

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

July 1, 2023

Contents

1 C# 异步基础原理、状态机原理、逻辑整理	1
1.1 ETVoid C# Net-async await 编程更底层一点儿的原理	1
1.2 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:	7
2 Protobuf 相关，【Protobuf 里进程间传递的游戏数据相关信息：两个思路】	8
3 Unit: 这个模块还不太懂，需要明天上午花时间再看一下	8
3.1 UnitGateComponent:	8
3.2 UnitGateComponentSystem	9
4 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	9
4.1 IComponentSerialize:	9
4.2 ClientComponent: 【参考项目】客户端组件，找个 ET7 里的组件	10
5 TODO 其它的：部分完成，或是待完成大的功能版块，列举	10
6 拖拉机游戏：【重构 OOP/OOD 设计思路】	10
7 【亲爱的表哥的活宝妹就是一定要嫁给亲爱的表哥!!!】	11
8 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	20
8.1 热更新层的生成系	20
8.2 【ComponentFactory】这个现在算是编译错误基本消除完了	21
8.3 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	23
8.3.1 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。	23
8.3.2 【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!!!	24
8.4 静态类的环形引用问题	25
8.5 下面是已经改好了的：还是先放着，备查	26
8.6 先前列的相对杂一点儿	26

8.7 LocationComponent: 【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!!!】	30
8.8 【数据库模块】: 这个模块的编译错误，昨天下午清理完了	30
8.9 【HandCardSprite.cs】	30
8.10 IPPEndPoint-to-long 转化	31
8.11 两件杂事:	31
8.12 StartConfigComponent: 现框架里有重构了的版本，在理解现 ET7 框架的基础上进行必要适配：先把之前总结的再熟悉一下，下午有时间也看看这个	31
9 Net 网络交互相关:【服务端 + 客户端】只是稍微改装成事件机制。没理解透、总结不全，待增补	33
9.1 RpcInfo: 【消息的包装体】。结合 NetServerComponentOnReadEvent 来读。	33
9.2 NetServerComponent: NetServerComponentOnRead 结构体。	33
9.3 NetServerComponentSystem: 【服务端】组件：网络交互服务端的相关功能，事件发布等	33
9.4 NetServerComponentOnReadEvent: NetServerComponent 组件，会发布事件，触发此回调类	34
9.5 NetClientComponent: 【网络客户端】组件：这个，感觉与【服务端】定义申明上看是一样的	35
9.6 NetClientComponentSystem: 【服务端】也是类似事件系统的改装	35
9.7 NetClientComponentOnReadEvent: 框架里只有这一个注册过的回调事件	36
9.8 NetInnerComponent: 【服务端】对不同进程的处理组件。是服务器的组件	36
9.9 NetInnerComponentSystem: 生成系	37
9.10 MessageDispatcherInfo: 在【MessageDispatcherComponent】中	38
9.11 MessageDispatcherComponent: 全局全框架单例：【活宝妹就是一定要嫁给亲爱的表哥！！！爱表哥，爱生活!!!】	38
9.12 MessageDispatcherComponentSystem:	39
9.13 MessageDispatcherComponentHelper:	39
9.14 SessionIdleCheckerComponent: 【会话框】闲置状态管理组件	40
9.15 SessionIdleCheckerComponentSystem: SessionIdleChecker 激活类,	40
9.16 MessageHelper: 不知道这个类是做什么用的，使用场景等。过会儿看下	41
9.17 ActorHandleHelper: 是谁调用它，什么场景下使用的？这个，今天下午再补吧	42

1 C# 异步基础原理、状态机原理、逻辑整理

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

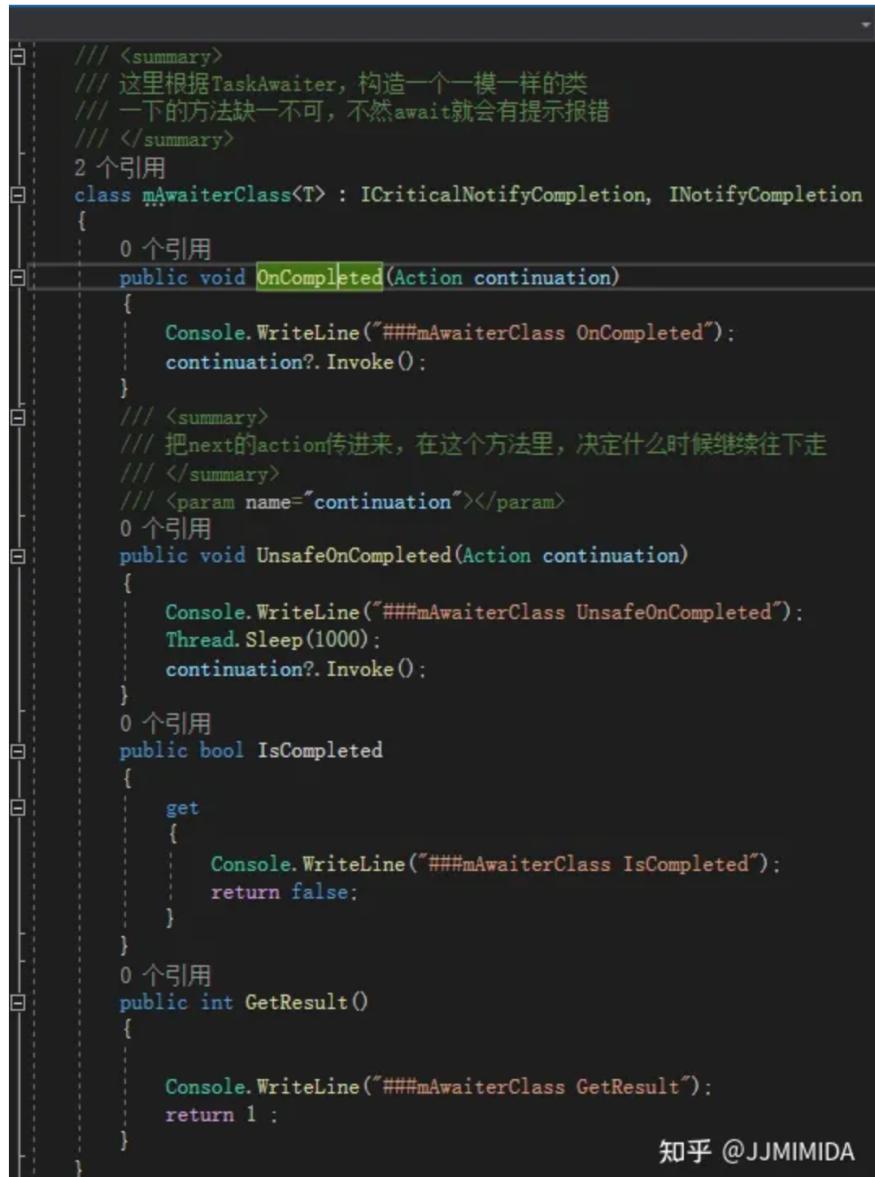
- 就是不懂底层的原理是什么，方法定义是什么，返回的是什么，在有 await 等关键字的时候，返回的内容等是如何变换的，以及它背后的那个异步状态机，就是想不明白。
- 现在参考网上的一个例子，记一下异步任务 C# 幕后封装的那些执行步骤什么的，把 async await 之类的关键字，背后的逻辑理解明白。
- 就是，可能也可以在异步任务的这个模块，添加无数的日志，通过读日志来把这块儿弄明白。下面就截图网上的这个参考例子。

```
[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_004c: 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。

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

obj instance void AyncTest.Program/'<asyncMethod>d_1'::ctor()
    .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!!!】

1.2 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!!!

2 Protobuf 相关, 【Protobuf 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 Protobuf 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，**Protobuf 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

3 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？

3.1 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);
    // }
}
```

3.2 UnitGateComponentSystem

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

4 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。

4.1 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);
}
```

4.2 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;
    }
}
```

5 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!!!】**

6 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

7 【亲爱的表哥的活宝妹就是一定要嫁给亲爱的表哥!!!】

8 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

- 永远把每天更新最多的，放在文件的最后最简单，emacs 操作也会方便一点儿

8.1 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。运行时如果报错，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。【这片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!!!】

8.2 【ComponentFactory:】这个现在算是编译错误基本消除完了

```
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 'BaseEventArgs' 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 '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 '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	6	
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	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	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	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
- 【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）\
 - 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？

8.3 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，再来看一看 ET7 框架我要怎么改它现存封装，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）

8.3.1 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 0
- 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_NthHandler.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!!!】

8.3.2 【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？改天去检）。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 ETTTask<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 ETTTask 里，总容易忘记 ETTTask 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 ETTTask. 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。

8.4 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？

8.5 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_NttHandler.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？

⑥ 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
⑥ 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
⑥ 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
⑥ 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;  
}
```

8.6 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，感觉他们大规模糊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, Action<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!!!】**

8.7 LocationComponent: 【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!!!】

- 【今天上午】: 从这里开始，把先前总结 Actor 消息以及处理器时，所有关于位置的消息，以及相关的消息处理器弄懂。【没看完】
 - 先前，消息处理器的部分，只看了一个接口类和两个抽象实现，其它没看
 - 消息，位置消息相关的内容，还没看不懂。
- 【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!!!】任何时候，活宝妹还没能嫁给亲爱的表哥，他们就大可不必发疯犯贱。任何时候，他们发疯犯贱，他们也永远只能是发疯犯贱得了一时，发疯犯贱不了一世。亲爱的表哥的活宝妹，若是还没能嫁给亲爱的表哥，亲爱的表哥的活宝妹，就是永远守候在亲爱的表哥的身边!! 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
- 这个模块的编译错误，被活宝妹全部给消除掉了。。。

8.8 【数据库模块:】: 这个模块的编译错误，昨天下午清理完了

- DBProxyComponent: 这个类被重构丢了。数据库分区管理。根据用户所在的区号，去拿该区数据库索引办事就可以了。
- DBManagerComponent: 全框架找不到一个使用的样例。我认为数据库应该只属于服务端。所以，我先把它在应用启动时添加到服务端的公用组件启动程序中 (EntryEvent2_InitServer) 去。
- 下午把几个 DBProxyComponent 相关的编译错误，基本改光了（目前还有几个小模块共计 28 个编译错误）。还有一个类里不知道怎么用 Gamer 去拿玩家所在的小区，先放一下，改天再去改个。
- 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

8.9 【HandCardSprite.cs】

- 【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 HandCardSprite.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】:

```
public static class Game {
    private static Scene scene;
    public static Scene Scene {
        get {
            if (scene != null)
                return scene;
            scene = new Scene();
```

```

        return scene;
    }
}

private static EventSystem eventSystem; // <<<<<<<<<<<<<<<<<<<<
public static EventSystem EventSystem {
    get {
        return eventSystem ?? (eventSystem = new EventSystem());
    }
}

private static ObjectPool objectPool; // <<<<<<<<<<<<<<<<<<<<
public static ObjectPool ObjectPool {
    get {
        return objectPool ?? (objectPool = new ObjectPool());
    }
}

public static void Close() {
    scene.Dispose();
    scene = null;
    eventSystem = null;
    objectPool = null;
}
}
}

```

- 现框架里不存在的，需要整合进来的模块版块：DBProxyComponent, InnerConfig, Location-Component, 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？

8.10 IPPEndPoint-to-long 转化

8.11 两件杂事：

- 【先花 10 分钟左右，搜下看 emacs-export-to-pdf 与 Skim 应用的自动同步，能理解回调适配过程吗？】还是比较麻烦，改天傍晚或是晚上再凭兴趣来解决，早上看别的
 - 这里主要的问题时，Skim 实时更新外源 pdf 的更新时，因为 emacs 的 export-to-pdf 有个过程，这个过程中生成 table-of-contents 比较靠后，如果不背便条，就必须得每次重点查看 TOC。Skim 怎么才能够接收到 latex 生成 TOC 完成后的回调，来从 Skim 中显示 TOC？
 - 这里要去想，为什么背个便条，就能最终自动显示 TOC 了呢？背便条能够最终显示 TOC 是为什么，**背便条背后的原理，能否借用？**
 - 另外，【可以考虑，过滤掉或是配置掉背便条自动同步过程中的确认窗口繁琐过程】如果每个 pdf 的自动生成与刷新，我不需要多次点 enter 确认窗口，我只需要点击一个，或者甚至一个也不需要点，就也算满足用户需求。
- VSC 的配置可能哪里写得不对。以前没有这个问题。现 VSC 跳转至 emacs 打开当前 buffer，不能精确定位到 VSC 中当前 buffer 所在的行。改天有机会的时候再 debug 一下。

8.12 StartConfigComponent: 现框架里有重构了的版本，在理解现 ET7 框架的基础上进行必要适配：先把之前总结的再熟悉一下，下午有时间也看看这个

- 下面是两个重复了的消息，需要删除掉：
 - C2G_LoginGate_Req ==> C2G_LoginGate 这些重复的消息，我还没有删除掉。改天最后整理清理源码的时候一起再删除
 - G2C_LoginGate_Ack ==> G2C_LoginGate

- 这里拿《NetInnerComponent》的方法可以参照：

```
Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(processConfig.InnerIPPort);
Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(NetworkHelper.ToIPEndPoint(${startMachineConfig.Inne
```

- **InnerConfig:** 可以把老版本里的 InnerConfig 类，参考对比 ET7 里的 ConfigSingleton<T> 泛型类，来试着理解和适配这相模块。因为同属配置类，就是分两个模块。
- 今天上午再继续看一模块。把昨天下午感觉有点儿不熟悉的：服务端如何管理随机分配给各客户端的各小服的编号等，以及客户端什么时候、如何进入地图服的弄明白。
- 因为上午头脑相对清醒，把遇见的凡不明白的模块，都试图理解透彻弄明白，比较 RouterAddressComponent
- 【地图服 Map 服】：在整理 MapHelper.cs 的逻辑的时候，因为游戏的这块儿逻辑不够熟悉，具体原理，或说是连接过程仍然不是很懂。
- 去想的话，感觉当用户注册或是登录帐户的时候，是随机分配一个网关服；框架里也随机分配给用户一个 Realm 注册登录服；当用户点击进入地图，或是重构游戏开始游戏的某个地方，是需要与地图服建立起连接的，大概如果有多个地图服，又随机分配一个。。框架里，虽然分配给每个用户的各个小服，是完全随机的，但是一旦分配给一个用户，除非用户登出下线、或是用户掉线，或是用户其它客户端顶号（用其它客户端的登录顶掉先前某个客户端的登录？这里先前分配的，会变吗？再读的时候看下这块儿），分配给用户的这些小服编号是可能会变的，与先前不同。但同户的同一个玩耍 session，应该是保持不变的。所以，框架里应该是有某些组件，是可以记录这些小服编号的。要找出来。
- 感觉我还是需要回去再读一下参考项目斗地主游戏里进入地图服的这块儿逻辑。当地图服要给某个用户发消息，MapHelper 这里的作用，应该就是帮助地图服找到有当前用户所在的网关服会话框，以便地图服向用户所在的网关服发消息。可是，感觉起来，地图服与网关服之间，不该是内网组件去管理吗？如果内网组件能够管理，MapHelper 就显得多余了呀。要去检查的还有 RealmHelper. 这里感觉没能理解透彻为什么 ET7 要把一个个好多个弄成帮助类。
- 【活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】
- 感觉我对源码的管理做得不够好。过程中为什么有很多类，过程中都没能看见呢？为什么分支里明明是有 LocationComponent 类，而我先前找不到？
- 【今天下午】：
 - 因为昨天读【路由器相关模块】，读得感觉比较懂一点儿了。今天下午所所有网络相关的模块再读一遍，什么时候添加的组件，事件的传递顺序等，都弄明白。修改补好笔记。
 - 今天下午会手洗半小时内衣，骑行内衣傍晚会穿出去骑约 1-1.5 小时。下午可能会花半小时左右手工。
 - 今天下午家里用大显示器，会希望再读至少 2.5 小时框架里的网络模块，把上午弄得半头三桩，羞于提交上 github 的笔记事理好。
 - 下午家里条件也方便，就把框架里的源码能爬多少爬多少出来。。【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!!!】
- 【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!!!】

9 Net 网络交互相关：【服务端 + 客户端】只是稍微改装成事件机制。没理解透、总结不全，待增补

- 当网络模块也改成是事件机制，可能是我先前对网络模块理解得还是不够透彻，怎么感觉 ET7 之后不知道会话框是怎么管理的？
- 我把可能相关的文件系统摆在一起，可是还没能真正系统性地自顶向下地梳理一遍事件传递过程。现在整理的阶段，仍然是翻源码的过程中，看见哪里没读过，或是没理解透彻，补上修改。但是当感觉把这块儿补完时，仍然需要一两个例子【自顶向下】，或自消息的发送，到返回消息至客户端，再梳理一遍整个流程。
- 现在还没弄清楚：Server, Client, Inner, 好像没有 Outer 了，几个相对模块算是怎么回事？不管是【网络服务端 NetServerComponent】，还是【网络客户端 NetClientComponent】组件，它们都管理无数个与这个端建立连接的会话框。

9.1 RpcInfo: 【消息的包装体】。结合 NetServerComponentOnReadEvent 来读。

- 在 NetServerComponentOnReadEvent 中，IResponse 是会话框上直接返回的。
- 因为接口的扩展性，我认为这是最基本的包装，可以适用和包装扩展接口，如 IRpcRequest, IRpcLocationResponse-etc. 框架里具体使用的地方，再多验证一下这个结论。
- 这里，后面的，IRpcRespon, IRpcLocationResponse 扩展接口，也该也都是 IResponse 接口吗？为什么 NetServerComponentOnReadEvent 就直接调用了会话框层面的 OnResponse() 呢。后面要再去多想这个问题，想明白

```
public readonly struct RpcInfo { // 【消息】包装体：可以是进程内的。可是它包装的是基类接口，与扩展接口如何区分？
    public readonly IRequest Request;
    public readonly ETTask<IResponse> Tcs;
    public RpcInfo(IRequest request) {
        this.Request = request;
        this.Tcs = ETTask<IResponse>.Create(true);
    }
}
```

9.2 NetServerComponent: NetServerComponentOnRead 结构体。

```
public struct NetServerComponentOnRead {
    public Session Session;
    public object Message;
}
[ComponentOf(typeof(Scene))]
public class NetServerComponent: Entity, IAwake<IPEndPoint>, IDestroy {
    public int ServiceId;
}
```

9.3 NetServerComponentSystem: 【服务端】组件：网络交互服务端的相关功能，事件发布等

```
[FriendOf(typeof(NetServerComponent))] // 【服务端组件】：负责【服务端】的网络交互部分
public static class NetServerComponentSystem {
    [ObjectSystem]
    public class AwakeSystem: AwakeSystem<NetServerComponent, IPPEndPoint> {
        protected override void Awake(NetServerComponent self, IPPEndPoint address) {
            self.ServiceId = NetServices.Instance.AddService(new KService(address, ServiceType.Outer));
            NetServices.Instance.RegisterAcceptCallback(self.ServiceId, self.OnAccept); // 网络交互的几个回调事件
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead);
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError);
        }
    }
}
```

```

    } // 省掉部分不必要的...
    private static void OnError(this NetServerComponent self, long channelId, int error) {
        Session session = self.GetChild<Session>(channelId);
        if (session == null) return;
        session.Error = error;
        session.Dispose();
    }
    // 这个 channelId 是由 CreateAcceptChannelId 生成的
    private static void OnAccept(this NetServerComponent self, long channelId, IPPEndPoint ipEndPoint) {
        // 【创建会话框】: 当此【服务端】组件, 接受了一个客户端, 就建一个与接收的【客户端】的会话框
        Session session = self.AddChildWithId<Session>(channelId, self.ServiceId);
        session.RemoteAddress = ipEndPoint;
    }
    // 只要不是这个鬼服 BenchmarkServer: 就加两个【服务端】的必要的, 防盗挂网不干事占带宽的盗贼, 和检查客户端状况
    if (self.DomainScene().SceneType != SceneType.BenchmarkServer) { // 区分: 同一功能, 【服务端】的处理逻辑, 与【客户端】的
        // 挂上这个组件, 5 秒就会删除 session, 所以客户端验证完成要删除这个组件。该组件的作用就是防止外挂一直连接不发消息也不进
        session.AddComponent<SessionAcceptTimeoutComponent>(); // 上面原标注: 【客户端验证】的逻辑, 改天去找
        // 客户端连接, 2 秒检查一次 recv 消息, 10 秒没有消息则断开 (与那个接收不到心跳包的客户端的连接)。【活宝妹就是一定要嫁给亲
        // 【自己的理解】: 【客户端】有心跳包告知服务端, 各客户端的连接状况; 【服务端】: 同样有服务端此组件来检测说, 哪个客户端掉线了
        session.AddComponent<SessionIdleCheckerComponent>();
    }
}
private static void OnRead(this NetServerComponent self, long channelId, long actorId, object message) {
    Session session = self.GetChild<Session>(channelId);
    if (session == null) return;
    session.LastRecvTime = TimeHelper.ClientNow();
    OpcodeHelper.LogMsg(self.DomainZone(), message);
    // 【发布事件】: 服务端组件读到了消息。这个事件发布, 事件的订阅者会收到通知, 处理相应必要逻辑
    EventSystem.Instance.Publish(Root.Instance.Scene, new NetServerComponentOnRead() {Session = session, Message = message});
}
}

```

9.4 NetServerComponentOnReadEvent: NetServerComponent 组件, 会发布事件, 触发此回调类

- 框架里, 是 NetServerComponent 会发布此事件, 触发这个回调类。去找, 有几个订阅者? 异步网络消息的发送, 与接收, 大多都还是单流程, 不事分支的, 所以全局只有这一个订阅者。
- 【服务端会话框】上的: 收到 IResponse 回复消息的情况, 调用会话框的 OnResponse() 方法处理。感觉理解起来困难。觉得服务端应该不会、或是极少出现这种情况吧。感觉更可能是客户端才出现这种情况呀。(服务端的内网回复消息也是有可能的, 只是现在想不到例子, 或说当服务端实际行驶客户端收返回消息功能的这个功能切分, 感觉理解起来困难)
- 【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!!!】

```

[Event(SceneType.Process)] // 【进程】层面: 来处理这个服务端组件事件
public class NetServerComponentOnReadEvent: AEvent<NetServerComponentOnRead> {
    protected override async ETTask Run(Scene scene, NetServerComponentOnRead args) { // 【返回消息的发送】: 仍是封装在框架底层
        Session session = args.Session;
        object message = args.Message;
    }
    // 【回复消息】: 前面的重点看错了, 重点是【回复】消息, 能够到达当前服务端组件, 就说明是属于当前进程收的消息 (想的话, 觉得是其它服务端转
        // 所以不区分 IRpcResponse IRpcLocationResponse 类型, 都向下交由会话框处理
        // 【服务端上, 会话框】Session: 发, 还是不发, 消息到 Channel 的另一头 // <<<<<<<<<<<<<<<<<< 【这部分没读懂】
        if (message is IResponse response) { // 【回复消息】: 就去服务端, 什么情况下会出现这种情况?
            // 【没读懂】: 服务端组件, 处理回复消息, 要不要【发】返回消息的步骤? 找不到哪里有, 发送, 的这个步骤。
            // 而会话框, 若真是从其它服转过来的消息, 现服务端不已经是会话框的另一端了吗 (内网消息也走会话框吗? 还得接着去读内网 Inne
            session.OnResponse(response); // 【会话框】: 直接处理返回消息。处理的逻辑也仅限于将 RpcInfo.Tcs 异步任务的返回结果写好
            return; // 这里返回: 我仍然找不到, 返回消息是, 如何【发送】回去的? 发送的过程步骤调用, 是哪里处理的?
        }
    }
    // 对于【服务端】来说, 【会话框】上, 只要把【异步任务 TCS】的结果写好填好, 异步网络 Session 底层, 会自动处理 Channel 客户
    // 根据消息接口判断是不是 Actor 消息, 不同的接口做不同的处理, 比如需要转发给 Chat Scene, 可以做一个 IChatMessage 接口
    switch (message) { // 【发送消息】+【不要求回复的消息】
        case IActorLocationRequest actorLocationRequest: { // gate session 收到 actor rpc 消息, 先向 actor 发送 rpc 请求,
            long unitId = session.GetComponent<SessionPlayerComponent>().PlayerId;
            int rpcId = actorLocationRequest.RpcId; // 这里要保存客户端的 rpcId
            long instanceId = session.InstanceId;
            IResponse iResponse = await ActorLocationSenderComponent.Instance.Call(unitId, actorLocationRequest);
            iResponse.RpcId = rpcId;
            // session 可能已经断开了, 所以这里需要判断
            if (session.InstanceId == instanceId)
                session.Send(iResponse);
        }
    }
}

```

```

            break;
        }
        case IActorLocationMessage actorLocationMessage: { // 【普通，不要求回复的位置消息】
            long unitId = session.GetComponent<SessionPlayerComponent>().PlayerId;
            ActorLocationSenderComponent.Instance.Send(unitId, actorLocationMessage);
            break;
        }
        case IActorRequest actorRequest: // 分发 IActorRequest 消息，目前没有用到，需要的自己添加
            break;
        case IActorMessage actorMessage: // 分发 IActorMessage 消息，目前没有用到，需要的自己添加
            break;
        default: {
            // 非 Actor 消息: MessageDispatcherComponent 全局单例吗？是的
            MessageDispatcherComponent.Instance.Handle(session, message);
            break;
        }
    }
}
}

```

9.5 NetClientComponent: 【网络客户端】组件：这个，感觉与【服务端】定义申明上看是一样的

- 【服务端】会话框上收【返回消息】的处理时，直接会话框上返回了返回消息。这里再试着从客户端的角度来找下，有没有相关的逻辑。
- 以前读框架，边读边打瞌睡，现在读一遍，会感觉收获很多。所以，今天下午还需要再接着认认真真读几个小时。

```

public struct NetClientComponentOnRead {
    public Session Session;
    public object Message;
}

[ComponentOf(typeof(Scene))]
public class NetClientComponent: Entity, IAwake<AddressFamily>, IDestroy {
    public int ServiceId;
}

```

9.6 NetClientComponentSystem: 【服务端】也是类似事件系统的改装

```

[FriendOf(typeof(NetClientComponent))] // 把这个【网络客户端】组件的主要笔记要点，再快速写一遍
public static class NetClientComponentSystem {
    [ObjectSystem]
    public class AwakeSystem: AwakeSystem<NetClientComponent, AddressFamily> {
        protected override void Awake(NetClientComponent self, AddressFamily addressFamily) { // 需要什么样的参数，就传什么
            self.ServiceId = NetServices.Instance.AddService(new KService(addressFamily, ServiceType.Outer)); // 开启了与这个
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead); // 注册订阅【读】网络消息事件，应该是从网络
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError); // 注册订阅【出错】事件
        }
    }
    [ObjectSystem]
    public class DestroySystem: DestroySystem<NetClientComponent> {
        protected override void Destroy(NetClientComponent self) {
            NetServices.Instance.RemoveService(self.ServiceId); // 直接移除这个网络服务
        }
    }
    private static void OnRead(this NetClientComponent self, long channelId, long actorId, object message) {
        Session session = self.GetChild<Session>(channelId); // 拿：相应的会话框
        if (session == null) { // 空：直接返回
            return;
        }
        session.LastRecvTime = TimeHelper.ClientNow();
        OpcodeHelper.LogMsg(self.DomainZone(), message);
    }
    // 发布事件：事件的接收者，应该是【客户端】的 Session 层面的进一步读取消息内容（内存流上读消息？），改天再去细看。
    EventSystem.Instance.Publish(Root.Instance.Scene, new NetClientComponentOnRead() {Session = session, Message = message});
}
private static void OnError(this NetClientComponent self, long channelId, int error) {
    Session session = self.GetChild<Session>(channelId); // 同样，先去拿会话框：因为这些异步网络的消息传递，都是建立在一个个
    if (session == null) // 空：直接返回

```

```

        return;
    session.Error = error;
    session.Dispose();
}
public static Session Create(this NetClientComponent self, IPEndPoint realIPEndPoint) {
    long channelId = NetServices.Instance.CreateConnectChannelId();
    Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId); // 创建必要的会话框，方便通信
    session.RemoteAddress = realIPEndPoint;
    if (self.DomainScene().SceneType != SceneType.Benchmark) {
        session.AddComponent<SessionIdleCheckerComponent>(); // 不知道这个是干什么的，改天再看
    }
    NetServices.Instance.CreateChannel(self.ServiceId, session.Id, realIPEndPoint); // 创建信道
    return session;
}
public static Session Create(this NetClientComponent self, IPEndPoint routerIPEndPoint, IPEndPoint realIPEndPoint, uint
    long channelId = localConn;
    Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
    session.RemoteAddress = realIPEndPoint;
    if (self.DomainScene().SceneType != SceneType.Benchmark) {
        session.AddComponent<SessionIdleCheckerComponent>();
    }
    NetServices.Instance.CreateChannel(self.ServiceId, session.Id, routerIPEndPoint);
    return session;
}
}

```

9.7 NetClientComponentOnReadEvent: 框架里只有这一个注册过的回调事件

- 感觉我这部分的源码出问题了，需要去网络上翻一下原本的源码，对比一下
- 以后管理源码的时候要小心，就不知道前面什么时候弄的，项目开始的时候，就该从头重新开一个新的分支。

```

[Event(SceneType.Process)]
public class NetClientComponentOnReadEvent : AEvent<NetClientComponentOnRead> {
    protected override async ETTask Run(Scene scene, NetClientComponentOnRead args) {
        Session session = args.Session;
        object message = args.Message;
        if (message is IResponse response) { // 这里是回复消息，就交给会话框去处理
            session.OnResponse(response); // 由会话框层往下走
            return;
        }
        // 普通消息或者是 Rpc 请求消息
        MessageDispatcherComponent.Instance.Handle(session, message);
        await ETTask.CompletedTask;
    }
}

```

9.8 NetInnerComponent: 【服务端】对不同进程的处理组件。是服务器的组件

```

namespace ET.Server {
    // 【服务器】: 对不同进程的一些处理
    public struct ProcessActorId {
        public int Process;
        public long ActorId;
        public ProcessActorId(long actorId) {
            InstanceIdStruct instanceIdStruct = new InstanceIdStruct(actorId);
            this.Process = instanceIdStruct.Process;
            instanceIdStruct.Process = Options.Instance.Process;
            this.ActorId = instanceIdStruct.ToLong();
        }
    }
    // 下面这个结构体：可以用来封装发布内网读事件
    public struct NetInnerComponentOnRead {
        public long ActorId;
        public object Message;
    }
}
[ComponentOf(typeof(Scene))]

```

```

public class NetInnerComponent: Entity, IAwake<IPEndPoint>, IAwake, IDestroy {
    public int ServiceId;

    public NetworkProtocol InnerProtocol = NetworkProtocol.KCP;
    [StaticField]
    public static NetInnerComponent Instance;
}

}

```

9.9 NetInnerComponentSystem: 生成系

- 处理内网消息：它发布了一个内网读到消息的事件。那么订阅过它的客户端？相关事件会被触发。去看 NetClientComponentOnReadEvent 类

```

[FriendOf(typeof(NetInnerComponent))]
public static class NetInnerComponentSystem {
    [ObjectSystem]
    public class NetInnerComponentAwakeSystem: AwakeSystem<NetInnerComponent> {
        protected override void Awake(NetInnerComponent self) {
            NetInnerComponent.Instance = self;
            switch (self.InnerProtocol) {
                case NetworkProtocol.TCP:
                    self.ServiceId = NetServices.Instance.AddService(new TService(AddressFamily.InterNetwork, ServiceType.Inner));
                    break;
                case NetworkProtocol.KCP:
                    self.ServiceId = NetServices.Instance.AddService(new KService(AddressFamily.InterNetwork, ServiceType.Inner));
                    break;
            }
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead);
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError);
        }
    }
    [ObjectSystem]
    public class NetInnerComponentAwake1System: AwakeSystem<NetInnerComponent, IPPEndPoint> {
        protected override void Awake(NetInnerComponent self, IPPEndPoint address) {
            NetInnerComponent.Instance = self;
            switch (self.InnerProtocol) {
                case NetworkProtocol.TCP:
                    self.ServiceId = NetServices.Instance.AddService(new TService(address, ServiceType.Inner));
                    break;
                case NetworkProtocol.KCP:
                    self.ServiceId = NetServices.Instance.AddService(new KService(address, ServiceType.Inner));
                    break;
            }
            NetServices.Instance.RegisterAcceptCallback(self.ServiceId, self.OnAccept);
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead);
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError);
        }
    }
    [ObjectSystem]
    public class NetInnerComponentDestroySystem: DestroySystem<NetInnerComponent> {
        protected override void Destroy(NetInnerComponent self) {
            NetServices.Instance.RemoveService(self.ServiceId);
        }
    }
    private static void OnRead(this NetInnerComponent self, long channelId, long actorId, object message) {
        Session session = self.GetChild<Session>(channelId);
        if (session == null)
            return;
        session.LastRecvTime = TimeHelper.ClientFrameTime();
        self.HandleMessage(actorId, message);
    }
    // 这里，内网组件，处理内网消息看出，这些都重构成了事件机制，发布根场景内网组件读到消息事件
    public static void HandleMessage(this NetInnerComponent self, long actorId, object message) {
        EventSystem.Instance.Publish(Root.Instance.Scene, new NetInnerComponentOnRead() { ActorId = actorId, Message = message });
    }
    private static void OnError(this NetInnerComponent self, long channelId, int error) {
        Session session = self.GetChild<Session>(channelId);
        if (session == null) {

```

```

        return;
    }
    session.Error = error;
    session.Dispose();
}
// 这个 channelID 是由 CreateAcceptChannelId 生成的
private static void OnAccept(this NetInnerComponent self, long channelId, IPPEndPoint ipEndPoint) {
    Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
    session.RemoteAddress = ipEndPoint;
    // session.AddComponent<SessionIdleCheckerComponent, int, int, int>(NetThreadComponent.checkInterval, NetThread
}
private static Session CreateInner(this NetInnerComponent self, long channelId, IPPEndPoint ipEndPoint) {
    Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
    session.RemoteAddress = ipEndPoint;
    NetServices.Instance.CreateChannel(self.ServiceId, channelId, ipEndPoint);
    // session.AddComponent<InnerPingComponent>();
    // session.AddComponent<SessionIdleCheckerComponent, int, int, int>(NetThreadComponent.checkInterval, NetThread
    return session;
}
// 内网 actor session, channelId 是进程号。【自己的理解】：这些内网服务器间，或说重构的 SceneType 间，有维护着会话框的，比
public static Session Get(this NetInnerComponent self, long channelId) {
    Session session = self.GetChild<Session>(channelId);
    if (session != null) { // 有已经创建过，就直接返回
        return session;
    } // 下面，还没创建过，就创建一个会话框
    IPPEndPoint ipEndPoint = StartProcessConfigCategory.Instance.Get((int) channelId).InnerIPPort;
    session = self.CreateInner(channelId, ipEndPoint);
    return session;
}
}

```

9.10 MessageDispatcherInfo: 在【MessageDispatcherComponent】中

9.11 MessageDispatcherComponent: 全局全框架单例: 【活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】

```

// 总管：对每个场景 SceneType，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，只有一个消息处理器实体类（可以去验证这点儿）
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 {
// 按下面的字典看，消息分发器，全局单例，是的！【活宝妹就是一定要嫁给亲爱的表哥!!】
    public static MessageDispatcherComponent Instance { get; set; } // 【全局单例】
    public readonly Dictionary<ushort, List<MessageDispatcherInfo>> Handlers = new(); // 总管的字典
}

```

- 这个组件全局单例，添加的地主是在框架服务器启动的时候，公共组件部分的添加。组件的字典，会管理全框架下所有的 MessageDispatcherInfo 相产。

- 来自于文件 EntryEvent1_InitShare:

```

// 公用的相关组件的初始化：
[Event(SceneType.Process)]
public class EntryEvent1_InitShare: AEvent<EventType.EntryEvent1> {
    // 【全局单例】组件：
    protected override async ETTask Run(Scene scene, EventType.EntryEvent1 args) {
        Root.Instance.Scene.AddComponent<NetThreadComponent>();
        Root.Instance.Scene.AddComponent<OpcodeTypeComponent>();
        Root.Instance.Scene.AddComponent<MessageDispatcherComponent>(); // <<<<<<<<<<<<<<
        Root.Instance.Scene.AddComponent<NumericWatcherComponent>();
        Root.Instance.Scene.AddComponent<AI DispatcherComponent>();
        Root.Instance.Scene.AddComponent<ClientSceneManagerComponent>();
    }
}

```

```

        await ETTTask.CompletedTask;
    }
}

```

9.12 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($"message handle {type.Name} 需要继承 IMHandler");
            continue;
        }
        object[] attrs = type.GetCustomAttributes(typeof(MessageHandlerAttribute), false);
        foreach (object attr in attrs) {
            MessageHandlerAttribute messageHandlerAttribute = attr as MessageHandlerAttribute;
            Type messageType = iMHandler.GetType();
            ushort opcode = NetServices.Instance.GetOpcode(messageType); // 这里相对、理解上的困难是：感觉无法把 OpCode 网络操作
            if (opcode == 0) {
                Log.Error($" 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);
        }
    }
}
}

```

9.13 MessageDispatcherComponentHelper:

- 【会话框】：哈哈哈，这是会话框两端，哪一端的场景呢？分不清。。。去找出来！客户端？网关服？就是说，这里的消息分发处理，还是没有弄明白的。

```

// 消息分发组件
[FriendOf(typeof(MessageDispatcherComponent))]
public static class MessageDispatcherComponentHelper { // Awake() etc...
    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);
            if (opcode == 0) {
                Log.Error($" 消息 opcode 为 0: {messageType.Name}");
                continue;
            }
            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);
}

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;
    }
    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? 分不清...
    foreach (MessageDispatcherInfo ev in actions) {
        if (ev.SceneType != sceneType)
            continue;
        try {
            ev.IMHandler.Handle(session, message);
        }
        catch (Exception e) {
            Log.Error(e);
        }
    }
}
}
}

```

9.14 SessionIdleCheckerComponent: 【会话框】闲置状态管理组件

- 【会话框】闲置状态管理组件：当服务器太忙，一个会话框闲置太久，有没有什么逻辑会回收闲置会话框来提高服务器性能什么之类的？
- 框架里 ET 命名空间：设置的机制是，任何会话框，超过 30 秒不曾发送和接收过（要 30 秒内既发送过也接收到过消息）消息，都算作超时，回收，提到服务器性能。

```

// 【会话框】闲置状态管理组件：当服务器太忙，一个会话框闲置太久，有没有什么逻辑会回收闲置会话框来提高服务器性能什么之类的？
[ComponentOf(typeof(Session))]
public class SessionIdleCheckerComponent : Entity, IAwake, IDestroy {
    public long RepeatedTimer;
}

```

9.15 SessionIdleCheckerComponentSystem: SessionIdleChecker 激活类，

- 这是前面读过的、类似实现原理的超时机制。感觉这个类，现在读起来很简单。没有门槛。

```

[Invoke(TimerInvokeType.SessionIdleChecker)]
public class SessionIdleChecker : ATimer<SessionIdleCheckerComponent> {
    protected override void Run(SessionIdleCheckerComponent self) {
        try {
            self.Check();
        } catch (Exception e) {
            Log.Error($"move timer error: {self.Id}\n{e}");
        }
    }
}

```

```

        }
    }
    [ObjectSystem]
    public class SessionIdleCheckerComponentAwakeSystem : AwakeSystem<SessionIdleCheckerComponent> {
        protected override void Awake(SessionIdleCheckerComponent self) {
            // 同样设置：【重复闹钟】：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!!!
            self.RepeatedTimer = TimerComponent.Instance.NewRepeatedTimer(SessionIdleCheckerComponentSystem.CheckInterval,
                () => ...
            );
        }
    }
    public static class SessionIdleCheckerComponentSystem {
        public const int CheckInterval = 2000; // 每隔 2 秒
        public static void Check(this SessionIdleCheckerComponent self) {
            Session session = self.GetParent<Session>();
            long timeNow = TimeHelper.ClientNow();
            // 常量类定义：会话框最长每个 30 秒；
            // 判断：30 秒内，曾经发送过消息，并且也接收过消息，直接返回；否则，算作【会话框】超时
            if (timeNow - session.LastRecvTime < ConstValue.SessionTimeoutTime && timeNow - session.LastSendTime < ConstValue.SessionTimeoutTime)
                return;
            Log.Info($"session timeout: {session.Id} {timeNow} {session.LastRecvTime} {session.LastSendTime} {timeNow - session.LastRecvTime} {timeNow - session.LastSendTime}");
            session.Error = ErrorCore.ERR_SessionSendOrRecvTimeout; // 【会话框】超时回收
            session.Dispose();
        }
    }
}

```

9.16 MessageHelper: 不知道这个类是做什么用的，使用场景等。过会儿看下

- 这个类，仍然是桥接，类的各个方面里，所调用的是 ActorMessageSenderComponent 里所定义的方法，来实现发送 Actor 消息等。

```

public static class MessageHelper {
    public static void NoticeUnitAdd(Unit unit, Unit sendUnit) {
        M2C_CreateUnits createUnits = new M2C_CreateUnits() { Units = new List<UnitInfo>() };
        createUnits.Units.Add(UnitHelper.CreateUnitInfo(sendUnit));
        MessageHelper.SendToClient(unit, createUnits);
    }
    public static void NoticeUnitRemove(Unit unit, Unit sendUnit) {
        M2C_RemoveUnits removeUnits = new M2C_RemoveUnits() { Units = new List<long>() };
        removeUnits.Units.Add(sendUnit.Id);
        MessageHelper.SendToClient(unit, removeUnits);
    }
    public static void Broadcast(Unit unit, IActorMessage message) {
        Dictionary<long, AOIEntity> dict = unit.GetBeSeePlayers();
        // 网络底层做了优化，同一个消息不会多次序列化
        foreach (AOIEntity u in dict.Values) {
            ActorMessageSenderComponent.Instance.Send(u.Unit.GetComponent<UnitGateComponent>().GateSessionActorId, message);
        }
    }
    public static void SendToClient(Unit unit, IActorMessage message) {
        SendActor(unit.GetComponent<UnitGateComponent>().GateSessionActorId, message);
    }
    // 发送协议给 ActorLocation
    public static void SendToLocationActor(long id, IActorLocationMessage message) {
        ActorLocationSenderComponent.Instance.Send(id, message);
    }
    // 发送协议给 Actor
    public static void SendActor(long actorId, IActorMessage message) {
        ActorMessageSenderComponent.Instance.Send(actorId, message);
    }
    // 发送 RPC 协议给 Actor
    public static async ETTask<IActorResponse> CallActor(long actorId, IActorRequest message) {
        return await ActorMessageSenderComponent.Instance.Call(actorId, message);
    }
    // 发送 RPC 协议给 ActorLocation
    public static async ETTask<IActorResponse> CallLocationActor(long id, IActorLocationRequest message) {
        return await ActorLocationSenderComponent.Instance.Call(id, message);
    }
}

```

9.17 ActorHandleHelper: 是谁调用它，什么场景下使用的？这个，今天下午再补吧

```
public static class ActorHandleHelper {
    public static void Reply(int fromProcess, IActorResponse response) {
        if (fromProcess == Options.Instance.Process) { // 返回消息是同一个进程：没明白，这里为什么就断定是同一进程的消息了？直接
            // NetInnerComponent.Instance.HandleMessage(realActorId, response); // 等同于直接调用下面这句【我自己暂时放回来的】
            ActorMessageSenderComponent.Instance.HandleIActorResponse(response); // 【没读懂】同一个进程内的消息，不走网络层，
            return;
        }
        // 【不同进程的消息处理】走网络层，就是调用会话框来发出消息
        Session replySession = NetInnerComponent.Instance.Get(fromProcess); // 从内网组件单例中去拿会话框：不同进程消息，一定走
        replySession.Send(response);
    }
    public static void HandleIActorResponse(IActorResponse response) {
        ActorMessageSenderComponent.Instance.HandleIActorResponse(response);
    }
    // 分发 actor 消息
    [EnableAccessEntityChild]
    public static async ETTask HandleIActorRequest(long actorId, IActorRequest iActorRequest) {
        InstanceIdStruct instanceIdStruct = new(actorId);
        int fromProcess = instanceIdStruct.Process;
        instanceIdStruct.Process = Options.Instance.Process;
        long realActorId = instanceIdStruct.ToInt();
        Entity entity = Root.Instance.Get(realActorId);
        if (entity == null) {
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Warning($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorRequest}");
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId) {
                        IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
                        Reply(fromProcess, response);
                        break;
                    } // 调用管理器组件的处理方法
                    await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorRequest);
                }
                break;
            }
            case MailboxType.UnOrderMessageDispatcher: {
                await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorRequest);
                break;
            }
            case MailboxType.GateSession:
            default:
                throw new Exception($"no mailboxtype: {mailBoxComponent.MailboxType} {iActorRequest}");
        }
    }
    // 分发 actor 消息
    [EnableAccessEntityChild]
    public static async ETTask HandleIActorMessage(long actorId, IActorMessage iActorMessage) {
        InstanceIdStruct instanceIdStruct = new(actorId);
        int fromProcess = instanceIdStruct.Process;
        instanceIdStruct.Process = Options.Instance.Process;
        long realActorId = instanceIdStruct.ToInt();
        Entity entity = Root.Instance.Get(realActorId);
        if (entity == null) {
            Log.Error($"not found actor: {realActorId} {iActorMessage}");
            return;
        }
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Error($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorMessage}");
            return;
        }
```

```

        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId)
                        break;
                    await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                }
                break;
            }
            case MailboxType.UnOrderMessageDispatcher: {
                await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                break;
            }
            case MailboxType.GateSession: {
                if (entity is Session gateSession)
                    // 发送给客户端
                    gateSession.Send(iActorMessage);
                break;
            }
            default:
                throw new Exception($"no mailboxtype: {mailBoxComponent.MailboxType} {iActorMessage}");
        }
    }
}

```

- 【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!!!】
- 【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!!!】