1.网络编程就是编写程序使两台联网的计算机交换数据

2.socket=插座=套接字，通信方式

应用在web服务器和浏览器

浏览器接收用户输入的Url,向服务器发起请求，服务器解析收到的url，将相关页面内容返回给浏览器，浏览器渲染完呈现给用户

对硬件的操作-接口

对设备---文件

每个文件配一个ID---文件描述符

1. 标准输入文件 键盘
2. 标准输出文件 显示器

网络连接也是一个文件，也有文件描述符

Socket()函数创建网络连接，或者打开一个网络文件 返回值是文件描述符

有了文件描述符就可以使用普通文件操作函数来运输数据

Read()读取从远程计算机传来的数据

Write()向远程计算机写入数据

在unix/linux系统叫文件描述符，Windows系统叫文件句柄

Windows会区分socket和文件，需要调用专门的数据传输函数

流格式套接字=面向连接的套接字，SOCK\_STREAM表示

数据在传输过程不会消失，顺序传输，发送和接收不同步

使用TCP协议

科普：TCP确保数据正确性，ip数据控制如何从源头到目的地，路由

流格式套接字内部有缓冲区，接收端只有缓冲区满了才一次性读取，或者分好几次读取

浏览器的http协议就是这个面向连接的套接字

数据报格式套接字-无连接的套接字，SOCK\_DGRAM

不作校验，无法重传

传输效率高

快速传输，数据可能损毁或丢失，限制每次传输数据大小，数据的发送和接收同步

跟快递似的

不可靠、不按顺序的追求速度的套接字

使用IP 协议作路由，但是用UDP用户数据报协议不使用TCP

QQ视频聊天和语音聊天就是用SOCK\_DGRAM传输数据

OSI-开放式系统互联

七层：

物理层-数据链路层-网络层-传输层-会话层-表示层-应用层

4层：接口层-网络层-传输层-应用层

网络层负责数据的传输 路径、地址选择，IP ARP地址解析协议

传输层确认数据传输、进行纠错，TCP、UDP

应用层：HTTP、FTP、SMTP简单邮件传输协议

网络模型-对数据进行封装

Socket编程就是在传输层基础上，使用TCP/UDP协议

上面协议-TCP/IP协议族

IP：一个计算机有一个、一个局域网有一个（对外就像一个计算机）

网络通信必须知道对方iP;

MAC地址：

Ip只能单位到一个局域网，具体的计算机要看mac地址，出厂设置好了

局域网中的路由器、交换机记录每台计算机MAC地址

数据包中有对方的IP地址和MAC地址

端口号：

虽然有IP和MAC地址，但是计算机不知道将数据包交给哪个网络程序处理，所以要用端口号

区分网络程序用端口号：

Web 80;FTP 21;SMTP 25

创建套接字

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

Af-地址族：AF\_INET ipv4；AF\_INET6 ipv6地址

127.0.0.1本机地址

Type：数据传输方式 常用：SOCK\_STREAM，SOCK\_DGRAM

Protocol：IPPROTO\_TCP,IPPROTO\_UDP TCP和UDP传输协议

LINUX下的该函数返回一个int类型的文件描述符，win的返回socket类型，

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

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

系统自动推算出传输协议，所以0取代第三个参数

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

绑定套接字并建立连接

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

int bind(SOCKET sock, const struct sockaddr \*addr, int addrlen); //Windows

//创建套接字

int serv\_sock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

//创建sockaddr\_in结构体变量

struct sockaddr\_in serv\_addr;

memset(&serv\_addr, 0, sizeof(serv\_addr)); //每个字节都用0填充

serv\_addr.sin\_family = AF\_INET; //使用IPv4地址

serv\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); //具体的IP地址

serv\_addr.sin\_port = htons(1234); //端口

//将套接字和IP、端口绑定

bind(serv\_sock, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr));

这里我们使用 sockaddr\_in 结构体，然后再强制转换为 sockaddr 类型

struct sockaddr\_in{

sa\_family\_t sin\_family; //地址族（Address Family），也就是地址类型

uint16\_t sin\_port; //16位的端口号

struct in\_addr sin\_addr; //32位IP地址

char sin\_zero[8]; //不使用，一般用0填充

};

sin\_prot 为端口号。uint16\_t 的长度为两个字节，理论上端口号的取值范围为 0~65536，但 0~1023 的端口一般由系统分配给特定的服务程序，例如 Web 服务的端口号为 80，FTP 服务的端口号为 21，所以我们的程序要尽量在 1024~65536 之间分配端口号。

struct in\_addr{

in\_addr\_t s\_addr; //32位的IP地址

};

建立连接

int connect(int sock, struct sockaddr \*serv\_addr, socklen\_t addrlen); //Linux

int connect(SOCKET sock, const struct sockaddr \*serv\_addr, int addrlen); //Windows

服务端-bind之后，listen让套接字进入被监听状态

Accept函数接听客户端请求

int listen(int sock, int backlog); //Linux

int listen(SOCKET sock, int backlog); //Windows

 backlog 的值设置为 SOMAXCONN----表示系统决定请求队列长度

accept() 返回一个新的套接字来和客户端通信，addr 保存了客户端的IP地址和端口号，而 sock 是服务器端的套接字，大家注意区分。后面和客户端通信时，要使用这个新生成的套接字，而不是原来服务器端的套接字。

两台计算机之间的通信相当于两个套接字之间的通信，在服务器端用 write() 向套接字写入数据，客户端就能收到，然后再使用 read() 从套接字中读取出来，就完成了一次通信。

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

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

fd 为要读取的文件的描述符，buf 为要接收数据的缓冲区地址，nbytes 为要读取的数据的字节数。  
  
read() 函数会从 fd 文件中读取 nbytes 个字节并保存到缓冲区 buf，成功则返回读取到的字节数（但遇到文件结尾则返回0），失败则返回 -1。

Windows的

从服务器端发送数据使用 send() 函数，它的原型为：

int send(SOCKET sock, const char \*buf, int len, int flags);

在客户端接收数据使用 recv() 函数，它的原型为：

int recv(SOCKET sock, char \*buf, int len, int flags);

Socket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。

一旦将数据写入到缓冲区，函数就可以成功返回

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

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。

输入输出缓冲区的默认大小一般都是 8K，可以通过 getsockopt() 函数获取

阻塞模式：

冲区的可用空间长度小于要发送的数据 write()/send() 会被阻塞（暂停执行）

如果TCP协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒。

如果要写入的数据大于缓冲区的最大长度，那么将分批写入

直到所有数据被写入缓冲区 write()/send() 才能返回

读时候：

如果缓冲区中有数据，那么就读取，否则函数会被阻塞，直到网络上有数据到来

如果要读取的数据长度小于缓冲区中的数据长度，那么就不能一次性将缓冲区中的所有数据读出，剩余数据将不断积压，直到有 read()/recv() 函数再次读取。

直到读取到数据后 read()/recv() 函数才会返回，否则就一直被阻塞。

上一步动作没有完成，下一步动作将暂停

数据粘包问题

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。