**Resources:(尽量避免使用)**

1、资源文件夹的使用使得细粒度的内存管理变得更加困难。

2、资源文件夹的使用不当会增加应用程序启动时间和构建时间。(随着资源文件夹的数量的增加，这些文件夹中资产的管理变得非常困难。)

3、资源系统降低了项目将定制内容传送到特定平台的能力，并消除了增量内容升级的可能性。(AssetBundle变体是Unity用于根据设备调整内容的主要工具。)

4、Resources在快速原型设计和实验过程使用。但是，当项目进入全面生产阶段时，强烈建议不要使用“资源”文件夹。

5、**适用于Resources的情况如下**:

存储在资源文件夹中的内容不是内存密集型的

内容通常在项目的整个生命周期中都是需要的

内容很少需要修补

内容不会跨平台或设备变化。

全局使用的单例的MonoBehaviours预制体或托管第三方配置数据的资产，例如Facebook App ID

Resources文件夹的特点：

1、只读，即不能动态修改。所以想要动态更新的资源不要放在这里。

2、会将文件夹内的资源打包集成到.asset文件里面。因此建议可以放一些Prefab，因为3、Prefab在打包时会自动过滤掉不需要的资源，有利于减小资源包的大小。

4、主线程加载。

5、资源读取使用Resources.Load()。

**构建项目时Resources的所有文件夹中的资产和对象都合并为一个序列化的文件**。还包含元数据和索引信息(包括用来解决给定对象的名称到其相应的文件GUID和本地ID序列化查找树,也用于在序列化文件的主体中定位特定字节偏移处的对象)，类似于AssetBundle。

查找数据结构（在大多数平台上）是一个平衡搜索树，**随着“资源中的对象”文件夹数量的增加，这种增长也会导致索引的加载时间增长超过线性**。

**资源序列化的不可逆**，在显示初始化非交互式启动画面时在应用程序启动时发生。

AssetBundles是在编辑期间在Unity编辑器中创建的文件，可以在运行时通过构建项目来使用。

**AssetBundles可以包含资产文件，如模型，材质，纹理和场景。**

**注意：AssetBundles不能包含脚本**

按需加载(流式传输和异步加载)

动态地从应用程序中加载和卸载

实现发布后的DLC:首次部署应用程序的磁盘占用空间或大小可以减少，资产在安装应用程序之后加载，只有在需要资产时才加载

可以加载平台和设备特定资产，而无需为其他平台或解决方案下载或存储冗余资产。

本地化:通过根据用户的位置，语言或偏好而仅下载和安装所需的资源。

1、Ab尽量避免全部下载。有的ab包含的资源不是立刻使用，如果可以独立分离就分离，因为下载会占用带宽和磁盘空间

2、Ab不需要全部加载。下载ab后，Assets要从ab中有选择性的加载

3、资源依赖。如 模型资源 依赖于网格数据，而网格数据也依赖于材质资源，材质资源也依赖于纹理资源。**（****注意加载的先后顺序，避免丢失依赖的问题出现）**

4、资源共享依赖**（注意加载的时候不能冗余重复加载依赖关系）**

5、Ab包对项目都是有开销的，ab包的数量上的管理要注意

6、在大量的小型ab包和大量包含冗余数据的ab包的之间平衡

7、**unity构建ab包的时候，ab的内容会针对当前目标平台进行编译和优化，所以每个目标平台(Android、IOS)都要有自己的构建专门的ab，不能通用**。

**Manifest 和 依赖管理**：

1、资产依赖性永远不会丢失。

资源冗余：

如果在构建AssetBundles时，相关资产尚未分配给任何AssetBundle，则相关资产将与所选资产一起自动添加到AssetBundle中，防止从属资产的损失。但是，这也会导致资产冗余重复。如两个模型共享相同材质，材质没有打包进明确的ab包，两个模型在不同的ab，这时共享的材质会有2份分别存储进两个模型各自的ab中。

2、Manifest存储所有ab的依赖关系还有其他信息存储

制作编辑器工具，对ab进行分析

3、ab变体：

特定用例，将不同资产的选择(分辨率，语言，本地化或用户偏好等)重新映射到项目中的单个对象。AssetBundle变体可以保存所需的各种资产，以涵盖对象的所有支持的选项，并且可以根据需要将所需的资产映射到所选的AssetBundle变体。AssetBundles可以根据用户的偏好将Variant映射到场景中的对象

**md5检测ab资源发生变化 ——> 构建ab包 ——> 上传ab包到服务器 ——> 客户端运行下载md5差异性的ab包 ——> 从新的ab包加载新的对象**

注意：一些AssetBundles可以作为默认设置存储在本地，以便立即加载。当无法访问可下载内容时将从本地AssetBundle加载默认语言和本地化数据。

AssetBundle名称是严格小写的

ab变体:

根据设备，位置或用户偏好(分辨率，语言，本地化或用户偏好等)将不同的资产集加载到项目中

提供不同版本的相同资产以分配给场景中的对象，将不同的资产完全重新映射到同一个对象

只有一个AssetBundle的变体在任何时候都被加载

可以使用不同内容创建资源变体：对于每种受支持的语言，区域或主题，文本，图像，纹理和字体可以不同。这些资产被保存在一系列相同构建的资产包中，并通过变体名称进行标识。

为了兼容AssetBundle变体，AssetBundle的文件夹结构和内容必须匹配。

父目录已分配给AssetBundle，且没有任何子项已分配给AssetBundle，因此所有子项都将在构建时添加到父项的AssetBundle中

为AssetBundle名称创建分层菜单结构

AB打包

--全量：第一次全包，相当于给所有的资源文件分配归属AB

--增量：不仅要对新增的文件分配归属AB，还要把已经删除的文件从AB里面移除，最后确定修改的AB文件列表

**注意：增量包存在不一致性的问题**

对于新创建的资源，由于guid发生变化会导致增量包不一致。对于Prefab和Scene资源，则是尽管Object并没有发生变化，但是Object与Object之间的顺序会发生变化，这也会导致增量包不一致。

**Unity没有解决资源一致性这个问题**。这里通过**自己保存资源以及维护资源修改列表**来解决这个问题。顺便这个做法还能节省很多的构建时间，做**增量预处理**，一举两得。

**打包日志log文件可以分析打包出错问题**

AssetBundle的结果是：**减小内存峰值和加快加载速度**

1、**根据依赖树**进行的最优打包策略（这里不考虑加载因素），**公共资源单独打ab**，**独立资源打到一起**，**再将shader字体等其他细碎并且需要常驻内存的资源打包到一起，启动游戏的时候常驻内存**。

2、**打ab的目的是进行资源的增量更新**，如果规划的不好就会出现我本来只更新小部分资源（几十k），但是更新ab的时候需要更新几m，在加载ab的时候也是需要进行规划和及时的卸载，这些都需要根据项目的具体需求进行具体分析。

1、项目一开始**按经验（考虑资源量、加载频率、同时加载量、更新频率等）规划好目录结构**，

2、然后通过配置对不同目录做不同的合并规则。

3、这样功能上是通用的，**通过配置来对不同项目不同需求进行差异化**

**将需要打成包的资源放在同一个目录，弄个配置表格来做合并规则**

自己维护一个配置文件，**用正则表达式来分类匹配，这样只要满足规则的资源都会归到一类包中，顺便还能对一些bundle做特殊的标记**。

这个分包配置可以分为两部分，**一部分是手动配置，一部分是用代码生成的配置**，两者相互补充。

手动配置如下：

格式：  
资源路径的正则表达式 => 输出包名 : [options] //输出包名中的格式化参数是从正则表达式中捕获的字符串拼成  
可选参数比如：immortal 可标记该bundle常驻内存等等， bigbundle 大包标记

匹配规则：  
从上到下匹配：如果满足则返回，也就是配置文件中，上面的匹配优先级最高；  
从父到子匹配：如果一个匹配有子匹配，那么继续递归匹配到满足为止

例子：  
//====================================================================  
// 场景相关资源  
//====================================================================  
Assets/(map\d+|Login)/.+ => {0}\_level\_assets : bigbundle // 标记这个包可以超过2M，不标记则会按序号拆开  
Assets/GameAssets/LightMaps/(map\d+)/.+ => {0}\_level\_assets : bigbundle  
{  
//这里可以加上满足父节点匹配后，继续细分的子匹配规则  
}  
//====================================================================  
// shaders 把 工程中的shader分成两个常驻内存的包  
//====================================================================  
Assets/Resources/Shaders/(.+/)?.+ => main\_shaders : immortal  
Assets/Editor/External/NGUI/Resources/Shaders/.+.shader => NGUI\_default\_shaders : explicit | immortal  
//===================================================================

下面是自动生成根据依赖分析产生的匹配文件，把那些独立的，较少公用的资源归到一起，注意公用的shader资源不在此匹配文件中，因为已经在手动配置中设置好了

Assets/Resources/Prefab/role/player\_3/player\_3\_2.prefab => player\_3\_2\_prefab  
Assets/GameAssets/role/player\_3/player\_3\_2.mat => player\_3\_2\_prefab  
Assets/GameAssets/role/player\_3/player\_3\_2.png => player\_3\_2\_prefab  
Assets/GameAssets/role/player\_3/player\_3\_2.FBX => player\_3\_2\_prefab

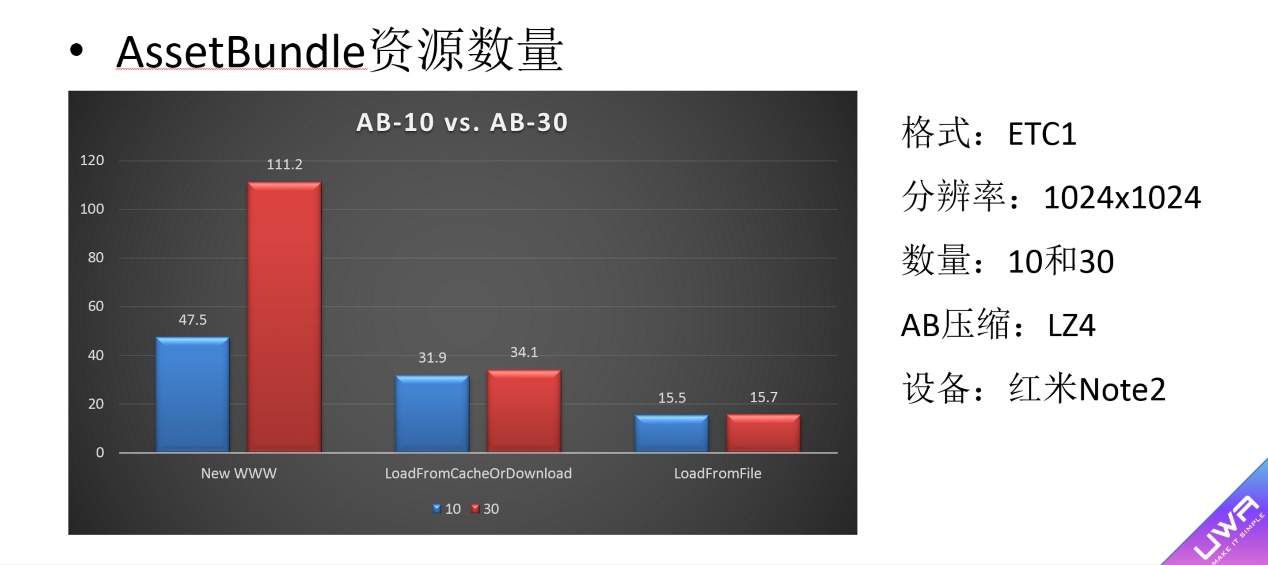
就是这样，**项目文件夹结构稳定后，这个配置文件基本不用变了**。而且即使要变也很方便。

1、没有最好的打包方式，只有最适合项目需求的打包方式

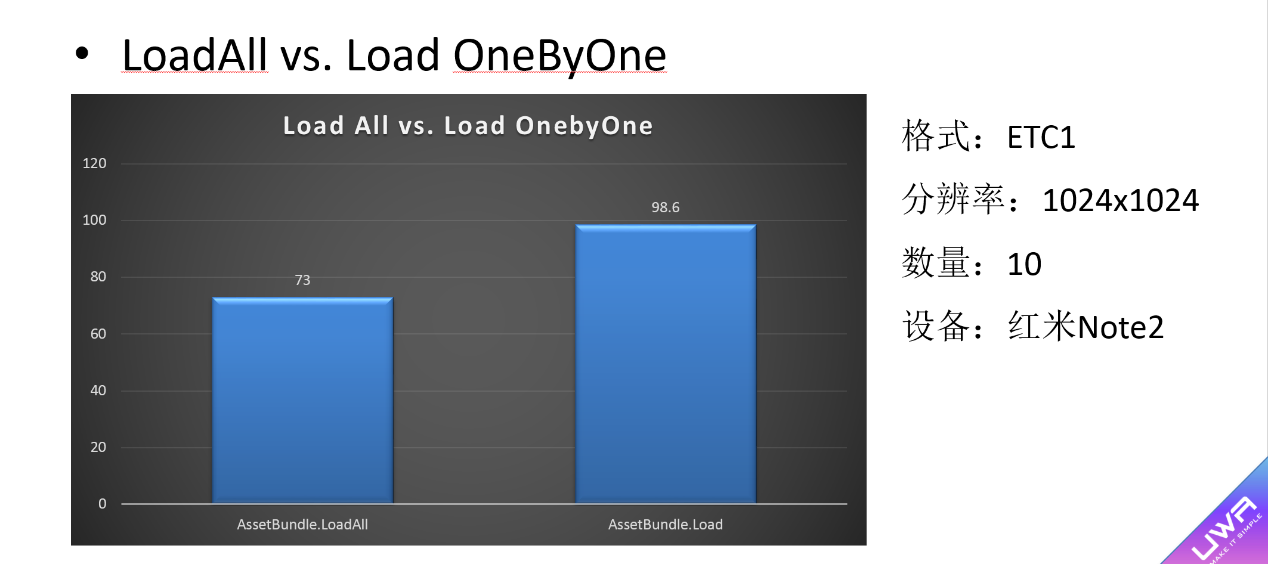
2、无论是粗粒度打包还是细粒度打包，现在都有成功的项目在采用，这说明两种打包方式都是很好的

1、**AssetBundle粒度建议不宜过细**，特别是一个资源一个AB。粒度过细，一方面会导致加载IO次数过多，从而增大了硬件设备耗能和发热的压力；另一方面，在我们测试过的Unity 5.3 ~ 5.5 版本中，Android平台上在不Unload的情况下，每个AssetBundle的加载，其每个文件的SerializedFile内存占用均为512KB（远高于其他平台），所以当内存中贮存了大量AssetBundle时，其SerializedFile的内存占用将会非常巨大同时。同时，需要进一步说明的是，该问题已在Unity5.6中进行完善。

2、在Unity5.3版本之后，**对于AssetBundle文件的文件大小其实不必再限定于1MB之内**。之前有这一限定是出于两方面的考虑：一是New WWW在Unity 5.3之前是使用最为频繁的AssetBundle加载方式（虽然现在也是…），**在5.3版本之前，New WWW加载会形成一个比较大的WebStream，一般来说是压缩AB的4~5倍，会占用比较高的内存**；另一方面，当AB较大（比如大于5MB）时，其加载开销会很大，由于是子线程中运行，所以真正反映到Profiler中时，大家会看到的一个“诡异”的CPU高开销——Graphics.PresentAndSync，不知道大家是否有注意到过，不过具体原因可以看这里（ [扒一扒Profiler中这几个“占坑鬼”](https://blog.uwa4d.com/archives/presentandsync.html) ）。所以，我们对此做了一些实验，发现将AB压在1MB以下是一个加载比较可以接受的情况。以上是我们之前为什么会提出1MB的主要原因。但Unity5.3之后，随着LZ4的引入，很多情况已经变化了，基于其Chunk的加载特点，AB加载很快，且内存占用要比之前小很多。所以**LZ4的AB其实可以考虑更加粗粒度一些（可大于2M）**。  
但是，这里**仍然有以下三点注意**：  
(1) **对于需要热更新的AB需要考虑实际情况控制AB的大小**；  
(2) 即便是LZ4的AssetBundle，其加载方式不同，加载效率也可能完全不一致。以下是我们在UWA Day 2017的分享，我们在两个不同的AB（LZ4格式）中都加载同一个资源，唯一的不同是一个AB包含10个Asset，而另一个AB包含30个资源，以下是三种不同方式从加载AB到AB.Load的耗时对比。可以看出，New WWW加载出现了明显的时间差异。因此，**在Unity5.3之后，尽可能建议通过****LoadFromFile(Async)来对AB进行加载**。



(3) **对于AB的打包，尽可能把逻辑上同时出现（一个预设中非共享的资源）、小而细碎的资源（Shader、Material、粒子系统等）尽可能打包在一起，并通过LoadAll来进行加载**，因为这样会带来更好的加载效率。下图为**LoadAll**和Load One By One的性能对比。在我们做过的实验中，**LoadAll确实会带来更好的性能开销**。



为了避免资源重复打包，我们把所有的资源进行分包处理，而不是由unity自动关联其依赖的资源

1、遍历所有需要打包的资源，根据正则表达式将其分入不同的分组，没有设定规则的文件则使用其文件名，作为一个单独的分组。

常用的分包规则有：Resources中的UI预设归为一个分组，所有的Shader放在一个分组，Resources中的配置文件放在一个分组。

2、通过 AssetDatabase.GetDependencies 生成分组的有向图，每个分组是有向图的顶点，依赖关系为其入度关系，如A依赖B，则B->A。

注意，要检查生成的有向图中不能存在环，如果存在则修改第一步中的正则表达式。

3、寻找顶点间的最长路径，删除非最长路径，如 {C->B->A, C->A}，则删除{C->A}，只保留{C->B->A}。

4、寻找出度为1的顶点，根据其是否接受出度合并 及 其出度顶点是否接受入度合并的规则，进行合并。

{C->B->A, C->B->D},C允许出度合并，B允许入度合并，则合并为 {CB->A, CB->D}

5、对于存在禁止入度合并配置，使用虚拟合并的方式（只合并有向图，不合并资源）进行优化。

{C->A, B->A} 如果A禁止入度合并，可将其合并为{CB->A}

6、观察输出的调试信息，并使用XDot生成有向图，继续完善正则表达式。

(调试命令 dot -Tpng -O 1.txt)

资源热修复：

ZGame ABSys：

1. LZ4和LZMA：
   1. LZ4 体积比 LZMA 大，但是加载AB的耗时和内存都比LZMA要小很多。
   2. ZGame本身是ARPG大地图类型游戏，一方面内存容易崩溃因此需要做好控制，另一方面现在的游戏大小已经出现3G大小的游戏，采用LZ4更合适。
   3. U3d内置的LZMA压缩本身应该不是7zip的最新压缩算法，如果使用了7zip作为压缩，则需要将7zip植入到打包程序中。
2. 编辑器代码设置AssetBundleName，通过AssetBundle Brower 插件可以看到哪些资源有冗余。方便测试。
3. 打包后可以使用 AssetBundleExtractor 工具查看是否存在冗余的AB资源。
4. 客户端代码开放接口，模拟AB读取，接口在编辑器模式下会直接读取指定目录下的AB作为测试。编辑器模式下测试保证AB正常加载与显示
5. 打包规则：
   1. 依赖关系打包
   2. 正则表达式匹配规则（怎么写？）
   3. BuildAssetBundleOptions.UncompressedAssetBundle（AssetBundle.LoadFromFile / AssetBundle.LoadFromFileAsync读取的是解压后的AB文件，所以需要设置为未压缩模式）
6. 打包结束，对每个AB包生成对应的MD5还有对应的AB包的大小，存放到JSON文件中。AB包的名称为MD5值。
7. JSON文件上传到CDN服务器上：

{

"1.1.0": [

{

“name”:“AB包名”

"md5": "MD5值1",

"size": "100868399"

},

{

“name”:“AB包名2”

"md5": "MD5值2",

"size": "10086839934"

},

{

“name”:“AB包名3”

"md5": "MD5值3",

"size": "1008684848"

}

]

}

1. 增量打包，客户端每次打包的时候会对比JSON文件，根据MD5比较得到需要更新的AB包，如果新打出来的AB包的名字不在JSON中或者AB包对应的MD5值不一样，则表示这个AB包是新的需要增量出去的。
2. 根据版本号生成对应的AB包，AB包存放在对应的版本号文件夹下，将记录版本号还有MD5的JSON文件还有存放AB包版本文件夹压缩成一个zip包，压缩包放到CDN服务器上然后在WinSCP上直接解压缩，解压成功后需要对整个文件夹的大小查看防止解压出来的大小有错误。
3. 增量更新：客户端进入游戏，先从CDN服务器拉取配置文件，里面记录了版本号还有打包出来的存放AB的文件夹的大小，发现版本号有更新（检测版本号，如果主版本号和次版本号一致，但是修订版本号发生差异，则取到差异的一个或者多个修复版本号然后将所有的增量版本对应的AB下载下来并且检测），就从CDN上下载对应的版本对应的差异的AssetBundle包，下载成功后需要做AB文件大小的检测，防止文件在传输的过程中缺漏，如果有缺漏就重新下载单个AB包
4. 网上下载的AB包放在哪个目录？
5. AB包解压保存在Application.persistentDataPath目录下？
6. 怎么删除已经下载下来的发生下载错误的AB包
7. 解压成功后 删除原来的AB下载压缩包
8. 新的AB怎么替换原来错误的AB从而让玩家运行最新的AB包？版本号控制？
9. 选择同步/异步加载？
10. 加载方式：
    1. 从本地文件加载 AssetBundle.LoadFromFile / AssetBundle.LoadFromFileAsync。读取已经解压后的AB文件
    2. 从内存（二进制数据）加载 LoadFromMemory/ LoadFromMemoryAsync
    3. 从缓存加载 [WWW.LoadFromCacheOrDownload](http://WWW.LoadFromCacheOrDownload)
    4. UnityWebRequest加载 UnityWebRequest.GetAssetBundle
11. Android的APP版本号规范：
    1. 主版本号.次版本号.修订版本号

范例：2.0.0

主版本号：接口做了不支持向下兼容的更改，升级主版本号；

次版本号：做了功能性升级，支持向下兼容，老版本依然正常使用；

修订号：做问题修正，通常修复缺陷

一般不变化的放入Resources（不支持热更新的资源并且不能太大）

Font字体、

文本数据表、

UI预设、

Loading的Texture、

可能经常发生变化的资源：放入Resources\_AB

独立的数值配置表（数值发生变化，调高or降低难度）、

场景（场景优化之类）、

图集（图片资源可能发生变化、如文本图片之类的）、

模型资源（模型修改之类）、

Shader、

音频、

Texture贴图

资源路径介绍：

Unity3D中的资源路径

|  |  |
| --- | --- |
| Application.dataPath | 此属性用于返回程序的数据文件所在文件夹的路径。例如在Editor中就是Assets了。 |
| Application.streamingAssetsPath | 此属性用于返回流数据的缓存目录，返回路径为相对路径，适合设置一些外部数据文件的路径。 |
| Application.persistentDataPath | 此属性用于返回一个持久化数据存储目录的路径，可以在此路径下存储一些持久化的数据文件。 |
| Application.temporaryCachePath | 此属性用于返回一个临时数据的缓存目录。 |

android平台

|  |  |
| --- | --- |
| Application.dataPath | /data/app/xxx.xxx.xxx.apk |
| Application.streamingAssetsPath | jar:file:///data/app/xxx.xxx.xxx.apk/!/assets |
| Application.persistentDataPath | /data/data/xxx.xxx.xxx/files |
| Application.temporaryCachePath | /data/data/xxx.xxx.xxx/cache |

IOS平台

|  |  |
| --- | --- |
| Application.dataPath | Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data |
| Application.streamingAssetsPath | Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data/Raw |
| Application.persistentDataPath | Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Documents |
| Application.temporaryCachePath | Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Library/Caches |

dataPath和streamingAssetsPath的路径位置一般是相对程序的安装目录位置

persistentDataPath和temporaryCachePath的路径位置一般是相对所在系统的固定位置

**StreamingAsset文件夹：**

Resources文件夹中的内容在打包时会被压缩和加密。而StreamingAsset文件夹中的内容则会原封不动的打入包中，因此StreamingAssets主要用来存放一些二进制文件。下面也同样1、跟Resources文件夹一样只读不可写。

2、主要用来存放二进制文件。

3、只能用过WWW类来读取。

**PersistentDataPath**：

这个路径下是可读写。而且在IOS上就是应用程序的沙盒，但是在Android可以是程序的沙盒，也可以是sdcard。并且在Android打包的时候，ProjectSetting页面有一个选项Write Access，可以设置它的路径是沙盒还是sdcard。

下面同样简单的总结一下：

1. 内容可读写，不过只能运行时才能写入或者读取。
2. 提前将数据存入这个路径是不可行的。
3. 无内容限制。你可以从StreamingAsset中读取二进制文件或者从AssetBundle读取文件来写入PersistentDataPath中。
4. 写下的文件，可以在电脑上查看。同样也可以清掉。

该目录为应用程序的沙盒目录，应用程序安装后才会出现。该目录独特之处在于是可写的，所以我们**一般将下载的AssetBundle存放于此**。使用Application.persistentDataPath访问。

各平台PersistentDataPath路径打印：   
　　**Win**：C:/Users/lodypig/Appdata/LocalLow/myCompany/myProj   
　　**Mac** : /Users/lodypig/Library/Application Support/myCompany/myProj   
　　**Andorid**：/data/data/com.myCompany.myProj/files

**iOS**:/var/mobile/Containers/Data/Appliction/A112252D-6B0E-459Z-9D49-CD3EAC6D47D/Documents

1. 协程
2. LoadFromFile
3. AB的引用计数是会在加载AB的时候用到，主要是为了我们卸载资源，因为如果我们在加载完AB包之后没有做引用计数的话，如果A依赖了B，C也依赖了B，A与C同时加载到场景中，如果没有做引用计数，卸载A的时候会把B也卸载掉，那么C就会出现资源丢失的情况；此外还会出现内存泄露的情况，所以我们需要去维护一个引用计数，来保证正常的卸载