重庆邮电大学

学生实验实习报告册

学年学期：2025-2026 学年（1）学期

课程名称：通信软件开发与应用

学生学院：通信与信息工程学院

专业班级：01042301

学生学号：2023211281

学生姓名：丁同勖

联系电话：18019582857

重庆邮电大学教务处制

课程名称	通信软件开发与应用			**课程编号**	A2012230		
实验地点	YF315			**实验时间**	11 周 9-11 节		
校外指导教师	无			**校内指导教师**	梁燕		
实验名称	IP 抓包程序设计与实现						
评阅人签字		**操作成绩 80%**		**报告成绩 20%**		**总评成绩**	

一、实验目的

1、掌握 IP 数据报的结构
2、掌握原始套接字使用方法
3、掌握本机 IP 枚举方法
4、掌握网卡混杂模式设置方法
5、掌握地址过滤、端口过滤、协议过滤等基本数据包过滤方法

二、实验思路

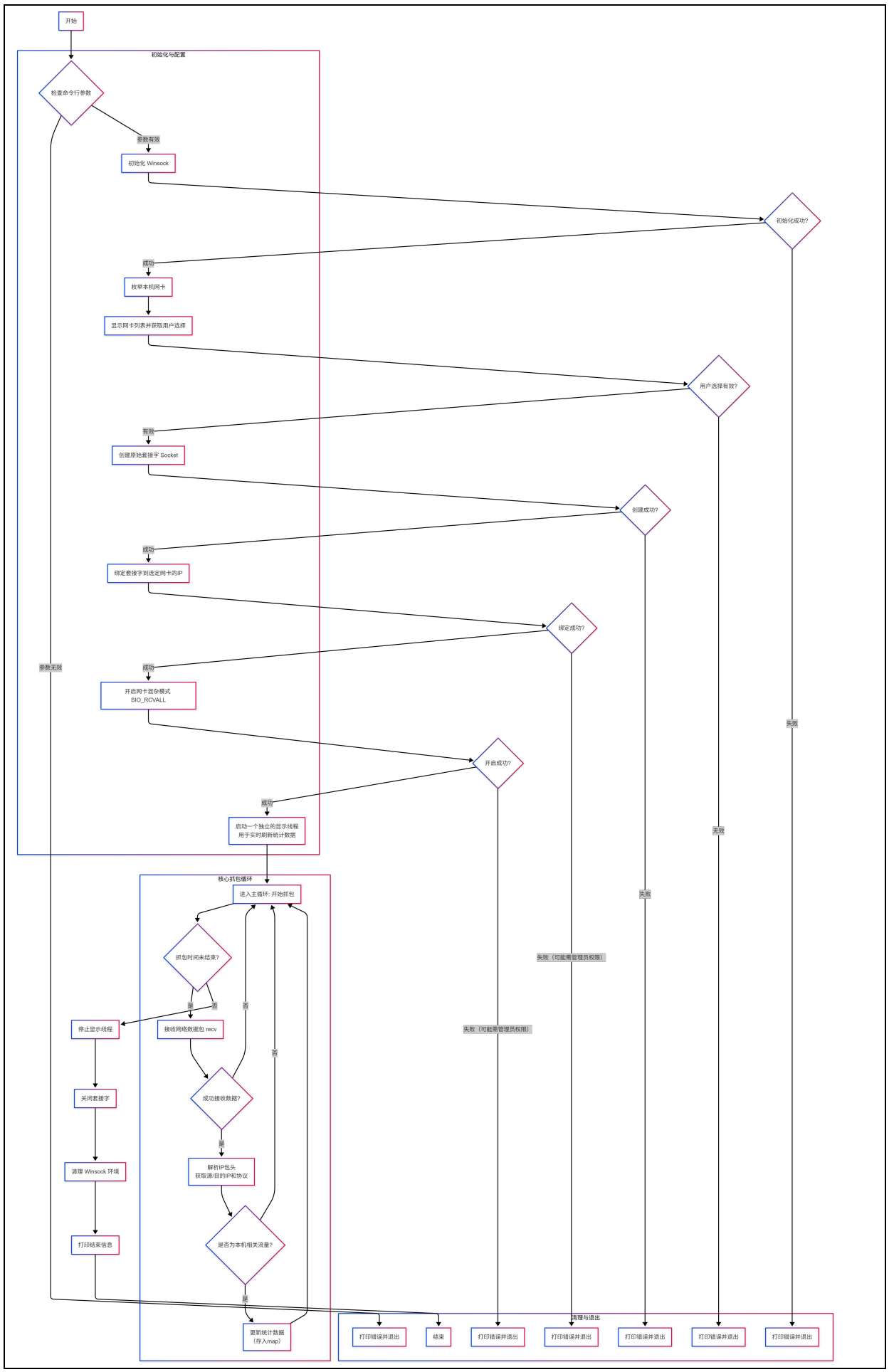
这是一个基于 Windows 平台的网络抓包程序，其核心功能是捕获流经特定网络接口卡的 IP 数据包，并实时统计和显示不同数据流（由源 IP、目的 IP 和协议定义）的数据包数量。以下是该程序的主要执行流程：

1. **参数检查与环境初始化**
   - **检查命令行参数**：程序启动时，首先检查是否提供了一个必需的命令行参数——抓包时长（秒）。如果参数缺失或无效（小于等于 0），则打印错误信息并退出。
   - **初始化 Winsock**：调用 WSAStartup 函数来初始化 Windows Socket API (WSA) 环境，这是在 Windows 上进行任何网络编程前必须执行的步骤。
2. **网络适配器选择**
   - **枚举网卡**：使用 GetAdaptersInfo API 获取本机上所有网络适配器的详细信息，包括描述和 IP 地址。
   - **显示与选择**：将所有可用的网卡列表打印到控制台，并提示用户输入一个编号来选择要监控的网卡。
   - **保存选择**：根据用户的输入，保存选定的网卡描述及其 IP 地址，用于后续的套接字绑定和信息显示。
3. **原始套接字配置**
   - **创建原始套接字**：调用 socket 函数创建一个 SOCK_RAW 类型的原始套接字，并指定协议为 IPPROTO_IP。这使得程序能够直接访问和处理网络层的 IP 数据包。
   - **绑定套接字**：调用 bind 函数将创建的套接字与用户选择的网卡 IP 地址进行绑定。这一步是必需的，以便后续开启混杂模式时能正确地作用于指定的网卡。
   - **开启混杂模式**：通过 WSAIoctl 函数和 SIO_RCVALL 控制码，将网卡设置为混杂模式（Promiscuous Mode）。在此模式下，网卡会接收所有流经它的数据包，而不仅仅是发往本机 IP 地址的数据包。**此操作通常需要管理员权限**。
4. **多线程实时显示**
   - **启动显示线程**：程序创建一个独立的子线程 (displayThread)，专门负责 UI 的刷新。
   - **实时刷新**：该线程在一个循环中运行，每秒钟清空一次控制台 (system("cls"))，然后打印出当前的抓包进度、网卡信息以及存储在 statistics map 中的实时数据包统计结果。这种设计将数据捕获与数据显示分离，避免了 UI 刷新阻塞主线程的抓包逻辑。
5. **核心抓包与分析循环**
   - **循环抓包**：主线程进入一个 while 循环，循环的终止条件是当前时间超过预设的抓包

时长。

- **接收数据**：在循环内部，调用 recv 函数阻塞式地等待并接收一个 IP 数据包到缓冲区 buffer 中。
- **解析 IP 头**：成功接收到数据后，程序直接从缓冲区解析 IP 包头。它提取出协议类型（如 TCP, UDP, ICMP）、源 IP 地址和目的 IP 地址。
- **数据包过滤**：程序会进行简单的过滤，忽略掉广播包（目的 IP 为 255.255.255.255）以及与本机 IP 无关的数据包（即源 IP 和目的 IP 都不是本机 IP 的包）。
- **更新统计**：对于通过过滤的包，程序构造一个 Key（包含源 IP、目的 IP、协议），并在 statistics map 中为该 Key 对应的计数值加一。

6. **清理与退出**
   - **停止显示**：当抓包时间结束后，主循环退出。程序将 running 标志位设为 false，通知显示线程结束其循环，并调用 join() 等待该线程安全退出。
   - **资源释放**：最后，程序调用 closesocket 关闭原始套接字，并调用 WSACleanup 释放 Winsock 库占用的所有资源。
   - **结束**：打印抓包结束信息，程序正常退出。


开始
初始化与配置
检查命令行参数
参数有效
初始化 Winsock
初始化成功?
成功
枚举本机网卡
显示网卡列表并获取用户选择
用户选择有效?
有效
创建原始套接字 Socket
创建成功?
成功
绑定套接字到选定网卡的IP
绑定成功?
参数无效
成功
开启网卡混杂模式
SIO_RCVALL
开启成功?
失败
成功
启动一个独立的显示线程
用于实时刷新统计数据
无效
核心抓包循环
进入主循环: 开始抓包
失败
抓包时间未结束?
失败（可能需管理员权限）
是
否
否
停止显示线程
接收网络数据包 recv
失败（可能需管理员权限）
否
关闭套接字
成功接收数据?
是
清理 Winsock 环境
解析IP包头
获取源/目的IP和协议
打印结束信息
是否为本机相关流量?
是
更新统计数据
（存入map）
清理与退出
打印错误并退出
结束
打印错误并退出
打印错误并退出
打印错误并退出
打印错误并退出
打印错误并退出

三、源代码

```
// 禁用一些旧版函数API的安全警告，以便于使用一些传统的Windows Socket函数
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS

// --------------------------------
// 包含头文件
// --------------------------------
#include <winsock2.h>     // Windows Socket 2 主要头文件
#include <ws2tcpip.h>     // TCP/IP 协议相关的附加定义，如 inet_pton
#include <iphlpapi.h>    // IP 帮助程序 API，用于获取网络适配器信息
#include <mstcpip.h>      // 包含 SIO_RCVALL 等 IOCTL 控制码的定义
#include <iostream>       // 标准输入输出流
#include <iomanip>        // 用于格式化输出，如 setw
#include <map>            // 用于存储统计数据
#include <vector>         // 用于存储网络适配器列表
#include <string>         // C++ 字符串操作
#include <thread>         // C++11 线程支持，用于实时显示
#include <chrono>         // C++11 时间库，用于计时

// --------------------------------
// 链接库
// --------------------------------
#pragma comment(lib, "ws2_32.lib")      // 链接 Windows Socket 2 库
#pragma comment(lib, "iphlpapi.lib")    // 链接 IP 帮助程序 API 库

// --------------------------------
// 全局常量与命名空间
// --------------------------------
// 定义带有颜色的控制台输出消息前缀，以区分不同类型的消息
const std::string ErrorMsg = "\033[31m[ERR]\033[0m\t";      // 红色错误消息
const std::string InformationMsg = "\033[32m[INFO]\033[0m\t"; // 绿色信息消息
const std::string WarningMsg = "\033[33m[WARN]\033[0m\t";    // 黄色警告消息

using namespace std;

// --------------------------------
// 数据结构定义
// --------------------------------

/**
 * @brief 用于作为 map 的键，以统计不同的数据流。
 *        一个数据流由源IP、目的IP和协议唯一确定。
 */
struct Key {
    string src;   // 源IP地址
```

```
    string dst;   // 目的IP地址
    string proto; // 协议名称

    /**
     * @brief 重载小于运算符，使得 Key 结构体可以被用作 std::map 的键。
     * @param other 另一个 Key 对象。
     * @return 如果当前对象小于 other，则返回 true。
     */
    bool operator<(const Key& other) const {
        return std::tie(src, dst, proto) < std::tie(other.src, other.dst, other.proto);
    }
};

/**
 * @brief 将IP头中的协议号转换为可读的字符串名称。
 * @param proto IP数据包头中的协议字段值 (1 byte)。
 * @return 对应的协议名称字符串。
 */
string protocolName(unsigned char proto) {
    switch (proto) {
    case 1: return "ICMP";
    case 2: return "IGMP";
    case 6: return "TCP";
    case 17: return "UDP";
    default: return "Other"; // 其他未识别的协议
    }
}

// ------------------------------
// 主函数
// ------------------------------
int main(int argc, char* argv[])
{
    // --- 1. 参数检查 ---
    if (argc != 2) {
        cout << ErrorMsg << "用法: IP_Monitor.exe <抓包时间(秒)>\n";
        return -1;
    }

    int captureSeconds = atoi(argv[1]); // 将命令行参数从字符串转换为整数
    if (captureSeconds <= 0) {
        cout << ErrorMsg << "抓包时间无效\n";
        return -1;
    }

    // --- 2. 初始化 Winsock ---
```

```
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
        cout << ErrorMsg << "WSAStartup 初始化失败\n";
        return -1;
    }

    // --- 3. 枚举并选择网络适配器 ---
    ULONG len = 15000; // 预分配一个足够大的缓冲区
    IP_ADAPTER_INFO* pAdapterInfo = (IP_ADAPTER_INFO*)malloc(len);

    // 第一次调用 GetAdaptersInfo 获取所需的缓冲区大小
    if (GetAdaptersInfo(pAdapterInfo, &len) == ERROR_BUFFER_OVERFLOW) {
        free(pAdapterInfo); // 释放旧缓冲区
        pAdapterInfo = (IP_ADAPTER_INFO*)malloc(len); // 按实际大小重新分配
    }
    // 第二次调用获取信息
    if (GetAdaptersInfo(pAdapterInfo, &len) != NO_ERROR) {
        cout << ErrorMsg << "GetAdaptersInfo() 失败\n";
        free(pAdapterInfo);
        return -1;
    }

    vector<IP_ADAPTER_INFO*> adapters; // 存储所有适配器信息的指针
    IP_ADAPTER_INFO* p = pAdapterInfo;

    cout << InformationMsg << "本机网卡列表：\n";
    int idx = 1;
    int maxDescriptionLen = 0; // 用于对齐输出

    // 遍历链表，找到最长的描述长度
    while (p) {
        int currentLen = strlen(p->Description);
        if (currentLen > maxDescriptionLen)
            maxDescriptionLen = currentLen;
        adapters.push_back(p);
        p = p->Next;
    }

    // 打印所有适配器信息
    for (auto* adp : adapters) {
        cout << "\t" << "\033[33m" << idx++ << ".\033[0m " << left << setw(maxDescriptionLen + 2)
<< adp->Description
            << "\033[33mIP: \033[0m" << adp->IpAddressList.IpAddress.String << "\n";
    }

    // 获取用户选择
```

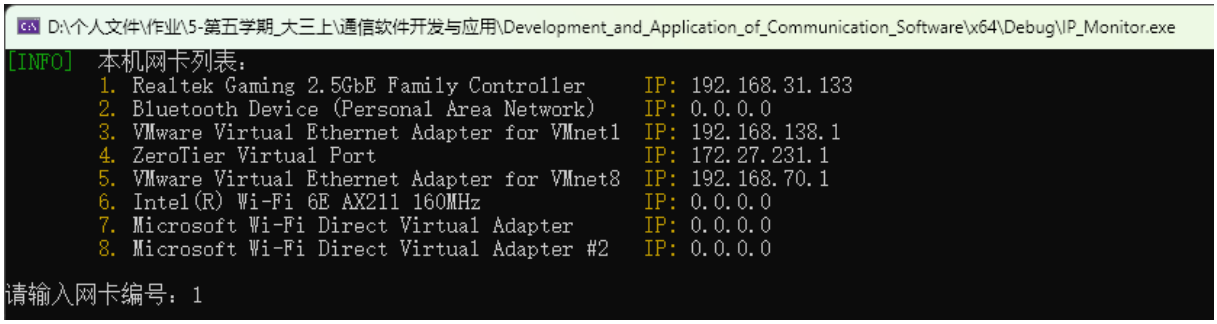
```
    cout << "\n请输入网卡编号：";
    int choice;
    cin >> choice;

    if (choice <= 0 || choice > adapters.size()) {
        cout << ErrorMsg << "无效的网卡编号\n";
        free(pAdapterInfo);
        return -1;
    }

    // 保存用户选择的网卡信息和IP地址
    string AdapterInfo = adapters[choice - 1]->Description;
    string localIP = adapters[choice - 1]->IpAddressList.IpAddress.String;

    // 释放为适配器信息分配的内存
    free(pAdapterInfo);

    // --- 4. 创建原始套接字并绑定 ---
    // AF_INET: IPv4, SOCK_RAW: 原始套接字, IPPROTO_IP: 捕获IP层数据包
    SOCKET s = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    if (s == INVALID_SOCKET) {
        cout << ErrorMsg << "socket 创建失败，错误代码: " << WSAGetLastError() << "\n";
        WSACleanup();
        return -1;
    }

    // 绑定套接字到选择的网卡IP上
    sockaddr_in localAddr;
    localAddr.sin_family = AF_INET;
    inet_pton(AF_INET, localIP.c_str(), &localAddr.sin_addr);
    localAddr.sin_port = 0; // 端口号对于原始套接字无意义

    if (bind(s, (sockaddr*)&localAddr, sizeof(localAddr)) == SOCKET_ERROR) {
        cout << ErrorMsg << "bind 失败（需要以管理员权限运行），错误代码: " << WSAGetLastError()
<< "\n";
        closesocket(s);
        WSACleanup();
        return -1;
    }

    // --- 5. 开启网卡混杂模式 ---
    // SIO_RCVALL: 控制码，使网卡接收所有流经它的数据包，而不仅仅是发给本机的数据包
    DWORD dwValue = 1; // 1 表示开启
    if (WSAIoctl(s, SIO_RCVALL, &dwValue, sizeof(dwValue),
        NULL, 0, &dwValue, NULL, NULL) == SOCKET_ERROR) {
        cout << ErrorMsg << "无法开启混杂模式，请确保以管理员权限运行！错误代码: " <<
```

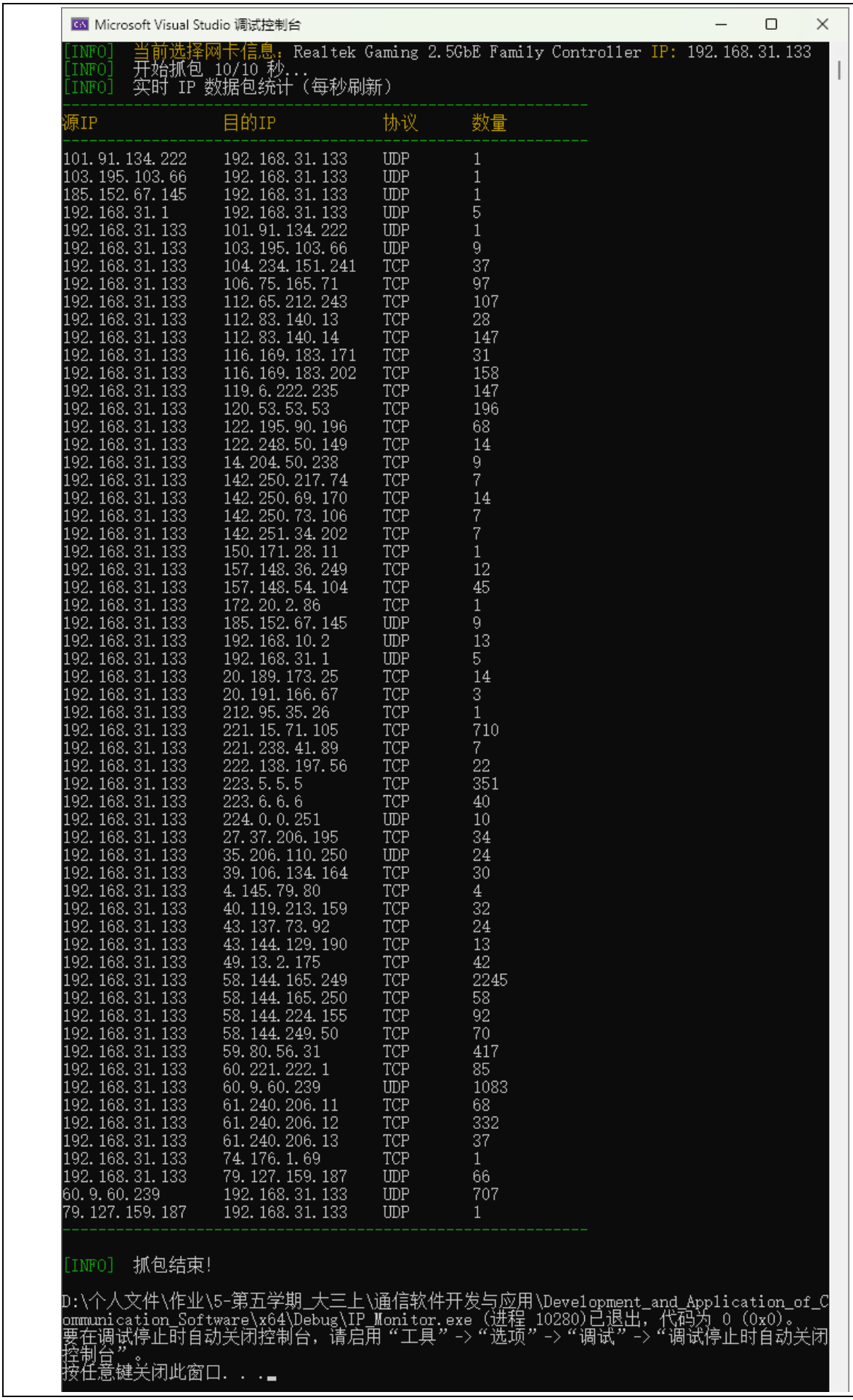
```
WSAGetLastError() << "\n";
        closesocket(s);
        WSACleanup();
        return -1;
    }

    // --- 6. 准备数据统计与多线程显示 ---
    map<Key, int> statistics; // 用于存储数据包统计结果
    bool running = true; // 控制显示线程的循环

    auto startTime = chrono::steady_clock::now(); // 记录抓包开始时间
    auto endTime = startTime + chrono::seconds(captureSeconds); // 计算抓包结束时间

    // 创建一个子线程，用于实时刷新和显示统计数据
    thread displayThread([&]() {
        while (running) {
            system("cls"); // 清空控制台屏幕

            // 打印标题和当前状态
            cout << InformationMsg << "\033[33m当前选择网卡信息：\033[0m" << AdapterInfo <<
"\033[33m IP: \033[0m" << localIP << endl;
            auto elapsed = chrono::duration_cast<chrono::seconds>(chrono::steady_clock::now() -
startTime).count() + 1;
            if (elapsed > captureSeconds) {
                cout << InformationMsg << "开始抓包 " << captureSeconds << "/" << captureSeconds
<< " 秒...\n";
            }
            else {
                cout << InformationMsg << "开始抓包 " << elapsed << "/" << captureSeconds << "
秒...\n";
            }
            cout << InformationMsg << "实时 IP 数据包统计（每秒刷新）\n";
            cout <<
"\033[32m------------------------------------------------------------\033[0m\n";
            cout << "\033[33m" << left << setw(18) << "源IP"
                << setw(18) << "目的IP"
                << setw(10) << "协议"
                << setw(8) << "数量" << "\033[0m" << endl;
            cout <<
"\033[32m------------------------------------------------------------\033[0m\n";

            // 遍历 map 并打印统计数据
            for (auto& kv : statistics) {
                cout << left << setw(18) << kv.first.src
                    << setw(18) << kv.first.dst
                    << setw(10) << kv.first.proto
```

```
                    << setw(8) << kv.second << endl;
            }

            cout <<
"\033[32m-------------------------------------------------------------\033[0m\n";
            this_thread::sleep_for(chrono::seconds(1)); // 每秒刷新一次
        }
        });

    // --- 7. 主循环：抓包与分析 ---
    unsigned char buffer[65536]; // 定义一个足够大的缓冲区来接收数据包

    while (chrono::steady_clock::now() < endTime) {
        // 从套接字接收数据，recv会阻塞直到有数据到达
        int ret = recv(s, (char*)buffer, sizeof(buffer), 0);
        if (ret <= 0) continue; // 如果接收失败或没有数据，则继续下一次循环

        // ---- 解析IP头 ----
        // 接收到的 buffer 直接就是 IP 头开始的数据
        unsigned char* ip = buffer;

        // IP头第10个字节(偏移量为9)是协议字段
        unsigned char proto = ip[9];

        // 解析源和目的IP地址
        // 源IP地址在偏移量 12-15 字节
        // 目的IP地址在偏移量 16-19 字节
        char srcIP[16], dstIP[16];
        sprintf(srcIP, "%d.%d.%d.%d", ip[12], ip[13], ip[14], ip[15]);
        sprintf(dstIP, "%d.%d.%d.%d", ip[16], ip[17], ip[18], ip[19]);

        // ---- 数据包过滤 ----
        // 过滤掉广播包
        if (strcmp(dstIP, "255.255.255.255") == 0)
            continue;

        // 只统计与本机IP相关的数据包（本机作为源或目的）
        if (localIP != srcIP && localIP != dstIP)
            continue;

        // ---- 更新统计信息 ----
        Key k{ srcIP, dstIP, protocolName(proto) }; // 创建一个Key
        statistics[k]++; // 更新对应数据流的计数
    }

    // --- 8. 结束与清理 ---
```

```
    running = false; // 通知显示线程退出循环
    displayThread.join(); // 等待显示线程执行完毕

    // 关闭套接字并清理 Winsock 环境
    closesocket(s);
    WSACleanup();

    cout << "\n" << InformationMsg << "抓包结束！\n";

    return 0;
}
```

四、实验结果及分析




Microsoft Visual Studio 调试控制台
[INFO] 当前选择网卡信息：Realtek Gaming 2.5GbE Family Controller IP: 192.168.31.133
[INFO] 开始抓包 10/10 秒...
[INFO] 实时 IP 数据包统计（每秒刷新）
源IP 目的IP 协议 数量
101.91.134.222 192.168.31.133 UDP 1
103.195.103.66 192.168.31.133 UDP 1
185.152.67.145 192.168.31.133 UDP 1
192.168.31.1 192.168.31.133 UDP 5
192.168.31.133 101.91.134.222 UDP 1
192.168.31.133 103.195.103.66 UDP 9
192.168.31.133 104.234.151.241 TCP 37
192.168.31.133 106.75.165.71 TCP 97
192.168.31.133 112.65.212.243 TCP 107
192.168.31.133 112.83.140.13 TCP 28
192.168.31.133 112.83.140.14 TCP 147
192.168.31.133 116.169.183.171 TCP 31
192.168.31.133 116.169.183.202 TCP 158
192.168.31.133 119.6.222.235 TCP 147
192.168.31.133 120.53.53.53 TCP 196
192.168.31.133 122.195.90.196 TCP 68
192.168.31.133 122.248.50.149 TCP 14
192.168.31.133 14.204.50.238 TCP 9
192.168.31.133 142.250.217.74 TCP 7
192.168.31.133 142.250.69.170 TCP 14
192.168.31.133 142.250.73.106 TCP 7
192.168.31.133 142.251.34.202 TCP 7
192.168.31.133 150.171.28.11 TCP 1
192.168.31.133 157.148.36.249 TCP 12
192.168.31.133 157.148.54.104 TCP 45
192.168.31.133 172.20.2.86 TCP 1
192.168.31.133 185.152.67.145 UDP 9
192.168.31.133 192.168.10.2 UDP 13
192.168.31.133 192.168.31.1 UDP 5
192.168.31.133 20.189.173.25 TCP 14
192.168.31.133 20.191.166.67 TCP 3
192.168.31.133 212.95.35.26 TCP 1
192.168.31.133 221.15.71.105 TCP 710
192.168.31.133 221.238.41.89 TCP 7
192.168.31.133 222.138.197.56 TCP 22
192.168.31.133 223.5.5.5 TCP 351
192.168.31.133 223.6.6.6 TCP 40
192.168.31.133 224.0.0.251 UDP 10
192.168.31.133 27.37.206.195 TCP 34
192.168.31.133 35.206.110.250 UDP 24
192.168.31.133 39.106.134.164 TCP 30
192.168.31.133 4.145.79.80 TCP 4
192.168.31.133 40.119.213.159 TCP 32
192.168.31.133 43.137.73.92 TCP 24
192.168.31.133 43.144.129.190 TCP 13
192.168.31.133 49.13.2.175 TCP 42
192.168.31.133 58.144.165.249 TCP 2245
192.168.31.133 58.144.165.250 TCP 58
192.168.31.133 58.144.224.155 TCP 92
192.168.31.133 58.144.249.50 TCP 70
192.168.31.133 59.80.56.31 TCP 417
192.168.31.133 60.221.222.1 TCP 85
192.168.31.133 60.9.60.239 UDP 1083
192.168.31.133 61.240.206.11 TCP 68
192.168.31.133 61.240.206.12 TCP 332
192.168.31.133 61.240.206.13 TCP 37
192.168.31.133 74.176.1.69 TCP 1
192.168.31.133 79.127.159.187 UDP 66
60.9.60.239 192.168.31.133 UDP 707
79.127.159.187 192.168.31.133 UDP 1
[INFO] 抓包结束!
D:\个人文件\作业\5-第五学期_大三上\通信软件开发与应用\Development_and_Application_of_Communication_Software\x64\Debug\IP_Monitor.exe (进程 10280)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .

五、实验中问题及分析

1. 一开始创建套接字失败，在调试属性中将UAC 执行级别调整为requireAdministrator后成功执行。
2. 对于每秒更新结果的方式进行了较多次的迭代，从每秒钟叠加输出完整抓取结果，变更为清空后重新输出，更加美观直观。

六、心得体会

1. 对底层网络工作机制的深刻理解

理论学习网络协议（如 TCP/IP）和实际动手编写嗅探器是两种完全不同的体验。

- **原始套接字是关键：**平时我们使用的 SOCK_STREAM (TCP) 或 SOCK_DGRAM (UDP) 套接字都是经过操作系统协议栈层层处理后，将应用层数据交给我们的。而使用 SOCK_RAW 才让我们有机会直接触碰到网络层（IP 层）的原始数据包，这是实现嗅探器的基石。
- **混杂模式的威力：**代码中通过 WSAIoctl 和 SIO_RCVALL 开启了网卡的混杂模式。这让我直观地认识到，物理上，网卡能“听到”所有流经它的数据帧，而不仅仅是发给自己的。开启此模式就是授权程序去处理这些“路过”的数据，极大地扩展了监控范围。
- **数据包结构不再抽象：**当亲手从 unsigned char* buffer 中按字节偏移量（如 ip[9] 取协议，ip[12] 开始取源 IP）解析数据时，IP 包头的格式、每个字段的含义和长度都变得具体而清晰。这比单纯看协议图解要印象深刻得多。

2. C++ 特性在系统编程中的应用

学习了通过现代 C++特性来编写结构清晰、功能强大的系统级应用。

- **RAII 思想的重要性：**虽然代码中使用了手动的 closesocket 和 WSACleanup，但可以想象在更复杂的应用中，将 Socket 句柄、内存分配等封装在类中，利用构造函数和析构函数自动管理资源，能让代码更健壮，避免资源泄漏。
- **多线程提升用户体验：**项目最大的亮点之一是**将数据捕获与 UI 显示分离**。主线程专注于高效地执行阻塞的 recv 调用并处理数据，而子线程 (displayThread) 则负责定时刷新统计界面。这种设计是构建响应式、高性能网络应用的典范，避免了因 UI 刷新而丢失数据包，或因等待数据包而导致界面卡顿。
- **STL 容器的使用：**使用 std::map 配合自定义的 struct Key 来进行数据统计非常优雅。只需要为 Key 重载 operator<，就可以轻松地以“源 IP-目的 IP-协议”为唯一标识符，自动完成数据流的分类和计数，代码简洁且高效。

3. 实践中的经验

编写过程中会遇到一些理论学习时不会注意到的细节问题。

- **管理员权限是前提：**创建原始套接字、绑定 IP 以及开启混杂模式，这些都属于特权操作。初次运行时因为权限不足而导致 bind 或 WSAIoctl 失败。这让我认识到网络底层操作与操作系统安全策略的紧密联系。
- **字节序问题（潜在）：**在这个项目中，由于只解析 IP 地址和协议号（单字节），没有涉及到端口号等（2 字节）字段，所以没有显式处理网络字节序（Big-Endian）和主机字节序（Little-Endian）的转换。但在更深入的解析（如 TCP/UDP 端口）中，使用 ntohs() 等函数是必须的，这是一个极易被忽略的陷阱。