玩转 C、C++ socket 编程

—— socket 通信技术成神之路

前言

socket 是“套接字”的意思，是计算机之间进行通信的一种约定，也可以认为是一种技术。学习 socket，就是学习计算机之间如何通信，并能够用编程语言开发出实用的程序。

本教程并不要求读者有 Linux 和 Windows 开发的经验，也不需要深入了解 TCP/IP 协议，涉及到相关知识时我们都进行了说明。

同时学习 Linux 和 Windows 的原因

大多数项目是在 Linux 下开发服务器端，而在 Windows 下开发客户端，需要经常在两大平台之间进行切换，单独学习一种平台没有实践意义。

值得欣慰的是，两大平台下的 socket 编程非常相似，并不会增加多少学习成本。

网络编程就是编写程序使两台联网的计算机相互交换数据。这就是 socket 的全部内容了吗？是的！socket 编程远比想象中的简单很多。

目录

前言 1
第一节 ： Socket 简介 3
第二节 一个简单的 Linux 下的 socket 程序 4
第三节 一个简单的 Windows 下的 socket 程序 8
第四节 动态链接库 DLL 的加载：隐式加载(载入时加载)和显式加载(运行时加载) 11
第五节 WSAStartup()函数以及 DLL 的加载 15
第六节 使用 socket()函数创建套接字 17
第七节 使用 bind()和 connect()函数 19
第八节 使用 listen()和 accept()函数 22
第九节 socket 数据的接收和发送 23
第十节 实战：回声客户端/服务端模型 24
第十一节 实战：迭代客户端/服务端模型 27
第十二章 socket 缓冲区以及阻塞模式 29
第十三节 TCP 的粘包问题以及数据的无边界性 31
第十四节 TCP 数据报结构以及三次握手（图解） 33
第十五节 TCP 数据的传输过程 36
第十六节 TCP 四次握手断开连接（图解） 38
第十七节 优雅的断开连接--shutdown() 40
第十八节 socket 文件传输功能的实现 41

第一节 ：Socket 简介

在计算机通信领域，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。

socket 的典型应用就是 Web 服务器和浏览器：浏览器获取用户输入的 URL，向服务器发起请求，服务器分析接收到的 URL，将对应的网页内容返回给浏览器，浏览器再经过解析和渲染，就将文字、图片、视频等元素呈现给用户。

学习 socket，也就是学习计算机之间如何通信，并编写出实用的程序。

IP 地址（IP Address）

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。

其实，我们的计算机并不知道 IP 地址对应的地理位置，当要通信时，只是将 IP 地址封装到要发送的数据包中，交给路由器去处理。路由器有非常智能和高效的算法，很快就会找到目标计算机，并将数据包传递给它，完成一次单向通信。

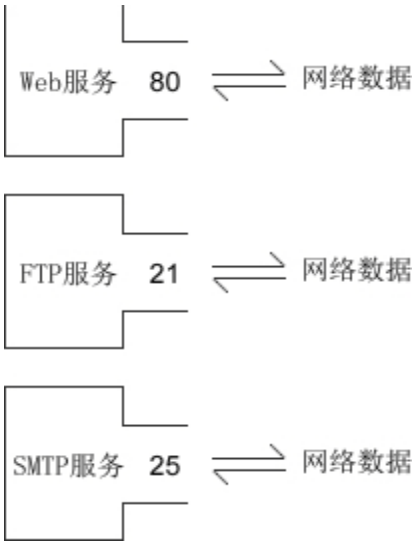
目前大部分软件使用 IPv4 地址，但 IPv6 也正在被人们接受，尤其是在教育网中，已经大量使用。

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：



协议（Protocol）

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。

协议仅仅是一种规范，必须由计算机软件来实现。例如 IP 协议规定了如何找到目标计算机，那么各个开发商在开发自己的软件时就必须遵守该协议，不能另起炉灶。

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。

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。

注意：SOCK_DGRAM 没有想象中的糟糕，不会频繁的丢失数据，数据错误只是小概率事件。

有可能多种协议使用同一种数据传输方式，所以在 socket 编程中，需要同时指明数据传输方式和协议。

综上所述：IP 地址和端口能够在广袤的互联网中定位到要通信的程序，协议和数据传输方式规定了如何传输数据，有了这些，两台计算机就可以通信了。

第二节 一个简单的 Linux 下的 socket 程序

和 C 语言教程一样，我们从一个简单的“Hello World!”程序切入 socket 编程。

本节演示了 Linux 下的代码，server.cpp 是服务器端代码，client.cpp 是客户端代码，要实现的功能是：客户端从服务器读取一个字符串并打印出来。

服务器端代码 server.cpp:

```
1.  #include <stdio.h>
2.  #include <string.h>
```

```
3.  #include <stdlib.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  #include <netinet/in.h>
8.
9.  int main(){
10.     //创建套接字
11.     int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
12.
13.     //将套接字和 IP、端口绑定
14.     struct sockaddr_in serv_addr;
15.     memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
16.     serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
17.     serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
18.     serv_addr.sin_port = htons(1234);  //端口
19.     bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
20.
21.     //进入监听状态，等待用户发起请求
22.     listen(serv_sock, 20);
23.
24.     //接收客户端请求
25.     struct sockaddr_in clnt_addr;
26.     socklen_t clnt_addr_size = sizeof(clnt_addr);
27.     int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr,
&clnt_addr_size);
28.
29.     //向客户端发送数据
30.     char str[] = "Hello World!";
31.     write(clnt_sock, str, sizeof(str));
32.
33.     //关闭套接字
34.     close(clnt_sock);
```

```
35.         close(serv_sock);
36.
37.         return 0;
38.     }
```

客户端代码 client.cpp:

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <stdlib.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.
8.  int main(){
9.      //创建套接字
10.     int sock = socket(AF_INET, SOCK_STREAM, 0);
11.
12.     //向服务器（特定的 IP 和端口）发起请求
13.     struct sockaddr_in serv_addr;
14.     memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用 0 填充
15.     serv_addr.sin_family = AF_INET;  //使用 IPv4 地址
16.     serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的 IP 地址
17.     serv_addr.sin_port = htons(1234);  //端口
18.     connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
19.
20.     //读取服务器传回的数据
21.     char buffer[40];
22.     read(sock, buffer, sizeof(buffer)-1);
23.
24.     printf("Message form server: %s\n", buffer);
25.
26.     //关闭套接字
```

```
27.         close(sock);
28.
29.         return 0;
30.  }
```

先编译 server.cpp 并运行:

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

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

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:

1) 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。

2) 上面的源文件后缀为.cpp，是 C++代码，所以要用 g++命令来编译。

C++和 C 语言的一个重要区别是：在 C 语言中，变量必须在函数的开头定义；而在 C++中，变量可以在函数的任何地方定义，使用更加灵活。这里之所以使用 C++代码，是不希望在函数开头堆砌过多变量。

源码解析

1) 先说一下 server.cpp 中的代码。

第 11 行通过 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。

第 19 行通过 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。

第 22 行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”。

第 27 行的 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。

第 31 行的 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。

第 19 行代码通过 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。

第 23 行代码通过 read() 从套接字文件中读取数据。和 C 语言教程一样，我们从一个简单的“Hello World!”程序切入 socket 编程。

第三节　一个简单的 Windows 下的 socket 程序

上节演示了 Linux 下的 socket 程序，这节来看一下 Windows 下的 socket 程序。同样，server.cpp 为服务器端代码，client 为客户端代码。

服务器端代码 server.cpp:

```
1.  #include <stdio.h>
2.  #include <winsock2.h>
3.  #pragma comment (lib, "ws2_32.lib")  //加载 ws2_32.dll
4.
5.  int main(){
6.      //初始化 DLL
7.      WSADATA wsaData;
8.      WSAStartup( MAKEWORD(2, 2), &wsaData);
9.
10.     //创建套接字
11.     SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
12.
13.     //绑定套接字
14.     sockaddr_in sockAddr;
15.     memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
```

```
16.         sockAddr.sin_family = PF_INET;    //使用 IPv4 地址
17.         sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");    //具体的 IP 地址
18.         sockAddr.sin_port = htons(1234);    //端口
19.         bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
20.
21.         //进入监听状态
22.         listen(servSock, 20);
23.
24.         //接收客户端请求
25.         SOCKADDR clntAddr;
26.         int nSize = sizeof(SOCKADDR);
27.         SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);
28.
29.         //向客户端发送数据
30.         char *str = "Hello World!";
31.         send(clntSock, str, strlen(str)+sizeof(char), NULL);
32.
33.         //关闭套接字
34.         closesocket(clntSock);
35.         closesocket(servSock);
36.
37.         //终止 DLL 的使用
38.         WSACleanup();
39.
40.         return 0;
41.  }
```

客户端代码 client.cpp:

```
i.      #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();

```

```
35.         system("pause");
36.         return 0;
37.  }
```

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。DLL 有两种加载方式，请参看下一节：动态链接库 DLL 的加载

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。

第四节 动态链接库 DLL 的加载：隐式加载(载入时加载)和显式加载(运行时加载)

静态链接库在链接时，编译器会将 .obj 文件和 .LIB 文件组织成一个 .exe 文件，程序运行时，将全部数据加载到内存。

如果程序体积较大，功能较为复杂，那么加载到内存中的时间就会比较长，最直接的一个例子就是双击打开一个软件，要很久才能看到界面。这是静态链接库的一个弊端。

动态链接库有两种加载方式：隐式加载和显示加载。

隐式加载又叫载入时加载，指在主程序载入内存时搜索 DLL，并将 DLL 载入内存。隐式加载也会有静态链接库的问题，如果程序稍大，加载时间就会过长，用户不能接受。

显式加载又叫运行时加载，指主程序在运行过程中需要 DLL 中的函数时再加载。显式加载是将较大的程序分开加载的，程序运行时只需要将主程序载入内存，软件打开速度快，用户体验好。

隐式加载

首先创建一个工程，命名为 cDemo，添加源文件 main.c，内容如下：

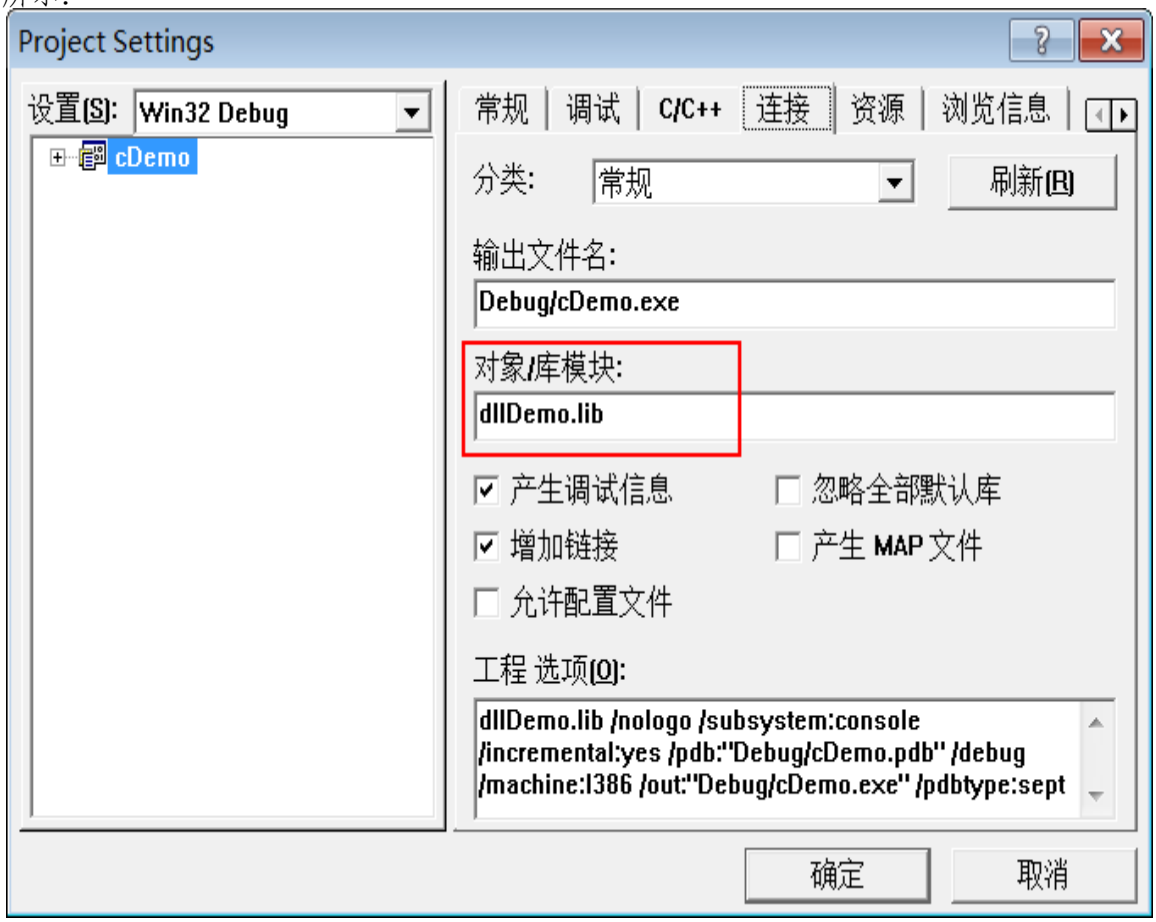
```
1.  #include<stdio.h>
2.
3.  extern int add(int, int);   // 也可以是 _declspec(dllimport) int add(int, int);
4.  extern int sub(int, int);   // 也可以是 _declspec(dllimport) int sub(int, int);
5.
6.  int main(){
```

```
7.          int a=10, b=5;
8.          printf("a+b=%d\n", add(a, b));
9.          printf("a-b=%d\n", sub(a, b));
10.         return 0;
11.  }
```

找到上节创建的 dllDemo 工程，将 debug 目录下的 dllDemo.lib 和 dllDemo.dll 复制到当前工程目录下。

前面已经说过：.lib 文件包含 DLL 导出的函数和变量的符号名，只是用来为链接程序提供必要的信息，以便在链接时找到函数或变量的入口地址；.dll 文件才包含实际的函数和数据。所以首先需要将 dllDemo.lib 引入到当前项目。

选择”工程(Project) -> 设置(Settings)“菜单，打开工程设置对话框，选择”链接(link)“选项卡，在”对象/库模块(Object/library modules)“编辑框中输入 dllDemo.lib，如下图所示：



但是这样引入 .lib 文件有一个缺点，就是将源码提供给其他用户编译时，也必须手动引入 .lib 文件，麻烦而且容易出错，所以最好是在源码中引入 .lib 文件，如下所示：

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

更改上面的代码:

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

_declspec(dllimport) int add(int, int);
_declspec(dllimport) int sub(int, int);

int main(){
    int a=10, b=5;
    printf("a+b=%d\n", add(a, b));
    printf("a-b=%d\n", sub(a, b));
    return 0;
}
```

点击确定回到项目，编译、链接并运行，输出结果如下:

Congratulations! DLL is loaded!

a+b=15

a-b=5

在 main.c 中除了用 extern 关键字声明 add() 和 sub() 函数来自外部文件，还可以用 _declspec(dllimport) 标识符声明函数来自动态链接库。

为了更好的进行模块化设计，最好将 add() 和 sub() 函数的声明放在头文件中，整理后的代码如下:

dllDemo.h

```
#ifndef _DLLDEMO_H
#define _DLLDEMO_H

#pragma comment(lib, "dllDemo.lib")
_declspec(dllexport) int add(int, int);
_declspec(dllexport) int sub(int, int);

#endif
```

main.c

```
#include<stdio.h>
#include "dllDemo.h"

int main(){
```

```
    int a=10, b=5;
    printf("a+b=%d\n", add(a, b));
    printf("a-b=%d\n", sub(a, b));
    return 0;
}
```

显式加载

显式加载动态链接库时，需要用到 LoadLibrary() 函数，该函数的作用是将指定的可执行模块映射到调用进程的地址空间。LoadLibrary() 函数的原型声明如下所示:

HMODULE LoadLibrary(LPCTSTR 1pFileName);

LoadLibrary() 函数不仅能够加载 DLL(.dll)，还可以加载可执行模块(.exe)。一般来说，当加载可执行模块时，主要是为了访问该模块内的一些资源，例如位图资源或图标资源等。LoadLibrary() 函数有一个字符串类型(LPCTSTR)的参数，该参数指定了可执行模块的名称，既可以是一个.dll 文件，也可以是一个.exe 文件。如果调用成功， LoadLibrary() 函数将返回所加载的那个模块的句柄。该函数的返回类型是 HMODULE。 HMODULE 类型和 HINSTANCE 类型可以通用。

当获取到动态链接库模块的句柄后，接下来就要想办法获取该动态链接库中导出函数的地址，这可以通过调用 GetProcAddress() 函数来实现。该函数用来获取 DLL 导出函数的 地址，其原型声明如下所示:

FARPROC GetProcAddress(HMODULE hModule, LPCSTR 1pProcName);

可以看到，GetProcAddress 函数有两个参数，其含义分别如下所述:

hModule: 指定动态链接库模块的句柄，即 LoadLibrary() 函数的返回值。

1pProcName: 字符串指针，表示 DLL 中函数的名字。

首先创建一个工程，命名为 cDemo，添加源文件 main.c，内容如下:

```
#include<stdio.h>
#include<stdlib.h>
#include<windows.h>  // 必须包含 windows.h

typedef int (*FUNADDR)();  // 指向函数的指针

int main(){
    int a=10, b=5;

    HINSTANCE dllDemo = LoadLibrary("dllDemo.dll");
    FUNADDR add, sub;
    if(dllDemo){
        add = (FUNADDR)GetProcAddress(dllDemo, "add");
        sub = (FUNADDR)GetProcAddress(dllDemo, "sub");
    }else{
```

```
        printf("Fail to load DLL!\n");
        system("pause");
        exit(1);
    }

    printf("a+b=%d\n", add(a, b));
    printf("a-b=%d\n", sub(a, b));

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

找到上节创建的 dllDemo 工程，将 debug 目录下的 dllDemo.dll 复制到当前工程目录下。注意，只需要 dllDemo.dll，不需要 dllDemo.lib。

运行程序，输出结果与上面相同。

HMODULE 类型、HINSTANCE 类型在 windows.h 中定义；LoadLibrary() 函数、GetProcAddress() 函数是 Win32 API，也在 windows.h 中定义。

通过以上的例子，我们可以看到，隐式加载和显式加载这两种加载 DLL 的方式各有 优点，如果采用动态加载方式，那么可以在需要时才加载 DLL，而隐式链接方式实现起来比较简单，在编写程序代码时就可以把链接工作做好，在程序中可以随时调用 DLL 导出的函数。但是，如果程序需要访问十多个 DLL，如果都采用隐式链接方式加载它们的话， 那么在该程序启动时，这些 DLL 都需要被加载到内存中，并映射到调用进程的地址空间， 这样将加大程序的启动时间。而且，一般来说，在程序运行过程中只是在某个条件满足时才需要访问某个 DLL 中的某个函数，其他情况下都不需要访问这些 DLL 中的函数。但是这时所有的 DLL 都已经被加载到内存中，资源浪费是比较严重的。在这种情况下，就可以采用显式加载的方式访问 DLL，在需要时才加载所需的 DLL，也就是说，在需要时 DLL 才会被加载到内存中，并被映射到调用进程的地址空间中。有一点需要说明的是，实际上， 采用隐式链接方式访问 DLL 时，在程序启动时也是通过调用 LoadLibrary() 函数加载该进程需要的动态链接库的。

第五节 WSAStartup()函数以及 DLL 的加载

本节讲解 Windows 下 DLL 的加载，学习 Linux Socket 的读者可以跳过。

WinSock（Windows Socket）编程依赖于系统提供的动态链接库(DLL)，有两个版本：

较早的 DLL 是 wsock32.dll，大小为 28KB，对应的头文件为 winsock1.h;

最新的 DLL 是 ws2_32.dll，大小为 69KB，对应的头文件为 winsock2.h。

几乎所有的 Windows 操作系统都已经支持 ws2_32.dll，包括个人操作系统 Windows 95 OSR2、Windows 98、Windows Me、Windows 2000、XP、Vista、Win7、Win8、Win10 以及服务器操作系统 Windows NT 4.0 SP4、Windows Server 2003、Windows Server 2008 等，所以你可以毫不犹豫地使用最新的 ws2_32.dll。

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。

关于 WinSock 规范

WinSock 规范的最新版本号为 2.2，较早的有 2.1、2.0、1.1、1.0，ws2_32.dll 支持所有的规范，而 wsock32.dll 仅支持 1.0 和 1.1。

wsock32.dll 已经能够很好的支持 TCP/IP 通信程序的开发，ws2_32.dll 主要增加了对其他协议的支持，不过建议使用最新的 2.2 版本。

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(1, 2); //主版本号为 1，副版本号为 2，返回 0x0201

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

关于 WSAData 结构体

WSAStartup() 函数执行成功后，会将与 ws2_32.dll 有关的信息写入 WSAData 结构体变量。WSAData 的定义如下：

```
typedef struct WSAData {
    WORD            wVersion;    //ws2_32.dll 建议我们使用的版本号
    WORD            wHighVersion;    //ws2_32.dll 支持的最高版本号
    //一个以 null 结尾的字符串，用来说明 ws2_32.dll 的实现以及厂商信息
    char            szDescription[WSADESCRIPTION_LEN+1];
    //一个以 null 结尾的字符串，用来说明 ws2_32.dll 的状态以及配置信息
    char            szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short iMaxSockets;    //2.0 以后不再使用
    unsigned short iMaxUdpDg;    //2.0 以后不再使用
    char FAR        *lpVendorInfo;    //2.0 以后不再使用
} WSADATA, *LPWSADATA;
```

最后 3 个成员已弃之不用，szDescription 和 szSystemStatus 包含的信息基本没有实用价值，读者只需关注前两个成员即可。请看下面的代码：

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

```
4.
5.  int main(){
6.      WSADATA wsaData;
7.      WSAStartup( MAKEWORD(2, 2), &wsaData);
8.
9.      printf("wVersion: %d.%d\n", LOBYTE(wsaData.wVersion),
HIBYTE(wsaData.wVersion));
10.     printf("wHighVersion: %d.%d\n", LOBYTE(wsaData.wHighVersion),
HIBYTE(wsaData.wHighVersion));
11.     printf("szDescription: %s\n", wsaData.szDescription);
12.     printf("szSystemStatus: %s\n", wsaData.szSystemStatus);
13.
14.     return 0;
15. }
```

运行结果:

wVersion: 2.2

wHighVersion: 2.2

szDescription: WinSock 2.0

szSystemStatus: Running

ws2_32.dll 支持的最高版本为 2.2，建议使用的版本也是 2.2。

综上所述：WinSock 编程的第一步就是加载 ws2_32.dll，然后调用 WSAStartup() 函数进行初始化，并指明要使用的版本号。

第六节 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”。

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，后面的教程会经常用到。

你也可以使用 PF 前缀，PF 是“Protocol Family”的简写，它和 AF 是一样的。例如，PF_INET 等价于 AF_INET，PF_INET6 等价于 AF_INET6。

2) type 为数据传输方式，常用的有 SOCK_STREAM 和 SOCK_DGRAM，在《socket 是什么意思》一节中已经进行了介绍。

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。

有了地址类型和数据传输方式，还不足以决定采用哪种协议吗？为什么还需要第三个参数呢？

正如大家所想，一般情况下有了 af 和 type 两个参数就可以创建套接字了，操作系统会自动推演出协议类型，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。

该教程使用 IPv4 地址，参数 af 的值为 PF_INET。如果使用 SOCK_STREAM 传输数据，那么满足这两个条件的协议只有 TCP，因此可以这样来调用 socket() 函数：

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

这种套接字称为 TCP 套接字。

如果使用 SOCK_DGRAM 传输方式，那么满足这两个条件的协议只有 UDP，因此可以这样来调用 socket() 函数：

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

这种套接字称为 UDP 套接字。

上面两种情况都只有一种协议满足条件，可以将 protocol 的值设为 0，系统会自动推演出应该使用什么协议，如下所示：

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

后面的教程中多采用这种简化写法。

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

第七节 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。

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

下面以 Linux 为例进行讲解，Windows 与此类似。

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，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));
```

这里我们使用 sockaddr_in 结构体，然后再强制转换为 sockaddr 类型，后边会讲解为什么这样做。

sockaddr_in 结构体

接下来不妨先看一下 sockaddr_in 结构体，它的成员变量如下：

```
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，后面会讲解为什么。

3) sin_addr 是 struct in_addr 结构体类型的变量，下面会详细讲解。

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：

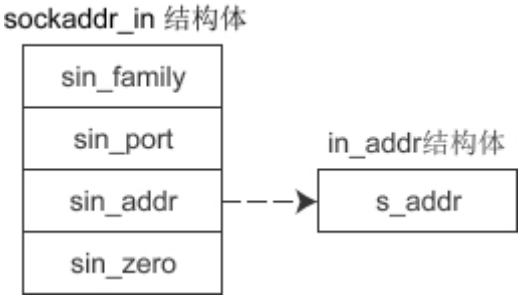
```
struct in_addr{
    in_addr_t  s_addr;  //32 位的 IP 地址
};
```

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



图解 sockaddr_in 结构体

为什么要搞这么复杂，结构体中嵌套结构体，而不用 sockaddr_in 的一个成员变量来指明 IP 地址呢？socket() 函数的第一个参数已经指明了地址类型，为什么在 sockaddr_in 结构体中还要再说明一次呢，这不是 嗦吗？

这些繁琐的细节确实给初学者带来了一定的障碍，我想，这或许是历史原因吧，后面的接口总要兼容前面的代码。各位读者一定要有耐心，暂时不理解没有关系，根据教程中的代码“照猫画虎”即可，时间久了自然会接受。

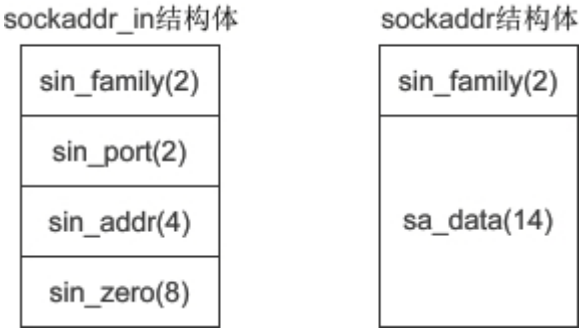
为什么使用 sockaddr_in 而不使用 sockaddr

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。

可以认为，sockaddr 是一种通用的结构体，可以用来保存多种类型的 IP 地址和端口号，而 sockaddr_in 是专门用来保存 IPv4 地址的结构体。另外还有 sockaddr_in6，用来保存 IPv6 地址，它的定义如下：

```
struct sockaddr_in6 {
    sa_family_t sin6_family;   //(2)地址类型，取值为 AF_INET6
    in_port_t sin6_port;   //(2)16 位端口号
    uint32_t sin6_flowinfo;   //(4)IPv6 流信息
    struct in6_addr sin6_addr;   //(4)具体的 IPv6 地址
    uint32_t sin6_scope_id;   //(4)接口范围 ID
};
```

正是由于通用结构体 sockaddr 使用不便，才针对不同的地址类型定义了不同的结构体。

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
```

各个参数的说明和 bind() 相同，不再赘述。

第八节 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。

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。

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。

第九节 socket 数据的接收和发送

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。

前面我们说过，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。

size_t 是通过 typedef 声明的 unsigned int 类型；ssize_t 在 "size_t" 前面加了一个 "s"，代表 signed，即 ssize_t 是通过 typedef 声明的 signed int 类型。

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。

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，初学者不必深究。

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);
```

第十节 实战：回声客户端/服务端模型

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。

考虑到大部分初学者使用 Windows 操作系统，本节将实现 Windows 下的回声程序，Linux 下稍作修改即可，不再给出代码。

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 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;
}
```

客户端 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: hello world ！

Message form server: hello

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“hello”；如果不希望把空格作为字符串的结束符，可以使用 gets() 函数。

通过本程序可以发现，客户端也可以向服务器端发送数据，这样服务器端就可以根据不同的请求作出不同的响应，http 服务器就是典型的例子，请求的网址不同，返回的页面也不同。

第十一节 实战：迭代客户端/服务端模型

前面的程序，不管服务器端还是客户端，都有一个问题，就是处理完一个请求立即退出了，没有太大的实际意义。能不能像 Web 服务器那样一直接受客户端的请求呢？能，使用 while 循环即可。

修改前面的回声程序，使服务器端可以不断响应客户端的请求。

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 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();

```

```
44.     return 0;
45. }
```

客户端 client.cpp：

```
1.  #include <stdio.h>
2.  #include <WinSock2.h>
3.  #include <windows.h>
4.  #pragma comment(lib, "ws2_32.lib")  //加载 ws2_32.dll
5.
6.  #define BUF_SIZE 100
7.
8.  int main(){
9.      //初始化 DLL
10.     WSADATA wsaData;
11.     WSAStartup(MAKEWORD(2, 2), &wsaData);
12.
13.     //向服务器发起请求
14.     sockaddr_in sockAddr;
15.     memset(&sockAddr, 0, sizeof(sockAddr));  //每个字节都用 0 填充
16.     sockAddr.sin_family = PF_INET;
17.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
18.     sockAddr.sin_port = htons(1234);
19.
20.     char bufSend[BUF_SIZE] = {0};
21.     char bufRecv[BUF_SIZE] = {0};
22.
23.     while(1){
24.         //创建套接字
25.         SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
26.         connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
27.         //获取用户输入的字符串并发送给服务器
28.         printf("Input a string: ");
29.         gets(bufSend);
30.         send(sock, bufSend, strlen(bufSend), 0);
31.         //接收服务器传回的数据
32.         recv(sock, bufRecv, BUF_SIZE, 0);
33.         //输出接收到的数据
34.         printf("Message form server: %s\n", bufRecv);
35.
36.         memset(bufSend, 0, BUF_SIZE);  //重置缓冲区
37.         memset(bufRecv, 0, BUF_SIZE);  //重置缓冲区
38.         closesocket(sock);  //关闭套接字
39.     }
40.
41.     WSACleanup();  //终止使用 DLL
42.     return 0;
43. }
```

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。后续我们会进行详细讲解。

第十二章 socket 缓冲区以及阻塞模式

在《socket 数据的接收和发送》一节中讲到，可以使用 write()/send() 函数发送数据，使用 read()/recv() 函数接收数据，本节就来看看数据是如何传递的。

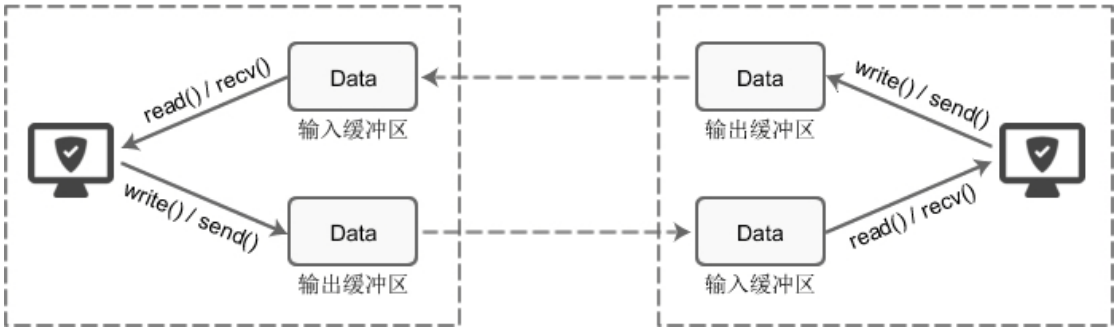
socket 缓冲区

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。



图：TCP 套接字的 I/O 缓冲区示意图

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:

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

运行结果:

Buffer length: 8192

这里仅给出示例，后面会详细讲解。

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。

TCP 套接字默认情况下是阻塞模式，也是最常用的。当然你也可以更改为非阻塞模式，后续我们会讲解。

第十三节 TCP 的粘包问题以及数据的无边界性

上节我们讲到了 socket 缓冲区和数据的传递过程，可以看到数据的接收和发送是无关的，read()/recv() 函数不管数据发送了多少次，都会尽可能多的接收数据。也就是说，read()/recv() 和 write()/send() 的执行次数可能不同。

例如，write()/send() 重复执行三次，每次都发送字符串"abc"，那么目标机器上的 read()/recv() 可能分三次接收，每次都接收"abc"；也可能分两次接收，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"。

假设我们希望客户端每次发送一位学生的学号，让服务器端返回该学生的姓名、住址、成绩等信息，这时候可能就会出现问题，服务器端不能区分学生的学号。例如第一次发送 1，第二次发送 3，服务器可能当成 13 来处理，返回的信息显然是错误的。

这就是数据的“粘包”问题，客户端发送的多个数据包被当做一个数据包接收。也称数据的无边界性，read()/recv() 函数不知道数据包的开始或结束标志（实际上也没有任何开始或结束标志），只把它们当做连续的数据流来处理。

下面的代码演示了粘包问题，客户端连续三次向服务器端发送数据，服务器端却一次性接收到所有数据。

服务器端代码 server.cpp:

```
#include <stdio.h>
#include <windows.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
    SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);

    Sleep(10000);   //注意这里，让程序暂停 10 秒

    //接收客户端发来的数据，并原样返回
    int recvLen = recv(clntSock, buffer, BUF_SIZE, 0);
    send(clntSock, buffer, recvLen, 0);

    //关闭套接字并终止 DLL 的使用
    closesocket(clntSock);
    closesocket(servSock);
    WSACleanup();

    return 0;
}
```

客户端代码 client.cpp:

```
#include <stdio.h>
#include <stdlib.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);

    //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
    char bufSend[BUF_SIZE] = {0};
    printf("Input a string: ");
    gets(bufSend);
```

```
29.     for(int i=0; i<3; i++){
30.         send(sock, bufSend, strlen(bufSend), 0);
31.     }
32.     //接收服务器传回的数据
33.     char bufRecv[BUF_SIZE] = {0};
34.     recv(sock, bufRecv, BUF_SIZE, 0);
35.     //输出接收到的数据
36.     printf("Message form server: %s\n", bufRecv);
37.
38.     closesocket(sock);   //关闭套接字
39.     WSACleanup();   //终止使用 DLL
40.
41.     system("pause");
42.     return 0;
43. }
```

先运行 server，再运行 client，并在 10 秒内输入字符串"abc"，再等数秒，服务器就会返回数据。运行结果如下：

Input a string: abc

Message form server: abcabcabc

本程序的关键是 server.cpp 第 31 行的代码 Sleep(10000);，它让程序暂停执行 10 秒。在这段时间内，client 连续三次发送字符串"abc"，由于 server 被阻塞，数据只能堆积在缓冲区中，10 秒后，server 开始运行，从缓冲区中一次性读出所有积压的数据，并返回给客户端。

另外还需要说明的是 client.cpp 第 34 行代码。client 执行到 recv() 函数，由于输入缓冲区中没有数据，所以会被阻塞，直到 10 秒后 server 传回数据才开始执行。用户看到的直观效果就是，client 暂停一段时间才输出 server 返回的结果。

client 的 send() 发送了三个数据包，而 server 的 recv() 却只接收到一个数据包，这很好的说明了数据的粘包问题。

第十四节 TCP 数据报结构以及三次握手（图解）

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的通信协议，数据在传输前要建立连接，传输完毕后还要断开连接。

客户端在收发数据前要使用 connect() 函数和服务器建立连接。建立连接的目的是保证 IP 地址、端口、物理链路等正确无误，为数据的传输开辟通道。

TCP 建立连接时要传输三个数据包，俗称三次握手（Three-way Handshaking）。可以形象的比喻为下面的对话：

- [Shake 1] 套接字 A：“你好，套接字 B，我这里有数据要传送给你，建立连接吧。”
- [Shake 2] 套接字 B：“好的，我这边已准备就绪。”
- [Shake 3] 套接字 A：“谢谢你受理我的请求。”

TCP 数据报结构

我们先来看一下 TCP 数据报的结构：

带阴影的几个字段需要重点说明一下：

1) 序号：Seq（Sequence Number）序号占 32 位，用来标识从计算机 A 发送到计算机 B 的数据包的序号，计算机发送数据时对此进行标记。

2) 确认号：Ack（Acknowledge Number）确认号占 32 位，客户端和服务器端都可以发送，Ack = Seq + 1。

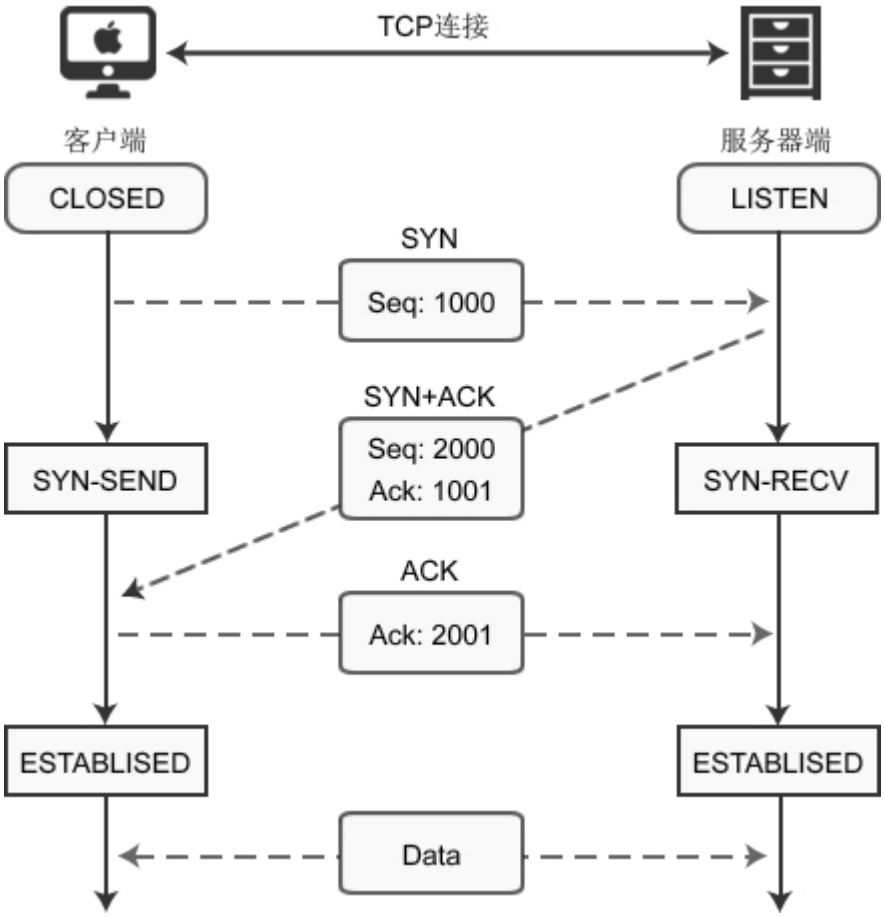
3) 标志位：每个标志位占用 1Bit，共有 6 个，分别为 URG、ACK、PSH、RST、SYN、FIN，具体含义如下：

- URG：紧急指针（urgent pointer）有效。
- ACK：确认序号有效。
- PSH：接收方应该尽快将这个报文交给应用层。
- RST：重置连接。
- SYN：建立一个新连接。
- FIN：断开一个连接。

对英文字母缩写的总结：Seq 是 Sequence 的缩写，表示序列；Ack(ACK) 是 Acknowledge 的缩写，表示确认；SYN 是 Synchronous 的缩写，愿意是“同步的”，这里表示建立同步连接；FIN 是 Finish 的缩写，表示完成。

连接的建立（三次握手）

使用 connect() 建立连接时，客户端和服务器端会相互发送三个数据包，请看下图：



客户端调用 socket() 函数创建套接字后，因为没有建立连接，所以套接字处于 CLOSED 状态；服务器端调用 listen() 函数后，套接字进入 LISTEN 状态，开始监听客户端请求。

这个时候，客户端开始发起请求：

1) 当客户端调用 connect() 函数后，TCP 协议会组建一个数据包，并设置 SYN 标志位，表示该数据包是用来建立同步连接的。同时生成一个随机数字 1000，填充“序号（Seq）”字段，表示该数据包的序号。完成这些工作，开始向服务器端发送数据包，客户端就进入了 SYN-SEND 状态。

2) 服务器端收到数据包，检测到已经设置了 SYN 标志位，就知道这是客户端发来的建立连接的“请求包”。服务器端也会组建一个数据包，并设置 SYN 和 ACK 标志位，SYN 表示该数据包用来建立连接，ACK 用来确认收到了刚才客户端发送的数据包。

服务器生成一个随机数 2000，填充“序号（Seq）”字段。2000 和客户端数据包没有关系。

服务器将客户端数据包序号（1000）加 1，得到 1001，并用这个数字填充“确认号（Ack）”字段。

服务器将数据包发出，进入 SYN-RECV 状态。

3) 客户端收到数据包，检测到已经设置了 SYN 和 ACK 标志位，就知道这是服务器发来的“确认包”。客户端会检测“确认号（Ack）”字段，看它的值是否为 1000+1，如果是就说明连接建立成功。

接下来，客户端会继续组建数据包，并设置 ACK 标志位，表示客户端正确接收了服务器发来的“确认包”。同时，将刚才服务器发来的数据包序号（2000）加 1，得到 2001，并用这个数字来填充“确认号（Ack）”字段。

客户端将数据包发出，进入 ESTABLISED 状态，表示连接已经成功建立。

4) 服务器端收到数据包，检测到已经设置了 ACK 标志位，就知道这是客户端发来的“确认包”。服务器会检测“确认号（Ack）”字段，看它的值是否为 2000+1，如果是就说明连接建立成功，服务器进入 ESTABLISED 状态。

至此，客户端和服务器都进入了 ESTABLISED 状态，连接建立成功，接下来就可以收发数据了。

最后的说明

三次握手的关键是要确认对方收到了自己的数据包，这个目标就是通过“确认号（Ack）”字段实现的。计算机会记录下自己发送的数据包序号 Seq，待收到对方的数据包后，检测“确认号（Ack）”字段，看 Ack = Seq + 1 是否成立，如果成立说明对方正确收到了自己的数据包。

第十五节 TCP 数据的传输过程

建立连接后，两台主机就可以相互传输数据了。如下图所示：

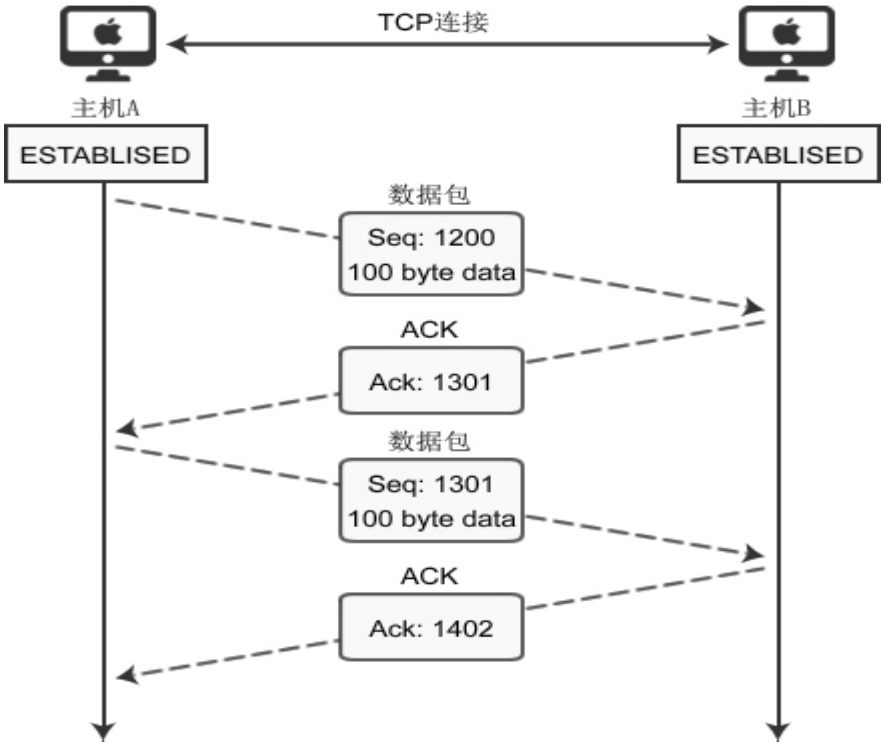


图 1：TCP 套接字的数据交换过程

上图给出了主机 A 分 2 次（分 2 个数据包）向主机 B 传递 200 字节的过程。首先，主机 A 通过 1 个数据包发送 100 个字节的数据，数据包的 Seq 号设置为 1200。主机 B 为了确认这一点，向主机 A 发送 ACK 包，并将 Ack 号设置为 1301。

为了保证数据准确到达，目标机器在收到数据包（包括 SYN 包、FIN 包、普通数据包等）包后必须立即回传 ACK 包，这样发送方才能确认数据传输成功。

此时 Ack 号为 1301 而不是 1201，原因在于 Ack 号的增量为传输的数据字节数。假设每次 Ack 号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确 100 字节全部正确传递还是丢失了一部分，比如只传递了 80 字节。因此按如下的公式确认 Ack 号：

Ack 号 = Seq 号 + 传递的字节数 + 1

与三次握手协议相同，最后加 1 是为了告诉对方要传递的 Seq 号。

下面分析传输过程中数据包丢失的情况，如下图所示：

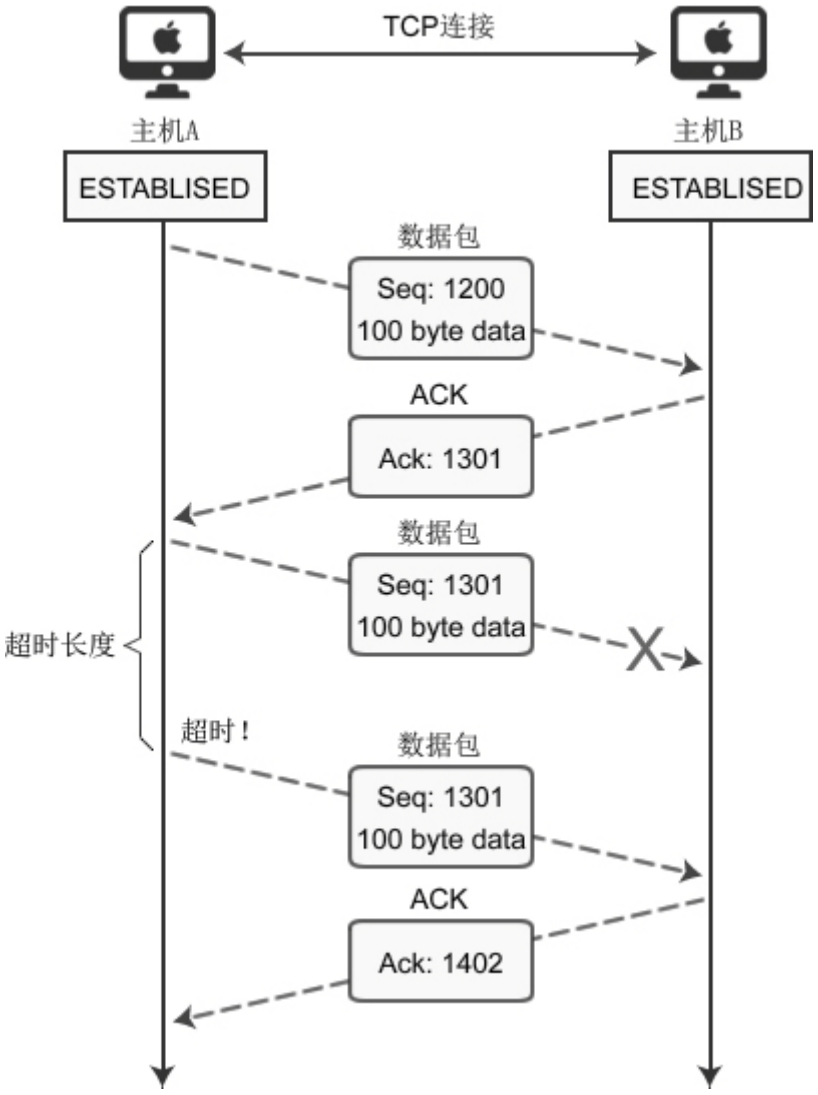


图 2：TCP 套接字数据传输过程中发生错误

上图表示通过 Seq 1301 数据包向主机 B 传递 100 字节的数据，但中间发生了错误，主机 B 未收到。经过一段时间后，主机 A 仍未收到对于 Seq 1301 的 ACK 确认，因此尝试重传数据。

为了完成数据包的重传，TCP 套接字每次发送数据包时都会启动定时器，如果在一定时间内没有收到目标机器传回的 ACK 包，那么定时器超时，数据包会重传。

上图演示的是数据包丢失的情况，也会有 ACK 包丢失的情况，一样会重传。

重传超时时间（RTO, Retransmission Time Out）

这个值太大了会导致不必要的等待，太小会导致不必要的重传，理论上最好是网络 RTT 时间，但又受制于网络距离与瞬态时延变化，所以实际上使用自适应的动态算法（例如 Jacobson 算法和 Karn 算法等）来确定超时时间。

往返时间（RTT，Round-Trip Time）表示从发送端发送数据开始，到发送端收到来自接收端的 ACK 确认包（接收端收到数据后便立即确认），总共经历的时延。

重传次数

TCP 数据包重传次数根据系统设置的不同而有所区别。有些系统，一个数据包只会被重传 3 次，如果重传 3 次后还未收到该数据包的 ACK 确认，就不再尝试重传。但有些要求很高的业务系统，会不断地重传丢失的数据包，以尽最大可能保证业务数据的正常交互。

最后需要说明的是，发送端只有在收到对方的 ACK 确认包后，才会清空输出缓冲区中的数据。

第十六节 TCP 四次握手断开连接（图解）

建立连接非常重要，它是数据正确传输的前提；断开连接同样重要，它让计算机释放不再使用的资源。如果连接不能正常断开，不仅会造成数据传输错误，还会导致套接字不能关闭，持续占用资源，如果并发量高，服务器压力堪忧。

建立连接需要三次握手，断开连接需要四次握手，可以形象的比喻为下面的对话：

[Shake 1] 套接字 A: “任务处理完毕，我希望断开连接。”

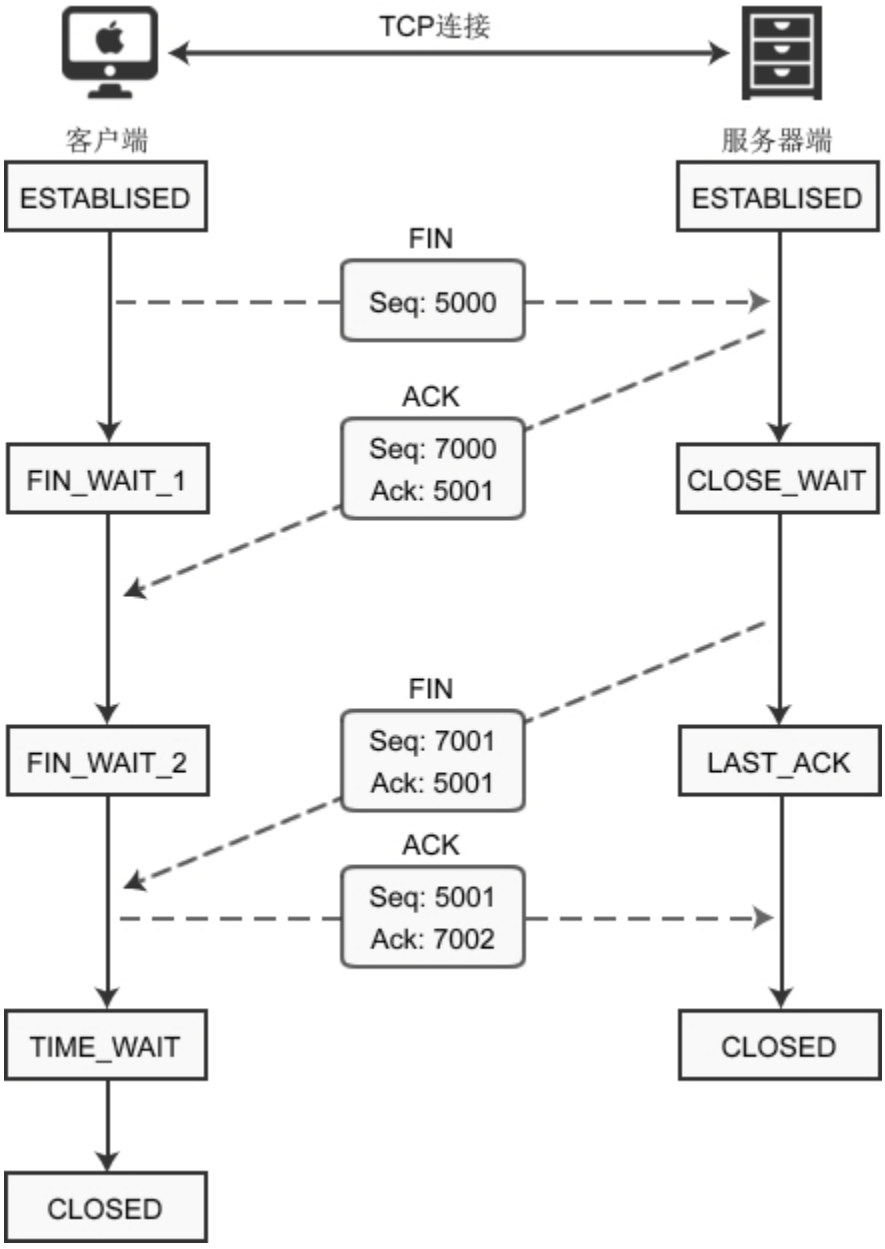
[Shake 2] 套接字 B: “哦，是吗？请稍等，我准备一下。”

等待片刻后……

[Shake 3] 套接字 B: “我准备好了，可以断开连接了。”

[Shake 4] 套接字 A: “好的，谢谢合作。”

下图演示了客户端主动断开连接的场景：



建立连接后，客户端和服务器都处于 ESTABLISED 状态。这时，客户端发起断开连接的请求：

1) 客户端调用 close() 函数后，向服务器发送 FIN 数据包，进入 FIN_WAIT_1 状态。FIN 是 Finish 的缩写，表示完成任务需要断开连接。

2) 服务器收到数据包后，检测到设置了 FIN 标志位，知道要断开连接，于是向客户端发送“确认包”，进入 CLOSE_WAIT 状态。

注意：服务器收到请求后并不是立即断开连接，而是先向客户端发送“确认包”，告诉它我知道了，我需要准备一下才能断开连接。

3) 客户端收到“确认包”后进入 FIN_WAIT_2 状态，等待服务器准备完毕后再次发送数据包。

4) 等待片刻后，服务器准备完毕，可以断开连接，于是再主动向客户端发送 FIN 包，告诉它我准备好了，断开连接吧。然后进入 LAST_ACK 状态。

5) 客户端收到服务器的 FIN 包后，再向服务器发送 ACK 包，告诉它你断开连接吧。然后进入 TIME_WAIT 状态。

6) 服务器收到客户端的 ACK 包后，就断开连接，关闭套接字，进入 CLOSED 状态。

关于 TIME_WAIT 状态的说明

客户端最后一次发送 ACK 包后进入 TIME_WAIT 状态，而不是直接进入 CLOSED 状态关闭连接，这是为什么呢？

TCP 是面向连接的传输方式，必须保证数据能够正确到达目标机器，不能丢失或出错，而网络是不稳定的，随时可能会毁坏数据，所以机器 A 每次向机器 B 发送数据包后，都要求机器 B”确认“，回传 ACK 包，告诉机器 A 我收到了，这样机器 A 才能知道数据传送成功了。如果机器 B 没有回传 ACK 包，机器 A 会重新发送，直到机器 B 回传 ACK 包。

客户端最后一次向服务器回传 ACK 包时，有可能会因为网络问题导致服务器收不到，服务器会再次发送 FIN 包，如果这时客户端完全关闭了连接，那么服务器无论如何也收不到 ACK 包了，所以客户端需要等待片刻、确认对方收到 ACK 包后才能进入 CLOSED 状态。那么，要等待多久呢？

数据包在网络中是有生存时间的，超过这个时间还未到达目标主机就会被丢弃，并通知源主机。这称为报文最大生存时间（MSL，Maximum Segment Lifetime）。TIME_WAIT 要等待 2MSL 才会进入 CLOSED 状态。ACK 包到达服务器需要 MSL 时间，服务器重传 FIN 包也需要 MSL 时间，2MSL 是数据包往返的最大时间，如果 2MSL 后还未收到服务器重传的 FIN 包，就说明服务器已经收到了 ACK 包。

第十七节 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”。

图 1：close()/closesocket() 断开连接

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：

```
1. int shutdown(int sock, int howto);  //Linux
2. 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。

至于什么时候需要调用 shutdown() 函数，下节我们会以文件传输为例进行讲解。

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。\

第十八节 socket 文件传输功能的实现

这节我们来完成 socket 文件传输程序，这是一个非常实用的例子。要实现的功能为：client 从 server 下载一个文件并保存到本地。

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:

```
1. //Server 代码
2. int nCount;
3. while( (nCount = fread(buffer, 1, BUF_SIZE, fp)) > 0 ){
4.     send(sock, buffer, nCount, 0);
5. }
6.
```

```
//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。

注意：读取完缓冲区中的数据 recv() 并不会返回 0，而是被阻塞，直到缓冲区中再次有数据。

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。

本节以 Windows 为例演示文件传输功能，Linux 与此类似，不再赘述。请看下面完整的代码。

服务器端 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;
}
```

客户端代码：

```
#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);
    }

```

```
20.     WSADATA wsaData;
21.     WSAStartup(MAKEWORD(2, 2), &wsaData);
22.     SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
23.
24.     sockaddr_in sockAddr;
25.     memset(&sockAddr, 0, sizeof(sockAddr));
26.     sockAddr.sin_family = PF_INET;
27.     sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
28.     sockAddr.sin_port = htons(1234);
29.     connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
30.
31.     //循环接收数据，直到文件传输完毕
32.     char buffer[BUF_SIZE] = {0};   //文件缓冲区
33.     int nCount;
34.     while( (nCount = recv(sock, buffer, BUF_SIZE, 0)) > 0 ){
35.         fwrite(buffer, nCount, 1, fp);
36.     }
37.     puts("File transfer success!");
38.
39.     //文件接收完毕后直接关闭套接字，无需调用 shutdown()
40.     fclose(fp);
41.     closesocket(sock);
42.     WSACleanup();
43.     system("pause");
44.     return 0;
45. }
```

在 D 盘中准备好 send.avi 文件，先运行 server，再运行 client:

Input filename to save: D:\\recv.avi↙

//稍等片刻后

File transfer success!

打开 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 行代码，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。