

ET 框架学习笔记（三） - - 网络交互相关

deepwaterooo

June 15, 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: 这个组件里有个计时器自动计时的超时时段、特定超时类型的超时时长成员变量	1
1.3 ActorMessageSenderComponentSystem: 这个类 ETTask 的细节, 要再多看几遍	2
1.4 LocationProxyComponent: 这个代理, 什么情况下会用到?	4
1.5 LocationProxyComponentSystem	4
1.6 ActorLocationSender: 知道对方的 Id, 使用这个类发 actor 消息	5
1.7 ActorLocationSenderComponent: 位置发送组件	5
1.8 ActorLocationSenderComponentSystem: 这个类, 也要明天上午再看一下	5
1.9 ActorHelper: 帮助创建 IActorResponse 回复消息。很简单	7
1.10ActorMessageDispatcherInfo ActorMessageDispatcherComponent	7
1.11ActorMessageDispatcherComponentHelper: 帮助类	7
1.12ActorMessageHandlerAttribute 标签系: 去找几个典型标签看看	8
1.13[ActorMessageHandler(SceneType.Gate)] 标签使用举例	8
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	8
1.15MailboxType	8
1.16【服务端】ActorHandleHelper 帮助类。【需要去深挖一下】	8
1.17NetInnerComponentOnReadEvent:	10
2 StartConfigComponent: 找【各种服】的起始初始化地址	11
2.1 ConfigSingleton<T>: ProtoObject, ISingleton	11
2.2 SceneFactory 里可以给【匹配服】添加组件	11
2.3 RouterAddressComponent: 路由器组件	12
2.4 RouterAddressComponentSystem: 路由器的生成系	12
2.5 RouterHelper: 路由器帮助类, 向路由器注册、申请?	12
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!!!】	13
2.7 StartSceneConfig: ISupportInitialize 【各种服 - 配置, 场景配置】	14
2.8 StartSceneConfigCategory : 【Matches!】ConfigSingleton<StartSceneConfigCategory>, IMerge	15
2.9 HttpGetRouterResponse: 这个 ProtoBuf 的消息类型	16
2.10HttpGetRouterHandler : IHttpHandler: 获取各路路由器的地址	16
2.11HttpHandler 标签系: 标签自带场景类型	17
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	17
2.13GateSessionKeyComponent:	17

3 ET7 数据库相关【服务端】	17
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	18
3.2 DBComponent: 带生成系。可以查表，查询数据	18
3.3 DBManagerComponent: 有上面的 DBComponent 数组。数组长度固定吗?	18
3.4 DBManagerComponentSystem: 主要是要查询某个区服的数据库，从数组里	18
3.5 DBProxyComponent: 【参考项目】里的。有生成系。	18
3.6 StartZoneConfigCategory: 单例区服配置管理类	19
4 网关服：客户端信息发送的直接代理，中转站，组件分析	19
4.1 NetServerComponent:	20
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）	20
6 Session 会话框相关	21
7 Unit:	21
7.1 UnitGateComponent:	21
7.2 UnitGateComponentSystem	21
8 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	21
8.1 IComponentSerialize:	21
8.2 ClientComponent: 【参考项目】客户端组件，找个 ET7 里的组件	22
9 Protobuf 相关，【Protobuf 里进程间传递的游戏数据相关信息：两个思路】	22

1 Actor 消息相关

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

- Tcs 成员变量：精华在这里：因为内部自带一个 IActorResponse 的异步任务成员变量，可以帮助实现异步消息的自动回复
- 正是因为内部成员自带一个异步任务，所以会多一个成员变量，就是标记是否要抛异常。这是异步任务成员变量带来的

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

1.2 ActorMessageSenderComponent: 这个组件里有个计时器自动计时的超时时段、特定超时类型的超时时长成员变量

- 超时时间：这个组件有计时器自动计时和超时激活的逻辑，这里定义了这个组件类型的超时时长，在 ActorMessageSenderComponentSystem.cs 文件的 **Invoke(TimerInvokeType.ActorMess**

标注的 ActorMessageSenderChecker 里会用到，检测超时与否

```
[ComponentOf(typeof(Scene))]
public class ActorMessageSenderComponent: Entity, IAwake, IDestroy {
// 超时时间：这个组件有计时器自动计时和超时激活的逻辑，这里定义了这个组件类型的超时时长，在【Invoke(TimerInvokeType.ActorMes
    public const long TIMEOUT_TIME = 40 * 1000;
    public static ActorMessageSenderComponent Instance { get; set; }
    public int RpcId;
// 下面：这么理解：ET7 里是组件就是一个管理类，管理了对所有：不同 rpc-id 的消息发送器，所以有序字典管理，方便添加与移除
    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 ETTask 的细节，要再看几遍

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

```
[FriendOf(typeof(ActorMessageSenderComponent))]
public static class ActorMessageSenderComponentSystem {
// 要怎么理解这个消息发送超时检测中间过程步骤呢？
// 它自带个计时器，就是说，当服务器繁忙处理不过来，它就极有可能会自动超时，若是超时了，就返回个超时消息回去发送者告知一下，
// 那么，它就是一个服务端超负载下的自动减压逻辑
[Invoke(TimerInvokeType.ActorMessageSenderChecker)] // 另一个新标签，激活系：它标记说，这个激活系类，是 XXX 类型；紧
public class ActorMessageSenderChecker: ATimer<ActorMessageSenderComponent> {
    protected override void Run(ActorMessageSenderComponent self) { // 申明方法的接口是：ATimer<T> 抽象实现类，它实
        try {
            self.Check(); // 调用组件自己的方法
        } catch (Exception e) {
            Log.Error($"move timer error: {self.Id}\n{e}");
        }
    }
}
// 下面的方法是：帮助发送回复消息？好像不对，感觉异步任务结果已经出来了，只差最后写结果的步骤。
// 下面的方法是：回复消息是怎么发回去的？回复消息，只有最小初始化？从来不会有任何特异性？
// 传进来的参数：是一个 IActorResponse 实例，是有最小预处理（初始化了最基本成员变量）、【写了个半好】的结果。结果还没写进异步任务
private static void Run(ActorMessageSender self, IActorResponse response) {
// 对于每个超时了的消息：超时错误码都是：ErrorCore.ERR_ActorTimeout，所以会从发送消息超时异常里抛出异常，不用发送错误码
if (response.Error == ErrorCore.ERR_ActorTimeout) { // 写：发送消息超时异常。因为同步到异步任务 ETTask 里，所以只
    self.Tcs.SetException(new Exception($"Rpc error: request, 注意 Actor 消息超时，请注意查看是否死锁或者没有 re
    return;
}
// ETTask 模块的自动抛出异常：与当前发送消息超时过滤器里的异常逻辑相互独立？
// 【没想明白】：这个 Run() 方法，并不是只有 Check() 【发送消息超时异常】一个方法调用。什么情况下的调用，会走到下面的
// ActorMessageSenderComponent 一个组件，一次只执行一个（返回）消息发送任务，成员变量永远只管当前任务，
// 也是因为 Actor 机制是并行的，一个使者一次只能发一个消息 ...
// 【组件管理器的执行频率，Run() 方法的调用频率】：要是消息太多，发不完怎么办呢？这里没懂
if (self.NeedException && ErrorCore.IsRpcNeedThrowException(response.Error)) { // 若有异常（判断条件：消息要
    self.Tcs.SetException(new Exception($"Rpc error: actorId: {self.ActorId} request: {self.Request}, respons
    return;
}
self.Tcs.SetResult(response); // 【写结果】：将【写了个半好】的消息，写进同步到异步任务的结果里；把异步任务的状态设置
// 上面【异步任务 ETTask.SetResult()】，会调用注册过的一个回调，所以 ETTask 封装，设置结果这一步，会自动触发调用注册
// 【掉线的是】：ETTask.SetResult() 异步任务写结果了，非空回调是会调用。非空回调是什么，是把返回消息发回去吗？整个流
```

```

// 再去想 IMHandler: 它是消息处理器。问题就变成是, 当返回消息写好了, 写好了一个完整的可以发送、待发送的消息, 谁来处理
// 这个服, 这个自带计时器减压装置自带的消息处理器逻辑会处理?【这个逻辑在哪里, 要找出来!!!】要不然, 就没理解透彻。
}
private static void Check(this ActorMessageSenderComponent self) {
    long timeNow = TimeHelper.ServerNow();
    foreach ((int key, ActorMessageSender value) in self.requestCallback) {
        // 因为是顺序发送的, 所以, 检测到第一个不超时的就退出
        // 超时触发的激活逻辑: 是有至少一个超时的消息, 才会【激活触发检测】; 而检测到第一个不超时的, 就退出下面的循环。
        if (timeNow < value.CreateTime + ActorMessageSenderComponent.TIMEOUT_TIME)
            break;
        self.TimeoutActorMessageSenders.Add(key);
    }
}
// 超时触发的激活逻辑: 是有至少一个超时的消息, 才会【激活触发检测】; 而检测到第一个不超时的, 就退出上面的循环。
// 检测到第一个不超时的, 理论上说, 一旦有一个超时消息就会触发超时检测, 但实际使用上, 可能存在当检测逻辑被触发走到这里, 实际中
foreach (int rpcId in self.TimeoutActorMessageSenders) { // 一一遍历【超时的消息】:
    ActorMessageSender actorMessageSender = self.requestCallback[rpcId];
    self.requestCallback.Remove(rpcId);
    try { // ActorHelper.CreateResponse() 框架系统性的封装: 也是通过对消息的发送类型与对应的回复类型的管理, 使用帮
        // 对于每个超时的消息: 超时错误码都是: ErrorCore.ERR_ActorTimeout
        IActorResponse response = ActorHelper.CreateResponse(actorMessageSender.Request, ErrorCore.ERR_ActorTimeout);
        Run(actorMessageSender, response); // 猜测: 方法逻辑是, 把回复消息发送给对应的接收消息的 rpcId
    }
    catch (Exception e) {
        Log.Error(e.ToString());
    }
}
self.TimeoutActorMessageSenders.Clear();
}
// public static class ActorHelper { // 接上面 ActorHelper.CreateResponse() 用作参考
//     public static IActorResponse CreateResponse(IActorRequest iActorRequest, int error) {
//         Type responseType = OpcodeTypeComponent.Instance.GetResponseTypes(iActorRequest.GetType()); // 框架系统
//         IActorResponse response = (IActorResponse)Activator.CreateInstance(responseType); // 创建一个返回消息的
//         response.Error = error; // 写实例的出错结果、类型
//         response.RpcId = iActorRequest.RpcId; // 返回消息的接收者 RpcId: 实际就是, 消息是谁发来的, 就返回消息给谁
//         return response; // 返回这个最小初始化过的特定类型消息实例
//     }
// }
public static void Send(this ActorMessageSenderComponent self, long actorId, IMessage message) { // 发消息
    if (actorId == 0) {
        throw new Exception($"actor id is 0: {message}");
    }
    ProcessActorId processActorId = new(actorId);
    // 这里做了优化, 如果发向同一个进程, 则直接处理, 不需要通过网络层
    if (processActorId.Process == Options.Instance.Process) { // 没看懂: 这里怎么就说, 消息是发向同一进程的了?
        NetInnerComponent.Instance.HandleMessage(actorId, message);
        return;
    }
    Session session = NetInnerComponent.Instance.Get(processActorId.Process);
    session.Send(processActorId.ActorId, message);
}
public static int GetRpcId(this ActorMessageSenderComponent self) {
    return ++self.RpcId;
}
public static async ETask<IActorResponse> Call(
    this ActorMessageSenderComponent self,
    long actorId,
    IActorRequest request,
    bool needException = true
) {
    request.RpcId = self.GetRpcId();
    if (actorId == 0) {
        throw new Exception($"actor id is 0: {request}");
    }
    return await self.Call(actorId, request.RpcId, request, needException);
}
public static async ETask<IActorResponse> Call( // 发消息: 细节比较难懂。感觉还是对 ETask 异步任务没能理解透彻
    this ActorMessageSenderComponent self,
    long actorId,
    int rpcId,
    IActorRequest iActorRequest,
    bool needException = true
) {
    if (actorId == 0) {
        throw new Exception($"actor id is 0: {iActorRequest}");
    }
}

```

[illegible]

- 几个类看懂: ActorHandleHelper, 以及再上面的, NetInnerComponentOnReadEvent 事件发布等, 上层调用的几座桥连通了, 才算把整个流程看懂了。

1.4 LocationProxyComponent: 这个代理，什么情况下会用到？

- 就是有个启动类管理 `StartSceneConfigCategory` 类，它会分门别类地管理一些什么网关、注册登录服，地址服之类的东西。然后从这个里面拿位置服务器地址？大概意思是这样。
- 这个类先前仔细读过。还记得小伙伴搬家吗？有的小伙伴搬得很慢，要花很久，它搬家过程中就要上锁。大致是这类位置转移管理，位置添加、更新等相关管理操作。

```
[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 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 ETask<IActorResponse> CallInner(this ActorLocationSenderComponent self, ActorLocationSender actorLocationSender, int rpcId, IActorRequest iActorRequest) {
    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 ErrorCode.ERR_ActorTimeout:
        throw new RpcException(response.Error, $"{iActorRequest}");
    }
    if (ErrorCode.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: 帮助类

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

```

[FriendOf(typeof(ActorMessageDispatcherComponent))] // Actor 消息分发组件：对于管理器里的，对同一发送消息类型，不同场景下
public static class ActorMessageDispatcherComponentHelper { // Awake() Load() Destroy() 省略掉了
    private static void Load(this ActorMessageDispatcherComponent self) { // 加载：程序域回载的时候
        self.ActorMessageHandlers.Clear(); // 清空字典
        var types = EventSystem.Instance.GetTypes(typeof(ActorMessageHandlerAttribute)); // 扫描程序域里的特定消息处理器
        foreach (Type type in types) {
            object obj = Activator.CreateInstance(type); // 加载时：框架封装，自动创建【消息处理器】实例
            IActorHandler imHandler = obj as IActorHandler;
            if (imHandler == null) {
                throw new Exception($"message handler not inherit IActorHandler abstract class: {obj.GetType().FullName}");
            }
            object[] attrs = type.GetCustomAttributes(typeof(ActorMessageHandlerAttribute), false);
            foreach (object attr in attrs) {
                ActorMessageHandlerAttribute actorMessageHandlerAttribute = attr as ActorMessageHandlerAttribute;
                Type messageType = imHandler.GetRequestType(); // 因为消息处理接口的封装：可以拿到发送类型
                Type handleResponseType = imHandler.GetResponseTypes(); // 因为消息处理接口的封装：可以拿到返回消息的类型
                if (handleResponseType != null) {
                    Type responseType = OpcodeTypeComponent.Instance.GetResponseTypes(messageType);
                    if (handleResponseType != responseType) {
                        throw new Exception($"message handler response type error: {messageType.FullName}");
                    }
                }
            }
        }
    }
}

```



```

    }
    // 将必要的消息【发送类型】【返回类型】存起来，统一管理，备用
    // 这里，对于同一发送消息类型，是会、是可能存在【从不同的场景类型中返回，带不同的消息处理器】以致于必须得链表管理
    // 这里，感觉因为想不到、从概念上也地无法理解，可能会存在的适应情况、上下文场景，所以这里的链表管理同一发送消息类型，
    ActorMessageDispatcherInfo actorMessageDispatcherInfo = new(actorMessageHandlerAttribute.SceneType,
    self.RegisterHandler(messageType, actorMessageDispatcherInfo); // 存在本管理组件，所管理的字典里
    }
    }
}
private static void RegisterHandler(this ActorMessageDispatcherComponent self, Type type, ActorMessageDispatcherInfo actorMessageDispatcherInfo)
// 这里，对于同一发送消息类型，是会、是可能存在【从不同的场景类型中返回，带不同的消息处理器】以致于必须得链表管理
// 这里，感觉因为想不到、从概念上也地无法理解，可能会存在的适应情况、上下文场景，所以这里的链表管理同一发送消息类型，
if (!self.ActorMessageHandlers.ContainsKey(type))
    self.ActorMessageHandlers.Add(type, new List<ActorMessageDispatcherInfo>());
self.ActorMessageHandlers[type].Add(handler);
}
public static async ETask Handle(this ActorMessageDispatcherComponent self, Entity entity, int fromProcess, object message)
List<ActorMessageDispatcherInfo> list;
if (!self.ActorMessageHandlers.TryGetValue(message.GetType(), out list)) // 根据消息的发送类型，来取所有可能的消息处理器
    throw new Exception($"not found message handler: {message}");
SceneType sceneType = entity.DomainScene().SceneType; // 定位：当前消息的场景类型
foreach (ActorMessageDispatcherInfo actorMessageDispatcherInfo in list) { // 遍历：这个发送消息类型，所有存在该类型的消息处理器
    if (actorMessageDispatcherInfo.SceneType != sceneType) // 场景不符就跳过
        continue;
    // 定位：是当前特定场景下的消息处理器，那么，就调用这个处理器，要它去干事。【爱表哥，爱生活!!! 任何时候，活宝妹就是爱生活】
    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】

- 读了 ActorMessageSenderComponentSystem.cs 的具体的消息内容处理、发送，以及计时器消息的超时自动抛超时错误码过滤等底层逻辑处理，
- 读上下面的顶层的 NetInnerComponentOnReadEvent.cs 的顶层某个某些服，读到消息后的消息处理逻辑
- 知道，当前帮助类，就是衔接上面的两条顶层调用，与底层具体处理逻辑的桥，把框架上中下层连接连通起来。
- 分析这个类，应该可以理解底层不同逻辑方法的前后调用关系，消息处理的逻辑模块先后顺序，以及必要的可能的调用频率，或调用上下文情境等。明天上午再看一下

```

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

```

1.17 NetInnerComponentOnReadEvent:

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

// 这个，应该是服务端发布读事件后，触发的订阅者处理读到消息的回调逻辑：分消息类型，进行不同的处理

[Event(SceneType.Process)]

public class NetInnerComponentOnReadEvent: AEvent<NetInnerComponentOnRead> {

protected override async ETask Run(Scene scene, NetInnerComponentOnRead args) {

try {

long actorId = args.ActorId;

object message = args.Message;

// 收到 actor 消息，放入 actor 队列

switch (message) { // 分不同的消息类型，借助 ActorHandleHelper 帮助类，对消息进行处理。既处理【请求消息】，也处理【返回

case IActorResponse iActorResponse: {

ActorHandleHelper.HandleIActorResponse(iActorResponse);

break;

}

case IActorRequest iActorRequest: {

await ActorHandleHelper.HandleIActorRequest(actorId, iActorRequest);

break;

}

case IActorMessage iActorMessage: {

await ActorHandleHelper.HandleIActorMessage(actorId, iActorMessage);


```

    }
}

```

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 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 ETask<(uint, IPEndPoint)> GetRouterAddress(Scene clientScene, IPEndPoint address, uint localConn,
        Log.Info($"start get router address: {clientScene.Id} {address} {localConn} {remoteConn}");
        // return (RandomHelper.RandUInt32(), address);
        RouterAddressComponent routerAddressComponent = clientScene.GetComponent<RouterAddressComponent>();
        IPEndPoint routerInfo = routerAddressComponent.GetAddress();
        uint recvLocalConn = await Connect(routerInfo, address, localConn, remoteConn);
        Log.Info($"finish get router address: {clientScene.Id} {address} {localConn} {remoteConn} {recvLocalConn} {routerInfo}");
        return (recvLocalConn, 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[] 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} {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; }
}

```

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 {

```



```

    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.Matches.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 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 ETTask 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，这里还不是狠懂，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 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.RouterHttpPort, ConstValue.RouterHttpAddress);
                await routerAddressComponent.Init();
                clientScene.AddComponent<NetClientComponent, AddressFamily>(routerAddressComponent.RouterManagerIPAddress, AddressFamily.IPv4);
            }
            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.RealmAddress));
            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) // 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 ETask<Scene> CreateServerScene(Entity parent, long id, long instanceId, int zone, string name, SceneType sceneType) {
        await ETask.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>(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.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 Unit:

7.1 UnitGateComponent:

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

7.2 UnitGateComponentSystem

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

8 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。

8.1 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, 很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);
}

```

8.2 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;
    }
}

```

9 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