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**	10 周 9-11 节		
校外指导教师	无			**校内指导教师**	梁燕		
实验名称	ARP 协议探测局域网内活动主机 MAC 地址						
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 ARP 协议工作原理
2、掌握本机 IP 地址枚举方法
3、掌握 Win32API 函数使用方法
4、掌握程序编辑、编译、链接、调试等基本概念
5、掌握程序基本排错方法及断点调试技巧

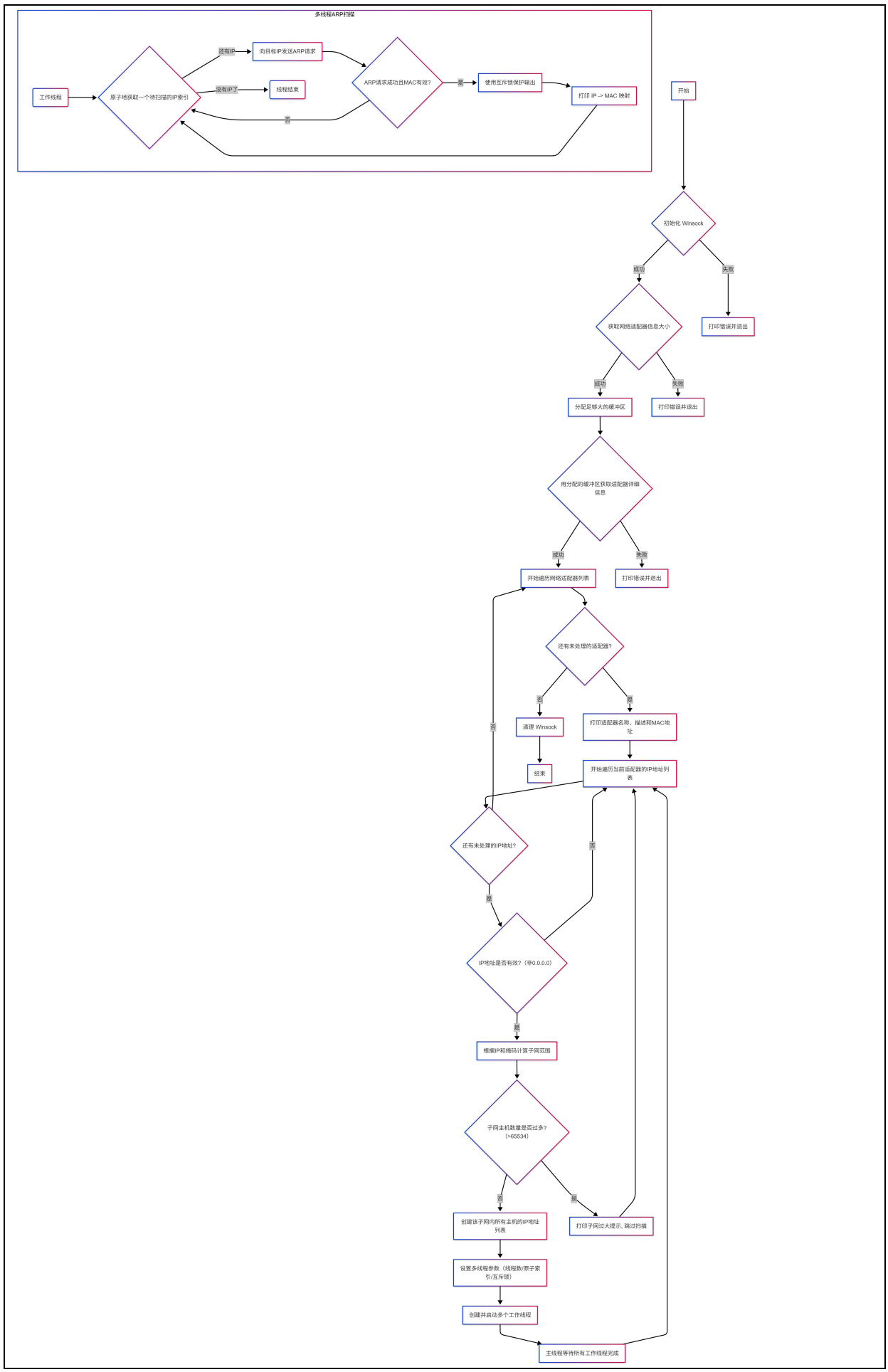
二、实验思路

此程序是一个使用多线程在 Windows 环境下扫描局域网内活动主机并获取其 MAC 地址的工具。其主要工作流程如下：

1. **环境初始化**:
    - 程序首先调用 WSAStartup 初始化 Winsock 库，为后续的网络操作准备环境。如果初始化失败，程序将打印错误信息并退出。
2. **获取网络适配器信息**:
    - 通过两次调用 GetAdaptersInfo 函数来获取本机所有网络适配器的详细信息。第一次调用是为了确定需要多大的缓冲区来存储这些信息，第二次调用则将信息实际读入到分配好的缓冲区中。任何一步失败都会导致程序退出。
3. **遍历适配器与 IP 地址**:
    - 程序会遍历获取到的所有网络适配器。对于每一个适配器，它会打印其名称、描述和物理地址（MAC 地址）。
    - 接着，程序会遍历该适配器上配置的所有 IP 地址。它会跳过无效的 "0.0.0.0" 地址。
4. **计算子网范围**:
    - 对于每个有效的 IP 地址及其子网掩码，程序会计算出所在的子网的网络地址和广播地址，从而确定需要扫描的 IP 地址范围（通常是网络地址+1 到 广播地址-1）。
    - 为了防止对大型网络（如 B 类子网）进行扫描导致程序耗时过长和资源消耗过大，程序会检查子网内的主机数量。如果数量超过阈值（如 65534），则会跳过该子网的扫描。
5. **多线程并行扫描**:

- **准备工作**: 确定要扫描的 IP 地址列表后，程序会根据当前系统的 CPU 核心数确定要创建的线程数量，以实现最高效的并行扫描。同时，创建一个原子计数器 nextIndex 用于在线程间安全地分配扫描任务，以及一个互斥锁 printMutex 来避免多线程同时向控制台输出造成信息混乱。
- **工作线程**: 程序会启动多个工作线程。每个线程在一个循环中不断地从 nextIndex 获取一个新的 IP 地址进行处理，直到所有 IP 地址都被分配完毕。
- **ARP 请求**: 线程获取到 IP 地址后，会调用 SendARP 函数向该 IP 发送一个 ARP 请求。
- **结果处理**: 如果 SendARP 调用成功返回，并且获取到了一个有效的、非零的 MAC 地址，线程就会锁住 printMutex，然后将目标 IP 地址和对应的 MAC 地址打印到控制台。

6. **等待与清理**:
   - 在启动所有工作线程后，主线程会调用 join() 等待所有线程完成扫描任务。
   - 当一个适配器的所有子网都扫描完毕后，程序会继续处理下一个适配器。
   - 在所有适配器都处理完毕后，程序调用 WSACleanup 释放 Winsock 库资源，然后正常退出。


多线程ARP扫描
工作线程
原子地获取一个待扫描的IP索引
还有IP
向目标IP发送ARP请求
没有IP了
线程结束
ARP请求成功且MAC有效?
是
否
使用互斥锁保护输出
打印 IP -> MAC 映射
开始
初始化 Winsock
成功
失败
打印错误并退出
获取网络适配器信息大小
成功
失败
分配足够大的缓冲区
打印错误并退出
用分配的缓冲区获取适配器详细信息
成功
失败
开始遍历网络适配器列表
打印错误并退出
还有未处理的适配器?
否
是
清理 Winsock
打印适配器名称、描述和MAC地址
否
结束
开始遍历当前适配器的IP地址列表
还有未处理的IP地址?
是
否
是
IP地址是否有效?（非0.0.0.0）
是
根据IP和掩码计算子网范围
子网主机数量是否过多?（>65534）
否
是
创建该子网内所有主机的IP地址列表
打印子网过大提示, 跳过扫描
设置多线程参数（线程数/原子索引/互斥锁）
创建并启动多个工作线程
主线程等待所有工作线程完成

三、源代码

```
// 在某些较新的 Visual Studio 版本中，一些传统的 Winsock 函数被标记为“已弃用”，定义此宏以禁用关于使用旧版 Winsock 函数的警告。
#define _WINSOCK_DEPRECATED_NO_WARNINGS

#include <winsock2.h>      // Winsock 核心功能，用于网络编程
#include <iphlpapi.h>      // IP Helper API，用于获取网络适配器信息，如 GetAdaptersInfo, SendARP
#include <ws2tcpip.h>      // Winsock 2 TCP/IP 协议的附加定义
#include <windows.h>       // Windows 核心 API，包含基本数据类型和函数
#include <iostream>        // 用于标准输入输出流 (cout, cerr)
#include <iomanip>         // 用于流操作，如 setfill, setw
#include <string>          // C++ 字符串类 (std::string)
#include <vector>          // C++ 动态数组 (std::vector)
#include <cstdint>         // 标准整数类型，如 uint32_t
#include <sstream>         // 字符串流，用于构建字符串
#include <thread>          // C++11 线程支持 (std::thread)
#include <atomic>          // C++11 原子操作，用于线程安全计数器
#include <mutex>           // C++11 互斥锁，用于保护共享资源

#pragma comment(lib, "iphlpapi.lib") // 链接 IP Helper API 库，提供网络管理功能
#pragma comment(lib, "ws2_32.lib")   // 链接 Winsock 2.2 库，提供核心网络套接字功能

const std::string ErrorMsg = "\033[31m[ERR]\033[0m\t";
const std::string InformationMsg = "\033[32m[INFO]\033[0m\t";
const std::string WarningMsg = "\033[33m[WARN]\033[0m\t";

using namespace std;

/**
 * @brief 将字节数组格式化为冒号分隔的十六进制 MAC 地址字符串。
 * @param addr 指向包含 MAC 地址的字节数组的指针。
 * @param len MAC 地址的长度（以字节为单位）。
 * @return 格式化后的 MAC 地址字符串。如果长度为 0，则返回空字符串。
 */
static string formatMac(const BYTE* addr, ULONG len) {
    if (len == 0) return "";
    ostringstream oss;
    oss << hex << setfill('0'); // 设置为十六进制输出，并用 '0' 填充
    for (ULONG i = 0; i < len; ++i) {
        if (i) oss << ':'; // 在字节之间添加冒号
        oss << setw(2) << static_cast<int>(addr[i]); // 格式化每个字节为两位十六进制数
    }
    return oss.str();
}

/**
```

```
 * @brief 检查给定的 MAC 地址是否为全零。
 * @param addr 指向包含 MAC 地址的字节数组的指针。
 * @param len MAC 地址的长度（以字节为单位）。
 * @return 如果 MAC 地址全为零，则返回 true，否则返回 false。
 */
static bool isZeroMac(const BYTE* addr, ULONG len) {
    for (ULONG i = 0; i < len; ++i)
        if (addr[i] != 0) return false; // 一旦发现非零字节，立即返回 false
    return true;
}

int main() {
    // 初始化 Winsock
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << ErrorMsg << "WSAStartup failed" << endl;
        return 1;
    }

    // --- 获取网络适配器信息 ---
    ULONG outBufLen = 0;
    // 第一次调用 GetAdaptersInfo，传入 nullptr，以获取所需的缓冲区大小
    DWORD dwRet = GetAdaptersInfo(nullptr, &outBufLen);
    if (dwRet != ERROR_BUFFER_OVERFLOW) {
        cerr << ErrorMsg << "GetAdaptersInfo failed to get buffer size: " << dwRet << endl;
        WSACleanup();
        return 1;
    }

    // 根据获取的大小分配缓冲区
    vector<BYTE> buffer(outBufLen);
    PIP_ADAPTER_INFO pAdapterInfo = reinterpret_cast<PIP_ADAPTER_INFO>(buffer.data());

    // 第二次调用 GetAdaptersInfo，传入分配好的缓冲区以获取适配器信息
    dwRet = GetAdaptersInfo(pAdapterInfo, &outBufLen);
    if (dwRet != NO_ERROR) {
        cerr << ErrorMsg << "GetAdaptersInfo failed: " << dwRet << endl;
        WSACleanup();
        return 1;
    }

    cout << InformationMsg << "网络适配器与扫描结果：" << endl;

    // --- 遍历所有网络适配器 ---
    for (PIP_ADAPTER_INFO pAdapter = pAdapterInfo; pAdapter != nullptr; pAdapter = pAdapter->Next)
{
```

```
        cout << endl << InformationMsg << "\033[32m-------------------------[适配器信
息]-------------------------\033[0m\t" << endl;
        cout << InformationMsg << "\033[33m适配器:\033[0m" << pAdapter->AdapterName << " (" <<
pAdapter->Description << ")" << endl;
        cout << InformationMsg << "\033[33mMAC: \033[0m" << formatMac(pAdapter->Address,
pAdapter->AddressLength) << endl;

        // --- 遍历适配器的每个 IP 地址 ---
        for (IP_ADDR_STRING* ipAddr = &pAdapter->IpAddressList; ipAddr != nullptr; ipAddr =
ipAddr->Next) {
            if (ipAddr->IpAddress.String[0] == '\0') break; // 如果 IP 地址字符串为空，则停止
            string ipStr = ipAddr->IpAddress.String;
            string maskStr = ipAddr->IpMask.String;

            // 忽略无效的 "0.0.0.0" 地址
            if (ipStr == "0.0.0.0") continue;

            cout << InformationMsg << "\033[33mIP: \033[0m" << ipStr << "\033[33m 掩码: \033[0m"
<< maskStr << "\033[33m 网关: \033[0m" << pAdapter->GatewayList.IpAddress.String << endl;
             cout << InformationMsg << "\033[32m[开始扫描局域网...]\033[0m\t" << endl;

            // --- 计算子网范围 ---
            uint32_t ip = ntohl(inet_addr(ipStr.c_str()));   // 将点分十进制 IP 字符串转为网络字
节序整数，再转为主机字节序
            uint32_t mask = ntohl(inet_addr(maskStr.c_str())); // 同上处理子网掩码
            if (mask == 0) continue; // 如果掩码无效，则跳过

            uint32_t network = ip & mask;                    // 计算网络地址
            uint32_t broadcast = network | (~mask);          // 计算广播地址

            // 打印子网范围
            cout << InformationMsg << "\033[33m扫描子网范围: \033[0m" << ((network >> 24) & 0xFF)
<< "." << ((network >> 16) & 0xFF) << "." << ((network >> 8) & 0xFF) << "." << (network & 0xFF)
                << "\033[33m - \033[0m" << ((broadcast >> 24) & 0xFF) << "." << ((broadcast >> 16)
& 0xFF) << "." << ((broadcast >> 8) & 0xFF) << "." << (broadcast & 0xFF) << endl;

            // --- 准备要扫描的主机地址 ---
            uint32_t start = network + 1; // 子网的第一个可用主机地址
            uint32_t end = (broadcast > 0) ? (broadcast - 1) : broadcast; // 子网的最后一个可用
主机地址
            uint64_t hostCount = (end >= start) ? (uint64_t)(end - start + 1) : 0;

            if (hostCount == 0) {
                cout << WarningMsg << "No hosts to scan on this interface." << endl;
                continue;
            }
```

```
            // 避免扫描过大的子网（例如 /16），这会消耗大量时间和资源
            if (hostCount > 65534) {
                cout << WarningMsg << "Subnet too large (" << hostCount << " hosts). Skipping detailed scan." << endl;
                continue;
            }

            // 创建要扫描的 IP 地址列表（网络字节序）
            vector<uint32_t> candidates;
            candidates.reserve(static_cast<size_t>(hostCount));
            for (uint32_t h = start; h <= end; ++h) {
                candidates.push_back(htonl(h)); // 存入网络字节序，因为 SendARP 需要这种格式
            }

            // --- 多线程扫描 ---
            // 确定要使用的线程数
            unsigned int hc = thread::hardware_concurrency(); // 获取硬件支持的并发线程数
            if (hc == 0) hc = 4; // 如果无法获取，则默认为 4
            unsigned int maxThreads = hc;
            if (maxThreads > 64) maxThreads = 64; // 限制最大线程数为 64
            if (maxThreads > candidates.size()) maxThreads = static_cast<unsigned int>(candidates.size()); // 线程数不超过 IP 数量
            if (maxThreads == 0) maxThreads = 1; // 至少使用一个线程

            cout << InformationMsg << "使用 " << maxThreads << " 个线程扫描 " << candidates.size() << " 台主机..." << endl;

            atomic<size_t> nextIndex(0); // 原子计数器，用于线程安全地分配任务
            mutex printMutex;            // 互斥锁，用于保护 cout 输出，防止多线程同时写入导致混乱

            // 工作线程函数
            auto worker = [&](unsigned int /*threadId*/) {
                for (;;) {
                    // 原子地获取并增加索引，确保每个 IP 只被一个线程处理
                    size_t idx = nextIndex.fetch_add(1);
                    if (idx >= candidates.size()) break; // 如果所有 IP 都已分配，则线程退出

                    uint32_t candidate = candidates[idx]; // 获取要扫描的 IP 地址
                    BYTE macAddr[8] = { 0 }; // 存储 MAC 地址的缓冲区
                    ULONG macAddrLen = sizeof(macAddr);

                    // 发送 ARP 请求
                    DWORD arpRet = SendARP((IPAddr)candidate, 0, macAddr, &macAddrLen);

                    // 如果 ARP 请求成功，且返回了有效的、非零的 MAC 地址
```

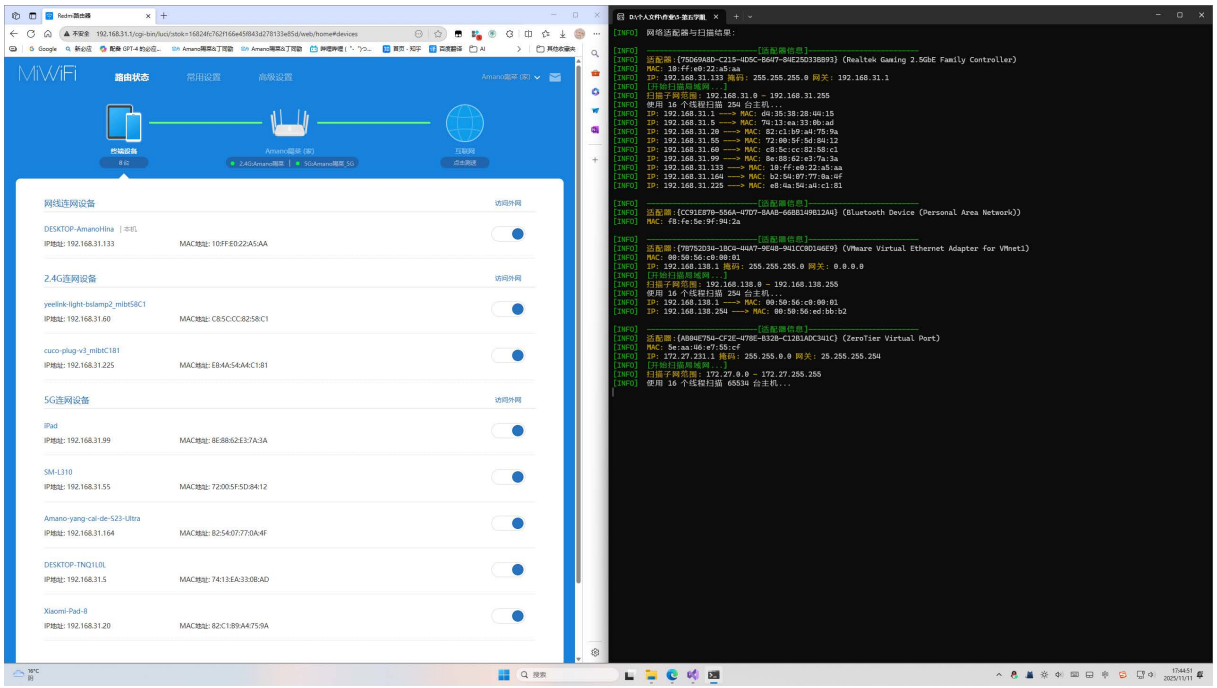
```
                    if (arpRet == NO_ERROR && macAddrLen > 0 && !isZeroMac(macAddr, macAddrLen))
{
                        in_addr a;
                        a.S_un.S_addr = candidate;
                        const char* ipBuf = inet_ntoa(a); // 将 IP 地址转回字符串格式
                        string macs = formatMac(macAddr, macAddrLen); // 格式化 MAC 地址

                        // 使用互斥锁保护控制台输出
                        lock_guard<mutex> lk(printMutex);
                        cout << InformationMsg << "\033[33mIP: \033[0m" << ipBuf << "\033[33m --->
\033[0m" << "\033[33mMAC: \033[0m" << macs << endl;
                    }
                }
                };

            // 创建并启动线程
            vector<thread> threads;
            threads.reserve(maxThreads);
            for (unsigned int t = 0; t < maxThreads; ++t) {
                threads.emplace_back(worker, t);
            }
            // 等待所有线程完成
            for (auto& th : threads) {
                th.join();
            }
        } // 遍历 IP 地址结束
    } // 遍历适配器结束

    // 清理 Winsock
    WSACleanup();
    return 0;
}
```

四、实验结果及分析



可见实体网卡 Realtek Gaming 2.5GbE Family Controller 的子网扫描结果与上游设备路由器显示连接设备相对应，结果完全正确。

五、实验中问题及分析

初始版本的程序扫描速度太慢，后增加并发扫描以提速。

初始版本输出结构不清晰美观，后增加格式化输出更美观直观。

六、心得体会

1. 对 Windows 网络编程的深入理解

本次实验让我从理论走向实践，深刻理解了 Windows 环境下网络编程的核心组件。

- **Winsock 与 IP Helper API 的协同工作：**我明白了基础的套接字功能（由 ws2_32.lib 提供）和更高级的网络信息管理功能（由 iphlpapi.lib 提供）是如何协同工作的。WSAStartup 是所有网络操作的起点，而 GetAdaptersInfo 这类 IP Helper API 则提供了直接访问网络配置信息的强大能力，避免了复杂的底层操作。
- **“两步走”的 API 调用模式：**通过使用 GetAdaptersInfo，我掌握了一种 Windows API 中常见的设计模式：先传入空指针调用一次以获取所需缓冲区的大小，然后分配内存，再调用一次以获取实际数据。这让我对 API 的健壮性和灵活性设计有了更深的认识。
- **ARP 协议的威力：**实验的核心是利用 SendARP 函数。我体会到，在局域网环境中，ARP 请求是探测主机存活状态的一种极为高效且可靠的方式。相比于 ICMP（Ping），ARP 工作在数据链路层，通常不会被防火墙拦截，因此扫描结果更为准确。

2. 对 C++现代并发编程的实践应用

这是本次实验最大的收获之一。将一个线性的扫描过程并行化，让我直观地感受到了多线程带来的巨大性能提升。

- **线程的创建与管理：**通过使用 C++11 的 <thread> 库，我能够轻松地创建和管理线程池。根据 thread::hardware_concurrency()动态调整线程数，既能充分利用 CPU 资源，又避免了创建过多线程带来的额外开销。
- **线程安全的重要性：**当多个线程同时工作时，线程安全问题立刻显现出来。

    - **原子操作 (std::atomic)**：对于任务分配的索引 nextIndex，我使用了 std::atomic。它的 fetch_add 操作保证了每个线程都能安全、无冲突地获取到一个唯一的 IP 地址进行扫描，并且效率远高于使用互斥锁。
    - **互斥锁 (std::mutex)**：对于共享资源 std::cout 的访问，我使用了 std::mutex 和 lock_guard。这确保了在任何时刻只有一个线程可以向控制台输出，从而避免了输出信息的混乱和交错。lock_guard 的 RAII 机制也让我体会到了 C++在资源管理上的优雅，它能确保锁在离开作用域时自动释放，有效防止了死锁。
- **Lambda 表达式的便利**：将工作线程的执行逻辑封装在一个 Lambda 表达式中，使得代码结构更清晰、更紧凑。

3. 对网络基础知识的巩固

这个实验强制我将书本上的网络知识应用到代码中。

- **IP 地址与子网掩码**：我通过代码实践了如何利用 IP 地址和子网掩码进行按位与（&）运算得到网络地址，以及按位或（|）与反码（~）运算得到广播地址。这个过程让我对子网划分和 IP 地址范围的理解更加透彻。
- **字节序转换**：htonl 和 ntohl 的使用让我明白了网络字节序和主机字节序的区别及其在网络编程中的重要性。所有要在网络上传输或由网络 API 处理的地址数据，都需要统一为网络字节序。

4. 软件工程实践的重要性

- **模块化与代码复用**：将 MAC 地址格式化和零 MAC 地址检查等功能封装成独立的静态函数，提高了代码的可读性和可维护性。
- **错误处理与健壮性**：代码中对每个关键 API 调用的返回值都进行了检查，并在出错时打印详细信息。这不仅是保证程序稳定运行的基础，也是调试过程中的重要帮手。
- **人性化设计**：在扫描前增加了对子网规模的判断，对于过大的子网（如超过 65534 个主机）则跳过扫描。这是一个非常实际的考量，体现了在设计程序时不仅要考虑功能实现，还要考虑真实场景下的性能和用户体验。