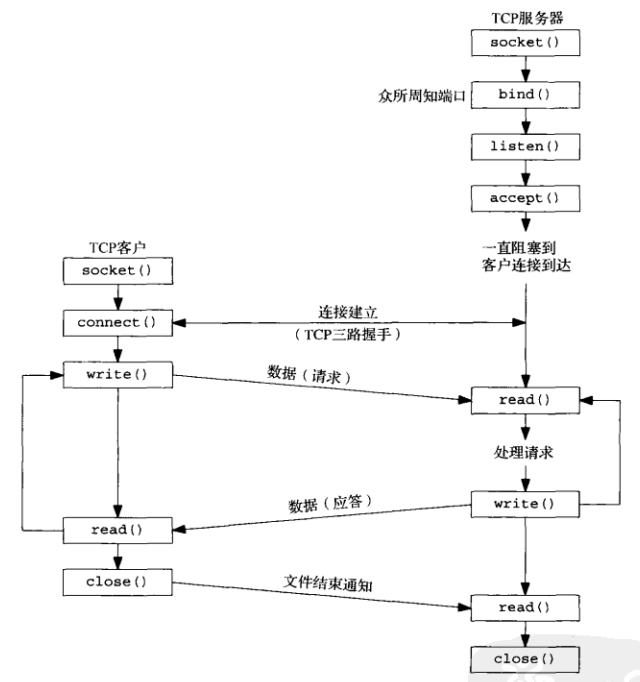
这一节首先演示一个linux下的socket示例流程，再介绍socket编程主要的接口函数，最后简单对比linux、windows下的主要代码差异和兼容的处理方式。

1. socket实例演示

首先给出一个**TCP套接字编程流程图**，以流程图来实现linux下的socket编程的”hello world!”代码示例。



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。

1.1 server端代码

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. int main(){
9. //创建套接字
10. int serv\_sock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);
11. //将套接字和IP、端口绑定
12. **struct** sockaddr\_in serv\_addr;
13. memset(&serv\_addr, 0, **sizeof**(serv\_addr)); //每个字节都用0填充
14. serv\_addr.sin\_family = AF\_INET; //使用IPv4地址
15. serv\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); //具体的IP地址
16. serv\_addr.sin\_port = htons(8080); //端口
17. bind(serv\_sock, (**struct** sockaddr\*)&serv\_addr, **sizeof**(serv\_addr));
18. //进入监听状态，等待用户发起请求
19. listen(serv\_sock, 5);
20. //接收客户端请求
21. **struct** sockaddr\_in clnt\_addr;
22. socklen\_t clnt\_sock\_len;
23. int clnt\_sock = accept(serv\_sock, (**struct** sockaddr\*)&clnt\_addr, &sock\_len);
24. //向客户端发送数据
25. char str[] = "hello world!";
26. write(clnt\_sock, str, **sizeof**(str));
27. //关闭套接字
28. close(clnt\_sock);
29. close(serv\_sock);
30. **return** 0;
31. }

1.2 client端代码

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. int main(){
8. //创建套接字
9. int sock = socket(AF\_INET, SOCK\_STREAM, 0);
10. //向服务器（特定的IP和端口）发起请求
11. **struct** sockaddr\_in serv\_addr;
12. memset(&serv\_addr, 0, **sizeof**(serv\_addr)); //每个字节都用0填充
13. serv\_addr.sin\_family = AF\_INET; //使用IPv4地址
14. serv\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); //具体的IP地址
15. serv\_addr.sin\_port = htons(8080); //端口
16. connect(sock, (**struct** sockaddr\*)&serv\_addr, **sizeof**(serv\_addr));
17. //读取服务器传回的数据
18. char buffer[40];
19. read(sock, buffer, **sizeof**(buffer)-1);
20. printf("Message form server: %s\n", buffer);
21. //关闭套接字
22. close(sock);
23. **return** 0;
24. }

启动一个终端（Shell），先编译 server.cpp 并运行：

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

[admin@localhost ~]$ ./server

#等待请求的到来

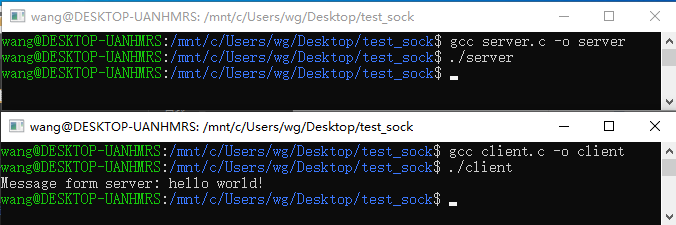
正常情况下，程序运行到 accept() 函数就会被阻塞，等待客户端发起请求。

接下再启动一个终端，编译 client.cpp 并运行：

[admin@localhost ~]$ g++ client.cpp -o client

[admin@localhost ~]$ ./client

Message form server: hello world



client 接收到从 server发送过来的字符串就运行结束了，同时，server 完成发送字符串的任务也运行结束了。可以通过两个打开的终端来观察。

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。

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。

1.3 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套接字，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，这一点已在《socket是什么》中进行了讲解。

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。

1. **socket主要函数接口**

**2.1 创建套接字**

#include <sys/socket.h>

int socket(int family, int type, int protocol); // 成功返回非负描述符，出错返回-1

其中family指明协议簇，可取AF\_INET、AF\_INET6、AF\_LOCAL等；

参数type 描述要建立的套接字的类型，如SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW等；参数protocol指定套接字使用的特定协议，可选IPPROTO\_TCP、IPPROTO\_UDP，通常置为0选择默认连接方式即可。

**2.2 绑定本地地址**

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr \*myaddr, socklen\_t addrlen); //成功返回0，出错返回-1

bind()将所创建的套接字绑定到本地协议地址，协议地址是32位的IPv4地址（AF\_INET）或128位的IPv6地址(AF\_INET6)与16位的TCPh或UDP端口的组合。第二个参数myaddr是执行特定协议的地址指针结构，不同的协议可能导致第三个参数addrlen值不同。

bind()函数通常用于服务端需要调用该函数，以绑定到总所周知的的地址和端口；客户端并不需要这么做，在进行connet或者listen时内核会选择一个临时端口和ip地址。客户端可以使用getsockname获取本地协议地址信息，服务端也可以通过getpeername获取客户端的协议地址信息。

**2.3 建立套接字连接**

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr \*myaddr, socklen\_t addrlen); // 成功返回0，出错返回-1

参数同bind函数()，TCP客户端与TCP服务端建立连接。

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*clientaddr, socklen\_t\* addrlen); // 成功返回非负的已连接套接字，出错返回-1

使用前之前已调用listen()函数，服务端从已完成连接的队列中返回下一个已完成连接，如果队列为空且阻塞则进入睡眠。**参数clientaddr用于表示已连接的客户端的协议地址，addrlen用于确定地址结构长度**（可传入NULL，不获取客户端协议地址）。返回值为已连接的套接字描述符。

一个服务器通常只有一个监听套接字，在服务器生命周期一直存在。内核会为每个由服务器进程接受的客户连接创建一个已连接套接字。当服务器完成对某个客户的服务时，响应的已连接套接字就被关闭。

**2.4 监听连接**

#include <sys/socket.h>

int listen (int sockfd, int backlog); // 成功返回0，出错返回-1

该函数仅TCP服务器使用，且需要在调用socket()和bind()之后使用，表示服务器意愿从当前的sockfd上接受客户端的请求，排队请求的个数最多为backlog。

**2.5 数据传输**

#include <unistd.h>

ssize\_t read (int fd, void \*buf, size\_t n);

ssize\_t write (int fd, const void \*buf, size\_t n);

从文件描述符fd中，读取n个字节到buf或写入buf的n个字节，成功返回实际的字节数，失败返回-1。在unix中，所有socket也当做文件，所以可以使用read或write读写socket数据。

#include <sys/socket.h>

ssize\_t send (int fd, const void \*buf, size\_t n, int flags);

ssize\_t recv (int fd, void \*buf, size\_t n, int flags);

比read write多了一个参数flags，可以理解为比read write操作更细化的函数，但仅用于套接字。

#include <sys/socket.h>

ssize\_t recvfrom(int sockfd, void\* buff, size\_t n, int flags, struct sockaddr\* from, socklent\_t\* addr\_len);

ssize\_t sendto(int sockfd, void\* buff, size\_t n, int flags, const struct sockaddr\* to, socklen\_t addrlen);

前三个参数含义同函数send/recv，flags置为0，from\to 存放对端地址结构，addrlen地址结构长度。

recvfrom接收来自对端from的信息，并存于buff中；sendto发送信息buff到对端to。

一般这样使用：

socklen\_t addr\_len = sizeof(addr);

recvfrom(sockfd,buff,sizeof(buff),0,(struct sockaddr \*)&addr,&addr\_len);

（1）根据addr\_len的值，判断客户端addr是地址类型是IPv4还是IPv6。

（2）对于UDP，recvfrom返回0是可以接受的；而TCP的read返回0则表示对端关闭连接。

（3）如果不关心对端的协议地址，可以将recvfrom的最后两个参数置为空指针。

（4）recvfrom和sendto都可以用于TCP，但通常不这么做。

**一般在UDP套接字中使用recvfrom/sendto；在TCP套接字中使用read/write.**

**2.6 关闭套接字**

#include <unistd.h>

int close(int sockfd)

int shutdown(int sockfd)

close()关闭套接字sockfd，并释放当前进程分配给该套接字的资源；如果涉及一个打开的TCP连接，则该连接被释放。

shutdown是一种优雅地单方向或者双方向关闭socket的方法。 而close则立即双方向强制关闭socket并释放相关资源。

如果有多个进程共享一个socket，shutdown影响所有进程，而close只影响本进程。

1. **平台差异**

主要给出主要接口函数使用下的差异，并给出跨平台下的解决方式。

**3.1 主要差异**

（1）头文件

windows下winsock.h/winsock2.h

linux下sys/socket.h 错误处理：errno.h

（2）初始化、库加载

Windows下需要初始化,而linux下不需要。

程序开始需要调用初始化函数WSAStartup()，对应的退出清理用WSACleanup();

3)socket类型

windows下SOCKET

linux下int

4)关闭socket

windows下closesocket(...)

linux下close(...)

5)获取错误码  
windows下WSAGetLastError()

linux下errno变量extern int errno;

int geterror(){return errno;}

6)设置非阻塞

windows下

int ul = 1

ioctlsocket(server\_socket,FIONBIO,&ul);

linux下

#include <fcntl.h>

fcntl(server\_socket,F\_SETFL, O\_NONBLOCK);

7)编译连接

windows下ws2\_32.lib  
linux下

连接是使用参数：-lstdc  
运行时需要libstdc++.so.5，可在/usr/lib目录中创建一个链接。属于系统环境，不需要额外处理。

**3.2 解决方案**

解决上述跨平台之间的代码差异，可以用如下代码简单解决

#ifdef \_WIN32

#if defined(\_MSC\_VER)

#ifdef \_WIN64

using ssize\_t = \_\_int64;

#else

using ssize\_t = int;

#endif

#include <io.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#ifdef \_MSC\_VER

#pragma comment(lib, "ws2\_32.lib")

#endif

using socket\_t = SOCKET;

#else // not \_WIN32

#include <sys/socket.h>

#include <netinet/in.h> // sockaddr\_in, inet\_addr

#include <arpa/inet.h>

#include <netdb.h>

#include <unistd.h> // close,shutdown, write, read

#include <cstring.h>

#endif

编写通用函数

int close\_socket(socket\_t sock) {

#ifdef \_WIN32

return closesocket(sock);

#else

return close(sock);

#endif

}

int shutdown\_socket(socket\_t sock) {

#ifdef \_WIN32

return shutdown(sock, SD\_BOTH);

#else

return shutdown(sock, SHUT\_RDWR);

#endif

}

void set\_nonblocking(socket\_t sock, bool nonblocking) {

#ifdef \_WIN32

auto flags = nonblocking ? 1UL : 0UL;

ioctlsocket(sock, FIONBIO, &flags);

#else

auto flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL,

nonblocking ? (flags | O\_NONBLOCK) : (flags & (~O\_NONBLOCK)));

#endif

}

bool get\_error() {

#ifdef \_WIN32

return WSAGetLastError;

#else

return errno;

#endif

}

源代码中添加初始化的代码

#ifdef \_WIN32

class WSInit {

public:

WSInit() {

WSADATA wsaData;

WSAStartup(0x0002, &wsaData);

}

~WSInit() { WSACleanup(); }

};

static WSInit wsinit\_; // 全局变量，windows下程序执行时构造初始化，退出时析构

#endif