# UE4中组件的概念与基本继承关系

以A开头的都是可以放置在场景中的，以U开头的组件只能依附于其他组件存在，不能单独放置在场景中。

## 组件

USceneComponent 场景组件（变换、位置等信息皆在该组件内，无显示信息， 一般为根组件）

UStaticMeshComponent 静态网格体组件

UParticleSystemComponent 粒子组件

UAudioComponent 声音组件

UBoxComponent 盒体组件

UCapsuleComponent 胶囊体组件

USphereComponent 球体组件

UTextRenderComponent 文本组件

UDecalComponent 贴花组件

UPrimitiveComponent 可模拟物体，且拥有变换的组件

UDecalComponent 用UDecalComponent显示印花图案，可以在非平面的地形上，按照地面的凹凸地势自动贴紧地面。

UDamageType 定义和描述一种特定形式的损害，并提供一种途径来定制对各种来源的损害的响应。

# 引擎内建类

## 模式相关

（1）编辑器模式：启动编辑器后的游戏状态。

（2）Runtime模式：游戏打包后的运行状态。

（3）PIE模式：（Play in editor）在编辑器中点击Play后运行状态。

## UEngine\* GEngine

代表引擎，数量为1； Editor或Runtime模式都是全局唯一，从进程启动开始创建，进程退出时销毁。

## UEditorEngine\* GEditor

代表编辑器，数量1；只在编辑器下存在且全局唯一，从编辑器启动开始创建，到编辑器退出时销毁。

## UGameInstance\* GameInstance

代表一场游戏，数量1。 从游戏的启动开始创建，游戏退出时销毁。这里的一场游戏指的是Runtime或PIE模式的运行的都算，一场游戏里可能会创建多个World切换。

## UWorld\* World

代表一个世界，数量可能>1。World和GameMode是关联的，可以包含多个Level，默认情况下OpenLevel常常会切换World。因此其生命周期，其实跟GameMode是一起的。编辑器模式下视口里的场景其实也是个World，因此EWorldType其实有多个类型：Game，Editor，PIE，EditorPreview，GamePreview等。

## ULocalPlayer\* LocalPlayer

代表本地玩家，数量可能>1。 UE支持本地分屏多玩家类型的游戏，但往往最常见的是就只有一个。LocalPlayer虽然往往跟PlayerController一起访问，但是其生命周期其实是跟UGameInstance一起的(默认一开始的时候就创建好一定数量的本地玩家)，或者更准确的说是跟LocalPlayer的具体数量挂钩（当然你也可以运行时动态调用AddLocalPlayer)。

# 代码声明相关

[UE4 c++编程，最近常用的头文件\_实习生小黄的博客-CSDN博客](https://blog.csdn.net/J_Wayne/article/details/105026889)

## [UCLASS（）](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Objects/)

UCLASS（）宏可使虚幻引擎识别新的类；使用 C++ 类向导创建的类声明自动通过 UCLASS() 宏进行处理。 UCLASS() 宏使得引擎意识到这个类的存在，并且还可以同键盘修饰符结合使用来在引擎中设置该类的行为。

UCLASS 包含定义类的一套属性和函数。这些是本地代码可用的普通 C++ 函数和变量，但被虚幻引擎特有的元数据所标记，它们在对象系统中的行为也因此受到控制。

## [GENERATED\_BODY（）](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Objects/)

GENERATED\_BODY宏不获取参数，但会对类进行设置，以支持引擎要求的基础结构。所有UCLASS均有此要求。

## [UPROPERTY（）](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/GameplayArchitecture/Properties/)

属性声明，通过UPROPERTY宏来定义属性元数据和变量说明符，说明函数、结构体等与引擎和编辑器各方面的相处方式；以UPROPERTY（）宏作为声明序列的变量可被引擎执行垃圾回收，也可在虚幻编辑器中显示和编辑。

UPROPERTY([specifier, specifier, ...], [meta(key=value, key=value, ...)])

Type VariableName;

（类声明包含很多变量和函数声明。变量与函数可以通过UPROPERTY和 [UFUNCTION](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/)宏分别进行处理，这些宏的功能和UCLASS() 宏类似。组件也可与UPROPERTY宏一起设置）

## PrimaryActorTick.bCanEverTick

在构造函数中出现，声明是否每帧都调用Tick()函数，若是则将其设置为True。

## SetupAttachment()

A->SetupAttachment(B)：将A附加到组件B上。

# [资源引用](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/GameplayArchitecture/Classes/)

理想状态下，类中的资源引用并不存在。硬编码资源引用很脆弱，优选方法是使用蓝图配置资源属性。然而，仍然完全支持硬编码引用。不需要在每次构造对象时搜索资源，因此这些搜索只执行一次。一个静态结构体可确保只执行一次资源搜索。

引用方式：

ConstructorHelpers::FObjectFinder<T> namestring(TEXT(“资源文件路径”)

若未找到资源，会报告失败。

# 位置、缩放相关API

## 获取Actor位置

GetActorLocation()：获取组件位置，返回类型为Fvector，分量为x、y、z。

GetActorRotation()：获取组件角度，返回类型为FRotator，分量为Pitch、yaw、roll，分别表示绕y、z、x轴旋转。

## 获取游戏运行时间

GetGameTimeSinceCreation ()：返回类型为float。

## 设置Actor位置

SetActorLocationAndRotation(Fvector, FRotator)

## 获取组件缩放比例

GetComponentScale()：

返回FVector，可通过加“.X”、“.Y”、“.Z”获取各个方向比例。

SetRelativeScale3D()

设置组件相对于其父组件的非均匀比例

## 设置缩放比例

SetWorldScale3D()：

输入FVector，设置三个方向的缩放比例

## 获取Actor向前的方向

（1）Rotator1.Vector()

根据物体的旋转信息得到其向前的方向向量（FVector），Rotator类型数据

（2）// 找出向前方向

**const** FRotator Rotation = Controller->GetControlRotation();

**const** FRotator YawRotation(0, Rotation.Yaw, 0);

// 获取向前矢量

**const** FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis:: X);

AddMovementInput(Direction, AxisValue); //位置修改，AxisValue为移动量

# 控制器相关

## APlayerController 控制器类

## 获取控制器

UGameplayStatics::GetPlayerController(const UObject\* WorldContextObject, int32 Playerindex)：

第一个参数：世界中物体的引用

第二个参数：玩家控制器的索引，0表示本地玩家

## 获取控制器对应相机

GetViewTarget()

## 设置摄像机

SetViewTarget()

SetViewTargetWithBlend ()：设置摄像机平滑过渡

## [控制器输入](https://docs.unrealengine.com/4.26/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/PlayerInput/)

（1）生成的Pawn类中带有“SetupPlayerInputComponent()”函数，在函数体中可通过“InputComponent->BindAction()”函数绑定操作映射，函数参数列表中可输入动作名称、函数指针等内容；同理，通过“InputComponent->BindAxis()”函数绑定轴映射。

（2）可通过“AutoPossessPlayer = EAutoReceiveInput::Player0”将pawn设置为由最小编号的玩家控制器控制（一般最小编号指本地玩家控制器）。

## [角色移动](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/2/)（character）

（1）获取角色X方向向量（返回类型为FVector）：

FRotationMatrix(Controller->GetControlRotation()).GetScaledAxis(EAxis::X)

（2）在某一方向移动：

AddMovementInput(方向向量,value)

第一个系数为移动方向，第二个为系数

（3）对角色添加视角转动

AddControllerYawInput函数，Pawn类的内置函数，视角转动；

UE4使用左手坐标系，roll、pitch、yaw分别表示绕X、Y、Z旋转。

（4）角色跳跃

Acharacter中内置了跳跃相关的布尔变量“bPressedJump”，按下跳跃键为True，松开为False；通过改变其值可实现跳跃。

## SetOnlyOwnerSee(布尔型数值)

A->SetOnlyOwnerSee(true)：设置A网格体仅对有此角色的PlayerController可见。

## 禁用玩家输入：Pawn->DisableInput()

## 获取玩家控制器

Pawn->GetController()；

## 切换控制角色

一个或许可行的思路：

GetAllActorsOfClass(this, 要查找的类型, 返回的数组)

用GetAllActorsOfClass()获取所有类的实例（可知参数列表中指定类型），可获取玩家控制器，将其控制的角色切换

# [Actor](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/)

[Actors | 虚幻引擎文档 (unrealengine.com)](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/)

## [生成Actor](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/Spawning/)

[SpawnActor方法 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/139547340)

SpawnActor()：

UWorld的成员之一，成功生成Actor后，引擎调用其BeginPlay()

**函数参数：**

AActor\* UWorld::SpawnActor

(

UClass\* Class,

FName InName,

FVector **const**\* Location,

FRotator **const**\* Rotation,

AActor\* Template,

**bool** bNoCollisionFail,

**bool** bRemoteOwned,

AActor\* Owner,

APawn\* Instigator,

**bool** bNoFail,

ULevel\* OverrideLevel,

**bool** bDeferConstruction

)

## 销毁Actor

（1）通过Destroy()销毁Actor，在该过程中将调用EndPlay()，能在Actor进入回收站之前执行自定义逻辑。

（2）另一个控制Actor生命周期时长的方法是使用 Lifespan 成员。可以在对象的构造函数中设置Actor的时间跨度，也可以在运行时使用其他代码进行设置。当这段时间到期后，会自动对该Actor调用 Destroy。

**设置生命周期：InitialLifeSpan=3.0f;**

## [获取Actor视角](https://docs.unrealengine.com/5.0/en-US/API/Runtime/Engine/GameFramework/AActor/GetActorEyesViewPoint/https:/docs.unrealengine.com/5.0/en-US/API/Runtime/Engine/GameFramework/AActor/GetActorEyesViewPoint/)

AActorLLGetActorEyesViewPoint(Location, Rotation)

## 查找Actor

获取所有的Actor：GetAllActorsofClass（）

# 坐标相关

## 坐标变换

将坐标从物体空间坐标系转换至世界空间坐标系

FTransform(Rotation1).TransformVector(FVector1)

其中，Rotation1为物体的旋转变换，FVector1为物体空间坐标系下的坐标，返回类型为FVector。

# [设置材质](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/3/)

## 动态材质实例：UMaterialInstanceDynamic（是一种类型）

此材质是动态的，可发生改变

## 材质类型：UMaterial

## 设置材质

//获取材质资源

**static** ConstructorHelpers::FObjectFinder<UMaterial> Material(TEXT(""));

**if** (Material.Succeeded())

{

//为声明的动态材质实例赋值

MaterialInstance = UMaterialInstanceDynamic::Create(Material.Object, MeshComp onent);

}

MeshComponent->**SetMaterial**(0, MaterialInstance);

# 函数相关

## 函数定义

例：

[UFUNCTION( )](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/GameplayArchitecture/Functions/)

void fun( )

UFUNCTION与变量声明时的UPROPERTY功能类似，是对函数属性的定义。如通过在UFUNCTION（）中添加“BlueprintCallable”属性，可在蓝图中对函数进行调用。

# [序列化](https://blog.csdn.net/mohuak/article/details/83027211)

## 概念

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](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)、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。

# 反射机制

## 概念

UE4使用其自己的反射实现来支持动态功能，如垃圾回收、序列化、网络复制和蓝图/C++通信。这些功能是可选的，意味着您必须将正确的标记添加到类型，否则虚幻将忽略它们，而不会为它们生成反射数据。下面是对基本标记的简要概述：

UCLASS() - 用于告诉虚幻为结构体生成反射数据。类必须派生自UObject。

USTRUCT() - 用于告诉虚幻为结构体生成反射数据。

`GENERATED\_BODY()\*\* - UE4将这个标记替换为将为该类型生成的所有必要的样板代码。

UPROPERTY() - 支持将UCLASS的成员变量或USTRUCT用作UPROPERTY。UPROPERTY有很多用法。它可以允许复制变量、序列化变量和从蓝图访问变量。它们可以供垃圾回收程序使用，用来跟踪对 UObject 的引用次数。

UFUNCTION() - 支持将UCLASS的类方法或USTRUCT用作UFUNCTION。UFUNCTION可以允许从蓝图调用类方法，用作RPC等多种用途。

# 类命名前缀

派生自 Actor 的类带有 A 前缀，如 AController。

派生自 Object 的类带有 U 前缀，如 UComponent。

Enums 的前缀是 E，如 EFortificationType。

Interface 的前缀通常是 I，如 IAbilitySystemInterface。

Template 的前缀是 T，如 TArray。

派生自 SWidget 的类（Slate UI）带有前缀 S，如 SButton。

其他类的前缀为字母F，如 FVector。

# [字符串](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/StringHandling/)

## FString

FString 是一个可变字符串，类似于 std::string。FString 拥有很多方法，方便对字符串进行处理。要创建新的 FString，请使用 TEXT 宏。

## FText

FText 类似于FString，但旨在用于本地化文本。要创建新的 FText，请使用 NSLOCTEXT 宏。该宏将使用默认语言的名称空间、键和值。

## FName

FName 存储通常反复出现的字符串作为辨识符，以在比较时节省内存和CPU时间。如果有多个对象引用一个字符串，FName 使用较小的存储空间索引来映射到给定字符串，而不是在引用它的每个对象中多次存储完整字符串。这样会将字符串内容存储一次，节省在多个对象中使用该字符串时占用的内存。FName 比较更快是因为UE4能够检查其索引值来确认其是否匹配，而无须检查每一个字符是否相同。

# [容器](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/)

## [TArray](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/TArrays/)

[常用API](https://docs.unrealengine.com/5.0/en-US/API/Runtime/Core/Containers/TArray/)

其功能与 **std::vector** 十分相似，但会提供更多功能。

以下是一些常见操作：

TArray<AActor\*> ActorArray = GetActorArrayFromSomewhere();

// 告知当前ActorArray中存储了多少个元素（AActor）。

int32 ArraySize = ActorArray.Num();

// TArray基于0（第一个元素将位于索引0处）

int32 Index = 0;

// 尝试检索给定索引处的元素

AActor\* FirstActor = ActorArray[Index];

// 在数组末尾添加新元素

AActor\* NewActor = GetNewActor();

ActorArray.Add(NewActor);

// 在数组末尾添加元素，但前提必须是该元素尚不存在于数组中

ActorArray.AddUnique(NewActor); // 不会改变数组，因为已经添加了NewActor。

// 从数组中移除"NewActor"的所有实例

ActorArray.Remove(NewActor);

// 移除指定索引处的元素

// 索引之上的元素将下移一位来填充空白空间

ActorArray.RemoveAt(Index);

// 更高效版本的"RemoveAt"，但不能保持元素的顺序

ActorArray.RemoveAtSwap(Index);

// 移除数组中的所有元素

ActorArray.Empty();

## [TMap](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/TMap/)

[常用API](https://docs.unrealengine.com/5.0/en-US/API/Runtime/Core/Containers/TMapBase/)

TMap 是键-值对的集合，类似于 std::map。TMap 具有一些根据元素键查找、添加和移除元素的快速方法。您可以使用任意类型来表示键，因为它定义有 GetTypeHash 函数。

## TSet

[常用API](https://docs.unrealengine.com/en-US/API/Runtime/Core/Containers/TSet)

TSet存储唯一值集合,类似于 std::set。虽然通过 TArray 可通过其AddUnique 和 Contains 方法支持类似集的行为，TSet 可以更快的实现这些运算且不会自动添加非独有元素。

## 容器迭代器

通过使用迭代器，可以循环遍历容器的所有元素。

以TSet为例，其初始化方式为：

TSet<AEnemy\*> EnemySet;

Auto EnemyIterator =EnemySet.CreateIterator();

//获取当前元素的索引

int32 Index = EnemyIterator.GetIndex();

//将迭代器复位到第一个元素

EnemyIterator.Reset();

# [UE\_LOG](https://zhuanlan.zhihu.com/p/463724067)

日志Logging）意味着在运行期间的某个时间保持事件、函数调用、变量值等的有序记录。这通常以文本的形式保存在日志文件中。日志对于Unreal应用开发的调试排错非常有帮助。

## UE\_LOG语法

UE\_LOG(LogTemp, Warning, TEXT(“Hello”));

UE\_LOG是将日志消息输出到日志文件中的[宏。](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/Macro_%28computer_science%29" \t "_blank)它采用的第一个输入参数是日志记录类别的名称。引擎中已经内置了许多这些类别，在 CoreGlobals.h 中定义。

第二个参数为Log的类型，包括ERROR、Warning、Log三类。

第三个参数为要显示的文字。

## 消息格式化示例

含FString参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("The Actor's name is %s"), \*YourActor->GetName());

含布尔参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("The boolean value is %s"), ( bYourBool ? TEXT("true") : TEXT("false") ));

含整型参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("The integer value is: %d"), YourInteger);

含浮点型参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("The float value is: %f"), YourFloat);

含FVector参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("The vector value is: %s"), \*YourVector.ToString());

含多个参数的消息：

UE\_LOG(LogTemp, Warning, TEXT("Current values are: vector %s, float %f, and integer %d "), \*YourVector.ToString(), YourFloat, YourInteger);

## 将消息打印到屏幕上

GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::White, TEXT("This message will appear on the screen!"));

# [委托](https://docs.unrealengine.com/5.0/zh-CN/delegates-and-lamba-functions-in-unreal-engine/)

## 官方解释

委托是一种泛型但类型安全的方式，可在C++对象上调用成员函数。可使用委托动态绑定到任意对象的成员函数，之后在该对象上调用函数，即使调用程序不知对象类型也可进行操作。复制委托对象很安全。你也可以利用值传递委托，但这样操作需要在堆上分配内存，因此通常并不推荐。请尽量通过引用传递委托。虚幻引擎共支持三种类型的委托：

单点委托

组播委托

动态(UObject, serializable)

## [网上解释](https://www.cnblogs.com/kekec/p/10678905.html#:~:text=UE4%E4%B8%AD%E7%9A%84%20delegate%20%EF%BC%88,%E5%A7%94%E6%89%98%20%EF%BC%89%E5%B8%B8%E7%94%A8%E4%BA%8E%E8%A7%A3%E8%80%A6%E4%B8%8D%E5%90%8C%E5%AF%B9%E8%B1%A1%E4%B9%8B%E9%97%B4%E7%9A%84%E5%85%B3%E8%81%94%EF%BC%9A%E5%A7%94%E6%89%98%E7%9A%84%E8%A7%A6%E5%8F%91%E8%80%85%E4%B8%8D%E4%B8%8E%E7%9B%91%E5%90%AC%E8%80%85%E6%9C%89%E7%9B%B4%E6%8E%A5%E5%85%B3%E8%81%94%EF%BC%8C%E4%B8%A4%E8%80%85%E9%80%9A%E8%BF%87%E5%A7%94%E6%89%98%E5%AF%B9%E8%B1%A1%E9%97%B4%E6%8E%A5%E5%9C%B0%E5%BB%BA%E7%AB%8B%E8%81%94%E7%B3%BB%20%E7%9B%91%E5%90%AC%E8%80%85%E9%80%9A%E8%BF%87%E5%B0%86%E5%93%8D%E5%BA%94%E5%87%BD%E6%95%B0%E7%BB%91%E5%AE%9A%E5%88%B0%E5%A7%94%E6%89%98%E4%B8%8A%EF%BC%8C%E4%BD%BF%E5%BE%97%E5%A7%94%E6%89%98%E8%A7%A6%E5%8F%91%E6%97%B6%E7%AB%8B%E5%8D%B3%E6%94%B6%E5%88%B0%E9%80%9A%E7%9F%A5%EF%BC%8C%E5%B9%B6%E8%BF%9B%E8%A1%8C%E7%9B%B8%E5%85%B3%E9%80%BB%E8%BE%91%E5%A4%84%E7%90%86)

UE4中的delegate（委托）常用于解耦不同对象之间的关联：委托的触发者不与监听者有直接关联，两者通过委托对象间接地建立联系；监听者通过将响应函数绑定到委托上，使得委托触发时立即收到通知，并进行相关逻辑处理；委托，又称代理，本质是一个特殊类的对象，它内部可以储存（一个或多个）函数指针、调用参数和返回值

委托的作用如同函数指针，但它更安全（支持编译期类型检查），而且更易于使用。

个人理解：

某些类中的函数要在一定条件下才触发，因此需对该触发条件进行监听，条件满足时执行函数。委托便是将“条件监听”与“函数执行”交给另一个委托类来做，将要调用的函数指针交给委托类；委托类发现触发条件满足时，直接执行相应函数。

## 示例

在球体组件发生碰撞时，调用相应函数：

Sphere-> OnComponentBeginOverlap.AddDynamic(**this**,&ALightSwitchCodeOnly::OnOverlapBegin); // 当此组件与某对象重叠时，设置通知

OnComponentBeginOverlap发生时调用OnOverlapBegin。

# [UGameplayStatics](https://docs.unrealengine.com/5.0/en-US/API/Runtime/Engine/Kismet/UGameplayStatics/)

静态类，提供了一系列可玩性相关的实用函数，可在蓝图与C++中调用。

（可以理解成一个函数库，头文件#include "Kismet/GameplayStatics.h"）

[分类整理文档](https://zhuanlan.zhihu.com/p/71254108)

# OnHit、AddDynamic

碰撞、动态绑定参考[虚幻四C++入坑指南合集版\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV14K411J7v2?p=15&spm_id_from=pageDriver)

# [TSubclassOf()](https://docs.unrealengine.com/5.0/zh-CN/typed-object-pointer-properties-in-unreal-engine/)

**TSubclassOf** 是提供 UClass 类型安全性的模板类。例如在创建一个投射物类，允许设计者指定伤害类型。可只创建一个 UClass 类型的 UPROPERTY，让设计者指定派生自 UDamageType 的类；或者可使用 TSubclassOf 模板强制要求此选择。

如：

（1）UPROPERTY(EditDefaultsOnly, Category=Damage)

UClass\* DamageType;

（2）UPROPERTY(EditDefaultsOnly, Category=Damage)

TSubclassOf<UDamageType> DamageType;

在选择伤害类型时，方式（1）可选择任意的Class，方式（2）只能选择派生自UDamageType的类。

# [变量、定时器和事件](https://docs.unrealengine.com/4.26/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/VariablesTimersEvents/)

## 定时器句柄

FTimerHandle CountdownTimerHandle;

每次指定 **定时器** 运行函数时，都会得到 **定时器句柄**。需充分利用此句柄，以便在倒数结束时关闭定时器。（句柄可理解为一个标识）

## 停止运行定时器

GetWorldTimerManager().**ClearTimer**(CountdownTimerHandle);

## 设置定时器

GetWorldTimerManager().**SetTimer**(CountdownTimerHandle, **this**, &ACountdo wn::AdvanceTimer, 1.0f, **true**);

参数：CountdownTimerHandle为定时器句柄；ACountdo wn::AdvanceTimer为定时器要调用的函数；1.0f表示每一秒执行一次相应函数；“true”表示循环执行，若为false表示只执行一次。（放在BeginPlay()函数中）

# GameMode

## GetWorld()->GetAuthGameMode()

//获取GameMode，对于多人游戏，仅可在服务器调用

# [组件](https://docs.unrealengine.com/4.26/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/Components/)

**组件** 是一种特殊类型的 **对象**，**Actor** 可以将组件作为子对象附加到自身。组件适用于共享相同的行为，例如显示视觉表现、播放声音。它们还可以表示项目特有的概念，例如载具解译输入和改变其速度与方向的方式。举例而言，某个项目拥有用户可控制车辆、飞机和船只。可以通过更改载具Actor所使用的组件来实现载具控制和移动的差异。

## Actor组件

UActorComponent 是所有组件的基类。由于组件是渲染网格体和图像、实现碰撞和播放音频的唯一方法，因此玩家游戏期间在场景中看到或进行交互的一切其实都是某一类组件的成果。

**Actor组件**（类 UActorComponent）最适用于抽象行为，例如移动、物品栏或属性管理，以及其他非物理概念。Actor组件没有变换，即它们在场景中不存在任何物理位置或旋转。

默认情况下，Actor组件不更新。为了让Actor组件逐帧更新，必须在构造函数中将 **PrimaryComponentTick. bCanEverTick** 设置为 true 来启用tick。之后，在构造函数中或其他位置处，必须调用 **PrimaryComponentTick. SetTickFunctionEnable(true)** 以开启更新。之后可调用 **PrimaryComponentTick. SetTickFunctionEnable(false)** 停用tick。如果知道组件永远不需要更新，或者打算手动调用自己的更新函数（也许从拥有的Actor类），将 **PrimaryComponent Tick.bCanEverTick** 保留为默认值 false 即可，这样可以稍微改善性能。

## 场景组件

场景组件是指存在于场景中特定物理位置处的Actor组件。该位置由 **变换**（类 [FTransform](https://api.unrealengine.com/INT/API/Runtime/Core/Math/FTransform/index.htm)）定义，其中包含组件的位置、旋转和缩放。场景组件能够通过将彼此连接起来形成树，Actor可以将单个场景组件指定为"根"，意味着这个Actor的场景位置、旋转和缩放都根据此组件来绘制。

场景组件（类 USceneComponent、UActorComponent 的子项）支持基于位置的行为，这类行为不需要几何表示。这包括弹簧臂、摄像机、物理力和约束（但不包括物理对象），甚至音频。

**只有场景组件（USceneComponent 及其子类）可以彼此附加**，因为需要变换来描述子项和父项组件之间的空间关系。虽然场景组件可以拥有任意数量的子项，但只能拥有一个父项，或可直接放置在场景中。场景组件系统不支持附加循环。两种主要方法分别是 SetupAttachment 和 AttachToComponent。前者在构造函数中、以及处理尚未注册的组件时十分实用；后者会立即将场景组件附加到另一个组件，在游戏进行中十分实用。该附加系统还允许将Actor彼此之间进行附加，方法是将一个Actor的根组件附加到属于另一个Actor的组件。

## Primitive组件

Primitive组件（类 UPrimitiveComponent、USceneComponent 的子项）是拥有几何表示的场景组件，通常用于渲染视觉元素或与物理对象发生碰撞或重叠。这包括静态或骨架网格体、Sprite或公告板、粒子系统以及盒体、胶囊体和球体碰撞体积。

**基元组件**（类 UPrimitiveComponent）是包含或生成某类几何的场景组件，通常用于渲染或碰撞。各种类型的几何体，目前最常用的是 **盒体组件**、**胶囊体组件**、**静态网格体组件** 和 **骨架网格体组件**。盒体组件和胶囊体组件生成不可见的几何体进行碰撞检测，而静态网格体组件和骨架网格体组件包含将被渲染的预制几何体，需要时也可以用于碰撞检测。

# 碰撞

## SetCollisionEnabled()

该函数决定了碰撞的用途，不同参数的含义为：

NoCollision：如果是这个，那么其他碰撞设置都会被忽略不计，因为不再考虑碰撞了；

Query Only：仅用于查询，不会考虑物理仿真；

Physics Only：仅用于物理仿真，不考虑查询；

Collision Enabled (Query and Physics)：同时用于查询与仿真；

## SetCollisionResponseToAllChannels()

统一设置碰撞形式（Block、overlap、ignore）

## SetCollisionResponseToChannel( , )

设置针对指定通道的碰撞，第一个参数为发生碰撞的类型，第二个参数为碰撞形式；

## BodyInstance

碰撞组件中存放相关物理信息的部分；

CollisionComponent->BodyInstance.SetCollisionProfileName (*TEXT*("Projectile"));

上述操作可用于设置组件的碰撞形式，其中TEXT()中存放预设好的碰撞形式。

## AddImpulseAtLocation(FVector Impulse, FVector Location, FName name);

第一个参数表示冲击的大小和方向，矢量本身就包含方向信息，模为大小；

第二个参数本身应用冲击的位置；

第三个参数表示如果冲击对象是骨骼体，则此参数表示具体骨骼名称。

## 碰撞响应的两种方式

（1）virtual void NotifyActorBeginOverlap(*AActor*\* OtherActor) override;

重写上述函数；

当这个参与者与另一个参与者重叠时的事件，如一个玩家走进一个触发器。

当对象发生阻塞碰撞时，例如玩家撞到一堵墙，可以查看“Hit”事件。

这个和其他Actor上的组件必须将bGenerateOverlapEvents设置为true才能生成重叠事件。

（2）.AddDynamic()

通过.AddDynamic()将碰撞事件与要调用的函数绑定。

# 发射物移动组件

## [UProjectileMovementComponent](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/3/)

对应头文件：#include "GameFramework//ProjectileMovementComponent.h"

该组件中内置了许多运动相关的参数，如初始速度、最大速度等

## 示例

//通过发射物移动组件驱动发射物移动

if (!ProjectileMovementComponent)

{

ProjectileMovementComponent=CreateDefaultSubobject<UProjectileMovementComponent>(*TEXT*("ProjectileMovementComponent"));

//绑定要控制的组件

ProjectileMovementComponent->SetUpdatedComponent(Collision Component);

//设置初始速度

ProjectileMovementComponent->InitialSpeed=3000.0f;

//设置最大速度

ProjectileMovementComponent->MaxSpeed=3000.0f;

//子弹的旋转将在每一帧中更新，以匹配其速度

ProjectileMovementComponent->bRotationFollowsVelocity=true;

//设置反弹

ProjectileMovementComponent->bShouldBounce=true;

//设置反弹系数，即反弹能力的强弱

ProjectileMovementComponent->Bounciness=0.3f;

//设置重力系数

ProjectileMovementComponent->ProjectileGravityScale=0.0f;

//设置生命周期（多久自动销毁）

InitialLifeSpan=3.0f; //3秒后销毁

}

## 设置发射物速度

ProjectileMovementComponent->Velocity=初始速度\*速度方向；

# 设置镜头移动

## APawn::AddControllerYawInput

UE4 在APawn类中内置了镜头左右旋转的函数，直接与输入绑定即可；

## APawn::AddControllerPitchInput

镜头上下移动

# 弹簧臂组件

## USpringArmComponent

头文件：#include "GameFramework/SpringArmComponent.h"

## 示例

//弹簧臂组件

CamerSpringArm=CreateDefaultSubobject<USpringArmComponent>(*TEXT*("CamerSpringArm"));

CamerSpringArm->SetupAttachment(RootComponent);

//设置弹簧臂长度

CamerSpringArm->TargetArmLength=50.0f;

//开启摄像机延迟（若为真，则摄像机滞后于物体平滑运动）

CamerSpringArm->bEnableCameraLag=true;

//设置摄像机延迟速度

CamerSpringArm->CameraLagSpeed=3.0f;

//相机组件

GunCameraComponent=CreateDefaultSubobject<UCameraComponent>(*TEXT*("GunCameraComponent"));

//确保相机组件不为空，为空则不进行后续操作

check(GunCameraComponent!=nullptr);

//将相机附加到弹簧臂组件

GunCameraComponent->SetupAttachment(CamerSpringArm,USpringArmComponent::SocketName);

# 材质

[UE4 材质系统\_Papals的博客-CSDN博客\_ue4材质](https://blog.csdn.net/qq_23030843/article/details/104206455)

[[浅析]UE4材质基础总结\_鬼泣依旧的博客-CSDN博客\_ue4材质](https://blog.csdn.net/choa12345/article/details/120075714)

## 示例

//直接访问材质文件，并赋值

static ConstructorHelpers::FObjectFinder<UMaterial>Material(*TEXT*("'/Game/Materials/Projectile\_Red.Projectile\_Red'"));

if (Material.Succeeded())

{

//设置材质，并将其与发射物网格体组件绑定

ProjectileMaterialInstance=UMaterialInstanceDynamic::Create(Material.Object,ProjectileMeshComponent);

}

ProjectileMeshComponent->SetMaterial(0,ProjectileMaterialInstance);

# 声音

## USoundBase

UGameplayStatics::PlaySoundAtLocation(this, *FireSound*, GetActorLocation());

在指定位置释放声音。

# 粒子效果

## UParticleSystem

播放粒子效果：

UGameplayStatics::SpawnEmitterAttached()

UGameplayStatics::SpawnEmitterAtLocation()

# 获取组件的Actor

//获取该组件外部链中最接近的该类型

AFPSGun\* C=HitComponent->GetTypedOuter<AFPSGun>();

# 生成Actor

示例：

UWorld\* CurWorld=GetWorld();

FVector Location=GetActorLocation();

FRotator Rotation=GetActorRotation();

if (CurWorld)

{

//参数设置

FActorSpawnParameters SpawnParams;

SpawnParams.Owner=this; //拥有者

SpawnParams.Instigator=GetInstigator(); //施加伤害的对象

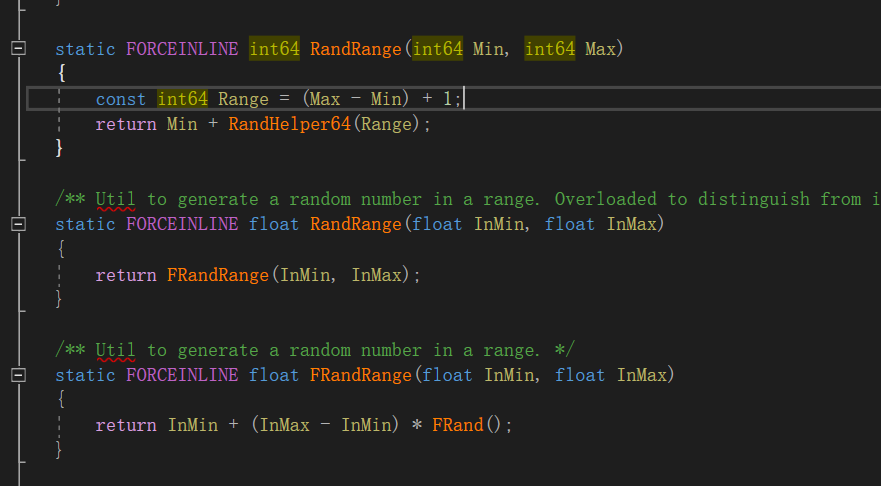
// ActorClassBeSpawned是前面定义的要生成的类型

*AFPSNPC*\* NPC = CurWorld->SpawnActor<*AFPSNPC*>(ActorClassBeSpawned,Location,Rotation,SpawnParams);

}

# 随机数

FMath中预设了诸多生成随机数的API。例如：



# 摄像机

## Control->SetViewTargetWithBlend()

平滑的切换视角，可以是不同类之间的视角切换。

# 弹射角色

LaunchCharacter()

# [AI](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/)（）

## 构建AI的思路

行为树处理决策进程，AI感知系统将源自环境的刺激（例如感官信息）传输到行为树，然后环境查询系统（EQS） 处理关于环境本身的查询。

在BlackBoard中添加行为树相关的参数，以游戏AI的逻辑构建行为树，在AI控制器中根据触发场景设置BlackBoard中的参数，实现行为树分支执行。

# [行为树](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/)

个人理解：根据一系列判断性语句决定AI应执行哪条流程，即通过上述中的环境信息来执行对应行动。（基于轮询的机制）

行为树依赖于一种称为“黑板”的资源，其包含数个用户定义的键，这些键会保存行为树用于进行决策的信息。

## 寻路网格体边界体积（Nav Mesh Bounds Volume）

使我们的AI角色能够在场景中移动，其覆盖范围为AI可自动寻路的范围。

## 黑板（BlackBoard）

AI的“大脑”，存储一些可以反应状态的变量，即决定AI后续操作的变量。可在行为树中使用BlackBoard中的键值来执行不同行为。

黑板中变量的详细信息设置：



## 行为树中节点

（1）



行为树中紫色节点，表示它是一个任务节点。任务节点是你希望行为树执行的具体操作。如果某种原因使行为树让选择器对应的部分子节点皆运行失败，那么任务节点将接管事件。（等待（Wait））

（2）用于辅助判断分支是否可执行等操作。



## 行为树执行顺序

行为树会从左到右和自上而下执行，因此节点的排列很重要。对AI最重要的动作通常应该放在左边，而次要的动作（或退却行为）应该放在右边。子分支会以相同的方式执行，如果任何子分支失败，整个分支将会停止执行，导致失败并返回上级树，此时上级树将其转移到另一支子分支。

## [任务](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeUserGuide/)

行为树节点下可连接具体要执行的任务，任务包括UE4内置任务与自行设计的任务两类。

## 一般命名规则

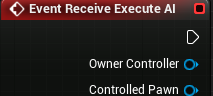
BTT：行为树任务节点（蓝图）

BTD：行为树装饰器节点

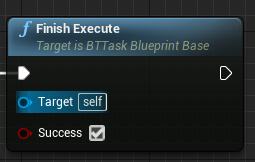
BTS：行为树服务节点

## BTT中节点相关

（1）在行为树中触发此任务时，会触发Event Receive Execte AI 事件节点。



（2）任务结束后，需要添加Finish Execute节点，表示任务节点结束。

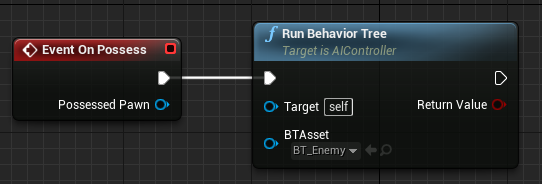


## AI控制器相关

可在控制器内对黑板（BlackBoard）的参数进行设置。

Event On Possess：一旦控制器绑定AI角色便触发；

Run Behavior Tree：绑定对应的行为树。



## AI感知组件（在AI控制器中）

用于在AI感知系统中创建一个刺激监听器，收集可以响应的已注册刺激（视觉、听觉等）。这将使我们可以根据玩家与AI间的一些关系，做出反应。

## [装饰器节点](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceDecorators/)（条件语句）

决定该分支是否可执行。

行为树中的蓝色节点。它连接到一个合成节点，用于验证该黑板键是否为true。这决定了该分支的其余部分是否能够执行。紫色节点是任务节点（Task），是AI可以完成的动作。

## [服务节点](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceServices/)

服务节点是与任意合成节点（选择器节点、序列节点或者简单平行节点）相关联的一种特殊节点，它能够针对指定秒数的每个回调进行注册，并能对多种需要周期性出现的类型进行更新。

以一定的频率执行，通常用于检查和更新黑板

例如：当AI Pawn面对当前敌人、继续在其行为树中正常行动时，可以使用服务节点为该Pawn确定最适合追逐的敌人。

只要执行仍位于服务节点所加入的合成节点的分支树中，服务节点便为活跃状态。

## [合成节点](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceComposites/)

此类节点定义分支的根以及执行该分支的基本规则。

## [任务节点](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceTasks/)

行为树的叶，是可执行的操作，无输出连接。

## 观察者中止

标准平行节点的一个常见用处是不断检查条件，一旦任务所需的条件变成false，则该任务就可以中止。

# [RPC（远程过程调用）](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Overview/)

## 概念

远程过程调用也称为复制函数。可在任何机器上进行调用，但会指示其的实现在与网络会话连接的特定机器上发生。有三种RPC：

（1）Server：仅在主持游戏的服务器上运行；

（2）Client：仅在拥有该函数所属Actor的客户端上运行。若Actor未拥有连接，将不会执行此逻辑；

（3）NetMulticast：在于服务器连接的所有客户端及服务器本身调用。

提供对应 UFUNCTION 宏中的 Server、Client 或 NetMulticast 说明符，可在将C++函数指定为RPC。其代码将在代码实现中使用后缀 \_Implementation。

## 可靠性

必须将RPC指定为可靠或不可靠。在蓝图中，函数和事件默认为不可靠。要将函数指定为可靠，将细节面板（Details Panel）中的可靠（Reliable）设置设为 true。在C++中，必须将 Reliable 或 Unreliable 说明符作为 Server、Client 或 NetMulticast 函数，添加到RPC的 UFUNCTION 宏及其状态。

不可靠RPC无法保证必会到达预定目的地，但其发送速度和频率高于可靠的RPC。其最适用于对gameplay而言不重要或经常调用的函数。例如，由于Actor移动每帧都可能变换，因此使用不可靠RPC复制该Actor移动。

可靠的RPC保证到达预定目的地，并在成功接收之前一直保留在队列中。其最适合用于对gameplay很关键或者不经常调用的函数。相关例子包括碰撞事件、武器发射的开始或结束，或生成Actor。

（对于使用频繁的函数，应将其置为不可靠RPC）

# 多人游戏基本指南

## 基本复制Actor清单

（1）将Actor的复制设置为True；

（2）若复制Actor需要移动，将复制移动（Replicates Movement）设为True；

（3）生成或销毁复制Actor时，确保在服务器上执行该操作；

（4）设置必须在机器间共享的变量，以便进行复制。这通常适用于以gameplay为基础的变量；

（5）尽量使用虚幻引擎的预制移动组件，其已针对复制进行构建；

（6）若使用服务器授权模型，需确保玩家可执行的新操作均由服务器函数触发。

## 网络提示

（1）尽可能少用RPC或复制蓝图函数。在合适情况下改用RepNotify；

（2）组播函数会导致会话中各连接客户端的额外网络流量，需尤其少用。

（3）若能保证非复制函数仅在服务器上执行，则服务器RPC中无需包含纯服务器逻辑。

（4）将可靠RPC绑定到玩家输入时需谨慎。玩家可能会快速反复点击按钮，导致可靠RPC队列溢出。应采取措施限制玩家激活此项的频率。

（5）若游戏频繁调用RPC或复制函数，如tick时，则应将其设为不可靠；

（6）部分函数可重复使用。调用其响应游戏逻辑，然后调用其响应RepNotify，确保客户端和服务器拥有并列执行即可。

（7）检查Actor的网络角色可查看其是否为 ROLE\_Authority。此方法适用于过滤函数中的执行，该函数同时在服务器和客户端上激活。

（8）使用C++中的 IsLocallyControlled 函数或蓝图中的Is Locally Controlled函数，可检查Pawn是否受本地控制。基于执行是否与拥有客户端相关来过滤函数时，此方法十分拥有。

（9）构造期间Pawn可能未被指定控制器，因此避免在构造函数脚本中使用 IsLocallyControlled。

# Actor复制

Actor是实现复制的主要推动者。服务器将保留一份Actor列表并定期更新客户端，以便客户端保留每个Actor（那些需被同步的Actor）的近似复本。

## Actor更新的两种方式

（1）属性更新：可以在发生变化时随时自动更新；

（2）RPC（远程过程调用）：只能在被执行时获得调用更新。

示例：

让所有客户端在某个位置看到同一场爆炸。可以设计以位置和半径为参数的 RPC 函数，同时在每次发生爆炸时调用它。也可以将此存储为一组属性，通过同步的方式将其传达给客户端，但这种做法会损失一些效率，因为爆炸出现的频繁度也许不会高得有必要将它们作为属性。

# [组件复制](https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/Components/)

## 静态组件

随Actor一起创建的组件；即在客户端或者服务器上生成所属Actor时，这些组件也会同时生成，与组件是否被复制无关。静态组件作为默认子对象在C++构造函数中创建，他们将默认存在。

## 动态组件

在运行时在服务器上生成的组件，其创建和删除操作也将被复制到客户端。与静态组件不同，动态组件通过复制的方式存在于所有客户端。另外，客户端可生成自己的本地非复制组件。（只有当那些在服务器上触发的属性或者事件需要自动同步到客户端时，才触发复制行为）

## 组件复制设置

通过调用AActorComponent::SetIsReplicated(true)设置组件为可复制。

## 时间轴

时间轴必须通过其属性中的Replicated选项来启用复制。这会将服务器控制的运行位置、速率和方向复制到客户端。这是一种基本的实施，可以根据需求的变化而进行演变。大多数时间轴都无需复制。和所有游戏对象复本一样，时间轴复本只应当在服务器上直接操作 (start/stop etc)。客户端只应当查看运行位置的复本，而不应尝试改变时间轴本身。在进行复制更新的间歇，客户端将推测 运行位置。

## 一般性的子对象复制

所有Actor子对象都可以复制，而不只限于组件。

这是一种很有用的手段，因为它能在 Actor 通道的层面上使用 UObject 和多态（polymorphism）。之前用于复杂数据结构的复制方法只适合那些 在 Actor 类中对类型进行静态定义的结构。利用子对象复制，可以享受诸多优势，例如建立一个道具栏系统，使其中的每个物品作为一个从基本道具栏类扩展而来的类， 也可以进行完整复制，同时无需让这些项成为 Actor（资源负担太大）。

# 参考文档

[1] [UE4中的C++编程简介 | 虚幻引擎文档 (unrealengine.com)](https://docs.unrealengine.com/4.27/zh-CN/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/)