未语愁眸

setsockopt()用法（参数详细说明）

标签： socket tcp 服务器 freebsd struct linux

2008-05-27 12:37 121391人阅读 评论(17)

分类： 程序相关（125）


```
int setsockopt(
SOCKET s,
int level,
int optname,
const char* optval,
int optlen
);
```

s(套接字): 指向一个打开的套接口描述字

level:(级别)： 指定选项代码的类型。

SOL_SOCKET: 基本套接口

IPPROTO_IP: IPv4套接口

程序相关 (126)
网络方面 (9)
资源链接 (3)
邮箱专家 (1)
随感 (5)
Linux知识 (7)
opencv (3)
RPC框架 (8)
大数据 (2)
业余爱好 (2)
架构 (1)
智能硬件 (2)

文章存档

2017年06月 (1)
2017年05月 (2)
2017年02月 (2)
2017年01月 (3)
2016年12月 (1)

展开

阅读排行

setsockopt()用法（参数详细说… (121359)
基于OpenCV读取摄像头进行人… (35322)
openssl之aes加密（源码分析 A… (30890)
pkcs#5和pkcs#7填充的区别 (14867)
使用libcurl POST数据和上传文… (10043)
Unicode与UTF-8互转(C语言实… (9746)
C语言基础-结构体和联合体 (9739)
openssl编译出错解决 (9723)
关于进程的拒绝访问 (9618)

IPPROTO_IPV6: IPv6套接口

IPPROTO_TCP: TCP套接口

optname(选项名)： 选项名称

optval(选项值): 是一个指向变量的指针 类型：整形，套接口结构， 其他结构类型:linger{}, timeval{ }

optlen(选项长度) ： optval 的大小

返回值：标志打开或关闭某个特征的二进制选项

[/code:1:59df4ce128]

==========

SOL_SOCKET

SO_BROADCAST 允许发送广播数据 int

适用於 UDP socket。其意义是允许 UDP socket 「广播」（broadcast）讯息到网路上。

SO_DEBUG 允许调试 int

SO_DONTROUTE 不查找路由 int

SO_ERROR 获得套接字错误 int

SO_KEEPALIVE 保持连接 int

检 测对方主机是否崩溃，避免（服务器）永远阻塞于TCP连接的输入。 设置该选项后，如果2小时内在此套接口的任一方向都没有数据交换，TCP就自动给对方 发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的TCP分节.它会导致以下三种情况： 对方接收一切正常：以期望的 ACK响应。2小时后，TCP将发出另一个探测分节。 对方已崩溃且已重新启动：以RST响应。套接口的待处理错误被置为ECONNRESET，套接 口本身则被关闭。 对方无任何响应：源自berkeley的TCP发送另外8个探测分节，相隔75秒一个，试图得到 一个响应。在发出第一个探测分节11分钟15秒后若仍无响应就放弃。套接口的待处理错 误被置为ETIMEOUT，套接口本身则被关闭。如ICMP错误是“host unreachable (主机不 可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

SO_DONTLINGER 若为真，则SO_LINGER选项被禁止。

SO_LINGER 延迟关闭连接 struct linger

libcurl使用认证证书 https认证 (7514)

评论排行

基于OpenCV读取摄像头进行人.. (22)
setsockopt()用法（参数详细说… (17)
libcurl使用认证证书 https认证 (8)
C语言基础-结构体和联合体 (6)
C++ 0x新特性:详细讲解lambda… (4)
6年软件开发经验总结 (4)
openssl编译出错解决 (3)
Unicode与UTF-8互转(C语言实… (3)
我的第一个发布到网上的程序… (3)
用openssl跟Gmail的smtp对话（.. (2)

最新评论

基于OpenCV读取摄像头进行人脸检测和人..
lb549049865 ：VideoMFC.exe 中的 0x00db51cb 处有未经处理的异常: 0xC0000005: ...

使用openssl库实现RSA、AES数据加密
a1adsda1a1a1a2 ：想问下楼主,如果明文过长，用openssl如何分段加密？117个字节，但是openssl是传字符串的...

基于OpenCV读取摄像头进行人脸检测和人..
baidu_38133986 ：很棒啊

基于OpenCV读取摄像头进行人脸检测和人..
qq_37170705 ：这个帖子比较老了，新的OpenCV3.0以上就不用这个了。有个class类CascadeClassi...

基于OpenCV读取摄像头进行人脸检测和人..
defuchocolate520 ：#include 请问楼主这个opencv2下没有这个目录文件怎么办?

openssl编译出错解决
hf8818147 ：赞同楼主，是正解。enable-shared 不要加2个减号

基于OpenCV读取摄像头进行人脸检测和人..
qq_26526863 ：楼主，为何我运行的结果显示都是predict=0啊

基于OpenCV读取摄像头进行人脸检测和人..

上面这两个选项影响close行为

选项 间隔 关闭方式 等待关闭与否

SO_DONTLINGER 不关心 优雅 否

SO_LINGER 零 强制 否

SO_LINGER 非零 优雅 是

若 设置了SO_LINGER（亦即linger结构中的l_onoff域设为非零，参见2.4，4.1.7和4.1.21各节），并设置了零超时间隔，则 closesocket()不被阻塞立即执行，不论是否有排队数据未发送或未被确认。这种关闭方式称为“强制”或“失效”关闭，因为套接口的虚电路立即被 复位，且丢失了未发送的数据。在远端的recv()调用将以WSAECONNRESET出错。

若设置了SO_LINGER并确定了非零的超时间 隔，则closesocket()调用阻塞进程，直到所剩数据发送完毕或超时。这种关闭称为“仍 请

注意如果套接口置为非阻塞且 SO_LINGER设为非零超时，则closesocket()调用将以WSAEWOULDBLOCK错误返回。

若在一个流类套接口上设置了 SO_DONTLINGER（也就是说将linger结构的l_onoff域设为零；参见2.4，4.1.7，4.1.21节），则 closesocket()调用立即返回。但是，如果可能，排队的数据将在套接口关闭前发送。请注意，在这种情况下WINDOWS套接口实现将在一段不确 定的时间内保留套接口以及其他资源，这对于想用所以套接口的应用程序来说有一定影响。

SO_OOBINLINE 带外数据放入正常数据流,在普通数据流中接收带外数据 int

SO_RCVBUF 接收缓冲区大小 int

设置接收缓冲区的保留大小

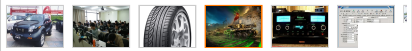
与 SO_MAX_MSG_SIZE 或TCP滑动窗口无关，如果一般发送的包很大很频繁，那么使用这个选项

SO_SNDBUF 发送缓冲区大小 int

设置发送缓冲区的保留大小

与 SO_MAX_MSG_SIZE 或TCP滑动窗口无关，如果一般发送的包很大很频繁，那么使用这个选项

每 个套接口都有一个发送缓冲区和一个接收缓冲区。 接收缓冲区被TCP和UDP用来将接收到的数据一直保存到由应用进程来读。 TCP：TCP通告另一端的窗口大小。 TCP套接口接收缓冲区不可能溢出，因为对方不允许发出超过所通告窗口大小的数据。 这就是TCP的流量控制，如果对方无视窗口大小而发出了超过宙口大小的数据，则接 收方TCP将丢弃它。 UDP：当接收到的数据报装不进套接口接收缓冲区时，此数据报就被丢弃。UDP是没有 流量控制的；快的发送者可以很容易地就淹没慢的接收者，导致接收方的UDP丢弃数据报。



SO_RCVLOWAT 接收缓冲区下限 int

SO_SNDLOWAT 发送缓冲区下限 int

每 个套接口都有一个接收低潮限度和一个发送低潮限度。它们是函数selectt使用的， 接收低潮限度是让select返回“可读”而在套接口接收缓冲区中必须有的数据总量。 ——对于一个TCP或UDP套接口，此值缺省为1。发送低潮限度是让select返回“可写” 而在套接口发送缓冲区中必须有的可用空间。对于TCP套接口，此值常缺省为2048。 对于UDP使用低潮限度， 由于其发送缓冲区中可用空间的字节数是从不变化的，只要 UDP套接口发送缓冲区大小大于套接口的低潮限度，这样的UDP套接口就总是可写的。 UDP没有发送缓冲区，只有发送缓冲区的大小。

SO_RCVTIMEO 接收超时 struct timeval

SO_SNDTIMEO 发送超时 struct timeval

SO_REUSERADDR 允许重用本地地址和端口 int

充许绑定已被使用的地址（或端口号），可以参考bind的man

SO_EXCLUSIVEADDRUSE

独占模式使用端口,就是不充许和其它程序使用SO_REUSEADDR共享的使用某一端口。

在确定多重绑定使用谁的时候，根据一条原则是谁的指定最明确则将包递交给谁，而且没有权限之分，也就是说低级权限的用户是可以重绑定在高级权限如服务启动的端口上的,这是非常重大的一个安全隐患,

如果不想让自己程序被监听，那么使用这个选项

SO_TYPE 获得套接字类型 int

SO_BSDCOMPAT 与BSD系统兼容 int

==========

IPPROTO_IP

IP_HDRINCL 在数据包中包含IP首部 int

这个选项常用于黑客技术中，隐藏自己的IP地址

IP_OPTINOS IP首部选项 int

IP_TOS 服务类型

IP_TTL 生存时间 int

以下IPV4选项用于组播

IPv4 选项 数据类型 描 述

IP_ADD_MEMBERSHIP struct ip_mreq 加入到组播组中

IP_ROP_MEMBERSHIP struct ip_mreq 从组播组中退出

IP_MULTICAST_IF struct ip_mreq 指定提交组播报文的接口

IP_MULTICAST_TTL u_char 指定提交组播报文的TTL

IP_MULTICAST_LOOP u_char 使组播报文环路有效或无效

在头文件中定义了ip_mreq结构：

[code:1:63724de67f]

struct ip_mreq {

struct in_addr imr_multiaddr; /* IP multicast address of group */

struct in_addr imr_interface; /* local IP address of interface */

};

[/code:1:63724de67f]

若进程要加入到一个组播组中，用soket的setsockopt()函数发送该选项。该选项类型是ip_mreq结构，它的第一个字段imr_multiaddr指定了组播组的地址，第二个字段imr_interface指定了接口的IPv4地址。

IP_DROP_MEMBERSHIP

该选项用来从某个组播组中退出。数据结构ip_mreq的使用方法与上面相同。

IP_MULTICAST_IF

该选项可以修改网络接口，在结构ip_mreq中定义新的接口。

IP_MULTICAST_TTL

设置组播报文的数据包的TTL（生存时间）。默认值是1，表示数据包只能在本地的子网中传送。

IP_MULTICAST_LOOP

组播组中的成员自己也会收到它向本组发送的报文。这个选项用于选择是否激活这种状态。

无双 回复于：2003-05-08 21:21:52

IPPRO_TCP

TCP_MAXSEG TCP最大数据段的大小 int

获 取或设置TCP连接的最大分节大小(MSS)。返回值是我们的TCP发送给另一端的最大 数据量，它常常就是由另一端用SYN分节通告的MSS，除非我们的TCP选择使用一个比 对方通告的MSS小些的值。如果此值在套接口连接之前取得，则返回值为未从另·—端 收到Mss选项的情况下所用的缺省值。小于此返回值的信可能真正用在连接上，因为譬 如说使用时间戳选项的话，它在每个分节上占用12字节的TCP选项容 :P 将 发送的每个分节的最大数据量也可在连接存活期内改变，但前提是TCP要支持路径MTU 发现功能。如果到对方的路径改变了， 周 整。

TCP_NODELAY 不使用Nagle算法 int

指定TCP开始发送保持存活探测分节前以秒为单位的连接空闲时间。缺省值至少必须为7200秒，即2小时。此选项仅在SO_KEPALIVEE套接口选项打开时才有效。

TCP_NODELAY 和 TCP_CORK，

这 两个选项都对网络连接的行为具有重要的作用。许多UNIX系统都实现了TCP_NODELAY选项，但是，TCP_CORK则是**Linux**系统所独有的而且相对较新；它首先在内核版本2.4上得以实现。此外，其他UNIX系统版本也有功能类似的选项，值得注意的是，在某种由BSD派生的系统上的TCP_NOPUSH选项其实就是TCP_CORK的一部分具体实现。

TCP_NODELAY和TCP_CORK基本上控制了包的 “Nagle化”，Nagle化在这里的含义是采用Nagle算法把较小的包组装为更大的帧。John Nagle是Nagle算法的发明人，后者就是用他的名字来命名的，他在1984年首次用这种方法来尝试解决福特汽车公司的网络拥塞问题（欲了解详情请参 看IETF RFC 896）。他解决的问题就是所谓的silly window syndrome ，中文称“愚蠢窗口症候群”，具体含义是，因为普遍终端应用程序每产生一次击键操作就会发送一个包，而典型情况下一个包会拥有一个字节的数据载荷以及40 个字节长的包头，于是产生4000%的过载，很轻易地就能令网络发生拥塞,。 Nagle化后来成了一种标准并且立即在因特网上得以实现。它现在已经成为缺省配置了，但在我们看来，有些场合下把这一选项关掉也是合乎需要的。

现 在让我们假设某个应用程序发出了一个请求，希望发送小块数据。我们可以选择立即发送数据或者等待产生更多的数据然后再一次发送两种策略。如果我们马上发送 数据，那么交互性的以及客户/服务器型的应用程序将极大地受益。例如，当我们正在发送一个较短的请求并且等候较大的响应时，相关过载与传输的数据总量相比 就会比较低，而且，如果请求立即发出那么响应时间也会快一些。以上操作可以通过设置套接字的TCP_NODELAY选项来完成，这样就禁用了Nagle算 法。

另外一种情况则需要我们等到数据量达到最大时才通过网络一次发送全部数据，这种数据传输方式有益于大量数据的通信性能，典型的应用就是文件服 务器。应用 Nagle算法在这种情况下就会产生问题。但是，如果你正在发送大量数据，你可以设置TCP_CORK选项禁用Nagle化，其方式正好同 TCP_NODELAY相反（TCP_CORK 和 TCP_NODELAY 是互相排斥的）。下面就让我们仔细分析下其工作原理。

假设应用程序 使用sendfile()函数来转移大量数据。应用协议通常要求发送某些信息来预先解释数据，这些信息其实就是报头内容。典型情况下报头很小，而且套接字 上设置了TCP_NODELAY。有报头的包将被立即传输，在某些情况下（取决于内部的包计数器），因为这个 方收到后需要请求对方确认。这 样，大量数据的传输就会被推迟而且产生了不必要的网络流量交换。

但是，如果我们在套接字上设置了TCP_CORK（可以比喻为在管道上插入“塞子”）选项，具有报头的包就会填补大量的数据，所有的数据都根据大小自动地通过包传输出去。当数据传输完成时，最好取消TCP_CORK 选项设置给连接“拔去塞子”以便任一部分的帧都能发送出去。这同“塞住”网络连接同等重要。

总而言之，如果你肯定能一起发送多个数据集合（例如HTTP响应的头和正文），那么我们建议你设置TCP_CORK选项，这样在这些数据之间不存在延迟。能极大地有益于WWW、FTP以及文件服务器的性能，同时也简化了你的工作。示例代码如下：

```
intfd, on = 1;
…
/* 此处是创建套接字等操作，出于篇幅的考虑省略*/
…
setsockopt (fd, SOL_TCP, TCP_CORK, &on, sizeof (on)); /* cork */
write (fd, …);
fprintf (fd, …);
sendfile (fd, …);
write (fd, …);
```

sendfile (fd, …);

…

on = 0;

setsockopt (fd, SOL_TCP, TCP_CORK, &on, sizeof (on)); /* 拔去塞子 */

不幸的是，许多常用的程序并没有考虑到以上问题。例如，Eric Allman编写的sendmail就没有对其套接字设置任何选项。

Apache HTTPD 是因特网上最流行的Web服务器，它的所有套接字就都设置了TCP_NODELAY选项，而且其性能也深受大多数用户的满意。这是为什么呢？答案就在于实 现的差别之上。由BSD衍生的TCP/IP协议栈（值得注意的是FreeBSD）在这种状况下的操作就不同。当在TCP_NODELAY 模式下提交大量小数据块传输时，大量信息将按照一次write()函数调用发送一块数据的方式发送出去。然而，因为负责请求交付确认的记数器是面向字节而 非面向包（在 Linux上）的，所以引入延迟的概率就降低了很多。结果仅仅和全部数据的大小有关系 第一包到达之后就要求确认，FreeBSD则在进行如此操作之前会等待好几百个包。

在Linux系统上，TCP_NODELAY的效果同习惯于BSD TCP/IP协议栈的开发者所期望的效果有很大不同，而且在Linux上的Apache性能表现也会更差些。其他在Linux上频繁采用TCP_NODELAY的应用程序也有同样的问题。

TCP_DEFER_ACCEPT

我 们首先考虑的第1个选项是TCP_DEFER_ACCEPT（这是Linux系统上的叫法，其他一些操作系统上也有同样的选项但使用不同的名字）。为了理 解TCP_DEFER_ACCEPT选项的具体思想，我们有必要大致阐述一下典型的HTTP客户/服务器交互过程。请回想下TCP是如何与传输数据的目标 建立连接的。在网络上，在分离的单元之间传输的信息称为IP包（或IP 数据报）。一个包总有一个携带服务信息的包头，包头用于内部协议的处理，并且它也可以携带数据负载。服务信息的典型例子就是一套所谓的标志，它把包标记代 表TCP/IP协议栈内的特殊含义，例如收到包的成功确认等等。通常，在经过“标记”的包里携带负载是完全可能的，但有时，内部逻辑迫使TCP/IP协议 栈发出只有包头的IP包。这些包经常会引发讨厌的网络延迟而且还增加了系统的负载，结果导致网络性能在整体上降低。

现在服务器创建了一个套接字同 时等待连接。TCP/IP式的连接过程就是所谓“3次握手”。首先，客户程序发送一个设置SYN标志而且不带数据负载的TCP包（一个SYN包）。服务器 则以发出带SYN/ACK标志的数据包（一个SYN/ACK包）作为刚才收到包的确认响应。客户随后发送一个ACK包确认收到了第2个包从而结束连接过 程。在收到客户发来的这个SYN/ACK包之后，服务器会唤醒一个接收进程等待数据到达。当3次握手完成后，客户程序即开始把“有用的”的数据发送给服务 器。通常，一个HTTP请求的量是很小的而且完全可以装到一个包里。但是，在以上的情况下，至少有4个包将用来进行双向传输，这样就增加了可观的延迟时 间。此外，你还得注意到，在“有用的”数据被发送之前，接收方已经开始

在等待信息了。

为了减轻这些问题所带来的影响，Linux（以及其他的一些 操作系统）在其TCP实现中包括了TCP_DEFER_ACCEPT选项。它们设置在侦听套接字的服务器方，该选项命令内核不等待最后的ACK包而且在第 1个真正有数据的包到达才初始化侦听进程。在发送SYN/ACK包之后，服务器就会等待客户程序发送含数据的IP包。现在，只需要在网络上传送3个包了， 而且还显著降低了连接建立的延迟，对HTTP通信而言尤其如此。

这一选项在好些操作系统上都有相应的对等物。例如，在FreeBSD上，同样的行为可以用以下代码实现：

```
/* 为明晰起见，此处略去无关代码 */
struct accept_filter_arg af = { "dataready", "" };
setsockopt(s, SOL_SOCKET, SO_ACCEPTFILTER, &af, sizeof(af));
```

这 个特征在FreeBSD上叫做“接受过滤器”，而且具有多种用法。不过，在几乎所有的情况下其效果与TCP_DEFER_ACCEPT是一样 不等待最后的ACK包而仅仅等待携带数据负载的包。要了解该选项及其对高性能Web服务器的重要意义的更多信息请参考Apache文容。

就HTTP 客户/服务器交互而言，有可能需要改变客户程序的行为。客户程序为什么要发送这种“无用的”ACK包呢？这是因为，TCP协议栈无法知道ACK包的状态。 如果采用FTP而非HTTP，那么客户程序直到接收了FTP服务器提示的数据包之后才发送数据。在这种情况下，延迟的ACK将导致客户/服务器交互出现延 迟。为了确定ACK是否必要，客户程序必须知道应用程序协议及其当前状态。这样，修改客户行为就成为必要了。对Linux客户程序来说，我们还可 以采用另一个选项，它也被叫做TCP_DEFER_ACCEPT。我们知道，套接字分成两种类型，侦听套接字和连接套接字，所以它们也各自具有相应的 TCP选项集合。因此，经常同时采用的这两类选项却具有同样的名字也是完全可能的。在连接套接字上设置该选项以后，客户在收到一个SYN/ACK包之后就 不再发送ACK包，而是等待用户程序的下一个发送数据请求；因此，服务器发送的包也就相应减少了。

TCP_QUICKACK

阻 止因发送无用包而引发延迟的另一个方法是使用TCP_QUICKACK选项。这一选项与 TCP_DEFER_ACCEPT不同，它不但能用作管理连接建立过程而且在正常数据传输过程期间也可以使用。另外，它能在客户/服务器连接的任何一方设 置。如果知道数据不久即将发送，那么推迟ACK包的发送就会派上用场，而且最好在那个携带数据的数据包上设置ACK 标志以便把网络负载减到最小。当发送方肯定数据将被立即发送（多个包）时，TCP_QUICKACK 选项可以设置为0。对处于“连接”状态下的套接字该选项的缺省值是1，首次使用以后内核将把该选项立即复位为1（这是个一次性的选项）。

在某些情形下，发出ACK包则非常有用。ACK包将确认数据块的接收，而且，当下一块被处理时不至于引入延迟。这种数据传输模式对交互过程是相当典型的，因为此类情况下用户的输入时刻无法预测。在Linux系统上这就是缺省的套接字行为。

在 上述情况下，客户程序在向服务器发送HTTP请求，而预先就知道请求包很短所以在连接建立之后就应该立即发送，这可谓HTTP的典型工作方式。既然没有必 要发送一个纯粹的ACK包，所以设置TCP_QUICKACK为0以提高性能是完全可能的。在服务器方，这两种选项都只能在侦听套接字上设置一次。所有的 套接字，也就是被接受呼叫间接创建的套接字则会继承原有套接字的所有选项。

通过TCP_CORK、TCP_DEFER_ACCEPT和 TCP_QUICKACK选项的组合，参与每一HTTP交互的数据包数量将被降低到最小的可接受水平（根据TCP协议的要求和安全方面的考虑）。结果不仅 是获得更快的数据传输和请求处理速度而且还使客户/服务器双向延迟实现了最小化。

二、使用例子说明

1.closesocket（一般不会立即关闭而经历TIME_WAIT的过程）后想继续重用该socket：

```
BOOL bReuseaddr=TRUE;
setsockopt(s,SOL_SOCKET ,SO_REUSEADDR,(const char*)&bReuseaddr,sizeof(BOOL));
```

2. 如果要已经处于连接状态的soket在调用closesocket后强制关闭，不经历TIME_WAIT的过程：

```
BOOL bDontLinger = FALSE;
setsockopt(s,SOL_SOCKET,SO_DONTLINGER,(const char*)&bDontLinger,sizeof(BOOL));
```

3.在send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限：

```
int nNetTimeout=1000;//1秒
//发送时限
setsockopt(socket，SOL_S0CKET,SO_SNDTIMEO，(char *)&nNetTimeout,sizeof(int));
//接收时限
setsockopt(socket，SOL_S0CKET,SO_RCVTIMEO，(char *)&nNetTimeout,sizeof(int));
```

4.在send()的时候，返回的是实际发送出去的字节(同步)或发送到socket缓冲区的字节(异步);系统默认的状态发送和接收一次为8688字节(约为8.5K)；在实际的过程中发送数据和接收数据量比较大，可以设置socket缓冲区，而避免了send(),recv()不断的循环收发：

// 接收缓冲区

int nRecvBuf=32*1024;//设置为32K

setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));

//发送缓冲区

int nSendBuf=32*1024;//设置为32K

setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));

5. 如果在发送数据的时，希望不经历由系统缓冲区到socket缓冲区的拷贝而影响

程序的性能：

int nZero=0;

setsockopt(socket，SOL_S0CKET,SO_SNDBUF，(char *)&nZero,sizeof(nZero));

6.同上在recv()完成上述功能(默认情况是将socket缓冲区的内容拷贝到系统缓冲区)：

int nZero=0;

setsockopt(socket，SOL_S0CKET,SO_RCVBUF，(char *)&nZero,sizeof(int));

7.一般在发送UDP数据报的时候，希望该socket发送的数据具有广播特性：

BOOL bBroadcast=TRUE;

setsockopt(s,SOL_SOCKET,SO_BROADCAST,(const char*)&bBroadcast,sizeof(BOOL));

8.在client连接服务器过程中，如果处于非阻塞模式下的socket在connect()的过程中可

以设置connect()延时,直到accpet()被呼叫(本函数设置只有在非阻塞的过程中有显著的

作用，在阻塞的函数调用中作用不大)

BOOL bConditionalAccept=TRUE;

setsockopt(s,SOL_SOCKET,SO_CONDITIONAL_ACCEPT,(const char*)&bConditionalAccept,sizeof(BOOL));

9.如果在发送数据的过程中(send()没有完成，还有数据没发送)而调用了closesocket(),以前我们

一般采取的措施是"从容关闭"shutdown(s,SD_BOTH),但是数据是肯定丢失了，如何设置让程序满足具体

应用的要求(即让没发完的数据发送出去后在关闭socket)?

```
struct linger {
u_short l_onoff;
u_short l_linger;
};
linger m_sLinger;
m_sLinger.l_onoff=1;//(在closesocket()调用,但是还有数据没发送完毕的时候容许逗留)
// 如果m_sLinger.l_onoff=0;则功能和2.)作用相同;
m_sLinger.l_linger=5;//(容许逗留的时间为5秒)
setsockopt(s,SOL_SOCKET,SO_LINGER,(const char*)&m_sLinger,sizeof(linger));
```

setsockopt()用法

2007/12/05 19:01

一下来源于互联网：

1.closesocket（一般不会立即关闭而经历TIME_WAIT的过程）后想继续重用该socket：

```
BOOL bReuseaddr=TRUE;
setsockopt(s,SOL_SOCKET ,SO_REUSEADDR,(const char*)&bReuseaddr,sizeof(BOOL));
```

2. 如果要已经处于连接状态的soket在调用closesocket后强制关闭，不经历

TIME_WAIT的过程：

```
BOOL bDontLinger = FALSE;
setsockopt(s,SOL_SOCKET,SO_DONTLINGER,(const char*)&bDontLinger,sizeof(BOOL));
```

3.在send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限：

```
int nNetTimeout=1000;//1秒
//发送时限
setsockopt(socket，SOL_S0CKET,SO_SNDTIMEO，(char *)&nNetTimeout,sizeof(int));
```

//接收时限

setsockopt(socket，SOL_S0CKET,SO_RCVTIMEO，(char *)&nNetTimeout,sizeof(int));

4.在send()的时候，返回的是实际发送出去的字节(同步)或发送到socket缓冲区的字节(异步);系统默认的状态发送和接收一次为8688字节(约为8.5K)；在实际的过程中发送数据和接收数据量比较大，可以设置socket缓冲区，而避免了send(),recv()不断的循环收发：

// 接收缓冲区

int nRecvBuf=32*1024;//设置为32K

setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));

//发送缓冲区

int nSendBuf=32*1024;//设置为32K

setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));

5. 如果在发送数据的时，希望不经历由系统缓冲区到socket缓冲区的拷贝而影响程序的性能：

int nZero=0;

setsockopt(socket，SOL_S0CKET,SO_SNDBUF，(char *)&nZero,sizeof(nZero));

6.同上在recv()完成上述功能(默认情况是将socket缓冲区的内容拷贝到系统缓冲区)：

int nZero=0;

setsockopt(socket，SOL_S0CKET,SO_RCVBUF，(char *)&nZero,sizeof(int));

7.一般在发送UDP数据报的时候，希望该socket发送的数据具有广播特性：

BOOL bBroadcast=TRUE;

setsockopt(s,SOL_SOCKET,SO_BROADCAST,(const char*)&bBroadcast,sizeof(BOOL));

8.在client连接服务器过程中，如果处于非阻塞模式下的socket在connect()的过程中可

以设置connect()延时,直到accpet()被呼叫(本函数设置只有在非阻塞的过程中有显著的

作用，在阻塞的函数调用中作用不大)

```
BOOL bConditionalAccept=TRUE;
setsockopt(s,SOL_SOCKET,SO_CONDITIONAL_ACCEPT,(const char*)&bConditionalAccept,sizeof(BOOL));
```

9.如果在发送数据的过程中(send()没有完成，还有数据没发送)而调用了closesocket(),以前我们

一般采取的措施是"从容关闭"shutdown(s,SD_BOTH),但是数据是肯定丢失了，如何设置让程序满足具体

应用的要求(即让没发完的数据发送出去后在关闭socket)?

```
struct linger {
u_short l_onoff;
u_short l_linger;
};
linger m_sLinger;
m_sLinger.l_onoff=1;//(在closesocket()调用,但是还有数据没发送完毕的时候容许逗留)
// 如果m_sLinger.l_onoff=0;则功能和2.)作用相同;
m_sLinger.l_linger=5;//(容许逗留的时间为5秒)
setsockopt(s,SOL_SOCKET,SO_LINGER,(const char*)&m_sLinger,sizeof(linger));
```