计算机网络实践教程

周芳　　张爽

北京科技大学计算机系

2019 年 5 月

说明及图例

❖ 命令行格式说明：

粗体：命令及关键字，命令中必须包含。

斜体：采用斜体字表示命令行参数，命令中用实际值替代。

[ ] ：用[ ] 括起来的部分为命令可选部分。

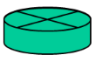
{ x | y | … }：从 x、y、…中选取一个。

[x | y | …]：从 x、y、…中选取一个或者不选。

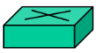
!：由感叹号!开始到行尾的内容表示注释或说明信息。

❖ 图例

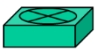
主机（PC机），网络结点



路由器，网络结点



二层交换机（L2 Switch）

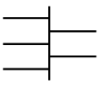


三层交换机（L3 Switch），可以成为网络结点

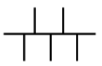
网络（或子网）

广域网，V.35线缆

以太网链路，连接两台设备



以太网链路（交换机或集线器）



以太网链路（交换机或集线器）

目　　录

第三部分　网络应用程序开发

第6章　网络接口编程原理

网络应用程序是为了实现某种具体的网络应用，网络应用是最贴近普通用户实际生活的计算机应用方式之一，远程文件传输、电子邮件、万维网、即时讯息、IP 电话、视频会议等网络应用既方便了人们的日常生活，又改变了人们的工作交流方式，极大地促进了人类社会的发展。

网络应用程序是运行在网络终端的程序，是网络应用层协议的具体实现。网络协议栈的应用层有很多针对不同应用的协议，如 HTTP、FTP、SMTP 等等。每一种应用层协议都规定了应用程序之间进行数据交换所遵循的规则。基于某种应用层的协议，可以有不同的网络应用程序。比如我们日常使用的浏览新闻的浏览器，有的用户习惯 Google 浏览器，有的喜欢火狐浏览器，还有的就使用 Windows 操作系统自带的 IE 浏览器，这些浏览器都是基于应用层协议 HTTP/HTTPS（超文本传输协议/安全的超文本传输协议）开发的网络应用程序。HTTP/HTTPS 规定了用户的客户机和提供 WEB 服务的服务器之间的网络通信规则，只要遵循该规则，不管是什么语言开发、什么界面，只要是能提供用户使用 WEB 服务，就完成了网络应用程序的功能。因此，针对某种具体的应用层协议，可以开发不同的网络应用程序。必须指出，有些网络应用程序自身就包含了某个应用层协议的具体实现，这也是网络应用程序的一种，这种网络应用程序直接规定了各网络主机之间在应用层的通信规则及程序实现。

6.1 网络协议

为讨论方便起见，本书涉及的计算机网络协议栈采取为五层协议栈模型（以因特网为基础），协议栈模型见图 6-1。网络协议栈的 5 层模型，从下向上为：物理层、数据链路层、网络层、传输层和应用层，每层都有对应层的网络协议负责同层的通信功能；上层协议使用下层协议实现的服务来完成本层协议功能，即层与层之间使用通过服务来连接。

网络应用程序运行在应用层之上，对于网络应用程序员来讲，不需要考虑相互通信的应用数据怎么通过网络协议栈的物理层、数据链路层、网络层及运输层，只需要考虑使用何种运输层的协议，应用进程之间如何交互实现应用层协议，完成网络应用功能即可（也有的应用程序直接调用网络层的协议）。应用层以下的网络通信功能由网络协议栈来实现，应用程序员只需要指定和网络协议栈的接口即可。这样大大减少了应

用程序员的工作量，使其专注于网络应用功能的设计和实现，提高网络应用程序的开发能力。

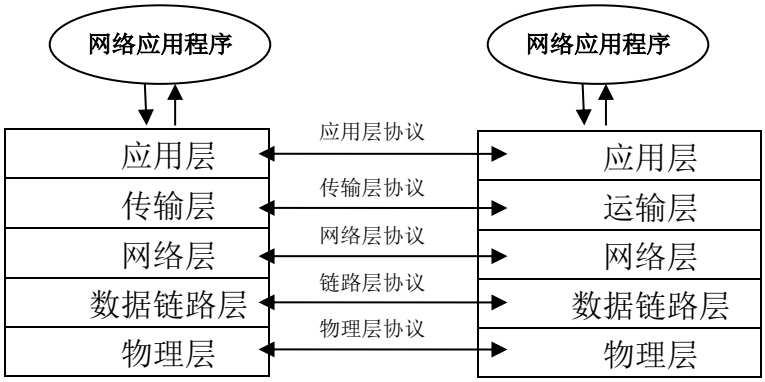


图 6-1 网络协议栈

6.1.1 TCP/IP 协议族

如图 6-1 所示， 网络协议栈每层都有对应层的协议来保证该层功能的实现，而且每层协议都可以不同的具体的协议，因此有多种多样的网络协议。学习网络协议的时候，一定要注意该协议位于网络协议栈的哪层？具体功能是什么？该协议的服务对象是谁。以现在普遍使用的 TCP/IP 协议族为例，其对应网络协议栈的具体网络协议的关系见图 6-2。

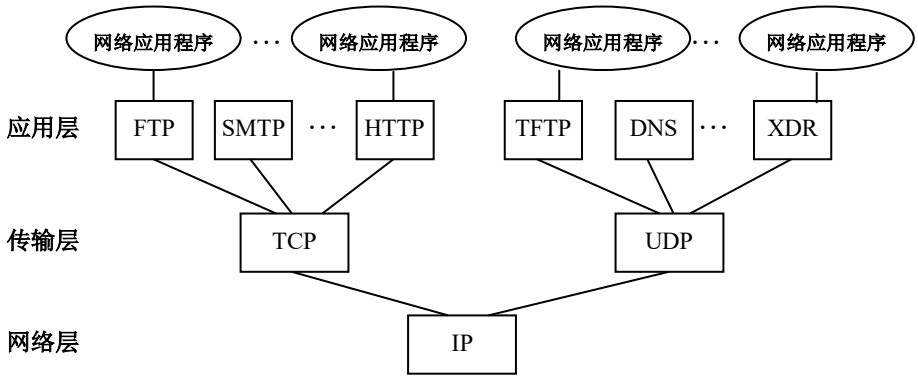


图 6-2 TCP/IP 协议族

网络层的 IP 协议负责完成主机在网络层的地址识别和主机通信功能；传输层的 TCP 协议和 UDP 协议在下层 IP 协议提供的服务基础上，都能完成传输层的进程识别和复用分用功能，TCP 协议提供可靠传输服务，而 UDP 协议提供不可靠传输服务，由

上层应用层协议根据需要选择使用具体哪种传输层协议；应用层的 FTP 协议负责网络文件传输，SMTP 协议负责邮件发送，POP 负责邮件接收，…，每种应用层协议都对应一种具体的网络功能，丰富多样的网络应用对应有众多的应用层协议。

应用进程需网络传送的数据通过网络协议栈时，为了实现对该层的网络协议功能，会在数据的首部（有的是首尾都有，如数据链路层）增加附加协议控制的信息，这个过程称之为封装；与之对应的，到达数据接收方时，数据从下向上有一个对应层协议功能实现的过程，并在此过程中去掉该层附加的首部信息，该过程称之为解封。图 6-3 展示了应用数据的封装和解封的过程。

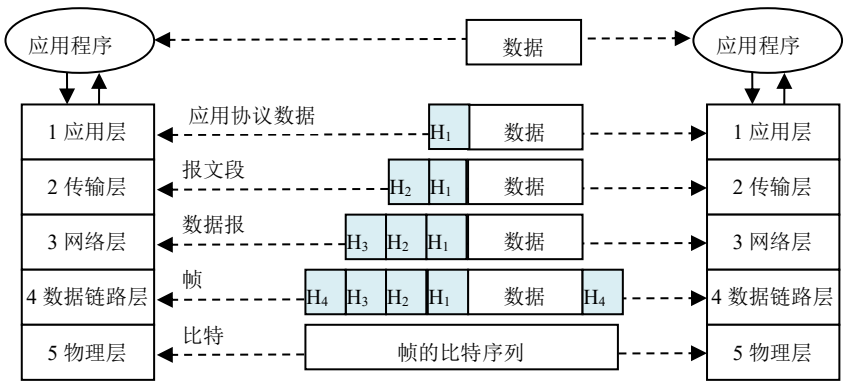


图 6-3 网络应用程序数据封装和解封过程

在实际的网络链路上传输的是经过调制后的数据链路层的帧的比特序列。一个具有真正分组意义的数据传输是在数据链路层，这个层次上的数据分组称之为“帧”，“帧”包含的内容如图 6-4 所示，包含有帧的首部、网络层协议首部、传输层协议首部、应用层协议首部、应用程序数据和帧的尾部信息。对于数据链路层来说，只看到帧的首部和尾部信息，并根据这些信息进行数据链路层的协议控制，以向网络层提供服务。至于帧的首部和尾部之间的信息（包含网络层协议首部、传输层协议首部和应用层协议首部），数据链路层都认为是网络传输的“数据”，不能对其进行任何操作，只是忠实地按照规则进行传输。

图 6-4 网络传输的数据分组组成

具体到 TCP/IP 协议族，假设应用程序是浏览器程序，采用以太网的网络适配器，则网络上传输的数据分组“帧”的组成信息如下：（各层首部均不使用选项字段）

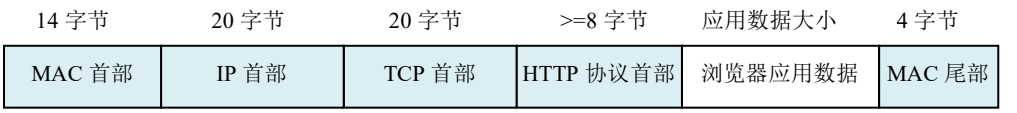


图 6-5 网络传输的浏览器应用程序的数据分组组成

浏览器应用程序的功能是和 WEB 服务器通信，取得 WEB 服务器上的信息资源（文本和图像等），其采用的应用层协议是 HTTP/HTTPS（超文本传输协议/安全的超文本传输协议），传输层使用保证可靠传输的 TCP 协议，网络层选用不可靠的 IP 协议，数据链路层采用以太网 MAC 帧协议。网络上传输的数据分组的第一个字节是 MAC 帧首部字段的第一个字节，14 个字节之后是 IP 首部的第一个字节，20 个字节之后是 TCP 首部的第一个字节，又 20 个字节之后是 HTTP 协议首部的第一个字节，一直到 HTTP 协议首部字段完全结束后（HTTP 有多个首部字段选项，根据具体需要选择具体首部字段）才是浏览器应用程序真正需要和 WEB 服务器通信的应用层数据。

6.1.2 网络程序体系结构

网络应用程序开发必须考虑到运行在网络端系统上的应用程序之间的通信模式，即网络应用进程的体系结构。现代网络应用程序的主流体系结构有两种：C/S 客户服务器结构、P2P 对等结构。

C/S 客户服务器结构中，网络通信的两个端系统中，有一个总是打开的服务器，它能接受客户端系统的网络连接请求并提供服务，如 WEB 服务器接收客户连接请求，并返回其请求内容；另一个则是客户端系统，负责连接服务器并提出请求，如浏览器程序在需要访问 WEB 服务器的时候连接 WEB 服务器，并发出请求服务报文。

C/S 客户服务器结构中，服务器是被动响应方，工作流程一般如下：

（1）打开通信通道，某“永久”IP 地址的熟知端口上被动地接收客户连接请求；

（2）接收客户方请求，发送应答信号并处理该请求（激活一个新的线程处理）；

（3）返回到（1）等待新的客户请求到来。

C/S 结构中，客户方是服务主动请求方，工作流程一般如下：

（1）打开通信通道，连接服务器的监听端口；

（2）向服务器发出请求报文，等待并接收应答，继续提出请求报文，按照应用层协议规则和服务器方会话；

（3）请求结束后，关闭通信通道并终止连接。

C/S 结构中，客户端系统之间不能直接进行通信，只能是客户—服务器进行通信并完成相关网络应用功能。因此要求服务器使用永久 IP 地址和熟知端口号，以便客户端

能正常连接到服务器，并请求服务；客户端则不必有这个要求，可以在请求连接的同时提供给服务器端自己的 IP 地址和端口号。

P2P 对等结构中，网络通信的两个端系统地位平等，对等的意思就是没有一个总是打开的服务器，该结构中的任意两个端系统之间均可以进行网络连接并完成相关网络应用功能。在连接建立后，实际的数据传输过程中，仍然是资源的请求和服务，也即是每次数据传输还是可以称为客户和服务器，当 P2P 对等方提供资源时，充当了服务器端；而当 P2P 对等方请求资源时，则充当了客户端。

P2P 结构的应用程序要解决：

（1）如何定位 P2P 结构中的活动主机，P2P 结构中的主机频繁地加入和离开，如何高效定位和维护加入的主机网络地址和资源信息是关键问题；

（2）资源服务质量问题也是 P2P 结构的网络应用程序要考虑的，选择合适资源提供方进行网络连接并传输。

不同的体系结构，影响到网络应用程序设计的模式和设计流程，因此，在开发应用程序之前一定要弄清楚应用程序基于的应用层协议的体系结构。

6.2 套接字——网络编程接口

套接字是网络应用软件和网络协议软件的接口，是对网络协议栈通信服务功能的封装和抽象。从网络应用程序的实现角度，套接字是网络应用程序通过网络协议栈进行双向通信的通信端点，定义了网络应用程序与网络协议软件进行通讯时使用的一组操作（以及所需参数），是应用程序和协议栈软件之间的通道。通过套接字进行远程数据交换的进程，首先要能标识网络主机的 IP 地址；到达主机后再通过端口号找到通信的进程；还有同一个端口号在不同的传输层协议下的含义不同。因此，一个套接字应包含三个信息：IP 地址、端口号和传输层协议。

IP 地址和端口号组合被称作套接字地址，分为本地套接字地址和远程套接字地址。本地套接字地址即本地 IP 地址和本地端口组合，远程套接字地址即远程 IP 地址和远程端口的组合。两台网络主机只要连接自己的套接字，就能通过网络协议栈进行远程数据交换，不用操心网络协议分几层，不用管理应用数据具体传输形式等过程，如同图 6-6 所示。

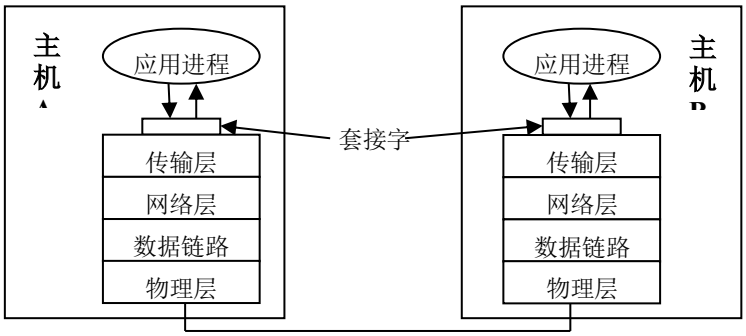


图 6-6　网络应用进程、套接字和协议栈的关系

套接字编程起源于加州大学伯克利分校开发的 UNIX 操作系统，是该操作系统的一部分，该套接字基于 TCP/IP 协议族，为该协议族的上层应用程序进行网络通信提供了一套编程接口。随着网络的普及，其他操作系统大都以 UNIX 的伯克利套接字为规范，定义了自己的套接字编程接口，如 Windows Socket 和 Linux Socket。操作系统提供了对套接字进行操作的一组编程接口（API），应用程序使用这些 API 函数，可以构造套接字，安装绑定套接字，连接套接字，通过套接字交换数据，关闭套接字，在此基础上实现各种网络应用程序。

为保证正常通信，网络通信的双方应该使用相同的通信协议，如都使用 TCP/IP 协议族。根据下层的传输协议提供的服务，套接字分为下述三种类型，不同套接字的工作方式并不一样，必须类型相同的套接字才能相互通信。

流式套接字（Stream Socket）：提供连接的、可靠的数据流传输服务。在 TCP/IP 协议族中对应的是传输层中的 TCP 协议，具有 TCP 协议为上层提供的可靠传输的服务特点，需要连接的建立、传输、撤销过程，保证数据有序、无重复地到达目的地。为了区分在一个并发处理的服务器中存在多条本地套接字地址一样的 TCP 连接（如 TCP 服务器），一个已经连接的 TCP 套接字对应于一个四元组：[本地 IP 地址，本地端口，远程 IP 地址，远程端口]。

数据报套接字（Datagram Socket）：提供无连接、不可靠的数据报传输服务。在 TCP/IP 协议族中对应的是传输层中的 UDP 协议，具有 UDP 协议为上层提供的不可靠服务的特点，同时具有连接时间短，同时向多个目标地址发送广播数据报。UDP 服务器对于并发的多个客户请求一个接着一个，不存在多个 UDP 连接使用同一个本地套接字连接多个远程客户进程，因此，UDP 套接字仅需要二元组：[本地 IP 地址，本地端口]。

原始套接字（Raw Socket）：允许应用程序直接访问较低层次的协议，如 IP 协议等，

一般用来实现新的协议。使用原始套接字需要应用程序员定义上层协议的首部格式，所以定义并实现新协议时，新协议的通信规则对应的首部字段格式用原始套接字封装，通信双方按照新协议的规则理解首部格式并执行相应操作。

第7章　Pcap 网络编程技术

Pcap 技术是网络底层开发的重要工具，允许应用程序避开成熟的网络协议，直接处理网络数据包。Pcap 技术直接对进出网卡的原始数据包进行处理，即用户自己对要传输的网络数据按照协议的首部格式进行封装，用户自己完成协议需要封装的内容，操作的是原始数据包套接字。套接字是操作系统提供给应用程序的编程接口，经过了操作系统处理（网络协议处理），提供的数据是剥离了网络协议的网络数据。

Winpcap 是 Windows 平台下，在数据链路层进行网络数据捕获和网络分析的开源库。它避开了操作系统对网络数据的隐藏，直接对数据链路层的数据分组进行处理，也就是原始的网络数据进行操作，可以方便的进行网络数据的封装和处理。对已经学过计算机网络网络协议原理的编程初学人员来讲，使用 Winpcap 开发网络应用程序，既能学习网络编程的基本过程，又能在编程过程中加深对网络协议栈的深入理解，是深入学习计算机网络协议的很好的途径。

Winpcap 提供的基本功能有：

- 捕获经由主机的数据包；
- 根据应用程序提供的规则（程序员自己定义）过滤数据包；
- 发送原始数据包到网络上；
- 统计和收集网络流量信息。

7.1 Winpcap 概述

Winpcap 包括三个组成部分：内核级的包过滤驱动程序 NPF(Netgroup Packet Filter)、低级动态链接库 packet.dll 和用户级的 wpcap.dll。这三个组成部分的相互关系如图 7-1 所示：

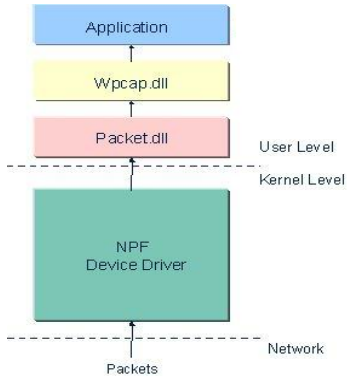


图 7-1　Winpcap 三个组成部分的关系

第一个模块是内核网络数据包过滤器（Netgroup Filter，NPF），它是 Winpcap 的核心部分，负责直接处理网络上传输的数据包，将进出网络适配器的数据包拷贝，并且对用户提供捕获、发送和分析等功能。

第二个模块是低级动态链接库 packet.dll，在 Win32 平台上提供了与 NPF 的一个通用接口，包含了一个底层 API 库，这些 API 函数可以直接用来访问操作系统内核。

第三个模块是用户级动态链接库 wpcap.dll，提供了基于用户级的 API 函数，提供一组更加友好、功能更强大的高层函数库，应用程序员可以方便的调用。

用户编程实现网络捕获功能时，可以直接使用第一个模块的函数，也可以直接使用第二个模块的函数，当然也可以使用第三个模块的函数。应用程序和上述三个模块之间的关系如下图所示。

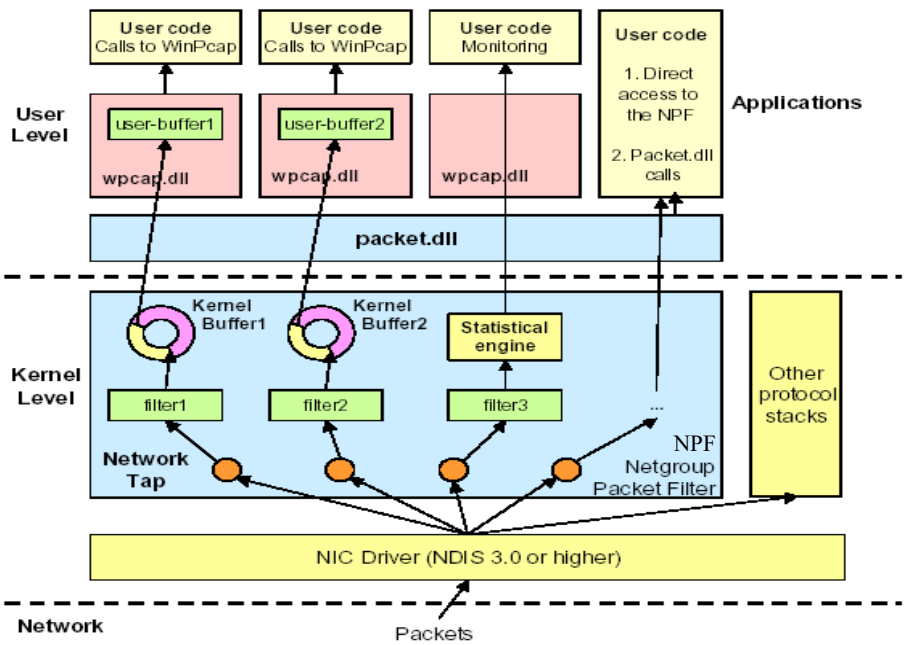


图 7-2　Winpcap 开发模块与应用程序的关系

7.1.1 Winpcap 资料下载

使用 Winpcap 开源库必须下载并安装 Winpcap Driver、DLL 和 wpdpack（developer's pack）。Winpcap 的官方网站上有这两个软件的最新版本。其下载地址是：http://www.winpcap.org/archive/

步骤 1：下载并安装，安装后重启机器。 Winpcap 驱动的安装包(如：

Winpcap_4_1.exe) ；

步骤 2：下载程序员开发包(如：WpdPack_4_1.zip)，解压后会看到其中包含了 docs、Include、lib、Examples 等文件夹。

Winpcap 中文使用手册地址：http://www.ferrisxu.com/WinPcap/html/index.html

Winpcap 功能强大，效率高，使用方便，适应很多平台。通常高校的教学语言都是基于 VC++6.0 版本，学生实践开发时，比较常使用 Visul Studio C/C++。这两种开发环境下的 Winpcap 的环境配置如下所述。

7.1.2 Winpcap 环境配置

- **VC++6.0 环境配置**

VC++6.0 是高校初学 C 类语言的教学环境， Winpcap 的配置步骤如下：

步骤 1：在 VC++中设定 Include 目录。打开 VC++菜单，Tools->Option ->Directories，在 include files 中添加……\wpdpack\Include 目录（安装 wpdpack 中得到的）；

步骤 2：在 VC++中设定 Library 目录，在 Library files 中添加……\wpdpack\Lib 目录；

步骤 3：在 VC++的新建工程中，在 Project->settings->Link，在 Object/library modules 中加上 wpcap.lib。

- **VS2017 环境配置**

Visul Studio 是目前项目开发人员习惯使用的开发环境，在该环境中，配置 Winpcap 非常方便。以 VS2017 为例，配置 Winpcap 的要点如下：

步骤 1：新建一个空项目，同时建立一个空的 C++源文件。

步骤 2：右键项目选择属性，选择配置属性 -> C/C++ -> 预处理器，如图所示在预处理器定义处添加 WPCAP 和 HAVE_REMOTE 这两个宏定义：

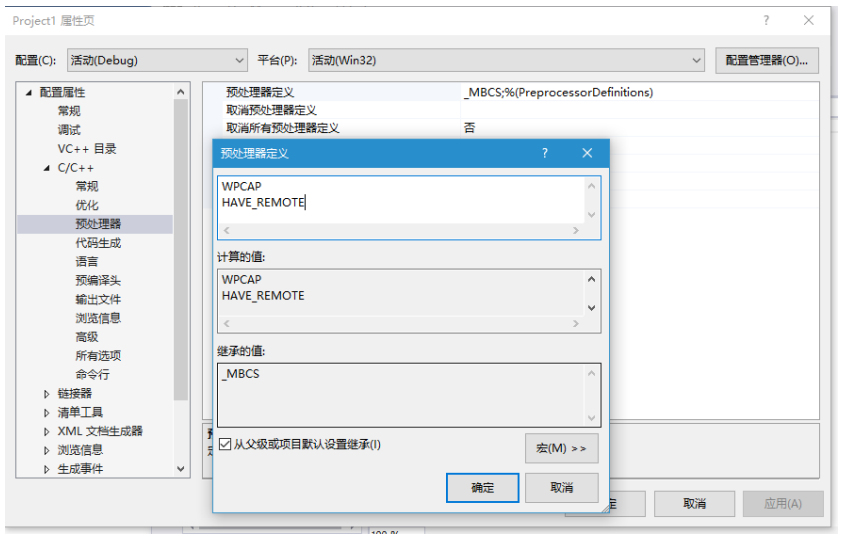


图 7-3　Winpcap 的 VS2017 配置窗口

步骤 3：在属性配置界面，配置属性—>链接器—>输入，在附加依赖项添加 wpcap.lib 和 ws2_32.lib 两个库：

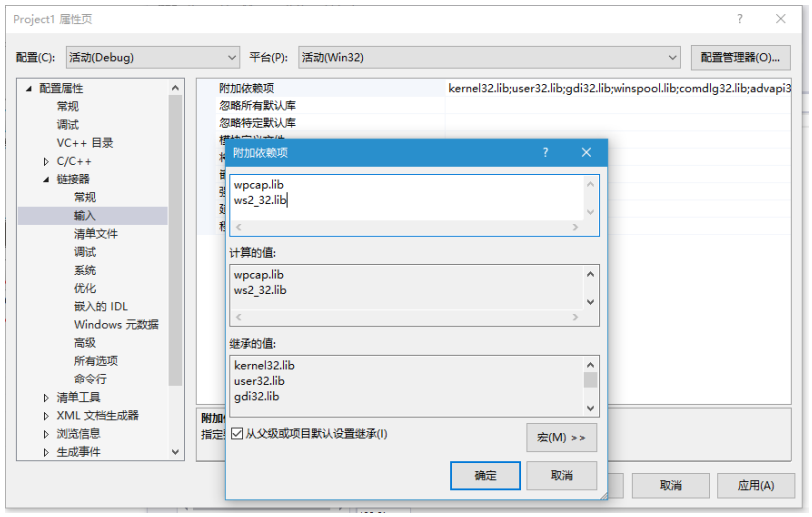


图 7-4　VS2017 属性配置界面

步骤 4：在属性配置界面，配置属性—>VC++目录，添加下载好的 WpdPack 文件夹中的包含路径（Include 目录）和库路径（Lib 目录）：

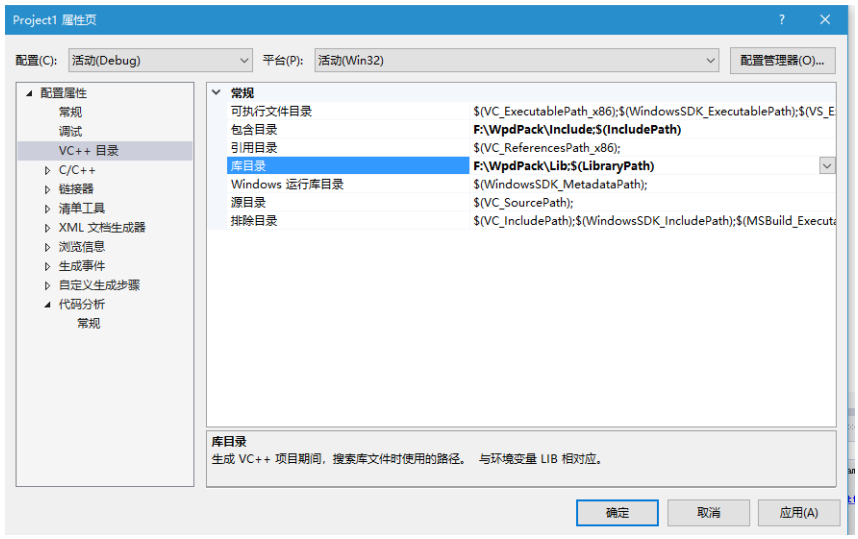


图 7-5 VS2017 目录选项卡

步骤 5： 在建立的 cpp 源程序文件中，代码行开始的地方，添加 pcap 头文件和 #pragma comment(lib, "wpcap.lib")即可。

```
#include "pcap.h"
#pragma comment ( lib, "wpcap.lib")
```

7.2 Winpcap 编程

7.2.1 Winpcap 的开发流程

Winpcap 开发网络应用程序的流程大致一样：首先是获取主机安装的网络设备列表——选择并激活某个网络设备——封装数据包/过滤数据包——发送数据包/解析数据包。以捕获并解析数据包为例，其开发流程大致如下：

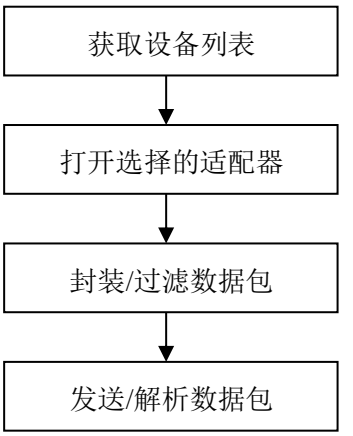


图 7-6　Winpcap 的开发流程

7.2.2 Winpcap 基本网络编程函数

■ **获取设备列表 pcap_findalldevs( )**

Winpcap 提供 pcap_findalldevs( ) 函数完成查找主机网络适配器功能，该函数返回本地主机上安装的所有适配器的一个列表。函数返回一个相连的 pcap_if_t 结构的列表，该列表的每一项包含关于适配器的详细的信息。

pcap_findalldevs 函数原型如下：

int　pcap_findalldevs(pcap_if_t　**alldev,　char *errbuf)

参数：alldev：指向列表的第一个元素，列表元素每个都是 pcap_if_t 类型，如果没有已连接并打开的适配器，则为 NULL；

errbuf：存储错误信息。

函数返回值：成功返回 0；失败返回-1。

例程：

```
pcap_if_t   *alldevs，*d;
int   i=0;
char errbuf[PCAP_ERRBUF_SIZE];
  if (pcap_findalldevs(&alldevs, errbuf) == -1)
  {   fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
        exit(1);
  }
  for(d=alldevs;d;d=d->next)
{   printf("%d. %s", ++i, d->name);
     if (d->description)
          printf(" (%s)\n", d->description);
        else
          printf(" (No description available)\n");
  }
  if (i==0)
  {   printf("\nNo interfaces found! \n");
     return ;
   }
  pcap_freealldevs(alldevs);
```

pcap_findalldevs（）带回的网络适配器的详细信息存储在一个结构体 pcap_if_t 中，pcap_if_t 结构的定义如下：

```
struct        pcap_if_t
    { struct   pcap_if_t   *next;              //如果不为空，则指向下一个元素
                char     *name;                //设备名称
                char     *description;         //描述设备
    struct   pcap_addr   *addresses;           //接口地址列表
              bpf_u_int32   flags;             //标志位，标志是否 loopback 设备
};
```

每个网络适配器的 pcap_if_t 结构都包含了一个 pcap_addr 结构的列表：

- 该接口的地址列表
- 网络掩码的列表（每个网络掩码对应地址列表中的一项）
- 广播地址的列表（每个广播地址对应地址列表中的一项）
- 目标地址的列表（每个目标地址对应地址列表中的一项）

pcap_addr 结构定义如下：

```
struct   pcap_addr   pcap_addr_t;
struct     pcap_addr
{    struct   pcap_addr   *next;            //如果不为空，则指向下一个元素
     struct   sockaddr   *addr;             //接口 IP 地址
     struct   sockaddr   *netmask;          //接口网络掩码
     struct   sockaddr   *broadaddr;        //接口广播地址
     struct   sockaddr   *dstaddr;          //接口 P2P 目的地址
};
```

■ **打开选择的适配器 pcap_open_live（）**

通过 pcap_findalldevs( )找到本地主机上安装的所有网络适配器，选择其中的一个进行网络编程必须打开它，使用的函数是 pcap_open_live（），其功能是打开本地主机上的网络适配卡，函数原型描述如下：

```
pcap_t * pcap_open_live( const char * device, int snaplen, int   promisc, int   to_ms,
                         char   * ebuf )

   参数：device：所选择的网络适配器设备标识（字符串）；
         snaplen：进行捕获的数据包长度限制，字节为单位；
         promisc：是否以混杂模式进行捕获；
         to_ms：  捕获数据包能容忍的超时时间，毫秒为单位；
         ebuf：出错信息缓存地址；

   返回值：pcap_t 类型的指针
```

函数 pcap_open_live（）的返回值是 pcap_t 类型的指针，pcap_t 结构体对用户透明，提供了对一个已打开的适配器实例的描述。windows 平台上 pcap_t 的主要成员有（只列举几个重要成员）：

```
typedef  struct  pcap  pcap_t;
struct   pcap
 {  ADAPTER   *  adapter;
    LPPACKET      Packet;
              int       linktype;              //数据链路层类型
              int       linktype_ext;          // linktype 成员扩展信息
              int       offset;                //时区偏移
              int       activated;             //捕获准备好否
    …  };
```

例如：

```
pcap_t *adhandle =   pcap_open_live( d->name,                  //指定适配器名字
                                     65535,     //捕获包最大字节限制 65535
                                        1,          //混杂模式
                                     1000,          //读取超时最大 1000 毫秒
                                     errbuf         //错误信息保存在 errbuf
                                     );
```

■ **关闭适配器 pcap_close（）**

打开某个网络适配器，使用结束后必须关闭它。关闭适配器的函数是 pcap_close（），其函数原型描述如下：

```
void   pcap_close ( pcap_t   *p);
    参数：p：pcap_t 类型的指针，指向要关闭的适配器；
    返回值：无
```

该函数的参数是 pcap_t 类型的指针，函数执行的结果释放掉 p 指针指向的网络适配器。

■ **捕获数据包**

捕获数据包通常有两种方式：一种是直接捕获；另一种是回调模式。

– **直接捕获数据包方式**

使用 int pcap_next_ex（）直接捕获数据包，其函数原型描述如下：

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

参数： p：指定的适配器名称；

pkt_header：指向捕获数据包的首部；

pkt_data：指向捕获的数据包具体数据

返回值：整数值，具体取值的含义如下：

- 1：数据包读取成功；
- 0：如果超时时间到，则 pkt_header 和 pkt_data 都不指向有用的数据包；
- -1：出现错误；
- -2：离线捕获（文件操作）遇到文件尾部的 EOF。

该函数从适配器或脱机文件读取一个数据包。用于接收下一个可用的数据包。pcap_next_ex() 目前只在 Win32 下可用，因为它不是属 libpcap 原始的 API。这意味着含有这个函数的代码将不能被移植到 UNIX 上。

该函数的第 2 个参数是指向捕获数据包的首部指针，注意这个首部不是网络协议的首部字段，而是对捕获数据包的时间、长度等信息的统计信息。它是一个结构体指针，对应的结构体 pcap_pkthdr 定义如下：

```
struct pcap_pkthdr {    struct timeval   ts;              //时间戳
                        bpf_u_int32   caplen;             //当前分组长度
                        bpf_u_int32   len;                //数据包的长度
                    };
```

例程：

```
while ((res = pcap_next_ex(adhandle, &header, &pkt_data)) >= 0)
 {    if (res == 0)
       { continue;                                        //超时，时间到
      }
   ltime = localtime(&header->ts.tv_sec);                 //时间戳格式转换
   strftime(timestr, sizeof(timestr), "%H:%M:%S", ltime);
   printf("%s, %.6d len:%d\n", timestr, header->ts.tv_usec, header->len);
```

```
}
if (res == -1)
{ printf("Error reading the packets: %s\n", pcap_geterr(adhandle));
 return -1;
}
```

– **回调方式捕获数据包**

使用 int pcap_loop（）直接捕获数据包，函数原型描述如下：

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

参数：p：指定适配器的名称；
cnt：指定捕获的数据包数目；
callback：回调函数；
user：留给用户。

返回值：整数值。

其中，回调函数 pcap_handler 函数必须是全局函数或静态函数，参数默认，其原型定义如下：

```
void (* pcap_handler) ( u_char * user, const struct pcap_pkthdr* pkt_header,
                        const u_char * pkt_data );
```

参数： user：pcap_loop 函数传导的 user；
pkt_header：捕获数据包的首部指针；
pkt_data ：捕获数据包的内容。

返回值：无

例程：

```
void pcap_handler (u_char* user, const struct pcap_pkthdr* pkt_header,
                   const u_char* pkt_data);
const struct pcap_pkthdr *header;
const u_char *pkt_data;
```

```
int main( )
{       pcap_t    *adhandle;
        pcap_loop( adhanlde, 0, packet_handler, NULL);
        return    0;
}
void     pcap_handler  (u_char*  user,     const  struct  pcap_pkthdr*  pkt_header,
                                const u_char* pkt_data);
{
                                //捕获的数据包通过该函数回传给应用程序
}
```

■ **过滤数据包**

Winpcap 提供的常用的过滤数据包的函数：一个是 pcap_compile( )，另一个是 pcap_setfilter( )。

使用 pcap_compile( ) 使用捕获的数据包的过滤。该函数编译一个数据包过滤器，将一个高级的、布尔形式表示的字符串转换成低级的、二进制过滤语句，能够被过滤虚拟机执行。pcap_setfilter() 在核心驱动中将过滤器和捕获过程结合在一起。从这一时刻起，所有网络的数据包都要经过过滤，通过过滤的数据包将被传入应用程序。这两个函数的函数原型描述如下：

```
int    pcap_compile ( pcap_t *    p, struct    bpf_program *    fp, char *    str,      int
                        optimize, bpf_u_int32    netmask )

    参数：        p：指定适配器的名称；
                  fp：pcap_compile（）返回结果；
                  str：过滤规则表达式；
              optimize：控制是否对最终生成的字节码优化；
              netmask：需捕获数据包的 IPV4 掩码。

    返回值：整数值，执行成功返回 0，否则返回-1。
```

```
int    pcap_setfilter ( pcap_t *    p, struct    bpf_program *    fp );

    参数：p：指定适配器的名称；
          fp：pcap_compile（）返回结果。

  返回值：整数值，执行成功返回 0，否则返回-1。
```

其中结构体 bpf_program 定义如下：

```
struct    bpf_program
{                  u_int    bf_len;            //BPF 中谓词判断指令的数目
struct    bpf_insn     *bf_insns;          //指向第一个谓词判断指令
 };
```

函数 pcap_compile( )执行成功返回 0，否则返回-1。可以调用 pcap_geterr 函数显示所发生的错误信息。

pcap_setfilter( )被调用时，该过滤器被应用到来自网络的所有数据包上，所有符合过滤要求的数据包将会被存储到内核缓冲区中。pcap_setfilter( )函数执行成功返回 0，失败返回-1，调用 pcap_geterr 函数显示所发生的错误信息。

例程：

```
char packet_filter[] = "ip and udp";
struct bpf_program fcode;
            // 获取接口地址的掩码，如果没有掩码，认为该接口属于一个 C 类网络
if (d->addresses != NULL)
                    netmask=((struct sockaddr_in *)(d->addresses->netmask))->
                                    sin_addr.S_un.S_addr;
 else    netmask=0xffffff;
if (pcap_compile(adhandle, &fcode, packet_filter, 1, netmask) <0 )
{    fprintf(stderr,"\nUnable to compile the filter. Check the syntax.\n");
        pcap_freealldevs(alldevs);
  return -1;
}
if    (pcap_setfilter(adhandle, &fcode)<0)
 {     fprintf(stderr,"\nError setting the filter.\n");
       pcap_freealldevs(alldevs);
       return -1;
 }
```

pcap_compile（）中的过滤表达式由一个或多个原语组成。原语通常由一个 id（名称或者数字）和在它前面的一个或几个修饰符组成。有 3 种不同的修饰符：

- **类型**　指明 id 名称或者数字指的是哪种类型

可能是 host，net 和 port。例如 “host foo”、“net 128.3”、“port 20”。如果没有类型修饰符，缺省为 host。

- **方向** 指明向和/或 从 id 传输等方向的修饰符

可能的方向有 src、dst、src or dst 和 src and dst。例如“src foo”、“dst net 128.3”、“src or dst port ftp-data”。如果缺省为 src or dst。

- **协议** 指明符合特定协议的修饰符

目前的协议包括 ether、fddi、ip、ip6、arp、rarp、tcp 和 udp 等。 例如“ether src foo”、“arp net 128.3”。

如果没有协议修饰符，则表示声明类型的所有协议。

例如“src foo”表示“（ip or arp or rarp） src foo”

“port 53”表示“（tcp or udp） port 53”。

编译并设置过滤器过滤函数的实例如下：

例程：

```
if (d->addresses != NULL)                    //获取接口第一个地址的掩码
    netmask=((struct sockaddr_in *)(d->addresses->netmask))->sin_addr. S_un.S_addr;
else
                netmask=0xffffff;  //如果该接口没有地址，则假设在 C 类网络中
    }
                                                                    //编译过滤器规则
if (pcap_compile(adhandle, &fcode, "ip and tcp", 1, netmask) < 0)
{   fprintf(stderr,"nUnable to compile the packet filter. Check the syntax.n");
pcap_freealldevs(alldevs);                                  //释放设备列表
return -1;
}
if (pcap_setfilter(adhandle, &fcode) < 0)                       //设置过滤器
{     fprintf(stderr,"nError setting the filter.n");
                    pcap_freealldevs(alldevs);              //释放设备列表
return -1;
}
```

■ **发送数据包**

Winpcap 提供两种方式发送数据包，一种是单个数据包单独发送，一次发送一个数

据包；另一种是连续发送，将待发送的数据包放在发送队列中，按照队列顺序挨个发送。

– **发送单个数据包 pcap_sendpacket（）**

打开适配器后，调用 pcap_sendpacket( )函数来发送一个手写的数据包。pcap_sendpacket()用一个包含要发送的数据的缓冲区、该缓冲区的长度和发送它的适配器作为参数。注意该缓冲区是不经任何处理向外发出的，这意味着，如果想要发送的数据正确的被通信一方理解的话，应用程序必须产生正确的（现在通用的网络协议或双方自定义网络协议）协议首部。

```
int   pcap_sendpacket ( pcap_t *   p,              u_char *   str,       int
                        optimize) ;

    参数：  p：已经打开的适配器；
              str：要发送数据包的内容；
          optimize：发送数据包的长度。

    返回值：整数值。
```

例程：

```
u_char   packet[100];
if((fp = pcap_open_live(argv[1], 100, 1, 1000, error) ) == NULL)
{    fprintf(stderr,"\nError opening adapter: %s\n", error);
     return;
}
 packet[0…5]=1;              //设置 MAC 帧首部的目的 MAC 地址为 1:1:1:1:1:1
 packet[6…11]=2;             //设置 MAC 帧首部的源 MAC 地址为 2:2:2:2:2:2
 for(i=12;i<100;i++)                                  //填充该 MAC 帧的其余部分
     {   packet[i]=i%256;      }
 pcap_sendpacket ( fp,   packet, 100);                          //发送该 MAC 帧
```

一个完整的打开适配器，发送数据包的例子如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
```

```
void main(int argc, char **argv)
{    pcap_t *fp;
char errbuf[PCAP_ERRBUF_SIZE];
u_char packet[100];
int i;
if (argc != 2)                                          // 检查命令行参数的合法性
{    printf("usage: %s interface (e.g. 'rpcap://eth0')", argv[0]);
     return;
}
                                                        // 打开输出设备
if ( (fp= pcap_open(argv[1],                            // 设备名
                    100,                                //只捕获前 100 个字节
 PCAP_OPENFLAG_PROMISCUOUS,                             // 混杂模式
                    1000,                               // 读超时时间设为 1000 毫秒
                    NULL,                               // 远程机器验证
                    errbuf                              // 错误缓冲区
                    ) ) == NULL)
{
fprintf(stderr,"nUnable to open the adapter. %s is not supported by WinPcapn",  argv[1]);
 return;
}
                                  //设置 MAC 帧的目的 MAC 地址为 1:1:1:1:1:1
packet[0]=1;
packet[1]=1;
packet[2]=1;
packet[3]=1;
packet[4]=1;
packet[5]=1;
                                      //设置 MAC 帧的源 MAC 地址为 2:2:2:2:2:2
packet[6]=2;
packet[7]=2;
packet[8]=2;
```

```
packet[9]=2;
packet[10]=2;
packet[11]=2;
for(i=12;i<100;i++)                                    // 填充该 MAC 帧余下内容
{ packet[i]=i%256;
}
if (pcap_sendpacket(fp, packet, 100 /* size */) != 0)          //发送数据包
{ fprintf(stderr,"nError sending the packet: n", pcap_geterr(fp));
 return;
}
return;
}
```

■ **发送队列连续发送数据包**

pcap_sendpacket( ) 提供了一种简单而直接的方法来发送单个数据包，而 send_queue( ) 则提供了一种更高级，结构更优的方法来发送一组数据包。发送队列是一个容器，它能容纳不同数量的数据包，这些数据包将被发送到网络上。队列的大小表示存储的数据包最大数量。 打开适配器后，调用 send_queue_trasmit( )函数来发送一个队列中的数据包。

使用发送队列来发送数据分组，一般分为以下 4 个步骤：

[1] 队列创建

[2] 队列添加数据包

[3] 发送队列中的数据包

[4] 销毁发送队列，释放内存空间

发送队列中的结构体 pcap_send_queue( )的定义如下：

```
struct   pcap_send_queue (   u_int   maxlen;          //队列最大长度（字节数）
                             u_int   len;           //队列当前字节数
                             char*   buffer;        //存储待发数据包缓冲区
                           );
```

对应发送队列发送数据的步骤，有 Winpcap 使用队列创建函数、数据包添加函数、队列发送函数和队列释放函数来完成连续发送队列中的数据包。

– **发送队列创建函数 pcap_sendqueue_alloc( )**

pcap_send_queue * pcap_sendqueue_alloc (u_int memsize)；

参数：memsize：发送队列的大小，字节为单位。

返回值：执行成功返回所分配队列的内存指针，否则返回 NULL。

队列发送函数 pcap_sendqueue_alloc( )在设置 memsize 参数时，应注意该参数指的发送队列大小包括两个部分：要发送数据包的数据（包含网络协议首部），和描述数据包发送信息的结构体 pcap_pkthdr 内容。

例：

squeue = pcap_sendqueue_alloc(unsigned int)((packetlen + sizeof(struct pcap_pkthdr)) * npacks));

上述例子中，packetlen 为数据包的长度，sizeof (struct pcap_pkthdr) 是每个数据包信息结构体长度，npacks 是数据包个数。

发送队列一旦创建，添加数据包函数 pcap_sendqueue_queue( )就可以将数据包添加到发送队列中。

- **添加数据包函数 pcap_sendqueue_queue（）**

int pcap_sendqueue_queue(pcap_send_queue* queue, const struct pcap_pkthdr* pkt_header, const u_char * pkt_data)；

参数：queue：指向 pcap_sendqueue_alloc 函数分配的发送队列；

pkt_header：待发送数据包所附加的数据首部信息；

pkt_data：待发送数据包的数据信息。

返回值：执行成功返回 0，否则返回-1。

发送队列创建和添加数据包完毕后，就可使用 pcap_sendqueue_transmit( )将队列发送出去。

- **发送队列函数 pcap_sendqueue_transmit( )**

u_ int pcap_sendqueue_transmit(pcap_t *p, pcap_send_queue *queue, int sync)；

参数：p：指向发送数据包的适配器；

queue：指向待发送的数据包的队列；

sync：为 0 表示尽快发送，不为 0 则根据时间戳发送。

返回值：执行成功返回实际发送的字节数，否则表示发送过程中出现错误。

发送队列中已经没有排队发送的数据包，并且当不需要再使用该发送队列发送数据包时，应该摧毁所创建的发送队列，释放掉发送队列有关的所有内存资源。

void pcap_sendqueue_destroy(pcap_send_queue *queue)； 参数：queue 指向待发送的数据包队列； 返回值：执行成功释放与发送队列有关的所有资源。

按照发送队列创建、添加数据包、发送和撤销流程，使用发送队列从文件中将数据包填充到发送队列，并发送数据包的完整例子如下所示：

例程：

```
squeue = pcap_sendqueue_alloc(caplen);                    //分配发送队列空间
while ((res = pcap_next_ex( indesc, &pktheader, &pktdata)) == 1)
{ if (pcap_sendqueue_queue(squeue, pktheader, pktdata) == -1)
 { printf("Warning: packet buffer too small, not all the packets will be sent.n");
       break;
 }
   npacks++;
}
if (res == -1)
{   printf("Corrupted input file.n");
 pcap_sendqueue_destroy(squeue);
 return;
}
                                                          // 发送队列
cpu_time = (float)clock ();
if ((res = pcap_sendqueue_transmit(outdesc, squeue, sync)) < squeue->len)
{ printf("An error occurred sending the packets: %s. Only %d bytes were sentn",
 pcap_geterr(outdesc), res);
}
cpu_time = (clock() - cpu_time)/CLK_TCK;
printf ("nnElapsed time: %5.3fn", cpu_time);
```

```
printf ("nTotal packets generated = %d", npacks);
printf ("nAverage packets per second = %d", (int)((double)npacks/cpu_time));
printf ("n");
pcap_sendqueue_destroy(squeue);                                    //释放发送队列
```

7.3 Winpcap 分析实例

使用 Winpcap 进行网络低层开发的基本步骤和常用函数前面已经做了详细的讲解，本章通过两个网络捕获的实例来练习怎样具体开发网络程序。

7.3.1 捕获解析 UDP 数据包实例

第一个程序的主要目标是解析所捕获的 UDP 数据包的协议首部。UDP 协议首部非常简单，只有 8 个字节，下层网络层使用 IP 协议，作为入门实例，很容易学习。

实例代码如下：

```
---------------------------------------------------------------------------
#include "pcap.h"
typedef struct ip_address{                                         // 4 字节的 IP 地址
        u_char      byte1;
        u_char      byte2;
        u_char      byte3;
        u_char      byte4;
}ip_address;
                                                                   // IPv4 首部
typedef struct ip_header{
        u_char ver_ihl;          // 版本 (4 bits) + 首部长度 (4 bits)
        u_char tos;                                                // 服务类型(8 bits)
        u_short tlen;                                              // 总长(16 bits)
        u_short identification;                                    // 标识(16 bits)
        u_short flags_fo;        // 标志位(3 bits) + 段偏移量    (13 bits)
        u_char ttl;                                // 存活时间(8 bits)
        u_char proto;                              // 协议(8 bits)
        u_short crc;                               // 首部校验和(16 bits)
        ip_address saddr;                          // 源地址(32 bits)
        ip_address daddr;                          // 目的地址(32 bits)
```

```
        u_int op_pad;                    // 选项与填充(32 bits)
    };
                                         // UDP 首部
typedef struct udp_header{
        u_short sport;                   //源端口(16 bits)
        u_short dport;                   //目的端口(16 bits)
        u_short len;                     // UDP 数据包长度(16 bits)
        u_short crc;                     //校验和(16 bits)
        };

void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data);                                          // 回调函数原型
                                                     // 主函数
main()
{ pcap_if_t *alldevs;
pcap_if_t *d;
int inum;
int i=0;
pcap_t *adhandle;
char errbuf[PCAP_ERRBUF_SIZE];
u_int netmask;
char packet_filter[] = "ip and udp";
struct bpf_program fcode;
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) == -1)
                                                     // 获得设备列表
{ fprintf(stderr,"Error in pcap_findalldevs: %sn", errbuf);
exit(1);
}
for(d=alldevs; d; d=d->next)
{                                                    // 打印网络适配器列表
printf("%d. %s", ++i, d->name);
if (d->description)
```

```
printf(" (%s)n", d->description);
else
printf(" (No description available)n");
}
if(i==0)
{ printf("nNo interfaces found! Make sure WinPcap is installed.n");
return -1;
}
printf("Enter the interface number (1-%d):",i);
scanf("%d", &inum);
if(inum < 1 || inum > i)
{ printf("nInterface number out of range.n");
      pcap_freealldevs(alldevs);                                    //释放设备列表
return -1;
}
                                                                    //跳转到已选网络适配卡
for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);
if ( (adhandle= pcap_open(d->name,          65536,
PCAP_OPENFLAG_PROMISCUOUS,      1000,                NULL,
                errbuf)                                             //打开网络适配卡
) == NULL)
{
fprintf(stderr,"nUnable to open the adapter. %s is not supported by WinPcapn");
pcap_freealldevs(alldevs);                                          // 释放设备列表
return -1;
}
                                                                    // 检查数据链路层，只考虑以太网
if(pcap_datalink(adhandle) != DLT_EN10MB)
{ fprintf(stderr,"nThis program works only on Ethernet networks.n");
  pcap_freealldevs(alldevs);                                        // 释放设备列表
return -1;
}
```

```
if(d->addresses != NULL)
                                        // 获得接口第一个地址的掩码
netmask=((struct sockaddr_in* ) (d->addresses-> netmask ) )-> sin_addr. S_un.S_addr ;
else                                    //如果接口没有地址，则假设一个 C 类的掩码
netmask=0xffffff;
if (pcap_compile(adhandle, &fcode, packet_filter, 1, netmask) <0 ) //编译过滤器
{ fprintf(stderr,"nUnable to compile the packet filter. Check the syntax.n");
    pcap_freealldevs(alldevs);                                  // 释放设备列表
return -1;
}
if (pcap_setfilter(adhandle, &fcode)<0)                         //设置过滤器
{ fprintf(stderr,"nError setting the filter.n");
pcap_freealldevs(alldevs);                                      // 释放设备列表
return -1;
}
printf("nlistening on %s...n", d->description);
pcap_freealldevs(alldevs);                                      // 释放设备列表
pcap_loop(adhandle, 0, packet_handler, NULL);                   // 开始捕捉
return 0;
}
                                //回调函数，当收到每一个数据包时会被 libpcap 所调用
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data)
{ struct tm *ltime;
char timestr[16];
ip_header *ih;
udp_header *uh;
u_int ip_len;
u_short sport,dport;
time_t local_tv_sec;
local_tv_sec = header->ts.tv_sec;                   //将时间戳转换成可识别的格式
    ltime=localtime(&local_tv_sec);
```

```
strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);
                                                    //打印数据包的时间戳和长度
printf("%s.%.6d len:%d ", timestr, header->ts.tv_usec, header->len);
                                                    //获得 IP 数据包头部的位置
ih = (ip_header *) (pkt_data + 14);
      ip_len = (ih->ver_ihl & 0xf) * 4;                    //加上以太网头部长度
uh = (udp_header *) ((u_char*)ih + ip_len);      // 获得 UDP 首部的位置
sport = ntohs( uh->sport );              //将网络字节序列转换成主机字节序列
dport = ntohs( uh->dport );              //打印 IP 地址和 UDP 端口
printf("%d.%d.%d.%d.%d -> %d.%d.%d.%d.%dn",
ih->saddr.byte1,
ih->saddr.byte2,
ih->saddr.byte3,
ih->saddr.byte4,
sport,
ih->daddr.byte1,
ih->daddr.byte2,
ih->daddr.byte3,
ih->daddr.byte4,
dport);
}
--------------------------------------------------------------------------------
```

首先，我们将过滤器设置成"ip and udp"。在这种方式下， packet_handler()只会收到基于 IPv4 的 UDP 数据包，这将简化解析过程，提高程序的效率。

其次，对于回调函数 packet_handler( )的处理, 尽管受限于单个协议的解析（比如基于 IPv4 的 UDP），但是它的使用体现了捕获解析网络数据包的复杂过程，类似 TcpDump 或 WinDump 对网络数据流进行解码。该例中，虽然我们跳过了对 MAC 首部的解析，但是为了确保捕获的是以太网的帧，在开始捕获诉举报前，使用了 pcap_datalink() 对 MAC 层进行了检测，以确定后面的捕获解析是在以太网协议上进行。

以太网协议是数据链路层网络协议，它的 MAC 首部是 14 个字节，尾部是 2 个字节。 IP 数据包的首部就位于 MAC 首部的后面，按照 IP 协议的首部格式，IP 数据包的 length 字段得到 IP 数据包的长度，上例中没有选项字段，IP 数据包长度为 20 个字

节；还可以从 IP 数据包的首部解析到通信双方的源 IP 地址和目的 IP 地址。上例中的传输层协议是 UDP 协议，UDP 报文段的首部紧跟在 IP 首部后面，上例中，20 个字节的 IP 数据包首部之后就是 UDP 首部的第一个字节，再根据 UDP 协议首部格式，很轻松就能解析到 UDP 的源端口和目的端口信息。

被解析出来的值被打印在屏幕上，形式如下所示：

1. DevicePacket_{A7FD048A-5D4B-478E-B3C1-34401AC3B72F} (Xircom t 10/100 Adapter)

Enter the interface number (1-2):1

listening on Xircom CardBus Ethernet 10/100 Adapter...

16:13:15.312784 len:87 130.192.31.67.2682 -> 130.192.3.21.53

16:13:15.314796 len:137 130.192.3.21.53 -> 130.192.31.67.2682

16:13:15.322101 len:78 130.192.31.67.2683 -> 130.192.3.21.53

7.3.2 打印通过适配器的数据包实例

第二个实例程序将每一个通过适配器的数据包打印出来。仅仅打印经过适配器的数据包的统计信息。

```
---------------------------------------------------------------------------
#include "pcap.h"
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data);                    //回调函数函数原型
main()
{ pcap_if_t *alldevs;
pcap_if_t *d;
int inum;
int i=0;
pcap_t *adhandle;
char errbuf[PCAP_ERRBUF_SIZE];                    //获取本机设备列表
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) == -1)
{ fprintf(stderr,"Error in pcap_findalldevs: %sn", errbuf);
```

```
    exit(1);
    }
    for(d=alldevs; d; d=d->next)
    {                                                    //打印网络设备
列表
    printf("%d. %s", ++i, d->name);
    if (d->description)
    printf(" (%s)n", d->description);
    else
    printf(" (No description available)n");
    }
    if(i==0)
    { printf("nNo interfaces found! Make sure WinPcap is installed.n");
    return -1;
    }
    printf("Enter the interface number (1-%d):",i);
    scanf("%d", &inum);
    if(inum < 1 || inum > i)
    { printf("nInterface number out of range.n");
    pcap_freealldevs(alldevs);                           //释放设备列表
    return -1;
    }
    for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);      //跳转到指定的适配器
    if ( (adhandle= pcap_open( d->name,   65536,         //打开网络适配卡
PCAP_OPENFLAG_PROMISCUOUS, 1000, NULL,    errbuf)) == NULL)
    { fprintf(stderr,"nUnable to open the adapter. %s is not supported by WinPcapn",
d->name);
    pcap_freealldevs(alldevs);                           //释放设备列表
```

```
return -1;
}
printf("nlistening on %s...n", d->description);
pcap_freealldevs(alldevs);                                    //释放设备列表
pcap_loop(adhandle, 0, packet_handler, NULL);                 // 开始捕获
return 0;
}
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data)                                                    //回调函数
{ struct tm *ltime;
char timestr[16];
time_t local_tv_sec;                                          //将时间戳转换成可识别的格式
local_tv_sec = header->ts.tv_sec;
ltime=localtime(&local_tv_sec);
strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);
printf("%s,%.6d len:%dn", timestr, header->ts.tv_usec, header->len);
}
```

上面的程序将每一个数据包的时间戳和长度从 pcap_pkthdr 的首部解析出来，并打印在屏幕上。

第8章　Socket 网络编程技术

8.1 WinSock 概述

Windows Socket 是从 UNIX Socket 继承发展而来，最新的版本是 2.2。Windows 系统提供的套接字函数通常封装在 Ws2_32.dll 动态链接库中，其头文件 Winsock2.h 提供了套接字函数的原型，库文件 Ws2_32.lib 提供了 Ws2_32.dll 动态链接库。在使用套接字函数前，用户需要引用 Winsock2.h 头文件，并链接 Ws2_32.lib 库文件。在代码前加入下面两行代码：

```
#include "winsock2.h"                                    //引用头文件
#pragma comment (lib,"ws2_32.lib")                       //链接库文件
```

8.1.1 WinSock 的初始化和终止

WinSock 应用程序通过初始化函数 WSAStartup（）来初始化 WinSock 库，在程序结束时，使用终止函数 WSAClieanup（）释放系统资源。

– **WSAStartup（）**

该函数用于初始化 WinSock 库。语法格式如下：

```
int WSAStartup ( WORD wVersionRequested,    LPWSADATA lpWSAData );
```

参数：wVersionRequested：表示 Socket 的版本，高字节为修订版本，低字节为主版本。如 WinSocket 的版本为 2.2，则高字节为 2，低字节为 2。

pWSAData：指向 WSADATA 结构的指针，记录 Windows 套接字详细信息。

返回值：执行成功返回 0，否则返回错误代码。

WSADATA 结构表示 Winsock 库的版本信息，定义如下：

```
struct    WSADATA
          { WORD wVersion;                    //Winsokt 版本号
           WORD  wVersion;                    //WinSock.dll 支持的最高版本
            char          szDescription[ ];   //Winsokt 开发商识别信息
            char          szSystemStastus[ ]; //系统状态和配置信息
  unsigned short      szDescription[];        //一个进程最多可使用套接字数
```

```
    unsigned short      szDescription[];        //可发送最大数据报的字节数
        char *          lpVendorInfo;           //开发商专用数据结构
    };
```

例：

```
WORD        wVersionRequested;
WSADATA         wsadata={0};
int         err;
wVersionRequested=MAKEWORD(2, 2);               //宏 MAKEWORD(high, low)
err=WSAStartup (wVersionRequested，&wsadata );
if (err!=0)  {cout<<"初始化错误"；}
```

– **WSACleanup（）**

该函数用于释放 WinSock 库初始化时系统分配的资源。语法格式如下：

```
int WSACleanup (void);
```

参数：无

返回值：执行成功返回 0，否则返回 SOCKET_ERROR。

例：

```
WSACleanup (    );      //Winsock 套接字调用结束，释放套接字资源
```

应用程序中的每一次 WSAStartup（）调用都需要有一个 WSACleanup（）做清除工作，注意匹配使用。

8.1.2 创建和释放套接字

■ **创建套接字——socket（）**

功能：根据指定的网络地址族和协议，创建一个新的套接字，分配资源并返回该套接字的描述符，该描述符是一个整数的值。根据套接字描述符就可以找到对应的套接字数据结构，也叫做套接字句柄。语法格式如下：

```
SOCKET socket ( int af，    int type，    int procotol);
```

参数： af: 指定地址族，TCP/IP 使用的地址族是 AF_INET；

type: 使用的套接字类型，SOCK_STREAM、SOCK_DGRAM、SOCK_RAW 三种套接字类型；

procotol: 使用的协议，一般设为 0，表示使用默认协议（根据参数 2

的类型自动选择）。

返回值：执行成功返回整数值，不成功返回错误类型。

例：

```
SOCKET socket_id=SOCKET(AF_INET, SOCK_STREAM, 0);
                                    //创建一个流式套接字，用 socket_id 使用该套接字
```

■ **关闭套接字——closesocket( )**

功能：关闭套接字 s，释放为 s 分配的资源。格式如下：

```
int    closesocket( SOCKET s);
```

参数：s：要关闭的套接字描述符。

返回值：执行成功返回 0；否则返回 SOCKET_ERROR。

例：

```
closesocket(socket_id);
                                    //关闭 socket_id 套接字
```

8.2 WinSock 编程

网络应用程序根据应用层协议来进行开发，大部分应用层协议使用客户/服务器的体系结构，主动发起通信连接的是客户进程，被动接收连接请求并提供服务的是服务器进程。采用对等体系结构的应用层协议，其本质仍然是客户进程和服务器进程通信的结构，只不过在对等方的每一方都既有客户进程，又有服务器进程。传输层向应用层提供两种传输服务：面向连接的可靠传输服务（TCP 协议），无连接的不可靠传输服务（UDP 协议）。采用不同传输服务的客户进程和服务器进程在进行网络连接和后续的数据传输过程不一样，主要区别在于是否需要连接建立、数据传输的发起和结束的控制。

8.2.1 WinSock 开发流程

面向连接的传输服务，客户进程和服务器进程创建套接字后，必须在传输数据前建立连接，然后可以双向传输数据，最后断开连接；无连接的传输服务，客户进程和服务器进程在创建套接字后直接就可以传输数据。因此，采用不同传输服务的网络应用程序的开发过程有所不同。

前面讲过，网络应用程序的体系机构有客户/服务器（C/S）模式和对等（P2P）模式，并且本质上对等模式也是有客户进程和服务器进程。客户进程主动发起通信请求，并向服务器申请服务；服务器进程在已知端口提供某种服务，当收到客户端请求后，

响应客户端的请求并返回结果。通常服务器需要同时服务多个客户端，其通信模型如下图所示。

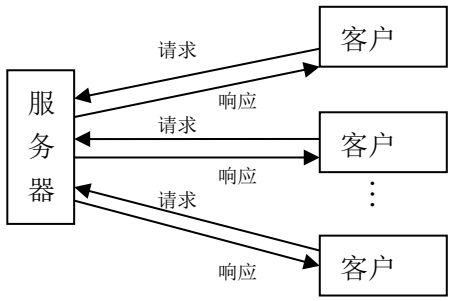


图 8-1　客户/服务器模型

WinSock 编程中，客户进程和服务器进程建立套接字 socket 后，必须按照网络协议的地址进行网络通信。TCP/IP 协议族的网络主机网络地址是“IP 地址”标识，通过 IP 地址找到主机后，还需要在主机运行的多个进程中找到要通信的进程，还需要“端口号”来标识进程。另外在加上传输层协议，一个三元组“IP 地址、传输协议、端口号”就能标识相互通信的两个网络进程。在 WinSock API 中，确定这个三元组的过程称之为绑定。绑定后，选择不同的传输层协议则对应有不同的开发流程，其具体的 WinSock API 函数也有所区别。

面向无连接的 WinSock 开发流程如下图所示：

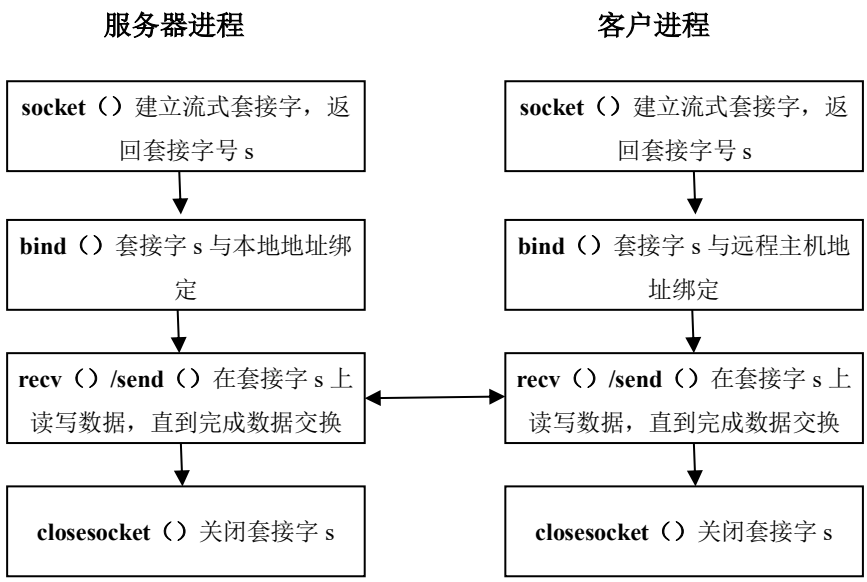


图 8-2　面向无连接的 WinSock 流程

面向连接的 WinSock 开发流程如下图所示：

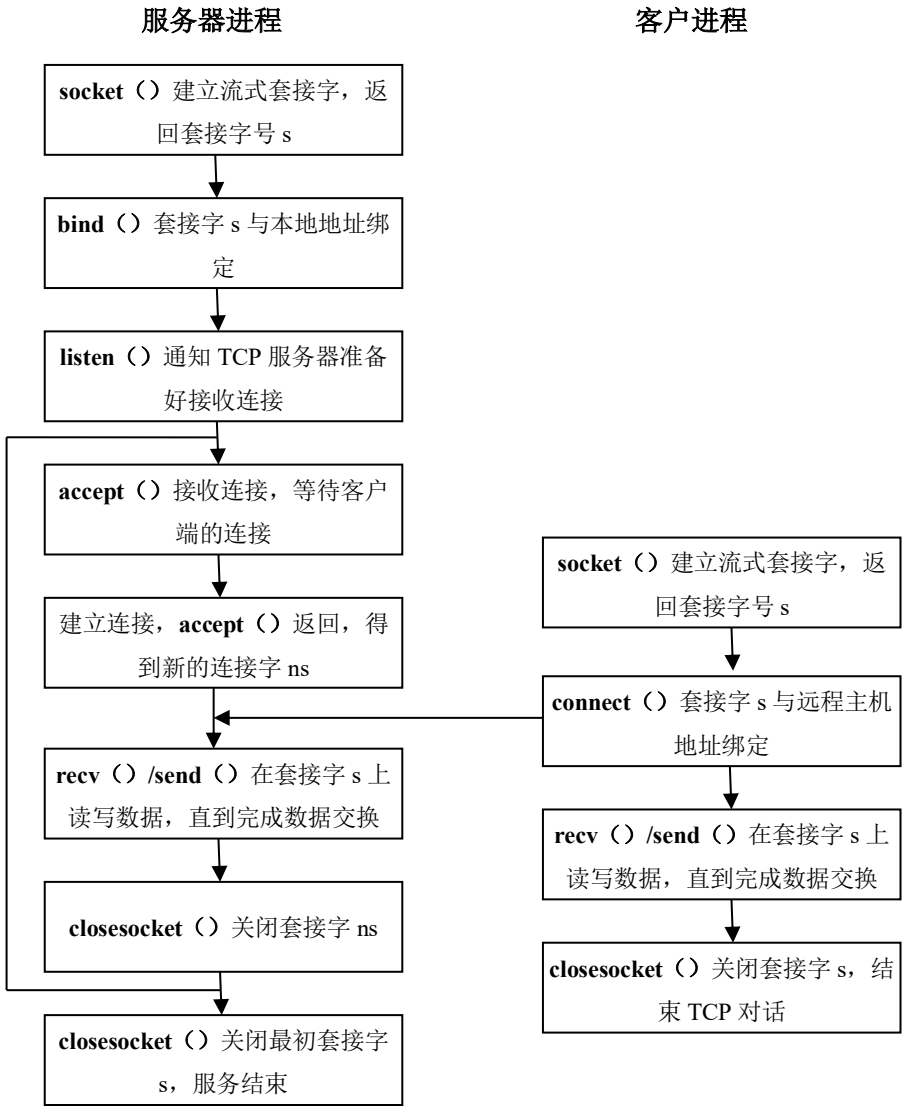


图 8-3　面向连接的 WinSock 流程

从上面的过程看到面向连接的 WinSock 服务器和客户端在数据传输之前需要建立一个连接，即 connect( )和 accept( )完成的操作。无连接的 WinSock 服务器和客户端在数据传输之前不用建立连接，没有这一个步骤，在建立 socket( )后直接就可以进行数据传输。上述 WinSock 编程流程中涉及的函数都是 WinSock API 函数，WinSock 初始化和释放等函数在上面 3.1 节中已经阐述，其他函数的具体语法和功能在下节中讲述。

8.2.2 WinSock 基本函数

■ **绑定本地地址——bind( )**

功能：将端口和地址与所创建的套接字联系起来，格式如下：

```
int    bind(SOCKET s, const struct sockaddr * name, int namelen);
```

参数：　s：未经绑定的套接字描述符；
　　name：指向 sockaddr 结构变量的指针；
　　namelen：sockaddr 地址结构的长度。
返回值：执行成功返回 0，否则返回 SOCKET_ERROR。

其中，sockaddr 结构变量（TCP/IP 地址族）的说明：

```
struct sockaddr_in{
        short      sin_family;          //指定地址族，AF_INET
      u_short      sin_port;            //16 位端口号，网络字节顺序
struct in_addr     sin_addr;            //32 位 IP 地址，网络字节顺序
        char       sin_zero[8];         //保留
                   };

struct    in_addr {
              union {
                    struct {u_char    s_b1, s_b2, s_b3, s_b4;}    S_un_b;
                    struct {u_short    s_w1, s_w2;}    S_un_w;
                    U_long         S_addr;
                    };
              };
```

in_addr 是网络字节顺序表示的主机的点分十进制表示的 IP 地址，如 IP 地址为 1.2.3.4，则在 X86 系统的计算机中存储为：ox04030201。实际编程通常使用 inet_addr( )

将字符串表示的 IP 地址转换为整数表示；或者使用 htonl( )将主机地址转换为网络字节顺序地址。

例：

```
socketaddr_in     my_addr;
unsigned short    port=20000;
my_addr.sin_family=AF_INET;
my_addr.sin_port=htonl(port);
my_addr.sin_addr.s_addr=htonl(INADDR_ANY);
int    err;
int    slen=sizeof(sockaddr);
SOCKET socket_id=SOCKET(AF_INET, SOCK_STREAM, 0);
bind(socket_id, (sockaddr* )&my_addr, slen);
                    //套接字 socket_id 与 IP 地址 INADDR_ANY 在端口 20000 绑定
```

- **请求连接——connect( )**

功能：客户进程请求与服务器进程建立连接，格式如下：

```
int    connect( SOCKET s, const struct sockaddr    * name, int namelen);
```

参数：s：已绑定，还未连接的客户进程的套接字描述符；

name： 指向要连接的服务器进程的监听套接字的 sockaddr 结构指针；

namelen：sockaddr 地址结构的长度。

返回值：执行成功返回 0，否则返回 SOCKET_ERROR。

例：

```
    SOCKET socket_id=SOCKET(AF_INET, SOCK_STREAM, 0);
struct socketadr_in    daddr;
memset( (void *)&daaar, 0, sizeof(daddr));
daddr.sin_family=AF_INET;
daddr.sin_port=htons(21);
daddr.sin_addr_s_addr=inet_addr(“202.204.60.11);
connet(socket_id, (struct socketaddr *) &daddr, sizeof(daddr));
    //客户进程套接字 socket_id 与主机 202.204.60.11 在端口 21 建立连接
```

- **服务器监听连接——listen( )**

功能：仅用于面向连接的套接字，用于流式套接字，用于服务器端。监听套接字必须绑定到特定网络地址上，该函数启动监听套接字开始监听来自客户端的连接请求，将来自客户端的连接请求放入等待队列，并且规定等待连接队列的最大长度。格式如下：

int　listen(SOCKET s, int backlog);

参数：　s：服务器端的套接字描述符（已绑定），一般称为监听套接字；
　　backlog：连接请求队列的最大长度，一般为 5。
返回值：执行成功返回 0；否则返回 SOCKET_ERROR。

例：
listen(s, 5);
　　//套接字 s 置于监听状态，处理客户请求队列最大为 5

■ **接收连接请求——accept( )**

功能：服务器进程从监听套接字 s 的等待队列中抽取第一个连接请求，创建一个与 s 同类型的新套接口与请求连接的客户进程套接字建立连接通道。格式如下：

SOCKET accept(SOCKET s, struct sockaddr * name, int * addrlen);

参数：　s：服务器进程监听套接字描述符；
　　name：指向要接收的请求连接一方的套接字 sockaddr 结构指针；
　　namelen：sockaddr 地址结构的长度。
返回值：执行成功返回 SOCKET 类型的描述符;否则返回 INVALID_SOCKET。

例：
　　sockaddr_in　saClient;
int　nSALen=sizeof(sockaddr);
SOCKET socket_id=accept(s, (sockaddr*)&saClient, &nSALen);
　　//服务器进程创建一个新的套接字 socket_id 来处理监听套接字 s 接受的一个客户连接请求，远程客户进程的套接字地址放入 saClient 存储。

■ **发送数据——send( )**

功能：向本地已经建立连接的数据报或流式套接口发送数据。格式如下：

int　send(SOCKET s, const char * buf, int len, int flags);

参数：　　s：数据发送方（与接收方已建立连接）的套接字描述符；

buf：指向用户进程的字符缓冲区（要发送的数据）的指针；

len：用户缓冲区中数据的长度，以字节为单位；

flags：执行此调用的方式，一般设置为 0。

返回值：执行成功返回实际发送出去的数据的字节总数，该数可能小于 len 规定的值；否则返回 SOCKET_ERROR。

例：

```
sum=512;                                    //假设需要接收的字节数为 512
send_num=0
while (sum>0)
    { ret= send(s, &buf[send_num], sum, 0);
                                    //发送缓冲区 buf 的数据，发送大小为 sum 字节
    if   (ret == SOCKET_ERROR)   { cout<<"发送错误"; }
    sum - = ret;                            //需要接收的字节数减 1
    send_num +=ret;                         //已发送的字节数加 1
    }
```

- **接收数据——recv( )**

功能：从本地已经建立连接的数据报或流式套接口接收数据。格式如下：

```
int   recv( SOCKET s,   const char * buf,   int len, int flags);
```

参数：　　s：数据接收方（与发送方已建立连接）的套接字描述符；

buf：用于接收数据的字符缓冲区指针；

len：用户缓冲区长度，以字节为单位；

flags：执行此调用的方式，一般设置为 0。

返回值：执行成功返回从套接字 s 实际读入到 buf 中的数据字节总数；如果连接中止，返回 0；否则返回 SOCKET_ERROR。

例：

```
sum=512;                                    //假设需要接收的字节数为 512
recv_num=0
while (sum>0)
    { ret= recv(s, &buf[recv_num], sum, 0);
```

```
                    //接收的数据放在缓冲区 buf，接收数据最多不超过 sum 字节
if   (ret == SOCKET_ERROR)   { cout<<"接收错误"；}
else   if (ret ==0)   { cout<<"连接意外被关闭"；}
recv_num +=ret;                                    //已经接收的字节数加 1
sum - = ret;                                       //需要接收的字节数减 1
}
```

■ **按照指定目的地向数据报套接字发送数据——sendto( )**

功能：专用于数据报套接字，用于向发送端的本地套接字发送一个数据报。套接字将数据下交给传输层的 UDP 协议，由它向对方发送。格式如下：

```
int   sendto( SOCKET s, const char * buf, int len, int flags，struct sockaddr* to, int
          tolen);
```

参数： s：数据发送方的数据报套接字描述符；
buf：指向用户进程的发送缓冲区的字符串指针；
len：用户发送缓冲区中数据的长度，以字节为单位；
flags：执行此调用的方式，一般设置为 0；
to：指向接收数据报的目的套接字的 sockaddr 结构指针（网络地址）；
tolen：to 地址的长度，等于 sizeof(struct sockaddr)。
返回值：执行成功返回实际发送的字节总数，该数可能小于 len 规定的值；否则返回 SOCKET_ERROR。

例：

```
char   sendbuf[1024];
int    buflen=1024;
sockaddr_in   sclient;
int    sclientsize=sizeof(sclient);
int    sret;
sret = sendto( s, sendbuf, buflen, 0，(sockaddr *) &sclient, sclientsize);
```

■ **从数据报套接字接收数据——recvfrom( )**

功能：从 s 套接字的接收缓冲区队列中，取出第一个数据报，将它放入用户进程的缓冲区 buf，最多不超过用户缓冲区的大小。如果数据报大于用户缓冲区长度，则缓冲区中只有数据报的前面部分，后面的数据会丢失，并且返回错误。格式如下：

int recvfrom(SOCKET s, const char * buf, int len, int flags, struct sockaddr *from, int* fromlen);

参数： s：接收端的数据报套接字描述符，从该套接字接收数据报；

buf：用于接收数据报的用户进程的字符缓冲区指针；

len：用户接收缓冲区最大长度，以字节为单位；

flags：执行此调用的方式，一般设置为 0。

from：指向 sockaddr 结构指针（网络地址），调用执行成功则返回发送方的 sockaddr 结构的网络地址；

fromlen：整数指针，调用执行成功返回 from 中的网络地址长度。

返回值：执行成功返回从套接字 s 实际读入到 buf 中的数据字节总数；否则返回 SOCKET_ERROR。

例：

```
char    recvbuf[1024];
int     buflen=1024;
sockaddr_in    sclient;
int     sclientsize=sizeof(sclient);
int     sret;
sret = recv( s, recvbuf, buflen, 0，(sockaddr *) &sclient, &sclientsize);
```

■ **停止数据传输——shutdown( )**

功能：停止数据传输用于停止 TCP 套接字的数据传输；UDP 套接字不用 shutdown 停止数据传输，直接调用 closesocket 释放资源。格式如下：

int shutdown(SOCKET s, int how);

参数：s：要关闭的套接字描述符；

how：设成 SD_SEND，表示发送方不能发送任何数据，可以调用 recv 函数接收数据，直到 recv 返回 0，就可以释放套接字资源了。

返回值：执行成功返回 0；否则返回 SOCKET_ERROR。

例：

```
shutdown(s，SD_SEND);
                                        //停止 s 套接字的数据发送
```

8.2.3 WinSock 的 I/O 模型

WinSock 进行发送数据和接收数据的时候，本质上是应用进程在进行数据的 I/O 操作，只不过这些数据是网络数据。应用进程进行 I/O 操作的时候，有两种工作模式：阻塞模式或非阻塞模式，也称为同步模式或异步模式。阻塞模式指一个被设为阻塞模式的套接字进行某些 I/O 操作时，如果这些操作不能立即完成，则它们将进入等待状态，直到操作完成或发生错误时才返回。非阻塞模式则不一样，当被设为非阻塞模式的套接字进行某些 I/O 操作时，如果这些操作不能立即完成，也会马上返回值，返回值表示了操作执行的结果，操作顺利完成或发生某种错误。

阻塞模式下，应用进程调用了一个 WinSock 的 I/O 函数时，在 I/O 操作完成前，执行 I/O 操作的 WinSock 函数会一直等候，不会立即返回调用它的程序，独占 CPU 控制权，在 I/O 操作完成前，其他代码都无法执行。应用进程这时候即处于阻塞状态，既不能响应用户操作，也不能响应其他程序，只能一直等待下去。如果服务器进程采取该种模式，则会导致服务器创建大量线程，线程的调度和切换会占据大量的 CPU 资源。

非阻塞模式下，应用进程调用了一个 WinSock 的 I/O 函数时，无论 I/O 操作是否完成，执行 I/O 操作的 WinSock 函数都会立即返回调用它的程序。这时就有两种情况：I/O 操作恰好完成，I/O 调用成功，完成输入输出操作；另一种是大多数情况下，调用失败，返回 WSAEWOULDBLOCK 错误，表示 I/O 操作的条件尚不具备，没时间完成输入输出操作。非阻塞模式下的应用进程 I/O 函数调用会频繁返回错误，所以编程要注意函数调用失败的错误处理策略。

WinSock 的 I/O 模型可以分为三大类：阻塞 I/O 模型、就绪通告 I/O 模型和重叠 I/O 模型（异步 I/O 模型）。

- **阻塞 I/O 模型**

阻塞 I/O 模型是最简单的 WinSock I/O 模型，应用进程每次处理网络输入输出操作（如：send( )或 recv( )）时，进程都将挂起，直到 I/O 操作完成。

- **就绪通告 I/O 模型**

就绪通告 I/O 模型是应用进程处理网络输入输出操作时，由 WinSock 负责检测多个套接字的状态，并在数据准备好时通知有关进程，进程收到通知后去调用相关的 WinSock 函数，这时应用进程调用 WinSock 函数通常都会成功（如：send( )或 recv( )）。

就绪通告 I/O 模型又可细分为三种 I/O 模型：select 模型、WSAAsyncSelect 模型、WSAEvenSelect 模型。

- **重叠 I/O 模型**

重叠 I/O 模型是更复杂的 I/O 模型。应用进程处理网络输入输出操作时，由系统来完成 I/O 操作，进程无需等待操作完成，在系统执行 I/O 操作的同时继续执行其他操作，系统完成 I/O 操作后通知进程，因此称作重叠 I/O 模型。重叠 I/O 模型和就绪通告 I/O 模型都是将输入输出操作交给别人负责，当操作准备好了后被通知，那区别在哪里呢？主要区别是：就绪通告 I/O 模型被通知后，需要再次调用相关的 I/O 函数完成操作；重叠 I/O 模型被通知时，I/O 操作已经成功完成，不需要再调用函数。因此重叠 I/O 模型也称作完成通告模型。

根据重叠 I/O 模型完成通告方式的不同，重叠 I/O 模型也可细分为三种 I/O 模型：重叠 I/O +事件通告模型、重叠 I/O +回调通告模型、重叠 I/O+完成端口通告模型。

下面讲述 WinSock I/O 模型有关的常用函数以及使用规则。

■ **阻塞 I/O 模型**

阻塞 I/O 模型是最简单的 I/O 模型，应用进程直接调用 WinSock 的数据发送函数 send( )和数据接收函数 recv( )来完成操作即可，数据没有准备好就直接被挂起，因此没有对应的 WinSock 函数对应 I/O 操作的检测和执行。

新创建的套接字在默认情况下处于阻塞模式，改变套接字为非阻塞模式的函数是：

int ioctlsocket(SOCKET s, long cmd, u_long * argp)；

参数： s：需改变阻塞模式的套接字 s；
cmd：必须为 FIONBIO，表示要改变的是阻塞模式；
argp：指向一个无符号整数，0 为阻塞；1 为非阻塞。

返回值：整型数值。

例程：

```
u_long    nNoBlock = 1;
ioctlsocket( s , FIONBIO, &nNoBlock);
                                    //设置套接字 s 为非阻塞模式
```

■ **就绪通告 I/O 模型——select( )**

功能：实现对多个套接字 I/O 的管理，检测一个或多个套接字状态，判断套接字上是否存在数据，或者能否向一个套接字写入数据。只有在条件满足时，才对套接字进行 I/O 操作，从而避免 I/O 函数调用失败，频繁产生错误。格式如下：

int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set * exceptfds, const struct

timeval* timeout);

参数： nfds：保持与早期的 Berkeley 套接字应用程序的兼容，一般忽略；

readfds：指向要做读检测的指针，检测以下条件是否满足：

- 有数据到达，可以读入；
- 连接已经关闭、重设或终止；
- 如已调用 listen，且一个连接正建立，则 accept 调用会成功；

writefds：指向要做写检测的指针，检测以下条件是否满足：

- 发送缓冲区已空，可以发送数据
- 如已完成对一个非锁定连接调用的处理，连接就会成功；

exceptfds：指向要检测是否出错的指针，检测以下条件是否满足：

- 如已完成对一个非锁定连接调用的处理，连接会失败；
- 有带外数据可供读取。

timeout：指向 timeval 结构的指针，决定 select 等待 I/O 操作完成的最长时间。如 timeout 是一个空指针，则 select 调用会无限期等待，知道至少有一个符合指定条件的套接字。

返回值：执行成功返回所有 fd_set 集合中符合条件的套接字句柄的总数；如果超过 timeval 设定的时间，则返回 0；如果调用失败，则返回 SCCKET_ERROR。

其中的几个数据结构说明如下。

fd_set 数据结构类型代表一系列特定套接字集合，WinSock 提供了一系列宏操作对 fd_set 数据类型进行操作，参数 s 是要检查的套接字，参数 set 是 fd_set 集合类型的指针：

- FD_CLR(s，*set)：从 set 中删除套接字 s；
- FD_ISSET(s，*set)：检查 s 是否是 set 集合的一名成员，是则返回 TRUE；
- FD_SET(s，*set)：将套接字 s 加入 set 集合；
- FD_ZERO(*set)：将 set 初始化成空集合。

timeval 数据结构类型代表等待时间，具体结构定义如下：

```
struct  timeval {
        long    tv_sec;                 //以秒为单位指定等待时间
        long    tv_usec;                //以毫秒为单位指定等待时间
        };
```

select 模型的操作步骤：

[1] 使用 FD_ZERO，初始化感兴趣的每一个 fd_set 集合；

[2] 使用 FD_SET，将要检查的套接字句柄添加到每个 fd_set 集合；

[3] 调用 select 函数，然后等待。select 函数会修改每个 fd_set 结构，在 fd_set 集合中返回符合条件的套接字；

[4] 根据 select 函数返回值，使用 FD_ISSET 检查每个 fd_set 集合，判断是否还存在未完成 I/O 操作的套接字；

[5] 对步骤[4]中返回的未完成 I/O 操作的套接字进行处理，然后返回[1]。

例：

```
while (TRUE)
{ FD_ZERO(&fdread);                                    //清除套接字集合变量
 FD_SET(s, &fdread);                                   //将套接字 s 添加到 fdread 集合
 if ( ret = select( 0, &fdread, NULL, NULL, NULL)) == SOCKET_ERROR)
                                                       //调用 select 函数，检查 s 是否有数据可读
        { cout<< “错误”;                                                    // 错 误 处 理
代码
        }
   if ( ret > 0)
        { if (FD_ISSET(s, &fdread))
             {                                         //对套接字进行读操作
             }
        }
}
```

■ **就绪通告 I/O 模型——WSAAsyncSelect( )**

功能：就绪通告 I/O 模型的一种，select( )使用 fd_set 来通知进程 I/O 操作已经就绪；WSAAsyncSelect( )使用 Windows 窗口过程和消息的网络事件通知。当针对一个套接字的一个 I/O 操作就绪的时候，系统会发送一个窗口消息给进程，进程在响应该消息时再去调用相关的 I/O 函数，且一般都会立即成功，不会发生阻塞。语法格式如下：

```
int WSAAsyncSelect( SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

参数：　　s：指定的套接字 s；

hWnd：指定一个窗口或对话框句柄，网络事件发生后，该窗口或对话

框会收到消息，并自动执行对应的回调函数；

wMsg：指定在发生网络时间时，打算接收的消息；

lEvent：指定一个位掩码，代表应用程序感兴趣的一系列事件。

返回值：

WSAAsyncSelect 函数的网络事件类型说明如下：

- FD_READ：应用程序想接收是否有可读的通知，以便读入数据；
- FD_WRITE：应用程序想接收是否有可写的通知，以便发送数据；
- FD_OOB：应用程序想要接收是否有带外数据到达的通知；
- FD_ACCEPT：应用程序想接收与进入的连接请求有关的通知；
- FD_CONNECT：应用程序想接收一次连接请求操作完成的通知；
- FD_CLOSE：应用程序想接收与套接字关闭的通知。

使用 WSAAsyncSelect 函数之前，必须创建一个窗口，并为窗口提供一个窗口回调函数。设置好窗口框架后，创建套接字，调用 WSAAsyncSelect 函数，指定关注的套接字、窗口句柄、打算接收的消息和应用程序感兴趣的套接字时间。当 WSAAsyncSelect 函数执行时，注册的套接字时间之一发生时，指定窗口会收到指定的消息，并自动执行回调函数，回调函数的代码由用户编写以处理响应事件。

窗口回调函数的定义如下：

LRESULT CALLBACK Windowproc(HWND hWnd， UINT uMsg, WPARAM wParam, LPARAM lParam);

参数：hWnd：指示触发回调函数的窗口的句柄；

uMsg,：引发调用此函数的消息；

wParam：指示在该参数上发生了一个网络事件的套接字；

lParam：lParam 的低位表示已经发生的网络事件，高位表示可能出现的任何错误代码。

调用 WSAAsyncSelect( )会自动把套接字设为非阻塞模式，因此调用 WinSock 的 I/O 函数（如：send( )和 recv( )）时，要注意检查 WSAWOULDBLOCK 错误；使用 WSAAsyncSelect( )系统会通过 FD_CLOSE 来告知对方关闭了连接，所以不用再判断 recv 是否返回 0；recv 会重新触发 WinSock 对 FD_READ 消息的发送，因此可以指定接收 0 字节，这样调用 recv 虽然可能会失败，但是会再次触发 WinSock 对发送缓冲区中有用消息的发送；使用 WSAAsyncSelect( )系统会通过 send( )来发送缓冲区的数据，FD_WRITE 消息表示发送缓冲区有可用空间，send( )可以成功，因此进程在收到了第

一个 FD_WRITE 消息后，可以一直调用 send( )来发送数据，直到 send 返回 WSAEWOULDBLOCK 为止，表示发送缓冲区已满。

■ **就绪通告 I/O 模型——WSAEventSelect( )**

功能：就绪通告 I/O 模型的一种，WSAEventSelect()既有 select( )的特点，也有 WSAAsyncSelect( ) 的特点。和上述两个就绪通告的就绪通告方式不同，WSAEventSelect( )使用 Windows 的事件内核对象来作为就绪通告的手段，接收以事件为基础的网络事件通知。和 WSAAsyncSelect（）类似，应用程序也要调用一个函数来注册感兴趣的套接字以及 I/O 消息，其区别在于网络事件会投递到一个事件对象句柄，而不是一个窗口。其语法格式定义如下：

int WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents);

参数： s：指定的套接字 s；

hEventObject：指定与套接字关联在一起的事件对象（用 WSACreatEvent 创建的那个事件）；

lNetworkEvents：对应一个位掩码，指定应用程序感兴趣的各种网络事件，与 WSAAsyncSelect 中的 lEvent 用法相同。

返回值：

WSAEventSelect 函数的网络事件类型说明同 WSAAsyncSelect 函数中定义的一样。使用 WSAEventSelect 函数之前，必须首先创建一个事件对象。可以调用 WSACreatEvent 函数创建，其定义如下：

WSAEVENT WSACreatEvent(void);

函数的返回值是一个创建好的事件对象句柄。

和 WSAAsyncSelect( )一样，WSAEventSelect( )的调用也会自动把套接字设为非阻塞模式；通过 FD_CLOSE 来告知对方关闭了连接；recv 会重新触发 WinSock 对 FD_READ 消息的发送；使用 FD_WRITE 消息表示发送缓冲区有可用空间等特点。另外，WSAEventSelect( )还通过使用函数 WSAWaitForMultipleEvents 来等待多个事件内核对象，该函数等待事件的上限值为 64，即一个进程最多只能服务 64 个客户连接。

■ **重叠 I/O 模型**

WinSock 2 引入了新的函数和结构来实现 I/O 操作的异步控制，因此有了重叠 I/O

模型。重叠 I/O 模型的原理是应用程序使用一个重叠的数据结构，一次投递多个 I/O 请求，由操作系统完成 I/O 操作后通知应用程序。Windows 操作系统提供了 4 种方式来通知应用程序 I/O 操作已完成，分别是：对设备内核对象设置信号；对事件内核对象设置信号；调用一个回调函数；使用 I/O 完成端口（IOCP）。根据这 4 种通知方式分为具体的 4 中重叠 I/O 模型。

为了适用重叠 I/O 模型，WinSock 2 中新定义的函数和结构和前面讲述的 WinSock 基本函数原理和使用方式基本类似，有的增加了新的成员，有的结构是用于异步 I/O 控制新增加的，具体语法格式如下所述：

- **创建套接字 WSASocket（）**

SOCKET WSASocket(int af, int type, int protocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwflags);

参数：af，type，protocol：和 socket( )中含义完全一样；

lpProtocolInfo：指向 WSAPROTOCOL_INFO 结构，指定套接字特性；

g：预留字段；

dwflags：指定套接字属性，重叠 I/O 模型中为 WSA_FLAG_OVERLAPPED。

返回值：执行成功返回套接字句柄；否则返回 INVALID_SOCKET。

- **WSAOVERLAPPED 结构**

重叠 I/O 模型的核心是重叠数据结构 WSAOVERLAPPED，该结构与 WinSock 1 的 OVERLAPPED 结构兼容。其数据结构定义如下：

```
struct  _WSAOVERLAPPED
    {DWORD    Internal;          //重叠 I/O 实现的实体内部字段，使用
                                  socket 的时候，被底层操作系统使用
     DWORD    InternalHigh;      //重叠 I/O 实现的实体内部字段，使用
                                  socket 的时候，被底层操作系统使用
     DWORD Offset;               //使用 socket 时被忽略
     DWORD OffsetHigh;    //使用 socket 时被忽略
     WSAEVENT    hEvent;        //事件对象句柄，把 Windows 事件对象
                                  关联到 WSAOVERLAPPED 结构
     };
```

重叠 I/O 模型下，套接字的读写调用会立即返回，系统会自动完成具体的 I/O 操作。

应用程序可以同时发出多个读写调用，当系统完成 I/O 操作后，会将 WSAOVERLAPPED 的 hEvents 置为授信状态，通过 WSAWaitFor- MultipleEvents( )函数来等等 I/O 完成通知，并在得到通知信号后，调用 WSAGetOverlappedResult( )来查询 I/O 操作结果，进行相应处理。

– **数据输入输出函数**

重叠 I/O 模型中，使用 WSASend( )、WSASend To( )、WSARecv( )和 WSARecvFrom( )来发送和接收数据。下面以 WSASend( )为例，说明这些函数在重叠 I/O 模型中的定义和使用规则：

```
int WSASend( SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
             LPDWORD lpNumberOfBytesSent, DWORD dwFlags,
             LPWSAOVERLAPPED lpOverlapped,
             LPWSAOVERLAPPED_COMPLETION_ROUTINE
             lpCompletionRoutine );
```

参数：s：标识一个已连接套接字的描述符；

lpBuffers：指向 WSABUF 结构数组的指针，WSABUF 结构包含缓冲区指针和缓冲区的大小；

dwBufferCount：记录 lpBuffers 数组中 WSABUF 结构的数目；

lpNumberOfBytesSent：如果发送操作立即完成，则为发送数据字节数的指针；

dwFlags：标志位，和 send( )调用的 flag 字段类似；

lpOverlapped：指向 WSAOVERLAPPED 结构的指针，该参数对非重叠套接字无效；

lpCompletionRoutine：指向发送操作完成后调用的完成例程的指针，该参数对非重叠套接字无效。

返回值：执行成功返回 0；否则，返回 SOCKET_ERROR。

– **WSAWaitForMultipleEvents 函数**

该函数用来等待一个或所有事件转变为“已传信”状态。函数的返回值是事件对象数组中的序号，通过该返回值检索事件对象数组可得到“已传信”状态事件和对应的套接字，然后就调用 WSAGetOverlappedResult( )得到 I/O 操作结果。该函数定义如下：

```
DWORD WSAWaitForMultipleEvents( DWORD cEvents, const WSAEVENT *
```

lphEvents, BOOL fWaitAll, DWORD dwTimeout, BOOL fAlertable);

参数： cEvents：等候事件的总数量；

lphEvents：事件数组的指针；

fWaitAll：设为 TRUE 表示事件数组中所有事件被传信时，函数才返回；设为 FALSE 表示事件数组中任意一个事件被传信，函数就返回；

dwTimeout：设置超时时间，如果超时，函数返回 WSA_WAIT_TIMEOUT；如设置为 0，函数会立即返回；如设置为 WSA_INFINITE，则只有某个事件被传信后才返回；

fAlertable：在完成例程方式中使用，事件通知时设为 FALSE。

返回值：如返回 WSA_WAIT_ TIMEOUT，继续等待；返回 WSA_WAIT_ FAILED，出现错误，检查 cEvents 和 lphEvents 两个参数。

使用 WSAWaitForMultipleEvents 函数要注意，该函数只能等待 64 个事件，超过 64 个事件，要另外创建线程池。

– **WSAGetOverlappedResult 函数**

该函数用来查询套接字上重叠 I/O 操作完成的结果。函数定义如下：

BOOL WSAGetOverlappedResult (SOCKET s, LPWSAOVERLAPPED lpOverlapped, LPDWORD lpcbTransfer, BOOL fWait, LPDWORD lpdwFlags);

参数： s：套接字句柄；

lpOverlapped：套接字 s 关联的 WSAOVERLAPPED 结构；

lpcbTransfer：指向字节计数指针，接收一次重叠发送/接收实际字节数；

fWait：确定函数是否等待标志：设为 TRUE，直到操作完成函数才返回；设为 FALSE，操作仍处于未完成状态，函数返回 FALSE；

lpdwFlags：接收完成状态的附加标志。

返回值：执行成功，返回 TRUE；否则，返回 FALSE，可调用函数 WSAGetLastError( )获取失败原因。

8.2.4 WinSock 辅助函数

由于要在网络主机之间进行数据交换，除了和网络协议栈交互之外，套接字编程还涉及到很多方面的问题，如网络信息表达和主机内部信息表达的一致性问题、IP 地址表示问题等等。套接字 API 也提供了一些辅助函数来解决这些问题，常用的 Winsock

辅助函数如下所述。

■ **字节顺序转换函数**

不同计算机的多字节数据的存储顺序是不同的，有的机器按照低位字节优先存储（低序字节存储在低地址），有的是高位字节优先存储（高序字节存储在低地址），称之为本机字节顺序。网络协议中对于多字节数据在首部字段中的表示规定高位字节优先存储，称之为网络字节顺序。

套接字中必须使用网络字节顺序，因此网络数据到达本机后要转换成本机字节顺序；同理，本机数据发送到网络上必须转换成网络字节顺序。在 sockaddr_in 结构中，sin_addr 表示 IP 地址，封装在 IP 首部；sin_port 表示端口号，封装在 TCP 或 UDP 首部；这两个地址都是网络字节顺序。sin_family 表示地址族类型，具有本地含义，是本机字节顺序。

套接字 API 提供了以下 4 种字节顺序转换函数，方便进行相互转换。

- **u_long htonl (u_long hostlong)**：本机字节顺序转为网络字节顺序，hostlong 是 32 位的 IP 地址（长整数）；
- **u_short htons (u_short hostshort)**：本机字节顺序转为网络字节顺序，hostshort 是 16 位的端口号（短整数）；
- **u_long ntohl (u_long netlong)**：网络字节顺序转换为本机字节顺序，netlong 是 32 位的 IP 地址（长整数）；
- **u_short ntohs (u_short netshort)**：网络字节顺序转为本机字节顺序，netshort 是 16 位的端口号（短整数）；

■ **IP 地址转换函数**

网络 IP 地址通常用点分十进制形式表示，而在套接字的 sockaddr_in 结构中，IP 地址是无符号长整型数。套接字 API 提供了以下 2 种 IP 地址转换函数。

- **u_long inet_addr(const char * cp)**：cp 是一个字符串指针，指向点分十进制的 IP 地址；返回值是无符号长整型数，表示的是网络字节顺序 IP 地址。
- **char* inet_ntoa(struct in_addr in)**：in 是无符号长整型数，表示的是网络字节顺序 IP 地址；返回值是指向点分十进制表示的 IP 地址的字符指针。

■ **套接口信息查询函数**

套接字 API 中还提供了一组信息查询函数，方便用户查询套接口相关的网络地址信息等，如，主机域名对应的 IP 地址的查询函数。常用的有：

- **int gethostname(char* name, int namelen)**：该函数调用可以得到本地主机

名，将本地主机名以字符串的形式存放到 name 指定的缓冲区。参数：name 是指向主机域名的字符串，namelen 是缓冲区的长度；返回值：执行成功返回 0，错误则返回 SOCKET_ERROR。

- **struct hostent * gethostbyname(const char * name)**：该函数调用可以得到对应与给定主机名的主机信息，函数调用返回指向 hostent 结构的指针，hostent 结构包含主机名、返回地址的类型（一般是 AF-INET）、地址长度的字节数和已符合网络字节顺序的主机网络地址等。参数：name 是指向主机域名的字符串；返回值：hostent 结构的指针。

```
struct    hostent {
            char*     h_name;                          //正规的主机名字
            char**    h_aliases;            //以空指针结尾的可选主机名队列
            short      h_addrtype;                  //地址族类型 AF_INET
            short      h_length;       //每个地址的字节长度，IP 地址为 4
            char**     h_addr_list;        //以空指针结尾的主机地址列表
            };
```

- **struct hostent * gethostbyaddr(const char * addr, int len, int type)**：该函数调用可以根据 IP 地址得到相应的主机信息，函数调用返回指向 hostent 结构的指针。参数：addr 是指向网络字节顺序的 IP 地址的指针，len 是地址的长度，一般是 4，type 是地址类型，应为 AF_INET；返回值：hostent 结构的指针。
- **struct servent * getservbyname(const char * name, const char * proto)**：该函数调用可以得到对应与给定服务名和协议名的服务信息，函数调用返回指向 servent 结构的指针，servent 结构包含服务名、端口号、协议名等。参数：name 是指向服务名的字符指针，proto 是可选项，是指向协议名的字符指针；返回值：servent 结构的指针。

```
struct    servent {
            char*     s_name;              //正规的服务名
            char**    s_aliases;           //以空指针结尾的可选服务名队列
            short      s_port;         //连接该服务的端口号，网络字节顺序
            char*      s_proto;        //连接该服务时用到的协议名
            };
```

- **struct servent * getservbyport(int   port, const char * proto)**：该函数调用可

以得到对应与给定端口号和协议名的服务信息，函数调用返回指向 servent 结构的指针。参数：port 是给定的端口号，proto 是可选项，是指向协议名的字符指针；返回值：servent 结构的指针。

- **struct protoent * getprotobyname(const char * name)**：该函数调用可以得到对应与给定协议名的相关协议信息，函数调用返回指向 protoent 结构的指针，protoent 结构包含协议信息。参数：name 是指向协议名的字符指针；返回值：protoent 结构的指针。

```
struct   protoent {
            char*     p_name;          //正规的协议名
            char**    p_aliases;       //以空指针结尾的可选协议名队列
            short      p_proto;        //协议号，主机字节顺序
            };
```

- **struct protoent * getprotobynumber(int number)**：该函数调用可以得到对应与给定协议号的相关协议信息，函数调用返回指向 protoent 结构的指针。参数：number 是协议号；返回值：protoent 结构的指针。

■ **WinSock 错误处理函数**

WinSock 函数在执行时，执行成功会有正常的返回值，如果出错，则会返回 SOCKET_ERROR。具体的出错信息有专门的代码表示，可以通过套接字 API 的上一次错误查询得到错误代码。

- **int WSAGetLastError(void);** 函数调用返回本线程上一次 Winsock 函数调用时的错误代码。

  常见的错误代码：

  WSANOTINITIAISED：使用该函数前，没有成功调用 WSAStartup( )

  WSAENTDOWN：Winsock 检测到网络子系统已经失效

  WSAEINPROGRESS：正在运行的 Winsock 调用被阻塞，系统没时间处理

8.3 WinSock 编程实例

如上述开发流程所示，服务器端和客户端开发相应的应用程序。下面是一个 WinSock 编程的 C/S 结构的网络应用程序实例。

WinSock 服务器端流程如下：

加载套接字->创建监听的套接字->绑定套接字->监听套接字->处理客户端相关请求。

```
---------------------------------------------------------------------------------------------------
#include <Winsock2.h>
#include <stdio.h>
   void main()
{                                                                          //加载套接字
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested=MAKEWORD(1,1);
              err=WSAStartup(wVersionRequested,&wsaData);
    if (err!=0)
    {         return;
    }
     if (LOBYTE(wsaData.wVersion)!=1|| HIBYTE(wsaData.wVersion)!=1)
       {
          WSACleanup();
          return;
    }
                                                                          //创建监听的套接字
    SOCKET sockSrv=socket(AF_INET,SOCK_STREAM,0);
    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr=htonl(INADDR_ANY);        //把 U_LONG 的主机
                                                字节顺序转换为 TCP/IP 网络字节顺序
    addrSrv.sin_family=AF_INET;
    addrSrv.sin_port=htons(6000);
                                                                          //绑定套接字
    bind(sockSrv,(SOCKADDR*)&addrSrv,sizeof(SOCKADDR));
                                          //将套接字设置为监听模式，准备接受用户请求
    listen(sockSrv,5);
    SOCKADDR_IN addrClient;
    int len=sizeof(SOCKADDR);
    printf("%s\n","welcome,the serve is started...");
    while (1)
```

```
    {
                                                        //等待用户请求到来
        SOCKET sockConn=accept(sockSrv,(SOCKADDR*)&addrClient,&len);
        char sendBuf[100];
        sprintf(sendBuf,"welcome %s to
http://unblue2008.javaeye.com",inet_ntoa(addrClient.sin_addr));
                                                        //发送数据
         send(sockConn,sendBuf,100,0);
        char revBuf[100];
                                                        //接收数据
         recv(sockConn,revBuf,100,0);
                                                    //打印接收的数据
         printf("%s\n",revBuf);
                                                        //关闭套接字
         closesocket(sockConn);
    }
}
```

WinSock 客户端同样需要先加载套接字，然后创建套接字，不过之后不用绑定和监听了，而是直接连接服务器，发送相关请求。Socket 客户端程序如下：

```
#include <Winsock2.h>
#include <stdio.h>
void main()
{                                                       //加载套接字
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested=MAKEWORD(1,1);
    err=WSAStartup(wVersionRequested,&wsaData);
    if (err!=0)
    {    return;
```

```
    }
    if (LOBYTE(wsaData.wVersion)!=1|| HIBYTE(wsaData.wVersion)!=1)
    {    WSACleanup();
         return;
    }
                                                                    //创建套接字
    SOCKET sockClient=socket(AF_INET,SOCK_STREAM,0);
    SOCKADDR_IN addrSrv;
    addrSrv.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");       //把 U_LONG 的主机
                                          字节顺序转换为 TCP/IP 网络字节顺序
    addrSrv.sin_family=AF_INET;
    addrSrv.sin_port=htons(6000);
                                                                    //向服务器发送请求
    connect(sockClient,(SOCKADDR*)&addrSrv,sizeof(SOCKADDR));
                                                                    //接收数据
    char recBuf[100];
    recv(sockClient,recBuf,100,0);
    printf("%s\n",recBuf);
                                                                    //发送数据
    send(sockClient,"this is  客户机",strlen("this is  客户机")+1,0);
                                                                    //关闭套接字
    closesocket(sockClient);
    WSACleanup();
}
```

第9章 网络课程设计任务

9.1 网络课程设计原则

计算机网络课程设计可以有两种选择：第一种选择需要完成两个任务，包括 9.2.1 节的题目，并从 9.2.2 节中选择一个题目；第二种选择需要完成一个任务，即从 9.2.3 节创新题中选择一个题目。

课程设计采取小组的形式，每组 3-5 人，共同完成设计内容。以小组形式提交课程设计报告和源程序。

9.2 网络课程设计任务

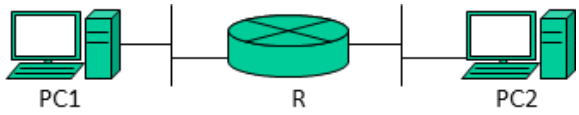
9.2.1 数据分组的发送和解析

设计目的：

- ARP 协议用于已知邻居的 IP 地址时得到其 MAC 地址；ICMP 协议是网络层的重要协议；TCP 是运输层的可靠传输协议。通过编程封装、发送、捕获并解析这些协议的数据分组，加深对网络协议的理解，掌握 ARP/ICMP/TCP 的协议数据结构和工作原理及其对协议栈的贡献。

设计要求：

- 按照下图所示的拓扑结构连接网络设备和 PC 机（也可再加入交换机/路由器扩展网络拓扑结构），并对其进行网络配置，测试 PC 机之间的连通性，保证 2 台 PC 机能正常进行网络通信。



- 如果不具备上述实验条件，则可使用以太网或 WLAN 技术建立一个局域网。例如，家庭环境可以使用无线路由器或手机热点组建一个 WLAN，用笔记本电脑访问同一 WLAN 内的无线路由器或手机，也可通过无线路由器或手机热点访问互联网。
- 选择 ARP、ICMP 或 TCP 之一，在一台计算机上编写、编译和运行程序，使其访问另一台计算机（可以是路由器）。两台计算机可以位于一个网络内（ARP/ICMP/TCP），也可以在不同网络中（ICMP/TCP）。
- 程序功能：①根据 ARP/ICMP/TCP 协议数据的结构，封装成数据帧发送给另一台计算机（可以是手机）；②捕获网络中包含 ARP/ICMP/TCP 协议数据的数据帧，解析协议数据的内容，并在标准输出中显示报文首部字段的内容，同

时写入日志文件。

- 以命令行或图形界面形式运行程序。
- 运行程序的同时开启 Wirshark 抓包软件，检验本地计算机发出与收到的数据分组。

设计分析：

- 使用原始套接字或者 WinPcap 实现（也可使用和 WinPcap 功能近似的库，如 jpcap 或 libpcap 等）。
- 定义 ARP/ICMP/TCP 首部的数据结构。
- 自定义并填充数据包，发送数据包，捕获数据包。

9.2.2 网络小应用程序

课程设计的第二个任务是使用网络开发技术开发网络小应用程序。要求基于运输层或其下层协议进行开发，而不能直接调用高级语言封装好的协议库或插件等。下面的题目是一些可选择的网络应用程序，各组可以任选一个，注意各题目难度和所需环境不一样，请根据自己小组的实际情况进行选择。

题目 1：FTP 客户端

设计目的：

- 设计并实现一个 FTP 客户端程序，了解 FTP 协议的基本原理和工作过程，加深对 TCP 协议和流式套接字编程的理解。

设计要求：

- 以命令行或图形界面形式运行程序。
- 要求在 FTP 客户端程序中**至少**实现目录变换（CWD），列表（LIST），下载（RETR）功能等。
- 输出内容：FTP 客户端与服务器交互过程中的命令与应答信息。下载公共 FTP 服务器的文件（学校 FTP 网址：202.204.60.11，不限于学校的 FTP 网站）。

题目 2：FTP 服务器

设计目的：

- 设计并实现一个 FTP 服务器的程序，了解 FTP 协议的基本原理和工作过程，加深对 TCP 协议和流式套接字编程的理解。

设计要求：

- 在后台运行服务器程序。
- 支持多个客户端同时连接服务器。
- 要求在 FTP 服务器程序中至少实现目录变换（CWD），列表（LIST），下载（RETR）功能。
- 输出内容：使用 Windows 或 Linux 命令行下的 FTP 客户端程序或其他的 FTP 客户端与 FTP 服务器交互，运行“cd”、“ls”、“get”等命令显示和下载文件。

题目 3：局域网聊天室

设计目的：

- 设计并实现一个局域网内的聊天小程序，至少能完成局域网内主机之间文字信息传送。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- **至少**实现一对一文字对话。
- **至少**实现一对多文字对话。

题目 4：HTTP/HTTPS 浏览器

设计目的：

- 设计并实现一个 HTTP/HTTPS 浏览器，能根据输入的 URL 访问网页并显示。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- **至少能实现文本显示，可尝试显示图片和多媒体流。**

题目 5：多线程 Web 服务器

设计目的：

- 设计并实现一个 WEB 服务器，可并行服务处理多个请求。

设计要求：

- 在后台运行服务器程序。

- 使用 Socket API 或 WinPcap 技术。
- 主线程监听客户机的连接建立请求，为每次请求/响应创建一个单独的 TCP 连接，一个单独的线程将处理这些连接。
- 用浏览器打开，显示请求页面内容即可；如有差错，则显示出错信息。

题目 6：路由跟踪小程序

设计目的：

- 设计并实现一个基于 IP 的路由跟踪小程序，根据输入的 IP 地址（或域名），输出本机到该地址（或域名）所属计算机的路径上经过的路由器，包括 IP 地址和 RTT 往返时间，类似于 tracert。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- 可考虑如果无法通过 ICMP 协议得到某路由器的 IP 地址时，有什么替代办法。

题目 7：子网内文件传送

设计目的：

- 设计并实现一个局域网内部的文件传送工具。采用 P2P 方式通信。可参考飞鸽传书软件。

设计要求：

- 以命令行或图形界面形式运行程序。
- 自行定义局域网内的可靠传输协议。
- 使用 Socket API 或 WinPcap 技术。
- 不同结点上文件自动同步。

题目 8： 多线程 WEB 代理服务器

设计目的：

- 设计并实现一个 WEB 代理服务器，记录管辖范围内的 GET 请求，缓存 web 页面，接收浏览器发来的 GET 报文，向目的服务器转发 GET 报文和响应报文。存储返回的响应报文，记录生存时间。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。

题目 9：简单邮件代理

设计目的：

- 设计并实现一个邮件收发器（类似于 outlook），使用 SMTP 协议和 POP3 协议（或 IMAP 协议）。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- 至少完成简单的文本发送和接收。
- 可考虑实现带附件的邮件收发。

题目 10：网页及链接下载

设计目的：

- 设计并实现一个下载程序，根据输入的网页 URL，将该网页和网页上链接的资源（包括链接的网页）都下载并存储。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- 只下载输入 URL 路径下及子路径下的相关资源。
- 可指定下载的目录层数，并保证资源不会重复下载。
- 下载后的资源间要保持相对路径，下载的网页文件中的超链接要修改为指向下载得到的本地资源。

题目 11：简单远程程序调用

设计目的：

- 设计并实现一个 C/S 应用，客户端接受用户从键盘输入的命令，并发送给服务器，服务器端执行该命令，并将命令的执行结果返回给客户端，客户端显示命令执行结果。类似于 Telnet 应用。

设计要求：

- 以命令行或图形界面形式运行程序。

- 使用 Socket API 或 WinPcap 技术。
- 如果服务器端无法识别该命令，给出错误提示。

题目 12：网络拓扑发现程序

设计目的：

- 设计并实现一个网络拓扑发现程序，通过 ICMP、UDP 或/和 SNMP 协议，发现网络节点和节点之间的互联关系，并输出。

设计要求：

- 以命令行或图形界面形式运行程序。

题目 13：实现一个可靠传输协议

设计目的：

- 设计并实现一个两个主机间通信的可靠传输协议（如停等协议、Go-Back-N 或 Selective Repeat），建立在 IP 或 UDP 基础之上。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- 自行定义可靠传输协议的协议数据单元（PDU）格式和发送/接收端的动作。

题目 14：统计协议流量与子网内各活动主机状态

设计目的：

- 设计并实现一个监听 TCP/IP 网络流量的小程序，统计该网卡子网内的不同协议流量。
- 记录局域网内活动主机数量和活动端口，记录该活动主机的 IP 地址和端口号。

设计要求：

- 以命令行或图形界面形式运行程序。

题目 15：UDP 网络吞吐量测试小程序

设计目的：

- 设计并实现一个 C/S 应用，客户端向服务器的指定端口发送 UDP 数据

包，服务器向客户端返回收到的数据量，由客户端计算吞吐量并显示。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 UDP 协议。
- 应用分为握手、发包测试，和统计结果三个阶段。
- 由客户端启动和停止发包。
- 由客户端指定发包大小和发包频率。

题目 16：隧道技术

设计目的：

- 设计并实现一个 IP 或 HTTP 隧道，客户端和服务器作为隧道的出入口，其他程序可以通过隧道程序向对端发送数据。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。

题目 17：域名查询程序

设计目的：

- 设计并实现一个类似 nslookup 的 DNS 客户端，可以根据用户输入的查询请求，向本地域名服务器或根域名服务器发出请求，并显示查询结果。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。

题目 18：本地域名服务器程序

设计目的：

- 设计并实现一个简单的本地域名服务器程序，实现基本的本地域名服务器功能，即响应客户请求，代替客户程序访问域名系统，向客户返回查

询结果。

设计要求：

- 以命令行或图形界面形式运行程序。
- 使用 Socket API 或 WinPcap 技术。
- 至少实现迭代查询。

题目 19：自拟题目

设计要求：使用基本的 Socket API 或 WinPcap（JPcap）函数开发一个网络应用程序，该应用程序能够实现 TCP/IP 协议族的协议或自行设计的网络协议（基于网络通信的应用协议）。总之，自行拟定的设计应是与网络底层开发相关的任务，不能是直接调用高层接口。

9.2.3 创新题

题目 1：新型网络架构 SDN 智能路由

设计目的：

- SDN 网络是新型网络架构，通过将网络设备的控制面与数据面分离开来，从而实现了网络流量的灵活控制，使网络作为管道变得更加智能，为核心网络及应用的创新提供了良好的平台；这样可以在控制面中设计不同的算法来控制数据包的转发，如采用深度学习、强化学习等方法，对数据包进行智能转发
- 设计并实现一个智能路由算法，能够结合网络 QoS 的一些指标对网络性能进行优化。

设计要求：

- 在控制面实现机器学习（或深度学习、强化学习）算法的智能路由算法，基于 Mininet 搭建仿真环境进行实验，控制器可采用 RYU 等控制器（不限）
- 基于胖树拓扑（基础要求）和更加复杂的拓扑（可选）进行测试，和传统的 OSPF 算法进行对比，性能有何提升？对结果进行可视化分析

题目 2：基于 SDN 的 DDoS 攻击检测和防御技术

设计目的：

- 熟悉 SDN 架构及 mininet 仿真环境搭建
- 模拟 DDoS 攻击并实现一种 DDoS 攻击的检测方法
- 利用 sFlow 验证 DDoS 攻击的防御功能

设计要求：

- 基本要求：基于 Mininet 搭建仿真环境进行实验，控制器可采用 RYU 等控制器（不限），部署 sFlow 工具，获取接口流量信息并解析数据，对解析后的数据进行分析判断，能够检测出 DDoS 攻击，并对比防御未开启和开启后的流量变化，要求有可视化结果
- 进阶要求： 对上述工作进行改进，可考虑优化的点有：利用机器学习等方法检测 DDoS 攻击（检测精度）、更加复杂的防御策略，实验模拟正常的背景流量和攻击流量对 DDoS 攻击检测和防御方法进行测试

题目 3：基于 SDN 的数据中心流量负载均衡技术研究

设计目的：

- 熟悉 SDN 架构及 mininet 仿真环境搭建
- 了解负载均衡的概念、作用和原理，掌握基于 SDN 的负载均衡实现方法

设计要求：

- 基本要求：基于 Mininet 搭建仿真环境进行实验，控制器可采用 RYU 等控制器（不限），对链路负载进行计算，然后规划最优路径（使用最短路径算法完成），基于胖树拓扑进行测试并验证方法的有效性。
- 进阶要求：对上述工作进行改进，可考虑优化的点有：预测链路负载状况、考虑为用户制定个性化 QoS 服务、自适应的进行负载均衡等（可考虑一个或多个算法的创新），最后基于胖树拓扑（基础）和更加复杂的拓扑进行测试，给出性能提升可视化结果。

创新题提示：可阅读相关论文并复现论文的工作，同时可基于 Github 上的源码进行二次开发，最终需要阐明自己的工作。

附录 A　Winpcap 结构体和常用函数

■ Winpcap 结构体

在程序中，将要进行分析的各数据包头格式用结构体进行定义。这样便于对数据包的解析，使每个字段清楚易懂。

■ 以太帧首部格式结构体

以太帧的首部共 14 个字节，结构体实现如下：

```
typedef struct ether_header
    {    unsigned char ether_dhost[6];        //目的 MAC 地址(48 bits)
         unsigned char ether_shost[6];        //源 MAC 地址(48 bits)
         unsigned short ether_type;           //协议类型(16 bits)
    };
```

■ IPv4 首部格式结构体

IPv4 数据报的首部共 20 个字节，结构体实现如下：

```
typedef struct ipv4_header
{    unsigned char ver_ihl;                   //版本(4 bits) + 首部长度(4 bits)
     unsigned char tos;                       //服务类型(8 bits)
     unsigned short tlen;                     //数据报总长度(16 bits)
     unsigned short identification;           //标识(16 bits)
     unsigned short flags_fo;                 /标志 (3 bits) + 片偏移 (13 bits)
     unsigned char ttl;                       //生存时间(8 bits)
     unsigned char proto;                     //协议(8 bits)
     unsigned short crc;                      //首部校验和(16 bits)
     u_char ip_src[4];                        //源 IP 地址(32 bits)
     u_char ip_dst[4];                        //目的 IP 地址(32 bits)
    };
```

■ IPv6 首部格式结构体

IPv6 的首部共 40 个字节，结构体实现如下：

```
typedef struct ipv6_header
{    u_char ver_tf;                           //版本号（4 bit）
```

```
    u_char traffic;                 //优先级（8 bit）
    u_short label;                  //流标识（20 bit）
    u_char length[2];               //报文长度（16 bit）
    u_char next_header;             //下一首部（8 bit）
    u_char limits;                  //跳数限制（8 bit）
    u_char Srcv6[16];               //源 IPv6 地址（128 bit）
    u_char Destv6[16];              //目的 IPv6 地址（128 bit）
};
```

- **TCP 首部格式结构体**

TCP 的首部共 20 个字节，结构体实现如下：

```
typedef  struct tcp_header
{   WORD   SourPort;          //源端口号(16 bits)
    WORD   DestPort;          //目的端口号(16 bits)
    DWORD      SeqNo;         //序号(32 bits)
    DWORD      AckNo;         //确认序号(32 bits)
    BYTE       HLen;          //首部长度（保留位）
    BYTE       Flag;          //标识（保留位）
    WORD   Window;            //窗口大小(16 bits)
    WORD   ChkSum;            //校验和(16 bits)
    WORD   UrgPtr;            //紧急指针(16 bits)
};
```

- **UDP 首部格式结构体**

UDP 的首部共 8 个字节，结构体实现如下：

```
typedef  struct udp_header
{   u_short sport;            //源端口号(16 bits)
    u_short dport;            //目的端口号(16 bits)
    u_short len;              //数据报长度(16 bits)
    u_short crc;              //校验和(16 bits)
};
```

- **WinPcap 常用函数**

LPPACKET PacketAllocatePacket(void)

VOID PacketInitPacket(LPPACKET lpPacket, PVOID Buffer, UINT Length)

VOID PacketFreePacket(LPPACKET lpPacket)

VOID PacketCloseAdapter(LPADAPTER lpAdapter)

BOOLEAN PacketGetAdapterNames(LPSTR pStr,PULONG BufferSize)

LPADAPTER PacketOpenAdapter(LPTSTR AdapterName)

BOOLEAN PacketReceivePacket(LPADAPTER AdapterObject,LPPACKET lpPacket, BOOLEAN Sync)

BOOLEAN PacketSetHwFilter(LPADAPTER AdapterObject,ULONG Filter)

BOOLEAN PacketGetNetInfoEx(LPTSTR AdapterNames,npf_ip_addr *buff, PLONG NEntries)

BOOLEAN PacketSendPacket(LPADAPTER AdapterObject,LPPACKET lpPacket, BOOLEAN Sync)

int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)

char *pcat_lookupdev(char *errbuf)

int pcap_lookupnet(char *device, bpf_u_int32 *netp,bpf_u_int32 *maskp, char *errbuf)

pcap_dumper_t *pcap_dump_open(pcap_t *p,char *filename)

Pcap_t * pcap_open_live(char * DeviceName,int snaplen,int promisc,int to_ms,char *errbuf)

int pcap_compile (pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)

int pcap_setfilter (pcap_t *p, struct bpf_program *fp)

int pcap_dispatch(pcap_t *p, int cnt,pcap_handler callback, u_char *user)

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

pcap_read()

u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)

void pcap_close (pcap_t *p)

int pcap_setbuff(pcap_t *p,int dim)

int pcap_setmode(pcap_t *p,int mode)

pcap_stats(pcap_t *p, struct pcap_stat *ps)

int pcap_sendpacket(pcap_t *p,char *buf,int size)

FILE *pcap_file(pcap_t *p)

int pcap_fileno(pcap_t *p)

附录 B　WinSock 结构体和常用函数

■ **WinSock 结构体**

```
struct sockaddr
  {    u_short     sa_family;                          //地址族
              char        sa_data[14];                 // 14 字节的直接地址
    };
struct sockaddr_in
    {    short       sin_family;
       u_short             sin_port;
       struct   in_addr   sin_addr;
           char            sin_zero[8];
     };
struct in_addr                                        //网络字序 IP 地址(32 bits)
    {   union {
            struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
            struct { u_short s_w1,s_w2; } S_un_w;
            u_long        S_addr;
           } S_un;
#define s_addr   S_un.S_addr
                                                      // most tcp & ip code
#define s_host   S_un.S_un_b.s_b2
                                                      // host on imp
#define s_net    S_un.S_un_b.s_b1
                                                      // network
#define s_imp     S_un.S_un_w.s_w2
                                                      // imp
#define s_impno S_un.S_un_b.s_b4
                                                      // imp
#define s_lh       S_un.S_un_b.s_b3
```

```
                                                    // logical host
};
struct    hostent
  {       char      * h_name;                       // 规范主机名字
          char      * * h_aliases;                  // 主机的别名列表
          short     h_addrtype;                     // 主机的地址类型
          short     h_length;                       // 地址长度
          char      * * h_addr_list;                // 地址列表
#define h_addr    h_addr_list[0]                    // 地址，和后面地址兼容
    };
```

- **WinSock 常用函数**

```
int WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);
int WSACleanup(void);
SOCKET accept (SOCKET s, struct sockaddr *addr, int *addrlen);
int bind (SOCKET s, const struct sockaddr *addr, int namelen);
int closesocket (SOCKET s);
int connect (SOCKET s, const struct sockaddr *name, int namelen);
int ioctlsocket (SOCKET s, long cmd, u_long *argp);
int getsockopt (SOCKET s, int level, int optname, char * optval, int *optlen);
u_long htonl (u_long hostlong);
u_short htons (u_short hostshort);
unsigned long inet_addr (const char * cp);
char * inet_ntoa (struct in_addr in);
int listen (SOCKET s, int backlog);
u_long ntohl (u_long netlong);
u_short ntohs (u_short netshort);
int recv (SOCKET s, char * buf, int len, int flags);
int recvfrom (SOCKET s, char * buf, int len, int flags, struct sockaddr *from, int * fromlen);
```

int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timeval *timeout);

int send (SOCKET s, const char * buf, int len, int flags);

int sendto (SOCKET s, const char * buf, int len, int flags, const struct sockaddr *to, int tolen);

int setsockopt (SOCKET s, int level, int optname, const char * optval, int optlen);

int shutdown (SOCKET s, int how);

SOCKET socket (int af, int type, int protocol);

struct hostent * gethostbyaddr(const char * addr, int len, int type);

struct hostent * gethostbyname(const char * name);

int gethostname (char * name, int namelen);

struct servent * getservbyport(int port, const char * proto);

struct servent * getservbyname(const char * name, const char * proto);

struct protoent * getprotobynumber(int proto);

struct protoent * getprotobyname(const char * name);