Unity网络以及AssetBundle

一：网络下载WWW类的使用

二：AssetBundle创建以及使用

三：Asset资源动态加载

四：Lua热更新实现

一：网络下载WWW类的使用

WWW类是Unity网络开发中使用频率非常高的一个工具类，主要提供一般HTTP访问的功能，以及从网络上动态的下载图片，声音，视频资源等等。

目前WWW支持的协议有http://,https://,file://,ftp://,其中FIle是访问本地文件。

下面我们通过一个实例来讲解WWW的使用，学习具体的使用方式。

首先，新建一个场景，拖入一个立方体和一个球体，我们下面通过下载的资源修改立方体的材质，具体代码如下所示：

```
public class WWWDemo1 : MonoBehaviour {
  public GameObject goCube;
  public GameObject goSphere;

  private Texture2D DownloadTexture;

  public void DownloadTexturesByWWW(){
       StartCoroutine ("StartDownloadTexture");
  }

  public void DownloadTexturesFromHttp(){
       StartCoroutine ("StartDownloadTextureFromHTTP");
  }
  // Use this for initialization
  void Start () {
       DownloadTexturesByWWW ();
       DownloadTexturesFromHttp ();
  }

  // Update is called once per frame
  void Update () {

  }

  IEnumerator StartDownloadTexture(){
       WWW loadloadTexture = new WWW ("file:////Users/a/
Desktop/sand.png");
       yield return loadloadTexture;
```

```
        goCube.GetComponent<Renderer>
().material.mainTexture = loadloadTexture.texture;
  }

  IEnumerator StartDownloadTextureFromHTTP(){
        WWW loadloadTexture = new WWW ("http://
image.tianjimedia.com/uploadImages/2015/129/56/
J63MI042Z4P8_1000x500.jpg");
        yield return loadloadTexture;
        goSphere.GetComponent<Renderer>
().material.mainTexture = loadloadTexture.texture;

  }


}
```

其中，一个是下载本地资源，一个是下载网络资源。为了看起来更加直观，可以在界面添加两个按钮，分别是下载本地资源和网络资源，可以看到，贴图发生了变化。

二：AssetBundle创建以及使用

2.1 AssetBundle介绍

AssetBundle是将资源使用Unity提供的一种用于存储资源的压缩格式打包后的集合，它可以存储任何一种Unity可以识别的资源，如模型，纹理图，音频，场景等资源。也可以加载开发者自定义的二进制文件。他们的文件类型是.assetbundle/.unity3d,他们先前被设计好，很容易就下载到我们的游戏或者场景当中。

一般情况下AssetBundle的具体开发流程如下：

（1）创建Asset bundle，开发者在unity编辑器中通过脚本将所需要的资源打包成AssetBundle文件

（2）上传服务器。开发者将打包好的AssetBundle文件上传至服务器中。使得游戏客户端能够获取当前的资源，进行游戏的更新

（3）下载AssetBundle，首先将其下载到本地设备中，然后再通过AsstBudle的加载模块将资源加到游戏之中。

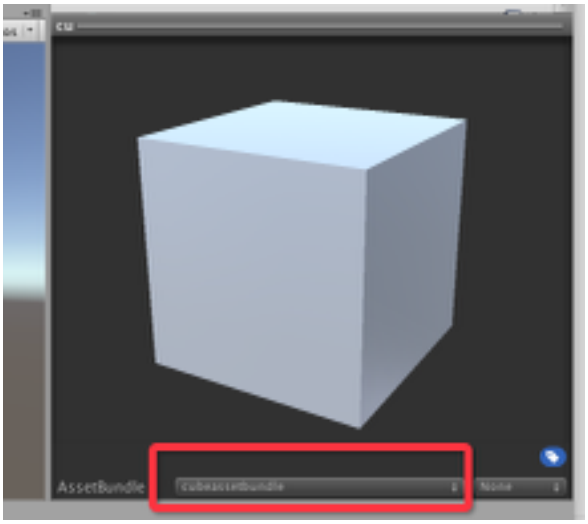
（4）加载，通过Unity提供的API可以加载资源里面包含的模型、纹理图、音频、动画、场景等来更新游戏客户端

（5）卸载AssetBundle，卸载之后可以节省内存资源，并且要保证资源的正常更新。

2.2 创建AssetBundle

（1）只有在Asset窗口中的资源才可以打包，我们单击GameObject->CUbe,然后在Asset窗口创建一个预设体，命名为cubeasset,讲Cube拖到该预设体上。

（2）单击刚创建的预制件，按照课上的步骤创建即可。



2.3 打包AssetBundle

AssetBundle创建之后需要导出，这一个过程就需要编写相应的代码实现，从Unity5.x之后，提供了一套全新简单的API来实现打包功能。大大简化了开发者手动遍历资源自行打包的过程，更加方便快捷。具体实现代码如下：

```
using UnityEngine;
using System.Collections;
using UnityEditor;
public class ExportAsset : MonoBehaviour {
  [@MenuItem("Test/BuildAssetBundles")]

  static void BuildAssetBundles(){
       BuildPipeline.BuildAssetBundles ("Assets/
AssetBundles");
  }
  // Use this for initialization
  void Start () {

  }

  // Update is called once per frame
  void Update () {

  }
}
```

此脚本需要创建在Asset的Editor目录下，编写完成之后不需要挂载，会直接在Unity的集成环境之中生成Test项，单击即可进行打包。

其中，AssetBundles文件夹需要我们手动创建。

所有的AssetBundle已经被导出，此时每一个AssetBundle资源会有一个和文件相关的Mainfest 的文本类型的文件，该文件提供了所打包资源的CRC和资源依赖的信息。

除此之外，还有一个AssetBundle文件会在生成的时候被创建，记录者整个资源列表以及列表之间的关系。

2.4 资源动态加载AssetBundle技术

我们采用下面的案例，场景进入的时候，依次从服务器给三个已经创建的对象加载纹理，材质，以及根据预设创建一个新的对象。代码如下所示：

```
using UnityEngine;
using System.Collections;

public class DownloadAssetBundles : MonoBehaviour {
	public GameObject go_ChangeMat;
	public GameObject go_ChangeTxt;
	public Transform clonePosition;
	private string url1="http://127.0.0.1:8080/good4?dev=ios";
	private string url2="http://127.0.0.1:8080/mat_cube?dev=ios";
	// Use this for initialization
	private string url3="http://127.0.0.1:8080/prerotate?
dev=ios";
	void Start () {
		DownMat ();
		DownPrefab ();
		DownTexture ();
	}

	// Update is called once per frame
	void Update () {

	}

	public void DownMat(){
		StartCoroutine (Download_Mat (url2));
	}
	public void DownTexture(){
		StartCoroutine (Download_Texture (url1));
	}
	public void DownPrefab(){
		StartCoroutine (Download_Prefab (url3));
	}
	IEnumerator Download_Mat(string path){
		WWW downloadAsset = new WWW (path);
		yield return downloadAsset;

go_ChangeMat.GetComponent<Renderer>().material=(Material)downloadA
sset.assetBundle.mainAsset;
		downloadAsset.assetBundle.Unload(false);
	}
	IEnumerator Download_Texture(string path){
		WWW downloadAsset = new WWW (path);
```

```
            yield return downloadAsset;

go_ChangeTxt.GetComponent<Renderer>().material.mainTexture=(Textur
e)downloadAsset.assetBundle.LoadAsset("good4");
            downloadAsset.assetBundle.Unload(false);
        }
        IEnumerator Download_Prefab(string path){
            WWW downloadAsset = new WWW (path);
            yield return downloadAsset;
            GameObject gpPrefabs = (GameObject)Instantiate
(downloadAsset.assetBundle.LoadAsset("Pre_SelfRotate"));
            gpPrefabs.transform.position =
clonePosition.transform.position;

            downloadAsset.assetBundle.Unload (false);
        }



}
```

四：Lua热更新实现

任何一款手游上线之后，我们都需要进行游戏Bug的修复或者是在遇到节日时发布一些活动，这些都需要进行游戏的更新，而且通常都会涉及到代码的更新。什么是游戏的热更新，下面我们依次介绍。

热更细是指用户直接重启客户端就能实现的客户端资源代码更新操作。游戏热更新会减少游戏中的打包次数，提升程序调试效率，游戏运营时候减少大版本更新次数，可以有效防止用户流失。

lua语言在热更新中会被广泛使用，我们这里采取的是ulua，ulua是unity+lua+cstolua组成的，开发者已经为我们封装成SimpleFramework_UGUI和SimpleFramework_NGUI框架，让开发者在使用的时候更加快捷方便。

4.1 框架介绍以及具体使用

下面我们对游戏每个文件夹进行大致的介绍，介绍如下：

- Examples:框架自带的Demo例子
- Lua:框架自带的Lua源码目录，用户自定义的Lua脚本就是放在这里的，最后打包的时候，打包脚本会按照目录结构生成到StreamingAssets,然后将其放置到Web服务器，用于被每个游戏客户端下载更新他们的LUa脚本，达到热更新的目的。
- Plugins：ulua底层库所在的目录，里面存放的是不同底层的底层库
- Scripts：框架的C#脚本层

- StreamingAssets：框架打包资源的文件夹，讲所有的资源打包进文件夹

文件夹介绍完毕之后，我们就可以按照Demo中的提示来运行我们的热更新案例了。

（1）打开案例之前，框架会提示一个弹窗。在这个弹窗里会介绍框架的使用步骤。单击Game->Build xxx Resources，这个命令的作用是根据不同的平台生成不同的assetbundle资源。

（2）单击Lua菜单中的GenLuaWrapFile子菜单，框架就会wrap文件。这个菜单的作用是将unity中的类文件生成wrap供Lua调用。当然还有清除wrap文件的选项。

（3）我们当添加我们自己的类文件之后，清空一下文件重新生成。可以在ulua/Source/LuaWrap/Wrap文件夹中看到自己添加的类文件。

（4）打开Examples例子中的Scene下的login，运行时候，会提示一个弹窗，包含运行步骤。可以看到此时我们没有打开服务器，更新失败。

（5）将生成的StreamingAssets放到服务器的根目录下，启动服务器。

（6）再次运行该游戏，单击Button按钮就能弹出一个新的界面，点击即可关闭弹出的界面。这就是框架中提供的例子。

4.2 热更新具体案例

通过前面的介绍，大家会对热更细有一个简单的了解。接下来我们将通过UGUI来制作并且实现UI界面的热更新。让读者明白如何使用框架来生成自己的UI界面并且通过Web服务器更新界面。

（1）新建项目，导入SimpleFramework_UGUI，删除自带的Examples文件夹。

(2)在Asset目录下新建一个文件夹并且命名为Demo，这个文件夹用来存放我们的资源文件，例如声音图片等资源。

（3）新建一个场景，命名为MyDemo，然后在Demo文件夹下新建Demo,Demo2,Texture文件夹用来存放资源文件。

（4）在场景中将摄像机命名为GuiCamera，并且设置其tag值为GuiCamera，依次单击GameObject-UI-Canvas 创建画布并且将摄像机设置为其子物体。通过GameObject->CreateEmpty创建一个空对象，重命名为GlobalGenerator,并为其挂载GlobalGenerator脚本。

（5）接下来使用UGUI搭建UI界面，将面板分别命名为DemoPanel和Demo2Panel,创建的Panel必须以＊＊＊Panel进行命名。本例中有两个界面，通过第一个界面的开始按钮打开第二个界面，通过另一个界面的关闭按钮可以关闭第二个界面。完成之后将两个Panel制作成预设体放置在Demo,Demo2两个文件夹下。

（6）将两个预设体制作成为AssetBundle，这样在后面打包的时候才能生成。制作完之后，讲场景中的界面删除。

（7）接下来编写代码，Lua代码编写之后存放在Lua文件夹下的Controller,View下。

（8）首先编写View文件夹下的DemoPanel.lua脚本，该脚本用于获取主界面的打开按钮控件，添加其相关变量。脚本内容如下：

```
local transform;
local gameObject;
DemoPanel={};
local this=DemoPanel;
function DemoPanel.Awake(obj)
  gameObject=obj;
  transform=obj.transform;
  this.InitPanel();

  warn("Awake la--->"..gameObject.name);

end

function DemoPanel.InitPanel()
  this.demoopen=transform:FindChild("Button").gameObject;

this.democlose=transform:FindChild("Button1").gameObject;
end

function DemoPanel.OnDestroy()
  warn("OnDestroy");
end
```

（9）View文件夹下的另外一个界面的内容与该脚本内容大体类似。如下

```
local transform;
local gameObject;

Demo1Panel = {};
local this = Demo1Panel;

function Demo1Panel.Awake(obj)
  gameObject = obj;
  transform = obj.transform;

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

function Demo1Panel.InitPanel()
  this.demoClose =
transform:FindChild("Button").gameObject;
end

function Demo1Panel.OnDestroy()
```

```
  warn("OnDestroy---->>>");
end
```

（10）接下来我们编写对应的DemoCtrl，该界面主要负责一些相关功能的实现，lua文件为DemoCtrl,内容如下：

```
require "Common/define"

DemoCtrl={};
local this=DemoCtrl;
local panel;
local gameObject;
local transform;
local demo;
function DemoCtrl.New()
  return this;
end

function DemoCtrl.Awake()
  warn("DemoCtrl.New");
  PanelManager:CreatePanel("Demo",this.OnCreate);
end

function DemoCtrl.OnCreate(obj)
  gameObject=obj;
  transform=obj.transform;

  panel=transform:GetComponent("UIPanel");
  demo=transform:GetComponent("LuaBehaviour");
  warn("Start Lua"..gameObject.name);

  demo:AddClick(DemoPanel.demoopen,this.OnClick);
  demo:AddClick(DemoPanel.democlose,this.OnClick2);


end

function DemoCtrl.OnClick(go)
 PanelManager:CreatePanel("Demo1",Demo1Ctrl.OnCreate);
  --热更新更换场景，也可以更换具体的逻辑
  --Application.LoadLevel(“TestNewScene”);
end

function DemoCtrl.OnClick2(go)
  print("Demo2Clicked");
end
```

(11)另外一个界面的控制器脚本和此界面大体类似，内容如下：

```
require "Common/define"
```

```
Demo1Ctrl = {};
local this = Demo1Ctrl;

local panel;
local demo2;
local transform;
local gameObject;

function Demo1Ctrl.New()
  warn("Demo2Ctrl.New--->>");
  return this;
end

function Demo1Ctrl.Awake()
  warn("Demo2Ctrl.Awake--->>");
  PanelManager:CreatePanel('Demo1', this.OnCreate);
end

function Demo1Ctrl.OnCreate(obj)
  gameObject = obj;
  transform = obj.transform;

  panel = transform:GetComponent('UIPanel');
  demo2 = transform:GetComponent('LuaBehaviour');
  warn("Start lua--->>"..gameObject.name);

  demo2:AddClick(Demo1Panel.demoClose, this.OnClick);
end

function Demo1Ctrl.OnClick(go)
  destroy(gameObject);
end

function Demo1Ctrl.Close()
  PanelManager:ClosePanel(CtrlName.Demo);
end
```

（12）脚本编写完成之后，需要对框架内的脚本进行修改使得框架能够加载对应的脚本，打开Asset/Lua/Common/define.lua,在CtrlName中添加刚才编写的控制器脚本。

```
CtrlName = {
    --Prompt = "PromptCtrl",
    --Message = "MessageCtrl",
    Demo = "DemoCtrl",
    Demo1 = "Demo1Ctrl"
}
```

```
require "Common/define"
--require "Controller/PromptCtrl"
--require "Controller/MessageCtrl"
require "Controller/DemoCtrl"
require "Controller/Demo1Ctrl"

CtrlManager = {};
local this = CtrlManager;
local ctrlList = {};    --控制器列表--

function CtrlManager.Init()
    warn("CtrlManager.Init----->>>");
    --ctrlList[CtrlName.Prompt] = PromptCtrl.New();
    --ctrlList[CtrlName.Message] = MessageCtrl.New();
    ctrlList[CtrlName.Demo] = DemoCtrl.New();
    ctrlList[CtrlName.Demo1] = Demo1Ctrl.New();
    return this;
end
```

打开Lua / Logic / CtrlManager脚本修改下面的代码

(13) 接下来打开Lua/Logic/GameManager.lua脚本，并且修改代码，否则显示的还是系统提供的Demo

```
require "Logic/LuaClass"
require "Logic/CtrlManager"
require "Common/functions"
--require "Controller/PromptCtrl"
require "Controller/DemoCtrl"
require "Controller/Demo1Ctrl"

--管理器--
GameManager = {};
local this = GameManager;

local game;
local transform;
local gameObject;

function GameManager.LuaScriptPanel()
    return 'Demo','Demo1';
end
```

修改默认初始化界面

```lua
--初始化完成，发送链接服务器信息--
function GameManager.OnInitOK()
    AppConst.SocketPort = 2012;
    AppConst.SocketAddress = "127.0.0.1";
    NetManager:SendConnect();

    this.test_class_func();
    this.test_pblua_func();
    this.test_cjson_func();
    this.test_pbc_func();
    this.test_lpeg_func();
    this.test_sproto_func();

    --createPanel("Prompt");
    CtrlManager.Init();
    local ctrl = CtrlManager.GetCtrl(CtrlName.Demo);
    if ctrl ~= nil and AppConst.ExampleMode then
        ctrl:Awake();
    end
```

（14）接下来修改 Scripts/Utility/Util.cs脚本，本案例不需要对框架性能进行测试，所以需要将图中的三句代码注释。

```
#if UNITY_EDITOR
            int resultId = Util.CheckRuntimeFile();
            if (resultId == -1) {
                Debug.LogError("没有找到框架所需要的资源，单击Game菜单下Build xxx Resource生成！！");
            //    EditorApplication.isPlaying = false;
                return false;
            } else if (resultId == -2) {
                Debug.LogError("没有找到Wrap脚本缓存，单击Lua菜单下Gen Lua Wrap Files生成脚本！！");
            //    EditorApplication.isPlaying = false;
                return false;
            }
#endif
            if (Application.loadedLevelName == "Test" && !AppConst.DebugMode) {
                Debug.LogError("测试场景，必须打开调试模式，AppConst.DebugMode = true！！");
            //    EditorApplication.isPlaying = false;
                return false;
            }
            return true;
        }
    }
}
```

（15）完成之后，修改AppConst.cs文件，开启更新模式，设置服务

```
public const bool UpdateMode = true;                    //更新模式-默认关闭

public const int TimerInterval = 1;
public const int GameFrameRate = 30;                    //游戏帧频

public const bool UsePbc = true;                        //PBC
public const bool UseLpeg = true;                       //LPEG
public const bool UsePbLua = true;                      //Protobuff-lua-gen
public const bool UseCJson = true;                      //CJson
public const bool UseSproto = true;                     //Sproto
public const bool LuaEncode = false;                     //使用LUA编码

public const string AppName = "SimpleFramework";        //应用程序名称
public const string AppPrefix = AppName + "_";          //应用程序前缀
public const string ExtName = ".assetbundle";           //素材扩展名
public const string AssetDirname = "StreamingAssets";   //素材目录
public const string WebUrl = "http://localhost:8080/StreamingAssets/";      //测试更新地址
```

器地址。

（16）接下来进行更新测试即可。

4.3 热更新原理分析

（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。

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。