# 1.A guide to AssetBundles and Resources 资源包和资源指南

This is a series of articles that provides an in-depth discussion of Assets and resource management in the Unity engine. It seeks to provide expert developers with deep, source-level knowledge of Unity's Asset and serialization systems. It examines both the technical underpinnings of Unity's AssetBundle system and the current best practices for employing them.  
这是一系列文章，深入讨论了 Unity 引擎中的资源和资源管理。它旨在为专家开发人员提供有关 Unity 资产和序列化系统的深入源代码级知识。它检查了Unity资产捆绑包系统的技术基础以及当前使用它们的最佳实践。

The guide is broken down into four chapters:  
该指南分为四个章节：

1. **Assets, Objects and serialization** discusses the low-level details of how Unity serializes Assets and handles references between Assets. *It is strongly recommended that readers begin with this chapter* as it defines terminology used throughout the guide.  
   **资源、对象和序列化** 讨论了 Unity 如何序列化资源以及处理资源之间的引用的低级详细信息。 *强烈建议读者从本章开始，* 因为它定义了整个指南中使用的术语。
2. **The Resources folder** discusses the built-in Resources API.  
   **“Resources” 文件夹** 讨论内置的资源 API。
3. **AssetBundle fundamentals** builds on the information in chapter 1 to describe how AssetBundles operate, and discusses both the loading of AssetBundles and the loading of Assets from AssetBundles.  
   **AssetBundle 基础知识以**第 1 章中的信息为基础，描述了 AssetBundle 的运作方式，并讨论了 AssetBundle 的加载和从 AssetBundle 加载资产。
4. **AssetBundle usage patterns** is a long article discussing many of the topics surrounding the practical uses of AssetBundles. It includes sections on assigning Assets to AssetBundles and on managing loaded Assets, and describes many common pitfalls encountered by developers using AssetBundles.  
   **AssetBundle 使用模式** 是一篇长篇文章，讨论了围绕 AssetBundle 实际用途的许多主题。它包括有关将资产分配给AssetBundle和管理加载的资产的部分，并描述了使用AssetBundle的开发人员遇到的许多常见陷阱。

*Note:* This guide's terms for *Objects* and *Assets* differ from Unity's public API naming conventions.  
*注意：*本指南中关于*对象*和资源的定义与 Unity 的公共 API 命名约定不同。

The data this guide calls *Objects* are called *Assets* in many public Unity APIs, such as [AssetBundle.LoadAsset](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadAsset.html) and [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html). The files this guide calls *Assets* are rarely exposed to any public APIs. When they are exposed, it is generally only in build-related code, such as [AssetDatabase](http://docs.unity3d.com/ScriptReference/AssetDatabase.html) and [BuildPipeline](http://docs.unity3d.com/ScriptReference/BuildPipeline.html). In these cases, they are called *files* in public APIs.  
本指南所称的*对象*在许多公共 Unity API 中称为*资产*，例如  [AssetBundle.LoadAsset](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadAsset.html) 和 [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html)。本指南所称的*资产*很少向任何公共 API 公开。当它们被公开时，它通常只在与构建相关的代码中，例如 [AssetDatabase](http://docs.unity3d.com/ScriptReference/AssetDatabase.html) 和 [BuildPipeline](http://docs.unity3d.com/ScriptReference/BuildPipeline.html)。在这些情况下，它们在公共 API 中称为*文件*。

# 2.Assets, Objects and serialization 资产、对象和序列化

This chapter covers the deep internals of Unity's serialization system and how Unity maintains robust references between different Objects, both in the Unity Editor and at runtime. It also discusses the technical distinctions between Objects and Assets. The topics covered here are fundamental to understanding how to efficiently load and unload Assets in Unity. Proper Asset management is crucial to keeping loading times short and memory usage low.  
本章介绍 Unity 序列化系统的深层内部结构，以及 Unity 如何在 Unity 编辑器和运行时维护不同对象之间的可靠引用。它还讨论了对象和资产之间的技术区别。此处介绍的主题对于了解如何在 Unity 中高效加载和卸载资源至关重要。适当的资产管理对于保持较短的加载时间和较低的内存使用量至关重要。

## 2.1. Inside Assets and Objects 内部资产和对象

To understand how to properly manage data in Unity, it is important to understand how Unity identifies and serializes data. The first key point is the distinction between ***Assets*** and ***UnityEngine.Objects***.  
要了解如何在 Unity 中正确管理数据，了解 Unity 如何识别和序列化数据非常重要。第一个关键点是 ***Assets*** 和 ***UnityEngine.Objects*** 之间的区别。

An ***Asset*** is a file on disk, stored in the *Assets* folder of a Unity project. Textures, 3D models, or audio clips are common types of Assets. Some Assets contain data in formats native to Unity, such as materials. Other Assets need to be processed into native formats, such as FBX files.  
***Asset*** 是磁盘上的文件，存储在 Unity 项目的 *“Assets*” 文件夹中。纹理、3D 模型或音频剪辑是常见的资源类型。某些资源包含 Unity 原生格式的数据，例如材质。其他资产需要处理为本机格式，例如 FBX 文件。

A ***UnityEngine.Object***, or ***Object*** with a capitalized 'O', is a set of serialized data collectively describing a specific instance of a resource. This can be any type of resource which the Unity Engine uses, such as a mesh, sprite, AudioClip or AnimationClip. All Objects are subclasses of the [UnityEngine.Object](http://docs.unity3d.com/ScriptReference/Object.html) base class.  
***UnityEngine.Object*** 或 带有大写“O”的对象是一组序列化数据，共同描述资源的特定实例。这可以是 Unity 引擎使用的任何类型的资源，例如网格、精灵、音频剪辑或动画剪辑。所有对象都是 [UnityEngine.Object](http://docs.unity3d.com/ScriptReference/Object.html) 基类的子类。

While most Object types are built-in, there are two special types.  
虽然大多数对象类型都是内置的，但有两种特殊类型。

1. A [ScriptableObject](http://docs.unity3d.com/ScriptReference/ScriptableObject.html) provides a convenient system for developers to define their own data types. These types can be natively serialized and deserialized by Unity, and manipulated in the Unity Editor's Inspector window.  
   [ScriptableObject](http://docs.unity3d.com/ScriptReference/ScriptableObject.html) 为开发人员提供了一个方便的系统来定义自己的数据类型。这些类型可以由 Unity 本机序列化和反序列化，并在 Unity 编辑器的检查器窗口中进行操作。
2. A [MonoBehaviour](http://docs.unity3d.com/ScriptReference/MonoBehaviour.html) provides a wrapper that links to a [MonoScript](http://docs.unity3d.com/ScriptReference/MonoScript.html). A MonoScript is an internal data type that Unity uses to hold a reference to a specific scripting class within a specific assembly and namespace. The MonoScript does *not* contain any actual executable code.  
   MonoBehavior提供了一个链接到[MonoScript](http://docs.unity3d.com/ScriptReference/MonoBehaviour.html)的包装器。MonoScript 是一种内部数据类型，Unity 使用它来保存对特定程序集和命名空间中特定脚本类的引用。MonoScript不包含任何实际的可执行代码。

There is a one-to-many relationship between Assets and Objects; that is, any given Asset file contains one or more Objects.  
资产和对象之间存在一对多关系，也就是说，任何给定的资源文件都包含一个或多个对象。

2.2. Inter-Object references  
对象间引用

All UnityEngine.Objects can have references to other UnityEngine.Objects. These other Objects may reside within the same Asset file, or may be imported from other Asset files. For example, a material Object usually has one or more references to texture Objects. These texture Objects are generally imported from one or more texture Asset files (such as PNGs or JPGs).  
所有 UnityEngine.Objects 都可以引用其他 UnityEngine.Objects。这些其他对象可能位于同一资源文件中，也可以从其他资源文件导入。例如，材质对象通常具有对纹理对象的一个或多个引用。这些纹理对象通常是从一个或多个纹理资源文件（如 PNG 或 JPG）导入的。

When serialized, these references consist of two separate pieces of data: a ***File GUID*** and a ***Local ID***. The File GUID identifies the Asset file where the target resource is stored. A locally unique Local ID identifies each Object within an Asset file because an Asset file may contain multiple Objects. (Note: AA Local ID is unique from all the other Local IDs for the same Asset file.)  
序列化时，这些引用由两个单独的数据片段组成： ***文件 GUID*** 和 ***本地 ID***。文件 GUID 标识存储目标资源的资产文件。本地 ID 是本地唯一的，它标识资产文件中的每个对象，因为单个资产文件可能包含多个对象。（注意：本地 ID 在同一资源文件中都是互不相同的。）

File GUIDs are stored in .meta files. These .meta files are generated when Unity first imports an Asset, and are stored in the same directory as the Asset.  
文件 GUID 存储在 meta 文件中。这些 .meta 文件是在 Unity 首次导入资源时生成的，并存储在与资源相同的目录中。

The above identification and referencing system can be seen in a text editor: create a fresh Unity project and change its Editor Settings to expose Visible Meta Files and to serialize Assets as text. Create a material and import a texture into the project. Assign the material to a cube in the scene and save the scene.  
上述识别和引用系统可以在文本编辑器中看到：创建一个新的 Unity 项目并更改其编辑器设置为显示 Meta 文件和将 Asset 序列化为文本。创建材质并将纹理导入到项目中。将材质分配给场景中的立方体并保存场景。

Using a text editor, open the .meta file associated with the material. A line labeled "guid" will appear near the top of the file. This line defines the material Asset's File GUID. To find the Local ID, open the material file in a text editor. The material Object's definition will look like this:  
使用文本编辑器打开与材料关联的 .meta 文件。标有“guid”的行将显示在文件顶部附近。此行定义材料资产的文件 GUID。要查找本地 ID，请在文本编辑器中打开材质文件。材质对象的定义如下所示：

--- !u!21 &2100000

Material:

serializedVersion: 3

... more data …

In the above example, the number preceded by an ampersand is the material's Local ID. If this material Object were located inside an Asset identified by the File GUID "abcdefg", then the material Object could be uniquely identified as the combination of the File GUID "abcdefg" and the Local ID "2100000".  
在上面的示例中，&符号前面的数字是材料的本地 ID。如果此材料对象位于由文件 GUID “abcdefg” 标识的资产中，则可以将材料对象唯一标识为文件 GUID “abcdefg” 和本地 ID “2100000” 的组合。

## 2.3. Why File GUIDs and Local IDs? 为什么需要文件 GUID 和本地 ID？

Why is Unity's File GUID and Local ID system necessary? The answer is robustness and to provide a flexible, platform-independent workflow.  
为什么需要 Unity 的文件 GUID 和本地 ID 系统？答案是稳健性，并提供灵活的、独立于平台的工作流程。

The File GUID provides an abstraction of a file's specific location. As long as a specific File GUID can be associated with a specific file, that file's location on disk becomes irrelevant. The file can be freely moved without having to update all Objects referring to the file.  
文件 GUID 提供文件特定位置的抽象。只要特定文件 GUID 可以与特定文件相关联，该文件在磁盘上的位置就变得无关紧要。文件可以自由移动，而无需更新引用该文件的所有对象。

As any given Asset file may contain (or produce via import) multiple UnityEngine.Object resources, a Local ID is required to unambiguously distinguish each distinct Object.  
由于任何给定的资源文件可能包含（或通过导入生成）多个 UnityEngine.Object 资源，因此需要一个本地 ID 来明确区分每个不同的对象。

If the File GUID associated with an Asset file is lost, then references to all Objects in that Asset file will also be lost. This is why it is important that the .meta files must remain stored with the same file names and in the same folders as their associated Asset files. Note that Unity will regenerate deleted or misplaced .meta files.  
如果与资产文件关联的文件 GUID 丢失，则对该资产文件中所有对象的引用也将丢失。这就是为什么 meta 文件必须保持与其关联的资产文件有相同的文件名，而且存储在相同文件夹中的原因。请注意，Unity 将重新生成已删除或放错位置的 meta 文件。

The Unity Editor has a map of specific file paths to known File GUIDs. A map entry is recorded whenever an Asset is loaded or imported. The map entry links the Asset's specific path to the Asset's File GUID. If the Unity Editor is open when a .meta file goes missing and the Asset's path does not change, the Editor can ensure that the Asset retains the same File GUID.  
Unity 编辑器持有已知文件 GUID 和文件路径之间的映射表。每当加载或导入资产时，都会记录映射表条目。映射条目将资产的特定路径链接到资产的文件 GUID 。如果在打开 Unity 编辑器时，meta 文件丢失且资源路径未更改，则编辑器可以确保资源保留相同的文件 GUID。

If the .meta file is lost while the Unity Editor is closed, or the Asset's path changes without the .meta file moving along with the Asset, then all references to Objects within that Asset will be broken.

发生以下情况中任何一种时，对该资源中所有对象的所有引用都将被破坏： 1、在 Unity 编辑器关闭时丢失了 .meta 文件。 2、资源的路径发生了更改，而 .meta 文件没有随资源一起移动。

## 2.4. Composite Assets and importers 复合资产和导入资产

As mentioned in the Inside Assets and Objects section, non-native Asset types must be imported into Unity. This is done via an asset importer. While these importers are usually invoked automatically, they are also exposed to scripts via the [AssetImporter](http://docs.unity3d.com/ScriptReference/AssetImporter.html) API. For example, the [TextureImporter](http://docs.unity3d.com/ScriptReference/TextureImporter.html) API provides access to the settings used when importing individual texture Assets, such as PNG files.  
如内部资源和对象部分所述，非原生资源类型必须导入到 Unity 中。这是通过资产导入器完成的。虽然这些导入程序通常是自动调用的，但它们也会通过 [AssetImporter](http://docs.unity3d.com/ScriptReference/AssetImporter.html) API 向脚本公开。例如，[TextureImporter](http://docs.unity3d.com/ScriptReference/TextureImporter.html) API 提供对导入单个纹理资源（如 PNG 文件）时使用的设置的访问。

The result of the import process is one or more UnityEngine.Objects. These are visible in the Unity Editor as multiple sub-assets within the parent Asset, such as multiple sprites nested beneath a texture Asset that has been imported as a sprite atlas. Each of these Objects will share a File GUID as their source data is stored within the same Asset file. They will be distinguished within the imported texture Asset by a Local ID.  
导入过程的结果是一个或多个 UnityEngine.Objects。这些在 Unity 编辑器中显示为父资源中的多个子资源，例如嵌套在已作为精灵图集导入的纹理资源下的多个精灵。所有子资源对象都将共享一个文件 GUID，因为它们的源数据存储在同一资产文件中。它们将在导入的纹理资源中通过本地 ID 进行区分。

The import process converts source Assets into formats suitable for the target platform selected in the Unity Editor. The import process can include a number of heavyweight operations, such as texture compression. As this is often a time-consuming process, imported Assets are cached in the *Library* folder, eliminating the need to re-import Assets again on the next Editor launch.  
导入过程会将源资源转换为适合 Unity 编辑器所选目标平台的格式。导入过程可以包括许多重量级操作，例如纹理压缩。由于这通常是一个耗时的过程，因此导入的资源将缓存在 *Library* 文件夹中，无需在下次启动编辑器时再次重新导入资源。

Specifically, the results of the import process are stored in a folder named for the first two digits of the Asset's File GUID. This folder is stored inside the *Library/metadata/* folder. The individual Objects from the Asset are serialized into a single binary file that has a name identical to the Asset's File GUID.  
具体而言，导入过程的结果存储在以资产的文件 GUID 的前两位数字命名的文件夹中。此文件夹存储在 *Library/metadata/* 文件夹中。该资产中的所有对象将序列化为单个二进制文件，该文件的名称与资产的文件 GUID 相同。

This process applies to *all* Assets, not just non-native Assets. Native assets do not require lengthy conversion processes or re-serialization.  
此过程适用于所有资产，而不仅仅是非原生资产。本机资产不需要冗长的转换过程或重新序列化。

## 2.5. Serialization and instances 序列化和实例

While File GUIDs and Local IDs are robust, GUID comparisons are slow and a more performant system is needed at runtime. Unity internally maintains a cache that translates File GUIDs and Local IDs into simple, session-unique integers (note: Internally, this cache is called the PersistentManager.) These integers are called Instance IDs, and are assigned in a simple, monotonically-increasing order when new Objects are registered with the cache.  
虽然文件 GUID 和本地 ID 可靠，但 GUID 的比较操作速度很慢，运行时需要性能更高的系统。Unity 在内部维护一个缓存，用于将文件 GUID 和本地 ID 转换为简单的会话唯一整数（注意：在内部，此缓存称为持久管理器）。这些整数称为实例 ID，在向缓存注册新对象时以简单的单调递增顺序分配。

The cache maintains mappings between a given Instance ID, File GUID and Local ID defining the location of the Object's source data, and the instance of the Object in memory (if any). This allows UnityEngine.Objects to robustly maintain references to each other. Resolving an Instance ID reference can quickly return the loaded Object represented by the Instance ID. If the target Object is not yet loaded, the File GUID and Local ID can be resolved to the Object's source data, allowing Unity to load the object just-in-time.  
缓存维护 “实例 ID” => “文件 GUID + 本地 ID” 之间的映射，定义对象的源数据的位置以及对象在内存中的实例（如果有）。这允许 UnityEngine.Objects 可靠地维护彼此之间的引用。解析实例 ID 引用可以快速返回由实例 ID 表示的加载对象。如果尚未加载目标对象，则可以将文件 GUID 和本地 ID 解析为对象的源数据，从而允许 Unity 实时加载对象。

At startup, the Instance ID cache is initialized with data for all Objects immediately required by the project (i.e., referenced in built Scenes), as well as all Objects contained in the Resources folder. Additional entries are added to the cache when new assets are imported at runtime and when Objects are loaded from AssetBundles (note: An example of an Asset created at runtime would be a Texture2D Object created in script, like so: var myTexture = new Texture2D(1024, 768);). Instance ID entries are only removed from the cache when an AssetBundle providing access to a specific File GUID and Local ID is unloaded. When this occurs, the mapping between the Instance ID, its File GUID and Local ID are deleted to conserve memory. If the AssetBundle is re-loaded, a *new* Instance ID will be created for each Object loaded from the re-loaded AssetBundle.  
启动时，项目立即需要的所有对象（即在构建场景中引用的对象）以及 Resources 文件夹中包含的所有对象的数据都会进行初始化以获得实例 ID 缓存。在运行时导入新资产以及从 AssetBundles 加载对象时，会将其他条目添加到缓存中（注意：在运行时创建的资产的一个示例是在脚本中创建的 Texture2D 对象，如下所示：var myTexture = new Texture2D(1024, 768); )。仅当卸载提供对特定文件 GUID 和本地 ID 的访问权限的资产包时，才会从缓存中删除实例 ID 条目。发生这种情况时，将删除 “实例 ID” => “其文件 GUID + 本地 ID” 之间的映射以节省内存。如果重新加载资产包，将为从重新加载的资产包加载的每个对象创建一个新的实例 ID。

For a deeper discussion of the implications of unloading AssetBundles, see the **Managing Loaded Assets** section in the **AssetBundle Usage Patterns** step.  
有关卸载资产包的影响的更深入讨论，请参阅资产**包使用模式**步骤中的**管理已加载资产**部分。

On specific platforms, certain events can force Objects out of memory. For example, graphical Assets can be unloaded from graphics memory on iOS when an app is suspended. If these Objects originated in an AssetBundle that has been unloaded, Unity will be unable to reload the source data for the Objects. Any extant references to these Objects will also be invalid. In the preceding example, the scene may appear to have invisible meshes or magenta textures.   
在特定平台上，某些事件可能会强制将对象从内存中清理掉。例如：在iOS 上，当应用程序被挂起时，系统可能会清理显存中的图形资源。如果这些对象源对应的 AssetBundle 已经被卸载，Unity 将无法重新加载对象的源数据，对这些对象的任何现有引用也将无效。在前面的示例中，场景可能看起来具有不可见的网格或洋红色纹理

*Implementation note:* At runtime, the above control flow is not literally accurate. Comparing File GUIDs and Local IDs at runtime would not be sufficiently performant during heavy loading operations. When building a Unity project, the File GUIDs and Local IDs are deterministically mapped into a simpler format. However, the concept remains identical, and thinking in terms of File GUIDs and Local IDs remains a useful analogy during runtime. This is also the reason why Asset File GUIDs cannot be queried at runtime.  
*实现说明：* 在运行时，上述控制流在字面上并不准确。在繁重的加载操作期间，在运行时比较文件 GUID 和本地 ID 的性能不足。因此生成 Unity 项目时，文件 GUID 和本地 ID 将确定性地映射到更简单的格式。但是，概念保持不变，在运行时，从文件 GUID 和本地 ID 的角度思考仍然是一个有用的类比。这也是无法在运行时查询资产文件 GUID 的原因。

## 2.6. MonoScripts

It is important to understand that a MonoBehaviour has a reference to a MonoScript, and MonoScripts simply contain the information needed to *locate* a specific script class. Neither type of Object contains the executable code of script class.  
重要的是要了解 MonoBehavior 具有对 MonoScript 的引用，而 MonoScripts 仅包含 *查找* 特定脚本类所需的信息。两种类型的对象都不包含脚本类的可执行代码。

A MonoScript contains three strings: assembly name, class name, and namespace.  
MonoScript 包含三个字符串：程序集名称、类名称和命名空间。

While building a project, Unity compiles all the loose script files in the Assets folder into Mono assemblies. C# scripts outside of the *Plugins* subfolder are placed into *Assembly-CSharp.dll*. Scripts within the *Plugins* subfolder are placed into *Assembly-CSharp-firstpass.dll*, and so on. In addition, Unity 2017.3 also introduces the ability to [define custom managed assemblies](https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html).  
在构建项目时，Unity 会将 Assets 文件夹中的所有松散脚本文件编译为 Mono 程序集。*Plugins*子文件夹之外的 C# 脚本被放置在 *Assembly-CSharp.dll* 中。*Plugins*子文件夹中的脚本被放置在*Assembly-CSharp-firstpass.dll*中，依此类推。此外，Unity 2017.3 还引入了[定义自定义托管程序集](https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html)的功能。

These assemblies, as well as pre-built assembly DLL files, are included in the final build of a Unity application. They are also the assemblies to which a MonoScript refers. Unlike other resources, all assemblies included in a Unity application are loaded on application start-up.  
这些程序集以及预构建的程序集 DLL 文件包含在 Unity 应用程序的最终版本中。它们也是 MonoScript 引用的程序集。与其他资源不同，Unity 应用程序中包含的所有程序集都在应用程序启动时加载。

This MonoScript Object is the reason why an AssetBundle (or a Scene or a prefab) does not actually contain executable code in any of the MonoBehaviour Components in the AssetBundle, Scene or prefab. This allows different MonoBehaviours to refer to specific shared classes, even if the MonoBehaviours are in different AssetBundles.  
MonoScript 对象是 AssetBundle（或场景或预制件）实际上不包含其中任何 MonoBehavior 组件中的可执行代码的原因。这允许不同的 MonoBehavior 引用特定的共享类，即使 MonoBehavior 位于不同的 AssetBundle 中也是如此。

## 2.7. Resource lifecycle 资源生命周期

To reduce loading times and manage an application's memory footprint, it's important to understand the resource lifecycle of UnityEngine.Objects. Objects are loaded into/unloaded from memory at specific and defined times.  
为了减少加载时间并管理应用程序的内存占用，了解 UnityEngine.Object 的资源生命周期非常重要。对象在特定和定义的时间加载到内存中或者从内存中卸载。

An Object is loaded automatically when:  
在以下情况同时满足时，自动加载对象：

1. The Instance ID mapped to that Object is dereferenced  
   映射到该对象的实例 ID 被使用
2. The Object is currently not loaded into memory  
   对象当前未加载到内存中
3. The Object's source data can be located.  
   可以找到对象的源数据。

Objects can also be explicitly loaded in scripts, either by creating them or by calling a resource-loading API (e.g., [AssetBundle.LoadAsset](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadAsset.html)). When an Object is loaded, Unity tries to resolve any references by translating each reference's File GUID and Local ID into an Instance ID. An Object will be loaded on-demand the first time its Instance ID is dereferenced if two criteria are true:  
也可以在脚本中显式加载对象，通过创建它们或调用资源加载 API（例如， [AssetBundle.LoadAsset](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadAsset.html)）。加载对象时，Unity 会尝试通过将每个引用的文件 GUID 和本地 ID 转换为实例 ID 来解析任何引用。如果（同时）满足以下两个条件，那么对象将在其实例 ID 首次被使用时按需加载：

1. The Instance ID references an Object that is not currently loaded  
   实例 ID 引用的对象当前还未加载
2. The Instance ID has a valid File GUID and Local ID registered in the cache  
   实例 ID 具有在缓存中注册的有效文件 GUID 和本地 ID

This generally occurs very shortly after the reference itself is loaded and resolved.  
这通常会在引用本身被加载和解析后不久发生。

If a File GUID and Local ID do not have an Instance ID, or if an Instance ID with an unloaded Object references an invalid File GUID and Local ID, then the reference is preserved but the actual Object will not be loaded. This appears as a "(Missing)" reference in the Unity Editor. In a running application, or in the Scene View, "(Missing)" Objects will be visible in different ways, depending on their types. For example, meshes will appear to be invisible, while textures may appear to be magenta.  
如果文件 GUID 和本地 ID 没有实例 ID，或者实例 ID 所引用的对象尚未卸载，但是对象对应了无效的文件 GUID 和本地 ID，则会保留引用，但不会加载实际对象。这在 Unity 编辑器中显示为“（缺失）”引用。在正在运行的应用程序或场景视图中，“（缺失）”对象将以不同的方式显示，具体取决于其类型。例如，网格看起来不可见，而纹理可能显示为洋红色。

Objects are unloaded in three specific scenarios:  
在以下三种特定情况中，对象将被卸载：

1. Objects are automatically unloaded when unused Asset cleanup occurs. This process is triggered automatically when scenes are changed destructively (i.e. when [SceneManager.LoadScene](https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html) is invoked non-additively), or when a script invokes the [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) API. This process only unloads unreferenced Objects; an Object will only be unloaded if no Mono variable holds a reference to the Object, and there are no other live Objects holding references to the Object. Furthermore, note that anything marked with [HideFlags.DontUnloadUnusedAsset](https://docs.unity3d.com/ScriptReference/HideFlags.DontUnloadUnusedAsset.html) and [HideFlags.HideAndDontSave](https://docs.unity3d.com/ScriptReference/HideFlags.HideAndDontSave.html) will not be unloaded.  
   清理未使用的资产时，会自动卸载对象。当场景被破坏性地更改时（即，当 [SceneManager.LoadScene](https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html) 以非累加方式调用时），或者当脚本调用[Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) API 时，此过程会自动触发。此过程仅卸载未引用的对象;仅当没有 Mono 变量保存对该对象的引用，并且没有其他活动对象保存对该对象的引用时，才会卸载对象。此外，请注意，任何标有 [HideFlags.DontUnloadUnusedAsset](https://docs.unity3d.com/ScriptReference/HideFlags.DontUnloadUnusedAsset.html) 和 [HideFlags.HideAndDontSave](https://docs.unity3d.com/ScriptReference/HideFlags.HideAndDontSave.html) 的内容都不会被卸载。

* Objects sourced from the Resources folder can be explicitly unloaded by invoking the [Resources.UnloadAsset](http://docs.unity3d.com/ScriptReference/Resources.UnloadAsset.html) API. The Instance ID for these Objects remains valid and will still contain a valid File GUID and LocalID entry. If any Mono variable or other Object holds a reference to an Object that is unloaded with [Resources.UnloadAsset](http://docs.unity3d.com/ScriptReference/Resources.UnloadAsset.html), then that Object will be reloaded as soon as any of the live references are dereferenced.  
  来自 Resources 文件夹的对象可以通过调用 [Resources.UnloadAsset](http://docs.unity3d.com/ScriptReference/Resources.UnloadAsset.html) API 显式卸载。这些对象的实例 ID 仍然有效，并且仍将包含有效的文件 GUID 和 LocalID 条目。如果任何 Mono 变量或其他 Object 包含对使用 [Resources.UnloadAsset](http://docs.unity3d.com/ScriptReference/Resources.UnloadAsset.html) 卸载的对象的引用，则一旦使用任何实时引用，该对象将立即重新加载。
* Objects sourced from AssetBundles are automatically and immediately unloaded when invoking the [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html)(true) API. This invalidates the File GUID and Local ID of the Object's Instance ID, and any live references to the unloaded Objects will become "(Missing)" references. From C# scripts, attempting to access methods or properties on an unloaded object will produce a *NullReferenceException*.  
  来自 AssetBundle 的对象在调用 [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html)（true） API 时会自动立即卸载。这将使对象的实例 ID 的文件 GUID 和本地 ID 失效，并且对已卸载对象的任何实时引用都将成为“（缺失）”引用。在 C# 脚本中，尝试访问卸载对象上的方法或属性将生成 *NullReferenceException*。

If [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html)(false) is called, live Objects sourced from the unloaded AssetBundle will not be destroyed, but Unity will invalidate the File GUID and Local ID references of their Instance IDs. It will be impossible for Unity to reload these Objects if they are later unloaded from memory and live references to the unloaded Objects remain.   
如果调用 [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html)（false），则不会销毁来自已卸载的资源包的实时对象，但 Unity 将使其实例 ID 的文件 GUID 和本地 ID 引用失效。如果稍后从内存中卸载这些对象，并且保留对已卸载对象的实时引用，Unity 将无法重新加载这些对象。

(Note: The most common case where Objects are removed from memory at runtime without being unloaded occurs when Unity loses control of its graphics context. This may occur when a mobile app is suspended and the app is forced into the background. In this case, the mobile OS usually evicts all graphical resources from GPU memory. When the app returns to the foreground, Unity must reload all needed Textures, Shaders and Meshes to the GPU before scene rendering can resume.)  
（注意：在运行时从内存中删除对象而不卸载的最常见情况发生在 Unity 失去对其图形上下文的控制时。当移动应用挂起并且应用被强制进入后台时，可能会发生这种情况。在这种情况下，移动操作系统通常会从 GPU 内存中逐出所有图形资源。当应用返回到前台时，Unity 必须将所有需要的纹理、着色器和网格重新加载到 GPU，然后才能恢复场景渲染。

## 2.8. Loading large hierarchies 加载大型层次结构

When serializing hierarchies of Unity GameObjects, such as during prefabs serialization, it is important to remember that the entire hierarchy will be fully serialized. That is, every GameObject and Component in the hierarchy will be individually represented in the serialized data. This has interesting impacts on the time required to load and instantiate hierarchies of GameObjects.

When creating any GameObject hierarchy, CPU time is spent in several different ways:  
序列化 Unity 游戏对象的层次结构时（例如在预制件序列化期间），请务必记住，整个层次结构将被完全序列化。也就是说，层次结构中的每个游戏对象和组件都将在序列化数据中单独表示。这对加载和实例化游戏对象层次结构所需的时间产生了有趣的影响。

创建任何游戏对象层次结构时，CPU 时间以几种不同的方式被消耗：

* Reading the source data (from storage, from an AssetBundle, from another GameObject, etc.)  
  读取源数据（从存储、资产包、另一个游戏对象等）
* Setting up the parent-child relationships between the new Transforms  
  设置新 Transform 之间的父子关系
* Instantiating the new GameObjects and Components  
  实例化新的游戏对象和组件
* Awakening the new GameObjects and Components on the main thread  
  在主线程上唤醒新游戏对象和组件

The latter three time costs are generally invariant regardless of whether the hierarchy is being cloned from an existing hierarchy or is being loaded from storage. However, the time to read the source data increases linearly with the number of Components and GameObjects serialized into the hierarchy, and is also multiplied by the speed of the data source.  
后三种时间成本通常是不变的，无论层次结构是从现有层次结构克隆还是从存储加载。但是，读取源数据的时间随着序列化到层次结构中的组件和游戏对象的数量而线性增加，并且还会乘以数据源的速度。

On all current platforms, it is considerably faster to read data from elsewhere in memory rather than loading it from a storage device. Further, the performance characteristics of the available storage media vary widely between different platforms. Therefore, when loading prefabs on platforms with slow storage, the time spent reading the prefab's serialized data from storage can rapidly exceed the time spent instantiating the prefab. That is, the cost of the loading operation is bound to storage I/O time.  
在所有当前平台上，从内存中的其他位置读取数据比从存储设备加载数据要快得多。此外，可用存储介质的性能特征在不同平台之间差异很大。因此，在存储速度较慢的平台上加载预制件时，从存储中读取预制件序列化数据所花费的时间可能会迅速超过实例化预制件所花费的时间。也就是说，加载操作的成本与存储 I/O 时间绑定。

As mentioned before, when serializing a monolithic prefab, every GameObject and component's data is serialized separately, which may duplicate data. For example, a UI screen with 30 identical elements will have the identical element serialized 30 times, producing a large blob of binary data. At load time, the data for all of the GameObjects and Components on each one of those 30 duplicate elements must be read from disk before being transferred to the newly-instantiated Object. This file reading time is a significant contributor to the overall cost of instantiating large prefabs. Large hierarchies should be instantiated in modular chunks, and then be stitched together at runtime.  
如前所述，在序列化整体预制件时，每个游戏对象和组件的数据都是单独序列化的，这可能会重复数据。例如，具有 30 个相同元素的 UI 屏幕将序列化相同的元素 30 次，从而生成大量二进制数据。在加载时，这 30 个重复元素中每个元素上的所有游戏对象和组件的数据必须先从磁盘读取，然后才能传输到新实例化的对象。此文件读取时间是实例化大型预制件的总体成本的重要因素。大型层次结构应在模块化块中实例化，然后在运行时拼接在一起。

*Unity 5.4 note:* Unity 5.4 altered the representation of transforms in memory. Each root transform's entire child hierarchy is stored in compact, contiguous regions of memory. When instantiating new GameObjects that will be instantly reparented into another hierarchy, consider using the new [GameObject.Instantiate](https://docs.unity3d.com/ScriptReference/Object.Instantiate.html) overloaded variants which accept a parent argument. Using this overload avoids the allocation of a root transform hierarchy for the new GameObject. In tests, this speeds up the time required for an instantiate operation by about 5-10%.  
*Unity 5.4 注意：*Unity 5.4 更改了内存中 Transform 的表示形式。每个 root transform的整个子层次结构都存储在紧凑的连续内存区域中。实例化将立即重定到另一个层次结构中的新游戏对象时，请考虑使用[GameObject.Instantiate](https://docs.unity3d.com/ScriptReference/Object.Instantiate.html)(parent)。使用此重载可避免为新游戏对象分配根转换层次结构。在测试中，这将实例化操作所需的时间加快了大约 5-10%。

# 3.The Resources folder

This chapter discusses the *Resources* system. This is the system that allows developers to store Assets within one or more folders named *Resources* and to load or unload Objects from those Assets at runtime using the [Resources](http://docs.unity3d.com/ScriptReference/Resources.html) API.  
本章讨论 *Resources* 系统。该系统允许开发人员将资产存储在一个或多个名为 *Resources*的文件夹中，并在运行时使用[Resources](http://docs.unity3d.com/ScriptReference/Resources.html) API 从这些资产加载或卸载对象。

## 3.1. Best Practices for the Resources System Resources系统的最佳实践

***Don't use it.  
不要使用它。***

This strong recommendation is made for several reasons:  
提出这一强烈建议有几个原因：

* Use of the Resources folder makes fine-grained memory management more difficult  
  使用 Resources 文件夹使细粒度内存管理更加困难
* Improper use of Resources folders will increase application startup time and the length of builds  
  使用不当的话，Resources文件夹会增加应用程序的启动时间和构建时间
* As the number of Resources folders increases, management of the Assets within those folders becomes very difficult  
  随着Resources文件夹数量的增加，管理这些文件夹中的资产变得非常困难
* The Resources system degrades a project's ability to deliver custom content to specific platforms and eliminates the possibility of incremental content upgrades  
  Resources系统降低了项目向特定平台交付自定义内容的能力，并消除了增量内容升级的可能性
* AssetBundle Variants are Unity's primary tool for adjusting content on a per-device basis  
  AssetBundle变体是 Unity 用于按设备调整内容的主要工具

## 3.2. Proper uses of the Resources system 正确使用Resources系统

There are two specific use cases where the Resources system can be helpful without impeding good development practices:  
在两个特定的用例中，Resource系统可以在不妨碍良好开发实践的情况下提供帮助：

1. The ease of the Resources folder makes it an excellent system to rapidly prototype. However, when a project moves into full production, the use of the Resources folder should be eliminated.  
   Resources文件夹的易用性使其成为快速原型的绝佳系统。但是，当项目进入完全生产时，应消除对Resources文件夹的使用。
2. The Resources folder may be useful in some trivial cases, if the content is:  
   Resources文件夹在某些微不足道的情况下可能很有用，如果内容是：
   1. Generally required throughout a project's lifetime  
      通常在项目的整个生命周期内都需要
   2. Not memory-intensive  
      不占用大量内存
   3. Not prone to patching, or does not vary across platforms or devices  
      不易打补丁，或者不同平台或设备没有差异
   4. Used for minimal bootstrapping  
      用于最少的引导

Examples of this second case include MonoBehaviour singletons used to host prefabs, or ScriptableObjects containing third-party configuration data, such as a Facebook App ID.  
第二种情况的示例包括用于管理预制件的 MonoBehavior 单例，或包含第三方配置数据（如 Facebook 应用 ID）的 ScriptableObjects。

## 3.3. Serialization of Resources Resources的序列化

The Assets and Objects in all folders named "Resources" are combined into a single serialized file when a project is built. This file also contains metadata and indexing information, similar to an AssetBundle. As described in the [AssetBundle documentation](https://docs.unity3d.com/Manual/AssetBundlesIntro.html), this index includes a serialized lookup tree that is used to resolve a given Object's name into its appropriate File GUID and Local ID. It is also used to locate the Object at a specific byte offset in the serialized file's body.  
生成项目时，名为 Resources 的所有文件夹中的资产和对象将合并到单个序列化文件中。此文件还包含元数据和索引信息，类似于资产包。如 [AssetBundle 文档](https://docs.unity3d.com/Manual/AssetBundlesIntro.html)中所述，此索引包括一个序列化查找树，用于将给定对象的名称解析为其相应的文件 GUID 和本地 ID。它还用于在序列化文件正文中的特定字节偏移量处定位 Object。

On most platforms, the lookup data structure is a balanced search tree, which has a construction time that grows at an O(n log(n)) rate. This growth also causes the index's loading time to grow more-than-linearly as the number of Objects in Resources folders increases.  
在大多数平台上，查找数据结构是一个平衡的搜索树，其构建时间以 O（n log（n）） 速率增长。这种增长还会导致索引的加载时间随着Resources文件夹中的对象数量的增加而线性增长。

This operation is unskippable and occurs at application startup time while the initial non-interactive splash screen is displayed. Initializing a Resources system containing 10,000 assets has been observed to consume multiple seconds on low-end mobile devices, even though most of the Objects contained in Resources folders are rarely actually needed to load into an application's first scene.  
此操作是不可跳过的，在应用程序启动时发生，同时显示初始非交互式初始屏幕。据观察，初始化包含 10，000 个资产的Resources系统在低端移动设备上会消耗数秒，即使 Resources 文件夹中包含的大多数对象实际上很少需要加载到应用程序的第一个场景中。

# 4.AssetBundle fundamentals AssetBundle基础知识

This chapter discusses AssetBundles. It introduces the fundamental systems upon which AssetBundles are built, as well as the core APIs used to interact with AssetBundles. In particular, it discusses both the loading and unloading of AssetBundles themselves, as well as the loading and unloading of specific Asset and Objects from AssetBundles.

For more patterns and best practices on the uses of AssetBundles, see the next chapter in this series.  
本章讨论AssetBundle。它介绍了构建AssetBundle的基本系统，以及用于与AssetBundle交互的核心API。特别是，它讨论了AssetBundle本身的加载和卸载，以及从AssetBundle中加载和卸载特定资产和对象。

有关使用AssetBundle的更多模式和最佳实践，请参阅本系列的下一章。

## 4.1. Overview

The AssetBundle system provides a method for storing one or more files in an archival format that Unity can index and serialize. AssetBundles are Unity's primary tool for the delivery and updating of non-code content after installation. This permits developers to submit a smaller app package, minimize runtime memory pressure, and selectively load content optimized for the end-user's device.  
资源包系统提供了一种以archival format存储一个或多个文件的方法，Unity 可以对其进行索引和序列化。资源包是 Unity 在安装后交付和更新非代码内容的主要工具。这允许开发人员提交较小的应用包，最大程度地减少运行时内存压力，并有选择地加载针对最终用户设备优化的内容。

Understanding the way AssetBundles work is essential to building a successful Unity project for mobile devices. For an overall description of AssetBundle contents, review the [AssetBundle documentation](https://docs.unity3d.com/Manual/AssetBundlesIntro.html).  
了解资源包的工作方式对于为移动设备构建成功的 Unity 项目至关重要。有关资产包内容的总体描述，请查看 [AssetBundle documentation](https://docs.unity3d.com/Manual/AssetBundlesIntro.html)。

## 4.2. AssetBundle layout

To summarize, an AssetBundle consists of two parts: a header and data segment.  
总而言之，资产捆绑包由两部分组成：标题和数据段。

The header contains information about the AssetBundle, such as its identifier, compression type, and a manifest. The manifest is a lookup table keyed by an Object's name. Each entry provides a byte index that indicates where a given Object can be found within the AssetBundle's data segment. On most platforms, this lookup table is implemented as a balanced search tree. Specifically, Windows and OSX-derived platforms (including iOS) employ a red-black tree. Therefore, the time needed to construct the manifest will increase *more than linearly* as the number of Assets within an AssetBundle grows.  
标头包含有关资产捆绑包的信息，例如其标识符、压缩类型和清单。清单是一个由对象名称键入的查找表。每个条目都提供一个字节索引，指示在资产包的数据段中可以找到给定对象的位置。在大多数平台上，此查找表作为平衡搜索树实现。具体来说，Windows和OSX派生的平台（包括iOS）采用了红黑树。因此，构建清单所需的时间将随着资产包中资产数量的增加而*线性增加*。

The data segment contains the raw data generated by serializing the Assets in the AssetBundle. If LZMA is specified as the compression scheme, the complete byte array for all serialized assets is compressed. If LZ4 is instead specified, bytes for separate Assets are individually compressed. If no compression is used, the data segment will remain as raw byte streams.  
数据段包含通过序列化资产包中的资产生成的原始数据。如果将 LZMA 指定为压缩方案，则会压缩所有序列化资产的完整字节数组。如果改为指定 LZ4，则会单独压缩单独资产的字节。如果未使用压缩，数据段将保留为原始字节流。

Prior to Unity 5.3, Objects could not be compressed individually inside an AssetBundle. As a consequence, if a version of Unity before 5.3 is instructed to read one or more Objects from a compressed AssetBundle, Unity had to decompress the entire AssetBundle. Generally, Unity cached a decompressed copy of the AssetBundle to improve loading performance for subsequent loading requests on the same AssetBundle.  
在 Unity 5.3 之前，无法在资源包中单独压缩对象。因此，如果指示 Unity 5.3 之前的版本从压缩的资源包中读取一个或多个对象，Unity 必须解压缩整个资源包。通常，Unity 会缓存资源包的解压缩副本，以提高同一资源包上后续加载请求的加载性能。

## 4.3. Loading AssetBundles

AssetBundles can be loaded via four distinct APIs. The behavior of these four APIs is different depending on two criteria:  
资产包可以通过四个不同的 API 加载。这四个 API 的行为因两个条件而异：

1. Whether the AssetBundle is LZMA compressed, LZ4 compressed or uncompressed  
   资产包是 LZMA 压缩、LZ4 压缩还是未压缩
2. The platform on which the AssetBundle is being loaded  
   加载资产包的平台

These APIs are:

* AssetBundle.LoadFromMemory(Async optional)
* AssetBundle.LoadFromFile(Async optional)
* UnityWebRequest's DownloadHandlerAssetBundle
* WWW.LoadFromCacheOrDownload (on Unity 5.6 or older)

### 4.3.1 AssetBundle.LoadFromMemory(Async)

***Unity's recommendation is not to use this API.  
Unity 建议不要使用此 API。***

[AssetBundle.LoadFromMemoryAsync](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromMemoryAsync.html) loads an AssetBundle from a managed-code byte array (*byte[]* in C#). It will always copy the source data from the managed-code byte array into a newly-allocated, contiguous block of native memory. If the AssetBundle is LZMA compressed, it will decompress the AssetBundle while copying. Uncompressed and LZ4-compressed AssetBundles will be copied verbatim.  
[AssetBundle.LoadFromMemoryAsync](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromMemoryAsync.html) 从托管代码字节数组（C# 中的 *byte[]*）加载 AssetBundle。它将始终将源数据从托管代码字节数组复制到新分配的连续本机内存块中。如果资产包是 LZMA 压缩的，它将在复制时解压缩资产包。未压缩和 LZ4 压缩的资产包将被逐字复制。

The peak amount of memory consumed by this API will be at least twice the size of the AssetBundle: one copy in native memory created by the API, and one copy in the managed byte array passed to the API. Assets loaded from an AssetBundle created via this API will therefore be duplicated *three* times in memory: once in the managed-code byte array, once in the native-memory copy of the AssetBundle and a third time in GPU or system memory for the asset itself.  
此 API 消耗的峰值内存量至少是 AssetBundle 大小的两倍：API 创建的本机内存中的一个副本，以及传递给 API 的托管字节数组中的一个副本。因此，从通过此 API 创建的 AssetBundle 加载的资产将在内存中复制 *三次* ：一次在托管代码字节数组中，一次在 AssetBundle 的本机内存副本中，第三次在资产本身的 GPU 或系统内存中。

*Prior to Unity 5.3.3, this API was known as \*\*AssetBundle.CreateFromMemory\*\*. Its functionality has not changed.*

### 4.3.2. AssetBundle.LoadFromFile(Async)

[AssetBundle.LoadFromFile](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromFile.html) is a highly-efficient API intended for loading uncompressed or LZ4-compressed AssetBundle from local storage, such as a hard disk or an SD card.  
[AssetBundle.LoadFromFile](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromFile.html) 是一个高效的 API，用于从本地存储（如硬盘或 SD 卡）加载未压缩或 LZ4 压缩的资产捆绑包。

On desktop standalone, console, and mobile platforms, the API will only load the AssetBundle's header, and will leave the remaining data on disk. The AssetBundle's Objects will be loaded on-demand as loading methods (e.g. *AssetBundle.Load*) are called or as their InstanceIDs are dereferenced. No excess memory will be consumed in this scenario. In the Unity Editor, the API will load the entire AssetBundle into memory, as if the bytes were read off disk and *AssetBundle.LoadFromMemoryAsync* was used. This API can cause memory spikes to appear during AssetBundle loading if the project is profiled in the Unity Editor. This should not affect performance on-device and these spikes should be re-tested on-device before taking remedial action.  
在桌面独立、控制台和移动平台上，API 将仅加载资产包的标头，并将剩余数据保留在磁盘上。资产包的对象将在调用加载方法（例如 *AssetBundle.Load*）或使用其实例 ID 时按需加载。在这种情况下，不会消耗多余的内存。在 Unity 编辑器中，API 会将整个 AssetBundle 加载到内存中，就像从磁盘读取字节并使用 *AssetBundle.LoadFromMemoryAsync* 一样。如果在 Unity 编辑器中分析项目，则此 API 可能会导致在资源包加载期间出现内存峰值。这应该不会影响设备上的性能，在采取补救措施之前，应在设备上重新测试这些峰值。

*Note:* On Android devices with Unity 5.3 or older, this API will fail when trying to load AssetBundles from the Streaming Assets path. This issue has been resolved in Unity 5.4. For more details, see the section **Distribution - shipped with project** section of the **AssetBundle usage patterns** step.  
*注意：*在装有 Unity 5.3 或更早版本的 Android 设备上，尝试从流资产路径加载资源包时，此 API 将失败。此问题已在 Unity 5.4 中得到解决。有关更多详细信息，请参阅 **AssetBundle 使用模式**步骤的**分发 - 随项目一起提供**部分。

*Prior to Unity 5.3, this API was known as \*\*AssetBundle.CreateFromFile\*\*. Its functionality has not been changed.  
在Unity 5.3之前，此API称为\*\*AssetBundle.CreateFromFile\*\*。其功能未更改。*

### 4.3.3. AssetBundleDownloadHandler

The [UnityWebRequest](http://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html) API allows developers to specify exactly how Unity should handle downloaded data and allows developers to eliminate unnecessary memory usage. The simplest way to download an AssetBundle using UnityWebRequest is to call [UnityWebRequest.GetAssetBundle](http://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.GetAssetBundle.html).  
[UnityWebRequest](http://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html) API 允许开发人员准确指定 Unity 应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 的最简单方法是调用 [UnityWebRequest.GetAssetBundle](http://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.GetAssetBundle.html)。

For the purposes of this guide, the class of interest is [DownloadHandlerAssetBundle](http://docs.unity3d.com/ScriptReference/Networking.DownloadHandlerAssetBundle.html). Using a worker thread, it streams downloaded data into a fixed-size buffer and then spools the buffered data to either temporary storage or the AssetBundle cache, depending on how the Download Handler has been configured. All of these operations occur in native code, eliminating the risk of expanding the managed heap. Additionally, this Download Handler does *not* keep a native-code copy of all downloaded bytes, further reducing the memory overhead of downloading an AssetBundle.  
就本指南而言，感兴趣的类别是 [DownloadHandlerAssetBundle](http://docs.unity3d.com/ScriptReference/Networking.DownloadHandlerAssetBundle.html" \t "_blank)。它使用工作线程将下载的数据流式传输到固定大小的缓冲区，然后将缓冲的数据传到临时存储或 AssetBundle 缓存，具体取决于下载处理程序的配置方式。所有这些操作都在本机代码中发生，从而消除了扩展托管堆的风险。此外，此下载处理程序不会保留所有下载字节的本机代码副本，从而进一步减少了下载 AssetBundle 的内存开销。

LZMA-compressed AssetBundles will be decompressed during download and cached using LZ4 compression. This behavior may be changed by setting [Caching.CompressionEnabled](https://docs.unity3d.com/ScriptReference/Caching-compressionEnabled.html).  
LZMA 压缩的资产包将在下载过程中解压缩并使用 LZ4 压缩进行缓存。可以通过设置[Caching.CompressionEnabled](https://docs.unity3d.com/ScriptReference/Caching-compressionEnabled.html)来更改此行为。

When the download is complete, the [assetBundle](http://docs.unity3d.com/ScriptReference/Networking.DownloadHandlerAssetBundle-assetBundle.html) property of the Download Handler provides access to the downloaded AssetBundle, as if *AssetBundle.LoadFromFile* had been called on the downloaded AssetBundle.  
下载完成后，下载处理程序的 assetBundle 属性提供对下载的资产包的访问，就好像在下载的资产包 上调用了 *AssetBundle.LoadFromFile*。

If caching information is provided to a UnityWebRequest object, and the requested AssetBundle already exists in Unity's cache, then the AssetBundle will become available immediately and this API will operate identically to *AssetBundle.LoadFromFile*.  
如果向 UnityWebRequest 对象提供了缓存信息，并且请求的资产包已存在于 Unity 的缓存中，则资源包将立即可用，并且此 API 的操作方式与 *AssetBundle.LoadFromFile* 相同。

Prior to Unity 5.6, the UnityWebRequest system used a fixed pool of worker threads and an internal job system to safeguard against excessive, concurrent downloads. The size of the thread pool was not configurable. In Unity 5.6, these safeguards have been removed to accommodate more modern hardware, and allow for faster access to HTTP response codes and headers.  
在 Unity 5.6 之前，UnityWebRequest 系统使用固定的工作线程池和内部作业系统来防止过多的并发下载。线程池的大小不可配置。在 Unity 5.6 中，删除了这些保护措施，以适应更现代的硬件，并允许更快地访问 HTTP 响应代码和标头。

### 4.3.4. WWW.LoadFromCacheOrDownload

*Note: Beginning in Unity 2017.1,* [***WWW.LoadFromCacheOrDownload***](http://docs.unity3d.com/ScriptReference/WWW.LoadFromCacheOrDownload.html) *simply wraps around UnityWebRequest. Accordingly, developers using Unity 2017.1 or higher should migrate to UnityWebRequest.* ***WWW.LoadFromCacheOrDownload*** *will be deprecated in a future release.*

*The following information is applicable to Unity 5.6 or older.*

[WWW.LoadFromCacheOrDownload](http://docs.unity3d.com/ScriptReference/WWW.LoadFromCacheOrDownload.html) is an API that allows loading of Objects from both remote servers and local storage. Files can be loaded from local storage via a file:// URL. If the AssetBundle is present in the Unity cache, this API will behave exactly like *AssetBundle.LoadFromFile*.

If an AssetBundle has not yet been cached, then *WWW.LoadFromCacheOrDownload* will read the AssetBundle from its source. If the AssetBundle is compressed, it will be decompressed using a worker thread and written into the cache. Otherwise, it will be written directly into the cache via the worker thread. Once the AssetBundle is cached, *WWW.LoadFromCacheOrDownload* will load header information from the cached, decompressed AssetBundle. The API will then behave identically to an AssetBundle loaded with *AssetBundle.LoadFromFile*. This cache is shared between *WWW.LoadFromCacheOrDownload* and *UnityWebRequest*. Any AssetBundle downloaded with one API will also be available via the other API.

While the data will be decompressed and written to the cache via a fixed-size buffer, the WWW object will keep a full copy of the AssetBundle's bytes in native memory. This extra copy of the AssetBundle is kept to support the [WWW.bytes](http://docs.unity3d.com/ScriptReference/WWW-bytes.html) property.

Due to the memory overhead of caching an AssetBundle's bytes in the WWW object, AssetBundles should remain small - a few megabytes, at most. For more discussion of AssetBundle sizing, see the**Asset assignment strategies** section in the**AssetBundle usage patterns** chapter.

Unlike UnityWebRequest, each call to this API will spawn a new worker thread. Accordingly, on platforms with limited memory, such as mobile devices, only a single AssetBundle at a time should be downloaded using this API, in order to avoid memory spikes. Be careful of creating an excessive number of threads when calling this API multiple times. If more than 5 AssetBundles need to be downloaded, create and manage a download queue in script code to ensure that only a few AssetBundle downloads are running simultaneously.

### 4.3.5. Recommendations

In general, AssetBundle.LoadFromFile should be used whenever possible. This API is the most efficient in terms of speed, disk usage and runtime memory usage.

For projects that must download or patch AssetBundles, it is strongly recommended to useUnityWebRequest for projects using Unity 5.3 or newer, and WWW.LoadFromCacheOrDownload for projects using Unity 5.2 or older. As detailed in the**Distribution** section, it is possible to prime the AssetBundle Cache with Bundles included within a project's installer.  
通常，应尽可能使用 AssetBundle.LoadFromFile。此 API 在速度、磁盘使用情况和运行时内存使用情况方面效率最高。

对于必须下载或修补资源包的项目，强烈建议使用 Unity 5.3 或更高版本，对于 WWW 的项目应该使用 UnityWebRequest。LoadFromCacheOrDownload 适用于使用 Unity 5.2 或更早版本的项目。如 **“分发** ”部分所述，可以使用项目安装程序中包含的捆绑包来启动资产包缓存。

When using either *UnityWebRequest* or *WWW.LoadFromCacheOrDownload*, ensure that the downloader code properly calls *Dispose* after loading the AssetBundle. Alternately, C#'s [using](https://msdn.microsoft.com/en-us/library/yh598w02.aspx) statement is the most convenient way to ensure that a *WWW* or *UnityWebRequest* is safely disposed of.  
无论使用 *UnityWebRequest 还是* *WWW.LoadFromCacheOrDownload*，确保下载程序代码在加载资产包后正确调用*Dispose*。C# 的 [using](https://msdn.microsoft.com/en-us/library/yh598w02.aspx) 语句是确保安全 dispose *WWW* 或 *UnityWebRequest* 的最便捷方法。

For projects with substantial engineering teams that require unique, specific caching or downloading requirements, a custom downloader may be considered. Writing a custom downloader is a non-trivial engineering task, and any custom downloader should be made compatible with *AssetBundle.LoadFromFile.* See the **Distribution** section of the next step for more details.  
对于具有大量工程团队的项目，需要独特的特定缓存或下载要求，可以考虑自定义下载器。编写自定义下载器是一项不平凡的工程任务，任何自定义下载器都应与 *AssetBundle.LoadFromFile 兼容。* 有关更多详细信息，请参阅下一步的**“分发**”部分。

## 4.4. Loading Assets From AssetBundles 从资源包加载资源

UnityEngine.Objects can be loaded from AssetBundles using three distinct APIs that are all attached to the *AssetBundle* object, which have both synchronous and asynchronous variants:  
UnityEngine.Objects可以使用三个不同的API从AssetBundle加载，这些API都是*AssetBundle*对象实例的方法，这些对象具有同步和异步变体：

* [LoadAsset](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAsset.html) ([LoadAssetAsync](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAssetAsync.html))
* [LoadAllAssets](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAllAssets.html) ([LoadAllAssetsAsync](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAllAssetsAsync.html))
* [LoadAssetWithSubAssets](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAssetWithSubAssets.html) ([LoadAssetWithSubAssetsAsync](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadAssetWithSubAssetsAsync.html))

The synchronous versions of these APIs will always be faster than their asynchronous counterpart, by at least one frame.  
这些 API 的同步版本将始终比其异步对应版本快至少一帧。

Asynchronous loads will load multiple Objects per frame, up to their time-slice limits. See the**Low-level loading details** section for the underlying technical reasons for this behavior.  
异步加载将每帧加载多个对象，直至达到其时间片限制。有关此行为的基本技术原因，请参阅**低级别加载详细信息**部分。

*LoadAllAssets* should be used when loading multiple independent UnityEngine.Objects. It should only be used when the majority or all of the Objects within an AssetBundle need to be loaded. Compared to the other two APIs, *LoadAllAssets* is slightly faster than multiple individual calls to *LoadAssets*. Therefore, if the number of assets to be loaded is large, but less than 66% of the AssetBundle needs to be loaded at a single time, consider splitting the AssetBundle into multiple smaller bundles and using *LoadAllAssets*.  
加载多个独立的 UnityEngine.Object 时，应使用 LoadAllAssets。仅当需要加载资产包中的大多数或全部对象时，才应使用它。与其他两个API相比，*LoadAllAssets*比对*LoadAssets*的多次单独调用略快。因此，如果要加载的资产数量很大，但一次需要加载的资产包少于 66%，请考虑将资产包拆分为多个较小的捆绑包并使用 *LoadAllAssets*。

*LoadAssetWithSubAssets* should be used when loading a composite Asset which contains multiple embedded Objects, such as an FBX model with embedded animations or a sprite atlas with multiple sprites embedded inside it. If the Objects that need to be loaded all come from the same Asset, but are stored in an AssetBundle with many other unrelated Objects, then use this API.  
加载包含多个嵌入对象的复合资源时，应使用 LoadAssetWithSubAssets，例如具有嵌入式动画的 FBX 模型或嵌入多个精灵的精灵图集。如果需要加载的对象都来自同一资产，但与许多其他不相关的对象一起存储在资产包中，则使用此 API。

For any other case, use *LoadAsset* or *LoadAssetAsync*.  
对于任何其他情况，请使用 *LoadAsset* 或 *LoadAssetAsync*。

### 4.4.1. Low-level loading details 低级加载详细信息

UnityEngine.Object loading is performed off the main thread: an Object's data is read from storage on a worker thread. Anything which does not touch thread-sensitive parts of the Unity system (scripting, graphics) will be converted on the worker thread. For example, VBOs will be created from meshes, textures will be decompressed, etc.  
UnityEngine.Object 加载是在主线程之外执行的：对象的数据从工作线程上的存储中读取。任何不接触 Unity 系统线程敏感部分（脚本、图形）的内容都将在工作线程上进行转换。例如，将从网格创建 VBO，解压缩纹理等。

From Unity 5.3 onward, Object loading has been parallelized. Multiple Objects are deserialized, processed and integrated on worker threads. When an Object finishes loading, its *Awake* callback will be invoked and the Object will become available to the rest of the Unity Engine during the next frame.  
从 Unity 5.3 开始，对象加载已并行化。多个对象在工作线程上反序列化、处理和集成。对象完成加载后，将调用其 *Awake* 回调，并且该对象将在下一帧期间可供 Unity 引擎的其余部分使用。

The synchronous *AssetBundle.Load* methods will pause the main thread until Object loading is complete. They will also time-slice Object loading so that Object integration does not occupy more than a certain number of milliseconds of frame time. The number of milliseconds is set by the property *Application.backgroundLoadingPriority*:  
*AssetBundle.Load* 的同步方法将暂停主线程，直到对象加载完成。它们还将对对象加载进行时间切片，以便对象集成占用的帧时间不会超过一定毫秒数。毫秒数由属性 *Application.backgroundLoadPriority* 设置：

* *ThreadPriority.High*: Maximum 50 milliseconds per frame
* *ThreadPriority.Normal*: Maximum 10 milliseconds per frame
* *ThreadPriority.BelowNormal*: Maximum 4 milliseconds per frame
* *ThreadPriority.Low*: Maximum 2 milliseconds per frame.

From Unity 5.2 onwards, multiple Objects are loaded until the frame-time limit for Object loading is reached. Assuming all other factors are equal, the asynchronous variants of the asset loading APIs will always take longer to complete than the comparable synchronous version due to the minimum one-frame delay between issuing the asynchronous call and the object becoming available to the Engine.  
从 Unity 5.2 开始，将加载多个对象，直到达到对象加载的帧时间限制。假设所有其他因素都相同，资产加载 API 的异步变体将始终比类似的同步版本花费更长的时间才能完成，因为发出异步调用和对象可供引擎使用之间的至少有一帧延迟。

### 4.4.2. AssetBundle dependencies

The dependencies among AssetBundles are automatically tracked using two different APIs, depending on the runtime environment. In the Unity Editor, AssetBundle dependencies can be queried via the [AssetDatabase](http://docs.unity3d.com/ScriptReference/AssetDatabase.html) API. AssetBundle assignments and dependencies can be accessed and changed via the [AssetImporter](http://docs.unity3d.com/ScriptReference/AssetImporter.html) API. At runtime, Unity provides an optional API to load the dependency information generated during an AssetBundle build via a ScriptableObject-based [AssetBundleManifest](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.html) API.  
资产包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](http://docs.unity3d.com/ScriptReference/AssetDatabase.html) API 查询资产包依赖项。可以通过 [AssetImporter](http://docs.unity3d.com/ScriptReference/AssetImporter.html) API 访问和更改 AssetBundle 分配和依赖项。在运行时，Unity 提供了一个可选的 API，用于通过基于脚本对象的 [AssetBundleManifest](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.html) API 加载在资产捆绑构建期间生成的依赖项信息。

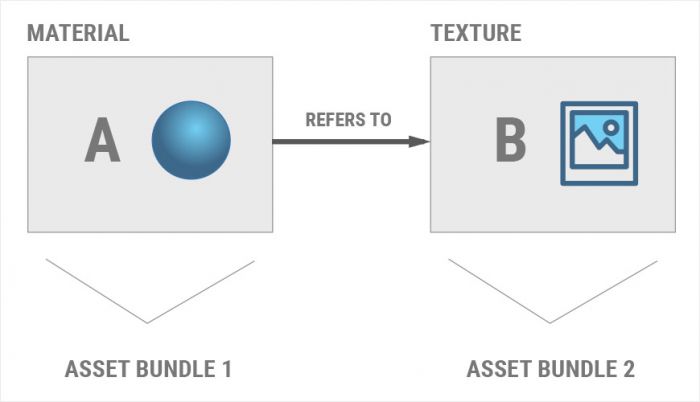
An AssetBundle is dependent upon another AssetBundle when one or more of the parent AssetBundle's UnityEngine.Objects refers to one or more of the other AssetBundle's UnityEngine.Objects. For more information on inter-Object references, see the**Inter-Object references** section of the**Assets, Objects and Serialization** step.  
当一个或多个父资产包的 UnityEngine.Objects 引用一个或多个其他资产包的 UnityEngine.Objects 时，资源包依赖于另一个资产包。有关对象间引用的详细信息，请参阅**资产、对象和序列化**步骤的对象**间引用**部分。

As described in the **Serialization and instances** section of that step, AssetBundles serve as sources for the source data identified by the FileGUID & LocalID of each Object contained within the AssetBundle.  
如该章节的**序列化和实例**部分所述，资产包充当由资产包中包含的每个对象的 FileGUID 和 LocalID 标识的源数据的源。

Because an Object is loaded when its Instance ID is first dereferenced, and because an Object is assigned a valid Instance ID when its AssetBundle is loaded, the order in which AssetBundles are loaded is not important. Instead, it is important to load all AssetBundles that contain dependencies of an Object before loading the Object itself. Unity will not attempt to automatically load any child AssetBundles when a parent AssetBundle is loaded.  
由于对象是在首次使用其实例 ID 时加载的，并且由于在加载对象的资产包时为其分配了有效的实例 ID，因此加载资产包的顺序并不重要。相反，在加载对象本身之前，加载包含对象依赖项的所有资产包非常重要。加载父资源包时，Unity 不会尝试自动加载任何子资源包。

**Example:**

Assume *material A* refers to *texture B*. Material A is packaged into AssetBundle 1, and texture B is packaged into AssetBundle 2.  
假设 *材质 A* 使用 *纹理 B*。材质 A 打包到资产包 1 中，纹理 B 打包到资产包 2 中。



Select image to expand

In this use case, AssetBundle 2 must be loaded *prior* to loading Material A out of AssetBundle 1.  
在此用例中，必须先加载资源包 2，*然后再*从资产包 1 中加载材料 A。

This does not imply that AssetBundle 2 must be loaded before AssetBundle 1, or that Texture B must be loaded explicitly from AssetBundle 2. It is sufficient to have AssetBundle 2 loaded prior to loading Material A out of AssetBundle 1.  
这并不意味着必须在资产包 1 之前加载资产包 2，或者纹理 B 必须从资产包 2 显式加载。在将材质 A 从资产包 1 中加载之前加载资源包 2 就足够了。

However, Unity *will not* automatically load AssetBundle 2 when AssetBundle 1 is loaded. This must be done manually in script code.

For more information on AssetBundle dependencies, refer to the [manual page](https://docs.unity3d.com/Manual/AssetBundles-Dependencies.html).  
但是，Unity *不会*在加载资源包 1 时自动加载资源包 2。这必须在脚本代码中手动完成。

有关资源包依赖关系的更多信息，请参阅 [手册页](https://docs.unity3d.com/Manual/AssetBundles-Dependencies.html)。

### 4.4.3. AssetBundle manifests 资产包清单

When executing the AssetBundle build pipeline using the *BuildPipeline.BuildAssetBundles* API, Unity serializes an Object containing each AssetBundle's dependency information. This data is stored in a separate AssetBundle, which contains a single Object of the [AssetBundleManifest](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.html) type.  
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 依赖项信息的对象。此数据存储在单独的资产包中，该资产包包含资产[AssetBundleManifest](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.html)类型的单个对象。

This Asset will be stored in an AssetBundle with the same name as the parent directory where the AssetBundles are being built. If a project builds its AssetBundles to a folder at *(projectroot)/build/Client/*, then the AssetBundle containing the manifest will be saved as *(projectroot)/build/Client/Client.manifest*.

The AssetBundle containing the manifest can be loaded, cached and unloaded just like any other AssetBundle.  
此资产将存储在与构建资产包的父目录同名的资产包中。如果项目将其资产包构建到*（projectroot）/build/*Client/的文件夹中，则包含清单的资产包将另存为*（projectroot）/build/Client/Client.manifest*。

包含清单的资产包可以像任何其他资产包一样加载、缓存和卸载。

The AssetBundleManifest Object itself provides the [GetAllAssetBundles](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetAllAssetBundles.html) API to list all AssetBundles built concurrently with the manifest and two methods to query the dependencies of a specific AssetBundle:  
AssetBundleManifest 对象本身提供了 [GetAllAssetBundles](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetAllAssetBundles.html) API，用于列出与清单同时构建的所有 AssetBundle，以及两种查询特定 AssetBundle 依赖项的方法：

* [AssetBundleManifest.GetAllDependencies](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetAllDependencies.html) returns all of an AssetBundle's hierarchical dependencies, which includes the dependencies of the AssetBundle's direct children, its children's children, etc.  
  [AssetBundleManifest.GetAllDependencies](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetAllDependencies.html) 返回某个 AssetBundle 的所有分层依赖项，其中包括 AssetBundle 的直接子级、子级的子级等依赖项。
* [AssetBundleManifest.GetDirectDependencies](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetDirectDependencies.html) returns only an AssetBundle's direct children  
  [AssetBundleManifest.GetDirectDependencies](http://docs.unity3d.com/ScriptReference/AssetBundleManifest.GetDirectDependencies.html) 仅返回某个 AssetBundle 的直接子级。

Note that both of these APIs allocate string arrays. Accordingly, they should only be used sparingly, and not during performance-sensitive portions of an application's lifetime.  
请注意，这两个 API 都allocate了字符串数组。因此，应仅谨慎使用它们，而不是在应用程序生存期的性能敏感部分使用。

### 4.4.4. Recommendations

In many cases, it is preferable to load as many needed Objects as possible before players enter performance-critical areas of an application, such as the main game level or world. This is particularly critical on mobile platforms, where access to local storage is slow and the memory churn of loading and unloading Objects at play-time can trigger the garbage collector.  
在许多情况下，最好在玩家进入应用程序的性能关键区域（例如主游戏关卡或世界）之前加载尽可能多的所需对象。这在移动平台上尤其重要，因为对本地存储的访问速度很慢，并且在播放时加载和卸载对象的内存变动可能会触发垃圾回收器。

For projects that must load and unload Objects while the application is interactive, see the**Managing loaded assets** section of the **AssetBundle usage patterns** step for more information on unloading Objects and AssetBundles.  
对于必须在应用程序交互时加载和卸载对象的项目，请参阅资产包**使用模式**步骤的管理**加载的资产**部分，了解有关卸载对象和资产包的更多信息。

# 5.AssetBundle usage patterns

The previous step in this series covered the**fundamentals of AssetBundles,** which included the low-level behavior of various loading APIs. This chapter discusses problems and potential solutions to various aspects of using AssetBundles in practice.  
本系列的上一步介绍了 **AssetBundle 的基础知识，** 其中包括各种加载 API 的低级行为。本章讨论了在实践中使用资产包的各个方面的问题和潜在解决方案。

## 5.1 Managing loaded Assets

It is critical to carefully control the size and number of loaded Objects in memory-sensitive environments. Unity does not automatically unload Objects when they are removed from the active scene. Asset cleanup is triggered at specific times, and it can also be triggered manually.  
在内存敏感型环境中仔细控制加载对象的大小和数量至关重要。Unity 不会在对象从活动场景中移除时自动卸载对象。资产清理在特定时间触发，也可以手动触发。

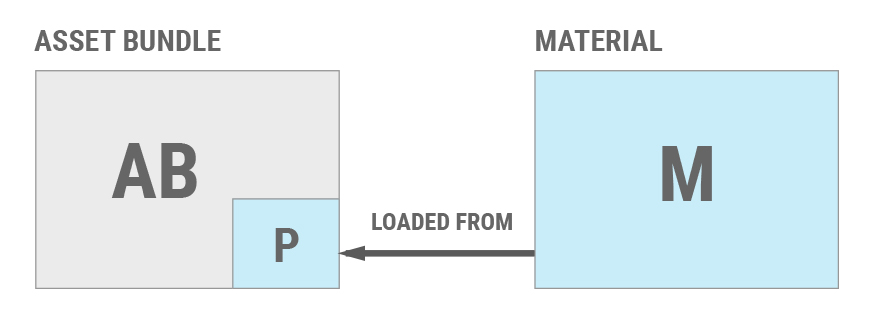
AssetBundles themselves must be carefully managed. An AssetBundle backed by a file on local storage (either in the Unity cache or one loaded via [AssetBundle.LoadFromFile](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromFile.html)) has minimal memory overhead, rarely consuming more than a few dozen kilobytes. However, this overhead can still become problematic if a large number of AssetBundles are present.  
必须仔细管理资产包本身。由本地存储上的文件（在 Unity 缓存中或通过 [AssetBundle.LoadFromFile](http://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromFile.html) 加载的文件）支持的资产包具有最小的内存开销，很少消耗超过几十 KB。但是，如果存在大量资产包，则此开销仍可能成为问题。

As most projects allow users to re-experience content (such as replaying a level), it's important to know when to load or unload an AssetBundle. ***If an AssetBundle is unloaded improperly, it can cause Object duplication in memory.*** Improperly unloading AssetBundles can also result in undesirable behavior in certain circumstances, such as causing textures to go missing. To understand why this can happen, refer to the **Inter-Object references** section of the **Assets, Objects, and serialization** step.  
由于大多数项目允许用户重新体验内容（例如重播关卡），因此了解何时加载或卸载 AssetBundle 非常重要。***如果资产包卸载不正确，可能会导致内存中的对象重复。*** 在某些情况下，不正确地卸载资源包也可能导致不良行为，例如导致纹理丢失。若要了解为什么会发生这种情况，请参阅**资产、对象和序列化**步骤的对象**间引用**部分。

The most important thing to understand when managing assets and AssetBundles is the difference in behavior when calling [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html) with either true or false for the *unloadAllLoadedObjects* parameter.  
在管理资产和资产包时要了解的最重要的事情是调用 [AssetBundle.Unload](http://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html) 时的行为差异，对于 *unloadAllLoadedObjects* 参数，使用 true 或 false 进行 卸载。

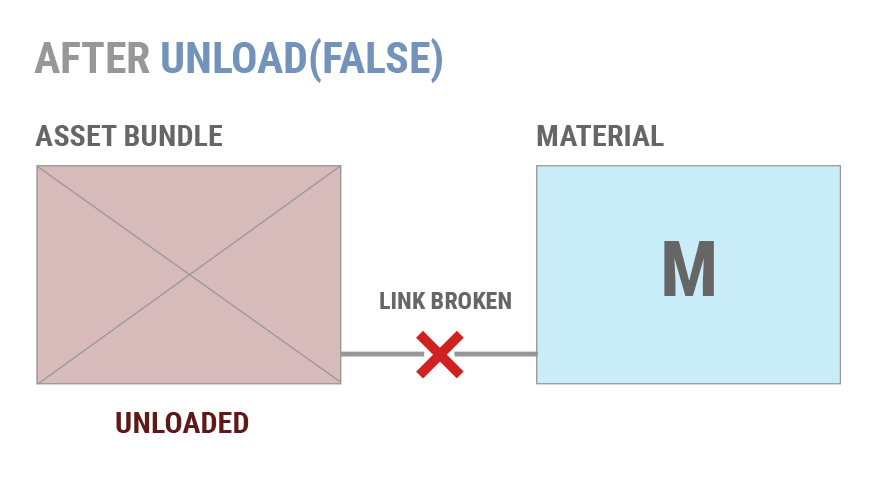
This API will unload the header information of the AssetBundle being called. The *unloadAllLoadedObjects* parameter determines whether to also unload all Objects instantiated from this AssetBundle. If set to *true*, then all Objects originating from the AssetBundle will also be immediately unloaded – even if they are currently being used in the active scene.  
此 API 将卸载正在调用的资产包的标头信息。*unloadAllLoadedObjects* 参数确定是否同时卸载从此资产包实例化的所有对象。如果设置为 *true*，则来自 AssetBundle 的所有对象也将立即卸载 - 即使它们当前正在活动场景中使用。

For example, assume a material *M* was loaded from an AssetBundle *AB*, and assume *M* is currently in the active scene.  
例如，假设材质 M 是从资产包 *AB* 加载的，并假设 *M* 当前处于活动场景中。



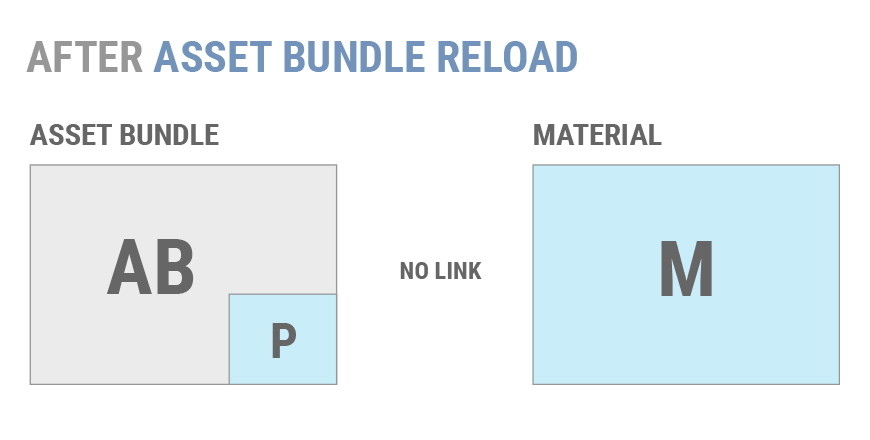
Select image to expand

If *AssetBundle.Unload(true)* is called, then *M* will be removed from the scene, destroyed and unloaded. However, if *AssetBundle.Unload(false)* is called, then *AB*'s header information will be unloaded but *M* will remain in the scene and will still be functional. Calling *AssetBundle.Unload(false)* breaks the link between *M* and *AB*. If *AB* is loaded again later, then fresh copies of the Objects contained in *AB* will be loaded into memory.  
如果调用 *AssetBundle.Unload（true），*则 *M* 将从场景中移除、销毁和卸载。但是，如果调用 *AssetBundle.Unload（false），*则 *AB* 的标头信息将被卸载，但 *M* 将保留在场景中并且仍然起作用。调用 *AssetBundle.Unload（false）* 会断开 *M* 和 *AB* 之间的链接。如果稍后再次加载 AB，则 *AB* 中包含的对象的新副本 将加载到内存中。



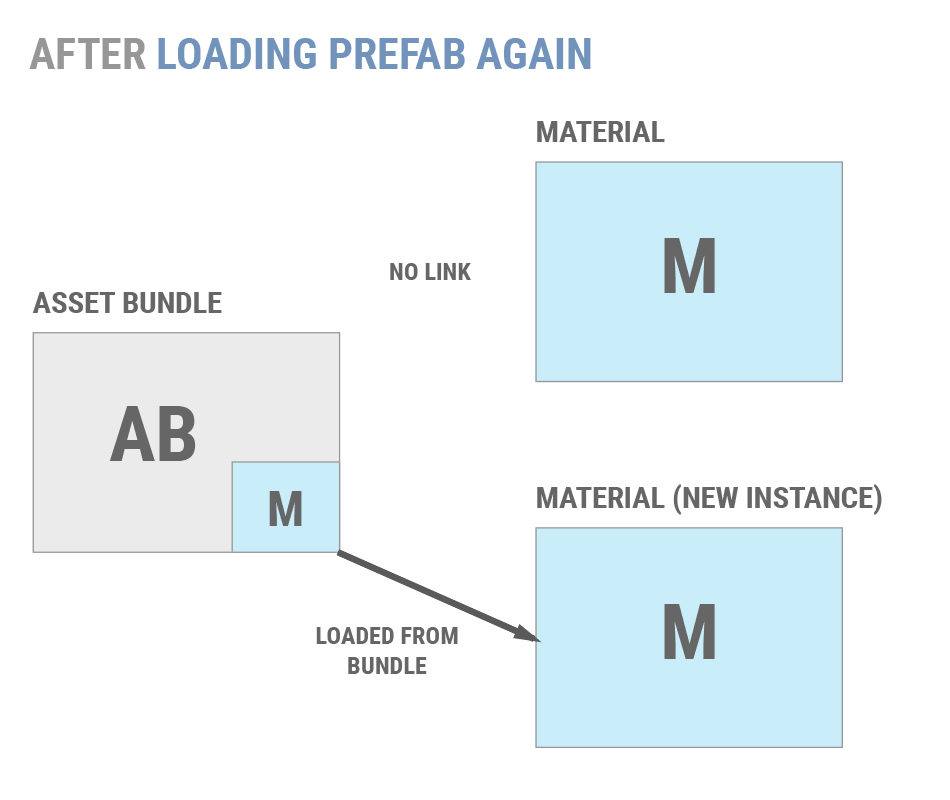
Select image to expand

If *AB* is loaded again later, then a new copy of the AssetBundle's header information will be reloaded. However, *M* was not loaded from this new copy of *AB*. Unity does not establish any link between the new copy of *AB* and *M*.  
如果稍后再次加载 *AB*，则将重新加载资产包标头信息的新副本。但是，*M* 不是从这个新的 AB 副本加载的。Unity 不会在 AB 和 M 的新副本之间建立任何链接 。



Select image to expand

If *AssetBundle.LoadAsset()* were called to reload *M*, Unity would not interpret the old copy of *M* as being an instance of the data in *AB*. Therefore, Unity will load a new copy of *M* and there will be **two** identical copies of *M* in the scene.  
如果调用 *AssetBundle.LoadAsset（）* 来重新加载 M，Unity 不会将 M 的旧副本解释 为 AB 中数据的实例。因此，Unity 将加载 M 的新副本，并且场景中将**有两个**相同的 *M* 副本。



Select image to expand

For most projects, this behavior is undesirable. Most projects should use *AssetBundle.Unload(true)* and adopt a method to ensure that Objects are not duplicated. Two common methods are:  
对于大多数项目，此行为是不可取的。大多数项目应该使用 *AssetBundle.Unload（true）* 并采用一种方法来确保对象不重复。两种常用方法是：

1. Having well-defined points during the application's lifetime at which transient AssetBundles are unloaded, such as between levels or during a loading screen. This is the simpler and most common option.  
   在应用程序的生存期内具有明确定义的点，在该点上卸载瞬态资产包，例如在关卡之间或加载屏幕期间。这是更简单和最常见的选项。
2. Maintaining reference-counts for individual Objects and unload AssetBundles only when all of their constituent Objects are unused. This permits an application to unload and reload individual Objects without duplicating memory.  
   维护单个对象的引用计数，并仅在其所有组成对象未使用时才卸载资源包。这允许应用程序卸载和重新加载单个对象，而无需复制内存。

If an application must use *AssetBundle.Unload(false)*, then individual Objects can only be unloaded in two ways:  
如果应用程序必须使用 *AssetBundle.Unload（false），*则只能通过两种方式卸载单个对象：

1. Eliminate all references to an unwanted Object, both in the scene and in code. After this is done, call [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html).  
   消除场景和代码中对不需要的对象的所有引用。完成此操作后，调用 [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html)。
2. Load a scene non-additively. This will destroy all Objects in the current scene and invoke [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) automatically.  
   以非累加方式加载场景。这将销毁当前场景中的所有对象，并自动调用 [Resources.UnloadUnusedAssets](http://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) 。

If a project has well-defined points where the user can be made to wait for Objects to load and unload, such as in between game modes or levels, these points should be used to unload as many Objects as necessary and to load new Objects.  
如果项目具有明确定义的点，用户可以在其中等待对象加载和卸载，例如在游戏模式或关卡之间，则应使用这些点来卸载任意数量的对象并加载新对象。

The simplest way to do this is to package discrete chunks of a project into scenes, and then build those scenes into AssetBundles, along with all of their dependencies. The application can then enter a "loading" scene, fully unload the AssetBundle containing the old scene, and then load the AssetBundle containing the new scene.  
最简单的方法是将项目的离散区块打包到场景中，然后将这些场景及其所有依赖项构建到 AssetBundle 中。然后，应用程序可以进入“加载”场景，完全卸载包含旧场景的资产包，然后加载包含新场景的资产包。

While this is the simplest flow, some projects require more complex AssetBundle management. As every project is different, there is no universal AssetBundle design pattern.  
虽然这是最简单的流程，但某些项目需要更复杂的资产包管理。由于每个项目都不同，因此没有通用的资产捆绑包设计模式。

When deciding how to group Objects into AssetBundles, it is generally best to start by bundling Objects into AssetBundles if they must be loaded or updated at the same time. For example, consider a role-playing game. Individual maps and cutscenes can be grouped into AssetBundles by scene, but some Objects will be needed in most scenes. AssetBundles could be built to provide portraits, the in-game UI, and different character models and textures. These latter Objects and Assets could then be grouped into a second set of AssetBundles that are loaded at startup and remain loaded for the lifetime of the app.  
在决定如何将对象分组到资产包中时，对于必须同时加载或更新对象，通常将他们捆绑到资产包中。例如，考虑一个角色扮演游戏，单个地图和场景自身元素可以按场景分组到资源包中。但大多数场景中都需要一些公共对象，所以可以构建资产包以提供肖像、游戏内 UI 以及不同的角色模型和纹理。然后，可以将后一组对象和资产分组到第二组资产包中，这些资产包在启动时加载，并在应用程序的生命周期内始终保持加载状态。

Another problem can arise if Unity must reload an Object from its AssetBundle after the AssetBundle has been unloaded. In this case, the reload will fail and the Object will appear in the Unity Editor's hierarchy as a (Missing) Object.  
如果 Unity 必须从被卸载的资源包中重新加载对象，则可能会出现另一个问题。在这种情况下，重新加载将失败，对象将在 Unity 编辑器的层次结构中显示为（缺失）对象。

This primarily occurs when Unity loses and regains control over its graphics context, such as when a mobile app is suspended or the user locks their PC. In this case, Unity must re-upload textures and shaders to the GPU. If the source AssetBundle for these assets is unavailable, the application will render Objects in the scene as magenta.  
这主要发生在 Unity 失去并重新获得对其图形上下文的控制时，例如当移动应用程序挂起或用户锁定其 PC 时。在这种情况下，Unity 必须将纹理和着色器重新上传到 GPU。如果这些资产的源资产包不可用，应用程序会将场景中的对象渲染为洋红色。

## 5.2. Distribution 分配

There are two basic ways to distribute a project's AssetBundles to clients: installing them simultaneously with the project or downloading them after installation.  
有两种基本方法可以将项目的资产包分发给客户端：与项目同时安装或在安装后下载它们。

The decision whether to ship AssetBundles within or after installation is driven by the capabilities and restrictions of the platforms on which the project will run. Mobile projects usually opt for post-install downloads to reduce initial install size and remain below over-the-air download size limits. Console and PC projects generally ship AssetBundles with their initial install.  
是否在安装内或安装后交付资产包的决定取决于运行项目的平台的功能和限制。移动项目通常选择安装后下载以减少初始安装大小，并保持在无线下载大小限制以下。控制台和 PC 项目通常在初始安装时附带资源包。

Proper architecture permits patching new or revised content into your project post-install regardless of how the AssetBundles are delivered initially. For more information on this, see the [Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html) section of the Unity Manual.  
正确的架构允许在安装后将新的或修订的内容修补到项目中，而不管 AssetBundle 最初是如何交付的。有关此内容的更多信息，请参阅 Unity 手册的[Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html)部分。

### 5.2.1. Shipped with project 随项目一起分发

Shipping AssetBundles with the project is the simplest way to distribute them as it does not require additional download-management code. There are two major reasons why a project might include AssetBundles with the install:  
随项目一起交付资产包是分发它们的最简单方法，因为它不需要额外的下载管理代码。项目在安装时可能包含资产包有两个主要原因：

* To reduce project build times and permit simpler iterative development. If these AssetBundles do not need to be updated separately from the application itself, then the AssetBundles can be included with the application by storing the AssetBundles in Streaming Assets. See the **Streaming Assets** section, below.  
  减少项目构建时间并允许更简单的迭代开发。如果这些资产包不需要与应用程序本身分开更新，则可以通过将资产包存储在流资产中来将资产包包含在应用程序中。请参阅下面的 **Streaming Assets**部分。
* To ship an initial revision of updatable content. This is commonly done to save end-users time after their initial install or to serve as the basis for later patching. Streaming Assets is not ideal for this case. However, if writing a custom downloading and caching system is not an option, then an initial revision of updatable content can be loaded into the Unity cache from Streaming Assets (See the **Cache Priming** section, below).  
  提供可更新内容的初始修订版。这样做通常是为了节省最终用户在初始安装后的时间或作为以后修补的基础。Streaming Assets在这种情况式处理资产并不理想。但是，如果无法编写自定义下载和缓存系统，则可以将可更新内容的初始修订版从Streaming Assets加载到 Unity 缓存中（请参阅下面的 **Cache Priming** 部分）。

#### 5.2.1.1. Streaming Assets

The easiest way to include any type of content, including AssetBundles, within a Unity application at install time is to build the content into the */Assets/StreamingAssets/* folder, prior to building the project. Anything contained in the *StreamingAssets* folder at build time will be copied into the final application.  
在安装时在 Unity 应用程序中包含任何类型的内容（包括资源包）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。生成时 *StreamingAssets* 文件夹中包含的任何内容都将复制到最终应用程序中。

The full path to the *StreamingAssets* folder on local storage is accessible via the property [Application.streamingAssetsPath](http://docs.unity3d.com/ScriptReference/Application-streamingAssetsPath.html) at runtime. The AssetBundles can then be loaded with via *AssetBundle.LoadFromFile* on most platforms.  
本地存储上StreamingAssets 文件夹的完整路径可通过运行时的 [Application.streamingAssetsPath](http://docs.unity3d.com/ScriptReference/Application-streamingAssetsPath.html) 属性访问。然后，可以在大多数平台上通过*AssetBundle.LoadFromFile加载AssetBundles*。

***Android Developers:*** On Android, assets in the StreamingAssets folders are stored into the APK and may take more time to load if they are compressed, as files stored in an APK can use different storage algorithms. The algorithm used may vary from one Unity version to another. You can use an archiver such as 7-zip to open the APK to determine if the files are compressed or not. If they are, you can expect AssetBundle.LoadFromFile() to perform more slowly. In this case, you can retrieve the cached version by using [UnityWebRequest.GetAssetBundle](https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.GetAssetBundle.html) as a workaround. By using UnityWebRequest, the AssetBundle will be uncompressed and cached during the first run, allowing for following executions to be faster. Note that this will take more storage space, as the AssetBundle will be copied to the cache. Alternatively, you can export your Gradle project and add an extension to your AssetBundles at build time. You can then edit the build.gradle file and add that extension to the noCompress section. Once done, you should be able to use AssetBundle.LoadFromFile() without having to pay the decompression performance cost.  
***安卓开发者：***在安卓上，StreamingAssets文件夹中的资产会存储到APK中，如果压缩，可能需要更多时间才能加载，因为存储在APK中的文件可以使用不同的存储算法。使用的算法可能因 Unity 版本而异。您可以使用 7-zip 等存档器打开 APK 以确定文件是否已压缩。如果是，可以预见 AssetBundle.LoadFromFile（）的执行速度会更慢。在这种情况下，您可以使用 [UnityWebRequest.GetAssetBundle](https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.GetAssetBundle.html) 作为解决方法来检索缓存的版本。通过使用 UnityWebRequest，资产包将在第一次运行时解压缩和缓存，从而加快后续执行速度。请注意，这将占用更多存储空间，因为资产包将被复制到缓存中。或者，您可以在构建时向资产包添加扩展名。然后，您可以导出 Gradle 项目并编辑build.gradle文件，将该扩展名添加到noCompress部分。完成后，您应该能够使用 AssetBundle.LoadFromFile（），而无需支付解压缩性能成本。

*Note:* Streaming Assets is not a writable location on some platforms. If a project's AssetBundles need to be updated after installation, either use *WWW.LoadFromCacheOrDownload* or write a custom downloader.  
*注意：* 在某些平台上，Streaming Assets不是可写位置。如果项目的资源包在安装后需要更新，请使用 *WWW.LoadFromCacheOrDownload* 或编写自定义下载器。

### 5.2.2. Downloaded post-install 安装后下载

The favored method of delivering AssetBundles to mobile devices is to download them after app installation. This also allows the content to be updated after installation without forcing users to re-download the entire application. On many platforms, application binaries must undergo an expensive and lengthy recertification process. Therefore, developing a good system for post-install downloads is vital.  
将资产包交付到移动设备的首选方法是在安装应用程序后下载它们。这也允许在安装后更新内容，而无需强制用户重新下载整个应用程序。在许多平台上，应用程序二进制文件必须经历昂贵且冗长的重新认证过程。因此，为安装后下载开发一个好的系统至关重要。

The simplest way to deliver AssetBundles is to place them on a web server and deliver them via *UnityWebRequest*. Unity will automatically cache downloaded AssetBundles on local storage. If the downloaded AssetBundle is LZMA compressed, the AssetBundle will be stored in the cache either as uncompressed or re-compressed as LZ4 (dependent on the [Caching.compressionEnabled](https://docs.unity3d.com/ScriptReference/Caching-compressionEnabled.html) setting), for faster loading in the future. If the downloaded bundle is LZ4 compressed, the AssetBundle will be stored compressed. If the cache fills up, Unity will remove the least recently used AssetBundle from the cache. See the **Built-in caching** section for more details.  
交付资产包的最简单方法是将它们放在 Web 服务器上并通过 *UnityWebRequest* 交付它们。Unity 会自动将下载的资源包缓存在本地存储上。如果下载的资产包是 LZMA 压缩的，则资产包将作为未压缩或重新压缩的 LZ4 存储在缓存中（取决于 [Caching.compressionEnabled](https://docs.unity3d.com/ScriptReference/Caching-compressionEnabled.html) 设置），以便将来更快地加载。如果下载的捆绑包是 LZ4 压缩的，则资源包将以压缩方式存储。如果缓存已满，Unity 将从缓存中删除最近最少使用的资源包。有关更多详细信息，请参阅下面的**内置缓存**部分。

It is generally recommended to start by using *UnityWebRequest* when possible, or *WWW.LoadFromCacheOrDownload* only if using Unity 5.2 or older. Only invest in a custom download system if the built-in APIs' memory consumption, caching behavior or performance are unacceptable for a specific project, or if a project must run platform-specific code to achieve its requirements.  
通常建议尽可能使用 *UnityWebRequest。仅* 当使用 Unity 5.2 或更早版本时，才使用*WWW.LoadFromCacheOrDownload*。仅当内置 API 的内存消耗、缓存行为或性能对于特定项目不可接受，或者项目必须运行特定于平台的代码才能实现其要求时，才考虑自定义下载系统。

Examples of situations which may prevent the use of *UnityWebRequest* or *WWW.LoadFromCacheOrDownload*:  
可能阻止使用 *UnityWebRequest* 或*WWW.LoadFromCacheOrDownload*的情况：

* When fine-grained control over the AssetBundle cache is required  
  当需要对资产包缓存进行精细控制时
* When a project needs to implement a custom compression strategy  
  当项目需要实现自定义压缩策略时
* When a project wishes to use platform-specific APIs to satisfy certain requirements, such as the need to stream data while inactive.  
  当项目希望使用特定于平台的 API 来满足某些要求时，例如需要在非活动时流式传输数据。
* *Example:* Using iOS' Background Tasks API to download data while in the background.  
  *示例：* 使用 iOS 的后台任务 API 在后台下载数据。
* When AssetBundles must be delivered over SSL on platforms where Unity does not have proper SSL support (such as PC).  
  当资源包必须在 Unity 没有适当 SSL 支持的平台上通过 SSL 交付时（例如 PC）。

### 5.2.3. Built-in caching 内置缓存

Unity has a built-in AssetBundle caching system that can be used to cache AssetBundles downloaded via the *UnityWebRequest* API, which has an overload accepting an AssetBundle version number as an argument. This number is *not* stored inside the AssetBundle, and is *not* generated by the AssetBundle system.  
Unity 有一个内置的 AssetBundle 缓存系统，可用于缓存通过 *UnityWebRequest* API 下载的资产包，该系统具有接受 AssetBundle 版本号作为参数的重载。此编号*不*存储在资产包中，也*不*由资产包系统生成。

The caching system keeps track of the last version number passed to *UnityWebRequest*. When this API is called with a version number, the caching system checks to see if there is a cached AssetBundle by comparing version numbers. If these numbers match, the system will load the cached AssetBundle. If the numbers do not match, or there is no cached AssetBundle, then Unity will download a new copy. This new copy will be associated with the new version number.  
缓存系统会跟踪传递给 *UnityWebRequest* 的最后一个版本号。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资产包。如果这些数字匹配，系统将加载缓存的资产包。如果数字不匹配，或者没有缓存的资源包，则 Unity 将下载新副本。此新副本将与新版本号相关联。

***AssetBundles in the caching system are identified only by their file names***, and not by the full URL from which they are downloaded. This means that an AssetBundle with the same file name can be stored in multiple different locations, such as a Content Delivery Network. As long as the file names are identical, the caching system will recognize them as the same AssetBundle.  
***缓存系统中的资产包仅通过其文件名进行标识***，而不由下载它们的完整 URL 标识。这意味着具有相同文件名的资产包可以存储在多个不同的位置，例如内容交付网络。只要文件名相同，缓存系统就会将它们识别为相同的资产包。

It is up to each individual application to determine an appropriate strategy for assigning version numbers to AssetBundles, and to pass these numbers to *UnityWebRequest*. The numbers may come from unique identifiers of sorts, such as a CRC value. Note that while AssetBundleManifest.GetAssetBundleHash() may also be used for this purpose, we don’t recommend this function for versioning, as it provides just an estimation, and not a true hash calculation).  
由每个单独的应用程序确定将版本号分配给资产包的适当策略，并将这些编号传递给 *UnityWebRequest*。这些数字可能来自各种唯一标识符，例如 CRC 值。请注意，虽然 AssetBundleManifest.GetAssetBundleHash（） 也可以用于此目的，但我们不建议将此函数用于版本控制，因为它仅提供估计值，而不是真正的哈希计算）。

See the [Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html) section of the Unity Manual for more details.  
有关更多详细信息，请参阅 Unity 手册的[Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html)部分。

In Unity 2017.1 onward, the [Caching](https://docs.unity3d.com/ScriptReference/Caching.html) API has been extended to provide more granular control, by allow developers to select an active cache from multiple caches. Prior versions of Unity may only modify [Caching.expirationDelay](https://docs.unity3d.com/560/Documentation/ScriptReference/Caching-expirationDelay.html) and [Caching.maximumAvailableDiskSpace](https://docs.unity3d.com/560/Documentation/ScriptReference/Caching-maximumAvailableDiskSpace.html) to remove cached items (these properties remain in Unity 2017.1 in the [Cache class](https://docs.unity3d.com/ScriptReference/Cache.html)).  
在 Unity 2017.1 及更高版本中，[Caching](https://docs.unity3d.com/ScriptReference/Caching.html) API 已得到扩展，允许开发人员从多个缓存中选择活动缓存，从而提供更精细的控制。以前版本的 Unity 只能修改 [Caching.expirationDelay](https://docs.unity3d.com/560/Documentation/ScriptReference/Caching-expirationDelay.html) 和 [Caching.maximumAvailableDiskSpace](https://docs.unity3d.com/560/Documentation/ScriptReference/Caching-maximumAvailableDiskSpace.html) 以删除缓存的项目（这些属性保留在 Unity 2017.1 的[Cache class](https://docs.unity3d.com/ScriptReference/Cache.html)中）。

[expirationDelay](http://docs.unity3d.com/ScriptReference/Caching-expirationDelay.html) is the minimum number of seconds that must elapse before an AssetBundle is automatically deleted. If an AssetBundle is not accessed during this time, it will be deleted automatically.  
[expirationDelay](http://docs.unity3d.com/ScriptReference/Caching-expirationDelay.html) 是自动删除资产包之前必须经过的最小秒数。如果在此期间未访问资产包，它将自动删除。

[maximumAvailableDiskSpace](http://docs.unity3d.com/ScriptReference/Caching-maximumAvailableDiskSpace.html) specifies the amount of space on local storage, in bytes, that the cache may use before it begins deleting AssetBundles that have been used less recently than the *expirationDelay*. When the limit is reached, Unity will delete the AssetBundle in the cache which was least recently opened (or marked as used via *Caching.MarkAsUsed*). Unity will delete cached AssetBundles until there is sufficient space to complete the new download.  
[maximumAvailableDiskSpace](http://docs.unity3d.com/ScriptReference/Caching-maximumAvailableDiskSpace.html) 指定缓存在开始删除最近使用的时间少于过期延迟的资产包之前可能使用的本地存储空间量（以字节为单位）。达到限制后，Unity 将删除缓存中最近未打开（或通过 *Caching.MarkAsUsed* 标记为已使用）的资源包。Unity 将删除缓存的资源包，直到有足够的空间来完成新下载。

#### 5.2.3.1. Cache Priming 缓存启动

Because AssetBundles are identified by their file names, it is possible to "prime" the cache with AssetBundles shipped with the application. To do this, store the initial or base version of each AssetBundle in */Assets/StreamingAssets/*. The process is identical to the one detailed in the**Shipped with project** section.  
由于资产包由其文件名标识，因此可以使应用程序附带的资产包也“启用”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。该过程与“**随项目一起提供”**部分中详细介绍的过程相同。

The cache can be populated by loading AssetBundles from *Application.streamingAssetsPath* the first time the application is run. From then on, the application can call *UnityWebRequest* normally (UnityWebRequest can also be used to initially load AssetBundles from the StreamingAssets path as well).  
可以通过在应用程序首次运行时从 *Application.streamingAssetsPath* 加载 AssetBundles 来填充缓存。从那时起，应用程序可以正常调用 UnityWebRequest （UnityWebRequest也可用于最初从StreamingAssets路径加载AssetBundle）。

### 5.2.4. Custom downloaders 自定义下载器

Writing a custom downloader gives an application full control over how AssetBundles are downloaded, decompressed and stored. As the engineering work involved is non-trivial, we recommend this approach only for larger teams. There are four major considerations when writing a custom downloader:  
编写自定义下载器使应用程序能够完全控制资产包的下载、解压缩和存储方式。由于所涉及的工程工作并非易事，因此我们建议仅对较大的团队使用此方法。编写自定义下载程序时有四个主要注意事项：

* Download mechanism  
  下载机制
* Storage location  
  存储位置
* Compression type  
  压缩类型
* Patching  
  修补

For information on patching AssetBundles, see the [Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html) section of the Unity Manual.  
有关修补资源包的信息，请参阅 Unity 手册的 [Patching with AssetBundles](https://docs.unity3d.com/Manual/AssetBundles-Patching.html)部分。

#### 5.2.4.1. Downloading

For most applications, HTTP is the simplest method to download AssetBundles. However, implementing an HTTP-based downloader is not the simplest task. Custom downloaders must avoid excessive memory allocations, excessive thread usage and excessive thread wakeups. Unity's WWW class is unsuitable for reasons exhaustively described in the **WWW.LoadFromCacheOrDownload** section of the **AssetBundle fundamentals** step.  
对于大多数应用程序，HTTP 是下载资产包的最简单方法。但是，实现基于 HTTP 的下载程序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。Unity 的 WWW 类不适合，原因在**AssetBundle fundamentals的** **WWW.LoadFromCacheOrDownload**部分。

When writing a custom downloader, there are three options:

* C#'s HttpWebRequest and WebClient classes
* Custom native plugins
* Asset store packages

**5.2.4.1.1. C# classes**

If an application does not require HTTPS/SSL support, C#'s [WebClient](https://msdn.microsoft.com/en-us/library/system.net.webclient%28v=vs.110%29.aspx) class provides the simplest possible mechanism for downloading AssetBundles. It is capable of asynchronously downloading any file directly to local storage without excessive managed memory allocation.  
如果应用程序不需要HTTPS/SSL支持，C#的 [WebClient](https://msdn.microsoft.com/en-us/library/system.net.webclient%28v=vs.110%29.aspx) 类提供了下载AssetBundles的最简单机制。它能够将任何文件直接异步下载到本地存储，而无需过多的托管内存分配。

To download an AssetBundle with WebClient, allocate an instance of the class and pass it the URL of the AssetBundle to download and a destination path. If more control is required over the request's parameters, it is possible to write a downloader using C#'s [HttpWebRequest](https://msdn.microsoft.com/en-us/library/system.net.httpwebrequest%28v=vs.90%29.aspx) class:  
要使用 WebClient 下载资产包，请分配类的实例，并向其传递要下载的资产包的 URL 和目标路径。如果需要对请求的参数进行更多控制，则可以使用 C# 的 [HttpWebRequest](https://msdn.microsoft.com/en-us/library/system.net.httpwebrequest%28v=vs.90%29.aspx) 类编写下载器：

1. Get a byte stream from *HttpWebResponse.GetResponseStream*.
2. Allocate a fixed-size byte buffer on the stack.
3. Read from the response stream into the buffer.
4. Write the buffer to disk using C#'s File.IO APIs, or any other streaming IO system.

**5.2.4.1.2. Asset Store Packages**

Several asset store packages offer native-code implementations to download files via HTTP, HTTPS and other protocols. Before writing a custom native-code plugin for Unity, it is recommended to evaluate available Asset Store packages.  
一些资源商店包提供本机代码实现，可通过 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，建议评估可用的资源商店包。

**5.2.4.1.3. Custom Native Plugins**

Writing a custom native plugin is the most time-intensive, but most flexible method for downloading data in Unity. Due to the high programming time requirements and high technical risk, this method is only recommended if no other method is capable of satisfying an application's requirements. For example, a custom native plugin may be necessary if an application must use SSL communication on platforms without C# SSL support in Unity.  
编写自定义原生插件是在 Unity 中下载数据最耗时但最灵活的方法。由于编程时间要求高，技术风险高，仅当没有其他方法能够满足应用程序的要求时，才建议使用此方法。例如，如果应用程序必须在 Unity 中不支持 C# SSL 的平台上使用 SSL 通信，则可能需要自定义本机插件。

A custom native plugin will generally wrap a target platform's native downloading APIs. Examples include [NSURLConnection](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/) on iOS and [java.net.HttpURLConnection](http://download.java.net/jdk7/archive/b123/docs/api/java/net/HttpURLConnection.html) on Android. Consult each platform's native documentation for further details on using these APIs.  
自定义原生插件通常会包装目标平台的原生下载 API。示例包括iOS上的 [NSURLConnection](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/) 和Android上的 [java.net.HttpURLConnection](http://download.java.net/jdk7/archive/b123/docs/api/java/net/HttpURLConnection.html) 。有关使用这些 API 的更多详细信息，请参阅每个平台的本机文档。

#### 5.2.4.2. Storage

On all platforms, *Application.persistentDataPath* points to a writable location that should be used for storing data that should persist between runs of an application. When writing a custom downloader, it is strongly recommended to use a subdirectory of *Application.persistentDataPath* to store downloaded data.  
在所有平台上，*Application.persistentDataPath* 指向一个可写位置，该位置应用于存储应在应用程序运行之间保留的数据。编写自定义下载器时，强烈建议使用 *Application.persistentDataPath* 的子目录来存储下载的数据。

*Application.streamingAssetPath* is not writable and is a poor choice for an AssetBundle cache. Example locations for *streamingAssetsPath* include:  
*Application.streamingAssetPath* 不可写，对于 AssetBundle 缓存来说是一个糟糕的选择。*Application.streamingAssetPath* 的位置举例如下：

* **OSX**: Within .app package; not writable.
* **Windows**: Within install directory (e.g. *Program Files*); usually not writable
* **iOS**: Within .ipa package; not writable
* **Android**: Within .apk file; not writable

## 5.3. Asset Assignment Strategies 资产分配策略

Deciding how to divide a project's assets into AssetBundles is not simple. It is tempting to adopt a simplistic strategy, such as placing all Objects in their own AssetBundle or using only a single AssetBundle, but these solutions have significant drawbacks:  
决定如何将项目的资产划分为资产包并不简单。采用简单的策略是很诱人的，例如将所有对象放在自己的资产包中或仅使用单个资产包，但这些解决方案具有明显的缺点：

* Having too few AssetBundles...
* Increases runtime memory usage
* Increases loading times
* Requires larger downloads
* Having too many AssetBundles...
* Increases build times
* Can complicate development
* Increases total download time

The key decision is how to group Objects into AssetBundles. The primary strategies are:  
关键决策是如何将对象分组到资产包中。主要策略是：

* Logical entities
* Object Types
* Concurrent content

More information about these grouping strategies can be found in the [Manual](https://docs.unity3d.com/Manual/AssetBundles-Preparing.html).

## 5.4. Common pitfalls 常见陷阱

This section describes several problems that commonly appear in projects using AssetBundles.  
本节介绍使用资源包的项目中经常出现的几个问题。

### 5.4.1. Asset duplication

Unity 5's AssetBundle system will discover all dependencies of an Object when the Object is built into an AssetBundle. This dependency information is used to determine the set of Objects that will be included in an AssetBundle.  
Unity 5 的资源包系统将在对象构建到资源包中时发现对象的所有依赖项。此依赖关系信息用于确定将包含在资产包中的对象集。

Objects that are explicitly assigned to an AssetBundle will only be built into that AssetBundle. An Object is "explicitly assigned" when that Object's AssetImporter has its *assetBundleName* property set to a non-empty string. This can be done in the Unity Editor by selecting an AssetBundle in the Object's Inspector, or from Editor scripts.  
显式分配给资产包的对象将仅内置到该资产包中。当对象的 *AssetImporter 将其 assetBundleName* 属性设置为非空字符串时，将“显式分配”对象。这可以在 Unity 编辑器中通过在对象的检查器中选择资源包或从编辑器脚本中完成。

Objects can also be assigned to an AssetBundle by defining them as part of an [AssetBundle building map](https://docs.unity3d.com/ScriptReference/AssetBundleBuild.html), which is to be used in conjunction with the overloaded [BuildPipeline.BuildAssetBundles()](https://docs.unity3d.com/ScriptReference/BuildPipeline.BuildAssetBundles.html) function that takes in an array of AssetBundleBuild.  
还可以通过将对象定义为 [AssetBundle building map](https://docs.unity3d.com/ScriptReference/AssetBundleBuild.html)的一部分来将对象分配给 [AssetBundle](https://docs.unity3d.com/ScriptReference/AssetBundleBuild.html)，该映射将与Overloaded [BuildPipeline.BuildAssetBundles()](https://docs.unity3d.com/ScriptReference/BuildPipeline.BuildAssetBundles.html)函数结合使用，该函数接受 AssetBundleBuild 数组。

***Any Object that is not explicitly assigned in an AssetBundle will be included in all AssetBundles that contain 1 or more Objects that reference the untagged Object.  
任何未在资产包中显式分配的对象都将包含在包含 1 个或多个引用未标记对象的对象的所有资产包中。***

For example, if two different Objects are assigned to two different AssetBundles, but both have references to a common dependency Object, then that dependency Object will be copied into *both* AssetBundles. The duplicated dependency will also be instanced, meaning that the two copies of the dependency Object will be considered different Objects with different identifiers. This will increase the total size of the application's AssetBundles. This will also cause two different copies of the Object to be loaded into memory if the application loads both of its parents.  
例如，如果将两个不同的对象分配给两个不同的资源包，但都引用了公共依赖对象，则该依赖对象将被复制到 *两个* 资源包中。复制的依赖项也将实例化，这意味着依赖项对象的两个副本将被视为具有不同标识符的不同对象。这将增加应用程序资产包的总大小。如果应用程序同时加载其父副本，这也将导致将对象的两个不同副本加载到内存中。

There are several ways to address this problem:  
有几种方法可以解决此问题：

1. Ensure that Objects built into different AssetBundles do not share dependencies. Any Objects which do share dependencies can be placed into the same AssetBundle without duplicating their dependencies.  
   确保内置到不同资产包中的对象不共享依赖关系。任何共享依赖关系的对象都可以放入同一个资源包中，而无需复制其依赖关系。

* This method usually is not viable for projects with many shared dependencies. It produces monolithic AssetBundles that must be rebuilt and re-downloaded too frequently to be convenient or efficient.  
  此方法通常不适用于具有许多共享依赖项的项目。它生成整体式资产捆绑包，必须过于频繁地重新构建和重新下载。

1. Segment AssetBundles so that no two AssetBundles that share a dependency will be loaded at the same time.  
   对资产包进行分段，以便不会同时加载共享依赖项的两个资产包。

* This method may work for certain types of projects, such as level-based games. However, it still unnecessarily increases the size of the project's AssetBundles, and increases both build times and loading times.  
  此方法可能适用于某些类型的项目，例如基于关卡的游戏。但是，它仍然不必要地增加了项目资产包的大小，并增加了构建时间和加载时间。

1. Ensure that all dependency assets are built into their own AssetBundles. This entirely eliminates the risk of duplicated assets, but also introduces complexity. The application must track dependencies between AssetBundles, and ensure that the right AssetBundles are loaded before calling any *AssetBundle.LoadAsset* APIs.  
   确保所有依赖项资产都内置到其自己的资产包中。这完全消除了重复资产的风险，但也带来了复杂性。应用程序必须跟踪 AssetBundle 之间的依赖关系，并确保在调用任何 *AssetBundle.LoadAsset* API 之前加载正确的 AssetBundle。

Object dependencies are tracked via the *AssetDatabase* API, located in the *UnityEditor* namespace. As the namespace implies, this API is only available in the Unity Editor and not at runtime. *AssetDatabase.GetDependencies* can be used to locate all of the immediate dependencies of a specific Object or Asset. Note that these dependencies may have their own dependencies. Additionally, the *AssetImporter* API can be used to query the AssetBundle to which any specific Object is assigned.  
对象依赖关系通过位于 *UnityEditor* 命名空间中的*AssetDatabase* API 进行跟踪。正如命名空间所暗示的那样，此 API 仅在 Unity 编辑器中可用，在运行时不可用。*AssetDatabase.GetDependencies* 可用于查找特定对象或资产的所有直接依赖项。请注意，这些依赖项可能有自己的依赖项。此外，*AssetImporter* API 可用于查询分配给任何特定对象的资产包。

By combining the *AssetDatabase* and *AssetImporter* APIs, it is possible to write an Editor script that ensures that all of an AssetBundle's direct or indirect dependencies are assigned to AssetBundles, or that no two AssetBundles share dependencies that have not been assigned to an AssetBundle. Due to the memory cost of duplicating assets, it is recommended that all projects have such a script.  
通过组合资产 *AssetDatabase* 和 *AssetImporter* API，可以编写一个编辑器脚本，以确保将资产包的所有直接或间接依赖项分配给资产包，或者没有两个资产包共享尚未分配给资产包的依赖项。由于复制资源的内存成本，建议所有项目都有这样的脚本。

### 5.4.2. Sprite atlas duplication

Any automatically-generated sprite atlas will be assigned to the AssetBundle containing the Sprite Objects from which the sprite atlas was generated. If the sprite Objects are assigned to multiple AssetBundles, then the sprite atlas will not be assigned to an AssetBundle and will be duplicated. If the Sprite Objects are not assigned to an AssetBundle, then the sprite atlas will also not be assigned to an AssetBundle.  
任何自动生成的精灵图集都将分配给包含从中生成精灵图集的精灵对象的资源包。如果精灵对象被分配给多个资源包，则精灵图集将不会分配给资产包，而是会被复制。如果精灵对象未分配给资源包，则精灵图集也不会分配给资源包。

To ensure that sprite atlases are not duplicated, check that all sprites tagged into the same sprite atlas are assigned to the same AssetBundle.  
为确保精灵图集不重复，请检查标记到同一精灵图集中的所有精灵是否都分配给同一资源包。

Note that in Unity 5.2.2p3 and older, automatically-generated sprite atlases will never be assigned to an AssetBundle. Because of this, they will be included in any AssetBundles containing their constituent sprites and also any AssetBundles referencing their constituent sprites. Because of this problem, it is strongly recommended that all Unity 5 projects using Unity's sprite packer upgrade to Unity 5.2.2p4, 5.3 or any newer version of Unity.  
请注意，在 Unity 5.2.2p3 及更早版本中，自动生成的精灵图集永远不会分配给资源包。因此，它们将被包含在包含其组成精灵的任何资产包中，以及引用其构成精灵的任何资产包中。由于此问题，强烈建议所有使用 Unity 精灵打包程序的 Unity 5 项目升级到 Unity 5.2.2p4、5.3 或任何更新版本的 Unity。

### 5.4.3. Android textures 安卓纹理

Due to heavy device fragmentation in the Android ecosystem, it is often necessary to compress textures into several different formats. While all Android devices support ETC1, ETC1 does not support textures with alpha channels. Should an application not require OpenGL ES 2 support, the cleanest way to solve the problem is to use ETC2, which is supported by all Android OpenGL ES 3 devices.  
由于 Android 生态系统中的设备碎片化严重，通常需要将纹理压缩成几种不同的格式。虽然所有 Android 设备都支持 ETC1，但 ETC1 不支持带有 Alpha 通道的纹理。如果应用程序不需要OpenGL ES 2支持，解决问题的最干净方法是使用ETC2，所有Android OpenGL ES 3设备都支持ETC2。

Most applications need to ship on older devices where ETC2 support is unavailable. One way to solve this problem is with Unity 5's AssetBundle Variants (refer to Unity's Android optimization guide for details on other options).  
大多数应用程序需要在不支持 ETC2 的旧设备上交付。解决此问题的一种方法是使用 Unity 5 的资源包变体（有关其他选项的详细信息，请参阅 Unity 的 Android 优化指南）。

To use AssetBundle Variants, all textures that cannot be cleanly compressed using ETC1 must be isolated into texture-only AssetBundles. Next, create sufficient variants of these AssetBundles to support the non-ETC2-capable slices of the Android ecosystem, using vendor-specific texture compression formats such as DXT5, PVRTC and ATITC. For each AssetBundle Variant, change the included textures' TextureImporter settings to the compression format appropriate to the Variant.  
要使用资产包变体，必须将无法使用 ETC1 完全压缩的所有纹理隔离到仅纹理资源包中。接下来，使用特定于供应商的纹理压缩格式（如 DXT5、PVRTC 和 ATITC），创建这些资产捆绑包的足够变体，以支持 Android 生态系统中不支持 ETC2 的切片。对于每个资源包变体，将包含纹理的纹理导入器设置更改为适合该变体的压缩格式。

At runtime, support for the different texture compression formats can be detected using the [SystemInfo.SupportsTextureFormat](http://docs.unity3d.com/ScriptReference/SystemInfo.SupportsTextureFormat.html) API. This information should be used to select and load the AssetBundle Variant containing textures compressed in a supported format.  
在运行时，可以使用 [SystemInfo.SupportsTextureFormat](http://docs.unity3d.com/ScriptReference/SystemInfo.SupportsTextureFormat.html) API 检测对不同纹理压缩格式的支持。此信息应用于选择和加载包含以受支持格式压缩的纹理的资产包变体。

More information on Android texture compression formats can be found [here](http://developer.android.com/guide/topics/graphics/opengl.html#textures).

### 5.4.4. iOS file handle overuse

***Current versions of Unity are not affected by this issue.***

In versions prior to Unity 5.3.2p2, Unity would hold an open file handle to an AssetBundle the entire time that the AssetBundle is loaded. This is not a problem on most platforms. However, iOS limits the number of file handles a process may simultaneously have open to 255. If loading an AssetBundle causes this limit to be exceeded, the loading call will fail with a "Too Many Open File Handles" error.

This was a common problem for projects trying to divide their content across many hundreds or thousands of AssetBundles.

For projects unable to upgrade to a patched version of Unity, temporary solutions are:

* Reducing the number of AssetBundles in use by merging related AssetBundles
* Using *AssetBundle.Unload(false)* to close an AssetBundle's file handle, and managing the loaded Objects' lifecycles manually

## 5.5. AssetBundle Variants 资产捆绑包变体

A key feature of the AssetBundle system is the introduction of AssetBundle Variants. The purpose of Variants is to allow an application to adjust its content to better suit its runtime environment. Variants permit different UnityEngine.Objects in different AssetBundle files to appear as being the "same" Object when loading Objects and resolving Instance ID references. Conceptually, it permits two UnityEngine.Objects to appear to share the same File GUID & Local ID, and identifies the actual UnityEngine.Object to load by a string Variant ID.  
资产捆绑包系统的一个关键功能是引入了资产捆绑包变体。变体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。变体允许不同资源包文件中的不同 UnityEngine.Objects 在加载对象和解析实例 ID 引用时显示为“相同”对象。从概念上讲，它允许两个 UnityEngine.Objects 共享相同的文件 GUID 和本地 ID，并通过字符串 Variant ID 标识要加载的实际 UnityEngine.Object。

There are two primary use cases for this system:  
此系统有两个主要用例：

1. Variants simplify the loading of AssetBundles appropriate for a given platform.  
   变体简化了适用于给定平台的资产包的加载。
2. Example: A build system might create an AssetBundle containing high-resolution textures and complex shaders suitable for a standalone DirectX11 Windows build, and a second AssetBundle with lower-fidelity content intended for Android. At runtime, the project's resource loading code can then load the appropriate AssetBundle Variant for its platform, and the Object names passed into the AssetBundle.Load API do not need to change.  
   示例：构建系统可能会创建一个资产包，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，以及第二个资产包，其中包含适用于 Android 的低保真内容。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 变体，并且传递到 AssetBundle.Load API 的对象名称不需要更改。
3. Variants allow an application to load different content on the same platform, but with different hardware.  
   变体允许应用程序在同一平台但使用不同的硬件的情况下加载不同的内容。
4. This is key for supporting a wide range of mobile devices. An iPhone 4 is incapable of displaying the same fidelity of content as the latest iPhone in any real-world application.  
   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。
5. On Android, AssetBundle Variants can be used to tackle the immense fragmentation of screen aspect ratios and DPIs between devices.  
   在Android上，AssetBundle变体可用于解决设备之间屏幕纵横比和DPI的巨大碎片化问题。

### 5.5.1. Limitations

A key limitation of the AssetBundle Variant system is that it requires Variants to be built from distinct Assets. This limitation applies even if the only variations between those Assets is their import settings. If the only distinction between a texture built into Variant A and Variant B is the specific texture compression algorithm selected in the Unity texture importer, Variant A and Variant B must still be entirely different Assets. This means that Variant A and Variant B must be separate files on disk.  
资产捆绑变体系统的一个关键限制是，它要求从不同的资产构建变体。即使这些资源之间的唯一变化是其导入设置，此限制也适用。如果变体 A 和变体 B 中内置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。这意味着变体 A 和变体 B 必须是磁盘上的单独文件。

This limitation complicates the management of large projects as multiple copies of a specific Asset must be kept in source control. All copies of an Asset must be updated when developers wish to change the content of the Asset. There are no built-in workarounds for this problem.  
此限制使大型项目的管理复杂化，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产的内容时，必须更新资产的所有副本。此问题没有内置的解决方法。

Most teams implement their own form of AssetBundle Variants. This is done by building AssetBundles with well-defined suffixes appended to their filenames, in order to identify the specific variant a given AssetBundle represents. Custom code programmatically alters the importer settings of the included Assets when building these AssetBundles. Some developers have extended their custom systems to also be able to alter parameters on components attached to prefabs.  
大多数团队都实现了自己的资产捆绑变体形式。这是通过构建资产包来完成的，该资产包在其文件名后附加了明确定义的后缀，以便识别给定资产包所代表的特定变体。自定义代码在构建这些资产包时以编程方式更改所包含资产的导入程序设置。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。

## 5.6. Compressed or uncompressed? 压缩还是未压缩？

Whether to compress AssetBundles requires several important considerations, which include:  
是否压缩资源包需要几个重要的注意事项，其中包括：

* **Loading time**: Uncompressed AssetBundles are much faster to load than compressed AssetBundles when loading from local storage or a local cache.  
  **加载时间**：从本地存储或本地缓存加载时，未压缩的资产包加载速度比压缩的资产包快得多。
* **Build time**: LZMA and LZ4 are very slow when compressing files, and the Unity Editor processes AssetBundles serially. Projects with a large number of AssetBundles will spend a lot of time compressing them.  
  **构建时间**：LZMA 和 LZ4 在压缩文件时非常慢，Unity 编辑器会串行处理资源包。具有大量资源包的项目将花费大量时间来压缩它们。
* **Application size**: If the AssetBundles are shipped in the application, compressing them will reduce the application's total size. Alternatively, the AssetBundles can be downloaded post-install.  
  **应用程序大小**：如果资产包在应用程序中提供，则压缩它们将减少应用程序的总大小。或者，可以在安装后下载资源包。
* **Memory usage**: Prior to Unity 5.3, all of Unity's decompression mechanisms required the entire compressed AssetBundle to be loaded into memory prior to decompression. If memory usage is important, use either uncompressed or LZ4 compressed AssetBundles.  
  **内存使用**：在Unity 5.3之前，Unity 的所有解压缩机制都需要在解压缩之前将整个压缩的资源包加载到内存中。如果内存使用很重要，请使用未压缩或 LZ4 压缩的资产包。
* **Download time**: Compression may only be necessary if the AssetBundles are large, or if users are in a bandwidth-constrained environment, such as downloading on low-speed or metered connections. If only a few tens of megabytes of data are being delivered to PCs on high-speed connections, it may be possible to omit compression.  
  **下载时间**：仅当 AssetBundle 较大或用户处于带宽受限的环境中（例如在低速或按流量计费的连接上下载）时，才可能需要压缩。如果只有几十兆字节的数据通过高速连接传送到PC，则可以省略压缩。

### 5.6.1. Crunch Compression 紧缩压缩

Bundles which consist primarily of DXT-compressed textures which use the Crunch compression algorithm should be built uncompressed.  
主要由使用 Crunch 压缩算法的 DXT 压缩纹理组成的捆绑包应构建为未压缩的。

## 5.7. AssetBundles and WebGL

All AssetBundle decompression and loading in a WebGL project must occur on the main thread, due to Unity’s WebGL export option not currently supporting worker threads. The downloading of AssetBundles is delegated to the browser using XMLHttpRequest. Once downloaded, compressed AssetBundles will be decompressed on Unity’s main thread, therefore stalling execution of the Unity content depending on the size of the bundle.  
由于 Unity 的 WebGL 导出选项当前不支持工作线程，WebGL 项目中的所有 AssetBundle 解压缩和加载都必须在主线程上进行。AssetBundles 的下载使用 XMLHttpRequest 委托给浏览器。下载后，压缩的资源包将在 Unity 的主线程上解压缩，因此根据捆绑包的大小延迟 Unity 内容的执行。

Unity recommends that developers prefer small asset bundles to avoid incurring performance issues. This approach will also be more memory efficient than using large asset bundles. Unity WebGL only supports LZ4-compressed and uncompressed asset bundles, however, it is possible to apply gzip/brotli compression on the bundles generated by Unity. In that case you will need to configure your web server accordingly so that the files are decompressed on download by the browser. See [here](https://docs.unity3d.com/Manual/webgl-deploying.html) for more details.  
Unity 建议开发人员首选小型资源包，以避免出现性能问题。这种方法还将比使用大型资产包更节省内存。Unity WebGL 仅支持 LZ4 压缩和未压缩的资源包，但是，可以在 Unity 生成的捆绑包上应用 gzip/brotli 压缩。在这种情况下，您需要相应地配置Web服务器，以便在浏览器下载时解压缩文件。有关更多详细信息，请参阅[此处](https://docs.unity3d.com/Manual/webgl-deploying.html)。

If you are using Unity 5.5 or older, consider avoiding LZMA for your AssetBundles and compress using LZ4 instead, which is decompressed very efficiently on-demand. Unity 5.6 removes LZMA as a compression option for the WebGL platform.  
如果您使用的是 Unity 5.5 或更早版本，请考虑避免将 LZMA 用于资源包，而是使用 LZ4 进行压缩，LZ4 可以非常有效地按需解压缩。Unity 5.6删除了LZMA作为WebGL平台的压缩选项。