# 网络编程

# 一、socket简介

socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。

## 1、IP地址

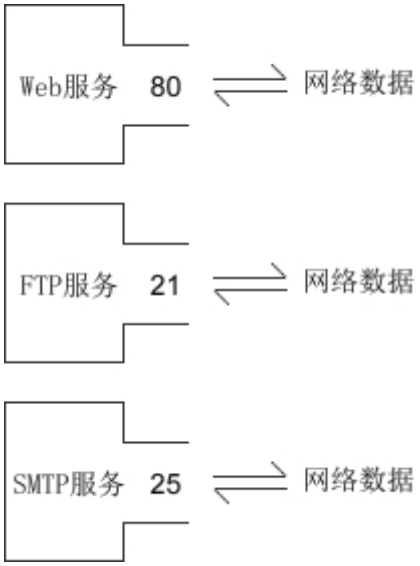
计算机分布在世界各地，要想和它们通信，必须要知道确切的位置。确定计算机位置的方式有多种，IP 地址是最常用的，例如，114.114.114.114 是国内第一个、全球第三个开放的 DNS 服务地址，127.0.0.1 是本机地址。

## 2、端口（Port）

有了 IP 地址，虽然可以找到目标计算机，但仍然不能进行通信。一台计算机可以同时提供多种网络服务，例如Web服务、FTP服务（文件传输服务）、SMTP服务（邮箱服务）等，仅有 IP 地址，计算机虽然可以正确接收到数据包，但是却不知道要将数据包交给哪个网络程序来处理，所以通信失败。

为了区分不同的网络程序，计算机会为每个网络程序分配一个独一无二的**端口号**（Port Number），例如，Web服务的端口号是 80，FTP 服务的端口号是 21，SMTP 服务的端口号是 25。

端口（Port）是一个虚拟的、逻辑上的概念。可以将端口理解为一道门，数据通过这道门流入流出，每道门有不同的编号，就是端口号。如下图所示：




## 3、协议

协议（Protocol）就是网络通信的约定，通信的双方必须都遵守才能正常收发数据。协议有很多种，例如 TCP、UDP、IP 等，通信的双方必须使用同一协议才能通信。协议是一种规范，由计算机组织制定，规定了很多细节，例如，如何建立连接，如何相互识别等。

所谓协议族（Protocol Family），就是一组协议（多个协议）的统称。最常用的是 TCP/IP 协议族，它包含了 TCP、IP、UDP、Telnet、FTP、SMTP 等上百个互为关联的协议，由于 TCP、IP 是两种常用的底层协议，所以把它们统称为 TCP/IP 协议族。

## 4、数据传输方式

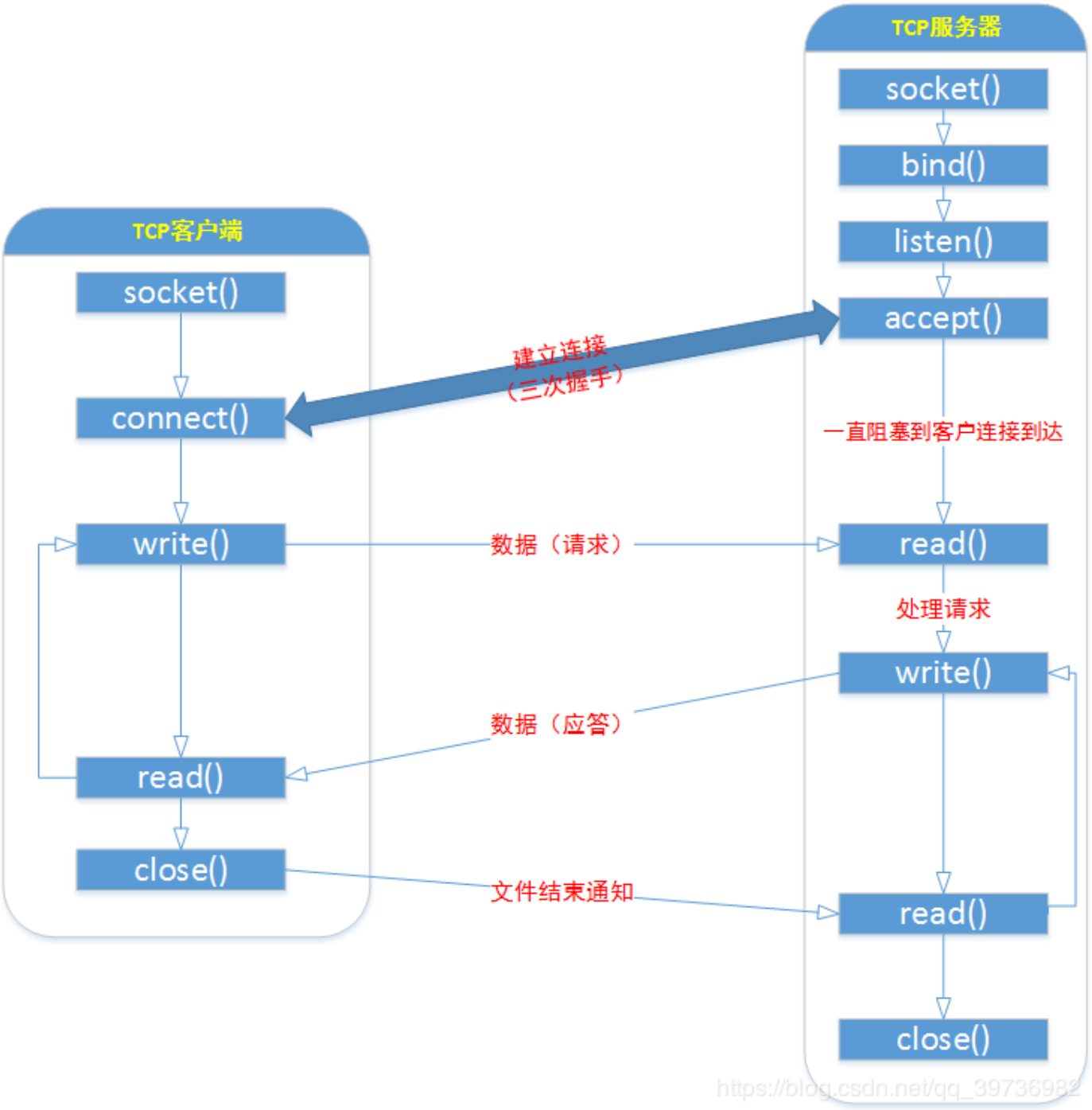
计算机之间有很多数据传输方式，各有优缺点，常用的有两种：**SOCK_STREAM** 和 **SOCK_DGRAM**。

1. SOCK_STREAM 表示面向连接的数据传输方式。数据可以准确无误地到达另一台计算机，如果损坏或丢失，可以重新发送，但效率相对较慢。常见的 http 协议就使用 SOCK_STREAM 传输数据，因为要确保数据的正确性，否则网页不能正常解析。

2. SOCK_DGRAM 表示无连接的数据传输方式。计算机只管传输数据，不作数据校验，如果数据在传输中损坏，或者没有到达另一台计算机，是没有办法补救的。也就是说，数据错了就错了，无法重传。因为 SOCK_DGRAM 所做的校验工作少，所以效率比 SOCK_STREAM 高。

QQ 视频聊天和语音聊天就使用 SOCK_DGRAM 传输数据，因为首先要保证通信的效率，尽量减小延迟，而数据的正确性是次要的，即使丢失很小的一部分数据，视频和音频也可以正常解析，最多出现噪点或杂音，不会对通信质量有实质的影响。

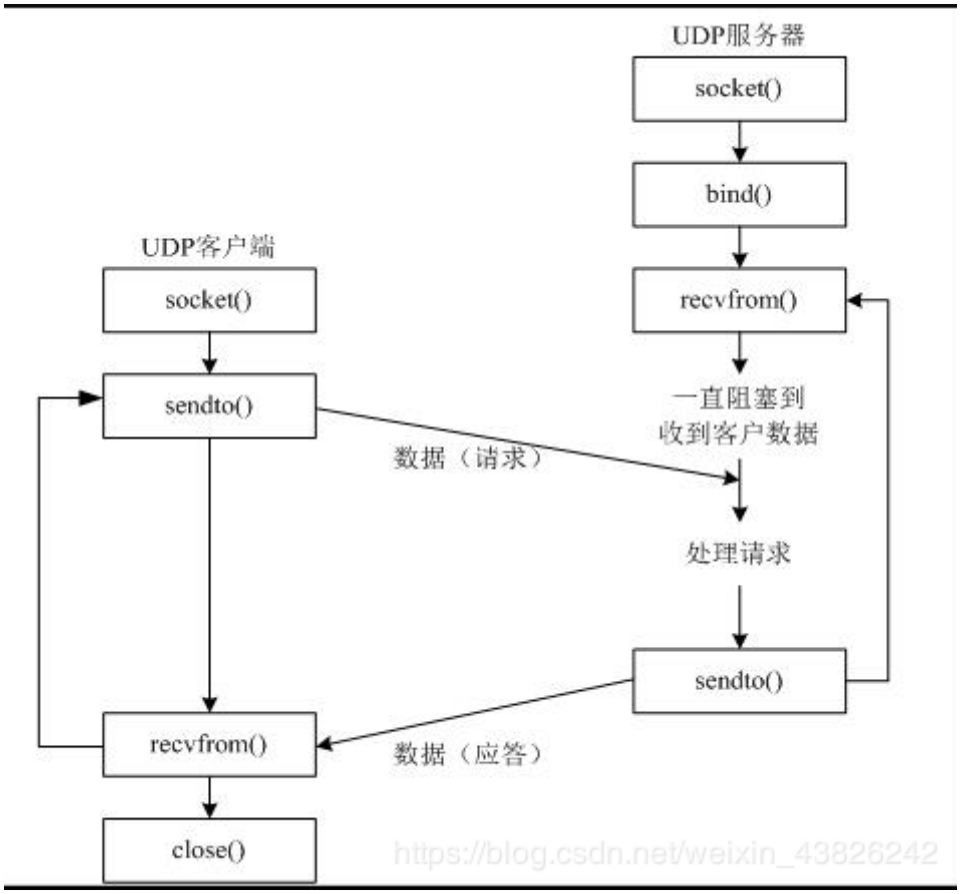
## 5、TCP和UDP的socket模型




基于TCP(面向连接)的socket编程的服务器端程序如下：

1、创建套接字（socket）
2、将套接字绑定到一个本地地址和端口上（bind）
3、将套接字设为监听模式，准备接收客户端请求（listen）
4、等待客户请求到来，当请求到来后，接收连接请求，返回一个新的对应于此次连接的套接字（accept）
5、用返回的套接字和客户端进行通信（send/recv）
6、返回，等待另一客户请求
7、关闭套接字
基于TCP(面向连接)的socket编程的客户端程序如下：

1、创建套接字（socket）
2、向服务器发出连接请求（connect）
3、和服务器端进行通信（send/recv）
4、关闭套接字




基于UDP(面向对象)的socket编程的服务器端程序如下：

1、创建套接字（socket）
2、将套接字绑定到一个本地地址和端口上（bind）
3、等待接收数据（recvfrom）
4、关闭套接字
基于UDP(面向对象)的socket编程的客户端程序如下：

1、创建套接字（socket）
2、向服务器发送数据（sendto）
3、关闭套接字

# 二、程序示例

## 1、Linux socket程序演示

**server.cpp**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(){
    //创建套接字
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    //将套接字和IP、端口绑定
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
    serv_addr.sin_family = AF_INET;  //使用IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    serv_addr.sin_port = htons(1234);  //端口
    bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    //进入监听状态，等待用户发起请求
    listen(serv_sock, 20);
    //接收客户端请求
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_size = sizeof(clnt_addr);
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_ad
dr_size);
    //向客户端发送数据
    char str[] = "Hello World!";
    write(clnt_sock, str, sizeof(str));

    //关闭套接字
    close(clnt_sock);
    close(serv_sock);
    return 0;
}
```

**client.cpp**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
```

```
int main(){
    //创建套接字
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    //向服务器（特定的IP和端口）发起请求
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
    serv_addr.sin_family = AF_INET;  //使用IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    serv_addr.sin_port = htons(1234);  //端口
    connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    //读取服务器传回的数据
    char buffer[40];
    read(sock, buffer, sizeof(buffer)-1);

    printf("Message form server: %s\n", buffer);

    //关闭套接字
    close(sock);
    return 0;
}
```

先编译 server.cpp 并运行：

```
g++ server.cpp -o server
./server
```

正常情况下，程序运行到 accept() 函数就会被阻塞，等待客户端发起请求。

接下来编译 client.cpp 并运行：

```
g++ client.cpp -o client
./client
Message form server: Hello World!
```

client 运行后，通过 connect() 函数向 server 发起请求，处于监听状态的 server 被激活，执行 accept() 函数，接受客户端的请求，然后执行 write() 函数向 client 传回数据。client 接收到传回的数据后，connect() 就运行结束了，然后使用 read() 将数据读取出来。

需要注意的是：
server 只接受一次 client 请求，当 server 向 client 传回数据后，程序就运行结束了。如果想再次接收到服务器的数据，必须再次运行 server，所以这是一个非常简陋的 socket 程序，不能够一直接受客户端的请求。

**源码解析**

**1) 先说一下 server.cpp 中的代码。**

通过 socket() 函数创建了一个套接字，参数 AF_INET 表示使用 IPv4 地址，SOCK_STREAM 表示使用面向连接的数据传输方式，IPPROTO_TCP 表示使用 TCP 协议。在 Linux 中，socket 也是一种文件，有文件描述符，可以使用 write() / read() 函数进行 I/O 操作。

通过 bind() 函数将套接字 serv_sock 与特定的IP地址和端口绑定，IP地址和端口都保存在 sockaddr_in 结构体中。

socket() 函数确定了套接字的各种属性，bind() 函数让套接字与特定的IP地址和端口对应起来，这样客户端才能连接到该套接字。

调用listen()让套接字处于被动监听状态。所谓被动监听，是指套接字一直处于“睡眠”中，直到客户端发起请求才会被“唤醒”。

accept() 函数用来接收客户端的请求。程序一旦执行到 accept() 就会被阻塞（暂停运行），直到客户端发起请求。

write() 函数用来向套接字文件中写入数据，也就是向客户端发送数据。

和普通文件一样，socket 在使用完毕后也要用 close() 关闭。

**2) 再说一下 client.cpp 中的代码。**

client.cpp 中的代码和 server.cpp 中有一些区别。

通过 connect() 向服务器发起请求，服务器的IP地址和端口号保存在 sockaddr_in 结构体中。直到服务器传回数据后，connect() 才运行结束。

通过 read() 从套接字文件中读取数据。

## 2、Windows socket程序演示

**servver.cpp**

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    //绑定套接字
    sockaddr_in sockAddr;
```

```
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //进入监听状态
    listen(servSock, 20);
    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
    //向客户端发送数据
    char *str = "Hello World!";
    send(clntSock, str, strlen(str)+sizeof(char), NULL);
    //关闭套接字
    closesocket(clntSock);
    closesocket(servSock);
    //终止 DLL 的使用
    WSACleanup();
    return 0;
}
```

**client.cpp**

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    //向服务器发起请求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
    connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //接收服务器传回的数据
```

```
    char szBuffer[MAXBYTE] = {0};
    recv(sock, szBuffer, MAXBYTE, NULL);
    //输出接收到的数据
    printf("Message form server: %s\n", szBuffer);
    //关闭套接字
    closesocket(sock);
    //终止使用 DLL
    WSACleanup();
    system("pause");
    return 0;
}
```

将 server.cpp 和 client.cpp 分别编译为 server.exe 和 client.exe，先运行 server.exe，再运行 client.exe，输出结果为：

```
Message form server: Hello World!
```

Windows 下的 socket 程序和 Linux 思路相同，但细节有所差别：

1. Windows 下的 socket 程序依赖 Winsock.dll 或 ws2_32.dll，必须提前加载。
2. Linux 使用“文件描述符”的概念，而 Windows 使用“文件句柄”的概念；Linux 不区分 socket 文件和普通文件，而 Windows 区分；Linux 下 socket() 函数的返回值为 int 类型，而 Windows 下为 SOCKET 类型，也就是句柄。
3. Linux 下使用 read() / write() 函数读写，而 Windows 下使用 recv() / send() 函数发送和接收。
4. 关闭 socket 时，Linux 使用 close() 函数，而 Windows 使用 closesocket() 函数。

## 2.1WSAStartup()函数以及DLL的加载

WinSock（Windows Socket）编程依赖于系统提供的动态链接库(DLL)，最新的DLL是 ws2_32.dll，大小为 69KB，对应的头文件为 winsock2.h。

使用DLL之前必须把DLL加载到当前程序，你可以在编译时加载，也可以在程序运行时加载。
这里使用#pragma命令，在编译时加载：

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

**WSAStartup() 函数**
使用DLL之前，还需要调用 WSAStartup() 函数进行初始化，以指明 WinSock 规范的版本，它的原型为：

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

wVersionRequested 为 WinSock 规范的版本号，低字节为主版本号，高字节为副版本号（修正版本号）；lpWSAData 为指向 WSAData 结构体的指针。

wVersionRequested 参数用来指明我们希望使用的版本号，它的类型为 WORD，等价于 unsigned short，是一个整数，所以需要用 MAKEWORD() 宏函数对版本号进行转换。例如：

```
MAKEWORD(2, 2);  //主版本号为2，副版本号为2，返回 0x0202
```

# 三、使用socket()函数创建套接字

在Linux中，一切都是文件，除了文本文件、源文件、二进制文件等，一个硬件设备也可以被映射为一个虚拟的文件，称为设备文件。例如，stdin 称为标准输入文件，它对应的硬件设备一般是键盘，stdout 称为标准输出文件，它对应的硬件设备一般是显示器。对于所有的文件，都可以使用 read() 函数读取数据，使用 write() 函数写入数据。

“一切都是文件”的思想极大地简化了程序员的理解和操作，使得对硬件设备的处理就像普通文件一样。所有在Linux中创建的文件都有一个 int 类型的编号，称为**文件描述符**（File Descriptor）。使用文件时，只要知道文件描述符就可以。例如，stdin 的描述符为 0，stdout 的描述符为 1。

在Linux中，socket 也被认为是文件的一种，和普通文件的操作没有区别，所以在网络数据传输过程中自然可以使用与文件 I/O 相关的函数。可以认为，两台计算机之间的通信，实际上是两个 socket 文件的相互读写。

文件描述符有时也被称为**文件句柄**（File Handle），但“句柄”主要是 Windows 中术语，所以本教程中如果涉及到 Windows 平台将使用“句柄”，如果涉及到 Linux 平台将使用“描述符”。

## 1、在Linux下创建socket

在 Linux 下使用 <sys/socket.h> 头文件中 socket() 函数来创建套接字，原型为：

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

1. af 为地址族（Address Family），也就是 IP 地址类型，常用的有 AF_INET 和 AF_INET6。AF 是“Address Family”的简写，INET是“Inetnet”的简写。AF_INET 表示 IPv4 地址，例如 127.0.0.1；AF_INET6 表示 IPv6 地址，例如 1030::C9B4:FF12:48AA:1A2B。

大家需要记住127.0.0.1，它是一个特殊IP地址，表示本机地址。

2. type 为数据传输方式，常用的有 SOCK_STREAM 和 SOCK_DGRAM。

3. protocol 表示传输协议，常用的有 IPPROTO_TCP 和 IPPTOTO_UDP，分别表示 TCP 传输协议和 UDP 传输协议。有时也可以填写0可以自动推导出协议类型。除非遇到这样的情况：有两种不同的协议支持同一种地址类型和数据传输类型。如果我们不指明使用哪种协议，操作系统是没办法自动推演的。

## 2、在Windows下创建socket

Windows 下也使用 socket() 函数来创建套接字，原型为：

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

除了返回值类型不同，其他都是相同的。Windows 不把套接字作为普通文件对待，而是返回 SOCKET 类型的句柄。请看下面的例子：

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);  //创建TCP套接字
```

# 四、使用bind()和connect()函数

socket() 函数用来创建套接字，确定套接字的各种属性，然后服务器端要用 bind() 函数将套接字与特定的IP地址和端口绑定起来，只有这样，流经该IP地址和端口的数据才能交给套接字处理；而客户端要用 connect() 函数建立连接。

## 1、bind()函数

bind() 函数的原型为：

```
int bind(int sock, struct sockaddr *addr, socklen_t addrlen);  //Linux
int bind(SOCKET sock, const struct sockaddr *addr, int addrlen);  //Windows
```

sock 为 socket 文件描述符，addr 为 sockaddr 结构体变量的指针，addrlen 为 addr 变量的大小，可由 sizeof() 计算得出。

下面的代码以Linux为例，将创建的套接字与IP地址 127.0.0.1、端口 1234 绑定：

```
//创建套接字
int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
//创建sockaddr_in结构体变量
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
serv_addr.sin_family = AF_INET;  //使用IPv4地址
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
serv_addr.sin_port = htons(1234);  //端口
//将套接字和IP、端口绑定
bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```

**sockaddr_in 结构体**

```
struct sockaddr_in{
    sa_family_t     sin_family;   //地址族（Address Family），也就是地址类型
    uint16_t        sin_port;     //16位的端口号
    struct in_addr  sin_addr;     //32位IP地址
    char            sin_zero[8];  //不使用，一般用0填充
};
```

1. sin_family 和 socket() 的第一个参数的含义相同，取值也要保持一致。

2. sin_prot 为端口号。uint16_t 的长度为两个字节，理论上端口号的取值范围为 0~65536，但 0~1023 的端口一般由系统分配给特定的服务程序，例如 Web 服务的端口号为 80，FTP 服务的端口号为 21，所以我们的程序要尽量在 1024~65536 之间分配端口号。

端口号需要用 htons() 函数转换。

3. sin_addr 是 struct in_addr 结构体类型的变量。

4. sin_zero[8] 是多余的8个字节，没有用，一般使用 memset() 函数填充为 0。上面的代码中，先用 memset() 将结构体的全部字节填充为 0，再给前3个成员赋值，剩下的 sin_zero 自然就是 0 了。

**in_addr 结构体**

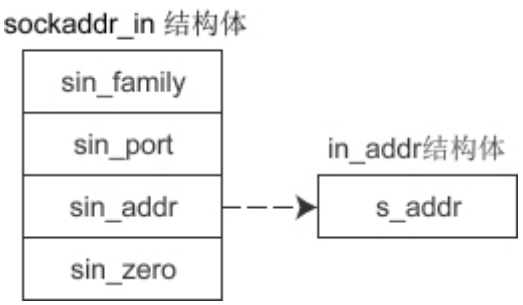
sockaddr_in 的第3个成员是 in_addr 类型的结构体，该结构体只包含一个成员，如下所示：

```
struct in_addr{
    in_addr_t  s_addr;  //32位的IP地址
};
```

in_addr_t 在头文件 <netinet/in.h> 中定义，等价于 unsigned long，长度为4个字节。也就是说，s_addr 是一个整数，而IP地址是一个字符串，所以需要 inet_addr() 函数进行转换，例如：

```
unsigned long ip = inet_addr("127.0.0.1");
printf("%ld\n", ip);
```

运行结果：16777343

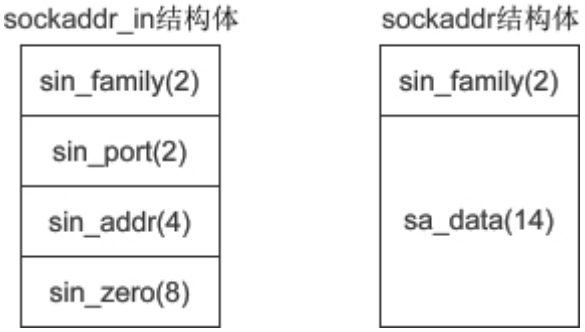


**为什么使用 sockaddr_in 而不使用 sockaddr**
bind() 第二个参数的类型为 sockaddr，而代码中却使用 sockaddr_in，然后再强制转换为 sockaddr，这是为什么呢？

sockaddr 结构体的定义如下：

```
struct sockaddr{
    sa_family_t  sin_family;   //地址族（Address Family），也就是地址类型
    char         sa_data[14];  //IP地址和端口号
};
```

下图是 sockaddr 与 sockaddr_in 的对比（括号中的数字表示所占用的字节数）：




sockaddr 和 sockaddr_in 的长度相同，都是16字节，只是将IP地址和端口号合并到一起，用一个成员 sa_data 表示。要想给 sa_data 赋值，必须同时指明IP地址和端口号，例如"127.0.0.1:80"，遗憾的是，没有相关函数将这个字符串转换成需要的形式，也就很难给 sockaddr 类型的变量赋值，所以使用 sockaddr_in 来代替。这两个结构体的长度相同，强制转换类型时不会丢失字节，也没有多余的字节。

## 2、connect()函数

connect() 函数用来建立连接，它的原型为：

```
int connect(int sock, struct sockaddr *serv_addr, socklen_t addrlen);  //Linux
int connect(SOCKET sock, const struct sockaddr *serv_addr, int addrlen);  //Windows
```

# 五、使用listen()和accept()函数

对于服务器端程序，使用 bind() 绑定套接字后，还需要使用 listen() 函数让套接字进入被动监听状态，再调用 accept() 函数，就可以随时响应客户端的请求了。

## 1、listen()函数

通过 listen() 函数可以让套接字进入被动监听状态，它的原型为：

```
int listen(int sock, int backlog);  //Linux
int listen(SOCKET sock, int backlog);  //Windows
```

sock 为需要进入监听状态的套接字，backlog 为请求队列的最大长度。

所谓被动监听，是指当没有客户端请求时，套接字处于“睡眠”状态，只有当接收到客户端请求时，套接字才会被“唤醒”来响应请求。

**请求队列**
当套接字正在处理客户端请求时，如果有新的请求进来，套接字是没法处理的，只能把它放进缓冲区，待当前请求处理完毕后，再从缓冲区中读取出来处理。如果不断有新的请求进来，它们就按照先后顺序在缓冲区中排队，直到缓冲区满。这个缓冲区，就称为**请求队列**（Request Queue）。

缓冲区的长度（能存放多少个客户端请求）可以通过 listen() 函数的 backlog 参数指定，但究竟为多少并没有什么标准，可以根据你的需求来定，并发量小的话可以是10或者20。

如果将 backlog 的值设置为 **SOMAXCONN**，就由系统来决定请求队列长度，这个值一般比较大，可能是几百，或者更多。

当请求队列满时，就不再接收新的请求，对于 Linux，客户端会收到 ECONNREFUSED 错误，对于 Windows，客户端会收到 WSAECONNREFUSED 错误。

注意：listen() 只是让套接字处于监听状态，并没有接收请求。接收请求需要使用 accept() 函数。

## 2、accept()函数

当套接字处于监听状态时，可以通过 accept() 函数来接收客户端请求。它的原型为：

```
int accept(int sock, struct sockaddr *addr, socklen_t *addrlen);  //Linux
SOCKET accept(SOCKET sock, struct sockaddr *addr, int *addrlen);  //Windows
```

它的参数与 listen() 和 connect() 是相同的：sock 为服务器端套接字，addr 为 sockaddr_in 结构体变量，addrlen 为参数 addr 的长度，可由 sizeof() 求得。

accept() 返回一个新的套接字来和客户端通信，addr 保存了客户端的IP地址和端口号，而 sock 是服务器端的套接字，大家注意区分。后面和客户端通信时，要使用这个新生成的套接字，而不是原来服务器端的套接字。

最后需要说明的是：listen() 只是让套接字进入监听状态，并没有真正接收客户端请求，listen() 后面的代码会继续执行，直到遇到 accept()。accept() 会阻塞程序执行（后面代码不能被执行），直到有新的请求到来。

# 六、socket数据的发送和接收

## 1、Linux下数据的接收和发送

Linux 不区分套接字文件和普通文件，使用 write() 可以向套接字中写入数据，使用 read() 可以从套接字中读取数据。

两台计算机之间的通信相当于两个套接字之间的通信，在服务器端用 write() 向套接字写入数据，客户端就能收到，然后再使用 read() 从套接字中读取出来，就完成了一次通信。

**write() 的原型为**：

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

fd 为要写入的文件的描述符，buf 为要写入的数据的缓冲区地址，nbytes 为要写入的数据的字节数。

write() 函数会将缓冲区 buf 中的 nbytes 个字节写入文件 fd，成功则返回写入的字节数，失败则返回 -1。

**read() 的原型为**：

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

fd 为要读取的文件的描述符，buf 为要接收数据的缓冲区地址，nbytes 为要读取的数据的字节数。

read() 函数会从 fd 文件中读取 nbytes 个字节并保存到缓冲区 buf，成功则返回读取到的字节数（但遇到文件结尾则返回0），失败则返回 -1。

## 2、Windows下数据的接收和发送

Windows 和 Linux 不同，Windows 区分普通文件和套接字，并定义了专门的接收和发送的函数。

从服务器端发送数据使用 **send() 函数**，它的原型为：

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

sock 为要发送数据的套接字，buf 为要发送的数据的缓冲区地址，len 为要发送的数据的字节数，flags 为发送数据时的选项。

返回值和前三个参数不再赘述，最后的 flags 参数一般设置为 0 或 NULL。

在客户端接收数据使用 **recv() 函数**，它的原型为：

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

# 七、回声客户端的实现

所谓“回声”，是指客户端向服务器发送一条数据，服务器再将数据原样返回给客户端，就像声音一样，遇到障碍物会被“反弹回来”。

对！客户端也可以使用 write() / send() 函数向服务器发送数据，服务器也可以使用 read() / recv() 函数接收数据。

**server.cpp**

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);
    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //进入监听状态
    listen(servSock, 20);
    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
    char buffer[BUF_SIZE];  //缓冲区
    int strLen = recv(clntSock, buffer, BUF_SIZE, 0);  //接收客户端发来的数据
    send(clntSock, buffer, strLen, 0);  //将数据原样返回
    //关闭套接字
    closesocket(clntSock);
    closesocket(servSock);
    //终止 DLL 的使用
    WSACleanup();
    return 0;
}
```

**client.cpp**

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    //向服务器发起请求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
    connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //获取用户输入的字符串并发送给服务器
    char bufSend[BUF_SIZE] = {0};
    printf("Input a string: ");
    scanf("%s", bufSend);
    send(sock, bufSend, strlen(bufSend), 0);
    //接收服务器传回的数据
    char bufRecv[BUF_SIZE] = {0};
    recv(sock, bufRecv, BUF_SIZE, 0);
    //输出接收到的数据
    printf("Message form server: %s\n", bufRecv);
    //关闭套接字
    closesocket(sock);
    //终止使用 DLL
    WSACleanup();
    system("pause");
    return 0;
}
```

先运行服务器端，再运行客户端，执行结果为：

```
Input a string: c-language java cpp↙
Message form server: c-language
```

scanf() 读取到空格时认为一个字符串输入结束，所以只能读取到“c-language”；如果不希望把空格作为字符串的结束符，可以使用 gets() 函数。

## 八、实现迭代服务器端和客户端

服务器一直接收客户端消息，不退出。

**server.cpp**

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);
    //绑定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //进入监听状态
    listen(servSock, 20);
    //接收客户端请求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE] = {0};  //缓冲区
    while(1){
        SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
        int strLen = recv(clntSock, buffer, BUF_SIZE, 0);  //接收客户端发来的
数据
        send(clntSock, buffer, strLen, 0);  //将数据原样返回
        closesocket(clntSock);  //关闭套接字
        memset(buffer, 0, BUF_SIZE);  //重置缓冲区
    }
    //关闭套接字
    closesocket(servSock);
    //终止 DLL 的使用
    WSACleanup();
```

```
    return 0;
}
```

**client.cpp**

```
#include <stdio.h>
#include <WinSock2.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    //向服务器发起请求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用0填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);

    char bufSend[BUF_SIZE] = {0};
    char bufRecv[BUF_SIZE] = {0};
    while(1){
        //创建套接字
        SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
        //获取用户输入的字符串并发送给服务器
        printf("Input a string: ");
        gets(bufSend);
        send(sock, bufSend, strlen(bufSend), 0);
        //接收服务器传回的数据
        recv(sock, bufRecv, BUF_SIZE, 0);
        //输出接收到的数据
        printf("Message form server: %s\n", bufRecv);

        memset(bufSend, 0, BUF_SIZE);  //重置缓冲区
        memset(bufRecv, 0, BUF_SIZE);  //重置缓冲区
        closesocket(sock);  //关闭套接字
    }
    WSACleanup();  //终止使用 DLL
```

```
    return 0;
}
```

先运行服务器端，再运行客户端，结果如下：

```
Input a string: c language
Message form server: c language
Input a string: C语言中文网
Message form server: C语言中文网
Input a string: 学习C/C++编程的好网站
Message form server: 学习C/C++编程的好网站
```

while(1) 让代码进入死循环，除非用户关闭程序，否则服务器端会一直监听客户端的请求。客户端也是一样，会不断向服务器发起连接。

需要注意的是：server.cpp 中调用 closesocket() 不仅会关闭服务器端的 socket，还会通知客户端连接已断开，客户端也会清理 socket 相关资源，所以 client.cpp 中需要将 socket() 放在 while 循环内部，因为每次请求完毕都会清理 socket，下次发起请求时需要重新创建。

# 九、socket缓冲区以及阻塞模式

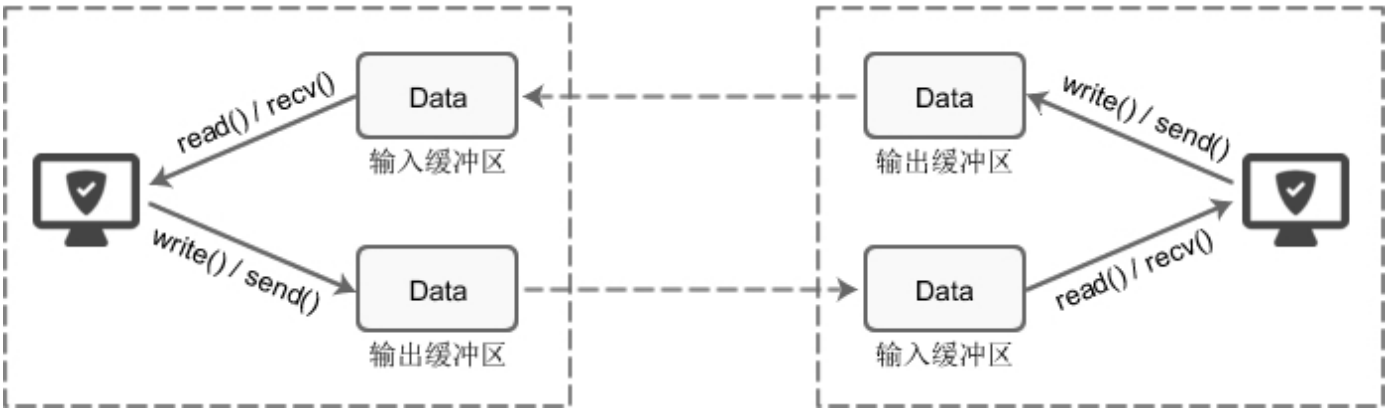
## 1、socket缓冲区

每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。

write()/send() 并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由TCP协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议负责的事情。

TCP协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

read()/recv() 函数也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。



这些I/O缓冲区特性可整理如下：

- I/O缓冲区在每个TCP套接字中单独存在；
- I/O缓冲区在创建套接字时自动生成；
- 即使关闭套接字也会继续传送输出缓冲区中遗留的数据；
- 关闭套接字将丢失输入缓冲区中的数据。

输入输出缓冲区的默认大小一般都是 8K，可以通过 getsockopt() 函数获取：

```
unsigned optVal;
int optLen = sizeof(int);
getsockopt(servSock, SOL_SOCKET, SO_SNDBUF, (char*)&optVal, &optLen);
printf("Buffer length: %d\n", optVal);

运行结果：
Buffer length: 8192
```

## 2、阻塞模式

对于TCP套接字（默认情况下），当使用 write()/send() 发送数据时：

1. 首先会检查缓冲区，如果缓冲区的可用空间长度小于要发送的数据，那么 write()/send() 会被阻塞（暂停执行），直到缓冲区中的数据被发送到目标机器，腾出足够的空间，才唤醒 write()/send() 函数继续写入数据。
2. 如果TCP协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒。
3. 如果要写入的数据大于缓冲区的最大长度，那么将分批写入。
4. 直到所有数据被写入缓冲区 write()/send() 才能返回。

当使用 read()/recv() 读取数据时：

1. 首先会检查缓冲区，如果缓冲区中有数据，那么就读取，否则函数会被阻塞，直到网络上有数据到来。
2. 如果要读取的数据长度小于缓冲区中的数据长度，那么就不能一次性将缓冲区中的所有数据读出，剩余数据将不断积压，直到有 read()/recv() 函数再次读取。
3. 直到读取到数据后 read()/recv() 函数才会返回，否则就一直被阻塞。

这就是TCP套接字的阻塞模式。所谓阻塞，就是上一步动作没有完成，下一步动作将暂停，直到上一步动作完成后才能继续，以保持同步性。

# 十、优雅的断开连接--shutdown()

调用 close()/closesocket() 函数意味着完全断开连接，即不能发送数据也不能接收数据，这种“生硬”的方式有时候会显得不太“优雅”。


上图演示了两台正在进行双向通信的主机。主机A发送完数据后，单方面调用 close()/closesocket() 断开连接，之后主机A、B都不能再接受对方传输的数据。实际上，是完全无法调用与数据收发有关的函数。

一般情况下这不会有问题，但有些特殊时刻，需要只断开一条数据传输通道，而保留另一条。
使用 shutdown() 函数可以达到这个目的，它的原型为：

```
int shutdown(int sock, int howto);  //Linux
int shutdown(SOCKET s, int howto);  //Windows
```

sock 为需要断开的套接字，howto 为断开方式。

howto 在 Linux 下有以下取值：

- SHUT_RD：断开输入流。套接字无法接收数据（即使输入缓冲区收到数据也被抹去），无法调用输入相关函数。
- SHUT_WR：断开输出流。套接字无法发送数据，但如果输出缓冲区中还有未传输的数据，则将传递到目标主机。
- SHUT_RDWR：同时断开 I/O 流。相当于分两次调用 shutdown()，其中一次以 SHUT_RD 为参数，另一次以 SHUT_WR 为参数。

howto 在 Windows 下有以下取值：

- SD_RECEIVE：关闭接收操作，也就是断开输入流。
- SD_SEND：关闭发送操作，也就是断开输出流。
- SD_BOTH：同时关闭接收和发送操作。

**close()/closesocket()和shutdown()的区别**
确切地说，close() / closesocket() 用来关闭套接字，将套接字描述符（或句柄）从内存清除，之后再也不能使用该套接字，与C语言中的 fclose() 类似。应用程序关闭套接字后，与该套接字相关的连接和缓存也失去了意义，TCP协议会自动触发关闭连接的操作。

shutdown() 用来关闭连接，而不是套接字，不管调用多少次 shutdown()，套接字依然存在，直到调用 close() / closesocket() 将套接字从内存清除。

调用 close()/closesocket() 关闭套接字时，或调用 shutdown() 关闭输出流时，都会向对方发送 FIN 包。FIN 包表示数据传输完毕，计算机收到 FIN 包就知道不会再有数据传送过来了。

默认情况下，close()/closesocket() 会立即向网络中发送FIN包，不管输出缓冲区中是否还有数据，而shutdown() 会等输出缓冲区中的数据传输完毕再发送FIN包。也就意味着，调用 close()/closesocket() 将丢失输出缓冲区中的数据，而调用 shutdown() 不会。

# 十一、socket文件传输功能的实现

编写这个程序需要注意两个问题：

1. 文件大小不确定，有可能比缓冲区大很多，调用一次 write()/send() 函数不能完成文件内容的发送。接收数据时也会遇到同样的情况。

要解决这个问题，可以使用 while 循环，例如：

```
//Server 代码
int nCount;
while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0 ){
    send(sock, buffer, nCount, 0);
}
//Client 代码
int nCount;
while( (nCount = recv(clntSock, buffer, BUF_SIZE, 0)) > 0 ){
    fwrite(buffer, nCount, 1, fp);
}
```

对于 Server 端的代码，当读取到文件末尾，fread() 会返回 0，结束循环。

对于 Client 端代码，有一个关键的问题，就是文件传输完毕后让 recv() 返回 0，结束 while 循环。

2. Client 端如何判断文件接收完毕，也就是上面提到的问题——何时结束 while 循环。

最简单的结束 while 循环的方法当然是文件接收完毕后让 recv() 函数返回 0，那么，如何让 recv() 返回 0 呢？**recv() 返回 0 的唯一时机就是收到FIN包时。**

FIN 包表示数据传输完毕，计算机收到 FIN 包后就知道对方不会再向自己传输数据，当调用 read()/recv() 函数时，如果缓冲区中没有数据，就会返回 0，表示读到了“socket文件的末尾”。

这里我们调用 shutdown() 来发送FIN包：server 端直接调用 close()/closesocket() 会使输出缓冲区中的数据失效，文件内容很有可能没有传输完毕连接就断开了，而调用 shutdown() 会等待输出缓冲区中的数据传输完毕。

**server.cpp**

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

```
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 1024
int main(){
    //先检查文件是否存在
    char *filename = "D:\\send.avi";  //文件名
    FILE *fp = fopen(filename, "rb");  //以二进制方式打开文件
    if(fp == NULL){
        printf("Cannot open file, press any key to exit!\n");
        system("pause");
        exit(0);
    }
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    listen(servSock, 20);
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
    //循环发送数据，直到文件结尾
    char buffer[BUF_SIZE] = {0};  //缓冲区
    int nCount;
    while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0 ){
        send(clntSock, buffer, nCount, 0);
    }
    shutdown(clntSock, SD_SEND);  //文件读取完毕，断开输出流，向客户端发送FIN包
    recv(clntSock, buffer, BUF_SIZE, 0);  //阻塞，等待客户端接收完毕
    fclose(fp);
    closesocket(clntSock);
    closesocket(servSock);
    WSACleanup();
    system("pause");
    return 0;
}
```

**client.cpp**

```c
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")
#define BUF_SIZE 1024
int main(){
    //先输入文件名，看文件是否能创建成功
    char filename[100] = {0};  //文件名
    printf("Input filename to save: ");
    gets(filename);
    FILE *fp = fopen(filename, "wb");  //以二进制方式打开（创建）文件
    if(fp == NULL){
        printf("Cannot open file, press any key to exit!\n");
        system("pause");
        exit(0);
    }
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);
    connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //循环接收数据，直到文件传输完毕
    char buffer[BUF_SIZE] = {0};  //文件缓冲区
    int nCount;
    while( (nCount = recv(sock, buffer, BUF_SIZE, 0)) > 0 ){
        fwrite(buffer, nCount, 1, fp);
    }
    puts("File transfer success!");
    //文件接收完毕后直接关闭套接字，无需调用shutdown()
    fclose(fp);
    closesocket(sock);
    WSACleanup();
    system("pause");
    return 0;
}
```

server.cpp 中recv() 并没有接收到 client 端的数据，当 client 端调用 closesocket() 后，server 端会收到FIN包，recv() 就会返回，后面的代码继续执行。

# 十二、socket网络字节序以及大端序小端序

不同CPU中，4字节整数1在内存空间的存储方式是不同的。4字节整数1可用2进制表示如下：

```
00000000 00000000 00000000 00000001
```

有些CPU以上面的顺序存储到内存，另外一些CPU则以倒序存储，如下所示：

```
00000001 00000000 00000000 00000000
```

若不考虑这些就收发数据会发生问题，因为保存顺序的不同意味着对接收数据的解析顺序也不同。

## 1、大端序和小端序

CPU向内存保存数据的方式有两种：

- 大端序（Big Endian）：高位字节存放到低位地址（高位字节在前）。
- 小端序（Little Endian）：高位字节存放到高位地址（低位字节在前）。

假设在 0x20 号开始的地址中保存4字节 int 型数据 0x12345678，大端序CPU保存方式如下图所示：

| 0x20号 | 0x21号 | 0x22号 | 0x23号 |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

对于大端序，最高位字节 0x12 存放到低位地址，最低位字节 0x78 存放到高位地址。小端序的保存方式如下图所示：

| 0x20号 | 0x21号 | 0x22号 | 0x23号 |
|---|---|---|---|
| 0x78 | 0x56 | 0x34 | 0x12 |

网络字节序统一为大端序。
主机A先把数据转换成大端序再进行网络传输，主机B收到数据后先转换为自己的格式再解析。

## 2、网络字节序转换函数

htons() 用来将当前主机字节序转换为网络字节序，其中h代表主机（host）字节序，n代表网络（network）字节序，s代表short，htons 是 h、to、n、s 的组合，可以理解为"将short型数据从当前主机字节序转换为网络字节序"。

常见的网络字节转换函数有：

- htons()：host to network short，将short类型数据从主机字节序转换为网络字节序。
- ntohs()：network to host short，将short类型数据从网络字节序转换为主机字节序。
- htonl()：host to network long，将long类型数据从主机字节序转换为网络字节序。
- ntohl()：network to host long，将long类型数据从网络字节序转换为主机字节序。

- inet_addr() 除了将字符串转换为32位整数，同时还进行网络字节序转换。还可以检测无效IP地址。

# 十三、在socket中使用域名

客户端中直接使用IP地址会有很大的弊端，一旦IP地址变化（IP地址会经常变动），客户端软件就会出现错误。

而使用域名会方便很多，注册后的域名只要每年续费就永远属于自己的，更换IP地址时修改域名解析即可，不会影响软件的正常使用。

## 1、通过域名获取IP地址

域名仅仅是IP地址的一个助记符，目的是方便记忆，通过域名并不能找到目标计算机，通信之前必须要将域名转换成IP地址。

gethostbyname() 函数可以完成这种转换，它的原型为：

```
struct hostent *gethostbyname(const char *hostname);
```

hostname 为主机名，也就是域名。使用该函数时，只要传递域名字符串，就会返回域名对应的IP地址。返回的地址信息会装入 hostent 结构体，该结构体的定义如下：

```
struct hostent{
    char *h_name;  //official name
    char **h_aliases;  //alias list
    int  h_addrtype;  //host address type
    int  h_length;  //address lenght
    char **h_addr_list;  //address list
}
```

从该结构体可以看出，不只返回IP地址，还会附带其他信息，各位读者只需关注最后一个成员 h_addr_list。下面是对各成员的说明：

- h_name：官方域名（Official domain name）。官方域名代表某一主页，但实际上一些著名公司的域名并未用官方域名注册。
- h_aliases：别名，可以通过多个域名访问同一主机。同一IP地址可以绑定多个域名，因此除了当前域名还可以指定其他域名。
- h_addrtype：gethostbyname() 不仅支持 IPv4，还支持 IPv6，可以通过此成员获取IP地址的地址族（地址类型）信息，IPv4 对应 AF_INET，IPv6 对应 AF_INET6。
- h_length：保存IP地址长度。IPv4 的长度为4个字节，IPv6 的长度为16个字节。
- h_addr_list：这是最重要的成员。通过该成员以整数形式保存域名对应的IP地址。对于用户较多的服务器，可能会分配多个IP地址给同一域名，利用多个服务器进行均衡负载。

# 十四、理解UDP套接字

TCP 是面向连接的传输协议，建立连接时要经过三次握手，断开连接时要经过四次握手，中间传输数据时也要回复ACK包确认，多种机制保证了数据能够正确到达，不会丢失或出错。

UDP 是非连接的传输协议，没有建立连接和断开连接的过程，它只是简单地把数据丢到网络中，也不需要ACK包确认。

UDP 传输数据就好像我们邮寄包裹，邮寄前需要填好寄件人和收件人地址，之后送到快递公司即可，但包裹是否正确送达、是否损坏我们无法得知，也无法保证。UDP 协议也是如此，它只管把数据包发送到网络，然后就不管了，如果数据丢失或损坏，发送端是无法知道的，当然也不会重发。

既然如此，TCP应该是更加优质的传输协议吧?

如果只考虑可靠性，TCP的确比UDP好。但UDP在结构上比TCP更加简洁，不会发送ACK的应答消息，也不会给数据包分配Seq序号，所以UDP的传输效率有时会比TCP高出很多，编程中实现UDP也比TCP简单。

UDP 的可靠性虽然比不上TCP，但也不会像想象中那么频繁地发生数据损毁，在更加重视传输效率而非可靠性的情况下，UDP是一种很好的选择。比如视频通信或音频通信，就非常适合采用UDP协议；通信时数据必须高效传输才不会产生“卡顿”现象，用户体验才更加流畅，如果丢失几个数据包，视频画面可能会出现“雪花”，音频可能会夹带一些杂音，这些都是无妨的。

与UDP相比，TCP的生命在于流控制，这保证了数据传输的正确性。

最后需要说明的是：TCP的速度无法超越UDP，但在收发某些类型的数据时有可能接近UDP。例如，每次交换的数据量越大，TCP 的传输速率就越接近于 UDP。

# 十五、基于UDP的服务器端和客户端

**UDP中的服务器端和客户端没有连接**
UDP不像TCP，无需在连接状态下交换数据，因此基于UDP的服务器端和客户端也无需经过连接过程。也就是说，不必调用 listen() 和 accept() 函数。UDP中只有创建套接字的过程和数据交换的过程。

**UDP服务器端和客户端均只需1个套接字**
TCP中，套接字是一对一的关系。如要向10个客户端提供服务，那么除了负责监听的套接字外，还需要创建10套接字。但在UDP中，不管是服务器端还是客户端都只需要1个套接字。之前解释UDP原理的时候举了邮寄包裹的例子，负责邮寄包裹的快递公司可以比喻为UDP套接字，只要有1个快递公司，就可以通过它向任意地址邮寄包裹。同样，只需1个UDP套接字就可以向任意主机传送数据。

**基于UDP的接收和发送函数**
创建好TCP套接字后，传输数据时无需再添加地址信息，因为TCP套接字将保持与对方套接字的连

接。换言之，TCP套接字知道目标地址信息。但UDP套接字不会保持连接状态，每次传输数据都要添加目标地址信息，这相当于在邮寄包裹前填写收件人地址。

发送数据使用**sendto() **函数：

```
ssize_t sendto(int sock, void *buf, size_t nbytes, int flags, struct sockadd
r *to, socklen_t addrlen);  //Linux
int sendto(SOCKET sock, const char *buf, int nbytes, int flags, const struct
 sockadr *to, int addrlen);  //Windows
```

Linux和Windows下的 sendto() 函数类似，下面是详细参数说明：

- sock：用于传输UDP数据的套接字；
- buf：保存待传输数据的缓冲区地址；
- nbytes：带传输数据的长度（以字节计）；
- flags：可选项参数，若没有可传递0；
- to：存有目标地址信息的 sockaddr 结构体变量的地址；
- addrlen：传递给参数 to 的地址值结构体变量的长度。

UDP 发送函数 sendto() 与TCP发送函数 write()/send() 的最大区别在于，sendto() 函数需要向他传递目标地址信息。

接收数据使用 **recvfrom()**函数：

```
ssize_t recvfrom(int sock, void *buf, size_t nbytes, int flags, struct socka
dr *from, socklen_t *addrlen);  //Linux
int recvfrom(SOCKET sock, char *buf, int nbytes, int flags, const struct soc
kaddr *from, int *addrlen);  //Windows
```

由于UDP数据的发送端不不定，所以 recvfrom() 函数定义为可接收发送端信息的形式，具体参数如下：

- sock：用于接收UDP数据的套接字；
- buf：保存接收数据的缓冲区地址；
- nbytes：可接收的最大字节数（不能超过buf缓冲区的大小）；
- flags：可选项参数，若没有可传递0；
- from：存有发送端地址信息的sockaddr结构体变量的地址；
- addrlen：保存参数 from 的结构体变量长度的变量地址值。

## 1、基于UDP的回声服务器端/客户端

UDP不同于TCP，不存在请求连接和受理过程，因此在某种意义上无法明确区分服务器端和客户端，只是因为其提供服务而称为服务器端。

**server.cpp**

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
    //绑定套接字
    sockaddr_in servAddr;
    memset(&servAddr, 0, sizeof(servAddr));  //每个字节都用0填充
    servAddr.sin_family = PF_INET;  //使用IPv4地址
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY); //自动获取IP地址
    servAddr.sin_port = htons(1234);  //端口
    bind(sock, (SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    //接收客户端请求
    SOCKADDR clntAddr;  //客户端地址信息
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE];  //缓冲区
    while(1){
        int strLen = recvfrom(sock, buffer, BUF_SIZE, 0, &clntAddr, &nSize);
        sendto(sock, buffer, strLen, 0, &clntAddr, nSize);
    }
    closesocket(sock);
    WSACleanup();
    return 0;
}
```

在创建套接字时，向 socket() 第二个参数传递 SOCK_DGRAM，以指明使用UDP协议。

使用htonl(INADDR_ANY)来自动获取IP地址。

利用常数 INADDR_ANY 自动获取IP地址有一个明显的好处，就是当软件安装到其他服务器或者服务器IP地址改变时，不用再更改源码重新编译，也不用在启动软件时手动输入。而且，如果一台计算机中已分配多个IP地址（例如路由器），那么只要端口号一致，就可以从不同的IP地址接收数据。所以，服务器中优先考虑使用INADDR_ANY；而客户端中除非带有一部分服务器功能，否则不会采用。

**client.cpp**

```c
#include <stdio.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
#define BUF_SIZE 100
int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    //创建套接字
    SOCKET sock = socket(PF_INET, SOCK_DGRAM, 0);
    //服务器地址信息
    sockaddr_in servAddr;
    memset(&servAddr, 0, sizeof(servAddr));  //每个字节都用0填充
    servAddr.sin_family = PF_INET;
    servAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servAddr.sin_port = htons(1234);
    //不断获取用户输入并发送给服务器，然后接受服务器数据
    sockaddr fromAddr;
    int addrLen = sizeof(fromAddr);
    while(1){
        char buffer[BUF_SIZE] = {0};
        printf("Input a string: ");
        gets(buffer);
        sendto(sock, buffer, strlen(buffer), 0, (struct sockaddr*)&servAddr,
 sizeof(servAddr));
        int strLen = recvfrom(sock, buffer, BUF_SIZE, 0, &fromAddr, &addrLe
n);
        buffer[strLen] = 0;
        printf("Message form server: %s\n", buffer);
    }
    closesocket(sock);
    WSACleanup();
    return 0;
}
```