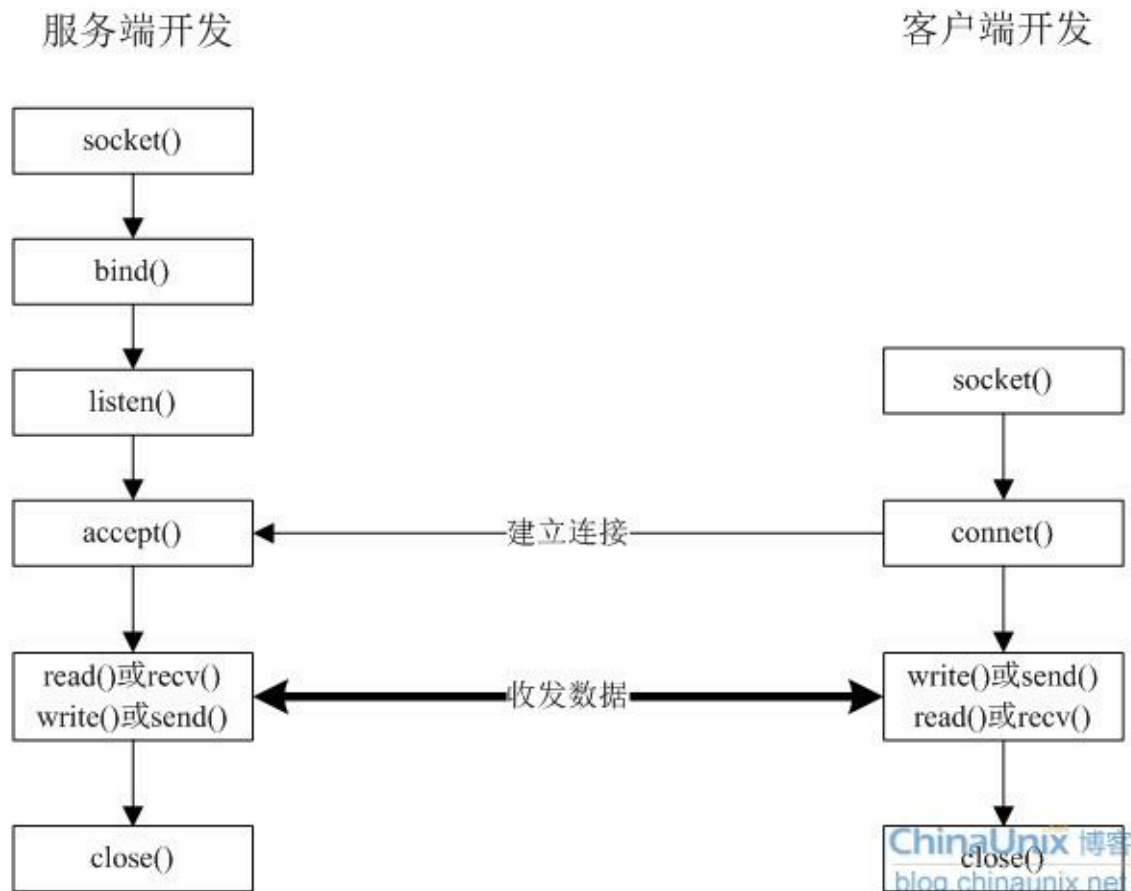


Linux网络编程：基于TCP的程序开发

星期四, 8月 23 2012, 10:45 下午

面向连接的TCP程序设计

基于TCP的程序开发分为服务器端和客户端两部分，常见的核心步骤和流程：



其实按照上面这个流程调用系统API确实可以完全实现应用层程序的开发，一点问题没有。可随着时间的推移，你会觉得这样子的开发毫无激情。为什么TCP的开发就要按照这样的流程来呢？而且一般出的问题几乎都不在这几个系统调用上，原因何在？当我们弄清网络开发的本质，协议栈的设计原理、数据流向等这些问题的答案也就会慢慢浮出水面了。接下来这几篇博文主要是围绕网络编程展开，目的是引出后面对于Linux下TCP/IP协议栈的相关分析做铺垫。

1、 创建socket

到目前为止我们知道socket的作用其实是非常大的，它不仅能够实现不同远端主机间的数据通信，还是可以实现不同进程间的通信，以及用户空间和内核空间的通信等等。其函数原型定义如下：

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

domain: 大家习惯性将其翻译成“协议域”或者“地址簇”，牵扯到“协议”这个词估计很多人就又晕了。也难怪，TCP、UDP我们叫协议，IP我们也叫协议，到处都是协议。那么到底什么是协议？说白了，协议其实就是事先规定好的数据通信方式。例如我们经常说的TCP/IP协议默认情况下都指的是IPv4版本协议，现在还有IPv6，它属于另外一种协议，还有在电信网里面经常听到的X.25协议，用户空间和内核空间通信的netlink协议等等。每种协议都有其特定的应用场景。这里，我们可以通俗的将domain理解成“协议种类”。

type: 直译就是“类型”的意思。因为不同种类的协议提供了不同的数据传输方式，我们常见的有面向连接的流式传输模式、无连接的数据报传输方式等。

protocol: 才是我们具体的协议类型，如TCP、UDP之类的。怎么理解？我们知道在IPv4协议族里，面向连接的流式套接字一般指的都是TCP协议。在电信网中NO.7信令体系中，实现了面向连接的流式套接字的协议就有ISUP和TUP协议。为什么平时一提到面向流式的套接字大家条件反射的就想起了TCP，那是因为我们心里已经默认了前提是IPv4协议族了。一般情况下，对于某种具体的套接字类型(流式或数据报式)都只有一种对应的实现协议，当然你心里必须要知道对于某些协议族有可能有多种协议。对于我们的IPv4而言，在创建socket套接字时，当指定了地址簇、套接字类型、protocol协议字段一般都设置为0。

关于socket()函数我们再打个比方：

假如说socket()函数就是个造人的机器，我们给它输入三个指令：

OK，当我们输入：

socket(黑种人，会说话，英文) => 马丁路德；

socket(黑种人，会说话，英文) => 威尔史密斯；

因为黑人缺省情况下说的话都是英文，所以他们会说话的功能也就仅局限英文了，不会有二义性。

可我们高贵的黄种人就不一样了(JP不在此范畴)，所以会说话的黄种人就多了去了。所以如果我们要造一个会说话的黄种人，必须进一步限定说话的语言才行。

socket(黄种人，会说话，中文) => 孔子；

socket(黄种人，会说话，朝鲜语) => 金正日；

通过这个例子，大家就可以好好体会一下socket()函数三个参数的作用和意义。

2、 绑定地址

它的主要作用是将由socket()函数创建socket文件描述符和一个本地地址结构体绑定起来。本地地址结构体一般都指明了我们这个socket文件描述符所属的协议簇，如果本地是多网卡多IP地址的话，可以在指定与哪个地址进行绑定，如果是任

意IP地址一般将地址字段设置为htonl(INADDR_ANY)，以及所使用的端口。函数原型如下：

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

第二个参数my_addr的类型的定义如下：

```
struct sockaddr{
    unsigned short  as_family;
    char           sa_data[14];
};
```

我们在Internet编程中一般不用这个结构，而是用：

```
struct sockaddr_in{
    unsigned short    sin_family;
    unsigned short int sin_port;
    struct in_addr    sin_addr;
    unsigned char     sin_zero[8];
}
```

在调用bind()函数时将一个sockaddr_in{}类型的对象强制转换成sockaddr{}类型赋给bind()函数的第二个参数。

第三个参数addrlen指的是第二个参数在sockaddr{}类型下的实际长度。

3、 系统监听

一般在基于流式的套接字编程中服务器端的开发需要执行这一步。为什么？以后解释。这一步我们主要调用的API函数是：

```
int listen(int sockfd, int backlog);
```

该函数的主要作用就是将sockfd变成被动的连接的监听套接字。backlog指明那些已经经过了TCP三次握手的处于established状态的连接项在被系统调度前的最大排队等候数。

4、 主动连接

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

该函数是客户端主动连接服务器的API调用。流式套接字可以调用，面向无连接的套接字也可以使用。但在实际应用中，后者调用该函数的情形还是比较少。

对于面向连接的套接字，当调用该函数时，客户端会向服务端发起一个SYN的连接请求，对于TCP来说此时就开始了3次握手的流程。如果客户端设置为阻塞模式，那么connect()函数会一直阻塞，直到3次握手成功或超时失败才返回；

对于无连接的套接字，调用该函数只是说明了客户端默认情况下发送和接受的

数据报都来自serv_addr地址，仅此而已。

5、 接受连接

当客户端主动用connect()去连接服务器时，针对于TCP这样的流式套接字，此时便后触发其3次握手的流程。当然3次握手成功的速度依赖于网络的健康状况和终端的处理性能。当服务器端收到客户端发来的第一个SYN报文，该连接就为半连接状态，处于半连接队列中；对于那些完成了3次握手的连接，TCP会将其从半连接队列移到一个名为established状态的已连接队列里，在accept()函数返回之前该连接暂时是不可用的：

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

该函数会从处于established状态的连接队列头部取出一个连接，然后生成一个新的socket连接，并返回新的socket文件描述。其中第二个参数addr中存储了来自客户端的一些地址信息，第三个参数addrlen指明了第二个参数所占的字节数。该函数目前也仅用在基于流式的套接字编程里

至此，前面介过listen()函数中backlog的意思现在我们应该已经很明确了。

6、 收发数据

收数据：recv、recvfrom、recvmsg和read

对于recvfrom 和 recvmsg，可同时应用于面向连接的和无连接的套接字。recv一般只用在面向连接的套接字。只要将recvfrom的第五个参数设置NULL，几乎等同于recvfrom，。如果消息太大，无法完整存放在所提供的缓冲区时会根据不同的套接字，多余的字节会丢弃。假如套接字上没有消息可以读取，除非套接字已被设置为非阻塞模式，否则接收调用会等待消息的到来。

发数据：send、sendto、sendmsg和write

send只可用于基于连接的套接字，send 和 write唯一的不同点是标志的存在，当标志为0时，send等同于write。sendto 和 sendmsg既可用于无连接的套接字，也可用于基于连接的套接字。除非套接字设置为非阻塞模式，否则调用将会阻塞直到数据被发送完。

也就是说是在TCP编程中，收发数据一般用得最多的就是recv和send或者read和write。

不管怎么说，记住一点就行：recv和send函数提供了和read和write差不多的功能，不过它们提供了第四个参数来控制读写操作。

这里我们仅讨论一下read和write这种更通用的IO操作：

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

这两个函数分别由fd所指定的socket套接口发送(接收)count个字节的数据，然后将其存储在buf变量所指向的缓冲区里。这两个函数均返回实际成功发送(接受)的字节数。

7、 关闭连接

int close(int fd);

调用该函数时对于TCP编程而言会触发一个FIN报文导致连接关闭。

还是看个小小的例子，热热身：

TCP服务器端的开发

点击[\(此处\)](#)折叠或打开

```
1.  //TCP示例服务器端 tcpSrv.c
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.  #include <errno.h>
5.  #include <string.h>
6.  #include <unistd.h>
7.  #include <netdb.h>
8.  #include <sys/socket.h>
9.  #include <netinet/in.h>
10. #include <sys/types.h>
11. #include <arpa/inet.h>
12. int main(int argc, char *argv[])
13. {
14.     int skfd,cnfd,addr_len;
15.     struct sockaddr_in srv_addr,clt_addr;
16.     int portnumber;
17.     char hello[]="Hello! Long time no see.\n";
18.     if(2 != argc || 0 > (portnumber=atoi(argv[1])))
19.     {
20.         printf("Usage:%s port\n",argv[0]);
21.         exit(1);
22.     }
23.
24.     /* 创建IPv4的流式套接字描述符 */
25.     if(-1 == (skfd=socket(AF_INET,SOCK_STREAM,0)))
26.     {
27.         perror("Socket Error:");
28.         exit(1);
29.     }
30.
31.     /* 填充服务器端sockaddr地址结构 */
32.     bzero(&srv_addr,sizeof(struct sockaddr_in));
33.     srv_addr.sin_family=AF_INET;
```

```

34.     srv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
35.     srv_addr.sin_port=htons(portnumber);
36.
37.     /* 将套接字描述符skfd和地址信息结构体绑定起来 */
38.     if(-1 == bind(skfd,(struct sockaddr *)&srv_addr,sizeof(struct sockaddr)))
39.     {
40.         perror("Bind error:");
41.         exit(1);
42.     }
43.
44.     /* 将skfd转换为被动建通模式 */
45.     if(-1 == listen(skfd,4))
46.     {
47.         perror("Listen error:");
48.         exit(1);
49.     }
50.
51.     while(1)
52.     {
53.         /* 调用accept,服务器端一直阻塞,直到客户程序与其建立连接成功为止*/
54.         addr_len=sizeof(struct sockaddr_in);
55.         if(-1 == (cnfd=accept(skfd,(struct sockaddr *)&clt_addr,&addr_len)))
56.         {
57.             perror("Accept error:");
58.             exit(1);
59.         }
60.         printf("Connect from %s:%u ...!\n",inet_ntoa(clt_addr.sin_addr),ntohs(clt_addr.sin_port));
61.         if(-1 == write(cnfd,hello,strlen(hello))){
62.             perror("Send error:");
63.             exit(1);
64.         }
65.         close(cnfd);
66.     }
67.     close(skfd);
68.     exit(0);
69. }

```

TCP客户端的开发如下

点击[\(此处\)](#)折叠或打开

```

1. //TCP示例客户端 tcpclt.c
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <errno.h>
5. #include <string.h>

```

```
6.  #include <unistd.h>
7.  #include <fcntl.h>
8.  #include <netdb.h>
9.  #include <sys/socket.h>
10. #include <netinet/in.h>
11. #include <sys/types.h>
12. #include <arpa/inet.h>
13. int main(int argc, char *argv[])
14. {
15.     int skfd;
16.     char buf[1024] = {0};
17.     struct sockaddr_in server_addr;
18.     struct hostent *host;
19.     int portnumber, nbytes;
20.     if(3 != argc || 0 > (portnumber = atoi(argv[2])))
21.     {
22.         printf("Usage: %s hostname portnumber \n");
23.         exit(1);
24.     }
25.     if(NULL == (host = gethostbyname(argv[1])))
26.     {
27.         perror("Gethostname error:");
28.         exit(1);
29.     }
30.
31.     /* 创建socket描述符 */
32.     if(-1 == (skfd = socket(AF_INET, SOCK_STREAM, 0)))
33.     {
34.         perror("Socket Error:");
35.         exit(1);
36.     }
37.
38.     /* 客户端填充需要连接的服务器的地址信息结构体 */
39.     bzero(&server_addr, sizeof(server_addr));
40.     server_addr.sin_family = AF_INET;
41.     server_addr.sin_port = htons(portnumber);
42.     server_addr.sin_addr = *((struct in_addr *) host->h_addr);
43.
44.     /* 客户端调用connect主动发起连接请求 */
45.     if(-1 == connect(skfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)))
46.     {
47.         perror("Connect Error:");
48.         exit(1);
49.     }
```

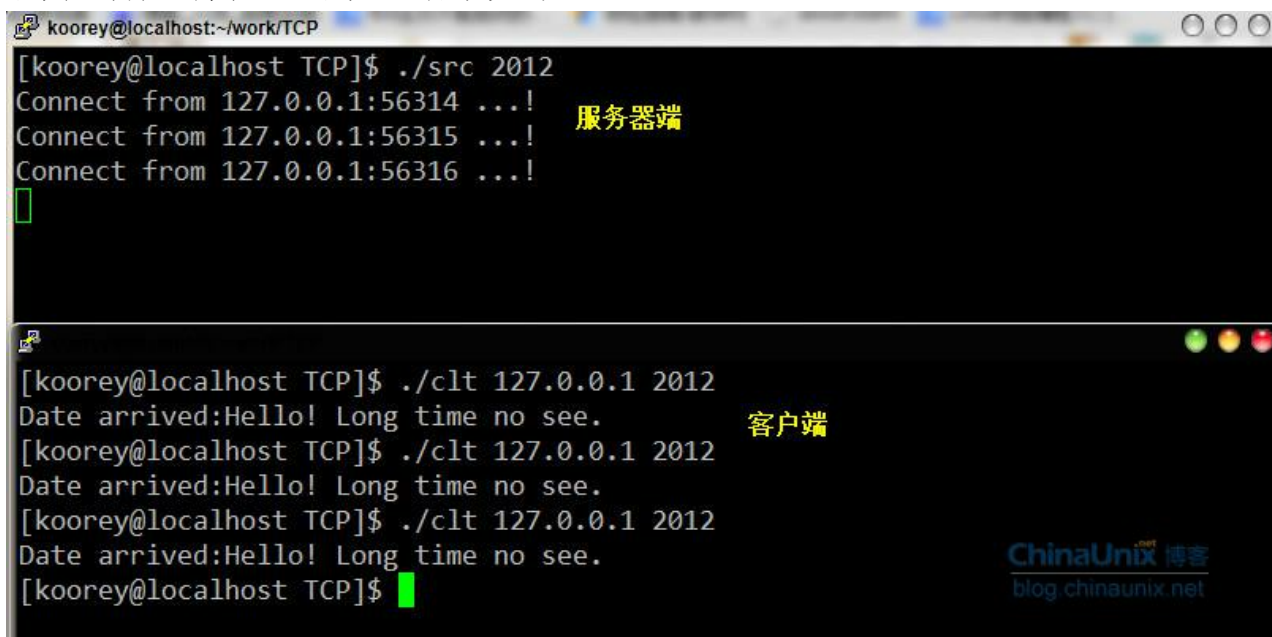


```

50.
51.     /*客户端只接收服务器发来的数据，然后就退出*/
52.     if(-1 == read(skfd,buf,1024)){
53.         perror("Recv Error:");
54.     }
55.     printf("Date arrived:%s",buf);
56.
57.     /* 拆除TCP连接 */
58.     close(skfd);
59.     exit(0);
60. }

```

以上只是我们对TCP网路程序开发流程的一个简单回顾，主要目的不是为了写出多漂亮的代码，所以没考虑多进程并发访问。只是简单的当客户端和服务端建立TCP连接后，服务器打印出客户端的地址信息“IP+PORT”，然后服务器向客户端发送一个“Hello! Long time no see.”简单数据，而客户端也仅一个接收函数，将其打印出来出来后客户端就退出了。结果如下：



```

koorey@localhost:~/work/TCP
[koorey@localhost TCP]$ ./src 2012
Connect from 127.0.0.1:56314 ...!
Connect from 127.0.0.1:56315 ...!
Connect from 127.0.0.1:56316 ...!
█

koorey@localhost TCP$ ./clt 127.0.0.1 2012
Date arrived:Hello! Long time no see.
[koorey@localhost TCP]$ ./clt 127.0.0.1 2012
Date arrived:Hello! Long time no see.
[koorey@localhost TCP]$ ./clt 127.0.0.1 2012
Date arrived:Hello! Long time no see.
[koorey@localhost TCP]$ █

```

今天主要简单回顾了一下基于TCP的程序开发流程而已。