# Unity资源处理机制（Assets/WWW/AssetBundle/...）读取和加载资源方式详解

Unity资源机制

1、概述

本文意在阐述Unity资源机制相关的信息，以及一些关于个人的理解与试验结果。另外还会提及一些因机制问题可能会出现的异常以及处理建议。大部分机制信息来源于官方文档，另外为自我验证后的结果。

2、资源

概述

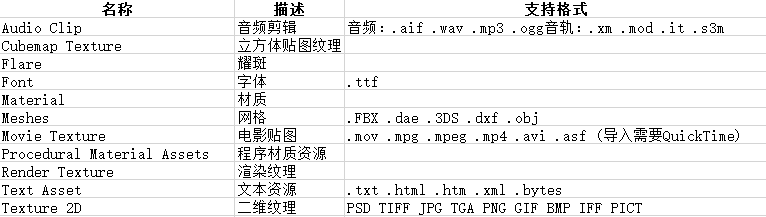
Unity必须通过导入将所支持的资源序列化，生成AssetComponents后，才能被Unity使用。以下是Unity对Assets的描述：

Assets are the models，textures，sounds and all other “content”files from which you make your game。

资源(Asset)是硬盘中的文件，存储在Unity工程的Assets文件夹内。有些资源的数据格式是Unity原声支持的，有些资源则需要转换为源生的数据格式后才能被使用。

对象(UnityEngine.Object)，代表序列化数据的集合，表示某个资源的具体实例。它可以是Unity使用的任何类型的资源，所有对象都是UnityEngine.Object基类的子类

资源与对象时一对多的关系。



除此之外，想使用Unity不支持导入，或者未经导入的资源，只能使用IO Stream或者WWW 方法，这些将在下文对应栏目中说明。

注意：AssetBundle不是资源组件，故无法用资源组件的方式载入，只能使用WWW或者AssetBundle相关接口载入与读取

GUID与fileID(本地ID)

Unity会为每个导入到Assets目录中的资源创建一个meta文件，文件中记录了GUID，GUID用来记录资源之间的引用关系。还有fileID（本地ID），用于标识资源内部的资源。资源间的依赖关系通过GUID来确定；资源内部的依赖关系使用fileID来确定。

InstanceID(实例ID)

Unity为了在运行时，提升资源管理的效率，会在内部维护一个缓存表，负责将文件的GUID与fileID转换成为整数数值，这个数值在本次会话中是唯一的，称作实例ID(InstanceID)。

程序启动时，实例ID缓存与所有工程内建的对象(例如在场景中被引用)，以及Resource文件夹下的所有对象，都会被一起初始化。如果在运行时导入了新的资源，或从AssetBundle中载入了新的对象，缓存会被更新，并为这些对象添加相应条目。实例ID仅在失效时才会被从缓存中移除，当提供了指定文件GUID和fileID的AssetBundle被卸载时会产生移除操作。

卸载AssetBundle会使实例ID失效，实例ID与其文件GUID和fileID之间的映射会被删除以便节省内存。重新载入AssetBundle后，载入的每个对象都会获得新的实例ID。

资源的生命周期

Object从内存中加载或卸载的时间点是定义好的。Object有两种加载方式：自动加载与外部加载。当对象的实例ID与对象本身解引用，对象当前未被加载到内存中，而且可以定位到对象的源数据，此时对象会被自动加载。对象也可以外部加载，通过在脚本中创建对象或者调用资源加载API来载入对象（例如：AssetBundle.LoadAsset）

对象加载后，Unity会尝试修复任何可能存在的引用关系，通过将每个引用文件的GUID与FileID转化成实例ID的方式。一旦对象的实例ID被解引用且满足以下两个标准时，对象会被强制加载：

实例ID引用了一个没有被加载的对象。

实例ID在缓存中存在对应的有效GUID和本地ID。

如果文件GUID和本地ID没有实例ID，或一个已卸载对象的实例ID引用了非法的文件GUID和本地ID，则引用本身会被保留，但实例对象不会被加载。在Unity编辑器中表现为空引用，在运行的应用中，或场景视图里，空对象会以多种方式表示，取决于丢失对象的类型：网格会变得不可见，纹理呈现为紫红色等等。

MonoScripts:一个MonoScripts含有三个字符串：程序库名称，类名称，命名空间。

构建工程时，Unity会收集Assets文件夹中独立的脚本文件并编译他们，组成一个Mono程序库。Unity会将Assets目录中的语言分开编译，Assets/Plugins目录中的脚本同理。Plugin子目录之外的C#脚本会放在Assembly-CSharp.dll中。而Plugin及其子目录中的脚本则放置在Assembly-CSharp-firstpass.all中。

这些程序库会被MonoScripts所引用，并在程序第一次启动时被加载。

3、资源文件夹

Assets

为Unity编辑器下的资源文件夹，Unity项目编辑时的所有资源都将置入此文件夹内。在编辑器下，可以使用以下方法获得资源对象：

AssetDatabase.LoadAssetAtPath("Assets/x.txt");

注意：此方法只能在编辑器下使用，当项目打包后，在游戏内无法运作。参数为包含Assets内的文件全路径，并且需要文件后缀。

Assets下的资源除特殊文件夹内，或者在会打入包内的场景中引用的资源，其余资源不会被打入包中。

Resources

资源载入

Assets下的特殊文件夹，此文件夹内的资源将会在项目打包时，全部打入包内，并能通过以下方法获得对象：

Resources.Load("fileName");

注意：函数内的参数为相对于Resource目录下的文件路径与名称，不包含后缀。Assets目录下可以拥有任意路径及数量的Resources文件夹，在运行时,Resources下的文件路径将被合并。

例：Assets/Resources/test.txt与 Assets/TestFloder/Resources/test.png在使用Resource.Load("test")载入时，将被视为同一资源，只会返回第一个符合名称的对象。如果使用Resource.Load(“test”)将返回text.txt；

如果在Resources下有相同路径及名称的资源，使用以上方法只能获得第一个符合查找条件的对象，使用以下方法能或得到所有符合条件的对象：

Object[] assets = Resources.LoadAll("fileName");

TextAsset[] assets = Resources.LoadAll("fileName");

相关机制

在工程进行打包后，Resource文件夹中的资源将进行加密与压缩，打包后的程序内将不存在Resource文件夹，故无法通过路径访问以及更新资源。

依本文2.3章节所述，在程序启动时会为Resource下的所有对象进行初始化，构建实例ID。随着Resource内资源的数量增加，此过程耗时的增加是非线性的。故会出现程序启动时间过长的问题，请密切留意Resource内的资源数量。

卸载资源

所有实例化后的GameObject 可以通过Destroy函数销毁。请留意Object与GameObject之间的区别与联系

Object可以通过Resources中的相关Api进行卸载

Resources.UnloadAsset(Object);//卸载对应Object

Resources.UnloadUnusedAssets();//卸载所有没有被引用以及实例化的Object

注意以下情况：

Object obj = Resources.Load("MyPrefab");

GameObject instance = Instantiate(obj) as GameObjct;

......

Destroy(instance);

Resources.UnloadUnusedAssets();

此时UnloadUnusedAssets将不会生效，因为obj依然引用了MyPrefab，需要将obj = null，才可生效。

StreamingAssets

概述

StreamingAssets文件夹为流媒体文件夹，此文件夹内的资源将不会经过压缩与加密，原封不动的打包进游戏包内。在游戏安装时，StreamAssets文件件内的资源将根据平台，移动到对应的文件夹内。StreamingAssets文件夹在Android与IOS平台上为只读文件夹.

你可以使用以下函数获得不同平台下的StreamingAssets文件夹路径：

Application.streamingAssetsPath

请参考以下各平台下StreamingAssets文件夹的等价路径，Application.dataPath为程序安装路径。Android平台下的路径比较特殊，请留意此路径的前缀，在一些资源读取的方法中是不必要的（AssetBundle.LoadFromFile，下详）

Application.dataPath+"/StreamingAssets"//Windows OR MacOS

Application.dataPath+"/Raw" //IOS

"jar:file://"+Application.dataPath+"!/assets/" //Android

文件读取

StreamingAssets文件夹下的文件在游戏中只能通过IO Stream或者WWW的方式读取（AssetBundle除外）

IO Stream方式

using(FileStream stream = File.Open(Application.streamingAssetsPath+"fileName",

FileMode.Open)) {

//处理方法

}

WWW方式（注意协议与不同平台下路径的区别）

using(WWW www = new WWW(

Application.streamingAssetsPath+"fileName"))

{

yield return www;

www.text;

www.texture;

}

AssetBundle特有的同步读取方式（注意安卓平台下的路径区别）

string assetbundlePath =

#if UNITY\_ANDROID

Application.dataPath+"!/assets";

#else

Application.streamingAssetsPath;

#endif

AssetBundle.LoadFromFile(assetbundlePath+"/name.unity3d");

PersistentDataPath

Application.persistentDataPath

Unity指定的一个可读写的外部文件夹，该路径因平台及系统配置不同而不同。可以用来保存数据及文件。该目录下的资源不会在打包时被打入包中，也不会自动被Unity导入及转换。该文件夹只能通过IO Stream以及WWW的方式进行资源加载。

4、WWW载入资源

概述

WWW是一个Unity封装的网络下载模块，支持Http以及file两种URL协议，并会尝试将资源转换成Unity能使用的AssetsComponents（如果资源是Unity不支持的格式，则只能取出byte[]）。具体对应的格式参考第一章表格。WWW加载是异步方法。

byte[] bytes = WWW.bytes;

string text = WWW.text;

Texture2D texture = WWW.texture;

MovieTexture movie = WWW.movie;

AssetBundle assetbundle = WWW.assetBundle;

AudioClip audioClip = WWW.audioClip;

相关机制

new WWW

每次new WWW时，Unity都会启用一个线程去进行下载。通过此方式读取或者下载资源，会在内存中生成WebStream，WebStream为下载文件转换后的内容，占用内存较大。使用WWW.Dispose将终止仍在加载过程中的进程，并释放掉内存中的WebStream。

如果WWW不及时释放，将占用大量的内存，推荐搭配using方式使用，以下两种方式等价。

WWW www = new WWW(Application.streamingAssetsPath+"fileName");

try

{

yield return www;

www.text;

www.texture;

}

finally

{

www.Dispose();

}

using(WWW www = new WWW(

Application.streamingAssetsPath+"fileName"))

{

yield return www;

www.text;

www.texture;

}

如果载入的为Assetbundle且进行过压缩，则还会在内存中占用一份AssetBundle解压用的缓冲区Deompresion Buffer,AssetBundle压缩格式的不同会影响此区域的大小。

WWW.LoadFromCacheOrDownload

int version = 1;

WWW.LoadFromCacheOrDownload(PathURL+"/fileName",version);

使用此方式加载，将先从硬盘上的存储区域查找是否有对应的资源，再验证本地Version与传入值之间的关系，如果传入的Version>本地，则从传入的URL地址下载资源，并缓存到硬盘，替换掉现有资源，如果传入Version<=本地，则直接从本地读取资源；如果本地没有存储资源，则下载资源。此方法的存储路径无法设定以及访问。使用此方法载入资源，不会在内存中生成 WebStream（其实已经将WebStream保存在本地），如果硬盘空间不够进行存储，将自动使用new WWW方法加载，并在内存中生成WebStream。在本地存储中，使用fileName作为标识符，所以更换URL地址而不更改文件名，将不会造成缓存资源的变更。

保存的路径无法更改，也没有接口去获取此路径

5、 AssetBundle

概述

AssetBundles let you stream additional assets via the WWW class and instantiate them at runtime. AssetBundles are created via BuildPipeline.BuildAssetBundle.

AssetBundle是Unity支持的一种文件储存格式，也是Unity官方推荐的资源存储与更新方式，它可以对资源(Asset)进行压缩，分组打包，动态加载，以及实现热更新，但是AssetBundle无法对Unity脚本进行热更新，因为其需要在打包时进行编译。

Assetbundle打包

平台兼容性

AssetBundle适用于多种平台，但不同平台所使用的AssetBundle并不相同，在创建AssetBundle时需要通过参数来指定目标平台，其关系如下表



创建API

public enum BuildAssetBundleOptions

{

None = 0,

//Build assetBundle without any special option.

UncompressedAssetBundle = 1,

//Don't compress the data when creating the asset bundle.

CollectDependencies = 2,

//Includes all dependencies.

CompleteAssets = 4,

//Forces inclusion of the entire asset.

DisableWriteTypeTree = 8,

//Do not include type information within the AssetBundle.

DeterministicAssetBundle = 16,

//Builds an asset bundle using a hash for the id

ForceRebuildAssetBundle = 32,

//Force rebuild the assetBundles.

IgnoreTypeTreeChanges = 64,

//Ignore the type tree changes when doing the incremental build check.

AppendHashToAssetBundleName = 128,

//Append the hash to the assetBundle name.

ChunkBasedCompression = 256

//Use chunk-based LZ4 compression when creating the AssetBundle.

}

AssetBundleManifest manifest =

BuildPipeline.BuildAssetBundles("OutputPath",BuildAssetBundleOptions,tragetPlatform);

在Unity的5.3版本中，简化了AssetBundle的打包方式，只留下了一个api与寥寥几个设置参数，而之前最让人头痛的资源依赖管理，也被默认进行处理。 而在每个Asset文件的Inspector面板上都会多出一个Asset Labels的设定栏：



AssetBundle name：需要将此资源打包的AssetBundle名称

AssetBundle Variant：需要将此资源打包的AssetBundle的变体名

Variant

Variant是5.3以后新添加的一个概念，这个值其实是一个尾缀，将添加在对应AssetBundle的名称之后，如：ddzgame.hd，hd就是Variant(从此以后AssetBundle的尾缀已经跟其文件类型本身没有任何联系)。

自动打包脚本

从以上可知，如果需要一个一个的对资源设置AssetBundle Name与Variant实在太过繁琐与麻烦，也可能出现纰漏，好在可以通过脚本去批量设置这两个参数：

AssetImporter assetImporter = AssetImporter.GetAtPath("path");

assetImporter.assetBundleName = "Assetbundle Name";

assetImporter.assetBundleVariant = "Assetbundle Variant";

其中path是资源在Assets目录下的路径。

Scene打包

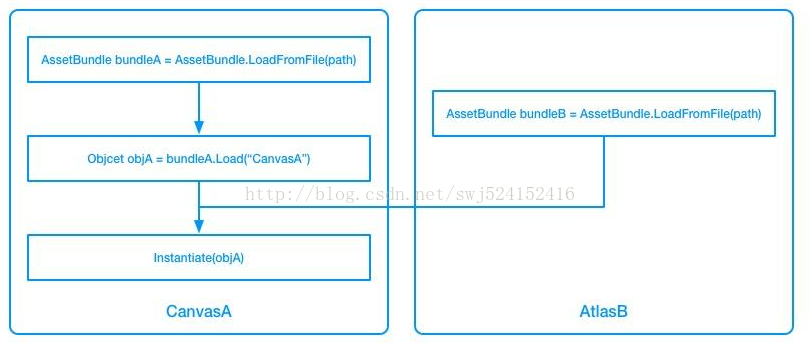
Scene打包跟资源打包无异，唯一需要注意的是：Scene只能与Scene打入同一个AssetBundle内，而无法与其他资源打入同一个AssetBundle。

PS：AssetBundle内的Scene需要在AssetBundle加载后，通过SceneManager来加载。

AssetBundle依赖

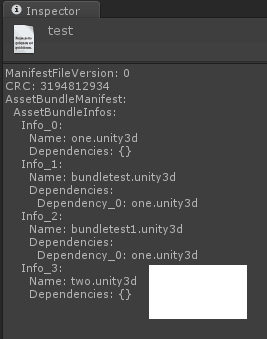
依赖机制

假设有AssetBundleA与 AssetBundleB两个AssetBundle，AssetBundle中的资源引用了AssetBundleB中的资源，则称AssetBundleA依赖于AssetBundleB。具体实例请看下图注意被依赖AssetBundle需要加载的时机



注意其依赖的机制: AssetBundle中保存有其中所有资源的GUID,FileID等序列化信息，AssetBundle只会在内存中寻找其依赖资源所在的AssetBundle，并自动从中加载出所需资源。具体可参考本文2.3章节

Manifest



在前面有提到，在5.3中，Unity会自动处理AssetBundle中资源的依赖关系。在默认情况下，如果AssetBundle间有交叉的资源引用，不会再重复打包，在打包AssetBundle后，会发现其在输出目录多出了一个与目录名称相同的无后缀AssetBundle文件，其为自动生成的AssetBundleManifest文件，其内保存有此次生成的所有AssetBundle之间的依赖关系与清单。我们可以在载入这个AssetBundle后使用以下方法获得此对象。

AssetBundle.LoadAsset("AssetBundleManifest");

Manifest保存有重要的依赖信息，在载入AssetBundle时，可以通过Manifest查询其是否有依赖的AssetBundle，然后我们手动对其进行管理，避免依赖项丢失而出现bug

string[] fullnames = AssetBundle.GetDirectDependencies(fullname);

string[] fullnames = AssetBundle.GetAllDependencies(fullname);

Direct方法会返回所有直接依赖的AssetBundle名称数组，All方法会返回所有依赖的AssetBundle名称数组，fullname包括名称与Variant。推荐使用Direct方法做递归处理，避免重复载入。

AssetBundle加载

加载方式

之前已经提及，不再详细说明，使用WWW 或者 AssetBundle相关API加载，其中AssetBundle的API只能进行本地加载。

AssetBundle.LoadfromMemory（byte[] bytes）

此API是一个例外，用来对加密的Assetbundle进行读取，可以结合WWW使用。

压缩

LZMA（Ziv-Markov chain algorithm）格式

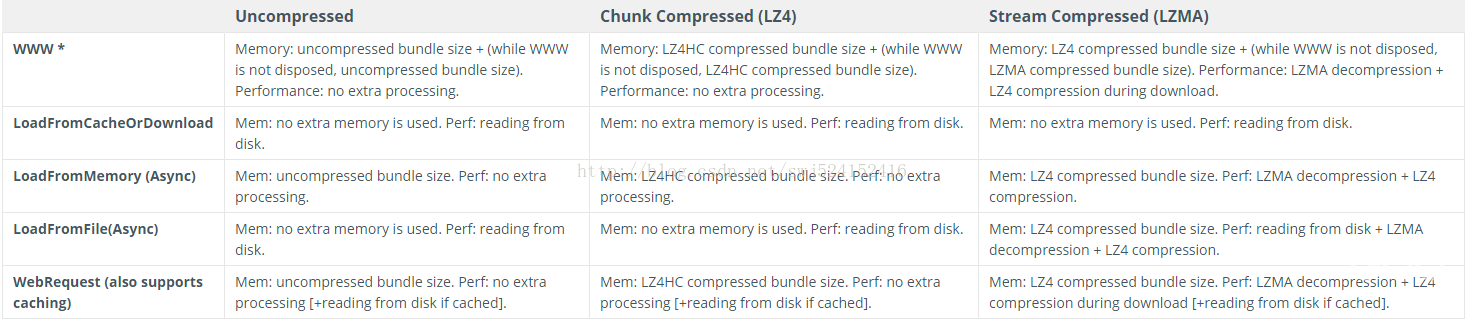
Unity打包成AssetBundle时的默认格式，会将序列化数据压缩成LZMA流，使用时需要整体解包。优点是打包后体积小，缺点是解包时间长，且占用内存。

LZ4格式

5.3新版本添加的压缩格式，压缩率不及LZMA，但是不需要整体解压。LZ4是基于chunk的算法，加载对象时只有响应的chunk会被解压。

压缩格式在打包时通过AssetBundleOption参数选择。

内存占用



AssetBundle加载后会在内存中生成AssetBundle的序列化架构的占用，一般来说远远小于资源本身，除非包含复杂的序列化信息(复杂多层级关系或复杂静态数据的prefab等)

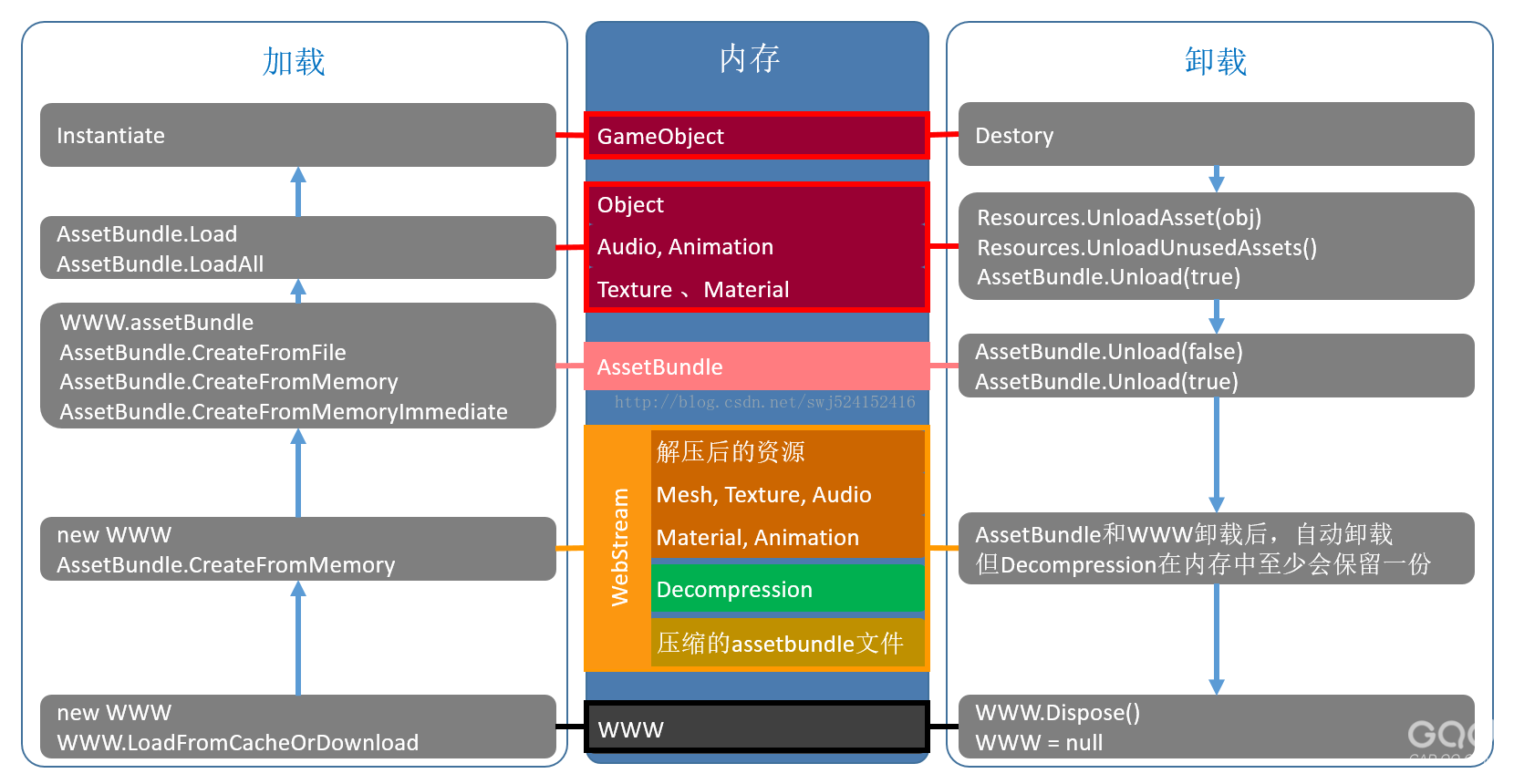
AssetBundle卸载

卸载API

AssetBundle.Unload(bool unloadAllLoadedObjects);

AssetBundle只有唯一的一个卸载函数，传入的参数用来选择是否将已经从此AssetBundle中加载的资源一起卸载。另外，已经从AssetBundle中加载的资源可以通过Resources.UnloadAsset(Object)卸载。如果想通过Resources.UnloadUnusedAssets()卸载从AssetBundle加载的资源，一定要先将AssetBundle卸载后才能生效。

资源卸载总览



当AssetBundle被卸载后，实例ID与其文件GUID和本地ID之间的映射会被删除， 即其无法被其后加载的依赖于它的资源所查找及引用。详情请参考本文2.3章节

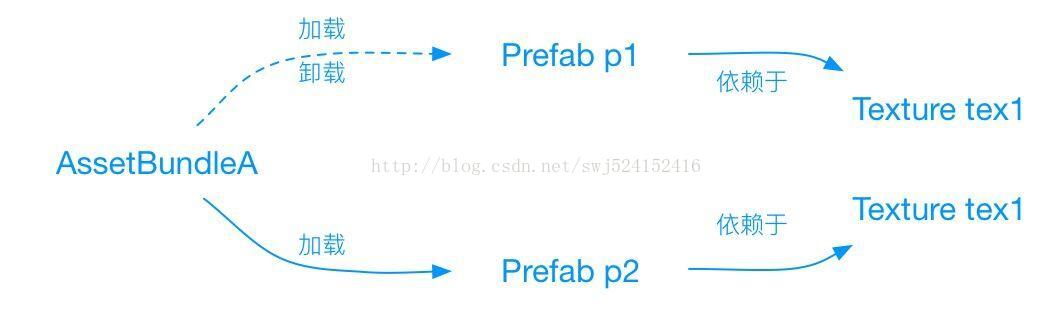
案例分析

案例1 游戏切换到后台一段时候切回，出现shader或者Texture丢失。

在移动平台，当程序切到主界面或者在后台长时间运行时，GPU会自动对后台程序的资源进行清理。如果shader或者Texture是从AssetBundle中加载出来，而此AssetBundle已经被卸载的话，Unity无法在程序恢复时从内存中加载这些资源，从而造成丢失。有人会问，这些资源不是已经加载到内存中了么？但是，他们在被加载到GPU之后会被从内存中清除。因此要防止此状况最稳健的方法，就是在场景切换前，不要卸载掉其所属的AssetBundle。

案例2 当经常使用AssetBundleB.Unload(false)卸载时，有时会发现AssetBundle中的资源在内存中有多份同时存在。

问题的根源在于从AssetBundle中加载出来的资源，在该AssetBundle卸载之后与其的联系就断开了。



例如：从AssetBundleA中加载出来一个Prefab p1，p1依赖资源tex1也会自动加载到内存中。然后用AssetBundle.Unload(false)卸载AssetBundleA，此时p1与AssetBundleA的联系断开。之后，从AssetBundleA中加载Prefab p2，p2也依赖资源tex1，那么在加载p2时tex1会再次被加载到内存中，导致重复。

---------------------

原文：https://blog.csdn.net/swj524152416/article/details/54022282