前言

序列化测试一定要注意保存的场景中的Actor，如果修改了序列化的加载/保存函数，则重新打开时会加载数据错误。

## 尝试进行自定义的序列化保存

<https://blog.csdn.net/mohuak/article/details/83027211>

<https://wiki.unrealengine.com/Save_System,_Read_%26_Write_Any_Data_to_Compressed_Binary_Files>

<http://runedegroot.com/saving-and-loading-actor-data-in-unreal-engine-4/#comment-93>

主要是参照第三个

首先定义一个接口，用来进行基础的ActorSaveDataLoaded以及ActorSaveDataSaved();

创建一个Struct用来存储Actor的基本序列化数据

再创建一个整个游戏的用来存储序列化数据的数据结构。

最后，最重要的一步，创建一个继承自FObjectAndNameAsStringProxyArchive的struct用来保存Actor的数据！只有继承自该结构的Ar才可以正确进行Actor的序列化！

UGameplayStatics::GetAllActorsWithInterface(GetWorld(), USaveableActorInterface::StaticClass(), Actors);

可以获取到所有继承了某个接口的Actor

SerializeScriptProperties,跟进查看对于标记UPROPERTY的参数是如何进行反序列化的:

调用SerializeTaggedProperties，序列化对象属性，并且加入tag;

最终跟进是通过 PropertyTag的

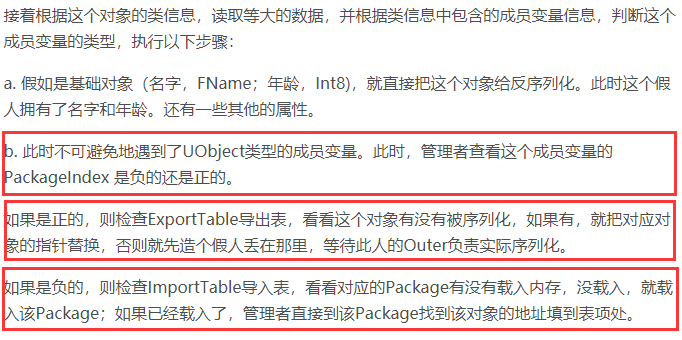
FPropertyTag::SerializeTaggedProperty(FStructuredArchive::FSlot Slot, UProperty\* Property, uint8\* Value, uint8\* Defaults) const

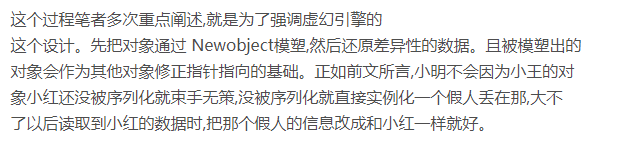
函数，将UProperty（测试时是看的UStaticMeshComponent\* GroundMeshComponet）调用

Property->SerializeItem(Slot, Value, Defaults); 进行初始化。

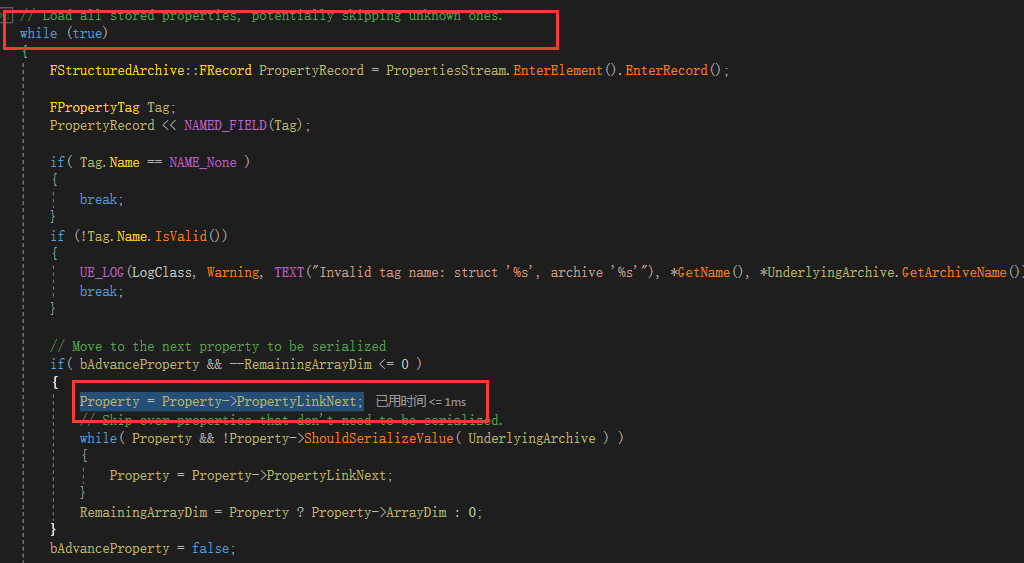
void UObjectProperty::SerializeItem( FStructuredArchive::FSlot Slot, void\* Value, void const\* Defaults ) const

测试时看到，因为之前序列化保存的GroundGridMgrComponent还没有反序列化初始化出来，所以GroundObj中的GroundGridMgrComponent并没有进行完整的赋值，只是创建了一个默认的GroundGridMgrComponent并将指针指向，在正式的GroundGridMgrComponent反序列化之后，才会将该指针指向的结构初始化（还是替换为正确的？）生动的说明在此：





当一个UPROPERTY属性初始化完成之后，



通过While循环会将指针指向Next进行初始化。

如果是UStruct，则会通过

UStructProperty::SerializeItem(FStructuredArchive::FSlot Slot, void\* Value, void const\* Defaults) const

进行初始化

保存时序列化也是走类似的流程。

如果该对象从来没有被加载过，则包内不会出现该对象，此时应该手动加载。

### 关于组件好像并没有跟着Actor序列化的问题：

看一下Level.cpp可以看到，是Level在serialize中对各个Actor做了自己对应的处理，才能让各个Actor保存的那么好。

主题医院的测试中，list类型的序列化，严重依赖于序列化的先后顺序，如果要仿造UE4的结构进行序列化的处理，可以预计会十分复杂，因此在此放弃全部使用序列化存储的方式，改为进行序列化+动态生成动态链接的方式进行处理。

### 猜测：

UE自身的序列化，应该也是多次调用的！先序列化出主Actor，将指针所对应的子Object生成假人放着，当真正的子Object生成了之后，再调用自身的Serialize进行重新序列化！跟源代码看看。

场景从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。

### 初步试验模块成功，可以正常保存以及加载继承自自定义SaveableInterface的Object了！

通过查阅文档发现，ArrayProperty，EnumProperty之类的都是存在的！！那么，就可以对Obj中的TArray等结构进行遍历处理，找到其对应的UObject并且将对应的按照一定方法保存生成refurrence！

在遍历Object的过程中发现，会有UObject的那些Property也被算进来，所以，如果想要序列化那些自己声明的引擎自身的UPROPERTY组件等，

一：自定义一套UPROPERTY的声明规则，或是用自己的Meta或是自定义Category等，在遍历的时候对应单独处理。

二：还是按照原来的处理方式，只对我们自己定义了Saveable的进行存储，可以将原生的类包一层继承我们自己的Object。这样其实更方便更实用。原生的类虽然不继承我们的Saveableinterface，但是还是可以作为正常指针附加到我们的需要引用的指针上。

**类中还是要自定义自己的Serialize后的处理方案，例如反序列化后，重新创建某些组件之类的，因为反序列化，是在OnConstruct以及构造函数之后才调用的！**

设置Property的值，必须使用SetObjectPropertyValue等系列函数，不能直接赋值指针。

### 如果在不同Object上创建的相同组件，有可能会造成因为ID同名，而反序列化时只创建一个的问题！

所以在使用的时候需要特别注意，NewObject时通过Outer设置不同的名字，或是！更改序列化的模块，尝试判断如果是不同的Outer/Owner，则也会进行序列化。

不行，还是会有同名ID的问题，那么，使用SaveableInterface接口来进行处理！在接口中加入Static变量来控制是不行的，因为UE4接口不能声明变量

加上一个外部的Outer进行标识

## 那么问题来了，为什么不直接使用配置存储进入游戏后生成而要使用序列化？

个人想法是，用配置存储，需要根据Actor/Component的参数不同而定义不同的存储类型，存储结构，存储table等，而使用序列化的方式，则可以进行一个统一的存储结构的定义以及一个统一的初始化、替换指针引用的目的。

之后还需要再添加判断，当前的存档是否对应当前的地图MAP！！！

## 后记

Actor的SpawnActor生成，是生成在当前world对应的Level上的！所以通过Outer生成Actor时，如果该Actor的Outer不是Level的话，那么就不应该使用SpawnActor而是应该使用普通的NewOjbect方法去生成他的Outer

## 设计一个存档系统，该系统将Level与存档一起保存，读档的时候，会先加载地图再加载上面的Actor

LoadPackageAsync使用该函数对包进行预加载。

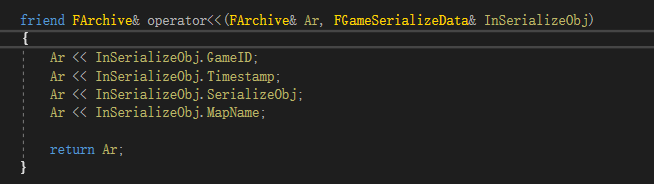
# 完整记录文档

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，只会记录些比较有用的部分以及最终自己的实现方式，会有些杂乱，敬请谅解。

## UE4序列化解析：

序列化中最重要的结构，即FArchive的类，向该类中写入二进制数据，使用的是<<操作符，对于我们自定义的结构等，需要重载<<操作符，例如



在我们自定义序列化保存时，才能正常使用。

网上能查到的的东西就不在此啰嗦了，这篇文章浅析写得很好，基本概括了UE序列化的流程。

<https://blog.csdn.net/mohuak/article/details/83027211>

## 那么问题来了，为什么不直接使用配置存储参数进入游戏后生成而要使用序列化？

个人看法是，记录生成的参数，写文件记录存档信息，一是有数据量庞大，且重新生成时需要很多的辅助代码来生成，效率低下；二是在UE中对应的文档结构也好，JSON也好，都是需要对应的数据结构（Struct/Ojb）来进行解析和处理，需要自定义大量的数据结构，劳心劳力；

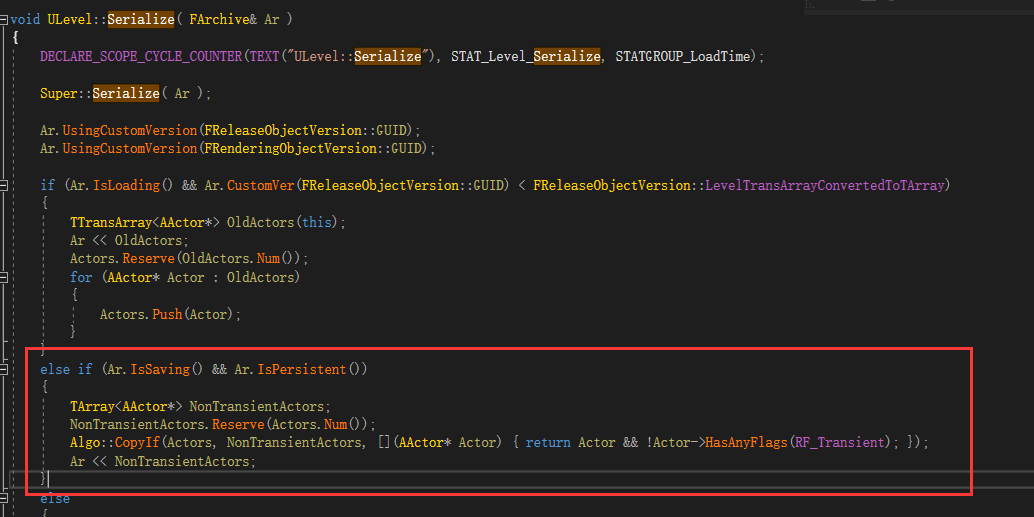
使用序列化方式的好处，可以对所有的数据对象做一个统一的处理，只是前期编写会复杂些。

## 如果你想要保存游戏runtime的场景，并且其中包含了自定义的Object、Actor等等，获取所有的Actor并且调用Serialize并不能达到保存的效果

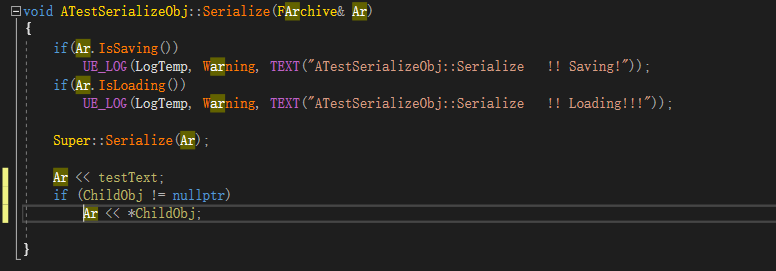
#### 1、序列化时，并不会调用对象上的指针指向的对象的序列化

UE4虽然在Object中定义了基类Serialize并且可以比较良好的运行，但是如果在游戏运行时中要动态的保存场景、保存新添加/新获得的道具武器、保存新增加的敌人等等，直接获取场景中的所有Actor并调用Actor->Serialize(Ar) ，并将Ar写入本地文件，是没法完整的还原整个场景的，原因在于，虽然Object实现了单个对象的序列化，但是并不会对该对象上的所有成员变量调用Serialize ，因此，指针的成员变量并不能正确保存，就算反序列化后，对应的依然是空指针。

但是在UE4自身的序列化逻辑中例如将预定义的Actor蓝图拖入场景中保存后，是能正常保存的，因为需要在自己想要保存的结构中，对成员变量做处理。从Level的序列化代码中可以看出：



Level将自身包含的Actor等额外存入了Ar中，包括Model等，所以如果是简单的存储需求，我们可以在代码中使用类似



这样的方式来定义保存我们自己的数据结构,如上图ChildObj就随着我们的FArchive一并进行了序列化，如果采用这样的序列化写法，则反序列化的部分也应该在该Object的Serialize步奏中完成，并且需要将指针引用做对应。

记得对应的结构要重载运算符operator<<将自定义的结构传入及保存即可。

#### 2、反序列化的顺序，对容器结构的影响也是巨大的

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。

#### 3、反序列化的跟进步奏：

SerializeScriptProperties,跟进查看对于标记UPROPERTY的参数是如何进行反序列化的:

调用SerializeTaggedProperties，序列化对象属性，并且加入tag;

最终跟进是通过 PropertyTag的

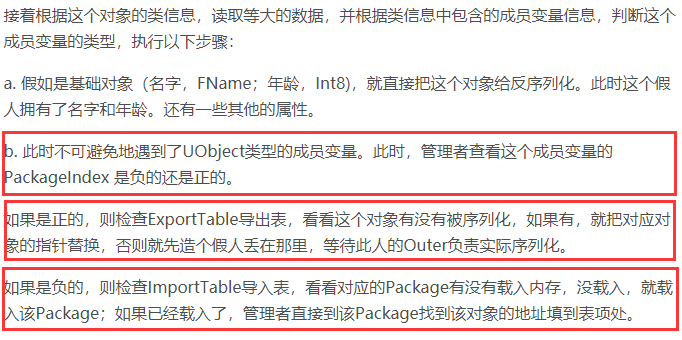
FPropertyTag::SerializeTaggedProperty(FStructuredArchive::FSlot Slot, UProperty\* Property, uint8\* Value, uint8\* Defaults) const

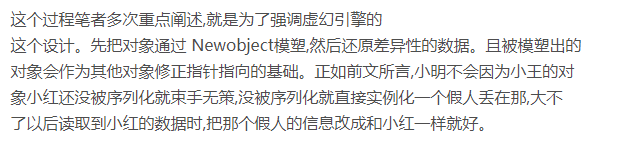
函数，将UProperty（测试时是看的UStaticMeshComponent\* GroundMeshComponet）调用

Property->SerializeItem(Slot, Value, Defaults); 进行初始化。

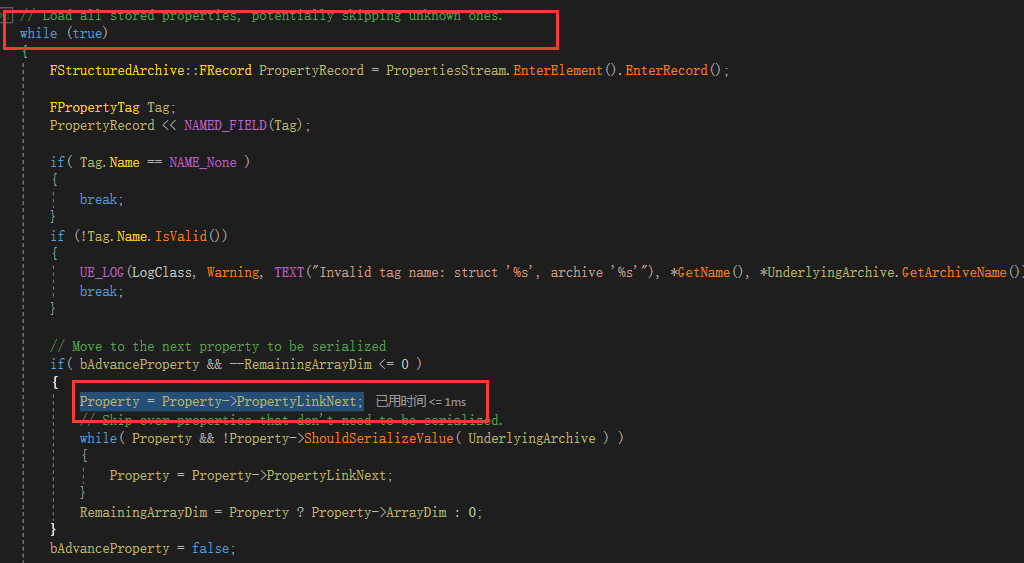
void UObjectProperty::SerializeItem( FStructuredArchive::FSlot Slot, void\* Value, void const\* Defaults ) const

测试时看到，因为之前序列化保存的GroundGridMgrComponent还没有反序列化初始化出来，所以GroundObj中的GroundGridMgrComponent并没有进行完整的赋值，只是创建了一个默认的GroundGridMgrComponent并将指针指向，在正式的GroundGridMgrComponent反序列化之后，才会将该指针指向的结构初始化（还是替换为正确的？）生动的说明在此：





当一个UPROPERTY属性初始化完成之后，



通过While循环会将指针指向Next进行初始化。

如果是UStruct，则会通过

UStructProperty::SerializeItem(FStructuredArchive::FSlot Slot, void\* Value, void const\* Defaults) const

进行初始化

保存时序列化也是走类似的流程。

如果该对象从来没有被加载过，则包内不会出现该对象，此时应该手动加载。

场景从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。

## 我所使用的保存场景方式：

#### 思路：

仿照UE4序列化的理念进行场景序列化保存。

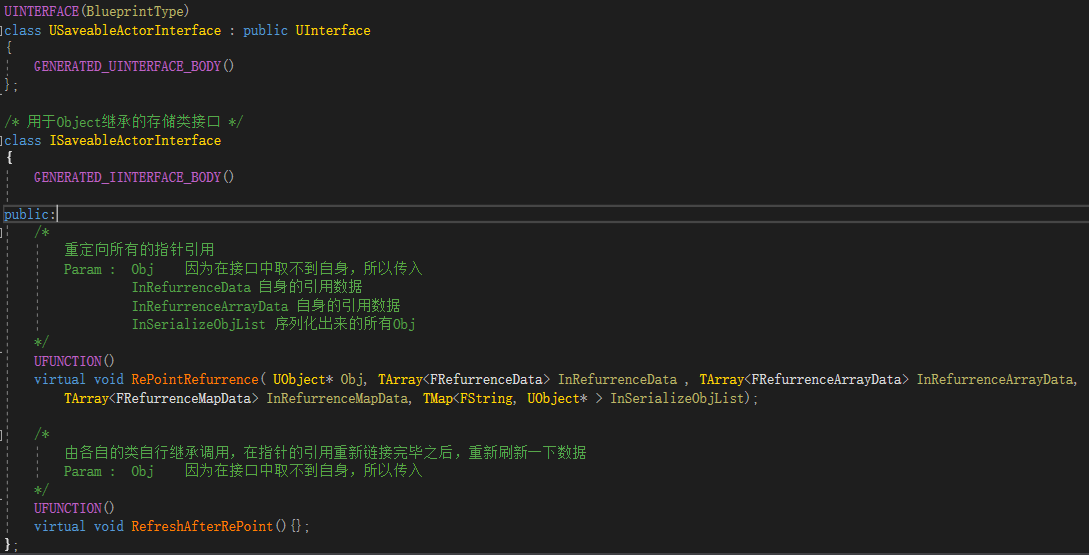
首先从场景中选出需要进行保存的Actor，逐个调用Actor->Serialize() ；

对每一个Actor调用序列化后，遍历其身上的Property字段，取出其中需要保存的Object对象以及包含了Object对象的TArray、TMap，并将其也进行序列化再递归遍历Property的过程，并且将这些Property标记为需要更新其外部指针引用的Property。所有序列化完成之后，将二进制文件写入本地。

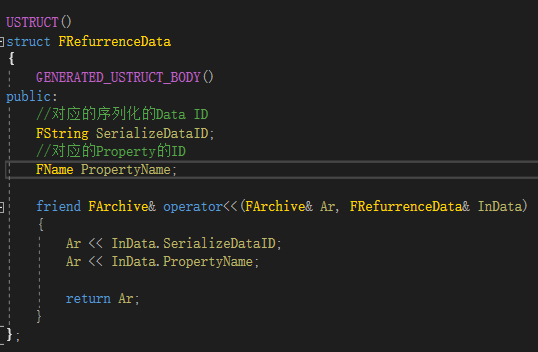
返序列化时，先将所有的Actor/Object反序列化，再根据序列化时输出的Property引用对应关系，对每一个Actor/Object设置其指向的指针对象，再调用该Actor/Object的Refresh更新一下需要更新的部分，就能很好的完成整个场景保存的需求。

#### 部分代码逻辑

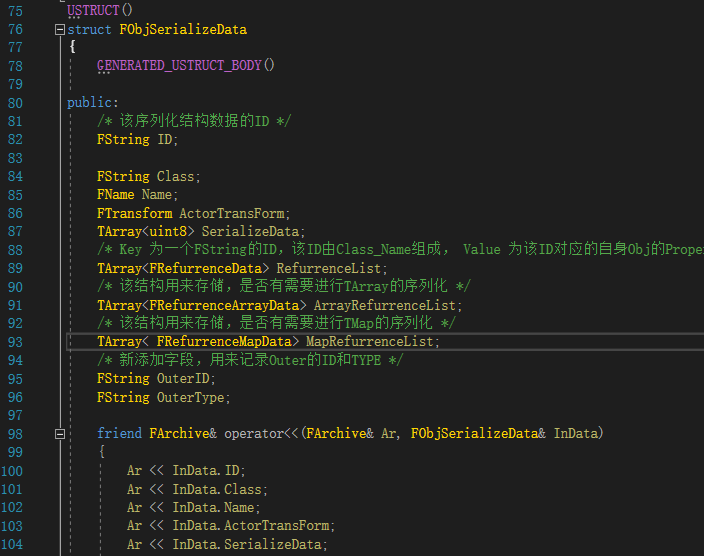
首先定义一个接口，所有在场景中需要保存的对象，都继承自该接口，该接口提供了一个方法为RePointRefurrence，在反序列化时根据对应数据设置自身的指针引用。



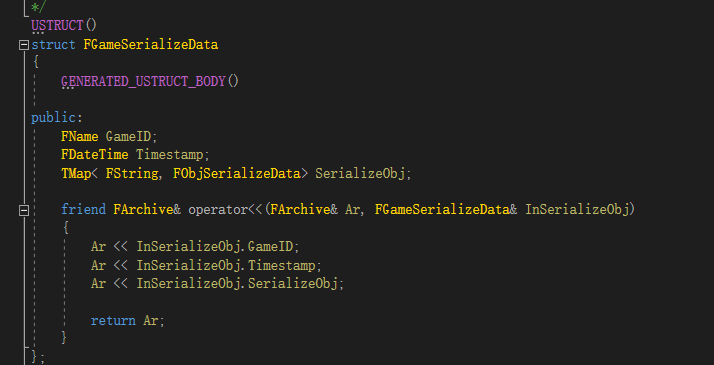
之后需要定义引用关系结构，因为有些变量为指针，需要在反序列化完成之后，根据对应的PropertyName根据反射进行设置对应的指针地址。



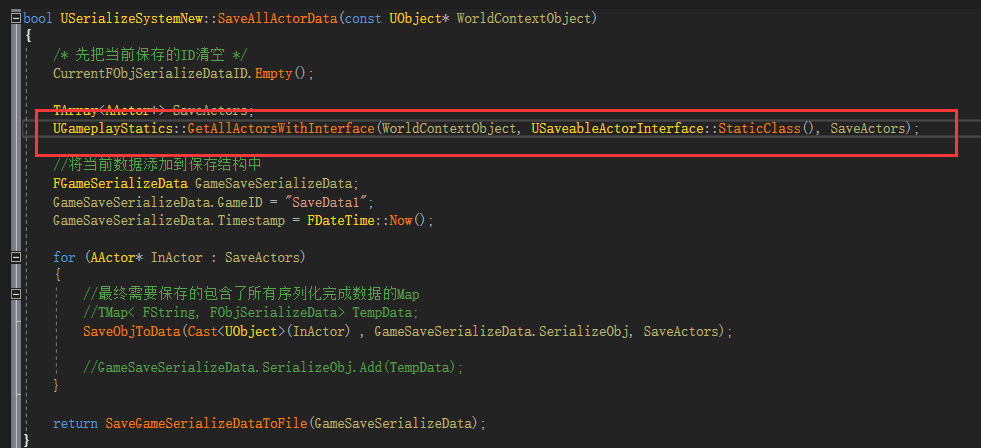
再定义一个序列化后，用来存储每一个Object的Struct



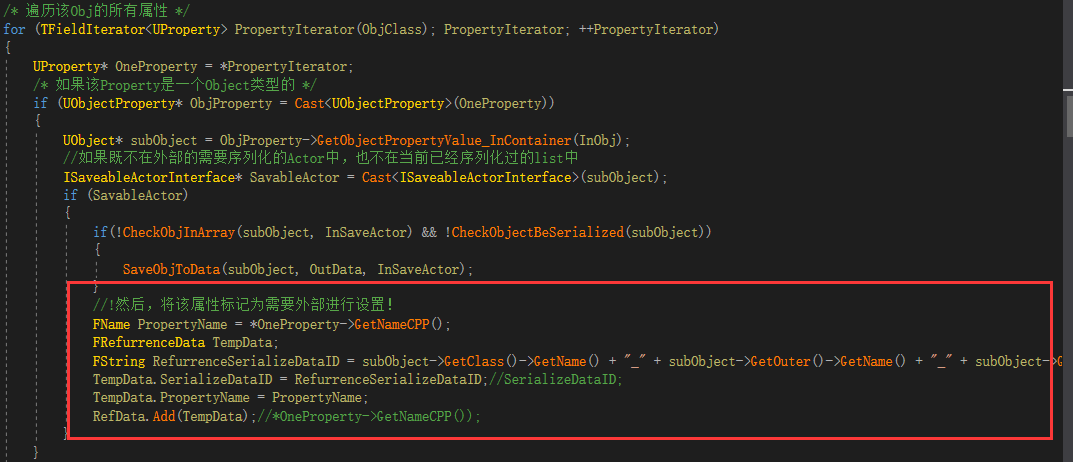
以及存储所有Object的整个的Game存档Struct



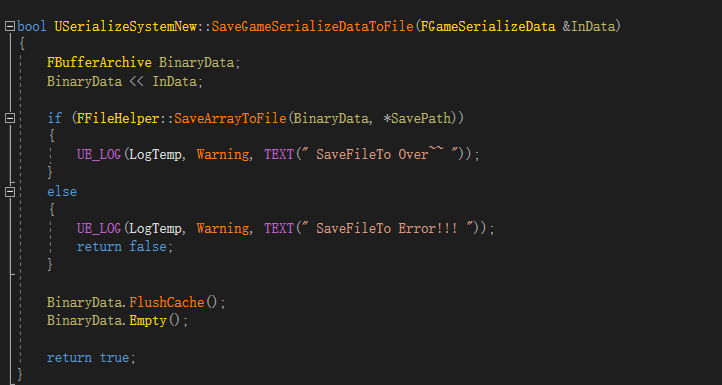
之后进行序列化时，

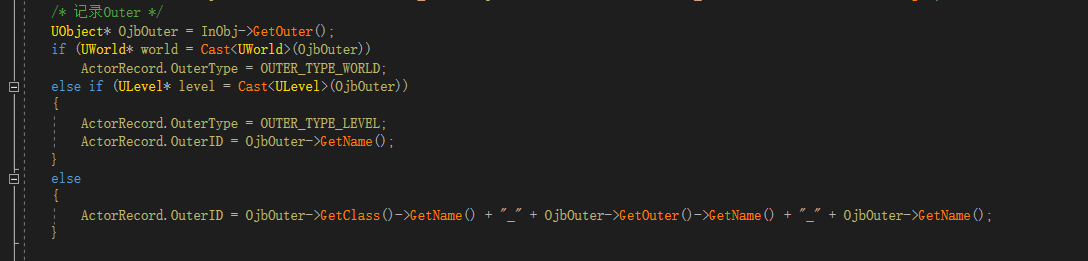


取出场景中的所有的Actor并递归进行序列化。



递归遍历其Property并且输出引用对应关系。

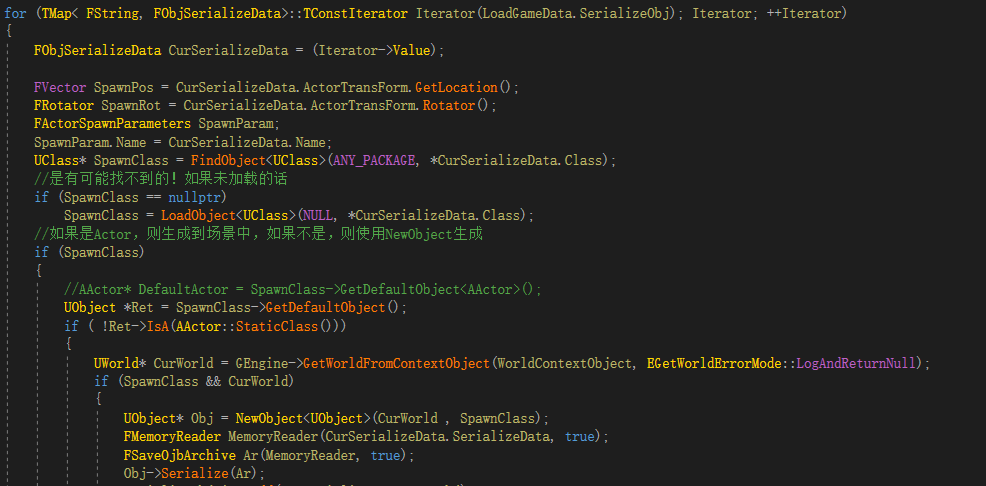




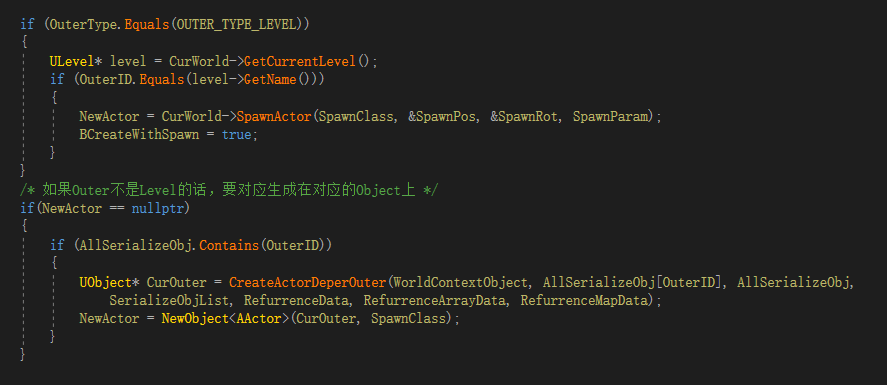
找到每一个需要生成的Obj的outer并进行存储，对于Outer部分非常重要，如果Outer没有对应上有几个问题，比如如果SpawnActor，所有的Actor都会放置在场景上不管需不需要的，右侧的Level栏会特别的凌乱，也无法通过GetComponent之类的方法获取到正确的组件等；下次保存的时候所有的数据依旧会错乱等。所以，保存并且正确根据Outer生成Actor非常重要。

将文件写出到本地。

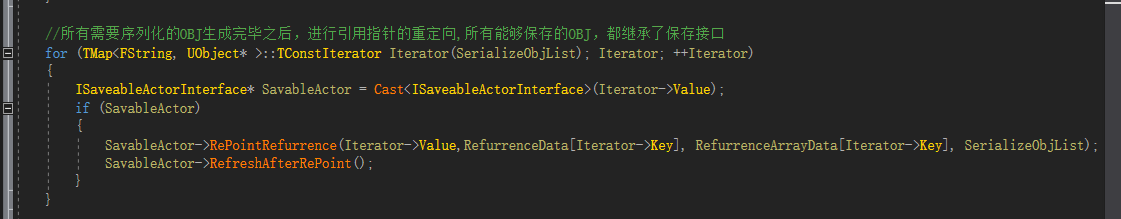
反序列化时，

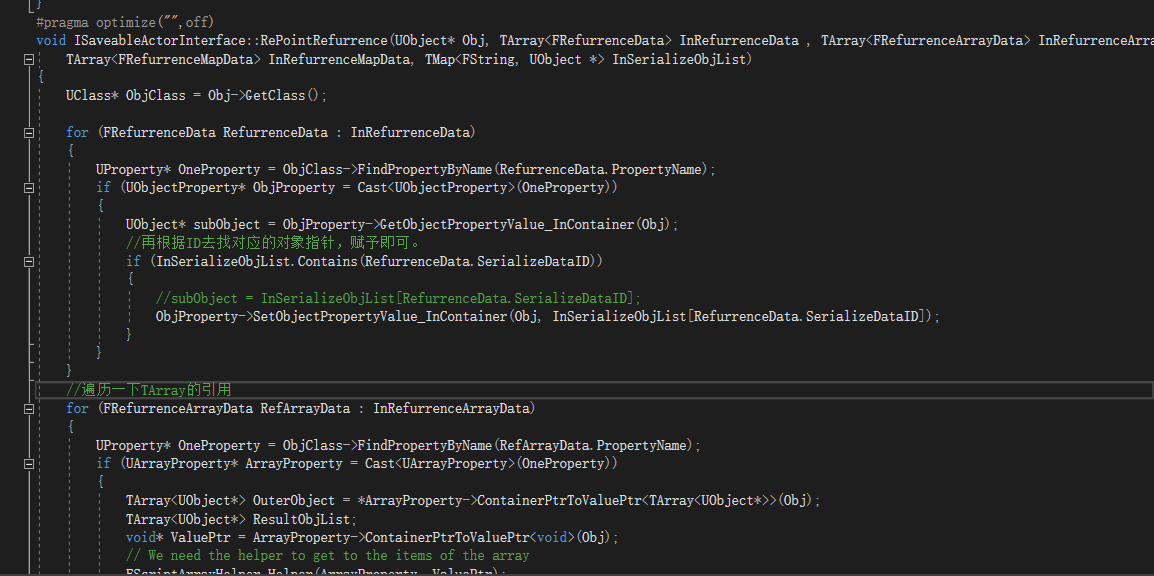


将每一个记录的Actor进行反序列化，需要特别注意，关于Outer的处理，不是所有的Actor都调用Spawn生成在地图上的



当所有需要加载的Actor创建完成后，遍历进行指针的替换。





对所有反序列化完成的Actor更新其Property引用，并且调用刷新函数更新数据（如果该Actor自身需要更新数据）。