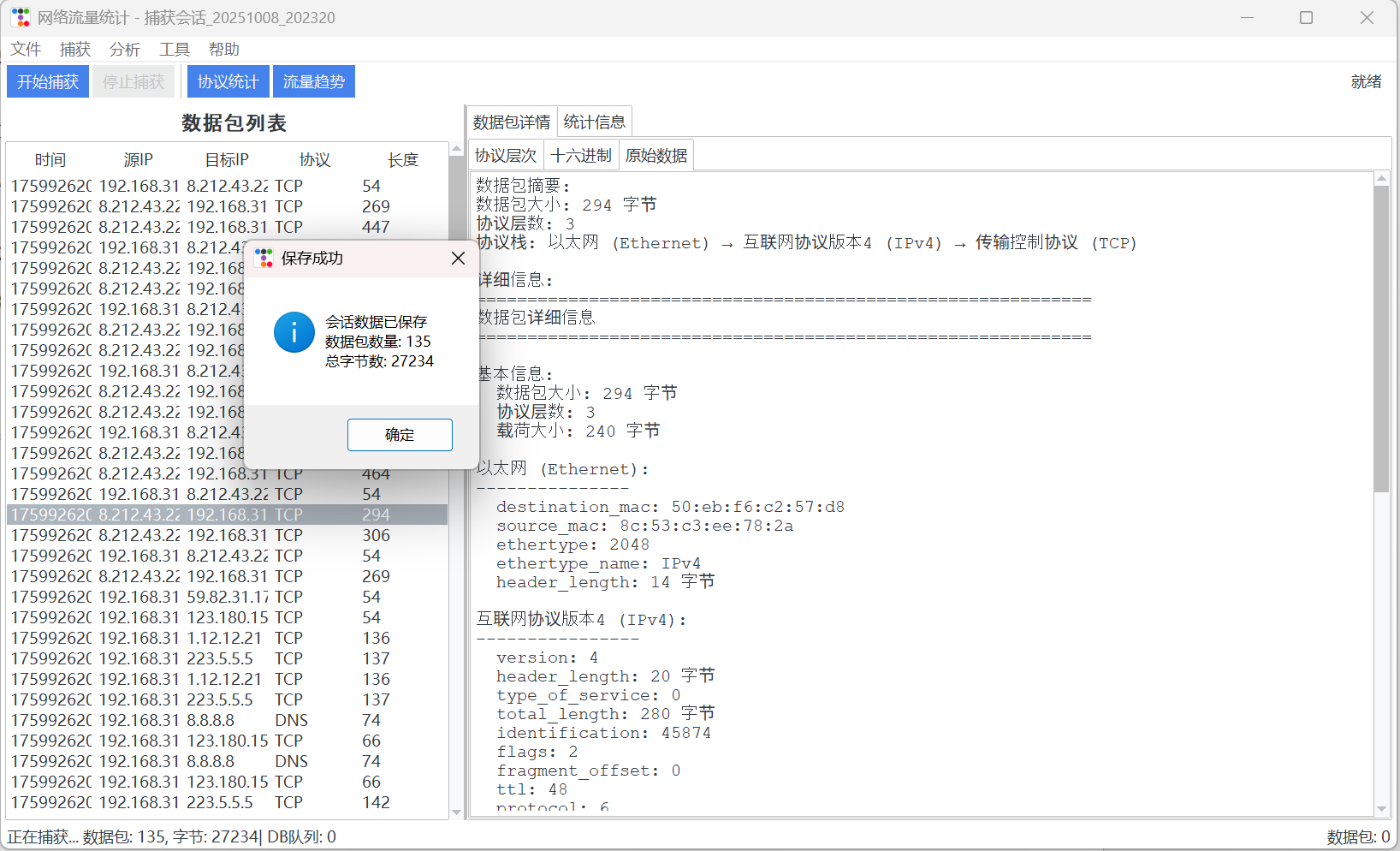






可以正常分析出数据包协议层次、十六进制数据、原始数据信息

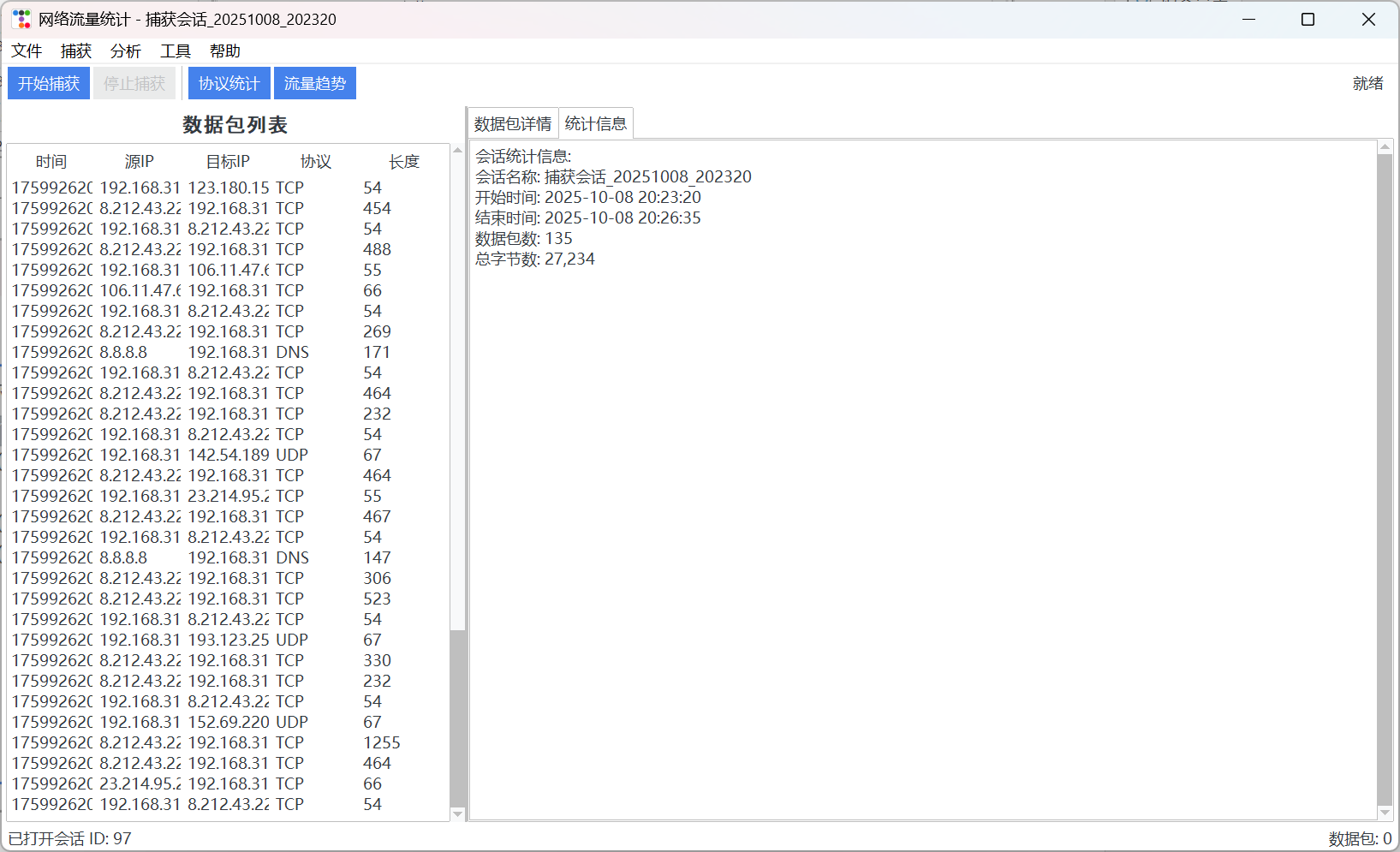
5.3.5 会话管理模块



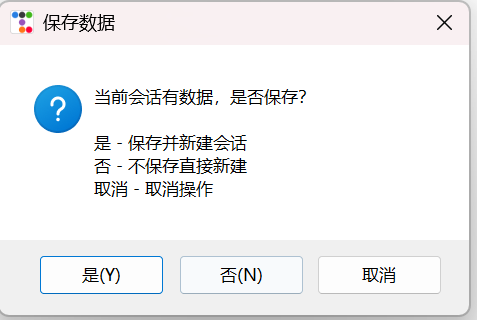
可以正常保存会话

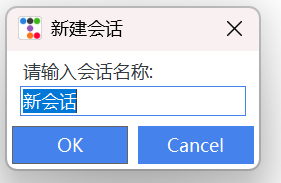


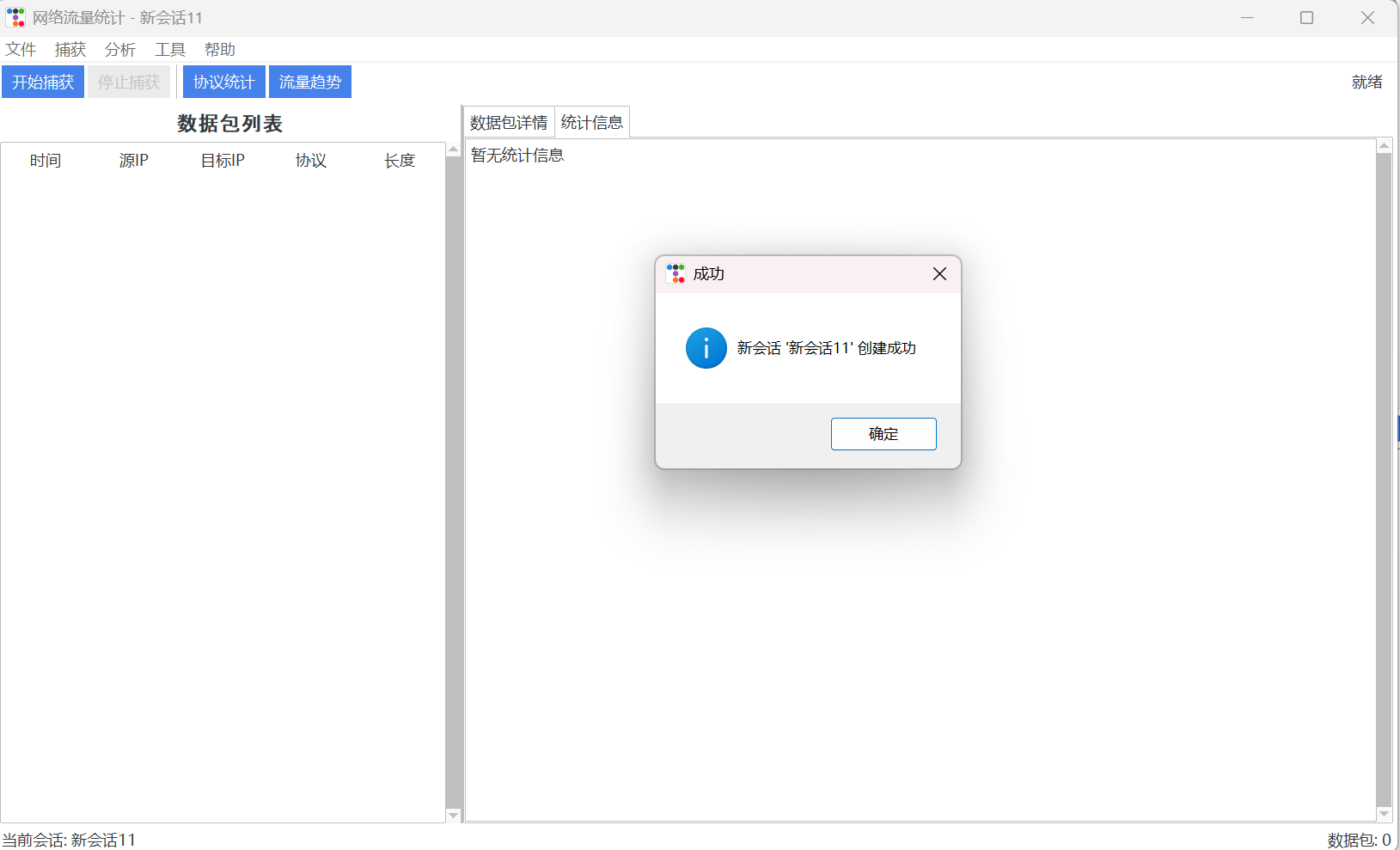
可以正常选择历史会话



可以正常打开历史会话

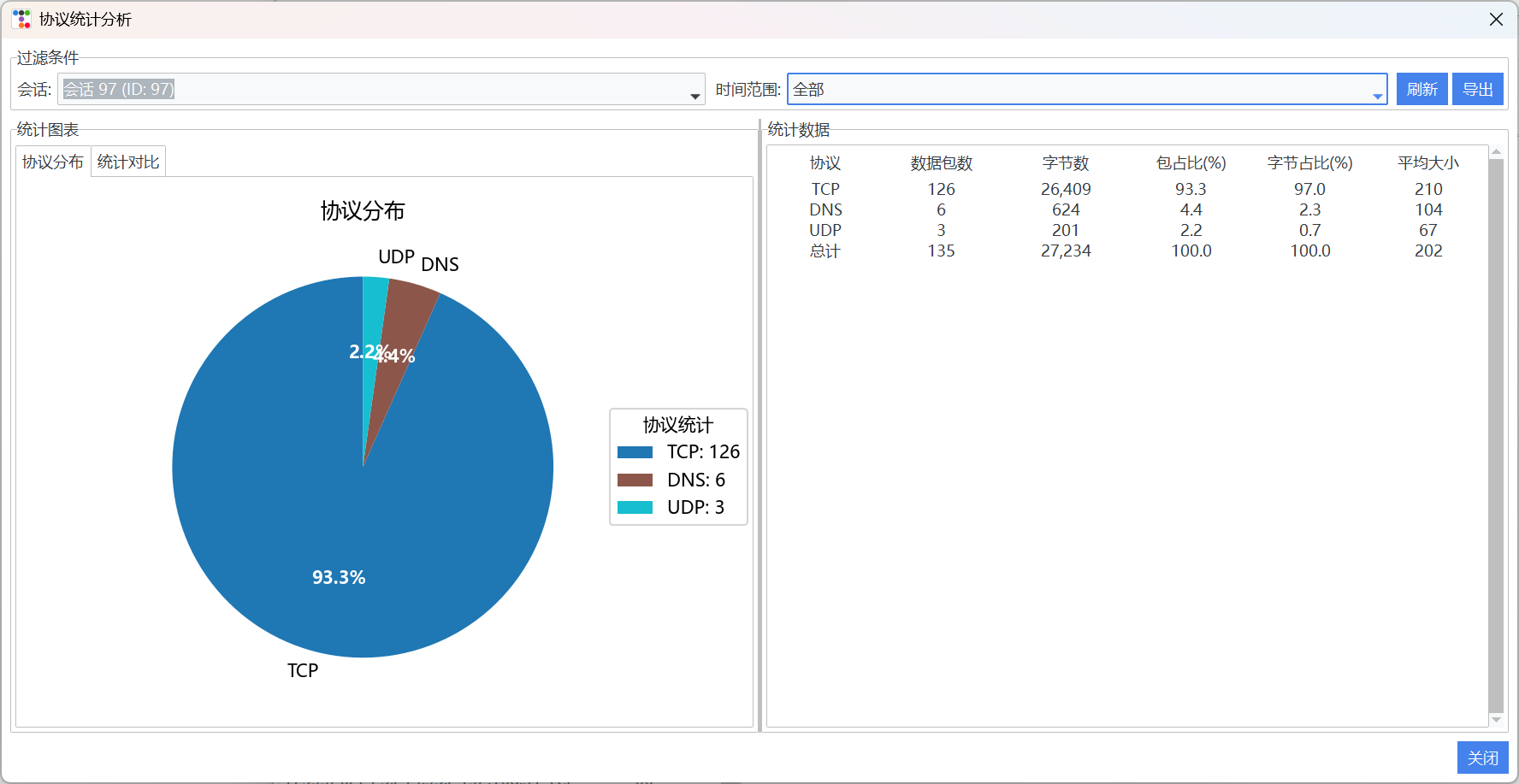


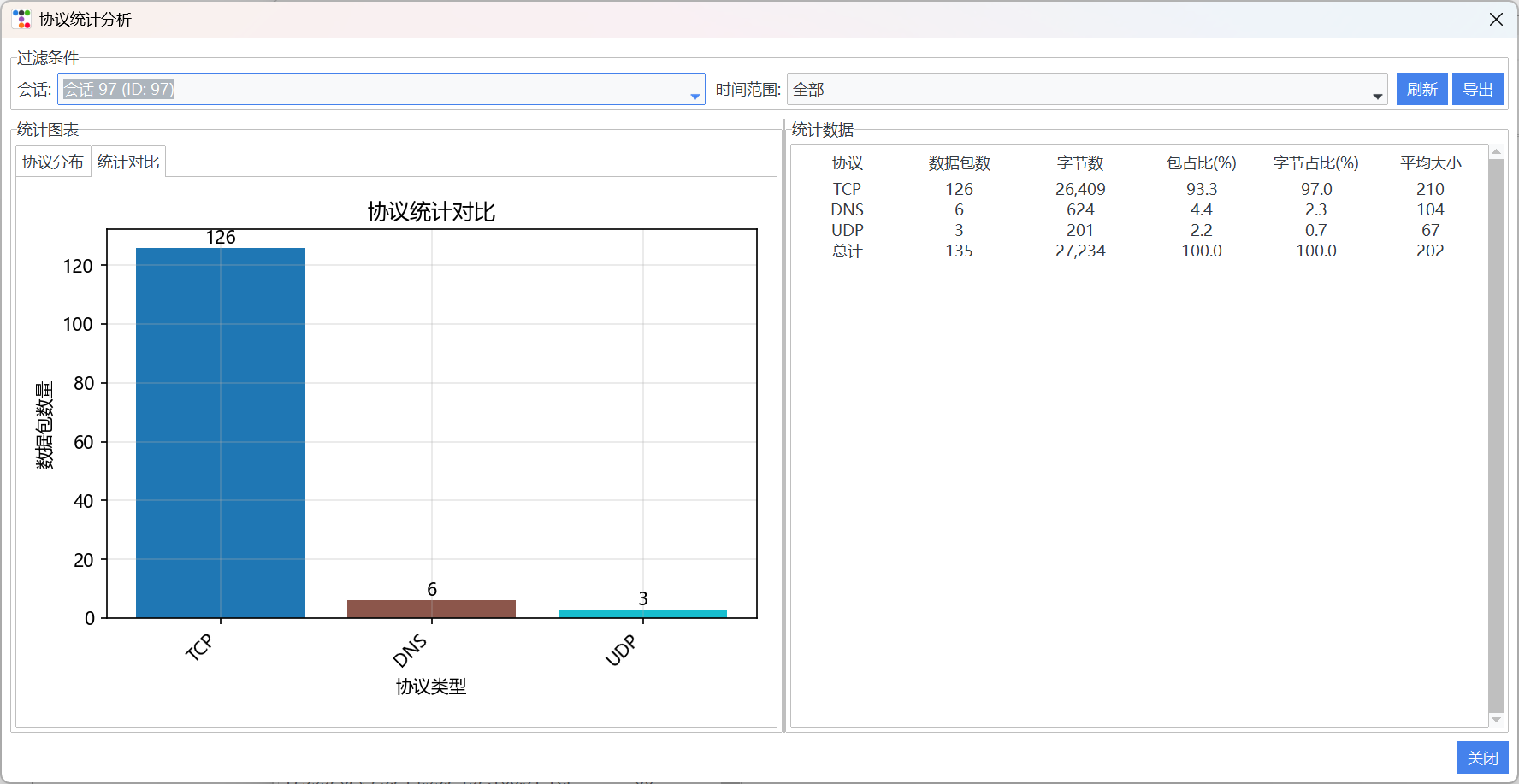




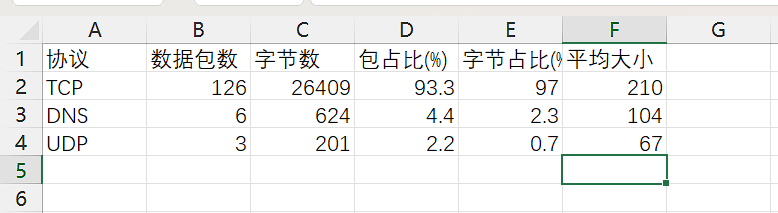
可以正常新建会话

5.3.6 协议统计分析模块



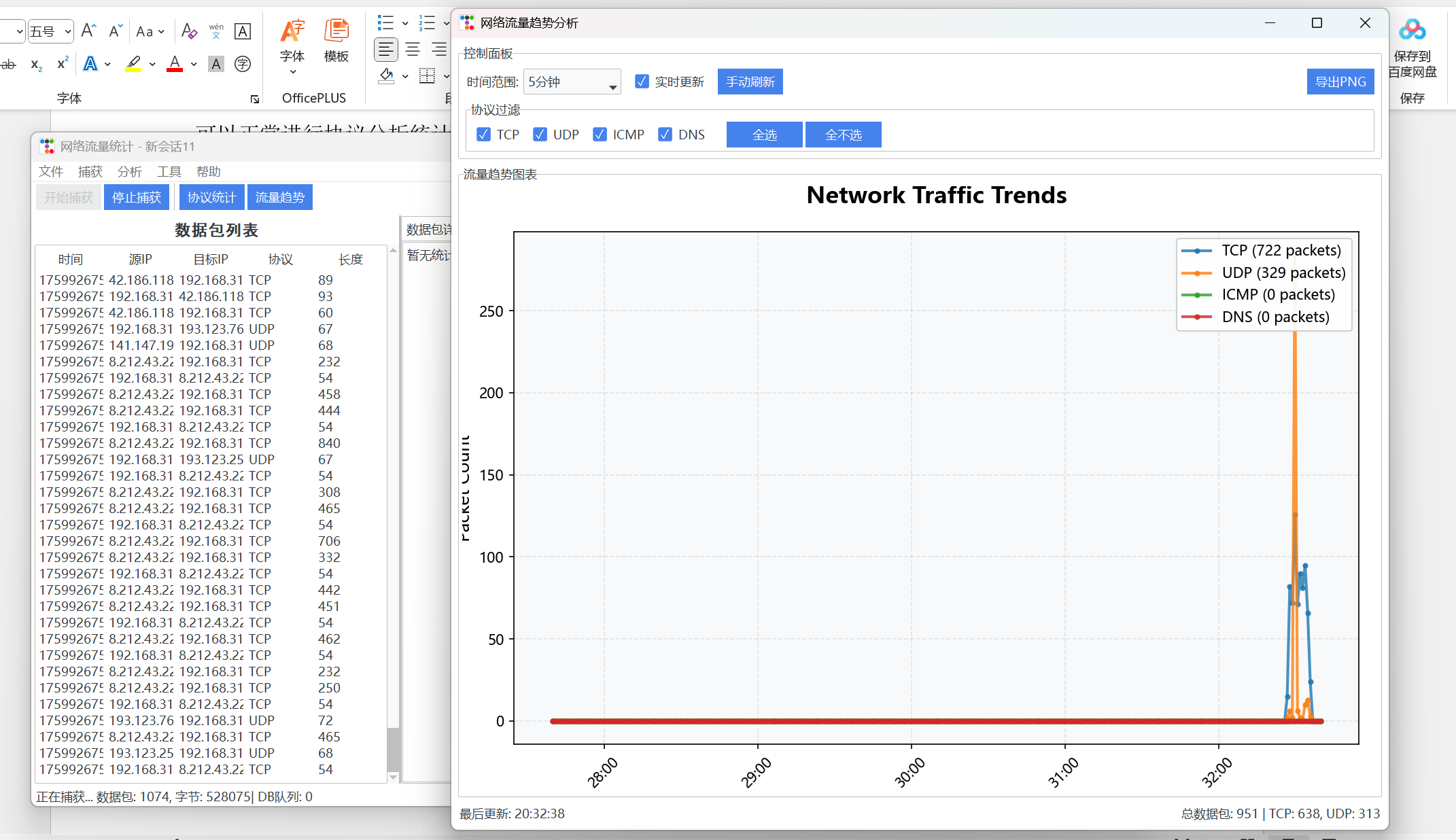


可以正常进行协议分析统计并且进行图表可视化



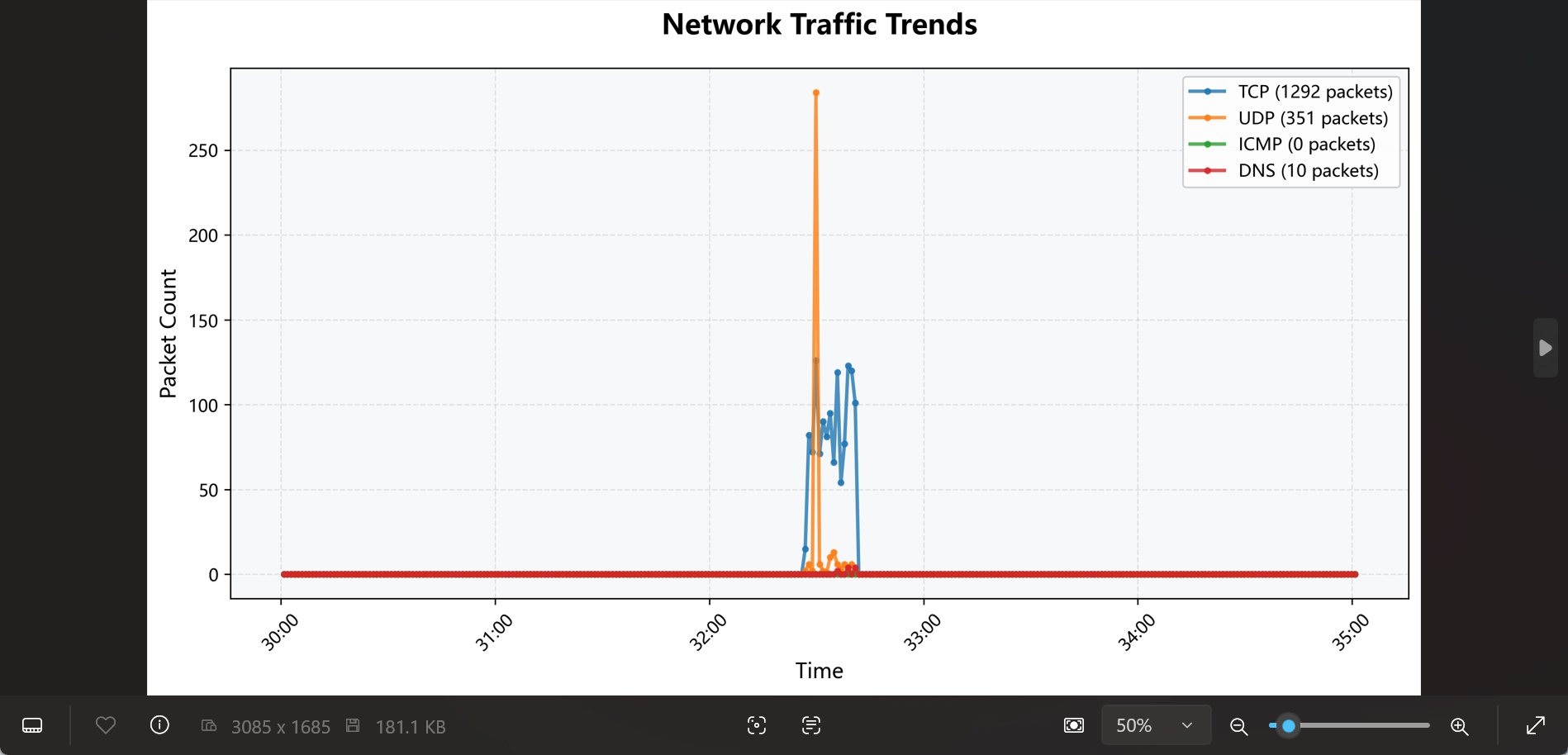
也可以导出协议统计信息到.csv文件查看

5.3.7 流量趋势模块

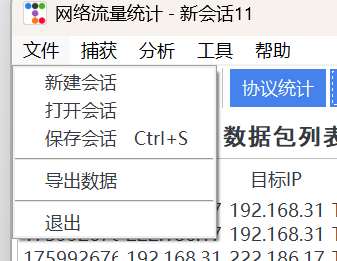


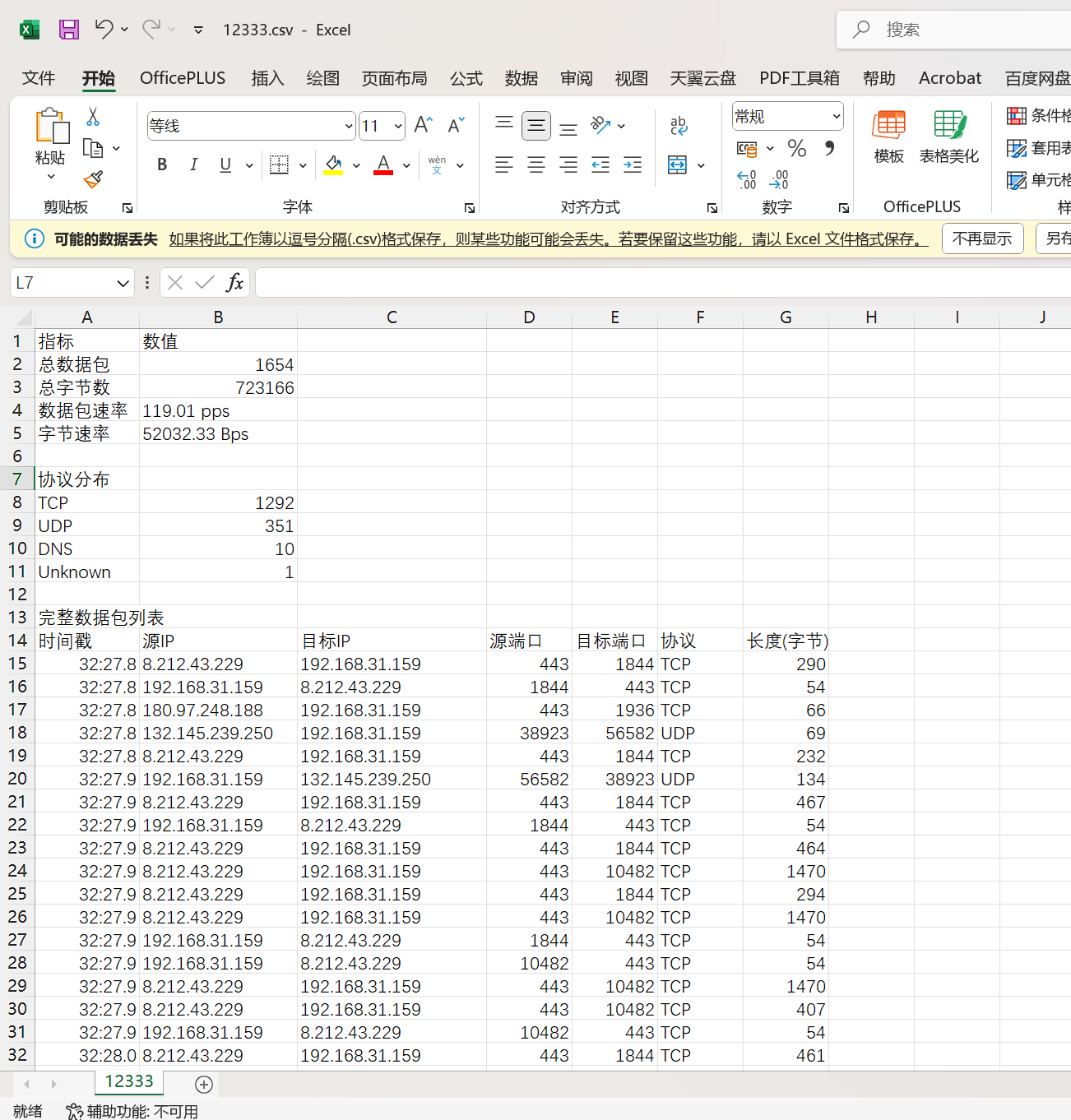
开始捕获后打开流量趋势分析模块可以动态显示出各个协议的趋势变化图表

也可以把图表导出为图片



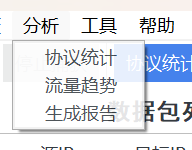
5.3.8 导出数据模块

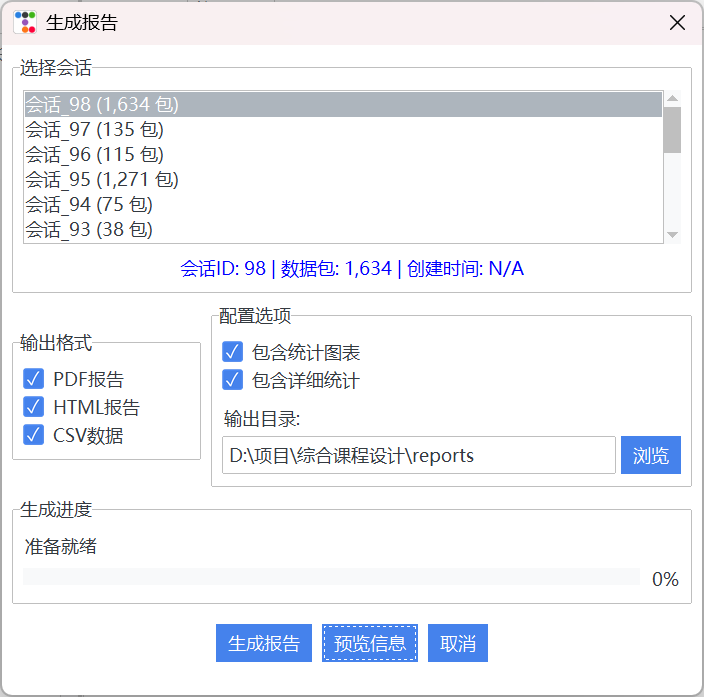


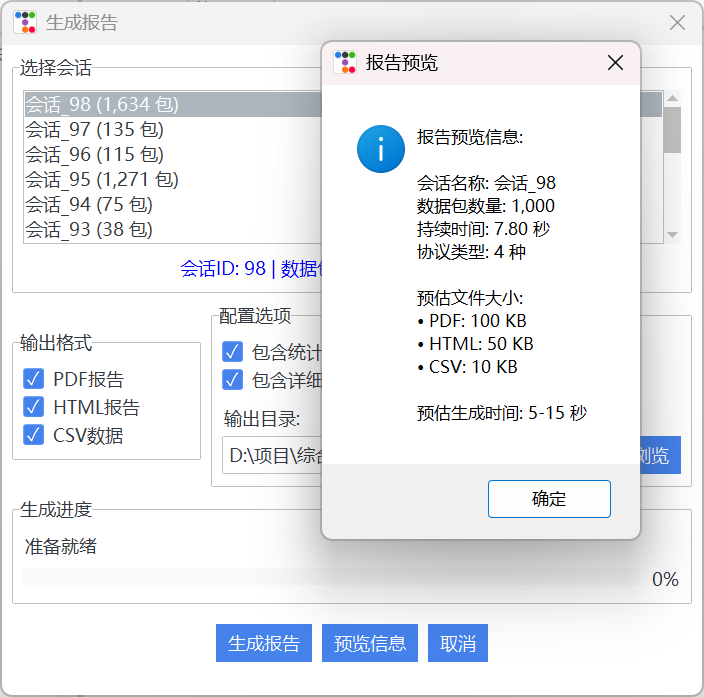


可以导出数据包数据为.csv文件查看

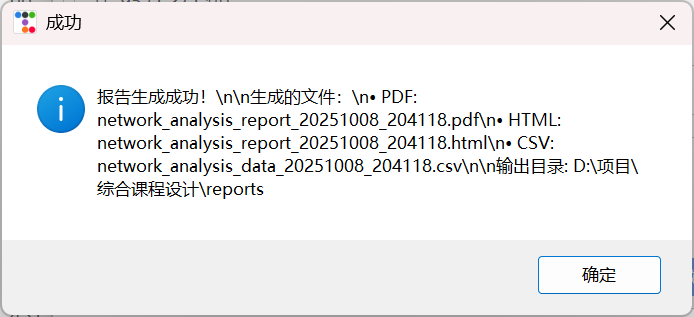
5.3.9 分析报告生成导出模块



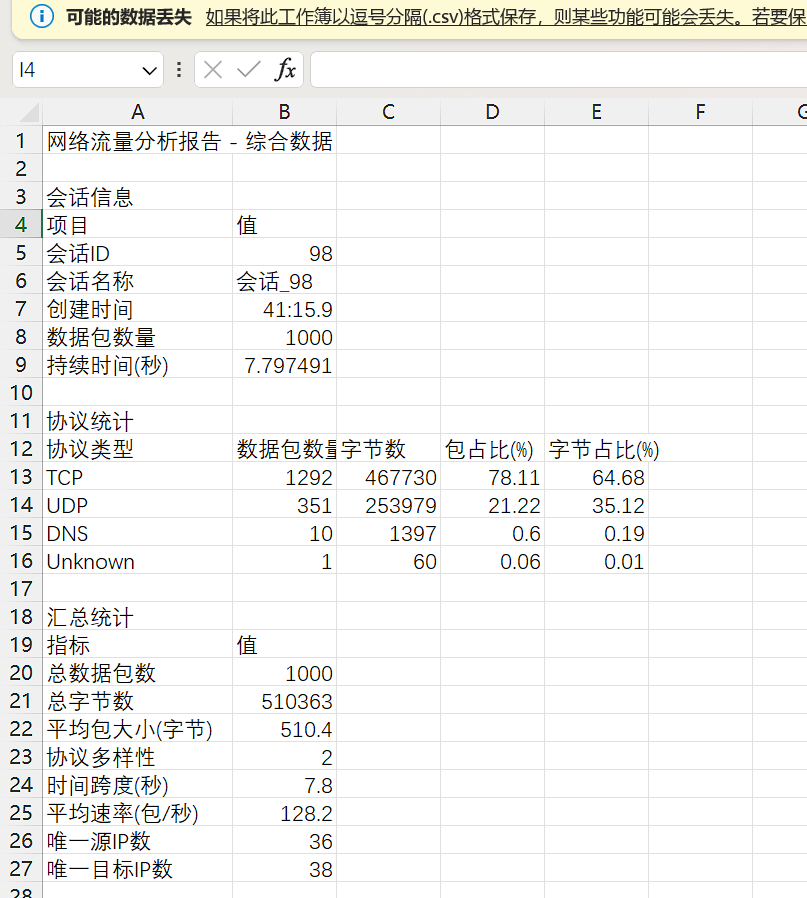




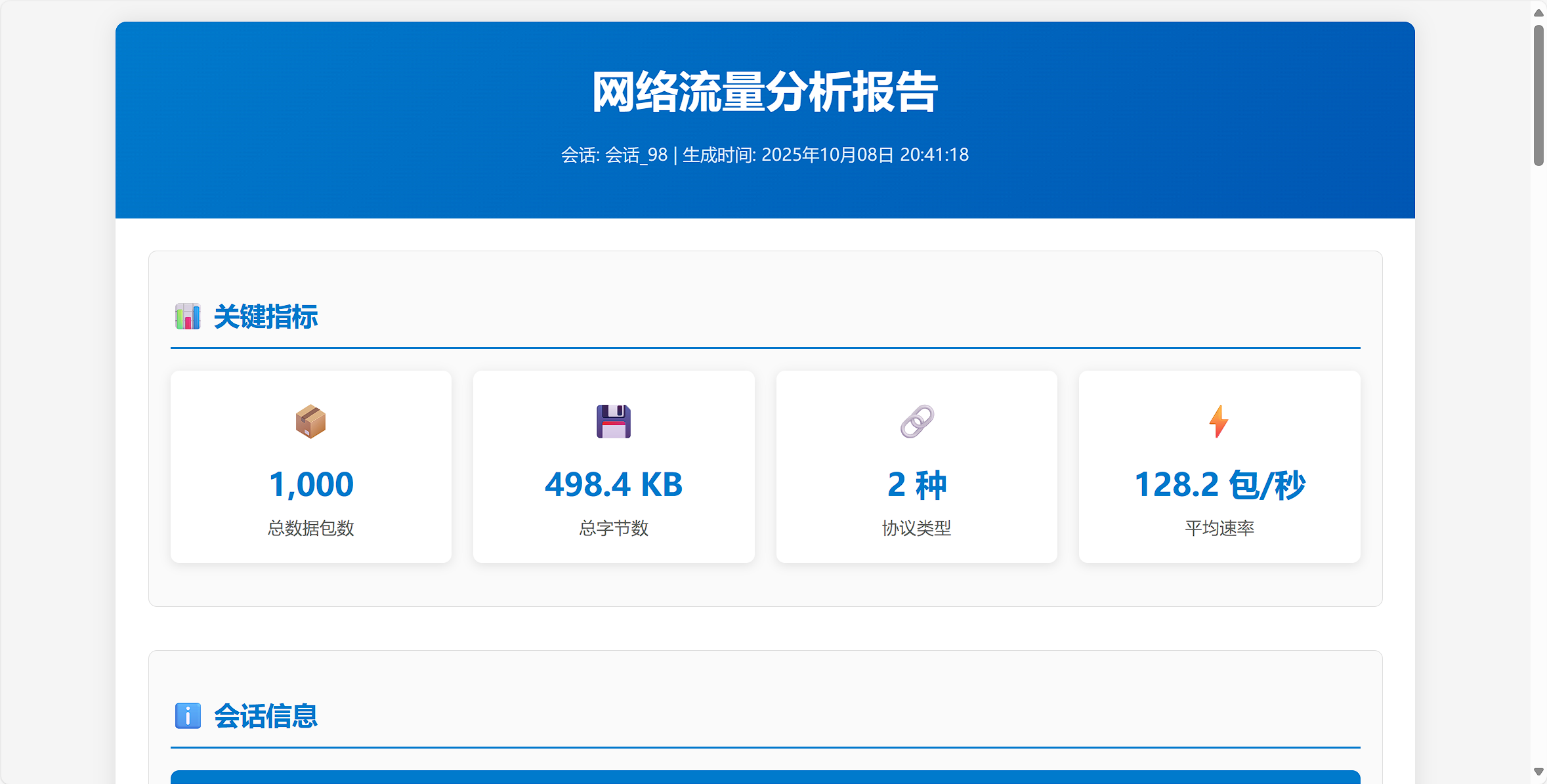
可以选择历史会话，生成报告并导出，可以选择pdf、html、csv格式查看



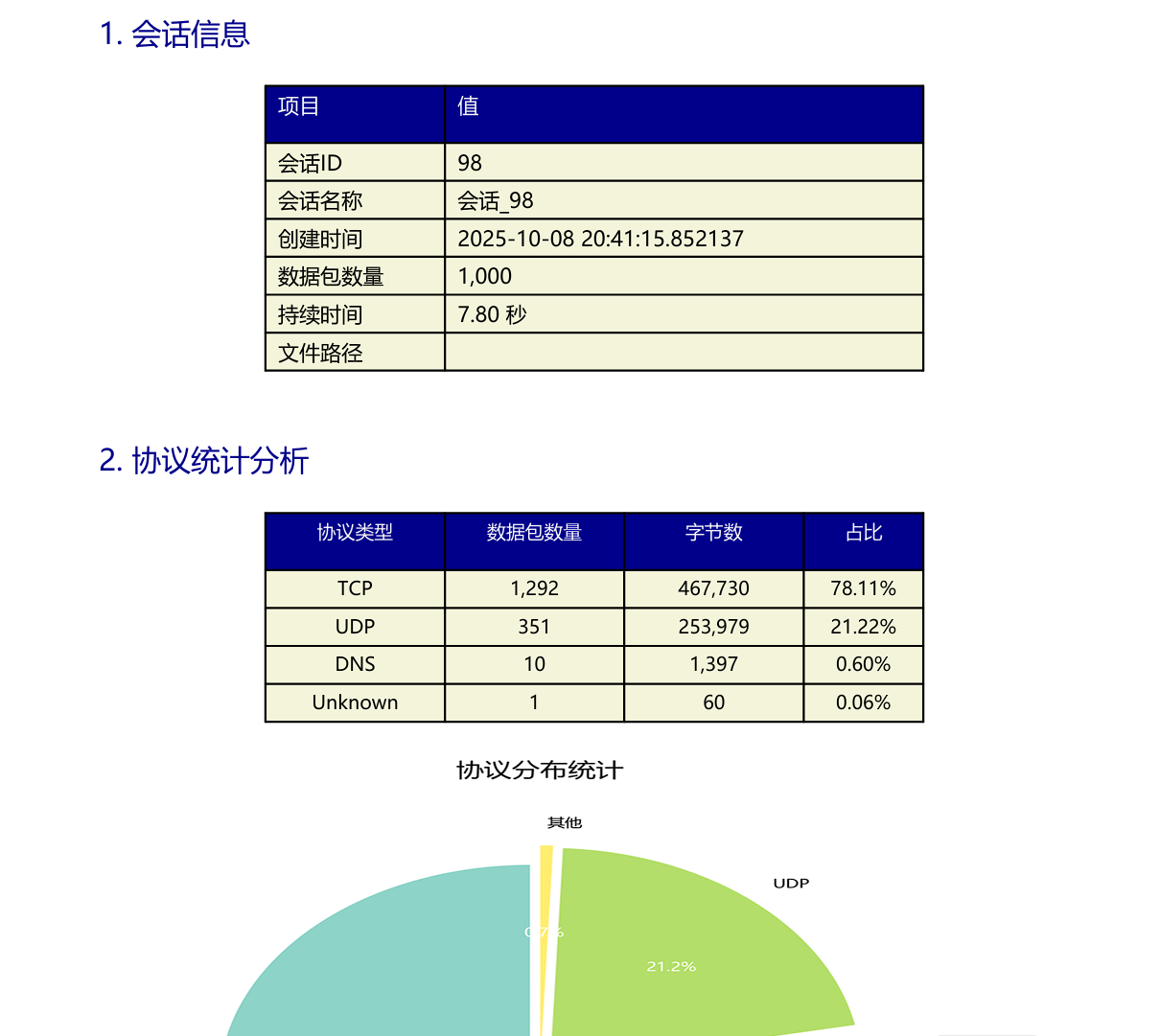
csv文件：



HTML文件：

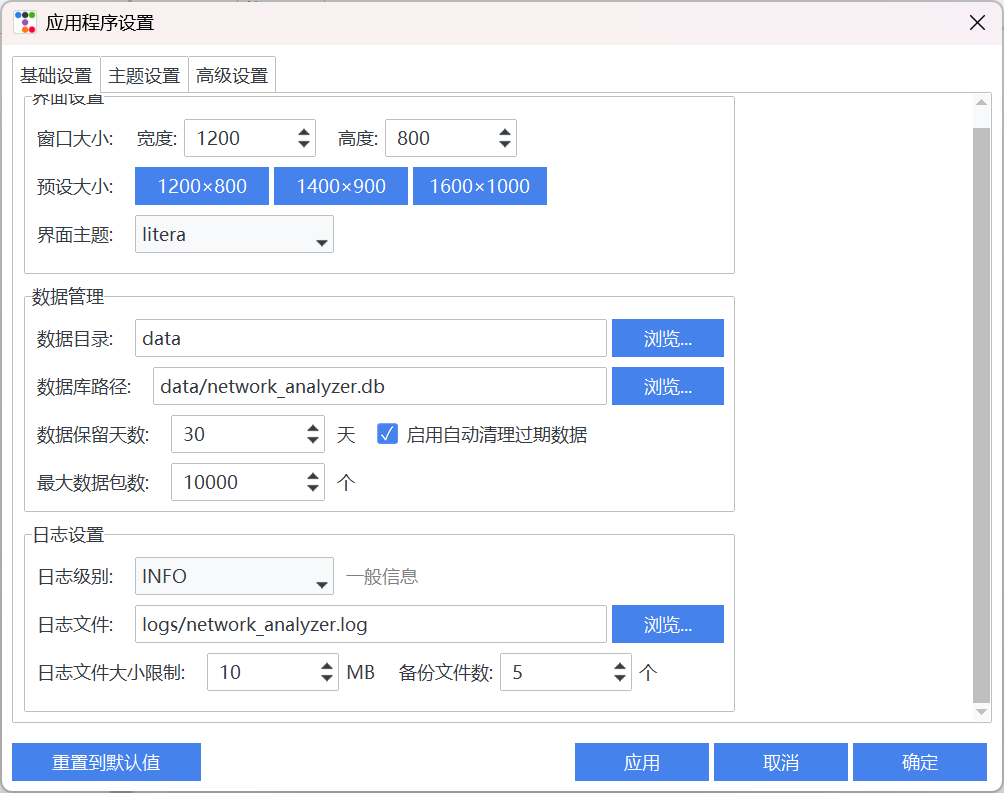


PDF文件：



5.3.10 设置模块

设置模块可以进行一些基础的设置，比如数据库路径目录、窗口大小、日志设置

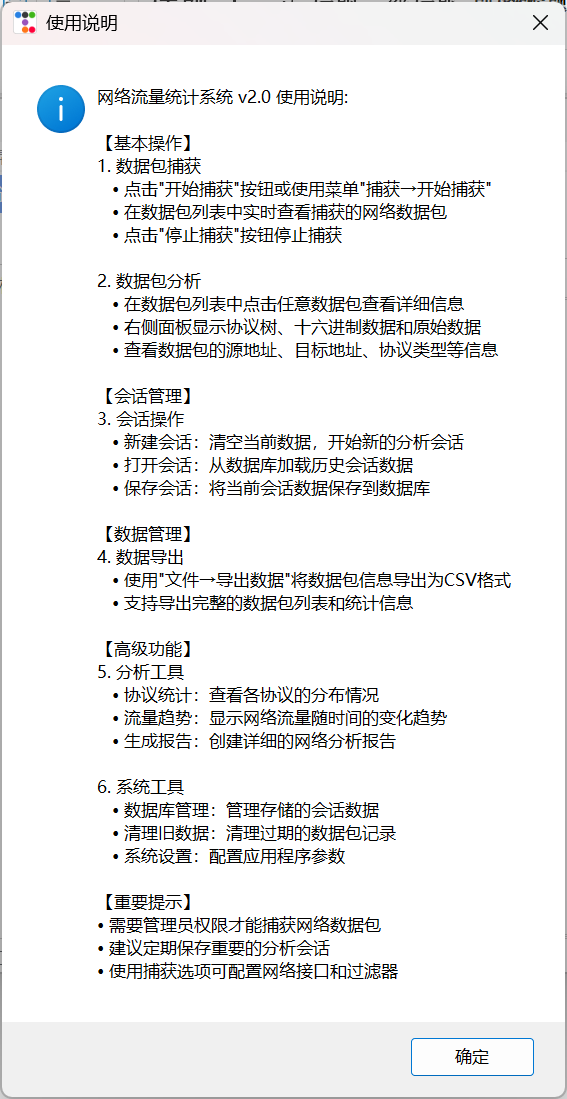


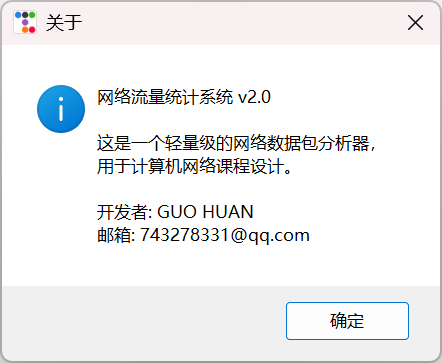
在主题设置界面也可以切换不同颜色的主题



5.3.11 帮助模块

帮助模块有使用说明和关于信息





# 第六章 结束语

6.1 项目总结

本次课程设计成功开发了一个集网络流量捕获、协议分析、数据统计和报告生成于一体的综合性桌面应用。通过这个项目，我深入实践了计算机网络的核心理论，特别是TCP/IP协议栈的工作原理。在技术实现上，熟练运用了Python及其生态中的多个优秀库，如Scapy的强大抓包能力、Tkinter/ttkbootstrap的GUI快速开发特性、Matplotlib的数据可视化功能以及SQLite的轻量级数据持久化方案。项目遵循了模块化的软件设计思想，实现了清晰的分层架构，提高了代码的可读性和可维护性。整个开发过程，从需求分析到设计、编码、测试，完整地走了一遍软件工程的生命周期，极大地锻炼了我的项目实践能力和问题解决能力。。

6.2 项目创新点

1. **轻量级与易用性的结合**：与Wireshark等功能强大但复杂的专业工具相比，本项目聚焦于核心功能，提供了一个更轻量、更易于上手的替代方案，特别适合教学和快速网络诊断场景。
2. **异步化与批量处理**：在数据处理流水线中引入了异步和批量处理思想，显著降低了高流量下丢包的可能性，保证了数据处理的稳定性和效率。
3. **模块化与可扩展性**：系统采用了清晰的模块化设计（捕获、处理、存储、UI），各模块职责单一、耦合度低。这不仅使得代码易于理解和维护，也为未来新功能的扩展打下了良好的基础。

6.3 不足与展望

尽管当前系统已基本满足设计目标，但仍存在一些可以改进和扩展的空间：

1. **协议解析深度有限**：目前主要依赖Scapy进行协议解析，对于一些协议（如HTTP/2, STMP,IMAP）的解析不够深入，或者无法解析。未来可以引入更专业的协议解析库或自定义解析逻辑，以支持更丰富的协议细节展示。
2. **高级分析功能缺失**：缺少一些高级网络分析功能，如TCP流重组、数据包内容深度搜索、恶意流量检测等。可以在现有框架基础上，增加专门的分析模块来实现这些功能。
3. **性能瓶颈**：高流量（>10kpps）下Queue满丢包。改进方法：分担解析，Redis缓存统计
4. **跨平台兼容性测试有待完善**：目前主要在Windows环境下开发和测试，虽然理论上核心组件是跨平台的，但在Linux和macOS下的兼容性和表现有待进一步的完整测试和适配。
5. **数据库安全机制弱：**无加密存储/访问控制。改进：SQLCipher加密DB，OAuth用户认证。
6. **数据库性能瓶颈：**当一个会话存储几万条以上的数据包数据时，SQLite数据库会发生性能不足的情况，读取或者保存会话的时候会造成界面卡死无响应，滚动查看数据会变得非常卡。