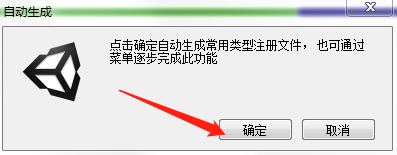
Lua文档

目录

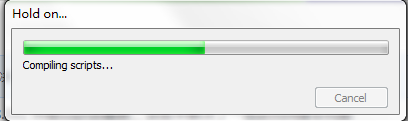
[导入tolua框架 2](#_Toc16694643)

# 导入tolua框架

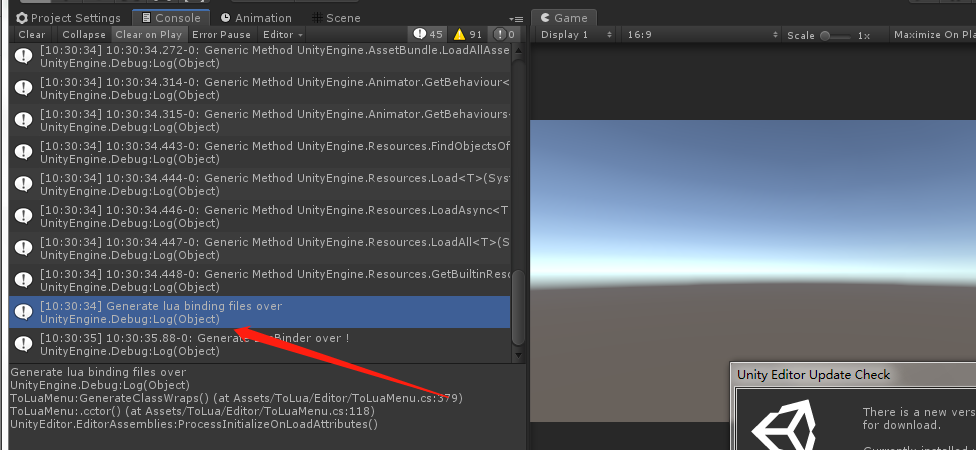


点击确定

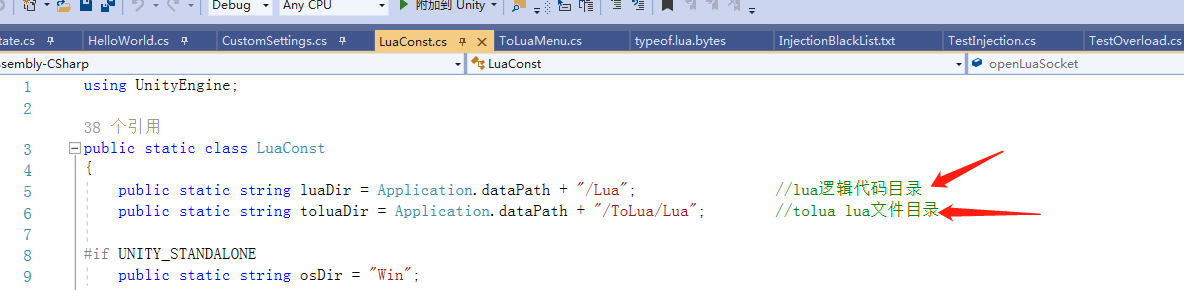
然后，自动，进行，开始导入的工作：



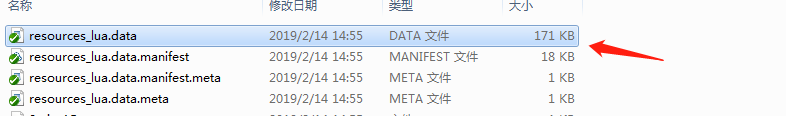
Lua脚本的编辑。



Lua代码的逻辑代码和框架代码：



在我们Lua脚本打包时，会，先将Lua脚本，打包成.byte的文件格式，之后，在将.byte的所在的文件夹，打一个文件的包，上传。

读取AB包，需要，先将Lua脚本全部解析至 内存中，便于快速的调用。（所占用的内存不大）

100多KB，类似于加载一张图片的大小。

之后，根据名称，找到对应的Lua脚本，执行 即可。（参考）

暂时没有找到对应的Lua编辑工具。（暂时使用的是SubLime Text的编辑器）

通过变量，控制是否是使用本地包，或者是使用的网络包

IsLocal = true；

然后根据，变量名称，直接修改 读取字节时的代码：



搭建一个，可以随时移植到其他项目中的Lua功能列表。

将脚本和预制件，关联，以及 脚本的可扩展性，需要挂载在物体上之后，修改对应的参数，可以读取 不同的Lua脚本

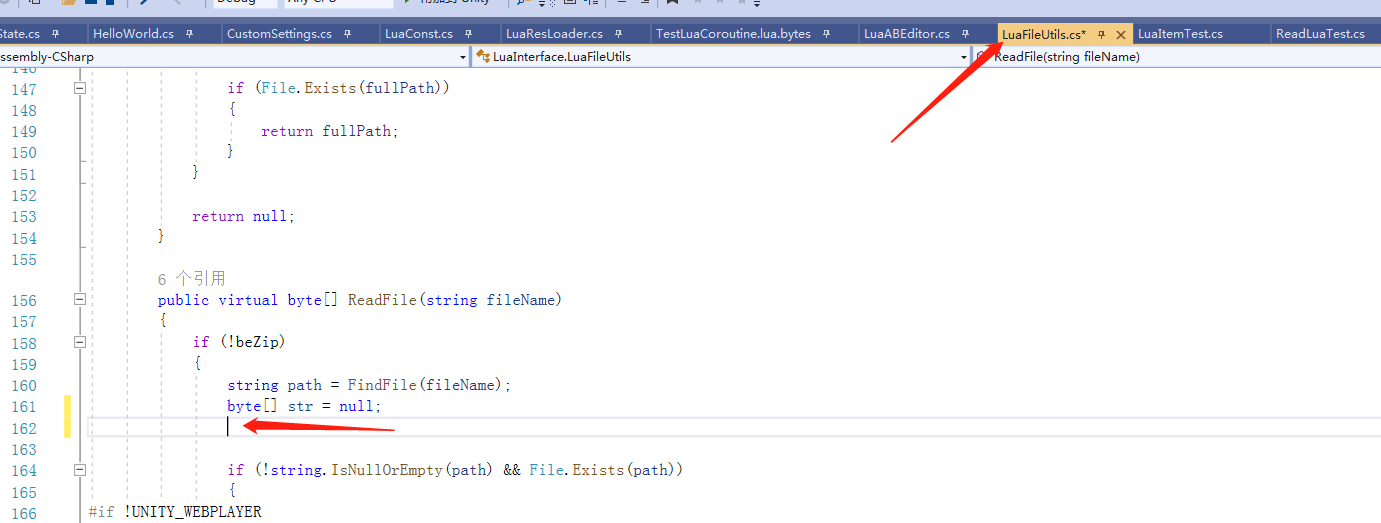
Lua 脚本之间的通信，比如LuaFunction的应用，（C#中，调用Lua中的方法，也可以是，Lua调用C#方法后的回调）

Lua脚本的打包

1. 将Lua脚本，复制到同一个目录下
2. 将该目录，打包成AB
3. 读取是，按照键值对的文件名称去读取即可。

将Lua模块独立出来，建立单独的

在这个地方，区分，是AB包，还是本地包，既可以咯



每次state.DoFile("TestPerf.lua");

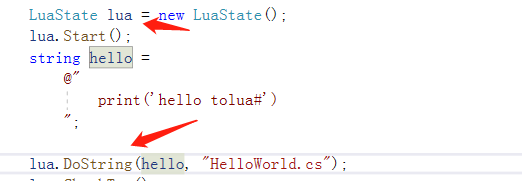
doFile 都会调用，读取那一块。

关键的代码。--主要用于将Lua代码加载出来，使用

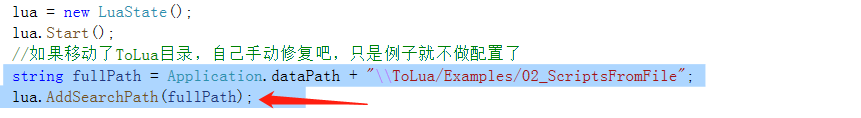
LuaState 去调用Lua代码，入围Lua代码的入口。

需要二次封装，用于方便调用。

初始化都需要参数：



如果需要添加，Lua脚本检索路径：一般在本地时有作用，如果是网络包，直接读取字典中的对应的文件名字节，就可以咯。



不停的执行，刷新的脚本



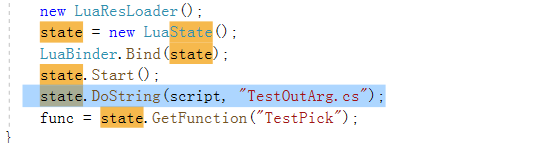


执行这个脚本时，需要，先在编辑器下，执行：

use menu Lua->Copy lua files to Resources.

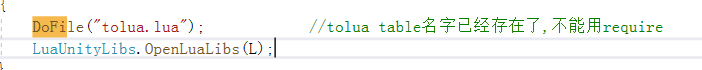
否则，可能在手机上，看不到这种效果。

LuaResLoader，使用这种读取方式时。



Dofile和dostring，需要了解这两个函数的作用

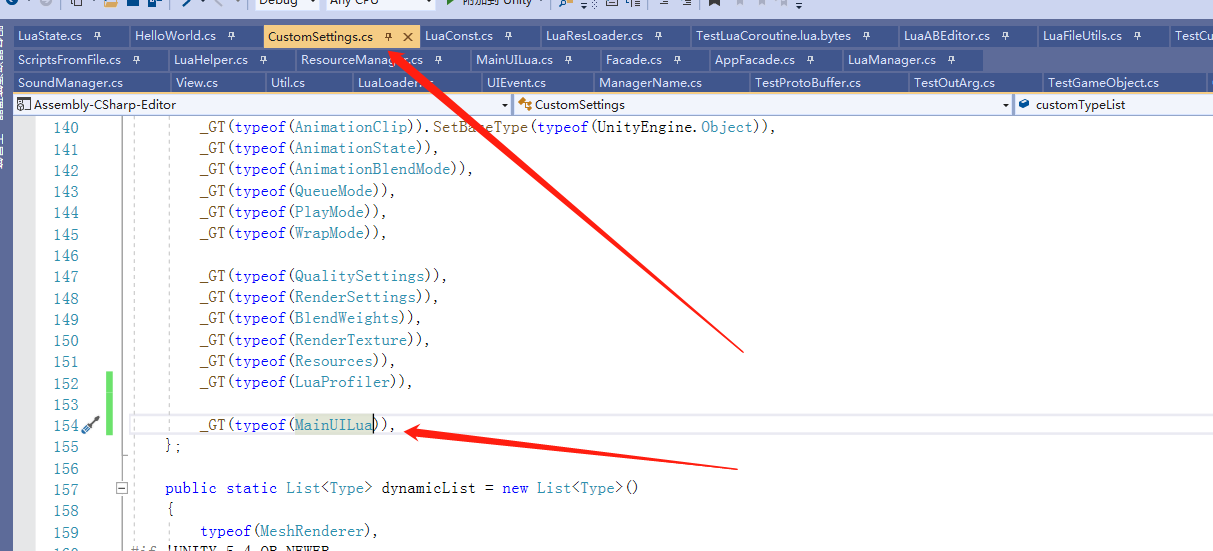
什么时候使用require，



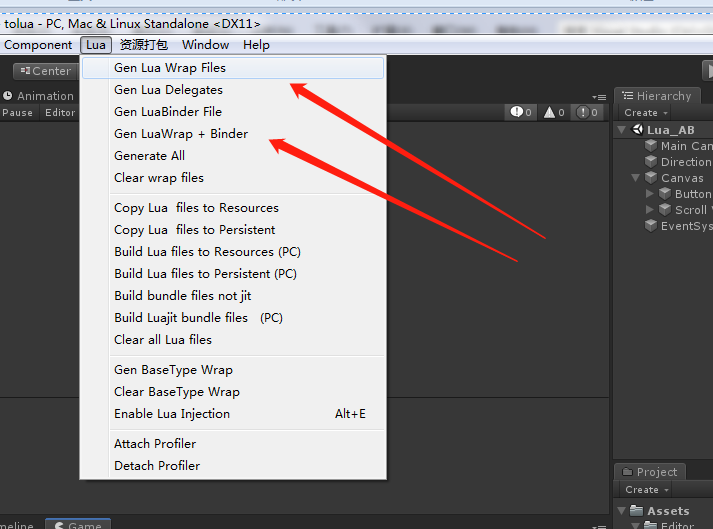
初始化基类，用于

如果，在lua中，访问不到对应的属性变量，需要将对应的脚本组件，进行wrap。

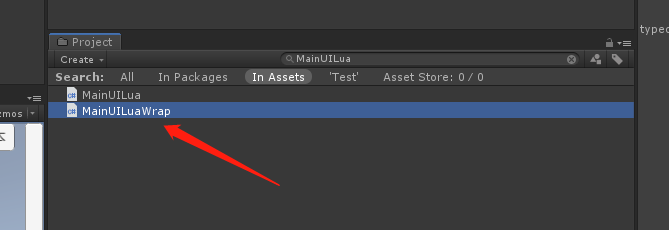
现在CustomSetting的脚本中，添加需要wrap的组件脚本，



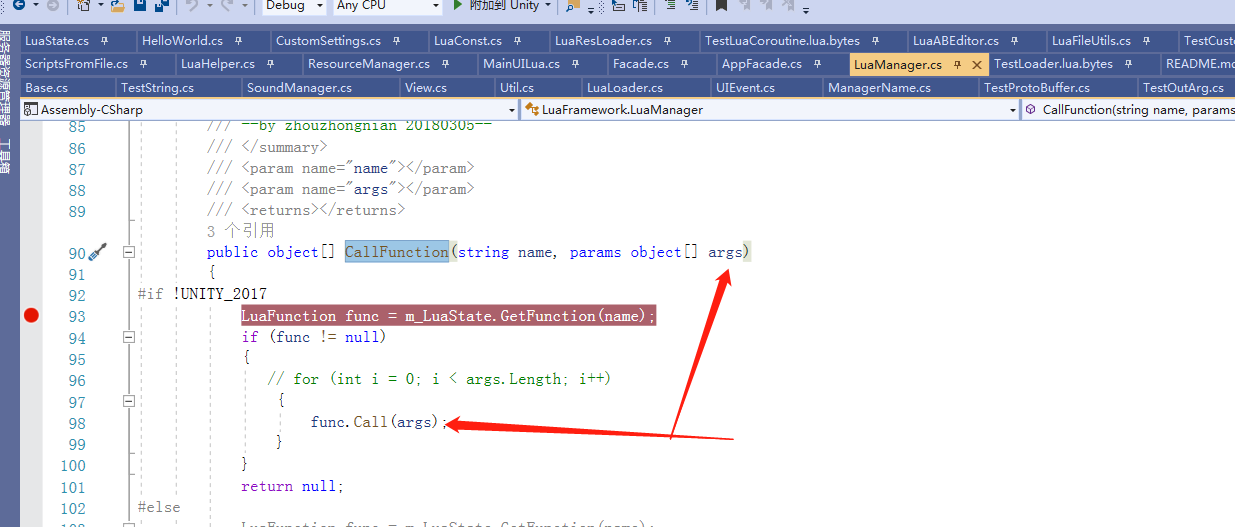
然后，点击 wrap即可。



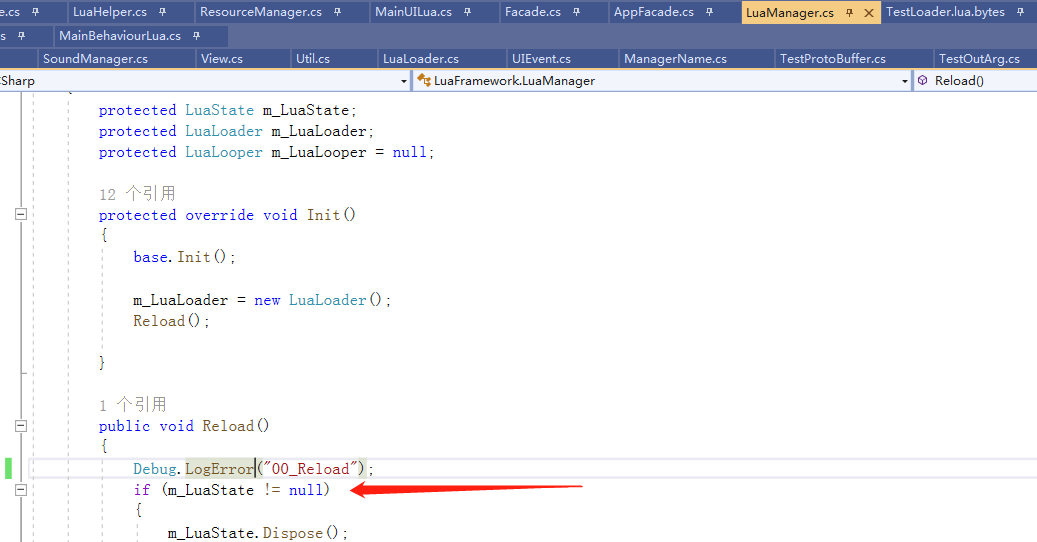
执行完之后，会出现xxxxwrap的这个文件



这里的参数，是个参数列表。所以在Lua脚本中解析时，注意是解析的数组结构。

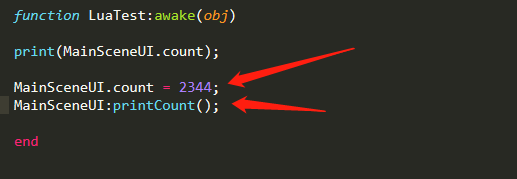


//  使用同一个的LuaState

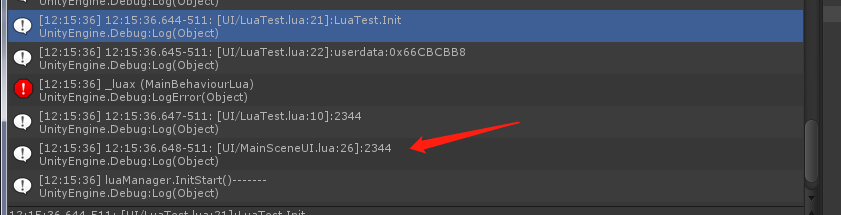


只调用了一次的LuaState，那么调用的所有的Lua脚本中，都可以相互关联的。

例如，在调用MainSceneUI.lua脚本时，成功后，调用了LuaTest.lua脚本，在LuaTest脚本中调用，其MainSceneUI的lua脚本中的参数，



输出的结果，如下。



结论：

只要LuaState是同一个，那么，其中的Lua脚本中的相关的变量和逻辑脚本，可以互相影响和调用。

（可以在一个项目中，只实例化一个LuaState，用来保证，整个项目中lua的各个脚本之间的关联性。）

如果是相同的脚本，被加载两次，就是被覆盖（覆盖原来的Lua中的所有的变量和属性值）

为了解决这一问题，

1.保留原来的覆盖原来已经加载好的Lua脚本。

2.如果Lua脚本已经被加载好，直接调用原来初始化成功后的Lua脚本。（多用于在跳转场景时，销毁了，很多的Lua中的静态变量参数，结果在新的场景中，继续引用其变量导致的错误）

如果是多模块时，就是有多个不相关联的Lua脚本，我们可以采用

一些常用的方法，

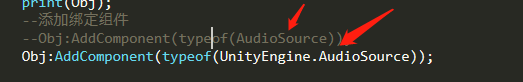
组件的获取：

Util.GetComponent(this.southLoading, "UnityEngine", "RectTransform");

this.instance.gameObject:GetComponent(typeof(DragUIBox));

self.transform:Find("UIPanel/ARLock"):GetComponent(typeof(UnityEngine.UI.Image));

添加组件，必须添加，命名空间



物体的查找：

this.transform:Find

this.transform:Find("CanvasAR/AR\_SaoYiSao/BackUp")

事件的绑定（按钮）

UIEvent.AddButtonClick(btn.gameObject, \_Act);

查找GameObject对象

Util.FindGameObject("Loading");

加载资源

Util.LoadResource("Texture/AR\_SaoYiSao/锁2副本","UnityEngine","Texture2D");

动态创建

动态弹窗设计与事件回调

日志打印

Print（）

初始化，类似于构造函数的那种。主要是关键字setmetatable.

function BaseView:new()

return setmetatable({ transform = nil, }, { \_\_index = self });

end

类似于基类。（然后使用new 实例化出来）

Self.transform 这样，可以直接使用

一般，定义一个Lua脚本的架构是这样的。

local BaseView = {};

function BaseView:new()

return setmetatable({ transform = nil, }, { \_\_index = self });

end

。。。。

return BaseView;

local DataManager = {};

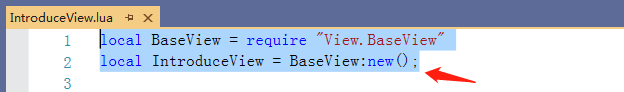
DataManager.\_\_index = DataManager;

function DataManager:new()

return setmetatable({ isLoaded = false, isLoading = false }, self);

end

调用的时候，（将BaseView当成基类，来调用）



设置物体的显示和隐藏

self.ReturnAR.gameObject:SetActive(bool);

协成等待的写法

WaitForEndOfFrame();

StartCoroutine(function () self:LoadAsync(); end);

function DataManager:LoadAsync()

lua中的while循环

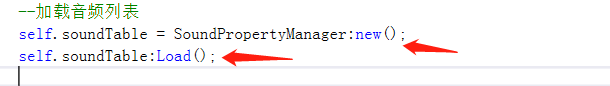
while self.isLoading do

WaitForEndOfFrame();

End

初始化，之后，就可以调用Lua脚本中的公用的方法了





soundTable，类似于这个对象，可以被创建多个。

Lua对应类中的for循环（经常使用的）

for i = 1, table.getn(self.events) do

if self.events[i].isRunning then

self:DropEvent(self.events[i]);

end

end

数组数据的插入

event = {};

table.insert(self.events, event);

类似资源调用，或者方法调用的方面，尽量在C#中统一处理，便于，代码的美观。

新创建的Lua脚本，不是通过C#的代码去调用的。

这个是时候，需要实例化一个lua脚本，就要通过

function DataPropertyManager:new()

return setmetatable({ set = {} }, { \_\_index = self} );

end

类似于这种方式去实现。

在需要该脚本的地方，直接，require后，new 就可以，直接使用其中的方法，以及对应的属性变量。

譬如创建了一个脚本：

--调用其他lua脚本，并实例化

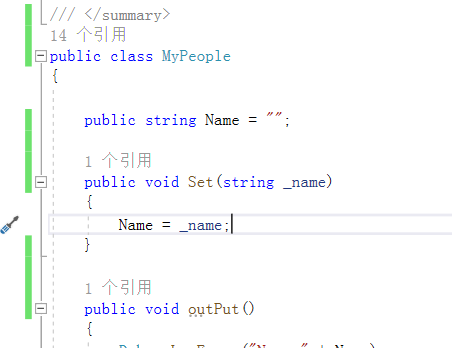
self.LuaComponent = Luacomponent:new();

self.LuaComponent:getComponent(obj.gameObject);

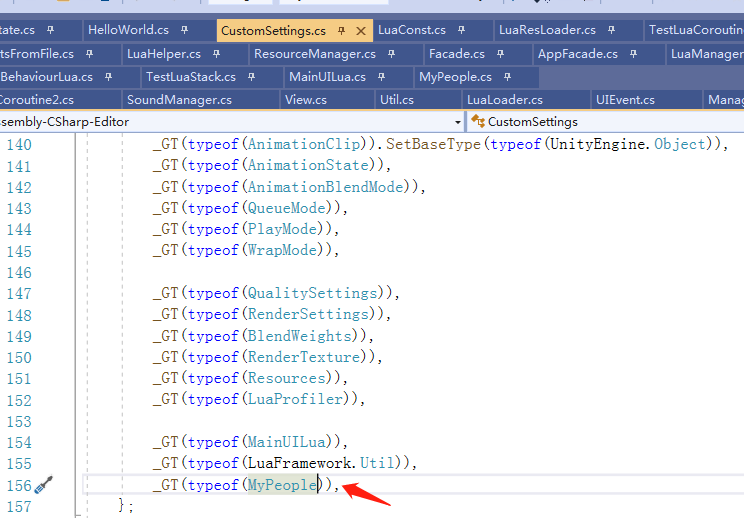
类似这样的，就相当于，创建的非Mon的脚本的class类，在这边的实例化。

那么，C#中的Class类，在Lua中，怎么被实例化

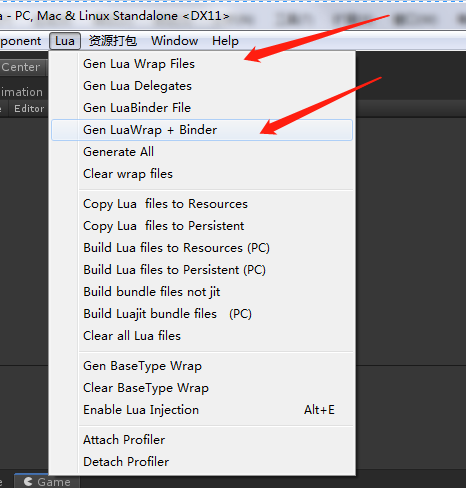
首先创建C#的Class类，MyPeople



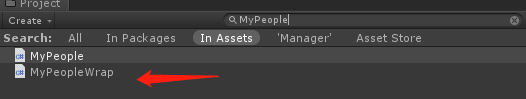
添加lua组件化脚本



点击，生成

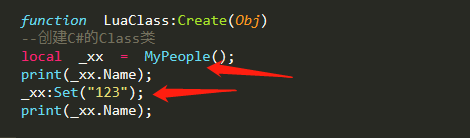


这个时候，在project的资源列表中，就会多一个文件



说明，此时，可以在Lua脚本中，使用该脚本。

接下来，在Lua 脚本中，使用该C#脚本，包括实例化的过程。

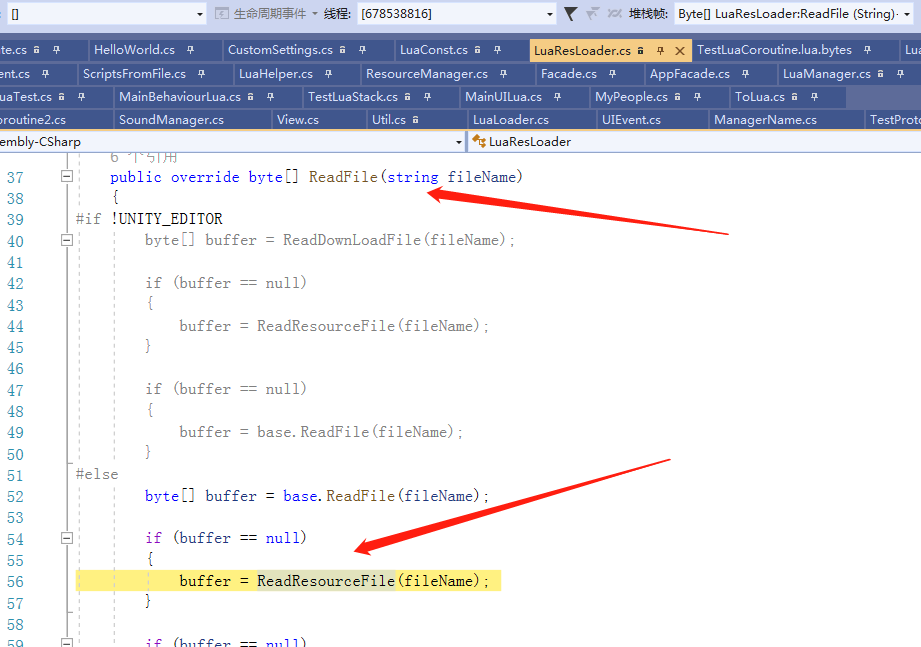


在Lua中，不需要关键字new，直接 MyPeople（）就可以咯，这里，如果调用该类中的方法时，要使用\_xx:Set("123"); 冒号的方式，这个经常会犯这种错误。

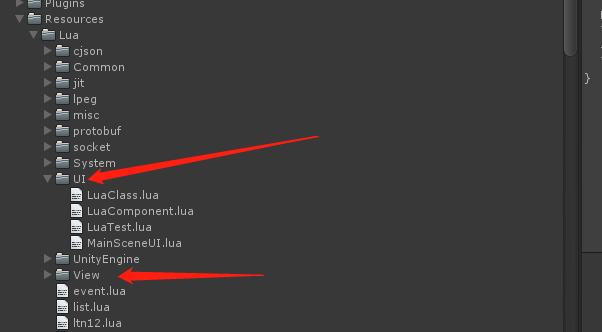
这里的类，没有命名空间，所以直接MyPeople（）。如果是含有命名空间的类，则需要，在前面加上命名空间，来使用。

添加 Lua脚本的查找，这样，就不需要，关系，你的lua脚本放在什么地方，都能被找到（），但是如果打包成AB包，的话，就不需要去检索文件夹，节省了查找的时间。如果是放在Resource中的话，需要（但是用的很少，基本上不用，测试的时候，使用，（就是在手机上测试，该Lua脚本是否可行的时候，使用该方式））

如果，没有找到对应的Lua脚本，就会去找Resource中的Lua脚本



此时的路径是这样的



所以需要，新建一个Lua脚本路径的txt，用于灵活读取，resource中的脚本内容

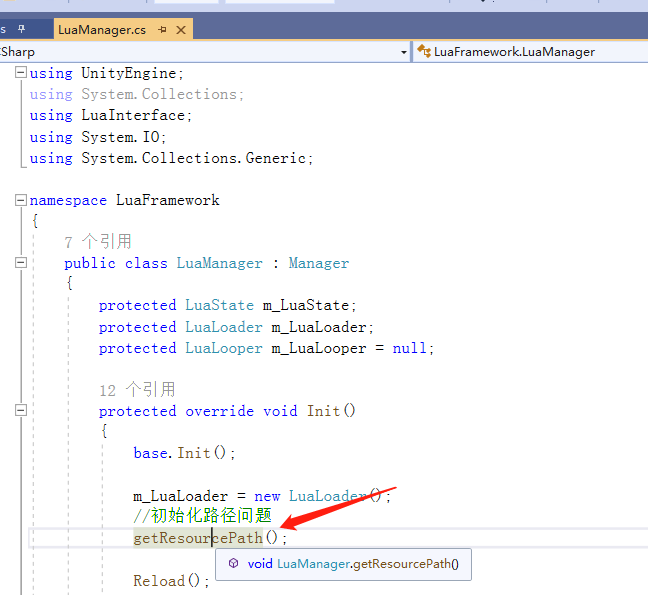
luaList。

这个配置文件在



生成Lua字节代码的时候，创建和写入。

读取时，在



只初始化一次，

读取配置表中的数据，只处理dofile中的事件，因为



解决，脚本查找的路径问题，现在改成自动查找，便于本地的脚本的执行调试和打包的执行调试。

只有在本地找不到的情况下，才会去读取Resource下的lua脚本资源

插播：

单例模式（单例模式，是一种特殊的静态）

public class Singleton<T> : MonoBehaviour where T : MonoBehaviour

{

public static T Instance

{

get

{

if (\_Instance == null)

{

\_Instance = FindObjectOfType<T>();

}

return \_Instance;

}

}

static T \_Instance;

}

制作Demo。

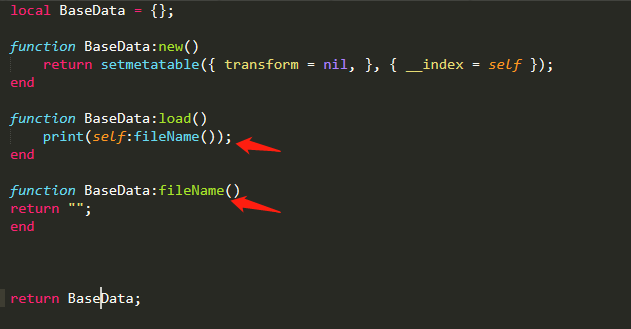
包含协成的demo。

包含获取组件的Demo

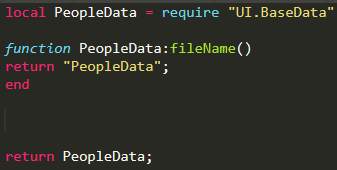
# Lua脚本的父子继承关系

新建一个lua脚本：

父类脚本：



子类脚本：



调用时



