

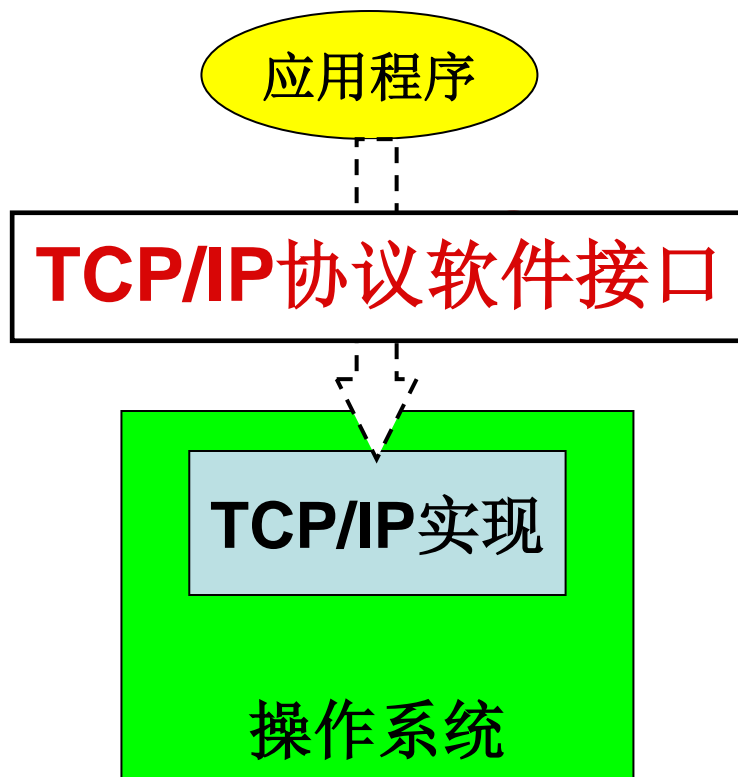
# 网络程序设计

协议软件接口

## 4.1 协议软件接口

- 协议软件接口承当应用程序与操作系统协议实现之间的桥梁作用。
- 它封装了协议实现的基本功能，开放系统调用接口以简化操作，使得应用程序可以用操作系统调用的方式方便地使用协议实现提供的数据传输功能。

## 4.1.1 协议软件接口的位置



## 4.1.2 协议软件接口的功能

- 现象：TCP/IP标准没有规定应用软件与TCP/IP协议软件如何接口的细节，只建议了所需的功能集。
- 特点：不精确指明
  - TCP/IP的设计者尽量避免使用任何一家厂商的内部数据表示。
  - TCP/IP标准尽量避免让接口使用那些只在某一家厂商的操作系统中可用的特征
- 目的：使得协议的兼容性强，可以运行在多厂商的环境之中。

## 4.1.2 协议软件接口的功能

- 应用软件与TCP/IP协议软件的接口必须支持如下概念性操作：
  - 分配用于通信的本地资源
  - 指定本地和远程通信端点
  - (客户端)启动连接
  - (服务器端)等待连接到来
  - 发送或接收数据
  - 判断数据何时到达
  - 产生紧急数据
  - 处理到来的紧急数据
  - 从容终止连接
  - 处理来自远程端点的连接终止
  - 异常终止通信
  - 处理错误条件或连接异常终止
  - 连接结束后释放本地资源

## 4.1.2 协议软件接口的功能

- 目前的常用的协议软件接口
  - **Berkeley UNIX: 套接字接口（或socket）**
  - **微软: Windows Socket**
  - **AT&T UNIX(System V):TLI(Transport Layer Interface)**

## 4.1.2 协议软件接口的功能

- 协议软件接口的功能是为网络应用程序和操作系统协议栈建立调用的关联。如创建用于关联的标识，为网络操作分配资源、拷贝数据、读取信息等。
- 真实的网络通信功能由协议具体完成
- 程序逻辑由网络应用程序部署。

## 4.1.2 协议软件接口的功能

- 数据发送是由发送接口完成还是由系统协议栈完成？
- 发送接口函数仅完成两个工作：
  - 第一，将数据从应用程序缓冲区拷贝到内核缓冲区；
  - 第二，向系统内核通知应用程序有新的数据要发送。
- 真正负责发送数据的是协议栈
- 发送接口函数成功返回并非意味着数据已经发送出去。此时数据有可能还保留在协议栈中等待发送，也有可能已经被发送到网络中。



# 4.2.1如何访问TCP/IP协议？

- 两种基本方法：
  - 设计者发明一种新的系统调用，应用程序用它们来访问**TCP/IP**；
    - 列举出所有的概念性操作
    - 为每个操作指定一个名字和参数
    - 将每个操作实现为一个系统调用
  - 设计者使用一般的I/O调用访问**TCP/IP**
    - 扩充一般的I/O原语
- 两种方法综合

# 问题的引入

## Linux系统的I/O模型

一般模式：打开——读/写——关闭

步骤：

- ① 调用“打开”获得对文件或设备的使用权，并返回整型的文件描述符，此后使用它对该文件或设备进行I/O操作；
- ② 多次调用“读/写”传输数据；
- ③ 所有传输操作完成后，用户调用“关闭”，通知操作系统已经完成了对某个对象的使用。

- **Linux**中提供的基本I/O功能

操作	含义
<b>open</b>	为输入或输出操作准备一个设备或文件
<b>close</b>	终止使用以前已经打开的设备或文件
<b>read</b>	从输入设备或文件中获得数据，将数据放到应用程序的存储器中
<b>write</b>	将数据从应用程序的存储器传导输出设备或文件
<b>lseek</b>	转到文件或设备中的某个指定位置
<b>ioctl</b>	控制设备或用于访问该设备软件（如：指明缓存的大小或改变字符集的映射）

- **Linux**中对文件操作

- 打开一个文件

```
int desc;  
desc=open(“filename”,O_RDWR,0);
```

- 读取一个文件

```
read(desc, buffer, 128);
```

- 关闭文件

```
close(desc);
```

## 4.2.3实现网络进程通信必须解决的问题

- **TCP/IP的角色:**

相当于一种新的I/O操作，但比普通应用程序与传统的I/O操作的相互作用复杂得多。

- **网络应用程序要解决的问题:**

- 1)网络进程标识问题
- 2)多重协议识别问题
- 3)多种通信服务的选择问题

## 4.2.3实现网络进程通信必须解决的问题

- 在继承一般I/O操作的基础上,协议软件接口的设计扩展了以下若干环节:
  - 扩展了文件描述符集。使应用程序可以创建能被网络通信所使用的描述符（即套接字）
  - 扩展了读和写这两个系统调用，使其支持网络数据的发送和接收
  - 增加了对通信双方的标识
  - 指明通信所采用的协议
  - 确定通信扮演的角色（客户端或服务端）
  - 增加了对网络数据格式的识别和处理
  - 增加了对网络操作的控制等。

## 4.3.1 套接字编程接口的起源与发展

- 套接字起源于 20 世纪 70 年代加利福尼亚大学伯克利分校版本的 **Unix**,即人们所说的 **BSD Unix**。因此,有时人们也把套接字称为“伯克利套接字”或“**BSD 套接字**”。

# socket ( 套接字 )

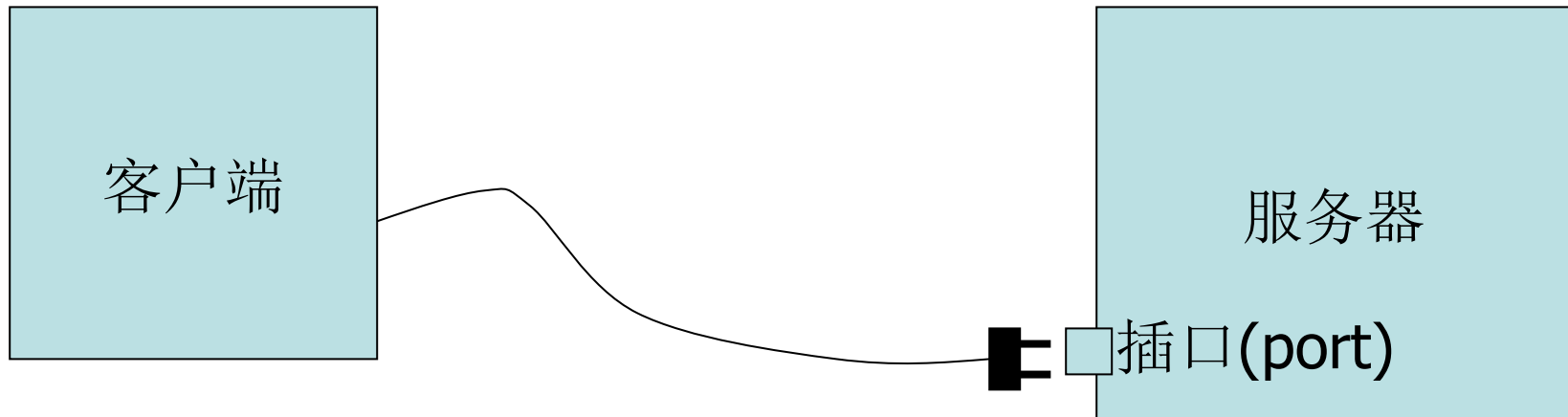
## socket

一个本地主机建立或拥有的应用程序，在操作系统控制下的，与其它(远程)应用进程之间发送和接收数据的接口。



# socket的抽象表示

- *socket* 是进行程序间通讯(IPC)的 BSD 方法。
- 客户将插头插入一个服务器端口
- 建立一个双向的连接管道



## 4.3.2套接字的抽象概念

- 套接字接口并没有直接用协议类型来标识通信时的协议，
- 而是采用:协议簇+套接字类型

## 4.3.2套接字的抽象概念

- 常用的协议簇
  - PF\_INET: IPv4协议簇
  - PF\_INET6: IPv6协议簇
  - PF\_IPX: IPX/SPX协议簇
  - PF\_NETBIOS: NetBIOS协议簇

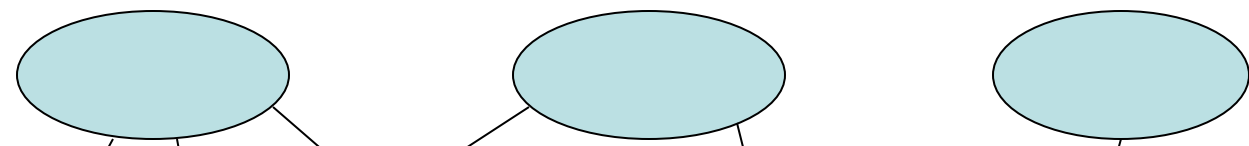
## 4.3.2套接字的抽象概念

- 常用的套接字包括三类
  - ① 流式套接字 (**SOCK\_STREAM**) : 提供面向连接的、可靠的字节流服务, 用于TCP。
  - ② 数据报套接字(**SOCK\_DGRAM**) : 提供无连接的, 不可靠的数据报服务, 用于UDP。
  - ③ 原始套接字 (**SOCK\_RAW**) : 允许对较低层的协议, 如IP、ICMP直接访问。

## 4.3.3套接字接口层的位置与内容

- 应用程序、套接字、协议和端口号之间的逻辑关系
  - 1) 一个应用程序可以同时使用多个套接字
  - 2) 多个应用程序可以同时使用同一个套接字
  - 3) 每个套接字都有一个关联的本地**TCP**或**UDP**端口
  - 4) **TCP**和**UDP**的端口号是独立使用的。一个**TCP**端口号可以关联多个套接字

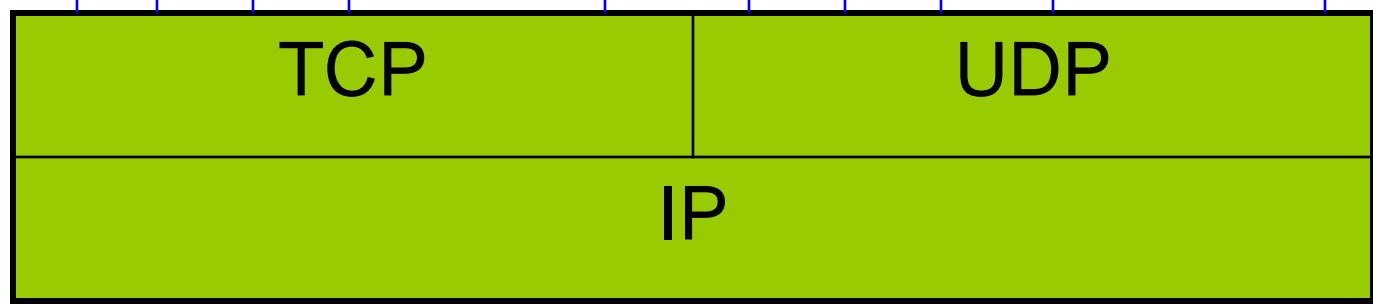
应用程序



TCP套接字



UDP套接字



## 4.3.3 套接字接口层的位置与内容

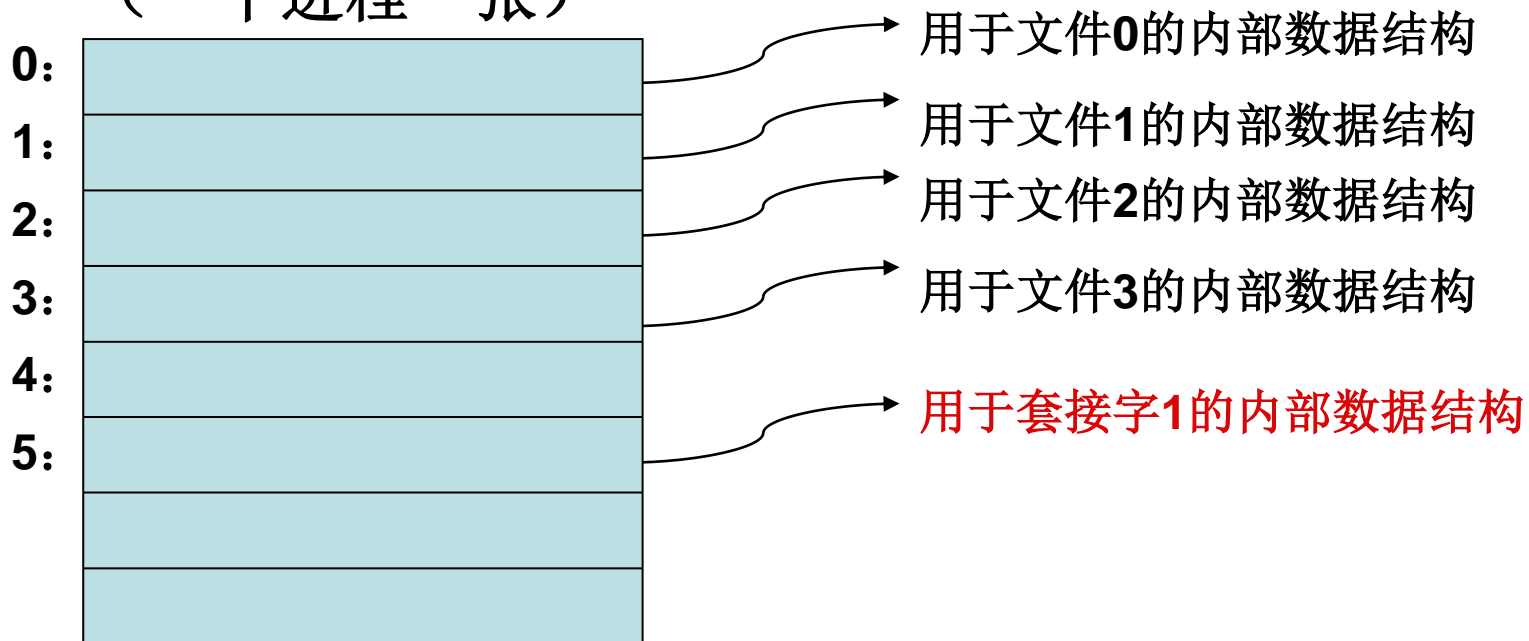
- 具体实现时，套接字表现为系统中的一个整型标识符，用于标识操作系统为该套接字分配的套接字结构，该结构保存了一个通信端点的数据集合。
- 套接字数据结构包括：
  - 套接字创建时所声明的通信协议类型
  - 与套接字关联的本地和远端IP地址和端口号
  - 一个等待向应用程序递送数据的接收队列和一个等待向协议栈传输数据的发送队列。
  - 对于流式套接字，还包含与打开和关闭TCP连接相关的协议状态信息

# 套接字描述符

- 套接字描述符和文件描述符

文件描述符表

(一个进程一张)

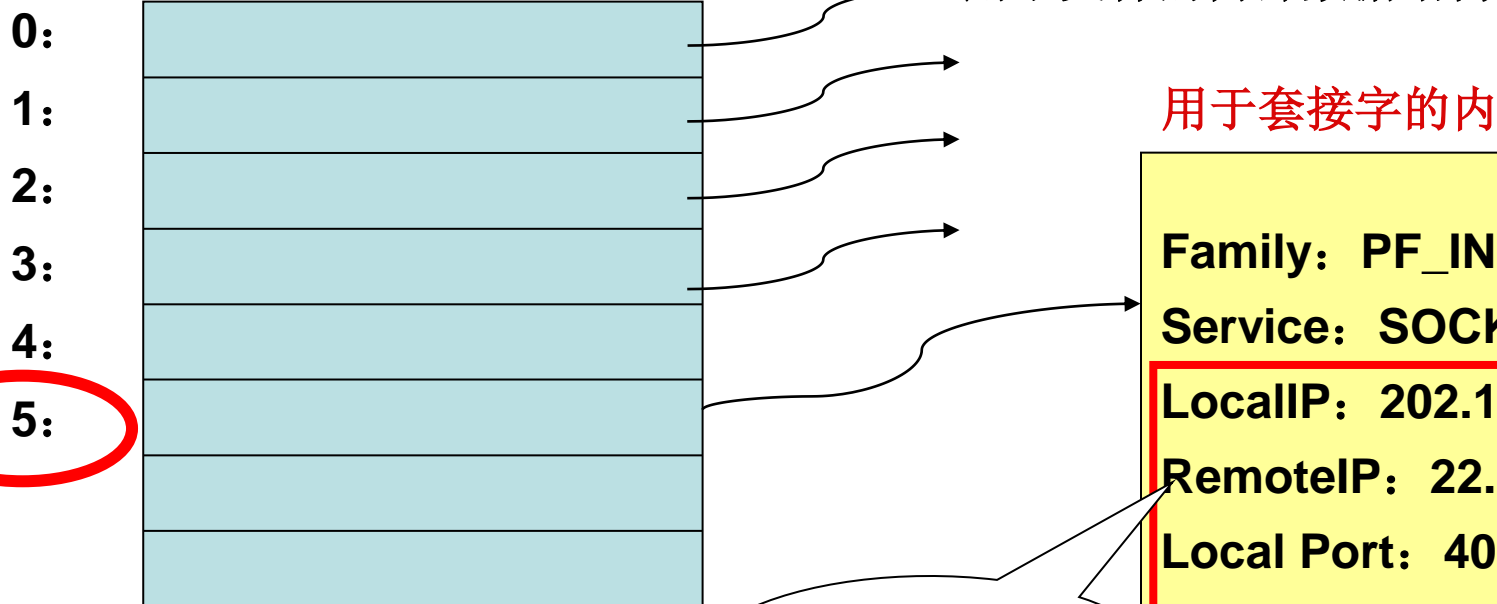




# • 针对套接字的系统数据结构

文件描述符表

(一个进程一张)



**Family: PF\_INET**  
**Service: SOCK\_STREAM**  
**LocalIP: 202.196.63.5**  
**RemoteIP: 22.196.56.3**  
**Local Port: 4000**  
**Remote Port: 8000**

端点地址

## 4.3.4套接字通信

- 进行网络通信至少需要一对套接字。
- 其中一个运行在客户端，称之为客户端套接字。
- 一个运行在服务器端，称之为服务器套接字。
- 网络应用程序的通信是以套接字为线索，以五元组(协议,源IP,源端口,目的IP,目的端口)为通信基础进行数据传输。

## 4.3.4套接字通信

- **socket编程步骤**
  - 建立一个socket
  - 配置socket
  - 连接socket
  - 通过socket发送数据
  - 通过socket接收数据
  - 关闭socket

## 4.3.4套接字通信

- 使用套接字进行数据处理有两种基本模式
  - **同步模式**：连接、接收、发送数据时，进程处于堵塞状态，直到I/O条件满足（处理完）才能继续。适合于数据处理不太多的场合。
  - **异步模式**：连接、接收、发送数据时，进程不等待，直接继续执行。通过其它多种机制来处理I/O结果。适合于大数据量处理。

## 4.4.1 Winsock Socket规范

- Windows Sockets是微软以BSD UNIX的Berkeley Sockets规范为基础，定义的基于Windows的网络编程接口规范；
- Windows Sockets规范给出了一套库函数的函数原型和函数功能说明，这些函数由底层网络软件供应商实现，供高层网络应用程序开发者使用；
- 不仅包括了Berkeley Sockets风格的库函数，同时也提供了一套Windows所特有的扩展库函数，使程序员能够使用Windows系统的消息驱动机制。

## 4.4.2 Windows Sockets的版本

- 1.1983年，加利福尼亚大学Berkely学院推出了UNIX下的网络通信接口Socket。
- 2.90年代初，Sun Microsystems、JSB Corporation、FTP software、Microdyne和微软等公司共同参与制定了Windows Socket规范，试图使Windows下的Sockets程序设计标准化。
- 3.1992年制定Windows Socket规范1.0版，将Socket从UNIX移植到DOS和Windows下。
- 4.1993年1月，制定了Windows Socket 1.1版。
- 5.1994年5月，WinSock小组启动WinSock2规范制定工作。
- 6.1997年5月，WinSock2的正式规范版本2.2.1发布。

# Windows Sockets 1.1的变更

- 增加了`gethostname()`调用，简化主机名字和地址的获取
- 将DLL中小于1000的序数定义为Windows Sockets保留,而对于大于1000的序数则没有保留。
- 为`WSAStartup()`和`WSACleanup()`函数增加引用计数，并要求两个函数调用时成对出现

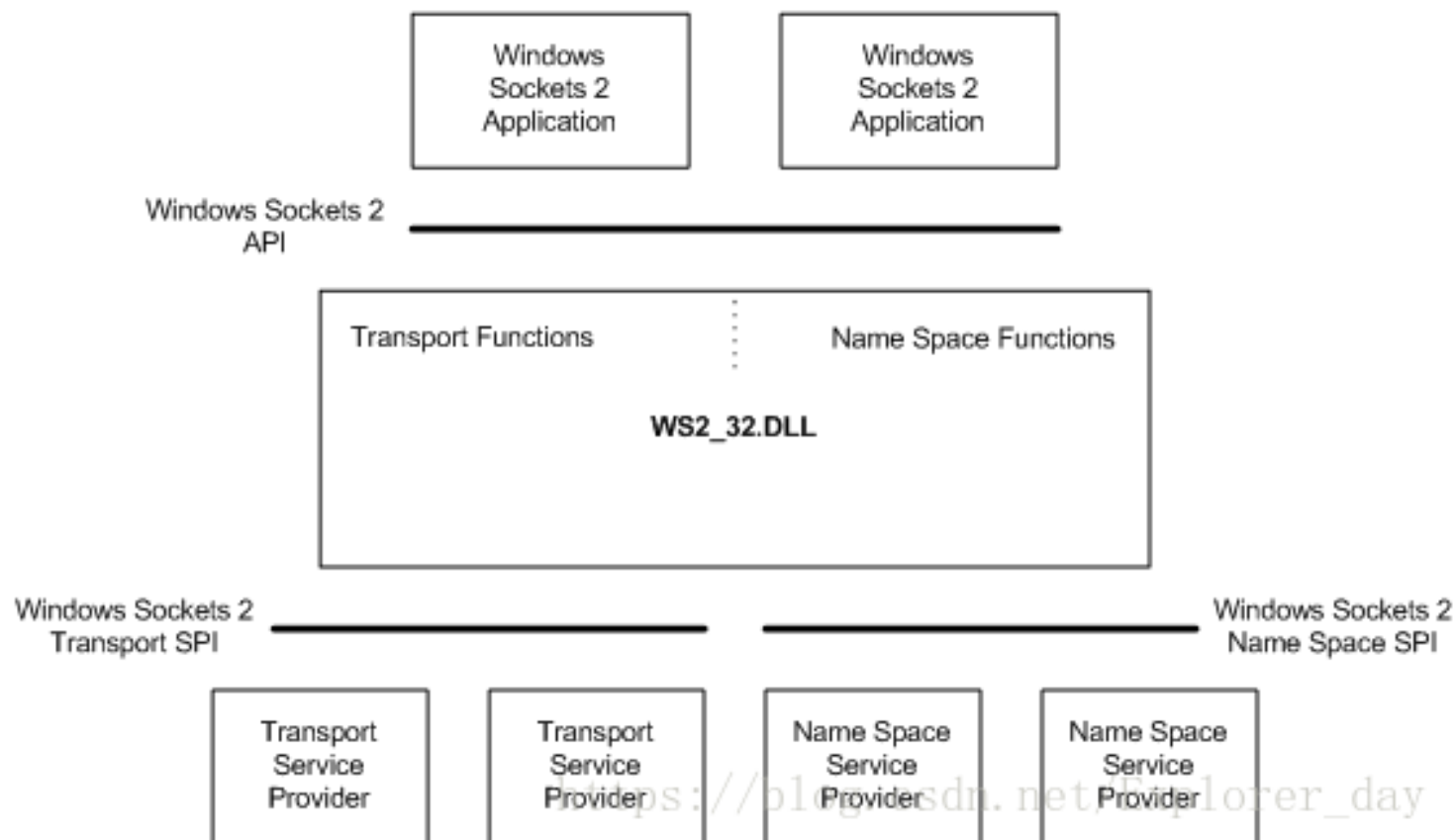
# Windows Sockets 2的扩展

- 1) 体系结构的改变
- 2) 套接字句柄的改变
- 3) 对多协议簇的支持
- 4) 协议独立的名字解析能力
- 5) 分散/聚集I/O支持
- 6) 服务质量控制
- 7) 与WinSock1.1的兼容性
- 8) 协议独立的多播和多点
- 9) 其他经常需要的扩展



# Windows Sockets 2的扩展

- 1) 体系结构的改变
- WinSock 2通过在WinSock DLL和协议栈之间定义一个标准的服务提供者接口(Service Provider Interface, SPI),改变了原WinSock 1.1和底层协议栈之间的私有接口模式,这使得从单个WinSock DLL中同时访问来自多个厂商的多个协议栈成为可能。



# Windows Sockets 2的扩展

- 2) 套接字句柄的改变
- 在Windows Sockets 2中，套接字句柄可以  
是文件句柄。
- 这意味着可以将套接字句柄用于Windows  
文件I/O函数。
- 并非所有的传输服务提供者都支持这个选项。

# Windows Sockets 2的扩展

- 3)对多协议簇的支持
  - 使应用程序可以利用熟悉的套接字接口访问其它已安装的传输协议。Windows Sockets 1.1仅支持TCP/IP协议簇
- 4)协议独立的名字解析能力
  - 包含一套标准API用于现有的大量名字解析域名系统。

# Windows Sockets 2的扩展

- 5) 分散/聚集I/O支持
  - 提供的**API**以应用程序缓冲区作为输入参数，用于执行分散/聚集I/O操作
  - 当应用程序传送的信息除了信息体外还包含一个或多个固定长度的首部时，发送之前不需要由应用程序将这些首部和数据连接到一个连续的缓冲区中， 就可以直接将分散的多个缓冲区的数据发送出去。

# Windows Sockets 2的扩展

- 6) 服务质量控制
  - WinSock 2为应用程序提供了协商所需服务等级的能力
- 7) 与WinSock1.1的兼容性
  - WinSock 2与WinSock 1.1在两个级别上保持兼容：源码级和二进制级
- 8) 协议独立的多播和多点
  - 可以发现传输层提供的多播或多点能力的类型
- 9) 其他经常需要的扩展
  - 如共享套接字、附条件接收等

## 4.4.3. WinSock的组成

### ① 开发组件

**功能：** 供程序员开发Windows Socket应用程序

**组成：**

- a. 介绍Windows Socket实现的文档
- b. Windows Socket应用程序接口（API）导入库
- c. 一些头文件

其中**WINSOCK.H**：包括了WinSock实现所定义的宏、常数值、数据结构和函数调用接口原型；

### ② 运行组件

**WINSOCK.DLL**：Windows应用程序接口的动态链接库

**（DLL）**，应用程序在执行时通过装入它实现网络通信功能。

两组运行必须的组件：

第一版： **winsock.h**    **winsock.dll**    **winsock.lib**

第二版： **winsock2.h**    **ws2\_32.dll**    **ws2\_32.lib**

# Windows Socket 1.1库函数

- ① 套接字函数
- ② 数据库函数
- ③ **Windows**扩充的专有函数



# 一. 套接字函数

## 第一类：套接字函数

### 功能：

完成套接字的创建、关闭以及对套接字的命名和名字获取。

.....

函数名 ↵	功能 ↵	↵
<u>bind()</u> ↵	给套接字绑定本地地址和端口 ↵	↵
<u>closesocket()</u> ↵	关闭套接字 ↵	↵
<u>getpeername()</u> ↵	获取与指定套接字连接的对等方的地址和端口号 ↵	↵
<u>getsockname()</u> ↵	获取指定套接字关联的地址和端口号 ↵	↵
<u>socket()</u> ↵	创建套接字 ↵	↵

# 一. 套接字函数

## 第二类：网络连接函数

### 功能：

完成网络连接的建立与关闭。

函数名	功能
<code>accept()</code>	确认外来连接，并将它与一个立即建立的数据套接字联系起来。
<code>connect()</code>	在指定的套接字上与远程主机的端口建立连接
<code>listen()</code>	在指定套接字上监听外来连接
<code>shutdown()</code>	关闭全双工连接中一个方向上的连接

# 一. 套接字函数

## 第三类：数据传输函数

### 功能：

完成数据的发送与接收。

函数名 ↵	功能 ↵
<u>recv()</u> ↵	从一个面向连接的 <u>套接字</u> 上接收数据 ↵
<u>recvfrom()</u> ↵	从一个面向连接或无连接的 <u>套接字</u> 上接收数据 ↵
<u>send()</u> ↵	使用面向连接的 <u>套接字</u> 发送数据 ↵
<u>sendto()</u> ↵	使用面向连接或无连接的 <u>套接字</u> 发送数据 ↵

# 一. 套接字函数

## 第四类：字节顺序转换函数

功能：

完成主机字节顺序和网络字节顺序之间的转换。

函数名↵	功能↵
<u>htonl()</u> ↵	将一个 32 位整数从主机字节顺序转换为网络字节顺序↵
<u>htons()</u> ↵	将一个 16 位整数从主机字节顺序转换为网络字节顺序↵
<u>ntohl()</u> ↵	将一个 32 位整数从网络字节顺序转换为主机字节顺序↵
<u>ntohs()</u> ↵	将一个 16 位整数从网络字节顺序转换为主机字节顺序↵

# 一. 套接字函数

## 第五类：地址转换函数

### 功能：

完成**IP**地址的点分十进制形式和二进制整数形式之间的转换。

函数名	功能
<u>inet_addr()</u> ×	将一个用点分十进制表示的字符串地址转换为整数形式表示的 IP 地址
<u>inet_ntoa()</u> ×	将整数表示的 IP 地址转化为点分十进制表示的字符串地址

# 一. 套接字函数

## 第六类：套接字控制函数

### 功能：

设置/获取套接字的选项；控制/检测套接字的工作状态。

函数名↴	功能↴
<code>getsockopt()</code> ↴	获取与指定套接字相关的选项↴
<code>ioctlsocket()</code> ↴	为套接字提供控制↴
<code>select()</code> ↴	获得一组套接字的状态，以实现异步I/O操作
<code>setsockopt()</code> ↴	设置与指定套接字相关的选项↴

## 二. 数据库函数

函数名 ↗	功能 ↗
<u>gethostbyaddr()</u> ↗ ✗	通过网络地址获取主机信息 ↗
<u>gethostbyname()</u> ↗ ✗	通过主机名字获取主机信息 ↗
<u>gethostname()</u> ↗	获取本地主机名 ↗
<u>getprotobyname()</u> ↗	通过协议名获取协议信息 ↗
<u>getprotobynumber()</u> ↗	通过协议号获取协议信息 ↗
<u>getservbyname()</u> ↗	通过服务名获取服务的名字和端口等信息 ↗
<u>getservbyport()</u> ↗	通过端口获取服务名字和端口等信息 ↗

# 三. Windows Socket专用的增设函数

## 第一类：启动与终止函数

函数名 ↗	功能 ↗
<u>WSAStartup()</u> ↗	初始化 Windows Sockets DLL ↗
<u>WSACleanup()</u> ↗	终止对 Windows Sockets DLL 的调用 ↗



# 三. Windows Socket专用的增设函数

## 第二类：异步服务函数

函数名 ↗	功能 ↗
<u>WSAAsyncGetHostByAddr()</u> ↗ X	标准 Berkeley 函数 <u>getXbyY</u> 的异步版本，例如： ↗ <u>WSAAsyncGetHostByAddr()</u> 提供了标准 Berkeley 函数 <u>gethostbyaddr()</u> 的一种基于消息的异步实现。 ↗
<u>WSAAsyncGetHostByName()</u> ↗ X	
<u>WSAAsyncGetProtoByName()</u> ↗ X	
<u>WSAAsyncGetProtoByNumber()</u> ↗ X	
<u>WSAAsyncGetServByName()</u> ↗ X	
<u>WSAAsyncGetServByPort()</u> ↗ X	
<u>WSACancelAsyncRequest()</u> ↗	

# 三. Windows Socket专用的增设函数

## 第三类：基于消息机制的异步I/O函数

函数名	功能
WSAAsyncSelect( ) X	Berkeley套接字中的select在WinSock中的扩充，获得一组套接字的状态，以实现基于消息机制的异步I/O操作。

## 第四类：阻塞处理函数

函数名 ↕	功能 ↕
<u>WSAIsBlocking()</u> ↕ X	检测隐含的 Windows Sockets DLL 是否阻塞了一个当前线程的调用 ↕
<u>WSACancelBlockingCall()</u> ↕ X	取消一个执行中的“阻塞”API 调用 ↕
<u>WSASetBlockingHook()</u> ↕ X	设置应用程序自己的“阻塞”处理函数 ↕
<u>WSAUnhookBlockingHook()</u> ↕ X	恢复原来的“阻塞”处理函数 ↕

# 三. Windows Socket专用的增设函数

## 第五类：错误处理函数

函数名	功能
<u>WSAGetLastError()</u>	设置 Windows Sockets API 的最近错误码
<u>WSASetLastError()</u>	获取 Windows Sockets API 的最近错误码

**说明：**

**调试程序时非常有用；**

**MSDN索引：Win32 Error Codes和Net Error Codes；**

**例如：**

**10060 ( WSAETIMEDOUT )：连接超时**

## 四、Windows Socket 2 扩展

说明：

扩充了新的内容，同时与**Windows Sockets 1.1**向后兼容。

编写应用程序时需要包含头文件：  
**“Winsock2.h.”**

在**32位 Windows**下应包含**WS2\_32.lib**，使用**WS2\_32.DLL**。

# Visual C++使用WinSock的步骤

1. 包含WinSock头文件：在程序文件首部使用编译预处理命令“`#include`”，将WinSock头文件包含进来。例如：

```
#include <WinSock2.h>
```

2. 链接WinSock导入库，有两种方式：

- 通过在项目属性页中的“配置属性\链接器\输入”的“附加依赖项”中直接添加导入库名字
- 在程序中使用预处理命令“`#pragma comment`”。例如，程序要使用WinSock2时，可使用如下预处理命令：

```
#pragma comment (lib, "Ws2_32.lib")
```

3.加载WinSock动态链接库：WinSock库函数是以动态链接库的形式实现的，因此，在程序中调用WinSock函数时，必须已经加载了WinSock动态链接库。

- 加载WinSock动态链接库使用WSAStartup()函数，函数圆形如下：

```
int WSAStartup (  
    WORD  wVersionRequested,  //版本号  
    LPWSADATA  lpWSADATA  //  
);
```

返回值是一个整数，函数调用成功则返回0

# WSAStartup函数

- 作用：启动win\_socket的dll库，初始化winsock所对应的ws2\_32.dll，完成套接字初始化
- 加载WinSock DLL的相应版本

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

返回一个WSADATA结构

WinSock库的版本

高位字节指定副版本号  
低位字节指定主版本号

可以用宏MAKEWORD(X, Y)  
方便地设置

- 函数参数:

- **wVersionRequested**: 该参数是一个双字节类型数值，用于指明程序中要使用的WinSock库的版本号，其中高位字节指定副版本、低位字节指定主版本。

早期Windows平台一般使用WinSock1.1，其版本号是1.1，现在常用的则是WinSock2的2.2版。指定该参数值时可使用16进制方式给出，比如要加载WinSock2.2版，该值可设定为0x0202，但更常用的方法则是使用宏**MAKEWORD**（X，Y），参数X为副版本号，Y为主版本号。



- **lpWSAData**: 用于返回关于使用的WinSock版本的详细信息，它是一个指向**WSADATA** 结构体变量的指针。
- **WSADATA** 结构体的定义如下

```

typedef struct WSADATA {
    WORD                wVersion;
    WORD                wHighVersion;

#ifdef _WIN64
    unsigned short      iMaxSockets;
    unsigned short      iMaxUdpDg;
    char FAR *          lpVendorInfo;
    char                szDescription[WSADESCRIPTION_LEN+1];
    char                szSystemStatus[WSASYS_STATUS_LEN+1];
#else
    char                szDescription[WSADESCRIPTION_LEN+1];
    char                szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short      iMaxSockets;
    unsigned short      iMaxUdpDg;
    char FAR *          lpVendorInfo;
#endif
} WSADATA, FAR * LPWSADATA;

```

```
#define WSADESCRIPTION_LEN 256
#define WSASYSSTATUS_LEN 128
Typedef struct WSADATA {
WORD wVersion; //期望程序使用的 WinSock版本号
WORD wHighVersion; //加载的Winsock库支持的最高版本号
char szDescription[WSADESCRIPTION_LEN+1]; //Winsock库说明
char szSystemStatus[WSASYSSTATUS_LEN+1]; //状态和配置信息
//以下字段被Winsock2及其后版本忽略
unsigned short iMaxSockets;
//能同时打开的socket的最大数
unsigned short iMaxUdpDg; //可发的UDP送数据报的最大字节数
char FAR * lpVendorInfo; //厂商指定信息
} WSADATA;
```

- 假如一个程序要使用2.2版本的Winsock，那么程序中可采用如下代码加载Winsock动态链接库：

```
WSADATA wsaData;  
WORD wVersionRequested;  
wVersionRequested = MAKEWORD( 2, 2 );  
int err = WSAStartup( wVersionRequested, &wsaData );  
if (err!=0)  
{  
    //Winsock初始化错误处理代码  
    ...  
}
```

## 4. 注销WinSock动态链接库：通信工作完成后，程序关闭之前应卸载WinSock动态链接库并释放资源。

- 注销WinSock动态链接库使用WSACleanup:

`int WSACleanup (void);`

- 该函数无参数，执行后将返回一个整数值，如果操作成功返回0，否则返回SOCKET\_ERROR。常量SOCKET\_ERROR在Winsok2.h（或Winsok.h）中定义，其对应值为-1。
- 对应于一个程序中的每一次WSAStartup（）调用，都应该有一个WSACleanup（）调用。

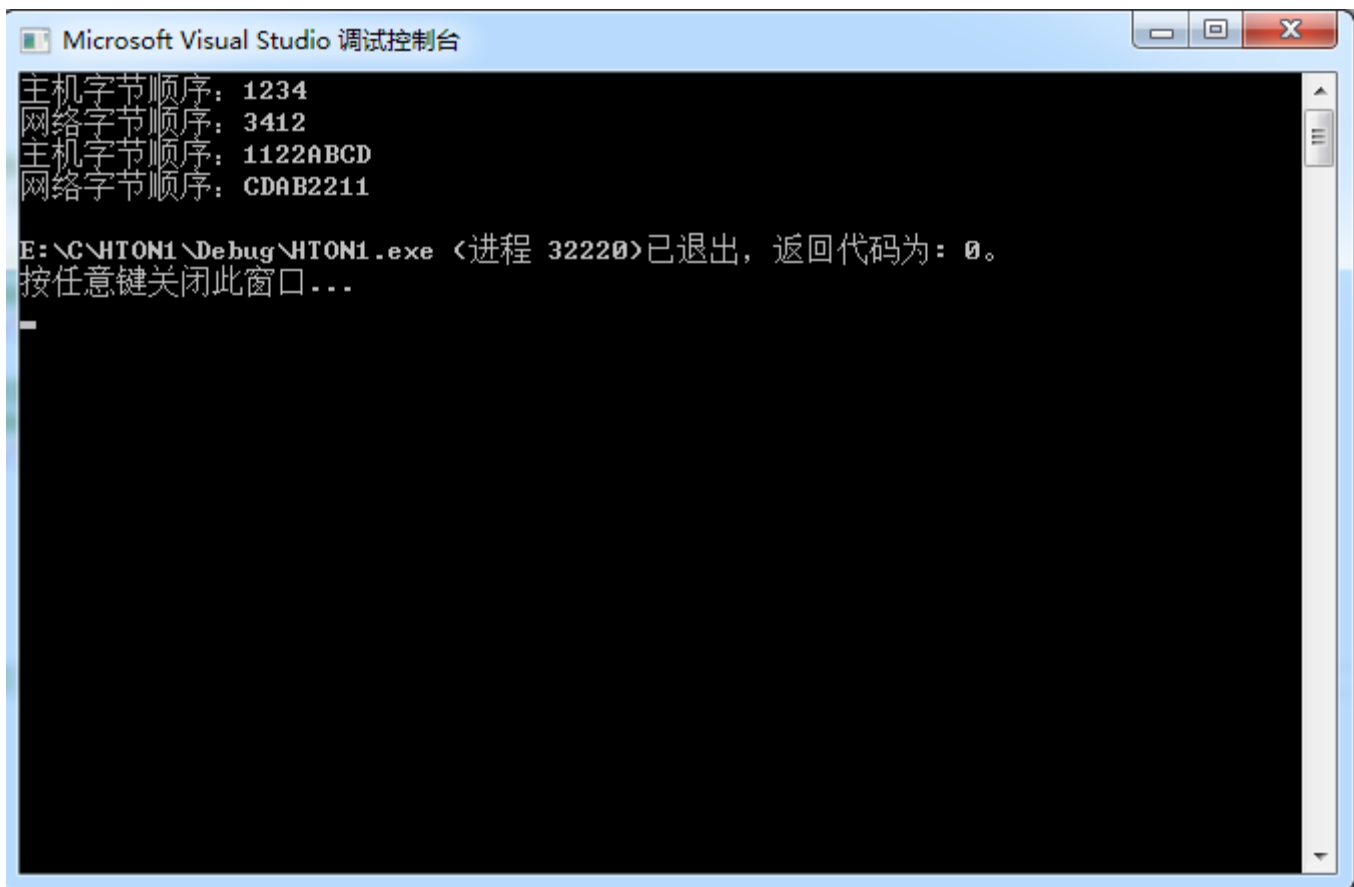
```
#include <iostream>
#include <WinSock2.h> //使用WinSock2进行开发需要用到头文件<winsock.h>
#pragma comment(lib, "ws2_32.lib") //加载库文件“ws2_32.lib”

int main()
{
    //初始化Winsock DLL
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 2); //生成版本号2.2
    if (WSAStartup(wVersionRequested, &wsaData) != 0) return 0;

    u_short x, y = 0x1234; //定义无符号短整型变量x、y，并将y初始化为十六进制数0x1234
    x = htons(y); //将y的值转换为网络字节顺序并将转换结果存入变量x
    printf("主机字节顺序：%X\n网络字节顺序：%X\n", y, x);

    u_long a, b = 0x1122ABCD; //定义无符号长整型变量a、b，并将b初始化
    a = htonl(b); //将b的值转换为网络字节顺序并将转换结果存入变量a
    printf("主机字节顺序：%X\n网络字节顺序：%X\n", b, a);

    //注销Winsock DLL
    WSACleanup();
    return 0;
}
```



# WinSock2中的扩展字节顺转换函数

- `int WSAHtons( SOCKET s, u_short hostshort, u_short* lpnetshort);`
- 函数参数
  - **s**: 套接字描述符。
  - **hostshort**: 一个待转换的主机字节顺序的**16**位无符号短整型数据。
  - **lpnetshort**: 指向一个**16**位无符号短整型变量的指针, 该指针用于存储转后的网络字节顺序的**16**位数据。
- 返回值

函数调用成功则返回**0**。函数调用失败, 则返回**SOCKET\_ERROR**



- `int WSANTohs( SOCKET s, u_short netshort, u_short* lphostshort);`
- 函数参数
  - `s`: 套接字描述符。
  - `netshort`: 一个待转换的网络字节顺序的**16**位无符号短整型数据。
  - `lphostshort`: 指向一个**16**位无符号短整型变量的指针，该指针用于存储转后的主机字节顺序的**16**位数据。
- 返回值

函数调用成功则返回**0**。函数调用失败，则返回**SOCKET\_ERROR**

- `int WSAHtonl( SOCKET s, u_long  
hostlong, u_long* lpnetlong);`
- `int WSANtohl ( SOCKET s, u_long netlong,  
u_long* lphostlong);`

```

#include <iostream>
#include <WinSock2.h> //使用WinSock2进行开发需要用到头文件<winsock.h>
#pragma comment(lib, "ws2_32.lib") //加载库文件 "ws2_32.lib"

using namespace std;

int main()
{
    //初始化Winsock DLL
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 2); //生成版本号2.2
    if (WSAStartup(wVersionRequested, &wsaData) != 0) return 0;

    SOCKET s; //定义套接字
    s = socket(AF_INET, SOCK_STREAM, 0); //创建套接字

    /*将0x1122ABCD转换为网络字节顺序存入变量a中*/
    u_long a;
    WSAHtonl(s, 0x1122ABCD, &a);
    cout << "0x1122ABCD的网络字节顺序为:" << hex << a << endl;

    /**定义无符号长整型变量x、y，并将y初始化为十六进制数0x1122abcd***/
    u_long x, y = 0xEEDFABCD;
    WSAHtonl(s, y, &x);
    cout << "主机字节顺序:" << hex << y << endl << "网络字节顺序:" << x << endl;

    //注销WinSock DLL
    WSACleanup();
    return 0;
}

```

Microsoft Visual Studio 调试控制台

0x1122ABCD的网络字节顺序为: cdab2211

主机字节顺序:eedfabcd

网络字节顺序:cdabddfee

E:\C\HTON2\Debug\HTON2.exe <进程 31060>已退出, 返回代码为: 0。

按任意键关闭此窗口...

## 4.5.3 WinSock的网络地址表示

- 在IP网络环境下，对于一个通信进程而言，必须明确三方面信息：
  - 进程所在的主机IP地址：用于区分网络中的不同主机
  - 通信所采用的协议：指明通信所使用的传输层协议是TCP还是UDP
  - 协议端口号：用于区分同一主机中运行的采用同一传输层协议的不同进程。
- 通过这三方面的信息可以唯一确定在网络中参与通信的一个进程，因此进程的网络地址可以使用三元组（协议，IP地址，端口号）来标识。

## 4.5.3 Winsock的地址描述

- 套接字适用于多种协议簇，它既没有指明如何定义端点地址，也没有定义一种特定的协议地址格式，而是改为允许每个协议簇自由定义。
- 套接字为每种类型的地址定义了一个地址族。
- 一个协议簇可以使用一种或多种地址族来定义地址表示方式

## 4.5.3 Winsock的地址描述

- 常见的地址簇
  - AF\_INET: IPv4地址族
  - AF\_INET6: IPv6地址族
  - AF\_IPX: IPX/SPX地址族
  - AF\_NETBIOS: NetBIOS地址族

Value	Meaning
<b>AF_UNSPEC</b> 0	The address family is unspecified.
<b>AF_INET</b> 2	The Internet Protocol version 4 (IPv4) address family.
<b>AF_NETBIOS</b> 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
<b>AF_INET6</b> 23	The Internet Protocol version 6 (IPv6) address family.
<b>AF_IRDA</b> 26	The Infrared Data Association (IrDA) address family. This address family is only supported if the computer has an infrared port and driver installed.
<b>AF_BTH</b> 32	The Bluetooth address family. This address family is only supported if a Bluetooth adapter is installed on Windows Server 2003 or later.



## 4.5.3 Winsock的地址描述

- 套接字的一般地址结构
  - {地址族, 端点地址}
- 最一般的地址结构定义为`sockaddr`结构, 其精确形式依赖于地址族
- 常用的IPv4地址结构是`sockaddr_in`

# sockaddr结构体

- 通用结构，用来保存socket信息

```
struct sockaddr {  
    u_short sa_family;  
    char    sa_data[14];  
};
```

```
typedef struct sockaddr {  
    #if ...  
        u_short      sa_family;  
    #else  
        ADDRESS_FAMILY sa_family;  
    #endif  
    CHAR              sa_data[14];  
} SOCKADDR, *PSOCKADDR, *LPSOCKADDR;
```

# sockaddr\_in结构体

- 指定IPv4地址结构

```
struct sockaddr_in {  
    short        sin_family;  
    u_short      sin_port;  
    struct in_addr sin_addr;  
    char         sin_zero[8];  
};
```

# in\_addr结构体

IP地址常用点分法来表示： 192.168.0.1

计算机中使用无符号长整数（ unsigned long ）来存储和表示IP地址

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

# sockaddr\_in6结构体

```
typedef struct sockaddr_in6 {  
    ADDRESS_FAMILY sin6_family; // AF_INET6.  
    USHORT sin6_port;           // Transport level port number.  
    ULONG sin6_flowinfo;        // IPv6 flow information.  
    IN6_ADDR sin6_addr;         // IPv6 address.  
    union {  
        ULONG sin6_scope_id;    // Set of interfaces for a scope.  
        SCOPE_ID sin6_scope_struct;  
    };  
} SOCKADDR_IN6_LH, *PSOCKADDR_IN6_LH, FAR *LPSOCKADDR_IN6_LH;
```

# in6\_addr结构体

```
//  
= typedef struct in6_addr {  
=     union {  
         UCHAR      Byte[16];  
         USHORT     Word[8];  
     } u;  
} IN6_ADDR, *PIN6_ADDR, FAR *LPIN6_ADDR;
```

# addrinfo

- WinSock 2增加了addrinfo结构用于描述地址信息，该结构在getaddrinfo()函数中使用，提供了一个以链表形式保存的地址信息。

# addrinfo

```
typedef struct addrinfo {  
    int             ai_flags; //getaddrinfo函数调用选项  
    int             ai_family; //地址族  
    int             ai_socktype; //套接字类型  
    int             ai_protocol; //协议  
    size_t          ai_addrlen; //ai_addr指向的缓冲区字节长度  
    char            *ai_canonname; //主机的正规名称  
    struct sockaddr *ai_addr; //以sockaddr结构描述的信息  
    struct addrinfo *ai_next; //链表下一个节点  
} ADDRINFOA, *PADDRINFOA;
```

地址结构中的内容都以网络字节顺序表示



# getaddrinfo

- 提供一种协议无关的地址获取和表示方式

```
INT WINAPI getaddrinfo(  
    PCSTR                pNodeName,    //主机名或ip地址字符串  
    PCSTR                pServiceName, //服务名或端口号  
    const ADDRINFOA *pHints,    //线索信息  
    PADDRINFOA          *ppResult //返回的结果  
);
```

[getaddrinfo](#)详细信息

使用getaddrinfo不要忘了用**freeaddrinfo**来释放获取的ppResult地址信息

## 4.5.3 地址转换函数

- `inet_addr`函数：将点分十进制字符串表示的IP地址转换为32位的无符号长整型数，它是以网络字节顺序表示的32位二进制IP地址。

`unsigned long inet_addr (const char * cp);`

- 函数参数
  - **cp**: 指向存放有一个点分十进制表示的IP地址的字符串。
- 如果转换成功，则返回网络字节顺序存储的32位二进制IPv4地址
- 如果传入的字符串是一个非法的IP地址，函数返回值是常量 `INADDR_NONE`。
- 新版本已废弃，改用`inet_pton`

# inet\_pton

```
INT WINAPI inet_pton(  
    INT    Family,      //地址族AF_INET 或AF_INET6  
    PCSTR  pszAddrString, //地址字符串  
    PVOID  pAddrBuf     //输出的地址结构体指针IN_ADDR或IN6_ADDR  
);
```

`char * inet_ntoa (struct in_addr in);`

- 将一个包含在`in_addr`结构变量`in`中的长整型IP地址转换点分十进制形式。
- 参数`in`：是一个保存有32位二进制IP地址的`in_addr`结构变量。
- 函数调用成功返回一个字符指针，该指针指向一个`char`型缓冲区，该缓冲区保存有由参数`in`的值转换而来的点分十进制表示的IP地址字符串。如果函数调用失败，则返回一个空指针`NULL`。
- 新版本已废弃，改用`inet_ntop`

# inet\_ntop

```
PCSTR WINAPI inet_ntop(  
    INT                Family, //地址族AF_INET 或AF_INET6  
    const VOID *pAddr, //指向地址结构体指针IN_ADDR或IN6_ADDR  
    PSTR               pStringBuf, //字符串缓冲区，存放结果  
    size_t             StringBufSize //缓冲区长度  
);
```

- 为了扩展对IPv6地址转换的支持，WinSock 2分别提供了WSAStringToAddress()和WSAAddressToString两个函数

```

#include <iostream>
#include "WinSock2.h"
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
using namespace std;

int main()
{
    char hostname[256];
    struct addrinfo* result = NULL;
    struct addrinfo* ptr = NULL;
    char strIP[16];

    //初始化WinSock DLL
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 2); //生成版本号2.2
    if (WSAStartup(wVersionRequested, &wsaData) != 0) return 0;

    if (gethostname(hostname, sizeof(hostname))) {
        cout << "gethostname calling error" << endl;
        WSACleanup(); return 0;
    }
    cout << "本机名字为: " << hostname << endl;
}

```

```

if (getaddrinfo("", "", NULL, &result)) {
    cout << "名字解析失败" << endl;
    WSACleanup();
    return 0;
}

/**输出IP地址**/
cout << "本机IP地址：" << endl;
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
    inet_ntop(AF_INET, &(((sockaddr_in*)(ptr->ai_addr))->sin_addr), strIP, 16);
    cout << strIP << endl;
}
freeaddrinfo(result); //释放获取的地址信息结构

cout << "输入要解析的域名：" << endl;
cin >> hostname;
if (getaddrinfo(hostname, "", NULL, &result)) {
    cout << "名字解析失败" << endl;
    WSACleanup();
    return 0;
}

/**输出IP地址**/
cout << "主机IP地址：" << endl;
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
    inet_ntop(AF_INET, &(((sockaddr_in*)(ptr->ai_addr))->sin_addr), strIP, 16);
    cout << strIP << endl;
}
freeaddrinfo(result); //释放获取的地址信息结构

WSACleanup();
return 0;

```



```
Microsoft Visual Studio 调试控制台

本机名字为: C0UHBCQ13PWLRI0
本机IP地址:
0.0.0.0
0.0.0.0
0.0.0.0
0.0.0.0
0.0.0.0
0.0.0.0
192.168.1.2
192.168.230.1
169.254.191.61
输入要解析的域名:
www.fzu.edu.cn
主机IP地址:
0.0.0.0
59.77.231.60

E:\C\HOSTNAME\Debug\HOSTNAME.exe <进程 10648>已退出, 返回代码为: 0。
按任意键关闭此窗口...
```

# 协议查询

- 因特网的每一个协议都有一个正式的名字，比如IP、ICMP、TCP等，但在网络内部，协议并不是使用它们的名字标识的，而是使用分配给它们的一个唯一编号来表示的，比如IP的编号为0，ICMP的编号为1，TCP的编号为6等。
- 因特网协议的编号是由IANA统一管理的。协议的名字主要便于人们阅读，而协议的编号则有利于计算机处理。
- 协议号目前主要用于IP报文的协议字段
- [协议号速查](#)
- WinSock提供了两个函数getprotobyname()和getprotobynumber()来帮助程序完成协议名字和编号之间的转换。

- 1) `getprotobyname()`
- 根据协议的名字查询相应的协议编号等信息。

`struct protoent *getprotobyname(const char * name);`

- 函数参数
  - **name**: 一个指向协议名的指针。
- 返回值
  - 函数调用成功，将返回一个指向**protoent**结构的指针，该**protoent**结构由**WinSock**创建并管理，应用程序不能修改或者释放它的任何部分；函数调用失败则返回一个空指针。

- protoent结构的声明如下：

```
struct protoent {
```

```
char p_name;      //正规的协议名。
```

```
char **p_aliases; //一个以空指针结尾的可选  
    协议名队列。
```

```
short p_proto;    //主机字节顺序的协议号。
```

```
};
```

- `getprotobynumber()`
- 根据协议号查询协议的名字等信息。

`struct protoent * getprotobynumber(int number);`

- 函数参数
  - **number**: 一个以主机顺序排列的协议号。
- 返回值
  - 函数调用成功，将返回一个指向**protoent**结构的指针，该**protoent**结构由WinSock创建并管理，应用程序不能修改或者释放它的任何部分；函数调用失败则返回一个空指针。

```

#include <iostream>
#include <WinSock2.h> //使用WinSock2进行开发需要用到头文件<winsock.h>
#pragma comment(lib, "ws2_32.lib") //加载库文件 "ws2_32.lib"

using namespace std;

int main()
{
    protoent* pro=NULL; //协议数据
    char pronom[10]; //协议名
    short proid; //协议号

    //初始化Winsock DLL
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 2); //生成版本号2.2
    if (WSAStartup(wVersionRequested, &wsaData) != 0) return 0;

    cout << "输入协议名:" << endl;
    cin >> pronom;
    pro = getprotobyname(pronom);
    if (pro != NULL)
        cout << pronom<<"的协议号是" << pro->p_proto << endl;
    else
        cout << pronom<<"协议名不合法";

    cout << "输入协议号:" << endl;
    cin >> proid;
    pro = getprotobynumber(proid);
    if (pro != NULL)
        cout << proid << "的协议名是" << pro->p_name << endl;
    else
        cout << proid << "协议号没有对应的协议名";

    //注销Winsock DLL
    WSACleanup();
    return 0;
}

```

## 4.5.4套接字选项和I/O控制命令

- **TCP/IP**协议的开发者考虑了可以满足大多数应用程序的默认行为，简化了开发的工作。
- 套接字的默认行为是可以被改变的。
- 通过设置套接字选项或**I/O**控制命令可以操作套接字的属性。
- 有些套接字选项仅仅是返回信息，还有些选项可以影响套接字的默认行为。

# 套接字选项

- 获取和设置套接字的函数分别是 `getsockopt()` 和 `setsockopt()`



# getsockopt

```
int getsockopt(  
    SOCKET s, //套接字句柄  
    int level, //选项被定义的级别,常用选项级别见表4-6, 更多  
    int optname, //选项名称, 见表4-6  
    char *optval, //缓冲区, 存储所请求的选项的值  
    int *optlen //缓冲区大小  
);
```

函数成功返回0，否则返回SOCKET\_ERROR  
[详情](#)

# setsockopt

```
int setsockopt(  
    SOCKET      s,    //套接字句柄  
    int         level, //选项被定义的级别,常用选项级别见表4-6, 更多  
    int         optname, //选项名称, 见表4-6  
    const char *optval, //缓冲区, 存储所请求的选项的值  
    int         optlen  //缓冲区大小  
);
```

函数成功返回0, 否则返回SOCKET\_ERROR  
[详情](#)

# 常见的套接字选项

- **SOL\_SOCKET**级别
  - **SO\_BROADCAST(BOOL)**:套接字是否可传输和接收广播信息（对**SOCK\_STREAM**类型套接字无效）
  - **SO\_DONTROUTE(BOOL)**:忽略路由表，直接发送数据到套接字绑定的接口，**windows**平台忽略此选项
  - **SO\_KEEPALIVE(BOOL)**:面向连接的套接字允许发送Keep\_Alive探测包
  - **SO\_LINGER(LINGER)**:如果存在尚未发送的数据，延迟close返回
  - **SO\_REUSEADDR(BOOL)**:套接字可以绑定到一个已经被另一个套接字使用的地址，或是绑定到一个处于**TIME\_WAIT**状态的地址。但是两个不同的套接字不能绑定到相同的本地地址去监听到来的连接。

# 常见的套接字选项

- SOL\_SOCKET级别
  - SO\_EXCLUSIVEADDRUSE(BOOL):套接字绑定到的本地端口不能被其他进程重用。
  - SO\_RCVBUF(int):套接字内部为接收操作分配的缓冲区的大小
  - SO\_SNDBUF(int):套接字内部为发送操作分配的缓冲区的大小
  - SO\_RCVTIMEO(DWORD):套接字上接收数据段的超时时间(以毫秒为单位)
  - SO\_SNDTIMEO(DWORD):套接字上发送数据段的超时时间(以毫秒为单位)

# 常见的套接字选项

- IPPROTO\_IP级别
  - IP\_OPTIONS(char[]):IP首部中的IP选项
  - IP\_HDRINCL(BOOL):发送函数在数据构造时包含IP首部，该选项仅对SOCK\_RAW类型的套接字有效。
  - IP\_TTL(int):IP首部中的TTL参数
  - IP\_MULTICAST\_LOOP(BOOL):是否允许多播套接字接收它发送的分组

# 常见的套接字选项

- IPPROTO\_TCP级别
  - TCP\_NODELAY(BOOL):是允许用于Nagle算法的延迟

# 常见的套接字选项

- IPPROTO\_IPv6级别
  - IPv6\_V6ONLY(BOOL):套接字只能进行IPv6通信
  - IPv6\_UNICAST\_HOPS(int):单播分组的TTL
  - IPv6\_MULTICAST\_HOPS(int):多播分组的TTL
  - IPv6\_MULTICAST\_LOOP(BOOL):是否允许套接字接收它发送的多播分组

# I/O控制命令

- I/O控制命令用来控制套接字上I/O的行为，也可以用来获取套接字上未决的I/O信息。
- 向套接字发送I/O控制命令的函数有两个：一个源于WinSock1.1的`ioctlsocket()`，另一个是WinSock2引进的`WSAIoctl()`。



# ioctlsocket

```
int ioctlsocket(  
    SOCKET s,    //套接字句柄  
    long cmd,    //套接字命令, 详情  
    u_long *argp //指向命令参数的指针  
);
```

成功执行返回0，否则返回SOCKET\_ERROR  
[详情](#)

# WSAIoctl

```
int WINAPI WSAIoctl(  
    SOCKET          s,           //套接字句柄  
    DWORD           dwIoControlCode, //操作的控制代码  
    LPVOID          lpvInBuffer,  //指向输入缓冲区  
    DWORD           cbInBuffer,   //输入缓冲区大小  
    LPVOID          lpvOutBuffer, //指向输出缓冲区  
    DWORD           cbOutBuffer,  //输出缓冲区大小  
    LPDWORD         lpcbBytesReturned, //输出参数，返回输出实际字节数的地址  
    LPWSAOVERLAPPED lpOverlapped, //指向WSAOVERLAPPED的指针  
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine //指向操作结束后调用的例  
);                               程指针
```

最后两个参数在使用重叠I/O时才用

成功执行返回0，否则返回SOCKET\_ERROR

[详情](#)

# 常用的I/O控制命令

- **FIONBIO**:启动或关闭套接字上的非堵塞模式
- **FIONREAD**:返回在套接字上要读的数据量的大小
- **SIOCATMARK**:确定带外数据（紧急数据）是否可读
- **SIO\_RCVALL**:接收网络上所有的封包

## 3.5.5 WinSock的错误处理

- WinSock函数在执行结束时都会返回一个值，如果函数执行成功，需要返回函数执行结果的函数通常返回值就是执行结果（该结果一般是用户程序所需要的），而对无执行结果的函数，例如WSACleanup ()函数，则返回0；
- Windows Sockets函数的错误可能有以下几种情况
  - 返回SOCKET\_ERROR(-1)指示发生了错误
  - 返回INVALID\_SOCKET(0xffff)指示发生了错误
  - 返回NULL指示发生了错误
- 虽然通过该返回值可以知道函数调用不成功，但无法判断函数不能成功执行的原因，WinSock提供了一个函数解决该问题。

# WSAGetLastError

- 使用WSAGetLastError（）函数可获取上一次WinSock函数调用时的错误代码。该函数的函数原型如下：

**int WSAGetLastError（void）；**

- 函数的返回值是一个整数，它是上一次调用WinSock函数出错时所出错误对应的错误代码。

- 由于引起WinSock函数调用出错的原因很多，为了便于编写出界面友好的应用程序和方便编程者调试程序，WinSock规范列举出了所有的出错原因并给每一种原因定义了一个整数类型（int）的编号，该编号被称作错误码。
- 在头文件Winsok2.h（Winsock1.1对应的头文件是winsock.h）中对每一个错误码都定义了一个对应的符号常量。
- 例如，当客户程序使用流式套接字试图与远程服务器建立连接，而远程服务器并没有响应（可能是服务其软件没有启动），这时客户程序中的connect（）函数调用将会出错，其错误码为10061，对应的符号常量为WSAECONNREFUSED。
- [错误码详细](#)

## 3.6 MFC中使用WinSock

- 创建对话框模式的项目时选择” **windows**”套接字
- 无需使用WSAStartup和WSACleanup。

# MFC 应用程序

## 高级功能选项

应用程序类型

文档模板属性

用户界面功能

高级功能

生成的类

高级功能:

- ☐ 打印和打印预览(P)
- ☐ 自动化(U)
- ☐ ActiveX 控件(R)
- ☐ MAPI (Messaging API)(I)
- ☒ Windows 套接字(W)
- ☐ Active Accessibility(A)
- ☐ 公共控件清单(M)
- ☐ 支持重启管理器(G)
- ☐ 重新打开以前打开的文档(Y)
- ☐ 支持应用程序恢复(V)

高级框架窗格:

- ☐ 资源管理器停靠窗格(D)
- ☐ 输出停靠窗格(O)
- ☐ 属性停靠窗格(S)
- ☐ 导航窗格(I)
- ☐ 标题栏(B)
- ☐ 高级帧菜单项显示/激活窗格(F)

最近文件列表上的文件数(N)

4

上一步

下一步

完成

取消



## 在CWinApp::InitInstance 的重载函数中调用AfxSocketInit

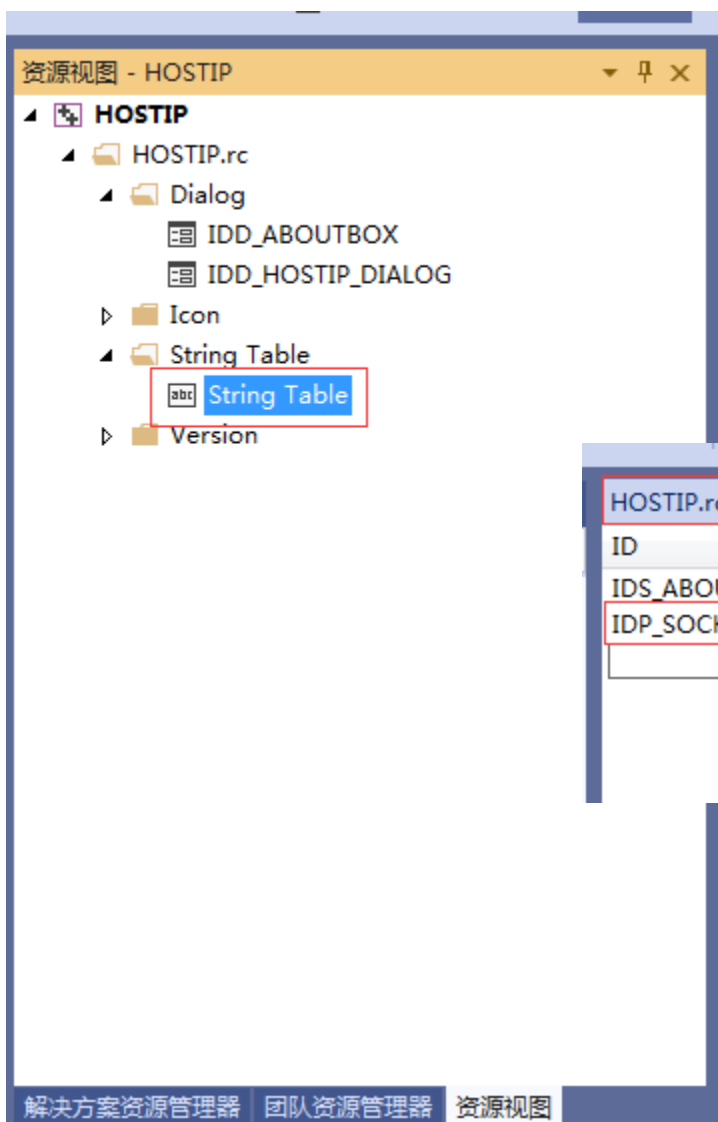
```
HOSTIP.cpp  X
HOSTIP  (全局范围)

31 // 唯一的 CHOSTIPApp 对象
32
33 CHOSTIPApp theApp;
34
35
36 // CHOSTIPApp 初始化
37
38 BOOL CHOSTIPApp::InitInstance()
39 {
40     CWinApp::InitInstance();
41
42     if (!AfxSocketInit())
43     {
44         AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
45         return FALSE;
46     }
47
48
49 // 创建 shell 管理器，以防对话框包含
50 // 任何 shell 树视图控件或 shell 列表视图控件。
51 CShellManager *pShellManager = new CShellManager;
52
53 // 激活“Windows Native”视觉管理器，以便在 MFC 控件中启用主题
54 CMFCVisualManager::SetDefaultManager(RUNTIME_CLASS(CMFCVisualManagerWindows));
55
56 // 标准初始化
57 // 如果未使用这些功能并希望减小
58 // 最终可执行文件的大小，则应移除下列
59 // 不需要的特定初始化例程
60 // 更改用于存储设置的注册表项
61 // TODO: 应适当修改该字符串，
62 // 例如修改为公司或组织名
63 SetRegistryKey(_T("应用程序向导生成的本地应用程序"));
64
65 CHOSTIPDlg dlg;
66 m_pMainWnd = &dlg;
67 INT_PTR nResponse = dlg.DoModal();
```

Resource.h X HOSTIP.cpp

HOSTIP

```
1  //{{NO_DEPENDENCIES}}
2  // Microsoft Visual C++ 生成的包含文件。
3  // 由 HOSTIP.rc 使用
4  //
5  #define IDR_MAINFRAME 128
6  #define IDM_ABOUTBOX 0x0010
7  #define IDD_ABOUTBOX 100
8  #define IDS_ABOUTBOX 101
9  #define IDD_HOSTIP_DIALOG 102
10 #define IDP_SOCKETS_INIT_FAILED 103
11
12 // 新对象的下一组默认值
13 //
14 #ifdef APSTUDIO_INVOKED
15     #ifndef APSTUDIO_READONLY_SYMBOLS
16
17         #define _APS_NEXT_RESOURCE_VALUE 129
18         #define _APS_NEXT_CONTROL_VALUE 1000
19         #define _APS_NEXT_SYMED_VALUE 101
20         #define _APS_NEXT_COMMAND_VALUE 32771
21     #endif
22 #endif
23
```



HOSTIP.rc - String Table targetver.h		
ID	值	标题
IDS_ABOUTBOX	101	关于 HOSTIP(&A)...
IDP_SOCKETS_INIT_FAILED	103	Windows 套接字初始化失败。

# 例题

- 编写程序查询本机的主机名称及**IP**地址。程序的运行界面如图所示。



- ①使用“MFC应用程序向导”创建一个基于对话框的一个应用程序框架；
- ②为静态文本框和列表框添加控件变量；

```
void CHOSTIPDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialogEx::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_HOST_NAME, m_sName);
    DDX_Control(pDX, IDC_LIST1, m_IPList);
}
```

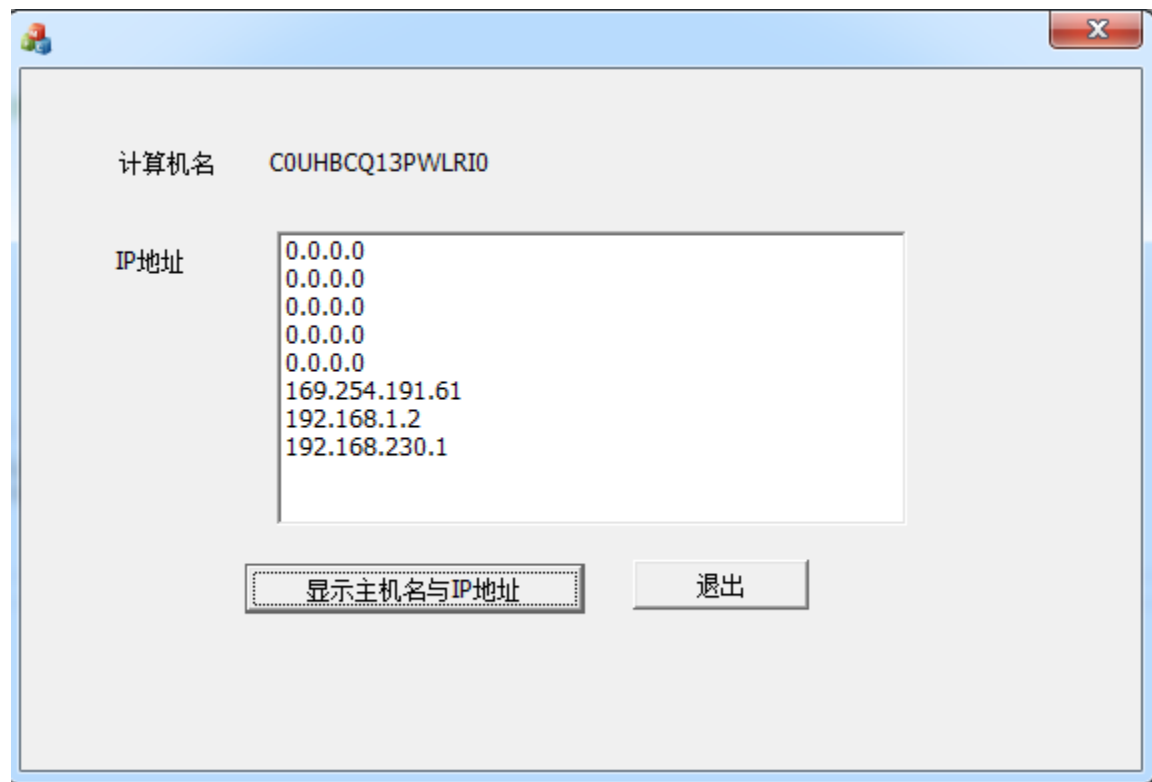
添加命令按钮的消息处理函数。为“确定”按钮的“BN\_CLICKED”消息添加处理函数。

```
void CHOSTIPDlg::OnBnClickedButton1()
{
    // TODO: 在此添加控件通知处理程序代码
    char hostname[32];
    if (gethostname(hostname, sizeof(hostname)))
    {
        m_sName.SetWindowText(_T("gethostname calling error\n"));
    }
    else
    {
        CString a(hostname);
        m_sName.SetWindowText(a);

        int itemCount = m_IPList.GetCount(); //获取列表框控件中内容的条数
        //清空列表框
        for (int i = 0; i < itemCount; i++)
            m_IPList.DeleteString(0);

        //获取地址填充下列表框
        struct addrinfo* result = NULL;
        struct addrinfo* ptr = NULL;
        char strIP[16];

        if (getaddrinfo(hostname, "", NULL, &result)) return;
        for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
            inet_ntop(AF_INET, &(((sockaddr_in*)(ptr->ai_addr))->sin_addr), strIP, 16);
            m_IPList.AddString(strIP);
        }
    }
}
```



# Win sdk网卡信息获取

- 在windows sdk中，用 **IP\_ADAPTER\_ADDRESSES** 结构体存储网卡信息，包括网卡名、网卡描述、网卡MAC地址、网卡IP等
- 调用 **GetAdaptersAddresses** 函数来获取相关网卡信息





```

typedef struct _IP_ADAPTER_ADDRESSES_LH {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            IF_INDEX IfIndex;
        };
    };
    struct _IP_ADAPTER_ADDRESSES_LH *Next;
    PCHAR AdapterName;
    PIP_ADAPTER_UNICAST_ADDRESS_LH FirstUnicastAddress; //单播地址
    PIP_ADAPTER_ANYCAST_ADDRESS_XP FirstAnycastAddress;
    PIP_ADAPTER_MULTICAST_ADDRESS_XP FirstMulticastAddress;
    PIP_ADAPTER_DNS_SERVER_ADDRESS_XP FirstDnsServerAddress;
    PWCHAR DnsSuffix;
    PWCHAR Description;
    PWCHAR FriendlyName;
    BYTE PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
    ULONG PhysicalAddressLength;
    union {
        ULONG Flags;
        struct {
            ULONG DdnsEnabled : 1;
            ULONG RegisterAdapterSuffix : 1;
            ULONG Dhcpv4Enabled : 1;
            ULONG ReceiveOnly : 1;
            ULONG NoMulticast : 1;
            ULONG Ipv6OtherStatefulConfig : 1;
            ULONG NetbiosOverTcpipEnabled : 1;
            ULONG Ipv4Enabled : 1;
            ULONG Ipv6Enabled : 1;
            ULONG Ipv6ManagedAddressConfigurationSupported : 1;
        };
    };
};

```

```

ULONG                Mtu;
IFTYPE               IfType;
IF_OPER_STATUS       OperStatus;
IF_INDEX             Ipv6IfIndex;
ULONG               ZoneIndices[16];
PIP_ADAPTER_PREFIX_XP FirstPrefix; //地址前缀
ULONG64             TransmitLinkSpeed;
ULONG64             ReceiveLinkSpeed;
PIP_ADAPTER_WINS_SERVER_ADDRESS_LH FirstWinsServerAddress;
PIP_ADAPTER_GATEWAY_ADDRESS_LH    FirstGatewayAddress;
ULONG                Ipv4Metric;
ULONG                Ipv6Metric;
IF_LUID              Luid;
SOCKET_ADDRESS       Dhcpv4Server;
NET_IF_COMPARTMENT_ID CompartmentId;
NET_IF_NETWORK_GUID  NetworkGuid;
NET_IF_CONNECTION_TYPE ConnectionType;
TUNNEL_TYPE          TunnelType;
SOCKET_ADDRESS       Dhcpv6Server;
BYTE                 Dhcpv6ClientDuid[MAX_DHCPV6_DUID_LENGTH];
ULONG                Dhcpv6ClientDuidLength;
ULONG                Dhcpv6Iaid;
PIP_ADAPTER_DNS_SUFFIX FirstDnsSuffix;
} IP_ADAPTER_ADDRESSES_LH, *PIP_ADAPTER_ADDRESSES_LH;

```

[详细](#)

```

typedef struct _IP_ADAPTER_ADDRESSES_XP {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD IfIndex;
        };
    };
    struct _IP_ADAPTER_ADDRESSES_XP *Next;
    PCHAR AdapterName;
    PIP_ADAPTER_UNICAST_ADDRESS_XP FirstUnicastAddress;
    PIP_ADAPTER_ANYCAST_ADDRESS_XP FirstAnycastAddress;
    PIP_ADAPTER_MULTICAST_ADDRESS_XP FirstMulticastAddress;
    PIP_ADAPTER_DNS_SERVER_ADDRESS_XP FirstDnsServerAddress;
    PWCHAR DnsSuffix;
    PWCHAR Description;
    PWCHAR FriendlyName;
    BYTE PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD PhysicalAddressLength;
    DWORD Flags;
    DWORD Mtu;
    DWORD IfType;
    IF_OPER_STATUS OperStatus;
    DWORD Ipv6IfIndex;
    DWORD ZoneIndices[16];
    PIP_ADAPTER_PREFIX_XP FirstPrefix;
} IP_ADAPTER_ADDRESSES_XP, *PIP_ADAPTER_ADDRESSES_XP;

```

[详细](#)

```

:
-#if (NTDDI_VERSION >= NTDDI_VISTA)
typedef IP_ADAPTER_UNICAST_ADDRESS_LH IP_ADAPTER_UNICAST_ADDRESS;
typedef IP_ADAPTER_UNICAST_ADDRESS_LH *PIP_ADAPTER_UNICAST_ADDRESS;
-#elif (NTDDI_VERSION >= NTDDI_WINXP)
typedef IP_ADAPTER_UNICAST_ADDRESS_XP IP_ADAPTER_UNICAST_ADDRESS;
typedef IP_ADAPTER_UNICAST_ADDRESS_XP *PIP_ADAPTER_UNICAST_ADDRESS;
#endif
:

```

单播地址结构的定义

```

typedef struct _IP_ADAPTER_UNICAST_ADDRESS_LH {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD Flags;
        };
    };
};

struct _IP_ADAPTER_UNICAST_ADDRESS_LH *Next;
SOCKET_ADDRESS Address; //IP地址
IP_PREFIX_ORIGIN PrefixOrigin; //网络地址部分来源（枚举类型）
IP_SUFFIX_ORIGIN SuffixOrigin; //主机地址部分来源(枚举类型)
IP_DAD_STATE DadState; //地址冲突检测状态
ULONG ValidLifetime;
ULONG PreferredLifetime;
ULONG LeaseLifetime;
UINT8 OnLinkPrefixLength; //掩码长度
} IP_ADAPTER_UNICAST_ADDRESS_LH, *PIP_ADAPTER_UNICAST_ADDRESS_LH;

```

[详细](#)

```

typedef struct _IP_ADAPTER_UNICAST_ADDRESS_XP {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD Flags;
        };
    };
    struct _IP_ADAPTER_UNICAST_ADDRESS_XP *Next;
    SOCKET_ADDRESS Address;
    IP_PREFIX_ORIGIN PrefixOrigin;
    IP_SUFFIX_ORIGIN SuffixOrigin;
    IP_DAD_STATE DadState;
    ULONG ValidLifetime;
    ULONG PreferredLifetime;
    ULONG LeaseLifetime;
} IP_ADAPTER_UNICAST_ADDRESS_XP, *PIP_ADAPTER_UNICAST_ADDRESS_XP;

```

# SOCKET\_ADDRESS

```
typedef struct _SOCKET_ADDRESS {  
    LPSOCKADDR lpSockaddr;  
    INT         iSockaddrLength;  
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, *LPSOCKET_ADDRESS;
```



```
IPHLPAPI_DLL_LINKAGE ULONG GetAdaptersAddresses(  
    ULONG                Family,  
    ULONG                Flags,  
    PVOID                Reserved,  
    PIP_ADAPTER_ADDRESSES AdapterAddresses,  
    PULONG               SizePointer  
);
```

[详细](#)

# 示例

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <stdio.h>
#include <stdlib.h>

// Link with Iphlpapi.lib
#pragma comment(lib, "IPHLPAPI.lib")
#pragma comment(lib, "ws2_32.lib") //加载库文件“ws2_32.lib”

#define MAX_TRIES 3

int main()
{
    PIP_ADAPTER_ADDRESSES pAddresses = NULL;
    PIP_ADAPTER_ADDRESSES pCurrAddresses = NULL;
    PIP_ADAPTER_UNICAST_ADDRESS pUnicast = NULL; //单播地址
    ULONG outBufLen = 0;
    DWORD dwRetVal = 0;
    ULONG Iterations = 0;
    unsigned int i = 0;
    char strIP[16];
```

```

do {
    pAddresses = (IP_ADAPTER_ADDRESSES*)HeapAlloc(GetProcessHeap(), 0, outBufLen);
    if (pAddresses == NULL) {
        printf("Memory allocation failed for IP_ADAPTER_ADDRESSES struct\n");
        exit(1);
    }

    dwRetVal = GetAdaptersAddresses(AF_INET, 0, NULL, pAddresses, &outBufLen);
    if (dwRetVal == ERROR_BUFFER_OVERFLOW) {
        HeapFree(GetProcessHeap(), 0, pAddresses);
        pAddresses = NULL;
    }
    else {
        break;
    }
    Iterations++;
} while ((dwRetVal == ERROR_BUFFER_OVERFLOW) && (Iterations < MAX_TRIES));

```

正确的读取适配器信息的姿势

GetAdaptersAddress调用失败时，会自动更新outBufLen的值，使得下次调用成功

```

if (dwRetVal == NO_ERROR) {
    pCurrAddresses = pAddresses;
    while (pCurrAddresses) {
        printf("\t接口序号 (IPv4 接口): %u\n", pCurrAddresses->IfIndex);
        printf("\t接口内部名称: %s\n", pCurrAddresses->AdapterName);
        printf("\t接口显示名称: %wS", pCurrAddresses->FriendlyName);
        printf("\n");
        printf("\t接口描述: %wS\n", pCurrAddresses->Description);

        pUnicast = pCurrAddresses->FirstUnicastAddress;
        if (pUnicast != NULL) {
            printf("\tIP地址:\n");
            for (i = 0; pUnicast != NULL; i++) {
                sockaddr_in sa = *(sockaddr_in*) (pUnicast->Address.lpSockaddr);
                inet_ntop(AF_INET, &sa.sin_addr, strIP, 16);
                printf("\t\t地址: %s\n", strIP);
                printf("\t\t掩码长度: %d\n", pUnicast->OnLinkPrefixLength);
                pUnicast = pUnicast->Next;
            }
            printf("\t单播地址个数: %d\n", i);
        }
        else
            printf("\t没有单播地址\n");

        printf("\n");

        pCurrAddresses = pCurrAddresses->Next; //下一个接口
    }
}

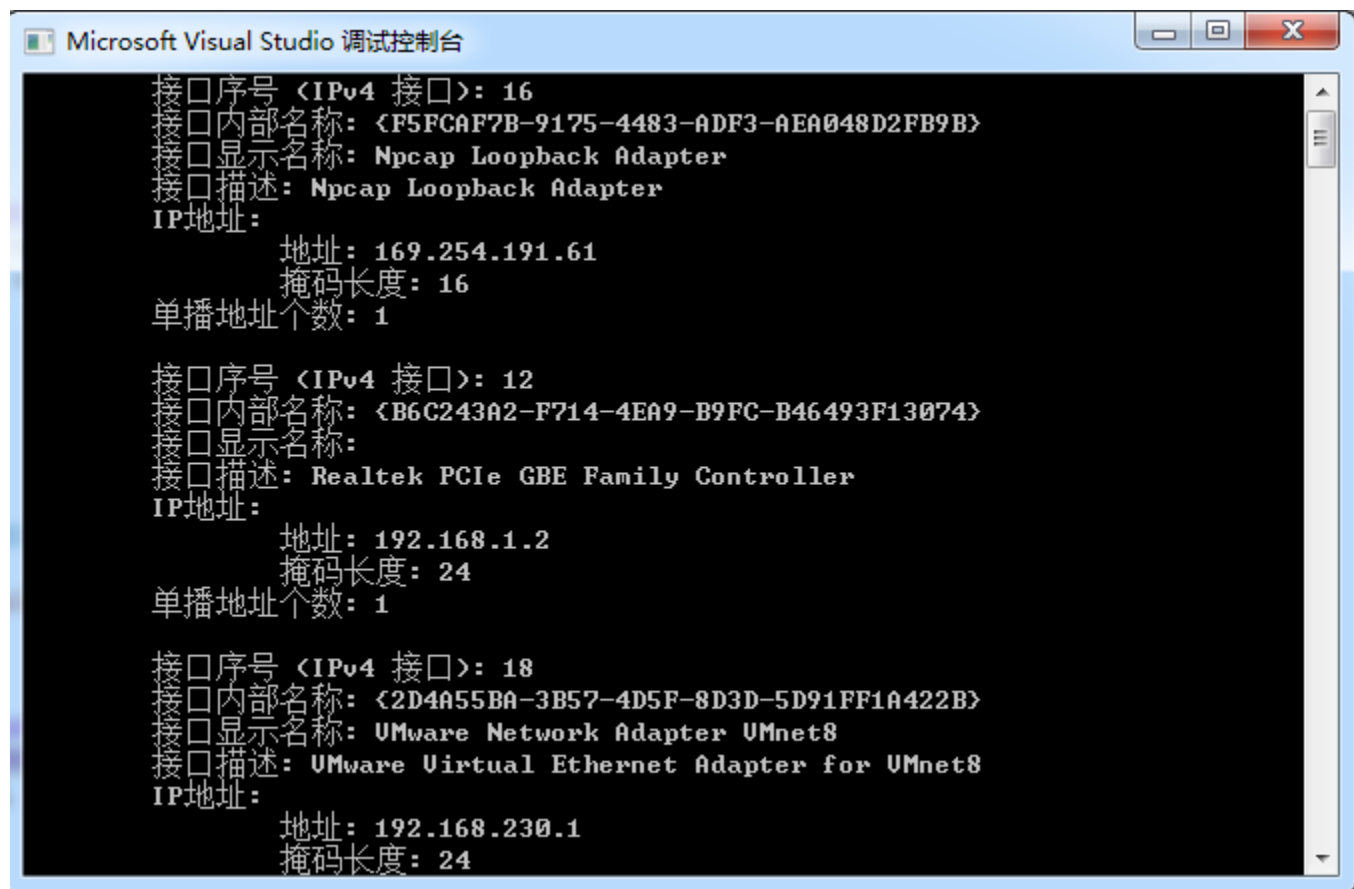
```

读取适配器上的各种信息

```
if (pAddresses) {  
    HeapFree(GetProcessHeap(), 0, pAddresses);  
}  
return 0;
```

最后不要忘了释放分配的堆栈空间，避免内存泄漏

# 执行效果



```
Microsoft Visual Studio 调试控制台

接口序号 <IPv4 接口>: 16
接口内部名称: <F5FCAF7B-9175-4483-ADF3-AEA048D2FB9B>
接口显示名称: Npcap Loopback Adapter
接口描述: Npcap Loopback Adapter
IP地址:
    地址: 169.254.191.61
    掩码长度: 16
单播地址个数: 1

接口序号 <IPv4 接口>: 12
接口内部名称: <B6C243A2-F714-4EA9-B9FC-B46493F13074>
接口显示名称:
接口描述: Realtek PCIe GBE Family Controller
IP地址:
    地址: 192.168.1.2
    掩码长度: 24
单播地址个数: 1

接口序号 <IPv4 接口>: 18
接口内部名称: <2D4A55BA-3B57-4D5F-8D3D-5D91FF1A422B>
接口显示名称: VMware Network Adapter VMnet8
接口描述: VMware Virtual Ethernet Adapter for VMnet8
IP地址:
    地址: 192.168.230.1
    掩码长度: 24
```