Unity 资源系统

序

用了这么久的 AssetBundle，知道它的基本原理吗？知道使用它的正确姿势吗？知道 Unity 的资源系统吗？ 本教程对正在写 Unity 项目的资源管理的同学应该有一定的参考价值。关于 AssetBundle 的加载和卸载是每一个 Unity 项目的客户端同学都应该深入了解的一个内容。本文翻译的是官方最佳实践系列的一篇关于资源的指南。如果有翻译和理解错误，还望不吝指出，会接受批评并加以改正

本译作采用署名-非商业性使用-相同方式共享 4.0 国际知识共享协议(CC BY-NC-SA 4.0) (Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0))

译文原址：《 Assets, Resources and AssetBundles》

译者邮箱： lingzelove2008@163.com

凌泽

2018.10.01

第一章：前言

这是一个系列文章, 它对 Unity 引擎的资源序列化和资源管理进行深层次的探讨；同时详细的讲解了 AssetBundle 系统的基本原理以及使用它的最佳实践(本文所有内容均已用 Unity2017.3 验证)

本指南分为四部分：

1. **资源、对象以及序列化** 探讨了 Unity 资源序列化以及对象之间相互引用的详细过程

2. **Resources 文件夹** 讨论了 Unity 内置的 Ressources API

3. **AssetBundle 基本原理** 讲解了 AssetBundle 的基本原理、AssetBundle 的加载以及从 AssetBundle 内加载对象

4. **AssetBundle 最佳实践** 讨论了很多 AssetBundle 相关的主题，包括将资源打包成 AssetBundle 的策略、管理已加载的资源；同时展示了开发者在使用 AssetBundle 的时候经常会犯的一些错误和认知误区

注意：指南中的对象（Objects）和资源（Assets）这两个术语与 Unity 提供的 API 中的 Objects、Assets 并不是同一个概念

指南中对象(Objects)所表示的数据，其实在 Unity 的一些公共 API 中叫做 Assets，比如：通过 AssetsBundle.LoadAsset 所获取的 Assets；以及 Resources.UnloadUnusedAssets 所释放的 Assets

指南中资源(Assets)所表示的内容很少在 API 中使用，当它们被使用时，通常只在与构建相关的代码中，比如：AssetDatabase 和 BuildPipeline；在这种情况下，我们一般称它们为资源文件

第二章：资源、对象和序列化

本章将从 Unity 编辑器环境 和 运行时环境 两个角度出发，探讨以下三个方面的内容：

- 资源序列化的内部实现
- Unity 如何维护对象之间的强引用关系
- 资源和对象之间的技术区别

这章是理解 Unity 加载和卸载资源的基础，良好的资源管理对应用的加载时长以及内存占用都至关重要

2.1 理解资源(Assets)和对象(Objects)

在理解 Untiy 是如何高效管理数据之前，要知道 Unity 是如何识别和序列化数据。这个关键点就是要正确区分资源(Assets)和对象(Objects)之间的区别

2.1.1 资源(Asset)

资源其实就是硬盘里的文件，存储在 Unity 工程的 Assets 文件夹内。例如:文本(txt)、纹理(Texture)、材质(Material) 等都是常见的资源。有些资源的数据格式是 Unity 原生支持的，比如：材质(Material)；有些资源则需要转换成 Unity 原生支持的数据格式才能被使用，比如:FBX 文件

2.1.2 对象(UnityEngine.Object)

UnityEngine.Object(以大写 O 开头的 Object), 则是一组已经序列化的数据的集合，表示某个资源的具体实例，它可以是 Unity 引擎可使用的任何类型资源。例如：网格(mesh)、精灵(Sprite)、音频剪辑(AudioClip)或者动画剪辑(AnimationClip)这些对象都是 UnityEngines.Object 的子类

2.1.3 特殊类型

几乎所有的对象类型都是 Unity 内置(built-in)的，但有两种类型比较特殊：

- ScriptableObject 允许开发者自定义数据类型。这些类型能够被 Unity 序列化和反序列化，并可以在编辑器的 Inspector 窗口中进行编辑操作
- MonoBehaviour 它提供链接到 MonoScript 的封装 ；MonoScript 是 Unity 的内部数据类型，它保存了指向程序集和命名空间中的具体脚本类的引用，MonoScript 本身不包含任何实际可执行的代码（下文有详细讲解）

2.1.4 资源与对象的关系

资源(Assets)和对象(Objects)之间是一对多关系；一个资源可能包含一个或多个对象

2.2 对象之间的引用

所有对象都可以引用其他的对象，被引用的对象可以存在于同一个资源文件中, 也可以由其他不同资源文件导入。例如：一个材质球对象通常引用一个或者多个纹理对象，这些纹理对象通常是从一个或者多个纹理资源文件导入的(比如 .png 、. jpg 等)

当资源序列化的时候，会引用两个独立的概念：File GUID 和 Local ID，File GUID 是该资源文件的唯一表示； Local ID 是用来区分一个资源中的多个对象（局部唯一）

2.2.1 File GUID

File GUID 存储在.meta 文件中，Unity 会在首次导入资源文件时生成和资源文件同名，后缀为.meta 的文件，并和资源文件存储在相同的目录中

File GUID 可以使用文本编辑器来查看，步骤如下：

- 依次打开菜单 Edit–Projects Settings–Editor 将 Version Control 设为：Visible Meta Files； Asset Serialization Mode 设为：Force Text
- 新建一个材质 New Material.mat，然后用文件编辑器打开同目录下的 New Material.mat.meta 文件
  内容大概如下：

fileFormatVersion: 2
guid: a04b7559b17973d468087b3a036011fc
timeCreated: 1538226064
–other data–

可以看到，有个 guid 字段. 它就是该资源文件(New Material.mat)在 Unity 资源系统内的唯一标识

2.2.2 Local ID

现在，我们直接用文本编辑器打开然后用文本编辑器打开 New Material.mat，内容大概如下：

%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
— !u!21 &2100000
Material:
serializedVersion: 6
–other data–

&符号后面的数据，就是 Material 的 Local ID(注意：因为 New Material.mat 文件只包含自己一个对象，所以只能看到一个 Local ID,你可以找包含很多对象的资源查看，比如：prefab 文件，场景文件等)

2.3 为什么需要 File GUID 和 Local ID？

Unity 为什么要采用 File GUID 和 Local ID 这套系统呢？答案是为了提供一个健壮的，灵活的与使用平台无关的资源机制

File GUID 提供了一个资源文件的特定位置的抽象实现，只要 File GUID 与某个文件相关联，那么该文件存储在哪个位置就变得无关紧要了(你可以测试一下，移动资源文件到其他文件夹，他的 GUID 是不会改变的，当然，是指在 Unity 编辑器内移动资源文件)

但是为什么又需要一个 Local ID 呢？因为一个资源文件可能包含多个对象,所以就需要一个 Local ID 来明确区分每个不同的对象

如果一个资源文件与对应的 File GUID 之间的关联丢失了，那么对该资源包含的对象的引用也将丢失。这就是为什么.meta 文件必须保存相同的文件名，并保存在相同的文件夹中。注意，Unity 会自动重新生成已删除或被移动到其他目录的.meta 文件

那么 Unity 是怎么将 File GUID 与文件实际存储路径相互绑定的呢？Unity 编辑器保存了资源文件路径到对应 File GUID 的映射，无论何时加载或导入资源，Unity 编辑器都会记录并更新对应的映射条目

如果 Unity 编辑器发现.meta 文件丢失，而资源的路径没有改变时，编辑器就可以确保该资源沿用相同的 File GUID，而且会重新生成一个与之对应的.meta 文件

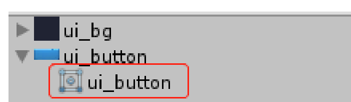
如果在 Unity 编辑器关闭后，移动资源文件到其它目录而不把它对应的.meta 文件也移动到相应的目录时，那么所有对该资源(或者该资源内的对象)的引用就会丢失。 所以，在 Unity 编辑器关闭后，不要随意移动 Assets 目录下的文件，否则容易造成对象引用丢失

2.4 资源的导入

2.4.1 资源导入器

正如 2.1.1 节所提到的，不能被 Unity 原生直接支持的资源类型必须经过导入才能被使用，这是通过 Unity 自带的导入器(importer)完成的 。 导入器通常是自动调用的，但 Unity 也提供了一些公开的导入接口。例如：当我们导入单个纹理资源(比如：png 文件)，TextureImporter API 提供了导入时可以更改纹理属性的方法(比如：纹理压缩格式，压缩类型等)

导入之后的结果是一系列对象(UnityEngine.Object)的整合体，这在 Unity 编辑器中表现为，父资源下可能有多个子对象；例如 SpriteAtlas，其下属会有多个嵌套的 Sprite 对象，这些子对象都公用一个 File GUID,因为他们的资源数据存储在一个资源文件中,他们通过 Local ID 来区分彼此。示例如下，红色框标记的是导入之后生成子对象



2.4.2 Library 文件夹

导入之后会将源文件转换成 Unity 编辑器中选择的目标平台可序列化的资源文件，导入过程可能包括一些非常耗时的操作，比如纹理压缩。所有导入后的资源被缓存在 Library 文件夹中，这样下次再打开 Unity 就不需要再重新导入了

具体地说，导入之后的结果存储在一个文件夹中，这个文件夹是以资源的 File GUID 的前两个数字命名的；这个文件夹存储在 library/metadata/文件夹中。资源中的单个对象被序列化成一个单独的二进制文件，它的名称与资源的 File GUID 相同

实际上即使是 Unity 原生支持的资源文件，也会先导入并将导入结果存储在对应文件中，只不过原生资源不需要进行转换或者序列化

2.5 序列化和实例化

虽然 File GUID 和 Local ID 搭配可以很健壮的维护资源之间的关系；但 GUID 之间的比较效率很低（译者注：因为它是个很长的字符串，字符串与字符串之间的比较操作，效率很低），所以在运行的时候我们需要一个效率更高的解决方案；Unity 内部维护一个缓存（事实上, 这个缓存叫做: PersistentManager），它会将 File GUID 和 Local ID 转换成简单的整数，这个整数在运行的过程中是唯一的，这个整数称为：Instance ID ；它会简单地以单调递增的方式分配给缓存中新注册的对象

Unity 内部的缓存机制，大概就是 Instance ID 作为缓存字典的 Key，它对应的 Value 结构有两部分，第一部分是已经实例化的对象的内存地址，第二部分则是这个对象来源的 File GUID 和 Local ID。 所以根据 Instance ID 如果实例化对象存在，则可以快速拿到已经实例化的对象（因为知道它的内存地址了）。如果实例化对象不存在，那么可以根据 File GUID 和 Local ID 去加载并实例化对象

在启动应用时，项目内置对象（比如场景中使用的对象）的数据以及在 Resources 文件夹中的对象的数据将被初始化到 Instance ID 缓存中。当新的资源在运行中导入时(比如:脚本中创建 Texture2D 对象：var myTexture =

new Texture2D(1024, 768))，以及从 AssetBundle 中加载对象时，新增的条目被添加到缓存中。Instance ID 仅在失效(已过时)才会被从缓存中移除。当一个 AssetBundle 被卸载时，除了会导致对应的 Instance ID 被认为已经过时之外，Instance ID 和 File GUID 以及 Local ID 之间的映射数据也会被从缓存中删除。如果 AssetBundle 被重新加载，那么从该 AssetBundle 中加载的每一个对象都会新分配一个 Instance ID

注意在某些平台上，有一些系统事件会使得对象在内存中被强行卸载。例如 iOS 平台上，当一个 app 处于挂起状态时，图形资源就会从显存中卸载。如果这些对象都来自于一个已被卸载的 AssetBundle 时，Unity 将无法再次从源数据处加载这些对象。任何已有的对这些对象的引用都会失效。在 Unity 应用中导致的后果就是出现网格不可见(丢失),或模型的纹理和材质呈现为洋红色（Shader 丢失）

提示：在运行时，对上述控制流程的描述并非完全准确。构建 Unity 工程时，File GUID 和 Local ID，确切地说会被映射到一种更简单的格式中。不过概念还是大致相同的，在考虑“运行时”的时候以 File GUID 和 Local ID 的工作方式为思路还是有一些参考意义的(这也是在运行时资源的 File GUID 无法被调取的原因，因为它已经被映射到了一种新的方式中)

2.6 MonoScripts

理解 MonoBehaviour 组件很重要的一点是知道它包含了一个对 MonoScript 的引用，而 MonoScript 包含了用于定位到一个具体脚本类所需的信息，他们都不包含脚本类的可执行代码(请注意理解 MonoBehaviour 组件和 .cs 脚本文件，MonoBehaviour 组件和 MonoScript 都不包含具体执行的代码)

一个 MonoScript 包含三个字符串：程序集(assembly)，类(class)以及命名空间(namespace)

在构建项目时,Unity 会将 Assets 目录下的松散在各个文件夹下的脚本文件编译成 Mono 程序集(Mono assemblies),在 Plugins 文件外的 C#脚本会被编译到 Assembly-CSharp.dll 中，在 Plugins 内的脚本会被编译成 Assembly-

CSharp-firstpass.dll，此外，Unity 2017.3 还引入了定义定制托管程序集的能力。这些程序集(再加上预编译的 dll 程序集)都会被包含在最终的应用程序中。这些程序集就是 MonoScript 引用的程序集，和其他资源不同，所有程序集在应用程序第一次启动时会被全部加载进来。

我们从外部加载一个 AssetsBundle，明明这个 AssetsBundle 里面并不包含真正的可执行代码(它只挂了个 MonoBehaviour 组件)，为什么就能正确执行相应的代码呢？就是因为 MonoBehaviour 组件包含了一个 MonoScript 引用，MonoScript 包含了 MonoBehaviour 组件所需要的类对应的程序集类名和命名空间，根据程序集、类名和命名空间就可以执行相应的类操作

2.7 资源的生命周期

为了减少加载时间和管理应用程序的内存占用，了解对象的生命周期是很重要的。对象在特定的时间被加载/卸载。当以下情况发生时,对象会被自动加载：

1) 该对象的 Instance ID 被间接引用
2) 对象没有加载到内存中(比如：切换 Home 键，应用内的对象可能被操作系统从内存中删除，这时候再切回游戏，就会自动重新加载对象到内存)
3) 可以定位对象的源数据（比如：Resources 目录下的文件）

对象也可以外部加载，通过在脚本中创建对象或调用资源加载 API 来加载对象(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，通过将每个引用的 File GUID 和 Local ID 转化成为 Instance ID 的方式

对象被卸载有以下三种情况：

1) 垃圾回收机制启动以后，一些对象会被自动卸载。该过程通常会在切换场景(参数为 LoadSceneMode.Single 的时候)，或者脚本中调用了 Resources.UnloadUnusedAssets 时触发，该过程仅卸载没有被引用的对象

2) Resources 目录中的对象可以通过调用 Resources.UnloadAsset API 主动卸载。卸载后对象的 Instance ID 会保持可用状态，对 File GUID 和 Local ID 的条目会被保留且仍然有效。如果有 Mono 变量或其他有指向该对象的活动对象引用了被 Resources.UnloadAsset 卸载的对象，则该对象会在任意有效的引用被间接引用时立刻重新加载

3) 调用 AssetBundle.Unload(true) API 时，加载自 AssetBundle 的对象会被立刻自动卸载。该操作会释放对象 Instance ID 的 File GUID 和 Local ID 引用，任何对卸载对象的引用都会变成“（空）”引用。C#脚本中，任何试图访问已卸载对象上的方法和属性都会导致抛出空引用异常（NullReferenceException）

如果调用了 AssetBundle.Unload(false)，被卸载的 AssetBundle 中仍然处于激活状态的对象不会被回收，但 Unity 会释放其 Instance ID 的 File GUID 和 Local GUID 引用。之后假如它们被从内存中卸载，只剩下对这些被卸载对象的引用，Unity 无法再次重新加载这些对象

2.8 加载复杂层级结构树

当序列化含有大量 Unity 游戏对象的结构树时（例如序列化 Prefab），要记住一点，即整个结构树都会被完全序列化。这就是说，结构树中的每一个游戏对象和组件都会在序列化数据中单独表示。这会对游戏对象结构的载入与实例化的耗时带来有趣的影响

假设一个代码块实例化了一定数量的游戏对象，结构树非常庞大的单个 Prefab 的实例化，会比分别实例化结构树的多个模块然后运行时组合花费更多的 CPU 时间。

深度分析底层数据后发现，实例化和唤醒游戏对象所花费的 CPU 时间在各种情况下大致是相同的，单个独立的 Prefab 的实例化和唤醒仅需非常少量的 CPU 时间（因为不需要 tranmpolining 和 SendTransformChanged 回调）但是，这些细小的时间节省相对于花在读取和序列化数据上的时间是很不值的

正如之前提到的，序列化单独的 Prefab 时，每个游戏对象和其组件都会被分开序列化——即便数据是重复的。一个有 30 个独立元素的 UI 界面中，Unity 会为这些元素序列化 30 次，这就会产生大量的序列化数据。载入时，所

有 30 个重复元素的游戏对象和组件在被转化为新实例对象之前都需要从硬盘中进行读取。正是这里讨论的读取时间决定了实例化大型 Prefab 的性能花销。

Unity 支持嵌套 Prefab 之后，对于有载入大型结构体游戏对象需求的工程而言，要想减少这类工程的载入时间，可以考虑将大型 Prefab 中的可重用元素分开存储到不同的 Prefab 中，并在运行时对它们进行实例化，而不是完全依靠 Unity 的序列化和 Prefab 系统

第三章：Resources 文件夹

本章讨论 Unity 引擎的 Resources 系统，该系统允许开发人员将资源对象存储在一个或多个名为 Resources 的文件夹中，并在运行时使用 Resources API 去加载或卸载资源对象

3.1 Resources 系统的最佳实践

不要用它,这么强烈的建议主要有以下几个方面的原因：

- 使用 Resources 文件夹让精细化的内存管理变得很困难
- 不正确的使用 Resources 系统(Resources 文件夹内资源过多或者有很多个 Resources 文件夹),会导致项目构建时间和游戏启动的时间变长
- 使用 Resources 文件夹,会降低我们将不同资源用于不同平台和不同性能设备的能力(AssetBundle Variants 是 Unity 引擎针对不同性能的机型而加载不同资源的主要方式之一)
- 使用 Resources 文件夹,不利于资源的增量更新

3.2 适当使用 Resources 系统

虽然 Resources 系统问题很多，但以下 2 种情况，Resources 系统还是非常好用的:

1) 因为 Resources 系统使用起来非常简单方便, 所以在项目初期需要快速出 Demo 的时候，可以适当使用
2) 资源符合以下条件

- 贯穿整个生命周期
- 不是内存密集型资源
- 不需要打补丁更新
- 没有平台和设备性能之分
- 被引用的最小单位 prefab(比如:UILabel,UIButton,UIInput 等我们自定义的最小单位 prefab)

第二种情况我举两个例子解释一下，比如：1. 一些被 MonoBehaviour 代码内 public 变量在 Inspector 内直接拖入的 prefabs; 2. 用于存储配置数据的 ScriptableObjects(比如存储 Facebook 应用 id,打包的一些参数等)

3.3 Resources 资源的序列化

在构建项目时，所有名为 Resources 的文件夹中的资源和对象都会被合并到一个序列化的文件中。这个文件还包含元数据和索引信息，类似于一个 AssetBundle。正如在 AssetBundle 文档中描述的那样，该索引包括一个序列化的查找树，用于将给定对象的名称解析为其相应的 File GUID 和 Local ID，还用于在序列化文件主体中的特定字节偏移位置来定位对象

在大多数平台上，都会使用平衡查找树这种数据结构来查找数据，它的构建时间以 O(n log(n))的速度增长，这导致生成资源的平衡查找树的时间随着 Resources 文件夹中对象数量的增长而超线性增长；此操作不可跳过，并在应用程序启动时执行

低端移动设备上初始化一个包含 10,000 个资源的 Resources 系统需要花费好几秒钟，尽管资源文件夹中的绝大多数资源在应用程序的第一个场景中压根没有被用到

第四章：AssetBundle 基本原理

本文将会探讨 AssetBundle， 介绍它的基本原理和与它交互的一些核心 API，也会介绍加载 AssetBundle 本身以及 加载 AssetBundle 内包含的对象

4.1 概述

AssetBundle 系统可以将一个或者多个文件打包成 Unity 可索引和可序列化的存档格式。AssetBundle 是 Unity 引擎针对用户安装应用以后，更新非代码内容的主要方式。这允许开发人员提交一个较小的应用安装包；最大程度的减少应用程序运行时的内存压力；并且可以根据用户的设备性能加载不同的内容。理解 AssetBundle 的工作原理对于构建 Unity 移动项目至关重要

4.2 AssetBundle 结构

简单来说，一个 AssetBundle 由两部分组成: 头部和数据段

头部包含 AssetBundle 的一些基本信息，比如：标识符(identifier), 压缩类型，清单(manifest). 清单是一个查找表(Lookup table),它的键是对象的名字,它的值则是对象的字节索引， 指示对应对象在 AssetBundle 的数据段内的字节位置，利用这个索引，能准确的得到要获取的对象。 在大多数平台上，这个查找表是用平衡查找树实现的(Windows 和 OSX 平台(包括 iOS)使用红黑树实现)。因此，构建清单所需的时间会随着 AssetBundle 中包含对象数量的增加而超线性增加(译者注：因为红黑树的插入时间复杂度是: nlog(n))

数据段对应的是它包含的所有对象序列化之后的数据，如果 AssetBundle 采用 LZMA 的压缩方式，那么 AssetBundle 包含的所有对象序列化之后的数据会被统一使用 LZMA 算法压缩成一个完整字节数组；如果 AssetBundle 采用采用 LZ4 的压缩方式，那么 AssetBundle 包含的所有对象序列化之后的数据会针对单个对象使用 LZ4 算法单独压缩；如果 AssetBundle 采用不压缩的方式，那么它就是 AssetBundle 包含的对象序列化之后的原始字节流

在 Unity 5.3 之前, AssetBundle 不允许针对单个对象压缩。所以在 Unity5.3 之前的版本，如果要从压缩的 AssetBundle 里读取一个或者多个对象，

必须先解压整个 AssetBundle。通常，Unity 会缓存解压过的 AssetBundle 用来提高之后的加载性能（后续的加载就不需要再重新解压这个 AssetBundle）

4.3 加载 AssetBundle

Unity 提供了四个独立的 API 来加载 AssetBundle，这四个 API 的行为不同取决于以下两个标准:

1. AssetBundle 的压缩类型 (压缩类型有: LZMA 压缩,LZ4 压缩,以及不压缩）
2. 加载 AssetBundle 的设备的操作系统平台(主要有: iOS，Android)

这四个 API 分别是：

- AssetBundle.LoadFromMemory(有对应异步 API）
- AssetBundle.LoadFromFile(有对应异步 API)
- UnityWebRequest ownloadHandlerAssetBundle
- WWW.LoadFromCacheOrDownload (Unity5.6 及以上版本已弃用)

4.3.1 AssetBundle.LoadFromMemory(Async)

Unity 官方不推荐使用此 API 来加载 AssetBundle

AssetBundle.LoadFromMemoryAsync 从托管代码字节数组（在 C#中就是 byte[]）中加载 AssetBundle, 它会从托管代码字节数组拷贝一份原始数据到新申请的托管堆连续内存块中，如果加载的 AssetBundle 是采用 LZMA 压缩，则会在拷贝的时候进行解压，如果采用未压缩或者采用 LZ4 压缩，则是直接拷贝。(译者注：这个 API 的主要用处，我觉得是用于加载已加密的 Assetbundle；将资源打包成 AssetBundle 以后，再对其文件二进制数据加密，加载这个 AssetBundle 之前，先获取它的二进制数据，解密成正确的二进制数组，再使用 AssetBundle.LoadFromMemoryAsync 来”创建”(也可以说”还原”)AssetBundle，此 API 旧版本的名字是 CreateFromMemory, 其实很贴切它的功能)

使用该 API 加载 AssetBundle 的内存峰值至少是 AssetBundle 本身大小的 2 倍，AssetBundle 本身二进制数组一份内存，此 API 拷贝的在托管堆中的

另一份内存。所以使用该 API 加载 AssetBundle 内的对象，将至少会有 3 份内存，一份是在 AssetBundle 原始二进制数组，一份在拷贝之后的托管堆上，还要一份在是在 GPU 或者系统内存中的正在使用的实例（Instantiate 之后的实例）

4.3.2 AssetBundle.LoadFromFile(Async)

AssetBundle.LoadFromFile 是一个高效的 API，它用于从本地存储（硬盘或 SD 卡）加载未压缩或采用 LZ4 压缩的 AssetBundle

在桌面应用、控制台应用和移动平台应用上, 该 API 只加载 AssetBundle 的头部内容，并将剩余的数据留在磁盘上。如果有加载方法(比如: AssetBundle.Load)被调用或者它子对象的 InstanceID 被引用(被其他对象依赖), AssetBundle 的对象才会被加载进内存，因此该 API 加载 AssetBundle 不会有过多的内存占用。但是在 Unity 编辑器环境中, 该 API 会将整个 AssetBundle 加载到内存中，就好像从磁盘读取字节数组并使用 AssetBundle.LoadFromMemoryAsync 创建 AssetBundle 一样。因此，若在编辑器环境下对项目进行内存分析, 该 API 可能导致在 AssetBundle 加载期间出现内存峰值，但实际上这只会发生在编辑器环境下

注意:在安卓平台下，如果你的应用是基于 Unity5.3 或更早的版本构建的，那么使用当前 API 加载 StreamingAssets 目录下的 AssetBundle 会失败; 这个问题已在 Unity5.4 中得到解决

4.3.3 AssetBundleDownloadHandler

UnityWebRequest API 允许开发人员准确地指定 Unity 应该如何处理下载的数据， 并允许开发人员消除不必要的内存使用。使用 UnityWebRequest 下载 AssetBundle 最简单的方法是调用 UnityWebRequest.GetAssetBundle()

就本文主旨而言，我们感兴趣的类是 DownloadHandlerAssetBundle，在工作线程中,它将下载的数据流存储到固定大小的缓冲区中, 然后将数据存入到到临时内存或 AssetBundle 缓存中，这取决于如何处理 Download 句柄
译者注：这里可能有点费解，它其实说的是如下两种情况

```
UnityWebRequest uwr = UnityWebRequest.GetAssetBundle(uri);
//此时数据会存入临时变量中
byte[] bytes = uwr.downloadHandler.data;
//此时数据会被存入 AssetBundle 缓存中
AssetBundle ab = (uwr.downloadHandler as
DownloadHandlerAssetBundle).assetBundle;
```

所有这些操作都发生在 native code 中(区别于 managed code,它不是由 CLR 而是由操作系统直接执行，所以它不受 CLR 的垃圾回收机制控制，也就不会有扩大 CLR 维护的托管堆内存的风险)；此外，这个 Download 句柄并没有保留 native code 下载的数据的拷贝（它只是类似一个指针，指向下载的数据），进一步减少了下载 AssetBundle 的内存开销

LZMA 压缩的 AssetBundle 会在下载期间自动解压并使用 LZ4 压缩格式存入缓存(缓存压缩默认是启用的，由属性 Caching.compressionEnabled 控制)

下载完成以后，Download 句柄的 assetBundle 属性可以用来直接访问下载的 AssetBundle， 就好像 AssetBundle.LoadFromFile 在 AssetBundle 加载完成以后自动调用了

如果将缓存信息提供给 UnityWebRequest 对象，并且请求的 AssetBundle 已经存在于 Unity 的缓存中，那么 AssetBundle 将立即可用，这个 API 将与 AssetBundle.LoadFromFile 操作相同(译者注：说的是这个 API UnityWebRequest GetAssetBundle(string uri, CachedAssetBundle cachedAssetBundle, uint crc))

在 Unity 5.6 之前，UnityWebRequest 系统使用了一个固定的线程池和一个内部作业系统来防止过度的并发下载，线程池的大小是固定的。在 Unity 5.6 中，为了适应当前快速发展的硬件，并允许更快地访问 HTTP 响应代码和头部，这些安全措施已经被移除

4.3.4 WWW.LoadFromCacheOrDownload

*注:从 Unity 2017.1 开始，WWW.LoadFromCacheOrDownload 简单地围绕 UnityWebRequest 展开。因此，使用 Unity 2017.1 或更高版本的开发者应该

迁移到 UnityWebRequest；WWW.LoadFromCacheOrDownload 在未来的版本中会被弃用*

详细内容就不展开翻译，因为官方已弃用

4.3.5 官方推荐

一般情况下， 应该尽可能的使用 AssetBundle.LoadFromFile，就内存占用、加载速度、以及磁盘占用等方面来看，这个 API 是最高效的

对于需要热更新打补丁的项目来说，如果使用的是 Unity 5.3 或更高版本，那么强烈推荐使用 UnityWebRequest；如果使用 Unity 5.2 或更旧版本，则建议使用 WWW.LoadFromCacheOrDownload

使用 UnityWebRequest 或者 WWW.LoadFromCacheOrDownload 时，请确保下载器代码在加载 AssetBundle 后正确调用 Dispose，C＃的 using 语法糖是确保 WWW 或 UnityWebRequest 安全处置的最便捷方式

译者注：如果类实现了 IDisposable 接口，而你又不知道什么时候该主动调用 Dispose()的时候，使用 using 语法糖真心是个很不错的方式，示例如下

```
using(UnityWebRequest uwr = UnityWebRequest.Get(url))
{
    yield return uwr.SendWebRequest();
    if (actionResult != null)
    {
        actionResult(uwr);
    }
}
```

对于需求独特，下载量巨大的项目，可以考虑使用自定义下载器。编写自定义下载器是一个很好的挑战，任何自定义下载器都应与 AssetBundle.LoadFromFile 兼容

4.4 从 AssetBundle 中加载对象

UnityEngine.Objects 可以使用三个不同的 API 从 AssetBundles 加载对象，它们都具有同步和异步方式：

- LoadAsset (LoadAssetAsync)
- LoadAllAssets (LoadAllAssetsAsync)
- LoadAssetWithSubAssets (LoadAssetWithSubAssetsAsync)

异步 API 每一帧会加载多个对象，上限由他们的时间片（time-slice）决定，明白它的底层加载细节就能明白它这种行为的原因，下一节将会介绍

加载多个独立的对象时应该使用 LoadAllAssets，但最好只在需要加载 AssetBundle 中的大部分或全部对象时才能使用它 。LoadAllAssets 比调用 LoadAssets 多次要快。因此，如果要加载的资源数量很多，但又不到 AssetBundle 内包含的对象的 66%时，请考虑将 AssetBundle 拆分为多个较小的包并使用 LoadAllAssets 一次性加载

加载包含多个嵌套对象的复合对象时，应使用 LoadAssetWithSubAssets，例如嵌入动画的 FBX 模型或嵌入多个精灵的精灵图集。如果需要加载的对象全部来自同一个对象，但与许多其他不相关的对象一起存储在 AssetBundle 中，则使用此 API

对于任何其他情况，请使用 LoadAsset 或 LoadAssetAsync

4.4.1 底层加载细节

Unity 对象（UnityEngine.Object）的加载不是主线程执行的，从磁盘读取一个对象的数据是在工作线程完成的。 任何不涉及 Unity 系统的线程敏感部分（脚本，图形）的内容都将在工作线程上进行转换。例如：根据网格数据生成 VBO（Vertex Buffer Objects 顶点缓冲对象），纹理解压等

从 Unity 5.3 开始，对象加载已经并行化。 多个对象在工作线程上被反序列化，处理和集成。 当一个对象完成加载时，它的 Awake 回调将被调用，并且该对象将在下一帧期间可用于 Unity Engine 的其余部分

同步的 AssetBundle.Load 方法将暂停主线程，直到对象加载完成。 他们还会对对象加载进行时间片分割，以便对象集成不会占用超过一定数量的毫秒帧时间。 毫秒数由属性 Application.backgroundLoadingPriority 设置：

- ThreadPriority.High:每帧最多 50ms
- ThreadPriority.Normal: 每帧最多 10ms
- ThreadPriority.BelowNormal:每帧最多 4ms
- ThreadPriority.Low: 每帧最多 2ms

从 Unity 5.2 开始，多个对象被加载，直到达到对象加载的帧时间限制。假设所有其他因素相同，由于发出异步调用和引擎可用对象之间的最小一帧延迟，对象加载的异步 API 的完成时间总是比可比较的同步 API 要长一帧时间

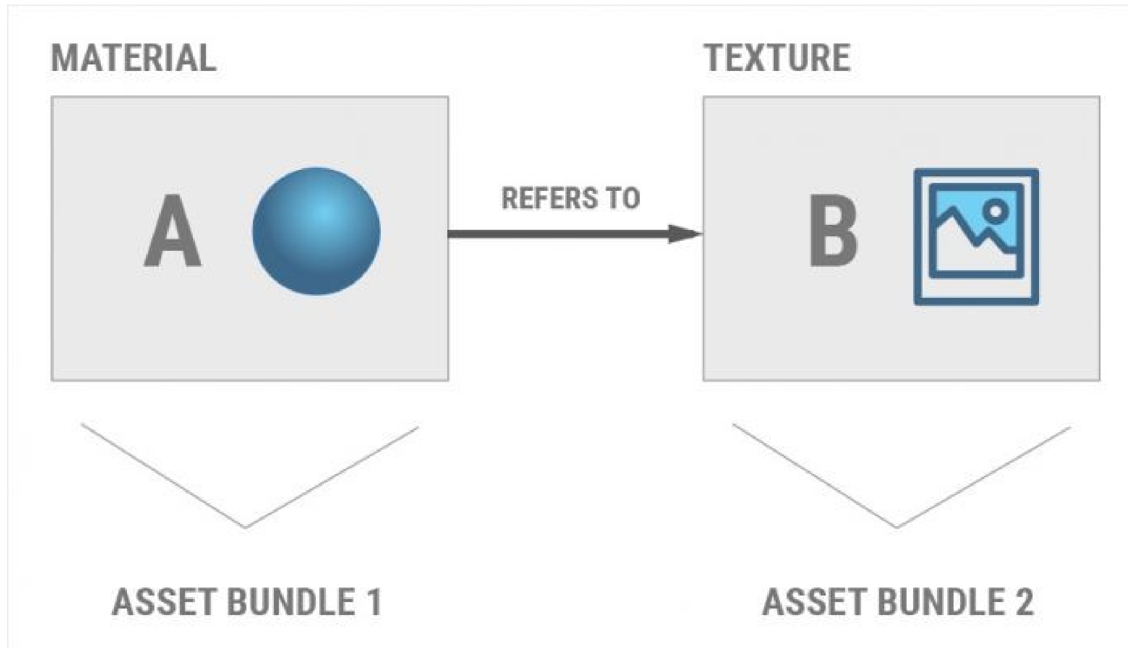
4.4.2 AssetBundle 依赖

AssetBundles 之间的依赖关系使用两个不同的 API 自动跟踪，具体取决于运行时环境。在 Unity 编辑器中，可以通过 AssetDatabase API 查询 AssetBundle 依赖关系。可以通过 AssetImporter API 访问和更改 AssetBundle 分配和依赖关系。在运行时，Unity 提供了一个可选的 API 来加载 AssetBundle 构建期间通过基于脚本对象的 AssetBundleManifest API 生成的依赖信息。

当一个或多个父 AssetBundle 的 UnityEngine.Objects 引用一个或多个其他 AssetBundle 的 UnityEngine.Objects 时，AssetBundle 依赖于另一个 AssetBundle。有关对象间引用的更多信息，请参阅资[译]Unity 资源系列 2—资源对象和序列化 如该文章的序列化和实例部分所述，AssetBundles 充当由 AssetBundle 中包含的每个对象的 FileGUID＆LocalID 标识的源数据的源

因为一个对象在其实例 ID 被第一次引用时被加载，并且因为一个对象在加载其 AssetBundle 时被分配了有效的实例 ID，所以 AssetBundles 加载的顺序并不重要。相反，加载对象本身之前加载包含 Object 的依赖关系的所有 AssetBundles 是非常重要的。装载父级 AssetBundle 时，Unity 不会尝试自动加载任何子 AssetBundles

举个例子：



假设材质 A 引用了纹理 B.材质 A 被打包到 AssetBundle 1 中，纹理 B 被打包到 AssetBundle 2 中

这并不意味着必须在 AssetBundle 1 之前加载 AssetBundle 2，或者必须从 AssetBundle 2 中显式加载纹理 B。在从 AssetBundle 1 加载材质 A 之前加载 AssetBundle 2 就足够了

注意，在加载 AssetBundle 1 时，Unity 不会自动加载 AssetBundle 2,这必须在脚本代码中手动完成

4.4.3 AssetBundle 清单（manifests）

当使用 BuildPipeline.BuildAssetBundles API 执行 AssetBundle 构建管道时，Unity 会序列化一个包含每个 AssetBundle 的依赖性信息的对象。此数据存储在单独的 AssetBundle 中，其中包含一个单独的 AssetBundleManifest 对象

此资产将存储在 AssetBundle 中，其名称与构建 AssetBundles 的父目录相同。如果项目将其 AssetBundles 构建到（projectroot）/ build / Client /

文件夹，则包含该清单的 AssetBundle 将被保存为（projectroot）/build/Client/Client.manifest

包含清单的 AssetBundle 可以像其他任何 AssetBundle 一样加载，缓存和卸载

AssetBundleManifest 对象本身提供 GetAllAssetBundles API 来列出与清单并发构建的所有 AssetBundles，以及两个方法来查询特定 AssetBundle 的依赖关系：

- AssetBundleManifest.GetAllDependencies 返回需要查询的 AssetBundle 的所有分层依赖项，其中包括 AssetBundle 的直接子项，子项的子项等的依赖项
- AssetBundleManifest.GetDirectDependencies 只返回一个需要查询的 AssetBundle 的直接子项

请注意，这两个 API 都分配字符串数组。相应地，它们只应该谨慎使用，而不应该在应用程序生命周期的性能敏感部分中使用

4.4.4 建议

在大多情况下，在玩家进入应用程序的性能关键区域（如主场景或战斗场景）之前，最好预先加载所需的对象。 这在移动平台上尤其重要，因为访问本地存储的速度很慢，而且在播放时加载和卸载对象可能会触发 GC

第五章：AssetBundle 最佳实践

前几章讲述了 AssetBundle 的基本原理，包括几种加载 API 的底层行为, 本章讨论在实践中使用 AssetBundle 时可能遇到的问题和对应的解决方案

5.1 管理已加载的资源（Assets）

在对内存敏感的环境中，控制加载对象的大小和数量非常有必要。当对象从当前活动场景中移除时，Unity 不会自动卸载它们。资源清理会在特定时间触发，当然，也可以手动触发

无论是通过 Unity 缓存还是通过 AssetBundle.LoadFromFile 去加载一个存储在本地(硬盘或存储卡)的 AssetBundle 文件，其开销并不大(因为它只会加载 AssetBundle 的头部内容)，一般也就几十 Kb；当然，如果有大量的 AssetBundle 头部保存在托管堆内存中，那么内存开销就不容忽视了

因为多数应用允许用户重复体验某内容(比如重玩某个关卡)；所以知道什么时候去加载和卸载 AssetBundle 就尤为重要了。如果 AssetBundle 卸载不当，可能会导致同一个对象占用两份内存（下文会详细解释）；而且在某些情况下会导致与期望不符的情况发生，例如导致纹理丢失（模型，场景变成洋红色）

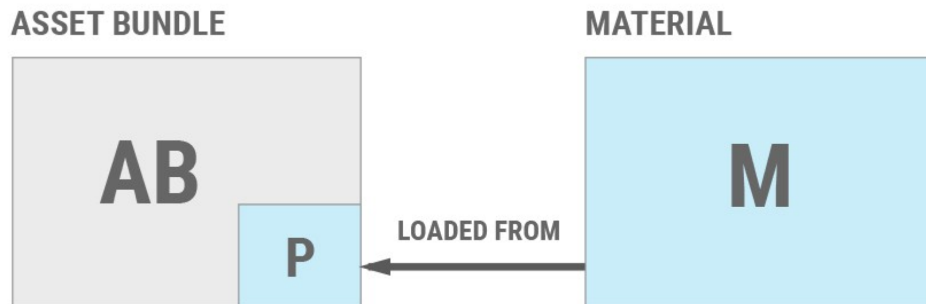
管理 AssetBundle 本身以及管理从 AssetBundle 中加载的对象很重要的一件事情，是要搞清楚调用 AssetBundle 实例的 unload 方法时，它的 unloadAllLoadedObjects 参数为 true 或者 false 的时候，会发生什么 ？ 到底有什么不一样？

译者注：这一段其实说的搞清如下示例最后 2 行代码的区别非常重要

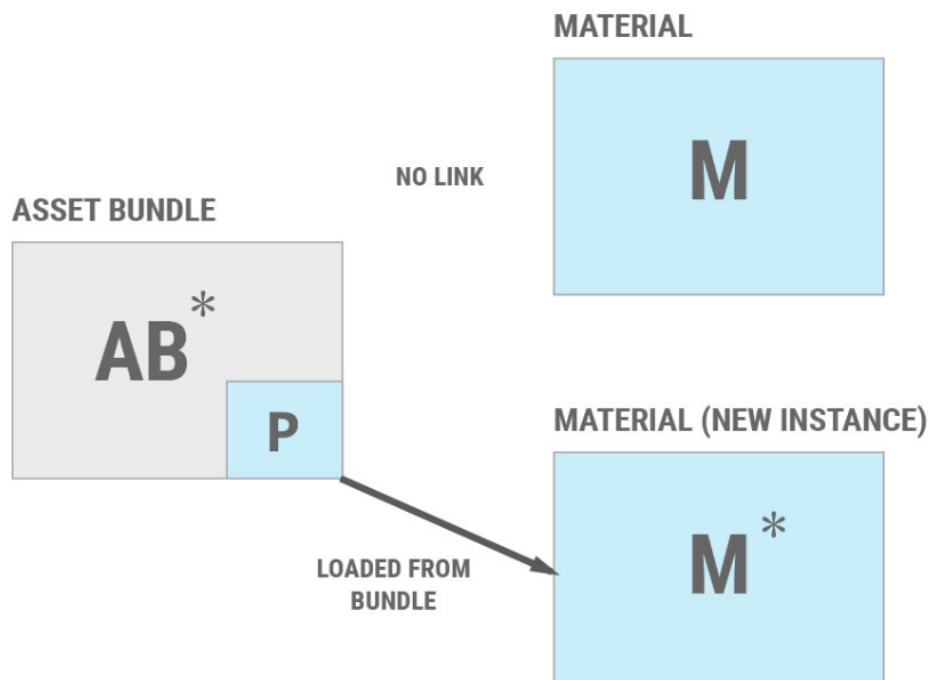
```
AssetBundle ab = AssetBundle.LoadFromFile(abpath);
//参数是：unloadAllLoadedObjects
//public void Unload(bool unloadAllLoadedObjects);
ab.Unload(true);
ab.Unload(false);
```

调用这个 API，无论参数是 true 还是 false，都会将该 AssetBudnle 的头部信息卸载掉；其参数 unloadAllLoadedObjects 标记是否同时也卸载掉那些从该 AssetBundle 实例化的对象。如果参数是 true，那么所有从这个 AssetBundle 创建的对象，即使当前活动场景中正在使用那些对象，也会被立即卸载掉

举例来说，假设材质球 M 是从 AssetBundle AB 中加载的，假设 M 当前在活动场景中



如果 AB.Unload(true)被调用，那么 M 将会从场景中被移除，销毁和卸载如果 AB.Unload(false)被调用，会 M 依然留在场景中，并且将会一直有效。调用 AssetBundle.Unload(false)将会打断 M 和 AB 之间的连接。如果 AB 稍后再次加载，那么从新的 AB*中加载的 M*和以前的 M 会同时存在内存中，Unity 并不会在 AB*和 M 之间建立联系



对于大多数项目来说，出现这种情况是不可取的。大多数项目应该使用 AssetBundle.unload(true)，并采用某种方法来确保不会出现对象重复（比如：上面出现了 M 和 M*两个对象），两种常用的方法是：

1. 在应用程序的生命周期中，一些明显的界限点(不同场景之间，或者加载界面上)，将生命周期短暂的 AssetBundle 卸载掉（比如：一些特效，不会一直在游戏的过程中使用的），这是最简单和常用的方法
2. 维护单个对象的引用计数，并仅在所有组成对象都未使用时卸载 AssetBundle，这允许应用程序卸载和重新加载单个对象，也不会有多份内存占用

如果一个应用必须使用 AssetBundle.unload(false)，那么只能通过下面的两种方法将单独的对象卸载：

1. 消除这个不想要的对象的所有引用，场景中和代码中都要清除掉，都做好了，然后手动调用 Resources.UnloadUnsedAssets

2. 以非叠加的方式加载一个场景，这样会销毁当前场景中所有的对象，然后自动调用 Resources. UnloadUnsedAssets

要做到这点最简单的方法就是，将工程分割成一块块的场景，然后将这些场景连同他们的依赖项打包到 AssetBundles，应用进入到一个加载场景，完全卸载老场景的 AssetBundle，然后加载新场景的 AssetBundle

这种流程太简单了，而一些项目需要更为复杂的 AssetBundle 管理。每个项目的数据是不同的，这里并没有统一的 AssetBundle 设计模式。当决定如何分类对象，将他们打包到 AssetBundles 时，一般来说开始时，最好以那些需要同时加载或更新的对象打包在一起为原则。

举例来说，一个角色扮演游戏，除了一些大多场景都会用到的对象，单独的地图和过场动画可以按场景归类到 AssetBundles，但是一些对象会被多个场景需求。比如：玩家的头像，Ui 资源，不同角色的模型以及对应纹理也可以将 AssetBundles 打包成肖像包，Ui 包和不同的角色模型和纹理，这些资源可

以单独打包，然后在场景加载之前，预先加载这些公共的对象，并在应用程序的生命周期内，保持加载的状态

如果 Unity 在一个 AssetBundle 被卸载后，又需要从这个 AssetBundle 中重新加载一个对象，在这个情况下，加载将会失败，这个对象将会以一个（Missing）Object 的形式出现在 Unity 编辑器的层级面板中

这种情况主要发生在 Unity 失去再重获图形上下文控制权的时候，比如：移动 app 被暂停，或用户锁住 PC 的时候。这个时候，Unity 必须重新上传纹理和 shaders 到 GPU 中才行。如果此时，这些 assets 的源 AssetBundle 不可用了，那么应用将以洋红色（ “missing shader” ）渲染这些场景中对象

5.2 AssetBundle 的发布

将项目的 AssetBundle 发布给应用有两种基本方法：与应用包同时安装或在安装后下载它们；无论最初如何发布 AssetBundle，良好的体系结构应该允许在应用安装以后，还可以下载新的 AssetBundle 内容或者旧的 AssetBundle 的补丁用于更新或者修复应用

5.2.1 随项目安装

将 AssetBundles 依附在项目中，是发布他们最简单的方法，因为这样就不需要额外的下载管理代码了。让项目在安装时包含 AssetBundles，有两个主要原因：

- 减少项目构建时间，允许简单的迭代开发。如果这些 AssetBundles 不需要单独更新，那么 AssetBundles 可以直接包含在应用中，通过 Steaming Assets 的形式存储在应用中
- 可作为更新内容的初始版本。一般这么做是为了减少用户在初始安装后的时间，或作为后续更新的基础从而节约时间

在安装时在 Unity 应用程序中包含任何类型的内容的最简单方法是在构建项目之前将内容构建到/ Assets / StreamingAssets /文件夹中。构建时 StreamingAssets 文件夹中包含的任何内容都将复制到最终应用程序中

可以在运行时通过属性 Application.streamingAssetsPath 访问本地存储上 StreamingAssets 文件夹的完整路径。然后可以在大多数平台上通过 AssetBundle.LoadFromFile 加载 AssetBundle

注意：在 Android 上，StreamingAssets 文件夹中的资源存储在 APK 中，如果压缩它们可能需要更多时间加载，您可以导出 Gradle 项目并在构建时向 AssetBundles 添加扩展。然后，您可以编辑 build.gradle 文件并将该扩展添加到 noCompress 部分。这也是为什么有些人说项目用 Gradle 打包以后，项目加载很慢，是因为 Android Studio 默认帮你压缩了 StreamingAssets 目录下的内容而导致的

5.2.2 安装后下载

移动设备上最受欢迎的 AssetBundles 交付方法还是在应用安装后进行下载。这样允许在用户安装后更新或添加新的内容，而不用强制用户去重新下载整个应用。在移动平台上，应用必须经过一个痛苦而耗时的审核过程。因此，开发一个好的系统来支持安装后下载，至关重要

交付 AssetBundle 最简单的方法，就是把他们放在一个 web 服务器上，然后通过 UnityWebRequest 去加载，Unity 会在本地存储中自动缓存下载好的 AssetBundles。如果下载的 AssetBundle 是 LZMA 压缩格式，为了之后更快的加载，缓存中的 AssetBundle 是被解压过的。如果下载下来的包是 LZ4 压缩的，在缓存中 AssetBundle 将会保持压缩格式不变。

如果缓存满了，Unity 将会从缓存里把最不常用的 AssetBundle 删除；在意内置 API 的内存消耗，对缓存行为和表现都无法接受的；或者需要用平台语言来达到要求的特殊项目来说，则需要在定制下载系统上下功夫

说说几个不便使用 UnityWebRequest 的情景：

- 需要对 AssetBundle 缓存进行细粒度控制的
- 项目需要实现一个定制化的压缩策略
- 当项目需要使用平台相关的 API 来满足一些特殊需求，比如：在非激活状态下流动数据（使用 IOS 后台任务 API，在后台进行下载数据）

- 必须在一些 Unity 不支持 SSL 的平台（比如 PC）上通过 SSL 交付 AssetBundles

5.3 Unity 内置缓存系统

5.3.1 缓存介绍

Unity 有个内置的 AssetBundle 缓存系统专门用来缓存通过 UnityWebRequest 下载的 AssetBudle . 这个 API 有重载方法，可以传入一个版本号当做参数，这个数字没有存在 AssetBundle 中，也不会由 AssetBundle 系统生成。缓存系统跟踪传递给 UnityWebRequest 的最新版本号。当使用版本号调用此 API 时，缓存系统会通过比较版本号来检查是否存在缓存的 AssetBundle。如果这些数字匹配，系统将加载缓存的 AssetBundle。如果数字不匹配，或者没有缓存的 AssetBundle，则 Unity 将下载新 AssetBundle，此新 AssetBundle 将与新版本号相关联

缓存系统中的 AssetBundle 仅由其文件名标识，而不是由下载它们的完整 URL 标识。这意味着具有相同文件名的 AssetBundle 可以存储在多个不同的位置，例如内容分发网络。只要文件名相同，缓存系统就会将它们识别为相同的 AssetBundle（译者注：这一点很重要，因为加载的文件一般会通过 CDN 分发，所以同一个文件的完整 URL 可能是不一样的）

每个应用程序都可以确定将版本号分配给 AssetBundles 的适当策略，并将这些数字传递给 UnityWebRequest。这些数字可能来自各种类型的唯一标识符，例如 CRC 值。请注意，虽然 AssetBundleManifest.GetAssetBundleHash()也可用于此目的，但我们不建议使用此函数进行版本控制，因为它只提供估算，而不是真正的哈希计算

在 Unity 2017.1 之后，Caching API 扩展了更多内容，提供更精细的控制允许开发人员从多个缓存中选择活动的缓存。Unity 的早期版本只能通过修改 Caching.expirationDelay 和 Caching.maximumAvailableDiskSpace 来删除缓存项（这些属性在 Unity 2017.1 中依然有保留，但被标记为已过时）

expirationDelay 是自动删除 AssetBundle 之前必须经过的最小秒数。如果在此期间未访问 AssetBundle，则会自动删除它

maximumAvailableDiskSpace 是指缓存在本地存储的大小(以字节为单位)；当存储达到最大限制时，Unity 将删除缓存中最近最少打开的 AssetBundle(如果你不想某个 AssetBundle 被删除，可以采用 Caching.MarkAsUsed 将使用它的时间修改为当前时间)，Unity 将删除已缓存的 AssetBundle，直到有足够的空间来完成新的下载并缓存

5.3.2 缓存启动

由于 AssetBundles 由其文件名标识，因此可以使用随应用程序提供的 AssetBundle“填充”缓存。为此，请将每个 AssetBundle 的初始版本或基本版本存储在/ Assets / StreamingAssets /中

可以通过在第一次运行应用程序时从 Application.streamingAssetsPath 加载 AssetBundle 来填充缓存。从那时起，应用程序可以正常调用 UnityWebRequest（UnityWebRequest 也可以用于从 StreamingAssets 路径初始加载 AssetBundle）

5.4 自定义下载

编写自定义下载程序使应用程序可以完全控制 AssetBundles 的下载，解压缩和存储方式。由于涉及的工程工作非常重要，我们建议这种方法仅适用于大型团队。编写自定义下载程序时有四个主要注意事项：

- 下载机制
- 存储位置
- 压缩类型
- 修复补丁

5.4.1 下载

对于大多数应用程序，HTTP 是下载 AssetBundles 的最简单方法。但是，实现基于 HTTP 的下载器并不是最简单的任务。自定义下载程序必须避免过多的内存分配，过多的线程使用和过多的线程唤醒。

编写自定义下载程序时，有三个选项：

- C＃的 HttpWebRequest 和 WebClient 类
- Asset Store 中的插件包
- 自定义原生下载插件

5.4.1.1 C＃类

如果应用程序不需要 HTTPS / SSL 支持，C＃的 WebClient 类提供了最简单的下载 AssetBundle 的机制。它能够将任何文件异步下载到本地存储，而无需过多的托管内存分配

要使用 WebClient 下载 AssetBundle，请分配该类的实例，并将要下载的 AssetBundle 的 URL 和目标路径传递给它。如果需要对请求的参数进行更多控制，则可以使用 C＃的 HttpWebRequest 类编写下载程序：

1. 从 HttpWebResponse.GetResponseStream 获取字节流
2. 在堆栈上分配固定大小的字节缓冲区
3. 从响应流读入缓冲区
4. 使用 C＃的 File.IO API 或任何其他流式 IO 系统将缓冲区写入磁盘

5.4.1.2 Asset Store 中的插件包

Asset Store 内有很多的代码实现，以通过 HTTP，HTTPS 和其他协议下载文件。在为 Unity 编写自定义本机代码插件之前，建议您评估可用的 Asset Store 软件包

5.4.1.3 自定义原生下载插件（Native-Plugins）

编写自定义原生下载插件是最耗时但最灵活的方法。由于开发原生插件需要大量的时间，而且技术风险高，只有在没有其他方法能够满足应用要求的情况下才推荐使用此方法。例如，如果应用程序必须在 Unity 中没有 C＃ SSL 支持的平台上使用 SSL 通信

自定义平台下载插件通常会包装目标平台的下载 API，比如：iOS 上的 NSURLConnection 和 Android 上的 java.net.HttpURLConnection。有关使用这些 API 的更多详细信息，请参阅每个平台的文档

5.4.2 存储

在所有平台上，Application.persistentDataPath 都指向一个可写位置，该位置应该用于存储应在应用程序运行之间保留的数据。编写自定义下载程序时，强烈建议使用 Application.persistentDataPath 的子目录来存储下载的数据

5.5 资源分配策略

决定如何将项目的资源划分为 AssetBundle 并不简单。采用简单的策略是很诱人，例如将所有对象放在他们自己的 AssetBundle 中或只使用一个 AssetBundle，但这些解决方案有明显的缺点：

- AssetBUndle 太少
    - 增加运行时内存占用
    - 增加加载时间
    - 不利于补丁更新
- AssetBundle 太多
    - 增加构建时间
    - 使开发复杂化
    - 增加总下载时间

5.6 常见的问题

本节介绍使用 AssetBundle 的项目中常见的几个问题

5.6.1 资源重复

Unity5 以后的 AssetBundle 系统，当打包一个资源文件时，系统会遍历这个资源引用的其他资源，然后根据其他资源的信息来判断是否将他们打到当前文件的 AssetBundle 内（译者注：想一下为什么不是直接打包进去而要判断呢？）

在导入一个资源到 Unity 项目中的时候，如果我们根据导入器 API 主动设置了它的 assetBundleName 属性，或者我们在编辑器下的 Inspector 界面最下方设置 assetBundle 如下图：

那么，这个资源只会被打包到名为 assetBundleName 的 AssetBundle 内，即便其他资源引用了这个资源，也不会打包到其他 assetBundle 内

除了在导入的时候根据导入器的 API 以及在 Inspector 界面手动设置之外，还可以根据重载的 BuildPipeline.BuildAssetBundles（outputPath, AssetBundleBuild[] builds, Options, targetPlatform)），来设置并直接打包；AssetBundleBuild 类里面就包含了 assetBundleName 以及对应的对象

如果该资源未通过以上三种方式设置 AssetBundleName，那么如果有其他资源有用到该资源，那么该资源会被打包到直接引用了该资源的所有的 AssetBudnle 内

比如：资源 A 和资源 B，分别打包为 AssetBundleA 和 AssetBundleB，两者都引用了资源 C，那么资源 C 将被复制到两个 AssetBundle 中，这将增加应用程序的 AssetBundles 的总大小，而且资源 C 的两个副本将被视为具有不同标识符的不同对象

通过组合 AssetDatabase 和 AssetImporter API，可以编写一个 Editor 脚本，确保将所有 AssetBundle 的直接或间接依赖项分配给 AssetBundle，或者没有两个 AssetBundle 共享尚未分配给 AssetBundle 的依赖项。Unity 官方也提供了这样的工具，建议使用 Unity 官方工具

5.6.2 SpriteAtlas 重复

SpriteAtlas 针对旧版本的图集打包系统 Sprite Packer 在性能和易用性上的不足，进行了全面改善。除此之外，相比 Sprite Packer，Sprite Atlas 将对精灵更多的控制权交还给用户。针对 SpriteAtlas 的使用，要注意一些问题，如果 AssetBundle1 引用 SpriteAtlas 内的一张或者多张 Sprite, 那么整个 SpriteAtlas 都会被打包进 AssetBundle 1，而且如果 AssetBundle2 正好也引用了 SpriteAtlas 内的一张或者多张 Sprite, 那么 SpriteAtlas 也会被打包到 AssetBundle2 中，如果 SpriteAtlas 内所有的 Sprite 没被任何 AssetBundle 引用，则它也不会被包到其他 AssetBundle 中

为了确保 SpriteAtlas 不会被多次打包到其他 AssetBundle 中，请确保 SpriteAtlas 内的所有 Sprite 和 SpriteAtlas 本身是打包到同一个 AssetBundle 中（译者注：SpriteAtlas 和它包含的 Sprite 打包 AssetBundle 以后，其他 AssetBundle 有包含或者引用它的 Sprite，也不会把整个 SpriteAtlas 打包进 AssetBundle 中，只会记录一个引用）

请注意，在 Unity 5.2.2p3 及更早版本中，不会将自动生成的 SpriteAtlas 打包 AssetBundle。因此，它们会被打进引用了它包含的 Sprite 的 AssetBundle 中。因为这个问题，建议使用 Unity5 的 sprite packer 功能的所有项目升级到 Unity 5.2.2p4，或更新的版本

5.6.3 Android 纹理

关于纹理相关的内容，可以参考我的一篇文章，移动游戏纹理

5.5.4 iOS 文件处理过度使用

当前版本的 Unity 不受此问题的影响

5.7 AssetBundle Variants

AssetBundle 系统的一个关键特性是引入了 AssetBundle Variants，Variants 的目的是允许应用程序调整其内容以更好地适应其运行时环境。变量允许不同 AssetBundle 文件中的不同 UnityEngine.Object 在加载对象和解析 Instance ID 引用时显示为“相同”对象。从概念上讲，它允许两个 UnityEngine.Objects 共享相同的 File GUID 和 Local ID，并通过字符串 Variant ID 标识要加载的实际 UnityEngine.Object

该系统有两个主要用例：

1.Variants 简化了给不同平台加载与之匹配的 AssetBundle

- 示例：构建系统可能会创建一个 AssetBundle，其中包含适用于独立 DirectX11 Windows 构建的高分辨率纹理和复杂着色器，以及第二个具有适用于 Android 的低保真内容的 AssetBundle。在运行时，项目的资源加载代码可以为其平台加载适当的 AssetBundle Variant，而传递给 AssetBundle.Load API 的 Object 名称不需要更改

2. Variants 允许应用程序在同一平台上根据硬件性能加载不同的内容

- 这是支持各种移动设备的关键。iPhone 4 无法在任何实际应用程序中显示与最新 iPhone 相同的内容保真度
- 在 Android 上，AssetBundle Variants 可用于解决设备之间屏幕宽高比和 DPI 的巨大碎片问题

5.7.1 限制

AssetBundle Variant 系统的一个关键限制是它要求 Variants 从不同的 A 资源构建，即使这些资源之间的唯一差异是其导入设置。如果构建到 Variant A 和 Variant B 中的纹理之间的唯一区别是在 Unity 纹理导入器中选择的特定纹理压缩算法，则变体 A 和变体 B 必须仍然是完全不同的资源。这意味着 Variant A 和 Variant B 必须是磁盘上的单独文件

此限制使大型项目的管理变得复杂，因为特定资产的多个副本必须保留在源代码管理中。当开发人员希望更改资源的内容时，必须更新资源的所有副本，此问题没有内置的解决方法。

大多数团队都实施自己的 AssetBundle Variants 形式。这是通过构建 AssetBundles 来实现的，并在其文件名后附加明确定义的后缀，以便识别给定 AssetBundle 所代表的特定变体。在构建这些 AssetBundle 时，自定义代码以编程方式更改包含的 Assets 的导入器设置。一些开发人员已经扩展了他们的自定义系统，以便能够更改附加到预制件的组件上的参数。

5.8 AssetBundle 压缩还是未压缩？

是否压缩 AssetBundle 需要考虑几个重要因素，其中包括：

加载时间：从本地存储或本地缓存加载时，未压缩的 AssetBundle 加载速度比压缩的 AssetBundle 快得多

构建时间：LZMA 和 LZ4 在压缩文件时非常慢，Unity Editor 按顺序处理 AssetBundles。具有大量 AssetBundle 的项目将花费大量时间来压缩它们

应用程序大小：如果 AssetBundle 在应用程序中提供，压缩它们将减少应用程序的总大小。或者，可以在安装后下载 AssetBundles

内存使用：在 Unity 5.3 之前，所有 Unity 的解压缩机制都需要在解压缩之前将整个压缩的 AssetBundle 加载到内存中。如果内存使用很重要，请使用未压缩或 LZ4 压缩的 AssetBundle

下载时间：仅当 AssetBundle 很大，或者用户处于带宽受限的环境中时（例如在低速或计量连接上下载），才可能需要压缩。如果在高速连接上仅向 PC 传送几十兆字节的数据，则可以省略压缩

5.8.1 Crunch 压缩

由使用 Crunch 压缩算法的 DXT 压缩纹理组成的 Bundles 应该被构建为未压缩类型

5.9 AssetBundles 和 WebGL

由于 Unity 的 WebGL 导出选项当前不支持工作线程，因此 WebGL 项目里 AssetBundle 的解压，加载资源等操作全部都在主线程完成。AssetBundle 的下载则可以通过 XMLHttpRequest 委托给浏览器去下载

Unity 建议开发人员用更小的 AssetBundle，以避免出现性能问题。与使用大型 AssetBundle 相比，此方法的内存效率更高。Unity WebGL 仅支持 LZ4 压缩和未压缩资产包，但是，可以对 Unity 生成的包应用 gzip / brotli 压缩。在这种情况下，您需要相应地配置 Web 服务器，以便浏览器在下载时解压缩文件

如果您使用的是 Unity 5.5 或更早版本，请考虑为您的 AssetBundle 避免使用 LZMA，而使用 LZ4 进行压缩，这可以按需非常有效地解压缩。Unity 5.6 删除了 LZMA 作为 WebGL 平台的压缩选项