ET 框架学习笔记（五） - - 网络交互相关与 Actor 机制

deepwaterooo

August 7, 2023

Contents

1 Actor 消息相关：跟上个章节 Net 相关一起总结，两个都不太清楚。放一起总结，希望都能够理解清楚 **1**
1.1 ActorMessageSender: 知道对方的 instanceId，使用这个类发 actor 消息 1
1.2 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，背后有套计时器管理组件，自动检测消息的发送超时。 2
1.3 ActorMessageSenderComponentSystem: 这个类底层封装比较多，功能模块因为是服务器端不太敦悉，多看几遍 2
1.4 LocationProxyComponent: 【位置代理组件】：为什么称它为代理？ 5
1.5 LocationProxyComponentSystem: 5
1.6 ：一个添加位置信息的请求消息处理类，示例 5
1.7 ActorLocationSender: 知道对方的 Id，使用这个类发 actor 消息 5
1.8 ActorLocationSenderComponent: 位置发送组件 5
1.9 ActorLocationSenderComponentSystem: 这个类，也要明天上午再看一下 5
1.10 ActorHelper: 帮助创建 IActorResponse 回复消息。狠简单 7
1.11 Actor 消息处理器：基本原理 7
1.12 MailboxType 7
1.13 ActorMessageDispatcherInfo | ActorMessageDispatcherComponent: 【消息分发器组件】 7
1.14 ActorMessageDispatcherComponentHelper: 帮助类 8
1.15 ActorMessageHandlerAttribute 标签系：去找几个典型标签看看 9
1.16 [ActorMessageHandler(SceneType.Gate)] 标签使用举例: 9
1.17 MailBoxComponent: 挂上这个组件表示该 Entity 是一个 Actor, 接收的消息将会队列处理 9
1.18 【服务端】ActorHandleHelper 帮助类：连接上下层的中间层桥梁 9
1.19 NetInnerComponentOnReadEvent: 11

2 Net 网络交互相关：【服务端 + 客户端】只是稍微改装成事件机制。模块没理解透、总结不全，还需要借助总结，和改掉所有编译错误后的运行、以及运行日志，来理解这个流程 **12**
2.1 RpcInfo: 【消息的包装体】。内部包装一个 Tcs 异步任务，桥接异步结果给调用方。合并入其它小节 12
2.2 NetThreadComponent: 12
2.3 NetServerComponent: NetServerComponentOnRead 结构体。 12
2.4 NetServerComponentSystem: 场景上的【服务端】组件，可发布【服务端读到消息事件】 13
2.5 NetServerComponentOnReadEvent: NetServerComponent 组件，会发布事件，触发此回调类 14
2.6 Option 单例类： 15

2.7 NetClientComponent: 【网络客户端】组件：这个，感觉与【服务端】定义申明上看是一样的 . . . 16
2.8 NetClientComponentSystem: 【服务端】也是类似事件系统的改装 . . . 17
2.9 NetClientComponentOnReadEvent: 【网络客户端】读到消息事件：它，要如何处理读到的消息呢? . . . 18
2.10 NetInnerComponent: 【服务端】对不同进程的处理组件。是服务器的组件 . . . 18
2.11 NetInnerComponentSystem: 生成系 . . . 18
2.12 MessageDispatcherInfo: 在【MessageDispatcherComponent】中 . . . 20
2.13 MessageDispatcherComponent: 全局全框架单例：【活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】 . . . 20
2.14 MessageDispatcherComponentSystem: . . . 20
2.15 MessageDispatcherComponentHelper: . . . 21
2.16 SessionIdleCheckerComponent: 【会话框】闲置状态管理组件 . . . 22
2.17 SessionIdleCheckerComponentSystem: SessionIdleChecker 激活类， . . . 22
2.18 MessageHelper: 不知道这个类是作什么用的，使用场景等。过会儿看下 . . . 22
2.19 ActorHandleHelper: 是谁调用它，什么场景下使用的? 这个，今天下午再补吧 . . . 23

3 StartConfigComponent: 找【各种服】的起始初始化地址 25
3.1 OptionAttribute: 系统里的标签属性。 . . . 25
3.2 Options 单例类： . . . 25
3.3 模块里所用到的几个。NET 里的接口, 以及自定义的框架底层辅助体系类等 . . . 26
3.3.1 ISupportInitialize: 【初始化】的支持接口，就是提供了【初始化之前】【初始化之后】的回调，两个 API . . . 26
3.3.2 IInvoke: 抽象类会在事件系统 EventSystem.cs 中被用到 . . . 26
3.3.3 ISingleton 单例类接口：框架最底层，有狠多必要的单例类包装，统一实现这个单例接口，就是抽象提纯到框架最底层封装 . . . 26
3.3.4 IMerge: 在 Proto 相关的地方，某些类如 StartProcessConfig.cs 会实现这个接口，进程中以消息的形式传递这部分原理也要弄懂 . . . 27
3.4 ProtoObject: 继承自上面的系统接口，定义必要的回调抽象 API . . . 27
3.5 ConfigLoader.cs: 【服务端】是理解接下来部分的基础。【客户端】有不同逻辑。所以要把两边的都看一下 . . . 27
3.6 ConfigLoader:【客户端】 . . . 28
3.7 ConfigComponent 组件：单例类。底层组件，负责服务端配置相关管理? . . . 29
3.8 ConfigSingleton<T>: ProtoObject, ISingleton . . . 30
3.9 StartMachineConfig: 抓四大单例管理类中的一个来读一下 . . . 31
3.10 StartProcessConfig: 【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!!!】 . . . 32
3.11 StartSceneConfig: ISupportInitialize 【各种服 - 配置，场景配置】 . . . 33
3.12 StartSceneConfigCategory : 【Matchs!】ConfigSingleton<StartSceneConfigCategory>, IMerge . . . 34

1 Actor 消息相关：跟上个章节 Net 相关一起总结，两个都不太清楚。放一起总结，希望都能够理解清楚

- 跨进程【发送消息】与【返回消息】的过程，总感觉无法完整地看通一遍。这个是狠久前的总结，还是修改更新下。等亲爱的表哥的活宝妹搬进新住处后，会改完所有的编译错误，会需要把这个重构游戏写完整。

- ET 中，正常的网络消息需要建立一个 session 链接来发送，这类消息对应的 proto 需要由 IMessage，IResponse，IRequest 来修饰。(这是最常规，感觉最容易理解的)

- 另外还有一种消息机制，称为 **【Actor 机制】**，挂载了 MailBoxComponent 的实体会成为一个 actor. 而向 Actor 发送消息可以根据实体的 instanceId 来发送，不需要自己建立 session 链接，这类消息在 proto 中会打上 IActorRequest, IActorResponse, IActorMessage 的注释，标识为 Actor 消息。这种机制极大简化了服务器间向 Actor 发送消息的逻辑，使得实体间通信更加灵活方便。

- 上面的，自己去想明白，挂载了 MailBoxComponent 的组件实体，知道对方实体的 instanceId，背后的封装原理，仍然是对方实体 instanceId 之类的生成得比较聪明，自带自家进程 id，让 MailBoxCompoent 能够方便拿到发向收消息的进程？忘记了，好像是这样的。就是本质上仍是第一种，但封装得狠受用户弱弱程序员方便实用。。。

- 但有的时候实体需要在服务器间传递（这一块儿还没有涉入，可以简单理解为玩家 me 从加州地图，重入到了亲爱的表哥身边的地图，不嫁给亲爱的表哥就永远不再离开。me 大概可以理解为从一个地图服搬家转移重入到了另一个地图服，me 所属的进程可能已经变了），每次传递都会实例化一个新的，其 instanceId 也会变，但实体的 id 始终不会变，所以为了应对实体传递的问题，增加了 proto 需要修饰为 IActorLocationRequest, IActorLocationResponse, IActorLocationMessage 的消息【这一块儿仍不懂，改天再捡】，它可以根据实体 Id 来发送消息，不受实体在服务器间传递的影响，很好的解决了上面的问题。

1.1 ActorMessageSender: 知道对方的 instanceId，使用这个类发 actor 消息

- Tcs 成员变量：精华在这里：因为内部自带一个 IActorResponse 的异步任务成员变量，可以帮助实现异步消息的自动回复

- 正是因为内部成员自带一个异步任务，所以会多一个成员变量，就是标记是否要抛异常。这是异步任务成员变量带来的

```
public readonly struct ActorMessageSender {
    public long ActorId { get; }
    public long CreateTime { get; } // 最近接收或者发送消息的时间
    public IActorRequest Request { get; }      // 结构体，也自动封装了，发送的消息
    public bool NeedException { get; }         // 这上下三行：就帮助实现，返回消息的自动回复的结构包装
    public ETTask<IActorResponse> Tcs { get; } // <<<<<<<<<<<<<<<<<<<< 精华在这里：因为内部自带一个 IActorResponse 的异
    public ActorMessageSender(long actorId, IActorRequest iActorRequest, ETTask<IActorResponse> tcs, bool needExcepti
        this.ActorId = actorId;
        this.Request = iActorRequest;
        this.CreateTime = TimeHelper.ServerNow();
        this.Tcs = tcs;
        this.NeedException = needException;
    }
}
```

1.2 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，背后有套计时器管理组件，自动检测消息的发送超时。

- 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.ActorMess**标注的 ActorMessageSenderChecker 里会用到，检测超时与否

- **【组件里消息自动超时 Timer 的计时器机制】**：
  - long TimeoutCheckTimer 是个重复闹钟
  - **【TimerComponent】**：是框架里的单例类，那么应该是，框架里所有的 Timer 定时计时器，应该是由这个单例管理类统一管理。那么这个组件应该能够负责相关逻辑。

```
[ComponentOf(typeof(Scene))]
public class ActorMessageSenderComponent: Entity, IAwake, IDestroy {
// 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【Invoke(TimerInvokeType.Ac
    public const long TIMEOUT_TIME = 40 * 1000;
    public static ActorMessageSenderComponent Instance { get; set; }
    public int RpcId;
    public readonly SortedDictionary<int, ActorMessageSender> requestCallback = new SortedDictionary<int, Actor
// 这个 long: 是重复闹钟的闹钟实例 ID，用来区分任何其它闹钟的
    public long TimeoutCheckTimer;
    public List<int> TimeoutActorMessageSenders = new List<int>(); // 这帧更新里：待发送给的（接收者 rpcId）接收者
}
```

1.3 ActorMessageSenderComponentSystem: 这个类底层封装比较多，功能模块因为是服务器端不太敦悉，多看几遍

- 这个类，可以看见 ET7 框架更为系统化、消息机制的更为往底层或说更进一步的封装，就是今天下午看见的，以前的 handle() 或是 run() 方法，或回调实例 Action<T> reply, 现在的封装里，这些什么创建回复实例之类的，全部封装到了管理器或是帮助类
- 如果发向同一个进程，则直接处理，不需要通过网络层。内网组件处理内网消息：这个分支可以再跟一下源码，理解一下重构的事件机制流程
- 这个生成系，前半部分的计时器消息超时检测，看懂了；后半部分，还没看懂连能。今天上午能连多少连多少
- 后半部分：是消息发送组件的相对底层逻辑。上层逻辑连通内外网消息，消息处理器，和读到消息发布事件后的触发调用等几个类。要把它们的连通流通原理弄懂。

```
[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，
    [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
    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
            try {
                self.Check(); // 调用组件自己的方法
             } catch (Exception e) {
                Log.Error($"move timer error: {self.Id}\n{e}");
            }
        }
    }
    [ObjectSystem]
    public class ActorMessageSenderComponentAwakeSystem: AwakeSystem<ActorMessageSenderComponent> {
//  【组件重复闹钟的设置】: 实现组件内，消息的自动计时，超时触发 Invoke 标签，调用相关逻辑来检测超时消息
        protected override void Awake(ActorMessageSenderComponent self) {
            ActorMessageSenderComponent.Instance = self;
// 这个重复闹钟，是消息自动计时超时过滤器的上下文连接桥梁
// 它注册的回调 TimerInvokeType.ActorMessageSenderChecker, 会每个消息超时的时候，都会回来调用 checker 的 Run()==>Check()
// 应该是重复闹钟每秒重复一次，就每秒检查一次，调用一次 Check() 方法来检查超时？是过滤器会给服务器减压；但这里的自动检测会把压
// 这个重复间隔 1 秒钟的时间间隔，它计 1 秒钟开始，重复的逻辑是重复闹钟处理
            self.TimeoutCheckTimer = TimerComponent.Instance.NewRepeatedTimer(1000, TimerInvokeType.ActorMessageSende
        }
    }//...
// Run() 方法：通过同步异常到 ETTask, 通过 ETTask 封装的抛异常方式抛出两类异常并返回；和对正常非异常返回消息，同步结果到 ETT
// 传进来的参数：是一个 IActorResponse 实例，是有最小预处理（初始化了最基本成员变量：异常类型)、【写了个半好】的结果（异常)。结
    private static void Run(ActorMessageSender self, IActorResponse response) {
        // 对于每个超时了的消息：超时错误码都是：ErrorCore.ERR_ActorTimeout, 所以会从发送消息超时异常里抛出异常，不用发送错
        if (response.Error == ErrorCore.ERR_ActorTimeout) { // 写：发送消息超时异常。因为同步到异步任务 ETTask 里，所以异
            self.Tcs.SetException(new Exception($"Rpc error: request, 注意 Actor 消息超时，请注意查看是否死锁或者没有 re
            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
// 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
        if (self.NeedException && ErrorCore.IsRpcNeedThrowException(response.Error)) { // 若是有异常（判断条件：消息要投
            self.Tcs.SetException(new Exception($"Rpc error: actorId: {self.ActorId} request: {self.Request}, respons
            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
            // 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
            // 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
            // 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
            // 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，
        }
        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，实际中有
            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
                    // 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
                    IActorResponse response = ActorHelper.CreateResponse(actorMessageSender.Request, ErrorCore.ERR_ActorT
                    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
            if (actorId == 0)
                throw new Exception($"actor id is 0: {message}");
            ProcessActorId processActorId = new(actorId);
            // 这里做了优化，如果发向同一个进程，则直接处理，不需要通过网络层
            if (processActorId.Process == Options.Instance.Process) { // 没看懂：这里怎么就说，消息是发向同一进程的了？
                NetInnerComponent.Instance.HandleMessage(actorId, message); // 原理清楚：本进程消息，直接交由本进程内网组件处
                return;
            }
            Session session = NetInnerComponent.Instance.Get(processActorId.Process); // 非本进程消息，去走网络层
            session.Send(processActorId.ActorId, message);
        }
        public static int GetRpcId(this ActorMessageSenderComponent self) {
            return ++self.RpcId;
        }
        // 这个方法：只对当前进程的发送要求 IActorResponse 的消息，封装自家进程的 rpcId，也就是标明本进程发的消息，来自其它进程的
        public static async ETTask<IActorResponse> Call(
            this ActorMessageSenderComponent self,
            long actorId,
            IActorRequest request,
            bool needException = true
            ) {
            request.RpcId = self.GetRpcId(); // 封装本进程的 rpcId
            if (actorId == 0) throw new Exception($"actor id is 0: {request}");
            return await self.Call(actorId, request.RpcId, request, needException);
        }
        // 【艰深诲涩难懂!!】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
        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
            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。用这个异步任务实例，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
            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
            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
```

```
    // 不是等异步任务的创建完成（同步方法很快），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
    // 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
            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
            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
    // 【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
    // 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
        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
            if (!self.requestCallback.TryGetValue(response.RpcId, out actorMessageSender)) // 这里取不到，是说，这个返回消息
                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); // <<<<<<<<<<<<<<<<<<<
        }
    }
```

- 几个类弄懂：ActorHandleHelper, 以及再上面的，NetInnerComponentOnReadEvent 事件发布等，上层调用的几座桥连通了，才算把整个流程弄懂了。
- 现在不懂的就变成为：更为底层的，Session 会话框，socket 层的机制。但是因为它们更为底层，亲爱的表哥的活宝妹，现在把有限的精力投入支理解这个框架，适配自己的游戏比较重要。其它不太重要，或是更为底层的，改天有必要的时候再捡再看。【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】

1.4 LocationProxyComponent: 【位置代理组件】：为什么称它为代理？

- 就是有个启动类管理 StartSceneConfigCategory 类，它会分门别类地管理一些什么网关、注册登录服，地址服之类的东西。然后从这个里面拿位置服务器地址？大概意思是这样。写得可能不对。今天剩下一点儿时间，再稍微看一下
- 感觉先前、上面仍然是写得不伦不类。总之，位置相关组件就是管理框架里各种可收发消息的实例，他们所在的（场景？位置？服务器地址？）相关位置信息。【亲爱的表哥的活宝妹就是一定要嫁给亲爱的表哥!! 活宝妹只是在等：亲爱的表哥同活宝妹的一纸结婚证。活宝妹若是还没能嫁给亲爱的表哥，活宝妹就永远守候在亲爱的表哥的身边!! 爱表哥，爱生活!!!】

```
[ComponentOf(typeof(Scene))]
public class LocationProxyComponent: Entity, IAwake, IDestroy {
    [StaticField]
    public static LocationProxyComponent Instance;
}
```

1.5 LocationProxyComponentSystem:

- 为什么要加那堆什么也没曾看懂的源码在那里？

```
// [ObjectSystem] awake() etc
```

1.6 ：一个添加位置信息的请求消息处理类，示例

1.7 ActorLocationSender: 知道对方的 Id，使用这个类发 actor 消息

```
[ChildOf(typeof(ActorLocationSenderComponent))]
public class ActorLocationSender: Entity, IAwake, IDestroy {
    public long ActorId;
```

```
    public long LastSendOrRecvTime; // 最近接收或者发送消息的时间
    public int Error;
}
```

1.8 ActorLocationSenderComponent: 位置发送组件

```
 [ComponentOf(typeof(Scene))]
 public class ActorLocationSenderComponent: Entity, IAwake, IDestroy {
     public const long TIMEOUT_TIME = 60 * 1000;
     public static ActorLocationSenderComponent Instance { get; set; }
     public long CheckTimer;
 }
```

1.9 ActorLocationSenderComponentSystem: 这个类，也要明天上午再看一下

```
[Invoke(TimerInvokeType.ActorLocationSenderChecker)]
public class ActorLocationSenderChecker: ATimer<ActorLocationSenderComponent> {
    protected override void Run(ActorLocationSenderComponent self) {
        try {
            self.Check();
        }
        catch (Exception e) {
            Log.Error($"move timer error: {self.Id}\n{e}");
        }
    }
}
// [ObjectSystem] // ...
[FriendOf(typeof(ActorLocationSenderComponent))]
[FriendOf(typeof(ActorLocationSender))]
public static class ActorLocationSenderComponentSystem {
    public static void Check(this ActorLocationSenderComponent self) {
        using (ListComponent<long> list = ListComponent<long>.Create()) {
            long timeNow = TimeHelper.ServerNow();
            foreach ((long key, Entity value) in self.Children) {
                ActorLocationSender actorLocationMessageSender = (ActorLocationSender) value;
                if (timeNow > actorLocationMessageSender.LastSendOrRecvTime + ActorLocationSenderComponent.TIMEOUT_TIME)
                    list.Add(key);
            }
            foreach (long id in list) {
                self.Remove(id);
            }
        }
    }
    private static ActorLocationSender GetOrCreate(this ActorLocationSenderComponent self, long id) {
        if (id == 0)
            throw new Exception($"actor id is 0");
        if (self.Children.TryGetValue(id, out Entity actorLocationSender)) {
            return (ActorLocationSender) actorLocationSender;
        }
        actorLocationSender = self.AddChildWithId<ActorLocationSender>(id);
        return (ActorLocationSender) actorLocationSender;
    }
    private static void Remove(this ActorLocationSenderComponent self, long id) {
        if (!self.Children.TryGetValue(id, out Entity actorMessageSender))
            return;
        actorMessageSender.Dispose();
    }
    public static void Send(this ActorLocationSenderComponent self, long entityId, IActorRequest message) {
        self.Call(entityId, message).Coroutine();
    }
    public static async ETTask<IActorResponse> Call(this ActorLocationSenderComponent self, long entityId, IActorRequest iA
        ActorLocationSender actorLocationSender = self.GetOrCreate(entityId);
        // 先序列化好
        int rpcId = ActorMessageSenderComponent.Instance.GetRpcId();
        iActorRequest.RpcId = rpcId;
        long actorLocationSenderInstanceId = actorLocationSender.InstanceId;
        using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.ActorLocationSender, entityId)) {
            if (actorLocationSender.InstanceId != actorLocationSenderInstanceId)
                throw new RpcException(ErrorCore.ERR_ActorTimeout, $"{iActorRequest}");
            // 队列中没处理的消息返回跟上个消息一样的报错
```

```
            if (actorLocationSender.Error == ErrorCore.ERR_NotFoundActor)
                return ActorHelper.CreateResponse(iActorRequest, actorLocationSender.Error);
            try {
                return await self.CallInner(actorLocationSender, rpcId, iActorRequest);
            }
            catch (RpcException) {
                self.Remove(actorLocationSender.Id);
                throw;
            }
            catch (Exception e) {
                self.Remove(actorLocationSender.Id);
                throw new Exception($"{iActorRequest}", e);
            }
        }
    }
    private static async ETTask<IActorResponse> CallInner(this ActorLocationSenderComponent self, ActorLocationSender actor
        int failTimes = 0;
        long instanceId = actorLocationSender.InstanceId;
        actorLocationSender.LastSendOrRecvTime = TimeHelper.ServerNow();
        while (true) {
            if (actorLocationSender.ActorId == 0) {
                actorLocationSender.ActorId = await LocationProxyComponent.Instance.Get(actorLocationSender.Id);
                if (actorLocationSender.InstanceId != instanceId)
                    throw new RpcException(ErrorCore.ERR_ActorLocationSenderTimeout2, $"{iActorRequest}");
            }
            if (actorLocationSender.ActorId == 0) {
                actorLocationSender.Error = ErrorCore.ERR_NotFoundActor;
                return ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            }
            IActorResponse response = await ActorMessageSenderComponent.Instance.Call(actorLocationSender.ActorId, rpcId, i
            if (actorLocationSender.InstanceId != instanceId)
                throw new RpcException(ErrorCore.ERR_ActorLocationSenderTimeout3, $"{iActorRequest}");
            switch (response.Error) {
                case ErrorCore.ERR_NotFoundActor: {
                    // 如果没找到 Actor, 重试
                    ++failTimes;
                    if (failTimes > 20) {
                        Log.Debug($"actor send message fail, actorid: {actorLocationSender.Id}");
                        actorLocationSender.Error = ErrorCore.ERR_NotFoundActor;
                        // 这里不能删除 actor, 要让后面等待发送的消息也返回 ERR_NotFoundActor, 直到超时删除
                        return response;
                    }
                    // 等待 0.5s 再发送
                    await TimerComponent.Instance.WaitAsync(500);
                    if (actorLocationSender.InstanceId != instanceId)
                        throw new RpcException(ErrorCore.ERR_ActorLocationSenderTimeout4, $"{iActorRequest}");
                    actorLocationSender.ActorId = 0;
                    continue;
                }
                case ErrorCore.ERR_ActorTimeout:
                    throw new RpcException(response.Error, $"{iActorRequest}");
            }
            if (ErrorCore.IsRpcNeedThrowException(response.Error)) {
                throw new RpcException(response.Error, $"Message: {response.Message} Request: {iActorRequest}");
            }
            return response;
        }
    }
}
```

1.10 ActorHelper: 帮助创建 IActorResponse 回复消息。狠简单

```
public static class ActorHelper {
    public static IActorResponse CreateResponse(IActorRequest iActorRequest, int error) {
        Type responseType = OpcodeTypeComponent.Instance.GetResponseType(iActorRequest.GetType());
        IActorResponse response = (IActorResponse)Activator.CreateInstance(responseType);
        response.Error = error;
        response.RpcId = iActorRequest.RpcId;
        return response;
    }
}
```

1.11 Actor 消息处理器：基本原理

- 消息到达 MailboxComponent，MailboxComponent 是有类型的，不同的类型邮箱可以做不同的处理。目前有两种邮箱类型 GateSession 跟 MessageDispatcher。
  - GateSession 邮箱在收到消息的时候会立即转发给客户端。Actor 消息是指来自于服务端的消息（一定是来自于服务端的消息？Actor 一定是进程间，来自于其它服务端的?）。网关服是小区下所有用户的接收消息的代理。所以，网关服一旦收到服务端的返回消息，作为小区下所有用户的代理，就直接转发相应用户。【亲爱的表哥，永远是活宝妹的代理！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!!!】
  - MessageDispatcher 类型会再次对 Actor 消息进行分发到具体的 Handler 处理，默认的 MailboxComponent 类型是 MessageDispatcher。

1.12 MailboxType

```
public enum MailboxType {
    MessageDispatcher, // 消息分发器
    UnOrderMessageDispatcher,// 无序分发
    GateSession,// 网关？
}
```

1.13 ActorMessageDispatcherInfo | ActorMessageDispatcherComponent: 【消息分发器组件】

```
public class ActorMessageDispatcherInfo {
    public SceneType SceneType { get; }
    public IMActorHandler IMActorHandler { get; }
    public ActorMessageDispatcherInfo(SceneType sceneType, IMActorHandler imActorHandler) {
        this.SceneType = sceneType;
        this.IMActorHandler = imActorHandler;
    }
}
// Actor 消息分发组件
[ComponentOf(typeof(Scene))] // 场景的子组件
public class ActorMessageDispatcherComponent: Entity, IAwake, IDestroy, ILoad {
    [StaticField]
    public static ActorMessageDispatcherComponent Instance; // 全局单例吗？好像是，只在【服务端】添加了这个组件
    // 下面的字典：去看下，同一类型，什么情况下会有一个链表的不同消息分发处理器？
    public readonly Dictionary<Type, List<ActorMessageDispatcherInfo>> ActorMessageHandlers = new();
}
```

- 添加全局单例组件的地方是在：

```
[Event(SceneType.Process)]
public class EntryEvent2_InitServer: AEvent<ET.EventType.EntryEvent2> {
    protected override async ETTask Run(Scene scene, ET.EventType.EntryEvent2 args) {
        // 发送普通 actor 消息
        Root.Instance.Scene.AddComponent<ActorMessageSenderComponent>(); // 【服务端】几个组件：现在这个组件，最熟悉
        // 自己添加：【数据库管理类组件】
        Root.Instance.Scene.AddComponent<DBManagerComponent>(); // 【服务端】几个组件：现在这个组件，最熟悉
        // 发送 location actor 消息
        Root.Instance.Scene.AddComponent<ActorLocationSenderComponent>();
        // 访问 location server 的组件
        Root.Instance.Scene.AddComponent<LocationProxyComponent>();
        Root.Instance.Scene.AddComponent<ActorMessageDispatcherComponent>();
        Root.Instance.Scene.AddComponent<ServerSceneManagerComponent>();
        Root.Instance.Scene.AddComponent<RobotCaseComponent>();
        Root.Instance.Scene.AddComponent<NavmeshComponent>();
        // 【添加组件】：这里，还可以再添加一些游戏必要【根组件】，如果可以在服务器启动的时候添加的话。会影响服务器启动性能
// ....
}
```

1.14 ActorMessageDispatcherComponentHelper: 帮助类

- Actor 消息分发组件：对于管理器里的，对同一发送消息类型，不同场景下不同处理器的链表管理，多看几遍
- 这里，对于同一发送消息类型，是会、是可能存在【从不同的场景类型中返回，带不同的消息处理器】以致于必须得链表管理同一发送消息类型的不同可能处理情况。

```
[FriendOf(typeof(ActorMessageDispatcherComponent))] // Actor 消息分发组件：对于管理器里的，对同一发送消息类型，不同场景下
public static class ActorMessageDispatcherComponentHelper {// Awake() Load() Destroy() 省略掉了
    private static void Load(this ActorMessageDispatcherComponent self) { // 加载：程序域回载的时候
        self.ActorMessageHandlers.Clear(); // 清空字典
        var types = EventSystem.Instance.GetTypes(typeof (ActorMessageHandlerAttribute)); // 扫描程序域里的特定消息处理
        foreach (Type type in types) {
            object obj = Activator.CreateInstance(type); // 加载时：框架封装，自动创建【消息处理器】实例
            IMActorHandler imHandler = obj as IMActorHandler;
            if (imHandler == null) {
                throw new Exception($"message handler not inherit IMActorHandler abstract class: {obj.GetType().FullN
            }
            object[] attrs = type.GetCustomAttributes(typeof(ActorMessageHandlerAttribute), false);
            foreach (object attr in attrs) {
                ActorMessageHandlerAttribute actorMessageHandlerAttribute = attr as ActorMessageHandlerAttribute;
                Type messageType = imHandler.GetRequestType(); // 因为消息处理接口的封装：可以拿到发送类型
                Type handleResponseType = imHandler.GetResponseType();// 因为消息处理接口的封装：可以拿到返回消息的类型
                if (handleResponseType != null) {
                    Type responseType = OpcodeTypeComponent.Instance.GetResponseType(messageType);
                    if (handleResponseType != responseType) {
                        throw new Exception($"message handler response type error: {messageType.FullName}");
                    }
                }
                // 将必要的消息【发送类型】【返回类型】存起来，统一管理，备用
                // 这里，对于同一发送消息类型，是会、是可能存在【从不同的场景类型中返回，带不同的消息处理器】以致于必须得链表
                // 这里，感觉因为想不到、从概念上也地无法理解，可能会存在的适应情况、上下文场景，所以这里的链表管理同一发送消
                ActorMessageDispatcherInfo actorMessageDispatcherInfo = new(actorMessageHandlerAttribute.SceneType, 
                self.RegisterHandler(messageType, actorMessageDispatcherInfo); // 存在本管理组件，所管理的字典里
            }
        }
    }
    private static void RegisterHandler(this ActorMessageDispatcherComponent self, Type type, ActorMessageDispatcherI
        // 这里，对于同一发送消息类型，是会、是可能存在【从不同的场景类型中返回，带不同的消息处理器】以致于必须得链表管理
        // 这里，感觉因为想不到、从概念上也地无法理解，可能会存在的适应情况、上下文场景，所以这里的链表管理同一发送消息类型，
        if (!self.ActorMessageHandlers.ContainsKey(type))
            self.ActorMessageHandlers.Add(type, new List<ActorMessageDispatcherInfo>());
        self.ActorMessageHandlers[type].Add(handler);
    }
    public static async ETTask Handle(this ActorMessageDispatcherComponent self, Entity entity, int fromProcess, obje
        List<ActorMessageDispatcherInfo> list;
        if (!self.ActorMessageHandlers.TryGetValue(message.GetType(), out list)) // 根据消息的发送类型，来取所有可能的处
            throw new Exception($"not found message handler: {message}");
        SceneType sceneType = entity.DomainScene().SceneType; // 定位：当前消息的场景类型
        foreach (ActorMessageDispatcherInfo actorMessageDispatcherInfo in list) { // 遍历：这个发送消息类型，所有存在注
            if (actorMessageDispatcherInfo.SceneType != sceneType)  // 场景不符就跳过
                continue;
            // 定位：是当前特定场景下的消息处理器，那么，就调用这个处理器，要它去干事。【爱表哥，爱生活!!! 任何时候，活宝妹就是
            await actorMessageDispatcherInfo.IMActorHandler.Handle(entity, fromProcess, message);
        }
    }
}
```

1.15 ActorMessageHandlerAttribute 标签系: 去找几个典型标签看看

```
public class ActorMessageHandlerAttribute: BaseAttribute {
    public SceneType SceneType { get; }
    public ActorMessageHandlerAttribute(SceneType sceneType) {
        this.SceneType = sceneType;
    }
}
```

1.16 【ActorMessageHandler(SceneType.Gate)】标签使用举例:

- 是以前框架中或是参考项目中的例子。标签使用申明说，这是【网关服】上的一个 Actor 消息处理器定义类。
- 框架中这个标签的例子还有很多。这里是随便抓一个出来。

```
[ActorMessageHandler(SceneType.Gate)]
public class Actor_MatchSucess_NttHandler : AMActorHandler<User, Actor_MatchSucess_Ntt> {
    protected override void Run(User user, Actor_MatchSucess_Ntt message) {
        user.IsMatching = false;
        user.ActorID = message.GamerID;
        Log.Info($" 玩家 {user.UserID} 匹配成功");
    }
}
```

1.17 MailBoxComponent: 挂上这个组件表示该 Entity 是一个 Actor, 接收的消息将会队列处理

```
// 挂上这个组件表示该 Entity 是一个 Actor, 接收的消息将会队列处理
[ComponentOf]
public class MailBoxComponent: Entity, IAwake, IAwake<MailboxType> {
    // Mailbox 的类型
    public MailboxType MailboxType { get; set; }
}
```

1.18 【服务端】ActorHandleHelper 帮助类：连接上下层的中间层桥梁

- 读了 ActorMessageSenderComponentSystem.cs 的具体的消息内容处理、发送，以及计时器消息的超时自动抛超时错误码过滤等底层逻辑处理，
- 读上下面的顶层的 NetInnerComponentOnReadEvent.cs 的顶层某个某些服，读到消息后的消息处理逻辑
- 知道，当前帮助类，就是衔接上面的两条顶层调用，与底层具体处理逻辑的桥，把框架上中下层连接连通起来。
- 分析这个类，应该可以理解底层不同逻辑方法的前后调用关系，消息处理的逻辑模块先后顺序，以及必要的可能的调用频率，或调用上下文情境等。明天上午再看一下
- 是谁调用这个帮助类? **IMHandler 类的某些继承类**。我目前仍只总结和清楚了两个抽象继承类，但还不曾熟悉任何实现子类，要去弄那些，顺便把位置相关的也弄懂了
- 上面 **【ActorMessageSenderComponentSystem.cs】的使用情境**：有个 **【服务端热更新的帮助】类 MessageHelper.cs**, 发 Actor 消息，与 ActorLocation 位置消息，也会都是调用 ActorMessageSenderComponentSystem.cs 里定义的底层逻辑。

```
public static class ActorHandleHelper {
    public static void Reply(int fromProcess, IActorResponse response) {
        if (fromProcess == Options.Instance.Process) { // 返回消息是同一个进程: 没明白，这里为什么就断定是同一进程的消息了
            // NetInnerComponent.Instance.HandleMessage(realActorId, response); // 等同于直接调用下面这句【我自己暂时放
            ActorMessageSenderComponent.Instance.HandleIActorResponse(response); // 【没读懂:】同一个进程内的消息，不走
            return;
        }
        // 【不同进程的消息处理:】走网络层，就是调用会话框来发出消息
        Session replySession = NetInnerComponent.Instance.Get(fromProcess); // 从内网组件单例中去拿会话框: 不同进程消息,
        replySession.Send(response);
    }
    public static void HandleIActorResponse(IActorResponse response) {
        ActorMessageSenderComponent.Instance.HandleIActorResponse(response);
    }
    // 分发 actor 消息
    [EnableAccessEntiyChild]
    public static async ETTask HandleIActorRequest(long actorId, IActorRequest iActorRequest) {
```

```
        InstanceIdStruct instanceIdStruct = new(actorId);
        int fromProcess = instanceIdStruct.Process;
        instanceIdStruct.Process = Options.Instance.Process;
        long realActorId = instanceIdStruct.ToLong();
        Entity entity = Root.Instance.Get(realActorId);
        if (entity == null) {
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Warning($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorRequest}");
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId) {
                        IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundAct
                        Reply(fromProcess, response);
                        break;
                    } // 调用管理器组件的处理方法
                    await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorRequest);
                }
                break;
            }
            case MailboxType.UnOrderMessageDispatcher: {
                await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorRequest);
                break;
            }
            case MailboxType.GateSession:
            default:
                throw new Exception($"no mailboxtype: {mailBoxComponent.MailboxType} {iActorRequest}");
        }
    }
    // 分发 actor 消息
    [EnableAccessEntiyChild]
    public static async ETTask HandleIActorMessage(long actorId, IActorMessage iActorMessage) {
        InstanceIdStruct instanceIdStruct = new(actorId);
        int fromProcess = instanceIdStruct.Process;
        instanceIdStruct.Process = Options.Instance.Process;
        long realActorId = instanceIdStruct.ToLong();
        Entity entity = Root.Instance.Get(realActorId);
        if (entity == null) {
            Log.Error($"not found actor: {realActorId} {iActorMessage}");
            return;
        }
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Error($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorMessage}");
            return;
        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId)
                        break;
                    await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                }
                break;
            }
            case MailboxType.UnOrderMessageDispatcher: {
                await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                break;
            }
            case MailboxType.GateSession: {
                if (entity is Session gateSession)
                    // 发送给客户端
                    gateSession.Send(iActorMessage);
                break;
            }
            default:
```

```
                throw new Exception($"no mailboxtype: {mailBoxComponent.MailboxType} {iActorMessage}");
            }
        }
    }
```

1.19 NetInnerComponentOnReadEvent:

- 框架相对顶层的：某个某些服，读到消息后，发布读到消息事件后，触发的消息处理逻辑
- 这个，应该是服务端发布读事件后，触发的订阅者处理读到消息的回调逻辑：分消息类型，进行不同的处理

```
// 这个，应该是服务端发布读事件后，触发的订阅者处理读到消息的回调逻辑：分消息类型，进行不同的处理
[Event(SceneType.Process)]
public class NetInnerComponentOnReadEvent: AEvent<NetInnerComponentOnRead> {
    protected override async ETTask Run(Scene scene, NetInnerComponentOnRead args) {
        try {
            long actorId = args.ActorId;
            object message = args.Message;
            // 收到 actor 消息，放入 actor 队列
            switch (message) { // 分不同的消息类型，借助 ActorHandleHelper 帮助类，对消息进行处理。既处理【请求消息】，也处理【返回
                case IActorResponse iActorResponse: {
                    ActorHandleHelper.HandleIActorResponse(iActorResponse);
                    break;
                }
                case IActorRequest iActorRequest: {
                    await ActorHandleHelper.HandleIActorRequest(actorId, iActorRequest);
                    break;
                }
                case IActorMessage iActorMessage: {
                    await ActorHandleHelper.HandleIActorMessage(actorId, iActorMessage);
                    break;
                }
            }
        }
        catch (Exception e) {
            Log.Error($"InnerMessageDispatcher error: {args.Message.GetType().Name}\n{e}");
        }
        await ETTask.CompletedTask;
    }
}
```

2 Net 网络交互相关：【服务端 + 客户端】只是稍微改装成事件机制。模块没理解透、总结不全，还需要借助总结，和改掉所有编译错误后的运行、以及运行日志，来理解这个流程

- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
- 感觉核心逻辑，跨进程发消息，收返回消息，基本都看懂了。更底层的，可是相对高层？的服务之间，【NetThreadComponent 组件】等，仍是不懂。
- 这个模块：感觉就是 **【模块，自顶向下，异步网络调用的传递方向等，弄不懂；或底层信道上发消息两端的底层回调，不懂!】**
- 现在还没弄清楚：Server, Client, Inner, 好像没有 Outer 了，几个相对模块算是怎么回事？
- 不管是【网络服务端 NetServerComponent】，还是【网络客户端 NetClientComponent】组件，它们都管理无数个与【这个端】建立连接的【会话框】。

2.1 RpcInfo: 【消息的包装体】。内部包装一个 Tcs 异步任务，桥接异步结果给调用方。合并入其它小节

- 结合 NetServerComponentOnReadEvent 来读。
- 在 NetServerComponentOnReadEvent 中，IResponse 【返回消息】是会话框上直接返回同步异步任务的异步结果，将【返回消息】异步给调用方。

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

2.2 NetThreadComponent:

```
namespace ET {
// 【NetThreadComponent 组件】：网络交互的底层原理不懂。没有生成系，只有一个【NetInnerComponentSystem】。外网组件找不见
// 这个模块：感觉就是模块，自顶向下，异步网络调用的传递方向等，弄不懂；或底层信道上发消息两端的底层回调，不懂！
// 是每个场景【SceneType?】：里都必须有的异步线程组件. 场景 Scene, 与场景类型 SceneType
    [ComponentOf(typeof(Scene))]
    public class NetThreadComponent: Entity, IAwake, ILateUpdate, IDestroy {
        [StaticField]
        public static NetThreadComponent Instance; // 单例
        public Thread thread;
        public bool isStop;
    }
}
```

2.3 NetServerComponent: NetServerComponentOnRead 结构体。

- 【必须去想】:【服务端】到底是什么？不是每个进程上的什么东西，而是每个【场景 Scene】所启动的该场景上的服务类型。不同场景之间的服务类型，可以不同？
- 证实这一点儿的【半确认：因为这个组件可能是我从参考项目里，我自己搬过来的的】依据，是有【Realm 注册登录服】和【Gate 网关服】添加了这个组件。
- 不同结构体的封装，是根据需要来的。框架里，有封装过【Session 会话框】的，[rpcId] 的，【Tcs 异步任务】的。看需求。

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

2.4 NetServerComponentSystem: 场景上的【服务端】组件，可发布【服务端读到消息事件】

- 【生成系】重点：它可以 **发布 NetServerComponentOnRead 事件**。理解这个事件的【发布】与【订阅者回调】的过程，如下：
  - 一个核一个进程上，可能有的【1-N】个场景中，某个场景充当【服务端】发布了该事件。当前核上的这个场景发布事件的触发原因是：主线程回调到了这个场景（网络异步线程）读到消息事件？（这里的主线程，与异步线程，想起来仍奇怪。可是同一核同一进程里，就只能多线程，每个线程当作一个场景了）

  - 【事件的订阅者】：进程上的 NetServerComponentOnReadEvent
  - 进程被【1-N】个不同场景共享，是更底层。这里发出事件，【消息的接收者】，可能在【同一进程其它场景】，也可能在【其它进程】其它场景
  - 这里，【事件发布】到【事件订阅者】的过程，更像是，由某个场景，到【1-N】个可能场景所共享的，更底层的对应核，的过程
  - 【1-N】个可能场景所共享的，更底层的【这一个对应核】：订阅了事件。处理逻辑：是本进程的场景，接收场景去处理；不同进程？rpc 。。。

- 现在，我先把它想成是：一个进程可以有的【1-N】个场景中，每个场景，所启动的服务类型。不同场景，所启动的服务类型，应该可以不同？

- 当这个场景充当了【服务端】，其它所有与这个【当前场景服务端】建立的会话框的另一端，就都自动当作【客户端】。（感觉这里理解不太透，暂时仍这么想）

```
[FriendOf(typeof(NetServerComponent))] // 【服务端组件】：负责【服务端】的网络交互部分
public static class NetServerComponentSystem {
    [ObjectSystem]
    public class AwakeSystem: AwakeSystem<NetServerComponent, IPEndPoint> {
        protected override void Awake(NetServerComponent self, IPEndPoint address) {
            // 当一个【场景启动】起来，向 NetServices 单例总管，注册三大回调。当向总管注册三回调的时候，它，不是相当于是总管
            // 更像是，【单线程多进程架构】里，异步网络线程，向主线程，注册三大回调
            self.ServiceId = NetServices.Instance.AddService(new KService(address, ServiceType.Outer));
            NetServices.Instance.RegisterAcceptCallback(self.ServiceId, self.OnAccept); // 三个回调
            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) {
        // 【创建会话框】：当此【服务端】组件，接受了一个客户端，就建一个与接收的【客户端】的会话框
        Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
        session.RemoteAddress = ipEndPoint; // 【当前会话框】，它的远程是，一个【客户端】的 IP 地址
        if (self.DomainScene().SceneType != SceneType.BenchmarkServer) { // 区分：同一功能，【服务端】的处理逻辑，与【客户
            // 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
            // 【客户端】逻辑，客户端验证的地方：C2G_LoginGateHandler: 这个例子，当前自称服务端组件，才更像【客户端】呢
            session.AddComponent<SessionAcceptTimeoutComponent>(); // 上面原标注：【客户端验证】的逻辑
            // 客户端连接，2 秒检查一次 recv 消息，10 秒没有消息则断开（与那个，此服务端接收不到心跳包的客户端，的连接）。【活
            //【自己的理解】：【客户端】有心跳包告知服务端，各客户端的连接状况；【服务端】：同样有服务端此组件来检测说，哪个客户端
            session.AddComponent<SessionIdleCheckerComponent>(); // 检查【会话框】是否有效：【30 秒内】至少发送过消息，至
        }
    }
    // 从这里继续往前倒，去找哪里发布事件， message 是什么类型，什么内容？【这里就是不懂】
    private static void OnRead(this NetServerComponent self, long channelId, long actorId, object message) {
        Session session = self.GetChild<Session>(channelId); // 从当前【服务端】所管理的所有会话框（连接的所有客户端）里，
        if (session == null) return;
        session.LastRecvTime = TimeHelper.ClientNow();
        OpcodeHelper.LogMsg(self.DomainZone(), message);
        // 【发布事件】：服务端组件读到了消息。
        EventSystem.Instance.Publish(Root.Instance.Scene, new NetServerComponentOnRead() {Session = session, Message
        // 【事件的订阅者】：进程上的 NetServerComponentOnReadEvent
        // 进程被【1-N】个不同场景共享，是更底层。这里发出事件，【消息的接收者】，可能在【同一进程其它场景】，也可能在【其它进程
        // 这里，【事件发布】到【事件订阅者】的过程，更像是，由某个场景，到【1-N】个可能场景所共享的，更底层的对应核，的过程
        // 【1-N】个可能场景所共享的，更底层的【这一个对应核】：订阅了事件。处理逻辑：是本进程的场景，接收场景去处理；不同进程
    }
}
```

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

- 框架里什么地方添加了这些【NetServerComponent 服务端】的组件？Realm 注册登录服，和网关服。（虽然这两个小服添加了这个服务端组件，但还不知道，是不是自己干的好事儿！！）SceneFactory 类里的。也就是说：这个组件，是有可能，重构后的框架里，是不需要的？都是自己没能把源码管理好，给混的。。

```
case SceneType.Realm: // 注册登录服：
    scene.AddComponent<NetServerComponent, IPEndPoint>(startSceneConfig.InnerIPOutPort);
    break;
case SceneType.Gate:
    scene.AddComponent<NetServerComponent, IPEndPoint>(startSceneConfig.InnerIPOutPort);
    scene.AddComponent<PlayerComponent>();
    scene.AddComponent<GateSessionKeyComponent>();
    break;
```

- 如果消息类型是【返回消息】：就【会话框】上，调用会话框的 OnResponse() 方法处理。处理逻辑，也就是把（来自同一进程其它场景，或来自其它进程的【并不能限定只来自于本进程】）返回的【返回消息】内容，同步到封装的 Tcs 异步结果里。当异步正常结果写好，框架的异步封装，就自动实现了异步结果、异步回给调用方（逻辑在调用【发送消息】发送过程的方法里）。

- 这个类里，对于其它消息类型，上次并没能读完整和理解透彻：就是那个【发送位置消息请求】的请求者，与【被索要位置信息】的被请求者，协程锁，锁的是哪个？现在，感觉两个都可以上锁，可是两个都锁、都有必要吗？是的，框架里是两方都上锁的，既锁向位置服要地址的发送者，也锁被要地址小伙伴所在进程，锁在【进程】层面上？

  - 【位置服】里，被请求位置信息的，同时间可以有多个不同进程的索要者，要上锁；
  - 请求消息的发送者，同时间，有什么多进程，同时要它发消息的情况？（这里暂时想不出来）理论是客观存在的。多进程队列安全，就得上锁。意思是说，多个不同进程，都想要入队列要当前 actorId 发送消息，只能按照分配给它们的【独占锁】的先后顺序入队、修改共享队列里的消息内容（这里是添加消息）

- 判断【位置消息】里的 actorId，是发送者的，还是被请求者的，去找消息发送之前，消息创建的地方。看框架能否找到一个例子。现在就是找不到一个这样的真正发送出去的位置消息的例子

- 【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！！！】

```
// 为什么 Realm 注册登录服，与 Gate 网关服里【服务端】组件发布的事情，会有这个场景的订阅者接收事件？
// 【SceneType.Process】：需要特殊理解，极为特殊的进程场景。它是每个核每个进程必备的一个特殊场景吗？是。Root 单根，首先启动进程场景。
[Event(SceneType.Process)]  // 【进程】场景？来处理这个服务端组件事件？外网组件添加的地方是在：【Realm 注册登录服】与【网关服】。是
public class NetServerComponentOnReadEvent: AEvent<NetServerComponentOnRead> {
    protected override async ETTask Run(Scene scene, NetServerComponentOnRead args) {
        Session session = args.Session;
        object message = args.Message;
        // 【服务端上，会话框】Session：收到回复消息，会去处理【会话框】上字典管理的回调，将回调的 Tcs 异步结果写好。写好了，即刻异
        if (message is IResponse response) { // 到达本进程的【返回消息】：本进程上将结果写回去，狠简单
            // 借由 Tcs 异步，会话框上会同步【返回消息】的内容到 Tcs 异步任务的结果；Tcs 任务结果一旦写好，消息请求方就能收到结果
            session.OnResponse(response);
            return;
        }
        // 根据消息接口判断是不是 Actor 消息，不同的接口做不同的处理，比如需要转发给 Chat Scene，可以做一个 IChatMessage 接口
        switch (message) { //  【发送消息】+【不要求回复的消息】
            //  【ActorLocationSenderComponent】：先把这一两个组件逻辑给理顺了
            case IActorLocationRequest actorLocationRequest: { // gate session 收到 actor rpc 消息，先向 actor 发送 rpc 请求，
                long unitId = session.GetComponent<SessionPlayerComponent>().PlayerId;
                int rpcId = actorLocationRequest.RpcId; // 这里要保存客户端的 rpcId
                long instanceId = session.InstanceId;
                IResponse iResponse = await ActorLocationSenderComponent.Instance.Call(unitId, actorLocationRequest); //  【
                iResponse.RpcId = rpcId; //  【发送消息】与【返回消息】的 rpcId 是一样的。可是这里的设置，感觉狠奇怪。【位置服】是怎
                // session 可能已经断开了，所以这里需要判断
```

```
                if (session.InstanceId == instanceId)
                    session.Send(iResponse);
                break;
            }
            case IActorLocationMessage actorLocationMessage: { //  【普通，不要求回复的位置消息】
                long unitId = session.GetComponent<SessionPlayerComponent>().PlayerId;
                ActorLocationSenderComponent.Instance.Send(unitId, actorLocationMessage); // 把这里发送位置消息再看一遍，快速
                break;
            }
            case IActorRequest actorRequest:  // 分发 IActorRequest 消息，目前没有用到，需要的自己添加
                break;
            case IActorMessage actorMessage:  // 分发 IActorMessage 消息，目前没有用到，需要的自己添加
                break;
        default: { // 非 Actor 消息的话：应该就是本进程消息，不走网络层，进程内处理
                // 非 Actor 消息：  MessageDispatcherComponent 全局单例吗？是的
            MessageDispatcherComponent.Instance.Handle(session, message);
                break;
            }
        }
    }
}
```

2.6 Option 单例类：

- 上面，留了一个不懂的地方：一台物理机上同一个核，同一个进程内，的【多线程多场景管理】里，为什么有一个专用的 SceneType.Process. 这个场景如【亲爱的表哥在活宝妹心中的地位一样特殊】，要把这个理解透彻。现在把这个翻一遍。
- 理解上，在同一进程的多线程管理里，是会区分【主线程】与【异步网络线程】的。这个 SceneType.Process 像是【主线程】，需要处理【本核本进程内，多线程管理，主线程与异步线程的同步等逻辑】，也负责处理【多核多进程间，或与其它物理机等的网络交互】等主线程逻辑；而任意（可能受一个进程所可以开辟的多线程数目，硬件限制?）添加的【0-N】个任务线程，充当框架里可以随时再添加的同一进程上的【其它场景 SceneType】。
- SceneType.Process: 每个核、进程上的【进程场景】
- OptionAttribute: 命令行的选项标签。这里似乎也看不出什么来。

```
public class Options: Singleton<Options> { // 这个【单例类】，确实还没能看懂。单例类，不是组件添加形式。把【OptionAttribute】标签
    [Option("AppType", Required = false, Default = AppType.Server, HelpText = "AppType enum")]
    public AppType AppType { get; set; }
    [Option("StartConfig", Required = false, Default = "StartConfig/Localhost")]
    public string StartConfig { get; set; }
    [Option("Process", Required = false, Default = 1)]
    public int Process { get; set; }

    [Option("Develop", Required = false, Default = 0, HelpText = "develop mode, 0 正式 1 开发 2 压测")]
    public int Develop { get; set; }
    [Option("LogLevel", Required = false, Default = 2)]
    public int LogLevel { get; set; }
    [Option("Console", Required = false, Default = 0)]
    public int Console { get; set; }
    // 进程启动是否创建该进程的 scenes
    [Option("CreateScenes", Required = false, Default = 1)]
    public int CreateScenes { get; set; }
}
```

- 框架里第一次调用【Options 单例类】实例的地方，也就是这个【单例类】的初始化的过程，看一下。
  - 第一次调用的地方，是在双端框架的 Init 类里。先把这里的源码放一点儿，再去找哪里调双端的 Init.cs? Program.cs 里会调这个类的 Init.Start() 方法。同一个 Init 类找出三个文件来。
- 比如【客户端】起始的时候，命令行，Init 里会去 Parse 命令行里传进来的参数，放将命令行的参数配置写入记入到 Options 单例类里。后来看见过的，也是用这个单例类里的 Process 来判定，比如【返回消息】是否为【本进程消息】等判定进程是否为同一个。

- 只是上面，进程是一个核。现在能想到的是命令行启动一台物理机 N 个核，每个核也是可以命令行单独、以核为单位配制的?【这里不懂，框架里，是如何封装，命令行来启动一个核的?】

```
public static class Init {
    public static void Start() {
        try {
            AppDomain.CurrentDomain.UnhandledException += (sender, e) => {
                Log.Error(e.ExceptionObject.ToString());
            };
            // 异步方法全部会回掉到主线程
            Game.AddSingleton<MainThreadSynchronizationContext>();
            // 命令行参数
            Parser.Default.ParseArguments<Options>(System.Environment.GetCommandLineArgs())
                .WithNotParsed(error => throw new Exception($" 命令行格式错误! {error}"))
                .WithParsed(Game.AddSingleton);

            Game.AddSingleton<TimeInfo>();
            Game.AddSingleton<Logger>().ILog = new NLogger(Options.Instance.AppType.ToString(), Options.Instance.Process,
            Game.AddSingleton<ObjectPool>();
            Game.AddSingleton<IdGenerater>();
            Game.AddSingleton<EventSystem>();
            Game.AddSingleton<TimerComponent>();
            Game.AddSingleton<CoroutineLockComponent>();

            ETTask.ExceptionHandler += Log.Error;
            Log.Console($"{Parser.Default.FormatCommandLine(Options.Instance)}");
            Game.AddSingleton<CodeLoader>().Start();
        } catch (Exception e) {
            Log.Error(e);
        }
    }
```

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

- 先去看：框架里，什么上下文添加了这个组件?【客户端组件】一定是添加在【客户端】。【客户端场景 ClientScene】添加有这个组件。
- 什么是【客户端】?

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

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

```
[FriendOf(typeof(NetClientComponent))] // 把这个【网络客户端】组件的主要笔记要点，再快速写一遍
public static class NetClientComponentSystem {
    [ObjectSystem]
    public class AwakeSystem: AwakeSystem<NetClientComponent, AddressFamily> {
        protected override void Awake(NetClientComponent self, AddressFamily addressFamily) { // 需要什么样的参数，就传什么样
            self.ServiceId = NetServices.Instance.AddService(new KService(addressFamily, ServiceType.Outer)); // 开启了与这个
            NetServices.Instance.RegisterReadCallback(self.ServiceId, self.OnRead); // 注册订阅【读】网络消息事件，应该是从网络
            NetServices.Instance.RegisterErrorCallback(self.ServiceId, self.OnError); // 注册订阅【出错】事件
        }
    }
    [ObjectSystem]
    public class DestroySystem: DestroySystem<NetClientComponent> {
        protected override void Destroy(NetClientComponent self) {
            NetServices.Instance.RemoveService(self.ServiceId); // 直接移除这个网络服务
        }
    }
    private static void OnRead(this NetClientComponent self, long channelId, long actorId, object message) {
        Session session = self.GetChild<Session>(channelId); // 拿：相应的会话框
```

```
        if (session == null) { // 空：直接返回
            return;
        }
        session.LastRecvTime = TimeHelper.ClientNow();
        OpcodeHelper.LogMsg(self.DomainZone(), message);
// 发布事件：事件的接收者，应该是【客户端】的 Session 层面的进一步读取消息内容（内存流上读消息？），改天再去细看。
        EventSystem.Instance.Publish(Root.Instance.Scene, new NetClientComponentOnRead() {Session = session, Message = mess
    }
    private static void OnError(this NetClientComponent self, long channelId, int error) {
        Session session = self.GetChild<Session>(channelId); // 同样，先去拿会话框：因为这些异步网络的消息传递，都是建立在一个个
        if (session == null)  // 空：直接返回
            return;
        session.Error = error;
        session.Dispose();
    }
    public static Session Create(this NetClientComponent self, IPEndPoint realIPEndPoint) {
        long channelId = NetServices.Instance.CreateConnectChannelId();
        Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId); // 创建必要的会话框，方便交通
        session.RemoteAddress = realIPEndPoint;
        if (self.DomainScene().SceneType != SceneType.Benchmark) {
            session.AddComponent<SessionIdleCheckerComponent>(); // 不知道这个是干什么的，改天再看
        }
        NetServices.Instance.CreateChannel(self.ServiceId, session.Id, realIPEndPoint); // 创建信道
        return session;
    }
    public static Session Create(this NetClientComponent self, IPEndPoint routerIPEndPoint, IPEndPoint realIPEndPoint, uint
        long channelId = localConn;
        Session session = self.AddChildWithId<Session, int>(channelId, self.ServiceId);
        session.RemoteAddress = realIPEndPoint;
        if (self.DomainScene().SceneType != SceneType.Benchmark) {
            session.AddComponent<SessionIdleCheckerComponent>();
        }
        NetServices.Instance.CreateChannel(self.ServiceId, session.Id, routerIPEndPoint);
        return session;
    }
}
```

2.9 NetClientComponentOnReadEvent: 【网络客户端】读到消息事件：它，要如何处理读到的消息呢？

```
[Event(SceneType.Process)] // 作用单位：进程【一个核】。一个进程可以有多个不同的场景。
public class NetClientComponentOnReadEvent: AEvent<NetClientComponentOnRead> { // 事件 NetClientComponentOnRead 的发出者是：
    protected override async ETTask Run(Scene scene, NetClientComponentOnRead args) {
        Session session = args.Session;
        object message = args.Message;
        if (message is IResponse response) {  // 【返回消息】：待同步结果到 Tcs
            session.OnResponse(response); // 【会话框】上将【返回消息】写入、同步到 Tcs 异步任务的结果中去
            return;
        }
        // 【普通消息或者是 Rpc 请求消息?】：前面我写得对吗？这里说，【网络客户端组件】读到消息事件，接下来，分配到相应【会话框场景】去
        MessageDispatcherComponent.Instance.Handle(session, message);
        await ETTask.CompletedTask;
    }
}
```

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

- 服务端的内网组件：这个组件，要想一下，同其它组件有什么不同?

```
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;
    }
}
```

2.11 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;
        }
    }
```

2.12 MessageDispatcherInfo: 在【MessageDispatcherComponent】中

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

```
// 总管：对每个场景 SceneType，消息分发器
// 这个类，可以简单地理解为：先前的各种服，现在的各种服务端场景，它们所拥有的消息处理器实例的封装。
// 那么默认，每种场景，只有一个消息处理器实体类 ( 可以去验证这点儿 )
public class MessageDispatcherInfo {
    public SceneType SceneType { get; }
    public IMHandler IMHandler { get; }
    public MessageDispatcherInfo(SceneType sceneType, IMHandler imHandler) {
        this.SceneType = sceneType;
        this.IMHandler = imHandler;
    }
}
// 消息分发组件
[ComponentOf(typeof(Scene))]
public class MessageDispatcherComponent: Entity, IAwake, IDestroy, ILoad {
// 按下面的字典看，消息分发器，全局单例，是的!【活宝妹就是一定要嫁给亲爱的表哥!!】
    public static MessageDispatcherComponent Instance { get; set; } // 【全局单例】
    public readonly Dictionary<ushort, List<MessageDispatcherInfo>> Handlers = new(); // 总管的字典
}P
```

- 这个组件全局单例，添加的地主是在框架服务器启动的时候，公共组件部分的添加。组件的字典，会管理全框架下所有的 MessageDispatcherInfo 相产。
  - 来自于文件 EntryEvent1_InitShare:

```
// 公用的相关组件的初始化：
[Event(SceneType.Process)]
```

```
public class EntryEvent1_InitShare: AEvent<EventType.EntryEvent1> {
    // 【全局单例】组件:
    protected override async ETTask Run(Scene scene, EventType.EntryEvent1 args) {
        Root.Instance.Scene.AddComponent<NetThreadComponent>();
        Root.Instance.Scene.AddComponent<OpcodeTypeComponent>();
        Root.Instance.Scene.AddComponent<MessageDispatcherComponent>(); // <<<<<<<<<<<<<<<<<<<<
        Root.Instance.Scene.AddComponent<NumericWatcherComponent>();
        Root.Instance.Scene.AddComponent<AIDispatcherComponent>();
        Root.Instance.Scene.AddComponent<ClientSceneManagerComponent>();
        await ETTask.CompletedTask;
    }
}
```

2.14 MessageDispatcherComponentSystem:

```
// 扫描框架里的标签系【MessageHandler(SceneType)】
private static void Load(this MessageDispatcherComponent self) {
    self.Handlers.Clear();
    HashSet<Type> types = EventSystem.Instance.GetTypes(typeof (MessageHandlerAttribute));
    foreach (Type type in types) {
        IMHandler iMHandler = Activator.CreateInstance(type) as IMHandler;
        if (iMHandler == null) {
            Log.Error($"message handle {type.Name} 需要继承 IMHandler");
            continue;
        }
        object[] attrs = type.GetCustomAttributes(typeof(MessageHandlerAttribute), false);
        foreach (object attr in attrs) {
            MessageHandlerAttribute messageHandlerAttribute = attr as MessageHandlerAttribute;
            Type messageType = iMHandler.GetMessageType();
            ushort opcode = NetServices.Instance.GetOpcode(messageType); // 这里相对、理解上的困难是: 感觉无法把 OpCode 网络操
            if (opcode == 0) {
                Log.Error($" 消息 opcode 为 0: {messageType.Name}");
                continue;
            } // 下面: 下面是创建一个包装体, 注册备用
            MessageDispatcherInfo messageDispatcherInfo = new (messageHandlerAttribute.SceneType, iMHandler);
            self.RegisterHandler(opcode, messageDispatcherInfo);
        }
    }
}
private static void RegisterHandler(this MessageDispatcherComponent self, ushort opcode, MessageDispatcherInfo handler) {
    if (!self.Handlers.ContainsKey(opcode))
        self.Handlers.Add(opcode, new List<MessageDispatcherInfo>());
    self.Handlers[opcode].Add(handler); // 加入管理体系来管理
}
public static void Handle(this MessageDispatcherComponent self, Session session, object message) {
    List<MessageDispatcherInfo> actions;
    ushort opcode = NetServices.Instance.GetOpcode(message.GetType());
    if (!self.Handlers.TryGetValue(opcode, out actions)) {
        Log.Error($" 消息没有处理: {opcode} {message}");
        return;
    }
    // 这里就不明白: 它的那些 Domain 什么的
    SceneType sceneType = session.DomainScene().SceneType; //  【会话框】: 哈哈哈, 这是会话框两端, 哪一端的场景呢？感觉像是会话框的
    foreach (MessageDispatcherInfo ev in actions) {
        if (ev.SceneType != sceneType)
            continue;
        try {
            ev.IMHandler.Handle(session, message); // 处理分派消息: 也就是调用 IMHandler 接口的方法来处理消息
        } catch (Exception e) {
            Log.Error(e);
        }
    }
}
```

2.15 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);
            }
        }
    }
}
```

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

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

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

2.17 SessionIdleCheckerComponentSystem: SessionIdleChecker 激活类，

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

```
    [Invoke(TimerInvokeType.SessionIdleChecker)]
    public class SessionIdleChecker: ATimer<SessionIdleCheckerComponent> {
        protected override void Run(SessionIdleCheckerComponent self) {
            try {
                self.Check();
            } catch (Exception e) {
                Log.Error($"move timer error: {self.Id}\n{e}");
            }
        }
    }
    [ObjectSystem]
    public class SessionIdleCheckerComponentAwakeSystem: AwakeSystem<SessionIdleCheckerComponent> {
        protected override void Awake(SessionIdleCheckerComponent self) {
            // 同样设置：【重复闹钟】：任何时候，亲爱的表哥的活宝妹就是一定要嫁给亲爱的表哥!!!
            self.RepeatedTimer = TimerComponent.Instance.NewRepeatedTimer(SessionIdleCheckerComponentSystem.CheckInteral,
        }
    }// 。。。
    public static class SessionIdleCheckerComponentSystem {
        public const int CheckInteral = 2000; // 每隔 2 秒
        public static void Check(this SessionIdleCheckerComponent self) {
            Session session = self.GetParent<Session>();
            long timeNow = TimeHelper.ClientNow();
            // 常量类定义：会话框最长每个 30 秒；
            // 判断：30 秒内，曾经发送过消息，并且也接收过消息，直接返回；否则，算作【会话框】超时
            if (timeNow - session.LastRecvTime < ConstValue.SessionTimeoutTime && timeNow - session.LastSendTime < ConstV
                return;
            Log.Info($"session timeout: {session.Id} {timeNow} {session.LastRecvTime} {session.LastSendTime} {timeNow - s
            session.Error = ErrorCore.ERR_SessionSendOrRecvTimeout; //  【会话框】超时回收
            session.Dispose();
        }
    }
```

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

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

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

```
    }
}
```

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

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

```
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Error($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorMessage}");
            return;
        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId)
                        break;
                    await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                }
                break;
            }
            case MailboxType.UnOrderMessageDispatcher: {
                await ActorMessageDispatcherComponent.Instance.Handle(entity, fromProcess, iActorMessage);
                break;
            }
            case MailboxType.GateSession: {
                if (entity is Session gateSession)
                    // 发送给客户端
                    gateSession.Send(iActorMessage);
                break;
            }
            default:
                throw new Exception($"no mailboxtype: {mailBoxComponent.MailboxType} {iActorMessage}");
        }
    }
}
```

- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】

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

- 【服务端】启动时的特殊管理，感觉先前不曾真正接触过服务端的活宝妹，总是对这块儿读不懂。这个章节是自己最近总结的这一块儿。上午顺着这个先前总结的思路，把服务端的这些起始配置再看一遍。希望能够多些收获，或像昨天晚上哪怕收获不多，读懂了心里会有确信，不用想也会知道这个模块的功能原理等。【如同，亲爱的表哥的活宝妹，永远不用想也知道，亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】

- 现在，先特殊重点理解：一台【服务端】物理机起来，N 核 N 进程里【主线程 Process 场景】的启动过程。这里，把前面读到过的配置里的几个区分联系起来的看：Machine, Process, Scene, Zone, 物理机、多核多进程、每个核每个进程可以有多场景，**【Zone 不明白】**活宝妹想把 gmail.coml 里该死的几个数字去掉隐藏掉，看见它们烦人。。。【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】

- 【服务端、各服务器的配置、启动初始化】：是这个模块想要总结的内容。这个模块，因为框架重构里所接入的【路由器系统】的整合（感觉起来，就是通过网络，一台台服务端的服务器起来，一台台起来的服务器都向某个路由服，如同各客户端实时向位置服更新客户端的位置信息般，各小服专职服都向路由服上班打卡？要把这些看明白），让活宝妹理解起这个模块来显得相对困难，大概明天上午一上午的时间，都会花在这个模块上。

3.1 OptionAttribute: 系统里的标签属性。

- 那么就是，命令行相关的。也就是 ET 框架里所封装的，可以命令行，发布命令，启动【服务端】时的配置相关。【快看懂了吧。爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】

```
namespace CommandLine {
    // 【CommandLine】: 那么就是，命令行相关的。也就是 ET 框架里所封装的，可以命令行，发布命令，启动【服务端】时的配置相关
    [AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
    public sealed class OptionAttribute : BaseAttribute {
        public OptionAttribute();
        public OptionAttribute(string longName);
        public OptionAttribute(char shortName);
        public OptionAttribute(char shortName, string longName);
        public string LongName { get; }
        public string ShortName { get; }
        public string SetName { get; set; }
        public char Separator { get; set; }
        public string Group { get; set; }
    }
}
```

3.2 Options 单例类：

- 不知道这个单例类，是什么时候生成，什么情况下值会发生变化？跟命令行相关的话，当命令行启动【服务端】时，会启动这个【单例类】吗？网上搜下。

```
public class Options: Singleton<Options> { // 这个【单例类】，确实还没能看懂。单例类，不是组件添加形式。把【OptionAttribut
    [Option("AppType", Required = false, Default = AppType.Server, HelpText = "AppType enum")]
    public AppType AppType { get; set; }
    [Option("StartConfig", Required = false, Default = "StartConfig/Localhost")]
    public string StartConfig { get; set; }
    [Option("Process", Required = false, Default = 1)]
    public int Process { get; set; }

    [Option("Develop", Required = false, Default = 0, HelpText = "develop mode, 0 正式 1 开发 2 压测")]
    public int Develop { get; set; }
    [Option("LogLevel", Required = false, Default = 2)]
    public int LogLevel { get; set; }

    [Option("Console", Required = false, Default = 0)]
    public int Console { get; set; }
    // 进程启动是否创建该进程的 scenes
    [Option("CreateScenes", Required = false, Default = 1)]
    public int CreateScenes { get; set; }
}
```

3.3 模块里所用到的几个。NET 里的接口，以及自定义的框架底层辅助体系类等

3.3.1 ISupportInitialize: 【初始化】的支持接口，就是提供了【初始化之前】【初始化之后】的回调，两个 API

```
namespace System.ComponentModel {
    public interface ISupportInitialize {
        void BeginInit();
        void EndInit();
    }
}
```

3.3.2 IInvoke: 抽象类会在事件系统 EventSystem.cs 中被用到

```
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);
}
```

3.3.3 ISingleton 单例类接口：框架最底层，有狠多必要的单例类包装，统一实现这个单例接口，就是抽象提纯到框架最底层封装

```
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() {
    }
}
```

3.3.4 IMerge: 在 Proto 相关的地方，某些类如 StartProcessConfig.cs 会实现这个接口，进程中以消息的形式传递这部分原理也要弄懂

- 这个接口，框架里定义了，主要用来帮助实现【动态路由】的。动态路由：网络中的路由器彼此之间互相通信，传递各自的路由信息，利用收到的路由信息来『自动合并』更新和维护自己路由表的过程。【动态路由特点】：自动化程度高，减少管理任务，错误率较低，但是占用网络资源。

- 它定义了一个合并接口。因为这模块类中的诸多 Protobuf 相关的标签，活宝妹想，它们应该是可以以消息的形式进程间传递的。

- 那么如果服务端的配置可以以消息的形式进程间传递，它合并时，谁与谁，如何合并的？感觉狠复杂的样子，要解一解。。。它是用在【动态路由系统】的模块。当一个路由器自动每 10 分钟周期性去扫描周围是否存在路由器邻居的时候，会自动合并。用行话说是，动态路由是网络中路由器之间互相通信，传递路由信息，利用收到的路由信息更新路由表的过程。这里【更新路由表】，说的就是当扫到了周围存在的路由器邻居，就更新自己当前路由器的路由表 Info 成员变量。

- 它能实时的适应网络结构的变化。如果路由更新信息表明网络发生了变化，路由选择软件就会重新计算路由，并发出新的路由更新信息。这些信息通过各个网络，引起各路由器重新启动其路由算法，并更新各自的路由表以动态的反映网络拓扑的变化。

- 因为关于进程间消息自动合并？的这一块儿不懂，可以去找一下，什么情况下会调用这个合并？

```
public interface IMerge {
    void Merge(object o);
}
```

3.4 ProtoObject: 继承自上面的系统接口，定义必要的回调抽象 API

```
public abstract class ProtoObject: Object, ISupportInitialize {
    public object Clone() { //  【进程间可传递的消息】: 为什么这里的复制过程，是先序列化，再反序列化？
        // 不明白: 消息明明就是反序列化好的，为什么再来一遍: 序列化、反序列化（虽然这个再一遍的过程是 ProtoBuf 里的序列化与反序列化
        // 翻到 Protobuf 里的反序列化方法，去查看: ET 框架的封装里，
            // 在底层内存流上的反序列化方法时（ProtobufHelper.Deserialize()），会调用 ISupportInitialize 的 EndInit() 回调，序列
            // 序列化前的回调，是哪里调用的？BeginInit() 回调在框架里，只有在 MongoHelper.cs 的 Json 序列化前，会调用；ProtoBuf
            // 就是提供了两个接口: 调用与不调用，还是分不同的序列化工具
        byte[] bytes = SerializeHelper.Serialize(this);
        return SerializeHelper.Deserialize(this.GetType(), bytes, 0, bytes.Length);
    }
    public virtual void BeginInit() {
    }
    public virtual void EndInit() {
    }
    public virtual void AfterEndInit() { // 这个回调，与上一个 EndInit() 区别是？
    }
}
```

3.5 ConfigLoader.cs: 【服务端】是理解接下来部分的基础。【客户端】有不同逻辑。所以要把两边的都看一下

- 这个类名奇怪的地方是：它明明是定义了两个 Invoke 标签事件的触发回调逻辑，为什么它的名字叫的是 ConfigLoader? 感觉是扫描程序域里所有的【Config】标签一样。。。
- 【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!!!】
- 这个文件的 GetAllConfigBytes 类中的回调：会去事件系统拿程序域里所有标记【Config】标签的类型，并根据这些标签类型是否为四大单例类之一来确认读取配置的位置。就是四个单例管理类的配置位置会相对特殊一点儿。

```
[Invoke] // 激活系: 这个激活系是同属 ET 强大的事件系统的一个标签和回调逻辑，处理两种类型： GetAllConfigBytes 和 GetOneConfigBytes
public class GetAllConfigBytes: AInvokeHandler<ConfigComponent.GetAllConfigBytes, Dictionary<Type, byte[]>> {
    public override Dictionary<Type, byte[]> Handle(ConfigComponent.GetAllConfigBytes args) {
        Dictionary<Type, byte[]> output = new Dictionary<Type, byte[]>();
        List<string> startConfigs = new List<string>() {
            "StartMachineConfigCategory",  // 涉及底层配置的几个单例类，为什么这四个单例类类型重要： Machine, Process 进程、Scen
            "StartProcessConfigCategory",
            "StartSceneConfigCategory",
            "StartZoneConfigCategory",
        };
// 类型: 这里，扫的是所有【Invoke】标签（好像不对），还是说如【Invoke(TimerInvokeType.ActorMessegaeSenderChecker)】之类的 Invoke
        HashSet<Type> configTypes = EventSystem.Instance.GetTypes(typeof (ConfigAttribute)); //  【Config】标签: 返回程序域里
        foreach (Type configType in configTypes) {
            string configFilePath;
            if (startConfigs.Contains(configType.Name)) { //  【单例管理类型】: 有特异性的配置路径
                configFilePath = $"../Config/Excel/s/{Options.Instance.StartConfig}/{configType.Name}.bytes";
            } else { // 其它: 人海里的路人甲，读下配置就扔掉
                configFilePath = $"../Config/Excel/s/{configType.Name}.bytes";
            }
            output[configType] = File.ReadAllBytes(configFilePath);
        }
        return output;
    }
}
[Invoke]
public class GetOneConfigBytes: AInvokeHandler<ConfigComponent.GetOneConfigBytes, byte[]> {
    public override byte[] Handle(ConfigComponent.GetOneConfigBytes args) {
        //  【Invoke 回调逻辑】: 从框架特定位置，读取特定属性条款的配置，返回字节数组
        byte[] configBytes = File.ReadAllBytes($"../Config/{args.ConfigName}.bytes");
```

```
        return configBytes;
    }
}
```

3.6 ConfigLoader:【客户端】

- 【客户端】与【服务端】不同的是，客户端需要区分当前的运行，是在编辑器模式下，还是真正运行在客户端设备（PC 平台）。编辑器模式下，如服务端，去特定的位置去读配置文件；而真正的客户端，就需要从热更新资源服务器（斗地主参考项目中，仍是有个其它语言的最小最精致热更新资源包专职服务器的，ET7 里好像没有了，而是放在一个特定的文件夹下?）服务端来下载配置资源包，读取资源包里的配置内容，并字典管理，

```
[Invoke]
public class GetAllConfigBytes: AInvokeHandler<ConfigComponent.GetAllConfigBytes, Dictionary<Type, byte[]>> {
    public override Dictionary<Type, byte[]> Handle(ConfigComponent.GetAllConfigBytes args) {
        Dictionary<Type, byte[]> output = new Dictionary<Type, byte[]>();
        HashSet<Type> configTypes = EventSystem.Instance.GetTypes(typeof (ConfigAttribute));
        if (Define.IsEditor) { //  【编辑器模式下】:
            string ct = "cs";
            GlobalConfig globalConfig = Resources.Load<GlobalConfig>("GlobalConfig"); // 加载全局模式: 这里没有看懂
            CodeMode codeMode = globalConfig.CodeMode;
            switch (codeMode) {
                case CodeMode.Client:
                    ct = "c";
                    break;
                case CodeMode.Server:
                    ct = "s";
                    break;
                case CodeMode.ClientServer:
                    ct = "cs";
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
            List<string> startConfigs = new List<string>() {
                "StartMachineConfigCategory",
                "StartProcessConfigCategory",
                "StartSceneConfigCategory",
                "StartZoneConfigCategory",
            };
            foreach (Type configType in configTypes) {
                string configFilePath;
                if (startConfigs.Contains(configType.Name)) {
                    configFilePath = $"../Config/Excel/{ct}/{Options.Instance.StartConfig}/{configType.Name}.bytes";
                } else {
                    configFilePath = $"../Config/Excel/{ct}/{configType.Name}.bytes";
                }
                output[configType] = File.ReadAllBytes(configFilePath);
            }
        } else {
            using (Root.Instance.Scene.AddComponent<ResourcesComponent>()) { // <<<<<<<<<<<<<<<<<<<<
                const string configBundleName = "config.unity3d";
                ResourcesComponent.Instance.LoadBundle(configBundleName);

                foreach (Type configType in configTypes) {
                    TextAsset v = ResourcesComponent.Instance.GetAsset(configBundleName, configType.Name) as TextAsset;
                    output[configType] = v.bytes;
                }
            }
        }
        return output;
    }
}
[Invoke]
public class GetOneConfigBytes: AInvokeHandler<ConfigComponent.GetOneConfigBytes, byte[]> {
    public override byte[] Handle(ConfigComponent.GetOneConfigBytes args) {
        // TextAsset v = ResourcesComponent.Instance.GetAsset("config.unity3d", configName) as TextAsset;
        // return v.bytes;
        throw new NotImplementedException("client cant use LoadOneConfig");
    }
}
```

3.7 ConfigComponent 组件：单例类。底层组件，负责服务端配置相关管理？

- 这个底层组件的内部，涉及 ET 标签事件系统的扫描【Config】标签，并 Invoke 相关（服务端的配置与启动?）这里花点儿时间，再进去把 ET 事件系统中各小服服务端根据 (excel? 等) 配置文件来加载和启动服务端（或是服务端的必要配置）的原理弄懂

- 框架事件系统里，有对各种不同标签的处理逻辑。Invoke 同理。程序域加载时，它扫描和管理框架里的所有必要相关标签，同 Invoke 标签同样有字典（套字典）纪录管理不同参数类型（args）的字典，字典里不同类型（type) 的激活处理器。对于特定的参数类型，type 类型，如果能够找到激活处理器，就会触发调用此激活回调，来作相应的处理。

```
public T Invoke<A, T>(int type, A args) where A: struct {
    // 先试着去拿，框架里这个【特定 args 类型】的所有标签申明过的 invokeHandlers
    if (!this.allInvokes.TryGetValue(typeof(A), out var invokeHandlers)) {
        throw new Exception($"Invoke error: {typeof(A).Name}");
    }
    // 再试着去拿，【特定类型 type】的 invokeHandler 处理器
    if (!invokeHandlers.TryGetValue(type, out var invokeHandler)) {
        throw new Exception($"Invoke error: {typeof(A).Name} {type}");
    }
    var aInvokeHandler = invokeHandler as AInvokeHandler<A, T>;
    if (aInvokeHandler == null) {
        throw new Exception($"Invoke error, not AInvokeHandler: {typeof(T).Name} {type}");
    }
    return aInvokeHandler.Handle(args); // 调用【Invoke】标签的相应处理回调逻辑
}
public void Invoke<A>(A args) where A: struct {
    Invoke(0, args);
}
public T Invoke<A, T>(A args) where A: struct {
    return Invoke<A, T>(0, args);
}
```

- 框架最底层的封装原理如此。这里，更多的是需要去找当前配置系，激活处理器的具体实现逻辑（在 ConfigLoader.cs 文件里，两个回调类类型），来理解这个初始化加载模块。

- 感觉今天上午把目前看到的这些，读得还算比较透彻。【亲爱的表哥，活宝妹一定要嫁的亲爱的表哥!!! 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!!!】

```
// Config 组件会扫描所有的有【Config】标签的配置，加载进来：它借助了两套加载系统，加载一个配置，与加载所有配置。而配置仍是通过【Conf
public class ConfigComponent: Singleton<ConfigComponent> {
    public struct GetAllConfigBytes {  }
    public struct GetOneConfigBytes {
        public string ConfigName;// 只是用一个字符串来区分不同配置
    }
    private readonly Dictionary<Type, ISingleton> allConfig = new Dictionary<Type, ISingleton>();
    public override void Dispose() {
        foreach (var kv in this.allConfig) {
            kv.Value.Destroy();
        }
    }
    public object LoadOneConfig(Type configType) {
        this.allConfig.TryGetValue(configType, out ISingleton oneConfig);// oneConfig：这里算是自定义变量的【申明与赋值】?
        if (oneConfig != null) {
            oneConfig.Destroy();
        }
        // 跟进 Invoke：去看一下框架里事件系统，找到具体的激活回调逻辑定义类：ConfigLoader.cs，去查看里面对 GetOneConfigBytes 类型
        byte[] oneConfigBytes = EventSystem.Instance.Invoke<GetOneConfigBytes, byte[]>(new GetOneConfigBytes() {ConfigName
        object category = SerializeHelper.Deserialize(configType, oneConfigBytes, 0, oneConfigBytes.Length);
        ISingleton singleton = category as ISingleton;
        singleton.Register(); //  【单例类初始化】：如果已经初始化过，会抛异常；单例类只初始化一次
        this.allConfig[configType] = singleton; // 底层：管理类单例类，不同类型，各有一个。框架里就有上面看过的四大单例类
        return category;
    }
    public void Load() { //  【加载】：系统加载，程序域加载
        this.allConfig.Clear(); // 清空
        //  【原理】：借助框架强大事件系统，扫描域里【Invoke|()】标签 (2 种)；根据参数类型，调用触发激活逻辑，到服务端特定路径特定文件
        Dictionary<Type, byte[]> configBytes = EventSystem.Instance.Invoke<GetAllConfigBytes, Dictionary<Type, byte[]>>(new
        foreach (Type type in configBytes.Keys) {
            byte[] oneConfigBytes = configBytes[type];
```

```
            this.LoadOneInThread(type, oneConfigBytes);
        }
    }
    public async ETTask LoadAsync() { // 哪里会调用这个方法？Entry.cs 服务端起来的时候，会调用此底层组件，加载各单例管理类。细看一
        this.allConfig.Clear();
        Dictionary<Type, byte[]> configBytes = EventSystem.Instance.Invoke<GetAllConfigBytes, Dictionary<Type, byte[]>>(new
        using ListComponent<Task> listTasks = ListComponent<Task>.Create();
        foreach (Type type in configBytes.Keys) {
            byte[] oneConfigBytes = configBytes[type];
// 四大单例管理类（Machine,Process,Scene,Zone）：每个单例类，开一个任务线路去完成？好像是这样的。
// 不明白为什么必须管理那四个，多不同场景可以位于同一进程，一台机器可以多核多进程？区区区。。。不明白
            Task task = Task.Run(() => LoadOneInThread(type, oneConfigBytes));
            listTasks.Add(task);
        }
        await Task.WhenAll(listTasks.ToArray());
    }

    private void LoadOneInThread(Type configType, byte[] oneConfigBytes) {
        object category = SerializeHelper.Deserialize(configType, oneConfigBytes, 0, oneConfigBytes.Length);
        lock (this) {
            ISingleton singleton = category as ISingleton;
            singleton.Register(); // 注册单例类：就是启动初始化一个单例类吧，框架里 Invoke 配置相关，有四大单例类
            this.allConfig[configType] = singleton;
        }
    }
}
```

3.8 ConfigSingleton<T>: ProtoObject, ISingleton

- 【配置单例泛型类】：实现 ISingleton 接口，适用于各种不同类型的单例类管理（生成 Register, 销毁 Destroy, 以及加载完成后的回调管理）。

```
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() { } // <<<<<<<<<<<<<<<<<<<< 一个回调接口 API
        public virtual void Dispose() { }
    }
```

3.9 StartMachineConfig: 抓四大单例管理类中的一个来读一下

- 确定的各小服【自底向上】传各小服配置的方式：感觉狠像【动态路由系统】的每 10 分钟重新自动扫描邻居，小伙伴云游自动上报上锁般，自底向上去上报各小服（这里是物理机?）配置，实现了 IMerge 接口，可以进程间传递配置消息（?）
- **【IMerge 接口类所申明的 merge() 方法，真正调用的地方，ExcelExporter.cs!!】**：找出来。找不到，是说 Protobuf 跨进程消息传递库，什么地方，使用 IMerge ？
- 【现有理解】：ET7 重构后，使用 Json 各种.txt 配置文件来启动服务器。同一类型，如同 Machine|Process|Scene|Zone, 不同配置文件合并的时候，会调用这里几大类型所实现过的

IMerge 接口里的 Merge() 方法，把对，比如所有小区的管理，纳入单例区 Zone 管理类 StartZoneConfigCategory.cs. 以后再有更深入理解再添加。

- 同样的命名空间，同一个文件，完全相同的类型，没弄明白的是，它为什么会在框架里出现两遍？是叫 partial-class, 可是这样的原理、两个文件的区别，以及用途，是在哪里是什么？
- 这个单例类型只存在于【服务端】。但是 ET 框架里，双端框架有多种不同运行模式。客户端可以作为独立客户端来运行，也可以作为双端模式运行（就是内自带一个服务端）。这里的服务端就同理，可是作为独立服务端，只作服务端，也可以作为客户端在双端运行模式中，客户端自身所携带的服务端来运行。所以，框架里它出现了两次。
- 另一个问题是：这个类是 Generated（/Users/hhj/pubFrameWorks/ET/Unity/Assets/Scripts/Codes/M，是框架自动生成的类，没有看懂。为什么框架会生成这个类？
- 独立的服务端，框架生成的文件？作为客户端双端运行模式下的服务端：框架生成的文件？
- Proto 相关的标签，各种各样的标签，看得懂的标签还好，不懂的 Proto 标签看得。。。

```
[ProtoContract]
[Config]
public partial class StartMachineConfigCategory : ConfigSingleton<StartMachineConfigCategory>, IMerge { // 实现了这个合并接口
    [ProtoIgnore]
    [BsonIgnore]
    private Dictionary<int, StartMachineConfig> dict = new Dictionary<int, StartMachineConfig>();
    [BsonElement]
    [ProtoMember(1)]
    private List<StartMachineConfig> list = new List<StartMachineConfig>();
    public void Merge(object o) { // 实现接口里申明的方法
        StartMachineConfigCategory s = o as StartMachineConfigCategory;
        this.list.AddRange(s.list); // 这里就可以是，进程间可传递的消息，的自动合并
    }
    [ProtoAfterDeserialization]
    public void ProtoEndInit() {
        foreach (StartMachineConfig config in list) {
            config.AfterEndInit();
            this.dict.Add(config.Id, config);
        }
        this.list.Clear();
        this.AfterEndInit();
    }
    public StartMachineConfig Get(int id) {
        this.dict.TryGetValue(id, out StartMachineConfig item);
        if (item == null)
            throw new Exception($" 配置找不到，配置表名: {nameof (StartMachineConfig)}, 配置 id: {id}");
        return item;
    }
    public bool Contain(int id) {
        return this.dict.ContainsKey(id);
    }
    public Dictionary<int, StartMachineConfig> GetAll() {
        return this.dict;
    }
    public StartMachineConfig GetOne() {
        if (this.dict == null || this.dict.Count <= 0)
            return null;
        return this.dict.Values.GetEnumerator().Current;
    }
}
[ProtoContract]
public partial class StartMachineConfig: ProtoObject, IConfig {
    [ProtoMember(1)]
    public int Id { get; set; }
    [ProtoMember(2)]
    public string InnerIP { get; set; }
    [ProtoMember(3)]
    public string OuterIP { get; set; }
    [ProtoMember(4)]
    public string WatcherPort { get; set; }
}
```

- 也没有看出这两个文件有任何的区别，只是任何一个具备服务端功能的项目（.csproj）都还是需要这个文件而已。
- 下面的文件就不放了，因为四大单例类（Machine, Process, Scene, Zone）还各不同，只抓一个只代表四分之一。。。得一个一个去分析。

3.10 StartProcessConfig: 【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!!!】

- 按现有的理解，Machine 是一个相对大的单位；一个 Machine 可以多核多进程多 Process; 一个核一个进程一个 Process 可以多线程多任务管理，一个 Process 里可以并存多个不同的 SceneType 【并存多个相同或不同功能的小服：登录服，网关服，房间服。。】；Zone 区，还不懂算是什么意思
- 与上面的 Machine 不同的是，Process 真正涉及了 Partial 的概念。同上一样，存在于【服务端】。可是因为 config 部分类的存在，框架里有四个文件。这里要把 partial 的原因弄明白.
- 就是两个文件，分别存在于 Config 文件夹，与 ConfigPartial 文件夹，不明白是为什么
- 这里，把一个版本的源码先贴这里，改天再看

```
[ProtoContract]
[Config]
public partial class StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge {
    [ProtoIgnore]
    [BsonIgnore]
    private Dictionary<int, StartProcessConfig> dict = new Dictionary<int, StartProcessConfig>();
    [BsonElement]
    [ProtoMember(1)]
    private List<StartProcessConfig> list = new List<StartProcessConfig>();
    public void Merge(object o) {
        StartProcessConfigCategory s = o as StartProcessConfigCategory;
        this.list.AddRange(s.list);
    }
    [ProtoAfterDeserialization]
    public void ProtoEndInit() {
        foreach (StartProcessConfig config in list) {
            config.AfterEndInit();
            this.dict.Add(config.Id, config);
        }
        this.list.Clear();
        this.AfterEndInit();
    }
    public StartProcessConfig Get(int id) {
        this.dict.TryGetValue(id, out StartProcessConfig item);
        if (item == null) {
            throw new Exception($" 配置找不到，配置表名: {nameof (StartProcessConfig)}, 配置 id: {id}");
        }
        return item;
    }
    public bool Contain(int id) {
        return this.dict.ContainsKey(id);
    }
    public Dictionary<int, StartProcessConfig> GetAll() {
        return this.dict;
    }
    public StartProcessConfig GetOne() {
        if (this.dict == null || this.dict.Count <= 0) {
            return null;
        }
        return this.dict.Values.GetEnumerator().Current;
    }
}
[ProtoContract]
public partial class StartProcessConfig: ProtoObject, IConfig {
    [ProtoMember(1)]
    public int Id { get; set; }
    [ProtoMember(2)]
```

```
    public int MachineId { get; set; }
    [ProtoMember(3)]
    public int InnerPort { get; set; }
}
```

3.11 StartSceneConfig: ISupportInitialize 【各种服 - 配置，场景配置】

```
public partial class StartSceneConfig: ISupportInitialize {
    public long InstanceId;
    public SceneType Type; // 场景类型

    public StartProcessConfig StartProcessConfig {
        get {
            return StartProcessConfigCategory.Instance.Get(this.Process);
        }
    }
    public StartZoneConfig StartZoneConfig {
        get {
            return StartZoneConfigCategory.Instance.Get(this.Zone);
        }
    }
    // 内网地址外网端口，通过防火墙映射端口过来
    private IPEndPoint innerIPOutPort;
    public IPEndPoint InnerIPOutPort {
        get {
            if (innerIPOutPort == null) {
                this.innerIPOutPort = NetworkHelper.ToIPEndPoint($"{this.StartProcessConfig.InnerIP}:{this.OuterPort}");
            }
            return this.innerIPOutPort;
        }
    }
    // 外网地址外网端口
    private IPEndPoint outerIPPort;
    public IPEndPoint OuterIPPort {
        get {
            if (this.outerIPPort == null) {
                this.outerIPPort = NetworkHelper.ToIPEndPoint($"{this.StartProcessConfig.OuterIP}:{this.OuterPort}");
            }
            return this.outerIPPort;
        }
    }
    public override void AfterEndInit() {
        this.Type = EnumHelper.FromString<SceneType>(this.SceneType);
        InstanceIdStruct instanceIdStruct = new InstanceIdStruct(this.Process, (uint) this.Id);
        this.InstanceId = instanceIdStruct.ToLong();
    }
}
```

3.12 StartSceneConfigCategory : 【Matchs!】ConfigSingleton<StartSceneConf IMerge

- 为什么这个类，会是写了两遍呢？有什么不同？跟前面类似，存在于任何具备服务端功能的模块。【服务端】【双端】
- 读里面的登录服，会知道它是如何管理登录服的（就是后面的例子，当它要拿登录服的地址的时候），它们是区服，就是分各个小区管理。如果集群是这个样子，大概匹配服也就是一样分小区管理了。
- 那么这个配置管理里，因为我要用匹配服与地图服，也要对至少是匹配服进行管理。那么，我在申请匹配的时候，网关服才能拿到匹配服的地址。
- 只在【服务端】存在。但是在双端模式、与服务端模式下，每种端有两个文件来定义这个类。。一个在【ProtoContract】里，可能可以进程间消息传递？一个在 ConfigPartial 文件夹里
- 这里的部分类 partial-class 仍然是没弄明白。什么情况下使用哪个类，不同部分类的实现原理。

- 【重构】：因为我现在还比较喜欢使用 Unity 下自带的双端模式，可是暂时只改【双端模式 ClientServer】下的文件，另一个专职服务端可能晚点儿再补上去。不用昨天晚上一样每个文件都改。
- 不知道下面的源码，属于端的两种模式、部分类的两个文件，四个中的哪一个？

```
// 配置文件处理，或是服务器启动相关类，以前都没仔细读过
public partial class StartSceneConfigCategory {
    public MultiMap<int, StartSceneConfig> Gates = new MultiMap<int, StartSceneConfig>();
    public MultiMap<int, StartSceneConfig> ProcessScenes = new MultiMap<int, StartSceneConfig>();
    public Dictionary<long, Dictionary<string, StartSceneConfig>> ClientScenesByName = new Dictionary<long, Dictionary<stri
    public StartSceneConfig LocationConfig;
    public List<StartSceneConfig> Realms = new List<StartSceneConfig>();
    public List<StartSceneConfig> Matchs = new List<StartSceneConfig>(); // <<<<<<<<<<<<<<<<<<< 添加管理
    public List<StartSceneConfig> Routers = new List<StartSceneConfig>();
    public List<StartSceneConfig> Robots = new List<StartSceneConfig>();
    public StartSceneConfig BenchmarkServer;

    public List<StartSceneConfig> GetByProcess(int process) {
        return this.ProcessScenes[process];
    }
    public StartSceneConfig GetBySceneName(int zone, string name) {
        return this.ClientScenesByName[zone][name];
    }
    public override void AfterEndInit() {
        foreach (StartSceneConfig startSceneConfig in this.GetAll().Values) {
            this.ProcessScenes.Add(startSceneConfig.Process, startSceneConfig);

            if (!this.ClientScenesByName.ContainsKey(startSceneConfig.Zone)) {
                this.ClientScenesByName.Add(startSceneConfig.Zone, new Dictionary<string, StartSceneConfig>());
            }
            this.ClientScenesByName[startSceneConfig.Zone].Add(startSceneConfig.Name, startSceneConfig);

            switch (startSceneConfig.Type) {
            case SceneType.Realm:
                this.Realms.Add(startSceneConfig);
                break;
            case SceneType.Gate:
                this.Gates.Add(startSceneConfig.Zone, startSceneConfig);
                break;
            case SceneType.Match:                   // <<<<<<<<<<<<<<<<<<< 自己加的
                this.Matchs.Add(startSceneConfig); // <<<<<<<<<<<<<<<<<<<
                break;
            case SceneType.Location:
                this.LocationConfig = startSceneConfig;
                break;
            case SceneType.Robot:
                this.Robots.Add(startSceneConfig);
                break;
            case SceneType.Router:
                this.Routers.Add(startSceneConfig);
                break;
            case SceneType.BenchmarkServer:
                this.BenchmarkServer = startSceneConfig;
                break;
            }
        }
    }
}
```

- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】