# 序列化记录及使用序列化保存场景

该文档为初步学习UE4序列化以及将其应用于自己的小游戏中对场景进行保存时所做的一些总结与记录，只是序列化的九牛一毛，并且肯定会有理解错误和不足的地方，请多指正。

网上都查得到的概念性东西以及偏官话的基础部分，在此都不做累赘，会比较初步的带过。

## 序列化说明

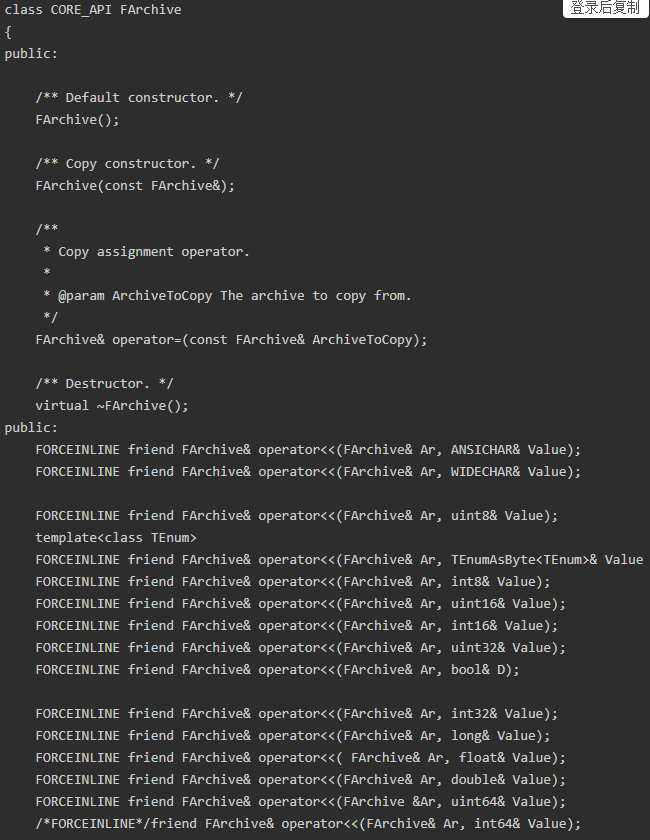
序列化是指将对象转换成字节流，从而存储对象或将对象传输到内存、数据库或文件等的过程。 它的主要用途是保存对象的状态，以便能够在需要时重新创建对象。 反向过程称为“反序列化”。 （通俗来说就是保存和读取的过程分别为序列化和反序列化）

Ue4的序列化使用了访问者模式(Vistor Pattern)，将序列化的存档接口抽象化，其中FArchive为访问者， 其它UObject实现了void Serialize( FArchive& Ar )，接口的类为被访问者。FArchive可以是磁盘文件访问, 内存统计，对象统计等功能。

序列化中最为重要的结构即是FArchive类，该类有众多的继承：

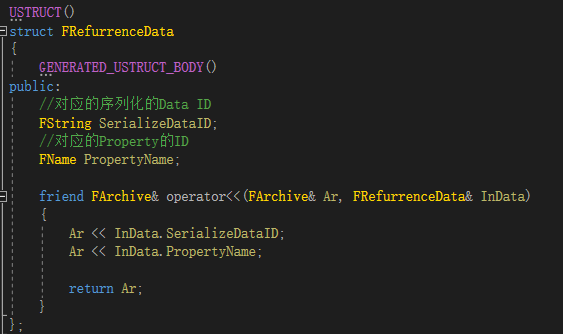


从FArchive的代码中可以看到（该代码只展示了接口），



该类已经实现了一些基本类型的存取，通过重定义<<操作符进行存取。

当我们自定义结构需要序列化时，也必须要重载操作符，例如

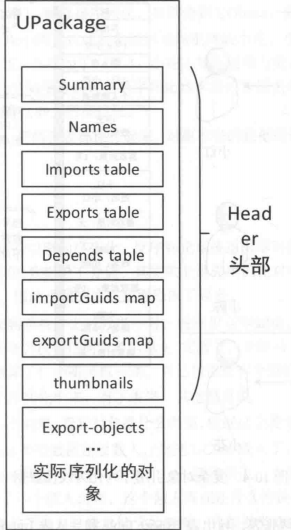


### 初步分析序列化与反序列化

序列化与反序列化都是通过Serialize来进行的，序列化的过程通俗点来说，函数会遍历该UObject上被标识为UPROPERTY()的对象，并将其写为2进制的数据进行存储。

#### UPackage对象简易说明

在我们游戏中生成的蓝图也好，地图也好，都会存储为uasset或是umap格式资源，而每一个资源对应的就是一个UPackage，存储一个Upackage时，会将该资源下对应的所有对象存储到该包中，一个包的结构如下图所示：



File Summary 文件头信息。

Name Table 包中对象的名字表。

Import Table 存放被该包中对象引用的其它包中的对象信息(路径名和类型)。

Export Table 该包中的对象信息(路径名和类型)。

Export Objects 所有Export Table中对象的实际数据。

有时候我们FindObject会出现找不到的情况，就是包没有加载，此时可以使用LoadObject进行包的加载。

#### FlinkerLoad和FlinkerSave

负责将uasset文件中的对象加载到内存中。

#### 序列化反序列化

有了包的概念之后，我们以场景举例，对序列化与反序列化进行分析。

##### 序列化

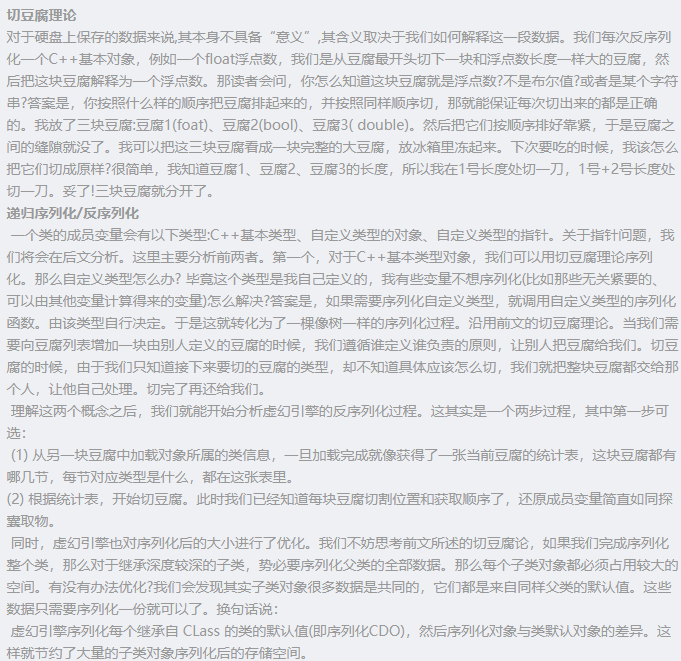
在包最前方有两张表，导出表 Export Table和导入表 Import Table，前者相当于该类自身的属性名单，后者相当于外部引用的属性名单，例如引用的其他类指针等。

当序列化一个对象时，如果遇上UObject指针，不会直接记录地址值，而是记录下一个编号，也就是会对UPROPERTY打上标记，标记在序列化过程中非常重要，他会标记一个属性是否有是其余对象的引用，是否需要存储等等等等。

如果遇到对象为外部的包引用，则会进行记录与标记，存放导ImportTable中。

其余的普通变量，则直接进行序列化后放入包中（存入二进制中）。

这里引用大象无形中的对序列化的一段话：



##### 反序列化

当反序列化时，遍历包中的对象列表，对每一个对象进行反序列化。

先根据GetClass函数获取当前类的信息，通过GetOuter函数获取Outer。这个Outer实际上指定了当前UObject会被当作为哪一个对象的子对象进行序列化。

判断当前等待序列化的对象的类UClass的信息是否被载入，没有的话:

a. 预载入当前类的信息；

b. 预载入当前类的默认对象CDO的信息；

也就是说，会先New一个该类型的空白对象放在那里，接着载入名字，设置类对象信息等等，之后开始载入对象的所有脚本成员变量信息，此时因为对象已经创建出来了，所以可以根据反射查找到需要设置数据的UPROPERTY，对应的函数为

SerializeScriptProperties，序列化在类中定义的对象属性。

当对象信息中碰到了UObject类型的成员变量时，根据包中记录的导出表 Export Table和导入表 Import Table，判断是包内的OJB还是包外的OBJ：

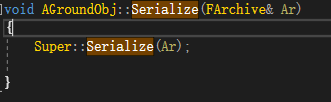
如果是包内的，查找该对象是否已经被初始化，如果已经初始化完成，则直接将指针对应替换，指向已经生成的对象。如果没有，则先New一个对应类型的UObject放在那里，设置好指针，等待该UObject序列化完成。

如果该对象在包外，则判断该包是否已经载入内存，如果没有则把包载入内存，如果已经载入完成，则直接在该包内进行查找并且完成指针替换。

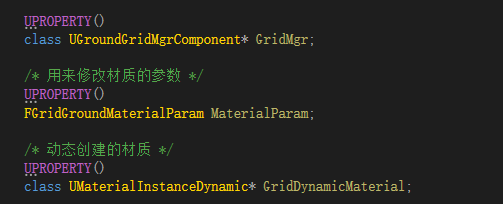
此过程中，会不断的调用各个Object的Serialize函数进行指针替换，最终完成了一个包的反序列化。

#### 实例分析与问题分析

有这样一種情况，我们在自己的代码中对对象进行序列化保存，如下所示：

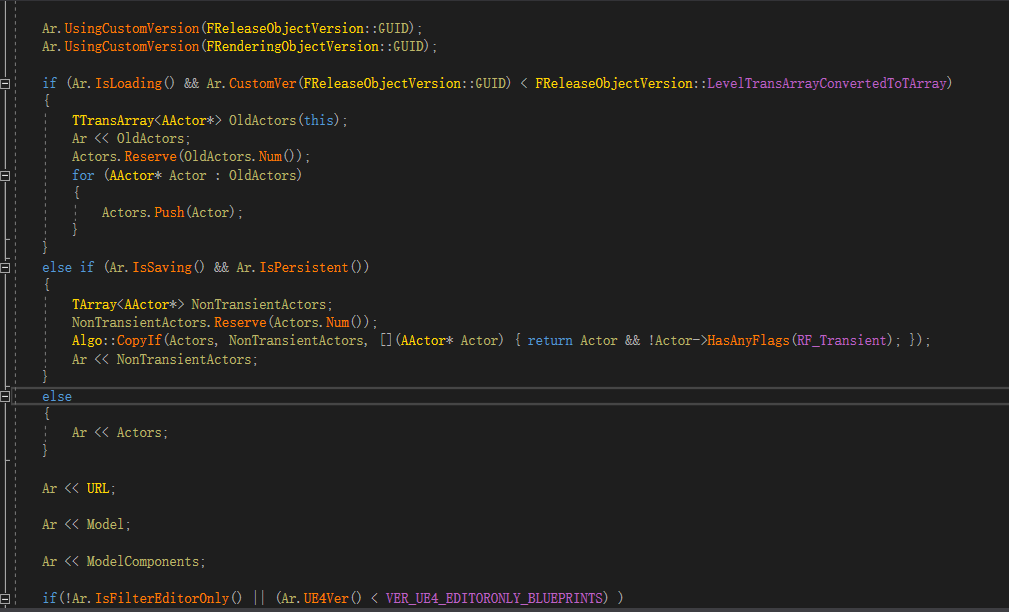


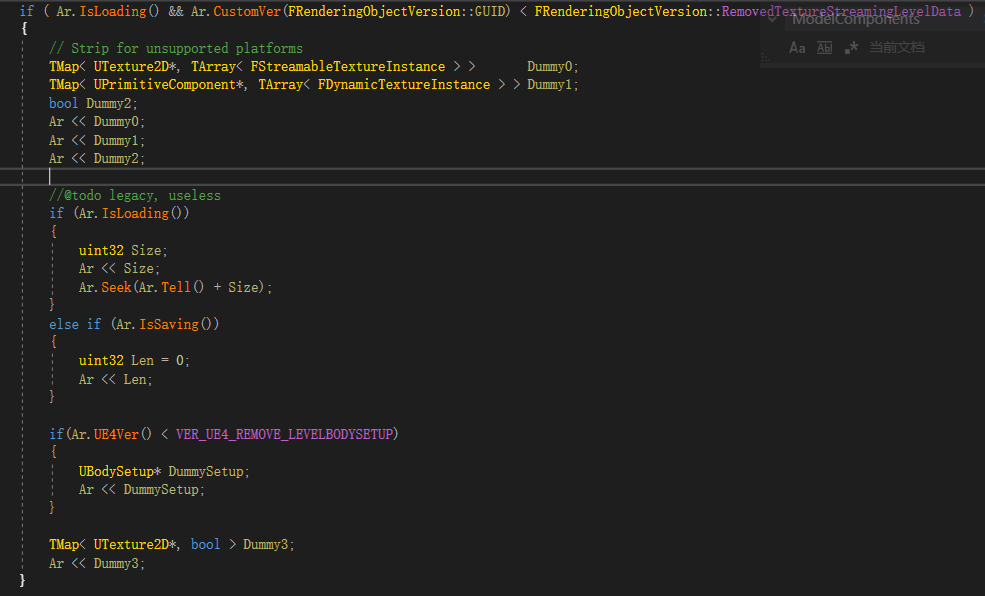
希望能够将我们自定义的一些UPROPERTY进行存储。



其中UGroundGridMgrComponent为我们自定义的组件，此时会发现，UGroundGridMgrComponent 序列化并没有被调用，当然对应的数据与指针也不会被保存，如果重新反序列化，只会得到包含了基本数据的AGroundObj。一些挂载在UObject上的Component也不能被正确的序列化，因为我们并没有对AGroundObj的序列化做出特殊的处理，UE4也并不会自动对该UObject上的所有对象调用其自身的序列化。

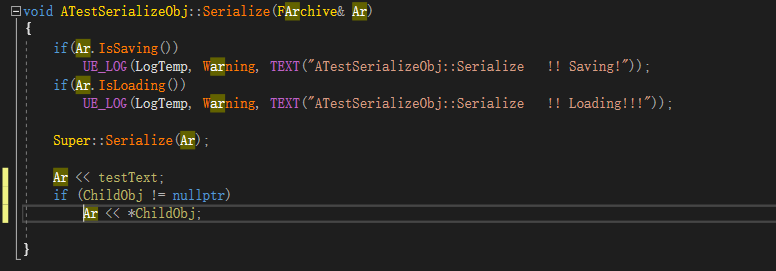
由ULevel的Serialize函数中可以看到，





ULevel中无论是序列化还是反序列化，都是在Super的Serialize之后，对放置上Level上的Actor、Model、ModelComponents进行了存储，**所以关键在于，需要对你想要保存的UObject指针进行单独的存储序列化。**

例如我们以这样的方式：



就能将该Object上存储的ChildObj一起保存，当前前提是实现了ChildObj的operator<<重载。

就算我们手动将对象写入了FArchive中，**反序列化的顺序，对容器结构的影响也是巨大的**

比如定义一个Actor:A，

对于在A上的变量TArray<Object\*> TMap<int32 , Object\*> 类似这样的结构，如果容器中的Object比A先序列化完成，则TArray中该项对应的Object指针能够正确赋值，反之，则TArray、TMap中项的指针会为空。

如果不自行进行对应的处理的话，是不能保证完整序列化出正确数据的。

从ULevel的Serialize保存过程中可以看到，使用了FReplaceReferenceHelper的

FindAndReplaceReferences方法，会对场景中的具有引用的关系做一个保存，也就是后期反序列化时的替换依据了。FArchiveHasReferences::GetAllReferencers 在该函数中，会调用引用的对象的Serialize进行序列化.有互相引用，就会不断互相调序列化，这也是为什么某一个UObject会被不断地调用Serialize的原因。（此处带有个人猜想，若有不对，请指正）

## 我所使用的场景保存方式

### 思路

仿照UE4序列化的理念进行场景序列化的保存。

首先从当前运行时场景中，挑选出我们需要进行保存的对象（我使用的方式是将需要保存的UObject都继承自一个接口）并逐个调用其Serialize(),如果对象有自己的Serialize需求，则自己添加。

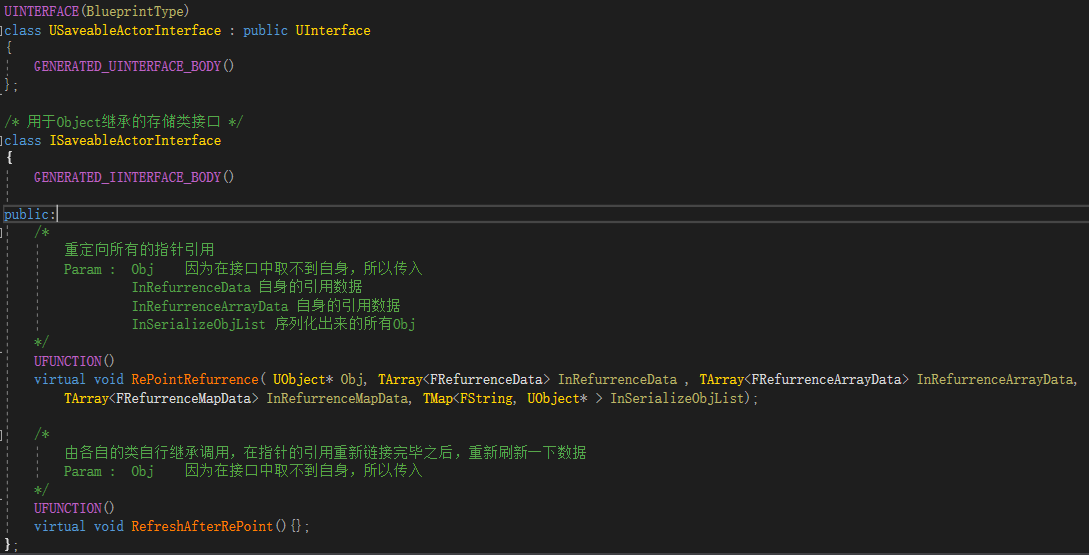
之后遍历对象身上的标记为UPROPERTY的变量，取出其中的我们需要保存的UObject指针变量以及包含了UObject指针对象的TArray，TMap变量，并且对应将其也进行序列化（递归调用该序列化过程）并记录，并且将自身对象的该字段标识为具有外部引用，需要替换指针。当所有的对象的序列化完成之后，**此时二进制文件中不仅包含了所有序列化后的对象，还包含了每个对象的外部引用指针情况，将在反序列化时用来更新指针的引用。**将处理好的二进制文件保存到本地作为我们的存档文件，并且在该存档中加入时间，存档ID，对应地图等需要的数据。

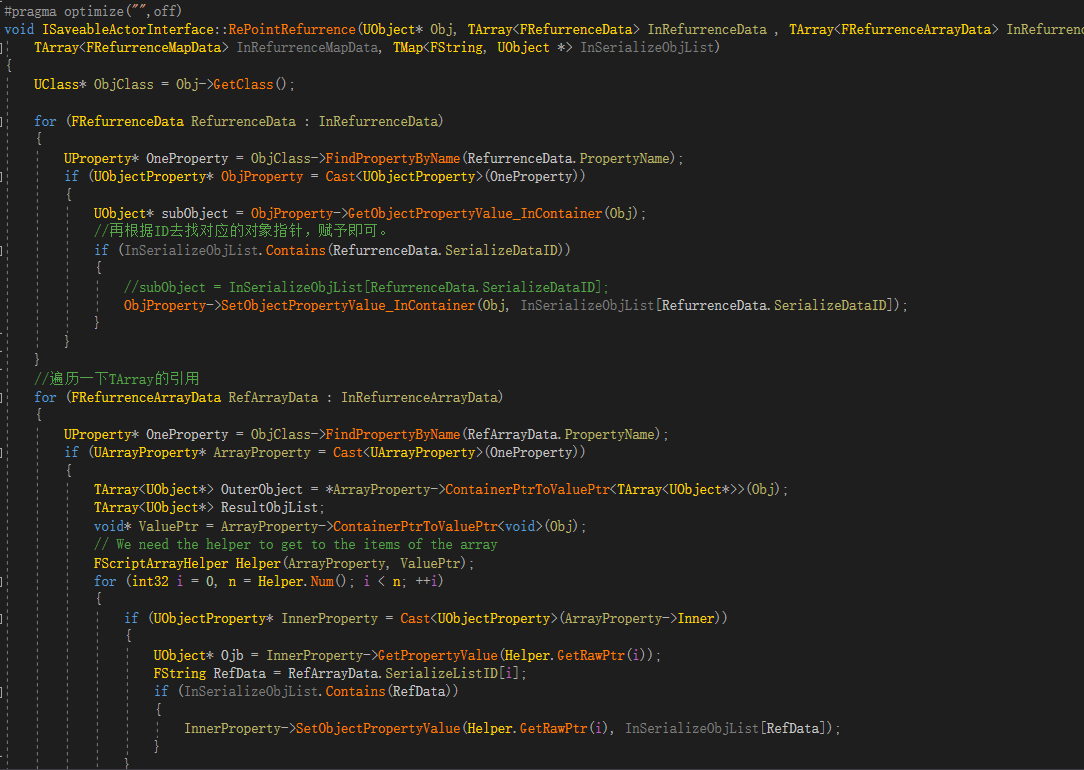
反序列化时，先将存档中保存的所有对象生成并反序列化，再根据存档中的对象变量的外部引用指针情况来进行更新引用关系，就可以很好地完成场景的保存和还原。

### 部分代码以及逻辑说明

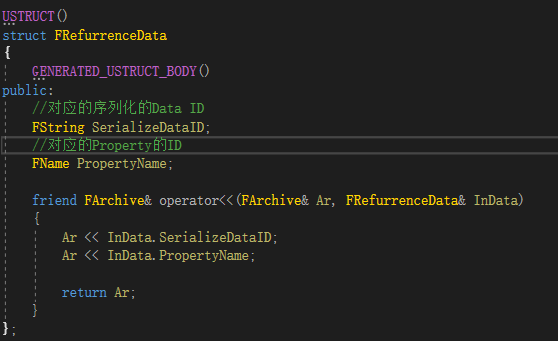
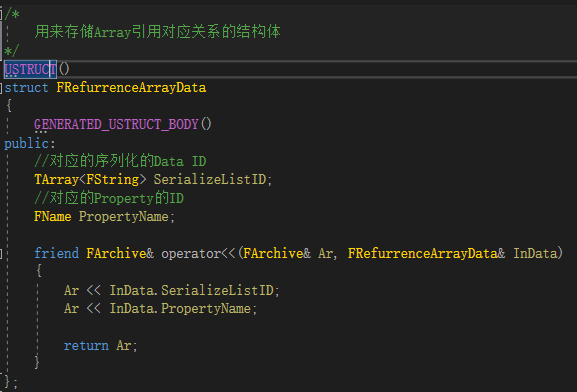
首先定义一个接口，所有在场景中需要保存的对象，都继承自该接口，该接口提供了一个方法为RePointRefurrence，在反序列化时根据对应数据设置自身的指针引用。

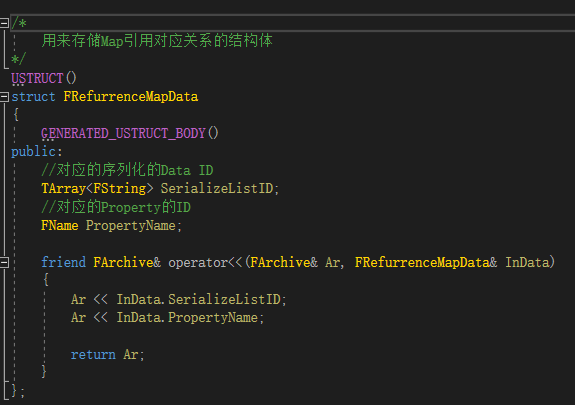
一个方法为RefreshAfterRePoint 再重定义指针后，调用以更新对象中需要更新的数据例如模型等。



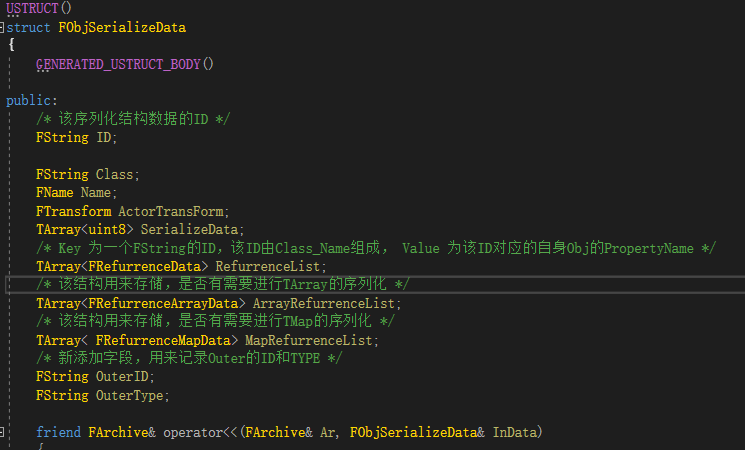


之后需要定义引用关系结构，因为有些变量为指针，需要在反序列化完成之后，根据对应的PropertyName根据反射进行设置对应的指针地址。

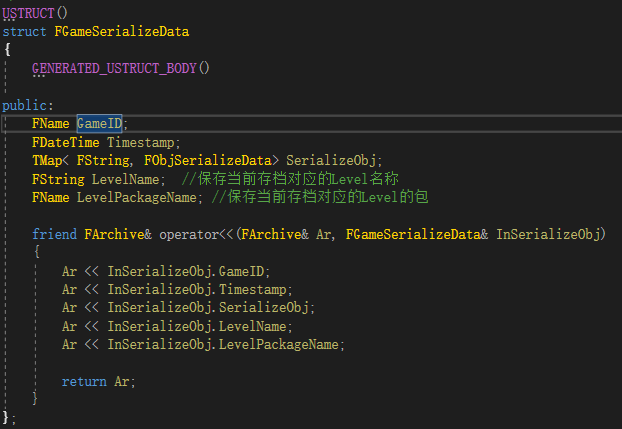
 



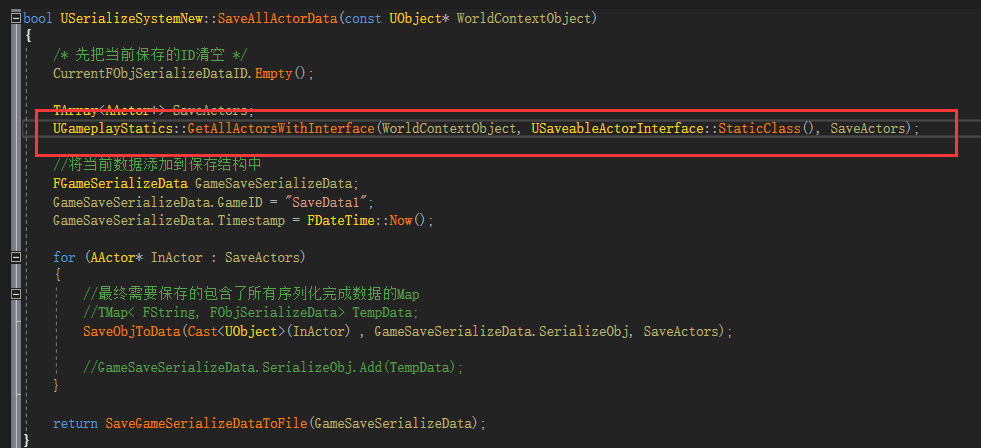
再定义一个序列化后，用来存储每一个Object的Struct



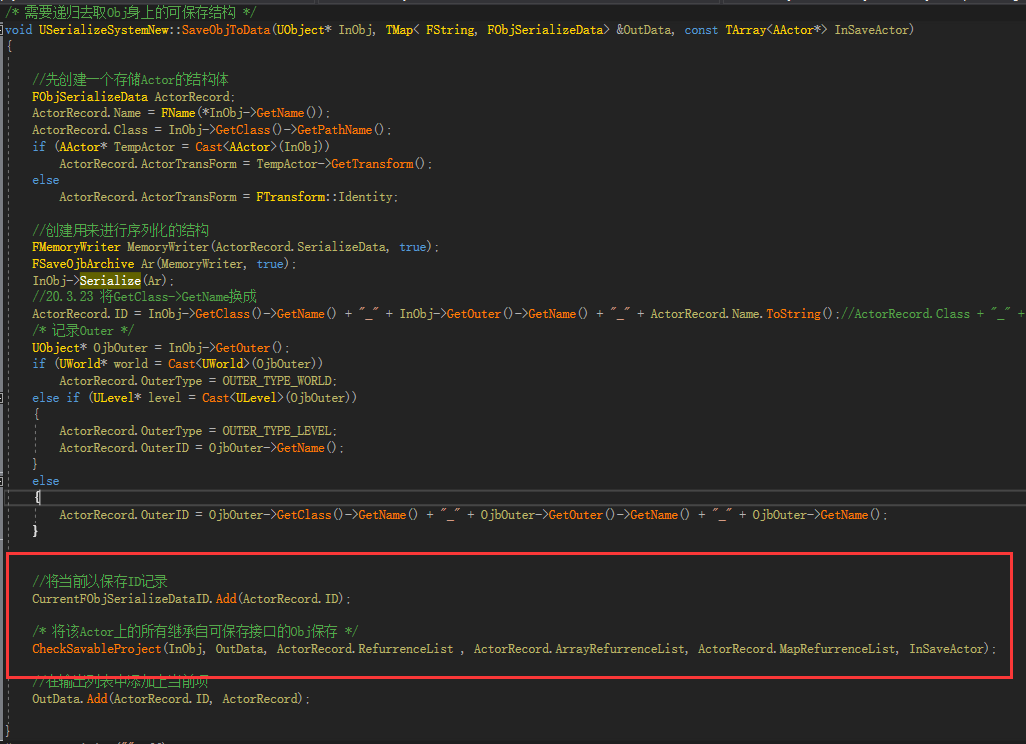
以及存储所有Object的整个的Game存档Struct

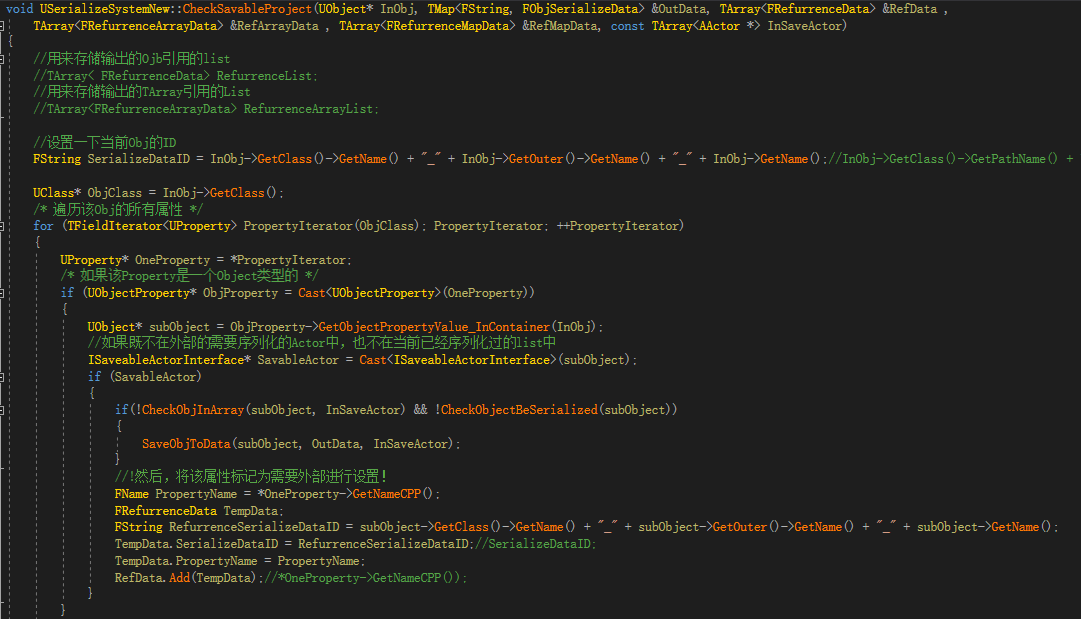


之后进行序列化时,找到具有保存接口的对象：

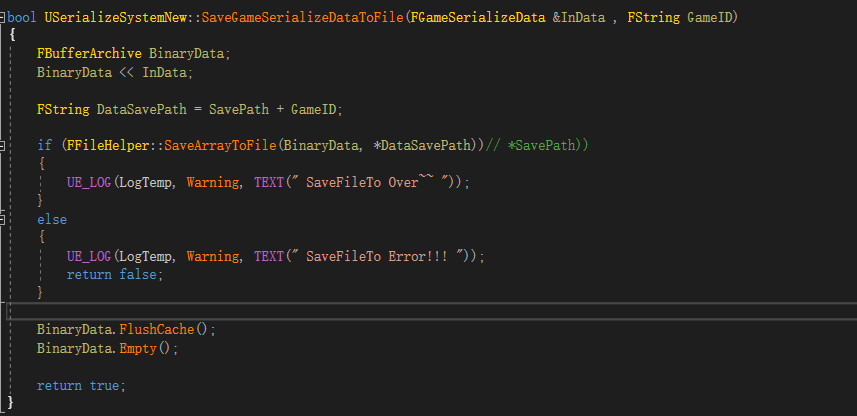


再递归取每一个属性对应的指针引用来更新数据，并且输出引用关系。



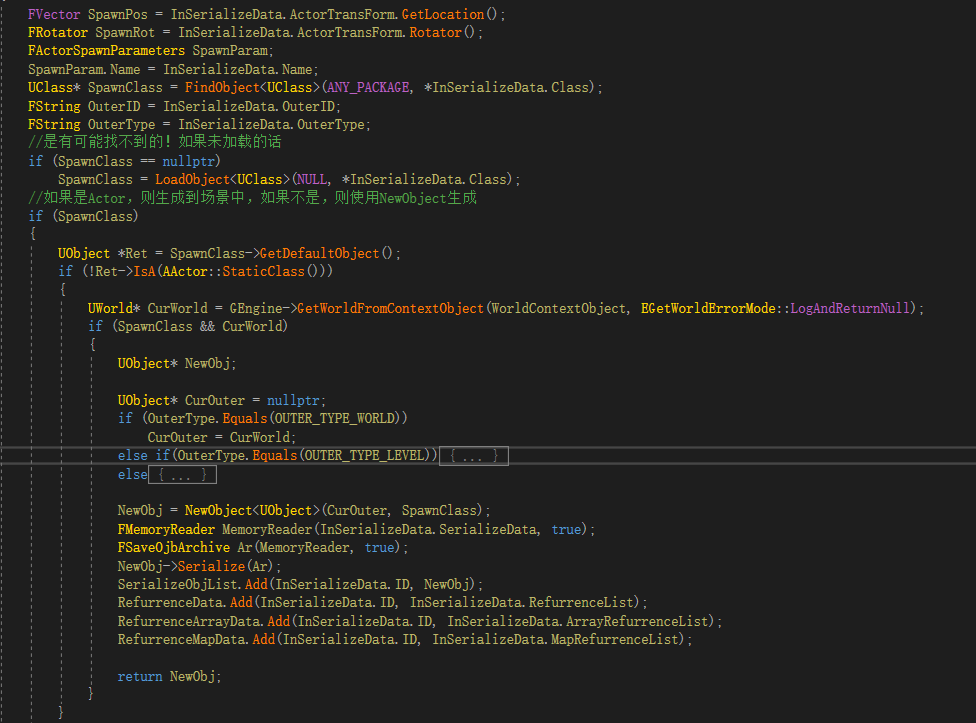
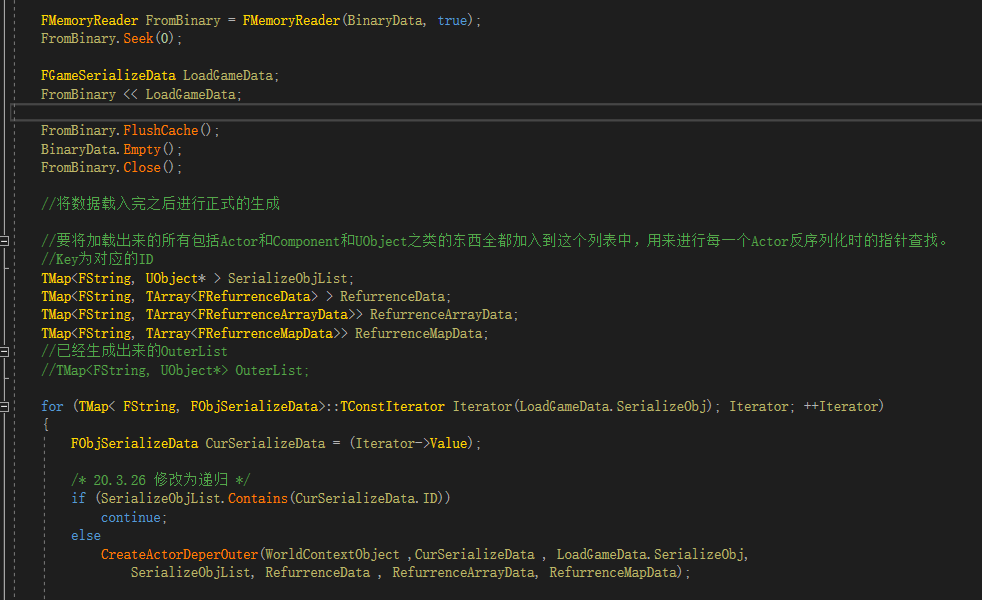


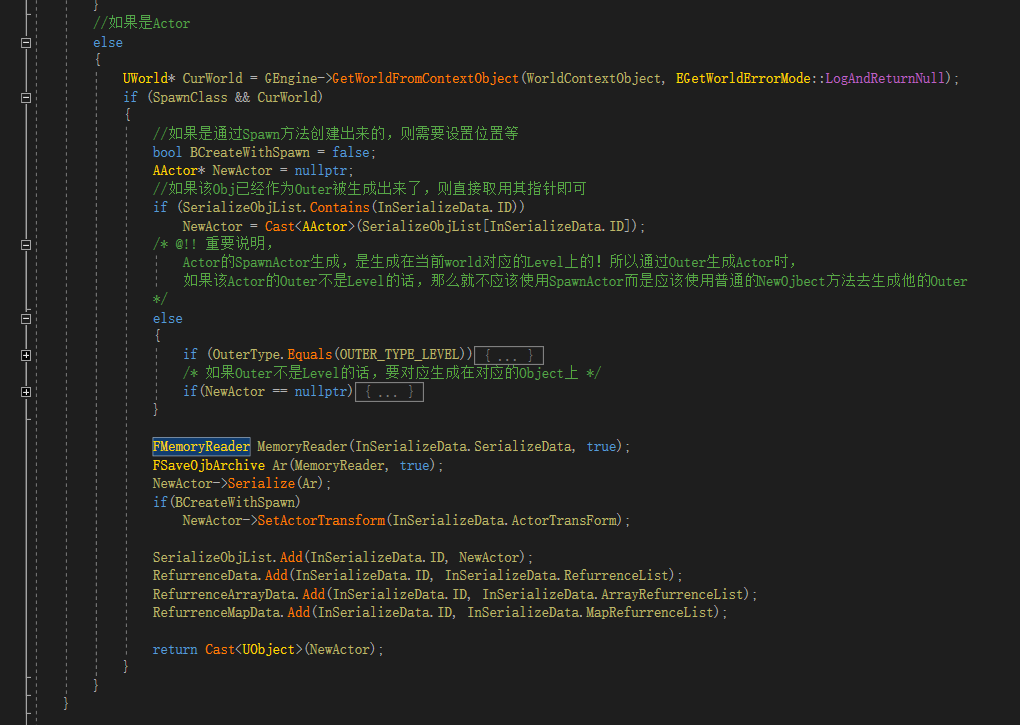
再将处理好的二进制写出到存档即可



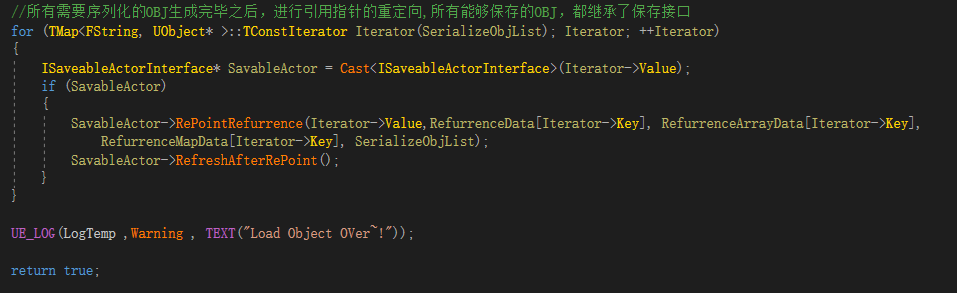
**有一点需要特别注意的是，在序列化对象时，要将其Outer一起做一个保存，反序列化生成时，使用该Outer进行生成，如果该Outer还未反序列化则先反序列化其Outer，这样才能保证原始的从属关系不变。**

反序列化时，读取存档并做递归生成





当所有的对象加载完毕后，重新进行指针的替换即可。



这里有一点不好的地方，是在所有对象生成完毕后遍历所有对象来进行指针的引用替换，复杂度略高，之后可以进行优化。

## 一些提出的思考的问题

#### 为什么不在UObject中通过Serialize函数重写来达到存储其Actor上的指针、组件等的目的？

反序列化Obj需要一个外部的流程来管控，例如在UE的Editor中是使用Level作为基础来进行序列化记录。

假设当前我们有A,B,C三个Obj，且A依赖于B，B依赖于C，C又依赖于A，就算在A,B,C三者中，都响应实现了Serialize的重载，将依赖的变量进行了序列化调用，那么不可避免的需要一个外部的逻辑来控制ABC的加载顺序以及ABC的初始化，设置指针的指向地址,设置Transform等，

那么在一个自定义的游戏中，特别是在Runtime的时候，需要一个具备这样的序列化逻辑的Outer，但是并不能保证这个Outer是所有Actor的父类（Root），则有可能需要多个这样的Outer，那么就有了提出为一个统一模块的需要。

其二是如果将每一个Obj上的指针序列化放在自身的Serialize函数中的话，逻辑复杂且每次增加变量都需要额外添加，特别是对于TArray TMap等容器中包含了指针的情况，更是复杂，且如果有复杂的依赖关系，更不好处理。那么就有了抽象为一个通用接口的必要。