腾讯GAD
游戏开发者平台
做有梦想的游戏人!

贪吃蛇大作战 2

Tencent，SlicolTang

0 课程概述

- 0.1 上次课程的简要回顾
- 0.2 部分同学问题的解答
- 0.3 本次课程的主要内容

为什么会有第2季课程？

第1季课程的主要目标是带领大家经历从0开始完成一个可发布的游戏开发所需要经历的关键设计步骤。
而很多同学对于其中一些关键模块（特别是与当前游戏无关基础模块）的实现细节感兴趣。
本季课程每一章将增加【练习】这一环节

0.1 上次课程的简要回顾

课序	标题	主要内容
1	规划游戏功能	归纳了经典贪吃蛇游戏的玩法功能
2	功能模块划分	将功能划分为若干个模块，对这些模块分类分层，梳理依赖关系
3	系统架构设计	设计一套【用来组织、管理、解耦上述功能模块】的模块管理器模块
4	开始创建项目	创建一个Unity工程，设置参数，创建场景和启动类
5	让架构先跑起来	在Unity工程中实现第【3】课的设计，并实现日志系统以观察验证其设计
6	UI框架与MVC模式	设计和实现UI管理器，并对UI进行分类和分层，实现了常用的UI基类
7	主城及相关模块	资源导入，Protobuf引用，假的登录模块和用户管理模块
8	核心玩法	贪吃蛇玩法的设计和实现
9	帧同步	介绍帧同步的基本原理，通讯原理，逻辑原理，整体框架
10	局域网通讯	介绍局域网对战原理，配对方案，以及设计和实现一个简单的RPC框架
11	换肤及特效	适用该游戏的皮肤及特效制作方式
12	游戏打包	分别演示了Android和IOS平台的打包流程

0.2 部分同学问题的解答

- 问题1：
  - 在打开UI时，由于UIRoot没法通过类似“AAA/BBB/CCC”的路径找到已经存在的UI，造成UI重复打开。

- 解答：
  - 在早期Unity版本，Find函数对于“ /”并不会做特别处理，而新版本Unity会将“/”当作对子集的查找。
  - 在第2季课程中，会对系统架构进行一次优化。优化后的系统架构将能更好地应对更多的问题。

0.2 部分同学问题的解答

- 问题2：
  - 模块之间的通讯为什么要用消息，而不是直接调用目标模块的函数？为什么有了消息通讯，还需要事件通讯？

- 解答：
  - 一切都是为了解耦。解耦的目的是，浅显地讲就是，A同学在设计A模块时，不需要依赖B模块的设计接口，只需要A同学与B同学约定好消息，A与B模块可以同时进行设计和编码；深入地讲就是，A模块在没有B模块的情形下，也可以独立运行，以此来规范A同学和B同学在开发时，将相关的功能内聚在各自的模块内。
  - 模块间的事件通讯其实是多余的，是我的试验性设计。在第2季的课程中，被优化掉了

0.2 部分同学问题的解答

- 问题3：
  - 有什么好用的MVC框架？

- 解答：
  - 还是那句话，大道无形，与其去寻找MVC框架，不如让MVC回归设计模式的本质。它只是一种设计思路，因为这种思路可以解决某一类型的问题，于是就形成了固定的模式。
  - 追求形式上的MVC，会让代码变得很臃肿。

0.2 部分同学问题的解答

- 问题4：
  - 帧同步的实现细节

- 解答：
  - 第1季课程的主要目标是：带领大家从0开始开发一个可发布的游戏，所需要经历的关键设计步骤。对于一些技术细节没有过多涉及。
  - 在第2季课程里，将会用一个专题来讲帧同步的前后台实现细节。

0.2 部分同学问题的解答

- 问题5：
  - 为什么要用局域网通讯，而不实现一个真正的服务器？

- 解答：
  - 因为局域网通讯的实现比较简单。而真正的服务器非常复杂，并不是我所擅长的领域。
  - 在第2季课程中，我会尝试着利用我对服务器的理解，设计一个可以运行在外网的服务器程序。并且，该程序可以支持大约5000人同时大区在线。后面的帧同步专题也是基于此服务器架构来实现的。

0.2 部分同学问题的解答

- 问题6：
  - 能不能讲一下热更新相关的内容？

- 解答：
  - 目前很多公司项目的热更新都是用Lua。网络上有很多关于Lua的更热教程。在第2季课程中，将会采用另一个技术来实现热更新。

0.2 部分同学问题的解答

- 问题7：
  - 为什么Debuger类单独放在Debuger.dll里？

- 解答：
  - 如果将Debuger类放在主工程里，在Unity的日志输出窗口里点击日志，会跳入到Debuger类里面的代码中去。但其实，我们真正想跳转的是到调用Debuger的代码。

0.3 本次课程的主要内容

- 第1季的课程侧重点是：游戏开发过程中，从设计到编码，再到发布所需要经历的过程。
- 那么第2季的侧重点是：几个重点模块的设计（优化）和编码

课序	标题	主要内容
1	系统架构优化	对模块框架和UI框架进行优化，以及为什么要优化，并且修复一些BUG
2	通用网络模块	设计和实现可用于真正项目的通讯网络模块，包括后台的设计和实现
3	轻量级服务器框架	实现一个轻量的服务器框架
4	登录模块示例	可以利用网络模块登录真正的服务器，包括后台的设计和实现
5	房间模块示例	参照真正的房间模块进行设计和实现，包括后台的设计和实现
6	帧同步通讯专题	将删掉所有已经完成的代码，从0开始手把手教你完成前后台的编码，并进行真正的联调和优化
7	热更新整体架构	基于ILRuntime的热更新整体架构，包括与模块框架和UI框架的结合

1 系统架构优化

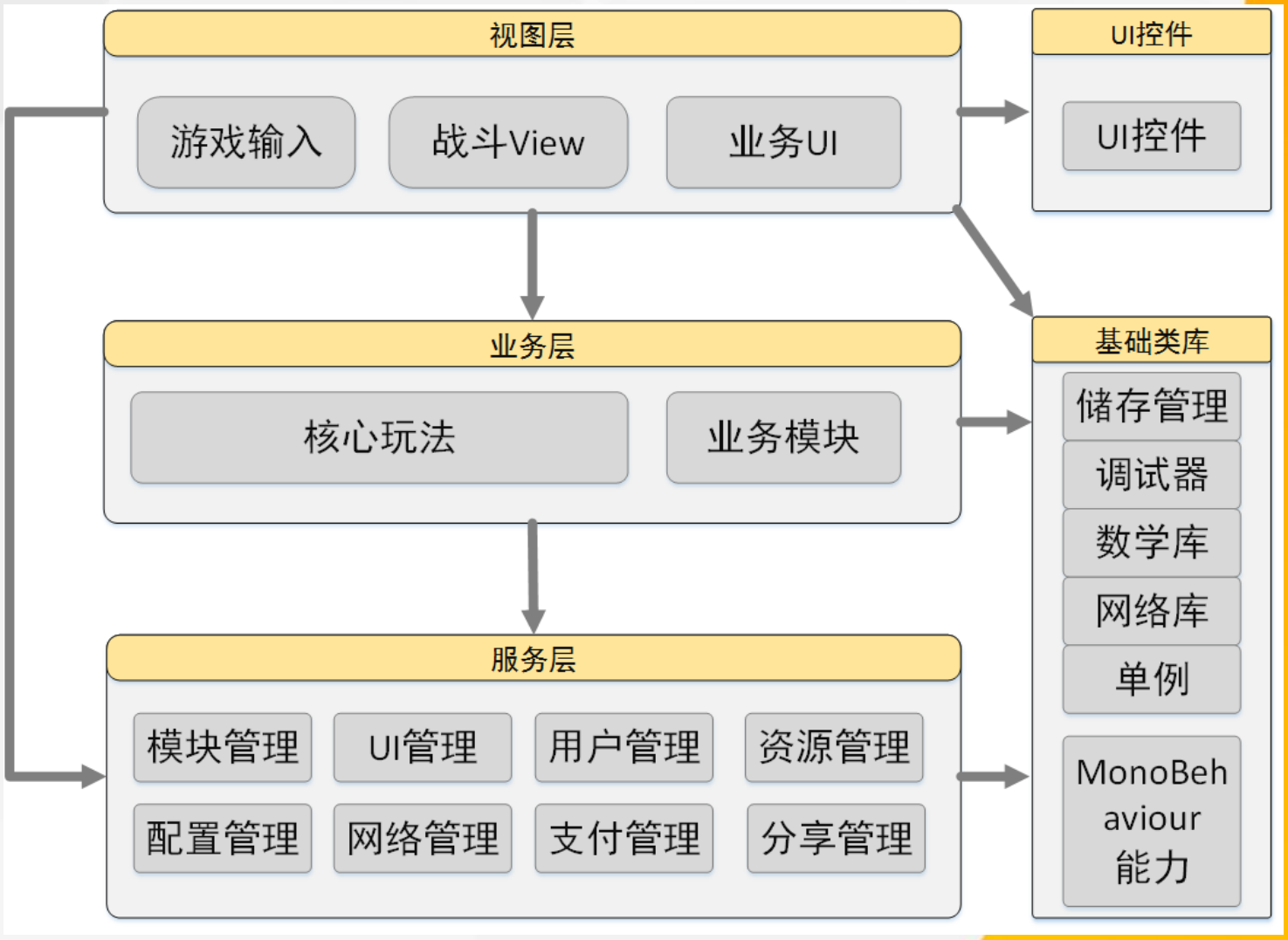
- 1.1 基础框架重构
- 1.2 模块框架重构
- 1.3 UI框架重构
- 1.4 示例及练习

为什么要对【系统架构】进行优化？

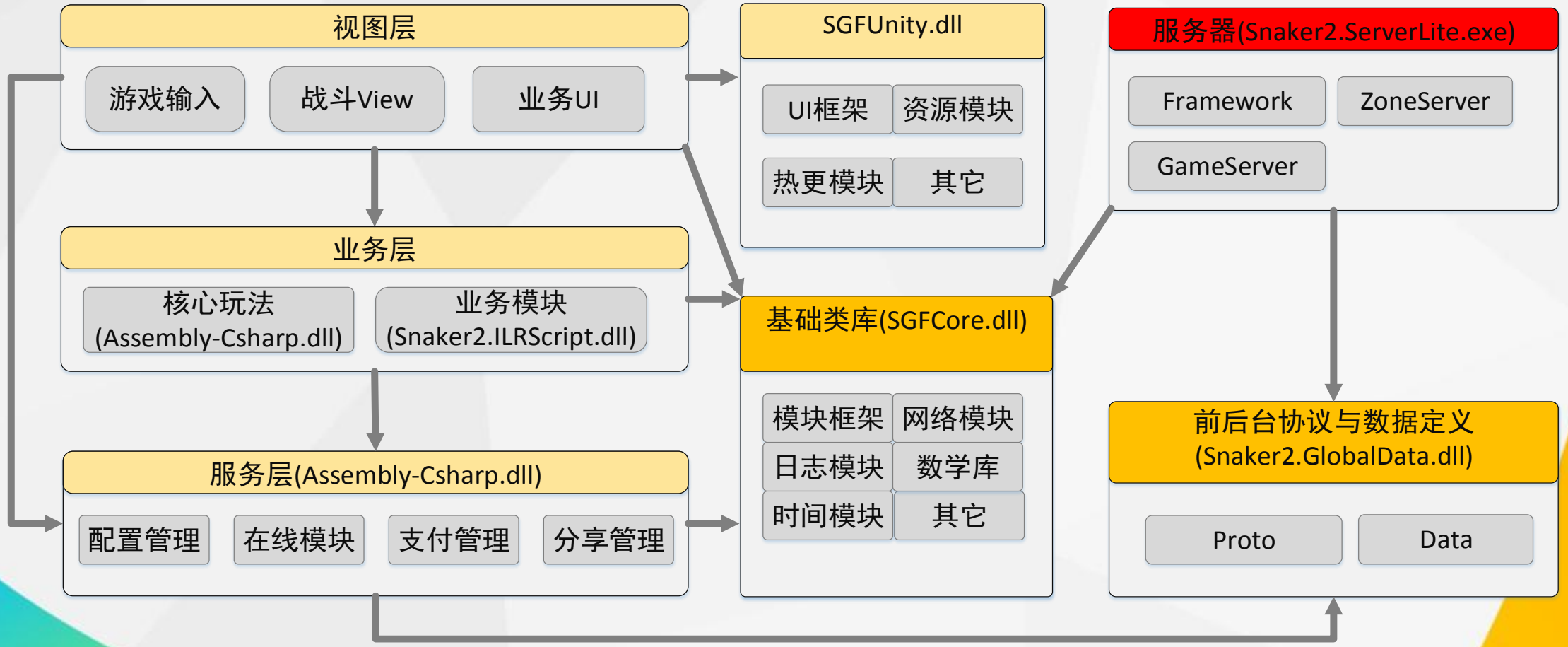
游戏架构是非常复杂、庞大和分层的。一些底层的部分是与具体游戏无关的。正因为如此，我们希望这套架构足够简洁、优雅、通用，而持续重构是达到这一切的唯一方式。

1.1 基础框架重构

- 旧的框架
  - 所有的模块都装在同一个Assembly
  - 模块之间的层次依赖很容易被误操作（新手）打乱
  - 【基础类库】无法前后台重用（也无法直接被新项目重用）



1.1 基础框架重构


视图层
游戏输入
战斗View
业务UI
业务层
核心玩法
(Assembly-Csharp.dll)
业务模块
(Snaker2.ILRScript.dll)
服务层(Assembly-Csharp.dll)
配置管理
在线模块
支付管理
分享管理
SGFUnity.dll
UI框架
资源模块
热更模块
其它
基础类库(SGFCore.dll)
模块框架
网络模块
日志模块
数学库
时间模块
其它
服务器(Snaker2.ServerLite.exe)
Framework
ZoneServer
GameServer
前后台协议与数据定义
(Snaker2.GlobalData.dll)
Proto
Data

1.1 基础框架重构

- SGFCore.dll
  - 与Unity无关的类库，可以前后台重用，也可给新项目使用
  - 主要包括：
    - Console：ConsoleInput（用于在服务器程序中输入）
    - Event：SGFEvent（实现类似UnityEvent的事件模块）
    - Module：（非常弹性的模块框架）
    - Netowrk：General（通用网络模块，包括前后台），FSPLite（帧同步模块）
    - Time：SGFTime（提供类似Unity的Time的功能）
    - Utils：FileUtils，PathUtils，TimeUtils，URLUtils

1.1 基础框架重构

- SGFUnity.dll
  - 与Unity相关的基础类库，也可给新项目使用
  - 主要包括：
    - ILR：（基于ILRScript的热更模块，可以与模块框架无逢结合，对开发者透明）
    - UI：（基于Unity的UI框架，同时提供常用的UI组件，比如MsgBox，Loading）

1.1 基础框架重构

- 架构部署
  - 操作演示

- Snaker2（Unity工程）
  - Assets/Plugins/ManagedLib
    - SGFCore.dll、SGFUnity.dll、SGFDebuger.dll
    - ILRuntime.dll、protobuf-net.dll、GlobalData.dll
  - StreamingAssets/ILR（热更DLL）
    - ILRScript.dll
  - Snaker
    - Game（游戏核心玩法）
    - GlobalUI（公共UI面板）
    - Modules（业务模块）
    - Services（服务模块）
    - AppMain.cs、AppLoading.cs、MoudleDef.cs等等
  - Plugins（VS工程）
    - Snaker2.GlobalData（公共数据工程）
    - Snaker2.ILRScript（热更模块工程）
  - Server
    - Snaker2.ServerLite（服务器主工程）
    - Snaker2.GameServer（战斗服务器工程）
- SGF（VS工程）
  - SGFCore
  - SGFUnity

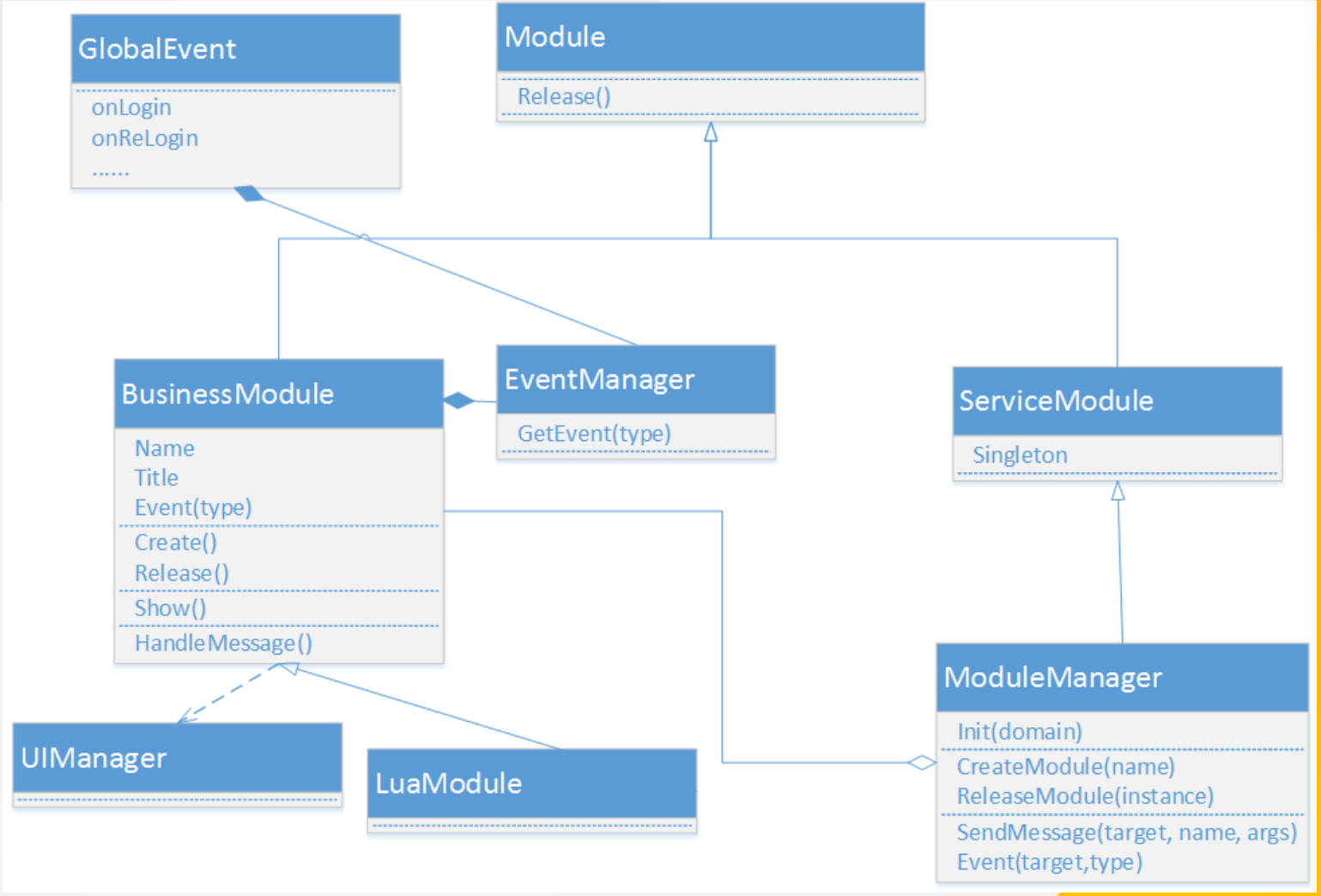
1.1 基础框架重构

- 场景
  - Boot.unity -> Main.unity <-> Game.unity

- 启动代码
  - AppMain.cs
  - ModuleDef.cs
  - AppConfig.cs
  - AppLoading.cs
  - GlobalEvent.cs

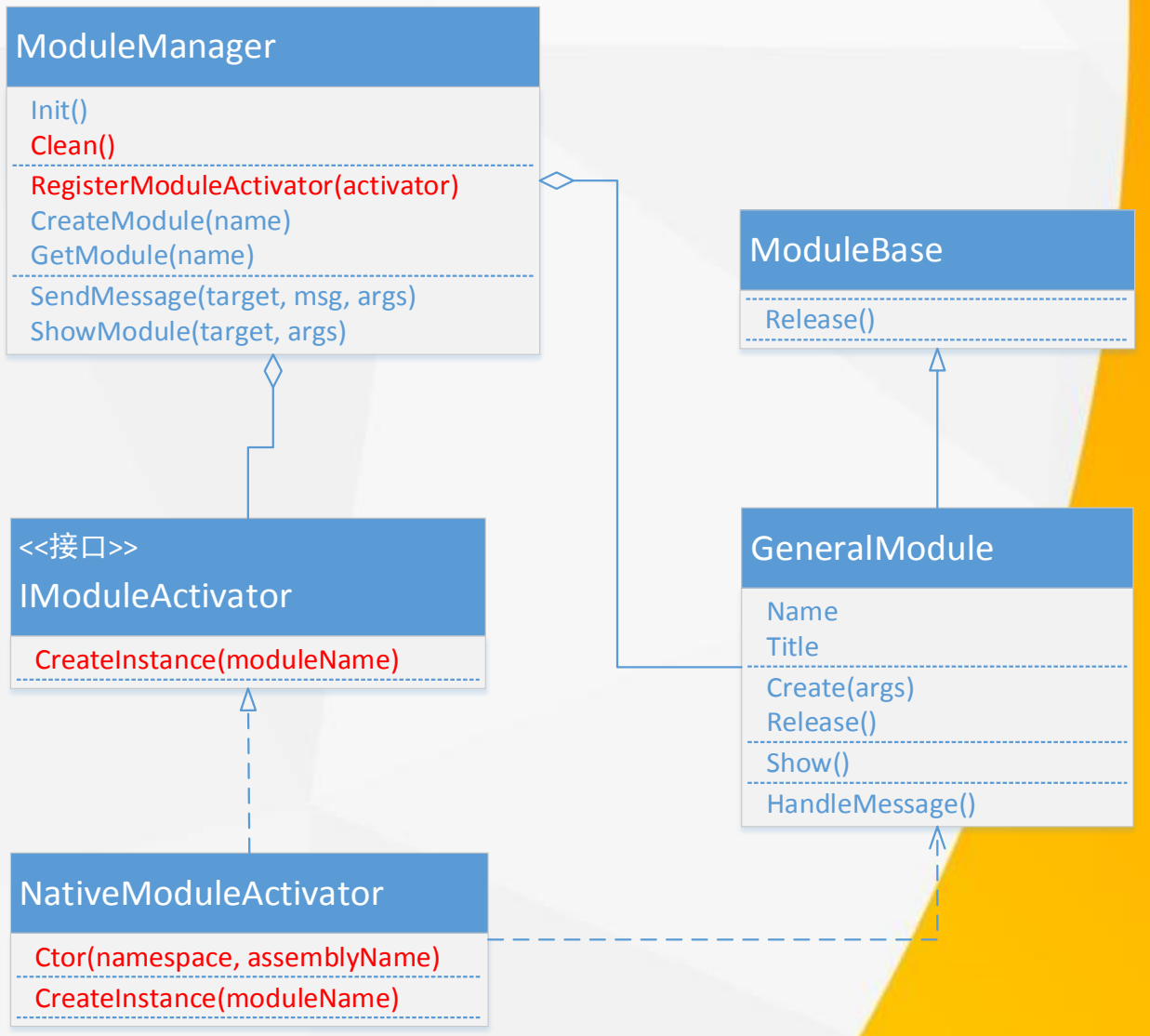
1.2 模块框架重构

- 旧的框架
  - 模块之间不应该存在事件通讯
  - 框架的弹性不够，如果想支持其它类型（非Lua）的模块怎么办？
  - 需要优化一些不用的函数



1.2 模块框架重构

- 新的框架
  - 去掉事件通讯
  - IModuleActivator由外部实现模块创建器
  - NativeModuleActivator用于创建C#原生模块
  - 后续根据热更方案提供ILRModuleActivator（在【热更新整体架构】那一课详细讲述） 或者LuaModuleActivator



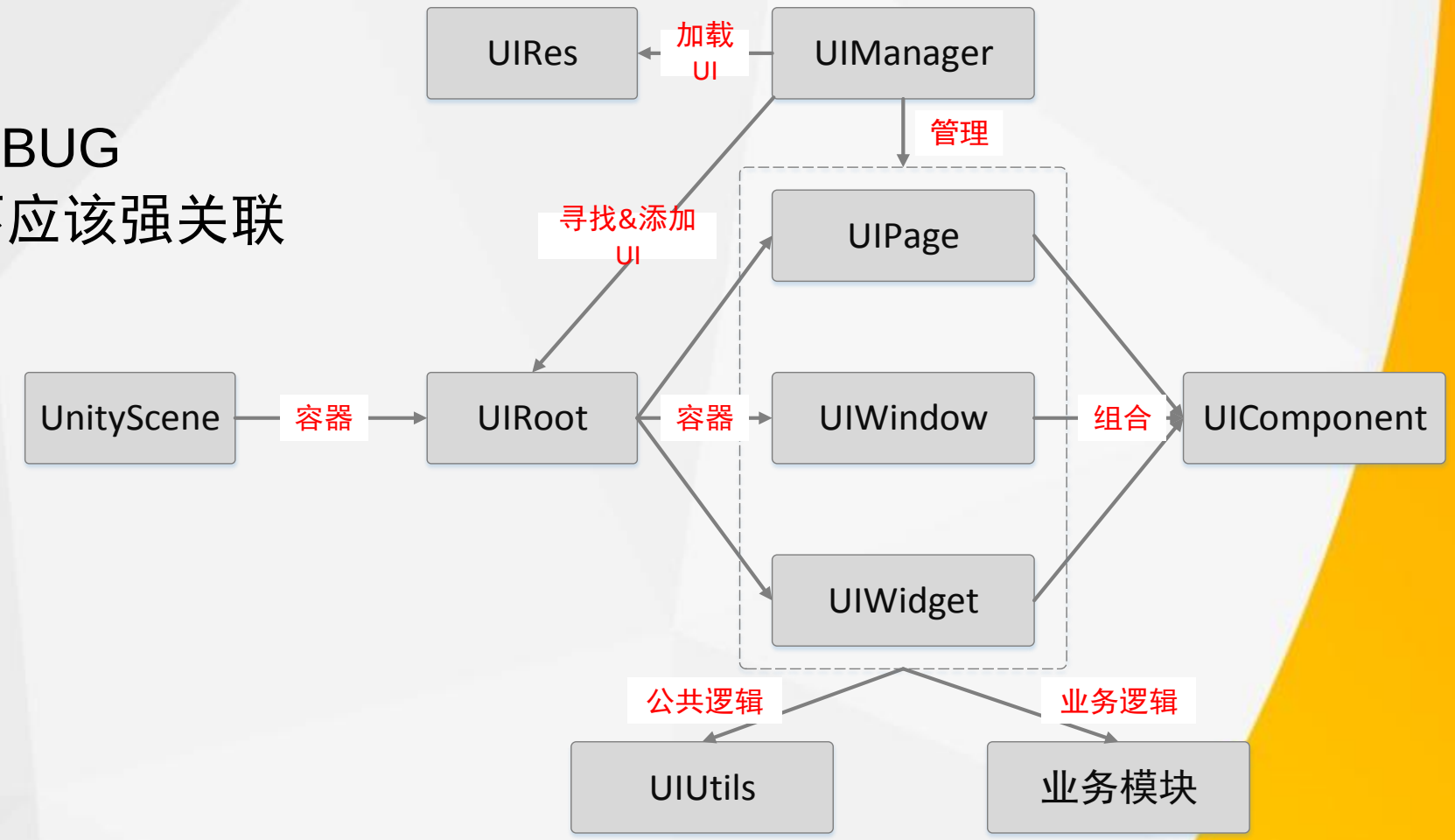
1.2 模块框架重构

- 编码实现
  - ModuleBase.cs
  - GeneralModule.cs
  - IModuleActivator.cs
  - ModuleManager.cs
  - NativeModuleActivator.cs

  - 以上类构成了一个完整的、弹性的模块框架。
  - 考虑到第2季课程的同学有可能没有听过第1季课程，这里再完整演示编码一次。后续涉及到编码实现的内容，都会完整演示编码。

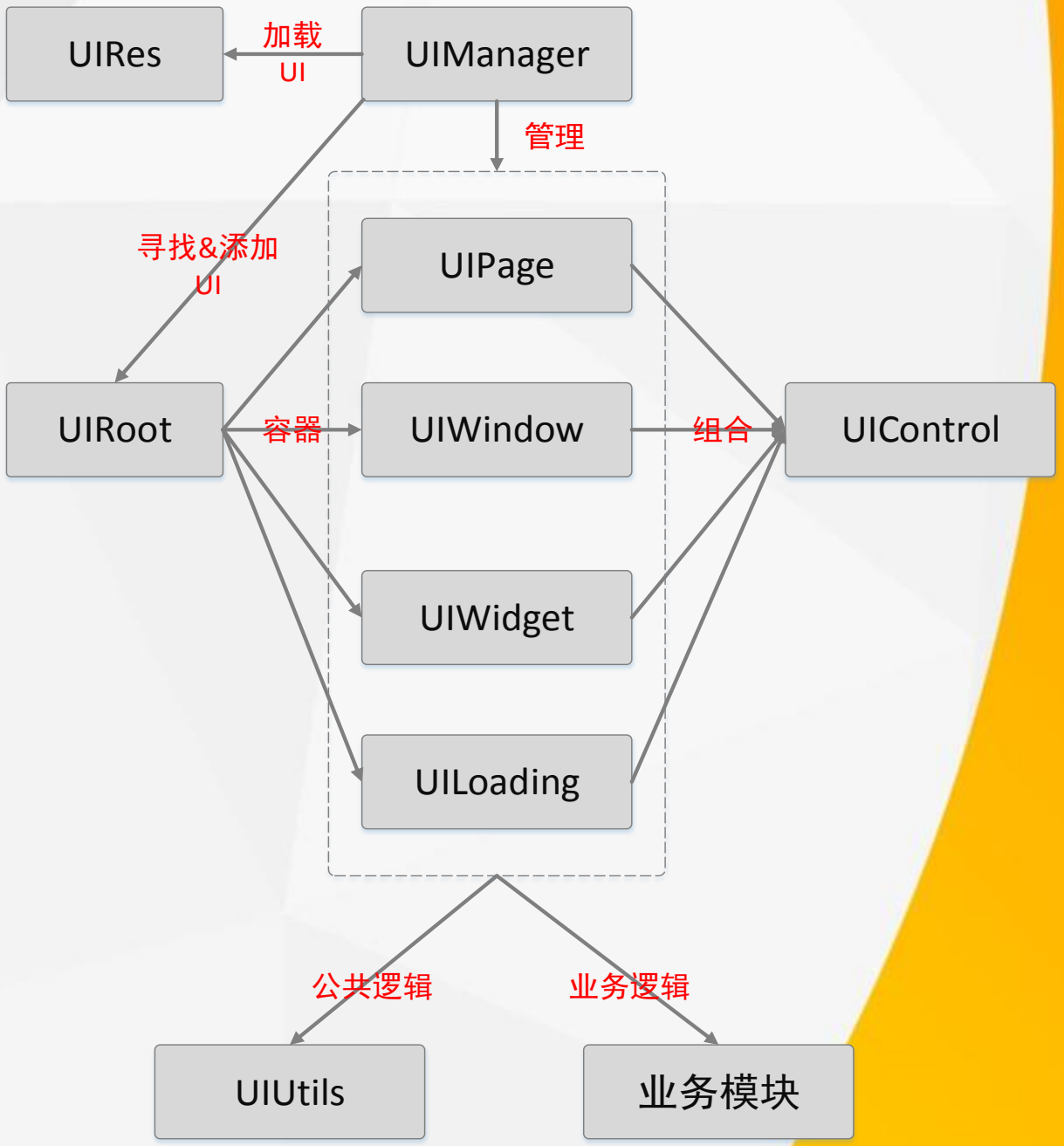
1.3 UI框架重构

- 旧的框架
  - UIRoot存在BUG
  - UI之间的层级存在BUG
  - UI与UnityScene不应该强关联

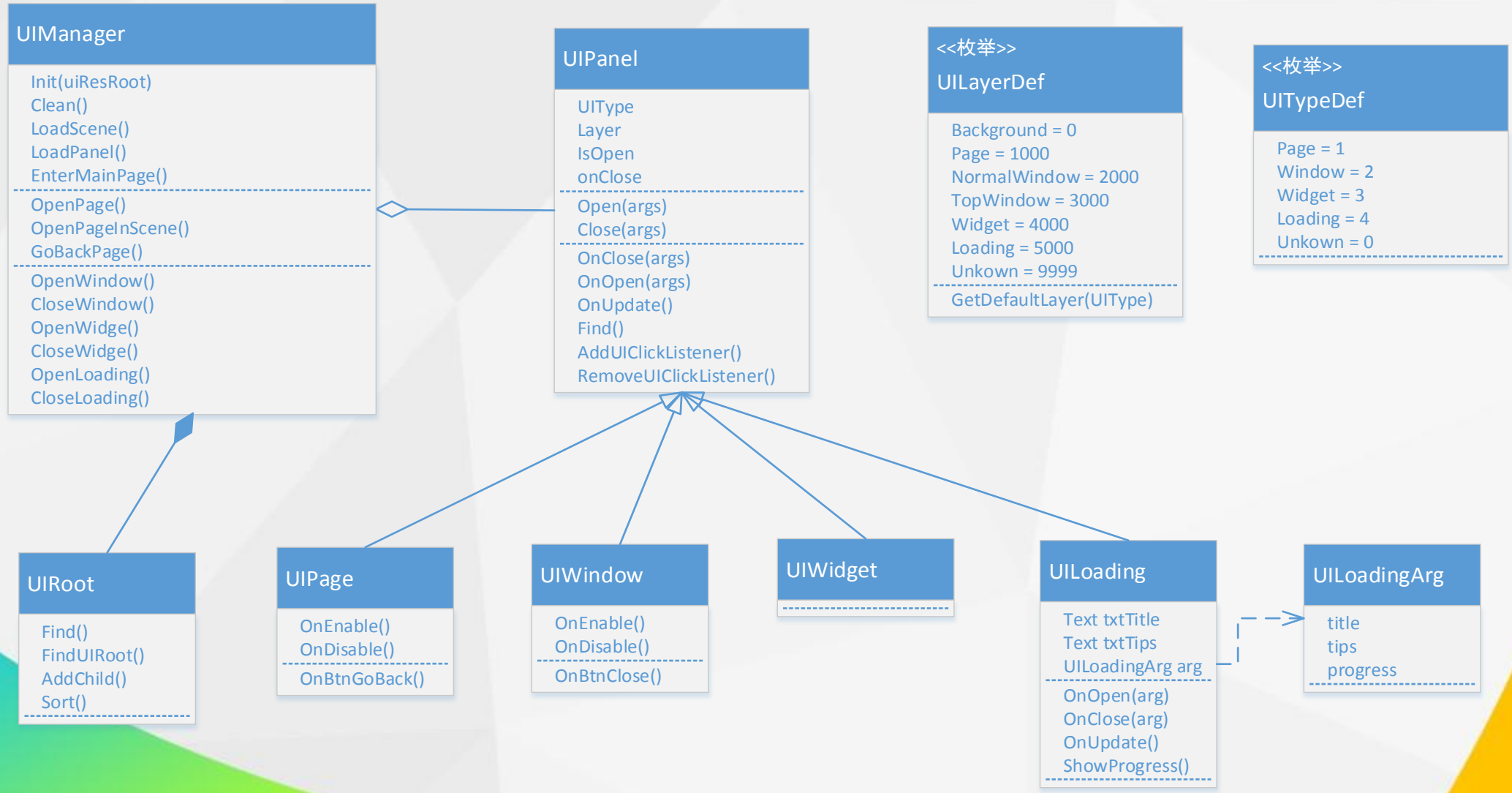


1.3 UI框架重构

- 新的框架
  - 新增UILoading
  - UnityScene切换时显示Loading
  - UI与UnityScene不再强关联（切换UnityScene时，不会强制关闭UI，交给业务层来决定）
  - UIPanel增加动画效果机制



1.3 UI框架重构


UIManager
Init(uiResRoot)
Clean()
LoadScene()
LoadPanel()
EnterMainPage()
OpenPage()
OpenPageInScene()
GoBackPage()
OpenWindow()
CloseWindow()
OpenWidge()
CloseWidge()
OpenLoading()
CloseLoading()
UIPanel
UIType
Layer
IsOpen
onClose
Open(args)
Close(args)
OnClose(args)
OnOpen(args)
OnUpdate()
Find()
AddUIClickListener()
RemoveUIClickListener()
<<枚举>>
UILayerDef
Background = 0
Page = 1000
NormalWindow = 2000
TopWindow = 3000
Widget = 4000
Loading = 5000
Unkown = 9999
GetDefaultLayer(UIType)
<<枚举>>
UITypeDef
Page = 1
Window = 2
Widget = 3
Loading = 4
Unkown = 0
UIRoot
Find()
FindUIRoot()
AddChild()
Sort()
UIPage
OnEnable()
OnDisable()
OnBtnGoBack()
UIWindow
OnEnable()
OnDisable()
OnBtnClose()
UIWidget
UILoading
Text txtTitle
Text txtTips
UILoadingArg arg
OnOpen(arg)
OnClose(arg)
OnUpdate()
ShowProgress()
UILoadingArg
title
tips
progress

1.3 UI框架重构

- 编码实现
  - UIDefine.cs（定义UITypeDef类和UILayerDef类）
  - UIEventTrigger.cs（为非Button类UI控件提供事件响应能力）
  - UIPanel.cs （UI面板的基类，所有UI面板都继承它）
  - UIRoot.cs（UI面板在场景中的根结点，始终保持只有1个）
  - UIUtils.cs
  - UIRes.cs
  - UIWindow.cs/UIWidge.cs/UIPage.cs/UILoading.cs
  - UIManager.cs

1.4 示例及练习

- 示例
  - 创建一个模块：ExampleA
  - 创建它的主UI：UIExampleAPage
  - 创建其它UI：UIMsgBox，UIMsgTips，UISceneLoading
  - 增加Loading动画：LoadingAnimation
  - 增加Window动画：WindowAnimation
  - 在AppMain里显示该模块

1.4 示例及练习

- 练习
  - 创建一个模块：ExampleB
  - 创建它的主UI：IUExampleBPage
  - UI上有一个返回按钮，点击返回上一个Page
  - 为IUExampleBPage增加动画效果
  - 在AppMain里显示该模块

2 通用网络模块

- 2.1 需求分析
- 2.2 框架设计
- 2.3 公共模块编码
- 2.4 前台编码
- 2.5 后台编码
- 2.6 示例及练习

什么是【通用】网络模块？

就是它可以适应大部分网络通讯的应用场景。一般来讲，用来满足核心战斗外的网络通讯需求是没有任何问题的。一些采用状态同步的核心战斗也可以用它。它是与具体游戏无关的。

2.1 需求分析

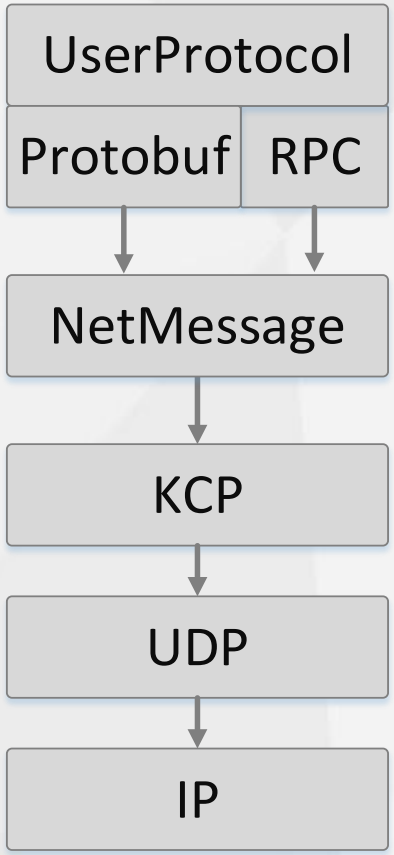
- 需求
  - 具备前后台配套的框架
  - 具备很好的扩展性
  - 能够很好地适应弱网络
  - 具备友好的调用方式

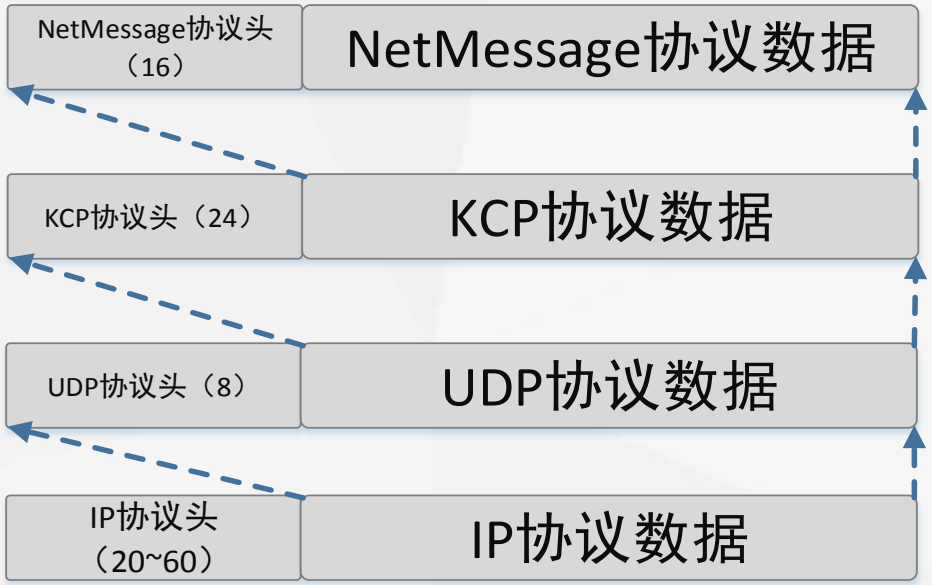
2.1 需求分析

- 功能定义
  - 具备前后台配套的框架
    - 网络模块前台框架、网络模块后台框架
  - 具备很好的扩展性
    - 定义IConnection接口，逐步实现不同的连接方式
    - 目前已经实现了RUDP（基本上可以取代TCP、UDP）
    - 基于HTTP的连接，一般会与具体业务相关，可以在具体业务需要时实现
  - 能够很好地适应弱网络
    - 由于采用RUDP，在底层会保证只要网络恢复了，数据一定会【可靠】传输
  - 具备友好的调用方式
    - 支持传统的Protobuf协议和RPC协议，2种协议定义方式
    - 支持单播、多播、广播，3种消息接收方式

2.2 框架设计

- 协议栈定义



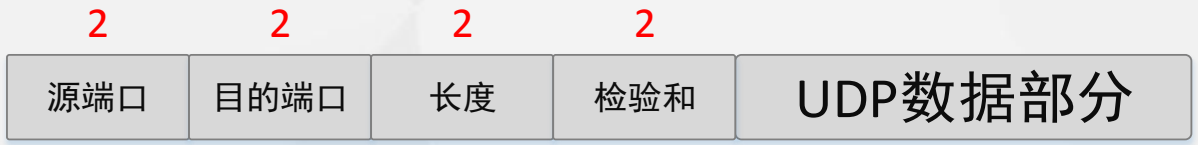


2.2 框架设计

- 协议栈定义
  - IP协议
    - 以下只是大概介绍，详细资料可以去网络查找
    - 协议头：20到60字节，如下图所示

2.2 框架设计

- 协议栈定义
  - UDP协议
    - 以下只是大概介绍，详细资料可以去网络查找
    - 协议头：8字节，如下图所示



2.2 框架设计

- 协议栈定义
  - KCP协议
    - 以下只是大概介绍，详细资料可以去网络查找
    - 协议头：24字节，如下图所示
    - Conv字段需要在一个连接的两个端点保持一致

2.2 框架设计

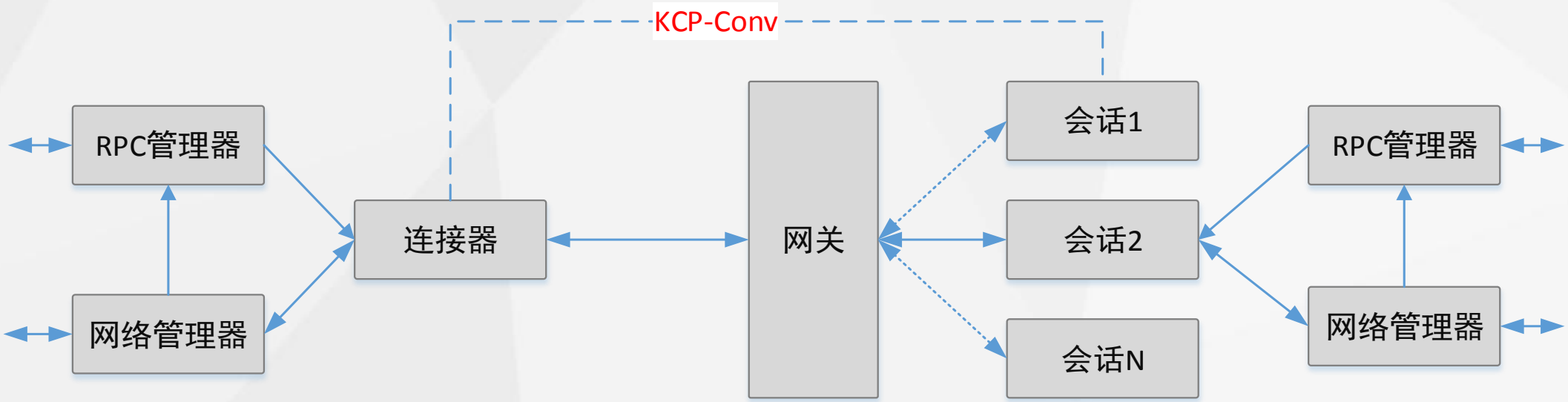
- 协议栈定义
  - NetMessage协议
    - 这是当前网络框架层协议
    - 协议头：16字节，如下图所示

2.2 框架设计

- 模块划分
  - Client
    - 连接器：负责实现KCP/UDP/IP协议层
    - 网络管理器：主要是管理协议的收发
    - RPC管理器：负责实现UserProtocol-RPC协议
  - Server：
    - 网关/会话：负责管理与Client的连接一一对应的基于KCP/UDP/IP通讯的会话
    - 网络管理器：同Client
    - RPC管理器：同Client

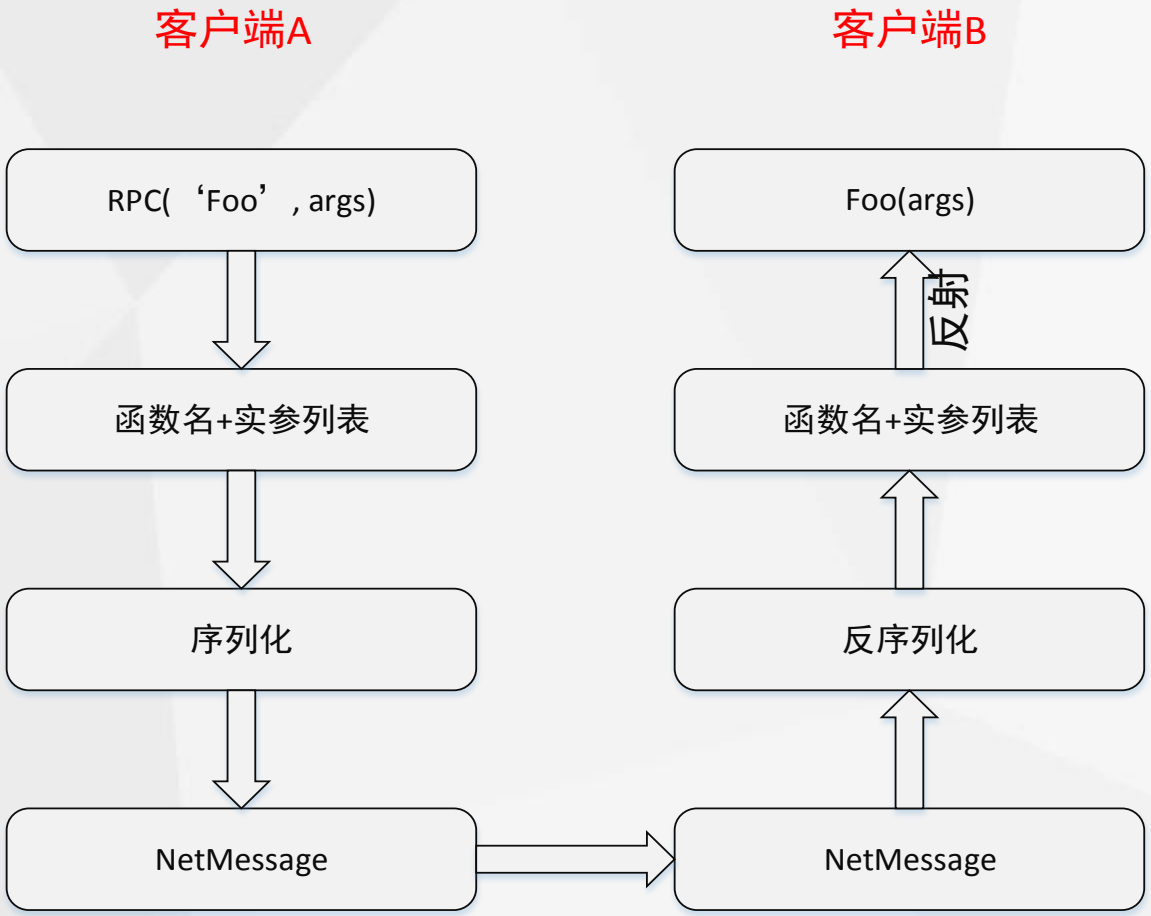
2.2 框架设计

- 基本原理



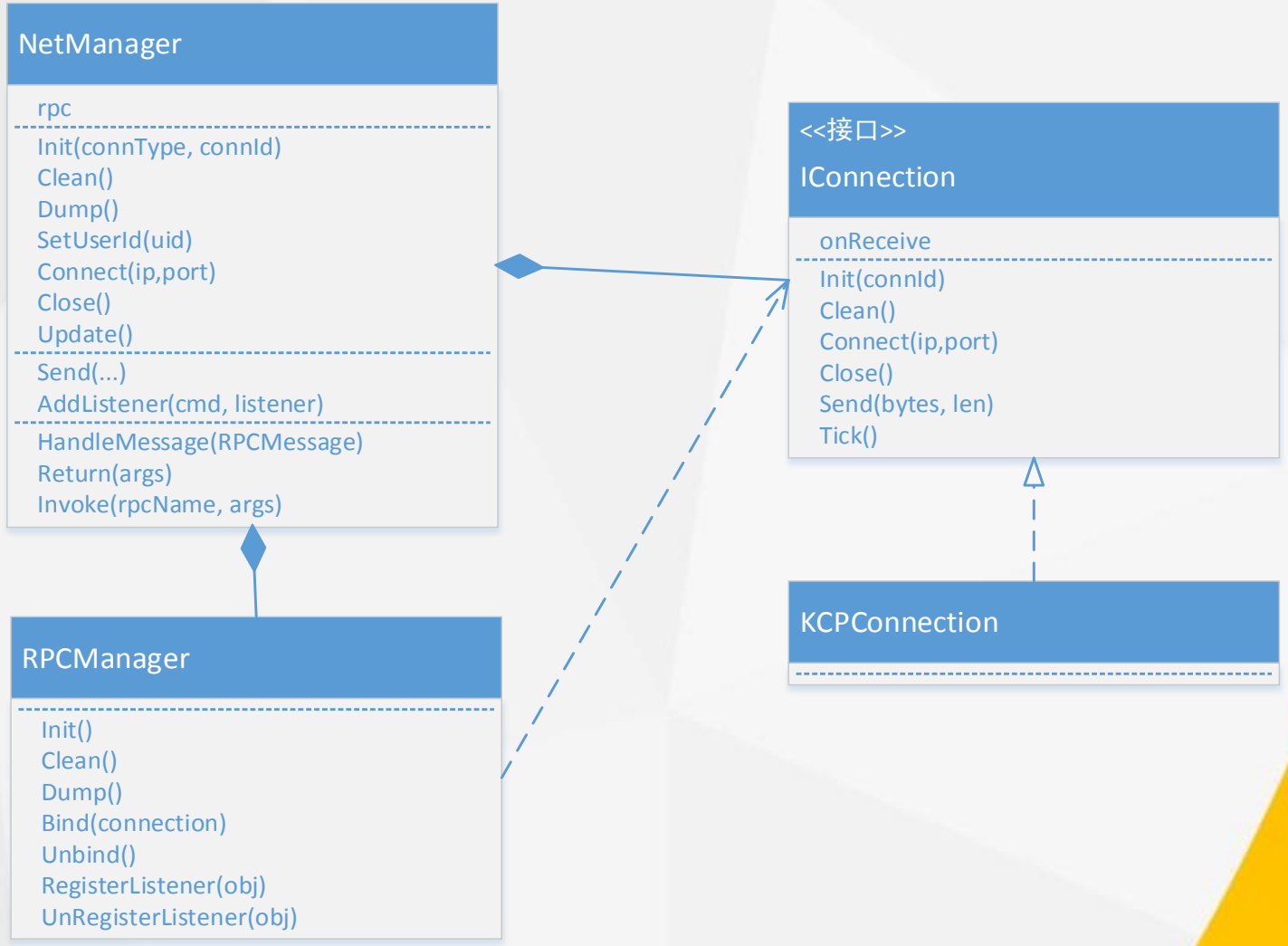
2.2 框架设计

- RPC原理



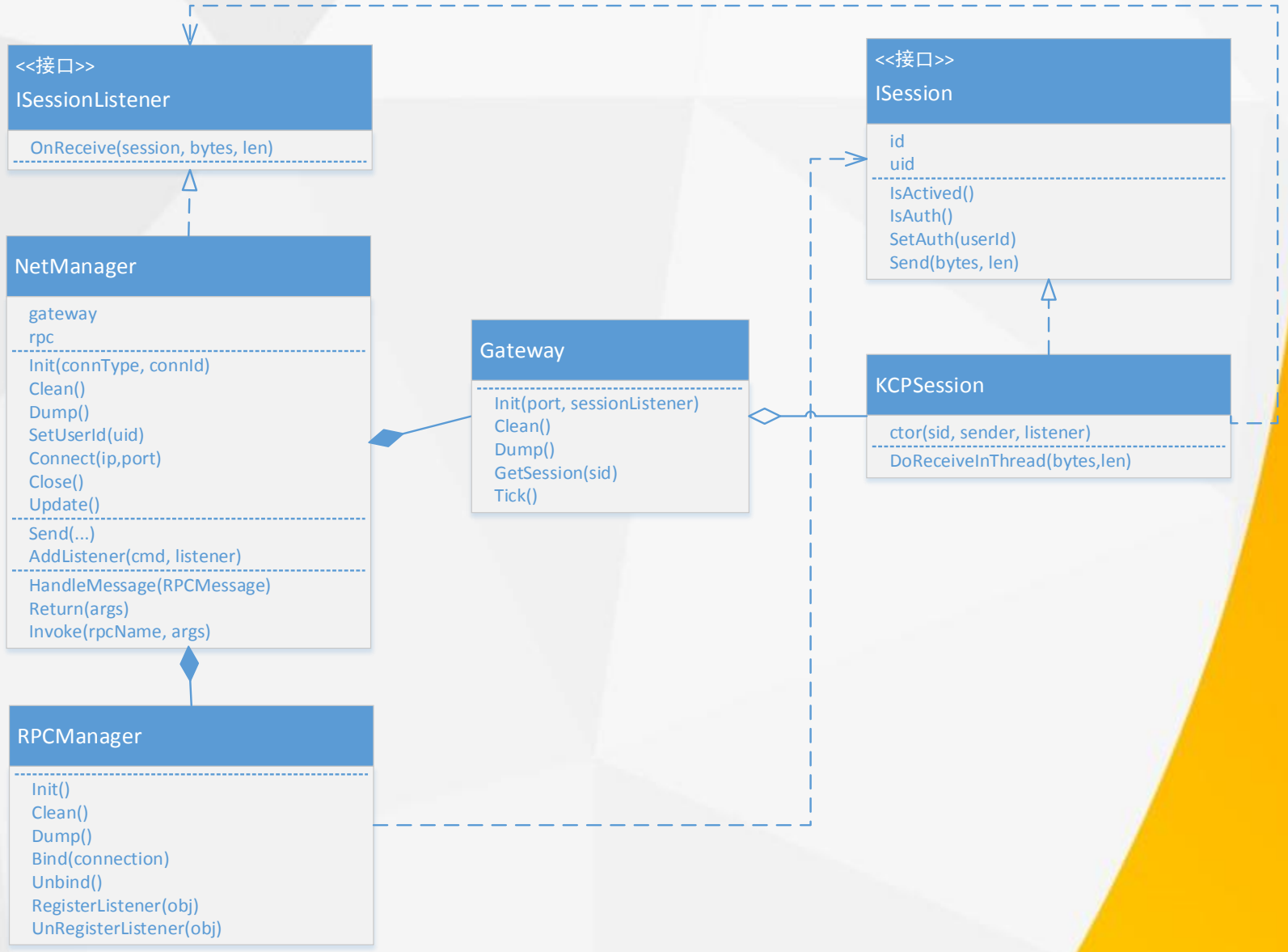
2.2 框架设计

- 概要设计
  - Client



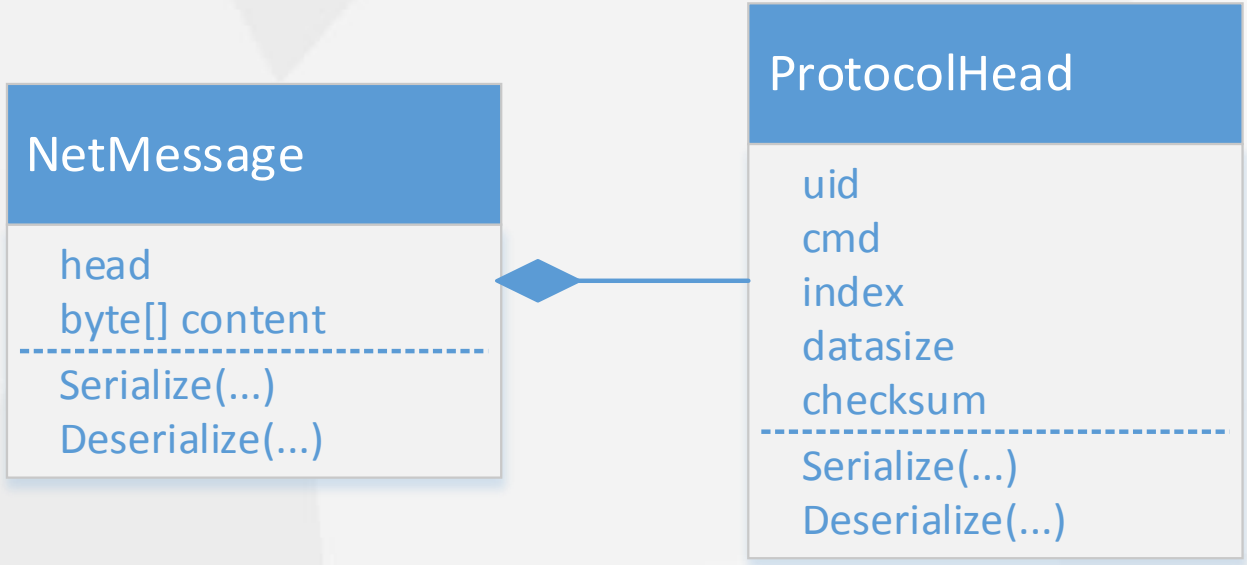
2.2 框架设计

- 概要设计
  - Server



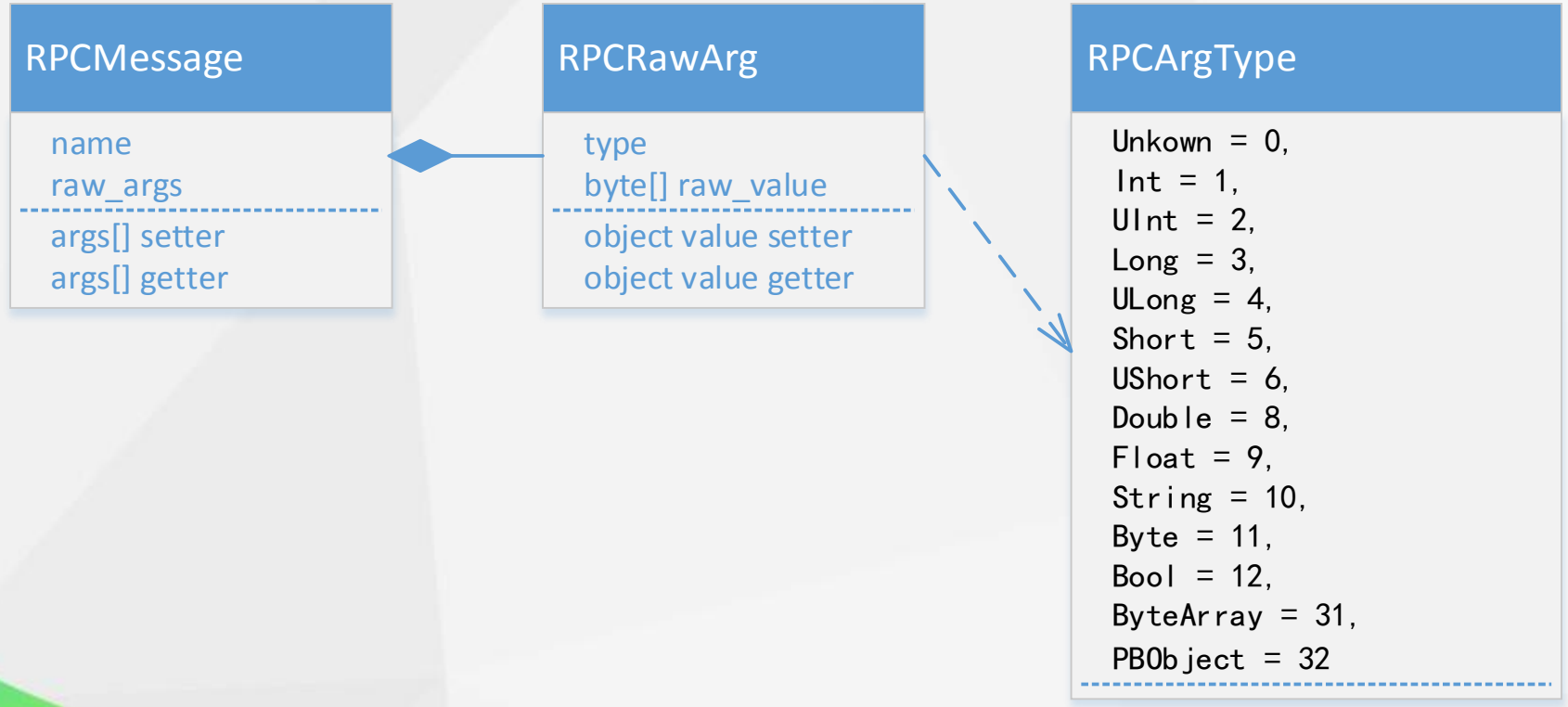
2.2 框架设计

- 概要设计
  - NetMessage



2.2 框架设计

- 概要设计
  - RPCMessage



2.3 公共模块编码

- 数据结构
  - NetMessage、ProtocolHead
  - RPCMessage、RPCArgType、RPCRawArg

2.3 公共模块编码

- RPC模块
  - RPCManagerBase（重点）
  - RPCMethodHelper
  - RPCAttribute

2.4 前台编码

- 连接器
  - IConnection
  - KCPConnection（重点）

2.4 前台编码

- 网络管理器
  - MessageIndexGenerator
  - NetManager（重点）

2.5 后台编码

- 网关/会话
  - ISession、ISessionListener、SessionID
  - KCPSession（重点）
  - Gateway（重点）

2.5 后台编码

- 网络管理器
  - 后台的编码与前台类似，但不完全一致，所以需要单独实现
  - NetManager

2.6 示例及练习

- 示例
  - 创建一个C#的服务器工程，定义一个服务器类
  - 定义Protobuf协议：LoginReq、LoginRsp
  - 初始化服务器端的NetManager及相关逻辑
  - 初始化客户端的NetManager及相关逻辑
  - 定义RPC协议：StartGameRequest，NotifyStartGame

2.6 示例及练习

- 练习
  - 创建一个新的C#的服务器工程
  - 初始化服务器NetManager及相关逻辑
  - 初始化客户端NetManager及相关逻辑
  - 定义Protobuf协议：RegisterReq，RegisterRsp
  - 定义RPC协议：JoinRoom，UpdateRoomInfo

3 轻量服务器框架实现

- 3.1 需求分析
- 3.2 基本原理
- 3.3 概要设计
- 3.4 编码实现
- 3.5 示例及练习

3.1 需求分析

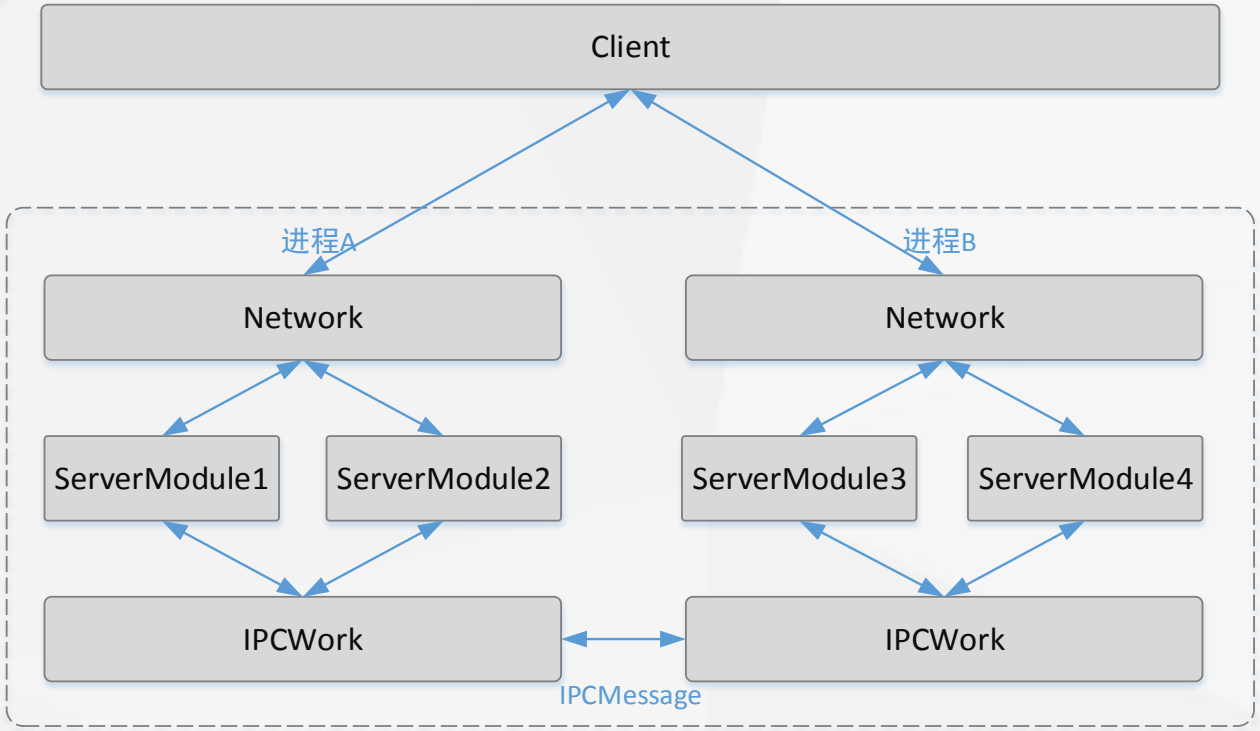
- 为什么要实现一个服务器框架？
  - 为了能够验证我们的网络模块
  - 为了快速实现后续【登录】、【房间】等模块的逻辑功能

3.1 需求分析

- 为什么是【轻量】服务器框架？
  - 架构简洁，方便学习
  - 方便开发独立游戏（大部分独立游戏开发者都只擅长前台开发）
    - 一般在大公司里，其服务器框架非常复杂，依赖大量公司内部的组件。其学习成本非常高，想要单独拿出来自己做一个游戏几乎不可能。

3.2 基本原理

- 单服多进程架构
  - 轻量级的框架，没有采用复杂的多服架构
  - 多服架构非常不好部署，没有完善的工具支持，以及丰富的后台开发经验，很难驾驭。

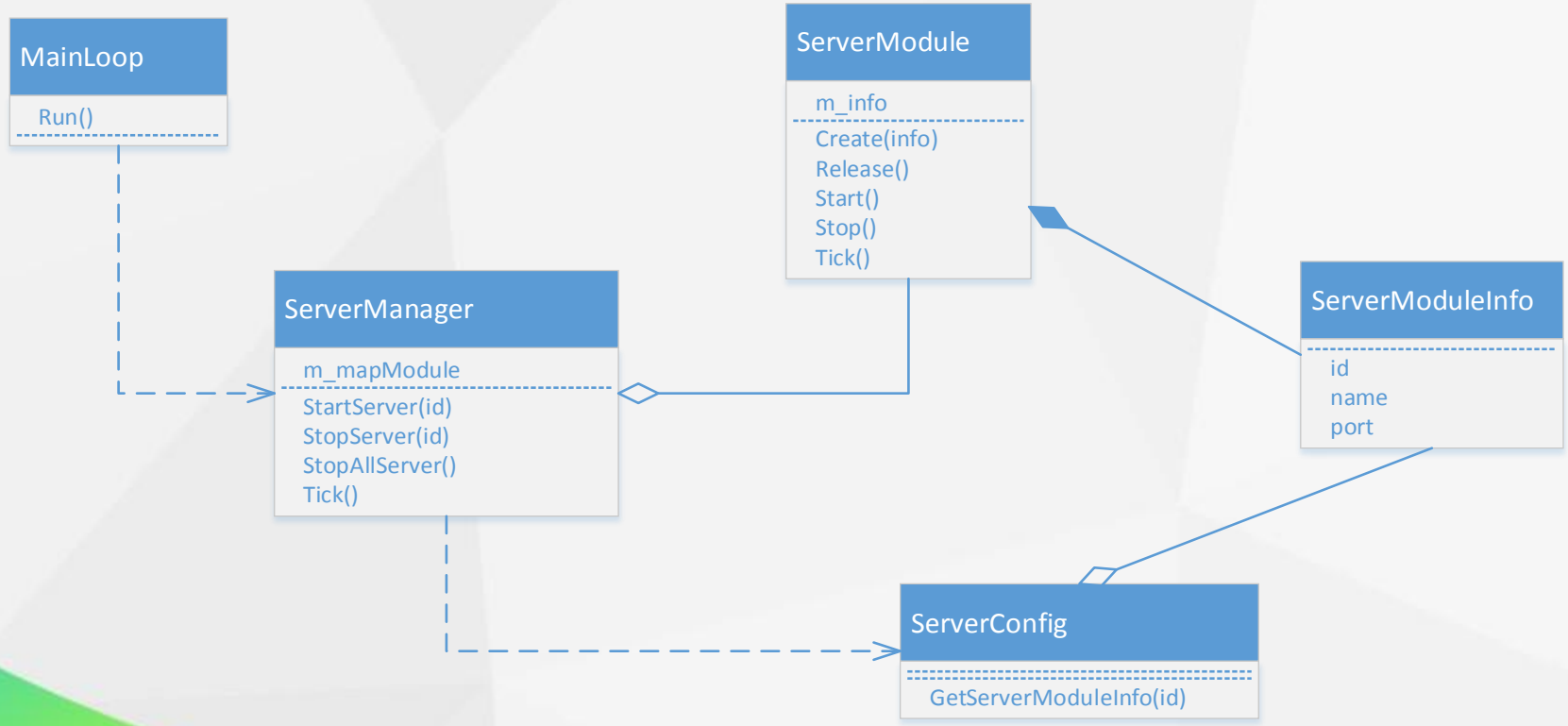


3.2 基本原理

- ServerN
  - 一个Server模块，Server模块之间相互独立
  - 一个进程可以容纳多个Server模块
- Network
  - 负责与Client之间的通讯（详见第2章）
- IPCWork
  - 进程间通讯组件，负责Server模块之间的通讯
  - 由于只支持单服进程通讯，其实现方案非常简单

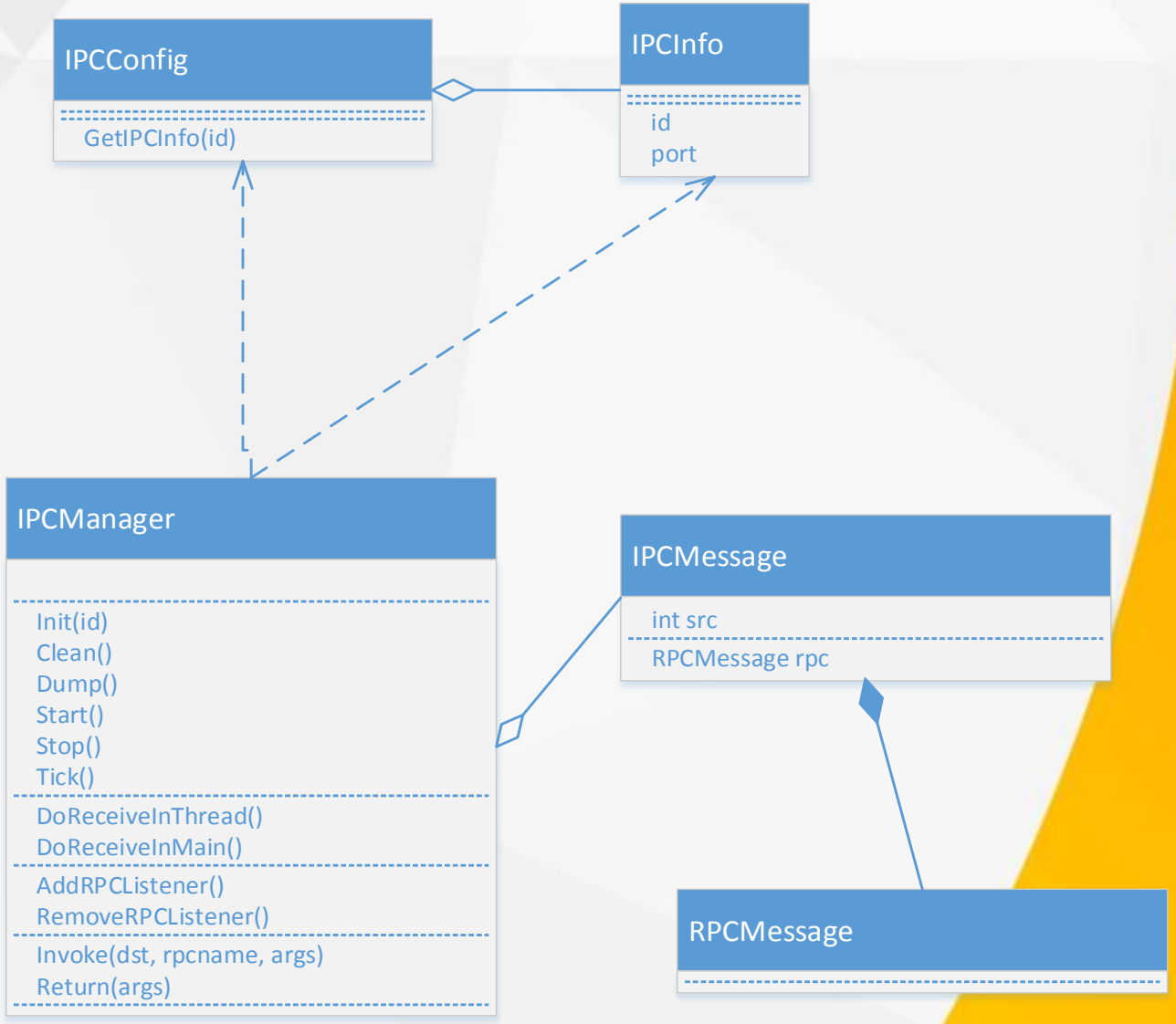
3.3 概要设计

- 概要设计
  - 服务模块管理



3.3 概要设计

- 概要设计
  - IPCWork
    - 其核心思想就是用Socket在两个进程之间进行通讯。
    - 由于进程间通讯在单服内进行，没有网络干扰，所以可以采用UDP协议。
    - 为了使通讯接口友好，接入了现有的RPC模块。



3.4 编码实现

- 服务模块管理器
  - MainLoop
  - ServerConfig、ServerModuleInfo
  - ServerModule
  - ServerManager

3.4 编码实现

- IPCWork的实现
  - IPCConfig、IPCInfo
  - IPCMessage
  - IPCManager

3.5 示例及练习

- 示例
  - 定义3个Server模块：Server1，Server2，Server3
  - 其中Server1与Server2在同一个进程中，Server3在另一个进程中
  - 定义2个RPC，从Server1调到Server2，再从Server2调到Server3

3.5 示例及练习

- 练习
  - 定义ZoneServer和GameServer
  - 定义RPC：StartGame，从ZoneServer调到GameServer
  - 定义RPC：OnStartGame，从GameServer调到ZoneServer

4 登录模块示例

- 4.1 需求分析
- 4.2 概要设计
- 4.3 协议定义
- 4.4 编码实现
- 4.5 联调演示

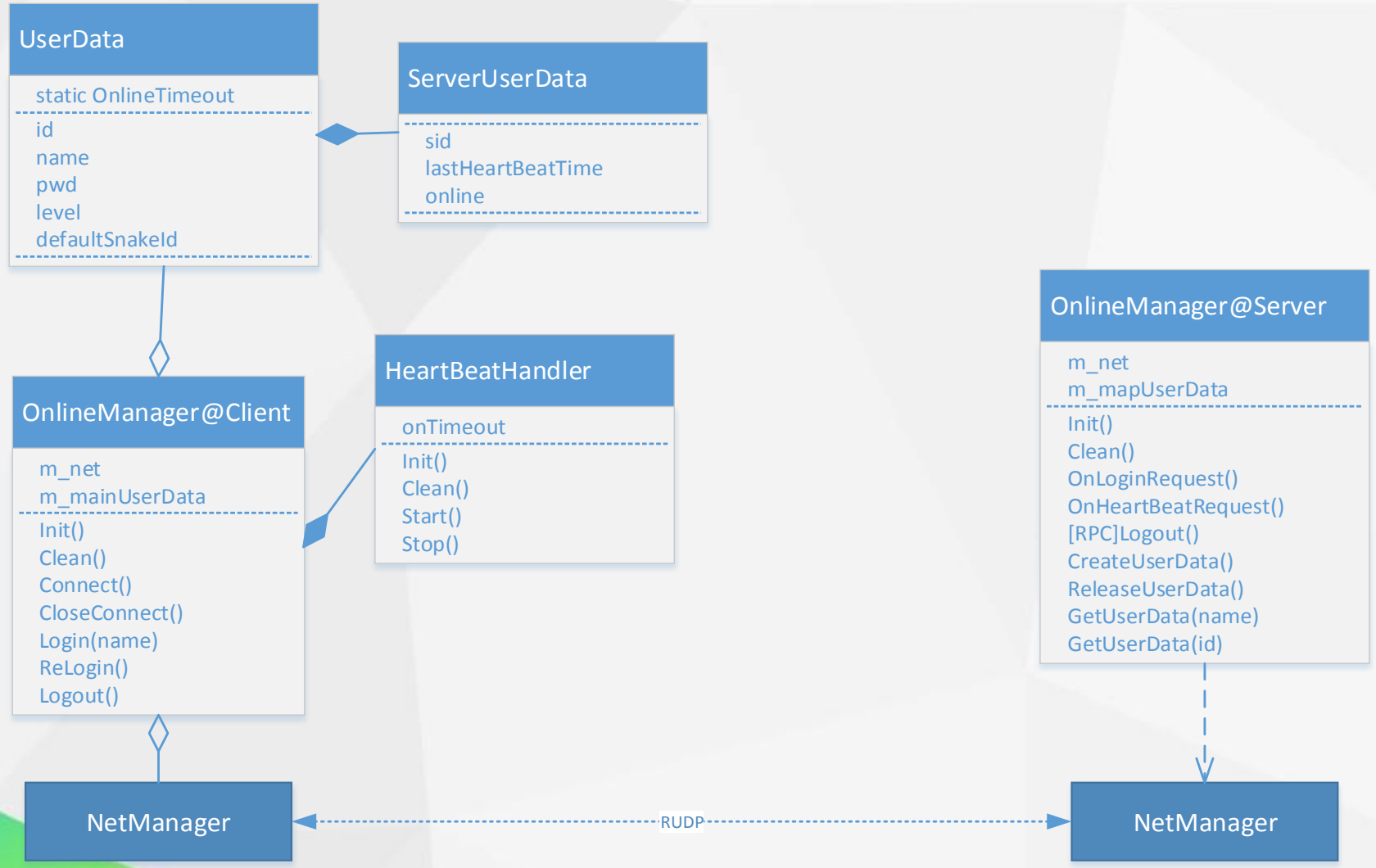
4.1 需求分析

- 在第1季的课程中，并没有实现真实的登录。
- 因为当时把该课程定位为一个纯前端课程，没有实现后端实现。
- 现在我们已经有了一个后端框架（详见第3课），那么就可以实现真实的登录了。

4.1 需求分析

- 由于登录的目的只是为了创建房间，那么登录功能非常简单：
  - 在服务器保存一个【在线玩家列表】。
  - 当玩家登录时，便将其信息记录在列表中。
  - 当玩家离线时，将其信息从列表中删除。
  - 需要为玩家分配一个唯一的ID。
  - 需要判断玩家的名字是否已经被其它玩家占用。
  - 实现心跳逻辑，以判断玩家是否在线。

4.2 概要设计


UserData
static OnlineTimeout
id
name
pwd
level
defaultSnakeId
ServerUserData
sid
lastHeartBeatTime
online
OnlineManager@Client
m_net
m_mainUserData
Init()
Clean()
Connect()
CloseConnect()
Login(name)
ReLogin()
Logout()
HeartBeatHandler
onTimeout
Init()
Clean()
Start()
Stop()
OnlineManager@Server
m_net
m_mapUserData
Init()
Clean()
OnLoginRequest()
OnHeartBeatRequest()
[RPC]Logout()
CreateUserData()
ReleaseUserData()
GetUserData(name)
GetUserData(id)
NetManager
RUDP
NetManager

4.3 协议定义

- 登录协议
  - 上行：LoginReq
    - Id：用户Id，第1次登录时为0，由服务器分配。重登时，可以填写之前的Id
    - Name：用户名，不能为空，且与其它玩家互斥，否则登录失败！
  - 下行：LoginRsp
    - ReturnCode：统一的返回码。如果为0，则表示协议执行成功，否则失败。
    - UserData：服务器返回的用户数据。

4.3 协议定义

- 心跳协议
  - 上行：HeartBeatReq
    - ping：客户端当前的PING值，用于服务器评估客户端的网络情况，以后有可能会用到。
    - timestamp：协议发送时的时间戳，用于计算自己的PING值
  - 下行：HeartBeatRsp
    - ReturnCode：统一的返回码。如果为0，则表示协议执行成功，否则失败。
    - timestamp：服务器透传过来的，上行包中发送时的时间戳，与当前时间相减计算PING值

4.4 编码实现

- 公共数据
  - UserData
  - ServerUserData

4.4 编码实现

- 前端模块
  - OnlineManager
  - HeartBeatHandler

4.4 编码实现

- 后端模块
  - OnlineManager

4.5 联调演示

- 联调演示

5 房间模块示例

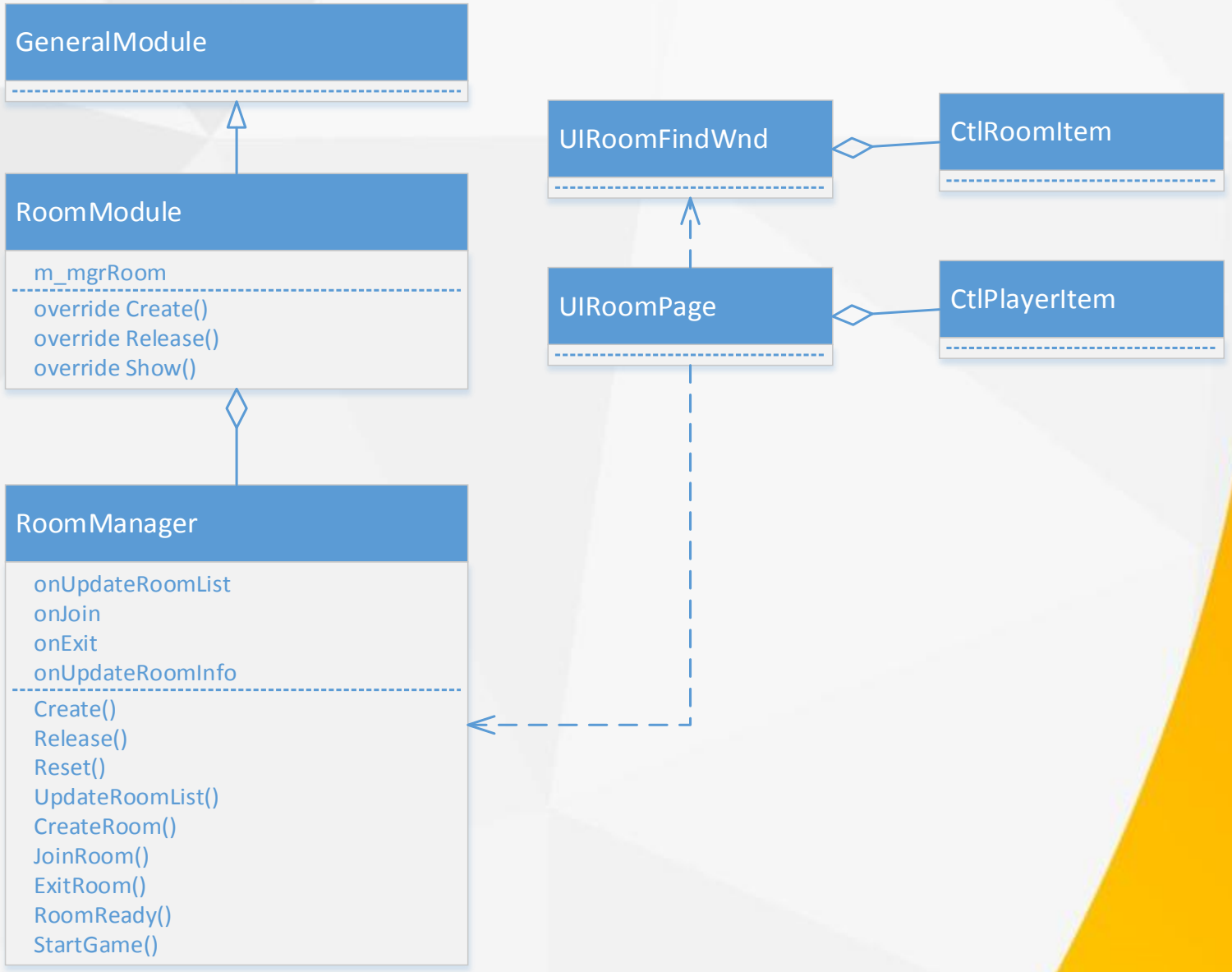
- 5.1 需求分析
- 5.2 概要设计
- 5.3 协议定义
- 5.4 编码实现
- 5.5 联调演示

5.1 需求分析

- 前端
  - 玩家可以创建1个房间
  - 玩家可以获取房间列表
  - 玩家可以加入、退出1个房间
  - 玩家可以准备、取消准备、开始游戏
- 后端
  - 服务器管理一个房间列表
  - 房间有一个房主，房主无法退出房间，但是可以解散房间
  - 当1个房间的玩家信息发生变化时，需要同步给房间内所有玩家
  - 房主开启游戏时，将开始游戏消息通知给房间内所有玩家
  - 房间模块可以调用GameServer启动一个游戏单局

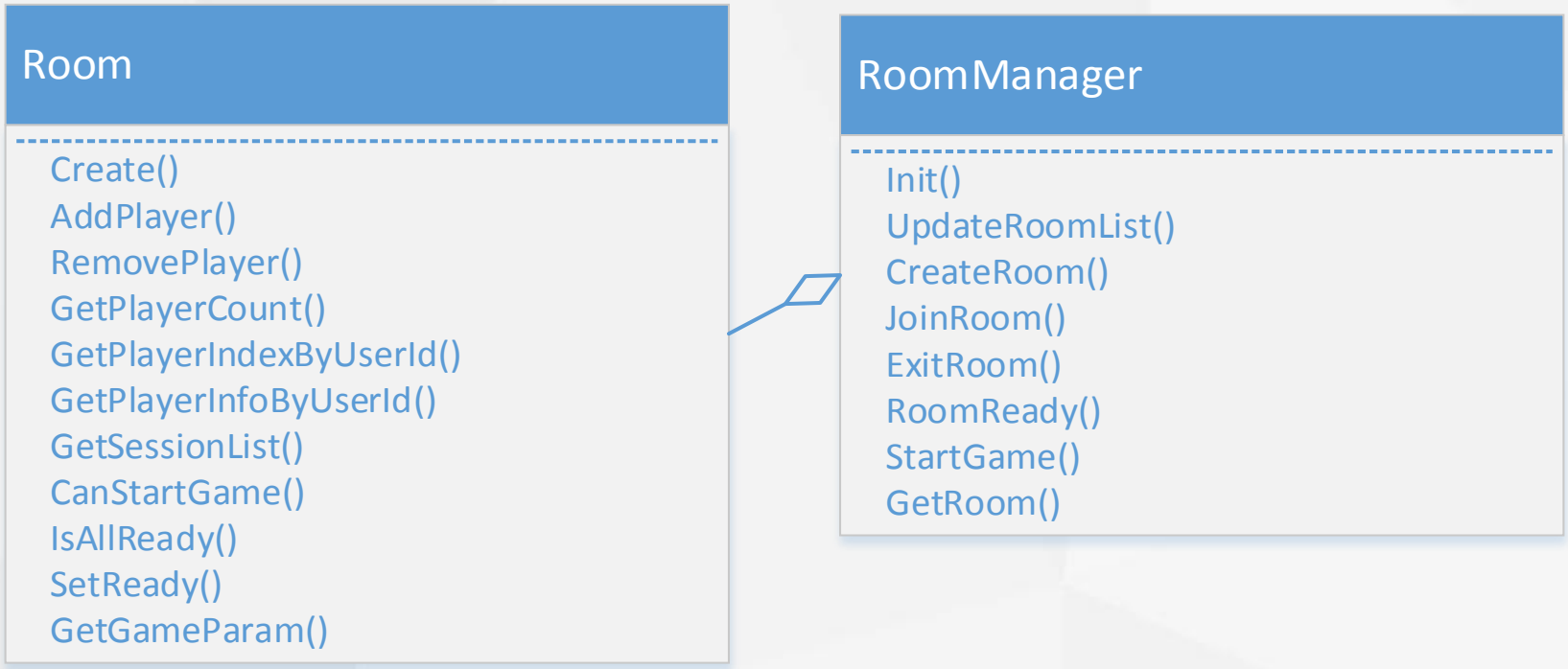
5.2 概要设计

- 前端


GeneralModule
RoomModule
m_mgrRoom
override Create()
override Release()
override Show()
RoomManager
onUpdateRoomList
onJoin
onExit
onUpdateRoomInfo
Create()
Release()
Reset()
UpdateRoomList()
CreateRoom()
JoinRoom()
ExitRoom()
RoomReady()
StartGame()
UIRoomFindWnd
CtlRoomItem
UIRoomPage
CtlPlayerItem

5.2 概要设计

- 后端


Room
Create()
AddPlayer()
RemovePlayer()
GetPlayerCount()
GetPlayerIndexByUserId()
GetPlayerInfoByUserId()
GetSessionList()
CanStartGame()
IsAllReady()
SetReady()
GetGameParam()
RoomManager
Init()
UpdateRoomList()
CreateRoom()
JoinRoom()
ExitRoom()
RoomReady()
StartGame()
GetRoom()

5.3 协议定义

- 上行
  - void UpdateRoomList()
  - void CreateRoom(uint userId, string roomName)
  - void JoinRoom(uint userId, uint roomId)
  - void ExitRoom(uint userId, uint roomId)
  - void RoomReady(uint userId, uint roomId, bool ready)
  - void StartGame(uint userId, uint roomId)

5.3 协议定义

- 下行
  - void OnUpdateRoomList(RoomListData data)
  - void OnCreateRoom(RoomData data)
  - void OnJoinRoom(RoomData data)
  - void NotifyRoomUpdate(RoomData data)
  - void NotifyGameStart(PVPStartParam param)
  - void NotifyGameResult(int reason)

5.4 编码实现

- 公共数据
  - SnakeData
  - PlayerData
  - RoomData
  - RoomListData
  - PVPStartParam

5.4 编码实现

- 前端模块
  - RoomModule
  - RoomManager
  - UI
    - UIRoomPage
    - CtlPlayerItem
    - UIRoomFindWnd
    - CtlRoomItem

5.4 编码实现

- 后端模块
  - Room
  - RoomManager

5.5 联调演示

- 联调演示

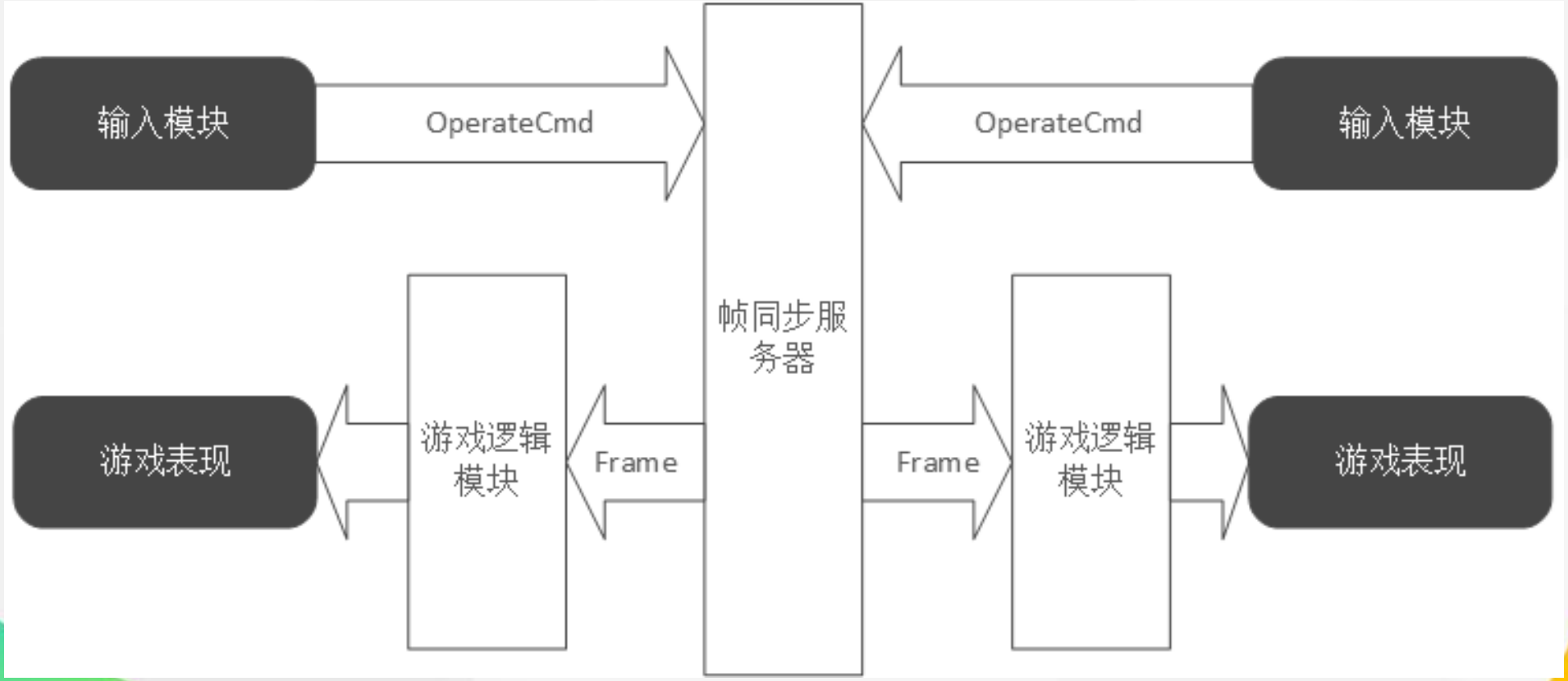
6 帧同步专题

- 6.1 基本原理回顾
- 6.2 整体框架
- 6.3 数据定义
- 6.4 前台设计
- 6.5 前台实现
- 6.6 后台设计
- 6.7 后台实现
- 6.8 联调演示

这一季课主要详细讲解帧同步的编码实现
其基本原理，主要还是参考第1季的课程

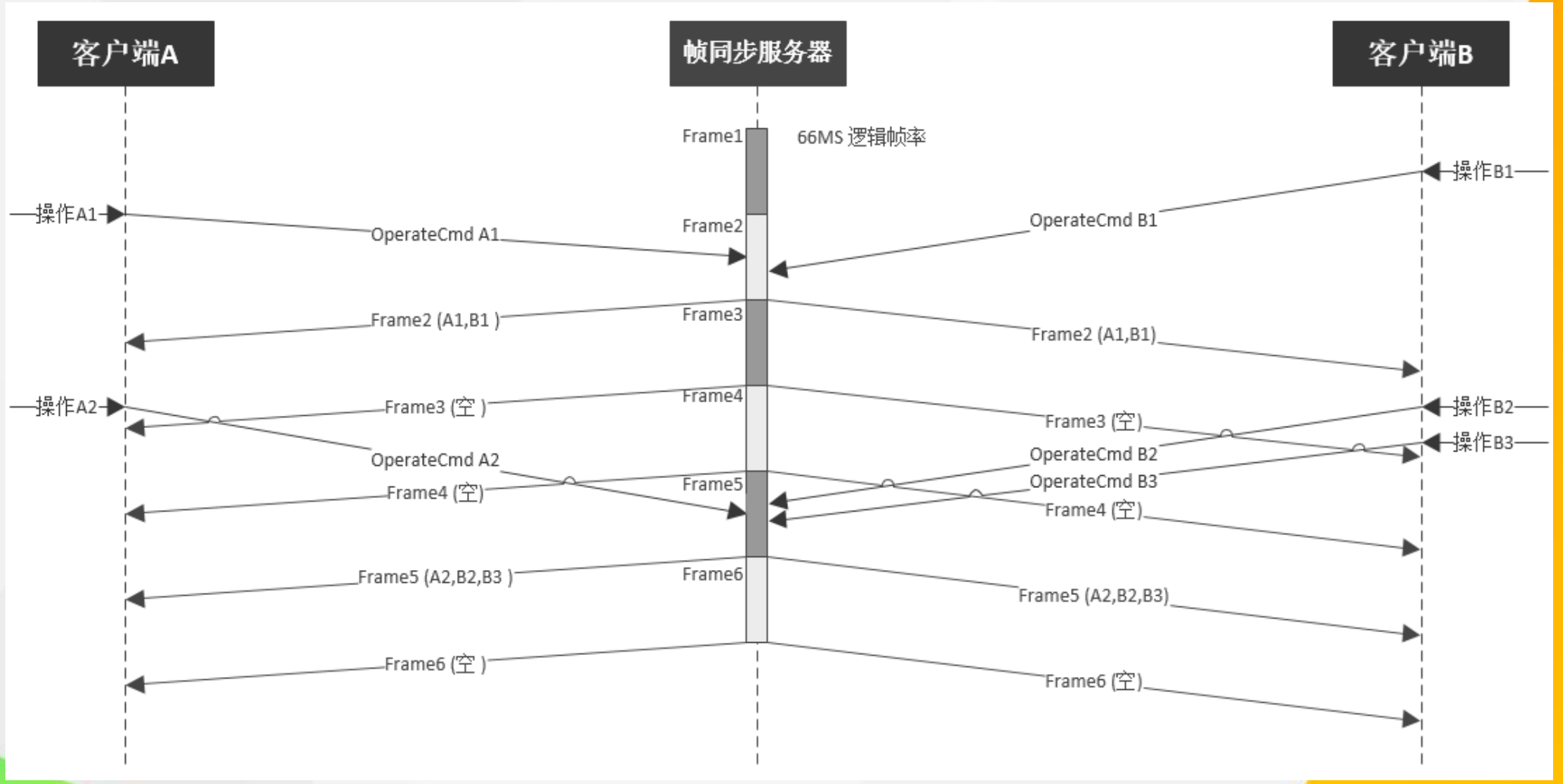
6.1 基本原理回顾

- 示意图



6.1 基本原理回顾

- 时序图



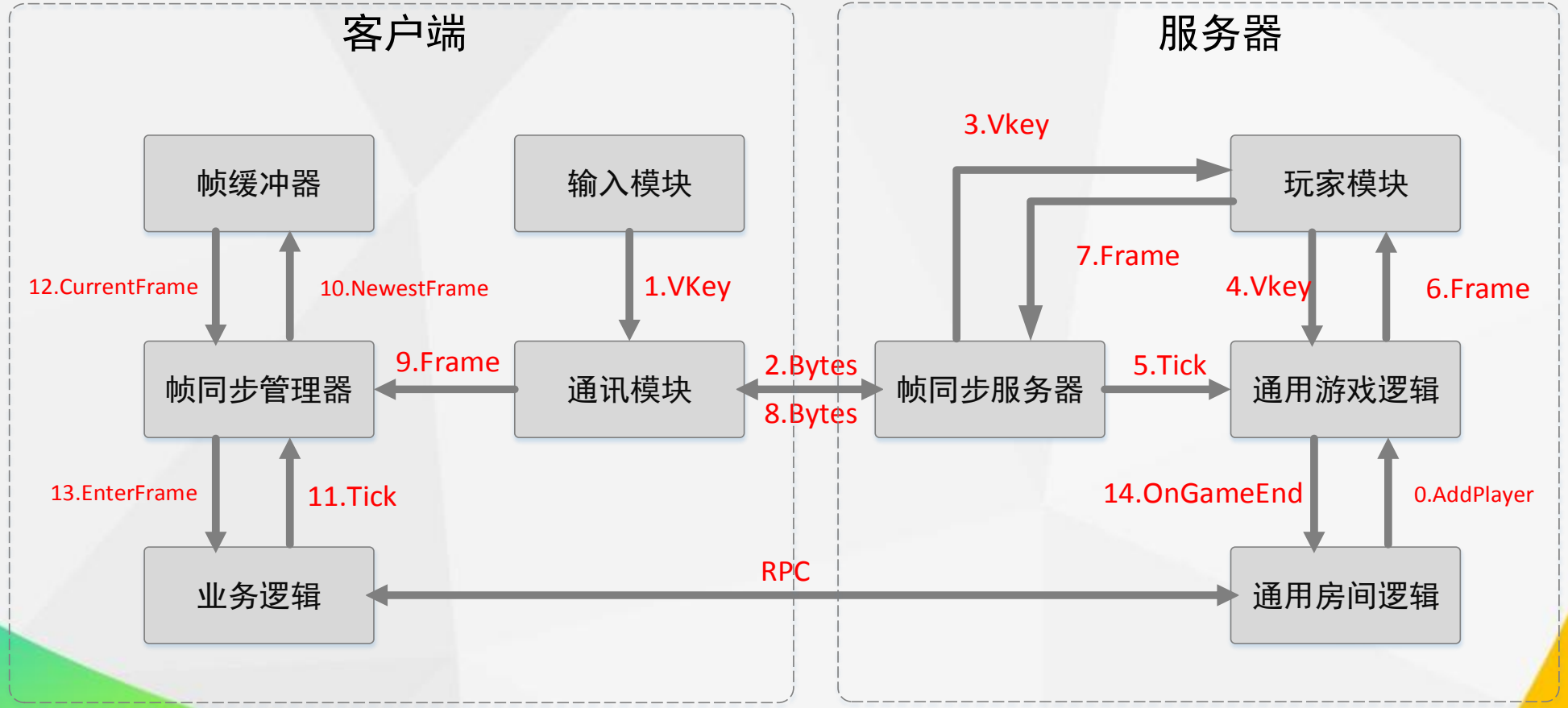
6.1 基本原理回顾

- 通讯原理
  - 采用UDP作为底层通讯协议
  - 对比市面上的RUDP实现方案，最终采用KCP作为RUDP的实现方案

网络环境	TCP平均延迟	UDP平均延迟	结论
网络良好	133	100	RUDP略优
5%丢包	173	143	RUDP略优
50%丢包	262	115	RUDP优势明显
50ms抖动	198	130	RUDP优势明显

6.2 整体框架

- 整体框架


客户端
服务器
帧缓冲器
输入模块
玩家模块
3.Vkey
7.Frame
12.CurrentFrame
10.NewestFrame
1.VKey
4.Vkey
6.Frame
帧同步管理器
9.Frame
通讯模块
2.Bytes
8.Bytes
帧同步服务器
5.Tick
通用游戏逻辑
13.EnterFrame
11.Tick
14.OnGameEnd
0.AddPlayer
RPC
业务逻辑
通用房间逻辑

6.3 数据定义

- 启动参数
  - FSPParam

参数	定义
Host	服务器IP
Port	服务器端口
Sid	会话ID，如果是局域网服务器，则等于PlayerId
serverFrameInternal	服务器 1 帧的时间
serverTimeout	服务器判断客户端掉线的超时
clientFrameRateMultiple	客户端与服务器帧率倍数

6.3 数据定义

- 帧同步消息
  - FSPMessage

字段	定义
Int cmd	虚拟按键
Int[] args	参数列表
int custom	自定义字段

6.3 数据定义

- 上行协议
  - FSPDataC2S

字段	定义
uint Sid	SessionId
List<FSPMessage> msgs	消息列表

6.3 数据定义

- 下行协议
  - FSPDataS2C

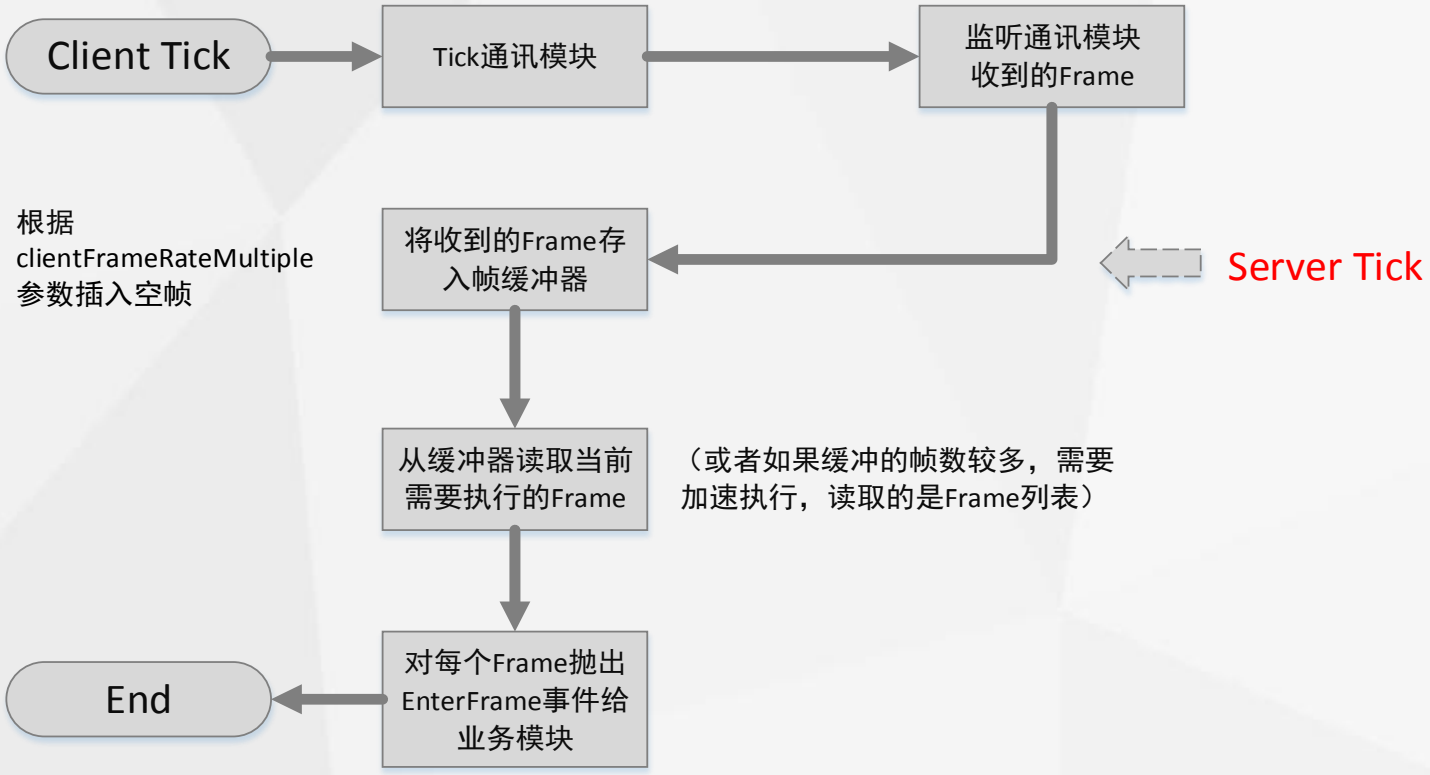
字段	定义
List<FSPFrame> frames	帧列表

  - FSPFrame

字段	定义
Int frameId	帧Id
List<FSPMessage> msgs	消息列表

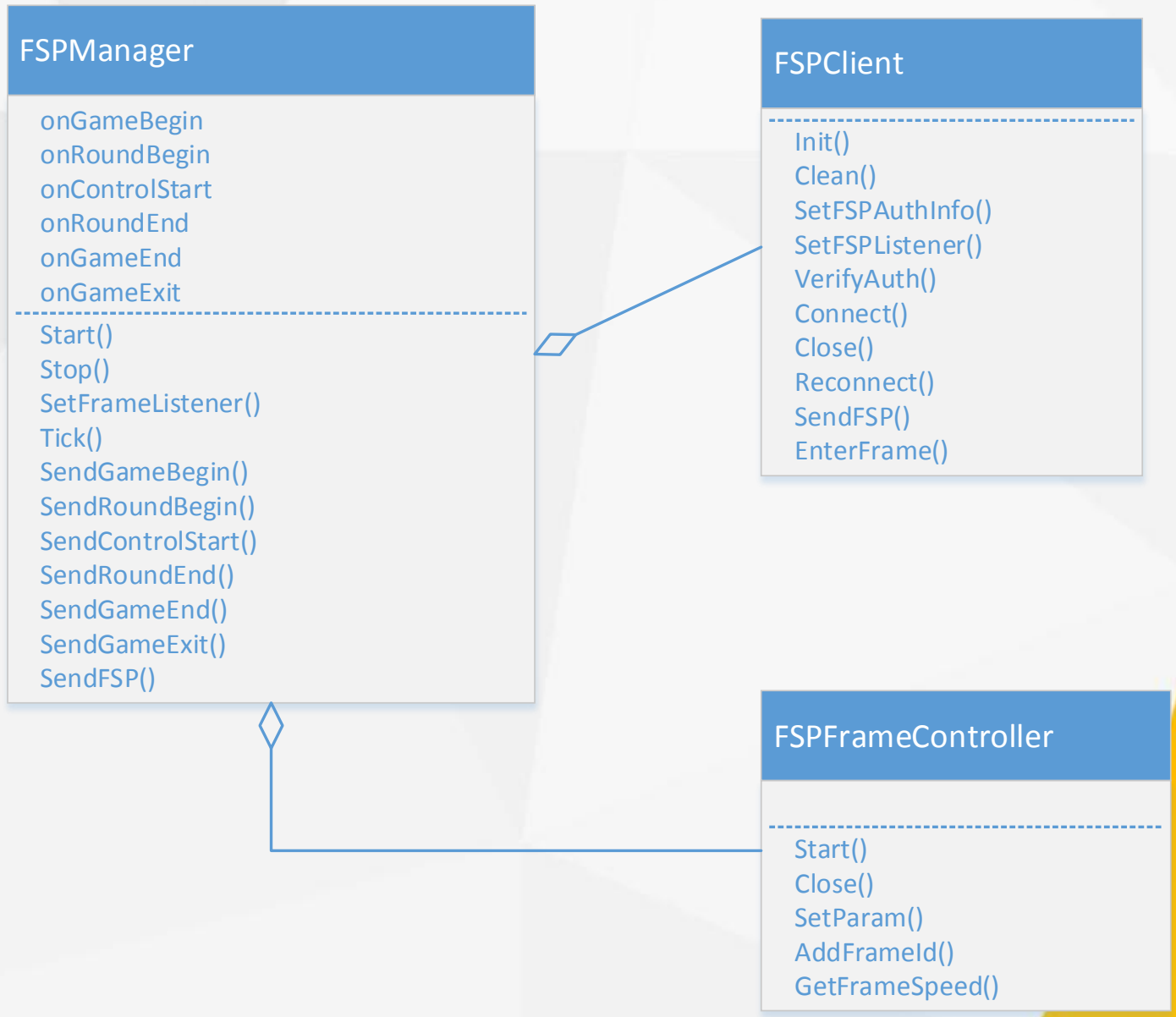
6.4 前台设计

- 基本流程


Client Tick
Tick通讯模块
监听通讯模块
收到的Frame
根据
clientFrameRateMultiple
参数插入空帧
将收到的Frame存
入帧缓冲器
Server Tick
从缓冲器读取当前
需要执行的Frame
（或者如果缓冲的帧数较多，需要
加速执行，读取的是Frame列表）
对每个Frame抛出
EnterFrame事件给
业务模块
End

6.4 前台设计

- 概要设计
  - FSPClient
    - 负责通讯
  - FSPFrameController
    - 负责缓冲追帧控制
  - FSPManager
    - 主体逻辑

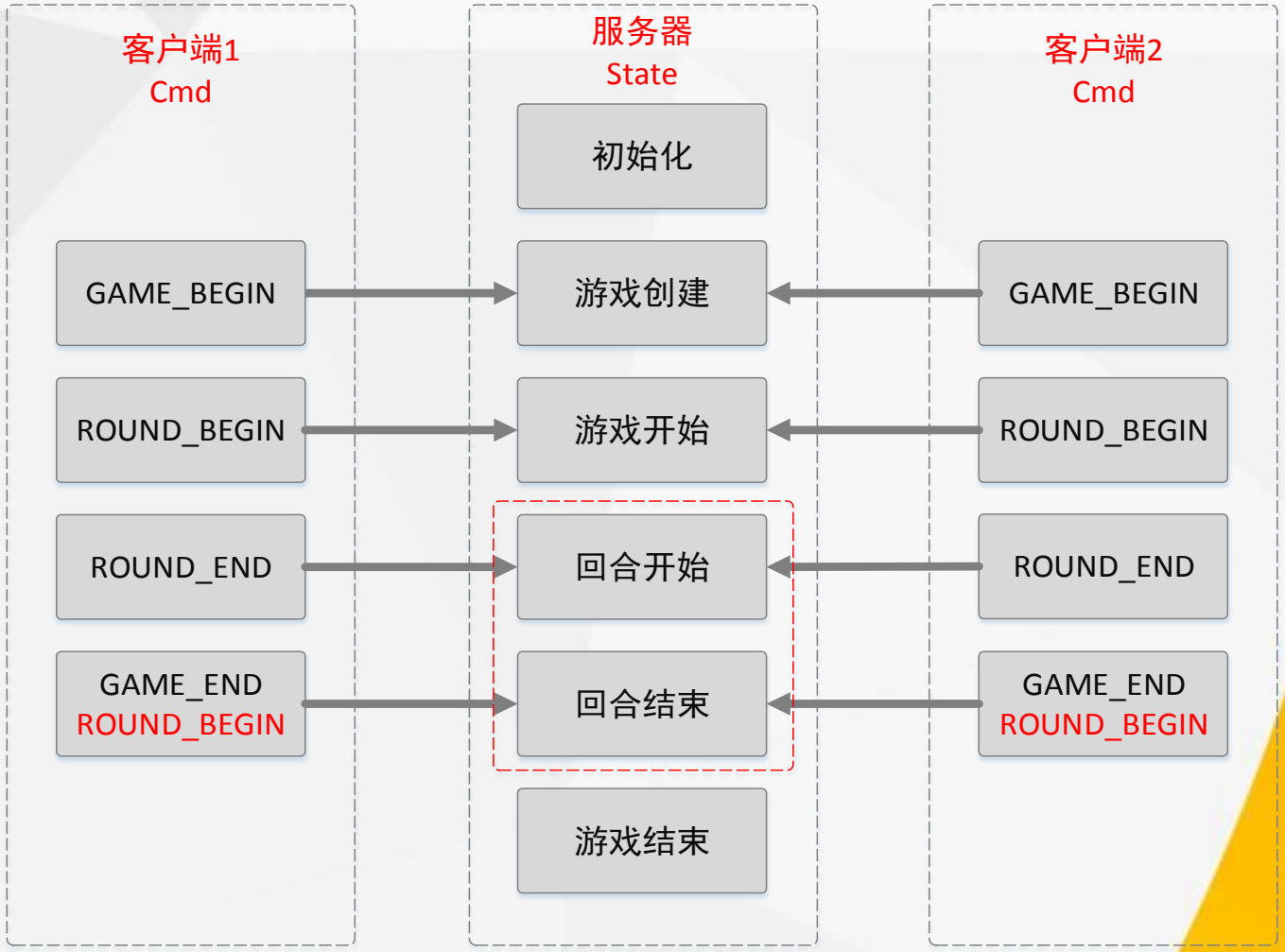


6.5 前台实现

- 模块编码
  - FSPClient
  - FSPManager
  - FSPFrameController

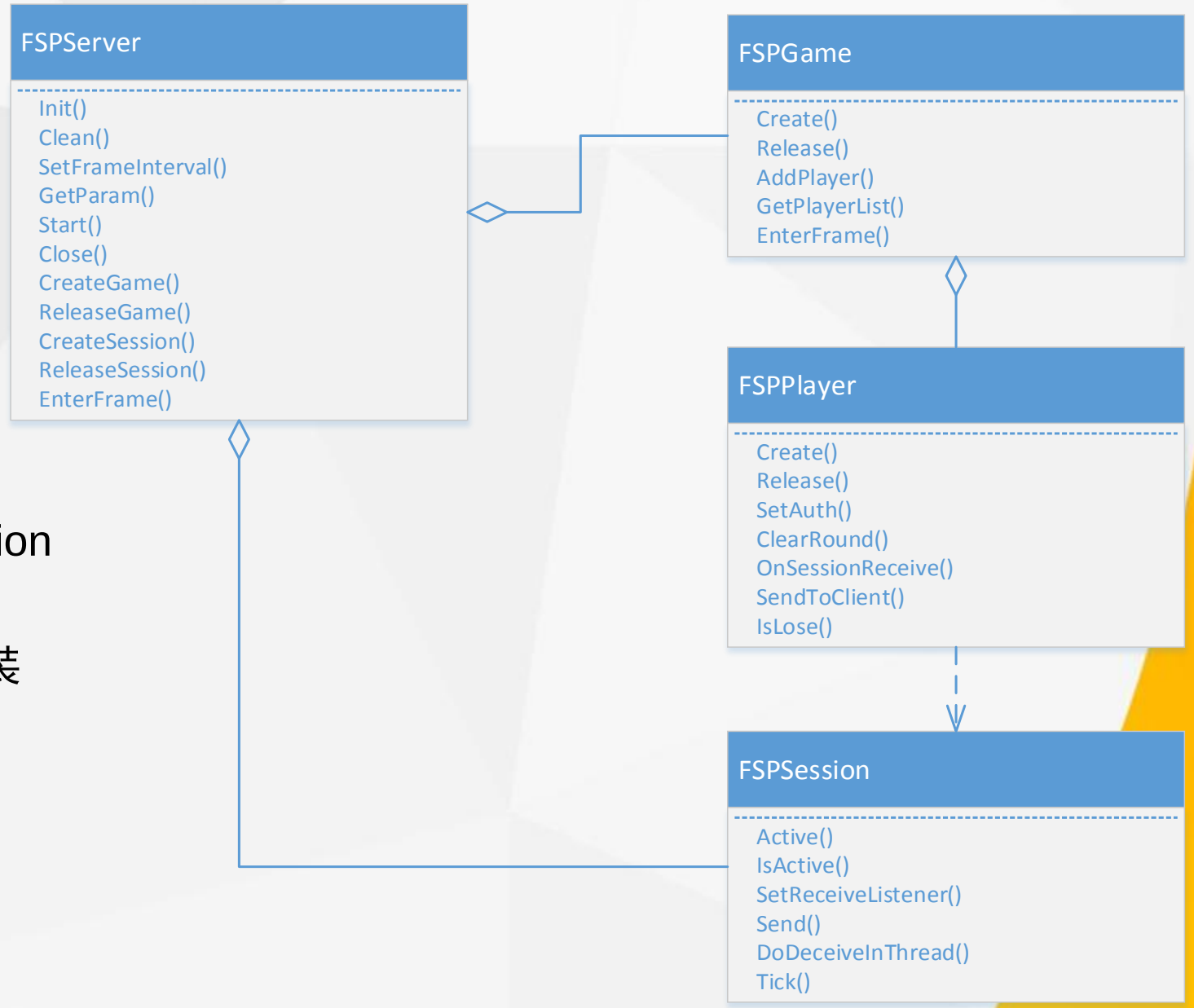
6.6 后台设计

- 基本流程
  - 主体逻辑是一个状态机



6.6 后台设计

- 概要设计
  - FSPServer
    - 负责总体调度
  - FSPSession
    - 负责与FSPClient通讯
    - 每1个玩家对应1个Session
  - FSPPlayer
    - 一个玩家在服务器的封装
  - FSPGame
    - 单局逻辑



6.7 后台实现

- 模块编码
  - FSPGateway
  - FSPSession
  - FSPManager
  - FSPPlayer
  - FSPGame

6.8 联调演示

- 联调演示

7 热更新应用框架

- 7.1 需求分析
- 7.2 架构原理
- 7.3 框架设计
- 7.4 编码实现
- 7.5 示例及练习

这里讲的架构不是指热更VM的架构，而是指应用层面的架构。就是怎么利用已经选型的热更基础技术方案，结合现有的系统架构，实现一个与现有系统架构融合的热更应用架构。

7.1 需求分析

- 现状
  - 由于IOS不支持JIT，所以IOS的热更需要用脚本解析的方式来实现。目前常用的脚本是Lua。
  - 但是Android是支持JIT。如果采用Lua，那么Android版本就无法利用JIT的优势。
  - 由于Lua开发缺少强力IDE的支持，语言本身也不支持OOP（需要用一些取巧的办法来支持），当项目规模变大，越来越多的模块整体都用Lua来开发，会更得项目越来越难维护。
  - Lua作为一种面对过程的脚本语言，相对而言不适合做大规模开发。
  - 同一个游戏版本，在同等匹配的IOS和Android系统中，性能表现不同。

7.1 需求分析

- ILRuntime
  - ILRuntime是一种将C#作为脚本来执行的开源方案。
  - 由于有强力IDE的支持，采用ILRuntime，其开发效率应该非常高
  - 通过合理的框架设计，可以在IOS版本的热更新和Android版本的JIT模式之间无缝兼容

  - 可能的风险是：
    - ILRuntime并没有成功的项目来验证其可靠性

7.1 需求分析

- 从应用层面分析
  - 需要与模块管理器无缝兼容
  - 需要与UI管理器无缝兼容
  - 需要支持解析和原生模式无缝切换
  - 需要有一定的扩展性

7.1 需求分析

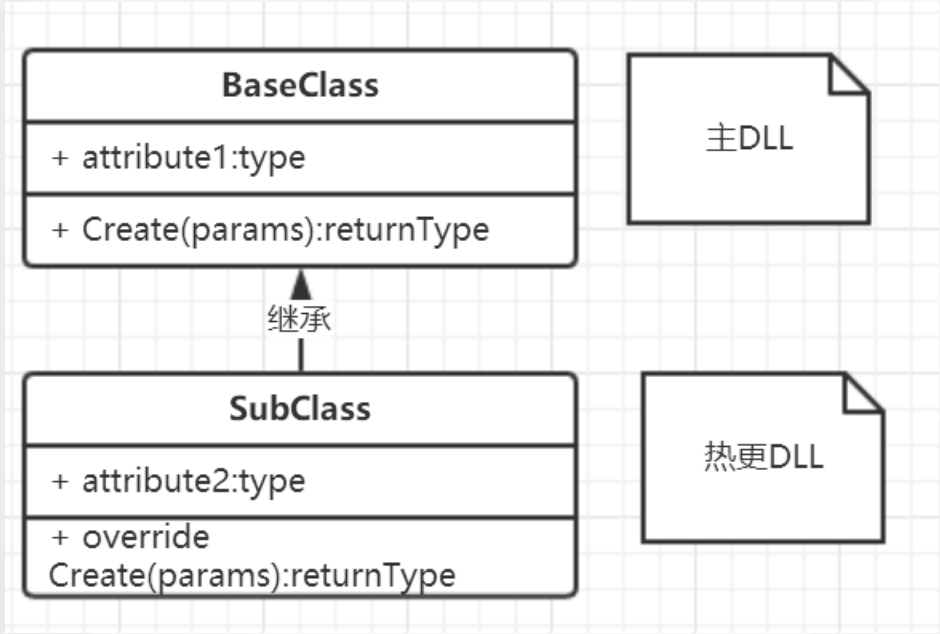
- 从架构层面分析
  - 引用：
    - 热更类引用原生类
    - 原生类引用热更类
  - 继承：
    - 热更类继承原生的基类
    - 热更类重写基类的方法
    - 热更类调用基类的方法

7.2 架构原理

- 基本思路
  - 引用
    - 无论是基于Lua的，还是基于C#的更热方案，对于引用这一块的支持，都是在VM中实现的，我们不需要过多担心。

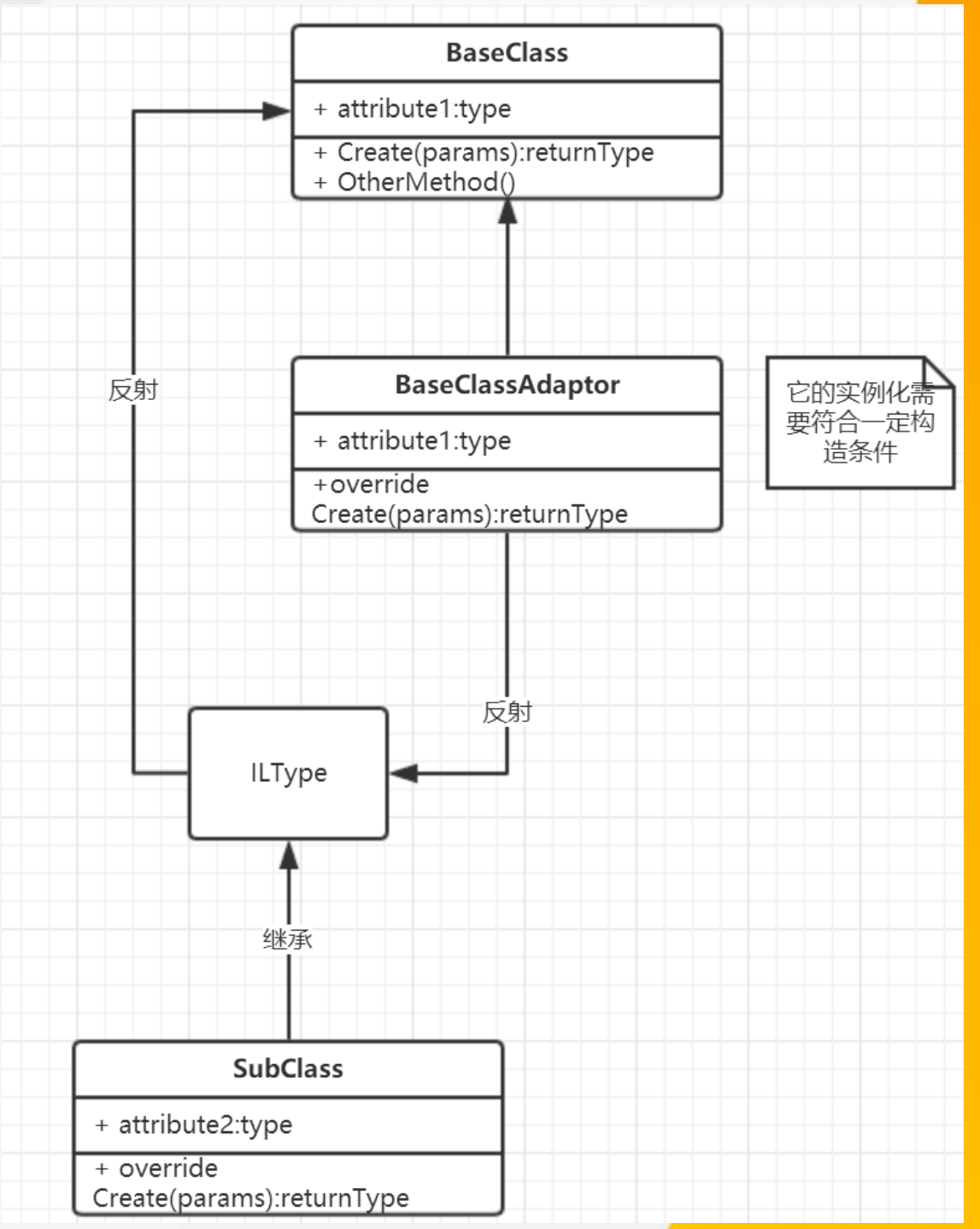
7.2 架构原理

- 基本思路
  - 承继
    - 在Lua方案中，一方面因为语言不同，另一方面因为Lua面向过程，所以没有人关注承继的问题。
    - 在C#方案中，会很自然地遇到：热更类会承继原生的基类。



7.2 架构原理

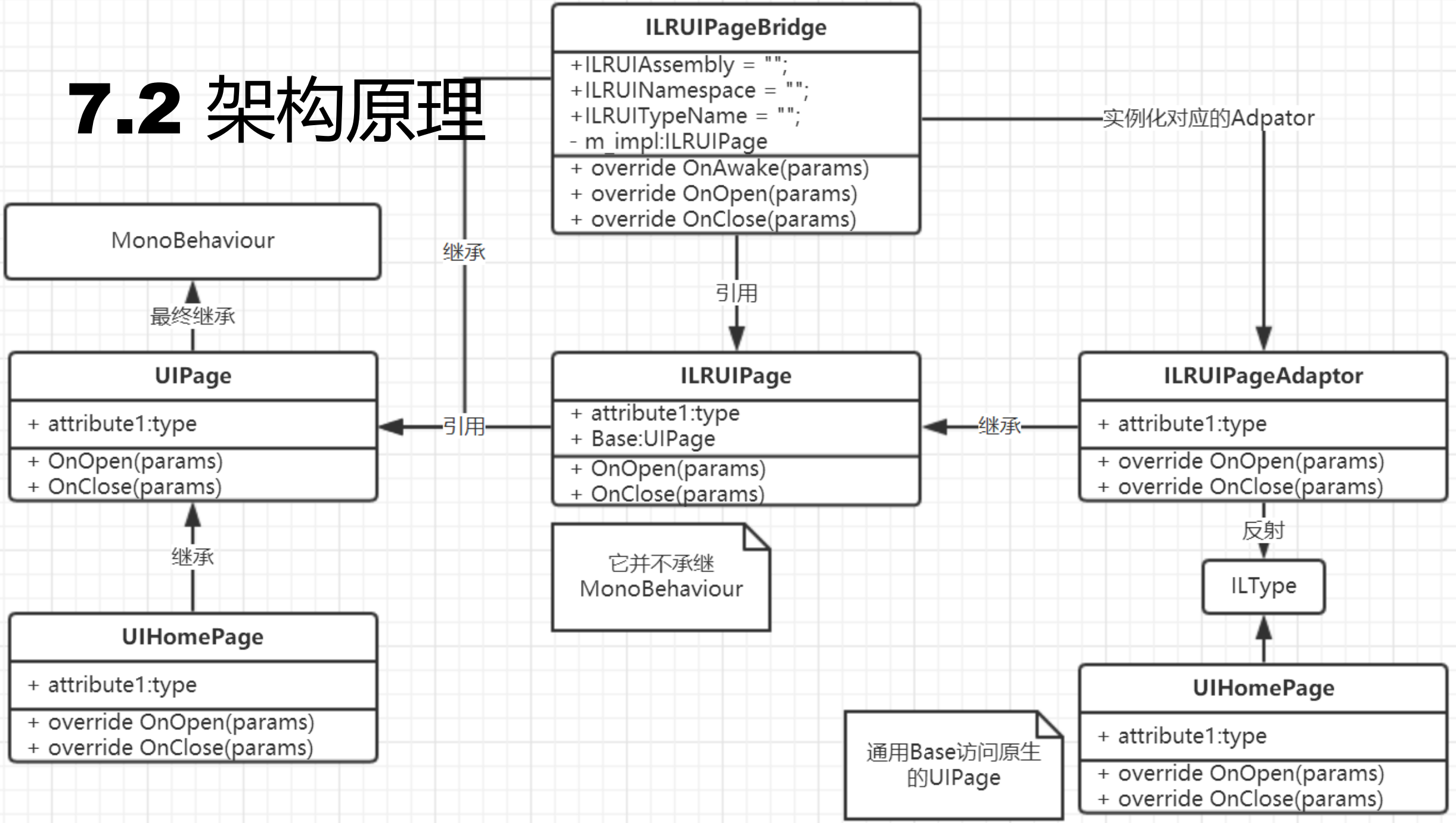
- 继承
  - Adaptor的构造需要一定条件，我们需要自己实例化它。
  - 但是如果BaseClass是MonoBehaviourk怎么办?MonoBehaviour是在GameObject内部构造的！
    - ILRuntime可以对一个原生类的方法进行重定向处理。
    - 比如对AddComponent和GetComponent重定向为自己的实现方式，这样就可以用自己的方式构造Adpator了。
    - 但是没必要这样做~~



7.2 架构原理

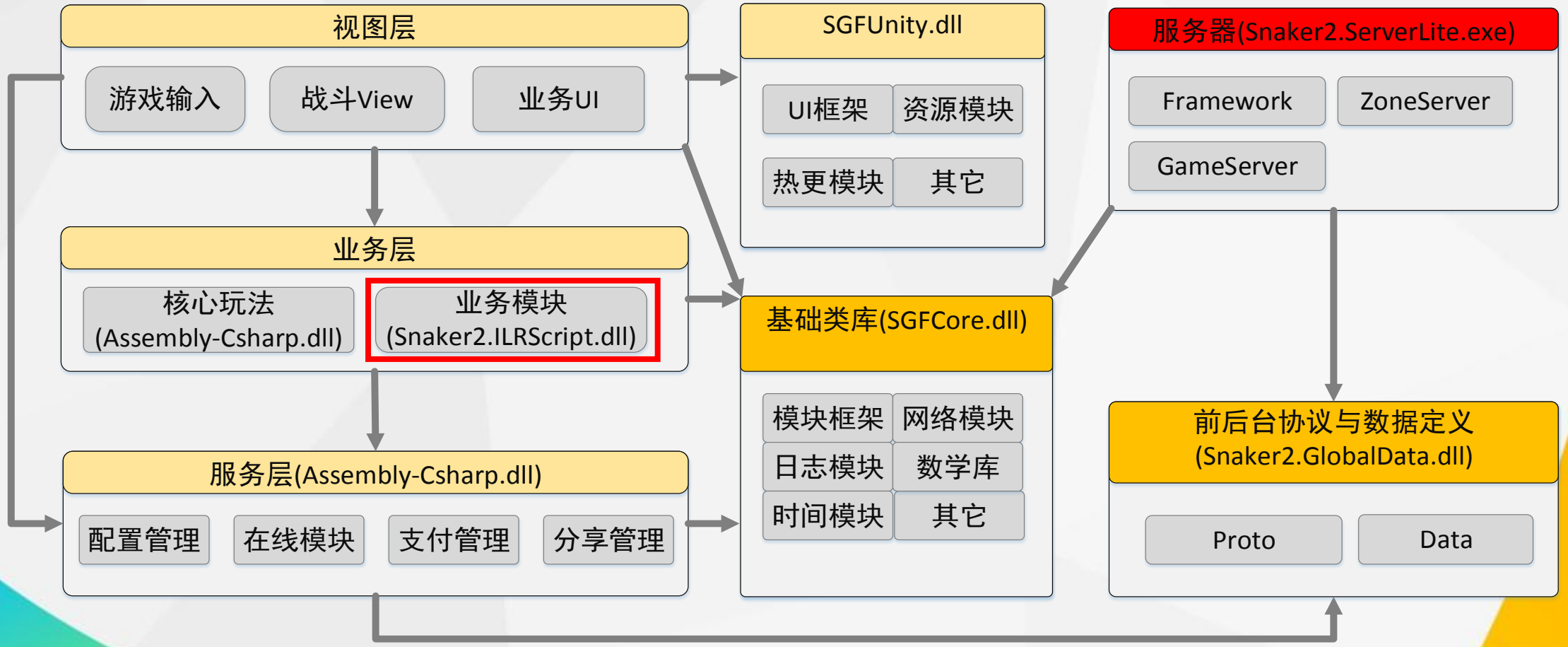
- 模块框架
  - 基于上图的设计已经完全可以满足模块框架
  - 将BaseClass替换为GeneralModule即可
- UI框架
  - 由于UI的基类是MonoBehaviour类，如果不想重定向MonoBehaviour的GetComponent和AddComponent函数，那么只能通过【桥接】的方式实现。
  - 由于桥接的功能有限，子类无法获取MonoBehaviour的全部功能，但是在UI框架里对MonoBehaviour的封装，本身就只是为了使用MonoBehaviour的一部分功能而已。
  - 所以推荐采用桥接的方案。

7.2 架构原理

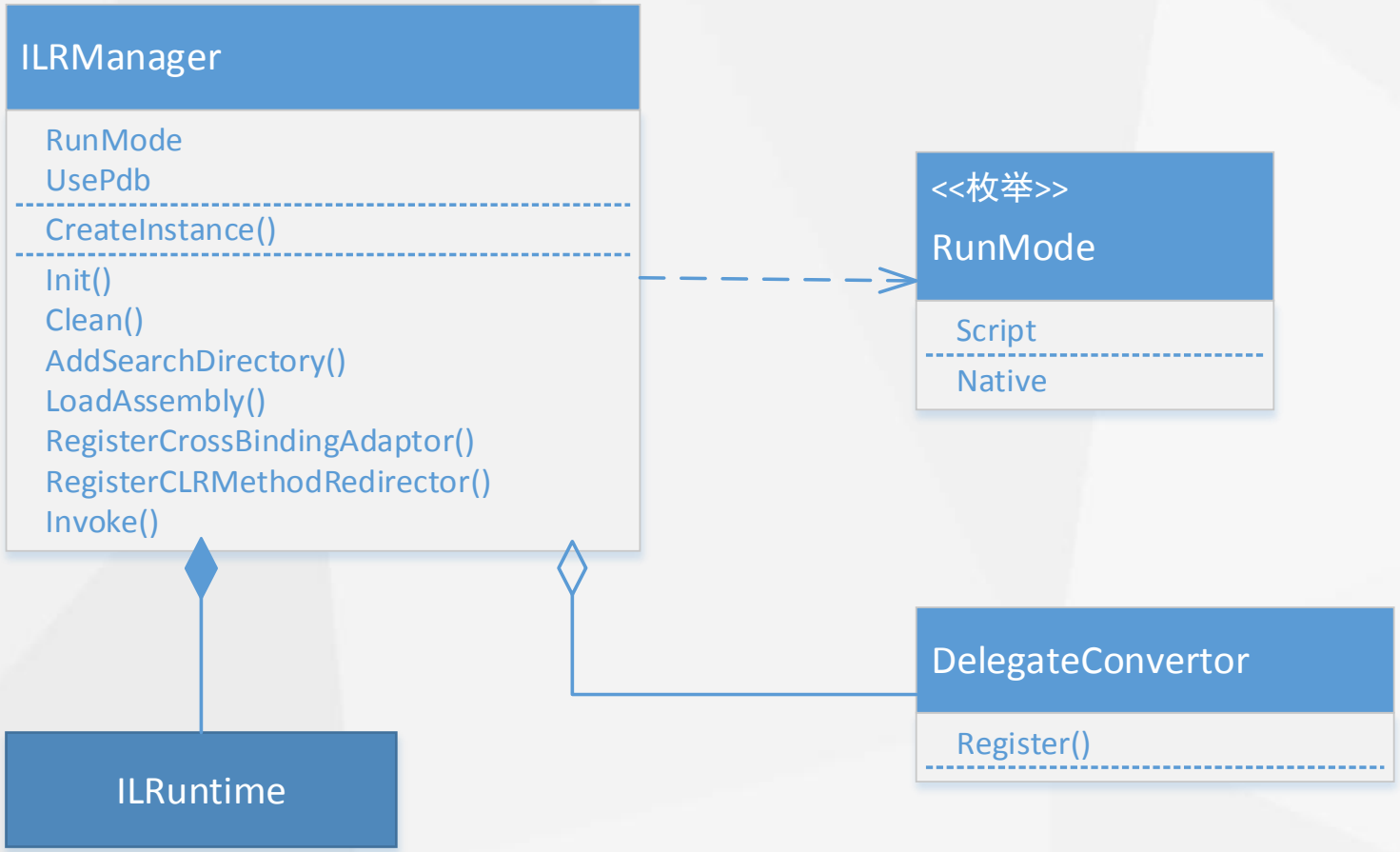

ILRUIPageBridge
+ILRUIAssembly = "";
+ILRUINamespace = "";
+ILRUITypeName = "";
- m_impl:ILRUIPage
+ override OnAwake(params)
+ override OnOpen(params)
+ override OnClose(params)
实例化对应的Adpator
MonoBehaviour
继承
引用
最终继承
UIPage
+ attribute1:type
+ OnOpen(params)
+ OnClose(params)
ILRUIPage
+ attribute1:type
+ Base:UIPage
+ OnOpen(params)
+ OnClose(params)
ILRUIPageAdaptor
+ attribute1:type
+ override OnOpen(params)
+ override OnClose(params)
引用
继承
反射
它并不承继
MonoBehaviour
ILType
继承
UIHomePage
+ attribute1:type
+ override OnOpen(params)
+ override OnClose(params)
通用Base访问原生
的UIPage
UIHomePage
+ attribute1:type
+ override OnOpen(params)
+ override OnClose(params)

7.2 架构原理

可以将所有【业务模块】划分为【热更模块】



7.3 框架设计

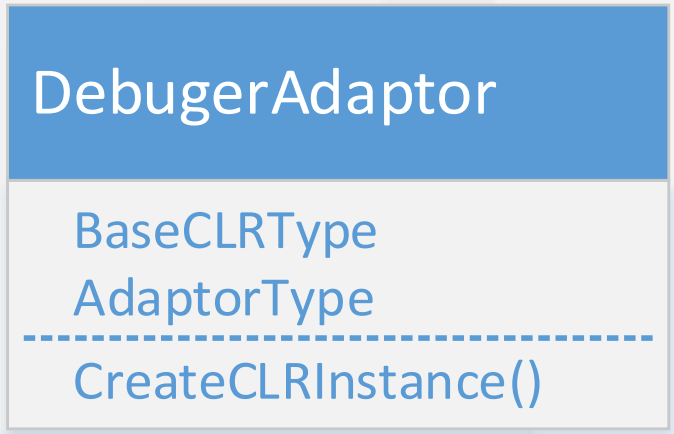

ILRManager
RunMode
UsePdb
CreateInstance()
Init()
Clean()
AddSearchDirectory()
LoadAssembly()
RegisterCrossBindingAdaptor()
RegisterCLRMethodRedirector()
Invoke()
<<枚举>>
RunMode
Script
Native
DelegateConvertor
Register()
ILRuntime

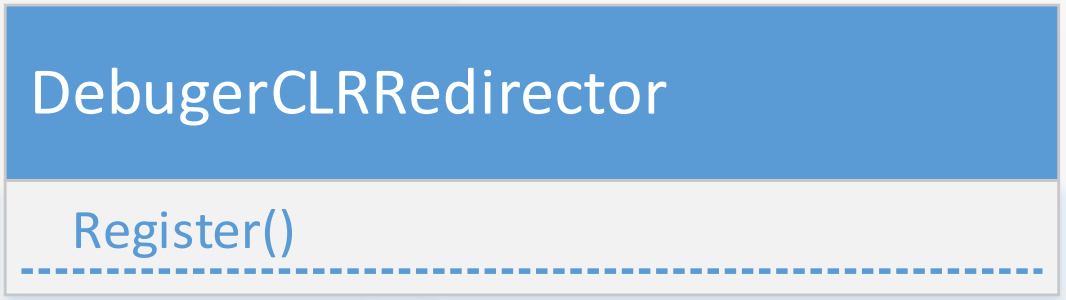
7.3 框架设计

- 关键概念
  - CrossBindingAdaptor
    - 用来实现跨脚本继承。
    - 即，假如要在Script中ClassA继承Native的类ClassX，需要实现一个ClassXAdaptor
    - 在Script中实例化ClassA时，Native侧获取到的是ClassX的Adaptor。通过Adaptor，Native侧的调用可以调到Script侧。
  - CLRMethodRedirector
    - 用来重定义Script侧调用Native侧的函数。
    - 可以利用该方法提高特定函数的调用性能。
    - 也可以利用该方法在函数调用中注入有用的信息，比如Debuger信息。

7.3 框架设计

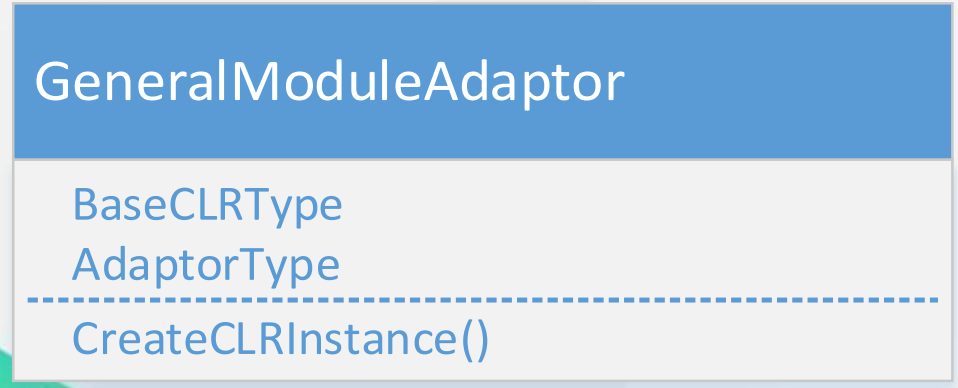
- 对Debuger的支持
  - DebugerCLRRedirector
    - 用于在Script侧调用Debuger.Log时，输入正确的日志信息
  - DebugerAdaptor
    - 为了支持在热更类中继承ILogTag这个接口

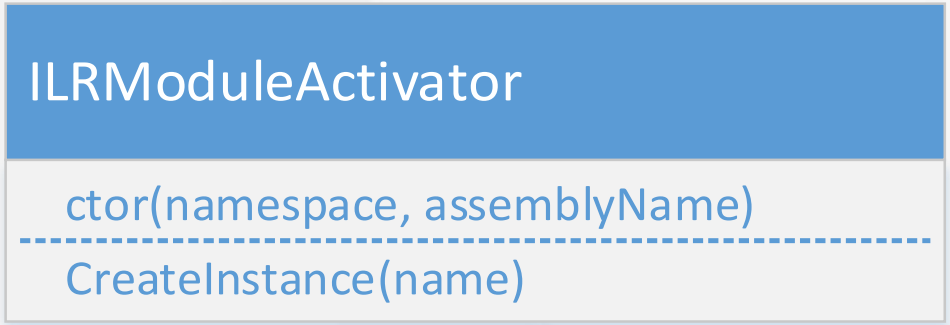




7.3 框架设计

- 对模块管理器的支持
  - ILRModuleActivator
    - 用于在Native侧实例化Script侧定义的Module
  - GeneralModuleAdaptor
    - 用于支持在热更类中继承GeneralModule





7.3 框架设计

- 对UI管理器的支持
  - 已经在架构原理中专门介绍了

7.4 编码实现

- 核心模块
  - ICLRMethodRedirector
  - RunMode
  - ILRManager

7.4 编码实现

- 扩展实现
  - 支持Debuger
    - ILogTagAdapter
    - DebugerCLRRedirector
  - 支持Delegate
    - DelegateConvertor
  - 支持Module
    - ILRModuleActivator
    - GeneralModuleAdapter

7.4 编码实现

- 扩展实现
  - 支持UI框架
    - ILRUIPanel, ILRUIPanelAdaptor
    - ILRUIPage、ILRUIPageAdaptor
    - ILRUIWindow、ILRUIWindowAdaptor
    - ILRUILoading、ILRUILoadingAdaptor
    - ILRUIWidget、ILRUIWidgetAdaptor

7.5 示例及练习

- 示例
  - 以HomeModule为例，将其移至热更DLL中

7.5 示例及练习

- 练习
  - 将RoomModule移植到热更DLL中

8 课程总结与展望

THANKS