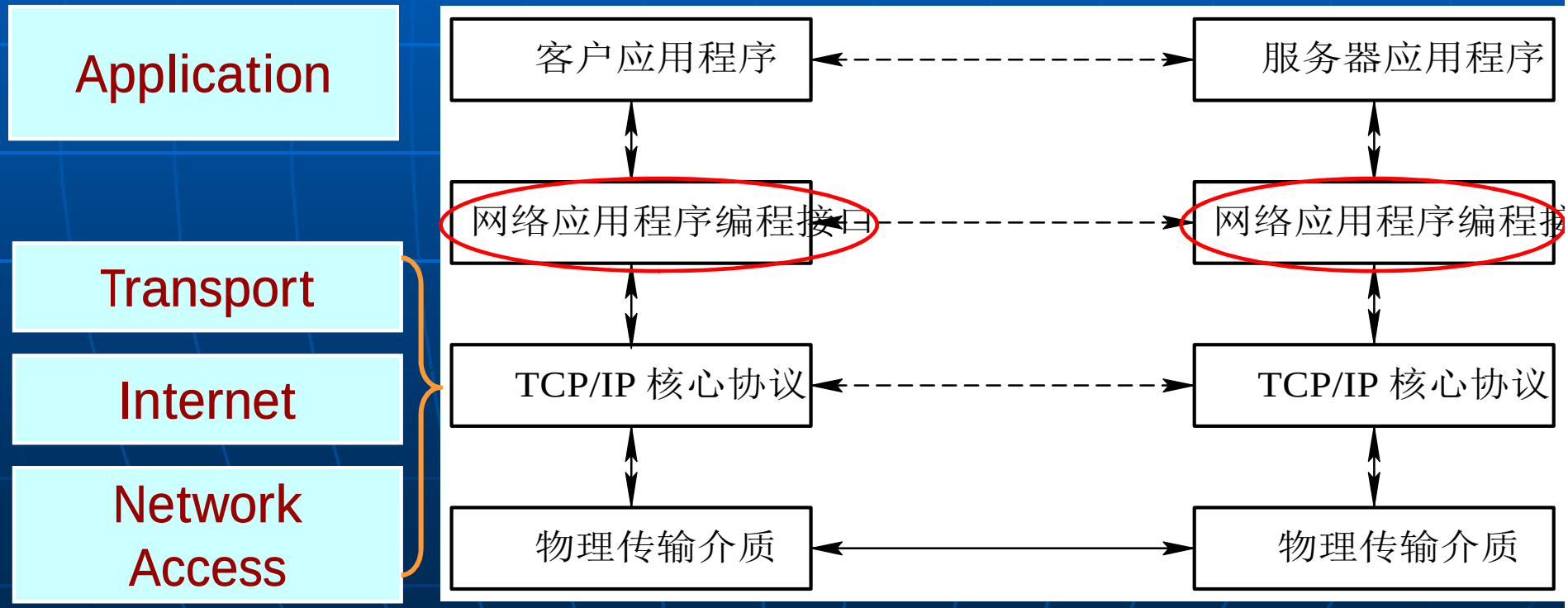
第12章 网络编程基础

- 网络编程概述
- 套接字
- 地址和地址操作函数
- 套接字函数
- 编程例子——面向连接的客户/服务器通信
  - 功能，原理，程序，说明

12.1 网络编程概述

- TCP/IP协议的核心内容在层次结构的低三层，即网络接口层、传输层和网际层，这三层的功能嵌入在操作系统的内核中
- TCP/IP应用程序使用TCP/IP核心协议，必须通过调用API (Application Programming Interface)才能通信



两台主机的进程通过网络编程接口进行通信的原理图

12.1 网络编程概述

- 用户进行网络编程时，首先要熟悉系统平台所提供的网络应用编程接口（API）

网络协议	编程接口
NetBIOS 协议	NetBIOS API
IPX/SPX 协议	IPX/SPX API
TCP/IP 协议	Socket API

12.2 套接字（Socket）

- Socket API 是于 1983 年在 Berkeley Socket Distribution (BSD) Unix 中引进的，常被称为伯克利套接口（Berkeley Sockets）

- Scoket类型：
  - Berkeley Sockets
  - Windows Sockets (Winsock)
  - Java Sockets
  - Python sockets
  - Perl sockets

本课程学习的目标

Winsock

- 在Windows 平台上的移植版本称为Winsock
  - 它不仅包含前者的大部分函数，还包含一组针对Windows 系统的扩展库函数（通常以字母WSA 打头）
  - 这些函数使编程人员能充分利用windows 的消息机制以及Win32 平台下的高性能I/O模型。
  - 不同的TCP/IP 协议栈供应商需要提供自己的Socket 接口实现的动态链接库，因此底层协议栈与Winsock 的接口在不同实现之间是完全不同的。

Winsock

- 最初在Windows 平台下的移植版本是winsock1.1
- 在Winsock1.1的基础上，微软又进一步提供了winsock2 接口。
- winsock2 的改进
  - 它在winsock 接口与协议栈之间定义了一种标准服务提供接口（SPI)，使得同一个winsock 动态链接库能同时访问多个由不同供应商提供的协议栈。
  - 协议栈供应商不再需要提供自己的winsock 接口实现链接库，微软提供了单独的ws2_32.dll ， 该链接库已经成为Windows 系统的一个基本组件。

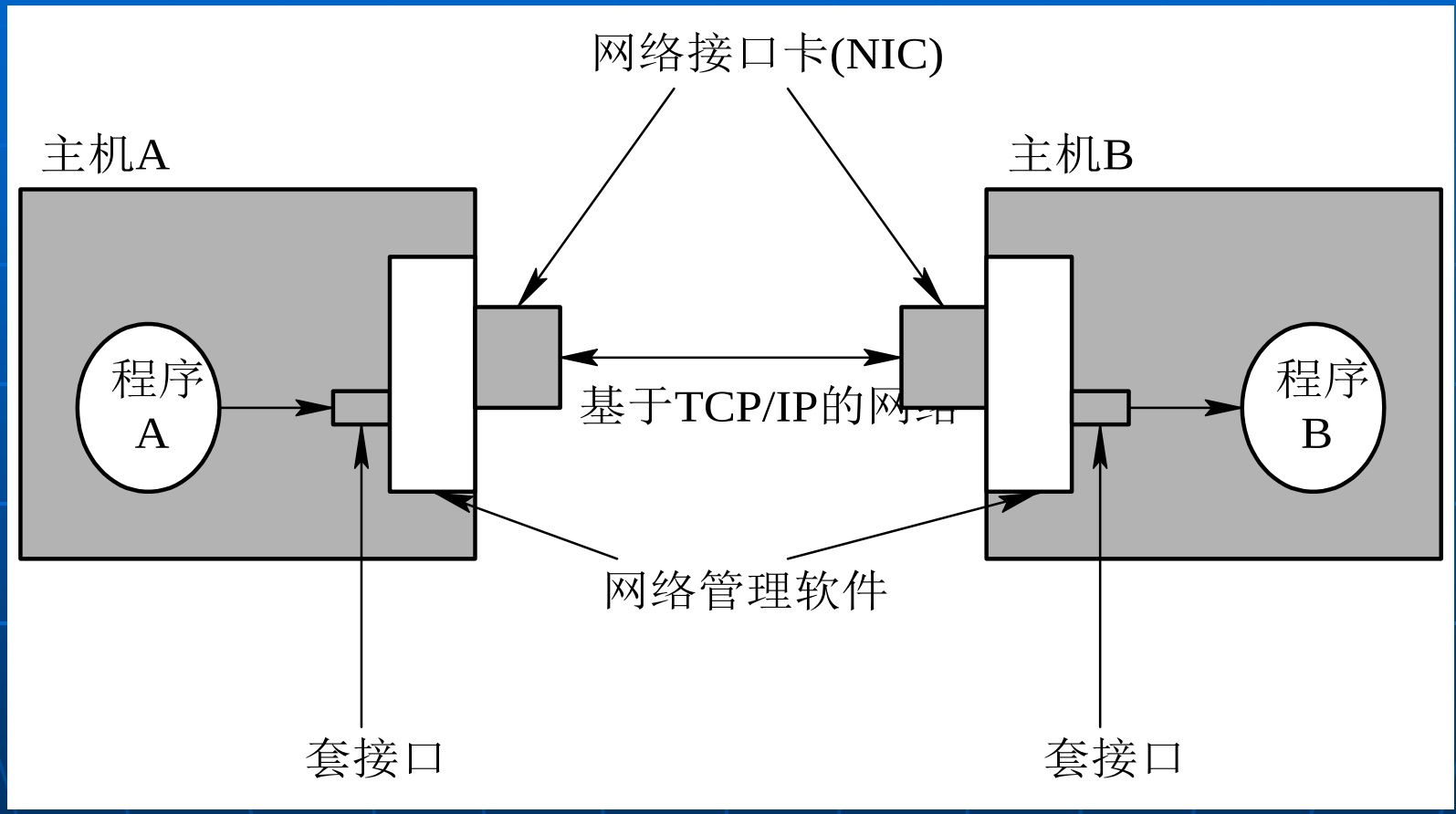
- 在Windows中，Socket是以 DLL 的形式实现的
- 在编写winsock 应用程序时，一般并不直接使用 ws2_32.dll，而是用其对应的静态链接库 ws2_32.lib
  - VC++菜单栏中选择Project→Settings → Link，再添加库函数 ws2_32.lib
  - 或在程序首部加入

    ```
    #pragma  comment ( lib , "ws2_32.lib" )
    ```

套接字（Socket）

- 套接口可以看成是两个网络应用程序进行通信时，各自通信连接中的一个端点。
- 套接口为用户提供了一种发送和接收数据的机制
  - 通信时，其中的一个网络应用程序将要传输的一段信息写入它所在主机的Socket中，该Socket通过网络接口卡(NIC)的传输介质将这段信息发送到另一台主机的Socket中，使这段信息能传送到其他程序中。

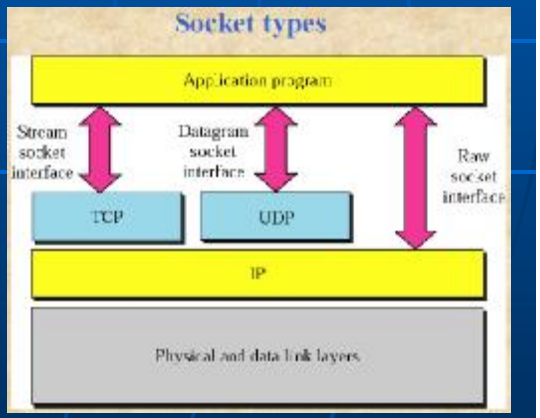
套接字（Socket）



套接口示意图

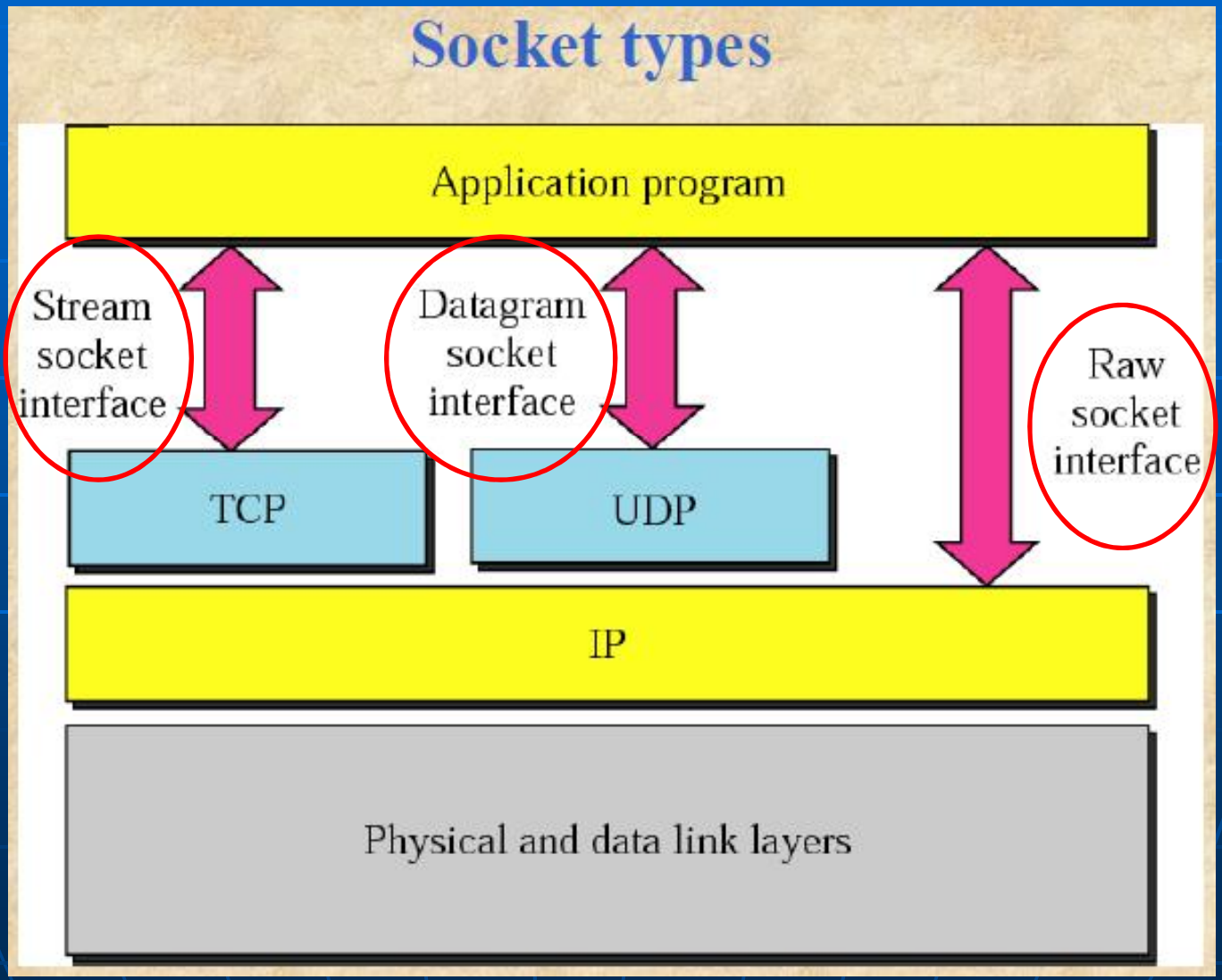
套接字的类型

- 为了满足不同的通信程序对通信质量和性能的要求，一般的网络系统提供了三种不同类型的套接口，以供用户在设计网络应用程序时根据不同的要求来选择
- 三种类型套接口：
  - 流式套接口 （SOCK_STREAM）
  - 数据报套接口（SOCK_DGRAM）
  - 原始套接口 （SOCK_RAW）



套接字的类型

应用程序接口



- **SOCK_STREAM：流式套接口，对应于TCP 协议，向应用程序提供可靠的面向连接的数据传输服务。也称面向连接的套接口、TCP 套接口等。**
- **SOCK_DGRAM：数据报套接口，对应于UDP 协议，向应用程序提供不可靠的、无连接的数据报通信方式。也称无连接套接口、面向消息套接口、UDP套接口等。**
- **SOCK_RAW：原始套接口，使得可以直接读写IP、ICMP 、IGMP 报文；从IP 头构造自己的IP报文；捕获所有经过本机的IP包。**

通信进程的标识问题

- 同一系统内应用程序之间的通信可通过系统分配的进程号唯一标识
- 不同系统间应用程序之间的通信需要标识主机（用IP地址）和进程（用端口号）
- 若要理解通信的数据，进程间的通信还需要遵循相同的协议
- 所以唯一标识网络通信中的一个进程：
  - （协议，IP地址，本地端口号）
- 唯一标识网络通信的两个进程（或一条连接）：
  - （协议，本地IP地址，本地端口号，远程IP地址，远程端口号）
- 对于应用层，可选择的传输层通信协议为 TCP 或 UDP

12.3 地址与地址操作函数

- 地址结构
  - sockaddr_in : INET 协议族地址结构
  - in_addr : IPv4地址结构
  - sockaddr : 通用地址结构
- 地址操作函数

12.3.1 INET 协议族地址结构 (sockaddr_in)

- 地址结构名中的最后两个字母“in”，是Internet的简写，说明该结构仅适用于采用TCP/IP协议的网络。

- 结构定义如下：

```
struct  sockaddr _ in {
        short              sin_family ;   // 地址族
        u_short            sin_port ;     // 端口号
        struct   in_addr   sin_addr ;     // IP地址
        char               sin_zero [ 8 ] ;
};
```

INET 协议族地址结构—— sockaddr_in

结构体各成员说明：

- **sin_family：** 地址族，一般填为AF_INET
- **sin_port：** 16 位的端口，网络字节顺序
- **sin_addr：** 32位的IPv4 地址，网络字节顺序
- **sin_zero：** 8 个字节的0 值填充，惟一的作用是使sockaddr_in结构大小与通用地址结构sockaddr 相同。

网络字节顺序：“大序在前”，即起始地址存放整数的高序号字节
主机字节顺序：“小序在前”，即起始地址存放整数的低序号字节

INET 协议族地址结构—— sockaddr_in

- 一般在给结构体赋值之前先将其全部初始化为零
- 下面的两个函数经常被用来完成清零工作
  - **VOID ZeroMemory (PVOID destination , SIZE_T lenqth) ;**
  - **void * memset (void * dest, int c, size _t count) ;**

由于前者仅适用于Win32 平台，推荐使用后者

如：

```
struct  sockaddr_in   to ;
memset( &to, 0, sizeof (to) );
```

地址与地址操作函数

- 地址结构
  - sockaddr_in：INET 协议族地址结构
  - in_addr　　：IPv4地址结构
  - sockaddr　　：通用地址结构

12.3.2 IPv4 地址结构——in_addr

- 用于存储32位IPv4 地址的数据结构，其定义如下：

```
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;
    } S_un;
    # define  s_net       S_un.S_un_b.s_b1
    # define  s_host      S_un.S_un_b.s_b2
    # define  s_lh        S_un.S_un_b.s_b3
    # define  s_impno     S_un.S_un_b.s_b4
    # define  s_imp       S_un.S_un_w.s_w2
    # define  s_addr      S_un.S_addr
};
```

IPv4 地址结构——in_addr

- 最常用的赋值接口是 S_un 和 s_addr
- S _ addr : 32 位的无符号整数，对应32 位IPv4 地址
  - 若要将地址202.119.9.199 赋给in_addr 结构，可以使用如下代码：

```
in_addr   addr ;
addr. S_un. S_addr = inet_addr("202.119.9.199") ;
```

或简写为：

```
in_addr   addr ;
addr. s_addr = inet_addr("202.119.9.199") ;
```

其中，函数 inet_addr 用于转换点字符串表示的IP 地址。

IPv4 地址结构——in_addr

- 假设主机上有多块以太网卡，每块网卡都配有IP地址，并且不关心应用程序具体使用哪个接口，那么在给addr.s_addr 赋值时可用常量：

  INADDR_ANY

  它在winsock2. h 中被定义为（u _ long ）0X00000000 ，即本地的任意以太网接口IP 地址

- 一般用于服务器，在准备侦听时设置

  代码如下：

```
in_addr   addr;
addr.s_addr =  INADDR_ANY ;
```

地址与地址操作函数

- 地址结构
  - sockaddr_in : INET 协议族地址结构
  - in_addr : IPv4地址结构
  - sockaddr : 通用地址结构

12.3.3 通用地址结构— sockaddr

定义如下：

```
struct sockaddr {
        u_short   sa_ family;   // address family /
        char sa_data[14] ; // up to 14 bytes of protocol address
};
```

- 在最初设计套接口函数接口时，面临着这样的选择：是专门开发一套为TCP/IP 协议所用的API，还是提供一种通用的编程接口以服务于多种网络协议。
- 两者之间的差别非常明显
  - 如果采用前者，那么提供的函数接口就会相对简单，
  - 对于后者，程序员在使用时必须提供足够的信息（参数）来告诉接口自己所采用的协议族。
- 在使用bind( )、accept( )、connect( )等函数时，需要转换为sockaddr 地址格式，使其适用于不同的网络协议环境。

12.3.4 地址操作函数

1． 函数 inet_addr( )

- 将包含点分格式的IPv4地址字符串转化为 in_addr 地址结构适用的32位整数。
- 定义如下：

unsigned long inet_addr (const char FAR * cp) ;

  - 参数：cp，点分IPv4 字符串。
  - 如果没有错误发生，函数返回32 位的地址信息。
  - 如果cp字符串包含的不是合法的IP 地址，那么函数返回 INADDR_NONE 。

示例：in_addr addr ;

```
addr.s_addr = inet_addr("202.119.9.10") ;
```

地址操作函数

2． 函数 inet_ntoa( )

- 将一个in_addr 地址值转化为标准的点分IP 地址字符串。
- 定义如下：

  char FAR *inet_ntoa (struct in_addr in) ;

  - in : IPv4 地址结构。
  - 返回值：如果没有错误发生，函数inet_ntoa 返回一指向包含点分IP 地址的静态存储区字符指针; 否则返回NULL 。
  - 说明：保存在该指针指向的存储区中的信息仅确保在下一次winsock 调用之前有效，因此应该及时加以复制。

示例代码:

```
WSAStartup(…);
……
SOCKET  sock = socket( AF_INET, SOCK_STREAM, 0 );
struct   sockaddr_in  to;
memset( &to, 0, sizeof (to) );
to.sin_addr.s_addr = inet_addr("202.119.9.10");
to.sin_family = AF_INET;
to.sin_port = htons(5555);
……
connect ( sock,  (struct sockaddr *) & to,  sizeof(to) );
……
WSACleanup( );
```

地址操作函数

3． gethostbyname( )

- 完成域名解析功能
- 函数定义如下：

struct hostent FAR * gethostbyname (const char FAR * name）；

- **name**：待解析的域名字符串。
- 返回值：
  - 如果没有错误发生，函数返回包含域名地址信息的**hostent**结构数据，在**hostent** 结构中有一个**h_addr_list** 域，它是所获取的 **IP** 地址列表。
  - 否则返回空指针，可以调用**WSAGetLastError** 函数来获得具体的错误码。

hostent结构体定义如下：

```
struct   hostent
{
    char  *h_name ;      //主机的规范名
    char  **h_aliases ;  //主机别名
    int     h_addrtype ;  //主机ip地址的类型
    int    h_length ;      //主机ip地址的长度
    char  **h_addr_list ;   //主机地址列表
};
```

例1：解析www.qq.com，并输出获得的IP 地址

```
WSADATA   wsaData ;
WSAStartup( WINSOCK_VERSION,  &wsaData );
hostent*   hostinfo = gethostbyname("www.qq.com");
struct in_addr   addr ;
if ( hostinfo  !=  NULL )
{
  for ( int  i = 0;  hostinfo -> h_addr_list[i] != NULL;  i++)
  {
    memset( &addr, 0, sizeof(addr) );
    memcpy( &addr.S_un.S_addr, hostinfo->h_addr_list[i],
hostinfo->h_length );
    printf("%s: %s\n", hostinfo->h_name, inet_ntoa(addr));
  }
 }
 WSACleanup(  );
 getchar(  );
```

输出结果如下：

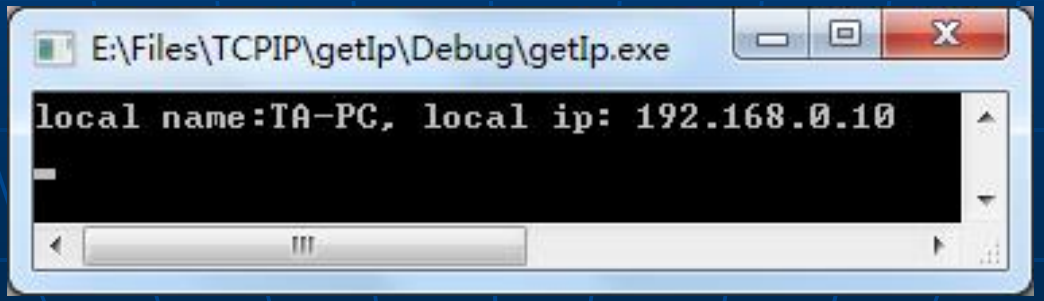
```
www.qq.com: 119.147.15.13
www.qq.com: 119.147.15.17
www.qq.com: 119.147.74.18
```

例2：获取本机的IP地址

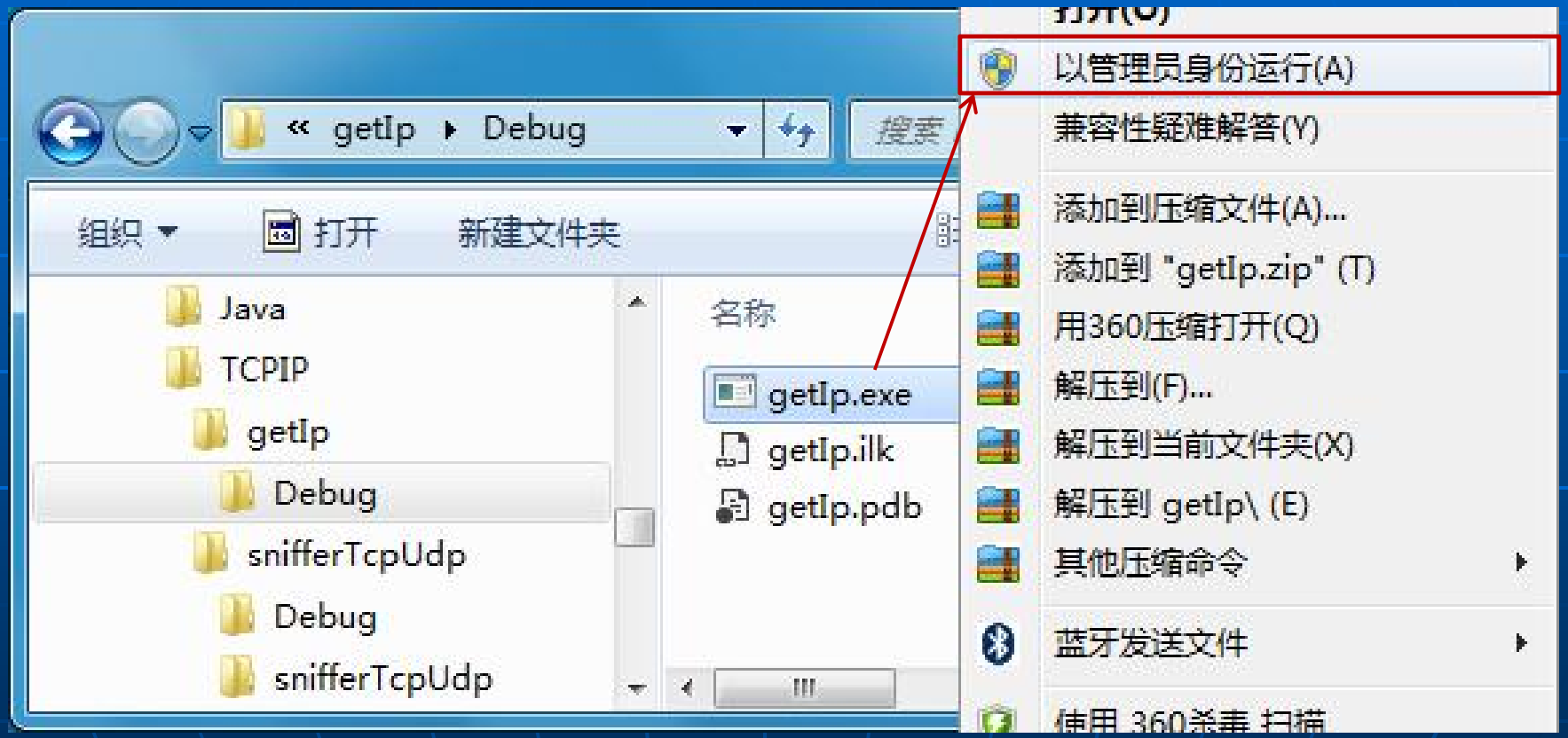
- 用代码获取本机IP地址，常用于软件自动化配置

```
#include <winsock2.h>
#include <stdio.h>
#define MAX_HOSTNAME_LEN 255  //最大主机名长度
#pragma comment (lib,"ws2_32.lib")//链接库

void main(int argc, char ** argv){
    char name[MAX_HOSTNAME_LEN];
    WSADATA wsaData;

    //初始化SOCKET
    WSAStartup(0x0202, &wsaData);

    gethostname(name, MAX_HOSTNAME_LEN); // 获取主机名

    struct hostent * pHostent;
    pHostent = (struct hostent *)malloc(sizeof(struct hostent));
    pHostent = gethostbyname(name); // 域名解析
```

```
SOCKADDR_IN sa;
sa.sin_family = AF_INET;  //使用互联网协议（IP）
sa.sin_port = htons(6000);  // 设置端口

if (pHostent != NULL && pHostent->h_addr_list[0] != NULL){
    // 设置IP地址
    memcpy(&sa.sin_addr.s_addr, pHostent->h_addr_list[0], pHostent->h_length);
    printf("local name:%s, local ip: %s\n", name, inet_ntoa(sa.sin_addr));
}
else
    printf("get null");

getchar();
```



- 生成的exe文件需以“管理员”的身份执行



12.4 套接字函数

- 套接字综合使用示例
- **WSAStartup( )**
- **socket( )**
- **bind( )**
- **listen( )**
- **accept( )**
- **connect( )**
- **recv( )、recvfrom( )**
- **send( )、sendto( )**
- **closesocket( )**
- **WSACleanup( )**

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

客户端与服务器TCP通信建立过程

TCP服务器示例程序

```
// ********** TCP服务器示例 **********

#include "winsock2.h"
#include "stdio.h"
#pragma comment (lib, "ws2_32.lib")  // 指定链接库

void main()
{
    // step 1: 初始化winsock
    WSAData wsa;
    WSAStartup(WINSOCK_VERSION,&wsa);

    // step 2: 创建winsock
    SOCKET ServerSocket;
    ServerSocket = socket(AF_INET,SOCK_STREAM,IPPROTO_IP);
    if(ServerSocket == INVALID_SOCKET)
    {
        printf("Creating Socket error!");
        return ;
    }
```

```
// step 3: 创建地址结构
struct sockaddr_in  ServerAddr;
memset( &ServerAddr , 0, sizeof(ServerAddr) );
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1"); // INADDR_ANY;
ServerAddr.sin_port = htons(5000);

// step 4: 对于TCP服务器，需要绑定套接口
if( bind(ServerSocket,(struct sockaddr *)&ServerAddr,sizeof(ServerAddr)) == SOCKET_ERROR )
{
    printf("Binding error!");
    closesocket(ServerSocket);
    WSACleanup();
    return ;
}

// step 5: 对于TCP服务器，绑定后就可以在本地端口侦听
if( listen(ServerSocket,5) == SOCKET_ERROR )
{
    printf("Listening error!");
    closesocket(ServerSocket);
    WSACleanup();
    return ;
}
else
{
    printf("Server start...\n");
}
```

```
// step 6: 对于TCP服务器，要随时准备接受远程连接请求
while( 1 )
{
    SOCKET  SocketClient ;
    struct sockaddr_in  from ;
    int len = sizeof(from) ;
    SocketClient =  accept(ServerSocket,(struct sockaddr*)&from , &len);  // 接受连接请求
    if( SocketClient != INVALID_SOCKET )
    {
        printf("远程主机: %s 通过端口: %d 连接到了服务器...\n", inet_ntoa(from.sin_addr) , ntohs(from.sin_port) );
        // 向远程主机发欢迎信息
        char buf[1024] , sendData[100] = "Welcome";
        len = strlen(sendData);
        sendData[len] = '\0' ;  // 字符串结束符号
        if( send(SocketClient,sendData,strlen(sendData)+1,0) != SOCKET_ERROR )
        {
            printf("   send welcome to client successful \n" );
        }
        // 接受远程主机发来的消息
        if( recv(SocketClient,buf,sizeof(buf),0) > 0 )
        {
            printf("   receive data from client: %s \n", buf );
        }
        if( closesocket(SocketClient) != SOCKET_ERROR )
        {
            printf("   close client Socket successful \n\n");
        }
    }
}// end of while(1)
}
```

TCP客户端程序

```
// ************** TCP客户端程序 **********************

#include "winsock2.h"
#include "stdio.h"
#pragma comment (lib, "ws2_32.lib")  // 指定链接库

void main()
{
    // step 1: 初始化winsock
    WSAData wsa;
    WSAStartup(WINSOCK_VERSION,&wsa);

    // step 2: 创建winsock
    SOCKET ClientSocket;
    ClientSocket = socket(AF_INET,SOCK_STREAM,IPPROTO_IP);
    if(ClientSocket == INVALID_SOCKET)
    {
        printf("Creating Socket error!");
        return ;
    }
```

```
// step 3: 创建准备连接到的服务器的地址结构
struct sockaddr_in  ServerAddr;
memset( &ServerAddr , 0, sizeof(ServerAddr) );
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
ServerAddr.sin_port = htons(5000);

// step 4: 连接到服务器
if(connect(ClientSocket,(struct sockaddr *)&ServerAddr,sizeof(ServerAddr))==SOCKET_ERROR)
{
    printf("connect error!");
    closesocket(ClientSocket);
    WSACleanup();
    return ;
}
else
{
    printf("connect to server: %s successfull \n", inet_ntoa(ServerAddr.sin_addr) );
}
```

```
// step 5: 接受服务器发来的信息
char buf[1024] = "\0" ;
if( recv(ClientSocket,buf,sizeof(buf),0) > 0 )
{
    printf("   receive data from server: %s \n", buf );
}
// step 6: 也可以向服务器发送数据
char data[100] = "Hello";
int len = strlen(data);
data[len] = '\0' ;  // 字符串结束符号
if( send(ClientSocket,data,strlen(data)+1,0) == SOCKET_ERROR )
{
    printf("   send hello to server error");
}
else
{
    printf("   send hello to server successful \n");
}
```

```
    // step 7: 关闭
    if( ClientSocket != INVALID_SOCKET )
    {
        closesocket(ClientSocket);
    }
    WSACleanup();

}
```

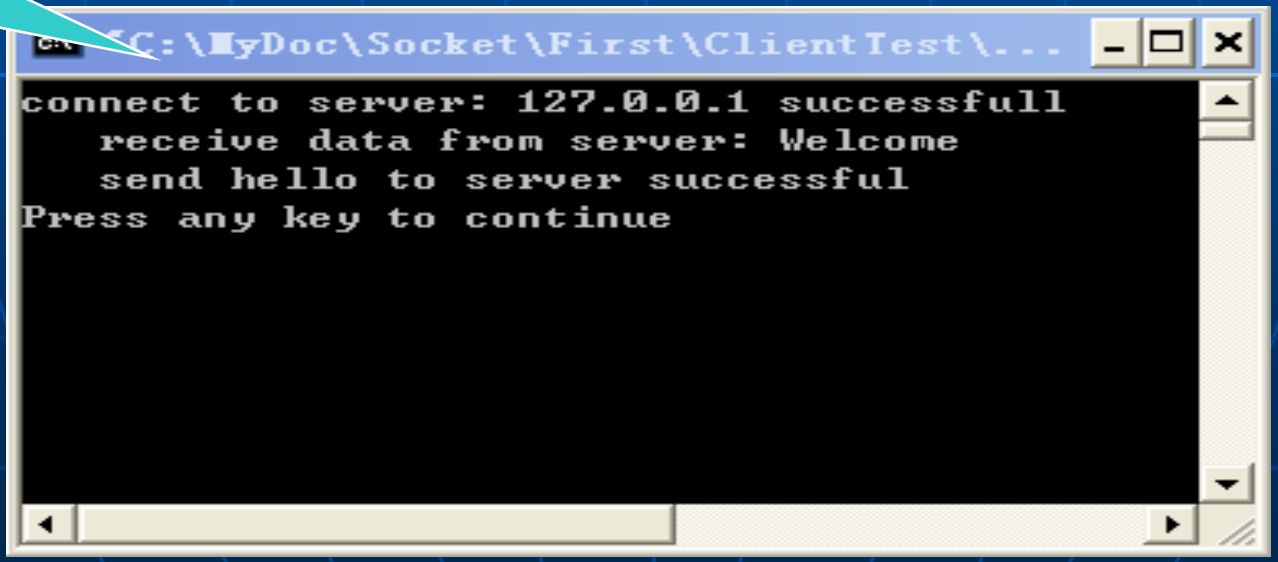
运行结果截图

服务器运行结果

第1次客户端连接

第2次客户端连接

客户端运行结果

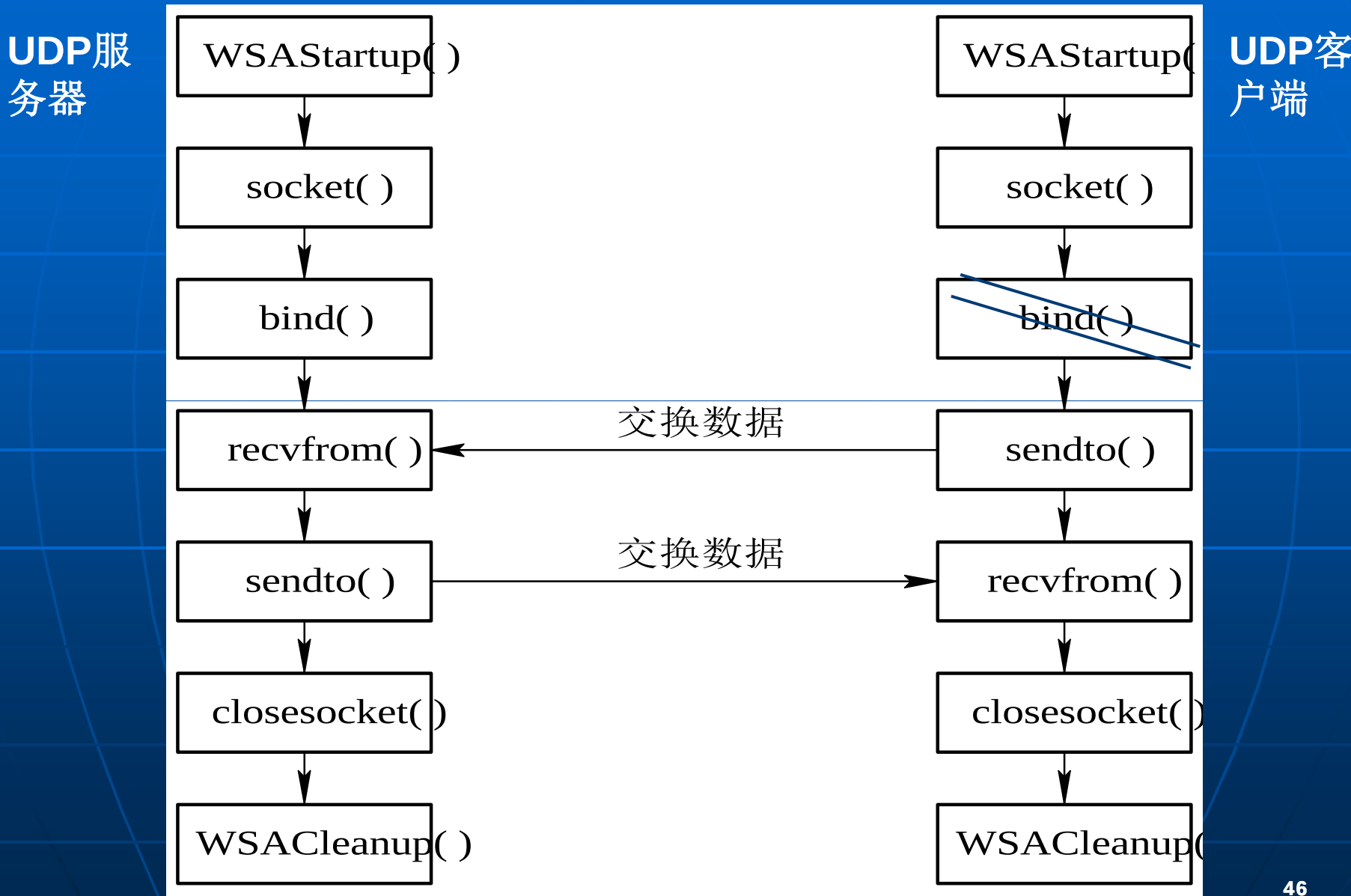


练习：服务器多线程

修改

```
// step 6: 对于TCP服务器，要随时准备接受远程连接请求
while( 1 )
{
    SOCKET  SocketClient ;
    struct sockaddr_in  from ;
    int len = sizeof(from) ;
    SocketClient =  accept(ServerSocket,(struct sockaddr*)&from , &len);  // 接受连接请求
    if( SocketClient != INVALID_SOCKET )
    {
        printf("远程主机: %s 通过端口: %d 连接到了服务器...\n", inet_ntoa(from.sin_addr) , ntohs(from.sin_port) );
        // 向远程主机发欢迎信息
        char buf[1024] , sendData[100] = "Welcome";
        len = strlen(sendData);
        sendData[len] = '\0' ;  // 字符串结束符号
        if( send(SocketClient,sendData,strlen(sendData)+1,0) != SOCKET_ERROR )
        {
            printf("   send welcome to client successful \n" );
        }
        // 接受远程主机发来的消息
        if( recv(SocketClient,buf,sizeof(buf),0) > 0 )
        {
            printf("   receive data from client: %s \n", buf );
        }
        if( closesocket(SocketClient) != SOCKET_ERROR )
        {
            printf("   close client Socket successful \n\n");
        }
    }
}// end of while(1)
}
```

无连接的UDP通信过程



无连接的UDP通信工作流程

(1)使用WSAStartup函数检查系统协议栈的安装情况。

(2)使用socket函数创建套接口，以确定相关五元组的协议。

(3) 使用bind函数将创建的套接口与本地地址绑定(服务器方)

(4)使用sendto函数发送数据，使用recvfrom函数接收数据。

(5)使用closesocket函数关闭套接口。

(6)调用WSACleanup函数，结束socket的使用。

至此，一次无连接的数据报传输过程结束。

无连接的UDP通信工作流程

无连接的客户/服务器编程注意事项：

（1）客户方可以不用bind函数绑定IP地址和端口

（2）若没有绑定地址，直接用sendto函数发数据时，系统会自动分配地址和端口

（3）在recvfrom函数接收数据前，套接字必须有指定的本地地址和端口

（4）服务器的一方必须先启动，否则客户请求传不到服务进程

（5）一般由不绑定IP地址和端口的一方首先向绑定地址的一方发送数据

（6）发送数据时，发送方除指定本地套接口的地址外，还需指定接收方套接口的地址，从而在数据收发过程中动态地建立全相关

UDP服务器源代码

```
/********************* 服务器端程序 ********************************************
调试环境：VC++6.0
程序名称：UDPserver.cpp
服务器IP地址：由系统指定
服务器接收端口：5050
服务器发送端口：由系统指定
功能：
1.从端口5050接收客户端发送来的数据，接收成功后显示收到的数据、客户端的IP地址和端口号；
2.向客户端发送"Hello! I am a server."字符串
*****************************************************************************/

#include<Winsock2.h>
#include<stdio.h>
#include<stdlib.h>
//指定链接库
#pragma comment (lib, "ws2_32.lib")
```

```
//服务器端口号为5050
#define DEFAULT_PORT  5050
//接收数据缓冲区长度
#define BUFFER_LENGTH  1024
void main( )
{
     int        iPort=DEFAULT_PORT;
     WSADATA    wsaData;
     SOCKET     sSocket;
     //客户地址长度
     int        iLen;
     //发送的数据长度
     int        iSend;
     //接收的数据长度
     int        iRecv;
     //要发送给客户的信息
     char       send_buf[ ]="Hello! I am a server.\0";
     //接收数据的缓冲区
     char       recv_buf[BUFFER_LENGTH];
     //客户端地址
     struct sockaddr_in  ser , cli;
     //初始化Socket
     if(WSAStartup(MAKEWORD(2,2),&wsaData)!=0)
     {
        printf("Failed to load Winsock.\n ");
        return;
     }
```

```
//产生服务器端套接口
sSocket = socket(AF_INET,SOCK_DGRAM,0);
if(sSocket == INVALID_SOCKET)
{
   printf("socket( ) Failed: %d\n", WSAGetLastError());
   return;
}
//建立服务器端地址
ser.sin_family=AF_INET;
ser.sin_port=htons(DEFAULT_PORT);
ser.sin_addr.s_addr=inet_addr("192.168.132.128");
//绑定地址
if( bind(sSocket, (const sockaddr*)&ser, sizeof(ser)) == SOCKET_ERROR )
{
    printf("bind() Failed: %d \n", WSAGetLastError());
    return;
}
//服务器开始运行
printf("  Server start ...  \n\n");
iLen=sizeof(cli);
```

```
//进入一个无限循环，等待接收客户端数据
while(1)
{
    //初始化接收缓冲区
    memset(recv_buf,'\0',sizeof(recv_buf));
    //从客户端接收数据
    iRecv=recvfrom(sSocket,recv_buf,BUFFER_LENGTH,0,(SOCKADDR*)&cli,&iLen);
    if(iRecv==SOCKET_ERROR)
    {
        printf("recvfrom( ) Failed:%d \n", WSAGetLastError( ));
        continue;
    }
    else    if(iRecv==0)
        continue;
    else
    {
        //输出接收到的数据
        printf("\n---------------------------------------\n");
        printf("  recvfrom( ): %s\n",recv_buf);
        //输出客户IP地址和端口号
        printf("  Accepted client IP:[%s],port:[%d]\n",
              inet_ntoa(cli.sin_addr), ntohs(cli.sin_port));
    }
```

```
    //给客户发送信息
    iSend=sendto(sSocket,send_buf,strlen(send_buf)+1,0,
        (SOCKADDR*)&cli,sizeof(cli));
    if(iSend==SOCKET_ERROR)
    {
        printf("  sendto( ) Failed.: %d\n",     WSAGetLastError( ));
    }
    else    if(iSend==0)
    {    }
    else
    {
        printf("  sendto( ) succeeded!\n");
    }
}

closesocket(sSocket);
WSACleanup( );
}
```

UDP客户端源代码

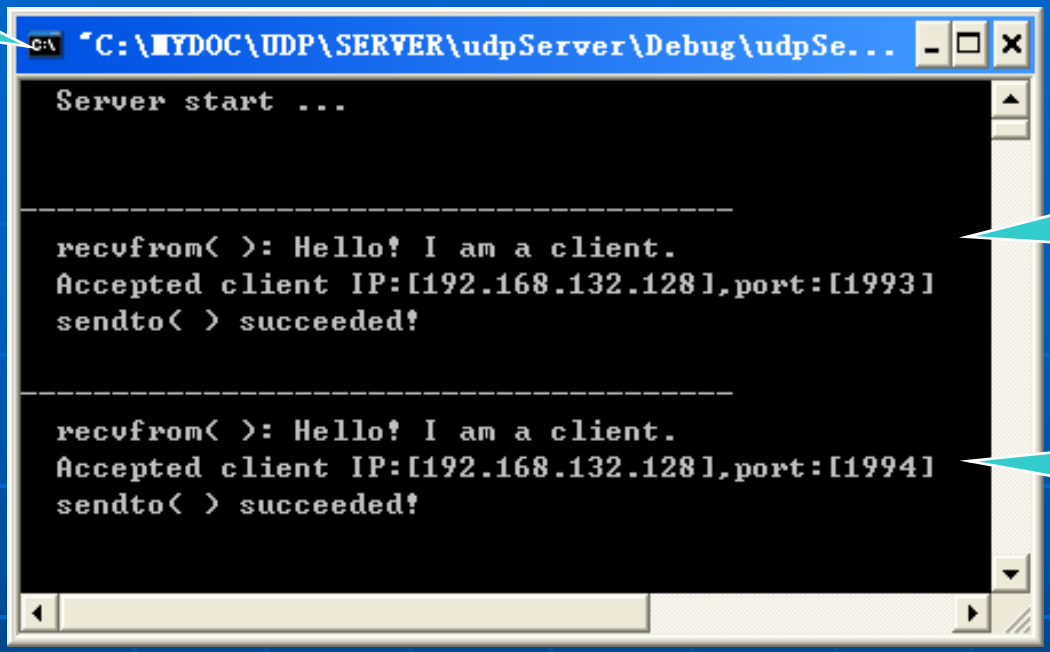
```
/********************* 客户端程序 ***********************************
调试环境：VC++6.0
程序名称：UDPclient.cpp
客户端IP地址和端口：由系统指定
程序功能：
1.客户端程序向服务器发送数据“Hello! I am a client.”;
2.客户端程序从服务器接收数据并进行显示。
*********************************************************************/

#include<Winsock2.h>
#include<stdio.h>
//指定链接库
#pragma comment (lib, "ws2_32.lib")
//服务器端口号为5050
#define Ser_PORT  5050
//缓冲区长度
#define DATA_BUFFER   1024
```

```
void main( )
{
	WSADATA wsaData;
	SOCKET sClient;
	//服务器地址长度
	int		iLen;
	//接收数据的缓冲
	int		iSend;
	int		iRecv;
	//要发送给服务器的信息
	char send_buf[ ]="Hello! I am a client.";
	//接收数据的缓冲区
	char recv_buf[DATA_BUFFER];
	//服务器端地址
	struct sockaddr_in ser;
	//初始化Socket
	if(WSAStartup(MAKEWORD(2,2),&wsaData)!=0)
	{
		printf("Failed to load Winsock.\n");
		return;
	}
```

```
//建立客户端数据报套接口
sClient=socket(AF_INET,SOCK_DGRAM,0);
if(sClient==INVALID_SOCKET)
{
   printf("socket( )Failed:%d\n",WSAGetLastError( ));
   return;
}
//建立服务器端地址
ser.sin_family=AF_INET;
ser.sin_port=htons(Ser_PORT);
ser.sin_addr.s_addr=inet_addr("192.168.132.128");
iLen=sizeof(ser);
//向服务器发送数据
iSend=sendto(sClient,send_buf,sizeof(send_buf),0,
            (struct sockaddr*)&ser,iLen);
if(iSend==SOCKET_ERROR)
{
   printf("sendto( )Failed:%d\n",WSAGetLastError( ));
   return;
}
else
   if(iSend==0)
       return;
else
   printf("sendto( ) succeeded.\n");
```

```
//接收数据的缓冲区初始化
memset(recv_buf,'\0',sizeof(recv_buf));
//从服务器接收数据
iRecv=recvfrom(sClient,recv_buf,sizeof(recv_buf),0, (struct sockaddr*)&ser,&iLen);
if(iRecv==SOCKET_ERROR)
{
   printf("recvfrom( )Failed.:%d\n", WSAGetLastError( ));
   return;
}
else   if (iRecv==0)
       return;
else
   {
       //显示从服务器收到的信息
       printf("recvfrom( ):%s\n",recv_buf);
       printf("------------------------------\n");
   }

closesocket(sClient);
WSACleanup( );
}
```

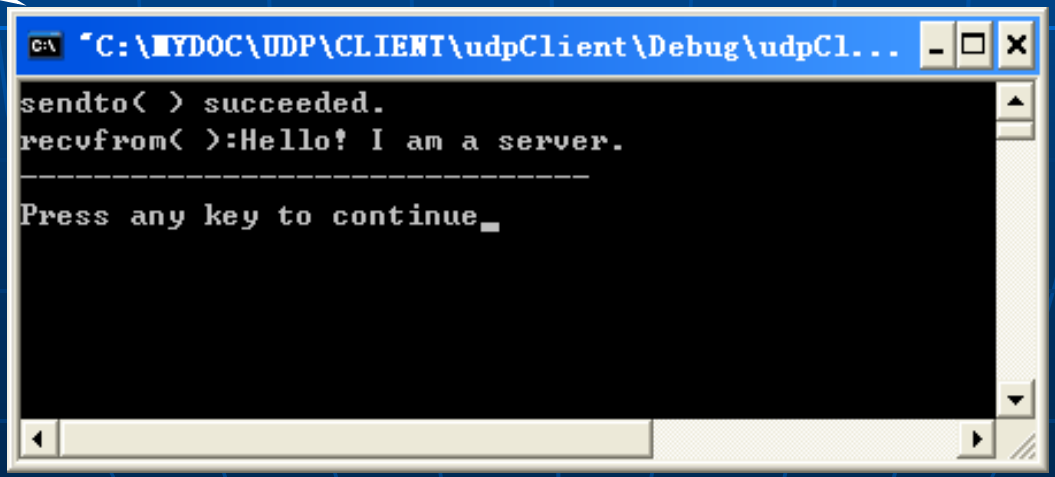
服务器运行结果



第1次客户端连接

第2次客户端连接

客户端运行结果



12.4 套接字函数

- 套接字综合使用示例
- **WSAStartup( )**
- **socket( )**
- **bind( )**
- **listen( )**
- **accept( )**
- **connect( )**
- **recv( )、recvfrom( )**
- **send( )、sendto( )**
- **closesocket( )**
- **WSACleanup( )**

WSAStartup函数

1. 函数格式：

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

2. 说明：

应用程序或DLL只能在一次成功的WSAStartup( )调用之后才能进一步调用其他的Windows Sockets API函数。用WSAStartup初始化ws2_32.dll，并且协商版本号。

WSAStartup

3. 函数参数说明

- **wVersionRequested**：此参数是一个WORD型(双字节型)数值，它指定准备在应用程序中要使用的Winsock库的版本号。
  - 用高位字节指定副版本，用低位字节指定主版本。
  - 如果需要加载Winsock 2.2版，指定此参数的值为0x0202；也可使用宏MAKEWORD(X,Y)，其中X为高位字节，Y为低位字节，如MAKEWORD(2,2)。
- **lpWSAData**：此参数是一个指向WSADATA结构的指针。当该函数被调用时，lpWSAData返回关于Windows Sockets实现的详细信息

示例

WSAStartup( )函数在程序中的基本使用方法：

```
#include  <Winsock2.h>
...//其他代码
WORD wVersionRequested;
WSADATA wsaData;
wVersionRequested = MAKEWORD(2 , 2);

//初始化SOCKET
WSAStartup(wVersionRequested,&wasData);

或直接:
WSAStartup(0x0202, &wsaData);
```

WSAStartup

4. 函数返回信息

- WSAStartup( )函数的返回值是一个整数，如果调用成功则返回0。
- 应用程序或dll请求的版本号低于系统的最低版本，则函数调用失败。
- WSAStartup( )函数调用不成功时返回如下的错误信息：
  - WSASYSNOTREADY：在Winsock的头文件Winsock2.h中，该错误代码定义的数值为10091，它表明加载的Winsock DLL不存在或底层的网络子系统无法使用。
  - WSAVERNOTSUPPORTED：该代码的数值为10092，所需的Windows Sockets API的版本未由特定的Windows Sockets实现提供。
    - 如果由wVersion返回的版本用户不能接受,则要调用WSACleanup( )函数清除对Winsock的加载。

```
if( WSAStartup(wVersionRequested, &wasData) != 0 )
{
    //Winsock初始化错误
    printf("WSAStartup failed");
    return;
}
//版本号匹配的检查
if( wsaData.wVersion != wVersionRequested )
{
        //Winsock版本号不匹配
        printf("version error");
        WSACleanup( );
        return;
    }
 //...
//说明Winsock DLL的加载正确，可以执行其他代码
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

创建socket

- 应用程序在使用套接口通信前，必须要拥有一个套接口。
- 使用socket( )或WSASocket( )函数来给应用程序创建一个套接口。

1. 函数格式

- Winsock 1中提供的创建套接口函数socket( )

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

创建socket

•在Winsock 2中提供的该函数的扩展格式如下：

```
SOCKET WSASocket(
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    Group g,
    int iFlags
);
```

创建socket

2. 函数参数说明

以上两种格式中，前面三个参数的含义是一样的，说明如下：

- **af：** 该参数说明套接口要使用的协议地址族。如果想建立一个UDP或TCP套接口，只能用常量AF_INET表示使用互联网协议(IP)地址。当然，Winsock 2还支持其他的协议，但一般在程序中很少使用。
- **type：** 该参数描述套接口的协议类型。当第一个参数af是AF_INET时，它只能使用SOCK_STREAM、SOCK_DGRAM或SOCK_RAW三个协议类型中的任一个，分别表示要创建的是流式套接口、数据报套接口或原始套接口。

创建socket

- **protocol：** 该参数说明该套接口使用的特定协议。当协议地址族af和协议类型type已经确定后，协议字段可以使用的值是限定的，如表所示。
  - 如果调用者不希望特别指定所使用的协议，可以将此参数设置为0，系统就根据前两个参数的值自动确定一个协议字段的取值。

协议	地址族	套接口类型	套接口类型使用的值	协议字段
互联网协议(IP)	AF_INET	TCP	SOCK_STREAM	IPPROTO_TCP
		UDP	SOCK_DGRAM	IPPROTO_UDP
		Raw sockets	SOCK_RAW	IPPROTO_IP IPPROTO_ICMP

套接口参数表

创建socket

3. 函数返回信息

- 该函数调用成功后，返回新创建的套接字描述符，它被定义成是一个无符号的整型数据。
- 函数调用错误时返回INVALID_SOCKET，应用程序可进一步调用WSAGetLastError( )函数来获取相应的错误代码。可能获得的错误代码说明如下：
  - WSANOTINITIALISED：在调用本API之前应成功调用WSAStartup( );
  - WSAENETDOWN：网络子系统失效；
  - WSAEAFNOSUPPORT：不支持指定的地址族；
  - ……

创建socket

4. 函数使用说明

- 要创建一个流套接口，可使用下列三种格式之一：
  - **SOCKET sockid=**
    socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_STREAM, 0);
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);

创建socket

- 要创建一个数据报套接口，可使用格式如下：
  - **SOCKET sockid=**
    socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_DGRAM, 0);
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_DGRAM, 0, NULL , 0, WSA_FLAG_OVERLAPPED);

创建socket

- 要创建一个原始套接口，可使用格式如下：
  - **SOCKET sockid=**
    socket(AF_INET, SOCK_RAW, IPPROTO_IP) ;
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_RAW, 0) ;
  - **SOCKET sockid=**
    WSASocket(AF_INET, SOCK_RAW, 0, NULL , 0,
    WSA_FLAG_OVERLAPPED) ;

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

bind函数

格式：**int bind(**

SOCKET s ,

const struct sockaddr FAR *name ,

int namelen

);

功能：将套接字（s）与地址（name）相绑定

说明：服务器方必须绑定；客户方可以不绑定

参数：

- **s**：待绑定地址的套接字
- **name**：赋予套接字的地址
- **namelen**：地址结构的长度

地址结构

- 第二个参数的类型是通用地址结构：

```
struct  sockaddr {
        u_short   sa_ family;   // 地址族
        char sa_data[14] ;      // 协议地址
};
```

- 在使用时常常由INET地址结构转换：

```
struct  sockaddr _ in {
        short               sin_family ;   // 地址族
        u_short             sin_prot ;     // 端口号
        struct   in_addr    sin_addr ;     // IP地址
        char                sin_zero [ 8 ] ;
} ;
```

示例

```
// 创建地址结构
struct  sockaddr_in  sAddr;
memset( &sAddr , 0, sizeof(sAddr) );
sAddr.sin_family = AF_INET;
sAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
sAddr.sin_port = htons(5000);

// 绑定套接口
if( bind( s , (struct sockaddr *)&sAddr, sizeof(sAddr) )
    == SOCKET_ERROR )
      {   printf("Binding  error!  \n");   }
else
      {   printf("Binding  ok!  \n");    }
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

listen函数

格式：

int listen(SOCKET s, int backlog);

功能：使套接字s处于侦听状态

说明：

1）使用前提：必须先bind

2）处于侦听状态的套接字 s 将维护一个客户连接请求队列，该队列最多容纳backlog个客户连接请求。

返回值：

1）成功：返回0；

2）失败：返回SOCKET_ERROR

示例

```
// 对于TCP服务器，绑定后就可以在本地端口侦听
if( listen(ServerSocket,  5) == SOCKET_ERROR )
{
        printf(“Listening error!  \n");
}
else
{
        printf("Server start...  \n");
}
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

accept函数

格式：

```
SOCKET accept(
    SOCKET  s ,
    struct  sockaddr  FAR  *addr ,
    int  FAR  *addrlen
);
```

功能：从处于侦听状态的流套接字s的客户连接请求队列中，取出排在最前的一个客户请求，并且创建一个新的套接字来与客户套接字通信

说明：

1）用于TCP协议，UDP协议不需要accept

2）之前必须完成地址绑定与侦听，使流套接字 s 处于侦听状态

3）如果队列中没有连接请求，则阻塞调用(默认)直至请求到来

返回值：

1）成功：返回新创建的套接字

2）失败：返回INVALID_SOCKET

参数说明

- **SOCKET s：处于侦听状态的套接口（服务器方）**
- **struct sockaddr FAR *addr：一个指针，指向装有客户方源地址的缓冲区。对侦听套接字的具体协议来说，addr参数是一个指向SOCKADDR结构的指针。函数调用返回时，SOCKADDR结构内便填入客户方主机的地址；**
- **int FAR *addrlen：客户方套接口地址结构的长度值**

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

connect函数

格式：

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

功能：使套接字 s（本地）与侦听于地址name的远程计算机（服务器）的特定端口上的Socket进行连接

说明：

1）一般用于TCP协议的客户方

2）如果套接字s没有绑定本地IP地址和端口，系统会自动分配

3）将激发TCP的三路握手过程，仅在连接成功或出错时才返回

返回值：

1）成功：返回0

2）失败：返回SOCKET_ERROR

示例

```
// 连接到服务器
if( connect( ClientSocket , (struct sockaddr *)&ServerAddr ,
    sizeof(ServerAddr) ) == SOCKET_ERROR )
{
    printf("connect error!  \n");
}
else
{
    printf("connect to server: %s successfull \n" ,
            inet_ntoa(ServerAddr.sin_addr) );
}
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

接收数据与发送数据

- 面向连接的通信
  - recv ( )函数：接受数据
  - send( )函数：发送数据
- 面向无连接的通信
  - recvfrom( )函数：接受数据
  - sendto( )函数：发送数据

recv函数与recvfrom函数

1. 格式：

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

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

2. 功能：从套接口 s 处接收数据

3. 说明：recvfrom函数若没有最后两个参数，则等同于recv函数

4. 默认为阻塞调用（可由函数ioctlsocket设置阻塞方式）

接收数据recvfrom

4. 参数说明:

- s：标识一个套接口的描述字；
- buf：接收数据缓冲区；
- len：缓冲区长度；
- flags：调用操作方式，可以是以下值之一：
  - 0: 表示接收的是正常数据，无特殊行为
  - MSG_PEEK: 表示有用的数据复制到所提供的接收端缓冲区内，但是没有从系统缓冲区中将数据删除
  - MSG_OOB: 表示处理带外数据

接收数据recvfrom

- **from：(可选)指针，指向装有源地址的缓冲区，对侦听套接字的具体协议来说，from参数是一个指向SOCKADDR结构的指针。函数调用返回时，SOCKADDR结构内便填入发送数据的主机的地址；**
- **fromlen：(可选)指针，指向from缓冲区长度值**

n **5．改变阻塞调用方式:**

- **u_long iMode = 1; // 1:控制为非阻塞方式; 0:控制为阻塞方式**
- **ioctlsocket(sock, FIONBIO, &iMode); // 将套接字sock设置为非阻塞方式**

接收数据recvfrom

5. 函数返回值

- 成功：返回所接收的字节数
- 连接已终止：返回0（对于TCP连接）
- 失败：返回SOCKET_ERROR错误

n失败时可通过WSAGetLastError( )获取相应错误代码：

•WSANOTINITIALISED：在调用本API之前应成功调用WSAStartup( )；

•WSAENETDOWN：网络子系统失效；

•WSAEFAULT：lpFromlen参数非法，lpFrom缓冲区太小，无法容纳远端地址(对recvfrom( )来说，fromlen参数非法，from缓冲区大小无法装入端地址)；

……

示例

```
sockaddr_in   ClientAddr ;  // 远程客户端地址变量
memset( &ClientAddr , 0 , sizeof(ClientAddr) ); // 初始化为空
n = sizeof( ClientAddr );
memset( &buf , '\0', sizeof( buf ) ) ;

// 接收数据，把内容填入buf中，把地址填入ClientAddr 中
n = recvfrom ( ServerSock , buf , sizeof( buf ) , 0 ,
          ( struct sockaddr *)&ClientAddr ,  &n ) ;
if( n != SOCKET_ERROR )
{
    printf("接收远程主机 %s 的数据：  %s \n",
                inet_ntoa(ClientAddr.sin_addr) ,  buf ) ;
}
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

send函数和sendto函数

1. 函数格式：

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

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

2. 功能：从套接口 s 处发送数据

3. 说明：sendto函数若没有最后两个参数，则等同于send函数

发送数据sendto

4. 函数参数说明:

- s：一个标识套接口的描述字；
- buf：待发送数据的缓冲区；
- len：指明buf缓冲区中要发送的数据长度；
- flags：用于控制数据传输方式：
  - 0：表示按正常方式发送数据；
  - MSG_DONTROUTE：说明系统目标主机就在直接连接的本地网络中，无需路由选择；
  - MSG_OOB：指出数据是按带外数据发送的。
- to：(可选)指针，指向接收数据的目的套接口的地址；
- tolen：to所指地址的长度。

5. 函数返回值

- 成功：返回所发送的字节数
- 连接结束：返回0（对于TCP连接）
- 失败：返回SOCKET_ERROR错误

- 当失败时可通过WSAGetLastError( )获取相应错误代码：
  - WSANOTINITIALISED：在调用本API之前应成功调用WSAStartup( )；
  - WSAENETDOWN：网络子系统失效；
  - WSAEACCES：请求的地址为广播地址，但未设置相应的标志位；

  ……

发送数据sendto（续）

6. 函数使用说明

对于数据报套接口，必须注意发送数据长度不应超过通信子网的IP包最大长度。

- IP包最大长度在WSAStartup( )调用返回的WSAData结构量的iMaxUdpDg元素中。如果数据太长无法自动通过下层协议，则返回WSAEMSGSIZE错误，数据不会被发送。
- 成功地完成sendto( )调用并不意味着数据已可靠传送到目标。

示例

```
// 设置服务器地址
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = inet_addr("192.168.1.114");
ServerAddr.sin_port = htons(SEND_PORT);
// 设置传送的数据
memset( &buf , '\0' , sizeof( buf ) ) ;
strcpy(buf , "This is a client\0" ) ;
// 开始传送
n = sendto( sock , buf , strlen( buf ) +1 , 0 ,
     (struct sockaddr *)&ServerAddr ,  sizeof(ServerAddr) ) ;
if( n == SOCKET_ERROR)
    {  printf("  send false! \n") ;   }
else if (n == 0)
    {    }
else
    {  printf("  send ok! \n") ; }
```

12.4 套接字函数

- 套接字综合使用示例
- WSAStartup( )
- socket( )
- bind( )
- listen( )
- accept( )
- connect( )
- recv( )、recvfrom( )
- send( )、sendto( )
- closesocket( )
- WSACleanup( )

关闭套接字closesocket

1. 函数格式：

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

2. 功能：关闭套接口，释放与该套接口关联的所有资源，包括等候处理的数据

3. 函数参数说明

closesocket( )函数中的参数s标识一个将被关闭的套接口

关闭套接字closesocket

4. 函数返回信息

如果没有错误发生，**closesocket( )**返回**0**；否则返回**SOCKET_ERROR**错误.

12.4 套接字函数

- 套接字综合使用示例
- **WSAStartup( )**
- **socket( )**
- **bind( )**
- **listen( )**
- **accept( )**
- **connect( )**
- **recv( )、recvfrom( )**
- **send( )、sendto( )**
- **closesocket( )**
- **WSACleanup( )**

WSACleanup

- 当在应用程序中不再使用Winsock API中的任何函数时，必须调用WSACleanup( )将其从Windows Sockets的实现中注销，以释放为应用程序或DLL分配的任何资源。
- 对应于一个任务进行的每一次WSAStartup( )调用，必须有一个WSACleanup( )调用，因为每次WSAStartup( )函数的启动调用都会增加对加载Winsock DLL的引用次数，它要求调用同样多次的WSACleanup( )调用，以此抵消引用次数。

WSACleanup

1. 函数格式

   int WSACleanup (void);

2. 函数参数说明

   该函数是一个无参函数。

3. 函数返回信息

   该函数调用成功返回0。调用失败时，返回的错误信息有下面几类：

   - WSANOTINITIALISED：使用本API前必须要进行一次成功的WSAStartup( )调用；
   - WSAENETDOWN：Windows Sockets的实现已经检测到网络子系统故障；
   - WSAEINPROGRESS：一个阻塞的Windows Sockets操作正在进行。

WSACleanup

4. 函数使用说明

- WSACleanup( )函数是任何一个Winsock应用程序在最后必须要调用的函数。
- 在一个多线程的环境下，WSACleanup( )函数中止了Windows Sockets在所有线程上的操作。
- 只有在一次成功调用WSAStartup( )函数后，才能调用WSACleanup( )函数，否则，WSACleanup( )函数返回WSANOTINITIALISED错误信息。

谢谢！