**UnityClient2.0架构设计**

**选型考虑**

随着Unity进入2018版本，Android进入8.0， IOS审核门槛越来越高，现有的客户端技术整体方案已经越来越适应不了新的需求变化，主要体现在几个方面：

1、Android64位版本的要求(google play, 2019 deadline), Android的I2CPP的迫切需求，可能导致Android的CSharp更新方案面临比较大的挑战。

2、IOS,Android 双端互通的迫切性，对全平台的代码更新需求越来越强烈。

3、新版本Unity带来越来越多通用的框架功能， 而我们普遍使用的NGUI在Unity新版本存在各种兼容和性能内存的问题，也迫切需要切换到新版本的组件框架。

所以新的客户端架构重点解决两个方面的痛点需求：

1、全平台代码更新

2、选择主流框架，包括UGUI， 剧情编辑器等。

下面的内容主要介绍代码更新相关的方案

**代码更新的选择**

**ILRuntime**

优点：C# 开发

缺点：很少有大型项目使用，性能是硬伤

**Lua**

优点：性能高，广泛使用在各种商业项目上

缺点：lua语言本身作为一个弱类型语言，在开发大型项目的时候，无论是工具链上（编辑器和调试工具），语法层面的支持上都要和C#差距甚远。开发效率和维护是痛点

有没有更好的方案和参考选择？

**C# 开发，翻译成Lua**

@类似的做法行业类也有大规模这么干的，参考javascript和Typescipt的关系

除了解决上面说的开发效率问题，还有一个优势是lua和C#的无缝切换，开发出来的游戏，发布的时候，可以选择c#版本还是lua版本，这样带来的灵活性有着太多的好处，我们可以针对 Android平台选择c#版本发布,而IOS则选择用lua版本去发布。

另外，从长远来看，整个团队的研发积累，都是基于C#开发，也就是Unity的官方开发语言，无论后续的引擎版本怎么演化，IOS多么变态的政策出来，至少有一个技术底线，我们的积累可以无缝的跟上行业的趋势发展。

**技术可行性**

研究了目前的开源方案，有两个方案可以选择：

**CS2Lua和Slua方案**

翻译代码和机制比较复杂繁琐，逻辑代码需要控制的地方太多，协作开发起来，会有很多坑，会花大量的时间在处理兼容性上。

另外，作者适配了slua，tolua，这两lua虽然是开源项目，但作者维护的精力和更新力度比较有限，目前仍然使用Lua5.1版本

**CSharp.lua方案**

<https://github.com/yanghuan/CSharp.lua>

https://github.com/yanghuan/CSharpLuaForUnity

架构和设计最专业和简洁的开源项目，可扩展性非常强大，缺点是没有针对unity去适配，这些需要自己去做所有的适配工作。

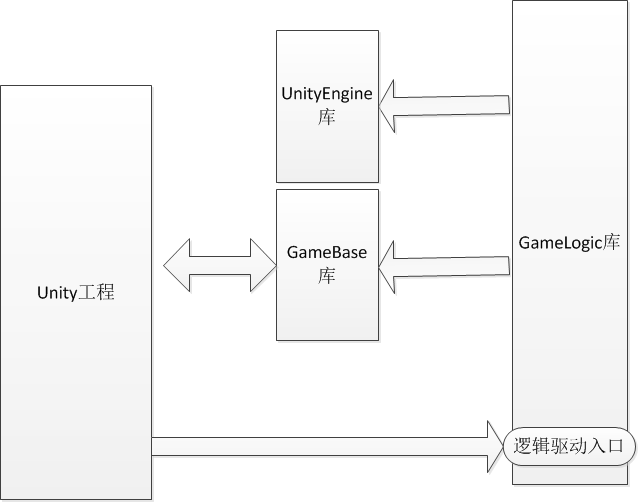
另外，对于Lua引擎的方案，个人还是更加偏向于XLua, 主要原因是Xlua有腾讯专业的团队在维护，对于一些bug和lua版本的跟进会比较及时，性能和优化各方面都有比较好的保障。

另外Xlua现在用的Lua5.3版本的引擎，引擎默认支持utf8和big64。

综合考虑，最终选择了 CShapr.Lua和XLua的方案，虽然适配的工作辛苦一点，但基本上适配完后，团队也可以完全掌握这套工具链了，由于其优异的设计和代码质量，后期的维护和项目的跟进也会快捷很多。

**整体工程设计**

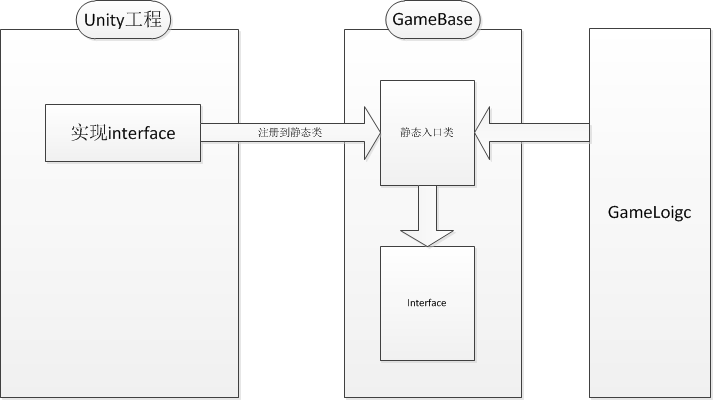
为了保持翻译工具链的干净和可控，同时能够尽量避免Unity本身的各种功能特性带来的耦合（比如协程,SDK等等），这儿将lua的逻辑层代码和Unity工程代码进行拆分出来，改为独立的代码库开发，避免Unity工程代码对lua代码库的污染。



**GameBase的结构设计**

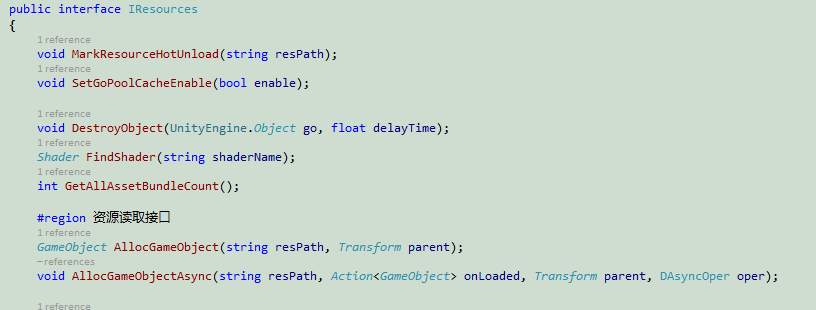
这个库主要是为了设计一个代理层，把Unity工程里的工作封装成统一的接口，然后开放给GameLogic使用，对GameLogic屏蔽了大量的Unity工程细节和函数接口。

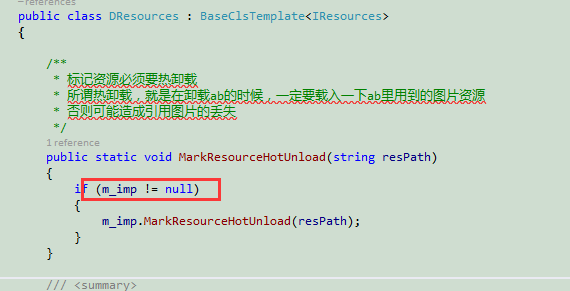
那么如何去设计Base库里的代码呢，可以让GameLogic依赖Base库，同时Base库在不直接依赖Unity工程的情况下能封装这些代码。下面是典型的设计模式，其实很简单：



举个资源管理的范例，封装资源相关的接口

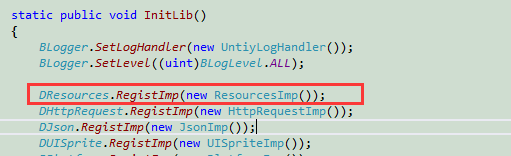
**GameBase**库里的代码，可以看到，没有任何Unity工程里的依赖





**在Unity工程里**

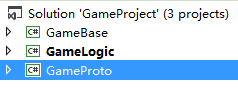




把功能的实现注册到BaseLib

**GameLogic工程的拆分**

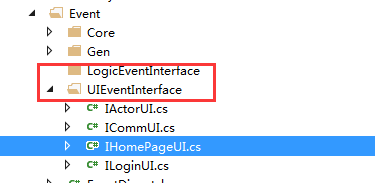
由于协议和配置生成的代码比较庞大，如果每次转lua都要一起转的话，效率将会非常的低下，所以把协议和代码拆封成单独的项目库GameProto, 最终开发相关的库为三个：



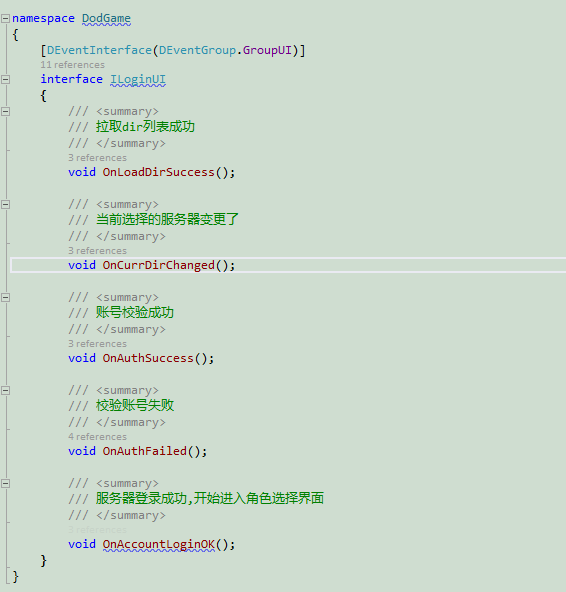
**备注：拆分后，GameLogic本身的编译也快了很多，节省不少等待时间**

**新的消息机制**

传统的消息方式是采取消息ID, 加一堆泛型参数，来隔离游戏模块之间的耦合，但随着消息的数量越来越多，特别是UI和逻辑层的复杂交互，变得也越来越难维护和管理，特别是接口的参数约束也不是很清晰，理解成本普遍偏高，所以重新设计了新的消息机制，对消息进行分组和统一管理。



全局的消息分为两大类，一个是UI的消息，另外一个是逻辑模块之间的消息，新的消息改为定义接口的方式，比如ILoginUI，定义了启动到登录相关的所有消息通知：

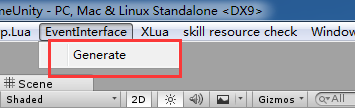


整体的开发流程分为下面几部：

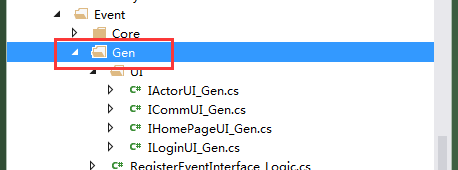
1、定义好接口，注意一定要在接口的上方加上Attribute (DEventInterface)，这样代码生成工具才能识别这些接口。

定义好后，编译GameLogic库，一定要编译成功后，才能执行下面的步骤

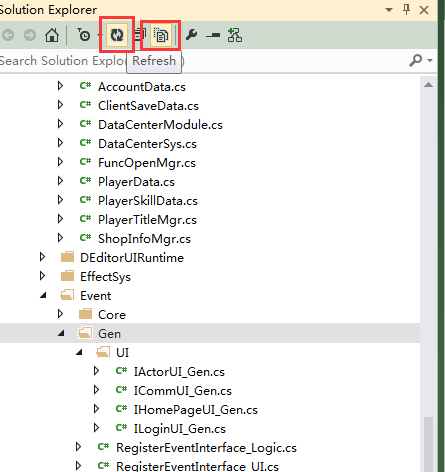
2、打开Unity编辑器，运行Generate,生成中间代码

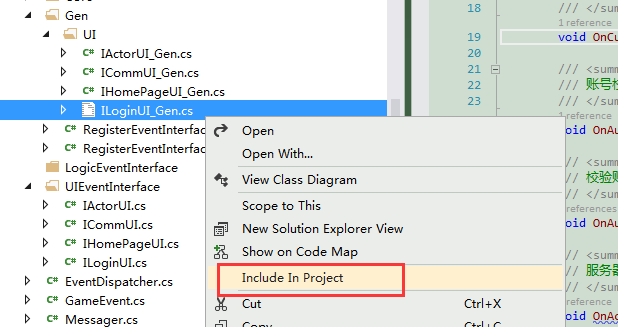


生成的代码全部在这个目录



切换到GameProject工程，把新生成的代码加入到工程里，看不到的，可以点击下刷新，然后把新的文件include到项目路





3、定义消息处理代码

有两种消息处理的方式

和以前的一样，单个消息注册：

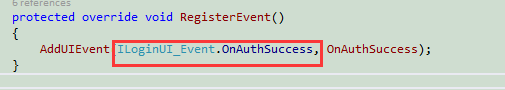
GameEvent.AddListener()

如果是界面窗口里的代码，可以调用

AddUIEvent, 在界面关闭的时候，会自动释放消息注册

具体可以参考下项目代码的ILoginUI相关的, 事件ID在第二步里已经根据接口自动生成了，规则是

接口名称\_Event.函数名称, 比如下面：



4、发送消息

GameEvent.Get<接口>().函数调用, 明确的传递参数类型等信息

clipboard.png

提示：如果需要修改interface，定义新的函数，会出现编译错误，主要是gen的代码没有实现新的函数接口，这个时候到gen的类里，简单的添加下函数定义，先编译过，然后再generate重新生成最新的代码,就可以了