<https://blog.csdn.net/u010377179/article/details/52922710>

<https://blog.csdn.net/Daniel_love/article/details/51699128>

<http://www.cnblogs.com/ybgame/p/3973177.html>

<https://blog.csdn.net/langresser_king/article/details/44208585/>

<https://blog.csdn.net/huutu/article/details/46477471>

<https://blog.csdn.net/hanghangaidoudou/article/details/78170591>

<https://www.cnblogs.com/lan-yt/p/7787290.html>

<https://www.cnblogs.com/WongSiuming/p/7574312.html>

解决疑惑的文章

<https://blog.csdn.net/bloodshadow/article/details/52923492>

<https://blog.csdn.net/UWA4D/article/details/53708698>

[https://blog.csdn.net/lodypig/article/category/6315960](https://blog.csdn.net/lodypig/article/category/6315960   )

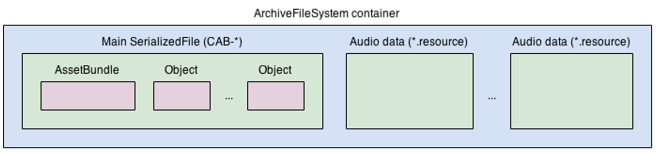
assetbundle五篇详细讲解

AssetBundle即资源包，是Unity Pro提供和推荐的资源打包方式，它可以把多个自定义的游戏对象或者资源以二进制形式保存到AssetBundle文件中，AssetBundle支持所有unity可识别的格式：模型、贴图、音频、整个场景等，其中最为方便的是可以将关联的内容制作成一个prefab，例如一个模型的贴图、动作和模型等，然后将整个prefab导出到AssetBundle，Unity会自己收集该Prefab使用到的关联文件，将其一并打入AssetBundle文件，并保留prefab中资源和脚本之间相互关联。

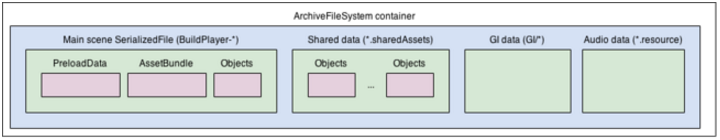
（1）AssetBundle内部格式

一个AssetBundle本质上是将一些对象组合成一个序列化文件，根据是普通包（normal bundle）还是场景包（scene bundle），AssetBundle的数据文件展开略有不同

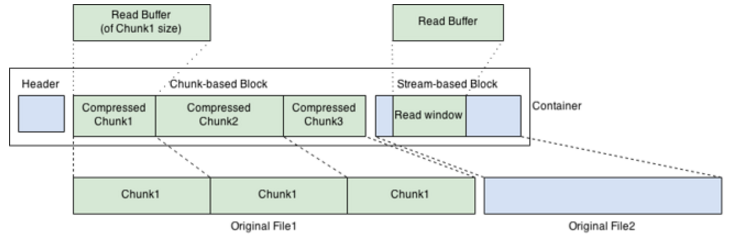
普通包：普通包由本身信息（AssetBundle）、各个对象和音频文件组成



场景包：场景包还包含了预加载数据（PreloadData）、共享数据（shadedData）和全局光照数据（Global Illumination Data）



AssetBundle可以选择是否压缩，Unity5.3之前只能使用LZMA压缩，Unity5.3之后支持LZ4压缩



官网意译：图中压缩部分表示可有块压缩（chunk-based）、流压缩（strean-baseed）两种方式。流压缩（LZMA）在处理整个数据块时使用同一个字典，它提供了最大可能的压缩率但只支持顺序读取。块压缩（LZ4）指的是原始数据被分成大小相同的子块并单独压缩。如果想要实时解压/随机读取开销小，应该使用这种方式。

<https://blog.csdn.net/ldghd/article/details/8069489>

Unity里面有两种动态加载机制：一是Resources.Load，一是通过AssetBundle。其实两者本质上没有什么区别，Resources.Load就是从一个缺省打进程序包里的AssetBundle里加载资源，而一般AssetBundle文件需要你自己创建，运行时动态加载，可以指定路径和来源，其实场景中所有静态的对象也有这么一个加载过程，只是Unity后台自动完成了。

**内部格式：**

<https://blog.codingnow.com/2014/08/unity3d_asset_bundle.html>

这里有一篇[Unity3D asset bundle 格式简析](http://blog.codingnow.com/2014/08/unity3d_asset_bundle.html)，分析了unity3.5下Assetbundle的内部格式，总结下来结构如下：

* **AssetBundleFileHead** : 记录了版本、是否压缩等主要描述信息。
* **AssetFileHeader** ：包含一个文件列表，记录了每个资源的name，offset，length等。
* **Asset1** ： 第一个资源本身，内部结构如下   
  + **AssetHeader** ：包含了TypeTree大小、文件大小、format等。
  + **TypeTree（可选）**:记录了asset 对象的class id，在 [Unity3d 的官方文档](http://docs.unity3d.com/Manual/ClassIDReference.html)（https://docs.unity3d.com/Manual/ClassIDReference.html） 可以查到每个id的对象。
  + **ObjectPath** ：主要记录了一个pathID（资源唯一索引id）。
  + **AssetRef**：记录了AssetBundle对外部资源的引用情况。
* **Asset2** ：第二个资源，结构跟上面一样
* ……> ： 更多资源

打包时的BuildAssetBundleOptions

[Flags]

public enum BuildAssetBundleOptions

{

None = 0, //没有任何特殊要求

UncompressedAssetBundle = 1,//不压缩

[Obsolete("This has been made obsolete. It is always enabled in the new AssetBundle build system introduced in 5.0.")]

CollectDependencies = 2, //

[Obsolete("This has been made obsolete. It is always enabled in the new AssetBundle build system introduced in 5.0.")]

CompleteAssets = 4, //

DisableWriteTypeTree = 8, //AssetBundle中不包含Type信息

DeterministicAssetBundle = 16, //使用资源的Hash ID来导出AssetBundle，使用ID可避免资源改名。移动位置等导致重新导出

ForceRebuildAssetBundle = 32, //强制重新导出，对已有的AssetBundle，在资源没有变化时，Unity不会重新导出

IgnoreTypeTreeChanges = 64, //增量打包时忽略Type信息变化

AppendHashToAssetBundleName = 128, //在AssetBundle名称后添加”\_”加上Hash值

ChunkBasedCompression = 256, //使用块压缩，即LZ4压缩

StrictMode = 512, //

OmitClassVersions = 1024, //

}

**同步加载**

AssetBundle AssetBundle.LoadFromFile(string path);

Android上直接使用Application.streamingAssetPath进行加载会包can not archive的错误，原因是加载时路径不对，可以看到，android平台下Application.streamingAssetPath以jar://file://开头，可其他平台都不一样，这这段去掉就就能够加载，而去掉后等价于Application.datapath+”!assets”，所以在android下我们用该路径加载

#if UNITY\_ANDROID

STREAMING\_PATH\_SYNC = Application.dataPath + "!assets";

#else

STREAMING\_PATH\_SYNC = Application.streamingAssetsPath;

#endif

**异步加载**

AssetBundle AssetBundle.LoadFromFileAsync(string path);路径同LoadFromFile

www异步加载：

#if UNITY\_ANDROID

Application.streamingAssetsPath; // 路径与上面不同，安卓直接用这个

#elif

"file://" + Application.streamingAssetsPath; // 反而其他平台加file://

#endif

以上三种方式都可以从一个完整的StreamingAssetPath下，加载到一个AssetBundle对象，接着我们可以从AssetBundle对象中，加载出Asset资源，也有同步和异步两种方式：

Object AssetBundle.LoadAsset(string assetName, Type resType);

AssetBundleRequest AssetBundle.LoadAssetAsync(string assetName, Type resType);

在Unity的导出中，有一个BuildAssetBundleOptions，其实还有隐藏的选项，即BuildAssetBundleOptions.CollectDependencies，在Unity5中这一项是默认开启的，代表会自动收集所选资源的依赖资源。例如一个prefab中引用到了一张图片，unity5会自动将该图片也一并打包，避免因资源遗漏导致关联丢失的情况出现。

**采用Resources加载资源**

GameObject prefab = (GameObject)Resources.Load("prefab2", typeof(GameObject));

GameObject t = Instantiate(prefab) as GameObject;

t.name = "clone\_prefab1";

Debug.Log("name = " + prefab.name);

在上面的例子中尝试过建立多个Resources文件，实际上都可以找到prefab，注意Resources.Load不需要添加资源后缀名

**AssetBundle运行时加载**

来自文件就用CreateFromFile（注意这种方法只能用于standalone程序），这是最快的加载方法

也可以来自Memory，用CreateFromMemory(byte[])，这个byte[]可以来自文件读取的缓冲，www的下载或者其他可能的方式

其实www的assetBundle就是内部数据读取万后自动创建了一个assetBundle而已

Create完成以后，等于把硬盘或者网络的一个文件读到内存的 一个区域，这时候只是一个AssetBundle内存镜像数据块，还没有Assets的概念

Assets加载：

用AssetBundle.Load（同Resources.Load）这才会从AssetBundle的内存镜像中读取并创建一个Asset对象，创建Asset对象同时也会分配相应内存用于存放（反序列化），异步读取用AssetBundle.LoadAsync，也可以一次读取多个，用AssetBundle.LoadAll

**AssetBundle的释放**

AssetBundle.Unload(false)是释放AssetBundle文件的内存镜像，不包含Load创建的Asset内存对象

AssetBundle.Unload(true)是释放AssetBundle文件的内存镜像并销毁所有用 Load创建的Asset内存对象

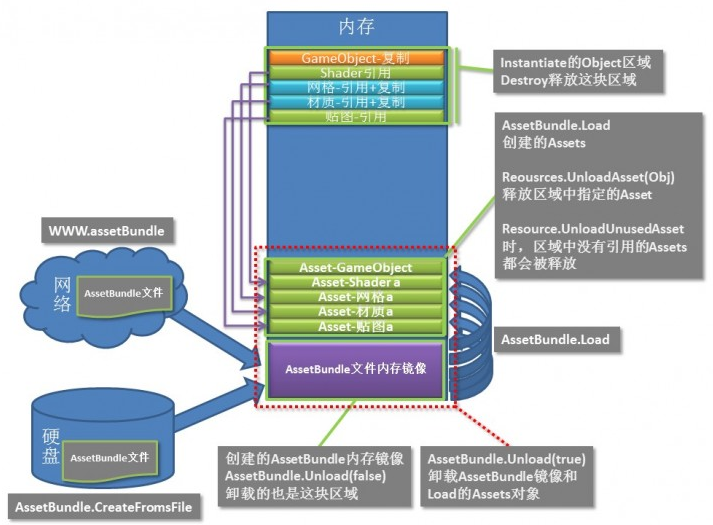
一个prefab从assetbundle里load出来，里面可能包括：GameObject Transform Mesh Texture Material Shader Script和各种其他的Assets，你Instaniate一个Prefab，是一个对Asset进行Clone（复制）+引用结合的过程，GameObject transform是clone新生成的，其他mesh texture material shader等，这些是纯引用关系，包括：Texture和TerrainData，还有引用和复制同时存在的，包括：Mesh material phycisMaterial。引用的Asset对象不会被复制，只是简单的指针指向已经load的asset对象。

专门要提的一个特殊的东西：Script Asset，看起来很奇怪，Unity里每个Script都是一个封闭的Class定义而已，并没有调用代码，光Class的定义脚本是不会工作的。其实Unity引用就是那个调用代码，Clone一个script asset等于new一个class实例，实例才会完成工作。把它挂到Unity主线程的调用链里去，Class实例里的OnUpdate OnStart等才会被执行。多个物体挂同一个脚本，其实就是在多个物体上挂了那个脚本类的多个实例而已。在new Class这个剁成中，数据区是复制的，代码区是共享的，算是一种特殊的复制+引用

你可以再Instaniate一个同样的Prefab，还是这套mesh texture material shader，这时候会有新的GameObject，但是不会创建新的应用对象比如Texture

**所以，load出来的Assets其实就是一个数据源，用于生成新对象或者被引用，生成的过程可能是（复制clone）也可能是引用（指针）**

当你Destroy一个实例时，只是释放了那些clone对象，并不会释放应用对象和clone的数据源对象，Destroy并不知道是否还有别的object在引用那对象。等到没有任何游戏场景物体在用这些Assets以后，这些Assets就成了没有应用的游离数据块了，是UnusedAssets了，这时候就可以通过Resources.UnloadUnusedAssets来释放，Destroy不能完成这个任务，AssetBundle.Unload(false)也不行，AssetBundle.Unload(true)可以但不安全，除非你很清楚没有任何对象在用这些Assets了



内存管理

创建时：

先建立一个AssetBundle，无论是从www还是memory

用assetBundle.Load加载需要的Asset

加载完之后立即调用AssetBundle.Unload(false)，释放AssetBundle文件本身的内存镜像，但不销毁加载的Asset对象（这样就不用保存AssetBundle的应用并且可以立即释放一部分内存）

释放时：

如果有Instantiate的对象，用Destroy进行销毁

在合适的地方调用Resources.UnloadUnusedAssets，释放已经没有应用的Asset

**系统在加载新场景时，所有的内存对象都会被自动销毁，包括用AssetBundle.Load加载的对象和Instaniate克隆的，但是不包括AssetBundle文件自身的内存镜像，那个必须用Unload来释放，用.net的术语，这种数据缓存是非托管的**

总结一下各种加载和初始化的用法:  
AssetBundle.CreateFrom.....：创建一个AssetBundle内存镜像，注意同一个assetBundle文件在没有Unload之前不能再次被使用  
WWW.AssetBundle：同上，当然要先new一个再 yield return 然后才能使用  
AssetBundle.Load(name)：从AssetBundle读取一个指定名称的Asset并生成Asset内存对象，**如果多次Load同名对象，除第一次外都只会返回已经生成的Asset对象，也就是说多次Load一个Asset并不会生成多个副本（singleton）。**Resources.Load(path&name)：同上,只是从默认的位置加载。  
Instantiate（object)：Clone一个object的完整结构，包括其所有Component和子物体（详见官方文档）,浅Copy，并不复制所有引用类型。有个特别用法，虽然很少这样用，其实可以用Instantiate来完整的拷贝一个引用类型的Asset,比如Texture等，要拷贝的Texture必须类型设置为Read/Write able。  
  
总结一下各种释放  
Destroy:主要用于销毁克隆对象，也可以用于场景内的静态物体，不会自动释放该对象的所有引用。虽然也可以用于Asset,但是概念不一样要小心，如果用于销毁从文件加载的Asset对象会销毁相应的资源文件！但是如果销毁的Asset是Copy的或者用脚本动态生成的，只会销毁内存对象。  
AssetBundle.Unload(false):释放AssetBundle文件内存镜像  
AssetBundle.Unload(true):释放AssetBundle文件内存镜像同时销毁所有已经Load的Assets内存对象  
Reources.UnloadAsset(Object):显式的释放已加载的Asset对象，只能卸载磁盘文件加载的Asset对象  
Resources.UnloadUnusedAssets:用于释放所有没有引用的Asset对象  
GC.Collect()强制垃圾收集器立即释放内存 Unity的GC功能不算好，没把握的时候就强制调用一下

**用AssetBundle加载asset一样可以用Resources.UnloadUnusedAssets卸载，但必须先AssetBundle.Unload，才会被识别为无用的asset**

Unity动态加载prefab方式的差异：

其实在unity中存在三种加载prefab的方式

1. 静态引用，建立一个public的变量，在inspector里把prefab拉上去，用的时候instantiate
2. Resource.Load，Load以后instaniate
3. AssetBundle.Load，Load以后instantiate

这三种方式有细节差异，前两种方式，引用对象texture是在instantiate时加载，而assetbundle.load会把prefab的全部assets都加载，instantiate时只是生成clone，多有前两种方式，除非提前加载相关引用对象，否则第一次instantiate时会包含加载引用类assets的操作，导致第一次加载的时间有延时。具体看<https://blog.csdn.net/u011866450/article/details/51220764>

使用Resources.Load的时候在第一次Instantiate之前，相应的Asset对象还没有被创建，直到第一次Instantiate时才会真正去读取文件创建这些Assets。它的目的是实现一种OnDemand的使用方式，到该资源真正使用时才会去创建这些资源。  
而使用AssetBundle.Load方法时，会直接将资源文件读取出来创建这些Assets，因此第一次Instantiate的代价会相对较小。

几种AssetBundle创建方式的差异：

CreateFromFile：这种方式不会把整个硬盘的AssetBundle文件都加载到内存来，而是类似建立一个文件操作句柄和缓冲区，需要时才实时Load，，所以这种加载方式是最节省资源的，基本上AssetBundle本身不占什么内存，只需要Asset对象的内存，可惜只能在PC/Mac Standalone程序重视会用

CreateFromMemory和[www.assetBundle](http://www.assetBundle)：这两种方式AssetBundle文件会整个镜像于内存中，理论文件多大就需要多大的内存，之后load时还要占用额外的内存去生成Asset对象

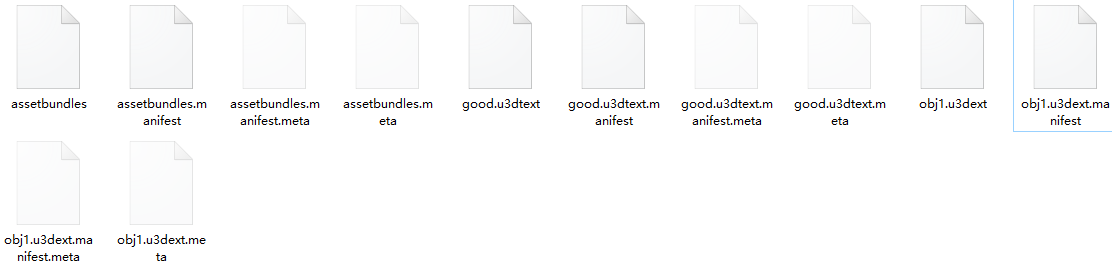
**什么时候才是UnusedAssets?**  
看一个例子：  
Object obj = Resources.Load("MyPrefab");  
GameObject instance = Instantiate(obj) as GameObject;  
.........  
Destroy(instance);  
创建随后销毁了一个Prefab实例，这时候 MyPrefab已经没有被实际的物体引用了，但如果这时：  
Resources.UnloadUnusedAssets();  
内存并没有被释放，原因：MyPrefab还被这个变量obj所引用  
这时候：  
obj  = null;  
Resources.UnloadUnusedAssets();  
这样才能真正释放Assets对象  
所以：UnusedAssets不但要没有被实际物体引用，也要没有被生命周期内的变量所引用，才可以理解为 Unused(引用计数为0)  
所以所以：如果你用个全局变量保存你Load的Assets，又没有显式的设为null，那在这个变量失效前你无论如何UnloadUnusedAssets也释放不了那些Assets的。如果你这些Assets又不是从磁盘加载的，那除了UnloadUnusedAssets或者加载新场景以外没有其他方式可以卸载之。

**Unity5的打包**

资源下方有两个下拉框，左边用于指定AssetBundle的名字，后边的用于指定AssetBundle Variants的名字

BuildPipeline.BuildAssetBundles(Application.dataPath + "/assetbundles", BuildAssetBundleOptions.None, EditorUserBuildSettings.activeBuildTarget);

这样会在assetbundles文件夹中看到打包的文件



其中会产生一个与文件夹名字相同的manifest文件，其他每个资源都产生一个manifest文件

实际上我们通常可以在打包的时候指定AssetBundle的名字以及资源路径

[MenuItem("Assets/Build AssetBundle To One Bundle")]

public static void BuildAssetBundleToOneBundle()

{

string[] folderName = new string[] { "Prefabs", "Textures", "Shaders" };

List<AssetBundleBuild> builds = new List<AssetBundleBuild>();

foreach (var name0 in folderName)

{

string path = Path.Combine(Application.dataPath, name0);

if (Directory.Exists(path))

{

DirectoryInfo info = new DirectoryInfo(path);

FileInfo[] files = info.GetFiles("\*", SearchOption.AllDirectories);

for (int i = 0, size = files.Length; i < size; i++)

{

if (files[i].Name.EndsWith(".prefab") || files[i].Name.EndsWith("JPG") || files[i].Name.EndsWith("png") ||

files[i].Name.EndsWith(".shader"))

{

string name = files[i].Name;

string variant = "";

string path0 = files[i].FullName.Substring(files[i].FullName.IndexOf("Assets"));

string[] assetNames = new string[] { path0 };

builds.Add(CreateAssetBundleBuild(name, variant, assetNames));

}

}

}

}

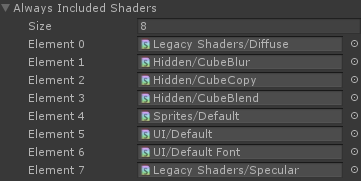
BuildAssetBundleInternal(outputPath, builds.ToArray(), BuildAssetBundleOptions.ChunkBasedCompression);

}

对于自己写的shader，如果有FallBack “Specular”会加入一个模型的Specular Shader，这个Shader比较大，如果是Diffuse就会很小，实际上是Diffuse默认就被加进去了，如果我们想Specular不进入包体，则也可以将Specular提前加载进去

将内置的Shader打包可以使用：

依次点击菜单栏的 Edit -> Project Settings -> Graphics -> Always Included Shaders



在查看AssetBundle的manifest文件时候可以看到，依赖项是不包含内置Shader的：

Dependencies:

- D:/profiles/unity/test\_projects/Shaders/Assets/StreamingAssets/grid.png.u3dext

- D:/profiles/unity/test\_projects/Shaders/Assets/StreamingAssets/singletexture.shader.u3dext

获取所有assetBundle

[MenuItem("Assets/GetAllAssetBundle")]

public static void GetAllAssetBundle()

{

string[] names = AssetDatabase.GetAllAssetBundleNames();

for (int i = 0, size = names.Length; i < size; i++)

{

Debug.Log(string.Format("i = {0}, AssetBundle name = {1}", i, names[i]));

string[] paths = AssetDatabase.GetAssetPathsFromAssetBundle(names[i]);

foreach (var path in paths)

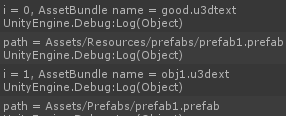
{

Debug.Log("path = " + path);

}

}

}



**AssetBundle的压缩**

你可以不对AssetBundles进行压缩，这样虽然文件很大，但是一旦下载好之后访问是最快的，通常都会进行压缩，并且Unity的AssetBundle支持LZMA和LZ4两种格式的压缩

LZMA格式：这是unity打包成AssetBundle时的默认格式，会将序列化数据压缩成一个LZMA流，使用时需要整体解包。优点是打包后体积小，缺点是解包时间长导致加载时间长。

LZ4格式：相对于LZMA会生成更大的压缩文件，但优点是使用时不需要整体解压。LZ4是一种基于chunk的算法，加载对象时只有相应的chunk会被解压，该格式需要unity5.3以上的版本，以前的版本并不支持，BuildAssetBundleOptions选择ChunkBasedCompression

**什么是AssetBundle？**

AssetBundle只是一种使用LZMA压缩方式压缩的资源文件，至于它的后缀名是什么，一点关系都没有，可以自己定。

AssetBundle打包的时，可以指定一个mainAsset，那么加载完之后就可以通过AssetBundle.mainAsset来获取到了，也可以不指定mainAsset，直接打包一堆内容进去，然后加载通过AssetBundle.LoadAsset指定名字读取出来。在资源之间，存在着依赖关系，你可以把资源拆分得很细，比如一个模型，由网格模型、材质、贴图构成，可以将他们分开，各自打包成压缩文件。当Unity需要加载使用时，把该模型的所有依赖的小资源都加载起来，然后根据依赖关系组装就变回我们看到的资源了。

**打包AssetBundle**

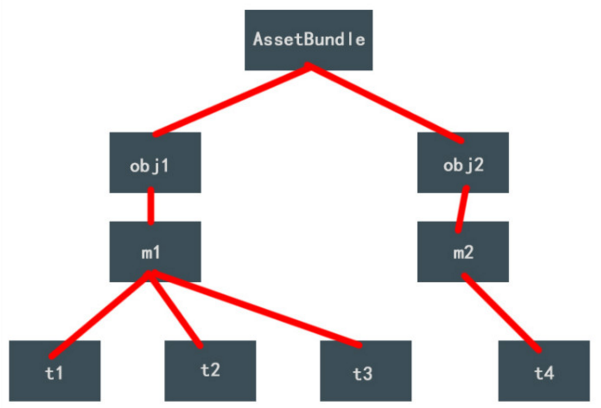
Unity5.x大幅度简化了AssetBundle的打包过程，理论上只需要调用一个API既可以完成打包, BuildPipeline.BuildAssetBundles

但是资源之间的依赖关系，避免资源重复打包的问题还是要自己解决

Unity5.x中会将设置AssetBundleName相同的资源打包到一起，所以我们的打包过程其实就是把需要打包到一起的资源设置成相同的AssetBundleName，为每个资源设置AssetBundleName，最后调用打包API即可完成打包

**AssetBundle的依赖结构**

要说明依赖关系，还需要使用一个例子，一个模型，分为了网络模型、材质、贴图。那么它们是怎样依赖的呢？然后在unity5的打包里面，它们是怎样表现出依赖关系的呢？举个例子：



在上图中obj代表被设置为prefab的GameObject，m代表材质，t代表纹理

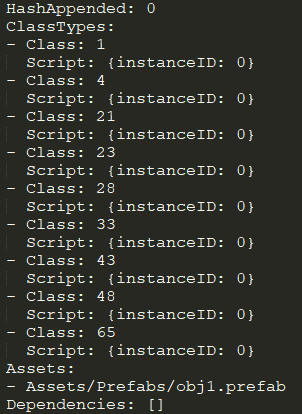
首先，我们只将obj1和obj2设置为asssetbundle：

**Obj1：**

Assetbundle文件资源：



Manifest文件：



我们会发现此时obj1的assetbundle没有依赖，并且它会将所有的依赖资源全部进包，包括依赖的三涨纹理t1、t2、t3

此时的obj1.ab大小为323k

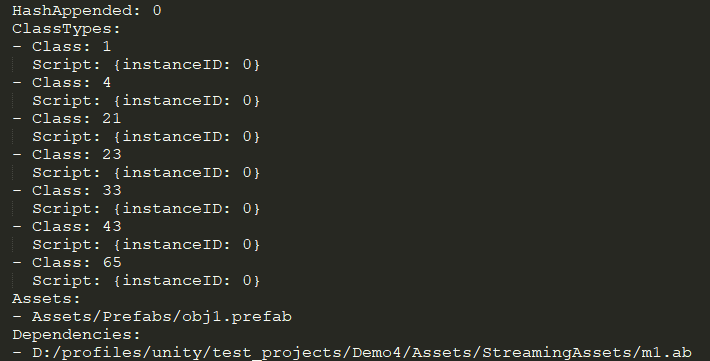
接下来我们将m1打包为m1.ab文件

**Obj1：**

Assetbundle文件资源：



Manifest文件：



此时obj1.ab为5.86k，而m1.ab为319k，通过unitystudio可以看到此时的t1、t2、t3都在m1.ab中

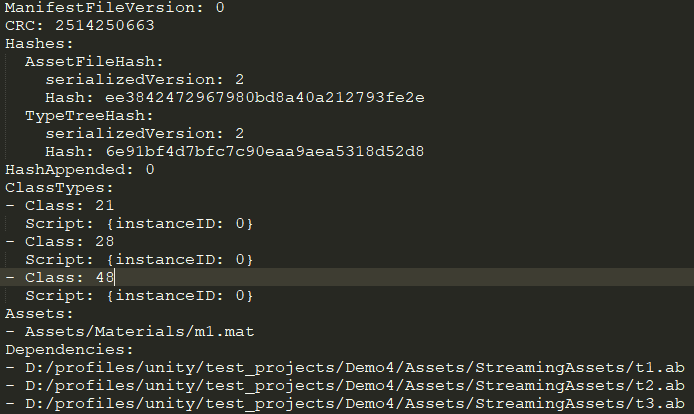
接下来，我们将t1、t2、t3、t4打包为t1.ab、t2.ab、t3.ab、t4.ab

M1:

AssetBundle资源文件：



Manifest文件：



如果现在将t2不导出，发现t2这张纹理直接进入了m1.ab

个人总结：如果所需要的资源在其他的AssetBundle中，则直接依赖，否则则直接将资源打进AssetBundle中

实际上在改变资源后，只有总的AssetBundle.manifest，还有依赖于它本身的manifest会修改，其他不依赖的部分是不需要重新打包的，这就是AssetBundle的链式结构和增量打包了。

还有一点需要注意的是，除了manifest文件以外，还有一个没有后缀名称的AssetBundle文件。这个文件其实才是包含了很多有依赖关系的总的依赖关系配置文件，刚才我们打开的manifest，都只是用来做本地依赖关系和增量打包的时候用的，我们加载AssetBundle的时候，完全不需要加载那些manifest文件，只需要那个没有后缀名称的AssetBundle文件（具体名字和导出的文件夹有关）就行了，它代表的是该项目的所有AssetBundle的依赖关系。所以，刚才我们看到manifest里面用的都是本地的绝对路径，那是针对你本地打包时用的，和加载无关。

总的AssetBundle资源结构：



我们可以通过AssetImporter来修改导出的AssetBundle名字

**加载AssetBundle的步骤**

通过上面导出AssetBundle的说明，估计现在想要把它加载起来就变得简单了

首先需要明白一个规则，资源的依赖关系组装是unity本身会自动完成的。比如一个资源A，它是依赖于资源B和资源C的，那么如果我们需要加载资源A进来并正确的显示出来，我们必须先把资源B和资源C加载进来，然后再加载资源A。当资源A加载进来后，发现内存里面有资源B和资源C了，它会自动的组装起来。

如果不先加载依赖项，直接加载obj1

void Start()

{

#if UNITY\_ANDROID

STREAMING\_PATH\_SYNC = Application.dataPath + "!assets";

#else

STREAMING\_PATH\_SYNC = Application.streamingAssetsPath;

#endif

Debug.Log("STREAMING\_PATH\_SYNC = " + STREAMING\_PATH\_SYNC);

AssetBundle assetBundle = AssetBundle.LoadFromFile(STREAMING\_PATH\_SYNC + "/obj1.ab");

GameObject o1 = assetBundle.LoadAsset<GameObject>("obj1");

GameObject obj1 = Instantiate<GameObject>(o1);

Debug.Log("o1 = " + o1);

}



那么再看看正确加载的步骤了：

void Start()

{

#if UNITY\_ANDROID

STREAMING\_PATH\_SYNC = Application.dataPath + "!assets";

#else

STREAMING\_PATH\_SYNC = Application.streamingAssetsPath;

#endif

AssetBundle[] assetBundles = LoadAssetBundle(STREAMING\_PATH\_SYNC, "StreamingAssets", "obj1.ab");

AssetBundle assetBundle = assetBundles[assetBundles.Length - 1];

UnloadAssetBundle(assetBundles); // 如果不加，则会提示重复加载的错误

AssetBundle[] assetBundles0 = LoadAssetBundle(STREAMING\_PATH\_SYNC, "StreamingAssets", "obj1.ab");

AssetBundle assetBundle1 = assetBundles0[assetBundles.Length - 1];

GameObject obj1 = assetBundle1.LoadAsset<GameObject>("obj1");

Instantiate<GameObject>(obj1);

}

AssetBundle[] LoadAssetBundle(string path, string totalAssetBundleName, string assetBundleName)

{

// 获取总的AssetBundle配置文件

List<AssetBundle> assetBundleList = new List<AssetBundle>();

AssetBundle totalAssetBundle = AssetBundle.LoadFromFile(Path.Combine(path, totalAssetBundleName));

// 这里的AssetBundleManifest是固定名字

AssetBundleManifest menifest = totalAssetBundle.LoadAsset<AssetBundleManifest>("AssetBundleManifest");

assetBundleList.Add(totalAssetBundle);

// 依赖AssetBundle

string[] assetBundleNames = menifest.GetAllDependencies(assetBundleName);

for (int i = 0, size = assetBundleNames.Length; i < size; i++)

{

Debug.Log("dependeny i = " + i + ", name = " + assetBundleNames[i]);

AssetBundle assetBundle = AssetBundle.LoadFromFile(Path.Combine(path, assetBundleNames[i]));

if (assetBundle) assetBundleList.Add(assetBundle);

}

AssetBundle resultAssetBundle = AssetBundle.LoadFromFile(Path.Combine(path, assetBundleName));

assetBundleList.Add(resultAssetBundle);

return assetBundleList.ToArray();

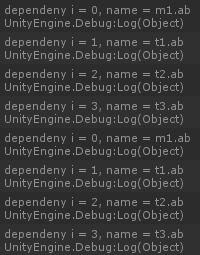
}

void UnloadAssetBundle(AssetBundle[] list)

{

for (int i = 0, size = list.Length; i < size; i++) list[i].Unload(false);

}





总结：

对于多次引用的资源，必须明确出现在某个AssetBundle的资源列表中，Unity才能形成依赖关系，不重复打包。而且这些Assetbundle必须存在于同一个AssetBundleBuild[]列表中，一次调用BuildPipeline.BunldAssetBundles()，分批调用build接口也不能形成依赖，同时建议公共的资源单独抽出来做一个AssetBundle，而不是合并到某个依赖项中。

备注：同理，使用编辑器导出，多次引用的资源也要明确指定其AssetBundle名称，不指定名称仅通过依赖会导致重复打包。

**Shader依赖打包加载**

为了使AssetBundle不重复打包shader，按照上面的思路，很自然地想到将shader单独打包，但很快就会遇到两个问题：Shader加载和内置shader丢失

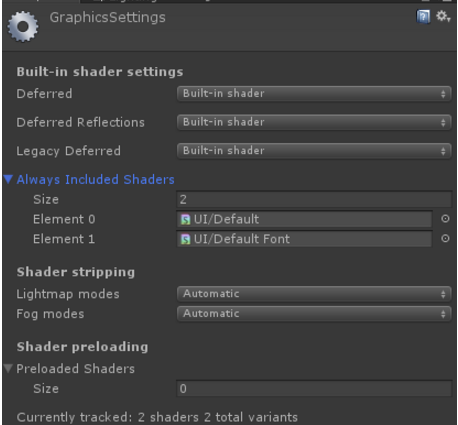
Shader最好在启动时加载，加载后调用Shader.WarmupAllShaders()编译，并永不释放，代码如下：

AssetBundle ab = AssetBundle.Load("shader")

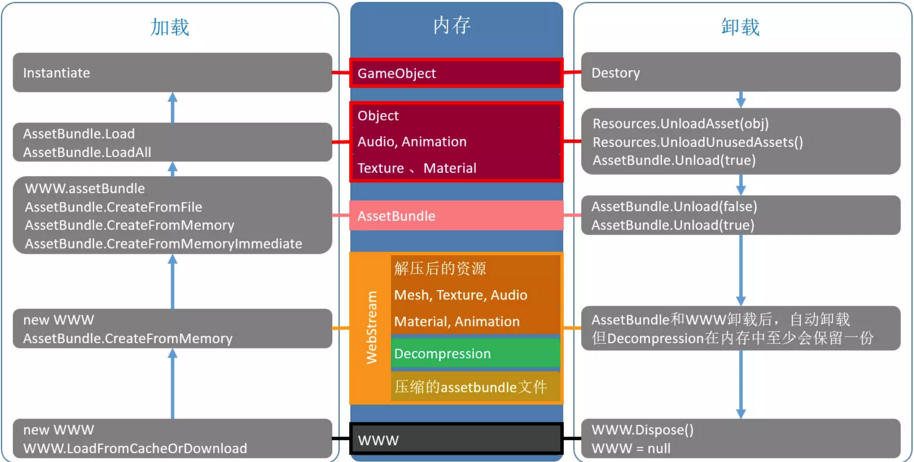
ab.LoadAllAssets();

Shader.WarmupAllShaders();　　// 如果不想启动时预编译，也可以不加这句

对于Unity内置的shader，例如UGUI使用的shader，我们无法将其达到AssetBundle并通过上面的方式加载。这里需要将其添加到Always Included Shaders。在Editor->Project Setting->Graphics中可以找到。Unity将在启动时候解析这些Shader，避免在此添加大量shader



另一种方式是从官网下载Unity内置shader，加入到依赖包中即可



我们从AssetBundle中加载资源一般会经过三个步骤：

1 www、LoadFromFile、LoadFromMemory等接口加载AssetBundle本身

2 通过AssetBundle.LoadAsset等接口从AssetBundle中加载资源

3 对于GameObject类资源，还需要通过GameObject.Instaniate创建clone、

这几个步骤都会产生内存分配，并且分别对应了图中不同颜色的部分。而图的左边已经写出了产生内存分配的接口，右边则标明了释放内存的方法：

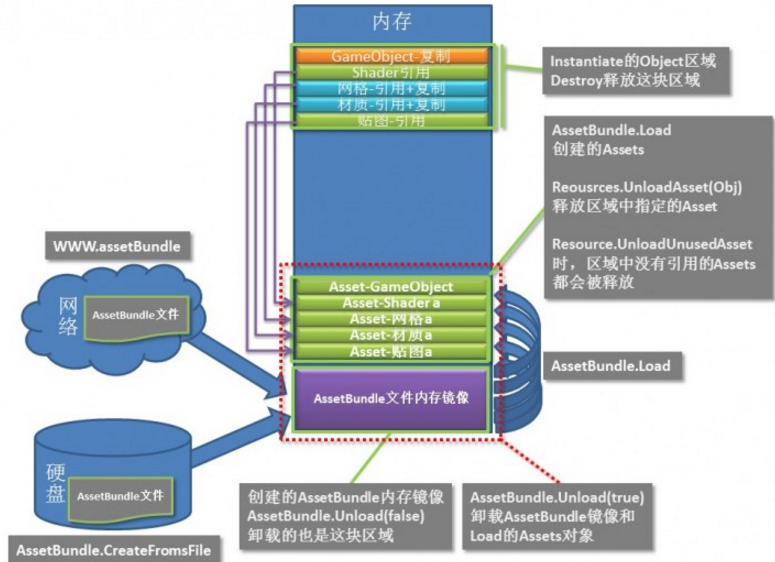
**WebStream中已经包含原始AssetBundle的数据**

**黑色部分：**www类本身占用内存，通过www这个加载AssetBundle才会有这部分的内存，www对象保留了一份对WebStream（粉色部分）的引用。使用www=null或者[www.dispose](http://www.dispose)释放。其中[www.dispose](http://www.dispose)会立即释放，而www=null会等待垃圾回收。释放www后WebStream的应用计数会相应减一

**橙色部分：**官方称为WebStream数据，是数据真正的储存区域。当AssetBundle被加载进来后，这部分内存就会分配了。它包含了3个内容：压缩后的AssetBundle本身，解压后的资源以及一个解压缓冲区（图绿）。无论www（黑色部分）还是后面会提到的AssetBundle（粉色部分），都只是有一个结构指向WebStream数据，从而能对外部提供操作真正资源数据的方法。而当www对象和AssetBundle对象释放时，WebStream数据的引用计数也会相应减一，当WebStream数据应用计数为0时，系统会自动释放。但为了不频繁地开辟和销毁解压Buffer，其中绿色Decompression解压缓冲区Unity会至少保留一份。例如，同时加载3个AssetBundle时，系统会生成3个Decompression Buffer，当解压完成后，系统会销毁两个。

**粉色部分：**AssetBundle对象，引用了橙色部分WebStream数据部分，并提供了从WebStream数据中加载资源的接口。通过AssetBundle.Unload释放。如果调用AssetBundle.Unload(flase)，将释放Assetbundle对象本身，其对WebStream引用也将较少，从而可能引起WebStream释放，我们也就无法再通过接口或者依赖关系从该AssetBundle加载资源。但已加载的资源还可以正常使用。如果调用的是AssetBundle.Unload(true)，不仅会释放WebStream部分，所有被加载出来的资源将被释放，无论true或false，AssetBundle.Unload都将销毁AssetBundle，销毁后调用该AssetBundle对象的任何方法都不会生效或产生报错，也就是说这个接口只能被调用一次，不能先调用unload(false)再调用unload(true)

**红色部分：**看下面这张图



这里的紫色区域“AssetBundle文件内存镜像”就是上面的WebStream和www（黑橙）部分，虚线里的资源就是从AssetBundle加载出来的原始资源了，也属于WebStream范围，我们不再讨论。

而绿色实线内的资源（对应第一张图红色部分），也就是我们通过Instantiate创建的GameObject所包含的资源。这些包含的资源又根据类型，与AssetBundle原始资源（WebStream资源部分）有不同的关系。有些入Texture、shader资源，我们通常甚至是使用，并不会对其作出改动，所以仅仅是引用关系；而每个GameObject都是特殊的，所有是完全复制了一份；至于Mesh和Material，则是应用+复制的关系

建议：

* www加载完毕后调用www.dispose()或www = null。
* 对于shader、通用图集等常驻且需要保留依赖关系的资源，在合适时机加载进来，不调用AssetBundle.Unload()接口。
* 其他根据不同使用情况，选择AssetBundle.Unload()策略。
  + 对于界面等存在明确生命周期，又可能动态加载的资源，在生命周期结束后调用AssetBundle.Unload(true)，将全部资源一起释放。
  + 对于加载完后不再需要主动和被动依赖加载的资源，在加载完成后调用AssetBundel.Unload(false)，立即释放掉AssetBundle资源，当资源使用完毕，代用Resources.UnloadAsset()或Resources.UnloadUnusedAssets()释放资源内存。
* Resources.UnloadUnusedAssets()有较大消耗，尽量减少调用次数。
* 不能调用AssetBundle.Unload(false)后再调用AssetBundle.Unload(true).
* 不再使用的GameObject直接Destroy即可。

