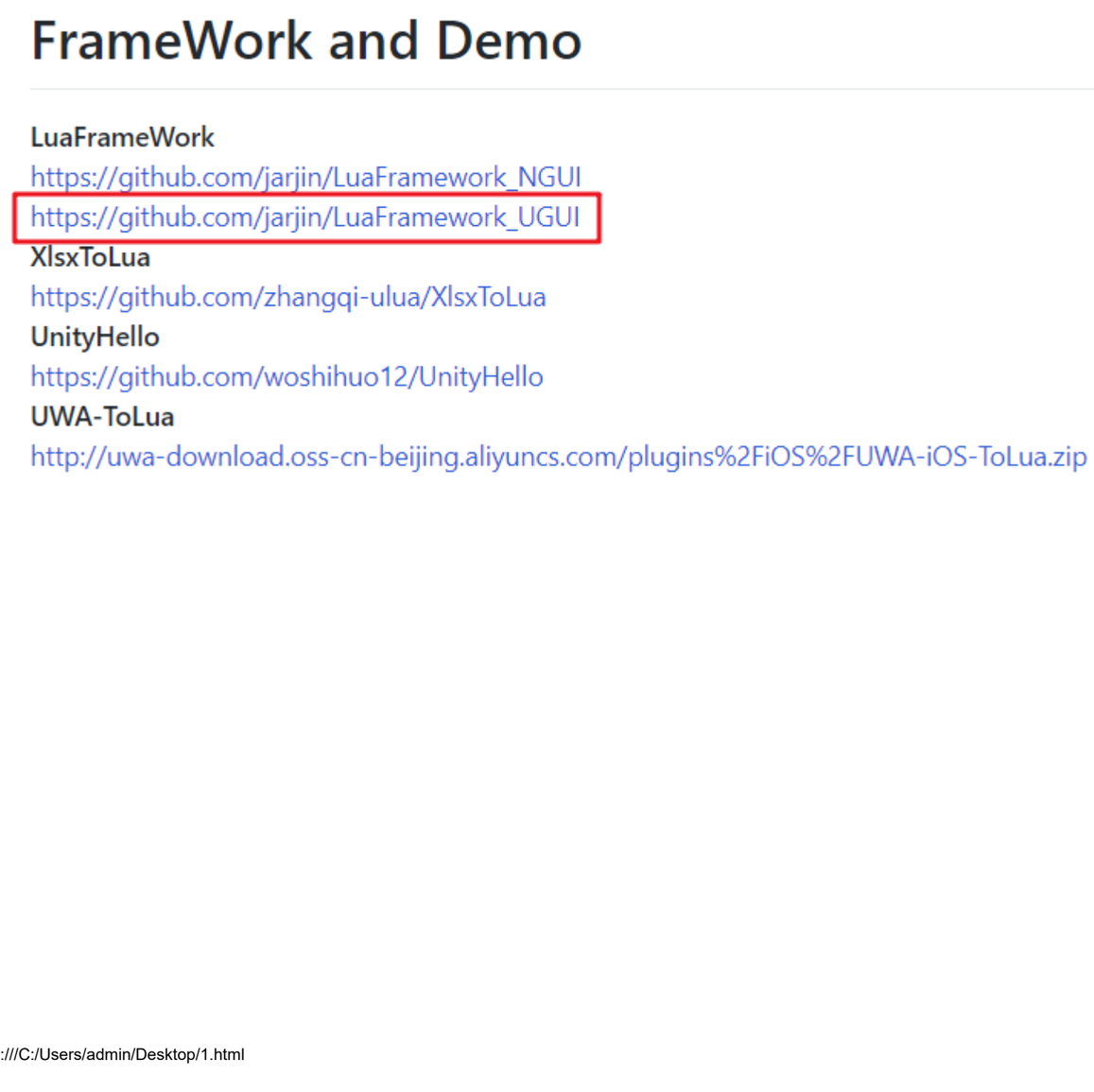
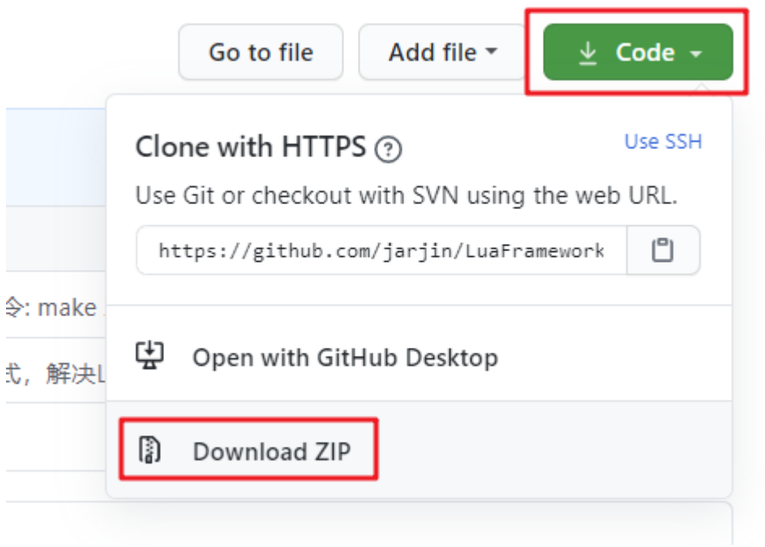
文章目录

一、tolua下载

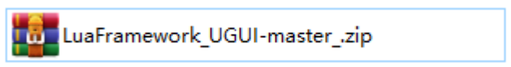
`tolua` 的 `GitHub` 下载地址：https://github.com/topameng/tolua



假设我们下载的是LuaFramework_UGUI，它是基于 Unity 5.0 + UGUI + tolua 构建的工程



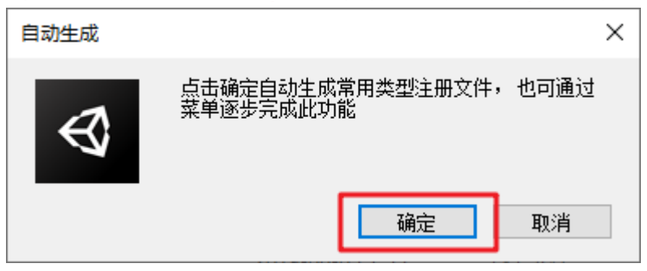
下载下来得到一个LuaFramework_UGUI-master.zip



二、运行Demo

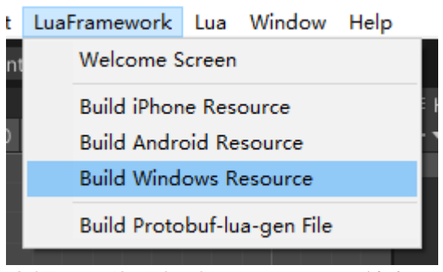
1、生成注册文件

解压之后就是一个 Unity 的工程，直接用 Unity 打开，首次打开工程会询问生成注册文件，点击确定即可



2、将lua打成AssetBundle

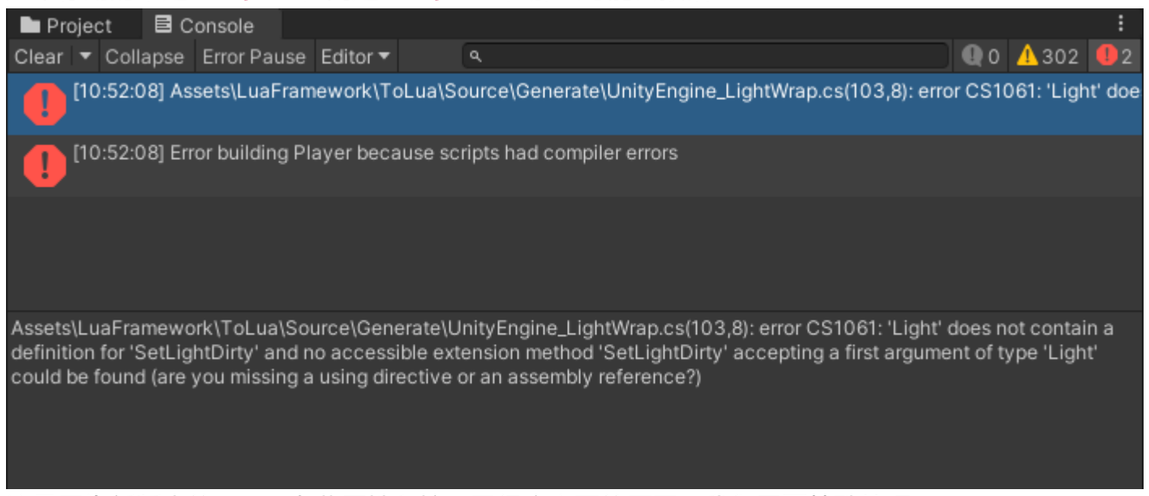
首先要执行 lua 资源的生成（打 AssetBundle ），点击菜单 【LuaFramework】-【Build Windows Resource】



会把 lua 代码打成 AssetBundle 放在 StreamingAssets 中。

3、解决报错

如果你用的不是 Unity5.x ，而是 Unity2020 ，那么可能会报错：



这是因为新版本的 Unity 有些属性和接口已经废弃了的原因，我们需要特殊处理一下

一个是 Light 类，一个是 QualitySettings 类，这两个类我们一般不需要在 lua 中使用，所以我们不对他们生产 Wrap 即可：

1. 打开 CustomSettings.cs ，把 _GT(typeof(Light)), 和 _GT(typeof(QualitySettings)), 这两行注释掉
2. 然后单击菜单 【Lua】-【Clear wrap files】 清理掉 Wrap
3. 然后再单击菜单 【Lua】-【Generate All】 重新生成 Wrap ，
4. 然后再重新点击菜单 【LuaFramework】-【Build Windows Resource】 生成 lua 资源。

执行 【Lua】-【Generate All】 菜单的时候，你可能会报错

定位到报错的位置

ToLuaExport.cs

GenParamTypes.for.else → else

Assembly-CSharp-Editor | ToLuaExport._MethodBase

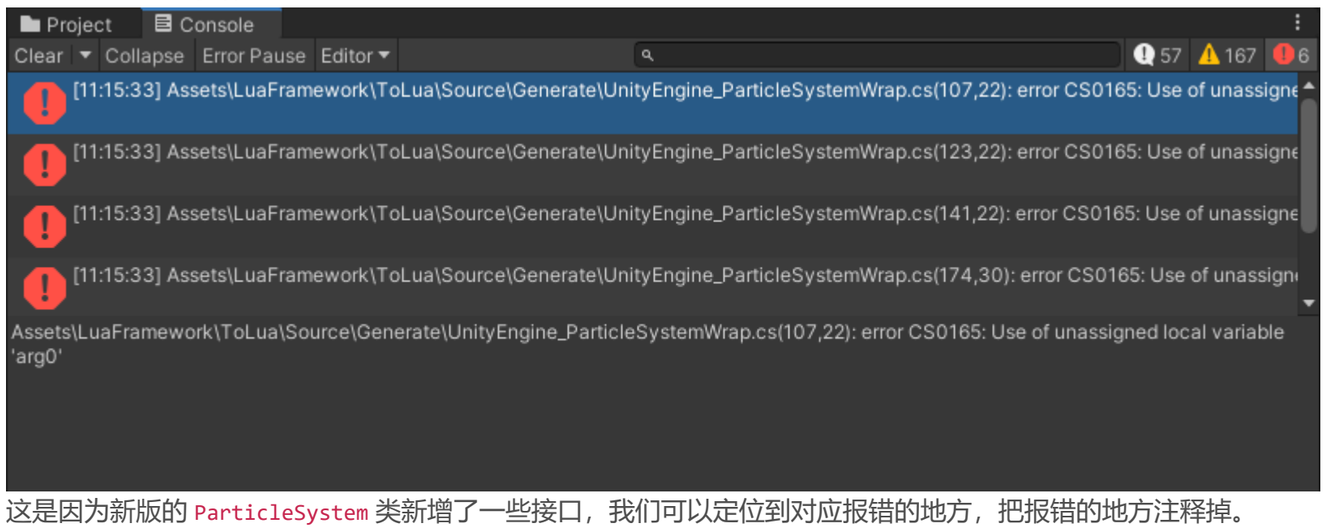
```
        6 个引用|0 项更改|0 名作者，0 项更改
271     public string GenParamTypes(int offset = 0)
272     {
273         StringBuilder sb = new StringBuilder();
274         List<Type> list = new List<Type>();
275
276         if (!method.IsStatic)
277         {
278             list.Add(type);
279         }
280
281         for (int i = 0; i < args.Length; i++)
282         {
283             if (IsParams(args[i]))
284             {
285                 continue;
286             }
287
288             if (args[i].Attributes != ParameterAttributes.Out)
289             {
290                 list.Add(GetGenericBaseType(method, args[i].ParameterType));
291             }
292             else
293             {
294                 Type genericClass = typeof(LuaOut<>);
295                 Type t = genericClass.MakeGenericType(args[i].ParameterType.GetElementType());
296                 list.Add(t);
297             }
298         }
299
```

添加判空

```
291             }
292             else
293             {
294                 Type genericClass = typeof(LuaOut<>);
295                 var elementType = args[i].ParameterType.GetElementType();
296                 if(null != elementType)
297                 {
298                     Type t = genericClass.MakeGenericType(elementType);
299                     list.Add(t);
300                 }
301             }
302         }
303
304         for (int i = offset; i < list.Count - 1; i++)
```

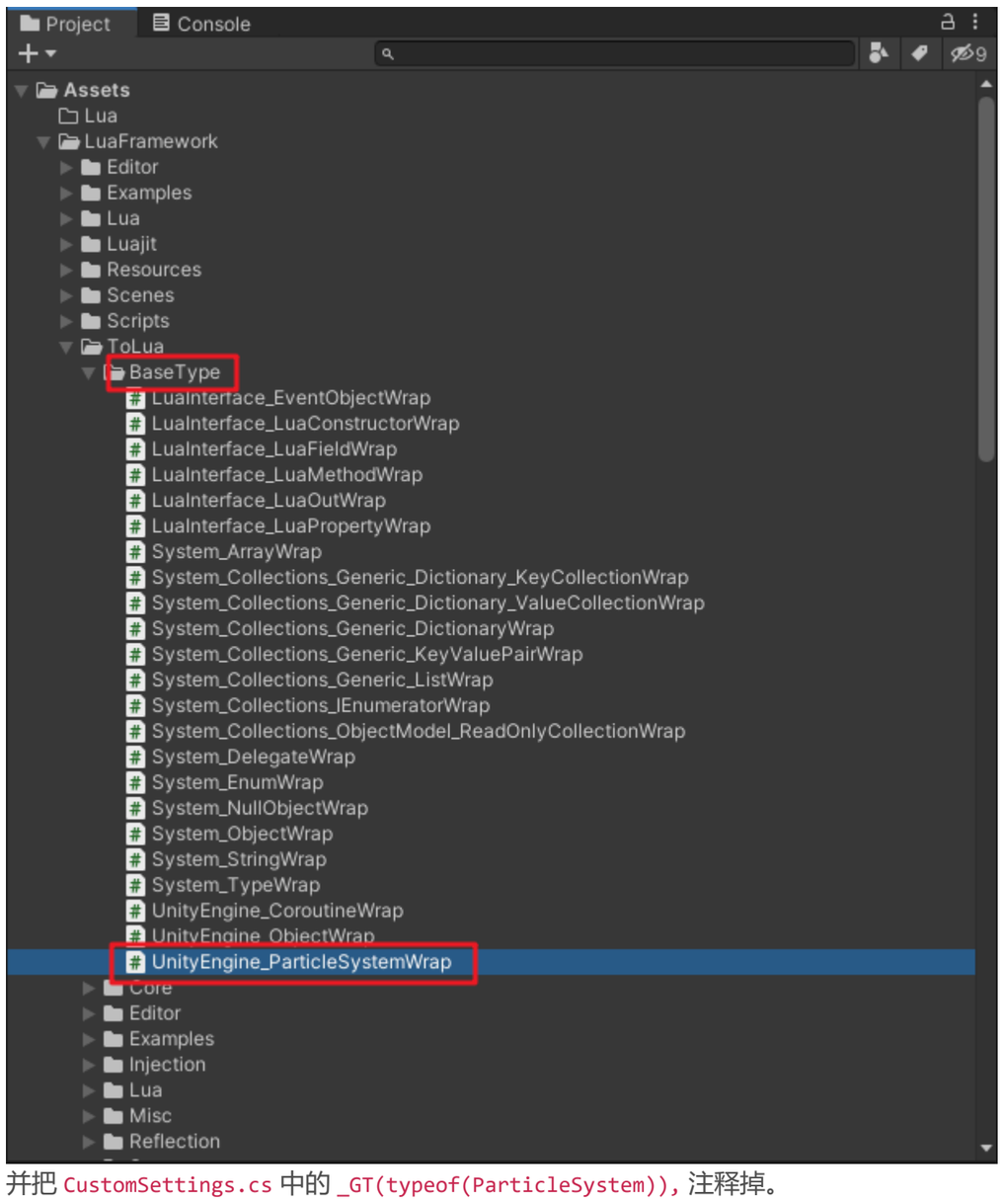
重新执行 `【Lua】-【Generate All】` 菜单

生成后应该还有报错



这是因为新版的 `ParticleSystem` 类新增了一些接口，我们可以定位到对应报错的地方，把报错的地方注释掉。

不过为了防止下次执行 `【Lua】-【Generate All】` 菜单时又被覆盖导致报错，我们可以把 `UnityEngine_ParticleSystemWrap.cs` 移动到 `BaseType` 目录中



并把 `CustomSettings.cs` 中的 `_GT(typeof(ParticleSystem)),` 注释掉。

并在 `LuaState.cs` 注册 `ParticleSystemWrap` 类，要注意调用点要放在对应的 `BeginModul` 和 `EndModule` 之间，是什么命名空间下的，就放在什么 `Modul` 之下，如果是多级命名空间，则是嵌套多个 `BeginModul` 和 `EndModule` 。

```
1 // LuaState.cs
2 void OpenBaseLibs()
3 {
```

```
    // ...

    BeginModul("UnityEngine");
    // ...
    UnityEngine_ParticleSystemWrap.Register(this);
    EndModule();    //end UnityEngine

}
```

同理， `UnityEngine_MeshRendererWrap.cs` 可能也会报错，按上面的处理方式处理。

4、为何一些没有在CustomSettings.cs注册的类也会生成Wrap类

假设我们把某个 `Wrap` 类手动移动到 `BaseType` 目录中，并在 `CustomSettings.cs` 中注释掉对应的 `_GT(typeof(xxx))` ，理论上应该不会生成对应的 `Wrap` 类，但事实上可能还是生成了，为什么?
这是因为 `ToLua` 会将在 `CustomSettings.cs` 中注册的类的父类进行递归生成。
举个例子， `CustomSettings.cs` 中把 `_GT(typeof(Component))` 注释掉，执行 `【Lua】-【Generate All】` 菜单，依然会生成 `UnityEngine_ComponentWrap.cs` ，为什么?
因为在 `CustomSettings.cs` 中有 `_GT(typeof(Transform))` ，而 `Transform` 的父类是 `Component` ，所以依然会生成 `UnityEngine_ComponentWrap.cs` 。
具体逻辑可以看 `ToLuaMenu.cs` 的 `AutoAddBaseType` 函数，它里面就是进行递归生成父类的 `Wrap` 类的。
如果你将 `UnityEngine_ComponentWrap.cs` 移动到 `BaseType` 目录中，并且不想重新生成 `UnityEngine_ComponentWrap.cs` ，可以在 `ToLuaMenu.cs` 的 `dropType` 数组中添加 `typeof(UnityEngine.Component)` 即可，不过不建议这么做，因为这里有个坑!
这个坑就是 `Component` 的子类生成 `Wrap` 类是错误的。举个例子， `Transform` 是继承 `Component` ，生成的 `UnityEngine_TransformWrap` 代码是这样的:

```
public class UnityEngine_TransformWrap
{
    public static void Register(LuaState L)
    {
        L.BeginClass(typeof(UnityEngine.Transform), typeof(UnityEngine.Component));

        // ...
    }
}
```

当你在 `dropType` 数组中添加 `typeof(UnityEngine.Component)` ，那么生成出来的 `UnityEngine_RendererWrap` 是这样的:

```
public class UnityEngine_TransformWrap
{
    public static void Register(LuaState L)
    {
        L.BeginClass(typeof(UnityEngine.Transform), typeof(UnityEngine.Object));

        // ...
    }
}
```

发现没有，会认为 `Transform` 是继承 `Object` ，而事实上， `Transform` 是继承 `Component` 的，这样会导致你在 `lua` 中对于 `Component` 子类的对象无法访问 `Component` 的 `public` 成员、属性和方法。
比如下面这个会报错，提示不存在 `gameObject` 成员或属性。

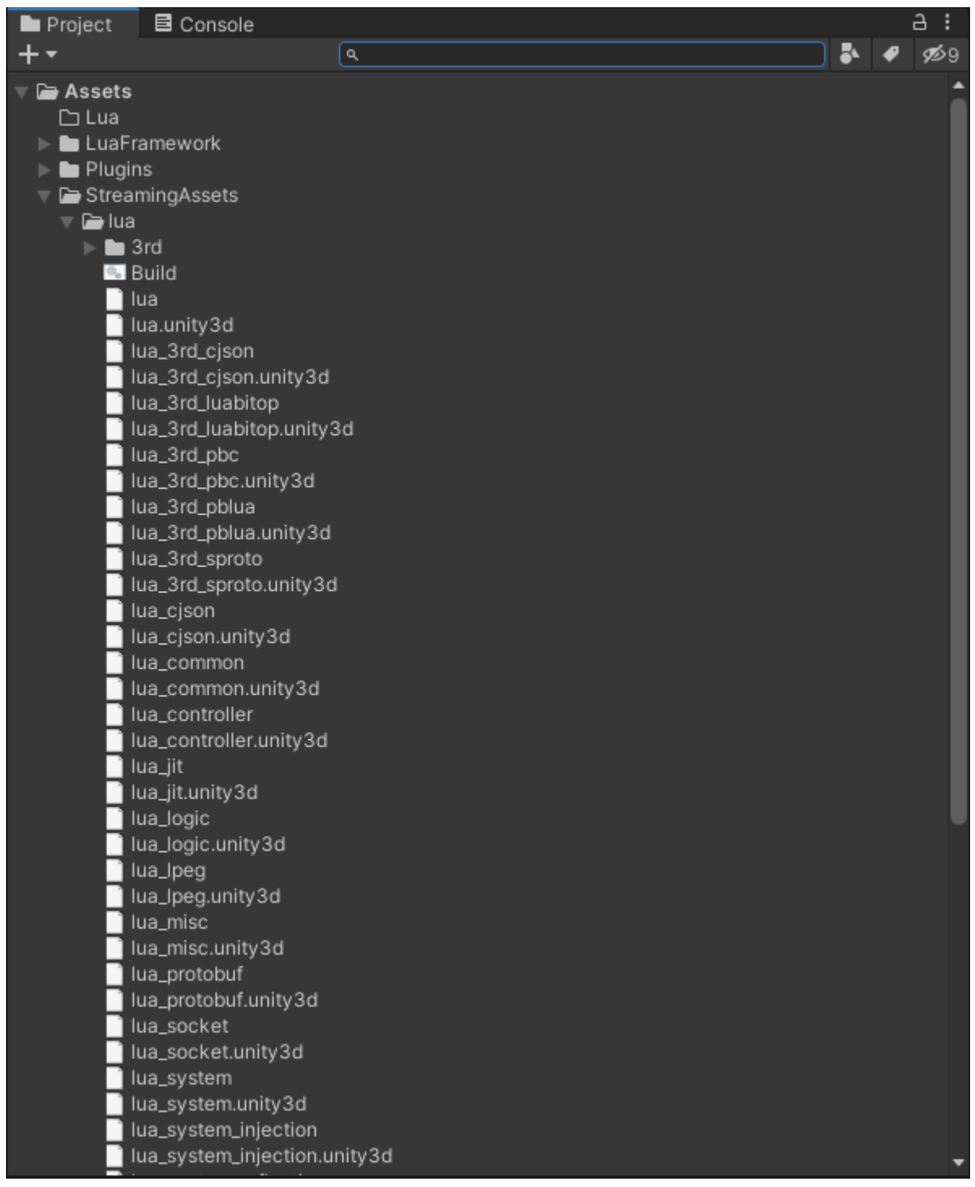
```
-- 假设r是Transform对象
print(t.gameObject)
```

解决办法就是不要在 `dropType` 数组中添加过滤类，而是在 `ToLuaExport.cs` 类的 `Generate` 方法中进行过滤，例:

```
// ToLuaExport.cs
public static void Generate(string dir)
{
    // ...
    if(type(Component) == type)
    {
        return;
    }
    // ...
}

```

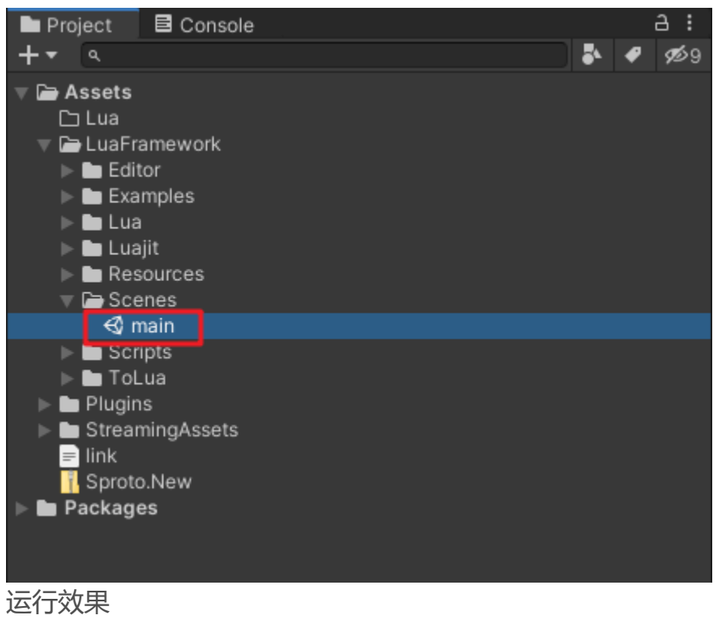
5、顺利生成AssetBundle

最后，【LuaFramework】-【Build Windows Resource】成功生成 AssetBundle，我们可以在 StreamingAssets 中看到很多 AssetBundle 文件。

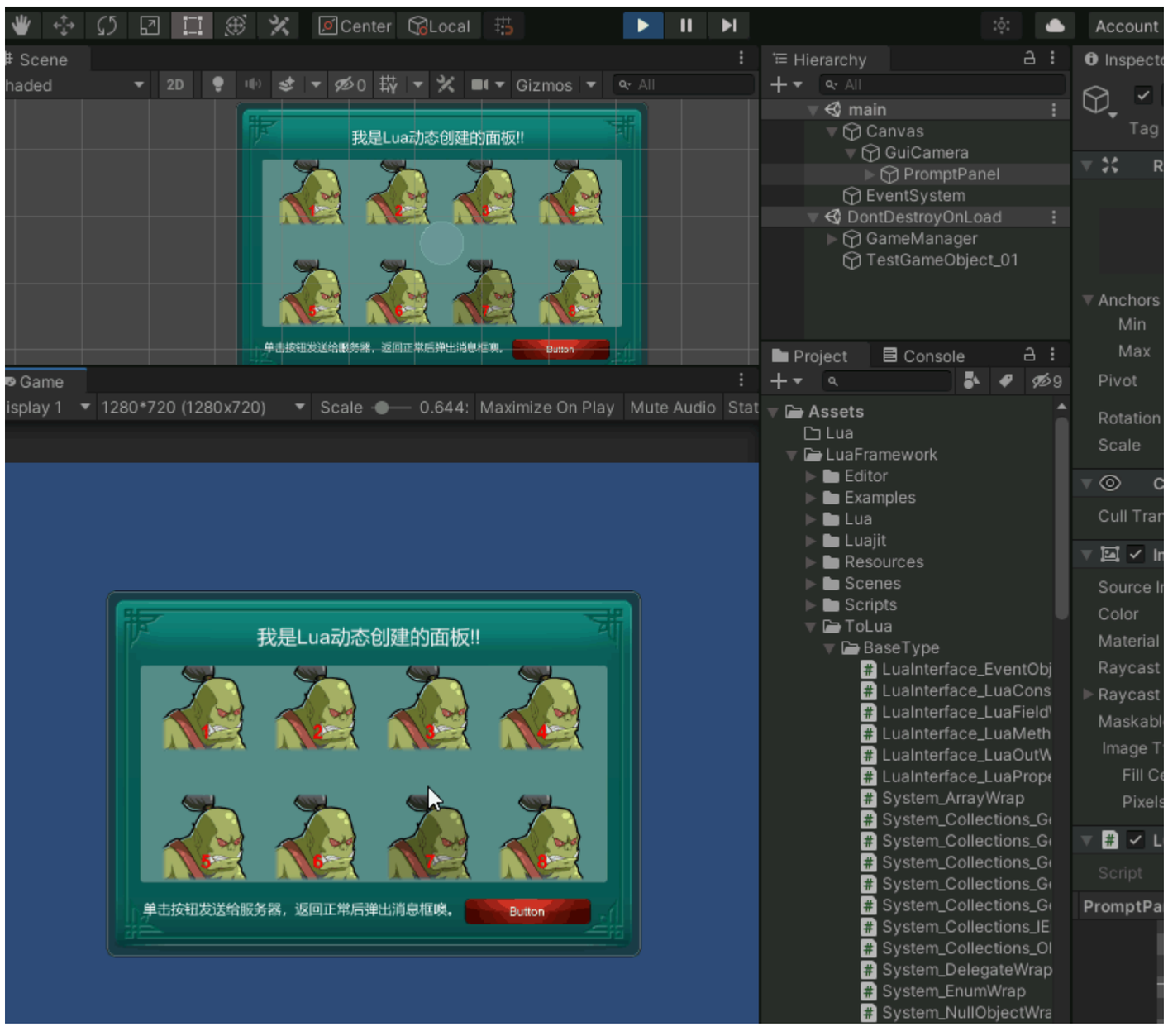


6、运行Demo场景

接下来，我们就可以运行 Demo 场景了。打开 main 场景



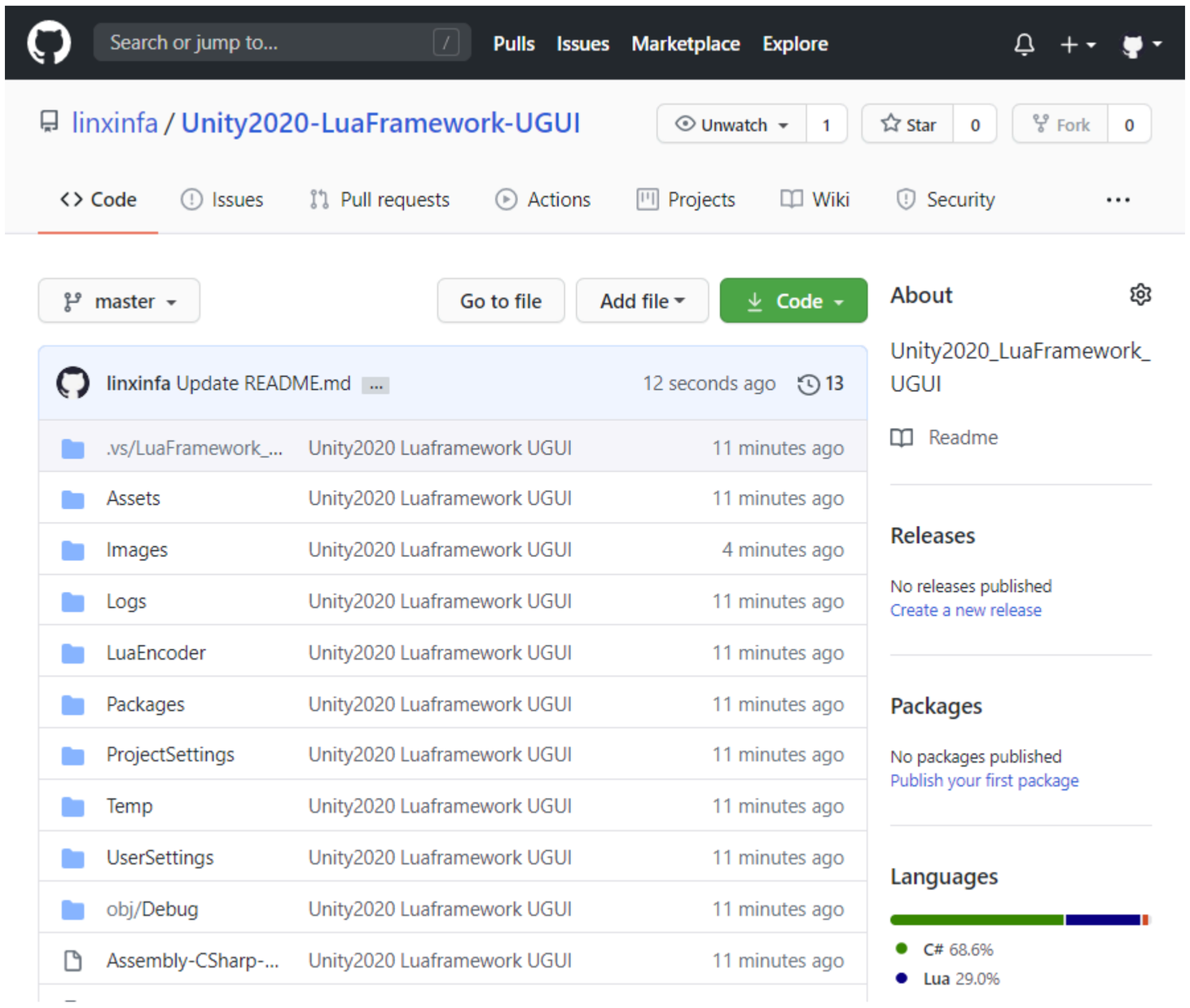
运行效果



7、Unity2020无报错版LuaFramework-UGUI

如果你不想手动修复上报的报错，我将修复好的版本上传到了 GitHub，使用 Unity2020 可以直接运行。

GitHub 工程地址：https://github.com/linxinfa/Unity2020-LuaFramework-UGUI



三、开发环境IDE

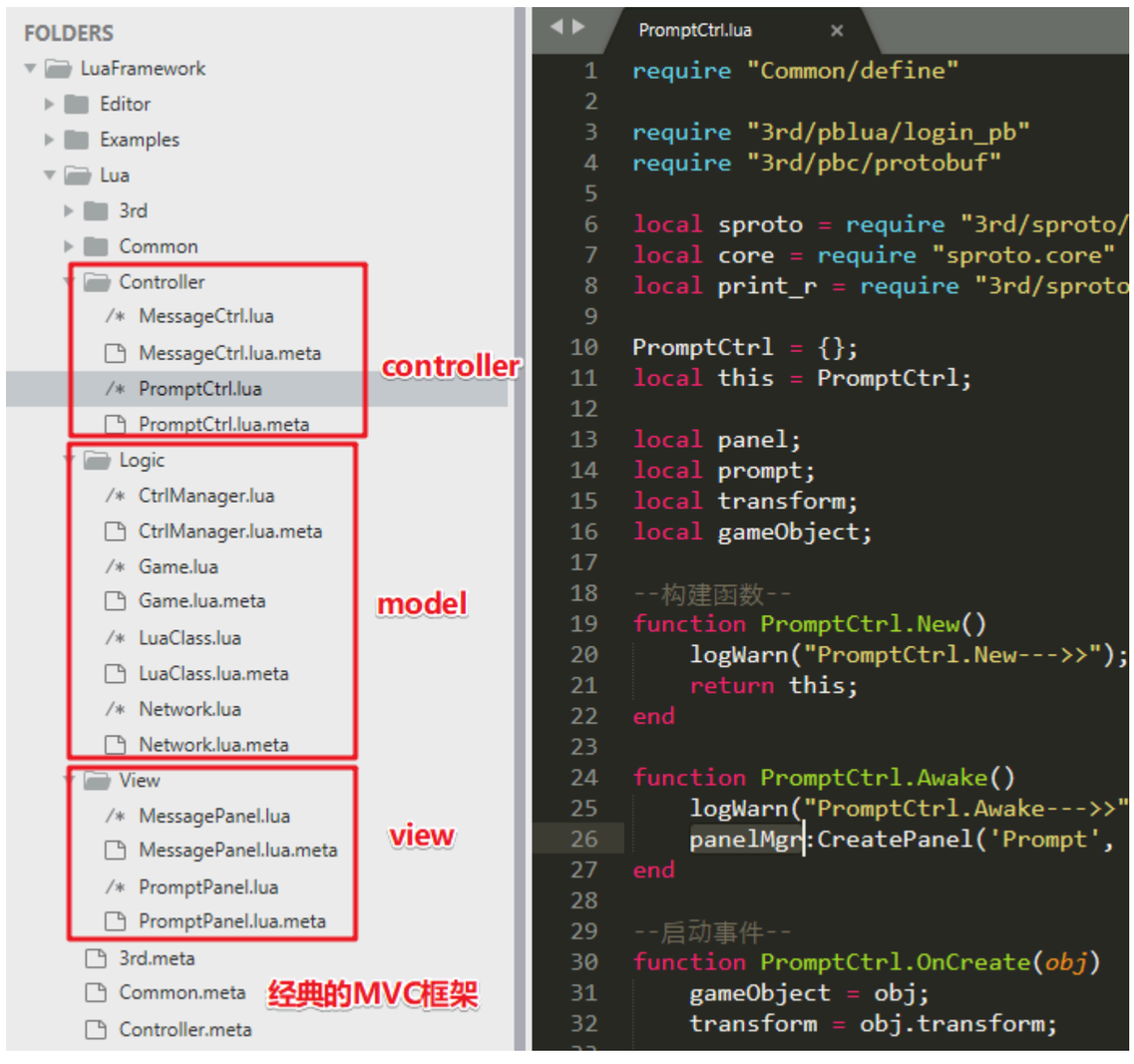
可以使用 `subline`， 也可以使用 `visual studio`， 个人偏好使用 `visual studio`， 配合插件 `BabeLua`

> Unity写lua代码的vs插件:BabeLua: https://blog.csdn.net/linxinfa/article/details/88191485

四、接口讲解

1、MVC框架

上面这个 `Lua` 动态创建出来的面板的控制逻辑在 `PromptCtrl.lua` 脚本中，我们可以看到 `lua` 工程中使用了经典的 `MVC` 框架。



> MVC全名是Model View Controller，是模型(model) - 视图(view) - 控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

所有的 `controler` 在 `CtrlManager` 中注册

```lua
-- CtrlManager.lua
function CtrlManager.Init()
    logWarn("CtrlManager.Init----->>>");
    ctrlList[CtrlNames.Prompt] = PromptCtrl.New();
    ctrlList[CtrlNames.Message] = MessageCtrl.New();
    return this;
end
```

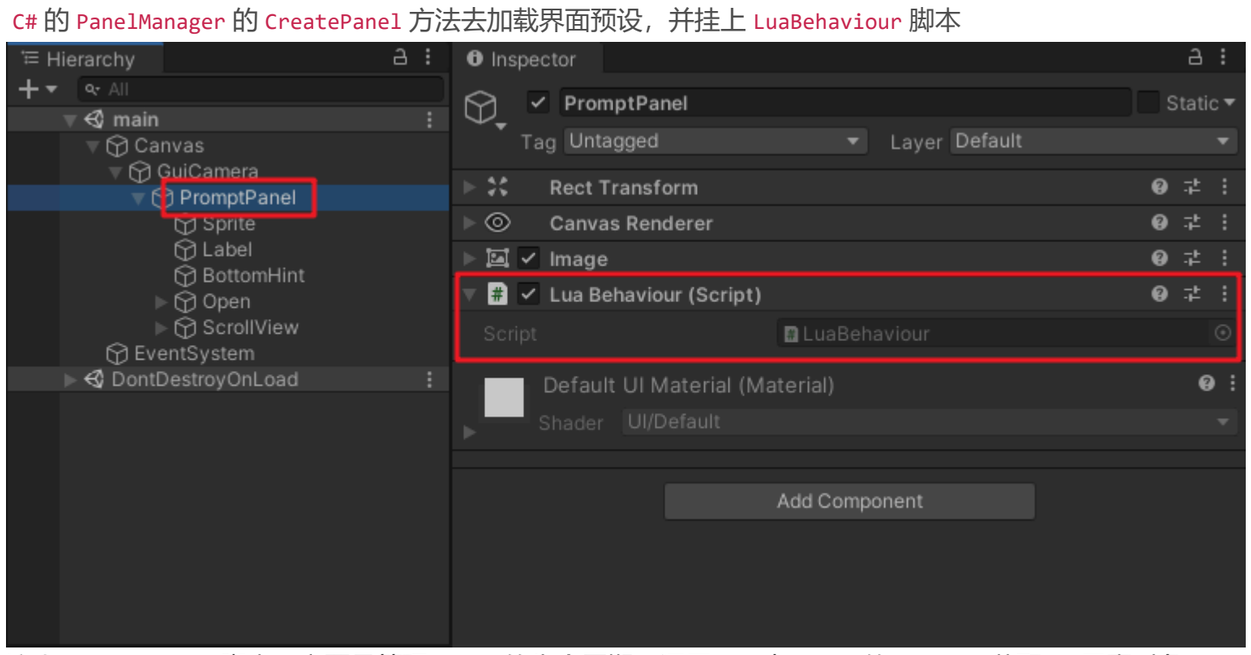
通过 `CtrlManager` 获取对应的 `controler` 对象，调用 `Awake()` 方法

```lua
-- CtrlManager.lua
local ctrl = CtrlManager.GetCtrl(CtrlNames.Prompt);
if ctrl ~= nil then
    ctrl:Awake();
end
```

`controler` 类中， `Awake()` 方法中调用 `C#` 的 `PanelManager` 的 `CreatePanel` 方法

```lua
-- PromptCtrl.lua
function PromptCtrl.Awake()
    logWarn("PromptCtrl.Awake--->>");
    panelMgr:CreatePanel('Prompt', this.OnCreate);
end
```

`C#` 的 `PanelManager` 的 `CreatePanel` 方法去加载界面预设，并挂上 `LuaBehaviour` 脚本



这个 `LuaBehaviour` 脚本，主要是管理 `panel` 的生命周期，调用 `lua` 中 `panel` 的 `Awake` ， 获取 `UI` 元素对象

```lua
-- PromptPanel.lua

local transform;
local gameObject;

PromptPanel = {};
local this = PromptPanel;

--启动事件--
function PromptPanel.Awake(obj)
    gameObject = obj;
    transform = obj.transform;

    this.InitPanel();
    logWarn("Awake lua--->>"..gameObject.name);
end

--初始化面板--
function PromptPanel.InitPanel()
    this.btnOpen = transform:Find("Open").gameObject;
    this.gridParent = transform:Find('ScrollView/Grid');
end

--单击事件--
function PromptPanel.OnDestroy()
    logWarn("OnDestroy--->>>");
end
```

`panel` 的 `Awake` 执行完毕后，就会执行 `controler` 的 `OnCreate()` ， 在 `controler` 中对 `UI` 元素对象添加一些事件和控制

```lua
-- PromptCtrl.lua
--启动事件--
function PromptCtrl.OnCreate(obj)
    gameObject = obj;
    transform = obj.transform;

    panel = transform:GetComponent('UIPanel');
    prompt = transform:GetComponent('LuaBehaviour');
    logWarn("Start lua--->>"..gameObject.name);

    prompt:AddClick(PromptPanel.btnOpen, this.OnClick);
```

```
    resMgr:LoadPrefab('prompt', { 'PromptItem' }, this.InitPanel);
end
```

2、StartUp启动框架

```
AppFacade.Instance.StartUp();   //启动游戏
```

这个接口会抛出一个NotiConst.START_UP事件，对应的响应类是StartUpCommand

```
using UnityEngine;
using System.Collections;
using LuaFramework;

public class StartUpCommand : ControllerCommand {

    public override void Execute(IMessage message) {
        if (!Util.CheckEnvironment()) return;

        GameObject gameMgr = GameObject.Find("GlobalGenerator");
        if (gameMgr != null) {
            AppView appView = gameMgr.AddComponent<AppView>();
        }
        //----------------关联命令----------------------
        AppFacade.Instance.RegisterCommand(NotiConst.DISPATCH_MESSAGE, typeof(SocketCommand));

        //----------------初始化管理器----------------------
        AppFacade.Instance.AddManager<LuaManager>(ManagerName.Lua);
        AppFacade.Instance.AddManager<PanelManager>(ManagerName.Panel);
        AppFacade.Instance.AddManager<SoundManager>(ManagerName.Sound);
        AppFacade.Instance.AddManager<TimerManager>(ManagerName.Timer);
        AppFacade.Instance.AddManager<NetworkManager>(ManagerName.Network);
        AppFacade.Instance.AddManager<ResourceManager>(ManagerName.Resource);
        AppFacade.Instance.AddManager<ThreadManager>(ManagerName.Thread);
        AppFacade.Instance.AddManager<ObjectPoolManager>(ManagerName.ObjectPool);
        AppFacade.Instance.AddManager<GameManager>(ManagerName.Game);
    }
}
```

这里初始化了各种管理器，我们可以根据具体需求进行改造和自定义。

3、LuaManager核心管理器

`LuaManager` 这个管理器是必须的，掌管整个 `lua` 虚拟机的生命周期。它主要是加载 `lua` 库，加载 `lua` 脚本，启动 `lua` 虚拟机，执行 `Main.lua` 。

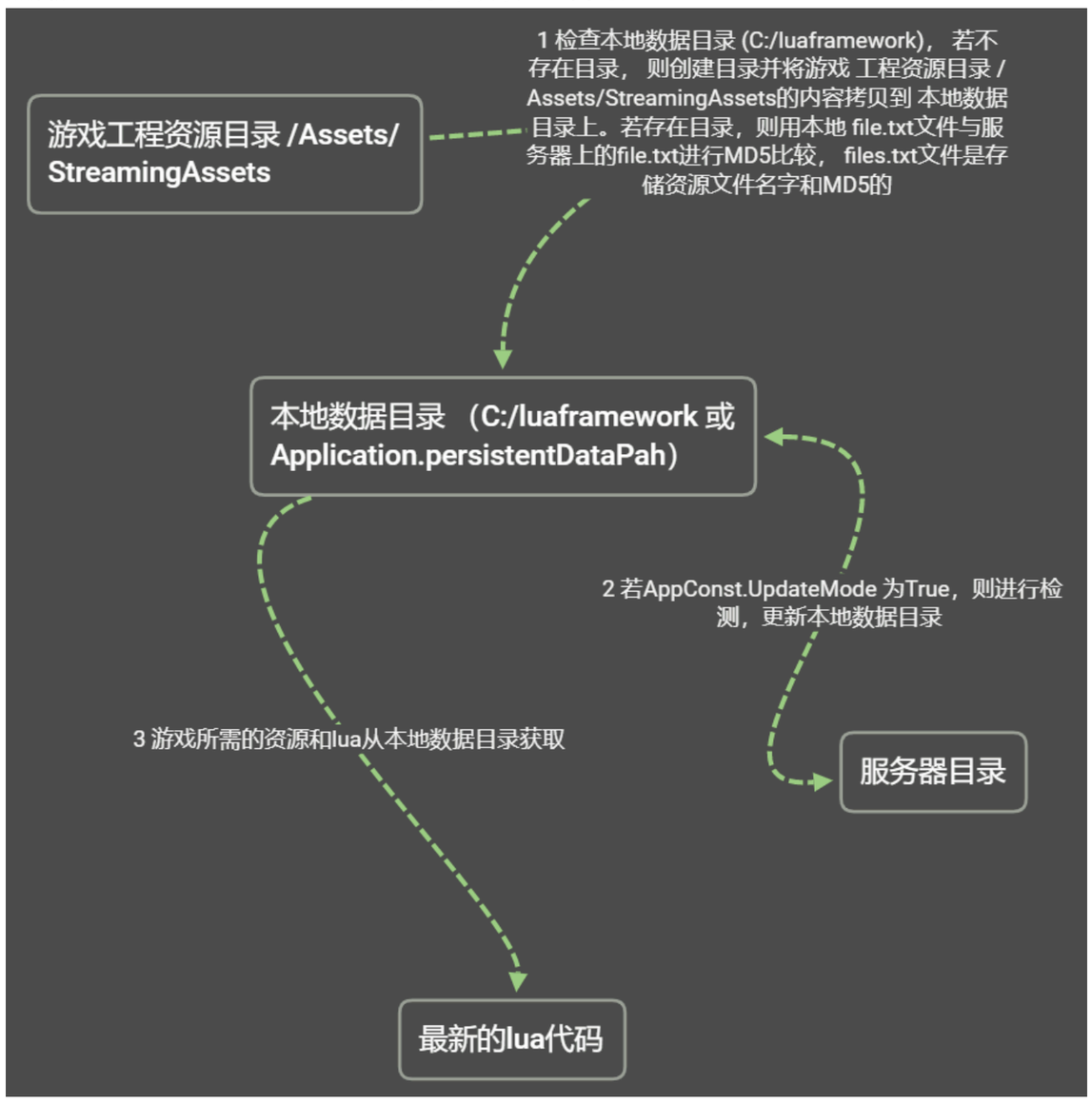
4、AppConst常量定义

`AppConst` 定义了一些常量。
其中AppConst.LuaBundleMode是 `lua` 代码 `AssetBundle` 模式。它会被赋值给LuaLoader的 `beZip` 变量，在加载 `lua` 代码的时候，会根据 `beZip` 的值去读取 `lua` 文件， `false` 则去 `search path` 中读取 `lua` 文件，否则从外部设置过来的 `bundle` 文件中读取 `lua` 文件。默认为 `true` 。在Editor环境下，建议把AppConst.LuaBundleMode设为false，这样方便运行，否则写完lua代码需要生成AssetBundle才可以运行到。

```
#if UNITY_EDITOR
        public const bool LuaBundleMode = false;                    //Lua代码AssetBundle模式
#else
        public const bool LuaBundleMode = true;                    //Lua代码AssetBundle模式
#endif
```

5、Lua代码的读取

LuaLoader和LuaResLoader都继承LuaFileUtils。 `lua` 代码会先从LuaFramework.Util.AppContentPath目录解压到LuaFramework.Util.DataPath目录中， `lua` 文件列表信息记录在 `files.txt` 中，此文件也会拷贝过去。然后从LuaFramework.Util.DataPath目录中读取 `lua` 代码。



```
/// LuaFramework.Util.DataPath

/// <summary>
/// 应用程序内容路径
/// AppConst.AssetDir = "StreamingAssets"
/// </summary>
public static string AppContentPath() {
    string path = string.Empty;
    switch (Application.platform) {
        case RuntimePlatform.Android:
            path = "jar:file://" + Application.dataPath + "!/assets/";
        break;
        case RuntimePlatform.IPhonePlayer:
            path = Application.dataPath + "/Raw/";
        break;
        default:
            path = Application.dataPath + "/" + AppConst.AssetDir + "/";
        break;
    }
    return path;
}

/// <summary>
/// 取得数据存放目录
/// </summary>
public static string DataPath {
    get {
        string game = AppConst.AppName.ToLower();
        if (Application.isMobilePlatform) {
            return Application.persistentDataPath + "/" + game + "/";
        }
        if (AppConst.DebugMode) {
            return Application.dataPath + "/" + AppConst.AssetDir + "/";
        }
        if (Application.platform == RuntimePlatform.OSXEditor) {
            int i = Application.dataPath.LastIndexOf('/');
            return Application.dataPath.Substring(0, i + 1) + game + "/";
        }
        return "c:/" + game + "/";
    }
}
```

完了之后，再进行远程的更新检测，看看用不用热更 `lua` 代码，远程 `url` 就是AppConst.WebUrl，先下载 `files.txt` ，然后再读取 `lua` 文件列表进行下载。

6、GameManager游戏管理器

启动框架后，会创建 `GameManager` 游戏管理器，它负责检测 `lua` 逻辑代码的更新检测和加载（ `Main.lua` 是在 `LuaManager` 中执行的），我们可以在 `GameManager` 中 `DoFile` 我们自定义的 `lua` 脚本，比如 `Game.lua` 脚本。

7、C#中如何直接调用lua的某个方法

`GameManager` 可以获取到 `LuaManager` 对象，通过LuaManager.CallFunction接口调用。
也可以用Util.CallMethod接口调用，两个接口的参数有差异，需要注意。

```
/// LuaManager.CallFunction接口
public object[] CallFunction(string funcName, params object[] args) {
      LuaFunction func = lua.GetFunction(funcName);
      if (func != null) {
          return func.LazyCall(args);
      }
      return null;
  }

/// Util.CallMethod接口
public static object[] CallMethod(string module, string func, params object[] args) {
    LuaManager luaMgr = AppFacade.Instance.GetManager<LuaManager>(ManagerName.Lua);
    if (luaMgr == null) return null;
    return luaMgr.CallFunction(module + "." + func, args);
}
```

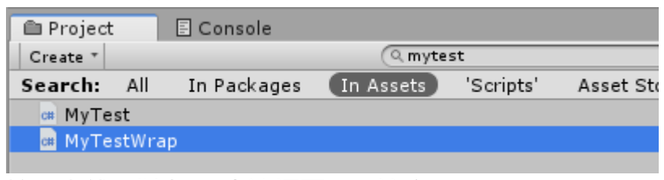
8、lua中如何调用C#的方法

假设现在我们有一个 `C#` 类

```
using UnityEngine;

public class MyTest : MonoBehaviour
{
    public int myNum;

    public void SayHello()
    {
        Debug.Log("Hello,I am MyTest,myNum: " + myNum);
    }

    public static void StaticFuncTest()
    {
        Debug.Log("I am StaticFuncTest");
    }
}
```

我们想在 `lua` 中访问这个 `MyTest` 类的函数。首先，我们需要在CustomSettings.cs中的 `customTypeList` 数组中添加类的注册：
`_GT(typeof(MyTest)),`

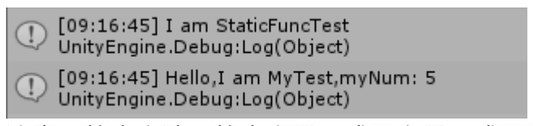
然后然后再单击菜单【Lua】-【Generate All】生成 Wrap，生成完我们会看到一个MyTestWrap类



接下来就可以在lua中访问了。（注意AppConst.LuaBundleMode的值要设为false，方便Editor环境下运行lua代码，否则需要先生成AssetBundle才能运行）

```lua
function Game.TestFunc()
    -- 静态方法访问
    MyTest.StaticFuncTest()

    local go = UnityEngine.GameObject("go")
    local myTest = go:AddComponent(typeof(MyTest))

    -- 成员变量
    myTest.myNum = 5
    -- 成员方法
    myTest:SayHello()
end
```

调用 Game.TestFunc()



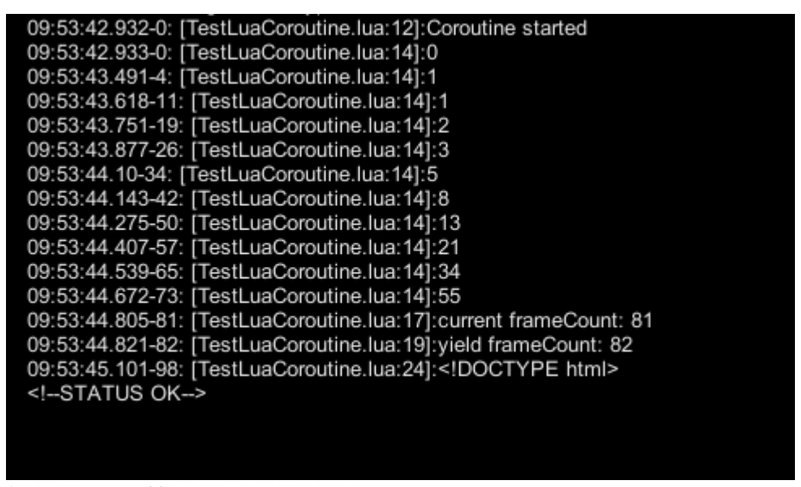
注意，静态方法、静态变量、成员变量、成员属性使用 “.” 来访问，比如上面的 myTest.myNum，成员函数使用 “:” 来访问，比如上面的 myTest:SayHello()

9、lua中如何使用协程

```lua
function fib(n)
    local a, b = 0, 1
    while n > 0 do
        a, b = b, a + b
        n = n - 1
    end

    return a
end

function CoFunc()
    print('Coroutine started')
    for i = 0, 10, 1 do
        print(fib(i))
        coroutine.wait(0.1)
    end
    print("current frameCount: "..Time.frameCount)
    coroutine.step()
    print("yield frameCount: "..Time.frameCount)

    local www = UnityEngine.WWW("http://www.baidu.com")
    coroutine.www(www)
    local s = tolua.tolstring(www.bytes)
    print(s:sub(1, 128))
    print('Coroutine ended')
end
```

调用

```lua
coroutine.start(CoFunc)
```



如果要 stop 协程，则需要这样

```lua
local co = coroutine.start(CoFunc)
coroutine.stop(co)
```

10、lua解析json

假设现在有这么一份json文件

```lua
{
    "glossary": {
        "title": "example glossary",
                "GlossDiv": {
            "title": "S",
                        "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                                        "SortAs": "SGML",
                                        "GlossTerm": "Standard Generalized Mark up Language",
                                        "Acronym": "SGML",
```

```
                                    "Abbrev": "ISO 8879:1986",
                                    "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                                            "GlossSeeAlso": ["GML", "XML"]
                    },
                                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

假设我们已经把上面的json文件的内容保存到变量jsonStr字符串中，现在在lua中要解析它

```lua
local json = require 'cjson'

function Test(str)
    local data = json.decode(str)
    print(data.glossary.title)
    s = json.encode(data)
    print(s)
end
```

调用 `Test(jsonStr)`

```
10:13:01.648-0: [LuaState.cs:6]:example glossary
10:13:01.649-0: [LuaState.cs:8]:{"glossary":{"title":"example
glossary","GlossDiv":{"title":"S","GlossList":{"GlossEntry":{"GlossDef":{"GlossSeeAlso":["GML","XML"]
,"para":"A meta-markup language, used to create markup languages such as
DocBook."},"Abbrev":"ISO 8879:1986","Acronym":"SGML","GlossTerm":"Standard Generalized Mark
up Language","ID":"SGML","SortAs":"SGML","GlossSee":"markup"}}}}}
```

11、lua调用C#的托管

```
// c#传托管给Lua
System.Action<string> cb = (s) => { Debug.Log(s); };
Util.CallMethod("Game", "TestCallBackFunc", cb);
```

```lua
-- lua调用C#的托管
function Game.TestCallBackFunc(cb)
    if nil ~= cb then
       System.Delegate.DynamicInvoke(cb,"Hello, I am lua, I call Delegate")
    end
end
```

```
[13:46:54] Hello, I am lua, I call Delegate
UnityEngine.Debug:Log(Object)
```

12、lua通过反射调用C#

有时候，我们没有把我们的 `C#` 类生成 `Wrap` ，但是又需要在 `lua` 中调用，这个时候，可以通过反射来调用。
假设我们有一个 `C#` 类： `MyClass`

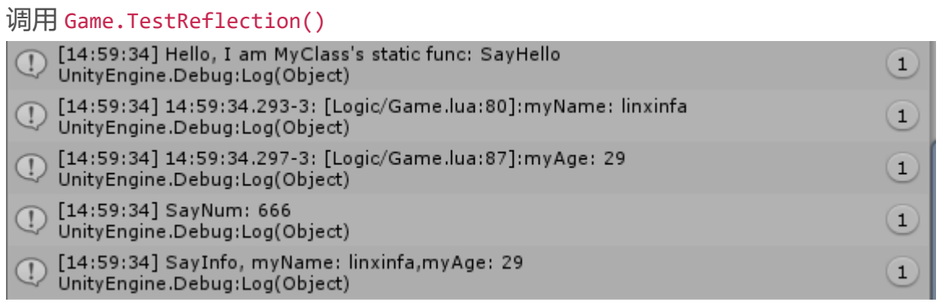
```
// MyClass.cs
public sealed class MyClass
{
    //字段
    public string myName;
    //属性
    public int myAge { get; set; }

    //静态方法
    public static void SayHello()
    {
        Debug.Log("Hello, I am MyClass's static func: SayHello");
    }

    public void SayNum(int n)
    {
        Debug.Log("SayNum: " + n);
    }

    public void SayInfo()
    {
        Debug.Log("SayInfo, myName: " + myName + ",myAge: " + myAge);
    }
}
```

在 `lua` 中

```lua
-- Game.lua
function Game.TestReflection()
    require 'tolua.reflection'
    tolua.loadassembly('Assembly-CSharp')
    local BindingFlags = require 'System.Reflection.BindingFlags'

    local t = typeof('MyClass')
    -- 调用静态方法
    local func = tolua.getmethod(t, 'SayHello')

```

```
    func:Call()
    func:Destroy()
    func = nil

    -- 实例化
    local obj = tolua.createinstance(t)
    -- 字段
    local field = tolua.getfield(t, 'myName')
    -- 字段Set
    field:Set(obj, "linxinfa")
    -- 字段Get
    print('myName: ' .. field:Get(obj))
    field:Destroy()

    -- 属性
    local property = tolua.getproperty(t, 'myAge')
    -- 属性Set
    property:Set(obj, 29, null)
    -- 属性Get
    print('myAge: ' .. property:Get(obj, null))
    property:Destroy()

    --public成员方法SayNum
    func = tolua.getmethod(t, 'SayNum', typeof('System.Int32'))
    func:Call(obj, 666)
    func:Destroy()

    --public成员方法SayInfo
    func = tolua.getmethod(t, 'SayInfo')
    func:Call(obj)
    func:Destroy()
end
```

调用 `Game.TestReflection()`



13、nil和null

`nil` 是 `lua` 对象的空， `null` 表示 `c#` 对象的空。假设我们在 `c#` 中有一个 `GameObject` 对象传递给了 `lua` 的对象 `a` ， 接下来我们把这个 `GameObject` 对象 `Destroy` 了，并在 `c#` 中把这个 `GameObject` 对象赋值为 `null` ， 此时 `lua` 中的对象 `a` 并不会等于 `nil`

如果要在 `lua` 中判断一个对象是否为空，安全的做法是同时判断 `nil` 和 `null`

```
-- lua中对象判空
function IsNilOrNull(o)
    return nil == o or null == o
end
```

14、获取今天是星期几

```
-- 1是周日，2是周一，以此类推
function GetTodayWeek()
    local t = os.date("*t", math.floor(os.time()))
    return t.wday
end
```

15、获取今天的年月日

方法一

```
function GetTodayYMD()
    local t = os.date("*t", math.floor(os.time()))
    return t.year .. "/" .. t.month .. "/" .. t.day
end
```

方法二

```
function GetTodayYMD()
    -- 如果要显示时分秒，则用"%H:%M:%S"
    return os.date("%Y/%m%d", math.floor(os.time()))
end
```

16、字符串分割

```
-- 参数str是你的字符串，比如"小明|小红|小刚"
-- 参数sep是分隔符，比如"|"
-- 返回值为{"小明","小红","小刚"}
function SplitString(str, sep)
    local sep = sep or " "
    local result = {}
    local pattern = string.format("([^%s]+)", sep)
    string.gsub(s, pattern, function(c) result[#result + 1] = c end)
    return result
end
```

17、大数字加逗号分割（数字会转成字符串）

```lua
-- 参数num是数字，如3428439，转换结果"3,428,439"
function FormatNumStrWithComma(num)
    local numstr = tostring(num)
    local strlen = string.len(numstr)
    local splitStrArr = {}
    for i = strlen, 1, -3 do
        local beginIndex = (i - 2 >= 1) and (i - 2) or 1
        table.insert(splitStrArr, string.sub(numstr, beginIndex, i))
    end
    local cnt = #splitStrArr
    local result = ""
    for i = cnt, 1, -1 do
        if i == cnt then
            result = result .. splitStrArr[i]
        else
            result = result .. "," .. splitStrArr[i]
        end
    end
    return result
end
```

18、通过组件名字添加组件

```lua
-- 缓存
local name2Type = {}
-- 参数gameObject物体对象
-- 参数componentName，组件名字，字符串
function AddComponent(gameObject, componentName)
    local component = gameObject:GetComponent(componentName)
    if nil ~= component then return component end

    local componentType = name2Type[componentName]
    if nil == componentType then
        componentType  = System.Type.GetType(componentName)
        if nil == componentType then
            print("AddComponent Error: " .. componentName)
            return nil
        else
            name2Type[componentName] = componentType
        end
    end
    return gameObject:AddComponent(componentType)
end
```

19、深拷贝

lua中的table是引用类型，有时候我们为了不破坏原有的table，可能要用到深拷贝

```lua
function DeepCopy(t)
    if nil == t then return nil end
    local result = ()
    for k, v in pairs(t) do
        if "table" == type(v) then
            result[k] = DeepCopy(v)
        else
            result[k] = v
        end
    end
    return result
end
```

20、四舍五入

```lua
function Round(fnum)
    return math.floor(fnum + 0.5)
end
```

21、检测字符串是否含有中文

```lua
-- 需要把C#的System.Text.RegularExpressions.Regex生成Wrap类
function CheckIfStrContainChinese(str)
    return System.Text.RegularExpressions.Regex.IsMatch(str, "[\\u4e00-\\u9fa5]")
end
```

22、数字的位操作get、set

```lua
-- 通过索引获取数字的某一位，index从1开始
function GetBitByIndex(num, index)
    if nil == index then
        print("LuaUtil.GetBitByIndex Error, nil == index")
        return 0
    end
    local b = bit32.lshift(1,(index - 1))
    if nil == b then
        print("LuaUtil.GetBitByIndex Error, nil == b")
        return 0
    end
    return bit32.band(num, b)
end

```

```lua
-- 设置数字的某个位为某个值，num：目标数字，index：第几位，从1开始，v：要设置成的值，0或1
function SetBitByIndex(num, index, v)
    local b = bit32.lshift(1,(index - 1))
    if v > 0 then
        num = bit32.bor(num, b)
    else
        b = bit32.bnot(b)
        num = bit32.band(num, b)
    end
    return num
end
```

23、限制字符长度，超过进行截断

有时候，字符串过长需要截断显示，比如有一个昵称叫“我的名字特别长一行显示不下”，需求上限制最多显示5个字，超过的部分以…替代，即"我的名字特…"。首先要计算含有中文的字符串长度，然后再进行截断

```lua
-- 含有中文的字符串长度
function StrRealLen(str)
    if str == nil then return 0 end
    local count = 0
    local i = 1
    while (i < #str) do
        local curByte = string.byte(str, i)
        local byteCount = 1
        if curByte >= 0 and curByte <= 127 then
            byteCount = 1
        elseif curByte >= 192 and curByte <= 223 then
            byteCount = 2
        elseif curByte >= 224 and curByte <= 239 then
            byteCount = 3
        elseif curByte >= 240 and curByte <= 247 then
            byteCount = 4
        end
        local char = string.sub(str, i, i + byteCount - 1)
        i = i + byteCount
        count = count + 1
    end
    return count
end

-- 限制字符长度（多少个字）
-- 参数str，为字符串
-- 参数limit为限制的字数，如8
-- 参数extra为当超过字数时，在尾部显示的字符串，比如"..."
function LimitedStr(str, limit, extra)
    limit = limit or 8
    extra = extra or ""
    local text = ""
    -- 含有中文的字符串长度
    if StrRealLen(str) > limit then
        text = LuaUtil.sub_chars(str, limit) .. "..." .. extra
    else
        text = str .. extra
    end
    return text
end
```

24、判断字符串A是否已某个字符串B开头

```lua
-- 判断字符串str是否是以某个字符串start开头
function StringStartsWith(str, start)
    return string.sub(str, 1, string.len(start)) == start
end
```

五、热更lua与资源

1、热更lua

打 `app` 整包的时候，备份一份 `lua` 全量文件，后面打 `lua` 增量包的时候，根据文件差异进行比对，新增和差异的lua文件打成一个 `lua_update.bundle` ，放在一个 `update` 文件夹中，并压缩成 `zip` ，放到服务器端，客户端通过 `https` 下载增量包并解压到 `Application.persistentDataPath` 目录。游戏加载 `lua` 文件的时候，优先从 `update` 文件夹中的 `lua_update.bundle` 中查找 `lua` 脚本。

2、热更资源热更资源

做个编辑器工具，指定某个或某些资源文件（预设、音频、动画、材质等），打成多个 `assetbundle` ，放在一个 `update` 文件夹中，并压缩成一个 `zip` ，放到服务器端，客户端通过 `https` 下载增量包并解压到 `Application.persistentDataPath` 目录。
游戏加载资源文件的时候，优先从 `update` 文件夹中查找对应的资源文件。

3、真机热更资源存放路径

```
persistentDataPath/res/
                    ├──/update/
                    │       ├──/lua/
                    │       │    └──lua_update.bundle           #lua增量bundle
                    │       ├──/res/
                    │       │    ├──aaa.bundle                  #预设aaa的bundle
                    │       │    ├──bbb.bundle                  #音频bbb的bundle
                    │       │    └──...                         #其他各种格式的资源bundle
                    │       └──/cfg/
                    │            ├──cfg.bundle                 #配置增量bundle
                    │            └──...                        #其他文本或二进制文件增量bundle
```

```
├──out_put.log                    #游戏日志
└──...
```

关于 `persistentDataPath`，可以参见我这篇博客：https://blog.csdn.net/linxinfa/article/details/51679528