**Unity3D脚本：Unity3D实现动态加载游戏资源**

Posted on 2013年02月05日 by U3d / [Unity3D脚本/插件](http://www.unitymanual.com/category/script)/被围观 267 次

用Unity3D制作基于web的网络游戏，不可避免的会用到一个技术-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。

[post]

一个简单的场景配置文件的例子：

MyDemoSence.txt

Json代码：

{  
"AssetList" : [{  
"Name" : "Chair 1",  
"Source" : "Prefabs/Chair001.unity3d",  
"Position" : [2,0,-5],  
"Rotation" : [0.0,60.0,0.0]  
},  
{  
"Name" : "Chair 2",  
"Source" : "Prefabs/Chair001.unity3d",  
"Position" : [1,0,-5],  
"Rotation" : [0.0,0.0,0.0]  
},  
{  
"Name" : "Vanity",  
"Source" : "Prefabs/vanity001.unity3d",  
"Position" : [0,0,-4],  
"Rotation" : [0.0,0.0,0.0]  
},  
{  
"Name" : "Writing Table",  
"Source" : "Prefabs/writingTable001.unity3d",  
"Position" : [0,0,-7],  
"Rotation" : [0.0,0.0,0.0],  
"AssetList" : [{  
"Name" : "Lamp",  
"Source" : "Prefabs/lamp001.unity3d",  
"Position" : [-0.5,0.7,-7],  
"Rotation" : [0.0,0.0,0.0]  
}]  
}]  
}

AssetList：场景中资源的列表，每一个资源都对应一个Unity3D的gameobjectName：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：

C#代码：

。。。  
public class MainMonoBehavior : MonoBehaviour {  
public delegate void MainEventHandler(GameObject dispatcher);  
public event MainEventHandler StartEvent;  
public event MainEventHandler UpdateEvent;  
public void Start() {  
ResourceManager.getInstance().LoadSence("Scenes/MyDemoSence.txt");  
if(StartEvent != null){  
StartEvent(this.gameObject);  
}  
}  
public void Update() {  
if (UpdateEvent != null) {  
UpdateEvent(this.gameObject);  
}  
}  
}  
。。。  
}

　　这里面用到了C#的事件机制。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

C#代码：

。。。  
private MainMonoBehavior mainMonoBehavior;  
private string mResourcePath;  
private Scene mScene;  
private Asset mSceneAsset;  
private ResourceManager() {  
mainMonoBehavior = GameObject.Find("Main Camera").GetComponent<MainMonoBehavior>();  
mResourcePath = PathUtil.getResourcePath();  
}  
public void LoadSence(string fileName) {  
mSceneAsset = new Asset();  
mSceneAsset.Type = Asset.TYPE\_JSON;  
mSceneAsset.Source = fileName;  
mainMonoBehavior.UpdateEvent += OnUpdate;  
}  
。。。

　　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。

C#代码：

public void OnUpdate(GameObject dispatcher) {  
if (mSceneAsset != null) {  
LoadAsset(mSceneAsset);  
if (!mSceneAsset.isLoadFinished) {  
return;  
}  
//clear mScene and mSceneAsset for next LoadSence call  
mScene = null;  
mSceneAsset = null;  
}  
mainMonoBehavior.UpdateEvent -= OnUpdate;  
}

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：

C#代码：

private Asset LoadAsset(Asset asset) {  
string fullFileName = mResourcePath + "/" + asset.Source;  
//if www resource is new, set into www cache  
if (!wwwCacheMap.ContainsKey(fullFileName)) {  
if (asset.www == null) {  
asset.www = new WWW(fullFileName);  
return null;  
}  
if (!asset.www.isDone) {  
return null;  
}  
wwwCacheMap.Add(fullFileName, asset.www);  
}  
。。。

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。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。继续LoadAsset方法：

C#代码：

。。。  
if (asset.Type == Asset.TYPE\_JSON) { //Json  
if (mScene == null) {  
string jsonTxt = mSceneAsset.www.text;  
mScene = JsonMapper.ToObject<Scene>(jsonTxt);  
}  
//load scene  
foreach (Asset sceneAsset in mScene.AssetList) {  
if (sceneAsset.isLoadFinished) {  
continue;  
} else {  
LoadAsset(sceneAsset);  
if (!sceneAsset.isLoadFinished) {  
return null;  
}  
}  
}  
}  
。。。

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：

C#代码：

。。。  
else if (asset.Type == Asset.TYPE\_GAMEOBJECT) { //Gameobject  
if (asset.gameObject == null) {  
wwwCacheMap[fullFileName].assetBundle.LoadAll();  
GameObject go = (GameObject)GameObject.Instantiate(wwwCacheMap[fullFileName].assetBundle.mainAsset);  
UpdateGameObject(go, asset);  
asset.gameObject = go;  
}  
if (asset.AssetList != null) {  
foreach (Asset assetChild in asset.AssetList) {  
if (assetChild.isLoadFinished) {  
continue;  
} else {  
Asset assetRet = LoadAsset(assetChild);  
if (assetRet != null) {  
assetRet.gameObject.transform.parent = asset.gameObject.transform;  
} else {  
return null;  
}  
}  
}  
}  
}  
asset.isLoadFinished = true;  
return asset;  
}

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。UpdateGameObject方法：

C#代码：

private void UpdateGameObject(GameObject go, Asset asset) {  
//name  
go.name = asset.Name;  
//position  
Vector3 vector3 = new Vector3((float)asset.Position[0], (float)asset.Position[1], (float)asset.Position[2]);  
go.transform.position = vector3;  
//rotation  
vector3 = new Vector3((float)asset.Rotation[0], (float)asset.Rotation[1], (float)asset.Rotation[2]);  
go.transform.eulerAngles = vector3;  
}

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：

C#代码：

public class Scene {  
public List<Asset> AssetList {  
get;  
set;  
}  
}  
public class Asset {  
public const byte TYPE\_JSON = 1;  
public const byte TYPE\_GAMEOBJECT = 2;  
public Asset() {  
//default type is gameobject for json load  
Type = TYPE\_GAMEOBJECT;  
}  
public byte Type {  
get;  
set;  
}  
public string Name {  
get;  
set;  
}  
public string Source {  
get;  
set;  
}  
public double[] Bounds {  
get;  
set;  
}  
public double[] Position {  
get;  
set;  
}  
public double[] Rotation {  
get;  
set;  
}  
public List<Asset> AssetList {  
get;  
set;  
}  
public bool isLoadFinished {  
get;  
set;  
}  
public WWW www {  
get;  
set;  
}  
public GameObject gameObject {  
get;  
set;  
}  
}

　　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。