本章笼统归为“高级I/O”的各个函数和技术，包含三部分内容。第一，在I/O操作上设置超时，有三种方法；第二，read与write函数的变体，recv与send、readv与writev、recvmsg与sendmsg。第三，确定套接字缓冲区数据量和其他有关说明。

**1、套接字超时**

套接字I/O操作上设置超时有三种方法：

（1）调用alarm在指定超时期满产生SIGALRM信号。涉及系统信号处理，不同系统实现存在差异，且可能干扰进程中现有的alarm调用。

（2）在select上阻塞等待I/O，以代替直接阻塞在read或write调用上。

（3）使用SO\_RCVTIMEO和SO\_SNDTIMEO套接字选项。

以上三个方法都适用于输入和输出操作（例如read、write及其诸如recvfrom、sento之类的变体）。**select可以用来在connect上设置超时的先决条件是套接字处于非阻塞模式**。**前两个技术适用于任何描述符**，**第三个技术仅用于套接字描述符**，并且这两个套接字选项对connect不适用。

**1.1 使用SIGALRM实现套接字超时**

**1.1.1 为connect设置超时**

在调用connect之前设立一个SIGALRM信号处理函数，并设定一个报警时间nsec，当调用connect时间超过报警时间，则进入信号处理函数，并退出当前connect\_timeo\_sigalrm函数。当调用connect过程中被中断会返回EINTR错误，将errno设置成ETIOMEOUT，避免三次握手继续进行。

|  |
| --- |
| static void sigalarm\_handle (int signo)  {  return;  }  static int connect\_timeo\_sigalrm(int sock\_fd, const sockaddr\* addr, socklen\_t socklen, int nsec)  {  \_\_sighandler\_t sigfunc = signal(SIGALRM, sigalarm\_handle);  int ret;    if(alarm(nsec) != 0) LOG("connect timeout: alarm was already set");    ret = connect(sock\_fd, addr, socklen);  if( ret < 0 ){  //close(sock\_fd);  if(errno == EINTR) errno = ETIMEDOUT;  }  alarm(0);  signal(SIGALRM, sigfunc); // restore previous signal handler  return ret;  } |

当前方法**能够减少connect的超时时限，但无法延长内核现有的超时**。内核中connect默认的超时时间是75s，如果这里设置alarm报警时间超过75，connect仍将在75s后发生超时。

**1.1.2 为recvform设置超时**

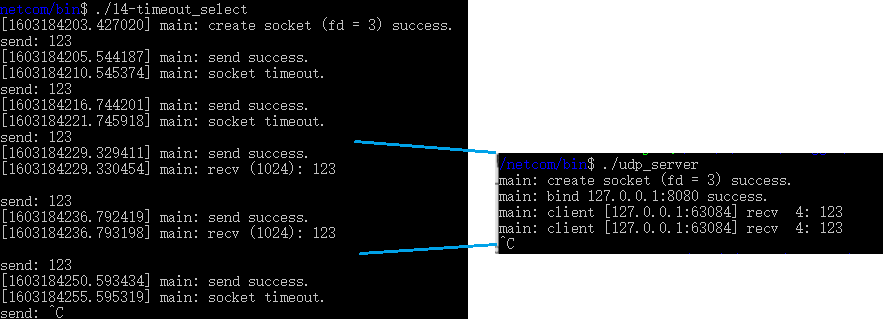
|  |
| --- |
| static int recvform\_timeo\_sigalrm(int socket\_fd, int nsec)  {  char buf[1024];  signal(SIGALRM, sigalarm\_handle);  while (1)  {  alarm(nsec);  int len = ::recvfrom(socket\_fd, buf, sizeof(buf), 0, NULL, NULL);  if (len < 0){  LOG("recv failed.");  if (errno == EINTR)  LOG("timeout. %s", strerror(errno));  else  break;  }else if(len == 0){  LOG("client closed.");  break;  }else{  buf[len] = '\0';  LOG("client recv %d: %s", len, buf);  alarm(0);  }  }  alarm(0);  return 0;  } |

**1.2 使用select为recvfrom设置超时**

正常调用recvfrom时会阻塞直到数据到来，使用select检查套接字是否可读，之后再执行recvfrom接收数据。该**select设置超时的方法能使用在任何描述符上**。

|  |
| --- |
| static int readable\_timeo(int fd, int sec)  {  fd\_set rset;  timeval tv;  FD\_ZERO(&rset);  FD\_SET(fd,&rset);  tv.tv\_sec = sec;  tv.tv\_usec = 0;  return select(fd+1, &rset, NULL, NULL, &tv);  }  int main()  {  /// 1、创建socket  int socket\_fd = socket(AF\_INET,SOCK\_DGRAM, 0); // udp  if (socket\_fd == -1){  LOG("create socket failed. %s", strerror(errno));  return 1;  }else{  LOG("create socket (fd = %d) success.", socket\_fd);  }  /// 2、连接服务端  sockaddr\_in servaddr;  servaddr.sin\_family = AF\_INET;  inet\_pton(servaddr.sin\_family, "127.0.0.1", &servaddr.sin\_addr);  servaddr.sin\_port = htons(8080);  // 3、 发送到服务端  int len ;  char buf[1024];  while (1)  {  printf("send: ");  fgets(buf, sizeof(buf), stdin); // 等待用户输入  if (strcmp(buf, "exit") == 0)  break;  // 发送  len = ::sendto(socket\_fd, buf, strlen(buf),0, (sockaddr\*)&servaddr, sizeof(servaddr));  if(len < 0){  LOG("send failed.Exit. %s", strerror(errno));  break;  }  LOG("send success.");  // 接收  if( readable\_timeo(socket\_fd, 5) == 0){  LOG("socket timeout.");  }  else{  len = ::recvfrom(socket\_fd, buf, sizeof(buf), 0, NULL, NULL);  if(len < 0){  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d): %s", len, buf);  }  }  /// 4、关闭连接  ::close(socket\_fd);  } |

未启动服务端时，发送两次消息，等待5秒未收到回复输出超时；启动服务端后，发送消息能立即收到回复。如图所示。



**1.3 使用SO\_RCVTIMEO套接字选项为recvfrom设置超时**

**使用select设置超时，需要在每一次调用recvfrom时都检测一次**。一旦对套接字描述符使用**SO\_RCVTIMEO套接字选项，其超时设置将应用于该描述符上的所有读操作**，不需要重复设置。同样的，SO\_SNDTIMEO选项将应用超时设置在所有写操作上。这**两个选项仅能用于套接字描述符，但是都不能用于为connect设置超时**。

|  |
| --- |
| static int setsockopt\_rcvtimeo(int fd, int sec)  {  timeval tv;  tv.tv\_sec = sec;  tv.tv\_usec = 0;  return setsockopt(fd, SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv));  } |

|  |
| --- |
| // 接收  len = ::recvfrom(socket\_fd, buf, sizeof(buf), 0, NULL, NULL);  if(len < 0){  if(errno == EWOULDBLOCK){ // 套接字超时，不退出  LOG("recv timeout.");  continue;  }  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d): %s", len, buf); |

实际的处理结果同1.2节中使用select对recvfrom设置超时。

**2、recv和send函数**

**2.1 函数原型**

#include <sys/socket.h>

ssize\_t recv(int sockfd, void \*buff, size\_t nbytes, int flags);

ssize\_t send(int sockfd, const void \* buff, size\_t nbytes, int flags);

//返回：若成功则为读入或写出的字节数，若出错返回-1

函数recv和send的前三个参数等同于read和write的前三个参数。flags参数的值或为0，或为下表中的一个或多个值的逻辑或。

|  |  |  |  |
| --- | --- | --- | --- |
| flag | 说明 | recv | send |
| MSG\_DONTROUTE | 绕过路由表查找本标志高速内核目的主机在某个直接连接的本地网络上，因为无需执行路由表查找。  *可以使用SO\_DONTROUTE套接字对某个套接字上的所有输出操作都开启。* | ○ | ● |
| MSG\_DONTWAIT | 仅本操作非阻塞本标志在无需打开响应套接字的非阻塞标志的前提下，把单个IO操作临时定位非阻塞，接着执行IO操作，然后关闭非阻塞标志。 | ● | ● |
| MSG\_OOB | 发送或接收带外数据对于send，本标志指明即将发送带外数据。TCP连接上只有一个字节可以作为带外数据发送。对于recv本标志指明即将读入的是带外数据而不是普通数据。 | ● | ● |
| MSG\_PEEK | 窥看外来消息本标志适用于recv和recvfrom，它允许我们查看已可读取的数据，而且系统不再recv或recvfrom返回后丢弃这些数据。（后面小节中讨论） | ● | ○ |
| MSG\_WAITALL | 等待所有数据，它告知内核不要在尚未读入请求数目的字节之前让一个读操作返回。 | ● | ○ |

参数flags是一个值参数，只能用于从用户进程内核传递标志，而不能反向传回参数。在新的OSI协议中提出了随输入操作向用户进程返送MSG\_EOR标志的需求，决定保持常用函数（recv和recvfrom）的参数不变，改用**recvmsg和sendmsg所用的msghdr结构，其结构中新增一个msg\_flags的值-结果参数，以实现内核返回修改后的标志**。

**3、readv和writev函数**

这两个函数有助于提高数据通信效率，它们能对数据进行整合传输及发送，适当使用这2个函数可以减少I/O函数的调用次数。

**3.1 函数原型**

#include <sys/uio.h>

struct iovec {

void \*iov\_base; //buffer的起始地址

size\_t iov\_len; //buffer数量

};

ssize\_t readv(int filedes, const struct iovec \* iov, int iovcnt);

ssize\_t writev(int filedes, const struct iovec \* iov, int iovcnt);

//返回: 读到或写出的字节数，出错时为-1//返回：若成功则为读入或写出的字节数，若出错返回-1

两个函数类似于read和write，**不过readv和writev允许单个系统调用读入或写出自1个或多个缓冲区**。来自读操作的输入数据分散到多个应用缓冲区中，称为**分散读**；来自多个应用缓冲区的输出数据被集中起来供给单个写操作，分为**集中写**。

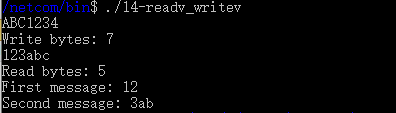
两个函数的第二个参数都是指向某个iovec结构数组的一个指针，数组中元素的范围存在限制，取决于具体实现定义为宏IOV\_MAX。

**readv和writev两个函数可用于任何描述符**，不仅限于套接字描述符。另外writev是原子操作，一次writev调用只产生单个UDP数据报。对于TCP\_NODELAY套接字选项，避免write多个数据触发Nagle算法，首先办法之一是针对多个数据调用writev。

**3.2 示例**

|  |
| --- |
| #include <stdio.h>  #include <sys/uio.h>  int test\_readv()  {  struct iovec vec[2];  char buf1[] = "ABCDEFG";  char buf2[] = "1234567";  int str\_len;  vec[0].iov\_base = buf1;  vec[0].iov\_len = 3;  vec[1].iov\_base = buf2;  vec[1].iov\_len = 4;  size\_t iovLen= sizeof(vec)/sizeof(vec[0]);    // 多个缓冲中的数据一次写出  str\_len = writev(1, vec, iovLen); // fileno(stdout) = 1 ,是系统标准输出文件描述符  puts("");  printf("Write bytes: %d \n", str\_len);  return 0;  }  int test\_writev()  {  struct iovec vec[2];  char buf1[100] = {};  char buf2[100] = {};  int str\_len;  vec[0].iov\_base = buf1;  vec[0].iov\_len = 2;  vec[1].iov\_base = buf2;  vec[1].iov\_len = 3;  size\_t iovLen= sizeof(vec)/sizeof(vec[0]);  //把数据放到多个缓冲中储存  str\_len = readv(2, vec, iovLen); // fileno(stdout) =2 是从标准输入接收数据  printf("Read bytes: %d \n", str\_len);  printf("First message: %s \n", buf1);  printf("Second message: %s \n", buf2);  return 0;  }  int main()  {  test\_readv();  test\_writev();  } |

运行交互结果截图



**4、recvmsg和sendmsg函数**

最通用的I/O函数，recvmsg可以替换使用read、readv、recv和recvfrom读入函数，类似地sendmsg可以替换各种输出函数。

**4.1 函数原型**

#include <sys/socket.h>

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

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

//返回: 成功时为读入或写出的字节数，出错时为-1

两个参数将大部分参数封装到一个msghdr结构中

struct msghdr {

void \*msg\_name; /\* 协议地址 \*/

socklen\_t msg\_namelen; /\* 协议地址长度 \*/

struct iovec \*msg\_iov; /\* 分散读/写的iovec数组结构地址 \*/

int msg\_iovlen; /\* iovec数组元素个数 \*/

void \*msg\_control; /\* 辅助数据结构数组(cmsghdr struct)的地址 \*/

socklen\_t msg\_controllen; /\* 辅助数据结构数组的元素个数\*/

int msg\_flags; /\* 用于recvmsg()的标志，sendmsg()忽略 \*/

};

* msg\_name 和 msg\_namelen 这两个成员用于套接字未连接的场合（如未连接 UDP 套接字）。它们类似 recvfrom 和 sendto 的第五个和第六个参数：
  + msg\_name 指向一个套接字地址结构，调用者在其中存放接收者（对于 sendmsg 调用）或发送者（对于recvmsg调用）的协议地址。如果无需指明协议地址（如对于 TCP 套接字或已连接 UDP 套接字），msg\_name 应置为空指针。
  + msg\_namelen 对于 sendmsg 是一个值参数，对于 recvmsg 却是一个值-结果参数。
* msg\_iov 和 msg\_iovlen 这两个成员指定输入或输出缓冲区数组（即iovec结构数组），类似 readv 或 writev 的第二个和第三个参数。
* msg\_control 和 msg\_controllen 这两个成员指定可选的辅助数据的位置和大小。msg\_controllen 对于 recvmsg 是一个值-结果参数。

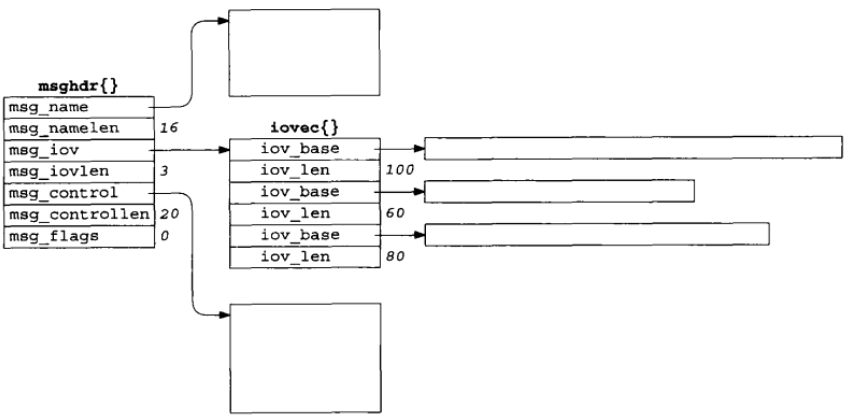
对于 recvmsg 和 sendmsg，必须区别它们的两个标志变量：一个是传递值的 flags 参数；另一个是所传递 msghdr 结构的 msg\_flags 成员，它传递的是引用，因为传递给函数的是该结构的地址。

* 只有 recvmsg 使用 msg\_flags 成员。recvmsg 被调用时，flags 参数被复制到 msg\_flags 成员，并由内核使用其值驱动接收处理过程。内核还依据 recvmsg 的结果更新 msg\_flags 成员的值。
* sendmsg 则忽略 msg\_flags 成员，因为它直接使用 flags 参数驱动发送处理过程。这一点意味着如果想在某个 sendmsg 调用中设置 MSG\_DONTWAIT 标志，那就把 flags 参数设置为该值，把 msg\_flags 成员设置为该值不起作用。

recvmsg 返回的 7 个标志如下：

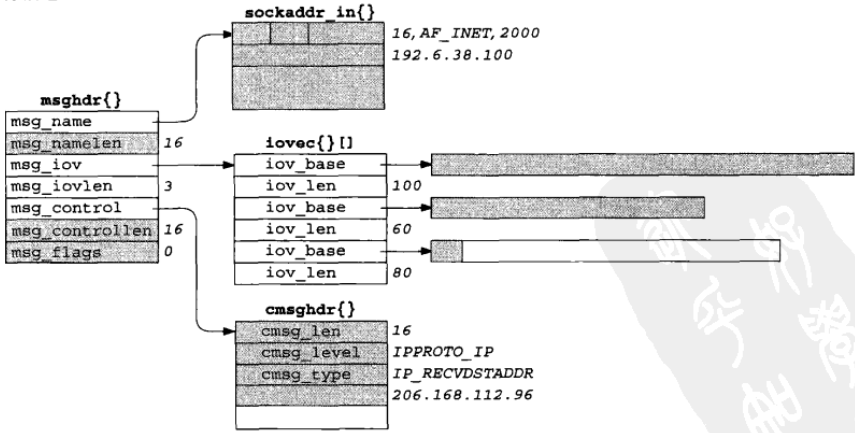
* MSG\_BCAST：本标志随 BSD/OS 引入，相对较新。它的返回条件是本数据包作为链路层广播收取或者其目的 IP 地址是一个广播地址。与 IP\_RECVD-STADDR 套接字选项相比，本标志是用于判定一个 UPD 数据包是否发往某个广播地址的更好方法。
* MSG\_MCAST：本标志随 BSD/OS 引入，相对较新。它的返回条件是本数据报作为链路层多播收取。
* MSG\_TRUNC：本标志的返回条件是本数据报被截断，也就是说，内核预备返回的数据超过进程事先分配的空间（所有 iov\_len 成员之和）。
* MSG\_CTRUNC：本标志的返回条件是本数据报的辅助数据被截断，也就是说，内核预备返回的辅助数据超过进程事先分配的空间（msg\_controllen）。
* MSG\_EOR：本标志的返回条件是返回数据结束一个逻辑记录。TCP 不使用本标志，因为它是一个字节流协议。
* MSG\_OOB：本标志绝不为 TCP 带外数据返回。它用于其他协议族（如 OSI 协议族）。
* MSG\_NOTIFICATION：本标志由 SCTP 接收者返回，指示读入的消息是一个事先通知，而不是数据消息。

下图展示了一个 msghdr 结构以及它指向的各种信息。图中假设进程即将对一个 UDP 套接字调用 recvmsg：



图中协议地址分配了16个字节，辅助地址分配了20个字节，为缓冲区初始化了一个3个iovec结构构成的数组：第一个缓冲区100字节，第二个缓冲区60个字节，第三个缓冲区80个字节。同时为套接字设置了IP\_RECVDSTADDR选项以接收读取UDP数据报的目的IP地址。

假设从198.6.38.100端口2000达到一个170字节的UDP数据报，它的目标IP地址为206.168.112.96。下图是recvmsg返回时msghdr结构体中的所有信息。



1)msg\_name成员指向的缓冲区被填以一个套接字地址结构，其中所有受到数据报的源IP地址和源UDP端口

2)msg\_namelen成员(值--结构参数)被更新为存放在msg\_name所指缓冲区中的数据量

3)所收取数据报的前100字节数据放在第一个缓冲区中，中间60字节放在第二个缓存区中，最后10个字节数据放在第三个缓冲区

4)msg\_control成员指向的缓冲区被填以一个cmsghdr结构，该cmsghdr结构中，cmsg\_len成员值为16，cmsg\_level成员值为IPPROTO\_IP，cmsg\_type成员值为IP\_RECVDSTADDR，随后4个字节存放所有收到UDP数据报的目的IP地址

5)msg\_controllen成员被更新为所存放辅助数据的实际数据量，也是一个值--结果参数

6)msg\_flags成员同样被recfmsg更新，不过没有标志返回给进程

**4.2 示例**

|  |
| --- |
| #include <stdio.h>  #include <sys/socket.h>  #include <arpa/inet.h> // sockaddr\_in, inet\_addr  #include <unistd.h> // close  #include <cstring> // memset, bezro, strerror  #include <errno.h>  #include <fcntl.h> // fcntl  #include <signal.h>  #include <sys/time.h>  #define LOG(fmt, arg...) \  do \  { \  struct timeval tv; \  gettimeofday(&tv, NULL); \  printf("[%ld.%06d] %s: " fmt "\n", tv.tv\_sec, tv.tv\_usec, \_\_func\_\_, ##arg); \  } while (0)  int main(){  /// 1、创建socket  int socket\_fd = socket(AF\_INET,SOCK\_DGRAM, 0); // udp  if (socket\_fd == -1){  LOG("create socket failed. %s", strerror(errno));  return 1;  }else{  LOG("create socket (fd = %d) success.", socket\_fd);  }  /// 2、连接服务端  sockaddr\_in servaddr;  servaddr.sin\_family = AF\_INET;  inet\_pton(servaddr.sin\_family, "127.0.0.1", &servaddr.sin\_addr);  servaddr.sin\_port = htons(8080);  // 3、 发送到服务端  int len ;  char buf[1024];  while (1)  {  printf("send: ");  fgets(buf, sizeof(buf), stdin); // 等待用户输入  if (strcmp(buf, "exit") == 0)  break;  // 发送  iovec iovectobuf;  iovectobuf.iov\_base = buf;  iovectobuf.iov\_len = strlen(buf);  msghdr msgto;  memset(&msgto, 0, sizeof(msgto));  msgto.msg\_name = &servaddr;  msgto.msg\_namelen = sizeof(servaddr);  msgto.msg\_iov = &iovectobuf;  msgto.msg\_iovlen = 1; // 1个发送缓冲区iovec  len = sendmsg(socket\_fd, &msgto, 0);  //len = ::sendto(socket\_fd, buf, strlen(buf),0, (sockaddr\*)&servaddr, sizeof(servaddr));  if(len < 0){  LOG("send failed.Exit. %s", strerror(errno));  break;  }  LOG("send success.");  // 接收  memset(buf,0, 20); // buf缓冲区清零，分两段使用    iovec iovecfrombuf[2];  iovecfrombuf[0].iov\_base = buf;  iovecfrombuf[0].iov\_len = 5;  iovecfrombuf[1].iov\_base = buf+10;  iovecfrombuf[1].iov\_len = 5;  sockaddr\_in clientaddr;  msghdr msgfrom;  memset(&msgfrom, 0, sizeof(msgfrom));  // msgfrom.msg\_name = &clientaddr; // 可选择是否需要解析对端地址, NULL为不需要  // msgfrom.msg\_namelen = sizeof(clientaddr);  msgfrom.msg\_iov = iovecfrombuf;  msgfrom.msg\_iovlen = 2; // 2个接收缓冲区iovec  len = recvmsg(socket\_fd, &msgfrom, 0);  //len = ::recvfrom(socket\_fd, buf, sizeof(buf), 0, NULL, NULL);  if(len < 0){  if(errno == EWOULDBLOCK){ // 套接字超时，不退出  LOG("recv timeout.");  continue;  }  LOG("recv failed. %s", strerror(errno));  break;  }  //char client\_ip[INET6\_ADDRSTRLEN];  //inet\_ntop(AF\_INET, &clientaddr.sin\_addr, client\_ip, sizeof(clientaddr));  //int port = ntohs(clientaddr.sin\_port);  //LOG("recv %s:%d(%d): msg1=%s, msg2=%s", client\_ip, port, len,  LOG("recv (%d): msg1=%s, msg2=%s", len,  iovecfrombuf[0].iov\_base, iovecfrombuf[1].iov\_base);  }  /// 4、关闭连接  ::close(socket\_fd);  } |

使用sendmsg发送数据，指定了接收的对端地址、集中读的iovec数据，函数返回值为实际发送的数据长度。使用recvmsg接收数据，指定分散写的两个元素的iovec接收数组。另外，可以选择是否需要解析对端套接字地址结构。



**4.3 五组I/O函数之间的差异**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 函数 | 任何  描述符 | 仅套接字  描述符 | 单个读/写  缓冲区 | 分散/集  中读/写 | 可选标志 | 可选对  端地址 | 可选控制信息 |
| read, write | ● |  | ● |  |  |  |  |
| readv, writev | ● |  |  | ● |  |  |  |
| recv, send |  | ● | ● |  | ● |  |  |
| recvfrom, sendto |  | ● | ● |  | ● | ● |  |
| recvmsg, sendmsg |  | ● |  | ● | ● | ● | ● |

**5、排队的数据量**

有时候需在不正真读取数据的前提下，要知道套接字或文件描述符上已有多少数据排队等着读取。有三种方式获悉已排队数据量。

**5.1 使用非阻塞I/O**

如果获悉已排队数据量的目的在于避免读操作阻塞在内核中，即没有数据可读时能做其他事情。这种情况值仅需要知道是否有数据而关心具体排队数量时，可以使用非阻塞IO。详细介绍在后面章节讨论。

**5.2 使用MSG\_PEEK标志的系统调用**

既想查看数据，又想数据留在接收队列以供进程中其他业务部分稍后读取，可以使用MSG\_PEEK标志。注意，设定recv或recvfrom等系统调用的flags不能肯定对否真有数据可读，可以结合非阻塞套接字使用，也可以组合使用MSG\_DONTWAIT标志。

另外，对于流式套接字TCP和数据报套接字UDP，先后两次调用recv或recvfrom的结果有一定差异，下面分情况说明。

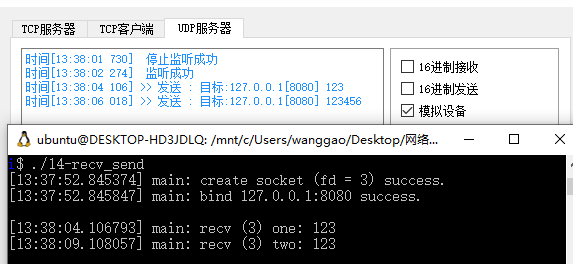
**5.2.1 获取UDP套接字上的待读取数量**

假设UDP套接字接收队列中已有一个数据，如果第一次加上该标志调用recv/recvfrom一次，接着不加该标志再调用recv/recvfrom一次，即使另有数据报在这两次调用中间加入该套接字的接收队列，这两个返回值也完全相同。

（1）以recv为例的udp服务端，创建和bindsocket之后，给出简单接收数据部分代码

|  |
| --- |
| int len ;  char buf[100]={};  while (1)  {  // 接收  len = ::recv(socket\_fd, buf, sizeof(buf), MSG\_PEEK);  if(len < 0){  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d) one: %s",len, buf);  sleep(5); // 在第二次调用recv之前客户端继续发送数据  int buf2[100]={};  len = ::recv(socket\_fd, buf2, sizeof(buf2), 0);  if(len < 0){  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d) two: %s\n",len, buf2);  break;  } |

第一次发送3个字符，服务端收到进入中间sleep(5)，第二次发送6个字符，服务端sleep结束后结果不变，仍然为3个字符。如下截图

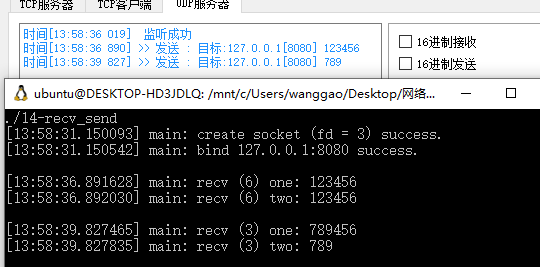


（2）根据第一次调用带MSG\_PEEK标记的结果，第二次调用读取指定数量数据

|  |
| --- |
| int len ;  char buf[100]={};  while (1)  {  // 接收  len = ::recv(socket\_fd, buf, sizeof(buf), MSG\_PEEK); //阻塞以等待数据到来  if(len < 0){  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d) one: %s",len, buf);  int buf2[100]={};  len = ::recv(socket\_fd, buf2, len, 0); // 读取已就绪的数据，长度为len  if(len < 0){  LOG("recv failed. %s", strerror(errno));  break;  }  LOG("recv (%d) two: %s\n",len, buf2);  } |

实际测试时，发现recv(socket\_fd, buf, sizeof(buf), MSG\_PEEK);会阻塞直到接收到数据，并返回队列中的数据长度。若不执行recv(socket\_fd, buf2, len, 0);读取指定长度数据，再次调用recv(socket\_fd, buf, sizeof(buf), MSG\_PEEK);会立即返回队列中的待读取数据。

先使用指定MSG\_PEEK标志的读取当前套接字上就绪的数据量，第二次不指定该标志读取队列中的就绪数据。



**5.2.2 获取TCP套接字上的待读取数量**

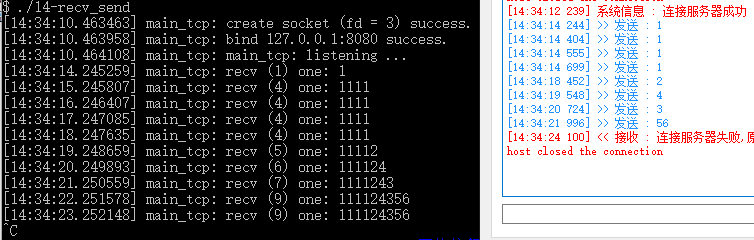
不同于udp，两次recv之间若有新的数据进来，第二次调用recv会返回更新后就绪数据的总长度。

(1) 以TCP服务端为例，给出连续调用两次使用MSG\_PEEK标志的recv

服务端代码两次调用recv之间调用一次sleep(5)。客户端发送4个字节数据，服务端收到后进入sleep(5)，客户端立即再发送2个字节，服务端第二次调用recv结果为6个字符，而不同于UDP时的6个字符。



(2)若一直没有读取队列中就绪数据，函数将返回队列中最新的就绪数据长度。例如修改服务端每个一秒调用一次带标记的recv



(3) 先调用带标志的recv一次，再调用不带标志的recv一次

第一次调用获取队列中已就绪的数据长度，第二次调用读取指定长度的数据。结果同udp。

**5.3 使用ioctl函数**

目前有些实现支持ioctl的FIONREAD命令，第三个参数是一个指向某整数的指针，用于接收内核返回当前套接字接收队列中可读的数据。

以UDP为例，用ioctl查询当前套接字接收队列的数据量，当数据量达到某个阈值进行打印输出给出。

|  |
| --- |
| // 3、 发送到服务端  while (1)  {  // ioctl查询就绪数据  int len;  if(ioctl(socket\_fd, FIONREAD, &len) < 0){  LOG("ioctl FIONREAD failed. %s", strerror(errno));  }  if(len<1){  //LOG("ioctl get bytes to read len=%d", len);  usleep(1000);  continue;  }  LOG("buf len=%d", len);  // 等到数据达到指定长度才进行读取  if(len < 10) {  sleep(1);  continue;  }  // 接收  char buf[100]={};    // 方式1  // len = ::recv(socket\_fd, buf, sizeof(buf), 0); //仅能读取上一次缓冲数据  // if(len < 0){  // LOG("recv failed. %s", strerror(errno));  // break;  // }  // LOG("recv (%d): %s",len, buf);    // 方式2  // len = ::recv(socket\_fd, buf, len, MSG\_WAITALL); //阻塞模式使用  // LOG("recv (%d): %s",len, buf); // 仅能读取上一次的数据，不能读取完全  //方式3  int readN=0;  while(readN < len){  int tmpN = ::recv(socket\_fd, buf + readN, len, 0);  readN += tmpN;  }  LOG("recv (%d): %s",len, buf);  } |

前两种方式不能一次性读完当前接收队列的中的数据，尝试在读取前进行sleep一段时间也不行（原因暂未深究）。第三种进行循环recv是可以的。

