

ET 框架学习笔记（二） - - 网络交互相关

deepwaterooo

May 18, 2023

Contents

1 Actor 消息相关	1
1.1 ActorMessageSender: 知道对方的 instanceId, 使用这个类发 actor 消息	1
1.2 ActorMessageSenderComponent:	1
1.3 ActorMessageSenderComponentSystem: 这个类, 今天晚上没有看懂, 明天上午再看一下	1
1.4 LocationProxyComponent	3
1.5 LocationProxyComponentSystem	3
1.6 ActorLocationSender: 知道对方的 Id, 使用这个类发 actor 消息	3
1.7 ActorLocationSenderComponent: 位置发送组件	4
1.8 ActorLocationSenderComponentSystem: 这个类, 也要明天上午再看一下	4
1.9 ActorHelper: 帮助创建 IActorResponse 回复消息。很简单	5
1.10ActorMessageDispatcherInfo ActorMessageDispatcherComponent	5
1.11ActorMessageDispatcherComponentHelper: 感觉名字不系统化, 不知道是不是自己干的	6
1.12ActorMessageHandlerAttribute 标签系: 去找几个典型标签看看	6
1.13[ActorMessageHandler(SceneType.Gate)] 标签使用举例	7
1.14MailBoxComponent: 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	7
1.15MailboxType	7
1.16【服务端】ActorHandleHelper 帮助类。【需要去深挖一下】	7
2 StartConfigComponent: 找【各种服】的起始初始化地址	8
2.1 ConfigSingleton<T>: ProtoObject, ISingleton	9
2.2 SceneFactory 里可以给【匹配服】添加组件	9
2.3 RouterAddressComponent: 路由器组件	10
2.4 RouterAddressComponentSystem: 路由器的生成系	10
2.5 RouterHelper: 路由器帮助类, 向路由器注册、申请?	10
2.6 StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge: 【任何时候, 活宝妹就是一定要嫁给亲爱的表哥!!!】	11
2.7 StartSceneConfig: ISupportInitialize 【各种服 - 配置, 场景配置】	12
2.8 StartSceneConfigCategory : 【Matches!】ConfigSingleton<StartSceneConfigCategory>, IMerge	13
2.9 HttpGetRouterResponse: 这个 ProtoBuf 的消息类型	14
2.10HttpGetRouterHandler : IHttpHandler: 获取各路由器的地址	14
2.11HttpHandler 标签系: 标签自带场景类型	15
2.12LoginHelper: 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	15
2.13GateSessionKeyComponent:	15

3 ET7 数据库相关【服务端】	15
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	16
3.2 DBComponent: 带生成系。可以查表，查询数据	16
3.3 DBManagerComponent: 有上面的 DBComponent 数组。数组长度固定吗?	16
3.4 DBManagerComponentSystem: 主要是要查询某个区服的数据库，从数组里	16
3.5 DBProxyComponent: 【参考项目】里的。有生成系。	16
4 网关服：客户端信息发送的直接代理，中转站，组件分析	17
4.1 NetServerComponent:	17
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）	17
6 Session 会话框相关	18
7 Protobuf 相关，【Protobuf 里进程间传递的游戏数据相关信息：两个思路】	18
8 写在最后：反而是自己每天查看一再更新的	19
9 现在的修改内容：	19
10TODO 其它的：部分完成，或是待完成的大的功能版块，列举	20
11拖拉机游戏：【重构 OOP/OOD 设计思路】	20

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, ActorMessageSender>();
    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($"Rpc error: actorId: {self.ActorId} request: {self.Request}, response: {response}"));
            return;
        }
        self.Tcs.SetResult(response);
    }
    private static void Check(this ActorMessageSenderComponent self) {
        long timeNow = TimeHelper.ServerNow();
        foreach ((int key, ActorMessageSender value) in self.requestCallback) {
            // 因为是顺序发送的, 所以, 检测到第一个超时的就退出
            if (timeNow < value.CreateTime + ActorMessageSenderComponent.TIMEOUT_TIME)
                break;
            self.TimeoutActorMessageSenders.Add(key);
        }
        foreach (int rpcId in self.TimeoutActorMessageSenders) {
            ActorMessageSender actorMessageSender = self.requestCallback[rpcId];
            self.requestCallback.Remove(rpcId);
            try {
                IActorResponse response = ActorHelper.CreateResponse(actorMessageSender.Request, ErrorCore.ERR_ActorTimeout);
                Run(actorMessageSender, response);
            }
            catch (Exception e) {
                Log.Error(e.ToString());
            }
        }
        self.TimeoutActorMessageSenders.Clear();
    }
    public static void Send(this ActorMessageSenderComponent self, long actorId, IMessage message) {
        if (actorId == 0) {
            throw new Exception($"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 ETask<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 ETask<IActorResponse> Call(
        this ActorMessageSenderComponent self,
        long actorId,
        int rpcId,
        IActorRequest iActorRequest,

```

```

    bool needException = true
    ) {
    if (actorId == 0) {
        throw new Exception($"actor id is 0: {iActorRequest}");
    }
    var tcs = ETask<IActorResponse>.Create(true);
    self.requestCallback.Add(rpcId, new ActorMessageSender(actorId, iActorRequest, tcs, needException));
    self.Send(actorId, iActorRequest);
    long beginTime = TimeHelper.ServerFrameTime();
    IActorResponse response = await tcs;
    long endTime = TimeHelper.ServerFrameTime();
    long costTime = endTime - beginTime;
    if (costTime > 200) {
        Log.Warning($"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 ETask Add(this LocationProxyComponent self, long key, long instanceId) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                new ObjectAddRequest() { Key = key, InstanceId = instanceId });
    }
    public static async ETask 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 ETask 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 ETask Remove(this LocationProxyComponent self, long key) {
        await ActorMessageSenderComponent.Instance
            .Call(GetLocationSceneId(key),
                new ObjectRemoveRequest() { Key = key });
    }
    public static async ETask<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 ETask AddLocation(this Entity self) {

```

```

        await LocationProxyComponent.Instance.Add(self.Id, self.InstanceId);
    }
    public static async ETask 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 ETask<IActorResponse> Call(this ActorLocationSenderComponent self, long entityId, IActorRequest iActorRequest) {
    ActorLocationSender actorLocationSender = self.GetOrCreate(entityId);
    // 先序列化好
    int rpcId = ActorMessageSenderComponent.Instance.GetRpcId();
    iActorRequest.RpcId = rpcId;
    long actorLocationSenderId = actorLocationSender.InstanceId;
    using (await CoroutineLockComponent.Instance.Wait(CoroutineLockType.ActorLocationSender, entityId)) {
        if (actorLocationSender.InstanceId != actorLocationSenderId) {
            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 ETask<IActorResponse> CallInner(this ActorLocationSenderComponent self, ActorLocationSender actorLocationSender) {
    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, iActorRequest);
        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。很简单

```
public static class ActorHelper {
    public static IActorResponse CreateResponse(IActorRequest iActorRequest, int error) {
        Type responseType = OpcodeTypeComponent.Instance.GetResponseTypes(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 IActorHandler IActorHandler { get; }
    public ActorMessageDispatcherInfo(SceneType sceneType, IActorHandler imActorHandler) {
        this.SceneType = sceneType;
        this.IActorHandler = 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);
            IActorHandler imHandler = obj as IActorHandler;
            if (imHandler == null) {
                throw new Exception($"message handler not inherit IActorHandler 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.GetResponseTypes();
                if (handleResponseType != null) {
                    Type responseType = OpcodeTypeComponent.Instance.GetResponseTypes(messageType);
                    if (handleResponseType != responseType) {
                        throw new Exception($"message handler response type error: {messageType.FullName}");
                    }
                }
                ActorMessageDispatcherInfo actorMessageDispatcherInfo = new(actorMessageHandlerAttribute.SceneType, imHandler);
                self.RegisterHandler(messageType, actorMessageDispatcherInfo);
            }
        }

        private static void RegisterHandler(this ActorMessageDispatcherComponent self, Type type, ActorMessageDispatcherInfo handler) {
            if (!self.ActorMessageHandlers.ContainsKey(type))
                self.ActorMessageHandlers.Add(type, new List<ActorMessageDispatcherInfo>());
            self.ActorMessageHandlers[type].Add(handler);
        }

        public static async ETask Handle(this ActorMessageDispatcherComponent self, Entity entity, int fromProcess, object message) {
            List<ActorMessageDispatcherInfo> list;
            if (!self.ActorMessageHandlers.TryGetValue(message.GetType(), out list))
                return;
            foreach (ActorMessageDispatcherInfo info in list) {
                info.IActorHandler.Handle(entity, fromProcess, message);
            }
        }
    }
}
```

```

        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_MatchSuccess_NttHandler : AMActorHandler<User, Actor_MatchSuccess_Ntt> {
    protected override void Run(User user, Actor_MatchSuccess_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

- 这些组群服务器的起始被全部重构了，重构成配置单例了
- 这里是，昨天之前理这部分的时候，理得不够透彻。去重构一个服的初始化与配置，就需要理清、找到这个服所关联的所有的起始配置相关。昨天之前做得还不够好。
- 昨天晚上稍微理解得再深一点儿。应该也是，总结自己理一个相对大型框架的方法总结吧。
- 今天早上的脑袋还是相对比较清醒，就把这块儿对自己来说相对不够熟悉相对难点儿的，再看一遍，希望有更多的收获！

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, SceneType sceneType) {
        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.StartPort);
                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;
        }
    }
}

```



```

        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 ETask<Session> CreateRouterSession(Scene clientScene, IPEndPoint address) {
        (uint rcvLocalConn, IPEndPoint routerAddress) = await GetRouterAddress(clientScene, address, 0, 0);
        if (rcvLocalConn == 0)
            throw new Exception($"get router fail: {clientScene.Id} {address}");
        Log.Info($"get router: {rcvLocalConn} {routerAddress}");
        Session routerSession = clientScene.GetComponent<NetClientComponent>().Create(routerAddress, address, rcvLocalConn);
        routerSession.AddComponent<PingComponent>();
        routerSession.AddComponent<RouterCheckComponent>();
        return routerSession;
    }
    public static async ETask<(uint, IPEndPoint)> GetRouterAddress(Scene clientScene, IPEndPoint address, uint localConn,
        Log.Info($"start get router address: {clientScene.Id} {address} {localConn} {remoteConn}");
        // return (RandomHelper.RandUInt32(), address);
        RouterAddressComponent routerAddressComponent = clientScene.GetComponent<RouterAddressComponent>();
        IPEndPoint routerInfo = routerAddressComponent.GetAddress();
        uint rcvLocalConn = await Connect(routerInfo, address, localConn, remoteConn);
        Log.Info($"finish get router address: {clientScene.Id} {address} {localConn} {remoteConn} {rcvLocalConn} {routerInfo}");
        return (rcvLocalConn, routerInfo);
    }
    // 向 router 申请
    private static async ETask<uint> Connect(IPEndPoint routerAddress, IPEndPoint realAddress, uint localConn, uint remoteConn) {
        uint connectId = RandomGenerator.RandUInt32();
        using Socket socket = new Socket(routerAddress.AddressFamily, SocketType.Dgram, ProtocolType.Udp);
        int count = 20;
        byte[] sendCache = new byte[512];
        byte[] rcvCache = 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 rcvIPEndPoint = new IPEndPoint(IPAddress.Any, 0);
        long lastSendTimer = 0;
        while (true) {
            long timeNow = TimeHelper.ClientFrameTime();
            if (timeNow - lastSendTimer > 300) {
                if (--count < 0) {
                    Log.Error($"router connect timeout fail! {localConn} {remoteConn} {routerAddress} {realAddress}");
                    return 0;
                }
                lastSendTimer = timeNow;
                // 发送
                socket.SendTo(sendCache, 0, addressBytes.Length + 13, SocketFlags.None, routerAddress);
            }
            await TimerComponent.Instance.WaitFrameAsync();
            // 接收
            if (socket.Available > 0) {
                int messageLength = socket.ReceiveFrom(rcvCache, ref rcvIPEndPoint);
                if (messageLength != 9) {
                    Log.Error($"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} {routerAddress}");
    return recvLocalConn;
}
}
}
}
}

```

2.6 StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, 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; }
}

```


[illegible]

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
    ^^Repeated string Realms = 1;
    ^^Repeated string Routers = 2;
    ^^Repeated string Matches = 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 ETask Handle(Entity domain, HttpListenerContext context) {
        HttpGetRouterResponse response = new HttpGetRouterResponse();
        response.Realms = new List<string>();
        response.Matches = 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.Matches) {
            response.Matches.Add(startSceneConfig.InnerIPOutPort.ToString());
        }
        foreach (StartSceneConfig startSceneConfig in StartSceneConfigCategory.Instance.Routers) {
            response.Routers.Add($"{startSceneConfig.StartProcessConfig.OuterIP}:{startSceneConfig.OuterPort}");
        }
        HttpHelper.Response(context, response);
        await ETask.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 ETask 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.RouterHttp,
                await routerAddressComponent.Init();
                clientScene.AddComponent<NetClientComponent, AddressFamily>(routerAddressComponent.RouterManagerIPAddress,
            }
            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。。
- 这里不急着重整理。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)
        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;
}
}
}

```

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;
}

```

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, Scene scene) {
        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分发

        switch (scene.SceneType) {
            case SceneType.Router:
                scene.AddComponent<RouterComponent, IPEndPoint, string>(startSceneConfig.OuterIPPort, startSceneConfig.StartSceneType);
                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.InnerIP0OutPort);
            // // break;
            case SceneType.Gate:
                scene.AddComponent<NetServerComponent, IPEndPoint>(startSceneConfig.InnerIP0OutPort);
                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 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
{
  ^^Int32 RpcId = 90;
  ^^Int64 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，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

8 写在最后：反而是自己每天查看一再更新的

- 因为感觉还是不曾系统性地读 ET7 的源码，或者说有效阅读，因为没有带着实际问题的看源码，感觉都不叫看读源码呀。这里会记自己的感觉需要赶快查看的地方。
- **【ET 框架的整体架构】**：感觉把握不够。常常命名空间分不清。要把这个大的框架，比较高层面的架构再好好看下。然后就是对自顶向下的不同层级场景，所需要的主要的不同组件，分不清，仍需要再熟悉一下源码
- **【问题】**：某些消息，还分不清是内网还是外网消息，暂时先放一下，到时再改
- **【问题】**：上次那个 ET-EUI 框架的时候，曾经出现过 opcode 不对应，也就是说，我现在生成的进程间消息，有可能还是会存在服务器码与客户端码不对应，这个完备的框架，这次应该不至于吧？
- **【ClientComponent】**：新框架里重构丢了，去找怎么替代？那么现在去追一下，客户端的起始与场景加载或是切换大致过程。它变成了什么客户端场景管理？
- **【UIType】** 部分类：这个类出现在了三个不同的程序域，现在重构了，好像添加得不对。要再修改

9 现在的修改内容：

- **【Windows 下 proto2cs 消息转化】**：上面拿匹配服地址，一个路由器相关消息加了参数，要重新生成一下。这个 mac 系统下不难，改天有时间也可以把 mac 下的运行环境配置好，这个功能模块就不用一定要切换电脑了。**【手动做了一下】**
- **【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**. 必须记笔记。不记会忘。
- **【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。

10 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!!!】**

11 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!!!】**