ET 框架小游戏--斗地主源码学习 - - 参考来帮助拖拉机重构游戏

deepwaterooo

May 20, 2023

Contents

1 UILobby 匹配按钮的回调 **1**

1.1 【客户端请求】LandlordsLobbyComponent ==> OnStartMatch() 1

1.2 【服务端】C2G_StartMatch_ReqHandler:【网关服】处理来自客户端的匹配请求 . . 1

1.3 MatchComponent:【匹配功能?】组件，匹配逻辑在 MatchComponentSystem 扩展. 这里是处理匹配逻辑的组件：它就需要【申请匹配者人】+【匹配玩家所到的游戏房间】两大部分 2

1.4 MatchComponentSystem: Update() 更新系。后面跟几个组件及个体类 2

1.4.1 MatchRoomComponent: 游戏房间组件，分玩家满，等更多的玩家，和空房间等几种情况 3

1.4.2 Room | RoomState: 4

1.4.3 MatcherComponent: 匹配申请者、被匹配者，的管理类组件。管理者类，就管理了所有发出过这个申请的申请者 4

1.4.4 Matcher: 匹配申请者，被匹配者组件。是指具体的一个个的申请者 5

1.5 【服务端】MH2MP_CreateRoom_ReqHandler:【地图服】会创建新的游戏房间 . . . 5

1.5.1 DeckComponent: 牌库组件 5

1.5.2 DeskCardsCacheComponent: 上面一个组件可能不够用，不得不加几个组件来组合 6

1.5.3 OrderControllerComponent: 玩家出牌顺序什么之类的游戏逻辑的管理 6

1.5.4 GameControllerComponent: 游戏控制类 6

1.5.5 RoomComponent: 房间管理组件 7

1.5.6 RoomConfig: 房间配置，房间的基本参数，什么的 7

1.6 Actor_PlayerEnterRoom_ReqHandler: 玩家进入游戏房间 7

1.6.1 UnitGateComponent|UnitGateComponentAwakeSystem 9

1.6.2 RoomSystem: 房间内部逻辑生成系，可以添加移除玩家、广播消息等 9

1.6.3 GamerState: 玩家状态消息，id, UserIdentity, 是地主吗? 10

1.6.4 HandCardsComponent: 为进入了（和正在处理进入）房间的玩家，添加手里的牌组件 10

1.7 Actor_GameStart_NttHandler: 游戏开始逻辑处理 10

1.7.1 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!!】 11

1.7.2 【爱表哥，爱生活!!!】 11

1.7.3 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!!】 11

2 源码梳理：用作【参考项目】来指导拖拉机项目的重构。 **11**

1 UILobby 匹配按钮的回调

1.1 【客户端请求】LandlordsLobbyComponent ==> OnStartMatch()

- 这个类向服务器发消息前，会先检查用户是否余额不足。
- 初始化时，因为用户是已经登录时来的，所以会去数据库拿用户的信息

```
public class LandlordsLobbyComponent : Component { // 大厅界面组件
    public void Awake() {
        Init();
    }
    // 开始匹配按钮事件
    public async void OnStartMatch() {
        try {
            // 发送开始匹配消息
            C2G_StartMatch_Req c2G_StartMatch_Req = new C2G_StartMatch_Req();
            G2C_StartMatch_Ack g2C_StartMatch_Ack = await SessionComponent.Instance.Session.Call(c2G_StartMatch_Req) as G2C
            if (g2C_StartMatch_Ack.Error == ErrorCode.ERR_UserMoneyLessError) {
                Log.Error(" 余额不足"); // 就是说，当且仅当余额不足的时候才会出这个错误？
                return;
            }
            // 匹配成功了：UI 界面切换，切换到房间界面
            UI room = Game.Scene.GetComponent<UIComponent>().Create(UIType.LandlordsRoom); // 装载新的 UI 视图
            Game.Scene.GetComponent<UIComponent>().Remove(UIType.LandlordsLobby);          // 卸载旧的 UI 视图
            // 将房间设为匹配状态
            room.GetComponent<LandlordsRoomComponent>().Matching = true;
        }
        catch (Exception e) {
            Log.Error(e.ToStr());
        }
    }
```

1.2 【服务端】C2G_StartMatch_ReqHandler:【网关服】处理来自客户端的匹配请求

```
[MessageHandler(AppType.Gate)] // 网关服：处理客户端 StartMatch 请求消息
public class C2G_StartMatch_ReqHandler : AMRpcHandler<C2G_StartMatch_Req, G2C_StartMatch_Ack> {
    protected override async void Run(Session session, C2G_StartMatch_Req message, Action<G2C_StartMatch_Ack> reply) {
        G2C_StartMatch_Ack response = new G2C_StartMatch_Ack();
        try {
            if (!GateHelper.SignSession(session)) { // 验证 Session
                response.Error = ErrorCode.ERR_SignError;
                reply(response);
                return;
            }
            User user = session.GetComponent<SessionUserComponent>().User;
            // 验证玩家是否符合进入房间要求，默认为 100 底分局
            RoomConfig roomConfig = RoomHelper.GetConfig(RoomLevel.Lv100);// 有不同标准的游戏房间
            UserInfo userInfo = await Game.Scene.GetComponent<DBProxyComponent>().Query<UserInfo>(user.UserID, false); //
            if (userInfo.Money < roomConfig.MinThreshold) {
                response.Error = ErrorCode.ERR_UserMoneyLessError; // 玩家钱不够，不能玩
                reply(response);
                return;
            }
// 这里先发送响应，让客户端收到后切换房间界面，否则可能会出现重连消息在切换到房间界面之前发送导致重连异常【这个应该是，别人的源标注了
// 这里的顺序就显得关键：因为只有网关服向客户端返回服务器的匹配响应【并不一定说已经匹配完成，但告诉客户端服务器在着手处理这个工作。。。】
            reply(response);
            // 向匹配服务器发送匹配请求
            StartConfigComponent config = Game.Scene.GetComponent<StartConfigComponent>();
            IPEndPoint matchIPEndPoint = config.MatchConfig.GetComponent<InnerConfig>().IPEndPoint; // 匹配服务器的远程 IP
            Session matchSession = Game.Scene.GetComponent<NetInnerComponent>().Get(matchIPEndPoint); // 拿到与这个匹配服务
            M2G_PlayerEnterMatch_Ack m2G_PlayerEnterMatch_Ack = await matchSession.Call(new G2M_PlayerEnterMatch_Req() { //
                    PlayerID = user.InstanceId,
                        UserID = user.UserID,
                        SessionID = session.InstanceId,
                        }) as M2G_PlayerEnterMatch_Ack;
            user.IsMatching = true;
        }
        catch (Exception e) {
            ReplyError(response, e, reply);
        }
    }
}
```

1.3 MatchComponent:【匹配功能 ?】组件,匹配逻辑在 MatchComponentSystem 扩展. 这里是处理匹配逻辑的组件：它就需要【申请匹配者人】+【匹配玩家所到的游戏房间】两大部分

- **【MatchRoomComponent】**：被匹配到的房间，成为这个组件的另一个组成部分。
- **【Matcher 被匹配者】**：是这个匹配功能的一大内在版块

```
// 匹配组件，匹配逻辑在 MatchComponentSystem 扩展. 这里是处理匹配的组件，与 Matcher 被匹配者相区分开来
public class MatchComponent : Component {
    // 游戏中匹配对象列表：值是 roomId
    public readonly Dictionary<long, long> Playing = new Dictionary<long, long>();
    // 匹配成功队列
    public readonly Queue<Matcher> MatchSuccessQueue = new Queue<Matcher>();
    // 创建房间消息加锁，避免因为延迟重复发多次创建房间消息
    public bool CreateRoomLock { get; set; }
}
```

1.4 MatchComponentSystem: Update() 更新系。后面跟几个组件及个体类

- 与服务器的交互再涉及两个类：创建新房间组件，与申请匹配成功的玩家进入房间组件

```
public static class MatchComponentSystem {
    public static void Update(this MatchComponent self) {
        while (true) {
            MatcherComponent matcherComponent = Game.Scene.GetComponent<MatcherComponent>();// 玩家管理组件
            Queue<Matcher> matchers = new Queue<Matcher>(matcherComponent.GetAll());         // 玩家们
            MatchRoomComponent roomManager = Game.Scene.GetComponent<MatchRoomComponent>(); // 游戏房间
            Room room = roomManager.GetReadyRoom(); // 返回的是：人员不满 < 3 个的一个房间       // 房间个体
            if (matchers.Count == 0)
                // 当没有匹配玩家时直接结束
                break;
            if (room == null && matchers.Count >= 3) // 分配一个空房间
                // 当还有一桌匹配玩家且没有可加入房间时使用空房间
                room = roomManager.GetIdleRoom();
            if (room != null) { // 只要房间不为空，就被强按到这个房间里了，没有任何其它逻辑考量
                // 当有准备状态房间且房间还有空位时匹配玩家直接加入填补空位
                while (matchers.Count > 0 && room.Count < 3) // 是个循环：可以匹配好几个玩家，到好几个有空位的游戏房间
                    self.JoinRoom(room, matcherComponent.Remove(matchers.Dequeue().UserID));
            }
            else if (matchers.Count >= 3) {
                // 当还有一桌匹配玩家且没有空房间时创建新房间
                self.CreateRoom();
                break;
            } else break;
            // 移除匹配成功玩家
            while (self.MatchSuccessQueue.Count > 0)
                matcherComponent.Remove(self.MatchSuccessQueue.Dequeue().UserID);
        }
    }
    // 创建房间
    public static async void CreateRoom(this MatchComponent self) {
        if (self.CreateRoomLock)
            return;
        // 消息加锁，避免因为延迟重复发多次创建消息
        self.CreateRoomLock = true;
        // 发送创建房间消息：这里几个相关组件，可能重构的时候，也会被 ET7 重构去掉，所以没有看。重点看：【大型网络游戏中需要与
        IPEndPoint mapIPEndPoint = Game.Scene.GetComponent<AllotMapComponent>().GetAddress().GetComponent<InnerConfig
        Session mapSession = Game.Scene.GetComponent<NetInnerComponent>().Get(mapIPEndPoint);
        MP2MH_CreateRoom_Ack createRoomRE = await mapSession.Call(new MH2MP_CreateRoom_Req()) as MP2MH_CreateRoom_Ack
        Room room = ComponentFactory.CreateWithId<Room>(createRoomRE.RoomID);
        Game.Scene.GetComponent<MatchRoomComponent>().Add(room);
        // 解锁
        self.CreateRoomLock = false;
    }
```

```
        // 加入房间: 逻辑极简单, 就只要钱够就可以了。多出了房间服务器【任何时候, 活宝妹就是一定要嫁给亲爱的表哥!!!】
        public static async void JoinRoom(this MatchComponent self, Room room, Matcher matcher) {
            // 玩家加入房间, 移除匹配队列
            self.Playing[matcher.UserID] = room.Id;
            self.MatchSuccessQueue.Enqueue(matcher);
            // 向房间服务器发送玩家进入请求
            ActorMessageSender actorProxy = Game.Scene.GetComponent<ActorMessageSenderComponent>().Get(room.Id);
            IResponse response = await actorProxy.Call(new Actor_PlayerEnterRoom_Req() {
                    PlayerID = matcher.PlayerID,
                        UserID = matcher.UserID,
                        SessionID = matcher.GateSessionID
                        });
            Actor_PlayerEnterRoom_Ack actor_PlayerEnterRoom_Ack = response as Actor_PlayerEnterRoom_Ack;
            Gamer gamer = GamerFactory.Create(matcher.PlayerID, matcher.UserID, actor_PlayerEnterRoom_Ack.GamerID);
            room.Add(gamer);
            // 向玩家发送匹配成功消息
            ActorMessageSenderComponent actorProxyComponent = Game.Scene.GetComponent<ActorMessageSenderComponent>();
            ActorMessageSender gamerActorProxy = actorProxyComponent.Get(gamer.PlayerID);
            gamerActorProxy.Send(new Actor_MatchSucess_Ntt() { GamerID = gamer.Id });
        }
    }
```

1.4.1 MatchRoomComponent: 游戏房间组件, 分玩家满, 等更多的玩家, 和空房间等几种情况

```
// 匹配房间管理组件, 逻辑在 MatchRoomComponentSystem 扩展
public class MatchRoomComponent : Component {
    // 所有房间列表
    public readonly Dictionary<long, Room> rooms = new Dictionary<long, Room>();
    // 游戏中房间列表
    public readonly Dictionary<long, Room> gameRooms = new Dictionary<long, Room>();
    // 等待中房间列表
    public readonly Dictionary<long, Room> readyRooms = new Dictionary<long, Room>();
    // 空闲房间列表
    public readonly Queue<Room> idleRooms = new Queue<Room>();
    // 房间总数
    public int TotalCount { get { return this.rooms.Count; } }
    // 游戏中房间数
    public int GameRoomCount { get { return gameRooms.Count; } }
    // 等待中房间数: 只要人数不够的房间, 都算等待中。。。。。
    public int ReadyRoomCount { get { return readyRooms.Where(p => p.Value.Count < 3).Count(); } }
    // 空闲房间数
    public int IdleRoomCount { get { return idleRooms.Count; } }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        foreach (var room in this.rooms.Values) {
            room.Dispose();
        }
    }
}
```

1.4.2 Room | RoomState:

- 后面, 还有个 RoomComponent 管理者类。下一节

```
// 房间状态
public enum RoomState : byte {
    Idle,
    Ready,
    Game
}
public sealed class Room : Entity { // 房间对象
    public readonly Dictionary<long, int> seats = new Dictionary<long, int>();
    public readonly Gamer[] gamers = new Gamer[3];
    // 房间状态
    public RoomState State { get; set; } = RoomState.Idle;
    // 房间玩家数量
    public int Count { get { return seats.Values.Count; } }
    public override void Dispose()
        if (this.IsDisposed) {
            return;
        base.Dispose();
        seats.Clear();
        for (int i = 0; i < gamers.Length; i++)
            if (gamers[i] != null) {
                gamers[i].Dispose();
                gamers[i] = null;
            }
        State = RoomState.Idle;
    }
}
```

1.4.3 MatcherComponent: 匹配申请者、被匹配者，的管理类组件。管理者类，就管理了所有发出过这个申请的申请者

```
// 匹配对象管理组件
public class MatcherComponent : Component {
    private readonly Dictionary<long, Matcher> matchers = new Dictionary<long, Matcher>();
    // 匹配对象数量
    public int Count { get { return matchers.Count; } }
    // 添加匹配对象
    public void Add(Matcher matcher) {
        this.matchers.Add(matcher.UserID, matcher);
    }
    // 获取匹配对象
    public Matcher Get(long id) {
        this.matchers.TryGetValue(id, out Matcher matcher);
        return matcher;
    }
    // 获取所有匹配对象
    public Matcher[] GetAll() {
        return this.matchers.Values.ToArray();
    }
    // 移除匹配对象并返回
    public Matcher Remove(long id) {
        Matcher matcher = Get(id);
        this.matchers.Remove(id);
        return matcher;
    }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        foreach (var matcher in this.matchers.Values) {
            matcher.Dispose();
        }
    }
}
```

1.4.4 Matcher: 匹配申请者，被匹配者组件。是指具体的一个个的申请者

- 它像是个自觉醒组件。同一个文件里也添加了 Awake()

```
// 匹配对象: 匹配的玩家系统
public sealed class Matcher : Entity {
    // 用户 ID (唯一)
    public long UserID { get; private set; }
    // 玩家 GateActorID
    public long PlayerID { get; set; }
    // 客户端与网关服务器的 SessionID
    public long GateSessionID { get; set; }
    public void Awake(long id) {
        this.UserID = id;
    }
    public override void Dispose() {
        if(this.IsDisposed) return;
        base.Dispose();
        this.UserID = 0;
        this.PlayerID = 0;
        this.GateSessionID = 0;
    }
}
```

1.5 【服务端】MH2MP_CreateRoom_ReqHandler:【地图服】会创建新的游戏房间

- 工厂化生产了一个房间。并为房间添加了几个管理者类组件：DeckComponent, DeskCardsCacheComponent, OrderControllerComponent, GameControllerComponent,
- 为游戏房间添加了邮箱组件，方便游戏房间里聊天，“再不出牌我就要打 120 了呀。。活宝妹就是一定要嫁给亲爱的表哥!!!”【活宝妹就是一定要嫁给亲爱的表哥!!!】
- 把当前刚生产的房间加入管理者的统管范围。RoomComponent
- 这里只是大致了解，游戏客户端与服务端的交互设计，游戏里元件组件的拆分，里面的连接逻辑，元件组件间的交互逻辑还没有细看。有必要时会细看。

```
[MessageHandler(AppType.Map)]
public class MH2MP_CreateRoom_ReqHandler : AMRpcHandler<MH2MP_CreateRoom_Req, MP2MH_CreateRoom_Ack> {
    protected override async void Run(Session session, MH2MP_CreateRoom_Req message, Action<MP2MH_CreateRoom_Ack> rep
        MP2MH_CreateRoom_Ack response = new MP2MH_CreateRoom_Ack();
        try {
            // 创建房间
            Room room = ComponentFactory.Create<Room>(); // 工厂化生产一个房间
            room.AddComponent<DeckComponent>();
            room.AddComponent<DeskCardsCacheComponent>();
            room.AddComponent<OrderControllerComponent>();
            room.AddComponent<GameControllerComponent, RoomConfig>(RoomHelper.GetConfig(RoomLevel.Lv100));
            await room.AddComponent<MailBoxComponent>().AddLocation();// 去查看一下：是否是为了方便游戏房间里聊天？
            Game.Scene.GetComponent<RoomComponent>().Add(room);
            Log.Info($" 创建房间 {room.InstanceId}");
            response.RoomID = room.InstanceId;
            reply(response);
        }
        catch (Exception e) {
            ReplyError(response, e, reply);
        }
    }
}
```

1.5.1 DeckComponent: 牌库组件

```
public class DeckComponent : Component { // 牌库组件
    // 牌库中的牌
    public readonly List<Card> library = new List<Card>();
    // 牌库中的总牌数
    public int CardsCount { get { return this.library.Count; } }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        library.Clear();
    }
}
```

1.5.2 DeskCardsCacheComponent: 上面一个组件可能不够用，不得不加几个组件来组合

```
public class DeskCardsCacheComponent : Component {
    // 牌桌上的牌
    public readonly List<Card> library = new List<Card>();
    // 地主牌
    public readonly List<Card> LordCards = new List<Card>();
    // 牌桌上的总牌数
    public int CardsCount { get { return this.library.Count; } }
    // 当前最大牌型：这里为什么要纪录当前最大牌型？哪家的？读源码来搞明白
    public CardsType Rule { get; set; }
    // 牌桌上最小的牌
    public int MinWeight { get { return (int)this.library[0].CardWeight; } }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        library.Clear();
        LordCards.Clear();
        Rule = CardsType.None;
    }
}
```

1.5.3 OrderControllerComponent: 玩家出牌顺序什么之类的游戏逻辑的管理

```
// 这些都算是: 游戏逻辑控制的组件化拆分。以前自己的游戏可能是一个巨大无比的控制器文件, 这里折分成了很多个小组件控制
public class OrderControllerComponent : Component {
    // 先手玩家
    public KeyValuePair<long, bool> FirstAuthority { get; set; }
    // 玩家抢地主状态
    public Dictionary<long, bool> GamerLandlordState = new Dictionary<long, bool>();
    // 本轮最大牌型玩家
    public long Biggest { get; set; }
    // 当前出牌玩家
    public long CurrentAuthority { get; set; }
    // 当前抢地主玩家
    public int SelectLordIndex { get; set; }

    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        this.GamerLandlordState.Clear();
        this.Biggest = 0;
        this.CurrentAuthority = 0;
        this.SelectLordIndex = 0;
    }
}
```

1.5.4 GameControllerComponent: 游戏控制类

```
// 感觉个类, 更多的是【一座桥】: 把游戏的这个单位级件, 全连接起来
public class GameControllerComponent : Component {
    // 房间配置
    public RoomConfig Config { get; set; }
    // 底分: 这里呈现出与房间的这些设置不一致的状态。是说, 三个玩家, 可以在既定房间的基础上提升玩乐标准?
    public long BasePointPerMatch { get; set; }
    // 全场倍率
    public int Multiples { get; set; }
    // 最低入场门槛
    public long MinThreshold { get; set; }

    public override void Dispose() {
        if (this.IsDisposed) return;
        base.Dispose();
        this.BasePointPerMatch = 0;
        this.Multiples = 0;
        this.MinThreshold = 0;
    }
}
```

1.5.5 RoomComponent: 房间管理组件

- ET 框架源码读多也, 也该明白, 所有的 Component 组件, 全都是管理者组件。

```
// 房间管理组件
public class RoomComponent : Component {
    private readonly Dictionary<long, Room> rooms = new Dictionary<long, Room>();
    // 添加房间
    public void Add(Room room) {
        this.rooms.Add(room.InstanceId, room);
    }
    // 获取房间
    public Room Get(long id) {
        Room room;
        this.rooms.TryGetValue(id, out room);
        return room;
    }
```

```
        // 移除房间并返回
        public Room Remove(long id) {
            Room room = Get(id);
            this.rooms.Remove(id);
            return room;
        }
        public override void Dispose() {
            if (this.IsDisposed) return;
            base.Dispose();
            foreach (var room in this.rooms.Values) {
                room.Dispose();
            }
        }
    }
```

1.5.6 RoomConfig: 房间配置，房间的基本参数，什么的

```
// 房间配置
public struct RoomConfig {
    // 房间初始倍率
    public int Multiples { get; set; }
    // 房间底分
    public long BasePointPerMatch { get; set; }
    // 房间最低门槛
    public long MinThreshold { get; set; }
}
```

1.6 Actor_PlayerEnterRoom_ReqHandler: 玩家进入游戏房间

- 为玩家添加邮箱，方便玩家收发消息。那前面，为什么房间也要添加邮箱？集中消息？可是每个玩家看见的都是自己的往返消息，集中消息给谁看？
- 广播：新玩家进场
- 通过代理发送：【游戏开始】的消息？不知道这个消息是怎么处理的。逻辑不通，每个玩家都发，谁说了算，得查逻辑

```
[ActorMessageHandler(AppType.Map)]
public class Actor_PlayerEnterRoom_ReqHandler : AMActorRpcHandler<Room, Actor_PlayerEnterRoom_Req, Actor_PlayerEnterF
    protected override async Task Run(Room room, Actor_PlayerEnterRoom_Req message, Action<Actor_PlayerEnterRoom_Ack>
        Actor_PlayerEnterRoom_Ack response = new Actor_PlayerEnterRoom_Ack();
        try {
            Gamer gamer = room.Get(message.UserID);
            if (gamer == null) { // 当前玩家，在这个被分配的房间里，还没被初始化
                // 创建房间玩家对象
                gamer = GamerFactory.Create(message.PlayerID, message.UserID);
                await gamer.AddComponent<MailBoxComponent>().AddLocation(); // 只有给玩家挂上这个组件，并向中央邮件注册
                gamer.AddComponent<UnitGateComponent, long>(message.SessionID);
                // 加入到房间
                room.Add(gamer); // 这里就又多一步逻辑处理：这里当服务器匹配成功一个玩家，就去做相应的客户端视图层相应的变动
                Actor_GamerEnterRoom_Ntt broadcastMessage = new Actor_GamerEnterRoom_Ntt();
                foreach (Gamer _gamer in room.GetAll()) {
                    if (_gamer == null) {
                        // 添加空位：添加所有的，当前这个消息的接受者
                        broadcastMessage.Gamers.Add(new GamerInfo());
                        continue;
                    }
                    // 添加玩家信息
                    GamerInfo info = new GamerInfo() { UserID = _gamer.UserID, IsReady = _gamer.IsReady };
                    broadcastMessage.Gamers.Add(info);
                }
                // 广播消息：给房间内的所有玩家，新人驾到，请多关照
                room.Broadcast(broadcastMessage);
                Log.Info($" 玩家 {message.UserID} 进入房间");
            } else { // 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!!】
                // 玩家重连
                gamer.isOffline = false;
                gamer.PlayerID = message.PlayerID;
                gamer.GetComponent<UnitGateComponent>().GateSessionActorId = message.SessionID;
                // 玩家重连，移除托管组件
                gamer.RemoveComponent<TrusteeshipComponent>(); // 这个好像是使玩家可以自动机器人帮出牌的
                Actor_GamerEnterRoom_Ntt broadcastMessage = new Actor_GamerEnterRoom_Ntt();
                foreach (Gamer _gamer in room.GetAll()) {
                    if (_gamer == null) {
                        // 添加空位
                        broadcastMessage.Gamers.Add(default(GamerInfo));
                        continue;
                    }
                    // 添加玩家信息
                    GamerInfo info = new GamerInfo() { UserID = _gamer.UserID, IsReady = _gamer.IsReady };
                    broadcastMessage.Gamers.Add(info);
                }
                // 发送房间玩家信息
                ActorMessageSender actorProxy = gamer.GetComponent<UnitGateComponent>().GetActorMessageSender();
                actorProxy.Send(broadcastMessage);
                // 这部分：看看清楚
                List<GamerCardNum> gamersCardNum = new List<GamerCardNum>();
                List<GamerState> gamersState = new List<GamerState>();
                GameControllerComponent gameController = room.GetComponent<GameControllerComponent>();
                OrderControllerComponent orderController = room.GetComponent<OrderControllerComponent>();
                DeskCardsCacheComponent deskCardsCache = room.GetComponent<DeskCardsCacheComponent>();
                foreach (Gamer _gamer in room.GetAll()) {
                    HandCardsComponent handCards = _gamer.GetComponent<HandCardsComponent>(); // 游戏开始里，Actor_Gar
                    gamersCardNum.Add(new GamerCardNum() {
                            UserID = _gamer.UserID,
                                Num = _gamer.GetComponent<HandCardsComponent>().GetAll().Length
                                });
                    GamerState gamerState = new GamerState() {
                        UserID = _gamer.UserID,
                        UserIdentity = handCards.AccessIdentity
                    };
                    if (orderController.GamerLandlordState.TryGetValue(_gamer.UserID, out bool state)) {
                        if (state)
                            gamerState.State = GrabLandlordState.Grab;
                        else
                            gamerState.State = GrabLandlordState.UnGrab;
                    }
                    gamersState.Add(gamerState);
                }
                // 发送游戏开始消息
                Actor_GameStart_Ntt gameStartNotice = new Actor_GameStart_Ntt(); // 因为这个逻辑比较多，后面的没有再看
                gameStartNotice.HandCards.AddRange(gamer.GetComponent<HandCardsComponent>().GetAll());
                gameStartNotice.GamersCardNum.AddRange(gamersCardNum);
                actorProxy.Send(gameStartNotice);
                Card[] lordCards = null;
                if (gamer.GetComponent<HandCardsComponent>().AccessIdentity == Identity.None) {
                    // 广播先手玩家
                    actorProxy.Send(new Actor_AuthorityGrabLandlord_Ntt() { UserID = orderController.CurrentAuthority
                } else {
                    if (gamer.UserID == orderController.CurrentAuthority) {
                        // 发送可以出牌消息
                        bool isFirst = gamer.UserID == orderController.Biggest;
                        actorProxy.Send(new Actor_AuthorityPlayCard_Ntt() { UserID = orderController.CurrentAuthority
                    }
                    lordCards = deskCardsCache.LordCards.ToArray();
                }
                // 发送重连数据
                Actor_GamerReconnect_Ntt reconnectNotice = new Actor_GamerReconnect_Ntt() {
                    UserId = orderController.Biggest,
                    Multiples = room.GetComponent<GameControllerComponent>().Multiples
                };
                reconnectNotice.GamersState.AddRange(gamersState);
                reconnectNotice.Cards.AddRange(deskCardsCache.library);
                if (lordCards != null)
                    reconnectNotice.LordCards.AddRange(lordCards);
                actorProxy.Send(reconnectNotice);
                Log.Info($" 玩家 {message.UserID} 重连");
            }
            response.GamerID = gamer.InstanceId;
            reply(response);
        }
        catch (Exception e) {
            ReplyError(response, e, reply);
        }
    }
    }
```

1.6.1 UnitGateComponent|UnitGateComponentAwakeSystem

- 有了这个组件，好像是玩家间就可以发消息了？

```
[ObjectSystem]
public class UnitGateComponentAwakeSystem : AwakeSystem<UnitGateComponent, long> {
    public override void Awake(UnitGateComponent self, long a) {
        self.Awake(a);
    }
}
public class UnitGateComponent : Component, ISerializeToEntity {
    public long GateSessionActorId;
    public bool IsDisconnect;
    public void Awake(long gateSessionId) {
        this.GateSessionActorId = gateSessionId;
    }
    public ActorMessageSender GetActorMessageSender() {
        return Game.Scene.GetComponent<ActorMessageSenderComponent>().Get(this.GateSessionActorId);
    }
}
```

1.6.2 RoomSystem: 房间内部逻辑生成系，可以添加移除玩家、广播消息等

```
public static class RoomSystem {
    // 添加玩家
    public static void Add(this Room self, Gamer gamer) {
        int seatIndex = self.GetEmptySeat();
        // 玩家需要获取一个座位坐下
        if (seatIndex >= 0) {
            self.gamers[seatIndex] = gamer;
            self.seats[gamer.UserID] = seatIndex;
            gamer.RoomID = self.InstanceId;
        }
    }
    // 获取玩家
    public static Gamer Get(this Room self, long id) {
        int seatIndex = self.GetGamerSeat(id);
        if (seatIndex >= 0)
            return self.gamers[seatIndex];
        return null;
    }
    // 获取所有玩家
    public static Gamer[] GetAll(this Room self) {
        return self.gamers;
    }
    // 获取玩家座位索引
    public static int GetGamerSeat(this Room self, long id) {
        if (self.seats.TryGetValue(id, out int seatIndex))
            return seatIndex;
        return -1;
    }
    // 移除玩家并返回
    public static Gamer Remove(this Room self, long id) {
        int seatIndex = self.GetGamerSeat(id);
        if (seatIndex >= 0) {
            Gamer gamer = self.gamers[seatIndex];
            self.gamers[seatIndex] = null;
            self.seats.Remove(id);
            gamer.RoomID = 0;
            return gamer;
        }
        return null;
    }
    // 获取空座位
    // <returns> 返回座位索引，没有空座位时返回-1</returns>
    public static int GetEmptySeat(this Room self) {
        for (int i = 0; i < self.gamers.Length; i++)
            if (self.gamers[i] == null)
                return i;
        return -1;
    }
    // 广播消息
    public static void Broadcast(this Room self, IActorMessage message) {
        foreach (Gamer gamer in self.gamers) {
            if (gamer == null || gamer.isOffline)
                continue;
            ActorMessageSender actorProxy = gamer.GetComponent<UnitGateComponent>().GetActorMessageSender();
            actorProxy.Send(message);
        }
    }
}
```

1.6.3 GamerState: 玩家状态消息，id, UserIdentity, 是地主吗？

```
message GamerState {
    int64 UserID = 1;
    ETModel.Identity UserIdentity = 2;
^^IGrabLandlordState State = 3;
}
```

1.6.4 HandCardsComponent: 为进入了（和正在处理进入）房间的玩家，添加手里的牌组件

```
public class HandCardsComponent : Component {
    // 所有手牌
    public readonly List<Card> library = new List<Card>();
    // 身份：地主，还是平民老百姓？
    public Identity AccessIdentity { get; set; }
    // 是否托管：自动出牌吗
    public bool IsTrusteeship { get; set; }
    // 手牌数
    public int CardsCount { get { return library.Count; } }
    public override void Dispose() {
        if (this.IsDisposed) return;
        base.Dispose();
        this.library.Clear();
        AccessIdentity = Identity.None;
        IsTrusteeship = false;
    }
}
```

1.7 Actor_GameStart_NttHandler: 游戏开始逻辑处理

```
[MessageHandler]
public class Actor_GameStart_NttHandler : AMHandler<Actor_GameStart_Ntt> {
    protected override void Run(ETModel.Session session, Actor_GameStart_Ntt message) {
        UI uiRoom = Game.Scene.GetComponent<UIComponent>().Get(UIType.LandlordsRoom);
        GamerComponent gamerComponent = uiRoom.GetComponent<GamerComponent>();
        // 初始化玩家 UI
        foreach (GamerCardNum gamerCardNum in message.GamersCardNum) {
            Gamer gamer = uiRoom.GetComponent<GamerComponent>().Get(gamerCardNum.UserID);
            GamerUIComponent gamerUI = gamer.GetComponent<GamerUIComponent>();
            gamerUI.GameStart();
            HandCardsComponent handCards = gamer.GetComponent<HandCardsComponent>();
            if (handCards != null)
                handCards.Reset();
            else
                handCards = gamer.AddComponent<HandCardsComponent, GameObject>(gamerUI.Panel);
            handCards.Appear();
            if (gamer.UserID == gamerComponent.LocalGamer.UserID)
                // 本地玩家添加手牌
                handCards.AddCards(message.HandCards);
            else
                // 设置其他玩家手牌数
                handCards.SetHandCardsNum(gamerCardNum.Num);
        }
        // 显示牌桌 UI
        GameObject desk = uiRoom.GameObject.Get<GameObject>("Desk");
        desk.SetActive(true);
        GameObject lordPokers = desk.Get<GameObject>("LordPokers");
        // 重置地主牌
        Sprite lordSprite = CardHelper.GetCardSprite("None");
        for (int i = 0; i < lordPokers.transform.childCount; i++)
            lordPokers.transform.GetChild(i).GetComponent<Image>().sprite = lordSprite;
        LandlordsRoomComponent uiRoomComponent = uiRoom.GetComponent<LandlordsRoomComponent>();
        // 清空选中牌
        uiRoomComponent.Interaction.Clear();
        // 设置初始倍率
        uiRoomComponent.SetMultiples(1);
    }
}
```

1.7.1 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!!】

1.7.2 【爱表哥，爱生活!!!】

1.7.3 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!!】

2 源码梳理：用作【参考项目】来指导拖拉机项目的重构。

- 这个文件，以前不知道总结的是些什么乱七八糟的。现在重点梳理：斗地主的游戏逻辑相关
- 目的是用作参考，来指导自己【拖拉机游戏】的重构。
- 所以就按照界面相关的形式，或是几个按钮回调的形式来梳理游戏逻辑的【客户端】请求与【服务端】的处理请求
- 【任何时候，活宝妹就是一定要、一定会嫁给偶亲爱的表哥!!! 爱表哥，爱生活!!!】