# ET 框架学习笔记（三） - - 网络交互相关

deepwaterooo

July 18, 2023

## Contents

# 1 网关服：客户端信息发送的直接代理，中转站，组件分析

- SceneFactory: 【初始化】时，带如下几个组件

```
public static class SceneFactory {
    public static async ETTask<Scene> CreateServerScene(Entity parent, long id, long instanceId, int zone, string name, Sce
        await ETTask.CompletedTask;
        Scene scene = EntitySceneFactory.CreateScene(id, instanceId, zone, sceneType, name, parent);
        // 任何场景: 无序消息分发器，可接收消息，队列处理; 发呢？
        scene.AddComponent<MailBoxComponent, MailboxType>(MailboxType.UnOrderMessageDispatcher); // 重构？应该是对进程间消息发

        switch (scene.SceneType) {
            case SceneType.Router:
                scene.AddComponent<RouterComponent, IPEndPoint, string>(startSceneConfig.OuterIPPort, startSceneConfig.Star
                break;
            case SceneType.RouterManager: // 正式发布请用 CDN 代替 RouterManager
```

```
            // 云服务器在防火墙那里做端口映射
            scene.AddComponent<HttpComponent, string>($"http:// *:{startSceneConfig.OuterPort}/");
            break;
        // // 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; // ...
```

### 1.1 NetServerComponent:

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

## 2 服务器的功能概述：各服务器的作用（这个不是 ET7 版本的，以前的）

- Manager：连接客户端的外网和连接内部服务器的内网，对服务器进程进行管理，自动检测和启动服务器进程。加载有内网组件 NetInnerComponent，外网组件 NetOuterComponent，服务器进程管理组件。自动启动突然停止运行的服务器，保证此服务器管理的其它服务器崩溃后能及时自动启动运行。
- Realm：对 Actor 消息进行管理（添加、移除、分发等），连接内网和外网，对内网服务器进程进行操作，随机分配 Gate 服务器地址。内网组件 NetInnerComponent，外网组件 NetOuterComponent，Gate 服务器随机分发组件。客户端登录时连接的第一个服务器，也可称为登录服务器。
- Gate：对玩家进行管理，对 Actor 消息进行管理（添加、移除、分发等），连接内网和外网，对内网服务器进程进行操作，随机分配 Gate 服务器地址，对 Actor 消息进程进行管理，对玩家 ID 登录后的 Key 进行管理。加载有玩家管理组件 PlayerComponent，管理登陆时联网的 Key 组件 GateSessionKeyComponent。
- Location：连接内网，服务器进程状态集中管理（Actor 消息 IP 管理服务器）。加载有内网组件 NetInnerComponent，服务器消息处理状态存储组件 LocationComponent。对客户端的登录信息进行验证和客户端登录后连接的服务器，登录后通过此服务器进行消息互动，也可称为验证服务器。
- Map：连接内网，对 ActorMessage 消息进行管理（添加、移除、分发等），对场景内现在活动物体存储管理，对内网服务器进程进行操作，对 Actor 消息进程进行管理，对 Actor 消息进行管理（添加、移除、分发等），服务器帧率管理。服务器帧率管理组件 ServerFrameComponent。
- AllServer：将以上服务器功能集中合并成一个服务器。另外增加 DB 连接组件 DBComponent
- Benchmark：连接内网和测试服务器承受力。加载有内网组件 NetInnerComponent，服务器承受力测试组件 BenchmarkComponent。

# 3 Session 会话框相关

- 当需要连的时候，比如网关服与匹配服，新的框架里连接时容易出现困难，找不到组件，或是用不对组件，或是组件用得不对，端没能分清楚。理解不够。
- 就是说，这个新的 ET7 框架下，服务端的这些，事件机制的，没弄明白没弄透彻。

# 4 Root 客户端根场景管理以及必要的组件：【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!!!】

- 昨天晚上，今天上午把这个根场景下的必要的组件，疑难点大致又看了一遍过了一遍。以后再有什么不懂，或是理解一点儿新的，再添加。【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!!!】
- 把这个客户端的根场景相关的管理组件整理一下。在系统启动起来的时候，公用组件以及客户端组件的时候，分别有添加一些必要的组件。
- 这里，当把根场景 Root.Instance.Scene 下添加的组件整理好，就看见，几乎所有客户端必要需要的组件这里都添加了，那么刚才几分钟前，我想要自己添加一个的 SceneType.AllServer 里模仿参考项目想要添加的。这里需要想一下：两个都需要吗，还是 SceneType.AllServer 因为这个根场景下都加了，我自己画蛇添足的全服就可以不要了？

## 4.1 Root.cs

```
// 管理根部的 Scene: 这个根部，是全局视图的根节点
public class Root: Singleton<Root>, ISingletonAwake { // 单例类，自觉醒
    // 管理所有的 Entity:
    private readonly Dictionary<long, Entity> allEntities = new();
    public Scene Scene { get; private set; }
    public void Awake() {
        this.Scene = EntitySceneFactory.CreateScene(0, SceneType.Process, "Process");
    }
    public override void Dispose() {
        this.Scene.Dispose();
    }
    public void Add(Entity entity) {
        this.allEntities.Add(entity.InstanceId, entity);
    }

    public void Remove(long instanceId) {
        this.allEntities.Remove(instanceId);
    }
    public Entity Get(long instanceId) {
        Entity component = null;
        this.allEntities.TryGetValue(instanceId, out component);
        return component;
    }

    public override string ToString() {
        StringBuilder sb = new();
        HashSet<Type> noParent = new HashSet<Type>();
        Dictionary<Type, int> typeCount = new Dictionary<Type, int>();
        HashSet<Type> noDomain = new HashSet<Type>();
        foreach (var kv in this.allEntities) {
            Type type = kv.Value.GetType();
            if (kv.Value.Parent == null) {
                noParent.Add(type);
            }
            if (kv.Value.Domain == null) {
                noDomain.Add(type);
            }
```

```
            if (typeCount.ContainsKey(type)) {
                typeCount[type]++;
            }
            else {
                typeCount[type] = 1;
            }
        }
        sb.AppendLine("not set parent type: ");
        foreach (Type type in noParent) {
            sb.AppendLine($"\t{type.Name}");
        }
        sb.AppendLine("not set domain type: ");
        foreach (Type type in noDomain) {
            sb.AppendLine($"\t{type.Name}");
        }
        IOrderedEnumerable<KeyValuePair<Type, int>> orderByDescending = typeCount.OrderByDescending(s => s.Value);
        sb.AppendLine("Entity Count: ");
        foreach (var kv in orderByDescending) {
            if (kv.Value == 1) {
                continue;
            }
            sb.AppendLine($"\t{kv.Key.Name}: {kv.Value}");
        }
        return sb.ToString();
    }
}
```

## 4.2 EntryEvent1_InitShare.cs: Root 根场景添加组件

- 这里是双端共享组件启动的时候，也就是说，Root.Instance.Scene 并不仅仅只是客户端场景，也是服务端场景。

```
// 公用的相关组件的初始化:
[Event(SceneType.Process)]
public class EntryEvent1_InitShare: AEvent<EventType.EntryEvent1> {

    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;
    }
}
```

## 4.3 EntryEvent2_InitServer: 服务端启动的时候添加的组件

```
[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>();
        // 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

        StartProcessConfig processConfig = StartProcessConfigCategory.Instance.Get(Options.Instance.Process);
        switch (Options.Instance.AppType) {
        case AppType.Server: {
            Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(processConfig.InnerIPPort);
            var processScenes = StartSceneConfigCategory.Instance.GetByProcess(Options.Instance.Process);
            foreach (StartSceneConfig startConfig in processScenes) {
                await SceneFactory.CreateServerScene(ServerSceneManagerComponent.Instance, startConfig.Id, startConfig.Inst
```

```
            }
            break;
        }
        case AppType.Watcher: {
            StartMachineConfig startMachineConfig = WatcherHelper.GetThisMachineConfig();
            WatcherComponent watcherComponent = Root.Instance.Scene.AddComponent<WatcherComponent>();
            watcherComponent.Start(Options.Instance.CreateScenes);
            Root.Instance.Scene.AddComponent<NetInnerComponent, IPEndPoint>(NetworkHelper.ToIPEndPoint($"{startMachineConfi
            break;
        }
        case AppType.GameTool:
            break;
        }
        if (Options.Instance.Console == 1) {
            Root.Instance.Scene.AddComponent<ConsoleComponent>();
        }
    }
}
```

# 5 ETTask 和 ETVoid: 第三方库的 ETTask, 参考 ET-EUI 框架

- 特异包装：主要是实际了异步调用的流式写法。它方法定义的内部，是封装有协程异步状态机的？IAsyncStateMachine. 当要运行协程的下一步，也是调用和运行。NET 库里的 IAsyncStateMachine.moveNext()
- .NET 还提供了 AsyncMethodBuilder 的 type trait 来让你自己实现这个状态机和你自己的 Task 类型，因此你可以最大程度发挥想象来编写你想控制的一切。ETTask|ETVoid 就是使用底层的这些方法来封装的结果。async/await 是一个纯编译器特性。
- 这个框架里 ET7 里，就有相关模块 **【具体说是，两个实体类，实际定义了两种不同返回值 ETTask-ETVoid 的协程编译生成方法】**，能够实现对这个包装的自动编译成协程的编译逻辑方法定义。理解上，感觉像是 ET7 框架里，为了这个流式写法，定义了必要的标签系，和相关的协程生成方法，来帮助这个第三方库实现异步调用的流式写法。
- 上面的，写得把自己都写昏了。就是 ET7 框架是如何实现异步调用的流式写法的呢？它把异步调用封装成协程。面对 ET7 框架里广泛用到的 ETTask|ETVoid 两类稍带个性化异步任务，如同 ETTask 和 ETVoid 是框架自己的封装一样，这个框架，也使用.NET 里的 IAsyncStateMachine 等底层接口 API 等，自定义了异步协程任务的生成方法。
- 这类方法里，都封装有一个 ETTask, 因为自定义封装在这些自定义类里，就对可能会用到的操作提供了必要的 API, 比如设置异常，拿取任务等等。
- 上面的自定义方法生成器:有三类,分别是 AsyncETVoidMethodBuilder, AsyncETTaskMethodBuilder 和 AsyncETTaskCompletedMethodBuilder
- 感觉因为这两大返回类型，我没有能看懂看透，所以上面一个部分的消息处理，两个函数 Handle() 和 Run() 的返回类型，以及参数被我改得乱七八糟，是不应该的。
- 磨刀不误砍柴工，我应该投入时间把这个第三方库一样的包装理解透彻，然后再去弄懂上面一个部分，再去改那些编译错误。
- **【ET-EUI】里：**原本类的定义什么的，也是一样的，那就是主要去看，他是怎么使用 ETVoid, 为什么它使用 ETVoid 不会报错，而我在 ET7 里用就会。
- **【多线程同步】**关于多线程同步的理解：来自于网络：
  - ETTASK 的由于没有开新线程，也没有使用线程池 Task，所以肯定是在主线程运行的，那么游戏开始的 SynchronizationContext.SetSynchronizationContext(OneThreadSynchronizatio 这句代码有啥用呢?

  - 个人理解为，**在 ET 中虽然主逻辑是单线程的，但是与 IO 设备，比如从 socket 读取数据，或者从 TCP,KCP 获取网络数据得时候，是多线程的获取数据的，所以当数据到达时，为了保证是单线程，所以在获取数据的地方，以回调得方式，将回调方法扔到 OneThreadSynchronizationContext 中执行（async 设置了同步上下文是线程安全的**，说的应该也是这个 OneThreadSynchronizationContext() 什么的相关的）
  - 白话多线程同步原理如下：下面的也是 ET 框架中网络异步线程同步中干过的同步执行逻辑。那个类大概是 NetService.cs. 就是分主线程，异步线程，有队列，Update() 里同步的。
    * ET 是单线程的，所以不会管理线程
    * 跨线程都是把委托投递到一个队列，主线程不停从队列中取出委托执行
    * 你看看 asynctool 的代码，本质上就是把委托投递到主线程
    * 每帧取完队列中的所有委托，执行完
  - 这个细节，是自己第一个游戏里使用 ET-EUI 作为服务端，非 ET 框架的客户端与服务端连接时，自己曾经遇到过的。非 ET 框架的客户端，是使用了一个其它的 UnityPlayer 里一个 API 相关的第三方来同步异步线程的结果到主线程。所以这个细节还是印象深刻。

- 首先要能把控得住多线程，才能谈性能。其次，et 是服务端多进程，同样能利用多核。et 是逻辑单线程，并不意味着只能单线程，你能把控得住，照样可以多线程，一般人是不行的。（这些，看不懂，感觉更像是避重就轻吹牛皮一样。。。）

- **【ETTask-await 后面的执行线程:】**

  - async await 如果用的 Task, await 后面的部分是不确定在哪个线程执行的, 猫大以前 4.0 的做法就是把上下文抛到主线程, 让主线程执行.
  - 如果用的是 ETTask, await 后面的部分是一定在主线程执行的. 就完全相当于写了个回调方法了
  - Task 实际上也是回调, 不过这个回调方法的执行原本可能不在主线程罢了

- ETVoid 是代替 async void，意思是新开一个协程。ettask 跟 task 一样。当然 task 不去 await 也相当于新开协程，但是编辑器会冒出提示，提示你 await。所以新开协程最好用 ETVoid。4.0 用 async void。使用场景，自己写写就明白啦. 协程就是回调.

- **无 GC ETTask**, 其实是利用对象池，注意，必须小心使用，不懂不要乱用。无 GC 的原理同自己写第一个游戏，使用资源池是一样的，就是说，当一个 ETTask 使用完毕，不再使用的时候，不是要 GC 来回收，而是程序的逻辑自己管理，回收到对象池管理器，对于应用程序来说，就是不释放，自己管理它的再使用。不释放就不会引起 GC 回收，所以叫无 GC.

  - 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!
  - 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配出来的 ETTask，产生灾难性的后果。（自己的理解，await 之后，再操作 ETTask, 那么操作的极有可能是【当 boolean fromPool = true】从对象池新取出的一个异步任务，不是预期行为，当然就会引起一片混乱。。。可是，框架里仍然有狠多对异步任务 SetResult() 的地方，尤其是各种服的消息处理器处理逻辑里。什么情境下可以安全地使用 SetResult(), 需要自己去搞明白）
  - SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult. （这里，对一个异步任务，设置结果 SetResult(), 可能会设置多次吗?）
  - 大部分情况可以用 objectwait 代替 ettask，推荐使用，绝对不会出问题

- 这里因为弄不明白，他们建议的学习方法是:

  - ettask 还要啥教程?

  - 要搞懂 ettask 下载一个 jetbrain peek 工具，反编译下看下生成的代码就行了。
  - 参考 Timercomponent，看懂就全明白了
  - 看网上的文章看十年也不会明白，自己写一下 timercomponet 啥都懂了
  - 接下来，自己尝试理解这部分的方法应该是：给 VS 2022 安装第三方插件库 ILSpy, 然后借用插件把编译码自己弄出来，插日志，作任何可以帮助自己理解的东西来理解这部分。**【先给 VS 安装一个插件 ILSpy, 这样更容易反编译代码进行查看，另外要注意反编译 async 和 await 的时候，要把 C# 代码版本改为 4.0 哦。】**前在，这是网上提示的反编译方法。这个，改天再接着看，先再事理理解一点儿别的。今天一定更新一下。明天出行，没时间看和更新。

- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】

## 5.1 IAsyncStateMachine

- 异步方法中，每遇见一个 await 调用，都会生成一个异步状态类，这个异步状态类会实现这个接口

```
namespace System.Runtime.CompilerServices {
    public interface IAsyncStateMachine {
        void MoveNext();
        void SetStateMachine(IAsyncStateMachine stateMachine);
    }
}
```

## 5.2 enum AwaiterStatus: IAwaiter.cs 文件里. 理解为异步任务的现执行进展状态

- 现框架里，扩展 IAwaiter, 自定义的现框架 ETTask 所可能有的三种不同状态。

```
public enum AwaiterStatus: byte {
    // The operation has not yet completed.
    Pending = 0,// 这个用在判断语句里比较多，主要用它来判断：异步任务是否已经完成
    // The operation completed successfully.
    Succeeded = 1,
    // The operation completed with an error.
    Faulted = 2,
}
```

## 5.3 ETTaskCompleted: 已经完成了的异步任务。比较特殊：可以简单进行写结果？等等的必要回收工作，就可以返回异步任务对象池回收再利用？

- 因为我把 AsyncMethodBuilder 理解成为：异步任务的协程编译器编译逻辑。
- 所以这个类就是定义，异步任务协程中的一个特殊状态：异步任务结束了，但是还没能写结果时的 IAsyncStateMachine|IAwater 的一个最为特殊的状态。它可以用作可能需要写结果时的一个准备，但也可能不需要再写结果？在框架里用得狠多。所以它狠轻量，可以快速写结果或是快速回收到对象池复用。
- 因为它是协程异步状态机中的一个相对特殊状态，本质上是异步状态机中的一个极特殊的状态，也提供了必要的 API, 比如写结果。

```
[AsyncMethodBuilder(typeof (AsyncETTaskCompletedMethodBuilder))]
public struct ETTaskCompleted: ICriticalNotifyCompletion {
    [DebuggerHidden]
// 能不能理解为，已经结束了的异步任务 ETTaskCompleted, 也是协程异步状态机中的一个状态，是 IAwaker 的实体类实现状态，返回这个
    public ETTaskCompleted GetAwaiter() {
        return this;
    }
    [DebuggerHidden]
```

```
        public bool IsCompleted => true;
        [DebuggerHidden]
        public void GetResult() {
        }
    // 就是说：下面的两个回调函数，可以帮  助把异步任务的执行结果给返回回去
        [DebuggerHidden]
        public void OnCompleted(Action continuation) {
        }
        [DebuggerHidden]
        public void UnsafeOnCompleted(Action continuation) {
        }
    }
```

## 5.4 struct ETVoid: ICriticalNotifyCompletion. 这里涉及协程的分阶段的执行相关逻辑的生成方法自动化相关的标签

```
[AsyncMethodBuilder(typeof (AsyncETVoidMethodBuilder))]// 【异步方法生成标签】：是。NET CompilerService 里的属性标签。自动生成协
internal struct ETVoid: ICriticalNotifyCompletion {
    [DebuggerHidden]
        public void Coroutine() { }
    [DebuggerHidden]
        public bool IsCompleted => true;
    [DebuggerHidden]
        public void OnCompleted(Action continuation) { }
    [DebuggerHidden]
        public void UnsafeOnCompleted(Action continuation) { }
}
```

## 5.5 ETTask: ICriticalNotifyCompletion:

- 这个类的定义比较大，分普通类，和泛型类。我的笔记需要记在同一个地方。今天早上这个类又记错地方记到 ET-EUI 上去了

```
    [AsyncMethodBuilder(typeof (ETAsyncTaskMethodBuilder))]
    public class ETTask: ICriticalNotifyCompletion {
        public static Action<Exception> ExceptionHandler;// 异常回调
        public static ETTaskCompleted CompletedTask {    // 异步任务结束后的封装
            get {
                return new ETTaskCompleted();
            }
        }
        private static readonly Queue<ETTask> queue = new Queue<ETTask>();// 异步任务对象池
        // 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!
        // 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配出来的 ETTask，产生灾难性的后果
        // SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult
        public static ETTask Create(bool fromPool = false) {
            if (!fromPool)
                return new ETTask();
            if (queue.Count == 0)
                return new ETTask() {fromPool = true};
            return queue.Dequeue();
        }
        private void Recycle() { // 涉及 ETTask 无 GC 的逻辑实现：
            if (!this.fromPool)  // 因为不返回对象池，所以会 GC
                return; // 原则：只有从池里取出来的，才返回池
            this.state = AwaiterStatus.Pending;// 【没明白:】回收时还设置为 Pending，什么时候写的当前结果？应该是在回收前
            this.callback = null;
            if (queue.Count > 1000)  // 因为对象池中，异步任务数目已达 1000，不再回收，也会产生 GC
                return;
            queue.Enqueue(this); // 真正无 GC，因为回收到对象池，队列里去了
        }
        private bool fromPool;
        private AwaiterStatus state;
        private object callback; // Action or ExceptionDispatchInfo
        private ETTask() {  }
        //  【不明白下面两个方法】：不知道这两个方法，绕来绕去，在干什么？
        [DebuggerHidden] // 下面，旦凡带 async 关键字的方法，都是异步方法，编译器编译 async 方法时，会自动生成方法所对应的 Coro
        private async ETVoid InnerCoroutine() { // 这里，怎么就可以用 ETVoid 了呢？ private 内部异步方法
            await this; //  【不明白】：每次看见 await 后面接一个单词，就不知道是在等什么？等待这个 ETTask 异步任务类初始化完成？
        }
        [DebuggerHidden]
```

```
    public void Coroutine() { // 公用无返回，非异步方法。它调用了类内部私有的异步方法 InnerCoroutine()
        InnerCoroutine().Coroutine();// 这里因为理解不透，总感觉同上面的方法，返回 this，又调用了自己本方法 Coroutine() -
    }
    [DebuggerHidden]
    public ETTask GetAwaiter() {
        return this;
    }
    public bool IsCompleted {
        [DebuggerHidden]
        get {
            return this.state != AwaiterStatus.Pending; // 只要不是 Pending 状态，就是异步任务执行结束
        }
    }
    [DebuggerHidden]
    public void UnsafeOnCompleted(Action action) {
        if (this.state != AwaiterStatus.Pending) { // 如果当前异步任务执行结束，就触发非空回调
            action?.Invoke();
            return;
        }
        this.callback = action; // 任务还没有结束，就纪录回调备用
    }
    [DebuggerHidden]
    public void OnCompleted(Action action) {
        this.UnsafeOnCompleted(action);
    }
    [DebuggerHidden]
    public void GetResult() {
        switch (this.state) {
            case AwaiterStatus.Succeeded:
                this.Recycle();
                break;
            case AwaiterStatus.Faulted:
                ExceptionDispatchInfo c = this.callback as ExceptionDispatchInfo;
                this.callback = null;
                this.Recycle();
                c?.Throw();
                break;
            default:
                throw new NotSupportedException("ETTask does not allow call GetResult directly when task not complete
        }
    }
    [DebuggerHidden]
    public void SetResult() {
        if (this.state != AwaiterStatus.Pending) {
            throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
        }
        this.state = AwaiterStatus.Succeeded;
        Action c = this.callback as Action;
        this.callback = null;
        c?.Invoke();
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    [DebuggerHidden]
    public void SetException(Exception e) {
        if (this.state != AwaiterStatus.Pending) {
            throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
        }
        this.state = AwaiterStatus.Faulted;
        Action c = this.callback as Action;
        this.callback = ExceptionDispatchInfo.Capture(e);
        c?.Invoke();
    }
}
[AsyncMethodBuilder(typeof (ETAsyncTaskMethodBuilder<>))]
public class ETTask<T>: ICriticalNotifyCompletion {
    private static readonly Queue<ETTask<T>> queue = new Queue<ETTask<T>>();
    // 请不要随便使用 ETTask 的对象池，除非你完全搞懂了 ETTask!!!
    // 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配出来的 ETTask，产生灾难性的后果
    // SetResult 的时候请现将 tcs 置空，避免多次对同一个 ETTask SetResult
    public static ETTask<T> Create(bool fromPool = false) {
        if (!fromPool)
            return new ETTask<T>();
        if (queue.Count == 0)
            return new ETTask<T>() { fromPool = true };
        return queue.Dequeue();
```

```
}
private void Recycle() {
    if (!this.fromPool)
        return;
    this.callback = null;
    this.value = default;
    this.state = AwaiterStatus.Pending;
    // 太多了
    if (queue.Count > 1000)
        return;
    queue.Enqueue(this);
}
private bool fromPool;
private AwaiterStatus state;
private T value;
private object callback; // Action or ExceptionDispatchInfo
private ETTask() {
}
[DebuggerHidden]
private async ETVoid InnerCoroutine() {
    await this;
}
[DebuggerHidden]
public void Coroutine() {
    InnerCoroutine().Coroutine();
}
[DebuggerHidden]
public ETTask<T> GetAwaiter() {
    return this;
}
[DebuggerHidden]
public T GetResult() {
    switch (this.state) {
    case AwaiterStatus.Succeeded:
        T v = this.value;
        this.Recycle();
        return v;
    case AwaiterStatus.Faulted:
        ExceptionDispatchInfo c = this.callback as ExceptionDispatchInfo;
        this.callback = null;
        this.Recycle();
        c?.Throw();
        return default;
    default:
        throw new NotSupportedException("ETask does not allow call GetResult directly when task not completed. Pl
    }
}
public bool IsCompleted {
    [DebuggerHidden]
    get {
        return state != AwaiterStatus.Pending;
    }
}
[DebuggerHidden]
public void UnsafeOnCompleted(Action action) {
    if (this.state != AwaiterStatus.Pending) {
        action?.Invoke();
        return;
    }
    this.callback = action;
}
[DebuggerHidden]
public void OnCompleted(Action action) {
    this.UnsafeOnCompleted(action);
}
[DebuggerHidden]
public void SetResult(T result) {
    if (this.state != AwaiterStatus.Pending) {
        throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
    }
    this.state = AwaiterStatus.Succeeded;
    this.value = result;
    Action c = this.callback as Action;
    this.callback = null;
    c?.Invoke();
```

```
        }
        [DebuggerHidden]
        public void SetException(Exception e) {
            if (this.state != AwaiterStatus.Pending) {
                throw new InvalidOperationException("TaskT_TransitionToFinal_AlreadyCompleted");
            }
            this.state = AwaiterStatus.Faulted;
            Action c = this.callback as Action;
            this.callback = ExceptionDispatchInfo.Capture(e);
            c?.Invoke();
        }
    }
```

## 5.6 ETCancellationToken: 管理所有的取消？回调：因为可能不止一个取消回调，所以 HashSet 管理

```
public class ETCancellationToken {// 管理所有的【取消】回调：因为可能不止一个取消回调，所以 HashSet 管理
    private HashSet<Action> actions = new HashSet<Action>();
    public void Add(Action callback) {
        // 如果 action 是 null，绝对不能添加，要抛异常，说明有协程泄漏
        // 【不喜欢这个注释，看不懂，感觉它吓唬人的。。】
        this.actions.Add(callback);
    }
    public void Remove(Action callback) {
        this.actions?.Remove(callback);
    }
    public bool IsDispose() {
        return this.actions == null;
    }
    public void Cancel() {
        if (this.actions == null) {
            return;
        }
        this.Invoke();
    }
    private void Invoke() {
        HashSet<Action> runActions = this.actions;
        this.actions = null;
        try {
            foreach (Action action in runActions) {
                action.Invoke();
            }
        }
        catch (Exception e) {
            ETTask.ExceptionHandler.Invoke(e);
        }
    }
}
```

## 5.7 ETTaskHelper: 有个类中类 CoroutineBlocker 看不懂

```
public static class ETTaskHelper {
    public static bool IsCancel(this ETCancellationToken self) {
        if (self == null)
            return false;
        return self.IsDispose();
    }
    // 【看不懂】：感觉理解这个类有难度
    private class CoroutineBlocker {
        private int count; // 不知道，这个变量记的是什么？
        private ETTask tcs;
        public CoroutineBlocker(int count) {
            this.count = count;
        }
        public async ETTask RunSubCoroutineAsync(ETTask task) {
            try {
                await task;
            }
            finally {
                --this.count;
                if (this.count <= 0 && this.tcs != null) { // 写结果？
```

```
                ETTask t = this.tcs;
                this.tcs = null;
                t.SetResult();
            }
        }
    }
    public async ETTask WaitAsync() {
        if (this.count <= 0)
            return;
        this.tcs = ETTask.Create(true);
        await tcs;
    }
}
public static async ETTask WaitAny(List<ETTask> tasks) {
    if (tasks.Count == 0)
        return;
    CoroutineBlocker coroutineBlocker = new CoroutineBlocker(1);
    foreach (ETTask task in tasks) {
        coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
    }
    await coroutineBlocker.WaitAsync();
}
public static async ETTask WaitAny(ETTask[] tasks) {
    if (tasks.Length == 0)
        return;
    CoroutineBlocker coroutineBlocker = new CoroutineBlocker(1);
    foreach (ETTask task in tasks) {
        coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
    }
    await coroutineBlocker.WaitAsync();
}
public static async ETTask WaitAll(ETTask[] tasks) {
    if (tasks.Length == 0)
        return;
    CoroutineBlocker coroutineBlocker = new CoroutineBlocker(tasks.Length);
    foreach (ETTask task in tasks) {
        coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
    }
    await coroutineBlocker.WaitAsync();
}
public static async ETTask WaitAll(List<ETTask> tasks) {
    if (tasks.Count == 0)
        return;
    CoroutineBlocker coroutineBlocker = new CoroutineBlocker(tasks.Count);
    foreach (ETTask task in tasks) {
        coroutineBlocker.RunSubCoroutineAsync(task).Coroutine();
    }
    await coroutineBlocker.WaitAsync();
}
}
```

## 5.8 ETAsyncTaskMethodBuilder: 同样是换汤不换药的两个部分：普通类与泛型类

```
public struct ETAsyncTaskMethodBuilder {
    private ETTask tcs;
    // 1. Static Create method.
    [DebuggerHidden]
    public static ETAsyncTaskMethodBuilder Create() {
        ETAsyncTaskMethodBuilder builder = new ETAsyncTaskMethodBuilder() { tcs = ETTask.Create(true) };
        return builder;
    }
    // 2. TaskLike Task property.
    [DebuggerHidden]
    public ETTask Task => this.tcs;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception exception) {
        this.tcs.SetException(exception);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult() {
```

```
        this.tcs.SetResult();
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
public struct ETAsyncTaskMethodBuilder<T> {
    private ETTask<T> tcs;
    // 1. Static Create method.
    [DebuggerHidden]
    public static ETAsyncTaskMethodBuilder<T> Create() {
        ETAsyncTaskMethodBuilder<T> builder = new ETAsyncTaskMethodBuilder<T>() { tcs = ETTask<T>.Create(true) };
        return builder;
    }
    // 2. TaskLike Task property.
    [DebuggerHidden]
    public ETTask<T> Task => this.tcs;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception exception) {
        this.tcs.SetException(exception);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult(T ret) {
        this.tcs.SetResult(ret);
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
```

## 5.9 AsyncETTaskCompletedMethodBuilder:

```
public struct AsyncETTaskCompletedMethodBuilder {
    // 1. Static Create method.
    [DebuggerHidden]
    public static AsyncETTaskCompletedMethodBuilder Create() {
        AsyncETTaskCompletedMethodBuilder builder = new AsyncETTaskCompletedMethodBuilder();
        return builder;
```

```
    }
    // 2. TaskLike Task property(void)
    public ETTaskCompleted Task => default;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception e) {
        ETTask.ExceptionHandler.Invoke(e);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult() { // do nothing
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.UnsafeOnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
```

## 5.10 AsyncETVoidMethodBuilder: 定义的是 async ETVoid 的编译方法?

```
// 异步 ETVoid 内部生成方法:
internal struct AsyncETVoidMethodBuilder {
    // 1. Static Create method.
    [DebuggerHidden]
    public static AsyncETVoidMethodBuilder Create() {
        AsyncETVoidMethodBuilder builder = new AsyncETVoidMethodBuilder();
        return builder;
    }
    // 2. TaskLike Task property(void)
    [DebuggerHidden]
    public ETVoid Task => default;
    // 3. SetException
    [DebuggerHidden]
    public void SetException(Exception e) {
        ETTask.ExceptionHandler.Invoke(e);
    }
    // 4. SetResult
    [DebuggerHidden]
    public void SetResult() {
        // do nothing: 因为它实际的返回值是 void
    }
    // 5. AwaitOnCompleted
    [DebuggerHidden]
    public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwai
        awaiter.OnCompleted(stateMachine.MoveNext);
    }
    // 6. AwaitUnsafeOnCompleted
    [DebuggerHidden]
    [SecuritySafeCritical]
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where
        awaiter.UnsafeOnCompleted(stateMachine.MoveNext);
    }
    // 7. Start
    [DebuggerHidden]
    public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine {
        stateMachine.MoveNext();
    }
    // 8. SetStateMachine
```

```
    [DebuggerHidden]
    public void SetStateMachine(IAsyncStateMachine stateMachine) {
    }
}
```

### 5.11 ICriticalNotifyCompletion:

```
namespace System.Runtime.CompilerServices {
// 接口类：提供了一个，任务完成后的回调接口
    public interface ICriticalNotifyCompletion : INotifyCompletion {
        [SecurityCritical]
        void UnsafeOnCompleted(Action continuation);
    }
}
```

### 5.12 AsyncMethodBuilderAttribute:.NET 系统的标签

- 自己先前没能理解，为什么标记了【AsyncMethodBuilder(typeof(className))】就能标记某个类的协程生成方法
- 是因为这个系统标签，它申明了 AttributeUsage 属性，申明了适用类型，可以是 (AttributeTargets.Class | AttributeTargets.Struct) 等等
- 所以，当 ETTask 异步库自定义了 ETTask, ETVoid, 和 ETTaskCompleted 三个类，就可以使用上面的系统标签，来标注申明：这个类是以上三个中特定指定此类的协程编译生成方法。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct | AttributeTargets.Enum | AttributeTargets.Interface
public sealed class AsyncMethodBuilderAttribute : Attribute {
    public AsyncMethodBuilderAttribute(Type builderType);
    public Type BuilderType { get; }
}// 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!! 爱表哥，爱生活!!!】
```

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

- 【服务端】是自己最不熟悉的模块。活宝妹可以啃下安卓，可以写游戏，学习和熟悉游戏引擎都没问题。学习上或是运动上，活宝妹喜欢像运行员一样可以挑战运动极限。活宝妹只有一样，那就是活宝妹就是一定要嫁给亲爱的表哥!!!
- 【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。
- 同步，需要把所涉及的为方便服务端各服务器初始化而定义的各接口，实现类，以及用法弄明白。

## 6.1 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

### 6.1.1 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();
    }
}
```

### 6.1.2 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);
}
```

### 6.1.3 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() {
    }
}
```

### 6.1.4 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);
}
```

## 6.2 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？
    }
}
```

## 6.3 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;
    }
}
```

## 6.4 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");
    }
}
```

## 6.5 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【Con
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;
        }
    }
}
```

## 6.6 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() { }
    }
```

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

- 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？
- 今天大概就只能读到这里了，剩下的明天上午再读。。。
- 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。

## 6.8 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; }
}
```

## 6.9 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();
    }
}
```

## 6.10 StartSceneConfigCategory : 【Matchs!】ConfigSingleton<StartSceneCon 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;
            }
        }
    }
}
```

# 7 Actor 消息相关

- 下面套一点儿网络上的总结，可以帮助自己从比较高的层次上来理解和区分细节。

- 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。

## 7.1 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;
    }
}
```

## 7.2 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>(); // 这桢更新里：待发送给的（接收者 rpcId）接收者
}
```

## 7.3 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：消息要抛
```

```
                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，实际中存
            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);
        }
        //  【艰森诲涩难懂!!】是更底层的实现细节，它封装帮助实现 ET7 里消息超时自动过滤抛异常、返回消息的底层封装自动回复、封装了异
        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。用这个异步作务实例，去创建下面的消息发送器实例。这里的 IActorResponse T 应该只是一个索引。因为前
            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（同步方法狠快），实际是等处理发送消息的服，处理完并写好返回消息，同步到异步任务。
        // 那个 ETTask 里的回调 callback，是怎么回调的？这里 Tcs 没有设置任何回调。ETTask 里所谓回调，是执行异步状态机的下一步，没有
        // 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!!!】

## 7.4 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;
}
```

## 7.5 LocationProxyComponentSystem:

- 为什么要加那堆什么也没曾看懂的源码在那里?

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

## 7.6 ：一个添加位置信息的请求消息处理类，示例

## 7.7 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;
}
```

## 7.8 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;
 }
```

## 7.9 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;
        }
    }
}
```

## 7.10 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;
    }
}
```

## 7.11 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。

## 7.12 MailboxType

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

## 7.13 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
// ....
}
```

## 7.14 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);
        }
    }
}
```

## 7.15 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;
    }
}
```

## 7.16 【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");
    }
}
```

## 7.17 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; }
}
```

## 7.18 【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}");
                }
            }
        }
```

## 7.19 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;
    }
}
```