附加：进程通信-实验2：文件套接字通信

一．实验目的

·理解并掌握本地文件套接字的使用方法

二．实验背景

·什么是套接字（socket）

·socket可以看作进程访问系统网络组件的接口，它有相应的一块内存，其中存放了它的各种属性。进程对socket的各种操作将转换为对网络组件的操作，从而通过网络收发数据。当属于不同进程的两个套接字之间建立了一个连接，那么，这两个进程就可以通过这一对套接字进行通信了。一个进程可以创建多个套接字，分别用于不同的通信目的。一个通信连接关联一对且只能是一对套接字。

·套接字不仅可用于本地通信，更可以用于网络通信

·客户/服务器模型（Client/Server）

在进行通信的两个进程中，主动发起通信请求的一方称为客户，被动响应的一方称为服务器。它们既可以是处在同一台计算机上的两个进程，也可以分别处于网络环境下的不同主机上。这种通信模型叫作客户/服务器模型，即Client/Server模型（简称C/S模型），是所有网络应用的基础。

·服务器实现

第1步：创建套接字

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

socket函数创建一个套接字，并返回一个套接字描述符，用于将来访问该套接字。

domain参数是套接字的域（协议族），最常用的域是AF\_UNIX和AF\_INET，前者用于通过Linux文件系统实现本地套接字，后者用于实现网络套接字。

type指定套接字类型，决定了套接字所采用的通信机制。有两种常见类型：流套接字和数据报套接字。

protocol指定通信所用的协议，一般由套接字域和类型来决定，一般将其设为0，表示使用默认协议。

第2步：为套接字命名

int bind(int socket, const struct sockaddr \*address, size\_t address\_len);

所谓命名（naming），其实就是将套接字绑定（binding）到一个特定的地址。对于AF\_UNIX套接字，就是将套接字关联到文件系统的一个路径名，而对于AF\_INET套接字是关联到一个IP端口号。

第3步：监听连接

int listen(int socket, int backlog);

Listen函数在服务套接字上监听客户端连接，它会创建一个队列来缓存未处理的连接；

其中，socket是服务套接字的标识符。backlog为连接队列的最大长度。

第4步：接受连接

int accept(int socket, struct sockaddr \*address, size\_t \*address\_len);

accept函数会创建一个新套接字来与所接受的客户进行通信，并返回新套接字的描述符。

·客户端实现

第1步：创建无名套接字

int sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

这里创建一个客户端套接字，对客户端套接字来说是无需命名的。

第2步：请求连接服务器

int connect(int socket, const struct sockaddr \*address, size\_t address\_len);

该函数在一个未命名的客户套接字和服务器套接字之间建立一个连接。

第3步：数据通信

连接一旦建立起来，就可以用连接所关联的一对套接字进行双向数据通信了,方法是向套接字读写数据。例如：

int a=100,b=200,c=0;

write(sock, &a, sizeof(int));

write(sock, &b, sizeof(int));

read(sock, &c, sizeof(int));

第4步：关闭套接字

int close(int socket);

三．关键代码及分析

/\* 创建套节字 \*/

unlink("server\_socket");

int server\_sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

/\* 为套节字命名 \*/

struct sockaddr\_un server\_address;

server\_address.sun\_family = AF\_UNIX;

strcpy(server\_address.sun\_path, "server\_socket");

int server\_len = sizeof(server\_address);

bind(server\_sock, (struct sockaddr \*)&server\_address, server\_len);

/\* 监听客户的连接请求（创建一个连接队列，等待客户的连接） \*/

listen(server\_sock, 5);

/\* 接受连接 \*/

int client\_sock, client\_len;

struct sockaddr\_un client\_address;

int a,b,c;

while(1) {

printf("server waiting...\n");

/\* 接受一个连接 \*/

client\_len = sizeof(client\_address);

client\_sock = accept(server\_sock, (struct sockaddr \*)&client\_address, &client\_len);

/\* 通过对client\_sock套节字的读写操作与客户进行通信 \*/

read(client\_sock, &a, sizeof(int));

read(client\_sock, &b, sizeof(int));

c=a+b;

write(client\_sock, &c, sizeof(int));

close(client\_sock);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

/\* 创建套节字 \*/

int sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

/\* 将该套节字连接到服务器套节字 \*/

struct sockaddr\_un address;

address.sun\_family = AF\_UNIX;

strcpy(address.sun\_path, "server\_socket");

int len = sizeof(address);

int result = connect(sock, (struct sockaddr \*)&address, len);

if(result == -1) {

perror("连接服务器失败");

exit(1);

}

/\* 通过该套节字读写数据 \*/

int a=100,b=200,c=0;

write(sock, &a, sizeof(int));

write(sock, &b, sizeof(int));

read(sock, &c, sizeof(int));

printf("来自服务器的数据为： %d\n", c);

/\* 关闭套接字 \*/

close(sock);

exit(0);

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

四．实验结果与分析

· //编译连接生成可执行程序

gcc -o sockFileServer sockFileServer.c

gcc -o sockFileClient sockFileClient.c

· //执行生产的可执行程序

./sockFileServer &

./sockFileClient

————————————————

版权声明：本文为CSDN博主「\_Hello Spring」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/wct3344142/article/details/104134889