Linux 下 Socket 编程

网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用 Socket()，该函数返回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。

什么是 Socket

Socket 接口是 TCP/IP 网络的 API，Socket 接口定义了许多函数或例程，程序员可以用它们来开发 TCP/IP 网络上的应用程序。要学 Internet 上的 TCP/IP 网络编程，必须理解 Socket 接口。

Socket 接口设计者最先是将接口放在 Unix 操作系统里面的。如果了解 Unix 系统的输入和输出的话，就很容易了解 Socket 了。网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用 Socket()，该函数返 回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。常用的 Socket 类型有两种：流式 Socket （SOCK_STREAM）和数据报式 Socket （SOCK_DGRAM）。流式是一种面向连接的 Socket，针对于面向连接的 TCP 服务应用；数据 报式 Socket 是一种无连接的 Socket，对应于无连接的 UDP 服务应用。

Socket 建立

为了建立 Socket，程序可以调用 Socket 函数，该函数返回一个类似于文件描述符的句柄。socket 函数原型为：

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

domain 指明所使用的协议族，通常为 PF_INET，表示互联网协议族（TCP/IP 协议族）；type 参数指定 socket 的类型： SOCK_STREAM 或 SOCK_DGRAM，Socket 接口还定义了原始 Socket（SOCK_RAW），允许程序使用低层协议；protocol 通常赋值"0"。 Socket() 调用返回一个整型 socket 描述符，你可以在后面的调用使用它。

Socket 描述符是一个指向内部数据结构的指针，它指向描述符表入口。调用 Socket 函数时，socket 执行体将建立一个 Socket，实际上"建立一个 Socket"意味着为一个 Socket 数据结构分配存储空间。Socket 执行体为你管理描述符表。

两个网络程序之间的一个网络连接包括五种信息：通信协议、本地协议地址、本地主机端口、远端主机地址和远端协议端口。Socket 数据结构中包含这五种信息。

Socket 配置

通过 socket 调用返回一个 socket 描述符后，在使用 socket 进行网络传输以前，必须配置该 socket。面向连接的 socket 客户端通过 调用 Connect 函数在 socket 数据结构中保存本地和远端信息。无连接 socket 的客户端和服务端以及面向连接 socket 的服务端通过调

用 bind 函数来配置本地信息。

Bind 函数将 socket 与本机上的一个端口相关联，随后你就可以在该端口监听服务请求。Bind 函数原型为：

int bind(int sockfd,struct sockaddr *my_addr, int addrlen);

Sockfd 是调用 socket 函数返回的 socket 描述符,my_addr 是一个指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针；addrlen 常被设置为 sizeof(struct sockaddr)。

struct sockaddr 结构类型是用来保存 socket 信息的：

```
struct sockaddr {
unsigned short sa_family; /* 地址族， AF_xxx */
char sa_data[14]; /* 14 字节的协议地址 */
};
```

sa_family 一般为 AF_INET，代表 Internet（TCP/IP）地址族；sa_data 则包含该 socket 的 IP 地址和端口号。

另外还有一种结构类型：

```
struct sockaddr_in {
short int sin_family; /* 地址族 */
unsigned short int sin_port; /* 端口号 */
struct in_addr sin_addr; /* IP 地址 */
unsigned char sin_zero[8]; /* 填充 0 以保持与 struct sockaddr 同样大小 */
};
```

这个结构更方便使用。sin_zero 用来将 sockaddr_in 结构填充到与 struct sockaddr 同样的长度，可以用 bzero()或 memset()函数将其置为零。指向 sockaddr_in 的指针和指向 sockaddr 的指针可以相互转换，这意味着如果一个函数所需参数类型是 sockaddr 时，你可以在函数调用的时候将一个指向 sockaddr_in 的指针转换为指向 sockaddr 的指针；或者相反。

使用 bind 函数时，可以用下面的赋值实现自动获得本机 IP 地址和随机获取一个没有被占用的端口号：

```
my_addr.sin_port = 0; /* 系统随机选择一个未被使用的端口号 */
my_addr.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */
```

通过将 my_addr.sin_port 置为 0，函数会自动为你选择一个未占用的端口来使用。同样，通过将 my_addr.sin_addr.s_addr 置为 INADDR_ANY，系统会自动填入本机 IP 地址。

注意在使用 bind 函数是需要将 sin_port 和 sin_addr 转换成为网络字节优先顺序；而 sin_addr 则不需要转换。

计算机数据存储有两种字节优先顺序：高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输，所以对于在内部是以低位字节优先方式存储数据的机器，

在 Internet 上传输数据时就需要进行转换，否则就会出现数据不一致。

下面是几个字节顺序转换函数：

·htonl()：把 32 位值从主机字节序转换成网络字节序

·htons()：把 16 位值从主机字节序转换成网络字节序

·ntohl()：把 32 位值从网络字节序转换成主机字节序

·ntohs()：把 16 位值从网络字节序转换成主机字节序

Bind()函数在成功被调用时返回 0；出现错误时返回"-1"并将 errno 置为相应的错误号。需要注意的是，在调用 bind 函数时一般不要将端口号置为小于 1024 的值，因为 1 到 1024 是保留端口号，你可以选择大于 1024 中的任何一个没有被占用的端口号。

连接建立

面向连接的客户程序使用 Connect 函数来配置 socket 并与远端服务器建立一个 TCP 连接，其函数原型为：

int connect(int sockfd, struct sockaddr *serv_addr,int addrlen);

Sockfd 是 socket 函数返回的 socket 描述符；serv_addr 是包含远端主机 IP 地址和端口号的指针；addrlen 是远端地质结构的长度。 Connect 函数在出现错误时返回-1，并且设置 errno 为相应的错误码。进行客户端程序设计无须调用 bind()，因为这种情况下只需知道目的机器 的 IP 地址，而客户通过哪个端口与服务器建立连接并不需要关心，socket 执行体为你的程序自动选择一个未被占用的端口，并通知你的程序数据什么时候到 打断口。

Connect 函数启动和远端主机的直接连接。只有面向连接的客户程序使用 socket 时才需要将此 socket 与远端主机相连。无连接协议从不建立直接连接。面向连接的服务器也从不启动一个连接，它只是被动的在协议端口监听客户的请求。

Listen 函数使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们。

int listen(int sockfd， int backlog);

Sockfd 是 Socket 系统调用返回的 socket 描述符；backlog 指定在请求队列中允许的最大请求数，进入的连接请求将在队列中等待 accept()它们（参考下文）。Backlog 对队列中等待 服务的请求的数目进行了限制，大多数系统缺省值为 20。如果一个服务请求到来时，输入队列已满，该 socket 将拒绝连接请求，客户将收到一个出错信息。

当出现错误时 listen 函数返回-1，并置相应的 errno 错误码。

accept()函数让服务器接收客户的连接请求。在建立好输入队列后，服务器就调用 accept 函数，然后睡眠并等待客户的连接请求。

int accept(int sockfd, void *addr, int *addrlen);

sockfd 是被监听的 socket 描述符，addr 通常是一个指向 sockaddr_in 变量的指针，该变量用来存放提出连接请求服务的主机的信息（某 台主机从某个端口发出该请求）；addrten 通常为一个指向值为 sizeof(struct sockaddr_in)的整型指针变量。出现错误时

accept 函数返回-1 并置相应的 errno 值。

首先，当 accept 函数监视的 socket 收到连接请求时，socket 执行体将建立一个新的 socket，执行体将这个新 socket 和请求连接进程的地址联系起来，收到服务请求的 初始 socket 仍可以继续在以前的 socket 上监听，同时可以在新的 socket 描述符上进行数据传输操作。

数据传输

Send()和 recv()这两个函数用于面向连接的 socket 上进行数据传输。

Send()函数原型为：

int send(int sockfd, const void *msg, int len, int flags);

Sockfd 是你想用来传输数据的 socket 描述符；msg 是一个指向要发送数据的指针；Len 是以字节为单位的数据的长度；flags 一般情况下置为 0（关于该参数的用法可参照 man 手册）。

Send()函数返回实际上发送出的字节数，可能会少于你希望发送的数据。在程序中应该将 send()的返回值与欲发送的字节数进行比较。当 send()返回值与 len 不匹配时，应该对这种情况进行处理。

```
char *msg = "Hello!";
int len, bytes_sent;
……
len = strlen(msg);
bytes_sent = send(sockfd, msg,len,0);
……
```

recv()函数原型为：

int recv(int sockfd,void *buf,int len,unsigned int flags);

Sockfd 是接受数据的 socket 描述符；buf 是存放接收数据的缓冲区；len 是缓冲的长度。Flags 也被置为 0。Recv()返回实际上接收的字节数，当出现错误时，返回-1 并置相应的 errno 值。

Sendto()和 recvfrom()用于在无连接的数据报 socket 方式下进行数据传输。由于本地 socket 并没有与远端机器建立连接，所以在发送数据时应指明目的地址。

sendto()函数原型为：

int sendto(int sockfd, const void *msg,int len,unsigned int flags,const struct sockaddr *to, int tolen);

该函数比 send()函数多了两个参数，to 表示目地机的 IP 地址和端口号信息，而 tolen 常常被赋值为 sizeof (struct sockaddr)。Sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1。

Recvfrom()函数原型为：

int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct sockaddr *from,int

*fromlen);

from 是一个 struct sockaddr 类型的变量，该变量保存源机的 IP 地址及端口号。fromlen 常置为 sizeof (struct sockaddr)。当 recvfrom()返回时，fromlen 包含实际存入 from 中的数据字节数。Recvfrom()函数返回接收到的字节数或 当出现错误时返回-1，并置相应的 errno。

如果你对数据报 socket 调用了 connect()函数时，你也可以利用 send()和 recv()进行数据传输，但该 socket 仍然是数据报 socket，并且利用传输层的 UDP 服务。但在发送或接收数据报时，内核会自动为之加上目地和源地址信息。

结束传输

当所有的数据操作结束以后，你可以调用 close()函数来释放该 socket，从而停止在该 socket 上的任何数据操作：

close(sockfd);

你也可以调用 shutdown()函数来关闭该 socket。该函数允许你只停止在某个方向上的数据传输，而一个方向上的数据传输继续进行。如你可以关闭某 socket 的写操作而允许继续在该 socket 上接受数据，直至读入所有数据。

int shutdown(int sockfd,int how);

Sockfd 是需要关闭的 socket 的描述符。参数 how 允许为 shutdown 操作选择以下几种方式：

·0-------不允许继续接收数据

·1-------不允许继续发送数据

·2-------不允许继续发送和接收数据，

·均为允许则调用 close ()

shutdown 在操作成功时返回 0，在出现错误时返回-1 并置相应 errno。

Socket 编程实例

代码实例中的服务器通过 socket 连接向客户端发送字符串"Hello, you are connected!"。只要在服务器上运行该服务器软件，在客户端运行客户软件，客户端就会收到该字符串。

该服务器软件代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
```

```
#include <sys/socket.h>
#include <sys/wait.h>
#define SERVPORT 3333 /*服务器监听端口号 */
#define BACKLOG 10 /* 最大同时连接请求数 */
main()
{
    int sockfd,client_fd; /*sock_fd：监听 socket；client_fd：数据传输 socket */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    struct sockaddr_in remote_addr; /* 客户端地址信息 */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
       perror("socket 创建出错！"); exit(1);
    }
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(SERVPORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero),8);
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
        perror("bind 出错！");
        exit(1);
    }
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen 出错！");
        exit(1);
    }
    while(1) {
       sin_size = sizeof(struct sockaddr_in);
       if ((client_fd = accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size)) == -1) {
            perror("accept 出错");
            continue;
       }
       printf("received a connection from %s\n", inet_ntoa(remote_addr.sin_addr));
       if (!fork()) { /* 子进程代码段 */
           if (send(client_fd, "Hello, you are connected!\n", 26, 0) == -1)
           perror("send 出错！");
           close(client_fd);
```

```
            exit(0);
        }
        close(client_fd);
        }
    }
}
```

服务器的工作流程是这样的：首先调用 socket 函数创建一个 Socket，然后调用 bind 函数将其与本机地址以及一个本地端口号绑定，然后调用 listen 在相应的 socket 上监听，当 accpet 接收到一个连接服务请求时，将生成一个新的 socket。服务器显示该客户机的 IP 地址，并通过 新的 socket 向客户端发送字符串"Hello，you are connected!"。最后关闭该 socket。

代码实例中的 fork()函数生成一个子进程来处理数据传输部分，fork()语句对于子进程返回的值为 0。所以包含 fork 函数的 if 语句是子进程代码部分，它与 if 语句后面的父进程代码部分是并发执行的。

客户端程序代码如下：

```
#include<stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define SERVPORT 3333
#define MAXDATASIZE 100 /*每次最大数据传输量 */
main(int argc, char *argv[]){
    int sockfd, recvbytes;
    char buf[MAXDATASIZE];
    struct hostent *host;
    struct sockaddr_in serv_addr;
    if (argc < 2) {
fprintf(stderr,"Please enter the server's hostname!\n");
exit(1);
}
    if((host=gethostbyname(argv[1]))==NULL) {
```

```
herror("gethostbyname 出错！ ");
exit(1);
}
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
perror("socket 创建出错！ ");
exit(1);
}
    serv_addr.sin_family=AF_INET;
    serv_addr.sin_port=htons(SERVPORT);
    serv_addr.sin_addr = *((struct in_addr *)host->h_addr);
    bzero(&(serv_addr.sin_zero),8);
    if (connect(sockfd, (struct sockaddr *)&serv_addr, \
       sizeof(struct sockaddr)) == -1) {
perror("connect 出错！ ");
exit(1);
}
    if ((recvbytes=recv(sockfd, buf, MAXDATASIZE, 0)) ==-1) {
perror("recv 出错！ ");
exit(1);
}
    buf[recvbytes] = '\0';
    printf("Received: %s",buf);
    close(sockfd);
}
```

客户端程序首先通过服务器域名获得服务器的 IP 地址，然后创建一个 socket，调用 connect 函数与服务器建立连接，连接成功之后接收从服务器发送过来的数据，最后关闭 socket。

函数 gethostbyname()是完成域名转换的。由于 IP 地址难以记忆和读写，所以为了方便，人们常常用域名来表示主机，这就需要进行域名和 IP 地址的转换。函数原型为：

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

函数返回为 hosten 的结构类型，它的定义如下：

```
struct hostent {
    char *h_name; /*  主机的官方域名  */
    char **h_aliases; /*  一个以 NULL 结尾的主机别名数组  */
    int h_addrtype; /*  返回的地址类型，在 Internet 环境下为 AF-INET */
```

```
int h_length; /*  地址的字节长度  */
char **h_addr_list; /*  一个以 0 结尾的数组，包含该主机的所有地址*/
};
#define h_addr h_addr_list[0] /*在 h-addr-list 中的第一个地址*/
```

当 gethostname()调用成功时，返回指向 struct hosten 的指针，当调用失败时返回-1。当调用 gethostbyname 时，你不能使用 perror()函数来输出错误信息，而应该使用 herror()函数来输出。

无连接的客户/服务器程序的在原理上和连接的客户/服务器是一样的，两者的区别在于无连接的客户/服务器中的客户一般不需要建立连接，而且在发送接收数据时，需要指定远端机的地址。

阻塞和非阻塞

阻塞函数在完成其指定的任务以前不允许程序调用另一个函数。例如，程序执行一个读数据的函数调用时，在此函数完成读操作以前将不会执行下一程序语句。当 服务器运行到 accept 语句时，而没有客户连接服务请求到来，服务器就会停止在 accept 语句上等待连接服务请求的到来。这种情况称为阻塞 （blocking）。而非阻塞操作则可以立即完成。比如，如果你希望服务器仅仅注意检查是否有客户在等待连接，有就接受连接，否则就继续做其他事情，则 可以通过将 Socket 设置为非阻塞方式来实现。非阻塞 socket 在没有客户在等待时就使 accept 调用立即返回。

```
#include <unistd.h>
#include <fcntl.h>
……
sockfd = socket(AF_INET,SOCK_STREAM,0);
fcntl(sockfd,F_SETFL,O_NONBLOCK)；
……
```

通过设置 socket 为非阻塞方式，可以实现"轮询"若干 Socket。当企图从一个没有数据等待处理的非阻塞 Socket 读入数据时，函数将立即返 回，返回值为-1，并置 errno 值为 EWOULDBLOCK。但是这种"轮询"会使 CPU 处于忙等待方式，从而降低性能，浪费系统资源。而调用 select()会有效地解决这个问题，它允许你把进程本身挂起来，而同时使系统内核监听所要求的一组文件描述符的任何活动，只要确认在任何被监控的文件 描述符上出现活动，select()调用将返回指示该文件描述符已准备好的信息，从而实现了为进程选出随机的变化，而不必由进程本身对输入进行测试而浪费 CPU 开销。Select 函数原型为:

```
int select(int numfds,fd_set *readfds,fd_set *writefds，
fd_set *exceptfds,struct timeval *timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select()监视的读、写和异常处理的文件

描述符集合。如果你希望确定是否可以 从标准输入和某个 socket 描述符读取数据，你只需要将标准输入的文件描述符 0 和相应的 sockdtfd 加入到 readfds 集合中；numfds 的值 是需要检查的号码最高的文件描述符加 1，这个例子中 numfds 的值应为 sockfd+1；当 select 返回时，readfds 将被修改，指示某个文件 描述符已经准备被读取，你可以通过 FD_ISSSET() 来测试。为了实现 fd_set 中对应的文件描述符的设置、复位和测试，它提供了一组宏：

FD_ZERO(fd_set *set)----清除一个文件描述符集；

FD_SET(int fd,fd_set *set)----将一个文件描述符加入文件描述符集中；

FD_CLR(int fd,fd_set *set)----将一个文件描述符从文件描述符集中清除；

FD_ISSET(int fd,fd_set *set)----试判断是否文件描述符被置位。

Timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 长时间后没有文件描述符准备好即返回。struct timeval 数据结构为：

```
struct timeval {
int tv_sec; /* seconds */
int tv_usec; /* microseconds */
};
```

POP3 客户端实例

下面的代码实例基于 POP3 的客户协议，与邮件服务器连接并取回指定用户帐号的邮件。与邮件服务器交互的命令存储在字符串数组 POPMessage 中，程序通过一个 do-while 循环依次发送这些命令。

```
#include<stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define POP3SERVPORT 110
#define MAXDATASIZE 4096

main(int argc, char *argv[]){
int sockfd;
struct hostent *host;
struct sockaddr_in serv_addr;
char *POPMessage[]={
```

```
"USER userid\r\n",
"PASS password\r\n",
"STAT\r\n",
"LIST\r\n",
"RETR 1\r\n",
"DELE 1\r\n",
"QUIT\r\n",
NULL
};
int iLength;
int iMsg=0;
int iEnd=0;
char buf[MAXDATASIZE];

if((host=gethostbyname("your.server"))==NULL) {
perror("gethostbyname error");
exit(1);
}
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
perror("socket error");
exit(1);
}
serv_addr.sin_family=AF_INET;
serv_addr.sin_port=htons(POP3SERVPORT);
serv_addr.sin_addr = *((struct in_addr *)host->h_addr);
bzero(&(serv_addr.sin_zero),8);
if (connect(sockfd, (struct sockaddr *)&serv_addr,sizeof(struct sockaddr))==-1){
perror("connect error");
exit(1);
}

do {
send(sockfd,POPMessage[iMsg],strlen(POPMessage[iMsg]),0);
printf("have sent: %s",POPMessage[iMsg]);
```

```
iLength=recv(sockfd,buf+iEnd,sizeof(buf)-iEnd,0);
iEnd+=iLength;
buf[iEnd]='\0';
printf("received: %s,%d\n",buf,iMsg);

iMsg++;
} while (POPMessage[iMsg]);

close(sockfd);
}
```