# Socket编程

## 套接字概念

Socket本身有“插座”的意思，在Linux环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

套接字的内核实现较为复杂，不宜在学习初期深入学习。

在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程。“IP地址+端口号”就对应一个socket。欲建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。因此可以用Socket来描述网络连接的一对一关系。

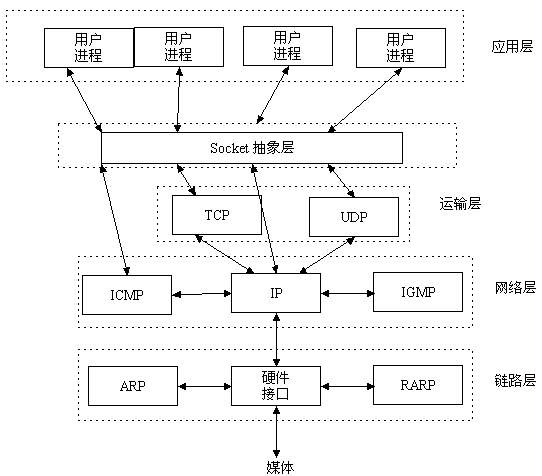
套接字通信原理如下图所示：



套接字通讯原理示意

**在网络通信中，套接字一定是成对出现的。**一端的发送缓冲区对应对端的接收缓冲区。我们使用同一个文件描述符索发送缓冲区和接收缓冲区。

TCP/IP协议最早在BSD UNIX上实现，为TCP/IP协议设计的应用层编程接口称为socket API。本章的主要内容是socket API，主要介绍TCP协议的函数接口，最后介绍UDP协议和UNIX Domain Socket的函数接口。



网络编程接口

## 预备知识

### 网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP协议规定，网络数据流应采用大端字节序，即低地址高字节。例如上一节的UDP段格式，地址0-1是16位的源端口号，如果这个端口号是1000（0x3e8），则地址0是0x03，地址1是0xe8，也就是先发0x03，再发0xe8，这16位在发送主机的缓冲区中也应该是低地址存0x03，高地址存0xe8。但是，如果发送主机是小端字节序的，这16位被解释成0xe803，而不是1000。因此，发送主机把1000填到发送缓冲区之前需要做字节序的转换。同样地，接收主机如果是小端字节序的，接到16位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32位的IP地址也要考虑网络字节序和主机字节序的问题。

为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换**。

#include <arpa/inet.h>

uint32\_t htonl(uint32\_t hostlong);

uint16\_t htons(uint16\_t hostshort);

uint32\_t ntohl(uint32\_t netlong);

uint16\_t ntohs(uint16\_t netshort);

h表示host，n表示network，l表示32位长整数，s表示16位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

### IP地址转换函数

早期：

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet\_aton(const char \*cp, struct in\_addr \*inp);

in\_addr\_t inet\_addr(const char \*cp);

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

只能处理IPv4的ip地址

注意参数是struct in\_addr

现在：

#include <arpa/inet.h>

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

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

支持IPv4和IPv6

其中inet\_pton和inet\_ntop不仅可以转换IPv4的in\_addr，还可以转换IPv6的in6\_addr。

因此函数接口是void \*addrptr。

### sockaddr数据结构

strcut sockaddr 很多网络编程函数诞生早于IPv4协议，那时候都使用的是sockaddr结构体,为了向前兼容，现在sockaddr退化成了（void \*）的作用，传递一个地址给函数，至于这个函数是sockaddr\_in还是sockaddr\_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。



sockaddr数据结构

struct sockaddr {

sa\_family\_t sa\_family; /\* address family, AF\_xxx \*/

char sa\_data[14]; /\* 14 bytes of protocol address \*/

};

使用 sudo grep -r "struct sockaddr\_in {" /usr 命令可查看到struct sockaddr\_in结构体的定义。一般其默认的存储位置：/usr/include/linux/in.h 文件中。

struct sockaddr\_in {

\_\_kernel\_sa\_family\_t sin\_family; /\* Address family \*/ 地址结构类型

\_\_be16 sin\_port; /\* Port number \*/ 端口号

struct in\_addr sin\_addr; /\* Internet address \*/ IP地址

/\* Pad to size of `struct sockaddr'. \*/

unsigned char \_\_pad[\_\_SOCK\_SIZE\_\_ - sizeof(short int) -

sizeof(unsigned short int) - sizeof(struct in\_addr)];

};

struct in\_addr { /\* Internet address. \*/

\_\_be32 s\_addr;

};

struct sockaddr\_in6 {

unsigned short int sin6\_family; /\* AF\_INET6 \*/

\_\_be16 sin6\_port; /\* Transport layer port # \*/

\_\_be32 sin6\_flowinfo; /\* IPv6 flow information \*/

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

\_\_u32 sin6\_scope\_id; /\* scope id (new in RFC2553) \*/

};

struct in6\_addr {

union {

\_\_u8 u6\_addr8[16];

\_\_be16 u6\_addr16[8];

\_\_be32 u6\_addr32[4];

} in6\_u;

#define s6\_addr in6\_u.u6\_addr8

#define s6\_addr16 in6\_u.u6\_addr16

#define s6\_addr32 in6\_u.u6\_addr32

};

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

\_\_kernel\_sa\_family\_t sun\_family; /\* AF\_UNIX \*/

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

};

Pv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr\_in结构体表示，包括16位端口号和32位IP地址，IPv6地址用sockaddr\_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。UNIX Domain Socket的地址格式定义在sys/un.h中，用sock-addr\_un结构体表示。各种socket地址结构体的开头都是相同的，前16位表示整个结构体的长度（并不是所有UNIX的实现都有长度字段，如Linux就没有），后16位表示地址类型。IPv4、IPv6和Unix Domain Socket的地址类型分别定义为常数AF\_INET、AF\_INET6、AF\_UNIX。这样，只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体，就可以根据地址类型字段确定结构体中的内容。因此，socket API可以接受各种类型的sockaddr结构体指针做参数，例如bind、accept、connect等函数，这些函数的参数应该设计成void \*类型以便接受各种类型的指针，但是sock API的实现早于ANSI C标准化，那时还没有void \*类型，因此这些函数的参数都用struct sockaddr \*类型表示，在传递参数之前要强制类型转换一下，例如：

struct sockaddr\_in servaddr;

bind(listen\_fd, (struct sockaddr \*)&servaddr, sizeof(servaddr)); /\* initialize servaddr \*/

## 网络套接字函数

### socket模型创建流程图



socket API

### socket函数

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

domain:

AF\_INET 这是大多数用来产生socket的协议，使用TCP或UDP来传输，用IPv4的地址

AF\_INET6 与上面类似，不过是来用IPv6的地址

AF\_UNIX 本地协议，使用在Unix和Linux系统上，一般都是当客户端和服务器在同一台及其上的时候使用

type:

SOCK\_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的socket类型，这个socket是使用TCP来进行传输。

SOCK\_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用UDP来进行它的连接。

SOCK\_SEQPACKET该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。

SOCK\_RAW socket类型提供单一的网络访问，这个socket类型使用ICMP公共协议。（ping、traceroute使用该协议）

SOCK\_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序

protocol:

传0 表示使用默认协议。

返回值：

成功：返回指向新创建的socket的文件描述符，失败：返回-1，设置errno

socket()打开一个网络通讯端口，如果成功的话，就像open()一样返回一个文件描述符，应用程序可以像读写文件一样用read/write在网络上收发数据，如果socket()调用出错则返回-1。对于IPv4，domain参数指定为AF\_INET。对于TCP协议，type参数指定为SOCK\_STREAM，表示面向流的传输协议。如果是UDP协议，则type参数指定为SOCK\_DGRAM，表示面向数据报的传输协议。protocol参数的介绍从略，指定为0即可。

### bind函数

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

sockfd：

socket文件描述符

addr:

构造出IP地址加端口号

addrlen:

sizeof(addr)长度

返回值：

成功返回0，失败返回-1, 设置errno

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。

bind()的作用是将参数sockfd和addr绑定在一起，使sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。前面讲过，struct sockaddr \*是一个通用指针类型，addr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。如：

struct sockaddr\_in servaddr;

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(6666);

首先将整个结构体清零，然后设置地址类型为AF\_INET，**网络地址为INADDR\_ANY，这个宏表示本地的任意IP地址**，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为6666。

### listen函数

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int listen(int sockfd, int backlog);

sockfd:

socket文件描述符

backlog:

排队建立3次握手队列和刚刚建立3次握手队列的链接数和

查看系统默认backlog

cat /proc/sys/net/ipv4/tcp\_max\_syn\_backlog

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未accept的客户端就处于连接等待状态，listen()声明sockfd处于监听状态，并且最多允许有backlog个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回0，失败返回-1。

### accept函数

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

sockdf:

socket文件描述符

addr:

传出参数，返回链接客户端地址信息，含IP地址和端口号

addrlen:

传入传出参数（值-结果）,传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小

返回值：

成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno

三方握手完成后，服务器调用accept()接受连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区addr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给addr参数传NULL，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

n = read(connfd, buf, MAXLINE);

......

close(connfd);

}

整个是一个while死循环，每次循环处理一个客户端连接。由于cliaddr\_len是传入传出参数，每次调用accept()之前应该重新赋初值。accept()的参数listenfd是先前的监听文件描述符，而accept()的返回值是另外一个文件描述符connfd，之后与客户端之间就通过这个connfd通讯，最后关闭connfd断开连接，而不关闭listenfd，再次回到循环开头listenfd仍然用作accept的参数。accept()成功返回一个文件描述符，出错返回-1。

### connect函数

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

sockdf:

socket文件描述符

addr:

传入参数，指定服务器端地址信息，含IP地址和端口号

addrlen:

传入参数,传入sizeof(addr)大小

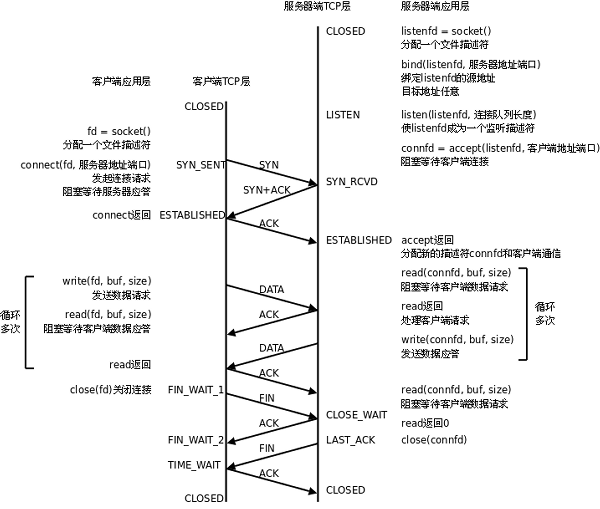
返回值：

成功返回0，失败返回-1，设置errno

客户端需要调用connect()连接服务器，connect和bind的参数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。connect()成功返回0，出错返回-1。

## C/S模型-TCP

下图是基于TCP协议的客户端/服务器程序的一般流程：



TCP协议通讯流程

服务器调用socket()、bind()、listen()完成初始化后，调用accept()阻塞等待，处于监听端口的状态，客户端调用socket()初始化后，调用connect()发出SYN段并阻塞等待服务器应答，服务器应答一个SYN-ACK段，客户端收到后从connect()返回，同时应答一个ACK段，服务器收到后从accept()返回。

数据传输的过程：

建立连接后，TCP协议提供全双工的通信服务，但是一般的客户端/服务器程序的流程是由客户端主动发起请求，服务器被动处理请求，一问一答的方式。因此，服务器从accept()返回后立刻调用read()，读socket就像读管道一样，如果没有数据到达就阻塞等待，这时客户端调用write()发送请求给服务器，服务器收到后从read()返回，对客户端的请求进行处理，在此期间客户端调用read()阻塞等待服务器的应答，服务器调用write()将处理结果发回给客户端，再次调用read()阻塞等待下一条请求，客户端收到后从read()返回，发送下一条请求，如此循环下去。

如果客户端没有更多的请求了，就调用close()关闭连接，就像写端关闭的管道一样，服务器的read()返回0，这样服务器就知道客户端关闭了连接，也调用close()关闭连接。注意，任何一方调用close()后，连接的两个传输方向都关闭，不能再发送数据了。如果一方调用shutdown()则连接处于半关闭状态，仍可接收对方发来的数据。

在学习socket API时要注意应用程序和TCP协议层是如何交互的： 应用程序调用某个socket函数时TCP协议层完成什么动作，比如调用connect()会发出SYN段 应用程序如何知道TCP协议层的状态变化，比如从某个阻塞的socket函数返回就表明TCP协议收到了某些段，再比如read()返回0就表明收到了FIN段

### server

下面通过最简单的客户端/服务器程序的实例来学习socket API。

server.c的作用是从客户端读字符，然后将每个字符转换为大写并回送给客户端。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define MAXLINE 80

#define SERV\_PORT 6666

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

n = read(connfd, buf, MAXLINE);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

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

buf[i] = toupper(buf[i]);

write(connfd, buf, n);

close(connfd);

}

return 0;

}

### client

client.c的作用是从命令行参数中获得一个字符串发给服务器，然后接收服务器返回的字符串并打印。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

char \*str;

if (argc != 2) {

fputs("usage: ./client message\n", stderr);

exit(1);

}

str = argv[1];

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

write(sockfd, str, strlen(str));

n = read(sockfd, buf, MAXLINE);

printf("Response from server:\n");

write(STDOUT\_FILENO, buf, n);

close(sockfd);

return 0;

}

由于客户端不需要固定的端口号，因此不必调用bind()，客户端的端口号由内核自动分配。注意，客户端不是不允许调用bind()，只是没有必要调用bind()固定一个端口号，服务器也不是必须调用bind()，但如果服务器不调用bind()，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。

客户端和服务器启动后可以使用netstat命令查看链接情况：

netstat -apn|grep 6666

## 出错处理封装函数

上面的例子不仅功能简单，而且简单到几乎没有什么错误处理，我们知道，系统调用不能保证每次都成功，必须进行出错处理，这样一方面可以保证程序逻辑正常，另一方面可以迅速得到故障信息。

为使错误处理的代码不影响主程序的可读性，我们把与socket相关的一些系统函数加上错误处理代码包装成新的函数，做成一个模块wrap.c：

### wrap.c

#include <stdlib.h>

#include <errno.h>

#include <sys/socket.h>

void perr\_exit(const char \*s)

{

perror(s);

exit(1);

}

int Accept(int fd, struct sockaddr \*sa, socklen\_t \*salenptr)

{

int n;

again:

if ( (n = accept(fd, sa, salenptr)) < 0) {

if ((errno == ECONNABORTED) || (errno == EINTR))

goto again;

else

perr\_exit("accept error");

}

return n;

}

int Bind(int fd, const struct sockaddr \*sa, socklen\_t salen)

{

int n;

if ((n = bind(fd, sa, salen)) < 0)

perr\_exit("bind error");

return n;

}

int Connect(int fd, const struct sockaddr \*sa, socklen\_t salen)

{

int n;

if ((n = connect(fd, sa, salen)) < 0)

perr\_exit("connect error");

return n;

}

int Listen(int fd, int backlog)

{

int n;

if ((n = listen(fd, backlog)) < 0)

perr\_exit("listen error");

return n;

}

int Socket(int family, int type, int protocol)

{

int n;

if ( (n = socket(family, type, protocol)) < 0)

perr\_exit("socket error");

return n;

}

ssize\_t Read(int fd, void \*ptr, size\_t nbytes)

{

ssize\_t n;

again:

if ( (n = read(fd, ptr, nbytes)) == -1) {

if (errno == EINTR)

goto again;

else

return -1;

}

return n;

}

ssize\_t Write(int fd, const void \*ptr, size\_t nbytes)

{

ssize\_t n;

again:

if ( (n = write(fd, ptr, nbytes)) == -1) {

if (errno == EINTR)

goto again;

else

return -1;

}

return n;

}

int Close(int fd)

{

int n;

if ((n = close(fd)) == -1)

perr\_exit("close error");

return n;

}

ssize\_t Readn(int fd, void \*vptr, size\_t n)

{

size\_t nleft;

ssize\_t nread;

char \*ptr;

ptr = vptr;

nleft = n;

while (nleft > 0) {

if ( (nread = read(fd, ptr, nleft)) < 0) {

if (errno == EINTR)

nread = 0;

else

return -1;

} else if (nread == 0)

break;

nleft -= nread;

ptr += nread;

}

return n - nleft;

}

ssize\_t Writen(int fd, const void \*vptr, size\_t n)

{

size\_t nleft;

ssize\_t nwritten;

const char \*ptr;

ptr = vptr;

nleft = n;

while (nleft > 0) {

if ( (nwritten = write(fd, ptr, nleft)) <= 0) {

if (nwritten < 0 && errno == EINTR)

nwritten = 0;

else

return -1;

}

nleft -= nwritten;

ptr += nwritten;

}

return n;

}

static ssize\_t my\_read(int fd, char \*ptr)

{

static int read\_cnt;

static char \*read\_ptr;

static char read\_buf[100];

if (read\_cnt <= 0) {

again:

if ((read\_cnt = read(fd, read\_buf, sizeof(read\_buf))) < 0) {

if (errno == EINTR)

goto again;

return -1;

} else if (read\_cnt == 0)

return 0;

read\_ptr = read\_buf;

}

read\_cnt--;

\*ptr = \*read\_ptr++;

return 1;

}

ssize\_t Readline(int fd, void \*vptr, size\_t maxlen)

{

ssize\_t n, rc;

char c, \*ptr;

ptr = vptr;

for (n = 1; n < maxlen; n++) {

if ( (rc = my\_read(fd, &c)) == 1) {

\*ptr++ = c;

if (c == '\n')

break;

} else if (rc == 0) {

\*ptr = 0;

return n - 1;

} else

return -1;

}

\*ptr = 0;

return n;

}

### wrap.h

#ifndef \_\_WRAP\_H\_

#define \_\_WRAP\_H\_

void perr\_exit(const char \*s);

int Accept(int fd, struct sockaddr \*sa, socklen\_t \*salenptr);

int Bind(int fd, const struct sockaddr \*sa, socklen\_t salen);

int Connect(int fd, const struct sockaddr \*sa, socklen\_t salen);

int Listen(int fd, int backlog);

int Socket(int family, int type, int protocol);

ssize\_t Read(int fd, void \*ptr, size\_t nbytes);

ssize\_t Write(int fd, const void \*ptr, size\_t nbytes);

int Close(int fd);

ssize\_t Readn(int fd, void \*vptr, size\_t n);

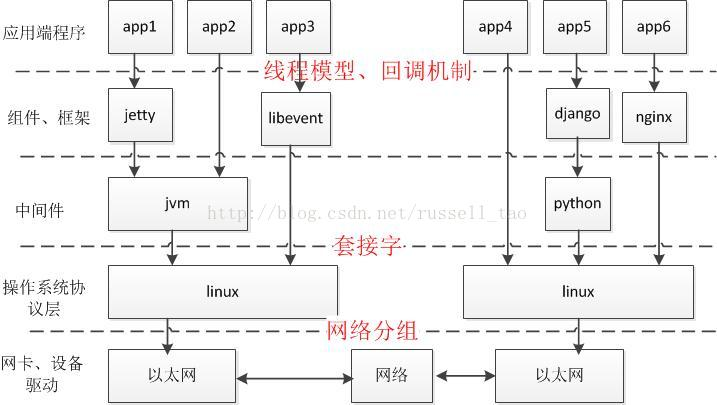
ssize\_t Writen(int fd, const void \*vptr, size\_t n);

ssize\_t my\_read(int fd, char \*ptr);

ssize\_t Readline(int fd, void \*vptr, size\_t maxlen);

#endif

# 高并发服务器



## 多进程并发服务器

使用多进程并发服务器时要考虑以下几点：

1. 父进程最大文件描述个数(父进程中需要close关闭accept返回的新文件描述符)
2. 系统内创建进程个数(与内存大小相关)
3. 进程创建过多是否降低整体服务性能(进程调度)

### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 800

void do\_sigchild(int num)

{

while (waitpid(0, NULL, WNOHANG) > 0)

;

}

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

pid\_t pid;

struct sigaction newact;

newact.sa\_handler = do\_sigchild;

sigemptyset(&newact.sa\_mask);

newact.sa\_flags = 0;

sigaction(SIGCHLD, &newact, NULL);

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

pid = fork();

if (pid == 0) {

Close(listenfd);

while (1) {

n = Read(connfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

}

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

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

buf[i] = toupper(buf[i]);

Write(connfd, buf, n);

}

Close(connfd);

return 0;

} else if (pid > 0) {

Close(connfd);

} else

perr\_exit("fork");

}

Close(listenfd);

return 0;

}

### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

} else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

## 多线程并发服务器

在使用线程模型开发服务器时需考虑以下问题：

1. 调整进程内最大文件描述符上限
2. 线程如有共享数据，考虑线程同步
3. 服务于客户端线程退出时，退出处理。（退出值，分离态）
4. 系统负载，随着链接客户端增加，导致其它线程不能及时得到CPU

### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

struct s\_info {

struct sockaddr\_in cliaddr;

int connfd;

};

void \*do\_work(void \*arg)

{

int n,i;

struct s\_info \*ts = (struct s\_info\*)arg;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

/\* 可以在创建线程前设置线程创建属性,设为分离态,哪种效率高内？ \*/

pthread\_detach(pthread\_self());

while (1) {

n = Read(ts->connfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

}

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &(\*ts).cliaddr.sin\_addr, str, sizeof(str)),

ntohs((\*ts).cliaddr.sin\_port));

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

buf[i] = toupper(buf[i]);

Write(ts->connfd, buf, n);

}

Close(ts->connfd);

}

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

int i = 0;

pthread\_t tid;

struct s\_info ts[256];

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

ts[i].cliaddr = cliaddr;

ts[i].connfd = connfd;

/\* 达到线程最大数时，pthread\_create出错处理, 增加服务器稳定性 \*/

pthread\_create(&tid, NULL, do\_work, (void\*)&ts[i]);

i++;

}

return 0;

}

### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

## 多路I/O转接服务器

多路IO转接服务器也叫做多任务IO服务器。该类服务器实现的主旨思想是，不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。

主要使用的方法有三种

### select

1. select能监听的文件描述符个数受限于FD\_SETSIZE,一般为1024，单纯改变进程打开的文件描述符个数并不能改变select监听文件个数
2. 解决1024以下客户端时使用select是很合适的，但如果链接客户端过多，select采用的是轮询模型，会大大降低服务器响应效率，不应在select上投入更多精力

#include <sys/select.h>

/\* According to earlier standards \*/

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, struct timeval \*timeout);

nfds: 监控的文件描述符集里最大文件描述符加1，因为此参数会告诉内核检测前多少个文件描述符的状态

readfds： 监控有读数据到达文件描述符集合，传入传出参数

writefds： 监控写数据到达文件描述符集合，传入传出参数

exceptfds： 监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数

timeout： 定时阻塞监控时间，3种情况

1.NULL，永远等下去

2.设置timeval，等待固定时间

3.设置timeval里时间均为0，检查描述字后立即返回，轮询

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

void FD\_CLR(int fd, fd\_set \*set); //把文件描述符集合里fd清0

int FD\_ISSET(int fd, fd\_set \*set); //测试文件描述符集合里fd是否置1

void FD\_SET(int fd, fd\_set \*set); //把文件描述符集合里fd位置1

void FD\_ZERO(fd\_set \*set); //把文件描述符集合里所有位清0

#### server

/\* server.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

int i, maxi, maxfd, listenfd, connfd, sockfd;

int nready, client[FD\_SETSIZE]; /\* FD\_SETSIZE 默认为 1024 \*/

ssize\_t n;

fd\_set rset, allset;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN]; /\* #define INET\_ADDRSTRLEN 16 \*/

socklen\_t cliaddr\_len;

struct sockaddr\_in cliaddr, servaddr;

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20); /\* 默认最大128 \*/

maxfd = listenfd; /\* 初始化 \*/

maxi = -1; /\* client[]的下标 \*/

for (i = 0; i < FD\_SETSIZE; i++)

client[i] = -1; /\* 用-1初始化client[] \*/

FD\_ZERO(&allset);

FD\_SET(listenfd, &allset); /\* 构造select监控文件描述符集 \*/

for ( ; ; ) {

rset = allset; /\* 每次循环时都从新设置select监控信号集 \*/

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready < 0)

perr\_exit("select error");

if (FD\_ISSET(listenfd, &rset)) { /\* new client connection \*/

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 0; i < FD\_SETSIZE; i++) {

if (client[i] < 0) {

client[i] = connfd; /\* 保存accept返回的文件描述符到client[]里 \*/

break;

}

}

/\* 达到select能监控的文件个数上限 1024 \*/

if (i == FD\_SETSIZE) {

fputs("too many clients\n", stderr);

exit(1);

}

FD\_SET(connfd, &allset); /\* 添加一个新的文件描述符到监控信号集里 \*/

if (connfd > maxfd)

maxfd = connfd; /\* select第一个参数需要 \*/

if (i > maxi)

maxi = i; /\* 更新client[]最大下标值 \*/

if (--nready == 0)

continue; /\* 如果没有更多的就绪文件描述符继续回到上面select阻塞监听,

负责处理未处理完的就绪文件描述符 \*/

}

for (i = 0; i <= maxi; i++) { /\* 检测哪个clients 有数据就绪 \*/

if ( (sockfd = client[i]) < 0)

continue;

if (FD\_ISSET(sockfd, &rset)) {

if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {

Close(sockfd); /\* 当client关闭链接时，服务器端也关闭对应链接 \*/

FD\_CLR(sockfd, &allset); /\* 解除select监控此文件描述符 \*/

client[i] = -1;

} else {

int j;

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Write(sockfd, buf, n);

}

if (--nready == 0)

break;

}

}

}

close(listenfd);

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

#### pselect

pselect原型如下。此模型应用较少，有需要的同学可参考select模型自行编写C/S

#include <sys/select.h>

int pselect(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, const struct timespec \*timeout,

const sigset\_t \*sigmask);

struct timespec {

long tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

用sigmask替代当前进程的阻塞信号集，调用返回后还原原有阻塞信号集

### poll

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

struct pollfd {

int fd; /\* 文件描述符 \*/

short events; /\* 监控的事件 \*/

short revents; /\* 监控事件中满足条件返回的事件 \*/

};

**POLLIN**  普通或带外优先数据可读,即POLLRDNORM | POLLRDBAND

POLLRDNORM 数据可读

POLLRDBAND 优先级带数据可读

POLLPRI 高优先级可读数据

**POLLOUT**  普通或带外数据可写

POLLWRNORM 数据可写

POLLWRBAND 优先级带数据可写

**POLLERR**  发生错误

POLLHUP 发生挂起

POLLNVAL 描述字不是一个打开的文件

nfds 监控数组中有多少文件描述符需要被监控

timeout 毫秒级等待

-1：阻塞等，#define INFTIM -1 Linux中没有定义此宏

0：立即返回，不阻塞进程

>0：等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值

如果不再监控某个文件描述符时，可以把pollfd中，fd设置为-1，poll不再监控此pollfd，下次返回时，把revents设置为0。

#### server

/\* server.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <poll.h>

#include <errno.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

#define OPEN\_MAX 1024

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

{

int i, j, maxi, listenfd, connfd, sockfd;

int nready;

ssize\_t n;

char buf[MAXLINE], str[INET\_ADDRSTRLEN];

socklen\_t clilen;

struct pollfd client[OPEN\_MAX];

struct sockaddr\_in cliaddr, servaddr;

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

client[0].fd = listenfd;

client[0].events = POLLRDNORM; /\* listenfd监听普通读事件 \*/

for (i = 1; i < OPEN\_MAX; i++)

client[i].fd = -1; /\* 用-1初始化client[]里剩下元素 \*/

maxi = 0; /\* client[]数组有效元素中最大元素下标 \*/

for ( ; ; ) {

nready = poll(client, maxi+1, -1); /\* 阻塞 \*/

if (client[0].revents & POLLRDNORM) { /\* 有客户端链接请求 \*/

clilen = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &clilen);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 1; i < OPEN\_MAX; i++) {

if (client[i].fd < 0) {

client[i].fd = connfd; /\* 找到client[]中空闲的位置，存放accept返回的connfd \*/

break;

}

}

if (i == OPEN\_MAX)

perr\_exit("too many clients");

client[i].events = POLLRDNORM; /\* 设置刚刚返回的connfd，监控读事件 \*/

if (i > maxi)

maxi = i; /\* 更新client[]中最大元素下标 \*/

if (--nready <= 0)

continue; /\* 没有更多就绪事件时,继续回到poll阻塞 \*/

}

for (i = 1; i <= maxi; i++) { /\* 检测client[] \*/

if ((sockfd = client[i].fd) < 0)

continue;

if (client[i].revents & (POLLRDNORM | POLLERR)) {

if ((n = Read(sockfd, buf, MAXLINE)) < 0) {

if (errno == ECONNRESET) { /\* 当收到 RST标志时 \*/

/\* connection reset by client \*/

printf("client[%d] aborted connection\n", i);

Close(sockfd);

client[i].fd = -1;

} else {

perr\_exit("read error");

}

} else if (n == 0) {

/\* connection closed by client \*/

printf("client[%d] closed connection\n", i);

Close(sockfd);

client[i].fd = -1;

} else {

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Writen(sockfd, buf, n);

}

if (--nready <= 0)

break; /\* no more readable descriptors \*/

}

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

#### ppoll

GNU定义了ppoll（非POSIX标准），可以支持设置信号屏蔽字，大家可参考poll模型自行实现C/S。

#define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

#include <poll.h>

int ppoll(struct pollfd \*fds, nfds\_t nfds,

const struct timespec \*timeout\_ts, const sigset\_t \*sigmask);

### epoll

epoll是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。

目前epell是linux大规模并发网络程序中的热门首选模型。

epoll除了提供select/poll那种IO事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。

可以使用cat命令查看一个进程可以打开的socket描述符上限。

cat /proc/sys/fs/file-max

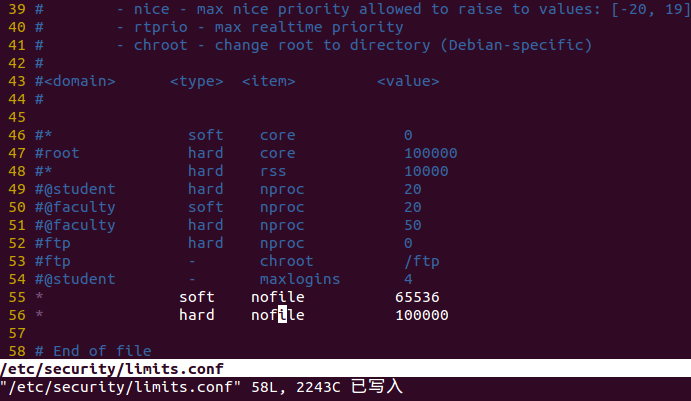
如有需要，可以通过修改配置文件的方式修改该上限值。

sudo vi /etc/security/limits.conf

在文件尾部写入以下配置,soft软限制，hard硬限制。如下图所示。

\* soft nofile 65536

\* hard nofile 100000



#### 基础API

1. 创建一个epoll句柄，参数size用来告诉内核监听的文件描述符的个数，跟内存大小有关。

#include <sys/epoll.h>

int epoll\_create(int size) size：监听数目

1. 控制某个epoll监控的文件描述符上的事件：注册、修改、删除。

#include <sys/epoll.h>

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)

epfd： 为epoll\_creat的句柄

op： 表示动作，用3个宏来表示：

EPOLL\_CTL\_ADD (注册新的fd到epfd)，

EPOLL\_CTL\_MOD (修改已经注册的fd的监听事件)，

EPOLL\_CTL\_DEL (从epfd删除一个fd)；

event： 告诉内核需要监听的事件

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

typedef union epoll\_data {

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

**EPOLLIN** ： 表示对应的文件描述符可以读（包括对端SOCKET正常关闭）

**EPOLLOUT**： 表示对应的文件描述符可以写

EPOLLPRI： 表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）

**EPOLLERR**： 表示对应的文件描述符发生错误

EPOLLHUP： 表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

1. 等待所监控文件描述符上有事件的产生，类似于select()调用。

#include <sys/epoll.h>

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout)

events： 用来存内核得到事件的集合，

maxevents： 告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size，

timeout： 是超时时间

-1： 阻塞

0： 立即返回，非阻塞

>0： 指定毫秒

返回值： 成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1

#### server

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/epoll.h>

#include <errno.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

#define OPEN\_MAX 1024

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

{

int i, j, maxi, listenfd, connfd, sockfd;

int nready, efd, res;

ssize\_t n;

char buf[MAXLINE], str[INET\_ADDRSTRLEN];

socklen\_t clilen;

int client[OPEN\_MAX];

struct sockaddr\_in cliaddr, servaddr;

struct epoll\_event tep, ep[OPEN\_MAX];

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*) &servaddr, sizeof(servaddr));

Listen(listenfd, 20);

for (i = 0; i < OPEN\_MAX; i++)

client[i] = -1;

maxi = -1;

efd = epoll\_create(OPEN\_MAX);

if (efd == -1)

perr\_exit("epoll\_create");

tep.events = EPOLLIN; tep.data.fd = listenfd;

res = epoll\_ctl(efd, EPOLL\_CTL\_ADD, listenfd, &tep);

if (res == -1)

perr\_exit("epoll\_ctl");

while (1) {

nready = epoll\_wait(efd, ep, OPEN\_MAX, -1); /\* 阻塞监听 \*/

if (nready == -1)

perr\_exit("epoll\_wait");

for (i = 0; i < nready; i++) {

if (!(ep[i].events & EPOLLIN))

continue;

if (ep[i].data.fd == listenfd) {

clilen = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &clilen);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (j = 0; j < OPEN\_MAX; j++) {

if (client[j] < 0) {

client[j] = connfd; /\* save descriptor \*/

break;

}

}

if (j == OPEN\_MAX)

perr\_exit("too many clients");

if (j > maxi)

maxi = j; /\* max index in client[] array \*/

tep.events = EPOLLIN;

tep.data.fd = connfd;

res = epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &tep);

if (res == -1)

perr\_exit("epoll\_ctl");

} else {

sockfd = ep[i].data.fd;

n = Read(sockfd, buf, MAXLINE);

if (n == 0) {

for (j = 0; j <= maxi; j++) {

if (client[j] == sockfd) {

client[j] = -1;

break;

}

}

res = epoll\_ctl(efd, EPOLL\_CTL\_DEL, sockfd, NULL);

if (res == -1)

perr\_exit("epoll\_ctl");

Close(sockfd);

printf("client[%d] closed connection\n", j);

} else {

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Writen(sockfd, buf, n);

}

}

}

}

close(listenfd);

close(efd);

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

## epoll进阶

### 事件模型

EPOLL事件有两种模型：

Edge Triggered (ET) 边缘触发只有数据到来才触发，不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发只要有数据都会触发。

思考如下步骤：

1. 假定我们已经把一个用来从管道中读取数据的文件描述符(RFD)添加到epoll描述符。
2. 管道的另一端写入了2KB的数据
3. 调用epoll\_wait，并且它会返回RFD，说明它已经准备好读取操作
4. 读取1KB的数据
5. 调用epoll\_wait……

在这个过程中，有两种工作模式：

#### ET模式

ET模式即Edge Triggered工作模式。

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。

1. 基于非阻塞文件句柄
2. 只有当read或者write返回EAGAIN(非阻塞读，暂时无数据)时才需要挂起、等待。但这并不是说每次read时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

#### LT模式

LT模式即Level Triggered工作模式。

与ET模式不同的是，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll，无论后面的数据是否被使用。

LT(level triggered)：LT是缺省的工作方式，并且同时支持block和no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。

ET(edge-triggered)：ET是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once).

### 实例一：

基于管道epoll ET触发模式

#include <stdio.h>

#include <stdlib.h>

#include <sys/epoll.h>

#include <errno.h>

#include <unistd.h>

#define MAXLINE 10

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

{

int efd, i;

int pfd[2];

pid\_t pid;

char buf[MAXLINE], ch = 'a';

pipe(pfd);

pid = fork();

if (pid == 0) {

close(pfd[0]);

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(pfd[1], buf, sizeof(buf));

sleep(2);

}

close(pfd[1]);

} else if (pid > 0) {

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

close(pfd[1]);

efd = epoll\_create(10);

/\* event.events = EPOLLIN; \*/

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

event.data.fd = pfd[0];

epoll\_ctl(efd, EPOLL\_CTL\_ADD, pfd[0], &event);

while (1) {

res = epoll\_wait(efd, resevent, 10, -1);

printf("res %d\n", res);

if (resevent[0].data.fd == pfd[0]) {

len = read(pfd[0], buf, MAXLINE/2);

write(STDOUT\_FILENO, buf, len);

}

}

close(pfd[0]);

close(efd);

} else {

perror("fork");

exit(-1);

}

return 0;

}

### 实例二：

基于网络C/S模型的epoll ET触发模式

#### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/epoll.h>

#include <unistd.h>

#define MAXLINE 10

#define SERV\_PORT 8080

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, efd;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

efd = epoll\_create(10);

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

printf("Accepting connections ...\n");

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

event.data.fd = connfd;

epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &event);

while (1) {

res = epoll\_wait(efd, resevent, 10, -1);

printf("res %d\n", res);

if (resevent[0].data.fd == connfd) {

len = read(connfd, buf, MAXLINE/2);

write(STDOUT\_FILENO, buf, len);

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#define MAXLINE 10

#define SERV\_PORT 8080

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, i;

char ch = 'a';

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(sockfd, buf, sizeof(buf));

sleep(10);

}

Close(sockfd);

return 0;

}

### 实例三：

基于网络C/S非阻塞模型的epoll ET触发模式

#### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/epoll.h>

#include <unistd.h>

#include <fcntl.h>

#define MAXLINE 10

#define SERV\_PORT 8080

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, efd, flag;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

efd = epoll\_create(10);

/\* event.events = EPOLLIN; \*/

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

printf("Accepting connections ...\n");

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

flag = fcntl(connfd, F\_GETFL);

flag |= O\_NONBLOCK;

fcntl(connfd, F\_SETFL, flag);

event.data.fd = connfd;

epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &event);

while (1) {

printf("epoll\_wait begin\n");

res = epoll\_wait(efd, resevent, 10, -1);

printf("epoll\_wait end res %d\n", res);

if (resevent[0].data.fd == connfd) {

while ((len = read(connfd, buf, MAXLINE/2)) > 0)

write(STDOUT\_FILENO, buf, len);

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#define MAXLINE 10

#define SERV\_PORT 8080

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, i;

char ch = 'a';

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(sockfd, buf, sizeof(buf));

sleep(10);

}

Close(sockfd);

return 0;

}

## 线程池并发服务器

1. 预先创建阻塞于accept多线程，使用互斥锁上锁保护accept
2. 预先创建多线程，由主线程调用accept

## UDP服务器

传输层主要应用的协议模型有两种，一种是TCP协议，另外一种则是UDP协议。TCP协议在网络通信中占主导地位，绝大多数的网络通信借助TCP协议完成数据传输。但UDP也是网络通信中不可或缺的重要通信手段。

相较于TCP而言，UDP通信的形式更像是发短信。不需要在数据传输之前建立、维护连接。只专心获取数据就好。省去了三次握手的过程，通信速度可以大大提高，但与之伴随的通信的稳定性和正确率便得不到保证。因此，我们称UDP为“无连接的不可靠报文传递”。

那么与我们熟知的TCP相比，UDP有哪些优点和不足呢？由于无需创建连接，所以UDP开销较小，数据传输速度快，实时性较强。多用于对实时性要求较高的通信场合，如视频会议、电话会议等。但随之也伴随着数据传输不可靠，传输数据的正确率、传输顺序和流量都得不到控制和保证。所以，通常情况下，使用UDP协议进行数据传输，为保证数据的正确性，我们需要在应用层添加辅助校验协议来弥补UDP的不足，以达到数据可靠传输的目的。

与TCP类似的，UDP也有可能出现缓冲区被填满后，再接收数据时丢包的现象。由于它没有TCP滑动窗口的机制，通常采用如下两种方法解决：

1. 服务器应用层设计流量控制，控制发送数据速度。
2. 借助setsockopt函数改变接收缓冲区大小。如：

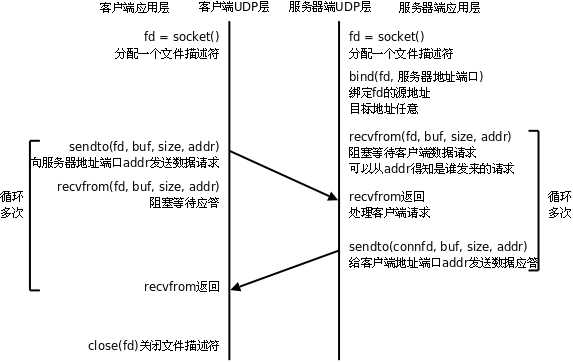
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

int n = 220x1024

setsockopt(sockfd, SOL\_SOCKET, SO\_RCVBUF, &n, sizeof(n));

## C/S模型-UDP



UDP处理模型

由于UDP不需要维护连接，程序逻辑简单了很多，但是UDP协议是不可靠的，保证通讯可靠性的机制需要在应用层实现。

编译运行server，在两个终端里各开一个client与server交互，看看server是否具有并发服务的能力。用Ctrl+C关闭server，然后再运行server，看此时client还能否和server联系上。和前面TCP程序的运行结果相比较，体会无连接的含义。

### server

#include <string.h>

#include <netinet/in.h>

#include <stdio.h>

#include <unistd.h>

#include <strings.h>

#include <arpa/inet.h>

#include <ctype.h>

#define MAXLINE 80

#define SERV\_PORT 6666

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int sockfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

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

servaddr.sin\_port = htons(SERV\_PORT);

bind(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

n = recvfrom(sockfd, buf, MAXLINE,0, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

if (n == -1)

perror("recvfrom error");

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

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

buf[i] = toupper(buf[i]);

n = sendto(sockfd, buf, n, 0, (struct sockaddr \*)&cliaddr, sizeof(cliaddr));

if (n == -1)

perror("sendto error");

}

close(sockfd);

return 0;

}

### client

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <strings.h>

#include <ctype.h>

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

int sockfd, n;

char buf[MAXLINE];

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

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

while (fgets(buf, MAXLINE, stdin) != NULL) {

n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr \*)&servaddr, sizeof(servaddr));

if (n == -1)

perror("sendto error");

n = recvfrom(sockfd, buf, MAXLINE, 0, NULL, 0);

if (n == -1)

perror("recvfrom error");

write(STDOUT\_FILENO, buf, n);

}

close(sockfd);

return 0;

}

## 多播(组播)

组播组可以是永久的也可以是临时的。组播组地址中，有一部分由官方分配的，称为永久组播组。永久组播组保持不变的是它的ip地址，组中的成员构成可以发生变化。永久组播组中成员的数量都可以是任意的，甚至可以为零。那些没有保留下来供永久组播组使用的ip组播地址，可以被临时组播组利用。

224.0.0.0～224.0.0.255 为预留的组播地址（永久组地址），地址224.0.0.0保留不做分配，其它地址供路由协议使用；

224.0.1.0～224.0.1.255 是公用组播地址，可以用于Internet；欲使用需申请。

224.0.2.0～238.255.255.255 为用户可用的组播地址（临时组地址），全网范围内有效；

239.0.0.0～239.255.255.255 为本地管理组播地址，仅在特定的本地范围内有效。

可使用ip ad命令查看网卡编号，如：

itcast$ ip ad

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo\_fast state DOWN group default qlen 1000

link/ether 00:0c:29:0a:c4:f4 brd ff:ff:ff:ff:ff:ff

inet6 fe80::20c:29ff:fe0a:c4f4/64 scope link

valid\_lft forever preferred\_lft forever

if\_nametoindex 命令可以根据网卡名，获取网卡序号。

### server

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <net/if.h>

#define SERVER\_PORT 6666

#define CLIENT\_PORT 9000

#define MAXLINE 1500

#define GROUP "239.0.0.2"

int main(void)

{

int sockfd, i ;

struct sockaddr\_in serveraddr, clientaddr;

char buf[MAXLINE] = "itcast\n";

char ipstr[INET\_ADDRSTRLEN]; /\* 16 Bytes \*/

socklen\_t clientlen;

ssize\_t len;

struct ip\_mreqn group;

/\* 构造用于UDP通信的套接字 \*/

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

bzero(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET; /\* IPv4 \*/

serveraddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\* 本地任意IP INADDR\_ANY = 0 \*/

serveraddr.sin\_port = htons(SERVER\_PORT);

bind(sockfd, (struct sockaddr \*)&serveraddr, sizeof(serveraddr));

/\*设置组地址\*/

inet\_pton(AF\_INET, GROUP, &group.imr\_multiaddr);

/\*本地任意IP\*/

inet\_pton(AF\_INET, "0.0.0.0", &group.imr\_address);

/\* eth0 --> 编号 命令：ip ad \*/

group.imr\_ifindex = if\_nametoindex("eth0");

setsockopt(sockfd, IPPROTO\_IP, IP\_MULTICAST\_IF, &group, sizeof(group));

/\*构造 client 地址 IP+端口 \*/

bzero(&clientaddr, sizeof(clientaddr));

clientaddr.sin\_family = AF\_INET; /\* IPv4 \*/

inet\_pton(AF\_INET, GROUP, &clientaddr.sin\_addr.s\_addr);

clientaddr.sin\_port = htons(CLIENT\_PORT);

while (1) {

//fgets(buf, sizeof(buf), stdin);

sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr \*)&clientaddr, sizeof(clientaddr));

sleep(1);

}

close(sockfd);

return 0;

}

### client

#include <netinet/in.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <string.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>

#include <net/if.h>

#define SERVER\_PORT 6666

#define MAXLINE 4096

#define CLIENT\_PORT 9000

#define GROUP "239.0.0.2"

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

{

struct sockaddr\_in serveraddr, localaddr;

int confd;

ssize\_t len;

char buf[MAXLINE];

/\* 定义组播结构体 \*/

struct ip\_mreqn group;

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

//初始化本地端地址

bzero(&localaddr, sizeof(localaddr));

localaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "0.0.0.0" , &localaddr.sin\_addr.s\_addr);

localaddr.sin\_port = htons(CLIENT\_PORT);

bind(confd, (struct sockaddr \*)&localaddr, sizeof(localaddr));

/\*设置组地址\*/

inet\_pton(AF\_INET, GROUP, &group.imr\_multiaddr);

/\*本地任意IP\*/

inet\_pton(AF\_INET, "0.0.0.0", &group.imr\_address);

/\* eth0 --> 编号 命令：ip ad \*/

group.imr\_ifindex = if\_nametoindex("eth0");

/\*设置client 加入多播组 \*/

setsockopt(confd, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP, &group, sizeof(group));

while (1) {

len = recvfrom(confd, buf, sizeof(buf), 0, NULL, 0);

write(STDOUT\_FILENO, buf, len);

}

close(confd);

return 0;

}

## socket IPC（本地套接字domain）

socket API原本是为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制，就是UNIX Domain Socket。虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。

UNIX Domain Socket是全双工的，API接口语义丰富，相比其它IPC机制有明显的优越性，目前已成为使用最广泛的IPC机制，比如X Window服务器和GUI程序之间就是通过UNIXDomain Socket通讯的。

使用UNIX Domain Socket的过程和网络socket十分相似，也要先调用socket()创建一个socket文件描述符，address family指定为AF\_UNIX，type可以选择SOCK\_DGRAM或SOCK\_STREAM，protocol参数仍然指定为0即可。

UNIX Domain Socket与网络socket编程最明显的不同在于地址格式不同，用结构体sockaddr\_un表示，网络编程的socket地址是IP地址加端口号，而UNIX Domain Socket的地址是一个socket类型的文件在文件系统中的路径，这个socket文件由bind()调用创建，如果调用bind()时该文件已存在，则bind()错误返回。

对比网络套接字地址结构和本地套接字地址结构：

struct sockaddr\_in {

\_\_kernel\_sa\_family\_t **sin\_family**; /\* Address family \*/ 地址结构类型

\_\_be16 **sin\_port**; /\* Port number \*/ 端口号

struct in\_addr **sin\_addr**; /\* Internet address \*/ IP地址

};

struct sockaddr\_un {

\_\_kernel\_sa\_family\_t **sun\_family**; /\* AF\_UNIX \*/ 地址结构类型

char **sun\_path**[UNIX\_PATH\_MAX]; /\* pathname \*/ socket文件名(含路径)

};

以下程序将UNIX Domain socket绑定到一个地址。

size = offsetof(struct sockaddr\_un, sun\_path) + strlen(un.sun\_path);

#define offsetof(type, member) ((int)&((type \*)0)->MEMBER)

### server

#include <stdlib.h>

#include <stdio.h>

#include <stddef.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <errno.h>

#define QLEN 10

/\*

\* Create a server endpoint of a connection.

\* Returns fd if all OK, <0 on error.

\*/

int serv\_listen(const char \*name)

{

int fd, len, err, rval;

struct sockaddr\_un un;

/\* create a UNIX domain stream socket \*/

if ((fd = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0)

return(-1);

/\* in case it already exists \*/

unlink(name);

/\* fill in socket address structure \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

strcpy(un.sun\_path, name);

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(name);

/\* bind the name to the descriptor \*/

if (bind(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -2;

goto errout;

}

if (listen(fd, QLEN) < 0) { /\* tell kernel we're a server \*/

rval = -3;

goto errout;

}

return(fd);

errout:

err = errno;

close(fd);

errno = err;

return(rval);

}

int serv\_accept(int listenfd, uid\_t \*uidptr)

{

int clifd, len, err, rval;

time\_t staletime;

struct sockaddr\_un un;

struct stat statbuf;

len = sizeof(un);

if ((clifd = accept(listenfd, (struct sockaddr \*)&un, &len)) < 0)

return(-1); /\* often errno=EINTR, if signal caught \*/

/\* obtain the client's uid from its calling address \*/

len -= offsetof(struct sockaddr\_un, sun\_path); /\* len of pathname \*/

un.sun\_path[len] = 0; /\* null terminate \*/

if (stat(un.sun\_path, &statbuf) < 0) {

rval = -2;

goto errout;

}

if (S\_ISSOCK(statbuf.st\_mode) == 0) {

rval = -3; /\* not a socket \*/

goto errout;

}

if (uidptr != NULL)

\*uidptr = statbuf.st\_uid; /\* return uid of caller \*/

/\* we're done with pathname now \*/

unlink(un.sun\_path);

return(clifd);

errout:

err = errno;

close(clifd);

errno = err;

return(rval);

}

int main(void)

{

int lfd, cfd, n, i;

uid\_t cuid;

char buf[1024];

lfd = serv\_listen("foo.socket");

if (lfd < 0) {

switch (lfd) {

case -3:perror("listen"); break;

case -2:perror("bind"); break;

case -1:perror("socket"); break;

}

exit(-1);

}

cfd = serv\_accept(lfd, &cuid);

if (cfd < 0) {

switch (cfd) {

case -3:perror("not a socket"); break;

case -2:perror("a bad filename"); break;

case -1:perror("accept"); break;

}

exit(-1);

}

while (1) {

r\_again:

n = read(cfd, buf, 1024);

if (n == -1) {

if (errno == EINTR)

goto r\_again;

}

else if (n == 0) {

printf("the other side has been closed.\n");

break;

}

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

buf[i] = toupper(buf[i]);

write(cfd, buf, n);

}

close(cfd);

close(lfd);

return 0;

}

### client

#include <stdio.h>

#include <stdlib.h>

#include <stddef.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <errno.h>

#define CLI\_PATH "/var/tmp/" /\* +5 for pid = 14 chars \*/

/\*

\* Create a client endpoint and connect to a server.

\* Returns fd if all OK, <0 on error.

\*/

int cli\_conn(const char \*name)

{

int fd, len, err, rval;

struct sockaddr\_un un;

/\* create a UNIX domain stream socket \*/

if ((fd = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0)

return(-1);

/\* fill socket address structure with our address \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

sprintf(un.sun\_path, "%s%05d", CLI\_PATH, getpid());

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(un.sun\_path);

/\* in case it already exists \*/

unlink(un.sun\_path);

if (bind(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -2;

goto errout;

}

/\* fill socket address structure with server's address \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

strcpy(un.sun\_path, name);

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(name);

if (connect(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -4;

goto errout;

}

return(fd);

errout:

err = errno;

close(fd);

errno = err;

return(rval);

}

int main(void)

{

int fd, n;

char buf[1024];

fd = cli\_conn("foo.socket");

if (fd < 0) {

switch (fd) {

case -4:perror("connect"); break;

case -3:perror("listen"); break;

case -2:perror("bind"); break;

case -1:perror("socket"); break;

}

exit(-1);

}

while (fgets(buf, sizeof(buf), stdin) != NULL) {

write(fd, buf, strlen(buf));

n = read(fd, buf, sizeof(buf));

write(STDOUT\_FILENO, buf, n);

}

close(fd);

return 0;

}

## 其它常用函数

### 名字与地址转换

gethostbyname根据给定的主机名，获取主机信息。

过时，仅用于IPv4，且线程不安全。

#include <stdio.h>

#include <netdb.h>

#include <arpa/inet.h>

extern int h\_errno;

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

{

struct hostent \*host;

char str[128];

host = gethostbyname(argv[1]);

printf("%s\n", host->h\_name);

while (\*(host->h\_aliases) != NULL)

printf("%s\n", \*host->h\_aliases++);

switch (host->h\_addrtype) {

case AF\_INET:

while (\*(host->h\_addr\_list) != NULL)

printf("%s\n", inet\_ntop(AF\_INET, (\*host->h\_addr\_list++), str, sizeof(str)));

break;

default:

printf("unknown address type\n");

break;

}

return 0;

}

gethostbyaddr函数。

此函数只能获取域名解析服务器的url和/etc/hosts里登记的IP对应的域名。

#include <stdio.h>

#include <netdb.h>

#include <arpa/inet.h>

extern int h\_errno;

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

{

struct hostent \*host;

char str[128];

struct in\_addr addr;

inet\_pton(AF\_INET, argv[1], &addr);

host = gethostbyaddr((char \*)&addr, 4, AF\_INET);

printf("%s\n", host->h\_name);

while (\*(host->h\_aliases) != NULL)

printf("%s\n", \*host->h\_aliases++);

switch (host->h\_addrtype) {

case AF\_INET:

while (\*(host->h\_addr\_list) != NULL)

printf("%s\n", inet\_ntop(AF\_INET, (\*host->h\_addr\_list++), str, sizeof(str)));

break;

default:

printf("unknown address type\n");

break;

}

return 0;

}

getservbyname

getservbyport

根据服务程序名字或端口号获取信息。使用频率不高。

getaddrinfo

getnameinfo

freeaddrinfo

可同时处理IPv4和IPv6，线程安全的。

### 套接口和地址关联

getsockname

根据accpet返回的sockfd，得到临时端口号

getpeername

根据accpet返回的sockfd，得到远端链接的端口号，在exec后可以获取客户端信息。