【Linux编程】

gcc -c -D\_GNU\_SOURCE -Wall 01socket.c 编译为.o文件

gcc 01socket.o -o 01socket 编译为可执行文件

g++ server.cpp -o server

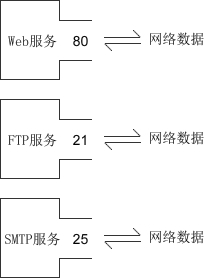
# 【网络概念】

## 【IP地址】

确定计算机位置的方式有多种，IP 地址是最常用的，例如，114.114.114.114 是国内第一个、全球第三个开放的 DNS 服务地址，127.0.0.1 是本机地址。  
 其实，我们的计算机并不知道 IP 地址对应的地理位置，当要通信时，只是将 IP 地址封装到要发送的数据包中，交给路由器去处理。路由器有非常智能和高效的算法，很快就会找到目标计算机，并将数据包传递给它，完成一次单向通信。

## 【端口】port

有了 IP 地址，虽然可以找到目标计算机，但仍然不能进行通信。一台计算机可以同时提供多种网络服务，例如Web服务、FTP服务、SMTP服务等，仅有 IP 地址，计算机虽然可以正确接收到数据包，但是却不知道要将数据包交给哪个网络程序来处理，所以通信失败。  
 为了区分不同的网络程序，计算机会为每个网络程序分配一个独一无二的端口号（Port Number），例如，Web服务的端口号是 80，FTP 服务的端口号是 21，SMTP 服务的端口号是 25。



## 【协议】protocol

协议就是网络通信的约定，由计算机组织制定，规定了很多细节，例如，如何建立连接，如何相互识别等。协议仅仅是一种规范，必须由计算机软件来实现。

所谓协议族（Protocol Family），就是一组协议（多个协议）的统称。最常用的是 TCP/IP 协议族，它包含了 TCP、IP、UDP、Telnet、FTP、SMTP 等上百个互为关联的协议，由于 TCP、IP 是两种常用的底层协议，所以把它们统称为 TCP/IP 协议族。

## 【数据传输方式】

### 【SOCK\_STREAM】

表示面向连接的数据传输方式。数据可以准确无误地到达另一台计算机，如果损坏或丢失，可以重新发送，但效率相对较慢。常见的http协议就使用 SOCK\_STREAM 传输数据，因为要确保数据的正确性，否则网页不能正常解析。

### 【SOCK\_DGRAM】

表示无连接的数据传输方式。计算机只管传输数据，不作数据校验，如果数据在传输中损坏，或者没有到达另一台计算机，是没有办法补救的。也就是说，数据错了就错了，无法重传。因为 SOCK\_DGRAM 所做的校验工作少，所以效率比 SOCK\_STREAM 高。

有可能多种协议使用同一种数据传输方式，所以在 socket 编程中，需要同时指明数据传输方式和协议。

综上所述：IP地址和端口能够在广袤的互联网中定位到要通信的程序，协议和数据传输方式规定了如何传输数据，有了这些，两台计算机就可以通信了。

# 【操作系统概念】

## 【文件描述符】file descriptor

Linux的哲学思想下，一切皆文件。文件可以是：网络连接（network connection），输入输出（FIFO），管道（a pipe），终端（terminal），硬盘上的实际文件，或者其它任何东西。

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向[内核](https://zh.wikipedia.org/wiki/%E5%86%85%E6%A0%B8" \o "内核)为每一个[进程](https://zh.wikipedia.org/wiki/%E8%BF%9B%E7%A8%8B" \o "进程)所维护的该进程打开文件的记录表。内核（kernel）利用文件描述符来访问文件。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。

在Linux中，socket 也被认为是文件的一种，和普通文件的操作没有区别，所以在网络数据传输过程中自然可以使用与文件 I/O 相关的函数。可以认为，两台计算机之间的通信，实际上是两个 socket 文件的相互读写。Socket与文件使用同一套文件描述符。

文件描述符有时也被称为文件句柄（File Handle），但“句柄”主要是 Windows 中术语，如果涉及到 Linux 平台将使用“描述符”。

我们应该记住3个已经打开的fd：

0：标准输入stdin STDIN\_FILENO

1：标准输出stdout STDOUT\_FILENO

2：标准错误stderr STDERR\_FILENO

以上宏定义在<unistd.h>中。一个最简单的使用fd的例子，就是使用<unistd.h>中的函数：write(1, "Hello, World!\n", 20);，在标准输出上显示“Hello, World!”。

# 【API】

<sys/socket.h> 头文件

## 【socket】

创建套接字，返回套接字文件描述符

int socket(int af, int type, int protocol);

|  |  |
| --- | --- |
| af | 地址族Address Family，也就是 IP 地址类型  常用的有 AF\_INET 和 AF\_INET6。AF 是“Address Family”的简写，INET是“Inetnet”的简写。AF\_INET 表示 IPv4 地址，例如 127.0.0.1；AF\_INET6 表示 IPv6 地址，例如 1030::C9B4:FF12:48AA:1A2B。 |
| type | 数据传输方式  常用的有 SOCK\_STREAM 和 SOCK\_DGRAM |
| protocol | 传输协议  常用的有 IPPROTO\_TCP 和 IPPTOTO\_UDP，分别表示 TCP 传输协议和 UDP 传输协 |

TCP套接字：

int tcp\_socket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

//IPPROTO\_TCP表示TCP协议

UDP套接字：

int udp\_socket = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

//IPPROTO\_UDP表示UDP协议

上面两种情况都只有一种协议满足条件，可以将 protocol 的值设为 0，系统会自动推演出应该使用什么协议，如下所示：

int tcp\_socket = socket(AF\_INET, SOCK\_STREAM, 0); //创建TCP套接字

int udp\_socket = socket(AF\_INET, SOCK\_DGRAM, 0); //创建UDP套接字

## 【bind】

把套接字绑定本机IP和端口

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

|  |  |
| --- | --- |
| sock | socket文件描述符 |
| addr | sockaddr结构体指针 |
| adderlen | addr变量的大小 |

### 【地址描述结构】

【sockaddr】

socket的通用address描述结构sockaddr：

struct sockaddr //16字节  
{  
    unsigned short sa\_family; //2字节AF,一般都是AF\_INET  
    char sa\_data[14]; //2字节端口号 + 4字节IP地址 + 8字节0  
};

这是一个16字节大小的结构（2+14），sa\_family可以认为是socket address family的缩写，也可能被简写成AF（Address Family）。另外的14字节是用来描述地址的。这是一种通用结构，事实上，当我们指定sa\_family=AF\_INET之后，sa\_data的形式也就被固定了下来：最前端的2字节用于记录16位的端口，紧接着的4字节用于记录32位的IP地址，最后的8字节清空为零。

【sockaddr\_in】

我们实际在构造sockaddr时候用到的结构是sockaddr\_in（意指socket address internet）：

struct sockaddr\_in //16字节  
{  
    unsigned short sin\_family; //2字节AF  
    unsigned short sin\_port; //2字节端口号  
    struct in\_addr sin\_addr; //4字节IP地址  
    char sin\_zero[8]; //8字节0  
};

我想，sin\_的意思，就是socket (address) internet吧，只不过把address省略掉了。sin\_addr被定义成了一个结构，这个结构实际上就是：

struct in\_addr  
{  
    unsigned long s\_addr; //4字节IP地址  
};



头文件依赖关系  
 <bits/socket.h>是包含在<sys/socket.h>中的，<netinet/in.h>是包含在<arpa/inet.h>中的，实际上我们在程序中往往就是：

#include <sys/socket.h>  
#include <arpa/inet.h>

### 【网络字节形式】

sockaddr结构中的IP地址（sin\_addr.s\_addr）和端口号（sin\_port）将被封装到网络上传送的数据包中，所以，它的结构形式需要保证是网络字节形式。我们这里用到的函数是htons()和htonl()，这些缩写的意思是：  
h: host，主机（本机）  
n: network，网络  
to: to转换  
s: short，16位（2字节，常用于端口号）  
l: long, 32位（4字节，常用于IP地址）  
“反过来”的函数也是存在的ntohs()和ntohl()。

## 【listen】

把套接字用于监听，并限制等待连接数

int listen(int sock, int backlog);

|  |  |
| --- | --- |
| sock | socket文件描述符 |
| backlog | 限制排队等待连接的socket的数量 |

当套接字正在处理客户端请求时，如果有新的请求进来，套接字是没法处理的，只能把它放进缓冲区，待当前请求处理完毕后，再从缓冲区中读取出来处理。如果不断有新的请求进来，它们就按照先后顺序在缓冲区中排队，直到缓冲区满。这个缓冲区，就称为请求队列（Request Queue）。

## 【accept】

阻塞等待客户端套接字的连接，返回用于通信的新套接字文件描述符

int accept(int listenSock,

struct sockaddr\* clientAddress,

unsigned int\* addressLength);

|  |  |
| --- | --- |
| listenSock | 处于listen的socket文件描述符 |
| clientAddress | 未初始化的结构体，因为服务端不知道哪个客户端会来连接 |
| addressLength | clientAddress的大小 |

当程序调用accept()的时候，程序有可能就停下来等accept()的结果。这就block（阻塞）。这如同调用std::cin的时候系统会等待输入直到回车一样。accept()是一个有可能引起block的函数。请注意我说的是“有可能”，这是因为accept()的block与否实际上决定与第一个参数sock的属性。这个文件描述符如果是block的，accept()就block，否则就不block。默认情况下，socket的属性是“可读可写”，并且，是阻塞的。

## 【connect】

client端connect()函数主动连接服务端，返回用于

int connect(int CommunicationSock,

struct sockaddr\* serverAddress,

unsigned int addressLength);

|  |  |
| --- | --- |
| CommunicationSock | socket文件描述符 |
| serverAddress | 服务端地址 |
| addressLength | serverAddress的大小 |

## 【recv和send】

用于通信的socket拥有S/C双方的信息，对这样的socket（实际上是S/C上的socket对）的操作，就如同对本地文件描述符的操作一样。但是，尽管我们可以使用read()和write()，但是，为了完美的控制，我们最好使用recv()和send()。

int send(int socket, const void\* msg, unsigned int msgLength, int flags);  
int recv(int socket, void\* rcvBuffer, unsigned int bufferLength, int flags);

## 【read和write】

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

|  |  |
| --- | --- |
| fd | 要读取的文件的描述符 |
| buf | 要接收的数据的缓冲区地址 |
| nbytes | 要读取的数据的字节数 |

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

|  |  |
| --- | --- |
| fd | 要写入的文件的描述符 |
| buf | 要写入的数据的缓冲区地址 |
| nbytes | 要写入的数据的字节数 |

## 【close】

这个函数用于关闭一个文件描述符，自然，也就可以用于关闭socket。

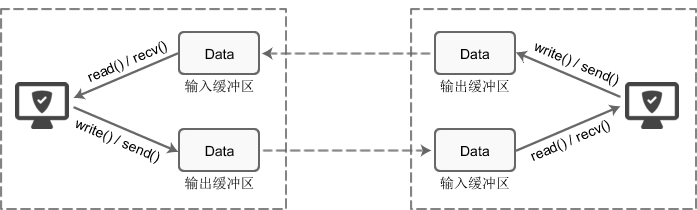
 int close (int fd);

## 【shutdown】

int shutdown(int sock, int howto);

默认情况下，close()/closesocket() 会立即向网络中发送FIN包，不管输出缓冲区中是否还有数据，而shutdown() 会等输出缓冲区中的数据传输完毕再发送FIN包。也就意味着，调用 close()/closesocket() 将丢失输出缓冲区中的数据，而调用 shutdown() 不会。

# 【缓冲区】

每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。  
  
 write()/send() 并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由TCP协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议负责的事情。  
 TCP协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

这些I/O缓冲区特性可整理如下：

I/O缓冲区在每个TCP套接字中单独存在；

I/O缓冲区在创建套接字时自动生成；

即使关闭套接字也会继续传送输出缓冲区中遗留的数据；

关闭套接字将丢失输入缓冲区中的数据。

【粘包问题】

数据的接收和发送是无关的，read()/recv() 函数不管数据发送了多少次，都会尽可能多的接收数据。也就是说，read()/recv() 和 write()/send() 的执行次数可能不同。  
 例如，write()/send() 重复执行三次，每次都发送字符串"abc"，那么目标机器上的 read()/recv() 可能分三次接收，每次都接收"abc"；也可能分两次接收，第一次接收"abcab"，第二次接收"cabc"；也可能一次就接收到字符串"abcabcabc"。  
  
假设我们希望客户端每次发送一位学生的学号，让服务器端返回该学生的姓名、住址、成绩等信息，这时候可能就会出现问题，服务器端不能区分学生的学号。例如第一次发送 1，第二次发送 3，服务器可能当成 13 来处理，返回的信息显然是错误的。  
  
这就是数据的“粘包”问题，客户端发送的多个数据包被当做一个数据包接收。也称数据的无边界性，read()/recv() 函数不知道数据包的开始或结束标志（实际上也没有任何开始或结束标志），只把它们当做连续的数据流来处理。