

**课 程 设 计 报 告**

**课程名称:** C#高级程序设计

**设计题目:** 3D坦克大战（服务器端）

**院（系）:** 计算机学院（软件学院）

**专业年级:**软件工程（游戏开发工程师）2017级

**学 号:** 171530425

**姓 名:** 徐 可 可

**指导教师:** 彭 伟 国

**2019 年 12 月 28 日**

目 录

[1 系统需求分析 1](#_Toc28585464)

[1.1项目意义 1](#_Toc28585465)

[1.2系统需求分析 1](#_Toc28585466)

[**1.2.1**系统需求调查研究 1](#_Toc28585467)

[**1.2.2**系统需求调查结果 2](#_Toc28585468)

[**1.2.3**系统功能设计要求 3](#_Toc28585469)

[2系统概要设计 5](#_Toc28585470)

[2.1登录注册功能 5](#_Toc28585471)

[2.2协议设计 5](#_Toc28585472)

[2.3心跳机制 6](#_Toc28585473)

[3系统详细设计 7](#_Toc28585474)

[3.1记事本 7](#_Toc28585475)

[3.1注册功能 7](#_Toc28585476)

[3.1登录功能 7](#_Toc28585477)

[3.2系统调试及解决方法 8](#_Toc28585478)

[4系统运行结果 10](#_Toc28585479)

[**5**项目评价 14](#_Toc28585480)

[参考文献 15](#_Toc28585481)

[附录：源代码 16](#_Toc28585482)

# 1 系统需求分析

## 1.1项目意义

对于我本人来说，开发调试一款较为复杂的项目，可以充分锻炼我各方面的能力。

首先锻炼的就是我对所使用的开发语言，C#掌握的能力。其次，由于该游戏基于Unity开发，也涉及到网络和数据库以及团队协同开发的操作，所以也会锻炼我对Unity、SQL、Git、TCP等的掌握和综合应用能力。

另外由于本游戏是开源的，任何人都能够获取到本游戏的源代码，并且里面不乏有一些优秀的设计之处，所以能够给其他开发者提供借鉴。

## 1.2系统需求分析

**1.2.1**系统需求调查研究

开发一款网络游戏，必不可少的要开发服务端，而开发服务端并不容易，首先就是需要有好的架构。如图：



图1-1 百度搜索服务器端架构

软件架构的分析，可以通过不同的层面入手。比较经典的软件架构描述，包含了以下几种架构：运行时架构、逻辑架构、物理架构、数据架构、开发架构。



图1-2 常见的架构描述

**1.2.2**系统需求调查结果

服务器端软件的本质，是一个会长期运行的程序，并且它还要服务于多个不定时，不定地点的网络请求[2]。所以这类软件的特点是要非常关注稳定性和性能。这类程序如果需要多个协作来提高承载能力，则还要关注部署和扩容的便利性；同时，还需要考虑如何实现某种程度容灾需求。由于多进程协同工作，也带来了开发的复杂度，这也是需要关注的问题[1]。

功能约束，是架构设计决定性因素。一个万能的架构，必定是无能的架构。一个优秀的架构，则是正好把握了对应业务领域的核心功能产生的。游戏领域的功能特征，于服务器端系统来说，非常明显的表现为几个功能的需求：

* 对于游戏数据和玩家数据的存储
* 数据解析存储
* 对玩家客户端进行数据广播
* 信息同步
* 把一部分游戏逻辑在服务器上运算，便于游戏更新内容，以及防止外挂。
* 信息校验

针对以上的需求特征，在服务器端软件开发上，我们往往会关注软件对电脑内存和CPU的使用，以求在特定业务代码下，能尽量满足承载量和响应延迟的需求。最基本的做法就是“时空转换”，用各种缓存的方式来开发程序，以求在CPU时间和内存空间上取得合适的平衡。在CPU和内存之上，是另外一个约束因素：网卡。网络带宽直接限制了服务器的处理能力，所以游戏服务器架构也必定要考虑这个因素。

对于游戏服务器架构设计来说，最重要的是利用游戏产品的需求约束，从而优化出对此特定功能最合适的“时—空”架构。并且最小化对网络带宽的占用。

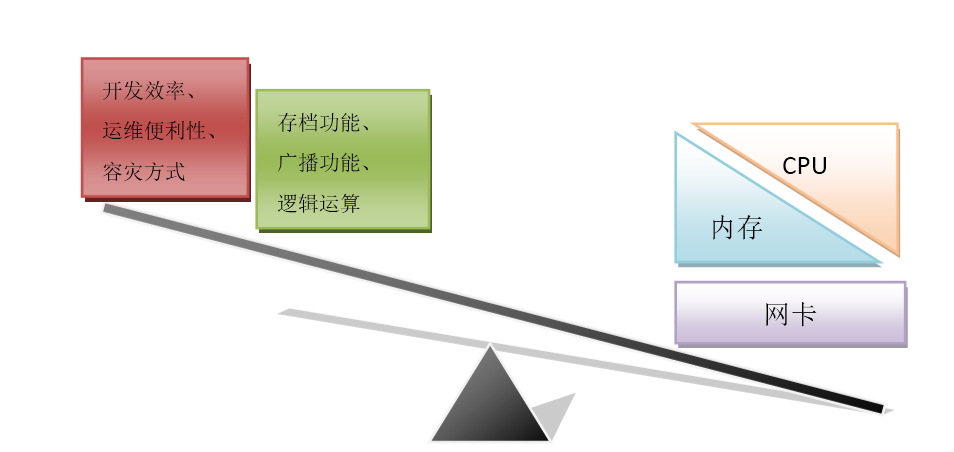


图1-3 游戏服务器的分析模型

**1.2.3**系统功能设计要求

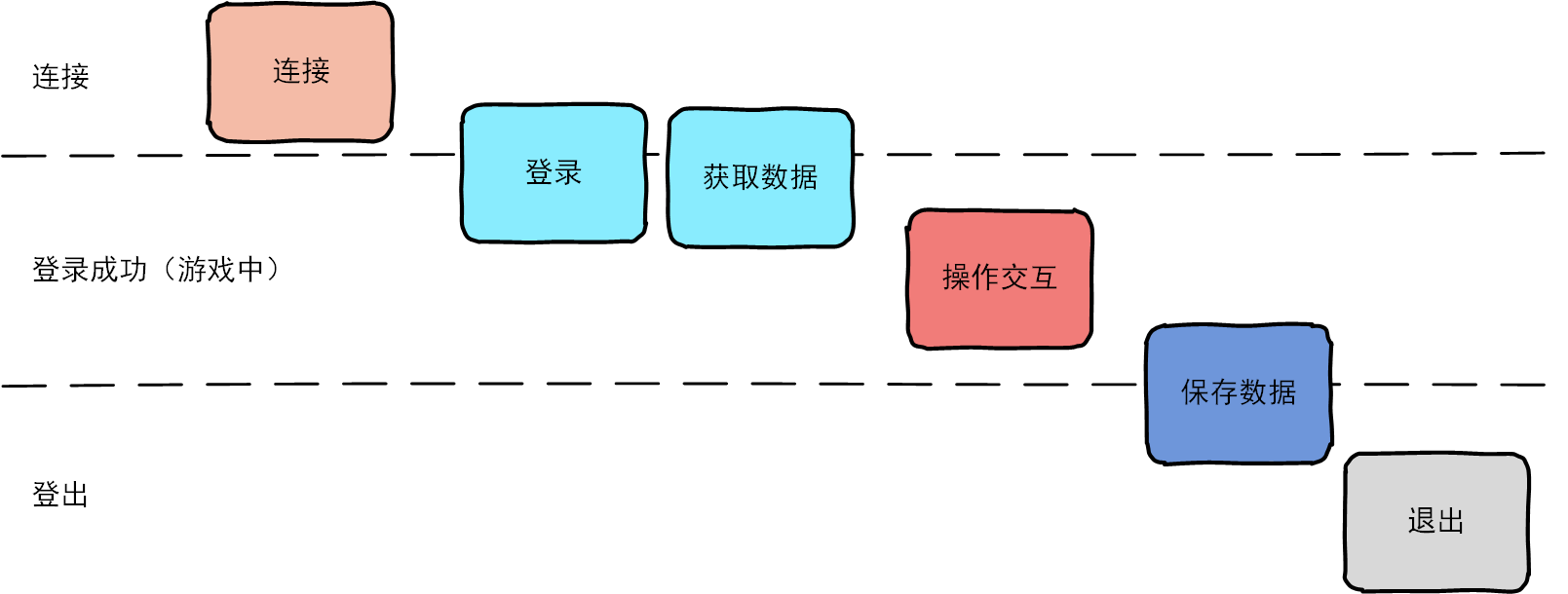
从服务端的角度看，一个玩家会经历连接、登录、获取数据、操作交互、保存数据和退出六个阶段，如图1-4所示。

图1-4 游戏流程

* **连接阶段**：客户端调用Connect连接服务端即为连接阶段。连接后双端即可通信，但服务端还不知道玩家控制的是哪个角色。于是客户端需要发送一条登录协议，协议中包含用户名、密码等信息，待检验通过后服务端会将网络连接与游戏角色对应起来，从数据库中获取该角色的数据后，才算登录成功。
* **交互阶段**：双端互通协议。MsgMove、MsgAttack，记事本程序的保存文本功能，都发生在这一阶段。
* **登出阶段**：玩家下线，服务端把玩家的数据保存到数据库中。对于保存玩家数据的时机，不同的服务端会有不同实现。有些服务端采用定时存储的方式，每隔几分钟把在线玩家的数据写回数据库；有些服务端采用下线时存储的方式，只有在玩家下线时才保存数据。上述方式各有优缺点，定时存储相对于下线时存储安全，在服务端突然挂掉的情况下，能够挽回一部分在线玩家数据，但也因为要频繁写数据库，性能较差。本系统采用玩家下线时才保存数据的方式。

对应于上述几个步骤，一个连接会有“连接但未登录”和“登录成功”两种状态，如表1-1所示。

表1-1 连接状态

# 2系统概要设计

## 2.1登录注册功能

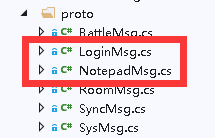
在服务端程序中为proto文件夹添加LoginMsg.cs和NotepadMsg.cs两个文件，用于定义登录和记事本相关的协议，如图2-1所示。

图2-1 添加LoginMsg.cs和NotepadMsg.cs两个文件

流程图，类图，说明

LoginMsg中包含了注册、登录和踢出三条协议。MsgRegister即注册协议，客户端需要发送id和pw字段，指定要注册的用户名和密码。服务端处理消息后，也会给客户端回应MsgRegister协议，如果服务端回应的result为0，代表注册成功，如果为1。

MsgLogin即登录协议，客户端也需要发送id和pw字段，指定要登录的用户名及其密码。服务端收到消息后，会判断密码是否正确，然后加载玩家数据，回应客户端。如果服务端回应的result为0,代表登录成功，如果为1，代表登录失败。

MsgKick是由服务端推送的“强制下线”协议。游戏中常有多个客户端同时登录同一个账号的情况，后登录的客户端会把早前登录客户端踢下线。服务端会给早前登录的客户端推送MsgKick协议，指明被踢下线的原因。

## 2.2协议设计

打开房间列表面板后，面板左侧显示玩家的战绩（总胜利次数和总失败次

数），需要定义查询战绩的协议MsgGetAchieve；面板右侧显示了房间列表，

需要获取房间列表的协议MsgGetRoomList；面板中有“新建房间”和“加入

房间”按钮，涉及MsgCreateRoom和MsgEnterRoom两条协议；若玩家加入

房间，需要获取房间信息(MsgGetRoomlnfo协议）；

玩家还可以选择离开房间(MsgLeaveRoom协议）或者开始战斗(MsgStartBattle)。综上，设计下面七条用于房间系统的协议。

查询战绩MsgGetAchieve协议：服务端收到MsgGetAchieve协议后，返回玩家的总胜利次数win和总失败次数lost。

查询房间列表MsgGetRoomList协议：服务端收到MsgGetRoomList协议后，会将所有房间信息发送给客户端。协议类包含Roomlnfo类型的数组，而Roomlnfo类包含了房间的各种信息，包括序号(id)、人数(count)、状态(status)。status为0代表“准备中”状态，status为1代表“开战中”状态。Roomlnfo由“[System.Serializable]”修饰，代表这个类是可以被序列化的。只有加上这个修饰符，Unity的JsonUtility才能够正确解析rooms数组。

创建房间MsgCreateRoom协议：服务端收到 MsgCreateRoom 协议后， 会创建一个新的房间并把玩家添加到新的房间里。返回值 result 代表执行结果，result为0 代表创建成功，其他数值代表创建失败。例如，如果玩家已经加入别的房间中，便不能创建新房间。

进入房间MsgEnterRoom协议：玩家请求加入房间时将房间序号 (id) 发送给服务端，服务端把玩家添加到房间中。服务端的返回值result代表执行结果，result为0代表成功进入，其他数值代表进入失败。例如玩家已经在房间中，就不能重复进入。

## 2.3心跳机制

断开连接时， 主动方会给对端发送FIN 信号，开启4次挥手流程。但在某些情况下，比如拿着手机进入没有信号的山区，更极端的，比如有人拿剪刀把网线剪断。虽然断开了连接，但主动方无法给对端发送FIN信号（网线剪断了还能干什么？）， 对端会认为连接有效，一直占用系统资源。

TCP 有一个连接检测机制， 就是如果在指定的时间内没有数据传送，会给对端发送一个信号（通过SetSocketOption 的KeepAlive 选项开启）。对端如果收到这个信号，回送一个TCP 的信号，确认已经收到，这样就知道此连接通畅。如果一段时间没有收到对方的响应，会进行重试，重试几次后，会认为网络不通，关闭socket。

游戏开发中，TCP 默认的KeepAlive 机制很“鸡肋＂，因为上述的“一段时间”太长，默认为2小时。一般会自行实现心跳机制。心跳机制是指客户端定时（比如每隔l分钟）向服务端发送PfNG消息，服务端收到后回应PONG 消息。服务端会记录客户端最后一次发送PING消息的时间，如果很久没有收到（比如3分钟），就假定连接不通，服务端会关闭连接，释放系统资源。后续章节“客户端网络模块”和“服务端框架”会有心跳机制的具体实现。

心跳机制也有缺点，比如在短暂的故障期间，它们可能引起一个良好连接被释放；PING 和PONG 消息占用了不必要的宽带；在流量如黄金的移动网络中， 会让玩家花费更多的流量费。

# 3系统详细设计

## 3.1记事本

在NotepadMsg.cs中编写读取和保存记事本的协议，客户端发送MsgGetText 协议后，服务端会返回带有test字段的同名协议，返回记事本文本。编辑完文本后，玩家点击保存按钮，客户端会发送MsgSaveText协议，并将修改后的文本以text字段发送给服务端。服务端收到后，更新文本，并返回同名协议。如果result为0，代表保存成功。

## 3.2注册功能

在服务端程序中添加LoginMsgHandle.cs和NotepadMsgHandle.cs两个文件， 用于处理登录注册和记事本的协议，如图3-1所示。

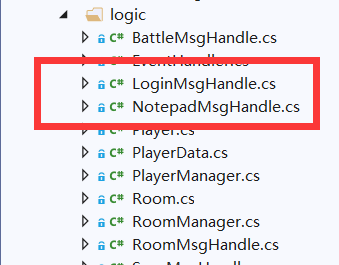


图3-1 添加LoginMsgHandle.cs和NotepadMsgHandle.cs两个文件

在LoginMsgHandle中编写MsgHandler类(partial class MsgHandler)，添加处理注册协议的方法MsgRegister。MsgRegister会调用DbManager.Register向account 表写入账号信息，再使用DbManager.CreatePlayer向game表写入默认的角色信息。最后调用NetManager. Send返回协议给客户端。

## 3.3登录功能

添加处理登录协议的方法MsgLogin,它相对复杂，因为要处理下面几项任务。

1. 验证密码：通过DbManager. CheckPassword验证用户名和密码，如果密码错误，返回result=1给客户端。
2. 状态判断：如果该客户端已经登录，不能重复登录。
3. 踢下线：通过PlayerManager.IsOnline判断该账户是否已经登录，如果已经登录，需要先把它踢下线。程序会通过PlayerManager.GetPlayer(msg.id) 获取已登录的玩家对象，给它发送MsgKick协议，通知被踢下线的客户端。最后调用NetManager.Close关闭Socket连接。
4. 读取数据：通过DbManager.GetPlayerData从数据库中读取玩家数据。
5. 构建Player：根据读取到的数据，构建player对象，并把它添加到PlayerManager的列表中，将客户端信息ClientState和player对象关联起来。

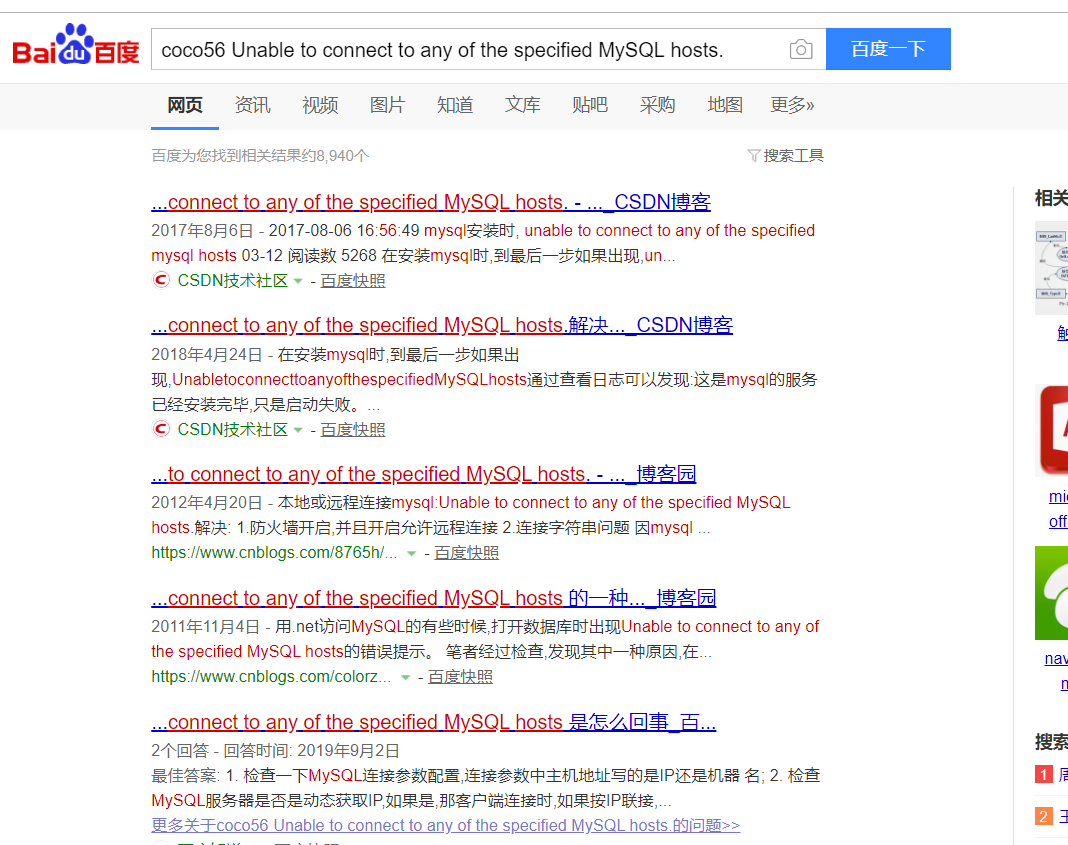
## 3.4系统调试及解决方法

实现过程中曾出现以下调试错误信息：

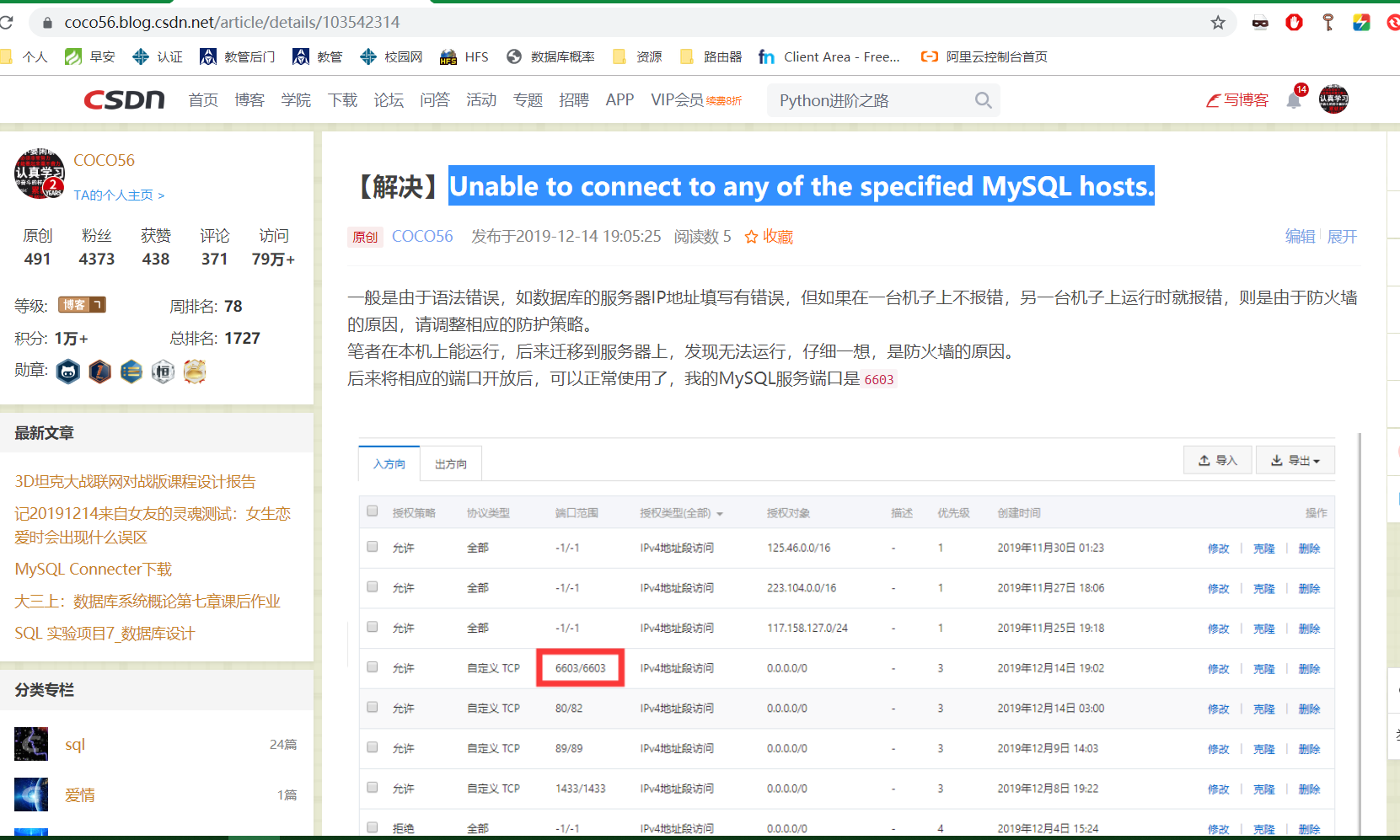
(1)、Unable to connect to any of the specified MySQL hosts.

解决方法：

通过百度搜索自己的曾经写的博文，成功解决问题。



3-2 百度搜索结果



3-3 本人自己之前写的博客

# 4系统运行结果

打开服务端应用程序，会看到如图4-1所示的界面和提示，此时代表连接数据库成功，并成功监听相应的端口，如果出错，则会闪退。

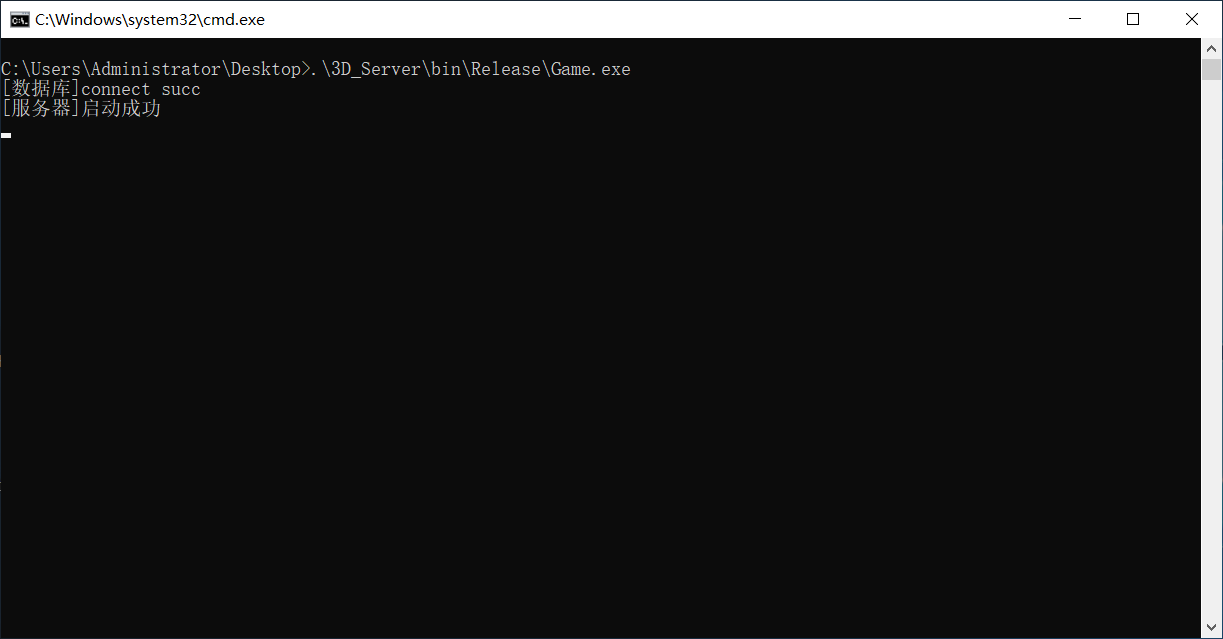


图4-1 启动成功时的提示

当有新客户端到达时，会有如图4-2所示的提示，图4-2所示的Accept 125.46.3.236:32323意为客户端的IP为125.46.3.236端口号为32323。

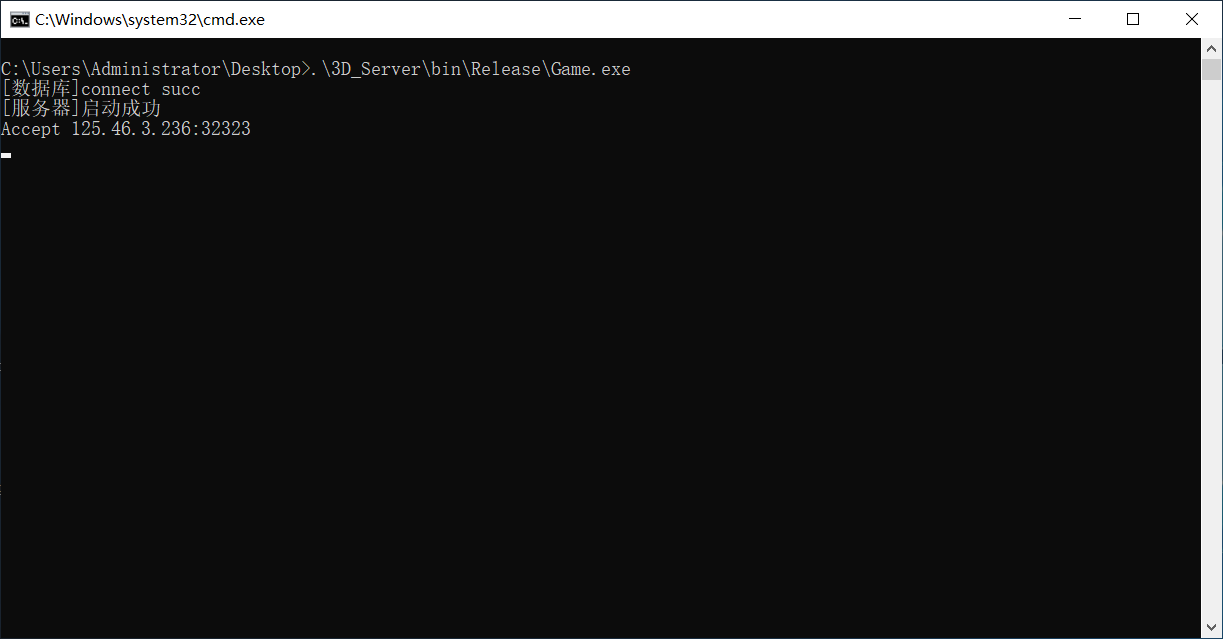


图4-2 有新客户端到达时的提示

当有客户端关闭时会有如图4-3所示的提示，图4-3所示的Socket Close 125.46.3.236:32323意为客户端的IP为125.46.3.236端口号为32323。

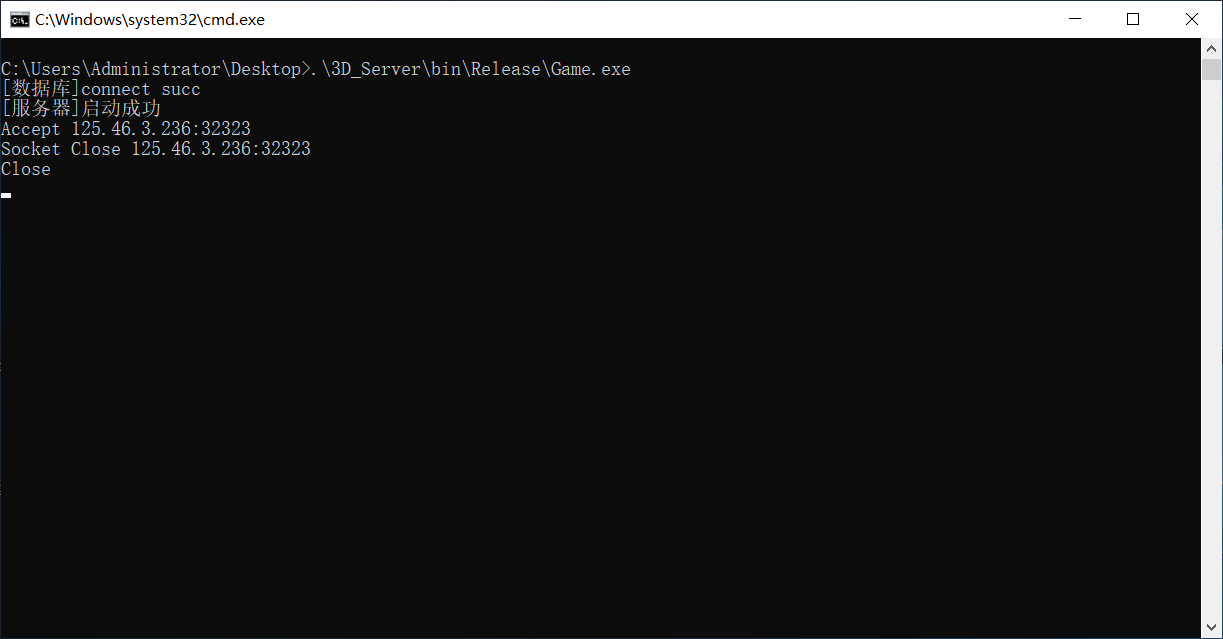


图4-3 客户端关闭时的提示

当客户端成功登录时，会自动进入房间。提示如图4-4所示。另外当客户端打开房间列表面板后，面板左侧会显示玩家的战绩（总胜利次数和总失败次数），因此进入房间时会请求查询战绩（MsgGetAchieve）；客户端面板右侧显示了房间列表，因此还会请求获取房间列表（MsgGetRoomList）；

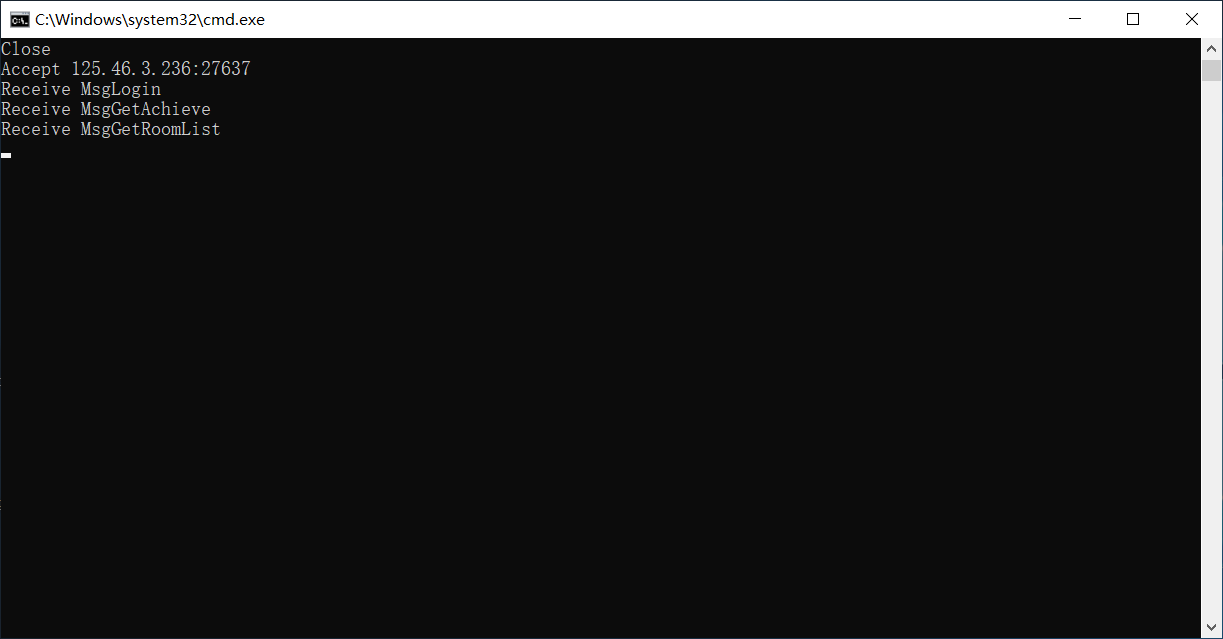


图4-4 客户端请求获取房间列表的提示

如果玩家拿着手机进入没有信号的山区，或者有人拿剪刀剪断网线，都会导致链路不通。但TCP本身的心跳机制太＂鸡肋＂，要经过2个小时的时间才能主动释放资源，游戏程序一般都会自行实现心跳机制。具体来说就是，客户端会定时（如30秒）给服务端发送PING协议，服务端收到后会回应PONG协议。正常情况下，客户端每隔一段时间（如30秒）必然会收到服务端的PONG协议（就算网络不通畅，最慢120秒也总该收到了吧）。如果客户端很长时间（如120秒）没有收到PONG协议，很大概率是网络不通畅或服务端挂掉，客户端程序可以释放Socket资源。其实对于客户端来说，释放不释放关系不大，毕竞只有一个Socket。但对服务端来说却很重要，因为服务端可能保持着数以万计的连接，当游戏在线人数很多时，只有及时释放资源，才能让玩家正常玩游戏[3]（不然，内存爆满服务器挂掉大家都玩不了）。所以客户端会定时向服务端发送MsgPing协议，服务端收到后也会回应MsgPong协议。

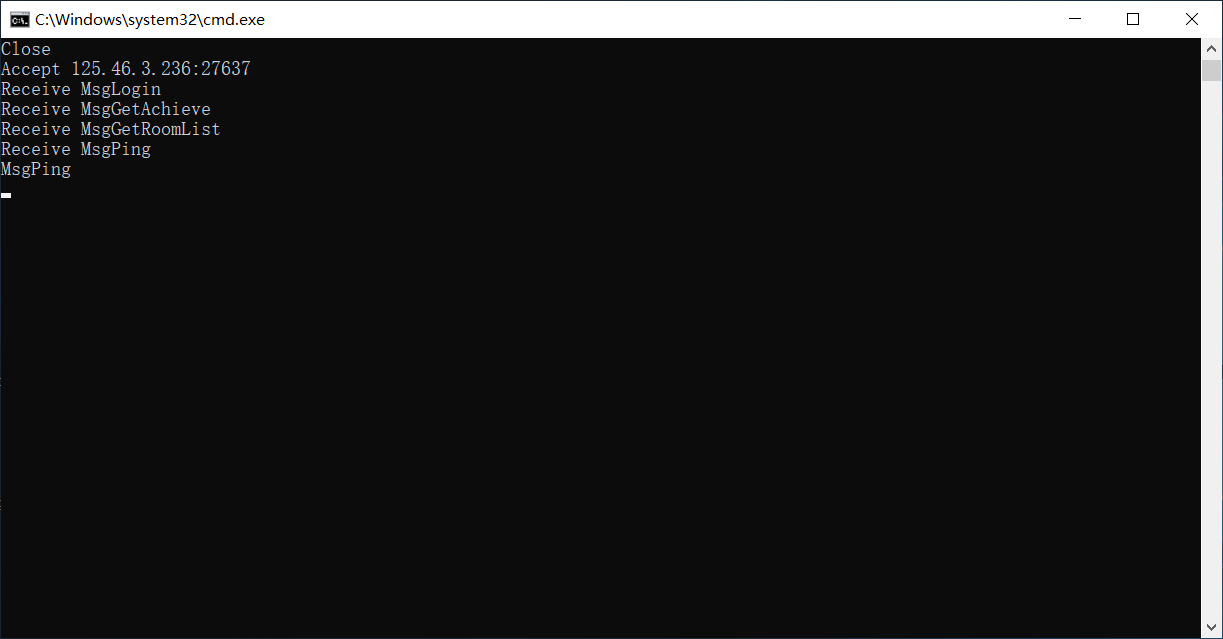


图4-5 客户端与服务端进行“乒乓”时的提示

客户端登录后面板中有“新建房间”和“加入房间”按钮，涉及MsgCreateRoom和MsgEnterRoom两条协议；若玩家加入房间，需要获取房间信息(MsgGetRoomlnfo协议）；

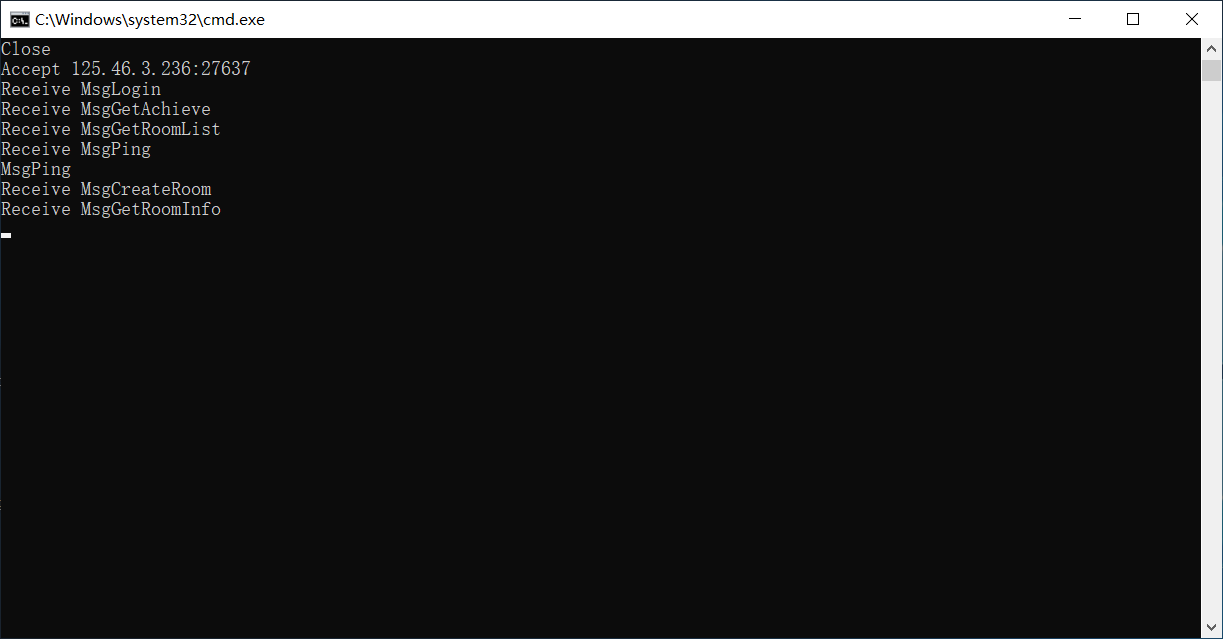


图4-6 客户端创建房间并获取房间信息的提示

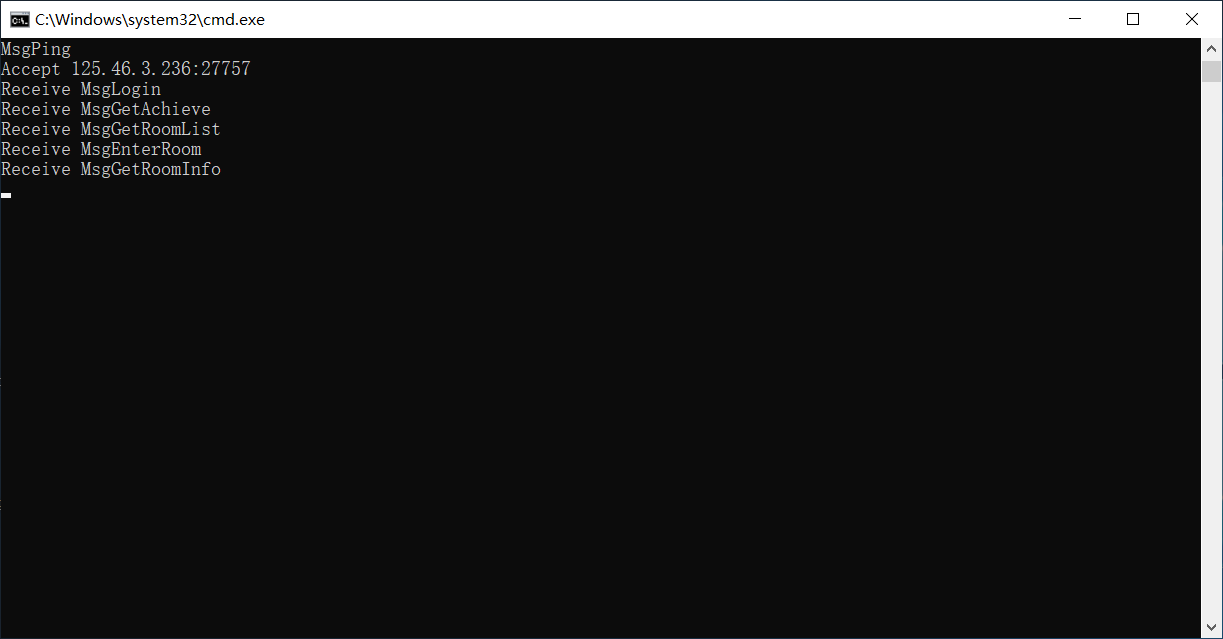


图4-7 客户端进入房间并获取房间信息的提示

战斗过程中，程序会通过MsgSyncTank,MsgFire,MsgHit等协议去同步坦克的位置、炮弹位置等信息[4]。当某个阵营取得胜利，服务端会广播MsgBattleResult协议，通知客户端哪个阵营获得了胜利。

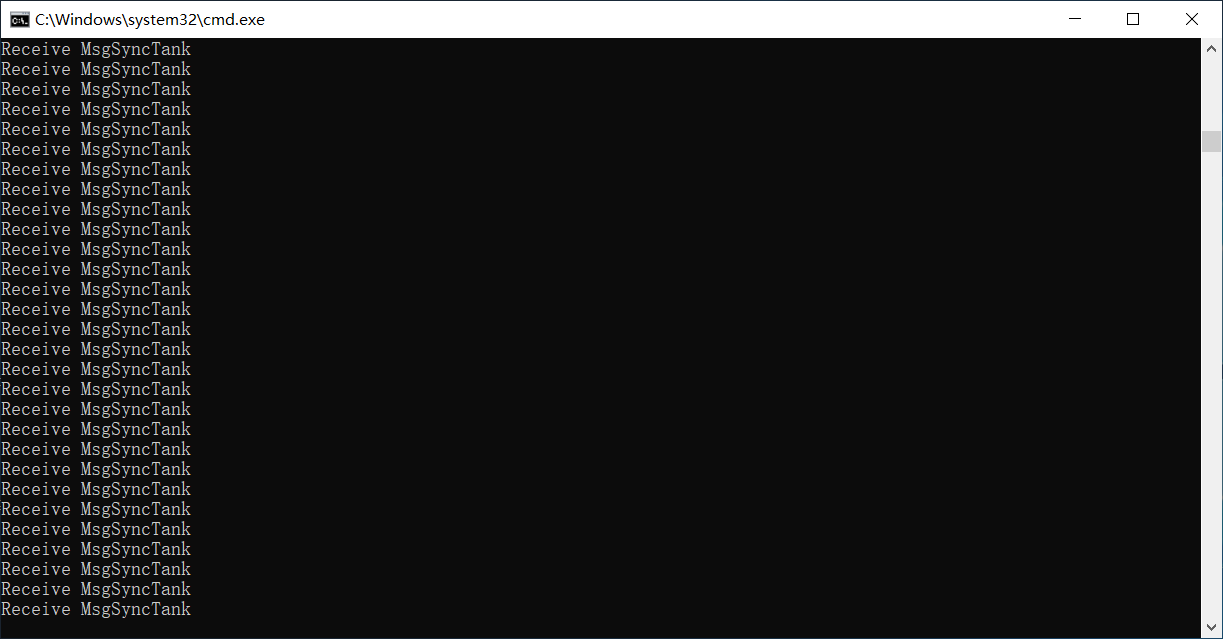


图4-8 客户端发送同步坦克信息时的提示

**5**项目评价

本游戏服务器的设计完成了基本的设计要求，但还有一些地方需要优化，比如如何处理高并发，以及如何支持在服务器集群上运行[5]。另外为了方便用户登录，也可以接入第三方服务，比如接入QQ快捷登录，这样用户登录起来就比较方便了，无需专门注册我们的游戏账号，直接用QQ扫一下二维码就能自动登录或注册了。

具体地：

由于本游戏不分区不分服，所以在设计服务器的时候，应按世界服的思想去设计，即服务器是一个n多台物理机的集群。当用户登陆服务器，创建房间时，可能根据负载均衡算法，它可以在任何一台服务器上面。这样，不管用户登陆到哪一台服务器上面了，都可以获得自己的数据。可以使用redis来做数据共享。

* 在同一局游戏中，应要求所有人都在同一个房间中，可以规定在同一个房间中的用户，必须登陆到同一台物理服务器上面。在创建房间完成之后，其他人根据房间号查找房间的时候，就可以根据房间号，获取这个房间所在的服务器ip和端口，判断一个当前用户登陆的服务器ip与房间所在的服务器ip是否相同，如果相同，就不做切换，如果不一样，客户端就使用ip和端口，连接到房间所在的服务器上面。
* 创建房间成功之后，接下来的操作都要保证它的顺序性，所以房间需要有一个它自己的消息队列。可以把每个房间到达服务器的消息封装为一个任务，把这个任务放到消息队列中，然后有一个任务执行者去按顺序执行这些任务。
* 在用户登录时一般都是需要接第三方登陆（如使用QQ登录），登陆这一块是http操作，我们统一提供一个web服务，用来做登陆验证。因为在登陆时，调用第三方的http服务，这个过程可能很慢，如果放在逻辑服务器的话，可能会卡业务逻辑任务。因为可能不同的玩家业务请求可能同在一个线程中，如果有任务卡了，那么这个任务以后新来的请求请会卡住，导致消息延迟。

# 参考文献

[1]邹晓峰,刘同强,周玉龙,李拓,李仁刚,公维锋.一种云服务器互连芯片交叉开关的设计与实现[J].信息技术,2019,43(12):6-10+14

[2]崔希进. 客户服务器模式下分布式物业管理系统的设计与实现[D].哈尔滨工程大学,2003.

[3]朱利文. 基于Android的小区物业管理系统的设计与实现[D].西安电子科技大学,2017.

[4]徐冶楠. 高性能M2M业务能力服务器的设计与实现[D].北京邮电大学,2015.

[5]郑智斌. 基于Erlang的移动互联网SNS游戏服务器研究与实现[D].中山大学,2012.ddd

附录：源代码

* 3D\_Server\script\logic\Player.cs

1. /// <summary>
2. /// 接口中的方法没有访问修饰符，默认是Public,方法没有定义(不带花括号)
3. /// 静态方法不能实现接口方法
4. /// </summary>
5. public interface IPlayer
6. {
7. /// <summary>
8. /// 约定每位玩家应该可以发送信息
9. /// </summary>
10. /// <param name="msgBase"></param>
11. void Send(MsgBase msgBase);
12. }
13. public class Player: IPlayer
14. {
15. //id
16. public string id = "";
17. //指向ClientState
18. public ClientState state;
19. //构造函数
20. public Player(ClientState state){
21. this.state = state;
22. }
23. //坐标和旋转
24. public float x;
25. public float y;
26. public float z;
27. public float ex;
28. public float ey;
29. public float ez;
30. //在哪个房间
31. public int roomId = -1;
32. //阵营
33. public int camp = 1;
34. //坦克生命值
35. public int hp = 100;
36. //数据库数据
37. public PlayerData data;
38. //发送信息
39. public void Send(MsgBase msgBase){
40. NetManager.Send(state, msgBase);
41. }
42. }

* 3D\_Server\Program.cs

1. using System;
2. namespace Game
3. {
4. class MainClass
5. {
6. public static void Main (string[] args)
7. {
8. //连接数据库
9. ///匿名委托
10. if (!ConnectDB(delegate () { Show(); })){
11. Console.ReadKey();
12. return;
13. }
14. //启动程序并监听82端口
15. NetManager.StartLoop(82);
16. }
17. public static bool ConnectDB(System.Action action)
18. {
19. //invoke表是同步执行指定的委托
20. action.Invoke();
21. return DbManager.getIns().Connect(DBConfiguration.db, DBConfiguration.ip, DBConfiguration.port, DBConfiguration.user, DBConfiguration.pw);
22. }
23. public static void Show()
24. {
25. Console.WriteLine("[数据库]正在连接数据库");
26. }
27. }
28. }

* 3D\_Server\script\net\NetManager.cs

1. using System;
2. using System.Net;
3. using System.Net.Sockets;
4. using System.Collections.Generic;
5. using System.Reflection;
6. class NetManager
7. {
8. //监听Socket
9. public static Socket listenfd;
10. //客户端Socket及状态信息
11. public static Dictionary<Socket, ClientState> clients = new Dictionary<Socket, ClientState>();
12. //Select的检查列表
13. static List<Socket> checkRead = new List<Socket>();
14. //ping间隔
15. public static long pingInterval = 30;
16. public static void StartLoop(int listenPort)
17. {
18. //Socket
19. listenfd = new Socket(AddressFamily.InterNetwork,
20. SocketType.Stream, ProtocolType.Tcp);
21. //Bind
22. IPAddress ipAdr = IPAddress.Parse("0.0.0.0");
23. IPEndPoint ipEp = new IPEndPoint(ipAdr, listenPort);
24. listenfd.Bind(ipEp);
25. //Listen
26. listenfd.Listen(0);
27. Console.WriteLine("[服务器]启动成功");
28. //循环
29. while(true){
30. ResetCheckRead();  //重置checkRead
31. Socket.Select(checkRead, null, null, 1000);
32. //检查可读对象
33. for(int i = checkRead.Count-1; i>=0; i--){
34. Socket s = checkRead[i];
35. if(s == listenfd){
36. ReadListenfd(s);
37. }
38. else{
39. ReadClientfd(s);
40. }
41. }
42. //超时
43. Timer();
44. }
45. }
46. //填充checkRead列表
47. public static void ResetCheckRead(){
48. checkRead.Clear();
49. checkRead.Add(listenfd);
50. foreach (ClientState s in clients.Values){
51. checkRead.Add(s.socket);
52. }
53. }
54. //读取Listenfd
55. public static void ReadListenfd(Socket listenfd){
56. try{
57. Socket clientfd = listenfd.Accept();
58. Console.WriteLine("Accept " + clientfd.RemoteEndPoint.ToString());
59. ClientState state = new ClientState();
60. state.socket = clientfd;
61. state.lastPingTime = GetTimeStamp();
62. clients.Add(clientfd, state);
63. }catch(SocketException ex){
64. Console.WriteLine("Accept fail" + ex.ToString());
65. }
66. }
67. //关闭连接
68. public static void Close(ClientState state){
69. //消息分发
70. MethodInfo mei =  typeof(EventHandler).GetMethod("OnDisconnect");
71. object[] ob = {state};
72. mei.Invoke(null, ob);
73. //关闭
74. state.socket.Close();
75. clients.Remove(state.socket);
76. }
77. //读取Clientfd
78. public static void ReadClientfd(Socket clientfd){
79. ClientState state = clients[clientfd];
80. ByteArray readBuff = state.readBuff;
81. //接收
82. int count = 0;
83. //缓冲区不够，清除，若依旧不够，只能返回
84. //当单条协议超过缓冲区长度时会发生
85. if(readBuff.remain <=0){
86. OnReceiveData(state);
87. readBuff.MoveBytes();
88. };
89. if(readBuff.remain <=0){
90. Console.WriteLine("Receive fail , maybe msg length > buff capacity");
91. Close(state);
92. return;
93. }
94. try{
95. count = clientfd.Receive(readBuff.bytes, readBuff.writeIdx, readBuff.remain, 0);
96. }catch(SocketException ex){
97. Console.WriteLine("Receive SocketException " + ex.ToString());
98. Close(state);
99. return;
100. }
101. //客户端关闭
102. if(count <= 0 ){
103. Console.WriteLine("Socket Close " + clientfd.RemoteEndPoint.ToString());
104. Close(state);
105. return;
106. }
107. //消息处理
108. readBuff.writeIdx+=count;
109. //处理二进制消息
110. OnReceiveData(state);
111. //移动缓冲区
112. readBuff.CheckAndMoveBytes();
113. }
114. //数据处理
115. public static void OnReceiveData(ClientState state){
116. ByteArray readBuff = state.readBuff;
117. //消息长度
118. if(readBuff.length <= 2) {
119. return;
120. }
121. //消息体长度
122. int readIdx = readBuff.readIdx;
123. byte[] bytes =readBuff.bytes;
124. Int16 bodyLength = (Int16)((bytes[readIdx+1] << 8 )| bytes[readIdx]);
125. if(readBuff.length < bodyLength){
126. return;
127. }
128. readBuff.readIdx +=2;
129. //解析协议名
130. int nameCount = 0;
131. string protoName = MsgBase.DecodeName(readBuff.bytes, readBuff.readIdx, out nameCount);
132. if(protoName == ""){
133. Console.WriteLine("OnReceiveData MsgBase.DecodeName fail");
134. Close(state);
135. return;
136. }
137. readBuff.readIdx += nameCount;
138. //解析协议体
139. int bodyCount = bodyLength - nameCount;
140. if(bodyCount <= 0){
141. Console.WriteLine("OnReceiveData fail, bodyCount <=0 ");
142. Close(state);
143. return;
144. }
145. MsgBase msgBase = MsgBase.Decode(protoName, readBuff.bytes, readBuff.readIdx, bodyCount);
146. readBuff.readIdx += bodyCount;
147. readBuff.CheckAndMoveBytes();
148. //分发消息
149. MethodInfo mi =  typeof(MsgHandler).GetMethod(protoName);
150. object[] o = {state, msgBase};
151. Console.WriteLine("Receive " + protoName);
152. if(mi != null){
153. mi.Invoke(null, o);
154. }
155. else{
156. Console.WriteLine("OnReceiveData Invoke fail " + protoName);
157. }
158. //继续读取消息
159. if(readBuff.length > 2){
160. OnReceiveData(state);
161. }
162. }
163. //发送
164. public static void Send(ClientState cs, MsgBase msg){
165. //状态判断
166. if(cs == null){
167. return;
168. }
169. if(!cs.socket.Connected){
170. return;
171. }
172. //数据编码
173. byte[] nameBytes = MsgBase.EncodeName(msg);
174. byte[] bodyBytes = MsgBase.Encode(msg);
175. int len = nameBytes.Length + bodyBytes.Length;
176. byte[] sendBytes = new byte[2+len];
177. //组装长度
178. sendBytes[0] = (byte)(len%256);
179. sendBytes[1] = (byte)(len/256);
180. //组装名字
181. Array.Copy(nameBytes, 0, sendBytes, 2, nameBytes.Length);
182. //组装消息体
183. Array.Copy(bodyBytes, 0, sendBytes, 2+nameBytes.Length, bodyBytes.Length);
184. //为简化代码，不设置回调
185. try{
186. cs.socket.BeginSend(sendBytes,0, sendBytes.Length, 0, null, null);
187. }catch(SocketException ex){
188. Console.WriteLine("Socket Close on BeginSend" + ex.ToString());
189. }
190. }
191. //定时器
192. static void Timer(){
193. //消息分发
194. MethodInfo mei =  typeof(EventHandler).GetMethod("OnTimer");
195. object[] ob = {};
196. mei.Invoke(null, ob);
197. }
198. //获取时间戳
199. public static long GetTimeStamp() {
200. TimeSpan ts = DateTime.UtcNow - new DateTime(1970, 1, 1, 0, 0, 0, 0);
201. return Convert.ToInt64(ts.TotalSeconds);
202. }
203. }