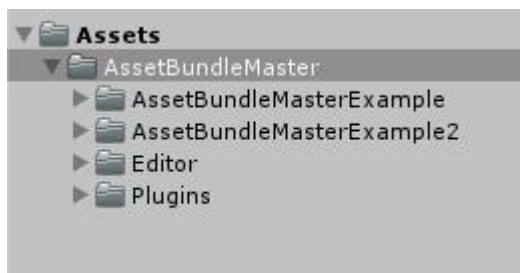


在开始使用之前, 建议先导入到一个空的工程里, 通过 ReadMe 的一步步引导使你对整个框架以及文件结构进行熟悉, 之后再考虑导入

到现有工程中使用, 完整看完教程大概需要 2 个小时左右. 先看看文件夹结构:



它在 AssetBundleMaster 文件夹下有几个子文件夹:

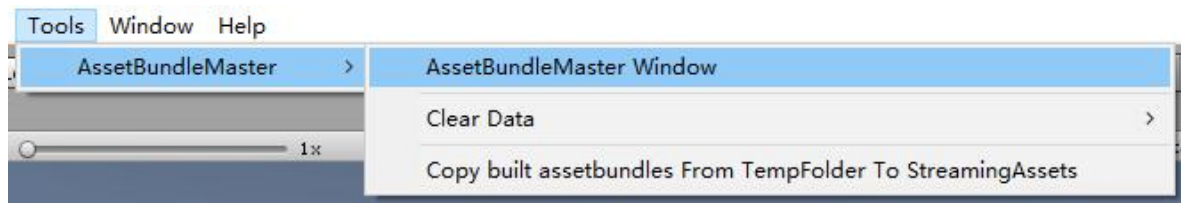
AssetBundleMasterExample 是基本的 API 使用 Demo, 包括资源加载, 场景加载, 以及相关的卸载操作.

AssetBundleMasterExample2 是针对 Built-In Shader 的多次编译以及其优化的测试场景.

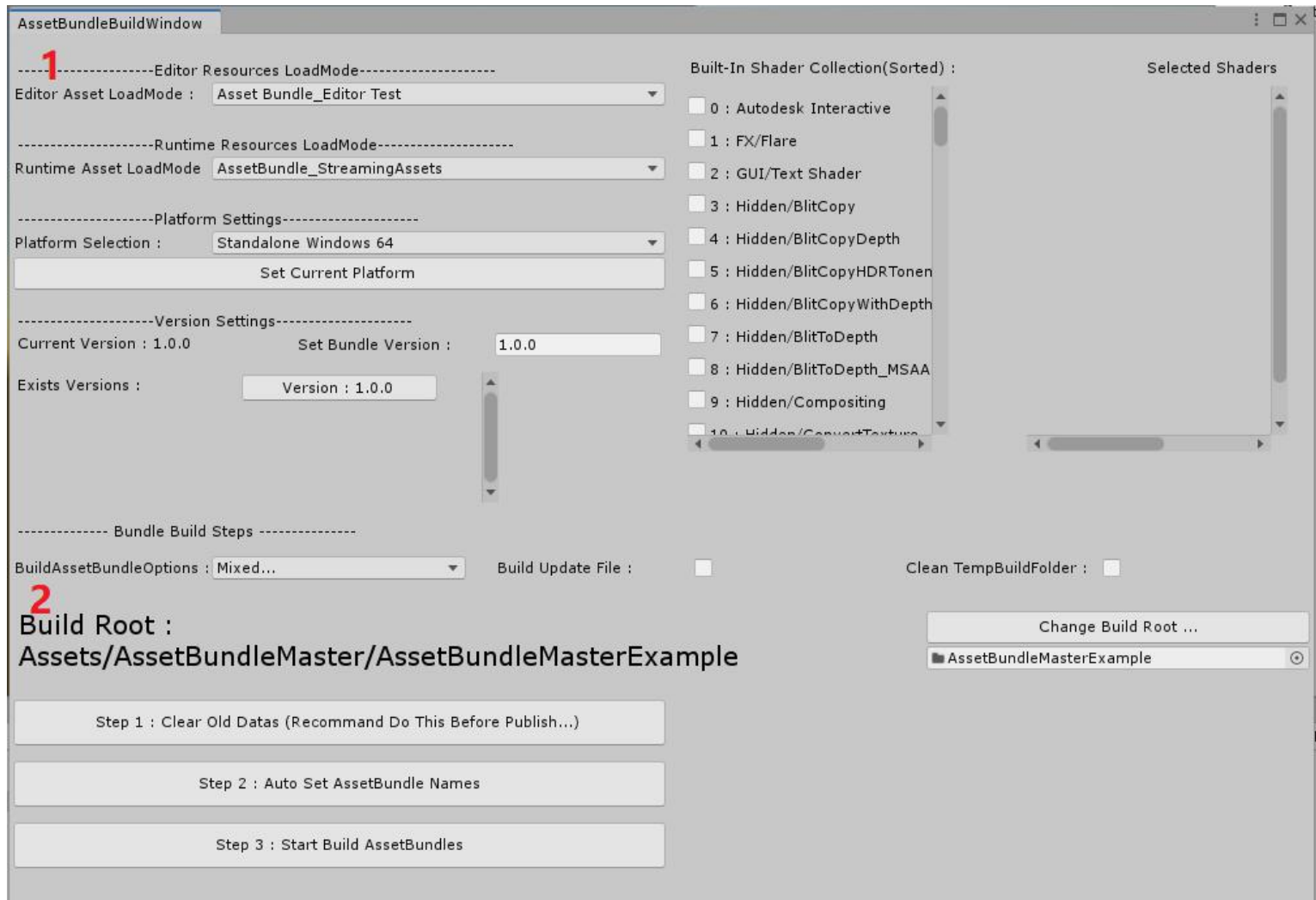
Editor 是编辑器脚本文件夹. 包含打包面板等.

Plugins 是运行时脚本文件夹. 包含所有核心代码. 因为资源加载是最基础的逻辑层, 所以放在 Plugins 里面.

让我们先在编辑器下运行基础 API 场景, 工具栏上打开 AssetBundleMaster 的编辑器面板:



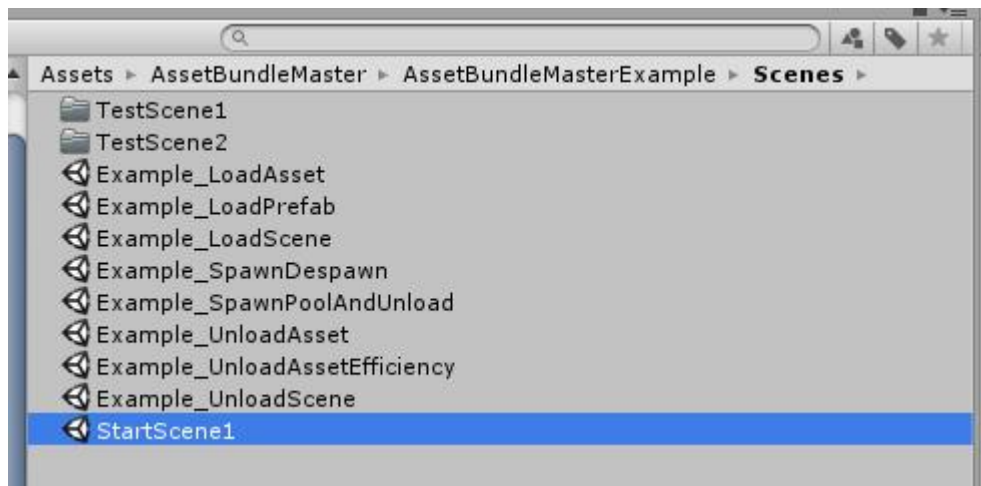
外观如图:

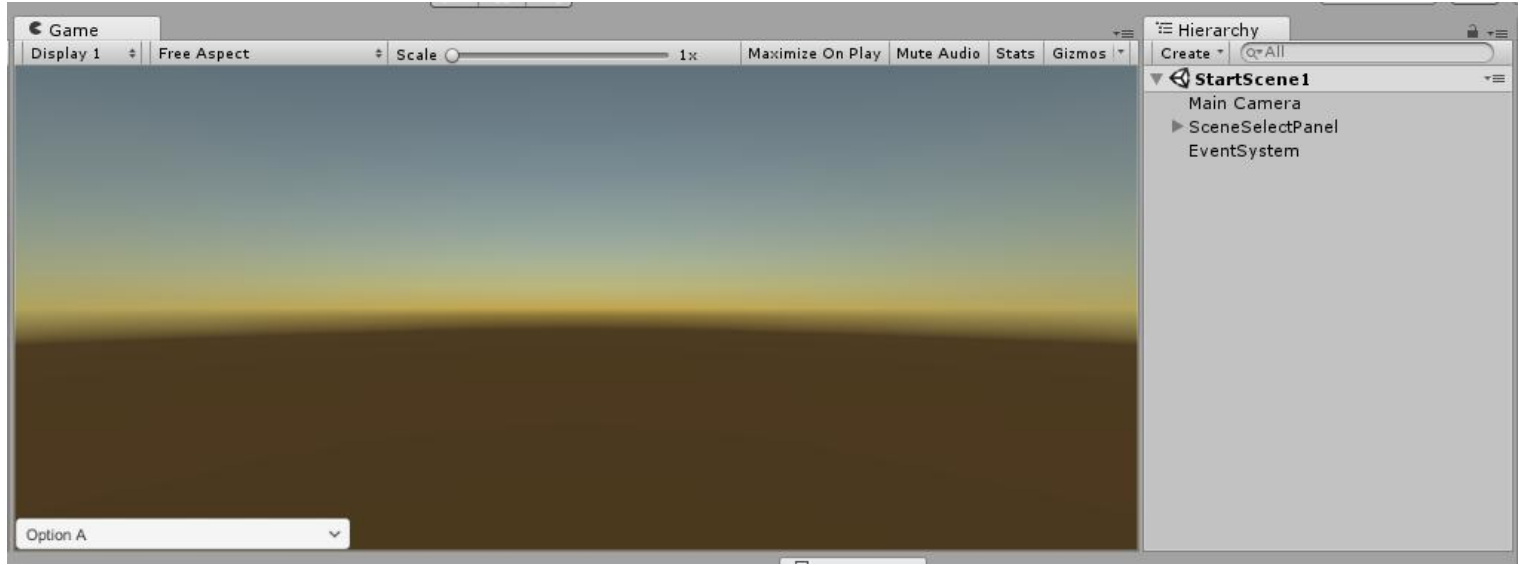


我们在编辑器运行, 只需要设定两个变量即可, 对开发者比较友好, 其它设定在打包的时候才需要了解, 我们在后面进行说明, 现在先忽略:

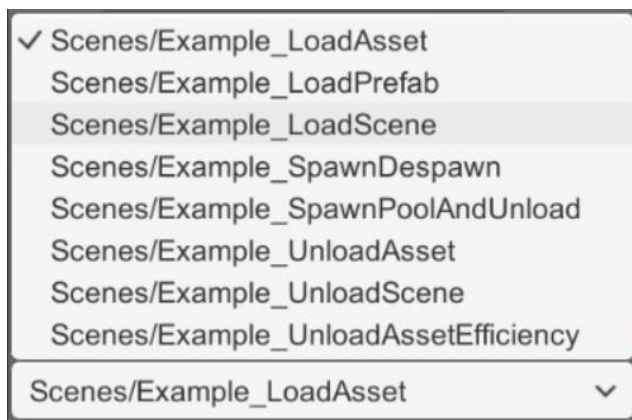
1. Editor Asset LoadMode : 编辑器下的资源读取方式. 先把它设为 AssetDataBase\_Editor, 直接读取资源.
2. Build Root : 它是资源读取的根文件夹, 按照图中设置即可进行 Demo 场景读取, 点击[Change Build Root...]按钮, 选择

AssetBundleMasterExample 文件夹即可. 接下来打开初始场景 StartScene1, 它在 AssetBundleMasterExample/Scenes 下面:



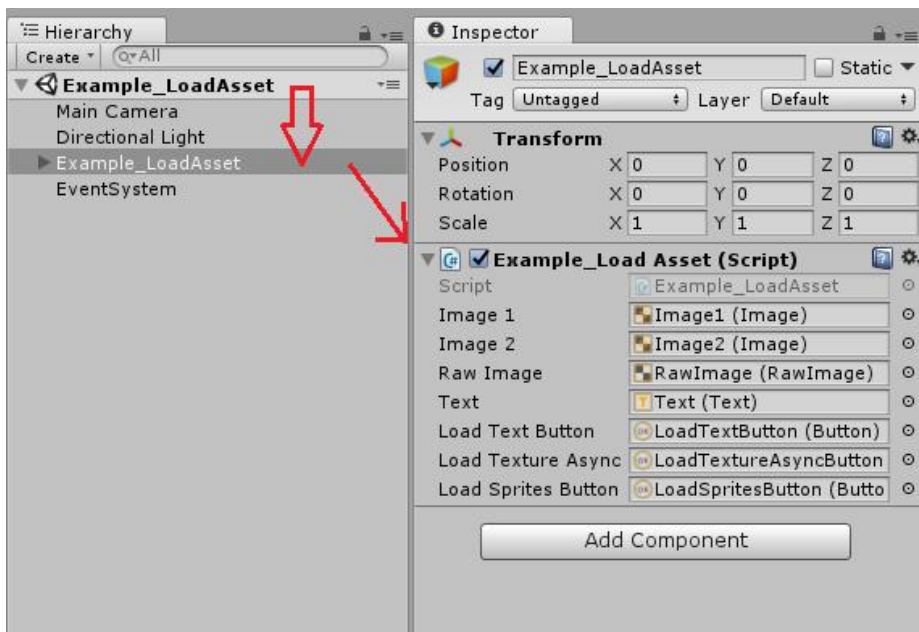


运行后左下角有场景选择：

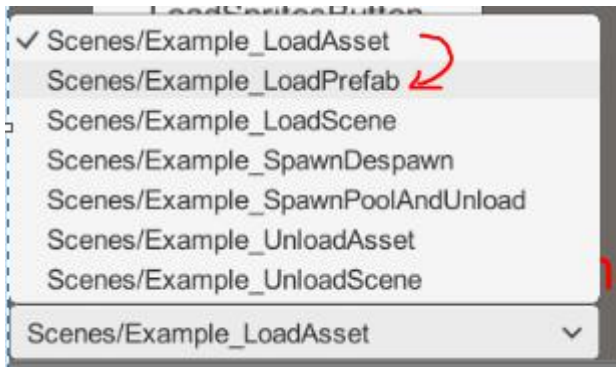


这些就是所有API的Demo场景了，选择任意一个场景可以看到测试代码与场景同名。找到场景下同名的GameObject，脚本就在上面，

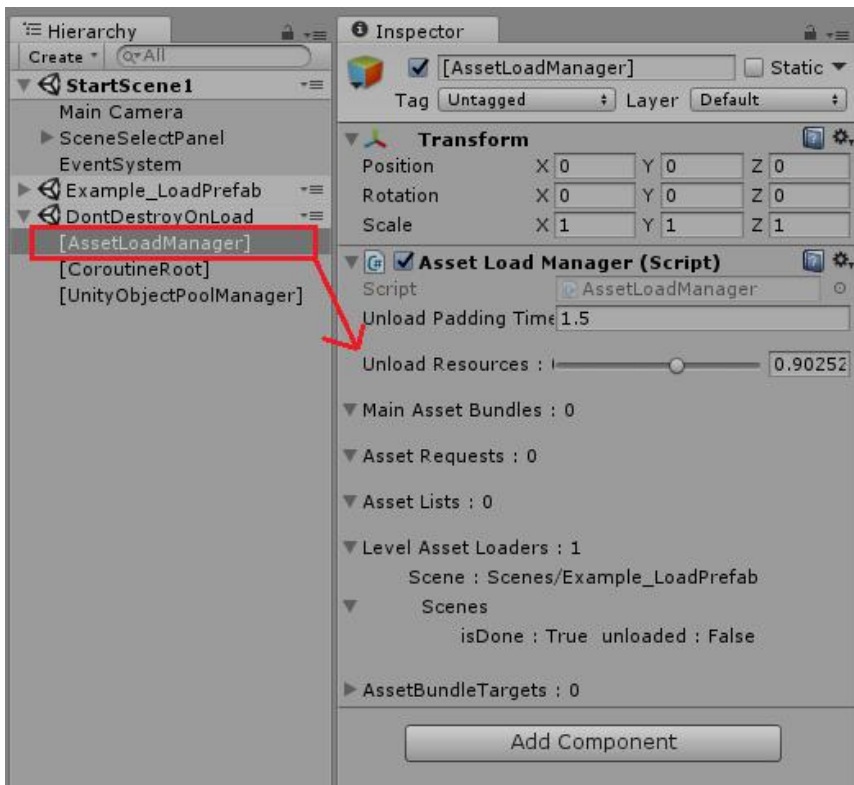
打开查看即可。



我们先说明一下场景加载的过程，在我们从下拉列表中选择某个场景之后，它会把你之前加载的所有东西都卸载掉，所以读取新场景之后每个场景中加载的资源都是全新的，方便测试。建议一直打开 Profiler 全程查看。卸载资源会有一个等待过程，这在你切换场景之后可以在 Hierarchy 的 [AssetLoadManager] 这个对象的检视面板中查看到，比如我们切换一下场景：



然后就可以在[AssetLoadManager]组件面板上看到一个倒计时 Slider，它表示卸载资源的等待时间：



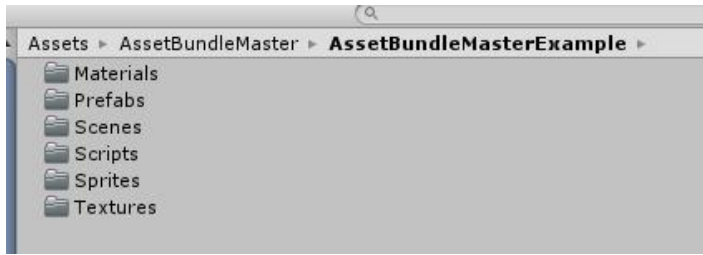
因此如果你使用 Profiler 的 Memory > Detailed > Take Sample 查看资源是否被正确卸载，在它执行之后查看。

下面来说明各个场景以及 API 的使用，我们刚才在编辑器面板设置了 Build Root 这个路径，所有资源的读取路径都是跟它的相对路径，这些资源被称为主要资源，是通过代码加载的资源，比如场景等，而放在这个文件夹之外的资源为引用资源，比如场景中引用到的 Mesh 等不会直接通过代码加载的资源，不需要将所有资源都放到 Build Root 文件夹下，这里只是放在一起方便用户删除 Demo 资源。之后不另外说明。

PS：在使用 AssetDataBase\_Editor 模式时，是没有异步加载的，所有异步加载都会变为同步加载，因为使用的加载方式为 AssetDataBase.LoadAssetAtPath 的编辑器加载方式。在资源的卸载方式上也不同，它是靠 Resources.UnloadUnusedAssets(); 的方式

卸载资源的, 而 AssetBundle 模式的话是通过 `AssetBundle.Unload();` 和 `Resources.UnloadUnusedAssets();` 的组合方式进行卸载的, 效率更高.

会被加载到的资源文件就在 `AssetBundleMasterExample` 文件夹下, 请自行查看.



这样编辑器下的运行设置就完成了, 对于开发人员来说并没有任何学习成本, 只需要修改 Build Root 为你们的工程的资源文件夹即可, 然后你可以看到编辑器下多出来几个文件夹和文件, 这些就是 AssetBundleMaster 的配置信息了, 只要上传到 SVN 等就可以同步设置给其他人了.



下面是 Demo 场景以及 API 的说明, 最终发布 AssetBundle 包的过程在文章末尾.

零. StartScene1 / StartScene2 (脚本 : StartScene.cs)

Demo 中有两个初始场景 StartScene1 和 StartScene2, 一般情况下你不需要对 AssetBundleMaster 进行初始化操作, 只有在加载远程 AssetBundle 的模式下需要等待初始化完成之后再进行资源读取, 因为需要预先下载一些基础文件, 我们可以先看看

StartScene.cs 脚本中的 Start() 函数 :

```

void Start()
{
#if DEVELOPMENT_BUILD
    Application.runInBackground = false;
#else
    Application.runInBackground = true;
#endif

    if(dropdown)
    {
        dropdown.ClearOptions();
        dropdown.AddOptions(loadScenePaths);

        AssetLoadManager.Instance.minUnloadAssetCounter = 0;    // Set min
        AssetLoadManager.Instance.unloadPaddingTime = 3f;    // Set unloa
        AssetUnloadManager.Instance.maxUnloadPerFrame = 1f;

        // This Is The Game Entry Point!!!
        // if remote-assetbundle mode, we must wait for the AssetLoadManag
        AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
        {
            // start your game logic after AssetLoadModuleInited
            dropdown.onValueChanged.AddListener(LoadLevel);
            LoadLevel(0);
        });
    }
}

```

API (命名空间 AssetBundleMaster.AssetLoad):

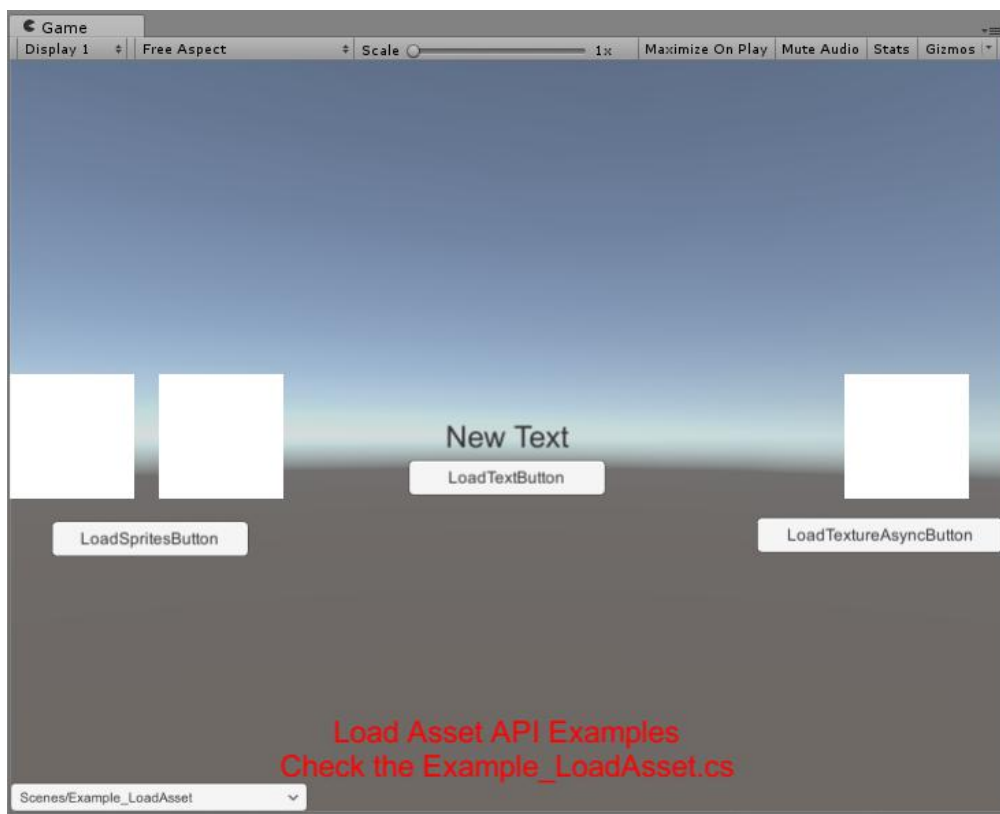
**AssetLoadManager.Instance.OnAssetLoadModuleInited(System.Action callback)**

把它当做逻辑的入口, 游戏逻辑在 AssetLoadManager.Instance.OnAssetLoadModuleInited() 的回调后进行. 虽然只有

AssetBundle\_Remote 模式下需要进行此等待, 不过其他模式下也可以使用, 不影响.

#### 一. Example\_LoadAsset (脚本 Example\_LoadAsset.cs)

此 Demo 主要演示怎样读取一般资源, 读取控制由 ResourceLoadManager 控制.



```

// Use this for initialization
void Start()
{
    LoadTextButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadTextButton Start Load at frame : " + Time.frameCount);

        // Load Single Asset Sync
        var text = ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1");
        if(text)
        {
            Text.text = text.text;
            Debug.Log(Text.text + " Loaded at frame : " + Time.frameCount);
        }
    });

    LoadTextureAsyncButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadTextureAsyncButton Start Load at frame : " + Time.frameCount);

        // Load Single Asset Async
        ResourceLoadManager.Instance.LoadAsync<Texture2D>("Textures/Pic3", (_tex) =>
        {
            RawImage.texture = _tex;
            Debug.Log(RawImage.texture + " Loaded at frame : " + Time.frameCount);
        });
    });

    LoadSpritesButton.onClick.AddListener(() =>
    {
        Debug.Log("LoadSpritesButton Start Load at frame : " + Time.frameCount);

        // load asset with ext name
        var sp1 = ResourceLoadManager.Instance.Load<Sprite>("Sprites/Pic1.tga");
        if(sp1)
        {
            Image1.overrideSprite = sp1;
            Debug.Log(Image1.overrideSprite + " Loaded at frame : " + Time.frameCount);
        }

        var sp2 = ResourceLoadManager.Instance.Load<Sprite>("Sprites/Pic1.png");
        if(sp2)
        {
            Image2.overrideSprite = sp2;
            Debug.Log(Image2.overrideSprite + " Loaded at frame : " + Time.frameCount);
        }
    });

    StartScene.RegisterClickFocus();
}

```

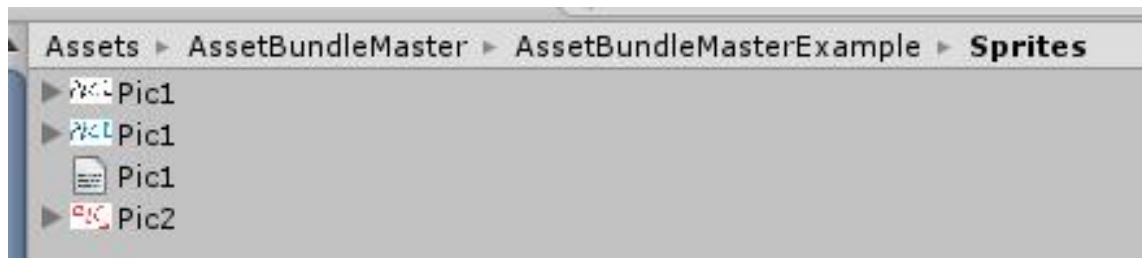
读取 API (命名空间 AssetBundleMaster.ResourceLoad):

```
ResourceLoadManager.Instance.Load<T>(string loadPath); // 同步读取
```

```
ResourceLoadManager.Instance.LoadAsync<T>(string loadPath, System.Action<T> loaded); // 异步读取
```

```
ResourceLoadManager.Instance.LoadAll<T>(string loadPath); // 同步读取多个文件(同名称)
```

它主要读取了 Sprites 和 Textures 文件夹下的文件, 使用方法很容易理解, "Sprites/Pic1" 这个路径下的资源比较特殊, 特别说明一下:



它有三个相同名称的文件 Pic1.tga / Pic1.png / Pic1.txt, 这是非常特殊的情况, 因为我们当前读取资源的路径是不带后缀名的(与 Resources.Load 等 API 统一), 所以这三个资源的读取路径是一样的, 而我们的泛型读取 API 可以通过类型限定来进行读取, 比如 ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1"); 它就读取了这个路径下的第一个 TextAsset 类型的资源, 而 ResourceLoadManager.Instance.LoadAll<Sprite>("Sprites/Pic1"); 则是返回了所有该路径下的 Sprite 资源数组, 如果你把 "Sprites/Pic1" 改为 "Sprites", 它指向了文件夹, 也是一样能读取出文件夹下的资源的, 逻辑上与 UnityEngine.Resources 提供的 API 相似.

所以通过不带后缀名的路径读取资源的话, 如果有同名资源, 想要读出指定类型的资源肯定要进行一次遍历读取, 这样就增加了无谓的资源读取开销. 而如果带后缀名的话, 在文件系统层面就保证了资源的唯一性(不能有重名文件), 因此我们强化了读取 API, 添加了带后缀名的读取路径的支持, 这样在像上面的 TextAsset 读取:

```
ResourceLoadManager.Instance.Load<TextAsset>("Sprites/Pic1").text;
```

就可以写成:



```
ResourceLoadManager.Instance.Load<TextureAsset>("Sprites/Pic1.txt").text;
```

这样就没有额外的资源遍历开销了, Demo 中的后两个 Sprite 读取就是通过带后缀读取的, 在实际使用中如果文件夹有同名资源并且只读取单个资源, 请使用带后缀名的路径, 以减少开销.

特别说明的是, AssetBundle 模式下, 在异步读资源时, 用户又同时请求了同步读取的话, 异步回调和同步回调都能正确触发, 只是会在编辑器 Console 面板中出现错误信息, 不过不影响程序的正确性, 请看下图:

```
void AsyncAndSync()
{
    Debug.Log("Start Loaded at frame : " + Time.frameCount);

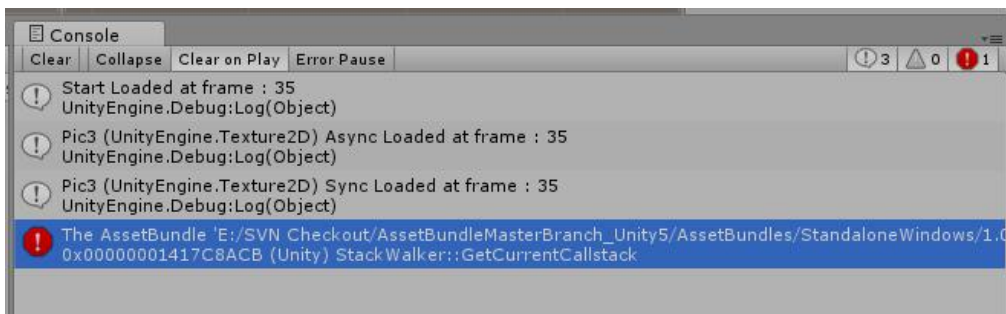
    ResourceLoadManager.Instance.LoadAsync<Texture2D>("Textures/Pic3", (_tex) =>
    {
        RawImage.texture = _tex;

        Debug.Log(RawImage.texture + " Async Loaded at frame : " + Time.frameCount);
    });

    var tex2D = ResourceLoadManager.Instance.Load<Texture2D>("Textures/Pic3");

    Debug.Log(tex2D + " Sync Loaded at frame : " + Time.frameCount);
}
```

代码在 Demo 中没有提供, 仅供参考.



这是因为 Unity 没有提供停止异步读取 AssetBundle 的 API, 并且在异步读取中进行同步读取就会报错, AssetBundleMaster 的 Low-Level API 对此进行了处理, 保证了回调的正确性, 用户不需要对此担心, 你可以在发布 AssetBundle 之后进行测试.

其余 API 在 ResourceLoadManager.cs 中可查, 看下图, 泛型和非泛型版本都有, 方便使用 Lua 脚本语言的用户 :

```

#region Main Funcs
/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public UnityEngine.Object Load(string loadPath, System.Type systemTypeInstance)...
/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public UnityEngine.Object Load(string loadPath)...
/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public T Load<T>(string loadPath) ...

/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
public UnityEngine.Object[] LoadAll(string loadPath, System.Type systemTypeInstance)...
/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
public UnityEngine.Object[] LoadAll(string loadPath)...
/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
public T[] LoadAll<T>(string loadPath) ...

/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public void LoadAsync(string loadPath, System.Type systemTypeInstance, System.Action<UnityEngine.Object> loaded)...
/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public void LoadAsync(string loadPath, System.Action<UnityEngine.Object> loaded)...
/// <summary> Load asset from a path. Notice : if in Resources mode, the load ...
public void LoadAsync<T>(string loadPath, System.Action<T> loaded) ...

/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
private void LoadAllAsync(string loadPath, System.Type systemTypeInstance, System.Action<UnityEngine.Object[]> loaded)...
/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
public void LoadAllAsync(string loadPath, System.Action<UnityEngine.Object[]> loaded)...
/// <summary> Load all assets from a path. Notice : if pass a directory path, ...
public void LoadAllAsync<T>(string loadPath, System.Action<T[]> loaded) ...

/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync(string loadPath, System.Type systemTypeInstance)...
/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync(string loadPath)...
/// <summary> This logic is just for compatible with Resources.LoadAsync(...), r ...
public ResourceLoadManager.ResourceRequest LoadAsync<T>(string loadPath) ...

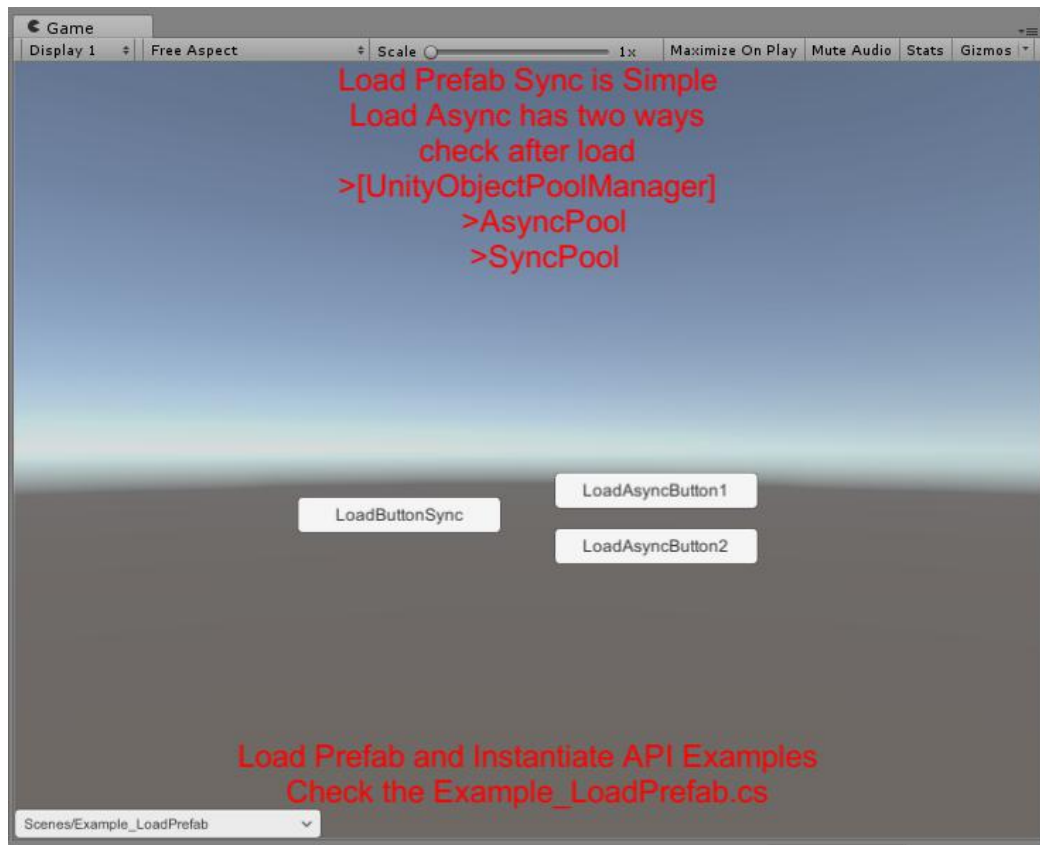
/// <summary> Unload Asset with target type and load path, Recommend use this AP ...
public void UnloadAsset(string loadPath, System.Type systemTypeInstance, bool inherit = false)...
/// <summary> Unload Asset with target type and load path, generic version
public void UnloadAsset<T>(string loadPath, bool inherit = false) ...
/// <summary> Unload All Assets
public void UnloadAllAssets()...
#endregion

```

提示：在此版本中添加了代码生成工具，一般的资源/GameObject/场景的加载和卸载逻辑代码都能通过工具生成，详细请看附录。

## 二. Example\_LoadPrefab (脚本 Example\_LoadPrefab.cs)

此 Demo 演示怎样读取和实例化 GameObject 对象，使用的是对象池。由 PrefabLoadManager 进行控制。实例化对象有同步和异步读取模式，在异步读取逻辑中可以有多种方法，会对性能有些微影响，请看下图：



```
void Start()
{
    LoadButtonSync.onClick.AddListener(() =>
    {
        for(int i = 0; i < spawnCount; i++)
        {
            var cube = PrefabLoadManager.Instance.Spawn(LoadPath, "SyncPool"); // spawn Sync
            RandomCubePos(cube);
        }
    });

    /*
    * These are two kinds of Async spawn way, The first (LoadAsyncButton1) is take a little overhead GC,
    * otherwise 2nd (LoadAsyncButton2) is less GC, see what is diferent
    */
    LoadAsyncButton1.onClick.AddListener(() =>
    {
        Debug.Log("Start Load at frame : " + Time.frameCount);
        for(int i = 0; i < spawnCount; i++)
        {
            // if Asset not yet loaded, this will call multi times the asset load Low-Level API
            PrefabLoadManager.Instance.SpawnAsync(LoadPath, (_cube) =>
            {
                RandomCubePos(_cube);
                Debug.Log("Loaded at frame : " + Time.frameCount);

            }, "AsyncPool");
        }
    });

    LoadAsyncButton2.onClick.AddListener(() =>
    {
        Debug.Log("Start Load at frame : " + Time.frameCount);
        // only call Low-Level API Once, wait until asset loaded
        PrefabLoadManager.Instance.LoadAssetToPoolAsync(LoadPath, (_prefab, _pool) =>
        {
            for(int i = 0; i < spawnCount; i++)
            {
                var cube = _pool.Spawn(LoadPath, _prefab); // you got the pool holds the Prefab
                RandomCubePos(cube);
                Debug.Log("Loaded at frame : " + Time.frameCount);
            }

        }, "AsyncPool");
    });
}
```

读取 API (命名空间 `AssetBundleMaster.ResourceLoad`):

```
PrefabLoadManager.Instance.Spawn(string loadPath, string poolName = null, bool active = true) // Spawn 是直接实例化
```

`GameObject`.

```
PrefabLoadManager.Instance.SpawnAsync(string loadPath, System.Action<GameObject> loaded = null, string poolName = null, bool active = true) // SpawnAsync 是异步读取资源后实例化 GameObject.
```

```
PrefabLoadManager.Instance.LoadAssetToPoolAsync(string loadPath, System.Action<GameObject, GameObjectPool> loaded = null, string poolName = null) // LoadAssetToPoolAsync 是将资源读取到对应的对象池.
```

上图中第二种异步读取方法只有在资源还未读取时, 相对第一种异步方法有微小性能优势, 不过使用起来比较麻烦, 推荐直接使用 `PrefabLoadManager.Instance.SpawnAsync` 减少代码复杂度. 其它接口请查看 `PrefabLoadManager.cs`, 我们强烈建议所有需要实例化的 `GameObject` 通过 `PrefabLoadManager` 这个类进行加载以及实例化(角色, UI, 特效, 武器等等, 所有 Prefab 皆可), 这样用户不仅获得了对象池的控制, 也获得了资源的自动控制. 资源的自动控制逻辑请参看附录.

你也可以调用 `LoadAssetToPoolAsync` 方法来进行预加载如下 :

```
PrefabLoadManager.Instance.LoadAssetToPoolAsync("Prefabs/Cube", null, "Characters");
```

这个方法把 Cube 加载到 "Character" 对象池中, 我们知道在加载资源时会产生 I/O, 解压缩, 反序列化, Shader Parse 等行为, 在系统空闲时进行预加载可以提高游戏体验.

### 三. Example\_LoadScene (脚本 `Example_LoadScene.cs`)

此 Demo 演示怎样读取场景以及读取场景的一些特点, 跟资源一样直接使用相对路径加载.



```
public const string SceneLoadPath1 = "Scenes/TestScene1/TestScene";  
public const string SceneLoadPath2 = "Scenes/TestScene2/TestScene";  
  
private List<int> m_loadedScenes = new List<int>();  
  
void LoadOneScene(string loadPath, AssetLoad.LoadThreadMode mode)  
{  
    Debug.Log("Start Load scene[" + loadPath + "] at frame : " + Time.frameCount);  
  
    var id = SceneLoadManager.Instance.LoadScene(loadPath,  
        mode,  
        UnityEngine.SceneManagement.LoadSceneMode.Additive,  
        (_id, _scene) =>  
        {  
            Debug.Log("Scene [" + _scene.path + "] Loaded at frame : " + Time.frameCount);  
        });  
    m_loadedScenes.Add(id);  
}
```

```
LoadSceneSyncButton1.onClick.AddListener(() =>  
{  
    LoadOneScene(SceneLoadPath1, AssetLoad.LoadThreadMode.Synchronous);  
});  
LoadSceneAsyncButton2.onClick.AddListener(() =>  
{  
    LoadOneScene(SceneLoadPath2, AssetLoad.LoadThreadMode.Asynchronous);  
});
```

读取场景使用的 API (命名空间 AssetBundleMaster.ResourceLoad):

```
SceneManager.LoadScene(sceneLoadPath,
```

```
AssetBundleMaster.AssetLoad.LoadThreadMode loadMode =
```

```
AssetBundleMaster.AssetLoad.LoadThreadMode.Asynchronous,
```

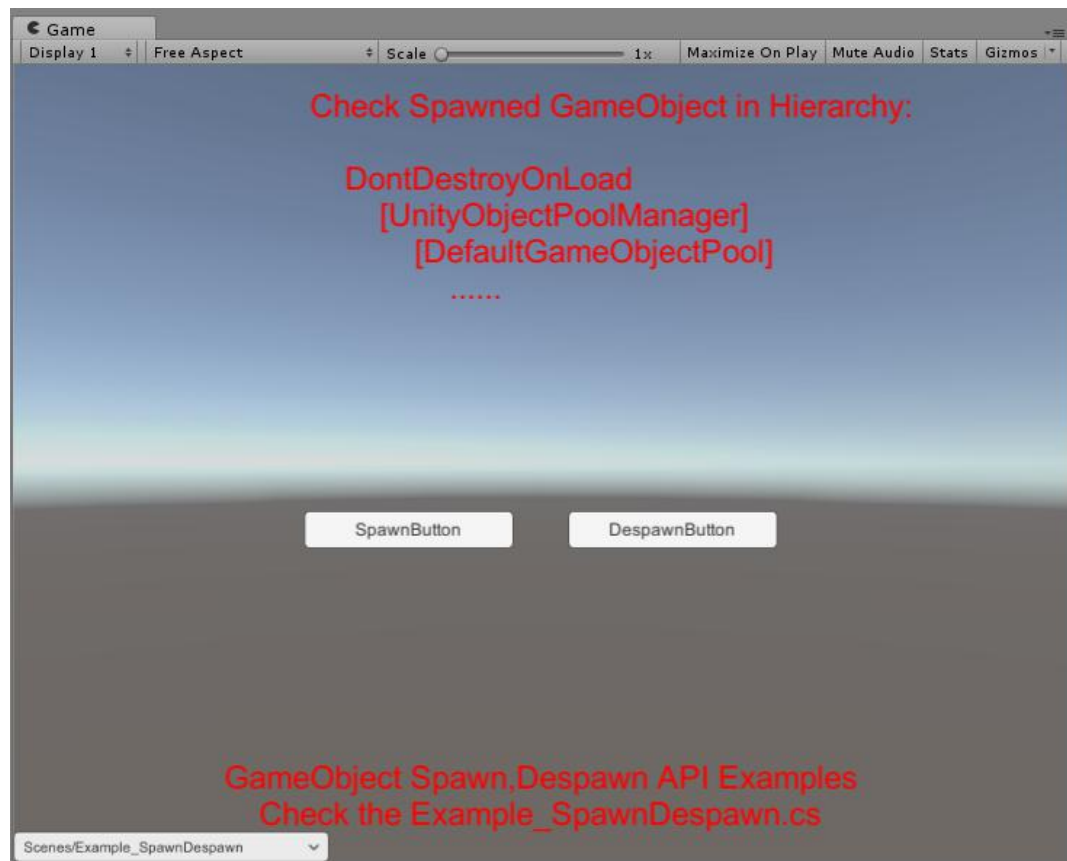
```
UnityEngine.SceneManagement.LoadSceneMode loadSceneMode = LoadSceneMode.Single,
```

```
System.Action<int, Scene> loaded = null )
```

它与资源一样通过相对路径来进行读取, 可以选择同步异步以及加载模式, Unity 在读取场景时无论是同步或是异步, 都需要等待场景读取完成的回调 `UnityEngine.SceneManagement.SceneManager.sceneLoaded` 的触发, 所以无论同步异步, 都建议用户通过 `loaded` 回调进行下一步操作. 此函数的返回是一个哈希码, 它代表的就是此场景的 ID, 用户可以通过这个 ID 对场景进行控制, 详见 Demo 七 `Example_UnloadScene`.

#### 四. Example\_SpawnDespawn (脚本 `Example_SpawnDespawn.cs`)

此 Demo 演示受对象池控制的 `GameObject` 的实例化和归还到对象池的操作.



主要 API (命名空间 `AssetBundleMaster.ResourceLoad`):

```
PrefabLoadManager.Instance.Spawn(string loadPath, string poolName = null, bool active = true);
```

```
PrefabLoadManager.Instance.Despawn(GameObject go, string poolName = null);
```

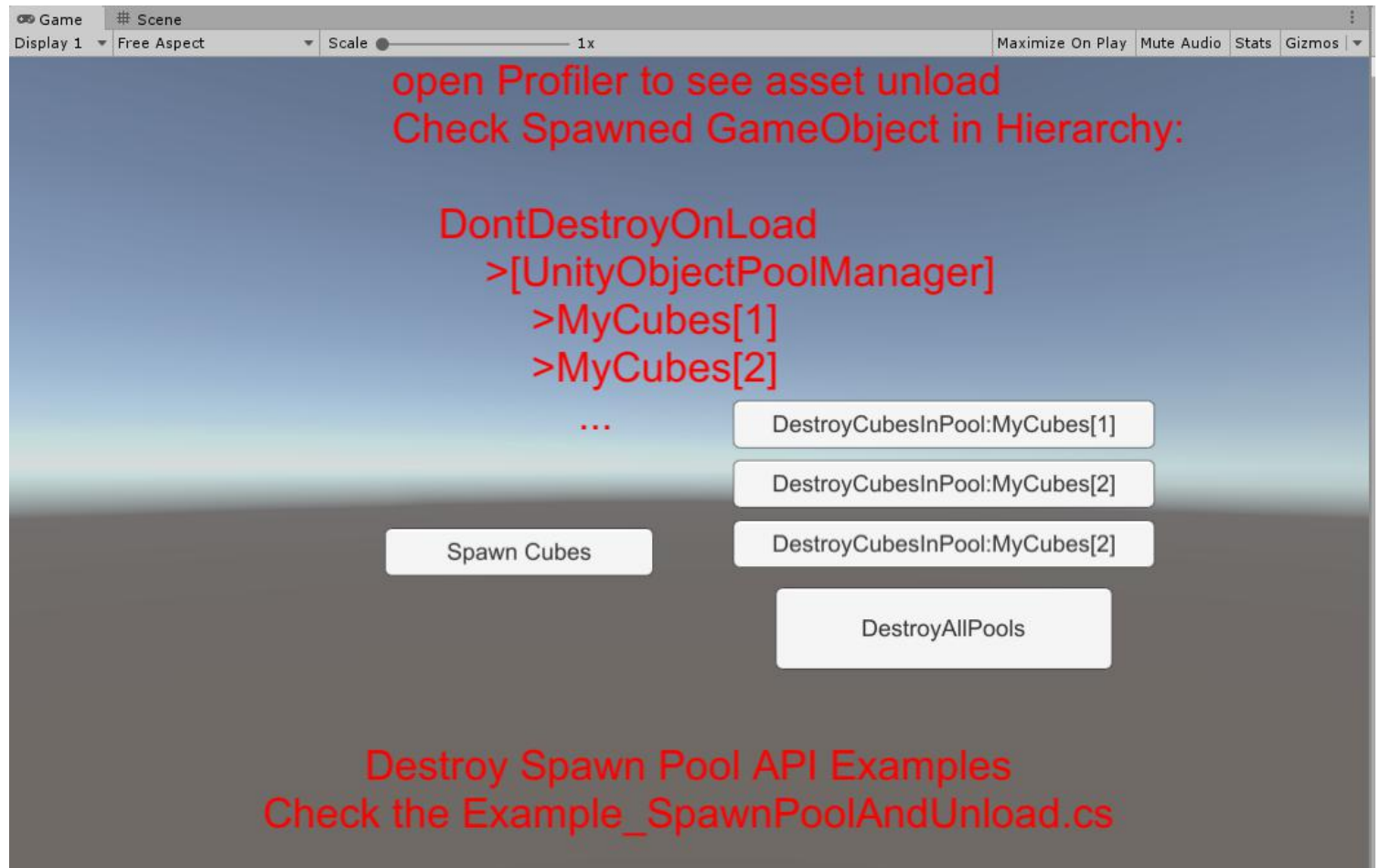
我们自动加载并实例化 Prefab 的时候可以设定对象池的名称 poolName(如果不填将会使用一个默认名称), 并且指定对象的 active, 在我们归还到对象池的时候, 也需要归还到对应名称的池中去, 遵循从哪个池中来, 回到哪个池中去的原则. 归还到池中的 GameObject, active 会被自动设置为 false, 并且 parent 节点会被修改为相应的池, 如果用户有特殊的需求, 可以自行修改相关代码 :

AssetBundleMaster.ObjectPool.GameObjectPool.Despawn 函数.

PS : 当 poolName 不正确的时候, 它会自动查询所有的 pool, 直到找到正确的 pool 然后归还对象, 如果对象池数量非常多的时候, 用户要注意 poolName 的正确性, 以免自动查询造成性能问题.

## 五. Example\_SpawnPoolAndUnload (脚本 Example\_SpawnPoolAndUnload.cs)

此 Demo 演示了怎样销毁对象池内的资源, 以及自动资源控制是怎样工作的.



```

public string PoolName1 = "MyCubes[1]";
public string PoolName2 = "MyCubes[2]";
public string PoolName3 = "MyCubes[3]";

public const string LoadPath = "Prefabs/Cube";

private List<GameObject> m_caches2 = new List<GameObject>();
private List<GameObject> m_caches3 = new List<GameObject>();

// Use this for initialization
void Start()
{
    SpawnButton.onClick.AddListener(() =>
    {
        CreateCube(LoadPath, PoolName1, 12);
        CreateCube(LoadPath, PoolName2, 15, m_caches2);
        CreateCube(LoadPath, PoolName3, 10, m_caches3);
    });

    // this will not unload asset
    DestroyPoolButton1.onClick.AddListener(() =>
    {
        PrefabLoadManager.Instance.DestroyTargetInPool(LoadPath, PoolName1, true);
    });
    DestroyPoolButton2.onClick.AddListener(() =>
    {
        foreach(var go in m_caches2)
        {
            PrefabLoadManager.Instance.DestroySpawned(LoadPath, go, PoolName2, true);
        }
        m_caches2.Clear();
    });
    DestroyPoolButton3.onClick.AddListener(() =>
    {
        foreach(var go in m_caches3)
        {
            PrefabLoadManager.Instance.DestroySpawned(go, PoolName3, true);
        }
        m_caches3.Clear();
    });

    // this will unload the asset
    DestroyPoolAndUnloadButton.onClick.AddListener(() =>
    {
        PrefabLoadManager.Instance.DestroyAllPools(true);
        StartScene.FocusAssetLoadManager();
    });
}

```

主要 API (命名空间 AssetBundleMaster.ResourceLoad):

`PrefabLoadManager.Instance.DestroyTargetInPool(string loadPath, string poolName = null, bool tryUnloadAsset = true);`

`PrefabLoadManager.Instance.DestroySpawned(GameObject target, string poolName = null, string loadPath = null, bool tryUnloadAsset = true);`

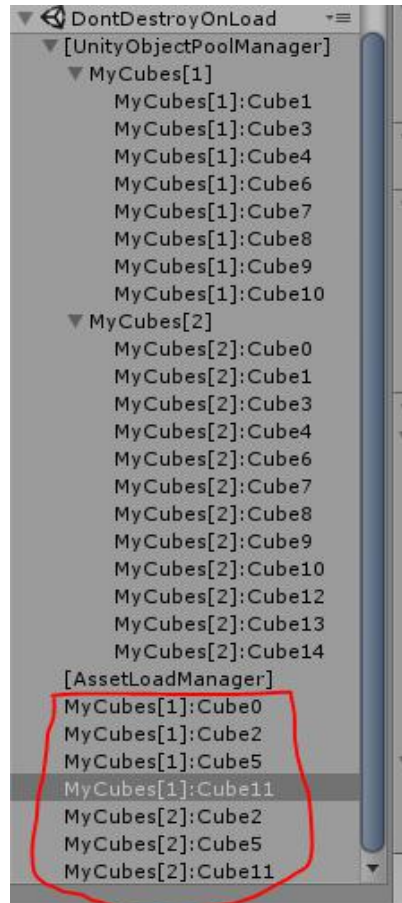
第一个 API 提供的是清除某个对象池中的某种资源, 通过读取路径作为标记, 适用于开发者知道何时进行资源卸载的情形( 销毁所有该对象池内通过 loadPath 实例化出来的对象 ).

第二个 API 提供删除单个实例 GameObject 的方法, 资源卸载控制由自动控制进行. 比如实例化( Spawn )出多个法术特效, 当法术结束时销毁该对象(DestroySpawned).

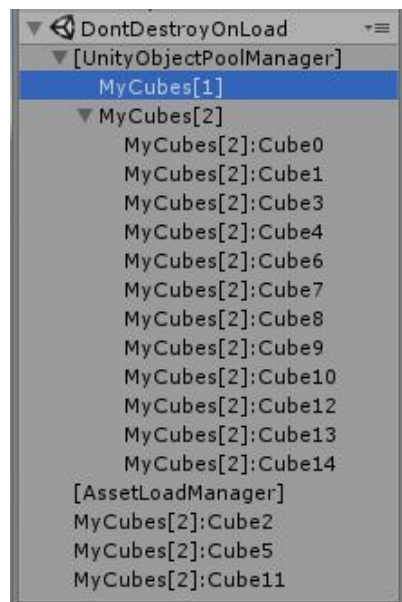


两个 API 都一样, 如果 `tryUnloadAsset` 为 `true`, 那么 `PrefabLoadManager` 会对引用资源进行检查, 如果在所有对象池中都没有继续使用的话, 会进行资源清除.

操作: 点击 `Spawn Cubes` 按钮之后, 场景中实例化了三个对象池, 并且每个对象池创建了一些 `Cube`, 看 `Hierarchy` 面板中显示的, 有些 `Cube` 的主节点被故意修改了, 这里是为了说明对象池与实例化对象的关联性与节点无关, 即使对象不在对象池节点下, 仍然是被对象池控制的( 比如武器和特效会挂载到人物节点之下, 仍然受到对象池控制 ):



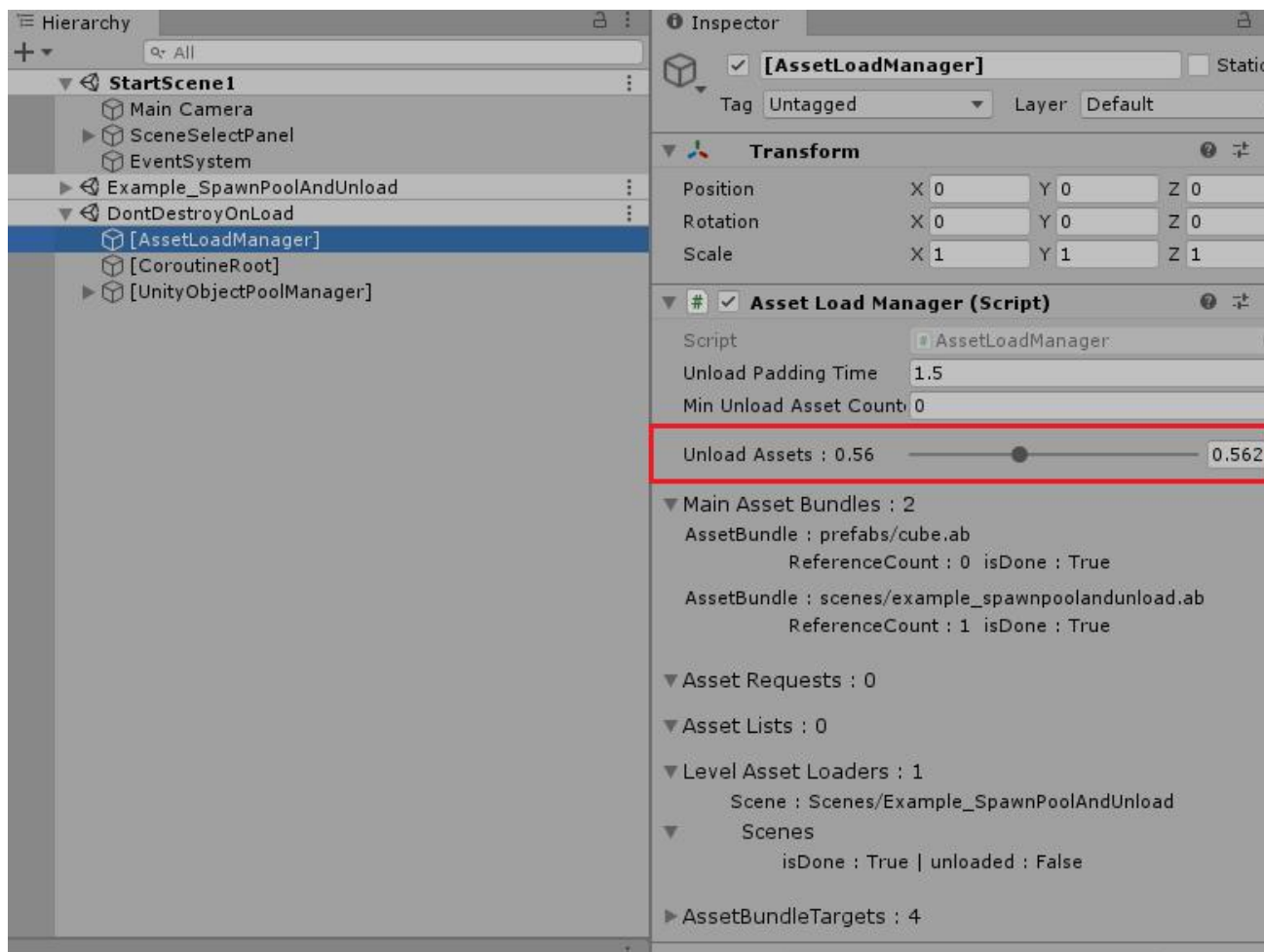
点击 `DestroyCubesInPool:MyCubes[1]` 按钮, 此时将 `MyCubes[1]` 对象池中的 `Cubes` 删除了, 可以看到对象池还在, 可是实例化的对象被删除了.



因为 Cube 的资源仍然被 MyCubes[2] 和 MyCubes[3] 对象池引用, 此时不触发任何删除资源的操作, 再点击

DestroyCubesInPool:MyCubes[2]和 DestroyCubesInPool:MyCubes[3]按钮, 此时所有实例化对象都被销毁了, 查看

AssetLoadManager 的检视面板, 可以看到资源删除倒计时:



所以 PrefabLoadManager 对 GameObject 的资源能够实现自动清除控制. 在团队开发的时候只需要使用正确的对象池名称进行控制

即可, 资源是否需要卸载的控制交给自动控制逻辑决定.

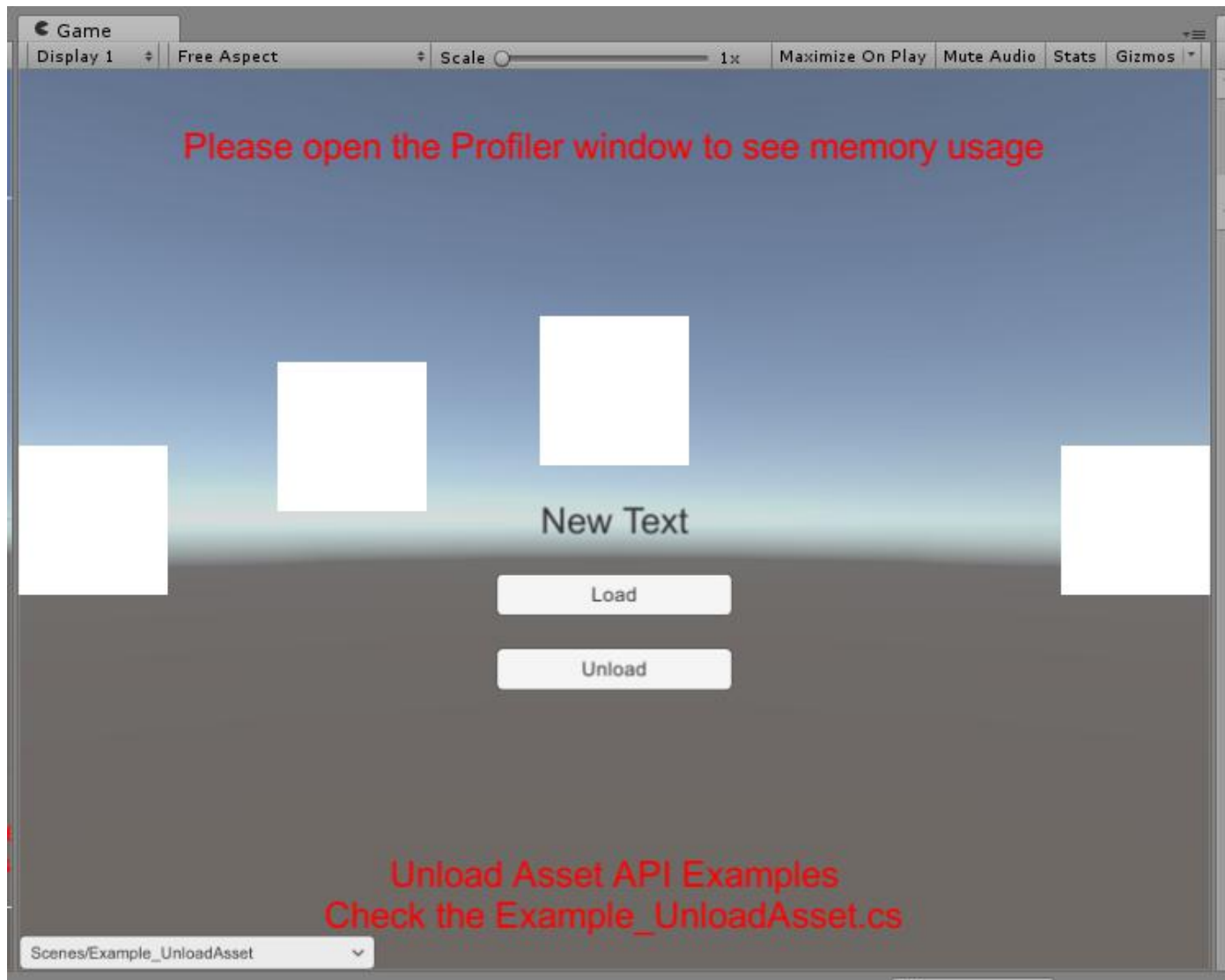
PS : 当前仍然使用的是 AssetDataBase\_Editor 模式, 资源卸载仍然使用的是 Resources.UnloadUnusedAssets(); 如果在

AssetBundle 模式下, 由于 GameObject 以及场景的资源是完全自动控制模式, 所以卸载资源使用的是 AssetBundle.Unload(true); 是更

精准和快速的资源卸载方式( 也有其它情况, 请参看附录 ).

## 六. Example\_UnloadAsset (脚本 Example\_UnloadAsset.cs)

此 Demo 演示一般资源的卸载过程. 加载过程与 Example\_LoadAsset 没有太大差别, 在这里不再进行描述, 下图为卸载资源的 API:



```

UnloadButton.onClick.AddListener(() =>
{
    // Pic1 loaded Sprite and Text, unload type <Object> means all these types
    /* Notice : Here the TextAsset is also unloaded */
    ResourceLoadManager.Instance.UnloadAsset<Object>("Sprites/Pic1", true); // true means <T> is base type

    // Pic2 is loaded as Sprite only
    ResourceLoadManager.Instance.UnloadAsset<Sprite>("Sprites/Pic2.png", false); // false means unload only type == <T>

    // Pic3 is loaded as Texture only, we can call unload like Resources.UnloadAsset
    ResourceLoadManager.Instance.UnloadAsset(RawImage.texture);

    // unload request is tick later, so you can clear reference after call UnloadAsset
    // Notick : unload request is not a force unload you should set reference to null
    Image1.overrideSprite = null;
    Image1_2.overrideSprite = null;
    Image2.overrideSprite = null;
    RawImage.texture = null;
});

```

主要 API (命名空间 AssetBundleMaster.ResourceLoad):

`ResourceLoadManager.Instance.UnloadAsset(UnityEngine.Object target);` // 不推荐使用

`ResourceLoadManager.Instance.UnloadAsset(string loadPath, System.Type systemTypeInstance, bool inherit = false);` //

推荐使用

`ResourceLoadManager.Instance.UnloadAsset<T>(string loadPath, bool inherit = false);` // 推荐使用

我们有几种卸载资源的方式, 你可以使用 `ResourceLoadManager.Instance.UnloadAsset(UnityEngine.Object target)`, 它就像 `UnityEngine.Resources.UnloadAsset` 的卸载方式, 比较容易理解. 或者使用 路径+类型 的方式, 这是因为同一个资源可能通过不同的类型加载出来, 比如说一张贴图可以加载为 `Texture2D` 或是 `Object`, 所以卸载逻辑也做了细分. 如上图所示, 卸载类型也是可以选择是否继承的, `ResourceLoadManager.Instance.UnloadAsset<Object>("Sprites/Pic1", true);` 的意思是卸载所有继承于 `Object` 并从 "Sprites/Pic1" 加载来的资源, 然后 `ResourceLoadManager.Instance.UnloadAsset<Sprite>("Sprites/Pic2", false);` 的意思是只卸载从 "Sprites/Pic2" 路径读取来的 `Sprite` 类型资源. 当然即使通过不同类型进行加载, 在内存上它们基本仍然是同一份, 并不会造成错误卸载的情况. 在这个框架下更推荐使用后两种 读取路径+类型 的方式, 因为你的资源对象并不一定是加载出来的资源( 比如你要使用的是 Prefab 上引用的另一张图片, 你使用 `UnloadAsset(xxx)` 传入了图片对象, 它在资源加载中并不存在 ), 而路径能很明显指代你加载的资源对象.

此 API 的主要逻辑就是通过资源的读取路径以及读取类型来唯一确定需要卸载的资源对象, 然后在底层的逻辑中减少对该资源的引用计数, 如果资源的引用计数为零的话, 就会触发资源卸载的过程, 在上图中我们将引用到图片的对象都置空了, 这样在后面的资源卸载中就能正确被卸载了. 而如果用户并没有将引用资源的对象置空, 卸载过程也会正常运行, 只不过运行之后也不会将资源卸载, 因为通过 `ResourceLoadManager` 加载的资源, 卸载是基于 `Resources.UnloadUnusedAssets();` 调用的, 有强引用的资源将不会被卸载. 而在中间层 `ResourceLoadManager` 中, 该资源是被弱引用对象引用的, 也就是说在这种情况下弱引用对象也是能一直存在的, 那么在下一次用户请求加载这些没有被释放的资源的时候, 返回的就是这个资源, 而不会触发资源加载的逻辑, 这就是保证资源在内存中不会重复的核心逻辑. 因此开发团队的成员都不需要对资源的控制进行额外的工作了, 只要在需要时加载, 不用时卸载即可, 因为资源不会因为错误的请求而在内存中重复, 所有人对资源的交叉控制都是非耦合的 ( 仍然有例外的情况, 请参看附录 ).

PS: 注意资源卸载过程在 `AssetDataBase_Editor` 和 `AssetBundle` 模式下是不一样的, `AssetDataBase_Editor` 是通过 `Resources.UnloadUnusedAssets();` 实现的, 而 `AssetBundle` 模式下是通过自动释放逻辑决定使用 `AssetBundle.Unload(true);` 或是 `Resources.UnloadUnusedAssets();` 的组合来释放资源的, 在 `AssetBundle` 模式下更高效且是免维护的.

每当你通过 `ResourceLoadManager` 释放一个资源, 会给 `AssetLoadManager` 的计数器增加一个计数, 如果大于 `AssetLoadManager.Instance.minUnloadAssetCounter` 的话就会触发 `Resources.UnloadUnusedAssets();` 操作并重置计数, 你可以修改这个数值以符合你的工程要求. 用户自行可修改的变量也只有两个, 很简单 :

**1. `AssetLoadManager.Instance.minUnloadAssetCounter`: 请求卸载至少多少个资源后触发卸载逻辑, 根据需求进行调整.**

2. `AssetBundleMaster.AssetLoad.AssetLoadManager.Instance.unloadPaddingTime` : 触发卸载逻辑到执行卸载的等待时间,

为了减少资源的重复加载, 我们给卸载过程添加的等待时间就是这个变量, 用户可以自行调整. 实际项目运行时因为经常会触发卸载逻辑, 你不需要过于关注它, 根据需求进行调整.

PS: 这两个数值都在 `StartScene.cs` 中被设为很小的数值, 是为了方便编辑器下 Profiler 测试.

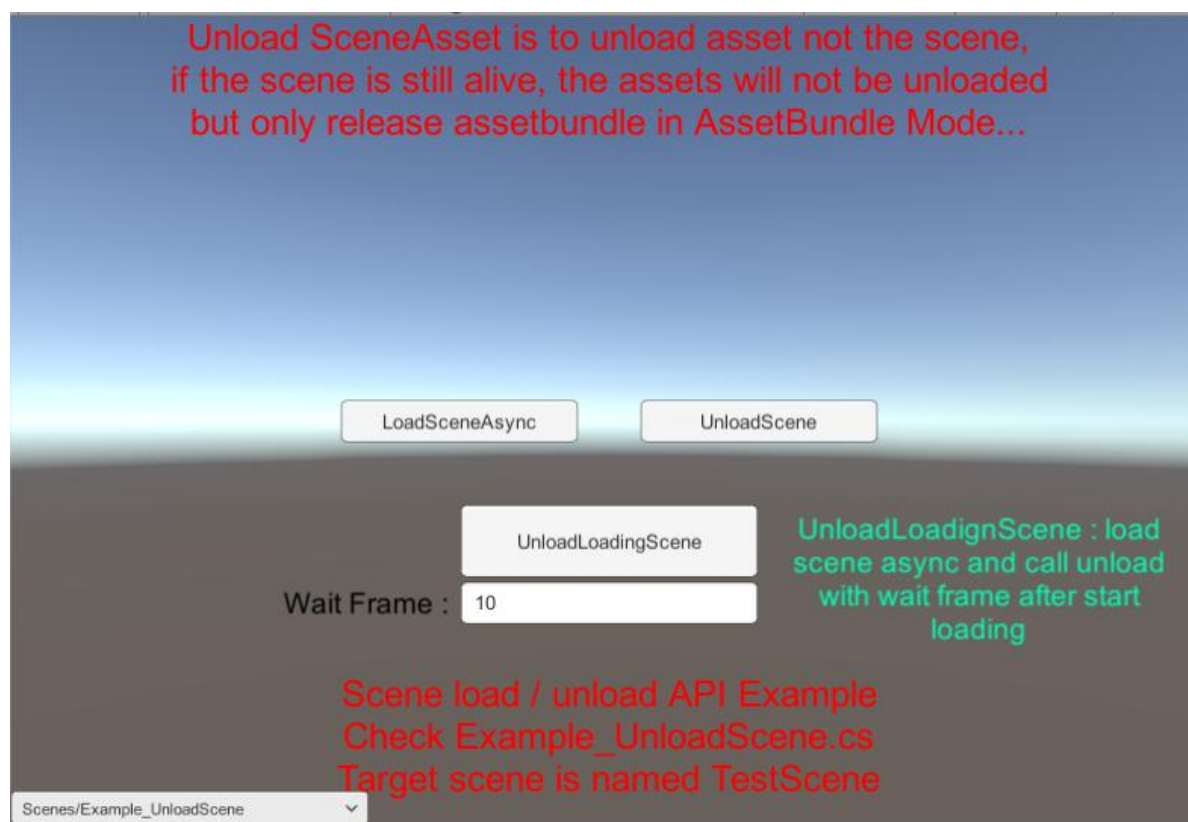
3. 在 `StartScene.cs::Start()` 里面可以看到第三个变量 --> `AssetUnloadManager.Instance.maxUnloadPerFrame = 1f`; 它是

卸载 `AssetBundle` 的管理类, 每帧都会检测是否有需要卸载的 `AssetBundle`, 而 `maxUnloadPerFrame` 就是每帧的最大卸载数量, 区间为 `[0.01f, 1000f]`, 它是一个浮点数, 可以小于 1, 如果小于 1 就是说每隔  $1 / \text{maxUnloadPerFrame}$  帧进行 1 个 `AssetBundle` 的卸载, 比如 0.5 的话就是每两帧卸载一个, 0.2 的话就是每 5 帧卸载一个. 这是因为在不同的平台下对于 `AssetBundle` 的卸载耗费的资源是不同的, 比如在编辑器模式下同时卸载 87 个 `AssetBundle` 用了 200ms, 造成了非常明显的卡顿, 而在 Build 后的 exe 中运行, 卸载 87 个 `AssetBundle` 只用了 13ms, 也造成了帧数降低, 所以这里提供了每帧卸载数量的设置, 将卸载分配到不同帧里, 用户根据实际情况进行调整.

## 七. Example\_UnloadScene (脚本 Example\_UnloadScene.cs)

此 Demo 演示卸载场景的一些特性, 因为场景有同步和异步加载过程, 并且场景加载完成都是通过 `SceneManager.sceneLoaded` 回

调触发的, 所以场景的卸载过程有些复杂( 过程复杂, 可是用户调用很简单 ):



```

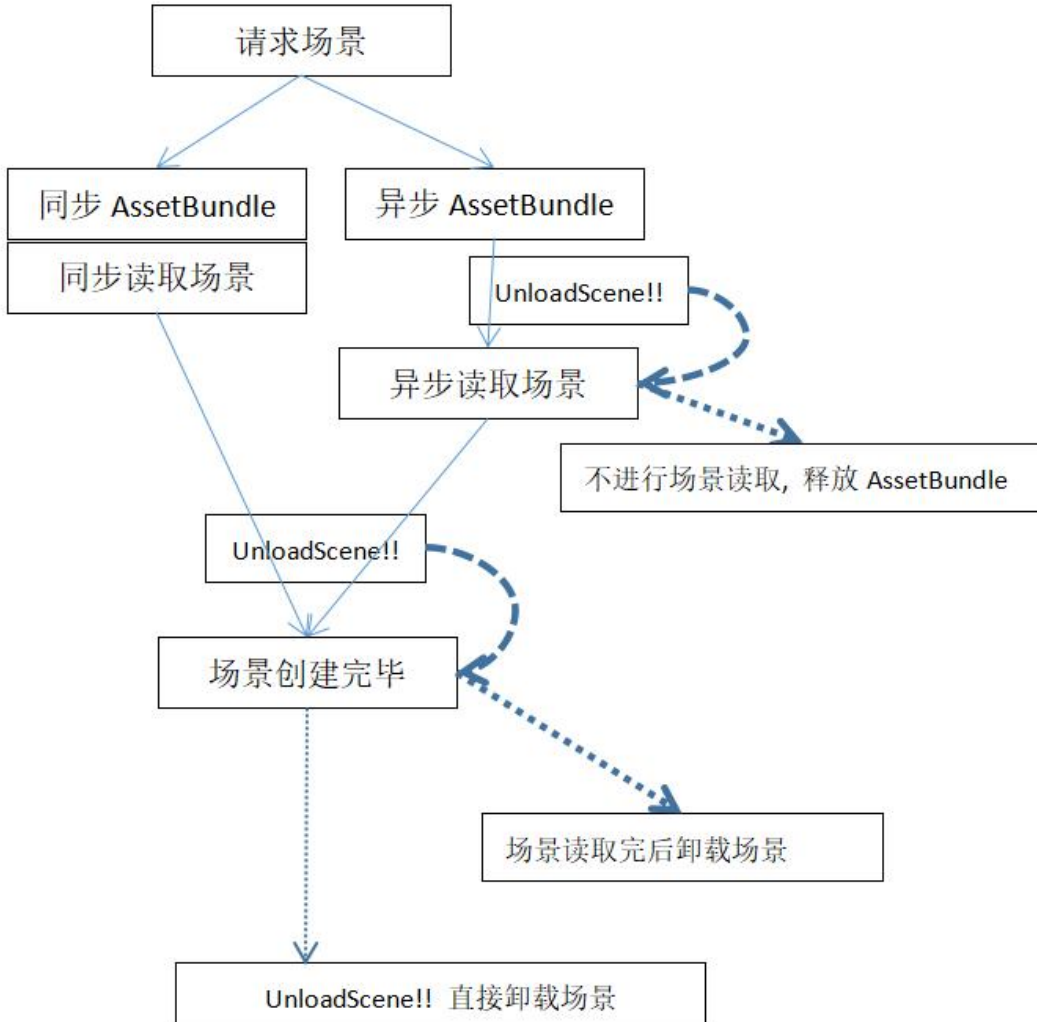
void UnloadOneScene()
{
    if(m_loadedScenes.Count > 0)
    {
        var sceneId = m_loadedScenes[0];
        m_loadedScenes.RemoveAt(0);
        Debug.Log("Request unload at frame : " + Time.frameCount);
        // Please Unload Scene In This Way
        SceneLoadManager.Instance.UnloadScene(sceneId, (_id, _scene) =>
        {
            Debug.Log("Unloaded " + _id + " [" + _scene.path + "] at frame : " + Time.frameCount);
        });
    }
}

```

主要 API (命名空间 AssetBundleMaster.ResourceLoad):

`SceneLoadManager.Instance.UnloadScene(int id, System.Action<Scene> unloaded = null);`

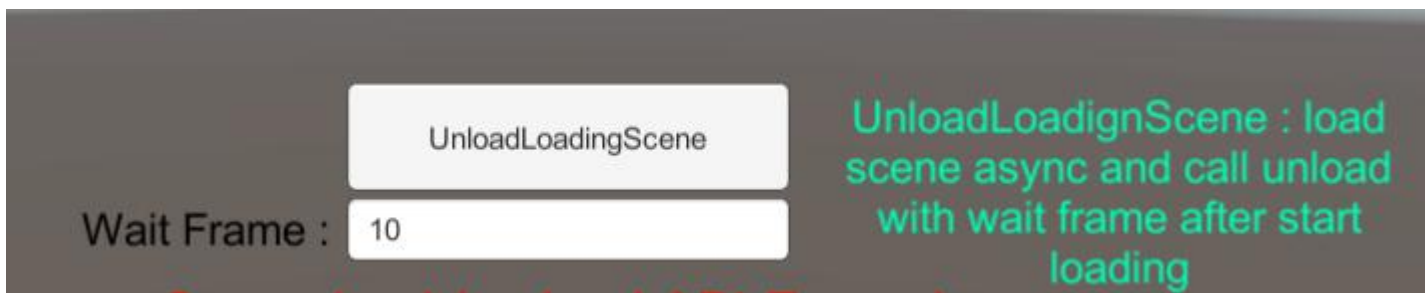
场景读取 API ( SceneLoadManager.Instance.LoadScene ) 返回的是一个 HashCode, 它指代的就是被加载出来的场景, 所以卸载场景的 API 参数也是 HashCode, 用来卸载指定场景. 场景资源也是完全自动控制的, 卸载场景都应该使用这个 API. 因为读取场景有很多异步过程, 所以卸载流程如下图( 如果非 AssetBundle 模式则没有 AssetBundle 加载步骤 ):



场景读取可能的异步过程有

1. 异步读取 AssetBundle
2. 异步读取场景
3. 等待异步回调, 不管同步加载还是异步加载场景, 都需要等待这个回调

所以 `UnloadScene` 函数内对其中的各个步骤都进行了封装, 即使在各个异步过程还在执行中进行场景卸载, 也能保证场景能够正确被卸载.



这个测试功能就是对卸载的测试, 它在开始异步加载场景后经过 `Wait Frame 10` 帧之后执行场景卸载, 把这个修改为 `1, 2, 3...`等数值可以看到在异步过程中卸载场景也是正确的.

以上就是全部的资源控制, 通过三个控制器对 `Asset, GameObject, Scene` 进行读取和卸载逻辑(命名空间

`AssetBundleMaster.ResourceLoad`):

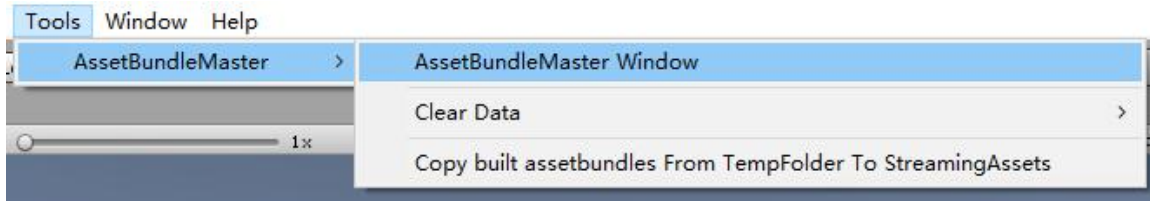
1. `ResourceLoadManager`
2. `PrefabLoadManager`
3. `SceneLoadManager`

我们如果运行在 `AssetDataBase_Editor` 模式下, 有些功能可能无法测试, 比如资源异步加载和资源重复的测试, 接下来就进行资源打包创建 `AssetBundle`, 并在编辑器下加载 `AssetBundle` 测试, 之后再发布测试.

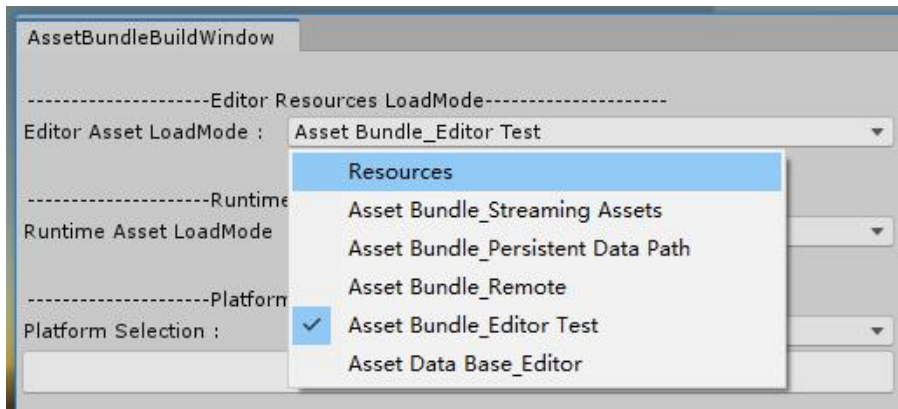
PS : `StartScene1` 左下角的下拉列表里还有一个场景 `Example_UnloadAssetEfficiency` 是用于说明资源卸载的相关特点的, 在附录中进行说明.

## Build AssetBundles

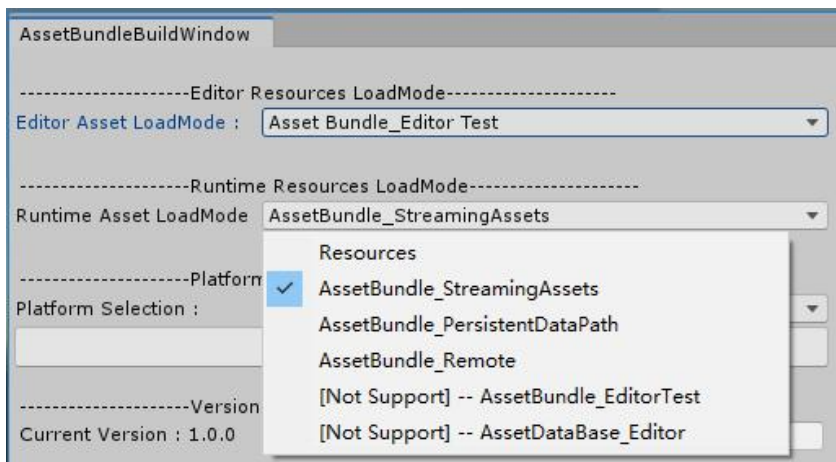
还是先打开编辑器界面



我们的读取模式有 6 种, 而发布后的支持的模式有 4 种



编辑器下可选模式



发布时可选模式

1. Resoueces 模式 : 这个是使用 Resources 文件夹下的资源的模式, 没有太多意义, 只是作为保留模式
2. AssetBundle\_StreamingAssets 模式 : 从 StreamingAssets 路径下读取 AssetBundle.
3. AssetBundle\_PersistentDataPath 模式 : 作为可更新资源模式, 在读取资源时会先检查 PersistentDataPath 下是否有资源, 没有的话会到 StreamingAssets 路径下读取资源. 查看附录的资源更新支持.

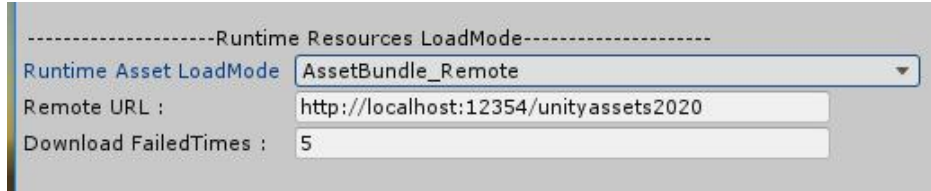


4. AssetBundle\_Remote 模式：远程 AssetBundle 加载模式, 可以从服务器等加载 AssetBundle 资源.

5. AssetBundle\_EditorTest 模式: 打包的 AssetBundle 会被放到临时路径下, 这个就是从临时路径读取 AssetBundle.

6. AssetDataBase\_Editor 模式：不需要打包直接进行编辑器下资源加载, 这个是开发者模式.

如果你使用 AssetBundle\_Remote 模式, 将会出现下列选项：

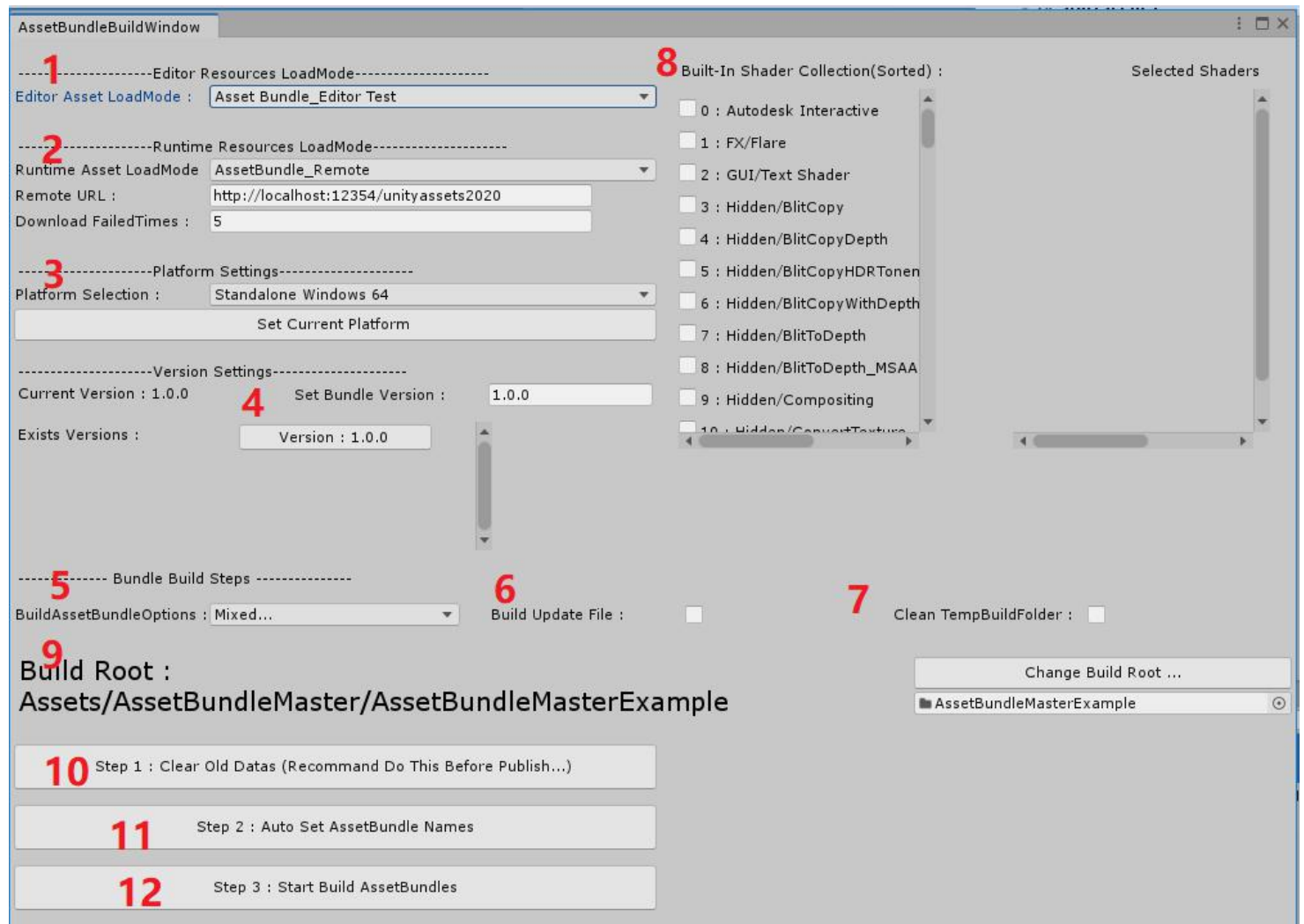


Remote URL 就是远程服务器地址, 跟你本地加载时选择的 Build Root 相似. Download FailedTimes 是下载 AssetBundle 可以失败

重试的次数, 将会把 AssetBundle 下载到本地 Cache 文件夹, 根据不同 Unity 版本使用 API 为 UnityWebRequestAssetBundle /

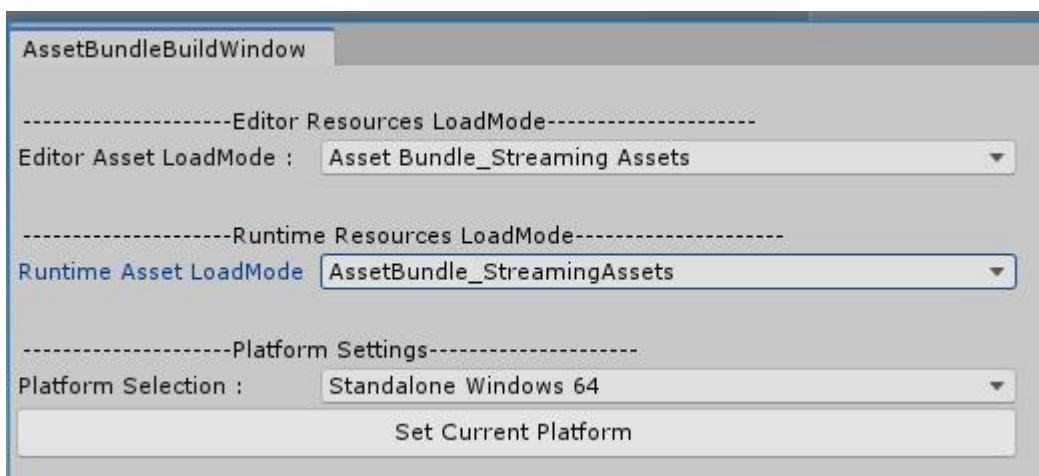
UnityWebRequest / LoadFromCacheOrDownload. 如果你使用 http 服务器进行加载测试, 注意 CORS 和 MIME 的问题.

接下来我们把面板上的所有设置看一遍:



1. Editor Asset LoadMode (EnumPop) : 编辑器下加载资源方式
2. Runtime Asset LoadMode (EnumPop) : 发布后的资源加载方式
3. Platform Selection (EnumPop) : AssetBundle 的目标平台, 点击 [Set Current Platform] 即可设置为你当前的平台
4. Set Bundle Version (Text) : 可以设置不同版本的 AssetBundle
5. BuildAssetBundleOptions (EnumFlagPop) : 打包设置
6. Build Update File (CheckBox) : 自动创建版本 Patch 文件, Patch 是 AssetBundleMaster.AssetLoad.LocalVersion.UpdateInfo 的 Josn 序列化.
7. Clean TempBuildFolder (CheckBox) : AssetBundle 被打包到临时文件夹, 临时文件夹可能含有非当前版本文件, 清除冗余文件.
8. Built-In Shader Collection (Scroll View CheckBox) : 对使用相同内建 shader 的材质进行整合, 减少重复编译.
9. Build Root (Label, Button) : 资源文件夹根目录, 所有此文件夹下的资源可以被读取.
10. Step1 Clear Old Datas (Button) : 清除已经被设置的 AssetBundleName. 可以减少冗余资源被打包.
11. Step 2 Set AssetBundle Names (Button) : 自动设置包名, 其中包含了资源的自动处理, 处理过程参看附录.
12. Step 3 Start Build AssetBundles (Button) : 开始打包.

以上就是所有设置项. 我们先把编辑器和发布的加载选项都选为 AssetBundle\_StreamingAssets, 这个是最简单的发布方式. 然后点击

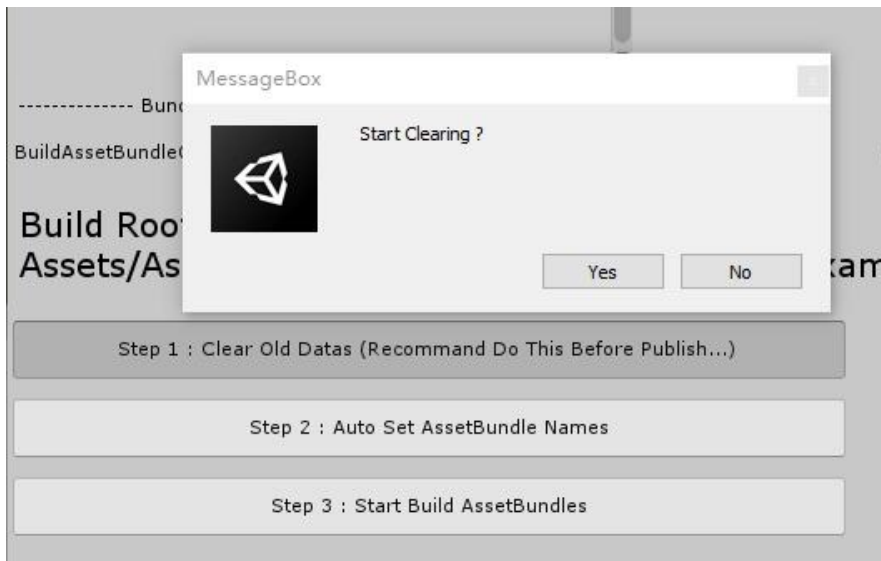


Step1...

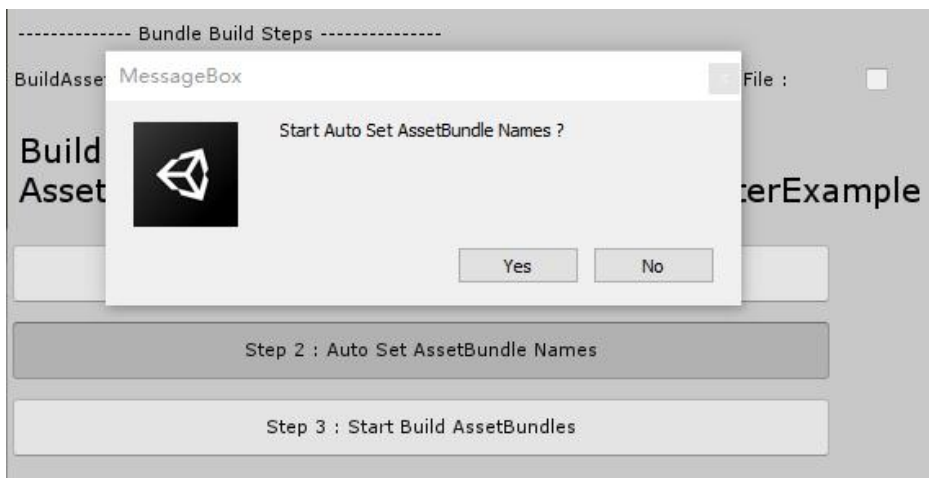
Step2...

Step3...

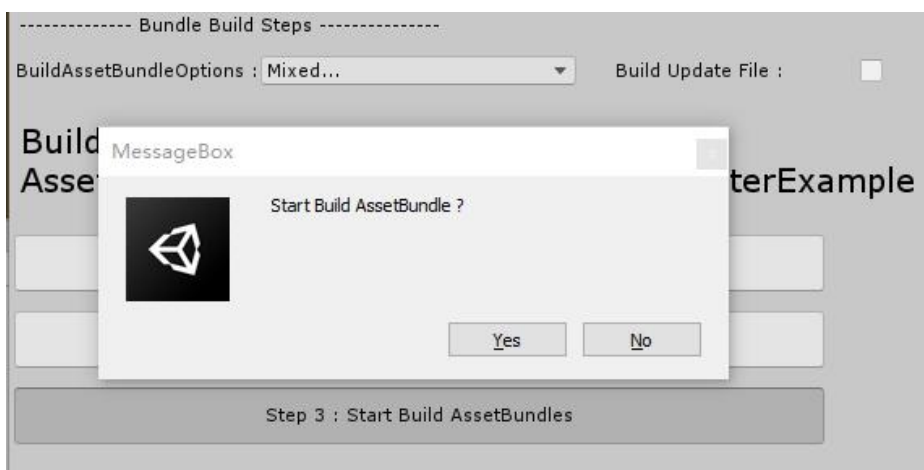
这三步就会自动设置和打包 AssetBundle 了。 每一步都有提示操作。



Step1

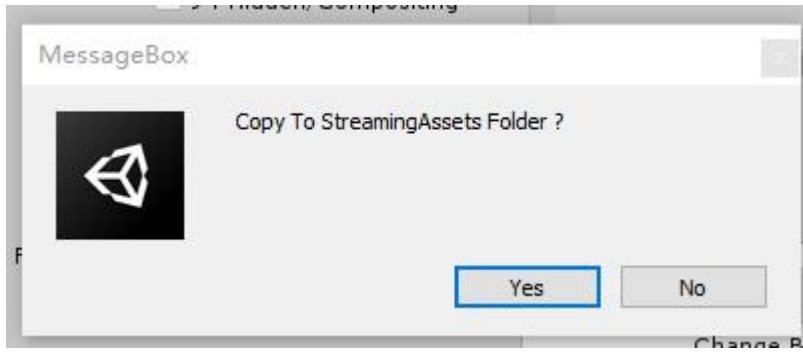


Step2

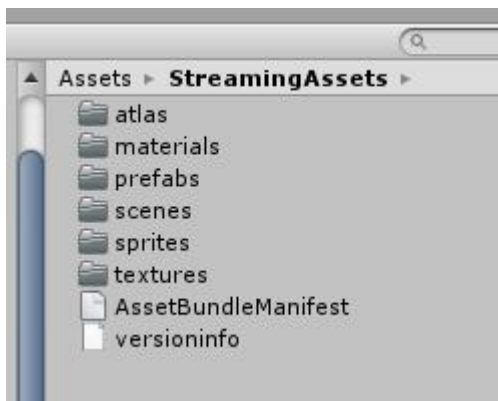


## Step3

当创建完成后会询问是否从临时文件夹复制 AssetBundle 到 StreamingAssets 文件夹下, 选择 Yes 即可.



可以看到 StreamingAssets 下面有打包好的文件:

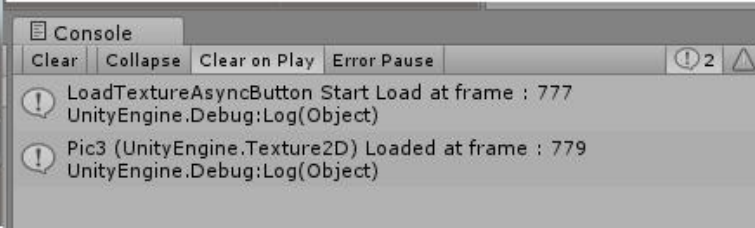


此版本添加了"一键打包"按钮, 它把 Step1, Step2, Step3 集合在一起了, 在右下角按钮 :



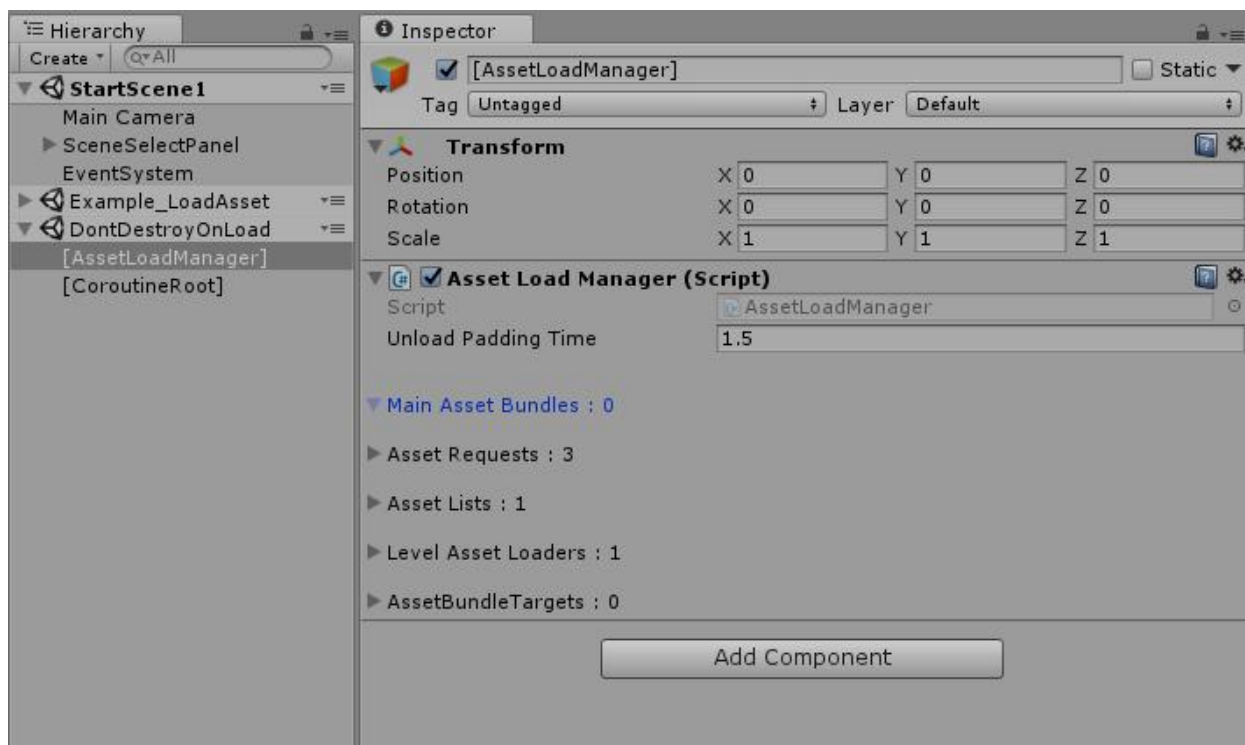
这样就打包好了, 如果在编辑器下运行, 只需要再打开 StartScene1 场景即可, 重复上面的 API 测试步骤, 可以看到异步读取打印的

Log 信息确实是异步的了.

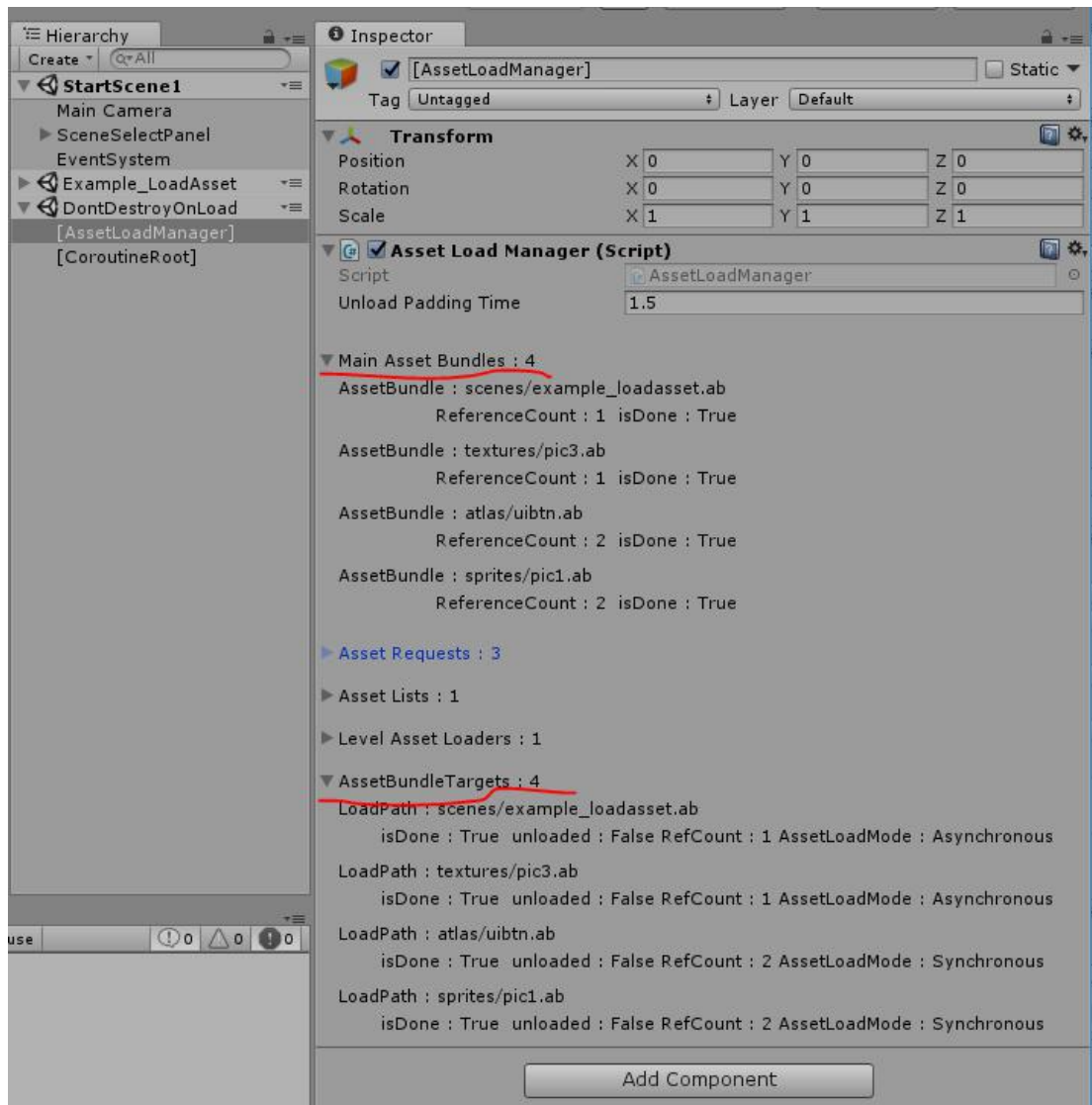


我们在 AssetDataBase\_Editor 模式和 AssetBundle\_StreamingAssets 模式下读取资源的内容也不一样了, 对比一下检视面板

[AssetLoadManager]:



AssetDataBase\_Editor 没有 AssetBundle 相关资源



AssetBundle\_StreamingAssets 模式下读取了 AssetBundle

这样就完成了打包和在编辑器下读取 AssetBundle 的流程了。请注意在编辑器下 AssetBundle 模式资源卸载可能不正确，甚至一些引用资源的卸载也不正确，因为编辑器下编辑器自身会引用到非运行资源比如 SpriteAtlas，在自动卸载资源的时候会造成卸载错误(比如 UI 图片丢失)，不过只在编辑器下会发生，所以最好的开发模式仍然是选择 AssetDataBase\_Editor 模式，而运行测试放在实际平台进行发布测试。目前发现的编辑器自动引用对象都是 SpriteAtlas，如果一定要在编辑器下进行 AssetBundle 模式的测试，请在生成 AssetBundle 之后手动删除编辑器下的 .spriteatlas 文件，这样就不会造成运行时错误卸载资源了(仅限于 SpriteAtlas 的情况)。

如果要发布到目标平台，那么在打包面板的 Platform Selection 选项中选择目标平台打包即可，接下来是最重要的一点，场景与 Build Settings :

在发布时，AssetBundle 模式是不需要把场景加到 Build Settings 中的，因为如果场景在 Build Root 文件夹中，场景会被打包成 AssetBundle，而加入到 Build Settings 中的场景也会被自动打包到 Resources 里面，这样场景就会被打包两次，浪费时间和空间。比如我们当前的工程，如果想发布的话，我们怎样让程序启动之后自动加载初始场景 StartScene1 呢？

第一种方法是把 StartScene1 加到 Build Settins 中，因为 StartScene1 很小很简单，被两次打包也不会产生什么影响：



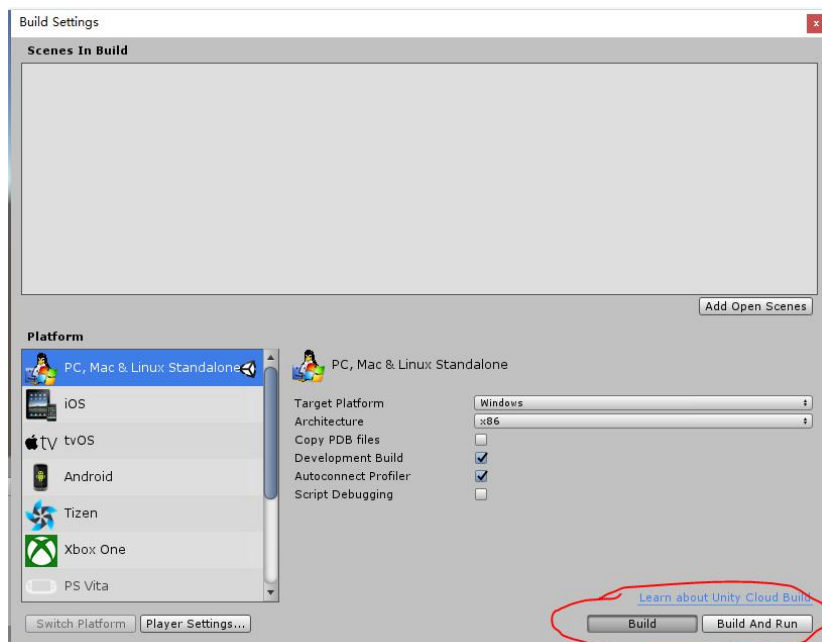
第二种方法是使用初始化调用属性来读取初始场景(可以写在任何代码中), 这种方式更好, 因为规避了二次打包的风险 :

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    SceneLoadManager.Instance.LoadScene("Scenes/StartScene1");
}
```

如果你使用的是 AssetBundle\_Remote 模式 :

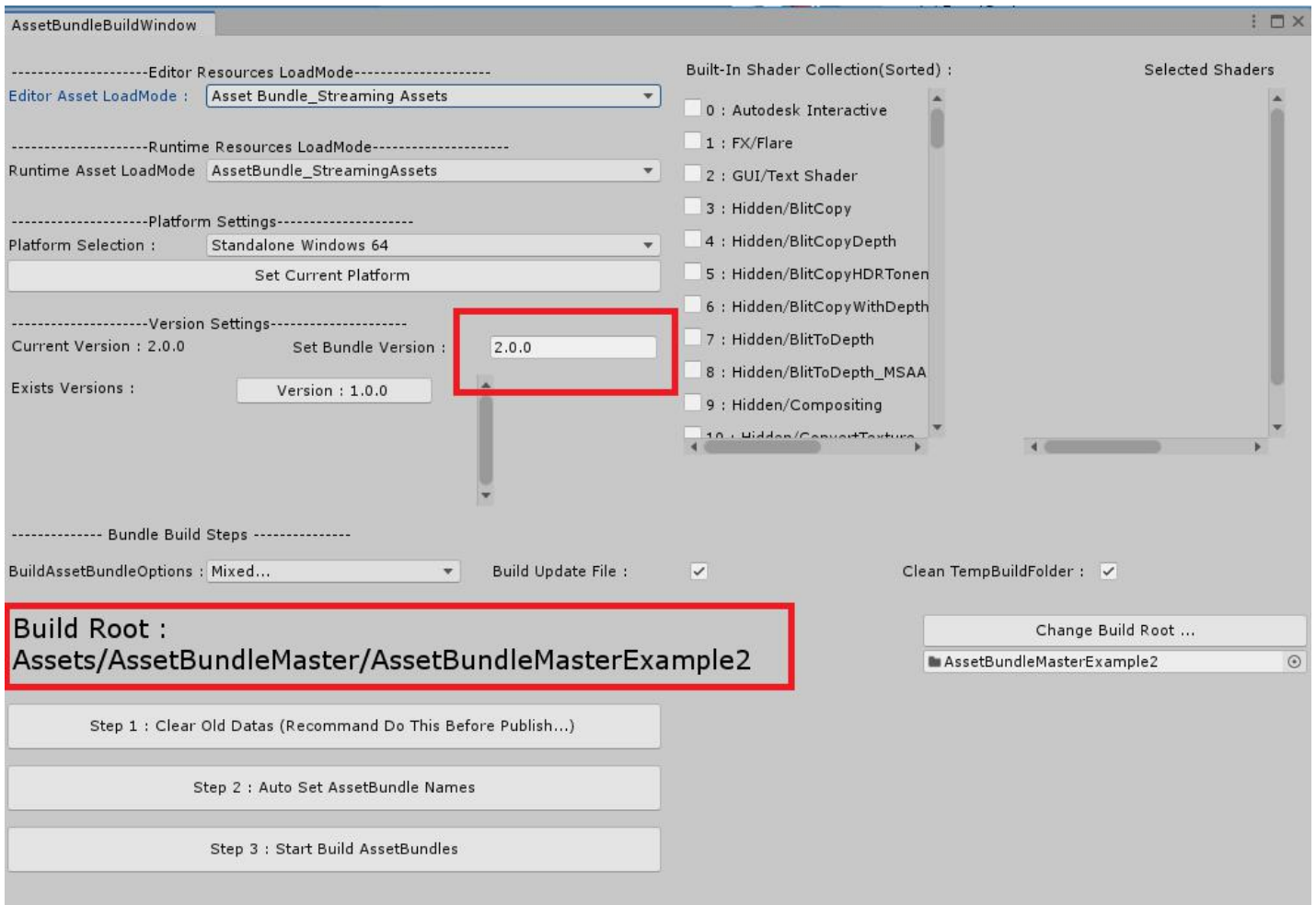
```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    // If AssetBundle_Remote mode, muse wait for Inited
    AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
    {
        SceneLoadManager.Instance.LoadScene("Scenes/StartScene1");
    });
}
```

这段代码请自行添加到任意脚本中. 这时 Build Settings 里面就不需要加场景了. 虽然需要写代码.



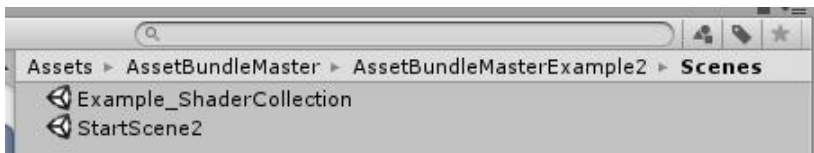
以上就是所有 API 以及 AssetBundle 创建和发布的流程了。

现在开始说明打包功能 8. Built-In Shader Collection 的作用, 我们直接修改 Build Root 读取 AssetBundleMasterExample2 :



只简单修改了一下版本, 打包 2.0.0 版本, 因为我们之前 Build 了 1.0.0 版本, 所以 Exists Versions 里面显示出了 1.0.0 版本, 这里我们

试一试 Update 文件创建, 勾选它, 修改 Build Root 为 AssetBundleMasterExample2, 我们看看这个资源文件夹下的文件:



Example\_ShaderCollection 场景中有一些建筑, 使用了大量标准材质(Standard Shader):

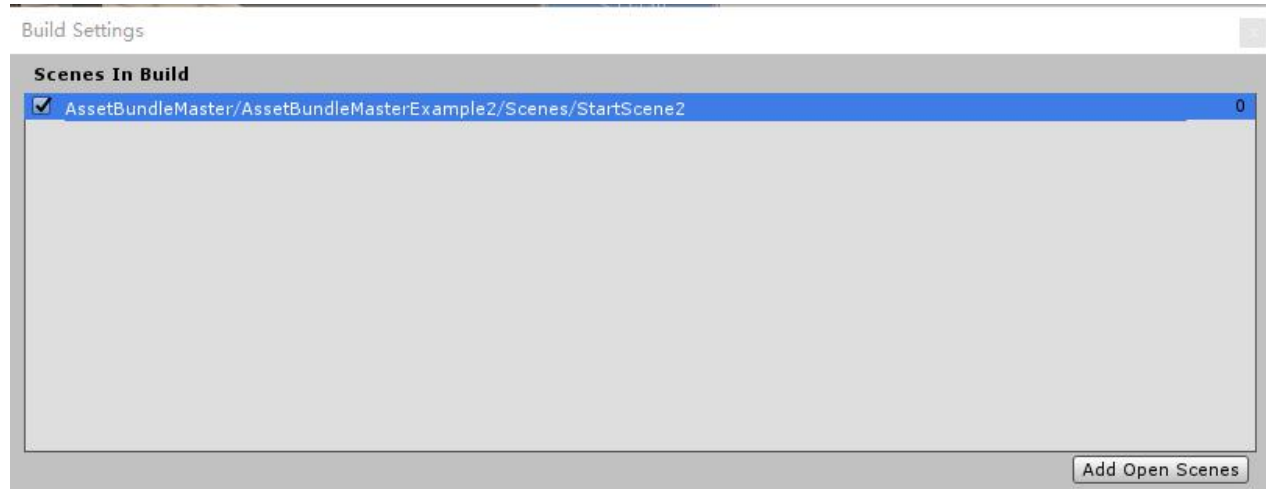




这个场景引用了很多标准材质, 会造成一些性能问题. 我们现在要加载的初始场景为 StartScene2(它自动读取

Example\_ShaderCollection 场景), 把刚才的初始场景改一改:

第一种方式 :



第二种方式 :

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    SceneLoadManager.Instance.LoadScene("Scenes/StartScene2");
}
```

如果你使用的是 AssetBundle\_Remote 模式 :

```
[UnityEngine.RuntimeInitializeOnLoadMethod]
public static void LoadEntryScene()
{
    // If AssetBundle_Remote mode, muse wait for Inited
    AssetLoadManager.Instance.OnAssetLoadModuleInited(() =>
    {
        SceneLoadManager.Instance.LoadScene("Scenes/StartScene2");
    });
}
```

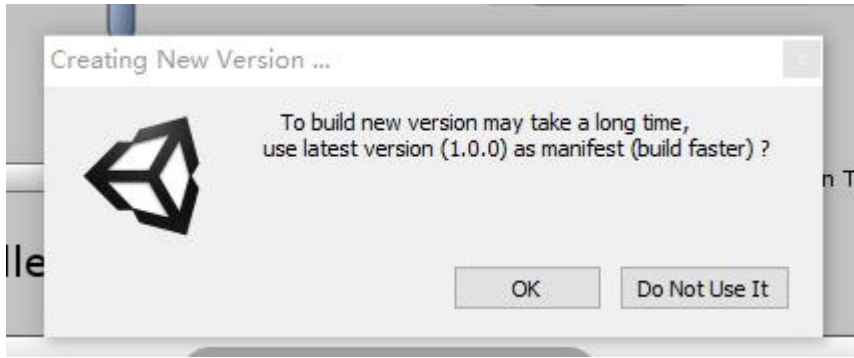
再次执行

Step1...

Step2...

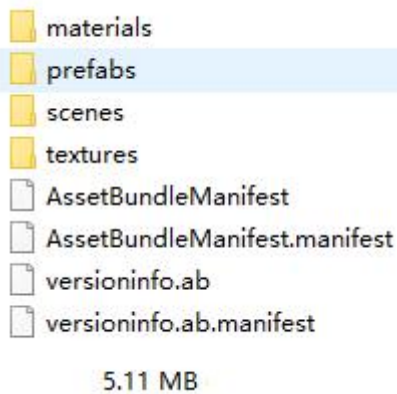
Step3...

这次在执行完 Step3 时出现了新的提示信息:

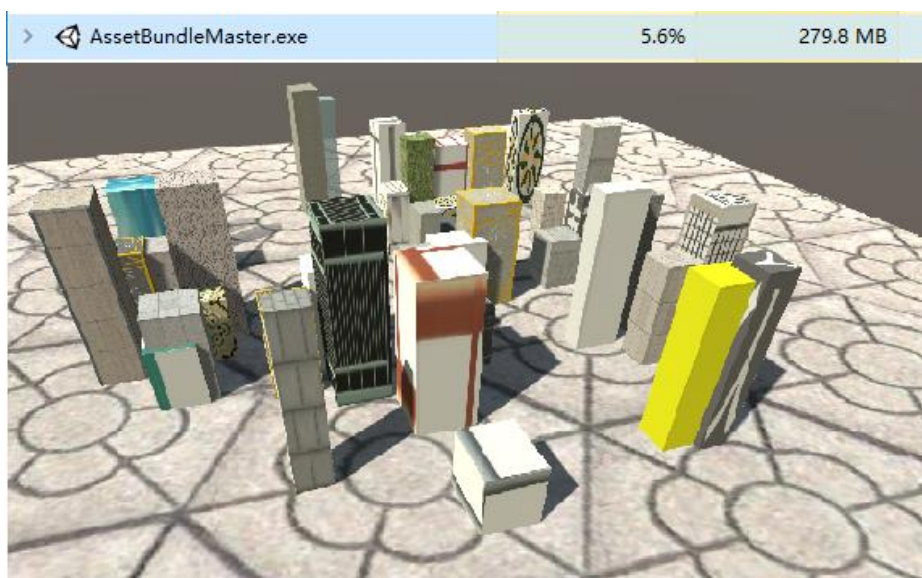


这是因为我们已经有 1.0.0 版本的 AssetBundle 包了, 如果我们是增量打包, 复制 1.0.0 的文件到 2.0.0 版本文件夹下, 再开始打包会比较重新打包快很多, 不过 2.0.0 打包的是 AssetBundleMasterExample2 的资源而 1.0.0 打包的是 AssetBundleMasterExample 的资源, 完全不同. 所以功能 7. Clean TempBuildFolder (CheckBox) 的功能就体现出来了, 它能清除打包完后不属于这个版本的文件. 不管你选择哪个都相当于重新打包, 等到打包结束, 就开始 Build 一个 app 来查看这个场景的特点吧.

在打包完成之后, 我们可以 Build app 然后运行来测试一下, 不要忘了在 Build Settings 勾选 [Development Build] 和 [Autoconnect Profiler]. 我的平台为 Win10, 当前测试使用 Unity2020.



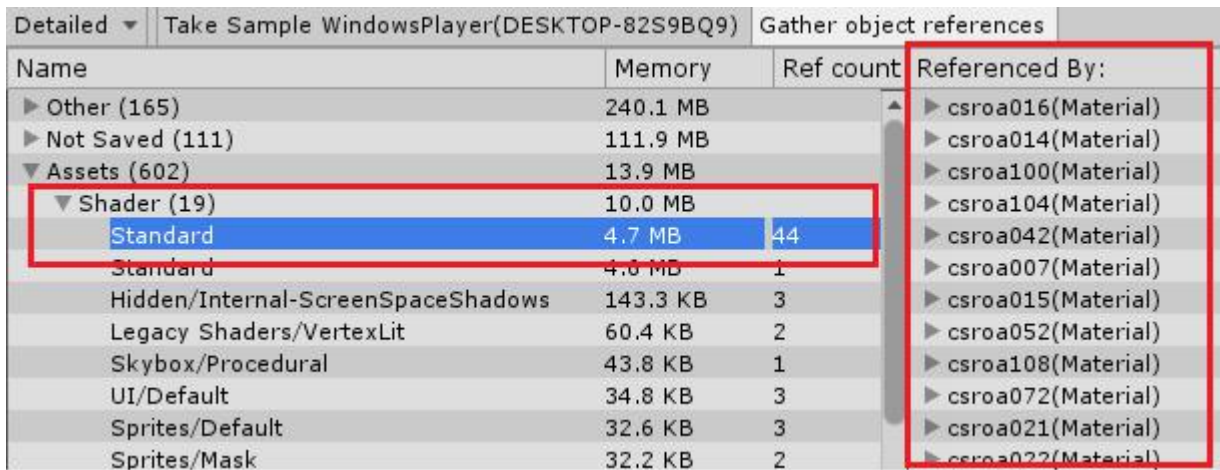
全部 AssetBundle 大小 : 5.11MB(包含 .manifest 文件)







## Task Manager 显示暂用内存 122.0MB VS 279.8MB



Name	Memory	Ref count	Referenced By:
▶ Other (165)	240.1 MB		▶ csroa016(Material)
▶ Not Saved (111)	111.9 MB		▶ csroa014(Material)
▼ Assets (602)	13.9 MB		▶ csroa100(Material)
▼ Shader (19)	10.0 MB		▶ csroa104(Material)
Standard	4.7 MB	44	▶ csroa042(Material)
Standard	4.6 MB	1	▶ csroa007(Material)
Hidden/Internal-ScreenSpaceShadows	143.3 KB	3	▶ csroa015(Material)
Legacy Shaders/VertexLit	60.4 KB	2	▶ csroa052(Material)
Skybox/Procedural	43.8 KB	1	▶ csroa108(Material)
UI/Default	34.8 KB	3	▶ csroa072(Material)
Sprites/Default	32.6 KB	3	▶ csroa021(Material)
Sprites/Mask	32.2 KB	2	▶ csroa022(Material)

Profiler : Shader 10M VS 113MB, Standard Shader 只进行了一次编译, 被引用了 44 次

可以看到两个版本的巨大差异, 在 2.0.0 版本浪费了更多的 I/O 时间, 消耗了更多的内存. 这个问题主要是 Built-In Shader 引起的( 根据使用方法的不同有不同的结果, 比如 UGUI 用的材质是内置材质, 都是同一个, 所以不会产生多次编译的问题 ).

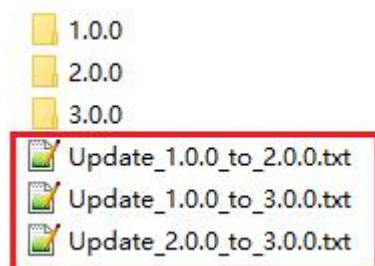
这个测试在各种版本之间都存在( Unity5/Unity2017/Unity2018/Unity2019/Unity2020 ), 几乎得到同样的结果.

PS : Shader collection 在小型项目中使用可以解决这些问题, 不过在大中型项目中如果大量使用内建材质的话, 不推荐勾选任何 shader collectoin. 请使用官方推荐的方案.

我们来看一下 AssetBundles 临时文件夹, 它在跟 Assets 文件夹同一层的 :



Assets/./AssetBundles/StandaloneWindows64/



你可以看到不同版本的文件夹以及升级补丁文件, 它是 Json 格式文件, 内容如下(Update\_2.0.0\_to\_3.0.0.txt) :

```

1 {
2   "buildTime": "2021-04-19 10:18:32",
3   "updateList": [
4     "prefabs/city.ab",
5     "shadercollection/standard.ab",
6     "versioninfo.ab"
7   ],
8   "deleteList": [
9     "materials/csroa000.ab",
10    "materials/csroa001.ab",
11    "materials/csroa002.ab",
12    "materials/csroa003.ab",
13    "materials/csroa004.ab",
14    "materials/csroa007.ab",
15    "materials/csroa008.ab",
16    "materials/csroa012.ab",
17    "materials/csroa014.ab",
18    "materials/csroa015.ab",
19    "materials/csroa016.ab",
20    "materials/csroa019.ab",
21    "materials/csroa020.ab",
22    "materials/csroa021.ab",
23    "materials/csroa022.ab",
24    "materials/csroa023.ab",
25    "materials/csroa026.ab",
26    "materials/csroa032.ab",
27    "materials/csroa034.ab",
28    "materials/csroa036.ab",
29    "materials/csroa041.ab",
30    "materials/csroa042.ab",
31    "materials/csroa052.ab",
32    "materials/csroa060.ab",
33    "materials/csroa066.ab",
34    "materials/csroa070.ab",
35    "materials/csroa071.ab",
36    "materials/csroa072.ab",
37    "materials/csroa073.ab",
38    "materials/csroa096.ab",
39    "materials/csroa098.ab",
40    "materials/csroa099.ab",
41    "materials/csroa100.ab",
42    "materials/csroa101.ab",
43    "materials/csroa102.ab",
44    "materials/csroa103.ab",
45    "materials/csroa104.ab",
46    "materials/csroa105.ab",
47    "materials/csroa106.ab",
48    "materials/csroa107.ab",
49    "materials/csroa108.ab",
50    "materials/csroa109.ab",
51    "materials/csroa110.ab",
52    "materials/csroa111.ab"
53  ]
54 }

```

这意思是你从 2.0.0 版本设计到 3.0.0 版本的话, 需要从服务器或 CDN 下载 "updateList" 里面的文件, 而删除 "deleteList" 里面的本地文件. 这个文件的反序列化是 `AssetBundleMaster.AssetLoad.LocalVersion.UpdateInfo` 类型.

```

public class UpdateInfo : BaseJsonSerialization
{
    [SerializeField]
    public string buildTime;

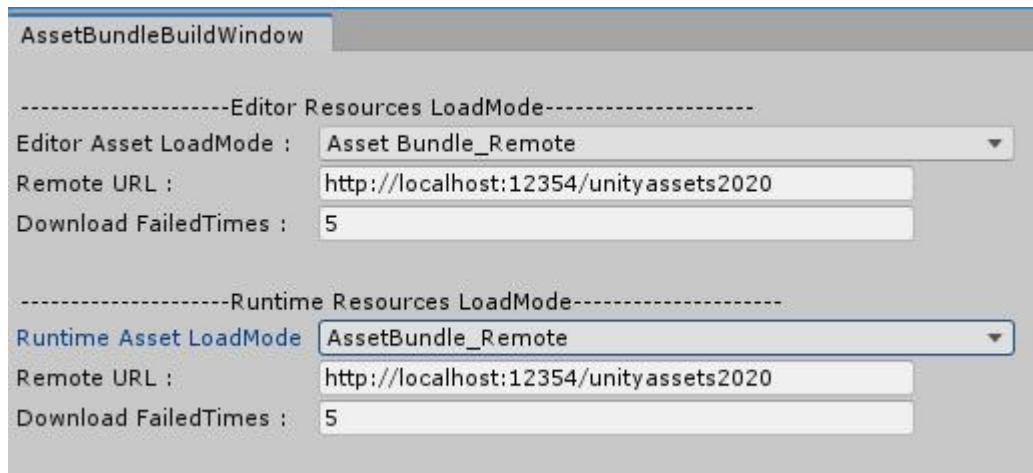
    public System.Version from;
    public System.Version to;

    [SerializeField]
    public List<string> updateList = new List<string>();
    [SerializeField]
    public List<string> deleteList = new List<string>();
}

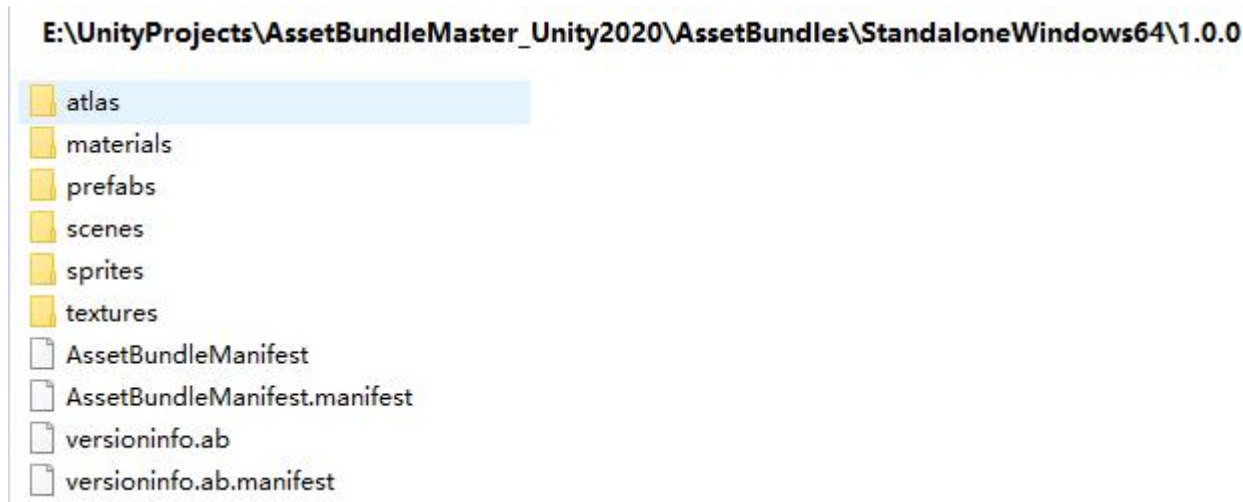
```

注意：这里只提供了升级补丁文件( update-patch files ), 并没有提供下载更新文件的逻辑, 用户必须自己实现下载更新的系统.

注意 2: 如果你只是需要从远程地址加载 AssetBundle,你只要把加载模式 AssetLoadMode 设为 AssetBundle\_Remote 即可, 它将会从本地或远程地址读取最新的资源(远程资源会缓存到本地).



使用 AssetBundle\_Remote 从服务器读取资源的例子. 只需要把打包好的资源放在服务器下即可, 比如这是本地打包的临时地址:

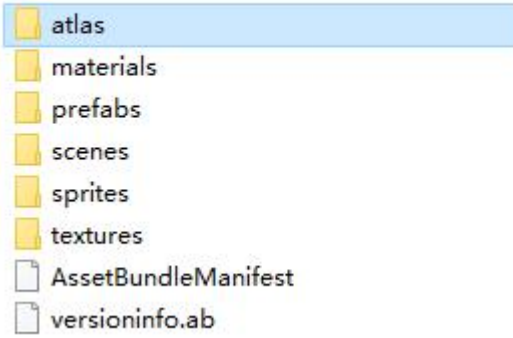


复制资源到服务器路径( 此处使用一个 IIS 服务器, 它的物理目录在 E:\LocalServer\WebGL\_Raw, port:12354), 也在Web.config 设置了文件传输标准:

```
<mimeMap fileExtension="." mimeType="application/octet-stream" />
<mimeMap fileExtension=".assetbundle" mimeType="application/octet-stream" />
<mimeMap fileExtension=".ab" mimeType="application/octet-stream" />
```

这样就可以从服务器下载资源了.

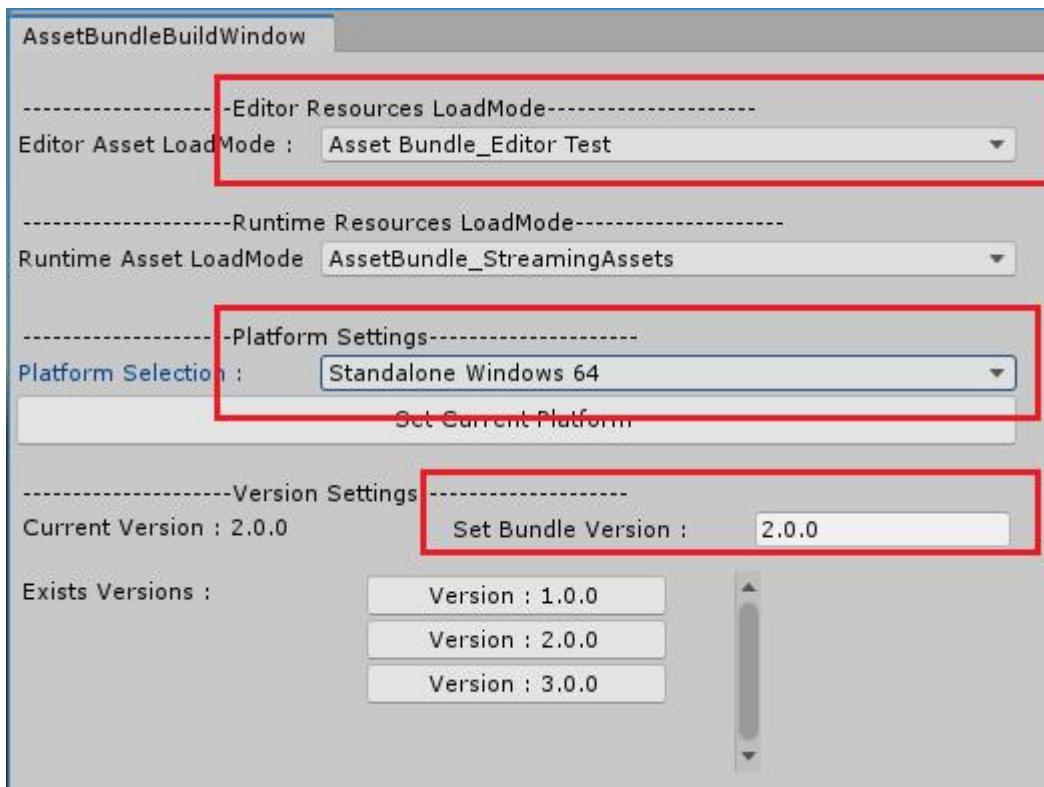
E:\LocalServer\WebGL\_Raw\unityassets2020



复制到服务器的文件可以忽略 .manifest 文件.

AssetBundle\_EditorTest 模式就是从临时文件夹内读取资源的, 所以可以很方便地在已经打包好的各个版本间切换, 方便测试. 读取

的平台和版本就是编辑器面板中的当前设定:



点击 Exists Versions 中的 Version:x.x.x 按钮也能切换到相应版本

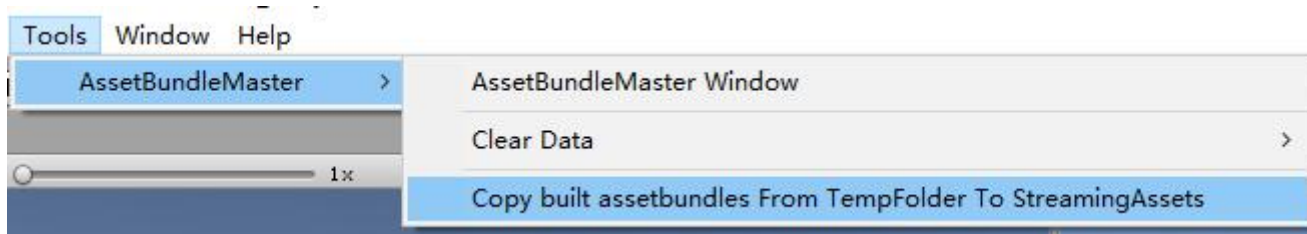
临时文件为 Assets/./AssetBundles/StandaloneWindows64/x.x.x/

PS : 你运行的时候可以看 LOG 信息, 它也会显示相关路径信息.

工具栏其余功能:

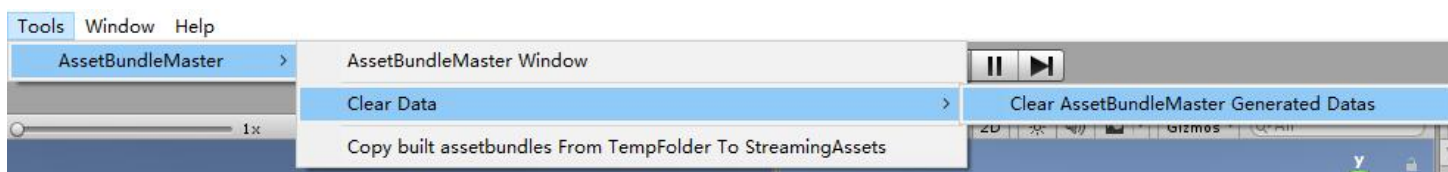


1. 从临时文件夹复制 AssetBundle 到 StreamingAssets 文件夹：



一般在打包完成后会出现提示框提示用户是否要复制到 StreamingAssets 文件夹, 这个是手动复制.

2. 清除 AssetBundleMaster 产生的数据：



它会清除一些编辑器产生的序列化文件. 没有这些文件 AssetBundleMaster 将无法运行.

3. Open Caching Folder

如果你使用 AssetBundle\_Remote 从服务器加载, 你可以看到 Cache 文件夹里有缓存文件, Unity5 显示的缓存地址没有缓存文件可能哪里有错误, Unity2017~Unity2020 缓存文件夹显示正常.

**这就是 AssetBundleMaster 提供的所有服务了, 这个解决方案最初设计为中小型项目提供方便且正确的资源管理的方案, 提供了灵活的资源管理流程.**

**当前的 AssetBundleMaster 3.3.0 版本, 通过提高框架效率, 以及一直围绕的无重复资源的核心设计, 已经可以支撑大型项目了. 希望能对你的项目有所帮助.**

附录

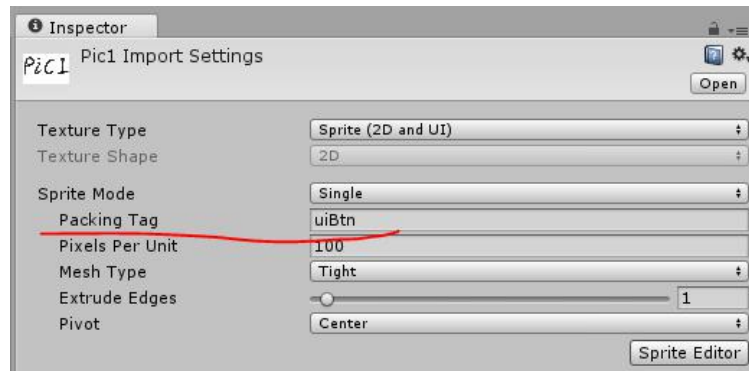
1. SpriteAtlas, 在 Unity5 中无法自己创建 SpriteAtlas, 在 Unity2017 之后可以创建, 代码在

AssetBundleBuildWindow.cs::CreateSpriteAtlasAssets 函数内, 通过把所有相同 Packing Tag 的 Sprite 资源设置到同一个 AssetBundle

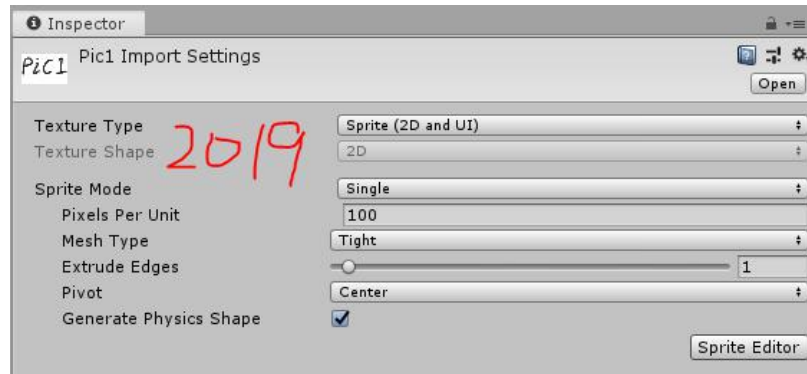
包内, 规避了 SpriteAtlas 可能被重复打包以及造成额外 DrawCall 的问题(Unity5 中同样适用). 这里使用了 Packing Tag 作为分包的依据,

可是在 Unity2019 中检视面板没有显示 Packing Tag 不过它仍然存在于序列化数据之中, 因此 AssetBundleMaster 提供了一个重载的检

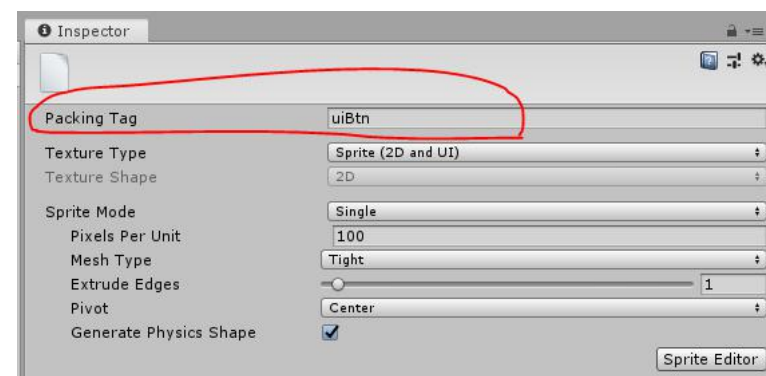
视面板, 添加了 Packing Tag 设置, 编辑器代码为 TextureImporterInspector.cs :



## Unity5



## Unity2019

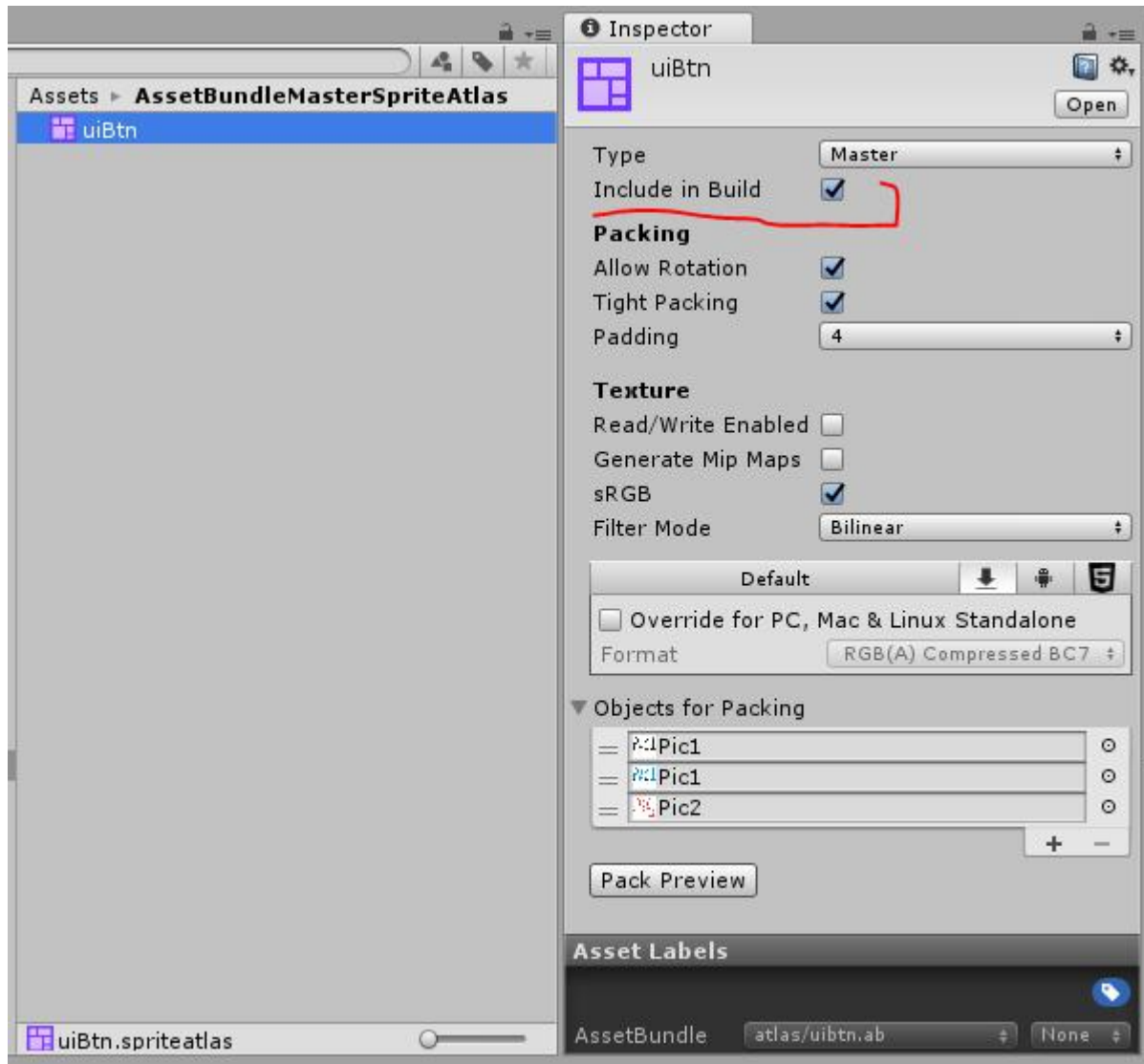


## AssetBundleMaster 重载了 Unity2019 检视面板

注意图集如果因为一些原因导致 Include in Build 的信息丢失(比如资源更新等), ResourceLoadManager 通过对

UnityEngine.U2D.SpriteAtlasManager.atlasRequested 添加了回调, 对这种情况作了补偿操作, 最大限度保证读取的正确性, 代码在

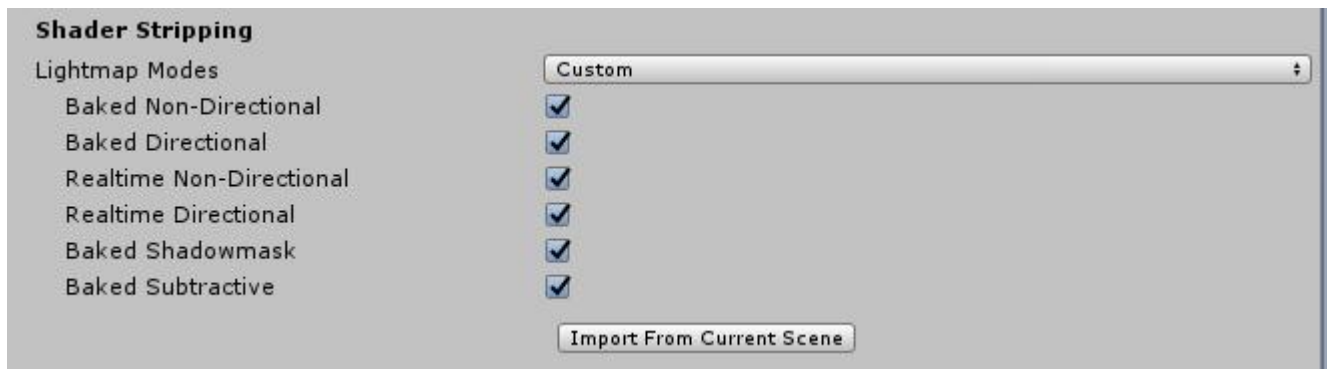
ResourceLoadManager.RequestAtlas, 用户如果碰到此问题请到这里进行调试(Unity2017 以及之后的版本).



SpriteAtlas 会被自动创建到 Assets/AssetBundleMasterSpriteAtlas 文件夹下(Unity2017 以及之后的版本).

2. EditorConfigSettings, 自动调整一些编辑器下的设置, 用户可以根据自己的工程修改此处代码. 代码在

AssetBundleBuildWindow::1679 行 EditorConfigSettings 函数内, 它首先修改了 UnityEditor.EditorSettings.spritePackerMode 以方便创建 SpriteAtlas, 并修改了 GraphicsSettings 的相关变量, 防止打包后场景中物体 LightMap 信息被错误剥离. 见图 :

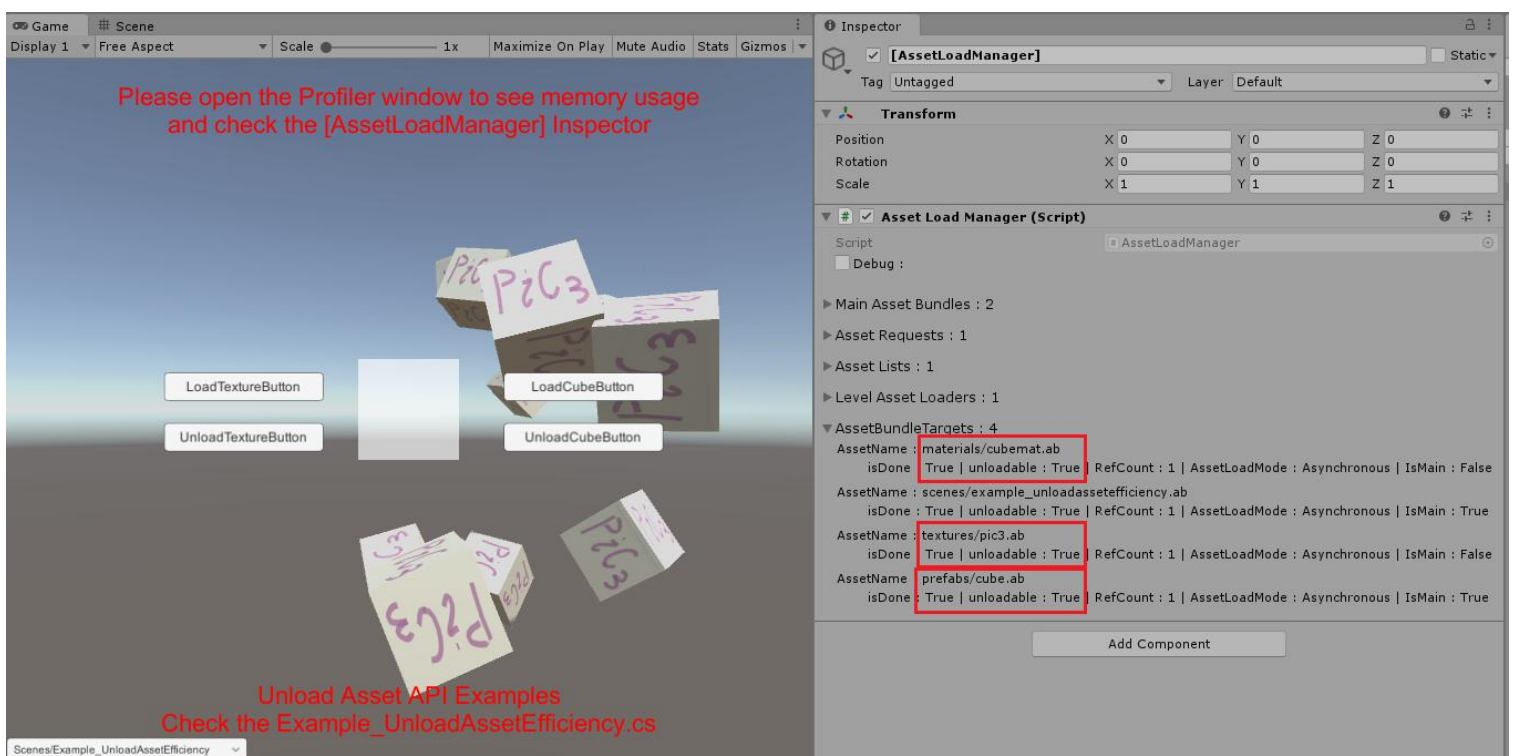


3. 资源自动控制

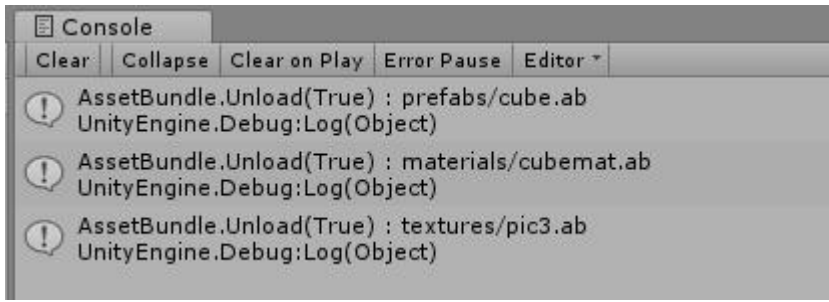
AssetBundle 模式下的自动资源释放逻辑, 在实现上比较复杂, 不过我们的核心是让资源不会在内存中重复, 我们使用弱引用来监听资源是否还存活, 如果资源仍然存活将不会卸载 AssetBundle 包, 从这方面来说它的逻辑又很简单明了. 我们可以通过 Demo : Example\_UnloadAssetEfficiency 进行说明. 用户请选择 AssetBundle 模式并打包, 然后仍然通过 StartScene1 的左下角选择框来进入此场景 :



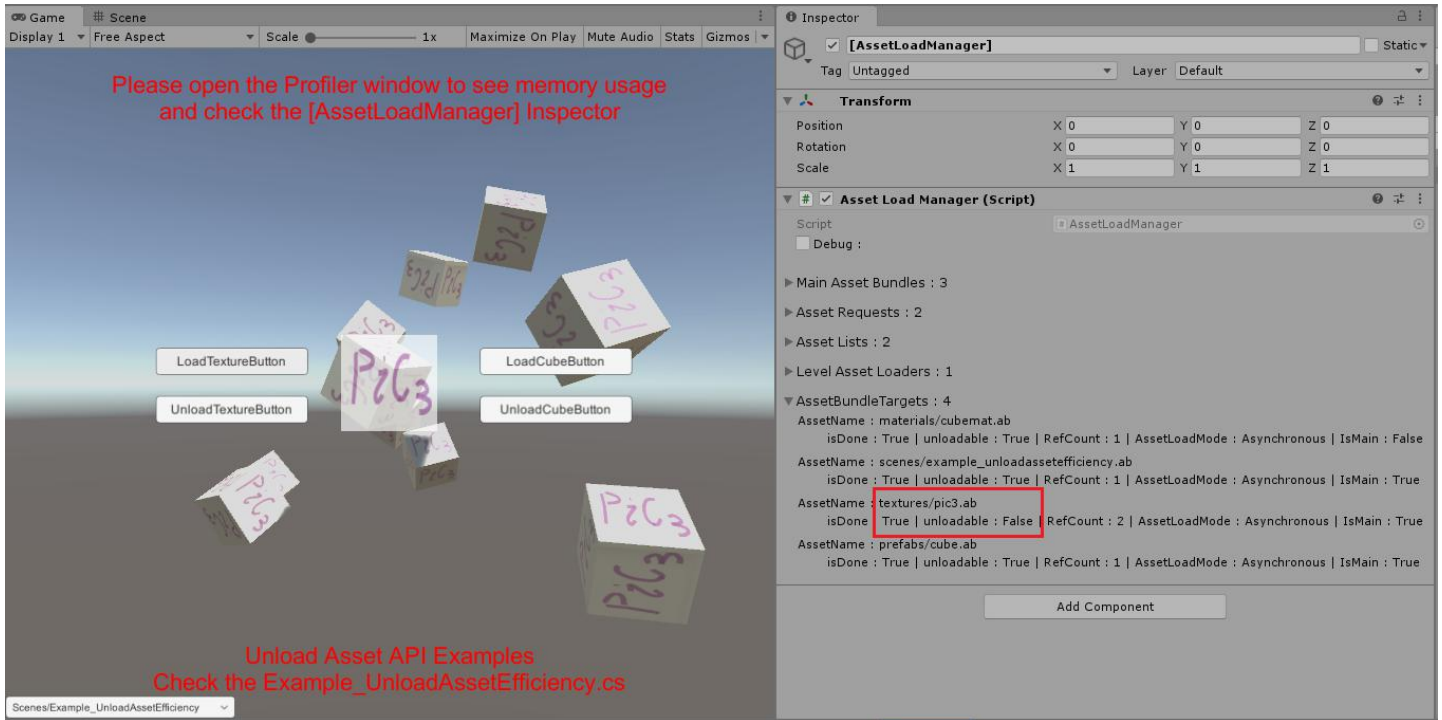
点击 LoadCubeButton 读取 GameObject, 可以看到所有 AssetBundle 都是完全控制的.



点击 UnloadCubeButton, 看到删除资源逻辑直接对 AssetBundle 进行了卸载.

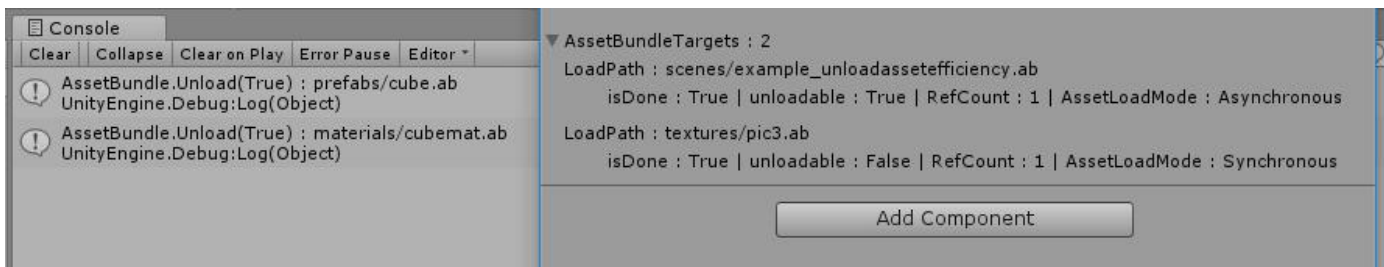


我们这次把 Cube 和 Pic3 都显式加载出来, 点击 LoadCubeButton, LoadTextureButton:



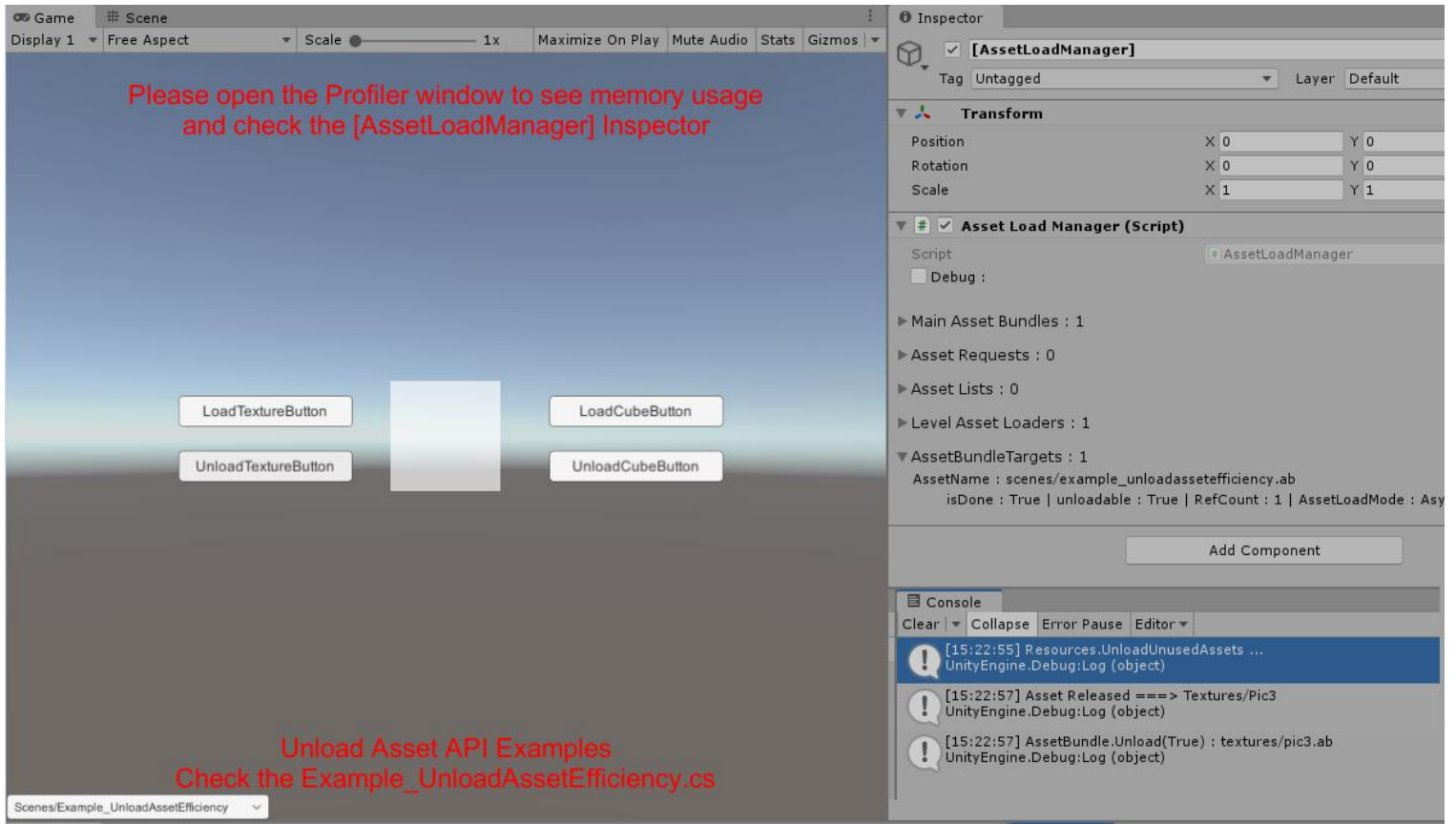
可以看到这时 textures/pic3.ab 被设置为了 unloadable: False, 意思是这个 Pic3 已经成了非完全控制资源了, 如果我们点击

UnloadCubeButton, 看到删除资源逻辑变成了:



textures/pic3.ab 这个 AssetBundle 被保留了, 因为它被 ResourceLoadManager 显式加载了, 并且有引用, 我们继续点击

UnloadTextureButton:

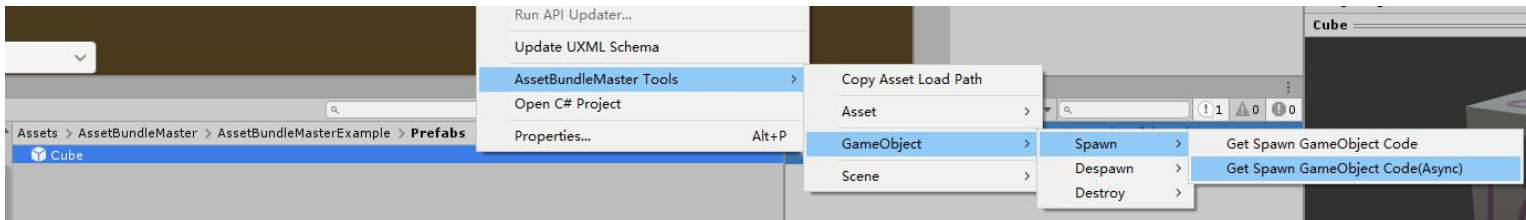


结果是先触发了 `Resources.UnloadUnusedAssets()`; 然后通过检测弱引用, 当资源被系统回收了之后, 再触发卸载 `AssetBundle` 的操作, 通过这些就保证了资源不会被重复加载.

以上就是自动卸载资源的逻辑基础了, 通过自动卸载保证了用户卸载资源的准确性和多人开发情况下资源不会被错误卸载的安全性, 更大的益处在于保证了资源不会被重复加载进内存造成溢出的特性.

#### 4. 快速工具 :

添加了一些快速代码复制的工具, 在 Project 面板的资源上右键点击可以展开 `AssetBundleMasterTools` 选项 :



如图在 `Cube.prefab` 文件上右键, 选择 `[Get Spawn GameObject Code(Async)]` 之后, 会将下面代码复制到剪贴板 :

```
AssetBundleMaster.ResourceLoad.PrefabLoadManager.Instance.SpawnAsync("Prefabs/Cube.prefab", (Cube_go)=> {});
```

通过 `Ctrl + V` 粘贴到代码中即可. 如果没有选中资源, 或者资源处于无法通过 `AssetBundleMaster` 读取的路径( 资源在 `Build Root` 之外时 ), 复制出来的代码路径就像下面这样 :

```
AssetBundleMaster.ResourceLoad.PrefabLoadManager.Instance.SpawnAsync("", (Cube_go)=>{ });
```

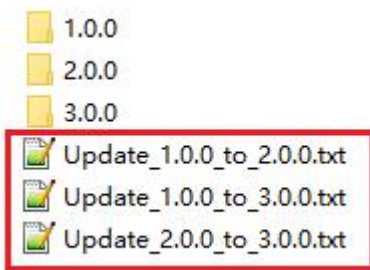
减少了代码工作量.

Update support :

1. 我们没有实现更新系统, 只实现了生成更新补丁文件, 只要有新的版本生成, 它就会刷新所有的补丁文件信息, 这样你的更新系统就

可以使用补丁文件作为参考.

Assets/./AssetBundles/[Platform]/



你可以通过 LocalVersion 中的 VersionInfo 来获取当前资源版本的信息 :

```
var version = AssetBundleMaster.AssetLoad.LocalVersion.Instance.versionInfo.BundleVersion;  
Debug.Log(version);
```

2. 如果你使用 AssetBundle\_PersistentDataPath 模式, AssetBundleMaster 会优先从 Application.persistentDataPath 路径中读取资源, 如果该路径没有文件, 则会从 Application.streamingAssetsPath 中去读取. 这就是说你可以发一个基础版本(1.0.0 或 2.0.0)并在 StreamingAssets 之中包含该版本的资源, 然后你可以下载新版本资源到 Application.persistentDataPath 路径下, 运行时就会读取新的资源了. 几乎所有的平台都是 persistentDataPath 可读写的, 你的资源版本就是可以更新的了.

3. 如果你选择 AssetBundle\_Remote 模式, 你就是从远程地址读取 AssetBundle, 你就不需要做更新系统了, 它总是读取最新的资源(从本地或者远程地址, 缓存到本地).