unity 游戏热更新服务端服务器

deepwaterooo

January 28, 2023

Contents

1 笔记

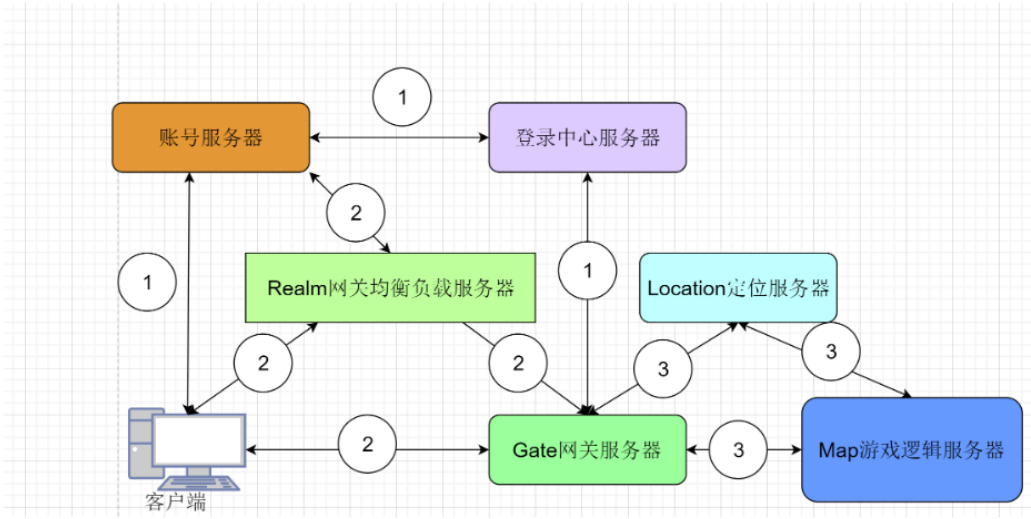
2 综述

- 这个框架相对比较平民化比较亲民，文档相对健全，关键模块和知识点讲解得相对透彻完善，更关键的是使用的人可能会比较多。自己遇到问题的时候能够网络上寻求的帮助来源多一点儿。会主要参考这个来搭写自己的框架

3 帐户服 + 数据库 + 登录中心服 + 网关服: 具体设计林框架参考: 今天上午终于感觉看懂一点儿了

- 下面主要记录别人站在相对比较高的角度总结出来的架构: https://blog.csdn.net/Q540670228/article/details/123592622

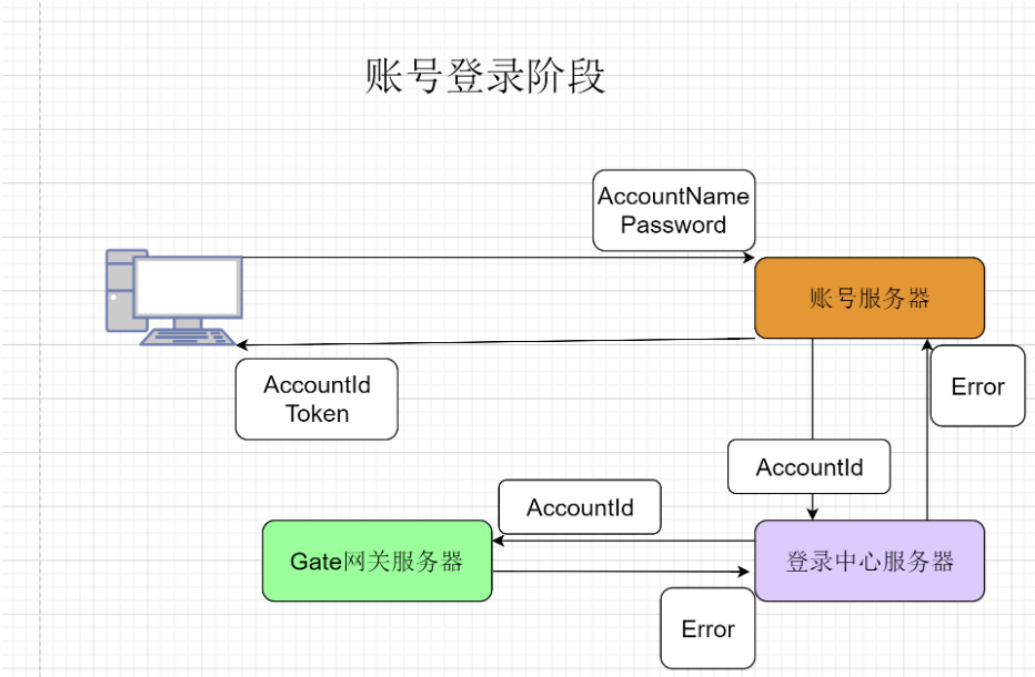
登陆系统全步骤图文



3.1 一. 账号登录

3.1.1 1. 客户端请求获取账户信息

1.客户端请求获取账户信息

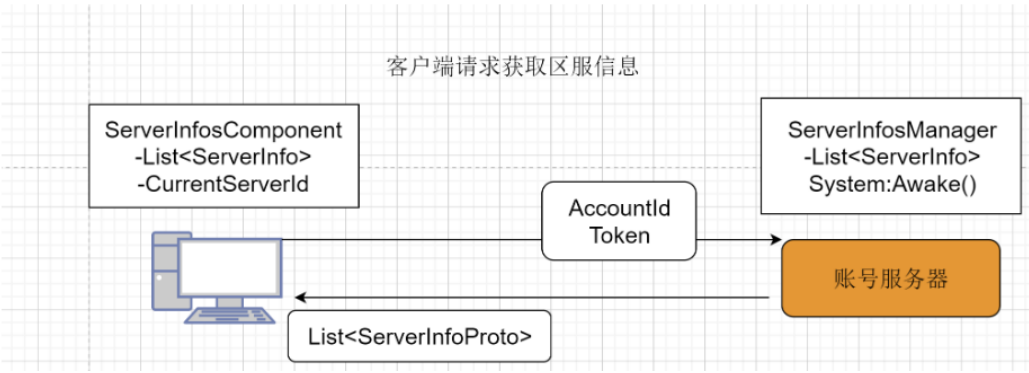


- 客户端向账号服务器发送账户名和密码的消息，请求进行登录
- 账号服务器和数据库交互，对信息进行验证或注册，获取账户唯一标识 AccountId
- 处理顶号逻辑，向登录中心服务器发送请求消息包含 AccountId
- 登录中心的组件存有 AccountId 和其所在区服 zone 的映射, 若不存在 AccountId 直接返回即可

- 若存在 AccounId, 根据其所在区服 zone 获取网关服务器的 InstanceId，进而向其发送下线消息
- 网关服务器存有所有客户端的映射实体 Player(存有 sessionInstanceId,UnitId,AccountId 等)
- 根据 AccountId 获取 Player 并通过 Player 的 SessionInstanceId 获取网关到客户端的 session 并释放
- 为 Player 添加下线组件 PlayerOfflineOutTimeComponent 后返回即可
- 登陆中心接到返回消息继续返回给账号服务器即可
- 账号服务器处理完顶号逻辑后，自身也应缓存 AccountId 和自身的 SessionId，做一次自身的顶号逻辑
- 最后随机生成一个 Token，把 Token 和 AccountId 发回给客户端
- 客户端将 AccountId Token 等基本信息保存在 zoneScene 的 AccountInfo 组件中, 供后续使用

3.1.2 2. 客户端请求获取区服信息

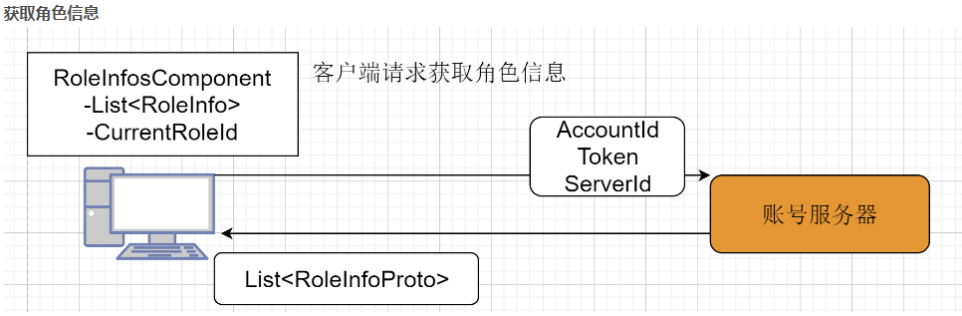
2. 客户端请求获取区服信息



- 定义 ServerInfo 区服信息实体 (区服名, 状态) 及与 proto 映射的相互转换行为，并在双端均定义组件用以保存区服信息列表。
- 服务器的组件要为其添加行为，Awake 时就应该将数据库中的区服信息读取出来放入组件的信息列表中
- 客户端向账号服务器发送信息，请求获取所有的区服信息，需要发送 AccountId 和 Token 用于验证账户
- 账号服务器将服务器信息列表转换为其对应的 ServerInfoProto 列表，并发送回客户端
- 客户端接收到后将 proto 转换回 ServerInfo 并保存到组件当中，显示到 UI 层供用户选择。
- **注意 ServerInfoProto 存在的意义，ET 框架下网络传输的必须是 Proto 对象，不能直接是实体，所以需要定义 Proto 作为传输对象，在双端进行转换使用。**

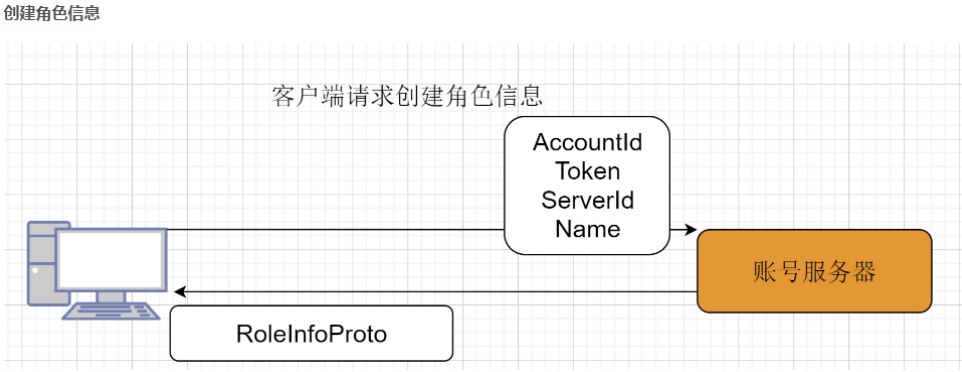
3.1.3 3. 客户端请求获取/创建/删除角色信息

1. 获取角色信息



- 获取角色信息的步骤和获取服务器信息很类似
  - 定义 RoleInfo 实体 (Name,AccountId,State,ServerId)，并为其提供和 proto 的相互转换行为
  - 实体是双端可以公用的，但账号服务器无需保存，只需在请求时从数据库获取完发回即可
  - 客户端向服务器发送消息请求获取当前账户在当前区服下的所有未冻结的角色
  - 服务器通过 Id 和 Token 验证身份后，从数据库中获取所有角色信息转换成 proto 对象发回客户端
  - 客户端收到后将 proto 转换成 RoleInfo 存储在相应的组件中，显示到 UI 层供用户选择。

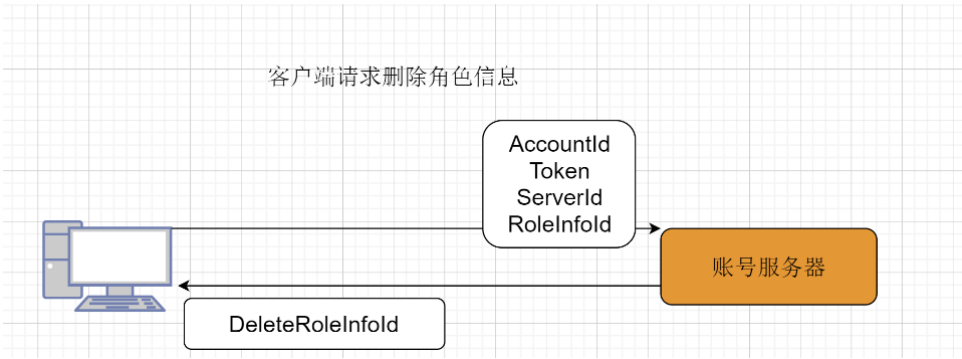
2. 创建角色信息



- 用户和 UI 交互输入名称并点击按钮创建角色
- 将用于验证的信息以及 ServerId 和用户输入的姓名一并发向服务器请求创建角色（区服间角色独立）
- 服务器判断是否有重复角色，若没有则创建新角色 RoleInfo，并对其各属性进行初始化。
- 初始化利用 Id 创建使用 GenerateUnitId，创建完成后保存到数据库中并转换成 Proto 发回客户端
- 客户端收到 RoleInfoProto 后转换成 RoleInfo 并缓存起来，然后在 UI 管理处进行刷新 UI 循环列表

3. 删除角色信息

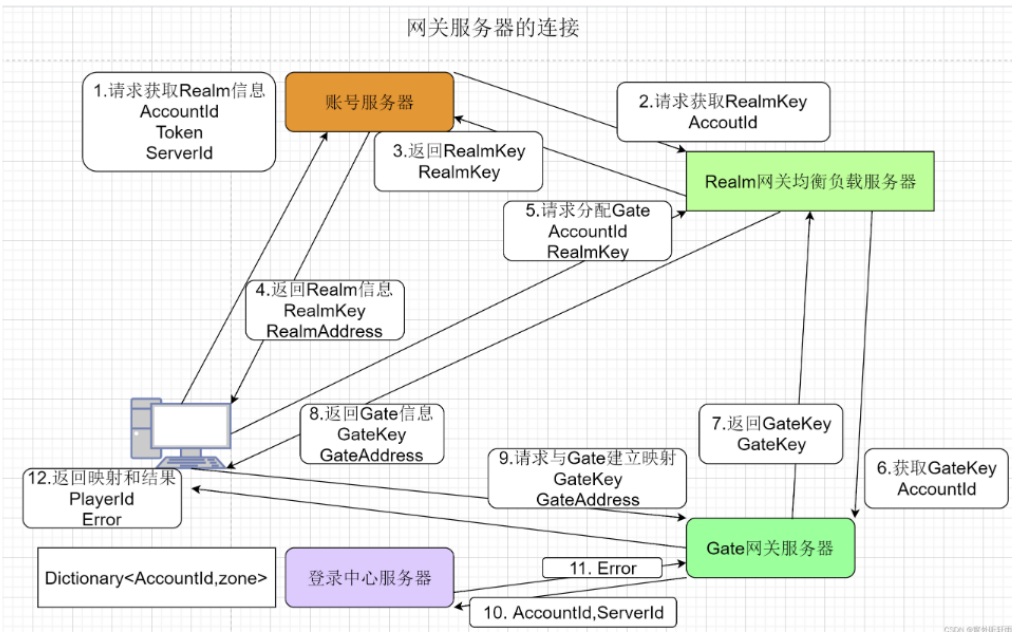
删除角色信息



- 在 UI 界面的循环列表为每个角色添加选择按钮，选择后会为组件的 CurrentRoleId 赋值选中的角色
- 向账号服务器发送请求删除角色的信息，其中的 RoleInfoId 即为选择的 CurrentRoleId。
- 账号服务器在客户端中查询到指定 Id 的 RoleInfo 将其状态设置为 Freeze 冻结并修改名称 (防止后续注册同名问题)
- 发回客户端删除的 RoleInfo 的 Id，客户端接收后在组件集合中将其移除并刷新 UI 界面。

3.2 二. 网关服务器的连接

二. 网关服务器的连接



- 网关服务器的的连接其实就是，客户端先和 Realm 网关连接请求其分配一个 Gate 网关，然后客户端去连接此 Gate 网关。

3.2.1 1. 请求连接 Realm 网关

- 向账号服务器请求获取 Realm 网关的地址和令牌，需要区服 Id，一般一个区服下有一个 Realm

- 账号服务器通过配置文件获取 Realm 网关的内网地址 (sceneInstanceId)，并向其请求获取 RealmKey 令牌。
- Realm 网关随机生成令牌 RealmKey 和 AccountId 将映射保存在组件中，将 Key 发回账号服务器
- 账号服务器通过配置文件获取 Realm 网关的外网地址 (OuterIPPort), 和令牌 RealmKey 一并发回客户端

3.2.2 2. 请求和 Gate 网关连接

- 客户端与账号服务器断连，与 Realm 建立连接，并向其请求分配网关服务器（即获取一个网关信息）
- 一个区服下一般有多个 Gate，Realm 通过与账户 Id 取模的方式固定分配给此账户一个 Gate，向此 Gate 请求获取 GateKey
- Gate 网关服务器随机生成一个 GateKey 并将 AccountId 和 GateKey 的映射关系保存供后续验证，并发回 Key
- Realm 服务器将 Gate 信息（key,address-配置文件得）发回客户端，客户端与 Realm 进行断开，准备连 Gate
- 连接 Realm 服务，验证登陆用户与密码
- 如果验证通过玩家需要与其他服务模块进行交互，不会要求玩家再进行一次密码认证。使用一种认证机制，即通过 Realm 服向 Gate 服请求一个认证 key，交给玩家。
- 当玩家连接 Gate 服时，拿着申请好的 key，直接登陆 Gate 服务即可通过 Gate 服的允许。
- 这里可以看见源码中的一个小细节

```
// ResponseType G2R_GetLoginKey: R2G ==> Realm to Game 从 Realm 服务发送给一条 actor 消息给 Gate 服，向它注册并申请一条登陆许可
message R2G_GetLoginKey // IActorRequest
{
    int32 RpcId = 1;
    string Account = 2;
}
```

3.2.3 3. 建立 Gate 映射对象 Player

- 客户端一般会与 Gate 长时间连线，需要为 Session 添加心跳组件 PingComponent，请求在 Gate 中创建映射对象 Player
- 步骤 10 和步骤 11，主要是客户端与 Gate 建立连接后，将账户 Id 和区服号发送至登陆中心服务器进行注册添加，登录逻辑中会通过此服务器的记录进行顶号逻辑，通过区服号和 AccountId 利用 Realm 帮助类能唯一确定 Gate, 再给 Gate 发送下线消息即可。

3.2.4 建立 Player 步骤

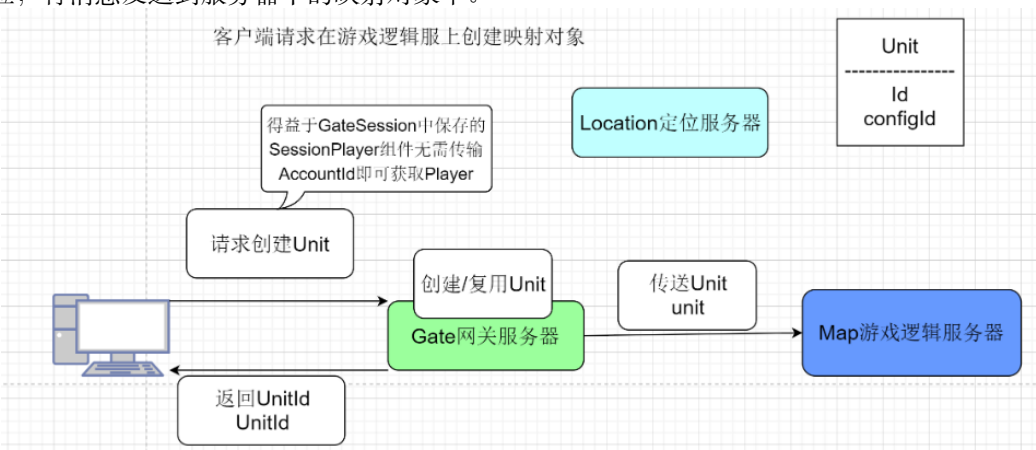
- 建立 Player 实体 (AccountId,UnitId,SessionInstanceId,state)，Player 和账户 ID, 网关和客户端的 Session 连接以及 Unit 达到一一对应
- 为网关到客户端的 Session 添加 PlayerComponent 保存所有 Player 实体 (AccountId 和 Player 映射字典), 并为其添加 SessionStateComponent, 用于判断网关连接是否处于 Normal 或 Game（便于后续 Unit 逻辑）
- 为网关到客户端 Session 添加 SessionPlayerComponent 组件 (AccountId,PlayerInstanceId) 和 Player 一一对应，即在网关连接 Session 的此组件上直接获取相应 Player，这样处理后续的游戏逻辑就不用每次都发送 AccountId 从 PlayerComponent 中获取了（节省传输量）

- 判断是否可以复用 Player，顶号下线时可以复用 (后面有流程图解释)，如果复用必须移除 Player 身上的下线组件，更新 Session，即更新 Player 身上的 SessionInstanceId 和 Session 身上的 SessionPlayerComponent 重新创建。
- 如果不是顶号等操作，直接创建 Player 并初始化即可，PlayerId 用 RoleId，UnitId 暂时用 RoleId，后续创建出游戏逻辑服 Unit 后用其替换。

将 PlayerId 返回给客户端供客户端可能使用。

3.3 三. 游戏逻辑服务器连接

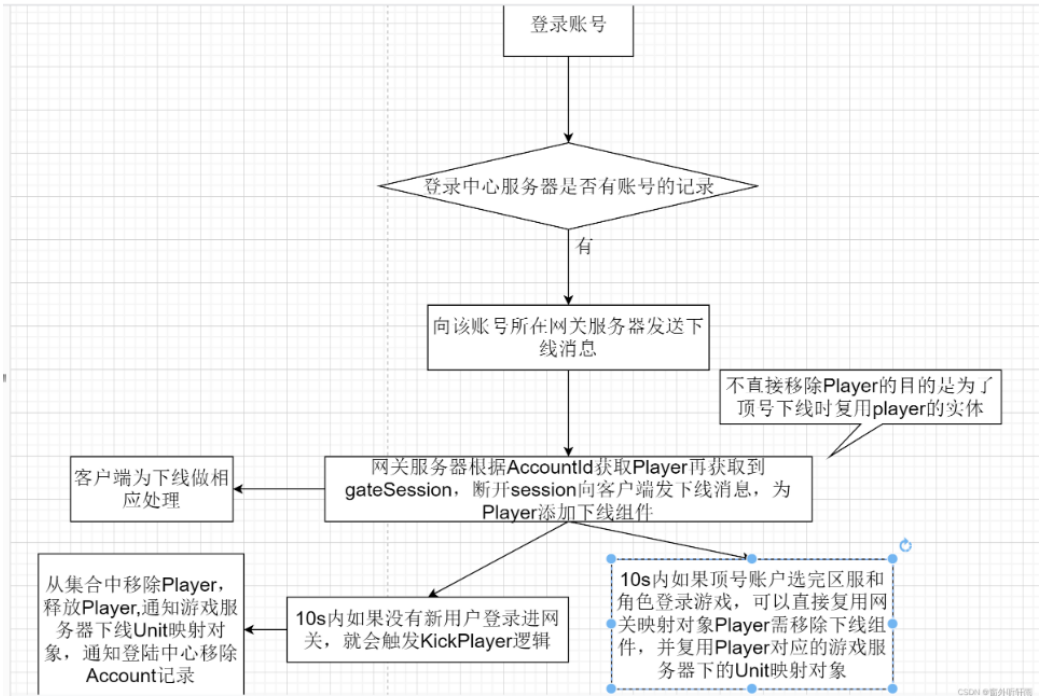
- 游戏逻辑服务器的连接本质上并不是客户端和其直接相连，而是通过在游戏服务器上建立一个映射对象和客户端绑定，客户端以后即可通过此映射对象的 Id，通过网关转发和 Location 服务器的定位，将消息发送到服务器下的映射对象中。



- 客户端向 Gate 网关请求在 Map 服务器上创建 Unit 映射对象
- 网关服务器先判断是否是顶号操作 (利用 Player 的状态并向 Player 的 Unit 发送测试消息)，验证成功后可以直接复用 Player 下的 Unit 并将 UnitId 返回客户端。
- 若非顶号，则需要先临时为 Player 添加 gateMapComponent 组件，其下有一个属性 Scene，在此 Scene 中创建一个 Map 场景用于后续传送 Unit，(TransferHelper 只能用于 Map 场景的传送，所以才做这一步)
- 在上述创建的 Map 场景下创建 Unit 对象，UnitId 可直接使用 Player 的 Id 即 RoleId，然后必须为 Unit 添加 UnitGateComponent，其中保存了 gateSessionActorId 即 gateSessionInstanceId，(这样就可以利用 Unit 直接给客户端下发消息了)。
- 利用配置文件获取 Map 服务器地址，利用 TransferHelper 的 Transfer 函数将 unit 传送到 Map 游戏逻辑服务器中
- Transfer 传送实现机制，实现以下机制后返回消息
  - 通知客户端切换场景（直接利用 unit 下组件中 gateSessionInstanceId 直接下发即可）
  - request 消息中保存 Unit 并将 Unit 下所有实现了 ITransfer 接口的组件保存起来一会一起传输过去
  - 删除当前 Unit 下的 MailBoxComponent 让发给此 Unit 的消息重发到正确位置（可能 Unit 还没传输过去就有信息发过来了）
  - 对 Location 定位服务器进行加锁，发送 IActor 消息传输给 Map 服务器，并释放当前 Unit

  - Map 服务器接收到消息将 Unit 和其组件重新添加 (AddChild) 到在此服务器下的 UnitComponent 中，将 Unit 添加到此组件集合中 (传输时无法传输原 Unit 对象下的组件，只能将原 Unit 下基础属性以 proto 传递过来，在此还需重新生成)
  - 向客户端发相应的消息和属性，让客户端同步显示出角色并将 Unit 实体加入 AOIEntity（AOI 作用笔者暂且还未研究大概跟客户端有联系）
- 传送完毕后将 UnitId 传回客户端即可，后续客户端就可利用 UnitId 发送 IActorLocation 消息和服务器上的 Unit 发送消息了。

3.4 四. 顶号逻辑流程图



- 顶号逻辑属于是账号系统较为复杂的逻辑，其主要用到了中心登录服务器暂存玩家当前状态，并创建了 Player 和 Unit 映射对象，通过 Player 暂存到网关中实现顶号逻辑可以无需重新创建 Player 和 Unit 直接更新属性复用，大大提高了顶号的效率。

4 帐户服 + 数据库 + 登录中心服 + 网关服：具体设计逻辑相关实现源码学习

- 上面是别人总结出来的大框架, 现在看仍是如云里雾里. 项目是可以运行起来, 并有 demo 小项目可以参考的
- 可以根据上面的步骤与日志, 把运行过程中的游戏端 (ServerClient 模式) 游戏热更新 Model.dll Hotfix.dll 的源码看懂, 弄明白这个框架是如何实现登录相关, 以及必要的游戏逻辑服务器热更新的 (如果有多余时间的话), 应该就会对这个框架有相对更好的理解, 可以考虑开始适配自己的简单服务器
- 可以比较两种不同的启动模式有什么不同?
- 先去找游戏客户端里, 程序的入口是在哪里, 逻辑如何连贯起来的? 因为项目比较大, 看一次不曾自己真正实现过, 狠容易就看一次忘记一次, 所以记好笔记狠重要

- MangoDB 数据库的连接，要把这个配置好，才能真正理解这个框架与范例

5 Bson 序列化与反序列化：Core/MongoHelper.cs 零碎的知识点大概总结在这些部分

- Mongo Bson 非常完善，是我见过功能最全使用最强大的序列化库，有些功能十分贴心。其支持功能如下：
  - 支持复杂的继承结构
  - 支持忽略某些字段序列化
  - 支持字段默认值
  - 结构多出多余的字段照样可以反序列化，这对多版本协议非常有用
  - 支持 ISupportInitialize 接口使用，这个在反序列化的时候简直就是神器
  - 支持文本 json 和二进制 bson 序列化
  - MongoDB 数据库支持
- 这里看一个关于反序列化时的继承关系所涉及到的点：
- 支持复杂的继承结构
- mongo bson 库强大的地方在于完全支持序列化反序列化继承结构。需要注意的是，**继承反序列化需要注册所有的父类**，有两种方法：
- a. 你可以在父类上面使用 [BsonKnownTypes] 标签声明继承的子类，这样 mongo 会自动注册，例如：

```
[BsonKnownTypes(typeof(Entity))]
public class Component
{
}
[BsonKnownTypes(typeof(Player))]
public class Entity: Component
{
}
public sealed class Player: Entity
{
    public long Id;

    public string Account { get; set; }

    public long UnitId { get; set; }
}
```

- 这样有缺陷，因为框架并不知道一个类会有哪些子类，**这样做对框架代码有侵入性，我们希望能解除这个耦合**。
- b. 可以扫描程序集中所有子类父类的类型，将他们注册到 mongo 驱动中

```
Type[] types = typeof(Game).Assembly.GetTypes();
foreach (Type type in types)
{
    if (!type.IsSubclassOf(typeof(Component)))
    {
        continue;
    }

    BsonClassMap.LookupClassMap(type); // <<<<<<<<<<<<<<<<<<<<
}
BsonSerializer.RegisterSerializer(new EnumSerializer<NumericType>(BsonType.String));
```

- 这样完全的自动化注册，使用者也不需要关系类是否注册。
- 这里还有点儿别的比较有价值的，但是今天没有看完：https://www.kktoo.com/wiki/etnotes/chapter1/3.2%E5%BC%BA%E5%A4%A7%E7%9A%84MongoBson%E5%BA%93.html 就是感觉要看 ET 框架中的源码，看正看明白了才会懂，现在看剩下的部分仍然是不懂（改天再看）

6 后面消息处理的部分大致逻辑：《参考网上的：可对对照源码，把它不懂的也解惑了。。。》

- 在消息处理这方面，它的逻辑是这样的，
- https://blog.csdn.net/qq_33574890/article/details/128244264?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EYuanLiJiHua%7EPosition-3-128244264-blog-88990234.pc_relevant_aa2&depth_1-utm_source=distribut pc_relevant.none-task-blog-2%7Edefault%7EYuanLiJiHua%7EPosition-3-128244264-blog-pc_relevant_aa2&utm_relevant_index=4 大家最开始也都同我一样，一知半解的，总有个熟悉的过程
- [与自己项目的本区别：] 自己的不是网络游戏，手游移动端，不是实现游戏模型逻辑的部分热更新，而是游戏的全部逻辑除了游戏入口在正常游戏程序域，其它都可以都全部放入了热更新程序域。所以自己的热更新并不涉及任何游戏过程逻辑与游戏逻辑服的交互，简化了无限多的大型网络游戏的逻辑模型热更新的步骤。下面 ET 框架更多的是用于大型网游的吧。。。。。
- 在点击进入地图的按钮之后他会发送一个消息到服务器，服务器会发送一个玩家 ID 回来 (这些都是，游戏过程中，一个客户端与游戏逻辑服的《实时？异步］交互)
- 这个玩家 ID 就是客户端的唯一标识
- 在收到服务器的回复消息时，再发送一个创建 Unit 的内部消息
- 这个内部消息的处理者会创建这个 Unit 并进行广播，为什么要广播，不是很懂》[不是要适配大型多人网络游戏吗，不广播新的玩家进来了，其它玩家如何知晓呢]？为什么不用事件而要用内部消息也很奇怪《事件系统可以区分不同玩家吗？消息机制可以跨进程，可以这个玩家发给同区不同区其它玩家跟发短信一样，广播消息都都用 actor 消息机制吗？为什么要用事件系统，事件系统定义了玩家生成，玩家步移，玩家跳动跑动，玩家打架，玩家失血，玩家挂掉了吗？本质上讲应该似乎两种都可以，但消息机制可能更为简单广泛通用?》?
- 在创建 Unit 的时候在他身上挂载了寻路组件和移动组件 (如果是事件系统，就只有玩家生成，与玩家移动两个事件了吗？最主要的，事件系统是否定义了关于玩家这些逻辑的所有相关功能模块？没有，上面就是消息机制最为简单)。
- 而游戏场景里面有一个 OperaCompoent，这个组件会实时监听按键的点击并且将点击的位置发送给服务器，服务器再传送回客户端
- 这边也挺奇怪的，[不奇怪，实现的是功能逻辑的元件组件化，可装载可卸载，相对更容易理解与精减维护源码]

- 客户端这边接收 ClickMapActor 消息后解析位置，调用 unit 的 Path 组件进行移动。大体流程就是这样。
- 但其实这样很不连贯，我发送一个消息，应该 await 等待获得消息，然后再做处理，这样就弄的 * 很分散不够集中 *[这个是说得很精准，以致于我现在入口加载都找上几十个文件弱弱拎不清楚。。。。。]

7 事件处理

- ET 的所有逻辑全部用事件来处理了
- 这带来一个坏处，就是没办法高内聚。《为什么要高内聚呢？本来目的出发点就是要低内聚，减少功能模块逻辑的偶合 decoupling 呀》
- 本来属于一个业务模型里面的逻辑，通过事件分散到了两到三个脚本里面。增加了阅读难度和上下连贯性。《这是如亲爱的表哥眼中的弱弱活宝妹般，小弱弱们读大型多人网络游戏框架的必经之痛，都得有这个过程》
- 单纯举例来说，游戏初始化后发送了 InitScenFinish 事件。
- UI 处理模块接收到事件创建 UI 物体并显示
- 在各个 UI 单独的组件比如 LoginCom 和 LobbyCom 里面进行 UI 逻辑撰写即并绑定事件
- 他把本属于 Model 层的代码全部写进了静态类 Helper 里面来调用。《是这样吗???》
- 也就是说 View 层直接调用了 Model 层代码。其实这样就强耦合了。《以小人之心度君子之腹，明明你自己也承认是帮助类，帮助类就不该是模型逻辑层？！在如亲爱的表哥眼中的活宝妹般弱弱的年代，这类错误概念设计上的理解，都是小弱弱们的辛酸泪，成长的代价。。。。。爱表哥，爱生活!!!》
- 直接调用 MapHelper
- 上面就当从网络上搜来的梗概提要，等自己读源码的时候对照来理解，修正补确

8 BuildModelAndHotfix: 一定要构建生成热更新程序集，项目狠大，两个程序集的内容没能消化好

- 它有几种不同的启动模式，可以再具体区分一下
- 这里面有好多个项目，也要区分哪些是可以热更新，哪些是不能够热更新的，

  - 没细看源码，竟然不知道，Unity.Model 里面的代码不能热更新，通常将游戏中不会变动的部分放在这个项目里

- 下面是根据范例加载过程的追踪来理解这么多个工程。自己项目的热更新等程序集都能弄懂并解决所有的问题。但是这个项目太大，感觉现在仍然有点儿无从下口的状态。。。。

9 Init.cs: 程序真正的入口

```
public class Init: MonoBehaviour {

    private void Start() {
        DontDestroyOnLoad(gameObject);

        AppDomain.CurrentDomain.UnhandledException += (sender, e) => {
            Log.Error(e.ExceptionObject.ToString());
        };

        Game.AddSingleton<MainThreadSynchronizationContext>(); // 线程上下文的无缝切换，可以高枕无忧不用管了
        // 命令行参数
        string[] args = "".Split(" ");
        Parser.Default.ParseArguments<Options>(args)
            .WithNotParsed(error => throw new Exception($" 命令行格式错误! {error}"))
            .WithParsed(Game.AddSingleton);

// 注意，每个被 Add 的组件，都会执行其 Awake（前提是他有类似的方法），这也是 ETBook 中的内容，不懂的同学回去补课哦
        Game.AddSingleton<TimeInfo>();
        Game.AddSingleton<Logger>().ILog = new UnityLogger();
        Game.AddSingleton<ObjectPool>();
        Game.AddSingleton<IdGenerater>();
        Game.AddSingleton<EventSystem>();
        Game.AddSingleton<TimerComponent>();
        Game.AddSingleton<CoroutineLockComponent>();

        ETTask.ExceptionHandler += Log.Error;
        Game.AddSingleton<CodeLoader>().Start(); // <<<<<<<<<<<<<<<<<<<<
    }

// 框架中关注过的，几个统一管理的生命周期回调函数的一致系统化管理调用.
    private void Update() {
        Game.Update(); // <<<<<<<<<<
    }
    private void LateUpdate() {
        Game.LateUpdate(); // <<<<<<<<<<
        Game.FrameFinishUpdate(); // <<<<<<<<<<
    }
    private void OnApplicationQuit() {
        Game.Close(); // <<<<<<<<<<
    }
}
```

10 Game.cs: 这个类会涉及到一些生命周期的管理等

```
public static class Game {
    [StaticField]
    private static readonly Dictionary<Type, ISingleton> singletonTypes = new Dictionary<Type, ISingleton>();
    [StaticField]
    private static readonly Stack<ISingleton> singletons = new Stack<ISingleton>();
    [StaticField]
    private static readonly Queue<ISingleton> updates = new Queue<ISingleton>();
    [StaticField]
    private static readonly Queue<ISingleton> lateUpdates = new Queue<ISingleton>();
    [StaticField]
    private static readonly Queue<ETTask> frameFinishTask = new Queue<ETTask>();

    public static T AddSingleton<T>() where T: Singleton<T>, new() {
        T singleton = new T();
        AddSingleton(singleton);
        return singleton;
    }

    public static void AddSingleton(ISingleton singleton) {
        Type singletonType = singleton.GetType();
```

```
        if (singletonTypes.ContainsKey(singletonType)) {
            throw new Exception($"already exist singleton: {singletonType.Name}");
        }
        singletonTypes.Add(singletonType, singleton);
        singletons.Push(singleton);

        singleton.Register();
        if (singleton is ISingletonAwake awake) {
            awake.Awake(); // 如果它实现过该接口，就会自动调用这个回调函数
        }

        if (singleton is ISingletonUpdate) {
            updates.Enqueue(singleton);
        }

        if (singleton is ISingletonLateUpdate) {
            lateUpdates.Enqueue(singleton);
        }
    }

// 这个类里，只有这个方法是，等待异步执行结果结束的，但是即便执行结束了，可能还没有设置结果，会晚些时候再设置结果
    public static async ETTask WaitFrameFinish() {
        ETTask task = ETTask.Create(true); // 从池里抓一个新的出来用
        frameFinishTask.Enqueue(task);     // 入队
// <<<<<<<<<<<<<<<<<<< 这里是，异步等待任务的执行吗？应该是 假如开启了池,await 之后不能再操作 ETTask，否则可能操作到再次从池中分配
        await task;
    }

    public static void Update() {
        int count = updates.Count;
        while (count-- > 0) {
            ISingleton singleton = updates.Dequeue();
            if (singleton.IsDisposed()) {
                continue;
            }
            if (singleton is not ISingletonUpdate update) {
                continue;
            }

            updates.Enqueue(singleton);
            try {
                update.Update();
            }
            catch (Exception e) {
                Log.Error(e);
            }
        }
    }
    public static void LateUpdate() {
        int count = lateUpdates.Count;
        while (count-- > 0) {
            ISingleton singleton = lateUpdates.Dequeue();

            if (singleton.IsDisposed()) {
                continue;
            }
            if (singleton is not ISingletonLateUpdate lateUpdate) {
                continue;
            }

            lateUpdates.Enqueue(singleton);
            try {
                lateUpdate.LateUpdate();
            }
            catch (Exception e) {
                Log.Error(e);
            }
        }
    }
    public static void FrameFinishUpdate() {
        while (frameFinishTask.Count > 0) {
// 为什么我会觉得这里它只是把 ETTask 从任务队列里取出来，并不曾真正执行过呢？它是在什么时候执行的，逻辑在哪里？前面那个异步方法调用的
            ETTask task = frameFinishTask.Dequeue();
            task.SetResult();
        }
    }
    public static void Close() {
        // 顺序反过来清理：反过来清理才能真正清理得干净
        while (singletons.Count > 0) {
```

```
            ISingleton iSingleton = singletons.Pop();
            iSingleton.Destroy();
        }
        singletonTypes.Clear();
    }
}
```

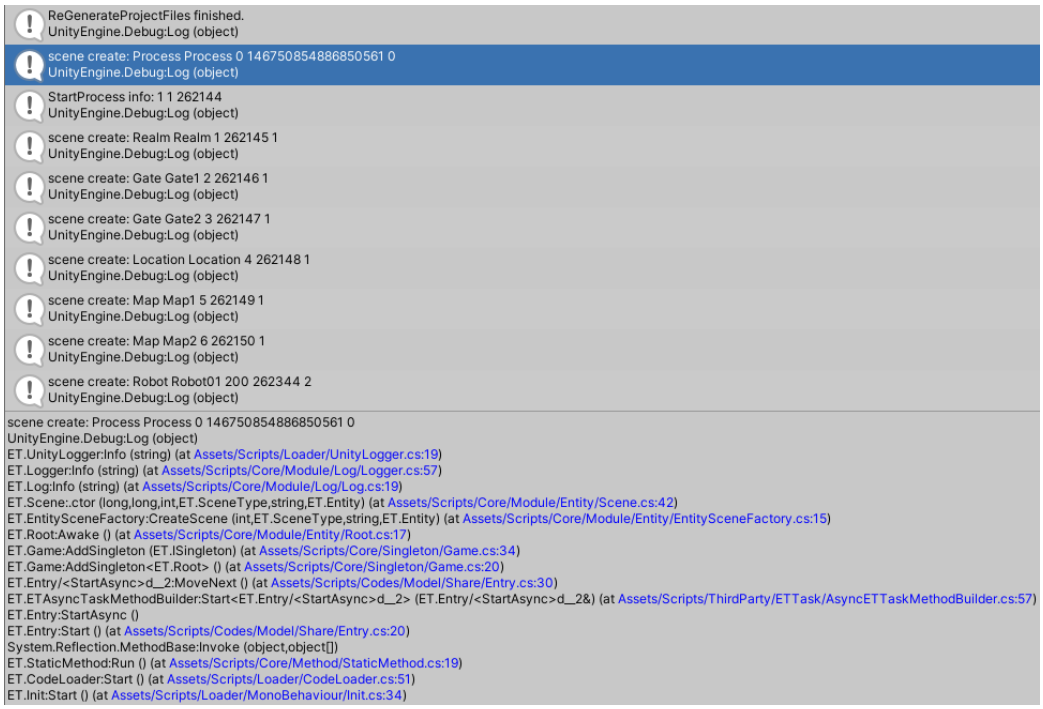
11 CodeLoader.cs: 加载热更新等各种程序集

```
public class CodeLoader: Singleton<CodeLoader> {

    private Assembly model; // <<<<<<<<<< Model.dll, Hotfix.dll

    public void Start() {
        if (Define.EnableCodes) {
            GlobalConfig globalConfig = Resources.Load<GlobalConfig>("GlobalConfig");
            if (globalConfig.CodeMode != CodeMode.ClientServer) {
                throw new Exception("ENABLE_CODES mode must use ClientServer code mode!");
            }
            Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
            Dictionary<string, Type> types = AssemblyHelper.GetAssemblyTypes(assemblies);
            EventSystem.Instance.Add(types);
            foreach (Assembly ass in assemblies) {
                string name = ass.GetName().Name;
                if (name == "Unity.Model.Codes") {
                    this.model = ass;
                }
            }
            IStaticMethod start = new StaticMethod(this.model, "ET.Entry", "Start"); // <<<<<<<<<< 调用热更新静态方法入口
            start.Run();
        } else {
            byte[] assBytes;
            byte[] pdbBytes;
            if (!Define.IsEditor) {
                Dictionary<string, UnityEngine.Object> dictionary = AssetsBundleHelper.LoadBundle("code.unity3d");
                assBytes = ((TextAsset)dictionary["Model.dll"]).bytes;
                pdbBytes = ((TextAsset)dictionary["Model.pdb"]).bytes;
                HybridCLRHelper.Load();
            } else {
                assBytes = File.ReadAllBytes(Path.Combine(Define.BuildOutputDir, "Model.dll"));
                pdbBytes = File.ReadAllBytes(Path.Combine(Define.BuildOutputDir, "Model.pdb"));
            }
            this.model = Assembly.Load(assBytes, pdbBytes);
            this.LoadHotfix();
            IStaticMethod start = new StaticMethod(this.model, "ET.Entry", "Start");
            start.Run();
        }
    }
    // 热重载调用该方法
    public void LoadHotfix() {
        byte[] assBytes;
        byte[] pdbBytes;
        if (!Define.IsEditor) {
            Dictionary<string, UnityEngine.Object> dictionary = AssetsBundleHelper.LoadBundle("code.unity3d");
            assBytes = ((TextAsset)dictionary["Hotfix.dll"]).bytes;
            pdbBytes = ((TextAsset)dictionary["Hotfix.pdb"]).bytes;
        } else {
            // 傻🅵 Unity 在这里搞了个傻逼优化，认为同一个路径的 dll，返回的程序集就一样。所以这里每次编译都要随机名字
            string[] logicFiles = Directory.GetFiles(Define.BuildOutputDir, "Hotfix_*.dll");
            if (logicFiles.Length != 1) {
                throw new Exception("Logic dll count != 1");
            }
            string logicName = Path.GetFileNameWithoutExtension(logicFiles[0]);
            assBytes = File.ReadAllBytes(Path.Combine(Define.BuildOutputDir, $"{logicName}.dll"));
            pdbBytes = File.ReadAllBytes(Path.Combine(Define.BuildOutputDir, $"{logicName}.pdb"));
        }
        Assembly hotfixAssembly = Assembly.Load(assBytes, pdbBytes);
        Dictionary<string, Type> types = AssemblyHelper.GetAssemblyTypes(typeof (Game).Assembly, typeof(Init).Assembly, thi
        EventSystem.Instance.Add(types);
    }
}
```

12 Entry.cs: Assets/Scripts/Codes/Model/Share/Entry.cs 不是真正的入口

```
namespace ET {

    namespace EventType {
        public struct EntryEvent1 {
        }

        public struct EntryEvent2 {
        }

        public struct EntryEvent3 {
        }
    }

// 这是程序的固定入口吗 ?  不是
    public static class Entry {
        public static void Init() {
        }

        public static void Start() {
            StartAsync().Coroutine();
        }
// 相关的初始化: Bson, ProtoBuf, Game.NetServices, Root etc
        private static async ETTask StartAsync() {
            WinPeriod.Init(); // Windows 平台 Timer Tick 的时间精度设置

            MongoHelper.Init();   // MongoDB 数据库的初始化: 这里像是没作什么工程, 但涉及类相关所有静态变量的初始化
            ProtobufHelper.Init();// 同上: 这个没有太细看, 改天用到可以补上

            Game.AddSingleton<NetServices>(); // 网络连接初始化: 还没有理解透彻
            Game.AddSingleton<Root>();        // 它说是, 管理场景根节点的, 没看
            await Game.AddSingleton<ConfigComponent>().LoadAsync(); // Config 组件会扫描所有的有 ConfigAttribute 标签的配置,
            await EventSystem.Instance.PublishAsync(Root.Instance.Scene, new EventType.EntryEvent1());
            await EventSystem.Instance.PublishAsync(Root.Instance.Scene, new EventType.EntryEvent2());
            await EventSystem.Instance.PublishAsync(Root.Instance.Scene, new EventType.EntryEvent3());
        }
    }
}
```

13 Root.cs UI 的启动过程



14 配置文件的加载过程: 就接上了 ConfigComponent ?

- 在加载完服务端的 Hotfix 和 Model 的 dll 后，开始后面就是开始读取服务端的配置，并根据配置加载相应的组件。
- 服务端的配置的读取方法使用的是 CommandLine，该类用于读取命令行输入，并且将输入参数解析成 Options 类，存放在 OptionComponent 组件里面。[这里面的步骤还没有找]
- 这里的参数来自客户端的配置文件，在 Config\StartConfig 目录下。
- 配置的读取是从客户端猫大写的工具里面读取的，具体在 ServerCommandLineEditor 类里面。

15 ServerCommandLineEditor.cs:

```
public enum DevelopMode {
    正式 = 0,
    开发 = 1,
    压测 = 2,
}
public class ServerCommandLineEditor: EditorWindow {
    [MenuItem("ET/ServerTools")]
    public static void ShowWindow() {
        GetWindow<ServerCommandLineEditor>(DockDefine.Types);
    }
    private int selectStartConfigIndex = 1;
    private string[] startConfigs;
    private string startConfig;
    private DevelopMode developMode;

    public void OnEnable() {
        DirectoryInfo directoryInfo = new DirectoryInfo("../Config/Excel/s/StartConfig");
        this.startConfigs = directoryInfo.GetDirectories().Select(x => x.Name).ToArray();
```

```
        }

        public void OnGUI() {
            selectStartConfigIndex = EditorGUILayout.Popup(selectStartConfigIndex, this.startConfigs);
            this.startConfig = this.startConfigs[this.selectStartConfigIndex];
            this.developMode = (DevelopMode) EditorGUILayout.EnumPopup(" 起服模式: ", this.developMode);
            string dotnet = "dotnet.exe";
#if UNITY_EDITOR_OSX
            dotnet = "dotnet";
#endif
            if (GUILayout.Button("Start Server(Single Process)")) {
                string arguments = $"App.dll --Process=1 --StartConfig=StartConfig/{this.startConfig} --Console=1";
                ProcessHelper.Run(dotnet, arguments, "../Bin/");
            }
            if (GUILayout.Button("Start Watcher")) {
                string arguments = $"App.dll --AppType=Watcher --StartConfig=StartConfig/{this.startConfig} --Console=1";
                ProcessHelper.Run(dotnet, arguments, "../Bin/");
            }
            if (GUILayout.Button("Start Mongo")) {
                ProcessHelper.Run("mongod", @"--dbpath=db", "../Database/bin/");
            }
        }
    }
```

- 感觉上面的配置类的定义, 仍然是看得不明不白, 看来这里得加把劲儿了
- 启动服务器的时候，传递的参数如下:

```
string arguments = $"App.dll --appId={startConfig.AppId} --appType={startConfig.AppType} --config=../Config/StartConfig/{th
ProcessStartInfo info = new ProcessStartInfo("dotnet", arguments) { UseShellExecute = true, WorkingDirectory = @"../Bin/" }
```

- 读取完配置后，将配置保存在 StartConfigComponent 组件里面，该组件在初始化时会根据配置的情况，分门别类将配置内容存进行缓存。接下来就是添加网络相关的组件 OpcodeTypeComponent 和 MessageDispatherComponent 组件，具体可看这篇文章《ET 框架学习——OpcodeTypeComponent 组件和 MessageDispatherComponent 组件》。https://blog.csdn.net/Tong1993222/article/details/86600357
- 最后就是根据配置的 AppType 类型，添加对应的组件，这里我选择的是 AllServer 类型，所有相关组件都会添加，具体可以参看源码。
- 当前链接: https://blog.csdn.net/Tong1993222/article/details/88990234
- 下面是自己的理解: 程序最开始加载程序集的时候, 或是某个什么地方 (没能理解透彻), 会自动扫描程序集中的带配置标签的配置, 进行配置

16 ConfigComponent.cs: Config 组件会扫描所有的有 ConfigAttribute 标签的配置, 加载进来

```
// Config 组件会扫描所有的有 ConfigAttribute 标签的配置, 加载进来
public class ConfigComponent: Singleton<ConfigComponent> {

    public struct GetAllConfigBytes {
    }
    public struct GetOneConfigBytes {
        public string ConfigName;
    }
    private readonly Dictionary<Type, ISingleton> allConfig = new Dictionary<Type, ISingleton>();

    public override void Dispose() {
        foreach (var kv in this.allConfig) {
            kv.Value.Destroy();
        }
    }
    public object LoadOneConfig(Type configType) {
        this.allConfig.TryGetValue(configType, out ISingleton oneConfig);
        if (oneConfig != null) {
            oneConfig.Destroy();
        }
```

```
        byte[] oneConfigBytes = EventSystem.Instance.Invoke<GetOneConfigBytes, byte[]>(0, new GetOneConfigBytes() {ConfigNa
        object category = SerializeHelper.Deserialize(configType, oneConfigBytes, 0, oneConfigBytes.Length);
        ISingleton singleton = category as ISingleton;
        singleton.Register();

        this.allConfig[configType] = singleton;
        return category;
    }

    public void Load() {
        this.allConfig.Clear();
        Dictionary<Type, byte[]> configBytes = EventSystem.Instance.Invoke<GetAllConfigBytes, Dictionary<Type, byte[]>>(0,
        foreach (Type type in configBytes.Keys) {
            byte[] oneConfigBytes = configBytes[type];
            this.LoadOneInThread(type, oneConfigBytes);
        }
    }

    public async ETTask LoadAsync() {
        this.allConfig.Clear();
        Dictionary<Type, byte[]> configBytes = EventSystem.Instance.Invoke<GetAllConfigBytes, Dictionary<Type, byte[]>>(0,
        using ListComponent<Task> listTasks = ListComponent<Task>.Create();

        foreach (Type type in configBytes.Keys) {
            byte[] oneConfigBytes = configBytes[type];
            Task task = Task.Run(() => LoadOneInThread(type, oneConfigBytes)); // <<<<<<<<<<<<<<<<<<<<
            listTasks.Add(task);
        }
        await Task.WhenAll(listTasks.ToArray());
        foreach (ISingleton category in this.allConfig.Values) {
            category.Register();
        }
    }

    private void LoadOneInThread(Type configType, byte[] oneConfigBytes) {
        object category = SerializeHelper.Deserialize(configType, oneConfigBytes, 0, oneConfigBytes.Length); // <<<<<<<<<<<
        lock (this) {
            this.allConfig[configType] = category as ISingleton;
        }
    }
}
```

17 ConfigAttribute.cs: 就是空定义, 用来标注这个标签就可以了

```
[AttributeUsage(AttributeTargets.Class)]
public class ConfigAttribute: BaseAttribute {
}
```

- 上面的不是是扫描配置标签吗，扫描完了就根据配置标签来配置呀

18 StartProcessConfig: ProtoObject, IConfig

- 这些自动生成的文件，还没有搞明白：为什么它们是自动生成的？为什么要把配置的过程定义成自动生成？一类配置自动生成的方法定义，在这个框架中有什么好处？可以提精提纯简化这类标签配置的源码吗？

```
    [ProtoContract]
  [Config]
  public partial class StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge {
      [ProtoIgnore]
      [BsonIgnore]
      private Dictionary<int, StartProcessConfig> dict = new Dictionary<int, StartProcessConfig>();

      [BsonElement]
      [ProtoMember(1)]
      private List<StartProcessConfig> list = new List<StartProcessConfig>();

      public void Merge(object o) {
          StartProcessConfigCategory s = o as StartProcessConfigCategory;
          this.list.AddRange(s.list);
      }
```

```
        [ProtoAfterDeserialization]
        public void ProtoEndInit() {
            foreach (StartProcessConfig config in list) {
                config.AfterEndInit();
                this.dict.Add(config.Id, config);
            }
            this.list.Clear();

            this.AfterEndInit();
        }

        public StartProcessConfig Get(int id) {
            this.dict.TryGetValue(id, out StartProcessConfig item);
            if (item == null) {
                throw new Exception($" 配置找不到，配置表名: {nameof (StartProcessConfig)}, 配置 id: {id}");
            }
            return item;
        }

        public bool Contain(int id) {
            return this.dict.ContainsKey(id);
        }
        public Dictionary<int, StartProcessConfig> GetAll() {
            return this.dict;
        }
        public StartProcessConfig GetOne() {
            if (this.dict == null || this.dict.Count <= 0) {
                return null;
            }
            return this.dict.Values.GetEnumerator().Current;
        }
    }

    [ProtoContract]
    public partial class StartProcessConfig: ProtoObject, IConfig {
        // <summary>Id</summary>
        [ProtoMember(1)]
        public int Id { get; set; }
        // <summary> 所属机器 </summary>
        [ProtoMember(2)]
        public int MachineId { get; set; }
        // <summary> 内网端口 </summary>
        [ProtoMember(3)]
        public int InnerPort { get; set; }
    }
```

19 partial class StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge

```
[ProtoContract]
[Config]
public partial class StartProcessConfigCategory : ConfigSingleton<StartProcessConfigCategory>, IMerge {
    [ProtoIgnore]
    [BsonIgnore]
    private Dictionary<int, StartProcessConfig> dict = new Dictionary<int, StartProcessConfig>();

    [BsonElement]
    [ProtoMember(1)]
    private List<StartProcessConfig> list = new List<StartProcessConfig>();

    public void Merge(object o) {
        StartProcessConfigCategory s = o as StartProcessConfigCategory;
        this.list.AddRange(s.list);
    }

    [ProtoAfterDeserialization] // 它说这里就是提供的狠好用的接口，方便在反序列化之后进行什么必要的操作
    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;
    }
}
```

20 StartProcessConfig.cs:

```
public partial class StartProcessConfig {

    private IPEndPoint innerIPPort;
    public long SceneId;

    public IPEndPoint InnerIPPort {
        get {
            if (this.innerIPPort == null) {
                this.innerIPPort = NetworkHelper.ToIPEndPoint($"{this.InnerIP}:{this.InnerPort}");
            }
            return this.innerIPPort;
        }
    }

    public string InnerIP => this.StartMachineConfig.InnerIP;
    public string OuterIP => this.StartMachineConfig.OuterIP;
    public StartMachineConfig StartMachineConfig => StartMachineConfigCategory.Instance.Get(this.MachineId);

    public override void AfterEndInit() {
        InstanceIdStruct instanceIdStruct = new InstanceIdStruct((int)this.Id, 0);
        this.SceneId = instanceIdStruct.ToLong();
        Log.Info($"StartProcess info: {this.MachineId} {this.Id} {this.SceneId}");
    }
}
```

- 不知道今天上午用 VSC 出什么问题，程序跑不通了，等再关几个窗口，重新运行得通的时候才可以再追踪日志，先等等
- 它有几个模式：好像要有一个一一对应，就是一个服务器，一个客户端，大致的意思是说，同在游戏引擎里，不能一次弄出两三个客户端之类的吧（就是要么 clientserver 模式，要么 server + client, 别弄了个 servier, 结果又构建了个 clientserver）这样程序就可以正常运行了
- 感觉那么追日志看得好艰难，跟不上. 就先把游戏中需要打包构建的几个主要的程序自己先理解一遍，不懂的上网查，再跟着日志看。现在看感觉仍然不知道在哪里

21 HotfixView 程序域：先看这部分，这里面的大部分的源码都还比较简单

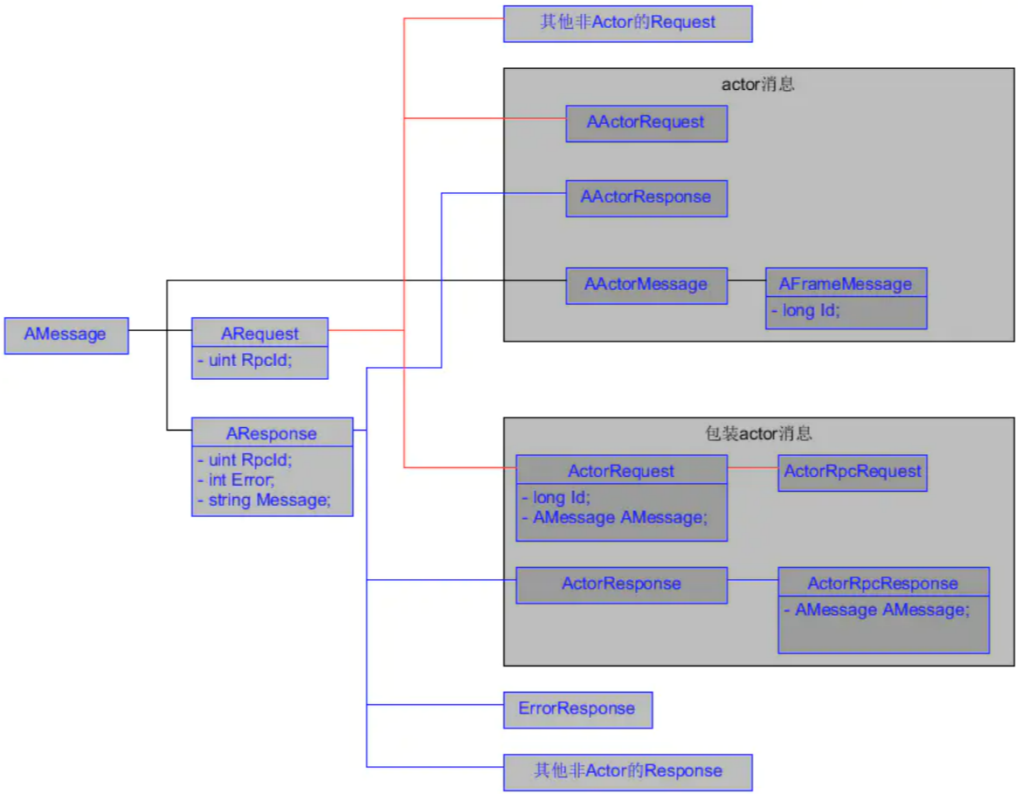
- 应快把样例工程中的源码爬一遍
- 现在基本的原理能够懂得：把大的版块基本能够看得懂的地方看懂，也稍微总结一下. 不懂的地方就用安卓应用快速地整几个应用来测试掌握一下

- **Awake, Load, Update, LateUpdate, Destroy:** 这个系统最简单，无处不在，类似 Unity3d 的组件，ET 框架也提供了组件事件，例如 Awake，Start，Update 等。要给一个 Component 或者 Entity 加上这些事件，必须写一个辅助类。
- ET 中大致类等的特点总结：
  - Entity：对应一个真实物体（至少也是一个逻辑实体）。特征：一定有一个代表这个实体的唯一 Id。
  - Component: 一个实体针对某个特定片面的状态数据，比如位置数据。注意，要求只有数据不能有函数（基本的 get,set 类函数可以包含）
  - System：对 Entity 中某些 Component 进行计算和更改的方法集合
  - Utility 函数：如果 System 之间相互依赖某些函数，则需要把这些函数提取为工具函数。也就是说，所有 System 之间需要确保不要太多相互依赖。
  - 单例 Component：如果某个状态是全局的，而不是和某个具体实体相关的，则需要提取

22 消息机制的原理，类图，总体概念理解

- https://www.jianshu.com/p/f2ecf148bc2f

22.1 消息分类



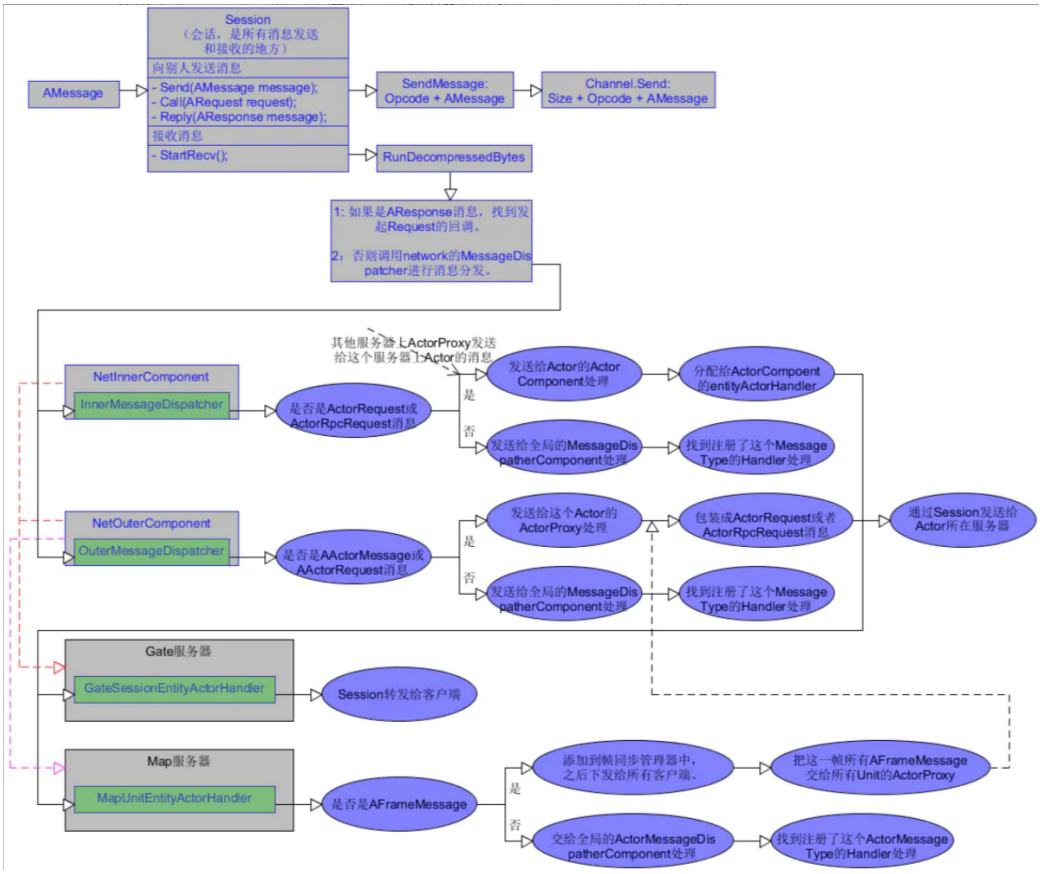
- 消息按照因果关系分类，可以分为 Request 和 Response，当然也有直接继承至 Message 的，表示我这个消息发出去后我就不 care 了。Request 和 Response 是成对的，我发了一个 Request，你必须回我一个对应的 Response 消息。

- 消息还可以按照类型分类，基本分为两大类，一般消息和 Actor 消息。Actor 消息可以认为是战斗相关消息，战斗 unit 和玩家 client 之间发送的都是 Actor 消息，比如帧同步消息（AFrameMessage）就是一种 ActorMessage，其他的都可以认为是一般消息，比如登陆，创角等。

- Actor 相关的消息在服务器间相互转发的时候，会被包装一下。想象一下，一个 Client 与 GateServer 之间的通信，消息里面是不用带玩家角色 Id 的，我们之间的 Session 对象就表明了我的身份。但是 GateServe 把这个消息转发给 MapServer 的时候，消息就得包装一下，带上 UnitId 的信息，这样 MapServer 收到这个消息后才知道是那个玩家发过来的。包装 actor 消息包括 ActorRequest 和 ActorRPCRequest，AActorMessage 不需要回复，就包装在 ActorRequest 中，而 AActorRequest 需要回复，包装在 ActorRPCRequest 中。所以 ActorRequest 和 ActorRPCRequest 包括他们的回复消息只在服务器之间传送 Actor 消息的时候用。

- 其实按照消息的路径，还可以把消息分为内部消息和外部消息，内部就是服务器内部之间发送的，外部则是负责服务器和客户端通信的。在 ET 中，有 InnerMessage 和 OuterMessage 两个文件，里面定义的消息就分别是内部和外部消息。只有 Realm 服务器和 Gate 服务器有 NetOuterComponent，可以与客户端通信，其他服务器都只有 NetInnerComponent，Realm 服务器作为初始登陆服务器，负责分配 Gate 服务器给玩家，之后玩家都直接与 Gate 服务器通信，之后其他服务器都通过 Gate 服务器与玩家通信。

22.2 ET 消息流程



- 这里面弄清了几个原则，基本就清楚了消息的处理原则：

- 1：Gate 服务器既需要与客户端通信，也需要与其他服务器通信。所以需要同时拥有 NetOuterComponent 和 NetInnerComponent。
- 2：Map 服务器不直接与客户端通信，需要通过 Gate 服务器转发，所以只有 NetInnerComponent。
- 3：客户端发送给 Gate 服务器的消息，都是通过 NetOuterComponent 走的，所以 Actor 消息都是 AActorMessage 或者 AActorRequest 类型（因为 Gate 服务器上接收消息的 Session 就表明了客户端的身份），而这些 Actor 消息是需要转发给 Map 服务器的，Gate 服务器和每个 Map 服务器之间都只有一个 Session（属于 NetInnerComponent），所有 Actor 消息都通过这个 Session 发送，所以 Actor 消息在这里需要包装成 ActorRequest 或者 ActorRpcRequest 消息，带 actorId，Map 服务器接收到这个消息后通过 actorId 才知道交给哪个 Actor 处理。
- 4：Map 服务器发送给 Gate 服务器的 ActorRequest 或者 ActorRpcRequest 消息，Gate 服务器只需要把包装里面的 AMessage 发送给对应的客户端即可。
- 这里加个链接：latex graphcs .9 倍的图片宽度，修改的话可以尝试：https://tex.stackexchange.com/questions/439918/set-default-value-for-max-width-of-includegraphics

22.3 这里，我在讲一下我自己的理解，方便看完后还是一脸懵逼的同学理解。

- 我们先别管那些类，我们先想想，我们需要发送的消息，都是些什么？
  - 1. 客户端发送给服务器的消息
  - 1. 不需要与其他服务器通信（普通消息）
  - 1. 不需要返回结果（普通的普通消息）
  - 2. 需要返回结果（普通的 RPC 消息）
  - 2. 需要与其他服务通信（Actor 消息）
  - 1. 不需要返回结果（普通的 Actor 消息）
  - 2. 需要返回结果（Actor RPC 消息）
  - 2. 服务器发送给客户端的消息
  - 1. 返回客户端请求的消息（根据客户端的请求消息类型发送对应的回复类型）
  - 2. 主动发送的消息，比如帧同步消息。
- 以上的属于服务端与客户端之间的消息类型，皆属于 OuterMessage。（外部消息）
- 3. 服务器与其他服务器对话的消息（属于内部消息 InnerMessage，且是 Actor 消息）
  - 1. 需要返回结果（Actor RPC 消息）
  - 2. 不需要返回结果（普通的 Actor 消息）
- 这里大家可以想象一下，服务器接收到其他服务器传来的 Actor 消息，其实就像是收到客户端传来的普通消息一样。所以 InnerMessageDispatcher 就没有必要再把消息发送给其他服务器上处理。
- 简单的说，就是这样。
  - 普通消息只要发送给一个服务器就能得到结果
  - actor 消息可能得通过其他服务器才能得到结果
  - actor 消息又分 actor rpc 消息，rpc 消息会返回结果。

23 几个链接需要再学习或是练习一下的:

- https://www.jianshu.com/p/2aaf4ab0682e
- https://blog.csdn.net/m0_48781656/article/details/123771424
- https://blog.csdn.net/tong1993222/article/details/89026556?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-89026556-pc_relevant_landingrelevant&spm=1001.2101.3001.4242.1&utm_relevant_index=3
- https://blog.csdn.net/norman_lin/article/details/79929284
- https://blog.csdn.net/weixin_34033624/article/details/86013121?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EEESLAND
7Edefault-3-86013121-blog-123771424.pc_relevant_landingrelevant&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EESLANDING%7Edefault-3-pc_relevant_landingrelevant&utm_relevant_index=6
- https://blog.csdn.net/Tong1993222/article/details/89026556
- https://blog.csdn.net/weixin_45029839/article/details/118491670
- 爱表哥，爱生活!!! 一定会嫁给偶亲爱的表哥!!!
- 现在还没有还不曾弄清楚的：MongoDB 的数据库：可视化界面： 可视化界面 Robo 3T: https://studio3t.com/download-now/#windows

24 ET 中的事件系统: 事件机制 EventSystem

- ECS 最重要的特性一是数据跟逻辑分离，二是数据驱动逻辑。什么是数据驱动逻辑呢？不太好理解，我们举个例子一个 moba 游戏，英雄都有血条，血条会在人物头上显示，也会在左上方头像 UI 上显示。这时候服务端发来一个扣血消息。我们怎么处理这个消息？第一种方法，在消息处理函数中修改英雄的血数值，修改头像上血条显示，同时修改头像 UI 的血条。这种方式很明显造成了模块间的耦合。第二种方法，扣血消息处理函数中只是改变血值，血值的改变抛出一个 hpchange 的事件，人物头像模块跟 UI 模块都订阅血值改变事件，在订阅的方法中分别处理自己的逻辑，这样各个模块负责自己的逻辑，没有耦合。ET 提供了多种事件，事件都是可以多次订阅的:
- 1.AwakeSystem，组件工厂创建组件后抛出，只抛出一次，可以带参数
- 2.StartSystem，组件 UpdateSystem 调用前抛出
- 3.UpdateSystem，组件每帧抛出
- 4.DestroySystem，组件删除时抛出
- 5.ChangeSystem，组件内容改变时抛出，需要开发者手动触发
- 6.DeserializeSystem，组件反序列化之后抛出
- 7.LoadSystem，EventSystem 加载 dll 时抛出，用于服务端热更新，重新加载 dll 做一些处理，比如重新注册 handler
- 8. 普通的 Event，由开发者自己抛出，可以最多带三个参数。另外客户端热更层也可以订阅 mono 层的 Event 事件
- 9. 除此之外还有很多事件，例如消息事件。消息事件使用 MessageHandler 来声明，可以带参数指定哪种服务器需要订阅。

```
// 1.AwakeSystem，组件工厂创建组件后抛出，只抛出一次，可以带参数
Player player = ComponentFactory.Create<Player>();
// 订阅 Player 的 Awake 事件
public class PlayerAwakeSystem: AwakeSystem<Player> {
    public override void Awake(Player self) {
    }
}

// 2.StartSystem，组件 UpdateSystem 调用前抛出
// 订阅 Player 的 Start 事件
public class PlayerStartSystem: StartSystem<Player> {
    public override void Start(Player self) {
    }
}

// 3.UpdateSystem，组件每帧抛出
// 订阅 Player 的 Update 事件
public class PlayerUpdateSystem: UpdateSystem<Player> {
    public override void Update(Player self) {
    }
}

// 4.DestroySystem，组件删除时抛出
// 订阅 Player 的 Destroy 事件
public class PlayerDestroySystem: DestroySystem<Player> {
    public override void Destroy(Player self) {
    }
}
Player player = ComponentFactory.Create<Player>();
// 这里会触发 Destroy 事件
player.Dispose();

// 5.ChangeSystem，组件内容改变时抛出，需要开发者手动触发
// 订阅 Player 的 Destroy 事件
public class PlayerChangeSystem: ChangeSystem<Player> {
    public override void Change(Player self) {
    }
}
Player player = ComponentFactory.Create<Player>();
// 需要手动触发 ChangeSystem
Game.EventSystem.Change(player);

// 6.DeserializeSystem，组件反序列化之后抛出
// 订阅 Player 的 Deserialize 事件
public class PlayerDeserializeSystem: DeserializeSystem<Player> {
    public override void Deserialize(Player self) {
    }
}
// 这里 player2 会触发 Deserialize 事件
Player player2 = MongoHelper.FromBson<Player>(player.ToBson());

// 7.LoadSystem，EventSystem 加载 dll 时抛出，用于服务端热更新，重新加载 dll 做一些处理，比如重新注册 handler
// 订阅 Player 的 Load 事件
public class PlayerLoadSystem: LoadSystem<Player> {
    public override void Load(Player self) {
    }
}

// 8. 普通的 Event，由开发者自己抛出，可以最多带三个参数。另外客户端热更层也可以订阅 mono 层的 Event 事件
int oldhp = 10;
int newhp = 5;
// 抛出 hp 改变事件
Game.EventSystem.Run("HpChange", oldhp, newhp);
// UI 订阅 hp 改变事件
[Event("HpChange")]
public class HpChange_ShowUI: AEvent<int, int> {
    public override void Run(int a, int b) {
        throw new NotImplementedException();
    }
}
// 模型头顶血条模块也订阅 hp 改变事件
[Event("HpChange")]
public class HpChange_ModelHeadChange: AEvent<int, int> {
    public override void Run(int a, int b) {
        throw new NotImplementedException();
    }
}

// 9. 除此之外还有很多事件，例如消息事件。消息事件使用 MessageHandler 来声明，可以带参数指定哪种服务器需要订阅。
```

```
[MessageHandler(AppType.Gate)]
public class C2G_LoginGateHandler : AMRpcHandler<C2G_LoginGate, G2C_LoginGate> {
    protected override void Run(Session session, C2G_LoginGate message, Action<G2C_LoginGate> reply) {
        G2C_LoginGate response = new G2C_LoginGate();
        reply(response);
    }
}
```

25 ET 的 ACTOR 的使用

- 普通的 Actor，我们可以参照 Gate Session。map 中 [这里，它说的应该是一个游戏逻辑服务器] 一个 Unit，Unit 身上保存了这个玩家对应的 gate session。这样，map 中的消息如果需要发给客户端，只需要把消息发送给 gate session，gate session 在收到消息的时候转发给客户端即可。map 进程发送消息给 gate session 就是典型的 actor 模型。它不需要知道 gate session 的位置，只需要知道它的 InstanceId 即可。MessageHelper.cs 中，通过 GateSessionActorId 获取一个 ActorMessageSender，然后发送。

```
// 从 Game.Scene 上获取 ActorSenderComponent，然后通过 InstanceId 获取 ActorMessageSender
ActorSenderComponent actorSenderComponent = Game.Scene.GetComponent<ActorSenderComponent>();
ActorMessageSender actorMessageSender = actorSenderComponent.Get(unitGateComponent.GateSessionActorId);
// send
actorMessageSender.Send(message);
// rpc
var response = actorMessageSender.Call(message);
```

- 问题是 map 中怎么才能知道 gate session 的 InstanceId 呢？这就是你需要想方设法传过去了，比如 ET 中，玩家在登录 gate 的时候，gate session 挂上一个信箱 MailBoxComponent，C2G_LoginGateHandler.cs 中

```
session.AddComponent<MailBoxComponent, string>(MailboxType.GateSession);
```

- 玩家登录 map 进程的时候会把这个 gate session 的 InstanceId 带进 map 中去，C2G_EnterMapHandl 中

```
M2G_CreateUnit createUnit = (M2G_CreateUnit)await mapSession.Call(new G2M_CreateUnit() { PlayerId = player.Id, GateSessionI
```

26 ACTOR 消息的处理

- 首先，消息到达 MailboxComponent，MailboxComponent 是有类型的，不同的类型邮箱可以做不同的处理。
- 目前有两种邮箱类型 GateSession 跟 MessageDispatcher。
  - GateSession 邮箱在收到消息的时候会立即转发给客户端，
  - MessageDispatcher 类型会再次对 Actor 消息进行分发到具体的 Handler 处理，
- **默认的 MailboxComponent 类型是 MessageDispatcher**。
- 自定义一个邮箱类型也很简单，继承 IMailboxHandler 接口，加上 MailboxHandler 标签即可。
- 那么为什么需要加这么个功能呢，在其它的 actor 模型中是不存在这个特点的，一般是收到消息就进行分发处理了。
  - 原因是 GateSession 的设计，并不需要进行分发处理，因此我在这里加上了邮箱类型这种设计。
- MessageDispatcher 的处理方式有两种: 一种是处理对方 Send 过来的消息，一种是 rpc 消息 < 这里，我可以简单理解为需要返回的类型吗？所以会有第三个参数，返回消息的类型 >

```
// 处理 Send 的消息，需要继承 AMActorHandler 抽象类，抽象类第一个泛型参数是 Actor 的类型，第二个参数是消息的类型
[ActorMessageHandler(AppType.Map)]
public class Actor_TestHandler : AMActorHandler<Unit, Actor_Test> {

    protected override ETTask Run(Unit unit, Actor_Test message) {
        Log.Debug(message.Info);
    }
}

// 处理 Rpc 消息，需要继承 AMActorRpcHandler 抽象类，抽象类第一个泛型参数是 Actor 的类型，第二个参数是消息的类型，第三个参数是返回
[ActorMessageHandler(AppType.Map)]
public class Actor_TransferHandler : AMActorRpcHandler<Unit, Actor_TransferRequest, Actor_TransferResponse> {

    protected override async ETTask Run(Unit unit, Actor_TransferRequest message, Action<Actor_TransferResponse> reply) {
        Actor_TransferResponse response = new Actor_TransferResponse();
        try {
            reply(response);
        }
        catch (Exception e) {
            ReplyError(response, e, reply);
        }
    }
}
```

- 我们需要注意一下，Actor 消息有死锁的可能，比如 A call 消息给 B，B call 给 C，C call 给 A。因为 MailboxComponent 本质上是一个消息队列，它开启了一个协程会一个一个消息处理，返回 ETTask 表示这个消息处理类会阻塞 MailboxComponent 队列的其它消息。所以如果出现死锁，我们就不希望某个消息处理阻塞掉 MailboxComponent 其它消息的处理，我们可以在消息处理类里面新开一个协程来处理就行了。例如:

```
[ActorMessageHandler(AppType.Map)]
public class Actor_TestHandler : AMActorHandler<Unit, Actor_Test> {

    protected override ETTask Run(Unit unit, Actor_Test message) {
        RunAsync(unit, message).Coroutine();
    }

    public ETVoid RunAsync(Unit unit, Actor_Test message) {
        Log.Debug(message.Info);
    }
}
```

- 我们可以感受到，Actor 机制实质上就是一种消息的托管机制，类似我们经常用到的事件订阅与分发，我们只要把包含完整数据的消息发出去，不用管谁会接收消息，我们的代码根据 ID 找到相应的接受者。

27 ACTOR LOCATION

- Actor 模型只需要知道对方的 InstanceId 就能发送消息，十分方便，但是有时候我们可能无法知道对方的 InstanceId，或者是一个 Actor 的 InstanceId 会发生变化。这种场景很常见，比如:
  - 很多游戏是分线的，一个玩家可能从 1 线换到 2 线，
  - 还有的游戏是分场景的，一个场景一个进程，玩家从场景 1 进入到场景 2。
- 因为做了进程迁移，玩家对象的 InstanceId 也就变化了。
- ET 提供了给这类对象发送消息的机制，叫做 Actor Location 机制。其原理比较简单:
  - 1. 因为 InstanceId 是变化的，对象的 Entity.Id 是不变的，所以我们首先可以想到使用 Entity.Id 来发送 actor 消息
  - 2. 提供一个位置进程 (Location Server)，Actor 对象可以将自己的 Entity.Id 跟 InstanceId 作为 kv 存到位置进程中。发送 Actor 消息前先去位置进程查询到 Actor 对象的 InstanceId 再发送 actor 消息。

  - 3.Actor 对象在一个进程创建时或者迁移到一个新的进程时，都需要把自己的 Id 跟 InstanceId 注册到 Location Server 上去
  - 4. 因为 Actor 对象是可以迁移的，消息发过去有可能 Actor 已经迁移到其它进程上去了，所以发送 Actor Location 消息需要提供一种可靠机制
  - 5.ActorLocationSender 提供两种方法，Send 跟 Call，Send 一个消息也需要接受者返回一个消息，只有收到返回消息才会发送下一个消息。
  - 6.Actor 对象如果迁移走了，这时会返回 Actor 不存在的错误，发送者收到这个错误会等待 1 秒，然后重新去获取 Actor 的 InstanceId，然后重新发送，目前会尝试 5 次，5 次过后，抛出异常，报告错误
  - 7.ActorLocationSender 发送消息不会每次都去查询 Location Server，因为对象迁移毕竟比较少见，只有第一次去查询，之后缓存 InstanceId，以后发送失败再重新查询。
  - 8.Actor 对象在迁移过程中，有可能其它进程发送过来消息，这时会发生错误，所以 location server 提供了一种 Lock 的机制。对象在传送前，删掉在本进程的信息，然后在 location server 上加上锁，一旦锁上后，其它的对该 key 的请求会进行队列。
  - 9. 传送前因为对方删除了本进程的 actor，所以其它进程会发送失败，这时候他们会进行重试。重试的时候会重新请求 location server，这时候会发现被锁了，于是一直等待
  - 10. 传送完成后，要 unlock location server 上的锁，并且更新新的地址，然后响应其它的 location 请求。其它发给这个 actor 的请求继续进行下去。

- 注意，Actor 模型是纯粹的服务端消息通信机制，跟客户端是没什么关系的，很多用 ET 的新人看到 ET 客户端消息也有 Actor 接口，以为这是客户端跟服务端通信的机制，其实不是的。ET 客户端使用这个 Actor 完全是因为 Gate 需要对客户端消息进行转发，我们可以正好利用服务端 actor 模型来进行转发，所以客户端有些消息也是继承了 actor 的接口。假如我们客户端不使用 actor 接口会怎么样呢？比如，Frame_ClickMap 这个消息

```
message Frame_ClickMap // IActorLocationMessage {
^^Iint64 ActorId = 93;
^^Iint64 Id = 94;

^^Ifloat X = 1;
^^Ifloat Y = 2;
^^Ifloat Z = 3;
}
```

- 我们可能就不需要 ActorId 这个字段，消息发送到 Gate，gate 看到是 Frame_ClickMap 消息，它需要转发给 Map 上的 Unit，转发还好办，gate 可以从 session 中获取对应的 map 的 unit 的位置，然后转发，问题来了，Frame_ClickMap 消息到了 map，map 怎么知道消息需要给哪个对象呢？这时候有几种设计：
  - 1. 在转发的底层协议中带上 unit 的 Id，需要比较复杂的底层协议支持。
  - 2. 用一个消息对 Frame_ClickMap 消息包装一下，包装的消息带上 Unit 的 Id，用消息包装意味着更大的消耗，增加 GC。个人感觉这两种都很差，不好用，而且就算分发给 unit 对象处理了，怎么解决消息重入的问题呢？unit 对象仍然需要挂上一个消息处理队列，然后收到消息扔到队列里面。这不跟 actor 模型重复了吗？目前 ET 在客户端发给 unit 的消息做了个设计，消息做成 actor 消息，gate 收到发现是 actor 消息，直接发到对应的 actor 上，解决的可以说很漂亮。其实客户端仍然是使用 session.send 跟 call 发送消息，发送的时候也不知道消息是 actor 消息，只有到了 gate，gate 才进行了判断，参考 OuterMessageDispatcher.cs

28 ACTOR LOCATION 消息处理

- ActorLocation 消息发送

```
// 从 Game.Scene 上获取 ActorLocationSenderComponent，然后通过 Entity.Id 获取 ActorLocationSender
ActorLocationSender actorLocationSender = Game.Scene.GetComponent<ActorLocationSenderComponent>().Get(unitId);
// 通过 ActorLocationSender 来发送消息
actorLocationSender.Send(actorLocationMessage);
// 发送 Rpc 消息
IResponse response = await actorLocationSender.Call(actorLocationRequest);
```

- ActorLocation 消息的处理跟 Actor 消息几乎一样，不同的是继承的两个抽象类不同，注意 actorlocation 的抽象类多了个 Location

```
// 处理 send 过来的消息，需要继承 AMActorLocationHandler 抽象类，抽象类第一个泛型参数是 Actor 的类型，第二个参数是消息的类型
[ActorMessageHandler(AppType.Map)]
public class Frame_ClickMapHandler : AMActorLocationHandler<Unit, Frame_ClickMap> {

    protected override ETTask Run(Unit unit, Frame_ClickMap message) {
        Vector3 target = new Vector3(message.X, message.Y, message.Z);
        unit.GetComponent<UnitPathComponent>().MoveTo(target).Coroutine();
    }
}
// 处理 Rpc 消息，需要继承 AMActorRpcHandler 抽象类，抽象类第一个泛型参数是 Actor 的类型，第二个参数是消息的类型，第三个参数是返回
[ActorMessageHandler(AppType.Map)]
public class C2M_TestActorRequestHandler : AMActorLocationRpcHandler<Unit, C2M_TestActorRequest, M2C_TestActorResponse> {

    protected override async ETTask Run(Unit unit, C2M_TestActorRequest message, Action<M2C_TestActorResponse> reply) {
        reply(new M2C_TestActorResponse(){Info = "actor rpc response"});
        await ETTask.CompletedTask;
    }
}
```

29 WebSocket 服务器的大致步骤

```
// ET 不一样的是在接收到 websocket 消息后 , 调用一个 OnAccept 方法 创建 session 组件 然后调用 start 方法
public void OnAccept(AChannel channel) {
    Session session = ComponentFactory.CreateWithParent<Session, AChannel>(this, channel);
    this.sessions.Add(session.Id, session);
    session.Start();
}

// start 方法调用 WChannel 的 StartRecv() StartSend() 方法
public async ETVoid StartRecv() {
    if (this.IsDisposed) {
        return;
    }
    try {
        while (true) {
#if SERVER
            ValueWebSocketReceiveResult receiveResult;
#else
            WebSocketReceiveResult receiveResult;
#endif
            int receiveCount = 0;
            do {
#if SERVER
                receiveResult = await this.webSocket.ReceiveAsync(
                    new Memory<byte>(this.recvStream.GetBuffer(), receiveCount, this.recvStream.Capacity - receiveCount),
                    cancellationTokenSource.Token);
#else
                receiveResult = await this.webSocket.ReceiveAsync(
                    new ArraySegment<byte>(this.recvStream.GetBuffer(), receiveCount, this.recvStream.Capacity - receiveCou
                    cancellationTokenSource.Token);
#endif
                if (this.IsDisposed) {
                    return;
                }
                receiveCount += receiveResult.Count;
            }
            while (!receiveResult.EndOfMessage);
            if (receiveResult.MessageType == WebSocketMessageType.Close) {
                this.OnError(ErrorCode.ERR_WebsocketPeerReset);
                return;
            }
            if (receiveResult.Count > ushort.MaxValue) {
```

```
                await this.webSocket.CloseAsync(WebSocketCloseStatus.MessageTooBig, $"message too big: {receiveResult.Count
                                                cancellationTokenSource.Token);
                this.OnError(ErrorCode.ERR_WebsocketMessageTooBig);
                return;
            }
            this.recvStream.SetLength(receiveResult.Count);
            this.OnRead(this.recvStream);
        }
    }
    catch (Exception e) {
        Log.Error(e);
        this.OnError(ErrorCode.ERR_WebsocketRecvError);
    }
}

// 从 message 字节流中接受消息 交由 Session.OnRead 方法处理  onread 方法如下
// 包的前 2 个字节是 opcode 编码 后面的是消息体
private void Run(MemoryStream memoryStream) {
    memoryStream.Seek(Packet.MessageIndex, SeekOrigin.Begin);
    ushort opcode = BitConverter.ToUInt16(memoryStream.GetBuffer(), Packet.OpcodeIndex);

#if !SERVER
    if (OpcodeHelper.IsClientHotfixMessage(opcode)) {
        this.GetComponent<SessionCallbackComponent>().MessageCallback.Invoke(this, opcode, memoryStream);
        return;
    }
#endif
    object message;
    try {
        OpcodeTypeComponent opcodeTypeComponent = this.Network.Entity.GetComponent<OpcodeTypeComponent>();
        object instance = opcodeTypeComponent.GetInstance(opcode);
        message = this.Network.MessagePacker.DeserializeFrom(instance, memoryStream);

        if (OpcodeHelper.IsNeedDebugLogMessage(opcode)) {
            Log.Msg(message);
        }
    }
    catch (Exception e) {
        // 出现任何消息解析异常都要断开 Session, 防止客户端伪造消息
        Log.Error($"opcode: {opcode} {this.Network.Count} {e}, ip: {this.RemoteAddress}");
        this.Error = ErrorCode.ERR_PacketParserError;
        this.Network.Remove(this.Id);
        return;
    }
    if (!(message is IResponse response)) {
        this.Network.MessageDispatcher.Dispatch(this, opcode, message);
        return;
    }

    Action<IResponse> action;
    if (!this.requestCallback.TryGetValue(response.RpcId, out action)) {
        throw new Exception($"not found rpc, response message: {StringHelper.MessageToStr(response)}");
    }
    this.requestCallback.Remove(response.RpcId);
    action(response);
}

// 如上代码所示, 解析出 opcode 后 从 OpcodeTypeComponent 组件中找到 opcode 对应的类型, 把消息体反序列化成 opcode 对应的请求或者相
// 如果非 IResponse 消息的话 会把消息转给 InnerMessageDispatcher 类进行处理
public void Dispatch(Session session, ushort opcode, object message) {
    // 收到 actor 消息, 放入 actor 队列
    switch (message) {
        case IActorRequest iActorRequest: {
            Entity entity = (Entity)Game.EventSystem.Get(iActorRequest.ActorId);
            if (entity == null) {
                Log.Warning($"not found actor: {message}");
                ActorResponse response = new ActorResponse {
                    Error = ErrorCode.ERR_NotFoundActor,
                    RpcId = iActorRequest.RpcId
                };
                session.Reply(response);
                return;
            }

            MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
            if (mailBoxComponent == null) {
                ActorResponse response = new ActorResponse {
                    Error = ErrorCode.ERR_ActorNoMailBoxComponent,
                    RpcId = iActorRequest.RpcId
```

```
                };
                session.Reply(response);
                Log.Error($"actor not add MailBoxComponent: {entity.GetType().Name} {message}");
                return;
            }

            mailBoxComponent.Add(new ActorMessageInfo() { Session = session, Message = iActorRequest });
            return;
        }
        case IActorMessage iactorMessage: {
            Entity entity = (Entity)Game.EventSystem.Get(iactorMessage.ActorId);
            if (entity == null) {
                Log.Error($"not found actor: {message}");
                return;
            }

            MailBoxComponent mailBoxComponent = entity.GetComponent<MailBoxComponent>();
            if (mailBoxComponent == null) {
                Log.Error($"actor not add MailBoxComponent: {entity.GetType().Name} {message}");
                return;
            }

            mailBoxComponent.Add(new ActorMessageInfo() { Session = session, Message = iactorMessage });
            return;
        }
        default: {
            Game.Scene.GetComponent<MessageDispatcherComponent>().Handle(session, new MessageInfo(opcode, message));
            break;
        }
    }
}
// 如上代码 消息类型有 2 种  IActorRequest   IActorMessage
// 然后根据 ActorId 获取消息处理类
// 这个 ActorId 是啥时候加到 eventsystem 中的?
// Game.EventSystem.Get(iActorRequest.ActorId);
// 然后交由 MailBoxComponent.Add 方法 把消息放到队列中
```

30 我们终于走完了消息创建-打包-发送-接收-解包-分发到相对应处理器处理的整个流程。

- 如果我们要自定义一个消息，怎么做呢?
  - 确定这个消息是否是请求（需要回复）。如果需要回复，则实现 IRequest 接口并且定义自定义回复结构，继承 IResponse。记得添加 Message 特性并且标注操作符，新的操作符可以添加到自定义枚举里。或者添加到 InnerOpcode、Opcode、OuterOpcode 里面。还得添加 ProtoContract，这是 Protobuf-net 的用法。
  - 定义消息结构，既消息的内容。消息的内容有必要的话得按照 Protobuf-net 的用法定义特性。
  - 定义处理消息的类。根据消息类型继承 AMRpcHandler 或者 AMHandler，并把消息类型当作泛型传入。处理消息的类需要添加 MessageHandler 特性并且标注 AppType。
  - 具体的处理消息方法需要重写处理类的 Run 方法。
- 经过上面的步骤，我们的 MessageDispatherComponent 跟 OpcodeTypeComponent 就会识别这些类，并注册相应的事件。当我们接收到相应的消息时，也会正确分发给对应的处理器处理
- 网络请求大致步骤过程，以及底层网络请求数据的包装等（ProtoBuf 协议）讲得比较彻底详细，再看一遍https://blog.csdn.net/Tong1993222/article/details/88779223?spm=1001.2101.3001.6650.12&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-12-88779223-blog-86600357.pc_relevant_multi_platformwhitelistv4&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-12-88779223-blog-86600357.pc_relevant_multi_platformwhitelistv4&utm_relevant_index=13

31 unity 篇-官方序列化接口 ISerializationCallbackReceiver

- 这是过程中狠小的一个知识点，很简单，只作下笔记

31.1 官方提供的解决方案

- 编写一个类继承 ISerializationCallbackReceiver 接口，通过编写 2 个回调让 List 类型代替 Dictionary 参与序列化

```
using UnityEngine;
using System;
using System.Collections.Generic;

public class SerializationCallbackScript : MonoBehaviour, ISerializationCallbackReceiver {

    public List<int> _keys = new List<int> { 3, 4, 5 };
    public List<string> _values = new List<string> { "I", "Love", "Unity" };

    // Unity doesn't know how to serialize a Dictionary
    public Dictionary<int, string>  _myDictionary = new Dictionary<int, string>();

    public void OnBeforeSerialize() {
        _keys.Clear();
        _values.Clear();
        foreach (var kvp in _myDictionary) {
            _keys.Add(kvp.Key);
            _values.Add(kvp.Value);
        }
    }

    public void OnAfterDeserialize() {
        _myDictionary = new Dictionary<int, string>();
        for (int i = 0; i != Math.Min(_keys.Count, _values.Count); i++)
            _myDictionary.Add(_keys[i], _values[i]);
    }

    void OnGUI() {
        foreach (var kvp in _myDictionary)
            GUILayout.Label("Key: " + kvp.Key + " value: " + kvp.Value);
    }
}
```

31.2 Dictionary 的序列化的泛型解决方案

- 但我们在 unity 使用 Dictionary 实在太频繁，不可能为每个类继承接口编写回调，一位牛人使用泛型编程为我们解决了这个问题

```
public class SerializationDictionary<TKey, TValue> : ISerializationCallbackReceiver {

    [SerializeField]
    private List<TKey> keys;

    [SerializeField]
    private List<TValue> values;

    private Dictionary<TKey, TValue> target;
    public Dictionary<TKey, TValue> ToDictionary() { return target; }

    public SerializationDictionary(Dictionary<TKey, TValue> target) {
        this.target = target;
    }

    public void OnBeforeSerialize() {
        keys = new List<TKey>(target.Keys);
        values = new List<TValue>(target.Values);
    }

    public void OnAfterDeserialize() {
        var count = Math.Min(keys.Count, values.Count);
        target = new Dictionary<TKey, TValue>(count);
        for (var i = 0; i < count; ++i) {
            target.Add(keys[i], values[i]);
```

```
        }
    }
}
```

- **泛型实在太有魅力了，我等懒人必会之**爱表哥，爱生活!!! 一定会嫁给偶亲爱的表哥!!!