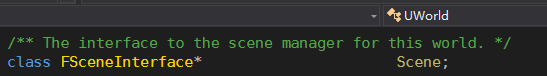
UE4分析

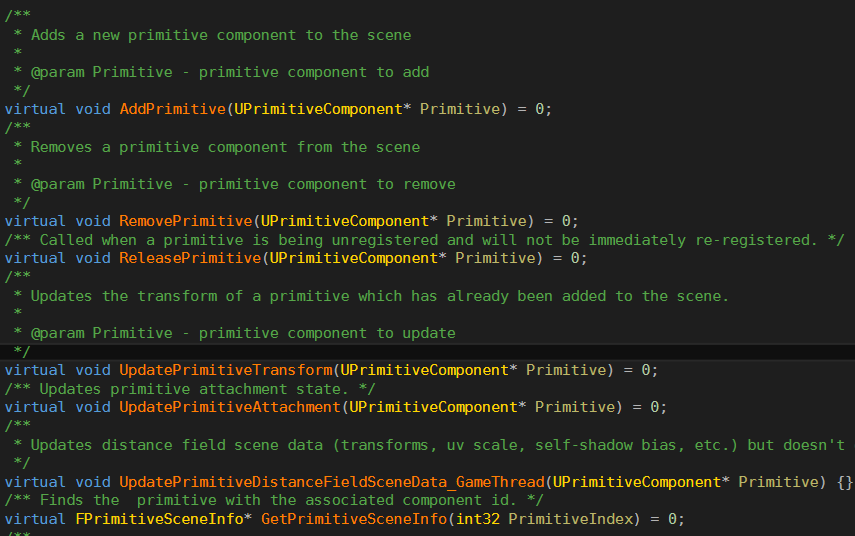
# 渲染

位于引擎中Engine\Source\Runtime\Engine\Public\SceneInterface.h的FSceneInterface接口类是一个UWorld对应的渲染接口：



它主要有以下接口函数：

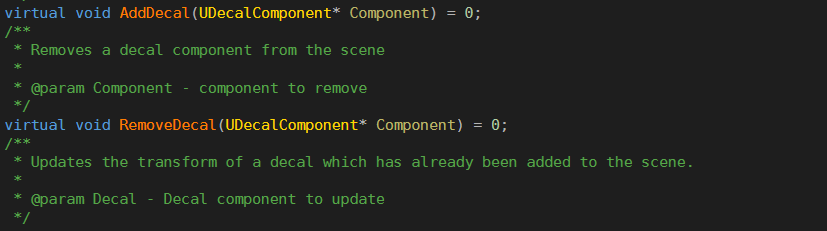
-相关Primitive组件：



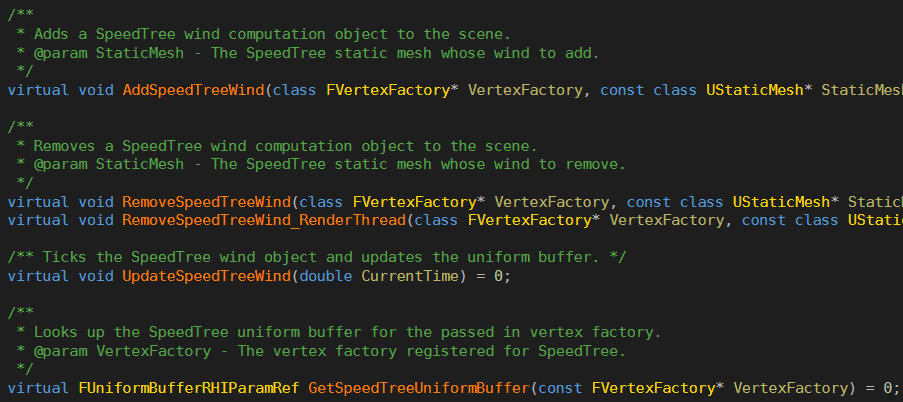
-相关Light组件：



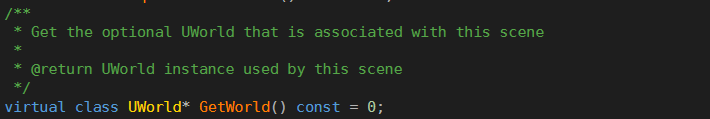
-相关Decal组件：



-等等其他的可渲染组件，其中也有SpeedTree的存在(SpeedTree是一款树的建模软件，有专门的UE4版本)

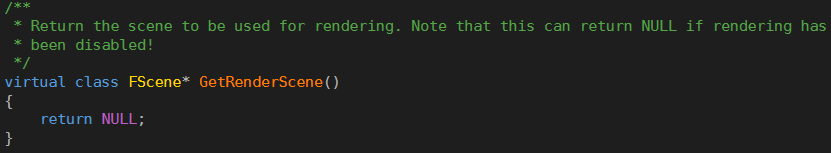


这个类是和UWorld相关联的

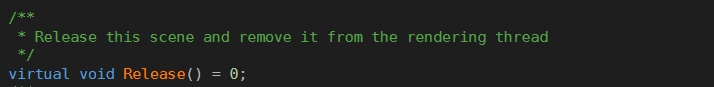




并且有相应的Scene



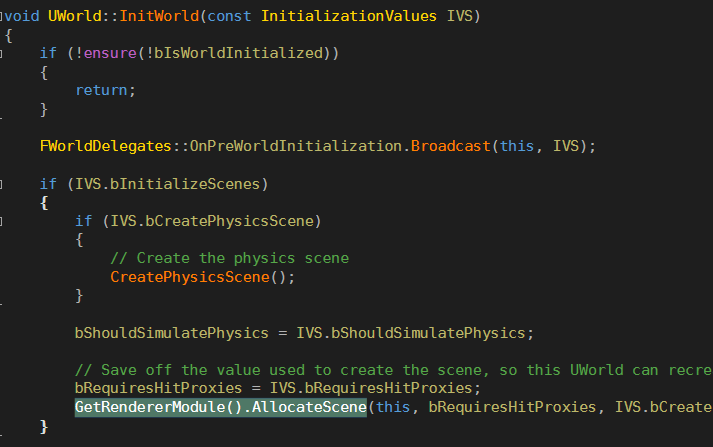
当然也和渲染线程相关（~只从接口类暂时不清楚如何关联~）



这个FSceneInterface接口类是一个实现了FSceneInterface的FScene的接口，而FScene通过GetRendererModule().AllocateScene构建；

其中找到的一个调用是在World.cpp中的InitWorld()初始化World时进行的：

从具体的实现中能够发现这个世界渲染与否和World的初始化参数中的bInitializeScenes参数相关。而Scene这个单词可以理解为是World在渲染模块中的概念

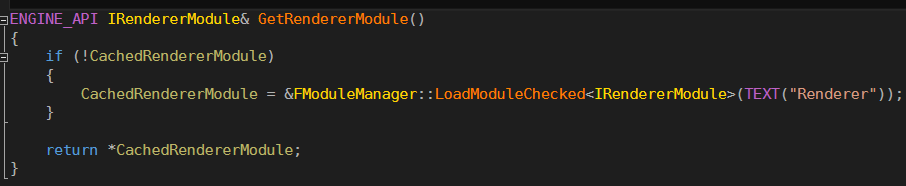


其中这个GetRendererModule()是UE4中的“ENGINE\_API”

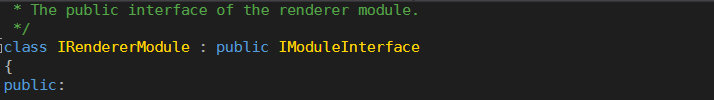
可能是UE4提供的引擎API

它是位于Engine\Source\Runtime\Engine\Public\EngineGlobals.h中

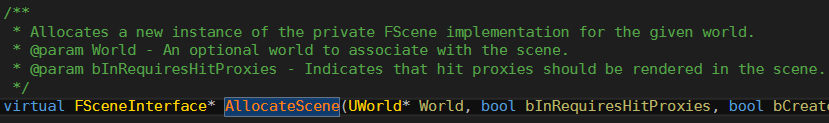
其功能一目了然：返回或加载返回渲染模块<IRendererModule>



IRendererModule是声明在Engine\Source\Runtime\RenderCore\Public\RendererInterface.h中的，继承自IModuleInterface（所有游戏引擎用到的模块都应当继承于此，它只有一些加载，卸载，等等的基函数）（注意: IRendererModule依然是一个接口类）



AllocateScene的定义



到此一个渲染模块已经存在，并且通过渲染模块的AllocateScene分配了FScene的实例到UWorld上，也就是 < FSceneInterface \*> ，从World中的RecreateScene()中可以看到World中的< FSceneInterface \*>Scene与渲染模块IRendererModule是紧密相关的，而且RecreateScene的操作只可能在AllocateScene中进行，因此推断AllocateScene生成了相关这个World的FScene对象，而且为这个World中的成员变量< FSceneInterface \*>Scene赋予了初始值即生成的FScene对象



这里小结一下：

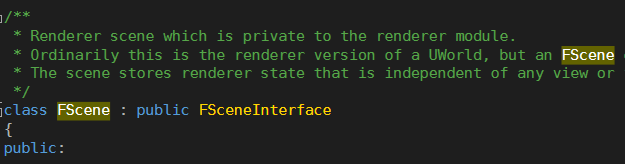
UWorld中存在一个成员变量<FSceneInterface\*>Scene【UWorld中主要UWorld::CreateFXSystem(),

UWorld::UpdateLevelStreamingInner(),

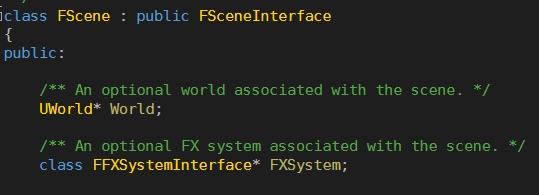
UWorld::UpdateParameterCollectionInstances()存在相关的调用】

而在UWorld的InitWorld()中，通过调用GetRendererModule().AllocateScene()生成了相关这个UWorld的FScene对象，且为这个UWorld中的成员变量< FSceneInterface \*>Scene赋予了这个初始值【注意：FScene是继承于FSceneInterface的，所以UWorld中的FSceneInterface\*变量命名是Scene】

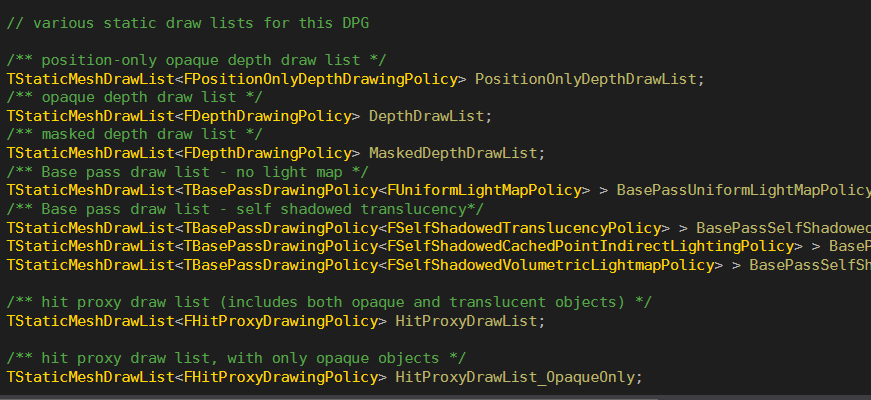
FScene声明于Engine\Source\Runtime\Renderer\Private\ScenePrivate.h中，继承自FSceneInterface



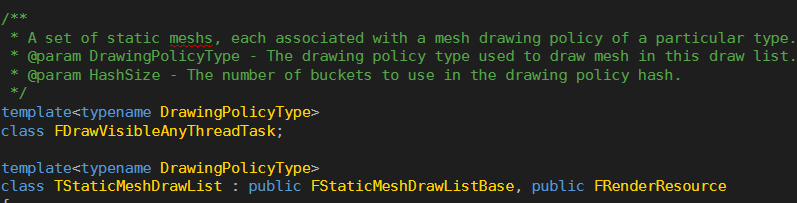
FScene中有当前相关世界的索引和FXSystem接口（FX在UE4中指一些特效）



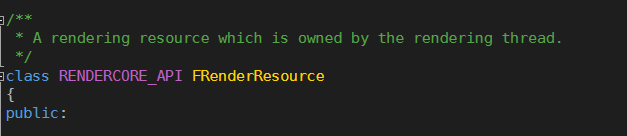
最重要的，它有各种DrawList，概念类似于渲染队列？



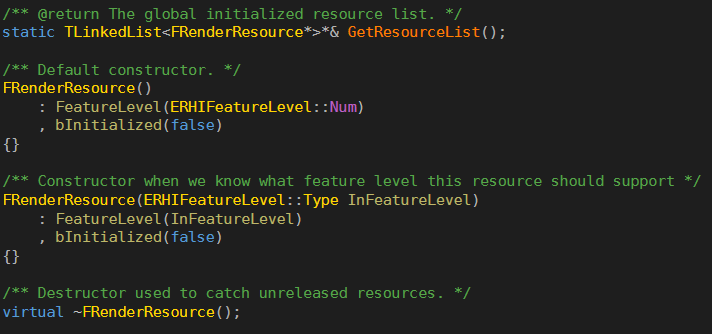
其注释如下：

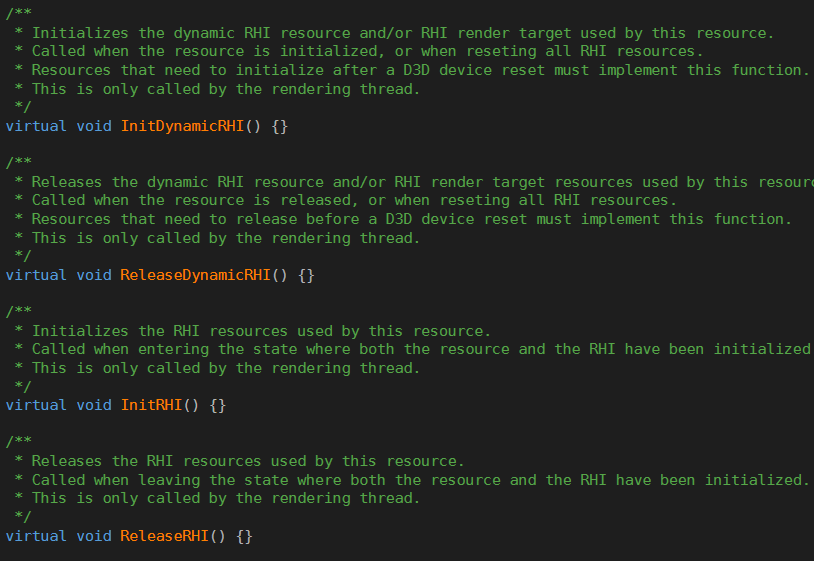


其中RenderResource位于Engine\Source\Runtime\RenderCore\Public\RenderResource.h，对应的注释如下：

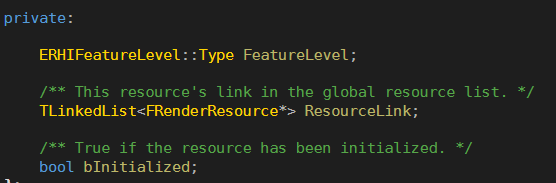


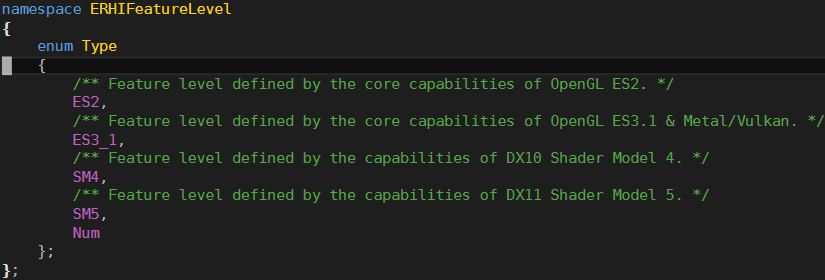
相关的函数：





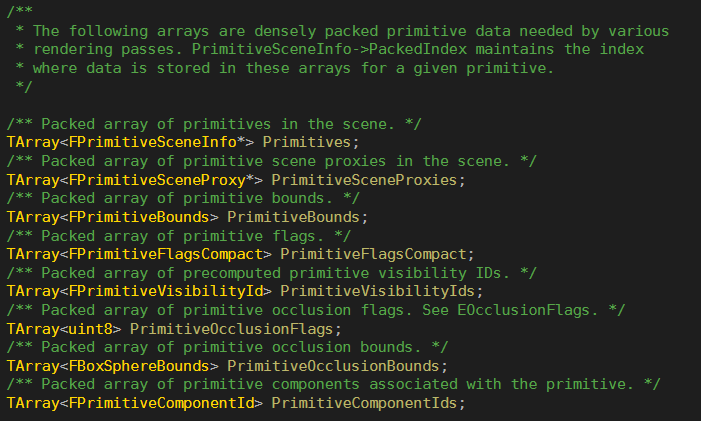
成员变量：



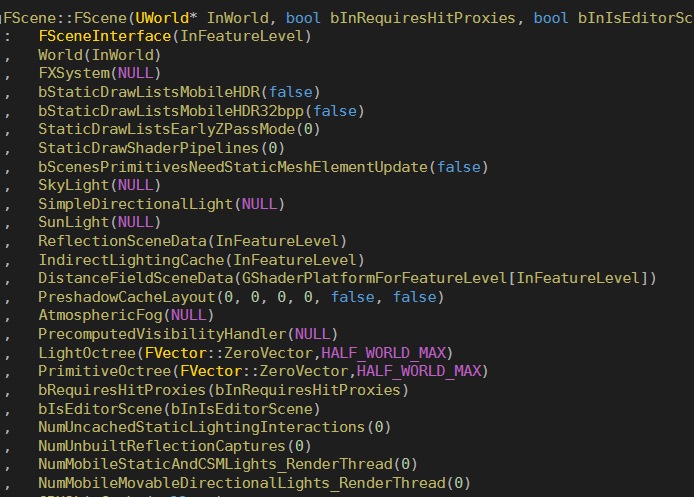


RHI: Render hardware interface 渲染硬件层接口，是虚幻4建立的一套与底层图形硬件平台无关的图形渲染API.

如注释所说的：FScene中存放有进行渲染需要的数据



FScene的构造函数，有一堆的初始化列表项



关键的过程：

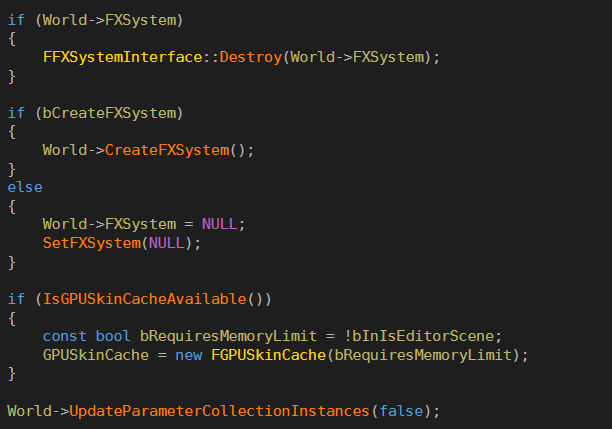
1.check World成员（初始化列表中初始过）并设置World中的接口引用自身



2.FScene中的FeatureLevel的赋值

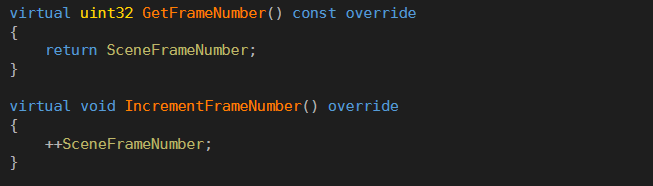


3.对World的设置（FXSystem相关等）



……

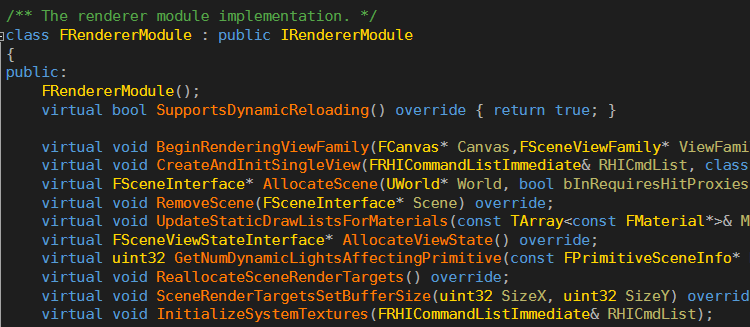
FScene中的Frame，通过这个可以大概知道FScene是一个什么样的地方，即FScene是UWorld在渲染中的表现，之前也说了FScene中有渲染所需的各种信息集，例如：TArray<FPrimitiveSceneProxy\*> PrimitiveSceneProxies; 而PrimitiveSceneProxies也是UPrimitiveComponent组件在渲染中的表现，和GamePlay游戏框架有相似之处



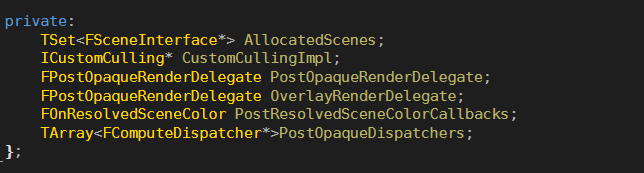


FRendererModule声明于Engine\Source\Runtime\Renderer\Private\RendererModule.h

继承于IRendererModule接口

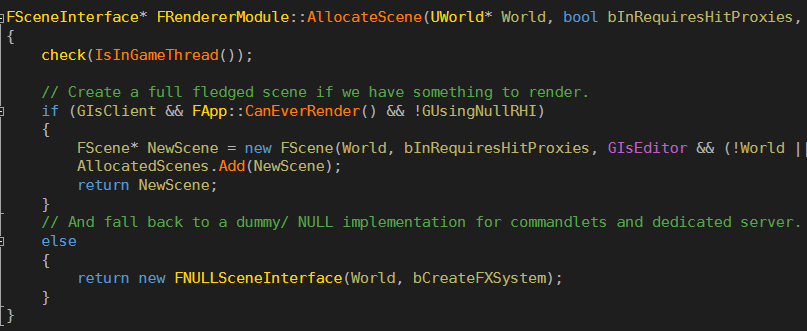


关键的成员变量：

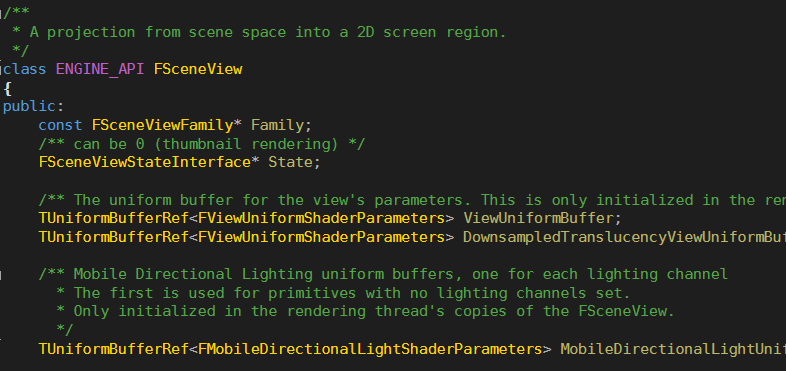


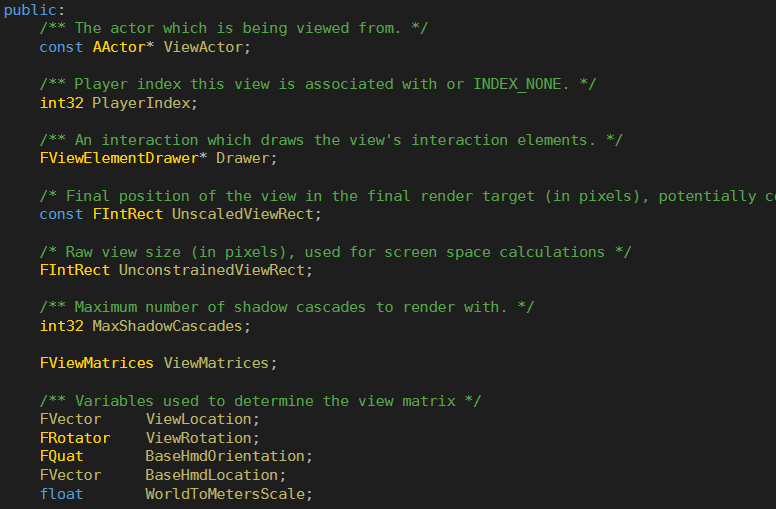
AllocateScene的实现：

FScene与UWorld之间的过程主要由FScene的构造处理，AllocateScene中主要过程是FScene与自身的关系：AllocatedScenes.Add(NewScene);因此可以看出FRendererModule是相当于一个管理器（管理FScene）的存在，（~但独立的游戏只有一个World，只有一个FScene，那么这个FRendererModule中只有一个元素~）



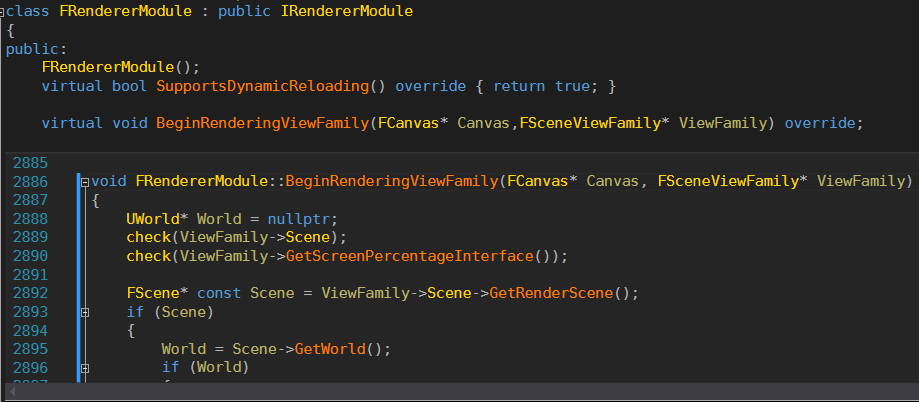
SceneView位于Engine\Source\Runtime\Engine\Public\SceneView.h

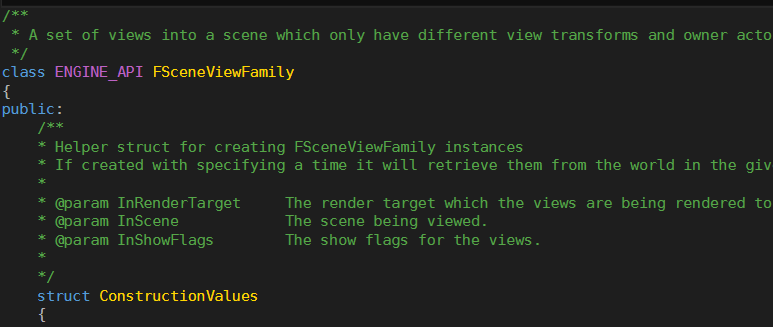




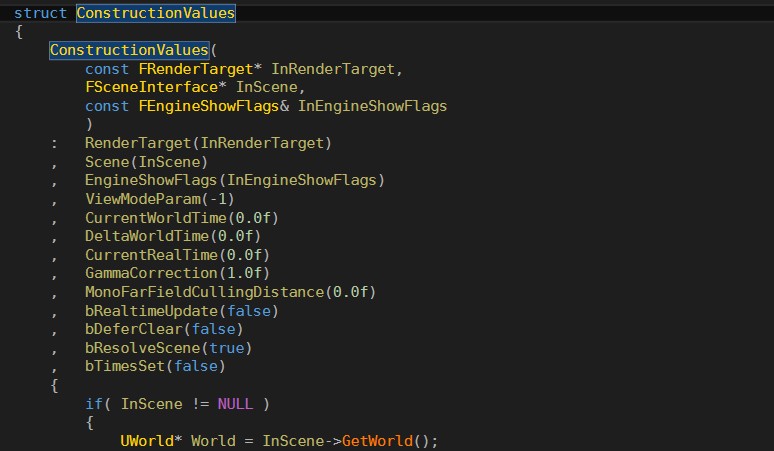
FSceneViewFamily也位于Engine\Source\Runtime\Engine\Public\SceneView.h

它在FRendererModule的BeginRenderingViewFamily()中作为参数被调用，本身没有存储相关的引用

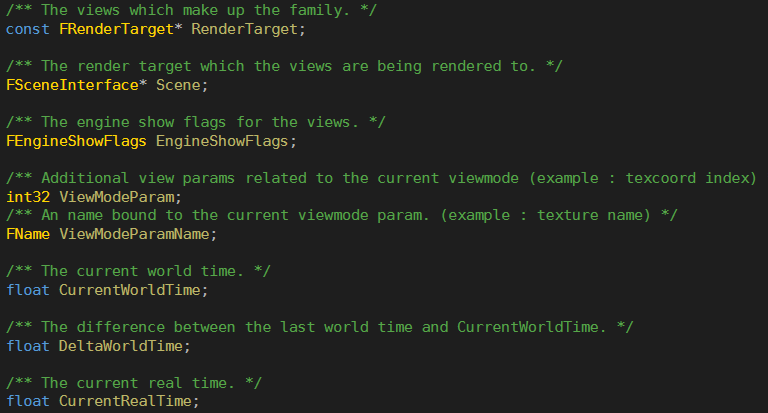




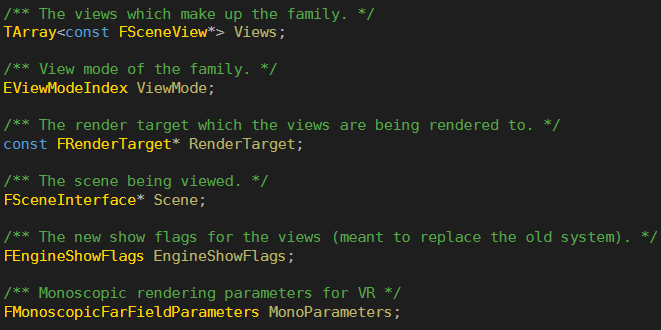
结构体ConstructionValues：



ConstructionValues中存在有以下成员：

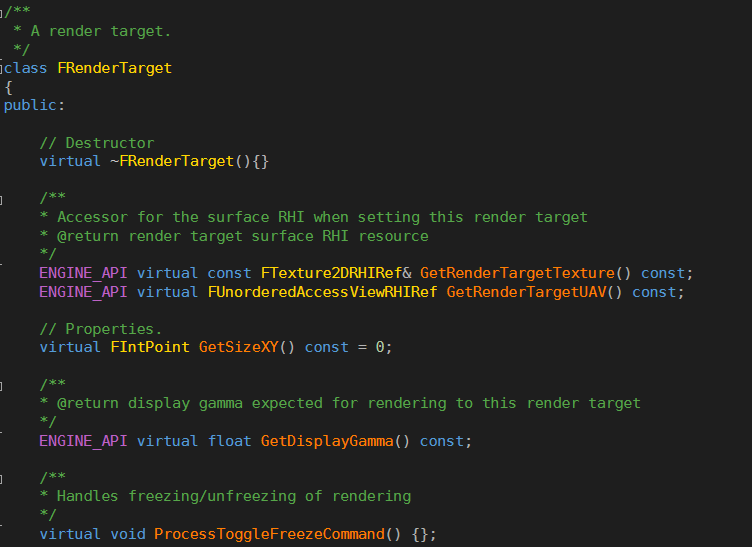


FSceneViewFamily也是由FSceneView构成的，以下是FSceneViewFamily的部分成员变量



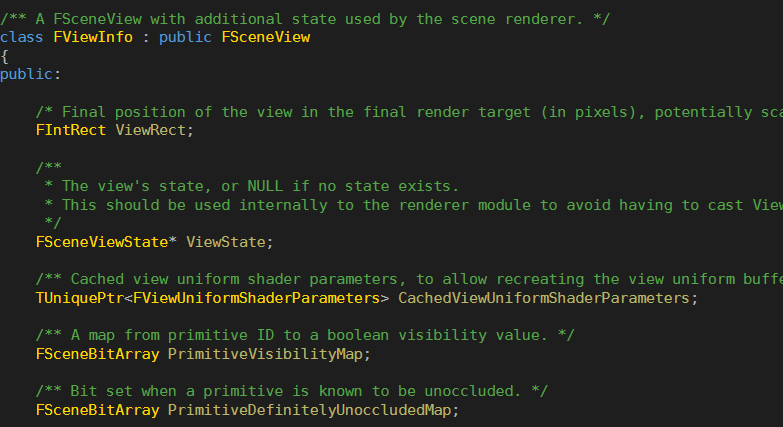
FRenderTarget位于Engine\Source\Runtime\Engine\Public\UnrealClient.h

Render Target在引擎中是一个对象，可用于做实时贴图，位于材质&贴图->创建Render Target对象，同时从所属模块来看也不是渲染实现相关的类

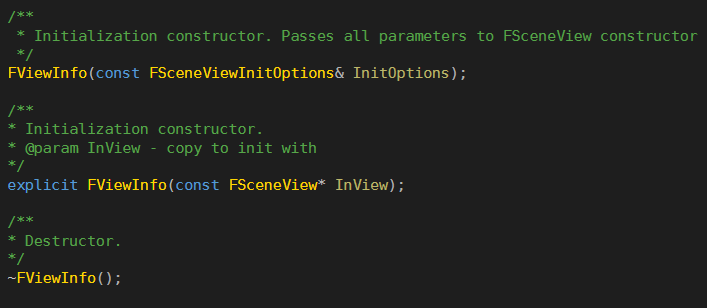


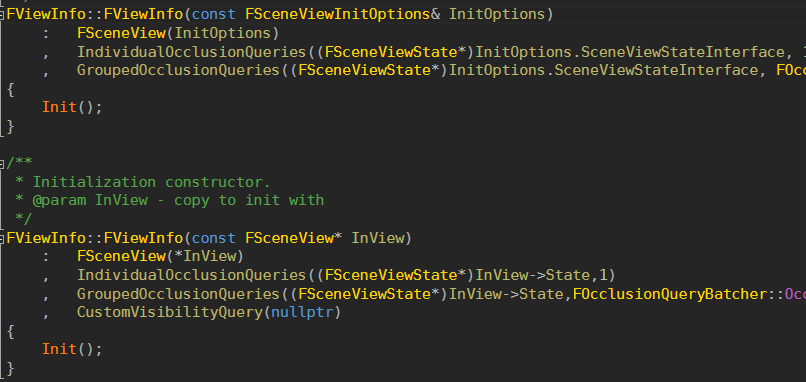
FViewInfo位于Engine\Source\Runtime\Renderer\Private\SceneRendering.h

注释中说FViewInfo是FSceneView和一些其他的状态信息组成的类，而且也是继承于FSceneView的，而FSceneView是Scene到2D的投影，但与其不同的是FViewInfo位于渲染模块下

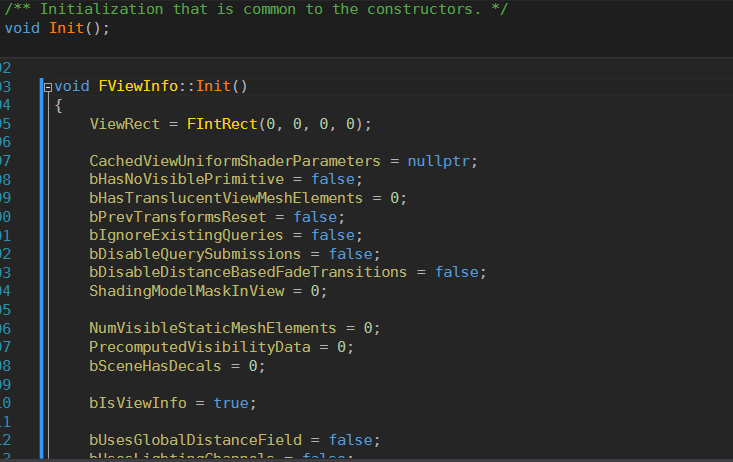


FViewInfo的构造函数：

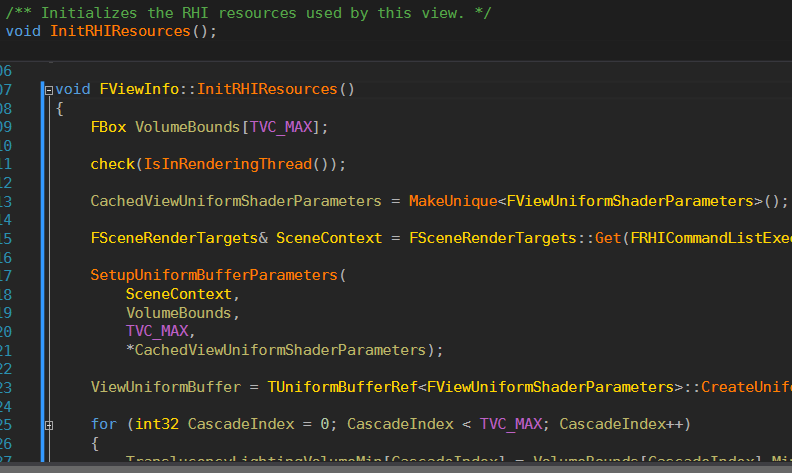




FViewInfo的Init()函数：进行大量的变量初始化

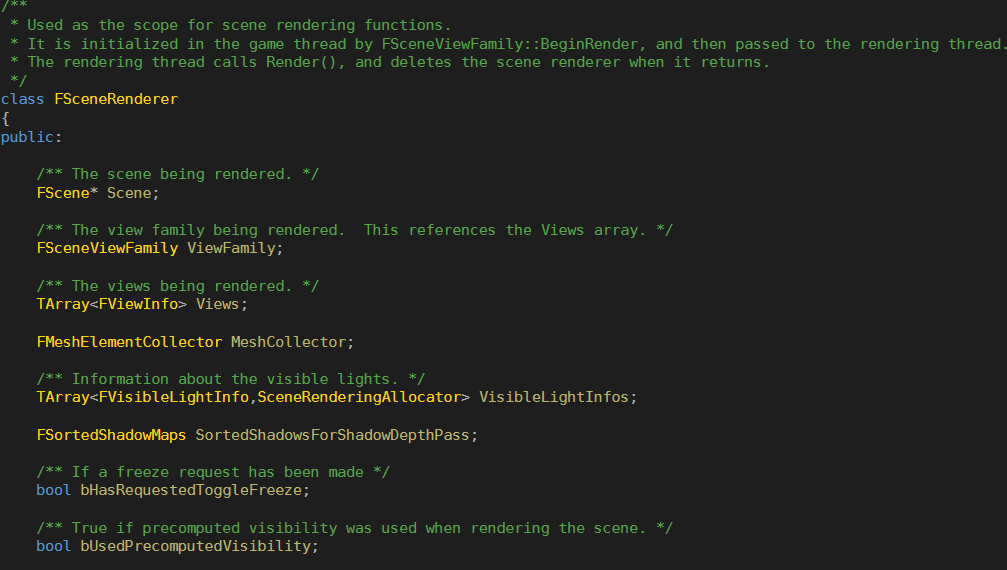


FviewInfo的其他函数

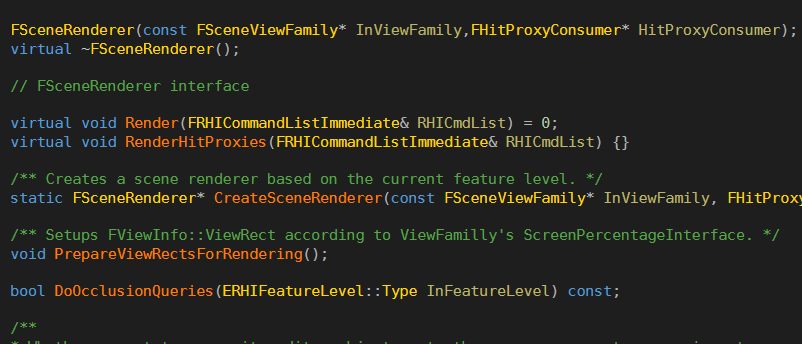


…

FSceneRenderer位于Engine\Source\Runtime\Renderer\Private\SceneRendering.h

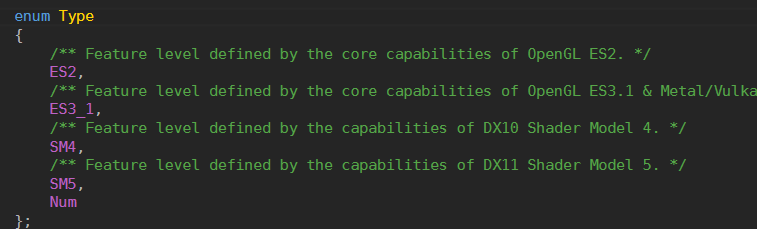


FSceneRenderer的重要函数：



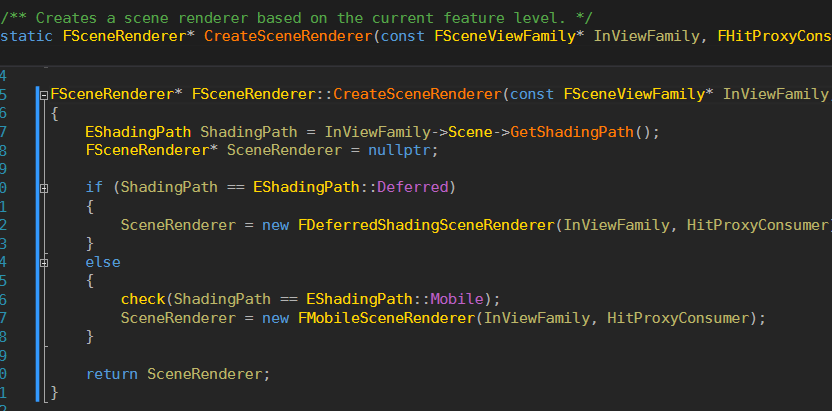
注意FeatureLevel是一个枚举类型：





FSceneRenderer:: CreateSceneRenderer ():

根据ShadingPath的不同，选择使用延迟渲染和面向移动端渲染（前向渲染）

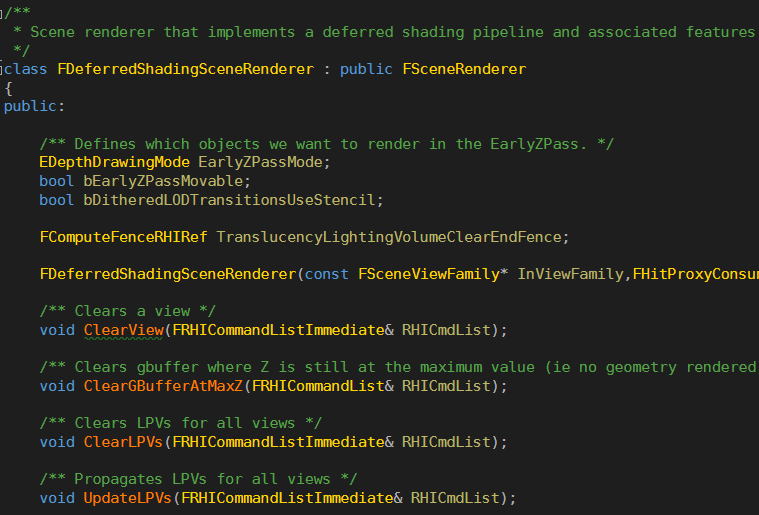


FMobileSceneRenderer位于Engine\Source\Runtime\Renderer\Private\SceneRendering.h（与FSceneRenderer同一下目录下）



FDeferredShadingSceneRenderer位于

Engine\Source\Runtime\Renderer\Private\DeferredShadingRenderer.h下



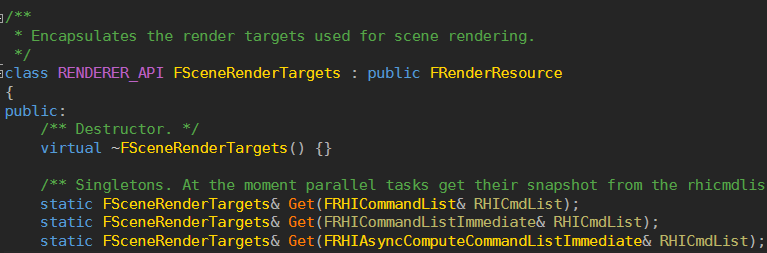
主要函数：

FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList)

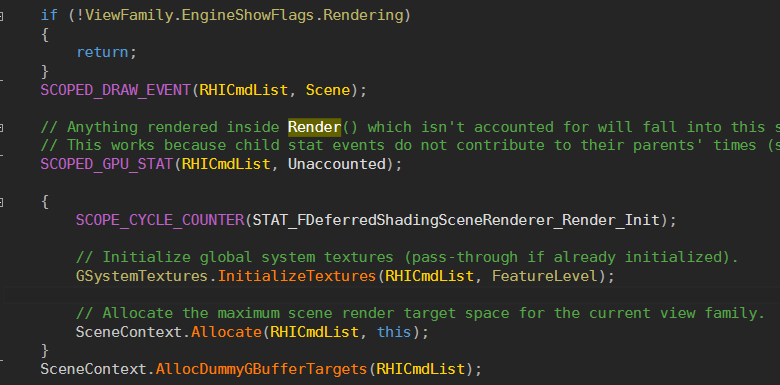
1.首先调用了PrepareViewRectsForRendering()，相关游戏的显示窗的百分比，接着命名事件，创建FSceneRenderTargets……



其中的FSceneRenderTargets的定义如下

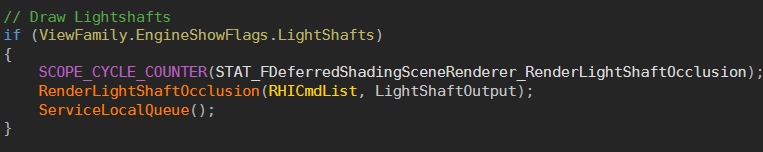


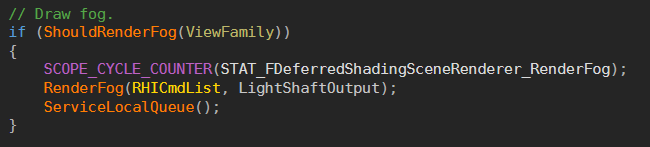
2根据ViewFamily的标记位决定是否继续渲染



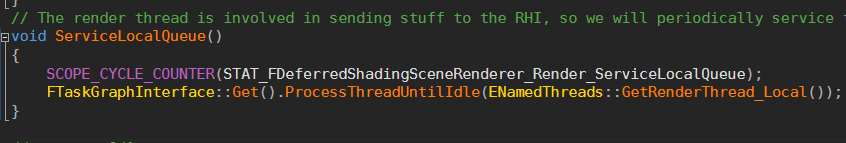
3后面实现根据各种标记位判断是否渲染相关（比如：Light，）……

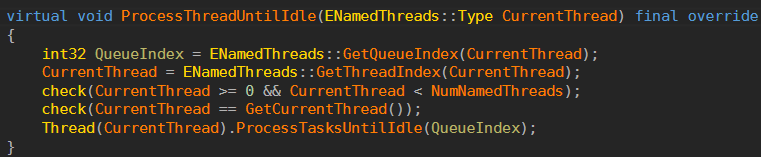
比如Lightshafts，fog，这些渲染实现相似，直接先调用相应的本类函数接着调用SeriveLocalQueue()

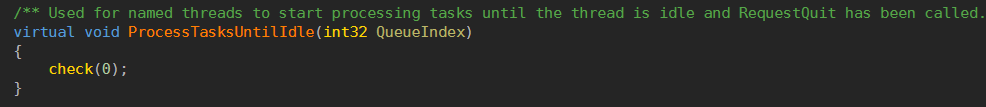




ServiceLocalQueue()实现如下：







……

FPrimitiveSceneProxy:

FPrimitiveSceneInfo:

小结：从FSceneRenderer中理解ViewFamily是一个相机(GamePlay框架中)的List，而Scene是这些相机所属的World(GamePlay框架中), Views则是所有FSceneView的对象再加上额外的渲染信息，两者所属模块不一样，前者所属Renderer，后者所属Engine，FSceneRenderer中的Render()默认UE4使用延迟渲染，否则使用前向渲染（透明物体）。两种FDeferredShadingSceneRenderer和FMobileSceneRenderer都继承于FSceneRenderer用于两种渲染方式，并重写相应的Render()实现

官方文档中如下介绍：

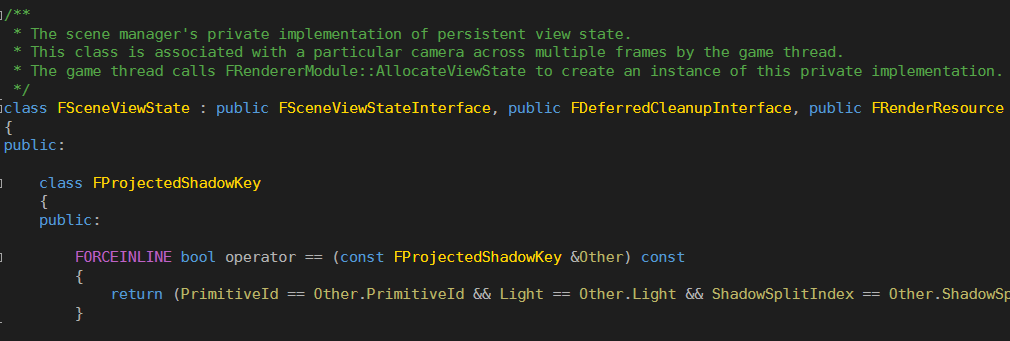
**FSceneView** : 单个视图到一个 FScene 的引擎代表。视图可以通过对 FSceneRenderer::Render 的不同调用的不同视图来渲染（多编辑器视口）或通过对 FSceneRenderer::Render 的同一调用中的多个视图来渲染（分屏游戏）。为每个帧构建新视图

**FViewInfo** : 渲染线程中描述View信息的对象，可以看成是view在渲染线程中的副本

**FSceneViewState** ： ViewState 存储有关在多个帧中需要的某个视图的私有渲染信息。在游戏中，每个 ULocalPlayer 都有一个视图状态

**FSceneRenderer** ： 每个帧都会被创建的类，用于封装跨帧的临时对象

FSceneViewState位于Engine\Source\Runtime\Renderer\Private\ScenePrivate.h中



其父类FSceneViewStateInterface

位于Engine\Source\Runtime\Engine\Public\SceneManagement.h中



总体来说：UE4为了解决渲染和Tick的时机问题，采用了游戏线程和渲染线程的概念，游戏线程更加接近于GamePlay的框架，有World，有Actor，有Player等等相关游戏和玩家的概念，而渲染线程是这些可渲染物体的仅相关渲染需要的一些例如顶点，面等信息，没有玩家生命，得分这些信息，这样便能解决Tick和Render某种情况下一个父物体的两个连在一块的子物体分裂开的情况，有了代理的渲染线程，UE4还有一个RHI线程的概念，渲染线程不直接与GPU交互，而是通过RHI间接处理，同时也能更容易适应多平台，UE4中可以说是只有组件才有可能是可见的（可渲染），Character等只是一个概念和支持它渲染的组件的集合体，而这些可渲染的组件都是继承自UPrimitiveComponent的，因此相关的渲染代理的创建等也是UPrimitiveComponent内的方法实现的，也就是说当渲染线程启动以后，在合适的时机渲染线程的各种组件的渲染代理向游戏线程中的本体获得渲染信息之后渲染线程再将相关的渲染指令交给RHI线程让GPU处理，RHI可能会因此阻塞，但渲染线程只是不断地向RHI指令（RHICommandList）中推送指令即可

# 物理

位于Engine\Source\Runtime\Engine\Classes\Components\StaticMeshComponent.h中的

StaticMeshComponent是经常用到的UE4的静态网格组件，与之相比还有一个常用的网格组件为SkeletalMeshComponent，(骨架网格组件)常用于玩家这些具有骨骼的模型

它位于Engine\Source\Runtime\Engine\Classes\Components\SkeletalMeshComponent.h中

它们的继承链如下：

UActorComponent

->USceneComponent

-->UPrimitiveComponent

--->UMeshComponent

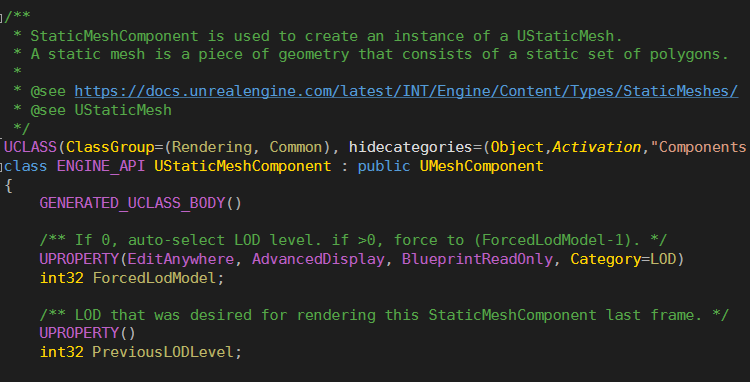
---->UStaticMeshComponent

---->USkinnedMeshComponent

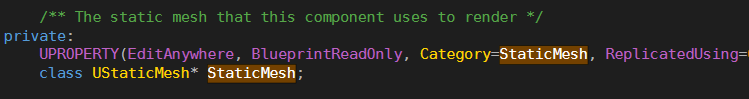
----->USkeletalMeshComponent

首先看StaticMeshComponent的声明:

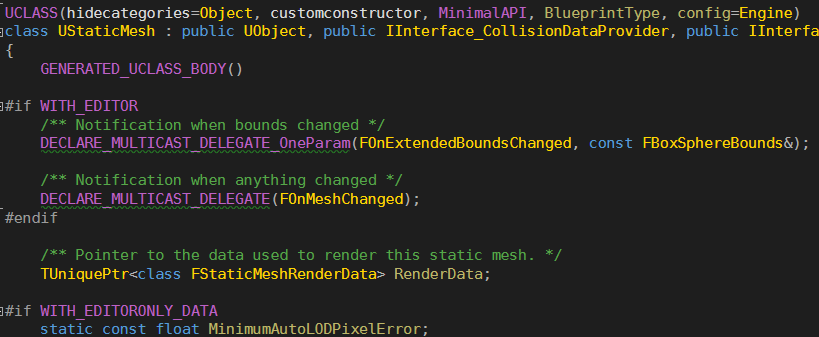
它是用于创建StaticMesh实例的类



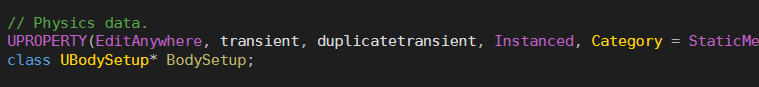
因此也拥有相关的原型



位于Engine\Source\Runtime\Engine\Classes\Engine\StaticMesh.h中的StaticMesh即是网格资源，可以被移动，旋转和缩放，但无法拥有其顶点动画



其中StaticMesh有一个成员为BodaySetup，它相关物理的数据

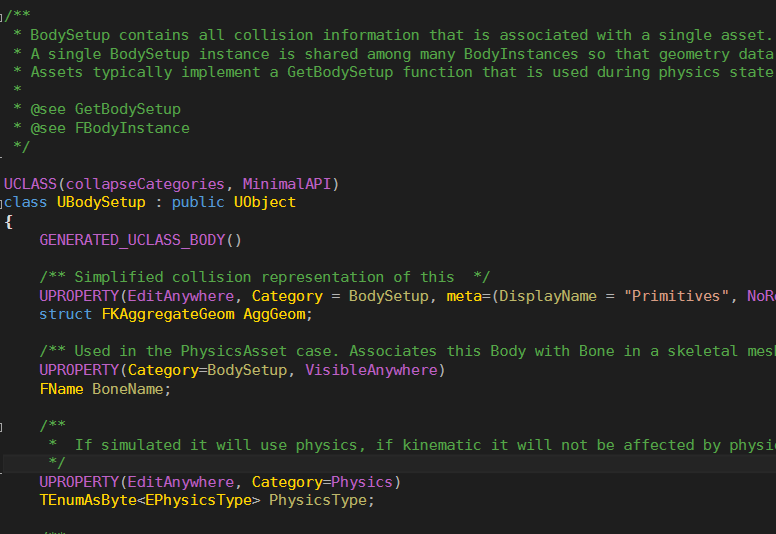


UBodySetup位于Engine\Source\Runtime\Engine\Classes\PhysicsEngine\BodySetup.h中

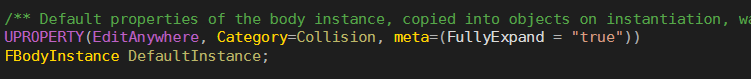
声明如下:

注释中解释为BodySetup中包含了与单个资产(asset)相关联的所有碰撞信息，它可以是很多BodyInstances共有的原型，资产(asset)通常在GetBodySetup()中产生。

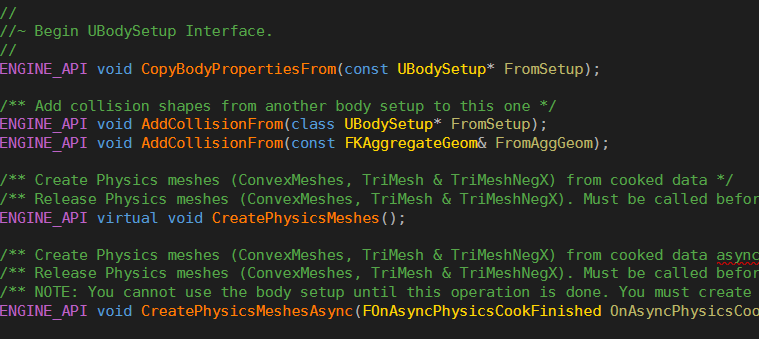
同时可以看到它的其中一个名为BoneName的成员，便可推知这个类在GamePlay框架中是一个比较基本的单元



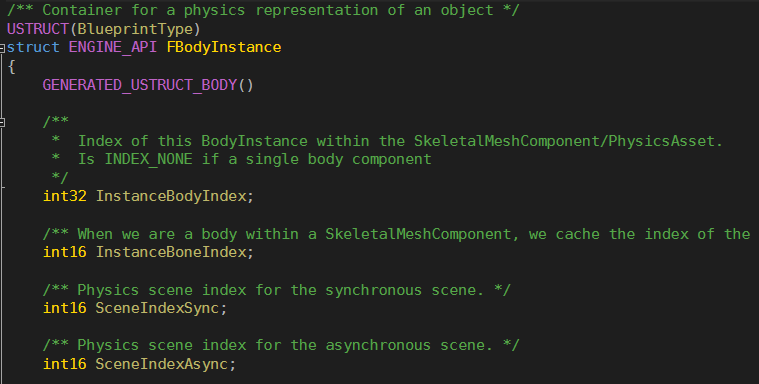
它有默认的FBodyInstance



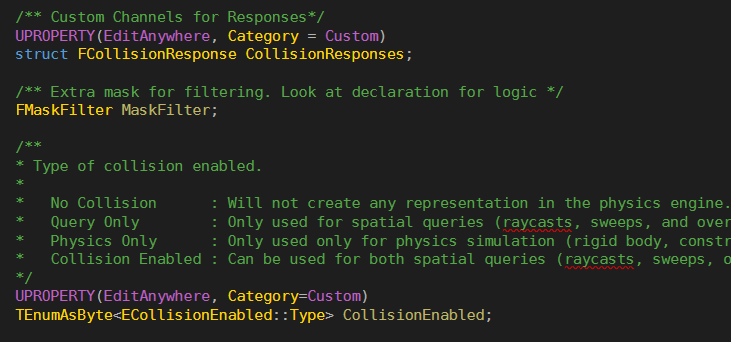
相关的重要函数



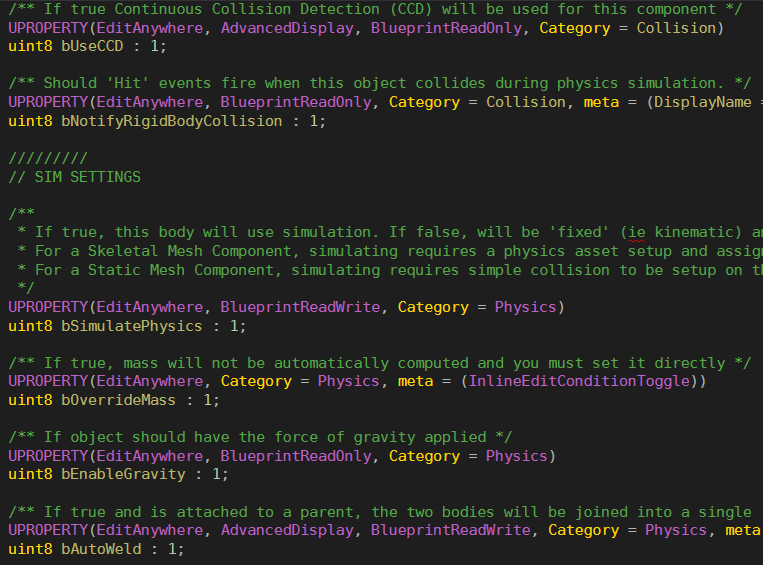
FBodyInstance位于Engine\Source\Runtime\Engine\Classes\PhysicsEngine\BodyInstance.h中，它是一个object的物理表示容器



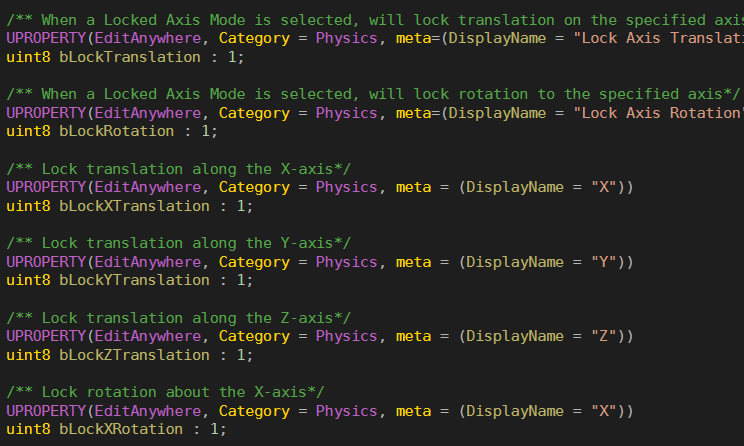
相关碰撞通道的成员变量



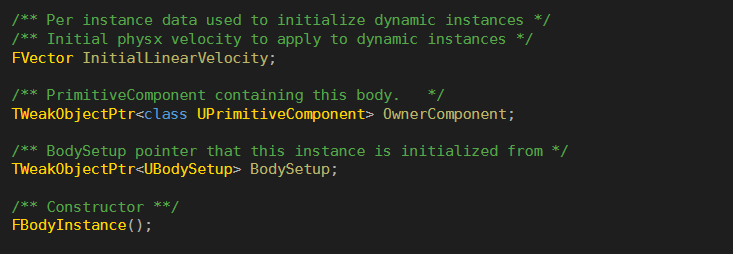
各种其他物理属性控制状态



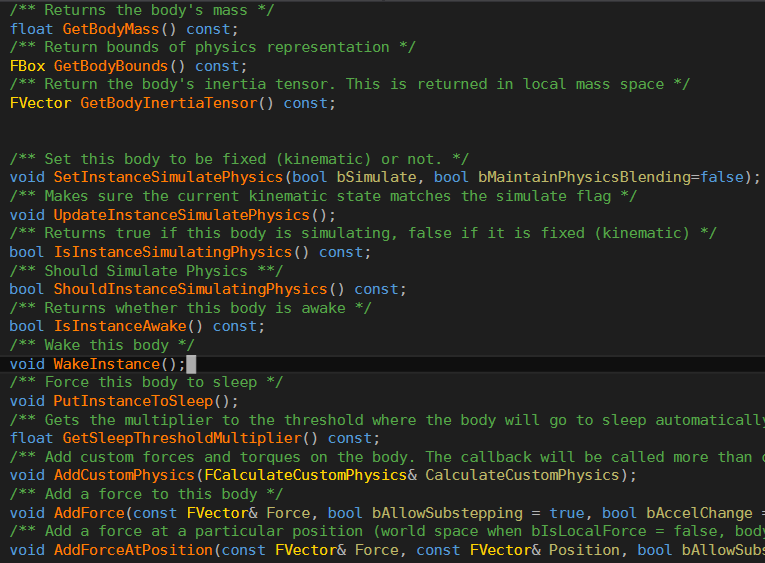
相关约束的控制变量例如约束在X轴上的移动等

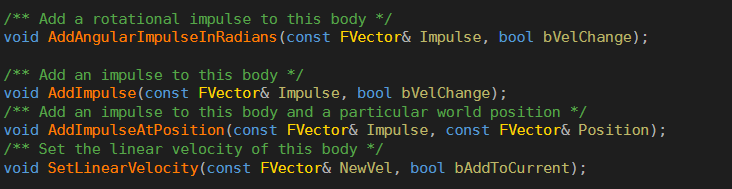


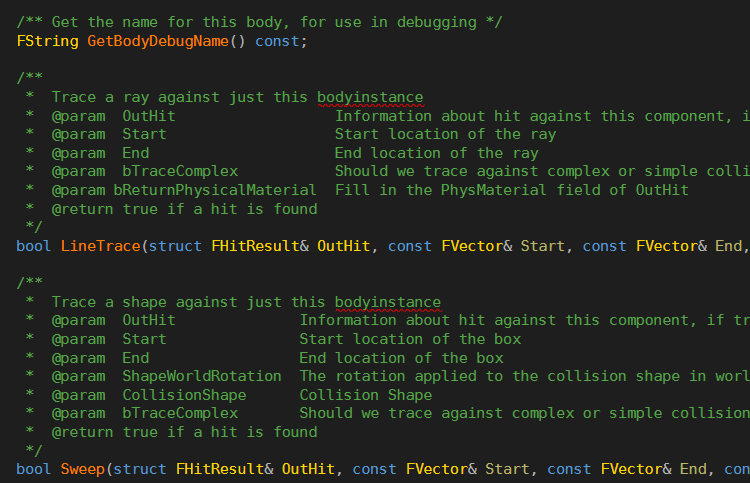
BodyInstance也有一个BodySetup的引用



诸多的方法



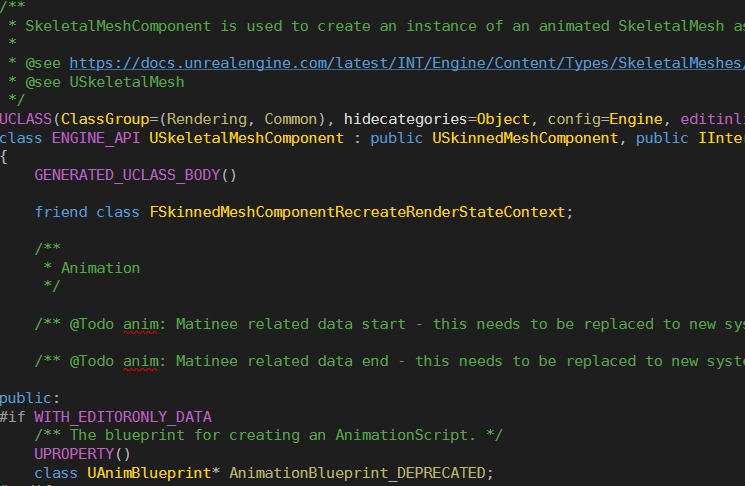




以上可以得知FbodyInstance是物理功能具体的调用集成类,几乎包括了所有可能的物理操作.

类似的分析SkeletalMeshComponent

SkeletalMeshComponent主要用于创建包含动画的SkeletalMesh资产

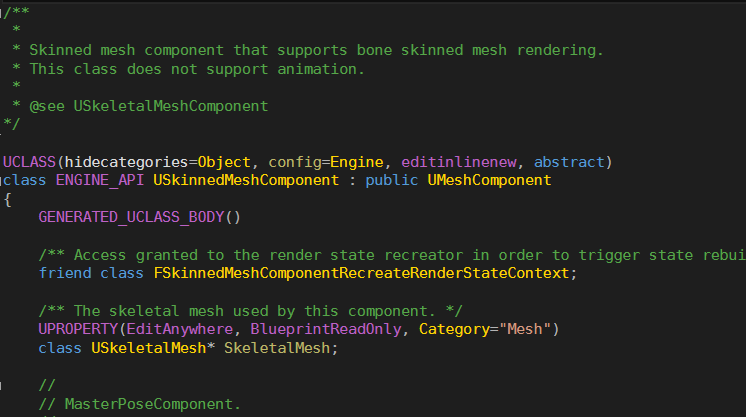


它的父类之一USkinnedMeshComponent

位于Engine\Source\Runtime\Engine\Classes\Components\SkinnedMeshComponent.h

它的布局与StaticMesh较为相似，与子类（SkeletalMeshComponent）的区别可能只有动画的支持与否

它也有类似UStaticMesh的USkeletalMesh



USkeletalMesh位于Engine\Source\Runtime\Engine\Classes\Engine\SkeletalMesh.h

但与UStaticMesh不同的是USkeletalMesh并不是一个单一的资产，而是类似很多的StaticMesh的集合的存在，因为骨架网格可以看做是很多静态网格拼接组成的

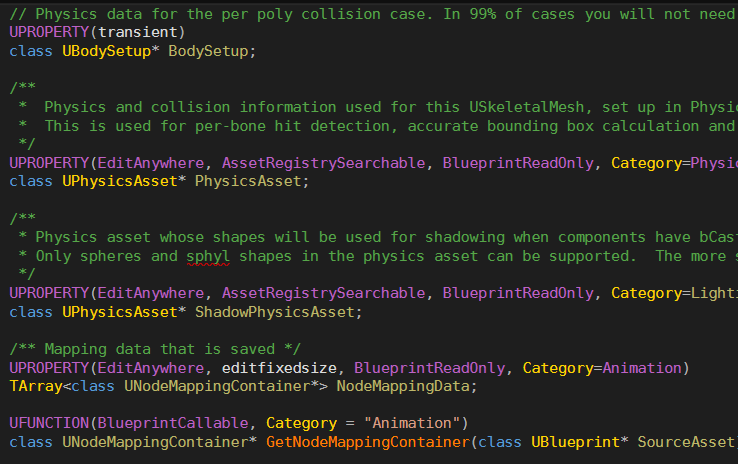
骨架网格由两部分组成：组成网格表面的一组多边形和可用于动画多边形的分层骨架

USkeletalMesh是支持动画的



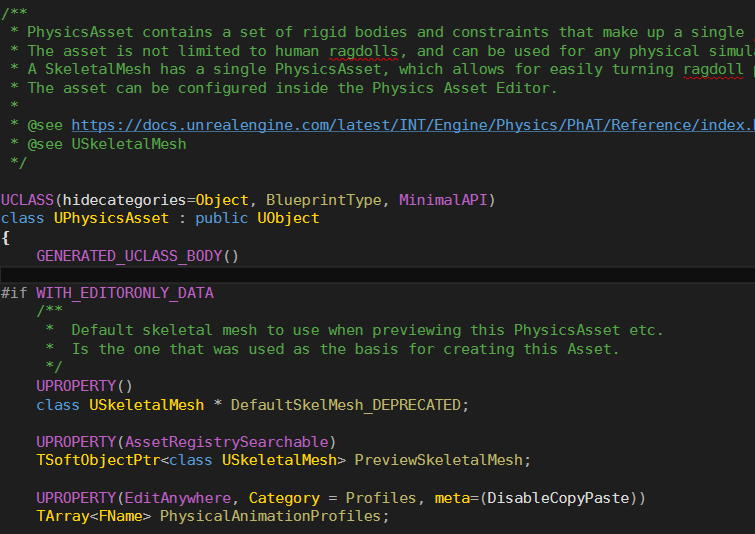
相比UStaticMesh，多了不少数据: 例如PhysicsAsset

注释中: PhysicsAsset用于这个SkeletalMesh的物理和碰撞信息，在Physics Asset编辑器中设置，主要用于精确检测具体命中的骨骼



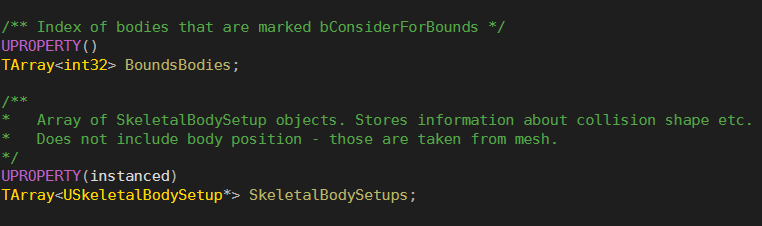
PhysicsAsset位于Engine\Source\Runtime\Engine\Classes\PhysicsEngine\PhysicsAsset.h

注释: 物理资产包含一组刚体和约束从而构成了一个单独的人偶，该资产不限于人形人偶，还可以用于任何其他身形和约束的物理仿真，SkeletalMesh具有一个PhysicsAsset，这允许对许多SkeletalMeshComponents轻松打开或关闭物理

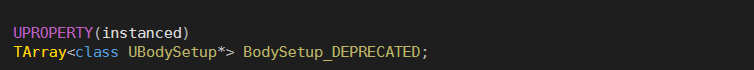


因为可能不止一个静态网格,以及动画数据(数据相对较大)处理相对复杂才会有这个PhysicsAsset中间类产生(与之前的StaticMesh相比)

因此它有一个USkeletalBodySetup的数组，这个就是UBodySetup在Skeletal的表现



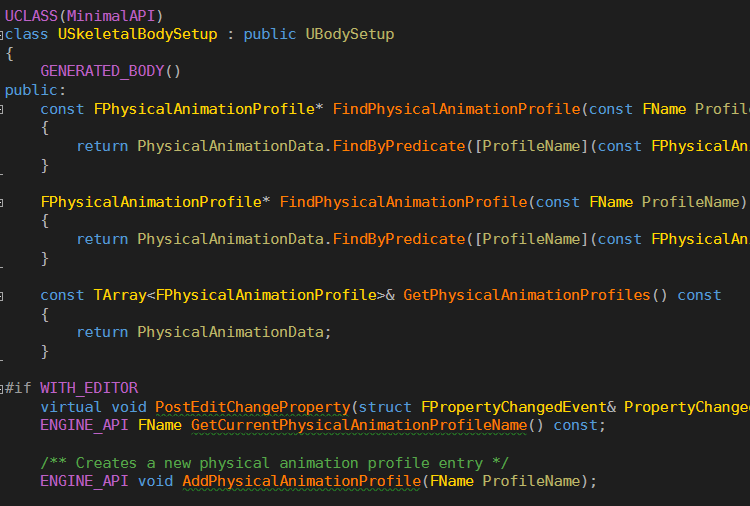
类的最后，还有不推荐使用的UBodySetup的数组，原本是没有USkeletalBodySetup类的存在的,后来将Skeletal相关的特例化从而形成了新的类，渐渐在Skeletal中舍弃UBodySetup而使用专门的USkeletalBodySetup，但实际上两者相差不大



USkeletalBodySetup

位于Engine\Source\Runtime\Engine\Classes\PhysicsEngine\PhysicsAsset.h

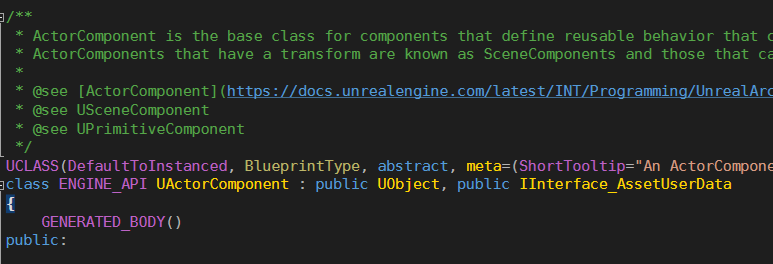
继承于UBodySetup，表面上看只是多了一些动画相关的成员，在此之后两种网格(StaticMesh和SketetalMesh)回归到了同一个底层FBodyInstance上了



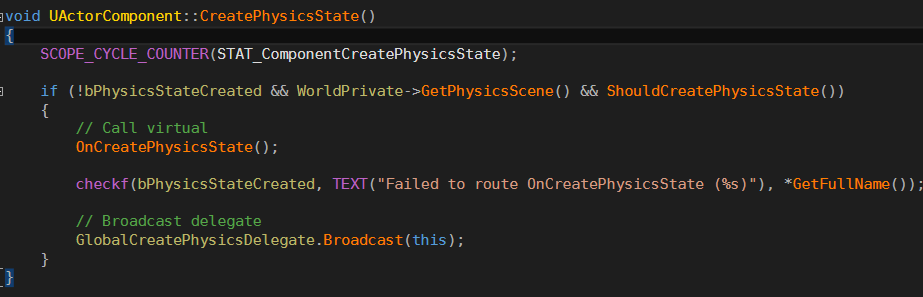
开关提到ActorComponent是两个网格类的父类

在GamePlay的框架中Actor会注册并初始化组件，而物理也是在此时进行创建的

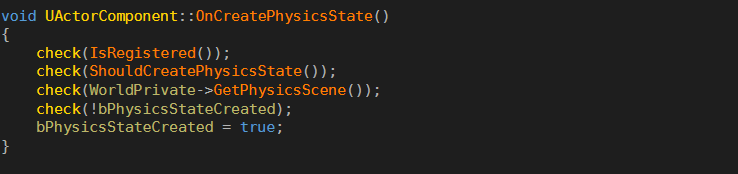
ActorComponent中有CreatePhysicsState()



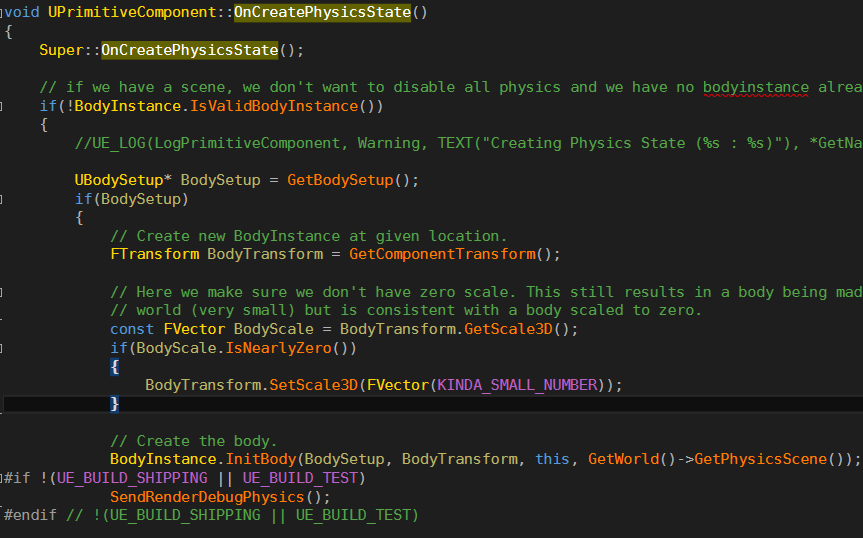
这个函数没有被子类重写过，因此其中的ShouldCreatePhysicsState()的返回值直接相关这个组件会不会创建物理

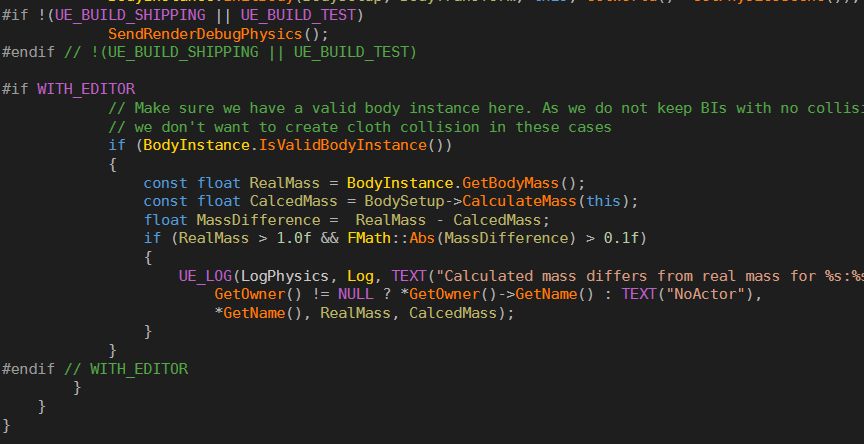


UActorComponent中的创建函数(OnCreatePhysicsState)



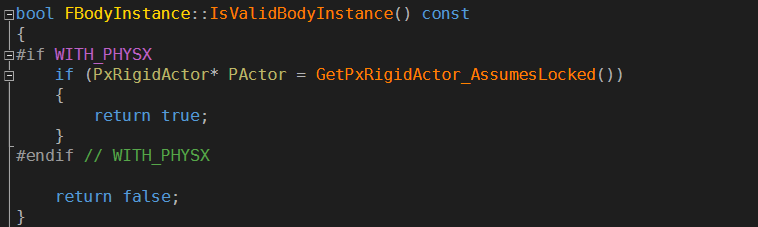
OnCreatePhysicsState在UPrimitiveComponent中被重写(因为UPrimitiveConponent是渲染的基本组件，相对从这里进行物理的创建比较合理)



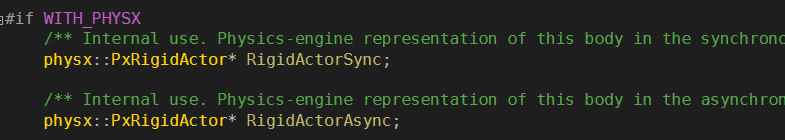


首先调用了父类的OncreatePhysicsState

如果能够获取一个PxRigidActor则继续下面的操作,否则就只是默认ActorComponent中的行为



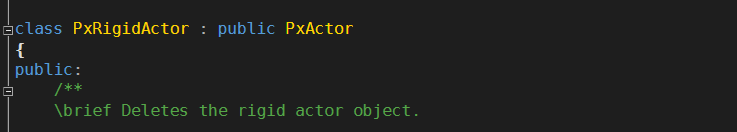
从给定的场景中返回PxRigidActor



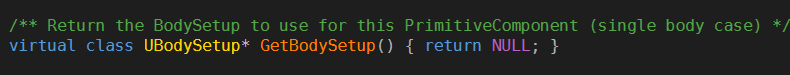


PxRigidActor位于Engine\Source\ThirdParty\PhysX3\PhysX\_3.4\Include\PxRigidActor.h

PxRigidActor表示在物理SDK中动态和静态刚体之间共享的基类, 通过定义一组附属的形状来指定对象的几何结构

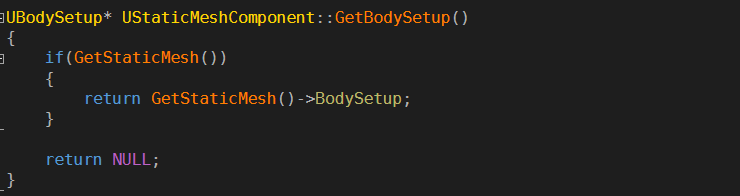


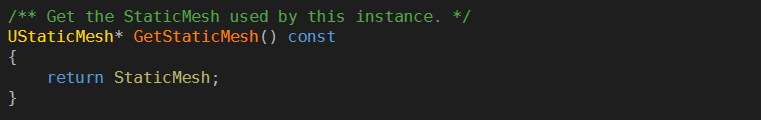
接下来是GetBodySetup()，在PrimitiveComponent中是返回NULL的，因此PrimitiveConponent组件的物理创建函数就此结束



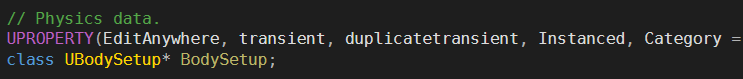
因此转到StaticMeshComponent中查看具体过程

首先GetStaticMesh(): 其实就是返回自身存储的StaticMesh(UStaticMesh\* )，进而返回其BodySetup



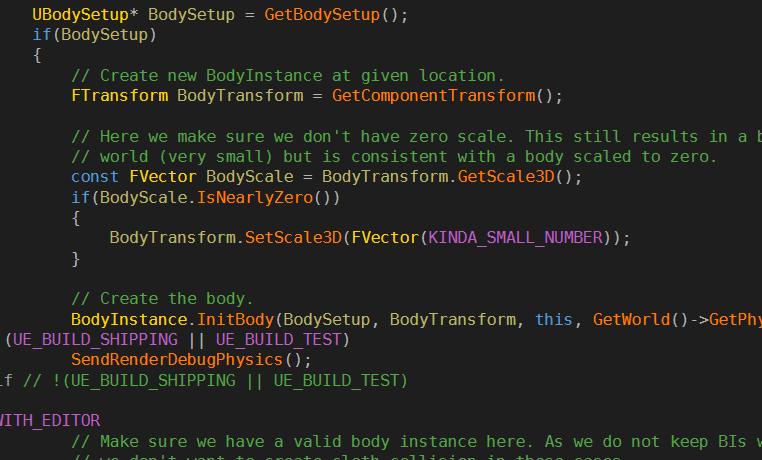


UStaticMesh中有BodySetup

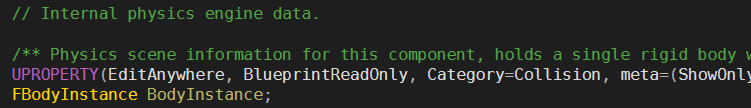


之后回到OnCreaterPhysicsState，现在已经获得了BodySetup，根据组件自身的位置等信息

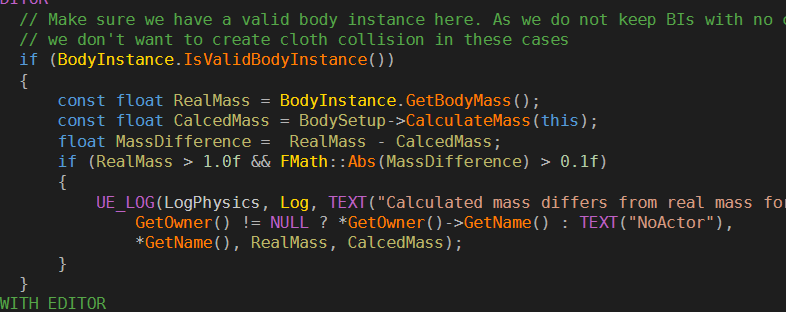
调用自身的bodyInstance的InitBody函数创建body



PrimitiveComponent中含有一个FBodyInstance

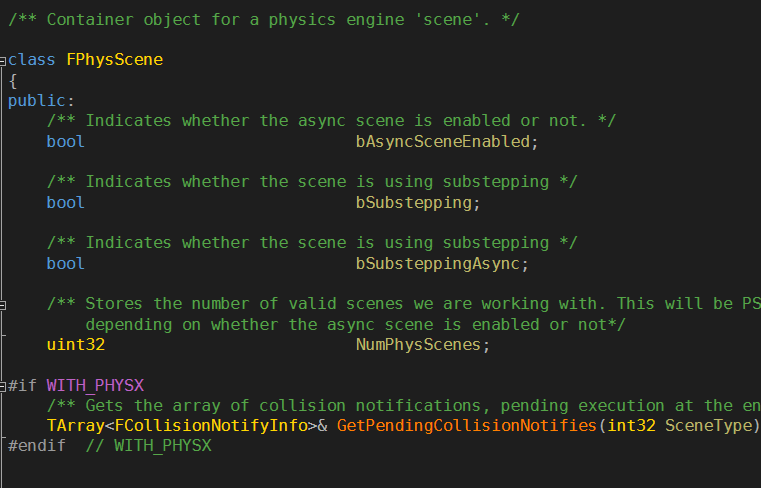


最后会再次检查BodyInstance的合法性，完成Create

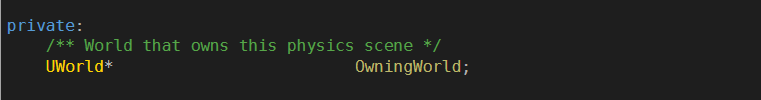


在BodyInstance.InitBody中存在一个参数是FPhysScene \* ，UWorld拥有其引用

位于Engine\Source\Runtime\Engine\Public\PhysicsPublic.h中



也有World的指针，同时类中也定义了变更World等方法



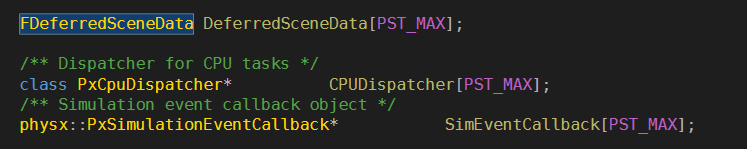
一些方法



有这样一个结构体



FPhysScene有相关的数据成员



因此大概猜想FPhysScene是UWorld的物理版(只关注物理对象)，FPhysScene会和UWorld进行数据上的同步。



