C/C++ socket編程教程：1天玩轉socket通信技術

來源：C語言編程網

socket是什麼意思

在計算機通信領域，socket 被翻譯為“套接字”，它是計算機之間進行通信的一種約定或一種方式。通過 socket 這種約定，一臺計算機可以接收其他計算機的數據，也可以向其他計算機發送數據。

socket 的典型應用就是 Web 服務器和瀏覽器：瀏覽器獲取用戶輸入的URL，向服務器發起請求，服務器分析接收到的URL，將對應的網頁內容返回給瀏覽器，瀏覽器再經過解析和渲染，就將文字、圖片、視頻等元素呈現給用戶。

學習 socket，也就是學習計算機之間如何通信，並編寫出實用的程序。

IP地址（IP Address）

計算機分佈在世界各地，要想和它們通信，必須要知道確切的位置。確定計算機位置的方式有多種，IP 地址是最常用的，例如，114.114.114.114 是國內第一個、全球第三個開放的 DNS 服務地址，127.0.0.1 是本機地址。

其實，我們的計算機並不知道 IP 地址對應的地理位置，當要通信時，只是將 IP 地址封裝到要發送的數據包中，交給路由器去處理。路由器有非常智能和高效的算法，很快就會找到目標計算機，並將數據包傳遞給它，完成一次單向通信。

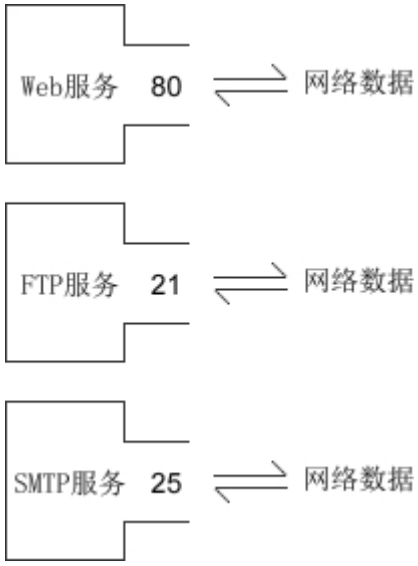
目前大部分軟件使用 IPv4 地址，但 IPv6 也正在被人們接受，尤其是在教育網中，已經大量使用。

端口（Port）

有了 IP 地址，雖然可以找到目標計算機，但仍然不能進行通信。一臺計算機可以同時提供多種網絡服務，例如Web服務、FTP服務（文件傳輸服務）、SMTP服務（郵箱服務）等，僅有 IP 地址，計算機雖然可以正確接收到數據包，但是卻不知道要將數據包交給哪個網絡程序來處理，所以通信失敗。

為了區分不同的網絡程序，計算機會為每個網絡程序分配一個獨一無二的端口號（Port Number），例如，Web服務的端口號是 80，FTP服務的端口號是 21，SMTP 服務的端口號是 25。

端口（Port）是一個虛擬的、邏輯上的概念。可以將端口理解為一道門，數據通過這道門流入流出，每道門有不同的編號，就是端口號。如下圖所示：



協議（Protocol）

協議（Protocol）就是網絡通信的約定，通信的雙方必須都遵守才能正常收發數據。協議有很多種，例如 TCP、UDP、IP 等，通信的雙方

必須使用同一協議才能通信。協議是一種規範，由計算機組織制定，規定了很多細節，例如，如何建立連接，如何相互識別等。

> 協議僅僅是一種規範，必須由計算機軟件來實現。例如 IP 協議規定了如何找到目標計算機，那麼各個開發商在開發自己的軟件時就必須遵守該協議，不能另起爐灶。

所謂協議族（Protocol Family），就是一組協議（多個協議）的統稱。最常用的是 TCP/IP 協議族，它包含了 TCP、IP、UDP、Telnet、FTP、SMTP 等上百個互為關聯的協議，由於 TCP、IP 是兩種常用的底層協議，所以把它們統稱為 TCP/IP 協議族。

數據傳輸方式

計算機之間有很多數據傳輸方式，各有優缺點，常用的有兩種：SOCK_STREAM 和 SOCK_DGRAM。

1) SOCK_STREAM 表示面向連接的數據傳輸方式。數據可以準確無誤地到達另一臺計算機，如果損壞或丟失，可以重新發送，但效率相對較慢。常見的 http 協議就使用 SOCK_STREAM 傳輸數據，因為要確保數據的正確性，否則網頁不能正常解析。

2) SOCK_DGRAM 表示無連接的數據傳輸方式。計算機只管傳輸數據，不作數據校驗，如果數據在傳輸中損壞，或者沒有到達另一臺計算機，是沒有辦法補救的。也就是說，數據錯了就錯了，無法重傳。因為 SOCK_DGRAM 所做的校驗工作少，所以效率比 SOCK_STREAM 高。

QQ 視頻聊天和語音聊天就使用 SOCK_DGRAM 傳輸數據，因為首

先要保證通信的效率，儘量減小延遲，而數據的正確性是次要的，即使丟失很小的一部分數據，視頻和音頻也可以正常解析，最多出現噪點或雜音，不會對通信質量有實質的影響。

> 注意：SOCK_DGRAM 沒有想象中的糟糕，不會頻繁的丟失數據，數據錯誤只是小概率事件。

有可能多種協議使用同一種數據傳輸方式，所以在 socket 編程中，需要同時指明數據傳輸方式和協議。

綜上所述：IP地址和端口能夠在廣袤的互聯網中定位到要通信的程序，協議和數據傳輸方式規定了如何傳輸數據，有了這些，兩臺計算機就可以通信了。

一個簡單的Linux下的socket程序

和C語言教程一樣，我們從一個簡單的“Hello World!”程序切入 socket 編程。

本節演示了 Linux 下的代碼，server.cpp 是服務器端代碼，client.cpp 是客戶端代碼，要實現的功能是：客戶端從服務器讀取一個字符串並打印出來。

服務器端代碼 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(){
    //創建套接字
    int serv_sock = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);

    //將套接字和IP、端口綁定
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每個
字節都用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");  //具體的IP地址
    serv_addr.sin_port = htons(1234);  //端口
```

```
    bind(serv_sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));

    //進入監聽狀態，等待用戶發起請求
    listen(serv_sock, 20);

    //接收客戶端請求
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_size = sizeof(clnt_addr);
    int clnt_sock = accept(serv_sock, (struct
sockaddr*)&clnt_addr, &clnt_addr_size);

    //向客戶端發送數據
    char str[] = "Hello World!";
    write(clnt_sock, str, sizeof(str));

    //關閉套接字
    close(clnt_sock);
    close(serv_sock);

    return 0;
}
```

客戶端代碼 client.cpp：

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

int main(){
    //創建套接字
```

```
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    //向服務器（特定的IP和端口）發起請求
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每個
字節都用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");  //具體的IP地址
    serv_addr.sin_port = htons(1234);  //端口
    connect(sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));

    //讀取服務器傳回的數據
    char buffer[40];
    read(sock, buffer, sizeof(buffer)-1);

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

    //關閉套接字
    close(sock);

    return 0;
}
```

先編譯 server.cpp 並運行：

```
[admin@localhost ~]$ g++ server.cpp -o server
[admin@localhost ~]$ ./server
|
```

正常情況下，程序運行到 accept() 函數就會被阻塞，等待客戶端發起請求。

接下來編譯 client.cpp 並運行：

```
[admin@localhost ~]$ g++ client.cpp -o client
[admin@localhost ~]$ ./client
Message form server: Hello World!
[admin@localhost ~]$
```

client 運行後，通過 connect() 函數向 server 發起請求，處於監聽狀態的 server 被激活，執行 accept() 函數，接受客戶端的請求，然後執行 write() 函數向 client 傳回數據。client 接收到傳回的數據後，connect() 就運行結束了，然後使用 read() 將數據讀取出來。

需要注意的是：
1) server 只接受一次 client 請求，當 server 向 client 傳回數據後，程序就運行結束了。如果想再次接收到服務器的數據，必須再次運行 server，所以這是一個非常簡陋的 socket 程序，不能夠一直接受客戶端的請求。

2) 上面的源文件後綴為`.cpp`，是C++代碼，所以要用`g++`命令來編譯。

> C++和C語言的一個重要區別是：在C語言中，變量必須在函數的開頭定義；而在C++中，變量可以在函數的任何地方定義，使用更加靈活。這裡之所以使用C++代碼，是不希望在函數開頭堆砌過多變量。

源碼解析

1) 先說一下 server.cpp 中的代碼。

第11行通過 socket() 函數創建了一個套接字，參數 AF_INET 表示使用 IPv4 地址，SOCK_STREAM 表示使用面向連接的數據傳輸方式，IPPROTO_TCP 表示使用 TCP 協議。在 Linux 中，socket 也是一種文件，有文件描述符，可以使用 write() / read() 函數進行 I/O 操作。

第19行通過 bind() 函數將套接字 serv_sock 與特定的IP地址和端口綁定，IP地址和端口都保存在 sockaddr_in 結構體中。

socket() 函數確定了套接字的各種屬性，bind() 函數讓套接字與特定的IP地址和端口對應起來，這樣客戶端才能連接到該套接字。

第22行讓套接字處於被動監聽狀態。所謂被動監聽，是指套接字一直處於“睡眠”中，直到客戶端發起請求才會被“喚醒”。

第27行的 accept() 函數用來接收客戶端的請求。程序一旦執行到 accept() 就會被阻塞（暫停運行），直到客戶端發起請求。

第31行的 write() 函數用來向套接字文件中寫入數據，也就是向客戶端發送數據。

和普通文件一樣，socket 在使用完畢後也要用 close() 關閉。

2) 再說一下 client.cpp 中的代碼。client.cpp 中的代碼和 server.cpp 中有一些區別。

第19行代碼通過 connect() 向服務器發起請求，服務器的IP地址和端口號保存在 sockaddr_in 結構體中。直到服務器傳回數據後，connect() 才運行結束。

第23行代碼通過 read() 從套接字文件中讀取數據。

一個簡單的Windows下的socket程序

上節演示了 Linux 下的 socket 程序，這節來看一下 Windows 下的 socket 程序。同樣，server.cpp 為服務器端代碼，client 為客戶端代碼。

服務器端代碼 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加載 ws2_32.dll

int main(){
    //初始化 DLL
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    //綁定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字節都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具體的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
```

```
    //進入監聽狀態
    listen(servSock, 20);

    //接收客戶端請求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock,
(SOCKADDR*)&clntAddr, &nSize);

    //向客戶端發送數據
    char *str = "Hello World!";
    send(clntSock, str, strlen(str)+sizeof(char),
NULL);

    //關閉套接字
    closesocket(clntSock);
    closesocket(servSock);

    //終止 DLL 的使用
    WSACleanup();

    return 0;
}
```

客戶端代碼 client.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加載 ws2_32.dll

int main(){
    //初始化DLL
    WSADATA wsaData;
```

```
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP);

    //向服務器發起請求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字
節都用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));

    //接收服務器傳回的數據
    char szBuffer[MAXBYTE] = {0};
    recv(sock, szBuffer, MAXBYTE, NULL);

    //輸出接收到的數據
    printf("Message form server: %s\n", szBuffer);

    //關閉套接字
    closesocket(sock);

    //終止使用 DLL
    WSACleanup();

    system("pause");
    return 0;
}
```

將 server.cpp 和 client.cpp 分別編譯為 server.exe 和 client.exe，先運行 server.exe，再運行 client.exe，輸出結果為：
Message form server: Hello World!

Windows 下的 socket 程序和 Linux 思路相同，但細節有所差別：
1) Windows 下的 socket 程序依賴 Winsock.dll 或 ws2_32.dll，必須提前加載。DLL 有兩種加載方式，請查看：動態鏈接庫DLL的加載

2) Linux 使用“文件描述符”的概念，而 Windows 使用“文件句柄”的概念；Linux 不區分 socket 文件和普通文件，而 Windows 區分；Linux 下 socket() 函數的返回值為 int 類型，而 Windows 下為 SOCKET 類型，也就是句柄。

3) Linux 下使用 read() / write() 函數讀寫，而 Windows 下使用 recv() / send() 函數發送和接收。

4) 關閉 socket 時，Linux 使用 close() 函數，而 Windows 使用 closesocket() 函數。

WSAStartup()函數以及DLL的加載

> 本節講解 Windows 下 DLL 的加載，學習 Linux Socket 的讀者可以跳過。

WinSock（Windows Socket）編程依賴於系統提供的動態鏈接庫(DLL)，有兩個版本：

- 較早的DLL是 wsock32.dll，大小為 28KB，對應的頭文件為 winsock1.h；
- 最新的DLL是 ws2_32.dll，大小為 69KB，對應的頭文件為 winsock2.h。

幾乎所有的 Windows 操作系統都已經支持 ws2_32.dll，包括個人操作系統 Windows 95 OSR2、Windows 98、Windows Me、Windows 2000、XP、Vista、Win7、Win8、Win10 以及服務器操作系統 Windows NT 4.0 SP4、Windows Server 2003、Windows Server 2008 等，所以你可以毫不猶豫地使用最新的 ws2_32.dll。

使用DLL之前必須把DLL加載到當前程序，你可以在編譯時加載，也可以在程序運行時加載，《C語言高級教程》中講到了這兩種加載方式，請猛擊：動態鏈接庫DLL的加載：隱式加載(載入時加載)和顯式加載(運行時加載)。

這裡使用`#pragma`命令，在編譯時加載：

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

WSAStartup() 函數

使用DLL之前，還需要調用 WSAStartup() 函數進行初始化，以指明 WinSock 規範的版本，它的原型為：

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

wVersionRequested 為 WinSock 規範的版本號，低字節為主版本號，高字節為副版本號（修正版本號）；lpWSAData 為指向 WSAData 結構體的指針。

關於 WinSock 規範

WinSock 規範的最新版本號為 2.2，較早的有 2.1、2.0、1.1、1.0，ws2_32.dll 支持所有的規範，而 wsock32.dll 僅支持 1.0 和 1.1。

wsock32.dll 已經能夠很好的支持 TCP/IP 通信程序的開發，ws2_32.dll 主要增加了對其他協議的支持，不過建議使用最新的 2.2 版本。

wVersionRequested 參數用來指明我們希望使用的版本號，它的類型為 WORD，等價於 unsigned short，是一個整數，所以需要用 MAKEWORD() 宏函數對版本號進行轉換。例如：

```
MAKEWORD(1, 2);  //主版本號為1，副版本號為2，返回 0x0201
MAKEWORD(2, 2);  //主版本號為2，副版本號為2，返回 0x0202
```

關於 WSAData 結構體

WSAStartup() 函數執行成功後，會將與 ws2_32.dll 有關的信息寫入 WSAData 結構體變量。WSAData 的定義如下：

```
typedef struct WSAData {
    WORD           wVersion;  //ws2_32.dll 建議我們使用的版本號
    WORD           wHighVersion;  //ws2_32.dll 支持的最高版本號
    //一個以 null 結尾的字符串，用來說明 ws2_32.dll 的實現以及廠商信息
    char
szDescription[WSADESCRIPTION_LEN+1];
    //一個以 null 結尾的字符串，用來說明 ws2_32.dll 的狀態以及配置信息
    char
szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;  //2.0以後不再使用
    unsigned short iMaxUdpDg;  //2.0以後不再使用
    char FAR       *lpVendorInfo;  //2.0以後不再使用
} WSADATA, *LPWSADATA;
```

最後3個成員已棄之不用，szDescription 和 szSystemStatus 包含的信息基本沒有實用價值，讀者只需關注前兩個成員即可。請看下面的代碼：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    printf("wVersion: %d.%d\n",
```

```
LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));
    printf("wHighVersion: %d.%d\n",
LOBYTE(wsaData.wHighVersion),
HIBYTE(wsaData.wHighVersion));
    printf("szDescription: %s\n",
wsaData.szDescription);
    printf("szSystemStatus: %s\n",
wsaData.szSystemStatus);

    return 0;
}
```

運行結果：

wVersion: 2.2

wHighVersion: 2.2

szDescription: WinSock 2.0

szSystemStatus: Running

ws2_32.dll 支持的最高版本為 2.2，建議使用的版本也是 2.2。

綜上所述：WinSock 編程的第一步就是加載 ws2_32.dll，然後調用 WSAStartup() 函數進行初始化，並指明要使用的版本號。

使用socket()函數創建套接字

在Linux中，一切都是文件，除了文本文件、源文件、二進制文件等，一個硬件設備也可以被映射為一個虛擬的文件，稱為設備文件。例如，stdin 稱為標準輸入文件，它對應的硬件設備一般是鍵盤，stdout 稱為標準輸出文件，它對應的硬件設備一般是顯示器。對於所有的文件，都可以使用 read() 函數讀取數據，使用 write() 函數寫入數據。

“一切都是文件”的思想極大地簡化了程序員的理解和操作，使得對硬件設備的處理就像普通文件一樣。所有在Linux中創建的文件都有一個 int 類型的編號，稱為文件描述符（File Descriptor）。使用文件時，只要知道文件描述符就可以。例如，stdin 的描述符為 0，stdout 的描述符為 1。

在Linux中，socket 也被認為是文件的一種，和普通文件的操作沒有區別，所以在網絡數據傳輸過程中自然可以使用與文件 I/O 相關的函數。可以認為，兩臺計算機之間的通信，實際上是兩個 socket 文件的相互讀寫。

文件描述符有時也被稱為文件句柄（File Handle），但“句柄”主要是 Windows 中術語，所以本教程中如果涉及到 Windows 平臺將使用“句柄”，如果涉及到 Linux 平臺將使用“描述符”。

在Linux下創建 socket

在 Linux 下使用 <sys/socket.h> 頭文件中 socket() 函數來創建套接字，原型為：

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

1) af 為地址族（Address Family），也就是 IP 地址類型，常用的有 AF_INET 和 AF_INET6。AF 是“Address Family”的簡寫，INET 是“Inetnet”的簡寫。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。

大家需要記住`127.0.0.1`，它是一個特殊IP地址，表示本機地址，後面的教程會經常用到。

> 你也可以使用PF前綴，PF是“Protocol Family”的簡寫，它和AF是一樣的。例如，PF_INET 等價於 AF_INET，PF_INET6 等價於 AF_INET6。

2) type 為數據傳輸方式，常用的有 SOCK_STREAM 和 SOCK_DGRAM，在《socket是什麼意思》一節中已經進行了介紹。

3) protocol 表示傳輸協議，常用的有 IPPROTO_TCP 和 IPPTOTO_UDP，分別表示 TCP 傳輸協議和 UDP 傳輸協議。

有了地址類型和數據傳輸方式，還不足以決定採用哪種協議嗎？為什麼還需要第三個參數呢？

正如大家所想，一般情況下有了 af 和 type 兩個參數就可以創建套接字了，操作系統會自動推演出協議類型，除非遇到這樣的情況：有兩種不同的協議支持同一種地址類型和數據傳輸類型。如果我們不指明使用哪種協議，操作系統是沒辦法自動推演的。

該教程使用 IPv4 地址，參數 af 的值為 PF_INET。如果使用 SOCK_STREAM 傳輸數據，那麼滿足這兩個條件的協議只有 TCP，因此可以這樣來調用 socket() 函數：

```
int tcp_socket = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);  //IPPROTO_TCP表示TCP協議
```

這種套接字稱為 TCP 套接字。

如果使用 SOCK_DGRAM 傳輸方式，那麼滿足這兩個條件的協議只有 UDP，因此可以這樣來調用 socket() 函數：

```
int udp_socket = socket(AF_INET, SOCK_DGRAM,
IPPROTO_UDP);  //IPPROTO_UDP表示UDP協議
```

這種套接字稱為 UDP 套接字。

上面兩種情況都只有一種協議滿足條件，可以將 protocol 的值設為 0，系統會自動推演出應該使用什麼協議，如下所示：

```
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0);  //
創建TCP套接字
int udp_socket = socket(AF_INET, SOCK_DGRAM, 0);  //
創建UDP套接字
```

後面的教程中多采用這種簡化寫法。

在Windows下創建socket

Windows 下也使用 socket() 函數來創建套接字，原型為：

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

除了返回值類型不同，其他都是相同的。Windows 不把套接字作為普通文件對待，而是返回 SOCKET 類型的句柄。請看下面的例子：

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

使用bind()和connect()函數

socket() 函數用來創建套接字，確定套接字的各種屬性，然後服務器端要用 bind() 函數將套接字與特定的IP地址和端口綁定起來，只有這樣，流經該IP地址和端口的數據才能交給套接字處理；而客戶端要用 connect() 函數建立連接。

bind() 函數

bind() 函數的原型為：

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

> 下面以Linux為例進行講解，Windows與此類似。

sock 為 socket 文件描述符，addr 為 sockaddr 結構體變量的指針，addrlen 為 addr 變量的大小，可由 sizeof() 計算得出。

下面的代碼，將創建的套接字與IP地址 127.0.0.1、端口 1234 綁定：

```
//創建套接字
int serv_sock = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);

//創建sockaddr_in結構體變量
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每個字節
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");
```

```
//具體的IP地址
serv_addr.sin_port = htons(1234);  //端口

//將套接字和IP、端口綁定
bind(serv_sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));
```

這裡我們使用 sockaddr_in 結構體，然後再強制轉換為 sockaddr 類型，後邊會講解為什麼這樣做。

sockaddr_in 結構體

接下來不妨先看一下 sockaddr_in 結構體，它的成員變量如下：

```
struct sockaddr_in{
    sa_family_t     sin_family;   //地址族（Address
Family），也就是地址類型
    uint16_t        sin_port;     //16位的端口號
    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() 的第一個參數的含義相同，取值也要保持一致。

2) sin_prot 為端口號。uint16_t 的長度為兩個字節，理論上端口號的取值範圍為 0~65536，但 0~1023 的端口一般由系統分配給特定的服務程序，例如 Web 服務的端口號為 80，FTP 服務的端口號為 21，所以我們的程序要儘量在 1024~65536 之間分配端口號。

端口號需要用 htons() 函數轉換，後面會講解為什麼。

3) sin_addr 是 struct in_addr 結構體類型的變量，下面會詳細講解。

4) sin_zero[8] 是多餘的8個字節，沒有用，一般使用 memset() 函數填充為 0。上面的代碼中，先用 memset() 將結構體的全部字節填充為 0，再給前3個成員賦值，剩下的 sin_zero 自然就是 0 了。

in_addr 結構體

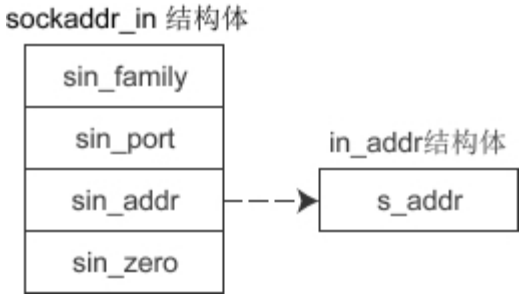
sockaddr_in 的第3個成員是 in_addr 類型的結構體，該結構體只包含一個成員，如下所示：

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

in_addr_t 在頭文件 <netinet/in.h> 中定義，等價於 unsigned long，長度為4個字節。也就是說，s_addr 是一個整數，而IP地址是一個字符串，所以需要 inet_addr() 函數進行轉換，例如：

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

運行結果：
16777343



圖解 sockaddr_in 結構體

為什麼要搞這麼複雜，結構體中嵌套結構體，而不用 sockaddr_in 的一個成員變量來指明IP地址呢？socket() 函數的第一個參數已經指明瞭地址類型，為什麼在 sockaddr_in 結構體中還要再說明一次呢，這不是囉嗦嗎？

這些繁瑣的細節確實給初學者帶來了一定的障礙，我想，這或許是歷史原因吧，後面的接口總要兼容前面的代碼。各位讀者一定要有耐心，暫時不理解沒有關係，根據教程中的代碼“照貓畫虎”即可，時間久了自然會接受。

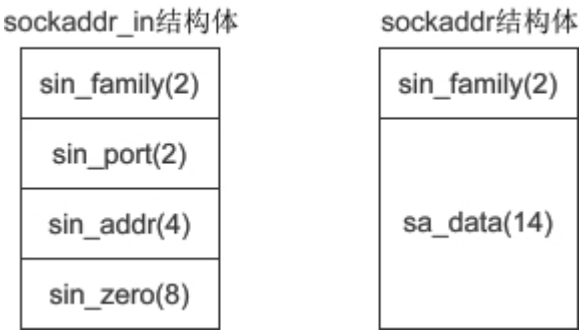
為什麼使用 **sockaddr_in** 而不使用 **sockaddr**

bind() 第二個參數的類型為 sockaddr，而代碼中卻使用 sockaddr_in，然後再強制轉換為 sockaddr，這是為什麼呢？

sockaddr 結構體的定義如下：

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

下圖是 sockaddr 與 sockaddr_in 的對比（括號中的數字表示所佔用的字節數）：



sockaddr 和 sockaddr_in 的長度相同，都是16字節，只是將IP地址和端口號合併到一起，用一個成員 sa_data 表示。要想給 sa_data 賦值，必須同時指明IP地址和端口號，例如"127.0.0.1:80"，遺憾的是，沒有相關函數將這個字符串轉換成需要的形式，也就很難給 sockaddr 類型的變量賦值，所以使用 sockaddr_in 來代替。這兩個結構體的長度相同，強制轉換類型時不會丟失字節，也沒有多餘的字節。

可以認為，sockaddr 是一種通用的結構體，可以用來保存多種類型的IP地址和端口號，而 sockaddr_in 是專門用來保存 IPv4 地址的結構體。另外還有 sockaddr_in6，用來保存 IPv6 地址，它的定義如下：

```
struct sockaddr_in6 {
    sa_family_t sin6_family;  //(2)地址類型，取值為
AF_INET6
    in_port_t sin6_port;  //(2)16位端口號
    uint32_t sin6_flowinfo;  //(4)IPv6流信息
    struct in6_addr sin6_addr;  //(4)具體的IPv6地址
    uint32_t sin6_scope_id;  //(4)接口範圍ID
};
```

正是由於通用結構體 sockaddr 使用不便，才針對不同的地址類型定義了不同的結構體。

connect() 函數

connect() 函數用來建立連接，它的原型為：

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

各個參數的說明和 bind() 相同，不再贅述。

使用listen()和accept()函數

對於服務器端程序，使用 bind() 綁定套接字後，還需要使用 listen() 函數讓套接字進入被動監聽狀態，再調用 accept() 函數，就可以隨時響應客戶端的請求了。

listen() 函數

通過 listen() 函數可以讓套接字進入被動監聽狀態，它的原型為：

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

sock 為需要進入監聽狀態的套接字，backlog 為請求隊列的最大長度。

所謂被動監聽，是指當沒有客戶端請求時，套接字處於“睡眠”狀態，只有當接收到客戶端請求時，套接字才會被“喚醒”來響應請求。

請求隊列

當套接字正在處理客戶端請求時，如果有新的請求進來，套接字是沒法處理的，只能把它放進緩衝區，待當前請求處理完畢後，再從緩衝區中讀取出來處理。如果不斷有新的請求進來，它們就按照先後順序在緩衝區中排隊，直到緩衝區滿。這個緩衝區，就稱為請求隊列（Request Queue）。

緩衝區的長度（能存放多少個客戶端請求）可以通過 listen() 函數的 backlog 參數指定，但究竟為多少並沒有什麼標準，可以根據你的需求來定，併發量小的話可以是10或者20。

如果將 backlog 的值設置為 SOMAXCONN，就由系統來決定請求隊列長度，這個值一般比較大，可能是幾百，或者更多。

當請求隊列滿時，就不再接收新的請求，對於 Linux，客戶端會收到 ECONNREFUSED 錯誤，對於 Windows，客戶端會收到 WSAECONNREFUSED 錯誤。

注意：listen() 只是讓套接字處於監聽狀態，並沒有接收請求。接收請求需要使用 accept() 函數。

accept() 函數

當套接字處於監聽狀態時，可以通過 accept() 函數來接收客戶端請求。它的原型為：

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

它的參數與 listen() 和 connect() 是相同的：sock 為服務器端套接字，addr 為 sockaddr_in 結構體變量，addrlen 為參數 addr 的長度，可由 sizeof() 求得。

accept() 返回一個新的套接字來和客戶端通信，addr 保存了客戶端的IP地址和端口號，而 sock 是服務器端的套接字，大家注意區分。後面和客戶端通信時，要使用這個新生成的套接字，而不是原來服務器端的套接字。

最後需要說明的是：listen() 只是讓套接字進入監聽狀態，並沒有真正接收客戶端請求，listen() 後面的代碼會繼續執行，直到遇到 accept()。accept() 會阻塞程序執行（後面代碼不能被執行），直到有新的請求到來。

socket數據的接收和發送

Linux下數據的接收和發送

Linux 不區分套接字文件和普通文件，使用 write() 可以向套接字中寫入數據，使用 read() 可以從套接字中讀取數據。

前面我們說過，兩臺計算機之間的通信相當於兩個套接字之間的通信，在服務器端用 write() 向套接字寫入數據，客戶端就能收到，然後再使用 read() 從套接字中讀取出來，就完成了一次通信。

write() 的原型為：

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

fd 為要寫入的文件的描述符，buf 為要寫入的數據的緩衝區地址，nbytes 為要寫入的數據的字節數。

> size_t 是通過 typedef 聲明的 unsigned int 類型；ssize_t 在 "size_t" 前面加了一個"s"，代表 signed，即 ssize_t 是通過 typedef 聲明的 signed int 類型。

write() 函數會將緩衝區 buf 中的 nbytes 個字節寫入文件 fd，成功則返回寫入的字節數，失敗則返回 -1。

read() 的原型為：

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

fd 為要讀取的文件的描述符，buf 為要接收數據的緩衝區地址，nbytes 為要讀取的數據的字節數。

read() 函數會從 fd 文件中讀取 nbytes 個字節並保存到緩衝區 buf，成功則返回讀取到的字節數（但遇到文件結尾則返回0），失敗則返回 -1。

Windows下數據的接收和發送

Windows 和 Linux 不同，Windows 區分普通文件和套接字，並定義了專門的接收和發送的函數。

從服務器端發送數據使用 send() 函數，它的原型為：

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

sock 為要發送數據的套接字，buf 為要發送的數據的緩衝區地址，len 為要發送的數據的字節數，flags 為發送數據時的選項。

返回值和前三個參數不再贅述，最後的 flags 參數一般設置為 0 或 NULL，初學者不必深究。

在客戶端接收數據使用 recv() 函數，它的原型為：

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

回聲客戶端的實現

所謂“回聲”，是指客戶端向服務器發送一條數據，服務器再將數據原樣返回給客戶端，就像聲音一樣，遇到障礙物會被“反彈回來”。

對！客戶端也可以使用 write() / send() 函數向服務器發送數據，服務器也可以使用 read() / recv() 函數接收數據。

考慮到大部分初學者使用 Windows 操作系統，本節將實現 Windows 下的回聲程序，Linux 下稍作修改即可，不再給出代碼。

服務器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加載
ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM,
0);

    //綁定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字
```

```
節都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr =
inet_addr("127.0.0.1");  //具體的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr,
sizeof(SOCKADDR));

    //進入監聽狀態
    listen(servSock, 20);

    //接收客戶端請求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    SOCKET clntSock = accept(servSock,
(SOCKADDR*)&clntAddr, &nSize);
    char buffer[BUF_SIZE];  //緩衝區
    int strLen = recv(clntSock, buffer, BUF_SIZE, 0);
//接收客戶端發來的數據
    send(clntSock, buffer, strLen, 0);  //將數據原樣返
回

    //關閉套接字
    closesocket(clntSock);
    closesocket(servSock);

    //終止 DLL 的使用
    WSACleanup();

    return 0;
}
```

客戶端 client.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加載 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP);

    //向服務器發起請求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字
節都用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));
    //獲取用戶輸入的字符串併發送給服務器
    char bufSend[BUF_SIZE] = {0};
    printf("Input a string: ");
    scanf("%s", bufSend);
    send(sock, bufSend, strlen(bufSend), 0);
    //接收服務器傳回的數據
    char bufRecv[BUF_SIZE] = {0};
    recv(sock, bufRecv, BUF_SIZE, 0);
```

```
    //輸出接收到的數據
    printf("Message form server: %s\n", bufRecv);

    //關閉套接字
    closesocket(sock);

    //終止使用 DLL
    WSACleanup();

    system("pause");
    return 0;
}
```

先運行服務器端，再運行客戶端，執行結果為：

Input a string: c-language java cpp↙

Message form server: c-language

scanf() 讀取到空格時認為一個字符串輸入結束，所以只能讀取到“c-language”；如果不希望把空格作為字符串的結束符，可以使用 gets() 函數。

通過本程序可以發現，客戶端也可以向服務器端發送數據，這樣服務器端就可以根據不同的請求作出不同的響應，http 服務器就是典型的例子，請求的網址不同，返回的頁面也不同。

實現迭代服務器端和客戶端

前面的程序，不管服務器端還是客戶端，都有一個問題，就是處理完一個請求立即退出了，沒有太大的實際意義。能不能像Web服務器那樣一直接受客戶端的請求呢？能，使用 while 循環即可。

修改前面的回聲程序，使服務器端可以不斷響應客戶端的請求。

服務器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加載 ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM, 0);

    //綁定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字節都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具體的IP地址
```

```
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr,
sizeof(SOCKADDR));

    //進入監聽狀態
    listen(servSock, 20);

    //接收客戶端請求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE] = {0};  //緩衝區
    while(1){
        SOCKET clntSock = accept(servSock,
(SOCKADDR*)&clntAddr, &nSize);
        int strLen = recv(clntSock, buffer, BUF_SIZE,
0);  //接收客戶端發來的數據
        send(clntSock, buffer, strLen, 0);  //將數據原
樣返回

        closesocket(clntSock);  //關閉套接字
        memset(buffer, 0, BUF_SIZE);  //重置緩衝區
    }

    //關閉套接字
    closesocket(servSock);

    //終止 DLL 的使用
    WSACleanup();

    return 0;
}
```

客戶端 client.cpp：

```
#include <stdio.h>
#include <WinSock2.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")  //加載 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //向服務器發起請求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字節都用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){
        //創建套接字
        SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
        //獲取用戶輸入的字符串併發送給服務器
        printf("Input a string: ");
        gets(bufSend);
        send(sock, bufSend, strlen(bufSend), 0);
        //接收服務器傳回的數據
```

```
        recv(sock, bufRecv, BUF_SIZE, 0);
        //輸出接收到的數據
        printf("Message form server: %s\n", bufRecv);

        memset(bufSend, 0, BUF_SIZE);  //重置緩衝區
        memset(bufRecv, 0, BUF_SIZE);  //重置緩衝區
        closesocket(sock);  //關閉套接字
    }

    WSACleanup();  //終止使用 DLL
    return 0;
}
```

先運行服務器端，再運行客戶端，結果如下：

Input a string: c language

Message form server: c language

Input a string: C語言中文網

Message form server: C語言中文網

Input a string: 學習C/C++編程的好網站

Message form server: 學習C/C++編程的好網站

while(1) 讓代碼進入死循環，除非用戶關閉程序，否則服務器端會一直監聽客戶端的請求。客戶端也是一樣，會不斷向服務器發起連接。

需要注意的是：server.cpp 中調用 closesocket() 不僅會關閉服務器端的 socket，還會通知客戶端連接已斷開，客戶端也會清理 socket 相關資源，所以 client.cpp 中需要將 socket() 放在 while 循環內部，因為每次請求完畢都會清理 socket，下次發起請求時需要重新創建。後續我們會進行詳細講解。

socket緩衝區以及阻塞模式

在《socket數據的接收和發送》一節中講到，可以使用 write()/send() 函數發送數據，使用 read()/recv() 函數接收數據，本節就來看看數據是如何傳遞的。

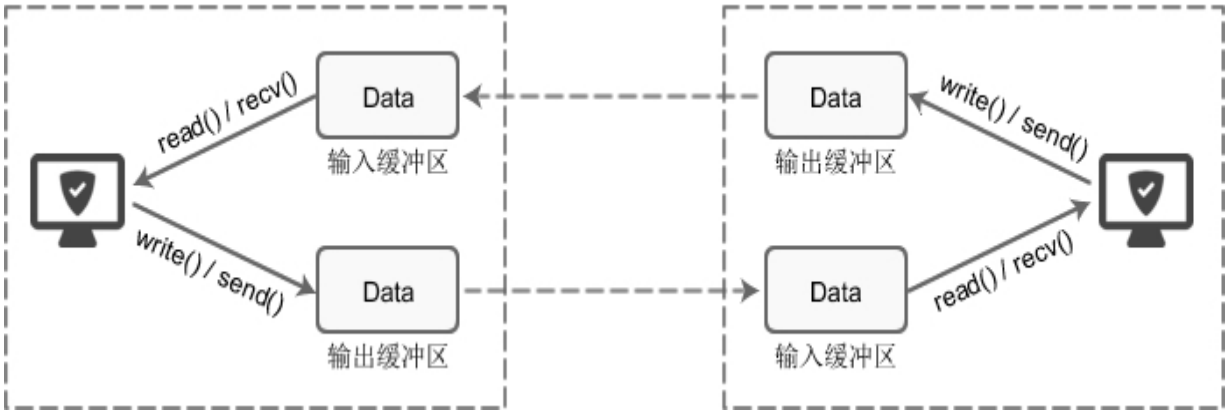
socket緩衝區

每個 socket 被創建後，都會分配兩個緩衝區，輸入緩衝區和輸出緩衝區。

write()/send() 並不立即向網絡中傳輸數據，而是先將數據寫入緩衝區中，再由TCP協議將數據從緩衝區發送到目標機器。一旦將數據寫入到緩衝區，函數就可以成功返回，不管它們有沒有到達目標機器，也不管它們何時被髮送到網絡，這些都是TCP協議負責的事情。

TCP協議獨立於 write()/send() 函數，數據有可能剛被寫入緩衝區就發送到網絡，也可能在緩衝區中不斷積壓，多次寫入的數據被一次性發送到網絡，這取決於當時的網絡情況、當前線程是否空閒等諸多因素，不由程序員控制。

read()/recv() 函數也是如此，也從輸入緩衝區中讀取數據，而不是直接從網絡中讀取。



圖：TCP套接字的I/O緩衝區示意圖

這些I/O緩衝區特性可整理如下：

- I/O緩衝區在每個TCP套接字中單獨存在；
- I/O緩衝區在創建套接字時自動生成；
- 即使關閉套接字也會繼續傳送輸出緩衝區中遺留的數據；
- 關閉套接字將丟失輸入緩衝區中的數據。

輸入輸出緩衝區的默認大小一般都是 8K，可以通過 getsockopt() 函數獲取：

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

運行結果：

Buffer length: 8192

> 這裡僅給出示例，後面會詳細講解。

阻塞模式

對於TCP套接字（默認情況下），當使用 write()/send() 發送數據時：
1) 首先會檢查緩衝區，如果緩衝區的可用空間長度小於要發送的數據，那麼 write()/send() 會被阻塞（暫停執行），直到緩衝區中的數據被髮送到目標機器，騰出足夠的空間，才喚醒 write()/send() 函數繼續寫入數據。

2) 如果TCP協議正在向網絡發送數據，那麼輸出緩衝區會被鎖定，不允許寫入，write()/send() 也會被阻塞，直到數據發送完畢緩衝區解鎖，write()/send() 才會被喚醒。

3) 如果要寫入的數據大於緩衝區的最大長度，那麼將分批寫入。

4) 直到所有數據被寫入緩衝區 write()/send() 才能返回。

當使用 read()/recv() 讀取數據時：
1) 首先會檢查緩衝區，如果緩衝區中有數據，那麼就讀取，否則函數會被阻塞，直到網絡上有數據到來。

2) 如果要讀取的數據長度小於緩衝區中的數據長度，那麼就不能一次性將緩衝區中的所有數據讀出，剩餘數據將不斷積壓，直到有 read()/recv() 函數再次讀取。

3) 直到讀取到數據後 read()/recv() 函數才會返回，否則就一直被阻塞。

這就是TCP套接字的阻塞模式。所謂阻塞，就是上一步動作沒有完成，下一步動作將暫停，直到上一步動作完成後才能繼續，以保持同步性。

> TCP套接字默認情況下是阻塞模式，也是最常用的。當然你也可以更改為非阻塞模式，後續我們會講解。

TCP的粘包問題以及數據的無邊界性

上節我們講到了socket緩衝區和數據的傳遞過程，可以看到數據的接收和發送是無關的，read()/recv() 函數不管數據發送了多少次，都會盡可能多的接收數據。也就是說，read()/recv() 和 write()/send() 的執行次數可能不同。

例如，write()/send() 重複執行三次，每次都發送字符串"abc"，那麼目標機器上的 read()/recv() 可能分三次接收，每次都接收"abc"；也可能分兩次接收，第一次接收"abcab"，第二次接收"cabc"；也可能一次就接收到字符串"abcabcabc"。

假設我們希望客戶端每次發送一位學生的學號，讓服務器端返回該學生的姓名、住址、成績等信息，這時候可能就會出現問題，服務器端不能區分學生的學號。例如第一次發送 1，第二次發送 3，服務器可能當成 13 來處理，返回的信息顯然是錯誤的。

這就是數據的“粘包”問題，客戶端發送的多個數據包被當做一個數據包接收。也稱數據的無邊界性，read()/recv() 函數不知道數據包的開始或結束標誌（實際上也沒有任何開始或結束標誌），只把它們當做連續的數據流來處理。

下面的代碼演示了粘包問題，客戶端連續三次向服務器端發送數據，服務器端卻一次性接收到所有數據。

服務器端代碼 server.cpp：

```
#include <stdio.h>
#include <windows.h>
#pragma comment (lib, "ws2_32.lib")  //加載
ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET servSock = socket(AF_INET, SOCK_STREAM,
0);

    //綁定套接字
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字
節都用0填充
    sockAddr.sin_family = PF_INET;  //使用IPv4地址
    sockAddr.sin_addr.s_addr =
inet_addr("127.0.0.1");  //具體的IP地址
    sockAddr.sin_port = htons(1234);  //端口
    bind(servSock, (SOCKADDR*)&sockAddr,
sizeof(SOCKADDR));

    //進入監聽狀態
    listen(servSock, 20);

    //接收客戶端請求
    SOCKADDR clntAddr;
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE] = {0};  //緩衝區
    SOCKET clntSock = accept(servSock,
(SOCKADDR*)&clntAddr, &nSize);
```

```
    Sleep(10000);  //注意這裡，讓程序暫停10秒

    //接收客戶端發來的數據，並原樣返回
    int recvLen = recv(clntSock, buffer, BUF_SIZE,
0);
    send(clntSock, buffer, recvLen, 0);

    //關閉套接字並終止DLL的使用
    closesocket(clntSock);
    closesocket(servSock);
    WSACleanup();

    return 0;
}
```

客戶端代碼 client.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")  //加載 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //向服務器發起請求
    sockaddr_in sockAddr;
    memset(&sockAddr, 0, sizeof(sockAddr));  //每個字
```

```
節都用0填充
    sockAddr.sin_family = PF_INET;
    sockAddr.sin_addr.s_addr =
inet_addr("127.0.0.1");
    sockAddr.sin_port = htons(1234);

    //創建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP);
    connect(sock, (SOCKADDR*)&sockAddr,
sizeof(SOCKADDR));

    //獲取用戶輸入的字符串併發送給服務器
    char bufSend[BUF_SIZE] = {0};
    printf("Input a string: ");
    gets(bufSend);
    for(int i=0; i<3; i++){
        send(sock, bufSend, strlen(bufSend), 0);
    }
    //接收服務器傳回的數據
    char bufRecv[BUF_SIZE] = {0};
    recv(sock, bufRecv, BUF_SIZE, 0);
    //輸出接收到的數據
    printf("Message form server: %s\n", bufRecv);

    closesocket(sock);  //關閉套接字
    WSACleanup();  //終止使用 DLL

    system("pause");
    return 0;
}
```

先運行 server，再運行 client，並在10秒內輸入字符串"abc"，再等數秒，服務器就會返回數據。運行結果如下：

Input a string: abc
Message form server: abcabcabc

本程序的關鍵是 server.cpp 第31行的代碼`Sleep(10000);`，它讓程序暫停執行10秒。在這段時間內，client 連續三次發送字符串"abc"，由於 server 被阻塞，數據只能堆積在緩衝區中，10秒後，server 開始運行，從緩衝區中一次性讀出所有積壓的數據，並返回給客戶端。

另外還需要說明的是 client.cpp 第34行代碼。client 執行到 recv() 函數，由於輸入緩衝區中沒有數據，所以會被阻塞，直到10秒後 server 傳回數據才開始執行。用戶看到的直觀效果就是，client 暫停一段時間才輸出 server 返回的結果。

client 的 send() 發送了三個數據包，而 server 的 recv() 卻只接收到一個數據包，這很好的說明了數據的粘包問題。

TCP數據報結構以及三次握手（圖解）

TCP（Transmission Control Protocol，傳輸控制協議）是一種面向連接的、可靠的、基於字節流的通信協議，數據在傳輸前要建立連接，傳輸完畢後還要斷開連接。

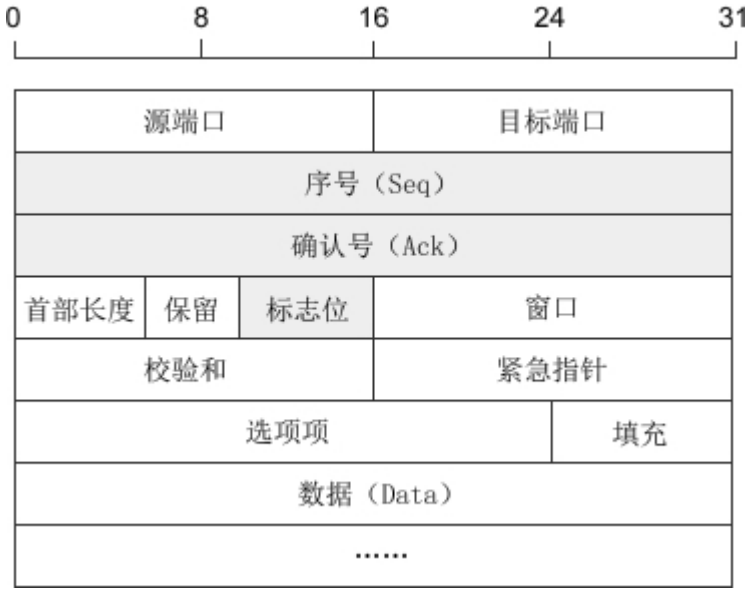
客戶端在收發數據前要使用 connect() 函數和服務器建立連接。建立連接的目的是保證IP地址、端口、物理鏈路等正確無誤，為數據的傳輸開闢通道。

TCP建立連接時要傳輸三個數據包，俗稱三次握手（Three-way Handshaking）。可以形象的比喻為下面的對話：

- [Shake 1] 套接字A：“你好，套接字B，我這裡有數據要傳送給你，建立連接吧。”
- [Shake 2] 套接字B：“好的，我這邊已準備就緒。”
- [Shake 3] 套接字A：“謝謝你受理我的請求。”

TCP數據報結構

我們先來看一下TCP數據報的結構：



帶陰影的幾個字段需要重點說明一下：

1) 序號：Seq（Sequence Number）序號佔32位，用來標識從計算機A發送到計算機B的數據包的序號，計算機發送數據時對此進行標記。

2) 確認號：Ack（Acknowledge Number）確認號佔32位，客戶端和服務器端都可以發送，Ack = Seq + 1。

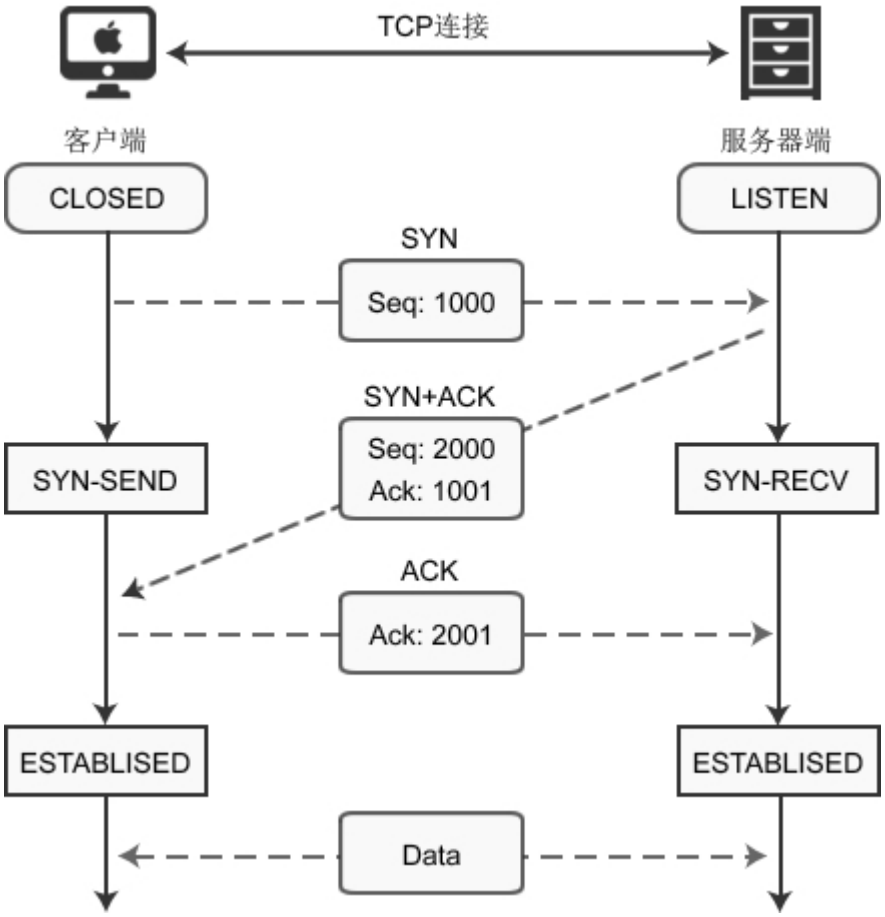
3) 標誌位：每個標誌位佔用1Bit，共有6個，分別為 URG、ACK、PSH、RST、SYN、FIN，具體含義如下：

- URG：緊急指針（urgent pointer）有效。
- ACK：確認序號有效。
- PSH：接收方應該儘快將這個報文交給應用層。
- RST：重置連接。
- SYN：建立一個新連接。
- FIN：斷開一個連接。

> 對英文字母縮寫的總結：Seq 是 Sequence 的縮寫，表示序列；Ack(ACK) 是 Acknowledge 的縮寫，表示確認；SYN 是 Synchronous 的縮寫，願意是“同步的”，這裡表示建立同步連接；FIN 是 Finish 的縮寫，表示完成。

連接的建立（三次握手）

使用 connect() 建立連接時，客戶端和服務器端會相互發送三個數據包，請看下圖：



客戶端調用 socket() 函數創建套接字後，因為沒有建立連接，所以套接字處於`CLOSED`狀態；服務器端調用 listen() 函數後，套接字進入`LISTEN`狀態，開始監聽客戶端請求。

這個時候，客戶端開始發起請求：
1) 當客戶端調用 connect() 函數後，TCP協議會組建一個數據包，並設置 SYN 標誌位，表示該數據包是用來建立同步連接的。同時生成一個隨機數字 1000，填充“序號（Seq）”字段，表示該數據包的序號。完成這些工作，開始向服務器端發送數據包，客戶端就進入了`SYN-SEND`狀態。

2) 服務器端收到數據包，檢測到已經設置了 SYN 標誌位，就知道這是客戶端發來的建立連接的“請求包”。服務器端也會組建一個數據包，並設置 SYN 和 ACK 標誌位，SYN 表示該數據包用來建立連接，ACK 用來確認收到了剛才客戶端發送的數據包。

服務器生成一個隨機數 2000，填充“序號（Seq）”字段。2000 和客戶端數據包沒有關係。

服務器將客戶端數據包序號（1000）加1，得到1001，並用這個數字填充“確認號（Ack）”字段。

服務器將數據包發出，進入`SYN-RECV`狀態。

3) 客戶端收到數據包，檢測到已經設置了 SYN 和 ACK 標誌位，就知道這是服務器發來的“確認包”。客戶端會檢測“確認號（Ack）”字段，看它的值是否為 1000+1，如果是就說明連接建立成功。

接下來，客戶端會繼續組建數據包，並設置 ACK 標誌位，表示客戶端正確接收了服務器發來的“確認包”。同時，將剛才服務器發來的數

據包序號（2000）加1，得到 2001，並用這個數字來填充“確認號（Ack）”字段。

客戶端將數據包發出，進入`ESTABLISED`狀態，表示連接已經成功建立。

4) 服務器端收到數據包，檢測到已經設置了 ACK 標誌位，就知道這是客戶端發來的“確認包”。服務器會檢測“確認號（Ack）”字段，看它的值是否為 2000+1，如果是就說明連接建立成功，服務器進入`ESTABLISED`狀態。

至此，客戶端和服務器都進入了`ESTABLISED`狀態，連接建立成功，接下來就可以收發數據了。

最後的說明

三次握手的關鍵是要確認對方收到了自己的數據包，這個目標就是通過“確認號（Ack）”字段實現的。計算機會記錄下自己發送的數據包序號 Seq，待收到對方的數據包後，檢測“確認號（Ack）”字段，看`Ack = Seq + 1`是否成立，如果成立說明對方正確收到了自己的數據包。

TCP數據的傳輸過程

建立連接後，兩臺主機就可以相互傳輸數據了。如下圖所示：

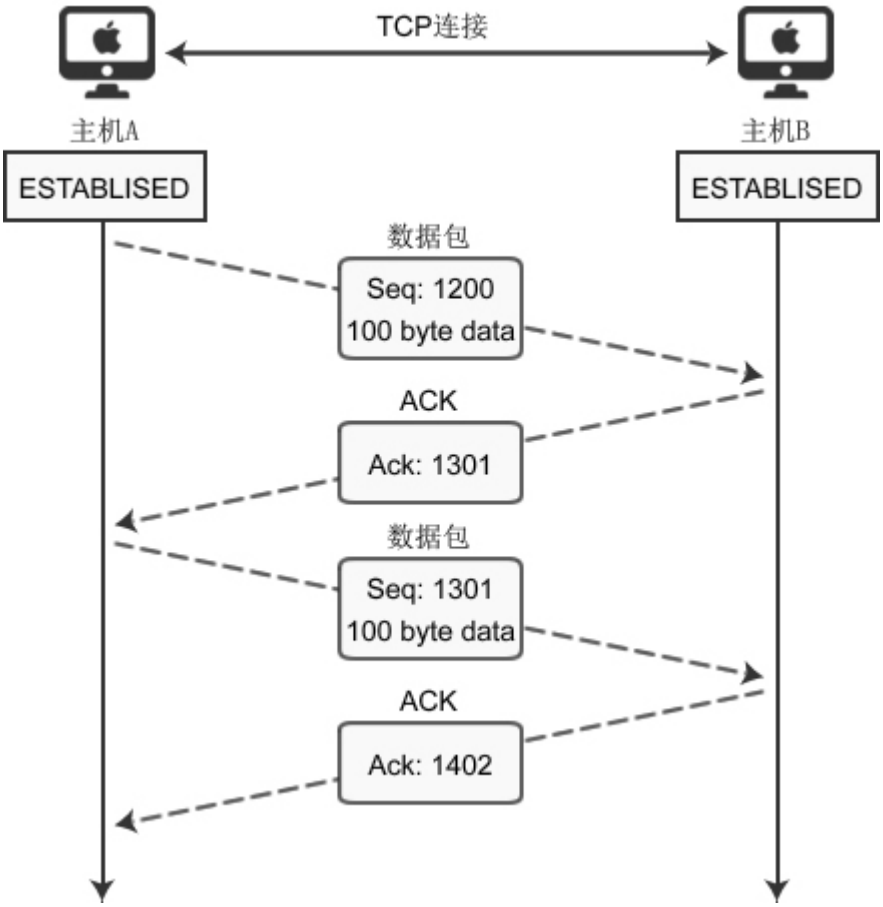


圖1：TCP 套接字的數據交換過程

上圖給出了主機A分2次（分2個數據包）向主機B傳遞200字節的過程。首先，主機A通過1個數據包發送100個字節的數據，數據包的 Seq 號設置為 1200。主機B為了確認這一點，向主機A發送 ACK 包，並將 Ack 號設置為 1301。

> 為了保證數據準確到達，目標機器在收到數據包（包括SYN包、FIN包、普通數據包等）包後必須立即回傳ACK包，這樣發送方才

> 能確認數據傳輸成功。

此時 Ack 號為 1301 而不是 1201，原因在於 Ack 號的增量為傳輸的數據字節數。假設每次 Ack 號不加傳輸的字節數，這樣雖然可以確認數據包的傳輸，但無法明確100字節全部正確傳遞還是丟失了一部分，比如只傳遞了80字節。因此按如下的公式確認 Ack 號：

Ack號 = Seq號 + 傳遞的字節數 + 1

與三次握手協議相同，最後加 1 是為了告訴對方要傳遞的 Seq 號。

下面分析傳輸過程中數據包丟失的情況，如下圖所示：

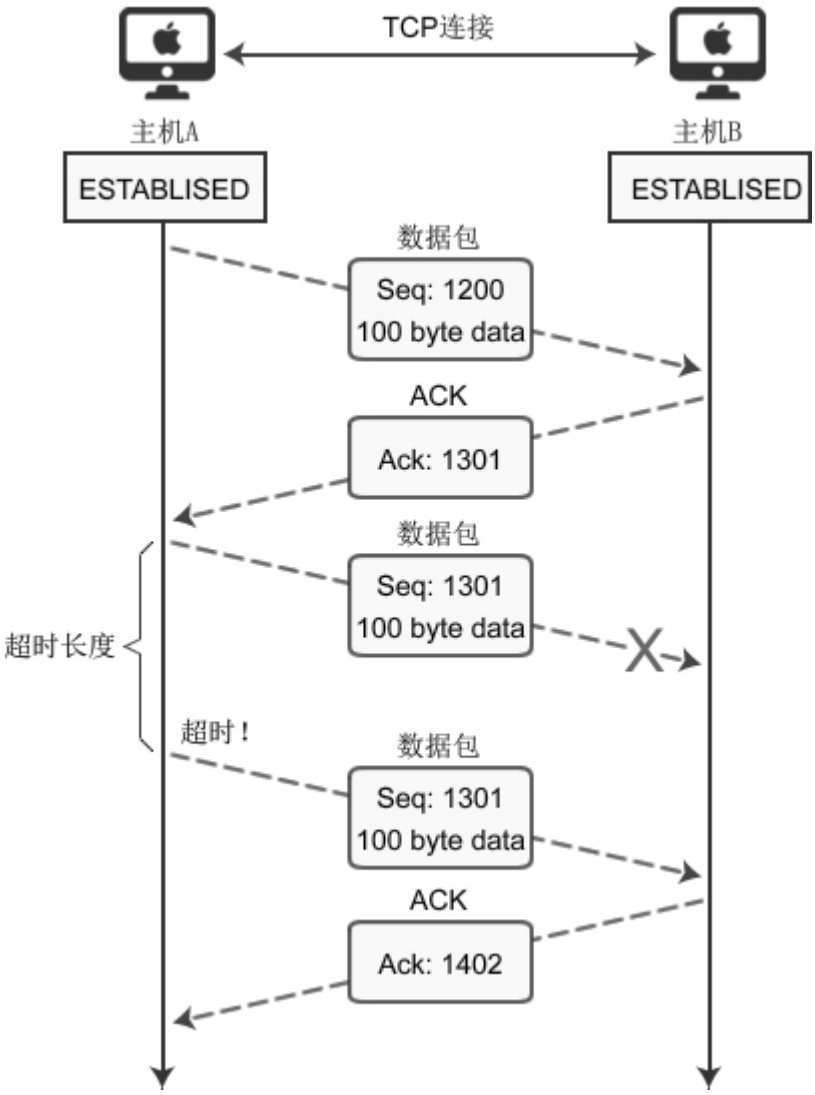


圖2：TCP套接字數據傳輸過程中發生錯誤

上圖表示通過 Seq 1301 數據包向主機B傳遞100字節的數據，但中間發生了錯誤，主機B未收到。經過一段時間後，主機A仍未收到對於 Seq 1301 的ACK確認，因此嘗試重傳數據。

為了完成數據包的重傳，TCP套接字每次發送數據包時都會啟動定時器，如果在一定時間內沒有收到目標機器傳回的 ACK 包，那麼定時器超時，數據包會重傳。

> 上圖演示的是數據包丟失的情況，也會有 ACK 包丟失的情況，一樣會重傳。

重傳超時時間（RTO, Retransmission Time Out）

這個值太大了會導致不必要的等待，太小會導致不必要的重傳，理論上最好是網絡 RTT 時間，但又受制於網絡距離與瞬態時延變化，所以實際上使用自適應的動態算法（例如 Jacobson 算法和 Karn 算法等）來確定超時時間。

> 往返時間（RTT，Round-Trip Time）表示從發送端發送數據開始，到發送端收到來自接收端的 ACK 確認包（接收端收到數據後便立即確認），總共經歷的時延。

重傳次數

TCP數據包重傳次數根據系統設置的不同而有所區別。有些系統，一個數據包只會被重傳3次，如果重傳3次後還未收到該數據包的 ACK 確認，就不再嘗試重傳。但有些要求很高的業務系統，會不斷地重傳丟失的數據包，以盡最大可能保證業務數據的正常交互。

最後需要說明的是，發送端只有在收到對方的 ACK 確認包後，才會清空輸出緩衝區中的數據。

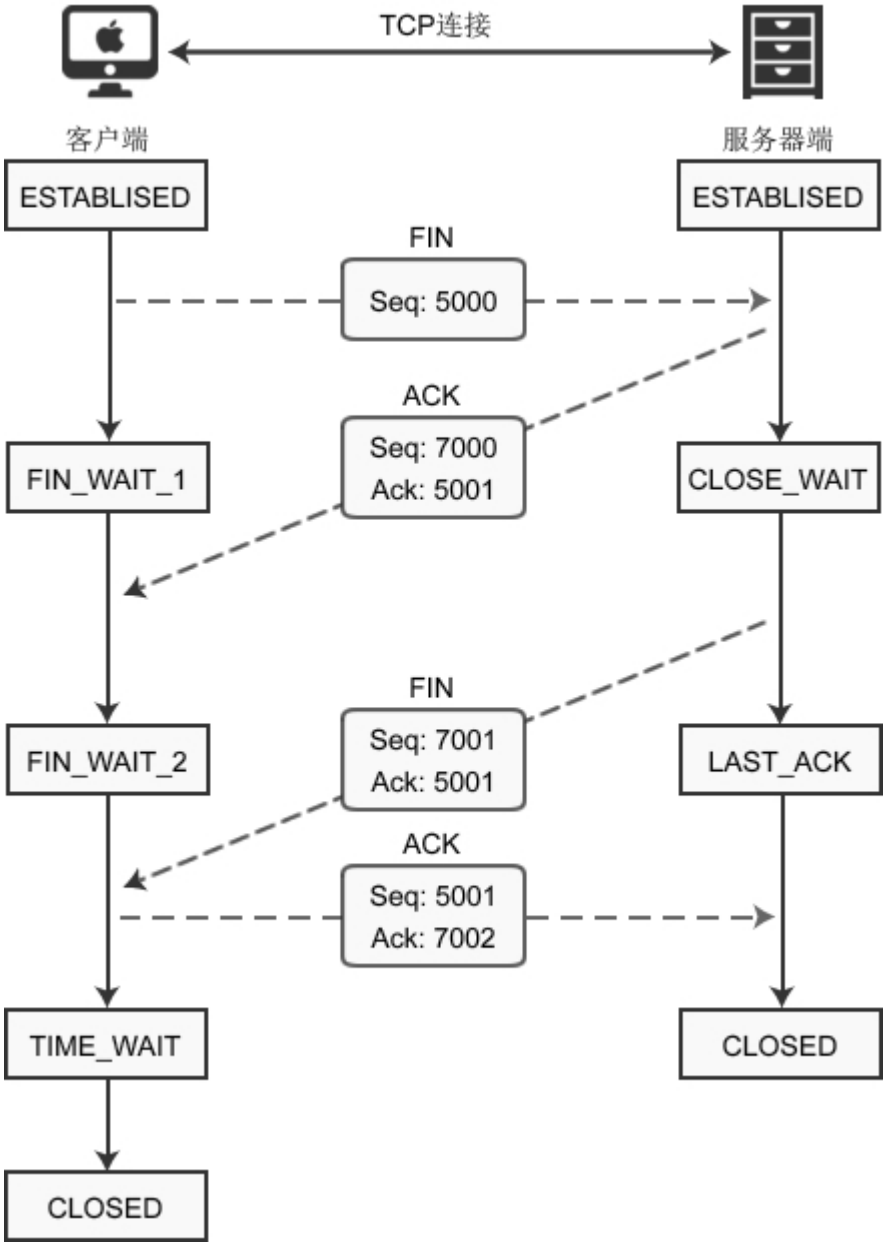
TCP四次握手斷開連接（圖解）

建立連接非常重要，它是數據正確傳輸的前提；斷開連接同樣重要，它讓計算機釋放不再使用的資源。如果連接不能正常斷開，不僅會造成數據傳輸錯誤，還會導致套接字不能關閉，持續佔用資源，如果併發量高，服務器壓力堪憂。

建立連接需要三次握手，斷開連接需要四次握手，可以形象的比喻為下面的對話：

- [Shake 1] 套接字A：“任務處理完畢，我希望斷開連接。”
- [Shake 2] 套接字B：“哦，是嗎？請稍等，我準備一下。”
- 等待片刻後……
- [Shake 3] 套接字B：“我準備好了，可以斷開連接了。”
- [Shake 4] 套接字A：“好的，謝謝合作。”

下圖演示了客戶端主動斷開連接的場景：



建立連接後，客戶端和服務器都處於`ESTABLISED`狀態。這時，客戶端發起斷開連接的請求：

1) 客戶端調用 close() 函數後，向服務器發送 FIN 數據包，進入`FIN_WAIT_1`狀態。FIN 是 Finish 的縮寫，表示完成任務需要斷開連接。

2) 服務器收到數據包後，檢測到設置了 FIN 標誌位，知道要斷開連接，於是向客戶端發送“確認包”，進入`CLOSE_WAIT`狀態。

注意：服務器收到請求後並不是立即斷開連接，而是先向客戶端發送“確認包”，告訴它我知道了，我需要準備一下才能斷開連接。

3) 客戶端收到“確認包”後進入`FIN_WAIT_2`狀態，等待服務器準備完畢後再次發送數據包。

4) 等待片刻後，服務器準備完畢，可以斷開連接，於是再主動向客戶端發送 FIN 包，告訴它我準備好了，斷開連接吧。然後進入`LAST_ACK`狀態。

5) 客戶端收到服務器的 FIN 包後，再向服務器發送 ACK 包，告訴它你斷開連接吧。然後進入`TIME_WAIT`狀態。

6) 服務器收到客戶端的 ACK 包後，就斷開連接，關閉套接字，進入`CLOSED`狀態。

關於 TIME_WAIT 狀態的說明

客戶端最後一次發送 ACK包後進入 TIME_WAIT 狀態，而不是直接進入 CLOSED 狀態關閉連接，這是為什麼呢？

TCP 是面向連接的傳輸方式，必須保證數據能夠正確到達目標機器，不能丟失或出錯，而網絡是不穩定的，隨時可能會毀壞數據，所以機器A每次向機器B發送數據包後，都要求機器B”確認“，回傳ACK包，告訴機器A我收到了，這樣機器A才能知道數據傳送成功了。如果機器

B沒有回傳ACK包，機器A會重新發送，直到機器B回傳ACK包。

客戶端最後一次向服務器回傳ACK包時，有可能會因為網絡問題導致服務器收不到，服務器會再次發送 FIN 包，如果這時客戶端完全關閉了連接，那麼服務器無論如何也收不到ACK包了，所以客戶端需要等待片刻、確認對方收到ACK包後才能進入CLOSED狀態。那麼，要等待多久呢？

數據包在網絡中是有生存時間的，超過這個時間還未到達目標主機就會被丟棄，並通知源主機。這稱為報文最大生存時間（MSL，Maximum Segment Lifetime）。TIME_WAIT 要等待 2MSL 才會進入 CLOSED 狀態。ACK 包到達服務器需要 MSL 時間，服務器重傳 FIN 包也需要 MSL 時間，2MSL 是數據包往返的最大時間，如果 2MSL 後還未收到服務器重傳的 FIN 包，就說明服務器已經收到了 ACK 包。

優雅的斷開連接--shutdown()

調用 close()/closesocket() 函數意味著完全斷開連接，即不能發送數據也不能接收數據，這種“生硬”的方式有時候會顯得不太“優雅”。

圖1：close()/closesocket() 斷開連接

上圖演示了兩臺正在進行雙向通信的主機。主機A發送完數據後，單方面調用 close()/closesocket() 斷開連接，之後主機A、B都不能再接受對方傳輸的數據。實際上，是完全無法調用與數據收發有關的函數。

一般情況下這不會有問題，但有些特殊時刻，需要只斷開一條數據傳輸通道，而保留另一條。

使用 shutdown() 函數可以達到這個目的，它的原型為：

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

sock 為需要斷開的套接字，howto 為斷開方式。

howto 在 Linux 下有以下取值：

- SHUT_RD：斷開輸入流。套接字無法接收數據（即使輸入緩衝區收到數據也被抹去），無法調用輸入相關函數。

- SHUT_WR：斷開輸出流。套接字無法發送數據，但如果輸出緩衝區中還有未傳輸的數據，則將傳遞到目標主機。
- SHUT_RDWR：同時斷開 I/O 流。相當於分兩次調用 shutdown()，其中一次以 SHUT_RD 為參數，另一次以 SHUT_WR 為參數。

howto 在 Windows 下有以下取值：

- SD_RECEIVE：關閉接收操作，也就是斷開輸入流。
- SD_SEND：關閉發送操作，也就是斷開輸出流。
- SD_BOTH：同時關閉接收和發送操作。

至於什麼時候需要調用 shutdown() 函數，下節我們會以文件傳輸為例進行講解。

close()/closesocket()和shutdown()的區別

確切地說，close() / closesocket() 用來關閉套接字，將套接字描述符（或句柄）從內存清除，之後再也不能使用該套接字，與C語言中的 fclose() 類似。應用程序關閉套接字後，與該套接字相關的連接和緩存也失去了意義，TCP協議會自動觸發關閉連接的操作。

shutdown() 用來關閉連接，而不是套接字，不管調用多少次 shutdown()，套接字依然存在，直到調用 close() / closesocket() 將套接字從內存清除。

調用 close()/closesocket() 關閉套接字時，或調用 shutdown() 關閉輸

出流時，都會向對方發送 FIN 包。FIN 包表示數據傳輸完畢，計算機收到 FIN 包就知道不會再有數據傳送過來了。

默認情況下，close()/closesocket() 會立即向網絡中發送FIN包，不管輸出緩衝區中是否還有數據，而shutdown() 會等輸出緩衝區中的數據傳輸完畢再發送FIN包。也就意味著，調用 close()/closesocket() 將丟失輸出緩衝區中的數據，而調用 shutdown() 不會。

socket文件傳輸功能的實現

這節我們來完成 socket 文件傳輸程序，這是一個非常實用的例子。要實現的功能為：client 從 server 下載一個文件並保存到本地。

編寫這個程序需要注意兩個問題：
1) 文件大小不確定，有可能比緩衝區大很多，調用一次 write()/send() 函數不能完成文件內容的發送。接收數據時也會遇到同樣的情況。

要解決這個問題，可以使用 while 循環，例如：

```
//Server 代碼
int nCount;
while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0
){
    send(sock, buffer, nCount, 0);
}

//Client 代碼
int nCount;
while( (nCount = recv(clntSock, buffer, BUF_SIZE, 0))
> 0 ){
    fwrite(buffer, nCount, 1, fp);
}
```

對於 Server 端的代碼，當讀取到文件末尾，fread() 會返回 0，結束循環。

對於 Client 端代碼，有一個關鍵的問題，就是文件傳輸完畢後讓 recv() 返回 0，結束 while 循環。

> 注意：讀取完緩衝區中的數據 recv() 並不會返回 0，而是被阻塞，直到緩衝區中再次有數據。

2) Client 端如何判斷文件接收完畢，也就是上面提到的問題——何時結束 while 循環。

最簡單的結束 while 循環的方法當然是文件接收完畢後讓 recv() 函數返回 0，那麼，如何讓 recv() 返回 0 呢？recv() 返回 0 的唯一時機就是收到FIN包時。

FIN 包表示數據傳輸完畢，計算機收到 FIN 包後就知道對方不會再向自己傳輸數據，當調用 read()/recv() 函數時，如果緩衝區中沒有數據，就會返回 0，表示讀到了”socket文件的末尾“。

這裡我們調用 shutdown() 來發送FIN包：server 端直接調用 close()/closesocket() 會使輸出緩衝區中的數據失效，文件內容很有可能沒有傳輸完畢連接就斷開了，而調用 shutdown() 會等待輸出緩衝區中的數據傳輸完畢。

本節以Windows為例演示文件傳輸功能，Linux與此類似，不再贅述。請看下面完整的代碼。

服務器端 server.cpp：

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加載 ws2_32.dll
```

```
#define BUF_SIZE 1024

int main(){
    //先檢查文件是否存在
    char *filename = "D:\\send.avi";  //文件名
    FILE *fp = fopen(filename, "rb");  //以二進制方式
打開文件
    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);

    //循環發送數據，直到文件結尾
```

```
    char buffer[BUF_SIZE] = {0};  //緩衝區
    int nCount;
    while( (nCount = fread(buffer, 1, BUF_SIZE, fp))
> 0 ){
        send(clntSock, buffer, nCount, 0);
    }

    shutdown(clntSock, SD_SEND);  //文件讀取完畢，斷開
輸出流，向客戶端發送FIN包
    recv(clntSock, buffer, BUF_SIZE, 0);  //阻塞，等待
客戶端接收完畢

    fclose(fp);
    closesocket(clntSock);
    closesocket(servSock);
    WSACleanup();

    system("pause");
    return 0;
}
```

客戶端代碼：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

#define BUF_SIZE 1024

int main(){
    //先輸入文件名，看文件是否能創建成功
    char filename[100] = {0};  //文件名
    printf("Input filename to save: ");
```

```
    gets(filename);
    FILE *fp = fopen(filename, "wb");  //以二進制方式
打開（創建）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));

    //循環接收數據，直到文件傳輸完畢
    char buffer[BUF_SIZE] = {0};  //文件緩衝區
    int nCount;
    while( (nCount = recv(sock, buffer, BUF_SIZE, 0))
> 0 ){
        fwrite(buffer, nCount, 1, fp);
    }
    puts("File transfer success!");

    //文件接收完畢後直接關閉套接字，無需調用shutdown()
    fclose(fp);
    closesocket(sock);
```

```
    WSACleanup();
    system("pause");
    return 0;
}
```

在D盘中準備好send.avi文件，先運行 server，再運行 client：

Input filename to save: D:\\recv.avi↙

//稍等片刻後

File transfer success!

打開D盘就可以看到 recv.avi，大小和 send.avi 相同，可以正常播放。

注意 server.cpp 第42行代碼，recv() 並沒有接收到 client 端的數據，當 client 端調用 closesocket() 後，server 端會收到FIN包，recv() 就會返回，後面的代碼繼續執行。

socket網絡字節序以及大端序小端序

不同CPU中，4字節整數1在內存空間的存儲方式是不同的。4字節整數1可用2進製表示如下：

00000000 00000000 00000000 00000001

有些CPU以上面的順序存儲到內存，另外一些CPU則以倒序存儲，如下所示：

00000001 00000000 00000000 00000000

若不考慮這些就收發數據會發生問題，因為保存順序的不同意味著對接收數據的解析順序也不同。

大端序和小端序

CPU向內存保存數據的方式有兩種：

- 大端序（Big Endian）：高位字節存放到低位地址（高位字節在前）。
- 小端序（Little Endian）：高位字節存放到高位地址（低位字節在前）。

僅憑描述很難解釋清楚，不妨來看一個實例。假設在 0x20 號開始的地址中保存4字節 int 型數據 0x12345678，大端序CPU保存方式如下圖所示：

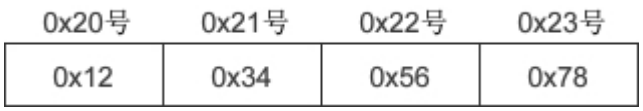


圖1：整數 0x12345678 的大端序字節表示

對於大端序，最高位字節 0x12 存放到低位地址，最低位字節 0x78 存放到高位地址。小端序的保存方式如下圖所示：

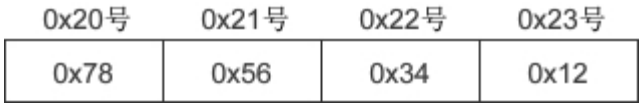


圖2：整數 0x12345678 的小端序字節表示

不同CPU保存和解析數據的方式不同（主流的Intel系列CPU為小端序），小端序系統和大端序系統通信時會發生數據解析錯誤。因此在發送數據前，要將數據轉換為統一的格式——網絡字節序（Network Byte Order）。網絡字節序統一為大端序。

主機A先把數據轉換成大端序再進行網絡傳輸，主機B收到數據後先轉換為自己的格式再解析。

網絡字節序轉換函數

在《使用bind()和connect()函數》一節中講解了 sockaddr_in 結構體，其中就用到了網絡字節序轉換函數，如下所示：

```
//創建sockaddr_in結構體變量
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每個字節都用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");
//具體的IP地址
serv_addr.sin_port = htons(1234);  //端口號
```

htons() 用來將當前主機字節序轉換為網絡字節序，其中`h`代表主機（host）字節序，`n`代表網絡（network）字節序，`s`代表short，htons

是 h、to、n、s 的組合，可以理解為”將short型數據從當前主機字節序轉換為網絡字節序“。

常見的網絡字節轉換函數有：

- htons()：host to network short，將short類型數據從主機字節序轉換為網絡字節序。
- ntohs()：network to host short，將short類型數據從網絡字節序轉換為主機字節序。
- htonl()：host to network long，將long類型數據從主機字節序轉換為網絡字節序。
- ntohl()：network to host long，將long類型數據從網絡字節序轉換為主機字節序。

通常，以`s`為後綴的函數中，`s`代表2個字節short，因此用於端口號轉換；以`l`為後綴的函數中，`l`代表4個字節的long，因此用於IP地址轉換。

舉例說明上述函數的調用過程：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

int main(){
    unsigned short host_port = 0x1234, net_port;
    unsigned long host_addr = 0x12345678, net_addr;
```

```
    net_port = htons(host_port);
    net_addr = htonl(host_addr);

    printf("Host ordered port: %#x\n", host_port);
    printf("Network ordered port: %#x\n", net_port);
    printf("Host ordered address: %#lx\n",
host_addr);
    printf("Network ordered address: %#lx\n",
net_addr);

    system("pause");
    return 0;
}
```

運行結果：

Host ordered port: 0x1234

Network ordered port: 0x3412

Host ordered address: 0x12345678

Network ordered address: 0x78563412

另外需要說明的是，sockaddr_in 中保存IP地址的成員為32位整數，而我們熟悉的是點分十進制表示法，例如 127.0.0.1，它是一個字符串，因此為了分配IP地址，需要將字符串轉換為4字節整數。

inet_addr() 函數可以完成這種轉換。inet_addr() 除了將字符串轉換為32位整數，同時還進行網絡字節序轉換。請看下面的代碼：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")
```

```
int main(){
    char *addr1 = "1.2.3.4";
    char *addr2 = "1.2.3.256";

    unsigned long conv_addr = inet_addr(addr1);
    if(conv_addr == INADDR_NONE){
        puts("Error occured!");
    }else{
        printf("Network ordered integer addr:
%#lx\n", conv_addr);
    }

    conv_addr = inet_addr(addr2);
    if(conv_addr == INADDR_NONE){
        puts("Error occured!");
    }else{
        printf("Network ordered integer addr:
%#lx\n", conv_addr);
    }

    system("pause");
    return 0;
}
```

運行結果：

Network ordered integer addr: 0x4030201

Error occured!

從運行結果可以看出，inet_addr() 不僅可以把IP地址轉換為32位整數，還可以檢測無效IP地址。

注意：為 sockaddr_in 成員賦值時需要顯式地將主機字節序轉換為網

絡字節序，而通過 write()/send() 發送數據時TCP協議會自動轉換為網絡字節序，不需要再調用相應的函數。

在socket中使用域名

客戶端中直接使用IP地址會有很大的弊端，一旦IP地址變化（IP地址會經常變動），客戶端軟件就會出現錯誤。

而使用域名會方便很多，註冊後的域名只要每年續費就永遠屬於自己的，更換IP地址時修改域名解析即可，不會影響軟件的正常使用。

關於域名註冊、域名解析、host 文件、DNS 服務器等本節並未詳細講解，請讀者自行腦補。本節重點講解如何使用域名。

通過域名獲取IP地址

域名僅僅是IP地址的一個助記符，目的是方便記憶，通過域名並不能找到目標計算機，通信之前必須要將域名轉換成IP地址。

gethostbyname() 函數可以完成這種轉換，它的原型為：

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

hostname 為主機名，也就是域名。使用該函數時，只要傳遞域名字符串，就會返回域名對應的IP地址。返回的地址信息會裝入 hostent 結構體，該結構體的定義如下：

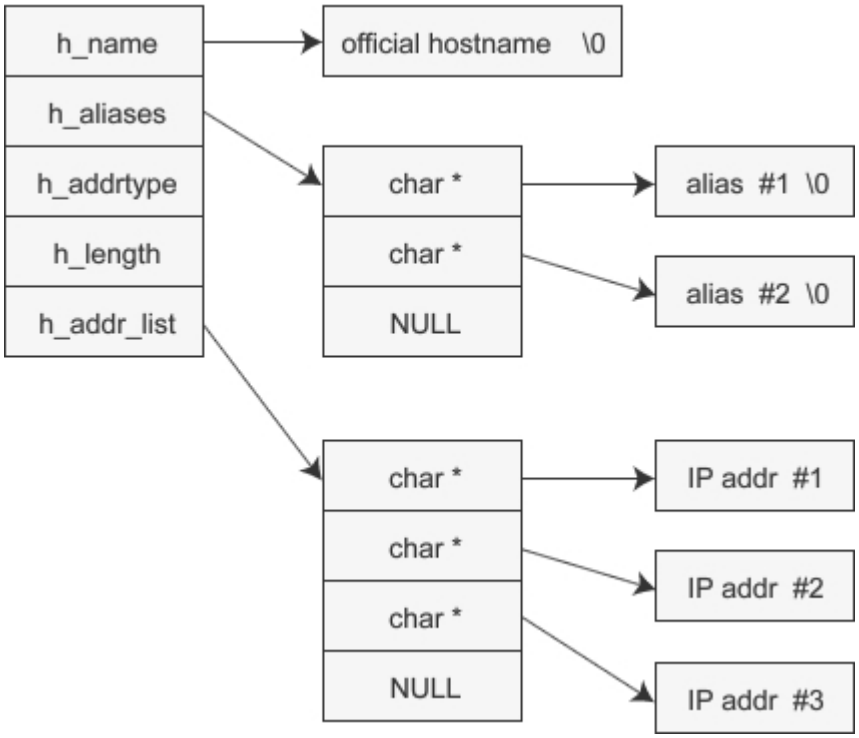
```
struct hostent{
    char *h_name;  //official name
    char **h_aliases;  //alias list
    int  h_addrtype;  //host address type
    int  h_length;  //address lenght
```

```
    char **h_addr_list;  //address list
}
```

從該結構體可以看出，不只返回IP地址，還會附帶其他信息，各位讀者只需關注最後一個成員 h_addr_list。下面是對各成員的說明：

- h_name：官方域名（Official domain name）。官方域名代表某一主頁，但實際上一些著名公司的域名並未用官方域名註冊。
- h_aliases：別名，可以通過多個域名訪問同一主機。同一IP地址可以綁定多個域名，因此除了當前域名還可以指定其他域名。
- h_addrtype：gethostbyname() 不僅支持 IPv4，還支持 IPv6，可以通過此成員獲取IP地址的地址族（地址類型）信息，IPv4 對應 AF_INET，IPv6 對應 AF_INET6。
- h_length：保存IP地址長度。IPv4 的長度為4個字節，IPv6 的長度為16個字節。
- h_addr_list：這是最重要的成員。通過該成員以整數形式保存域名對應的IP地址。對於用戶較多的服務器，可能會分配多個IP地址給同一域名，利用多個服務器進行均衡負載。

hostent 結構體變量的組成如下圖所示：



下面的代碼主要演示 gethostbyname() 的應用，並說明 hostent 結構體的特性：

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    struct hostent *host =
gethostbyname("www.baidu.com");
    if(!host){
        puts("Get IP address error!");
        system("pause");
        exit(0);
```

```
    }

    //别名
    for(int i=0; host->h_aliases[i]; i++){
        printf("Aliases %d: %s\n", i+1, host-
>h_aliases[i]);
    }

    //地址類型
    printf("Address type: %s\n", (host-
>h_addrtype==AF_INET) ? "AF_INET": "AF_INET6");

    //IP地址
    for(int i=0; host->h_addr_list[i]; i++){
        printf("IP addr %d: %s\n", i+1, inet_ntoa( *
(struct in_addr*)host->h_addr_list[i] ) );
    }

    system("pause");
    return 0;
}
```

運行結果：

Aliases 1: www.baidu.com

Address type: AF_INET

IP addr 1: 61.135.169.121

IP addr 2: 61.135.169.125

理解UDP套接字

TCP 是面向連接的傳輸協議，建立連接時要經過三次握手，斷開連接時要經過四次握手，中間傳輸數據時也要回復ACK包確認，多種機制保證了數據能夠正確到達，不會丟失或出錯。

UDP 是非連接的傳輸協議，沒有建立連接和斷開連接的過程，它只是簡單地把數據丟到網絡中，也不需要ACK包確認。

UDP 傳輸數據就好像我們郵寄包裹，郵寄前需要填好寄件人和收件人地址，之後送到快遞公司即可，但包裹是否正確送達、是否損壞我們無法得知，也無法保證。UDP 協議也是如此，它只管把數據包發送到網絡，然後就不管了，如果數據丟失或損壞，發送端是無法知道的，當然也不會重發。

既然如此，TCP應該是更加優質的傳輸協議吧？

如果只考慮可靠性，TCP的確比UDP好。但UDP在結構上比TCP更加簡潔，不會發送ACK的應答消息，也不會給數據包分配Seq序號，所以UDP的傳輸效率有時會比TCP高出很多，編程中實現UDP也比TCP簡單。

UDP 的可靠性雖然比不上TCP，但也不會像想象中那麼頻繁地發生數據損毀，在更加重視傳輸效率而非可靠性的情況下，UDP是一種很好的選擇。比如視頻通信或音頻通信，就非常適合採用UDP協議；通信時數據必須高效傳輸才不會產生“卡頓”現象，用戶體驗才更加流暢，

如果丟失幾個數據包，視頻畫面可能會出現“雪花”，音頻可能會夾帶一些雜音，這些都是無妨的。

與UDP相比，TCP的生命在於流控制，這保證了數據傳輸的正確性。

最後需要說明的是：TCP的速度無法超越UDP，但在收發某些類型的數據時有可能接近UDP。例如，每次交換的數據量越大，TCP 的傳輸速率就越接近於 UDP。

基於UDP的服務器端和客戶端

前面的文章中我們給出了幾個TCP的例子，對於UDP而言，只要能理解前面的內容，實現並非難事。

UDP中的服務器端和客戶端沒有連接

UDP不像TCP，無需在連接狀態下交換數據，因此基於UDP的服務器端和客戶端也無需經過連接過程。也就是說，不必調用 listen() 和 accept() 函數。UDP中只有創建套接字的過程和數據交換的過程。

UDP服務器端和客戶端均只需1個套接字

TCP中，套接字是一對一的關係。如要向10個客戶端提供服務，那麼除了負責監聽的套接字外，還需要創建10套接字。但在UDP中，不管是服務器端還是客戶端都只需要1個套接字。之前解釋UDP原理的時候舉了郵寄包裹的例子，負責郵寄包裹的快遞公司可以比喻為UDP套接字，只要有1個快遞公司，就可以通過它向任意地址郵寄包裹。同樣，只需1個UDP套接字就可以向任意主機傳送數據。

基於UDP的接收和發送函數

創建好TCP套接字後，傳輸數據時無需再添加地址信息，因為TCP套接字將保持與對方套接字的連接。換言之，TCP套接字知道目標地址信息。但UDP套接字不會保持連接狀態，每次傳輸數據都要添加目標地址信息，這相當於在郵寄包裹前填寫收件人地址。

發送數據使用 sendto() 函數：

```
ssize_t sendto(int sock, void *buf, size_t nbytes,
int flags, struct sockaddr *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() 函數類似，下面是詳細參數說明：

- sock：用於傳輸UDP數據的套接字；
- buf：保存待傳輸數據的緩衝區地址；
- nbytes：帶傳輸數據的長度（以字節計）；
- flags：可選項參數，若沒有可傳遞0；
- to：存有目標地址信息的 sockaddr 結構體變量的地址；
- addrlen：傳遞給參數 to 的地址值結構體變量的長度。

UDP 發送函數 sendto() 與TCP發送函數 write()/send() 的最大區別在於，sendto() 函數需要向他傳遞目標地址信息。

接收數據使用 recvfrom() 函數：

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

由於UDP數據的發送端不不定，所以 recvfrom() 函數定義為可接收發送端信息的形式，具體參數如下：

- sock：用於接收UDP數據的套接字；
- buf：保存接收數據的緩衝區地址；

- nbytes：可接收的最大字節數（不能超過buf緩衝區的大小）；
- flags：可選項參數，若沒有可傳遞0；
- from：存有發送端地址信息的sockaddr結構體變量的地址；
- addrlen：保存參數 from 的結構體變量長度的變量地址值。

基於UDP的回聲服務器端/客戶端

下面結合之前的內容實現回聲客戶端。需要注意的是，UDP不同於TCP，不存在請求連接和受理過程，因此在某種意義上無法明確區分服務器端和客戶端，只是因為其提供服務而稱為服務器端，希望各位讀者不要誤解。

下面給出Windows下的代碼，Linux與此類似，不再贅述。

服務器端 server.cpp：

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib")  //加載
ws2_32.dll

#define BUF_SIZE 100

int main(){
    WSADATA wsaData;
    WSAStartup( MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);

    //綁定套接字
    sockaddr_in servAddr;
```

```
    memset(&servAddr, 0, sizeof(servAddr));  //每個字
節都用0填充
    servAddr.sin_family = PF_INET;  //使用IPv4地址
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY); //
自動獲取IP地址
    servAddr.sin_port = htons(1234);  //端口
    bind(sock, (SOCKADDR*)&servAddr,
sizeof(SOCKADDR));

    //接收客戶端請求
    SOCKADDR clntAddr;  //客戶端地址信息
    int nSize = sizeof(SOCKADDR);
    char buffer[BUF_SIZE];  //緩衝區
    while(1){
        int strLen = recvfrom(sock, buffer, BUF_SIZE,
0, &clntAddr, &nSize);
        sendto(sock, buffer, strLen, 0, &clntAddr,
nSize);
    }

    closesocket(sock);
    WSACleanup();
    return 0;
}
```

代碼說明：

1) 第12行代碼在創建套接字時，向 socket() 第二個參數傳遞 SOCK_DGRAM，以指明使用UDP協議。

2) 第18行代碼中使用`htonl(INADDR_ANY)`來自動獲取IP地址。

利用常數 INADDR_ANY 自動獲取IP地址有一個明顯的好處，就是當軟件安裝到其他服務器或者服務器IP地址改變時，不用再更改源碼重

新編譯，也不用在啟動軟件時手動輸入。而且，如果一臺計算機中已分配多個IP地址（例如路由器），那麼只要端口號一致，就可以從不同的IP地址接收數據。所以，服務器中優先考慮使用INADDR_ANY；而客戶端中除非帶有一部分服務器功能，否則不會採用。

客戶端 client.cpp：

```
#include <stdio.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")  //加載 ws2_32.dll

#define BUF_SIZE 100

int main(){
    //初始化DLL
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    //創建套接字
    SOCKET sock = socket(PF_INET, SOCK_DGRAM, 0);

    //服務器地址信息
    sockaddr_in servAddr;
    memset(&servAddr, 0, sizeof(servAddr));  //每個字節都用0填充
    servAddr.sin_family = PF_INET;
    servAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servAddr.sin_port = htons(1234);

    //不斷獲取用戶輸入併發送給服務器，然後接受服務器數據
    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, &addrLen);
        buffer[strLen] = 0;
        printf("Message form server: %s\n", buffer);
    }

    closesocket(sock);
    WSACleanup();
    return 0;
}
```

先運行 server，再運行 client，client 輸出結果為：

Input a string: C語言中文網
Message form server: C語言中文網
Input a string: c.biancheng.net Founded in 2012
Message form server: c.biancheng.net Founded in 2012
Input a string:

從代碼中可以看出，server.cpp 中沒有使用 listen() 函數，client.cpp 中也沒有使用 connect() 函數，因為 UDP 不需要連接。