基于 Linux 系统的局域网模块化通信服务

一、实验目的：

1. 掌握 Linux 系统下的网络套接字编程方法及其规范

2. 掌握 Linux 系统下并发服务器的编程方法

3. 掌握 Linux 系统下模块化服务的框架搭建

4. 掌握 Linux 系统下建立多进程之间共享的缓存机制

二、程序功能：

服务器端：

初始化服务后等待客户连接，连接成功后向客户发送欢迎信息；接着接收该客户的昵称并显示；其次，服务器向客户端发送当前所提供的服务列表由客户端进行选择，服务器接收客户端的模式选择索引，服务器端自动载入相关的服务模块。

此实验的服务器向客户端提供了字符串的逆转服务和群聊服务。

如果客户端选择了字符串的逆转回显服务，则服务器接收来自客户的信息（字符串），将该字符串反转，并将结果送回客户。

如果客户端选择群聊服务，则服务器接收来自客户端发送的消息，然后检索已连接的群聊客户，并转发给所有的群聊用户。

并且要求服务器具有同时处理多个客户的能力，且选择不同功能服务之间用户互不干涉。

客户端：

客户端首先与服务器连接，接收用户输入客户的名字，将该名字发送给服务器；接着接收服务器发送过来的服务列表，接收用户输入服务选择索引，并发送给服务器。

当用户输入一行字符串时，自动接收字符串，并发送给服务器；

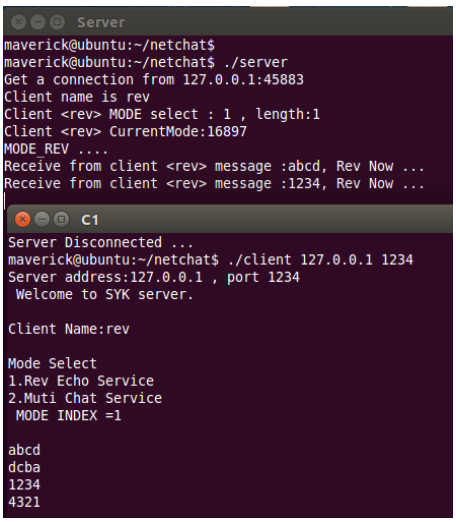
当服务器向当前客户端发送消息时，接收，并显示之。

当用户输入 Ctrl+C 时终止连接并退出。

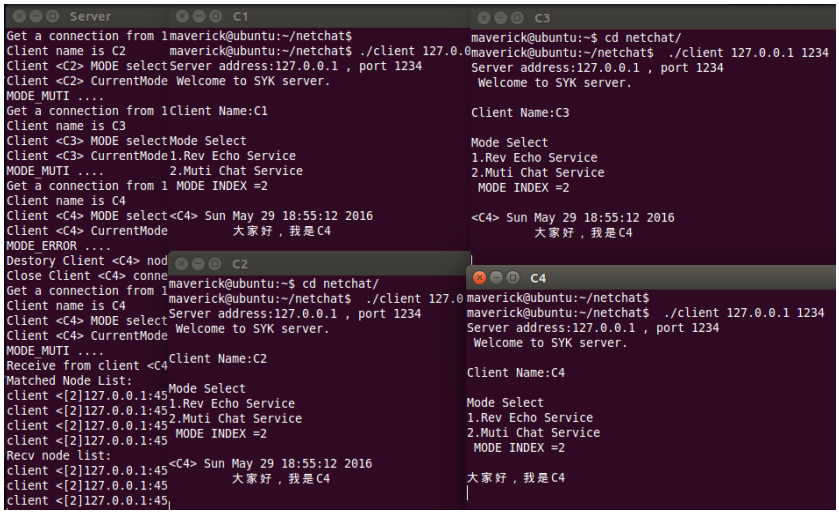
要求客户端程序具有通用性，即服务器端增删服务模块，客户端程序不需要更换。

三、程序运行截图：

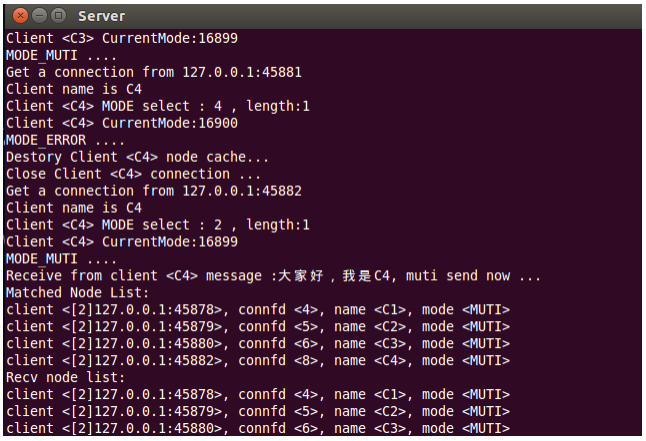
客户端使用字符串逆转服务：



客户端使用群聊服务：



服务器端与客户端交互时的 log 输出：

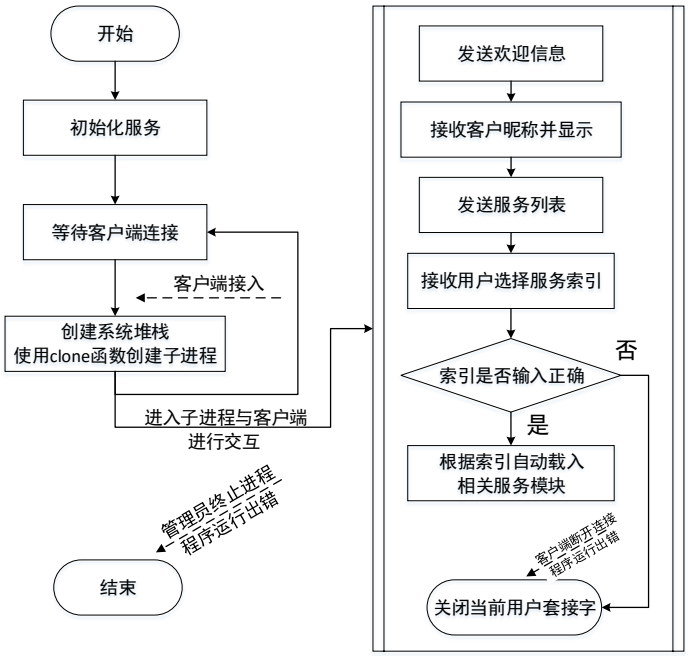


四、具体实现

由于此次实验最主要实现的功能就是模块化的通信服务，所以服务器端需要具有模块化的服务的能力，且客户端需要具有通用性。即服务的更新，客户端程序无需更新。

首先，大体介绍下服务器端的整体处理流程(如下图所示)。

4.1 服务器端的具体实现



首先，服务器端先进行对外开放服务的网络配置进行初始化，调用 socket 函数指明套接字类型，创建套接字并生成监听套接字描述符，该套接字描述符负责接收客户端请求；随后使用该套接字描述符调用 bind 函数进行对外提供服务的地址及端口号进行绑定，随后使用该套接字描述符调用 listen 函数对该套接字进行连接队列数量的限定，其核心初始化代码如下图所示。

```
if((listenfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
    perror("Create socket failed.");
    exit(-1);
}

setsockopt(listenfd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));

bzero(&server,sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(PORT);
server.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(listenfd,(struct sockaddr*)&server,sizeof(server)) == -1){
    perror("Bind error()");
    exit(-1);
}

if(listen(listenfd,BACKLOG) == -1){
    perror("listen error.");
    exit(-1);
}
```

随后，服务器端程序就调用 accept 函数来等待客户端的连接。客户端连接之前，调用 accept 函数会使得服务器主进程一直处于阻塞状态，直至有客户端的接入。如果成功建立连接，则将重新返回一个新的套接字描述符，该套接字描述符的作用是与已建立连接的客户端进行交互使用。并且将客户端的 IP 地址端口号等信息存入已传入的网际套接字地址结

构中。

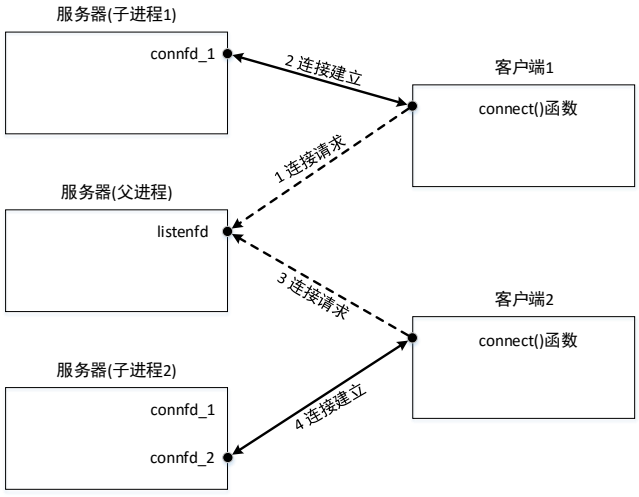
其次，服务器端程序调用 bind 函数创建新的子进程，将刚才 accept 函数返回的已连接套接字描述符和存储对端地址信息的网际套接字地址结构传入子进程，由子进程负责与客户端进行接下来的交互工作。父进程则继续执行 accept 函数使得服务器端程序主进程处于阻塞状态，等待新的客户端与服务器建立连接。

接收客户端连接与建立子进程与客户端进行交互的大体的核心代码如下图所示。

```
while(1){

    if((connectfd = accept(listenfd,(struct sockaddr*)&client,&addr_len)) == -1){
        perror("Accept error.");
        exit(-1);
    }

    bzero(&client_temp,sizeof(client_temp));
    connfd_temp = connectfd;
    client_temp.sin_family = client.sin_family;
    client_temp.sin_port = client.sin_port;
    client_temp.sin_addr.s_addr = client.sin_addr.s_addr;

    void ** stack;
    stack = (void **)malloc(STACK_SIZE);//为子进程申请系统堆栈
    if(!stack) {
            printf("Create stack failed\n");
            exit(0);
    }

    //创建子进程
    clone(&process_client, (char *)stack + STACK_SIZE, CLONE_VM|CLONE_FILES|CLONE_SIGHAND, 0);

}
```

其中，程序中创建子进程的时使用的是 clone 函数而非 fork 函数，其最主要的原因在于 fork 函数在管理父子进程资源的时候是基于写时拷贝的，即子进程中使用的资源是父进程调用 fork 函数之前的所有资源状态的副本，并且父子进程使用的资源互不干涉。子进程更改的只是父进程中的资源副本，子进程结束后子进程更改的资源在父进程中并不会改变。

又因为之前在功能描述中介绍到此服务器需要向客户端提供群聊服务，倘若创建子进程与服务器交互使用的是 fork 函数，即有可能出现先连接的客户端群发的消息对于后连接的客户端接收不到的问题（如下图所示）。



客户端 1 向服务器提交连接请求，服务器主进程中调用 accept 函数接收客户端 1 的请求，函数返回一个新的套接字描述符 connfd_1，用来与客户端 1 进行交互。

此时服务器主进程中可使用的资源除了监听套接字描述符 listenfd，还有一个与客户端 1 交互的套接字描述符 connfd_1，随后调用 fork 函数创建子进程专门用来与客户端 1 进行交互。至此，客户端与服务器连接建立，服务器正式与客户端 1 进行交互。子进程 1

除了与客户端 1 进行交互外，不能与其他客户端进行任何交互操作（如群发操作）。

随后，客户端 2 向服务器提交连接请求，服务器主进程中调用 accept 函数接收客户端 2 的请求，函数返回一个新的套接字描述符 connfd_2，用来与客户端 2 进行交互。

此时服务器主进程中可使用的资源除了监听套接字描述符 listenfd，一个与客户端 1 交互的套接字描述符 connfd_1，还有一个与客户端 2 进行交互的套接字描述符 connfd_2，随后调用了 fork 创建子进程专门用来与客户端 2 进行交互。至此，客户端与服务器连接建立，服务器正式与客户端 2 进行交互。子进程 2 除了可以与客户端 2 进行交互外，还可以与客户端 1 进行交互。

简而言之，如果客户端 1 先连接服务器，客户端 2 后连接服务器，调用 fork 函数创建子进程与客户端进行交互的话。客户端 2 群发的消息客户端 1 可以接收到，而客户端 1 群发消息客户端 2 接收不到。所以此路不通，需要另寻蹊径，父子进程使用的资源需要是实时同步的，且需要父子进程执行时各不干涉（即子进程执行时，父进程不能阻塞）。

实际上 Linux 下的 ANSI C 的函数库给我们提供了这样的函数，那就是 clone。fork()函数复制时将父进程的所有资源都通过复制数据结构进行了复制，然后传递给子进程，所以 fork()函数不带参数；clone()函数则是将部分父进程的资源的数据结构进行复制，复制哪些资源是可选择的，这个可以通过参数设定，所以 clone()函数带参数，没有复制的资源可以通过指针共享给子进程。clone()函数的声明如下：

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

这里 fn 是函数指针，我们知道进程的 4 要素，这个就是指向程序的指针，即想要创建进程的静态程序,child_stack 明显是为子进程分配系统堆栈空间（在 linux 下系统堆栈空间是 2 页面，就是 8K 的内存，其中在这块内存中，低地址上放入了值，这个值就是进程控制块 task_struct 的值）,flags 就是标志用来描述你需要从父进程继承那些资源，arg 就是传给子进程的参数)。下面是 flags 可以取的值：

标志	含义
CLONE_PARENT	创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
CLONE_FS	子进程与父进程共享相同的文件系统，包括 root、当前目录、umask
CLONE_FILES	子进程与父进程共享相同的文件描述符（file descriptor）表
CLONE_NEWNS	在新的 namespace 启动子进程，namespace 描述了进程的文件 hierarchy
CLONE_SIGHAND	子进程与父进程共享相同的信号处理（signal handler）表
CLONE_PTRACE	若父进程被 trace，子进程也被 trace
CLONE_VFORK	父进程被挂起，直至子进程释放虚拟内存资源
CLONE_VM	子进程与父进程运行于相同的内存空间
CLONE_PID	子进程在创建时 PID 与父进程一致
CLONE_THREAD	Linux 2.4 中增加以支持 POSIX 线程标准，子进程与父进程共享相同的线程群

根据之前的分析，如果要实现群发，父子进程使用的资源需要“实时刷新”，即父子进程需要使用相同的内存空间，需要有相同的文件描述符，且需要有相同的信号量处理。

所以第三个参数需要标明调用 CLONE_VM、CLONE_FILES 和 CLONE_SIGHAND，调用多个功能项目的时候只需要将多个 flag 按位或隔开即可（这一点和子网掩码的概念相同），即第三个标志位是 CLONE_VM|CLONE_FILES|CLONE_SIGHAND。

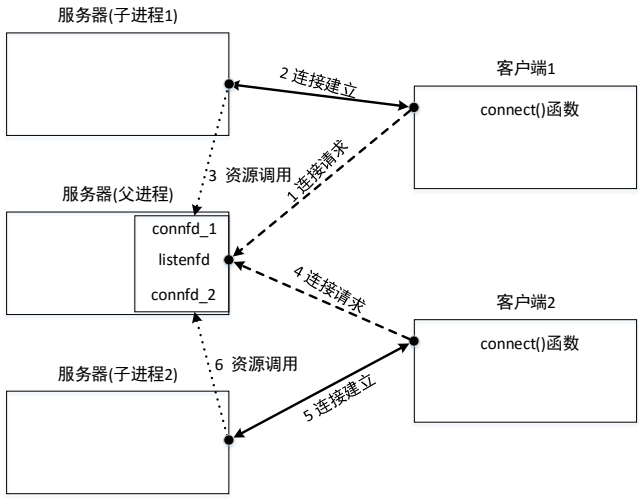
首先需要给新的进程创建个系统堆栈，这里我们使用 malloc 函数，堆栈大小为 8K，具体的代码为：

```
void ** stack;
stack = (void **)malloc(STACK_SIZE);//为子进程申请系统堆栈
if(!stack) {
     printf("Create stack failed\n");
     exit(0);
}
```

随后调用 clone 函数创建子进程，第一个参数填写子进程的处理函数，第二个参数填写系统堆栈的起始地址和堆栈大小，第三个参数填写之前确定的标志位的按位或，第四个参数默认填写 0，具体代码为：

```
//创建子进程
clone(&process_client, (char *)stack + STACK_SIZE, CLONE_VM|CLONE_FILES|CLONE_SIGHAND, 0);
```

随后在子进程中就可以使用与父进程同样的资源，客户端 1 先接入，客户端 2 再接入，子进程 1 中的资源是与父进程使用的同一份，并没有复制，实时更新的（如下图所示）。所以此时客户端 1 群发消息客户端 2 即可收到。



至此，服务器端的主进程介绍完毕，下面具体介绍子进程如何与客户端进行交互。

服务器端子进程进入后，子进程首先将需要用到的参数初始化，存放到自己的进程中私有化，然后打印客户端的连接信息，接着向客户端发送欢迎信息；然后接收客户端传来的昵称；接着服务器向客户端发送模式选择的字符串，由用户输入模式的索引标号，服务器接收索引。其相关代码为：

```
//打印客户端连接信息
printf("Get a connection from %s:%d\n",inet_ntoa(client.sin_addr),ntohs(client.sin_port));

//发送欢迎信息
send(connectfd,"Welcome to SYK server.\n",23,0);

//接收客户端昵称
recvlen = recv(connectfd,client_name,MAXDATASIZE,0);
if(recvlen == 0){
    close(connectfd);
    printf("Client disconnected.\n");
    return;
} else if(recvlen < 0){
    close(connectfd);
    printf("Client connect broked.\n");
    return;
}
```

```
client_name[recvlen] = '\0';
printf("Client name is %s\n",client_name);

//用户模式选择
{
    char modeTemp[MAXDATASIZE] ="\0";
    int tempLen = 0;
    int mode = MODE_ERROR;

    //发送模式选择提示字符串
    char modeSTR[MAXDATASIZE] ="Mode Select \n1.Rev Echo Service\n2.Muti Chat Service\n MODE INDEX =";
    send(connectfd,modeSTR,strlen(modeSTR),0);

    //接收用户模式索引
    tempLen = recv(connectfd,modeTemp,MAXDATASIZE,0);
    modeTemp[tempLen] = '\0';

    printf("Client <%s> MODE select : %s , length:%d\n",client_name,modeTemp,tempLen);

    if(tempLen == 0){
        close(connectfd);
        printf("Client disconnected.\n");
        return;
    } else if(tempLen < 0){
        close(connectfd);
        printf("Client connect broked.\n");
        return;
    }
```

随后，子进程根据刚才接收到的用户昵称、用户选择的模式、已连接套接字描述符和已连接用户的网际套接字来创建用户节点，随后将刚创建的用户节点信息更新至文件缓存中，其次子进程根据索引自动载入相关的服务模块。当服务模块停止执行后，子进程销毁当前用户的节点缓存；然后关闭当前客户端的连接。其核心代码为：

```
//当前用户创建用户节点
NetChatClient * client_node = createClientNodeParam(connectfd,client_name,client,currentMode);

//更新节点缓存
saveNodeData(client_node);

//根据索引自动载入相关服务模块
switch(currentMode){
    case MODE_REV:{//字符串逆置服务
        printf("MODE_REV ....\n");
        revService(client_node);
    }break;
    case MODE_MUTI:{//群聊服务
        printf("MODE_MUTI ....\n");
        mutiChatService(client_node);
    }break;
    case MODE_ERROR:{//模式选择错误
        printf("MODE_ERROR ....\n");
    }break;
}

//销毁当前用户节点缓存
printf("Destory Client <%s> node cache...\n",client_name);
destoryClientCache(client_node);

//关闭客户端连接
printf("Close Client <%s> connection ... \n",client_name);
close(connectfd);
```

其中，用户节点的存在意义在于因为服务器需要对不同用户提供不同的服务，后期也有可能拓展更多的服务，不同服务之间互不干涉，且这些信息在处理的时候都是零散的，所以需要一个数据结构来整合存储这些信息。其数据结构的结构体定义如下图所示：

```
typedef struct NetChatClient{
    int connfd;                     //套接字描述符
    char name[MAX_SIZE];            //用户昵称
    int mode;                       //当前用户节点所处模式
    struct sockaddr_in addr;        //当前用户网际套接字地址结构
    struct NetChatClient * next; //下一个用户节点地址指针
}NetChatClient;
```

第一个成员 connfd 是与当前用户交互的套接字描述符，第二成员 name 是存放当前用

户昵称字符串的，第三个成员 mode 是用来存放当前用户节点所处模式的，第四个成员 addr 是用来存储当前用户网络 IP 地址端口号等信息的网际套接字地址结构，第五个成员是用来存储下一个用户节点地址指针的地址变量以便可以动态生成链表。

其中，依据参数创建用户节点的代码为：

```
//依据参数创建一个用户节点
NetChatClient * createClientNodeParam(int connectfd, char *nickname, struct sockaddr_in client, int mode) {
    NetChatClient *node = createClientNode();
    node->connfd = connectfd;
    strcpy(node->name , nickname);
    node->addr.sin_family = client.sin_family;
    node->addr.sin_addr = client.sin_addr;
    node->addr.sin_port = client.sin_port;
    node->mode = mode;
    return node;
}
```

也许你心中会有疑问，为什么会有用户节点文件缓存的存在？其意义又是何在？父子进程不是共用内存，文件描述符，信号量等信息不是相同的吗？在内存中存放用户节点数据结构的链表不就可以了吗？这样既不是浪费服务器性能，又多此一举吗？

因为此实验是“基于 Linux 系统的局域网模块化通信服务”，是模块化的通信服务。服务器是负责提供服务的，也就是说服务器端的服务模块是有可能增减的，服务模块的增减是不能影响服务器整体代码架构的。你并不能保证服务模块中不调用 fork 函数来创建子进程，如果直接采用内存中存储用户节点链表的话，这样服务模块中的父子进程和服务器主进程中的资源会造成不同步，就会造成逻辑业务与方案设计不符合的情况，即之前说过的客户端 1 群发消息客户端 2 接收不到的问题。

所以，要形成通用的模块化服务器框架，必须要有缓存这个中介的存在，来解决上述介绍的服务模块之间资源不同步的问题。

此次实验为了便捷直接采用的是文件缓存的方式，可能在处理性能上有所损耗。如果需要具体应用到大规模高并发的场景中，可以将文件缓存换成内存日志持久型数据库（如 Redis 数据库），这样服务器在缓存操作上的性能就会提升一个等级。

其中，将用户节点添加到文件缓存中的代码为：

```
//将用户节点追加至缓存文件
void saveNodeData(NetChatClient *node){
    if(node==NULL) return;
    if(node->mode == MODE_ERROR) return;

    //使用“追加”的方式打开文件
    FILE *clients = fopen(DATA_FILE_NAME,"ab");
    //直接在文件末尾写入内容
    fwrite(node,sizeof(NetChatClient),1,clients);
    fclose(clients);
}
```

销毁当前用户节点缓存的代码为：

```
//从缓存文件中去除指定套接字描述符到用户节点
void destoryClientCache(NetChatClient *node){
    NetChatClient * head = NULL;
    int isFirstNode = 1 ;
    FILE *clients = fopen(DATA_FILE_NAME,"rb");
    while(!feof(clients)){
        NetChatClient * temp = createClientNode();
        fread(temp,sizeof(NetChatClient),1,clients);
        temp->next = NULL;
        //节点比较
        //如果是指定节点，则释放相应节点内存，并不执行当次循环
        if(ClientNodeCompare(node,temp)){
            free(temp);
            continue;
        }
        if(isFirstNode){
            isFirstNode = 0;
            head = temp;
        } else {
            addtoListEnd(head,temp);
        }
    }
    fclose(clients);
    //释放最后一个节点，因为文件结尾有一个结束符，有一个多余的“脏数据”
    destoryClientNode(head,getLastNode(head));
    //save data
    saveListData(head);
    //free list
    freeList(head);
}
```

如果用户选择了字符串逆置服务，服务器会自动进入字符串逆置服务的具体实现-----revService 函数，其核心代码为：

```
//逆回文服务的具体实现
void revService(NetChatClient *client){

    char recvbuf[MAX_SIZE];
    char sendbuf[MAX_SIZE];
    int recvlen;

    //接收用户发送过来的信息
    while((recvlen = recv(client->connfd,recvbuf,MAX_SIZE,0)) > 0){

        recvbuf[recvlen] = '\0';
        printf("Receive from client <%s> message :%s, Rev Now ... \n",client->name,recvbuf);

        //逆置字符串
        strcpy(sendbuf,recvbuf);
        rev(sendbuf);

        //发送逆置字符串
        send(client->connfd,sendbuf,strlen(sendbuf),0);

    }
}
```

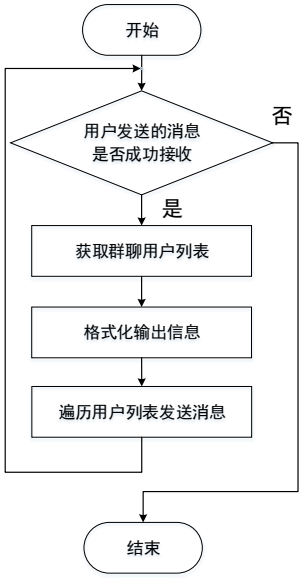
其实现的大体思路是：

a）循环调用 recv 函数接收客户端发送过来的字符串；

b）服务器对字符串进行逆置；

c）将字符串逆置结果反馈发送给客户端。

如果用户选择了群聊服务，服务器会自动进入群聊服务的具体实现 mutiChatService 函数，其实现的大体思路如下图所示。



其大体的实现核心思路为：

a) 循环调用 recv 函数接收客户端发送过来的消息；

b) 如果接收成功，则从文件缓存中获取群聊用户列表；

c) 格式化输出信息；

d) 遍历群聊用户列表，排除自己在外的所有用户调用 send 函数发送格式化的信息；

e) 如果接收不成功，则直接终止服务模块的执行。

其具体的实现代码为：

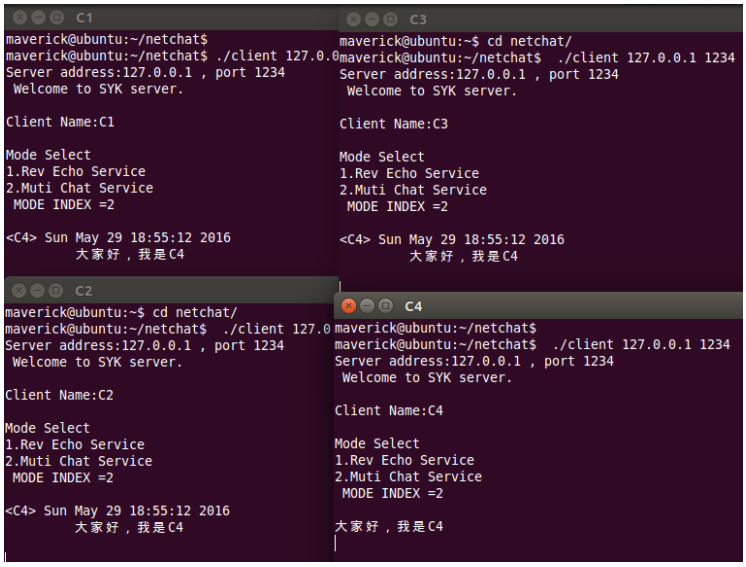
```
//接收用户发送过来的信息
while((recvlen = recv(client->connfd,recvbuf,MAX_SIZE,0)) > 0){

    recvbuf[recvlen] = '\0';
    printf("Receive from client <%s> message :%s, muti send now ...\n",client->name,recvbuf);

    //获取群聊用户列表
    NetChatClient *head  = loadClientDataWithMode(MODE_MUTI);
    NetChatClient *ptr = head;

    //获取当前系统时间
    time_t t1;
    t1 = time(NULL);//机器时间
    char *t2;
    t2 = ctime(&t1); //转换为字符串时间

    //格式化输出信息
    sprintf(sendbuf,"<%s> %s\t %s\n",client->name,t2,recvbuf);

    printf("Matched Node List:\n");
    displayForEach(head);

    printf("Recv node list:\n");
    //遍历用户列表发送信息
    while(ptr != NULL){
        if(ptr->connfd == client->connfd) continue;
        displayClientNode(ptr);
        send(ptr->connfd,sendbuf,strlen(sendbuf),0);
        ptr = ptr->next;
    }
}
```

例：有 4 个客户端（分别是 C1，C2，C3，C4）选择的是群聊服务，C4 群发了一条“大家好，我是 C4”的消息。

服务模块首先从缓存中获取所有的群聊用户列表，并遍历打印所有节点。随后遍历用户列表，排除自身在外所有节点发送信息，服务器端的 Log 日志如下图所示。

```
Receive from client <C4> message :大家好，我是C4, muti send now ...
Matched Node List:
client <[2]127.0.0.1:45878>, connfd <4>, name <C1>, mode <MUTI>
client <[2]127.0.0.1:45879>, connfd <5>, name <C2>, mode <MUTI>
client <[2]127.0.0.1:45880>, connfd <6>, name <C3>, mode <MUTI>
client <[2]127.0.0.1:45882>, connfd <8>, name <C4>, mode <MUTI>
Recv node list:
client <[2]127.0.0.1:45878>, connfd <4>, name <C1>, mode <MUTI>
client <[2]127.0.0.1:45879>, connfd <5>, name <C2>, mode <MUTI>
client <[2]127.0.0.1:45880>, connfd <6>, name <C3>, mode <MUTI>
```

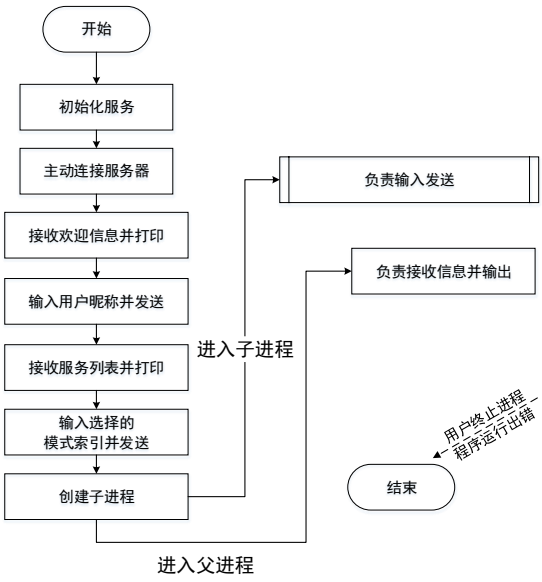
4 个客户端的响应如下图所示：



至此，服务器端整体代码架构及相关处理细节全部介绍完毕。

4.2 客户端的具体实现

客户端大体实现流程如下图所示：



首先，客户端程序先调用 socket 函数申明远端服务器使用的协议簇，所用协议，socket 函数返回一个套接字描述符，该套接字描述符用来与服务器进行交互。

随后定义一个网际套接字地址结构，地址结构中指明所使用协议簇，远端服务器 IP 地址和对外开放服务所使用的端口，接着 connect 函数主动连接远端服务器。

其初始化大体的代码为：

```
if((sockfd = socket(AF_INET,SOCK_STREAM,0))==-1){
    printf("socket() error \n");
    exit(1);
}

bzero(&server,sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(ser_port);
server.sin_addr.s_addr = inet_addr(ser_addr);

if(connect(sockfd , (struct sockaddr *)&server,sizeof(server))==-1){
    printf("connect() error\n");
    exit(1);
}
```

其后，客户端程序接收服务器传输过来的欢迎信息并打印在屏幕上；然后客户端输入用户昵称并发送给服务器。

然后，客户端程序接收来子服务器发送过来的服务列表字符串，客户端将之打印在屏幕上，然后让用户选择输入需要的服务索引号，接收后发送给服务器，其大体的代码如下图所示：

```
//客户端模式选择
{
    putchar('\n');

    char modeSTRBuf[MAXDATASIZE];
    int modeSTRLength;
    int modeInput;
    //接收服务列表并打印
    if((modeSTRLength = recv(sockfd,modeSTRBuf,MAXDATASIZE,0)) == -1){
        printf("Mode select Error");
        exit(-1);
    }
    modeSTRBuf[modeSTRLength] = '\0';
    printf("%s",modeSTRBuf);

    //输入模式索引，并发送
    scanf("%d",&modeInput);
    sprintf(modeSTRBuf,"%d",modeInput);
    send(sockfd,modeSTRBuf,strlen(modeSTRBuf),0);

    putchar('\n');
}
```

随后，客户端程序创建子进程，子进程负责监听用户输入，并将输入的字符串发送给服务器，父进程负责接收从服务器发送过来的消息并打印在屏幕上。其具体的代码为：

```
    //创建子进程，子进程负责发送，父进程负责接收
    pid_t pid;
    if((pid = fork())==0){//子进程：负责输入发送

        while(1){

            char inputBuf[MAXDATASIZE];
            int inputLength;
            int i = 0;
            char ch;
            while((ch=getchar())!='\n'){
                inputBuf[i++] = ch;
            }
            inputBuf[i] = '\0';
```

```
    inputLength = strlen(inputBuf);

    if(inputLength > 0){
        send(sockfd,inputBuf,inputLength,0);
        if(!strcmp(inputBuf,"bye")){
            exit(1);
            break;
        }
    }
}
} else if(pid > 0){//父进程:负责接收信息并输出

    while(1){

        if((num=recv(sockfd,buf,MAXDATASIZE,0))==-1){
            printf("recv() error\n");
            exit(1);
        }

        if(num == 0){
            printf("Server Disconnected ...\n");
            break;
        }

        buf[num] = '\0';
        printf("%s\n",buf);

    }

} else {
    perror("fork error.\n");
    exit(0);
}
```

至此，客户端整体处理流程及相关处理代码全部介绍完毕。

五、附录

5.1 源码目录结构

```
./netchat/              //项目工程根目录
|
├─── client  //客户端源码文件夹
|
|     └─── client.c  //客户端主程序
|
└─── server  //服务器端源码文件夹
     |
     ├─── cache.h      //客户端节点及客户端缓存机制具体的实现头文件
     |
     ├─── server.c     //服务器端主程序
     |
     └─── service.h    //服务模块具体实现头文件
```

5.2 函数及相关数据结构功能介绍

5.2.1 cache.h

```
#define MODE_REV    0x4201 //逆回文服务宏定义
#define MODE_MUTI   0x4203 //群聊服务宏定义
#define MODE_ERROR  0x4204 //模式选择错误宏定义
#define MAX_SIZE 100    //最大缓冲区大小
const char *DATA_FILE_NAME = "client.cache"; //用户数据缓存文件名
typedef struct NetChatClient {} NetChatClient; //用户节点数据结构体
NetChatClient * createClientNode();  //创建一个用户节点
NetChatClient * createClientNodeParam(int connectfd, char *nickname, struct sockaddr_in client, int mode);   //依据参数创建一个用户节点
//释放指定用户节点
void destoryClientNode(NetChatClient *head, NetChatClient *node);
void freeList(NetChatClient *head);    //释放链表空间
void displayClientNode(NetChatClient *node);    //打印指定用户节点
void displayForEach(NetChatClient *head);    //遍历打印所有用户节点
NetChatClient *getLastNode(NetChatClient *head);  //获取链表最后一个节点
//将指定到用户节点添加到链表尾部，并返回链表首节点指针
NetChatClient *addtoListEnd(NetChatClient *head,NetChatClient *node);
void saveNodeData(NetChatClient *node);    //将用户节点追加至缓存文件
void saveListData(NetChatClient *head);    //将链表所有节点存至缓存文件暂存
NetChatClient *loadClientData();          //从缓存文件中载入用户数据
//从缓存文件中载入指定模式用户列表
NetChatClient *loadClientDataWithMode(int mode);
```

```
//从缓存文件中去除指定套接字描述符到用户节点
void destoryClientCache(NetChatClient *node);
//更新缓存文件中指定的用户节点
void updateClientCache(NetChatClient *node)
//用户节点比较
int ClientNodeCompare(NetChatClient *node1 , NetChatClient *node2);
```

5.2.2 service.h

```
char *rev(char *s);//字符串逆置的具体实现
void revService(NetChatClient *client); //逆回文服务的具体实现
void mutiChatService(NetChatClient *client); //群聊服务具体实现
```

5.2.3 server.c

```
//表明该文件源码遵循 GNU 协议，只有标明该宏定义，才可以调用 clone 函数
#define _GNU_SOURCE
#define PORT 1234    //服务器对外开放服务使用端口号
#define BACKLOG 2    //服务器连接队列数量宏定义
#define MAXDATASIZE MAX_SIZE //缓冲区最大大小
#define STACK_SIZE 1024*8 //8K  ----- 子进程系统堆栈大小
int process_client();    //主要负责与客户端交互的子进程处理函数
int main(int argc,char **argv); //服务器端程序主入口
```

5.2.4 client.c

```
#define MAXDATASIZE 100   //缓冲区最大大小
int main(int argc,char **argv);  //客户端程序主入口
```