WINDOW WINSOCK 网　络　程　序　设　计

主要参考书目：

《TCP/IP 详解：卷一　协议》

《TCP/IP 详解：卷二　实现》

W.Richard Stevens 著　机械工业出版社

主要内容：
总体上由三大模块组成：

第一部分：网络程序设计技术（基础）

网络程序设计技术（基础），讨论实用的网络程序设计技术。不涉及高级复杂的网络设计技术。
分为两个子模块：

1）基于 Linux Socket(BSD Socket)的网络程序设计技术（Linux/Unix 平台）

2）基于 WinSock 2 的（Ws2_32.dll）网络程序设计技术（WIN32 平台）

（基于 TCP/IP 的 NetBios 协议的网络程序设计技术：不讨论）

第二部分：　协议与实现

对主要协议进行理论上的深入分析，并讨论其实现的技术。
分为：
一、传输层协议

RFC 793 TCP 协议（Transmission Control Protocol）

RFC 768 UDP 协议（User Datagram Protocol）

RFC 322 著名端口号规定（Well Known Socket Numbers ）

(TCP、UDP port numbers　最新 2003/9/15)

二、IP 层及以下

RFC 791 IP 协议（Internet Protocol）

RFC 826 ARP 协议（Address Resolution Protocol）

RFC 903 RARP 协议（A Reverse Address Resolution Protocol）

RFC 792 ICMP 协议（Internet Control Message Protocol）

RFC 1112 IGMP 协议（Internet Group Management Protocol）

RFC 1918 LNA 规定（Local Network Address）

三、应用层协议　或　其它的相关协议

RFC 2068 HTTP/1.1

RFC 1808、1738 URL 协议

RFC 959 FTP 协议

RFC 821 SMTP 协议

RFC 1939 POP3 协议

RFC 2060 IMAP 协议

RFC 1521 MIME 协议

RFC 854 Telnet 协议

RFC 1034、1035 DNS 域名服务协议

RFC 1996、2136、2137 动态 DNS（DDNS）协议

代理相关：

RFC 1928 SOCKSV5 SOCKSV5 代理协议

实时流相关：

RFC 3267、3389 、1889 RTP 协议

RFC 2326 RTSP 协议

与路由相关：

RFC 2328 OSPF V2

RFC 1721、1722、1723、1724 RIP V2

其它：

RFC 1534、1497、1084 BOOTP 协议

RFC 2132、3495 DHCP 协议

RFC 1661 PPP 协议

等

第三部分 网络程序设计技术（高级）

讨论 Socket 的复杂特性的技术

分为四个部分：

一、SOCKET 方面

1）基于 IP 层的 Socket 程序设计技术（IP、ICMP 及其它，特定包捕获技术）

2）基于数据链路层的 Socket 程序设计技术（ARP、RARP 及其它。基于帧的包捕获技术）

3）高级 Socket: (1) 多路利用 （2）非阻塞 socket

(3)信号驱动 I/O （4）广播与组播

二、基于 libpcap 库的网络分析技术、网络包捕获技术、视频会议技术（Linux/Unix 平台）

三、基于 winpcap 库的网络分析技术、网络包捕获技术、视频会议技术（Wwin32 平台）

四、代理报务程序设计、实时流媒体程序设计

五、RPC 程序设计

六、分布式组件程序设计（DCOM、CORBA、EJB）

七、P2P 网络程序设计

第一部分：网络程序设计技术（基础）

0、网络程序设计的基本概念

1）网络及其结构
2）协议、网络体系结构
3）TCP/IP 的发展简史
4）TCP/IP 结构
5）网络程序设计中的基本概念：网络标识（协议、地址、端口号）、Socket（TCP、UDP、RAW）
6）网络程序中的多进程与多线程（服务器方的工作模型）
7）网络程序设计型：C/S、B/S、多层 B/S

附：端口号分配表

著名（TCP、UDP）端口号列表：

端口号分三类： 著名端口号；已注册端口号；动态或私用端口号。

著名端口号：0 到 1023.（通常要求有超级用户的权限）

已注册端口号：1024 到 49151

动态或私用端口号：49152 到 65535

一、基于 WinSock2 的网络程序设计技术（Windows 平台）

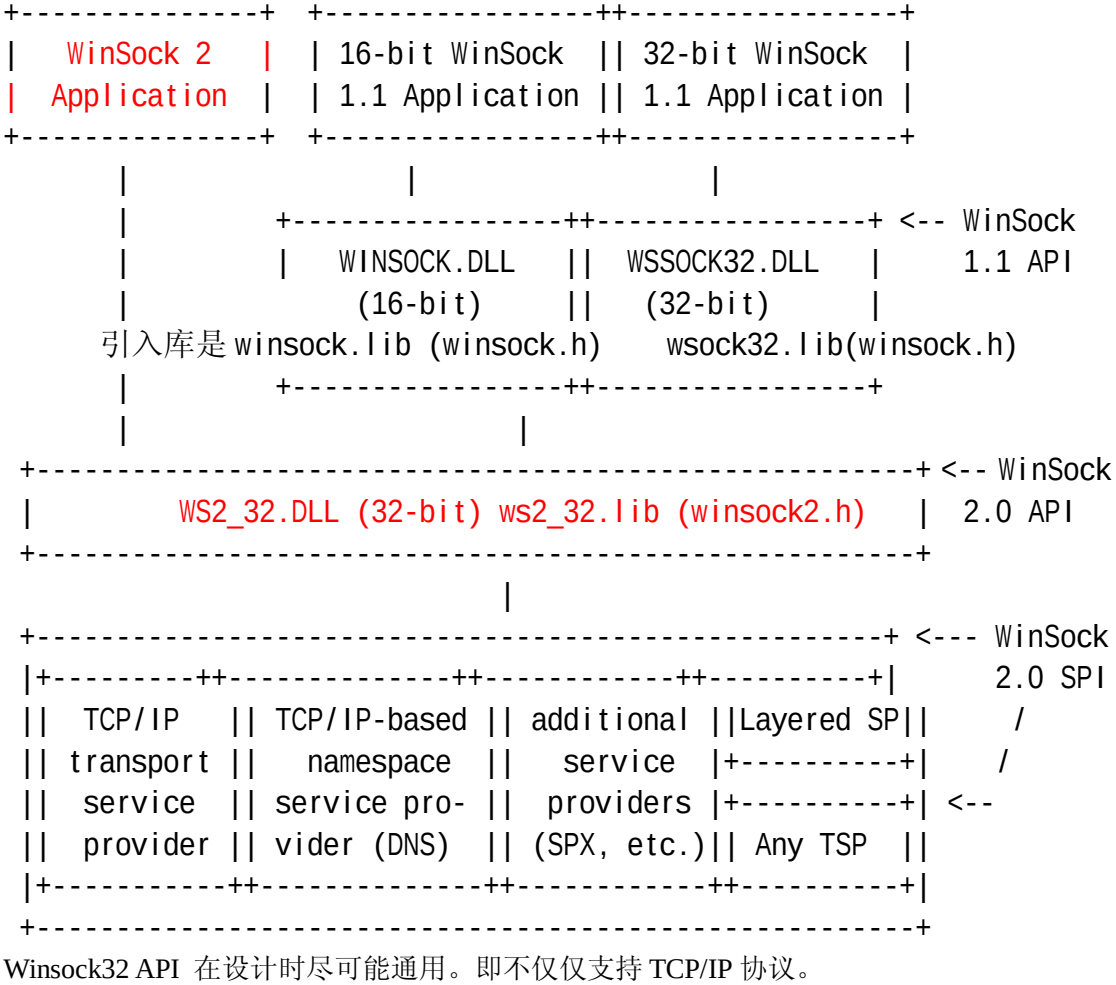
2．1 Winsock32 概述

Windows Sockets 规范(简称 "Winsock" 或 "WinSock")定义了 WIN 平台上的网络编程接口，该接口基于 BSD Unix 中普遍使用的“socket”网络编程接口模式。既提供了类似 Berkeley socket 风格的函数，也有面向 WIN 平台自身特定的扩展。如：

1. Winsock 1 应用程序可以要求 WinSock 以 window 消息形式进行通知。这样应用程序能方便并发处理网络消息，UI 消息，及后台处理数据等.

2. Winsock 2.x 定义了两套网络编程接口：应用编程接口(API) 和服务提供商接口(SPI).

1993 年 1 月，自 WinSock1.1 发布后即成为 win 平台的标准。



Winsock32 API 在设计时尽可能通用。即不仅仅支持 TCP/IP 协议。

2.2 Winsock2 需要的头文件与库

Winsock1 –16 位程序：winsock.h、winsock.lib、winsock.dll

Winsock1-32 位程序：winsock.h、wsock32.lib、wssock32.dll(Window CE)

Winsock2-32 位程序：winsock2.h、ws2_32.lib、ws2_32.dll

高性能的专用于 Windows 平台的网络程序：MSWSOCK.H、MSWSOCK.LIBb、MSWSOCK.DLL

2.3 在 VC 中的配置

1）Winsock.h 路径配置
Tools->Options->Directories，在 Show Directories for:下拉列表框中：选：Include files
查看是否有：<VS 的 Home 目录>\VC98\Include

2）Ws2_32.lib 路径配置
Tools->Options->Directories，在 Show Directories for:下拉列表框中：选：Library files
查看是否有：<VS 的 Home 目录>\VC98\Lib

3）在项目中增加 Ws2_32.lib 连接库
在创建一个项目后，Project->Setting->Link 卡片：
在 Object/Library Modules 下：添加：ws2_32.lib 即可。

4）简单测试。
输入如下代码，简单测试配置是否正确。

```
#include "winsock2.h"
#include "stdio.h"

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
{
   WSADATA wsaData;

   int r = WSAStartup(MAKEWORD(2,2),&wsaData);
   if( r != 0 )
   {
      MessageBox(NULL,"Winsock2 装载出错了!","Winsock2 测试",0);

     return -1;
   }

   if ( LOBYTE( wsaData.wVersion ) != 2 ||
            HIBYTE( wsaData.wVersion ) != 2 )
   {

      MessageBox(NULL,"Winsock2 装载的版本不对！ ","Winsock2 测试",0);
      WSACleanup();
      return -1;
   }//if

    char data[80];
```

```
    sprintf(data,"WSAStartup 装 载 成 功 ！ 版 本 是 ： %d.%d
",HIBYTE( wsaData.wVersion ),LOBYTE( wsaData.wVersion ));
    MessageBox(NULL,data,"WSAStartup",0);
    WSACleanup();
    return 0;
}//winmain
```

2.4 WinSock 的初始化

任何基于 Winsock 的程序，首先要将正确的 Winsock 的 DLL 装入到内存中。如：Winsock2 的应用程序，则要加载 ws2_32.dll 文件。

函数： int WSAAPI WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);[p103]

功能：加载相应的WinSock的dll版本。在使用WinSock函数前，若没有加载WinSock库，则函数就会返回一个SOCKET_ERROR，错误信息是WSANOTINITIALISED。
加载 Winsock 库是通过调用 WSAStartup 函数实现的。

参数：
1) WORD wVersionRequested [输入] 用于指定准备加载的Winsock的dll的版本，这个是调用者希望得到支持的最高的版本。高位字节指定所需要的Winsock 库的副版本，而低位字节则是主版本。然后，可用宏MAKEWORD(X,Y)（其中，x 是主版本，y 是副版本）方便地获得wVersionRequested 的正确值。如：winsock 2.2 版本，可以用：MAKEWORD（2，2）或 0x0202。

Win平台的版本情况：

平台	Winsock 版本
Windows 95	1.1 （2.2 ）
Windows 98	2.2
Windows NT 4.0	2.2
Windows 2000 及以上	2.2
Windows CE	1.1

2）lpWSAData [输出] 是指向LPWSADATA 结构的指针，目的是用于获取Winsock具体实现的更多的信息。定义如下：（在winsock2.h中定义）

```
typedef struct WSAData {
        WORD                    wVersion;
        WORD                    wHighVersion;
```

```
        char                    szDescription[WSADESCRIPTION_LEN+1];
        char                    szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short          iMaxSockets;
        unsigned short          iMaxUdpDg;
        char FAR *              lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

WSAStartup 用其加载的库版本有关的信息填在这个结构中：WSAStartup 把第一个字段wVersion 设成打算使用的Winsock 的版本（用户所请求的版本且winsock又支持该版本）。而wHighVersion 参数容纳的是现有的Winsock 库所能支持的最高版本（用户不一定请求使用该版本，尽管大部分情况下应该是相同的）。szDescription 是一个最多 256 个字符的以零结尾的串。用于说明该winsock的实现情况的说明。szSystemStatus 表明当前ws2_32.dll的工作状态。下面的三个字段在Winsock2 上都不要使用。

返回值：若成功，则返回 0.否则返回非零。

若程序运行结束，则在结束之前，一定要使用 int WSACleanup(void);来卸载所加载的 DLL 库。WSACleanup()调用运行时即使有错也没有特别大的关系。

[代码示例：testWSAStartup.cpp]

2.5 Winsock API 的错误检查与处理

对Winsock 函数来说，返回错误是非常常见的。多数情况下，这些错误都是无关紧要的，通信仍可在套接字上进行。尽管其返回的值并非一成不变，但不成功的Winsock 调用返回的最常见的值是SOCKET_ERROR 。
如果调用一个Winsock 函数，错误情况发生了，就可用WSAGetLastError 函数来获得一个代码，代码明确地表明发生的状况。该函数的定义见.h文件。定义如下：

```
int WSAGetLastError (void);
```

发生错误之后调用这个函数，就会返回最新的Winsock操作所发生的特定错误的代码。WSAGetLastError（）函数返回的这些错误都已预定义常量值，根据Winsock 版本的不同，这些值的声明不在Winsock.h 中，就会在Winsock 2.h 中。两个头字段的唯一差别是Winsock 2.h 中包含的错误代码（针对Winsock 2 中引入的一些新的API 函数和性能）更多。为各种错误代码定义的常量（带有#定义指令）一般都以WSAE 开头。（部分见.h文件）

注：一旦Winsock的函数表明发生了错误，则必须立即调用该函数。因为有些Winsock函数若操作成功会置最新的操作错误代码为 0.

[代码示例：TestWSAGetLastError.cpp]

常见错误代码及其含义见 Winsock2 规范中错误清单。

2.6 IPV4 地址的表示

IPV4 是一个 32 位的二进制数。在 Winsock2.h 中，定义了结构：struct sockaddr_in，表示 IPV4 的通信标识，而结构 struct in_addr 真正表示 IP 地址。

struct sockaddr_in 定义如下：

```
struct sockaddr_in
{
    short           sin_family; /* Address family          */
    u_short         sin_port; /* Port number               */
    struct in_addr  sin_addr; /* Internet address    v4    */
    char            sin_zero[8];
};
```

说明：该结构除了 sin_family 外，其它字段都是网络字节次序。

其中：

1）sin_family 可取值（表示协议家簇）：

```
    #define AF_UNIX         1               /* local to host (pipes, portals) */
    #define AF_INET         2               /* internetwork: UDP, TCP, etc. */
    #define AF_IMPLINK      3               /* arpanet imp addresses */
    #define AF_PUP          4               /* pup protocols: e.g. BSP */
    #define AF_CHAOS        5               /* mit CHAOS protocols */
    #define AF_NS           6               /* XEROX NS protocols */
    #define AF_IPX          AF_NS           /* IPX protocols: IPX, SPX, etc. */
    #define AF_ISO          7               /* ISO protocols */
    #define AF_OSI          AF_ISO          /* OSI is ISO */
    #define AF_ECMA         8               /* european computer manufacturers */
    #define AF_DATAKIT      9               /* datakit protocols */
    #define AF_CCITT        10              /* CCITT protocols, X.25 etc */
    #define AF_SNA          11              /* IBM SNA */
    #define AF_DECnet       12              /* DECnet */
    #define AF_DLI          13              /* Direct data link interface */
    #define AF_LAT          14              /* LAT */
```

```
    #define AF_HYLINK       15              /* NSC Hyperchannel */
    #define AF_APPLETALK    16              /* AppleTalk */
    #define AF_NETBIOS      17              /* NetBios-style addresses
*/
    #define AF_VOICEVIEW    18              /* VoiceView */
    #define AF_FIREFOX      19              /* Protocols from Firefox
*/
    #define AF_UNKNOWN1     20              /* Somebody is using this!
*/
    #define AF_BAN          21              /* Banyan */
    #define AF_ATM          22              /* Native ATM Services */
    #define AF_INET6        23              /* Internetwork Version 6
*/
    #define AF_CLUSTER      24              /* Microsoft Wolfpack */
    #define AF_12844        25              /* IEEE 1284.4 WG AF */


    #define AF_MAX          26
```

2）sin_port 是网络字节次序的端口号。

3）sin_addr 是网络字节次序的 IPV4 的 32 位地址表示

```
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
                                /* can be used for 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 */
};
```

如何将“192.168.0.200”点分十进制表示法转换成所要的结构？用如下方法：

unsigned long inet_addr(const char FAR *cp);

功能：将点分十进制表示法的 IP 地址转换成 32 位的二进制数。

如：

sin_addr.s_addr=inet_addr("192.168.0.200");

注：IP 地址转换函数：

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

功能：将“192.168.0.200”点分十进制表示法转换成 32 位的二进制数（IPV4）。

若转换成功，即是 IPV4。若转换不成功，则返回：**INADDR_NONE**。

参数：**cp:[**输入**]**。是以零结尾的点分十进制表示的字符串格式的 IPV4 。

返回值：合法的 IPV4 的 32 位二进制数。不合法返回：**INADDR_NONE**

说明：

CP 值	含义
"4.3.2.16"	十进制
"004.003.002.020"	八进制
"0x4.0x3.0x2.0x10"	十六进制
"4.003.002.0x10"	混合

2）char* FAR inet_ntoa(struct in_addr in);

功能：将 struct in_addr 所表示的 32 位的二进制数（IPV4）地址转换成如“192.168.0.200”这样的点分十进制表示法的 IPV4 地址。

参数：**in:[**输入**]** struct in_addr 所表示的 32 位的二进制数(IPV4)地址 。

返回值：若转换成功，返回点分十进制表示法的 IPV4 地址的字符串。若转换不成功，则返回：**NULL**。

说明：返回的串的内存空间是在 Winsock 中分配的。因此只能保证：在同一个线程中，当调用其它的 Winsock 函数之前，该返回串是有效的。若要一直使用该串，则应该将它的内容首先拷贝到自己的缓冲区中。

[代码示例：TestInet_Addr.cpp]

2.6.2 主机字节顺序与网络字节顺序

主机字节顺序：主机存储数据的顺序方式。

网络中存在多种类型的机器，如基于 Intel 芯片的 PC 机和基于 RISC 芯片的工

作站。这些不同类型的机器表示数据的字节顺序是不同的。设有一个 16 位的整数 A103 它由 2 个字节组成，高位字节是 AI，低位字节是 03。在内存中可以有两种方式来存储这个整数：

little-endian 方式： 低位字节存储在这个整数的开始地址位置，基于 Intel 芯片的机器采用的是这种方式。

big-endian 方式：高位字节存储在开始地址位置，大多数基于 RISC 芯片的机器采用的是这种方式。

网络协议中的数据只有采用统一的字节顺序，才能在不同类型的机器之间正确地发送和接收数据。Intemet 规定的网络字节顺序采用 big-endian 方式。例如 TCP 协议数据段中的 16 位端口号和 32 位 IP 地址就是使用网络字节顺序进行传送的。

4 个库函数来进行转换字节转换：winsock2.h 中定义（实质是在 winsock1 的.h 中定义）

u_long htonl (u_long hostlong);

u_short htons(u_short hostshort);

u_long ntohl(u_long netlong);

u_short ntohs(u_short netshort);

以上 4 个函数中 h 代表 host，n 代表 network，s 代表 short，l 代表 long。Short 是 16 位整数，long 是 32 位整数。前两个函数将主机字节顺序转换成网络字节顺序，而后两个函数则刚好相反。编程中，在需要使用网络字节顺序时，应该使用这几个函数来进行转换，绝对不要依赖于具体机器的表示方式。

[代码示例：testhtonx.cpp]

附：部分 Winsock 的类型定义（取自 winsock2.h）

```
typedef u_int           SOCKET;

#define INADDR_ANY              (u_long)0x00000000
#define INADDR_LOOPBACK         0x7f000001
#define INADDR_BROADCAST        (u_long)0xffffffff
```

```
#define INADDR_NONE             0xffffffff

#define ADDR_ANY                INADDR_ANY
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* up to 14 bytes of direct address */
};
typedef struct sockaddr_in SOCKADDR_IN;

#define SOCK_STREAM     1               /* stream socket */
#define SOCK_DGRAM      2               /* datagram socket */
#define SOCK_RAW        3               /* raw-protocol interface */
#define AF_UNIX         1               /* local to host (pipes, portals) */
#define AF_INET         2               /* internetwork: UDP, TCP, etc. */

#define MAKEWORD(low,high) \
        ((WORD)((BYTE)(low)) | (((WORD)(BYTE)(high))<<8)))

#define SOCKET_ERROR            (-1)

#define WSAAPI                  FAR PASCAL
/*
 * All Windows Sockets error constants are biased by WSABASEERR from
 * the "normal"
 */
#define WSABASEERR              10000
/*
 * Windows Sockets definitions of regular Microsoft C error constants
 */
#define WSAEINTR                (WSABASEERR+4)
#define WSAEBADF                (WSABASEERR+9)
#define WSAEACCES               (WSABASEERR+13)
#define WSAEFAULT               (WSABASEERR+14)
#define WSAEINVAL               (WSABASEERR+22)
#define WSAEMFILE               (WSABASEERR+24)

/*
 * Windows Sockets definitions of regular Berkeley error constants
 */
#define WSAEWOULDBLOCK          (WSABASEERR+35)
#define WSAEINPROGRESS          (WSABASEERR+36)
#define WSAEALREADY             (WSABASEERR+37)
#define WSAENOTSOCK             (WSABASEERR+38)
#define WSAEDESTADDRREQ         (WSABASEERR+39)
```

```
#define WSAEMSGSIZE             (WSABASEERR+40)
```

2.6.3 其它辅助的函数

1）获取主机名：gethostname()

```
int gethostname(
  char* name,
  int namelen
);
```

功能：返回本地主机的标准的主机名

参数：1）name:[输出]。存放输出的主机名。

2）namelen:输入。Name缓冲区的长度。

返回值：若成功，则返回0，否则返回：SOCKET_ERROR(-1)。

注：必须首先使用：WSAStartup()。

[代码示例：test_gethostname.cpp]

2)返回主机名对应的主机信息：gethosbyname()

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

功能：返回主机名对应的主机信息

参数： name:[输入]。要解析的主机名。若是NULL，则相当于：gethostname()。

返回值：若成功，则返回一个指向hostent的指针，否则返回：NULL。

注：1）必须首先使用：WSAStartup()。

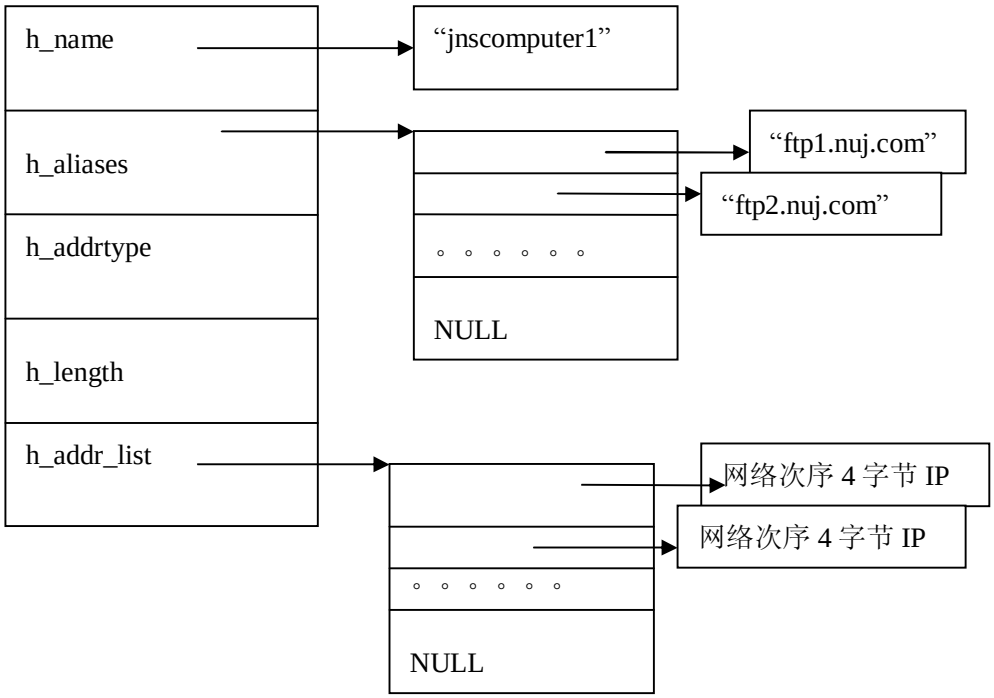
2)所返回的结构的空间是在Winsock中分配的。不允许释放该结构中任何部分的指针空间。且若要一直使用，则首先在调用其它winsock函数之前，将它的内容保存到自己的变量中。

3）点分十进制表示的IP地址的串，是不能解析成功的。应先使用inet_addr()将其转换成真正IP地址（struct in_addr），然后使用gethostbyaddr()转成hostent结构。

结构hostent定义如下：

```
typedef struct hostent {
  char FAR* h_name;
  char FAR  FAR** h_aliases;
  short h_addrtype;
  short h_length;
  char FAR  FAR** h_addr_list;
```

```
} hostent;
```



h_name

主机的正式名称。

h_aliases

主机别名列表。以 **Null** 结尾。

h_addrtype

地址类型如 **AF_INET** 代表 **IPV4.**

h_length

地址长度。

h_addr_list

主机的 **IP** 地址列表(**16** 进制串表示的 **32** 位 **IPV4** 地址)。以 **Null** 结尾。

[代码示例：test_gethostbyname.cpp]

若是对www.sina.com.cn进行处理，则运行结束如下：

```
正式主机名：jupiter.sina.com.cn
地址长度：4
```

地址类型：2
主机的别名列表如下：
www.sina.com.cn
主机的IP列表如下：
61.172.201.194
Press any key to continue

3）返回对应一个IP网络地址的主机的信息：gethostbyaddr()

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

功能：返回对应一个IP网络地址的主机的信息。
参数：addr[输入]网络字节次序的 32 位的IP地址。
len:[输入]addr的长度
type:[输入]地址类型。如：AF_INET
返回值：成功则返回指向hostent结构的指针。不成功则返回NULL。
注：1）必须首先使用：WSAStartup()。
2)所返回的结构的空间是在Winsock中分配的。不允许释放该结构中任何部分的指针空间。且若要一直使用，则首先在调用其它winsock函数之前，将它的内容保存到自己的变量中。

[代码示例：test_gethostbyaddr.cpp]

4)缓冲区清零：memset()
定义：在string.h或memory.h中定义

```
void *memset( void *dest, int c, size_t count );
```

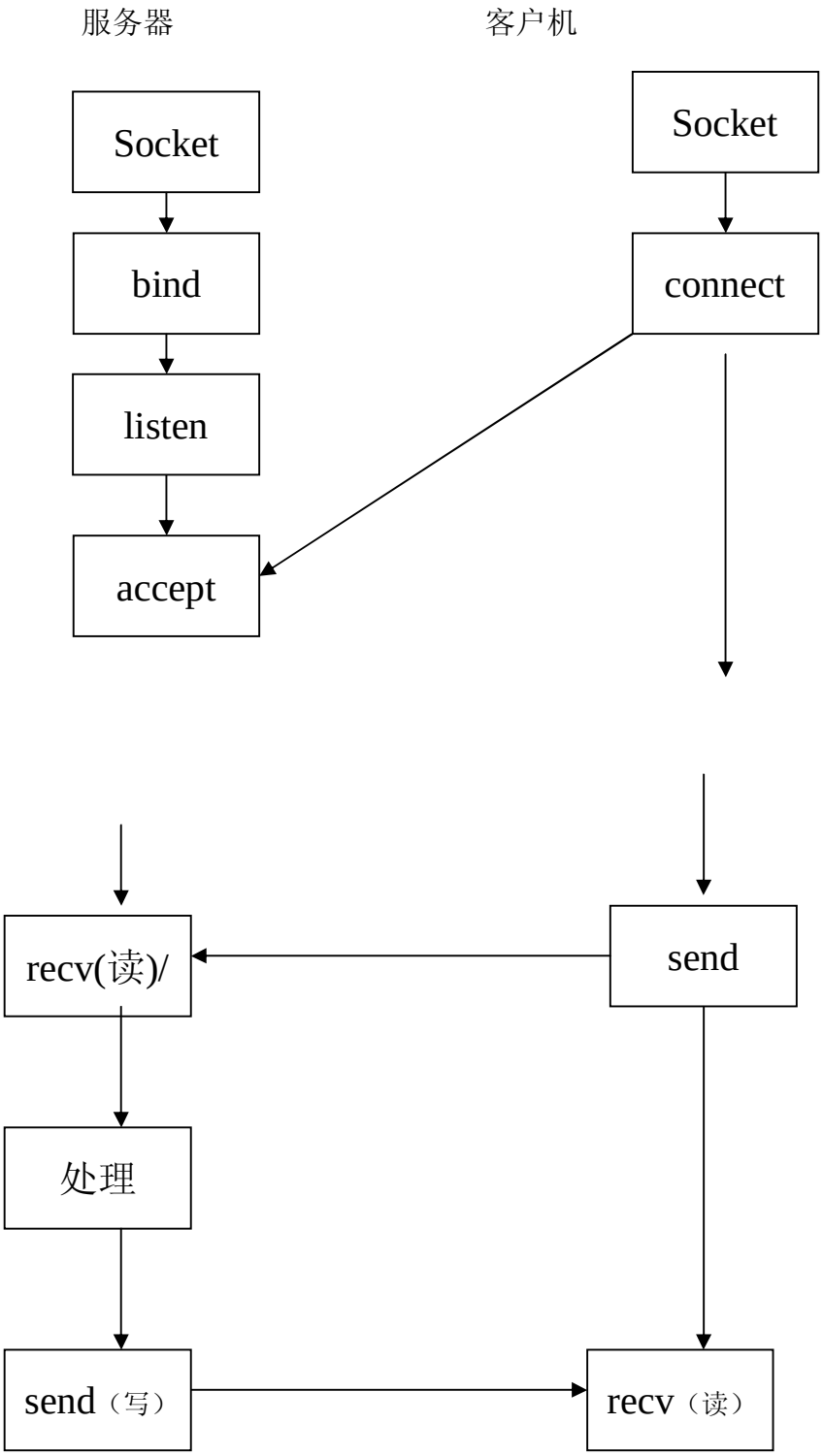
功能：将缓冲区dest中用字符c进行填充，共填充count个字符。
如：

```
struct sockaddr_in sockin;
memset(&sockin,0,sizeof(sockin));//将结构sockin全部清零。
```

2．7 面向连接的协议（TCP）：Winsock 函数

接收连接和建立连接所需要的Winsock 函数。如何监听客户机连接，并接受或拒绝一个连接。怎样初始化同服务器的一个连接。数据在连接会话中是如何传输的。
WinSock下TCP通信图式：

2.7.1 TCP 的 C/S 通信模型



2.7.2 基本套接字函数

本节主要讨论TCP套接字使用这些函数的情形 。

前面描述了TCP C/S间的交互过程，这是使用TCP协议进行网络通信的程序

的基本模型。

1. 函数socket

函数socket创建一个套接字描述符。其定义如下：

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

功能：创建一个未绑定的socket。SOCKET定义为u_int.

参数：1）af:[输入] 指定要创建的套接字的协议簇；TCP/IP协议使用：AF_INET

2）type：[输入]要创建的套接字类型：

字节流型：SOCK_STREAM

数据报型：SOCK_DGRAM

原始SOCKET：SOCK_RAW

3）protocol：[输入]指定使用哪种协议。通常设置为 0，表示使用该类型SOCKET的默认协议。对字节流型的SOCK_STREAM意味着是TCP，对数据报型的SOCK_DGRAM意味着是UDP，对原始SOCKET的SOCK_RAW则必须指明（如：ICMP或IGMP等），因为此处填写的值将直接写入IP包的包头中。

支持的protocol值定义有：

```
/*
 * Protocols
 */
#define IPPROTO_IP              0               /* dummy for IP */
#define IPPROTO_ICMP            1               /* control message protocol */
#define IPPROTO_IGMP            2               /* internet group management protocol
*/
#define IPPROTO_GGP             3               /* gateway^2 (deprecated) */
#define IPPROTO_TCP             6               /* tcp */
#define IPPROTO_PUP             12              /* pup */
#define IPPROTO_UDP             17              /* user datagram protocol */
#define IPPROTO_IDP             22              /* xns idp */
#define IPPROTO_ND              77              /* UNOFFICIAL net disk proto */
```

```
#define IPPROTO_RAW             255             /* raw IP packet */
#define IPPROTO_MAX             256

/*
 * Port/socket numbers: network standard functions
 */
#define IPPORT_ECHO             7
#define IPPORT_DISCARD          9
#define IPPORT_SYSTAT           11
#define IPPORT_DAYTIME          13
#define IPPORT_NETSTAT          15
#define IPPORT_FTP              21
#define IPPORT_TELNET           23
#define IPPORT_SMTP             25
#define IPPORT_TIMESERVER       37
#define IPPORT_NAMESERVER       42
#define IPPORT_WHOIS            43
#define IPPORT_MTP              57

/*
 * Port/socket numbers: host specific functions
 */
#define IPPORT_TFTP             69
#define IPPORT_RJE              77
#define IPPORT_FINGER           79
#define IPPORT_TTYLINK          87
#define IPPORT_SUPDUP           95
```

返回值：若成功则返回一个正整数，称为套接字描述符，标识这个套接字；否则，返回**INVALID_SOCKET**（该值不是**-1**）

例：创建一个TCP套接字的操作一般如下：

```
SOCKET   sockfd = socket(AF_INET，SOCK_STREAM，O)；

if(sockfd==INVALID_SOCKET){

printf("socket error：%d\n"，WSAGetLastError())；

……..

}
```

2、函数bind

函数bind将本地地址和端口号与套接字绑定在一起。其定义如下：

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

功能：将本地地址和端口号与套接字绑定在一起。

参数：

1）s：[输入]要绑定的套接字描述符；

2）name:[输入]实际上是使用struct sockaddr_in的变量的地址。在其中填写地址与端口号。

3）namelen:[输入]套接字地址结构的长度。

返回值：若成功则返回 0；否则返回SOCKET_ERROR（-1）。

服务器和客户机都可以调用函数bind来绑定套接字地址，但一般是服务器调用函数bind来绑定自己的公认端口号。绑定操作一般如下：

注意是使用：struct　sockaddr_in　　myaddr;

而不是：struct　sockaddr　　myaddr;

memset(&myaddr,0,sizeof(myaddr));//清 0　string.h

myaddr.sin_family = AF_INET；

myaddr.sin_port = htons(PORT)；

myaddr.sin_addr.s_addr = htonl(INADDR_ANY)；

if(bind(sockfd，(struct sockaddr *)&myaddr，sizeof(myaddr)) !＝0)

{

printf("Bind to port　%d　error\n"，WSAGetLastError())；

closesocket(sockfd);

……

}

绑定操作一般有如下 5 种组合方式:

程序类型	IP址	端口号	说　明
服务器	INADDR_ANY	非零值	指定服务器的公认端口号
服务器	本地IP地址	非零值	指定服务器的IP地址和公认端口号
客户机	INADDR_ANY	非零值	指定客户机的连接端口号
客户机	本地IP地址	非零值	指定客户机的IP地址和连接端口号
客户机	本地IP地址	零	指定客户机的IP地址

下面详细说明上表中列出的 5 种方式：

(1)服务器指定套接字的公认端口号，不指定IP地址。

服务器调用函数bind时，如果设置套接字的IP地址为特殊的INADDR_ANY，表示它愿意接收来自任何网络设备接口的客户机连接。这是服务器最经常使用的绑定方式。发送时，源IP是默认输出端口网卡的IP。这是多网卡的服务器上常用的方式。

(2)服务器指定套接字的公认端口号和IP地址。

服务器调用函数bind时，如果设置套接字的IP地址为某个本地IP地址，这表示服务器只接收来自对应于这个IP地址的特定网络设备接口的客户机连接。如果这台机器只有一个网络设备接口，这和第一种情况是没有区别的，但当这台机器有多个网络设备接口时，我们可以用这种方式来限制服务器的接收范围。

(3)客户机指定套接字的连接端口号，不指定地址(地址为INADDR_ANY)。

在一般情况下，客户机不用指定自己的套接字的端口号，当客户机调用函数connect进行TCP连接时，系统会自动为它选择一个未用的端口号，并且用本地的IP地址来填充套接字地址中的相应项。但在有的情况下，客户机需要使用

特定端口号，如Linux系统中的rlogin命令(见RFC1282)，因为rlogin命令需要使用保留端口号，而系统不会为客户机自动分配一个保留端口号，所以需要调用函数bind来和一个未用的保留端口号绑定。

(4)指定客户机的IP地址和连接端口号。

表示客户机使用指定的网络设备接口和端口号进行通信。

(5)指定客户机的IP地址，不指定端口号。

表示客户机使用指定的网络设备接口进行通信，系统自动为客户机选择一个未用的端口号（此时在：1024 到 5000 之间自动选择一个）。一般只有在主机有多个网络设备接口时使用。

注：在编写客户机程序时，一般不要使用固定的客户机端口号，除非是在必须使用特定端口号的特殊情况下。

3．函数listen

函数listen将一个套接字转换为倾听套接字(listening socket)。其定义如下：

int listen(SOCKET　sockfd，int backlog)；

功能：将一个已绑定的套接字转换为倾听套接字。

参数：

1）　sockfd：[输入]指定要转换的套接字描述符；

2）　backlog：[输入]设置请求队列的最大长度（处于等待建立TCP全连接的请求，通常是半打开的TCP连接）。

返回值：成功则返回 0；否则返回SOCKET_ERROR（-1）。

服务器需要调用函数listen将套接字转换成倾听套接字，以便接收客户机请求。函数listen的功能有两个：

(1)函数socket创建的套接字是主动套接字，可以用它来进行主动连接(调用函数connect)，但是不能接收连接请求，而服务器的套接字必须能够接收客户机的请求。函数listen将一个尚未连接的主动套接字转换成为一个被动套接

字：告诉TCP协议，这个套接字可以接收连接请求。执行函数listen之后，服务器的TCP状态由CLOSED状态转换成LISTEN状态。

(2)TCP协议将到达的连接请求排队，函数listen的第二个参数指定这个队列的最大长度。 要创建一个倾听套接字，必须首先调用函数socket创建一个主动套接字，然后调用函数bind将它与服务器套接字地址绑定在一起，最后调用函数listen进行转换。这 3 步操作是所有TCP服务器所必须的操作。若该请求队列已满，但又有新的请求到达时，则新的请求将被拒绝。backlog的最大值由具体实现指定。若程序员指定的值超过该值，则系统会自动选择一个最接近用户值但系统又能支持的值。

4．函数accept

函数accept从倾听套接字的已完成连接队列中接收—个连接（listen有两个对队：一个是未完成连接队列。其长度由上述backlog中定义。若TCP连接完成后，则将连接从未完成队列移到已完成队列中，等待由accept()函数取走）。如果已完成连接队列为空，那么这个进程睡眠。其定义如下：

SOCKET accept

(SOCKET sockfd，

struct sockaddr *addr，

int *addrlen)；

功能：从倾听套接字的已完成连接队列中接收—个连接。

参数：

1） sockfd：[输入]等待连接的倾听套接字描述符；

2） addr：[输出]为指向一个Internet套接字地址结构的指针（即 struct sockaddr_in *）；用于存放远方本次连接的IP地址与端口号。若对该信息

不感兴趣，则可置为：NULL。

3） addrlen：[输入/输出]：为指向一个整型变量的指针。输入时，该指针变量所指的内容为：addr的空间长度。输出时为远方IP地址的长度。当addr为NULL时，此域为NULL。

返回值：函数accept成功执行时，返回 3 个结果：

函数返回值为一个新的套接字描述符，标识这个接收的连接；参数addr指向的结构变量中存储客户机地址与端口号；参数addlen指向的整型变量中存储客户机地址的长度。如果对客户机的地址和长度不感兴趣，可以将参数addr和addrlen设置为NULL。函数accept执行失败时，返回 INVALID_SOCKET（定义这~0,即非 0）.

函数accept从倾听套接字的完成连接队列中接收一个已经建立起来的TCP连接，因为倾听套接字是专为接收客户机连接请求，完成 3 次握手操作而用的，所以TCP协议不能使用倾听套接字描述符来标识这个连接，于是TCP协议创建一个新的套接宇来标识这个要接收的连接，并将它的描述符返回给应用程序。现在有两个套接字，一个是调用函数accept时使用的倾听套接字，另一个是函数accept返回的连接套接字(connected socket)。这两个套接字的作用是完全不同的：一个服务器进程通常只需创建一个倾听套接字，在服务器进程的整个活动期间，用它来接收所有客户机的连接请求，在服务器进程终止前关闭这个倾听套接字；而对于每个接收的连接，TCP协议都创建一个新的连接套接字，来标识这个连接，服务器使用这个连接套接字与客户机进行通信操作，当服务器处理完这个客户机请求时，关闭这个连接套接字。

5．函数closesocket

函数close关闭一个套接字。其定义如下：

int closesocket(SOCKET　sockfd)；

功能：关闭一个已存在的套接字。释放相关的资源。

参数：sockfd：[输入]指定要关闭的套接字。

返回值：函数closesocket成功执行时，返回 0；否则，返回-1（SOCKET_ERROR）。

套接字描述符的closesocket操作和文件描述符的closesocket e操作一样：函数closesocket将套接字描述符的引用计数减1，如果描述符的引用计数大于0，则表示还有进程引用这个描述符，函数closesocket正常返回；如果描述符的引用计数变为 0，则表示再没有进程引用这个描述符，于是启动清除套接字描述符的操作，函数closesocket立即正常返回。清除套接字描述符的操作是：将这个套接字描述符标记为关闭状态(不同于TCP协议状态转换图中的CLOSED状态)，然后立即返回进程。调用了函数closesocket之后，进程将不再能够访问这个套接字，但是这不表示TCP协议删除了这个套接字。最基本的语义是：（真正的行为受：SO_DONTLINGER和SO_LINGER选项值的影响）TCP协议将继续使用这个套接字，将尚未发送的数据传递到对方，然后发送FIN数据段，执行关闭操作，一直等到这个TCP连接完全关闭之后，TCP协议才删除这个套接字。

6．函数connect

函数connect用于客户机向服务器发起一个连接。其定义如下：

int connect(SOCKET sockfd，struct sockaddr *servaddr, int addrlen)；

功能：用于客户机向服务器发起一个连接。

参数：

1） sockfd：[输入]客户机自己的socket套接字；

2） servaddr：[输入]在TCP/IP下，是远程服务器的struct sockaddr_in结构

的地址，在其中包含服务器的IP地址和端口号；

3） addrlen：[输入]servaddr所指的struct sockaddr_in结构的长度。

返回值：成功则返回O；否则返回SOCKET_ERROR（-1）

客户机在调用函数connect之前，需要首先指定服务器进程的套接字地址。客户机向服务器发起一个TCP连接的操作一般如下：

```
struct sockaddr_in   servaddr;
memset(&servaddr，sizeof(servaddr))；
servaddr.sin_family = AF_INET；
servaddr.sin_port = htons(SERVER_PORT)；
u_long   r =servaddr.sin_addr.s_addr = inet_addr(“192.168.0.1”);
if( r == INADDR_NONE )
{
printf("inet_aton error\n")；
exit(1)；
}
if(connect(sockfd，(struct sockaddr *)&servaddr,sizeof(servaddr)) ==
SOCKET_ERROR)
{
printf("connect error：%d"，WSAGetLastError())；
exit(1)；
}
```

2.7.3 网络数据传输

要在已建立连接的套接字上发送接收数据，可用API 函数：send 和WSASend 。同样地，在已建立了连接的套接字上接收可用API 函数：recv 和WSARecv 。所有关系到收发数据的缓冲都属于简单的char 类型。另外，所有收发函数返回的错误代码都是SOCKET_ERROR 。一旦返回错误，系统就会
调用WSAGetLastError 获得详细的错误信息。

1. 发送数据：send（）

要在已建立连接的套接字上发送数据，第一个可用的API 函数是send ，其原型为：

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

功能：在已建立连接的套接字上发送数据。
参数：
1）s:[输入] 已建立连接的套接字，将在这个套接字上发送数据。
2）buf：[输入] 字符缓冲区，其中包含即将发送的数据。
3）len ：[输入]指定即将发送的缓冲区内的字符数。
4）flags：[输入] 可为0 、MSG_DONTROUTE 或MSG_OOB 。另外，flags 还可以是对那些标志进行按位“或运算”的一个结果。MSG_DONTROUTE 标志要求传送层不要将它发出的包路由出去。由基层的传送决定是否实现这一请求（例如，若传送协议不支持该选项，这一请求就会被忽略）。MSG_OOB 标志预示数据应该被带外发送。
返回值：返回实际发送的字节数；若发生错误，就返回SOCKET_ERROR 。

2. 接受数据：recv（）
对在已连接套接字上接受数据来说，recv 函数是最基本的方式。它的定义如下：

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

功能：在已连接套接字上接受数据。
参数：
1）s：[输入]准备接收数据的那个套接字。
2）buf ：[输出]即将收到数据的字符缓冲区。

3）len：[输入] 准备接收的字节数或buf 缓冲的长度。
4）flags ：[输入]可以是下面的值：0 、MSG_PEEK 或MSG_OOB 。另外，还可对这些标志中的每一个进行按位和运算。当然，0 表示无特殊行为。MSG_PEEK 会使有用的数据复制到所提供的接收端缓冲内，但是没有从系统缓冲中将它删除。
返回值：返回已接受的字节数。若socket已关闭，则返回 0，若出错返回：**SOCKET_ERROR**。

[程序设计实例]

设计一个基于TCP的（单线程）网络程序，完成：客户端向服务器发送字符串，服务器向客户端返回其中包含的英文字母的个数。当客户端发送“quit”命令时，双方都结束运行。

（A） 服务器方程序基本设计

根据上述所讨论的Winsock的基本函数及TCP下C/S的工作模型，基本的TCP服务器程序的设计步骤如下：
S1、调用WSAStartup（）装载Winsock相应版本的DLL库。
S2、调用socket()创建一个socket。
S3、调用bind()绑定服务器的IP和PORT。
S4、调用listen()变成倾听socket。
S5、调用accept()等待客户机的连接。
S6、调用send()/recv()按应用协议进行网络通信。
S7、调用closesocket()关闭相应的socket。
[mytcpserver.cpp]

（B） 客户机方程序基本设计

根据上述所讨论的Winsock的基本函数及TCP下C/S的工作模型，基本的TCP客户机程序的设计步骤如下：
S1、调用WSAStartup（）装载Winsock相应版本的DLL库。
S2、调用socket()创建一个socket。
S3、调用connect 向服务器发起一个TCP连接。
S6、调用send()/recv()按应用协议进行网络通信。
S7、调用closesocket()关闭相应的socket。
[mytcpclient.cpp]

2.7.4 TCP 服务器工作模型：多线程模型

2.7.4.1 Win32 线程

1）定义 Win32 线程

注：由于在线程中要访问 I/O，故要使用支持多线程的 C 运行库版本。在 VC 中

设置如下：
创建一个项目后，选：Project->Settings…->C/C++:在 category 下选择：Code Generation，然后在 Use run-time library 下选择：Mutilthreaded 或 Debug Mutilthreaded，最后按“OK”。

一个 Win32 线程是如下格式定义的一个函数：（在多线程版本的 VC 中）

unsigned int (CALLBACK *start_address)(void *);
其中：输入给线程的参数是：void *。返回值是usigned int。返回值及其含义由程序员自己定义(即线程的出口代码)。
如：定义一个线程，打印1，2，3，。。，10.每隔2秒打印一个值。打印时同时打印出线程的ID，以表明是哪一个线程打印的。

```
unsigned int    CALLBACK    MyThread(void * param)
{
    for (int i=1;i<=10;i++)
    {
        printf("线程:%d  打印：%d \n",GetCurrentThreadId(),i);
        Sleep(2000);
    }
    return 1;
}
```

2)创建并运行一个Win32 线程

使用_beginthreadex()函数创建并运行一个线程。该函数在process.h 中定义如下：

```
unsigned long _beginthreadex(
void * security,
unsigned int stack_size,
unsigned int (CALLBACK * start_address)(void *    param),
void * arglist,
unsigned int initflag,
unsigned int * threadID
);
```

功能：创建并运行由start_address所表示的一个Win32 线程。

参数：
1）security:[输入]线程的安全属性。使用NULL，表示采用默认的安全属性。
2）stack_size:[输入]线程的栈的大小。通常是 0，表示：采用默认大小。
3）start_address:[输入]线程执行的代码。
4）arglist:[输入]传递给线程的参数。其实是传给param。
5）initflag:[输入]新创建的这个线程的运行状态参数。通常是 0.表示创建后立即运行。若是：**CREATE_SUSPENDED**，则表示创建后暂不运行，以后用ResumeThread()方法再运行。

6）threadID:[输出]新线程的ID。
返回值：成功则返回一个线程的handle。该handle必须强制转换成HANDLE类型来使用Win32API。不成功则返回 0.
注：有两个线程的标识：一个是线程ID，另一个是线程的HANDLE。这两个有何不同？线程ID仅仅是一个unsigned int，是线程在整个Win系统中的唯一的编号。在整个系统范围内或跨进程之间对线程进行操作，会需要线程ID。而线程的HANDLE不同。是与该线程相关联的在系统内核中的一块数据结构。登记、标识该线程的相关信息。这是一个内核对象！由KERNAL32.DLL进行管理。这个内核对象的空间必须用CloseHandle(…)来释放。

例：创建上述那个线程：MyThread()如下 ：

```
u_int id=0;
HANDLE hThread = (HANDLE)_beginthreadex(NULL,0,MyThread,NULL,0,&id);
if(hTHread!=0)
{
printf("线程ID：%d已经启动。\n",id);
}
```

3）释放内核对象：CloseHandle(HANDLE)
定义格式如下：BOOL CloseHandle（HANDLE hObject）；
功能：释放内核对象的空间。
参数：hObject:[输入]要释放的内核对象。
返回值：若成功则返回TRUE，否则返回FALSE。
注：所有的内核对象都可以拥有多个所有者（这些所有者只能是进程，不可能是线程。因为在WIN中只有进程才能拥有内核对象。）。故每一个内核对象都维持一个引用计数器。调用CloseHandle（）时将引用次数减一。当为零时，内核对象空间才真正撤消。

4）获取当前正在运行的线程ID：GetCurrentThreadId
方法定义如下：

```
DWORD WINAPI GetCurrentThreadId(void);
```

功能：返回当前正在运行的线程的ID。
返回值：线程ID。
注：另外一个方法是-beginthreadex()中的最后一个参数。

5）获取线程的出口代码：GetExitCodeThread（）
方法定义如下：

```
BOOL GetExitCodeThread(HANDLE hThread,LPDWORD lpExitCode);
```

功能：获取线程的出口代码。
参数：
hThread:[输入]线程HANDLE。

lpExitCode;[输出]线程的出口代码。
返回值：成功返回TRUE，否则返回FALSE。
该方法可用去判线程的状态。若返回TRUE，则根据返回码可判线程状态：若是STILL_ACTIVE，表示没有结束。否则已结束。若返回FALSE，则要用GetLastError()找原因。

6)使挂起的线程继续运行：ResumeThread
定义格式如下：

```
DWORD WINAPI ResumeThread(
  HANDLE hThread
);
```

功能：使挂起的线程继续运行
参数：hThread:[输入]要继续运行的线程
返回值：不成功则返回：-1，成功则返回线程目前挂起的次数。
代码示例：

```
HANDLE hThread ＝ （HANDLE）
_beginthreadex(NULL,0,MyThread,NULL,CREATE_SUSPENDED,0);
//线程创建后先不运行
…….
……
ResumeThread(hThread);//线程恢复运行
```

7）使当前线程睡眠：Sleep()
使当前线程睡眠。定义如下：

```
VOID WINAPI Sleep(
  DWORD dwMilliseconds
);
```

功能：使当前线程睡眠。
参数：*dwMilliseconds*:[输入]要睡眠的时间。毫秒。这是至少的睡眠时间。若是值：0，则当前线程立即放弃CPU，让同等优先级或更高优先级的线程占有CPU。若不存在这样的线程，则当前线程将继续运行。

8） 终止一个线程：_endthreadex
终止一个线程。注：当_beginthreadex()中线程运行结束会自动调用该函数。

```
void _endthreadex(
   unsigned retval
);
```

功能：立即强制结束一个线程的运行。
参数：retval:[输入]线程的出口代码。
注：程序启动后立即执行的线程是主线程（main()或WinMain()），其它线程由它产生。主线程负责GUI及事件循环。当主线程结束，其它由它产生的所有线程都将被强迫结束。因此通常主线程结束之前，等待它的所有线程结束后再结

束。主线程结束，则表示整个程序结束。

9） 挂起一个线程：SuspendThread

```
DWORD WINAPI SuspendThread(
  HANDLE hThread
);
```

功能：挂起指定的线程。
参数：hThread[输入]要挂起的线程。
返回值：不成功返回-1，成功则返回挂起的次数。
注：使用ResumeThread（）使线程继续运行。

10） 等待一个线程的结束：DWORD WaitForSingleObject()

线程的等待操作是十分重要的线程间简单的同步操作。应使用等待操作，绝对不要使用循环来不断检测线程是否已达到指定的条件！
方法定义如下：
DWORD WaitForSingleObject(HANDLE handle,DWORD dwMs)
功能：等待handle所表示的内核对象
参数：handle:[输入]要等待的内核对象
dwMs:[输入]最多等待多少毫秒。0：表示立即返回。INFINITE：表示一直等待。
返回值：若失败则返回：WAIT_FAILED。成功则返回可能值有：WAIT_OBJECT_0（表示等待条件已产生）、WAIT_TIMEOUT（等待条件没有发生但时间已到）。

[实例]设计一个程序，产生两个线程，每一个都打印出 1，2，3，。。，10.每隔 2 秒打出一个值。同时打印出线程的ID以表明是哪一个线程打印的。

11） 等待多个对象：WaitForMultipleObjects()
方法定义如下：

```
DWORD WaitForMultipleObjects(
DWORD nCount，
HANDLE * pHandles,
BOOL bWaitAll,
DWORD dwMs
);
```

功能：等待一个或所有handle所表示的内核对象

参数：
nCount:[输入]等待的HANDLE个数。个数不能超过MAXIMUM_WAIT_OBJECTS。如：64
handle:[输入]要等待的内核对象的数组。
bWaitAll:[输入]TRUE,表示等待所有，否则等待任一个。
dwMs:[输入]最多等待多少毫秒。0：表示立即返回。INFINITE：表示一直等待。
返回值：若失败则返回：WAIT_FAILED。成功则返回可能值有：若bWaitAll为TRUE，则返回WAIT_OBJECT_0，若是FALSE,则将返回值减去WAIT_OBJECT_0，则是数组中哪一个HANDLE等待条件产生了。
WAIT_TIMEOUT（等待条件没有发生但时间已到）。

[实例]用该API改造上例。

2.7.4.2 线程的同步与互斥

Win32 中同步与互斥有好几种方式。

1） CRITICAL_SECTION类型（windows.h）

对每一个临界区，程序员可人为地关联一个CRITICAL_SECTION变量。(人为地关联一把锁。)通过该类型的变量及其操作进行同步与互斥。
操作有：
a) 初始化： InitializeCriticalSection（）
定义如下：

```
void WINAPI InitializeCriticalSection(
  CRITICAL_SECTION * lpCriticalSection
);
```

对CRITICAL_SECTION变量进行初始化。
如：

```
CRITICAL_SECTION cr;
InitializeCriticalSection(&cr);
```

b) 进入临界区：EnterCriticalSection
定义如下：

```
void WINAPI EnterCriticalSection(
  CRITICAL_SECTION * lpCriticalSection
);
```

若进入不到临界区则线程在该临界区对象上睡眠。

c)离开临界区：LeaveCriticalSection();
定义如下：

```
void WINAPI LeaveCriticalSection(
  CRITICAL_SECTION * lpCriticalSection
```

);
离开时会唤醒其它在该临界区上睡眠的线程。

d)删除临界区：DeleteCriticalSection
定义如下：
```
void WINAPI DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```
释放该临界区对象中所有由 InitializeCriticalSection 方法分配的系统资源。
大致使用步骤
```
CRITICAL_SECTION cs;
InitializeCriticalSection（&cs）;
……
EnterCriticalSection(&cs);
临界区的操作
LeaveCriticalSection(&cs);
……
DeleteCriticalSection(&cs);
```

[实例]多个线程对同一个单向链表进行插入结点的操作。

2)互斥器：Mutex（windows.h）

（A）这是一个内核对象。锁定所要时间是 CS 的 100 倍
（B）跨进程使用。但 CS 只能在同一上进程中使用。
（C）可以指定等待时间。CS 不行。CS 可能会永远锁定。

a)创建一个互斥器：CreateMutex
定义如下：
```
HANDLE WINAPI CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,//安全属性。NULL 表示默认
  BOOL bInitialOwner,//TRUE:本线程是该互斥器的拥有者。
  LPCTSTR lpName//该互斥器的名字
);
```
成功，返回互斥器内核对象的 HANDLE，否则是 NULL。
类似于：InitializeCriticalSection（&cs）

b)删除一个互斥器
使用 CloseHandle(HANDLE)来删除。
类似于：DeleteCriticalSection(&cs);

c) 锁定一个互斥器
使用 WaitForSingleObject()或 WaitForMultipleObjects()来锁定互斥器。
类似于：EnterCriticalSection(&cs);

d)释放互斥器:ReleaseMutex()
BOOL ReleaseMutex(HANDLE hMutex);
TRUE-成功。FALSE-失败
类似于：LeaveCriticalSection(&cs);

[实例]使用 Mutex 改造 CS 的例

3）信号量：Semaphore

Mutex 是退化的 Semaphore。
（1） 创建信号量：CreateSemaphore
定义如下：
```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,    //NULL 是默认安全属性
  LONG lInitialCount, //初始值。必须>=0 且 <= lMaximumCount
  LONG lMaximumCount, //同一时间可以锁定的线程的最大值
  LPCTSTR lpName //名字
);
```
返回内核对象的 HANDLE。
类似于：InitializeCriticalSection（&cs）

（2） 锁定信号量。
使用 WaitForSingleObject()或 WaitForMultipleObjects()来锁定信号量。
只要信号量的资源>0，则锁定必成功，且资源减 1.
类似于：EnterCriticalSection(&cs);
一个线程可多次锁定信号量，每锁定一次，资源少一个。

（3） 释放信号量：ReleaseSemaphore
定义如下：
```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore, //信号量对象
  LONG lReleaseCount, //输入：每释放一次，资源增加的量（>0）。通常是 1
  LPLONG lpPreviousCount //输出：增加之前资源的值
);
```
成功返回 TRUE 否则 FALSE。

2.7.4.3 多线程 TCP 服务器设计

根据上述所讨论的Winsock的基本函数及TCP下C/S的工作模型，多线程的TCP服务器程序的设计步骤如下：
S1、调用WSAStartup（）装载Winsock相应版本的DLL库。
S2、调用socket()创建一个socket。
S3、调用bind()绑定服务器的IP和PORT。
S4、调用listen()变成倾听socket。
while(继续)
{
S5、调用accept()等待客户机的连接。
S6、创建一个新线程，传入通信用的socket,与客户进行通信。
}//while
S7、关闭倾听SOCKET。
S8、调用WSACleanup()

新线程代码设计步骤如下：
do{
1）调用send()/recv()按应用协议进行网络通信。
}while(通信没结束);
2）调用closesocket()关闭相应的socket。

客户机可保持不变。

2.7.4.3 LINUX 平台网络程序设计（TCP）

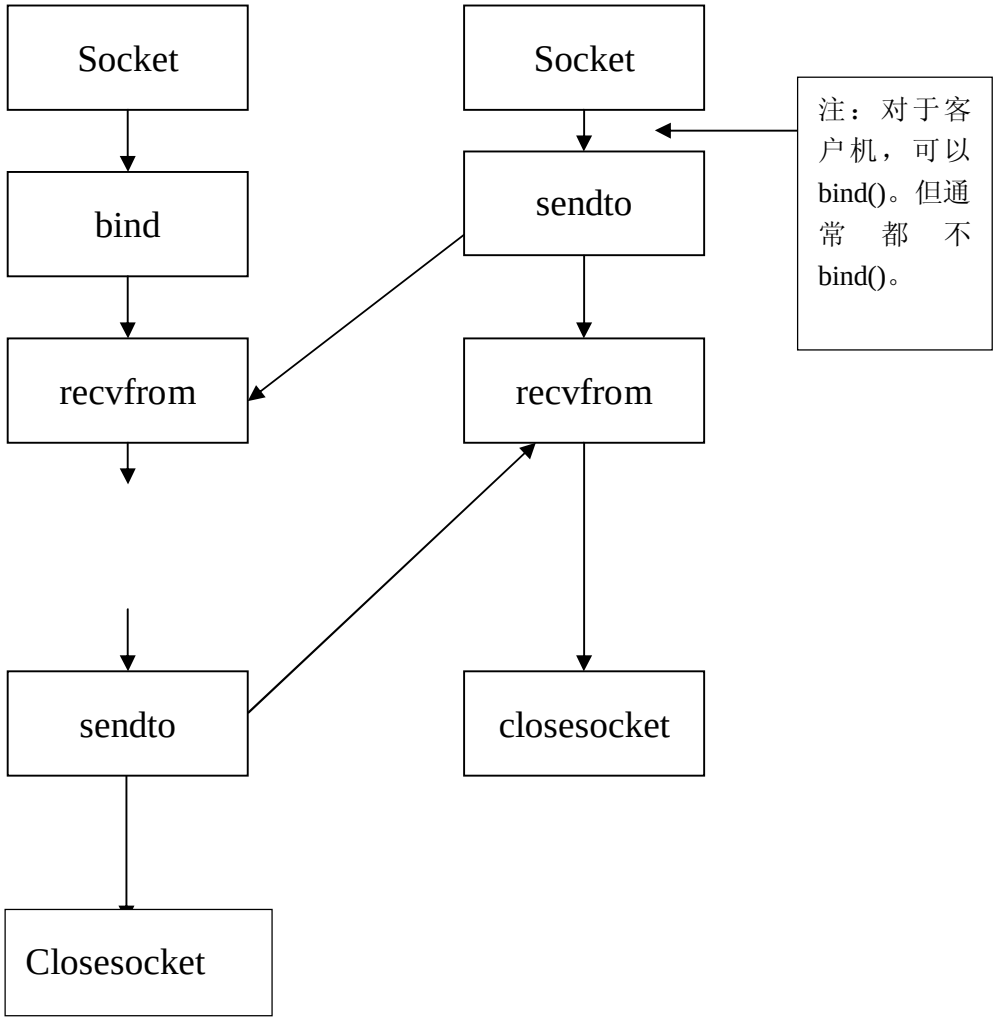
（见“LINUX 网络程序设计.odt”）

2.7.4.4 Java 的 TCP 程序设计

1）（单线程）基于ServerSocket类和Socket类的程序设计。
2）多线程的TCP程序设计

2.7.5 UDP 的 C/S 通信模型(WINDOW 平台)

服务器　　客户机



1) UDPSocket 的建立：

创建 UDPSocket，方式如下：

SOCKET udps = socket(AF_INET,SOCK_DGRAM,0);

If(udps == INVALID_SOCKET)

{//出错处理！

}

2）UDP 数据报的发送：sendto()

Sendto()函数用于发送一个 UDP 包。其定义的格式如下：

```
int sendto(
```

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

功能：用于发送一个 UDP 包。

参数：　1) s:[输入]要发送 UDP 包的 socket。

2) buf:[输入]要发送的数据的缓冲区。

3) len:[输入]要发送的数据的字节数。

4) flags:[输入] 可为 0 、MSG_DONTROUTE 或MSG_OOB 。另外，flags 还可以是对那些标志进行按位“或运算”的一个结果。MSG_DONTROUTE 标志要求传送层不要将它发出的包路由出去。由基层的传送决定是否实现这一请求（例如，若传送协议不支持该选项，这一请求就会被忽略）。MSG_OOB 标志预示数据应该被带外发送。
5)to:[输入]接受方的IP地址与端口号。
6) tolen:[输入]参数to所指的数据区的大小。

返回值：返回实际发送的字节数；若发生错误，就返回SOCKET_ERROR 。

3）接受一个UDP包：recvfrom（）
接受一个UDP包，使用recvfrom() 函数是最基本的方式。它的定义如下：

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

功能：接受一个UDP包。
参数：
1) s：[输入]准备接收UDP包的那个套接字。
2) buf：[输出]即将收到数据的字符缓冲区。
3) len：[输入] 准备接收的字节数或buf 缓冲的长度。
4) flags：[输入]可以是下面的值：0 、MSG_PEEK 或MSG_OOB 。

另外，还可对这些标志中的每一个进行按位和运算。当然，0 表示无特殊行为。M S G _ P E E K 会使有用的数据复制到所提供的接收端缓冲内，但是没有从系统缓冲中将它删除。
5)from:[输出]发送方的地址与端口号。
6) fromlen:[输入]from所指数据区的大小。

返回值：返回已接受的字节数。若socket已关闭，则返回 0，若出错返回：**SOCKET_ERROR**。

UDP实例：

设计一个基于UDP的多客户机的网络程序，完成：客户端向服务器发送字符串，服务器向客户端返回其中包含的英文字母的个数。当客户端发送“quit”命令时，客户机结束运行。

（A） UDP 服务器方程序基本设计(多客户机工作模型)

根据上述所讨论的Winsock的基本函数及UDP下C/S的工作模型，基本的UDP服务器程序的设计步骤如下：
S1、调用WSAStartup（）装载Winsock相应版本的DLL库。
S2、调用socket()创建一个UDP式的socket。
S3、调用bind()绑定服务器的IP和PORT。
while(UDP服务器继续运行)
{
S4、调用recvfrom()按应用协议进行UDP包的读取。
S5、调用sendto()将处理结果应答给客户机。
}
S6、调用closesocket()关闭相应的UDP socket。

服务器方：[udpserver.cpp]

（B） UDP 客户机方程序基本设计

根据上述所讨论的Winsock的基本函数及UDP下C/S的工作模型，基本的UDP客户机程序的设计步骤如下：
S1、调用WSAStartup（）装载Winsock相应版本的DLL库。
S2、调用socket()创建一个socket。
S3、调用sendto()/recvfrom()按应用协议进行UDP网络通信。
S4、调用closesocket()关闭相应的socket。
客户方：[udpclient.cpp]

（C）连接型 UDP

只用于UDP客户机。通常不用于UDP服务器。原因是要求内核进行UDP包的过滤。

基本思路：（UDP客户机）

[connectUDPClient.cpp]