linux Socket-应用编程-专题讲座

written by 王保明

# Socket编程基本实践

## 1Socket Api基本概念

|  |
| --- |
| 什么是socket？   * socket可以看成是用户进程与内核网络协议栈的编程接口。 * socket不仅可以用于本机的进程间通信，还可以用于网络上不同主机的进程间通信。 |
| tcp/ip通讯模型 |
| **IPv4套接口地址结构**   * IPv4套接口地址结构通常也称为“网际套接字地址结构”，它以“sockaddr\_in”命名，定义在头文件<netinet/in.h>中   struct sockaddr\_in {  uint8\_t sin\_len; 4  sa\_family\_t sin\_family; 4  in\_port\_t sin\_port; 2  struct in\_addr sin\_addr; 4  char sin\_zero[8]; 8  };   * sin\_len：整个sockaddr\_in结构体的长度，在4.3BSD-Reno版本之前的第一个成员是sin\_family. * sin\_family：指定该地址家族，在这里必须设为AF\_INET * sin\_port：端口 * sin\_addr：IPv4的地址； * sin\_zero：暂不使用，一般将其设置为0 |
| **通用地址结构**   * 通用地址结构用来指定与套接字关联的地址。unix域、tcpip域的结构强转为通用结构后传入socketapi中，查看结构 man 7 ip   struct sockaddr {  uint8\_t sin\_len;  sa\_family\_t sin\_family;  char sa\_data[14]; //14  };   * sin\_len：整个sockaddr结构体的长度 * sin\_family：指定该地址家族 * sa\_data：由sin\_family决定它的形式。 |
| **网络字节序**   * 字节序   大端字节序（Big Endian)  最高有效位（MSB：Most Significant Bit）存储于最低内存地址处，最低有效位（LSB：Lowest Significant Bit）存储于最高内存地址处。   * 小端字节序（Little Endian）   最高有效位（MSB：Most Significant Bit）存储于最高内存地址 处，最低有效位（LSB：Lowest Significant Bit）存储于最低内存地址处。   * 主机字节序   不同的主机有不同的字节序，如x86为小端字节序，Motorola 6800为大端字节序，ARM字节序是可配置的。   * 网络字节序   网络字节序规定为大端字节序  int a = 0x12345678;  char \*p = &a;  p[0] == 0x78 ? little : big; |
| **字节序转换函数**   * uint32\_t htonl(uint32\_t hostlong); * uint16\_t htons(uint16\_t hostshort); * uint32\_t ntohl(uint32\_t netlong); * uint16\_t ntohs(uint16\_t netshort); * 说明：在上述的函数中，h代表host；n代表network s代表short；l代表long |
| **地址转换函数**   * #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); |
| **套接字类型**   * 流式套接字(SOCK\_STREAM)   提供面向连接的、可靠的数据传输服务，数据无差错，无重复的发送，且按发送顺序接收。   * 数据报式套接字(SOCK\_DGRAM)   提供无连接服务。不提供无错保证，数据可能丢失或重复，并且接收顺序混乱。   * 原始套接字（SOCK\_RAW） |
|  |
|  |

## 2SocketApi基本编程模型

|  |
| --- |
| **TCP客户/服务器模型**        **简单服务器模型** |
|  |

## 3Socket Api基本实践

### Socket API基本用法

|  |
| --- |
| **socket函数**   * 包含头文件<sys/socket.h> * 功能：创建一个套接字用于通信 * 原型   + int socket(int domain, int type, int protocol); * 参数   + domain ：指定通信协议族（protocol family）   + type：指定socket类型，流式套接字SOCK\_STREAM，数据报套接字SOCK\_DGRAM，原始套接字SOCK\_RAW   + protocol ：协议类型 * 返回值：成功返回非负整数， 它与文件描述符类似，我们把它称为套接口描述字，简称套接字。失败返回-1 |
| **bind函数**   * 包含头文件<sys/socket.h> * 功能：绑定一个本地地址到套接字 * 原型   + int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen); * 参数   + sockfd：socket函数返回的套接字   + addr：要绑定的地址   + addrlen：地址长度 * 返回值：成功返回0，失败返回-1 |
| **listen函数**   * 一般来说，listen函数应该在调用socket和bind函数之后，调用函数accept之前调用。   1. 一旦调用listen函数，这个套接字sockfd将变成被动套接字，只能接受链接（侦听），不能主动发起链接，不能与客户端进行通信。   * 对于给定的监听套接口，内核要维护两个队列：   + 1、已由客户发出并到达服务器，服务器正在等待完成相应的TCP三路握手过程   + 2、已完成连接的队列 |
| **accept函数**   * 包含头文件<sys/socket.h> * 功能：从已完成连接队列返回第一个连接，如果已完成连接队列为空，则阻塞。 * 原型   + int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); * 参数   + sockfd：服务器套接字   + addr：将返回对等方的套接字地址   + addrlen：返回对等方的套接字地址长度 * 返回值：成功返回非负整数，失败返回-1   **connect函数**   * 包含头文件<sys/socket.h> * 功能：建立一个连接至addr所指定的套接字 * 原型   + int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen); * 参数   + sockfd：未连接套接字   + addr：要连接的套接字地址   + addrlen：第二个参数addr长度 * 返回值：成功返回0，失败返回-1   ps -u ussr\_name 查看用户进程  netstat -an | grep port 查看由系统内核协议栈维护的tcpip状态  netstat -an | grep tcp  系统调用说明：  1. 内核的tcpip协议栈维护tciip的状态和队列，应用程序通过系统调用设置和获取内核中tcpip的状态和队列信息。  2. 客户端 conncet -> 内核 -> 三次握手以及维护相应的状态 - 调用了 listen 的服务器端 -> 内核维护连接队列 -> 服务器端调用 accept 获取内核维护的已连接队列信息即socket描述字  例子：若多个客户端connect同一个服务端，但是服务端只调用了一次accept，多个客户端的connect是成功的，但是只能由一个正常工作，因为内核维护了多个客户端的三次握手，established，但是服务端没有全部accept，所以服务端应用不会处理客户端的请求。 |

### Socket API 中的地址复用

|  |
| --- |
| **SO\_REUSEADDR**   * 服务器端尽可能使用SO\_REUSEADDR * 在绑定之前尽可能调用setsockopt来设置SO\_REUSEADDR套接字选项。   使用SO\_REUSEADDR选项可以使得不必等待TIME\_WAIT状态消失就可以重启服务器  int optval = 1;  if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval)) < 0)  {  perror("setsockopt bind\n");  exit(0);  }  服务端、客户端分别重启测试 |

### Socket服务支持多并发（多客户端连接）

|  |
| --- |
| **分析最基本socket服务器客户端模型能否支持多客户端连接** |
|  |
|  |

### 点对点聊天程序设计与实现

|  |
| --- |
| 点对点聊天程序，功能说明  点对点聊天程序，设计思想 |
| 注意：客户端退出、服务器端退出，对各自的影响；分析父子进程生命周期关系。  注意：体会长链接和短连接。。。。 |
| 客户端实现思路：  void myhanldle(int sig)  {  printf("resv sig:%d 客户端父进程也退出\n", sig);  exit(0);  }  int main(void )  {  int sock;    sock = socket(PF\_INET, SOCK\_STREAM, 0);  //listenfd = socket(AF\_INET, IPPROTO\_TCP, 0);  if (sock < 0)  {  perror("socket:");  exit(0);  }    struct sockaddr\_in servaddr;  servaddr.sin\_family = AF\_INET;  servaddr.sin\_port = htons(8089);  servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); //绑定服务器任意一个i  //inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/    if ( connect(sock, (struct sockaddr \*)&servaddr, sizeof(servaddr)) < 0)  {  perror("connect:");  exit(0);  }  pid\_t pid;  pid = fork();  if (pid == -1)  {  perror("fork err:");  close(sock);  exit(0);  }    if (pid == 0)  {  char recvbuf[1024] = {0};  while (1)  {  memset(recvbuf, 0, sizeof(recvbuf));  int ret = read(sock, recvbuf, sizeof(recvbuf));  if (ret == -1)  {  perror("read err \n");  break;  }  else if (ret == 0)  {  printf("peer close \n");  break;  }  fputs(recvbuf, stdout);  }  close(sock);  //客户端检测到对方退出以后，发信号给父进程退出  kill(getppid(), SIGUSR1);    printf("客户端子进程退出\n");  exit(0);  }  else  {  char sendbuf[1024] = {0};  signal(SIGUSR1, myhanldle);  printf("客户端父进程要求输入数据:\n");  while ( fgets(sendbuf, sizeof(sendbuf), stdin ) != NULL )  {  printf("客户端获取数据长度:%d \n", strlen(sendbuf));  write(sock, sendbuf, strlen(sendbuf));  memset(sendbuf, 0, sizeof(sendbuf));  }  printf("客户端父进程退出\n");  printf("parent close\n");    //exit(EXIT\_SUCCESS);  }    //close(sock);  return 0;  } |
| 服务器端实现思路：  void mysighandler\_t(int sig)  {  printf("child resv sig: %d ,child quit\n", sig);  exit(0);  }  int main(void )  {  int listenfd;    listenfd = socket(PF\_INET, SOCK\_STREAM, 0);  //listenfd = socket(AF\_INET, IPPROTO\_TCP, 0);  if (listenfd < 0)  {  printf("func socket() err\n");  exit(0);  }    struct sockaddr\_in servaddr;  servaddr.sin\_family = AF\_INET;  servaddr.sin\_port = htons(8089);  //servaddr.sin\_addr.s\_addr = inet\_addr(INADDR\_ANY); 绑定服务器任意一个ip  servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); //绑定服务器任意一个ip  //inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/    int optval = 1;  //每个级别SOL\_SOCKET下，也有很多选项 不同的选择会有不同的结构  if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval)) < 0)  {  perror("setsockopt");  exit(0);  }    if ( bind(listenfd, (struct sockaddr \* )&servaddr, sizeof(servaddr)) < 0 )  {  perror("bind");  exit(0);  }    if ( listen(listenfd, SOMAXCONN) < 0)  {  perror("listen");  exit(0);  }    int conn;    struct sockaddr\_in peeraddr;  socklen\_t addrlen;  memset(&peeraddr, 0, sizeof(struct sockaddr\_in ));  addrlen = sizeof(struct sockaddr\_in );    conn = accept(listenfd, (struct sockaddr \*)&peeraddr, &addrlen );  if (conn < 0)  {  perror("accept");  exit(0);  }    printf("perradd:%s peerport:%d \n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));    int pid = fork ();  if (pid == -1)  {  perror("fork err:"); //创建子进程失败  close(conn); //关闭conn套接字  close(listenfd); //关闭监听套接字  exit(0);  }  close(listenfd); //关闭监听套接字    if (pid == 0)  {  signal(SIGUSR1, mysighandler\_t);      char sendbuf[1024] = {0};  while ( fgets(sendbuf, sizeof(sendbuf), stdin) !=NULL )  {  write(conn, sendbuf, strlen(sendbuf));  memset(sendbuf, 0, sizeof(sendbuf));  }  printf("child close\n");  exit(EXIT\_SUCCESS);  }  else if (pid > 0)  {  char recbuf[1024];  while (1)  {  memset(recbuf, 0, sizeof(recbuf));  //如果对方退出，捕捉  int ret = read(conn, recbuf, sizeof(recbuf));  if (ret == 0)  {  printf("ret == 0 peer close退出\n");  break; //去掉做实验  }  else if (ret < 0)  {  perror("ret<0");  break;  }  fputs(recbuf, stdout);  }  kill(pid, SIGUSR1);  printf("parent quit\n");  close(conn);  exit(0);  }    return 0;  } |
| 当客户端调用close或者挂掉的时候，内核会向服务端发送fin，一个含有\0的报文段（流协议），服务端内核检测到后，read返回0，得知对端关闭 |
|  |
|  |

## 4Socket Api编程进价

### 1流协议与粘包

|  |
| --- |
| **流协议与粘包**  c |
| 粘包产生的原因 |
| 说明  tcp 字节流 无边界  udp 消息、数据报 有边界  对等方，一次读操作，不能保证完全把消息读完。  对方接受数据包的个数是不确定的。 |
| 产生粘包问题的原因  1、SQ\_SNDBUF 套接字本身有缓冲区 （发送缓冲区、接收缓冲区）  2、tcp传送的网络数据最大值MSS大小限制  3、链路层也有MTU（最大传送单元）大小限制，如果数据包大于>MTU要在IP层进行分片，导致消息分割。（可以简单的认为MTU是MSS加包头数据。）  4、tcp的流量控制和拥塞控制，也可能导致粘包  5、tcp延迟发送机制 等等  结论：tcp/ip协议，在传输层没有处理粘包问题。 |
| **粘包解决方案**   * 本质上是要在应用层维护消息与消息的边界   + 定长包   + 包尾加\r\n（ftp）   + 包头加上包体长度   + 更复杂的应用层协议 |

### 2包头加上包体长度编程实践

|  |
| --- |
| 。 |
| **编程实践**   * readn * written * 发送数据：前4个字节代表报文长度（网络字节序）+报文内容 * 接受数据：先读4个字节，然后根据报文长度，再度内容。 |
| //1一次全部读走 //2次读完数据 //出错分析 //对方已关闭  //思想：tcpip是流协议，不能保证1次读操作，能全部把报文读走，所以要循环读指定长度的数据。  //按照count大小读数据，  //若读取的长度ssize\_t<count 说明读到了一个结束符，对方已关闭。  //@ssize\_t：返回读的长度 若ssize\_t<count 读失败失败  //@buf：接受数据内存首地址  //@count：接受数据长度  ssize\_t readn(int fd, void \*buf, size\_t count)  {  size\_t nleft = count;  ssize\_t nread;  char \*bufp = (char\*)buf;  while (nleft > 0)  {  if ((nread = read(fd, bufp, nleft)) < 0)  {  if (errno == EINTR)  continue;  return -1;  }  else if (nread == 0) //若对方已关闭  return count - nleft;  bufp += nread;  nleft -= nread;  }  return count;  }  //1一次全部读走 //2次读完数据 //出错分析 //对方已关闭  //思想：tcpip是流协议，不能1次把指定长度数据，全部写完  //按照count大小写数据  //若读取的长度ssize\_t<count 说明读到了一个结束符，对方已关闭。  //@ssize\_t：返回写的长度 -1失败  //@buf：待写数据首地址  //@count：待写长度  ssize\_t writen(int fd, const void \*buf, size\_t count)  {  size\_t nleft = count;  ssize\_t nwritten;  char \*bufp = (char\*)buf;  while (nleft > 0)  {  if ((nwritten = write(fd, bufp, nleft)) < 0)  {  if (errno == EINTR)  continue;  return -1;  }  else if (nwritten == 0)  continue;  bufp += nwritten;  nleft -= nwritten;  }  return count;  } |

### 3包尾加上\n编程实践

|  |
| --- |
| **\n作为协议的边界**   * ssize\_t recv(int s, void \*buf, size\_t len, int flags); * 与read相比，只能用于套接字文件描述符；read & write 通用，可用于socket/文件/标准输入输出 * recv & send socket * 多了一个flags   MSG\_OOB  This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at thehead of the normal data queue, and thus this flag cannot be used with such protocols.  带外数据 紧急指针  MSG\_PEEK  This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, asubsequent receive call will return the same data.  可以读数据，不从缓存区中读走，利用此特点可以方便的实现按行读取数据。  一个一个字符的读，方法不好；多次调用系统调用read方法 |
| recv函数会将套接字缓冲区中的内容读出，但不清空，与read函数的区别在此。此函数有一个flag标志位，设为MSG\_PEEK。偷窥操作，速度很快，内核只是拷贝一份数据不用维护移除数据的各种安全机制。  send函数会将缓冲区中的内容写入到套接字，也不清空，与write函数的区别在此。  用这两个函数可以先接收或发送缓冲区中的内容，然后再用readn（此时缓冲区中的内容依然存在）与write函数去继续判断换行符/n，对缓冲区内容实现换行输出。 |
| 参考例题： |
| fgets按行读数据，自动末尾加上\n |

### 4域名服务相关函数

getsockname & getpeername

|  |
| --- |
| 获取本地的地址（注意是已连接套接字）  NAME  getsockname - get socket name  SYNOPSIS  #include <sys/socket.h>  int getsockname(int s, struct sockaddr \*name, socklen\_t \*namelen); |
| struct sockaddr\_in localaddr;  socklen\_t addrlen = sizeof(localaddr);  //获取本地的地址  if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)  ERR\_EXIT("getsockname");  printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port)); |
|  |

gethostname、gethostbyname、gethostbyaddr

|  |
| --- |
| * gethostname、gethostbyname、gethostbyaddr |
| The domain name queries carried out by gethostbyname() and gethostbyaddr() use a combination of any or all of the name server named(8), a broken out  line from /etc/hosts, and the Network Information Service (NIS or YP), depending upon the contents of the order line in /etc/host.conf. The default  action is to query named(8), followed by /etc/hosts.  The hostent structure is defined in <netdb.h> as follows:  struct hostent {  char \*h\_name; /\* official name of host \*/  char \*\*h\_aliases; /\* alias list \*/  int h\_addrtype; /\* host address type \*/  int h\_length; /\* length of address \*/  char \*\*h\_addr\_list; /\* list of addresses \*/  }  #define h\_addr h\_addr\_list[0] /\* for backward compatibility \*/  The members of the hostent structure are: |
| int getlocalip(char \*ip)  {  char host[100] = {0};  if (gethostname(host, sizeof(host)) < 0)  return -1;  struct hostent \*hp;  if ((hp = gethostbyname(host)) == NULL)  return -1;  strcpy(ip, inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr));  return 0;  }  int main(void)  {  char host[100] = {0};  if (gethostname(host, sizeof(host)) < 0)  ERR\_EXIT("gethostname");    printf("\n\n\nhost:%s \n", host);  struct hostent \*hp;  if ((hp = gethostbyname(host)) == NULL)  ERR\_EXIT("gethostbyname");  int i = 0;  while (hp->h\_addr\_list[i] != NULL)  {  //char \*inet\_ntoa(struct in\_addr in); 要求填入一个结构体元素  printf("%s\n", inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr\_list[i]));  i++;  }    char ip[16] = {0};  getlocalip(ip);  printf("localip=%s\n", ip);  return 0;  }  说明：  本机可能有很多ip，一般在connect或者accept后调用，查看与对端建立连接的本端ip和端口等信息，  当accept 时没有 传入参数获取对端地址信息的时候，可以在下面调用getpeername 获取。 |

gdb -tui -q 进程