前面介绍了有关套接字的UDP和TCP网络编程，程序中都是在套接字之后（未经特殊操作）直接使用的，此时是通过默认的套接字特性进行通信的。之前的示例都较为简单，无需操作套接字特性，但有时候必须修改。

获取和设置影响套接字的选项函数有：getsockopt和setsockopt函数，fcntl函数，ioctl函数。这里主要介绍getsockopt和setsockopt函数，另外还简单介绍fcntl函数。

**1、getsockopt和setsockopt函数**

**1.1函数原型**

#include<sys/sockte.h>

int getsockopt(intsockfdintlevelintoptnamevoid\*optvalsocklen\_t\*optlen);

int setsockopt(intsockfdintlevelintoptnameconstvoid\*optvalsocklen\_toptlen);

//返回：成功返回1，出错返回-1

sockfd是一个打开的套接字资描述符，level指定系统中解释选项的代码或为通用套接字代码，或为某个特定于协议的代码（例如IPv4、IPv6、TCP或STCP）。

optval是一个指向某个变量（\*optval）的指针，setsockopt从\*optval中取得选项待设置的新值，getsockopt则把一个已获取的选项的当前值存放到\*optval中。\*optval的长度由最后一个参数指定，对于getsockopt是一个结果指针参数，对于setsockopt是一个值参数。对于有’标志’的含义的选项，例如是否开启、是否允许等，**optval非零为是，零为否**。

level和optname共同指定了唯一的套接字选项，见下一节。

**1.2套接字选项**

套接字选项由级别和名字共同组成，包含通用、IPv4、IPv6、TCP和SCTP。这里主要介绍常用的三类，如下

|  |  |  |
| --- | --- | --- |
| level(级别) | 功能 | 头文件 |
| SOL\_SOCKET | 套接字相关通用选项的设置 | #include<sys/socket.h> |
| IPPROTO\_IP | 在IPv4设置套接字相关属性 | #include<netinet/ip.h> |
| IPPROTO\_IP | 在TCP设置套接字相关属性 | #include<netinet/tcp.h> |

套接字选项粗分为量大基本类型：一是启用或禁止某个特性的二元选项（称为标志选项），二是取得并返回可以设置或获取的特定值的选项（称为值选项）。当选项是标志选项情况下，调用getsockopt时，\*optval是一个整数，其值为0表示响应选项被禁止或关闭，不为0表示响应选项被启用；同样，调用setsockopt时，传递\*optval值非0表示开启，传递\*optval值0表示关闭或者禁用。非标志选项，optval则为指定数据类型对象。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| level | optname | get | set | 说明 | 标志 | 数据类型 |
| SOL\_SOCKET | SO\_DEBUG | ● | ● | 调试跟踪 | ● | int |
| SO\_REUSEADDR | ● | ● | 重用本地地址 | ● | Int |
| SO\_TYPE | ● |  | 套接字类型 |  | int |
| SO\_ERROR | ● |  | 待处理错误清除 |  | int |
| SO\_DONTROUTE | ● | ● | 绕过外出路由表查询 | ● | int |
| SO\_BROADCAST | ● | ● | 发送广播数据报 | ● | Int |
| SO\_SNDBUF | ● | ● | 发送缓冲区大小 |  | int |
| SO\_RCVBUF | ● | ● | 接收缓冲区大小 |  | int |
| SO\_KEEPALIVE | ● | ● | 套接字保活 | ● | int |
| SO\_OOBINLINE | ● | ● | 接收的带外数据留存 | ● | int |
| SO\_LINGER | ● | ● | 等数据发送延迟关闭 |  | linger{} |
| SO\_REUSEPORT | ● | ● | 重用本地端口 | ● | Int |
| SO\_RCVLOWAT | ● | ● | 接收缓冲区下限 |  | int |
| SO\_SNDLOWAT | ● | ● | 发送缓冲区下限 |  | int |
| SO\_RCVTIMEO | ● | ● | 接收超时 |  | timeval{} |
| SO\_SNDTIMEO | ● | ● | 发送超时 |  | timeval{} |
| IPPROTO\_IP  IPPROTO\_IPv6 | IP\_TOS | ● | ● | 服务类型和优先权 |  | int |
| IP\_TTL | ● | ● | 存活时间 |  | int |
| IP\_MTU | ● | ● |  |  |  |
| IPV6\_UNICAST\_HOPS | ● | ● |  |  |  |
| IPV6\_V6ONLY | ● | ● | 禁止兼容IPv4 | ● | int |
| IPV6\_DONTFRAG | ● | ● | 大分组丢弃不切片 | ● | int |
| IPPROTO\_TCP | TCP\_NODELAY | ● | ● | 不使用Nagle算法 | ● | int |
| TCP\_MAXSEG | ● | ● | TCP最大分节大小 |  | int |
| TCP\_KEEPIDLE | ● | ● | TCP空闲发送保活包间隔 |  | int |
| TCP\_KEEPINTVL, | ● | ● | TCP保活包无响应重发间隔 |  | int |
| TCP\_KEEPCNT | ● | ● | TCP保活包无响应重发次数 |  | int |

**2、套接字选项示例**

介绍套接字可选项的设置、读取的示例，给出常用的选项说明和示例。

在介绍示例前，先提一点套接字状态的问题，针对套接字状态，在时序上需要考虑什么时候才能设置或获取选项。对于SO\_DEBUG、SO\_SONTROUTE、SO\_KEEPALIVE、SO\_OOBINLINE、SO\_RECVBUF、SO\_SNDBUF、SO\_RCVLOWAT、SO\_SNDLOWAT、TCP\_MAXMSG和TCP\_NODELAY是由TCP从已连接套接字从监听套接字继承来，因为accept一直要等到TCP层完成三路握手之后才会返回已连接套接字。也就是说，若想在三次握手完成时确保这些套接字选项中的某一个是给已连接套接字设置的，那么必须先要给监听套接字设置该选项。

**2.1 SO\_KEEPALIVE**

**2.1.1 选项说明**

对于面向连接的TCP连接，断开有两种情况

1. 连接正常关闭，调用close()会经历四次握手，send()、recv()会立即返回错误
2. 连接的对端异常关闭，例如网络断开、设备断电，这种情况对方是检测不到连接异常

解决这种异常断开的检测机制有两种

1. 在应用层定时发送心跳包判断连接是否正常，通用灵活，但改变了现有协议
2. 使用TCP协议自带的keepalive机制，使用简单，降低应用层复杂度。

keepalive原理：TCP内嵌有心跳包，以服务端为例，当server检测到超过一定时间(tcp\_keepalive\_time 7200s，即2个小时)没有数据传输，会向client发送一个keepalive packet，此时client会有三种反应：

1. client端连接正常，返回一个ACK，server接收到ACK后重置计时，两个小时之后再发送保活包（如果2个小时内有数据传输，则在改时间上继续后延2小时）。
2. client客户端异常关闭或网络断开，client无响应，server收不到ACK，再一定时间内（tcp\_keepalive\_intvl 75 即75秒）后重发保活包，并重发一定次数（tcp\_keepalive\_probes 9即9次），仍然得不到响应则返回超时。
3. 客户端崩溃后已经重启，server收到的响应是一个RST复位，server端断开连接。

**2.1.2 代码示例**

我们可以根据自己的需求来修改这三个参数的系统默认值，下面我们来通过一个简单的回响服务器代码来实现保活机制的逻辑实现。

|  |
| --- |
| #include <stdio.h>  #include <sys/socket.h>  #include <arpa/inet.h> // sockaddr\_in, inet\_addr  #include <unistd.h> // close  #include <cstring> // memset, bezro  #include <errno.h>  #include <netinet/tcp.h> // tcp options  #include <thread>  #include <sys/time.h>  #define LOG(fmt, arg...) \  do{ \  struct timeval tv; \  gettimeofday(&tv, NULL); \  printf("[%ld.%03ld] %s: " fmt, \  tv.tv\_sec, tv.tv\_usec / 1000, \_\_func\_\_, ##arg); \  } while (0)  //const char \*SRV\_ADDR = "127.0.0.1";  const char \*SRV\_ADDR = "192.168.0.101";  const int SRV\_PORT = 8080;  static  void client\_service(int sock\_id, sockaddr\_storage clientaddr, socklen\_t sock\_len)  {  if(sock\_len == 0){  sock\_len = sizeof(clientaddr);  getpeername(sock\_id, (sockaddr\*)&clientaddr, &sock\_len);  }  char client\_ip[INET6\_ADDRSTRLEN]; // 足够存IPv4和IPv6地址  int port;  if(sock\_len == sizeof(sockaddr\_in)){  sockaddr\_in \*cliAddr = (sockaddr\_in \*)&clientaddr;  //char\* client\_ip = inet\_ntoa(cliAddr->sin\_addr);  inet\_ntop(AF\_INET, &cliAddr->sin\_addr, client\_ip, sock\_len);  port = ntohs(cliAddr->sin\_port);  LOG("new socket id = %d, clent %s:%d\n", sock\_id, client\_ip, port);  }  else if(sock\_len == sizeof(sockaddr\_in6)){  sockaddr\_in6 \*cliAddr = (sockaddr\_in6 \*)&clientaddr;  inet\_ntop(AF\_INET6, &cliAddr->sin6\_addr, client\_ip, sock\_len);  port = ntohs(cliAddr->sin6\_port);  LOG("new socket id = %d, clent %s:%d\n", sock\_id, client\_ip, port);  }    // 给已连接套接字设置SO\_KEEPALIVE选项  int keepAlive = 1; // 开启keepalive属性. 缺省值: 0(关闭)  int ret = setsockopt(sock\_id, SOL\_SOCKET, SO\_KEEPALIVE, &keepAlive, sizeof(keepAlive));  if(ret < 0) LOG("set SOL\_SOCKET, SO\_KEEPALIVE failed.\n");    int len;  char buf[1024];  while (1)  {  len = read(sock\_id, buf, sizeof(buf));  if(len < 0){  LOG("recv failed. %s\n", strerror(errno));  break;  }else if(len == 0){ // tcp 读取长度为0,表示对端关闭连接  LOG("client socket %d, %s:%d disconnet. recv len = 0.\n", sock\_id, client\_ip, port);  break;  }  buf[len] = 0; //避免接收数据较上一次短，导致输出显示错误  if (strcmp(buf, "exit\n") == 0)  break;  LOG("recv from socket %d, %s:%d (%d): %s\n", sock\_id, client\_ip, port, len, buf );  // // echo  // len = ::write(sock\_id, buf, strlen(buf));  // if(len < 0){  // LOG("send failed. %s\n", strerror(errno));  // break;  // }  }  ::close(sock\_id);  }  int main()  {  /// 1、创建socket  int socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0); // tcp  //int socket\_fd = ::socket(AF\_INET,SOCK\_DGRAM, 0); // udp  if (socket\_fd == -1){  LOG("create socket failed. %s\n", strerror(errno));  return 1;  }  else{  LOG("create socket (fd = %d) success.\n", socket\_fd);  }  /////// keepalive test  // **给监听套接字设置SO\_KEEPALIVE选项**  int keepAlive = 1; // 开启keepalive属性. 缺省值: 0(关闭)  int ret = setsockopt(socket\_fd, SOL\_SOCKET, SO\_KEEPALIVE, &keepAlive, sizeof(keepAlive));  if(ret < 0) LOG("set SOL\_SOCKET, SO\_KEEPALIVE failed.\n");    int keepIdle = 10; // 如果在10秒内没有任何数据交互,则进行探测..缺省值:7200(s)  ret = setsockopt(socket\_fd, IPPROTO\_TCP, TCP\_KEEPIDLE, &keepIdle, sizeof(keepIdle));  if(ret < 0) LOG("set IPPROTO\_TCP, SO\_KEEPALIVE failed.\n");    int keepInterval = 5; // 探测时发探测包的时间间隔为5秒. 缺省值:75(s)  ret = setsockopt(socket\_fd, IPPROTO\_TCP, TCP\_KEEPINTVL, &keepInterval, sizeof(keepInterval));  if(ret < 0) LOG("set IPPROTO\_TCP, SO\_KEEPALIVE failed.\n");    int keepCount = 3; // 探测重试的次数. 全部超时则认定连接失效..缺省值:9(次)  ret = setsockopt(socket\_fd, IPPROTO\_TCP, TCP\_KEEPCNT, &keepCount, sizeof(keepCount));  if(ret < 0) LOG("set IPPROTO\_TCP, TCP\_KEEPCNT failed.\n");  /// 2、绑定到本地端口  sockaddr\_in servaddr;  servaddr.sin\_family = AF\_INET;  inet\_pton(AF\_INET, SRV\_ADDR, &servaddr.sin\_addr);  servaddr.sin\_port = htons(SRV\_PORT);  //int ret = ::bind(socket\_fd, (const sockaddr \*)&servaddr, sizeof(servaddr));  ret = ::bind(socket\_fd, (const sockaddr \*)&servaddr, sizeof(servaddr));  if (ret == -1){  LOG("bind %s:%d failed. %s\n", SRV\_ADDR, SRV\_PORT, strerror(errno));  return 1;  }else{  LOG("bind %s:%d success.\n", SRV\_ADDR, SRV\_PORT);  }  /// 3、监听  ret = ::listen(socket\_fd, 5);  if(ret < 0){  LOG("listen failed. %s\n", strerror(errno));  return 1;  }else{  LOG("listening ...\n");  }  while(true)  {  // 等待连接  sockaddr\_storage clientaddr; // 可以满足IPv4和IPv6的内存需求  // 必须给sock\_len初始长度, accept会根据实际情况重新赋值， 这里 128 -> 16  socklen\_t sock\_len = sizeof(clientaddr);  int sock\_id = ::accept(socket\_fd, (sockaddr\*)&clientaddr,&sock\_len);  if(sock\_id < 0){  LOG("accept error. %s\n", strerror(errno));  return 1;  }  std::thread(client\_service,sock\_id,clientaddr,sock\_len).detach();  }  /// 4、关闭连接  ::close(socket\_fd);  } |

注意代码中套接字选项如下设置：

**设置监听套接字SO\_KEEPALIVE选项**，

设置TCP\_KEEPIDLE保活空闲时间间隔10秒，

设置TCP\_KEEPINTVL保活包无响应时重发间隔时间5秒

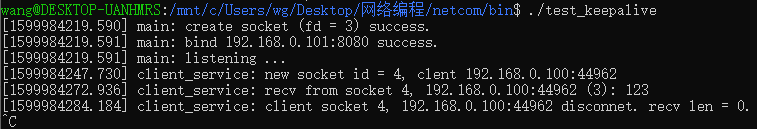
设置TCP\_KEEPINTVL保活包无响应时重复发送次数3次

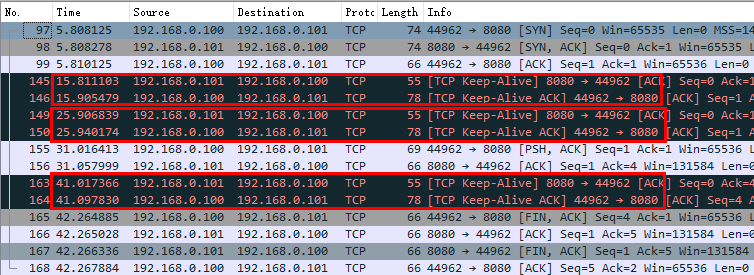
**设置客户端已连接套接字SO\_KEEPALIVE选项**，

代码演示

1. 客户端保持连接

三次握手建立连接之后，等待10秒服务端发送一个保活包，客户端立即回复保活响应。之后10秒中内没有任何TCP通信，服务端和客户端之间又进行一次保活包通信。之后5秒，客户端发送了一次消息给服务端。服务端从接受消息后开始计时，10秒之后再次发送保活包给客户端。

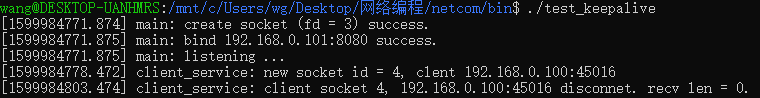


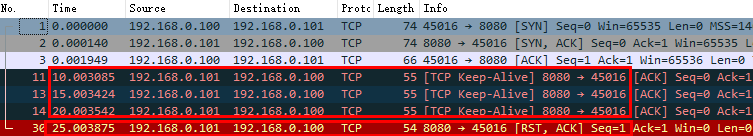


1. 客户端连接异常关闭（模拟断开网络）

服务端端在第10秒开始发送保活包，客户端无响应，间隔5秒，连续发送了3次保活包。

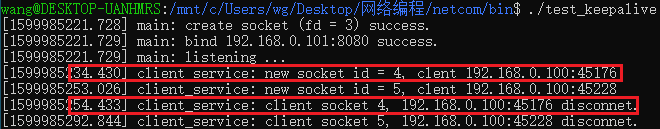
之后服务端发送RST消息，并断开了连接。

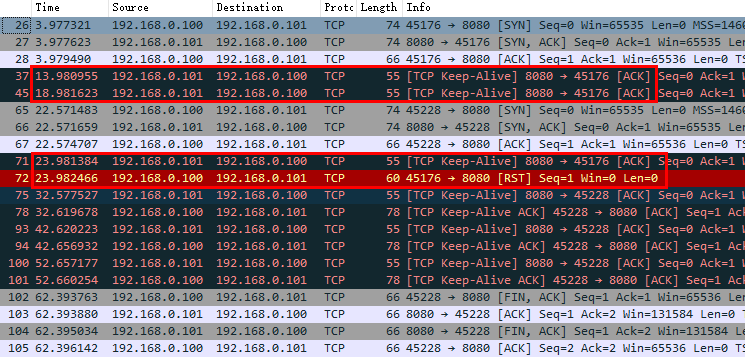




1. 客户端连接异常关闭（模拟断开网络）后重新连接

客户端重连后，服务端在第三个保活包之后收到了RST消息，断开连接。





**2.2 SO\_REUSEADDR**

该选项用于地址复用，会影响套接字关闭的TIME\_WAIT状态。在前文TCP套接字编程基础概念TCP状态转换图，主动发起断开连接的一端，在收到被动关闭一端的FIN包后会有一个ACK确认，为接收ACK确认会进入TIME\_WAIT状态以确认对端接收到ACK确认包，如果收到则关闭，否则重发。这个TIME\_WAIT状态只发生在主动关闭连接的一端，被动断开一方是没有该状态的。

作为服务器，通常都是客户端主动断开连接，但是有时候因为异常需要重启服务端程序，服务端此时为主动断开的一方，系统对于这个套接字断开有TIME\_WAIT状态，重新运行绑定到相同地址和端口就是提示已被使用的错误。这时，就需要等到TIME\_WAIT状态结束也就是默认的2分钟之后，服务端程序才能再次成功绑定到该端口。因此，对于服务端程序，通常需要在调用bind之前设置SO\_REUSEADDR选项。

|  |
| --- |
| #include <sys/socket.h>  int enableReuseAddr(int sock)  {  int optval = 1;  if (setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval)) == -1)  return -1;  return 0;  } |

**2.3 SO\_LINGER**

指定close()函数对面向连接的协议（例如TCP和SCTP，不是UDP）进行延迟关闭操作。默认操作是，在调用close()立即返回，若发送缓冲区还有数据，系统将试着把这些数据发送给对端。

SO\_LINGER套接字选项可以改变这个默认选项。

**2.3.1 struct lingers说明**

对于结构体参数的设置，

#include<sys/sockte.h>

struct linger{

int l\_onoff; // 开启(非0), 关闭(0)

inr l\_linger; // 滞留时间，设置大于0起作用， 秒

}

参数设置有以下情况，列举如下：

1. 参数l\_onoff = 0时，关闭该选项，l\_linger值被忽略，默认设置生效，close()立即返回。
2. 参数l\_onoff = 1非零时，参数l\_linger=0超时时间为0，调用close()立即返回，丢弃发送缓冲区的所有数据，并发送RST信号给对方（非正常的四次握手关闭，正常是FIN/ACK/FIN/ACK），不用经过2MSL的TIME\_WAIT状态，对方会读取到socket重置的错误。
3. 参数l\_onoff = 1非零时，参数l\_linger取值大于零。当调用close()时，系统将阻塞延迟关闭l\_linger指定时间。如果套接字发送缓冲区还有数据，那么进程将被休眠以下任意情况发生：

1) 发送缓冲区数据已被发送且收到对方确认消息（但不保证对方应用读取完这些数据）

2) 延迟时间消耗完，发送缓冲区数据丢弃，并发送RST消息

这两种情况下，如果socket被设置成非阻塞，程序不会等待close()返回，立即丢弃发送缓冲区的所有数据。因此，需要检查close()的返回值，当数据发送完、超时时间未用完时，clsoe()返回EWOULDBLOCK的错误。

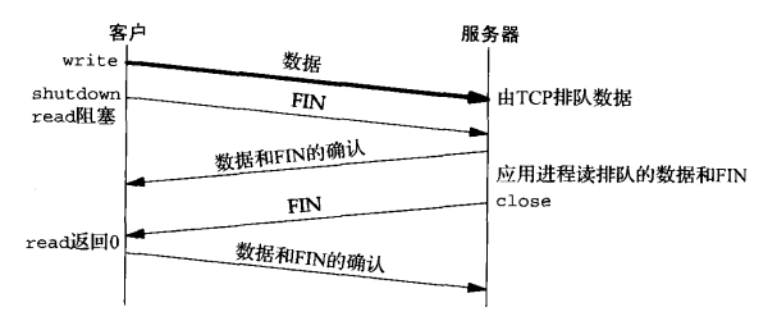
|  |
| --- |
| #include <sys/socket.h>  int enableLinger(int sock)  {  struct linger so\_linger;  so\_linger.l\_onoff = 1; //开启(非0) 关闭(0)  so\_linger.l\_linger = 5; //滞留时间，设置为大于0时才起作用  if (setsockopt(sock, SOL\_SOCKET, SO\_LINGER, &so\_linger, sizeof(so\_linger)) == -1) {  return 0;  }  return 1;  } |

**2.3.1 套接字关闭（主要针对TCP套接字断开）**

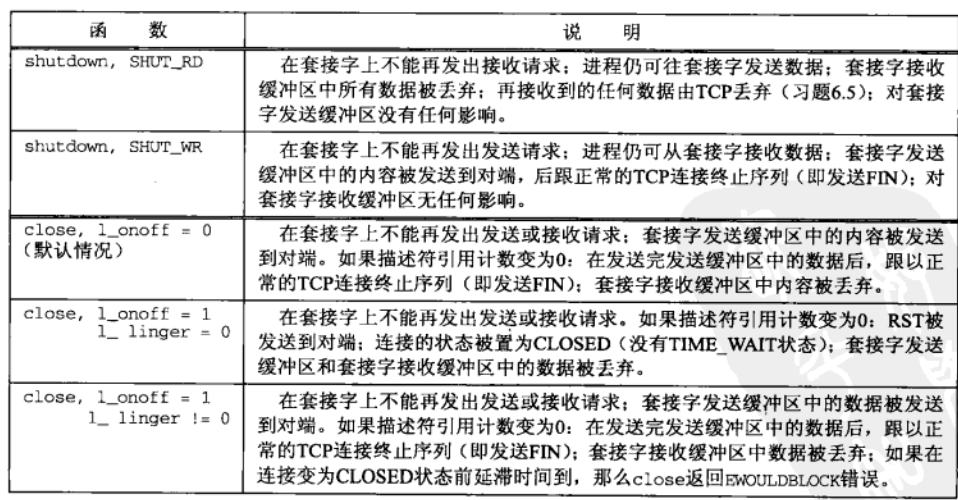
套接字是默认没有设置SO\_LINGER选项，调用close()时，将FIN写入发送缓冲区，立即返回不管数据是否发出。另外在多进程下，close()只是使套接字引用计数减1，该进程不能继续使用该套接字进行读写操作，其他进程正常使用；当引用计数减为0时，系统才释放该套接字。

在设置SO\_LINGER选项后，close()成功返回只是告诉我们先前已发送的数据（包括FIN）已被对端TCP确认，但不能告诉我们对端应用程序是否已经读取数据。如果不设置该选项，我们连对端TCP是否确认了数据都不知道。

为了让发送者知道对端已经读取其数据的一个方法时，改为调用shutdown()函数（其第二个参数是SHUT\_WR）而不是调用close()。此时发送者在shutdown后调用read函数阻塞，等待对端处理数据调用close()关闭数据发送FIN，read返回0。（注意，shutdown会同时影响所有进程的套接字读、写功能）



当关闭套接字连接时，根据所调用的函数（close或shutdown）以及是否设置了SO\_LINGER选项，列出其对TCP套接字的影响



**2.4 SO\_RCVBUF & SO\_SNDBUF**

SO\_RCVBUF、SO\_SNDBUF用于设置或获取套接字输入、输出缓冲区的大小，这个缓冲区是创建套接字时系统创建并分配好的。

对于TCP套接字，这两个选项的值限定了TCP通告对端的窗口的大小。TCP具有流量控制功能，正常使用时因为不允许发送超过通告窗口大小数据，缓冲区不会溢出，如果无视通告窗口大小，TCP会丢弃他们。

对于UDP套接字，是没有流量控制的，超过缓冲区大小的数据就会被丢弃，另外较快的发送端会淹没较慢的接收端，导致接收端数据报报本机丢弃。

**2.5 SO\_RCVLOWAT & SO\_SNDLOWAT**

设置接收、发送缓冲区的可读、可写的数据量下限标记，主要由IO复用系统如select函数使用。

SO\_RCVLOWAT是让select函数返回“可读”时套接字接收缓冲区中所需的数据量。对于TCP、UDP和SCTP，其默认值都是1。

SO\_ SNDLOWAT是让select函数返回“可写”时套接字接收缓冲区中所需的数据量。对于TCP、其默认值通常是2048。UDP也使用发送区下限标记，但是UDP套接字缓冲区中可用空间字节数保持不变，主要发送缓冲区大小大于接收缓冲区的下限标记，UDP套接字就总是可写的。

**2.6 SO\_RCVTIMEO & SO\_SNDTIMEO**

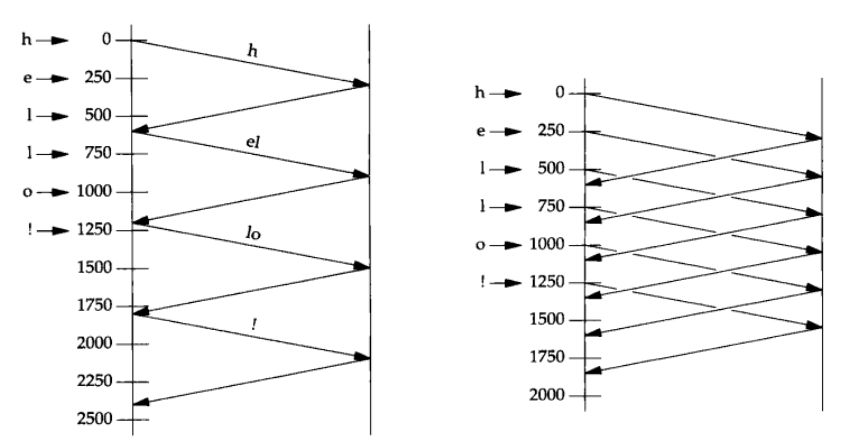
主要用于设置套接字调用读取、发送数据函数的超时时间。在平时使用中设置TCP套接字的超时选项不多，一般用在UDP中，特别是用于“一问一答”的UDP通信系统中。UDP套接字发送数据不用建立连接，所以数据发送出去并不知道对方是否收到，需要对方回复确认，否则超时重发。

接收超时影响5个输入函数：read、readv、recv、recvfrom和recvmsg。发送超时会影响5个输出函数：write、writev、send、sendto和sendmsg。将在后续的高级IO函数章节中继续讨论。

**2.7 TCP\_NODELAY**

TCP\_NODELAY 是否禁用Nagle算法，默认情况下该算法是启动的。

为了防止网络数据包过多而发生网络过载，Nagle算法在1984年诞生了，应用于TCP层，非常简单，是否使用差异看下图

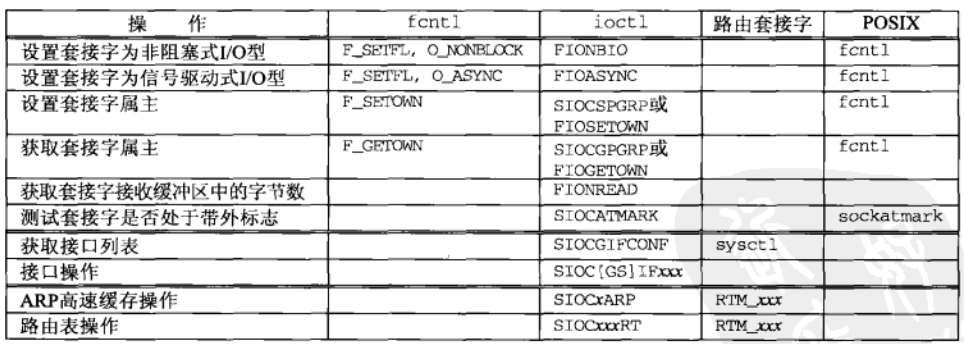


连续发送”hello”的6个字符，间隔250ms。TCP套接字默认使用Nagle算法进行数据交换，因此最大限度的进行缓冲，只有收到上个包的确认ACK后才会传输下个数据包。在不使用Nagle算法的情况下发送数据，不需要等待收到上个数据包的ACK确认，接着就开始发送下面的数据，这种情况下会对网络流量产生极大的负面影响，即使只传输1个字节的数据，其头信息都有可能是几十个字节。通常解决方式不调用两次write，是使用writev合并成一个分节发送，或者合并缓冲区调用一次write发送。因此为了提高网络传输效率，必须使用Nagle算法。但是对于大文件的网络传输，有时不使用Nagle算法能提高传输速度。

**3、fcntl函数**

与代表”file control”（文件控制）的名字相符，fcntl函数用于各种文件描述符控制的操作。

在说明该函数和对套接字影响前，给出fcntl、ioctl和路由套接字执行的不同操作



我们关注套接字的操作，前四个操作的执行方法不只一种，POSIX规定fcntl方法是首选，另外还推荐使用sockatmark函数测试是否处于带外标志的首选方法。Fcntl函数提供了与网络编程相关的如下特性：

* 非阻塞I/O。通过使用F\_SETFL参数设置O\_NONBLOCK文件状态标志，把一个套接字设置成非阻塞。
* 信号驱动I/O。通过使用F\_SETFL参数设置O\_ASYNV文件状态标志，把一个套接字设置成一旦其状态发生变化，内核就产生一个SIGIO信息。
* F\_SETOWN参数允许指定用于接收SIGIO、SIGURG信号的套接字属主(进程ID或进程组ID)。SIGIO是信号驱动IO套接字产生，SIGURG是带外数据到达产生。F\_GWTOWN用于获取套接字当前属主。

这里简单介绍fcntl开启和关闭阻塞IO的代码

#incude <fcntl.h>

int fcntl(int fd, int cmd, .../\*int arg\*/);

// 返回：若成功取决于cmd，出错则返回-1

1. 开启非阻塞I/O的典型代码

注意，设置文件的状态标志唯一正确方法是：先获取文件描述符的状态，与新标志逻辑或后再设置标志。否则会修改文件其他的状态标记。

|  |
| --- |
| int flags;  if(flags = fcntl(fd, F\_GETFL, 0) < 0){  perror("");  return -1;  }  **flags |= O\_NONBLOCK; // 与新标志逻辑或**  if(fcntl(fd, F\_SETFL, flags) < 0){  perror("");  return -1;  } |

1. 关闭非阻塞I/O的典型代码

注意，关闭文件的状态标志唯一正确方法是：先获取文件描述符的状态，与新标志的取反值与后再设置标志。否则会修改文件其他的状态标记。

|  |
| --- |
| int flags;  if(flags = fcntl(fd, F\_GETFL, 0) < 0){  perror("");  return -1;  }  **flags &= ~O\_NONBLOCK; // 与新标志的取反值逻辑与**  if(fcntl(fd, F\_SETFL, flags) < 0){  perror("");  return -1;  } |