Linux网络编程

1. socket简介

socket是网络编程的一个接口，它是一种特殊的I/O，用socket函数建立一个socket连接，函数返回一个整型的socket描述符，随后进行数据传输。

socket分为三种类型：流式socket，数据报socket和原始socket。

1. socket套接口简介

套接结构包含IP地址和端口信息。一般套接口地址结构都以“sockaddr”开头。socket根据所使用的的协议的不同，可以分为TCP套接口和UDP套接口，有时也称为流式套接口和数据套接口。

1. socket套接口的数据结构

sockaddr和sockaddr\_in。两个结构类型都是用来保存socket信息的，如IP地址，通信端口等。

|  |
| --- |
| struct sockaddr {  unsigned short sa\_family; /\* address family, AF\_xxx \*/  /\*sa\_family一般为AF\_INET，代表Internet（TCP/IP）地址族\*/  char sa\_data[14]; /\* 14 bytes of protocol address \*/  /\*sa\_data则包含该socket的IP地址和端口号\*/  　　}; |

|  |
| --- |
| struct sockaddr\_in {  　　short int sin\_family; /\*地址族 \*/  　　unsigned short int sin\_port; /\* 端口号 \*/  　　struct in\_addr sin\_addr; /\* IP地址 \*/  　　unsigned char sin\_zero[8]; /\* 填充0以保持与struct sockaddr同样的大小 \*/  　　};  　　sin\_family：指代协议族，在socket编程中只能是AF\_INET  　　sin\_port：存储端口号（使用网络字节顺序）  　　sin\_addr：存储IP地址，使用in\_addr这个数据结构  　　sin\_zero：是为了让sockaddr与sockaddr\_in两个数据结构保持大小相同而保留的空字节，可以用bzero()函数将其置0。  而其中in\_addr结构的定义如下：  typedef struct in\_addr {  　　union {  　　struct{ unsigned char s\_b1,s\_b2, s\_b3,s\_b4;} S\_un\_b;  　　struct{ unsigned short s\_w1, s\_w2;} S\_un\_w;  　　unsigned long S\_addr;  　　} S\_un;  　　} IN\_ADDR; |

1. TCP编程

网络上绝大多数的通信服务采用服务器机制（Client/Server），TCP提供的是一种可靠的、面向连接的服务。

基于TCP网络编程常用函数及其功能

|  |  |
| --- | --- |
| 函数名 | 功能 |
| bind | bind()将socket与本机的一个端口绑定，随后就可以在该端口监听服务请求。 |
| connect | 面向连接的客户程序使用connect()函数来配置socket，并与远程服务器建立一个TCP连接。 |
| listen | listen()函数使socket处于被动的监听模式，并为该socket建立一个输入数据队列，将到达的服务请求保存在队列中，直到程序处理它们。 |
| accept | accept()函数让服务器接收客户的连接请求。 |
| close | 停止在该socket上的任何数据请求。 |
| send | 数据发送函数。 |
| recv | 数据接收函数。 |

例. 服务器通过socket连接后，向客户端发送字符串“连接上了”。在服务器上显示客户端的IP地址或域名。



程序中主要语句说明如下：

1. 服务器端
2. 建立socket。程序调用socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个socket数据结构分配存储空间。用语句实现：

|  |
| --- |
| socket(AF\_INET,SOCK\_STREAM,0); |

其中，参数AF\_INET表示采用IPv4协议进行通信；参数SOCK\_STREAM表示采用流式socket，即TCP。

1. 绑定bind。将socket与本机上的一个端口绑定，随后就可以在该端口监听服务请求。用语句实现：

|  |
| --- |
| bind(sockfd,(struct sockaddr \*)&my\_addr,sizeof(struct sockaddr)); |

其中，参数my\_addr表示指向包含本机IP地址及端口号等信息。

1. 建立监听listen。使socket出于被动的监听模式，并为该socket建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们。用语句实现：

|  |
| --- |
| listen(sockfd,BACKLOG); |

其中，参数BACKLOG表示最大连接数。

1. 响应客户请求accept。用**函数accept生成一个新的套接口描述符**，让服务器接收客户的连接请求。用语句实现：

|  |
| --- |
| accept(sockfd,(struct sockaddr \*)&remote\_addr,&sin\_size); |

其中，参数remote\_addr用于接收客户端地址信息；参数sin\_size用于存放地址的长度。

1. 发送数据send。该函数用于在面向连接的socket上进行数据发送。用语句实现：

|  |
| --- |
| send(client\_fd,’连接上了 \n’,26,0); |

其中，26表示以字节为单位的长度。

1. 关闭close。停止在该socket上的任何数据操作。用语句实现：

|  |
| --- |
| close(client\_fd); |

1. 客户端
2. 建立socket。程序调用socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个socket数据结构分配存储空间。用语句实现：

|  |
| --- |
| socket(AF\_INET,SOCKET\_STREAM,0); |

1. 请求连接connect。启动和远端主机的直接连接。用语句实现：

|  |
| --- |
| connect(sockfd, (struct sockaddr\*) &serv\_addr, sizeof(struct sockaddr)); |

1. 接收数据recv。该函数用于面向连接的socket上进行数据接收。用语句实现：

|  |
| --- |
| recv(sockfd,buf,MAXDATASIZE,0); |

1. 关闭close。停止在该socket上的任何数据操作。用语句实现：

|  |
| --- |
| close(sockfd); |

服务器端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: main.c  \* Author: Mr Mahaitao  \*  \* Created on March 6, 2018, 3:44 AM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <sys/types.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <sys/wait.h>  #include <arpa/inet.h>  #define SERVPORT 3333 /\*服务器监听端口号\*/  #define BACKLOG 10 /\*最大同时连接请求数\*/  /\*  \*  \*/  int main(int argc, char\*\* argv) {  int sockfd, client\_fd; /\*sock\_fd:监听socket; client\_fd:数据传输socket\*/  struct sockaddr\_in my\_addr; /\*本机地址信息\*/  struct sockaddr\_in remote\_addr; /\*客户端信息\*/  int sin\_size;  if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {  perror("socket 创建失败！");  exit(1);  }  my\_addr.sin\_family = AF\_INET;  my\_addr.sin\_port = htons(SERVPORT); /\*htons()把16位值从主机字节序转换成网络字节序\*/  my\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\*IP地址设置成INADDR\_ANY,让系统自动获取本机的IP地址\*/  bzero(&(my\_addr.sin\_zero), 8); /\*保持与struct sockaddr同样大小\*/  if (bind(sockfd, (struct sockaddr \*) &my\_addr, sizeof (struct sockaddr)) == -1) {  perror("bind 失败！");  exit(1);  }  if (listen(sockfd, BACKLOG) == -1) {  perror("lisnten 失败！");  exit(1);  }  while (1) {  sin\_size = sizeof (struct sockaddr\_in);  if (client\_fd = accept(sockfd, (struct sockaddr \*) &remote\_addr, &sin\_size) == -1) {  perror("accept 失败!");  continue;  }  printf("收到一个来自：%s\n", inet\_ntoa(remote\_addr.sin\_addr));  if (!fork()) { /\*子进程代码段\*/  if (send(client\_fd, "连接上了 \n", 26, 0) == -1)  perror("send 失败！");  close(client\_fd);  exit(0);  }  close(client\_fd);  }  return (EXIT\_SUCCESS);  } |

注意：代码实例中的fork()函数生成一个子进程来处理数据传输部分，fork()语句对于子进程返回的值为0。所以，包含fork函数的if语句是子进程代码部分，它与if语句后面的父进程代码是并发执行的。

客户端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: client.c  \* Author: Mr Mahaitao  \*  \* Created on March 6, 2018, 5:21 AM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <netdb.h>  #include <sys/types.h>  #include <netinet/in.h>  #include <sys/socket.h>  #define SERVPORT 3333  #define MAXDATASIZE 100 /\*每次最大数据传输量\*/  /\*  \*  \*/  int main(int argc, char\*\* argv) {  int sockfd, recvbytes;  char buf[MAXDATASIZE];  struct hostent \*host;  struct sockaddr\_in serv\_addr;  if (argc < 2) {  fprintf(stderr, "Please enter the server's hostname!\n");  exit(1);  }  if ((host = gethostbyname(argv[1])) == NULL) {  herror("gethostbyname error! ");  exit(1);  }  if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {  perror("socket create error! ");  exit(1);  }  serv\_addr.sin\_family = AF\_INET;  serv\_addr.sin\_port = htons(SERVPORT);  serv\_addr.sin\_addr = \*((struct in\_addr \*) host->h\_addr);  bzero(&(serv\_addr.sin\_zero), 8);  if (connect(sockfd, (struct sockaddr \*) &serv\_addr, sizeof (struct sockaddr)) == -1) {  perror("connect error! ");  exit(1);  }  if ((recvbytes = recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {  perror("recv error！");  exit(1);  }  buf[recvbytes] = '\0';  printf("收到： %s", buf);  close(sockfd);  return (EXIT\_SUCCESS);  } |

相关函数介绍：

socket建立的函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | socket的建立 |
| 函数原型 | int socket(int domain, int type, int protocol); |
| 函数传入值 | domain AF\_INET：Ipv4 协议  AF\_INET6：Ipv6 协议  AF\_LOCAL：UNIX域协议  AF\_ROUTE：路由套接字  AF\_KEY：秘钥套接字  type SOCK\_STREAM：字节数据流套接字  SOCK\_DGRAM：数据报套接字  SOCK\_RAW：原始套接字  protocol 设置为0，表示自动选择 |
| 函数返回值 | 若成功，则返回socket描述符；若失败，则返回-1 |
| 备注 | socket描述符是一个指向内部数据结构的指针，它指向描述符表入口。调用socket函数时，socket执行体将建立一个socket，实际上“建立一个socket”意味着为一个socket数据结构分配存储空间。 |

bind函数说明如下

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/socket.h> |
| 函数功能 | 将socket与本机上的一个端口绑定，随后就可以在该端口监听服务请求 |
| 函数原型 | int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen); |
| 函数传入值 | sockfd 调用socket函数返回的socket描述符  my\_addr 指向包含有远端主机IP地址及端口号等信息的sockaddr类型的指针  addrlen 指结构体 |
| 函数返回值 | 若成功，则返回0；若失败，则返回-1 |
| 备注 | 通过socket调用返回一个socket描述符后，在使用socket进行网络传输以前，必须配置该socket。面向连接的socket客户端通过调用connect函数，在socket数据结构中保存本地地址和远端信息。无连接socket的客户端和服务器端以及面向连接socket的服务器端通过调用bind函数来配置本地信息 |

connect函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/socket.h> |
| 函数功能 | 面向连接的客户程序使用connect函数来配置socket并与远程服务器建立一个TCP连接 |
| 函数原型 | int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen); |
| 函数传入值 | sockfd 调用socket函数返回的socket描述符  my\_addr 指向包含有本机IP地址及端口号等信息的sockaddr类型的指针  addrlen 指结构体长度sizeof(struct sockaddr) |
| 函数返回值 | 若成功，则返回0；若失败，则返回-1 |
| 备注 | connect函数启动和远端主机的直接连接。只有面向连接的客户程序使用socket时，才需要将此socket与远端主机相连。无连接协议从不建立直接连接，面向连接的服务器从不启动一个连接，它只是被动地在协议端口监听客户的请求。 |

listen函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/socket.h> |
| 函数功能 | listen函数使socket处于被动的监听模式，并为该socket建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们 |
| 函数原型 | int listen(int sockfd, int backlog); |
| 函数传入值 | sockfd 调用socket函数返回的socket描述符  backlog 最大主机连接数 |
| 函数返回值 | 若成功，则返回0；若失败，则返回-1 |

accept函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | accept函数让服务器接收客户的连接请求 |
| 函数原型 | int accept(int sockfd,struct sockaddr \*addr,socklen\_t \*addrlen); |
| 函数传入值 | sockfd 调用socket函数返回的socket描述符  addr 这是一个结果参数，它用来接受一个返回值，该返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为NULL  addrlen 它也是结果的参数，用来接受上述addr的结构的大小的，它指明addr结构所占有的字节个数。同样的，它也可以被设置为NULL |
| 函数返回值 | 若成功，则返回一个新的套接字描述符；若失败，则返回-1。 |
| 备注 | 如果accept成功返回，则服务器与客户已经正确建立连接了，此时服务器通过accept返回的套接字来完成与客户的通信。 |

close函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/socket.h> |
| 函数功能 | 停止在该socket上的任何数据操作 |
| 函数原型 | int close(int sockfd); |
| 函数传入值 | socket描述符 |
| 函数返回值 | 若成功，则返回0；若失败，则返回-1 |

send函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | 数据发送 |
| 函数原型 | int send(int sockfd, const void \*msg, int len, int flags); |
| 函数传入值 | sockfd 用来传输数据的socket描述符  msg 一个指向要发送数据的指针  len 以字节为单位的数据的长度  flag 一般情况下置为0 |
| 函数返回值 | 若成功，则返回实际上发送出去的字节数；若失败，则返回-1 |

recv函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | 数据接收 |
| 函数原型 | int recv(int sockfd, void \*buf, int len, unsigned int flags); |
| 函数传入值 | sockfd 用来接收数据的socket描述符  buf 是存放接收数据的缓冲区  len 是缓冲区的长度  flags 一般情况下置为0 |
| 函数返回值 | 若成功，则返回实际上接收的字节数；若失败，则返回-1 |

listen和accept的比较：

来自于Stack Overflow上面的一个问答

Ask：I've been reading this tutorial to learn about socket programming. It seems that the listen() and accept() system calls both do the same thing, which is block and wait for a client to connect to the socket that was created with the socket() system call. Why do you need two separate steps for this? Why not just use one system call?

By the way, I have googled this question and found similar questions, but none of the answers were satisfactory. For example, one of them said that accept() creates the socket, which makes no sense, since I know that the socket is created by socket().

Answer1：The listen() function basically sets a flag in the internal socket structure marking the socket as a passive listening socket, one that you can call accept on. It opens the bound port so the socket can then start receiving connections from clients.

The accept() function asks a listening socket to accept the next incoming connection and return a socket descriptor for that connection. So, in a sense, accept() does create a socket, just not the one you use to listen() for incoming connections on.

Answer2：It is all part of the historic setup. listen prepares socket for the next accept call. Listen also allows one to setup the backlog - the number of connections which will be accepted by the system, and than put to wait until your program can really accept them. Everything which comes after the backlog is full well be rejected by the system right away. listen never blocks, while accept will block (unless the socket is in non-blocking mode) until the next connection comes along. Obviously, this does not have to be two separate functions - it is conceivable that accept() function could do everything listen does.

维基百科上的内容：

* listen() is used on the server side, and causes a bound TCP socket to enter listening state.
* accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.

1. UDP编程

UDP 是User Datagram Protocol的简称， 中文名是用户数据报协议。其主要特点是：

* UDP传送数据前并不与对方建立连接，即UDP是无连接的，在传输数据前，发送方和接收方相互交换信息使双方同步。
* UDP不对收到的数据进行排序，在UDP报文的首部中并没有关于数据顺序的信息（如TCP所采用的序号），而且报文不一定按顺序到达的，所以接收端无从排起。
* UDP对接收到的数据报不发送确认信号，发送端不知道数据是否被正确接收，也不会重发数据。
* UDP传送数据较TCP快速，系统开销也少。

基于UDP网络编程常用函数及功能

|  |  |
| --- | --- |
| 函数名 | 功能 |
| bind | 将socket与本机上的一个端口绑定，随后就可以在该端口监听服务请求 |
| close | 停止在该socket上的任何数据操作 |
| sendto | 数据发送函数 |
| recvfrom | 数据接收函数 |

例. 服务器端接收客户端发送的字符串。客户端将打开liu文件，读取文件中的三个字符串，传送给服务器端，当传送给服务器端的字符串为“stop”时，终止数据传送并断开连接。



程序中主要语句说明如下：

1. 服务器端
2. 建立socket。程序调用socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个socket数据结构分配存储空间。用语句实现：

|  |
| --- |
| socket(AF\_INET,SOCK\_DGRAM,0); |

1. 绑定bind。将socket与本机上的一个端口绑定，随后就可以在该端口监听服务请求。用语句实现：

|  |
| --- |
| bind(sockfd,(struct sockaddr \*)&adr\_inet,sizeof(adr\_inet)); |

1. 接收数据recvfrom。该函数用于数据接收。用语句实现：

|  |
| --- |
| recvfrom(sockfd,buf,0,( struct sockaddr \*)&adr\_clnt,&len); |

1. 关闭close。停止在该socket上的任何数据操作。用语句实现：

|  |
| --- |
| close(sockfd); |

1. 客户端
2. 建立socket。程序调用socket()函数，该函数返回一个类似于文件描述符的句柄，同时也意味着为一个socket数据结构分配存储空间。用语句实现：

|  |
| --- |
| socket(AF\_INET,SOCK\_DGRAM,0); |

1. 读取liu文件。读取liu文件里面的内容。用语句实现：

|  |
| --- |
| fopen(“liu”,”r”); |

1. 发送数据sendto。该函数用于在面向连接的socket上进行数据传输。用语句实现：

|  |
| --- |
| sendto(sockfd,buf,sizeof(buf),0,(struct sockaddr \*)&adr\_srvr,sizeof(adr\_srvr)); |

1. 关闭close。停止在该socket上的任何数据操作，用语句实现：

|  |
| --- |
| close(sockfd); |

服务器端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: server.c  \* Author: Mr Mahaitao  \*  \* Created on March 7, 2018, 12:50 AM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <netdb.h>  #include <errno.h>  #include <sys/types.h>  #define PORT 8888  /\*  \*  \*/  int main(int argc, char\*\* argv) {  int sockfd;  int len;  int z;  char buf[256];  struct sockaddr\_in adr\_inet;  struct sockaddr\_in adr\_clnt;  printf("等待客户端....\n");  /\*建立IP地址\*/  adr\_inet.sin\_family = AF\_INET;  adr\_inet.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\*自动获取IP地址\*/  adr\_inet.sin\_port = PORT;  bzero(&(adr\_inet.sin\_zero), 8);  len = sizeof (adr\_clnt);  /\*建立socket\*/  sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);  if (sockfd == -1) {  perror("socket 出错 ");  exit(1);  }  /\*绑定socket\*/  z = bind(sockfd, (struct sockaddr \*) & adr\_inet, sizeof (adr\_inet));  if (z == -1) {  perror("bind 出错 ");  exit(1);  }  while (1) {  /\*接收传来的信息\*/  z = recvfrom(sockfd, buf, sizeof (buf), 0, (struct sockaddr \*) &adr\_clnt, &len);  if (z < 0) {  perror("recvfrom 出错 ");  exit(1);  }  buf[z] = '\0';  printf("接收：%s", buf);  /\*收到stop字符串，终止连接\*/  if (strncmp(buf, "stop", 4) == 0) {  printf("结束....\n");  break;  }  }  close(sockfd);  return (EXIT\_SUCCESS);  } |

客户端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: client.c  \* Author: Mr Mahaitao  \*  \* Created on March 7, 2018, 2:23 AM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <netdb.h>  #include <errno.h>  #include <sys/types.h>  #define PORT 8888  /\*  \*  \*/  int main(int argc, char\*\* argv) {  int sockfd;  int i = 0;  int z;  char buf[80], strl[80];  struct sockaddr\_in adr\_srvr;  FILE \*fp;  printf("打开文件......\n");  /\*以只读的方式打开liu文件\*/  fp = fopen("liu.txt","r");  if (fp == NULL) {  perror("打开文件失败 ");  exit(1);  }  printf("连接服务器...\n");  /\*建立IP地址\*/  adr\_srvr.sin\_family = AF\_INET;  adr\_srvr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  adr\_srvr.sin\_port = PORT;  bzero(&(adr\_srvr.sin\_zero), 8);  sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);  if (sockfd == -1) {  perror("socket 失败");  exit(1);  }  printf("发送文件....\n");  /\*读取三行数据，传给udpserver\*/  for (i = 0; i < 3; i++) {  fgets(strl, 80, fp);  printf("%d:%s", i, strl);  sprintf(buf, "%d:%s", i, strl);  z = sendto(sockfd, buf, sizeof (buf), 0, (struct sockaddr\*) &adr\_srvr, sizeof (adr\_srvr));  if (z < 0) {  perror("sendto 失败");  exit(1);  }  }  printf("发送....\n");  sprintf(buf, "stop\n");  z = sendto(sockfd, buf, sizeof (buf), 0, (struct sockaddr\*) &adr\_srvr, sizeof (adr\_srvr));  if (z < 0) {  perror("sendto \"stop\" 失败");  exit(1);  }    fclose(fp);  close(sockfd);  exit(0);  return (EXIT\_SUCCESS);  } |

相关函数介绍：

sendto函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | 数据发送 |
| 函数原型 | int sendto(int sockfd, void \*buf, int len, unsigned int flags, const struct sockaddr\* to, int tolen); |
| 函数传入值 | sockfd用来传送数据的socket描述符  buf 缓存器指针，用来存放要传送的信息  len sizeof(buf)  flags 一般为0  to 接收端网络地址  tolen sizeof(to) |
| 函数返回值 | 若成功，则返回传送的位数；若失败，则返回-1 |

recvfrom函数说明如下：

|  |  |
| --- | --- |
| 所需的头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 函数功能 | 数据接收 |
| 函数原型 | int recvfrom(int sockfd, void \*buf, size\_t len, unsigned int flags, struct sockaddr \*from, socket\_t \*fromlen); |
| 函数传入值 | sockfd 用来传送数据的socket描述符  buf 缓存器指针，用来存放要传送的信息  len sizeof(buf)  flags 一般为0  from 服务器端的网络地址  tolen sizeof(to) |
| 函数返回值 | 若成功，则返回接收的位数；若失败，则返回-1 |

Linux网络高级编程

1. 高级编程简介

前面介绍了在socket通信中常用的函数，利用这些函数可以满足基本的socket通信需求；同时，也介绍了利用这些函数来编写面向连接的网络通信程序和面向无连接的网络通信程序。但是在socket应用中，还有一个非常重要的特性，就是任何处理阻塞，解决I/O多路复用问题。

在数据通信中，当服务器运行accept()函数时，假设没有客户机连接请求到来，那么服务器就一直会停在accept()语句上，等待客户机连接请求到来，出现这样的情况就称为阻塞。下面将具体介绍出来阻塞的方法。

1. select方法
2. 例1 程序运行当中，在10.5秒判断有没有按下回车键，有则返回“输入了”，否则返回“超时”。

代码如下：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: main.c  \* Author: Mr Henry Ma  \*  \* Created on March 7, 2018, 4:24 AM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <sys/types.h>  #include <unistd.h>  #define STDIN 0 /\*标准输入设备的描述符是0\*/  /\*  \*  \*/  int main(int argc, char\*\* argv) {  struct timeval tv;  fd\_set readfds;  tv.tv\_sec = 10;  tv.tv\_usec = 500000;/\*微秒\*/  FD\_ZERO(&readfds);  FD\_SET(STDIN,&readfds);  /\*don't care about writefds and exceptfds\*/  select(STDIN+1,&readfds,NULL,NULL,&tv);  if(FD\_ISSET(STDIN,&readfds))  printf("输入了！\n");  else  printf("超时！\n");  return (EXIT\_SUCCESS);  } |

select函数说明如下：

|  |  |
| --- | --- |
| 所需头文件 | #include <stdio.h>  #include <sys/time.h>  #include <unistd.h> |
| 函数功能 | select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., read(2)) without blocking. |
| 函数原型 | int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,fd\_set \*exceptfds, struct timeval \*timeout); |
| 函数传入值 | nfds 这个参数的值一般设置为读集合（readfds）、写集合（writefds）以及异常集合（exceptfds）中最大的描述符（fd）+1，当然也可以设置为FD\_SETSIZE。FD\_SETSIZE是操作 定义的一个宏，一般是1024。也就是说读写以及异常集合大小的最大值为1024，所以使用select最多只能管理1024个连接。如果大于1024个连接，select将会产生不确定行为。  readfds 指向可读描述符集的指针，如果我们关心连接的可读事件，需要把连接的描述符设置到读集合中。  writefds 指向可写描述符集的指针，如果我们关心连接的可写事件，需要把连接的描述符设置到读集合中。  exceptfds 指向异常描述符集的指针，如果我们关心连接是否发生异常，需要把连接的描述符设置到异常描述符集合中。  如果对某一个的条件不感兴趣，就可以把它设为空指针。struct fd\_set可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：  void FD\_ZERO(fd\_set \*fdset); //清空集合  void FD\_SET(int fd, fd\_set \*fdset); //将一个给定的文件描述符加入集合之中  void FD\_CLR(int fd, fd\_set \*fdset); //将一个给定的文件描述符从集合中删除  int FD\_ISSET(int fd, fd\_set \*fdset); // 检查集合中指定的文件描述符是否可以读写  timeout 告知内核等待所指定描述字中的任何一个就绪可花多少时间。其timeval结构用于指定这段时间的秒数和微秒数。  struct timeval {  long tv\_sec; //秒数  long tv\_usec; //微妙数  } |
| 函数返回值 | 若成功则返回就绪描述符的数目；超时返回0；出错返回-1 |

使用select函数的过程一般是：

先调用宏FD\_ZERO将指定的fd\_set清零，然后调用宏FD\_SET将需要测试的fd加入fd\_set，接着调用函数select测试fd\_set中的所有fd，最后用宏FD\_ISSET检查某个fd在函数select调用后，相应位是否仍然为1。

宏FD\_ISSET判断描述符fd是否在给定的描述符集fdset中，通常[配合](https://baike.baidu.com/item/%E9%85%8D%E5%90%88)select[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0)使用，由于select函数成功返回时会将未准备好的描述符位清零。通常我们使用FD\_ISSET是为了检查在select函数返回后，某个描述符是否准备好，以便进行接下来的处理操作。

当描述符fd在描述符集fdset中返回非零值，否则，返回零。

FD\_ZERO，FD\_SET，FD\_CLR，FD\_ISSET四个宏的具体实现：

|  |
| --- |
| /\* Access macros for `fd\_set'. \*/  #define FD\_SET(fd, fdsetp) \_\_FD\_SET (fd, fdsetp)  #define FD\_CLR(fd, fdsetp) \_\_FD\_CLR (fd, fdsetp)  #define FD\_ISSET(fd, fdsetp) \_\_FD\_ISSET (fd, fdsetp)  #define FD\_ZERO(fdsetp) \_\_FD\_ZERO (fdsetp)  //每个ulong为32位，可以表示32个bit。  //fd >> 5 即 fd / 32，找到对应的ulong下标i；fd & 31 即fd % 32，找到在ulong[i]内部的位置  #define \_\_FD\_SET(fd, fdsetp) (((fd\_set \*)(fdsetp))->fds\_bits[(fd) >> 5] |= (1<<((fd) & 31))) //设置对应的bit  #define \_\_FD\_CLR(fd, fdsetp) (((fd\_set \*)(fdsetp))->fds\_bits[(fd) >> 5] &= ~(1<<((fd) & 31))) //清除对应的bit  #define \_\_FD\_ISSET(fd, fdsetp) ((((fd\_set \*)(fdsetp))->fds\_bits[(fd) >> 5] & (1<<((fd) & 31))) != 0) //判断对应的bit是否为1  #define \_\_FD\_ZERO(fdsetp) (memset (fdsetp, 0, sizeof (\*(fd\_set \*)(fdsetp)))) //memset bitmap |

fcntl函数说明如下：

|  |  |
| --- | --- |
| 所需头文件 | #include <fcntl.h> |
| 函数功能 | 该函数可以改变已打开的文件的性质 |
| 函数原型 | int fcntl(int fd, int cmd);  int fcntl(int fd, int cmd, long arg);  int fcntl(int fd, int cmd, struct flock \*lock); |
| 函数返回值 | fcntl()的返回值与命令有关。如果出错，所有命令都返回－1，如果成功则返回某个其他值。下列三个命令有特定返回值：F\_DUPFD , F\_GETFD , F\_GETFL以及F\_GETOWN。  F\_DUPFD 返回新的文件描述符  F\_GETFD 返回相应标志  F\_GETFL , F\_GETOWN 返回一个正的进程ID或负的进程组ID |
| 备注 | 获得（设置）文件状态标记cmd=F\_GETFL（cmd=F\_SETFL） |

如果希望套接口不阻塞，就可以使用系统调用函数fcntl()函数：

|  |
| --- |
| #include <unistd.h>  #include <fcntl.h>  … …  … …  sockfd = socket(AF\_INET,SOCK\_STREAM,0);  **fcntl(sockfd,F\_SETFL,O\_NONBLOCK); /\*设置成非阻塞\*/** |

如果用此函数设置为非阻塞模式，缺点就是频繁地询问套接口，以便检查有无信息到来，这样会降低系统效率。

注意：对于fcntl()函数里面的参数，在调用此函数作为非阻塞处理时，都用其固定参数（sockfd,F\_SETFL,O\_NONBLOCK）。

1. 例2 使用select()和fcntl()函数编写一个网络聊天程序。



服务器端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: server.c  \* Author: Henry Ma  \*  \* Created on March 8, 2018, 12:45 AM  \*/  #include <stdlib.h>  #include <stdio.h>  #include <netdb.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <string.h>  #include <netinet/in.h>  #include <unistd.h>  #include <fcntl.h>  #include <arpa/inet.h>  #define MAXDATASIZE 256  #define SERVERPORT 4444 /\*服务器监听端口\*/  #define BACKLOG 1 /\*最大同时连接请求数\*/  #define STDIN 0 /\*标准输入文件描述符\*/  int main(int argc, char \*argv[]) {  FILE \*fp; //定义文件类型指针  int sockfd, client\_fd; //监听sockfd，数据传输sockefd  int sin\_size;  struct sockaddr\_in my\_addr, remote\_addr; // 本机地址信息，客户端地址信息  char message\_buff[256]; /\*用于聊天的缓冲区\*/  char name\_buff[256]; //用于输入用户名的缓冲区  char send\_str[256]; //最多发出去的字符数不能超过256  int recvbytes;  fd\_set rfd\_set, wfd\_set, efd\_set; //被select监听的读，写，异常处理的文件描述符集合  struct timeval timeout; // 本次select的超时结束时间  int ret; // 与client连接的结果  if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) { //错误检测  perror("socket 失败");  exit(1);  }  /\*填充sockaddr结构\*/  bzero(&my\_addr, sizeof (struct sockaddr\_in));  my\_addr.sin\_family = AF\_INET; // 地址族  my\_addr.sin\_port = htons(SERVERPORT); // htons函数将一个32位数从主机字节顺序转换成网络字节顺序  inet\_aton("127.0.0.1", &my\_addr.sin\_addr);  if (bind(sockfd, (struct sockaddr\*) &my\_addr, sizeof (struct sockaddr)) == -1) { // 错误检测  perror("bind 失败");  exit(1);  }    if(listen(sockfd,BACKLOG) == -1){ //错误检测  perror("listen 失败");  exit(1);  }  sin\_size = sizeof (struct sockaddr\_in);  if ((client\_fd = accept(sockfd, (struct sockaddr \*)&remote\_addr, &sin\_size)) == -1) { // 错误检测  perror("accept 失败");  exit(1);  }  fcntl(client\_fd, F\_SETFD, O\_NONBLOCK); // 服务器设置为非阻塞  recvbytes = recv(client\_fd, name\_buff, MAXDATASIZE, 0);  /\*接收从客户端传来的用户名\*/  name\_buff[recvbytes] = '\0';  fflush(stdout); //fflush(stdout)刷新标准输出缓冲区，把输出缓冲区里的东西打印到标准输出设备上  /\*强制立即内容\*/  if ((fp = fopen("name.txt", "a+")) == NULL) {  printf("can not open file,exit...\n");  return -1;  }  fprintf(fp, "%s\n", name\_buff);  /\*将用户名写入name.txt中\*/  while (1) {  FD\_ZERO(&rfd\_set); // 将select()监视的读的文件描述符集合清除  FD\_ZERO(&wfd\_set); // 将select()监视的写的文件描述符集合清除  FD\_ZERO(&efd\_set); // 将select()监视的异常的文件描述符集合清除  /\*将标准输入文件描述符加到select()监视的读的文件描述符集合中\*/  FD\_SET(STDIN, &rfd\_set);  /\*将新建的描述符加到select()监视的读的文件描述符集合中\*/  FD\_SET(client\_fd, &rfd\_set);  /\*将新建的描述符加到select()监视的写的文件描述符集合中\*/  FD\_SET(client\_fd, &wfd\_set);  /\*将新建的描述符加到select()监视的异常的文件描述符集合中\*/  FD\_SET(client\_fd, &efd\_set);  timeout.tv\_sec = 10; //select 在被监视的窗口等待的秒数  timeout.tv\_usec = 0; //select 在被监视的窗口等待的微秒数  ret = select(client\_fd + 1, &rfd\_set, &wfd\_set, &efd\_set, &timeout);  if (ret == 0) {  //printf("select 超时\n");  continue;  }  if (ret < 0) {  perror("select 失败");  exit(1);  }  /\*判断标准输入文件描述符是否还在给定的描述符集rfd\_set中,也就说判断是否有数据从标准输入进来\*/  if (FD\_ISSET(STDIN, &rfd\_set)) {  fgets(send\_str, 256, stdin); //从标准输入中取出要发送的内容  send\_str[strlen(send\_str) - 1] = '\0';  if (strncmp("quit", send\_str, 4) == 0) {// 退出程序  //printf("deubg1\n");  close(client\_fd);  close(sockfd); //关闭套接字  exit(0);  }  send(client\_fd, send\_str, strlen(send\_str), 0);  }  /\*判断新建的描述符是否还在给定的描述符集wfd\_set中,也就说判断是否有数据从客户端发来\*/  if (FD\_ISSET(client\_fd, &rfd\_set)) {  recvbytes = recv(client\_fd, message\_buff, MAXDATASIZE, 0); //接收从客户端发来的聊天消息  if (recvbytes == 0) {  //printf("debug2\n");  close(client\_fd);  close(sockfd); //关闭套接字  exit(0);  }  message\_buff[recvbytes] = '\0';  printf("%s:%s\n", name\_buff, message\_buff);  printf("Server: ");  fflush(stdout);  }  /\*判断新建的描述符是否还在给定的描述符集efd\_set中,也就说判断是否有异常产生\*/  if (FD\_ISSET(client\_fd, &efd\_set)) {  //printf("debug3\n");  close(client\_fd);  exit(1);  }  }  return 1;  } |

客户端代码：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: client.c  \* Author: Henry Ma  \*  \* Created on March 8, 2018, 9:16 PM  \*/  #include <stdio.h>  #include <stdlib.h>  #include <netdb.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <string.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #include <fcntl.h>  #define SERVERPORT 4444 //服务器监听端口号  #define MAXDATASIZE 256 //最大同时连接请求  #define STDIN 0 //标准输入文件描述符  /\*  \*  \*/  int main(int argc, char\*\* argv) {  int sockfd; //套接字描述符  int recvbytes;  char buf[MAXDATASIZE]; //用于处理输入的缓冲区  char \*str;  char name[MAXDATASIZE]; //定义用户名  char send\_str[MAXDATASIZE]; //最多发出的字符不能超过MAXDATASIZE  struct sockaddr\_in serv\_addr; //Internet 套接字地址结构  fd\_set rfd\_set, wfd\_set, efd\_set; //被select()监视的读，写，异常处理的文件描述符集合  struct timeval timeout; //本次select的超时结束时间  int ret; //与server连接的结果  /\*创建一个socket\*/  if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) { //错误检测  perror("socket 失败");  exit(1);  }  /\*填充sockaddr结构\*/  bzero(&serv\_addr, sizeof (struct sockaddr\_in));  serv\_addr.sin\_family = AF\_INET;  serv\_addr.sin\_port = htons(SERVERPORT);  inet\_aton("127.0.0.1", &serv\_addr.sin\_addr);  /\*请求连接\*/  if (connect(sockfd, (struct sockaddr\*) &serv\_addr, sizeof (struct sockaddr)) == -1) { //错误检测  perror("connect 失败");  exit(1);  }  fcntl(sockfd, F\_SETFD, O\_NONBLOCK);  printf("要聊天请首先输入你的名字：");  scanf("%s", name);  name[strlen(name)] = '\0';  printf("%s: ", name);  fflush(stdout);  send(sockfd, name, strlen(name), 0); // 发送用户名到sockfd  while (1) {  FD\_ZERO(&rfd\_set); // 将select()监视的读的文件描述符集合清除  FD\_ZERO(&wfd\_set); // 将select()监视的写的文件描述符集合清除  FD\_ZERO(&efd\_set); // 将select()监视的异常的文件描述符集合清除  /\*将标准输入文件描述符加到select()监视的读的文件描述符集合中\*/  FD\_SET(STDIN, &rfd\_set);  /\*将新建的描述符加到select()监视的读的文件描述符集合中\*/  FD\_SET(sockfd, &rfd\_set);  /\*将新建的描述符加到select()监视的异常的文件描述符集合中\*/  FD\_SET(sockfd, &efd\_set);  timeout.tv\_sec = 10; //select 在被监视的窗口等待的秒数  timeout.tv\_usec = 0; //select 在被监视的窗口等待的微秒数  ret = select(sockfd + 1, &rfd\_set, &wfd\_set, &efd\_set, &timeout);  if (ret == 0) {  //printf("select 超时\n");  continue;  }  if (ret < 0) {  perror("select error: ");  exit(1);  }  /\*判断标准输入文件描述符是否还在给定的描述符集rfd\_set中,也就说判断是否有数据从标准输入进来\*/  if (FD\_ISSET(STDIN, &rfd\_set)) {  fgets(send\_str, 256, stdin);  send\_str[strlen(send\_str) - 1] = '\0';  if (strncmp("quit", send\_str, 4) == 0) { //退出程序  close(sockfd);  exit(0);  }  send(sockfd, send\_str, strlen(send\_str), 0);  }  /\*判断新建的描述符是否还在给定的描述符集rfd\_set中,也就说判断是否有数据从服务器端发来\*/  if (FD\_ISSET(sockfd, &rfd\_set)) {  recvbytes = recv(sockfd, buf, MAXDATASIZE, 0);  if (recvbytes == 0) {  close(sockfd);  exit(1);  }  buf[recvbytes] = '\0';  printf("Server: %s\n", buf);  printf("%s: ", name);  fflush(stdout);  }  /\*判断新建的描述符是否还在给定的描述符集efd\_set中,也就说判断是否有异常产生\*/  if (FD\_ISSET(sockfd, &efd\_set)) {  close(sockfd);  exit(1);  }  }  return (EXIT\_SUCCESS);  } |

1. poll方法

轮询函数poll提供了与函数select相似的功能，它可以将套接口设置为非阻塞模式。但是和select()不一样，poll()没有使用低效的三个基于位的文件描述符集合，而是采用了一个单独的结构体pollfd数组。

技巧：非阻塞的I/O模式一般采用的是轮询的方法，可以使用一个循环测试语句来实现。

poll函数说明如下：

|  |  |
| --- | --- |
| 所需头文件 | #include <poll.h> |
| 函数功能 | poll() performs a similar task to select(): it waits for one of a set of file descriptors to become ready to perform I/O. |
| 函数原型 | int poll(struct pollfd \*fds, nfds\_t nfds, int timeout); |
| 函数传入值 | fds：可以传递多个结构体，也就是说可以监测多个驱动设备所产生的事件，只要有一个产生了请求事件，就能立即返回。  结构pollfd的定义如下：  struct pollfd {  int fd; /\* file descriptor \*/  short events; /\* requested events \*/  short revents; /\* returned events \*/  };  结构成员fd表示要测试的描述符。如果不关心某个描述符，可以将pollfd结构中的fd成员设为负值。  每一个pollfd结构体指定一个被监视的文件描述符，可以传递多个结构体，指示poll()监视多个文件描述符。每个结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域。revents域是文件描述符的操作结果事件掩码。内核在调用返回时设置这个域。events域中请求的任何事件都可以在revents域中返回。合法的事件如下：  POLLIN：有数据可读  POLLRDNORM：有普通数据可读（建议用POLLIN代替）  POLLRDBAND：有优先数据可读  POLLPRI：有紧迫数据可读  POLLOUT：写数据不会导致阻塞  POLLWRNORM：写普通数据不会导致阻塞  POLLMSG：SIGPOLL消息可用  此外，revents域中还可以返回下列事件：  POLLER：指定的文件描述符发生错误。  POLLHUP：指定的文件描述符挂起事件。  POLLNVAL：指定的文件描述符非法。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  POLLIN|POLLPRI 等价于 select()的读事件  POLLOUT|POLLWRBAND 等价于select()的写事件  POLLIN 等价于 POLLRDNORM|POLLRDBAND  POLLOUT 等价于 POLLWRNORM  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  例如：要同时监视一个文件描述符是否可读可写，我们可以设置events为POLLIN|POLLOUT。在poll返回时，我们可以检查revents中的标志，对应于文件描述符请求的events值。如果POLLIN事件被设置，则文件描述符可以被读取而不阻塞。如果POLLOUT被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可以被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。  nfds：要监视的描述符的数目。  timeout：超时时间，单位为ms，它有三种情况：1.当大于0时，则等待指定长度的时间；2.当等于0时，表示立即返回；3.当为INFTIM时，则表示永远等待，包含在<poll.h>，也有些系统包含在<sys/stropts.h>，也有些系统没有定义，需要自己定义#define INFTIM -1。 |
| 函数返回值 | 如果成功，poll()将返回结构体中revents域不为0的文件描述符个数；如果超时，则返回0；如果出错，则返回-1，并设置errno为下列值之一：  EBADF：一个或者多个结构体中指定的文件描述符无效。  EFAULT：fds指针指向的地址超出进程的地址空间。  EINTR：请求的事件之间产生一个信号，调用可以重新发起。  EINVAL：nfds参数超过PLIMIT\_NOFILE值。  ENOMEM：可用内存不足，无法完成请求。 |
| 备注 | 一般来说，轮询不是一个好方法，它将使程序一直处于等待状态查询套接口的数据，从而浪费大量的CPU的时间，并且它的实现比较麻烦。 |

例1代码：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/ioctl.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h> /\*文件控制\*/  #include <sys/select.h>  #include <sys/time.h> /\*时间方面的函数\*/  #include <errno.h> /\*有关错误方面的宏\*/  #include<sys/poll.h> //poll()  #include<fcntl.h>  #include<string.h> //memset()  int main(int argc, char \*argv[]) {  int fd;  int key\_value;  int ret;  struct pollfd event; //创建一个struct pollfd结构体变量，存放文件描述符、要等待发生的事件  fd = open("/dev/key", O\_RDWR);  if (fd < 0) {  perror("open /dev/key error!\n");  exit(1);  }  printf("open /dev/key sucessfully!\n");  while (1) { //poll结束后struct pollfd结构体变量的内容被全部清零，需要再次设置  memset(&event, 0, sizeof (event)); //memst函数对对象的内容设置为同一值  event.fd = fd; //存放打开的文件描述符  event.events = POLLIN; //存放要等待发生的事件  ret = poll((struct pollfd \*) &event, 1, 5000); //监测event，一个对象，等待5000毫秒后超时,-1为无限等待  //判断poll的返回值，负数是出错，0是设定的时间超时，整数表示等待的时间发生  if (ret < 0) {  printf("poll error!\n");  exit(1);  }  if (ret == 0) {  printf("Time out!\n");  continue;  }  if (event.revents & POLLERR) { //revents是由内核记录的实际发生的事件，events是进程等待的事件  printf("Device error!\n");  exit(1);  }  if (event.revents & POLLIN) {  read(fd, &key\_value, sizeof (key\_value));  printf("Key value is '%d'\n", key\_value);  }  }  close(fd);  return 0;  } |

例2代码：

|  |
| --- |
| #include<stdio.h>  #include<errno.h>  #include<string.h>  #include<poll.h>  int main(int argc,char \*argv[]) {  int read\_fd = 0; // 标准输入描述符  int write\_fd = 1; // 标准输出描述符  struct pollfd \_poll\_fd[2]; // 指定被监视的描述符，可以传递多个  \_poll\_fd[0].fd = read\_fd;  \_poll\_fd[0].events = POLLIN; // 有数据可读  \_poll\_fd[0].revents = 0; // 返回值  \_poll\_fd[1].events = POLLOUT; // 写数据不会导致阻塞  \_poll\_fd[1].fd = write\_fd;  \_poll\_fd[1].revents = 0;  nfds\_t fds = 2; //要监视的描述符的数目  int timeout = 1000; // 定时器  char buf[1024]; // 缓冲区  memset(buf, '\0', sizeof (buf)); // 初始化缓冲区  while (1) {  switch (poll(\_poll\_fd, 2, timeout)) {  case 0: // poll返回值为0时表示超时  printf("timeout\n");  break;  case -1: // poll返回值为-时表示出错  perror("poll error\n");  break;  default: // 否则返回准备好的描述符的个数  {  //int i=0;  if ((\_poll\_fd[0].revents) & (POLLIN)) { // 有输入  ssize\_t \_size = read(0, buf, sizeof (buf) - 1); // 读进缓冲区  if (\_size > 0) { // 读成功  buf[\_size] = '\0';  if ((\_poll\_fd[1].revents) & (POLLOUT)) { // 可以写  printf("%s\n", buf); // 打印出来  }  }  }  }  break;  }  }  return 0;  } |

例3 服务器端代码：

|  |
| --- |
| /\* server.c ---  \* Filename: server.c  \* Created: Mon Jan 8 16:45:14 2018 (+0800)  \*/  /\* Copyright Hamlet\_Jiaxiaoyu.  \*  \* 允许免费使用，拷贝，修改，发布，但在所有的拷贝上必须保留上述  \* copyright部分和本使用声明部分，除非显示声明，copyright的持有者  \* 不得作为再发布软件的广告。copyright的持有者对使用本软件的适用范围不做任何声明，  \*  \* THE COPYRIGHT HOLDERS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,  \* INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO  \* EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR  \* CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,  \* DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER  \* TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  \* OF THIS SOFTWARE.  \*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <errno.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <poll.h>  #include <unistd.h>  #include <sys/types.h>  #include <arpa/inet.h>  #define IPADDRESS "127.0.0.1"  #define PORT 8787  #define MAXLINE 1024  #define LISTENQ 5  #define OPEN\_MAX 1000  #define INFTIM -1  //函数声明  //创建套接字并进行绑定  static int socket\_bind(const char\* ip, int port);  //IO多路复用poll  static void do\_poll(int listenfd);  //处理多个连接  static void handle\_connection(struct pollfd \*connfds, int num);  int main(int argc, char \*argv[]) {  int listenfd, connfd, sockfd;  struct sockaddr\_in cliaddr;  socklen\_t cliaddrlen;  listenfd = socket\_bind(IPADDRESS, PORT);  listen(listenfd, LISTENQ);  do\_poll(listenfd);  return 0;  }  static int socket\_bind(const char\* ip, int port) {  int listenfd;  struct sockaddr\_in servaddr;  listenfd = socket(AF\_INET, SOCK\_STREAM, 0);  if (listenfd == -1) {  perror("socket error:");  exit(1);  }  bzero(&servaddr, sizeof (servaddr));  servaddr.sin\_family = AF\_INET;  inet\_pton(AF\_INET, ip, &servaddr.sin\_addr);  servaddr.sin\_port = htons(port);  if (bind(listenfd, (struct sockaddr\*) &servaddr, sizeof (servaddr)) == -1) {  perror("bind error: ");  exit(1);  }  return listenfd;  }  static void do\_poll(int listenfd) {  int connfd, sockfd;  struct sockaddr\_in cliaddr;  socklen\_t cliaddrlen;  struct pollfd clientfds[OPEN\_MAX];  int maxi;  int i;  int nready;  //添加监听描述符  clientfds[0].fd = listenfd;  clientfds[0].events = POLLIN;  //初始化客户连接描述符  for (i = 1; i < OPEN\_MAX; i++)  clientfds[i].fd = -1;  maxi = 0;  //循环处理  for (;;) {  //获取可用描述符的个数  nready = poll(clientfds, maxi + 1, INFTIM);  if (nready == -1) {  perror("poll error:");  exit(1);  }  //测试监听描述符是否准备好  if (clientfds[0].revents & POLLIN) {  cliaddrlen = sizeof (cliaddr);  //接受新的连接  if ((connfd = accept(listenfd, (struct sockaddr\*) &cliaddr, &cliaddrlen)) == -1) {  if (errno == EINTR)  continue;  else {  perror("accept error:");  exit(1);  }  }  fprintf(stdout, "accept a new client: %s:%d\n", inet\_ntoa(cliaddr.sin\_addr), cliaddr.sin\_port);  //将新的连接描述符添加到数组中  for (i = 1; i < OPEN\_MAX; i++) {  if (clientfds[i].fd < 0) {  clientfds[i].fd = connfd;  break;  }  }  if (i == OPEN\_MAX) {  fprintf(stderr, "too many clients.\n");  exit(1);  }  //将新的描述符添加到读描述符集合中  clientfds[i].events = POLLIN;  //记录客户连接套接字的个数  maxi = (i > maxi ? i : maxi);  if (--nready <= 0)  continue;  }  //处理客户连接  handle\_connection(clientfds, maxi);  }  }  static void handle\_connection(struct pollfd \*connfds, int num) {  int i, n;  char buf[MAXLINE];  memset(buf, 0, MAXLINE);  for (i = 1; i <= num; i++) {  if (connfds[i].fd < 0)  continue;  //测试客户描述符是否准备好  if (connfds[i].revents & POLLIN) {  //接收客户端发送的信息  n = read(connfds[i].fd, buf, MAXLINE);  if (n == 0) {  close(connfds[i].fd);  connfds[i].fd = -1;  continue;  }  // printf("read msg is: ");  write(STDOUT\_FILENO, buf, n);  //向客户端发送buf  write(connfds[i].fd, buf, n);  }  }  } |

例3 客户端代码

|  |
| --- |
| /\* client.c ---  \* Filename: client.c  \* Created: Mon Jan 8 16:45:33 2018 (+0800)  \*/  /\* Copyright Hamlet\_Jiaxiaoyu.  \*  \* 允许免费使用，拷贝，修改，发布，但在所有的拷贝上必须保留上述  \* copyright部分和本使用声明部分，除非显示声明，copyright的持有者  \* 不得作为再发布软件的广告。copyright的持有者对使用本软件的适用范围不做任何声明，  \*  \* THE COPYRIGHT HOLDERS DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,  \* INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO  \* EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR  \* CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,  \* DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER  \* TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  \* OF THIS SOFTWARE.  \*/  #include <netinet/in.h>  #include <sys/socket.h>  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <poll.h>  #include <time.h>  #include <unistd.h>  #include <sys/types.h>  #define MAXLINE 1024  #define IPADDRESS "127.0.0.1"  #define SERV\_PORT 8787  #define max(a,b) (a > b) ? a : b  static void handle\_connection(int sockfd);  int main(int argc, char \*argv[]) {  int sockfd;  struct sockaddr\_in servaddr;  sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  bzero(&servaddr, sizeof (servaddr));  servaddr.sin\_family = AF\_INET;  servaddr.sin\_port = htons(SERV\_PORT);  inet\_pton(AF\_INET, IPADDRESS, &servaddr.sin\_addr);  connect(sockfd, (struct sockaddr\*) &servaddr, sizeof (servaddr));  //处理连接描述符  handle\_connection(sockfd);  return 0;  }  static void handle\_connection(int sockfd) {  char sendline[MAXLINE], recvline[MAXLINE];  int maxfdp, stdineof;  struct pollfd pfds[2];  int n;  //添加连接描述符  pfds[0].fd = sockfd;  pfds[0].events = POLLIN;  //添加标准输入描述符  pfds[1].fd = STDIN\_FILENO;  pfds[1].events = POLLIN;  for (;;) {  poll(pfds, 2, -1);  if (pfds[0].revents & POLLIN) {  n = read(sockfd, recvline, MAXLINE);  if (n == 0) {  fprintf(stderr, "client: server is closed.\n");  close(sockfd);  }  write(STDOUT\_FILENO, recvline, n);  }  //测试标准输入是否准备好  if (pfds[1].revents & POLLIN) {  n = read(STDIN\_FILENO, sendline, MAXLINE);  if (n == 0) {  shutdown(sockfd, SHUT\_WR);  continue;  }  write(sockfd, sendline, n);  }  }  } |

1. epoll方法

为了克服以上缺点，linux2.6内核加入了epoll机制。epoll是linux2.6加入的用于I/O事件多路分离的一组函数，这组函数简化了反应式socket服务器的编程，并且很大程度上提高了性能。从一定程度上弥补了Linux内核对异步I/O支持的不足，epoll机制加上非阻塞I/O可以模拟实现异步I/O。

设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。（从用户态复制到内核态，再从内核态复制到用户态，这一过程资源消耗很大）

epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：

1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)

2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

3）调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。下面来看看Linux内核具体的epoll机制实现思路。

当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：

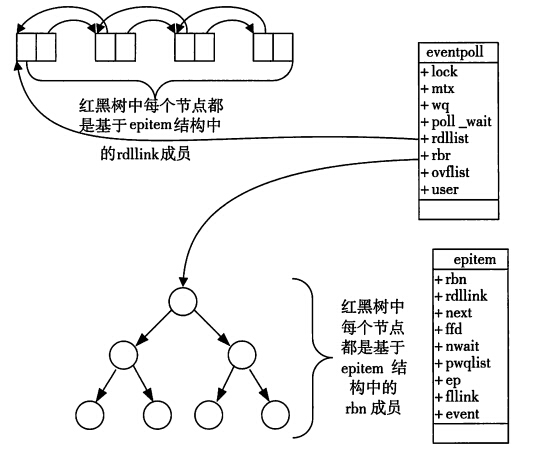
|  |
| --- |
| struct eventpoll{  ....  /\*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件\*/  struct rb\_root rbr;  /\*双链表中则存放着将要通过epoll\_wait返回给用户的满足条件的事件\*/  struct list\_head rdlist;  ....  }; |

每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：

|  |
| --- |
| struct epitem {  struct rb\_node rbn;//红黑树节点  struct list\_head rdllink;//双向链表节点  struct epoll\_filefd ffd; //事件句柄信息  struct eventpoll \*ep; //指向其所属的eventpoll对象  struct epoll\_event event; //期待发生的事件类型  } |

当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。



epoll数据结构示意图

从上面的讲解可知：通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。

OK，讲解完了Epoll的机理，我们便能很容易掌握epoll的用法了。一句话描述就是：三步曲。

第一步：epoll\_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll\_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。

第三步：epoll\_wait()系统调用。通过此调用收集在epoll监控中已经发生的事件。

epoll用到的数据结构和函数：

1. 数据结构

|  |
| --- |
| typedef union epoll\_data {  void \*ptr;  int fd;  uint32\_t u32;  uint64\_t u64;  } epoll\_data\_t; |
| struct epoll\_event {  uint32\_t events; /\* Epoll events \*/  epoll\_data\_t data; /\* User data variable \*/  }; |

结构体epoll\_event 被用于注册所感兴趣的事件和回传所发生待处理的事件，其中epoll\_data联合体用来保存触发事件的某个文件描述符相关的数据，例如一个client连接到服务器，服务器通过调用accept函数可以得到关于这个client对应的socket文件描述符，可以把这个文件描述符赋值给epoll\_data的fd字段以便后面的读写在这个文件描述符上进行。

epoll\_event 结构体的events字段表示感兴趣的事件和被触发的事件，可能的取值为：

events可以是以下几个宏的集合：

|  |
| --- |
| 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)来说的。  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. 函数
2. epoll\_create函数

|  |  |
| --- | --- |
| 函数申明 | int epoll\_create(int size); |
| 函数作用 | 该函数生成一个epoll专用的文件描述符，其中的参数是指定内核分配size个相关内核对象。 |

1. epoll\_ctl函数

|  |  |
| --- | --- |
| 函数申明 | int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event \*event); |
| 函数作用 | 该函数用于控制某个文件描述符上的事件，可以注册事件，修改事件，删除事件。 |
| 参数 | epfd：由epoll\_create生成的专用的文件描述符  op：要进行的操作，例如注册事件，可能的取值EPOLL\_CTL\_ADD注册、EPOLL\_CTL\_MOD修改、EPOLL\_CTL\_DEL删除。  fd：关联的文件描述符  event：指向epoll\_event的指针 |
| 返回值 | 如果调用成功返回0，不成功返回-1 |

1. epoll\_wait函数

|  |  |
| --- | --- |
| 函数申明 | int epoll\_wait(int epfd,struct epoll\_event\* events,int maxevents,int timeout); |
| 函数作用 | 该函数用于轮询I/O事件的发生。 |
| 参数 | epfd：由epoll\_create生成的专用的文件描述符  events：用于回传待处理事件的数组  maxevents：每次能处理的事件数  timeout：等待I/O事件发生的超时值 |
| 返回值 | When successful, epoll\_wait() returns the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready during the requested timeout milliseconds. When an error occurs,epoll\_wait() returns -1 and errno is set appropriately. |

1. 实际例子

epoll服务器端：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <errno.h>  #include <string.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #include <sys/socket.h>  #include <fcntl.h>  #include <sys/epoll.h>  #define MAXBUF 1024  #define MAXEPOLLSIZE 10000  /\*  \* 设置句柄为非阻塞方式  \*/  int setnonblocking(int sockfd) {  if (fcntl(sockfd, F\_SETFL, fcntl(sockfd, F\_GETFD, 0) | O\_NONBLOCK) == -1) {  return -1;  }  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* filename: epoll-server.c  \* 演示epoll接受海量socket并进行处理响应的方法  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int main(int argc, char \*\*argv) {  int listenfd, connfd, epfd, sockfd, nfds, n, curfds;  socklen\_t len;  struct sockaddr\_in my\_addr, their\_addr;  unsigned int myport, lisnum;  char buf[MAXBUF + 1];  // 声明epoll\_event结构体的变量，ev用于注册事件，events数组用于回传要处理的事件  struct epoll\_event ev;  struct epoll\_event events[MAXEPOLLSIZE];  if (argv[1])  myport = atoi(argv[1]);  else  myport = 7838;  if (argv[2])  lisnum = atoi(argv[2]);  else  lisnum = 2;  // 开启 socket 监听  if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) == -1) {  perror("socket");  exit(1);  } else  printf("socket 创建成功！\n");  // 把socket设置为非阻塞方式  setnonblocking(listenfd);  bzero(&my\_addr, sizeof (my\_addr));  my\_addr.sin\_family = PF\_INET;  my\_addr.sin\_port = htons(myport);  if (argv[3])  my\_addr.sin\_addr.s\_addr = inet\_addr(argv[3]);  else  my\_addr.sin\_addr.s\_addr = INADDR\_ANY;  if (bind(listenfd, (struct sockaddr \*) &my\_addr, sizeof (struct sockaddr)) == -1) {  perror("bind");  exit(1);  } else  printf("IP 地址和端口绑定成功\n");  if (listen(listenfd, lisnum) == -1) {  perror("listen");  exit(1);  } else  printf("开启服务成功！\n");  // 创建 epoll句柄，把监听 socket 加入到 epoll 集合里 \*/  epfd = epoll\_create(MAXEPOLLSIZE); /\*epoll专用的文件描述符\*/  len = sizeof (struct sockaddr\_in);  ev.events = EPOLLIN | EPOLLET;  ev.data.fd = listenfd;  // 将listenfd注册到epoll事件  if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev) < 0) {  fprintf(stderr, "epoll set insertion error: fd=%d\n", listenfd);  return -1;  } else  printf("监听 socket 加入 epoll 成功！\n");  curfds = 1;  while (1) {  // 等待有事件发生  nfds = epoll\_wait(epfd, events, curfds, -1);  if (nfds == -1) {  perror("epoll\_wait");  break;  }  // 处理所有事件  for (n = 0; n < nfds; ++n) {  // 如果新监测到一个SOCKET用户连接到了绑定的SOCKET端口，建立新的连接  if (events[n].data.fd == listenfd) { // 有新的连接过来  len = sizeof (struct sockaddr);  connfd = accept(listenfd, (struct sockaddr \*) &their\_addr, &len); // 为新的连接创建socket  if (connfd < 0) {  perror("accept");  continue;  }  else {  printf("有连接来自于： %s:%hu，分配的 socket 为:%d\n", inet\_ntoa(their\_addr.sin\_addr), ntohs(their\_addr.sin\_port), connfd);  }  setnonblocking(connfd);  // 设置用于注册的 读操作 事件  ev.events = EPOLLIN | EPOLLET;  // 设置用于读操作的文件描述符  ev.data.fd = connfd;  //注册ev  epoll\_ctl(epfd, EPOLL\_CTL\_ADD, connfd, &ev);  curfds++;  }  else if (events[n].events & EPOLLIN) // 如果是已经连接的用户，并且收到数据，那么进行读入  {  printf("EPOLLIN\n");  if ((sockfd = events[n].data.fd) < 0)  continue;  int len;  bzero(buf, MAXBUF + 1);  /\* 接收客户端的消息 \*/  /\*len = read(sockfd, buf, MAXBUF);\*/  len = recv(sockfd, buf, MAXBUF, 0);  if (len > 0)  printf("%d接收消息成功:'%s'，共%d个字节的数据\n", sockfd, buf, len);  else {  if (len < 0) {  printf("消息接收失败！错误代码是%d，错误信息是'%s'\n", errno, strerror(errno));  epoll\_ctl(epfd, EPOLL\_CTL\_DEL, sockfd, &ev);  curfds--;  continue;  }  }  // 设置用于写操作的文件描述符  ev.data.fd = sockfd;  // 设置用于注册的写操作事件  ev.events = EPOLLOUT | EPOLLET;  /\*修改sockfd上要处理的事件为EPOLLOUT\*/  epoll\_ctl(epfd, EPOLL\_CTL\_MOD, sockfd, &ev); //修改标识符，等待下一个循环时发送数据，异步处理的精髓!!!!! ?????  }  else if (events[n].events & EPOLLOUT) // 如果有数据发送  {  printf("EPOLLOUT\n");  sockfd = events[n].data.fd;  bzero(buf, MAXBUF + 1);  strcpy(buf, "Server already processes!");  send(sockfd, buf, strlen(buf), 0);  // 设置用于读操作的文件描述符  ev.data.fd = sockfd;  // 设置用于注册的读操作事件  ev.events = EPOLLIN | EPOLLET;  // 修改sockfd上要处理的事件为EPOLIN  epoll\_ctl(epfd, EPOLL\_CTL\_MOD, sockfd, &ev);  }  }  }  close(listenfd);  return 0;  } |

epoll客户端：

|  |
| --- |
| /\*  \* To change this license header, choose License Headers in Project Properties.  \* To change this template file, choose Tools | Templates  \* and open the template in the editor.  \*/  /\*  \* File: main.c  \* Author: root  \*  \* Created on March 10, 2018, 6:33 AM  \*/  #include <stdio.h>  #include <string.h>  #include <errno.h>  #include <sys/socket.h>  #include <resolv.h>  #include <stdlib.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #include <sys/time.h>  #include <sys/types.h>  #define MAXBUF 1024  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* filename: select-client.c  \* 演示网络异步通讯，这是客户端程序  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int main(int argc, char \*\*argv) {  int sockfd, len;  struct sockaddr\_in dest;  char buffer[MAXBUF + 1];  fd\_set rfds;  struct timeval tv;  int retval, maxfd = -1;  if (argc != 3) {  printf("参数格式错误！正确用法如下：\n\t\t%s IP地址 端口\n\t比如:\t%s 127.0.0.1 80\n此程序用来从某个 IP 地址的服务器某个端口接收最多 MAXBUF 个字节的消息", argv[0], argv[0]);  exit(0);  }  // 创建一个 socket 用于 tcp 通信  if ((sockfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {  perror("Socket");  exit(errno);  }  // 初始化服务器端（对方）的地址和端口信息  bzero(&dest, sizeof (dest));  dest.sin\_family = AF\_INET;  dest.sin\_port = htons(atoi(argv[2]));  if (inet\_aton(argv[1], (struct in\_addr \*) &dest.sin\_addr.s\_addr) == 0) {  perror(argv[1]);  exit(errno);  }  // 连接服务器  if (connect(sockfd, (struct sockaddr \*) &dest, sizeof (dest)) != 0) {  perror("Connect ");  exit(errno);  }  printf("\n准备就绪，可以开始聊天了……直接输入消息回车即可发信息给对方\n");  while (1) {  // 把集合清空  FD\_ZERO(&rfds);  // 把标准输入句柄0加入到集合中  FD\_SET(0, &rfds);  maxfd = 0;  // 把当前连接句柄sockfd加入到集合中  FD\_SET(sockfd, &rfds);  if (sockfd > maxfd)  maxfd = sockfd;  // 设置最大等待时间  tv.tv\_sec = 3;  tv.tv\_usec = 0;  // 开始等待  retval = select(maxfd + 1, &rfds, NULL, NULL, &tv);  if (retval == -1) {  printf("将退出，select出错！ %s", strerror(errno));  break;  }  else if (retval == 0) {  /\* printf("没有任何消息到来，用户也没有按键，继续等待……\n"); \*/  continue;  }  else {  if (FD\_ISSET(sockfd, &rfds)) // 连接的socket上有消息到来则接收对方发过来的消息并显示  {  bzero(buffer, MAXBUF + 1);  // 接收对方发过来的消息，最多接收 MAXBUF 个字节  len = recv(sockfd, buffer, MAXBUF, 0);  if (len > 0) {  printf("接收消息成功:'%s'，共%d个字节的数据\n", buffer, len);  } else {  if (len < 0) {  printf("消息接收失败！错误代码是%d，错误信息是'%s'\n", errno, strerror(errno));  } else {  printf("对方退出了，聊天终止！\n");  }  break;  }  }  if (FD\_ISSET(0, &rfds)) // 用户按键了，则读取用户输入的内容发送出去  {  bzero(buffer, MAXBUF + 1);  fgets(buffer, MAXBUF, stdin);  if (!strncasecmp(buffer, "quit", 4)) {  printf("自己请求终止聊天！\n");  break;  }  // 发消息给服务器  len = send(sockfd, buffer, strlen(buffer) - 1, 0);  if (len < 0) {  printf("消息'%s'发送失败！错误代码是%d，错误信息是'%s'\n", buffer, errno, strerror(errno));  break;  }  else {  printf("消息：%s\t发送成功，共发送了%d个字节！\n", buffer, len);  }  }  }  }  // 关闭连接  close(sockfd);  return 0;  } |

1. 总结

Nginx使用最新的epoll（Linux2.6 内核）和kqueue（FreeBSD）网络I/O模型，而Apache则使用的是传统的select模型。目前Linux下能够承受高并发访问的Squid、Memcached都采用的是epoll网络I/O模型。

处理大量连续的读写，Apache所采用的的select网络I/O模型比较低效。下面用一个比喻来解析Apache采用的select模型和Nginx采用的epoll模型之间的区别：

假设你在大学读书，住的宿舍楼有很多间房间，你的朋友要来找你。select版宿舍大妈就会带着你的朋友挨个房间去找，直到找到你为止。而epoll版宿舍大妈会先记下每位进入同学的房间号，你的朋友来时，只需要告诉你的朋友你住在那个房间即可，不必亲自带着你的朋友满宿舍楼找人。如果来了10000个人，都要找自己住在这栋楼的同学时，select版和epoll版宿管大妈，谁的效率更高，大家应该清楚了。同理，在高并发服务器中，轮询I/O是最耗时间的操作之一，select和epoll的性能谁更高，同样十分明了。

|  |  |  |
| --- | --- | --- |
|  | select | epoll |

|  |  |  |
| --- | --- | --- |
| 性能 | 随着连接数的增加，急剧下降。处理成千上万并发连接数时，性能很差。 | 随着连接数增加，性能基本上没有什么下降。处理成千上万个并发连接时，性能很好。 |

|  |  |  |
| --- | --- | --- |
| 连接数 | 连接数有限制，处理的最大连接数不超过1024.如果要处理超过1024个连接数，则需要修改FD\_SETSIZE宏，并重新编译。 | 连接数无限制。 |

|  |  |  |
| --- | --- | --- |
| 内在处理机制 | 线性轮询 | 回调callback |

|  |  |  |
| --- | --- | --- |
| 开发复杂性 | 低 | 中 |

几个重要的地址转换函数介绍

inet\_addr(), inet\_aton(), inet\_ntoa()和inet\_ntop(), inet\_pton()

inet\_addr()函数介绍如下：

|  |  |
| --- | --- |
| 头文件 | arpa/inet.h |

|  |  |
| --- | --- |
| 功能 | inet\_addr()函数用于将点分十进制IP地址转换成网络字节序IP地址; |

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

|  |  |
| --- | --- |
| 返回值 | 如果正确执行将返回一个无符号长整数型数。如果传入的字符串不是一个合法的IP地址，将返回INADDR\_NONE; |

inet\_aton()函数介绍如下：

|  |  |
| --- | --- |
| 头文件 | sys/socket.h |

|  |  |
| --- | --- |
| 功能 | inet\_aton()函数用于将点分十进制IP地址转换成网络字节序IP地址; |

|  |  |
| --- | --- |
| 原型 | int inet\_aton(const char \*string, struct in\_addr \*addr); |

|  |  |
| --- | --- |
| 返回值 | 如果这个函数成功，函数的返回值非零，如果输入地址不正确则会返回零; |

inet\_ntoa()函数介绍如下：

|  |  |
| --- | --- |
| 头文件 | arpa/inet.h |

|  |  |
| --- | --- |
| 功能 | inet\_ntoa()函数用于网络字节序IP转化成点分十进制IP; |

|  |  |
| --- | --- |
| 原型 | char \*inet\_ntoa (struct in\_addr); |

|  |  |
| --- | --- |
| 返回值 | 若无错误发生，inet\_ntoa()返回一个字符指针。否则的话，返回NULL。其中的数据应在下一个WINDOWS套接口调用前复制出来; |

应用实例：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/socket.h>  #include <arpa/inet.h>  int main()  {  int i;  char lo[] = "127.0.0.1";  struct in\_addr netAddr;  netAddr.s\_addr = inet\_addr(lo);  printf("NetIP: 0x%x\n", netAddr.s\_addr);  char \*strAddr = inet\_ntoa(netAddr);  printf("StrIP: %s\n", strAddr);  int ret = inet\_aton(strAddr, &netAddr);  printf("NetIP: 0x%x\n", netAddr.s\_addr);  return 0;  } |

输出结果：

|  |
| --- |
| NetIP: 0x100007f(网络序)  StrIP: 127.0.0.1  NetIP: 0x100007f(网络序) |

inet\_ntop()函数介绍如下：

|  |  |
| --- | --- |
| 头文件 | arpa/inet.h |

|  |  |
| --- | --- |
| 功能 | inet\_ntop()函数用于将网络字节序的二进制地址转换成文本字符串; |
| 原型 | const char \*inet\_pton(int domain, const void \*restrict addr, char \*restrict str, socklen\_t size); |

|  |  |
| --- | --- |
| 返回值 | 若成功，返回地址字符串指针；若出错，返回NULL; |

inet\_pton()函数介绍如下：

|  |  |
| --- | --- |
| 头文件 | arpa/inet.h |

|  |  |
| --- | --- |
| 功能 | inet\_pton()函数用于将文本字符串格式转换成网络字节序二进制地址; |
| 原型 | int inet\_pton(int domain, const char \*restrict str, void \*restrict addr); |

|  |  |
| --- | --- |
| 返回值 | 若成功，返回1；若格式无效，返回0；若出错，返回-1; |

应用实例：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <arpa/inet.h>  int main()  {  struct in\_addr addr;  if(inet\_pton(AF\_INET, "127.0.0.1", &addr.s\_addr) == 1)  printf("NetIP: %x\n", addr.s\_addr);  char str[20];  if(inet\_ntop(AF\_INET, &addr.s\_addr, str, sizeof str))  printf("StrIP: %s\n", str);  return 0;  } |

输出结果：

|  |
| --- |
| NetIP: 100007f  StrIP: 127.0.0.1 |