

参考链接:

官方文档: <https://docs.unity3d.com/Manual/AssetBundlesIntro.html>

大神博客: <https://davidsheh.github.io/> 之Unity中的AssetBundle详解

## AssetBundles

AssetBundle是一个存档文件，其中包含平台在运行时加载的特定资产（模型，纹理，预制，音频剪辑，甚至整个场景）。AssetBundles可以表示彼此之间的依赖关系;例如AssetBundle A中的一个材质可以引用AssetBundle B中的一个纹理。为了通过网络进行有效的传递，可以根据用例要求，选择内置算法（LZMA和LZ4）来对AssetBundles进行压缩。

AssetBundles可用于可下载内容（DLC），减少初始安装大小，加载为最终用户平台优化的资产，并降低运行时内存压力。

### 一个AssetBundle文件中有什么？

好的问题，实际上“AssetBundle”可以指两个不同但有关的事情。

第一个是磁盘上的实际文件。我们把这叫做AssetBundle存档，或在本文档只是短期档案。存档可以被认为是一个容器，就像一个文件夹，在其中保存其他文件。这些附加文件包括两种类型：序列化文件和资源文件。序列化的文件将您的资产分成各自的对象，并写入这个文件。资源文件只是存储在某些资产（纹理和音频）中的二进制数据块，允许我们在另一个线程上有效地将其从磁盘加载到内存。

第二个是您通过代码从特定存档加载资源的实际的AssetBundle对象。此对象包含您添加到此存档的资源的所有文件路径的映射到属于该资产的对象，该对象在需要时需要加载。（This object contains a map of all the file paths of the assets you added to this archive to the objects that belong to that asset that need to be loaded when you ask for it.）

### AssetBundle工作流程

要开始使用AssetBundles，请按照下列步骤操作。有关每个工作流程的更详细信息，请参见本部分文档中的其他页面。

### 将资产分配给AssetBundles

要将一个给定的资产分配给一个AssetBundle，请按照下列步骤操作：

1. 从“项目视图”中选中要分配给一个bundle的资产
2. 检查Inspector视图中的对象
3. 在Inspector视图的底部，可以看到一个部分来分配AssetBundles和Variants
4. 左侧的下拉菜单分配AssetBundle，而右侧的下拉菜单则指定Variants
5. 点击左侧的下拉菜单，其中显示“None”以显示当前注册的AssetBundle名称
6. 如果尚未创建，你将看到上面的图像中的列表

7. 点击 “New...” 创建一个新的AssetBundle
8. 键入所需的AssetBundle名称。请注意，AssetBundle名称支持一种类型的文件夹结构，具体取决于你键入的内容。要添加子文件夹，请使用 “/” 分隔文件夹名称。例如：AssetBundle名称 “environment/forest” 将在environment子文件夹下创建一个名为forest的bundle
9. 一旦你选择或创建了一个AssetBundle名称，你可以重复此过程，为右侧下拉菜单分配或创建一个不同的名称，以分配或创建一个Variant名称，如果你愿意的话。在构建AssetBundles时，Variant名称不是必需的。

要了解有关AssetBundle分配和随附策略的更多信息，请参阅[Preparing Assets for AssetBundles](#)的文档。

## 构建AssetBundles

在项目中的Assets文件夹下创建一个名为Editor的文件夹，并在文件夹中放置以下内容的脚本：

```
1 using UnityEditor;
2
3 public class CreateAssetBundles
4 {
5     [MenuItem("Assets/Build AssetBundles")]
6     static void BuildAllAssetBundles()
7     {
8         string assetBundleDirectory = "Assets/AssetBundles";
9         if(!Directory.Exists(assetBundleDirectory))
10         {
11             Directory.CreateDirectory(assetBundleDirectory);
12         }
13         BuildPipeline.BuildAssetBundles(assetBundleDirectory, BuildAssetBundleOptions.None, BuildTarget.Standalone);
14     }
15 }
```

该脚本将在Assets菜单的底部创建一个名为 “Build AssetBundles” 的菜单项，该菜单项将执行与该标记关联的功能中的代码。当您单击Build AssetBundles时，进度条将显示一个构建对话框。这将使那些用AssetBundle名称标记的所有资产打包进同一个文件，并将它们放在由assetBundleDirectory定义的路径上的文件夹中。

有关此代码正在执行的更多详细信息，请参阅有关[Building AssetBundles](#)的文档。

## 上传AssetBundles到非本地存储

这一步对于每个用户都是独一无二的，而不是一步一步可以告诉你如何做。如果您打算将AssetBundles上传到第三方托管网站，请在此处进行。如果您正在严格执行本地开发，并打算将所有AssetBundles都放在磁盘上，请跳到下一步。

## 加载AssetBundles和Assets

对于有意从本地存储加载的用户，您将对AssetBundles.LoadFromFile API感兴趣。看起来像这样：

```

1 public class LoadFromFileExample extends MonoBehaviour {
2     function Start() {
3         var myLoadedAssetBundle = AssetBundle.LoadFromFile(Path.Combine(Application.streamingAssetsPath, "myassetBundle"));
4         if (myLoadedAssetBundle == null) {
5             Debug.Log("Failed to load AssetBundle!");
6             return;
7         }
8         var prefab = myLoadedAssetBundle.LoadAsset<GameObject>("MyObject");
9         Instantiate(prefab);
10    }
11 }

```

LoadFromFile获取包文件的路径。

如果您自己托管AssetBundles并且需要将其下载到游戏中，那么您将对UnityWebRequest API感兴趣。这里有一个例子：

```

IEnumerator InstantiateObject()
{
    string uri = "file:/// " + Application.dataPath + "/AssetBundles/" + assetBundleName;
    UnityEngine.Networking.UnityWebRequest request = UnityEngine.Networking.UnityWebRequest.GetAssetBundle(uri, 0);
    yield return request.Send();
    AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(request);
    GameObject cube = bundle.LoadAsset<GameObject>("Cube");
    GameObject sprite = bundle.LoadAsset<GameObject>("Sprite");
    Instantiate(cube);
    Instantiate(sprite);
}

```

GetAssetBundle(string, int)获取AssetBundle的位置的uri以及要下载的包的版本。在这个例子中，我们仍然指向一个本地文件，但是字符串uri可以指向你托管AssetBundles的任何URL。

UnityWebRequest 具有处理 AssetBundles 的特定句柄（DownloadHandlerAssetBundle），DownloadHandlerAssetBundle 从请求中获取 AssetBundle。

无论使用的方法如何，你现在都可以访问AssetBundle对象。从该对象中加载资源，你将需要使用LoadAsset<T>(string)方法，该方法中泛型类型T表示你所要加载的Asset的类型，方法参数为所要加载的Asset对象的名称。这将返回您从AssetBundle加载的任何对象。您可以像Unity中的任何对象一样使用这些返回的对象。例如，如果要在场景中创建一个GameObject，则只需要调用Instantiate（gameObjectFromAssetBundle）。

## 为AssetBundles准备资源

使用AssetBundles时，您可以随意将任何Asset分配给所需的任何Bundle。但是，在设置Bundles时，需要考虑一些策略。这些分组策略可以使用到任何你认为适合的特定项目中。你可以随心所欲地混合和匹配这些策略。

### 逻辑实体分组

逻辑实体分组是根据其所代表的项目的功能部分将资产分配给AssetBundles的。这包括诸如用户界面、人物、环境以及在整个应用程序的整个生命周期中频繁出现的其他部分。

## 例子

- 把用户界面的所有纹理和布局数据打包到一起
- 把人物角色所包含的所有的模型和动画打包到一起
- 把多个关卡中共用的纹理和模型打包到一起

逻辑实体分组是可下载内容（DLC）的理想选择，因为以这种方式分离的所有内容，您可以对单个实体进行更改，而不需要下载其他未更改的资产。

能够正确实施这一策略的最大的窍门是，开发人员将资产分配给各自的Bundle必须熟悉项目中每个资产的使用时间和位置。

## 类型分组

对于此策略，您可以将相似类型的Assets（如音轨或语言本地化文件）分配给单个AssetBundle。

类型分组是构建要由多个平台使用的AssetBundles的更好策略之一。例如，如果您的音频压缩设置在Windows和Mac平台之间是相同的，您可以将所有音频数据自己打包到AssetBundles中，并重复使用这些Bundles，而着色器则倾向于使用更多的平台特定选项进行编译，因此你为Mac构建的着色器包可能不能在Windows上重复使用。此外，这种方法非常适合使你的AssetBundles与更多的Unity版本兼容，如同纹理压缩格式和设置的更改频率比你的脚本或预制体的更改频率更低。

## 并发内容分组（Concurrent Content Grouping）

并发内容分组是将要同时加载和使用的Assets捆绑在一起的想法。您可以将这些类型的Bundles用于基于关卡的游戏，其中每个关卡都包含完全独特的角色、纹理、音乐等。你要绝对肯定的是，包含在这些AssetBundles中一个Asset被使用的同时，其余的Assets也会被使用。对并发内容分组捆绑中的单个资产的依赖将导致加载时间的显著增加。你将因为这个单一Asset而被迫下载整个Bundle。

并发内容分组的Bundles最常用的用例是基于场景的Bundles。在此分配策略中，每个场景Bundle应该包含大部分或全部场景依赖关系。

注意，一个项目完全可以并且应该根据需要来组合这些策略。对于任何给定的场景使用最优资产分配策略可以极大地提高任何项目的效率。

例如，一个项目可能决定将其用于不同平台的用户界面（UI）元素分组到他们自己的特定于平台的UI Bundle中，但按照关卡或场景对其交互式内容进行分组。

不管你遵循什么策略，这里有一些额外的建议值得你铭记在心：

- 将经常更新的对象拆分成AssetBundles，这些对象与很少更改的对象分开
- 组合可能同时加载的对象。如一个模型，它的纹理和动画

- 如果你注意到多个AssetBundles中的多个对象取决于完全不同的AssetBundle中的单个资产，请将依赖关系移动到单独的AssetBundle。如果几个AssetBundles引用了其他AssetBundles中的同一组资产，可能值得将这些依赖关系拉入共享的AssetBundle以减少重复。
- 如果两套对象不太可能同时加载，例如标清和高清资产，请确保它们在自己的资产组合中。
- 如果一个AssetBundle中经常有少于50%的Bundle在同一时间加载，那么可以考虑拆分该AssetBundle。
- 如果有一些小的AssetBundles（少于5到10个资产的）经常同时被加载，可以考虑组合这些AssetBundles。
- 如果一组对象只是同一对象的不同版本，可考虑使用AssetBundle Variants

## 构建AssetBundles

在 [AssetBundle 工作流程](#) 的文档中，我们有一个示例代码，它将三个参数传递给 `BuildPipeline.BuildAssetBundles` 函数。让我们更深入地了解我们实际上在说什么。

*Assets/AssetBundles*: 这是AssetBundles将被输出到的目录。您可以将其更改为所需的任何输出目录，只需在尝试构建之前确保文件夹实际存在。

### BuildAssetBundleOptions

有多种不同的 `BuildAssetBundleOptions` 选项可以选择。相关的各个选项可以参阅脚本API文档中的关于 [BuildAssetBundleOptions](#) 的内容。

虽然随着需求的变化和增加，您可以自由组合 `BuildAssetBundleOptions` 选项，但有三个特定的 `BuildAssetBundleOptions` 是用来处理AssetBundle压缩：

- `BuildAssetBundleOptions.None`: 此bundle选项使用LZMA格式压缩，这种压缩是序列化的数据文件的单一压缩的LZMA流。LZMA压缩的文件在使用前需要对整个bundle解压缩。这导致最小的文件大小和由于解压缩而稍微增加的加载时间。值得注意的是，当使用此 `BuildAssetBundleOptions` 时，为了使用捆绑包中的任何资源，必须首先解压缩整个捆绑包。一旦bundle解压缩后，将使用LZ4在磁盘上重新压缩，LZ4压缩在使用bundle中的资产时，不需要提前对整个bundle进行解压缩。这最适合用于bundle中的资产，以便从bundle中使用一个资产将意味着所有资产将被加载。打包角色或场景的所有资源都是可能使用的捆绑包的一些示例。由于较小的初始文件大小，利用LZMA压缩仅推荐用于从异地主机下载资源包。一旦文件被下载，它会缓存为lz4压缩包。
- `BuildAssetBundleOptions.UncompressedAssetBundle`: 此bundle选项以数据完全未压缩的方式构建bundle。未压缩的缺点是较大的文件下载大小。但是，一旦下载的加载时间会更快。
- `BuildAssetBundleOptions.ChunkBasedCompression`: 此bundle选项使用称为LZ4的压缩方法，这导致比LZMA更大的压缩文件大小，但不像LZMA那样在使用之前不需要整个捆绑包解压缩。

LZ4使用基于块的算法，允许将AssetBundle以切片（pieces）或“块”（chunks）的形式加载。解压缩单个块允许使用包含的资产，即使AssetBundle的其他块未被解压缩。使用ChunkBasedCompression与未压缩的bundle具有可比较的加载时间，还具有减小磁盘大小的附加优势。

## BuildTarget

**BuildTarget.Standalone:** 在这里，我们正在告诉构建管道，我们将使用这些AssetBundles的目标平台。

您可以在[BuildTarget](#)的API的脚本参考文档中找到可用的显式构建目标的列表。但是，如果您不希望构建目标中进行硬编码，则可以随时利用EditorUserBuildSettings.activeBuildTarget，这将自动找到你目前的设置构建和构建AssetBundles基于的目标平台。

一旦正确设置了构建脚本，就可以构建bundles了。如果您按照上述脚本示例，请单击**Assets > Build AssetBundles**以启动该过程。

现在你已经成功构建了AssetBundles，你可能会注意到你的AssetBundles目录有可能比你预期更多的文件。确切地说，是 $2 * (n + 1)$  个文件。让我们花一点时间来看看BuildPipeline.BuildAssetBundles的产量。

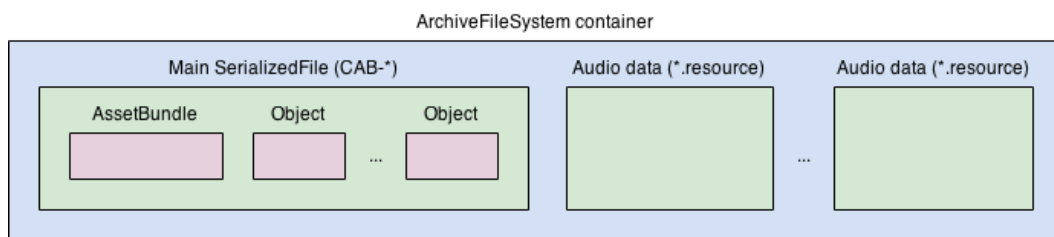
对于你在编辑器中指定的每个AssetBundle，你将注意到一个包含AssetBundle name 和 AssetBundle name +.manifest的文件。

将会有有一个额外的bundle和manifest，它不会与您创建的任何AssetBundle共享一个名称。而是以它位于的目录（AssetBundles构建到的目录）命名。这是Manifest Bundle。我们将在以后讨论更多的内容以及如何使用它。

## The AssetBundle File

这是缺少.manifest扩展名的文件，以及你在运行时加载的内容，以加载你的资产。

AssetBundle文件是一个在内部包含多个文件的存档。此存档的结构可能会稍有变化，具体取决于它是AssetBundle还是场景AssetBundle。这是一个正常的AssetBundle的结构：



ArchiveFileSystem

场景AssetBundle更改自标准的AssetBundles，因为它针对场景及其内容的流加载进行了优化。该图像显示场景bundle的内部结构：

## The Manifest File

对于生成的每个包，包括附加的清单包，都会生成关联的清单文件。清单文件可以使用任何文本编辑器打开，并且包含诸如循环冗余校验（CRC）数据和包的依赖性数据的信息。对于正常的AssetBundles，他们的清单文件将如下所示：



```

ManifestFileVersion: 0
CRC: 2422268106
Hashes:
  AssetFileHash:
    serializedVersion: 2
    Hash: 8b6db55a2344f068cf8a9be0a662ba15
  TypeTreeHash:
    serializedVersion: 2
    Hash: 37ad974993dbaa77485dd2a0c38f347a
HashAppended: 0
ClassTypes:
- Class: 91
  Script: {instanceID: 0}
Assets:
  Asset_0: Assets/Mecanim/StateMachine.controller
Dependencies: {}

Which shows the contained assets, dependencies, and other information.

The Manifest Bundle that was generated will have a manifest, but it'll look more like this:

ManifestFileVersion: 0
AssetBundleManifest:
  AssetBundleInfos:
    Info_0:
      Name: scene1assetbundle
      Dependencies: {}

```

这将显示AssetBundles如何关联以及它们的依赖关系。现在，只要明白这个bundle包含了AssetBundleManifest对象，这对于找出在运行时加载哪个bundle依赖是非常有用的。要了解有关如何使用此bundle和manifest对象的更多信息，请参阅[使用本地AssetBundles](#)的文档。

## AssetBundle依赖关系

如果一个或多个UnityEngine.Objects包含位于另一个bundle中的UnityEngine.Object的引用，则AssetBundles可以依赖于其他AssetBundles。如果UnityEngine.Object包含一个在其他任何AssetBundle中都不包含的UnityEngine.Object的引用，则不会发生依赖关系。在这种情况下，在构建AssetBundles时，将bundle所依赖的对象的副本复制到捆绑包中。如果多个bundle中的多个对象包含对未分配给bundle的同一对象的引用，那么对该对象具有依赖关系的每个bundle将各自制作一个该对象的副本并将其打包到内置的AssetBundle中。

如果AssetBundle包含依赖关系，则在加载要尝试实例化的对象之前，加载包含这些依赖关系的bundles是重要的。Unity不会尝试自动加载依赖关系。

考虑以下示例，**Bundle 1**中的材料引用了**Bundle 2**中的纹理：

在此示例中，加载**Bundle 1**中的Material之前，需要将**Bundle 2**加载到内存中。**Bundle 1**和**Bundle 2**的加载顺序无关紧要，重要的是在从**Bundle 1**加载Material之前加载**Bundle 2**。在下一节中，我们将讨论如何使用在[上一篇博客](#)中所涉及的AssetBundleManifest对象，在运行时确定和加载依赖关系。

## 使用本地AssetBundles

在 Unity 5 中，我们可以使用四种不同的 API 来加载。它们的行为根据正在加载的平台和AssetBundles 构建时使用的压缩方式（未压缩，LZMA，LZ4）而有所不同。

我们需要用到的四个 API 是：

- [AssetBundle.LoadFromMemoryAsync](#)
- [AssetBundle.LoadFromFile](#)
- [WWW.LoadFromCacheOrDownload](#)
- [UnityWebRequest](#)' s [DownloadHandlerAssetBundle](#) (Unity 5.3 或者更高的版本)

## AssetBundle.LoadFromMemoryAsync

### [AssetBundle.LoadFromMemoryAsync](#)

此函数使用包含 AssetBundle 数据的字节数组的参数。如果需要也可以传递一个 CRC 值。如果 Bundle 是 LZMA 压缩的，它将在加载时解压缩 AssetBundle。 LZ4 压缩的 Bundle 在压缩状态时被加载。

以下是使用此方法的一个示例：

```
IEnumerator LoadFromMemoryAsync(string path)
{
    AssetBundleCreateRequest createRequest = AssetBundle.LoadFromMemoryAsync(File.ReadAllBytes(path));

    yield return createRequest;

    AssetBundle bundle = createRequest.assetBundle;

    var prefab = bundle.LoadAsset<GameObject>("MyObject");
    Instantiate(prefab);
}
```

但是，这不是唯一可以使用 LoadFromMemoryAsync 的策略。可以用任何获取所需的字节数组的过程替代 File.ReadAllBytes(path) 方法。

## AssetBundle.LoadFromFile

### [AssetBundle.LoadFromFile](#)

从本地存储加载未压缩的 Bundles 时，此 API 非常高效。如果 Bundle 是未压缩或块（LZ4）压缩的，LoadFromFile 将直接从磁盘加载 Bundle。使用此方法加载完全压缩（LZMA）的 Bundle 将首先解压缩包，然后再将其加载到内存中。

如何使用 LoadFromFile 的一个例子：



```

1 public class LoadFromFileExample extends MonoBehaviour {
2     function Start() {
3         var myLoadedAssetBundle = AssetBundle.LoadFromFile(Path.Combine(Application.streamingAssetsPath, "myassetBundle"));
4         if (myLoadedAssetBundle == null) {
5             Debug.Log("Failed to load AssetBundle!");
6             return;
7         }
8         var prefab = myLoadedAssetBundle.LoadAsset<GameObject>("MyObject");
9         Instantiate(prefab);
10    }
11 }

```

注意：在使用 Unity 5.3 或更早版本的Android设备上，尝试从 Streaming Assets 路径加载 AssetBundles 时，此API将失效。这是因为该路径的内容将驻留在压缩的 .jar 文件中。Unity 5.4和更新版本可以使用这个API 加载 Streaming Assets 路径下的资源。

## WWW.LoadFromCacheOrDownload

### WWW.LoadFromCacheOrDownload

#### **此 API 将被废弃（请使用 UnityWebRequest）**

此 API 可用于从远程服务器下载 AssetBundles 或加载本地 AssetBundles。它是 UnityWebRequest API 的较旧且不太理想的版本。

从远程位置加载 AssetBundle 将自动缓存 AssetBundle。如果AssetBundle被压缩，那么一个工作线程将自动解压缩该包并将其写入缓存。一旦捆绑包解压缩并缓存，它将像 AssetBundle.LoadFromFile 一样加载。

如何使用 LoadFromCacheOrDownload 的一个例子：

```

1 using UnityEngine;
2 using System.Collections;
3
4 public class LoadFromCacheOrDownloadExample : MonoBehaviour
5 {
6     IEnumerator Start ()
7     {
8         while (!Caching.ready)
9             yield return null;
10
11         var www = WWW.LoadFromCacheOrDownload("http://myserver.com/myassetBundle", 5);
12         yield return www;
13         if(!string.IsNullOrEmpty(www.error))
14         {
15             Debug.Log(www.error);
16             yield return;
17         }
18         var myLoadedAssetBundle = www.assetBundle;
19
20         var asset = myLoadedAssetBundle.mainAsset;
21     }
22 }

```

由于缓存 AssetBundle 在 WWW 对象中的字节的内存开销，建议使用 WWW.LoadFromCacheOrDownload

的所有开发人员确保其 AssetBundles 保持较小——最多为几兆字节。还建议在限制内存平台（如移动设备）上运行的开发人员确保其代码一次只下载一个 AssetBundle，以避免内存尖峰。

如果缓存文件夹没有用于缓存附加文件的空间，LoadFromCacheOrDownload将从缓存中迭代删除最近最少使用的AssetBundle，直到有足够的空间可用于存储新的AssetBundle。如果无法进行空间（因为硬盘已满，或者当前正在使用缓存中的所有文件）释放，LoadFromCacheOrDownload() 将绕过缓存并将文件以流的形式加入内存。

为了强制 LoadFromCacheOrDownload，版本参数（第二个参数）将需要更改。如果传递给该函数的版本与当前缓存的AssetBundle的版本匹配，则AssetBundle将仅从缓存加载。

## UnityWebRequest

### [UnityWebRequest](#)

UnityWebRequest 有一个特定的 API 调用来处理 AssetBundles。首先，你需要使用 UnityWebRequest.GetAssetBundle 创建您的 Web 请求。返回请求后，将请求对象传递给 DownloadHandlerAssetBundle.GetContent(UnityWebRequest)。此 GetContent 函数调用将返回 AssetBundle 对象。

你还可以在下载 Bundle 后使用 [DownloadHandlerAssetBundle](#) 类中的 `assetBundle` 属性，以 `AssetBundle.LoadFromFile` 的效率加载 AssetBundle。

以下是一个如何加载包含两个 GameObject 的 AssetBundle 并实例化它们的示例。要开始这个过程，我们只需要调用 `StartCoroutine(InstantiateObject());`

```
1 IEnumerator InstantiateObject()
2 {
3     string uri = "file:/// " + Application.dataPath + "/AssetBundles/" + assetBundleName;
4     UnityEngine.Networking.UnityWebRequest request = UnityEngine.Networking.UnityWebRequest.GetAssetBundle(uri, 0);
5     yield return request.Send();
6     AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(request);
7     GameObject cube = bundle.LoadAsset<GameObject>("Cube");
8     GameObject sprite = bundle.LoadAsset<GameObject>("Sprite");
9     Instantiate(cube);
10    Instantiate(sprite);
11 }
```

使用 UnityWebRequest 的优点是它允许开发人员以更灵活的方式处理下载的数据，并可能消除不必要的内存使用情况。这是 UnityEngine.WWW 类中首选的 API。

## 从 AssetBundles 载入资产

现在，您已经成功下载了 AssetBundle，是时候最终加载某些资产了。

通用代码段：

**T objectFromBundle = bundleObject.LoadAsset<T>(assetName);**

**T** 是尝试加载的资产的类型。

决定如何加载资产时，有几种选择。分别是 LoadAsset、LoadAllAssets 和它们的对应的异步方法 LoadAssetAsync 和 LoadAllAssetsAsync。

下面代码展示了如何从AssetBundles同步加载资产：

加载单个 GameObject：

**GameObject gameObject = loadedAssetBundle.LoadAsset<GameObject>(assetName);**

加载所有资产：

```
Unity.Object[] objectArray = loadedAssetBundle.LoadAllAssets();
```

现在，正如上文所示的方法返回的对象类型或要加载的对象的数组，异步方法返回一个 [AssetBundleRequest](#)。访问资产之前，需要等待此操作完成。加载一个 Asset：

```
1 AssetBundleRequest request = loadedAssetBundle.LoadAssetAsync<GameObject>(assetName);
2 yield return request;
3 var loadedAsset = request.asset;
```

以及

```
AssetBundleRequest request = loadedAssetBundle.LoadAllAssetsAsync();
yield return request;
var loadedAssets = request.allAssets;
```

一旦你加载了你所需要的 Assets，接下来就可以像使用 Unity 中任何其他对象一样使用加载的对象。

## 加载 AssetBundle 清单文件 (Manifests)

加载 AssetBundle 清单文件非常有用。特别是在处理 AssetBundle 依赖项时。

要获得一个可用的 [AssetBundleManifest](#) 对象，你需要加载额外的 AssetBundle（名称与其所在文件夹相同），并从中加载类型为 AssetBundleManifest 的对象。

加载清单本身与加载 AssetBundle 中的任何其他资产完全相同：

```
1 AssetBundle assetBundle = AssetBundle.LoadFromFile(manifestFilePath);
2 AssetBundleManifest manifest = assetBundle.LoadAsset<AssetBundleManifest>("AssetBundleManifest");
```

现在，你可以通过上述示例中的 manifest 对象访问 [AssetBundleManifest](#) API 的调用。从这里你可以使用清单来获取有关你所构建的 AssetBundles 的信息。该信息包括 AssetBundles 的依赖关系数据、散列数据和变体数据。

在前面的部分，当我们讨论 AssetBundle Dependencies 并且如果一个 bundle 对另一个 bundle 有依赖关系，那么在从原始 bundle 加载任何资源之前，需要加载这些 bundle。清单对象使得动态地找到加载依赖性成为可能。假设我们要加载名为 “assetBundle” 的 AssetBundle 的所有依赖项。

```
1 AssetBundle assetBundle = AssetBundle.LoadFromFile(manifestFilePath);
2 AssetBundleManifest manifest = assetBundle.LoadAsset<AssetBundleManifest>("AssetBundleManifest");
3 string[] dependencies = manifest.GetAllDependencies("assetBundle"); //Pass the name of the bundle you want the dependency
4 foreach(string dependency in dependencies)
5 {
6     AssetBundle.LoadFromFile(Path.Combine(assetBundlePath, dependency));
7 }
```

上面代码表示正在加载 AssetBundles、AssetBundle依赖关系和资产，现在是时候讨论管理所有这些加载的 AssetBundles 了。

## 管理加载的 AssetBundles

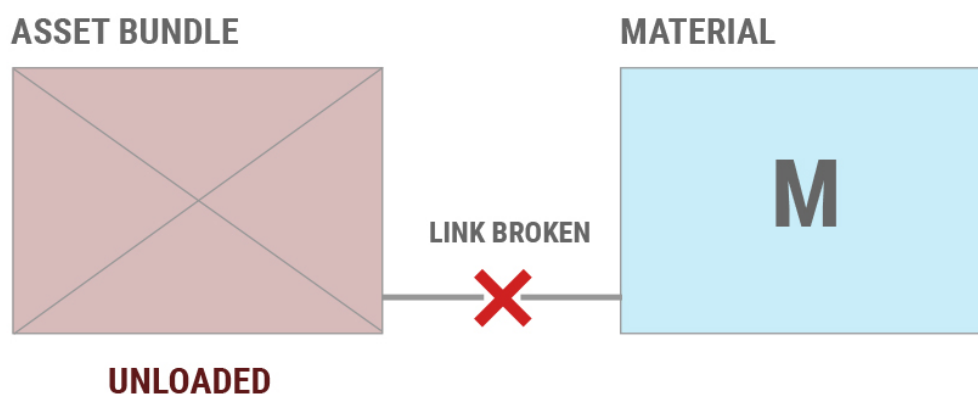
另请参阅：Unity 官方教程中有关 [Managing Loaded AssetBundles](#)的教程。

当它们从活动场景中删除时，Unity不会自动卸载对象。资产清理在特定时间触发，也可以手动触发。知道何时加载和卸载 AssetBundle 很重要。不正确卸载 AssetBundle 可能会导致内存中的对象复制或其他不合要求的情况（如缺少纹理）。

了解 AssetBundle 管理最重要的是什么时候调用 `AssetBundle.Unload(bool)`，并且在函数调用时你应该将 `true` 或 `false` 作为参数传递给函数。卸载 AssetBundle 是一个非静态的功能，此 API 卸载正在调用的 AssetBundle 的头部信息。该函数的参数表示是否也卸载从此 AssetBundle 实例化的所有对象。

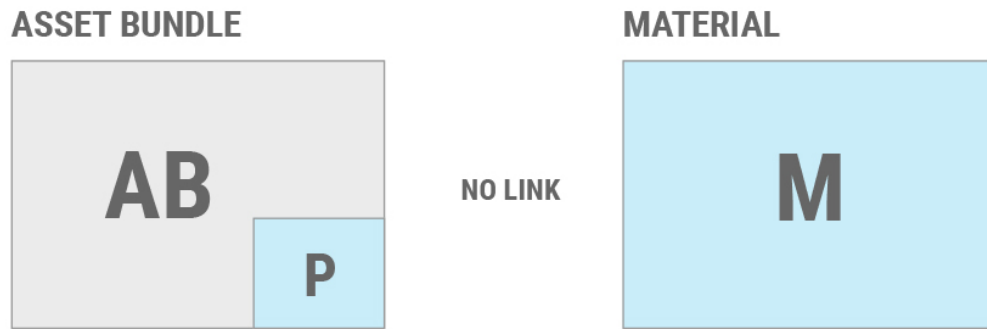
如果使用 `AssetBundle.Unload(true)` 方法，就会卸载从 AssetBundle 中加载的所有对象，即使它们正在当前活动的场景中使用。这正是我们前面提到的，这可能会导致纹理丢失。我们假设材质 M 是从 AssetBundle AB 加载的，如下所示。如果调用 `AB.Unload (true)`。活动场景中的任何 M 实例也将被卸载和销毁。如果你改为调用 `AB.Unload (false)`，它会破坏当前 M 和 AB 实例的链接。

## AFTER UNLOAD(FALSE)

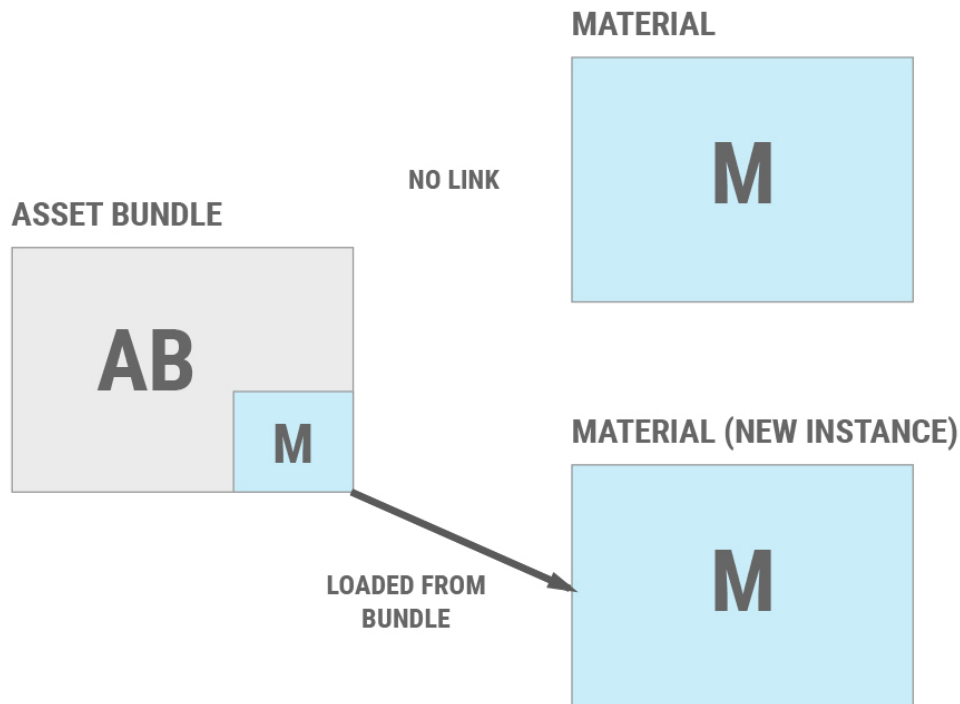


如果以后再次加载 AB，并调用 `AB.LoadAsset()`，Unity 将不会将 M 的现有副本重新链接到新加载的材质。那么 M 会加载两个副本。

## AFTER ASSET BUNDLE RELOAD



## AFTER LOADING PREFAB AGAIN



一般来说，使用 `AssetBundle.Unload(false)` 并不会导致理想的情况。大多数项目应该使用 `AssetBundle.Unload(true)` 来避免在内存中复制对象。

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被重复。两种常见的方法是：

- 在应用程序的生命周期中具有明确定义的地方，卸载临时 `AssetBundles`，例如关卡之间或加载屏幕期间。
- 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卸载 `AssetBundles`。这允许应用程序卸载和重新加载单个对象而没有重复的内存。

如果应用程序必须使用 `AssetBundle.Unload(false)`，那么单个对象只能以两种方式卸载：

- 在场景和代码中消除对不需要的对象的所有引用。完成之后，调用 [Resources.UnloadUnusedAssets](#)。
- 使用非叠加式加载场景。这将销毁当前场景中的所有对象并自动调用 [Resources.UnloadUnusedAssets](#)。

如果你不想自己管理加载 `AssetBundles`、依赖关系和 `Assets` 本身，你可能会发现自己需要 `AssetBundle Manager`。

## AssetBundle 管理器

`AssetBundle Manager` 可以在[这里](#)下载，是由 Unity 制作的一种工具，可以使 `AssetBundles` 更加精简。下载和导入 `AssetBundle Manager` 软件包不仅添加了一个新的 API 调用来加载和使用 `AssetBundles`，而且还添加了一些编辑器功能来简化工作流。此功能可以在“资产”菜单选项下找到。此新部分将包含以下选项：






### 模拟模式

启用模拟模式允许 `AssetBundle Manager` 使用 `AssetBundles`，但不需要实际构建捆绑包。编辑器会看到 `Assets` 已分配给 `AssetBundles`，并直接使用 `Assets`，而不是实际上从 `AssetBundle` 中提取 `Assets`。

使用模拟模式的主要优点是 `Assets` 可以被修改、更新、添加和删除，而不需要每次重新构建和部署 `AssetBundles`。值得注意的是，`AssetBundle` 变体不适用于模拟模式。如果你需要使用变体，则你需要使用本地 `AssetBundle` 服务器。

### 本地 AssetBundle 服务器

`AssetBundle Manager` 还可以启动一个本地 `AssetBundle` 服务器，该服务器可用于在编辑器或本地构建（包括移动端）中测试 `AssetBundles`。获取本地 `AssetBundle` 服务器工作的规定是必须在项目的根目录中创建一个名为 `AssetBundles` 的文件夹，该文件夹与 `Assets` 文件夹的级别相同。如下图：

Name	Date modified	Type	Size
 AssetBundles	22/07/2015 00:38	File folder	
 Assets	20/07/2015 20:26	File folder	
 Library	22/07/2015 00:38	File folder	
 ProjectSettings	22/07/2015 00:38	File folder	
 Temp	22/07/2015 00:38	File folder	

创建文件夹后，你需要将 `AssetBundles` 构建到此文件夹。为此，请从新菜单选项中选择 `Build AssetBundles`。这将为你构建它们到该目录。现在你已经建立了 `AssetBundles`（或已经决定使用模拟模式），并准备开始加载 `AssetBundles`。我们来看看 `AssetBundle Manager` 对我们提供的新的 API 调用。

### `AssetBundleManager.Initialize()`



此函数加载 AssetBundleManifest 对象。在使用 AssetBundle Manager 开始加载资产之前，你需要调用它。在一个非常简单的例子中，初始化 AssetBundle Manager 可能如下所示：

```
1 IEnumerator Start()
2 {
3     yield return StartCoroutine(Initialize());
4 }
5 IEnumerator Initialize()
6 {
7     var request = AssetBundleManager.Initialize();
8     if (request != null)
9         yield return StartCoroutine(request);
10 }
```

AssetBundle Manager 使用你在 Initialize() 期间加载的清单来帮助幕后的许多功能，包括依赖关系管理。

## Loading Assets

你正在使用 AssetBundle Manager，你已初始化它，现在你可以加载某些资产。我们来看看如何加载 AssetBundle 并从该 Bundle 中实例化一个对象：

```
IEnumerator InstantiateGameObjectAsync (string assetBundleName, string assetName)
{
    // Load asset from assetBundle.
    AssetBundleLoadAssetOperation request = AssetBundleManager.LoadAssetAsync(assetBundleName, assetName, typeof(GameObject));
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
    // Get the asset.
    GameObject prefab = request.GetAsset<GameObject> ();
    if (prefab != null)
        GameObject.Instantiate(prefab);
}
```

AssetBundle Manager 会异步执行所有的加载操作，因此它返回一个加载操作请求，它在调用 `yield return StartCoroutine (request)` 时加载 Bundle;接着我们需要做的是调用 `GetAsset<T>()` 从 AssetBundle 加载游戏对象。

## Loading Scenes

如果你有一个 AssetBundle 名称分配给场景，并且你需要加载该场景，则需要遵循稍微不同的代码路径。模式是相同的，但有微小的差异。以下是从 AssetBundle 加载场景的方法：

```
IEnumerator InitializeLevelAsync (string levelName, bool isAdditive)
{
    // Load level from assetBundle.
    AssetBundleLoadOperation request = AssetBundleManager.LoadLevelAsync(sceneAssetBundle, levelName, isAdditive);
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
}
```

正如你所看到的，加载场景也是一个异步的，LoadLevelAsync返回一个加载操作请求，需要传递给一个StartCoroutine 才能加载场景。

## Load Variants

使用 AssetBundle Manager 加载变量实际上并不会改变在场景或资产中加载的代码。所有需要完成的操作都是设置 AssetBundleManager 的 ActiveVariants 属性。ActiveVariants 属性是一个字符串数组。只需构建一个字符串数组，其中包含在将它们分配给资产时创建的变体名称。以下是如何使用 hd 变体加载场景 AssetBundle。

```
IEnumerator InitializeLevelAsync (string levelName, bool isAdditive, string[] variants)
{
    //Set the activeVariants.
    AssetBundleManager.ActiveVariants = variants;
    // Load level from assetBundle.
    AssetBundleLoadOperation request = AssetBundleManager.LoadLevelAsync(variantSceneAssetBundle, levelName, isAdditive)
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
}
```

在代码中的其他地方构建的字符串数组（可能是从按钮点击或其他一些情况）中传递的地方。如果可用，则将加载与设置的活动变体相匹配的 Bundles。

## 使用 AssetBundles 进行修补

修补 AssetBundles 与下载新的 AssetBundle 并替换已存在的 AssetBundle 一样简单。如果使用 [WWW.LoadFromCacheOrDownload](#) 或 `UnityWebRequest` 来管理应用程序的缓存 AssetBundles，则将不同的版本参数传递到所选的 API 将触发下载新的 AssetBundles。补丁系统中更难解决的问题是检测哪个 AssetBundles 需要被替换。补丁系统需要两个信息列表：

- 当前下载的 AssetBundles 的列表及其版本信息
- 服务器上的 AssetBundles 列表及其版本信息

修补程序应该下载服务器端 AssetBundles 的列表，并比较 AssetBundle 列表。应该重新下载缺少 AssetBundles 或版本信息已更改的 AssetBundles。还可以编写一个自定义系统来检测对 AssetBundles 的更改。编写自己系统的大多数开发人员都选择使用行业标准的数据格式来存储 AssetBundle 文件列表，例如JSON，以及用一个标准 C# 类来计算校验和，如 MD5。Unity 使用确定性方式排序的数据构建 AssetBundles。这允许具有自定义下载器的应用程序实现差分补丁（differential patching）。

Unity 不提供用于差分修补的任何内置机制，当使用内置缓存系统时，[WWW.LoadFromCacheOrDownload](#) 和 [UnityWebRequest](#) 都不会执行差异修补。如果需要进行差分修补，则必须写入自定义下载器。

## 故障排除

本节介绍通常在使用 AssetBundles 的项目中出现的几个问题。

### Asset 重复

当对象被构建到 AssetBundle 中时，Unity 5 的 AssetBundle 系统会发现对象的所有依赖关系。这是使用资产数据库完成的。此依赖关系信息用于确定将包含在 AssetBundle 中的对象集。明确分配给 AssetBundle 的对象只会内置到该 AssetBundle 中。当对象的 `AssetImporter` 的 `assetBundleName` 属性设置为非空字符串时，对象将被“显式分配”。在 AssetBundle 中未明确分配的任何对象将包含在所有的 AssetBundles 中，这些 AssetBundles 含有1个或多个引用未标记对象的对象。

如果两个不同的对象分配给两个不同的 AssetBundles，但都有一个公共依赖对象的引用，则该依赖对象将被复制到两个 AssetBundles 中。重复的依赖关系也将被实例化，这意味着依赖对象的两个副本将被视为具有不同标识符的不同对象。这将增加应用程序的 AssetBundles 的总大小。如果应用程序加载其父对象，这也将导致将两个不同的对象副本加载到内存中。

有几种方法来解决这个问题：

5. 确保内置到不同AssetBundles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置在相同的AssetBundle中，而不会重复它们的依赖关系。
  - 这种方法通常对于具有许多共享依赖关系的项目是不可行的。它可以生成碎片的AssetBundles，以至于必须频繁地重新构建和重新下载，而不方便和高效。
7. AssetBundles 切片，以便不会同时加载共享依赖的两个 AssetBundles。
  - 这种方法可能适用于某些类型的项目，例如基于关卡的游戏。然而，它仍然不必要地增加了项目的AssetBundles的大小，并且增加了构建时间和加载时间。
9. 确保所有依赖资产都内置在自己的 AssetBundles 中。这完全消除了重复资产的风险，但也引起了复杂性。应用程序必须跟踪 AssetBundles 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 AssetBundles。

在 Unity 5 中，通过位于 UnityEditor 命名空间中的 AssetDatabase API 跟踪对象依赖关系。正如命名空间所暗示的，此 API 仅在 Unity 编辑器中可用，而不在运行时可用。AssetDatabase.GetDependencies 可用于查找特定对象或资产的所有直接依赖关系。请注意，这些依赖关系可能有自己的依赖关系。此外，AssetImporter API 可用于查询分配给任何特定对象的 AssetBundle。

通过组合 AssetDatabase 和 AssetImporter API，可以编写一个编辑器脚本，以确保将 AssetBundle 的所有直接或间接依赖关系分配给 AssetBundles，或者两个 AssetBundles 没有共享尚未分配给 AssetBundle 的依赖关系。由于重复资产的内存成本，建议所有项目都有这样的脚本。

## 精灵图集重复

以下部分描述了与自动生成的精灵图集结合使用时，Unity 5 的资产依赖关系计算代码的一个奇怪癖。任何自动生成的精灵图集将被分配给包含生成精灵图集的 Sprite 对象的 AssetBundle。如果将精灵对象分配给多个 AssetBundles，则 Sprite 图集将不会分配给 AssetBundle，并将被复制。如果 Sprite 对象未分配给 AssetBundle，则 Sprite 图集也将不会分配给 AssetBundle。为了确保精灵地图集不被重复，请检查标记在同一精灵图集集中的所有精灵被分配到相同的资产组合。

在 Unity 5.2.2p3 及更老版本中，自动生成的精灵地图集将永远不会分配给 AssetBundle。因此，它们将被包含在任何含有其组成中的精灵的任何 AssetBundles 中，以及引用其组成中的精灵的任何 AssetBundles。

由于这个问题，强烈建议使用 Unity 的精灵打包程序的所有 Unity 5 项目升级到 Unity 5.2.2p4, 5.3 或任何新版本的Unity。

对于无法升级的项目，此问题有两种解决方法：

3. 简单的：避免使用 Unity 内置的精灵封隔器。由外部工具生成的 Sprite 地图集将是正常的资产，可以正确分配给 AssetBundle。
4. 困难的：将所有使用自动分配精灵的对象分配给与精灵相同的 AssetBundle。
  - 这将确保生成的精灵图集不被视为任何其他 AssetBundles 的间接依赖关系，不会被重复。
  - 该解决方案保留了使用 Unity 的精灵打包程序的工作流程，但是它降低了开发人员将资产分成不同 AssetBundles 的能力，并且在任何参考图集的组件上的任何数据更改时强制重新下载整个精灵图集，即使是图集本身没有变化。

## Android Textures

由于 Android 生态系统中的设备配置差距严重，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Alpha通道的纹理。应用程序不需要

OpenGL ES 2 的支持，解决问题的最简单的方法是使用所有Android OpenGL ES 3设备支持的 ETC2。

大多数应用程序需要在不支持 ETC2 的旧设备上运行。解决这个问题的一种方法是使用 Unity 5 的 AssetBundle 变体。（有关其他选项的详细信息，请参阅 Unity 的 Android 优化指南。）要使用 AssetBundle 变体，使用 ETC1 不能被干净压缩的所有纹理必须与纹理唯一的 AssetBundles 隔离。接下来，使用供应商特定的纹理压缩格式（如 DXT5，PVRTC 和 ATITC），创建足够的这些 AssetBundles 变体来支持 Android 生态系统的非 ETC2 功能片段。对于每个 AssetBundle 变体，将包含的纹理的 TextureImporter 设置更改为适用于 Variant 的压缩格式。

在运行时，可以使用 [SystemInfo.SupportsTextureFormat](#) API 检测不同纹理压缩格式的支持。该信息应用于选择并加载包含受支持格式压缩的纹理的 AssetBundle 变体。有关 Android 纹理压缩格式的更多信息，请点击[此处](#)。

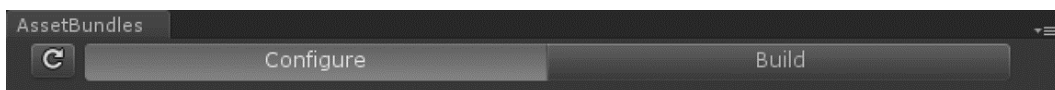
## iOS 文件句柄过度使用

以下部分中描述的问题已在 Unity 5.3.2p2 中修复。Unity 的当前版本不受此问题的影响。在 Unity 5.3.2p2 之前的版本中，Unity 将在 AssetBundle 加载的整个时间内为 AssetBundle 保留打开的文件句柄。这在大多数平台上不是问题。但是，iOS 限制进程可能同时打开 255 的文件句柄数。如果加载 AssetBundle 超出此限制，将导致加载失败，并显示 “Too Many Open File Handles” 错误。对于试图将其内容分成数百或数千个资产分类的项目，这是一个常见的问题。对于无法升级到 Unity 修补版本的项目，临时解决方案有：

- 通过合并相关的AssetBundles减少使用的AssetBundles数量
- 使用 AssetBundle.Unload (false) 关闭 AssetBundle 的文件句柄，并手动管理加载的对象的生命周期

## Unity 的 AssetBundle 浏览器工具

注意：此工具是 Unity 的标准功能的附加功能。要访问它，你必须从 [GitHub](#) 下载它，并从标准 Unity Editor 的下载和安装中单独安装。该工具使用户能够查看和编辑其 Unity 项目的 AssetBundle 的配置。它能防止创建无效软件包的编辑，并通知你现有软件包的任何问题。它还提供基本的构建功能。使用此工具替代选择资产并在检查器中手动设置其 AssetBundle。它可以放入 5.6 或更高版本的任何 Unity 项目中。它将在 **Window > AssetBundle Browser** 中创建一个新的菜单项。Bundle 配置和构建功能在新窗口中分为两个选项卡。



需要 Unity 版本在 5.6 以上

用法 —— 配置

注意：此实用程序处于预发布状态，因此，我们建议在使用项目之前创建备份。此窗口提供了一个资源管理器界面，用于管理和修改项目中的 AssetBundle。当第一次打开时，该工具将解析后台中的所有 Bundle 数据，缓慢地标记警告或其检测到的错误。它可以与项目保持同步，但不能总是意识到工具之外的活动。要强制快速通过错误检测，或使用外部更改更新工具，请点击左上角的刷新按钮。窗口分为四个部分：Bundle 列表，Bundle 详细信息，资产列表和资产详细信息。

