## 服务端并发模型:

我实现了两种模型（实际测试都可正常使用）：

### 静态线程池+IO多路复用（一个线程多个客户端）

#### 类说明：

**class Chat\_One\_Process**():  
  
 **def** \_\_init\_\_(self,sock,addr): # 接收到一个连接请求后，生成一个该对象,该对象存储该连接用户的所有信息  
 self.all\_buffer = '' # 该套接字接收缓冲区  
 self.one\_buffer = '' # 该套接字正在接收的单个数据包部分  
 self.flag = False # 报头已得到标识  
 self.head\_str = '' # 报头的字符串表示  
 self.data\_body\_len = 0 # 此数据报数据部分应该接受的长度  
 self.recive\_len = 0 # 当前数据报已收到的长度  
 self.sock = sock # 该对象处理的套接字  
 self.addr = addr # 该对象处理对象的地址  
  
 **def game\_process**(self,room\_name): # 处理每个房间游戏题目的发放和胜利玩家提示  
   
 **def send\_mess**(self, mess\_type, action\_type, str\_mess, sock):

# 被body\_process调用，将要发送的一个数据包放入发送队列  
   
 **def body\_process**(self,one\_buffer, head\_str, sock): # 该函数处理一个数据包

并给予客户相应的反馈，反馈信息放在消息队列中，并在套接字可写时发送出去

**def clear\_process**(self):# 清除连接断开后的全局信息  
   
  
**def mess\_process**(self): # 消息处理，当套接字可读时，调用该函数

该函数分析出一个完整的协议数据包后交给body\_process函数处理

该类主要负责处理一个用户，当新来一个连接时，就生成一个对象，该对象只处理该套接字，

**class MyThread**(threading.Thread): # 自定义线程类  
  
 **def** \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
 self.func = self.listen\_mess  
 self.chat\_users = {} # sock:chat-one  
 self.mess\_input = [] # 该线程处理的所有套接字  
  
 **def run**(self):  
 apply(self.listen\_mess,())  
  
 **def user\_append**(self,sock,addr):# 该线程新加一个套接字

# 当处理的mess\_input达到一定数字后就不再加入，而是交给其他线程

per = Chat\_One\_Process(sock,addr) # 加一个处理对象  
 self.chat\_users[sock] = per  
 self.mess\_input.append(sock)  
  
 **def listen\_mess**(self): # 线程调用的方法,监听mess\_input套接字

当可读时调用Chat\_One\_Process的消息接收函数进行包解析

当解析到对方关闭连接时，移除对该套接字的监听

当可写时从发送队列中取出消息写入套接字

当写入失败时，主动关闭该套接字，并且清除全局环境

#### 函数说明：

**def write\_file**(user): # 当有新用户注册时,将注册信息写入文件  
  
**def update\_time**(user\_time,user\_name):# 当一个用户退出后, 更新在线时长  
  **def write\_time**(user\_name): # 新注册一个用户, 初始化在线时长  
  
**def read\_file**(str): # 服务程序启动后从文件读取用户信息

### 多线程（一个线程一个客户端）

该模型大致处理函数都一致，只是每次都让一个线程处理一个客户端，这样就不需要设置

套接字非阻塞，也不会影响其他客户，函数与上一个模型全部一致

## 服务端全局变量说明

user\_time\_path = r'D:\online\_time.txt' # 配置存储用户在线时间目录  
user\_pwd\_path = r'D:\server\_user.txt' # 配置用户密码信息目录  
s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #申请监听套接字  
s.bind(('0.0.0.0', 5555)) # 绑定监听的地址和端口号  
s.listen(5) # 设置待accept套接字最大数目  
user\_time = {} # key为用户名 value 为在线时间 格式 wen@@3600 后面为秒数  
user\_pwd = read\_file(user\_pwd\_path) # 从文件读取用户信息  
all\_socket = [] # 所有套接字列表  
all\_user = {} # key为套接字 value 为用户名  
all\_room = [] # key 房间名  
mess\_queue = {} # key 为套接字, value为发送列表[mess1,mess2]  
room\_user = {} # key 房间名 value 该房间所有用户, 是一个list  
room\_answer = {} # key 房间名 value 是一个[u1 ,ans\_str] 第一个key是回答的用户，第二个是回答串  
answer\_valid = {} # key 房间名 value为当前回答是否有效, 就是是否在15秒之内被服务器收到  
num\_game = {} # key为房间名 value为list （当前21点游戏四个数字)  
threads = [] # 线程池信息

下面着重介绍包解析函数:

**def mess\_process**(self): # 消息处理，当套接字可读时，调用该函数  
 **global** mess\_queue # 全局消息队列, key为sock, value为待发送消息list  
  
 tmp\_buffer = self.sock.recv(1024) # 本次获取的数据  
 self.all\_buffer = self.all\_buffer + tmp\_buffer  
 **if not** tmp\_buffer: # 对方关闭连接  
 mess\_queue.pop(self.sock) # 清除消息队列  
 self.sock.close() # 关闭套接字  
 **if** self.sock **in** all\_socket: # 从全部变量中清除  
 all\_socket.remove(self.sock)  
 self.clear\_process() # 清除其他全局信息  
 **return  
  
 while** True: # 循环解析本次收到的数据包,一旦受到一个完整包就交给body\_process函数

直到self.all\_buffer不是一个完整的包  
 **if** self.flag: # 如果已得到报头  
 tmp\_len = self.data\_body\_len - self.recive\_len  
 **if** len(self.all\_buffer) < tmp\_len: # 不能收到完整数据包  
 self.one\_buffer += self.all\_buffer # 将所有数据放到单个包里,准备继续接收  
 self.recive\_len += len(self.all\_buffer) # 改变已接受长度  
 self.all\_buffer = '' # 清空接收缓冲区  
 **break  
 else**: # 已经收到完整数据包  
 self.one\_buffer += self.all\_buffer[0:tmp\_len]  
 self.all\_buffer = self.all\_buffer[tmp\_len + 1:] # 清空已经放到单个包的缓冲区数据  
 ###  
 # 处理  
 self.body\_process(self.one\_buffer, self.head\_str, self.sock)  
 ###  
 self.flag = False # 变成未接收到报头  
 self.one\_buffer = '' # 清空单包数据  
 self.head\_str = ''  
 self.recive\_len = 0 # 清空长度  
 **else**: # 未接收到报头  
 all\_len = len(self.all\_buffer) # 当前已经有的数据包长度  
 **if** all\_len >= 6: # 已经得到足够长度的报头  
 self.head\_str = self.all\_buffer[0:2] # 头部表示  
 self.data\_body\_len = (struct.unpack('<I', self.all\_buffer[2:6]))[0]  
 self.flag = True # 得到报头  
 self.recive\_len = 0 # 已接受长度0  
 self.all\_buffer = self.all\_buffer[6:] # 在缓冲区清掉头部  
 self.one\_buffer = '' # 设置单包数据缓冲区为空  
 **if** len(self.all\_buffer) >= self.data\_body\_len: # 足够得到一个完整包  
 self.one\_buffer = self.all\_buffer[:self.data\_body\_len]  
 self.all\_buffer = self.all\_buffer[self.data\_body\_len:]  
 ###  
 # 处理  
 self.body\_process(self.one\_buffer, self.head\_str, self.sock) # 处理该包  
 self.flag = False # 变成未接收到报头  
 self.one\_buffer = '' # 清空单包数据  
 self.head\_str = '' # 清除头部信息  
 self.recive\_len = 0 # 清空长度  
 **else**: # 等待下次接收数据后再解析  
 **break  
 else**: # 等待下次接收数据后再解析  
 **break**

主要是对接收到的缓冲区信息，判断是否足够得到一个完整包，如果足够，循环继续解析

如果不足以构成一个包，就等待下一次接收新数据后再解析

## 服务端文件存储说明

### online\_time.txt

存储用户在线时长

www@@166

ccc@@9836

格式：user + ‘@@’ + time(秒)

一个用户一行，每一行有一个@@分隔符，左边是用户名，右边是在线时长

单位是秒（所以用户注册时不能用@作为用户名字符）

### server\_user.txt

存储已经注册的用户名和密码

www@@123

ccc@@123

格式：user + ‘@@’ + passwd

一个用户一行，每一行有一个@@分隔符，左边是用户名，右边是密码，由于是作业

所以密码没有用加密算法