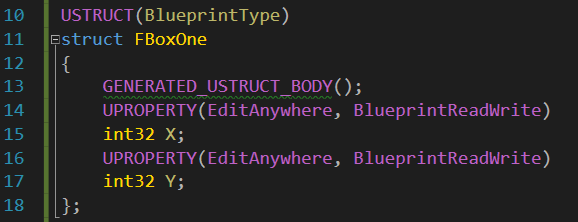
# 结构体

##### UECPP中的结构体

和普通的结构体构建相同

##### 蓝图通信的结构体

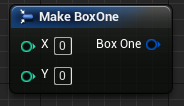
注意：**结构体名称必须以F开头**，加入标记宏，模式如下



注意宏 GENERATED\_USTRUCT\_BODY()后面加分号

* 如果希望结构体中的内容在蓝图中可被操作，需要加入属性标记宏UPROPERTY
* 如果希望在蓝图中属性可以被读取和修改就需要加入标记指令 BluprintReadWrite

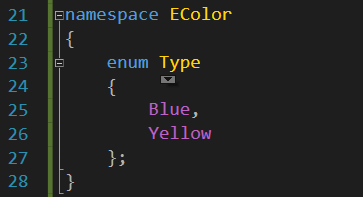
这样构建结构体的时候才可以出现编辑输入针脚，如图



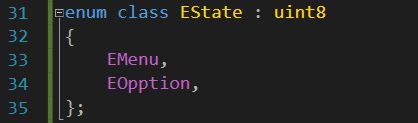
# 枚举的构建

UEC++中的枚举构建分两种

1. 空间构建枚举

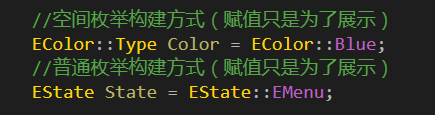


1. 普通构建枚举



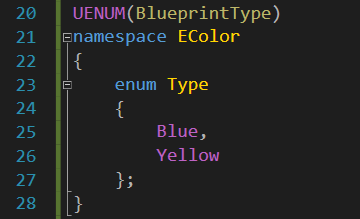
注意：继承自无符号整形，如果有符号将出现问题，无法暴露到蓝图

1. 使用



# 构建枚举和蓝图进行交互

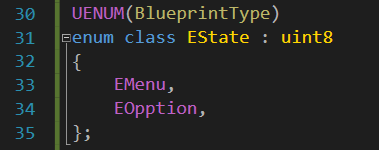
1. 空间枚举暴露到蓝图编辑



声明枚举变量



1. 普通构建方式

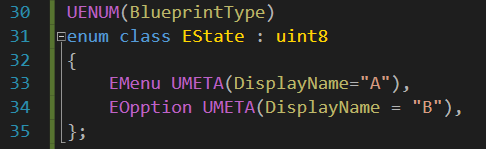


声明枚举



##### UMETA宏

可以帮助进行枚举项调整在蓝图中的显示名称，方便在蓝图中寻找



**注意：空间枚举不能使用别名**

# UPROPERTY宏

在虚幻中，所有的U类衍生类的指针类型，在声明成为成员属性时，都需要使用宏UPROPERTY标记，以增加指针引用计数的能力

# Cast函数

安全类型转换函数，将指针进行目标类型转换，如果转换成成功，则返回转换后的指针，如果转换失败，则返回null。

# 接口

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。

### UE中的接口操作

UE语言具备C++多继承特点，但是由于UE对C++做了深度定制，强制了继承自Uobject的类只能继承一个具有完整属性行为的父类，其他继承关系以接口方式来实现，接口只定义了行为的框架，不具有具体的逻辑行为，需要继承自接口的类亲自实现接口逻辑。

可以直接通过在UE中进行构建

**·接口如果想在蓝图中使用可以在UCLASS中加入Blueprintable或是不想蓝图使用NotBlueprinteable**

注意：

如果函数没有返回类型，则在蓝图中当作事件Event使用

如果函数存在返回类型，则在蓝图中当作函数使用（需要在函数的overlap中寻找）

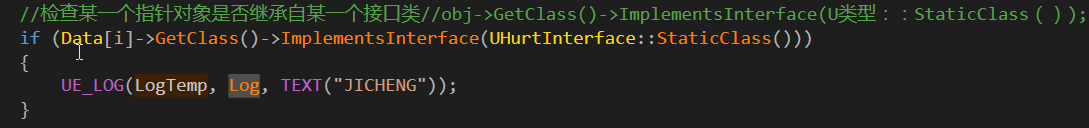
操作注意：

1. 在编辑器中通过构建C++类，选择构建接口类
2. 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UFUNCTION(BlueprintNativeEvent)
3. 接口函数禁止重载
4. 如需继承接口，继承I类，继承关系public
5. 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，但是建议添加，可以加强函数编写检查的准确性），在CPP文件中实现逻辑
6. 所有接口函数均需要进行定义
7. 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
8. 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



**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，继承自接口的类**必须实现接口内的所有行为**
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

# 函数指针

通过特殊定义可以声明函数指针，用来创建直接可以操作函数的指针对象，这样我们就可以将函数作为参数进行传递

函数指针不能指向 void\*（无类型指针）

##### 全局函数指针

格式

Void (\*p1)(int a, int b);

赋值

Void add(int, int)

P1 = add;//成员函数不能使用此方法，只适用于全局函数

或

P1 = &add;

声明了一个函数指针变量p1，并且没有指向任何函数

函数指针的赋值，必须要保证函数原型一致，否则赋值失败

调用方式：

P1(1, 1);

(\*p1) (1, 1);

##### 成员函数指针

静态成员函数指针和普通函数指针相同

非静态成员函数指针由于操作的特殊性（成员函数有归属关系），所以调用过程中需要提供调用对象数据

格式

返回类型 （从属类::\*指针名）（参数列表）;

赋值

类A中有函数 void Say（）;

声明成员函数指针

void (A：：\*pF)();

pF = &A：：Say

调用

A a；

(a.\*pF)();

##### Typedef函数指针

格式

Typedef 返回类型 （\*助记符名称）（参数列表）

使用

助记符名称 变量名

typedef int (A::\*pMax)(int, int);

pMax pM = &A::Max;

# 代理

为了解决回调通知中，对象类型不同，无法使用同一容器进行存储的问题。主要达到目的调用某个对象的函数，可以不知道对象类型也可以完成操作。

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）

### 单播代理

构建分两部：

1. 构建单播代理类型
2. 通过单播代理类型构建成员代理对象

通过宏进行构建，单播代理只能绑定一个通知对象，无法进行多个对象通知

宏语法

构建宏分为两种

一种有返回类型的，一种没有返回类型的

语法

DECLARE\_DELEGATE(GMDelegateOne)//构建宏

GMDelegateOne GmDel;//声明代理

###### 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

UnBind 解除绑定代理关系

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

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();

###### 构建步骤

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



