# 并发聊天室服务器

并发编程和网络编程是现今行业开发中常用的技术。Go语言强大的语法设定使得并发和网络编程都变的简洁而高效。

下面我们利用前面学到的知识，使用并发和网络实现一个简单的网络在线聊天室。体会下这两种技术的实际应用。在整个聊天室的项目中，充分利用了go程并发，处理不同任务。

整个聊天室程序可简单划分为如下模块，都分别使用go程来实现：

**主go程（服务器）：**

负责监听、接收用户（客户端）连接请求，建立通信关系。同时启动相应的go程处理任务。

**处理连接用户数据go程：HandleConnect**

负责新上线用户的存储，用户消息读取、发送，用户改名、下线处理及超时处理。

为了提高并发效率，同时给一个用户维护多个go程来并行处理上述任务。

**用户消息广播go程：Manager**

负责在线用户遍历，用户消息广播发送。需要与**HandleConnect** go程及用户子go程协作完成。

**go程间应用数据及通信：**

map：存储所有登录聊天室的用户信息， key：用户的ip+port。Value：Client结构体。

Client结构体：包含成员：用户名Name，网络地址Addr（ip+port），发送消息的通道C（channel）

通道message：协调并发go程间消息的传递。

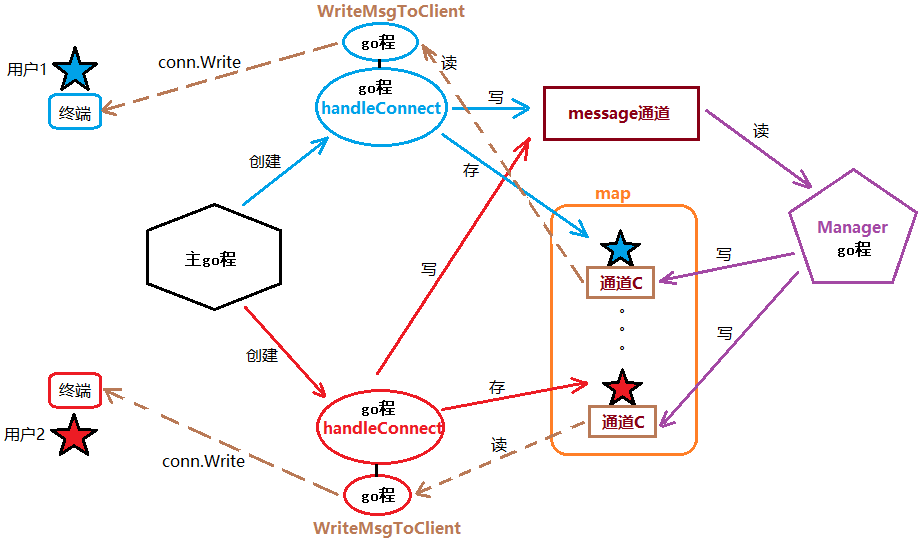
## 广播用户上线

首先，服务器启动，等待用户建立通信连接。当有用户连接上来，将其存储到map中，这样就维护了一个“在线用户”的列表。当再有新用户连接上来时，应向该列表中所有用户进行广播通知，提示xxx用户上线。

当然，简单实现手法可以循环读取列表中的用户，依次向其发送消息通知新用户上线。但这种方式无疑是一种串行的通信手段，实现简单，但执行效率较低。

在go语言中，我们利用go程轻便、高效、并发性好的特性，给每个登录用户维护多个go程来进行数据通信，借助channel不需要使用同步锁，就可以实现高效的并发通信。

下图充分利用goroutine和channel实现了新用户登录，向所有在线用户进行广播通知：



分析上图，主要分为几大模块。

全局位置定义用户结构体类型 Client，存储登录用户信息。成员包含channel、Name、Addr

type Client struct {

C chan string

Name string

Addr string

}

定义全局通道message处理消息。

定义全局map 存储在线用户信息。Key为用户网络地址。Value为用户结构体。

主go程，监听客户端连接请求，当有新的客户端连接，创建新go程handleConnet处理用户连接。

handleConnet go程，获取用户网络地址（Ip+port），创建新用户结构体，包含成员C、Name、Addr。新用户的Name和Addr初值都是用户网络地址（Ip+port）。将用户结构体存入map中。并创建WriteMsgToClient go程，专门负责给当前用户发送消息。组织新用户上线广播消息内容，写入全局通道message中。

WriteMsgToClient go程，读取用户结构体C中的数据，没有则阻塞等待，有数据写出给登录用户。

Manager go程，给map分配空间。循环读取 message 通道中是否有数据。没有，阻塞等待。有则解除阻塞，将message通道中读到的数据写到用户结构体中的C通道。

代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
*// 定义用户结构体类型***type** Client **struct** {  
 C **chan** string  
 Name string  
 Addr string  
}  
*// 定义全局 map 存储在线用户 key:IP+port, value:Client***var** onlineMap **map**[string]Client  
  
*// 定义全局 channel 处理消息***var** message = make(**chan** string)  
  
**func** WriteMsgToClient(clnt Client, conn net.Conn) {  
 *// 循环跟踪 clnt.C，有消息则读走，Write 给客户端* **for** msg := **range** clnt.C {  
 conn.Write([]byte(msg + **"\n"**)) *// 发送消息 给客户端* }  
}  
  
**func** MakeMsg(clnt Client, msg string) (buf string) {  
 buf = **"["** + clnt.Addr + **"]"** + clnt.Name + **": "** + msg  
 **return**}  
  
**func** HandleConnect(conn net.Conn) {  
 **defer** conn.Close()  
 *// 获取新连接上来的用户的网络地址(IP+port)* netAddr := conn.RemoteAddr().String()  
 *// 给新用户创建结构体。用户名、网络地址一样* clnt := Client{make(**chan** string), netAddr, netAddr}  
 *// 将新创建的结构体，添加到 map 中，key值为获取到的网络地址（IP+port）* onlineMap[netAddr] = clnt  
  
 *// 新创建一个go程，专门给当前客户端发送消息。* **go** WriteMsgToClient(clnt, conn)  
  
 *// 广播新用户上线  
 // message <- "[" + clnt.Addr + "]" + clnt.Name + ": login"* message <- MakeMsg(clnt, **"login"**)  
  
 **for** { *// 防止当前go程结束。* runtime.GC()  
 }  
}  
  
**func** Manager() {  
 *// 给map分配空间* onlineMap = make(**map**[string]Client)  
  
 *// 循环读取 message 通道中的数据* **for** {  
 *// 通道 message 中有数据读到 msg 中。 没有，则阻塞* msg := <-message  
  
 *// 一旦执行到这里，说明message中有数据了，解除阻塞。 遍历 map* **for** \_, clnt := **range** onlineMap {  
 clnt.C <- msg *// 把从Message通道中读到的数据，写到 client 的 C 通道中。* }  
 }  
}  
  
**func** main() {  
 *// 创建监听 socket* listener, err := net.Listen(**"tcp"**, **"127.0.0.1: 8000"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listener.Close()  
  
 *// 创建go程 处理消息* **go** Manager()  
  
 *// 循环接收客户端连接请求* **for** {  
 conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **continue** *// 失败，监听其他客户端连接* }  
  
 *// 给新连接的客户端，单独创建一个go程，处理客户端连接请求* **go** HandleConnect(conn)  
 }  
}

## 广播用户消息

当某个客户端向服务端发送消息后，服务端应将该消息广播给其它的客户端，达到聊天室的群聊效果。

开启一个新的go程，为方便传参，可以选择匿名go程。专门负责接收从客户端传递过来的数据，然后将接收到的数据写到messaage通道中。

在实现“广播用户上线”时，我已经完成：Manager go程会阻塞读message通道，一旦有数据，则遍历map中的在线用户。将数据写到结构体成员的C通道中。WriteMsgToClient go程会迭代C这个channel，最终将数据发送给客户端。

综上，实际上我们想完成“广播用户消息”给所有在线用户的功能，只需要将读到的数据写到message通道即可达到目的。

相关代码：

**func** HandleConnect(conn net.Conn) {

……

……  
 *// 广播新用户上线* message <- MakeMsg(clnt, **"login"**)  
  
 *// 创建一个新go程，循环读取用户发送的消息，广播给在线用户* **go func**() {buf := make([]byte, 2048)  *// 定义切片缓冲区，存储读到的用户消息*  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 { *// 用户退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 msg := string(buf[:n]) *// 保存用户写来的消息内容* message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户* }  
 }()  
  
 **for** { *// 不能让当前go程结束。* ;  
 }  
}

## 展示在线用户

因为nc工具默认会添加‘\n’， 所以conn.Read()读取用户消息后，修改保存用户消息内容实现语句：

msg := string(buf[:n-1]) 重新读取用户消息。

读到后，对消息内容进行判断：如果用户发送了“who”，则当成一个查询指令处理。遍历map中所有在线用户，取出每个用户的相关描述信息，组成提示消息，写给当前用户即可。

由于这里客户端我们使用nc工具模拟，该工具对中文支持较差，所以我们组织的用户描述信息中不要包含中文字符。

代码片段如下：

msg := string(buf[:n-1]) *// 保存用户写来的消息内容, nc 工具默认添加‘\n’***if** msg == **"who"** && len(msg) == 3 { *// 判断用户发送了 who 指令* conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap { *// 遍历map获取在线用户* userInfo := user.Addr + **":"** + user.Name + **"\n"** *// 组织在线用户信息* conn.Write([]byte(userInfo)) *// 写给当前用户* }  
} **else** {  
 message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户*}

## 修改用户名

前面我们查看用户信息时，用户名都是与用户网络地址相同的内容。主要由于用户登录时，创建该用户名不是用户自己完成的，无法洞悉用户的意图。当用户成功登录上来可以通过给服务器发送消息，来修改自己的用户名。

设定，如果用户发送“rename | Iron man”指令，既是想修改自己的用户名为“Iron man”。判断用户消息，是否包含“rename|”关键字：if len(msg) >= 8 && msg[:6] == "rename" {。如果是，那么拆分用户意欲修改的用户名保存。**strings.Split()**函数可以完成拆分字符串操作。

将该用户名替换当前用户的Name。使用用户的Addr作为key，找到map中当前用户，覆盖即可达到改名的目的。操作结束提示用户改名成功。

代码片段如下：

msg := string(buf[:n-1])  
**if** msg == **"who"** && len(msg) == 3 {  
 conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap {  
 userInfo := user.Addr + **":"** + user.Name + **"\n"** conn.Write([]byte(userInfo))  
 }  
 *// 判断用户输入的前6个字符是否为 rename*} **else if** len(msg) >= 8 && msg[:6] == **"rename"** { *// rename | Iron man* newName := strings.Split(msg, **"|"**)[1] *// 按"|"拆分，rename为[0], Iron man为[1]* clnt.Name = newName *// 替换掉当前用户原始Name* onlineMap[netAddr] = clnt *// 使用netAddr为key找到map中当前用户。覆盖* conn.Write([]byte(**"rename successful\n"**))  
} **else** {message <- MakeMsg(clnt, msg)  
}

## 用户退出

前面在“广播用户消息”时，当conn.Read() 读到0时，我们在服务器端，简单打印了“用户xxx退出登录”的提示。

但实际上，在聊天室中，有在线用户离开，我们应该将这一事件广播给所有用户知晓，并且将该用户从map在线用列表中移除。需要实时的监看在线用户的状态。可以创建channel来检测用户退出状态，并使用select来监听channel上的数据流动。

当channel上有数据时，select对应阻塞case语句得以执行。将用户从map中移除。同时通知所有在线用户。

代码片段：

**func** HandleConnect(conn net.Conn) {

……

message <- MakeMsg(clnt, **"login"**)  
**isQuit := make(chan bool)** *// 检测用户主动退出*  
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 {  
 **isQuit <- *true*** *// 用户主动退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 ……

}  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit: *// 用户不主动退出，阻塞*

close(clnt.C)   
delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"logout"**) *// 广播给在线用户，谁退出了* **return** *// 结束当前退出用户对应go程* }  
 }  
}

## 超时处理

如果客户端没有主动退出，并且长时间没有发送消息，会一直占用服务端的资源。服务器通常针对这种用户添加“超时强踢”机制，强制将该客户端与服务器连接断开。

可以借助并发编程时我们所学的select超时机制来实现。Select监听time.After(60 \* time.Second) 通道上的数据流动。如果在计时期间一直没有数据，通道中会被写入当前系统时间，select 的case满足读条件，不再阻塞。但，有一个问题，用户如果持续在输入数据，这个计时器依然在计时，时间到，依然会强制踢出用户。

因此，我们另外创建一个通道hasData来检测用户是否有数据发送，让Select也来监听这个channel。这样，当用户有数据输入时，select监听的这个hasData通道会满足case条件得以执行，但我们不做任何处理。目的是使得监听在select中的计时器被重新计时。

只有当真正持续60s没有数据发送时，select 中用于计时的case才满足条件，将用户与服务器连接断开。

代码片段：

**func** HandleConnect(conn net.Conn) {  
 ……

……  
isQuit := make(**chan** bool)  
 **hasData := make(chan bool)** *// 检测用户是否有消息发送*  
   
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 ……  
 msg := string(buf[:n-1]) if {

……

} else if {

……

} else {

……

}  
 hasData <- ***true*** *// 只要执行到这里，就说明用户有数据发送* }  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit:

close(clnt.C)delete(onlineMap, netAddr) message <- MakeMsg(clnt, **"logout"**) **return** **case** <-hasData:  
 *// 什么都不做，目的是让计时器归零* **case** <-time.After(60\*time.***Second***):

close(clnt.C)  
 delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"time out leave"**) *// 广播给在线用户，超时退出* **return** *// 结束当前退出用户对应go程* }  
 }  
}

这里需要注意的是，每循环一次，第三个case后面的时间都会重新计算。（例如:执行完case<-hasData后，紧跟着执行第三个case，发现时间是10秒，不到60秒，条件不成立，不会执行该case后面的代码，进入下次循环，这时时间重新计算）

当hasData没有数据，isQuit没有数据，60s时间没有到，这时三个case都阻塞等待。直到60秒后，前两个case条件依然不成立，第三个case满足，执行后面代码，断开客户端连接，踢下线。