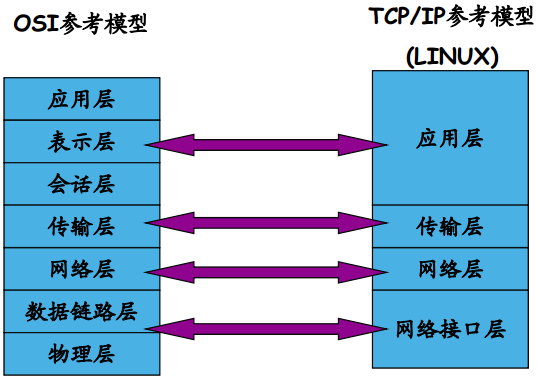
本章节主要介绍linxu网络模型、以及常用的网络协议分析以太网协议、IP协议、TCP协议、UDP协议

一、网络模型



TCP/IP分层模型的四个协议层分别完成以下的功能：

**第一层　网络接口层**

　 　网络接口层包括用于协作IP数据在已有网络介质上传输的协议。实际上TCP/IP标准并不定义与ISO数据链路层和物理层相对应的功能。相反，它定义像 地址解析协议（Address Resolution Protocol,ARP）这样的协议，提供TCP/IP协议的数据结构和实际物理硬件之间的接口。

**第二层　网络层**

　 　网络层对应于OSI七层参考模型的网络层。本层包含IP协议、RIP协议（Routing Information Protocol，路由信息协议），负责数据的包装、寻址和路由。同时还包含网间控制报文协议（Internet Control Message Protocol,ICMP）用来提供网络诊断信息。

**第三层　传输层**

　 　传输层对应于OSI七层参考模型的传输层，它提供两种端到端的通信服务。其中TCP协议（Transmission Control Protocol）提供可靠的数据流运输服务，UDP协议（Use Datagram Protocol）提供不可靠的用户数据报服务。

　　第四层　应用层

应用层对应于OSI七层参考模型的应用层和表达层。因特网的应用层协议包括Finger、Whois、FTP（文件传输协议）、Gopher、 HTTP（超文本传输协议）、Telent（远程终端协议）、SMTP（简单邮件传送协议）、IRC（因特网中继会话）、NNTP（网络新闻传输协议） 等。

OSI以及TCP/IP协议对应关系表：

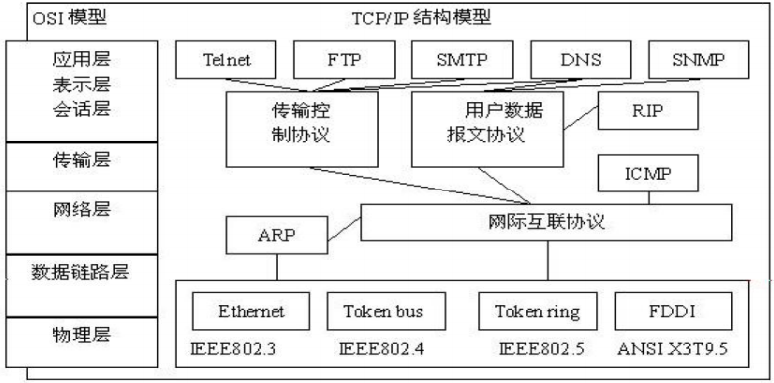
|  |  |  |
| --- | --- | --- |
| OSI七层模型 | TCP/IP四层模型 | 传输的数据 |
| 应用层 | 应用层 | 数据 |
| 表示层 | 应用层 | 数据 |
| 会话层 | 应用层 | 数据 |
| 传输层 | 传输层 | 段 |
| 网络层 | 网络层 | 包 |
| 数据链路层 | 网络接口层 | 帧 |
| 物理层 | 网络接口层 | 比特流 |

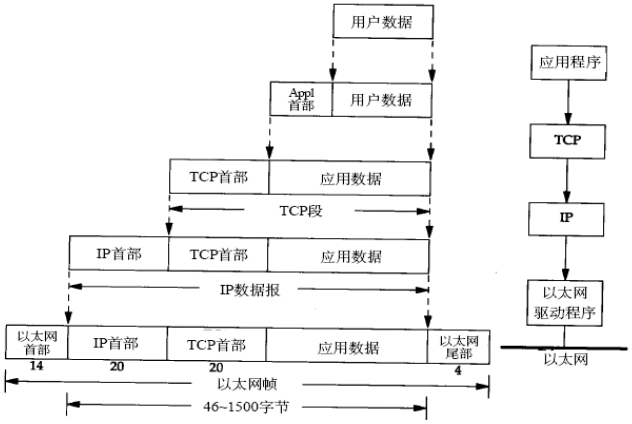
**二、协议分析**

网络协议是网络上所有设备(网络服务器、计算机及交换机、路由器、防火墙等)之间通信规则的集合，它规定了通信时信息必须采用的格式和这些格式的意义。大多数网络都采用分层的体系结构，每一层都建立在它的下层之上，向它的上一层提供一定的服务，而把如何实现这一服务的细节对上一层加以屏蔽。

一台设备上的第n层与另一台设备上的第n层进行通信的规则就是第n层协议。在网络的各层中存在着许多协议，接收方和发送方同层的协议必须一致， 否则一方将无法识别另一方发出的信息。网络协议使网络上各种设备能够相互交换信息。常见的协议有：TCP/IP协议、以太网、UDP协议等。

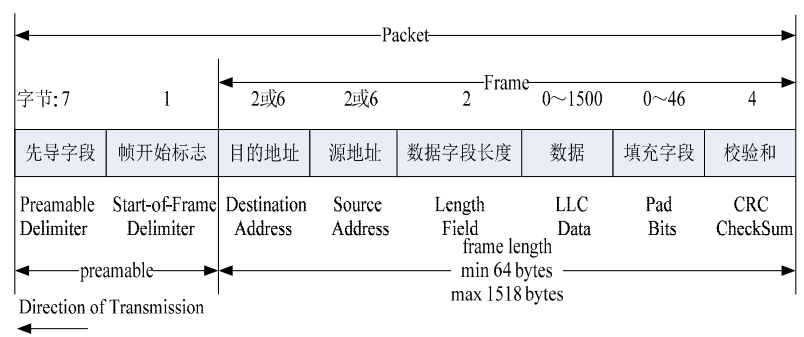
**2.1协议架构**





不同的协议层对数据包有不同的称谓，在传输层叫做段（segment），在网络层叫做数据报（datagram），在链路层叫做帧（frame）。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

其实在链路层之下还有物理层，指的是电信号的传递方式，比如现在以太网通用的网线（双绞线）、早期以太网采用的同轴电缆（现在主要用于有线电视）、光纤等都属于物理层的概念。



(1)其中的源地址和目的地址是指网卡的硬件地址（也叫MAC 地址），长度是48 位，是在网卡出厂时固化的。

(2)注意网卡芯片（例如DM9000A）收到的数据就是如上所示的一长串数据；其中包括以太网帧头、IP报报头、传输层协议段头、应用层所需数据。

(3)以太网帧中的数据长度规定最小46 字节，最大1500 字节，ARP 和RARP 数据包的长度不够46 字节，要在后面补填充位。最大值1500 称为以太网的最大传输单元（MTU），不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包度大于拨号链路的MTU了，则需要 对数据包进行分片fragmentation）。ifconfig 命令的输出中也有“MTU:1500”。注意，MTU 个概念指数据帧中有效载荷的最大长度，不包括帧首部的长度。

2.3IP协议格式



**版本:**占 4 位,指 IP 协议的版本目前的 IP 协议版本号为 4 (即 IPv4)

**首部长度:**占4位,可表示的最大数值是15个单位(一个单位为 4 字节)因此IP 的首部长度的最大值是 60 字节

**区分服务:**占8位,用来获得更好的服务,在旧标准中叫做服务类型,但实际上一直未被使用过.1998 年这个字段改名为区分服务.只有在使用区分服务(DiffServ)时,这个字段才起作用.一般的情况下都不使用这个字段

**总长度:**占16位,指首部和数据之和的长度,单位为字节,因此数据报的最大长度为 65535 字节.总长度必须不超过最大传送单元 MTU

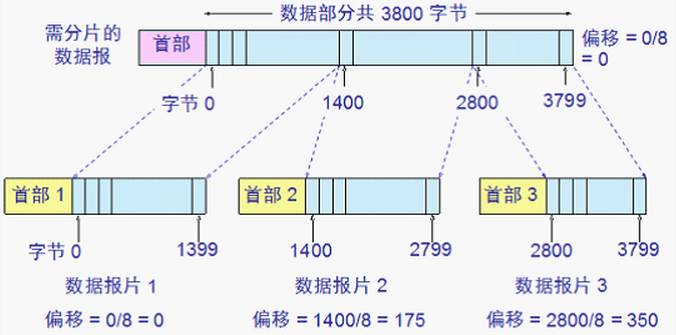
**标识:**占16位,它是一个计数器,用来产生数据报的标识

标志(flag):占3位,目前只有前两位有意义

MF标志字段的最低位是 MF (More Fragment) MF=1 表示后面“还有分片”。MF=0 表示最后一个分片

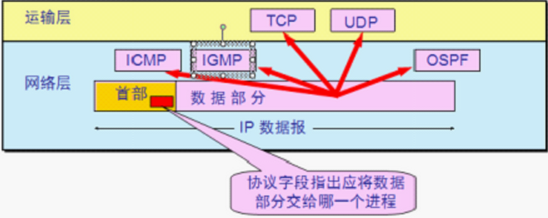
DF 标志字段中间的一位是 DF (Don't Fragment) 只有当 DF=0 时才允许分片

**片偏移:**占12位,指较长的分组在分片后某片在原分组中的相对位置.片偏移以 8 个字节为偏移单位



**生存时间:**占8位,记为TTL (Time To Live) 数据报在网络中可通过的路由器数的最大值,TTL 字段是由发送端初始设置一个 8 bit字段.推荐的初始值由分配数字 RFC 指定,当前值为 64.发送 ICMP 回显应答时经常把 TTL 设为最大值 255

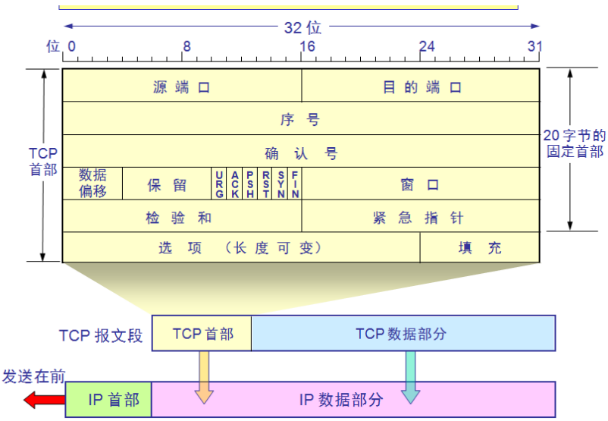
**协议:**占8位,指出此数据报携带的数据使用何种协议以便目的主机的IP层将数据部分上交给哪个处理过程, 1表示为 ICMP 协议, 2表示为 IGMP 协议, 6表示为 TCP 协议, 17表示为 UDP 协议



**首部检验和:**占16位,只检验数据报的首部不检验数据部分.这里不采用 CRC 检验码而采用简单的计算方法

**源地址和目的地址:**都各占 4 字节,分别记录源地址和目的地址

2.4 TCP协议格式



1. **源端口和目的端口** 各占2字节，端口号加上IP地址，共同构成socket。互相通信的进程使用一对socket，包括协议、源IP、源端口、目的IP、目的端口，这五个元素唯一确定一个TCP连接。

**2. 序号**占4字节，是TCP段所发送的数据部分第一个字节的序号。在TCP传送的数据流中，每一个字节都有一个序号。建立连接时，发送方将初始序号（Initial Sequence Number, ISN）填写到第一个发送的TCP段序号中。

**3. 确认号**　占4字节，是期望收到对方下次发送的数据的第一个字节的序号，也就是期望收到的下一个TCP段的首部中的序号，等于已经成功收到的TCP段的最 后一个字节序号加1。确认号在ACK标志为1时有意义，除了主动发起连接的第一个TCP段不设置ACK标志外，其后发送的TCP段都会设置ACK标志。

**4. 数据偏移** 占4比特，表示数据开始的地方离TCP段的起始处有多远。实际上就是TCP段首部的长度。由于首部长度不固定，因此数据偏移字段是必要的。数据偏移以32位为长度单位，因此TCP首部的最大长度是60（15\*4）个字节。

**5. 控制位** 一共6个，占6比特，设置为1时有效。按顺序依次为：URG、ACK、PSH、RST、SYN、FIN。

|  |  |
| --- | --- |
| URG | 紧急位，为1时，首部中的紧急指针有效 |
| ACK | 确认位，为1时，首部中的确认号有效 |
| PSH | 推位，为1时，要求把数据尽快交给应用程序 |
| RST | 复位标志，为1时，复位连接，一般在出错或关闭连接时使用 |
| SYN | 同步位，在建立连接时使用，当SYN=1而ACK=0时，表明这是一个连接请求报文段。对方若同意建立连接，在发回的报文段中使SYN=1和ACK=1 |
| FIN | 结束位，为1时，表示发送方完成了数据发送 |

**6. 窗口**占2字节，表示报文段发送方期望接收的字节数，可接收的序号范围是从接收方的确认号开始到确认号加上窗口大小之间的数据。

**7. 校验和**校验和包含了伪首部、TCP首部和数据，校验和是TCP强制要求的，由发送方计算，接收方验证。

**8. 紧急指针**　　URG标志为1时，紧急指针有效，表示数据需要优先处理。紧急指针指出在TCP段中的紧急数据的最后一个字节的序号，使接收方可以知道紧急数据共有多长。

**9. 选项**　最常用的选项是最大段大小（Maximum Segment Size，MSS），向对方通知本机可以接收的最大TCP段长度。MSS选项只在建立连接的请求中发送。

**2.5 UDP协议格式**

****

UDP是一种无连接的、不可靠的传输层协议；

在完成进程到进程的通信中提供了有限的差错检验功能；

设计比较简单的UDP协议的目的是希望以最小的开销来达到网络环境中的进程通信目的；

进程发送的报文较短，同时对报文的可靠性要求不高，那么可以使用UDP协议。

**linux中socket的理解---4**

**一、socket**

一般来说socket有一个别名也叫做套接字。

socket起源于Unix，都可以用”打 开open –> 读写write/read –> 关闭close”模式来操作。Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写 IO、打开、关闭）。

说白了Socket是应用层与TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后 面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议，而不需要让用户自己去定义什么时候需要指定哪个协议哪个函数。

其实socket也没有层的概念，它只是一个facade设计模式的应用，让编程变的更简单。是一个软件抽象层。在网络编程中，我们大量用的都是通过socket实现的。

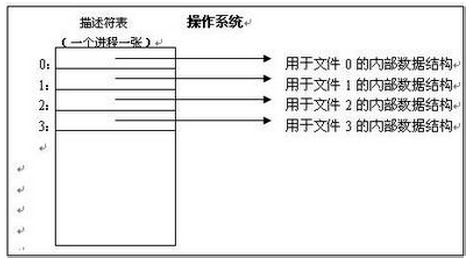
**1.1 套接字 描述符**

其实就是一个整数，我们最熟悉的句柄是0、1、2三个，0是标准输入，1是标准输出，2是标准错误输出。0、1、2是整数表示的，对应的FILE \*结构的表示就是stdin、stdout、stderr

套接字API最初是作为UNIX操作系统的一部分而开发的，所以套接字API 与系统的其他I/O设备集成在一起。特别是，当应用程序要为因特网通信而创建一个套接字（socket）时，操作系统就返回一个小整数作为描述符 （descriptor）来标识这个套接字。然后，应用程序以该描述符作为传递参数，通过调用函数来完成某种操作（例如通过网络传送数据或接收输入的数 据）。

在许多操作系统中，套接字描述符和其他I/O描述符是集成在一起的，所以应用程序可以对文件进行套接字I/O或I/O读/写操作。

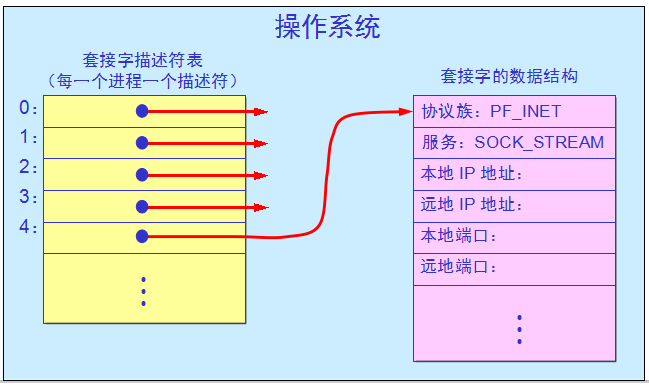
当应用程序要创建一个套接字时，操作系统就返回一个小整数作为描述符，应用程 序则使用这个描述符来引用该套接字需要I/O请求的应用程序请求操作系统打开一个文件。操作系统就创建一个文件描述符提供给应用程序访问文件。从应用程序 的角度看，文件描述符是一个整数，应用程序可以用它来读写文件。下图显示，操作系统如何把文件描述符实现为一个指针数组，这些指针指向内部数据结构。



对于每个程序系统都有一张单独的表。精确地讲，系统为每个运行的进程维护一张单 独的文件描述符表。当进程打开一个文件时，系统把一个指向此文件内部数据结构的指针写入文件描述符表，并把该表的索引值返回给调用者。应用程序只需记住 这个描述符，并在以后操作该文件时使用它。操作系统把该描述符作为索引访问进程描述符表，通过指针找到保存该文件所有的信息的数据结构。

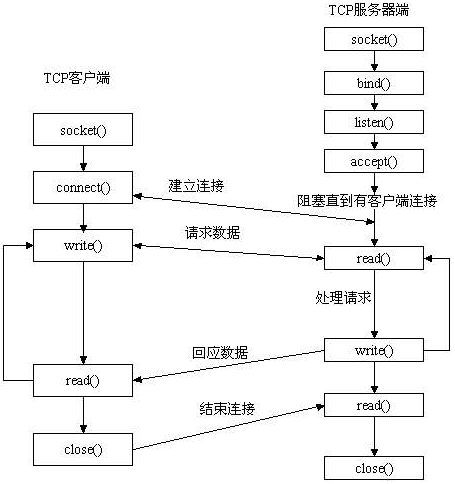
针对套接字的系统数据结构：

1）、套接字API里有个函数socket，它就是用来创建一个套接字。套接字设计的总体思路是，单个系统调用就可以创建任何套接字，因为套接字是相当 笼统的。一旦套接字创建后，应用程序还需要调用其他函数来指定具体细节。例如调用socket将创建一个新的描述符条目：



2)、虽然套接字的内部数据结构包含很多字段，但是系统创建套接字后，大多数字字段没有填写。应用程序创建套接字后在该套接字可以使用之前，必须调用其他的过程来填充这些字段。

二、基本的socket接口函数



服务器端先初始化/创建Socket，然后与端口绑定/绑定地址 (bind)，对端口进行监听(listen)，调用accept阻塞/等待连续，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连 接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送 给客户端，客户端读取数据，最后关闭连接，一次交互结束。

**2.1socket函数**

函数原型

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

返回值：

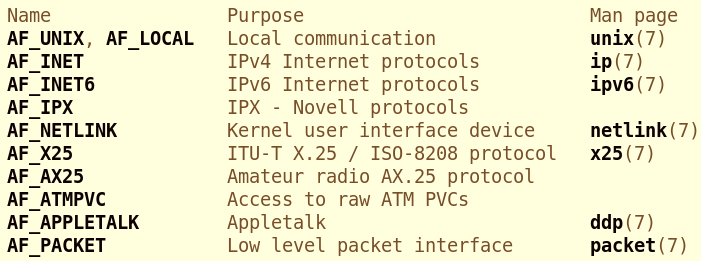
　//返回sockfd     sockfd是描述符,类似于open函数。

函数功能：

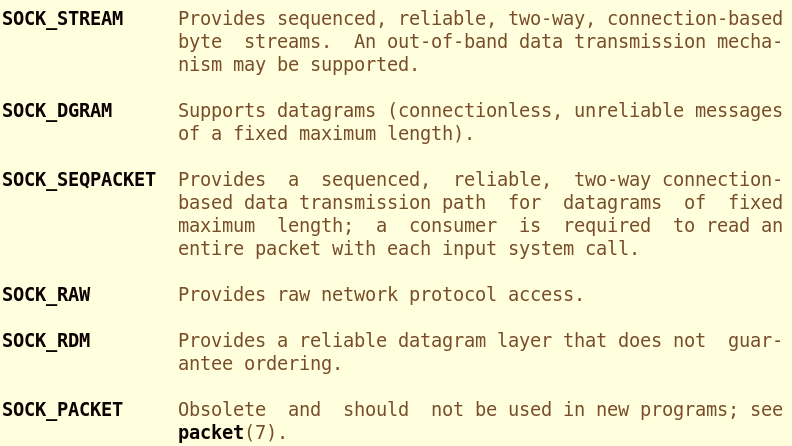
socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件 描述字，而socket()用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些 读写操作。

函数参数：

protofamily：即协议域，又称为协议族（family）。常用的协 议族有，AF\_INET(IPV4)、AF\_INET6(IPV6)、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、 AF\_ROUTE等等。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，如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。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等。



protocol：就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议

注意：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

当我们调用socket创建一个socket时，返回的socket描述字它存在 于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、 listen()时系统会自动随机分配一个端口

2.2bind()函数

函数功能：

bind()函数把一个地址族中的特定地址赋给socket，也可以说是绑定ip端口和socket。例如对应AF\_INET、AF\_INET6就是把一个ipv4或ipv6地址和端口号组合赋给socket。

函数原型：



函数参数：

1.函数的三个参数分别为：sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。

2.addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同，

3.addrlen：对应的是地址的长度。

通用函数类型：

struct sockaddr{

sa\_family\_t sa\_family;

char sa\_data[14];

}

如ipv4对应的是：

struct sockaddr\_in {

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

in\_port\_t sin\_port; /\* port in network byte order 2字节\*/

struct in\_addr sin\_addr; /\* internet address 4字节\*/

　　unsigned char sin\_zero[8];

};

/\* Internet address. \*/

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

struct in6\_addr {

unsigned char s6\_addr[16]; /\* IPv6 address \*/

};

Unix域对应的是：

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

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。

**2.2.1地址转换**

int\_addr\_t indet\_addr(const char \*cp)

功能：将字符串形式的IP地址转化为整数型的IP地址（网络字节序）

范例：int\_addr.saddr=inet\_addr("192.168.1.1");

char \*inet\_ntoa(struct in\_addr)

功能：将整数形式的IP地址转化为字符串形式的IP地址

**2.2.2网络字节序**

网络字节序定义：收到的第一个字节被当作高位看待，这就要求发送端发送的第一个字节应当是高位。而在发送端发送数据时，发送的第一个字节是该数字在内存中起始地址对应的字节。可见多字节数值在发送前，在内存中数值应该以大端法存放。

网络字节序说是大端字节序。

**小端法(Little-Endian)**就是低位字节排放在内存的低地址端即该值的起始地址，高位字节排放在内存的高地址端。

**大端法(Big-Endian)**就是高位字节排放在内存的低地址端即该值的起始地址，低位字节排放在内存的高地址端。

网络字节序转化：---->不论是数据还是地址只要大于两个字节就必须转换

uint32\_t htonl(uint32\_t hostlong);将32位的数据从主机字节序转换为网络字节序

in\_addr.saddr = htonl(INADDR\_ANY)

uint16\_t htons(uint16\_t hostshort);将16位的数据从主机字节序转换为网络字节序

uint32\_t ntohl(uint32\_t netlong);将32位的数据从网络字节序转换为主机字节序

uint16\_t ntohs(uint16\_t netshort);将16位的数据从网络字节序转换为主机字节序

**2.3、listen()、connect()函数**

如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。成功返回0，若连接失败则返回-1。

**2.4、accept()函数**

TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就向TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); //返回连接connect\_fd

参数sockfd 参数sockfd就是上面解释中的监听套接字，这个套接字用来监听一个端口，当有一个客户与服务器连接时，它使用这个一个端口号，而此时这个端口号正与这个套接字关联。当然客户不知道套接字这些细节，它只知道一个地址和一个端口号。

参数addr　这是一个结果参数，它用来接受一个返回值，这返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为NULL。

参数len 如同大家所认为的，它也是结果的参数，用来接受上述addr的结构的大小的，它指明addr结构所占有的字节个数。同样的，它也可以被设置为NULL。

如果accept成功返回，则服务器与客户已经正确建立连接了，此时服务器通过accept返回的套接字来完成与客户的通信。

注意：accept默认会阻塞进程，直到有一个客户连接建立后返回，它返回的是一个新可用的套接字，这个套接字是连接套接字。

此时我们需要区分两种套接字，

监听套接字: 监听套接字正如accept的参数sockfd，它是监听套接字，在调用listen函数之后，是服务器开始调用socket()函数生成的，称为监听socket描述字(监听套接字)

连接套接字：一个套接字会从主动连接的套接字变身为一个监听套接字；而accept函数返回的是已连接socket描述字(一个连接套接字)，它代表着一个网络已经存在的点点连接。

一个服务器通常通常仅仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

连接套接字socketfd\_new 并没有占用新的端口与客户端通信，依然使用的是与监听套接字socketfd一样的端口号

**2.5、recv()/send()函数**

　　当然也可以使用其他函数来实现数据传送，比如read和write。

2.5.1send函数

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

不论是客户还是服务器应用程序都用send函数来向TCP连接的另一端发送数据。

客户程序一般用send函数向服务器发送请求，而服务器则通常用send函数来向客户程序发送应答。

该函数的第一个参数指定发送端套接字描述符；

第二个参数指明一个存放应用程序要发送数据的缓冲区；

第三个参数指明实际要发送的数据的字节数；

第四个参数一般置0。

这里只描述同步Socket的send函数的执行流程。当调用该函数时，send先比较待发送数据的长度len和套接字s的发送缓冲的 长度，如果len大于s的发送缓冲区的长度，该函数返回SOCKET\_ERROR；如果len小于或者等于s的发送缓冲区的长度，那么send先检查协议是否正在发送s的发送缓冲中的数据，如果是就等待协议把数据发送完，如果协议还没有开始发送s的发送缓冲中的数据或者s的发送缓冲中没有数据，那么 send就比较s的发送缓冲区的剩余空间和len，如果len大于剩余空间大小send就一直等待协议把s的发送缓冲中的数据发送完，如果len小于剩余空间大小send就仅仅把buf中的数据copy到剩余空间里（注意并不是send把s的发送缓冲中的数据传到连接的另一端的，而是协议传的，send仅仅是把buf中的数据copy到s的发送缓冲区的剩余空间里）。如果send函数copy数据成功，就返回实际copy的字节数，如果send在copy数据时出现错误，那么send就返回SOCKET\_ERROR；如果send在等待协议传送数据时网络断开的话，那么send函数也返回SOCKET\_ERROR。

要注意send函数把buf中的数据成功copy到s的发送缓冲的剩余空间里后它就返回了，但是此时这些数据并不一定马上被传到连接的另一端。如果协议在后续的传送过程中出现网络错误的话，那么下一个Socket函数就会返回SOCKET\_ERROR。（每一个除send外的Socket函数在执行的最开始总要先等待套接字的发送缓冲中的数据被协议传送完毕才能继续，如果在等待时出现网络错误，那么该Socket函数就返回 SOCKET\_ERROR）

**2.5.2recv函数学习**

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

不论是客户还是服务器应用程序都用recv函数从TCP连接的另一端接收数据。

该函数的第一个参数指定接收端套接字描述符；

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；

第三个参数指明buf的长度；

第四个参数一般置0。

这里只描述同步Socket的recv函数的执行流程。当应用程序调用recv函数时，recv先等待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，如果s的发送缓冲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，如果s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，只到协议把数据接收完毕。当协议把数据接收完毕，recv函数就把s的接收缓冲中的数据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；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。

2.6、close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

#include <unistd.h>

int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。