ET 框架学习笔记（二） - - 网络交互相关

deepwaterooo

May 26, 2023

Contents

1 Actor 消息相关

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

```
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; }
    public bool NeedException { get; }
    public ETTask<IActorResponse> Tcs { get; }
    public ActorMessageSender(long actorId, IActorRequest iActorRequest, ETTask<IActorResponse> tcs, bool needException) {
    }
}
```

1.2 ActorMessageSenderComponent:

```
[ComponentOf(typeof(Scene))]
public class ActorMessageSenderComponent: Entity, IAwake, IDestroy {
    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, ActorMessageSende
    public long TimeoutCheckTimer;
    public List<int> TimeoutActorMessageSenders = new List<int>();
}
```

1.3 ActorMessageSenderComponentSystem: 这个类，今天晚上没有看懂，明天上午再看一下

```
[FriendOf(typeof(ActorMessageSenderComponent))]
public static class ActorMessageSenderComponentSystem {
    [Invoke(TimerInvokeType.ActorMessageSenderChecker)] // 另一个新标签，激活系
    public class ActorMessageSenderChecker: ATimer<ActorMessageSenderComponent> {
        protected override void Run(ActorMessageSenderComponent self) {
            try {
                self.Check();
            }
            catch (Exception e) {
                Log.Error($"move timer error: {self.Id}\n{e}");
            }
        }
    }
    // [ObjectSystem] // Awake() etc
    private static void Run(ActorMessageSender self, IActorResponse response) {
        if (response.Error == ErrorCore.ERR_ActorTimeout) {
            self.Tcs.SetException(new Exception($"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;
        }
        if (self.NeedException && ErrorCore.IsRpcNeedThrowException(response.Error)) {
            self.Tcs.SetException(new Exception($"Rpc error: actorId: {self.ActorId} request: {self.Request}, response: {re
            return;
        }
        self.Tcs.SetResult(response);
    }
    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
            if (timeNow < value.CreateTime + ActorMessageSenderComponent.TIMEOUT_TIME)
                break;
            self.TimeoutActorMessageSenders.Add(key);
        }
        foreach (int rpcId in self.TimeoutActorMessageSenders) {
            ActorMessageSender actorMessageSender = self.requestCallback[rpcId];
            self.requestCallback.Remove(rpcId);
            try {
                IActorResponse response = ActorHelper.CreateResponse(actorMessageSender.Request, ErrorCore.ERR_ActorTimeout
                Run(actorMessageSender, response);
            }
            catch (Exception e) {
                Log.Error(e.ToString());
            }
        }
        self.TimeoutActorMessageSenders.Clear();
    }
    public static void Send(this ActorMessageSenderComponent self, long actorId, IMessage message) {
        if (actorId == 0) {
            throw new Exception($"actor id is 0: {message}");
        }
        ProcessActorId processActorId = new(actorId);
        // 这里做了优化，如果发向同一个进程，则直接处理，不需要通过网络层
        if (processActorId.Process == Options.Instance.Process) {
            NetInnerComponent.Instance.HandleMessage(actorId, message);
            return;
        }
        Session session = NetInnerComponent.Instance.Get(processActorId.Process);
        session.Send(processActorId.ActorId, message);
    }
    public static int GetRpcId(this ActorMessageSenderComponent self) {
        return ++self.RpcId;
    }
    public static async ETTask<IActorResponse> Call(
        this ActorMessageSenderComponent self,
        long actorId,
        IActorRequest request,
        bool needException = true
        ) {
        request.RpcId = self.GetRpcId();
        if (actorId == 0) {
            throw new Exception($"actor id is 0: {request}");
        }
        return await self.Call(actorId, request.RpcId, request, needException);
    }
    public static async ETTask<IActorResponse> Call(
        this ActorMessageSenderComponent self,
        long actorId,
        int rpcId,
        IActorRequest iActorRequest,
        bool needException = true
        ) {
        if (actorId == 0) {
            throw new Exception($"actor id is 0: {iActorRequest}");
        }
        var tcs = ETTask<IActorResponse>.Create(true);
        self.requestCallback.Add(rpcId, new ActorMessageSender(actorId, iActorRequest, tcs, needException));
        self.Send(actorId, iActorRequest);
        long beginTime = TimeHelper.ServerFrameTime();
        IActorResponse response = await tcs;
        long endTime = TimeHelper.ServerFrameTime();
        long costTime = endTime - beginTime;
        if (costTime > 200) {
            Log.Warning($"actor rpc time > 200: {costTime} {iActorRequest}");
        }
        return response;
    }
    public static void HandleIActorResponse(this ActorMessageSenderComponent self, IActorResponse response) {
        ActorMessageSender actorMessageSender;
        if (!self.requestCallback.TryGetValue(response.RpcId, out actorMessageSender)) {
            return;
        }
        self.requestCallback.Remove(response.RpcId);
        Run(actorMessageSender, response);
```

```
    }
}
```

1.4 LocationProxyComponent

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

1.5 LocationProxyComponentSystem

```
// [ObjectSystem] awake() etc
public static class LocationProxyComponentSystem {
    private static long GetLocationSceneId(long key) {
        return StartSceneConfigCategory.Instance.LocationConfig.InstanceId;
    }
    public static async ETTask Add(this LocationProxyComponent self, long key, long instanceId) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                  new ObjectAddRequest() { Key = key, InstanceId = instanceId });
    }
    public static async ETTask Lock(this LocationProxyComponent self, long key, long instanceId, int time = 60000) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                  new ObjectLockRequest() { Key = key, InstanceId = instanceId, Time = time });
    }
    public static async ETTask UnLock(this LocationProxyComponent self, long key, long oldInstanceId, long instanceId) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                  new ObjectUnLockRequest() { Key = key, OldInstanceId = oldInstanceId, InstanceId = instanceId });
    }
    public static async ETTask Remove(this LocationProxyComponent self, long key) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                  new ObjectRemoveRequest() { Key = key });
    }
    public static async ETTask<long> Get(this LocationProxyComponent self, long key) {
        if (key == 0)
            throw new Exception($"get location key 0");
        // location server 配置到共享区，一个大战区可以配置 N 多个 location server，这里暂时为 1
        ObjectGetResponse response = (ObjectGetResponse) await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                new ObjectGetRequest() { Key = key });
        return response.InstanceId;
    }
    public static async ETTask AddLocation(this Entity self) {
        await LocationProxyComponent.Instance.Add(self.Id, self.InstanceId);
    }
    public static async ETTask RemoveLocation(this Entity self) {
        await LocationProxyComponent.Instance.Remove(self.Id);
    }
}
```

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

```
[ChildOf(typeof(ActorLocationSenderComponent))]
public class ActorLocationSender: Entity, IAwake, IDestroy {
    public long ActorId;
    public long LastSendOrRecvTime; // 最近接收或者发送消息的时间
    public int Error;
}
```

1.7 ActorLocationSenderComponent: 位置发送组件

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

```
    public static ActorLocationSenderComponent Instance { get; set; }
    public long CheckTimer;
}
```

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

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

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

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

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

1.10 ActorMessageDispatcherInfo | ActorMessageDispatcherComponent

```
public class ActorMessageDispatcherInfo {
    public SceneType SceneType { get; }
    public IMActorHandler IMActorHandler { get; }
    public ActorMessageDispatcherInfo(SceneType sceneType, IMActorHandler imActorHandler) {
        this.SceneType = sceneType;
        this.IMActorHandler = imActorHandler;
    }
}
[ComponentOf(typeof(Scene))] // Actor 消息分发组件
```

```
public class ActorMessageDispatcherComponent: Entity, IAwake, IDestroy, ILoad {
    [StaticField]
    public static ActorMessageDispatcherComponent Instance;
    public readonly Dictionary<Type, List<ActorMessageDispatcherInfo>> ActorMessageHandlers = new();
}
```

1.11 ActorMessageDispatcherComponentHelper: 感觉名字不系统化，不知道是不是自己干的

```
[FriendOf(typeof(ActorMessageDispatcherComponent))] // Actor 消息分发组件
public static class ActorMessageDispatcherComponentHelper {
    // [ObjectSystem] // awake() etc
    private static void Awake(this ActorMessageDispatcherComponent self) {
        self.Load();
    }
    private static void Load(this ActorMessageDispatcherComponent self) {
        self.ActorMessageHandlers.Clear();
        var types = EventSystem.Instance.GetTypes(typeof (ActorMessageHandlerAttribute));
        foreach (Type type in types) {
            object obj = Activator.CreateInstance(type);
            IMActorHandler imHandler = obj as IMActorHandler;
            if (imHandler == null) {
                throw new Exception($"message handler not inherit IMActorHandler abstract class: {obj.GetType().FullName}")
            }

            object[] attrs = type.GetCustomAttributes(typeof(ActorMessageHandlerAttribute), false);
            foreach (object attr in attrs) {
                ActorMessageHandlerAttribute actorMessageHandlerAttribute = attr as ActorMessageHandlerAttribute;
                Type messageType = imHandler.GetRequestType();
                Type handleResponseType = imHandler.GetResponseType();
                if (handleResponseType != null) {
                    Type responseType = OpcodeTypeComponent.Instance.GetResponseType(messageType);
                    if (handleResponseType != responseType) {
                        throw new Exception($"message handler response type error: {messageType.FullName}");
                    }
                }
                ActorMessageDispatcherInfo actorMessageDispatcherInfo = new(actorMessageHandlerAttribute.SceneType, imHandl
                self.RegisterHandler(messageType, actorMessageDispatcherInfo);
            }
        }
    }
    private static void RegisterHandler(this ActorMessageDispatcherComponent self, Type type, ActorMessageDispatcherInfo ha
        if (!self.ActorMessageHandlers.ContainsKey(type))
            self.ActorMessageHandlers.Add(type, new List<ActorMessageDispatcherInfo>());
        self.ActorMessageHandlers[type].Add(handler);
    }
    public static async ETTask Handle(this ActorMessageDispatcherComponent self, Entity entity, int fromProcess, object mes
        List<ActorMessageDispatcherInfo> list;
        if (!self.ActorMessageHandlers.TryGetValue(message.GetType(), out list))
            throw new Exception($"not found message handler: {message}");
        SceneType sceneType = entity.DomainScene().SceneType;
        foreach (ActorMessageDispatcherInfo actorMessageDispatcherInfo in list) {
            if (actorMessageDispatcherInfo.SceneType != sceneType)
                continue;
            await actorMessageDispatcherInfo.IMActorHandler.Handle(entity, fromProcess, message);
        }
    }
}
```

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

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

1.13 【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。

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

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

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

1.15 MailboxType

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

1.16 【服务端】ActorHandleHelper 帮助类。【需要去深挖一下】

```
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
            // NetInnerComponent.Instance.HandleMessage(realActorId, response); // 等同于直接调用下面这句【我自己暂时放回来的】
            ActorMessageSenderComponent.Instance.HandleIActorResponse(response);
            return;
        }
        Session replySession = NetInnerComponent.Instance.Get(fromProcess);
        replySession.Send(response);
    }
    public static void HandleIActorResponse(IActorResponse response) {
        ActorMessageSenderComponent.Instance.HandleIActorResponse(response);
    }
    // 分发 actor 消息
    [EnableAccessEntiyChild]
    public static async ETTask HandleIActorRequest(long actorId, IActorRequest iActorRequest) {
        InstanceIdStruct instanceIdStruct = new(actorId);
        int fromProcess = instanceIdStruct.Process;
        instanceIdStruct.Process = Options.Instance.Process;
        long realActorId = instanceIdStruct.ToLong();
        Entity entity = Root.Instance.Get(realActorId);
        if (entity == null) {
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
        if (mailBoxComponent == null) {
            Log.Warning($"actor not found mailbox: {entity.GetType().Name} {realActorId} {iActorRequest}");
            IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
            Reply(fromProcess, response);
            return;
        }
        switch (mailBoxComponent.MailboxType) {
            case MailboxType.MessageDispatcher: {
```

```
                using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.Mailbox, realActorId)) {
                    if (entity.InstanceId != realActorId) {
                        IActorResponse response = ActorHelper.CreateResponse(iActorRequest, ErrorCore.ERR_NotFoundActor);
                        Reply(fromProcess, response);
                        break;
                    }
                    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}");
        }
    }
}
```

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

2.1 ConfigSingleton<T>: ProtoObject, ISingleton

```
public abstract class ConfigSingleton<T>: ProtoObject, ISingleton where T: ConfigSingleton<T>, new() {
        [StaticField]
        private static T instance;
        public static T Instance {
            get {
```

```
                return instance ??= ConfigComponent.Instance.LoadOneConfig(typeof (T)) as T;
            }
        }
        void ISingleton.Register() {
            if (instance != null) {
                throw new Exception($"singleton register twice! {typeof (T).Name}");
            }
            instance = (T)this;
        }
        void ISingleton.Destroy() {
            T t = instance;
            instance = null;
            t.Dispose();
        }
        bool ISingleton.IsDisposed() {
            throw new NotImplementedException();
        }
        public override void AfterEndInit() { }
        public virtual void Dispose() { }
    }
```

2.2 SceneFactory 里可以给【匹配服】添加组件

```
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);
        scene.AddComponent<MailBoxComponent, MailboxType>(MailboxType.UnOrderMessageDispatcher);
        switch (scene.SceneType) {
        case SceneType.Router:
            scene.AddComponent<RouterComponent, IPEndPoint, string>(startSceneConfig.OuterIPPort, startSceneConfig.StartPro
            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:
            scene.AddComponent<NetServerComponent, IPEndPoint>(startSceneConfig.InnerIPOutPort);
            break;
        case SceneType.Match: // <<<<<<<<<<<<<<<<<<<< 这里是，我可以添加【匹配服】相关功能组件的地方。【参考项目原原码】感觉被我弄
            break;
        case SceneType.Gate:
            scene.AddComponent<NetServerComponent, IPEndPoint>(startSceneConfig.InnerIPOutPort);
            scene.AddComponent<PlayerComponent>();
            scene.AddComponent<GateSessionKeyComponent>();
            break;
        case SceneType.Map:
            scene.AddComponent<UnitComponent>();
            scene.AddComponent<AOIManagerComponent>();
            break;
        case SceneType.Location:
            scene.AddComponent<LocationComponent>();
            break;
//...
        }
        return scene;
    }
}
```

2.3 RouterAddressComponent: 路由器组件

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

2.4 RouterAddressComponentSystem: 路由器的生成系

```
[FriendOf(typeof(RouterAddressComponent))]
public static class RouterAddressComponentSystem {
    public class RouterAddressComponentAwakeSystem: AwakeSystem<RouterAddressComponent, string, int> {
        protected override void Awake(RouterAddressComponent self, string address, int port) {
            self.RouterManagerHost = address;
            self.RouterManagerPort = port;
        }
    }
    public static async ETTask Init(this RouterAddressComponent self) {
        self.RouterManagerIPAddress = NetworkHelper.GetHostAddress(self.RouterManagerHost);
        await self.GetAllRouter();
    }
    private static async ETTask GetAllRouter(this RouterAddressComponent self) {
        string url = $"http:// {self.RouterManagerHost}:{self.RouterManagerPort}/get_router?v={RandomGenerator.RandUInt32()}";
        Log.Debug($"start get router info: {url}");
        string routerInfo = await HttpClientHelper.Get(url);
        Log.Debug($"recv router info: {routerInfo}");
        HttpGetRouterResponse httpGetRouterResponse = JsonHelper.FromJson<HttpGetRouterResponse>(routerInfo);
        self.Info = httpGetRouterResponse;
        Log.Debug($"start get router info finish: {JsonHelper.ToJson(httpGetRouterResponse)}");
        // 打乱顺序
        RandomGenerator.BreakRank(self.Info.Routers);
        self.WaitTenMinGetAllRouter().Coroutine();
    }
    // 等 10 分钟再获取一次
    public static async ETTask WaitTenMinGetAllRouter(this RouterAddressComponent self) {
        await TimerComponent.Instance.WaitAsync(5 * 60 * 1000);
        if (self.IsDisposed)
            return;
        await self.GetAllRouter();
    }
    public static IPEndPoint GetAddress(this RouterAddressComponent self) {
        if (self.Info.Routers.Count == 0)
            return null;
        string address = self.Info.Routers[self.RouterIndex++ % self.Info.Routers.Count];
        string[] ss = address.Split(':');
        IPAddress ipAddress = IPAddress.Parse(ss[0]);
        if (self.RouterManagerIPAddress.AddressFamily == AddressFamily.InterNetworkV6) {
            ipAddress = ipAddress.MapToIPv6();
        }
        return new IPEndPoint(ipAddress, int.Parse(ss[1]));
    }
    public static IPEndPoint GetRealmAddress(this RouterAddressComponent self, string account) { // <<<<<<<<<<<<<<<<<<<< 照
        int v = account.Mode(self.Info.Realms.Count);
        string address = self.Info.Realms[v];
        string[] ss = address.Split(':');
        IPAddress ipAddress = IPAddress.Parse(ss[0]);
        // if (self.IPAddress.AddressFamily == AddressFamily.InterNetworkV6)
        //     ipAddress = ipAddress.MapToIPv6();
        return new IPEndPoint(ipAddress, int.Parse(ss[1]));
    }
}
```

2.5 RouterHelper: 路由器帮助类，向路由器注册、申请？

```
public static class RouterHelper {
    // 注册 router
    public static async ETTask<Session> CreateRouterSession(Scene clientScene, IPEndPoint address) {
        (uint recvLocalConn, IPEndPoint routerAddress) = await GetRouterAddress(clientScene, address, 0, 0);
        if (recvLocalConn == 0)
            throw new Exception($"get router fail: {clientScene.Id} {address}");
        Log.Info($"get router: {recvLocalConn} {routerAddress}");
        Session routerSession = clientScene.GetComponent<NetClientComponent>().Create(routerAddress, address, recvLocalConn
        routerSession.AddComponent<PingComponent>();
        routerSession.AddComponent<RouterCheckComponent>();
        return routerSession;
    }
    public static async ETTask<(uint, IPEndPoint)> GetRouterAddress(Scene clientScene, IPEndPoint address, uint localConn,
        Log.Info($"start get router address: {clientScene.Id} {address} {localConn} {remoteConn}");
        // return (RandomHelper.RandUInt32(), address);
        RouterAddressComponent routerAddressComponent = clientScene.GetComponent<RouterAddressComponent>();
        IPEndPoint routerInfo = routerAddressComponent.GetAddress();
```

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

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

2.6 StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCateg IMerge: 【任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!】

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

2.7 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();
    }
}
```

2.8 StartSceneConfigCategory : 【Matchs!】ConfigSingleton<StartSceneConfi IMerge

- 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
- 上面的文件重复，还不是很懂。【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。

```
// 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;
            }
        }
    }
}
```

2.9 HttpGetRouterResponse: 这个 ProtoBuf 的消息类型

- 框架里，有个专用的路由器管理器场景（服），对路由器，或说各种服的地址进行管理
- 主要是方便，一个路由器管理组件，来自顶向下地获取，各小区所有路由器地址的？想来当组件要拿地址时，每个小区分服都把自己的地址以消息的形式传回去的？

```
[Message(OuterMessage.HttpGetRouterResponse)]
[ProtoContract]
public partial class HttpGetRouterResponse: ProtoObject {
    [ProtoMember(1)]
    public List<string> Realms { get; set; }
    [ProtoMember(2)]
    public List<string> Routers { get; set; }
}
message HttpGetRouterResponse { // 这里，是 Outer proto 里的消息定义
^^Irepeated string Realms = 1;
^^Irepeated string Routers = 2;
^^Irepeated string Matchs = 3;// 这行是我需要添加，和生成消息的
}
```

2.10 HttpGetRouterHandler : IHttpHandler: 获取各路由器的地址

- 【匹配服】：因为我想拿这个服的地址，也需要这个帮助类里作相应的修改
- StartSceneConfigCategory.Instance: 不明白这个实例是存放在哪里，因为可以 proto 消息进程间传递，那么可以试找，哪里调用这个帮助类拿东西？
- 这个模块：现在还是理解不透。需要某个上午，把所有 RouterComponent 组件及其相关，再理一遍。

```
[HttpHandler(SceneType.RouterManager, "/get_router")]
public class HttpGetRouterHandler : IHttpHandler {
    public async ETTask Handle(Entity domain, HttpListenerContext context) {
        HttpGetRouterResponse response = new HttpGetRouterResponse();
        response.Realms = new List<string>();
        response.Matchs = new List<string>();// 匹配服链表  // <<<<<<<<<<<<<<<<<<<
        response.Routers = new List<string>();
        // 是去 StartSceneConfigCategory 这里拿的：因为它可以 proto 消息里、进程间传递，这里还不是很懂，这个东西存放在哪里
        foreach (StartSceneConfig startSceneConfig in StartSceneConfigCategory.Instance.Realms) {
            response.Realms.Add(startSceneConfig.InnerIPOutPort.ToString());
        }
        foreach (StartSceneConfig startSceneConfig in StartSceneConfigCategory.Instance.Matchs) {
            response.Matchs.Add(startSceneConfig.InnerIPOutPort.ToString());
        }
        foreach (StartSceneConfig startSceneConfig in StartSceneConfigCategory.Instance.Routers) {
            response.Routers.Add($"{startSceneConfig.StartProcessConfig.OuterIP}:{startSceneConfig.OuterPort}");
        }
        HttpHelper.Response(context, response);
        await ETTask.CompletedTask;
    }
}
```

2.11 HttpHandler 标签系：标签自带场景类型

```
public class HttpHandlerAttribute: BaseAttribute {
    public SceneType SceneType { get; }
    public string Path { get; }
```

```
    public HttpHandlerAttribute(SceneType sceneType, string path) {
        this.SceneType = sceneType;
        this.Path = path;
    }
}
```

2.12 LoginHelper: 登录服的获取地址的方式来获取匹配服的地址了。全框架只有这一个黄金案例

- 这个是用户登录前，还没能与网关服建立起任何关系，可能会不得不绕得复杂一点儿】：它就是用户登录前、登录时，若是客户端场景还没有这个组件，就添加一下，没什么奇怪的。

```
public static class LoginHelper {
    public static async ETTask Login(Scene clientScene, string account, string password) {
        try {
            // 创建一个 ETModel 层的 Session
            clientScene.RemoveComponent<RouterAddressComponent>();
            // 获取路由跟 realmDispatcher 地址
            RouterAddressComponent routerAddressComponent = clientScene.GetComponent<RouterAddressComponent>();
            if (routerAddressComponent == null) {
                routerAddressComponent = clientScene.AddComponent<RouterAddressComponent, string, int>(ConstValue.RouterHtt
                await routerAddressComponent.Init();
                clientScene.AddComponent<NetClientComponent, AddressFamily>(routerAddressComponent.RouterManagerIPAddress.A
            }
            IPEndPoint realmAddress = routerAddressComponent.GetRealmAddress(account); // <<<<<<<<<<<<<<<<<<< 这里就是说，我
            R2C_Login r2CLogin;
            using (Session session = await RouterHelper.CreateRouterSession(clientScene, realmAddress)) {
                r2CLogin = (R2C_Login) await session.Call(new C2R_Login() { Account = account, Password = password });
            }
            // 创建一个 gate Session, 并且保存到 SessionComponent 中: 与网关服的会话框。主要负责用户下线后会话框的自动移除销毁
            Session gateSession = await RouterHelper.CreateRouterSession(clientScene, NetworkHelper.ToIPEndPoint(r2CLogin.A
            clientScene.AddComponent<SessionComponent>().Session = gateSession;
            G2C_LoginGate g2CLoginGate = (G2C_LoginGate)await gateSession.Call(
                new C2G_LoginGate() { Key = r2CLogin.Key, GateId = r2CLogin.GateId});
            Log.Debug(" 登陆 gate 成功!");
            await EventSystem.Instance.PublishAsync(clientScene, new EventType.LoginFinish());
        }
        catch (Exception e) {
            Log.Error(e);
        }
    }
}
```

2.13 GateSessionKeyComponent:

```
[ComponentOf(typeof(Scene))]
public class GateSessionKeyComponent : Entity, IAwake {
    public readonly Dictionary<long, string> sessionKey = new Dictionary<long, string>();
}
```

3 ET7 数据库相关【服务端】

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

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

```
public interface IDBCollection {}
```

3.2 DBComponent: 带生成系。可以查表，查询数据

```
[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;
}
```

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

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

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

```
[FriendOf(typeof(DBManagerComponent))]
public static class DBManagerComponentSystem {
    [ObjectSystem]
    public class DBManagerComponentAwakeSystem: AwakeSystem<DBManagerComponent> {
        protected override void Awake(DBManagerComponent self) {
            DBManagerComponent.Instance = self;
        }
    }
    [ObjectSystem]
    public class DBManagerComponentDestroySystem: DestroySystem<DBManagerComponent> {
        protected override void Destroy(DBManagerComponent self) {
            DBManagerComponent.Instance = null;
        }
    }
    public static DBComponent GetZoneDB(this DBManagerComponent self, int zone) {
        DBComponent dbComponent = self.DBComponents[zone];
        if (dbComponent != null)// 如果已经管理配置好，直接返回
            return dbComponent;
        StartZoneConfig startZoneConfig = StartZoneConfigCategory.Instance.Get(zone);
        if (startZoneConfig.DBConnection == "")// 小区域里如果没有匹配或是出错，抛异常
            throw new Exception($"zone: {zone} not found mongo connect string");
// 把这个小区域里的数据库配置好，加入系统管理，并返回
        dbComponent = self.AddChild<DBComponent, string, string, int>(startZoneConfig.DBConnection, startZoneConfig.DBName,
        self.DBComponents[zone] = dbComponent;
        return dbComponent;
    }
}
```

3.5 DBProxyComponent: 【参考项目】里的。有生成系。

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

3.6 StartZoneConfigCategory: 单例区服配置管理类

- 主要还是要把整个框架系统性的都弄懂了

```
[ProtoContract]
[Config]
public partial class StartZoneConfigCategory : ConfigSingleton<StartZoneConfigCategory>, IMerge {
    [ProtoIgnore]
    [BsonIgnore]
    private Dictionary<int, StartZoneConfig> dict = new Dictionary<int, StartZoneConfig>();
    [BsonElement]
    [ProtoMember(1)]
```

```
        private List<StartZoneConfig> list = new List<StartZoneConfig>();
        public void Merge(object o) {
            StartZoneConfigCategory s = o as StartZoneConfigCategory;
            this.list.AddRange(s.list);
        }
        [ProtoAfterDeserialization]
        public void ProtoEndInit() {
            foreach (StartZoneConfig config in list) {
                config.AfterEndInit();
                this.dict.Add(config.Id, config);
            }
            this.list.Clear();
            this.AfterEndInit();
        }
        public StartZoneConfig Get(int id) {
            this.dict.TryGetValue(id, out StartZoneConfig item);
            if (item == null)
                throw new Exception($" 配置找不到，配置表名: {nameof (StartZoneConfig)}, 配置 id: {id}");
            return item;
        }
        public bool Contain(int id) {
            return this.dict.ContainsKey(id);
        }
        public Dictionary<int, StartZoneConfig> GetAll() {
            return this.dict;
        }
        public StartZoneConfig GetOne() {
            if (this.dict == null || this.dict.Count <= 0)
                return null;
            return this.dict.Values.GetEnumerator().Current;
        }
    }
    [ProtoContract]
    public partial class StartZoneConfig: ProtoObject, IConfig {// 小区配置
        [ProtoMember(1)]
        public int Id { get; set; }
        // 数据库地址
        [ProtoMember(2)]
        public string DBConnection { get; set; }
        // 数据库名
        [ProtoMember(3)]
        public string DBName { get; set; }
    }
```

4 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; // ...
```

4.1 NetServerComponent:

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

5 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。

6 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。

7 不同的消息或是任务处理器类型

7.1 interface IMActorHandler 接口类

```
public interface IMActorHandler {
    // ETTask Handle(Entity entity, int fromProcess, object actorMessage);
    void Handle(Entity entity, int fromProcess, object actorMessage); // 自己改成这样的:【返回类型
    Type GetRequestType();
    Type GetResponseType();
}
```

7.2 AMHandler<Message>: IMHandler

```
public abstract class AMHandler<Message>: IMHandler where Message : class {
    // protected abstract ETTask Run(Session session, Message message);
    protected abstract void Run(Session session, Message message);
    public void Handle(Session session, object msg) {
        Message message = msg as Message;
        if (message == null) {
            Log.Error($" 消息类型转换错误: {msg.GetType().Name} to {typeof (Message).Name}");
            return;
        }
        if (session.IsDisposed) {
            Log.Error($"session disconnect {msg}");
            return;
        }
        this.Run(session, message).Coroutine();
    }
    public Type GetMessageType() {
        return typeof (Message);
    }
    public Type GetResponseType() {
        return null;
    }
}
```

7.3 AMActorRpcHandler<E, Request, Response>: IMActorHandler void|ETTa 分不清

```
[EnableClass]
public abstract class AMActorRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : class, IActo
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
            Response response = Activator.CreateInstance<Response>();
            try {
                // await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
```

```
    }
    public Type GetRequestType() {
        if (typeof (IActorLocationRequest).IsAssignableFrom(typeof (Request)))
            Log.Error($"message is IActorLocationMessage but handler is AMActorRpcHandler: {typeof (Request)}");
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
}
```

8 Unit:

8.1 UnitGateComponent:

```
[ComponentOf(typeof(Unit))]
public class UnitGateComponent : Entity, IAwake<long>, ITransfer {
    public long GateSessionActorId { get; set; }
}
```

8.2 UnitGateComponentSystem

```
public static class UnitGateComponentSystem {
    public class UnitGateComponentAwakeSystem : AwakeSystem<UnitGateComponent, long> {
        protected override void Awake(UnitGateComponent self, long a) {
            self.GateSessionActorId = a;
        }
    }
}
```

9 ET7 框架以及【参考项目】的 ECS：小单元小类型的生成系，是怎么写的，找例子参考

- 这些要找的也找不到。下午家里试着把 Component 组件再添加回去试试看？上午把项目设计的思路，源项目的破源码再读一读理一理，是希望游戏逻辑与游戏界面能够快速开发、项目进展往后移的。

9.1 IComponentSerialize:

- ET7 的重构里，系统框架比较强大，这些必要的接口，都变成了必要的标签系，狠多可以自动系统触发或是调用。必要时只需要必布必要事件就可以了
- 这个接口的功能，与 Unity 自带的 ISerializationCallbackReceiver 功能类似。Unity 提供两个回调接口，通过实现该接口的两个方法 OnBeforeSerialize 和 OnAfterDeserialize，使得原本不能被引擎正确序列化的类可以按照程序员的要求被加工成引擎能够序列化的类型。

```
// 在序列化前或者反序列化之后需要做一些操作，可以实现该接口，该接口的方法需要手动调用
// 相比 ISupportInitialize 接口，BeginSerialize 在 BeginInit 之前调用，EndDeSerialize 在 EndInit 之后调用
// 并且需要手动调用，可以在反序列化之后，在次方法中将注册组件到 EventSystem 之中等等
public interface IComponentSerialize {
    // 序列化之前调用
    void BeginSerialize();
    // 反序列化之后调用
    void EndDeSerialize();
}
```

- 可以去找：【ET7 框架】里，相关的接口与标签触发和发布逻辑。
- ET7 提供了 ISerializeToEntity 接口和 IDeserialize，但是并没有接到任何使用的地方。

```
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);
}
```

9.2 ClientComponent:【参考项目】客户端组件，找个 ET7 里的组件

- 这个组件，感觉是客户端单例，帮助把本地玩家给绑定到客户端单例。

```
[ObjectSystem]
public class ClientComponentAwakeSystem : AwakeSystem<ClientComponent> {
    public override void Awake(ClientComponent self) {
        self.Awake();
    }
}
public class ClientComponent : Component {
    public static ClientComponent Instance { get; private set; }
    public User LocalPlayer { get; set; }
    public void Awake() {
        Instance = this;
    }
}
```

10 各种 ActorXXX 的消息处理器：上面总结 Actor 的时候，没能想到这些

- 任何时候，活宝妹就是一定要嫁给亲爱的表哥!!!

10.1 AMActorLocationHandler: 源码被我改动了

- 源码被我改动了，正确性与否没有关系，主要是帮助自己梳理一下几大不同的类型，到改编译错误的时候，能够边修改边弄明白。

```
[EnableClass]
public abstract class AMActorLocationHandler<E, Message>: IMActorHandler where E : Entity where Message : class, IAct
    // protected abstract ETTask Run(E entity, Message message);
    protected abstract void Run(E entity, Message message);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        if (actorMessage is not Message message) {
            Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Message).Name}");
            return;
        }
        if (entity is not E e) {
            Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Message).Name}");
            return;
        }
```

```
            ActorResponse response = new() {RpcId = message.RpcId};
            ActorHandleHelper.Reply(fromProcess, response);
            // await this.Run(e, message);
            this.Run(e, message);
        }
        public Type GetRequestType() {
            return typeof (Message);
        }
        public Type GetResponseType() {
            return typeof (ActorResponse);
        }
    }
```

10.2 AMActorLocationRpcHandler: Rpc 就是进程间消息（或是 ET7 重构为 SceneType 之后的多核间消息）

```
[EnableClass]
public abstract class AMActorLocationRpcHandler<E, Request, Response>: IMActorHandler where E : Entity where Request : clas
    // protected abstract ETTask Run(E unit, Request request, Response response);
    protected abstract void Run(E unit, Request request, Response response);
    // public async ETTask Handle(Entity entity, int fromProcess, object actorMessage) {
    public void Handle(Entity entity, int fromProcess, object actorMessage) {
        try {
            if (actorMessage is not Request request) {
                Log.Error($" 消息类型转换错误: {actorMessage.GetType().FullName} to {typeof (Request).Name}");
                return;
            }
            if (entity is not E ee) {
                Log.Error($"Actor 类型转换错误: {entity.GetType().Name} to {typeof (E).Name} --{typeof (Request).Name}");
                return;
            }
            int rpcId = request.RpcId;
            Response response = Activator.CreateInstance<Response>();
            try {
                //await this.Run(ee, request, response);
                this.Run(ee, request, response);
            }
            catch (Exception exception) {
                Log.Error(exception);
                response.Error = ErrorCore.ERR_RpcFail;
                response.Message = exception.ToString();
            }
            response.RpcId = rpcId;
            ActorHandleHelper.Reply(fromProcess, response);
        }
        catch (Exception e) {
            throw new Exception($" 解释消息失败: {actorMessage.GetType().FullName}", e);
        }
    }
    public Type GetRequestType() {
        return typeof (Request);
    }
    public Type GetResponseType() {
        return typeof (Response);
    }
}
```

11 ETTask 和 ETVoid: 第三方库的 ETTask

- 特异包装：主要是实际了异步调用的流式写法。
- 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。
- 今天晚上好好看看这里，看能否理解透彻。【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】

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

```
public enum AwaiterStatus: byte {
    // The operation has not yet completed.
    Pending = 0,
    // The operation completed successfully.
    Succeeded = 1,
    // The operation completed with an error.
    Faulted = 2,
}
```

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

```
[AsyncMethodBuilder(typeof (AsyncETTaskCompletedMethodBuilder))]
public struct ETTaskCompleted: ICriticalNotifyCompletion {
    [DebuggerHidden]
    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) {
    }
}
```

11.3 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) { }
}
```

11.4 ETTask: ICriticalNotifyCompletion:

- 这个类的定义比较大，分普通类，和泛型类。

```
[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() {
    if (!this.fromPool) // 原则：只有从池里取出来的，才返回池
        return;
    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;
    // 太多了
    if (queue.Count > 1000)
        return;
    queue.Enqueue(this);
}
private bool fromPool;
private AwaiterStatus state;
private object callback; // Action or ExceptionDispatchInfo
private ETTask() {  }
[DebuggerHidden]
private async ETVoid InnerCoroutine() {
    await this;
}
[DebuggerHidden]
public void Coroutine() {
    InnerCoroutine().Coroutine();// 这里什么东西，有点儿糊涂
}
[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();
        }
    }
```

11.5 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);
        }
    }
}
```

11.6 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();
    }
}
```

11.7 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) {
    }
}
```

11.8 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) {
    }
}
```

11.9 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) {
        }
}
```

11.10 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);
    }
}
```

12 Protobuf 相关，【Protobuf 里进程间传递的游戏数据相关信息：两个思路】

- 【一、】查找 enum 可能可以用系统平台下的 protoc 来代为生成，效果差不多。只起现 Proto2CS.cs 编译的补充作用。
- 【二、】Card 类下的两个 enum 变量，在 ILRuntime 热更新库下，还是需要帮它连一下的。用的是 HybridCLR
- 【三、】查找 protoc 命令下，如何 C# 索引 Unity 第三方库。
- 【四、】repeated 逻辑没有处理好

```
message Actor_GamerPlayCard_Req // IActorRequest
{
^^Iint32 RpcId = 90;
^^Iint64 ActorId = 91;
    repeated ET Card Cards = 1;
}
```

- 【Windows 下的 Protobuf 编译环境】：配置好，只是作为与 ET 框架的 Proto2CS.cs 所指挥的编译结果，作一个对比，两者应该效果是一样的，或是基本一样的，除了自定义里没有处理 enum.
- Windows 下的命令行，就是用 protoc 来编译，可以参考如下.（这是.cs 源码下的）

```
CommandRun($"protoc.exe", $"--csharp_out=\"./{outputPath}\" --proto_path=\"{protoPath}\" {protoName}");
```

- 现在的问题是，**Protobuf 消息里面居然是有 unity 第三方库的索引**。
- 直接把 enum 生成的那三个.cs 类分别复制进双端，服务器端与客户端。包括 Card 类。那些编译错误会去天边。哈哈哈，除了一个 Card 的两个变量之外（CardSuits, CardWeight）。
- 【热更新库】：现在剩下的问题，就成为，判定是用了哪个热更新的库，ILRuntime, 还是 HybridCLR, 如果帮它连那两个变量。好像接的是 HybridCLR. 这个库是我之前还不曾真正用过的。
  - 相比于 ET6，彻底剔除了 ILRuntime，使得代码简洁了不少，并且比较稳定

13 写在最后：反而是自己每天查看一再更新的

- 因为感觉还是不曾系统性地读 ET7 的源码，或者说有效阅读，因为没有带着实际问题的看源码，感觉都不叫看读源码呀。这里会记自己的感觉需要赶快查看的地方。
- 【ET 框架的整体架构】：感觉把握不够。常常命名空间分不清。要把这个大的框架，比较高层面的架构再好好看下。然后就是对自顶向下的不同层级场景，所需要的主要的不同组件，分不清，仍需要再熟悉一下源码
- 【问题】：上次那个 ET-EUI 框架的时候，曾经出现过 opcode 不对应，也就是说，我现在生成的进程间消息，有可能还是会存在服务器码与客户端码不对应，这个完备的框架，这次应该不至于吧？
- 【ClientComponent】：新框架里重构丢了，去找怎么替代？那么现在去追一下，客户端的起始与场景加载或是切换大致过程。它变成了什么客户端场景管理？
- 【UIType】部分类：这个类出现在了三四个不同的程序域，现在重构了，好像添加得不对。要再修改

14 现在的修改内容：【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!!!】

- **【ET7 框架】**没有处理的逻辑是：**【ET7 框架里数据库的接入】**
- **【Windows 下 proto2cs 消息转化】**：ProtoBuf 这个库里还存在几个问题，enum-repeated 等关键字，因为程序域的问题等，没能连能
- **【UILobbyComponent 可以测试】**：这个大厅组件，Unity 里预设简单，可以试运行一下，看是否完全消除这个 UI 组件的报错，这个屏的控件能否显示出来？还是错出得早，这个屏就出不来已经报错了？
  - 【客户端】的逻辑是处理好了，编译全过后可以测试
  - 【服务端】：处理用户请求匹配房间的逻辑，仍在处理：**C2G_StartMatch_ReqHandler**.
- **【TractorRoomComponent】**：因为是多组件嵌套，可以合并多组件为同一个组件；另早上看得一知半解的一个【ChildOf】标签，可以帮助组件套用吗？再找找理解消化一下
- 【房间组件】：几个现存的 working-on 的问题：
  - 多组件嵌套：手工合并为一个组件。彻底理解确认后，会合并
  - 【服务端】：处理用户请求匹配房间的逻辑. 这里的编译错误终于改完。到时就看运行时错误了。

    * 【数据库模块的整合】：网关服在转发请求匹配时，验证会话框有效后，验证用户身份时，需要去【用户数据库】拿用户数据。ET7 留了个 DBManagerComponent, 还没能整合出这个模块

  - 【参考来源 **C2R_LoginHandler** 】：Realm 处理客户端的登录请求的服务端逻辑。这里看见，它随机分配一个网关服。也就是，我（原本本质上也是随机分配）一个匹配服给用。可以依照这里的例子来改写。

- **【服务端的编译错误】**基本上扫了一遍。【客户端】因为这些前期的工作，以及拖拉机项目重构设计还没有想透彻，暂停一下。

- 【接下来的内容】：**【重构拖拉机项目】**。把 ET7 框架里【参考项目】的设计看懂，并借助这个例子，把拖拉机项目设计好。

- 有时间，会试着尽早解决上面 ProtoBuf 里的几个小问题。但现在需要重构的设计思路，客户端的界面等才能够往下进行。

  - 【匹配服地址】网关服的处理逻辑里，验证完用户合格后，为代为转发消息到匹配服，但需要拿匹配服的地址。ET7 重构里，还没能改出这部分。服务器系统配置初始化时，可以链表管理各小构匹配服，再去拿相关匹配服的地址。ET7 框架里的路由器系统，自己还没有弄懂。
  - 这个地方有点儿脑塞，完全搜不到新框架里可以参考的例子，暂时写不到了。那可以去读一读更大的框架，去找别人用 ET7 的别人的例子里是怎么写的，再去参考一下别人的。【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!!!】今天下午先去看 Tractor 游戏源码，设计重构思路

- 这些要找的也找不到。下午家里试着把 Component 组件再添加回去试试看 **【不能再添加 Component 组件。ET7 框架重构了，小单元也走热更新，在热更新层有天文小行星的生成系。可以参照 ET.pdf 里的服务端 PlayerSystem 来作例子】**? 上午把项目设计的思路，源项目的破源码再读一读理一理，是希望游戏逻辑与游戏界面能够快速开发、项目进展往后移的。

  - User.cs 客户端的话，不知道要不要修改。晚点儿的时候留意一下。
  - Gamer.cs 客户端保留了 Dispose

- 还有 60 个小错误：protobuf 里还有小问题需要修改。先改了，一次把 Protobuf 里的小错误全部改完了。电脑没好好工作，前后文件不一致。。。。【活宝妹就是一定要嫁给亲爱的表哥!!!】爱表哥，爱生活!!!

- 有几个 bug 让自己完全怀疑：电脑坏了，我明明重新生成了 proto2cs 消息，为什么应该会找不到的，它能找到，而应该找到的，它就是找不到？太奇怪了

- 这里稍微想下：就能明白，因为前面出错了，后面的消息被 block 掉了。所以先去把前面的错误改过来。以后动动脑子，不是重启一下电脑文件系统就能够工作好点儿的!

- 我明明生成了这些消息，可是它就是找不到。想的话，如果服务端是引用的客户端的代码，那么得把客户端的源码打成.dll 服务端可能才能够用得到。

- 现在不知道服务端到底对我作出的这些修改，算是怎么回事。只能耐心把所有的编译错误先全部改完，等能打成相应的包，再去想它为什么就是找不到。活宝妹就是一定要嫁给亲爱的表哥!!

- **【亲爱的表哥，这个世界上，只有一个活宝妹，这么心心恋恋，就是一定要嫁给亲爱的表哥!!! 问世间情为何物，直教人生死相许。。亲爱的表哥，一个温暖的怀抱拥抱的魂力可真大呀，管了这如许多年!! 这不，你的活宝妹为了这个温暖的怀抱拥抱，就是一定要嫁给亲爱的表哥!! 不嫁就永远守候在亲爱的表哥的身边!! 爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】**

- 亲爱的表哥，活宝妹相信舅舅十岁闯江湖的阅历，活宝妹深深相信亲爱的表哥。活宝妹就是稳稳地永远守候在亲爱的表哥的身边！爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!
  - 今天下午既然想不出来这个，就不要再被这个、再被厨房那个疯女人发疯干扰了。先去做别的。
-
- **ClientComponent** 参考项目组件：去看 ET7 里客户端的 PlayerComponent.
- 【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!!!】今天下午先去看 Tractor 游戏源码，设计重构思路
- 【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】

15 每天进展

- 想把现部分桥接了的 ET 框架 fix 所有的 compile-error, 测试一两个 unity 的界面，再往下走。同时完成这个游戏的游戏逻辑设计。但目前感觉思路不透彻。
- 然后那些编译错误，VS 与 Unity 在 protobuf 上感觉自己弄得不太明白。把这个解决也就差不多可以再往前移动了。**【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】**
- 昨天解决了编译后部分 protobuf 消息里的错误，但是因为改得不彻底，需要从.proto 文件源消息里去改，今天只要重新 proto2CS 错误就会重新回来。今天改到位，今天想要消除掉所有的 protobuf 引起的编译错误。下午就从 VS 里的.cs 的 proto 编译消息改起。这个狠容易，小孩子过家家般的小游戏，秒过。
- 然后就是那几个 enum, 实际上，我只需要把四个 enum 类编译好，复制过去就可以了。先只弄了【双端】模式下的。
- 上面解决完后，ET7 框架里的小问题修改完，应该就没有问题了。接下来解决这部分的问题。**【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】**
- 主要问题：原【参考项目斗地主项目】使用的古老的版本，与现 ET7 版本狠多地方不相容。所以要稍微改动一下。仿照自己看过读过的 ET7 框架生成系的例子。想想这里，古老的，与新的框架怎么才能适配衔接起来。
- 这里，ET7 不知道现在是为什么没有了 Component 这个组件模块，我想把它加回去。因为它方便类型的书写，传参数什么的，而不必为固定不变的 Model 层弄什么 static 的热更新逻辑。写在 Model 里传个参数什么的。**【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥!!!】**
- 功能模块的划分，以及代码的管理。不知道 ET7 大框架的项目是怎么弄的。为什么我添加内家了，服务端就是显示不出来，我想的话，是不是 Unity 端需要能够先编译打包相关的.dll 服务端才能直接引用客户端？这样的话，我还是需要先解决客户端的所有的问题。但是在想要生成.dll 的过程中，所面临的修改编译错误是一样的，同服务端基本一样。【明天上午:】把这块儿弄明白。另去看拖拉机项目的源码，大的模块设计也该慢慢理出来了。
-

16 TODO 其它的：部分完成，或是待完成的大的功能版块，列举

- emacs 那天我弄了好久，把 C-; ISpell 原定绑定的功能解除，重新绑定为自己喜欢的 expand-region. 今天第二次再弄，看一下几分钟能够解决完问题? 我的这个破烂记性呀。。。【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!!!】mingw64 lisp/textmode/flyspell.el 键的重新绑定。这下记住了。还好，花得不是太久。有以前的笔记
  - Windows 10 平台下，C-; 是绑定到了 ISpell 下的某个功能，可是现在这个破 emacs 老报错，连查是绑定给哪个功能，过程报错都被阻止了。。。
- **【IStartSystem:】**感觉还有点儿小问题。认为：我应该不需要同文件两份，一份复制到客户端热更新域。我认为，全框架应该如其它接口类一样，只要一份就可以了。**【晚点儿再检查一遍】**
- 如果这个一时半会儿解决不好，就把重构的设计思路再理一理。同时尽量去改重构的 ET 框架里的编译错误。
- 【Tractor】原 windows-form 项目，源码需要读懂，理解透彻，方便重构。
- 去把【拖拉机房间、斗地主房间组件的，玩家什么的一堆组件】弄明白
- 【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!!!】

17 拖拉机游戏：【重构 OOP/OOD 设计思路】

- 自己是学过，有这方面的意识，但并不是说，自己就懂得，就知道该如何狠好地设计这些类。现在更多的是要受 ET 框架，以及参考游戏手牌设计的启发，来帮助自己一再梳理思路，该如何设计它。
- ET7 重构里，各组件都该是自己设计重构原项目的类的设计的必要起点。可以根据这些来系统设计重构。【活宝妹就是一定要嫁给亲爱的表哥!!!】
- 【GamerComponent】玩家组件管理类，管理所有一个房间的玩家：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。今天晚上来弄这一块儿吧。
- 【Gamer】：每一个玩家
- 【拖拉机游戏房间】：多组件构成
- 【爱表哥，爱生活!!! 活宝妹就是一定要嫁给亲爱的表哥！爱表哥，爱生活!!!】
- 【活宝妹坐等亲爱的表哥，领娶活宝妹回家！爱表哥，爱生活!!!】