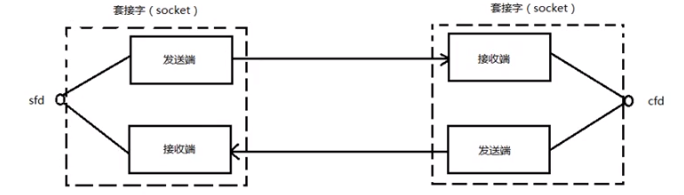
# Socket通信



Linux中的网络通过socket接口实现，socket即是一种特殊的IO，它也是一种文件描述符。一个完整的socket都有一个相关描述(协议，本地地址，本地端口，远程地址，远程端口)；每个socket有一个本地的唯一socket号，由操作系统分配。

注：**通信中的套接字成对出现**。一端的发送缓冲区对应对端的接收缓冲区，我们使用同一文件描述符发送缓冲区和接收缓冲区。

Socket是一种IPC方法，允许位于统一主机或者使用网络连接起来的不同主机上的应用程序之间交换数据。

Socket使用标准文件描述符（file description）和其他程序进行通讯。

## socket与HTTP

HTTP 协议：超文本传输协议，对应于应用层，用于如何封装数据。

TCP/UDP 协议：传输控制协议，对应于传输层，主要解决数据在网络中的传输。

IP 协议：对应于网络层，同样解决数据在网络中的传输。

传输数据的时候只使用 TCP/IP 协议(传输层)，如果没有应用层来识别数据内容，传输后的协议都是无用的。

应用层协议很多 FTP，HTTP，TELNET等，可以自己定义应用层协议。

web 使用 HTTP 作传输层协议，以封装 HTTP 文本信息，然后使用 TCP/IP 做传输层协议，将数据发送到网络上。

http为短连接：客户端发送请求都需要服务器端回送响应。请求结束后，主动释放链接，因此为短连接。通常的做法是，不需要任何数据，也要保持每隔一段时间向服务器发送"保持连接"的请求。这样可以保证客户端在服务器端是"上线"状态。

HTTP连接使用的是"请求-响应"方式，不仅在请求时建立连接，而且客户端向服务器端请求后，服务器才返回数据。

**socket 连接**

TCP 三次握手：握手过程中并不传输数据，在握手后服务器与客户端才开始传输数据，理想状态下，TCP 连接一旦建立，在通讯双方中的任何一方主动断开连接之前 TCP 连接会一直保持下去。

**Socket是对TCP/IP协议的封装，Socket只是个接口不是协议**，通过Socket我们才能使用TCP/IP协议，除了TCP，也可以使用UDP协议来传递数据。

创建 Socket 连接的时候，可以指定传输层协议，可以是 TCP 或者 UDP，当用 TCP 连接，该Socket就是个TCP连接，反之。

socket 连接至少需要一对套接字，分为 clientSocket，serverSocket 连接分为3个步骤：

(1) 服务器监听：服务器并不定位具体客户端的套接字，而是时刻处于监听状态；

(2) 客户端请求：客户端的套接字要描述它要连接的服务器的套接字，提供地址和端口号，然后向服务器套接字提出连接请求；

(3) 连接确认：当服务器套接字收到客户端套接字发来的请求后，就响应客户端套接字的请求，并建立一个新的线程,把服务器端的套接字的描述发给客户端。一旦客户端确认了此描述，就正式建立连接。而服务器套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

Socket为长连接：通常情况下Socket 连接就是 TCP 连接，因此 Socket 连接一旦建立，通讯双方开始互发数据内容，直到双方断开连接。在实际应用中，由于网络节点过多，在传输过程中，会被节点断开连接，因此要通过轮询高速网络，该节点处于活跃状态。

很多情况下，都是需要服务器端向客户端主动推送数据，保持客户端与服务端的实时同步。

若双方是 Socket 连接，可以由服务器直接向客户端发送数据。

若双方是 HTTP 连接，则服务器需要等客户端发送请求后，才能将数据回传给客户端。

因此，客户端定时向服务器端发送请求，不仅可以保持在线，同时也询问服务器是否有新数据，如果有就将数据传给客户端。

## 通信domain

Socket存在于一个通信domain中，它确定：

1、识别出一个socket的方法（即socket地址的格式）；

2、通信范围（即是在位于同一主机上的应用程序之间还是位于同一网络连接起来的主机应用程序之间）。

现在操作系统至少支持一下domain：

UNIX(AF\_UNIX)：domain允许在同一主机上的应用程序之间进行通信。

IPv4(AF\_INET)：domain允许在使用IOv4网络连接起来的主机上应用程序之间通信。

IPv6(AF\_INET6)：domain允许在使用IPv6网络连接起来主机上应用程序之间通信。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Domain | 执行通信 | 应用间通信 | 地址格式 | 地址结构 |
| AF\_UNIX | 内核中 | 同一主机 | 路径名 | sockaddr\_un |
| AF\_INET | 通过IPv4 | IPv4网络主机 | 32位IPv4地址+16位端口号 | sockaddr\_in |
| AF\_INET6 | 通过IPv6 | IPv6网络主机 | 128位IPv4地址+16位端口号 | sockaddr\_un6 |

说明：AF=Address Family，即地址族

## Socket类型

每个socket实现至少提供两种socket：流和数据报。

### 流式套接字

流socket（SOCK\_STREAM）：提供了一个可靠的双向的字节流通信信道。含义如下：

1）可靠的：表示可以保证发送者传输的数据会完整地送达接收应用程序或者收到一个传输失败的的通知；

2）双向：表示数据可以在两个socket之间的任意方向上传输；

3）字节流：表示与管道一样不存在消息边界的概念。

Telenet使用的是流式套接字，你输入的字符需要按照顺序到达：www浏览器使用的HTTP协议也是用它们来下载页面，实际上当你通过80telnet一个www站点，然后输入“GET pagename”的时候，也可以得到HTML的内容。

流式套接字之所以能够达到高质量的数据传输，是因为使用了“传输控制协议TCP”，TCP控制你的数据按照顺序到达没有错误。

### 数据报套接字

数据报socket（SOCK\_DGRAM）允许数据以被称为数据报的消息的形式进行交换。在数据报socket中，消息边界得到了保留，但是数据传输是不可靠的。消息的达到可能是无序的、重复的或者根本无法到达。

注：数据报套接字有时候也叫“无连接套接字”。

数据报也是使用IP做路由，但是不使用TCP。它是无连接的，只需要建立一个包，构造一个目标信息的IP头，然后发出去，无需建立连接。

数据报套接字的应用程序有tftp、bootp等。

**总结：**

在Internet domain中，数据报socket使用了用户数据报协议（UDP），而流socket则（通常）使用了传输控制协议（TCP），分别使用术语“UDP socket”和“TCP socket”。

## 字节序

有两种字节排列顺序：重要的字节（有时候叫octet，即8位位组）在前面，或者不重要的字节在前面（即我们所说的高8位低8位）。

前一种叫“网络字节顺序（Network Byte Order，NBO）”，有些机器在内部是按照这个顺序存储数据的。当某数据必须按照NBO顺序，那么要调用函数转换过来。如果没有提到NBO，那么就让它保持本机字节顺序。

目前CPU的累加器一次能够装在4个字节，那么这4个字节在内存中的排列顺序影响累加器累加的值，这就是字节序的问题。目前，PC都采用小端字节序。但是，格式化的数据在两台不同字节序的主机之间传递时，接收端必然会错误地解释。解决的办法就是发送端总是将字节序转换为大端字节序，然后发送，接收端知道传过来的是大端字节序，因此接收端总是能正确地接收数据。

### 概述

字节序指的是多字节的数据各字节的存储顺序。在几乎所有计算机中，多字节数据被存储为连续的字节序列。例如，一个4字节的int类型变量a，其存储的起始地址为0x804900，那么a的四个字节将被分别存在0x804900，0x804901，0x804902，0x804903的位置。但是问题来了，a的最低有效位可以存储在最前面，也可以存储在高最后面，就有两种不同的存储顺序。这就引出了大端序和小端序。

### 大小端

参考：

<https://www.cnblogs.com/Alandre/p/4878841.html>

实际上，如果最低有效位在最高有效位的前面，则该存储规则为小端序；反之，如果最低有效位在最高有效的后面，则该存储规则为大端序。不同的处理器体系，采用的字节序可能是不同的。例如，x86采用小端序，而PowerPc 970等采用大端序。那么如此一来，不同机器之间的数据传输是不是会出问题呢？

**本地序和网络序**

**本地序（也称主机序）**即指前面处理器本身所采用的字节序，因此有的大端序，有的小端序。而网络序，是指网络传输采用的字节序。所幸，网络序是标准化的，即一般统一采用大端序。因此，发送网络数据之前需要将数据转换为网络序，从而避免了前面所担心的问题。而C语言也针对整型数据提供了一组接口，htonl、htons用于本地序转网络序，以及ntohl、ntohs用于网络序转本地序。

我们的**X86 CPU都是采用小端模式，网络传输的都是大端，二者需要调用系统函数转换**。

比如我们输入0x1234（12是高字节，34是低字节），大端和小端的输出结果分别是：

小端：34 12（低地址存储在低位）

大端：12 34（高地址存储低位地址，最直观）

### 必要性

既然每次都发送网络数据之前都要转换，为什么不统一字节序呢？实际上，大小端各有优劣：

计算都是从低位开始的，因此计算机内部处理采用小端序，效率较高。

而大端序存储的时候，由于符号位在高位，因此对于数据征服或大小的判断也就方便许多。另外，大端序也更符合人类的阅读习惯。

再由于各个芯片厂商的坚持，字节序的问题也就一直没有统一。大小端争端起源于吃鸡蛋时先打破大端还是小端，有兴趣的读者可以搜索一下。

因为每个机器内部对变量的字节存储顺序不同(有的系统高位在前，低位在后，而

有的系统是低位在前，高位在后)，而网络传输的数据大家是一定要统一顺序的。

所以对于内部字节表示顺序和网络字节顺序不同的机器，就一定要对数据进行转换。

### 函数

#include<netinet/in.h>

//主机字节转换成网络字节

unsigned short int htons(unsigned short int hostshrot);

//网络字节转换成主机字节

unsigned short int ntohs(unsigned short int netshrot);

htons()用来将当前主机字节序转为网络字节序，其中h代表主机（host）字节序，n代表网络（network）字节序，s代表short，htos是h、to、n、s的组合，可以理解为“将short型数据从当前主机字节序转换为网络字节序”。

常见的网络字节转换函数有：

htons()：host to network short，将short类型数据从主机字节序转换为网路字节序；

htonl()：host to network long，将long类型数据从主机字节序转换为网络字节序；

ntohs()：network to host short，将short类型数据从网络字节序转换为主机字节序；

nohl()：network to host long，将long类型数据从网络字节序转换为主机字节序。

通常，以s后缀的函数中，s代表2字节short，因此用于端口号转换；以l为后缀的函数，l代表4字节的long，因此用于IP地址转换。

典型应用：

//创建sockadd\_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\_famliy= AF\_INET; //使用IPV4地址

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

serv\_addr.sin\_port = htons(8080); //端口号

### 总结

在网络应用中，字节序的问题不可忽略，否则可能出现无法预知的问题（如果两台机器本地序相同，且都不做字节序转换，那么侥幸不会出现什么问题）。通过前面的介绍和分析，我们总结出以下几点：

不同处理器之间采用的字节序可能不同。

有些处理器的字节序是确定的，有些处理器的字节序是可配置的。

网络序一般统一为大端序。

数据从本地传输到网络，需要转换为网络序，接收到的网络数据需要转换为本地序后使用。

C提供了一组接口用于整型数据在本地序和网络序之间的转换。

多字节数据对象才需要转字节序，例如int，short等，而char不需要。

由于处理器是按照IEEE标准处理float和double的（参考：对浮点数的一些理解），因此也不需要转字节序。

由于Java虚拟机的存在，Java不需要考虑大小端的问题。

## IP转换函数

专用socket地址结构体：

//TCP/IP协议族专用socket地址结构体

struct sockaddr\_in

{

sa\_family\_t sin\_family; //地址族：AF\_INET

u\_int16\_t sin\_port; //端口号，用于网络字节序表示

struct in\_addr sin\_addr; //IPv4地址结构体，是u\_int32\_t s\_addr

};

IP地址转换为长整型：

inet\_addr()返回的地址已经是网络字节格式，所以无需调用函数htonl()。

长整型转为IP地址：

inet\_ntoa() [ntoa=network to asci]

#include <arpa/inet.h>

//将字符串IP转换为整数IP

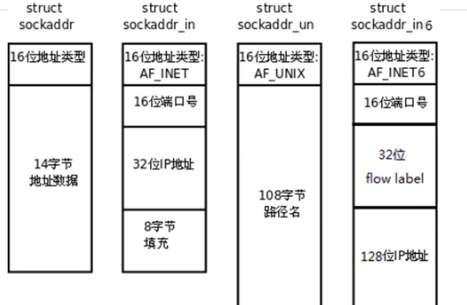
int inet\_pton(int af, const char\* src,void\* dst);

//将整数IP转换成字符串IP

const char\* inet\_ntop(int af, const void\* dst,socklen\_t cnt);

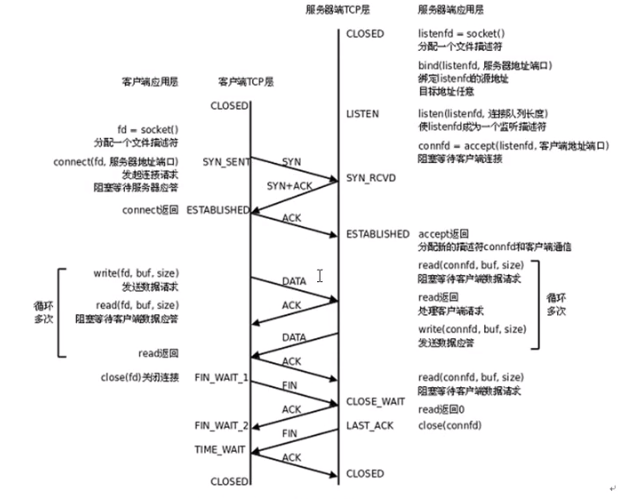
## sockaddr结构体

IP地址以及端口等相关的socket通信必须的信息需要经过必要的转换操作后绑定到结构体sockaddr上：



注：原来的sockaddr已经废弃，IPV4使用结构体sockaddr\_in，IPV6使用结构体sockaddr\_in6。

# 网络编程基础API



参考：

<https://blog.csdn.net/m0_37947204/article/details/80489431>

**Socket编程的send() recv() accept() socket()函数：**

send函数用来向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，send的作用是将要发送的数据拷贝到缓冲区，协议负责传输。

recv函数用来从TCP连接的另一端接收数据，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，然后从缓冲区中读取接收到的内容给应用层。

accept函数用了接收一个连接，内核维护了半连接队列和一个已完成连接队列，当队列为空的时候，accept函数阻塞，不为空的时候accept函数从上边取下来一个已完成连接，返回一个文件描述符。

## 创建socket

#include<sys/types>

#include<sys/socket.h>

**原型：**int socket(int domain, int byte, int protocol);

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

**参数：**domain——应该设置为AF\_INET

type——告诉内核socket类型，SOCK\_STREAM/SOCK\_DGRAM

protocol——设置为0

**返回值：**返回以后再调用中可能用到的socket描述符，或者在错误的时候返回-1，全局变量errno中将存储返回的错误值。

socket域:

AF\_INET IPV4英特网域

AF\_INET6 IPV6英特网域

AF\_UNIX unix域

AF\_UNSPEC 未指定

## 端口复用setsockopt

在服务端的TCP连接没有完全断开之前不允许监听是不合理的。因为，TCP连接没有完全断开指的是connfd（127.0.0.1:6666）没有完全断开，而我们重新监听的是listenfd（0.0.0.0:6666），虽然是占用同一个端口，但是IP地址不同，connfd对应的是与某个客户端通讯的一个具体的IP地址，而listenfd对应的是wildcard address。

解决这个问题的方法是使用setsockopt设置socket描述符的选项SO\_REUSERADD为1，表示**允许创建端口号相同但是IP地址不同的多个socket描述符**。

## 绑定bind

**原型：**bind(int sockfd, struct sockaddr \*my\_addr, int addlen);

int bind(int sockfd, const struct sockeaddr\* addr,socketlen\_t addrlen);

**作用：**命名socket，服务器绑定网络地址

**参数：**

sockfd——调用socket返回的文件描述符

my\_addr——指向数据结构struct sockaddr的指针，保存地址（即IP和端口）信息

addlen——设置为sizeof(struct sockaddr)

**返回值：**

bind函数成功返回0，否则为-1，常见error类型：

EACCES：被绑定的地址收到保护

EADDRINUSE：被绑定的地址正在使用中

对于服务器程序而言，创建的套接字必须绑定到一个本地的IP地址和端口号以用于TCP通信。Linux的bind()函数就是用于完成该工作。

在将套接字绑定到一个地址和端口号之后，服务器程序必须准备好接受来自远程客户端的连接请求。这是一个分为两步的过程：listen()和accept()。

注：bind()用于将套接字和机器上的某一个端口建立绑定关系，如果你想用listen()来侦听某个端口的数据，这是必要的操作，如果只想用connect()这一步不是必要的。使用connect与远程主机通信时，不需要关系本地的端口号，只需要简单调用connect就可以了，它会检查套接字是否绑定端口，如果没有，它会自己绑定一个没有使用的本地端口。

在调用bind的时候，不要采用小于1024的端口号，所有小于1024的端口号都被系统保留，可以选择1024到65535的端口（如果没有别的程序使用的话）。

## 监听listen

**原型：**int listen(int sockfd, int backlog);

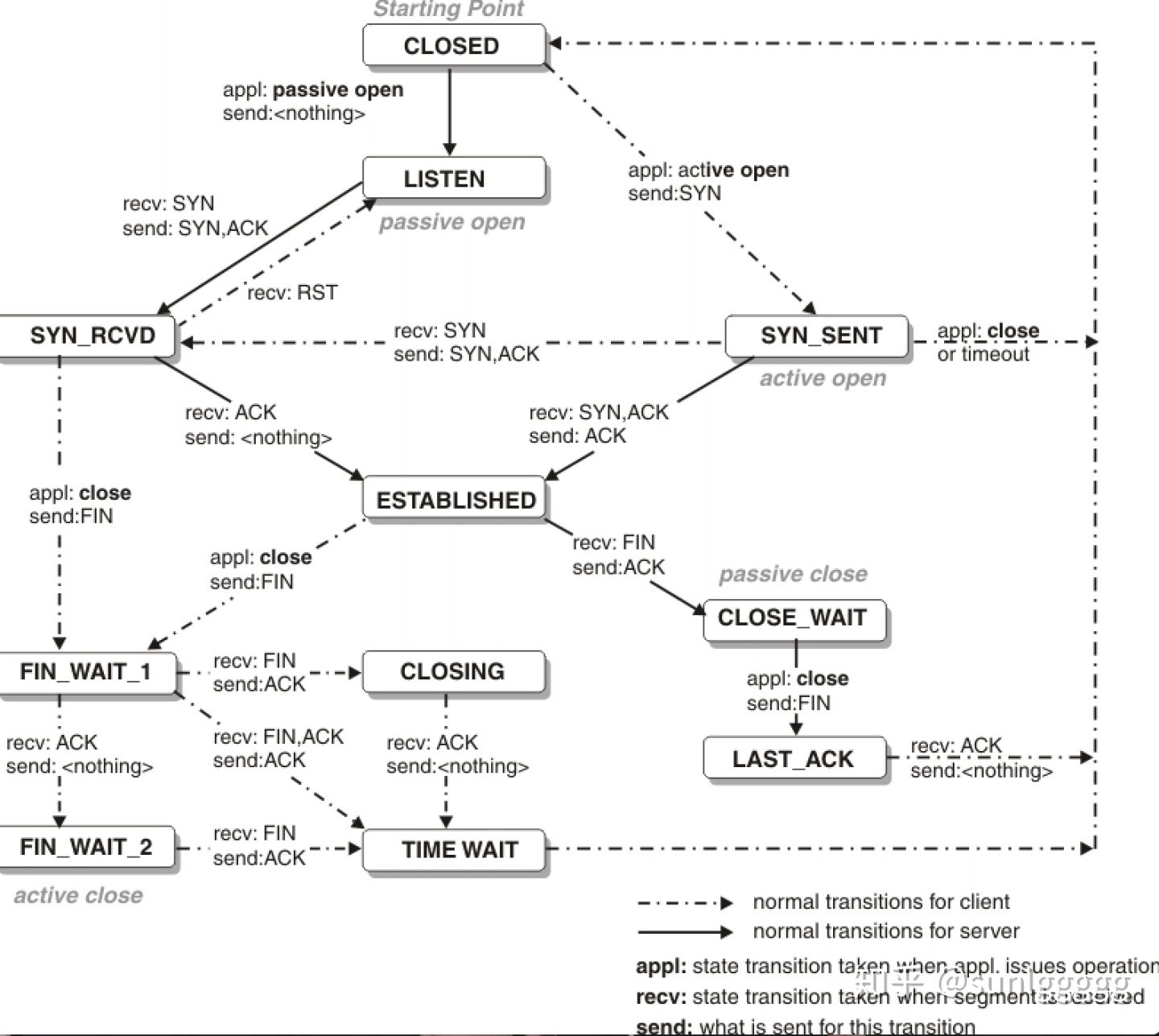
**功能：**设置同时与服务器建立连接的上限数（即同时进行3次握手的客户端数量）。

**参数：**

backlog——在进入队列中允许的连接数目（最大值128）。

backlog: an accumulation of uncompleted work or matters needing to be dealt with. （累计未完成的待处理事件）

程序调用listen()以开始监听客户端的连接，然后调用accept来真正地接受客户端的请求。



根据上述的状态图中的黑线，我们知道处于LISTEN状态的服务端socket收到SYN，建立一个新的状态为SYN\_REVD的连接。SYN\_REVD是一种半连接状态，而当收到客户端的ACK时，则进入ESTABLISHED状态。由于三次握手的原因，连接的发起过程中，服务器的连接状态要经过SYN\_REVD和ESTABLISHED两个状态。如何保存这两种状态的连接，不同系统有不同的实现。

在FressBSD中backlog就是描述状态为SYN\_REVD和ESTABLISHED的所有连接最大数量。

而Linux系统中，则使用两个队列syn queue, accept queue分别存储状态为SYN\_REVD和ESTABLISHED的连接，并且在linux2.2及以后，backlog表示accept queue的大小，而syn queue大小由 /proc/sys/net/ipv4/tcp\_max\_syn\_backlog配置。

最后通过系统调用 accept 去从accept queue中获取建立完成的连接。在开发服务端程序时，我们可以根据CPU性能和处理的复杂程度选择backlog来控制accept queue的大小。而syn queue更多时候取决于网络的流量。例如网络流量较大，但是处理较快可以减小syn queue。而流量较小时可以增大syn queue以接受更多的连接。

**内核参数somaxconn**

全称：socket max connections ，位置：/proc/sys/net/core/somaxconn 这是系统层面对于backlog的控制，实际上accept queue的大小 = min(somaxconn, backlog)。因此在listen这个系统调用层面，backlog最终还是受限于somaxconn。

如果backlog设置太小，在大并发情况下会导致响应慢：

<https://blog.csdn.net/yazhouren/article/details/46713145>

**查看队列**

ss -l

netid State Recv-Q Send-Q Local Address:Port Peer Address:Port

tcp LISTEN 0 128 [::]:ssh [::]:\*

在LISTEN状态下Recv-Q表示当前accept queue中的已连接数。Send-Q表示总大小。

注：**因为没有客户端连接的时候listen并不会阻塞，因此listen的作用不是执行监听操作，而是设置监听上限（同时与服务器socket通信的客户端连接数），真正监听是accpet完成的**。

## 接收连接accept

### accept

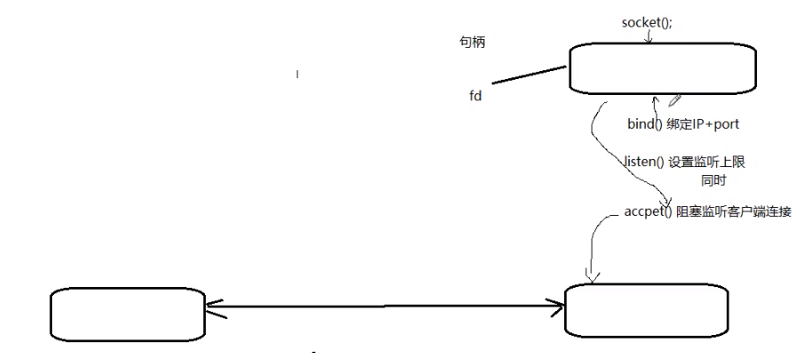
**原型：**int accept(int sockfd, struct sockaddr \*addr, int addlen);

**参数：**

**sockfd：socket函数返回值**

addr：出参，成功与服务器建立连接的那个客户端的IP+port地址结构（与bind的不一样，bind是入参且为const类型）

**功能：**在listen()函数调用之后，accept()函数必须被调用以等待进入的连接。accept()函数是一个**阻塞式**函数。程序的执行将暂停在accept()函数调用上直到有一个客户端的连接请求到达。



当一个连接请求被接受后，**accept()函数将返回一个新的套接字描述符**。这个新的套接字用于与远程客户通信，这样就有两个套接字了，原来的一个还在侦听你绑定的端口，新的正在准备发送send()和recv()数据。由socket()函数创建的原来的套接字仍然可以被用来继续监听更多的客户端连接。

注：**一个客户端与服务端通信，这样就是需要3个socket，而不是两个，accept返回新的socket用于与客户端通信，原来的则继续监听客户端**！

### 连接过程

#### 三次握手过程中内核的处理流程



两个队列满了后，新的请求到达了后的处理过程：

1. 若SYN队列满，则会直接丢弃请求；
2. 如果ACCEPT队列满，则不会导致放弃连接，也不会把连接从SYN队列中移出

#### 阻塞模式接收过程



#### 非阻塞模式接收过程



### 惊群

## 发起连接connect



### connect

**原型：**int connect(int sockfd, struct sockaddr \*serv\_addr, int addlen);

int connect(int sockfd,const struct sockaddr\* serv\_addr,socklen\_t

addrlen);

**参数：**

sockfd——系统调用socket()返回的套接字文件描述符

serv\_addr——保存着目的端口和IP地址的数据结构

addlen——设置为sizeof(struct sockaddr)

**返回值：**

connect函数成功返回0，否则为-1，常见error类型：

ECONNEFUSED：目标端口不存在

ETIMEOUT：连接超时

**UDP的connect函数：**

除非套接字已连接，否则异步错误是不会反悔到UDP套接字的。我们确实可以给UDP套接字调用connect，然而这样做的结果却与TCP连接不同的是没有三路握手过程。内核只是检查是否存在立即可知的错误，记录对端的IP地址和端口号，然后立即返回调用进程。

对于已连接UDP套接字，与默认的未连接UDP套接字相比，发生了三个变化。

其实一旦UDP套接字调用了connect系统调用，那么这个UDP上的连接就变成一对一的连接，但是通过这个UDP连接传输数据的性质还是不变的，仍然是不可靠的UDP连接。一旦变成一对一的连接，在调用系统调用发送和接受数据时也就可以使用TCP那一套系统调用了。

1、我们再也不能给输出操作指定目的IP地址和端口号。也就是说，我们不使用sendto，而改用write或send。写到已连接UDP套接字上的任何内容都自动发送到由connect指定的协议地址。可以给已连接的UDP套接字调用sendto，但是不能指定目的地址。sendto的第五个参数必须为空指针，第六个参数应该为0.

2、不必使用recvfrom以获悉数据报的发送者，而改用read、recv或recvmsg。在一个已连接UDP套接字上，由内核为输入操作返回的数据报只有那些来自connect指定协议地址的数据报。这样就限制一个已连接UDP套接字能且仅能与一个对端交换数据报。

3、由已连接UDP套接字引发的异步错误会返回给它们所在的进程，而未连接的UDP套接字不接收任何异步错误。

来自任何其他IP地址或断开的数据报不投递给这个已连接套接字，因为它们要么源IP地址要么源UDP端口不与该套接字connect到的协议地址相匹配。

UDP客户进程或服务器进程只在使用自己的UDP套接字与确定的唯一对端进行通信时，才可以调用connect。调用connect的通常是UDP客户，不过有些网络应用中的UDP服务器会与单个客户长时间通信TFTP，这种情况下，客户和服务器都可能调用connect。

### 发送过程

内核处理消息发送过程：



## 数据读写recv/send

### recv/send

对文件的读写，系统调用read、write同样适用于socket，但是这里提供了专用于socket的recv和send。

TCP数据读写API：

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

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

recv函数和send函数的flag参数选项：

MSG\_WAITALL：读操作读取到一定数量字节后返回

MSG\_PEEK：查看读缓冲找那个的数据而不取出

数据报套接字发送和接收函数：send()和recv()。

一旦一个连接请求被接受，服务器就可以使用新的套接字描述符向客户端发送数据或接收来自客户端的数据，send()函数不会阻塞程序的执行。缓冲区中的数据被发送到系统中底层的TCP发送缓冲区，然后函数调用将返回。有可能在send()函数中定义的缓冲区中的数据不会全部被发送出去。Send()函数调用返回的整数值将表示有多少字节的数据已被发送到TCP发送缓冲区。确认这个返回值匹配缓冲区的大小以确保所有的数据都已被发送出去是非常重要的。

recv()函数是一个阻塞式函数。程序的执行将暂停直到recv()函数从远程客户端接收到数据或者远程客户端明确地断开TCP会话。如果客户端断开TCP会话，recv()函数将返回0.如果客户端发送一个数据包，recv()函数将接收到的数据放入定义的缓冲区并返回接收到的字节数。

在设计客户端和服务器的应用程序时，同步发送和接收函数时非常重要的。如果服务器和客户端同时等待在recv()函数调用上，它们将会产生死锁，并且不会有通信发生。

### recvfrom/sendto

UDP数据读写API：

ssize\_t recvfrom(int sockfd,void\* buf,size\_t len,int flags,struct sockaddr\* src\_addr,socklen\_t\* addrlen);

ssize\_t sendto(int sockfd,cons void\* buf,size\_t len,int flags,const struct sockaddr\* dest\_addr,sockelen\_t addrlen);

因为UDP通信没有连接的概念，所以每次读取数据都需要获取发送端数据的大小。

在面向连接的套接字编程中，客户端使用connect()函数与服务器建立TCP连接。无连接套接字并不会在网络设备之间创建一个专用的连接。相反，数据被扔到网络中，并希望它们能够自己到达指定的目的地。无连接套接字的好处是传输数据的开销很小，它不需要认真地跟踪每一个数据包，这极大地降低了处理数据包所需的开销。正因为如此，无连接套接字比面向连接套接字有更好的吞吐量。

无连接套接字使用套接字类型SOCK\_DGRAM实现。它通常使用用户数据报协议（UDP）在网络设备之间传递数据。由于无连接套接字不需要建立连接，所以服务器和客户端程序看起来很相似。

要发送UDP消息，套接字不需要使用bind函数。我们使用sendto()函数来定义数据和它要发送到的目的地。

要接收UDP消息，套接字必须使用bind()函数绑定到一个UDP端口以监听进入的数据包，这个和TCP连接一样。在bind()函数调用之后，数据包可以使用recvfrom()函数来接收。

由于recvfrom()函数是无连接的，所以它可以接收来自任何网络设备的数据。发送主机的网络地址信息保存在结构from中，而接收到的消息保存在缓冲区msg中。

### 发送过程

TCP消息接收的第一种场景：

1. 网卡先收到消息，用户后调用接收函数
2. 套接字为阻塞模式
3. 接收低水位默认为1
4. 没有使用到prequeue队列



TCP消息接收的第二种场景：

1. 用户先调用接收函数，网卡后收到消息
2. 套接字为阻塞模式
3. 接收低水位为默认值1
4. 涉及了prequeue队列



TCP消息接收的第三种场景：

1. 网卡先收到消息，用户后调用接收函数
2. 套接字为阻塞模式
3. 设置接收低水位为n
4. 没有使用prequeue队列



## 关闭连接close/shutdown

### close

**原型：**int close(int fd);

说明：close并不是真正关闭连接，而是将文件描述符fd的引用数-1，只有当fd引用次数为0时，才会关闭连接。如果想立即关闭连接，可以使用shutdown。

调用close是关闭TCP连接的存在两个限制：

1. close其实只是将socket fd的引用计数减1
2. close会终止TCP的双工链路

当连接处理完所有的这些数据之后，我们可以调用close()函数终止连接而不会造成任何数据丢失。

如果你想在如何关闭套接字上有多一点的控制，可以使用函数shutdown，它允许你将一定方向上的通讯或者双向通讯关闭。

### shutdown

**原型：**

int shutdown(int sockfd,int howto);

**参数：**

shutdown函数中howto参数选项：

SHUT\_RE：关闭读这一半

SHUT\_WR：关闭写这一半

SHUT\_RDWR：同时关闭读和写

套接字描述符可以使用shutdown()函数来指定通信会话终止的方式。该函数的原型如下：

shutdown(int socket, int how);

**参数：**

shutdown()函数使用参数how来让你决定如何优雅地关闭连接。可使用的选项如下：

0——不再接收数据包；

1——不再发送数据包；

2——不再发送或接收数据包。

通过选择0或1，你可以禁止套接字接收或发送更多的数据，但是允许套接字继续将剩余的数据发送完或继续接收剩余的数据。

### 对比

shutdown函数可以不管引用计数就激发了TCP的正常连接终止序列

利用shutdown函数可以关闭全双工中的任意一端

so\_linger选项在起到确保传输的功能

### 关闭过程



## 获取主机信息

### getsockname()

获取本地套接口的名字，包括它的IP和端口。

getsockname()在指定的套接口绑定地址和端口后才能调用，即服务器在bind()后可调用，客户端在bind()或connect()之后可调用。getpeername()在连接建立之后才可调用。

**注意：**

1）对于server来说：

bind以后就可以调用getsockname来获取本地地址和端口了，只不过这样没啥意义，因为就是自己绑定的。

只有在accept以后（就是有连接之后才）才能用getpeername获取client的ip和端口。（client的ip和port也可以在accept函数的第二个参数中带出）。

2）对于client来说：

创建socket并不会分配ip和端口。用getsockname获取出来的数据全是0。

在连接（connect）之后才可以用getsockname获取自己的ip和端口。也可以用getpeername获取服务器的。

### getpeername()

在连接的流式套接字上谁在另一边。

### gethostname()

返回程序所运行主机的主机名称。

### gethostbyname

gethostbyname通过主机名获取主机上所有的IP地址。

# 套接字选项/setsockopt

参考：https://www.man7.org/linux/man-pages/man7/socket.7.html

## SOL\_SOCKET协议族选项

### 获取和设置套接字选项

#### 函数

#include<sys/socket.h>

int getsockopt(int s, int level, int optname, void \*optval.socklen\_t\*optlen);

int setsockopt(int s, int level, int optname, void\*optval.socklen\_t optlen);

参数：

s：将要设置或获取的套接字描述符

optname：操作内存缓冲区

optval.socklen\_t optlen：第4个参数长度

#### 错误代码

getsockopt()和setsockopt函数错误代码含义：

EBADF：参数s不是有效的文件描述符

EFAULT：optval指向的内存并非有效的进程空间

EINVAL：在调用setsockopt函数时，optlen无效

ENOPROTOOPT：指定的协议层不能识别选项

ENOTSOCK：s描述的不是套接字描述符

#### level

按照参数选项界别level值的不同，套接字选项可以分为3类：

通用套接选项：level的值为SOL\_SOCKET

IP选项：level的值为IPPROTO\_IP

TCP选项：level的值为IPPROTO\_TCP

### SO\_KEEPALIVE选项

SO\_KEEPALIVE 保持连接选项，用于设置TCP/IP的连接，2小时没有数据交互，发送探测报文，有三种回应：

回应一个ACK报文

回应一个RST报文

没有任何回应

注：通常在服务端设置，用于长时间没有响应的TCP连接场景。

使用：

int optval=1;

setsockopt(sockfd, SOL\_SOCKET,SO\_KEEPALIVE,&optval, sizeof(iptval));

### SO\_LINGER选项

SO\_LINGER 缓冲区处理方式选项，主要用于设置TCP连接**关闭**（即系统调用close）时的方式，它的操作通过一个结构体进行：

struct linger{

int l\_onoff; //开启（非0）还是关闭（0）该选项

int l\_linger; //滞留时间

};

SO\_LINGER选项使用方法有三种：

l\_onoff等于0，此时SO\_LINGER选项不起作用

l\_onoff不为0，l\_linger为0：close系统调用立即返回，TCP模块将丢弃被关闭的套接字对应的TCP发送缓冲区中残留的数据，同时给对方发送一个复位报文段

l\_onoff不为0，l\_linger不为0：此时close行为取决于两个条件：

1、被关闭的套接字对应的TCP发送缓冲区中是否含有残留的数据；

2、**该套接字是阻塞还是非阻塞，对于阻塞的，close将等待l\_linger时间，直到TCP模块发送完所有缓冲区残留数据，并得到对方确认，否则close系统调用返回-1，如果是非阻塞的，则返回-1**

使用方法：

optval.l\_onoff = 1;

optval.l\_linger = 60;

setsockopt(sockfd, SOL\_SOCKET, SO\_LINGER, &optval, sizeof(optval));

### SO\_RCVBUF和SO\_SNDBUF选项

SO\_RCVBUF和SO\_SNDBUF缓冲区大小选项：

1、设置TCP/UDP连接的接收/发送缓冲区的含义：在UDP连接中，由于是无连接，在发送缓冲区在数据通过网络设备发送后就可以丢弃，不用保存，而接收缓冲区需要保存数据，直到应用程序读取，由于UDP没有流量控制，当缓冲区过小时，局部时间内会产生爆发性数据传输，因此在使用UDP时需要将接收缓冲区调整到比较大的值；在TCP连接中，接收缓冲区的大小就是滑动窗口的大小，TCP的接收缓冲区不可能溢出，因为不允许对方发送超过接收缓冲区大小的数据。

2、在connect()函数调用之前设置：滑动窗口的协商是在建立连接时，通过同步报文获得的，对于客户端程序而言，接收缓冲区的大小在connect系统调用之前设置，因为connect需要通过同步报文段进行连接，而对于 服务器而言需要在listen之前设置接收缓冲区的大小，因为accept返回的套接字描述符是继承了listen的描述符属性。

发送缓冲区设置：

#include <sys/socket.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#define BUFFER\_SIZE 1024

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port number!\r\n");

return -1;

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

struct sockaddr\_in server\_address;

bzero(&server\_address,sizeof(server\_address));

server\_address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&server\_address.sin\_addr);

server\_address.sin\_port = htons(port);

int sock = socket(AF\_INET,SOCK\_STREAM,0);

if(sock < 0)

{

printf("create socket failed !\r\n");

return -1;

}

int sendbuf = atoi(argv[3]);

int len = sizeof(sendbuf);

setsockopt(sock,SOL\_SOCKET,SO\_SNDBUF,&sendbuf,sizeof(sendbuf));

getsockopt(sock,SOL\_SOCKET,SO\_SNDBUF,&sendbuf,(socklen\_t\*)&len);

printf("tcp send buffer size after setting is %d\r\n",sendbuf);

if(connect(sock,(struct sockaddr\*)&server\_address,sizeof(server\_address)) != -1)

{

char buffer[BUFFER\_SIZE];

memset(buffer,'a',BUFFER\_SIZE);

send(sock,buffer,BUFFER\_SIZE,0);

}

close(sock);

return 0;

}

接收缓冲区设置：

#include <sys/socket.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#define BUFFER\_SIZE 1024

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port number!\r\n");

return -1;

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

struct sockaddr\_in server\_address;

bzero(&server\_address,sizeof(server\_address));

server\_address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&server\_address.sin\_addr);

server\_address.sin\_port = htons(port);

int sock = socket(AF\_INET,SOCK\_STREAM,0);

if(sock < 0)

{

printf("create socket failed !\r\n");

return -1;

}

int recvbuf = atoi(argv[3]);

int len = sizeof(recvbuf);

setsockopt(sock,SOL\_SOCKET,SO\_RCVBUF,&recvbuf,sizeof(recvbuf));

getsockopt(sock,SOL\_SOCKET,SO\_RCVBUF,&recvbuf,(socklen\_t\*)&len);

printf("the tcp receive buffer size after setting is %d\r\n",recvbuf);

int ret = bind(sock,(struct sockaddr\*)&server\_address,sizeof(server\_address));

if(ret == -1)

{

printf("bind failed !\r\n");

return -1;

}

ret = listen(sock,5);

if(ret == -1)

{

printf("listen failed !\r\n");

return -1;

}

struct sockaddr\_in client;

socklen\_t client\_len = sizeof(client);

int connfd = accept(sock,(struct sockaddr\*)&client,&client\_len);

if(connfd < 0)

{

printf("accept failed !\r\n");

}

else

{

char buffer[BUFFER\_SIZE];

memset(buffer,0x00,sizeof(buffer));

while(recv(connfd,buffer,BUFFER\_SIZE,0) > 0) {}

close(connfd);

}

close(sock);

return 0;

}

### SO\_RCVLOWAT和SO\_SNDLOWAT选项

SO\_RCVTIMEO表示接收超时，SO\_SNDTIMEO表示发送超时（发送超时与接收超时与IO复用中select部分功能是一致的），设置是通过一个结构体来实现：

struct timeval{

time\_t tv\_sec;

seconds\_t tv\_usec;

};

### SO\_RCVTIMEO和SO\_SNDTIMEO选项

SO\_RCVLOWAT和SO\_SNDLOWAT选项表示接收缓冲区和发送缓冲区的低水位标记（一般被IO复用调用，判断socket是否可读/可写）。

当TCP接收缓冲区可读数据的总数大于其低水位标记时，IO复用系统调用将通知应用程序可以对对应的socket上读取数据；当TCP发送缓冲区中的空闲空间，即可写入数据的空间大于其低水位标记时，IO复用系统调用将通知应用程序可以对对应的socket上写入数据。

在默认情况下，TCP接收缓冲区的低水位标记和TCP发送缓冲区的低水位标记都是为1字节。

### SO\_REUSEADDR选项

SO\_REUSEADDR地址可重用选项，设置此选项可以使用被处于TIME\_WAIT状态的连接占用的socket地址：

int reuse=1；

setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &reuse, sizeof(reuse));

实例：

int sock = socket(AF\_INET,SOCK\_STREAM,0);

assert(sock >=0);

int reuse = 1;

setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&reuse,sizeof(reuse));

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int ret = bind(sock,(struct sockaddr\*)&address,sizeof(address));

## IPPROTO\_TCP选项

### TCP\_KEEPALIVE选项

TCP\_KEEPALIVE用来获取或设置存活探测的时间间隔（在SO\_KEEPALIVE设置的情况下才有效，默认情况下2小时）：

int alivetime = 60;

int fd = socket(AF\_INET, SOCK\_STREAM, 0);

setsockopt(fd, IPPOTO\_TCP, TCP\_KEPALIVE, &alivetime. Sizeof(alivetime));

怎样检测长连接是否中断？

1、在应用层使用heartbeat来主动检测

对于实时性要求较高的网络通信程序，往往需要更加及时的获取已经中断的连接，从而进行及时的处理。但如果对方的连接异常中断，往往是不能及时的得到对方连接已经中断的信息，操作系统检测连接是否中断的时间间隔默认是比较长的，即便它能够检测到，但却不符合我们的实时性需求，所以需要我们进行手工去不断探测。

2、改变socket的keepalive选项，以使socket检测连接是否中断的时间间隔更小，以满足我们的及时性需求。

有关的几个选项使用和解析如下：

1）我们在检测对端以一种非优雅的方式断开连接的时候，可以设置SO\_KEEPALIVE属性使得我们在2小时以后发现对方的TCP连接是否依然存在。

用法如下：

keepAlive = 1；

setsockopt(listenfd, SOL\_SOCKET, SO\_KEEPALIVE, (void\*)&keepAlive, sizeof(keepAlive));

2）如果我们不想使用这么长的等待时间，可以修改内核关于网络方面的配置参数，也可设置SOCKET的TCP层（SOL\_TCP）选项TCP\_KEEPIDLE、TCP\_KEEPINTVL和TCP\_KEEPCNT。

TCP\_KEEPIDLE：开始首次KeepAlive探测前的TCP空闭时间（默认2h）

TCP\_KEEPINTVL：两次KeepAlive探测间的时间间隔（默认75s）

TCP\_KEEPCNT：断开前的KeepAlive探测次数

如果心跳函数要维护客户端的存活，即服务器必须每隔一段时间必须向客户段发送一定的数据，那么使用SO\_KEEPALIVE是有很大的不足的。因为SO\_KEEPALIVE选项指"此套接口的任一方向都没有数据交换"。

在Linux 2.6系列上，上面话的理解是只要打开SO\_KEEPALIVE选项的套接口端检测到数据发送或者数据接受就认为是数据交换。

因此在这种情况下使用 SO\_KEEPALIVE选项 检测对方是否非正常连接是完全没有作用的，在每隔一段时间发包的情况， keep-alive的包是不可能被发送的。上层程序在非正常断开的情况下是可以正常发送包到缓冲区的。

非正常端开的情况是指服务器没有收到"FIN" 或者 "RST"包。

### TCP\_USER\_TIMEOUT选项

参考：<https://blog.csdn.net/a15010608983/article/details/102284766>

在TSQL中也有应用：

**背景：**

TCP的keepalive 在断电，拔网线的情况下，并不能及时感知到tcp断开链接，需要引入tcp\_user\_timeout的tcp选项。

**参数设置：**

新增gateway.security.keepalive.tcp\_user\_timeout，单位是ms，默认是10000ms，即10s。

### TCP\_KEEPIDLE选项

TCP\_KEEPIDLE：开始首次KeepAlive探测前的TCP空闭时间（默认2h）

### TCP\_KEEPINTVL选项

TCP\_KEEPINTVL：两次KeepAlive探测间的时间间隔（默认75s）

### TCP\_KEEPCNT选项

TCP\_KEEPCNT：断开前的KeepAlive探测次数

### TCP\_NODELAY和TCP\_CORK选项

TCP\_NODELAY和TCP\_CORK针对Nagle算法，此算法的基本原理如下：

1. 将小分组封装成大的分组/帧再发送
2. 使用延迟确认

TCP\_NODELAY选项的作用：客户端的请求不需要和其他分组合并

TCP\_CORK选项作用：需要等到发送的数据量达到最大值，一次性发送全部数据

### TCP\_QUICKACK选项

TCP\_QUICKACK（since Linux 2.4.4）设置后会立刻发送确认ACK，而不是延迟发送ack。若未开启，则Delay ack延迟确认，使得协议有机会合并ack，提高网络利用率，默认40ms超时确认，系统值可配置。

参考：<https://blog.csdn.net/WEI_GW2012/article/details/123213084>

## IPPROTO\_IP选项

### IP\_TOS选项

IP\_TOS可以设置服务类型，服务类型选项：

IPTOS\_LOWDELAY：表示最小延迟

IPTOS\_RELIABLITY：表示最大可靠性

IPTOS\_THROUGHPUT：表示最大吞吐量

IPTOS\_LOWCOST：表示最小成本

### IP\_TTL选项

IP\_TTL是一种生存时间选项，而生成时间选项：

可以获取或设置发送报文的TTL值（一般为64.对于套接字为255）

可以调整网络数据发送速度

实例：

int current\_TTL =0;

int set\_TTL = 32;

int length\_TTL =sizeof(int);

getsockopt(s,IPPROTO\_IP,IP\_TTL,&current\_TTL,&lenght\_TTL);

setsockopt(s,IPPROTO\_IP,IP\_TTL,&set\_TTL,length\_TTL);

sendto(s,buffer,buffer\_len,0,(struct sockaddr\*)&remote\_ip,sizeof(struct sockaddr));

setsockopt(s,IPPROTO\_IP,IP\_TTL,&current\_TTL,length\_TTL);

## ioctl()函数

ioctl、fcntl、setsockopt这三个系统函数都是用来设置文件描述符fd属性的，在Linux中，一切设备皆文件，这些文件范围广泛，有些属性是所有文件都具有的共性，而另外一些属性则是某个种类专有的特性。

Linux使用ioctl设置与IO设备高度相关的属性，使用fcntl设置文件描述符的属性，使用setsockopt设置网络套接字的属性，总体而言设备范围逐渐缩小，所设置的属性更具有网络编程针对性。

参考：<https://blog.csdn.net/ArtAndLife/article/details/114991563>

### 原型

函数：ioctl(int fd, int request, void \* arg)

定义：<sys/ioctl.h>

功能：控制I/O设备，提供了一种获得设备信息和向设备发送控制参数的手段。

参数：

fd：文件句柄，用于socket时, 是socket套接字。

request：函数定义的所有操作，关于socket的操作, 定义在<linux/sockios.h>文件中。

arg：指针的类型依赖于request参数。

### 命令选项



#### ioctl函数命令选项

ioctl函数和**内核协议栈**交互，主要的交互类型：

IO

文件

网络接口

路由

ARP

#### ioctl函数IO请求

套接字IO操作的命令请求有6个，第3个参数为整形指针：

SIOCATMARK：查看TCP连接是否有带外数据

SIOCSPGRP和FIOSETOWN：处理套接字的SIGIO和SIGURG信号

SIOCGPGRP和FIOGETOWN：设置接收SIGIO信号的进程ID

SIOCGSTAMP：获取最后一个数据报达到的事件

#### ioctl函数文件请求

文件请求的命令有3个：

FIONBIO：用于设置或清除套接字的非阻塞标志

FIOASYNC：设置或者清除套接字的异步信号

FIONREAD：获得当前套接字接收缓冲区的字节数

#### ioctl函数网络接口请求

获取网络接口的不同参数的不同命令选项：

SIOCGIFADDR和SIOCSIFADDR：获取和设置本地IP地址

SIOCGIFMTU和SIOCSIFMTU：获取和设置MTU

SIOCGIFHWADDR和SIOCSIFHADDR：获取和设置硬件地址

实例：

#include <stdio.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/ioctl.h>

#include <net/if\_arp.h>

#include <net/if.h>

#include <stdlib.h>

#include <string.h>

#include <net/if.h>

int main()

{

int s;

int err = -1;

s = socket(AF\_INET,SOCK\_DGRAM,0);

if(s < 0)

{

printf("socket create faield !\r\n");

return -1;

}

struct ifreq ifr;

strcpy(ifr.ifr\_name,"eth0");

err = ioctl(s,SIOCGIFFLAGS,&ifr);

if(err == -1)

{

printf("ioctl failed 0 \r\n");

return -1;

}

else

{

printf("SICGIFFLAGS:%d\r\n",ifr.ifr\_flags);

}

/\* MTU \*/

err = ioctl(s,SIOCGIFMTU,&ifr);

if(err == -1)

{

printf("ioctl failed 1 !\r\n");

return -1;

}

else

{

printf("SIOCGIFMTU:%d\r\n",ifr.ifr\_mtu);

}

/\* MAC ADDRESS\*/

err = ioctl(s,SIOCGIFHWADDR,&ifr);

if(err == -1)

{

printf("ioctl failed 2! \r\n");

return -1;

}

else

{

unsigned char\* hw = ifr.ifr\_hwaddr.sa\_data;

printf("SIOCGIFADDR:%02x:%02x:%02x:%02x:%02x:%02x\r\n",hw[0],hw[1],hw[2],hw[3],hw[4],hw[5]);

}

close(s);

return 0;

}

#### ioctl函数对ARP告诉缓存操作

ioctl函数获取ARP高速缓存，有三个命令字：

SIOCDARP：删除高速缓存中的一个记录

SIOCARP：设置或者修改一个记录

SIOCGARP：获得一个记录

实例：

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <net/if\_arp.h>

#include <net/if.h>

#include <sys/ioctl.h>

#include <string.h>

int main(int argc,char\* argv[])

{

int s;

struct arpreq arpreq;

struct ifreq ifr;

struct sockaddr\_in\* addr = (struct sockaddr\_in\*)&arpreq.arp\_pa;

unsigned char\* hw;

int err = -1;

if(argc<2)

{

printf("please input ip address !\r\n");

return 0;

}

s = socket(AF\_INET,SOCK\_STREAM,0);

if(s<0)

{

printf("create socket failed !\r\n");

return 0;

}

addr->sin\_family = AF\_INET;

inet\_pton(AF\_INET,argv[1],&addr->sin\_addr);

if(addr->sin\_addr.s\_addr == INADDR\_NONE)

{

printf("ip address error!\r\n");

return -1;

}

strcpy(arpreq.arp\_dev,"eth0");

err = ioctl(s,SIOCGARP,&arpreq);

if(err == -1)

{

printf("ioctl failed !\r\n");

}

else

{

hw = (unsigned char\*)&arpreq.arp\_ha.sa\_data;

printf("%s: ",argv[1]);

printf("%02x:%02x:%02x:%02x:%02x:%02x\r\n",hw[0],hw[1],hw[2],hw[3],hw[4],hw[5]);

}

close(s);

return 0;

}

## fcntl()函数

ioctl用于硬件设备I/O通道控制，控制命令和参数都与设备高度相关，通常也与系统高度相关。

**ioctl是底层的系统调用，所以跨平台性不好**（可能某些平台的IO属性在其他平台上并不具有）；而**fcntl是封装好的函数，各个OS都是支持的**。

在网络socket中，它们都可以用于设置socket描述字的某些属性，例如非阻塞：

fcntl(sockfd, F\_SETFL, fcntl(sockfd, F\_GETFL) | O\_NONBLOCK));

ioctl(sockfd, FIONBIO, 1); //1表示非阻塞，0表示阻塞

### 原型

函数原型：

用法：

int fcntl(int fd, int cmd);

int fcntl(int fd, int cmd, long arg);

int fcntl(int fd, int cmd, struct flock \*lock);

参数：

fd：文件描述词。

cmd：操作命令。

arg：供命令使用的参数。

lock：同上。

fcntl的操作对象是文件描述符，用来控制文件描述符属性的通用方法（例如 读写属性、是否阻塞等），setsockopt是用来专门设置套接字描述符的属性，针对网络编程中常见的属性设置的需求。

只要是文件描述符，就可以使用fcntl；setsockopt偏向于设置IP、TCP协议对应的套接字属性。

参考：<https://blog.csdn.net/zh_wang2010/article/details/8467436>

### 设置类型

#### F\_DUPFD

F\_DUPFD：复制文件描述词 。

#### FD\_CLOEXEC

FD\_CLOEXEC：设置close-on-exec标志。如果FD\_CLOEXEC位是0，执行execve的过程中，文件保持打开。反之则关闭。

#### F\_GETFD

F\_GETFD：读取文件描述词标志。

#### F\_SETFD

F\_SETFD：设置文件描述词标志。

#### F\_GETFL

F\_GETFL：读取文件状态标志。

#### F\_SETFL

F\_SETFL：设置文件状态标志。

其中O\_RDONLY， O\_WRONLY， O\_RDWR， O\_CREAT， O\_EXCL， O\_NOCTTY 和 O\_TRUNC不受影响，可以更改的标志有O\_APPEND，O\_ASYNC，O\_DIRECT，O\_NOATIME和O\_NONBLOCK。

#### F\_GETLK/F\_SETLK/F\_SETLKW

F\_GETLK, F\_SETLK 和 F\_SETLKW ：获取，释放或测试记录锁，使用到的参数是以下结构体指针：

F\_SETLK：在指定的字节范围获取锁（F\_RDLCK, F\_WRLCK）或者释放锁（F\_UNLCK）。如果与另一个进程的锁操作发生冲突，返回 -1并将errno设置为EACCES或EAGAIN。

F\_SETLKW：行为如同F\_SETLK，除了不能获取锁时会睡眠等待外。如果在等待的过程中接收到信号，会立即返回并将errno置为EINTR。

F\_GETLK：获取文件锁信息。

F\_UNLCK：释放文件锁。

为了设置读锁，文件必须以读的方式打开。为了设置写锁，文件必须以写的方式打开。为了设置读写锁，文件必须以读写的方式打开。

#### 信号管理

F\_GETOWN, F\_SETOWN, F\_GETSIG 和 F\_SETSIG 被用于IO可获取的信号。

F\_GETOWN：获取当前在文件描述词 fd上接收到SIGIO 或 SIGURG事件信号的进程或进程组标识 。

F\_SETOWN：设置将要在文件描述词fd上接收SIGIO 或 SIGURG事件信号的进程或进程组标识 。

F\_GETSIG：获取标识输入输出可进行的信号。

F\_SETSIG：设置标识输入输出可进行的信号。

使用以上命令，大部分时间程序无须使用select()或poll()即可实现完整的异步I/O。

#### 租约（Leases）

F\_SETLEASE 和 F\_GETLEASE 被用于当前进程在文件上的租约。文件租约提供当一个进程试图打开或折断文件内容时，拥有文件租约的进程将会被通告的机制。

F\_SETLEASE：根据以下符号值设置或者删除文件租约

F\_RDLCK设置读租约，当文件由另一个进程以写的方式打开或折断内容时，拥有租约的当前进程会被通告。

F\_WRLCK设置写租约，当文件由另一个进程以读或以写的方式打开或折断内容时，拥有租约的当前进程会被通告。

F\_UNLCK删除文件租约。

F\_GETLEASE：获取租约类型。

#### 文件或目录改变通告

（linux 2.4以上）当fd索引的目录或目录中所包含的某一文件发生变化时，将会向进程发出通告。arg参数指定的通告事件有以下，两个或多个值可以通过或运算组合。

1. DN\_ACCESS 文件被访问 (read, pread, readv)
2. DN\_MODIFY 文件被修改(write, pwrite,writev, truncate, ftruncate)
3. DN\_CREATE 文件被建立(open, creat, mknod, mkdir, link, symlink, rename)
4. DN\_DELETE 文件被删除(unlink, rmdir)
5. DN\_RENAME 文件被重命名(rename)
6. DN\_ATTRIB 文件属性被改变(chown, chmod, utime[s])

返回说明：

成功执行时，对于不同的操作，有不同的返回值

F\_DUPFD：新文件描述词

F\_GETFD：标志值

F\_GETFL：标志值

F\_GETOWN：文件描述词属主

F\_GETSIG： 读写变得可行时将要发送的通告信号，或者0对于传统的SIGIO行为

对于其它命令返回0。

失败返回-1，errno被设为以下的某个值

EACCES/EAGAIN: 操作不被允许，尚未可行

EBADF: 文件描述词无效

EDEADLK: 探测到可能会发生死锁

EFAULT: 锁操作发生在可访问的地址空间外

EINTR: 操作被信号中断

EINVAL： 参数无效

EMFILE: 进程已超出文件的最大可使用范围

ENOLCK: 锁已被用尽

EPERM:权能不允许

struct flock {

short l\_type; /\* 锁类型：F\_RDLCK, F\_WRLCK, F\_UNLCK \*/

short l\_whence;

/\* l\_start字段参照点：SEEK\_SET(文件头),

SEEK\_CUR(文件当前位置), SEEK\_END(文件尾) \*/

off\_t l\_start; /\* 相对于l\_whence字段的偏移量 \*/

off\_t l\_len; /\* 需要锁定的长度 \*/

pid\_t l\_pid; /\* 当前获得文件锁的进程标识（F\_GETLK） \*/

};

fcntl的典型应用：设置非阻塞IO：

int setnonblocking(int fd) {

int old\_option = fcntl(fd, F\_GETFL);

int new\_option = old\_option | O\_NONBLOCK;

fcntl(fd, F\_SETFL, new\_option);

return new\_option;

}

//另一种简洁写法：

int setnonblocking(int fd) {

int option = fcntl(fd, F\_SETFL, fcntl(fd, F\_GETFL) | O\_NONBLOCK);

return option;

}

### setsockopt()函数

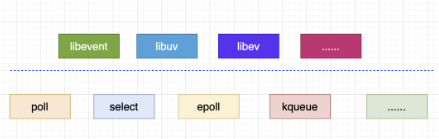
# 最大连接数

## C10K/C10M

在2000年初的时候，全球互联网的规模并不大，但是当时就已经提出了C10K问题，所谓C10K就是单机1w并发问题，虽然现在不觉得是个难题了，但是这在当初是很有远见和挑战的问题。

在APUE第三版都没有提到epoll，所以我们解决C10K问题的时间并不长，其中IO复用epoll/kqueue/iocp等技术对于C10k问题的解决起到了非常重要的作用。

开源大神们基于epoll/kqueue等开发了诸如libevent/libuv等网络库，从而大幅提高了高并发网络的开发效率，对于C/C++程序员来说并不陌生。



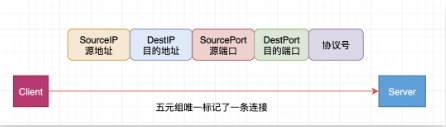
让单机达到1000w并发，现在听起来感觉不可思议，但是要达到这个目标，除了硬件上的提升，更重要的是对系统软件和协议栈的改造。

不要让OS内核执行所有繁重的任务：将数据包处理、内存管理、处理器调度等任务从内核转移到应用程序高效地完成，让诸如Linux这样的OS只处理控制层，数据层完全交给应用程序来处理。

## 服务器最大并发数

### 五元组

一个五元组可以唯一标记一个网络连接，所以要理解和分析最大并发数，就必须理解五元组：



这样的话，就可以基本认为：理论最大并发数=服务端唯一五元组数。

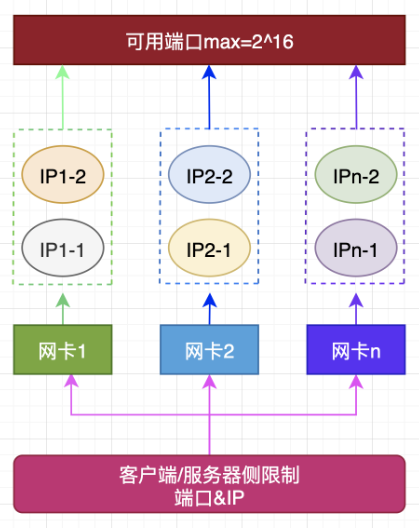
### 端口&IP组合数

那么对于服务器来说，服务端唯一五元组数最大是多少呢？

有人说是65535，显然不是，但是之所以会有这类答案是因为当前Linux的端口号是2字节大小的short类型，总计2^16个端口，除去一些系统占用的端口，可用端口确实只剩下64000多了。

对于服务端本身来说，DestPort数量确实有限，假定有多张网卡，每个网卡绑定多个IP，服务端的Port端口数和IP数的组合类型也是有限的。

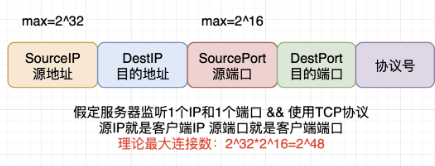
对于客户端来说，本身的端口和IP也是一样有限的，虽然这是个组合问题，但是数量还是有限的：



### 并发数理论极限

前面的计算都是针对单个服务器或者客户端的，但是实际上每个服务器会应对全网的所有客户端，那么从服务端看，源IP和源Port的数量是非常大的。

理论上服务端可以接受的客户端IP是2^32(按照IPv4计算）,端口数是2^16，目前端口号仍然是16bit的，所有这个理论最大值是2^48，果然很大！



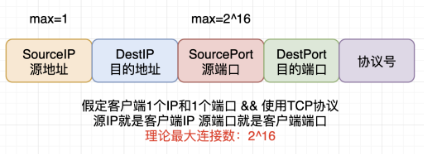
每一条连接都是要消耗系统资源的，所以实际中可能会设置最大并发数来保证服务器的安全和稳定，所以这个理论最大并发数是不可能达到的。

实际中并发数和业务是直接相关的，像Redis这种内存型的服务端并发十几万都是没问题的，大部分来讲几十/几百/几千/几万等是存在的。

## 客户端最大连接数

理解了服务器的最大并发数是2^48，那么客户端最多可以连接多少服务器呢？

对于客户端来说，当然可以借助于多网卡多IP来增加连接能力，我们仍然假定客户端只有1张网卡1个IP，由于端口数的限制到2^16，再去掉系统占用的端口，剩下可用的差不多64000。



也就是说，客户端虽然可以连接任意的目的IP和目的端口，但是客户端自身端口是有限的，所以客户端的理论最大连接数是2^16，含系统占用端口。

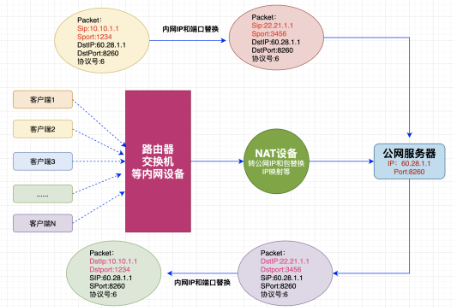
## NAT环境下的客户端

解决前面的两个问题之后，来看另外一个问题：

一个公网出口NAT服务设备最多可同时支持多少内网IP并发访问外网服务？

毕竟公网IP都是有限并且要花钱的，我们大部分机器都是在局域网中结合NAT来进行外网访问的，所以这个场景还是很熟悉的。

来看下内网机器访问外网时的IP&端口替换和映射还原的过程，就明白了：

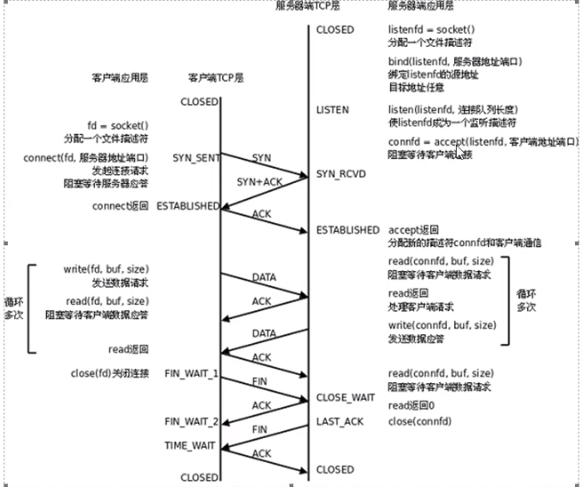


因为这时的客户端是NAT设备，所以NAT环境下最多支持65535个并发访问外网。

# TCP服务端/客户端开发

## 流程





## 示例

### 客户端

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <assert.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input server ip and port number\r\n");

return -1;

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

struct sockaddr\_in server\_address;

bzero(&server\_address,sizeof(server\_address));

server\_address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&server\_address.sin\_addr);

server\_address.sin\_port = htons(port);

int sockfd = socket(AF\_INET,SOCK\_STREAM,0);

if(sockfd < 0)

{

printf("socket create failed \r\n");

return -1;

}

if(connect(sockfd,(struct sockaddr\*)&server\_address,sizeof(server\_address))<0)

{

printf("connect failed \r\n");

return -1;

}

else

{

char data[64]="Where are you from ?";

char buffer[64];

int i,len;

for(i=0;i<10;i++)

{

memset(buffer,0x00,sizeof(buffer));

send(sockfd,data,strlen(data),0);

len = recv(sockfd,buffer,sizeof(buffer),0);

sleep(1);

printf("index is %d len is %d : %s\r\n",i,len,buffer);

}

}

close(sockfd);

return 0;

}

### 服务端

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(int argc,char\* argv[])

{

if(argc <=2 )

{

printf("please input ipaddress and port number\r\n");

return -1;

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int sock = socket(AF\_INET,SOCK\_STREAM,0);

if(sock <0)

{

printf("create sock failed \r\n");

return -1;

}

int ret = bind(sock,(struct sockaddr\*)&address,sizeof(address));

if(ret == -1)

{

printf("bind failed\r\n");

return -1;

}

ret = listen(sock,5);

if(ret == -1)

{

printf("listen failed \r\n");

return -1;

}

struct sockaddr\_in client;

socklen\_t client\_addrlength = sizeof(client);

int connfd = accept(sock,(struct sockaddr\*)&client,&client\_addrlength);

if(connfd < 0)

{

printf("accept failed \r\n");

}

else

{

char data[64]="I am from jikexueyuan!";

char buffer[64];

int i,len;

for(i=0;i<10;i++)

{

memset(buffer,0x00,sizeof(buffer));

len = recv(connfd,buffer,sizeof(buffer),0);

printf("index is %d len is %d: %s\r\n",i,len,buffer);

sleep(1);

send(connfd,data,strlen(data),0);

}

close(connfd);

}

close(sock);

return 0;

}

# UDP服务端/客户端开发

## 流程

## 示例

### 客户端

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#define PORT\_SERVER 8888

int main(int argc,char\* argv[])

{

int s;

struct sockaddr\_in addr\_server,addr\_client;

const char\* ip = "192.168.80.130";

bzero(&addr\_server,sizeof(addr\_server));

inet\_pton(AF\_INET,ip,&addr\_server.sin\_addr);

addr\_server.sin\_family = AF\_INET;

addr\_server.sin\_port = htons(PORT\_SERVER);

s = socket(AF\_INET,SOCK\_DGRAM,0);

printf("s is %d \r\n",s);

char data[64]="Where are you from ?";

char buffer[64];

int i;

int len = sizeof(addr\_server);

for(i=0;i<10;i++)

{

memset(buffer,0x00,sizeof(buffer));

int ret = sendto(s,data,strlen(data),0,(struct sockaddr\*)&addr\_server,len);

recvfrom(s,buffer,sizeof(buffer),0,(struct sockaddr\*)&addr\_client,&len);

printf("index is %d : %s \r\n",i,buffer);

sleep(1);

}

close(s);

return 0;

}

### 服务端

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define PORT\_SERVER 8888

int main(int argc,char\* argv[])

{

int s;

struct sockaddr\_in addr\_server,addr\_client;

const char\* ip = "192.168.80.130";

s = socket(AF\_INET,SOCK\_DGRAM,0);

memset(&addr\_server,0x00,sizeof(addr\_server));

addr\_server.sin\_family = AF\_INET;

addr\_server.sin\_addr.s\_addr = htonl(INADDR\_ANY);

addr\_server.sin\_port = htons(PORT\_SERVER);

bind(s,(struct sockaddr\*)&addr\_server,sizeof(addr\_server));

char data[64]="I am from jikexueyuan!";

char buffer[64];

int i;

int len = sizeof(addr\_server);

for(i=0;i<10;i++)

{

memset(buffer,0x00,sizeof(buffer));

recvfrom(s,buffer,sizeof(buffer),0,(struct sockaddr\*)&addr\_client,&len);

printf("index is %d : %s \r\n",i,buffer);

sendto(s,data,strlen(data),0,(struct sockaddr\*)&addr\_client,len);

sleep(1);

}

close(s);

return 0;

}