[Linux的SOCKET编程详解](http://blog.csdn.net/hguisu/article/details/7445768)

**1. 网络中进程之间如何通信**

进 程通信的概念最初来源于单机系统。由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进

程之间既互不干扰又协调一致工作，[**操作系统**](http://lib.csdn.net/base/operatingsystem)为进程通信提供了相应设施，如

UNIX BSD有：管道（pipe）、命名管道（named pipe）软中断信号（signal）

UNIX system V有：消息（message）、共享存储区（shared memory）和信号量（semaphore)等.

他们都仅限于用在本机进程之间通信。网间进程通信要解决的是不同主机进程间的相互通信问题（可把同机进程通信看作是其中的特例）。为此，首先要解决的是网间进程标识问题。同一主机上，不同进程可用进程号（process ID）唯一标识。但在网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机A赋于某进程号5，在B机中也可以存在5号进程，因此，“5号进程”这句话就没有意义了。 其次，操作系统支持的网络协议众多，不同协议的工作方式不同，地址格式也不同。因此，网间进程通信还要解决多重协议的识别问题。

其实TCP/IP协议族已经帮我们解决了这个问题，网络层的“**ip地址**”可以唯一标识网络中的主机，而传输层的“**协议+端口**”可以唯一标识主机中的应用程序（进程）。这样利用三元组（ip地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互。

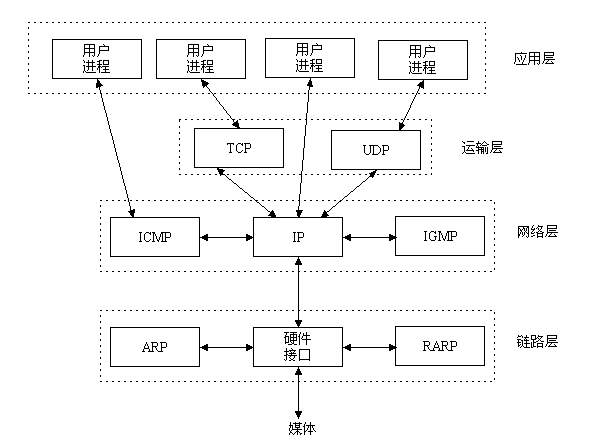
使用TCP/IP协议的应用程序通常采用应用编程接口：UNIX  BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，网络中进程通信是无处不在，这就是我为什么说“一切皆socket”。

**2. 什么是TCP/IP、UDP**

     TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WANs）设计的。

     TCP/IP协议存在于OS中，网络服务通过OS提供，在OS中增加支持TCP/IP的系统调用——Berkeley套接字，如Socket，Connect，Send，Recv等

    UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议。它是属于TCP/IP协议族中的一种。如图：



      TCP/IP协议族包括运输层、网络层、链路层，而socket所在位置如图，Socket是应用层与TCP/IP协议族通信的中间软件抽象层。



**3. Socket是什么**

**1、 socket套接字：**

     socket起源于Unix，而Unix/[**Linux**](http://lib.csdn.net/base/linux)基本哲学之一就是“一切皆文件”，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。

       注意：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。

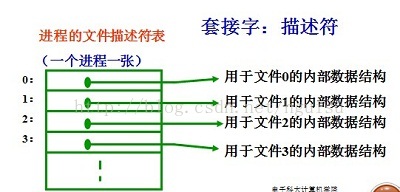
**2、套接字描述符**

          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**](http://lib.csdn.net/base/datastructure)。



     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。

**3、文件描述符和文件指针的区别：**

**文件描述符：**在linux系统中打开文件就会获得文件描述符，它是个很小的正整数。每个进程在PCB（Process Control Block）中保存着一份文件描述符表，文件描述符就是这个表的索引，每个表项都有一个指向已打开文件的指针。

**文件指针：**[**C语言**](http://lib.csdn.net/base/c)中使用文件指针做为I/O的句柄。文件指针指向进程用户区中的一个被称为FILE结构的数据结构。FILE结构包括一个缓冲区和一个文件描述符。而文件描述符是文件描述符表的一个索引，因此从某种意义上说文件指针就是句柄的句柄（在Windows系统上，文件描述符被称作文件句柄）。

详细内容请看[linux文件系统](http://blog.csdn.net/hguisu/article/details/6122513#t7)：<http://blog.csdn.net/hguisu/article/details/6122513#t7>

**4. 基本的SOCKET接口函数**

在生活中，A要电话给B，A拨号，B听到电话铃声后提起电话，这时A和B就建立起了连接，A和B就可以讲话了。等交流结束，挂断电话结束此次交谈。  打电话很简单解释了这工作原理：“open—write/read—close”模式。



    服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，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。

**这些接口的实现都是内核来完成。具体如何实现，可以看看linux的内核**

**4.1、socket()函数**

int **socket**(int protofamily, int type, int protocol);//返回sockfd

     sockfd是描述符。

  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。

      正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

* 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（socket的类型有哪些？）。
* protocol：故名思意，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（这个协议我将会单独开篇讨论！）。

**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。

**4.2、bind()函数**

正如上面所说bind()函数把一个地址族中的特定地址赋给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。

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

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。
* 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，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 \*/
* struct in\_addr sin\_addr; /\* internet address \*/
* };
* /\* 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 \*/

};

* addrlen：对应的是地址的长度。

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。

网络字节序与主机字节序

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。

**4.3、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。

**4.4、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。

        自然要问的是：为什么要有两种套接字？原因很简单，如果使用一个描述字的话，那么它的功能太多，使得使用很不直观，同时在内核确实产生了一个这样的新的描述字。

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

**4.5、read()、write()等函数**

万事具备只欠东风，至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

* read()/write()
* recv()/send()
* readv()/writev()
* recvmsg()/sendmsg()
* recvfrom()/sendto()

我推荐使用recvmsg()/sendmsg()函数，这两个函数是最通用的I/O函数，实际上可以把上面的其它函数都替换成这两个函数。它们的声明如下：

#include <unistd.h>

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

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

#include <sys/types.h>

#include <sys/socket.h>

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

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

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,

const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

read函数是负责从fd中读取内容.当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数。失败时返回-1，并设置errno变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接)。

其它的我就不一一介绍这几对I/O函数了，具体参见man文档或者baidu、Google，下面的例子中将使用到send/recv。

**4.6、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。

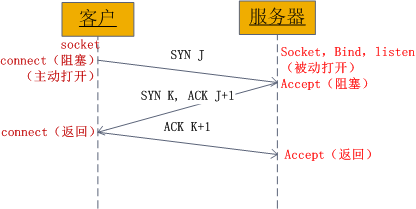
注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。

**5. Socket中TCP的建立（三次握手）**

TCP协议通过三个报文段完成连接的建立，这个过程称为三次握手(three-way handshake)，过程如下图所示。

第一次握手：建立连接时，客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；SYN：同步序列编号(Synchronize Sequence Numbers)。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；  
第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。  
一个完整的三次握手也就是： 请求---应答---再次确认。

对应的函数接口：  
       

从图中可以看出，当客户端调用connect时，触发了连接请求，向服务器发送了SYN J包，这时connect进入阻塞状态；服务器监听到连接请求，即收到SYN J包，调用accept函数接收请求向客户端发送SYN K ，ACK J+1，这时accept进入阻塞状态；客户端收到服务器的SYN K ，ACK J+1之后，这时connect返回，并对SYN K进行确认；服务器收到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

我们可以通过网络抓包的查看具体的流程：

比如我们服务器开启9502的端口。使用tcpdump来抓包：

**tcpdump -iany tcp port 9502**

然后我们使用telnet 127.0.0.1 9502开连接.:

telnet 127.0.0.1 9502

14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378, win 32792, options [mss 16396,sackOK,TS val 255474104 ecr 0,nop,wscale 3], length 0**（1）**  
14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379, win 32768, options [mss 16396,sackOK,TS val 255474104 ecr 255474104,nop,wscale 3], length 0  **（2）**  
14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.], ack 1, win 4099, options [nop,nop,TS val 255474104 ecr 255474104], length 0  **（3）**

14:13:01.415407 IP localhost.39870 > localhost.9502: Flags [P.], seq 1:8, ack 1, win 4099, options [nop,nop,TS val 255478182 ecr 255474104], length 7  
14:13:01.415432 IP localhost.9502 > localhost.39870: Flags [.], ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 0  
14:13:01.415747 IP localhost.9502 > localhost.39870: Flags [P.], seq 1:19, ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 18  
14:13:01.415757 IP localhost.39870 > localhost.9502: Flags [.], ack 19, win 4097, options [nop,nop,TS val 255478182 ecr 255478182], length 0

* 114:12:45.104687 时间带有精确到微妙
* localhost.39870 > localhost.9502 表示通信的流向，39870是客户端，9502是服务器端
* [S] 表示这是一个SYN请求
* [S.] 表示这是一个SYN+ACK确认包:
* [.] 表示这是一个ACT确认包， (client)SYN->(server)SYN->(client)ACT 就是3次握手过程
* [P] 表示这个是一个数据推送，可以是从服务器端向客户端推送，也可以从客户端向服务器端推
* [F] 表示这是一个FIN包，是关闭连接操作，client/server都有可能发起
* [R] 表示这是一个RST包，与F包作用相同，但RST表示连接关闭时，仍然有数据未被处理。可以理解为是强制切断连接
* win 4099 是指滑动窗口大小
* length 18指数据包的大小

我们看到 **（1）（2）（3）三步是建立tcp：**

**第一次握手：**

14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378

客户端IP localhost.39870 (客户端的端口一般是自动分配的) 向服务器localhost.9502 发送syn包(syn=j)到服务器》

syn包(syn=j) ： syn的seq= 2927179378  （j=2927179378）

**第二次握手：**

14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379,

收到请求并确认：服务器收到syn包，并必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包：  
此时服务器主机自己的SYN：seq：y= syn seq 1721825043。  
ACK为j+1 =（ack=j+1）=ack 2927179379 

**第三次握手：**

14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.], ack 1,

客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1)

客户端和服务器进入ESTABLISHED状态后，可以进行通信数据交互。此时和accept接口没有关系，即使没有accepte，也进行3次握手完成。

连接出现连接不上的问题，一般是网路出现问题或者网卡超负荷或者是连接数已经满啦。

紫色背景的部分：

IP localhost.39870 > localhost.9502: Flags [P.], seq 1:8, ack 1, win 4099, options [nop,nop,TS val 255478182 ecr 255474104], length 7

客户端向服务器发送长度为7个字节的数据，

IP localhost.9502 > localhost.39870: Flags [.], ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 0

服务器向客户确认已经收到数据

 IP localhost.9502 > localhost.39870: Flags [P.], seq 1:19, ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 18

然后服务器同时向客户端写入数据。

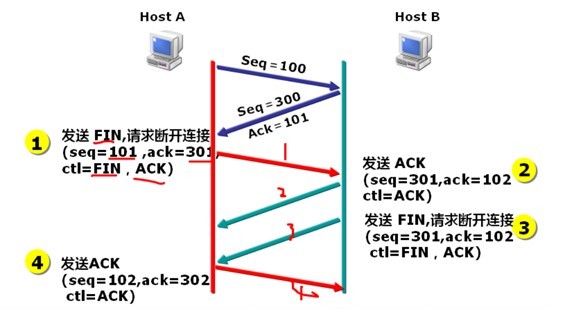
 IP localhost.39870 > localhost.9502: Flags [.], ack 19, win 4097, options [nop,nop,TS val 255478182 ecr 255478182], length 0

客户端向服务器确认已经收到数据

这个就是tcp可靠的连接，每次通信都需要对方来确认。

**6. TCP连接的终止（四次握手释放）**

建立一个连接需要三次握手，而终止一个连接要经过四次握手，这是由TCP的半关闭(half-close)造成的，如图：



由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

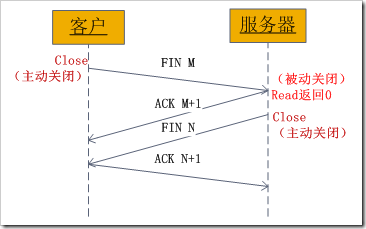
（1）客户端A发送一个FIN，用来关闭客户A到服务器B的数据传送（报文段4）。

（2）服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1（报文段5）。和SYN一样，一个FIN将占用一个序号。

（3）服务器B关闭与客户端A的连接，发送一个FIN给客户端A（报文段6）。

（4）客户端A发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

对应函数接口如图：



过程如下：

* 某个应用进程首先调用close主动关闭连接，这时TCP发送一个FIN M；
* 另一端接收到FIN M之后，执行被动关闭，对这个FIN进行确认。它的接收也作为文件结束符传递给应用进程，因为FIN的接收意味着应用进程在相应的连接上再也接收不到额外数据；
* 一段时间之后，接收到文件结束符的应用进程调用close关闭它的socket。这导致它的TCP也发送一个FIN N；
* 接收到这个FIN的源发送端TCP对它进行确认。

这样每个方向上都有一个FIN和ACK。

1．为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。

2．为什么TIME\_WAIT状态还需要等2MSL后才能返回到CLOSED状态？

这是因为虽然双方都同意关闭连接了，而且握手的4个报文也都协调和发送完毕，按理可以直接回到CLOSED状态（就好比从SYN\_SEND状态到ESTABLISH状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的ACK报文会一定被对方收到，因此对方处于LAST\_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报文，所以这个TIME\_WAIT状态的作用就是用来重发可能丢失的ACK报文。

**7. Socket编程实例**

服务器端：一直监听本机的8000号端口，如果收到连接请求，将接收请求并接收客户端发来的消息，并向客户端返回消息。

**[cpp]** [view plain](http://blog.csdn.net/hguisu/article/details/7445768/) [copy](http://blog.csdn.net/hguisu/article/details/7445768/)

[print?](http://blog.csdn.net/hguisu/article/details/7445768/)[在CODE上查看代码片](https://code.csdn.net/snippets/441451)

1. /\* File Name: server.c \*/
2. #include<stdio.h>
3. #include<stdlib.h>
4. #include<string.h>
5. #include<errno.h>
6. #include<sys/types.h>
7. #include<sys/socket.h>
8. #include<netinet/in.h>
9. #define DEFAULT\_PORT 8000
10. #define MAXLINE 4096
11. **int** main(**int** argc, **char**\*\* argv)
12. {
13. **int**    socket\_fd, connect\_fd;
14. **struct** sockaddr\_in     servaddr;
15. **char**    buff[4096];
16. **int**     n;
17. //初始化Socket
18. **if**( (socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1 ){
19. printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
20. exit(0);
21. }
22. //初始化
23. memset(&servaddr, 0, **sizeof**(servaddr));
24. servaddr.sin\_family = AF\_INET;
25. servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);//IP地址设置成INADDR\_ANY,让系统自动获取本机的IP地址。
26. servaddr.sin\_port = htons(DEFAULT\_PORT);//设置的端口为DEFAULT\_PORT
28. //将本地地址绑定到所创建的套接字上
29. **if**( bind(socket\_fd, (**struct** sockaddr\*)&servaddr, **sizeof**(servaddr)) == -1){
30. printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
31. exit(0);
32. }
33. //开始监听是否有客户端连接
34. **if**( listen(socket\_fd, 10) == -1){
35. printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
36. exit(0);
37. }
38. printf("======waiting for client's request======\n");
39. **while**(1){
40. //阻塞直到有客户端连接，不然多浪费CPU资源。
41. **if**( (connect\_fd = accept(socket\_fd, (**struct** sockaddr\*)NULL, NULL)) == -1){
42. printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
43. **continue**;
44. }
45. //接受客户端传过来的数据
46. n = recv(connect\_fd, buff, MAXLINE, 0);
47. //向客户端发送回应数据
48. **if**(!fork()){ /\*紫禁城\*/
49. **if**(send(connect\_fd, "Hello,you are connected!\n", 26,0) == -1)
50. perror("send error");
51. close(connect\_fd);
52. exit(0);
53. }
54. buff[n] = '\0';
55. printf("recv msg from client: %s\n", buff);
56. close(connect\_fd);
57. }
58. close(socket\_fd);
59. }

客户端：

**[cpp]** [view plain](http://blog.csdn.net/hguisu/article/details/7445768/) [copy](http://blog.csdn.net/hguisu/article/details/7445768/)

[print?](http://blog.csdn.net/hguisu/article/details/7445768/)[在CODE上查看代码片](https://code.csdn.net/snippets/441451)

1. /\* File Name: client.c \*/
3. #include<stdio.h>
4. #include<stdlib.h>
5. #include<string.h>
6. #include<errno.h>
7. #include<sys/types.h>
8. #include<sys/socket.h>
9. #include<netinet/in.h>
11. #define MAXLINE 4096

14. **int** main(**int** argc, **char**\*\* argv)
15. {
16. **int**    sockfd, n,rec\_len;
17. **char**    recvline[4096], sendline[4096];
18. **char**    buf[MAXLINE];
19. **struct** sockaddr\_in    servaddr;

22. **if**( argc != 2){
23. printf("usage: ./client <ipaddress>\n");
24. exit(0);
25. }

28. **if**( (sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0){
29. printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
30. exit(0);
31. }

34. memset(&servaddr, 0, **sizeof**(servaddr));
35. servaddr.sin\_family = AF\_INET;
36. servaddr.sin\_port = htons(8000);
37. **if**( inet\_pton(AF\_INET, argv[1], &servaddr.sin\_addr) <= 0){
38. printf("inet\_pton error for %s\n",argv[1]);
39. exit(0);
40. }

43. **if**( connect(sockfd, (**struct** sockaddr\*)&servaddr, **sizeof**(servaddr)) < 0){
44. printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
45. exit(0);
46. }

49. printf("send msg to server: \n");
50. fgets(sendline, 4096, stdin);
51. **if**( send(sockfd, sendline, strlen(sendline), 0) < 0)
52. {
53. printf("send msg error: %s(errno: %d)\n", strerror(errno), errno);
54. exit(0);
55. }
56. **if**((rec\_len = recv(sockfd, buf, MAXLINE,0)) == -1) {
57. perror("recv error");
58. exit(1);
59. }
60. buf[rec\_len]  = '\0';
61. printf("Received : %s ",buf);
62. close(sockfd);
63. exit(0);
64. }

inet\_pton 是Linux下IP地址转换函数，可以在将IP地址在“点分十进制”和“整数”之间转换 ，是inet\_addr的扩展。

**[cpp]** [view plain](http://blog.csdn.net/hguisu/article/details/7445768/) [copy](http://blog.csdn.net/hguisu/article/details/7445768/)

[print?](http://blog.csdn.net/hguisu/article/details/7445768/)[在CODE上查看代码片](https://code.csdn.net/snippets/441451)

1. **int** inet\_pton(**int** af, **const** **char** \*src, **void** \*dst);//转换字符串到网络地址:

第一个参数af是地址族，转换后存在dst中  
    af = AF\_INET:src为指向字符型的地址，即ASCII的地址的首地址（ddd.ddd.ddd.ddd格式的），函数将该地址转换为in\_addr的结构体，并复制在\*dst中  
　　af =AF\_INET6:src为指向IPV6的地址，函数将该地址转换为in6\_addr的结构体，并复制在\*dst中  
如果函数出错将返回一个负值，并将errno设置为EAFNOSUPPORT，如果参数af指定的地址族和src格式不对，函数将返回0。

[**测试**](http://lib.csdn.net/base/softwaretest)：

编译server.c

gcc -o server server.c

启动进程：

./server

显示结果：

======waiting for client's request======

并等待客户端连接。

编译 client.c

gcc -o client server.c

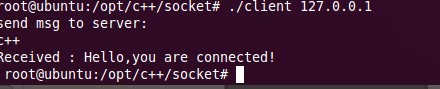
客户端去连接server：

./client 127.0.0.1

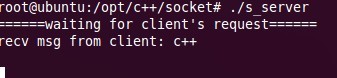
等待输入消息

http://my.csdn.net/uploads/201204/10/1334050427_5715.jpg

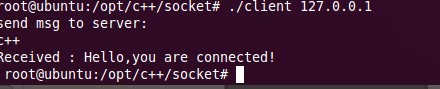
发送一条消息，输入：c++



此时服务器端看到：



客户端收到消息：



其实可以不用client,可以使用telnet来测试：

telnet 127.0.0.1 8000

