**Unity3D教程：网游资源动态加载**

Posted on 2013年06月09日 by U3d / [Unity3D 基础教程](http://www.unitymanual.com/category/manual/unity3d-%e5%9f%ba%e7%a1%80%e6%95%99%e7%a8%8b)/被围观 109 次

用Unity3D制作基于web的网络游戏，不可避免的会用到资源动态加载。比如想加载一个大场景的资源，不应该在游戏的开始让用户长时间等待全部资源的加载完毕。应该优先加载用户附近的场景资源，在游戏的过程中，不影响操作的情况下，后台加载剩余的资源，直到所有加载完毕。

本文包含一些代码片段讲述实现这个技术的一种方法。本方法不一定是最好的，希望能抛砖引玉。代码是C#写的，用到了Json，还有C#的事件机制。

在讲述代码之前，先想象这样一个网络游戏的开发流程。首先美工制作场景资源的3D建模，游戏设计人员把3D建模导进Unity3D，托托拽拽编辑场景，完成后把每个gameobject导出成XXX.unity3d格式的资源文件（参看BuildPipeline），并且把整个场景的信息生成一个配置文件，xml或者Json格式（本文使用Json）。最后还要把资源文件和场景配置文件上传到服务器，最好使用CMS管理。客户端运行游戏时，先读取服务器的场景配置文件，再根据玩家的位置从服务器下载相应的资源文件并加载，然后开始游戏，注意这里并不是下载所有的场景资源。在游戏的过程中，后台继续加载资源直到所有加载完毕。

一个简单的场景配置文件的例子：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | { |
| 02 | "AssetList" : [{ |
| 03 | "Name" : "Chair 1", |
| 04 | "Source" : "Prefabs/Chair001.unity3d", |
| 05 | "Position" : [2,0,-5], |
| 06 | "Rotation" : [0.0,60.0,0.0] |
| 07 | }, |
| 08 | { |
| 09 | "Name" : "Chair 2", |
| 10 | "Source" : "Prefabs/Chair001.unity3d", |
| 11 | "Position" : [1,0,-5], |
| 12 | "Rotation" : [0.0,0.0,0.0] |
| 13 | }, |
| 14 | { |
| 15 | "Name" : "Vanity", |
| 16 | "Source" : "Prefabs/vanity001.unity3d", |
| 17 | "Position" : [0,0,-4], |
| 18 | "Rotation" : [0.0,0.0,0.0] |
| 19 | }, |
| 20 | { |
| 21 | "Name" : "Writing Table", |
| 22 | "Source" : "Prefabs/writingTable001.unity3d", |
| 23 | "Position" : [0,0,-7], |
| 24 | "Rotation" : [0.0,0.0,0.0], |
| 25 | "AssetList" : [{ |
| 26 | "Name" : "Lamp", |
| 27 | "Source" : "Prefabs/lamp001.unity3d", |
| 28 | "Position" : [-0.5,0.7,-7], |
| 29 | "Rotation" : [0.0,0.0,0.0] |
| 30 | }] |
| 31 | }] |
| 32 | } |

AssetList：场景中资源的列表，每一个资源都对应一个unity3D的gameobject

Name：gameobject的名字，一个场景中不应该重名

Source：资源的物理路径及文件名

Position：gameobject的坐标

Rotation：gameobject的旋转角度

你会注意到Writing Table里面包含了Lamp，这两个对象是父子的关系。配置文件应该是由程序生成的，手工也可以修改。另外在游戏上线后，客户端接收到的配置文件应该是加密并压缩过的。

主程序：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 |  |
| 02 |  |
| 03 | **public** **class** MainMonoBehavior : MonoBehaviour { |
| 04 |  |
| 05 | **public** **delegate** **void** MainEventHandler(GameObject dispatcher); |
| 06 |  |
| 07 | **public** **event** MainEventHandler StartEvent; |
| 08 |  |
| 09 | **public** **event** MainEventHandler UpdateEvent; |
| 10 |  |
| 11 | **public** **void** Start() { |
| 12 |  |
| 13 | ResourceManager.getInstance().LoadSence(“Scenes/MyDemoSence.txt”); |
| 14 |  |
| 15 | **if**(StartEvent != **null**){ |
| 16 |  |
| 17 | StartEvent(**this**.gameObject); |
| 18 |  |
| 19 | } |
| 20 |  |
| 21 | } |
| 22 |  |
| 23 | **public** **void** Update() { |
| 24 |  |
| 25 | **if** (UpdateEvent != **null**) { |
| 26 |  |
| 27 | UpdateEvent(**this**.gameObject); |
| 28 |  |
| 29 | } |
| 30 |  |
| 31 | } |
| 32 |  |
| 33 | } |
| 34 |  |
| 35 | 。。。 |

这里面用到了C#的事件机制，大家可以看看我以前翻译过的国外一个牛人的文章。C# 事件和Unity3D

在start方法里调用ResourceManager，先加载配置文件。每一次调用update方法，MainMonoBehavior会把update事件分发给ResourceManager，因为ResourceManager注册了MainMonoBehavior的update事件。

ResourceManager.cs

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | 。。。 |
| 02 | **private** MainMonoBehavior mainMonoBehavior; |
| 03 | **private** **string** mResourcePath; |
| 04 | **private** Scene mScene; |
| 05 | **private** Asset mSceneAsset; |
| 06 |  |
| 07 | **private** ResourceManager() { |
| 08 | mainMonoBehavior = GameObject.Find("Main Camera").GetComponent<MainMonoBehavior>(); |
| 09 | mResourcePath = PathUtil.getResourcePath(); |
| 10 | } |
| 11 |  |
| 12 | **public** **void** LoadSence(**string** fileName) { |
| 13 | mSceneAsset = new Asset(); |
| 14 | mSceneAsset.Type = Asset.TYPE\_JSON; |
| 15 | mSceneAsset.Source = fileName; |
| 16 |  |
| 17 | mainMonoBehavior.UpdateEvent += OnUpdate; |
| 18 | } |
| 19 | 。。。 |
| 20 |  |

在LoadSence方法里先创建一个Asset的对象，这个对象是对应于配置文件的，设置type是Json，source是传进来的“Scenes/MyDemoSence.txt”。然后注册MainMonoBehavior的update事件。

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **public** **void** OnUpdate(GameObject dispatcher) { |
| 02 | **if** (mSceneAsset != **null**) { |
| 03 | LoadAsset(mSceneAsset); |
| 04 | **if** (!mSceneAsset.isLoadFinished) { |
| 05 | **return**; |
| 06 | } |
| 07 |  |
| 08 | *//clear mScene and mSceneAsset for next LoadSence call* |
| 09 | mScene = **null**; |
| 10 | mSceneAsset = **null**; |
| 11 | } |
| 12 |  |
| 13 | mainMonoBehavior.UpdateEvent -= OnUpdate; |
| 14 | } |

OnUpdate方法里调用LoadAsset加载配置文件对象及所有资源对象。每一帧都要判断是否加载结束，如果结束清空mScene和mSceneAsset对象为下一次加载做准备，并且取消update事件的注册。

最核心的LoadAsset方法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **private** Asset LoadAsset(Asset asset) { |
| 02 |  |
| 03 | **string** fullFileName = mResourcePath + “/” + asset.Source; |
| 04 |  |
| 05 | *//if www resource is new, set into www cache* |
| 06 |  |
| 07 | **if** (!wwwCacheMap.ContainsKey(fullFileName)) { |
| 08 |  |
| 09 | **if** (asset.www == **null**) { |
| 10 |  |
| 11 | asset.www = new WWW(fullFileName); |
| 12 |  |
| 13 | **return** **null**; |
| 14 |  |
| 15 | } |
| 16 |  |
| 17 | **if** (!asset.www.isDone) { |
| 18 |  |
| 19 | **return** **null**; |
| 20 |  |
| 21 | } |
| 22 |  |
| 23 | wwwCacheMap.**Add**(fullFileName, asset.www); |
| 24 |  |
| 25 | } |
| 26 |  |
| 27 | 。。。 |

传进来的是要加载的资源对象，先得到它的物理地址，mResourcePath是个全局变量保存资源服务器的网址，得到fullFileName类似http://www.mydemogame.com/asset/Prefabs/xxx.unity3d。然后通过wwwCacheMap判断资源是否已经加载完毕，如果加载完毕把加载好的www对象放到Map里缓存起来。看看前面Json配置文件，Chair 1和Chair 2用到了同一个资源Chair001.unity3d，加载Chair 2的时候就不需要下载了。如果当前帧没有加载完毕，返回null等到下一帧再做判断。这就是WWW类的特点，刚开始用WWW下载资源的时候是不能马上使用的，要等待诺干帧下载完成以后才可以使用。可以用yield返回www，这样代码简单，但是C#要求调用yield的方法返回IEnumerator类型，这样限制太多不灵活。Unity3D教程手册

继续LoadAsset方法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | 。。。 |
| 02 | **if** (asset.Type == Asset.TYPE\_JSON) { *//Json* |
| 03 | **if** (mScene == **null**) { |
| 04 | **string** jsonTxt = mSceneAsset.www.text; |
| 05 | mScene = JsonMapper.ToObject<Scene>(jsonTxt); |
| 06 | } |
| 07 |  |
| 08 | *//load scene* |
| 09 | **foreach** (Asset sceneAsset **in** mScene.AssetList) { |
| 10 | **if** (sceneAsset.isLoadFinished) { |
| 11 | **continue**; |
| 12 | } **else** { |
| 13 | LoadAsset(sceneAsset); |
| 14 | **if** (!sceneAsset.isLoadFinished) { |
| 15 | **return** **null**; |
| 16 | } |
| 17 | } |
| 18 | } |
| 19 | } |
| 20 | 。。。 |
| 21 |  |

代码能够运行到这里，说明资源都已经下载完毕了。现在开始加载处理资源了。第一次肯定是先加载配置文件，因为是Json格式，用JsonMapper类把它转换成C#对象，我用的是LitJson开源类库。然后循环递归处理场景中的每一个资源。如果没有完成，返回null，等待下一帧处理。

继续LoadAsset方法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | 。。。 |
| 02 | **else** **if** (asset.Type == Asset.TYPE\_GAMEOBJECT) { *//Gameobject* |
| 03 | **if** (asset.gameObject == **null**) { |
| 04 | wwwCacheMap[fullFileName].assetBundle.LoadAll(); |
| 05 | GameObject go = (GameObject)GameObject.Instantiate(wwwCacheMap[fullFileName].assetBundle.mainAsset); |
| 06 | UpdateGameObject(go, asset); |
| 07 | asset.gameObject = go; |
| 08 | } |
| 09 |  |
| 10 | **if** (asset.AssetList != **null**) { |
| 11 | **foreach** (Asset assetChild **in** asset.AssetList) { |
| 12 | **if** (assetChild.isLoadFinished) { |
| 13 | **continue**; |
| 14 | } **else** { |
| 15 | Asset assetRet = LoadAsset(assetChild); |
| 16 | **if** (assetRet != **null**) { |
| 17 | assetRet.gameObject.transform.parent = asset.gameObject.transform; |
| 18 | } **else** { |
| 19 | **return** **null**; |
| 20 | } |
| 21 | } |
| 22 | } |
| 23 | } |
| 24 | } |
| 25 |  |
| 26 | asset.isLoadFinished = **true**; |
| 27 | **return** asset; |
| 28 | } |
| 29 |  |

终于开始处理真正的资源了，从缓存中找到www对象，调用Instantiate方法实例化成Unity3D的gameobject。UpdateGameObject方法设置gameobject各个属性，如位置和旋转角度。然后又是一个循环递归为了加载子对象，处理gameobject的父子关系。注意如果LoadAsset返回null，说明www没有下载完毕，等到下一帧处理。最后设置加载完成标志返回asset对象。Unity3D教程手册

UpdateGameObject方法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **private** **void** UpdateGameObject(GameObject go, Asset asset) { |
| 02 | *//name* |
| 03 | go.name = asset.Name; |
| 04 |  |
| 05 | *//position* |
| 06 | Vector3 vector3 = new Vector3((**float**)asset.Position[0], (**float**)asset.Position[1], (**float**)asset.Position[2]); |
| 07 | go.transform.position = vector3; |
| 08 |  |
| 09 | *//rotation* |
| 10 | vector3 = new Vector3((**float**)asset.Rotation[0], (**float**)asset.Rotation[1], (**float**)asset.Rotation[2]); |
| 11 | go.transform.eulerAngles = vector3; |
| 12 | } |

这里只设置了gameobject的3个属性，眼力好的同学一定会发现这些对象都是“死的”，因为少了脚本属性，它们不会和玩家交互。设置脚本属性要复杂的多，编译好的脚本随着主程序下载到本地，它们也应该通过配置文件加载，再通过C#的反射创建脚本对象，赋给相应的gameobject。

最后是Scene和asset代码：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **public** **class** Scene { |
| 02 | **public** List<Asset> AssetList { |
| 03 | **get**; |
| 04 | **set**; |
| 05 | } |
| 06 | } |
| 07 |  |
| 08 | **public** **class** Asset { |
| 09 |  |
| 10 | **public** **const** **byte** TYPE\_JSON = 1; |
| 11 | **public** **const** **byte** TYPE\_GAMEOBJECT = 2; |
| 12 |  |
| 13 | **public** Asset() { |
| 14 | *//default type is gameobject for json load* |
| 15 | Type = TYPE\_GAMEOBJECT; |
| 16 | } |
| 17 |  |
| 18 | **public** **byte** Type { |
| 19 | **get**; |
| 20 | **set**; |
| 21 | } |
| 22 |  |
| 23 | **public** **string** Name { |
| 24 | **get**; |
| 25 | **set**; |
| 26 | } |
| 27 |  |
| 28 | **public** **string** Source { |
| 29 | **get**; |
| 30 | **set**; |
| 31 | } |
| 32 |  |
| 33 | **public** **double**[] Bounds { |
| 34 | **get**; |
| 35 | **set**; |
| 36 | } |
| 37 |  |
| 38 | **public** **double**[] Position { |
| 39 | **get**; |
| 40 | **set**; |
| 41 | } |
| 42 |  |
| 43 | **public** **double**[] Rotation { |
| 44 | **get**; |
| 45 | **set**; |
| 46 | } |
| 47 |  |
| 48 | **public** List<Asset> AssetList { |
| 49 | **get**; |
| 50 | **set**; |
| 51 | } |
| 52 |  |
| 53 | **public** **bool** isLoadFinished { |
| 54 | **get**; |
| 55 | **set**; |
| 56 | } |
| 57 |  |
| 58 | **public** WWW www { |
| 59 | **get**; |
| 60 | **set**; |
| 61 | } |
| 62 |  |
| 63 | **public** GameObject gameObject { |
| 64 | **get**; |
| 65 | **set**; |
| 66 | } |
| 67 | } |

在实际测试中，会看到gameobject一个个加载并显示在屏幕中，并不会影响到游戏操作。代码还需要进一步完善适合更多的资源类型，如动画资源，文本，字体，图片和声音资源。