

# ET 框架学习笔记（四）-- 框架总结【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】

deepwaterooo

June 25, 2023

## Contents

<b>1 IMHandler 接口实现的各种类型消息处理器：需要先理解透彻 ETTask 和 ETVoid:</b>	<b>1</b>
1.1 IMHandler: interface 消息处理器接口类。它有 2 个实现抽象类：AMHandler, AM-RpcHandler	2
1.2 AMHandler<Message>: abstract 抽象基类：两个方法的返回类型，成为现在全框架的理解与实现难点	2
1.3 AMRpcHandler: 去抓的 ET7 框架的源码，可以用来【校正】其它被自己改错的	3
1.4 C2R_LoginHandler: AMRpcHandler 的一个实体类，前后版本封装对比	4
1.5 IMHandler 【ET-EUI】：主要是与上面对比	5
1.6 IMActorHandler: 【ET-EUI: 仅作参考】大概参考 ET-EUI 来的，它的目的应该是把最基类的接口，与其它两类的接口相区分开来	5
1.7 AMActorLocationHandler: 源码被我改动了	5
1.8 AMActorLocationRpcHandler: 上面接口实现类的一个使用例子。我把返回参数改了	6
1.9 AMActorLocationRpcHandler: Rpc 就是进程间消息（或是 ET7 重构为 SceneType 之后的多核间消息）	6
1.10 ActorHandleHelper 静态帮助类：包装了必要的方法，帮助自动化回复相关回调消息 et3	7
1.11 MessageDispatcherComponent: 这些是一个系统的，需要放在一起总结	7
1.12 MessageDispatcherComponentSystem:	7
<b>2 C# 异步基础原理、状态机原理、逻辑整理</b>	<b>8</b>
2.1 ETVoid C# Net-async await 编程更底层一点儿的原理	8
2.2 如果方法声明为 async，那么可以直接 return 具体的值，不再用创建 Task，由编译器创建 Task:	14
<b>3 现在的修改内容：【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】几大版块</b>	<b>15</b>
3.1 框架里现存的错误：截图，明天上午把功能模块一一看一遍，现在不知道怎么改了，要再看再学习了才会改	15
3.2 【ComponentFactory】现在就可以清理掉这个组件的所有了	16
3.3 热更新层的生成系	18
3.4 内网消息等网络相关：请求消息的发送方法等。很多编译错误，要一点儿一点儿把他们都改掉	19

3.4.1 修改下面的 ActorMessageSenderComponent 因为功能模块逻辑重构，而带来的一堆编译错误。 . . . . .	20
3.4.2 【ActorMessageSenderComponent】：这个类很重要、很重要，现在是活宝妹理解网络模块的核心。爱表哥，爱生活!!! . . . . .	20
3.5 静态类的环形引用问题 . . . . .	22
3.6 我的 VS - - Visual Studio 被他们作手脚，他们可以操纵我的 VS 帐户，上次前不久某天下午已经注意到这个问题 . . . . .	22
3.7 下面是已经改好了的：还是先放着，备查 . . . . .	23
3.8 先前列的相对杂一点儿 . . . . .	23
<b>4 Protobuf 相关，【Protobuf 里进程间传递的游戏数据相关信息：两个思路】</b>	<b>27</b>
<b>5 Unit：这个模块还不太懂，需要明天上午花时间再看一下</b>	<b>27</b>
5.1 UnitGateComponent: . . . . .	27
5.2 UnitGateComponentSystem . . . . .	28
<b>6 ET7 框架以及【参考项目】的 ECS：小单元小类型的生成系，是怎么写的，找例子参考</b>	<b>28</b>
6.1 IComponentSerialize: . . . . .	28
6.2 ClientComponent: 【参考项目】客户端组件，找个 ET7 里的组件 . . . . .	29
<b>7 TODO 其它的：部分完成，或是待完成大的功能版块，列举</b>	<b>29</b>
<b>8 拖拉机游戏：【重构 OOP/OOD 设计思路】</b>	<b>29</b>

## 1 IMHandler 接口实现的各种类型消息处理器：需要先理解透彻 ETTask 和 ETVoid：

- 这个模块感觉还没有总结完。但因为还有 111 个编译错误，很多我还不知道算是怎么回事。这个版块的总结放在后面，再改错误的时候带着问题来看更有效率。
- 可以回去参考前一个游戏参考过的 ET-EUI 里有一部分 ETVoid 的相关使用，可以用作自己理解 ETTask ETVoid 的源码参考。ET-EUI 现笔记本里没有，可以拉一个下来。
  - 【抓下来后用法基本一样， ETVoid 使用的地方基本一样】。不一样的地方是，它扩展（或者说 重新自定义了）了不少 消息处理器接口类。那么就是说，ET-EUI 并没有、也做不到一个框架整个框架只使用一个消息处理器接口
- 【主要使用】主要是【服务端】处理客户端消息请求，用来定义处理逻辑。但是网络调用与返回大多是异步的，所以会有很多使用 ETTask 或是 ETVoid|Void 作为返回值的地方。主要是两个常用方法的接口定义，兼顾整个框架的接口定义。
- 【参照 ET-EUI】：如果再来参照这个例子项目，或许也可以多定义几个不同的消息处理器接口，就不必强制整个框架只实现一个接口而顾 A 顾不了 B 了。那么如果下午继续参照这个例子，头脑清醒的时候，就要搞明白：不同接口类，到底适用哪类消息？【可以把这部分再分析理解一下，总结在下面，但是区分清楚，哪个来自 ET-EUI】就是需要理解透彻再改，不要再循环无限制地改。
- 【两点不透彻】ETTask|ETVoid|Void 到底使用什么返回值？另则，async-await 方法是定义为异步，还是非异步。如果 async 定义异步，什么地方必须有 await 调用？
- 感觉 ETTask|ETVoid 基本弄明白了，可是这里仍然是整个框架，感觉最为复杂不好修改的地方。可能我还是把网络异步调用没能弄得狠明白。

- 要保证两个方法里，若是同步方法，方法中就一定不能有需要异步等待的地方（否则运行时会抛异常，对象为空之类的各种因为异步操作不能及时完成而抛的空异常）。就是，**如果需要使用异步方法，不可以改为同步，同步返回 void 或其它任何。**
- 改的过程中，方法中曾经有过 await 调用异步方法的逻辑（方法），不能因想掉编译错误就去掉 await 调用关键字，错误地改成同步方法，因为暂时去掉了编译错误，运行时一定会抛异常。
- 下午改这个：把 ET 框架的原分支，ET-EUI 项目的消息处理器接口相关的类，需要对照原项目，确认没被自己弄得乱七八糟的
- 活宝妹只是重构改一个游戏项目。他们大可不必发疯犯贱。写这些，都是在写活宝妹的心理阴影，被他们发疯弄怕了。。。
- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹，就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

## 1.1 IMHandler: interface 消息处理器接口类。它有 2 个实现抽象类: AMHandler, AMRpcHandler

- 实现这个接口类型主要分为两个抽象类：AMHandler, AMRpcHandler. 所以 ET 框架里现有一个接口 IMHandler, 两个抽象实现类。其它是参考自 ET-EUI.
- 今天上午暂时只看取了这个接口，和两个抽象方法这里。后面任何的个体实现类，都还没有细看，不太懂。

```
public interface IMHandler {
    // 下面，返回类型不对
    void Handle(Session session, object message); // 这里返回类型，仍然应该是 ETTask，或者可能的 ETVoid ?
    // 消息处理器帮助类，在程序域加载的时候，会自动扫描程序域里的 ActorMessageHandler 标签，会想要拿消息的【发送类型】与消息的【返回
    Type GetMessageType();
    Type GetResponseType(); // 这里现在想的话，可能存在问题是：有些消息不带返回类型，那么就是说，那类消息不需要处理吗？它只是不需要
}
```

## 1.2 AMHandler<Message>: abstract 抽象基类：两个方法的返回类型，成为现在全框架的理解与实现难点

- AMHandler 类，这个类相比 AMRpcHandler 更加简单一些，因为这个类型的处理，不需要关心回消息
- 实现了接口 IMHandler 的 Handle 异步方法，具体逻辑为：
  - 将传进来的 msg 首先转换为模板类，在 AMHandler 类里面为 Message，具体应该为实现 AMHandler 的类的具体数据类。
  - 根据数据类，以及 session 生成一些报错日志，方便调试
  - 调用 Run 方法，将 session 及具体的数据类传进去
  - 实际继承抽象类 AMHandler 的类型，会实现这个接口，从而走向各自的处理。

```
public abstract class AMHandler<Message> : IMHandler where Message : class {

    protected abstract ETTTask Run(Session session, Message message); // ET7 原本的
    // 虽然我这么改，可以暂时消掉编译错误。但改得不对，现在消掉了编译错误，等编译通过，运行时错误会一再崩出来的...
    // protected abstract void Run(Session session, Message message);

    public void Handle(Session session, object msg) {
        Message message = msg as Message;
        if (message == null) {
            Log.Error($" 消息类型转换错误: {msg.GetType().Name} to {typeof (Message).Name}");
            return;
        }
        if (session.IsDisposed) {
```

```

        Log.Error($"session disconnect {msg}");
        return;
    }
    this.Run(session, message).Coroutine(); // 同步方法：调用的是异步方法的协程？可以这么写吗？
}
public Type GetMessageType() {
    return typeof (Message);
}
public Type GetResponseType() {
    return null;
}
}
}

```

### 1.3 AMRpcHandler: 去抓的 ET7 框架的源码，可以用来【校正】其它被自己改错的

- 注意，在 ET7 框架里，IMHandler 接口，与 AMHandler 是定义在 Share 双端共用。而 AMRpcHandler 是定义在服务器端的，只有服务端存在进程间通信 Rpc 消息？
- 去抓的 ET7 框架的源码，可以用来校正其它被自己改错的类，或是方法定义。
- 这个类比 AMHandler 要多传入一个模板类，主要用于处理那些约定好带返回数据的。**【注意下面，是参考网络上别人的理解】**。他们同样与活宝妹一样，也是一知半解，很多地方说得也未必对。
- 简单解释一下，现在 ET 协议数据主要分为两个类型，一种是来了消息直接自己处理的，另外一种是来了消息，自己处理完毕后，还需要将一些数据给返回的。**【带返回类型，和不带返回类型】**的消息。觉得它理解得不对，不适用于这里。因为 Rpc 这里感觉，更多的是说，进程间，或是现在 ET7 重构后的不同 SceneType 之间，比如注册登录服与网关服间的消息，内网消息等？Rpc-rpc-rpc...
- 主要的处理流程与 AMHandler 大体相同，需要注意的：
  - 传入的模板类有类型要求，除了是 class 外，第一个需要是实现 IRequest 接口，第二个是实现 IResponse 接口，他们分别对应了传进来的协议数据类型，以及需要返回的协议数据类型。
  - IRequest 类型，具有 RpcId，这个 id 用来标识一个传入协议数据，同时又将它设置到 response 返回数据中的 RpcId 中，这样发送数据返回的时候，就能找到那个和他具有相同 RpcId 传入协议数据，这种关系一对一，从而能进行进一步处理。**【谁发来的消息，就返回消息给谁——发送者】**
  - 回调函数 Reply ()，即当处理完传入数据后，需要马上装配好返回数据，并将其发送回去，所以需要一个回调函数将 response，通过 session 发送回去。前面说的是以前的框架。现在的回调过程，直接通过 Session 会话框走网络层将消息发回去，不用再弄个 Action<T> 来触发调用回调了。
  - 在 Run 中，需要传输上下文 session，接受的协议数据 request，需要返回的协议数据 response，以及回调函数 reply
- 亲爱的表哥，活宝妹的小鼠标到了，可是活宝妹的小小手，也感觉这个好小，捏不住，不好用!!!

```

public abstract class AMRpcHandler<Request, Response>: IMHandler where Request : class, IRequest where Response : class, IResponse
// ET7 框架里原本的：
protected abstract ETTTask Run(Session session, Request request, Response response);
public void Handle(Session session, object message) {
    HandleAsync(session, message).Coroutine();
}
private async ETTTask HandleAsync(Session session, object message) {
    try {

```

```

Request request = message as Request;
if (request == null)
    throw new Exception($" 消息类型转换错误: {message.GetType().Name} to {typeof (Request).Name}");
int rpcId = request.RpcId;
long instanceId = session.InstanceId;
Response response = Activator.CreateInstance<Response>(); // 创建一个消息的回复实例
try { // 【找个例子看一下，例子在下面】不懂：下面一句是在干什么，执行对发送来消息的处理，写返回数据？这里调的是抽象异步方法
// 猜测：应该更多可能是，通过不同服的具体实现，将返回数据写好？【是这样的】因为发送还在后面 session.Send(response) 是发送出去
    await this.Run(session, request, response);
}
catch (Exception exception) { // 如果出异常：写异常结果
    Log.Error(exception);
    response.Error = ErrorCode.ERR_RpcFail;
    response.Message = exception.ToString();
}
// 等回调回来，session 可以已经断开了，所以需要判断 session InstanceId 是否一样
if (session.InstanceId != instanceId)
    return;
response.RpcId = rpcId; // 在这里设置 rpcId 是为了防止在 Run 中不小心修改 rpcId 字段。【谁发来的消息，就返回消息给谁】
session.Send(response); // 把返回消息发回去，这里才是真正的发返回消息回请求端
}
catch (Exception e) { // 捕获异步操作过程中的异常
    throw new Exception($" 解释消息失败: {message.GetType().FullName}", e);
}
}

public Type GetMessageType() {
    return typeof (Request);
}
public Type GetResponseType() {
    return typeof (Response);
}
}
}

```

## 1.4 C2R\_LoginHandler: AMRpcHandler 的一个实体类，前后版本封装对比

- 上面的 AMRpcHandler 抽象方法 Run() 不是看不懂吗，找到一个例子，有前后两个版本的对比，可以参看理解一下
- 感觉今天下午，终于可以把先前这个自己总感觉迷迷糊糊的各种服的消息处理弄明白了
- 今天下午，把所有这类各种服的消息处理里的这些问题、编译错误全部改掉。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

```

// 框架中原本有这个方法，为什么我们需要把它改成现在的这个样子？
[MessageHandler(SceneType.Realm)]
public class C2R_LoginHandler : AMRpcHandler<C2R_Login, R2C_Login> {
    // 【ET7 的进一步精简封装】：封装在 AMRpcHandler 抽象实现类里
    protected override async ETTask Run(Session session, C2R_Login request, R2C_Login response) {
        // 随机分配一个 Gate
        StartSceneConfig config = RealmGateAddressHelper.GetGate(session.DomainZone());
        Log.Debug($"gate address: {MongoHelper.ToJson(config)}");
        ^I^I^I^I
        // 向 gate 请求一个 key，客户端可以拿着这个 key 连接 gate
        G2R_GetLoginKey g2RGetLoginKey = (G2R_GetLoginKey) await ActorMessageSenderComponent.Instance.Call(
            config.InstanceId, new R2G_GetLoginKey() {Account = request.Account});
        response.Address = config.InnerIPOutPort.ToString();
        response.Key = g2RGetLoginKey.Key;
        response.GateId = g2RGetLoginKey.GateId;
    }

    // 【ET-EUI 版本中的实现】：用来作 ET7 重构的对比参考。【下面的方法不要，不再需要】
    protected override void Run(Session session, C2R_Login message, Action<R2C_Login> reply) { // 【古老封装】自己看看
        R2C_Login response = new R2C_Login(); // 创建一个回复消息的实例：是因为没有实例，参数只有回调方法函数，所以必须得造一个。
        try {
            // if (message.Account != "abcdef" || message.Password != "111111") // 这里作必要的错误检测
            // {
            //     response.Error = ErrorCode.ERR_AccountOrPasswordError;
            //     reply(response);
            //     return;
        }
    }
}

```

```

        // }
        // 随机分配一个 Gate: 这里原理不变
        StartSceneConfig config = RealmGateAddressHelper.GetGate(session.DomainZone());
        Log.Debug($"gate address: {MongoHelper.ToJson(config)}");
        // 向 gate 请求一个 key, 客户端可以拿着这个 key 连接 gate
        G2R_GetLoginKey g2RGetLoginKey = (G2R_GetLoginKey) await ActorMessageSenderComponent.Instance.Call(
            config.InstanceId, new R2G_GetLoginKey() { Account = message.Account });
        // 手动: 写返回消息的内容
        response.Address = config.InnerIP0utPort.ToString();
        response.Key = g2RGetLoginKey.Key;
        response.GateId = g2RGetLoginKey.GateId;
        reply(response); // 通过调用回调函数, 将回调回调到调用端, 就是把写好的消息返回给调用端, 供它拿数据接通知等
    }
    catch (Exception e) { // 网络异步的过程中, 捕获异常
        ReplyError(response, e, reply);
    }
}
}
}

```

## 1.5 IMHandler 【ET-EUI】: 主要是与上面对比

- 因为对于 ET-EUI 里, 它为什么要再多自定义几个接口, 这个版块不够熟悉
- 我觉得把两个框架的几个接口与抽象实体类提出来对比理解一下是对的, 因为没明白, 为什么 ET7 里会重构成那个样子。可能也网上查下再帮助理解一下。

```

public interface IMHandler {
    void Handle(Session session, object message);
    Type GetMessageType();
    Type GetResponseType();
    // ETTask Run(Session session, M2C_TestActorMessage message); // 把这里去掉, 感觉加得不对
}

```

## 1.6 IMActorHandler: 【ET-EUI: 仅作参考】大概参考 ET-EUI 来的, 它的目的应该是把最基类的接口, 与其它两类的接口相区分开来

- 从这个小节开始, 主要是整理的 ET-EUI 里面的接口与抽象实现类。可能主要两个实现扩展类: (A|I) MHandler, (A|I) MRpcHandler, 都看一下
- 大概参考 ET-EUI 来的, 它的目的应该是把最基类的接口, 与其它两类的接口相区分开来。但是, 我想、猜测、理解的话, 应该是上面一个接口, 如果能够把 ETTask 与 ETVoid 很好地统一的话, 应该一个接口可能可以足以整个框架使用的。
- 消息处理器: 基本可以理解为, 总都是【服务端】的消息处理器。【客户端】更多的是发请求消息, 和接收消息。客户端很少涉及什么消息处理。
- 下面的接口, 哪个例子也没有参照, 自己改的。极有可能都是错的

```

public interface IMActorHandler {
    // 下面, 参考的是 ET-EUI 可能是 6.0 版本。ET7 里, 可能接口还可以简化, 还是 Actor 消息机制模块简化了, 不一定如下面这样
    void Handle(Entity entity, int fromProcess, object actorMessage);
    // ETTask Handle(Entity entity, object actorMessage, Action<IActorResponse> reply);
    Type GetRequestType();
    Type GetResponseType();
}

```

## 1.7 AMActorLocationHandler: 源码被我改动了

- 源码被我改动了, 正确性与否没有关系, 主要是帮助自己梳理一下几大不同的类型, 到改编译错误的时候, 能够边修改边弄明白。

```

[EnableClass]
public abstract class AMActorLocationHandler<E, Message>: IMActorHandler where E : Entity where Message : class, IAct
    // protected abstract ETTask Run(E entity, Message message);
    protected abstract void Run(E entity, Message message);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        if (actorMessage is not Message message) {
            Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Message).Name}");
            return;
        }
        if (entity is not E e) {
            Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Message).Name}");
            return;
        }
        ActorResponse response = new() {RpcId = message.RpcId};
        ActorHandleHelper.Reply(fromProcess, response);
        // await this.Run(e, message);
        this.Run(e, message);
    }
    public Type GetRequestType() {
        return typeof (Message);
    }
    public Type GetResponseType() {
        return typeof (ActorResponse);
    }
}

```

## 1.8 AMActorLocationRpcHandler: 上面接口实现类的一个使用例子。我把返回参数改了

```

[EnableClass]
public abstract class AMActorLocationRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : clas
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
            Response response = Activator.CreateInstance<Response>();
            try {
                //await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
    }
    public Type GetRequestType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
    // 这里涉及的就是那个接口方法的定义
    public ETTask Handle(Entity entity, object actorMessage, Action<IActorResponse> reply) => throw new NotImplementedException();
}

```

## 1.9 AMActorLocationRpcHandler: Rpc 就是进程间消息（或是 ET7 重构为 SceneType 之后的多核间消息）

```
[EnableClass]
public abstract class AMActorLocationRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : clas
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
            Response response = Activator.CreateInstance<Response>();
            try {
                //await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
    }
    public Type GetRequestType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
}
```

## 1.10 ActorHandleHelper 静态帮助类：包装了必要的方法，帮助自动化回复相关回调消息 et3

## 1.11 MessageDispatcherComponent: 这些是一个系统的，需要放在一起总结

```
// 总管: 对每个场景, 消息分发器
// 这个类, 可以简单地理解为: 先前的各种服, 现在的各种服务端场景, 它们所拥有的消息处理器实例的封装。那么默认, 每种场景, 只有一个消息处
public class MessageDispatcherInfo {
    public SceneType SceneType { get; }
    public IMHandler IMHandler { get; }
    public MessageDispatcherInfo(SceneType sceneType, IMHandler imHandler) {
        this.SceneType = sceneType;
        this.IMHandler = imHandler;
    }
}
// 消息分发组件
[ComponentOf(typeof(Scene))]
public class MessageDispatcherComponent: Entity, IAwake, IDestroy, ILoad {
    public static MessageDispatcherComponent Instance { get; set; }
    public readonly Dictionary<ushort, List<MessageDispatcherInfo>> Handlers = new(); // 总管的字典
}
```

## 1.12 MessageDispatcherComponentSystem:

```
// 扫描框架里的标签系【MessageHandler(SceneType】】
private static void Load(this MessageDispatcherComponent self) {
    self.Handlers.Clear();
    HashSet<Type> types = EventSystem.Instance.GetTypes(typeof (MessageHandlerAttribute));
    foreach (Type type in types) {
        IMHandler iMHandler = Activator.CreateInstance(type) as IMHandler;
        if (iMHandler == null) {
            Log.Error($"message handle {type.Name} 需要继承 IMHandler");
            continue;
        }
        object[] attrs = type.GetCustomAttributes(typeof(MessageHandlerAttribute), false);
        foreach (object attr in attrs) {
            MessageHandlerAttribute messageHandlerAttribute = attr as MessageHandlerAttribute;
            Type messageType = iMHandler.GetMessageType();
            ushort opcode = NetServices.Instance.GetOpcode(messageType); // 这里相对、理解上的困难是：感觉无法把 OpCode 网络操
            if (opcode == 0) {
                Log.Error($" 消息 opcode 为 0: {messageType.Name}");
                continue;
            } // 下面：下面是创建一个包装体，注册备用
            MessageDispatcherInfo messageDispatcherInfo = new (messageHandlerAttribute.SceneType, iMHandler);
            self.RegisterHandler(opcode, messageDispatcherInfo);
        }
    }
}

private static void RegisterHandler(this MessageDispatcherComponent self, ushort opcode, MessageDispatcherInfo handler) {
    if (!self.Handlers.ContainsKey(opcode))
        self.Handlers.Add(opcode, new List<MessageDispatcherInfo>());
    self.Handlers[opcode].Add(handler); // 加入管理体系来管理
}

public static void Handle(this MessageDispatcherComponent self, Session session, object message) {
    List<MessageDispatcherInfo> actions;
    ushort opcode = NetServices.Instance.GetOpcode(message.GetType());
    if (!self.Handlers.TryGetValue(opcode, out actions)) {
        Log.Error($" 消息没有处理: {opcode} {message}");
        return;
    }
    // 这里就不明白：它的那些 Domain 什么的
    SceneType sceneType = session.DomainScene().SceneType; // 【会话框】：哈哈哈，这是会话框两端，哪一端的场景呢？感觉像是会话框的
    foreach (MessageDispatcherInfo ev in actions) {
        if (ev.SceneType != sceneType)
            continue;
        try {
            ev.IMHandler.Handle(session, message); // 处理分派消息：也就是调用 IMHandler 接口的方法来处理消息
        } catch (Exception e) {
            Log.Error(e);
        }
    }
}
```

## 2 C# 异步基础原理、状态机原理、逻辑整理

### 2.1 ETVoid C# Net-async|await 编程更底层一点儿的原理

- 就是不懂底层的原理是什么，方法定义是什么，返回的是什么，在有 await 等关键字的时候，返回的内容等是如何变换的，以及它背后的那个异步状态机，就是想不明白。
- 现在参考网上的一个例子，记一下异步任务 C# 幕后封装的那些执行步骤什么的，把 async await 之类的关键字，背后的逻辑理解明白。
- 就是，可能也可以在异步任务的这个模块，添加无数的日志，通过读日志来把这块儿弄明白。下面就截图网上的这个参考例子。

```
[AsyncMethodBuilder(typeof (AsyncETVoidMethodBuilder))]
internal struct ETVoid : ICriticalNotifyCompletion {
    [DebuggerHidden]
    public void Coroutine() { }
    [DebuggerHidden]
```

```

public bool IsCompleted => true;
[DebuggerHidden]
public void OnCompleted(Action continuation) { }
[DebuggerHidden]
public void UnsafeOnCompleted(Action continuation) { }
}

```

- 上面是找了一个最短小的类 ETVoid, 网上例子自己构建一个类, 这个类麻雀虽小五脏俱全的几个【缺一不可】的方法 (所以知道 ETTask|ETVoid 自定义封装, 这几个方法也是一定不能少的, 只是多了 Coroutine() 方法不知道是怎么回事儿?) 如上如下:

我们尝试去构造一个TaskAwaiter, 看看await里面如何调用。

```

/// <summary>
/// 这里根据TaskAwaiter, 构造一个一模一样的类
/// 一下的方法缺一不可, 不然await就会有提示报错
/// </summary>
2 个引用
class mAwaiterClass<T> : ICriticalNotifyCompletion, INotifyCompletion
{
    0 个引用
    public void OnCompleted(Action continuation)
    {
        Console.WriteLine("###mAwaiterClass OnCompleted");
        continuation?.Invoke();
    }
    /// <summary>
    /// 把next的action传进来, 在这个方法里, 决定什么时候继续往下走
    /// </summary>
    /// <param name="continuation"></param>
    0 个引用
    public void UnsafeOnCompleted(Action continuation)
    {
        Console.WriteLine("###mAwaiterClass UnsafeOnCompleted");
        Thread.Sleep(1000);
        continuation?.Invoke();
    }
    0 个引用
    public bool IsCompleted
    {
        get
        {
            Console.WriteLine("###mAwaiterClass IsCompleted");
            return false;
        }
    }
    0 个引用
    public int GetResult()
    {
        Console.WriteLine("###mAwaiterClass GetResult");
        return 1 ;
    }
}

```

知乎 @JJMIMIDA

- 它的测试用例是这么写的: 注意它传入的参数类型是 int. 后面的编译码里, 和它的讲解里会用到提到。

```
1 个引用
class AwaitTest<T>
{
    1 个引用
    public mAwaiterClass<T> GetAwaiter() { return new mAwaiterClass<T>(); }
}

0 个引用
class Program
{
    0 个引用
    static void Main(string[] args)
    {
        asyncMethod();
        Console.ReadLine();
    }

    1 个引用
    async static Task asyncMethod()
    {
        int a =await new AwaitTest<int>();
        Console.WriteLine("###asyncMethod GetResult:"+ a);
    }
}
```

知乎 @JJMIMIDA

- 看它编译出来的码（那堆编译出来的状态机的码），就是看不懂

```

// awaiter = new AwaitTest<int>().GetAwaiter();
IL_0000: nop
IL_0001: newobj instance void class AysncTest.AwaitTest`1<int32>::ctor()
IL_0014: call instance class AysncTest.mAwaiterClass`1<!0> class AysncTest.AwaitTest`1<int32>::GetAwaiter()
IL_0019: stloc.1
// if (!awaiter.IsCompleted)
IL_001a: ldloc.1
IL_001b: callvirt instance bool class AysncTest.mAwaiterClass`1<int32>::get_IsCompleted()
IL_0020: brtrue.s IL_0062

// num = (<>1__state = 0);
IL_0022: ldarg.0
IL_0023: ldc.i4.0
IL_0024: dup
IL_0025: stloc.0
IL_0026: stfld int32 AysncTest.Program/'<asyncMethod>d__1'::'<>1__state'
// <>u_1 = awaiter;
IL_002b: ldarg.0
IL_002c: ldloc.1
IL_002d: stfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u_1'
// <asyncMethod>d__1 stateMachine = this;
IL_0032: ldarg.0
IL_0033: stloc.2
// <>t_builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
IL_0034: ldarg.0
IL_0035: ldflda valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder Ays
IL_003a: ldloca.s 1
IL_003c: ldloca.s 2
IL_003e: call instance void [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder:::
// return;
IL_0043: nop
IL_0044: leave.s IL_00c3

// awaiter = (mAwaiterClass<int>)<>u_1;
IL_0046: ldarg.0
IL_0047: ldfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u_1'
IL_0048: castclass class AysncTest.mAwaiterClass`1<int32>
IL_0051: stloc.1
// <>u_1 = null;
IL_0052: ldarg.0
IL_0053: ldnull
IL_0054: stfld object AysncTest.Program/'<asyncMethod>d__1'::'<>u_1'
// num = (<>1__state = -1);
IL_0059: ldarg.0
IL_005a: ldc.i4.m1
IL_005b: dup
IL_005c: stloc.0
IL_005d: stfld int32 AysncTest.Program/'<asyncMethod>d__1'::'<>1__state'
// <>s_2 = awaiter.GetResult();
```

知乎 @JJMIMIDA

asyncMethod里面的真面目

```

summary>
F:\C#\AysncTest\bin\Debug\netcoreapp3.1\A
###mAwaiterClass IsCompleted
###mAwaiterClass UnsafeOnCompleted
||###mAwaiterClass GetResult
###asyncMethod GetResult:1
知乎 @JJMIMIDA
```

## • 结果分析：【异步方法状态机，背后的执行顺序与逻辑：】

- 先检查 IsCompleted 标志位，如果已经完成，则调用 GetResult 作为 await 的返回值返回。
- 如果未完成，经过 AsyncTaskMethodBuilder 的 AwaitUnsafeOnCompleted 方法之后，最后进入 UnsafeOnCompleted (nextAction)，并且把完成后的下一步回调传进来。
- 当我们获得 nextAction 之后，说明该调用由我们自己来控制，这里我在等待 1s 之后，执行 nextAction ()，下一步 GetResult 返回。

## • 【Async 关键字方法的编译原理：】

对于上述问题2：带async签名的方法编译之后生成了什么？为什么带async签名的方法返回值一定是void、Task、Task<T>？

```
.class nested private auto ansi sealed beforefieldinit <asyncMethod>d_1'
    extends [System.Runtime]System.Object
    implements [System.Runtime]System.Runtime.CompilerServices.IAsyncStateMachine
{
    .custom instance void [System.Runtime]System.Runtime.CompilerServices.CompilerGeneratedAttribute::ctor() = 
        01 00 00 00
    )
    // Fields
    .field public int32 '><1__state'
    .field public valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder '><t__builder'
    .field private int32 '><s_1'
    .field private int32 '><s_2'
    .field private object '><o_1'

    // Methods
    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed ...
    {
        .method private final hidebysig newslot virtual
        instance void MoveNext () cil managed ...
    }

    .method private final hidebysig newslot virtual
        instance void SetStateMachine (
            class [System.Runtime]System.Runtime.CompilerServices.IAsyncStateMachine stateMachine
        ) cil managed ...
    }

} // end of class <asyncMethod>d_1
```

- 这个 async 关键字所标记的异步方法，主要两个点儿：
  - 编译器，把这个异步方法，编译成了一个类 class <asyncMethod>d\_1;
  - 这个类 class, 它实现了 IAsyncStateMachine 接口，(实现了这个接口，返回的是什么类型呢？这个要想明白?)
  - 这个类 class, 的内部，有几个成员变量.field-sss.
  - 这个类 class, 的内部，有个特别重要的状态机执行函数 MoveNext() 来指挥指导，异步函数内不同节点如 await 节点等的执行逻辑。【这个类 class, 它实现了 **IAsyncStateMachine** 接口】，前面有列出 IAsyncStateMachine 接口定义的两个方法，所以实现实体类里也会有 SetStateMachine() 方法的实现。
  - 上面的逻辑，其实就是在扫描异步方法内，不同的 await 调用，每到一个这个关键字申明的异步调用，就是切换一个状态（背后有可能是线程的切换，不一定每个分支都用不同的线程，但线程的切换可能是，必要的时候需要切换的？）分段执行。
- 网络上的分析者还给出了下机的截图：不是狠懂，这个截图是什么意思？因为不懂，要把编译码的方法名带上，方便以后再读和理解。

```
: AysncTest.Program/'<asyncMethod>d_1',
type [System.Threading.Tasks]System.Runtime.CompilerServices.AsyncTaskMethodBuilder

obj instance void AysncTest.Program/'<asyncMethod>d_1'::ctor()
{
    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed ...
    {
        .method private final hidebysig newslot virtual
        instance void MoveNext () cil managed ...
    }

    .method private final hidebysig newslot virtual
        instance void SetStateMachine (
            class [System.Runtime]System.Runtime.CompilerServices.IAsyncStateMachine stateMachine
        ) cil managed ...
    }

} // end of class <asyncMethod>d_1
```

- 上面的异步方法，所生成的异步状态机类 class 里，有几个主要的方法：
  - 构造器方法 ctor():
  - Create() 方法：
  - Start() 方法：

- get\_Task() 方法:
- 可是上面的几个方法是谁，哪个接口定义的呢？
- 网络上的分析者，对上面两个截图的分析如下：【它讲解的这部分，我可能还是得自己编译一下，去具体看一下。】因为它的截图不完整，看不懂下面还有个别人总结的状态机套路，感觉说得更彻底透彻。
  - 签名为 async Task asyncMethod() 的方法里，先创建一个继承自 IAsyncStateMachine 的 asyncMethod 类
  - 创建一个 AsyncTaskMethodBuilder，然后赋值给 Machine.（不知道，它这句，说的是哪里？第一个图的最后 SetStateMachine()？）
  - 初始化 Machine 的 state = -1.（两个截图里看不见，找不到）
  - 调用 AsyncTaskMethodBuilder.Start 方法，start 里面会进入 Machine 的 moveNext () 方法，详见问题 1。
  - AsyncTaskMethodBuilder.get\_Task () 作为该方法的返回值返回。
- 多线程问题：Task 一定是多线程吗？
  - 不一定，在上述例子中，我们定义的 async static Task<int> aa()，里面就是在同一个线程执行的。只有调用 Task.Start 或者 Task.Run 里面自动启用多线程的时候，才是多线程。
- 看得另一个网页中的说法，因为感觉它也没有实现个什么公共定义约束的接口，理解得不够透彻。看下下面的：
- await 必须配合 Task/ValueTask 才能用吗？当然不是。
  - 在 C# 中 只要你的类中包含 **GetAwaiter()** 方法和 **bool IsCompleted** 属性，并且 **GetAwaiter()** 返回的东西包含一个 **GetResult()** 方法、一个 **bool IsCompleted** 属性和实现了 **INotifyCompletion**，那么这个类的对象就是可以 await 的。这里说得还是不清楚，不透彻，换一个表达得更清晰的说法如下：
  - 可以使用 await 的方法，返回值必须是 **awaitable 对象**，自定义 awaitable 对象比较麻烦，一个对象必须满足下列条件才行：
    - 必须有一个 **GetAwaiter()** 方法，扩展方法或者实例方法都可以
    - GetAwaiter() 方法返回值必须是 **awaiter 对象**。一个对象要成为 awaiter 对象必须满足下列条件：
      - \* 该对象 实现接口 **INotifyCompletion** 或者 **ICriticalNotifyCompletion**
      - \* 必须有 **IsCompleted** 属性
      - \* 必须有 **GetResult()** 方法，可以返回 void 或者其他返回值。
  - 比如下面的自定义类：把几个类的本质理解得再深一点儿了吗？【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

```

public class MyTask<T> {
    public MyAwaiter<T> GetAwaiter() { // 必须提供的方法
        return new MyAwaiter<T>();
    }
}

// 下面自定义的类 MyAwaiter<T> 亲爱的表哥 > 就是可以 await 的：
// 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥，爱生活!!】
public class MyAwaiter<T> : INotifyCompletion { // 必须实现的接口
    public bool IsCompleted { get; private set; } // 属性变量
    public T GetResult() { // 必须要有的方法
}

```

```

        throw new NotImplementedException();
    }
    public void OnCompleted(Action continuation) {
        throw new NotImplementedException();
    }
}
public class Program {
    static async Task Main(string[] args) {
        var obj = new MyTask<int>();
        await obj;
    }
}

```

- 【状态机套路】:

- `async` 关键字标记方法是一个异步方法，编译器通过这个标记 **【async 关键字】** 去改造这个方法体为创建状态机的方法。`await` 是关键字，是为了实现状态机中的一个状态，每当有一个 `await`，就会生成一个对应的状态。状态机就是根据这个状态，去一步步的调用异步委托，然后回调，包括状态机的解析。
- (1). 状态机的默认状态都是-1, 结束状态都是-2.
- (2). 每 `await` 一次就会产生一个 `TaskAwaiter awaiter`; 改变状态机的状态, 当有多个 `await` 的时候, 每个 `await` 都会改变状态机的状态, 比如改为 0,1,2,3,4 等等, 分别表示代码中 `await xxx` 这句话执行完成。
- (3). 状态机的执行套路:
  - A. 首先创建一个 `d_num` 的方法 (这里说错了, 应该是创建了一个类 `class`) , `xxx` 代表方法名, `num` 可能是 0,1,2,3 等, 实现 **IAsyncStateMachine** 接口。
  - B. 在 `MoveNext()` 方法中, 源代码中每个 `await xxxx` 都会对应生成是一个 `TaskAwaiter awaiter`, 然后 `xxxx.GetAwaiter()`
  - C. 判断状态机是否执行完 `if (!awaiter.IsCompleted)`,
    - \* 没有执行完的话走 `<>t _builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);` 代表释放当前线程
    - \* 执行完后走, `<>s_1 = awaiter.GetResult();` 拿到返回值, 继续走后面的代码。
- (此处写的比较抽象, 看下面 3 结合代码编译再分析)
- 感觉今天读这个状态机: <https://linuxcpp.0voice.com/?id=1380> 终于有点儿开窍了!!【爱表哥, 爱生活!!! 任何时候, 活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥, 爱生活!!!】

## 2.2 如果方法声明为 `async`, 那么可以直接 `return` 具体的值, 不再用创建 `Task`, 由编译器创建 `Task`:

```

// 只要标记了 async 就会被编译成状态机
// 如果方法声明为 async, 那么可以直接 return 具体的值, 不再用创建 Task, 由编译器创建 Task:
public static async Task<int> F2Async() {
    return 2;
}

```

- F2Async: 只加了 `async`, 会生成状态机, 但由于没有加 `await` 所以不会涉及到中间状态的变化, 从-1 默认状态变为结束的-2 状态。
- F3Async: 既有 `async` 也有 `await` (`await` 只有 1 个), 该方法是使用了 `Task.Run`, 我们把它归为计算型的异步方法。
- 亲爱的表哥, 活宝妹今天终于把这个看得稍微有点儿懂了, 希望能够赶快从这个 `ETTask` 模块 `move-forward`. 任何时候, 活宝妹就是一定要嫁给亲爱的表哥!!! 活宝妹还没能嫁给亲爱的表哥, 活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥, 爱生活!!!

3 现在的修改内容：【任何时候，活宝妹就是一定要嫁给亲爱的表哥！！！爱表哥，爱生活！！！】几大版块

**3.1 框架里现存的错误：截图，明天上午把功能模块一一看一遍，现在不知道怎么改了，要再看再学习了才会改**

- **【HandCardSprite.cs】**: 这个客户端文件里存在一堆关于 Unity 引用的错误。把这个有着巨多错误的类重新添加到了框架里。现在看着这些错误（加了这个文件，错误又多了二三十个!!!）。

- 【参考项目 Game.cs】客户端类里，存在 UnityEngineer 的诸多引用，所以 HandCard-Sprite.cs 可以通过 Game.EventSystem 等拿到引用。但 ET7 重构得没有边际。必须自己去看明白。这个类，更多的是，适配特定游戏需求的 ET7 框架外的一个桥梁适配类。

- 【参考项目热更域里的 Game.cs 类】:

- 现框架里不存在的，需要整合进来的模块版块：DBProxyComponent, InnerConfig, LocationComponent, StartConfigComponent
  - 组件管理类：某些组件，属于双端，但客户端与服务端的逻辑不一样，如 PlayerComponent；某些组件，只属于服务端；有只属于客户端的吗？
  - **【GamerComponent】**：它的逻辑设计应该是什么样的？当服务端有 PlayerComponent 对所有玩家进行管理，当前 GamerComponent 只管理一个拖拉机房间里的四个玩家，是 RoomComponent 的玩家组成对象（？还有房间组成对象，因为房间如玩家一样需要管理，对应不同拖拉机房间号）\ul>  - 参考项目放在热更新域里面，但是现项目是不允许申明组件放在热更新域里的。去参考项目中其它组件是否全在 Model 层申明组件，以及成员变量。暂时把它放到 Model 双端共用的地方。
  - 台式机好慢好慢，找了好久才找到这个类。现在应该可以往下改了。这个模块，今天就暂时改到这里，看不见什么相关的编译错误了

- 这里看出 ET 框架的局限：它把一切成员变量之类的在 Model 层里固定死了，也就意味着，热更新是无法热更新功能逻辑模块的重构，只能热更新小细节的实现逻辑。
- 【GamerComponent 管理类组件】：逻辑没有理清楚。它是服务端组件，还是客户端组件，还是如 PlayerComponent 双端组件，并实现不同的逻辑？

## 3.2 【ComponentFactory】现在就可以清理掉这个组件的所有了

```

Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'
Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'
Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Hotfix.dll' could not be found
Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found
Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found
Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found
Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.ModelView.dll' could not be found
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The name 'ComponentFactory' does not exist in the current context
The type or namespace name 'BaseEventData' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'ComponentWithId' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'ComponentWithId' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'EnterMapHelper' does not exist in the namespace 'ET.Client' (are you missing an assembly reference?)
The type or namespace name 'EnterMapHelper' does not exist in the namespace 'ET.Client' (are you missing an assembly reference?)
The type or namespace name 'Events' does not exist in the namespace 'UnityEngine' (are you missing an assembly reference?)
The type or namespace name 'EventSystems' does not exist in the namespace 'UnityEngine' (are you missing an assembly reference?)
The type or namespace name 'GamerComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'GamerComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'MonoBehaviour' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)
```

The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	GameFactory.cs	7
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	MatchFactory.cs	8
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	UserFactory.cs	8
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	MH2MP_CreateRoom_ReqHandle..	8
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	C2R_Register_ReqHandle.cs	19
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	C2R_Register_ReqHandler.cs	24
The name 'ComponentFactory' does not exist in the current context	DotNet.Hotfix	MatchComponentSystem.cs	54
The type or namespace name 'ComponentWithId' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Login_ReqHandler.cs	12
The type or namespace name 'ComponentWithId' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Register_ReqHandler.cs	12
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2G_GetUserInfo_ReqHandler.cs	17
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2G_GetUserInfo_ReqHandler.cs	17
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Login_ReqHandler.cs	9
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Login_ReqHandler.cs	9
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Register_ReqHandler.cs	10
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2R_Register_ReqHandler.cs	10
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	GameControllerComponentSyst..	241
The type or namespace name 'DBProxyComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	GameControllerComponentSyst..	241
The type or namespace name 'EnterMapHelper' does not exist in the namespace ET.Client (are you missing an assembly reference?)	DotNet.Hotfix	RobotCaseSystem.cs	54
The type or namespace name 'EnterMapHelper' does not exist in the namespace ET.Client (are you missing an assembly reference?)	DotNet.Hotfix	RobotCaseSystem.cs	70
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	C2G_LoginGate_ReqHandler.cs	27
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	MapHelper.cs	11
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	RealmHelper.cs	12
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	SessionUserComponentSystem.cs	18
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	SessionUserComponentSystem.cs	31
The type or namespace name 'InnerConfig' could not be found (are you missing an assembly reference?)	DotNet.Hotfix	MatchComponentSystem.cs	50
The type or namespace name 'InnerConfig' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	Unity.Hotfix_C2G_ReturnLobby_NthHandler.cs	15
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	SceneFactory.cs	39
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	ObjectAddRequestHandler.cs	7
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	ObjectGetRequestHandler.cs	7
The type or namespace name 'LocationComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	ObjectLockRequestHandler.cs	7
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	ObjectUnLockRequestHandler.cs	6
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2G_LoginGate_ReqHandler.cs	24
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	C2G_LoginGate_ReqHandler.cs	24
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	MapHelper.cs	11
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	RealmHelper.cs	11
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	SessionUserComponentSystem.cs	14
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	SessionUserComponentSystem.cs	14
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	AllotMapComponentSystem.cs	12
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	Unity.Hotfix_C2G_ReturnLobby_NthHandler.cs	11
The type or namespace name 'StartConfigComponent' could not be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	Unity.Hotfix_C2G_ReturnLobby_NthHandler.cs	11
禁止在Entity类中直接调用Child.GetComponent	DotNet.Hotfix	DeskCardsCacheComponentSyst..	27
禁止在Entity类中直接调用Child.GetComponent	DotNet.Hotfix	AllotMapComponentSystem.cs	12

• 【ComponentFactory】重构了的框架里，这个工厂类是被拆解到各自小部件的生产工厂里去了，就是一个框架底层封装的工厂类，拆解到 100 个不同的小部件里。所以我必须得要每个使用的小部件里，它的生产工厂里去再调用相应的逻辑。【可以找个例子出来看一下】

- Entity 类里面有，组件里添加一个新 new 出来的成员的办法。模仿 Player 的使用例子。这里的使用方法是：去拿它的管理组件的实例索引，用管理组件来生成各个元件

• 【PlayerSystem】：不是不知道框架里怎么用，找不出来一个使用的例子吗？它可能不需要用，它只需要框架底层的 Entity 里相关方法的封装，能够生成一个一个小单元(Player,Gamer,Matcher-etc) 之类的就可以了。就是框架底层原理，这一块儿的，还不太懂

- 这个工厂类，总是不懂，先去把基类 Entity.cs 好好再读一下
- 同样套用的话，GamerComponent 是房间组件的子组件，拿到这个组件后来创建. VSC 里面好像是有多余的类，所以从 VSC 里看源码，比较乱一点儿。【感觉这一块儿的思路，还没能理清楚。】
- Hotfix Server 【UnitFactory】生成创建一个单位。可以用作例子。unitComponent.AddChildWithId 调用的是 Entity 里最底封装逻辑。
  - \* 这个 UnitFactory 调用组件方法，来添加进自己的管理系，它所添加的组件是有独特身份 ID 的，不适用当前例子
  - \* 需要去找，自动生成特异性 ID，并创建实例的 Entity 里的方法的例子
- 上面的问题是，如果框架热更新域里可以如上 UnitFactory 一样添加工厂类，那么我的其它小单位 Gamer, Player 应该也是可以如上 Unit 一样提供他们自己的工厂生产类才对。
- 再试着多找几个如上的工厂生产类的例子看看。

• 【ComponentFactory.CreateWithId】重构了的框架里，这个工厂类是被拆解到各自小部分的生产工厂里去了，就是一个框架底层封装的工厂类，拆解到 100 个不同的小部件里。所以我必须得要每个使用的小部件里，它的生产工厂里去再调用相应的逻辑。【可以找个例子出来看一下】新框架里，上次不是找到过：先去拿管理器组件，再用管理器组件，通过调用基类 Entity 里的方法，来创建小部件的实例？可以再找个例子看一下

- 上午把【数据库模块的接入】、【InnerConfig】、【StartConfigComponent】、【LocationComponent】等相关模块：读下源码，理解透彻，必要的情况下下午家里接入并测试
  - 把 ActorLocation 相关的，今天晚上一个小时左右，再读一下
  - 【数据库模块】现框架中被砍得几乎不剩下什么，如果弄不懂，可以与参考项目斗地主中的数据库模块作对照来理解，和添加必要的类等。
  - 如果这一块儿，这会儿还没什么头绪，就先看看别的呀
  - 我听不见他们长辈的政治洗脑文，听不见他们的政治立场表忠心，活宝妹是美国人，活宝妹的余生会在美国度过。活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远也不再回中国大陆。
  - 被他们的政治洗脑听得恶心吧拉。。。先写两个算法题摆脱一下，被恶心得真难受。。。
- 把 IMHandler 接口，以及它的两个抽象实现类、相关模块的部分理解透彻
- 先看只看了一半的，比如什么 IMHandler 接口的实现类等，如果今天上午有机会，也可以看一下
- InnerConfig 和 StartConfigComponent 也都稍微看一下：感觉 ET7 重构后，这些自己基础相对欠缺的模块，还没仔细看
- 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】
- 其它几个杂项错误：

CS1503	Argument 1: cannot convert from 'ET.Entity' to 'ET.Scene'	DotNet.Hotfix	HttpComponentsystem.cs	82
CS1503	Argument 1: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	C2G_LoginGate_ReqHandler.cs	28
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	MapHelper.cs	12
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	RealmHelper.cs	13
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	SessionUserComponentsystem.cs	19
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	SessionUserComponentsystem.cs	32
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	MatchComponentsystem.cs	52
CS1503	Argument 2: cannot convert from 'System.Net.IPEndPoint' to 'long'	DotNet.Hotfix	C2G_ReturnLobby_Ntthandler.cs	18
IDE100	Error reading content of source file F:\ET\Unity\Assets\Scripts\Codes\Model\Server\Tractor\Entity\DB\UserInfo.cs..	DotNet.Model	Unity.Model.UserInfo.cs	1
ET0013	静态类ET卡牌数据引用存在环形依赖 请修改为单向依赖 静态类ET.Server.CardsHelper被 静态类ET.Server.DeskCardsCacheComponentSystem 引用	DotNet.Hotfix	CSC	1
ET0013	静态类Deck数据引用存在环形依赖 请修改为单向依赖 静态类ET.Server.DeskCardsCacheComponentSystem被 静态类ET.Server.CardsHelper 引用	DotNet.Hotfix	CSC	1

- 【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥！！任何时候，活宝妹就是一定要嫁给亲爱的表哥！！爱表哥，爱生活!!!!】
- 【任何时候，亲爱的表哥的活宝妹，就是一定要嫁给亲爱的表哥！！！爱表哥，爱生活!!!!】
- 因为框架很大，是一个大型网络游戏双端框架，因为内容比较多，现在已经总结的是四个文件，还要常作笔记，否则容易忘记，前后不连贯。所以难免小细节的地方，没能注意到，没什么大不了

### 3.3 热更新层的生成系

- 【热更新层的生成系】：下午家里试着把 Component 组件再添加回去试试看 【不能再添加 Component 组件。ET7 框架重构了，小单元也走热更新，在热更新层有天文小行星的生成系。可以参照 ET.pdf 里的服务端 PlayerSystem 来作例子】？上午把项目设计的思路，源项目的破源码再读一读理一理，是希望游戏逻辑与游戏界面能够快速开发、项目进展往后移的。
  - 【热更新层的生成系】：不少组件，我急着添加热更新层的生成系的时候，可能忽略了某些必要、不必要的 Awake() 系统。运行时如果抛错，可以补上必要的 Awake() 等必要的框架系统方法。比如：LandlordsGateSessionKeyComponentSystem，它需要 Awake() 系统吗？再列一个报错的例子：C2G\_LoginGate\_ReqHandler。【改法】：如果不强改成是单个游戏逻辑，ET 框架里有这个逻辑处理，可以去参考原框架的写法与生成系是如何自动绑定的。
  - User.cs 客户端的话，不知道要不要修改。晚点儿的时候留意一下。

- Gamer.cs 客户端保留了 Dispose
- **【IMHandler】:** 在 ET7 的框架里, Handle() 方法的定义, 主要是 Actor 消息机制大概又重构得更为自动化一点儿, 当有分门别类的 ActorMessageHandler 标签系实体类, 大概 ET7 框架里只需要调用接口的申明方法就可以了? 总之, 就是 Handle() Run() 两大类方法的方法定义发生了变化
  - 在 IMHandler 的两个抽象实现类的封装里, ET7 构架重构后, 各种服需要自定义的服务器处理逻辑被要求实现在 Run() 方法里; 而抽象类定义的 Handle() 方法里, 自动封装实现了带回复消息的请求消息的自动回复(通过抽象类里实现方法调用 Session.Send(XXX))。这个对需要返回消息的请求消息的自动回复的封装, 有利有不利
    - \* 利是: 当顺序不重要, 可以自动回复时, 由框架的底层帮实现自动回复, 方便
    - \* 不利是: 当顺序变得重要, 当回复消息后, 某个某些服还有其它逻辑需要服务器来处理, 这个框架底层的自动封装就会成为一个 blocker, 需要自己想办法去解决这些特殊服, 要如何实现必要情况下的, 先回复消息, 再处理服务器端的其它逻辑。**【这儿哪里作过这个笔记, 有个细节的服务器端的实现处理逻辑, 顺序重要】**
  - AMRpcHandler 的实体实现类, 我可能还需要再多找几个出来看下
  - 今天下午弄这些 IMHandler 以及两个抽象实现类, 和它们服务端的消息处理器类, 编译氏错误一堆, 感觉昨天下午基本都消干净了。接着崩出来的 214 个框架里的其它错误都能一一解决, 昨天晚上消掉了大概 100 个左右的编译错误。它们是框架里的细节, 是帮助活宝妹理解这个框架的方方面面点点滴滴的必经步骤。活宝妹不怕它们, 也没什么可怕的。。。**【爱表哥, 爱生活!!! 任何时候, 活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥, 爱生活!!!!】**

### 3.4 内网消息等网络相关: 请求消息的发送方法等。很多编译错误, 要一点儿一点儿把他们都改掉

- **【内网消息等网络相关: 请求消息的发送方法等】:** 在构架里是怎么写的, 有几种请求消息的发送方式?
- **明天上午把这块看完, 等着我改的编译错误包括:** 参考的斗地方游戏里, 各种服处理返回消息的逻辑。
  - 因为先前手动发送每个返回消息, 我需要将这部分一批消息处理器改为, 先试着适配 ET7 框架的重构与底层再封装。
  - 等改过了, 真正明白理解了自己重构游戏的需求, 再来看去看 ET7 框架我要怎么改它现有封装, 才能适配自己游戏的需求!! 例子: MatchComponentSystem 里的 JoinRoom 方法等相关逻辑。
  - **【下午还没有改到这里来。先从简单的改起, 因为一个热键的优化, 感觉 VS 好用一点儿了。先能改多少改多少, 再按模块来改像消掉所有的 ETTask 相关一样把一个模块的所有编译错误全部改完!!!】**
- 去看上面列过的那个例子 MatchComponentSystem, 参考项目里的各种服的消息处理, 怎么适配成 ET7 重构后的不用手动发返回消息(发送过程封装在框架底层), 和记录可能存在的问题(某些服的逻辑, 返回消息的发送时间与其它必要逻辑, 顺序变得重要的时候, 记下来, 晚点儿会再重构 ET7 框架适配游戏需求)

### 3.4.1 修改下面的 ActorMessageSenderComponent 因为功能模块逻辑重构，而带来的一堆编译错误。

- 修改方法过程步骤：去框架里搜索，其它任何地方发送消息的例子，看 【重构后的框架是如何发送消息的，Call() Send() 方法的调用等】这个明天上午一定看，因为不懂，不会改怎么发送消息的()
- 然后参照例子，把客户端和必要的小服里，所有需要发送消息的地方，改成上面看到总结的发送方法里。

CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	G2M_PlayerEnterMatch_ReqHan...
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	G2M_PlayerEnterMatch_ReqHan...
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	SessionUserComponentSystem.cs
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	TrusteeshipComponentSystem.cs
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	GameControllerComponentSyst...
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	MatchComponentSystem.cs
CS1929	'ActorMessageSenderComponent' does not contain a definition for 'Get' and the best extension method overload 'GateSessionKeyComponentSystem.Get(GateSessionKeyComponent, long)' requires a receiver of type 'GateSessionKeyComponent'	DotNet.Hotfix	MatchComponentSystem.cs
CS1501	No overload for method 'Call' takes 1 arguments	DotNet.Hotfix, Unity.Hotfix	C2G_ReturnLobby_Ntthandler.cs 28
CS1501	No overload for method 'Get' takes 2 arguments	DotNet.Hotfix	Actor_PlayerEnterRoom_ReqHan... 55
CS1501	No overload for method 'Send' takes 3 arguments	DotNet.Hotfix	Actor_PlayerEnterRoom_ReqHan... 90

- 【地图服 Unit 相关】：先前所有接触到这个框架，都只看了个头，就是只限于能够任何客户端连接到服务端能够注册登录的程度，后面的其它服、框架逻辑全都还不曾看。所以今天上午扫一眼地图服相关，是糊的。要把这些前前后后相关的原理总弄懂了。
- 去框架里搜发送的调用方法，可能现在 Mac 系统里有一点儿障碍的，就是 VSC 不报错，不知道搜出来的是对的，还是错的。但是几种不同的方法，先总结在这里，对照运行时的报错一一改过来。必须把这块儿弄明白了。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】
- 【拿到 Session 会话框，调用其 Send() 方法】：例子 PingComponentAwakeSystem 里的 PingAsync() 方法。它是一个心跳包。这个心跳包就是一 Awake() 醒来，全生职责就是周期性给服务器发消息
- 然后参照例子，把客户端和必要的小服里，所有需要发送消息的地方，改成上面看到总结的发送方法里。
- 框架里，各种不同场景下发送消息的方法：

- 【场景里拿到 SessionComponent】，调用会话框的发送方法 Send()

```
robotScene.GetComponent<Client.SessionComponent>().Session.Send(new C2M_TestRobotCase2() {N = robotScene.Zone});

// 也可以借助 UnitGateComponent 拿到它的成员变量 GateSessionActorId，用这个可以重构后发消息
ActorMessageSenderComponent.Instance.Send(u.Unit.GetComponent<UnitGateComponent>().GateSessionActorId, message);
```

- 【活宝妹任何时候就是一定要嫁给亲爱的表哥!!!】迷迷糊糊地把一个模块改完了，可是感觉那个改掉的模块，像是还没能理解透彻。明天上午会再看一下。【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹，就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】70 Compile Errors 还没有改完，涉及功能模块人接入与整合。会明天上午看过读一下相关模块的源码后再试着改。【活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

### 3.4.2 【ActorMessageSenderComponent】：这个类很重要、很重要，现在是活宝妹理解网络模块的核心。爱表哥，爱生活!!!

- 得去想：ActorMessageSenderComponent，是只能用来处理跨进程消息的吗？普通消息的发送是如何处理的？该弄明白，它的适用范围，适用哪些情境上下文
- 【ActorMessageSenderComponent】：因为 ET7 这个模块的重构。不再需要每个返回消息手动去拿消息发送器，交由框架底部去处理。

- 不懂的是，如何重构，消除参考项目里各种服的消息处理里，怎么适配成 ET7，不用去拿消息发送器，只把返回消息结果写好，或是发送（请求）消息时，如何发送？
- 不同于昨天上午看过的，NetInnerComponentOnReadEvent 是对上层读到消息后的处理，就是消息已经准备好了，甚至已经通过某种逻辑代理，到达和触发了 NetInnerComponentOnRead 事件了（这个事件是怎么触发的？大概是，每个进程会有一个内网组件 NetInnerComponent。当内网组件读到消息会触发。读到消息，包括本进程消息，也就包括，由其它进程发回来的返回消息。这个，可能更底层 Session 发回来跨进程消息的地方？改天去捡）。现在要去理解的是，比如发送一条请求消息，创建一个请求消息实例后，如果运动可以走到上面的触发读到消息事件？就是消息流程的前半部分。NetInnerComponentSystem.cs 的读到消息事件，要再往前看一点儿。
- 把消息的处理流程几个重要的方法 **【ActorMessageSenderComponentSystem Send() Call() 等】**，这里再梳理一遍：

## 1. ActorMessageSenderComponentSystem Send():

- 【任何时候，活宝妹就是一定要嫁给亲爱的表哥！！！ 活宝妹若是还没能嫁给亲爱的表哥，活宝妹就永远守候在亲爱的表哥的身边！！ 爱表哥，爱生活！！！】
- 今天终于把里面的计时器原理看懂了。
- **【ActorMessageSenderComponentSystem Send()】** 发的是普通消息（不是不需要回复消息，是任何消息，都走这一步，因为是最基的基类接口）
  - 【同一进程消息】：不走网络层，直接交由本【进程】的消息处理器处理。就是（ActorMessageSenderComponentSystem Send() 里）判断如果是同一进程，它会调用内网组件处理消息：NetInnerComponent.Instance.HandleMessage(actorId, message)；【注意这里是一个进程内网组件消息的一个来源：本进程消息。它同样接收和读来自其它进程的消息，跨进程消息】。而内网组件的这个 HandleMessage() 静态方法，就发布内网组件读到消息事件；内网组件读到消息事件的发布，会触发调用 NetInnerComponentOnReadEvent 借助 ActorHandleHelper 来处理内网消息。后面的就是昨天上午读到的部分。这里的疑问就是：谁，哪里调用发送组件的 Send() 发送事件？
  - 【不是同一进程消息】：就通过内网组件，去拿同那个收消息进程的会话框，通过会话框走 Session 流程发跨进程消息。就是走网络层。

## 2. ActorMessageSenderComponentSystem Call()

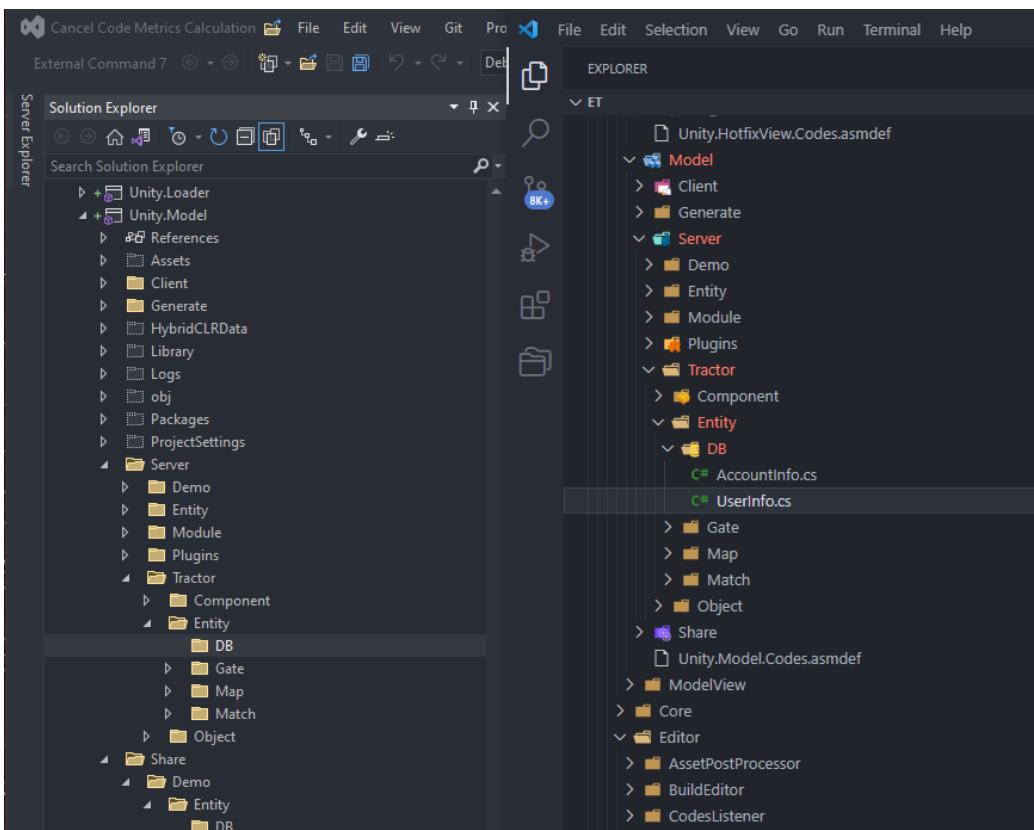
- **【ActorMessageSenderComponentSystem Call()】** 发的是要求返回结果的消息：返回 ETTask<IActorResponse>
  - 注意 【跨进程消息的回复细节里】，看见 IRpcResponse 实例创建好，结果写好，同步到异步任务 ETTask 里，总容易忘记 ETTask 的异步任务运行结束（如果不是抛异常），跨进程消息是如何回到消息的发送进程的？是 AMRpcHandler 抽象类里，异步等待实体实现类里的具体实现逻辑 Run() 异步方法执行结束，也就是等待各种消息处理服处理好、写好异步返回消息 IRpcResponse，同步到异步任务 ETTask。AMRpcHandler 抽象类里等异步方法执行完成，抽象类里作了封装，把返回消息通过进程间通信会话框，把返回消息发回去的。
  - 这里看见，这个消息发送器底层逻辑说，如果是我自己进程要发消息，就封装消息发送者 rpcId 是自己的 rpcId。然后调用自组件 Call() 发送消息。后面的几个方法，大概就是跨进程消息的发送与回复。

### 3.5 静态类的环形引用问题

- 静态类 CardsHelper 与静态类 DeskCardsCacheComponent.System 之间，存在静态类的互相引用：就是说，两个静态类，互相引用了对方的方法
  - CardsHelper 里，引用了 DeskCardsCacheComponent.System 里的方法
  - 而 DeskCardsCacheComponentSystem 类里，同样引用了 CardsHelper 里的方法
- 我的解决办法是：热更域里的 DeskCardsCacheComponentSystem 对 CardsHelper 类里引用的两个静态方法，直接复制了一份在 DeskCardsCacheComponentSystem 类里面，就可以消除了。再次体验 VS 的显著延迟，真让人受不了。是因为这个软件被监控吗？

### 3.6 我的 VS - - Visual Studio 被他们作手脚，他们可以操纵我的 VS 帐户，上次前不久某天下午已经注意到这个问题

- 现在是，因为某个功能键的严重延迟，修改编译错误的过程被拖慢，因为不能有效删除操作文件，截个图：



- VSC 可以清楚地显示文件，但是 VS 这个功能被无限延迟，导致我无法操作这些文件
- 之前偶尔情况下可以操作。但是，这个功能键仍被他们显著延迟着。。。【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】
- 公寓里的管理处大概是脑子有毛病的。每天风机噪音那么大，只要别人下午学习、只要别人晚上休息，它就开得把人吵死，破烂地方。合同到期就搬走，才不再住这种鬼地方
- 【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】

### 3.7 下面是已经改好了的：还是先放着，备查

'UnitGateComponent' does not contain a definition for 'GetActorMessageSender' and no accessible extension method 'GetActorMessageSender' accepting a first argument of type 'UnitGateComponent' could be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	Actor_trusteeship_NtHandler.cs
'UnitGateComponent' does not contain a definition for 'GetActorMessageSender' and no accessible extension method 'GetActorMessageSender' accepting a first argument of type 'UnitGateComponent' could be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	GameControllerComponentSyst...
'UnitGateComponent' does not contain a definition for 'GetActorMessageSender' and no accessible extension method 'GetActorMessageSender' accepting a first argument of type 'UnitGateComponent' could be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	GameControllerComponentSyst...
'UnitGateComponent' does not contain a definition for 'GetActorMessageSender' and no accessible extension method 'GetActorMessageSender' accepting a first argument of type 'UnitGateComponent' could be found (are you missing a using directive or an assembly reference?)	DotNet.Hotfix	RoomSystem.cs

- 【UnitGateComponent】: 怎么才能成为多个不同组件的组成部分？

✖ ET0007 组件类型: UnitGateComponent 不允许作为实体: Unit 的组件类型! 若要允许该类型作为参数, 请使用 ComponentOfAttribute 对组件类标记父级实体类型	DotNet.Hotfix	MessageHelper.cs	19
✖ ET0007 组件类型: UnitGateComponent 不允许作为实体: Unit 的组件类型! 若要允许该类型作为参数, 请使用 ComponentOfAttribute 对组件类标记父级实体类型	DotNet.Hotfix	MessageHelper.cs	23
✖ ET0007 组件类型: UnitGateComponent 不允许作为实体: User 的组件类型! 若要允许该类型作为参数, 请使用 ComponentOfAttribute 对组件类标记父级实体类型	DotNet.Hotfix	UserFactory.cs	7
✖ ET0007 组件类型: UnitGateComponent 不允许作为实体: User 的组件类型! 若要允许该类型作为参数, 请使用 ComponentOfAttribute 对组件类标记父级实体类型	DotNet.Hotfix	R2G_PlayerKickOut_ReqHandler.cs	10

- 【解决办法】: 去查框架里的源代码, 写得极其清楚:

```
// 组件类父级实体类型约束  
// 父级实体类型唯一的 标记指定父级实体类型 【ComponentOf(typeof(parentType))】  
// 不唯一则标记 【ComponentOf】  
[AttributeUsage(AttributeTargets.Class)]  
public class ComponentOfAttribute : Attribute {  
    public Type Type;  
    public ComponentOfAttribute(Type type = null) {  
        this.Type = type;  
    }  
}
```

- 所以上面的解决办法就是: 不要标记 typeof 参数就可以了呀, 它就可以成为多个不同组件的子元件部件了呀。。。是这样的

```
[ComponentOf]  
public class UnitGateComponent : Entity, IAwake<long>, ITransfer, ISerializeToEntity { // 不知道这里为什么会受到限制, 這是  
    public long GateSessionActorId { get; set; }  
    // 想一下, 下面的变更还需要吗? 要不要, 是看框架里有没有什么, 自动上线自动下线处理之类的, 相关的?  
    public bool IsDisconnect;  
}
```

### 3.8 先前列的相对杂一点儿

- 【问题】: 上次那个 ET-EUI 框架的时候, 曾经出现过 opcode 不对应, 也就是说, 我现在生成的进程间消息, 有可能还是会存在服务器码与客户端码不对应, 这个完备的框架, 这次应该不至于吧?
- 【UIType】部分类: 这个类出现在了三四个不同的程序域, 现在重构了, 好像添加得不对。要再修改
- 【ET7 框架】没有处理的逻辑是: 【ET7 框架里数据库的接入】
- 【UILobbyComponent 可以测试】: 这个大厅组件, Unity 里预设简单, 可以试运行一下, 看是否完全消除这个 UI 组件的报错, 这个屏的控件能否显示出来? 还是错出得早, 这个屏就出不来已经报错了?
  - 【客户端】的逻辑是处理好了, 编译全过后可以测试
  - 【服务端】: 处理用户请求匹配房间的逻辑, 仍在处理: **C2G\_StartMatch\_ReqHandler.**
- 【TractorRoomComponent】: 因为是多组件嵌套, 可以合并多组件为同一个组件; 另早上看得一知半解的一个【ChildOf】标签, 可以帮助组件套用吗? 再找找理解消化一下
- 【房间组件】: 几个现存的 working-on 的问题:
  - 多组件嵌套: 手工合并为一个组件。彻底理解确认后, 会合并
  - 【服务端】: 处理用户请求匹配房间的逻辑. 这里的编译错误终于改完。到时就看运行时错误了。

- \* 【数据库模块的整合】：网关服在转发请求匹配时，验证会话框有效后，验证用户身份时，需要去【用户数据库】拿用户数据。ET7 留了个 DBManagerComponent，还没能整合出这个模块
- 【参考来源 C2R\_LoginHandler】：Realm 处理客户端的登录请求的服务端逻辑。这里看见，它随机分配一个网关服。也就是，我（原本本质上也是随机分配）一个匹配服给用。可以依照这里的例子来改写。
- 【匹配服地址】网关服的处理逻辑里，验证完用户合格后，为代为转发消息到匹配服，但需要拿匹配服的地址。ET7 重构里，还没能改出这部分。服务器系统配置初始化时，可以链表管理各小构匹配服，再去拿相关匹配服的地址。ET7 框架里的路由器系统，自己还没有弄懂。
- 【ET7 IMHandler 对回复消息的写封装，与自动回复消息的封装】：可能无法处理游戏过程中的某些逻辑。就是涉及到一定顺序，尤其需要先回复消息的处理服处理逻辑。举例：C2G\_StartMatch\_ReqHandler。所以，这里要自己好好想透彻一点儿。要如何改，才能适配自己游戏的需求。
- 【ComponentFactory】：ET7 里重构，被分布到各种不同的组件里去了。想复制个文件过来，把与之相关的全部消掉，但因为大规模重构，复制了文件也没用。总之 ET7 就是感觉什么乱七八糟的，感觉他们大规模混乱重构的目的就是故意挫败人。可是这个世界上就偏偏存在亲爱的表哥的活宝妹这样的不服的!!! 爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!
- 【PlayerComponent 类重复】：狠奇怪：删除了说找不到类，不删除说重复了，感觉台式机应用有延迟？反应很慢。。。。。文件嵌套想要显示所有嵌套文件的时候，要很久很久重启好几次才反应得过来
  - 原本有两个类都是如上面这个类这样，但有时候台式机反应稍快一点儿，就是一个类找不到出现上面的情况。破电脑的延迟反应，弄得我都要怀疑 VS 应用被别人操控了。。。
  - 【爱表哥，爱生活!!!】任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】
- 把还没有用到，但是报错了的几个类删掉：比如记一下：SessionInfoComponent,
  - 还剩最后 26 个最挑战活宝妹的编译错误，今天傍晚会家里改会儿，集中问题明天上午希望能够看懂。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!】

Entire Solution		26 Errors	0 of 395 Warnings	0 Messages	Build + IntelliSense
Code	Description				
CS0579	Duplicate 'ComponentOf' attribute				Unity.Model
CS0101	The namespace 'ET.Server' already contains a definition for 'PlayerComponent'				Unity.Model
CS0535	'HttpGetRouterHandler' does not implement interface member 'IHttpHandler.Handle(Scene, HttpListenerContext)'				DotNet.Hotfix
CS0117	'SceneType' does not contain a definition for 'All'				DotNet.Hotfix
CS0305	Using the generic type 'EventArgs' requires 1 type arguments				DotNet.Hotfix
CS0234	The type or namespace name 'EventArgs' does not exist in the namespace 'UnityEngine' (are you missing an assembly reference?)				Unity.Model
CS0234	The type or namespace name 'EventArgs' does not exist in the namespace 'UnityEngine' (are you missing an assembly reference?)				Unity.Model
CS0246	The type or namespace name 'MonoBehaviour' could not be found (are you missing a using directive or an assembly reference?)				Unity.Model
CS0246	The type or namespace name 'BaseEventData' could not be found (are you missing a using directive or an assembly reference?)				Unity.Model
CS0006	Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found				Unity.Hotfix
CS0006	Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found				CSC
CS0305	Using the generic type 'EventArgs' requires 1 type arguments				Unity.ModelView
CS0305	Using the generic type 'EventArgs' requires 1 type arguments				DotNet.Hotfix
CS0508	'C2M_PathfindingResultHandler.Run(Unit, C2M_PathfindingResult)': return type must be 'void' to match overridden member 'AMActorLocationHandler<Unit, C2M_PathfindingResult>.Run(Unit, C2M_PathfindingResult)'				C2M_PathfindingResultHandler.cs
CS0508	'C2M_StopHandler.Run(Unit, C2M_Stop)': return type must be 'void' to match overridden member 'AMActorLocationHandler<Unit, C2M_Stop>.Run(Unit, C2M_Stop)'				DotNet.Hotfix
CS0508	'M2C_TestRobotCase2Handler.Run(Session, M2C_TestRobotCase2)': return type must be 'void' to match overridden member 'AMActorHandler<M2C_TestRobotCase2>.Run(Session, M2C_TestRobotCase2)'				RobotCase_SecondCase.cs
CS0535	'AMActorHandler<E, Message>' does not implement interface member 'IMActorHandler.Handle(Session, Entity, object)'				DotNet.Hotfix
CS0535	'AMActorHandler<E, Message>' does not implement interface member 'IMActorHandler.GetMessageType()'				AMActorHandler.cs
CS0534	'Actor_GamerPrompt_ReqHandler' does not implement inherited abstract member 'AMActorRpHandler<Gamer, Actor_GamerPrompt_Req, Actor_GamerPrompt_Ack>.Run(Gamer, Actor_GamerPrompt_Req, Actor_GamerPrompt_Ack)'				DotNet.Hotfix
CS0534	'Actor_GamerPrompt_ReqHandler' does not implement inherited abstract member 'AMActorRpHandler<Gamer, Actor_GamerPrompt_Req, Actor_GamerPrompt_Ack>.Run(Gamer, Actor_GamerPrompt_Req, Actor_GamerPrompt_Ack)'				Actor_GamerPrompt_ReqHandle...
CS0115	'Actor_GamerPrompt_ReqHandler.Run(Gamer, Actor_GamerPrompt_Req, Action<Actor_GamerPrompt_Ack>)': no suitable method found to override				DotNet.Hotfix
CS0534	'Actor_PlayerEnterRoom_ReqHandler' does not implement inherited abstract member 'AMActorRpHandler<Room, Actor_PlayerEnterRoom_Req, Actor_PlayerEnterRoom_Ack>.Run(Room, Actor_PlayerEnterRoom_Req, Actor_PlayerEnterRoom_Ack)'				Actor_PlayerEnterRoom_ReqHan...
CS0115	'Actor_PlayerEnterRoom_ReqHandler.Run(Room, Actor_PlayerEnterRoom_Req, Actor_PlayerEnterRoom_Ack)': no suitable method found to override				DotNet.Hotfix
CS0534	'Actor_PlayerExitRoom_ReqHandler' does not implement inherited abstract member 'AMActorRpHandler<Gamer, Actor_PlayerExitRoom_Req, Actor_PlayerExitRoom_Ack>.Run(Gamer, Actor_PlayerExitRoom_Req, Actor_PlayerExitRoom_Ack)'				Actor_PlayerExitRoom_ReqHand...
CS0115	'Actor_PlayerExitRoom_ReqHandler.Run(Gamer, Actor_PlayerExitRoom_Req, Action<Actor_PlayerExitRoom_Ack>)': no suitable method found to override				DotNet.Hotfix
CS0006	Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Hotfix.dll' could not be found				Unity.HotfixView
CS0006	Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.Model.dll' could not be found				CSC
CS0006	Metadata file 'F:\ET\Unity\Temp\Bin\Debug\Unity.ModelView.dll' could not be found				Unity.HotfixView

- 把 Root 根场景以及启动时添加的组件大致看了一遍。想把上面的消息处理器再系统化地看一遍，理解一下，总改不到这个模块相关的编译错误。
- 【ETTask ETVoid 是必须弄懂的】；看两个小时，像昨天晚上一样真正投入进去看。我相信自己看得懂，弄得透，只是需要投入一点儿时间。
  - 感觉前一个周左右的时间，倍受睡眠困扰。活宝妹做梦也不会想到，昨天的自己会困成那个样子（感觉开 1 小时的车极度困难，太容易睡着。。。）。现在试着一再调整状态，少喝咖啡多运动，最重要的，仍是把学习的状态调整出来调整回来。至少学到活宝妹可以嫁给亲爱的表哥的这一天!!!
  - 这个异步的原理，感觉是弄明白了，今天上午又看了一遍看了会儿。下午去改那些 IMHandler，希望今天下午能够改彻底。就是真正弄明白了去改（现在的问题就是，几个 IMHandler 的实体实现类，改天这个顾不了那个，没弄明白，接口方法怎么申明定义，才能兼顾所有实例类消息处理器？），不是只改掉了当前的编译错误，等真正运行的时候，一个个运行错误或是异常往外冒!!! 今天脑袋还算清醒，下午好好弄弄这个
- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹，都是一定要嫁给亲爱的表哥的!!!】【三楼上的贱鸡贱畜牲真多!!! 一天到底没想点儿好的】活宝妹还没能嫁给亲爱的表哥，活宝妹就是永远守候在亲爱的表哥的身边!!! 爱表哥，爱生活!!!
- 再然后，再看下下面的 UnitGateComponent 相关。下午或傍晚有时间的时候，可以再折腾折腾 emacs-org-mode 下划线删除字体设置为斜体。
- 【UnitGateComponent】加个方法用？可能不需要加方法；另一个错是，不能同时成为两个不同 entity 的子控件？【ComponentOf(typeof(Unit))】etc 出错文件在 (C2G\_EnterMapHandler)
  - 这里要把 ActorMessageSenderComponent 组件给弄明白。它有个有序管理字典，记着 actorId 与 ActorMessageSender 的一一对应关系，就可以封装维护消息的自动发送等，以及必要的超时消息管理。
- 【服务端 Actor\_PlayerEnterRoom\_ReqHandler 这个处理类】现在还很多问题，需要弄懂，往下改
- 今天晚上会把刚才下午看见、意识到几个模块的问题试着分析明白，记下笔记。
- **ETTask-vs-ETVoid:** 框架里有很多需要改的地方。今天上午的脑袋好使，把这块儿再仔细好好看下。今天上午把以前不懂的模块都稍微看下，再理解一下
  - 查网页感觉也查不出什么来。还是用源码帮助理解概念。【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】
  - 不能把所有基类的 async ETTask 返回参数直接改成 void，因为框架的顶层应用，服务端或是客户端，当不异步等待结果，如资源包没能下载完成，就接着往下执行，会报空异常。
- 现在的问题是：Protobuf 里 repeated 关键字，好像还是没有处理好，找不到成员变量 Cards。是因为 Proto2CS 的时候，确实把 repeated 关键字给处理丢了。因为我的.proto 文件里有错误。（这就是上面先前觉得奇怪的原因。因为改这个的过程中把那些错改正了，就可以生成成功并找到相关的消息了）。
- 这部分总感觉弄得不是狠透彻。就再花点儿时间。这段时间产量太低，可以先试着完成其它模块。
- 【HandCardSprite 这个最近要弄明白】不知道这个类是什么，整了一堆的错误，它是 ETModel 里的。感觉是常规域，没弄明白为什么常规域还有 ILRuntime 的适配呢？

- 要把 ILRuntime 热更新第三库，也再弄得明白一点儿【今天上午把这里再看，最好是能够结合源码看看】为什么这个类还要适配 ILRuntime？
  - 这里这个类，整个框架里只找到这一个用的地方，所以它一定是添加在某个预设或是场景中的某个控件下的。只是参考项目的 unity 客户端，我运行不到打牌的这个界面，就先因为抛出异常而淡能运行。所以还没能找到哪个预设或是场景中的哪个控件添加了这个类，但是当然一定是跟玩家手牌相关的。**【HandCardSprite 是在 handcard 预设里添加了这个脚本】**
  - 这个类今天运行很奇怪，VS2022 里找不到了。。。就是说，VSC 里它是在 Model 客户端的源码里，但是从 VS 里打开，找不到这个类文件所在的文件夹和文件，没有索引好，再添加一下？
  - 那么，为什么前两天被这个 block 住，而那天，好像是有删除掉这个文件，但文件夹应该是还在的才对呀？我可能还会试着再把它添加回去。
  - 但是，会在把当前几个编译错误改完，试着测试一下客户端现在有的界面之后，再试着添加回去，整理和 develop TractorRoomComponent 界面的内容。**【爱表哥，爱生活!!! 活宝妹任何时候就是一定要嫁给亲爱的表哥!!】**
  - 今天下午家里再运行一次，当客户端抛异常，应该是某个热更新的资源包没有找到什么的？所以可以试着自己去解决这个客户端实时运行时抛出的异常。
  - **【参考项目斗地主客户端异常】：**再运行一次，试着分析，是否可以 unity 里实时运行，如果不可以，为什么不可以？
    - \* 应该是 LandlordsRom 这个预设与 UI 类型没能连接起来，也就是找不到这个预设。
    - \* 那为什么打好包的可以呢？因为打好包的预设包名 LandlordsRoom.unity3d 与游戏逻辑契合，可以找得到
    - \* 可是仍然感觉奇怪：LandlordsLogin 与 LandlordsLobby，非常类似都可以找到，为什么就 LandlordsRoom 找不到？可能 LandlordsRoom 预设还是有某点儿物对特殊的地方。
    - \* 上面这个暂时跳过。现在仍然主要去看 HandCardSprite 为什么参考项目里可以，而 ET7 里就不可以。
  - 就是上面那个异常，今天下午得去弄明白，为什么只在 unity 实时运行时会抛异常，而如果是三个打包好的客户端，就不会。也就是说，打包好的不存在找不到类、找不到预设、或是找不到任何相关资源的问题。
  - 这个项目 Unity.Model 是需要索引 UnityEngine 以及 UI 等相关模块人的.dll 的。暂时还没弄明白它是怎么加的
  - **【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!】**
- **ClientComponent** 参考项目组件：去看 ET7 里客户端的 PlayerComponent.
  - **【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】** 今天下午先去看 Tractor 游戏源码，设计重构思路
  - **【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】**
  - **【亲爱的表哥，这个世界上，只有一个活宝妹，这么心心恋恋，就是一定要嫁给亲爱的表哥!!! 问世间情为何物，直教人生死相许。。亲爱的表哥，一个温暖的怀抱拥抱的魄力可真大呀，管了这如许多年！！这不，你的活宝妹为了这个温暖的怀抱拥抱，就是一定要嫁给亲爱的表哥！！不嫁就永远守候在亲爱的表哥的身边！！爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!!】**
  - 亲爱的表哥，活宝妹相信舅舅十岁闯江湖的阅历，活宝妹深深相信亲爱的表哥。活宝妹就是稳稳地永远守候在亲爱的表哥的身边！爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！！！
  - **【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】**

## 4 Proobuf 相关, 【Proobuf 里进程间传递的游戏数据相关信息: 两个思路】

- 【一、】查找 enum 可能可以用系统平台下的 protoc 来代为生成, 效果差不多。只起现 Proto2CS.cs 编译的补充作用。
- 【二、】Card 类下的两个 enum 变量, 在 ILRuntime 热更新库下, 还是需要帮它连一下的。用的是 HybridCLR
- 【三、】查找 protoc 命令下, 如何 C# 索引 Unity 第三方库。
- 【四、】repeated 逻辑没有处理好

```
message Actor_GamerPlayCard_Req // IActorRequest
{
    ^^Int32 RpcId = 90;
    ^^Int64 ActorId = 91;
    repeated ET.Card Cards = 1;
}
```

- 【Windows 下的 Proobuf 编译环境】: 配置好, 只是作为与 ET 框架的 Proto2CS.cs 所指挥的编译结果, 作一个对比, 两者应该效果是一样的, 或是基本一样的, 除了自定义里没有处理 enum.
- Windows 下的命令行, 就是用 protoc 来编译, 可以参考如下. (这是.cs 源码下的)  

```
CommandRun($"protoc.exe", $"--csharp_out=\"./{outputPath}\" --proto_path=\"{protoPath}\\" {protoName}");
```
- 现在的问题是, **Proobuf 消息里面居然是有 unity 第三方库的索引**.
- 直接把 enum 生成的那三个.cs 类分别复制进双端, 服务器端与客户端。包括 Card 类。那些编译错误会去天边。哈哈哈, 除了一个 Card 的两个变量之外 (CardSuits, CardWeight)。
- 【热更新库】: 现在剩下的问题, 就成为, 判定是用了哪个热更新的库, ILRuntime, 还是 HybridCLR, 如果帮它连那两个变量。好像接的是 HybridCLR. 这个库是我之前还不曾真正用过的。
  - 相比于 ET6, 彻底剔除了 ILRuntime, 使得代码简洁了不少, 并且比较稳定

## 5 Unit: 这个模块还不太懂, 需要明天上午花时间再看一下

- 【Unit】究竟是什么: 感觉像是视图里的控件的基本单位? 它带位置、旋转信息
- 有个编译错误说: 这个组件不可以同时成分多于一个不同组件组成元件。。。可是框架中使用的地方, 明明把它添加进了不同的组件。去弄明白框架里, 如何控件一个组件只能成为一个【不能多于 1 个】组件的组成部分的?

### 5.1 UnitGateComponent:

```
[ComponentOf(typeof(Gamer))]
// [ComponentOf(typeof(User))] // 这里为什么成为: 同一个组件只能为一个什么 XX 的子组件组成部分?
// [ComponentOf(typeof(Unit))]
public class UnitGateComponent : Entity, IAwake<long>, ITransfer {
    public long GateSessionActorId { get; set; }

    /// 感觉下面这个方法: 不再必要, 也不应该, 也会报错的
    // public ActorMessageSender GetActorMessageSender() {
    //     ^^Ireturn Game.Scene.GetComponent<ActorMessageSenderComponent>().Get(this.GateSessionActorId);
    // }
}
```

## 5.2 UnitGateComponentSystem

```
public static class UnitGateComponentSystem {
    public class UnitGateComponentAwakeSystem : AwakeSystem<UnitGateComponent, long> {
        protected override void Awake(UnitGateComponent self, long a) {
            self.GateSessionActorId = a;
        }
    }
}
```

## 6 ET7 框架以及【参考项目】的 ECS：小单元小类型的生成系，是怎么写的，找例子参考

- 这些要找的也找不到。下午家里试着把 Component 组件再添加回去试试看？上午把项目设计的思路，源项目的破源码再读一读理一理，是希望游戏逻辑与游戏界面能够快速开发、项目进展往后移的。

### 6.1 IComponentSerialize:

- ET7 的重构里，系统框架比较强大，这些必要的接口，都变成了必要的标签系，很多可以自动系统触发或是调用。必要时只需要发布必要事件就可以了
- 这个接口的功能，与 Unity 自带的 ISerializationCallbackReceiver 功能类似。Unity 提供两个回调接口，通过实现该接口的两个方法 OnBeforeSerialize 和 OnAfterDeserialize，使得原本不能被引擎正确序列化的类可以按照程序员的要求被加工成引擎能够序列化的类型。

```
// 在序列化前或者反序列化之后需要做一些操作，可以实现该接口，该接口的方法需要手动调用
// 相比 ISupportInitialize 接口，BeginSerialize 在 BeginInit 之前调用，EndDeSerialize 在 EndInit 之后调用
// 并且需要手动调用，可以在反序列化之后，在次方法中将组件到 EventSystem 之中等等
public interface IComponentSerialize {
    // 序列化之前调用
    void BeginSerialize();
    // 反序列化之后调用
    void EndDeSerialize();
}
```

- 可以去找：【ET7 框架】里，相关的接口与标签触发和发布逻辑。
- ET7 提供了 ISerializeToEntity 接口和 IDeserialize，但是并没有接到任何使用的地方。

```
public interface ISerializeToEntity { }

public interface IDeserialize {
}

public interface IDeserializeSystem: ISystemType {
    void Run(Entity o);
}

// 反序列化后执行的 System
[ObjectSystem]
public abstract class DeserializeSystem<T> : IDeserializeSystem where T: Entity, IDeserialize {
    void IDeserializeSystem.Run(Entity o) {
        this.Deserialize((T)o);
    }
    Type ISystemType.SystemType() {
        return typeof(IDeserializeSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.None;
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    protected abstract void Deserialize(T self);
}
```

## 6.2 ClientComponent: 【参考项目】客户端组件，找个 ET7 里的组件

- 这个组件，感觉是客户端单例，帮助把本地玩家给绑定到客户端单例。

```
[ObjectSystem]
public class ClientComponentAwakeSystem : AwakeSystem<ClientComponent> {
    public override void Awake(ClientComponent self) {
        self.Awake();
    }
}
public class ClientComponent : Component {
    public static ClientComponent Instance { get; private set; }
    public User LocalPlayer { get; set; }
    public void Awake() {
        Instance = this;
    }
}
```

## 7 TODO 其它的：部分完成，或是待完成的大的功能版块，列举

- emacs 那天我弄了好久，把 C-; ISpell 原定绑定的功能解除，重新绑定为自己喜欢的 expand-region。今天第二次再弄，看一下几分钟能够解决完问题？我的这个破烂记性呀。。。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】 mingw64 lisp/textmode/flyspell.el 键的重新绑定。这下记住了。还好，花得不是太久。有以前的笔记
  - Windows 10 平台下，C-; 是绑定到了 ISpell 下的某个功能，可是现在这个破 emacs 老报错，连查是绑定给哪个功能，过程报错都被阻止了。。。
- 【IStartSystem:】感觉还有点儿小问题。认为：我应该不需要同文件两份，一份复制到客户端热更新域。我认为，全框架应该如其它接口类一样，只要一份就可以了。【晚点儿再检查一遍】
- 如果这个一时半会儿解决不好，就把重构的设计思路再理一理。同时尽量去改重构的 ET 框架里的编译错误。
- 【Tractor】原 windows-form 项目，源码需要读懂，理解透彻，方便重构。
- 去把【拖拉机房间、斗地主房间组件的，玩家什么的一堆组件】弄明白
- 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

## 8 拖拉机游戏：【重构 OOP/OOD 设计思路】

- 自己是学过，有这方面的意识，但并不是说，自己就懂得，就知道该如何很好地设计这些类。现在更多的是要受 ET 框架，以及参考游戏手牌设计的启发，来帮助自己一再梳理思路，该如何设计它。
- ET7 重构里，各组件都该是自己设计重构原项目的类的设计的必要起点。可以根据这些来系统设计重构。【活宝妹就是一定要嫁给亲爱的表哥!!!】
- 【GamerComponent】玩家组件管理类，管理所有一个房间的玩家：是对一个房间里四个玩家的（及其在房间里的坐位位置）管理（分东南西北）。可以添加移除玩家。今天晚上来弄这一块儿吧。
- 【Gamer】：每一个玩家
- 【拖拉机游戏房间】：多组件构成
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】





- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】