

# ET 框架学习笔记（二） - - 网络交互相关

deepwaterooo

June 6, 2023

## Contents

<b>1</b>	<b>Net 网络交互相关：【服务端 + 客户端】只是稍微改装成事件机制而已</b>	<b>1</b>
1.1	NetInnerComponent: 【服务端】对不同进程的处理组件。是服务器的组件	1
1.2	NetInnerComponentSystem: 生成系	2
1.3	NetServerComponent:	3
1.4	NetServerComponentSystem: 生成系	3
1.5	NetClientComponent: 【客户端】组件	4
1.6	NetClientComponentSystem: 【服务端】也是类似事件系统的改装	4
1.7	NetClientComponentOnReadEvent: 框架里只有这一个注册过的回调事件	4
1.8	MessageDispatcherComponentHelper:	5
<b>2</b>	<b>IAwake 接口类系统, IStart 重构丢了</b>	<b>6</b>
2.1	IMessage,IRequest,IResponse: 进程内? 消息类	6
2.2	IActorMessage,IActorRequest,IActorResponse: 进程间的? 消息类	6
2.3	IActorLocationMessage: 进程间的位置消息相关	6
2.4	IMHandler,IMActorHandler: 消息处理器口类【傻傻分不清楚】	6
2.5	ILoad,ISystemType: 加载系	6
2.6	IAwake: 最多可以带四个参数	7
2.7	IStartSystem,StartSystem<T>: 自己加的。【还有问题】系统找不到	7
2.8	IUpdateSystem:	7
2.9	ILateUpdate: 好像是用于物理引擎, 或是相机什么的更新, 生命周期回调	8
2.10	ISingletonAwake Update LateUpdate: Singleton 生命周期回调	8
2.11	ISingleton,Singleton<T>: 单例	8
2.12	IDestroy,IDestroySystem,DestroySystem<T>: 销毁系	9
2.13	IEvent,AEvent<A>: 事件	9
2.14	IAddComponent: 添加组件系	9
2.15	IGetComponent: 获取组件系。【这里没有看明白】: 再去找细节 // ««««««««««	10
2.16	ISerializeToEntity,IDeserialize,IDeserializeSystem,DeserializeSystem<T>: 序列化, 反序列化	10
2.17	IInvoke,AInvokeHandler<A>,AInvokeHandler<A, T>: 激活类	10
2.18	ProtoBuf 相关: IExtensible,IExtension,IProtoOutput<TOutput>,IMeasuredProtoOutput<TOutput>	
	看不懂	11
2.18.1	IExtensible	11
2.18.2	IExtension	11
2.18.3	IProtoOutput<TOutput>,IMeasuredProtoOutput<TOutput>,MeasureState<T>:	
	看得头大	12

<b>3</b>	<b>【拖拉机游戏房间】组件: 分析</b>	<b>12</b>
3.1	TractorRoomEvent: 拖拉机房间, 【待修改完成】	12
3.2	GamerComponent: 玩家【管理类组件】, 是对房间里四个玩家的管理。	12
3.3	Gamer: 【服务端】一个玩家个例。对应这个玩家的相关信息	13
3.4	Gamer: 【客户端】一个玩家个例。它说只要一点儿信息就行	14
3.5	GamerUIComponent: 【客户端】玩家 UI 组件: 每个玩家背个小面板, 来显示必要信息 (钱, 抢不抢庄, 反过的主等)	14
3.6	Protobuf 里面的消息与参考	15
3.7	TractorRoomComponent: 游戏房间, 自带其它组件, 当有嵌套时, 如何才能系统化地、工厂化地、UI 上的事件驱动地, 生成这个组件呢?	16
3.8	TractorInteractionComponent: 感觉是视图 UI 上的一堆调控, 逻辑控制	16
<b>4</b>	<b>消息处理器: AMActorHandler&lt;E, Message&gt; 继承类的返回类型, 全改成了 void</b>	<b>16</b>
4.1	AMActorHandler<E, Message>: 基类的抽象方法 Run 的返回类型被固定死了, 报了很多错	16
4.2	IMActorHandler: 接口类的定义, 同样要改	17
<b>5</b>	<b>Protobuf 里的 enum: 【Identity】【Suits】【Weight】</b>	<b>17</b>
5.1	OuterMessage_C_10001.proto 里三四个类的定义	17
5.2	【参考项目】里: enum 是可以顺利写进 ETModel 声明的命名空间, 并且源码可见	18
5.3	ET7 框架里, enum 完全找不到	18
5.4	ETModel_Card_Binding: 奇异点, ILRuntime 热更新里, 似乎对 Card 类的两个成员变量作了辅助链接	19
<b>6</b>	<b>ET7 数据库相关【服务端】</b>	<b>19</b>
6.1	IDBCollection: 主要是方便写两个不同的数据库 (好像是 GeekServer 里两个数据库)。反正方便扩展吧	19
6.2	DBComponent: 带生成系。可以查表, 查询数据	19
6.3	DBManagerComponent: 有上面的 DBComponent 数组。数组长度固定吗?	19
6.4	DBManagerComponentSystem: 主要是查询某个区服的数据库, 从数组里	20
6.5	DBProxyComponent: 【参考项目】里的。有生成系。	20
<b>7</b>	<b>StartConfigComponent: 找【各种服】的起始初始化地址</b>	<b>20</b>
7.1	ConfigSingleton<T>: ProtoObject, ISingleton	20
7.2	StartProcessConfigCategory: ConfigSingleton<StartProcessConfigCategory>, IMerge:	21
7.3	SceneFactory 里可以给【匹配服】添加组件	21
7.4	RouterAddressComponent: 路由器组件	22
7.5	RouterAddressComponentSystem: 路由器的生成系	22
7.6	RouterHelper: 路由器帮助类, 向路由器注册、申请?	23
7.7	LoginHelper: 登录服的获取地址的方式来获取匹配服的地址了。全框架只有这一个黄金案例	24
7.8	HttpGetRouterResponse: 这个 ProtoBuf 的消息类型	24
<b>8</b>	<b>组件定义, 再澄明, 与去重</b>	<b>25</b>
8.1	OnlineComponent: 参考项目里的, 现框架里查找一下	25
8.2	框架 Game 类: 是单例的管理类, 与服务端或是客户端的总、根场景无关	25
8.3	ET7 的重构, 将数据库相关全部去掉了? 找不到数据库的踪影?	26
8.4	GamerFactory: 【加工厂】全部移除掉	26
<b>9</b>	<b>写在最后: 反而是自己每天查看一再更新的</b>	<b>26</b>
<b>10</b>	<b>现在的修改内容, 记忆</b>	<b>27</b>

## 1 Net 网络交互相关：【服务端 + 客户端】只是稍微改装成事件机制而已

- 以前看得比较空洞。今天读源码看见一个例子（同一进程内消息，不走网络层，直接处理消息的例子），顺着那个实例再理解一遍。
- 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？
  - 内网的话，内网组件创建、记录管理同其它内网场景的会话框；外网，可以再看一下
- 这次重构里，以前是 Model 里，生成系为非静态，这次全搬进热更新域里，生成系与方法全是静态的。
- 因为生成系全变成了静态，那么调用方法就成为比如：直接使用生成系的类名，与静态方法调用。

NetInnerComponent.Instance.HandleMessage(realActorId, response); // 等同于直接调用下面这句【这是它给出来的例子】  
 // 上面这种，就必须组件里，而非生成系里，已经声明了公用方法，否则用下面的  
 Session matchSession = NetInnerComponentSystem.Get(matchIPEndPoint);

### 1.1 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;
    }
}
```

### 1.2 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 NetInnerComponentAwakeSystem: AwakeSystem<NetInnerComponent, IPEndPoint> {
        protected override void Awake(NetInnerComponent self, IPEndPoint 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, IPEndPoint ipEndPoint) {
        Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
        session.RemoteAddress = ipEndPoint;
        // session.AddComponent<SessionIdleCheckerComponent, int, int, int>(NetThreadComponent.checkInterval, NetThreadComponent.checkInterval, NetThreadComponent.checkInterval);
    }
    private static Session CreateInner(this NetInnerComponent self, long channelId, IPEndPoint 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, NetThreadComponent.checkInterval, NetThreadComponent.checkInterval);
        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，比如 SessionManager。
    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
        IPEndPoint ipEndPoint = StartProcessConfigCategory.Instance.Get((int) channelId).InnerIPPort;
        session = self.CreateInner(channelId, ipEndPoint);
        return session;
    }
}

```

### 1.3 NetServerComponent:

```

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

```

### 1.4 NetServerComponentSystem: 生成系

```

[FriendOf(typeof(NetServerComponent))]
public static class NetServerComponentSystem {
    [ObjectSystem]
    public class AwakeSystem: AwakeSystem<NetServerComponent, IPEndPoint> {
        protected override void Awake(NetServerComponent self, IPEndPoint address) {
            self.ServiceId = NetServices.Instance.AddService(new KService(address, ServiceType.Outer));
            NetServices.Instance.RegisterAcceptCallback(self.ServiceId, self.OnAccept);
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead);
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError);
        }
    }
    [ObjectSystem]
    public class NetKcpComponentDestroySystem: DestroySystem<NetServerComponent> {
        protected override void Destroy(NetServerComponent self) {
            NetServices.Instance.RemoveService(self.ServiceId);
        }
    }
    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, IPEndPoint ipEndPoint) {
        Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
        session.RemoteAddress = ipEndPoint;
        if (self.DomainScene().SceneType != SceneType.BenchmarkServer) {
            // 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>();
            // 客户端连接，2 秒检查一次 recv 消息，10 秒没有消息则断开
            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);
        EventSystem.Instance.Publish(Root.Instance.Scene, new NetServerComponentOnRead() {Session = session, Message = message});
    }
}

```

## 1.5 NetClientComponent: 【客户端】组件

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

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

```
[FriendOf(typeof(NetClientComponent))]
public static class NetClientComponentSystem { // ... Awake() etc
    private static void OnRead(this NetClientComponent self, long channelId, long actorId, object message) {
        Session session = self.GetChild<Session>(channelId);
        if (session == null) // 总是检查: 会话框是否已经销毁了
            return;
        session.LastRecvTime = TimeHelper.ClientNow();
        OpcodeHelper.LogMsg(self.DomainZone(), message);
        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);
        if (session == null)
            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);
        session.RemoteAddress = realIPEndPoint;
        if (self.DomainScene().SceneType != SceneType.Benchmark) {
            session.AddComponent<SessionIdleCheckerComponent>();
        }
        NetServices.Instance.CreateChannel(self.ServiceId, session.Id, realIPEndPoint);
        return session;
    }
    public static Session Create(this NetClientComponent self, IPEndPoint routerIPEndPoint, IPEndPoint realIPEndPoint, uint localConn) {
        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;
    }
}
```

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

```
[Event(SceneType.Process)]
public class NetClientComponentOnReadEvent: AEvent<NetClientComponentOnRead> {
    protected override async ETask 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 ETask.CompletedTask;
    }
}
```

## 1.8 MessageDispatcherComponentHelper:

- **【会话框】**: 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？

```
// 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，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);
            }
        }
    }
}
```

## 2 IAwake 接口类系统, IStart 重构丢了

- 感觉还比较直接，就是帮助搭建热更新域与 Unity 常规工程域生命周期回调的桥，搭桥连线，连能就可以了。应该可以扩散出个 IStart 接口类

### 2.1 IMessage, IRequest, IResponse: 进程内？消息类

```
public interface IMessage {}
public interface IRequest: IMessage {
    int RpcId { get; set; }
}
```

```
public interface IResponse: IMessage {
    int Error { get; set; }
    string Message { get; set; }
    int RpcId { get; set; }
}
```

## 2.2 IActorMessage,IActorRequest,IActorResponse: 进程间的？消息类

```
// 不需要返回消息
public interface IActorMessage: IMessage {}
public interface IActorRequest: IRequest {}
public interface IActorResponse: IResponse {}
```

## 2.3 IActorLocationMessage: 进程间的位置消息相关

```
public interface IActorLocationMessage: IActorRequest {}
public interface IActorLocationRequest: IActorRequest {}
public interface IActorLocationResponse: IActorResponse {}
```

## 2.4 IMHandler,IMActorHandler: 消息处理器口类【傻傻分不清楚】

```
public interface IMHandler { // 同进程内的
    void Handle(Session session, object message);
    Type GetMessageType();
    Type GetResponseType();
}
public interface IMActorHandler { // 进程间的？
    // ETask Handle(Entity entity, int fromProcess, object actorMessage);
    void Handle(Entity entity, int fromProcess, object actorMessage); // 自己改成这样的
    Type GetRequestType();
    Type GetResponseType();
}
```

## 2.5 ILoad,ISystemType: 加载系

```
public interface ISystemType {
    Type Type();
    Type SystemType();
    InstanceQueueIndex GetInstanceQueueIndex();
}

public interface ILoad {
}
public interface ILoadSystem: ISystemType {
    void Run(Entity o);
}
[ObjectSystem]
public abstract class LoadSystem<T> : ILoadSystem where T: Entity, ILoad {
    void ILoadSystem.Run(Entity o) {
        this.Load((T)o);
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    Type ISystemType.SystemType() {
        return typeof(ILoadSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.Load;
    }
    protected abstract void Load(T self);
}
```

## 2.6 IAwake: 最多可以带四个参数

```
public interface IAwake {}
public interface IAwake<A> {}
public interface IAwake<A, B> {}
```



```
public interface IAwake<A, B, C> {}
public interface IAwake<A, B, C, D> {}
```

## 2.7 IStartSystem, StartSystem<T>: 自己加的。【还有问题】系统找不到

```
public interface IStart { }
public interface IStartSystem : ISystemType {
    void Run(Entity o);
}
[ObjectSystem]
public abstract class StartSystem<T> : IStartSystem where T: Entity, IStart {
    public void IStartSystem.Run(Entity o) {
        this.Start((T)o);
    }
    public Type ISystemType.Type() {
        return typeof(T);
    }
    public Type ISystemType.SystemType() {
        return typeof(IStartSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() { // 这里没看懂在干什么，大概还有个地方，我得去改
        return InstanceQueueIndex.Start;
    }
    public abstract void Start(T self);
}
// 整合进了系统: InstanceQueueIndex
public enum InstanceQueueIndex {
    None = -1,
    Start, // 需要把这个回调加入框架统筹管理里去
    Update,
    LateUpdate,
    Load,
    Max,
}
```

- 参考项目：除了原文件放在 ET 域。也【复制了一份到客户端的热更新域里】。可是感觉不应该。因为其它所有的回调都不用复制就可以用。我哪里可能还是没能设置对
- 改天再检查一下。但是否，对于非系统框架扩展接口，不得不这样？仍然感觉不应该，因为系统框架里其它的生命周期回调函数都不需要复制。改天再做。

## 2.8 IUpdateSystem:

```
public interface IUpdate {
}
public interface IUpdateSystem: ISystemType {
    void Run(Entity o);
}
[ObjectSystem]
public abstract class UpdateSystem<T> : IUpdateSystem where T: Entity, IUpdate {
    void IUpdateSystem.Run(Entity o) {
        this.Update((T)o);
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    Type ISystemType.SystemType() {
        return typeof(IUpdateSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.Update;
    }
    protected abstract void Update(T self);
}
```

## 2.9 ILateUpdate: 好像是用于物理引擎，或是相机什么的更新，生命周期回调

```
public interface ILateUpdate {
}
```

```

public interface ILateUpdateSystem: ISystemType {
    void Run(Entity o);
}
[ObjectSystem]
public abstract class LateUpdateSystem<T> : ILateUpdateSystem where T: Entity, ILateUpdate {
    void ILateUpdateSystem.Run(Entity o) {
        this.LateUpdate((T)o);
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    Type ISystemType.SystemType() {
        return typeof(ILateUpdateSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.LateUpdate;
    }
    protected abstract void LateUpdate(T self);
}

```

## 2.10 ISingletonAwake|Update|LateUpdate: Singleton 生命周期回调

```

public interface ISingletonAwake {
    void Awake();
}
public interface ISingletonUpdate {
    void Update();
}
public interface ISingletonLateUpdate {
    void LateUpdate();
}

```

## 2.11 ISingleton, Singleton<T>: 单例

```

public interface ISingleton: IDisposable {
    void Register();
    void Destroy();
    bool IsDisposed();
}
public abstract class Singleton<T>: ISingleton where T: Singleton<T>, new() {
    private bool isDisposed;
    [StaticField]
    private static T instance;
    public static T Instance {
        get {
            return instance;
        }
    }
    void ISingleton.Register() {
        if (instance != null) {
            throw new Exception($"singleton register twice! {typeof (T).Name}");
        }
        instance = (T)this;
    }
    void ISingleton.Destroy() {
        if (this.isDisposed) {
            return;
        }
        this.isDisposed = true;

        instance.Dispose();
        instance = null;
    }
    bool ISingleton.IsDisposed() {
        return this.isDisposed;
    }
    public virtual void Dispose() {
    }
}

```

## 2.12 IDestroy, IDestroySystem, DestroySystem<T>: 销毁系

```
public interface IDestroy {
}
public interface IDestroySystem: ISystemType {
    void Run(Entity o);
}
[ObjectSystem]
public abstract class DestroySystem<T> : IDestroySystem where T: Entity, IDestroy {
    void IDestroySystem.Run(Entity o) {
        this.Destroy((T)o);
    }
    Type ISystemType.SystemType() {
        return typeof(IDestroySystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.None;
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    protected abstract void Destroy(T self);
}
```

## 2.13 IEvent, AEvent<A>: 事件

```
public interface IEvent {
    Type Type { get; }
}
public abstract class AEvent<A>: IEvent where A: struct {
    public Type Type {
        get {
            return typeof(A);
        }
    }
    protected abstract ETask Run(Scene scene, A a);
    public async ETask Handle(Scene scene, A a) {
        try {
            await Run(scene, a);
        }
        catch (Exception e) {
            Log.Error(e);
        }
    }
}
```

## 2.14 IAddComponent: 添加组件系

```
public interface IAddComponent { }
public interface IAddComponentSystem: ISystemType {
    void Run(Entity o, Entity component);
}
[ObjectSystem]
public abstract class AddComponentSystem<T> : IAddComponentSystem where T: Entity, IAddComponent {
    void IAddComponentSystem.Run(Entity o, Entity component) {
        this.AddComponent((T)o, component);
    }
    Type ISystemType.SystemType() {
        return typeof(IAddComponentSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.None;
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    protected abstract void AddComponent(T self, Entity component);
}
```



```

    public Type Type {
        get {
            return typeof (A);
        }
    }
    public abstract T Handle(A a);
}

```

## 2.18 ProtoBuf 相关: IExtensible, IExtension, IProtoOutput<TOutput>, IMeasure

### 看不懂

### 2.18.1 IExtensible

```

// Indicates that the implementing type has support for protocol-buffer
// <see cref="IExtension">extensions</see>.
// <remarks>Can be implemented by deriving from Extensible.</remarks>
public interface IExtensible {
    // Retrieves the <see cref="IExtension">extension</see> object for the current
    // instance, optionally creating it if it does not already exist.
    // <param name="createIfMissing">Should a new extension object be
    // created if it does not already exist?</param>
    // <returns>The extension object if it exists (or was created), or null
    // if the extension object does not exist or is not available.</returns>
    // <remarks>The <c>createIfMissing</c> argument is false during serialization,
    // and true during deserialization upon encountering unexpected fields.</remarks>
    IExtension GetExtensionObject(bool createIfMissing);
}

```

### 2.18.2 IExtension

```

// Provides addition capability for supporting unexpected fields during
// protocol-buffer serialization/deserialization. This allows for loss-less
// round-trip/merge, even when the data is not fully understood.
public interface IExtension {
    // Requests a stream into which any unexpected fields can be persisted.
    // <returns>A new stream suitable for storing data.</returns>
    Stream BeginAppend();
    // Indicates that all unexpected fields have now been stored. The
    // implementing class is responsible for closing the stream. If
    // "commit" is not true the data may be discarded.
    // <param name="stream">The stream originally obtained by BeginAppend.</param>
    // <param name="commit">True if the append operation completed successfully.</param>
    void EndAppend(Stream stream, bool commit);
    // Requests a stream of the unexpected fields previously stored.
    // <returns>A prepared stream of the unexpected fields.</returns>
    Stream BeginQuery();
    // Indicates that all unexpected fields have now been read. The
    // implementing class is responsible for closing the stream.
    // <param name="stream">The stream originally obtained by BeginQuery.</param>
    void EndQuery(Stream stream);
    // Requests the length of the raw binary stream; this is used
    // when serializing sub-entities to indicate the expected size.
    // <returns>The length of the binary stream representing unexpected data.</returns>
    int GetLength();
}
// Provides the ability to remove all existing extension data
public interface IExtensionResettable : IExtension {
    void Reset();
}

```

### 2.18.3 IProtoOutput<TOutput>, IMeasuredProtoOutput<TOutput>, MeasureState<T>:

### 看得头大

```

// Represents the ability to serialize values to an output of type <typeparamref name="TOutput"/>
public interface IProtoOutput<TOutput> {
    // Serialize the provided value
    void Serialize<T>(TOutput destination, T value, object userState = null);
}
// Represents the ability to serialize values to an output of type <typeparamref name="TOutput"/>
// with pre-computation of the length

```

```

public interface IMeasuredProtoOutput<TOutput> : IProtoOutput<TOutput> {
    // Measure the length of a value in advance of serialization
    MeasureState<T> Measure<T>(T value, object userState = null);
    // Serialize the previously measured value
    void Serialize<T>(MeasureState<T> measured, TOutput destination);
}
// Represents the outcome of computing the length of an object; since this may have required computing lengths
// for multiple objects, some metadata is retained so that a subsequent serialize operation using
// this instance can re-use the previously calculated lengths. If the object state changes between the
// measure and serialize operations, the behavior is undefined.
public struct MeasureState<T> : IDisposable {
    // note: * does not actually implement this API;
    // it only advertises it for 3.* capability/feature-testing, i.e.
    // callers can check whether a model implements
    // IMeasuredProtoOutput<Foo>, and *work from that*
    public void Dispose() => throw new NotImplementedException();
    public long Length => throw new NotImplementedException();
}

```

## 3 【拖拉机游戏房间】组件：分析

### 3.1 TractorRoomEvent: 拖拉机房间，【待修改完成】

```

// UI 系统的事件机制：定义，如何创建拖拉机游戏房间【TODO:】UNITY 里是需要制作相应预设的
[UIEvent(UIType.TractorRoom)]
public class TractorRoomEvent: AUIEvent {
    public override async ETask<UI> OnCreate(UIComponent uiComponent, UILayer uiLayer) {
        await ETask.CompletedTask;
        await uiComponent.DomainScene().GetComponent<ResourcesLoaderComponent>().LoadAsync(UIType.TractorRoom.StringToAB())

        GameObject bundleGameObject = (GameObject) ResourcesComponent.Instance.GetAsset(UIType.TractorRoom.StringToAB(), UI
        GameObject room = UnityEngine.Object.Instantiate(bundleGameObject, UIEventComponent.Instance.GetLayer((int)uiLayer)
        UI ui = uiComponent.AddChild<UI, string, GameObject>(UIType.TractorRoom, room);
        // 【拖拉机游戏房间】：它可能由好几个不同的组件组成，这里要添加的不止一个
        ui.AddComponent<GamerComponent>(); // 玩家组件：这个控件带个 UI 小面板，要怎么添加呢？
        ui.AddComponent<TractorRoomComponent>(); // <----- 房间组件：合成组件系统，自带【互动组件】
        return ui;
    }
    public override void OnRemove(UIComponent uiComponent) {
        ResourcesComponent.Instance.UnloadBundle(UIType.TractorRoom.StringToAB());
    }
}

```

### 3.2 GamerComponent: 玩家【管理类组件】，是对房间里四个玩家的管理。

- 【GamerComponent】玩家组件：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。

```

// 组件：是提供给房间用，用来管理游戏中每个房间里的最多三个当前玩家
public class GamerComponent : Entity, IAwake { // 它也有【生成系】
    private readonly Dictionary<long, int> seats = new Dictionary<long, int>();
    private readonly Gamer[] gamers = new Gamer[4];
    public Gamer LocalGamer { get; set; } // 提供给房间组件用的：就是当前玩家...
    // 添加玩家
    public void Add(Gamer gamer, int seatIndex) {
        gamers[seatIndex] = gamer;
        seats[gamer.UserID] = seatIndex;
    }
    // 获取玩家
    public Gamer Get(long id) {
        int seatIndex = GetGamerSeat(id);
        if (seatIndex >= 0)
            return gamers[seatIndex];
        return null;
    }
    // 获取所有玩家
    public Gamer[] GetAll() {
        return gamers;
    }
    // 获取玩家座位索引

```

```

    public int GetGamerSeat(long id) {
        int seatIndex;
        if (seats.TryGetValue(id, out seatIndex))
            return seatIndex;
        return -1;
    }
    // 移除玩家并返回
    public Gamer Remove(long id) {
        int seatIndex = GetGamerSeat(id);
        if (seatIndex >= 0) {
            Gamer gamer = gamers[seatIndex];
            gamers[seatIndex] = null;
            seats.Remove(id);
            return gamer;
        }
        return null;
    }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        this.LocalGamer = null;
        this.seats.Clear();
        for (int i = 0; i < this.gamers.Length; i++)
            if (gamers[i] != null) {
                gamers[i].Dispose();
                gamers[i] = null;
            }
    }
}

```

### 3.3 Gamer: 【服务端】一个玩家个例。对应这个玩家的相关信息

```

// 房间玩家对象
public sealed class Gamer : Entity, IAwake<long> {
    // 用户 ID (唯一)
    public long UserID { get; private set; }
    // 玩家 GateActorID
    public long PlayerID { get; set; }
    // 玩家所在房间 ID
    public long RoomID { get; set; }
    // 是否准备
    public bool IsReady { get; set; }
    // 是否离线
    public bool isOffline { get; set; }
    public void Awake(long id) {
        this.UserID = id;
    }
    public override void Dispose() {
        if (this.IsDisposed) return;
        base.Dispose();
        this.UserID = 0;
        this.PlayerID = 0;
        this.RoomID = 0;
        this.IsReady = false;
        this.isOffline = false;
    }
}

```

### 3.4 Gamer: 【客户端】一个玩家个例。它说只要一点儿信息就行

- 传进程间消息的时候，也只传这两个关键参数。

```

public sealed class Gamer : Entity { // 玩家对象
    // 玩家唯一 ID
    public long UserID { get; set; }
    // 是否准备
    public bool IsReady { get; set; }
    public override void Dispose() {
        if (this.IsDisposed) return;
        base.Dispose();
        this.UserID = 0;
    }
}

```

```

        this.IsReady = false;
    }
}

```

### 3.5 GamerUIComponent: 【客户端】玩家 UI 组件：每个玩家背个小面板，来显示必要信息（钱，抢不抢庄，反过的主等）

```

public class GamerUIComponent : Entity, IStart { // 玩家 UI 组件
    public GameObject Panel { get; private set; } // UI 面板
    // 玩家昵称
    public string NickName { get { return name.text; } }
    private Image headPhoto;
    private Text prompt;
    private Text name;
    private Text money;
    public void Start() {
        if (this.GetParent<Gamer>().IsReady)
            SetReady();
    }
    // 重置面板
    public void ResetPanel() {
        ResetPrompt();
        this.headPhoto.gameObject.SetActive(false);
        this.name.text = "空位";
        this.money.text = "";
        this.Panel = null;
        this.prompt = null;
        this.name = null;
        this.money = null;
        this.headPhoto = null;
    }
    // 设置面板
    public void SetPanel(GameObject panel) {
        this.Panel = panel;
        // 绑定关联
        this.prompt = this.Panel.Get<GameObject>("Prompt").GetComponent<Text>();
        this.name = this.Panel.Get<GameObject>("Name").GetComponent<Text>();
        this.money = this.Panel.Get<GameObject>("Money").GetComponent<Text>();
        this.headPhoto = this.Panel.Get<GameObject>("HeadPhoto").GetComponent<Image>();
        UpdatePanel();
    }
    // 更新面板
    public void UpdatePanel() {
        if (this.Panel != null) {
            SetUserInfo();
            headPhoto.gameObject.SetActive(false);
        }
    }
    // 设置玩家身份
    public void SetIdentity(Identity identity) {
        if (identity == Identity.None) return;
        string spriteName = $"Identity_{Enum.GetName(typeof(Identity), identity)}";
        Sprite headSprite = CardHelper.GetCardSprite(spriteName);
        headPhoto.sprite = headSprite;
        headPhoto.gameObject.SetActive(true);
    }
    // 玩家准备
    public void SetReady() {
        prompt.text = "准备! ";
    }
    // 出牌错误
    public void SetPlayCardsError() {
        prompt.text = "您出的牌不符合规则! ";
    }
    // 玩家不出
    public void SetDiscard() {
        prompt.text = "不出";
    }
    // 打 2 时，玩家抢不抢庄：或者去想，玩家要不要反主牌花色
    public void SetGrab(GrabLandlordState state) {
        switch (state) {
            case GrabLandlordState.Not:
                break;
        }
    }
}

```



```

        case GrabLandlordState.Grab:
            prompt.text = " 抢地主";
            break;
        case GrabLandlordState.UnGrab:
            prompt.text = " 不抢";
            break;
    }
}
public void ResetPrompt() { // 重置提示
    prompt.text = "";
}
public void GameStart() { // 游戏开始
    ResetPrompt();
}
private async void SetUserInfo() { // 设置用户信息
    G2C_GetUserInfo_Ack g2C_GetUserInfo_Ack = await SessionComponent.Instance.Session.Call(new C2G_GetUserInfo_Req() {
        if (this.Panel != null) {
            name.text = g2C_GetUserInfo_Ack.NickName;
            money.text = g2C_GetUserInfo_Ack.Money.ToString();
        }
    }
}
public override void Dispose() {
    if (this.IsDisposed) return;
    base.Dispose();
    ResetPanel(); // 重置玩家 UI
}
}
}

```

### 3.6 Protobuf 里面的消息与参考

- 这里把 Protobuf 里面可以传的游戏相关也整理一下。

```

message GamerInfo {
    int64 UserID = 1;
    bool IsReady = 2;
}
message GamerScore {
    int64 UserID = 1;
    int64 Score = 2;
}
message GamerState {
    int64 UserID = 1;
    ET.Server.Identity UserIdentity = 2; // 命名空间的问题
    ^^IGrabLandlordState State = 3;
}
message GamerCardNum { // IMessage
    int64 UserID = 1;
    int32 Num = 2;
}
message Actor_GamerGrabLandlordSelect_Ntt { // IActorMessage 参考去想: 抢庄, 与反主牌花色, 如何写消息
    int32 RpcId = 90;
    int64 ActorId = 94;
    int64 UserID = 1;
    bool IsGrab = 2;
}

```

### 3.7 TractorRoomComponent: 游戏房间, 自带其它组件, 当有嵌套时, 如何才能系统化地、工厂化地、UI 上的事件驱动地, 生成这个组件呢?

```

public class TractorRoomComponent : Entity, IAwake {
    private TractorInteractionComponent interaction; // 嵌套组件: 互动组件
    private Text multiples;
    public readonly GameObject[] GamersPanel = new GameObject[4];
    public bool Matching { get; set; }
    public TractorInteractionComponent Interaction { // 组件里套组件, 要如何事件机制触发生成?
        get {
            if (interaction == null) {
                UI uiRoom = this.GetParent<UI>();
                UI uiInteraction = TractorInteractionFactory.Create(UIType.TractorInteraction, uiRoom);
                interaction = uiInteraction.GetComponent<TractorInteractionComponent>();
            }
        }
    }
}

```

```

        return interaction;
    }
}

```

### 3.8 TractorInteractionComponent: 感觉是视图 UI 上的一堆调控，逻辑控制

- 上下这两个组件里，除了 ProtoBuf 消息里传递的类找不到，没有其它错误
- 【嵌套】：是这里的难点。其它都可以一个触发一个地由事件发布触发订阅者的回调，可是当一个组件内存在嵌套，又是系统化【内部组件生成完成后，外部组件才生成完成】生成，我是要把这两个组件合并成一个吗？还是说，我不得不把它折成粒度更小的 UI 上的事件驱动机制，以符合系统框架？要去源码弄透。

```

// 【互动组件】：一堆的视图控件管理
public class TractorInteractionComponent : Entity, IAwake { // 多个按钮：有些暂时是隐藏的
    private Button playButton;
    private Button promptButton;
    private Button discardButton;
    private Button grabButton;
    private Button disgrabButton;
    private Button changeGameModeButton;
    private List<Card> currentSelectCards = new List<Card>();

    public bool isTrusteeship { get; set; }
    public bool IsFirst { get; set; }
}

```

## 4 消息处理器：AMActorHandler<E, Message> 继承类的返回类型，全改成了 void

### 4.1 AMActorHandler<E, Message>：基类的抽象方法 Run 的返回类型被固定死了，报了狠多错

- 这样，可以把所有自己继承类的报错去掉。可是因为还没能理解透彻，不知道先前的 ETVoid 是为什么，现在会不会产生什么其它意外的错。作个记号。

```

[EnableClass]
public abstract class AMActorHandler<E, Message>: IMActorHandler where E : Entity where Message : class, IActorMessage {

    // protected abstract ETTask Run(E entity, Message message); // <-----
    protected abstract void Run(E entity, Message message); // 可以改成是自己想要的，返回类型，因为只有自己的继承类在使用

    public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
        if (actorMessage is not Message msg) {
            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;
        }
        await this.Run(e, msg);
    }
    public Type GetRequestType() {
        if (typeof (IActorLocationMessage).IsAssignableFrom(typeof (Message))) {
            Log.Error($"message is IActorLocationMessage but handler is AMActorHandler: {typeof (Message)}");
        }
        return typeof (Message);
    }
    public Type GetResponseType() {
        return null;
    }
}

```

## 4.2 IActorHandler: 接口类的定义，同样要改

```
public interface IActorHandler {
    // ETask Handle(Entity entity, int fromProcess, object actorMessage);
    void Handle(Entity entity, int fromProcess, object actorMessage); // 自己改成这样的
    Type GetRequestType();
    Type GetResponseType();
}
```

## 5 Protobuf 里的 enum: 【Identity】【Suits】【Weight】

### 5.1 OuterMessage\_C\_10001.proto 里三四个类的定义

- 感觉更多的是命名空间没能弄对。同一份源码一式三份，分别放在【客户端】【双端】【服务端】下只有【客户端】下可以通过读 Card 类的定义，可以知道能自动识别，并且 Protobuf 里的 enum 生成的.cs 与参考项目不同。不知道是否是 Protobuf 版本问题，还是我没注意到的细节。

```
enum Identity { // 身份
    IdentityNone = 0;
    Farmer = 1;    // 平民
    Landlord = 2;  // 地主
}
enum Suits { // 花色
    Club = 0;      // 梅花
    Diamond = 1;   // 方块
    Heart = 2;     // 红心
    Spade = 3;     // 黑桃
    None = 4;
}
enum Weight { // 权重
    Three = 0;     // 3
    Four = 1;      // 4
    Five = 2;      // 5
    Six = 3;       // 6
    Seven = 4;     // 7
    Eight = 5;     // 8
    Nine = 6;      // 9
    Ten = 7;       // 10
    Jack = 8;      // J
    Queen = 9;     // Q
    King = 10;     // K
    One = 11;      // A
    Two = 12;      // 2
    SJoker = 13;   // 小王
    LJoker = 14;   // 大王
}
message Card {
    Weight CardWeight = 1;
    Suits CardSuits = 2;
}
```

### 5.2 【参考项目】里: enum 是可以顺利写进 ETModel 申明的命名空间，并且源码可见

```
namespace ETModel {
    #region Enums
    public enum Suits {
        Club = 0,
        Diamond = 1,
        Heart = 2,
        Spade = 3,
        None = 4,
    }
    public enum Weight {
        Three = 0,
        Four = 1,
        Five = 2,
    }
}
```

```

        Six = 3,
        Seven = 4,
        Eight = 5,
        Nine = 6,
        Ten = 7,
        Jack = 8,
        Queen = 9,
        King = 10,
        One = 11,
        Two = 12,
        Sjoker = 13,
        Ljoker = 14,
    }
    public enum Identity {
        None = 0,
        Farmer = 1,
        Landlord = 2,
    }
}
#endregion
#region Messages

```

### 5.3 ET7 框架里，enum 完全找不到

- 一种网络上没能理解透彻的可能是：我不能把三个 enum 类单独列出来，而是把三个类嵌套在必要的需要使用这些 enum 的 message 的定义里，举例如下：
- 如下，对于 Card 类应该是行得通的。可是问题是，我的 card 本来也没有问题。有问题的是，三个 enum 类找不到。那么也就是，我大概还是需要手动定义这三个类在程序的某些域某些地方。【确认一下】

```

message SearchRequest {
    string query = 1;
    int32 page_number = 2;
    enum Corpus { // enum 成员变量一定义嵌套
        UNIVERSAL = 0;
        WEB = 1;
        IMAGES = 2;
        LOCAL = 3;
        NEWS = 4;
        PRODUCTS = 5;
        VIDEO = 6;
    }
    Corpus corpus = 4; // enum 成员变量一定义赋值
}

```

- 觉得这个，是目前最主要的 compile-error 的来源，但不是自己重构项目的重点，还是去看其它的。看如何重构项目。这个晚上再弄。

### 5.4 ETModel\_Card\_Binding: 奇异点，ILRuntime 热更新里，似乎对 Card 类的两个成员变量作了辅助链接

- 还没有细看，不是狠懂这里的原理。但在解决上面的问题之后，如果这两个变量仍不通，会参考这里

```

unsafe class ETModel_Card_Binding {
    public static void Register(ILRuntime.Runtime.Environment.AppDomain app) {
        BindingFlags flag = BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static | BindingFlags.DeclaredOnly;
        MethodBase method;
        Type[] args;
        Type type = typeof(ETModel.Card);
        args = new Type[]{};
        method = type.GetMethod("GetName", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, GetName_0);
        args = new Type[]{};
        method = type.GetMethod("get_CardWeight", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_CardWeight_1);
        args = new Type[]{};
    }
}

```

```

        method = type.GetMethod("get_CardSuits", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_CardSuits_2);
        args = new Type[]{};
        method = type.GetMethod("get_Parser", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_Parser_3);
    }

```

## 6 ET7 数据库相关【服务端】

- 这个数据库系统，连个添加使用的范例也没有。。。就两个组件，一个管理类。什么也没留下。。
- 这里不急着重整理。现框架 **DB 放在服务端的 Model** 里。它的管理体系成为管理各个不同区服的数据库 DBComponent。
- 因为找不到任何参考使用的例子。我觉得需要搜索一下。在理解了参考项目数据库模块之后，根据搜索，决定是使用原参考项目总服务器代理系，还是这种相对改装了的管理区服系统？

### 6.1 IDBCollection: 主要是方便写两个不同的数据库（好像是 GeekServer 里两个数据库）。反正方便扩展吧

```
public interface IDBCollection {}
```

### 6.2 DBComponent: 带生成系。可以查表，查询数据

- 它的生成系就是解决对数据库的 CRUD 必要操作，单条信息的，或是批量处理的

```

[ChildOf(typeof(DBManagerComponent))] // 用来缓存数据
public class DBComponent: Entity, IAwake<string, string, int>, IDestroy {
    public const int TaskCount = 32;
    public MongoClient mongoClient;
    public IMongoDatabase database;
}

```

### 6.3 DBManagerComponent: 有上面的 DBComponent 数组。数组长度固定吗？

```

public class DBManagerComponent: Entity, IAwake, IDestroy {
    [StaticField]
    public static DBManagerComponent Instance;
    public DBComponent[] DBComponents = new DBComponent[IdGenerater.MaxZone]; // 没事吃饱了撑得，占一大堆空地
}

```

### 6.4 DBManagerComponentSystem: 主是要查询某个区服的数据库，从数组里

```

[FriendOf(typeof(DBManagerComponent))]
public static class DBManagerComponentSystem {
    [ObjectSystem]
    public class DBManagerComponentAwakeSystem: AwakeSystem<DBManagerComponent> {
        protected override void Awake(DBManagerComponent self) {
            DBManagerComponent.Instance = self;
        }
    }
    [ObjectSystem]
    public class DBManagerComponentDestroySystem: DestroySystem<DBManagerComponent> {
        protected override void Destroy(DBManagerComponent self) {
            DBManagerComponent.Instance = null;
        }
    }
    public static DBComponent GetZoneDB(this DBManagerComponent self, int zone) {
        DBComponent dbComponent = self.DBComponents[zone];
        if (dbComponent != null)

```

```

        return dbComponent;
        StartZoneConfig startZoneConfig = StartZoneConfigCategory.Instance.Get(zone);
        if (startZoneConfig.DBConnection == "")
            throw new Exception($"zone: {zone} not found mongo connect string");
        dbComponent = self.AddChild<DBComponent, string, string, int>(startZoneConfig.DBConnection, startZoneConfig.DBName,
        self.DBComponents[zone] = dbComponent;
        return dbComponent;
    }
}

```

## 6.5 DBProxyComponent: 【参考项目】里的。有生成系。

```

// 用来与数据库操作代理
public class DBProxyComponent: Component {
    public IPEndPoint dbAddress;
}

```

## 7 StartConfigComponent: 找【各种服】的起始初始化地址

- 这些组群服务器的起始被全部重构了，重构成配置单例了

### 7.1 ConfigSingleton<T>: ProtoObject, ISingleton

```

public abstract class ConfigSingleton<T>: ProtoObject, ISingleton where T: ConfigSingleton<T>, new() {
    [StaticField]
    private static T instance;
    public static T Instance {
        get {
            return instance ??= ConfigComponent.Instance.LoadOneConfig(typeof (T)) as T;
        }
    }
    void ISingleton.Register() {
        if (instance != null) {
            throw new Exception($"singleton register twice! {typeof (T).Name}");
        }
        instance = (T)this;
    }
    void ISingleton.Destroy() {
        T t = instance;
        instance = null;
        t.Dispose();
    }
    bool ISingleton.IsDisposed() {
        throw new NotImplementedException();
    }
    public override void AfterEndInit() { }
    public virtual void Dispose() { }
}

```

### 7.2 StartProcessConfigCategory: ConfigSingleton<StartProcessConfigCategory>, IMerge

- 当数据库集群成区服的形式，这里各服务器的配置，成了 ProtoBuf 里进程间可传的消息模式?。。。。
- 这里配置是从哪里来的呢？仍然是从各种配置文件里

```

[ProtoContract]
[Config]
public partial class StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge {
    [ProtoIgnore]
    [BsonIgnore]
    private Dictionary<int, StartProcessConfig> dict = new Dictionary<int, StartProcessConfig>(); // 管理字典
    [BsonElement]
    [ProtoMember(1)]
    private List<StartProcessConfig> list = new List<StartProcessConfig>();
}

```



```

        break;
//...
    }
    return scene;
}
}

```

## 7.4 RouterAddressComponent: 路由器组件

```

[ComponentOf(typeof(Scene))]
public class RouterAddressComponent: Entity, IAwake<string, int> {
    public IPAddress RouterManagerIPAddress { get; set; }
    public string RouterManagerHost;
    public int RouterManagerPort;
    public HttpGetRouterResponse Info;
    public int RouterIndex;
}

```

## 7.5 RouterAddressComponentSystem: 路由器的生成系

```

[FriendOf(typeof(RouterAddressComponent))]
public static class RouterAddressComponentSystem {
    public class RouterAddressComponentAwakeSystem: AwakeSystem<RouterAddressComponent, string, int> {
        protected override void Awake(RouterAddressComponent self, string address, int port) {
            self.RouterManagerHost = address;
            self.RouterManagerPort = port;
        }
    }

    public static async ETask Init(this RouterAddressComponent self) {
        self.RouterManagerIPAddress = NetworkHelper.GetHostAddress(self.RouterManagerHost);
        await self.GetAllRouter();
    }

    private static async ETask GetAllRouter(this RouterAddressComponent self) {
        string url = $"http:// {self.RouterManagerHost}:{self.RouterManagerPort}/get_router?v={RandomGenerator.RandUInt32()}";
        Log.Debug($"start get router info: {url}");
        string routerInfo = await HttpClientHelper.Get(url);
        Log.Debug($"recv router info: {routerInfo}");
        HttpGetRouterResponse httpGetRouterResponse = JsonHelper.FromJson<HttpGetRouterResponse>(routerInfo);
        self.Info = httpGetRouterResponse;
        Log.Debug($"start get router info finish: {JsonHelper.ToJson(httpGetRouterResponse)}");
        // 打乱顺序
        RandomGenerator.BreakRank(self.Info.Routers);
        self.WaitTenMinGetAllRouter().Coroutine();
    }

    // 等 10 分钟再获取一次
    public static async ETask WaitTenMinGetAllRouter(this RouterAddressComponent self) {
        await TimerComponent.Instance.WaitAsync(5 * 60 * 1000);
        if (self.IsDisposed)
            return;
        await self.GetAllRouter();
    }

    public static IPEndPoint GetAddress(this RouterAddressComponent self) {
        if (self.Info.Routers.Count == 0)
            return null;
        string address = self.Info.Routers[self.RouterIndex++ % self.Info.Routers.Count];
        string[] ss = address.Split(':');
        IPAddress ipAddress = IPAddress.Parse(ss[0]);
        if (self.RouterManagerIPAddress.AddressFamily == AddressFamily.InterNetworkV6) {
            ipAddress = ipAddress.MapToIPv6();
        }
        return new IPEndPoint(ipAddress, int.Parse(ss[1]));
    }

    public static IPEndPoint GetRealmAddress(this RouterAddressComponent self, string account) { // <===== 照
        int v = account.Mode(self.Info.Realms.Count);
        string address = self.Info.Realms[v];
        string[] ss = address.Split(':');
        IPAddress ipAddress = IPAddress.Parse(ss[0]);
        // if (self.IPAddress.AddressFamily == AddressFamily.InterNetworkV6)
        //     ipAddress = ipAddress.MapToIPv6();
        return new IPEndPoint(ipAddress, int.Parse(ss[1]));
    }
}

```



## 7.6 RouterHelper: 路由器帮助类，向路由器注册、申请？

```
public static class RouterHelper {
    // 注册 router
    public static async ETask<Session> CreateRouterSession(Scene clientScene, IPEndPoint address) {
        (uint recvLocalConn, IPEndPoint routerAddress) = await GetRouterAddress(clientScene, address, 0, 0);
        if (recvLocalConn == 0)
            throw new Exception($"get router fail: {clientScene.Id} {address}");
        Log.Info($"get router: {recvLocalConn} {routerAddress}");
        Session routerSession = clientScene.GetComponent<NetClientComponent>().Create(routerAddress, address, recvLocalConn);
        routerSession.AddComponent<PingComponent>();
        routerSession.AddComponent<RouterCheckComponent>();
        return routerSession;
    }

    public static async ETask<(uint, IPEndPoint)> GetRouterAddress(Scene clientScene, IPEndPoint address, uint localConn,
        Log.Info($"start get router address: {clientScene.Id} {address} {localConn} {remoteConn}");
        // return (RandomHelper.RandUInt32(), address);
        RouterAddressComponent routerAddressComponent = clientScene.GetComponent<RouterAddressComponent>();
        IPEndPoint routerInfo = routerAddressComponent.GetAddress();
        uint recvLocalConn = await Connect(routerInfo, address, localConn, remoteConn);
        Log.Info($"finish get router address: {clientScene.Id} {address} {localConn} {remoteConn} {recvLocalConn} {routerInfo}");
        return (recvLocalConn, routerInfo);
    }

    // 向 router 申请
    private static async ETask<uint> Connect(IPEndPoint routerAddress, IPEndPoint realAddress, uint localConn, uint remoteConn) {
        uint connectId = RandomGenerator.RandUInt32();
        using Socket socket = new Socket(routerAddress.AddressFamily, SocketType.Dgram, ProtocolType.Udp);
        int count = 20;
        byte[] sendCache = new byte[512];
        byte[] recvCache = new byte[512];
        uint synFlag = localConn == 0 ? KcpProtocalType.RouterSYN : KcpProtocalType.RouterReconnectSYN;
        sendCache.WriteTo(0, synFlag);
        sendCache.WriteTo(1, localConn);
        sendCache.WriteTo(5, remoteConn);
        sendCache.WriteTo(9, connectId);
        byte[] addressBytes = realAddress.ToString().ToByteArray();
        Array.Copy(addressBytes, 0, sendCache, 13, addressBytes.Length);
        Log.Info($"router connect: {connectId} {localConn} {remoteConn} {routerAddress} {realAddress}");

        IPEndPoint recvIPEndPoint = new IPEndPoint(IPAddress.Any, 0);
        long lastSendTimer = 0;
        while (true) {
            long timeNow = TimeHelper.ClientFrameTime();
            if (timeNow - lastSendTimer > 300) {
                if (--count < 0) {
                    Log.Error($"router connect timeout fail! {localConn} {remoteConn} {routerAddress} {realAddress}");
                    return 0;
                }
                lastSendTimer = timeNow;
                // 发送
                socket.SendTo(sendCache, 0, addressBytes.Length + 13, SocketFlags.None, routerAddress);
            }
            await TimerComponent.Instance.WaitFrameAsync();
            // 接收
            if (socket.Available > 0) {
                int messageLength = socket.ReceiveFrom(recvCache, ref recvIPEndPoint);
                if (messageLength != 9) {
                    Log.Error($"router connect error1: {connectId} {messageLength} {localConn} {remoteConn} {routerAddress}");
                    continue;
                }
                byte flag = recvCache[0];
                if (flag != KcpProtocalType.RouterReconnectACK && flag != KcpProtocalType.RouterACK) {
                    Log.Error($"router connect error2: {connectId} {synFlag} {flag} {localConn} {remoteConn} {routerAddress}");
                    continue;
                }
                uint recvRemoteConn = BitConverter.ToUInt32(recvCache, 1);
                uint recvLocalConn = BitConverter.ToUInt32(recvCache, 5);
                Log.Info($"router connect finish: {connectId} {recvRemoteConn} {recvLocalConn} {localConn} {remoteConn} {routerAddress}");
                return recvLocalConn;
            }
        }
    }
}
```



## 8.2 框架 Game 类：是单例的管理类，与服务端或是客户端的总、根场景无关

```
public static class Game { // 框架的 Game 类
    [StaticField]
    private static readonly Dictionary<Type, ISingleton> singletonTypes = new Dictionary<Type, ISingleton>();
    [StaticField]
    private static readonly Stack<ISingleton> singletons = new Stack<ISingleton>();
    [StaticField]
    private static readonly Queue<ISingleton> updates = new Queue<ISingleton>();
    [StaticField]
    private static readonly Queue<ISingleton> lateUpdates = new Queue<ISingleton>();
    [StaticField]
    private static readonly Queue<ETTask> frameFinishTask = new Queue<ETTask>();
    public static T AddSingleton<T>() where T: Singleton<T>, new() {
        T singleton = new T();
        AddSingleton(singleton);
        return singleton;
    }
    public static void AddSingleton(ISingleton singleton) { // 对单例的生命周期进行回调
        Type singletonType = singleton.GetType();
        if (singletonTypes.ContainsKey(singletonType))
            throw new Exception($"already exist singleton: {singletonType.Name}");
        singletonTypes.Add(singletonType, singleton);
        singletons.Push(singleton);
        singleton.Register();
        if (singleton is ISingletonAwake awake)
            awake.Awake();
        if (singleton is ISingletonUpdate)
            updates.Enqueue(singleton);
        if (singleton is ISingletonLateUpdate)
            lateUpdates.Enqueue(singleton);
    }
    public static async ETTask WaitFrameFinish() {
        ETTask task = ETTask.Create(true);
        frameFinishTask.Enqueue(task);
        await task;
    }
    public static void Update() {
        int count = updates.Count;
        while (count-- > 0) {
            ISingleton singleton = updates.Dequeue();
            if (singleton.IsDisposed())
                continue;
            if (singleton is not ISingletonUpdate update)
                continue;
            updates.Enqueue(singleton);
            try {
                update.Update();
            }
            catch (Exception e) {
                Log.Error(e);
            }
        }
    }
    public static void LateUpdate() {
        int count = lateUpdates.Count;
        while (count-- > 0) {
            ISingleton singleton = lateUpdates.Dequeue();
            if (singleton.IsDisposed())
                continue;
            if (singleton is not ISingletonLateUpdate lateUpdate)
                continue;
            lateUpdates.Enqueue(singleton);
            try {
                lateUpdate.LateUpdate();
            }
            catch (Exception e) {
                Log.Error(e);
            }
        }
    }
    public static void FrameFinishUpdate() {
        while (frameFinishTask.Count > 0) {
            ETTask task = frameFinishTask.Dequeue();
            task.SetResult();
        }
    }
}
```

```

    }
}
public static void Close() { // 顺序反过来清理
    while (singletons.Count > 0) {
        ISingleton iSingleton = singletons.Pop();
        iSingleton.Destroy();
    }
    singletonTypes.Clear();
}
}
}

```

### 8.3 ET7 的重构，将数据库相关全部去掉了？找不到数据库的踪影？

- 扔进什么狗屁的 AI 相关里去了。不用管，可以添加自己需要用到的

### 8.4 GamerFactory: 【加工厂】全部移除掉

- 工厂的逻辑，重构以后，全部放进了 AUIEvent 的实例继承类里。全部移除掉
- 有个 Factory 的文件夹，是会全部移除掉的

```

public static class GamerFactory {
    // 创建玩家对象
    public static Gamer Create(long playerId, long userId, long? id = null) {
        Gamer gamer = ComponentFactory.CreateWithId<Gamer, long>(id ?? IdGenerator.GenerateId(), userId);
        gamer.PlayerID = playerId;
        return gamer;
    }
}

```

## 9 写在最后：反而是自己每天查看一再更新的

- 因为感觉还是不曾系统性地读 ET7 的源码，或者说有效阅读，因为没有带着实际问题的看源码，感觉都不叫看读源码呀。这里会记自己的感觉需要赶快查看的地方。
- **【ET 框架的整体架构】**：感觉把握不够。常常命名空间分不清。要把这个大的框架，比较高层面的架构再好好看下。然后就是对自顶向下的不同层级场景，所需要的主要的不同组件，分不清，仍需要再熟悉一下源码
- **【问题】**：某些消息，还分不清是内网还是外网消息，暂时先放一下，到时再改
- **【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？
- **【ClientComponent】**：新框架里重构丢了，去找怎么替代？那么现在去追一下，客户端的起始与场景加载或是切换大致过程。它变成了什么客户端场景管理？
- **【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

## 10 现在的修改内容，记忆

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

## 11 TODO

- **Windows 下 org-mode 有几个【BUG:】** 1.org-mode 不能自动识别模式，除第一次加载可以正确，其它再加载不识别 org-mode; 2.org-export-to-pdf 在我换成为 msys64 里的 emacs 后就坏掉了。因为要花时间修，暂时还放着
- **【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】**
- **【Protobuf 里进程间传递的游戏数据相关信息:】** 这个现在成为重构的主要 compile-error. 因为找不到类。需要去弄懂
  - **【Proto2CS】**：进程间消息里的，**【牌相关的】**，尤其是它们所属的命名空间，没写对，现在总是找不到定义。
  - 包括 Identity, Weight, Suits, 抢不抢地主 **【抢不抢庄】**，以及可能的反不反主牌花色等。
  - 找不到的那些类，感觉更多的是命名空间没能开对。同一份源码一式三份，分别放在 **【客户端】** **【双端】** **【服务端】** 下只有 **【客户端】** 下可以自动识别，并且 Protobuf 里的 enum 生成的 .cs 与参考项目不同。不知道是否是 Protobuf 版本问题，还是我没注意到的细节。
  - **【Identity】与【Suits/Weight】三个【enum】**：外网消息里，怎么会找不到呢？再回去检查一遍。下午要把这个弄通，要开始思路怎么设计重构拖拉机项目。
- **Match【匹配服】**：不知道我哪根筋搭错，以为没有匹配服。可是它的配置。。。再一次从服务端看一遍起始源码，把匹配服的地址加载与获取找出来。。
- 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
- 把参考游戏里，打牌相关的逻辑与模块好好看下，方便自己熟悉自己重构项目的源码后，画葫芦画瓢地重构
- **【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!!!】**

## 12 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，就知道该如何狠好地设计这些类。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。
- **【GamerComponent】** 玩家组件：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。
- **【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!!!】**