**2018.9.9**

**UPROPERTY属性修饰符**

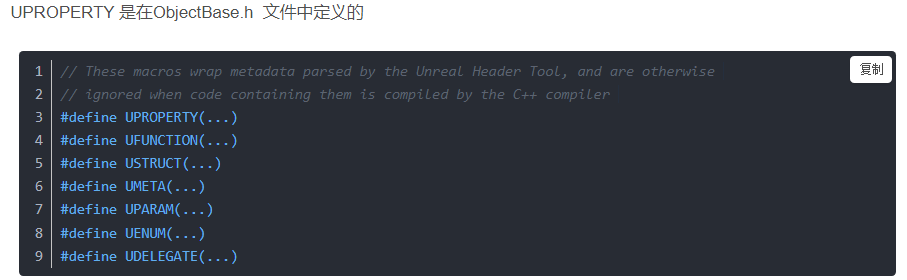
**基本功能**：定义能够被UE垃圾回收机制自动回收的属性变量

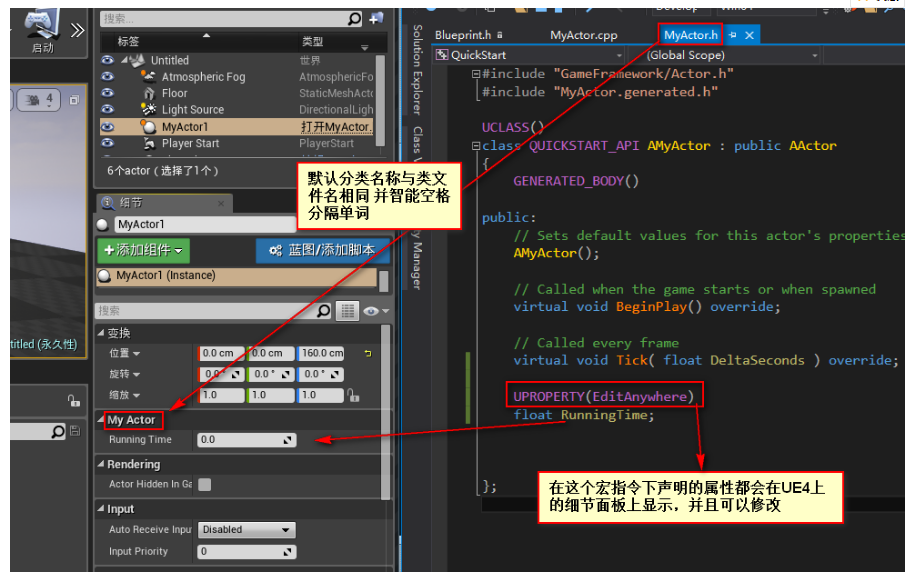
**参数扩展**：以下是官网给的例子解析

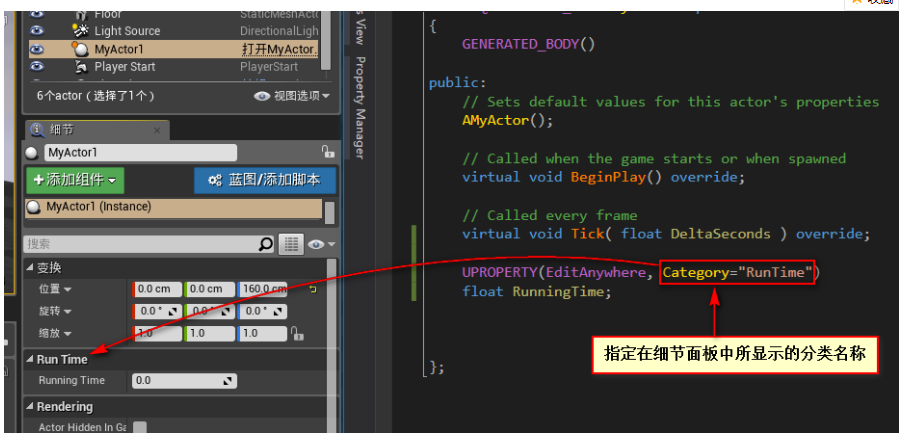
1. UPROPERTY(EditAnywhere)：用此修饰想要暴露在编辑器以便随时编辑的变量（显示在面板上）
2. UPROPERTY(EditAnywhere,Category="Damage")：把被修饰的属性分类到信息面板中Damage那一栏（面板上显示名称为Damage）
3. UPROPERTY(EditAnywhere,BlueprintReadWrite,Category=“Damage”)：支持蓝图对该变量的读写操作（蓝图中也显示）
4. UPROPERTY(BlueprintReadOnly,VisibleAnywhere,Transient,Category="Damage")：支持蓝图读操作，并在编辑器面板中可见，Transient表明不会被保存到硬盘或从硬盘加载

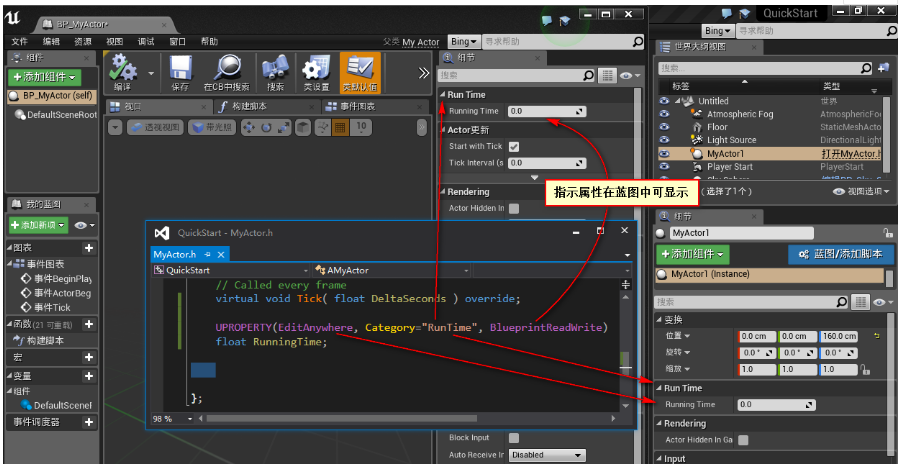
AdvancedDisplay  
属性被显示在细节面板的高级下拉框中。  
  
AssetRegistrySearchable  
AssetRegistrySearchable关键字表明此属性及其值将会为任意将其作为成员变量而包含的资源类示例被自动添加到资源注册中。不可用于结构体属性或参数。  
  
BlueprintAssignable  
仅能用于Multicast代理。应显示该属性，以供在蓝图中分配。  
  
BlueprintCallable  
仅能用于Multicast代理。应显示该属性，以在蓝图代码中调用。  
  
BlueprintReadOnly  
这个属性可以阅读蓝图,但不能修改。这个操作符与BlueprintReadWrite说明符是不兼容的。  
  
BlueprintReadWrite  
这个属性可以读取或写入的蓝图。这个操作符与BlueprintReadOnly说明符是不兼容的。  
  
Category  
定义属性的分类。使用方法: Category=CategoryName. （分类=分类名称）  
  
Config  
表示该变量将会成为可配置状态。当前值可被保存到ini文件中，并且将在创建时被载入。无法在默认属性中被赋值。只读  
  
Const  
这个变量是常量,应该导出为常量。在编辑器中const属性将是不可修改的。  
  
DuplicateTransient  
表示变量值应在任意类型的重复过程中（复制/粘贴，二进制文件复制等）被重置为类默认值。  
  
EditAnywhere  
表示该属性可从编辑器内的属性窗口编辑。  
  
EditDefaultsOnly  
表示该属性可通过属性窗口来编辑，但仅能对原型编辑。  
  
EditFixedSize  
仅限于动态数组。这使得用户不能通过UnrealEd属性窗口来变更数组的长度。  
  
EditInline  
通过此修饰符使得用户可编辑UnrealEd的属性查看器中的变量所引用的对象属性。（仅对对象引用可用，包括对象引用数组）。  
  
EditInstanceOnly  
表示该属性可通过属性窗口来编辑，但仅能对实例而非原型进行编辑。  
  
Export  
仅对对象属性（或对象数组）有效。表示当对象被复制（复制/粘贴）或导出到T3D时，被分配给该属性的对象应完全作为子对象区块来导出，而不是仅仅输出对象引用本身。  
  
GlobalConfig  
类似于config修饰符，区别是您不能在子类中重载它。无法在默认属性中被赋值。 只读 。  
  
Instanced  
仅能用于对象属性。当此类的实例被创建时，它会被赋予一个默认分配给此变量的对象的独特拷贝。用于对在类默认属性中定义的子对象进行实例化。类似EditInline和Export修饰符。  
  
Interp  
表示该值可由Matinee的浮点或向量属性轨迹来随时间驱动。  
  
Localized  
此变量的值将定义本地值。最常用于字符串。只读 。  
  
Native  
属性为native:C++代码负责对其序列化并显示给GC。  
  
NoClear  
防止该对象引用在编辑器中被设置为None.隐藏编辑器的清除（以及浏览）按钮。  
  
NoExport  
仅对native类有效。此变量不应被包含在自动生成的类声明中。  
  
NonTransactional  
表示变更为此变量值将不会被包含在编辑器的撤消/重做历史中。  
  
Ref  
该值在函数调用后被复制出来。仅在函数参数声明中有效。  
  
Replicated  
此变量应通过节点网络进行复制。  
  
ReplicatedUsing  
此变量应通过节点网络进行复制，在其接受到 Callback 函数后执行、  
使用方法: ReplicatedUsing=FunctionName（函数名称）  
  
RepRetry  
仅用于结构体属性。如无法被完全发送，请重试复制此属性（例如，对象引用尚无法通过节点网络来进行序列化）。对于简单引用来说，这是一个默认值，但对结构体来说，由于带宽消耗，很多情况下我们不需要。所以除非此标识被定义，否则其会被禁用.  
  
SaveGame  
SaveGame说明符是一个简单的方法,包括字段明确检查点/保存系统在属性级别。  
  
SerializeText  
Native属性应以文本形式进行序列化（导入文本，导出文本）。  
  
SimpleDisplay  
使属性在细节面板中默认为可见。  
  
Transient  
该属性为临时属性。不应被保存，在载入时会被填零。  
  
VisibleAnywhere  
表示该属性在属性窗口中可见，但根本无法被编辑。  
  
VisibleDefaultsOnly  
表示该属性仅在原型的属性窗口中可见，但无法被编辑。  
  
VisibleInstanceOnly  
表示该属性仅在实例的属性窗口中可见，但对原型则不行，并且无法被编辑。

# UPROPERTY 宏的作用









**UFUNCT ION用法**

**基本功能**：定义能够被UE识别的函数

**参数扩展**：

1. UFUNCTION(BlueprintCallable,Category="Damage")：表明被修饰的函数能够被蓝图调用
2. UFUNCTION(BlueprintImplementableEvent,Category="Damage")：允许在C++代码中调用在蓝图中实现的函数
3. UFUNCTION(BlueprintNativeEvent,Category="Damage")：允许在C++代码中调用在蓝图中实现的函数，同时当蓝图中找不到该实现时，调用代码中定义的默认函数（
   1. **void** AMyActor::CalledFromCpp\_Implementation(){
   2. //当蓝图中找不到时候，会调用这个写在代码里的函数
   3. }

）

函数修饰符:  
  
BlueprintAuthorityOnly  
如无网络权限，则该函数将不会从蓝图代码中执行  
  
BlueprintCallable  
该函数可在蓝图或关卡蓝图图表内执行。  
  
BlueprintCosmetic  
此函数为修饰函数而且无法运行在专属服务器上  
  
BlueprintImplementableEvent  
此函数可以在蓝图或关卡蓝图图表内进行重载。  
  
BlueprintNativeEvent  
此函数将由蓝图进行重载，但同时也包含native类的执行。提供一个名称为[FunctionName]\_Implementation的函数本体而非  
[FunctionName];自动生成的代码将包含转换程序,此程序在需要时会调用实施方式  
  
BlueprintPure  
此函数不会以任何方式影响其从属对象，并且可在蓝图或关卡蓝图图表中执行。  
  
Category  
当在蓝图编辑工具中显示时，定义函数的分类。 \* 用法：Category=CategoryName（分类名称）或   
Category="MajorCategory,SubCategory" （主分类，子分类）  
  
Client  
此函数仅在该函数从属对象所从属的客户端上执行。提供一个名称为[FunctionName]\_Implementation的函数主体，而不是  
[FunctionName]; 自动生成的代码将包含一个转换程序来在需要时调用实现方法。  
  
CustomThunk  
UnrealHeaderTool（虚幻头文件工具）的代码生成器将不会为此函数生成execFoo转换程序; 可由用户来提供。  
  
Exec  
此函数可从游戏中的控制台中执行。Exec命令仅在特定类中声明时才产生作用。  
  
NetMulticast  
无论Actor的NetOwner为何值，此函数都会在服务器上被本地执行且将被复制到所有的客户端  
  
Reliable  
此函数在网络间进行复制，并会忽略带宽或网络错误而被确保送达。仅在与客户端或服务器共同使用时可用  
  
Server  
此函数仅在服务器上执行。提供一个名称为[FunctionName]\_Implementation的函数主体，而不是[FunctionName]; 自动生成的代码  
将包含一个转换程序来在需要时调用实现方法。  
  
Unreliable  
此函数在网络间复制，但可能会由于带宽限制或网络错误而传送失败。仅在与客户端或服务器一起使用时有效。  
  
元数据修饰符:  
BlueprintInternalUseOnly  
此函数为内部实现细节，被用来实现另一个函数或节点。它从不在图表中直接展现。  
  
BlueprintProtected  
此函数仅能在蓝图中针对‘此'实例进行调用。无法针对另一个实例进行调用。  
  
DeprecatedFunction  
该函数被废弃，任何引用它的蓝图都会产生一个编译警告。  
  
DeprecationMessage  
提供废弃函数的自定义信息。  
  
UnsafeDuringActorConstruction  
在Actor构建时，调用此函数会出现问题。

**定时**

//定义一个时间语柄

FTimerHandle TimerHandle\_名;

//计时 ClearTimer(清除定时器)停止原先的定时器，并重新设定一个新的定时器

GetWorldTimerManager().ClearTimer(定时语柄);

//绑定函数，几秒后执行函数

GetWorldTimerManager().SetTimer(定时语柄,this,&A类名::绑定的函数,3.0f（倒计时），延迟);

声明时间头FTimerHandle Hand（可以自己定义）；

时间头可以控制定时器的停止暂停等功能 。

绑定函数GetWorldTimerManager（）.SetTimer（Hand，this，&XXX::Fun, 3.0f, false ，-1.0f）;

参数为 时间头 调用对象指针 调用函数 等待时间 是否循环调用 首次执行延时。

清除记时器GetWorldTimerManager（）.ClearTimer（Hand）；

暂停GetWorldTimerManager（）.PauseTimer（Hand）；

取消暂停GetWorldTimerManager（）.UnPauseTimer（Hand）；

<https://www.cnblogs.com/wodehao0808/category/1023012.html>

**2018.9.14**

**http://api.unrealengine.com/latest/INT/GettingStarted/FromUnity/index.html**

1,给角色添加组件：

      UCameraComponent\*Camera=CreateDefaultSubobject<UCameraComponent>(TEXT("Camera0"));

2,字符串组合（蓝图中组合字符串）：

   BuildString， string +int/float/Object/vector2/boolean……+ string

3.将属性公开到编辑器(UE4 Doc)

  UPROPERTY(EditAnyWhere,BlueprintReadWrite,Category="Damage")

  UPROPERTY(BlueprintReadOnly,VisibleAnywhere,Transient,Category="Damage")

  VisibleAnywhere标记意为在编辑器中可见，但不能进行编辑

  Transient意为无法从硬盘保存或加载，它应该为派生的非持久值

  UE4中蓝图中显示C++函数：

  UFUNCTION(BlueprintCallable,Category="Damage")

  C++/蓝图混合实现同一种函数：

  UFUNCTION(BlueprintNativeEvent,Category="Damage")

  void CalledFromCpp();

  CalledFromCpp\_Implementation(){}

  当你创建的类不是继承自Actor时，要找到你创建的UCLASS，你必须关掉Class Viewer中的过滤器中的Actors Only（默认的情况下是被检测的，也就是说会过滤掉非继承自Actor的自定义UCLASS）。

  用FStringAssetReferences 和 StaticLoadObject加载资源也是一种选择方式

  可以通过全局函数ConstructObject<> 或者NewObject<>来创建继承自UObject的类

  UPROPERTY(EditAnywhere,BlueprintReadWrite,Category=UClassNames)

  TSubclassOf<UUserProfile> UPBlueprintClassName;

  ObjectType\* object = ConstructObject<ObjectType>(UClassReference);

  例如：

  AChapterGameMode\*gm=Cast<AChapterGameMode>(GetWorld()->GetAuthGameMode());

  if(gm)

  {UUserProfile\* object = ConstructObject<UUSerProfile>(gm->UPBlueprintClassName);}

  //NewObject用法

  UProfile\* object = NewObject<UProfile>(GetTransientPackage(),uclassReference);

  删除一个非使用的UObject:

   objectInstance->ConditionalBeginDestroy();

   强制GC进行资源回收的方式：

   GetWorld()->ForceGarbageCollection(true)

   结构体的写法：

   #pragma once

   #include "Chapter2.h"

   #include "ColoredTexture.generated.h"

  USTRUCT()

  struct CHAPTER2\_API FColoredTexture

  {

  GENERATED\_USTRUCT\_BODY()

  public:

  UPROPERTY( EditAnywhere, BlueprintReadWrite, Category =

  HUD )

 UTexture\* Texture;

 UPROPERTY( EditAnywhere, BlueprintReadWrite, Category =

  HUD )

  FLinearColor Color;

  }

  枚举的写法：

UENUM()

enum Status

{

Stopped UMETA(DisplayName = "Stopped"),

Moving UMETA(DisplayName = "Moving"),

Attacking UMETA(DisplayName = "Attacking"),

};

使用方法：

 UPROPERTY(EditAnywhere,BlueprintReadWrite,Category=Status)

TEnumAsByte<Status> status;

继承自Actor的类使用NewObject<>和SpawnActor<>方式进行实例化，例如：

UAction\* action = NewObject<UAction>(GetTranientPackage(),UAction::StaticClass());

UObject类的回收（你也可以通过设置引用计数为0的方式来达到释放的目的）：

UObject \*o  = NewObject<UObject>(……);

o->ConditionalBeginDestory();

或者通过：

GetWorld->ForceGarbageCollection(true);

的方式也可以达到同样的目的。

UObject和它的派生类（创建的任何东西NewObject或ConstructObject）不能使用TSharedPtr！

TSharedPtr,TSharedRef,TWeakPtr,TAutoPtr基本使用方法：

class MyClass{}

TSharedPtr<MyClass> sharedPtr(new MyClass());

weak pointers 与 shared pointers之间有一些不同，弱指针没有能力保留对象的内存当引用计数到0时

弱指针的好处是它回收内存后，指针的引用会变成NULL，可以通过检测弱指针是否可用来进一步处理：

if(ptr.IsValid())

{

}

那些继承自UObjects的类不能使用TSharedRef

TScopedPointer<AWarrior>warrior(this);

TScopedPointer在作用域内避免被销毁的指针类型，也是通过引用计数的方式来进行管理的

生成继承自Actor的对象，例：

SpawnActor = GetWorld()->SpawnActor<AChapter04Actor>(AChapter04Actor::StaticClass(),SpawnLocation);

定时器的基本使用方法：

FTimerHandle Timer;

GetWorldTimerManager().SetTimer(Timer,this,&AUE4GameMode::DestroyActorFunction,10);

设置对象的生命周期可以对Actor进行销毁控制：

SetLifeSpan(10);

初始化物体组件：

auto MeshAsset = ConstructorHelpers::FObjectFinder<UStaticMesh>(TEXT("Static Mesh'/Engine/BasicShapes/Cube.Cube'"));

if(MeshAsset.Object!=nullptr)

{

Mesh->SetStaticMesh(MeshAsset.Object);

}

需要在头文件中加入：

#include “ConstructorHelpers.h”

Debug信息到屏幕上：

GEngine->AddOnScreenDebugMessage(-1, 1, FColor::Red,

FString::Printf(TEXT("%s left me"), \*(OtherActor-

>GetName())));

1、委托写法(无参)：

DECLARE\_DELEGATE(FStandardDelegateSignature)

定义成员：

FStandardDelegateSignature  MyStandardDelegate;

绑定：

MyStandardDelegate.BindUObject(this,&ADelegateListener::EnableLight);

简单的C++类对象函数绑定要用BindRaw

静态函数的绑定：BindStatic

执行：

MyStandardDelegate.ExecuteIfBound();

解除当前的绑定：

MyStandardDelegate.Unbind();

2、委托写法（含参）：

DECLARE\_DELEGATE\_OneParam(FParamDelegateSignature,FLinearColor)

|  |  |
| --- | --- |
| DECLARE\_EVENT( OwningType, EventName ) | 创建一个事件。 |
| DECLARE\_EVENT\_OneParam( OwningType, EventName, Param1Type ) | 创建带一个参数的事件。 |
| DECLARE\_EVENT\_TwoParams( OwningType, EventName, Param1Type, Param2Type ) | 创建带两个参数的事件。 |
| DECLARE\_EVENT\_<Num>Params( OwningType, EventName, Param1Type, Param2Type, ...) | 创建带 N 个参数的事件。 |

添加成员：

FParamDelegateSignature MyParamDelegate;

绑定：

同无参绑定类似

执行：

MyParamDelegate.ExecuteIfBound(FLinearColor)

3,多播委托的写法：

DELCARE\_MULTICAST\_DELEGATE(FMulticastDelegateSignature)

添加成员到指定的类中:

FMulticastDelegateSignature FMultiDelegate;

绑定类似：

FMultiDelegate->AddUObject(this,AMulticastDelegateListener::ToggleLight);

执行：

FMultiDelegate->Broadcast();

解除绑定：

FMultiDelegate->Remove(AMulticastDelegateListener::ToggleLight);

|  |  |
| --- | --- |
| DECLARE\_MULTICAST\_DELEGATE[\_Const, \_RetVal, etc.]( DelegateName ) | 创建一个多播的多播代理。 |
| DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE[\_Const, \_RetVal, etc.]( DelegateName ) | 创建一个动态的多播代理。 |
| DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE[\_Const, \_RetVal, etc.]( DelegateName ) | 创建一个封装的动态多播代理。 |

4,自定义事件：

DECLARE\_EVENT(AMyTriggerVolume,FPlayerEntered)

定义：

FPlayerEntered OnPlayerEntered;

执行：

OnPlayEntered.Broadcast();

绑定：

OnPlayEntered.AddUObject(this,&ATriggerVolEventListener::OnTriggerEvent);

获取当前关卡中所有的指定类型AActor：

 UGameplayStatics::GetAllActorsOfClass(const UObject\* WorldContextObject, TSubclassOf<AActor> ActorClass, TArray<AActor\*>& OutActors)

例如：

TArray<AActor\*> TimeOfDayHandlers;

UGameplayStatics::GetAllActorsOfClass(GetWorld(), ATimeOfDayHandler::StaticClass(), TimeOfDayHandlers);

UE4定时器的使用方法：

FTimerHandle MyTimer;

GetWorld()->GetTimerManager().SetTimer(MyTimer,this,&APickupSppawner::SpawnPickup,10,false);

UE4中的迭代器的用法：

AActor的迭代器位于“EngineUtil.h”头文件中，基本使用方法：

for(TActorIterator<AActor> It(GetWorld(),AActor::StaticClass());It;++It)

{

   AActor\* Actor = \*It;

   IMyInterface\* MyInterfaceInstance = Cast<IMyInterface>(Actor);

   if(MyInterfaceInstance)

   {

      MyInterfaceInstances.Add(MyInterfaceInstance);

   }

}

给指定的静态网格物体组件附加默认模型文件：

auto MeshAsset = ConstructorHelpers::FObjectFinder<UStaticMesh>(TEXT("Static Mesh'/Engine/BasicShapes/Cube.Cube'"));

if(MeshAsset.Object!=nullptr)

{

   MyMesh->SetStaticMesh(MeshAsset.Object);

}

UINTERFACE(meta=(CannotImplementInterfaceInBlueprint)) 可解决与接口方法UNFUNCTION(BlueprintCallable)的冲突问题

如果非实现的方法，可以在函数里写入unimplemented();比如说接口的方法。

暴露接口供蓝图使用：

首先，在接口中声明函数（当然为了避免蓝图可继承错误，需要借助上面的元数据声明,函数必须为虚函数，这个是从一个帖子上看到的，<https://www.2cto.com/kf/201608/532601.html>）

UFUNCTION(BlueprintCallable,Category = Test)

virtual void OnPostBeginPlay();

接着可以在cpp文件中实现该函数，

其他的继承用法和常规C++实现方式类似，就不记录了。

接口在蓝图中的实现：

主要的声明部分：

UFUNCTION(BlueprintImplementableEvent,BlueprintCallable,Category=AttackAvoider)

void AttackInComing(AActor\* AttackActor);

蓝图中覆盖C++中实现的接口函数：

主要的声明部分;

 UFUNCTION(BlueprintNativeEvent,BlueprintCallable,Category=Wearable)

void OnEquip(APawn\* Wearer);

函数的主体部分：

void IWearable::OnEquip\_Implementation(APawn\* Wearer)

{

}

继承子类的写法：

virtual void OnEquip\_Implementation(APawn\* Wearer) override

{

   IWearable::OnEquip\_Implementation(Wearer);

}

C++调用蓝图实现的接口：

基本是通过<IInterface>::Execute\_<FunctionName>的方式完成调用的

继承自UObject的对象可以通过声明UCLASS（BlueprintType）的方式来让蓝图识别并作为变量使用

UCLASS（Blueprintable）可以让蓝图继承该类

可以同时用UCLASS（Blueprintable，BlueprintType）的方式让该类即能作为蓝图变量使用又能作为可继承的基类使用

声明方法为UFUNCTION(BlueprintImplementableEvent)可以在蓝图中实现该事件并进行调用

向蓝图中暴露多播委托

首先声明一个多播委托：

DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE\_OneParam(FOnKingDeathSignature,AKing\*,DeathKing);

其次在类中声明委托变量：

UPROPERTY(BlueprintAssignable)

FOnKingDeathSignatrue OnKingDeath;

函数中调用：

OnKingDeath.Broadcast(this);

在蓝图可以挂在当前的多播事件，并通过调用OnKingDeath.Broadcast的函数完成调用

解绑绑定的函数，可以通过UnBind系列的函数完成

枚举在蓝图中的公开及使用：

1， UENUM(BlueprintType)

      enum TreeType{Tree\_Poplar,Tree\_Spruce,Tree\_Eucalyptus,Tree\_Redwood};

2,在类中的声明：

     UPROPERTY(BlueprintReadWrite)

     TEnumAsByte<TreeType> Type;

属性在细节面板中可见性及修改的控制：

        UPROPERTY(EditDefaultsOnly) //默认蓝图类列表中细节属性可修改

              bool EditDefaultsOnly;

       UPROPERTY(EditInstanceOnly)//实例化的蓝图类的细节列表中可修改

              bool EditInstanceOnly;

       UPROPERTY(EditAnywhere) //以上两者都能进行修改

              bool EditAnywhere;

       UPROPERTY(VisibleDefaultsOnly)//可见性控制，默认蓝图类列表中可见

              bool VisibleDefaultsOnly;

       UPROPERTY(VisibleInstanceOnly)//实例化的细节列表中可见

              bool VisibleInstanceOnly;

       UPROPERTY(VisibleAnywhere)//以上两者皆可见

              bool VisibleAnywhere;

属性在蓝图中调用及修改的基本控制：

     UPROPERTY(BlueprintReadWrite,Category=Cookbook)//读写

     bool ReadWriteProperty;

     UPROPERTY(BlueprintReadOnly,Category=Cookbook)//读

     bool ReadOnlyProperty;

C++中添加蓝图中的Construction事件脚本：

virtual void OnConstruction(const FTransform& Transform) override;

主要在该函数中完成。

创建一个新的编辑器模块：

1,在你的UE4 项目中的uproject文件中找到Module模块配置参数：加上实例类似的内容（一个完整的模块配置参数）：

{"Name":"UE4CookBookEditor","Type":"Editor","LoadingPhase":"PostEngineInit","AdditionalDependencies":["Engine","CoreUObject"]}

2,在你的项目源码目录文件夹中创建一个新的文件夹命名为：UE4CookBookEditor，并添加文本配置文件：UE4CookBookEditor.Build.cs

3,在文本配置文件中添加以下代码：

using UnrealBuildTool;

public class UE4CookBookEditor : ModuleRules

{

    public UE4CookBookEditor(TargetInfo Target)

    {

        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore","RHI", "RenderCore", "ShaderCore" });

        PublicDependencyModuleNames.Add("CookBook");

        PrivateDependencyModuleNames.AddRange(new string[] { "UnrealEd"});

    }

}

4，创建基本组件模块的头文件：UE4CookBookEditor.h和UE4CookBookEditor.cpp实现文件，他们的具体内容如下：

//头文件内容

#pragma once

#include "Engine.h"

#include "ModuleManager.h"

#include "UnrealEd.h"

class UE4CookBookEditorModule:public IModuleInterface

{

};

//实现文件的具体内容

#include "UE4CookBookEditor.h"

IMPLEMENT\_GAME\_MODULE(FUE4CookBookEditorModule,UE4CookBookEditor)

关闭VS,右键点击你的UE项目，选择Generate Visual Studio Project files文件

新添加的模块是不支持热更新的，就像运行时代码一样，如果你得到一个编译错误，其中提到了对生成的头文件的更改，只需关闭编辑器，然后从IDE中重新构建。

自定义资源类型的写法：

1，首先建立一个资源类，继承自UObject,例如：

#pragma once

#include "Object.h"

#include "MyCustomAsset.generated.h"

UCLASS()

class COOKBOOK\_API UMyCustomAsset :public UObject

{

       GENERATED\_BODY()

public:

       UPROPERTY(EditAnywhere, Category = "Custom Asset")

              FString Name;

};

2,创建继承自UFactory的自定义资源工厂，并重写创建函数FactoryCreateNew：

#pragma once

#include "Factories/Factory.h"

#include "CustomAssetFactory.generated.h"

UCLASS()

class COOKBOOK\_API UCustomAssetFactory :public UFactory

{

       GENERATED\_BODY()

public:

       UCustomAssetFactory();

       virtual UObject\* FactoryCreateNew(UClass\* InClass, UObject\* InParent, FName InName, EObjectFlags Flags,UObject\* Context, FFeedbackContext\* Warn, FName CallingContext) override;

};

3，实现文件：

#pragma once

#include "CustomAssetFactory.h"

#include "CookBook.h"

#include "MyCustomAsset.h"

UCustomAssetFactory::UCustomAssetFactory() :Super()

{

       bCreateNew = true;

       bEditAfterNew = true;

       SupportedClass = UMyCustomAsset::StaticClass();

}

UObject\* UCustomAssetFactory::FactoryCreateNew(UClass\* InClass, UObject\* InParent, FName InName,EObjectFlags Flags, UObject\* Context, FFeedbackContext\* Warn, FName CallingContext)

{

       auto NewObjectAsset = NewObject<UMyCustomAsset>(InParent,InClass,InName,Flags);

       return  NewObjectAsset;

}

4，别忘了添加模块引用 “UnrealEd",否则会出现链接错误！

UE4中关于文件读取的一些内容：

判定当前文件是否存在以及文件大小的本地化方式存储在头文件一下头文件中：

|  |
| --- |
| Runtime/Core/Public/GenericPlatform/GenericPlatformFile.h |

|  |
| --- |
| Runtime/Core/Public/HAL/PlatformFilemanager.h |

如果你想得到文件的大小，你可以这样做：

FPlatformFileManager::Get().GetPlatformFile().FileSize(\*MyFilePath);//当然你也可以直接调取GenericPlathformFile.h头文件来完成，只不过通过FPlatformFileManager可以通过一层单例模式获取

如果你要写文件读取相关的内容，你可以通过以下头文件和模块找到你想要的头文件：

#include "Paths.h" #include "Http.h"

FPlatformFileManager::Get().GetPlatformFile()下拥有打量的文件操作函数，其中包括，增删文件夹，创建文件，修改内容，移动文件，查询文件大小等

程序中控制鼠标的显示：

GetWorld()->GetFirstPlayerController()->bShowMouseCursor=true;

UE4 Log相关

UE\_Log(LogTemp,Warning,TEXT("Message %d"),1);

自定义Log：

首先打开项目头文件：

包含Engine头文件#include“Engine.h”

DECLARE\_LOG\_CATEGORY\_EXTERN(LogCustom,Log,All);

第一个参数：CategoryName:Log的名称

第二个参数：DefaultVerbosity:自定义的Log打印输出信息

第三个参数：CompileTimeVerbosity:这是对编译后的代码进行烘焙后的一段代码（有道翻译，没理解什么意思）

Message Log相关：

#define LOCTEXT\_NAMESPACE "CustomLogMessage"

extern FName LoggerName;

extern FMessageLog Logger;

#define FTEXT(x) LOCTEXT(x,x)

FName LoggerName("CustomLog")

FMessageLog CreateLog( FName name )

{

FMessageLogModule& MessageLogModule =

FModuleManager::LoadModuleChecked<FMessageLogModule>

("MessageLog");

FMessageLogInitializationOptions InitOptions;

InitOptions.bShowPages = true;// Don't forget this!

InitOptions.bShowFilters = true;

FText LogListingName = FTEXT( "Chapter 12's Log Listing"

);

MessageLogModule.RegisterLogListing( LoggerName,

LogListingName, InitOptions );

}

AChapter12GameMode::AChapter12GameMode()

{

//这个没有测试，估计是上面函数的调用，可能要改成CreateLog(LoggerName);

CreateLogger( LoggerName );

// Retrieve the Log by using the LoggerName.

FMessageLog logger( LoggerName );

logger.Warning(

FTEXT( "A warning message from gamemode ctor" ) );

}

调用

Logger.Info(FTEXT("Info Log"));

Logger.Warning(FTEXT("Warning Log"));

Logger.Error(FTEXT("Error Log"));

FMessageLog( LoggerName ).Info( FTEXT( "An info message") );

屏幕输出信息：

#include“Engine.h”

GEngine->AddOnScreenDebugMessage(-1,5.f,FColor::Red,TEXT("Send your Message to Screen!"));

旋转角的知识点：

FRotator(FVector);

FQuat(FVector Axis,float AngleRed); //四元数，FVector指的是旋转轴，angleRed指的是旋转的角度

**客户端**

SetReplicates(true); //复制把主机上的东西复制到客户端上

SetReplicateMovement(true); //设置复制品移动

Role == ROLE\_Authority//检查玩家是否在服务器上运行代码或是否主机玩家的操作//Role(角色)ROLE\_Authority(角色授权)

(!IsLocallyControlled())//检查是否仅受本体控制

FRotatorNewRot=CameraComponent->RelativeRotation;IsLocallyControlled//CameraComponent(摄像机组件) RemoteViewPitch（不能为负值）

GameMode 不能再客户端运行复制可用 GameState 配合使用

AFPSGameState\* GS = GetGameState<AFPSGameState>();类型转换括号里可以为空（特殊）

**客户端的函数定义：**

.h

// UFUNCTION(虚拟函数) 和 server(服务器) Reliable(可靠链接) WithValidation(进行验证)

UFUNCTION(Server, Reliable, WithValidation)

void ServerFire();

.cpp

//联机代码 \_Implementation(实现)

void AFPSCharacter::ServerFire\_Implementation()

{

}

// 当客户端发起return false(返回假值)这一反馈或请求时服务器会取消连接，因为出现了问题（比如：外挂或出现严重问题等会强行断开连接）

bool AFPSCharacter::ServerFire\_Validate()

{

return true;

}

AI 逻辑只在服务器上运行

//检查玩家是否在服务器上运行代码或是否主机玩家的操作

//Role(角色)ROLE\_Authority(角色授权)

if (Role == ROLE\_Authority){}

头文件 #include "Net/UnrealNetwork.h"

void AFPSAIGuard::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps)const

{

（配合使用确保所有客户端都能运行）

Super::GetLifetimeReplicatedProps(OutLifetimeProps);

DOREPLIFETIME(AFPSAIGuard, 要复制的变量);

}

/\*↓ 不懂 服务器 变量改变是则服务端，客户端会调用该函数\*/

UPROPERTY(ReplicatedUsing = OnRep\_GuardState)

EAIState GuardState;

UFUNCTION()

void OnRep\_GuardState();

/\*↑ 不懂\*/

// Replicated 复制变量到客户端

UPROPERTY(Replicated, BlueprintReadOnly, Category = "Gameplay")

bool bIsCarryingObjective;

人物枪口的指向

（与理解的不同）

//检查是否仅受（本地）控制 IsLocallyControlled

if (!IsLocallyControlled())

{

//CameraComponent(摄像机组件)

FRotator NewRot = CameraComponent->RelativeRotation;

/\*

该变量在系统中已有复制，我们可以利用它，观测另一位玩家是否向上向下看

RemoteViewPitch（不能为负值）

\*/

NewRot.Pitch = RemoteViewPitch \* 360.0f / 255.0f;

CameraComponent->SetRelativeRotation(NewRot);

}

**联机**

// 联机时的输入

// Open firstPersonExampleMap ? listen

// Open : 网段

**AI即行为树**

黑板是存放行为树运行时所需要的数据。

行为树的运行时从左到右，也就是说您要做某种决定的话，必须将优先级最高的任务放在左边，也就是最重要的任务

Sequence（ 序列；顺序；续发事件） 顺序执行

Selector （选择器；挑选者） 节点执行，如果有两个的话，要么用这个，要么用那个

FPSGame.Build.cs（文件名）

PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore","AIModule（要添加的否则AI无法读取出来）" });

**团体类（枪战）游戏总结**

按下 Alt +G 跳转到函数定义地方

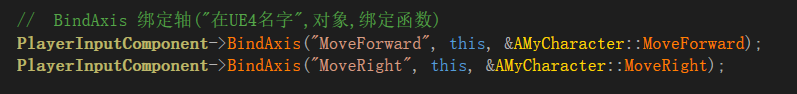
给人物添加移动操作：

PlayerInputComponent (玩家组件)

BindAxis (绑定轴(方向性的操作))

使用：

PlayerInputComponent->BindAxis(“名字(在UE4项目设置-输入-AxisMappings(映射轴)-对应的名字)”,This(指当前的类,&A自己的名字::要绑定的函数名))



AddMovementInput()(添加移动输入)

WorldDirection()(世界方向)

GetActorForwardVector()(获得类的向前的矢量) ForwardVector()(向前的矢量)



GetActorRightVector()(获得类的向前的矢量) RightVector() (向右的矢量)



// Value 价值，价格 函数定义是的局部变量，在这里接受传回来的值控制方向

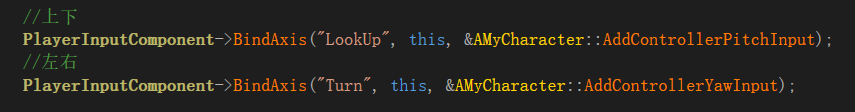
Ctrl + S 保存

Ctrl + Shift + S 全部保存

给人物添加视角操作：

AddControllerPitchInput(添加控制器的俯仰角输入：上下看)(内置函数) 是Y轴

AddControllerYawInput(添加控制器的俯仰角输入：左右看)(内置函数) 是X轴



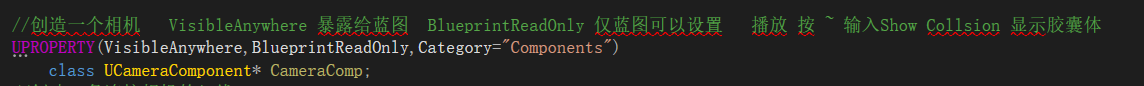
给人物添加第三人称视角：

Spring Arm Component(弹簧臂组件)

UCameraComponent(摄像机组件)（类型 : 类） （Camera（相机） Component（组件））

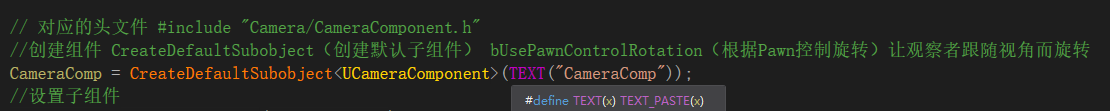
使用：

UCameraComponent\* 组件名



CreateDefaultSubobject(创建默认子组件)(为人物设置一个子组件)

使用：



CreateDefaultSubobject<填写类型UCameraComponent >(TEXT(“设置UR4中的名字”))

BUsePawnControlRotstion = True

(让摄像机能正常地跟随换看着者的视角而转动，头文件：Camera/CameraComponent.h)

USpringArmComponent(弹簧臂组件)(头文件 GameFramework/SpringArmComponent.h)

IE\_Pressed(按下键盘时)

IE\_Released(松开键盘时)

给人物添加动作：

启动对蹲伏功能的支持:

GetMovementComponent(得到运动组件)

GetNavAgentPropertiesRef(获取导航代理属性参考)\

BCanCrouch(能否蹲伏: 设置真假值)

头文件：GameFramework/PawnMovementComponent.h



创建武器（重点）：

USkeletalMeshComponent(虚幻骨骼网络体组件)（类）

FHitResult **Hit**;（这一结构中，能填写一系列数据，让我们知道如击中了什么物体，距离多远以及击中方向等有用的信息）

GetWorld()获得世界坐标

LineTraceSingleByChnnel(Hit,起始点，终点，碰撞通道)（单轨迹线）

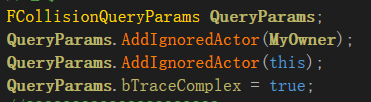
GetOwner(获得用户输入:好像是获取人的骨骼输入)(属于：actor类型)

GetActorEyeViewPoint(起始点，角度)(获取眼睛的位置)

FCollisionQueryParams(碰撞检测)

.AddIgnoredActor()(忽略谁的碰撞)

.bTraceComplex = (true/false) (是否启动精确碰撞)



绘制线：头文件DrawDebugHelpers.h头文件

DrawDebugLine(GetWorld(),起始点，终点，FColor::颜色，是否始终出现(true/false)，显示的持续时间，？，线的粗细)(绘制调试线)

添加开火特效：  
GetComponenentLocation(获得组件位置 )

Super::函数名()(继承原函数的定义)

**常用快捷键**

VS

Ctrl + Shift + F 查找和替换

Alt + O 切换头文件 .cpp/.h

Alt + Shift + O 搜素UE4中的文件

Alt + Shift + S 可用于查询任意符号

Alt + J 跳转到声明函数的头文件

UE4

Alt + P 开始播放

**添加粒子效果**

// UParticleSystem 粒子系统指针

UPROPERTY(EditDefaultsOnly, Category = "FX")

class UParticleSystem\* ExplosionEffect;

// 头文件 #include "Kismet/GameplayStatics.h"

// 添加粒子系统 UGameplayStatics

UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ExplosionEffect, GetActorLocation());

**创建类对象**

头文件 #include "Components/SphereComponent.h"

// 球体组件 USphereComponent

->SetSphereRadius(3000);

//设置圆的半径

头文件 #include "Components/BoxComponent.h"

// 盒体组件 UBoxComponent

->SetBoxExtent(FVector(200.0f,200.0f, 200.0f));

//设置物体的 X,Y,Z 的宽和高 FVector（200.0f）或 FVector(200.0f,200.0f, 200.0f)

头文件 #include "Components/StaticMeshComponent.h"

// 静态网格体组件 UStaticMeshComponent

头文件 #include "Components/DecalComponent.h"

//贴花 UDecalComponent

->DecalSize = FVector(200.0f, 200.0f, 200.0f);

//设置物体的 X,Y,Z 的宽和高 FVector（200.0f）或 FVector(200.0f,200.0f, 200.0f)

//音基 USoundBase

头文件 #include "Kismet/GameplayStatics.h"

//播放音乐 UGameplayStatics::PlaySound2D(this,创建的音基名);

头文件 #include "Perception/PawnSensingComponent.h"

UPawnSensingComponent 人体感应组件(作用：听觉，视觉方面的设置 )

UPawnNoiseEmitterComponent 人体发音组件

谁要发出声音添加 (制造声音)MakeNoise(1.0f（声音响度）,声音的发出者)   
只有拥有人体发音组件才能发出声音

在人物发射函数里添加 ActorSpawnParams.Instigator = this 设置子弹为发出声音的来源

**创建对象**

创建默认对象 CreateDefaultSubobject

对象名 = CreateDefaultSubobject<类型>(TEXT("对象名（在UE4显示）"));

**设置碰撞**

// ->SetCollisionEnabled(ECollisionEnabled::NoCollision); 注释：没有碰撞

// ->SetCollisionEnabled(ECollisionEnabled::PhysicsOnly); 注释：只有物理

// ->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics); 注释：查询和物理

// ->SetCollisionEnabled(ECollisionEnabled::QueryOnly); 注释：只有查询

// ->SetCollisionEnabled(ECollisionEnabled::Type); 注释：目前理解为自定义

->SetCollisionEnabled(ECollisionEnabled::QueryOnly);

// 设置所有通道的冲突响应 SetCollisionResponseToAllChannels

->SetCollisionResponseToAllChannels(ECR\_Ignore); // ECR\_Ignore 全部忽略

//将冲突响应设置为通道 SetCollisionResponseToChannel

// ECC\_Pawn 人，ECR\_Overlap 重叠, ECR\_Block 物块，ECR\_Ignore 忽略, ECR\_MAX ?

->SetCollisionResponseToChannel(ECC\_Pawn, ECR\_Overlap);

**碰撞相关的函数UE4自带**

// NotifyActorBeginOverlap 通知（Notify）类开始重叠

绑定函数

OverlapComp->OnComponentBeginOverlap.AddDynamic(this, &AFPSExtractionZone::HandleOverlap);

// OnComponentBeginOverlap 触发重叠时 绑定的函数需要特定的参数（可以用Alt + G实现查询）

// 为 OverlapComp 绑定函数 HandleOverlap，基本结构为 组件名->OnComponentBeginOverlap.AddDynamic(this, &A + 本文件的名字 :: 函数名);

GetOverlappingComponents(获得重叠组件)

**测试：显示输出**

UE\_LOG(LogTemp, Log, TEXT("要输出的内容"));显示到输出日志里

SetHiddenInGame(false); //关闭在播放中隐藏（边框线）

//画出调试球体头文件 #include "DrawDebugHelpers.h"

DrawDebugSphere(GetWorld(), 位置, 半径, 分段数, FColor::Red, false（？？？？）, 10.0f);

**数组的声明和视角的切换**

TArray<AActor\*> ReturnedActors;

//TArray 声明一个数组 <类型 \*> 数组名，创建一个指针类型的数组

UGameplayStatics::GetAllActorsOfClass(this, SpectatingViewpointClass, ReturnedActors);

//GetAllActorsOfClass 获取一种类型 SpectatingViewpointClass类型 ReturnedActors数组

//切换视角

SetViewTargetWithBlend(数组中的某一个, 0.5f（时间）, EViewTargetBlendFunction::VTBlend\_Cubic（切换的方式）);

**枚举**

//枚举的定义//标记为 UENUM ，设为 BlueprintType(蓝图类型)

UENUM(BlueprintType)

enum class EAIState : uint8 {

//三种形式（自定义）例如：

Idle,（闲散）

Suspicious,（疑惑）

Alerted（警戒）

};

使用：

.h  
EAIState GuardState;

void 函数名(EAIState NewState);

.cpp(在函数声明一个，枚举变量，在需要的时候调用函数选择自己设置的状态)

void AFPSAIGuard::SetGuardState(EAIState NewState) {}

SetGuardState(EAIState::Alerted);

**团体枪战类游戏总结**

名字 + Component（为什么什么组件）

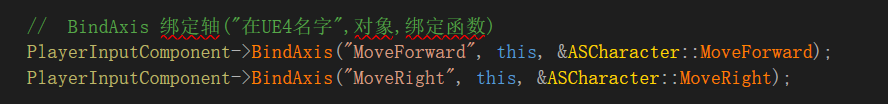
**人物动作**

ADWS人物的简单移动

// BindAxis 绑定轴("在UE4名字",对象,绑定函数)

* // PlayerInputComponent 玩家输入组件(项目设置->输入)

PlayerInputComponent->BindAxis("UE4中的", this, &A类名::函数名);



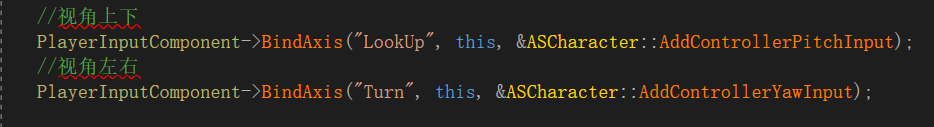
视角的转动

//视角上下 AddControllerPitchInput（UE4自带函数）

PlayerInputComponent->BindAxis("LookUp", this, &A类名::AddControllerPitchInput);

//视角左右 AddControllerYawInput（UE4自带函数）

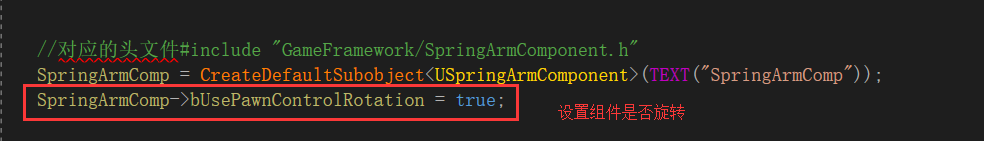
PlayerInputComponent->BindAxis("Turn", this, &A类名::AddControllerYawInput);



添加摄像机和弹簧臂组件

头文件 #include "GameFramework/SpringArmComponent.h"

USpringArmComponent（弹簧臂组件）



// 创建组件 CreateDefaultSubobject（创建默认子组件） bUsePawnControlRotation（根据Pawn控制旋转）让观察者跟随视角而旋转

// UCameraComponent(摄像机组件)

名 = CreateDefaultSubobject<UCameraComponent>(TEXT("名"));

名 -> SetupAttachment(某类);

人物的蹲伏

.h

//开始蹲伏

void BeginCrouch();

//结束蹲伏

void EndCrouch();

.cpp

//开始起跳

void ASCharacter::BeginCrouch()

{

Crouch();(UE4自带蹲伏)

}

//结束蹲伏

void ASCharacter::EndCrouch()

{

UnCrouch();(UE4自带蹲伏)

}

启动蹲伏

//对应的头文件#include "GameFramework/PawnMovementComponent.h"

//GetMovementComponent（获得移动组件）GetNavAgentPropertiesRef(获取导航代理属性参考) bCanCrouch（能否蹲伏）

GetMovementComponent()->GetNavAgentPropertiesRef().bCanCrouch = true;

绑定函数

//绑定动作

PlayerInputComponent->BindAction("Crouch", IE\_Pressed, this, &A类名::BeginCrouch);

//BindAction(绑定动作) Crouch(蹲伏) IE\_Released(松开按键时调用) IE\_Pressed(按键时调用)

PlayerInputComponent->BindAction("Crouch", IE\_Released, this, &A类名::EndCrouch);

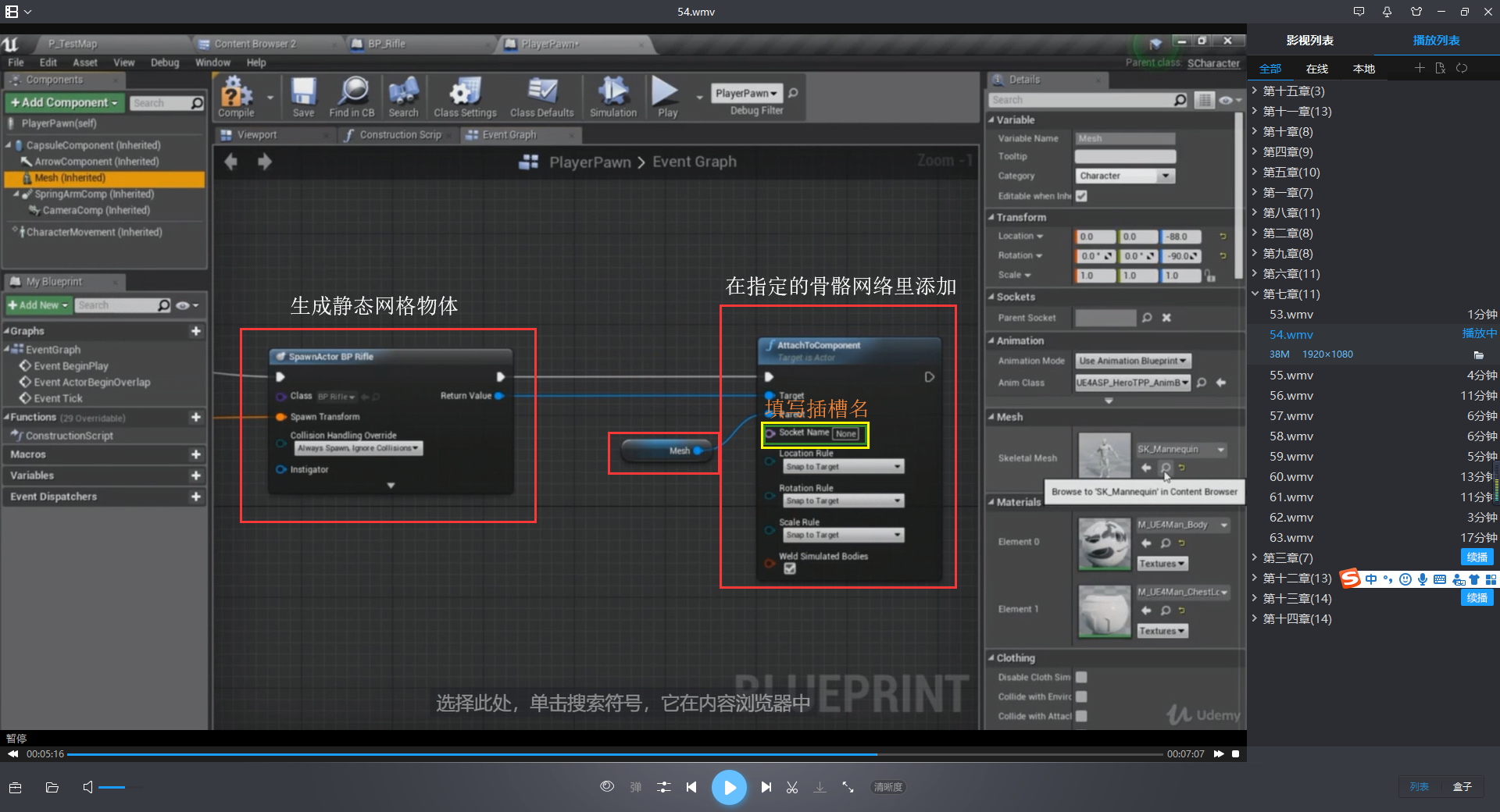
跳跃

//跳 Jump自带函数（一条带那就好）

PlayerInputComponent->BindAction("Jump", IE\_Pressed, this, &ASCharacter::Jump);

**枪**

UskeletalMeshComponent（虚幻骨骼网络体组件）



修改人物视角

.h

//获得任务眼睛的位置函数（UE4自带）View视角

virtual FVector GetPawnViewLocation() const override;

.cpp

//GetPawnViewLocation （获取人的视角位置）

FVector ASCharacter::GetPawnViewLocation() const

{

// 相机组件不为空的话，返回相机位置为视角位置，为空返回任务视角位置

//CameraComp相机组件

if (CameraComp)

{

return CameraComp->GetComponentLocation();

}

return Super::GetPawnViewLocation();

}

// 变量：眼睛看向的位置

FVector EyeLocation;

// 变量：眼睛转动的位置

FRotator EyeRotation;

//获取人物眼睛的位置和方向，执行函数时，会用到我们输入的参数

MyOwner->GetActorEyesViewPoint(EyeLocation, EyeRotation);

/\*

FHitResult Hit;

指定击中结果，用于接收数据

这一结构中，能填入一系列的数据，让我们知道如击中了什么物体，距离多远以及击中方向等有用的信息

射击函数

GetWorld()->LineTraceSingleByChannel(Hit, 起始点, 终点, 碰撞通道（后期自定义碰撞通道）, 碰撞查询参数（自定义 上，可以知道碰撞到了谁）)

\*/

定位骨骼位置添加特效

.h

UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Wrapon")

FName MuzzleSocketName; //附着点变量

.cpp

MuzzleSocketName = "MuzzleSocket";//枪口插槽

// UGameplayStatics::SpawnEmitterAttached(特效变量, 依附者, 附着点（MuzzleSocketName = "MuzzleSocket";//枪口插槽，枪的骨骼蓝图里面的枪口名字）);

UGameplayStatics::SpawnEmitterAttached(MuzzleEffect, MeshComp, MuzzleSocketName);

**倍镜**

.h

//判断鼠标是否按下

bool bWantsToZoom = false;

//相机移动的位置

UPROPERTY(EditDefaultsOnly, Category = "Players")

float ZoomedFOV;

//相机移动的速度

UPROPERTY(EditDefaultsOnly, Category = "Players", meta = (ClampMin = 0.1, ClampMax = 100))

float ZoomInterpSpeed;

//相机默认位置

float DefaultFOV;

void BeginZoom();

void EndZoom();

.cpp

ZoomedFOV = 200.0f;

ZoomInterpSpeed = 20;

float CurrentFOV = bWantsToZoom ? ZoomedFOV : DefaultFOV;

//FInterpTo(插值至)，（当前指的是相机组件的本身视野范围，目标，时间差值，差值速度）

float NewFOV = FMath::FInterpTo(CameraComp->FieldOfView, CurrentFOV, DeltaTime, ZoomInterpSpeed);

//放大镜 SetFieldOfView()视野范围设置

CameraComp->SetFieldOfView(NewFOV);

**创建控制台变量输出**

static int32 DebugWeaponDtawing = 0;声明一个变量

FAutoConsoleVariableRef CAVDebugWeaponDtawing(TEXT("接受变量的字符串 "), 变量, TEXT("描述"), ECVF\_Cheat（不知道）);

抖屏效果

// 创建摄像机抖动的子类

UPROPERTY(EditDefaultsOnly, Category = "Wrapon")

TSubclassOf<UCameraShake> FireCamShake;

// 类型转换获取武器的所有者GetOwner()

APawn\* MyOwner = Cast<APawn>(GetOwner());

if (MyOwner)

{

//开火抖屏功能

//CameraAnimation(摄像机动画)CameraShake(摄像机抖动)

APlayerController\* PC = Cast<APlayerController>(MyOwner->GetController());//强制转换为游戏控制器

if (PC)

{

// ClientPlayCameraShake 客户端相机抖动函数

PC->ClientPlayCameraShake(FireCamShake);

}

}

// 第一次延迟=上一次开火时间+射击时间间隔-减去当前游戏时间，获得游戏世界中的时间秒数

float FirstDelay = FMath::Max(LastFaireTime + TimeBetweenShots - GetWorld()->TimeSeconds, 0.0f);

// GetWorldTimerManager().SetTimer(定时处理程序, this, &ASWeapon::Fire, 频率, 循环(true/false), 第一次延迟);

GetWorldTimerManager().SetTimer(TimerHandle\_TimeBetweenShots, this, &ASWeapon::Fire, TimeBetweenShots, true, FirstDelay);