ET 框架学习笔记（二）- - 网络交互相关【爱表哥，爱生活!!! 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

deepwaterooo

August 7, 2023

Contents

1 IAwake 接口类系统，IStart 重构丢了

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

1.1 IMessage,IRequest,IResponse: 进程内的消息类。

- 这些接口的实现类是进程内的。一个使用的地方可以去看 NetServerComponentOnReadEvent.cs 类里的方法。
- 是一个进程内的消息，就直接调用（消息自带的会话框）会话框 OnResponse() 方法处理。

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

1.2 IActorMessage,IActorRequest,IActorResponse: 进程间的？消息类

- 这些，应该是就是【进程间消息】，必须使用消息处理器

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

1.3 IActorLocationMessage: 进程间的位置消息相关

- 【进程间位置相关消息】: 更具体一点儿。

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

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

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

1.5 ILoad,ISystemType: 加载系

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

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

1.6 IAwake: 最多可以带四个参数

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

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

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

- 参考项目：除了原文件放在 ET 域。也【复制了一份到客户端的热更新域里】。可是感觉不应该。因为其它所有的回调都不用复制就可以用。我哪里可能还是没能设置对

- 改天再检查一下。但是否，对于非系统框架扩展接口，不得不这样？仍然感觉不应该，因为系统框架里其它的生命周期回调函数都不需要复制。
- **【编译报错：】**热更新程序域里面，只能申明含有 BaseAttribute 的子类特性的类或静态类。那么也就是说，我上面的，我哪怕是把同名文件复制到热更新程序域，也是不对的，因为框架不允许这么做。我就必须去找前面，模仿它的框架系统扩展的这个方法，哪里没能连通好，为什么它的系统方法只存在 Model 域，就能运行好，而我添加的不可以？

1.8 IUpdateSystem:

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

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

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

1.10 ISingletonAwake|Update|LateUpdate: Singleton 生命周期回调

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

1.11 ISingleton,Singleton<T>: 单例

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

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

1.12 IDestroy,IDestroySystem,DestroySystem<T>: 销毁系

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

1.13 IEvent,AEvent<A>: 事件

```
public interface IEvent {
    Type Type { get; }
}
public abstract class AEvent<A>: IEvent where A: struct {
    public Type Type {
        get {
            return typeof (A);
        }
```

```
    }
    protected abstract ETTask Run(Scene scene, A a);
    public async ETTask Handle(Scene scene, A a) {
        try {
            await Run(scene, a);
        }
        catch (Exception e) {
            Log.Error(e);
        }
    }
}
```

1.14 IAddComponent: 添加组件系

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

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

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

1.16 ISerializeToEntity,IDeserialize,IDeserializeSystem,DeserializeSystem<T 序列化，反序列化

```
public interface ISerializeToEntity {
}
```

```
public interface IDeserialize {
}
public interface IDeserializeSystem: ISystemType {
    void Run(Entity o);
}
// 反序列化后执行的 System
[ObjectSystem]
public abstract class DeserializeSystem<T> : IDeserializeSystem where T: Entity, IDeserialize {
    void IDeserializeSystem.Run(Entity o) {
        this.Deserialize((T)o);
    }
    Type ISystemType.SystemType() {
        return typeof(IDeserializeSystem);
    }
    InstanceQueueIndex ISystemType.GetInstanceQueueIndex() {
        return InstanceQueueIndex.None;
    }
    Type ISystemType.Type() {
        return typeof(T);
    }
    protected abstract void Deserialize(T self);
}
```

1.17 IInvoke,AInvokeHandler<A>,AInvokeHandler<A, T>: 激活类

- 这个以前没有细看。现在修改编译错误的过程中，框架里有狠多细节的地方，需要修改的编译错误会一再崩出来，框架里出有狠多，有计时器来触发必要的超时等。所以今天，就把这类自带计时器，自动超时检测的激活系，这个功能模块理解一下。【半懂，大半懂，需要再多看几遍】
- 在以前理解了诸多标签，比如【ComponentOf(typeof())] 事件机制等，但是这个自动的激活系，一般与计时器联接紧密，要把这块儿理解透彻。

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

1.18 TimerInvokeType: 计时器可以自动触发的类型分类。

- 框架里有很多标签自动标记的标记系统。
- 这里类似。说，申明定义了这如下几类可以计时器自动触发的类型；当某个组件标记了可以计时器自动激活的标签，那么它申明的时间到，就会自动激活：某些某个特定的激活方法与逻辑，
- 如同 7/1/2023, 如果活宝妹还没能嫁给亲爱的表哥，活宝妹就解决活宝妹在亲爱的表哥的身边的小镇上的住宿问题一样，有计时器到 6/30/2023. 有激活：7/1/2023 开始找和买长期住处。希望可以一个月内解决问题，7/31/2023 可以搬进去入住。再也不想跟任何的国际贱鸡掺合，把人烦死了。。。。。

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

1.19 struct TimerCallback:

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

1.20 ATimer<T>: AInvokeHandler<TimerCallback>: 抽象类

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

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

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

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

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

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

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

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

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

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

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

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

1.23.1 IExtensible

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

1.23.2 IExtension

```
// Provides addition capability for supporting unexpected fields during
// protocol-buffer serialization/deserialization. This allows for loss-less
```

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

1.23.3 IProtoOutput<TOutput>,IMeasuredProtoOutput<TOutput>,MeasureState<T>: 看得头大

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

2 Protobuf 里的 enum: 【Identity】【Suits】【Weight】

2.1 OuterMessage_C_10001.proto 里三四个类的定义

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

```
enum Identity { // 身份
    IdentityNone = 0;
```

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

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

```
namespace ETModel {
#region Enums
    public enum Suits {
        Club = 0,
        Diamond = 1,
        Heart = 2,
        Spade = 3,
        None = 4,
    }
    public enum Weight {
        Three = 0,
        Four = 1,
        Five = 2,
        Six = 3,
        Seven = 4,
        Eight = 5,
        Nine = 6,
        Ten = 7,
        Jack = 8,
        Queen = 9,
        King = 10,
        One = 11,
        Two = 12,
        Sjoker = 13,
        Ljoker = 14,
    }
    public enum Identity {
        None = 0,
        Farmer = 1,
        Landlord = 2,
    }
#endregion
#region Messages
```

2.3 ET7 框架里，enum 完全找不到

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

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

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

2.4 ETModel_Card_Binding: 奇异点，ILRuntime 热更新里，似乎对 Card 类的两个成员变量作了辅助链接

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

```
unsafe class ETModel_Card_Binding {
    public static void Register(ILRuntime.Runtime.Enviorment.AppDomain app) {
        BindingFlags flag = BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static | BindingFlags.Declared
        MethodBase method;
        Type[] args;
        Type type = typeof(ETModel.Card);
        args = new Type[]{};
        method = type.GetMethod("GetName", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, GetName_0);
        args = new Type[]{};
        method = type.GetMethod("get_CardWeight", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_CardWeight_1);
        args = new Type[]{};
        method = type.GetMethod("get_CardSuits", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_CardSuits_2);
        args = new Type[]{};
        method = type.GetMethod("get_Parser", flag, null, args, null);
        app.RegisterCLRMethodRedirection(method, get_Parser_3);
    }
```

3 【拖拉机游戏房间】组件: 分析

3.1 TractorRoomEvent: 拖拉机房间，【待修改完成】

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

        GameObject bundleGameObject = (GameObject) ResourcesComponent.Instance.GetAsset(UIType.TractorRoom.StringToAB(), UI
```

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

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

- 【GamerComponent】玩家组件：是对一个房间里四个玩家的（及其在房间里的坐位位置）管理（分东南西北）。可以添加移除玩家。

```
// 组件: 是提供给房间用，用来管理游戏中每个房间里的最多三个当前玩家
public class GamerComponent : Entity, IAwake { // 它也有【生成系】
    private readonly Dictionary<long, int> seats = new Dictionary<long, int>();
    private readonly Gamer[] gamers = new Gamer[4];
    public Gamer LocalGamer { get; set; } // 提供给房间组件用的: 就是当前玩家。。。
    // 添加玩家
    public void Add(Gamer gamer, int seatIndex) {
        gamers[seatIndex] = gamer;
        seats[gamer.UserID] = seatIndex;
    }
    // 获取玩家
    public Gamer Get(long id) {
        int seatIndex = GetGamerSeat(id);
        if (seatIndex >= 0)
            return gamers[seatIndex];
        return null;
    }
    // 获取所有玩家
    public Gamer[] GetAll() {
        return gamers;
    }
    // 获取玩家座位索引
    public int GetGamerSeat(long id) {
        int seatIndex;
        if (seats.TryGetValue(id, out seatIndex))
            return seatIndex;
        return -1;
    }
    // 移除玩家并返回
    public Gamer Remove(long id) {
        int seatIndex = GetGamerSeat(id);
        if (seatIndex >= 0) {
            Gamer gamer = gamers[seatIndex];
            gamers[seatIndex] = null;
            seats.Remove(id);
            return gamer;
        }
        return null;
    }
    public override void Dispose() {
        if (this.IsDisposed)
            return;
        base.Dispose();
        this.LocalGamer = null;
        this.seats.Clear();
        for (int i = 0; i < this.gamers.Length; i++)
            if (gamers[i] != null) {
                gamers[i].Dispose();
                gamers[i] = null;
            }
    }
}
```

3.3 Gamer: 【服务端】一个玩家个例。对应这个玩家的相关信息

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

3.4 Gamer: 【客户端】一个玩家个例。它说只要一点儿信息就行

- 传进程间消息的时候，也只传这两个关键参数。

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

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

```
public class GamerUIComponent : Entity, IStart { // 玩家 UI 组件
    public GameObject Panel { get; private set; } // UI 面板
    // 玩家昵称
    public string NickName { get { return name.text; } }
    private Image headPhoto;
    private Text prompt;
    private Text name;
    private Text money;
    public void Start() {
        if (this.GetParent<Gamer>().IsReady)
            SetReady();
    }
    // 重置面板
    public void ResetPanel() {
        ResetPrompt();
        this.headPhoto.gameObject.SetActive(false);
        this.name.text = " 空位";
        this.money.text = "";
        this.Panel = null;
        this.prompt = null;
        this.name = null;
        this.money = null;
        this.headPhoto = null;
```

```
    }
    // 设置面板
    public void SetPanel(GameObject panel) {
        this.Panel = panel;
        // 绑定关联
        this.prompt = this.Panel.Get<GameObject>("Prompt").GetComponent<Text>();
        this.name = this.Panel.Get<GameObject>("Name").GetComponent<Text>();
        this.money = this.Panel.Get<GameObject>("Money").GetComponent<Text>();
p       this.headPhoto = this.Panel.Get<GameObject>("HeadPhoto").GetComponent<Image>();
        UpdatePanel();
    }
    // 更新面板
    public void UpdatePanel() {
        if (this.Panel != null) {
            SetUserInfo();
            headPhoto.gameObject.SetActive(false);
        }
    }
    // 设置玩家身份
    public void SetIdentity(Identity identity) {
        if (identity == Identity.None) return;
        string spriteName = $"Identity_{Enum.GetName(typeof(Identity), identity)}";
        Sprite headSprite = CardHelper.GetCardSprite(spriteName);
        headPhoto.sprite = headSprite;
        headPhoto.gameObject.SetActive(true);
    }
    // 玩家准备
    public void SetReady() {
        prompt.text = " 准备! ";
    }
    // 出牌错误
    public void SetPlayCardsError() {
        prompt.text = " 您出的牌不符合规则! ";
    }
    // 玩家不出
    public void SetDiscard() {
        prompt.text = " 不出";
    }
    // 打 2 时，玩家抢不抢庄：或者去想，玩家要不要反主牌花色
    public void SetGrab(GrabLandlordState state) {
        switch (state) {
        case GrabLandlordState.Not:
            break;
        case GrabLandlordState.Grab:
            prompt.text = " 抢地主";
            break;
        case GrabLandlordState.UnGrab:
            prompt.text = " 不抢";
            break;
        }
    }
    public void ResetPrompt() { // 重置提示
        prompt.text = "";
    }
    public void GameStart() { // 游戏开始
        ResetPrompt();
    }
    private async void SetUserInfo() { // 设置用户信息
        G2C_GetUserInfo_Ack g2C_GetUserInfo_Ack = await SessionComponent.Instance.Session.Call(new C2G_GetUserInfo_Req() {
        if (this.Panel != null) {
            name.text = g2C_GetUserInfo_Ack.NickName;
            money.text = g2C_GetUserInfo_Ack.Money.ToString();
        }
    }
    public override void Dispose() {
        if (this.IsDisposed) return;
        base.Dispose();
        ResetPanel(); // 重置玩家 UI
    }
}
```

3.6 Protobuf 里面的消息与参考

- 这里把 Protobuf 里面可以传的游戏相关也整理一下。

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

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

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

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

- 上下这一两个组件里，除了 ProtoBuf 消息里传递的类找不到，没有其它错误
- 【嵌套】：是这里的难点。其它都可以一个触发一个地由事件发布触发订阅者的回调，可是当一个组件内存在嵌套，又是系统化【内部组件生成完成后，外部组件才生成完成】生成，我是要把这两个组件合并成一个吗？还是说，我不得不把它折成粒度更小的 UI 上的事件驱动机制，以符合系统框架？要去所源码弄透。

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

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

4 ET7 数据库相关【服务端】

- 这个数据库系统，连个添加使用的范例也没有。。。就两个组件，一个管理类。什么也没留下。。
- 现框架 **DB 放在服务端的 Model** 里。它的管理体系成为管理各个不同区服的数据库 DBComponent。
- 因为找不到任何参考使用的例子。我觉得需要搜索一下。在理解了参考项目数据库模块之后，根据搜索，决定是使用原参考项目总服务器代理系，还是这种相对改装了的管理区服系统?
- 先前搜的时候，关于应用框架的数据缓存，什么时候需要一个缓存层，应用运行的时候，数据是否在内存等，为什么 ET7 框架使用 MongoDB, 就是这个这类数据库，为什么比较适合双端游戏框架，而为什么 MySQL 之类的破烂库就各种不适合？感觉这些比较上层的原理，或基础原理，自己理解得不够透彻，看过网上的别人的分析，但理解得还不够透彻。
- 我可能需要把 ET7 重构后、被破烂框架开发者各个主要模块、删除得几乎不剩下什么的模块、与重构前的 ET6 等模块，再多读一下源码，理解得透彻一点儿再来事理这个模块。现游戏里需要用数据库的地方，主要是用户帐户数据（这应该是注册登录服的逻辑），帐户管理与游戏数据需要相区分吗？账户管理，游戏数据

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

- 狠奇怪的是，框架里，居然没有一个实现这个接口的实现类?

```
public interface IDBCollection {}
```

4.2 DBComponent:

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

4.3 DBComponentSystem: 【CRUD】可以查表，查询数据等，各种数据库操作的基本方法。热更域生成系

- 它的生成系就是解决对数据库的 CRUD 必要操作，单条信息的，或是批量处理的
- 因为数据库操作的几个基本操作方法相对熟悉，这里不贴源码。只一点儿：服务端的远程数据库，仍属于是跨进程的进程间网络异步调用，所以几乎所有的方法也都异步 ETTask 包装。
- 【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!!!】

4.4 DBManagerComponent: 有上面的 DBComponent 数组。数组长度固定。

- 管理类组件：用来管理服务端不同分区里的 DBComponent 组件。
- 功能包括：根据区号，返回该区下的 DBComponent 组件，就是返回该区下的数据库，方便对该数据库进行相应的操作。
- 当引入这个区的概念，当要去取相应的区的数据库，其实也是说，小区下的所有用户的相关数据信息，应该是存放在用户所在的小区下的。这里区的概念，也就是框架的（包括数据库的）层级管理体系。

- 服务端自上而下的 Machine, Process, Scene, Zone 也算基本上都懂。可是关于区、分区、小区的概念现在仍不深入。每个小区里有什么？分区管理有什么好处呢？

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

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

```
[FriendOf(typeof(DBManagerComponent))]
public static class DBManagerComponentSystem {
    [ObjectSystem]
    public class DBManagerComponentAwakeSystem: AwakeSystem<DBManagerComponent> {
        protected override void Awake(DBManagerComponent self) {
            DBManagerComponent.Instance = self;
        }
    }
    [ObjectSystem]
    public class DBManagerComponentDestroySystem: DestroySystem<DBManagerComponent> {
        protected override void Destroy(DBManagerComponent self) {
            DBManagerComponent.Instance = null;
        }
    }
    public static DBComponent GetZoneDB(this DBManagerComponent self, int zone) {
        DBComponent dbComponent = self.DBComponents[zone];
        if (dbComponent != null)
            return dbComponent;
        StartZoneConfig startZoneConfig = StartZoneConfigCategory.Instance.Get(zone);
        if (startZoneConfig.DBConnection == "")
            throw new Exception($"zone: {zone} not found mongo connect string");
        dbComponent = self.AddChild<DBComponent, string, string, int>(startZoneConfig.DBConnection, startZoneConfig.DBName,
        self.DBComponents[zone] = dbComponent;
        return dbComponent;
    }
}
```

4.6 DBProxyComponent: 【参考项目】里的。有生成系。

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

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

5 位置服 LocationComponent 组件：

5.1 先前版本 LocationComponent 原理分析

- 框架的应用场景里，知道对方的 InstanceId 就可以给对方发消息。
- 问题是，对方的可以下线再上线，活宝妹可以从加州地图服重入亲爱的表哥所在的 WA 地图服（不同州的地图服服务器进程不一样），对方的 InstanceId 是变化的，小伙伴也可以搬家，搬家过程中位置不确认，还要先锁住，搬完才实时更新位置服管家。

- 【框架需求】：InstanceId 标识唯一身份。但仍需要对 Entity 级别（框架的最底最基类封装）不同对象的 InstanceId 进行管理（因为不同游戏实现里，可能会分线、可能会分地图服，不同地图服处于不同进程。一旦服务器进程变了，就需要对管家实时更新：更新要搬家，更新搬家完成了，位置确定了，活宝妹就是一定要嫁给亲爱的表哥!!）。所以会有当前【位置服】。

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

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

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

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

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

- 【亲爱的表哥的活宝妹，小呀头片子有点儿叨钻，可是上面想的都是对的，两分钟这个模块的几个编译错误全不见了。。。活宝妹就是一定要嫁给亲爱的表哥!!!】

- 【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要嫁给亲爱的表哥!! 爱表哥，爱生活!!!】

5.2 LocationComponent:

```
namespace ET.Server {
// 这个【ActorLocation】文件夹：原本只是没有 ActorLocationSenderOneType.cs 类。不曾细看【跨进程位置】相关
//  【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!!!】
    [ChildOf(typeof(LocationComponent))] //  【位置服】的组件：
    public class LockInfo: Entity, IAwake<long, CoroutineLock>, IDestroy { // 打包：协程锁的实例标记号 + 独占锁
    public long LockInstanceId;
    public CoroutineLock CoroutineLock;
    }
    [ComponentOf(typeof(Scene))]
    //  【位置组件】：去细看两个字典，所做的具体的事情。这里是，被自己弄丢了文件，还是自己添加了这个位置服，但是没整合生成系？去看源码，
    // 没整合生成系：就是这里 Model 域里定义了说有个【位置管理组件】，但是热更域里什么也没有，没有定义执行逻辑，需要一个 LocationCom
    public class LocationComponent: Entity, IAwake {
        public readonly Dictionary<long, long> locations = new Dictionary<long, long>();
        public readonly Dictionary<long, LockInfo> lockInfos = new Dictionary<long, LockInfo>();
        }
    }
}
```

5.3 LocationComponentSystem: 源码里是有这个文件的

- 几个最主要的方法，看起来狠简单。
- 两处：【异步返回类型】感觉相对生疏。

```
[FriendOf(typeof(LocationComponent))]
[FriendOf(typeof(LockInfo))]
public static class LocationComponentSystem {
// 添加【注册】的是：【被锁 actorId，当前——独占锁的实例标记号】这里好像被我写错了，值，应该是，被查小伙伴所在进程的地址
    public static async ETTask Add(this LocationComponent self, long key, long instanceId) {
        using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Location, key)) {
            self.locations[key] = instanceId;
            Log.Info($"location add key: {key} instanceId: {instanceId}");
        }
    }
    // 【下线销号移除】:
    public static async ETTask Remove(this LocationComponent self, long key) {
        using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Location, key)) {
            self.locations.Remove(key);
            Log.Info($"location remove key: {key}");
        }
    } // 【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!!!】
    // 【小伙伴云游】：上报、预报上锁时长
    public static async ETTask Lock(this LocationComponent self, long key, long instanceId, int time = 0) { // 忘记这个 Key
        // 【入队列站列排号】：直到获得异步资源，标记是拿到一把【独占锁】
        CoroutineLock coroutineLock = await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Location, key);
        LockInfo lockInfo = self.AddChild<LockInfo, long, CoroutineLock>(instanceId, coroutineLock);
        self.lockInfos.Add(key, lockInfo); // 再封装：【被锁 actorId, lock 结构体】，封装的是真正实时正在锁着的时长、过程
        Log.Info($"location lock key: {key} instanceId: {instanceId}");
// 小伙伴云游上报：要 2-617-1314 分钟【爱表哥，爱生活!!! 任何时候，亲爱的表哥的活宝妹就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表
        if (time > 0) {
            async ETTask TimeWaitAsync() { // 这里没懂：怎么还有个 ETTask 返回类型呢？  // <<<<<<<<<<<<<<<<<<<
                long lockInfoInstanceId = lockInfo.InstanceId; // 先记下：当前被锁资源【独占锁】的实例标记号
// 【异步等待】：被要求的时长。它返回【ETTask】类型 ==》这里可以决定这个内部局部异步方法的返回类型吗？
                await TimerComponent.Instance.WaitAsync(time);
                if (lockInfo.InstanceId != lockInfoInstanceId) // 再检查：独占锁的实例标记号，是否变了？什么情况下，有可能会变呢
                    return;
                Log.Info($"location timeout unlock key: {key} instanceId: {instanceId} newInstanceId: {instanceId}");
                self.UnLock(key, instanceId, instanceId);
            } // 上面：它定义了这个内部方法，只执行了一次，只等待了一次要求的时长，应该不是无限上锁直到再上报的
            TimeWaitAsync().Coroutine();
        }
    }
    public static void UnLock(this LocationComponent self, long key, long oldInstanceId, long newInstanceId) { // 解锁
        if (!self.lockInfos.TryGetValue(key, out LockInfo lockInfo)) { // 先检查几个异常
            Log.Error($"location unlock not found key: {key} {oldInstanceId}");
            return;
        }
        if (oldInstanceId != lockInfo.LockInstanceId) {
            Log.Error($"location unlock oldInstanceId is different: {key} {oldInstanceId}");
            return;
        }
        Log.Info($"location unlock key: {key} instanceId: {oldInstanceId} newInstanceId: {newInstanceId}");
        self.locations[key] = newInstanceId; // 写入小本：【被锁 actorId，被锁 actorId 所在的进程号】为的是方便它是必消息到被查地
        self.lockInfos.Remove(key); // 先从字典管理中移除
        // 解锁：就是回收掉了呀
        lockInfo.Dispose();
    }
    // 这些异步返回类型，看着还理解不太顺。。。
    public static async ETTask<long> Get(this LocationComponent self, long key) { // 查询：位置信息  // <<<<<<<<<<<<<<<<<<<<
        using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Location, key)) { // 挂号排队站队锁：这个步骤，是
            self.locations.TryGetValue(key, out long instanceId);
            Log.Info($"location get key: {key} instanceId: {instanceId}");
            return instanceId; // 返回 ETTask<lomg> 因为被排队挂号异步等待过
        }
    }
}
```

5.4 LocationProxyComponent:

- 不知道我先前对【SceneType.Process】的理解与定义，是否正确?：感觉它像是一台物理机 N 个核中每个进程上【1-M】个场景中最特殊的一个场景【SceneType.Process】。亲爱的表哥的活宝妹，把它理解成了：同一进程中多线程环境中的主线程。主线程一定存在，每个核都会有一个【SceneType.Process】场景；主线程要同步异步线程结果等，所以这个场景极为特殊。不知道对了没?
- 这里以一个核一个进程为单位。它是添加在每个进程上【SceneType.Process】，因为其特殊性，可供可代理【1-N】个场景需求与使用?
- 是【SceneType.Process】层面上添加了这个代理组件。去想，N 台物理机，每台 M 个核，共 N Œ M 个进程代理，与 N Œ M 中某一个进程下（可能有的【1-X】个不同场景不同小服中）的一条线程场景【SceneType.Location】【位置服】，与共 N Œ M 个进程代理，之间的链接。

```
[ComponentOf(typeof(Scene))] // 每个场景上？有个位置服代理？组件是添加在每个核每个进程的【SceneType.Process】进程场景上；全【服务
public class LocationProxyComponent: Entity, IAwake, IDestroy {
    [StaticField]
    public static LocationProxyComponent Instance; // 每个场景上，一个实例
}
```

5.5 LocationProxyComponentSystem:

- 现在做的事情是：在确认【位置服】位置组件正确，各场景里位置代码组件源码正确的基础上，把这个先前一直以为被框架源破坏者删除得什么也不剩下的模块，理解透彻。会去看【位置服】位置组件几个主要需要（位置的注册、更新、查询、小伙伴搬家上报上锁，搬完上报解锁等主要相关逻辑）既然对比完了两个主要系，提上去，笔记本上看起来更方便，把这个模块看完。【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!!!】
- 【位置服】是 SceneType.Location; 位置组件 LocationComponent 是管理类组件。一个位置服拥有一个位置组件；谁说一定是一个进程一个位置服？几台物理机只一个【位置服】也没问题呀?! 就是几台物理机 N*M 个核，只某个核的可能存在的【1-X】个场景中（特殊主线程场景 Process + Location,etc），拥有一个【位置服】SceneType.Location, 应该也是合理的，看服务端需求来。并且，并不是说就一定只有一个【位置服】，不是说可以分身分线什么的吗，就是一个【位置服】同时开 X 台线程。。。。
- 【细节注意的地方】：看懂 ETTask 之后，异步的逻辑能够相对明白一些。这里，要考虑【服务端】总共 MN 个核，可能有的一个或几个【位置服】，每个核的每个进程场景上一个【位置服代理】，需要注意帮助类与这里 LocationComponentSystem 逻辑桥接连接到不多不少。就是一部分【参考项目】LocationComponent 里的方法逻辑，可能已经被每个进程场景上一个【位置服代理】帮助类封装担去一部分，【服务端】总共的 1 个或是几个【位置服】只需要处理它不得不处理的最少精减逻辑。
- 【参考项目】里，位置服管理进程上、跨进程中央邮政所有小伙伴实例信息。它把数据写进数据库。上面重构，仍需要考虑，使用重构后的小区 DBProxy 代理实时到数据库。数据库应该也有一个专服？好像没有专服。知道分区代理。重构后的数据库相关，我消灭了编译错误，可是看来没有理解透，**重构后，【位置服】对数据的管理，是不需要再写进数据库的吗？**这个问题，可以自己消灭所有的编译错误后再回来做。
- 自己对【位置服】这个模块的理解基本正确，几处小地方不是太懂，比如两个异步返回类型，两处添加位置信息到【位置服】方法调用，但是都会弄明白的。
- **【明天上午看】**：现在还存在编译错误，感觉仍然需要再去读源码的、**不管是命令行启动服务器，还是怎么样，服务端场景的初始化配置相关**。现在是能够把物理机、内个核，核里的多场景，基本想明白。可是 **【服务端起始】初始化配置**，还要再去读。明天上午读。

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
- 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。
- 从这行开始，下面小章节之前，转载网络上的关于 ET6.0 的服务端配置或是启动的理解。（网络上的也未必正确）
- 它说 ET6.0, 服务端配置依赖几个配置表格，就是框架里有 Excel 读与解析的工具模块，是从表格配置服务器的。应该也是说服务端最初第一次启动的时候。因为服务端启动后，几乎可以做到永远不关服。【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!!! 活宝妹还没能嫁给亲爱的表哥，活宝妹就永远也不再离开这里，会永远守候在亲爱的表哥这里!!!】
- 自己文件夹里简单翻了一下：ET7 里不再是 Excel 表格，而是 Json 格式的 X.txt 配置文件。举个例子：StartMachineConfig.txt:

```
{"list":[
    {"_t":"StartMachineConfig","_id":1,"InnerIP":"127.0.0.1","OuterIP":"127.0.0.1","WatcherPort":"10000"},
]}
```

- 框架里规则定义了的几类文件的功能，分述如下：
  - 【StartMachineConfig】：启动的物理机器配置，每一条代表一台物理机，包括 id，内外网地址，以及一个守护进程端口
  - 【StartZoneConfig】：相关游戏区的配置，现在里面已经写好的配置，每一条代表一个区连接的数据库配置
  - 【StartProcessConfig】：启动的进程配置，当启动一个 ET6.0 的服务器时，会根据传入的数据读取这个配置确定服务器应该以什么样的配置运行当前进程
  - 【StartSceneConfig】：启动服务器时，这个表里面的配置，决定了给这个服务器添加什么类型的服务功能（暂且理解为，其中的一个场景就是一种服务，对应有：Realm 服（用于认证玩家），Gate 服（经过认证后所有与客户端通信），Location 服（用于提供查找 Actor 定位服），Map 服（用于管理玩家游戏实体，互相通信））。注意 Process 字段表示所属的进程，Zone 字段表示游戏区的概念。如果这里的 Process 填的进程 id 与 StartProcessConfig 对不上，那么没有任何服务功能会在启动的服务器里。

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

- 【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。
- 需要理解【服务端】命令行启动起始的配置过程。

```
[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[]>> {
//  【服务端】命令行的启动、起始，的大致过程:
    // 命令行，各种参数，下发命令，启动服务端【这里细节去找】: 是以物理机、进程、场景、Zone 为启动单位，可以在各个不同层面启动的吗？还
    // 框架里封装过的命令行解析: 解析入 Options 单例类，以及如下【四大文件】主要管理，以及其它小兵小将文件，如 UnitConfigCategory.
    // 这个文件提供两种查询方式: 拿到全局所有配置，和，可以来拿来读某种特例配置。拿的方式是: 这里去到了初始配置时写的几个文件里去读，
    public override Dictionary<Type, byte[]> Handle(ConfigComponent.GetAllConfigBytes args) {
        Dictionary<Type, byte[]> output = new Dictionary<Type, byte[]>(); // 准备一个小本字典: 准备记笔记
        // 它分了这几类来管理，然后就把【服务端】起来时的各种配置，写入了这几类文件。要用的时候，从保存过的这些配置文件里去读。。。?
        List<string> startConfigs = new List<string>() { // 几个类型的区分: 去看对这几种类型的管理，什么内容添加进了它们的管理里
            "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【Config】标签
        HashSet<Type> configTypes = EventSystem.Instance.GetTypes(typeof (ConfigAttribute)); // 【Config】标签：返回程序域里所
        foreach (Type configType in configTypes) {
            string configFilePath;
//  【路径中的参数】：Options.Instance，不是说是从命令行参数解析进单例类里的吗？那么【服务端】最初的激活启动是来自于命令行的命令。【服
            if (startConfigs.Contains(configType.Name)) { //  【单例管理类型】：有特异性的配置路径，读的都是【仅服务端】文件夹中的
                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。框架里保存过的配置包括：  UnitConfigCategory.bytes 和另外一个
                configFilePath = $"../Config/Excel/s/{configType.Name}.bytes";
            }
            output[configType] = File.ReadAllBytes(configFilePath); // 把读到的配置，一条一条记入小本
        }
        return output; // 返回所有配置
    }
}
[Invoke] // 这个标签的激活：现在框架里，好像还不存在，任何一个已经写过的 {args.ConfigName}.bytes 的例子，还没用到？
public class GetOneConfigBytes: AInvokeHandler<ConfigComponent.GetOneConfigBytes, byte[]> {
    public override byte[] Handle(ConfigComponent.GetOneConfigBytes args) {
        // 【Invoke 回调逻辑】：从框架特定位置，读取特定属性条款的配置，返回字节数组
        byte[] configBytes = File.ReadAllBytes($"../Config/{args.ConfigName}.bytes");
        return configBytes;
    }
}
```

6.4 ConfigLoader:【客户端】:【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!!!】

- 【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。
- **加载【配置资源包】的过程**：任何一个资源包，都可能会有 N 级嵌套的依赖包。方法里会去拿所有的依赖包并拓扑排序（嵌套层次最深、被依赖得层次最深的资源包，永远排最前、最先加载）。返回的排序依赖包，也包含了当前包，当前包排在最后。并按拓扑排序一一加载。**【活宝妹亲爱的表哥，永远是活宝妹的最最依赖的依赖包!! 永远排第一位。活宝妹的亲爱的表哥来找亲爱的表哥的活宝妹，一起手牵手去领结婚证，是所有资源包的头儿，是接下来后续一切美好的开始 ~~!!! 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!!!】**
- 这里，还没能找到热更新资源服务端，动态下载热更新资源包的逻辑。可能在其它模块。

```
[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 { // 这个分支：先花点儿时间，走一遍真正的【客户端】(而非编辑器模式) 从资源包读配置的过程，从哪里下载的资源包，以及找
            using (Root.Instance.Scene.AddComponent<ResourcesComponent>()) { // 这里只是： using 结束后，自动回收垃圾
                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？(这个不是这个组件的定义里，在被激活的一个类的拿全局所有配置的被激活方法里)，并 Invoke 一两个激活类里定义过的方法，来从全局配置文件（本地，特定路径，或是【客户端】从【热更新】资源包加载【配置资源包】）读取相关配置再返回。

- 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）
- 知道这些狠多的配置是单例类。仍然迷糊的是：这些文件这些配置管理，它们属于哪个功能模块，哪个场景或是进程或是应用，它们是谁，是如何发挥作用的？可以明白，**根据热更新过的资源包不关服热更新成不同配置，不关服热更新服务端配置**。它是 DOTNET 服务端应用吗？它是管理总后台？【这个要，查看明白】
- 细节上：不明白的是：**【配置字节数据的反序列化】**的必要，进一步去想，【服务端】N 台物理机，每台 M 个核，每个核一再添加多线程 SceneType.XScene 的启动过程？
- 所以要去想，【服务端】的启动过程，它们的配置，是否由每个核、每个进程上的各个小服，如【动态路由系统】般，小伙伴云游般，跨进程？各自上报，到管理总后台的？如此，【序列化】【反序列】才更好理解。要去框架里去找。。。
- Deserialize【序列化】与【反序列化】:【服务端】各小服的启动、配置【自底向上】自动上报、集合的过程???【服务端】根据必要的【热更新资源所】【自顶向下】不关服、动态更新配置的过程?
- 【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
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) { //  【服务端的、不关服、热更新配置】: 是这个方法里实现的吗？
        this.allConfig.TryGetValue(configType, out ISingleton oneConfig);// oneConfig: 这里算是自定义变量的【申明与赋值】?
// 细节: 这两行逻辑，实际已经实现了？【服务端不关服热更新】。这里说，以前加载过的，就见鬼去吧。。重新加载新的。就可以实现重新加载热更新
        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
        byte[] oneConfigBytes = EventSystem.Instance.Invoke<GetOneConfigBytes, byte[]>(new GetOneConfigBytes() {ConfigName
        // 下面的 Deserialize: 不知道这些配置管理是 dotnet 服务端管理总后台？为什么需要反序列化？知道 byte 转对象有个步骤，可是反
        // 所以要去想，【服务端】的启动过程，它们的配置，是否由每个核、每个进程上的各个小服，如【动态路由系统】般，小伙伴云游般，跨进
        // Deserialize【序列化】与【反序列化】:【服务端】各小服的启动、配置【自底向上】自动上报、集合的过程???【服务端】根据必要的【
        //  【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!!!】
        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); // 每种配置: 开一个线程去处理
        }
    }
    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); // 先反序列化
        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() { // 这里就是想要桥接：ProtoObject 里所实现过的【初始化前后】可以做的事情，接口，给框架使
    }
    public virtual void Dispose() {
    }
}
```

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

- 它，是不是，开启一台物理机的配置呢？再读，来回答这里 Y N
- 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<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;
            }
        }
    }
}
```