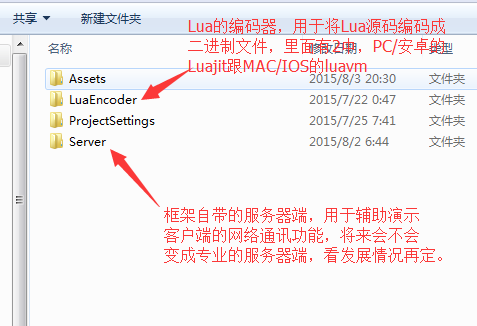
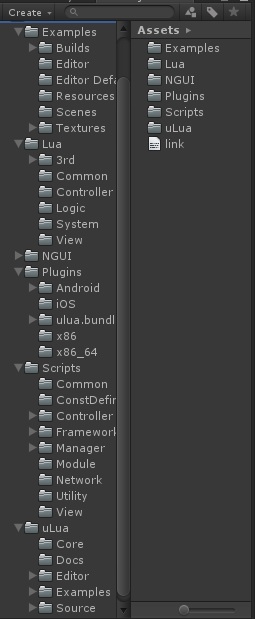
# **SimpleFramwork基础1**

先看看Unity工程外层两个重要的目录LuaEncoder跟Server是干嘛的：

[](http://bbs.ulua.org/bbs/UploadFile/2015-8/2015891130189840.png)

[](http://bbs.ulua.org/bbs/UploadFile/2015-8/2015891136964914.jpg)  
  
上面的图是目前SimpleFramework的基本目录结构，而且这个结构短时间内不会发生太多的变化。

我们可以简单介绍下每个目录的用途。

**Examples ：**框架自带的Demo例子，如果只需要框架的同学，里面的资源可以删除掉。去“疑难解答”里面查看方法。

**---Builds：**里面都是一些NGUI定义的图集啊、Prefab等资源。用于生成assetbundle而准备的资源。

**---Editor：**里面是例子用到的一个新手引导步骤演示的编辑器脚本。

**---Editor Default Resource：**目录是新手引导步骤对话框用的的图片资源。

**---Rsources：**例子里面用于演示的一个内建的GUI容器的Prefab。

**---Scenes：**里面一个login场景文件，另一个是cstolua自带的性能测试场景文件。

**---Textures：**里面是Buidls目录里面图集的原图文件。

**Lua：**框架自带的Lua源码目录，用户自定义的Lua脚本也就是放在这里面，最后打包的时候，打包脚本会将其按目录结构生成到StreaminAssets目录里面去，然后在将其上传到游戏的Web服务器上面，用于准备被每个游戏客户端下载更新他们本地的Lua脚本。达到热更目的。

**---3rd：**里面是第三方的一些插件lua、实例源码文件，比如：cjson、pbc、pblua、sproto等。

**---Common：**公用的lua文件目录，如define.lua文件，一些变量声明，全局配置等，functions.lua常用函数库，通讯的protocal.lua协议文件。

**---Controller：**控制器目录，它不依赖于某一个Lua面板，它是独立存活在Luavm中的一个操作类，操作数据、控制面板显示而已。

**---Logic：**目录里面存放的是一些管理器类，比如GameManager游戏管理器、NetworkManager网络管理器，如果你有新的管理器可以放到里面。

**---System：**这个目录是cstolua的系统目录，里面存放都是一些常用的lua类，为了优化lua调用速度，用lua重写的unity常用类。

**---View：**这是面板的视图层，里面都是一些被Unity调用的面板的变量，走的是Unity GameObject的生命周期的事件调用。

**Plugins：**ulua底层库所在的目录，里面存放的是不同平台的底层库，之所以ulua效率高，就是它是纯c的lua虚拟机，而不是c#解释型的。

**---Andriod：**安卓lua虚拟机底层库，里面分为armv7-a与Intel x86平台。

**---iOS：**里面就是苹果lua虚拟机底层库。

**---ulua.bundle：**里面是Mac机器的底层库。

**---x86：**里面是Win32/Linux32位机器的lua虚拟机底层库。

**---x86\_64：**里面是Win64/Linux64位机器的lua虚拟机底层库。

**Scripts：**框架的C#脚本层，之所以这个目录跟lua目录都放在最外层，为了让用户一眼都能找到，明白是什么。

**---Common：**框架的公用定义类。BehaviourBase(行为基类)、GlobalGenerator全局构造器、与luavm通知unity游戏对象的“LuaBehaviour”桥类。

**---ConstDefine：**常量定义目录，AppConst（应用常量）ManagerName（管理器名称）NotiConst（通知常量，用于mvc消息通知）。

**---Controller：**控制器目录，分为Boostraps启动控制器，跟Command常用逻辑控制器。

**---Framework：**经过修改过的PureMVC的框架文件。

**---Manager：**Unity提供基础功能的管理器类，音乐、面板、线程、资源等众多管理器。

**---Module：**PureMVC的模块层，目前里面将Socket作为一个模块代理放到里面了。

**---Network：**网络的常用辅助类，ByteBuffer字节操作封装类，网络协议类，转换器类。

**---Utility：**常用工具类。

**---View：**C#用的PureMVC的视图层。

**uLua：**ulua/cstolua的核心目录，里面还有经过我们修缮后ulua的基础使用例子，用户初学者最佳。

**---Core：**顾名思义，ulua的核心目录，所有c#与lua的交互都是通过它进行调度的。

**---Docs：**ulua自带的文档目录，因为ulua基于LuaInterface，所以这里面就是它的文档。

**---Editor：**这是供cstolua去反射定义Wrap文件列表的工具类目录。

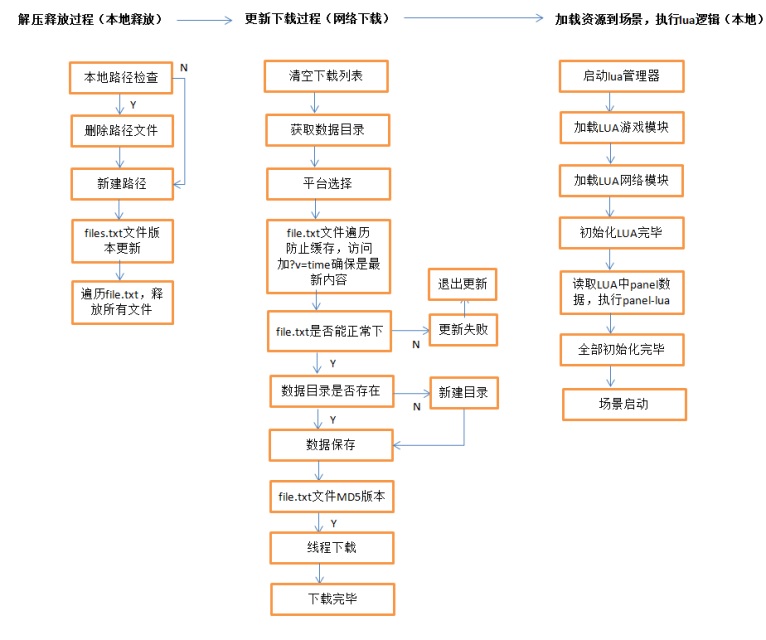
**---Examples：**经过我们修改增加后ulua自带的例子。

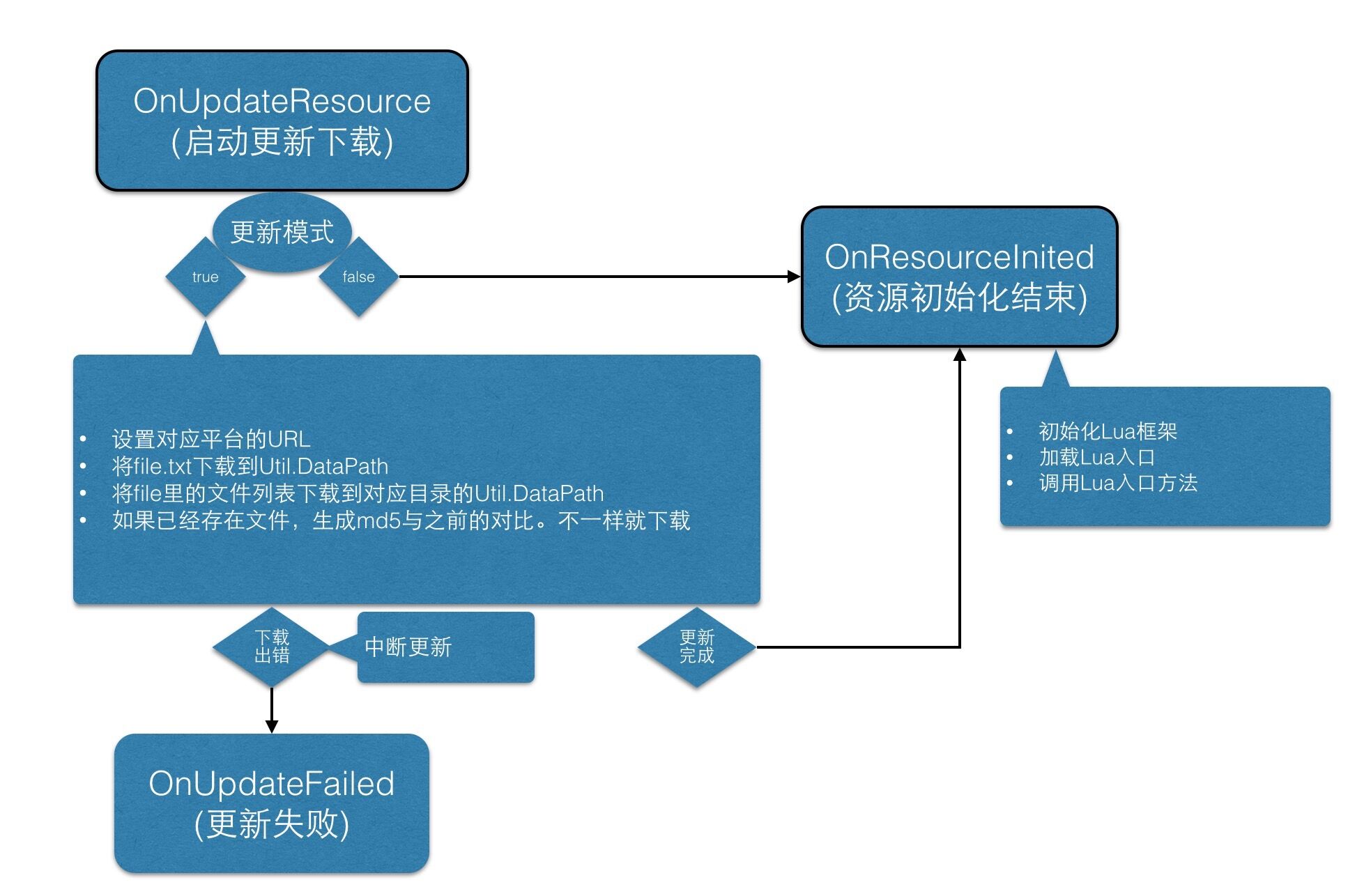
**---Source：**这个是cstolua的核心目录，里面有Base核心目录，与动态生成用于存放LuaWrap类的缓存目录。

上面的目录结构，就介绍到此为止了。既然是热更新游戏框架，接下来我们说下怎么做热更？

其实做热更的步骤不麻烦，因为Unity的c#类部分热更不了，那我们需要热更的就只有Lua代码部分了，当我们写完游戏测试OK了，就可以上线运营，并且在Web服务器上面部署最新的版本资源文件，我们就可以将Lua的代码编码后上传到Web服务器上面去，当游戏客户端启动的时候，它会启动解包流程、解包的资源一般都是当时做包时候的资源，到后面我吗修改以后，它内部的资源可能已经不是最新的了，但是大部分可能是新的，只有少部分需要更新，那紧接着就启动更新流程，从Web服务器上面的资源配置列表里面，通过MD5/CRC比较查询到最新的资源，下载更新本地的文件，达到更新最新版的目的，游戏顺利启动。

上面基本上就是热更的指导思想了，SimpleFramework框架也基本上以此思路在运行的，如果还不清楚的话，可以看下图：





# **SimpleFramwork基础2**

看完上面的图，不知道是否已经对热更有了新的概念，如果有的话，我们继续说下，SimpleFramework怎么来做热更的具体步骤。

（1）第一步打包资源到StreaminAssets目录，为啥要打包到这个目录下呢？懂U3D游戏开发的都知道，这个目录会随着Unity最终生成APK/IPA的包原目录打包出去，我们的游戏客户端框架可以通过代码读取到里面的资源，并且把里面的资源复制到玩家的手机本地存储里面，这叫做解包。

那怎么打包呢？打包的资源分为素材资源与代码资源，这两部分的打包，框架都集成了，你可以直接修改里面的脚本逻辑适应自己的游戏项目，我们打开ulua/Editor/Packager.cs打包脚本。这里面根据不同的平台打包相应的资源。

    [MenuItem("Game/Build iPhone Resource", false, 11)]

    public static void BuildiPhoneResource() {

        BuildTarget target;

#if UNITY\_5

        target = BuildTarget.iOS;

#else

        target = BuildTarget.iPhone;

#endif

        BuildAssetResource(target, false);

    }

    [MenuItem("Game/Build Android Resource", false, 12)]

    public static void BuildAndroidResource() {

        BuildAssetResource(BuildTarget.Android, true);

    }

    [MenuItem("Game/Build Windows Resource", false, 13)]

    public static void BuildWindowsResource() {

        BuildAssetResource(BuildTarget.StandaloneWindows, true);

    }

重要的函数是下面这个，它打包了框架自带例子的素材文件后，继续处理Lua代码文件的打包。

    public static void BuildAssetResource(BuildTarget target, bool isWin) {

        if (AppConst.ExampleMode) {

            HandleExampleBundle(target);

        }

        HandleLuaFile(isWin);

        AssetDatabase.Refresh();

    }

HandleExampleBundle函数就是将Examples/Builds下面的素材统一按照Unity的规则进行打包成assetbundle文件。没什么好说的。

我们主要是看HandleLuaFile函数的操作流程。

（1）在StreamingAssets目录下面新建lua目录，用于存放编码后的lua文件。

（2）遍历Lua目录下面所有的lua文件，并且根据目录结构创建相应目录树。

（3）如果AppConst.LuaEncode 设置了编码开关，就启动编码操作，否则直接复制源码过去。

（4）然后将前面的图片素材assetbundle跟相对应的lua文件目录，统一遍历计算出MD5/CRC，生成到files.txt里面。

这个代码就做了这几件事情，唯一需要展开介绍的只有怎么编码Lua文件了吧。这个函数其实并不长，

他就是根据相应的平台配置其编码器，编码器存放的路径也就是前面工程外的LuaEncoder的目录，拼凑出它的路径后，我们开始拼凑其编码的参数命令行，

需要注意的是luajit跟luavm的编码命令行不同：

**luajit的：luajit.exe -b srcfile outfile**

**luavm的：luac -o outfile srcfile**

命令行的格式其实都很简洁，然后我们启动C#的调用外部shell程序的代码，调用他们来编码出2进制文件即可。那最后统一计算MD5的文件就变成了编码的2进制的值。

 //------------------------------------------------------------------------------------------------------------

说完了打包流程，接下来说下怎么更新，只有能顺利更新下面后，才能够算作热更的最后一环。关于更新这一块，我们是用C#写成的，为啥？因为我们真实项目中，游戏的基础功能，比如下载、线程操作都建议在C#中完成，为了效率，而不是为了花架子都在lua中完成。因为这些基础功能，相对于游戏来说，更新的频率非常之低，用C#完成追求了效率，也没有任何损失。

我们打开Scripts/Manager/GameManager.cs文件，定位到IEnumerator OnUpdateResource()函数，

（1）它一上来就做些初始化的操作，找到更新的地址URL等。

（2）请求更新列表文件files.txt，因为里面存放了上面生成的目录结构及其MD5/CRC的信息。

（3）分析Web服务器上的files.txt文件内容，然后遍历检查本地的文件结构是否完整、MD5是否匹配。

（4）如果MD5不匹配，或者本地文件不存在，就开始创建一个下载请求，并且将它传递给线程，请求线程下载。

（5）本地协程会每一帧查询线程下载完后是否将下载的文件名存放到下载文件列表中，如果找到，继续下载下一个，直至全部下载完成。

其实当全部更新完成后，此时手机存储的文件结构及其内容已经是最新的了，客户端程序便可以顺利启动，完成了热更的最后一环。

2015/8/10 23:20:36

--------------------------------------------------------------------------------------------------------------

今天继续更新啊，说些啥呢，说下现在的Lua端的代码结构与怎么创建一个新的对话框出来，因为问的人较多，稍微介绍下这块。

经过改造过的代码架构有了变化，原来的lua除了管理器类，比如GameManager、NetworkManager等，每个面板文件直接就是一个类似xxxPanel.lua的格式，有人喜欢这个形式，有的人讨厌，觉得不够专业化，强调用过多。口味难调。

后来将xxxPanel.lua分开了，一个是纯粹的View层，名称还是xxxPanel.lua形式，它里面维护了面板的GameObject上面所有的控件变量，以便提供给xxxCtrl.lua使用，你可以用工具代码生成这一层。相比之下，大量的操作逻辑移到了xxxCtrl.lua里面，这样View层跟Controller层就此分离开了。那Module层因为没有数据，所以还是使用者自己添加吧，我就不画蛇添足，免得被人骂。

我们继续说Controller层，既然有了这一层，那就得有个管理器去管理Controller层，所以就出现了CtrlManager.lua，它里面维护了所有的控制器的关系，我们可以通过它获取需要的Controller，进而Open、Close操控View层。这样解释完，基本上都懂得为啥要这样操作了吧~~

接下来我们说下，怎么创建一个新的面板，样例代码请参考最新的版本，当下是0.3.7 f4版本。

（1）创建一个Unity3d的面板对象，不管你使用UGUI还是NGUI，请做好布局准备。

（2）如果是NGUI，请将这个面板的图集、Prefab等文件，加入到Editor/Package.cs的打包函数里面。

（3）如果是UGUI，请给这个面板的Prefab一个AssetBundle名称(我们用的U5新版打包方式)，

（4）面板的名称请以"xxxPanel"命名，然后单击Game菜单Build XXX Resource打包素材。

（5）在lua中的Controller目录中添加相应控制器，比如MessageCtrl.lua，里面的格式按照框架示例添加。

（6）在lua中的View目录中添加相应的视图层，比如MessagePanel.lua，里面添加好面板的Button、Image的变量关联。

（7）在Logic/GameManager.lua里面的LuaScriptPanel函数里面添加面板的名称，作用是lua告诉c#哪些面板的lua文件需要提前加载到luavm。

（8）在CtrlManager.lua里面按照相应的格式，添加初始化控制器代码。

（9）非常非常重要的一步！！（很多人都会忽略），记得在Common/define.lua的初始化面板table中添加"MessagePanel"，这样当C#的GameManager初始化的时候，才会将这个MessagePanel.lua注册进lua虚拟机里面。再当显示面板的时候，才不会出现类似"Push xxxPanel failed"的操作提示。

启动吧~

--------------------------------------------------------------------------------------------------------------

 今天帮大家梳理下新版框架的头，可能会说不完，慢慢更新吧。

看过老版框架同学应该知道，我的框架有个全局构造器“GlobalGenerator”对象，是框架的第一入口，之前的游戏管理器“GameManager”起源于它，InitGameMangager，也就是在它身上脚本用Prefab实例化的，但是新版里面没有了GameManager的Prefab，变成了直接new GameObject（），为了精简。

GameManager的对象创建了，需要往它身上添加管理器，这个跟原来的方式相同，没啥变化。但是变得有些绕，我也不能说PureMVC不好，但是的确不是我喜欢的，没办法，用的人多就是真理，顺了它了。

有一行代码AppFacade.Instance.StartUp();   //启动游戏，它其实就是启动PMVC，不懂的，自己查文档去，别问我。这行会启动PMVC框架，进到函数里面，可以看到，它里面就2行代码，一个是发送Command，一个是移除Command，一次性的用完就删，全民好习惯，不用夸我。

那这个命令是在什么地方注册的？上面

 override protected void InitializeController()

 {

        base.InitializeController();

        RegisterCommand(NotiConst.START\_UP, typeof(StartUpCommand));

 }

所以有了注册，下面有了发送，那我们开始找接收的地方，在Controller/Command/StartUpCommand.cs里面了，

里面除了检查框架运行环境的Util.CheckEnvironment()之外，后面还有3行启动命令，我日，好麻烦，什么玩意儿，我当时学到此，也是这种感觉。

        //BootstrapModels

        AddSubCommand(typeof(BootstrapModels));

        //BootstrapCommands

        AddSubCommand(typeof(BootstrapCommands));

        //BootstrapViewMediators

        AddSubCommand(typeof(BootstrapViewMediators));

其实就是启动MVC各自层的子命令，PMVC作者你过来过来，我保证不打死你。

别的先不看，先打开看下中间的BootstrapCommands类吧，进去之后，我们看到：

里面除了添加了一条新消息Facade.RegisterCommand(NotiConst.DISPATCH\_MESSAGE, typeof(SocketCommand));

为了Socket消息做准备外，下面就是一段怎么添加管理器，我日，facade啥东西？

在PMVC里面，它的含义叫“门面”，他们不建议用单例，建议facade是以前外部访问框架内部的总入口，为了解耦。

那我就按照它的规则用了：Facade.AddManager(ManagerName.Lua, new LuaScriptMgr());

挨个的添加每个管理器，老版用户对这一幕很熟悉，原来是用Util.Add<GameManager>(gameObject)添加的，那这个Facade.AddManager也是类似功能，只是将原来的Util.Add转移到了Facade中，那我们可以继续跟进去看下AddManager怎么操作的？

        //SimpleFramework Code By Jarjin lee

        void AddManager(string typeName, object obj);

        T AddManager<T>(string typeName) where T : Component;

        T GetManager<T>(string typeName) where T : class;

        void RemoveManager(string typeName);

      这是我新建的4个接口定义，原PMVC没有这些，单击AddManager的引用实现函数，大概在Facede.cs的523行位置，就是我4个函数体的实现部分，

        GameObject AppGameManager {

            get {

                if (m\_GameManager == null) {

                    m\_GameManager = GameObject.Find("GameManager");

                }

                return m\_GameManager;

            }

        }

原来如此，我找到新New出来的GameManager，然后根据传进来的不同的类型T，添加到了GameManager对象身上，完成了原来很简单的Util.Add操作，

多说两句，为啥我脑残非要放到这里？原因是facade的设计是全框架通用，任意位置都可以直接访问，这个构想跟我原来的ioo类是一样的，为了搭上它的顺风车，让框架任意地方都能访问到所有的管理器，因此，只能在这里找个安身之所。被这么绕了半天，我也挺需要安慰的，毕竟不是我设计的PMVC。

如果能看到PMVC作者，我非要第一个上去，给他一脚，但是谁让用它的多呢？

# **SimpleFramwork基础3**

昨天，痛骂完PMVC多么的操蛋，到了晚上，就遇到一个项目里面的问题，突然觉得接口无关性与PMVC的消息“无主”性比起来吧，还是PMVC更加科学、合理一些。至少简洁明了，不用拖泥带水的。

既然，首次觉得PMVC有好处，那么我们继续讲，关于昨天最后提供给大家一个“门面”facade，以及搭载它的顺风车的管理器们，

有一个问题，如果在普通的Unity对象上面，GameObject上面如何获得这些支持咧？其实 很简单，我们往常给一个类集成的时候都用Unity自带的MonoBehaviour，那现在用框架提供的，继承自BehaviourBase类即可，它本来也集成自MonoBehaviour，里面包装了所有管理器的获取、维护等操作。所有的子类都可以直接调用不同管理器，或者收发PMVC的消息。

其实这一次的改版，除了PMVC的集成引入外，还有个非常重要的变动，就是引入了另一个管理器“线程管理器”，引入它的目的在于真正的开发过程中，难免都需要做很多分包操作，你可以参考里面的实现，将一些费时操作交给线程去完成，完成后再通过PMVC通知主线程，完成一次echo，主线程就避免卡顿的用户体验。今天我们主要说下线程管理器跟更新系统的协同操作。

说线程类之前，首先说下，怎么传递消息给线程？当GameManager类中OnUpdateResource的时候，里面有个BeginDownload函数，是它启动了ThreadManager.AddEvent(ev, OnThreadCompleted);   //线程下载。

线程管理器的代码并不多，短短150行而已，里面演示较多的无非是下载相关的代码，当主线程通过下面函数：

        /// <summary>

        /// 添加到事件队列

        /// </summary>

        public void AddEvent(ThreadEvent ev, Action<NotiData> func) {

            lock (m\_lockObj) {

                this.func = func;

                events.Enqueue(ev);

            }

        }

将事件传递给线程，记得加锁，我用消息队列保存了所有的事件请求，然后在每一此OnUpdate中，解析每一个消息，根据请求的类型做“解包”、“下载”操作，将来你可以增添自己的操作类型。我们这里主要就是下载文件OnDownloadFile，解析完里面的下载URL，就开始启动异步下载DownloadFileAsync。因为是异步的，所以每一次的下载更新操作都在ProgressChanged完成，比如更新下载进度等。

那在更新函数里面，我拿到一个更新的下载进度，如何传递给主线程，并且让它做出刷新界面操作呢？这里用到了delegate，封装了一个通知数据类实例：NotiData data = new NotiData(NotiConst.Update\_PROGRESS, value); 里面是关于更新进度的操作，以及一些数据通过m\_SyncEvent回调给主线程，其实当进入m\_SyncEvent的时候，CPU已经身处主线程了，那就好说了嘛，

既然线程管理器也集成自BehaviourBase，那我直接获取父类的facade的成员变量，并且发送消息出去，关于谁接收？这里不关心。但是我们还是要找到接收的地方，怎么找？看到前面的NotiConst.Update\_PROGRESS了嘛，查找它的引用地方在哪儿？发现监听此消息的是AppMediator，这是PMVC的V层了，主要处理V层的逻辑的，那通过它：

case NotiConst.Update\_PROGRESS:     //更新下载进度

    view.UpdateProgress(body.ToString());

break;

通知View更新界面，也就完成了最后的操作，到达最后终点了。

当下载的这个文件完成以后，再次调用m\_SyncEvent，这次传递的消息不是更新进度，而是更新完成消息：NotiConst.Update\_DOWNLOAD。再通过事先AddEvent传进来的、保存到“完成事件回调函数”：func，回调给GameManager，它接收到知道线程完成当前操作，在回调函数OnThreadCompleted里面，将下载完成的文件名添加到下载队列中，然后更新协同开始接着请求下载下一个文件，直到下载全部完成为止。

当所有的操作都完成后，这个线程不会主动退出，而是一直等着主线程“交代”任务进来，这就是线程与下载的完整流程。

借楼层更新-----------------------------------------------------------------------------------------

今天周五，睡觉前再更新一篇帖子，这回咱们聊下在PMVC下的Socket跟NetworkManager如何交互细节。还记得前面的帖子有个地方记录了：在BootstrapCommands类注册了一个SocketCommand的地方吗？不记得自己翻去。里面添加了一条新消息Facade.RegisterCommand(NotiConst.DISPATCH\_MESSAGE, typeof(SocketCommand));为了Socket消息做准备，下面呢，就是注册了一大堆的管理器。

直接打开SocketCommand.cs文件，看看有啥？哦~~这不是原来NetworkManager里面Update函数里面的事件检测函数么？是的，就是从它复制来的，也就是说，当从NetworkManager接收到消息后，它一定会把消息通过facade发送出去，而且没有指定谁来接收，然后这里会接收，并且处理消息派发给Lua的网络管理器：Network组件进行再次分发。为啥？还用问，它不关心接受者，自然也就解耦了，解除了原来的强引用。

为了确定猜想，我们打开NetworkManager.cs文件，果不其然的代码如下：

/// <summary>

/// 交给Command，这里不想关心发给谁。

/// </summary>

void Update() {

    if (sEvents.Count > 0) {

        while (sEvents.Count > 0) {

             KeyValuePair<int, ByteBuffer> \_event = sEvents.Dequeue();

             facade.SendNotification(NotiConst.DISPATCH\_MESSAGE, \_event);

         }

     }

  }

其他的代码都是原来的，没什么可讲的，尽管如此，但是里面有个变量却不得不重点看下，就是下面的

  SocketProxy SocketClient {

       get {

            if (socket == null) {

                socket = facade.RetrieveProxy(SocketProxy.NAME) as SocketProxy;

            }

            return socket;

       }

  }

这个是新的框架才具备，这是虾米东东？其实就是原来框架的SocketClient.cs类，这里是我忘了在哪里看到的，把它当作的Moudule层的代理类。距离网络底层最近的地方，由他来接收网络数据，并且通过添加静态队列AddEvent()将消息传递给NetworkManager，它在通过PMVC的解耦消息发送出去，被SocketCommand类监听截获到之后，再次分发给C#还是LUA的Network组件，以便再次分发给Lua的模块。这就是这一块的思路流程。好不好，自做评判吧。

# **SimpleFramwork基础4**

周六早晨，咱接着更新，这次咱们说下SimpleFramework使用的4种网络协议层：bytebuffer、protobuf\_lua\_gen、pbc、sproto。

（1）bytebuffer：这个只是框架里面的一个c#类，起源很早了，大概是在我2012年创业的时候就有它了，它就是一个二进制socket协议字节流操作的类，那干吗用它来命名？因为大家都熟悉它，我也用惯了。ULUA支持它应该是协议层最早的，所以上面的顺序都是按照时间来排的。我觉得这个方式最没有什么什么可说的，因为它就是用Wrap的方式将这个类注册进Lua中，然后通过将所有类型的值压入一个队列，最终ToArray变成一个byte[]变量，然后通过c#的socket发出去。唯一值得说的是，目前新版框架的其他协议都依靠它作为基础，后面细说。这里补个PromptPanel.lua文件中的例子看下：

function PromptCtrl.TestSendBinary()

    local buffer = ByteBuffer.New();

    buffer:WriteShort(Login);

    buffer:WriteByte(ProtocalType.BINARY);

    buffer:WriteString("ffff我的ffffQ靈uuu");

    buffer:WriteInt(200);

    NetManager:SendMessage(buffer);

end

这里面最后发送消息是通过C#网络管理器，打开Scripts/Manager/NetworkManager.cs

        public void SendMessage(ByteBuffer buffer) {

            SocketClient.SendMessage(buffer);

        }

那SocketClient也就是将Socket封装进Module代理层的SocketClient，接着跟进去：

SendMessage->SessionSend->WriteMessage(这里数据已经从ByteBuffer转换成了byte[])->BeginWrite. 终于跟到头了,消息也就发送出去了。

我的函数命名规范应该很清晰了，都是一眼都能看出来的，没有什么乱七八糟的缩写。就是New开辟了一个数据块，然后往里面塞东西，那相应的服务器端，也得这样顺序读取才是。

这是发送，接收咧？对了，想想昨晚上一篇帖子，写的是啥，看下在SocketCommand.cs里面的代码：

KeyValuePair<int, ByteBuffer> message = (KeyValuePair<int, ByteBuffer>)body;

switch (message.Key) {

     default: Util.CallMethod("Network", "OnSocket", message.Key, message.Value); break;

}

它把所有的从NetworkManager接收到的数据一股脑的全Post给了Lua的Network模块，（其实如果你C#有需要，这里可以分开，这也是为啥从NetworkManager还要传递给SocketCommand的原因）让Lua再次分发，那我们看下Lua/Logic/Network.lua文件中模块怎么写的：

这个Lua文件，在一开始Network.Start() 就注册了大量的事件监听，当然在最后Network.Unload()，也要有移除监听，好习惯。

--Socket消息--

function Network.OnSocket(key, data)

    Event.Brocast(tostring(key), data);

end

因为在开始添加了监听，所以当c#掉上面这个Lua函数的时候，它就只需要广播一下消息来了，就好了，会有相应的监听函数去处理，那我们这个二进制的消息根据监听代码可以知道，应该是 function Network.OnLogin(buffer)，为啥叫OnLogin，其实应该叫OnData是吧，都行吧，我这个人随意，我的框架场景名也叫Login，主要是配合它，Login场景自然要有Login消息数据了，您可以随意改~

在这个函数里面，通过一个define.lua中定义的全局变量TestProtoType，来确定当前框架用那种协议测试。

TestProtoType = ProtocalType.BINARY;

**BINARY = 0,  PB\_LUA = 1,  PBC = 2,  SPROTO = 3,**

我们看到二进制的数据传递给了TestLoginBinary函数：this.TestLoginBinary(buffer);

那这个函数当中其实还有一个区分协议的依据，就要根据读取到的一个字节（下面代码中的protocal）来区分：

function Network.TestLoginBinary(buffer)

    local protocal = buffer:ReadByte();

    local str = buffer:ReadString();

    log('TestLoginBinary: protocal:>'..protocal..' str:>'..str);

end

开头这篇较长，后面的3种就不重复这些了。

借楼层更新-----------------------------------------------------------------------------------------

说下pblua，上面基础性的消息流程基本上已经都走通了，就不重复了，不明白的把上面帖子看明白。这篇帖子直接讲下protobuf\_lua\_gen

（2）protobuf\_lua\_gen：ULUA支持它也很早了，不过当时的ULUA还不是很成熟，蒙哥的csotolua还在发展过程当中，让 ULUA支持它的应该是群里面的大神: Chiuan 大C，后来教给我，当时需要修改很多地方，对于网络协议来说无非就是“进出”操作，protobuf\_lua\_gen在lua中序列化出来的字符串，local msg = login:SerializeToString();  这个msg其实是个c语言的char\*，这是我发现的，因为直接不能传给C#，所以需要在ULUA接收、通讯的地方，一个是需要将byte[]通过CopyMemory出来，另一端从c#拿到byte[]需要通过pushlstring将数据压进去，这样才完成了一次echo操作，当时真的是很痛苦的经历。幸好蒙哥csotolua有了LuaStringBuffer类，这个类没有物理文件，在uLua/Source/Base/LuaWrap.cs中，需要的同学可以看下这个文件。它简化了这种操作使char\*传到c#这边就变成了LuaStringBuffer对象，我们再从它里面拿到byte[]类型的成员buffer，就省去了之前好多繁杂的互传操作。好了，我们看下Lua怎么发送这种类型的数据给服务器端，上PromptCtrl.lua中的代码：

--测试发送PBLUA--

function PromptCtrl.TestSendPblua()

    local login = login\_pb.LoginRequest();

    login.id = 2000;

    login.name = 'game';

    login.email = 'jarjin@163.com';

    local msg = login:SerializeToString();

    ----------------------------------------------------------------

    local buffer = ByteBuffer.New();

    buffer:WriteShort(Login);

    buffer:WriteByte(ProtocalType.PB\_LUA);

**buffer:WriteBuffer(msg);**

    NetManager:SendMessage(buffer);

end

新版框架唯一需要说明的是，在ByteBuffer类中，我新增加了一个函数WriteBuffer，它实际上接收的就是pblua通过SerializeToString序列化出来的char\*，并且将它传递给c#层，到了C#层的ByteBuffer.cs类中，我们可以看到它的实现：

        //读buffer

        public void WriteBuffer(LuaStringBuffer strBuffer) {

            WriteBytes(strBuffer.buffer);

        }

        //写buffer

        public LuaStringBuffer ReadBuffer() {

            byte[] bytes = ReadBytes();

            return new LuaStringBuffer(bytes);

        }

很简洁吧，直接将byte[]类型的buffer成员写到字节数组里面去，然后整个类通过ToArray统一变成byte[]，发送给服务器端。服务器也是按位读出该有的数据，然后将数据传回来，然后通过上个帖子的消息传递流程走到Lua的Network模块里面，然后进行解析：

function Network.TestLoginPblua(buffer)

    local protocal = buffer:ReadByte();

    local data = buffer:ReadBuffer();

    local msg = login\_pb.LoginResponse();

    msg:ParseFromString(data);

    log('TestLoginPblua: protocal:>'..protocal..' msg:>'..msg.id);

end

这个函数里面，也会相对应的增加了ReadBuffer的函数，也就是将byte[]包装成了LuaStringBuffer，然后Post给Lua就变成了pblua需要的char\*，然后将其msg:ParseFromString(data);解析出来，就变成我们最终需要的Lua的Table格式，便可直接访问里面的数据成员，如msg.id了。这个流程操作也就顺利完成了。

pblua使用的是将protobuf协议文件编码成lua文件，然后在lua程序中require进来，赋值，序列化，发送。那它的编码工具，ulua.org提供了测试版，因为protobuf用的2.4.1老版本。测试可以，使用可以用心版本，下载地址为：[http://www.ulua.org/download.html](http://www.ulua.org/download.html" \t "http://doc.ulua.org/article/ngui/_blank)

**[protobuf-2.4.1.zip (d:/protobuf-2.4.1)](http://web01264.w31.vhost002.cn/downloads/protobuf-2.4.1.zip" \t "http://doc.ulua.org/article/ngui/_blank)**

**[protoc-gen-lua.zip (d:/protoc-gen-lua)](http://web01264.w31.vhost002.cn/downloads/protoc-gen-lua.zip" \t "http://doc.ulua.org/article/ngui/_blank)**

请按照给定的路径放置，然后启动一个命令行到D:\protobuf-2.4.1\python 目录下，（确保已安装python 2.7.3，并且bin目录添加到环境变量path中）输入下面命令：

**python setup.py build**

**python setup.py install**

编译：uLua/Editor/Packager.cs里面BuildProtobufFile函数。撑不下了，待续！

借楼层更新-----------------------------------------------------------------------------------------

上面还是说的多了，一个帖子没有撑下，这个帖子就不 废话了，看来装逼是有代价的，赶快写完去LOL才是王道。

（3）PBC：它是云风大神早期的一个对protobuf的解析库，相对于protobuf\_lua\_gen来说，不需要生成巨多的lua协议描述文件，可以直接读取protobuf官方代码编译出来的protoc.exe生成的pb二进制文件，简洁效率还高。所以我们的项目也采用了pbc作为我们的主要通讯协议层。pbc有两种解析模式：（A）直接读取二进制的pb文件，从里面获取协议描述信息，（B）通过lpeg直接解析协议描述文件。应该是前者效率高于后者。

顺便在解析代码之前说下pbc专有的东西，很多人不知道怎么编译pb二进制文件，其实挺简单的，到官网去下载protobuf的c源码，编译出protoc.exe来，然后通过下面的命令行编译出pb文件：

**protoc.exe -o 目标文件 源文件**

挺简单的吧？我协议的用的版本较老了2.4.1吧，如果为了测试可以去ulua.org去下载测试：[http://www.ulua.org/download.html](http://www.ulua.org/download.html" \t "http://doc.ulua.org/article/ngui/_blank)

**[protobuf-2.4.1.zip (d:/protobuf-2.4.1)](http://web01264.w31.vhost002.cn/downloads/protobuf-2.4.1.zip" \t "http://doc.ulua.org/article/ngui/_blank)**，对了顺便说下，使用pblua也需要它的源码编译出来的protoc.exe

开始看代码，继续打开lua/Controller/PromptCtrl.lua文件，看下怎么封装pbc的：

--测试发送PBC--

function PromptCtrl.TestSendPbc()

    local path = Util.DataPath.."lua/3rd/pbc/addressbook.pb";

    local addr = io.open(path, "rb")

    local buffer = addr:read "\*a"

    addr:close()

    protobuf.register(buffer)

    local addressbook = {

        name = "Alice",

        id = 12345,

        phone = {

            { number = "1301234567" },

            { number = "87654321", type = "WORK" },

        }

    }

    local code = protobuf.encode("tutorial.Person", addressbook)

    ----------------------------------------------------------------

    local buffer = ByteBuffer.New();

    buffer:WriteShort(Login);

    buffer:WriteByte(ProtocalType.PBC);

    buffer:WriteBuffer(code);

    NetManager:SendMessage(buffer);

end

这是直接读取pb二进制文件，直接用lpeg解析描述文件的例子，我没提供，你需要到pbc源码里面去找。流程是，打开pb文件-》注册描述数据结构-》填充数据-》编码，还是很清晰的逻辑。解码就不用我说了吧。直接打开Network.lua查看下面代码：

--PBC登录--

function Network.TestLoginPbc(buffer)

    local protocal = buffer:ReadByte();

    local data = buffer:ReadBuffer();

    local path = Util.DataPath.."lua/3rd/pbc/addressbook.pb";

    local addr = io.open(path, "rb")

    local buffer = addr:read "\*a"

    addr:close()

    protobuf.register(buffer)

    local decode = protobuf.decode("tutorial.Person" , data)

    print(decode.name)

    print(decode.id)

    for \_,v in ipairs(decode.phone) do

        print("\t"..v.number, v.type)

    end

    log('TestLoginPbc: protocal:>'..protocal);

end

使用pbc有个非常重要的一个问题，我不知道应不应该定位为bug，至少“云风大神”认为是个问题，但是不想改，我这里也附上解决方案，假如你有这么个消息，里面有个数组类型的repeat类型的数据字段，服务器连续发给你2次这个消息，而且里面一个成员字段的值都没有的话，pbc这两次给你的结构table是同一个内存指针地址，而不是每次都新分配的table。解决方案：你在接收这个数据之前，自己提前将数据表结构布局好，至少被copy的数据table要有个自己新建的空表{}，才能导致数据不会串。如下：

local data["aaa"] = {};

data["aaa"] = buffer.data;

下一篇接着说spoto，待续...

从接触PureMvc的历程来看，显示惊喜，在讨厌，再到喜欢~

借楼层更新-----------------------------------------------------------------------------------------

终于到了最后一个sproto了，这次可以少废话点了吧，我觉得可以，为啥？没怎么用过，哈哈经验不足，就简单描述下echo流程，今天上午的帖子就不写了。下午补些知识点帖子。

（4）sproto：这个是云风大神的新作品，号称比pbc包小、效率高效。因为这是大神自己创建出来的协议格式，不是protobuf的解析库。目前我知道的朋友们用它的也开始多起来，使用体验如何？我不清楚，没怎么用过，希望你们使用完了，告诉我下。

咱们继续打开lua/Controller/PromptCtrl.lua，看下里面关于sproto如何封包，发送出去的代码：

--测试发送SPROTO--

function PromptCtrl.TestSendSproto()

    local sp = sproto.parse [[

        --此处省略sproto描述字符串，帖子有字数限制，请直接参考lua代码--

   }

    local code = sp:encode("AddressBook", ab)

    ----------------------------------------------------------------

    local buffer = ByteBuffer.New();

    buffer:WriteShort(Login);

    buffer:WriteByte(ProtocalType.SPROTO);

    buffer:WriteBuffer(code);

    NetManager:SendMessage(buffer);

end

其实还是挺简单的，思路如pbc的一样，封包、发出去。然后我们继续看下Network里面如何解析的：

--SPROTO登录--

function Network.TestLoginSproto(buffer)

    local protocal = buffer:ReadByte();

    local code = buffer:ReadBuffer();

    local sp = sproto.parse [[

    .Person {

        name 0 : string

        id 1 : integer

        email 2 : string

        .PhoneNumber {

            number 0 : string

            type 1 : integer

        }

        phone 3 : \*PhoneNumber

    }

    .AddressBook {

        person 0 : \*Person(id)

        others 1 : \*Person

    }

    ]]

    local addr = sp:decode("AddressBook", code)

    print\_r(addr)

    log('TestLoginSproto: protocal:>'..protocal);

end

解析消息也超简单，也是decode，然后就是你想要的lua表数据结构。

终于废话完了，希望协议的系列帖子能帮你解决你手头的需求。。。