代理与接口

虚幻四高级程序开发工程师

01

代理

前言

代理可以帮助我们解决一对一或是一对多的任务分配工作。主要可以帮助我们解决通知问题。我们可以通过代理完成调用某一个对象的一个函数，而不直接持有该对象的任何指针。

代理就是为你跑腿送信的，你可以不用关心给送信的目标人具体是谁，只要按照约定好的信件格式进行送信即可

更简单理解，想去调用某个函数，但并不是直接去调用，而是通过另一个入口去调用（代理）

单播代理 只能进行通知一个人

多播代理 可以进行多人通知

动态代理 可以被序列化（这体现在于蓝图进行交互，C++中可以将通知事件进行蓝图广播）

02

单播代理

单播代理

单播代理**只能绑定一个通知，无法进行多个通知绑定。**单播主要将通知动作传递给函数，我们可以绑定到成员函数，或是普通函数上。函数可以具有返回类型

注意：单播对象禁止使用UPROPERTY宏进行标记（单播无法被序列化）

语法（其中一种）

```
DECLARE_DELEGATE(GMDelegateOne)//构建单播类型
GMDelegateOne GmDel;//声明单播对象
```

函数签名	声明宏
void Function()	DECLARE_DELEGATE(DelegateName)
void Function(<Param1>)	DECLARE_DELEGATE_OneParam(DelegateName, Param1Type)
void Function(<Param1>, <Param2>)	DECLARE_DELEGATE_TwoParams(DelegateName, Param1Type, Param2Type)
void Function(<Param1>, <Param2>, ...)	DECLARE_DELEGATE_<Num>Params(DelegateName, Param1Type, Param2Type, ...)
<RetVal> Function()	DECLARE_DELEGATE_RetVal(RetValType, DelegateName)
<RetVal> Function(<Param1>)	DECLARE_DELEGATE_RetVal_OneParam(RetValType, DelegateName, Param1Type)
<RetVal> Function(<Param1>, <Param2>)	DECLARE_DELEGATE_RetVal_TwoParams(RetValType, DelegateName, Param1Type, Param2Type)
<RetVal> Function(<Param1>, <Param2>, ...)	DECLARE_DELEGATE_RetVal_<Num>Params(RetValType, DelegateName, Param1Type, Param2Type, ...)

操作步骤

1. 通过宏进行声明代理对象类型（根据回调函数选择不同的宏）
2. 使用代理类型进行构建代理对象
3. 绑定回调对象，和操作函数
4. 执行代理对象回调

```
DECLARE_DELEGATE(CallTest);
DECLARE_DELEGATE_RetVal_OneParam(int32, CallTestRe, int32);
```

```
cb.BindUObject(am, &AMyActor::CallBackTest);

cb.ExecuteIfBound();

cbr.BindUObject(am, &AMyActor::CallBackRe);

int32 num = 0;
num = cbr.Execute(100);
```

常用绑定函数

BindUObject 绑定UObject类型对象成员函数的代理

BindSP 绑定基于共享引用的成员函数代理

BindRaw 绑定原始自定义对象成员函数的代理，操作调用需要注意执行需要检查IsBound

BindStatic 绑定全局函数成为代理

BindLambda 绑定到一个匿名函数上

BindUFunction 通过函数文本名称将一个对象的成员函数绑定到代理（成员函数需要使用UFUNCTION标记，必须是U类）

绑定需要注意，绑定中传递的对象类型必须和函数指针所属类的类型相同否则绑定会报错

```
GmDel.BindUObject(act, &AMyActor::Say);
```

调用执行

为了保证调用的安全性，执行Execute函数之前需要检查是否存在有效绑定使用函数IsBound

Execute 调用代理通知，不安全，需要注意

ExecuteIfBound 调用代理通知，安全，但是有返回类型的回调函数无法使用此函数执行回调

IsBound 检查当前是否存在有效代理绑定

GmDel.ExecuteIfBound();

解除绑定

UnBind()函数可以解除绑定，由于单播只绑定了一个通知的事件，所以解绑只有一种操作方式。

03

多播代理

多播代理

与单播不同的是，多播可以将动作传递到更多的事件上，可以进行多次绑定。多播本身在绑定时，不约束类型。但是**不建议将一个事件，多次绑定到同一个多播对象上。**

注意：多播无法绑定到具有返回信息的事件上，也就是多播构建的类型均无法绑定有返回类型的函数。

操作步骤：

1、构建多播对象类型

2、声明多播对象

3、绑定多播事件

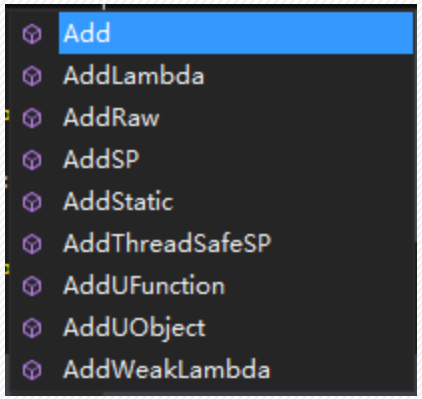
4、进行广播

多播代理

构建宏（其中一种）

```
#define DECLARE_MULTICAST_DELEGATE( DelegateName ) FUNC_DECLARE_MULTICAST_DELEGATE( DelegateName, void )
```

绑定函数



注意：AddUFunction绑定的函数必须加UFUNCTION标记

其中Add函数需要构建一个FInputActionHandlerSignature（本质是代理类型）类型数据，使用如同单播。将对象添加到多播中即可。

广播

调用函数Broadcast，但是调用不保证执行顺序的正确性

多播解除绑定

解除多播绑定提供了三中解除方式，借助代理句柄解除某一个绑定，通过给定对象指针解除对象所有绑定，解除所有绑定

1、借助FDelegateHandle句柄解除绑定（Remove函数）

当多播进行绑定添加时，会返回FDelegateHandle对象。如希望解除绑定是，调用多播的Remove函数，填入对象即可解绑。

2、借助对象指针解除绑定（RemoveAll）

当一个对象身上存在多个绑定到一个多播时，可以通过调用多播对象的RemoveAll函数（填入对象指针），移除该对象的所有绑定

3、移除所有绑定（Clear函数）

通过调用多播对象函数Clear，可以移除所有绑定到多播上的回调通知。

04

动态代理

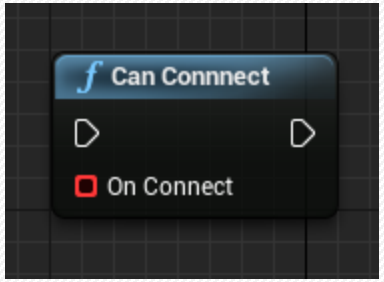
动态代理

允许被序列化的数据结构，这将使得代理可以被数据化提供给蓝图进行使用，达到在CPP中调用代理广播，事件通知到蓝图中。

动态代理和普通代理基本相同，分为单向和多向。动态单播允许绑定有返回值的事件，而动态多播不允许绑定带有返回值的事件（**动态单播单播无法被创建为成员变量，通过标记暴露到蓝图中使用，但是动态单播可以作为函数参数，被序列化，在蓝图中调用函数将会有事件绑定节点**）。创建时动态代理类型时，需要名称使用**F**开头，并且在构建宏后需要加**分号**。动态代理只能绑定到**UFUNCTION**标记的函数上。

UE中的大部分通知事件均使用动态代理（方便蓝图操作），如碰撞通知，UMG中的控件事件

动态单播做为函数参数，暴露到蓝图中，被当作绑定事件节点使用，如下图



构建步骤

1、构建动态代理类型（注意构建宏后需要加分号，代理类型名称首字母使用F开头）

2、构建动态代理对象

3、绑定事件到代理对象

4、代理进行广播

创建一个动态单播代理（其中一个）

DECLARE_DYNAMIC_DELEGATE[_Const, _RetVal, etc.](DelegateName)

创建一个动态的多播代理（其中一个）

DECLARE_DYNAMIC_MULTICAST_DELEGATE[_Const, _RetVal, etc.](DelegateName)

注意：动态代理构建时，如果带有参数，需要将参数的类型和名称均进行指出。

绑定函数

动态单播

BindUFuntion(UserObject, FuncName)

在动态单播调用函数BindUFuntion进行绑定。参数1、U类对象指针 2、函数名称（函数需要使用UFUNCTION标记）

动态多播

Add（FScriptDelegate）

FScriptDelegate类型类似动态单播，操作方式相同。**动态单播类的父类就是FScriptDelegate**

步骤：先构建FScriptDelegate类对象，然后进行绑定，参照截图代码

```
FScriptDelegate Del;

Del.BindUFunction(this, TEXT("FunctionName"));//函数必须加UFUNCTION标记
```

AddUnique(FScriptDelegate)

添加一个唯一的代理事件

绑定函数（宏）

被绑定的函数都必须使用UFUNCTION宏标记

BindDynamic(UserObject, FuncPtr)

在**动态单播**代理上调用BindDynamic()的辅助宏。参数1、U类对象指针 2、成员函数地址（UFUNCTION标记)

AddDynamic(UserObject, FuncPtr)

在**动态多播**代理上调用AddDynamic()的辅助宏。

移除绑定

动态多播移除绑定一共有4个函数可以使用，分别是Remove，RemoveAll，Clear。

1、Remove（FScriptDelegate），需要提供绑定时的FScriptDelegate类型对象

2、RemoveAll，通过U类对象指针，移除给定对象的绑定

3、Clear，移除所有绑定

4、RemoveDynamic(UserObject, FuncName)在动态多播代理上调用RemoveDynamic()的辅助宏。**FuncName传递函数地址。**

动态单播移除绑定只有一个函数UnBind

与单播多播区别

- 动态代理构建类型名称需要用F开头（动态代理实现机制构建了类）
- 当存在参数时，不光需要提供参数类型，还必须提供参数名称
- 动态代理对象类型可以使用UPROPERTY标记，其他代理均无法使用（不加编译可过，调用出错）
- 动态代理绑定对象的函数需要使用UFUNCTION进行描述（因为需要跟随代理被序列化）

05

动态代理用于蓝图

构建

需要注意的时，在构建动态代理提供蓝图使用时（**被当作事件调度器**），需要在代理上增加标记宏UPROPERTY(BlueprintAssignable)

如果添加的多播对象在组件中时，将会被序列化到面板上，直接通过加号进行绑定。

构建宏

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FGMDynMulDele);
```

对象声明

```
UPROPERTY(BlueprintAssignable)
FGMDynMulDele OnGmDynMulDele;
```

在C++中调用

```
if (OnGmDynMulDele.IsBound())
{
        OnGmDynMulDele.Broadcast();
}
```

呈现方式

在蓝图中以调度器方式存在，使用需要绑定

当持有动态多播代理的类被声明为变量时，则可在变量细节面板中查到对应的多播绑定，可以在面板中查看到

06

事件

前言

事件本身和多播代理一样，为了操作的安全性，事件提供了额外的操作限定。即禁止在声明事件对象的外部调用事件传播，清理，检查等函数。通过操作隔离，最大程度的增加了事件的安全性。派生类允许调用事件的广播。

在虚幻C++中事件和多播几乎相同。只是构建方式略不同

构建宏

事件类型构建宏由于需要限定事件对象调用约束关系，需要提供声明所在类型，并且需要在类内部进行声明。事件没有返回值。

OwningType即当前声明事件的类

```
DECLARE_EVENT( OwningType, EventName )
DECLARE_EVENT_OneParam( OwningType, EventName, Param1Type )
DECLARE_EVENT_TwoParams( OwningType, EventName, Param1Type, Param2Type )
DECLARE_EVENT_<Num>Params( OwningType, EventName, Param1Type, Param2Type, ...)
```

一般构架语法如下：将事件声明在类的内部，通过函数构建返回事件对象操作

```
public:
    DECLARE_EVENT(AMyActor, FChangeEvent)
    FChangeEvent& OnChanged() { return ChangeEvent; }
private:
    FChangeEvent ChangeEvent;
```

绑定函数与广播

函数	描述
Add()	添加函数代理到此多播代理的调用列表。
AddStatic()	添加一个原始的C++指针全局函数代理。
AddRaw()	添加一个原始的C++指针代理。原始指针不使用任何引用，所以如果从代理的底层删除了该对象，那么调用它可能是不安全的。因此，当调用Execute()时一定要小心!
AddSP()	添加一个基于共享指针（快速，非线程安全）的成员函数代理。共享指针代理保持到您的对象的弱引用。
AddUObject()	添加一个基于UObject的成员函数代理。UObject 代理保持到您的对象的弱引用。
Remove()	将函数从这个多播代理的调用列表中移除(性能为O(N))。请注意代理的顺序可能不会被保留！
RemoveAll()	将所有函数从与特定UserObject绑定的多播代理的调用列表中移除。请注意代理的顺序可能不会被保留！

调用函数Broadcast，但是调用不保证执行顺序的正确性。事件广播无需检查是否存在有效的绑定。**事件广播应发生在声明事件类型的类内部。**

07

接口

前言

词义广泛，用来陈述功能，选项，与其他程序结构进行沟通的方式。接口抽象出了交互结构，提供了两个未知逻辑交互的便捷性。对于编程中，如何更好的设计低耦合程序起到了至关重要的作用。设计者可以在互不关心的情况下，进行友好的程序设计，并且通过接口来完成设计的整合交互。

虚幻引擎中，加入了接口设计，从一定程度上“去掉了”多继承。接口可以帮助我们解决在不同类型的类之间却有相同行为的特性。接口的设计增加了代码编写的便捷性。

例如在设计射击类游戏时，我们需要子弹与场景中的物体进行交互，场景中的桌椅板凳，角色，怪物（都是独立的对象）都希望受到子弹的攻击伤害。那么子弹在打到目标后要逐一排查，审查目标是否属于上述的对象！这很麻烦！但是我们可以通过接口，增加上述目标具有受伤的能力。当子弹打到目标时，我只需要检查目标是否继承受伤的接口，如果有，则调用接口函数即可！

构建接口类

接口类可以直接在虚幻编辑器中选择继承，然后完成构建

7-1 仅在C++中使用接口

仅在C++中使用

当在构建的接口**仅需在C++中使用时**，则可以将接口类中的函数直接定义为**虚函数**。在需要使用接口的类中完成继承，并重写虚函数即可（尽量不要在接口中构建属性）。

如果编写的接口函数是纯虚函数，则无需完成定义，如果不是，则需要在源文件中添加定义函数。

接口调用，需要将对象类型进行转换为接口类（I类）然后调用接口函数即可。

以下是接口类

```
UINTERFACE(MinimalAPI)
class UDamageInterface : public UInterface
{
    GENERATED_BODY()
};

/**
 *
 */
class TANK90CPP_API IDamageInterface
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be inherited to implement this interface.
public:

    virtual void BulletDamage(class ABulletActor*, class UPrimitiveComponent*);

};
```

仅在C++中使用

继承接口的类

```
UCLASS()
class TANK90CPP_API AMapDirectorActor : public AActor, public IDamageInterface
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AMapDirectorActor();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    void SpawnWallComponent(int32 Code, EWallType Type);

    virtual bool BulletDamage(class ABulletActor* Bullet, class UPrimitiveComponent* Comp) override;
```

调用接口函数

```
35  void ABulletActor::OnComponentBeginOverlapEvent(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBod
36  {
37      IDamageInterface* Di = Cast<IDamageInterface>(OtherActor);
38
39      if (Di)
40      {
41          Di->BulletDamage(this, OtherComp);
42      }
43
```

7-2 C++中构建接口与蓝图一起使用

编写接口

如果希望接口类在C++中定义，在**蓝图中使用**，则需要将接口函数，进行标记，**此种方法不仅可以提供蓝图使用，也可以供C++中使用。**

如果希望编写的接口函数能够在蓝图中**被调用**，**则需要增加标记Blueprint Callable。**

接口类声明（**此种声明方式不需要在源文件中进行定义**）

```
9     // This class does not need to be modified.
10    UINTERFACE(MinimalAPI)
11    class UDamageInterface : public UInterface
12    {
13        GENERATED_BODY()
14    };
15
16    /**
17     *
18     */
19    class UECPP_API IDamageInterface
20    {
21        GENERATED_BODY()
22
23        // Add interface functions to this class. This is the class that will be inherited to implement this interface.
24    public:
25        UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
26        void DamagePointer(AActor* Instigator);
27        UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
28        void DamageRadius(AActor* Instigator);
29    };
```

在C++中实现接口

如果在C++中希望获得接口能力，则需要继承接口。需要注意的是，必须继承I开头的接口名称，并且继承修饰为public。不要尝试重写接口中的函数！

如果接口中的函数使用**BlueprintNativeEvent**说明，则在继承类（C++）中可以编写同名函数，并用后缀“_Implementation”进行标记。

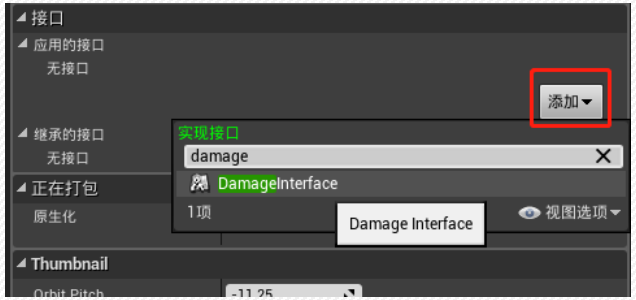
如果接口中的函数使用**BlueprintImplementableEvent**说明，**则无法在C++的继承类中实现接口函数**

继承接口的C++类（**注意，不要省略override，函数的返回值，参数列表需要和接口的一致**）

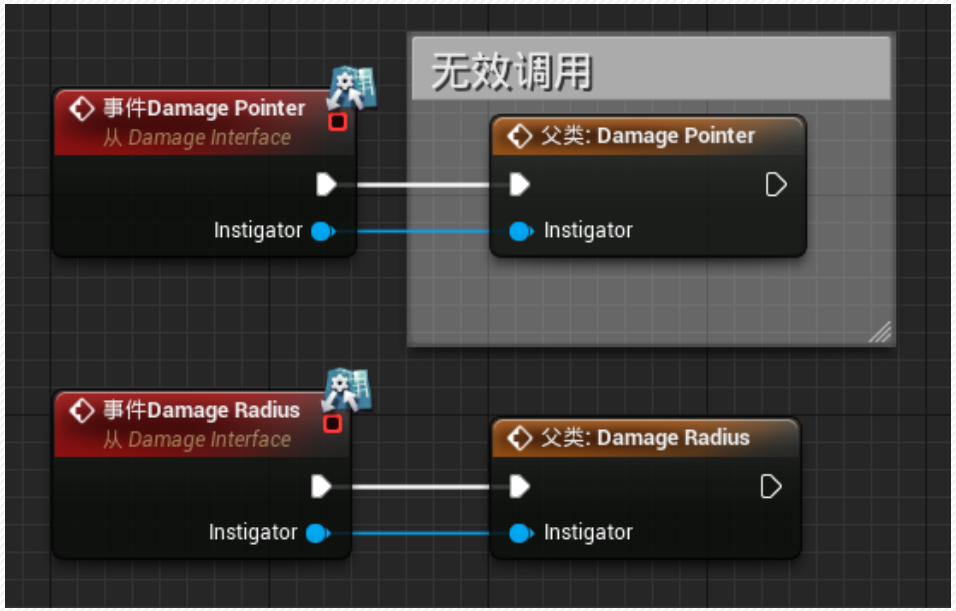
```
UCLASS()
class UECPP_API AInterfaceActor : public AActor, public IDamageInterface
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AInterfaceActor();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
    //当接口类中带有BlueprintNativeEvent标记的函数，则可以使用下面函数格式进行“重写”
    virtual void DamageRadius_Implementation(AActor* Instigator) override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

在蓝图中实现接口

如果在蓝图中希望继承C++中的接口，则需要在类设置中找到接口，手动添加。



当继承的C++类中有实现接口类，并且重写了接口函数（BlueprintNativeEvent 标记的函数），则在蓝图中可以通过调用父节点，主动调用父类重写函数中的逻辑。（左图父类没有继承接口，右图父类（父类在C++中）有继承接口）



蓝图继承接口注意

如果接口在蓝图中被继承，则需要注意下面的问题

1. 如果函数没有返回类型，则在蓝图中当作事件Event使用
2. 如果函数存在返回类型或是存在传递引用参数，则在蓝图中当作函数使用
3. 接口函数说明符使用BlueprintNativeEvent或是BlueprintImplementableEvent标记都可以在蓝图中找到
4. 实现接口函数方法与UFUNCTION中讲解标记BlueprintImplementableEvent和BlueprintNativeEvent规则一致。
5. **如果不希望接口类被蓝图继承，则只需要在标记中添加meta=(CannotImplementInterfaceInBlueprint)即可。**

调用操作（C++中）

调用函数，持有继承接口对象指针，第一步先转换到I类指针，调用Execute_接口函数名，参数第一位需要传递原对象指针，后面直接按照原函数参数填入即可。

如有返回值，Execute_函数也将返回数据。

注意：当接口被纯蓝图类继承后，则接口函数无法通过下面的方法在C++中实现调用。如果蓝图继承C++类，在C++中有继承接口类，则可以使用下面方法调用接口函数。

```
6   void AUECPPGameModeBase::AttackToActor(AActor* Target)
7   {
8       //检查对象是否继承接口
9       IDamageInterface* pI = Cast<IDamageInterface>(Target);
10      if (pI)
11      {
12          //执行调用 参数1是指实现接口的类实例， 参数2接口函数参数
13          pI->Execute_DamagePointer(Target, this);
14      }
```

调用操作（蓝图中）

在蓝图中调用接口函数时，需要注意接口函数需要标记BlueprintCallable，否则无法找到函数入口。

当调用函数的对象类型是**接口的对象类型**时，则可以直接调用。**注意：C++中的接口需要标记BlueprintType才可以通过Cast转换为接口类型对象（需要关掉情景关联搜索）。**

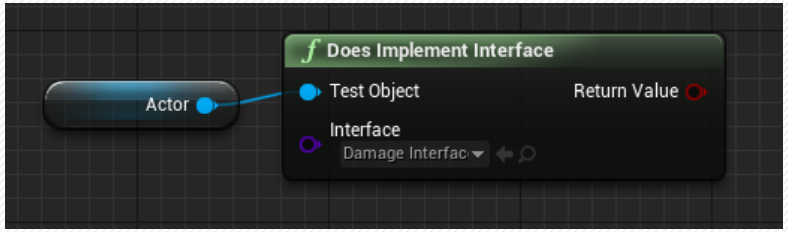
调用操作（蓝图中）

当调用函数的对象类型，不是继承接口的类型时，则需要通过**消息**进行调用。当对象继承接口则成功，否则失败。

检查对象是否继承接口

当获取到对象后，我们需要检查对象是否继承接口，则需要通过以下的方法

蓝图中



在C++中

```
//检查对象是否继承了接口类
Target->GetClass()->ImplementsInterface(UDamageInterface::StaticClass());
```

总结

操作注意：

1. 接口函数需要定在I开头的类中，不要修改访问域public关键字，声明需要使用宏标记BlueprintNativeEvent或BlueprintImplementableEvent（在C++继承类中无需编写声明（直接声明为纯虚函数即可），两个标记主要是给蓝图中使用）
2. 如果接口函数希望在蓝图中被调用，需要再追加宏**BlueprintCallable**
3. 如需继承接口，继承I类，继承关系public
4. 接口中的函数（被UFUNCTION标记）禁止在接口的源文件中定义
5. 当接口中函数使用BlueprintNativeEvent标记时，在继承类中实现接口函数，并添加后缀_Implementation，需要注意，函数前加入虚函数关键字virtual，函数结尾加override关键字（可以不添加，但是建议加上，加强函数编写正确性检查），在CPP文件中实现逻辑
6. 在C++中调用函数，持有继承接口对象指针，第一步先转换到I类指针，调用Execute_接口函数名，参数第一位需要传递原对象指针，后面直接按照原函数参数填入即可
7. 检查某一个类是否实现了对应接口可以使用如下语法进行检查

   obj->GetClass()->ImplementsInterface(U类型：：StaticClass（））；

   act->GetClass()->ImplementsInterface(UMyInterface::StaticClass());

   act是对象指指针

总结

接口优点

1. 具备多态特性，接口衍生类支持里氏转换原则
2. 接口可以使得整个继承系统更加的干净单一
3. 接口可以规范类的具体行为
4. 接口可以隔离开发中的开发耦合，我们只需要针对接口去编码，无需关心具体行为
5. 接口继承可以使得继承关系中出现真正的操作父类

接口缺点

1. 丢失了C++中的广泛继承特性
2. 接口拘束了类型的属性拓展，无法进行更详细的内容定义
3. 继承关系中容易让人混淆，接口本身不具备真正的继承特性

感谢观看

虚 幻 四 高 级 程 序 开 发