本项目主要搭建了一个基于go语言的服务端, 可以实现基础服务端的通信

# 服务端实现

## v 0.1 基础server构建

创建项目路径:

```
server.go 作为服务端的基本构建
main.go 作为当前程序的主入口
```

server.go :

```go
package main

type Server struct { //先创建一个Server类
    Ip   string
    Port int
}

func NewServer(ip string, port int) *Server { //作为一个Server类的构造器
    server := &Server{
        Ip:   ip,
        Port: port,
    }
    return server
```

```
}

func (this *Server) Start() { //用于启动服务器的方法
    //socket listen

    //accept

    //do handler

    //close listen socket
}
```

下面详细补充 start 方法

```go
func (this *Server) Start() {
    //socket listen
    //通过net.Listen(可查看源码)创建一个socket
    //传入网络类型(tcp服务器传tcp, udp传udp)和监听的地址, 返回监听对象和错误
    listener, err := net.Listen("tcp", fmt.Sprintf("%s:%d", this.Ip, this.Port))
    //用Sprintf拼接地址和端口为"127.0.0.1:8000"这种类型
    if err != nil {
        fmt.Println("net.listen err:  ", err)
        return
    }
}
```

```go
func (this *Server) Start() {
    //用大循环完成accept和do handler
    for {
        //accept
        //accept将会进行等待, 并且将下一个连接(Conn对象)传回
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("net.listen err: ", err)
            continue
        }
```

```
11
12          //do handler
13          //为了不阻塞下一次accept, 要用go携程来处理当前链接的业务
14          go this.Handler(conn)
15      }
16  }
17
18  func (this *Server) Handler(conn net.Conn) {
19      //处理当前链接的业务
20      fmt.Println("链接建立成功")
21  }
```

```
1  //关闭链接
2  //close listen socket
3  listener.Close()
```

在 main.go 中调用 server.go

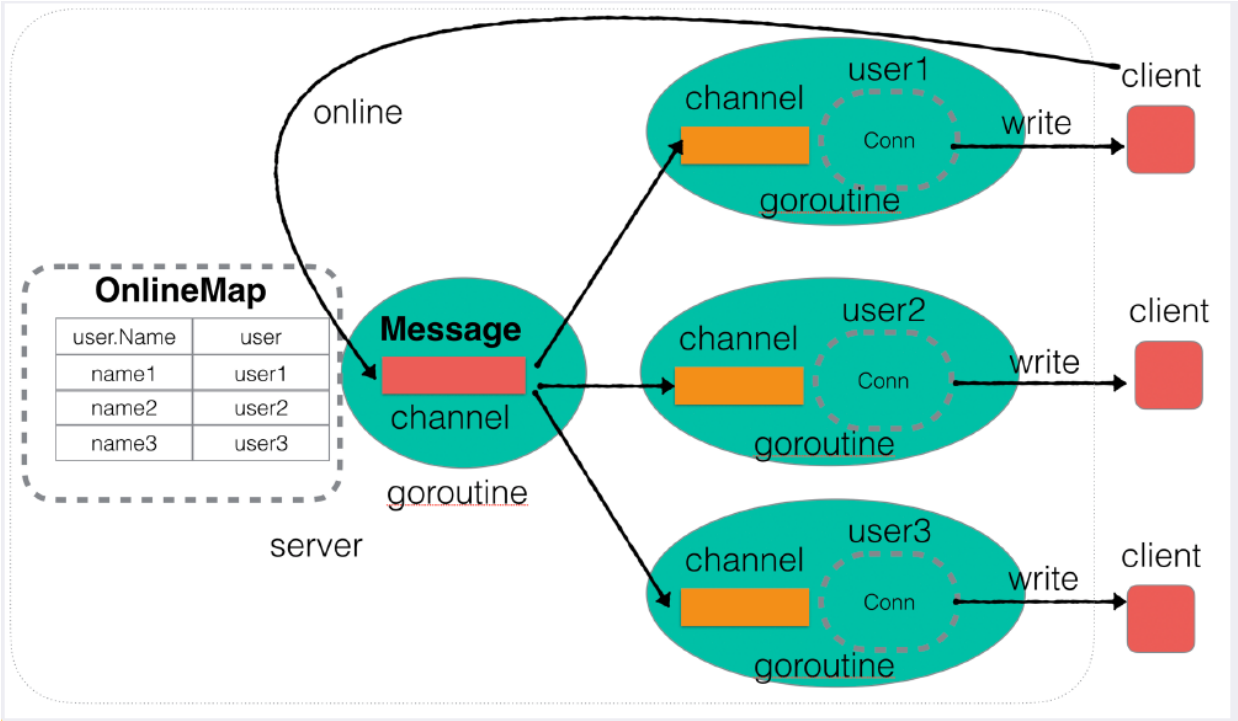
```
1  func main() {
2      server := NewServer("127.0.0.1", 8000)
3      server.Start()
4  }
```

编译执行 `go run main.go server.go` 即可让所写的 server 端进入等待状态链接状态, 如果此时有对应的客户端可尝试进行通信

在windows终端中输入 telnet (服务器地址) [端口] 即可进行服务器测试 (telnet如果不存在自行搜索解决方法)

# v 0.2 广播功能

增加从客户端传递广播消息到服务端, 再从服务端传递消息到所有在线用户




创建一个 user.go 文件

根据架构图, 首先我们需要有一个User类包含用户的所有属性

```
type User struct {
    Name string
    Addr string
    C    chan string //与当前用户绑定的channel
    conn net.Conn    //客户端通信链接
}
```

然后我们需要有实例化User类的接口

```
func NewUser(conn net.Conn) *User {
    userAddr := conn.RemoteAddr().String() //获得当前客户端地址

    user := &User{
        Name: userAddr, //以当前客户端地址作为用户名
        Addr: userAddr,
        C:    make(chan string),
        conn: conn,
    }
    go user.ListenMessage() //启用监听当前user channel消息的go程
    return user
}
```

根据架构图, 每个 User 都会启动一个 goroutine 监听客户端, 所以我们要提供监听的方法

```go
func (this *User) ListenMessage() {
    //监听当前的User channel,一旦有消息, 就直接发送给对接的客户端
    for {
        msg := <-this.C
        this.conn.Write([]byte(msg + "\n")) //将消息存下, 方便发送给客户端(要转换成二进制数组形式才能发送)
    }
}
```

根据架构, 我们需要在 server 中增加 OnlineMap 的 user 表, 同时加一个 message 管道

```go
//在server.go中更改
type Server struct { //先创建一个Server类
    Ip   string
    Port int

    //在线用户的列表
    OnlineMap map[string]*User
    mapLock   sync.RWMutex //由于OnlineMap是全局的, 所以可以加一个读写锁

    //消息广播的channel
    Message chan string
}
```

```go
func NewServer(ip string, port int) *Server { //作为一个Server类的构造器
    server := &Server{
        Ip:        ip,
        Port:      port,
        OnlineMap: make(map[string]*User),
        Message:   make(chan string),
    }
    return server
}
```

当用户上线后, 要进行广播, 用户上线时 listener.Accept 成功, 在accept之后要处理上线消息, 所以要在 Handler 中添加:

```go
func (this *Server) Handler(conn net.Conn) {
    user := NewUser(conn)

    //将用户加入到onlineMap中,在操作时要对map上锁
    this.mapLock.Lock()
    this.OnlineMap[user.Name] = user
    this.mapLock.Unlock()

    //广播用户上线消息
    this.BroadCast(user, "已上线")
}
```

补充广播的方法:

```
func (this *Server) BroadCast(user *User, msg string) {
    sendMsg := fmt.Sprintf("Address:[%s] %s%s", user.Addr, user.Name, msg)

    this.Message <- sendMsg //将消息放入msg channel中
} //处理要发送的消息, 存入msg channel
```

还要写一个监听 message 广播消息 channel 的goroutine, 一旦有消息就发送给全部的在线 user, 当server启动时就要启动这个goroutine

```go
func (this *Server) ListenMessage() {
    for {
        msg := <-this.Message

        //将msg发送给存储在OlineMap中的User
        this.mapLock.Lock()
        for _, cli := range this.OnlineMap
{
            cli.C <- msg //实现消息从message
的channel到user的channel
        }
        this.mapLock.Unlock()
    }
} //监听message, 一旦有消息就发送给在线用户
```

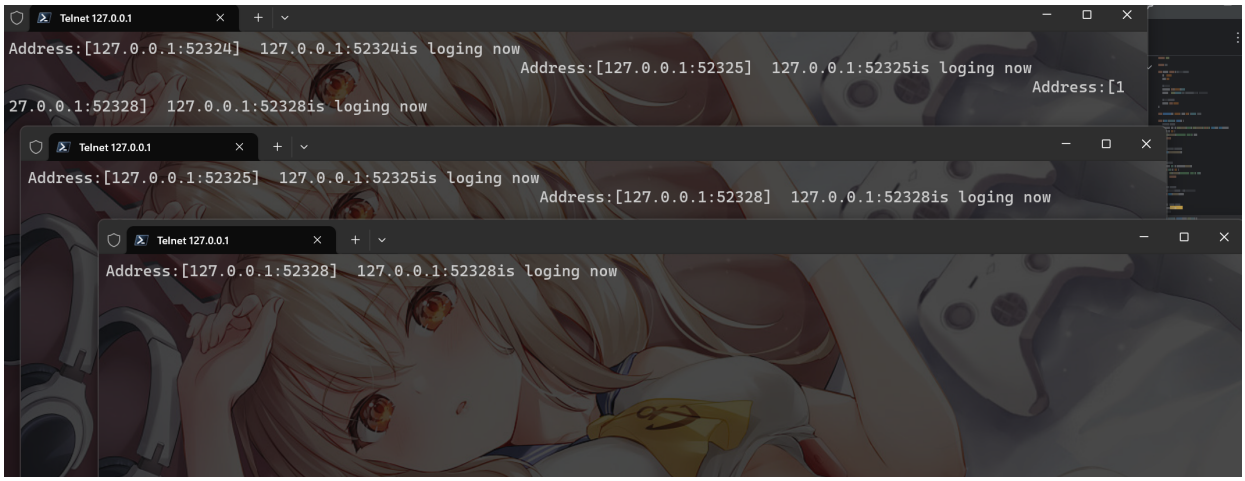
在 start 时添加启动监听message的goroutine

```go
func (this *Server) Start() {
    //socket listen

    //启动监听Message的goroutine
    go this.ListenMessage()

    //accept
    //do handler
    //close listen socket
}
```

最后将三个程序运行起来, 并且使用 telnet 进行服务器链接测试




# v 0.3 用户消息广播

该版本要实现将用户写入的消息进行广播

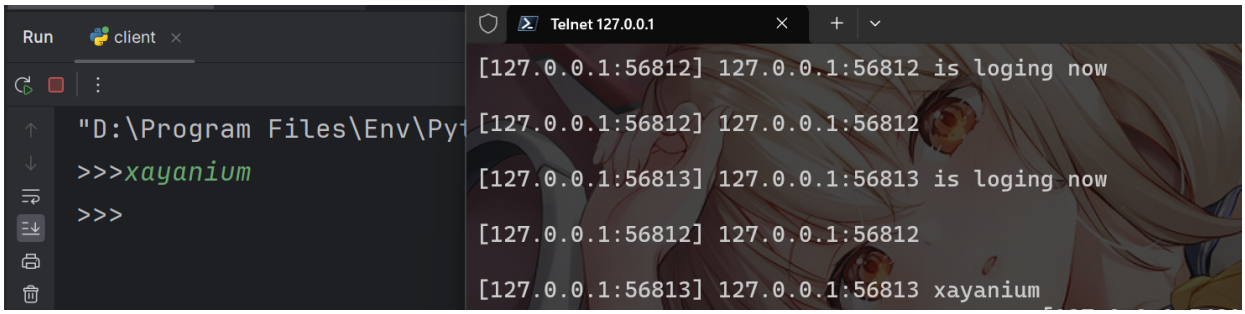
在 server 中实现接收客户端发来的的消息

```go
func (this *Server) Handler(conn net.Conn)
{
    //将用户加入到onlineMap中,在操作时要对map上锁
    //广播用户上线消息

    //接受客户端发来的消息
    go func() {
        buf := make([]byte, 4096)
```

```
8           //调用conn.Read方法从当前connection中读取数据到buf, 返回读取的字节数和错误
9           for {
10              n, err := conn.Read(buf)
11              if n == 0 { //表示客户端关闭
12                  this.BroadCast(user, "logout")
13                  return
14              }
15              if err != nil && err != io.EOF { //每次读完都会有个EOF标志结尾, 如果条件成立那就一定是进行了一次非法操作了
16                  fmt.Println("Conn Read err: ", err)
17              }
18              //提取用户的消息(去掉'\n')
19              msg := string(buf[:n-1])
20              //将得到的消息进行广播
21              this.BroadCast(user, msg)
22          }
23      }()
24  }
```

可以用之前的方法进行测试, 也可以用写好的客户端进行测试




# v 0.4 用户业务封装

之前的 server 中存在处理用户功能的业务, 这些业务最好一起封装在 user 中

所以我们可以给 user 提供一系列方法, 用这些方法替换 server 中的方法

```go
func (this *User) Online() {

} //用户上线的业务

func (this *User) Offline() {

} //用户下线的业务

func (this *User) DoMessage() {

} //用户处理消息的业务
```

我们想要将 server 中的一些方法封装进 user 中, 但是我们 user 目前无法访问当前的 server , 所以我们要考虑给当前 user 链接对应的 server

```
//增加一些东西
type User struct { //User类
    // ...
    server *Server //当前用户所属server
}

func NewUser(conn net.Conn, server *Server) //...
    user := &User{
        //...
        server: server,
    }
    //...
} //记得更改server中这个方法传过来的参数
```

完成用户上线业务:

```go
func (this *User) Online() {
    //用户上线, 将用户加入OnlineMap
    this.server.mapLock.Lock()
    this.server.OnlineMap[this.Name] = this
    this.server.mapLock.Unlock()

    //广播用户上线信息
    this.server.BroadCast(this, "log in")
} //用户上线的业务
```

完成下线任务和处理消息业务

```go
func (this *User) Offline() {
    //用户下线, 将用户从OnlineMap中删除
    this.server.mapLock.Lock()
    delete(this.server.OnlineMap, this.Name)
    this.server.mapLock.Unlock()
    //广播用户下线信息
    this.server.BroadCast(this, "log out")
} //用户下线的业务

func (this *User) DoMessage(msg string) {
    this.server.BroadCast(this, msg)
} //用户处理消息的业务
```

替换 server 中对应的业务

```
```

```go
func (this *Server) Handler(conn net.Conn)
{
    user := NewUser(conn, this)
    //this.mapLock.Lock()
    //this.OnlineMap[user.Name] = user
    //this.mapLock.Unlock()
    //this.BroadCast(user, "is log now")
    user.Online() //上线

    go func() {
        //...
        for {
            //...
            if n == 0 {
                //this.BroadCast(user,
"logout")
                user.Offline() //下线
                return
            }
            //...
            //this.BroadCast(user, msg)
            user.DoMessage(msg)
        }
    }()
}
```

至此我们就将用户所属的模块基本封装在了 user 类中

进行测试

# v 0.5 用户在线查询

设定一个协议, 用户一旦输入特定的指令，我们将全部的在线用户返回给当前进行查询的用户即可

我们应该在 DoMessage 中处理业务

```
func (this *User) DoMessage(msg string) {
    if msg == "who" {
        //查询当前在线用户都有哪些
        this.server.mapLock.Lock()
        for _, user := range this.server.OnlineMap {
            onlineMessage := fmt.Sprintf("%s is online\n", user.Name)
            //将查询到的消息传给发起查询的用户

 this.conn.Write([]byte(onlineMessage))
        }
        this.server.mapLock.Unlock()
    }   else {
        this.server.BroadCast(this, msg)
    }
} //用户处理消息的业务
```

进行测试

# v 0.6 自定义修改用户名

还是在 DoMessage 中处理

```
func (this *User) DoMessage(msg string) {
    if  {
        // ...
        } else if len(msg) > 7 && msg[:7] == "rename|" {
        //设定消息格式为 rename|xxxx
        newName := strings.Split(msg, "|")[1] //从字符串中通过某个字符截取，将两部分放入不同数组中

        //判断name是否已经存在
        _, ok := this.server.OnlineMap[newName]
        if ok {
            this.conn.Write([]byte("this username has been used\n"))
        } else {
            this.server.mapLock.Lock()
            delete(this.server.OnlineMap, this.Name) //删掉map中之前的名字
```

```
            this.Name = newName //更新user实例中的名字
            this.server.OnlineMap[newName] = this    //更改为想要的名字
            this.server.mapLock.Unlock()

            this.conn.Write([]byte("You have successfully updated your name\n"))
        }

    } else //...
} //用户处理消息的业务
```

# v 0.7 增加私聊功能

消息格式 : to|张三|消息内容

```
func (this *User) DoMessage(msg string) {
    if  {
        //...
    } else if {
        //...
    } else if len(msg) > 4 && msg[:3] == "to|" {
        //设定消息格式为 to|username|message
```

```
8           //1. 获取对方的用户名
9           remoteName := strings.Split(msg, "|")[1]
10          if remoteName == "" {
11              this.conn.Write([]byte("format error, please use the correct format"))
12              return
13          }
14          //2. 根据用户名得到user对象
15          remoteUser, ok := this.server.OnlineMap[remoteName]
16          if !ok {
17
this.conn.Write([]byte("username not exist"))
18          }
19          //3. 获取消息内容, 通过对方的User对象将消息发过去
20          content := strings.Split(msg, "|")[2]
21          if content == "" {
22
this.conn.Write([]byte("message is empty"))
23              return
24          }
```

```
25
    remoteUser.conn.Write([]byte(this.Name + "
    tell you " + content + "\n"))
26      } else //...
27  } //用户处理消息的业务
```

# 客户端实现

## v 1.1 建立连接

新建一个 `client.go`

首先创建基本的客户端类

```
1  type Client struct {
2      ServerIp   string
3      ServerPort int
4      Name       string
5      conn       net.Conn
6  }
```

然后写类的构造函数, go的客户端可以通过 Dial 进行网络连接

```

```

```go
func NewClient(serverIp string, serverPort int) *Client {
    //创建客户端对象
    client := &Client{
        ServerIp:   serverIp,
        ServerPort: serverPort,
    }
    //链接服务器
    conn, err := net.Dial("tcp", fmt.Sprintf("%s:%d", serverIp, serverPort)) //传入网络类型和ip地址, 返回连接对象和错误
    if err != nil {
        fmt.Println("net.Dial error:", err)
        return nil
    }
    client.conn = conn
    //返回创建对象
    return client
}
```

然后写客户端的启动程序

```go
func main() {
    client := NewClient("192.168.1.6", 8000)
    if client == nil {
        fmt.Println("connection error.....")
        return
    }

    time.Sleep(1 * time.Second)
    fmt.Println("connection start successfully!")

    select {} //之后再补充相应的功能
} //启动客户端
```

现在可以在 goland 中直接进行检测, 如果程序正常将会在客户端的终端中出现 connection start successfully!

## v 1.2 解析命令行

我们客户端的 IP 是直接写死的, 我们可以尝试让客户端通过命令行进行输入

解析命令行要借助 flag 库, 我们解析命令行要在 main 执行之前解析

go语言每个文件都会有一个 init 函数, 该函数是在 main 函数之前执行的

```
// 设定两个全局变量
var serverIp string
var serverPort int

//运用flag库进行命令行解析
func init() {
    //四个参数分别为: 要赋值的变量, 命令行中显示的名字, 默认值, 变量的说明
    flag.StringVar(&serverIp, "ip", "127.0.0.1", "set IP address")
    flag.IntVar(&serverPort, "port", 8000, "set Port")
}
```

此时对文件进行编译:

```
go build -o server.exe server.go user.go main.go
go build -o client.exe client.go
```

输入 `.\client.exe -h` 会告诉你执行时客户端可输入的内容:

```
PS D:\Code_Project\Go\golang-IM-System> .\client.exe -h
Usage of D:\Code_Project\Go\golang-IM-System\client.exe:
  -ip string
        set IP address (default "127.0.0.1")
  -port int
        set Port (default 8000)
```

启动服务器：`.\server.exe`

启动客户端：`.\client -ip 127.0.0.1 -port 8000`

这样就可以读取到命令行中传递的参数了

# v 1.3 实现菜单的显示

要给当前的 client 类绑定一个显示菜单的方法

```
func (this *Client) menu() bool {
    var flag int
    fmt.Println("input 1 : public talk")
    fmt.Println("input 2 : private talk")
    fmt.Println("input 3 : change username")
    fmt.Println("input 0 : exit")
```

```
7
8       fmt.Scanln(&flag)
9       if flag >= 0 && flag <= 3 {
10          this.flag = flag
11          return true
12      } else {
13          fmt.Println("Please input the
    correct number")
14          return false
15      }
16  }
```

对 client 类新增 flag 属性

```
1   type Client struct {
2       //...
3       flag        int //保存当前client的menu选择
4   }
5
6   func NewClient(serverIp string, serverPort
    int) *Client {
7       client := &Client{
8           //...
9           flag:        114514, //初始化时默认值
10      }
11      //...
12  }
```

增加执行客户端业务的方法

```
func (this *Client) run() {
    for this.flag != 0 {

        for this.menu() != true {
        }

        switch this.flag {
        case 1: //公聊模式
            fmt.Println("public")
            break
        case 2: //私聊模式
            fmt.Println("private")
            break
        case 3: //更新用户名
            fmt.Println("change username")
            break
        }
    }
}
```

进行测试

# v 1.4 实现更新用户名

实现 UpdateName 方法:

```
func (this *Client) UpdateName() bool {
    fmt.Println(">>>Please enter Your username:>>>")
    fmt.Scanln(&this.Name)

    sendMsg := "rename|" + this.Name + "\n"
    _, err := this.conn.Write([]byte(sendMsg))
    if err != nil {
        fmt.Println("conn.Write err:", err)
        return false
    }
    return true
}
```

此外我们还需要接受服务端传回的消息, 所以我们需要一个 goroutine 来实现

```go
func (this *Client) DealResponse() {
    //可以永久阻塞监听
    io.Copy(os.Stdout, this.conn) //不断等待服务端传回的数据，一旦服务端传回数据就立刻输出到终端
}
```

在 main 中开启此 goroutine

```go
package main

import (
    "flag"
    "fmt"
    "io"
    "net"
    "os"
)

type Client struct {
    ServerIp   string
    ServerPort int
    Name       string
    conn       net.Conn
    flag       int //保存当前client的menu选择
}

func NewClient(serverIp string, serverPort int) *Client {
```

```
20      //创建客户端对象
21      client := &Client{
22          ServerIp:   serverIp,
23          ServerPort: serverPort,
24          flag:       114514,
25      }
26      //链接服务器
27      conn, err := net.Dial("tcp",
fmt.Sprintf("%s:%d", serverIp,
serverPort)) //传入网络类型和ip地址, 返回连接对
象和错误
28      if err != nil {
29          fmt.Println("net.Dial error:",
err)
30          return nil
31      }
32      client.conn = conn
33      //返回创建对象
34      return client
35  }
36
37  func (this *Client) Menu() bool {
38      var flag int
39      fmt.Println("input 1 : public talk")
40      fmt.Println("input 2 : private talk")
41      fmt.Println("input 3 : change
username")
42      fmt.Println("input 0 : exit")
```

```
43
44      fmt.Scanln(&flag)
45      if flag >= 0 && flag <= 3 {
46          this.flag = flag
47          return true
48      } else {
49          fmt.Println("Please input the
    correct number")
50          return false
51      }
52  }
53
54  func (this *Client) Run() {
55      for this.flag != 0 {
56
57          for this.Menu() != true {
58          }
59
60          switch this.flag {
61          case 1: //公聊模式
62              fmt.Println("public")
63              break
64          case 2: //私聊模式
65              fmt.Println("private")
66              break
67          case 3: //更新用户名
68              this.UpdateName()
69              break
```

```
70          }
71      }
72  }
73
74  func (this *Client) UpdateName() bool {
75      fmt.Println(">>>Please enter Your
    username:>>>")
76      fmt.Scanln(&this.Name)
77
78      sendMsg := "rename|" + this.Name +
    "\n"
79      _, err :=
    this.conn.Write([]byte(sendMsg))
80      if err != nil {
81          fmt.Println("conn.Write err:",
    err)
82          return false
83      }
84      return true
85  }
86
87  func (this *Client) DealResponse() {
88      //可以永久阻塞监听
89      io.Copy(os.Stdout, this.conn) //不断等待
    服务端传回的数据，一旦服务端传回数据就立刻输出到终
    端
90  }
91
```

```
var serverIp string
var serverPort int

func init() {
    //...
    go client.DealResponse()
    //...
}
```

进行测试

# v 1.5 实现公聊功能

实现公聊的对应方法:

```
func (this *Client) PublicChat() {
    var chatMsg string
    fmt.Println(">>>Please input your chat message('exit' to quit):")
    fmt.Scanln(&chatMsg)

    for chatMsg != "exit" { //不断监听输入的发送消息

        if len(chatMsg) != 0 {
            sendMsg := chatMsg + "\n" //根据公聊的协议
```

```
            _, err := this.conn.Write([]byte(sendMsg))
            if err != nil {
                fmt.Println("conn.Write err:", err)
                break
            }
        }

        chatMsg = ""
        fmt.Println(">>>Please input your chat message('exit' to quit):")
        fmt.Scanln(&chatMsg)
    }
}
```

可自行更改输入相关的函数, 该函数无法输入空格

执行测试

# v 1.6 实现私聊功能

首先我们需要知道能向哪些用户发消息, 所以可以实现一个方法来获取在线用户:

```go
func (this *Client) SelectUser() {
    fmt.Println("users online:")
    sendMsg := "who\n" //根据之前服务端的协议
    _, err := this.conn.Write([]byte(sendMsg))
    if err != nil {
        fmt.Println("conn Write error:", err)
        return
    }
}
```

然后仿照公聊的方法来实现私聊

```go
func (this *Client) PrivateChat() {
    var remoteName string
    var chatMsg string

    this.SelectUser()
    time.Sleep(1000)
    fmt.Println(">>>Please input the username('exit' to quit):")
    fmt.Scanln(&remoteName)

    for remoteName != "exit" { //对消息发送的用户进行选择
        this.SelectUser()
```

```
12          fmt.Println("Please enter
    context('exit' to quit)")
13          fmt.Scanln(&chatMsg)
14
15          for chatMsg != "exit" { //不断监听输
    入的消息
16
17              if len(chatMsg) != 0 {
18                  //根据私聊的协议，因为我们写的
    服务端会截断最后一个换行，所以需要两个'\n'
19                  sendMsg := "to|" +
    remoteName + "|" + chatMsg + "\n\n"
20                  _, err :=
    this.conn.Write([]byte(sendMsg))
21                  if err != nil {
22
    fmt.Println("conn.Write err:", err)
23                      break
24                  }
25              }
26
27              chatMsg = ""
28              fmt.Println(">>>Please input
    your chat message('exit' to quit):")
29              fmt.Scanln(&chatMsg)
30          }
31
32          this.SelectUser()
```

```
33          fmt.Println(">>>Please input the
username('exit' to quit):")
34          fmt.Scanln(&remoteName)
35      }
36  }
```

进行测试

# 结尾

本项目到此算是大致完成架构图给出的东西了, 由于最后进行调试时除了少许问题, 所以该 md 中的代码不一定完全正确, 具体以实际上传代码为准