# ET 框架学习笔记（二）- - 网络交互相关【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!!!】

deepwaterooo

June 26, 2023

## Contents

# 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，已经申明了公用方法，否则用下面的
Session matchSession = NetInnerComponentSystem.Get(matchIPEndPoint);
// 下面再添加自己新改的方法，用作自己修改后面的参考：
// Room room = Root.Instance.Scene.GetComponent<RoomComponent>().Get(gamer.RoomID);
Room room = RoomComponentSystem.Get(Root.Instance.Scene.GetComponent<RoomComponent>(), gamer.RoomID);
// 现在会改热更域里的静态方法的调用了，就可以再消掉一大堆的编译错误了。。。
```

## 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, Service
                    break;
                }
                case NetworkProtocol.KCP: {
                    self.ServiceId = NetServices.Instance.AddService(new KService(AddressFamily.InterNetwork, Service
                    break;
                }
            }
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead);
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError);
        }
    }
    [ObjectSystem]
    public class NetInnerComponentAwake1System: 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
    }
    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.checkInteral, NetThrea
    }
    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.checkInteral, NetThrea
        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，
    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 = mess
    }
}
```

## 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 = mess
    }
    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
        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 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;
    }
}
```

## 1.8 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);
            }
        }
    }
}
```

## 1.9 【不记得之前这里写的是什么了。。】

- 前面大数是找的，框架中存在的网络处理相关的逻辑。
- 现在就带问题：现存的编译错误，当客户端需要发送消息，或是某个服需要发送内网消息时，可以如何把消息把出去?
- 带现存的编译问题，把这个发送消息相关的逻辑，分外网消息（自客户端发送）与内网消息（自某个服发送），弄明白。
- 【客户端】：可以通过客户端场景 ClientScene.GetComponent<SessionComponent>().Send() 的会话框，来发送消息。例子狠多。下面每一行都是一个实际使用的例子，分不同的控件？组

件？去拿 SessionComponent 的会话框。

```
clientScene.GetComponent<SessionComponent>().Session.Send(new C2M_Stop());
unit.ClientScene().GetComponent<SessionComponent>().Session.Send(msg);
```

- 【服务端】：MessageHelper 类，好像可以帮助发送不少消息到客户端，到其它服吗？这个发送消息的问题，可能就会连带 Actor 相关模块，没弄懂的 Rpc 进程间通信一起弄明白。

### 1.10 MessageHelper: 不知道这个类是作什么用的，使用场景等。过会儿看下

### 1.11 ActorHandleHelper: 是谁调用它，什么场景下使用的？

## 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？
    // ETTask 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。

- **【编译报错:】**热更新程序域里面，只能申明含有 BaseAttribute 的子类特性的类或静态类。那么也就是说，我上面的，我哪怕是把同名文件复制到热更新程序域，也是不对的，因为框架不允许这么做。我就必须去找前面，模仿它的框架系统扩展的这个方法，哪里没能连通好，为什么它的系统方法只存在 Model 域，就能运行好，而我添加的不可以?

## 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 ETTask Run(Scene scene, A a);
    public async ETTask 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);
}
```

## 2.15 IGetComponent: 获取组件系。【这里没有看明白】：再去找细节 // «««««««««««

```
// GetComponentSystem 有巨大作用，比如每次保存 Unit 的数据不需要所有组件都保存，只需要保存 Unit 变化过的组件
// 是否变化可以通过判断该组件是否 GetComponent，Get 了就记录该组件【这里没有看明白】：再去找细节  // <<<<<<<<<<<<<<<<<<<<
// 这样可以只保存 Unit 变化过的组件
// 再比如传送也可以做此类优化
public interface IGetComponent {
}
public interface IGetComponentSystem: ISystemType {
    void Run(Entity o, Entity component);
}
[ObjectSystem]
public abstract class GetComponentSystem<T> : IGetComponentSystem where T: Entity, IGetComponent {
    void IGetComponentSystem.Run(Entity o, Entity component) {
        this.GetComponent((T)o, component);
    }
    Type ISystemType.SystemType() {
        return typeof(IGetComponentSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.None;
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    protected abstract void GetComponent(T self, Entity component);
}
```

## 2.16 ISerializeToEntity,IDeserialize,IDeserializeSystem,DeserializeSystem<T 序列化，反序列化

```
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);
}
```

## 2.17 IInvoke,AInvokeHandler<A>,AInvokeHandler<A, T>: 激活类

- 这个以前没有细看。现在修改编译错误的过程中，框架里有狠多细节的地方，需要修改的编译错误会一再崩出来，框架里出有狠多，有计时器来触发必要的超时等。所以今天，就把这类自带计时器，自动超时检测的激活系，这个功能模块理解一下。【半懂，大半懂，需要再多看几遍】

- 在以前理解了诸多标签，比如【ComponentOf(typeof())] 事件机制等，但是这个自动的激活系，一般与计时器联接紧密，要把这块儿理解透彻。

```
public interface IInvoke {
    Type Type { get; }
}
public abstract class AInvokeHandler<A>: IInvoke where A: struct {
    public Type Type {
        get {
            return typeof (A);
        }
    }
    public abstract void Handle(A a);
}
public abstract class AInvokeHandler<A, T>: IInvoke where A: struct {
    public Type Type {
        get {
            return typeof (A);
        }
    }
    public abstract T Handle(A a);
}
```

## 2.18 TimerInvokeType: 计时器可以自动触发的类型分类。

- 框架里有很多标签自动标记的标记系统。

- 这里类似。说，申明定义了这如下几类可以计时器自动触发的类型；当某个组件标记了可以计时器自动激活的标签，那么它申明的时间到，就会自动激活：某些某个特定的激活方法与逻辑，

- 如同 7/1/2023, 如果活宝妹还没能嫁给亲爱的表哥，活宝妹就解决活宝妹在亲爱的表哥的身边的小镇上的住宿问题一样，有计时器到 6/30/2023. 有激活：7/1/2023 开始找和买长期住处。希望可以一个月内解决问题，7/31/2023 可以搬进去入住。再也不想跟任何的国际贱鸡掺合，把人烦死了。。。。。

```
[UniqueId(100, 10000)]
public static class TimerInvokeType {
    // 框架层 100-200，逻辑层的 timer type 从 200 起
    public const int WaitTimer = 100;
    public const int SessionIdleChecker = 101;
    public const int ActorLocationSenderChecker = 102;
    public const int ActorMessageSenderChecker = 103;
    // 框架层 100-200，逻辑层的 timer type 200-300
    public const int MoveTimer = 201;
    public const int AITimer = 202;
    public const int SessionAcceptTimeout = 203;
}
```

## 2.19 struct TimerCallback:

```
// 计时器：所涉及的方方面面
public enum TimerClass { // 类型：
    None,       // 无
    OnceTimer, // 一次性
    OnceWaitTimer,  // 一次性要等待的计时器
    RepeatedTimer,  // 重复性、周期性计时器
}
public class TimerAction {
    public static TimerAction Create(long id, TimerClass timerClass, long startTime, long time, int type, object obj) {
        TimerAction timerAction = ObjectPool.Instance.Fetch<TimerAction>();
        timerAction.Id = id;
        timerAction.TimerClass = timerClass;
        timerAction.StartTime = startTime;
        timerAction.Object = obj;
        timerAction.Time = time;
        timerAction.Type = type;
        return timerAction;
    }
    public long Id;
    public TimerClass TimerClass;
    public object Object;
    public long StartTime;
    public long Time;
    public int Type;
    public void Recycle() {
        this.Id = 0;
        this.Object = null;
        this.StartTime = 0;
        this.Time = 0;
        this.TimerClass = TimerClass.None;
        this.Type = 0;
        ObjectPool.Instance.Recycle(this);
    }
}
public struct TimerCallback { // 在标签系中会用到计时器的回调
    public object Args;
}
```

## 2.20 ATimer<T>: AInvokeHandler<TimerCallback>: 抽象类

```
public abstract class ATimer<T>: AInvokeHandler<TimerCallback> where T: class {
    public override void Handle(TimerCallback a) {
        this.Run(a.Args as T);
    }
    protected abstract void Run(T t);
}
```

## 2.21 InvokeAttribute: BaseAttribute, 【Invoke(type)】标签属性

- 这里仍然还没连通：先前只是定义了几个可以计时器定时到时激活的类型；这里只是属性标明激活类型
- 类型的幕后：怎么通过不同的类型，来区分不同长短的计时时间，并在特定的激活时间点，激活的？

- 不同超时类型的超时时长：举个例子：ActorMessageSenderComponent
  - ActorMessageSenderComponent: 这个组件里有个计时器自动计时的超时时段、特定超时类型的超时时长成员变量
  - 超时时间：这个组件有计时器自动计时和超时激活的逻辑，这里定义了这个组件类型的超时时长，在 ActorMessageSenderComponentSystem.cs 文件的**【Invoke(TimerInvokeType.A**标注的 ActorMessageSenderChecker 里会用到，检测超时与否

```
public class InvokeAttribute: BaseAttribute {
    public int Type { get; }
    public InvokeAttribute(int type = 0) {
        this.Type = type;
    }
}
```

## 2.22 ActorMessageSenderComponentSystem::ActorMessageSenderChecker 类中类，计时器自动计时标签激活系【诲涩难懂，多看几遍】

- 上面只是计时器的类型。不同类型内部自带计时器超时的特定类型所规定的超时时间。类型的内部自定义超时处理逻辑。用激活标签标明计时器超时的类型，以便与超时时长，和超时后的处理逻辑一一对应。【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!!!】
- 再找一个激活标签的实体类，作参考，把流程理解透彻。
- 【例子：计时器计时超时消息过滤器过滤超时消息原理】：过滤器里，一旦有某个消息超时，就会自动触发检测：是否有一批消息超时，检测到第一个不超时的，就退出循环检测；把所有超时的消息，一一返回超时错误码给消息发送者，提醒它们出错，必要时它们可以重发。。。
- 【还没连通的地方是:】写好错误码的返回消息，结果写到了 ETTask 异步任务的异常里，错误码抛出异常，ETTask 会同步异常、写入异常、并抛出异常。
  - 又想到一点，ActorMessageSender, 既可以是发送消息者发送消息的发送器，也可以是，错误码返回消息的发送器。那么就是说，ActorMessageSenderComponent 的循环逻辑某处，是可以发返回消息的。【上面想的不对。**在框架的相对上层，当内网 NetInnerComponent 读到消息，发布读到消息事件，会自动触发读到消息事件的订阅者——NetInnerComponentOnReadEvent 来，借助消息处理器帮助类 ActorHandleHelper 类，对不同类型的消息进行分发处理。而帮助类的内部，就是调用这里的底层方法定义。帮助类应该可以更好地区分消息处理的逻辑流程先后顺序。**】
  - 发送消息超时异常，不走发返回消息路径，而是直接由 ETTask 抛异常，不需要发返回消息。Run() 方法被其它情境下调用（被读到消息事件的订阅者，借助消息处理器帮助类，来调用这里的底层方法，处理正常的返回消息），才会发返回消息，系统的后半部分，有发送消息的逻辑。今天上午把这块读懂，下午回去改这块儿的重构与编译错误。
- 亲爱的表哥，感觉你活宝妹努力认真去读懂一个艰深诲涩难懂的模块或是功能逻辑的时候，活宝妹的小鼠标，还是会偶尔落到不小心落到永远不想去落的位置。敬请他们大可不必发疯犯贱，把人都烦死了。活宝妹永远只问：活宝妹嫁给亲爱的表哥了吗？活宝妹被他们的国际贱鸡折磨致死了吗？都还没有，他们就大可不必发疯犯贱。任何时候，亲爱的表哥的活宝妹，就是都是一定要嫁给亲爱的表哥的!!! 爱表哥，爱生活!!!

```
[FriendOf(typeof(ActorMessageSenderComponent))]
public static class ActorMessageSenderComponentSystem {
    // 它自带个计时器，就是说，当服务器繁忙处理不过来，它就极有可能会自动超时，若是超时了，就返回个超时消息回去发送者告知一下，必要时
    [Invoke(TimerInvokeType.ActorMessageSenderChecker)] // 另一个新标签，激活系：它标记说，这个激活系类，是 XXX 类型；紧跟着，就
    public class ActorMessageSenderChecker: ATimer<ActorMessageSenderComponent> {
        protected override void Run(ActorMessageSenderComponent self) { // 申明方法的接口是：ATimer<T> 抽象实现类，它实现了 AI
            try {
```

```
                self.Check(); // 调用组件自己的方法
            } catch (Exception e) {
                Log.Error($"move timer error: {self.Id}\n{e}");
            }
        }
    }//...
// Run() 方法：通过同步异常到 ETTask，通过 ETTask 封装的抛异常方式抛出两类异常并返回；和对正常非异常返回消息，同步结果到 ETTask，ET
// 传进来的参数：是一个 IActorResponse 实例，是有最小预处理（初始化了最基本成员变量：异常类型）、【写了个半好】的结果（异常）。结果还没
    private static void Run(ActorMessageSender self, IActorResponse response) {
        // 对于每个超时了的消息：超时错误码都是：ErrorCore.ERR_ActorTimeout，所以会从发送消息超时异常里抛出异常，不用发送错误码【消
        if (response.Error == ErrorCore.ERR_ActorTimeout) { // 写：发送消息超时异常。因为同步到异步任务 ETTask 里，所以异步任务
            self.Tcs.SetException(new Exception($"Rpc error: request，注意 Actor 消息超时，请注意查看是否死锁或者没有 reply: a
            return;
        }
// 这个 Run() 方法，并不是只有 Check() 【发送消息超时异常】一个方法调用。什么情况下的调用，会走到下面的分支？文件尾，有正常消息同步给
// ActorMessageSenderComponent 一个组件，一次只执行一个（返回）消息发送任务，成员变量永远只管当前任务，
// 也是因为 Actor 机制是并行的，一个使者一次只能发一个消息 ...
// 【组件管理器的执行频率， Run() 方法的调用频率】：要是消息太多，发不完怎么办呢？去搜索下面调用 Run() 方法的正常结果消息的调用处理频
        if (self.NeedException && ErrorCore.IsRpcNeedThrowException(response.Error)) { // 若是有异常（判断条件：消息要抛异常否
            self.Tcs.SetException(new Exception($"Rpc error: actorId: {self.ActorId} request: {self.Request}, response: {re
            return;
        }
        self.Tcs.SetResult(response); // 【写结果】：将【写了个半好】的消息，写进同步到异步任务的结果里；把异步任务的状态设置为完成
        // 上面【异步任务 ETTask.SetResult()】，会调用注册过的一个回调，所以 ETTask 封装，设置结果这一步，会自动触发调用注册过的一个
        // ETTask.SetResult() 异步任务写结果了，非空回调是会调用。非空回调是什么，是把返回消息发回去吗？不是。因为有独立的发送逻辑。
        // 再去想 IMHandler：它是消息处理器。问题就变成是，当返回消息写好了，写好了一个完整的可以发送、待发送的消息，谁来处理的？有
        // 这个服，这个自带计时器减压装配装置自带的消息处理器逻辑会处理？不是这个。减压装置，有发送消息超时，只触发最小检测，并抛发送
    }
    private static void Check(this ActorMessageSenderComponent self) {
        long timeNow = TimeHelper.ServerNow();
        foreach ((int key, ActorMessageSender value) in self.requestCallback) {
            // 因为是顺序发送的，所以，检测到第一个不超时的就退出
            // 超时触发的激活逻辑：是有至少一个超时的消息，才会【激活触发检测】；而检测到第一个不超时的，就退出下面的循环。
            if (timeNow < value.CreateTime + ActorMessageSenderComponent.TIMEOUT_TIME)
                break;
            self.TimeoutActorMessageSenders.Add(key);
        }
// 超时触发的激活逻辑：是有至少一个超时的消息，才会【激活触发检测】；而检测到第一个不超时的，就退出上面的循环。
// 检测到第一个不超时的，理论上说，一旦有一个超时消息就会触发超时检测，但实际使用上，可能存在当检测逻辑被触发走到这里，实际中存在两个
        foreach (int rpcId in self.TimeoutActorMessageSenders) { // 一一遍历【超时了的消息】：
            ActorMessageSender actorMessageSender = self.requestCallback[rpcId];
            self.requestCallback.Remove(rpcId);
            try { // ActorHelper.CreateResponse() 框架系统性的封装：也是通过对消息的发送类型与对应的回复类型的管理，使用帮助类，自
                // 对于每个超时了的消息：超时错误码都是：ErrorCore.ERR_ActorTimeout. 也就是，是个异常消息的回复消息实例生成帮助类
                IActorResponse response = ActorHelper.CreateResponse(actorMessageSender.Request, ErrorCore.ERR_ActorTimeout
                Run(actorMessageSender, response); // 猜测：方法逻辑是，把回复消息发送给对应的接收消息的 rpcId
            } catch (Exception e) {
                Log.Error(e.ToString());
            }
        }
        self.TimeoutActorMessageSenders.Clear();
    }

    public static void Send(this ActorMessageSenderComponent self, long actorId, IMessage message) { // 发消息：这个方法，发
        if (actorId == 0)
            throw new Exception($"actor id is 0: {message}");
        ProcessActorId processActorId = new(actorId);
        // 这里做了优化，如果发向同一个进程，则直接处理，不需要通过网络层
        if (processActorId.Process == Options.Instance.Process) { // 没看懂：这里怎么就说，消息是发向同一进程的了？
            NetInnerComponent.Instance.HandleMessage(actorId, message); // 原理清楚：本进程消息，直接交由本进程内网组件处理
            return;
        }
        Session session = NetInnerComponent.Instance.Get(processActorId.Process); // 非本进程消息，去走网络层
        session.Send(processActorId.ActorId, message);
    }
    public static int GetRpcId(this ActorMessageSenderComponent self) {
        return ++self.RpcId;
    }
// 这个方法：只对当前进程的发送要求 IActorResponse 的消息，封装自家进程的 rpcId，也就是标明本进程发的消息，来自其它进程的返回消息，
    public static async ETTask<IActorResponse> Call(
        this ActorMessageSenderComponent self,
        long actorId,
        IActorRequest request,
        bool needException = true
        ) {
```

```
        request.RpcId = self.GetRpcId(); // 封装本进程的 rpcId
        if (actorId == 0) throw new Exception($"actor id is 0: {request}");
        return await self.Call(actorId, request.RpcId, request, needException);
    }
// 【艰森诲涩难懂!!】是更底层的实现细节，它封装帮助实现 ET7 里消息超时自动过滤抛异常、返回消息的底层封装自动回复、封装了异步任务和必
    public static async ETTask<IActorResponse> Call( // 跨进程发请求消息（要求回复）：返回跨进程异步调用结果。是 await 关键字调用
        this ActorMessageSenderComponent self,
        long actorId,
        int rpcId,
        IActorRequest iActorRequest,
        bool needException = true
        ) {
        if (actorId == 0)
            throw new Exception($"actor id is 0: {iActorRequest}");
// 对象池里：取一个异步任务。用这个异步作务实例，去创建下面的消息发送器实例。这里的 IActorResponse T 应该只是一个索引。因为前面看见
        var tcs = ETTask<IActorResponse>.Create(true);
        // 下面，封装好消息发送器，交由消息发送组件管理；交由其管理，就自带消息发送计时超时过滤机制，实现服务器超负荷时的自动分压减压
        self.requestCallback.Add(rpcId, new ActorMessageSender(actorId, iActorRequest, tcs, needException));
        self.Send(actorId, iActorRequest); // 把请求消息发出去：所有消息，都调用这个
        long beginTime = TimeHelper.ServerFrameTime();
// 自己想一下的话：异步消息发出去，某个服会处理，有返回消息的话，这个服处理后会返回一个返回消息。
// 那么下面一行，不是等待创建 Create() 异步任务（同步方法狠快），而是等待这个处理发送消息的服，处理并返回来返回消息（是说，那个服，把
// 不是等异步任务的创建完成（同步方法狠快），实际是等处理发送消息的服，处理完并写好返回消息，同步到异步任务。
// 那个 ETTask 里的回调 callback，是怎么回调的？这里 Tcs 没有设置任何回调。ETTask 里所谓回调，是执行异步状态机的下一步，没有实际应
// 或说把返回消息的内容填好，【应该还没发回到消息发送者???】返回消息填好了，ETTask 异步任务的结果同步到位了，底层会自动发回来
// 【异步任务结果是怎么回来的？】是前面看过的 IMHandler 的底层封装（AMRpcHandler 的抽象逻辑里）发送回来的。ET7 IMHandler 不是重构实
        IActorResponse response = await tcs;  // 等待消息处理服处理完，写好同步好结果到异步任务、异步任务执行完成，状态为 Succeed
        long endTime = TimeHelper.ServerFrameTime();
        long costTime = endTime - beginTime;
        if (costTime > 200)
            Log.Warning($"actor rpc time > 200: {costTime} {iActorRequest}");
        return response; // 返回：异步网络调用的结果
    }
// 【组件管理器的执行频率， Run() 方法的调用频率】：要是消息太多，发不完怎么办呢？去搜索下面调用 Run() 方法的正常结果消息的调用处理频
// 【ActorHandleHelper 帮助类】：老是调用这里的方法，要去查那个文件。【本质：内网消息处理器的处理逻辑，一旦是返回消息，就会调用 Actor
// 下面方法：处理 IActorResponse 消息，也就是，发回复消息给收消息的人 XX，那么谁发，怎么发，就是这个方法的定义
    // 当是处理【同一进程的消息】：拿到的消息发送器就是当前组件自己，那么只要把结果同步到当前组件的 Tcs 异步任务结果里，异步任务结果就
    public static void HandleIActorResponse(this ActorMessageSenderComponent self, IActorResponse response) {
        ActorMessageSender actorMessageSender;
// 下面取、实例化 ActorMessageSender 来看，感觉收消息的 rpcId，与消息发送者 ActorMessageSender 成一一对应关系。上面的 Call() 方法
        if (!self.requestCallback.TryGetValue(response.RpcId, out actorMessageSender)) // 这里取不到，是说，这个返回消息的发送
            return;
        self.requestCallback.Remove(response.RpcId); // 这个有序字典，就成为实时更新：随时添加，随时删除
        Run(actorMessageSender, response); // <<<<<<<<<<<<<<<<<<<
    }
}
```

## 2.23 ProtoBuf 相关:IExtensible,IExtension,IProtoOutput<TOutput>,IMeasur 看不懂

### 2.23.1 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);
}
```

### 2.23.2 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();
}
```

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

## 3.1 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;
    }
```

## 3.2 【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
```

### 3.3 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。看如何重构现项目。这个晚上再弄。

### 3.4 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.Enviorment.AppDomain app) {
        BindingFlags flag = BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static | BindingFlags.Declared
        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);
    }
```

## 4 【拖拉机游戏房间】组件：分析

### 4.1 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 ETTask<UI> OnCreate(UIComponent uiComponent, UILayer uiLayer) {
        await ETTask.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());
    }
}
```

## 4.2 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;
            }
    }
}
```

## 4.3 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;
    }
}
```

## 4.4 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;
    }
}
```

## 4.5 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>();
p       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
    }
}
```

## 4.6 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;
}
```

## 4.7 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;
        }
    }
```

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

- 上下这一两个组件里，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？要去所源码弄透。

```
// 【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; }
```

# 5 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。。
- 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?
- 先前搜的时候，关于应用框架的数据缓存，什么时候需要一个缓存层，应用运行的时候，数据是否在内存等，为什么 ET7 框架使用 MongoDB, 就是这个这类数据库，为什么比较适合双端游戏框架，而为什么 MySQL 之类的破烂库就各种不适合? 感觉这些比较上层的原理，或基础原理，自己理解得不够透彻，看过网上的别人的分析，但理解得还不够透彻。
- 我可能需要把 ET7 重构后、被破烂框架开发者各个主要模块、删除得几乎不剩下什么的模块、与重构前的 ET6 等模块，再多读一下源码，理解得透彻一点儿再来事理这个模块。现游戏里需要用数据库的地方，主要是用户帐户数据（这应该是注册登录服的逻辑），帐户管理与游戏数据需要相区分吗? 账户管理，游戏数据

## 5.1 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

- 狠奇怪的是，框架里，居然没有一个实现这个接口的实现类?

```
public interface IDBCollection {}
```

## 5.2 DBComponent: 【CRUD】可以查表，查询数据等，各种数据库操作的基本方法

- 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;
}
```

## 5.3 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
}
```

## 5.4 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;
    }
}
```

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

- 没明白，以前的框架什么情境、或使用上下文下，需要使用代理。ET7 重构后，感觉就是下放到了各个小区，使用时去拿各区里的数据库。区里的数据库，感觉管理的也是各小区里什么相关数据。
- 代理里的操作方法【CRUD】前面定义的组件里，可以完成对数据库的各种基本操作。
- 什么时候需要先前如参考项目里的代理，ET7 不需要的话，还必须添加哪些吗?

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

## 6 组件定义，再澄明，与去重

### 6.1 OnlineComponent: 参考项目里的，现框架里查找一下

```
// 在线组件，用于记录在线玩家
public class OnlineComponent : Entity {
    private readonly Dictionary<long, int> dictionary = new Dictionary<long, int>();
    // 添加在线玩家
    public void Add(long userId, int gateAppId) {
        dictionary.Add(userId, gateAppId);
    }
    // 获取在线玩家网关服务器 ID
    public int Get(long userId) {
        int gateAppId;
        dictionary.TryGetValue(userId, out gateAppId);
        return gateAppId;
    }
    // 移除在线玩家
    public void Remove(long userId) {
        dictionary.Remove(userId);
    }
}
```

### 6.2 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();
    }
}
```

### 6.3 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

### 6.4 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 ?? IdGenerater.GenerateId(), userId);
        gamer.PlayerID = playerId;
        return gamer;
    }
}
```

## 7 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

## 8 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!!!】

## 9 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，画葫芦画飘地重构
- 【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!!!】

# 10 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!!!】
- 【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!!!】
- 【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!!!】
- 【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!!!】

# 11 先前版本 LocationComponent 原理分析

- 框架的应用场景里，知道对方的 InstanceId 就可以给对方发消息。
- 问题是，对方的可以下线再上线，活宝妹可以从加州地图服重入亲爱的表哥所在的 WA 地图服（不同州的地图服服务器进程不一样），对方的 InstanceId 是变化的，小伙伴也可以搬家，搬家过程中位置不确认，还要先锁住，搬完才实时更新位置服管家。
- 【框架需求】：InstanceId 标识唯一身份。但仍需要对 Entity 级别（框架的最底最基类封装）不同对象的 InstanceId 进行管理（因为不同游戏实现里，可能会分线、可能会分地图服，不同地图服处于不同进程。一旦服务器进程变了，就需要对管家实时更新：更新要搬家，更新搬家完成了，位置确定了，活宝妹就是一定要嫁给亲爱的表哥!!）。所以会有当前【位置服】。

- 功能一：【查询位置信息】。亲爱的表哥的活宝妹，想要给亲爱的表哥发消息，活宝妹就需要先知道亲爱的表哥的手机号才能发。怎么才能知道亲爱的表哥的手机号呢？活宝妹可以查询框架里，美国手机号管理位置服就行。因为是客户端的查询需求，服务端异步返回查询结果，同其它异步网络操作一样，封装异步任务。ET7 中异步任务重构的这块儿理解透彻了（适配和改的时候，能狠快完成）。现只关心位置服相关逻辑。

- 功能二：【更新位置信息】。半年前活宝妹搬家前，【客户端】活宝妹先通知位置服管家，活宝妹要搬家；【位置服】把活宝妹的位置信息上锁不给查，并所有查询活宝妹位置的跨进程消息全放进队列里等（超时了，大概？也会通知发送者，她搬家，现位置不知道，改天过段时间再来查询吧）；活宝妹搬完家了，【客户端】活宝妹通知位置服，活宝妹重入了亲爱的表哥所在的 WA 地图；【位置服】更新了活宝妹的最新位置（记字典小本本里），并一一回复队列里尚未超时的索要活宝妹位置的消息，一一回复他们，活宝妹现在在亲爱的表哥所在的 WA 这里。等活宝妹嫁给亲爱的表哥了，活宝妹可能还会想要出去玩耍。等亲爱的表哥的活宝妹嫁给亲爱的表哥了，如有需要，或任何以 Entity 为基类的实例有、会重入其它线地图服或进程切换需求，双端就会如活宝妹上次搬家般，实现对活宝妹，对任何客户端的位置进行管理。。。

- 解决问题的步骤：查看重构游戏项目框架里，这一模块的破烂开发者，是出于什么考虑，把这个模块删除得几乎不剩下什么。活宝妹现在要整合或是接入这个位置服组件，要如何整合、接入与适配？

- 感觉原理基本也都懂的，以前不同的参考项目，不同的版本，零零碎碎地都读过，可能稍微久缺一点儿系统化梳理这个服务器与模块功能。这里要整合或是接入这个位置服管理组件，下午就根据框架里现在存有的编译错误，来试着把这个功能模块整合或是接入完成。

- 其它模块，明天再看。现再看一会现重构项目里，需要使用个这管理组件的使用情境上下文。

  - 项目里，好像更多的是在定义和处理先前功能模块划分不够明确的各种破烂锁。下午我可以先试着把这个位置服管理组件的几个文件，先不加入项目（从.csproj 里标注项目不引用文件），先消除所有相关的编译错误。以后有再、还需要这个位置服逻辑的时候，再重新添加引用回来。

- 【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！！！】