# **Linux高性能服务器编程**

目录

[1. SocketAddress 的Api](#_Toc538992538_WPSOffice_Level1) [3](#_Toc538992538_WPSOffice_Level1)

[2. Socket的 Api](#_Toc91860774_WPSOffice_Level1) [3](#_Toc91860774_WPSOffice_Level1)

[A. 主机字节序和网络字节序](#_Toc91860774_WPSOffice_Level2) [4](#_Toc91860774_WPSOffice_Level2)

[B. Socket创建](#_Toc568667797_WPSOffice_Level2) [4](#_Toc568667797_WPSOffice_Level2)

[C. 命名Socket](#_Toc1824716092_WPSOffice_Level2) [4](#_Toc1824716092_WPSOffice_Level2)

[D. 监听socket](#_Toc1821452878_WPSOffice_Level2) [5](#_Toc1821452878_WPSOffice_Level2)

[E. 接受连接](#_Toc2041977979_WPSOffice_Level2) [5](#_Toc2041977979_WPSOffice_Level2)

[F. 发起连接](#_Toc1873149729_WPSOffice_Level2) [5](#_Toc1873149729_WPSOffice_Level2)

[G. 关闭了连接](#_Toc431442179_WPSOffice_Level2) [6](#_Toc431442179_WPSOffice_Level2)

[H. 数据的读写](#_Toc122974607_WPSOffice_Level2) [6](#_Toc122974607_WPSOffice_Level2)

[1) TCP数据的读写](#_Toc91860774_WPSOffice_Level3) [6](#_Toc91860774_WPSOffice_Level3)

[2) 通用数据读写函数](#_Toc568667797_WPSOffice_Level3) [7](#_Toc568667797_WPSOffice_Level3)

[I. 带外标记](#_Toc568667797_WPSOffice_Level1) [8](#_Toc568667797_WPSOffice_Level1)

[J. socket选项](#_Toc1772518686_WPSOffice_Level2) [9](#_Toc1772518686_WPSOffice_Level2)

[1) 重要的socket选项](#_Toc1824716092_WPSOffice_Level3) [10](#_Toc1824716092_WPSOffice_Level3)

[K. 网络信息Api](#_Toc962914725_WPSOffice_Level2) [10](#_Toc962914725_WPSOffice_Level2)

基本的API

## SocketAddress 的Api

定义在bits/socket.h中,sa\_faimly 代表地址族类型，sa\_data 存放socket地址的值。

struct sockaddr{

sa\_family\_t sa\_family;

char sa\_data[14];

}

由于sa\_data的容量太小，无法容纳多数协议的地址值，Linux 有定义了新的socketAddress结构体，socketaddr\_storage,不仅提供了足够大的空间用于存放地址值，

而且内存对齐的，\_\_ss\_padding用于填充的作用。

strut sockaddr\_storage{

sa\_family\_t sa\_family;

unsighned long int \_\_ss\_align;

char \_\_ss\_padding[128-sizeof(\_\_ss\_align)];

}

以上SocketAddress的结构体并不方便获得IP地址和端口，因此Linux为各个协议族定义了自己专门的socketAddress结构体

PF\_UNIX协议族

地址族类型通常与协议族类型一致,地址族类型与协议族类型也都定义在bits/socket.h文件中。

|  |  |
| --- | --- |
| 协议族类 | 地址族类型 |
| PF\_UNIX | AF\_UNIX |
| PF\_INET | AF\_INET |
| PF\_INET6 | AF\_INET6 |

## Socket的 Api

定义在sys/socket.h 头文件中，包括创建socket ,命名socket,监听socket,接受连接，发起连接，读写数据，获取地址，检测外带标记；

### 主机字节序和网络字节序

现代CPU的累加器可以一次加载4个字节（32位的机器），那么四个字节在内存中的排序，将影响被装载的值；大端字节序是指一个整数的高位字节存储到内存的地低地址处；小端字节序是指整数的高位排在内存高地址处；现代的PC大部分采用小端字节序，网络采用大端字节序；

在netinet/in.h文件中提供了主机字节序和网络字节序之间的转换；

//长整数据主机转成网络字节序 （IP地址转换）

Unsighned long int htonl(unsighned long int hostLong);

//长整数据网络转成主机字节序 （IP地址转换）

Unsighned long int ntohl(unsighned long int hostLong);

//短整数据主机转成网络字节序 （port端口转换）

Unsighned short int htons(unsighned shortint hostShort);

//短整数据网络转成主机字节序 （port端口转换）

Unsighned short int ntohs(unsighned long int hostShort);

### Socket创建

定义在sys/socket.h文件中

@Param domain 表示协议PF\_UNIX,PF\_INET,PF\_INET6

@Param type 若是SOCKET\_STREAM（传输层是TCP协议）若是SOCKET\_DGRAM

表示传输层是UDP协议

@Param protocol 若为0表示选择默认的协议；

@return 返回一个文件描述符号

int socket(int domain,int type,int protocol);

### 命名Socket

在创建socket时候，我们指定了地址族，但是并未指定使用该地址族的具体那个scoketaddress，我们将一个socket 与socket地址绑定称为socket命名；

定义在sys/socket.h

@Param socketfd socket文件描述符 创建socket的时候返回的文件描述符

@Param my\_addr 表示socketaddress分配给未命名的sockfd文件描述符号

@Param addrlen

@return 表示绑定的结果 0表示成功

int bind(int socketfd, const struct sockaddr \*my\_addr, socklen\_t addrlen);

### 监听socket

socket被命名之后，还不能立刻接受客户的连接，我们需要使用系统调用来创建一个监听队列以存放待处理的客户连接；

定义在sys/socket.h

@Param socketfd 表示文件描述符号

@backlog 内核监听队列的最大长度 在Linux 2.2版本之前包括半连接状态（SYN\_RCVD）和完全连接状态(ESTABLISHED)

@Return 0表示成功

int listen(int socketfd, int backlog)

### 接受连接

@return accept成功的时候，返回一个新连接的socket,该socket唯一地标识了被接受的这个连接，服务器可以通过读写该socket来与被接收连接对应的客户端通信；accept失败的是后返回-1并设置errno

int accpet(int sockfd, struct sockaddr\* addr,socklen\_t \*addrlen);

### 发起连接

客户端通过下面的方法主动与服务端发起连接

定义在sys/socket.h文件中

@param server\_addr 是服务器监听的socket地址

＠return 0表示连接成功

int connet(int sockfd, const struct sockaddr\* serv\_addr, socklen\_t addrlen);

### 关闭了连接

关闭一个连接实际就是关闭该连接对应的socket;

定义在unistd.h文件中

int close(int sockfd);

close并不是立即关闭一个连接，而是将sockfd的引用计数减少１，只有当引用计数为０的时候有才真正关闭连接；在多进程程序中，一次fork系统调用默认使父进程中打开的socket的引用计数加１；因此我们必须在父进程和子进程中都对该socket 执行close调用才能将连接关闭；如果无论如何都有立即终止连接可以使用shutdown系统调用；

@param howto SHUT\_RD 表示关闭读的操作；SHUT\_WR 表示关闭写的操作；SHUT\_RDWR 表示关闭读写；

@return ０表示关闭成功

int shutdown(int sockfd, int howto);

### 数据的读写

#### TCP数据的读写

定义在sys/socket.ｈ文件中，

读取sockfd上的数据，

＠param buf 是指定的读缓存区的位置

＠len 缓存区的大小

@flags 通常设置为０

＠return 返回读取到的数据的长度；若为０对方已经关闭了连接；若为-1表示出错；

ssize\_t recv(int sockfd,void\* buf,size\_t len, int flags);

往sockfd上写入数据

＠param buf表示写缓存区的位置

＠len 表示写的缓存区的大小

＠flags 参考一下列表　可以是某项或者通过几项的或

ssize\_t send(int sockfd,const void\*buf,size\_t len, int flags);

Flags选项名称

|  |  |  |  |
| --- | --- | --- | --- |
| 选项名字 | 含义 | send | recv |
| MSG\_CONFIRM | 提示数据链路层持续监听对方响应，直到得到答复；它仅限于SOCK\_DGRAM 和SOCK\_RAW的socket | Y | N |
| MSG\_DONTROUTE | 不查看路由表，直接将数据发送给本地局域网主机。表示发送者确切知道目标主机就在本地局域网内 | Ｙ | N |
| MSG\_DONTWAIT | 对scoket的此次操作是非堵塞的 | Y | Y |
| MSG\_MORE | 告诉内核应用程序还有更多的数据发送，内核将超时等待新数据写入TCP发送缓存区后一并发送。这样可以防止ＴCP发送过多的小的报文段，从而提高传输效率。 | y | N |
| MSG\_PEEK | 探寻读缓存区中的数据，此次操作不会导致这些数据清除 | N | Y |
| MSG\_WAITALL | 读操作在读取到一定数量的数据才会返回 | N | Y |
| MSG\_OOB | 发送或者接收紧急数据 | Ｙ | Y |
| MSG\_NOSIGNAL | 往读端关闭的管道或者socket连接时候写数据不引发SIGPIPE | Y | N |

#### 通用数据读写函数

socket 编程接口还提供了一对通用的数据读写系统调用，不仅能用于TCP流数据，也能用于UDP数据报；

sys/socket.h

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

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

struct msghdr{

void\* msg\_name;//通信对方的socket地址，对于TCP协议没有意义，必须为NULL

socklen\_t msg\_namelen;//socket地址长度

@param msg\_iov 对于recvmsg 而言数据将读取并且放存msg\_iovlen块分散的内存中，这些内存的地址和长度由msg\_iov指向的数组指定，这称为分散读；对于sendmsg而言，msg\_iovlen块分散内存中的数据将被一并发送，这称为集中写；

struct iovect\*msg\_iov;//分散内存块，指定了iovec这样的结构对象有多少个

void \*msg\_control;//指向辅助数据的起始位置，实现在进程间传递文件描述符号

socklen\_t msg\_controllen;//辅助数据的大小

int msg\_flags;//复制函数中的flags参数，并在调用过程中更新

}

struct iovec{

void\*iov\_base;//内存的起始地址

size\_t iov\_len;//这块内存地址长度

}

### 带外标记

在实际应用中，我们通常无法预期带外数据何时到来，好在Linux内核检测到TCP紧急标志的时候，将通知应用程序有带外数据需要接收；内核通知应用程序带外数据到达两种常见的方式：I/O复用产生的异常事件和SIGURG信号。但是即使应用程序得到了有带外数据需要接收的通知，还需要知道带外数据在数据流中的具体位置，才能准确接收带外数据。这一点可以通过如下系统调用实现：

sys/socket.h

@return 表示下一个被读取到的数据是否是带外数据 若返回1 此时就可以使用带MSG\_OOB标记的recv调用来接收带外数据。如果不是则返回0；

int sockatmark(int sockfd);

### socket选项

sys/socket.h

@Param level 表示那个协议IPV4 IPV6 TCP

@param option\_name

int getsockopt(int sockfd,int level,int option\_name,

void\*option\_value,scoket\_t\*restrict option\_len);

int setsockopt(int sockfd,int level int option\_name,const void\*option\_value,

socklen\_t option\_len);

|  |  |  |  |
| --- | --- | --- | --- |
| level | option\_name | 数据类型 | 说明 |
| SOL\_SOCKET | SO\_DEBUG | int | 打开调试信息 |
| SO\_REUSEADDR | int | 重复使用本地地址 |
| SO\_TYPE | int | 获取socket类型 |
| SO\_DONTROUTE | int | 不查看路由表，直接将数据发送给局域网内的主机 |
| SO\_RCVBUF | int | TCP接收缓存区大小 |
| SO\_SNDBUF | int | TCP发送缓存区大小 |
| SO\_KEEPALIVE | int | 发送周期性保活 |
| SO\_OOBINLINE | int | 接收到的带外数据将存储到普通数据输入队列中，此时我们不能使用MSG\_OOB标志读取带外数据应该像普通数据一样读取 |
| SO\_LINGER | linger | 若有数据待发送延迟关闭 |
| SO\_RCVLOEAT | int | TCP接收缓存区低水位标记 |
| SO\_SNDLOWAT | int | TCP发送区低水位标记 |
| SO\_RCVTIMEO | timeval | 接收数据超时 |
| SO\_SNDTIMEO | timeval | 发送数据超时 |
| IPPROTO\_IP | IP\_TOS | int | 服务类型 |
| IP\_TTL | int | 存活时间 |
| IPPROTO\_IPV6 | IPV6\_NEXTHOP | sockaddr\_in6 | 下一跳IP地址 |
| IPV6\_RECVPKINFO | int | 接收分组信息 |
| IPV6\_DONTFRAG | int | 截止分片 |
| IPV6\_RECCLASS | int | 接收通信类型 |
| IPPROTO\_TCP | TCP\_MAXSEG | int | TCP最大报文段小 |
| TCP\_NODELAY | int | 禁止NAGLE算法 |
|  |  |  |  |

值得指出的对于服务端，是对于部分设置项，必须在listen调用之前才会生效。这是因为连接socket只能由accept调用返回，但是accept从listen监听的队列中接受的链接至少完成TCP三次握手的前两个步骤（listen监听队列中的连接至少已进入SYN\_RCVD状态）；对于客户端而言，这些socket选项应该在connect调用之前设置，应为在connect调用成功之后TCP三次握手已经完成；

#### SO\_LINGER

SO\_LINGER选项用于控制close系统关闭链接时的行为。默认情况下，当close系统调用来关闭一个socket的时候，close立即返回，TCP模块负责将该socket对应的TCP发送缓存区的残留数据发送给对方；

sys/socket.h

struct linger{

@param 1\_onoff 0表示关闭，即当前选项不起作用，

使用默认行为关闭socket 非0表示开启

int 1\_onoff;

@param l\_linger在1\_onoff开启的时候（1\_onoff != 0）若l\_linger ==0 close系统调用立即返回，TCP模块丢弃socket发送缓存区的数据，同时发送对方一个复位报文；服务器提供了异常终止连接的一个方法；若l\_linger> 0,对于阻塞的socket的close 会等待待一段时间直到残留在发送缓存区的数据发送完毕，并得到对方确认收到。如果对方没有确认close()返回-1；若socket是非阻塞的，close 立即返回；

int l\_linger;//滞留时间

}

#### SO\_RCVLOWAT 和SO\_SNDLOWAT

SO\_RCVLOWAT和SO\_SNDLOWAT选项分别表示TCP接收缓冲区和发送缓冲区的低水位标记。它们一般被I/O复用系统调用（见第9章）用来判断socket是否可读或可写。当TCP接收缓冲区中可读数据的总数大于其低水位标记时，I/O复用系统调用将通知应用程序可以从对应的socket上读取数据；当TCP发送缓冲区中的空闲空间（可以写入数据的空间）大于其低水位标记时，I/O复用系统调用将通知应用程序可以往对应的socke上写入数据。

默认情况下，TCP接收缓冲区的低水位标记和TCP发送缓冲区的低水位标记均为1字节。

#### SO\_RCVBUF 和 SO\_SNDBUF

当我们通过setsockopt这个系统方法设置TCP的接收缓存区和发送缓存区大小的时候，系统会将其值加倍，并且不小于最小值。TCP接收缓存区最小值是256字节，TCP发送缓存区最小值是2048字节；

### 网络信息Api

socket地址的两个要素，即IP地址和端口号，都是数值表示的，这不便于记忆，也不便于扩展。因此在前面的章节中，我们用主机名来访问一台机器，而避免直接使用其IP地址。同样，我们用主机名来访问同一个机器，而避免直接使用器IP地址。同样我们使用服务名称来代替端口号。比如下面两条telnet命令具有完全相同作用：

telnet 127.0.0.1 80

telnet localhost www

#### gethostbyname 和gethostbyaddr

定义在/sys/netdb.h,实现主机名和IP地址之间的转换，以及服务名称和端口之间的转换；

当一个主机有多个网络接口的时候，自然有多个地址

struct hostent{

char \* h\_name;

char\*\* h\_aliases;//主机别名表，可能有多个

int h\_lenght;//地址长度

char\*\* h\_addr\_list;//按照网络字节序列出主机的IP地址列表

}

通过主机名字获得主机的完整信息，该函数一般通过/etc/hosts配置文件查找主机，没有找到通过DNS服务器查找；

@param name 主机名称

struct hostent\* gethostbyname(const char\*name);

通过ip地址获得主机的完整信息

@param addr IP地址

@len 地址的长度

@type AF\_INET

struct hostent\* gethostbyaddr(const void\*addr,size\_t len,int type);

#### getservbyname 和getservbyport

根据名称获得某个服务的完整信息

struct servent\*getservbyname(const char\*name,const char\*proto);

根据端口号获得某个服务的完整信息

struct servent\*getservbyport(int port,const char\*proto);

struct servent

{

char\*s\_name;/\*服务名称\*/

char\*\*s\_aliases;/\*服务的别名列表，可能有多个\*/

int s\_port;/\*端口号\*/

char\*s\_proto;/\*服务类型,通常是tcp或者udp\*/

};

#### getaddrinfo

getaddrinfo可以通过主机名字获得IP地址，也可以通过服务名称获得端口号

@param hostname 可以接收主机名称，也可以接收字符串表示的IP地址

@param service可以是服务名也可以是十进制的端口号

@param hints 若为NULL 表示接受任何的结果

@result 用于存储结果

int getaddrinfo(const char\*hostname, const char\*service,const struct addrinfo\*hints,struct addrinfo\*\*result);

struct addrinfo{

int ai\_flags;

int ai\_family;

int ai\_socktype;//SOCK\_STREAM或者SOCK\_DGRAM

int ai\_protocol;

socklen\_t ai\_addrlen;//

char\* ai\_canonname;//主机别名

struct sockaddr\*ai\_addr;

struct addinfo\*ai\_next;

}