

Linux 的 SOCKET 编程详解

1. 网络中进程之间如何通信

进程通信的概念最初来源于单机系统。由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进程

之间既互不干扰又协调一致工作，**操作系统**为进程通信提供了相应设施，如

UNIX BSD 有：管道（pipe）、命名管道（named pipe）软中断信号（signal）

UNIX system V 有：消息（message）、共享存储区（shared memory）和信号量（semaphore）等。

他们都仅限于用在本机进程之间通信。网间进程通信要解决的是不同主机进程间的相互通信问题（可把同机进程通信看作是其中的特例）。为此，首先要解决的是网间进程标识问题。同一主机上，不同进程可用进程号（process ID）唯一标识。但在网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机 A 赋予某进程号 5，在 B 机中也可以存在 5 号进程，因此，“5 号进程”这句话就没有意义了。其次，操作系统支持的网络协议众多，不同协议的工作方式不同，地址格式也不同。因此，网间进程通信还要解决多重协议的识别问题。

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。

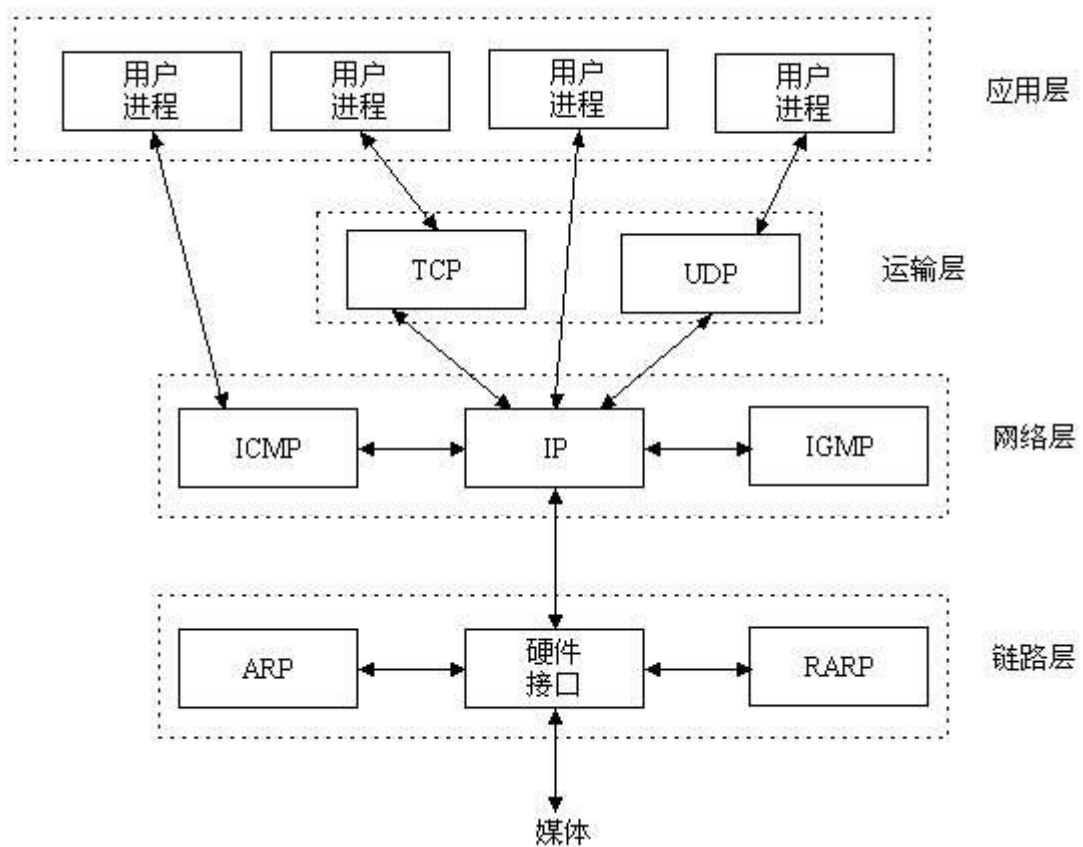
使用 TCP/IP 协议的应用程序通常采用应用编程接口：UNIX BSD 的套接字（socket）和 UNIX System V 的 TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用 socket，而现在又是网络时代，网络中进程通信是无处不在，这就是我为什么说“一切皆 socket”。

2. 什么是 TCP/IP、UDP

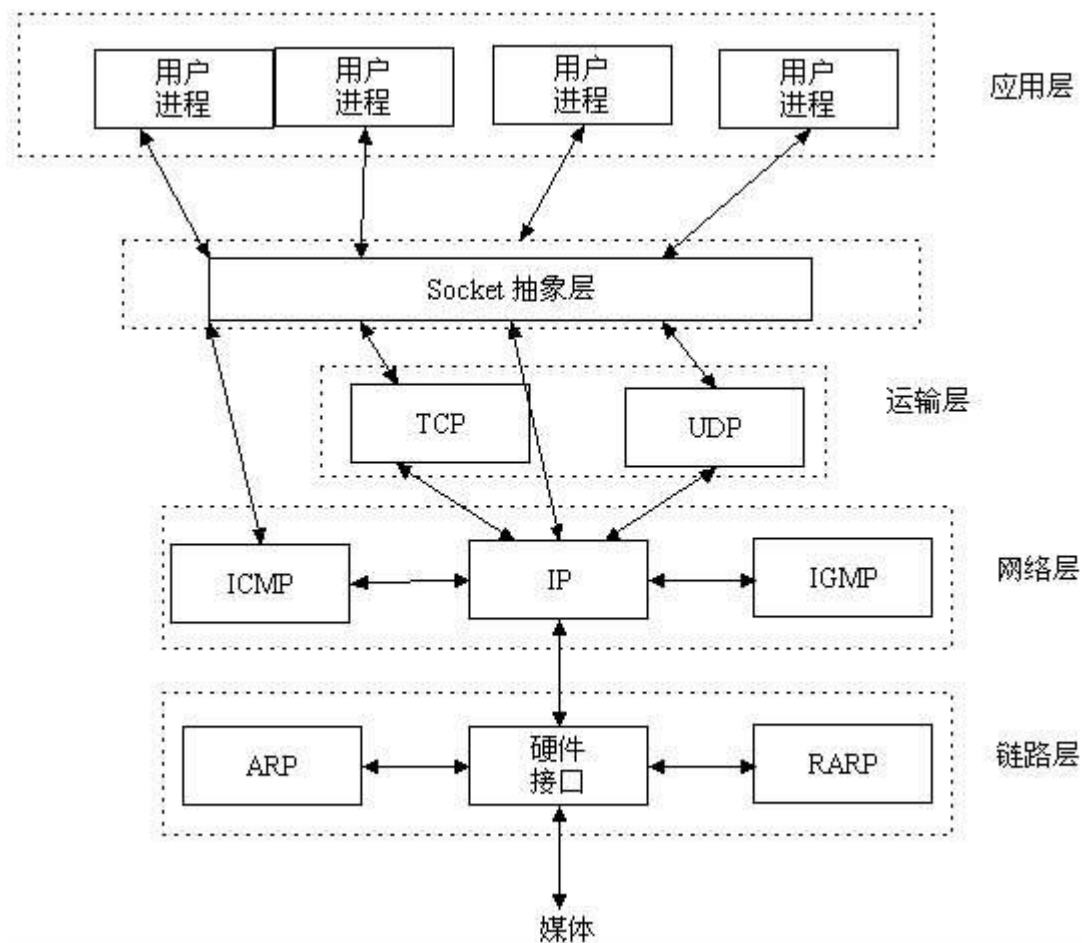
TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WANs）设计的。

TCP/IP 协议存在于 OS 中，网络服务通过 OS 提供，在 OS 中增加支持 TCP/IP 的系统调用——Berkeley 套接字，如 Socket, Connect, Send, Recv 等

UDP（User Data Protocol，用户数据报协议）是与 TCP 相对应的协议。它是属于 TCP/IP 协议族中的一种。如图：



TCP/IP 协议族包括运输层、网络层、链路层，而 socket 所在位置如图，Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层。



3. Socket 是什么

1、socket 套接字：

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。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）。

说白了 Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。

注意：其实 socket 也没有层的概念，它只是一个 facade 设计模式的应用，让编程变的更简单。是一个软件抽象层。在网络编程中，我们大量用的都是通过 socket 实现的。

2、套接字描述符

其实就是一个整数，我们最熟悉的句柄是 0、1、2 三个，0 是标准输入，1 是标准输出，2 是标准错误输出。0、1、2 是整数表示的，对应的 FILE *结构的表示就是 stdin、stdout、stderr

套接字 API 最初是作为 UNIX 操作系统的一部分而开发的，所以套接字 API 与系统的其他 I/O 设备集成在一起。特别是，当应用程序要为因特网通信而创建一个套接字（socket）时，操作系统就返回一个小整数作为描述符（descriptor）来标识这个套接字。然后，应用程序以该描述符作为传递参数，通过调用函数来完成某种操作（例如通过网络传送数据或接收输入的数据）。

在许多操作系统中，套接字描述符和其他 I/O 描述符是集成在一起的，所以应用程序可以对文件进行套接字 I/O 或 I/O 读/写操作。

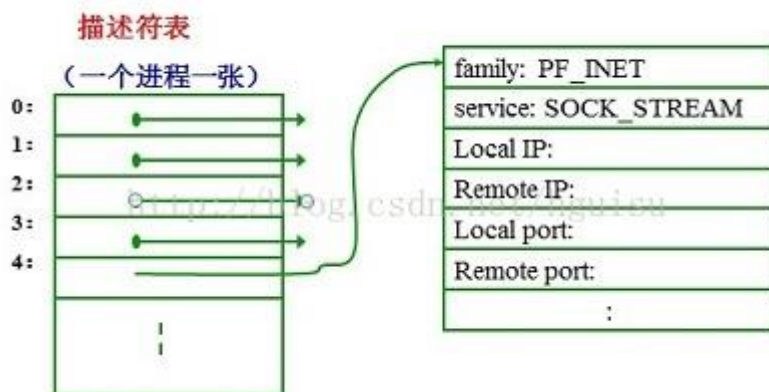
当应用程序要创建一个套接字时，操作系统就返回一个小整数作为描述符，应用程序则使用这个描述符来引用该套接字需要 I/O 请求的应用程序请求操作系统打开一个文件。操作系统就创建一个文件描述符提供给应用程序访问文件。从应用程序的角度看，文件描述符是一个整数，应用程序可以用它来读写文件。下图显示，操作系统如何把文件描述符实现为一个指针数组，这些指针指向内部**数据结构**。



对于每个程序系统都有一张单独的表。精确地讲，系统为每个运行的进程维护一张单独的文件描述符表。当进程打开一个文件时，系统把一个指向此文件内部数据结构的指针写入文件描述符表，并把该表的索引值返回给调用者。应用程序只需记住这个描述符，并在以后操作该文件时使用它。操作系统把该描述符作为索引访问进程描述符表，通过指针找到保存该文件所有的信息的数据结构。

针对套接字的系统数据结构：

1)、套接字 API 里有个函数 **socket**，它就是用来创建一个套接字。套接字设计的总体思路是，单个系统调用就可以创建任何套接字，因为套接字是相当笼统的。一旦套接字创建后，应用程序还需要调用其他函数来指定具体细节。例如调用 **socket** 将创建一个新的描述符条目：



2)、虽然套接字的内部数据结构包含很多字段，但是系统创建套接字后，大多数字字段没有填写。应用程序创建套接字后在该套接字可以使用之前，必须调用其他的过程来填充这些字段。

3、文件描述符和文件指针的区别：

文件描述符：在 linux 系统中打开文件就会获得文件描述符，它是个很小的正整数。每个进程在 PCB(Process Control Block) 中保存着一份文件描述符表，文件描述符就是这个表的索引，每个表项都有一个指向已打开文件的指针。

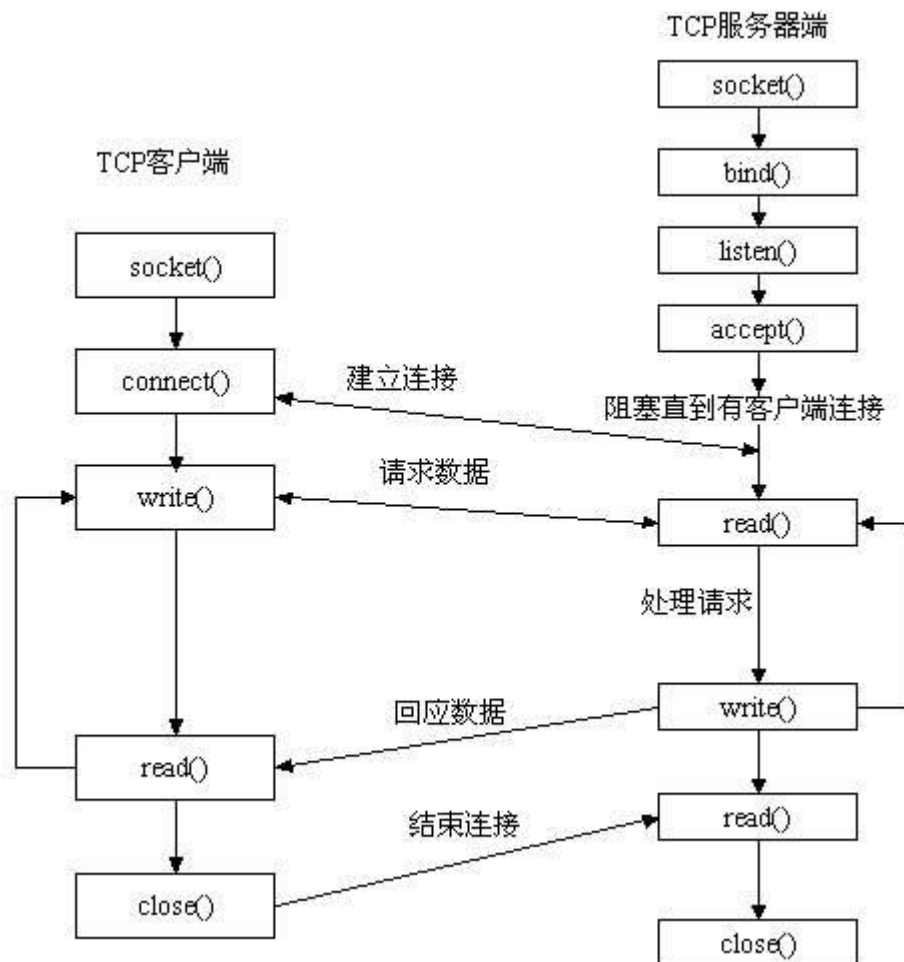
文件指针：**C 语言**中使用文件指针做为 I/O 的句柄。文件指针指向进程用户区中的一个被称为 FILE 结构的数据结构。FILE 结构包括一个缓冲区和一个文件描述符。而文件描述符是文件描述符表的一个索引，因此从某种意义上说文件指针就是句柄的句柄（在 Windows 系统上，文件描述符被称作文件句柄）。

详细内容请看 [linux 文件系统](http://blog.csdn.net/hguisu/article/details/6122513#t7)：<http://blog.csdn.net/hguisu/article/details/6122513#t7>

4. 基本的 SOCKET 接口函数

在生活中，A 要电话给 B，A 拨号，B 听到电话铃声后提起电话，这时 A 和 B 就建立起了连接，A 和 B 就可以讲话了。等交流结束，挂断电话结束此次交谈。打电话很简单解释了这工作原理：

“open—write/read—close”模式。



服务器端先初始化 Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用 accept 阻塞，等待客户端连接。在这时如果有个客户端初始化一个 Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

这些接口的实现都是内核来完成。具体如何实现，可以看看 **linux** 的内核

4.1、socket()函数

int **socket**(int protfamily, int type, int protocol);//返回 sockfd

sockfd 是描述符。

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。

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：

- **protfamily**: 即协议域, 又称为协议族 (family)。常用的协议族有, **AF_INET(IPV4)**、**AF_INET6(IPV6)**、**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?)。
- **protocol**: 故名思意, 就是指定协议。常用的协议有, **IPPROTO_TCP**、**IPPROTO_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!)。

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。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。

4.2、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。

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`。

4.3、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。

4.4、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); //返回连接 connect_fd
```

参数 `sockfd`

参数 `sockfd` 就是上面解释中的监听套接字，这个套接字用来监听一个端口，当有一个客户与服务器连接时，它使用这个一个端口号，而此时这个端口号正与这个套接字关联。当然客户不知道套接字这些细节，它只知道一个地址和一个端口号。

参数 `addr`

这是一个结果参数，它用来接受一个返回值，这返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为 `NULL`。

参数 `len`

如同大家所认为的，它也是结果的参数，用来接受上述 `addr` 的结构的大小的，它指明 `addr` 结构所占有的字节个数。同样的，它也可以被设置为 `NULL`。

如果 `accept` 成功返回，则服务器与客户已经正确建立连接了，此时服务器通过 `accept` 返回的套接字来完成与客户的通信。

注意：

`accept` 默认会阻塞进程，直到有一个客户连接建立后返回，它返回的是一个新可用的套接字，这个套接字是连接套接字。

此时我们需要区分两种套接字，

监听套接字：监听套接字正如 `accept` 的参数 `sockfd`，它是监听套接字，在调用 `listen` 函数之后，是服务器开始调用 `socket()` 函数生成的，称为**监听 socket 描述字**(监听套接字)

连接套接字：一个套接字会从主动连接的套接字变身为一个监听套接字；而 `accept` 函数返回的是**已连接 socket 描述字**(一个连接套接字)，它代表着一个网络已经存在的点点连接。

一个服务器通常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。

自然要问的是：为什么要有两种套接字？原因很简单，如果使用一个描述字的话，那么它的功能太多，使得使用很不直观，同时在内核确实产生了一个这样的新的描述字。

连接套接字 `socketfd_new` 并没有占用新的端口与客户端通信，依然使用的是与监听套接字 `socketfd` 一样的端口号

4.5、`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，即实现了网络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 **ECONNRESET** 表示网络连接出了问题。

write 函数将 **buf** 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两种可能。**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**。

4.6、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。

5. Socket 中 TCP 的建立（三次握手）

TCP 协议通过三个报文段完成连接的建立，这个过程称为三次握手(three-way handshake)，过程如下图所示。

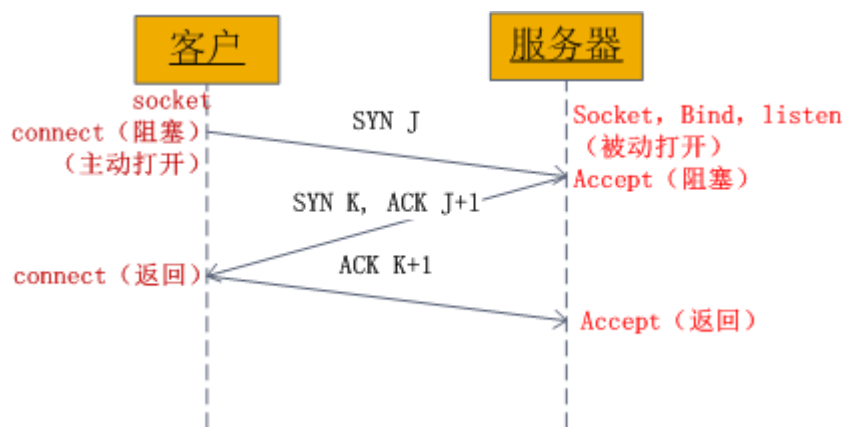
第一次握手：建立连接时，客户端发送 syn 包($\text{syn}=j$)到服务器，并进入 SYN_SEND 状态，等待服务器确认；SYN：同步序列编号(Synchronize Sequence Numbers)。

第二次握手：服务器收到 syn 包，必须确认客户的 SYN($\text{ack}=j+1$)，同时自己也发送一个 SYN 包($\text{syn}=k$)，即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK($\text{ack}=k+1$)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

一个完整的三次握手也就是：请求---应答---再次确认。

对应的函数接口：



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。

我们可以通过网络抓包的查看具体的流程：

比如我们服务器开启 9502 的端口。使用 tcpdump 来抓包：

tcpdump -iany tcp port 9502

然后我们使用 telnet 127.0.0.1 9502 开连接.:

telnet 127.0.0.1 9502

```
14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378, win 32792, options [mss 16396,sackOK,TS val 255474104 ecr 0,nop,wscale 3], length 0 (1)
14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379, win 32768, options [mss 16396,sackOK,TS val 255474104 ecr 255474104,nop,wscale 3], length 0 (2)
14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.], ack 1, win 4099, options [nop,nop,TS val 255474104 ecr 255474104], length 0 (3)
```

```

14:13:01.415407 IP localhost.39870 > localhost.9502: Flags [P.], seq 1:8, ack 1, win 4099, options
[nop,nop,TS val 255478182 ecr 255474104], length 7
14:13:01.415432 IP localhost.9502 > localhost.39870: Flags [.], ack 8, win 4096, options [nop,nop,TS val
255478182 ecr 255478182], length 0
14:13:01.415747 IP localhost.9502 > localhost.39870: Flags [P.], seq 1:19, ack 8, win 4096, options
[nop,nop,TS val 255478182 ecr 255478182], length 18
14:13:01.415757 IP localhost.39870 > localhost.9502: Flags [.], ack 19, win 4097, options [nop,nop,TS
val 255478182 ecr 255478182], length 0

```

- 14:12:45.104687 时间带有精确到微妙
- localhost.39870 > localhost.9502 表示通信的流向，39870 是客户端，9502 是服务器端
- [S] 表示这是一个 SYN 请求
- [S.] 表示这是一个 SYN+ACK 确认包：
- [.] 表示这是一个 ACK 确认包，(client)SYN->(server)SYN->(client)ACK 就是 3 次握手过程
- [P] 表示这个是一个数据推送，可以从服务器端向客户端推送，也可以从客户端向服务器端推
- [F] 表示这是一个 FIN 包，是关闭连接操作，client/server 都有可能发起
- [R] 表示这是一个 RST 包，与 F 包作用相同，但 RST 表示连接关闭时，仍然有数据未被处理。可以理解为是强制切断连接
- win 4099 是指滑动窗口大小
- length 18 指数据包的大小

我们看到 (1) (2) (3) 三步是建立 tcp:

第一次握手:

```
14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378
```

客户端 IP localhost.39870 (客户端的端口一般是自动分配的) 向服务器 localhost.9502 发送 syn 包(syn=j)到服务器》

syn 包(syn=j) : syn 的 seq= 2927179378 (j=2927179378)

第二次握手:

```
14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379,
```

收到请求并确认: 服务器收到 syn 包, 并必须确认客户的 SYN (ack=j+1), 同时自己也发送一个 SYN 包 (syn=k), 即 SYN+ACK 包:

此时服务器主机自己的 SYN: seq: y= syn seq 1721825043。

ACK 为 $j+1 = (\text{ack}=j+1) = \text{ack } 2927179379$

第三次握手:

```
14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.] , ack 1,
```

客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK($\text{ack}=k+1$)

客户端和服务端进入 ESTABLISHED 状态后，可以进行通信数据交互。此时和 accept 接口没有关系，即使没有 accept，也进行 3 次握手完成。

连接出现连接不上的问题，一般是网路出现问题或者网卡超负荷或者是连接数已经满啦。

紫色背景的部分:

```
IP localhost.39870 > localhost.9502: Flags [P.] , seq 1:8, ack 1, win 4099, options [nop,nop,TS val 255478182 ecr 255474104], length 7
```

客户端向服务器发送长度为 7 个字节的数据，

```
IP localhost.9502 > localhost.39870: Flags [.] , ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 0
```

服务器向客户端确认已经收到数据

```
IP localhost.9502 > localhost.39870: Flags [P.] , seq 1:19, ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 18
```

然后服务器同时向客户端写入数据。

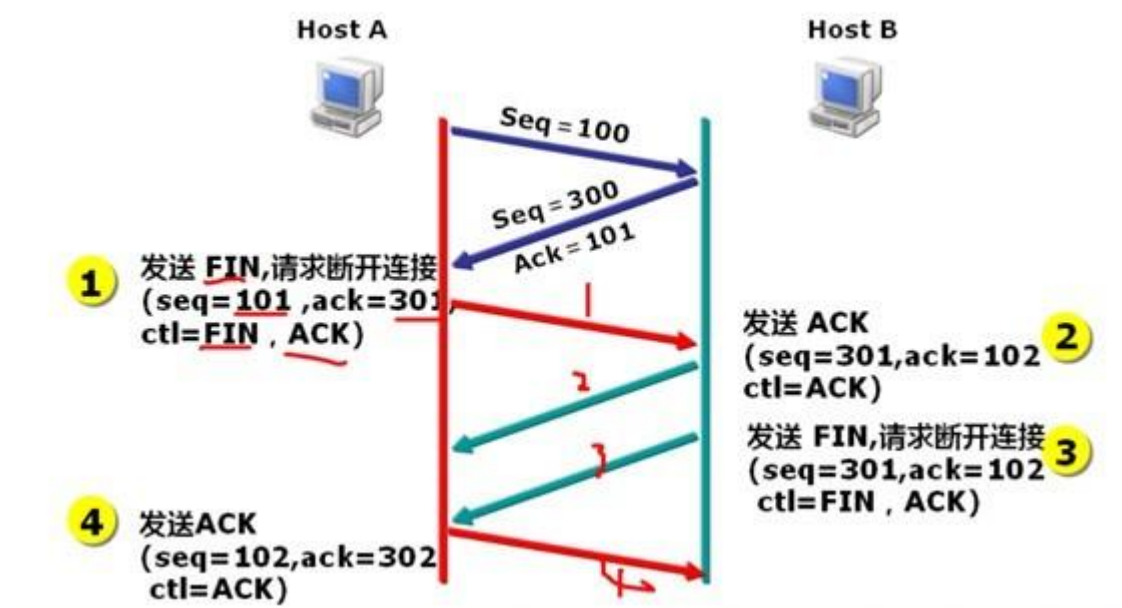
```
IP localhost.39870 > localhost.9502: Flags [.] , ack 19, win 4097, options [nop,nop,TS val 255478182 ecr 255478182], length 0
```

客户端向服务器确认已经收到数据

这个就是 tcp 可靠的连接，每次通信都需要对方来确认。

6. TCP 连接的终止（四次握手释放）

建立一个连接需要三次握手，而终止一个连接要经过四次握手，这是由 TCP 的半关闭(half-close)造成的，如图：



由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

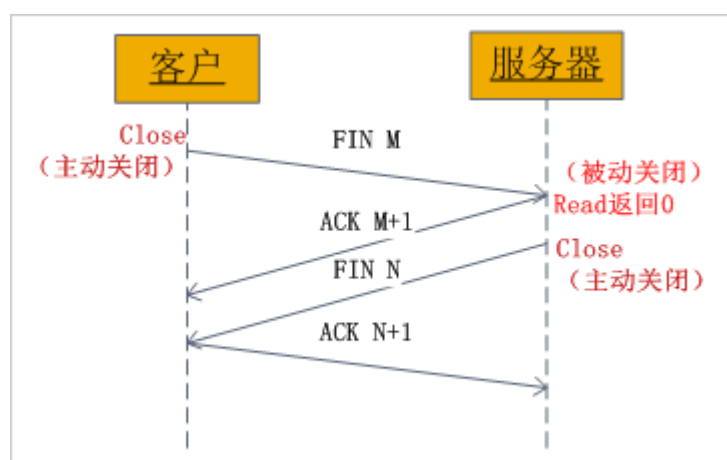
(1) 客户端 A 发送一个 FIN，用来关闭客户 A 到服务器 B 的数据传送（报文段 4）。

(2) 服务器 B 收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1（报文段 5）。和 SYN 一样，一个 FIN 将占用一个序号。

(3) 服务器 B 关闭与客户端 A 的连接，发送一个 FIN 给客户端 A（报文段 6）。

(4) 客户端 A 发回 ACK 报文确认，并将确认序号设置为收到序号加 1（报文段 7）。

对应函数接口如图：



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**。

1. 为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的 **LISTEN** 状态下的 **SOCKET** 当收到 **SYN** 报文的建连请求后，它可以把 **ACK** 和 **SYN** (**ACK** 起应答作用，而 **SYN** 起同步作用) 放在一个报文里来发送。但关闭连接时，当收到对方的 **FIN** 报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭 **SOCKET**，也即你可能还需要发送一些数据给对方之后，再发送 **FIN** 报文给对方来表示你同意现在可以关闭连接了，所以它这里的 **ACK** 报文和 **FIN** 报文多数情况下都是分开发送的。

2. 为什么 **TIME_WAIT** 状态还需要等 **2MSL** 后才能返回到 **CLOSED** 状态？

这是因为虽然双方都同意关闭连接了，而且握手的 4 个报文也都协调和发送完毕，按理可以直接回到 **CLOSED** 状态（就好比从 **SYN_SEND** 状态到 **ESTABLISH** 状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的 **ACK** 报文会一定被对方收到，因此对方处于 **LAST_ACK** 状态下的 **SOCKET** 可能会因为超时未收到 **ACK** 报文，而重发 **FIN** 报文，所以这个 **TIME_WAIT** 状态的作用就是用来重发可能丢失的 **ACK** 报文。

7. Socket 编程实例

服务器端：一直监听本机的 8000 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，并向客户端返回消息。

[[cpp](#)] [view plain](#) [copy](#)

[print?](#) 

```
1. /* File Name: server.c */
2. #include<stdio.h>
3. #include<stdlib.h>
```



```

4. #include<string.h>
5. #include<errno.h>
6. #include<sys/types.h>
7. #include<sys/socket.h>
8. #include<netinet/in.h>
9. #define DEFAULT_PORT 8000
10. #define MAXLINE 4096
11. int main(int argc, char** argv)
12. {
13.     int     socket_fd, connect_fd;
14.     struct  sockaddr_in     servaddr;
15.     char     buff[4096];
16.     int     n;
17.     //初始化 Socket
18.     if( (socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
19.         printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
20.         exit(0);
21.     }
22.     //初始化
23.     memset(&servaddr, 0, sizeof(servaddr));
24.     servaddr.sin_family = AF_INET;
25.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY); //IP 地址设置成 INADDR_ANY, 让系统自动
    获取本机的 IP 地址。
26.     servaddr.sin_port = htons(DEFAULT_PORT); //设置的端口为 DEFAULT_PORT
27.
28.     //将本地地址绑定到所创建的套接字上
29.     if( bind(socket_fd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
30.         printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
31.         exit(0);
32.     }
33.     //开始监听是否有客户端连接
34.     if( listen(socket_fd, 10) == -1){
35.         printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
36.         exit(0);
37.     }
38.     printf("====waiting for client's request====\n");
39.     while(1){
40. //阻塞直到有客户端连接, 不然多浪费 CPU 资源。
41.         if( (connect_fd = accept(socket_fd, (struct sockaddr*)NULL, NULL)) == -1){
42.             printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
43.             continue;
44.         }
45. //接受客户端传过来的数据

```

```

46.     n = recv(connect_fd, buff, MAXLINE, 0);
47. //向客户端发送回应数据
48.     if(!fork()){ /*紫禁城*/
49.         if(send(connect_fd, "Hello,you are connected!\n", 26,0) == -1)
50.             perror("send error");
51.         close(connect_fd);
52.         exit(0);
53.     }
54.     buff[n] = '\0';
55.     printf("recv msg from client: %s\n", buff);
56.     close(connect_fd);
57. }
58. close(socket_fd);
59. }

```

客户端:

[\[cpp\] view plain copy](#)

[print?](#) 

```

1.  /* File Name: client.c */
2.
3.  #include<stdio.h>
4.  #include<stdlib.h>
5.  #include<string.h>
6.  #include<errno.h>
7.  #include<sys/types.h>
8.  #include<sys/socket.h>
9.  #include<netinet/in.h>
10.
11. #define MAXLINE 4096
12.
13.
14. int main(int argc, char** argv)
15. {
16.     int sockfd, n,rec_len;
17.     char recvline[4096], sendline[4096];
18.     char buf[MAXLINE];
19.     struct sockaddr_in servaddr;
20.
21.
22.     if( argc != 2){
23.         printf("usage: ./client <ipaddress>\n");
24.         exit(0);
25.     }

```


```

26.
27.
28.     if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
29.         printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
30.         exit(0);
31.     }
32.
33.
34.     memset(&servaddr, 0, sizeof(servaddr));
35.     servaddr.sin_family = AF_INET;
36.     servaddr.sin_port = htons(8000);
37.     if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
38.         printf("inet_pton error for %s\n",argv[1]);
39.         exit(0);
40.     }
41.
42.
43.     if( connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
44.         printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
45.         exit(0);
46.     }
47.
48.
49.     printf("send msg to server: \n");
50.     fgets(sendline, 4096, stdin);
51.     if( send(sockfd, sendline, strlen(sendline), 0) < 0)
52.     {
53.         printf("send msg error: %s(errno: %d)\n", strerror(errno), errno);
54.         exit(0);
55.     }
56.     if((rec_len = recv(sockfd, buf, MAXLINE,0)) == -1) {
57.         perror("recv error");
58.         exit(1);
59.     }
60.     buf[rec_len] = '\0';
61.     printf("Received : %s ",buf);
62.     close(sockfd);
63.     exit(0);
64. }

```

inet_pton 是 Linux 下 IP 地址转换函数,可以在将 IP 地址在“点分十进制”和“整数”之间转换 , 是 inet_addr 的扩展。

[\[cpp\] view plain copy](#)

print? 

```
1. int inet_pton(int af, const char *src, void *dst); //转换字符串到网络地址:
```

第一个参数 `af` 是地址族，转换后存在 `dst` 中

`af = AF_INET:src` 为指向字符型的地址，即 ASCII 的地址的首地址（`ddd.ddd.ddd.ddd` 格式的），函数将该地址转换为 `in_addr` 的结构体，并复制在 `*dst` 中

`af = AF_INET6:src` 为指向 IPV6 的地址，函数将该地址转换为 `in6_addr` 的结构体，并复制在 `*dst` 中

如果函数出错将返回一个负值，并将 `errno` 设置为 `EAFNOSUPPORT`，如果参数 `af` 指定的地址族和 `src` 格式不对，函数将返回 0。

测试:

编译 `server.c`

```
gcc -o server server.c
```

启动进程:

```
./server
```

显示结果:

```
=====waiting for client's request=====
```

并等待客户端连接。

编译 `client.c`

```
gcc -o client server.c
```

客户端去连接 `server`:

```
./client 127.0.0.1
```

等待输入消息

```
root@ubuntu:/opt/c++/socket# ./client 127.0.0.1
send msg to server:
```

发送一条消息，输入: `c++`

```
root@ubuntu:/opt/c++/socket# ./client 127.0.0.1
send msg to server:
c++
Received : Hello,you are connected!
root@ubuntu:/opt/c++/socket#
```

此时服务器端看到:

```
root@ubuntu:/opt/c++/socket# ./s_server
=====waiting for client's request=====
recv msg from client: c++
```

客户端收到消息:

```
root@ubuntu:/opt/c++/socket# ./client 127.0.0.1
send msg to server:
c++
Received : Hello,you are connected!
root@ubuntu:/opt/c++/socket#
```

其实可以不用 client,可以使用 telnet 来测试:

telnet 127.0.0.1 8000

```
root@ubuntu:/opt/c++/socket# telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
socketsocketsocket
Hello,you are connected!
Connection closed by foreign host.
root@ubuntu:/opt/c++/socket#
```