

# 网络通信

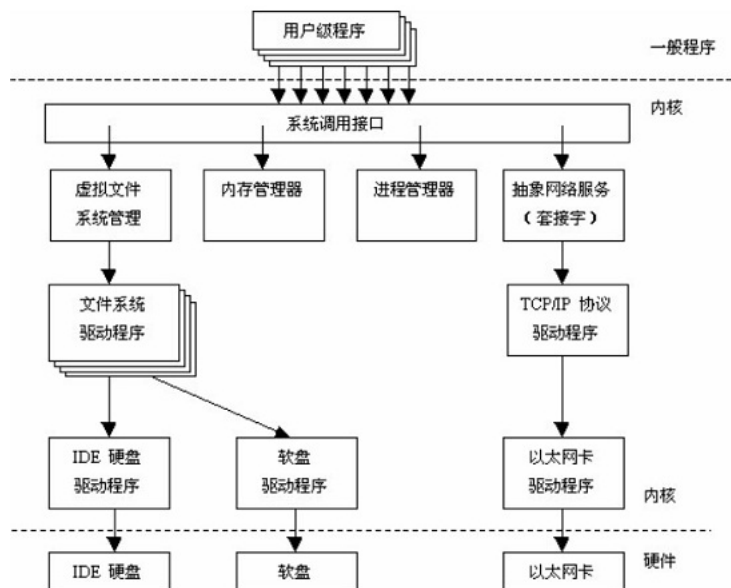
2022 年 3 月 8 日

## 1 Linux 基础知识

### 1.1 linux 内核

内核是操作系统的核心，具有很多最基本功能，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux 内核由如下几部分组成：内存管理、进程管理、设备驱动程序、文件系统和网络管理等。如图所示：



其中系统调用接口：SCI 层提供了某些机制执行从用户空间到内核的函数调用。

## 1.2 Linux 远程登录

Linux 系统中是通过 ssh 服务实现的远程登录功能，默认 ssh 服务端口号为 22。

安装 ssh: yum install ssh

启动 ssh: service sshd start

登录远程服务器: ssh -p 50022 my@127.0.0.1

输入密码:my@127.0.0.1: (-p 后面是端口,my 是服务器用户名,127.0.0.1 是服务器 ip)

回车输入密码即可登录。

## 1.3 Linux 进程

在 Linux 系统中，能够同时运行多个进程，Linux 通过在短的时间间隔内轮流运行这些进程而实现“多任务”。这一短的时间间隔称为“时间片”，让进程轮流运行的方法称为“进程调度”，完成调度的程序称为调度程序。

Linux 中常见的进程间通讯机制有信号、管道、共享内存、信号量和套接字等。

### 1.3.1 相关命令

可以使用 \$ps 命令来查询正在运行的进程，比如 \$ps -e,-o pid,-o comm,-o cmd,

其中-e 表示列出全部进程，-o pid,-o comm,-o cmd 分别表示需要 PID, COMMAND, CMD 信息。

三个信息的意义依次为：PID(process IDentity) 是每一个进程的身份(唯一)，是一个整数，进程也可以根据 PID 来识别其他的进程；COMMAND 是这个进程的简称；CMD 是进程所对应的程序以及运行时所带的参数。

### 1.3.2 Linux 进程的产生

当计算机开机的时候，内核(kernel) 只建立了一个 init 进程。Linux 内核并不提供直接建立新进程的系统调用。其他所有进程都是 init 进程通过

fork 机制建立的。

fork 表示：新的进程要通过老的进程复制自身得到，它是一个系统调用。进程存活于内存中。每个进程都在内存中分配有属于自己的一片空间 (address space)。当进程 fork 的时候，Linux 在内存中开辟出一片新的内存空间给新的进程，并将老的进程空间中的内容复制到新的空间中，此后两个进程同时运行。

老进程成为新进程的父进程 (parent process)，而相应的，新进程就是老的进程的子进程 (child process)。一个进程除了有一个 PID 之外，还会有一个 PPID (parent PID) 来存储的父进程 PID。如果我们循着 PPID 不断向上追溯的话，总会发现其源头是 init 进程。所以说，所有的进程也构成一个以 init 为根的树状结构。

### 1.3.3 Linux 进程的消失

当子进程终结时，它会通知父进程，并清空自己所占据的内存，并在内核里留下自己的退出信息 (exit code，如果顺利运行，为 0；如果有错误或异常状况，为 >0 的整数)。在这个信息里，会解释该进程为什么退出。

父进程在得知子进程终结时，有责任对该子进程使用 wait 系统调用。这个 wait 函数能从内核中取出子进程的退出信息，并清空该信息在内核中所占据的空间。

但是，如果父进程早于子进程终结，子进程就会成为一个孤儿 (orphan) 进程。孤儿进程会被过继给 init 进程，init 进程也就成了该进程的父进程。init 进程负责该子进程终结时调用 wait 函数。

注意：在 Linux 中，线程只是一种特殊的进程。多个线程之间可以共享内存空间和 IO 接口。所以，进程是 Linux 程序的唯一的实现方式。

## 1.4 Linux 内存管理

为了让有限的物理内存满足应用程序对内存的大需求量，Linux 采用了称为“虚拟内存”的内存管理方式。Linux 将内存划分为容易处理的“内存页”（对于大部分体系结构来说都是 4KB）。Linux 包括了管理可用内存的方式，以及物理和虚拟映射所使用的硬件机制。

## 1.5 Linux 文件系统

Linux 系统能支持多种目前流行的文件系统，如 EXT2、EXT3、FAT、FAT32、VFAT 和 ISO9660。

Linux 下面的文件类型主要有：

1) 普通文件：C 语言元代码、SHELL 脚本、二进制的可执行文件等。  
分为纯文本和二进制；

2) 目录文件：目录，存储文件的唯一地方；

3) 链接文件：指向同一个文件或目录的文件；

4) 设备文件：与系统外设相关的，通常在/dev 下面。分为块设备和字符设备；

5) 管道 (FIFO) 文件：提供进程之间通信的一种方式；

6) 套接字 (socket) 文件：该文件类型与网络通信有关；

可参考：<https://www.linuxprobe.com/linux-system-structure.html>

## 1.6 Linux 防火墙

### 1.6.1 常用命令

- 启动：systemctl start firewalld
- 重启：systemctl restart firewalld
- 停止：systemctl stop firewalld
- 重新加载：firewall-cmd --reload
- 查看 firewalld 的运行状态：  
firewall-cmd --state
- 取消开放 https 服务，即禁止 https 服务：  
firewall-cmd --zone=drop --remove-service=https
- 开放 22 端口：  
firewall-cmd --zone=drop --add-port=22/tcp
- 取消开放 22 端口：  
firewall-cmd --zone=drop --remove-port=22/tcp

- 开放两个端口：

```
firewall-cmd --zone=drop --add-port=8080-8081/tcp
```

- 查询 drop 区域开放了哪些端口：

```
firewall-cmd --zone=drop --list-ports
```

- 其他常用命令：

允许 icmp 协议流量，即允许 ping：

```
firewall-cmd --zone=drop --add-protocol=icmp
```

取消允许 icmp 协议的流量，即禁 ping：

```
firewall-cmd --zone=drop --remove-protocol=icmp
```

查询 drop 区域开放了哪些协议：

```
firewall-cmd --zone=drop --list-protocols
```

将原本访问本机 888 端口的流量转发到本机 22 端口：

```
firewall-cmd --zone=drop --add-forward-port =port=888:proto=tcp:toport=22
```

## 2 socket 基础

### 2.1 socket 通信的概念

套接字，运行在计算机中的两个程序通过 socket 建立起一个通道，数据在通道中传输。套接字可以看成是两个网络应用程序进行通信时，各自通信连接中的端点，这是一个逻辑上的概念。它是网络环境中进程间通信的 API(应用程序编程接口)（是 IP 地址和端口结合）。

socket 把复杂的 TCP/IP 协议族隐藏了起来，只要用好 socket 相关的函数，就可以完成网络通信。

### 2.2 socket 的分类

socket 提供了流（stream）和数据报（datagram）两种通信机制，即流 socket 和数据报 socket。

流 socket 基于 TCP 协议，是一个有序、可靠、双向字节流的通道，传输数据不会丢失、不会重复、顺序也不会错乱。类似于打电话。

数据报 socket 基于 UDP 协议, 不需要建立和维持连接, 可能会丢失或错乱。UDP 不是一个可靠的协议, 对数据的长度有限制, 但是它的速度比较高。类似于短信。

在实际开发中, 数据报 socket 的应用场景极少。

## 2.3 socket 通信的过程

### 2.3.1 服务端

- 创建服务端的 socket。socket()
- 把服务端指定的用于通信的 ip 地址和端口绑定到 socket 上。bind()
- 把 socket 设置为监听模式, 以监听用户请求。listen()
- 接受客户端的连接。accept()
- 与客户端通信, 接收客户端发过来的报文后, 回复处理结果。send()/recv()
- 不断的重复上一步步, 直到客户端断开连接。
- 关闭 socket, 释放资源。close()

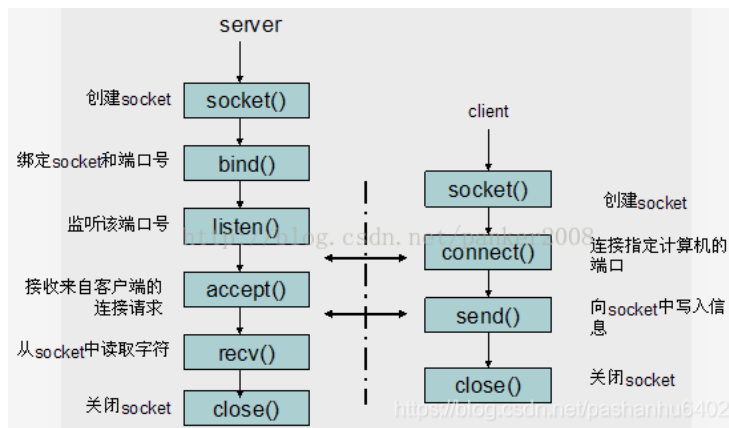
### 2.3.2 客户端

- 创建客户端的 socket。socket()
- 向服务器发起连接请求。connect()
- 与服务端通信, 发送一个报文后等待回复, 然后再发下一个报文。send()/recv()
- 不断的重复上一步, 直到全部的数据被发送完。
- 关闭 socket, 释放资源。close()

## 2.4 Socket

在 Linux 系统中, 一切输入输出设备皆文件。而 socket 本质上可以视为一种特殊的文件, 即通信的实现, 因此 socket 的通信过程也可以理解为通过“打开 open -> 读写 write/read -> 关闭 close”模式的操作过程。

注意:



a. 客户端有一个 socket，是用于发送报文的 socket；而服务端有两个 socket，一个是用于监听的 socket，还有一个就是客户端连接成功后，由 accept 函数创建的用于与客户端收发报文的 socket。

b. socket 是系统资源，操作系统打开的 socket 数量是有限的，在程序退出之前必须关闭已打开的 socket，就像关闭文件指针一样，就像 delete 已分配的内存一样，极其重要。

c. 关闭 socket 的代码不能只在 main 函数的最后，那是程序运行的理想状态，还应该在 main 函数的每个 return 之前关闭。

d. socket 类型的 servaddr 的地址，(struct sockaddr \*)&servaddr。

## 2.5 客户端服务端主要参数及结构体

### 2.5.1 socket() 返回的值

返回值称为 socket 描述符 (socket descriptor)，其本质是一个文件描述符，是一个整数。把它作为参数，通过它来进行一些读写操作。

0 表示标准输入，1 表示标准输出，2 表示标准错误。

注意：默认创建的 socket 是主动连接的。

### 2.5.2 有关定义

- 大端模式、小端模式：“大端”和“小端”表示多字节值的哪一端存储在该值的起始地址处；小端存储在起始地址处，即是小端字节序；大端存储在起始地址处，即是大端字节序。

- a. 大端字节序 (Big Endian): 最高有效位存于最低内存地址处, 最低有效位存于最高内存处;
  - b. 小端字节序 (Little Endian): 最高有效位存于最高内存地址, 最低有效位存于最低内存处。
- 网络字节序 NBO: 网络字节序是大端字节序。在进行网络传输的时候, 发送端发送的第一个字节是高位字节。
  - 主机字节序 HBO: 不同的机器 HBO 不相同, 与 CPU 的设计有关, 数据的顺序是由 CPU 决定的, 而与操作系统无关。不同体系结构的机器之间不能直接通信, 所以要转换成一种约定的顺序, 也就是网络字节顺序。存在专门的函数来进行二者的转换。

### 2.5.3 struct hostent\* h

域名和网络地址结构体:

```
struct hostent
{
    char *h_name; //主机名, 即官方域名
    char **h_aliases; //主机所有别名构成的字符串数组, 同一 IP 可
    绑定多个域名
    int h_addrtype; //主机 IP 地址的类型, 例如 IPV4 (AF_INET)
    还是 IPV6 【需转化】
    int h_length; //主机 IP 地址长度, IPV4 地址为 4, IPV6 地址则
    为 16。【常用】
    char **h_addr_list; //主机的 ip 地址, 以网络字节序存储。若要
    打印出这个 IP, 需要调用 inet_ntoa()。【常用】
};
```

- 注意: a. 通常用于客户端已知对方 ip 情况下获得对方网络其他信息。  
 b. 与下面的结构体一样, 定义时指的是哪个端, 其内数据就指哪个端。

### 操作函数

- `*gethostbyname(const char *name);`  
 用于客户端指定服务端的 ip 地址, 详见后文函数介绍, 注意参数为 char 类型的常量。成功执行该函数后结构体内各变量均可获得。



- `inet_ntop(h->h_addrtype);`

将网络字节序的二进制转换为文本字符串的格式。

#### 2.5.4 struct sockaddr\_in servaddr

表示地址信息的数据结构（体），存放了目标地址和端口。与结构体 `sockaddr` 把目标地址和端口信息混在了一起不同，其把 `port` 和 `addr` 分开储存在两个变量中。

```
struct sockaddr_in
{
    sa_family_t    sin_family; //协议族，在 socket 编程中只能是
    AF_INET。
    unit16_t    sin_port; //16 位的 TCP/UDP 端口号。
    struct in_addr    sin_addr; //32 位 IP 地址
    char    sin_zero[8]; //不使用。
}
```

其中：

```
struct in_addr
{
    in_addr_t s_addr; //32 位 IP v4 地址。
    (可表示为：servaddr.sin_addr.s_addr)
}
```

#### 操作函数

- `inet_addr()`: 将网络地址字符串 (IP 字符串) 转为网络二进制数字 (网络字节序)，返回的 IP 地址是网络字节序的。  
(`const char`—> `in_addr_t`)(ascii to network)
- `inet_ntoa()`: 将网络二进制数字 (网络字节序) 转为网络地址字符串 (IP 字符串)。  
(`in_addr_t`—> `const char`)(network to ascii)
- `inet_aton()`: 将网络地址转为网络字节序，与 `inet_addr` 的区别是，结果不是作为返回值，而是保存形参 `inp` 所指的 `in_addr_t` 结构中。  
函数原型: `int inet_aton(const char* cp, struct in_addr_t *inp)`

(const char--> in\_addr\_t)(ascii to network)

- htons() 作用是将端口号由主机字节序转换为网络字节序的整数值。  
(int to in\_addr\_t)(host to net)  
(不一定是 in\_addr\_t, 满足要求的字符即可, 例如 unit16\_t)
- htonl() 作用和 htons() 一样, 不过它针对的是 32 位的 (long), 而  
htons() 针对的是 16 位的 (short)。  
(host to net)
- 与 htonl() 和 htons() 作用相反的两个函数是: ntohl() 和 ntohs()。  
(net to host)
- 应用: 服务端绑定 (自身的)IP 地址:  

```
servaddr.sin_addr.s_addr = inet_addr("192.168.149.129"); // 指定 ip 地址
```

```
servaddr.sin_addr.s_addr = htonl(INADDR_ANY); // 本主机的任意 ip 地址。在实际开发中, 采用任意 ip 地址的方式比较多。
```
- 应用: 服务端绑定通信端口, 也可用于客户端指定服务端的通信端口:  

```
servaddr.sin_port = htons(5000); // 通信端口 5000
```
- 注意 htons() 的参数为 int 型的无符号短整形数, 若用 argv[] 来传递, 需要强制转换类型函数 atoi()。同理 htonl()。
- 注意: 不论是那个端的程序采用, servaddr 指的是哪个端, 端口、IP 地址就是哪个端。

注意: 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。

## 2.6 客户端服务端函数

以下为几种常用的接口函数:

### 2.6.1 socket() 函数

功能：用于创建一个新的 socket。对应于普通文件的打开操作。

返回值：成功则返回一个 socket 描述符，失败返回-1，错误原因存于 errno 中。

使用范围：客户端、服务端。

#### 函数声明

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

- domain：协议域，又称协议族（family）。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。

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。

流式 socket (SOCK\_STREAM) 是一种面向连接的 socket，针对于面向连接的 TCP 服务应用。数据报式 socket (SOCK\_DGRAM) 是一种无连接的 socket，对应于无连接的 UDP 服务应用。

- protocol：指定协议。

常用协议有 IPPROTO\_TCP、IPPROTO\_UDP、IPPROTO\_STCP、IPPROTO\_TIPC 等，分别对应 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。

注意为 0 则与 type 相匹配，与 type 不能随意匹配。

- 正常情况，第一个参数只能填 AF\_INET，第二个参数只能填 SOCK\_STREAM，第三个参数只能填 0。

使用：`int sockfd = socket(AF_INET,SOCK_STREAM,0)`

### 2.6.2 send() 函数

功能：用于把数据通过 socket 发送给对端。

返回值：函数返回已发送的字符数。出错时返回-1，错误信息 `errno` 被标记。

适用范围：客户端、服务端。

注意：注意，就算是网络断开，或 `socket` 已被对端关闭，`send` 函数不会立即报错，要过几秒才会报错。如果 `send` 函数返回的错误 ( $\leq 0$ )，表示通信链路已不可用。

### 函数声明

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

`sockfd`：为已建立好连接的对方端的 `socket`。

`buf`：为需要发送的数据的内存地址，可以是 C 语言基本数据类型变量的地址，也可以数组、结构体、字符串，内存中有什么就发送什么。

`len`：需要发送的数据的长度，为 `buf` 中有效数据的长度。

`flags`：填 0，其他数值意义不大。

使用：`iret=send(sockfd,buffer,strlen(buffer),0);`

### 2.6.3 `recv()` 函数

功能：用于接收对方 `socket` 发送过来的数据。

返回值：函数返回已接收的字符数。出错时返回-1，失败时不会设置 `errno` 的值。

适用范围：客户端、服务端。

注意：如果 `socket` 的对端没有发送数据，`recv` 函数就会等待，如果对端发送了数据，函数返回接收到的字符数。出错时返回-1。如果 `socket` 被对端关闭，返回值为 0。如果 `recv` 函数返回的错误 ( $\leq 0$ )，表示通信通道已不可用。

### 函数声明

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

`sockfd`：为已建立好连接的对方端的 `socket`。

`buf`：为用于接收数据的内存地址，可以是 C 语言基本数据类型变量的地址，也可以数组、结构体、字符串，只要是一块内存就行了。

`len`：需要接收数据的长度，不能超过 `buf` 的大小，否则内存溢出。

`flags`：填 0，其他数值意义不大。

使用: `iret=recv(sockfd,buffer,sizeof(buffer),0);`

#### 2.6.4 gethostbyname() 函数

功能: 把 ip 地址或域名转换为 hostent 结构体表达的地址。

返回值: 如果成功, 返回一个 hostent 结构指针, 失败返回 NULL。

适用范围: 客户端。

注意: 只要地址格式没错, 一般不会返回错误。失败时不会设置 errno 的值。

##### 函数声明

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

name: 域名或者主机名, 例如"192.168.1.3"、"www.freecplus.net" 等。

使用: `struct hostent *h = gethostbyname(argv[1])`

#### 2.6.5 附注: memset() 函数

```
void *memset(void *str, int c, size_t n)
```

str - 指向要填充的内存块。

c - 要被设置的值。该值以 int 形式传递, 但是函数在填充内存块时是使用该值的无符号字符形式。默认 0。

n - 大小, sizeof(str)。

#### 2.6.6 附注: memcpy() 函数

```
void *memcpy(void *str1, const void *str2, size_t n)
```

str1 - 指向用于存储复制内容的目标数组, 类型强制转换为 void\* 指针。

str2 - 指向要复制的数据源, 类型强制转换为 void\* 指针。

n - 要被复制的字节数。

#### 2.6.7 connect() 函数

功能: 客户端通过调用 connect 函数来建立客户端 socket 与服务器的连接。

返回值: 成功则返回 0, 失败返回-1, 错误原因存于 errno 中。

适用范围: 客户端。

注意: 如果服务端的地址错了, 或端口错了, 或服务端没有启动, connect 一定会失败。

### 函数声明

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockfd: 客户端的 socket 描述字。

\*addr: 服务端的 socket 地址信息。

addrlen: 服务端 socket 地址的长度 (addr 结构体的大小)。

使用: `connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))`

### 2.6.8 bind() 函数

功能: 服务端把自身用于通信的地址和端口绑定到 socket 上。

返回值: 成功则返回 0, 失败返回-1, 错误原因存于 errno 中。

适用范围: 服务端。

注意: 如果绑定的地址错误, 或端口已被占用, bind 函数一定会报错, 否则一般不会返回错误。

### 函数声明

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockfd: 服务端 socket。

\*addr: 服务端的 socket 地址信息。(一个 const struct sockaddr \* 指针, 指向要绑定给 sockfd 的协议地址。存放了服务端用于通信的地址和端口。)

addrlen: 服务端 socket 地址的长度。

使用: `bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));`

注意调用时参数使用强制类型转换。

### 2.6.9 listen() 函数

功能: 用于把主动连接 socket 变为被动连接的 socket, 使得这个 socket 可以接受其它 socket 的连接请求, 从而成为一个服务端的 socket。当调用 listen 之后, 服务端的 socket 就可以调用 accept 来接受客户端的连接请求。

返回值: 0-成功, -1-失败, 错误原因存于 errno 中。

listen 函数一般不会返回错误。

适用范围：服务端。

### 函数声明

```
int listen(int sockfd, int backlog);
```

sockfd：是服务端已经被 bind 过的 socket。

backlog：这个参数涉及到一些网络的细节，比较麻烦，填 5、10 都行，一般不超过 30。

使用：`listen(listenfd,5);`

### 2.6.10 accept() 函数

功能：用于服务端接受客户端的连接。

返回值：成功则返回 0，失败返回-1，错误原因存于 errno 中。

适用范围：服务端。

注意：

a. accept 在等待的过程中，如果被中断或其它的原因，函数返回-1，表示失败，如果失败，可以重新 accept。

b. accept 函数等待客户端的连接，如果没有客户端连上来，它就一直等待，这种方式称之为阻塞。

c. accept 等待到客户端的连接后，**创建一个新的 socket，函数返回值就是这个新的 socket，服务端使用这个新的 socket 和客户端进行报文的收发。**

### 函数声明

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

sockfd：是服务端已经被 listen 过的 socket。

addr：用于存放客户端的地址信息，用 sockaddr 结构体表达，如果需要客户端的地址，可以填 0。也可以先定义一个空的客户端地址，然后填进去。

addrlen：用于存放 addr 参数的长度，如果 addr 为 0，addrlen 也填 0。

使用：`clientfd=accept(listenfd,(struct sockaddr *)&clientaddr,(socklen_t*)&socklen);`

## 2.7 注意事项

- 每一个端，具有一个 socket 指示符、一个地址信息。

- 凡是函数参数含有地址信息的，均需要强制格式化：

(struct sockaddr \*)& + 地址名

- 

### 3

- 

- 

-