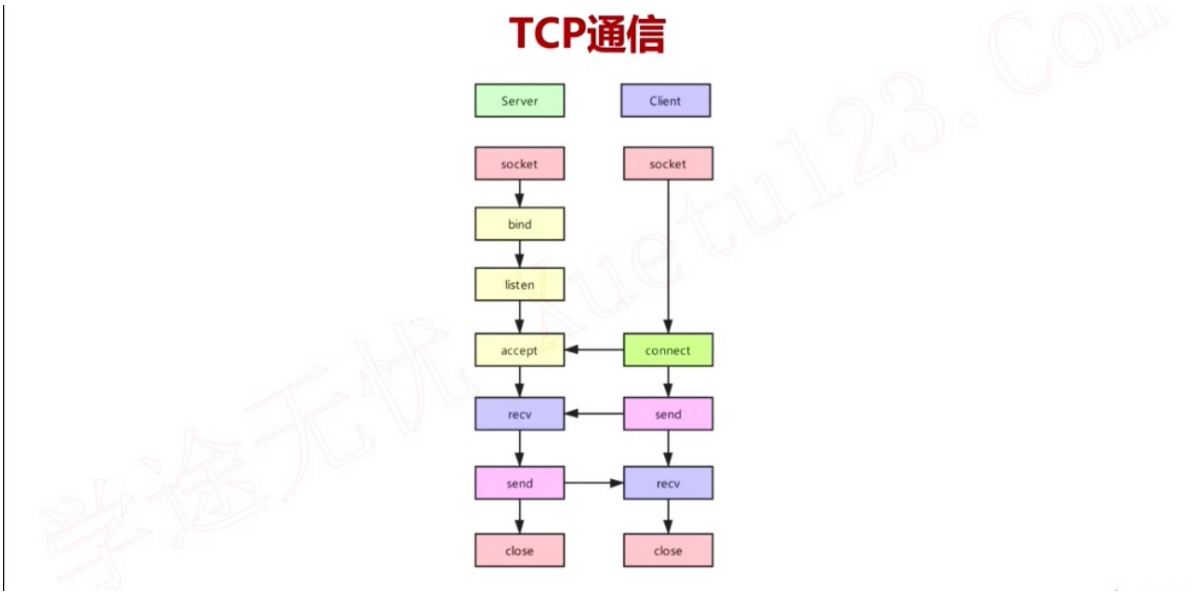
网络编程（常用函数与结构体说明）

基本的TCP套接字编程（详解）

TCP协议与相关套接字编程

UDP协议与相关套接字编程

一、TCP



1.socket常见的API

```
//创建套接字
int socket(int domain, int type, int protocol);
//绑定端口
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
//监听套接字
int listen(int sockfd, int backlog);
//接受请求
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
//建立连接
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

1.1 socket

```
// 创建 socket 文件描述符 (TCP/UDP, 客户端 + 服务器)
int socket(int domain, int type, int protocol);
```

domain参数：指明协议族

domain	说明
AF_INET	IPV4协议

domain	说明
AF_INET6	IPV6协议
AF_LOCAL	Unixt域协议
AF_ROUTE	路由套接字
AF_KEY	密钥套接字

type参数：指明套接字的类型

type	说 明
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_SEQPACKET	有序分组套接字
SOCK_RAW	原始套接字

如果你是要TCP通信的话，就要是要**SOCK_STREAM**作为类型，UDP就使用**SOCK_DGRAM**作为类型。

protocol参数：创建套接字的协议类别。你可以指明为TCP或UDP，但该字段一般直接设置为0就可以了，设置为0表示的就是默认，此时会根据传入的前两个参数自动推导出你最终需要使用的是哪种协议。

protocol	说 明
IPPROTO_TCP	TCP传输协议
IPPROTO_UDP	UDP传输协议
IPPROTO_SCTP	SCTP传输协议

返回值说明：

套接字创建成功返回一个文件描述符，创建失败返回-1，同时错误码会被设置。

1.2 bind

```
// 绑定端口号 (TCP/UDP, 服务器)
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

bind函数是把一个协议地址赋予一个套接字。

参数说明：

- sockfd参数：绑定的文件的文件描述符。也就是我们创建套接字时获取到的文件描述符。
- addr参数：这个参数是指向一个特定于协议的地址结构的指针。里面包含了协议族、端口号、IP地址等。（见下一节sockaddr结构中的介绍）
- addrlen参数：是该协议的地址结构的长度。

返回值说明：

绑定成功返回0，绑定失败返回-1，同时错误码会被设置。

1.3 listen

```
// 开始监听socket (TCP, 服务器)
int listen(int sockfd, int backlog);
```

listen函数仅由TCP服务器调用，表明服务器对外宣告它愿意接受连接请求，它做两件事：

1. 当socket函数创建一个套接字时，它被假设为一个主动套接字，也就是说将调用connect发起连接的客户端套接字。listen函数把一个未连接的套接字转换成一个被动套接字，指示内核应接受指向该套接字的连接请求。简单来说，服务器调用listen函数，就是告诉客户端你可以连接我了。
2. 第二个参数规定了内核应该为相应的套接字排队的最大连接个数。backlog提供了一个提示，提示系统该进程要入队的未完成连接的请求数量。其实际由系统决定，对于TCP而言，默认是128。

一旦队列满了，系统就会拒绝多余的连接请求，所有backlog的值应该基于服务器期望负载和处理量来选择，其中处理量是指接受连接请求与启动服务的数量。

一旦服务器调用了listen，所用的套接字就能接受连接请求。使用accept函数获得的连接请求并建立连接。

返回值：成功返回0，失败返回-1；

本函数通常应该在调用socket和bind这两个函数之后，并在调用accept函数之前调用。

1.4 accept

```
// 接收请求 (TCP, 服务器)
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

accept函数是由TCP服务器调用，用于从已完成连接队列队头返回下一个已完成连接。如果已完成连接队列为空，那么进程将被投入睡眠。

如果accept成功，那么其返回值是由内核自动生成的一个全新描述符，代表与所返回客户的TCP连接。我们常常称它的第一个参数为监听套接字(listening socket) 描述符(由socket创建，随后用作bind和listen的第一个参数的描述符)，称它的返回值为已连接套接字(connected socket) 描述符。区分这两个套接字非常重要。一个服务器通常仅仅创建一个监听套接字，它在该服务器的生命期内一直存在。内核为每个由服务器进程接受的客户连接创建一个已连接套接字(也就是说对于它的TCP三路握手过程已经完成)。当服务器完成对某个给定客户的服务时，相应的已连接套接字就被关闭。

总的来说，**函数accept所返回的文件描述符是新的套接字描述符**，该描述符连接到调用connect的客户端。**这个新的套接字描述符和原始套接字（sockfd）具有相同的套接字类型和地址族**。传给accept的原始套接字没有关联到这个连接，而是继续保持监听状态并接受其他连接请求。

1.5 connect

```
// 建立连接 (TCP, 客户端)
#included <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

说明：TCP客户端用connect函数来建立与TCP服务器的连接，客户端不用bind（），在操作系统底层自动实现

参数说明：

- sockfd参数：是由socket函数返回的套接字描述符，第二个以及第三个参数分别是指向套接字地址结构的指针和该结构的大小。

- 在connect中指定的地址是我们想要与之通信服务器地址。如果sockfd没有绑定到一个地址，connect会给调用者绑定一个默认地址。

  返回值说明：

```
若成功则为0，若出错则为-1；
```

1.6 write/send

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

功能：向套接字中发送数据

参数：sockfd：向套接字中发送数据

buf：要发送的数据的首地址

len：要发送的数据的字节

int flags：设置为MSG_DONTWAITMSG 时 表示非阻塞

设置为0时 功能和write一样

返回值：成功返回实际发送的字节数

失败：返回 -1

```
ssize_t write(int fd,const void *buf,size_t  count);
```

fd：要操作的文件的文件描述符，通过open函数打开文件时获取。

buf：指定写入数据对应的缓冲区，可以将需要的写入的内容存放到buf中，再将其写入文件里。

count：指定写入的字节数，单位是字节。

返回值：如果写操作顺利完成，则会返回写入的字节数；如果返回值为0，则表示未向文件中写入任何字符；如果写入出错，则会返回-1。

因为服务套接字本质就是一个文件描述符，因此可以直接使用write接口发送数据，具体方式与普通文件别无二致。

send接口和write接口使用上差别不大，前三个参数含义相同，最后一个参数代表发送策略，一般填0。

但有一点需要强调，write和send的第一个参数在服务端是服务套接字，因为每一个客户端（对端）的服务套接字各不相同，使用服务套接字可以标定唯一的客户端。在客户端就采用socket接口返回的套接字文件描述符即可。

1.7 recv/read

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

参数说明

1. 第一个参数指定接收端套接字描述符；

2. 第二个参数指明一个缓冲区，该缓冲区用来存放recv函数接收到的数据；
3. 第三个参数指明buf的长度；
4. 第四个参数一般置0。

同步Socket的recv函数的执行流程

当应用程序调用recv函数时：

1. recv先等待 SOCKET s 的发送缓冲中的数据被协议传送完毕，如果协议在传送s的发送缓冲中的数据时出现网络错误，那么recv函数返回**SOCKET_ERROR**；
2. 如果s的**发送缓冲区**中没有数据或者数据被协议成功发送完毕后，recv**先检查套接字s的接收缓冲区**；
3. 如果s的**接收缓冲区**中没有数据或者协议正在接收数据，那么recv就一直**等待**，直到协议把数据接收完毕；
4. 当协议把数据接收完毕，recv函数就把s的**接收缓冲区**中的数据**copy到buf**中。（注意协议接收到的数据可能大于buf的长度，所以 在这种情况下要调用几次recv函数才能把s的接收缓冲中的数据copy完。recv函数仅仅是copy数据，真正的接收数据是协议来完成的），recv函数返回其实际copy的字节数。如果recv在copy时出错，那么它返回**SOCKET_ERROR**；
5. 如果recv函数在等待协议接收数据时网络中断了，那么它**返回0**。

默认 socket 是阻塞的，阻塞与非阻塞`recv()`返回值没有区分，都是

`<0` 出错，
`=0` 连接关闭，
`>0` 接收到的数据长度大小

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

其中，fd是文件描述符，buf是接收数据的缓冲区地址，count表示期望读取的字节数。read函数会从指定的文件中读取count个字节到buf中，并返回实际读取到的字节数。在读取过程中，文件指针会根据读取的字节数偏移。

但这里有一个疑问：

首先我们知道UDP协议通信中，接收数据时规定接收长度最大为buf大小-1。这是防止接收数据超过buf大小，导致不能设置最后一位为'\0'：

```
int i = recvfrom(sockfd, buf, sizeof buf - 1, ...);
buf[i] = 0;
```

那为什么采用TCP协议接收数据时不再规定接收长度为size - 1呢?

这是因为TCP协议具有面向字节流的特点，也就是第一部分介绍面向字节流时说的一次传输数据可能不完整，需要多次传输。举个例子，假如客户端发送了"abcdefg"这个字符串，但是TCP的缓冲区太小，只存放了efg，那么服务端第一次接受的就是abcd，第二次接收时才能收到efg。因此接收长度应该是buf大小。

那么怎么判断何时数据全部接收了呢？**需要自定义通信协议**

1.8 close

头文件是unistd.h

它的原型如下：

```
#include <unistd.h>

int close(int fd);
```

close() 函数的返回值为 0 表示成功关闭文件描述符，返回值为 -1 表示关闭失败。

1.9 setsockopt

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

- sockfd：是套接字文件描述符，用于标识要设置选项的套接字。
- level：指定选项的级别，通常使用 SOL_SOCKET、SOL_TCP 或 SOL_UDP 等，具体取决于所设置选项的类型。
- optname：指定要
- 设置的选项的名称，如缓冲区大小、超时设置、广播选项等。
- optval：是一个指向存储选项值的缓冲区的指针。
- optlen：是 optval 缓冲区的长度。

1. 设置超时选项：

   ```
   int timeout = 5000; // 5秒
   setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));
   ```

   这将设置接收操作的超时时间为5秒。

2. 启用或禁用套接字广播：

   ```
   int broadcast = 1; // 启用广播
   setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast, sizeof(broadcast));
   ```

   这将启用套接字的广播功能。

3. 设置套接字缓冲区大小：

   ```
   int buffer_size = 8192;
   setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &buffer_size, sizeof(buffer_size));
   ```

   这将设置接收缓冲区大小为8192字节。

4. 设置套接字重用选项：

   ```
   int reuse = 1; // 启用套接字重用

   setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
   ```

这将启用套接字地址重用，允许多个套接字绑定到相同的地址。

5. 设置TCP选项（例如TCP_NODELAY）：

```
int tcp_nodelay = 1; // 启用TCP_NODELAY
setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &tcp_nodelay,
sizeof(tcp_nodelay));
```

这将启用TCP无延迟选项，用于减少延迟。

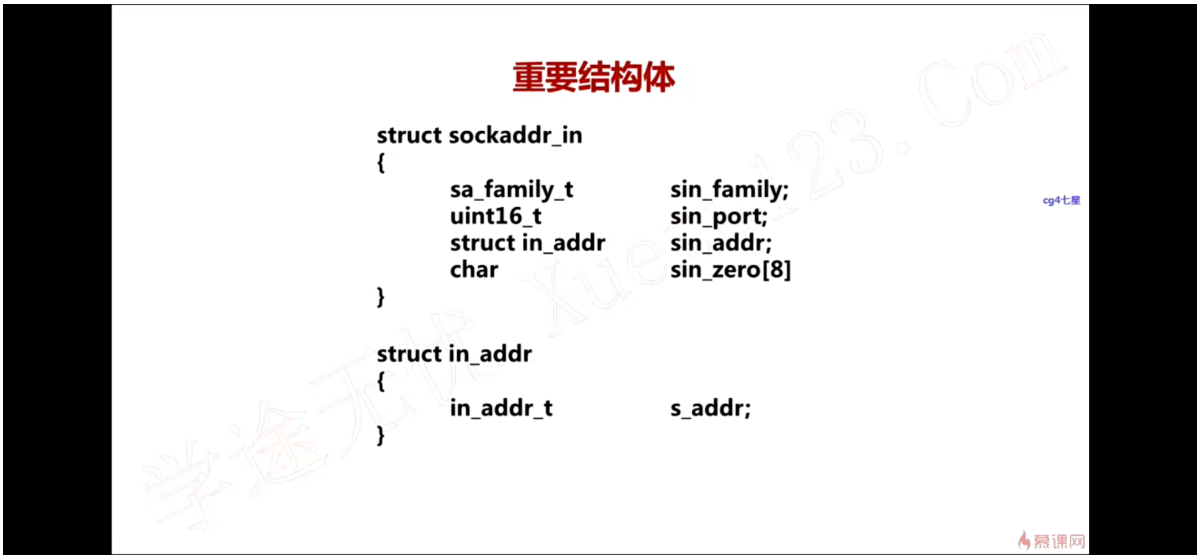
需要注意的是，不同的操作系统和套接字类型（如TCP套接字和UDP套接字）可能支持不同的选项。在使用 setsockopt() 函数时，务必查阅相关的系统文档或套接字编程文档，以确保正确设置选项。此外，错误处理也非常重要，以确保 setsockopt() 函数的调用是否成功。

1.10 bzero

将每个字节初始化为0。

```
例:
struct sockaddr_in local;
bzero(&local,sizeof(local));
```

2.sockaddr结构



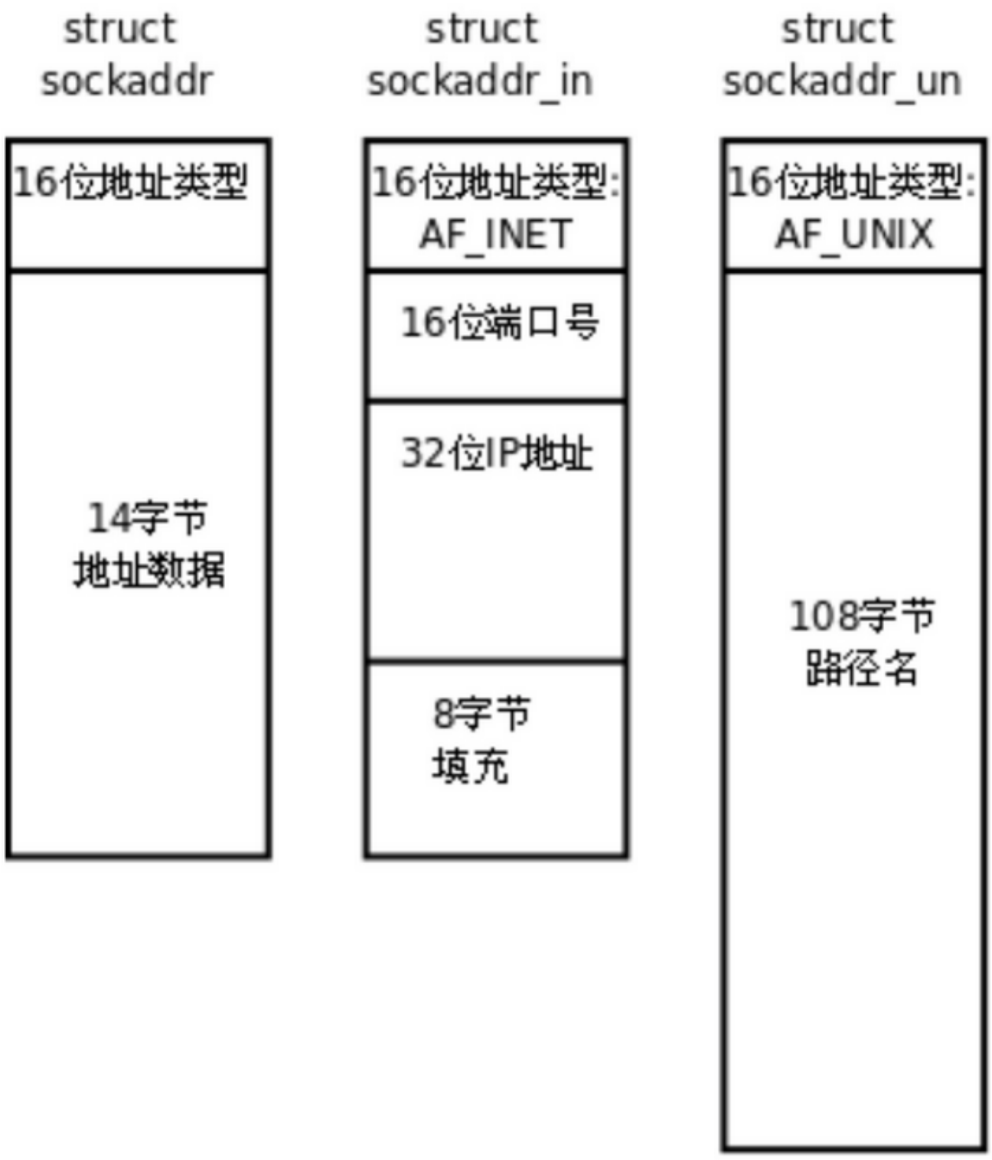
2.1 sockaddr

sockaddr在头文件`#include <sys/socket.h>`中定义，sockaddr的缺陷是：sa_data把目标地址和端口信息混在一起了，如下：

```
struct sockaddr {
     sa_family_t sin_family;//地址族
    //__SOCKADDR_COMMON (sa_);  另一种
     char sa_data[14]; //14字节，包含套接字中的目标地址和端口信息
    };
```

*2.2 sockaddr_in

socket API是一层抽象的网络编程接口,适用于各种底层网络协议,如IPv4、 IPv6， 然而，各种网络协议的地址格式并不相同



struct sockaddr_in：是通过网络来通信的sockaddr
struct sockaddr_un：是预间套接字，是通过本地来通信的sockaddr
struct sockaddr：是通用接口，想用网络就传in，用本地就传un，先对16位地址类型进行判断，是AF_INET，就在内部强转成in，是AF_UNIX，就强转成un

IPv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr_in结构体表示，包括16位地址类型，16位端口号和32位IP地址.
IPv4、 IPv6地址类型分别定义为常数AF_INET、 AF_INET6. 这样,只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体,就可以根据地址类型字段确定结构体中的内容.
socket API可以都用struct sockaddr *类型表示，在使用的时候需要强制转化成sockaddr_in；这样的好处是程序的通用性，可以接收IPv4，IPv6，以及UNIX Domain Socket各种类型的sockaddr结构体指针做为参数。

sockaddr_in 结构

以下类型在头文件<netinet/in.h>中

in_addr_t IPV4地址，一般为uint32_t
in_port_t TCP或UDP端口，一般为uint16_t

```
#define    __SOCKADDR_COMMON(sa_prefix) \
  sa_family_t sa_prefix##family

/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;         /* Port number.  */
    struct in_addr sin_addr;        /* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
               __SOCKADDR_COMMON_SIZE -
               sizeof (in_port_t) -
               sizeof (struct in_addr)];
  };
/*sin_zero用来将sockaddr_in结构填充到与struct sockaddr同样的长度，可以用bzero()或memset()函数将其置为零。指向sockaddr_in的指针和指向sockaddr的指针可以相互转换，这意味着如果一个函数所需参数类型是sockaddr类型时，你可以在函数调用的时候将一个指向sockaddr_in的指针转换为指向sockaddr的指针；或者相反。*/
```

in_addr结构

```
/* Internet address.  */
typedef uint32_t in_addr_t;
struct in_addr
  {
    in_addr_t s_addr;
  };
/*INADDR_ANY是一个常量，它指代的是一个特殊的IP地址，即0.0.0.0。在网络编程中，当一个进程需要绑定一个网络端口时，可以使用INADDR_ANY来指定该端口可以接受来自任何IP地址的连接请求。

具体来说，当一个进程需要监听某个网络端口时，需要调用bind()函数将该端口与一个IP地址绑定。如果使用INADDR_ANY作为IP地址参数，就表示该端口可以接受来自任何IP地址的连接请求。这样，无论是本地主机还是远程主机，只要它们能够访问该端口，就可以与该进程建立连接。

需要注意的是，使用INADDR_ANY绑定端口时，进程会接受所有来自该端口的连接请求，包括来自本地主机和远程主机的请求。因此，在实际应用中，需要根据实际需求选择合适的IP地址来绑定端口，以确保安全性和可靠性。*/
```

struct sockaddr_in结构体内部有三个参数：**sin_family**、**sin_port**、**sin_addr**。

第一个参数sin_family用以确定套接字类型，网络通信中就是网络套接字，即AF_INET

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
```

第二个参数sin_port用于确定该套接字需要连接的端口号，即本进程端口号。

这里需要格外注意的是，因为端口号需要用于网络传输使用，在网络中统一采用**大端存储的网络字节序**。**linux提供了接口htons来将数据转为大端存储形式**：

记忆这些函数的方法很简单，**htons**即代表**host**（主机）**to**（转到）**net**（网络）采用**short**短整形（uint16_t）的形式。其他函数类比即可。

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

```
uint16_t port = ...//端口号
addr.sin_port = htons(port);
```

第三个参数sin_addr用来表示本主机对应的IP地址。因为我们输入的IP地址一般采用字符串类型，但是在网络中需要使用in_addr_t类型（本质就是32位无符号整数）。

```
typedef uint32_t in_addr_t;
struct in_addr
{
 in_addr_t s_addr;
};
```

因此采用函数inet_addr可以将字符串转为in_addr_t类型，或者函数inet_aton：

inet_aton函数如果成功返回非0，失败返回0。

```
#include <netinet/in.h>
#include <arpa/inet.h>

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

in_addr_t inet_addr(const char *cp);
```



```
string IP = "127.0.0.1";//示例
addr.sin_addr.s_addr = inet_addr(IP.c_str());//方式一
int i = inet_aton(IP.c_str(), addr.sin_addr.s_addr);//方式二
```

3.类型转换函数

3.1 htonl()函数

3.2 htons()函数

3.3 ntohs()函数

3.4 ntohl()函数

3.5 新型网路地址转化函数inet_pton和inet_ntop

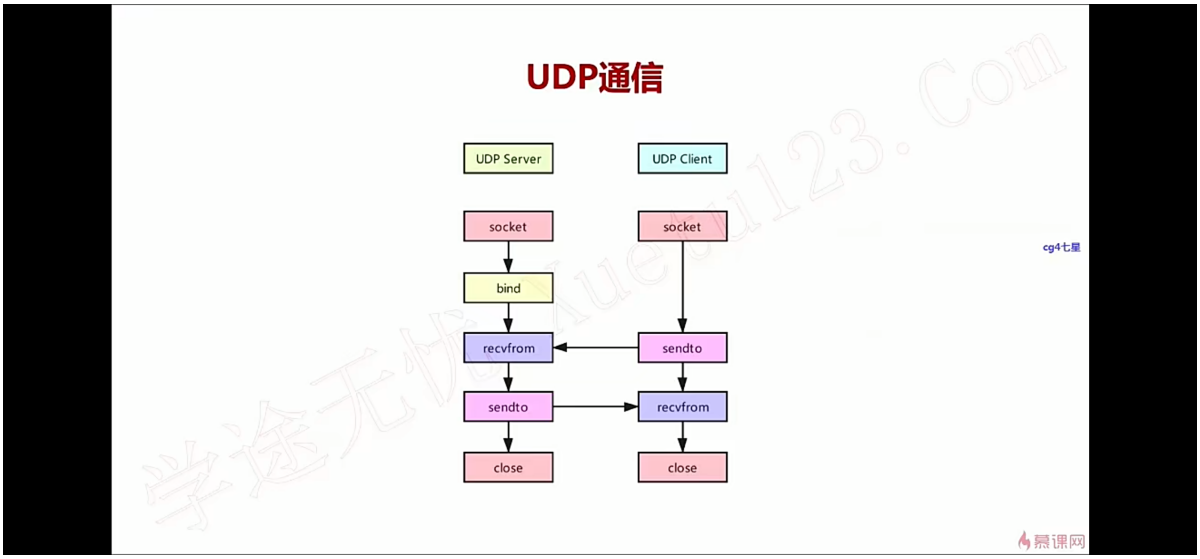
这两个函数是随IPv6出现的函数，对于IPv4地址和IPv6地址都适用，函数中p和n分别代表表达（presentation)和数值（numeric)。地址的表达格式通常是ASCII字符串，数值格式则是存放到套接字地址结构的二进制值。

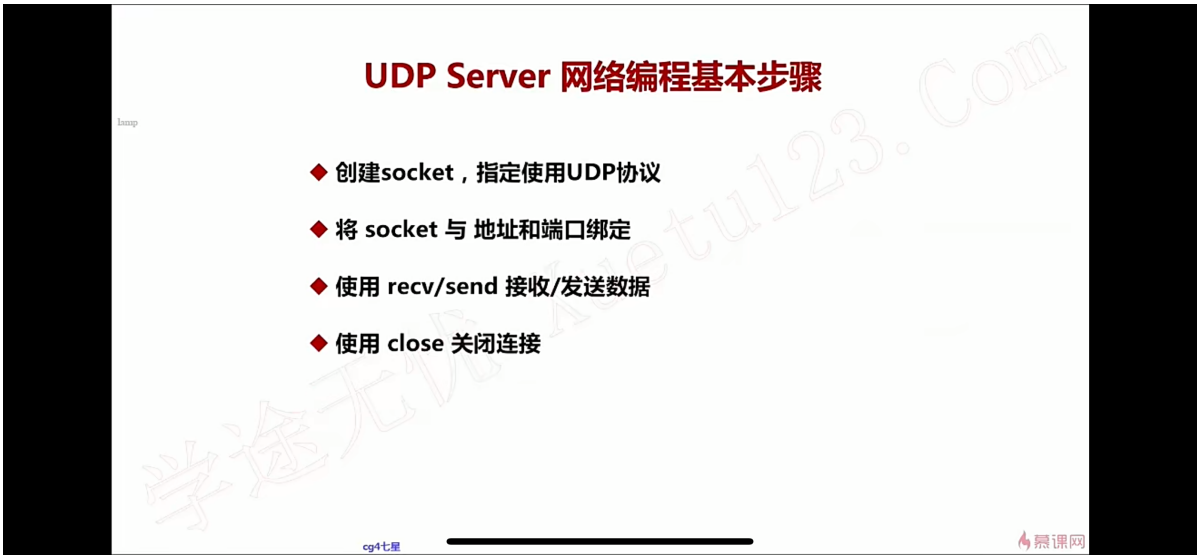
```
#include <arpe/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);      //将点分十进制的ip地址转化为用于网络传输的数值格式
        //返回值：若成功则为1，若输入不是有效的表达式则为0，若出错则为-1

const char * inet_ntop(int family, const void *addrptr, char *strptr, size_t len);      //将数值格式转化为点分十进制的ip地址格式
        ///返回值：若成功则为指向结构的指针，若出错则为NULL
```

3.6 inet_addr()、inet_aton()、inet_ntoa()、inet_pton()、inet_ntop()

二、UDP





1.常见函数

1.1 sendto()

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

第一个参数是本进程使用的套接字文件描述符（给对方发数据要把自己的信息传过去，这样对方才能回复）。

第二个参数是指针类型，指向要发送数据。第三个参数是数据大小。这两个参数可以参考write函数第二、三个参数含义。

第4个参数代表发送数据策略，一般填0代表正常操作。具体可以参考这篇博客：linux socket中 send recv函数的 flags参数

第四、五个参数代表数据接收方的struct sockaddr结构体。在使用sendto函数之前，要先获取到对方的IP地址和端口号，创建struct sockaddr_in结构体后，再调用sendto函数。

返回值为实际发送数据的长度，发送失败返回-1

1.2 recvfrom()

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,struct sockaddr
*src_addr, socklen_t *addrlen);
```

第一个参数同sendto函数，也是本进程创建的套接字文件描述符。

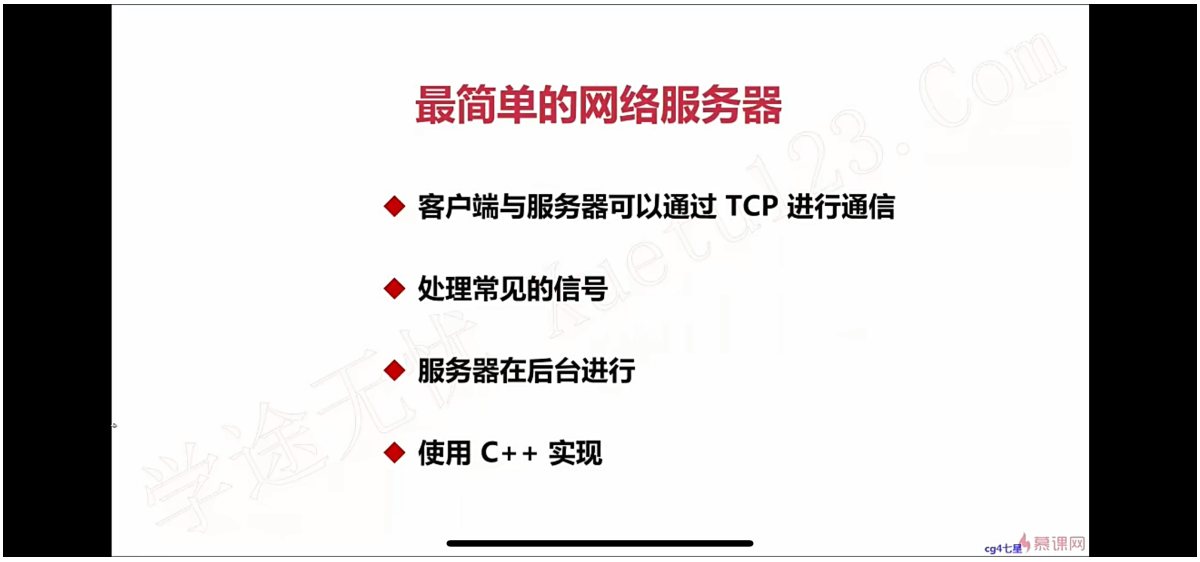
第二个参数为输出型参数，用于获取接收到的数据。

第三个参数是提供的接收数据容器的大小。这两个参数含义同系统read函数第二、三个参数。

第四个参数含义同sendto函数，不再解释，一般填0。

第五、六个参数也是输出型参数，用于获取数据发送方的网络通信结构体和结构体长度。本质是为了获取对方的IP地址和端口号，便于我方向对方主机回复数据。

返回值代表实际接收到的数据大小，接收失败返回-1。值得一提的是，如果对方进程关闭了，recvfrom函数并不会返回-1，而是一直阻塞



末尾小记

特殊数据类型

- socklen_t是一种用于表示socket地址结构长度的数据类型。在网络编程中，当需要传递socket地址结构时，需要指定该结构的长度，而socklen_t类型就是用来表示这个长度的。

  在不同的操作系统中，socklen_t类型可能会有所不同。在Linux系统中，socklen_t通常被定义为unsigned int类型，而在Windows系统中，则通常被定义为int类型。在使用socklen_t类型时，需要根据具体的操作系统和编译器来进行适当的类型转换。

- uint32_t是一种无符号的32位整数数据类型。在计算机编程中，经常会涉及到需要精确表示32位整数的情况，而uint32_t类型正好可以满足这个需求。

  在C和C++语言中，uint32_t类型是通过typedef定义的，实际上是unsigned int类型的别名。在使用uint32_t类型时，需要包含头文件<stdint.h>。

- pthread_t是一种用于表示线程ID的数据类型。在多线程编程中，每个线程都有一个唯一的线程ID，而pthread_t类型就是用来表示这个ID的。

  在使用pthread_t类型时，需要先创建线程，然后使用pthread_create函数来获取线程ID。pthread_t类型的变量通常会被声明为指针类型，并用于传递给其他线程相关的函数。

- pid_t是一种用于表示进程ID的数据类型。在操作系统中，每个进程都有一个唯一的进程ID，而pid_t类型就是用来表示这个ID的。

  在使用pid_t类型时，需要包含头文件<unistd.h>。

- size_t是一种用于表示内存大小的数据类型。在C和C++语言中，内存大小通常用字节(byte)为单位表示，而size_t类型就是用来表示这个大小的。

  在使用size_t类型时，需要包含头文件<stddef.h>。

总之，特殊的数据类型在编程中非常重要，它们可以帮助我们精确地表示各种不同的数据类型和数据结构，从而使程序更加健壮和可靠。

c_str()

功能：**c_str()** 函数可以**将 const string* 类型 转化为 const char* 类型**
头文件：`#include<cstring>`

> c_str()就是将C++的string转化为C的字符串数组，c_str()生成一个const char *指针，指向字符串的首地址
>
> 因为在c语言中没有string类型，必须通过string类对象的成员函数 c_str() 把 string 转换成c中的字符串样式

注意点：

c_str() 这个函数转换后**返回的是一个临时指针**，不能对其进行操作
所以因为这个数据是临时的，所以当有一个改变这些数据的成员函数被调用后，该数据就会改变失效；

大端存储的网络字节序

exit()函数

exit(1)；为异常退出　//只要括号内数字不为0都表示异常退出

exit(0)；为正常退出

bzero()函数

bzero() 能够将内存块（字符串）的前n个字节清零，在"string.h"头文件中，原型为：
void bzero(void *s, int n);

【参数】s为内存（字符串）指针，n 为需要清零的字节数。

bzero()将参数s 所指的内存区域前n 个字节全部设为零。

memset()函数

fgets()函数

字符串函数:strlen函数，strcpy函数，strcat函数，strcmp函数