[**http://www.bubuko.com/infodetail-1680082.html**](http://www.bubuko.com/infodetail-1680082.html)

**Assets和Object**

Asset是存储在硬盘上的文件，保存在unity项目内的Assets 文件夹中。比如：纹理贴图、材质和fbx都是Assets。一些Assets以Unity原生格式保存数据，例如材质。另一些Assets需要通过处理转换到原生格式，比如fbx

Object是一系列序列化数据，这些数据描述了具体的资源实例，这可以说Unity使用的任意类型的资源，比如Mesh、sprite、audio clip或animation clip，所有的Object都是UnityEngine.Object的子类

大部分Object；类型都是Unity内置的，但有两个特殊类型：

1 scriptableObject允许开发者定义他们自己的数据类型，这些类型能够由unity序列化和反序列化，并且在编辑器的Inspector窗口中进行操作

2 MonoBehaviour提供了链接都MonoScript的封装。MonoScript是Unity的内部数据类型，其中保存了指向在具体的程序集和命名空间中的具体脚本类的引用。MonoScript不包含任何实际可执行的代码

Assets和Object之间的关系存在一对多的关系：也就是说Asset文件内能够包含一个或多个Objects

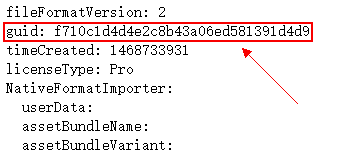
**内部对象引用**

所有的UnityEngine.Obejct都可以应用其他的UnityEngine.Object，被引用的Object可以和引用的Object位于同一个Asset文件中，也可以是由其他Asset文件导入的。例如，材质对象通常有一个或多个纹理对象的引用，这些纹理对象通常都是从纹理资源文件导入的（例如png或jpg）

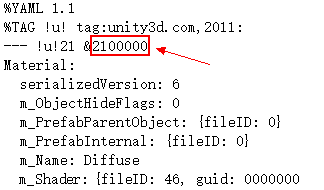
当序列化的时候，这些对象由令部分分离的数据组成：文件的GUID和Local ID。文件的GUID标记了存储资源的Asset文件。Local ID是局部唯一的（也就是说，在每个Asset文件中，Local ID都是唯一的），标记了Asset文件中的每个Object

文件的GUID存储在.meta文件中，这些.meta文件时Unity第一次导入Asset时候生成的，并且和Asset存储在同一个目录中，下面展示了Diffuse材质及其.meta文件

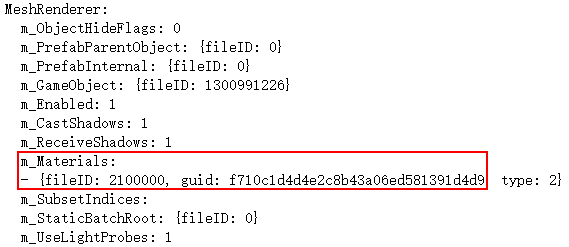
.meta文件包含了GUID：



打开材质文件本身，可以看到localID



如果在场景中有对象使用该材质进行渲染，那么打开场景文件后，就会发现该材质对象由GUID以及Local ID来标记



**为什么使用GUID和Local ID？**

GUID的功能时提供文件路径的抽象表示。只要使用GUID来关联具体的文件，那么文件在磁盘上的位置就无关紧要了。因此可以随意移动文件二部需要更新应用该文件的Objects（因为这些Objects存储的都是文件的GUID）

由于一个Asset文件可能包含多个UnityEngine.Object资源，因此需要用Local ID来明确的标记每个不同的Object

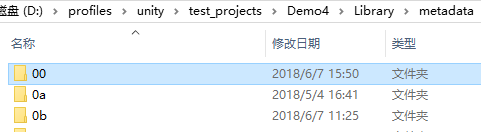
如果一个Asset文件关联的GUID丢失的话，那么所有对该Asset文件中的Object的引用都将丢失，当.meta文件丢失时，Unity会重新生成，Unity维护了具体文件路径与GUID的映射关系，当一个Asset被加载或导入时，就会新增一个映射项，该映射项将Asset的文件路径和Asset文件的GUID连接在一起。如果一个Asset的.meta文件丢失但其文件路径没有发生变化的话，Unity能确保重新生成的.meta中记录的GUID是保持不变的。

如果.meta文件在Unity关闭时丢失，或者Asset文件的路径发生了变化，但.meta没有跟着一起移动的话，那么所有对该Asset文件中的Object的应用都将丢失。

**资源及其导入**

非Unity原生资源必须导入进Unity中才能使用，这是通过asset importer完成的。这些importer在资源导入时候会被自动调用，同时你也可以用AssetImporter及其子类的API通过代码调整资源导入过程。

资源的导入过程包含十分耗时的操作，例如纹理压缩。所以如果每次打开unity都需要执行一遍资源导入过程的话会十分抵消，因此，unity将资源导入的结果缓存在Library文件夹中：具体来说，存储在以Asset文件的GUID前两个数字命名的文件夹中，这些文件夹位于目录Library/metadata



实际上即使是Unity原生资源，也会将导入结果存储在对应文件中，但是原生资源部需要很长的转换时间或重新序列化时间

**实例ID**

尽管GUID和Local ID健壮耐用，但是GUID的比较很耗时，而在运行时我们需要有个十分高效的系统。因此Unity在内部会维护一份缓存，这份缓存将GUID和Local ID转换成独一无二的整数，这些整数倍成为Instance ID，每当有新的Objects添加到缓存中时，Instance ID以简单的单调递增的方式进行赋值。缓存维护了Instance ID，GUID和Local ID（这两个定义了Object的源数据在磁盘的位置）以及Object在内存中的实例（如果Object已经被加载到内存中的话）之间的映射关系。这样UnityEngine.Objects就可以维护相互之间的引用关系。通过Instance ID可以快速找到对应的已加载的Object，如果对应的Object还没有加载，那么就可以通过GUID和Local ID来找到Object的源数据，然后加载相应的Object。

应用程序启动时，项目内置对象（比如场景中使用的对象）的数据以及在Resources文件夹中的对象的数据将被初始化到Instance ID缓存中。当运行时有新的资源被导入（比如通过脚本创建的Texture2D对象），以及当从AssetBundle中加载对象时，就会在缓存中添加Instance ID项。Instance ID只有在被认为已经过时的情况下才会从缓存中删除，这种情况发生在一个AssetBundle被卸载时。当一个AssetBundle被卸载时，除了会导致对应的Instance ID被认为已经过时，Instance ID和GUID以及Local ID之间的映射数据也会被从内存中删除。如果AssetBundle被重新加载的话，那么从该AssetBundle中加载的每一个对象都会创建一个新的Instance ID

**MonoScript**

一个MonoBehaviour包含了一个队MonoScript的引用，而MonoScript仅仅包含了用于定位到一个具体脚本类所需的信息，他们都不包含脚本类的可执行代码

一个MonoScript中包含了三个字符串：一个程序集名，一个类名以及一个命名空间名

当Unity构建项目时，会将Assets文件夹下的所有脚本文件编译到Mono程序集中。具体来说，Unity会为在Assets文件夹中使用的每种不同的编程语言编译一个程序集，并且会将在Assets/Plugins文件夹中的脚本单独编译到一个程序集中。在Assets/Plugins文件夹外的C#脚本会被编译到Assetmbly-CSharp.dll中，在Assets/Plugins文件夹外的Java脚本会被编译到Assembly-UnityScript.dll中，Assets/Plugins中的脚本会被编译到Assembly-CSharp-firstpass.dll中。