

unity 游戏热更新服务端服务器

deepwaterooo

January 24, 2023

Contents

1 笔记	1
2 综述	1
3 帐户服 + 数据库 + 登录中心服 + 网关服: 具体设计林框架参考	1
3.1 一. 账号登录	2
3.1.1 1. 客户端请求获取账户信息	2
3.1.2 2. 客户端请求获取区服信息	3
3.1.3 3. 客户端请求获取/创建/删除角色信息	3
3.2 二. 网关服务器的连接	5
3.2.1 1. 请求连接 Realm 网关	5
3.2.2 2. 请求和 Gate 网关连接	5
3.2.3 3. 建立 Gate 映射对象 Player	6
3.2.4 建立 Player 步骤	6
3.3 三. 游戏逻辑服务器连接	6
3.4 四. 顶号逻辑流程图	7
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	8
5 Bson 序列化与反序列化: Core/MongoHelper.cs	8
6 Init.cs: 程序真正的入口	9
7 Game.cs: 这个类会涉及到一些生命周期的管理等	10
8 CodeLoader.cs: 加载热更新等各种程序集	11
9 Entry.cs: Assets/Scripts/Codes/Model/Share/Entry.cs 不是真正的入口	12
10	13

1 笔记

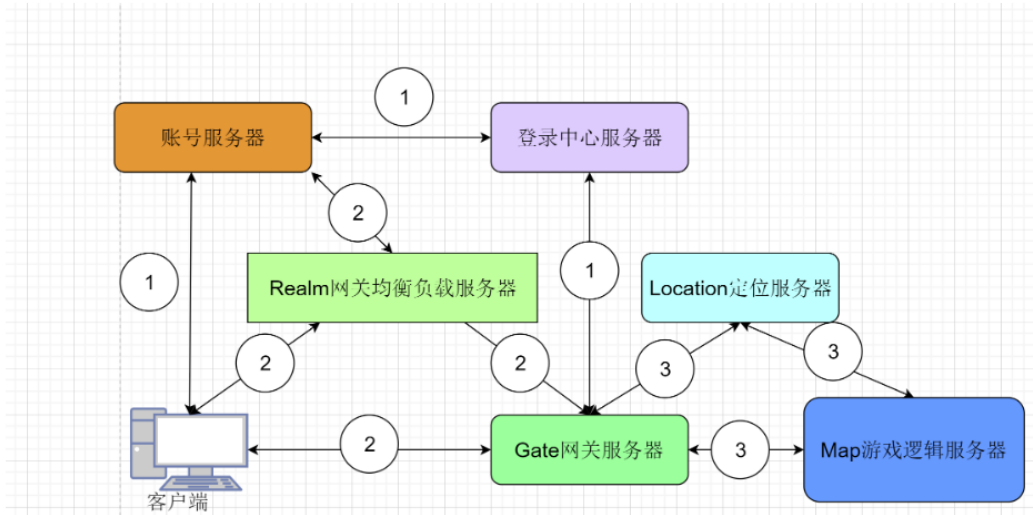
2 综述

- 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。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

- 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>

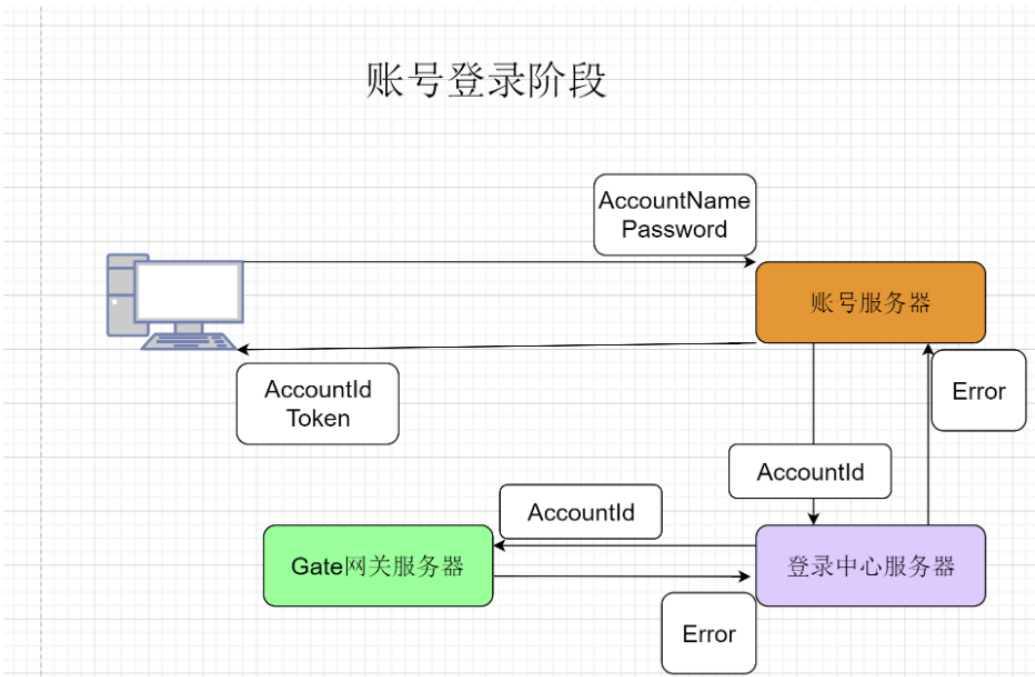
登陆系统全步骤图文



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

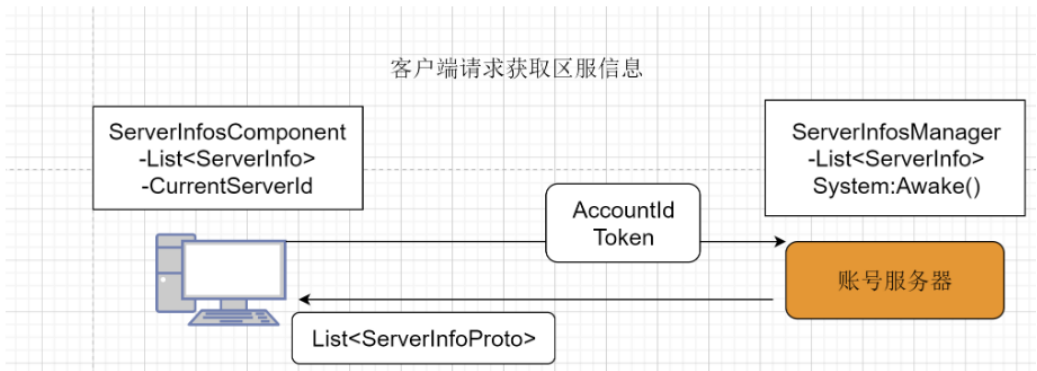


- 客户端向账号服务器发送账户名和密码的消息，请求进行登录
- 账号服务器和数据库交互，对信息进行验证或注册，获取账户唯一标识 AccountId

- 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 AccountId, 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 sessionId,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

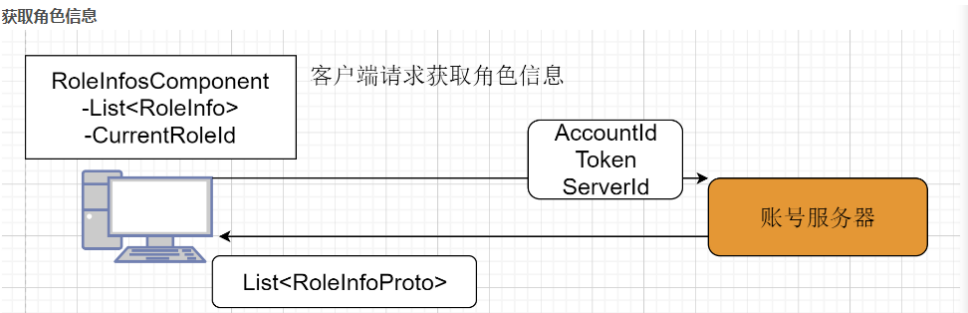
2. 客户端请求获取区服信息



- 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。

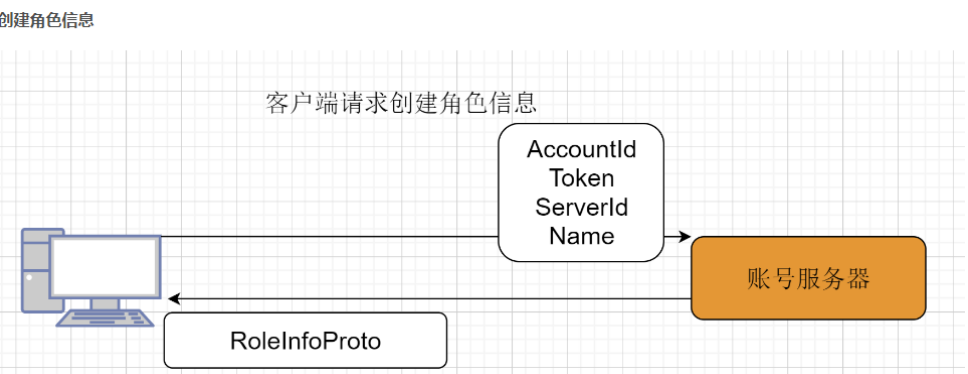
3.1.3 3. 客户端请求获取/创建/删除角色信息

1. 获取角色信息



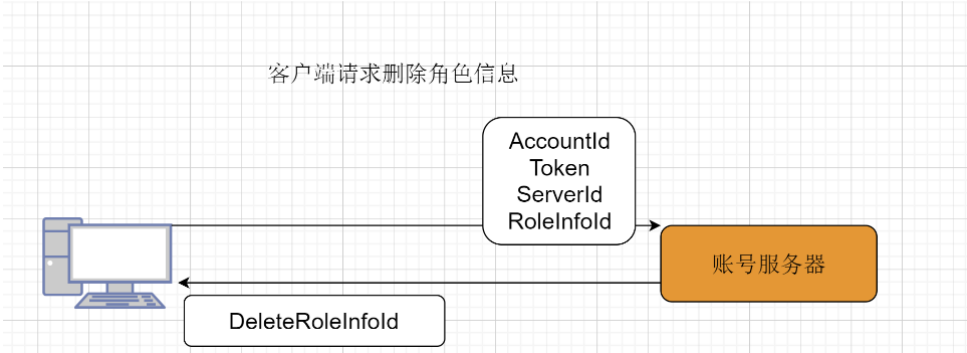
- 获取角色信息的步骤和获取服务器信息很类似
 - 定义 RoleInfo 实体 (Name,AccountId,State,ServerId)，并为其提供和 proto 的相互转换行为
 - 实体是双端可以公用的，但账号服务器无需保存，只需在请求时从数据库获取完发回即可
 - 客户端向服务器发送消息请求获取当前账户在当前区服下的所有未冻结的角色
 - 服务器通过 Id 和 Token 验证身份后，从数据库中获取所有角色信息转换成 proto 对象发回客户端
 - 客户端收到后将 proto 转换成 RoleInfo 存储在相应的组件中，显示到 UI 层供用户选择。

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

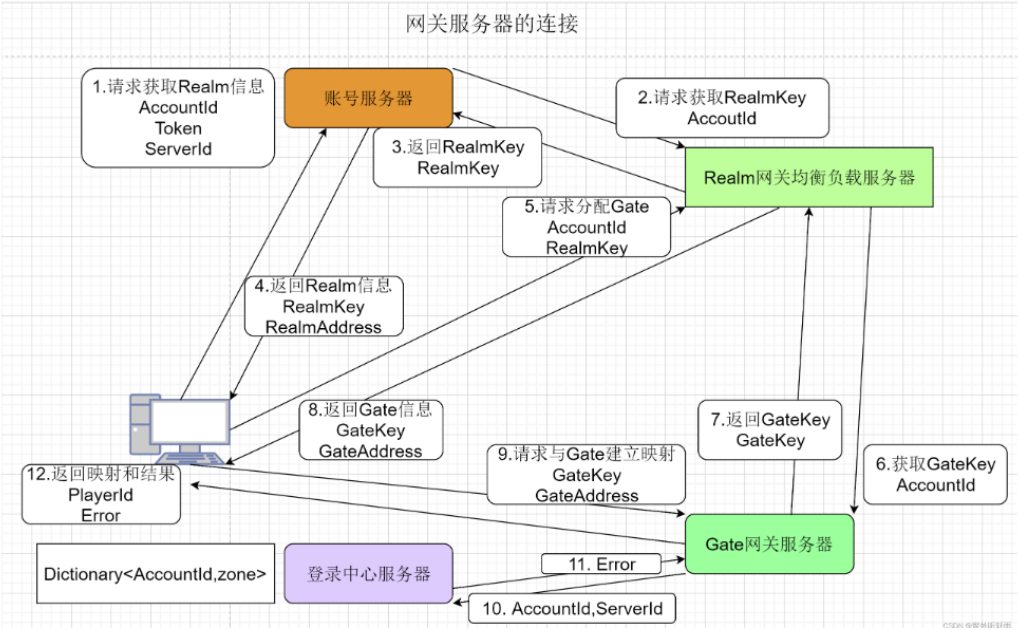
3. 删除角色信息



- 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)
- 一个区服下一般会有多个 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

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。

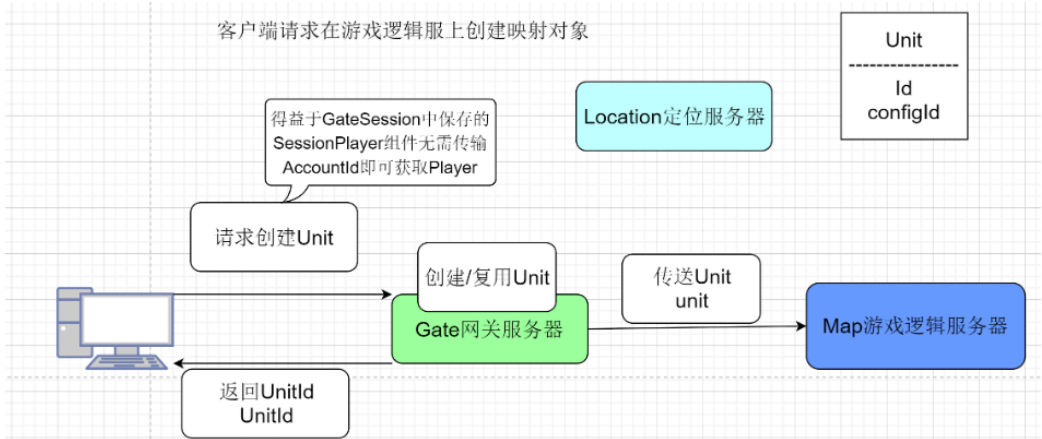
3.2.4 建立 Player 步骤

- 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。

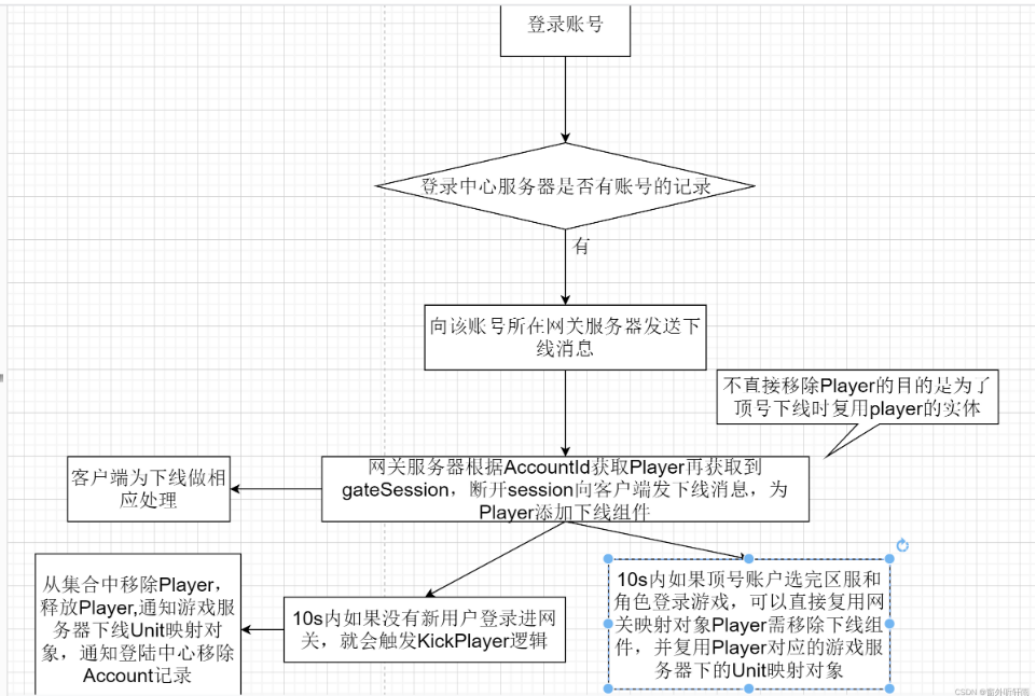
3.3 三. 游戏逻辑服务器连接

- 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 Unit-Component 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，很容易就看一次忘记一次，所以记好笔记很重要
- 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

```
Win32Exception: The system cannot find the file specified.
Win32Exception: The system cannot find the file specified.

System.Diagnostics.Process.StartWithShellExecuteEx (System.Diagnostics.ProcessStartInfo startInfo) (at <7224ac32e19348db9f4fb2ca83958d37>.0)
System.Diagnostics.Process.Start () (at <7224ac32e19348db9f4fb2ca83958d37>.0)
(wrapper remoting-invoke-with-check) System.Diagnostics.Process.Start()
System.Diagnostics.Process.Start (System.Diagnostics.ProcessStartInfo startInfo) (at <7224ac32e19348db9f4fb2ca83958d37>.0)
ET.ProcessHelper.Run (System.String exe, System.String arguments, System.String workingDirectory, System.Boolean waitExit) (at Assets/Scripts/Core/Helper/ProcessHelper.cs:44)
Rethrow as Exception: dir: F:\ET\Database\bin/, command: mongod --dbpath=db
ET.ProcessHelper.Run (System.String exe, System.String arguments, System.String workingDirectory, System.Boolean waitExit) (at Assets/Scripts/Core/Helper/ProcessHelper.cs:55)
ET.ServerCommandLineEditor.OnGUI () (at Assets/Scripts/Editor/ServerCommandLineEditor/ServerCommandLineEditor.cs:61)
UnityEditor.HostView.InvokeOnGUI (UnityEngine.Rect onGUIPosition) (at <ece5a5820976427dbcd927836361f740>.0)
UnityEditor.DockArea.DrawView (UnityEngine.Rect dockAreaRect) (at <ece5a5820976427dbcd927836361f740>.0)
UnityEditor.DockArea.OnGUI () (at <ece5a5820976427dbcd927836361f740>.0)
```


5 Bson 序列化与反序列化: Core/MongoHelper.cs

- **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

[illegible]

- 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 Init.cs: 程序真正的人口

```
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(); // <=====
    }
}
```

7 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
        }
    }
}
```


8 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 {
            // 傻瓜 Unity 在这里搞了个傻瓜优化, 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, this.hotfixAssembly);
        EventSystem.Instance.Add(types);
    }
}
```

9 Entry.cs: Assets/Scripts/Codes/Model/Share/Entry.cs 不是真正的人口

```
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 ETask 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());
}
}
}

```

10