第12章 Linux网络编程——基于套接字的网络编程

sizeof编译时运行

套接字网络编程

TCP/IP网络体系

TCP/IP先于OSI模型，不完全符合OSI标准

TCP/IP四层模型：

- 应用层
- 传输层
- 网络层
- 接口层

OSI七层模型：

- 应用层
- 表示层
- 会话层
- 传输层
- 网络层
- 数据链路层
- 物理层

重要协议：
传输层协议：TCP、UDP
网络层：IP、ICMP、IGMP（ICMP和IGMP是辅助IP协议的）
数据链路层：ARP、RARP

socket概念

socket定义

1. 传输层和网络层提供给应用层的标准化编程接口（或称为编程接口）
2. 在因特网上进程间进行数据传输的一个端点，应用程序间进行数据传输通过socket进行

socket类型

- 流式套接字：TCP
- 数据报套接字：UDP
- 原始套接字

TCP是一个**面向连接**的协议，这意味着在**通信之前**，发送方和接收方必须首先**建立连接。**这种连接是通过三次握手（three-way handshake）来建立的，确保双方都已准备好进行通信。

UDP是一个**面向无连接**的协议，这意味着在**通信之前**，发送方和接收方**不需要建立连接。**UDP只是简单地将数据包发送到网络中，而不关心对方是否准备好接收或数据包是否到达。

TCP和UDP的主要区别在于是否面向连接以及是否提供可靠的数据传输服务。TCP适用于需要确保数据完整性和顺序性的应用，而UDP则适用于对实时性要求较高但对数据完整性要求不高的应用。

socket标识

五元组

<sIP, sPort, dIP, dPort, protocol>

- sIP：源IP地址（本地IP地址）
- sPort：源端口（本地端口号），通常临时分配（1024-5000）
- dIP：目的IP地址（远程IP地址）
- dPort：目的端口（远程端口号），通常使用保留端口号（1-1023）
- protocol：协议

端口号

端口号用来识别同一台计算机中进行通信的不同应用程序。因此，它也被称为程序地址。
端口号由其使用的传输层协议决定。因此，不同的传输协议可以使用相同的端口号。
例如，TCP与UDP使用同一个端口号，但使用目的各不相同。这是因为端口号上的处理是根据每个传输协议的不同而进行的。

在实际进行通信时，要事先确定端口号。确定端口号的方法分为两种：

端口号如何确定

- 标准既定的端口号
  这种方法也叫静态方法。它是指每个应用程序都有其指定的端口号。但并不是说可以随意使用任何一个端口号。每个端口号都有其对应的使用目的（当然，这也不是说“绝对地只能有这样一个目的”。在更高级的网络应用中有时也会别作他用。）。
  例如，HTTP、TELNET、FTP等广为使用的应用协议中所使用的端口号就是固定的。这些端口号也被称之为**知名端口号**（**Well-Known Port Numbe**r）。知名端口号一般由**0到1023的数字分配而成。**
  应用程序应该避免使用知名端口号进行既定目的之外的通信，以免产生冲突。
- 时序（动态）分配法
  此时，服务端有必要确定监听端口号，但是接受服务的客户端没必要确定端口号。

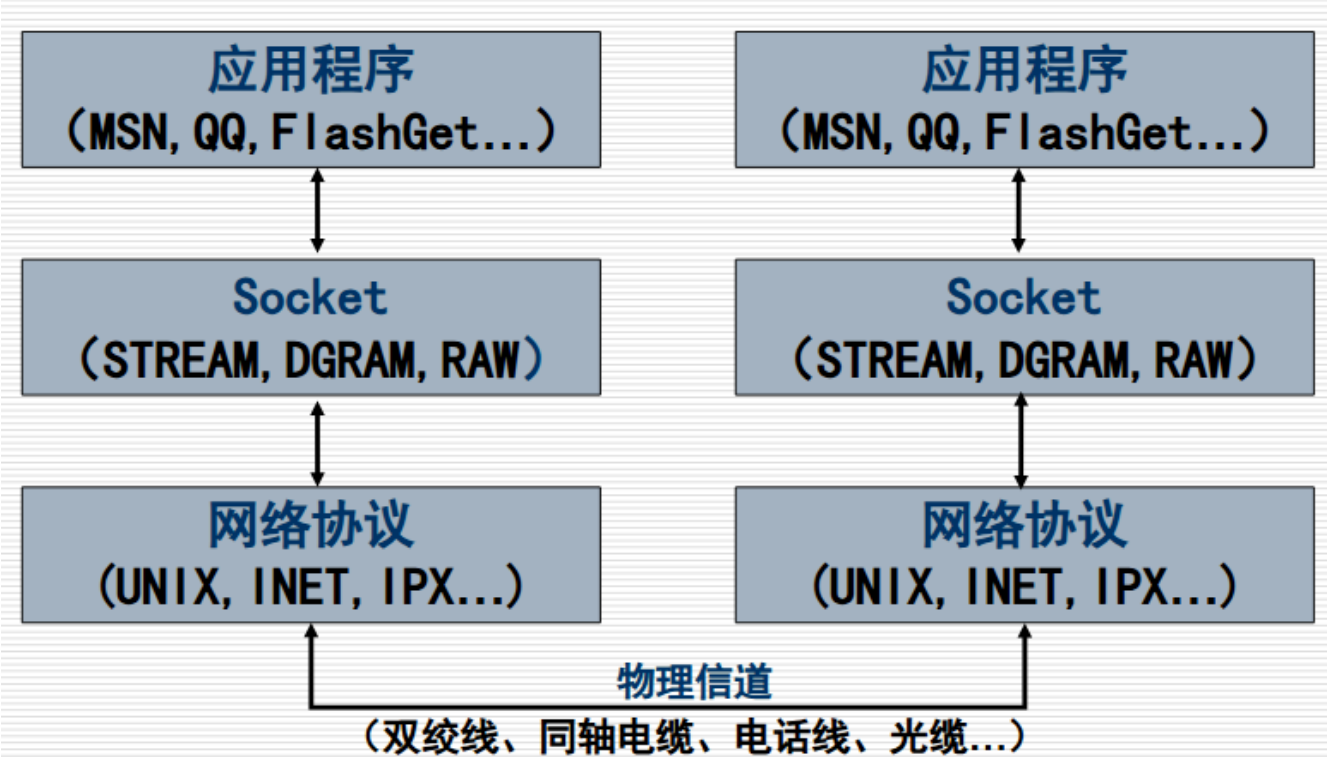
在这种方法下，客户端应用程序可以完全不用自己设置端口号，而全权交给操作系统进行分配。操作系统可以为每个应用程序分配互不冲突的端口号。例如，每需要一个新的端口号时，就在之前分配号码的基础上加1。这样，操作系统就可以动态地管理端口号了。

根据这种动态分配端口号的机制，即使是同一个客户端程序发起的多个TCP连接，识别这些通信连接的5部分数字也不会全部相同。

动态分配的端口号取值范围在49152到65535之间（在较老的系统中有时会依次使用1024以上空闲的端口）

socket基本原理

socket是应用程序间进行数据传输的端结点，为应用程序提供了访问底层网络协议的接口



socket编程

socket结构体

套接字地址

套接字接口允许指定任意类型的地址：

- IPv4
- IPv6
- 非TCP/IP协议族地址

Linux套接字地址结构

Linux定义了一种通用的地址结构

```
#include <sys/socket.h>
struct sockaddr {
    sa_family_t sa_family; /* 地址族，例如AF_INET、AF_INET6等 */
    char sa_data[14];      /* 具体的地址值，通常为112位(14 bytes) */
};
```

- `sa_family`：用于指定地址家族，通常为`AF_INET`（表示IPv4地址）或`AF_INET6`（表示IPv6地址）。
- `sa_data`：是一个字符数组，用于存储具体的地址信息。但由于`sa_data`将目标地址和端口信息混在一起，使用上不够直观，因此在实际编程中，通常会使用`sockaddr`的派生结构体，如`sockaddr_in`（用于IPv4）或`sockaddr_in6`（用于IPv6）。

TCP/IP套接字地址结构

```
#include  <netinet/in.h>
#include <sys/socket.h>
struct sockaddr_in {
    short int sin_family;   /* 地址家族，通常为AF_INET */

    unsigned short int sin_port; /* 端口号，网络字节序 */
```

```
    unsigned short int sin_port; /* 端口号，网络字节序 */
    struct in_addr sin_addr;  /* IP地址 */
    unsigned char sin_zero[8]; /* 填充字段，使得sockaddr_in与sockaddr大小相同 */
};
```

- `sin_family`：地址家族，通常为`AF_INET`。
- `sin_port`：端口号，以网络字节序存储。
- `sin_addr`：一个`in_addr`结构体，用于存储IP地址。

```
struct in_addr {
    u_long s_addr;
};
```

- `sin_zero`：用于填充的字段，确保`sockaddr_in`与`sockaddr`结构体的大小相同。

验证：
`sockaddr`字节数 = 2+14=16
`sockaddr_in`字节数 = 2+2+4+8=16

由于`sockaddr`的`sa_data`字段将地址和端口信息混在一起，使用上不够直观，因此在实际编程中，更推荐使用`sockaddr_in`。
在使用`sockaddr`或`sockaddr_in`时，需要注意地址和端口号的字节序问题。通常，网络编程中使用的地址和端口号都是以**网络字节序（大端字节序）**存储的，而主机上的字节序可能是小端或大端，因此在进行网络编程时，需要进行相应的字节序转换。

常用IP地址转换函数

IPv4

- inet_aton()

`inet_aton`是一个在 Unix-like 系统中常用的函数，用于将**点分十进制的 IPv4 地址**（如 "192.168.1.1"）转换为一个 **32 位的网络字节序整数**，并存储在一个`struct in_addr`结构体中。

函数原型如下：

```
int inet_aton(const char *cp, struct in_addr *inp);
```

参数说明：

- `cp`：一个指向点分十进制 IP 地址字符串的指针。
- `inp`：一个指向`struct in_addr`结构体的指针，该结构体用于存储转换后的 32 位整数。

返回值：

- 如果转换成功，函数返回非零值（通常是 1）。
- 如果转换失败（例如，因为`cp`不是一个有效的 IP 地址），函数返回零。
- inet_ntoa()

`inet_ntoa`是一个用于将 **32 位网络字节序的 IPv4 地址**（存储在`struct in_addr`结构体中）转换为**点分十进制格式的字符串**表示的函数。

函数原型如下：

```
char *inet_ntoa(const struct in_addr in);
```

参数说明：

- `in`：一个 `struct in_addr` 结构体，包含 32 位网络字节序的 IPv4 地址。

返回值：

- 如果成功，函数返回一个指向静态内存的指针，该内存包含点分十进制的 IP 地址字符串。由于这个内存是静态的，所以连续调用 `inet_ntoa` 可能会覆盖之前的结果。
- 如果失败（这通常不会发生，因为 `inet_ntoa` 总是尝试转换输入），函数返回一个空指针（NULL）。

注意：

- `inet_ntoa` 返回的指针指向**静态内存**，这意味着你**不能修改返回的字符串**，也不能在多线程环境中安全地使用它，因为连续调用可能会**覆盖**之前的结果。
- 如果你需要在**多线程环境**中安全地使用这个功能，或者需要**保存或修改返回的字符串**，你应该使用 `inet_ntop` 函数，它提供了更多的灵活性和线程安全性。

IPv4&IPv6

- inet_pton()
  `inet_pton` 是一个用于将**点分十进制（IPv4）** 或**冒号十六进制（IPv6）** 的 IP 地址字符串转换为**网络字节序二进制**表示的函数。这个函数在处理 IPv4 和 IPv6 地址时都很有用，因为它允许你指定地址族（`af` 参数）。

函数原型如下：

```
int inet_pton(int af, const char *src, void *dst);
```

参数说明：

- `af`：地址族，它可以是 `AF_INET`（IPv4）或 `AF_INET6`（IPv6）。
- `src`：指向 IP 地址字符串的指针。
- `dst`：指向一个足够大的缓冲区的指针，用于存储转换后的网络字节序二进制表示。对于 IPv4，这个缓冲区应该至少能够容纳一个 `struct in_addr`（通常是 4 个字节）。对于 IPv6，这个缓冲区应该至少能够容纳一个 `struct in6_addr`（通常是 16 个字节）。

返回值：

- 如果转换成功，函数返回 1。
- 如果输入不是有效的 IP 地址字符串，函数返回 0。
- 如果发生错误（例如，`af` 参数不是 `AF_INET` 或 `AF_INET6`），函数返回 -1，并设置 `errno` 以指示错误。
- inet_ntop()

`inet_ntop` 用于将**网络字节序的二进制 IP 地址（IPv4 或 IPv6）** 转换为**点分十进制（IPv4）** 或**冒号十六进制（IPv6）的字符串**表示。这个函数在处理 IP 地址转换时很有用，因为它提供了对 IPv4 和 IPv6 的统一接口，并且**允许指定目标字符串的缓冲区大小，以防止缓冲区溢出。**

函数原型如下：

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

参数说明：

- `af`：地址族，指定是 IPv4（`AF_INET`）还是 IPv6（`AF_INET6`）。
- `src`：指向包含网络字节序二进制 IP 地址的缓冲区的指针。对于 IPv4，这是一个指向 `struct in_addr` 的指针；对于 IPv6，这是一个指向 `struct in6_addr` 的指针。
- `dst`：指向用于存储点分十进制或冒号十六进制 IP 地址字符串的缓冲区的指针。
- `size`：指定 `dst` 缓冲区的大小。这个值必须足够大，以容纳完整的 IP 地址字符串以及一个终止的空字符（`\0`）。

返回值：

- 如果成功，函数返回一个指向 `dst` 缓冲区的指针，该缓冲区包含转换后的 IP 地址字符串。
- 如果失败（例如，`af` 参数无效，或者 `dst` 缓冲区太小），函数返回 `NULL`，并设置 `errno` 以指示错误。

主机结构体

hostent结构体

`struct hostent` 是 C 语言中用于表示**主机信息**的结构体，通常在网络编程中使用，特别是在处理**域名到 IP 地址的转换(DNS)** 时。
这个结构体在 `<netdb.h>` 头文件中定义，并且与 `gethostbyname()`，`gethostbyaddr()` 等函数一起使用。

```
struct hostent {
    char *h_name;        /* official name of host */
    char **h_aliases;    /* alias list */
    int h_addrtype;      /* host address type */
    int h_length;        /* length of address */
    char **h_addr_list;  /* list of addresses */
};
```

以下是 `struct hostent` 结构体中各个字段的详细说明：

- `h_name`：这是一个指向字符串的指针，表示**主机的官方名称**（通常是**域名**）。
- `h_aliases`：这是一个指向字符指针数组的指针，该数组包含了**主机的别名列表**。这个数组以 NULL 结尾，表示列表的结束。
- `h_addrtype`：这是一个整数，表示**主机地址的类型**。它通常是 `AF_INET`（IPv4）或 `AF_INET6`（IPv6）。
- `h_length`：这是一个整数，表示**地址的长度**（以**字节**为单位）。对于 IPv4 地址，它通常是 4；对于 IPv6 地址，它通常是 16。
- `h_addr_list`：这是一个指向字符指针数组的指针，该数组包含了**主机的网络地址列表**。每个地址都是 `h_length` 字节长。这个数组也以 NULL 结尾，表示列表的结束。注意，`h_addr_list` 中的地址通常是 `in_addr`（对于 IPv4）或 `in6_addr`（对于 IPv6）结构体的指针，但 `struct hostent` 将它们表示为 `char *` 以保持通用性。

> 请注意：`gethostbyname()`，`gethostbyaddr()` 已经过时，在编写新的网络应用程序时，建议使用 `getaddrinfo()` 和 `getnameinfo()` 函数，这些函数提供了更多的灵活性和更好的错误处理机制。

addrinfo结构体

`struct addrinfo` 是一个用于网络地址解析的灵活且可扩展的结构体，它允许开发者指定多个参数来查询主机名或服务名对应的网络地址。以下是这个结构体的详细定义（这里仅展示常见的字段，因为不同系统和库的实现可能有所不同）：

```
#include <netdb.h>
struct addrinfo {
    int    ai_flags;     // AI_PASSIVE, AI_CANONNAME, ...
    int    ai_family;    // AF_UNSPEC, AF_INET, AF_INET6, ...
    int    ai_socktype;  // SOCK_STREAM, SOCK_DGRAM, ...
    int    ai_protocol;  // 0, IPPROTO_TCP, IPPROTO_UDP, ...
    socklen_t  ai_addrlen;   // 长度（以字节为单位）of ai_addr
    struct sockaddr *ai_addr;      // socket address structure
    char   *ai_canonname;  // canonical name of service location
    struct addrinfo *ai_next;      // pointer to next in list };
```

这里是对每个字段的简要说明：

- `ai_flags`：控制函数的行为的标志位。
  - `AI_PASSIVE` 使得返回的套接字地址适用于绑定（bind），通常用于服务器端。
  - `AI_CANONNAME` 使得函数尝试解析主机名并返回其规范名。
- `ai_family`：指定地址族。常见的值有：
  - `AF_UNSPEC`（未指定，让函数选择）
  - `AF_INET`（IPv4）
  - `AF_INET6`（IPv6）
- `ai_socktype`：指定套接字类型。例如，
  - `SOCK_STREAM`（流套接字，如 TCP）
  - `SOCK_DGRAM`（数据报套接字，如 UDP）。
- `ai_protocol`：指定协议。通常为 0，表示让函数选择默认的协议，但也可以指定特定的协议，如 `IPPROTO_TCP` 或 `IPPROTO_UDP`。
- `ai_addrlen`：`ai_addr` 字段指向的 `sockaddr` 结构体的长度（以字节为单位）。
- `ai_addr`：指向 `sockaddr` 结构体的指针，它包含实际的网络地址信息。你可以将这个指针转换为更具体的类型（如 `sockaddr_in` 或 `sockaddr_in6`）以访问 IPv4 或 IPv6 地址。
- `ai_canonname`：如果 `ai_flags` 中设置了 `AI_CANONNAME`，则此字段包含请求的主机的规范名（canonical name）。否则，它被设置为 NULL。
- `ai_next`：指向链表中的下一个 `addrinfo` 结构体的指针。`getaddrinfo()` 函数可能会返回一个 `addrinfo` 结构体链表，其中包含了多个可能的地址。你可以遍历这个链表来访问所有的地址。

使用 `getaddrinfo()` 函数时，你需要先创建一个 `addrinfo` 结构体变量（或更常见地，一个指向 `addrinfo` 结构体的指针），然后调用 `getaddrinfo()` 函数并传入这个变量（或指针）作为参数。在查询完成后，你需要使用 `freeaddrinfo()` 函数来释放 `getaddrinfo()` 分配的内存。

相关函数

getaddrinfo()

函数原型：

```
int getaddrinfo(
    const char *node,     // 主机名或服务名（例如 "www.example.com" 或 "8.8.8.8"）
    const char *service,  // 服务名或端口号（例如 "http" 或 "80"）
    const struct addrinfo *hints, // 指向addrinfo结构的指针，包含对地址信息的请求
```

```
    struct addrinfo **res     // 指向addrinfo结构指针的指针，用于存储结果
);
```

功能：

- `getaddrinfo()` 函数用于将主机名（或IP地址）和服务名（或端口号）解析为套接字地址结构。
- 它返回一个 `addrinfo` 结构体链表，每个结构体代表一个可能的地址。
- `hints` 参数允许你指定你希望得到的地址类型（IPv4、IPv6）、套接字类型（流、数据报）等。

> 通常服务器端在调用 `getaddrinfo()` 之前， `hint` 的 `ai_flags` 设置为 `AI_PASSIVE` ，用于 `bind()` 函数（用于端口和地址的绑定，后面会讲到）主机名 `node` 通常会设置为NULL。
> 客户端调用 `getaddrinfo()` 时， `ai_flags` 一般不设置 `AI_PASSIVE` ，但是主机名 `node` 和服务名 `service` （端口）则应该不为空。

`getnameinfo()`

函数原型：

```
int getnameinfo(
    const struct sockaddr *sa,   // 指向套接字地址结构的指针
    socklen_t salen,             // 套接字地址结构的长度
    char *host,                  // 用于存储主机名的缓冲区
    size_t hostlen,              // 主机名缓冲区的大小
    char *serv,                  // 用于存储服务名的缓冲区
    size_t servlen,              // 服务名缓冲区的大小
    int flags                    // 控制函数行为的标志
);
```

功能：

- `getnameinfo()` 函数将套接字地址结构转换为相应的主机名和服务名。
- 这与 `getaddrinfo()` 的功能相反。
- 你可以使用它来查找给定套接字地址的主机名和服务名（如果可能的话）。
- `flags` 参数允许你控制函数的行为，例如是否执行反向查找。

`freeaddrinfo()`

函数原型：

```
void freeaddrinfo(struct addrinfo *res);
```

功能：

- `freeaddrinfo()` 函数用于释放 `getaddrinfo()` 函数返回的 `addrinfo` 结构体链表占用的内存。

`gai_strerror()`

函数原型：

```
const char *gai_strerror(int errcode);
```

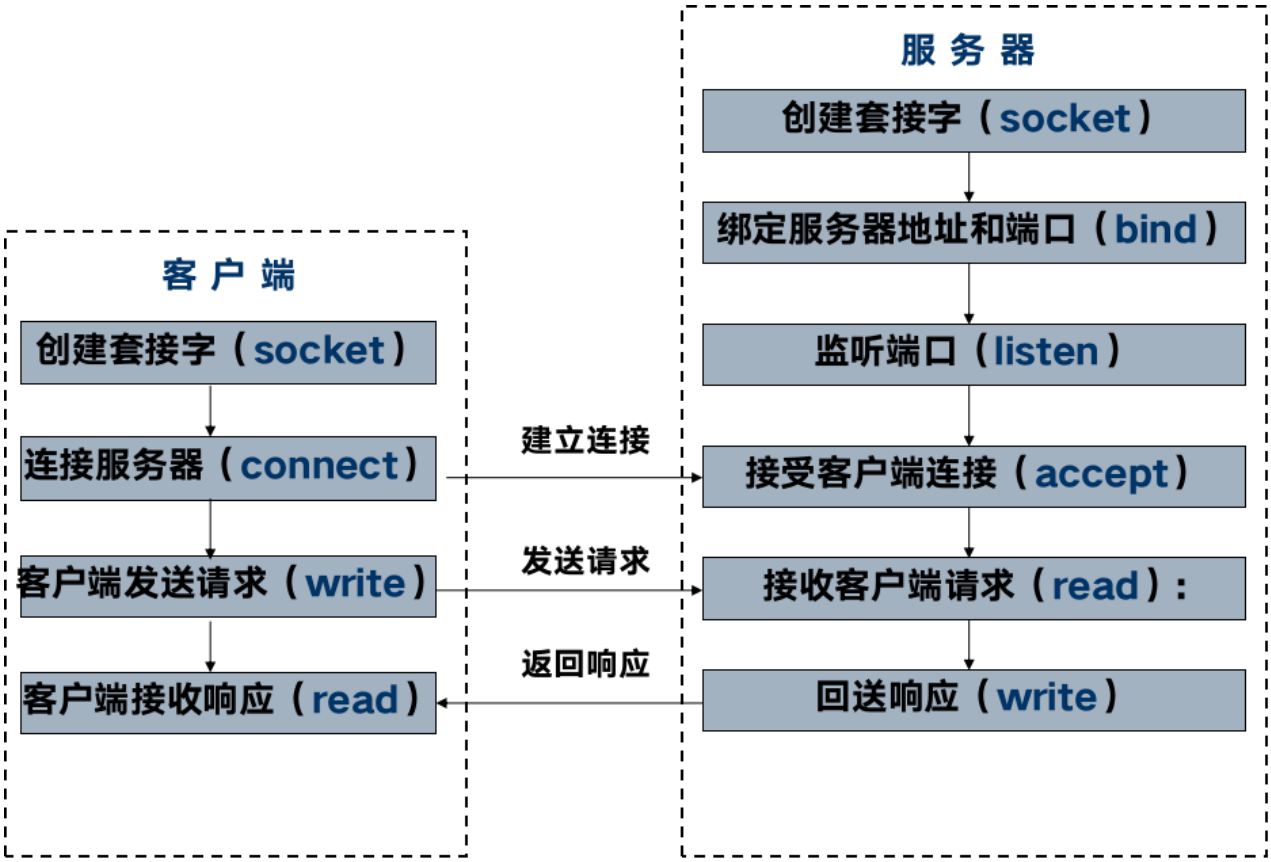
功能：

- `gai_strerror()` 函数将 `getaddrinfo()` 或 `getnameinfo()` 返回的错误码转换为可读的字符串描述。
- 可以用来打印或记录由这些函数返回的错误信息。

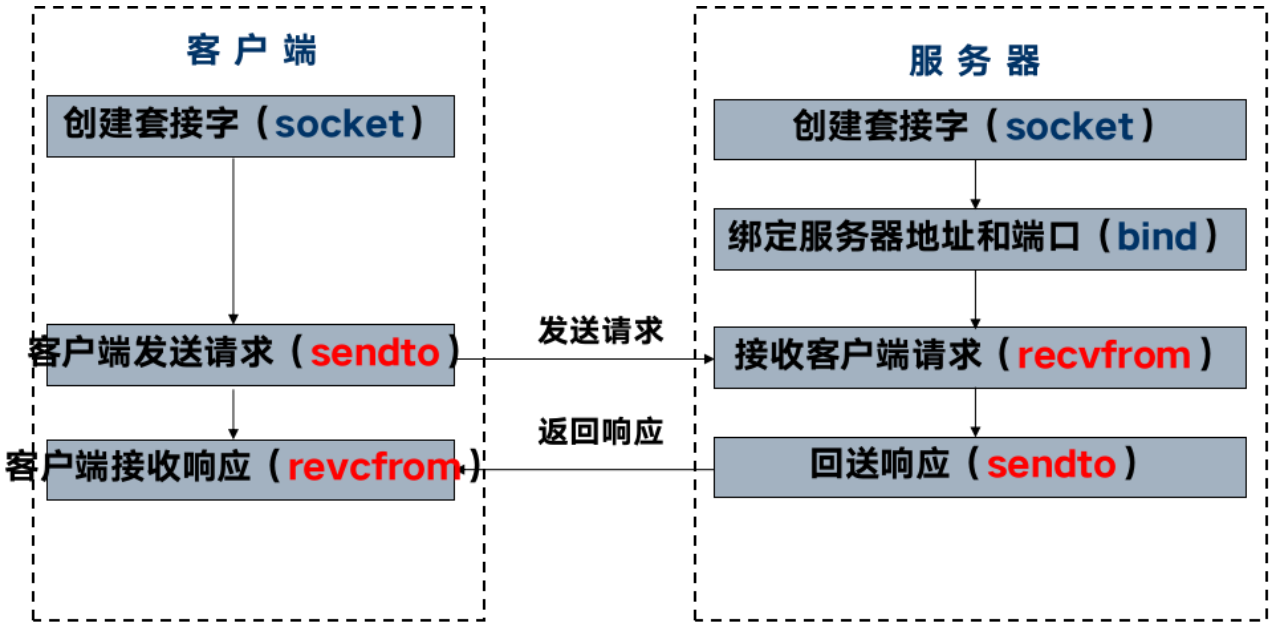
这些函数允许用户编写**不依赖于特定协议或地址族**的代码。

TCP套接字编程典型模型

1. TCP版



2. UDP版



步骤：

- 创建套接字 - socket
- 绑定服务器地址和端口 - bind
- 监听端口 - listen

- 连接服务器 - connect
- 接受客户端连接 - accept
- 接收和发送数据 - recv(read)、send(write)
- 关闭套接字 - close

创建套接字（服务器和客户端）

`socket()` 函数是 Unix/Linux 系统中用于创建新的套接字描述符的函数，允许指定套接字的类型、协议族和协议。

函数原型

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(
    int family,
    int type,
    int protocol
);
```

- `family`：指定协议族。常用的有 `AF_INET`（IPv4）和 `AF_INET6`（IPv6）、AF_UNIX。
- `type`：指定套接字类型。常用的有 `SOCK_STREAM`（流式套接字，TCP）、`SOCK_DGRAM`（数据报套接字，UDP）和 `SOCK_RAW`（原始套接字）。
- `protocol`：指定使用的协议。默认为 0，系统将根据前两个参数选择默认协议。

如果成功，`socket()` 返回一个非负整数，即新的套接字描述符。
如果失败，返回 -1 并设置 全局变量`errno` 以指示错误（strerror函数显示描述字符串）。

绑定服务器地址和端口（服务器）

`bind` 是 Unix/Linux 系统中网络编程的一个重要函数，用于将一个套接字（socket）绑定到一个特定的地址和端口上。这是网络通信中建立连接之前的一个关键步骤。

函数原型

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

- `sockfd`：这是由 `socket` 函数返回的文件描述符，表示一个已创建的套接字。
- `addr`：这是一个指向 `sockaddr` 结构体的指针，它包含了要绑定的地址和端口信息。在实际编程中，这个结构体通常会是一个更具体的类型，如 `sockaddr_in`（用于 IPv4）或 `sockaddr_in6`（用于 IPv6）。
- `addrlen`：这个参数表示 `addr` 指向的结构体的大小。通常，你可以使用 `sizeof` 运算符来获取这个值，例如 `sizeof(struct sockaddr_in)`。

函数的返回值是一个整数，如果绑定成功，则返回 0；如果失败，则返回 -1 并设置全局变量 `errno` 以指示错误。

在调用 `bind` 函数之前，通常已经通过 `socket` 函数创建了一个套接字，并且已经确定了要绑定的地址和端口。`bind` 函数确保没有其他套接字已经绑定了相同的地址和端口，并将这个套接字与指定的地址和端口关联起来。

> 注意：对于 TCP 套接字，在调用 `bind` 之后，还需要调用 `listen` 函数来监听连接请求；对于 UDP 套接字，则可以直接调用 `recvfrom` 或 `sendto` 来发送和接收数据。

```
srvaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- `htonl` 是一个函数，如前所述，它将32位整数从主机字节顺序转换为网络字节顺序。
- `INADDR_ANY` 是一个宏，它的值通常是 `0x00000000`，它告诉套接字绑定到所有可用的本地接口，这样无论哪个网络接口接收到了数据，套接字都能处理。

因此，这行代码的作用是将 `srvaddr.sin_addr.s_addr` 设置为网络字节顺序的 `INADDR_ANY`，这样套接字就可以监听所有网络接口上的数据。这在服务器编程中很常见，因为**服务器**需要能够接收**来自任何网络接口的客户机连接。**

监听端口（服务器）

`listen` 函数在 C 语言的 socket 编程中用于**使服务器端的 socket 进入监听状态**，等待客户端的连接请求。这个函数是在 `socket` 被创建并且已经通过 `bind` 绑定了特定的 IP 地址和端口号之后调用的。

函数原型如下：

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

- `sockfd`：这是由 `socket` 函数返回的 socket 文件描述符。
- `backlog`：这个参数定义了等待连接的队列的最大长度。当一个客户端尝试连接时，如果服务器端的 socket 还没有调用 `accept` 来接受这个连接，那么这个连接请求就会被放在队列中等待。`backlog` 参数定义了队列中可以等待的最大连接数。如果队列已满，客户端可能会收到一个错误，或者连接请求可能会被忽略。

`listen` 函数的返回值是一个整数。
如果成功，它返回 0；
如果失败，它返回 -1，并设置全局变量 `errno` 以指示错误。

执行listen后socket转换成**被动socket**，可以**接受连接**；
只能应用于**面向连接方式**的socket（TCP协议），比如**SOCK_STREAM**；
当一个连接请求到达时，被插入请求队列，服务器用accept()函数从队列中移走并响应请求；

> 注意：请求队列中的连接已经被TCP接受（即三次握手已经完成），但还没有被应用层所接受
> 注意区分：TCP接受一个连接是将其放入这个队列，而应用层接受连接是将其从该队列移出

连接服务器（客户端）

`connect` 函数在 socket 编程中用于建立与服务器端的连接。这个函数通常在客户端程序中使用，它尝试与由 `servaddr` 参数指定的服务器建立连接。

函数原型如下：

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

- `sockfd`：这是由 `socket` 函数返回的 socket 文件描述符。

- `servaddr`：这是一个指向 `sockaddr` 结构的指针，该结构包含了服务器端的 IP 地址和端口号。对于 IPv4 地址，通常会使用 `sockaddr_in` 结构体来填充这个参数。
- `addrlen`：这个参数指定了 `servaddr` 参数指向的 `sockaddr` 结构的大小。这通常是一个常量，如 `sizeof(struct sockaddr_in)`。
- `connect` 函数的返回值是一个整数。如果连接成功，它返回 0；如果连接失败，它返回 -1，并设置全局变量 `errno` 以指示错误。

> 客户机一般不指定自己的端口号，由系统分配一个临时端口；
>
> 对一个socket描述符**不能两次使用**connect函数

接受客户端连接（服务器）

`accept` 函数在socket 编程中用于从**已完成连接队列的头部**取出一个已完成的连接，并**返回一个新的 socket 文件描述符**来**代表这个连接。**
这个新的 socket 文件描述符被用来与客户端进行通信，而原始的 `sockfd`（即**监听 socket**）则**继续用于监听新的连接请求。**

函数原型如下：

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

- `sockfd`：这是由 `socket` 函数返回的，并经过 `bind` 和 `listen` 函数处理过的 socket 文件描述符。这个 socket 必须是被动打开的，并且处于监听状态。
- `clientaddr`：这是一个指向 `sockaddr` 结构的指针，该结构用来保存发起连接的客户端的地址信息。如果对这个信息不感兴趣，可以传递 `NULL`。
- `addrlen`：这是一个指向 `socklen_t` 变量的指针，它指向一个值，该值在调用时初始化为 `clientaddr` 指向的 `sockaddr` 结构的大小。函数返回时，这个值被设置为实际返回的地址的大小。如果 `clientaddr` 是 `NULL`，这个参数也可以设置为 `NULL`。

`accept` 函数的返回值是一个新的 socket 文件描述符，用于与客户端通信。
如果发生错误，返回 -1，并设置全局变量 `errno` 以指示错误。

accept函数返回的新创建的socket描述符是真正可以和客户端通信的socket，
服务器的侦听socket只负责侦听和接受连接，不用于通信；
accept函数在无连接请求时将阻塞进程。

> accept新创建的socket的端口号和原sockfd的端口号相同：
> 四元组区分TCP连接；
> 主监听套接字和新创建的套接字状态不同（listen/established），

- 处于established的套接字不能接收SYN报文段，
- 而处于listen的套接字不能接收数据报文段。

接收数据（客户端和服务器）

`read` 函数用于从文件描述符（`fd`）指向的文件或设备读取数据，在 socket 编程中，`read` 函数用于从**已连接的 socket 中**读取数据。

函数原型如下：

```
ssize_t read(int fd, void *buf, size_t len);
```

参数说明：

- `fd`：文件描述符，一个非负整数，通常是由 `open`、`socket` 等系统调用返回的。
- `buf`：一个指向缓冲区的指针，该缓冲区用于存储从文件中读取的数据。
- `len`：要读取的**最大字节数。**

返回值：

- 成功读取时，返回实际读取的字节数，可能小于请求的 `len` 字节（例如，当文件或套接字中的数据不足时）。
- 如果遇到文件末尾（对于文件），则返回 `0`。
- 如果发生错误，返回 `-1`，并设置全局变量 `errno` 以指示错误类型。

`read` 函数可以用于从套接字读取数据，这在网络编程中非常常见。例如，服务器使用 `read` 来接收客户端发送的数据。

> 注意，`read` 调用可能会阻塞，直到有数据可读或者发生错误。如果你不希望 `read` 阻塞，你可以将 socket 设置为非阻塞模式，但这需要额外的代码来处理 `EWOULDBLOCK` 或 `EAGAIN` 错误。

1. **buf指的是接收或发送进程的应用缓冲区**：
   `buf` 通常是用户空间中的一个缓冲区，用于存储从套接字接收缓冲区读取的数据，或者将要发送到套接字发送缓冲区的数据。在 `read` 或 `write` 系统调用中，这个缓冲区是应用程序提供的。
2. **套接字接收缓冲区和发送缓冲区**：
   每个 TCP 套接字在内核中都有两个缓冲区：接收缓冲区（receive buffer）和发送缓冲区（send buffer）。
   - 接收缓冲区用于暂存从网络上接收但尚未被用户进程读取的数据；
   - 发送缓冲区用于暂存用户进程想要发送但尚未被 TCP 协议发送到网络上的数据。
3. **read并不是从网络读取数据**：
   `read` 系统调用并不直接从网络上读取数据。相反，它从套接字的接收缓冲区中复制数据到用户进程提供的缓冲区中。如果接收缓冲区中没有数据可供读取（即缓冲区为空），`read` 系统调用可能会阻塞，直到有数据到来或者发生超时或错误。
4. **内核TCP协议处理数据的读写**：
   真正从网络读取数据到套接字接收缓冲区，以及将数据从套接字发送缓冲区发送到网络的过程，都是由内核中的 TCP 协议栈来处理的。这个过程对应用程序是透明的，应用程序只需要通过 `read` 和 `write` 系统调用来与套接字缓冲区进行交互。
5. **read/write跨越了用户态和内核态**：
   当应用程序调用 `read` 或 `write` 系统调用时，它实际上是在用户态和内核态之间进行切换。这些系统调用会陷入内核态，执行相应的内核代码来访问套接字缓冲区，并将数据从内核缓冲区复制到用户缓冲区（对于 `read`）或将数据从用户缓冲区复制到内核缓冲区（对于 `write`）。这个过程涉及到内存管理和权限检查等复杂的操作。
6. **本质是读写系统（内核）缓冲区**：
   从更广泛的角度来看，`read` 和 `write` 系统调用的本质是在用户空间和内核空间之间复制数据。对于套接字编程来说，这些数据通常是**从**或**到**套接字缓冲区的。但在其他类型的文件 I/O 中，这些数据可能是**从**或**到**文件系统的缓冲区或其他类型的内核缓冲区。

发送数据

将用户空间缓冲区 `buf` 中的 `len` 字节数据发送到由文件描述符 `fd` 指定的 socket 上。

参数：

- `fd`：文件描述符，一个非负整数，代表要发送数据的 socket。
- `buf`：指向用户空间缓冲区的指针，该缓冲区包含要发送的数据。
- `len`：要发送的字节数。

在 socket 编程中，`write` 函数用于将数据写入到套接字的发送缓冲区中。**成功返回并不直接意味着数据已经被发送到对方主机**，而只是说明**数据已经被成功地复制到本地系统的内核套接字发送缓冲区中**。之后，TCP 协议栈会负责将数据从发送缓冲区中取出并通过网络发送到对方主机。

关于 `write` 函数的返回值和行为：

1. **成功返回**：当 `write` 函数成功执行时，它返回实际写入的字节数。如果发送缓冲区有足够的空间来容纳整个请求的数据（即 `len` 字节），那么 `write` 函数将返回 `len`。

2. **发送缓冲区空间不足**：

   - **阻塞模式**：如果发送缓冲区没有足够的空间来容纳整个请求的数据，并且 socket 被设置为阻塞模式（默认模式），那么 `write` 函数将阻塞调用线程，直到有足够的空间来容纳请求的数据或发生错误为止。
   - **非阻塞模式**：如果 socket 被设置为非阻塞模式，并且发送缓冲区没有足够的空间来容纳整个请求的数据，那么 `write` 函数将立即返回一个较小的值，表示实际写入的字节数。这通常是一个小于 `len` 的值，表示只有一部分数据被写入了发送缓冲区。

3. **错误返回**：

   - **连接被复位**：如果连接在 `write` 函数调用期间被对方重置（例如，对方主机通过 TCP RST 数据包关闭了连接），那么 `write` 函数将返回 -1，并设置全局变量 `errno` 为一个表示错误的值（如 `ECONNRESET`）。
   - **阻塞过程中收到中断信号**：如果在 `write` 函数阻塞等待发送缓冲区空间的过程中，调用线程收到了一个中断信号（如 SIGINT 或 SIGTERM），那么 `write` 函数将返回 -1，并设置 `errno` 为 `EINTR`。这表示调用被中断，应用程序可以选择重新尝试调用 `write` 函数。

请注意，`write` 函数可能会在无法立即发送所有数据时只发送部分数据。因此，在网络编程中可能需要在一个循环中调用 `write`，直到所有数据都被发送完毕。

关闭套接字（客户端和服务器）

关闭由 `sockfd` 指定的 socket，释放与其关联的所有资源。

```
int close (
  int sockfd
);
```

参数说明：

- `sockfd`：文件描述符，对于文件来说，它是文件打开时返回的标识符；对于套接字来说，它是套接字返回的描述符。

返回值：

- 成功关闭时，返回 `0`。
- 如果发生错误，返回 `-1`，并设置全局变量 `errno` 以指示错误类型。

1. **资源释放**：调用 `close` 会关闭 socket 并释放与之关联的所有资源。这包括端口号、文件描述符、接收和发送缓冲区中的任何未处理的数据。因此，在调用 `close` 之后，不应再尝试在该 socket 上进行任何操作。在多进程或多线程应用程序中，每个进程或线程都应该关闭自己使用的套接字，因为一个套接字在所有引用它的描述符都被关闭之前不会真正被释放。
2. **数据传输**：在调用 `close` 之后，发送到该 socket 的任何数据都将被丢弃，并且接收方将收到一个 EOF（文件结束）或连接关闭的通知（对于 TCP）。因此，在关闭 socket 之前，应确保所有需要发送的数据都已被发送并确认。

常用字节处理函数

1. `bzero()`（已经被废弃）

```
void bzero(void *s, int n)      //将参数s指定的内存的前n个字节设置为0

bzero(buf, sizeof(buf));
```

2. `bcopy()`（不建议使用，某些编译器可能已经废弃）

```
void bcopy(const void *src, void *dest, size_t n); //将从`src`指向的内存区域复制`n`个字节的数据到`dest`指向的内存区域。
```

- **内存重叠**：使用`bcopy`函数时，源内存区域和目标内存区域不能重叠。如果它们重叠，可能会导致未定义的行为，甚至程序崩溃。
- **内存大小**：需要确保目标内存区域有足够的空间来存储从源内存区域复制的数据。即，目标内存区域的大小应至少为`n`个字节。
- **数据类型**：由于`bcopy`函数是按字节复制数据的，所以在拷贝结构体等复杂数据类型时，可能需要进行额外的处理，以确保正确复制数据。

3. `bcmp()`（不推荐）

```
int bcmp(const void *s1, const void *s2, size_t n);  //比较两个内存区域的内容是否相同。
```

4. `memset()`（推荐替代`bzero()`）

```
void *memset(void *s, int c, size_t n);   // 将参数s指定的内存区域的前n个字节设置为参数c

memset(buf, 0, sizeof(buf));
```

5. `memcpy()`（推荐替代`bcopy()`）

```
void *memcpy(void *dest, const void *src, size_t n) //功能与bcopy相似，但bcopy允许参数src和dest所指区域有重叠情形
```

6. `memcmp()`（推荐替代`bcmp()`）

```
int memcmp(const void *s1, const void *s2, size_t n)    //比较参数s1和参数s2指定区域的前n个字节内容
```

高级socket函数

`send` 和 `sendto` 这两个函数都用于在已连接的套接字上发送数据，但它们的用途和参数有所不同。

send()函数

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

- `sockfd`：要发送数据的套接字的文件描述符。
- `buf`：一个指向要发送数据的缓冲区的指针。
- `len`：要发送的数据的字节数。
- `flags`：控制发送行为的标志位。通常设置为 0。

返回值：≥0—成功，-1—失败

`send` 函数用于在已连接的套接字上发送数据。它通常用于 **TCP 套接字**，因为在 TCP 套接字上，数据会按照**字节流**的方式被传输。

sendto()函数

```
ssize_t sendto(
    int sockfd,
    const void *buf,
    size_t len,
    int flags,
    const struct sockaddr *dest_addr,
    socklen_t addrlen
);
```

- `sockfd`：要发送数据的套接字的文件描述符。
- `buf`：一个指向要发送数据的缓冲区的指针。
- `len`：要发送的数据的字节数。
- `flags`：控制发送行为的标志位。通常设置为 0。
- `dest_addr`：一个指向目标地址的指针，该地址是一个 `sockaddr` 结构体（或其变种，如 `sockaddr_in` 用于 IPv4）。
- `addrlen`：`dest_addr` 结构体的大小。

返回值：≥0—成功，-1—失败

`sendto` 函数用于在**无连接套接字**（如 **UDP 套接字**）上发送数据。它允许你**指定数据的目的地址**。与 `send` 相比，`sendto` 通常**用于需要明确指定接收者地址的协议**，如 UDP。

因为没有连接，所以目的地址需要自己指定。

recv()函数

同理可见send函数

recvfrom()函数

同理可见sendto函数

阻塞与非阻塞

- 阻塞方式（ block ），就是进程或是线程执行到这些函数时必须**等待**某个事件的发生，假如事件没有发生，进程或线程就被阻塞，函数不能立即返回。

- 非阻塞方式（ non-block ），就是进程或线程执行此函数时**不必非要等待**事件的发生，一旦执行肯定返回，以返回值的不同来反映函数的执行情况，假如事件发生则和阻塞方式相同，若事件没有发生则返回一个代码来告知事件未发生，而进程或线程继续执行，所以效率较高。

阻塞模型

在阻塞模型中，当一个进程调用一个I/O函数（如`recv`、`send`、`accept`等）时，如果条件不满足（例如没有数据可读、缓冲区已满等），进程会被挂起（即阻塞），直到条件满足为止。在此期间，进程不能执行其他操作，只能等待I/O操作完成。

优点：

1. 编程简单：不需要显式地检查条件是否满足，只需调用I/O函数即可。
2. 适用于简单应用：对于小型或不需要高并发的应用，阻塞模型通常足够使用。

缺点：

1. 资源利用率低：当进程被阻塞时，它不能执行其他任务，这可能导致CPU和资源的浪费。
2. 不适用于高并发场景：在高并发环境下，大量进程可能同时被阻塞，导致系统性能下降。

非阻塞模型

在非阻塞模型中，当一个进程调用一个I/O函数时，如果条件不满足，进程不会被挂起，而是立即返回一个错误（如`EWOULDBLOCK`）。这样，进程可以继续执行其他任务，而不必等待I/O操作完成。为了处理I/O事件，进程通常使用某种形式的轮询（polling）或事件通知（event notification）机制。

优点：

1. 资源利用率高：进程在等待I/O操作时可以继续执行其他任务，从而提高资源利用率。
2. 适用于高并发场景：在高并发环境下，非阻塞模型可以更好地处理大量并发连接。

缺点：

1. 编程复杂：需要显式地检查I/O条件是否满足，并可能需要使用轮询或事件通知机制。
2. 可能导致CPU忙等待：如果进程频繁地检查I/O条件而条件始终不满足，可能会导致CPU的忙等待现象。

解决方案

为了克服阻塞模型和非阻塞模型的缺点，现代网络编程中经常采用异步I/O（asynchronous I/O）和事件驱动编程（event-driven programming）技术。这些技术允许进程在I/O操作完成时得到通知，而不需要显式地检查条件或轮询。这可以进一步提高资源利用率和并发性能。

常见的异步I/O和事件驱动编程技术包括：

- **I/O多路复用**（I/O multiplexing）：如`select`、`poll`和`epoll`（Linux特有）等系统调用，允许一个进程监视多个文件描述符的状态变化。
- 信号驱动I/O（signal-driven I/O）：使用`SIGIO`或`SIGURG`等信号来通知进程I/O事件已发生。
- 异步I/O函数（asynchronous I/O functions）：如`aio_read`、`aio_write`等函数，允许进程启动异步I/O操作并在操作完成时得到通知。
- 事件驱动框架（event-driven frameworks）：如libevent、libuv等库，提供了高级的事件驱动编程接口，简化了异步I/O和事件处理的编程工作。

多路复用

select函数是用于处理 I/O 多路复用的重要系统调用。
以下是 `select` 函数的详细参数和功能描述：

select函数

```
#include <sys/select.h>
int select(
    int nfds,
    fd_set *readfds,
    fd_set *writefds,
    fd_set *exceptfds,
    struct timeval *timeout
);
```

- `nfds`：这是一个整数值，指定了**被监听的文件描述符集的最大值加 1。通常设置为**三个集合（`readfds`、`writefds`、`exceptfds`）中最大的文件描述符加 1。
- `readfds`：指向 `fd_set` 结构的指针，该结构中的每一位代表一个文件描述符。如果某一位被设置（即对应文件描述符被选中），则 `select` 会检查该文件描述符**是否可读。**
- `writefds`：指向 `fd_set` 结构的指针，该结构中的每一位代表一个文件描述符。如果某一位被设置（即对应文件描述符被选中），则 `select` 会检查该文件描述符**是否可写。**
- `exceptfds`：指向 `fd_set` 结构的指针，该结构中的每一位代表一个文件描述符。如果某一位被设置（即对应文件描述符被选中），则 `select` 会检查该文件描述符**是否发生异常。**
- `timeout`：指向 `timeval` 结构的指针，该结构指定了 `select` 函数的超时时间。如果设置为 `NULL`，则 `select` 会一直阻塞，直到至少有一个文件描述符就绪。

```
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec; /* microseconds */
};
```

```
`tv_sec`字段表示秒数。
`tv_usec`字段表示微秒数。
```

1. timeout=NULL，即不传入时间结构，就是将select置于阻塞状态，一定等到监视文件描述符集合中某个文件描述符发生变化为止；
2. timeout=0秒0毫秒，就变成一个纯粹的非阻塞函数，不管文件描述符是否有变化，都立刻返回继续执行，文件无变化返回0，有变化返回一个正值；
3. timeout>0，这就是等待的超时时间，即 select在timeout时间内阻塞，超时时间之内有事件到来就返回了，否则在超时后不管怎样一定返回，返回值同上述。
4. 如果将3个描述符集合都设定为NULL则select相当于sleep函数，只是时间可以精确到微秒

返回值：

- 如果函数成功，则返回就绪的文件描述符的个数（即 `readfds`、`writefds` 和 `exceptfds` 中至少有一个位被 `select` 设置为 1 的文件描述符的总数）。
- 如果在超时时间内没有文件描述符就绪，则返回 0。
- 如果出现错误，则返回 -1，并设置全局变量 `errno` 来指示错误类型。

描述符集合操作函数（宏函数）

1. `FD_ZERO(fd_set *fdset)`

这个宏用于清空一个文件描述符集合。在调用`select`之前，通常使用此宏来**初始化文件描述符集合。**

2. `FD_SET(int fd, fd_set *fdset)`

这个宏用于将一个文件描述符（通常是socket、文件或其他I/O资源）添加到文件描述符集合中。在调用`select`之前，使用此宏来**指定希望监控的文件描述符。**

3. `FD_CLR(int fd, fd_set *fdset)`

这个宏用于从文件描述符集合中**移除一个文件描述符。**

4. `FD_ISSET(int fd, fd_set *fdset)`

这个宏用于检查文件描述符集合中是否包含了特定的文件描述符，并且该描述符是否已准备好进行读、写或异常处理（取决于`select`的调用方式）。在`select`返回后，通常会使用此宏来**检查哪些文件描述符已准备好。**

网络协议

域名系统DNS

由域名得到IP地址，由IP地址得到域名

DNS主要使用UDP协议进行通信，使用53号端口。但在某些情况下（如主从服务器之间的区域文件传输），也可能使用TCP协议。

域名的层次结构通常包括顶级域名（如.com、.net、.org等）、二级域名（如google.com中的google）、子域名等。

DNS由多个级别的域名服务器组成，包括根域名服务器、顶级域名服务器、权威域名服务器和递归解析器。

- 当用户尝试访问一个域名时，其浏览器或操作系统会向本地DNS解析器发送查询请求。
- 本地解析器会首先检查其缓存中是否有该域名的记录。如果有，则直接返回IP地址；否则，它将向递归解析器发送查询。
- 递归解析器会向根域名服务器发送查询，然后逐级向下查询，直到找到权威域名服务器并获得正确的IP地址。
- 权威域名服务器是存储特定域名与IP地址映射关系的服务器。

万维网协议WEB

1. **资源**：在Web中，任何有用的事物都可以被称为一项资源。这些资源可以包括网页（HTML文档）、图片、音频文件、视频文件、PDF文档等。它们通过HTTP（Hypertext Transfer Protocol，超文本传输协议）协议在Internet上进行传输。
2. **URL（统一资源定位符）**：每个Web资源都有一个唯一的URL，用于标识和定位该资源。用户通过在浏览器中输入URL或点击链接来访问这些资源。
3. **HTTP**：HTTP是Web的基础协议，用于在客户端（如Web浏览器）和服务器之间传输超文本文档（如HTML文档）。当用户在浏览器中访问一个URL时，浏览器会向服务器发送一个HTTP请求，服务器会响应这个请求并返回相应的资源。
4. **Web与Internet的区别**：虽然Web是当今Internet上最流行的应用之一，但它并不等同于Internet。Internet是一个全球性的、由许多网络互联而成的网络，而Web只是Internet上的一个应用层服务，它使用Internet提供的网络连接来传输资源。

超文本传输协议HTTP

HTTP协议是应用层的协议，使用的传输层协议是TCP协议，

1. **客户端发起请求**：当用户在浏览器中输入一个URL或点击链接时，浏览器会发起一个HTTP请求。这个请求包括请求方法（如GET、POST）、请求头、请求体以及请求目标（URL）等信息。
2. **服务器处理请求**：服务器接收到客户端发送的请求后，会解析请求头和请求体，根据请求的URL找到对应的资源，并根据请求方法进行相应的处理。
3. **服务器返回响应**：服务器处理完请求后，会生成一个HTTP响应。这个响应包括响应状态码、响应头以及响应体。服务器将完整的响应发送回客户端。
4. **客户端接收响应**：客户端接收到服务器返回的响应后，会解析响应头和响应体。根据响应状态码来判断请求是否成功，根据响应头中的Content-Type来确定响应体的类型，并进行相应的处理。
5. **连接关闭**：在请求和响应完毕后，客户端和服务器会根据请求头的Connection属性来决定是否关闭连接。

网络时间协议NTP

用于在计算机网络中同步计算机系统时钟。NTP的设计考虑了网络延迟和系统时钟误差，并提供了高精度的时间同步。NTP通常用于将计算机时钟同步到某个参考时间源，如原子钟或协调世界时（UTC）。

NTP的工作原理基于客户端-服务器架构。客户端向服务器发送一个NTP请求，服务器响应请求并返回当前时间。客户端使用此响应来更新其本地时钟。为了补偿网络延迟，NTP使用一种称为“往返延迟测量”的技术。客户端发送一个时间戳到服务器，服务器返回这个时间戳以及它接收和发送时间戳的时间。客户端使用这些信息来计算网络延迟，并据此调整其时钟。