

《SimpleFramework_v0.2.7f2 注解—完整版》

作者：骏擎 CP（2015/04/11）

非常感谢大家使用 SimpleFramework，从上一次写文档注解（v0.1.5）到现在已经快半年了，因为业余时间有限，所以很抱歉让大家等了这么久。因为之前已经有 v0.1.0/v0.1.5 有一部分的注解了，所以这一版本我可能比较简略的说明下核心的部分内容。如果里面缺少的部分，出门左转：<http://ulua.org/simpleframework.html>，v0.1.0/v0.1.5 的注解文档。推荐先看那个结束后再看这个。也非常感谢那么多游戏厂商 CP 们对 ulua 的信任，查看众多上线商业案例地址：<http://www.ulua.org/showcase.html>，开发中的游戏上百款~~ 国内大厂商众多大作都已经使用此技术的公司：中手游、巨人网络、完美世界、扬讯等等。。中小厂商无以数计。

（1）ulua 是个什么玩意儿？

ulua 是一个 2014 年初，我为了自己的游戏《Q 灵三国》提供一个热更新方案，在 Unity 官方论坛上找到的一个老外开发的热更新插件，当时的版本是 1.02，后来发布 1.03 只是增加了源码部分(位置在 ulua/Source/)，功能未有增加。我当时在游戏上线以后，在 Unity 圣典上分享了一篇帖子介绍 ulua，没想到得到了众多同学的回应，以此形成了现在 ulua 的庞大社群。

到了 2014 年下半年，作者在发布源码后，就不再更新功能了，为了使 ulua 与时俱进，因此在这个社群中，众多的使用同学，齐心协力自己完善 ulua，增加更多的功能与修缮。里面最为大的功臣应该是 topameng 的 cstolua，他在 Unity 热更新领域，首次使用了众多的去反射，使用 lua 堆栈来跟 c# 进行交互操作，后来用 lua 提供众多常用类的重写（Vector2/Vecotor3/ray 等）极大的提升了 ulua 的运行效率。

为了增加 ulua 的功能性，可以让 lua 使用很多 c# 相同的功能（json,sqlite,protobuf 等），我先后增加了 pb-lua\pbc\cjson\lpeg\sqlite 众多 c 语言的成熟组件。再次避过 c# 与 lua 交互的 P/Invoke 低效，提升 lua 的效率。使 ulua 已经从一个普通的插件，进化成一个开发环境，周边的工具链都提供了。

以及苹果去年提出的需要 iOS64 的支持需求，我联合群里大牛们，一起攻克了支持 iOS64、luajit2.1 在 iOS64 上编码不兼容问题、Android Intel X86 平台的支持、luajit 与 MacUnity5 的冲突等等技术难关。

因此，从 ulua 1.05/1.07/1.08 以及后面的版本都出自 Unity uLua 技术讨论群 341746602，也当之无愧的变成了 ulua 的新的作者。里面每个人都为 ulua 发展提供了众多自己的建议跟贡献，尤其不得不让大家记得这些分享自己技术的大牛们：**topameng、最后的骄傲、Chiuan、李学武、rickyPu、蜀山、yefeng、江南**等等，还有众多提出问题的、愿意无数次的测试同学们。

（2）为啥要用热更新？热更新的必要性是什么？

（开发期）减少手游打包次数，大大缩短开发周期，提升程序调试效率。

（运营期）减少大版本更新次数，可以减少用户流失，抱怨，提高留存，付费等。

(3) Ulua 的组成部分有哪些？

<A>ulua 的底层库部分

Ulua前期的比较简单，luajit+luawrap.c，提供了基础的lua交互访问功能，到了后期，我们还是维护后，因为不同的需求，我们开始丰富了众多功能，比如，protobuf-lua-gen、pbc、cjson、lpeg、sqlite。而这一部分由于大家需求各不相同，很多同学总是跟我索取，目前已经共享了编译工程，大家可以根据自己的需求自定义编译的底层库，去掉不需要库，节省内存与尺寸大小。工程下载地址：<http://pan.baidu.com/s/lgd1Wyx9> mac、ios是luavm，需要xcode6.0以上，其他的平台是基于luajit0.2.3，编译安卓需要ndk r8d，windows下编译工具mingw32/64。

cstolua 原理简介

Cstolua 是大神 topameng 的经典之作，其实我更喜欢给他另一个名字 tolua c#，因为他的作用类似于 cocos 的 tolua++，当然我也尊重作者的意愿，叫它 cstolua。它的原理就是将原来 ulua 反射调用方式改变成去反射的 wrap 方式。后面解释了反射与去反射，简单的使用方式：cstolua 需要我们把要生成 c#的 wrap 类添加到 Editor/BindLua.cs 里面，形式如 _GT(type("C#类名")), 然后会在 Source/Luawrap/下面生成相对应的 xxxWrap.cs 类文件，并且会自动添加到索引类文件 Source/Luawrap/Base/ LuaBinder.cs 一个 Bind 方法里面，当 lua 虚拟机启动的时候会调用这个 Bind 方法，使其在 luavm 中注册相应的 Table，并且将里面的字段、方法等信息添加进去，这样 lua 就可以访问此 c#类了。

<C>lua 脚步部分

新版的 cstolua 在 Lua 目录下面添加了众多的 lua 类，比如 Vector3.lua/ray.lua/Timer.lua 等文件，原因是，由于 c#的一些结构信息如果到 lua 需要装箱、拆箱操作，而这些会产生效率问题，作者为了解决这个问题，用 lua 重写了这些类，避过了装箱低效问题。随着 lua 周边功能库的增加，我也随之添加了部分 lua 文件。

(4) Ulua 还有哪些坑，解决方案是什么？

Ulua 发展到现在，我们已经帮大家填掉了绝大部分的坑了（效率、功能），那你会问，那到现在还有什么坑呢？我负责任的说，目前还存在一部分问题，但是，绝不会影响使用。

[1]Ulua 起初基于 luajit，后来到了去年苹果非要所有 app 支持 iOS64 平台，但是 luajit0.23 版本不支持，后来作者开发了 luajit2.1 测试版，我们开始集成这个版本，后来发现这个版本在 iOS64 上运行，根本无法加载在 PC/MAC 上 luajit 以前编译的 lua 编码文件，也就是说需要区分 2 套编码文件，后来通过作者邮件列表才得知，如果需要正常运行，需要在 iOS 环境编译才行，然后我就做了个工程 luaConvert，让测试同学在 iPhone6 上进行编码运行，才得以通过，虽然难关攻克了，但是此方式实在蛋疼。谁愿意将大把的工程这样编码打包呢。后来我们只能将 lua 虚拟机换成 lua 原生的 vm，这样才统一的 lua 编码格式。

[2]再后来，我们终于等到重构一年的 Unity5 发布了，又有同学发现，在 MAC U5 编辑器模式下，一运行工程，就直接崩溃掉了，后来学武同学找到问题所在，原来新版的 MAC U5 编辑器跟 luajit 的内存分配冲突了，互不相容。我们只能在 iOS 之后将苹果的 ulua 底层库换成了原生的 luavm 虚拟机，才得以攻克。

[3]这样还未结束，我们知道既然更换了底层库的 lua 虚拟机，那我们的编码文件只能换成 luac 去编码了，但是，luac 常规编码文件是可以被反编译的，比如：刀塔传奇的初期被人反编译的倒霉案例，大家还言犹在耳吧。那该怎么办？我们可以修改 lpcode.h 的头部分信息解决加密问题。这样除了 MAC\IOS 使用原生 luavm 之外，其他的是 luajit。

[4]那还有没有别的解决方案了？有，等 luajit2.1 成熟吧。

（5）游戏已经开发到后期，如何接入 ulua？

（1）活动。

活动这部分变数最大，很多问题上线前是无法预知的，比如上线如果发生数据不理想，或者非常火爆，这些情况无法预知，根据这些情况做活动调整，这些很容易有更新需求。而且未必前期都能想到坐进去。运营策划都是要根据在线运营情况做未知的活动调整。

（2）计时器（单位秒）驱动 lua 的 update

还有一部分我称之为，程序给自己留的后路，如果绝大部分都是 c# 的话，很有可能产生上线后产生 bug，比如：新手引导，在什么地方卡住了等。客户端启动一个计时器，驱动 lua 的一个 onTimer，在里面根据游戏运行情况，动态调整对游戏的控制。

（3）网络管理留给 lua 能跟服务器交互的接口（现在未必用得着的）。

还有就是多给自己留一个协议的接口给 lua 备份用。

（6）Unity5 跟 NGUI 的不兼容。

U5 的新格式打包，NGUI 不支持，作者也不准备修复（让我高兴，自己改），老的打包函数又被标记丢弃，想想 Unity3.x 时代的 armv6，Unity4.x 时代的 AddComponent(“name”) 的函数重载，怎么看这些老版本 PUSH\POP 打包 API 都有被抛弃的可能，再转可能就有很大风险。所以新的项目推荐使用 UGUI，框架已经有 UGUI+UNITY5 新打包版本，地址：<http://ulua.org/simpleframework.html>

不支持部分：由于 U5 不再允许直接拖拽脚本到关联变量上，现在的方法是关联 GameObject，然后通过对象获取脚本组件。新版 Assetbundle 中共享图集的 prefab 不能有拖拽脚本，否则只要一加载依赖 Assetbundle，Unity 就会抛出异常，NGUI 里面哪个组件没有这个 atlas 拖拽脚本？没人自己愿意修改每个组件的 Atlas 变成 GameObject 吧？？？

（7）反射与去反射的分析考量

反射是 ulua 的开始运作方案，lua 通过查询 c# 对象的类信息，获取里面类成员变量跟成员函数，然后调用其中的数据与方法。众所周知，反射是有低效的，所以众多热更新方案，都在拼命的去反射，以此来提升效率，这也是 cstolua 的原理。但是有一个问题是，完全去掉反射真的好吗？有人说，我可以把所有的 c# 类都生成 wrap 给 lua 调用啊，嗯，那也倒是真的，但是你会把所有的 NGUI 的类都生成 wrap？或者其他的插件类都生成 wrap，如果 lua 虚拟机里面一次性加载这么多文件，不占内存吗？显然不是，每个 wrap 进入 luastate 都会生成一个 lua 的 table，生成的 wrap 越多，table 也越多，用不用的上是个问题，反正你得一次性注册进去。那就说，我少加一些，这样效率就提升了，那好，如果游戏运营期出现 bug（谁敢说自己的程序不会出现 bug，如果你真能做到无 bug，热更新也就可以去死了），你想调用一些无所预知的类信息，这时候靠的就是反射，反射可以弥补去反射的缺憾，因此你还

是觉得完全去反射真的好吗？还是看起来美丽？所以个人见解，反射+去反射合并使用是最佳方案，去反射解决已知效率问题，反射用来搞定未知问题。将来大版本更新的时候，再把反射的部分用去反射代替掉，反射再为了新的未知 bug 做准备。目前提供反射与去反射的 lua 热更新方案，只有 ulua。

（8）Lua 文件打包方案考量

目前 lua 的加载打包方式有几种：

<1>dofile: 这是目前框架采用的打包方式，这种方式最简单，别管是 lua 代码明文，还是经过 luajit、luac 编码过的文件可以直接被它加载进 luavm，但是它的问题就是，它是以一种物理文件形式存在于玩家手机磁盘上的。现在苹果歇斯底里的在反代码加载。在审核期，有些同学比较担心被苹果 fire 掉。

<2>dostring: 这种方式可以直接加载 lua 的代码明文，或者经过编码过的 bytes 都可以，缺点是，你得有一种存放数据的格式，或者是 txt 文本加载，或者是通过加载 bytes 字节数组加载。那如果我将编码过的、经过加密过的二进制 bytes 文件打包进入 assetbundle 后，苹果还能不能发现这是 lua 源码？那它岂不是先要把所有的加密算法都破解掉，才能知道？所以我个人觉得，苹果知道的可能性几乎为 0。

（9）框架的组成部分

我们看下框架的组成部分，及其介绍一下关键类文件的作用。

<1>Editor/BindLua.cs

这个类是 cstolua 的生成 wrap 的绑定文件，前面介绍过，我们将 c#类添加到里面的一个静态变量 `static BindType[] binds` 的数组里面，并且点击“`Lua/Gen Lua Wrap Files`”来生成 wrap 文件，并且在 LuaBinder.cs 添加 wrap 文件的索引。并且还存在清除 Wrap 文件的功能“`Lua/Clear LuaBinder File + Wrap Files`”。这里需要特殊说下的是菜单项“`Lua/Gen u3d Wrap Files`”，这个功能阿蒙是想一次性把 u3d 的 c#类都生成 wrap 类，但是事实上出现了众多问题，比如，在编辑器模式下运行很正常，但是到了生成安卓，或者苹果 IOS 打包的时候遇到了很多如：Assets/Source/LuaWrap/xxxWrap.cs(218,21): error CS1061: Type `UnityEngine.xxx' does not contain a definition for `xxx' and no extension method `xxx' of type `UnityEngine.xxx' could be found (are you missing a using directive or an assembly reference?) 编译错误解决方案：因为 unity 不同平台版本 API 各不相同，因此不是特别统一，遇到此问题直接删除错误代码即可。如果完全生成所有 u3d 的类，那这个不兼容的 API 就会很多，所以这个菜单项，请慎用。

<2>Editor/Packager.cs

这个类也是个工具类，它目前包含了 2 个功能：

[A] `Game/Build Bundle Resource`

这个功能是一个非常简化的资源打包功能，虽然小但是也包括了依赖包的打包流程，还有 Lua 文件等众多文件的复制到资源目录下，并且生成相对应的文件列表文件 files.txt，并且计算出每个文件的 md5 值，用于资源更新时候的比较操作。

[B] `Game/Build Protobuf File`

这个功能是简单的演示怎么样在编辑器模式下，通过 Windows 版本的 luajit 进行 lua 文件的编码操作。其他平台，比如 MACOSX，需要做相对路径的调整。

<3>Source/Base/LuaScriptMgr.cs

这个类是对 cstolua 及其 LuaInterface 的一个封装,我们需要在资源更新到最新版本以后创建类实例,并且保存引用,供其他类(BaseLua.cs 等)频繁调用。如果作为初级使用层面的话,我们无需知道它太多内部细节,我们只需要知道它有 3 项非常重要的功能:

[A]dofile: 加载 lua 文件到 luavm。

[B]dostring: 加载 lua 代码进入 luavm。

[C]callluafunction: 调用 lua 的函数。

通过上面 3 项功能,我们基本上就可以很简单的使用 ulua 了。

<4>Source/LuaWrap/Base/LuaBinder.cs

这个类的功能比较简单,就是一个 Wrap 类列表索引文件,里面包含了一个 bind 函数,当 LuaScriptMgr 初始化的时候,它会被调用将所有的 wrap 文件 Register 进 lua 虚拟机里面。

<5>Scripts/Common/BaseLua.cs

这个类是 c#层与 lua 层中间的一个粘合剂,俗称“桥”。几乎每一个被 lua 操控的 GameObject 身上都要附加它,它内部完全按照 c# MonoBehaviour 的执行流程,并且在相对应的函数事件内,调用 lua 文件中的函数。

BaseLua.cs	—>	xxx.lua
C# Start()	—>	Lua Start()
C# OnClick()	—>	Lua OnClick()
C# OnDestroy()	—>	Lua OnDestroy()

如上面对照图所示,当附加在 GameObject 身上的 BaseLua.cs 被执行的时候,它都会将其对应的事件 POST 给 lua,这样 lua 层面也就有了事件驱动,也就可以在各个事件里面做相应的操作。

<6>Scripts/Common/GameManager.cs

每一个游戏都要有一个游戏管理器,它也是游戏的初始化入口,这个类继承自 BaseLua.cs,因此它具备了父类的 CallMethod 等功能。并且执行流程如下所示:

<A>GameManager.Init 初始化操作 ->

 GameManager.CheckExtractResource 释放资源 ->

<C> GameManager.OnUpdateResource 更新资源 ->

<D> Util.Add<ResourceManager>(gameObject) 添加资源管理器 ->

<E> ResourceManager. Initialize 资源管理器初始化操作 ->

<F> GameManager.OnResourceInited 初始化完成,创建 LuaScriptMgr,加载 lua 脚本 ->

<G> [game.lua] GameManager. OnInitOK -> 调用 game.lua 中的初始化完成创建面板 ->

<H> [function.lua] CreatePanel -> 调用 function.lua 中的创建面板,回调面板管理器功能 ->

<I> PanelManager. CreatePanel 调用资源管理器加载素材->

<J> ResourceManager.LoadBundle 加载 Assetbundle,回调面板管理器 ->

<K> PanelManager. StartCreatePanel 载入 Prefab,并且实例化面板对象,添加 BaseLua 到对象上 ->

<L> BaseLua.Start 对象开始调用 Start 函数,并且调用 lua 中的 Start 函数->

<M> [PromptPanel.lua] PromptPanel.Start 初始化面板,并且给按钮添加单击事件 ->

<N> BaseLua. OnDestroy 面板销毁时候,清除事件监听,卸载 Assetbundle。完成流程。

PS: C#调用 lua 函数的形式是通过 [Table.Function]的方式调用，Table 可看作类名，这里就是 PromptPanel，那函数名就是 BaseLua 中 CallMethod(“Start”)中的字符串参数，因为面板对象的名称是 PromptPanel，所以初始化的时候最终调用的 uluaMgr.CallLuaFunction (“PromptPanel.Start”)。PromptPanel.lua 文件中的 Start 函数就被调用触发了。

(10) 框架代码视图解析（在原来基础上进行修改）

```
/// <summary>
/// 初始化
/// </summary>
void Init() {
    InitGui();
    DontDestroyOnLoad(gameObject); //防止销毁自己

    Util.Add<PanelManager>(gameObject);
    Util.Add<MusicManager>(gameObject);
    Util.Add<TimerManager>(gameObject);
    Util.Add<SocketClient>(gameObject);
    Util.Add<NetworkManager>(gameObject);

    CheckExtractResource(); //释放资源
    Screen.sleepTimeout = SleepTimeout.NeverSleep;
    Application.targetFrameRate = Const.GameFrameRate;
}
```

初始化 GUI 对象，它是个容器，所有的 UI 面板都作为他的子对象生成到下面，所以上来先创建初始化 GUI 对象。接下来添加管理器们，里面有一些基础的管理器，通过 ioo 这个静态类，可以调用到不同的管理器。就是个简洁通道。
并且开始检查资源解包操作：

```
/// <summary>
/// 释放资源
/// </summary>
public void CheckExtractResource() {
    bool isExists = Directory.Exists(Util.DataPath) &&
        Directory.Exists(Util.DataPath + "lua/") && File.Exists(U
    if (isExists || Const.DebugMode) {
        StartCoroutine(OnUpdateResource());
        return; //文件已经解压过了，自己可添加检查文件列表逻辑
    }
    StartCoroutine(OnExtractResource()); //启动释放协成
}
```

当资源解压完成后，开始执行资源下载更新逻辑，来保证本地资源文件是最新的。

```

        message = "解包完成!!!";
        yield return new WaitForSeconds(0.1f);
        message = string.Empty;

        //释放完成，开始启动更新资源
        StartCoroutine(OnUpdateResource());
    }

    /// <summary>
    /// 启动更新下载，这里只是个思路演示，此处可启动线程下载更新
    /// </summary>
    IEnumerator OnUpdateResource() {
        if (!Const.UpdateMode) {
            Util.Add<ResourceManager>(gameObject);
            yield break;
        }
        WWW www = null;
        string dataPath = Util.DataPath; //数据目录
        string url = string.Empty;
    }
}
UNITY_5

```

```

        yield return new WaitForEndOfFrame();
        message = "更新完成!!";
        Util.Add<ResourceManager>(gameObject);
    }
}

```

当更新完成后，开始将资源管理器附加到对象身上，并且启动资源管理器的初始化操作，因为最重要的莫过于资源管理器的加载，它的加载会带动后面逻辑的执行。

```

    /// <summary>
    /// 初始化
    /// </summary>
    void initialize() {
        byte[] stream;
        string uri = string.Empty;
        //-----Shared-----
        uri = Util.DataPath + "shared.assetbundle";
        Debug.LogWarning("LoadFile::>> " + uri);

        stream = File.ReadAllBytes(uri);
        shared = AssetBundle.CreateFromMemoryImmediate(stream);
    }
    #if UNITY_5
        shared.LoadAsset("Dialog", typeof(GameObject));
    #else
        shared.Load("Dialog", typeof(GameObject));
    #endif
    ioo.gameManager.OnResourceInited(); //资源初始化完成，回调游戏管理器
}

```



进入资源管理器的初始化函数，就通过读取 assetbundle，返回字节数组，然后通过流创建出 AssetBundle 对象，而且 CreateFromMemoryImmediate 函数是只有 Unity4.5.2 版本以后才有的新函数，之前版本的话只能通过异步加载到内存。

然后通过 AssetBundle 对象的 Load 方法把名叫 Dialog 的 Atlas Prefab 加载到内存里面。这样以此 Atlas 为依赖的面板纹理等都能正常显示了。这个非常重要，这也是关联资源打包，加载的部分。

处理完这些后，既然都把 Atlas 资源加载到内存中了，那就可以返回 GameManager 做后面的初始化了。当然这 bundle 的内存存在析构函数的时候把内存卸载掉。

```
/// <summary>
/// 资源初始化结束
/// </summary>
public void OnResourceInited() {
    luaMgr = new LuaScriptMgr();
    luaMgr.Start();

    luaMgr.DoFile("game");    //加载游戏
    luaMgr.DoFile("network"); //加载网络

    object[] panels = CallMethod("LuaScriptPanel");
    //-----Lua面板-----
    foreach (object o in panels) {
        string name = o.ToString().Trim();
        if (string.IsNullOrEmpty(name)) continue;
        name += "Panel";    //添加

        luaMgr.DoFile(name);
        Debug.LogWarning("LoadLua---->>>>" + name + ".lua");
    }
    //-----
    CallMethod("OnInitOK");    //初始化完成
}
```

当执行到 **OnResourceInited** 的时候，我们就可以加载 lua 的部分了，相对与之前的老版本的框架，是不是有些更清晰明了呢？的确简化了不少。

在这个函数内部首先给公共变量 LuaScriptMgr 创建对象，只有创建完对象，才能加载后面的 lua 脚本。

```
luaMgr.DoFile("game");    //加载游戏
luaMgr.DoFile("network"); //加载网络
```

这两行代码，就是直接加载游戏管理器的 lua 脚本与网络管理器的 lua 脚本，让他们脚本内容加载进 Lua 管理器里面。但是加载进去并不代表着里面的函数已经被执行了，只有当我们 c# 代码明确的 callfunction 的时候，lua 函数才能直接执行。切记！

```
object[] panels = CallMethod("LuaScriptPanel");!
```


这行代码直接调用 game.lua 里面的 LuaScriptPanel 函数,通过这个函数,得知,我后面要加载的 Lua 脚本面板有哪些。否则,你怎么能动态的修改 lua 文件添加与移除面板呢?当然了,也可以用其它方式,文本或者 www 获取都行,我这里偷懒了。图简单。

```
//-----Lua面板-----  
foreach (object o in panels) {  
    string name = o.ToString().Trim();  
    if (string.IsNullOrEmpty(name)) continue;  
    name += "Panel";    //添加  
  
    luaMgr.DoFile(name);  
    Debug.LogWarning("LoadLua--->>>" + name + ".lua");  
}
```

既然拿到了要初始化的面板数组,那就按个的 DoFile 加载吧,没有什么好解释的。

PS:如果想在任意 c#里面调用不同的 lua 函数,可以用下面代码做到:

```
LuaScriptMgr mgr = io.gameManager.luaMgr;  
mgr.CallLuaFunction("模块名.函数名",参数1,参数2,参数3...);
```

而模块名,就是你在 lua 文件的开头声明的一行语句:

```
module("GameManager", package.seeall)
```

有个问题:最后一行代码是 `CallMethod("OnInitOK");` 没有使用 `mgr.CallLuaFunction("GameManager.OnInitOK")`,其实,这两个是等价的,我的 GameManager 跟 NetworkManager 都继承自 BaseLua 类,这一行是让父类做的 lua 函数调用,实际上在 BaseLua 里面也执行了 `mgr.CallLuaFunction("模块名.函数名")` 相同效果的代码,为了封装性嘛!

其实执行到这里,c#的初始化工作已经算是完成了。接下来就交给了 game.lua 里面的 OnInitOK 函数了,我们 open 看看内容。

```
-- 初始化完成, 发送链接服务器信息--  
function OnInitOK()  
    warn('OnInitOK--->>>');  
    createPanel("Prompt");  
  
    Const.SocketPort = 2012;  
    Const.SocketAddress = "127.0.0.1";  
    io.networkManager:SendConnect();  
end
```

这个里面做了 2 个事情，第一个事情是创建了一个“提示”面板，第二个是像服务器发送 socket 连接请求。我们先来看第一个，怎么通过 createPanel 创建面板。

这个函数在 function.lua 里面，所以在开头的地方有加载

```
require "functions"
```

这里借着说一句，这个 lua 文件的加载依赖于 DoFile，那这个文件放在什么地方呢？打开 Util.cs 类，有个 LuaPath 的函数。里面决定着去哪里读取 lua 文件的路径。

在 function.lua 里面，看下 createPanel 函数：

```
3
4  -- 创建面板--
5  function createPanel(name)
6      io.panelManager:CreatePanel(name);
7  end
```

这个函数直接调用的 io 类，此 io 类非 lua 的 io 类，要分清了，可以自己把名字改了。通过 io 类里面的静态成员变量 panelManager 创建面板。

那问题出来了，io 哪里来的，现在的新版框架已经没有了老版本的 define.lua 的载入函数了，怎么就能使用了呢？这个问题你先别着急，到后面专门讲 cstolua 的使用的时候怎么把类倒进来供 lua 使用。。。。

既然这样，我们就要打开 c#代码的 PanelManager.cs 类，看看它是怎么创建面板的吧。

```
/// <summary>
/// 创建面板，请求资源管理器
/// </summary>
/// <param name="type"></param>
public void CreatePanel(string name) {
    AssetBundle bundle = io.resourceManager.LoadBundle(name);
    StartCoroutine(StartCreatePanel(name, bundle, string.Empty));
    Debug.LogWarning("CreatePanel::>> " + name + " " + bundle);
}
```

我们主要是看创建面板那里，通过 lua 传递过来的名字，请求资源管理器直接载入 Bundle 资源（其实这里相对于老版本也简化了很多），然后通过异步开始创建面板，并且把名字跟从资源管理器里面载入进来的 bundle 资源传递过去。

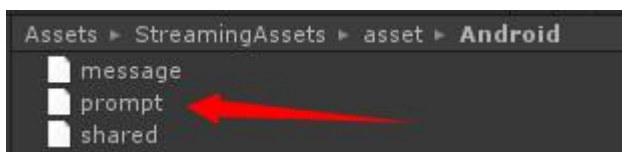
打开资源管理器 ResourceManager.cs 看 LoadBundle 函数。

```

/// <summary>
/// 载入素材
/// </summary>
public AssetBundle LoadBundle(string name) {
    byte[] stream = null;
    AssetBundle bundle = null;
    string uri = AssetPath + name.ToLower() + ".assetbundle";
    stream = File.ReadAllBytes(uri);
    bundle = AssetBundle.CreateFromMemoryImmediate(stream);
    return bundle;
}

```

相对于以前已经非常简化了，这几行代码也没必要说了，主要是 name 参数就是请求的名字，比如从 lua 传递过来的提示面板的名字 Prompt，通过这里直接去加载 prompt.assetbundle 的文件位置：



加载完成了，把素材 bundle 回传给面板管理器，开始后面的异步创建面板逻辑。

```

/// <summary>
/// 创建面板
/// </summary>
IEnumerator StartCreatePanel(string name, AssetBundle bundle, string text = null) {
    name += "Panel";
    GameObject prefab = bundle.Load(name) as GameObject;
    yield return new WaitForEndOfFrame();
    if (Parent.FindChild(name) != null || prefab == null) {
        yield break;
    }
    GameObject go = Instantiate(prefab) as GameObject;
    go.name = name;
    go.layer = LayerMask.NameToLayer("Default");
    go.transform.parent = Parent;
    go.transform.localScale = Vector3.one;
    go.transform.localPosition = Vector3.zero;

    yield return new WaitForEndOfFrame();
    go.AddComponent<BaseLua>().OnInit(bundle);

    Debug.Log("StartCreatePanel----->>>>" + name);
}

```

这里面也在老版注解里面说过了，不重复了，不懂的出门左转，不过我们还是顺着逻辑往下走，通过 BaseLua 的 OnInit 函数进去看看变化。

```

protected LuaScriptMgr luaMgr {
    get {
        if (mgr == null) {
            mgr = io.gameManager.luaMgr;
        }
        return mgr;
    }
}

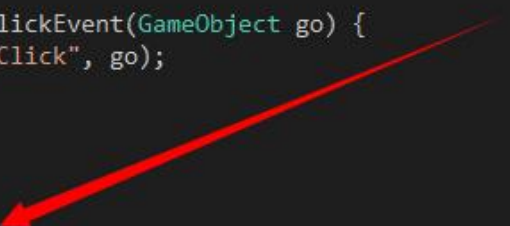
protected void Start() {
    CallMethod("Start");
}

protected void OnClick() {
    CallMethod("OnClick");
}

protected void OnClickEvent(GameObject go) {
    CallMethod("OnClick", go);
}

/// <summary>
/// 初始化面板
/// </summary>
public void OnInit(AssetBundle bundle, string text = null) {
    this.data = text; //初始化附加参数
    this.bundle = bundle; //初始化
    Debug.LogWarning("OnInit---->>>" + name + " text:>" + text);
}

```



还是比以前简化很多了，OnInit 除了把素材 bundle 传递进来，保存起来，也没啥操作了。那后面怎么执行呢？既然在面板管理器里面已经实例化了这个面板，它自然就要执行 Start 函数呗，还用问？？？

那这个面板的 Start 函数就会被执行，我们主要是看看怎么调用 lua 函数，之前老板的框架使用的多 luastate，现在使用阿萌的单 luastate 来维护，因此，当 dofile 的时候，都是从 GameManager 里面取来的 lua 管理器的引用。

我们看一个 Lua 函数调用执行：

```

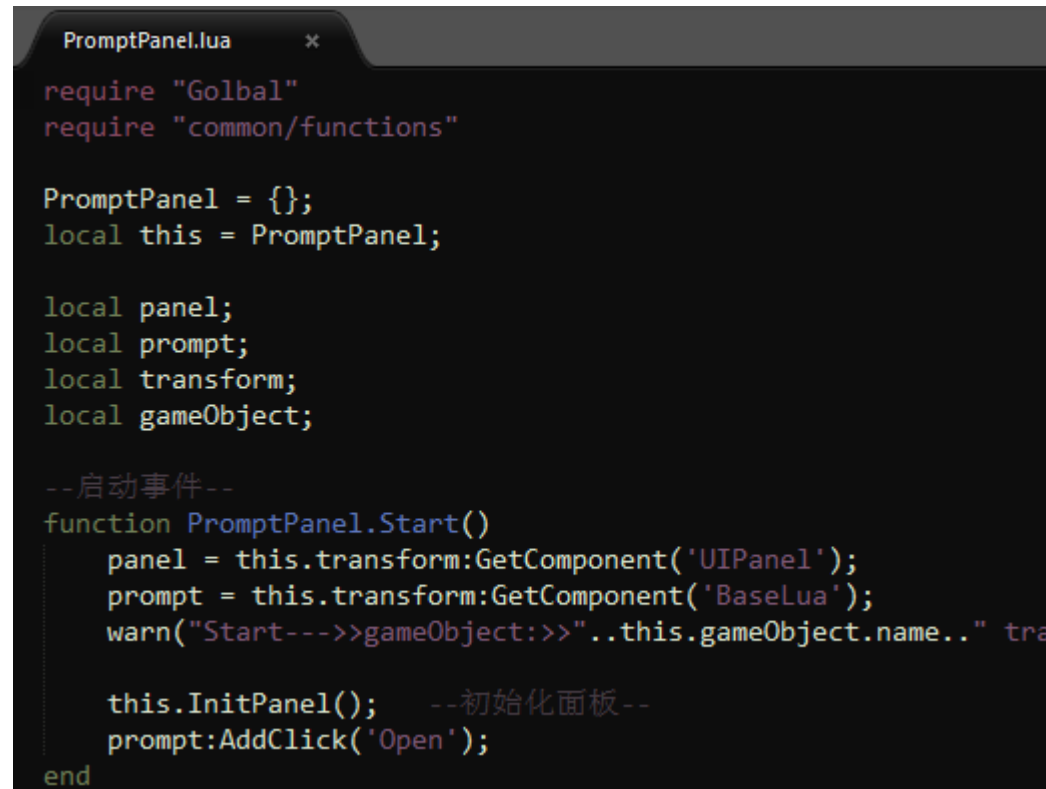
/// <summary>
/// 执行Lua方法-无参数
/// </summary>
protected object[] CallMethod(string func) {
    if (luaMgr == null) return null;
    string funcName = name + "." + func;
    funcName = funcName.Replace("(Clone)", "");
    return mgr.CallLuaFunction(funcName);
}

```

很简单就是拼一个“模块名.函数名”字符串格式，然后通过 mgr 管理器调用 lua 函数而已。单 luastate 是靠模块名来区分哪个 lua 模块里面的函数调用。

这时候，面板应该已经创建了，但是事情还没完，面板既然已经创建了，那面板上的东西 Item 就要开始创建了。我们就要找到“提示”面板的 Start 函数喽~

打开 PromptPanel.lua 文件，找到 Start 函数如下：



```
PromptPanel.lua
require "Golbal"
require "common/functions"

PromptPanel = {};
local this = PromptPanel;

local panel;
local prompt;
local transform;
local gameObject;

--启动事件--
function PromptPanel.Start()
    panel = this.transform:GetComponent('UIPanel');
    prompt = this.transform:GetComponent('BaseLua');
    warn("Start--->>gameObject:>>"..this.gameObject.name.." tra

    this.InitPanel();    --初始化面板--
    prompt.AddClick('Open');
end
```

这都是一些常规的函数调用了，没啥好说的，主要说下，我们在 InitPanel 函数之前，有新建面板上的按钮委托。找到这个叫 Open 的按钮 GameObject，然后添加它的 onClick 时间给本 lua 里面的 onClick 做好关联，这样当按钮被单击，就会执行到本 lua 的 onClick 函数了。

紧接着是 InitPanel 函数，这个函数主要功能是创建所有的 Item，


```

--初始化面板--
function PromptPanel.InitPanel()
    panel.depth = 1;    --设置纵深--
    local parent = this.transform:Find('ScrollView/Grid');
    local itemPrefab = prompt:GetGameObject('PromptItem');
    for i = 1, 100 do
        local go = newobject(itemPrefab);
        go.name = tostring(i);
        go.transform.parent = parent;
        go.transform.localScale = Vector3.one;
        go.transform.localPosition = Vector3.zero;

        local goo = go.transform:FindChild('Label');
        goo:GetComponent('UILabel').text = i;
    end
    local grid = parent:GetComponent('UIGrid');
    grid:Reposition();
    grid.repositionNow = true;
    parent:GetComponent('UIWrapGrid'):InitGrid();
end

```

这也是常规函数调用了，找到 Grid 对象，然后往下面 newobject 了，不过 Item 的 Prefab 哪里来的？有个函数叫 prompt.GetGameObject，这个函数实际上调用的 BaseLua 里面的函数，此时此刻，这个面板的 bundle 资源引用已经传递进 BaseLua 类了，通过它内部的这个函数就可以轻易的获取 bundle 除了面板之外的 prefab 对象进来，看下它的实现：

```

/// 获取一个GameObject资源
public GameObject GetGameObject(string name) {
    if (bundle == null) return null;
    return bundle.Load(name, typeof(GameObject)) as GameObject;
}

```

接下来一个注意的是 UIWrapGrid 的类的使用，这个是我为了创建大容量滚动不卡的滚动列表的一个类，NGUI 原生的滚动列表超过 100 个就卡了，体验很差，它的使用很简单，把这个类拖到跟 UIGrid 的对象上就好了，其它完全不用设置，只是在生成初始化完所有的面板 Item 之后，调用下 InitGrid，然后让它起作用。

对于界面加载部分的内容到此结束了。

接下来我们回到 game.lua 的 OnInitOK 函数里面，看下后半部分，连接服务器。并且加入了一些功能库的实例代码。

```
--初始化完成，发送链接服务器信息--
function GameManager.OnInitOK()
    warn('OnInitOK--->>>');
    createPanel("Prompt");

    Const.SocketPort = 2012;
    Const.SocketAddress = "127.0.0.1";
    io.networkManager:SendConnect();

    this.test_class_func();
    this.test_pblua_func();
    this.test_cjson_func();
    this.test_pbc_func();
    this.test_lpeg_func();
    this.test_sqlite_file();
    this.test_sqlite_memory();
end
```

这个功能是 v0.1.5 才加入的，顺便还加入了一个辅助测试网络功能的服务器端框架。

后半部分就是通过，io 找到网络管理器，然后发起连接请求。

```
/// <summary>
/// 发送链接请求
/// </summary>
public void SendConnect() {
    SocketClient.SendConnect();
}
```

具体 SocketClient 里面的实现怎么回事，自己找 c# socket 资料去吧，我就不讲了。注意里面的沾包处理，值得看看。

这时候客户端就会像 127.0.0.1 的 2012 端口发起连接请求，当连接上之后，网络管理器会收到消息。然后把这个消息通过父类 BaseLua 传递给 network.lua 文件。

```
public static void AddEvent(int _event, ByteBuffer data) {
    sEvents.Enqueue(new KeyValuePair<int, ByteBuffer>(_event, data));
}

void Update(){
    if (sEvents.Count > 0) {
        while (sEvents.Count > 0) {
            KeyValuePair<int, ByteBuffer> _event = sEvents.Dequeue();
            switch (_event.Key) {
                default: CallMethod("OnSocket", _event.Key, _event.Value); break;
            }
        }
    }
}
```

PS:网络消息的上抛都是固定的格式:

```
/// <summary>
/// 执行Lua方法-Socket消息
/// </summary>
protected object[] CallMethod(string func, int key, ByteBuffer buffer) {
    if (luaMgr == null) return null;
    string funcName = "network." + func;
    funcName = funcName.Replace("(Clone)", "");
    return mgr.CallLuaFunction(funcName, key, buffer);
}
```

因为网络消息只有 network.lua 才需要接收, 所以这里写死了。这样我们就转到了 network.lua 里面:

```
require "common/protocol"

Network = {};
local this = Network;

local transform;
local gameObject;

local islogging = false;

function Network.Start()
    warn("Network.Start!!");
end

--Socket消息--
function Network.OnSocket(key, buffer)
    if key == Connect then this.OnConnect(); end
    if key == Exception then this.OnException(); end
    if key == Disconnect then this.OnDisconnect(); end
    -----
    if key == Login then this.OnLogin(buffer); end
    --ModuleName.function--
end
```

这里面通过 key 与 protocol.lua 里面的协议格式定义, 来判断当前是哪个消息请求。这样在 lua 中就可以动态添加、删除、修改通信协议了。

```
--Buildin Table
Connect      = 101;    --连接服务器
Exception    = 102;    --异常掉线
Disconnect   = 103;    --正常断线
Login        = 104;    --Login
```

这样当服务器连接成功的消息, 回传之后, lua 也就可以通过上述流程接收到服务器的数据, 并且打印出来了。

PS:其实其它的发送, 接收服务器的推送消息都是通过上述流程拿到的。以后也

就不再重复这个流程了。

到此时，游戏界面已经显示完成，没有任何自启动的逻辑了，我们看到如下：



面板上有个按钮，就是前面做好单击事件关联的 Open 按钮，还记得吧，那我们就可以单击这个按钮，执行发送给服务器的逻辑操作了。

我们打开 PromptPanel.lua 文件, 找到 OnClick 函数, 看看它内部的实现方式:

```
--单击事件--
function PromptPanel.OnClick()
    local buffer = ByteBuffer.New();
    buffer:WriteShort(Login);
    buffer:WriteString("ffff我的ffffQ蛋uuu");
    buffer:WriteInt(200);
    ioo.networkManager.SendMessage(buffer);
    warn("OnClick---->>>"..gameObject.name);
end
```

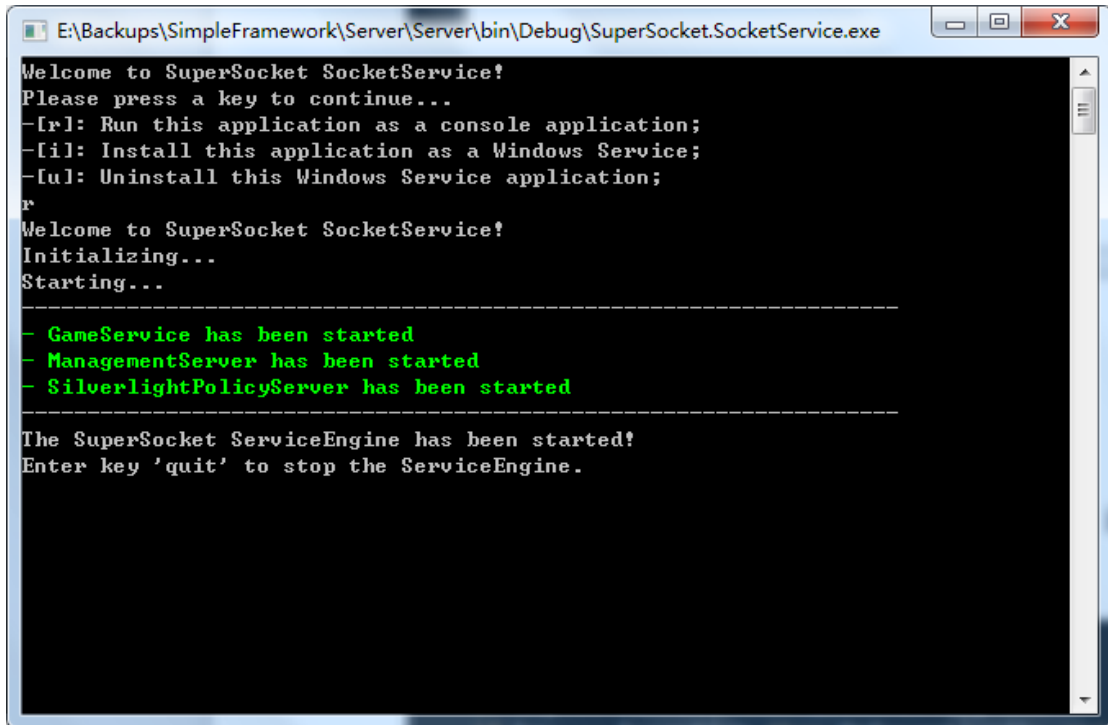
这个函数就是 new 了一个 ByteBuffer 类，这个类也是通过 tolua 倒进来的，可以填充数据流，一个 short（协议 ID，操作请求标识），一个字符串，一个 int，然后通过网络管理器把这个 bytearray 类给服务器端发送出去

/// 发送SOCKET消息

```
public void SendMessage(ByteBuffer buffer) {
    SocketClient.SendMessage(buffer);
}
```

```
}
```

当服务器处于正常运行状态的时候，



```
E:\Backups\SimpleFramework\Server\Server\bin\Debug\SuperSocket.SocketService.exe
Welcome to SuperSocket SocketService!
Please press a key to continue...
-[r]: Run this application as a console application;
-[i]: Install this application as a Windows Service;
-[u]: Uninstall this Windows Service application;
r
Welcome to SuperSocket SocketService!
Initializing...
Starting...

-----
- GameService has been started
- ManagementServer has been started
- SilverlightPolicyServer has been started
-----

The SuperSocket ServiceEngine has been started!
Enter key 'quit' to stop the ServiceEngine.
```

就会接收到一个回传消息，也是通过上述接收流程，通过网络管理器
->network.lua

```
--Socket消息--
function Network.OnSocket(key, buffer)
    if key == Connect then this.OnConnect(); end
    if key == Exception then this.OnException(); end
    if key == Disconnect then this.OnDisconnect(); end
    -----
    if key == Login then this.OnLogin(buffer); end
    --ModuleName.function--
end
```

```
--当登录时--
function Network.OnLogin(buffer)
    local result = buffer.ReadByte();
    if result == 0 then return; end
    islogging = true;
    local str = buffer.ReadString();
    warn('OnLogin---->>>'..str);
    createPanel("Message"); --Lua里创建面板
end
```

当正常接收消息，并解析数据包正常结束后，开始通过 createPanel 创建 Message

面板，如下：



注解文档到此结束了。感谢使用。

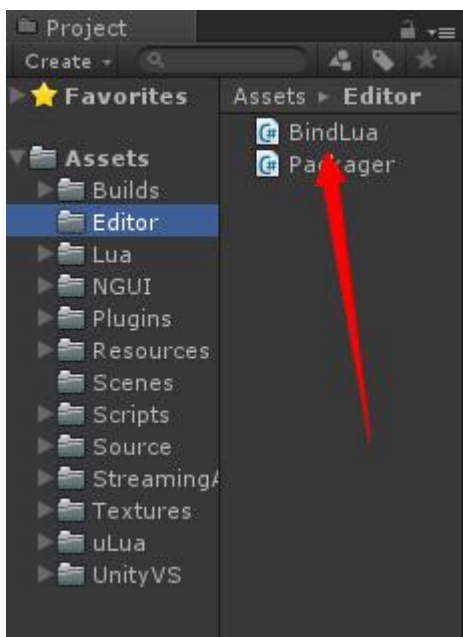
《cstolua 的使用》

作者: toameng

整理: 骏擎 CP

首先我们要感谢作者阿萌（我给起的昵称）给我们创造了一个非常优秀的 ulua 插件，使 ulua 可以提升数倍的性能。我跟阿萌也学习了很多东西，这里我就不班门弄斧了，主要是整理下简单的使用方法，供大家参考。

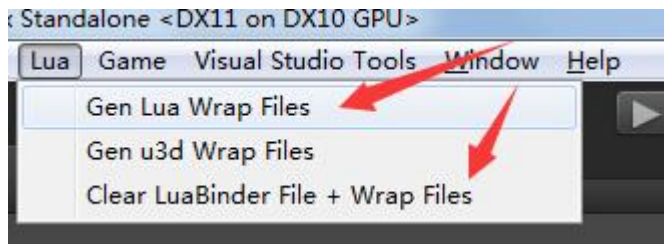
打开 Editor/BindLua.cs 文件，



在 static BindType[] binds = new BindType[]变量末尾一行加入要导入的 cs 类，

```
_GT(typeof(TimerManager)),  
_GT(typeof(LuaHelper)),  
_GT(typeof(BaseLua)),  
_GT(typeof(UIPanel)),  
_GT(typeof(UILabel)),  
_GT(typeof(UGrid)),  
_GT(typeof(UIWrapGrid)),  
_GT(typeof(LuaEnumType)),
```

保存等 u3d 编译完后，选择 “Lua/Gen LuaBinding Files” 编译即可，



以后就可以在 lua 代码里面使用此类了，而不要通过反射 load_type 导入进来。

PS: 如果有时候之前有个类修改过了过了，但是 wrap 类里面需要生成 wrap 文件，如果不

好用的话，请选择 “Lua/Clear LuaBinder File + Wrap Files” 清除 Wrap 文件列表。

等待编译完，重新单击菜单 “Lua/Gen LuaBinding Files” 生成 wrap 文件即可。