**一切皆Socket！”**

话虽些许夸张，但是事实也是，现在的网络编程几乎都是用的socket。

——有感于实际编程和开源项目研究。

我们深谙信息交流的价值，那网络中进程之间如何通信，如我们每天打开浏览器浏览网页时，浏览器的进程怎么与web服务器通信的？当你用QQ聊天时，QQ进程怎么与服务器或你好友所在的QQ进程通信？这些都得靠socket？那什么是socket？socket的类型有哪些？还有socket的基本函数，这些都是本文想介绍的。本文的主要内容如下：

* 1、网络中进程之间如何通信？
* 2、Socket是什么？
* 3、socket的基本操作
  + 3.1、socket()函数
  + 3.2、bind()函数
  + 3.3、listen()、connect()函数
  + 3.4、accept()函数
  + 3.5、read()、write()函数等
  + 3.6、close()函数
* 4、socket中TCP的三次握手建立连接详解
* 5、socket中TCP的四次握手释放连接详解
* 6、一个例子（实践一下）

1、网络中进程之间如何通信？

**本地的进程间通信（IPC）**有很多种方式，但可以总结为下面4类：

* **消息传递（管道、FIFO、消息队列）**
* **同步（互斥量、条件变量、读写锁、文件和写记录锁、信号量）**
* **共享内存（匿名的和具名的）**
* **远程过程调用（Solaris门和Sun RPC）**

但这些都不是本文的主题！我们要讨论的是网络中进程之间如何通信？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！在本**地可以通过进程PID来唯一标识一个进程**，但是在网络中这是行不通的。其实TCP/IP协议族已经帮我们解决了这个问题，**网络层的“ip地址”**可以唯一标识网络中的主机，而**传输层的“协议+端口”**可以唯一标识主机中的应用程序（进程）。这样利用**三元组**（**ip地址，协议，端口**）就**可以标识网络的进程**了，网络中的**进程通信**就可以利用**这个标志与其它进程进行交互**。

使用**TCP/IP协议的应用程序通常采用应用编程接口**：UNIX  BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来**实现网络进程之间的通信**。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，**网络中进程通信是无处不在**，这就是我为什么说“一切皆socket”。

2、什么是Socket？

上面我们已经知道网络中的进程是通过socket来通信的，那什么是socket呢？**socket起源于Unix，而Unix/Linux基本哲学之一就是“一切皆文件”，**都可以用**“打开open –> 读写write/read –> 关闭close”**模式来操作。我的理解就是Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭），这些函数我们在后面进行介绍。

socket一词的起源

在组网领域的首次使用是在1970年2月12日发布的文献[IETF RFC33](http://datatracker.ietf.org/doc/rfc33/" \t "_blank)中发现的，撰写者为Stephen Carr、Steve Crocker和Vint Cerf。根据美国计算机历史博物馆的记载，Croker写道：**“命名空间的元素都可称为套接字接口。一个套接字接口构成一个连接的一端，而一个连接可完全由一对套接字接口规定。**”计算机历史博物馆补充道：“这比BSD的套接字接口定义早了大约12年。”

3、socket的基本操作

既然socket是“open—write/read—close”模式的一种实现，那么socket就提供了这些操作对应的函数接口。**下面以TCP为例，介绍几个基本的socket接口函数。**

3.1、socket()函数

int **socket**(int domain, int type, int protocol);

调用该函数前，先要加载winsock，wsastartup（）.socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

如果成功就返回生成的SOCKET，如果失败就返回INVALID\_SOCKET(-1).

正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

* domain：**即协议域，又称为协议族（family）**。常用的协议族有，AF\_INET、AF\_INET6、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。**协议族决定了socket的地址类型**，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。
* type：**指定socket类型。常用的socket类型有**，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。

类型 解释

SOCK\_STREAM 提供有序的、可靠的、双向的和基于连接的[字节流](http://baike.baidu.com/view/1645587.htm" \t "_blank)，使用[带外数据](http://baike.baidu.com/view/567593.htm)传送机制，为Internet地址族使用TCP。

SOCK\_DGRAM 支持无连接的、不可靠的和使用固定大小（通常很小）缓冲区的[数据报服务](http://baike.baidu.com/view/2495672.htm" \t "_blank)，为Internet地址族使用UDP。

SOCK\_STREAM类型的套接口为全双向的字节流。对于流类套接口，在接收或发送数据前必需处于已连接状态。用[connect()](http://baike.baidu.com/view/569190.htm" \t "_blank)调用建立与另一套接口的连接，连接成功后，即可用[send()](http://baike.baidu.com/view/569214.htm" \t "_blank)和recv()传送数据。当会话结束后，调用closesocket()。[带外数据](http://baike.baidu.com/view/567593.htm" \t "_blank)根据规定用send()和recv()来接收。                                                                                                                原始套接字（SOCK\_RAW）：原始套接字与标准套接字（标准套接字指的是前面介绍的流套接字和数据包套接字）的[区别](http://baike.haosou.com/doc/6607128.html" \t "_blank)在于：原始套接字可以读写[内核](http://baike.haosou.com/doc/665564.html)没有处理的IP数据包，而流套接字只能读取TCP协议的数据，数据包套接字只能读取UDP协议的数据。因此，如果要访问其他协议发送数据必须使用原始套接字。

实现SOCK\_STREAM类型[套接口](http://baike.baidu.com/view/567586.htm" \t "_blank)的通讯协议保证数据不会丢失也不会重复。如果终端协议有缓冲区空间，且数据不能在一定时间成功发送，则认为连接中断，其后续的调用也将以WSAETIMEOUT错误返回。

SOCK\_DGRAM类型套接口允许使用[sendto()](http://baike.baidu.com/view/569215.htm" \t "_blank)和recvfrom()从任意端口发送或接收数据报。如果这样一个套接口用[connect()](http://baike.baidu.com/view/569190.htm)与一个指定端口连接，则可用[send()](http://baike.baidu.com/view/569214.htm" \t "_blank)和recv()与该端口进行数据报的发送与接收。

* protocol：**故名思意，就是指定协议。**常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

1 参数protocol用来指明所要接收的协议包，如果是象IPPROTO\_TCP(6)这种非0、非255的协议，当**[操作系统](http://lib.csdn.net/base/operatingsystem" \t "_blank" \o "操作系统知识库)**内核碰到ip头中protocol域和创建socket所使用参数protocol相同的IP包，就会交给这个raw socket来处理，因此，一般来说，要想接收什么样的数据包，就应该在参数protocol里来指定相应的协议。当内核向此raw socket交付数据包的时候，是包括整个IP头的，并且已经是重组好的IP包。  
2 如果protocol是IPPROTO\_RAW(255)，这时候，这个socket只能用来发送IP包，而不能接收任何的数据。发送的数据需要自己填充IP包头，并且自己计算校验和。  
3 对于protocol为0（IPPROTO\_IP)的raw socket。用于接收任何的IP数据包。其中的校验和和协议分析由程序自己完成。

这是include/[**Linux**](http://lib.csdn.net/base/linux)/in.h里的定义:

/\* Standard well-defined IP protocols. \*/  
enum {  
IPPROTO\_IP = 0,   /\* Dummy protocol for TCP   \*/  
IPPROTO\_ICMP = 1,   /\* Internet Control Message Protocol \*/  
IPPROTO\_IGMP = 2,   /\* Internet Group Management Protocol \*/  
IPPROTO\_IPIP = 4,   /\* IPIP tunnels (older KA9Q tunnels use 94) \*/  
IPPROTO\_TCP = 6,   /\* Transmission Control Protocol \*/  
IPPROTO\_EGP = 8,   /\* Exterior Gateway Protocol   \*/  
IPPROTO\_PUP = 12,   /\* PUP protocol     \*/  
IPPROTO\_UDP = 17,   /\* User Datagram Protocol   \*/  
IPPROTO\_IDP = 22,   /\* XNS IDP protocol    \*/  
IPPROTO\_DCCP = 33,   /\* Datagram Congestion Control Protocol \*/  
IPPROTO\_RSVP = 46,   /\* RSVP protocol    \*/  
IPPROTO\_GRE = 47,   /\* Cisco GRE tunnels (rfc 1701,1702) \*/

IPPROTO\_IPV6 = 41,   /\* IPv6-in-IPv4 tunnelling   \*/

IPPROTO\_ESP = 50,            /\* Encapsulation Security Payload protocol \*/  
IPPROTO\_AH = 51,             /\* Authentication Header protocol       \*/  
IPPROTO\_BEETPH = 94,        /\* IP option pseudo header for BEET \*/  
IPPROTO\_PIM    = 103,   /\* Protocol Independent Multicast \*/

IPPROTO\_COMP   = 108,                /\* Compression Header protocol \*/  
IPPROTO\_SCTP   = 132,   /\* Stream Control Transport Protocol \*/  
IPPROTO\_UDPLITE = 136, /\* UDP-Lite (RFC 3828)    \*/

IPPROTO\_RAW = 255,   /\* Raw IP packets    \*/  
IPPROTO\_MAX  
};

               注意：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

**当我们调用socket创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址**，其实在**调用socket函数创建socket时，内核还并未给socket分配源地址和源端口**。而对于UDP，我猜测在调用sendto发送数据时，在未捆绑端口的情况下，内核也会随机分配端口。。**如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、listen()时系统会自动随机分配一个端口。**

3.2、bind()函数

正如上面所说bind()函数**把一个地址族中的特定地址赋给socket。**例如对应AF\_INET、AF\_INET6就是**把一个ipv4或ipv6地址和端口号组合赋给socket**。

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

函数的三个参数分别为：

* sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。
* addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同，如ipv4对应的是：
* struct sockaddr\_in {
* sa\_family\_t sin\_family; /\* address family: AF\_INET \*/
* in\_port\_t sin\_port; /\* port in network byte order \*/
* struct in\_addr sin\_addr; /\* internet address \*/
* };
* /\* Internet address. \*/
* struct in\_addr {
* uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

struct in6\_addr {

unsigned char s6\_addr[16]; /\* IPv6 address \*/

};

Unix域对应的是：

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

* addrlen：对应的是地址的长度。

**通常服务器在启动的时候都会绑定一个众所周知的地址（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器**；而客户端就不用指定，有系统自动分配一个端口号和自身的ip地址组合。这就是为什么通常**服务器端在listen之前会调用bind()，**而**客户端就不会调用，而是在connect()时由系统随机生成一个。**

     在《UNIX网络编程》这本书中提到：“**如果一个TCP客户或者服务器未曾调用bind捆绑一个端口，当调用connect或listen时，内核就要为相应的套接字选择一个临时接口。”**从这句话中可以判断出，其实在**调用socket函数创建socket时，内核还并未给socket分配源地址和源端口。**而对于UDP，我猜测在调用sendto发送数据时，在未捆绑端口的情况下，内核也会随机分配端口。  
　　而我遇到的特殊应用要求我在用UDP发送数据之前要告诉对方我的发送端口，这也就意味着我在sendto之前必须要捆绑端口，因此我在发送数据之前就得调用bind函数绑定一下端口了。但是我就在想**内核既然有随机分配端口的能力，而我需要的也只是让它绑定一下而不用绑定在固定端口的业务**，socket中应该能够提供这种业务。然后果然我发现bind就具备这种能力，当bind的参数中端口地址为0的时候，这时候就是由内核分配端口。这样我就不用考虑端口地址重复的问题，而放心的把这个问题交给内核处理了。  
　　就在发现bind的这个机制的同时，我发现其实bind对于源地址也同样具备这种处理方式，当系统具有多IP（多网卡）的情况，当我们把bind函数中的ip参数置0时，就是由内核自己选择分配IP。而之前一直觉得很神奇的INADDR\_ANY其实一点也不神奇，它的值其实就是0。所以当我们只有单一IP的时候，我们就可以用INADDR\_ANY去代替那个单一的IP，因为内核分配的时候只能选择这一个IP。从而造成了INADDR\_ANY就是本机IP的现象。

网络字节序与主机字节序

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。

3.3、listen()、connect()函数

**如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，**如果**客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。**

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的**第一个参数即为要监听的socket描述字**，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

**connect函数的第一个参数即为客户端的socket描述字**，第二参数为服务器的socket地址，第三个参数为socket地址的长度。**客户端通过调用connect函数来建立与TCP服务器的连接。**

3.4、accept()函数

**TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。**TCP客户端依次调用**socket()、connect()之后就向TCP服务器发送了一个连接请求**。TCP服务器监听到这个请求之后，就会**调用accept()函数取接收请求**，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

**accept函数的第一个参数为服务器的socket描述字，第二个参数为指向struct sockaddr \*的指针，用于返回客户端的协议地址，第三个参数为协议地址的长度。**如果accpet成功，那么其返回值是由内核自动生成的一个全新的描述字，代表与返回客户的TCP连接。

注意：accept的**第一个参数为服务器的socket描述字**，是服务器开始调用socket()函数生成的，称为**监听socket描述字；**而accept函数**返回的**是**已连接的socket描述字。**一个服务器**通常仅仅只创建一个监听socket描述字**，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

3.5、read()、write()等函数

万事具备只欠东风，至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

* read()/write()
* recv()/send()
* readv()/writev()
* recvmsg()/sendmsg()
* recvfrom()/sendto()

我推荐使用recvmsg()/sendmsg()函数，这两个函数是最通用的I/O函数，实际上可以把上面的其它函数都替换成这两个函数。它们的声明如下：

#include <unistd.h>

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

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

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,

const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

**read函数是负责从fd中读取内容.**当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

**write函数将buff中的nbytes字节内容写入文件描述符fd.**成功时返回写的字节数。失败时返回-1，并设置errno变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接)。

其它的我就不一一介绍这几对I/O函数了，具体参见man文档或者baidu、Google，下面的例子中将使用到send/recv。

3.6、close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，**完成了读写操作就要关闭相应的socket描述字**，好比操作完打开的文件要调用fclose关闭打开的文件。

#include <unistd.h>

int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。

4、socket中TCP的三次握手建立连接详解

我们知道tcp建立连接要进行“三次握手”，即交换三个分组。大致流程如下：

* 客户端向服务器发送一个SYN J
* 服务器向客户端响应一个SYN K，并对SYN J进行确认ACK J+1
* 客户端再想服务器发一个确认ACK K+1

只有就完了三次握手，**但是这个三次握手发生在socket的那几个函数中呢？**请看下图：

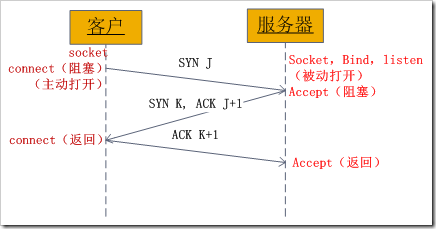
[](http://images.cnblogs.com/cnblogs_com/skynet/201012/201012122157467258.png)

图1、socket中发送的TCP三次握手

从图中可以看出，**当客户端调用connect时，触发了连接请求，向服务器发送了SYN J包，**这时**connect进入阻塞状态**；服务器监听到连接请求，即收到SYN J包，调用**accept函数接收请求向客户端发送SYN K** ，ACK J+1，**这时accept进入阻塞状态；**客户端收到服务器的SYN K ，ACK J+1之后，**这时connect返回，**并对SYN K进行确认；服务器收到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

总结：客户端的connect在三次握手的第二个次返回，而服务器端的accept在三次握手的第三次返回。

5、socket中TCP的四次握手释放连接详解

上面介绍了socket中TCP的三次握手建立过程，及其涉及的socket函数。现在我们介绍socket中的四次握手释放连接的过程，请看下图：

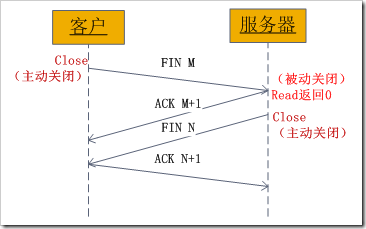
[](http://images.cnblogs.com/cnblogs_com/skynet/201012/201012122157487616.png)

图2、socket中发送的TCP四次握手

图示过程如下：

* 某个应用进程首先调用close主动关闭连接，这时TCP发送一个FIN M；
* 另一端接收到FIN M之后，执行被动关闭，对这个FIN进行确认。它的接收也作为文件结束符传递给应用进程，因为FIN的接收意味着应用进程在相应的连接上再也接收不到额外数据；
* 一段时间之后，接收到文件结束符的应用进程调用close关闭它的socket。这导致它的TCP也发送一个FIN N；
* 接收到这个FIN的源发送端TCP对它进行确认。

这样每个方向上都有一个FIN和ACK。

6、一个例子（实践一下）

说了这么多了，动手实践一下。下面编写一个简单的服务器、客户端（使用TCP）——服务器端一直监听本机的6666号端口，如果收到连接请求，将接收请求并接收客户端发来的消息；客户端与服务器端建立连接并发送一条消息。

服务器端代码：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif服务器端

[复制代码](http://blog.csdn.net/sight_/article/details/8138802)

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#define MAXLINE 4096

int main(int argc, char\*\* argv)

{

int listenfd, connfd;

struct sockaddr\_in servaddr;

char buff[4096];

int n;

if( (listenfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1 ){

printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);

exit(0);

}

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(6666);

if( bind(listenfd, (struct sockaddr\*)&servaddr, sizeof(servaddr)) == -1){

printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);

exit(0);

}

if( listen(listenfd, 10) == -1){

printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);

exit(0);

}

printf("======waiting for client's request======\n");

while(1){

if( (connfd = accept(listenfd, (struct sockaddr\*)NULL, NULL)) == -1){

printf("accept socket error: %s(errno: %d)",strerror(errno),errno);

continue;

}

n = recv(connfd, buff, MAXLINE, 0);

buff[n] = '\0';

printf("recv msg from client: %s\n", buff);

close(connfd);

}

close(listenfd);

}

[复制代码](http://blog.csdn.net/sight_/article/details/8138802)

客户端代码：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

#include<stdio.h> #include<stdlib.h> #include<string.h> #include<errno.h> #include<sys/types.h>#include<sys/socket.h> #include<netinet/in.h> #define MAXLINE 4096 int main(int argc, char\*\* argv) { int sockfd, n; char recvline[4096], sendline[4096]; struct sockaddr\_in servaddr; if( argc != 2){ printf("usage: ./client <ipaddress>\n"); exit(0); } if( (sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0){ printf("create socket error: %s(errno: %d)\n", strerror(errno),errno); exit(0); } memset(&servaddr, 0, sizeof(servaddr)); servaddr.sin\_family = AF\_INET; servaddr.sin\_port = htons(6666); if( inet\_pton(AF\_INET, argv[1], &servaddr.sin\_addr) <= 0){ printf("inet\_pton error for %s\n",argv[1]); exit(0); } if( connect(sockfd, (struct sockaddr\*)&servaddr, sizeof(servaddr)) <0){ printf("connect error: %s(errno: %d)\n",strerror(errno),errno); exit(0); } printf("send msg to server: \n"); fgets(sendline, 4096, stdin); if( send(sockfd, sendline, strlen(sendline), 0) <0) { printf("send msg error: %s(errno: %d)\n", strerror(errno), errno); exit(0); } close(sockfd); exit(0); }

复制代码

当然上面的代码很简单，也有很多缺点，这就只是简单的演示socket的基本函数使用。其实不管有多复杂的网络程序，都使用的这些基本函数。上面的服务器使用的是迭代模式的，即只有处理完一个客户端请求才会去处理下一个客户端的请求，这样的服务器处理能力是很弱的，现实中的服务器都需要有并发处理能力！为了需要并发处理，服务器需要fork()一个新的进程或者线程去处理请求等。

windows下：

**服务端：**

**[cpp]** [view plain](https://blog.csdn.net/qq_26399665/article/details/52421723) [copy](https://blog.csdn.net/qq_26399665/article/details/52421723)

1. // sockettest.cpp : 定义控制台应用程序的入口点。
2. //
4. #include "stdafx.h"
5. #include<iostream>
6. #include<cstdlib>
7. #include<cstring>
8. #include<cerrno>
9. #include<sys/types.h>
10. #include<winsock2.h>
11. //#include<netdb.h>
12. #pragma comment(lib,"ws2\_32.lib")
13. **using** **namespace** std;
15. #define MAXSIZE 4096
16. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
17. {
18. **WORD** requestVersion=MAKEWORD(2,2);
19. WSADATA wsadata;
20. **int** err=WSAStartup(requestVersion,&wsadata);
21. **if** (err!=0)
22. {
23. printf("WSAStartup failed witherror: %d\n", err);
24. **return** 1;
25. }
27. **int** listenfd,connfd;
28. sockaddr\_in servaddr;
29. **char** buff[4096];
30. memset(buff,0,**sizeof**(**char**)\*4096);
31. **int** n;
32. **if** ((listenfd=socket(AF\_INET,SOCK\_STREAM,0))==-1)
33. {
34. **char** tmp[4096]={0};
35. strerror\_s(tmp,errno);
36. cout<<"create socket error:";
37. printf("%s",tmp);
38. cout<<" : "<<errno<<endl;
39. exit(0);
40. }
41. memset(&servaddr,0,**sizeof**(servaddr));
42. servaddr.sin\_port=htons(6666);
43. servaddr.sin\_addr.S\_un.S\_addr=htonl(INADDR\_ANY);
44. servaddr.sin\_family=AF\_INET;
45. **if** (bind(listenfd,(sockaddr\*)&servaddr,**sizeof**(servaddr)))
46. {
47. **char** tmp[4096]={0};
48. strerror\_s(tmp,errno);
49. cout<<"bind socket error:";
50. printf("%s",tmp);
51. cout<<" : "<<errno<<endl;
52. exit(0);
53. }
54. **if** (listen(listenfd,10)==-1)
55. {
56. **char** tmp[4096]={0};
57. strerror\_s(tmp,errno);
58. cout<<"listen socket error:";
59. printf("%s",tmp);
60. cout<<" : "<<errno<<endl;
61. exit(0);
62. }
63. cout<<"==========waiting for client's request====================="<<endl;
64. **while**(1)
65. {
66. **if** ((connfd=accept(listenfd,(sockaddr\*)NULL,NULL))==-1)
67. {
68. **char** tmp[4096]={0};
69. strerror\_s(tmp,errno);
70. cout<<"accept socket error:";
71. printf("%s",tmp);
72. cout<<" : "<<errno<<endl;
73. **continue**;
74. }
75. memset(buff,0,**sizeof**(buff));
76. n=recv(connfd,buff,MAXSIZE,0);
77. buff[n]='\0';
78. cout<<"say:";
79. printf("%s",buff);
80. cout<<endl;
81. closesocket(connfd);
82. }
83. closesocket(listenfd);
84. **return** 0;
85. }

**客户端：**

**[cpp]** [view plain](https://blog.csdn.net/qq_26399665/article/details/52421723) [copy](https://blog.csdn.net/qq_26399665/article/details/52421723)

1. // ClientSocket.cpp : 定义控制台应用程序的入口点。
2. //
4. #include "stdafx.h"
5. #include<iostream>
6. #include<cstring>
7. #include<cerrno>
8. #include<sys/types.h>
9. #include<winsock2.h>
10. #pragma comment(lib,"ws2\_32")
11. **using** **namespace** std;
12. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
13. {
14. SOCKET sockfd,n;
15. **char** recvline[4096],sendline[4096];
16. sockaddr\_in servaddr;
17. /\*if (argc!=2)
18. {
19. cout<<"usage: ./client <ipaddress>\n";
20. exit(0);
21. }\*/
22. **WORD** wRequestVersion=MAKEWORD(2,2);
23. WSADATA wsadata;
24. **if** (WSAStartup(wRequestVersion,&wsadata)!=0)
25. {
26. cout<<"socket initial failed"<<endl;
27. exit(0);
28. }
29. memset(&servaddr,0,**sizeof**(servaddr));
30. servaddr.sin\_family=AF\_INET;
31. servaddr.sin\_addr.S\_un.S\_addr=inet\_addr("10.129.119.207");
32. servaddr.sin\_port=htons(6666);
33. cout<<"=============send msg to server========="<<endl;
34. **while**(1)
35. {
36. **if** ((sockfd=socket(AF\_INET,SOCK\_STREAM,0))<0)
37. {
38. cout<<"create socket failed"<<endl;
39. exit(0);
40. }
41. **if** (connect(sockfd,(sockaddr\*)&servaddr,**sizeof**(servaddr))<0)
42. {
43. cout<<"connect failed"<<endl;
44. cout<<WSAGetLastError();
45. exit(0);
46. }
47. memset(sendline,0,**sizeof**(sendline));
48. fgets(sendline,4096,stdin);
49. sendline[strlen(sendline)-1]=0;
50. **if** (strcmp(sendline,"exit")==0)
51. **break**;
52. **if** (send(sockfd,sendline,strlen(sendline),0)<0)
53. {
54. cout<<"send msg failed"<<endl;
55. cout<<WSAGetLastError();
56. exit(0);
57. }
58. }
59. closesocket(sockfd);
60. **return** 0;
61. }