网络协议分析软件设计文档

1. 文档概述

1.1 文档目的

本文档详细描述了网络协议分析软件的设计方案，包括总体架构、功能模块设计、数据结构设计、算法设计等内容。该文档旨在指导软件开发过程，并为后续维护和扩展提供参考依据。

1.2 术语定义

术语	解释
BPF	Berkeley Packet Filter，伯克利数据包过滤器，用于指定数据包捕获的过滤规则
PCAP	Packet Capture，数据包捕获文件格式，用于存储网络数据包
Scapy	一个强大的 Python 网络数据包处理库，用于捕获、发送、分析和构建网络数据包
PyQt5	Python 的 Qt5 绑定库，用于开发跨平台图形界面应用
协议识别	确定网络数据包所属的协议类型（如 TCP、UDP、HTTP 等）
协议解码	解析网络数据包的具体内容，提取协议字段信息

表 1 术语定义

2. 总体架构设计

2.1 架构概述

网络协议分析软件采用模块化、分层架构设计，各模块之间通过明确的接口进行通信，具有良好的可扩展性和可维护性。软件支持命令行和图形界面两种交互方式，可根据用户需求灵活选择。

2.2 系统架构图

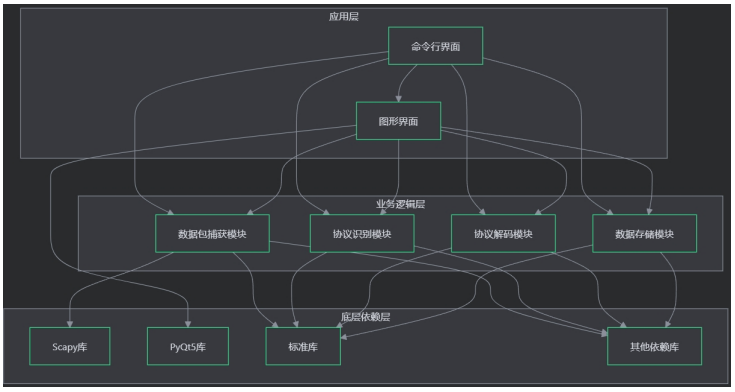

图 1 系统架构

2.3 模块关系图

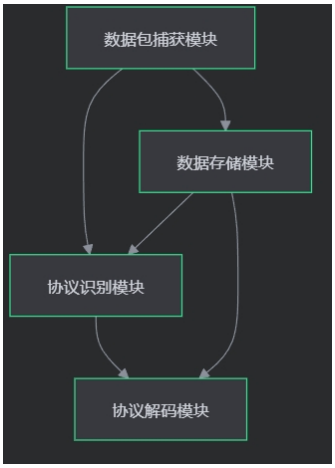

图 2 模块关系

2.3 总运行流程

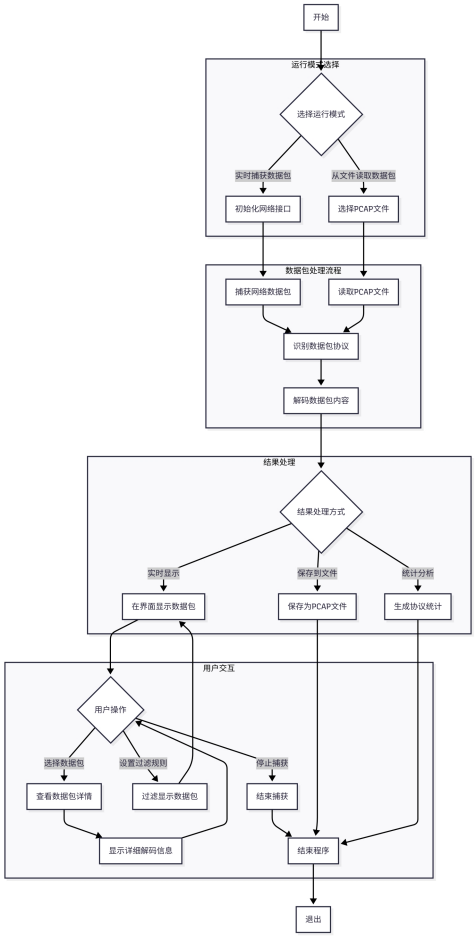

图 3 项目运行

3. 功能模块设计

3.1 数据包捕获模块

3.1.1 模块概述

数据包捕获模块负责从指定网络接口捕获实时数据包，支持 BPF 过滤规则，可配置捕获数量和超时时间。

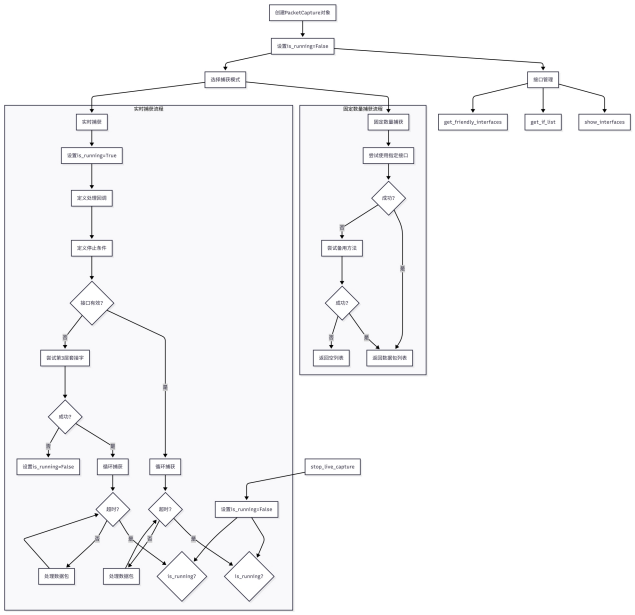

图 4 数据包捕获模块

3.1.2 核心功能

- 获取网络接口列表
- 实时捕获网络数据包
- 支持 BPF 过滤规则
- 提供友好的接口名称映射

3.1.3 类设计

类名	功能描述	核心方法
PacketCapture	数据包捕获主类	show_interfaces()- get_friendly_interfaces()- capture_packets()- start_live_capture()- stop_live_capture()

表 2 数据包捕获类设计

3.1.4 关键方法实现

```
def capture_packets(self, interface, count=10, filter_rule=None,
timeout=None):
    """捕获指定数量的数据包"""
    packets = []
    # 使用 Scapy 的 sniff 函数捕获数据包
    packets = sniff(
        iface=interface,
        count=count,
        filter=filter_rule,
        timeout=timeout,
        store=True
    )
    return packets
```

3.2 协议识别模块

3.2.1 模块概述

协议识别模块负责识别网络数据包所属的协议类型，支持多种常见网络协议（如 TCP、UDP、HTTP、DNS、ICMP 等）。

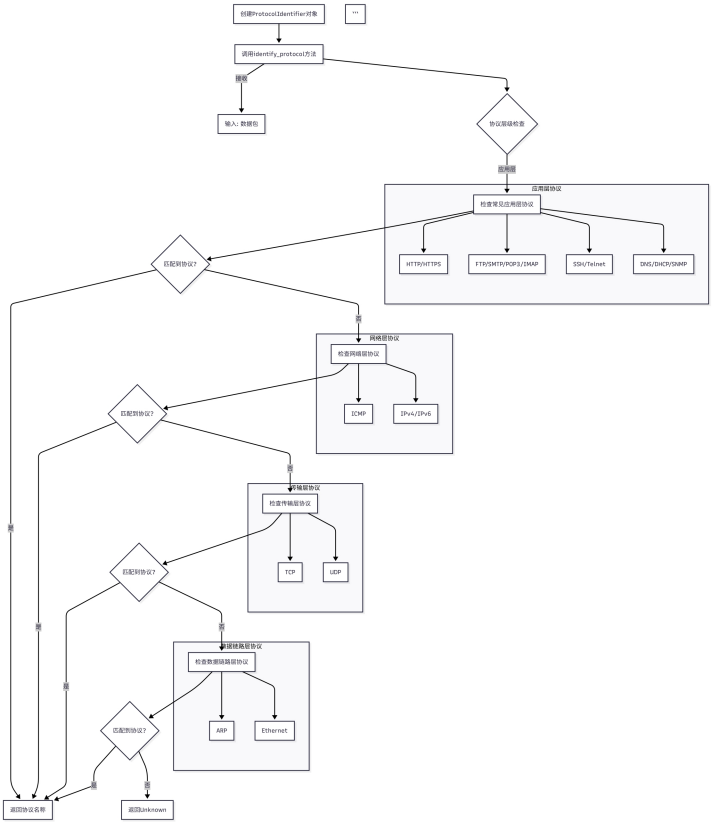

图 5 协议识别模块

3.2.2 核心功能

- 识别数据包的协议类型
- 支持多层协议识别
- 提供协议优先级管理

3.2.3 类设计

类名	功能描述	核心方法
ProtocolIdentifier	协议识别主类	identify_protocol()- _check_transport_layer()- _check_application_layer()

表 3 协议识别类设计

3.2.4 关键方法实现

```
def identify_protocol(self, packet):
    """识别数据包的协议类型"""
    # 检查数据链路层
    if packet.haslayer(Ether):
        # 检查网络层
        if packet.haslayer(IP):
            # 检查传输层
            protocol = self._check_transport_layer(packet)
            # 检查应用层
            app_protocol = self._check_application_layer(packet)
            return app_protocol if app_protocol else protocol
        elif packet.haslayer(IPv6):
            # IPv6 处理逻辑
            protocol = self._check_transport_layer(packet)
            app_protocol = self._check_application_layer(packet)
            return app_protocol if app_protocol else protocol
        elif packet.haslayer(ARP):
            return "ARP"
    return "Unknown"
```

3.3 协议解码模块

3.3.1 模块概述

协议解码模块负责解析网络数据包的具体内容，提取协议字段信息，并以可读格式输出。

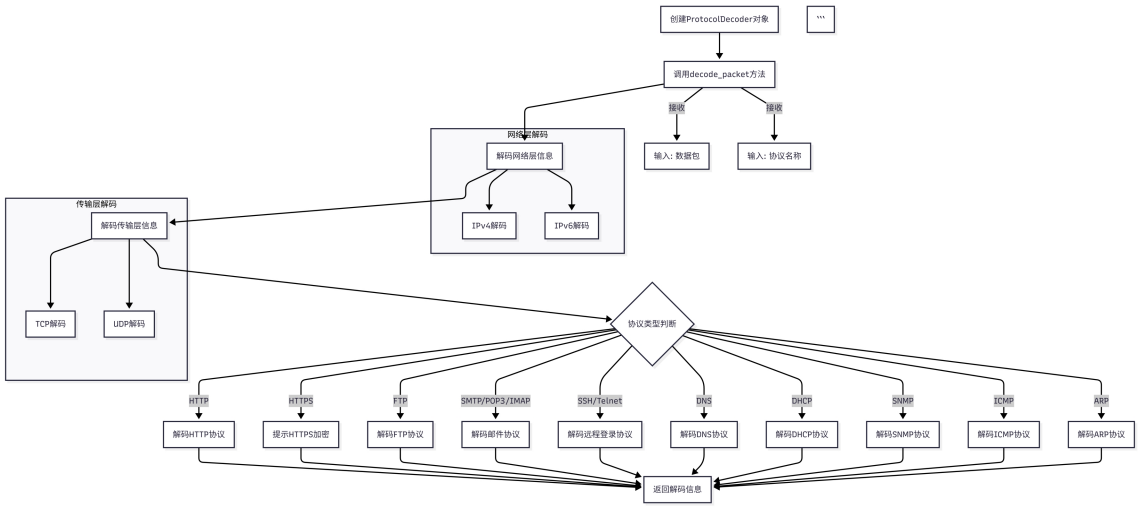

图 6 协议解码模块

3.3.2 核心功能

- 解码多种网络协议
- 提取协议字段信息
- 格式化输出解码结果
- 支持嵌套协议解码

3.3.3 类设计

类名	功能描述	核心方法
ProtocolDecoder	协议解码主类	decode_packet()- decode_ether()- decode_ip()- decode_tcp()- decode_udp()- decode_http()- decode_dns()

表 4 协议解码类设计

3.3.4 关键方法实现

```
def decode_packet(self, packet, protocol):
    """解码数据包"""
    result = []

    # 解码数据链路层
    if packet.haslayer(Ether):
        result.append("=== 数据链路层 (Ethernet) ===")
        result.append(self.decode_ether(packet))
```

```
        # 解码网络层
        if packet.haslayer(IP):
            result.append("\n=== 网络层 (IP) ===")
            result.append(self.decode_ip(packet))
        elif packet.haslayer(IPv6):
            result.append("\n=== 网络层 (IPv6) ===")
            result.append(self.decode_ipv6(packet))
        elif packet.haslayer(ARP):
            result.append("\n=== 网络层 (ARP) ===")
            result.append(self.decode_arp(packet))

        # 解码传输层
        if packet.haslayer(TCP):
            result.append("\n=== 传输层 (TCP) ===")
            result.append(self.decode_tcp(packet))
        elif packet.haslayer(UDP):
            result.append("\n=== 传输层 (UDP) ===")
            result.append(self.decode_udp(packet))
        elif packet.haslayer(ICMP):
            result.append("\n=== 传输层 (ICMP) ===")
            result.append(self.decode_icmp(packet))

        # 解码应用层
        if protocol == "HTTP":
            result.append("\n=== 应用层 (HTTP) ===")
            result.append(self.decode_http(packet))
        elif protocol == "DNS":
            result.append("\n=== 应用层 (DNS) ===")
            result.append(self.decode_dns(packet))

        return "\n".join(result)
```

3.4 数据存储模块

3.4.1 模块概述

数据存储模块负责将捕获的数据包保存为 PCAP 文件，以及从 PCAP 文件中读取数据包进行分析。

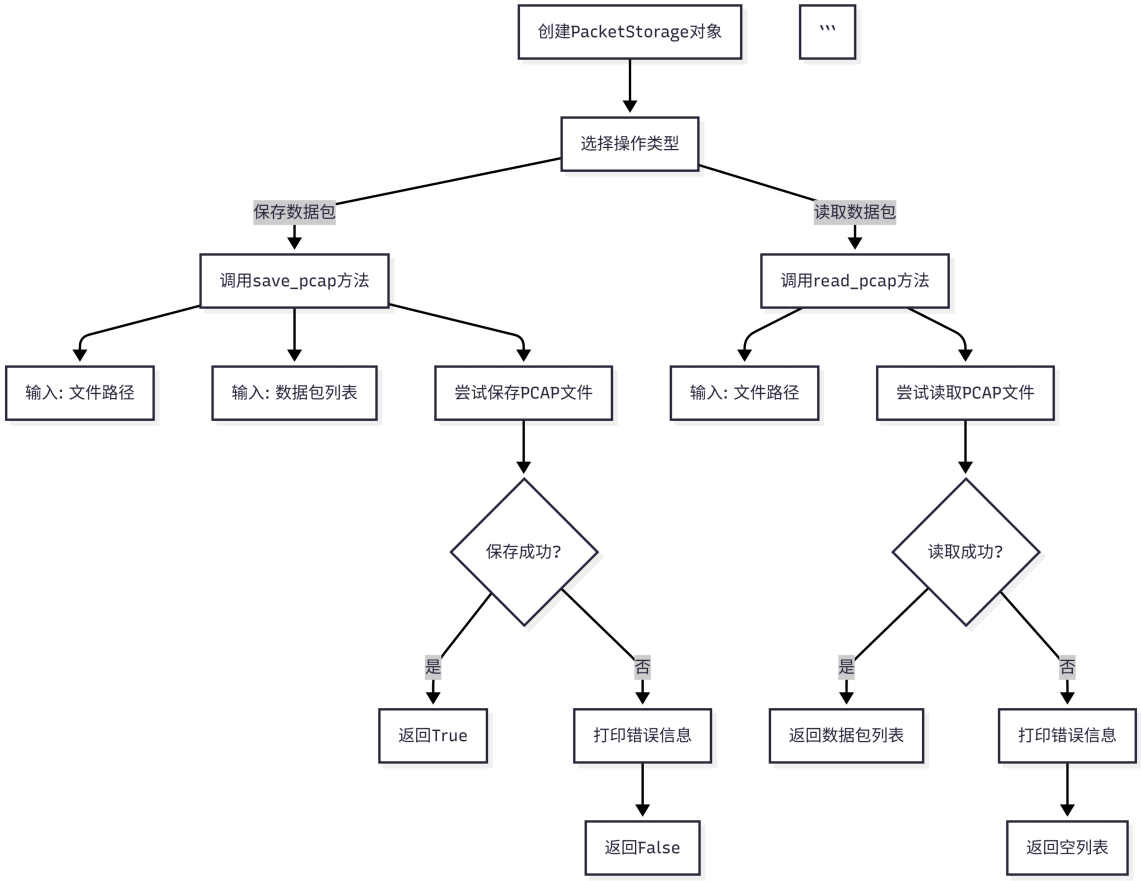

图 7 数据存储模块

3.4.2 核心功能

- 保存数据包为 PCAP 文件
- 从 PCAP 文件读取数据包
- 支持多种 PCAP 文件格式
- 提供文件校验和错误处理

3.4.3 类设计

类名	功能描述	核心方法
PacketStorage	数据存储主类	save_pcap() read_pcap()

表 5 数据存储类设计

3.4.4 关键方法实现

```
def save_pcap(self, file_path, packets):
    """保存数据包为 PCAP 文件"""
```

```
        try:
            wrpcap(file_path, packets)
            return True
        except Exception as e:
            print(f"保存 PCAP 文件失败: {e}")
            return False

    def read_pcap(self, file_path):
        """从 PCAP 文件读取数据包"""
        try:
            packets = rdpcap(file_path)
            return packets
        except Exception as e:
            print(f"读取 PCAP 文件失败: {e}")
            return []
```

3.5 命令行界面

3.5.1 模块概述

命令行界面模块提供了基于命令行的交互方式，支持各种参数配置和结果输出。

3.5.2 核心功能

- 解析命令行参数
- 调用各功能模块
- 格式化输出结果
- 支持批处理操作

3.5.3 实现方式

通过 argparse 库实现命令行参数解析，支持以下参数：

- -i, --interface: 指定网络接口
- -c, --count: 捕获数据包数量
- -f, --filter: BPF 过滤规则
- -o, --output: 输出 PCAP 文件路径
- -r, --read: 读取 PCAP 文件进行分析

3.6 图形界面模块

3.6.1 模块概述

图形界面模块基于 PyQt5 开发，提供可视化的操作界面，支持实时数据包捕获、协议分析、统计展示等功能。

3.6.2 核心功能

- 网络接口选择
- 实时数据包捕获与显示
- 协议统计与可视化
- 数据包详情查看
- 主题切换（浅色/深色）
- PCAP 文件导入/导出

3.6.3 类设计

类名	功能描述	核心方法
ProtocolAnalyzerGUI	主界面类	init_ui()- create_toolbar()- create_control_panel()- create_packet_list_tab()- create_statistics_tab()- create_interface_status_tab()- on_packet_captured()- on_packet_selected()
CaptureThread	捕获线程类	run() stop()

表 6 GUI 类设计

4. 数据结构设计

4.1 数据包数据结构

```
# 数据包列表，使用 Scapy 的 Sniffed 对象存储
packets = Sniffed(
    sessions={},
    comment='',
    filename='',
    raw_mode=0,
    *args, **kargs
)

# 单个数据包对象（Scapy 的 Packet 对象）
packet = Ether() / IP() / TCP() / "HTTP/1.1 GET
/index.html\r\nHost: example.com\r\n\r\n"
```

4.2 协议统计数据结构

```
# 协议统计字典
protocol_counts = {
    "TCP": 100,
    "UDP": 50,
    "HTTP": 30,
    "DNS": 20,
    "ARP": 5,
    "ICMP": 3
}

# 数据包列表，用于GUI显示
captured_packets = [
    {
        "id": 1,
        "time": "0.000000",
        "source": "192.168.1.1",
        "destination": "192.168.1.2",
        "protocol": "TCP",
        "length": 60,
        "info": "SYN"
    },
    # 更多数据包...
]
```

4.3 主题配置数据结构

```
themes = {
    "light": {
        "background": QColor(255, 255, 255),
        "text": QColor(0, 0, 0),
        "frame": QColor(240, 240, 240),
        "button": QColor(220, 220, 220),
        "button_text": QColor(0, 0, 0),
        "text_edit": QColor(255, 255, 255),
        "text_edit_text": QColor(0, 0, 0),
        "table_header": QColor(220, 220, 220),
        "table_alternate": QColor(240, 240, 240),
    },
    "dark": {
        "background": QColor(45, 45, 45),
        "text": QColor(255, 255, 255),
        "frame": QColor(61, 61, 61),
        "button": QColor(77, 77, 77),
        "button_text": QColor(255, 255, 255),
        "text_edit": QColor(45, 45, 45),
        "text_edit_text": QColor(255, 255, 255),
        "table_header": QColor(77, 77, 77),
        "table_alternate": QColor(55, 55, 55),
    }
}
```

5. 算法设计

5.1 协议识别算法

5.1.1 算法概述

采用分层协议识别算法，从数据链路层开始，逐层向上识别协议类型。对于应用层协议，根据特定协议的特征进行识别。

5.1.2 算法流程

1. 检查数据链路层协议
   a. 如果是 Ethernet，继续
   b. 否则返回"Unknown"

2. 检查网络层协议
   a. 如果是 IP，继续
   b. 否则如果是 IPv6，继续
   c. 否则如果是 ARP，返回"ARP"
   d. 否则返回"Unknown"

3. 检查传输层协议
   a. 如果是 TCP，继续
   b. 否则如果是 UDP，继续
   c. 否则如果是 ICMP，返回"ICMP"
   d. 否则返回传输层协议名称

4. 检查应用层协议
   a. 如果是 HTTP，返回"HTTP"
   b. 否则如果是 DNS，返回"DNS"
   c. 否则如果是 DHCP，返回"DHCP"
   d. 否则返回传输层协议名称

5.1.3 应用层协议识别特征

协议	识别特征
HTTP	TCP 端口 80 或 443，且数据载荷包含 HTTP 方法或状态码
DNS	UDP 或 TCP 端口 53，且数据载荷符合 DNS 报文格式
DHCP	UDP 端口 67 或 68，且数据载荷符合 DHCP 报文格式
FTP	TCP 端口 21，且数据载荷包含 FTP 命令或响应
SMTP	TCP 端口 25，且数据载荷包含 SMTP 命令或响应

表 7 应用层协议特征

5.2 数据包过滤算法

5.2.1 算法概述

基于 BPF 过滤规则，使用 Scapy 库的内置过滤功能实现数据包过滤。

5.2.2 实现方式

```
def capture_packets(self, interface, count=10, filter_rule=None,
timeout=None):
    # 使用 Scapy 的 sniff 函数，通过 filter 参数实现 BPF 过滤
    packets = sniff(
        iface=interface,
        count=count,
        filter=filter_rule,
        timeout=timeout,
        store=True
    )
    return packets
```

5.3 数据包统计算法

5.3.1 算法概述

遍历所有捕获的数据包，统计各协议类型的数量，并计算占比。

5.3.2 算法流程

1. 初始化协议统计字典 protocol_counts
2. 遍历所有捕获的数据包：
   a. 识别每个数据包的协议类型
   b. 在 protocol_counts 中更新对应协议的计数
3. 计算每个协议的占比
4. 返回统计结果

6. 接口设计

6.1 模块间接口

6.1.1 数据包捕获模块接口

方法	参数	返回值	描述
show_interfaces()	无	None	打印可用网络接口列表
get_friendly_interfaces()	无	dict	返回友好接口名称到实际接口名称的映射
capture_packets(interface, count=10, filter_rule=None, timeout=None)	interface: strcount: intfilter_rule: strtimeout: int	Sniffed	捕获指定数量的数据包
start_live_capture(interface, callback, filter_rule=None)	interface: strcallback: functionfilter_rule: str	None	开始实时捕获数据包，通过回调函数返回捕获的数据包
stop_live_capture()	无	None	停止实时捕获数据包

表 8 数据包捕获模块接口

6.1.2 协议识别模块接口

方法	参数	返回值	描述
identify_protocol(packet)	packet: Packet	str	识别数据包的协议类型

表 9 协议识别模块接口

6.1.3 协议解码模块接口

方法	参数	返回值	描述
decode_packet(packet, protocol)	packet: Packetprotocol: str	str	解码数据包，返回可读格式的解码结果

表 10 协议解码模块接口

6.1.4 数据存储模块接口

方法	参数	返回值	描述
save_pcap(file_path, packets)	file_path: strpackets: Sniffed	bool	保存数据包为 PCAP 文件，成功返回 True，失败返回 False
read_pcap(file_path)	file_path: str	Sniffed	从 PCAP 文件读取数据包

表 11 数据存储模块接口

6.2 外部接口

6.2.1 命令行接口

```
# 基本用法
python main.py -i <interface> -c <count> -f <filter> -o <output>

# 读取 PCAP 文件
```

```
python main.py -r <pcap_file>

# 示例
python main.py -i "Ethernet 2" -c 100 -f "tcp port 80" -o
capture.pcap
```

6.2.2 图形界面接口

- **菜单栏**：包含文件、捕获、分析、视图、帮助等菜单
- **工具栏**：提供常用操作的快捷按钮
- **控制面板**：用于配置捕获参数
- **数据包列表**：显示捕获的数据包摘要信息
- **数据包详情**：显示选中数据包的详细解码信息
- **统计视图**：显示协议分布统计信息

7. 实现细节

7.1 实时捕获实现

```
# 在图形界面中使用 QThread 实现实时捕获
class CaptureThread(QThread):
    packet_captured = pyqtSignal(object)
    capture_stopped = pyqtSignal()
    capture_error = pyqtSignal(str)

    def run(self):
        try:
            def callback(packet):
                if self.is_running:
                    self.packet_captured.emit(packet)

            self.capture.start_live_capture(
                self.interface,
                callback,
                self.filter_rule
            )
            self.capture_stopped.emit()
        except Exception as e:
            self.capture_error.emit(str(e))
```

7.2 主题切换实现

```
def init_theme(self):
    """初始化主题"""
    palette = QPalette()
```

```
    palette.setColor(QPalette.Window,
self.themes[self.current_theme]["background"])
    palette.setColor(QPalette.WindowText,
self.themes[self.current_theme]["text"])
    palette.setColor(QPalette.Base,
self.themes[self.current_theme]["text_edit"])
    palette.setColor(QPalette.Text,
self.themes[self.current_theme]["text_edit_text"])
    palette.setColor(QPalette.Button,
self.themes[self.current_theme]["button"])
    palette.setColor(QPalette.ButtonText,
self.themes[self.current_theme]["button_text"])
    self.setPalette(palette)

def toggle_theme(self):
    """切换主题"""
    self.current_theme = "dark" if self.current_theme == "light"
else "light"
    self.init_theme()
    self.apply_theme_to_widgets()
```

7.3 数据包列表显示实现

```
def on_packet_captured(self, packet):
    """捕获到数据包时的处理"""
    # 识别协议
    protocol = self.identifier.identify_protocol(packet)

    # 更新协议统计
    self.update_protocol_stats(protocol)

    # 获取基本信息
    source = ""
    destination = ""

    if packet.haslayer(IP):
        source = packet[IP].src
        destination = packet[IP].dst
    elif packet.haslayer(IPv6):
        source = packet[IPv6].src
        destination = packet[IPv6].dst

    # 添加到表格
    row = self.packet_table.rowCount()
    self.packet_table.insertRow(row)

    # 设置表格项
    items = [
        QTableWidgetItem(str(self.packet_count)),
        QTableWidgetItem("0.000000"),  # 时间
        QTableWidgetItem(source),
        QTableWidgetItem(destination),
        QTableWidgetItem(protocol),
        QTableWidgetItem(str(len(packet))),
```

```
        QTableWidgetItem(""),  # 信息
    ]

    for i, item in enumerate(items):
        self.packet_table.setItem(row, i, item)

    # 更新计数
    self.packet_count += 1
```

8. 部署与配置

8.1 依赖库

库名	版本	用途
scapy	>=2.4.5	数据包处理
pyqt5	>=5.15.0	图形界面开发
pyqtgraph	>=0.12.0	统计图表绘制

表 12 依赖库信息

8.2 使用方法

```
# 安装依赖
pip install -r requirements.txt

# 运行命令行界面
python main.py

# 运行图形界面
python qt_gui.py
```

9. 总结

网络协议分析软件采用模块化、分层架构设计，具有良好的可扩展性和可维护性。软件支持命令行和图形界面两种交互方式，能够满足不同用户的需求。通过详细的设计文档，为软件开发和维护提供了清晰的指导，确保软件能够高效、稳定地运行。

该软件可以帮助网络管理员、安全分析师等专业人员进行网络流量分析、故障排查和安全检测，具有广泛的应用前景。未来可以通过添加更多功能和优化性能，进一步提高软件的实用性和易用性。