平顶山学院

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

1.1 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。我首先使用百度搜索了一些文章，结果如图 1-1 所示：

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。我首先搜到了一些比较经典的软件架构：运行时架构、逻辑架构、物理架构、数据架构、开发架构，具体介绍如图 1-2 所示，我在开发这款游戏服务器程序时主要关注了软件的逻辑架构。

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：

- 数据解析存储
- 信息同步
- 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。

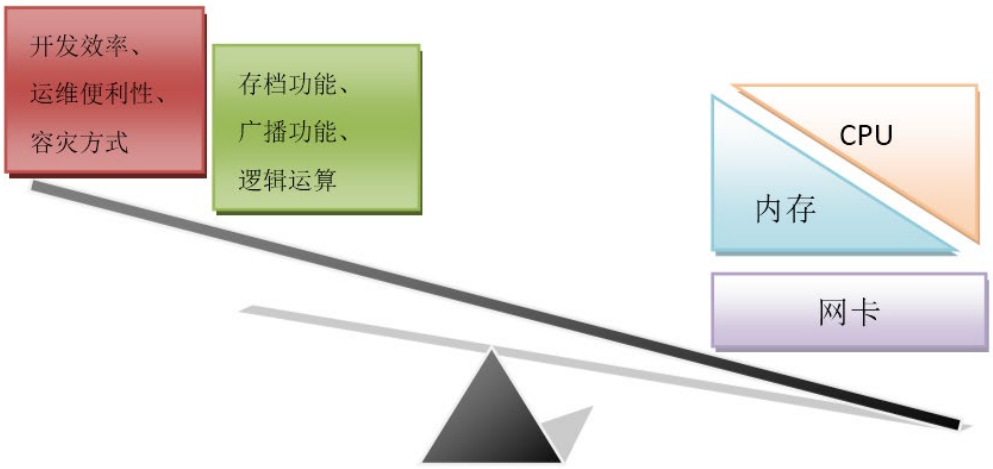


图 1-3 游戏服务器的分析模型

1.2.3 系统功能设计要求

□ 数据解析存储，主要涉及到对游戏数据和玩家数据的存储；信息同步，主要是对玩家客户端进行数据广播；信息校验，主要是对玩家身份进行校验，比如对登录信息进行校验，除此之外也可以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。

另外从服务端的角度看，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。

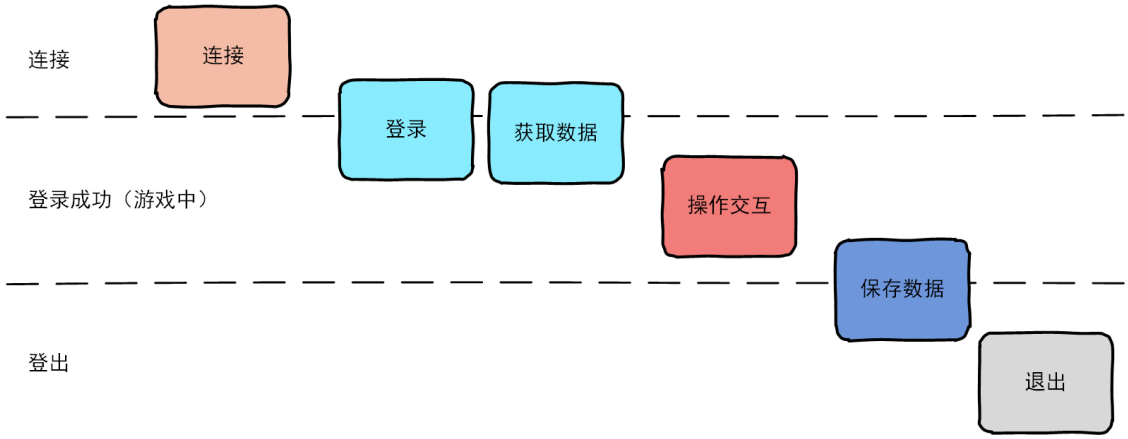


图 1-4 游戏流程

- **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

状态	说明
连接但未登录	客户端连接(Connect)服务端，服务端还不知道该客户端对应哪个游戏角色。玩家需要输入用户名、密码，服务端验证后从数据库读取角色数据，把连接和角色关联起来
登录成功	连接和角色关联后，玩家可以操作游戏角色，比如打副本、吃药水

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 数据解析存储

Unity 中提供 Json 辅助类 JsonUtility，通过 JsonUtility.ToJson 和 JsonUtility.FromJson 可以实现 Json 协议的编码和解码。例如有如下的两个协议类 MsgMove 和 MsgAttack, 其中 MsgMove 包含 x 、y、z 三个成员 MsgAttack 包含 desc 个成员。

编写一个测试程序。新建一个 MsgMove 对象，给成员赋值，然后调用 JsonUtility.ToJson 可将协议类转换成字符串。Json 字符串形如“{“成员 1":值,”成员 2”:值}”，很直观。

JsonUtility.FromJson 可以将字符串转换成指定的协议对象。FromJson 的第一个参数指定要解析的字符串 s, 第二个参数指定了要还原的协议类的类型，这里使用 Type.GetType 由协议类的名字来指定类型。字符串 s 包含了一些”\”，这是因为在 C＃的字符串中，引号由”\”” 表示。

JsonUtility 有多种解码方式，FromJsonOverwrite 是另外的一种，先定义要解析的协议对象，再调用 FromJsonOverwrite 给协议对象赋值。FromJson 或 FromJsonOverwrite 还具备一定的错误处理能力。如若 msgMove.x 应是 int 型，但 s 却指定 x 的值是字符串 "hehe"，无法解码。这时，JsonUtility 会把 x 设置为默认值 0。

2.2 信息校验功能

□ 信息校验登录注册功能，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。

LoginMsg 中包含了注册、登录和踢出三条协议，每条协议我们建了一个类，类图如图 2-1 所示。

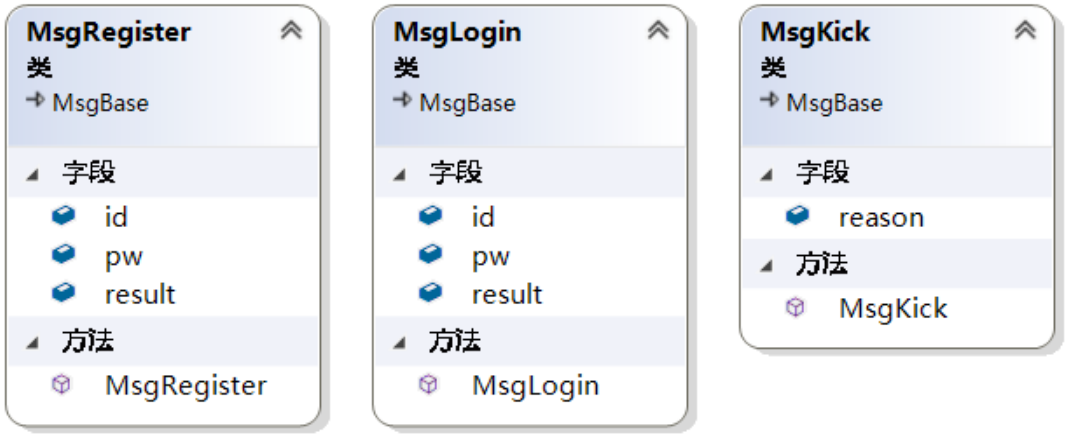


图 2-1 LoginMsg.cs 中的类

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。

在 NotepadMsg.cs 中编写读取和保存记事本的协议,每条协议我们建了一个类，类图如图 2-2 所示。

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。

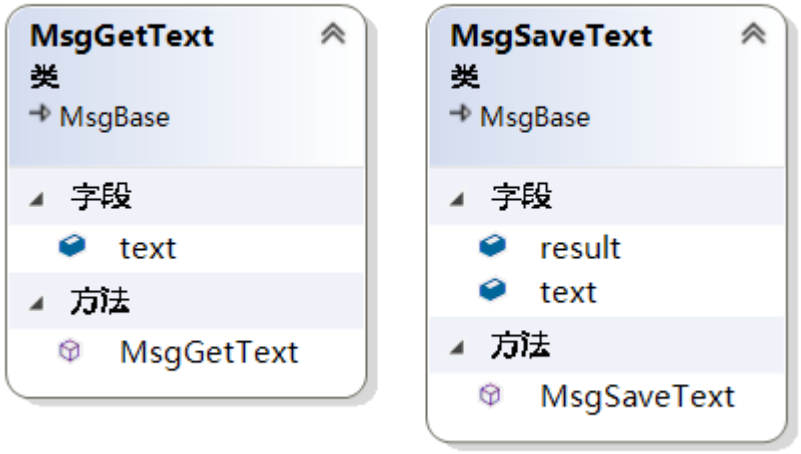


图 2-1 NotepadMsg.cs 中的类

2.3 信息同步功能

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。

3 系统详细设计

3.1 数据解析存储

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 信息校验功能

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。

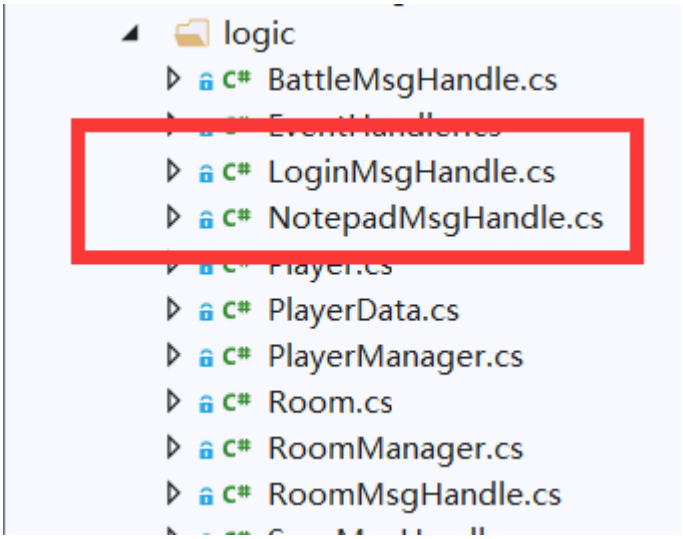


图 3-1 添加 LoginMsgHandle.cs 和 NotepadMsgHandle.cs 两个文件

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。

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。

3.3 信息同步功能

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 1 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。

3.4 系统调试及解决方法

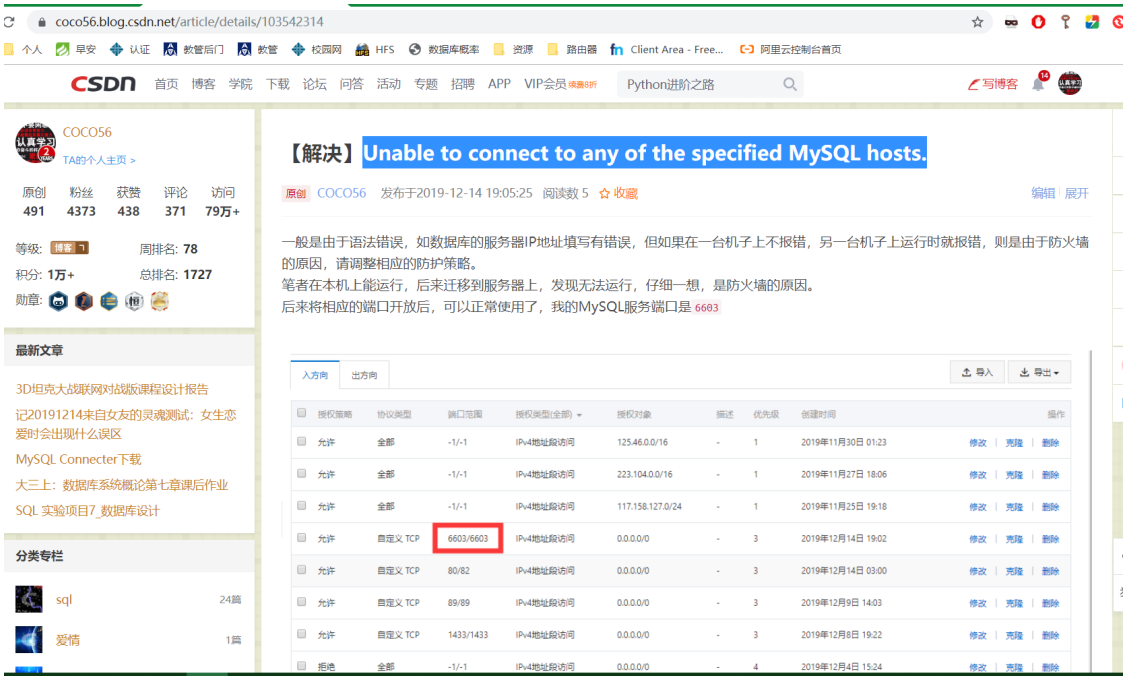
实现过程中曾出现以下调试错误信息：

(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。

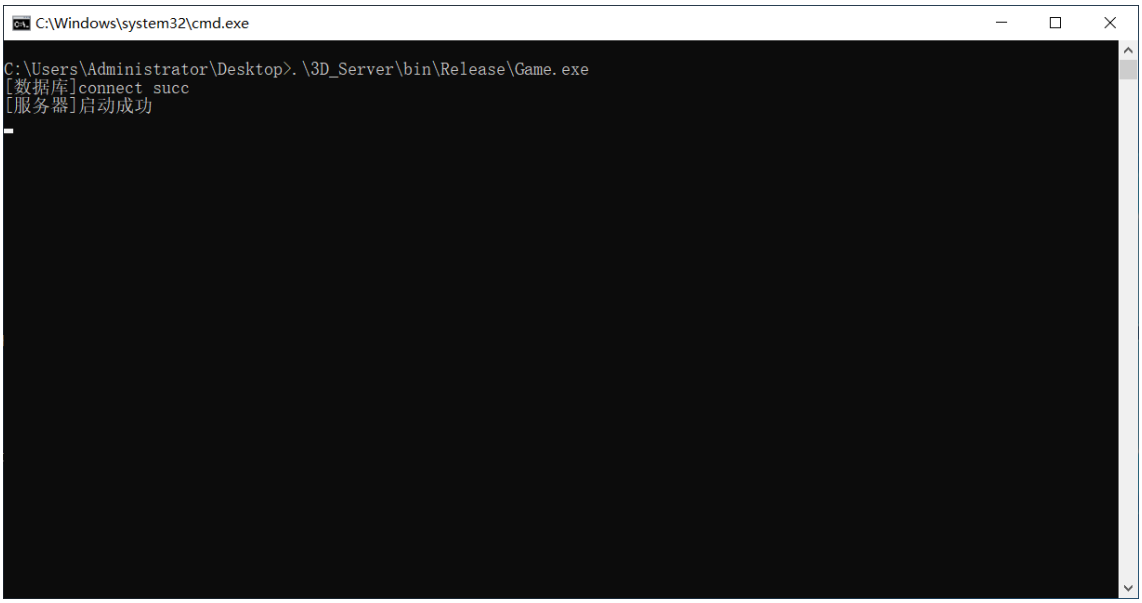


图 4-1 启动成功时的提示

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。

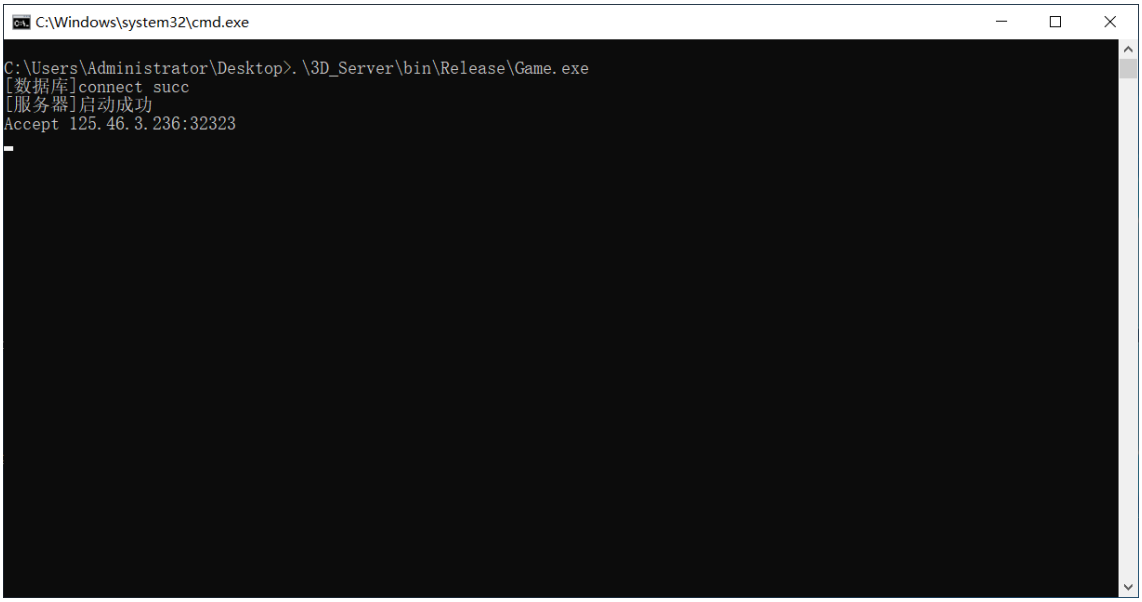


图 4-2 有新客户端到达时的提示

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。

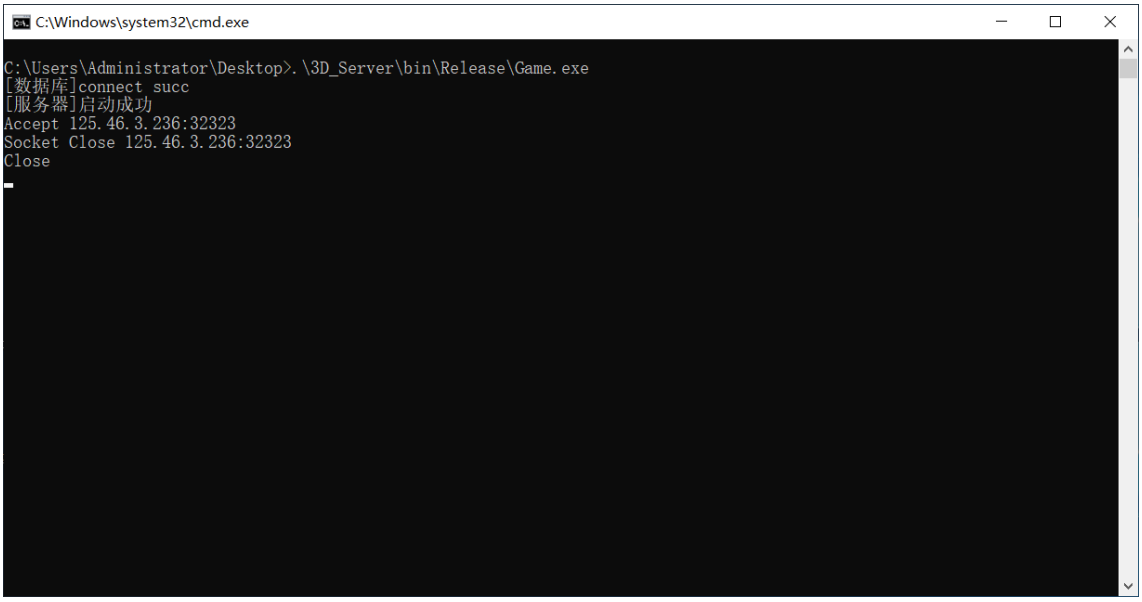


图 4-3 客户端关闭时的提示

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，毕竟只有一个

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

5.1 系统已经实现的功能

本游戏服务器程序的设计完成了基本的设计要求，完成了对数据的解析和存储，信息校验和同步功能。

“网络底层”用于处理粘包半包、协议解析等功能。消息处理模块当收到客户端的 MsgMove 协议时，服务端会在消息处理模块中记录玩家坐标，然后将 MsgMove 协议广播给所有客户端。当玩家上线，会做些初始化的操作；当玩家下线，会做些数据记录，这些都在事件处理模块中执行。数据库底层模块提供了保存玩家数据、读取玩家数据、注册、检验用户名密码是否正确等的功能，是服务端和数据库交互的一层封装。存储结构指定哪些数据需要保存，比如在线记事本中需要保存文本信息，对于大部分游戏需要存储玩家的金币、经验、等级等信息。

5.2 软件功能缺陷

本游戏服务器程序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。

5.3 进一步改进设想

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)
14) public class Player: IPlayer
15) {
16)     //id
17)     public string id = "";
18)     //指向 ClientState
19)     public ClientState state;
20)     //构造函数
21)     public Player(ClientState state){
22)         this.state = state;
23)     }
24)     //坐标和旋转
25)     public float x;
26)     public float y;
27)     public float z;
28)     public float ex;
29)     public float ey;
30)     public float ez;
31)
32)     //在哪个房间
33)     public int roomId = -1;
34)     //阵营
35)     public int camp = 1;
36)     //坦克生命值
37)     public int hp = 100;
38)
```

```
39)     //数据库数据
40)     public PlayerData data;
41)
42)     //发送信息
43)     public void Send(MsgBase msgBase){
44)         NetManager.Send(state, msgBase);
45)     }
46) }
```

- 3D_Server\Program.cs

```
1)  using System;
2)
3)  namespace Game
4)  {
5)      class MainClass
6)      {
7)          public static void Main (string[] args)
8)          {
9)              //连接数据库
10)             ///匿名委托
11)             if (!ConnectDB(delegate () { Show(); })){
12)                 Console.ReadKey();
13)                 return;
14)             }
15)
16)             //启动程序并监听 82 端口
17)             NetManager.StartLoop(82);
18)         }
19)
20)         public static bool ConnectDB(System.Action action)
21)         {
22)             //invoke 表是同步执行指定的委托
23)             action.Invoke();
24)             return DbManager.getIns().Connect(DBConfiguration.db,
    DBConfiguration.ip, DBConfiguration.port, DBConfiguration.user, DB
    Configuration.pw);
25)         }
26)
27)         public static void Show()
28)         {
29)             Console.WriteLine("[数据库]正在连接数据库");
30)         }
31)     }
32) }
```

- 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)
7)  class NetManager
8)  {
9)      //监听 Socket
10)     public static Socket listenfd;
11)     //客户端 Socket 及状态信息
12)     public static Dictionary<Socket, ClientState> clients = new Di
    ctionary<Socket, ClientState>();
13)     //Select 的检查列表
14)     static List<Socket> checkRead = new List<Socket>();
15)     //ping 间隔
16)     public static long pingInterval = 30;
17)
18)     public static void StartLoop(int listenPort)
19)     {
20)         //Socket
21)         listenfd = new Socket(AddressFamily.InterNetwork,
22)             SocketType.Stream, ProtocolType.Tcp);
23)         //Bind
24)         IPAddress ipAdr = IPAddress.Parse("0.0.0.0");
25)         IPEndPoint ipEp = new IPEndPoint(ipAdr, listenPort);
26)         listenfd.Bind(ipEp);
27)         //Listen
28)         listenfd.Listen(0);
29)         Console.WriteLine("[服务器]启动成功");
30)         //循环
31)         while(true){
32)             ResetCheckRead();  //重置 checkRead
33)             Socket.Select(checkRead, null, null, 1000);
34)             //检查可读对象
35)             for(int i = checkRead.Count-1; i>=0; i--){
36)                 Socket s = checkRead[i];
37)                 if(s == listenfd){
38)                     ReadListenfd(s);
39)                 }
40)                 else{
41)                     ReadClientfd(s);
```

```
42)                 }
43)             }
44)             //超时
45)             Timer();
46)         }
47)     }
48)
49)     //填充 checkRead 列表
50)     public static void ResetCheckRead(){
51)         checkRead.Clear();
52)         checkRead.Add(listenfd);
53)         foreach (ClientState s in clients.Values){
54)             checkRead.Add(s.socket);
55)         }
56)     }
57)
58)     //读取 Listenfd
59)     public static void ReadListenfd(Socket listenfd){
60)         try{
61)             Socket clientfd = listenfd.Accept();
62)             Console.WriteLine("Accept " + clientfd.RemoteEndPoint.
    ToString());
63)             ClientState state = new ClientState();
64)             state.socket = clientfd;
65)             state.lastPingTime = GetTimeStamp();
66)             clients.Add(clientfd, state);
67)         }catch(SocketException ex){
68)             Console.WriteLine("Accept fail" + ex.ToString());
69)         }
70)     }
71)
72)     //关闭连接
73)     public static void Close(ClientState state){
74)         //消息分发
75)         MethodInfo mei =  typeof(EventHandler).GetMethod("OnDiscon
    nect");
76)         object[] ob = {state};
77)         mei.Invoke(null, ob);
78)         //关闭
79)         state.socket.Close();
80)         clients.Remove(state.socket);
81)     }
82)
```

```
83)     //读取 Clientfd
84)     public static void ReadClientfd(Socket clientfd){
85)         ClientState state = clients[clientfd];
86)         ByteArray readBuff = state.readBuff;
87)         //接收
88)         int count = 0;
89)         //缓冲区不够，清除，若依旧不够，只能返回
90)         //当单条协议超过缓冲区长度时会发生
91)         if(readBuff.remain <=0){
92)             OnReceiveData(state);
93)             readBuff.MoveBytes();
94)         };
95)         if(readBuff.remain <=0){
96)             Console.WriteLine("Receive fail , maybe msg length > b
    uff capacity");
97)             Close(state);
98)             return;
99)         }
100)            try{
101)                count = clientfd.Receive(readBuff.bytes, readBuff.
    writeIdx, readBuff.remain, 0);
102)            }catch(SocketException ex){
103)                Console.WriteLine("Receive SocketException " + ex.
    ToString());
104)                Close(state);
105)                return;
106)            }
107)            //客户端关闭
108)            if(count <= 0 ){
109)                Console.WriteLine("Socket Close " + clientfd.Remot
    eEndPoint.ToString());
110)                Close(state);
111)                return;
112)            }
113)            //消息处理
114)            readBuff.writeIdx+=count;
115)            //处理二进制消息
116)            OnReceiveData(state);
117)            //移动缓冲区
118)            readBuff.CheckAndMoveBytes();
119)        }
120)
121)        //数据处理
```

```
122)        public static void OnReceiveData(ClientState state){
123)            ByteArray readBuff = state.readBuff;
124)            //消息长度
125)            if(readBuff.length <= 2) {
126)                return;
127)            }
128)            //消息体长度
129)            int readIdx = readBuff.readIdx;
130)            byte[] bytes =readBuff.bytes;
131)            Int16 bodyLength = (Int16)((bytes[readIdx+1] << 8 )| b
    ytes[readIdx]);
132)            if(readBuff.length < bodyLength){
133)                return;
134)            }
135)            readBuff.readIdx +=2;
136)            //解析协议名
137)            int nameCount = 0;
138)            string protoName = MsgBase.DecodeName(readBuff.bytes,
    readBuff.readIdx, out nameCount);
139)            if(protoName == ""){
140)                Console.WriteLine("OnReceiveData MsgBase.DecodeNam
    e fail");
141)                Close(state);
142)                return;
143)            }
144)            readBuff.readIdx += nameCount;
145)            //解析协议体
146)            int bodyCount = bodyLength - nameCount;
147)            if(bodyCount <= 0){
148)                Console.WriteLine("OnReceiveData fail, bodyCount <
    =0 ");
149)                Close(state);
150)                return;
151)            }
152)            MsgBase msgBase = MsgBase.Decode(protoName, readBuff.b
    ytes, readBuff.readIdx, bodyCount);
153)            readBuff.readIdx += bodyCount;
154)            readBuff.CheckAndMoveBytes();
155)            //分发消息
156)            MethodInfo mi =  typeof(MsgHandler).GetMethod(protoNam
    e);
157)            object[] o = {state, msgBase};
158)            Console.WriteLine("Receive " + protoName);
```

```
159)            if(mi != null){
160)                mi.Invoke(null, o);
161)            }
162)            else{
163)                Console.WriteLine("OnReceiveData Invoke fail " + p
    rotoName);
164)            }
165)            //继续读取消息
166)            if(readBuff.length > 2){
167)                OnReceiveData(state);
168)            }
169)        }
170)
171)        //发送
172)        public static void Send(ClientState cs, MsgBase msg){
173)            //状态判断
174)            if(cs == null){
175)                return;
176)            }
177)            if(!cs.socket.Connected){
178)                return;
179)            }
180)            //数据编码
181)            byte[] nameBytes = MsgBase.EncodeName(msg);
182)            byte[] bodyBytes = MsgBase.Encode(msg);
183)            int len = nameBytes.Length + bodyBytes.Length;
184)            byte[] sendBytes = new byte[2+len];
185)            //组装长度
186)            sendBytes[0] = (byte)(len%256);
187)            sendBytes[1] = (byte)(len/256);
188)            //组装名字
189)            Array.Copy(nameBytes, 0, sendBytes, 2, nameBytes.Lengt
    h);
190)            //组装消息体
191)            Array.Copy(bodyBytes, 0, sendBytes, 2+nameBytes.Length
    , bodyBytes.Length);
192)            //为简化代码，不设置回调
193)            try{
194)                cs.socket.BeginSend(sendBytes,0, sendBytes.Length,
     0, null, null);
195)            }catch(SocketException ex){
196)                Console.WriteLine("Socket Close on BeginSend" + ex
    .ToString());
```

```
197)            }
198)        }
199)
200)        //定时器
201)        static void Timer(){
202)            //消息分发
203)            MethodInfo mei =  typeof(EventHandler).GetMethod("OnTi
    mer");
204)            object[] ob = {};
205)            mei.Invoke(null, ob);
206)        }
207)
208)        //获取时间戳
209)        public static long GetTimeStamp() {
210)            TimeSpan ts = DateTime.UtcNow - new DateTime(1970, 1,
    1, 0, 0, 0, 0);
211)            return Convert.ToInt64(ts.TotalSeconds);
212)        }
213)    }
```