

**相关知识：**

虚幻类的命名前缀

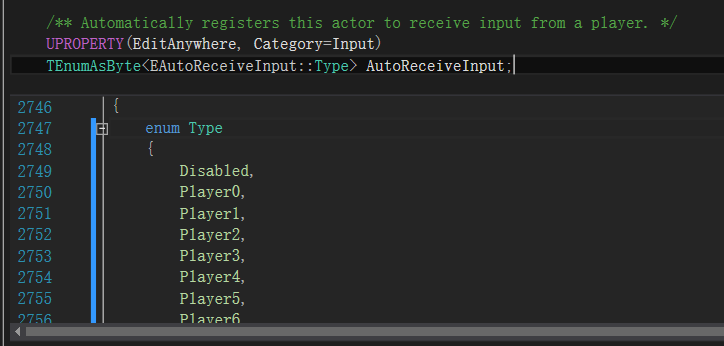
虚幻引擎提供了在构建过程中为您生成代码的工具。这些工具有一些类命名的期望，如果名称与预期不符，将会触发警告或错误。下面的类前缀列表描述了这些工具所期待的。

* 从**Actor（Class AActor）**派生的前缀要为**A**，例如AController。
* 从**Object（Class UObject）**派生的前缀要为**U**，例如UComponent。
* **Enums**类以为**E**前缀，例如EFortificationType。
* **Interface** 类通常以**I**为前缀，例如IAbilitySystemInterface。
* **Template**类以**T**为前缀，例如TArray。
* 从**SWidget (Slate UI)**派生的前缀要为**S**，例如SButton。
* 别的一切都以**F**为前缀，例如FVector。

Engine/GameFramework/Actor：

[AAcotr](http://api.unrealengine.com/INT/API/Runtime/Engine/GameFramework/AActor/index.html)：

Actor是可以在关卡中放置或产生的Object的基类。 Actor可能包含一系列ActorComponents，它们可以用来控制actor如何移动，如何渲染等等。Actor的另一个主要功能是在播放过程中通过网络复制属性和函数调用。



AutoReceiveInput属性：决定哪个player控制，其中TEnumAsByte是以类型安全的方式将枚举值存储为字节的模板，实际上作用就是在enum之上封装了安全性的一层，[具体的类说明](http://api.unrealengine.com/INT/API/Runtime/Core/Containers/TEnumAsByte/index.html)。

**相关知识：**

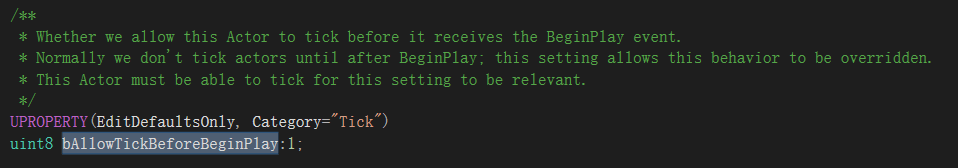
看到上面有UPROPERTY(EditAnywhere, Category=Input) 这里是UE4的反射机制：注册这个属性到反射系统，使对其可见，即是在编辑器中显示这个属性，并能够对该属性进行其他管理，比如垃圾回收（GC），【它根据root set（Singleton，管理着所有有效的索引）来进行垃圾回收操作的，通过UPROPERTY（）宏使root set保存有这个属性的引用，每次GC时就不会将其视为垃圾处理】

EditAnywhere参数说明表示该属性可以通过属性窗口，原型和实例进行编辑，并指定所在的Category为Input，也就是说可以在UE4编辑器中这个Actor的详细信息中的名为“Input”的类目中找到这个属性。

引用官方说明：反射是程序在运行时检查自身的能力。这非常有用，是虚幻引擎的基础技术，为编辑器，序列化，垃圾收集，网络复制和Blueprint / C ++通信等许多系统提供动力。然而，C ++本身并不支持任何形式的反射，所以虚幻拥有自己的系统来收集，查询和处理关于C ++类，结构体，函数，成员变量和枚举的信息。我们通常将反射称为属性系统，因为反射也是图形术语。

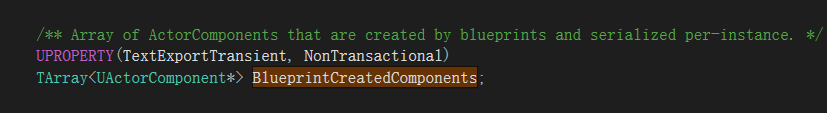
类似的还有：

* **UCLASS（）** - 用于告诉Unreal为一个类生成反射数据。该类必须来自UObject。
* **USTRUCT（）** - 用于告诉Unreal为结构生成反射数据。
* **GENERATED\_BODY（）** - UE4将其替换为为该类型生成的所有必需的样板代码。
* **UPROPERTY（）** - 使用UCLASS或USTRUCT的成员变量作为UPROPERTY。UPROPERTY有很多用途。它可以允许变量被复制，序列化和从蓝图访问。它们也被垃圾收集器用来跟踪UObject的引用数量。
* **UFUNCTION（）** - 启用UCLASS或USTRUCT的类方法作为UFUNCTION。UFUNCTION可以允许从Blueprints中调用类方法，并用作RPC等等。
* 它们的说明符表如下：
* [UCLASS说明符列表](https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Classes/Specifiers)
* [UPROPERTY说明符列表](https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Properties/Specifiers)
* [UFUNCTION指示符列表](https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Functions/Specifiers)
* [USTRUCT说明符列表](https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Structs/Specifiers)

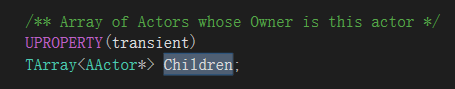


是否允许此Actor在收到BeginPlay事件之前进行tick的开始。

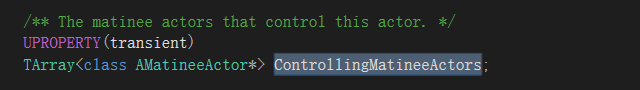
Tick这里是tick()函数，即每一帧的处理



由蓝图和序列化每个实例创建的ActorComponents数组。

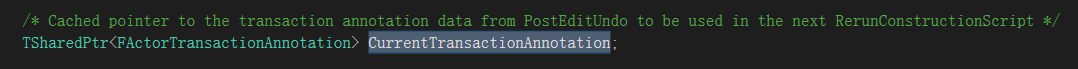


该Actor的子Actor数组

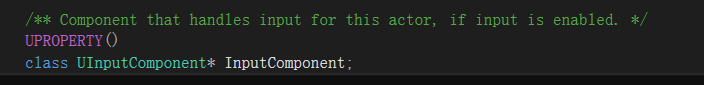


控制这个Actor的 MatineeActor 数组

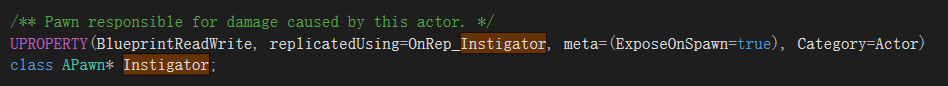
**Matinee序列主要用来描述轨迹，可用于动画**



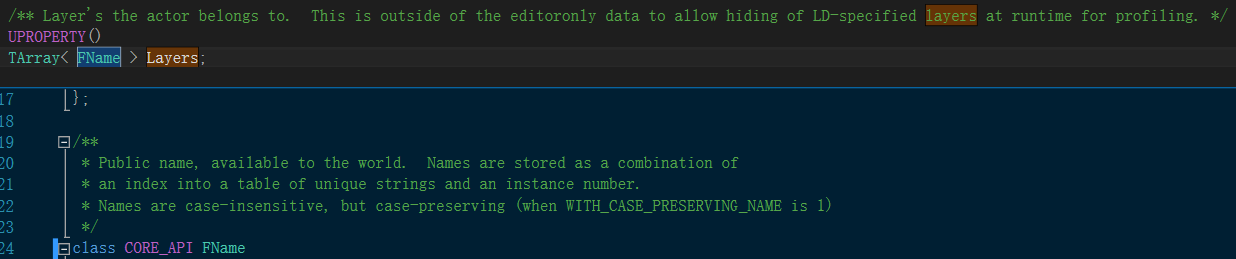
CurrentTransactionAnnotation：缓存指向PostEditUndo的事务注释数据的指针，以便在下一个RerunConstructionScript中使用



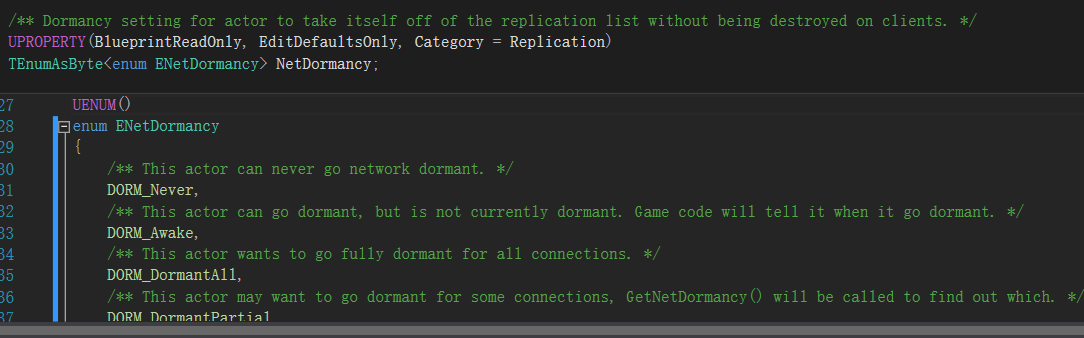
InputComponent：如果启用了输入，索引的控制组件能处理此Actor的输入



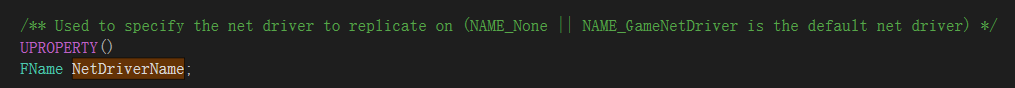
Instigator：是导致伤害（damage）的发起者



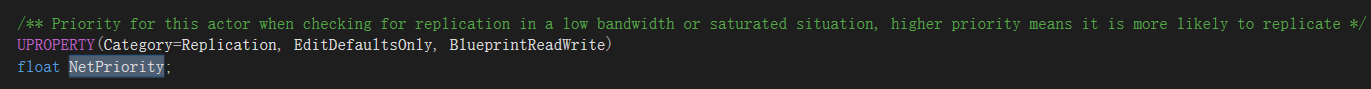
Layers: 此Actor所属的Layers，FName是World中可见的可索引的index



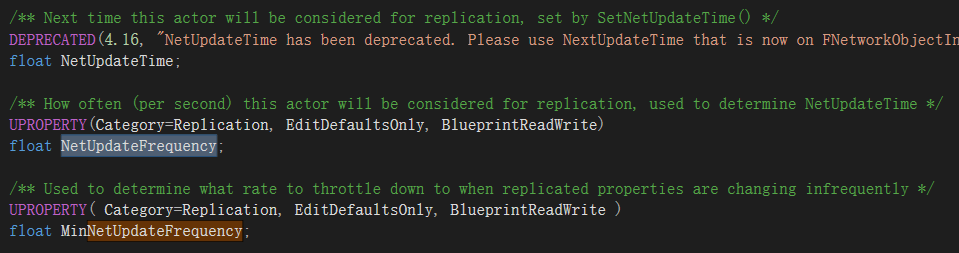
NetDormancy网络休眠：将Actor从网络复制中移除但并不需客户端上执行destroy



NerDriverName用于指定要复制的网络驱动程序（NAME\_None || NAME\_GameNetDriver是默认的网络驱动程序）

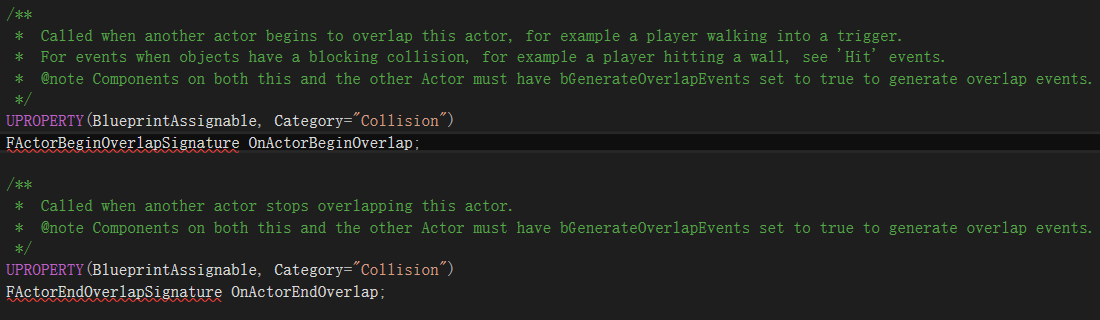


NewPriority当在低带宽或饱和情况下检查复制时优先考虑这个参与者，更高的优先级意味着它更可能复制



**NetUpdateFrequency**多久（每秒）该角色将被考虑用于复制，用于确定NetUpdateTime

**MinNetUpdateFrequency**用于确定在复制属性不经常更改时要降低的速率

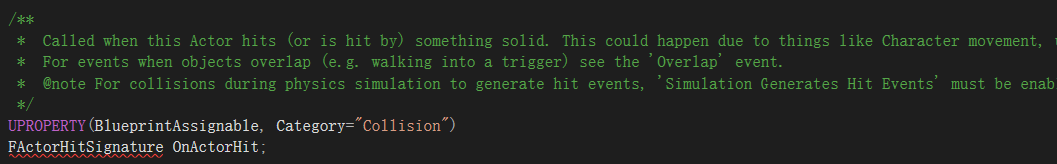


相关碰撞状态两个成员：开始重叠和结束重叠

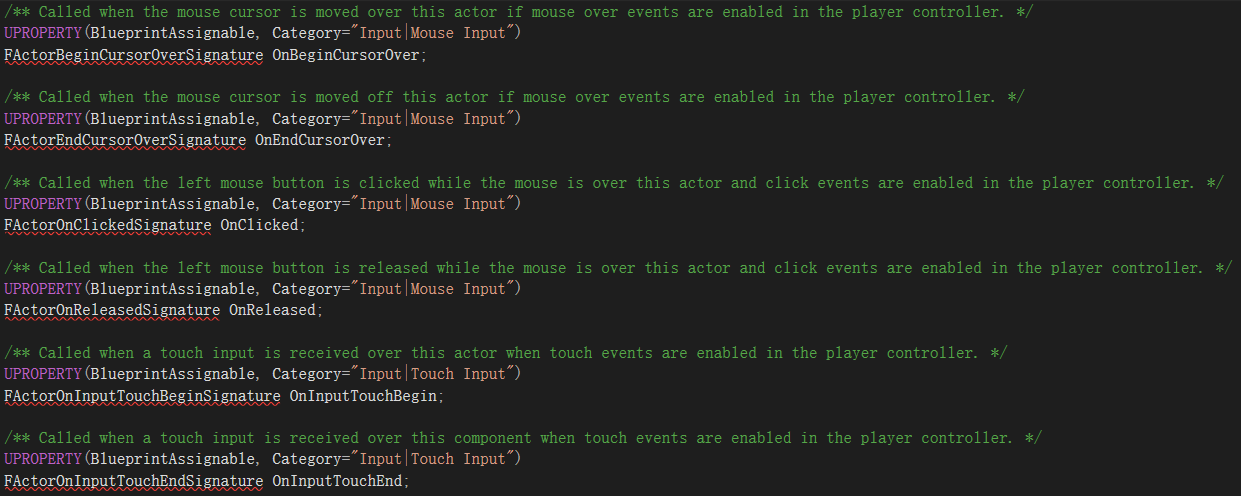
注意：两个碰撞的Actor的相关组件的bGenerateOverlapEvents设置为ture，才能产生重叠事件

FActorEndOverlapSignature等是一个Delegates,[参考资料](https://wiki.unrealengine.com/Event_handling)

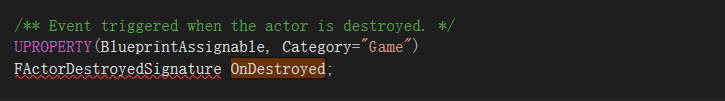
* **FActorBeginOverlapSignature:** void ( AActor\* OtherActor )
* **FActorEndOverlapSignature:** void ( AActor\* OtherActor )
* **FActorHitSignature:** void ( AActor\* SelfActor, AActor\* OtherActor, FVector NormalImpulse, const FHitResult& Hit )



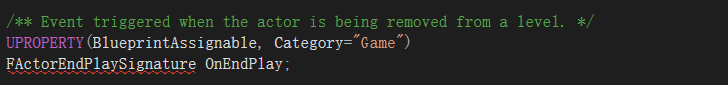
在Simulation Generates Hit Events启用的前提下overlap会触发这个成员



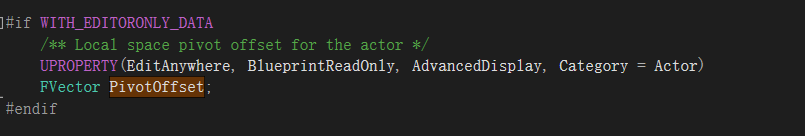
类似的还有其他的输入响应成员函数声明



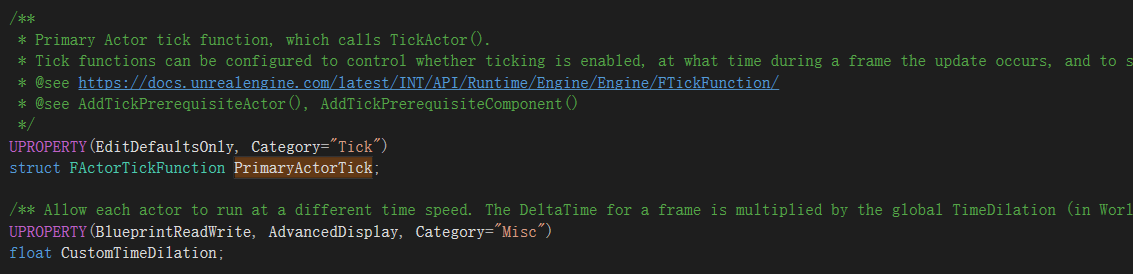
当Actor被销毁时触发的事件



当Actor从level被移除触发的事件

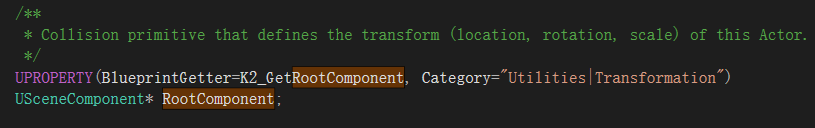


只是针对UE编辑器的属性：Actor的本地空间枢轴偏移量



PrimaryActorTick，调用TickActor()

CustomTimeDilation是这个Actor自用的时间间隔



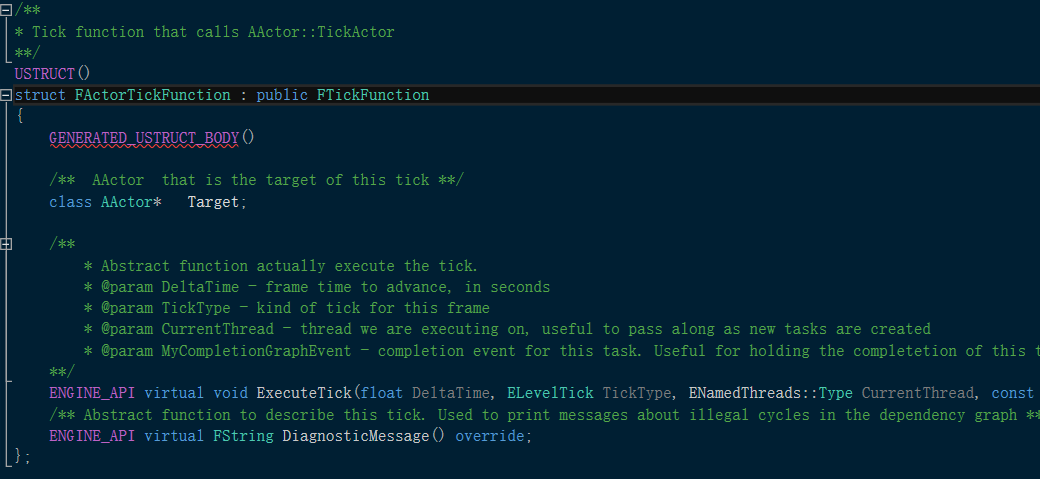
RootComponent定义了这个Actor的变换（位置，旋转，缩放）的碰撞基元。



AAcor的带参构造函数：首先先对PrimaryActorTick（主要相关每帧动作函数tick()）进行设置，【注意：如果是一些静态物体（没有tick的需要），最好false掉tick以提高运行效率】，而这里默认已经no tick。后面有很多的属性初始化

-------------------------以下关于Tick以及FTickFunction的相关分析----------------------

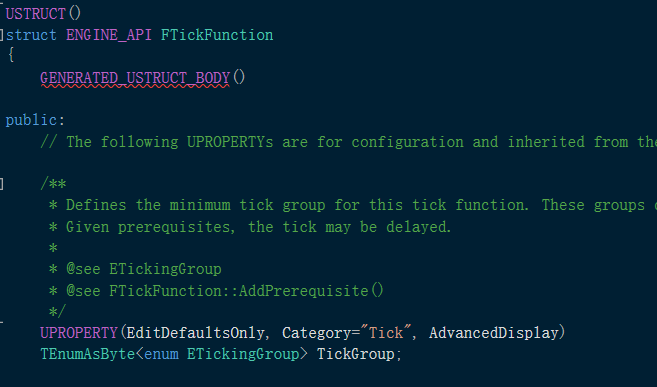
关于**FActorTickFunction**的定义：



FTickFunction结构体：[官方文档](http://api.unrealengine.com/INT/API/Runtime/Engine/Engine/FTickFunction/index.html)

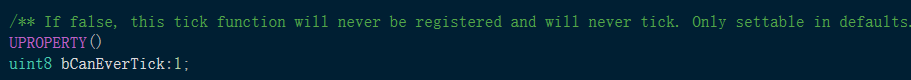
定义在Engine/EngineBaseType.h

是所有tick()的基类

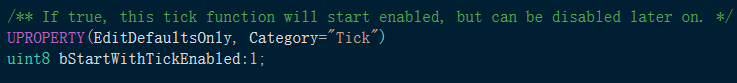


TickGroup:

定义此tick函数的最小tick组。这些组决定了帧更新期间对象的相对顺序。根据先决条件，tick可能会延迟。



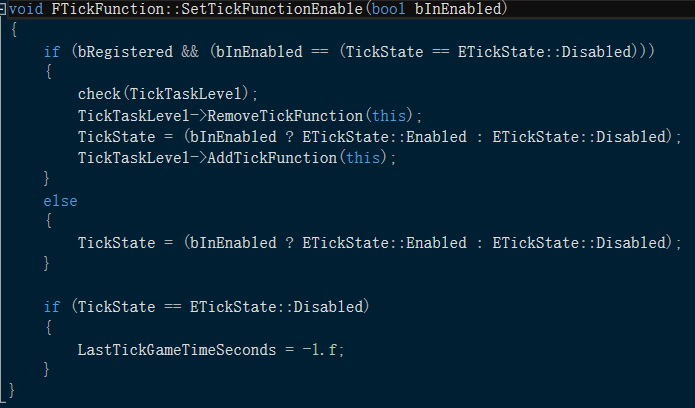
bCanEverTick是一个重要的tick注册开关参数，如果false，则永远禁用tick，并且不可更改



bStartWithTickEnabled是tick是否可以开始的参数，是tick注册之后的开关





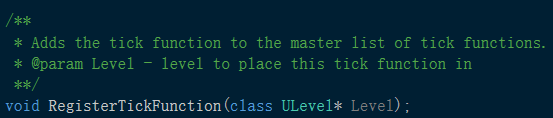


TickTaskLevel成员：如果这个tick函数已经注册则返回注册的FTickTaskLevel指针。

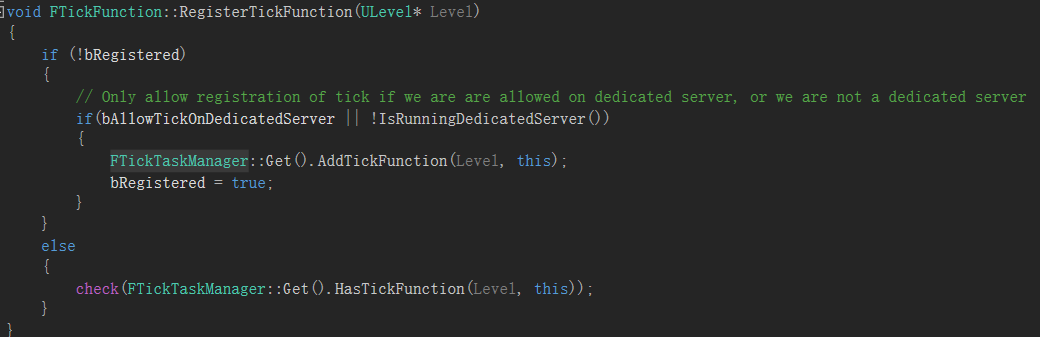
FTickTaskLevel是针对一个Ulevel的tick结构体（后面还会有说明）

**相关知识：**

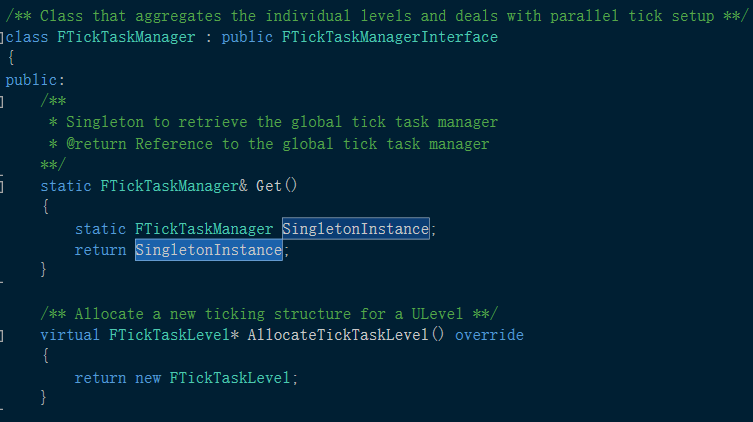
这里提及到的ULevel，一个游戏是一个UWord（单例），它有ULevel的索引，这些ULevel即是一个一个“关卡”相当于一个独立的场景区域，一次加载一个ULevel，这个Ulevel完成之后可以设置进入下一个ULevel



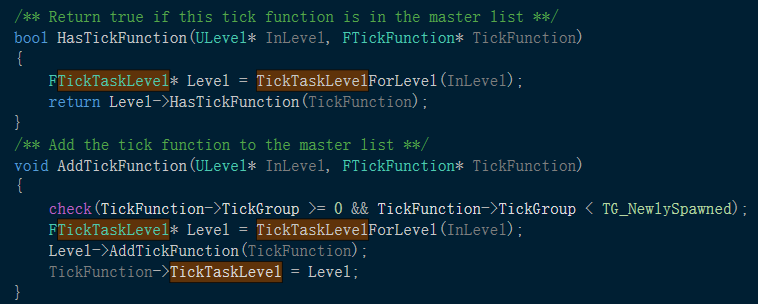
可以看到，tick函数是注册是相关ULevel的



通过管理着levels的FTickTaskManager实现

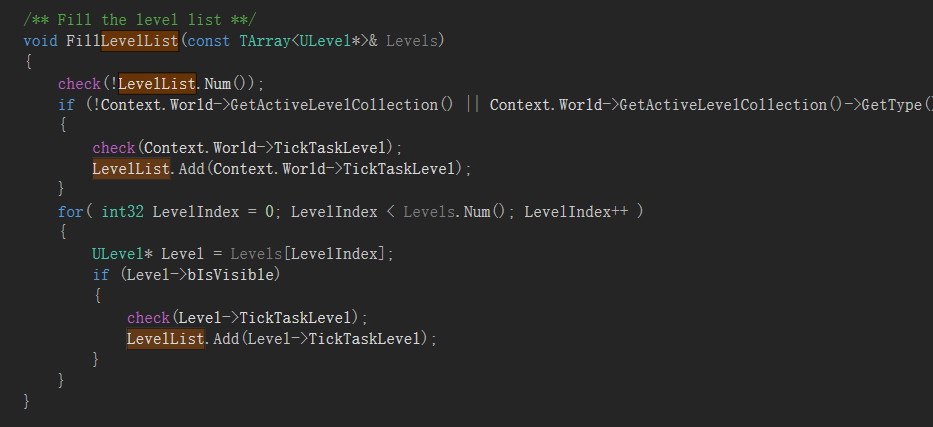


而FTickTaskManager用来聚集levels并且处理并行tick的设置，单例模式

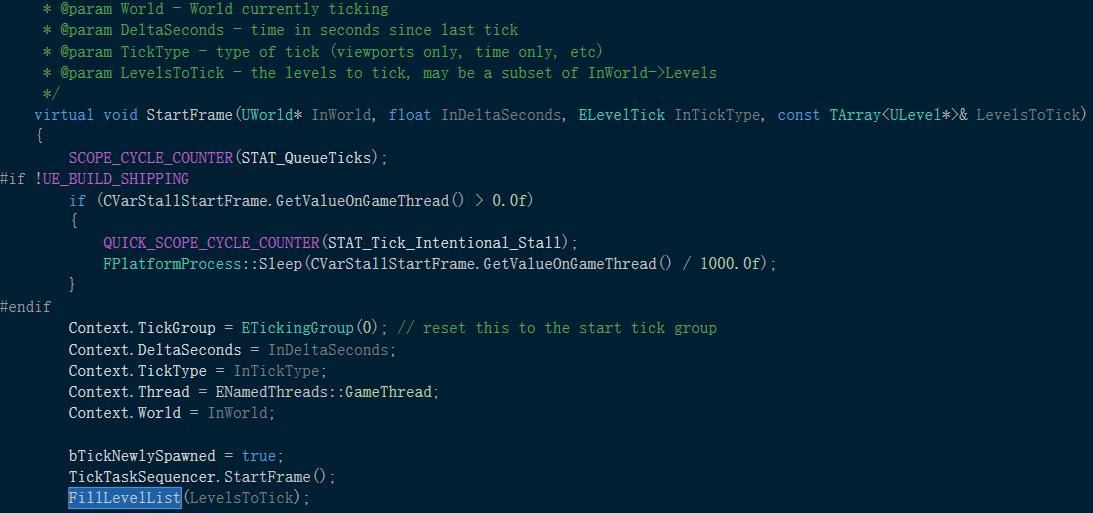




接着之前提到的FTickTaskLevel，FTickTaskManager内有以上方法，并有存储其索引的数组LevelList从而管理它们的Tick



但是在创建FTickTaskLevel并没有将其直接纳入LevelList之下，而是通过在本类(FTickTaskManager) StartFrame中调用FillLevelList函数集中填充



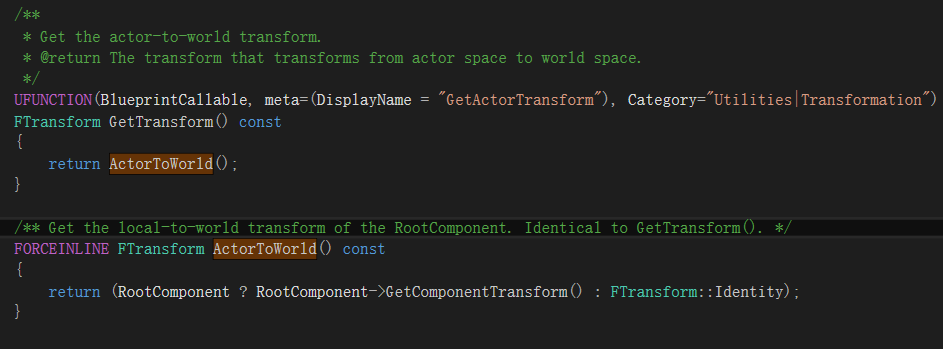
StartFrame中LevelsToTick即为需要tick的ULevel数组，StartFrame对每一个TickGroup只调用一次

到此总结一下：

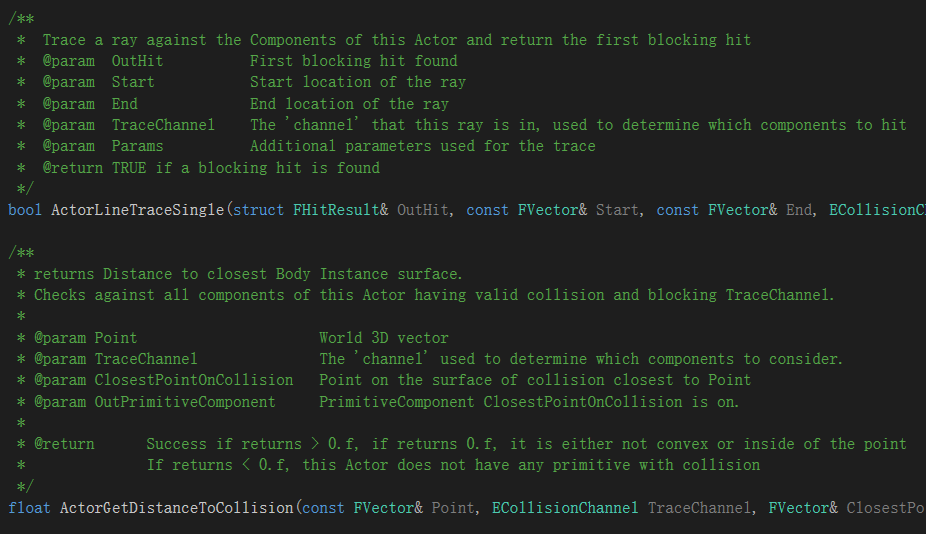
FTickFunction中要注册一个tick函数，通过调用FTickFunction::RegisterTickFunction(ULevel\*)，在函数中会去检查是否已经被注册过了（通过bRegistered）,如果没有被注册，通过FTickTaskManager::AddTickFunction(ULevel\*,FTickFunction\*)创建一个FTickTaskLevel并注册这个FTickFunction最后将FTickFunction ->TickTaskLevel赋值(注意：它们几个类之间是friend class)之后便能正常使用FTickFunction:: SetTickFunctionEnable了。

-------------------------以上关于Tick以及FTickFunction的相关分析----------------------

接着回来分析AActor

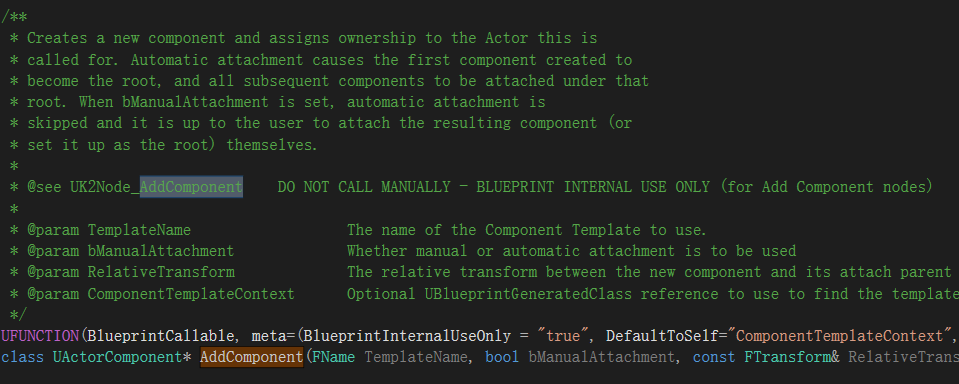


ActorToWorld():进一步说明Actor的坐标是以RootComponent为准的，用来获得Acotr的Transform



相关碰撞的两个函数ActorLineTraceSingle和ActorGetDistanceToCollision，第一个函数跟踪该Actor的组件并返回第一个阻挡命中，如果发现碰撞命中，则为TRUE

第二个函数返回到最近的实体表面的距离。检查该Actor的所有组件是否有有效的碰撞并阻塞TraceChannel



添加一个组件是Actor的重要功能之一，组件是用来丰富Actor的行为的

AddComponent：创建一个新组件，并将所有权分配给要调用的Actor。Automatic attachment会导致创建的第一个组件成为RootComponent，并且所有后续组件都将attach到该RootComponent下。如果设置了bManualAttachment，则会跳过Automatic attachment，并由用户自行attach生成的组件

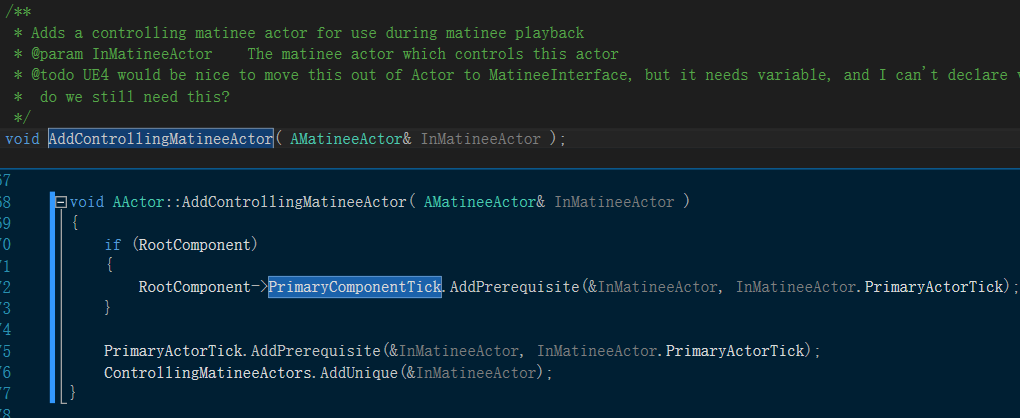
第一个参数是FName类型，之前提过是一个全局的命名，存储要添加的组件的名字

第二个参数决定是否手动attach

第三个参数是新组件与其附属父级之间的相对转换（仅限自动）

第四个参数用于查找模板的可选UBlueprintGeneratedClass引用。如果为null（或不是BPGC），则在此Actor的类中查找组件

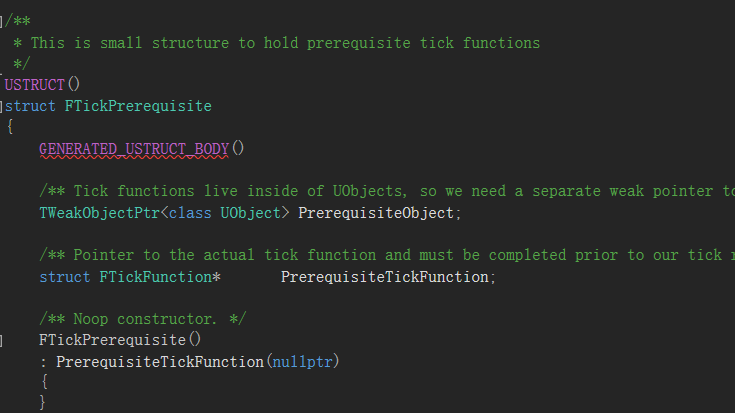
其实现大概步骤是通过查找组件名来创建一个UActorComponent然后声明一下这个组件的存在NewActorComp->OnComponentCreated()，之后通过cast转换为USceneComponent类型，如果！bManualAttachment则自动为RootComponent【创建第一个组件】并设置NewSceneComp->SetRelativeTransform(RelativeTransform)，之后NewActorComp->RegisterComponent()进行组件和注册，这会创建它的渲染/呈现状态，中间还有Word->UpdateCullDistanceVolumes操作，最后返回这个NewActorComp



AddControllingMatineeActor()：添加控制MatineeActor以供Matinee播放期间使用。

如果已有RootComponent，则同时调用RootComponent的PrimaryComponentTick的AddPrerequisite方法（）即添加一个 tick function to the list of prerequisites.换句话说adds the requirement that TargetTickFunction is called before this tick function is。

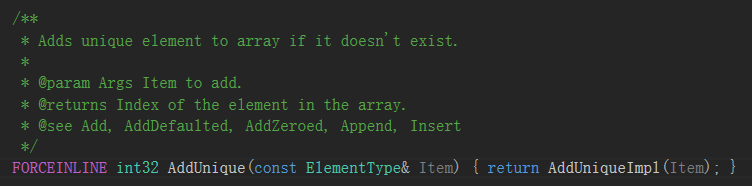




Prerequisites即是一个管理“先决条件”的FTickPrerequisite的数组类型

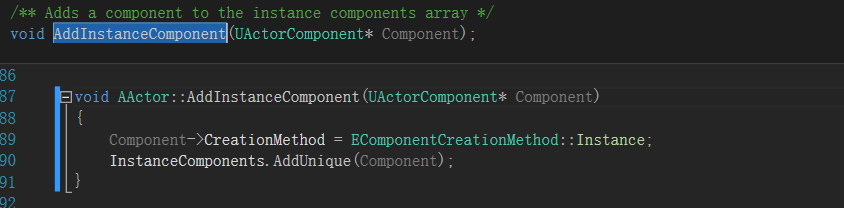
**相关知识：**

Tarray.h中

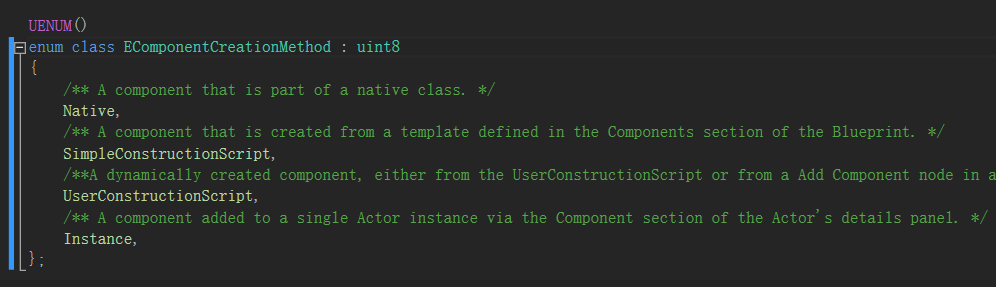


AddUnique()：如果不存在则添加，并返回这个Item的索引

低层使用TArray::Find()实现方式是通过对比两个的索引值因此同一种但不同的实例在AddUnique看来是不同的

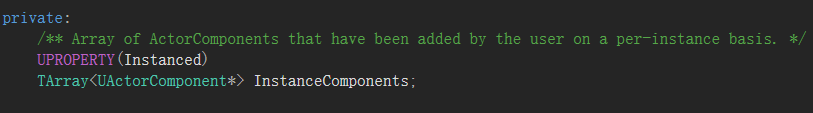


**AddInstanceComponent**向实例组件数组添加一个组件：首先设置了Component的CreationMethod，它的声明是EComponentCreationMethod CreationMethod;

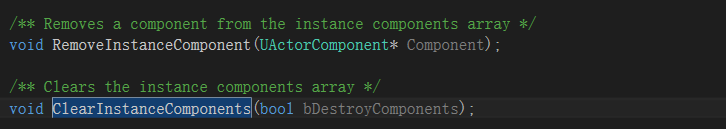


即首先将这个要被添加的Component标志为Instance【通过Actor详细面板的组件部分添加到单一实例Actor上（形似UE编辑器中直接在Actor面板上添加组件）】

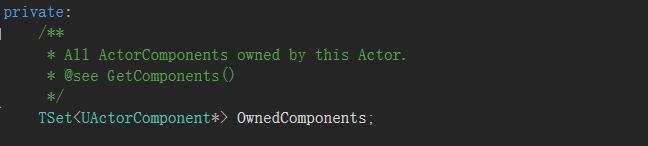
接着向AActor的InstanceComponents以unique的方式添加这个Component

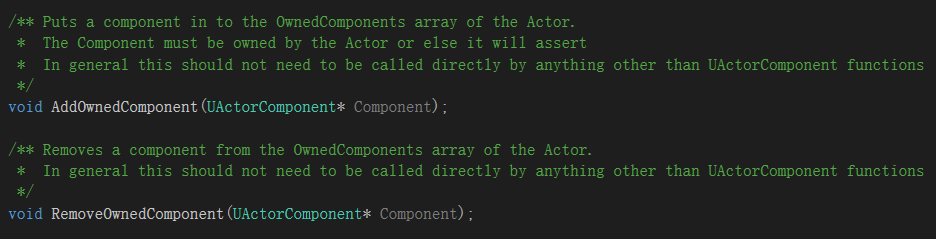


类似的



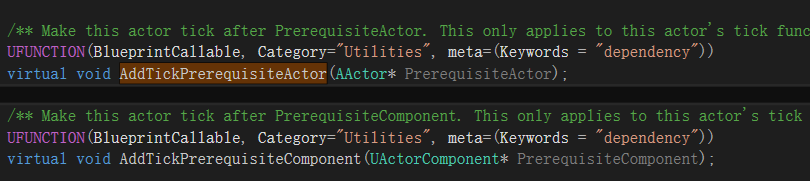
RemoveInstanceComponent和ClearInstanceComponents(省略)

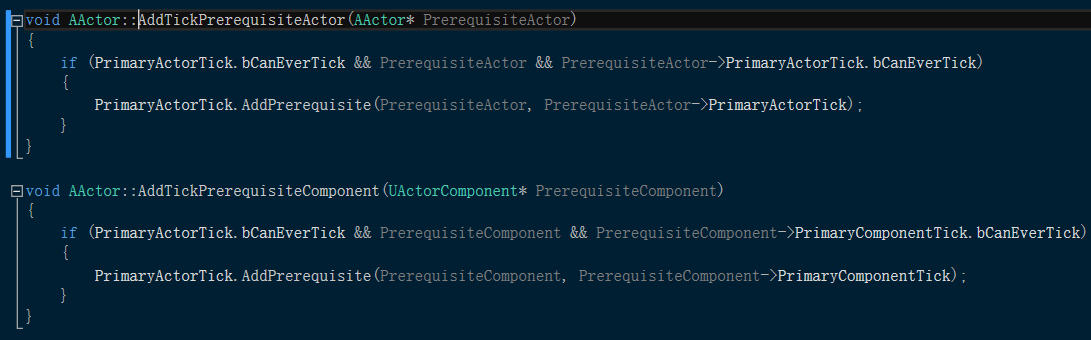




这里有OwnedComponent，也有相应的增删方法（通常被UAcotrComponent调用）

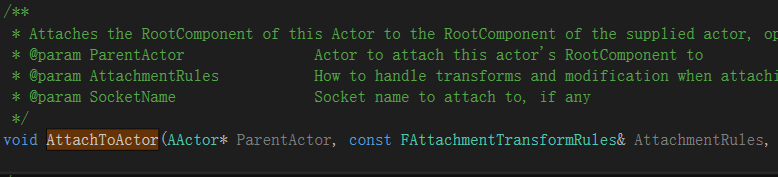
从注释上理解OwnedComponent是Actor的所有的自身的Component。它是最全的Component索引





AddTickPrerequisiteActor，AddTickPrerequisiteComponent主要用于在这个Actor本身进行Tick之前添加一个PrerequisiteActor或者PrerequisiteComponent

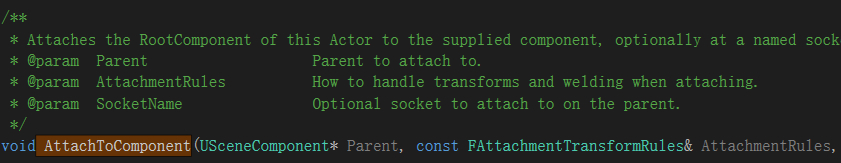
AddPrerequisite方法（）即添加一个 tick function to the list of prerequisites.换句话说adds the requirement that TargetTickFunction is called before this tick function is。



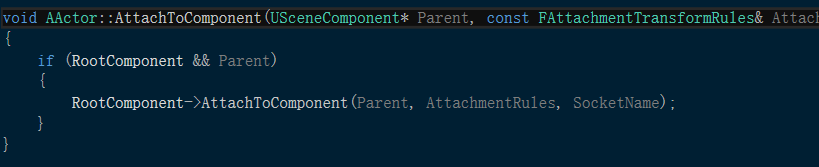
AttachToActor: 将此Actor的RootComponent附加到提供的actor的RootComponent上，可选地在指定的套接字上



实际上是调用了AttachToComponent()



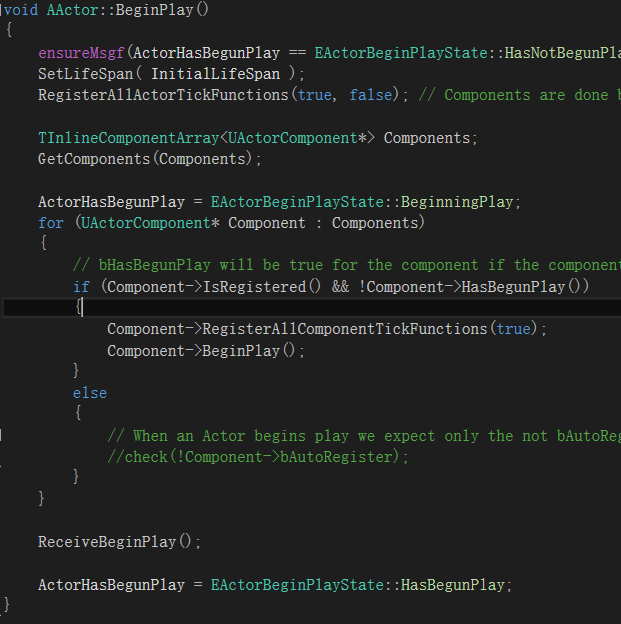
将此Actor的RootComponent附加到提供的组件，可选地位于指定的套接字上。对未注册的组件调用此方法是无效的。



实际上是调用了这个Actor的RootComponent的AttachToComponent()（具体省略）

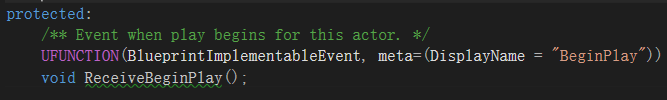


BeginPlay()：开始游玩时的事件处理（相对应于蓝图的BeginPlay事件）

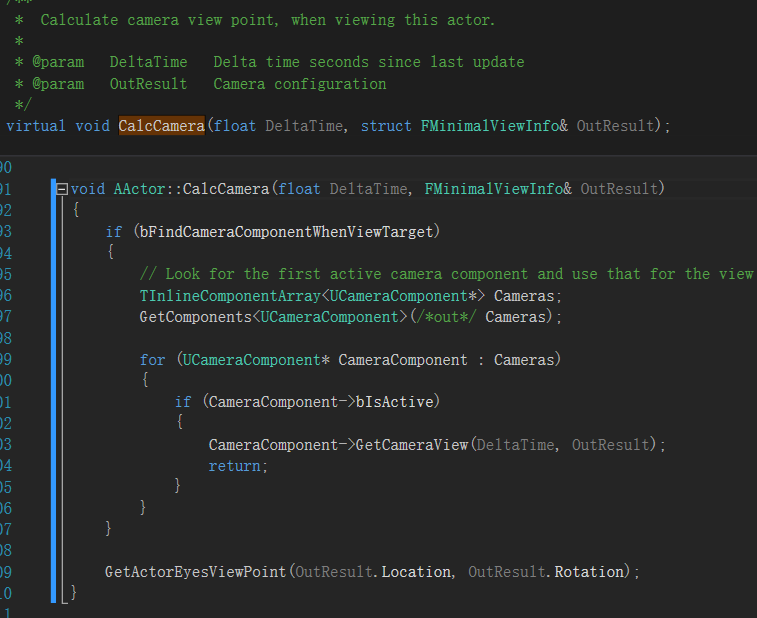


进行了一些初始化（Tick，Component等）

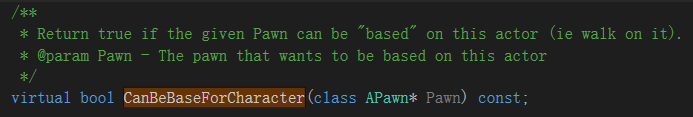
之后调用了ReceiveBeginPlay（）



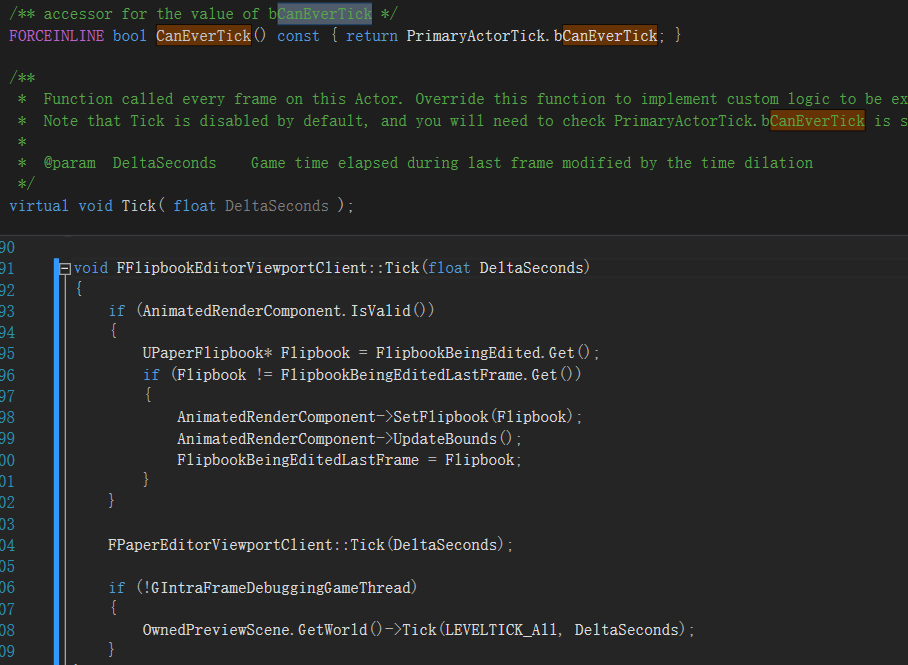
ReceiveBeginPlay：可以看到是BlueprintImplementableEvent，即可在蓝图中进行实现



CalcCamera()：当观察这个Actor时计算相应的camera view point。

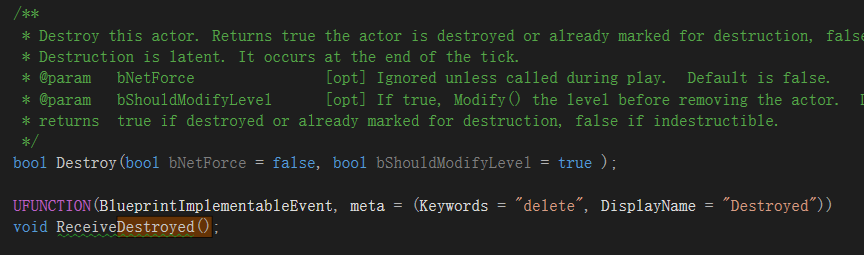


如果Pawn能够“based”在这个Actor上则返回Ture



Tick函数，即每一帧Actor的处理，默认是tick禁用的，需要使用设置PrimaryActorTick中的bCanEverTick为true





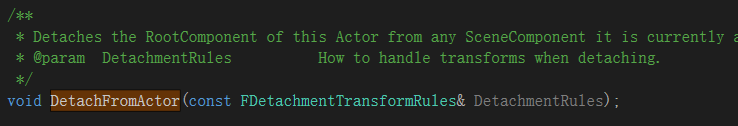


对于一个UObject被决定释放后不会立刻被回收，而是等待下一次的GC时才真正被回收，而这一期间的状态是PendingKill，可以通过IsPengdingKillPending()来检查，同样，对于一个指针，则会将其赋值为nullptr

Destroy函数中通过World来对这个Actor进行销毁

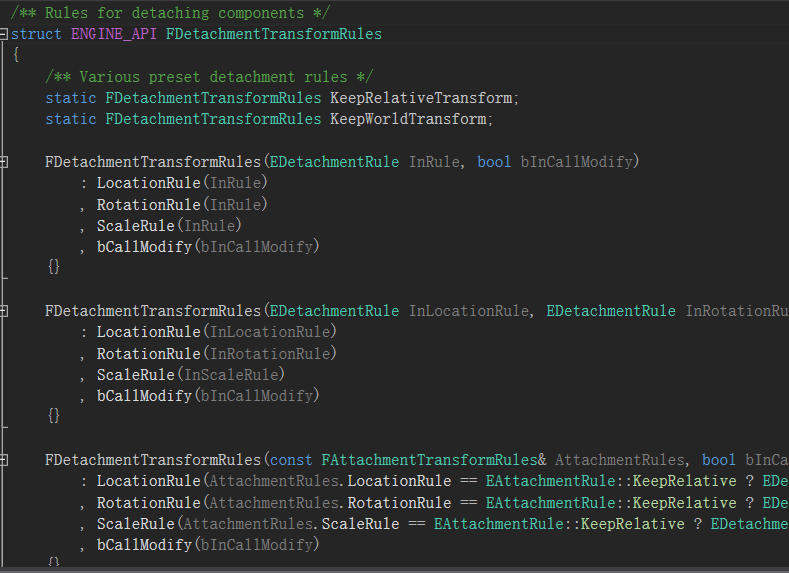
Destroyed()会在Actor被销毁时调用【UWorld::DestroyActor中会调用】

而ReceiveDestroyed()正如宏所表达的，可在蓝图中自定义实现的

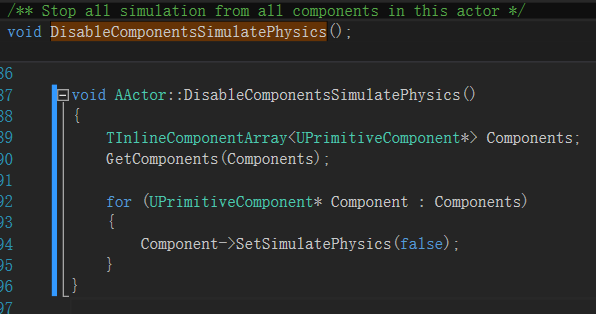


分离RootComponet从它当前attach的所有SceneComponent，实现是通过RootComponent调用USceneComponent::DetachFromComponent(…)方法完成的。

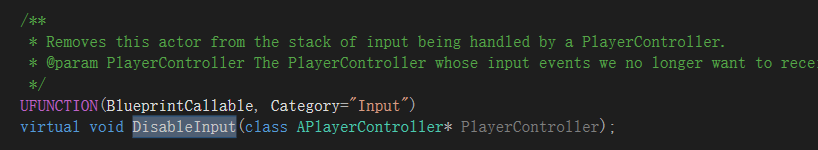
4.17版本之后推荐使用这个方法，而不是void DetachRootComponentFromParent(bool bMaintainWorldPosition = true);



FDetachmentTransformRules定义了详细的坐标变换规则用以应对分离后的RootComponent的坐标信息



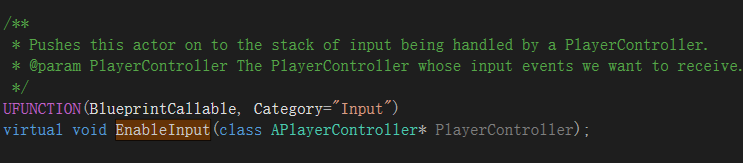
DisableComponentsSimulatePhysics()：停止所有Actor上的component的simulation(模拟)，顺便了解到AActor::GetComponents(…)的用法。

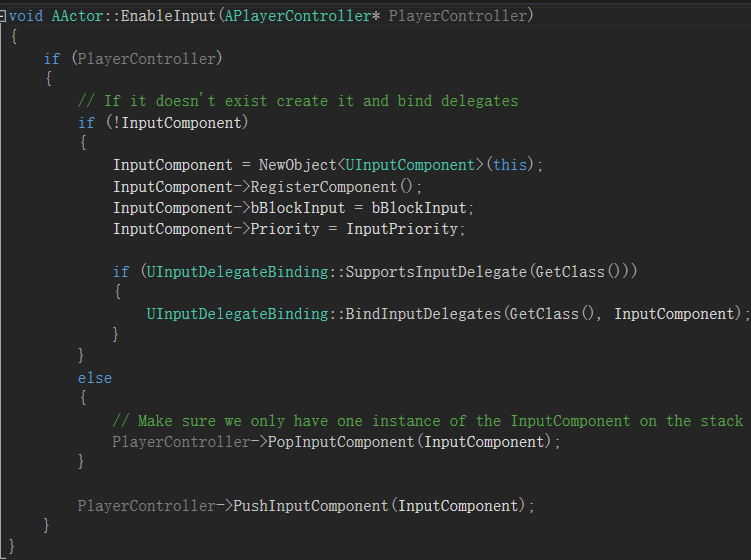


DisableInput():从PlayerController处理的输入堆栈中移除该actor

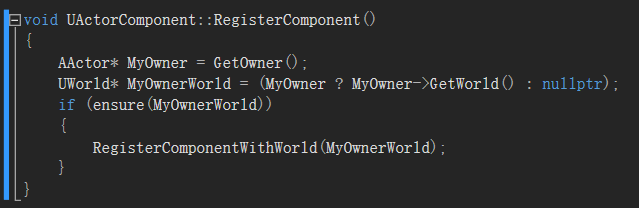


如果参数为null，则会从所有PlayerController中移除这个Actor

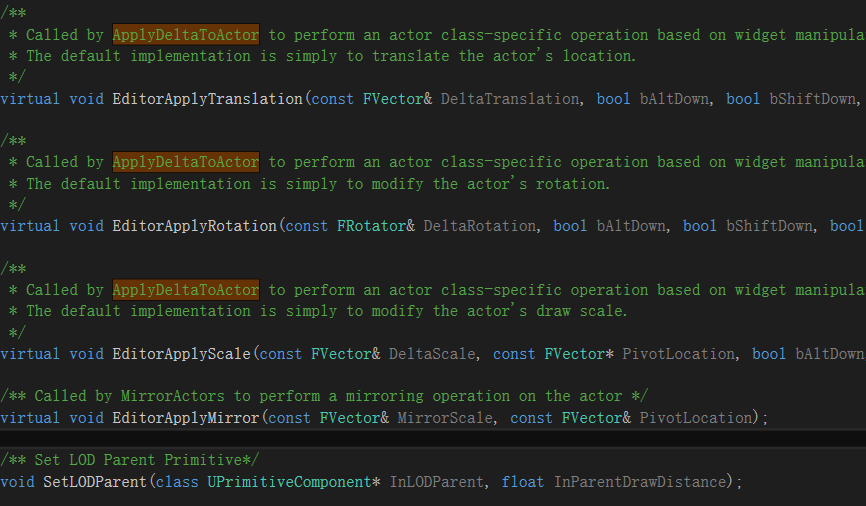




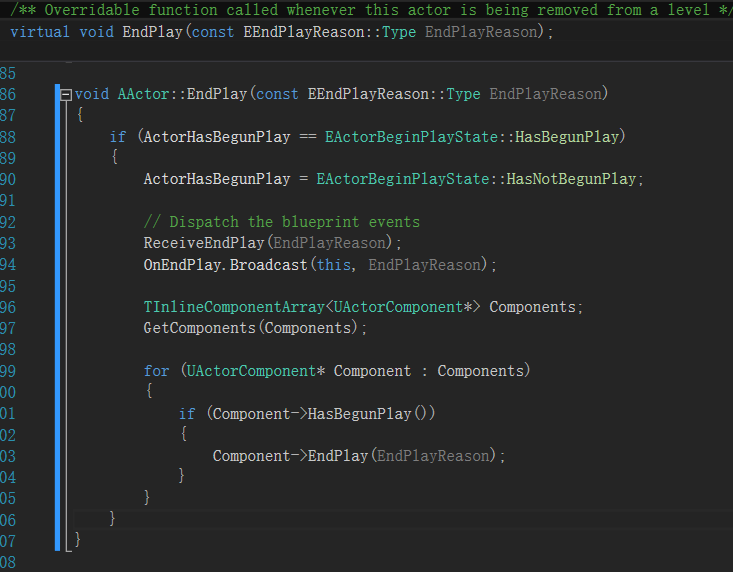
EnableInput()：一个Actor想要能够接收Input信息，则需要有InputComponent，一个组件申请之后要有RegisterComponent()



主要还是使World对其可见并管理

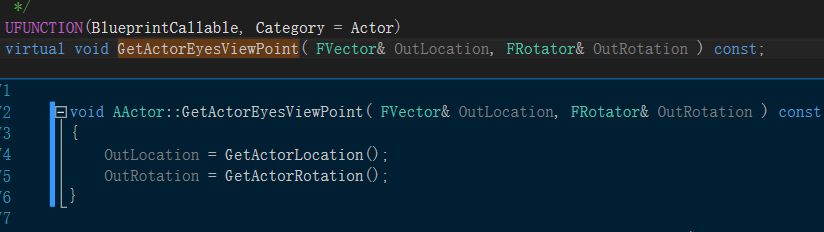


一些相关变换的函数。



当Actor从当前level中移除时被调用，同时也调用了Component的EndPlay函数

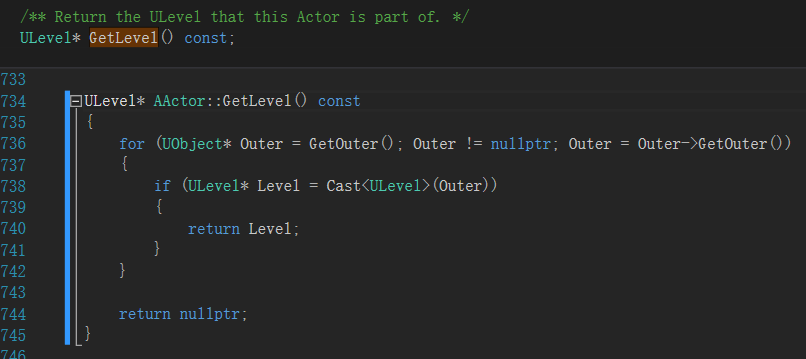
就像ReceiveDestroyed()一样，ReveiveEndPlay可在蓝图中自定义实现的



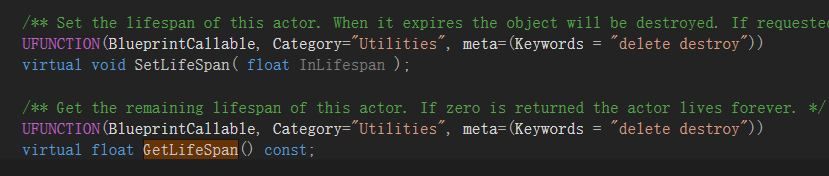
GetActorEyesViewPoint()：名称有迷惑性，其实只是返回RootComponent的Location和Rotation



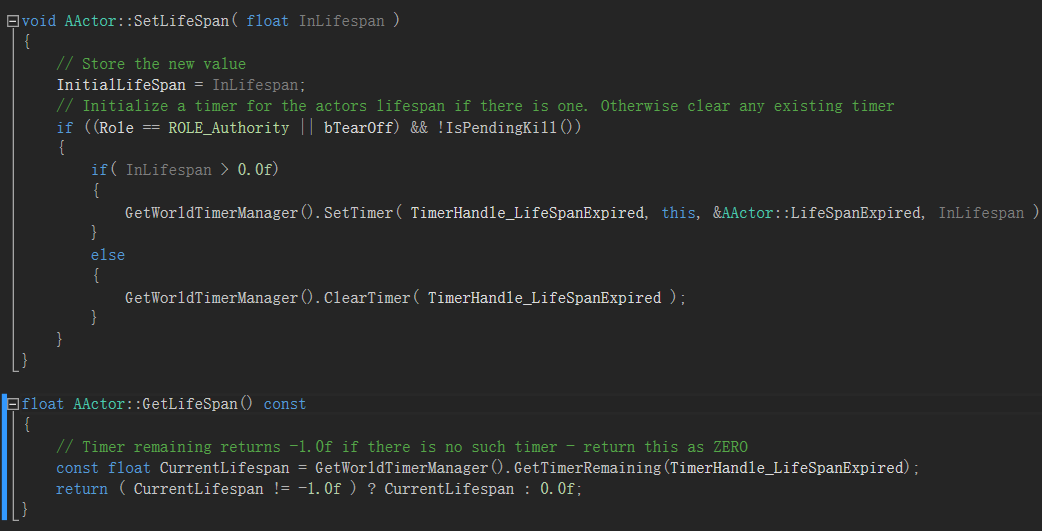
从RootComponent走上attach链，直到遇到不同的演员，然后将其返回。如果我们没有附加到不同actor中的组件，则返回nullptr



GetLevel()：返回Actor所属的ULevel，实现方法是通过不断GetOuter()，检查是否可cast ULevel。

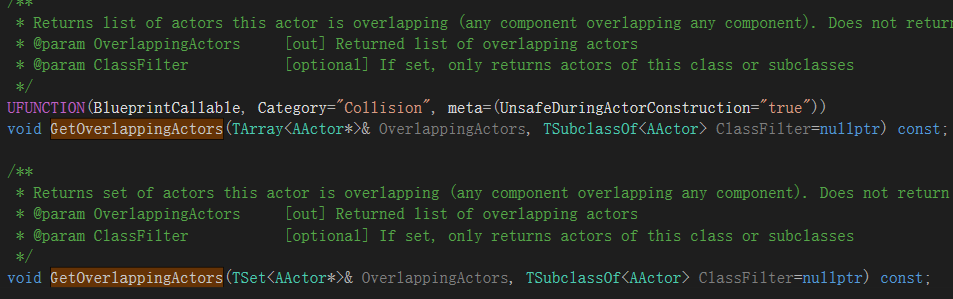


设置和获取Actor的生命时长

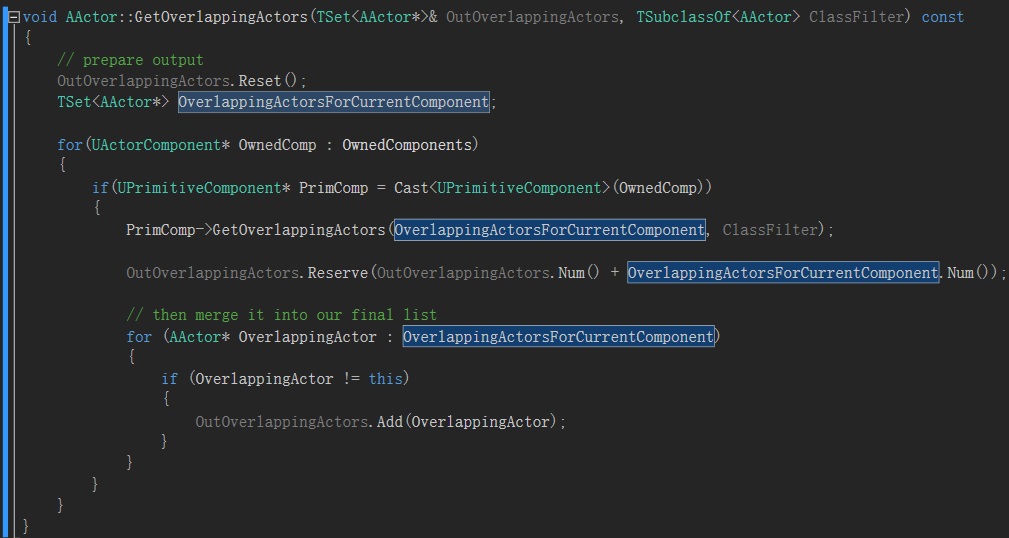




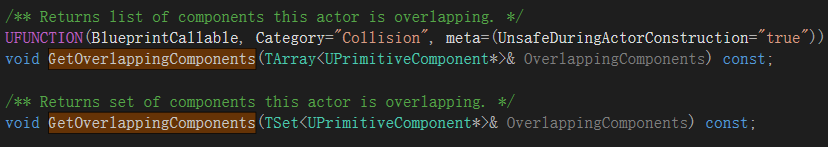
如果set<=0.0f则ClearTimer，不会死去，注意这里Timer在World的管理之下



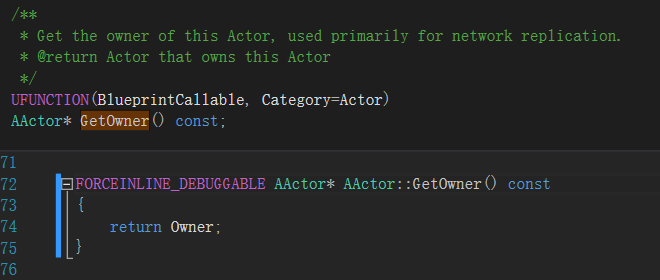
返回此Actor重叠的Actor的集合（任何组件与任何组件重叠）不会返回自身



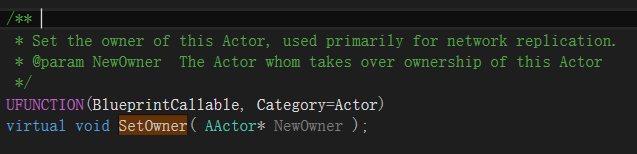
其中一个实现如上

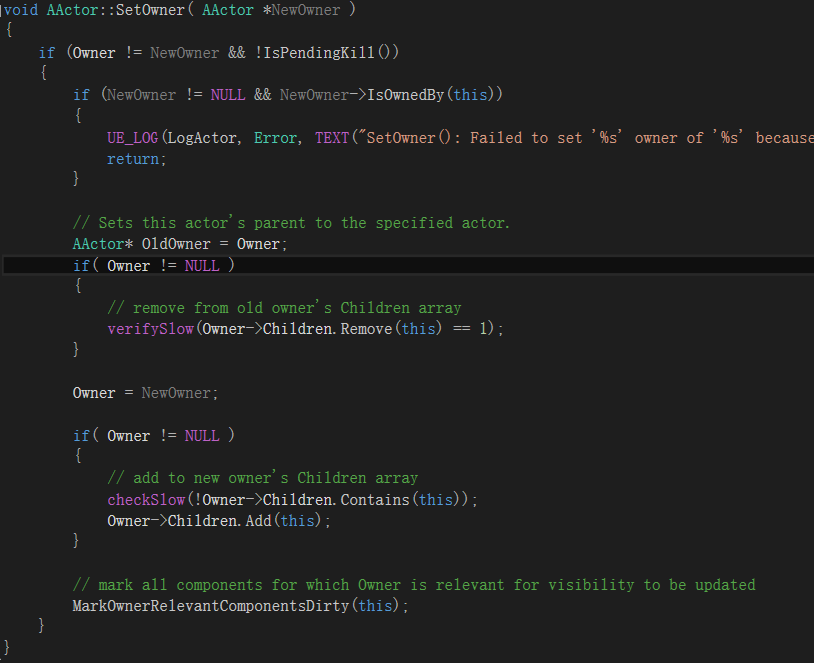


相似地，Component也有类似的操作，这里省略

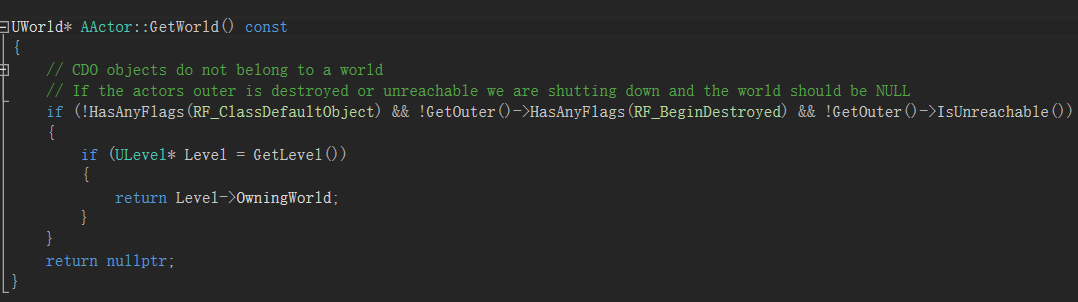


GetOwner()：在这之前的函数最常见的一个调用，主要用于网络复制

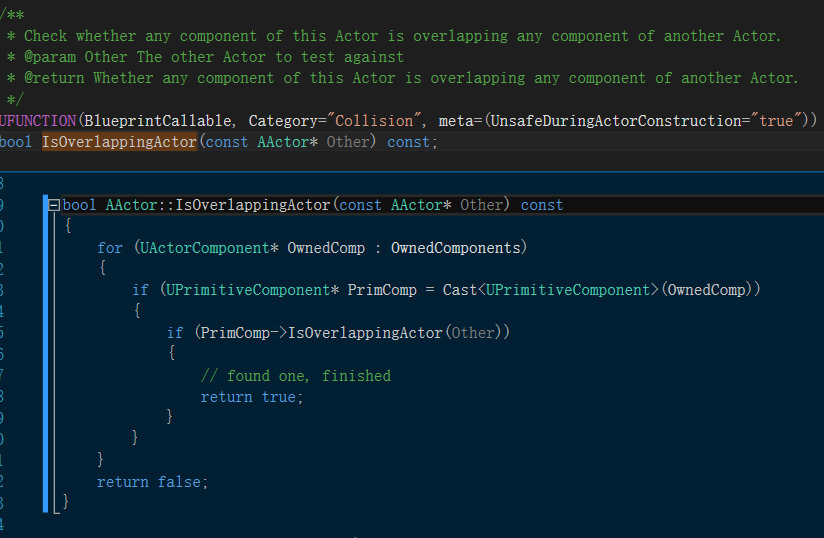




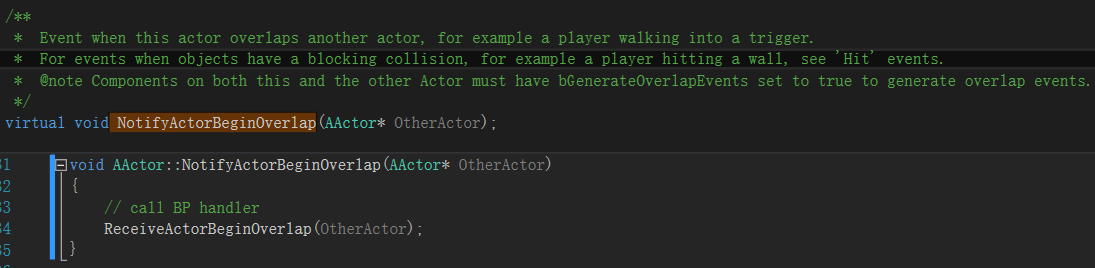
实现即是将旧的Owner移除this然后设置Owner并将其Children.Add(this)

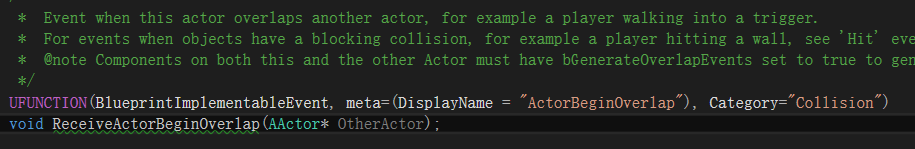


GetWorld()：AActor内没有World的索引，因此有点类似GetLevel()，先找到Level再通过Level索引World



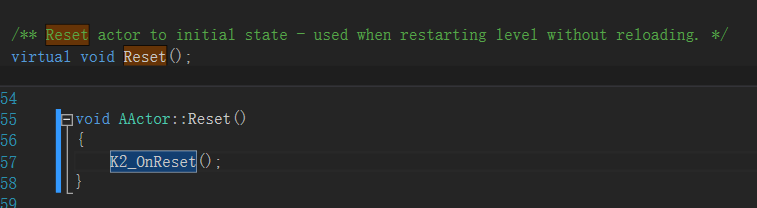
检查此Actor的任何Component是否与另一个Actor的任何Component重叠。

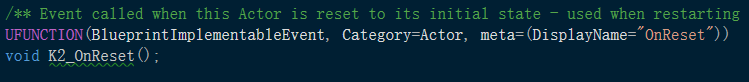




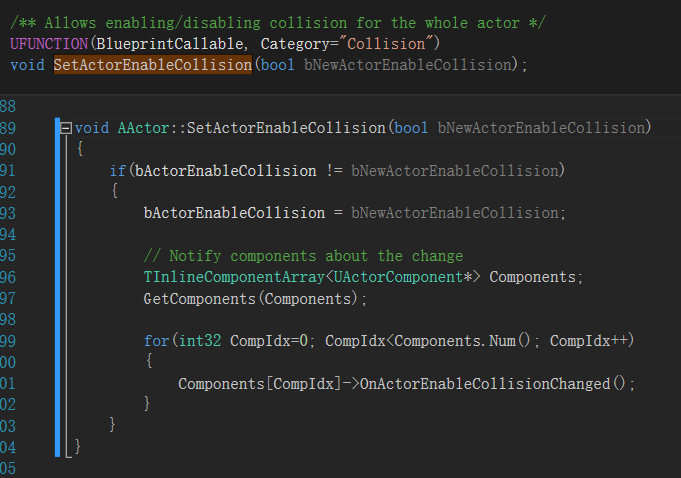
ReceiveActorBeginOverlap()：也可由蓝图实现的

类似的还有其他很多的Notify函数，也包括NotifyHit().



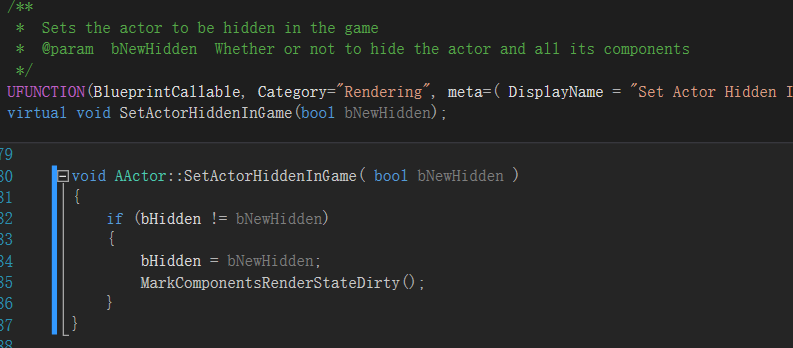


Reset()：将actor重置为初始状态 - 在不重新加载的情况下重新启动级别时使用。

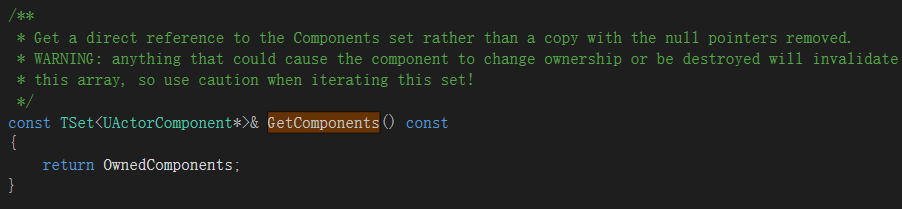


SetActorEnableCollision()：是否允许Actor的Collision

当有变更时，同时一起变更OwnedComponents的EnableCollision状态

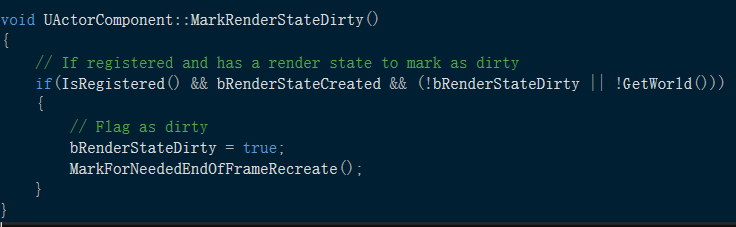






通过对OwnedComponents中所有已经注册的（正常下在刚创建后都已经register到World中了）Components递归处理（如果是一个ChildActorComponent）

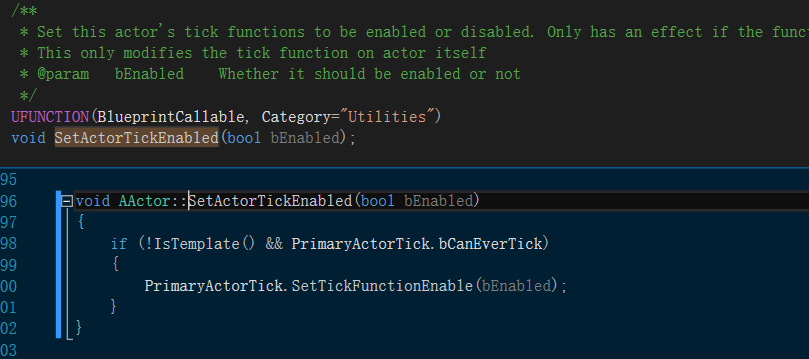




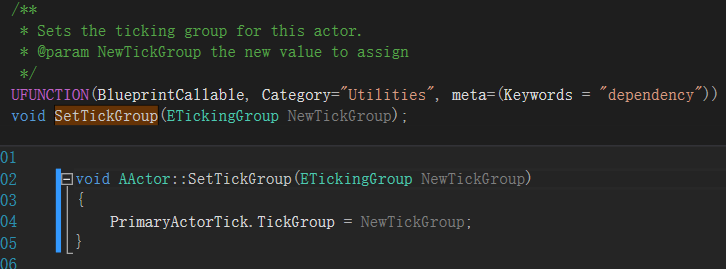
将渲染状态标记为脏-将被发送到每帧结尾的渲染线程



bRenderStateDirty为ture时则不会将其送到渲染器渲染，此时就完成了Hidden



SetActorTickEnabled()：开关Actor的tick函数，主要是通过PrimaryActorTick（**FActorTickFunction类型**）实现



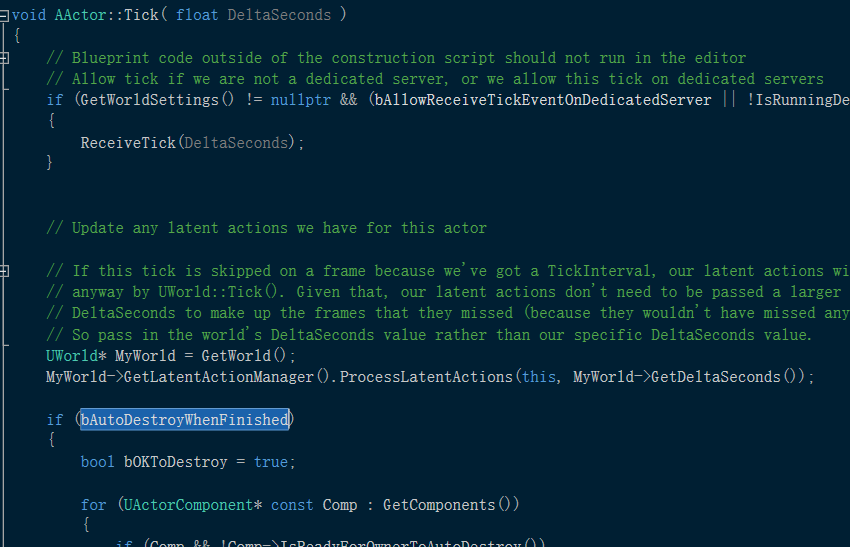
TickGroup是用来标志Tick发生的“时间”

ETickingGroup是一个枚举类

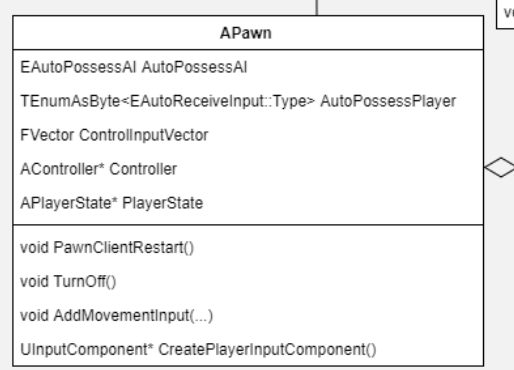
补充：Tick Dependency

AddTickPrerequisiteActor 和 AddTickPrerequisiteComponent 函数相关： set the actor or component on which the function is called to wait to tick until the specified other actor or component has finished ticking。

TickGroup和ETickingGroup是控制不同Component的Tick的先后发生的主要手段



Tick()：之前一直提到的Tick，这里因为有TickInterval存在，为了更新潜在的动作，中间调用了World的一些函数并以World的DeltaSeconds为准，同时也会每一帧最后检查bAutoDestroyWhenFinished并销毁

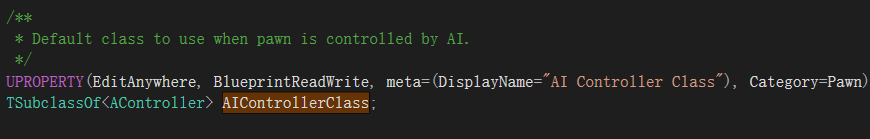


Engine/GameFramework/Pawn:

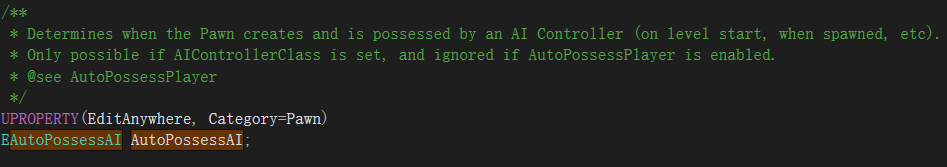
APawn

APawn继承于AActor，一切能显示的物体都是继承于AActor的，但一些特别的物体需要区别对待：就像静态场景物体与NPC的区别一样，总之：APawn是玩家或AI可以拥有的所有Actor的基类。它们是一个关卡中玩家和生物的物理表征。

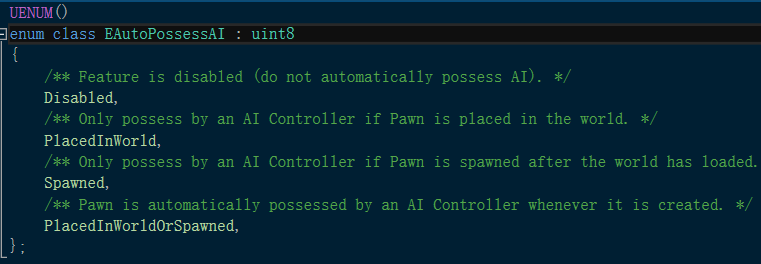
额外提供了3种功能 : 1.可以被Controller所控制 2.具有物理碰撞效果 3.相关移动功能接口

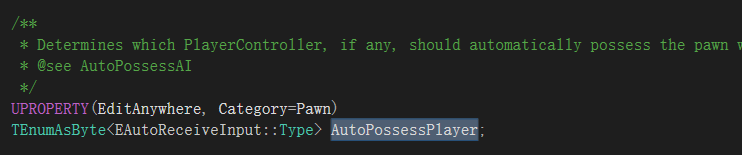


AI控制的Pawn使用的默认类



确定Pawn何时创建并由AI控制器拥有（在开始时，产生时等）。只有在设置了AIControllerClass的情况下才有可能，如果启用了AutoPossessPlayer，则会被忽略。

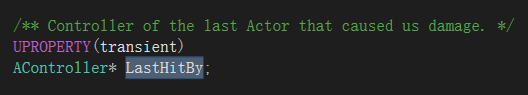




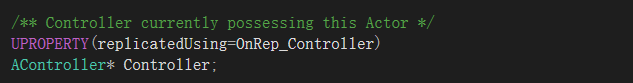
AutoPossessPlayer与上面的AI控制相对的Player控制

确定哪些PlayerController（如果有的话）应该在关卡开始时或产生Pawn时自动拥有Pawn

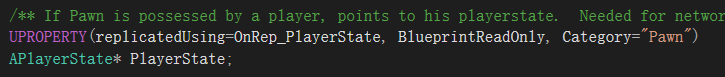
ControlInputVector：累积控制输入矢量，存储在世界空间中。这是悬而未决的输入，一旦消耗就清除（清零）

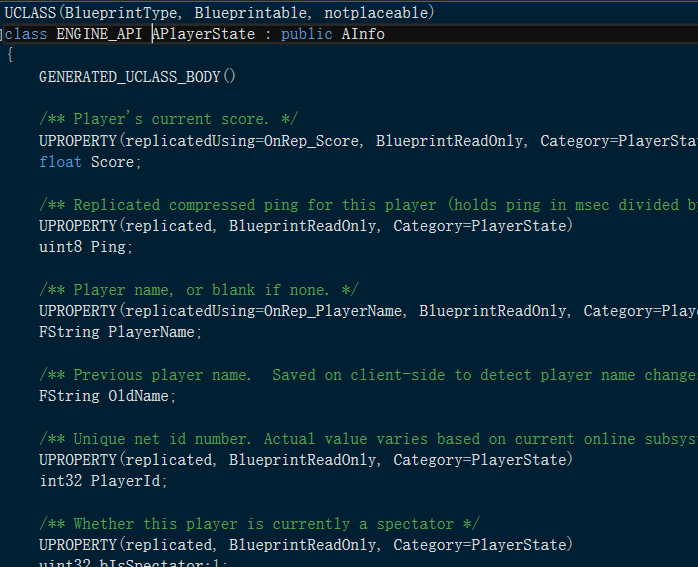


LastHitBy：导致我们受伤的最近一名Actor的Controller

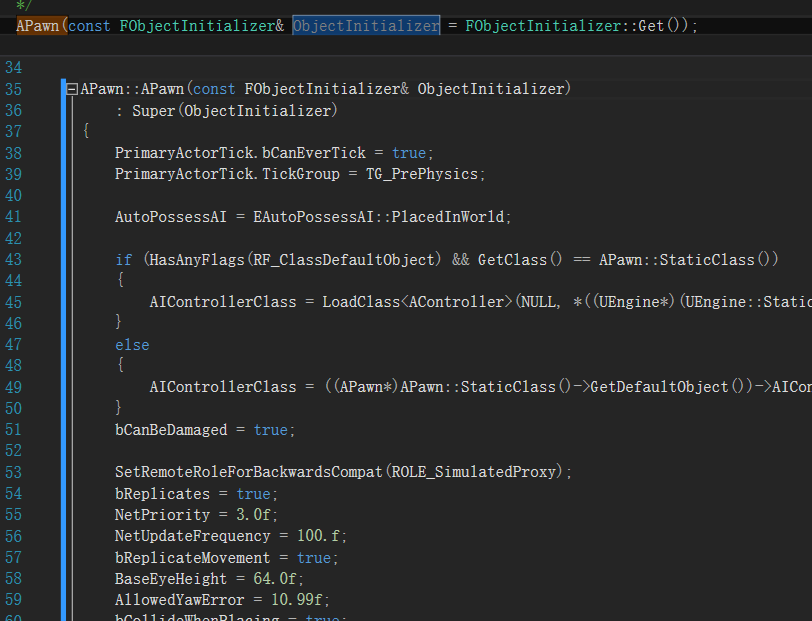


Controller：当前拥有这个Pawn的Controller

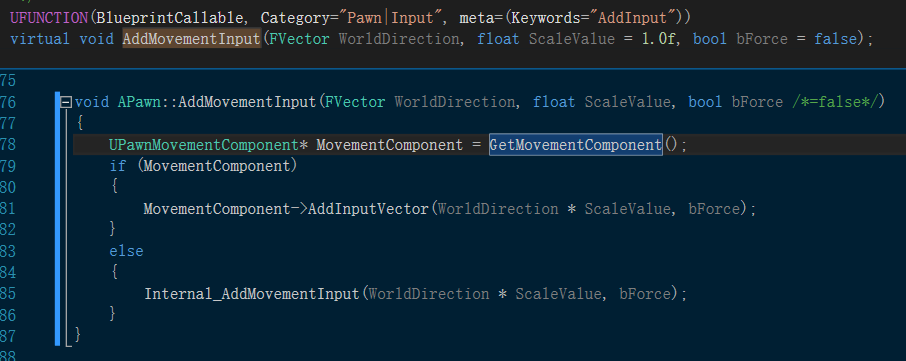




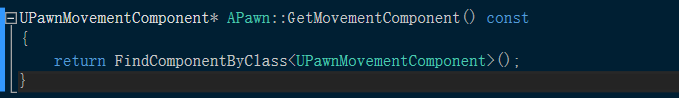
PlayerState：是一个APlayerState的指针，这个类主要存储玩家的信息，比如Name,Score,Id等等，如果 Pawn是被一个Player拥有的则指向他的PlayerState



APawn的唯一构造函数：和AActor（其父类）不同的是默认了Tick的存在，并先标记AutoPossessAI=PlacesdInWorld



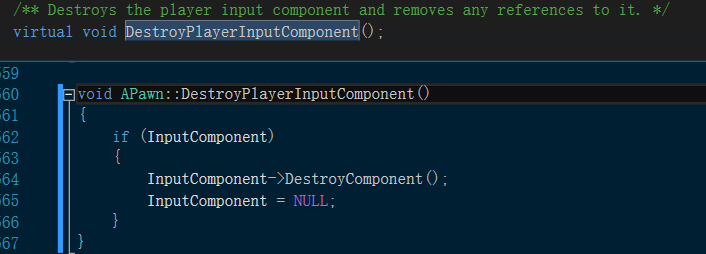
AddMovementInput()：沿着给定的世界方向矢量（通常是标准化的）添加运动输入，并按'ScaleValue'缩放。如果ScaleValue <0，则移动方向相反



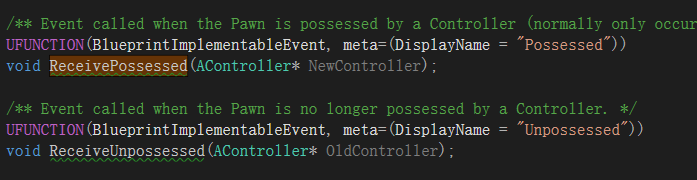
GetMovementComponent()：通过AActorL::FindComponentByClass返回



创建可用于自定义输入绑定的InputComponent，被PlayerController占据



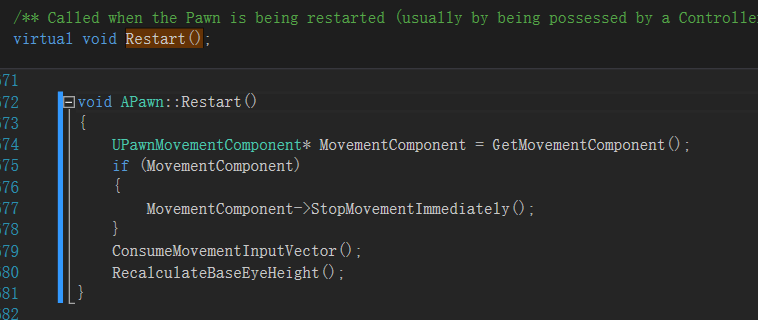
销毁InputComponent并删除对其的任何引用



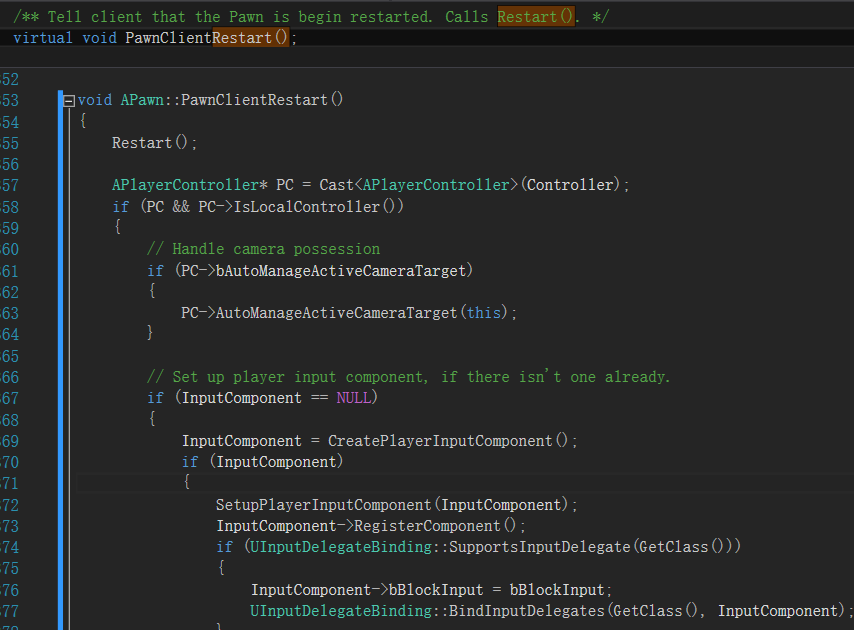
当Pawn拥有控制器时调用的事件（通常只发生在服务器/独立模式中）

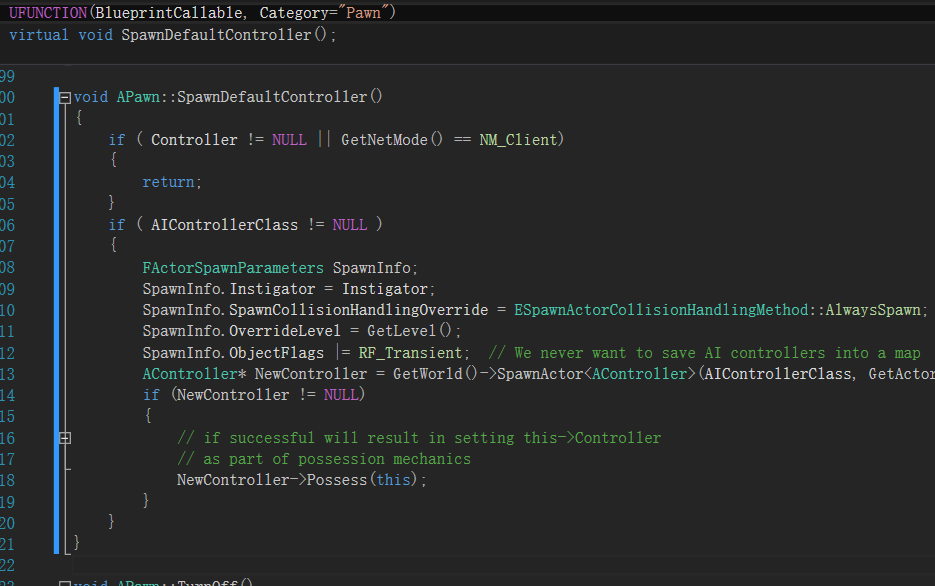
当控制器不再拥有Pawn时调用的事件

可由蓝图实现的

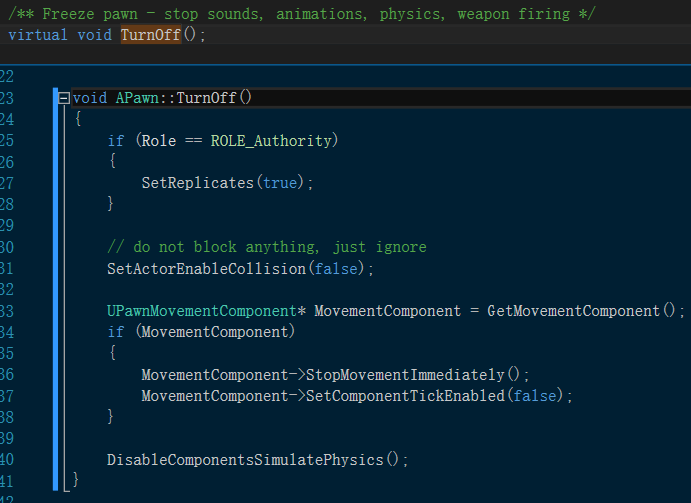


会被PawnClientRestart()调用

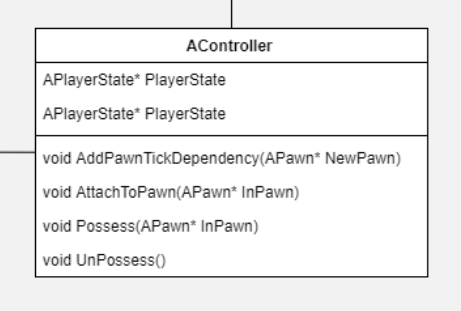




用于产生一个默认的Controller()(没有Contoller的情况下)，并使其拥有this：NewController->Possess(this);



TurnOff()：冻结Pawn的一切

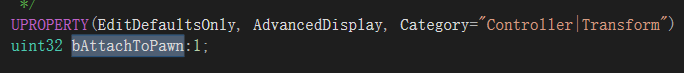


Engine/GameFramework/Controller

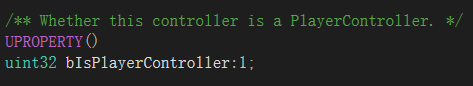
AController

控制者是非物理角色，可以拥有一个Pawn来控制其行为。玩家控制器被玩家用来控制棋子，而AIControllers为他们控制的棋子实施人工智能。控制器使用他们的Possess（） 方法来控制一个pawn ，并通过调用UnPossess（）来放弃对pawn的控制 。

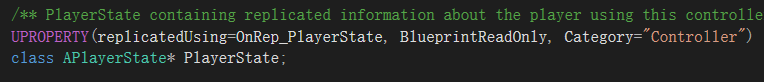
控制器接收许多他们正在控制的Pawn发生的事件的通知。这使控制器有机会实现响应此事件的行为，拦截事件并取代Pawn的默认行为。



bAttachToPawn：如果true，则控制器的位置将与拥有的Pawn的位置相匹配

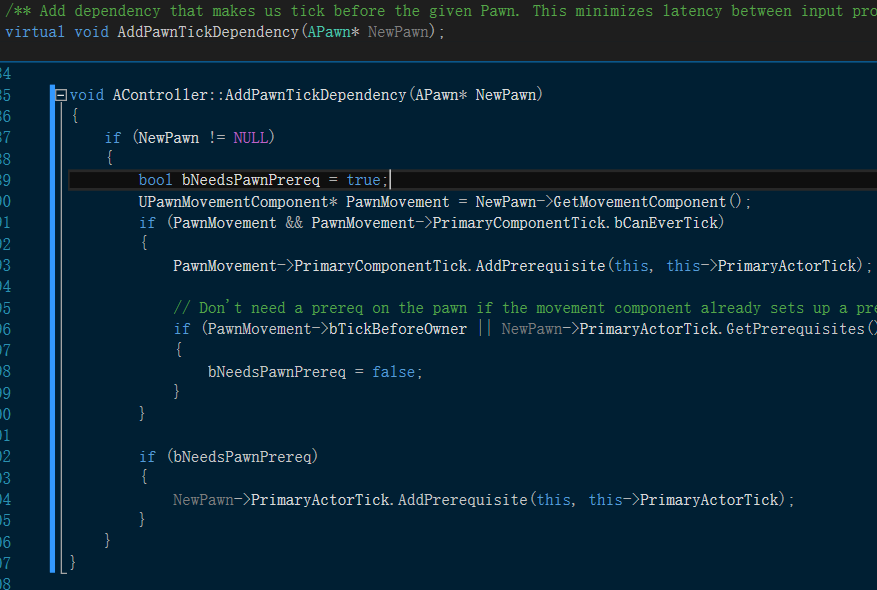


bIsPlayerController：该控制器是否为PlayerController

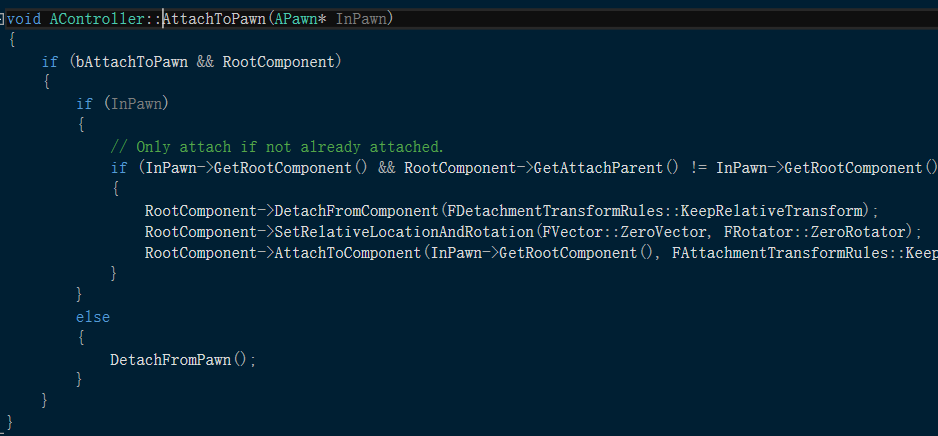


PlayerState：包含了使用了这个Controller的Player的PlayerState的拷贝信息（只在players有效）

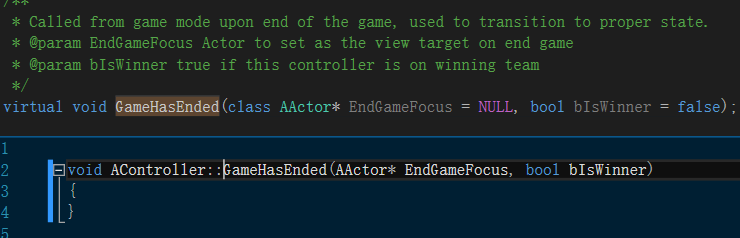
AController只有默认的构造函数



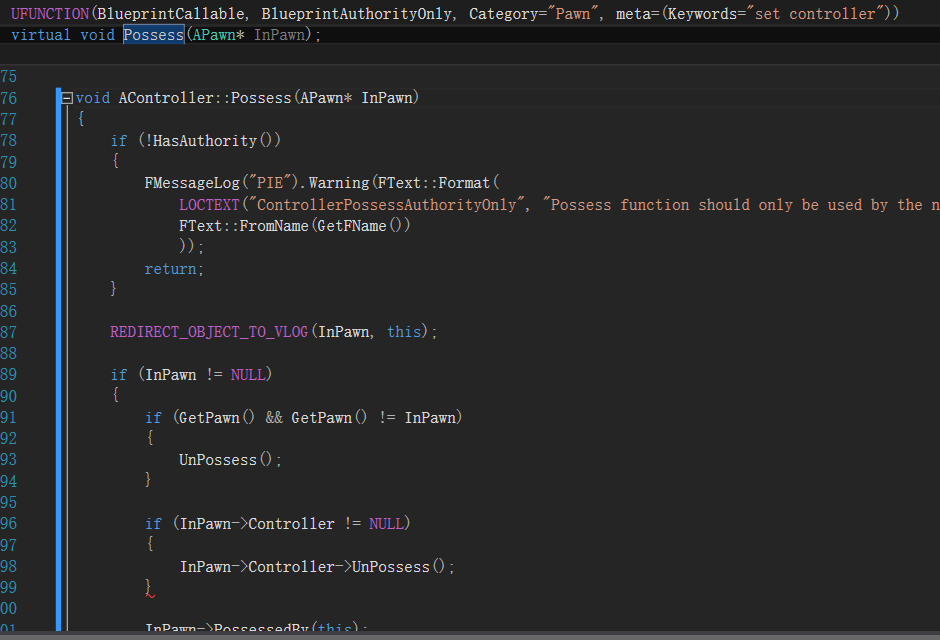
AddPawnTickDependency()：添加依赖关系，使我们在给定的Pawn之前Tick，这最大限度地减少了输入处理和Pawn运动之间的延迟。

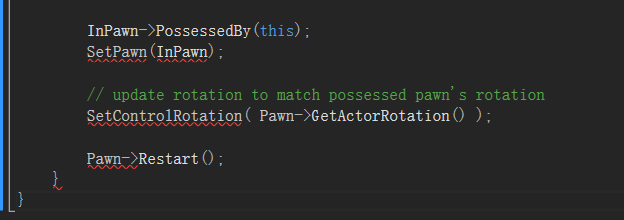


将控制器物理连接到指定的Pawn，以便Controller位置反映Pawn的位置，但rotation不变，若参数为nullptr则相当于DetachFromPawn()【将RootComponent从其父级分离，但前提是bAttachToPawn为true并将其连接到Pawn】



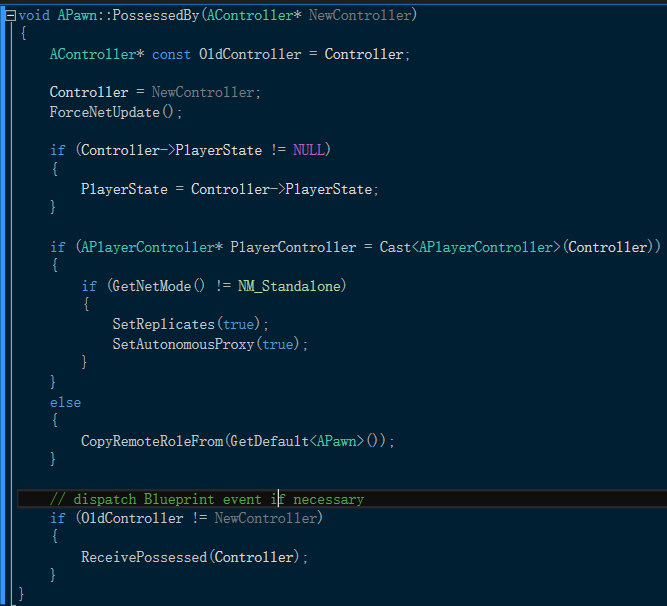
GameHasEnded()：在游戏结束时从游戏模式调用，用于转换到适当的状态



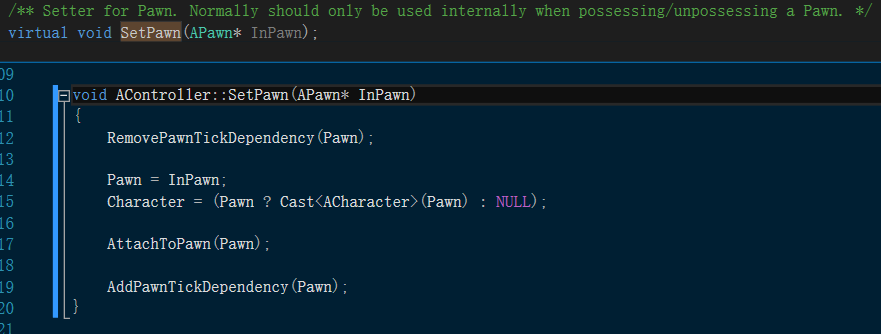


Possess()：使其“拥有”一个Pawn

实现：首先如果对象存在且不重复，首先执行UnPossess()，再检查Pawn对象中是否已有一个Controller，有则UnPossess()，之后调用Pawn的PossessedBy(…)函数

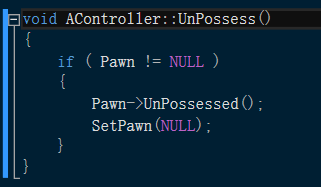


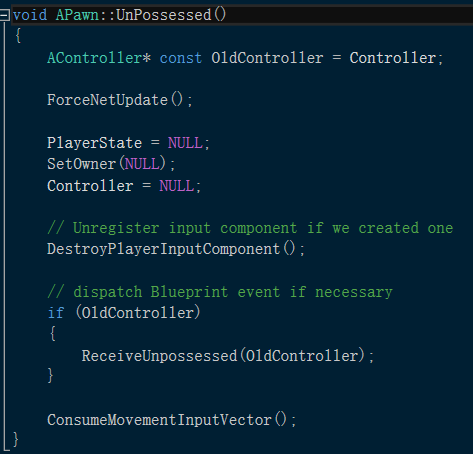
再调用SetPawn()



首先移除当前的Pawn的Tick依赖，之后AttachToPawn()，同时AddPawnTickDependency()来减少输入延迟

之后更新Controller的Rotation信息，最后调用Pawn的Restart()





UnPossess()相对简单一些，Pawn的处理比较彻底