# 简介

Linux中的网络通过socket接口实现，socket即是一种特殊的IO，它也是一种文件描述符。一个完整的socket都有一个相关描述(协议，本地地址，本地端口，远程地址，远程端口)；每个socket有一个本地的唯一socket号，由操作系统分配。

Socket是一种IPC方法，允许位于统一主机或者使用网络连接起来的不同主机上的应用程序之间交换数据。

Socket使用标准文件描述符（file description）和其他程序进行通讯。

## 通信domain

Socket存在于一个通信domain中，它确定：

1、识别出一个socket的方法（即socket地址的格式）；

2、通信范围（即是在位于同一主机上的应用程序之间还是位于同一网络连接起来的主机应用程序之间）。

现在操作系统至少支持一下domain：

UNIX(AF\_UNIX)：domain允许在同一主机上的应用程序之间进行通信。

IPv4(AF\_INET)：domain允许在使用IOv4网络连接起来的主机上应用程序之间通信。

IPv6(AF\_INET6)：domain允许在使用IPv6网络连接起来主机上应用程序之间通信。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Domain | 执行通信 | 应用间通信 | 地址格式 | 地址结构 |
| AF\_UNIX | 内核中 | 同一主机 | 路径名 | sockaddr\_un |
| AF\_INET | 通过IPv4 | IPv4网络主机 | 32位IPv4地址+16位端口号 | sockaddr\_in |
| AF\_INET6 | 通过IPv6 | IPv6网络主机 | 128位IPv4地址+16位端口号 | sockaddr\_un6 |

说明：AF=Address Family，即地址族

## Socket类型

每个socket实现至少提供两种socket：流和数据报。

### 流式套接字

流socket（SOCK\_STREAM）：提供了一个可靠的双向的字节流通信信道。含义如下：

1）可靠的：表示可以保证发送者传输的数据会完整地送达接收应用程序或者收到一个传输失败的的通知；

2）双向：表示数据可以在两个socket之间的任意方向上传输；

3）字节流：表示与管道一样不存在消息边界的概念。

Telenet使用的是流式套接字，你输入的字符需要按照顺序到达：www浏览器使用的HTTP协议也是用它们来下载页面，实际上当你通过80telnet一个www站点，然后输入“GET pagename”的时候，也可以得到HTML的内容。

流式套接字之所以能够达到高质量的数据传输，是因为使用了“传输控制协议TCP”，TCP控制你的数据按照顺序到达没有错误。

### 数据报套接字

数据报socket（SOCK\_DGRAM）允许数据以被称为数据报的消息的形式进行交换。在数据报socket中，消息边界得到了保留，但是数据传输是不可靠的。消息的达到可能是无序的、重复的或者根本无法到达。

注：数据报套接字有时候也叫“无连接套接字”。

数据报也是使用IP做路由，但是不使用TCP。它是无连接的，只需要建立一个包，构造一个目标信息的IP头，然后发出去，无需建立连接。

数据报套接字的应用程序有tftp、bootp等。

**总结：**

在Internet domain中，数据报socket使用了用户数据报协议（UDP），而流socket则（通常）使用了传输控制协议（TCP），分别使用术语“UDP socket”和“TCP socket”。

## 结构体

有两种字节排列顺序：重要的字节（有时候叫octet，即8位位组）在前面，或者不重要的字节在前面（即我们所说的高8位低8位）。

前一种叫“网络字节顺序（Network Byte Order，NBO）”，有些机器在内部是按照这个顺序存储数据的。当某数据必须按照NBO顺序，那么要调用函数转换过来。如果没有提到NBO，那么就让它保持本机字节顺序。

## 字节序

### 概述

字节序指的是多字节的数据各字节的存储顺序。在几乎所有计算机中，多字节数据被存储为连续的字节序列。例如，一个4字节的int类型变量a，其存储的起始地址为0x804900，那么a的四个字节将被分别存在0x804900，0x804901，0x804902，0x804903的位置。但是问题来了，a的最低有效位可以存储在最前面，也可以存储在高最后面，就有两种不同的存储顺序。这就引出了大端序和小端序。

### 大小端

实际上，如果最低有效位在最高有效位的前面，则该存储规则为小端序；反之，如果最低有效位在最高有效的后面，则该存储规则为大端序。不同的处理器体系，采用的字节序可能是不同的。例如，x86采用小端序，而PowerPc 970等采用大端序。那么如此一来，不同机器之间的数据传输是不是会出问题呢？

**本地序和网络序**

本地序（也称主机序）即指前面处理器本身所采用的字节序，因此有的大端序，有的小端序。而网络序，是指网络传输采用的字节序。所幸，网络序是标准化的，即一般统一采用大端序。因此，发送网络数据之前需要将数据转换为网络序，从而避免了前面所担心的问题。而C语言也针对整型数据提供了一组接口，htonl、htons用于本地序转网络序，以及ntohl、ntohs用于网络序转本地序。

我们的X86 CPU都是采用小端模式，网络传输的都是大端，二者需要调用系统函数转换。

比如我们输入0x1234（12是高字节，34是低字节），大端和小端的输出结果分别是：

小端：34 12（低地址存储在低位）

大端：12 34（高地址存储低位地址，最直观）

### 必要性

既然每次都发送网络数据之前都要转换，为什么不统一字节序呢？实际上，大小端各有优劣：

计算都是从低位开始的，因此计算机内部处理采用小端序，效率较高。

而大端序存储的时候，由于符号位在高位，因此对于数据征服或大小的判断也就方便许多。另外，大端序也更符合人类的阅读习惯。

再由于各个芯片厂商的坚持，字节序的问题也就一直没有统一。大小端争端起源于吃鸡蛋时先打破大端还是小端，有兴趣的读者可以搜索一下。

因为每个机器内部对变量的字节存储顺序不同(有的系统高位在前，低位在后，而

有的系统是低位在前，高位在后)，而网络传输的数据大家是一定要统一顺序的。

所以对于内部字节表示顺序和网络字节顺序不同的机器，就一定要对数据进行转换。

### 函数

字节序列转换：

* htons()---- "Host to Network Short"主机字节顺序转换为网络字节顺序
* htonl()----"Host to Network Long"主机字节顺序转换为网络字节顺序
* ntohs()----"Network to Host Short"网络字节顺序转换为主机字节顺序
* ntohl()----"Network to Host Long"网络字节顺序转换为主机字节顺序

### 总结

在网络应用中，字节序的问题不可忽略，否则可能出现无法预知的问题（如果两台机器本地序相同，且都不做字节序转换，那么侥幸不会出现什么问题）。通过前面的介绍和分析，我们总结出以下几点：

不同处理器之间采用的字节序可能不同。

有些处理器的字节序是确定的，有些处理器的字节序是可配置的。

网络序一般统一为大端序。

数据从本地传输到网络，需要转换为网络序，接收到的网络数据需要转换为本地序后使用。

C提供了一组接口用于整型数据在本地序和网络序之间的转换。

多字节数据对象才需要转字节序，例如int，short等，而char不需要。

由于处理器是按照IEEE标准处理float和double的（参考：对浮点数的一些理解），因此也不需要转字节序。

由于Java虚拟机的存在，Java不需要考虑大小端的问题。

## IP地址转换

IP地址转换为长整型：

inet\_addr()返回的地址已经是网络字节格式，所以无需调用函数htonl()。

长整型转为IP地址：

inet\_ntoa() [ntoa=network to asci]

# 系统调用

关键的socket系统调用包括以下几种：

socket()系统调用：创建一个新的socket；

bind()系统调用：将一个socket绑定到一个地址上，通常，服务器需要使用这个调用来将其socket绑定到一个众所周知的地址上使得客户端能够定位到该socket上；

listen()系统调用：允许一个流socket接受来自其他socket的接入连接；

accept()系统调用：在一个监听流socket上接受来自一个对等应用程序的连接，并可选地返回对等socket的地址；

connect()系统调用：建立一个与另一个socket之间的连接。

Socket I/O可以使用传统的read()和write()系统调用或使用一组socket特有的系统调用（如send()、recv()、sendto()、recvfrom()）来完成。在默认情况下，这些系统调用在I/O操作无法被立即完成时会被阻塞。通过使用fcntl() F\_SETFL操作来启用O\_NONBLOCK打开文件状态标记可以执行非阻塞I/O。

## socket()

原型：int socket(int domain, int byte, int protocol);

参数：domain——应该设置为AF\_INET

type——告诉内核socket类型，SOCK\_STREAM/SOCK\_DGRAM

protocol——设置为0

返回值：返回以后再调用中可能用到的socket描述符，或者在错误的时候返回-1，全局变量errno中将存储返回的错误值。

socket域:

AF\_INET IPV4英特网域

AF\_INET6 IPV6英特网域

AF\_UNIX unix域

AF\_UNSPEC 未指定

## htons

htons()用来将当前主机字节序转为王阔字节序，其中h代表主机（host）字节序，n代表网络（network）字节序，s代表short，htos是h、to、n、s的组合，可以理解为“将short型数据从当前主机字节序转换为网络字节序”。

常见的网络字节转换函数有：

htons()：host to network short，将short类型数据从主机字节序转换为网路字节序；

ntohs()：network to host short，将short类型数据从网络字节序转换为主机字节序；

htonl()：host to network long，将long类型数据从主机字节序转换为网络字节序；

nohl()：network to host long，将long类型数据从网络字节序转换为主机字节序。

通常，以s后缀的函数中，s代表2字节short，因此用于端口号转换；以l为后缀的函数，l代表4字节的long，因此用于IP地址转换。

典型应用：

//创建sockadd\_in结构体变量

struct sockaddr\_in, serv\_addr;

memset(&serv\_addr, 0, sizeof(serv\_addr)); //每个字节都用0填充

serv\_addr.sin\_famliy= AF\_INET; //使用IPV4地址

serv\_addr.sin\_addr.s\_addr = inet\_addr(“127.0.0.1”); //具体IP地址

serv\_addr.sin\_port = htons(8080); //端口号

## bind()

原型：bind(int sockfd, struct sockaddr \*my\_addr, int addlen);

参数：

sockfd——调用socket返回的文件描述符

my\_addr——指向数据结构struct sockaddr的指针，保存地址（即IP和端口）信息

addlen——设置为sizeof(struct sockaddr)

对于服务器程序而言，创建的套接字必须绑定到一个本地的IP地址和端口号以用于TCP通信。Linux的bind()函数就是用于完成该工作。

在将套接字绑定到一个地址和端口号之后，服务器程序必须准备好接受来自远程客户端的连接请求。这是一个分为两步的过程：listen()和accept()。

注：bind()用于将套接字和机器上的某一个端口建立绑定关系，如果你想用listen()来侦听某个端口的数据，这是必要的操作，如果只想用connect()这一步不是必要的。使用connect与远程主机通信时，不需要关系本地的端口号，只需要简单调用connect就可以了，它会检查套接字是否绑定端口，如果没有，它会自己绑定一个没有使用的本地端口。

在调用bind的时候，不要采用小于1024的端口号，所有小于1024的端口号都被系统保留，可以选择1024到65535的端口（如果没有别的程序使用的话）。

## listen()

原型：int listen(int sockfd, int backlog);

参数：

backlog——在进入队列中允许的连接数目

程序调用listen()以开始监听客户端的连接，然后调用accept来真正地接受客户端的请求。

## accept()

原型：int accept(int s, struct sockaddr \*addr, int addlen);

参数：

在listen()函数调用之后，accept()函数必须被调用以等待进入的连接。accept()函数是一个阻塞式函数。程序的执行将暂停在accept()函数调用上直到有一个客户端的连接请求到达。

当一个连接请求被接受后，accept()函数将返回一个新的套接字描述符。这个新的套接字用于与远程客户通信，这样就有两个套接字了，原来的一个还在侦听你绑定的端口，新的正在准备发送send()和recv()数据。由socket()函数创建的原来的套接字仍然可以被用来继续监听更多的客户端连接。

## connect()

原型：int connect(int sockfd, struct sockaddr \*serv\_addr, int addlen);

参数：

sockfd——系统调用socket()返回的套接字文件描述符

serv\_addr——保存着目的端口和IP地址的数据结构

addlen——设置为sizeof(struct sockaddr)

## send

数据报套接字发送和接收函数：send()和recv()。

一旦一个连接请求被接受，服务器就可以使用新的套接字描述符向客户端发送数据或接收来自客户端的数据，send()函数不会阻塞程序的执行。缓冲区中的数据被发送到系统中底层的TCP发送缓冲区，然后函数调用将返回。有可能在send()函数中定义的缓冲区中的数据不会全部被发送出去。Send()函数调用返回的整数值将表示有多少字节的数据已被发送到TCP发送缓冲区。确认这个返回值匹配缓冲区的大小以确保所有的数据都已被发送出去是非常重要的。

## recv

recv()函数是一个阻塞式函数。程序的执行将暂停直到recv()函数从远程客户端接收到数据或者远程客户端明确地断开TCP会话。如果客户端断开TCP会话，recv()函数将返回0.如果客户端发送一个数据包，recv()函数将接收到的数据放入定义的缓冲区并返回接收到的字节数。

在设计客户端和服务器的应用程序时，同步发送和接收函数时非常重要的。如果服务器和客户端同时等待在recv()函数调用上，它们将会产生死锁，并且不会有通信发生。

## sendto()

在面向连接的套接字编程中，客户端使用connect()函数与服务器建立TCP连接。无连接套接字并不会在网络设备之间创建一个专用的连接。相反，数据被扔到网络中，并希望它们能够自己到达指定的目的地。无连接套接字的好处是传输数据的开销很小，它不需要认真地跟踪每一个数据包，这极大地降低了处理数据包所需的开销。正因为如此，无连接套接字比面向连接套接字有更好的吞吐量。

无连接套接字使用套接字类型SOCK\_DGRAM实现。它通常使用用户数据报协议（UDP）在网络设备之间传递数据。由于无连接套接字不需要建立连接，所以服务器和客户端程序看起来很相似。

要发送UDP消息，套接字不需要使用bind函数。我们使用sendto()函数来定义数据和它要发送到的目的地。

## recvfrom()

要接收UDP消息，套接字必须使用bind()函数绑定到一个UDP端口以监听进入的数据包，这个和TCP连接一样。在bind()函数调用之后，数据包可以使用recvfrom()函数来接收。

由于recvfrom()函数是无连接的，所以它可以接收来自任何网络设备的数据。发送主机的网络地址信息保存在结构from中，而接收到的消息保存在缓冲区msg中。

## shutdown

套接字描述符可以使用shutdown()函数来指定通信会话终止的方式。该函数的原型如下：

shutdown(int socket, int how);

shutdown()函数使用参数how来让你决定如何优雅地关闭连接。可使用的选项如下：

0——不再接收数据包；

1——不再发送数据包；

2——不再发送或接收数据包。

通过选择0或1，你可以禁止套接字接收或发送更多的数据，但是允许套接字继续将剩余的数据发送完或继续接收剩余的数据。

## close()

原型：close(sockfd);

当连接处理完所有的这些数据之后，我们可以调用close()函数终止连接而不会造成任何数据丢失。

如果你想在如何关闭套接字上有多一点的控制，可以使用函数shutdown，它允许你将一定方向上的通讯或者双向通讯关闭。

## getpeername()

在连接的流式套接字上谁在另一边。

## gethostname()

返回程序所运行主机的主机名称。

# UNIX domain

# TCP/IP

# Internet domain

# socket服务器设计

# socket高级特性