



南開大學
Nankai University

计算机学院
网络技术与应用实验报告

实验 2：利用 NPcap 编程捕获数据包

姓名：王旭
学号：2312166
专业：计算机科学与技术

2025 年 11 月 4 日

目录

1 了解 NPcap 的架构	2
1.1 NPcap 的架构层次	2
1.2 NPcap 的工作流程	2
2 环境配置	3
2.1 NPcap 的下载安装	3
2.2 利用 VSCode 进行环境配置	3
2.2.1 新建 c_cpp_properties.json	4
2.2.2 新建 tasks.json	4
2.2.3 新建 launch.json	5
3 NPcap 的设备列表获取方法	5
3.1 参数说明:	6
3.2 相关的关键数据结构:	6
3.3 演示程序:	6
3.4 运行结果:	7
4 NPcap 的网卡设备打开方法	8
4.1 参数说明:	8
4.2 演示程序:	8
4.3 运行结果:	9
5 NPcap 的数据包捕获方法	10
5.1 回调函数方式 (pcap_loop) 捕获多个数据包	10
5.2 单次捕获方式 (pcap_next_ex) 捕获单个数据包	11
5.3 演示程序:	11
5.4 运行结果:	12
6 最终实现本机 IP 数据报捕获与校验和对比的完整代码	13
6.1 程序基本流程	13
6.2 完整代码及注释	13
6.3 运行结果	15

1 了解 NPcap 的架构

NPcap 是一个运行在 Windows 平台上的专业数据包捕获架构，它是在 WinPcap 基础上发展而来的现代化替代品。NPcap 提供了在 Windows 操作系统上进行网络数据包捕获和注入的能力，是网络分析工具如 Wireshark 的基础组件。

1.1 NPcap 的架构层次

(1) 应用层 (Application Layer)

1. 用户程序：使用 NPcap 库开发的网络分析工具（如我们的实验程序）
2. API 接口：通过调用 NPcap 提供的编程接口来实现数据包捕获功能

(2) NPcap 库层 (NPcap Library Layer)

1. wpcap.dll：提供与 Unix 平台 libpcap 兼容的 API 接口
2. packet.dll：处理与内核驱动程序的低级通信
3. 用户态与内核态的桥梁：负责将应用程序的请求转换为内核操作

(3) 内核驱动层 (Kernel Driver Layer)

1. NPcap 驱动程序 (NPcap.sys)：运行在 Windows 内核空间的核心组件
2. 网络协议栈挂钩：通过 NDIS(Network Driver Interface Specification) 接口与网络协议栈交互
3. 数据包过滤引擎：使用 BPF(Berkeley Packet Filter) 技术进行高效的数据包过滤

(4) 网络协议栈层 (Network Stack Layer)

1. Windows 网络协议栈：操作系统自带的 TCP/IP 协议实现
2. NDIS 接口：网络驱动程序接口规范，提供标准化的网络访问接口

1.2 NPcap 的工作流程

1. NPcap 首先进行应用程序的初始化。程序首先会调用 NPcap 提供的函数来获取当前计算机上所有可用的网络接口列表，比如有线网卡、无线网卡等，并展示给用户选择。
2. 用户选择要监听的网卡后，程序会尝试以“混杂模式”打开该设备；在这种模式下，网卡会接收所有流经其网络的数据包，而不仅仅是发给本机的数据包，这是实现网络监听的关键。
3. 接下来，程序可以设置一个可选的过滤器，例如只捕获 IP 数据包，这样可以提高效率，避免处理不相关的网络流量。设置完成后，程序便进入一个核心的循环，在这个循环中，它会不断地向 NPcap 查询是否有新的数据包到达。一旦有数据包被捕获，NPcap 就会将其返回给应用程序。
4. 应用程序收到数据包的原始字节流后，就可以根据网络协议的格式（如以太网帧头、IP 报头）进行解析，提取出源 MAC 地址、目标 MAC 地址、源 IP 地址等关键信息。这个捕获和解析的过程会持续进行，直到用户主动停止程序。

5. 最后，在程序退出前，它会关闭之前打开的网络设备并释放所有相关资源，确保操作系统的稳定性和资源不被浪费。

2 环境配置

2.1 NPcap 的下载安装

进入官网 <https://npcap.com/#download>，下载 NPcap 和 NPcap-SDK。

The free version of Npcap may be used (but not externally redistributed) on up to 5 systems ([free license details](#)). It may also be used on unlimited systems where it is only used with [Nmap](#), [Wireshark](#), and/or [Microsoft Defender for Identity](#). Simply run the executable installer. The full source code for each release is available, and developers can build their apps against the SDK. The improvements for each release are documented in the [Npcap Changelog](#).

- [Npcap 1.84 installer](#) for Windows 7/2008R2, 8/2012, 8.1/2012R2, 10/2016, 2019, 11 (x86, x64, and ARM64).
- [Npcap SDK 1.15 \(ZIP\)](#).
- [Npcap 1.84 debug symbols \(ZIP\)](#).
- [Npcap 1.84 source code \(ZIP\)](#).

The latest development source is in our [Github source repository](#). Windows XP and earlier are not supported; you can use [WinPcap](#) for these versions.

图 2.1: 进入官网下载界面

点击 *Npcap 1.84 installer* 链接，将 npcap-1.84.exe 下载到本地，然后点击 npcap-1.84.exe 程序点击 next 将 NPcap 完成安装。

然后点击 *Npcap SDK 1.15 (ZIP)* 链接，下载并解压 npcap-sdk-1.15.zip，得到下面的 npcap-sdk-1.15 文件夹：

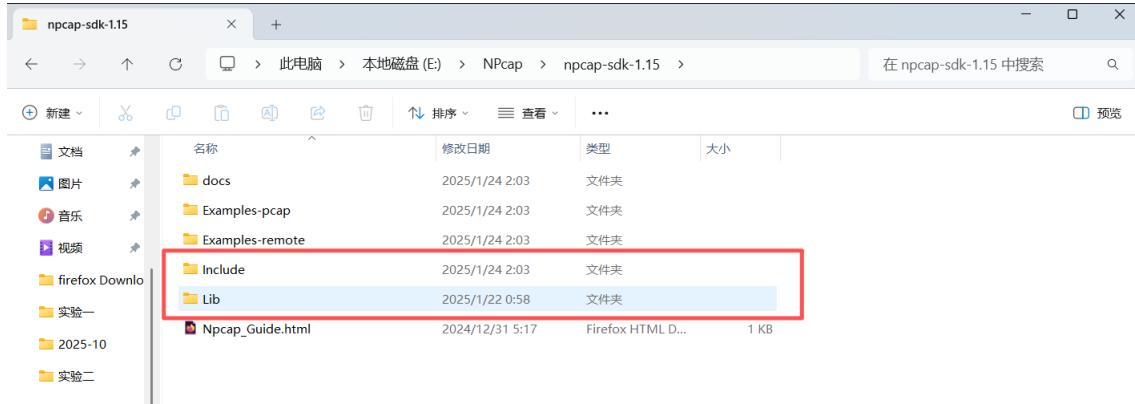


图 2.2: 解压得到 npcap-sdk-1.15 文件夹

2.2 利用 VSCode 进行环境配置

注意 VSCode 中安装 C/C++ 扩展。

首先将上面图中的红框圈出的两个文件夹 Include 和 Lib 复制到我们的项目目录下，比如我新建了一个 D:\NETWORK 文件夹，将来想在该 NETWORK 文件夹下进行代码的编写，那么我就把上面的两个 Include 和 Lib 文件夹复制到 NETWORK 文件夹下。

在 NETWORK 新建一个 **src** 文件夹，用于存放我们将来的代码。

再在 NETWORK 中新建一个 **.vscode** 文件夹，用于存放我们下面的三个配置文件。

2.2.1 新建 c_cpp_properties.json

c_cpp_properties.json

```

1  {
2      "configurations": [
3          {
4              "name": "Win32", // 配置名称
5              "includePath": [ // 头文件搜索路径（最重要的配置！）
6                  "${workspaceFolder}/Include", // NPcap头文件路径
7                  "${workspaceFolder}/**" // 工作区所有子目录
8              ],
9              "defines": [ // 预定义宏（相当于代码中的 #define）
10                  "_DEBUG", // 调试模式定义
11                  "UNICODE", // 使用 Unicode 字符集
12                  "_UNICODE" // 使用 Unicode 字符集
13              ],
14              "compilerPath": "D:/TDM-GCC/bin/g++.exe", // 编译器路径
15              "cStandard": "c17", // C 语言标准
16              "cppStandard": "c++17", // C++ 语言标准
17              "intelliSenseMode": "windows-gcc-x64" // 智能感知模式
18          }
19      ],
20      "version": 4
21  }

```

2.2.2 新建 tasks.json

tasks.json

```

1  {
2      "version": "2.0.0",
3      "tasks": [
4          {
5              "label": "build-npcap", // 任务名称（在 launch.json 中引用）
6              "type": "shell", // 在终端中执行
7              "command": "g++", // 执行的命令
8              "args": [ // 命令行参数（对应 g++ 的参数）
9                  "-g", // 生成调试信息
10                 "src/main.cpp", // 源文件
11                 "-o", "bin/main.exe", // 输出文件
12                 "-I", "${workspaceFolder}/Include", // 包含目录
13                 "-L", "${workspaceFolder}/Lib/x64", // 库目录
14                 "-l", "Packet", // 链接 Packet.lib
15                 "-l", "wpcap", // 链接 wpcap.lib
16                 "-l", "ws2_32", // 链接 Windows Socket 库
17                 "-l", "iphlpapi", // 链接 IP 帮助库
18                 "-static", // 静态链接
19                 "-fexec-charset=UTF-8", // 关键：执行字符集为 UTF-8

```

```

20         "-finput-charset=UTF-8"      // 关键：输入字符集为UTF-8
21     ],
22     "group": {
23       "kind": "build",           // 属于构建任务组
24       "isDefault": true        // 默认构建任务 (Ctrl+Shift+B)
25     },
26     "problemMatcher": ["$gcc"], // 错误信息匹配器
27     "detail": "编译 NPcap 项目"
28   }
29 ]
30 }
```

2.2.3 新建 launch.json

launch.json

```

1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "name": "调试 NPcap 程序",          // 调试配置名称
6       "type": "cppdbg",                  // C++ 调试
7       "request": "launch",               // 启动调试
8       "program": "${workspaceFolder}/bin/main.exe", // 调试的程序
9       "args": [],                      // 命令行参数
10      "stopAtEntry": false,             // 不在入口点停止
11      "cwd": "${workspaceFolder}",       // 工作目录
12      "environment": [],              // 环境变量
13      "externalConsole": true,         // 使用外部控制台（重要！）
14      "MIMode": "gdb",                // 使用 GDB 调试器
15      "preLaunchTask": "build-npcap" // 调试前先执行构建任务
16    }
17  ]
18 }
```

这样我们的环境就配置完成了，可以自己利用几个测试代码进行测试，也可以利用后面的获取设备列表的代码进行初步测试。

3 NPcap 的设备列表获取方法

设备列表获取是 NPcap 编程的第一步，它的核心目的是让程序能够发现系统中所有可用的网络接口。在 Windows 系统中，一个计算机可能有多个网络接口，比如有线网卡、无线网卡、虚拟网卡等。NPcap 通过特定的函数来枚举这些接口，并为每个接口提供详细的信息，方便我们选择要监控的目标。

NPcap 中用于获取设备列表的核心函数 pcap_findalldevs

```
1 int pcap_findalldevs(pcap_if_t **alldevsp, char *errmsg);
```

3.1 参数说明:

1. **alldevsp**: 这是一个指向设备链表头指针的指针，函数执行成功后，会通过这个参数返回设备链表的头指针
2. **errbuf**: 错误信息缓冲区，如果函数执行失败，会在这里存储错误描述
3. **返回值**: 成功返回 0，失败返回-1

3.2 相关的关键数据结构:

pcap_if_t 结构体（设备信息结构）

```

1 struct pcap_if {
2     struct pcap_if *next;           // 指向下一个设备的指针
3     char *name;                   // 设备名称（用于打开设备）
4     char *description;            // 设备描述（人类可读的描述）
5     struct pcap_addr *addresses; // 设备地址列表
6     u_int flags;                 // 设备标志位
7 };

```

pcap_addr_t 结构体（地址信息结构）

```

1 struct pcap_addr {
2     struct pcap_addr *next;        // 下一个地址
3     struct sockaddr *addr;        // IP地址
4     struct sockaddr *netmask;     // 子网掩码
5     struct sockaddr *broadaddr;   // 广播地址
6     struct sockaddr *dstaddr;    // 目标地址（点对点连接）
7 };

```

库函数会构建一个 pcap_if_t 结构的链表。链表中的每个节点代表一个网络接口，包含了该接口的所有相关信息。这个链表采用单向链表的形式组织，通过 next 指针连接各个设备。pcap_findalldevs() 函数会将链表的头指针通过参数返回，调用者就可以通过遍历这个链表来查看所有的可用设备。

3.3 演示程序:

设备列表获取方式的演示程序 device_list.cpp

```

1 #include <iostream>
2 #include "pcap.h"
3 int main() {
4     pcap_if_t* alldevs;
5     pcap_if_t* device;
6     char errbuf[PCAP_ERRBUF_SIZE];
7     int count = 0;
8     std::cout << "NPcap设备列表获取演示" << std::endl;
9     std::cout << "===== " << std::endl;
10    // 获取设备列表

```

```

11     if (pcap_findalldevs(&alldevs, errbuf) == -1) {
12         std::cout << "错误: " << errbuf << std::endl;
13         return 1;
14     }
15     // 显示设备列表
16     for (device = alldevs; device != nullptr; device = device->next) {
17         count++;
18         std::cout << count << ". " << device->name;
19         if (device->description) {
20             std::cout << " - " << device->description;
21         }
22         std::cout << std::endl;
23     }
24     std::cout << "共找到 " << count << " 个设备" << std::endl;
25     // 释放资源
26     pcap_freealldevs(alldevs);
27     system("pause");
28     return 0;
29 }
```

代码详见: 设备列表获取方式的演示程序 device_list.cpp

3.4 运行结果:

我们采用如下命令进行编译运行:

编译运行指令

```

1 # 编译
2 g++ src/device_list.cpp -o bin/device_list.exe -I Include -L Lib/x64 -lPacket -lwpcap
   -lws2_32 -liphlpapi -fexec-charset=UTF-8 -finput-charset=UTF-8
3 # 运行
4 ./bin/device_list.exe
```

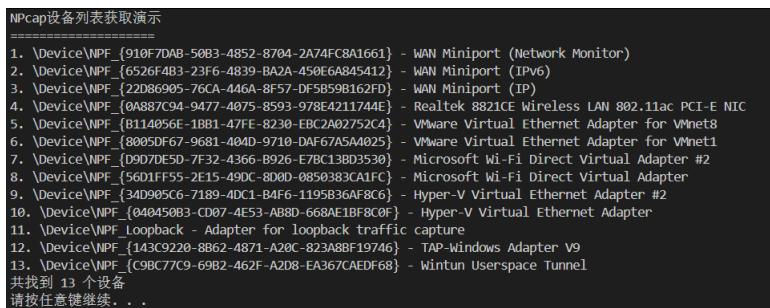


图 3.3: NPcap 设备列表获取演示

可以看到 NPcap 成功检测到系统中的所有网络接口，共找到 13 个网络设备，这包括物理网卡、虚拟网卡、VPN 适配器等。

4 NPcap 的网卡设备打开方法

在成功获取设备列表后，下一步就是打开选定的网络接口设备。打开设备是 NPcap 编程中的关键步骤，它建立了应用程序与网络接口之间的通信通道。这个过程涉及到设备句柄的创建、工作模式的设置以及必要的参数配置。

NPcap 中用于网卡设备打开的函数 pcap_open_live

```
1 pcap_t* pcap_open_live(const char* device, int snaplen, int promisc, int to_ms, char*
  errbuf);
```

4.1 参数说明:

1. **device**: 要打开的设备名称字符串，这个名称来自 pcap.findalldevs() 获取的设备列表中的 name 字段。
2. **snaplen**: 指定每个数据包要捕获的最大长度 (字节数)。
3. **promisc**: 是否启用混杂模式，0 表示非混杂模式，只捕获发给本机的数据包；1 表示混杂模式，捕获所有流经网络接口的数据包
4. **to_ms**: 指定在捕获数据包时的超时时间 (ms)，设置为 0 表示无超时 (阻塞模式)。
5. **errbuf**: 错误信息缓冲区，用于接收错误描述的字符数组
6. **返回值**: 成功：返回 pcap_t* 类型的设备句柄指针；失败则返回 NULL，错误信息存储在 errbuf 中

当调用 pcap_open_live() 函数时，NPcap 在底层执行一系列复杂的操作。首先，函数会验证传入的设备名称是否有效，检查该设备是否确实存在于系统中并且支持数据包捕获功能。这个验证过程涉及到与 NPcap 驱动程序的通信。

验证通过后，NPcap 会创建一个 pcap_t 结构体实例，这个结构体是 NPcap 的核心数据结构，它包含了设备的所有状态信息、配置参数以及各种回调函数指针。创建这个结构体相当于为后续的数据包捕获操作准备了完整的工作环境。

接下来，函数会根据传入的参数配置设备的工作模式。其中最重要的就是混杂模式的设置。在非混杂模式下，网络接口卡只会接收目的地为本机的数据包；而在混杂模式下，网卡会接收所有流经网络的数据包，这对于网络监控和分析至关重要。这个模式的设置需要操作系统层面的权限配合。

同时，函数还会设置数据包的截断长度和超时时间的设置。

最后，函数会初始化各种内部缓冲区和管理结构，为实际的数据包捕获做好准备。整个过程完成后，返回的设备句柄就成为了后续所有数据包操作的基础。

4.2 演示程序:

网卡设备打开的演示程序 open_device.cpp

```
1
2 # 在前面的设备列表获取之后，添加打开网卡设备的部分
3 #include <iostream>
```

```

4 #include "pcap.h"
5
6 int main() {
7     .....
8     // 3. 选择要打开的设备 (这里选择第4个设备, Realtek无线网卡)
9     std::cout << "选择要打开的设备 (输入设备编号) : ";
10    std::cin >> selected_device;
11    // 4. 打开选中的设备
12    device_handle = pcap_open_live(
13        device->name,      // 设备名称
14        65536,            // 捕获长度: 64KB
15        1,                // 混杂模式: 开启
16        1000,             // 超时时间: 1秒
17        errbuf            // 错误缓冲区
18    );
19    if (device_handle == nullptr) {
20        std::cout << "打开设备失败: " << errbuf << std::endl;
21        std::cout << "提示: 可能需要管理员权限运行程序" << std::endl;
22    } else {
23        std::cout << "设备打开成功!" << std::endl;
24        std::cout << "设备句柄: " << device_handle << std::endl;
25        // 5. 关闭设备
26        pcap_close(device_handle);
27        std::cout << "设备已关闭" << std::endl;
28    }
29    // 6. 释放设备列表
30    pcap_freealldevs(alldevs);
31    .....
32 }

```

代码详见: 网卡设备打开的演示程序 open_device.cpp

4.3 运行结果:

我们采用如下命令进行编译运行:

编译运行指令

```

1 # 编译
2 g++ src/open_device.cpp -o bin/open_device.exe -I Include -L Lib/x64 -lPacket -lwpcap
   -lws2_32 -liphpapi -fexec-charset=UTF-8 -finput-charset=UTF-8
3 # 运行
4 ./bin/open_device.exe

```

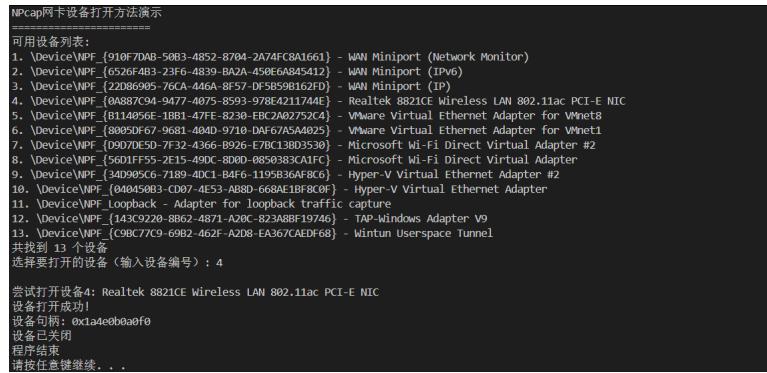


图 4.4: NPcap 网卡设备打开演示

5 NPcap 的数据捕获方法

数据包捕获是 NPcap 编程的核心环节，它实现了从网络接口实时获取数据包的功能。在成功打开设备后，我们需要选择合适的捕获方法来处理流经网络接口的数据。NPcap 提供了几种不同的捕获方式，每种方式都有其适用的场景和特点。

5.1 回调函数方式 (pcap_loop) 捕获多个数据包

NPcap 中用于捕获多个数据包的函数 pcap_loop

```
1 int pcap_loop(pcap_t* p, int cnt, pcap_handler callback, u_char* user);
```

参数说明:

1. **p**: 已打开的设备句柄，类型是我们前面打开设备后返回的 pcap_t 类型
2. **cnt**: 要捕获的数据包数量 (-1 表示无限捕获)
3. **callback**: 数据包到达时的回调处理函数
4. **user**: 传递给回调函数的用户数据

相关的回调函数原型:

相关的回调函数原型 packet_handler

```
1 void packet_handler(u_char* user, const struct pcap_pkthdr* header, const u_char*
packet);
```

当使用 pcap_loop() 函数启动数据包捕获时，NPcap 会进入一个专门的处理循环。在这个循环中，程序会等待网络接口上有数据包到达。一旦有数据包到达，NPcap 驱动会将其从内核空间复制到用户空间的缓冲区中。

数据包被成功复制后，NPcap 会调用预先注册的回调函数。这个回调函数会接收到三个关键参数：用户数据指针、数据包头信息和数据包内容的指针。用户数据指针允许我们在多次回调调用之间保持状态信息；数据包头包含了时间戳、数据包长度等元信息；数据包内容指针则指向实际的网络数据。

在回调函数内部，我们可以对数据包进行各种处理操作，比如协议解析、统计分析、内容检查等。处理完成后，函数返回，控制权交还给 NPcap，等待下一个数据包的到来。

如果设置了捕获数量限制，当达到指定数量后循环会自动终止。否则，捕获过程会一直持续，直到调用 pcap_breakloop() 函数或发生错误。

5.2 单次捕获方式 (pcap_next_ex) 捕获单个数据包

NPcap 中用于捕获单个数据包的函数 pcap_next_ex

```
1 int pcap_next_ex(pcap_t* p, struct pcap_pkthdr** pkt_header, const u_char** pkt_data);
```

相关的数据包的关键数据结构：

pcap_pkthdr 结构体（数据包头信息）

```
1 struct pcap_pkthdr {
2     struct timeval ts; // 时间戳（捕获时间）
3     bpf_u_int32 caplen; // 实际捕获的长度
4     bpf_u_int32 len; // 数据包原始长度
5 };
```

pcap_next_ex() 用于捕获单个数据包，返回值为 1 表示成功捕获数据包；为 0 表示超时，没有数据包；为 -1 表示错误，-2 表示遇到文件结束。

5.3 演示程序：

数据包捕获的演示程序 packet_capture.cpp

```
1 #include <iostream>
2 #include <iomanip>
3 #include "pcap.h"
4 // 以太网帧头结构
5 struct ethernet_header {
6     u_char dest_mac[6]; // 目的 MAC 地址
7     u_char src_mac[6]; // 源 MAC 地址
8     u_short ether_type; // 类型/长度字段
9 };
10 // 数据包处理回调函数
11 void packet_handler(u_char *user, const struct pcap_pkthdr *header, const u_char
12 *packet) {
13     .....
14 }
15 int main() {
16     std::cout << "设备打开成功，开始捕获数据包..." << std::endl;
17     std::cout << "将捕获5个数据包，请稍候..." << std::endl << std::endl;
18     // 开始捕获数据包（捕获5个后退出）
19     int packet_count = 5;
```

```

19     int result = pcap_loop(device_handle, packet_count, packet_handler, nullptr);
20     if (result == -1) {
21         std::cout << "捕获过程中出错: " << pcap_geterr(device_handle) << std::endl;
22     } else if (result == 0) {
23         std::cout << "成功捕获" << packet_count << "个数据包" << std::endl;
24     } else {
25         std::cout << "捕获被中断" << std::endl;
26     }
27     // 清理资源
28     pcap_close(device_handle);
29     pcap_freealldevs(alldevs);
30     std::cout << "程序结束" << std::endl;
31     system("pause");
32     return 0;
33 }
```

代码详见: 数据包捕获的演示程序 `packet_capture.cpp`

5.4 运行结果:

我们采用如下命令进行编译运行:

编译运行指令

```

1 # 编译
2 g++ src/packet_capture -o bin/packet_capture.exe -I Include -L Lib/x64 -lPacket
   -lwpcap -lws2_32 -liphlpapi -fexec-charset=UTF-8 -finput-charset=UTF-8
3 # 运行
4 ./bin/packet_capture.exe
```

```

NPcap数据包捕获方法演示
=====
选择设备: Realtek 8821CE Wireless LAN 802.11ac PCI-E NIC
设备打开成功, 开始捕获数据包...
将捕获5个数据包, 请稍候...

--- 数据包 #1 ---
时间戳: 1762346110秒
数据包长度: 451字节
源MAC地址: ce:1b:25:5d:58:5c
目的MAC地址: 01:00:5e:00:00:fb
类型/长度: 0x800 (IPv4)

--- 数据包 #2 ---
时间戳: 1762346110秒
数据包长度: 471字节
源MAC地址: ce:1b:25:5d:58:5c
目的MAC地址: 33:33:00:00:00:fb
类型/长度: 0x86dd (IPv6)

```

图 5.5: NPcap 捕获数据包演示

定义了以太网帧头结构的结构体，并实现 `packet_handler` 函数作为回调函数，NPcap 在捕获到每个数据包时自动调用此函数，将 MAC 地址格式化输出，最后我们可以看到实际的网络流量并识别不同类型的网络协议 (IPv4、IPv6、ARP 等)。

6 最终实现本机 IP 数据报捕获与校验和对比的完整代码

6.1 程序基本流程

我们首先来梳理一下实现本机 IP 数据报捕获与校验和对比程序的基本流程:

1. **初始化**: 加载库、初始化 Winsock
2. **设备枚举**: 使用 `pcap.findalldevs(&alldevs, errbuf)` 获取所有可用网络设备。
3. **设备选择**: 选择特定网络接口
4. **设备打开**: 使用 `pcap.open_live()` 以混杂模式打开选定的设备
5. **数据包捕获**: 使用 `pcap.loop(device_handle, 5, packet_handler, nullptr)` 循环捕获数据包, 调用回调函数处理
6. **数据包解析**: 解析以太网帧头和 IP 头部
7. **校验和计算**: 计算 IP 头部校验和并与原始值比较
8. **结果显示**: 输出数据包信息和校验结果
9. **资源释放**: 关闭设备、释放内存

6.2 完整代码及注释

完整的处理程序 final.cpp 的关键代码

```

1 // 略去结构体定义与辅助函数的实现等
2 #include <iostream> // 输入输出流, 用于控制台输出
3 #include <iomanip> // 输入输出格式化, 用于设置输出格式
4 #include <winsock2.h> // Windows Socket API, 用于网络编程
5 #include "pcap.h" // NPcap 库头文件, 用于数据包捕获
6 #pragma comment(lib, "ws2_32.lib") // 链接 Windows Socket 库
7 // 以太网帧头结构定义 (共14字节)
8 struct ethernet_header {
9     u_char dest_mac[6]; // 目的MAC地址 (6字节)
10    u_char src_mac[6]; // 源MAC地址 (6字节)
11    u_short ether_type; // 类型/长度字段 (2字节), 0x0800表示IPv4
12 };
13 // 主函数: 程序入口点
14 int main() {
15     pcap_if_t* alldevs; // 设备列表头指针
16     pcap_if_t* device; // 当前设备指针
17     pcap_t* device_handle; // 设备句柄
18     char errbuf[PCAP_ERRBUF_SIZE]; // 错误信息缓冲区
19     int count = 0; // 设备计数器
20     int selected_device = 4; // 选择的设备编号 (根据实际情况调整)
21
22     // 获取网络设备列表

```

```

23     if (pcap.findalldevs(&alldevs, errbuf) == -1) {
24         std::cout << "错误: " << errbuf << std::endl;
25         WSACleanup(); // 清理Winsock资源
26         return 1; // 退出程序
27     }
28     // 选择指定编号的设备 (这里选择第4个设备)
29     device = alldevs;
30     for (int i = 1; i < selected_device && device != nullptr; i++) {
31         device = device->next; // 遍历设备链表
32     }
33     // 检查设备选择是否有效
34     if (device == nullptr) {
35         std::cout << "设备选择无效" << std::endl;
36         pcap_freealldevs(alldevs); // 释放设备列表
37         WSACleanup(); // 清理Winsock资源
38         return 1; // 退出程序
39     }
40     // 显示选择的设备信息
41     std::cout << "选择设备: " << (device->description ? device->description :
42         "无描述") << std::endl;
43     // 打开网络设备进行数据包捕获
44     // 参数: 设备名, 最大捕获长度, 混杂模式, 超时时间, 错误缓冲区
45     device_handle = pcap_open_live(device->name, 65536, 1, 5000, errbuf);
46     if (device_handle == nullptr) {
47         std::cout << "打开设备失败: " << errbuf << std::endl;
48         pcap_freealldevs(alldevs); // 释放设备列表
49         WSACleanup(); // 清理Winsock资源
50         return 1; // 退出程序
51     }
52     // 开始捕获数据包
53     std::cout << "开始捕获IPv4数据包(最多等待5秒)..." << std::endl;
54     std::cout << "请进行网络活动以产生更多数据包..." << std::endl << std::endl;
55     // 启动数据包捕获循环
56     // 参数: 设备句柄, 捕获数量, 回调函数, 用户数据
57     pcap_loop(device_handle, 5, packet_handler, nullptr);
58     // 清理资源
59     pcap_close(device_handle); // 关闭设备
60     pcap_freealldevs(alldevs); // 释放设备列表
61     WSACleanup(); // 清理Winsock资源
62     std::cout << "程序结束" << std::endl;
63     system("pause"); // 暂停等待用户按键
64     return 0; // 正常退出
}

```

代码详见: 完整的处理程序 final.cpp

6.3 运行结果

```

PS D:\VS_Code\NETWORK> ./bin/final.exe
IP数据报捕获与校验和验证
=====
说明：程序将尝试捕获IPv4数据包，网络流量会影响捕获数量
建议在运行程序的同时进行网络活动（如浏览网页）

选择设备: Realtek 8821CE Wireless LAN 802.11ac PCI-E NIC
开始捕获IPv4数据包(最多等待5秒)...
请进行网络活动以产生更多数据包...

--- IP数据报 #1 ---
源MAC地址: f0:a6:54:8f:8d:d1
目的MAC地址: 00:00:5e:00:01:fe
源IP地址: 10.136.35.139
目的IP地址: 140.82.113.22
协议类型: TCP
数据包总长度: 52字节
原始校验和: 0xe2ec
计算校验和: 0xece2 (网络字节序: 0xe2ec)
校验结果: ✓ 匹配 (数据包完整)

--- IP数据报 #2 ---
源MAC地址: 00:00:5e:00:01:fe
目的MAC地址: f0:a6:54:8f:8d:d1
源IP地址: 140.82.113.22
目的IP地址: 10.136.35.139
协议类型: TCP
数据包总长度: 40字节
原始校验和: 0x76e8
计算校验和: 0xe876 (网络字节序: 0x76e8)
校验结果: ✓ 匹配 (数据包完整)

```

图 6.6: 最终实现本机 IP 数据报捕获与校验和对比运行结果

可以看到我们成功捕获到了数据报并对其进行了数据报的校验和字段的比较，并且对于输出结果，我们成功输出了源 MAC 地址、目的 MAC 地址，源 IP、目的 IP 地址、校验和字段的数值、程序计算出的校验和数值。并且发现，所有 5 个 IPv4 数据包的校验和都完全匹配，这说明，IP 数据包在传输过程中没有损坏，我们的校验和计算算法实现是正确的，并且处理了网络字节序与主机 x86 字节序的转换问题，成功完成了实现目标。