

C++宏清单

章节：

Specifier

UCLASS

UINTERFACE

USTRUCT

UENUM

UFUNCTION

UPARAM

UPROPERTY

Meta

Meta

Flags

EClassFlags

EStructFlags

EEEnumFlags

EFunctionFlags

EPropertyFlags

EObjectMark

UCLASS(标识符)

UHT

Name	引擎模块	功能描述	常用程度
NoExport	UHT	指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。	💀
Intrinsic	UHT	指定UHT完全不为此类生成代码，需要自己手写。	💀
Interface	UHT	标识这个Class是个Interface。	💀
UCLASS()	UHT	留空的默认行为是不能在蓝图中被继承，不能在蓝图中定义变量，但拥有反射的功能。	★★★★★

Name	引擎模块	功能描述	常用程度
不写UCLASS()	UHT	只是作为一个普通的C++对象，没有反射功能。	★
CustomThunkTemplates	UHT	Specifies the struct that contains the CustomThunk implementations	💀
CustomConstructor	UHT	阻止构造函数声明自动生成。	💀
CustomFieldNotify	UHT	阻止UHT为该类生成FieldNotify的相关代码。	💀

Blueprint

Name	引擎模块	功能描述	常用程度
Blueprintable	Blueprint	可以在蓝图里被继承，隐含的作用也可当变量类型	★★★★★
NotBlueprintable	Blueprint	不可在蓝图里继承，隐含作用也不可当作变量	★★★★
BlueprintType	Blueprint	可当做变量类型	★★★★★
NotBlueprintType	Blueprint	不可当做变量类型	★★★★
Abstract	Blueprint	指定此类为抽象基类。可被继承，但不可生成对象。	★★★★★
Const	Blueprint	表示本类的内部属性不可在蓝图中被修改，只读不可写。	★★★
ShowFunctions	Blueprint	在子类的函数覆盖列表里重新打开某些函数。	★★
HideFunctions	Blueprint	在子类的函数覆盖列表里隐藏掉某些函数。	★★
SparseClassDataType	Blueprint	让Actor的一些重复不变的数据存放在一个共同的结构里，以达到减少内存使用量的目的	★★★
NeedsDeferredDependencyLoading	Blueprint		💀

DllExport

Name	引擎模块	功能描述	常用程度
MinimalAPI	DllExport	不dll导出该类的函数，只导出类型信息当作变量。	★★★

Category

Name	引擎模块	功能描述	常用程度
ClassGroup	Category	指定组件在Actor的AddComponent面板里的分组，以及在蓝图右键菜单中的分组。	★★★
ShowCategories	Category	在类的ClassDefaults属性面板里显示某些Category的属性。	★★★
HideCategories	Category	在类的ClassDefaults属性面板里隐藏某些Category的属性。	★★★★
CollapseCategories	Category	在类的属性面板里隐藏所有带Category的属性，但是只对带有多个嵌套Category的属性才起作用。	★★
DontCollapseCategories	Category	使继承自基类的CollapseCategories说明符无效。	★★
AutoExpandCategories	Category	指定此类的对象在细节面板中应该自动展开的Category。	★
AutoCollapseCategories	Category	AutoCollapseCategories说明符使父类上的 AutoExpandCategories 说明符的列出类别的效果无效。	★
DontAutoCollapseCategories	Category	使列出的类别的继承自父类的AutoCollapseCategories说明符无效。	★
PrioritizeCategories	Category	把指定的属性目录优先显示在细节面板的前面。	★★★
ComponentWrapperClass	Category	指定该类为一个简单的封装类，忽略掉子类的Category相关设置。	★★
AdvancedClassDisplay	Category	把该类下的所有属性都默认显示在高级目录下	★★★★

TypePicker

Name	引擎模块	功能描述	常用程度
HideDropDown	TypePicker	在类选择器中隐藏此类	★★

Development

Name	引擎模块	功能描述	常用程度
Deprecated	Development	标明该类已经弃用。	★★★
Experimental	Development	标明该类是试验性版本，当前没有文档描述，之后有可能废弃掉。	★★★
EarlyAccessPreview	Development	标明该类是早期预览版，比试验版要更完善一些，但还是没到产品级。	★★★

Instance

Name	引擎模块	功能描述	常用程度
Within	Instance	指定对象创建的时候必须依赖于OuterClassName的对象作为Outer。	★★★
DefaultToInstanced	Instance	指定该类的所有实例属性都默认是UPROPERTY(instanced)，即都默认创建新的实例，而不是对对象的引用。	★★★★
EditInlineNew	Instance	指定该类的对象可以在属性细节面板里直接内联创建，要和属性的Instanced配合。	★★★★★
NotEditInlineNew	Instance	不能通过EditInline按钮创建	★

Scene

Name	引擎模块	功能描述	常用程度
NotPlaceable	Scene	标明该Actor不可被放置在关卡里	★★★
Placeable	Scene	标明该Actor可以放置在关卡里。	★★★
ConversionRoot	Scene	在场景编辑器里允许Actor在自身以及子类之间做转换	★

Config

Name	引擎模块	功能描述	常用程度
Config	Config	指定配置文件的名字，把该对象的值保存到ini配置文件中。	★★★★★

Name	引擎模块	功能描述	常用程度
PerObjectConfig	Config	在已经有config配置文件名字的情况下，指定应该按每个对象实例来存储值，而不是一个类一个存储值。	★★★★★
ConfigDoNotCheckDefaults	Config	指定在保存配置值的时候忽略上一级的配置值的一致性检查。	★
DefaultConfig	Config	指定保存到的配置文件层级是 Project/Config/DefaultXXX.ini。	★★★
GlobalUserConfig	Config	指定保存到的配置文件层级是全局用户设置 Engine/Config/UserXXX.ini。	★★★
ProjectUserConfig	Config	指定保存到的配置文件层级是项目用户设置 Project/Config/UserXXX.ini。	★★★
EditorConfig	Config	用来在编辑器状态下保存信息。	★

Serialization

Name	引擎模块	功能描述	常用程度
Transient	Serialization	指定该类的所有对象都略过序列化。	★★★
NonTransient	Serialization	使继承自基类的Transient说明符无效。	★★★
Optional	Serialization	标记该类的对象是可选的，在Cooking的时候可以选择是否要忽略保存它们。	★
MatchedSerializers	Serialization	指定类支持文本结构序列化	💀

UINTERFACE(标识符)

DllExport

Name	引擎模块	功能描述	常用程度
MinimalAPI	DllExport	指定该UInterface对象不导出到别的模块	★

Blueprint

Name	引擎模块	功能描述	常用程度
Blueprintable	Blueprint	可以在蓝图中实现	★★★★★
NotBlueprintable	Blueprint	指定不可以再蓝图中实现	★★★
ConversionRoot	Blueprint	Sets IsConversionRoot metadata flag for this interface.	💀

USTRUCT(标识符)

UHT

Name	引擎模块	功能描述	常用程度
NoExport	UHT	指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。	★
Atomic	UHT	指定该结构在序列化的时候总是一整个输出全部属性，而不是只输出改变的属性。	★
IsAlwaysAccessible	UHT	指定UHT在生成文件的时候总是可以访问到改结构的声明，否则要在gen.cpp里生成镜像结构定义	💀
HasDefaults	UHT	指定该结构的字段拥有默认值。这样如果本结构作为函数参数或返回值时候，函数则可以为其提供默认值。	💀
HasNoOpConstructor	UHT	指定该结构拥有ForceInit的构造函数，这样在作为BP function返回值的时候，可以调用来初始化	💀
IsCoreType	UHT	指定该结构是核心类，UHT在用它的时候不需要前向声明。	💀

Blueprint

Name	引擎模块	功能描述	常用程度
BlueprintType	Blueprint	允许这个结构在蓝图中声明变量	★★★★★
BlueprintInternalUseOnly	Blueprint	不可定义新BP变量，但可作为别的类的成员变量暴露和变量传递	★★
BlueprintInternalUseOnlyHierarchical	Blueprint	在BlueprintInternalUseOnly的基础上，增加了子类也不能定义新BP变量的限制。	★

Serialization

Name	引擎模块	功能描述	常用程度
immutable	Serialization	Immutable is only legal in Object.h and is being phased out, do not use on new structs!	💀

UENUM(标识符)

Trait

Name	引擎模块	功能描述	常用程度
Flags	Trait	把该枚举的值作为一个标志来拼接字符串输出。	★★★★★

Blueprint

Name	引擎模块	功能描述	常用程度
BlueprintType	Blueprint	可以作为蓝图变量	★★★★★

UFUNCTION(标识符)

Editor

Name	引擎模块	功能描述	常用程度
Category	Editor	在蓝图的右键菜单中为该函数指定类别分组，可以嵌套多级	★★★★★
CallInEditor	Editor	可以在属性细节面板上作为一个按钮来调用该函数。	★★★★★

Blueprint

Name	引擎模块	功能描述	常用程度
BlueprintCallable	Blueprint	暴露到蓝图中可被调用	★★★★★
BlueprintPure	Blueprint	指定作为一个纯函数，一般用于Get函数用来返回值。	★★★★★
BlueprintImplementableEvent	Blueprint	指定一个函数调用点，可以在蓝图中重载实现。	★★★★★
BlueprintNativeEvent	Blueprint	可以在蓝图总覆盖实现，但是也在C++中提供一个默认实现。	★★★★★
BlueprintGetter	Blueprint	指定该函数作为属性的自定义Get函数。	★★
BlueprintSetter	Blueprint	指定该函数作为属性的自定义Set函数。	★★

Behavior

Name	引擎模块	功能描述	常用程度
Exec	Behavior	在特定类里注册一个函数为作为控制台命令，允许接受参数。	★★★
SealedEvent	Behavior	无法在子类中覆盖此函数。SealedEvent关键词只能用于事件。对于非事件函数，请将它们声明为static或final，以密封它们。	💀

Network

Name	引擎模块	功能描述	常用程度
BlueprintAuthorityOnly	Network	这个函数只能在拥有网络权限的端上运行。	★★★
BlueprintCosmetic	Network	此函数为修饰性的，无法在DS上运行。	★★★
Client	Network	在Client-owned的Actor上 (PlayerController或Pawn) 执行一个RPC函数，只运行在客户端上。对应的实现函数会添加_Implementation后缀。	★★★★★
Server	Network	在Client-owned的Actor上 (PlayerController或Pawn) 执行一个RPC函数，只运行在服务器上。对应的实现函数会添加_Implementation后缀	★★★★★
NetMulticast	Network	定义一个多播RPC函数在服务器和客户端上都执行。对应的实现函数会添加_Implementation后缀。	★★★★★
Reliable	Network	指定一个RPC函数为“可靠的”，当遇见网络错误时会重发以保证到达。一般用在逻辑关键的函数上。	★★★★★
Unreliable	Network	指定一个RPC函数为“不可靠的”，当遇见网络错误时就会被丢弃。一般用在传播效果表现的函数上，就算漏掉也没有关系。	★★★★★
WithValidation	Network	指定一个RPC函数在执行前需要验证，只有验证通过才可以执行。	★★★★★
ServiceRequest	Network	此函数为RPC（远程过程调用）服务请求。rpc服务请求	💀
ServiceResponse	Network	此函数为RPC服务响应。rpc服务回复	💀

UHT

Name	引擎模块	功能描述	常用程度
BlueprintInternalUseOnly	Blueprint, UHT	指示不应向最终用户公开此函数。蓝图内部调用，不暴露给用户。	★★★
CustomThunk	UHT	指定UHT不为该函数生成蓝图调用的辅助函数，而需要用户自定义编写。	★★★
Variadic	Blueprint, UHT	标识一个函数可以接受任意类型的多个参数（包括input/output）。	★★★
FieldNotify	UHT	为该函数创建一个FieldNotify的绑定点。	★★★

UPARAM(标识符)

Blueprint

Name	功能描述	引擎模块	常用程度
DisplayName	更改函数参数在蓝图节点上的显示名字	Blueprint, Parameter	★★★★★
ref	使得函数的参数变成引用类型	Blueprint, Parameter	★★★★★
Const	指定该函数参数不可更改	Blueprint, Parameter	★
Required	指定函数的参数节点必须连接提供一个值	Blueprint, Parameter	★★

Network

Name	功能描述	引擎模块	常用程度
NotReplicated		Blueprint, Network, Parameter	💀

UPROPERTY(标识符)

Serialization

Name	引擎模块	功能描述	常用程度
Export	Serialization	在对Asset导出的时候，决定该类的对象应该导出内部的属性值，而是对象的路径。	★

Name	引擎模块	功能描述	常用程度
SaveGame	Serialization	在SaveGame存档的时候，只序列化有SaveGame标记的属性，而不序列化别的属性。	★★★★★
SkipSerialization	Serialization	二进制序列化时跳过该属性，但在ExportText的时候依然可以导出。	★★★
TextExportTransient	Serialization	在ExportText导出为.COPY格式的时候，忽略该属性。	★
Transient	Serialization	不序列化该属性，该属性初始化时候会被0填充。	★★★★★
DuplicateTransient	Serialization	在对象复制或COPY格式导出的时候，忽略该属性。	★★
NonPIEDuplicateTransient	Serialization	在对象复制的时候，且在不是PIE的场合，忽略该属性。	★

Sequencer

Name	引擎模块	功能描述	常用程度
Interp	Sequencer	指定该属性值可暴露到时间轴里编辑，在平常的Timeline或UMG的动画里使用。	★★★

Network

Name	引擎模块	功能描述	常用程度
Replicated	Network	指定该属性应随网络进行复制。	★★★★★
ReplicatedUsing	Network	指定一个通知回调函数，在属性通过网络更新后执行。	★★★★★
NotReplicated	Network	跳过复制。这只会应用到服务请求函数中的结构体成员和参数。	★★★
RepRetry	Network	只适用于结构体属性。如果此属性未能完全发送（举例而言：Object引用尚无法通过网络进行序列化），则重新尝试对其的复制。对简单引用而言，这是默认选择；但对结构体而言，这会产生带宽开销，并非优选项。因此在指定此标签之前其均为禁用状态。	💀

UHT

Name	引擎模块	功能描述	常用程度
FieldNotify	MVVM, UHT	在打开MVVM插件后，使得该属性变成支持 FieldNotify的属性。	★★★★

Instance

Name	引擎模块	功能描述	常用程度
Instanced	Instance	指定对该对象属性的编辑赋值应该新创建一个实例并作为 子对象，而不是寻找一个对象引用。	★★★

Editor

Name	引擎模块	功能描述	常用程度
NonTransactional	Editor	对该属性的改变操作，不会被包含进编辑器的 Undo/Redo命令中。	★★

DetailsPanel

Name	引擎模块	功能描述	常用程度
Category	DetailsPanel, Editor	指定属性的类别，使用 运算符定义 嵌套类目。	★★★★★
SimpleDisplay	DetailsPanel, Editor	在细节面板中直接可见，不折叠到高 级中。	★★★
AdvancedDisplay	DetailsPanel, Editor	被折叠到高级栏下，要手动打开。一 般用在不太常用的属性上面。	★★★★★
EditAnywhere	DetailsPanel, Editor	在默认值和实例的细节面板上均可编 辑	★★★★★
EditDefaultsOnly	DetailsPanel, Editor	只能在默认值面板里编辑	★★★★★
EditInstanceOnly	DetailsPanel, Editor	只能在实例的细节面板上编辑该属性	★★★★★
VisibleAnywhere	DetailsPanel, Editor	在默认值和实例细节面板均可见，但 不可编辑	★★★★★
VisibleDefaultsOnly	DetailsPanel, Editor	在默认值细节面板可见，但不可编辑	★★★★★

Name	引擎模块	功能描述	常用程度
VisibleInstanceOnly	DetailsPanel, Editor	在实例细节面板可见，但不可编辑	★★★★★
EditFixedSize	DetailsPanel, Editor	在细节面板上不允许改变该容器的元素个数。	★★★
NoClear	DetailsPanel, Editor	指定该属性的编辑选项中不出现Clear按钮，不允许置null。	★★★

Config

Name	引擎模块	功能描述	常用程度
Config	Config	指定该属性是一个配置属性，该属性可以被序列化读写到ini文件（路径由uclass的config标签指定）中。	★★★
GlobalConfig	Config	和Config一样指定该属性可作为配置读取和写入ini中，但只会读取写入到配置文件里基类的值，而不会使用配置文件里子类里的值。	★★★

Blueprint

Name	引擎模块	功能描述	常用程度
BlueprintAuthorityOnly	Blueprint, Network	只能绑定为BlueprintAuthorityOnly的事件，让该多播委托只接受在服务端运行的事件	★★★
BlueprintReadWrite	Blueprint	可从蓝图读取或写入此属性。	★★★★★
BlueprintReadOnly	Blueprint	此属性可由蓝图读取，但不能被修改。	★★★★★
BlueprintGetter	Blueprint	为属性定义一个自定义的Get函数来读取。	★★★
Getter	Blueprint	为属性增加一个C++的Get函数，只在C++层面应用。	★★★
Setter	Blueprint	为属性增加一个C++的Set函数，只在C++层面应用。	★★★
BlueprintSetter	Blueprint	采用一个自定义的set函数来读取。	★★★
BlueprintCallable	Blueprint	在蓝图中可以调用这个多播委托	★★★
BlueprintAssignable	Blueprint	在蓝图中可以为这个多播委托绑定事件	★★★

Behavior

Name	引擎模块	功能描述	常用程度
Localized	Behavior	此属性的值将拥有一个定义的本地化值。多用于字符串。暗示为 ReadOnly。该值有一个本地化值。最常标记在string上	💀
Native	Behavior	属性为本地：C++代码负责对其进行序列化并公开到垃圾回收。	💀

Asset

Name	引擎模块	功能描述	常用程度
AssetRegistrySearchable	Asset	标记该属性可以作为AssetRegistry的Tag和Value值来进行资产的过滤搜索	★★★

Meta = (元数据)

Any

Name	引擎模块	功能描述	常用程度
FullyExpand			💀
HideThen		隐藏异步蓝图节点的Then引脚	💀
MapKeyParam		指定一个函数参数为Map的Key，其根据MapParam指定的实际Map参数的Key类型而相应改变。	★★★
EditConditionHides		在已经有EditCondition的情况下，指定该属性在EditCondition不满足的情况下隐藏起来。	★★★★★
InlineEditConditionToggle		使这个bool属性在被用作EditCondition的时候内联到对方的属性行里成为一个单选框，而不是自己成为一个编辑行。	★★★★★
NeedsLatentFixup		用在FLatentActionInfo::Linkage属性上，告诉蓝图VM生成跳转信息	★
HideSpawnParms		在UGameplayTask子类生成的蓝图异步节点上隐藏UGameplayTask子类继承链中某些属性。	💀
ShowOnlyInnerProperties		把结构属性的内部属性直接上提一个层级直接展示	★★★
LatentInfo		和Latent配合，指明哪个函数参数是LatentInfo参数。	★★★

Name	引擎模块	功能描述	常用程度
NativeBreakFunc		指定一个函数采用BreakStruct的图标。	★
ShowCategories		显示类别	💀
HasNativeMake		为该结构指定一个C++内的UFunction函数作为Mreak节点的实现	★★★★★
EntryClass		限定EntryWidgetClass属性上可选类必须继承自的基类，用在DynamicEntryBox和ListView这两个Widget上。	★★★
NoEditInline		Object properties pointing to an UObject instance whos class is marked editinline will not show their properties inline in property windows. Useful for getting actor components to appear in the component tree but not inline in the root actor details panel.	💀
NotBlueprintThreadSafe		用在函数上，标记这个函数是不线程安全的	★
ShowWorldContextPin		放在UCLASS上，指定本类里的函数调用都必须显示WorldContext引脚，无论其本来是否默认隐藏	💀
ChildCannotTick		用于Actor或ActorComponent子类，标记允许其蓝图子类不可以接受响应Tick事件，哪怕父类可以Tick	★★★
DynamicOutputParam		配合DeterminesOutputType，指定多个支持动态类型的输出参数。	💀
HasDedicatedAsyncNode			💀
LatentCallbackTarget		用在FLatentActionInfo::CallbackTarget属性上，告诉蓝图VM在哪个对象上调用函数。	★
MapValueParam		指定一个函数参数为Map的Value，其根据MapParam指定的实际Map参数的Value类型而相应改变。	★★★
UseEnumValuesAsMaskValuesInEditor		指定枚举值已经是位移后的值，而不是位标记的索引下标。	★★
ExpandBoolAsExecs		是ExpandEnumAsExecs的别名，完全等价其功能。	★★★★★
AllowPrivateAccess		允许一个在C++中private的属性，可以在蓝图中访问。	★★★★★
AllowEditInlineCustomization		允许EditInline的对象属性可以自定义属性细节面板来编辑该对象内的数据。	★
BlueprintPrivate		指定该函数或属性只能在本类中被调用或读写，类似C++中的private的作用域限制。不可在别的蓝图类里访问。	★★

Name	引擎模块	功能描述	常用程度
CollapsableChildProperties		在TextureGraph模块中新增加的meta。用于折叠一个结构的内部属性。	💀
BitmaskEnum		使用位标记后采用的枚举名字	★★★★★
ShortTooltip		提供一个更简洁版本的提示文本，例如在类型选择器的时候显示	💀
ArrayTypeDependentParams		当ArryParam指定的函数拥有两个或以上Array参数的时候，指定哪些数组参数的类型也应该相应的被更新改变。	💀
ForceInlineRow		强制TMap属性里的结构key和其他Value合并到同一行来显示	★
CallableWithoutWorldContext		让函数也可以脱离WorldContextObject而使用	💀

Actor

Name	引擎模块	功能描述	常用程度
ChildCanTick	Actor	标记允许其蓝图子类可以接受响应Tick事件	★★★

AnimationGraph

Name	引擎模块	功能描述	常用程度
AnimNotifyBoneName	AnimationGraph	使得UAnimNotify或UAnimNotifyState下的FName属性作为BoneName的作用。	★★
AnimBlueprintFunction	AnimationGraph	标明是动画蓝图里的内部纯存根函数，只在动画蓝图编译时设置	💀
CustomizeProperty	AnimationGraph	使用在FAnimNode的成员属性上，告诉编辑器不要为它生成默认Details面板控件，后续会在DetailsCustomization里自定义创建相应的编辑控件。	★
AnimNotifyExpand	AnimationGraph	使得UAnimNotify或UAnimNotifyState下的属性直接展开到细节面板里。	💀
OnEvaluate	AnimationGraph		💀
FoldProperty	AnimationGraph	在动画蓝图中使得动画节点的某个属性成为FoldProperty。	★

Name	引擎模块	功能描述	常用程度
BlueprintCompilerGeneratedDefaults	AnimationGraph	指定该属性的值是编译器生成的，因此在编译后无需复制，可以加速一些编译性能。	💀
CustomWidget	AnimationGraph		💀
AllowedParamType	AnimationGraph		💀
PinShownByDefault	AnimationGraph	在动画蓝图中使得动画节点的某个属性一开始就暴露出来成为引脚，但也可以改变。	★★★
GetterContext	AnimationGraph	继续限定AnimGetter函数在哪个地方才可以使用，如果不填，则默认都可以用。	★★
AnimGetter	AnimationGraph	指定UAnimInstance及子类的该函数成为一个AnimGetter函数。	★★★

Asset

Name	引擎模块	功能描述	常用程度
DisallowAssetDataTags	Asset	在UObject*属性上指定Tags来进行过滤，必须没有拥有该Tags才可以被选择。	★★
RequiredAssetDataTags	Asset	在UObject*属性上指定Tags来进行过滤，必须拥有该Tags才可以被选择。	★★
ForceShowEngineContent	Asset	指定UObject*属性的资源可选列表里强制可选引擎的内建资源	★★
ForceShowPluginContent	Asset	指定UObject*属性的资源可选列表里强制可选其他插件里的内建资源	💀
GetAssetFilter	Asset	指定一个UFUNCTION来对UObject*属性的可选资源进行排除过滤。	★★★

Blueprint

Name	引擎模块	功能描述	常用程度
IgnoreTypePromotion	Blueprint	标记该函数不收录进类型提升函数库	★★
Variadic	Blueprint	指定该函数接受多个参数	★★★
ForceAsFunction	Blueprint	把C++里用 BlueprintImplementableEvent或 NativeEvent定义的事件强制改为函数在子类中覆写。	★★★
CannotImplementInterfaceInBlueprint	Blueprint	指定该接口不能在蓝图中实现	★★★

Name	引擎模块	功能描述	常用程度
CallInEditor	Blueprint	可以在Actor的细节面板上作为一个按钮来调用该函数。	★★★★★
BlueprintProtected	Blueprint	指定该函数或属性只能在本类以及子类中被调用或读写，类似C++中的protected作用域限制。不可在别的蓝图类里访问。	★★★
CommutativeAssociativeBinaryOperator	Blueprint	标记一个二元运算函数的运算支持交换律和结合律，在蓝图节点上增加一个“+”引脚，允许动态的直接添加多个输入值。	★★★★
CompactNodeTitle	Blueprint	使得函数的展示形式变成精简压缩模式，同时指定一个新的精简的名字	★★★
CustomStructureParam	Blueprint	被CustomStructureParam标记的函数参数会变成Wildcard的通配符参数，其引脚的类型会等于连接的变量类型。	★★★★★
DefaultToSelf	Blueprint	用在函数上，指定一个参数的默认值为Self值	★★★★★
ExpandEnumAsExecs	Blueprint	指定多个enum或bool类型的函数参数，自动根据条目生成相应的多个输入或输出执行引脚，并根据实参值不同来相应改变控制流。	★★★★★
ArrayParm	Blueprint	指定一个函数为使用Array<*>的函数，数组元素类型为通配符的泛型。	★★★
AdvancedDisplay	Blueprint	把函数的一些参数折叠起来不显示，需要手动点开下拉箭头来展开编辑。	★★★★★
SetParam	Blueprint	指定一个函数为使用Set的函数，元素类型为通配符的泛型。	★★★
MapParam	Blueprint	指定一个函数为使用TMap< TKey, TValue >的函数，元素类型为通配符的泛型。	★★★
InternalUseParam	Blueprint	用在函数调用上，指定要隐藏的参数名称，也可以隐藏返回值。可以隐藏多个	★★★★★
Keywords	Blueprint	指定一系列关键字用于在蓝图内右键找到该函数	★★★★★
Latent	Blueprint	标明一个函数是一个延迟异步操作	★★★★★
NativeMakeFunc	Blueprint	指定一个函数采用MakeStruct的图标	★
UnsafeDuringActorConstruction	Blueprint	标明该函数不能在Actor的构造函数里调用	★★
BlueprintAutocast	Blueprint	告诉蓝图系统这个函数是用来支持从A类型到B类型的自动转换。	★
DeterminesOutputType	Blueprint	指定一个参数的类型作为函数动态调整输出参数类型的参考类型	★★★
ReturnDisplayName	Blueprint	改变函数返回值的名字，默认是ReturnValue	★★★★★
WorldContext	Blueprint	指定函数的一个参数自动接收WorldContext对象，以便确定当前运行所处的World	★★★★★

Name	引擎模块	功能描述	常用程度
AutoCreateRefTerm	Blueprint	指定函数的多个输入引用参数在没有连接的时候自动为其创建默认值	★★★★★
ProhibitedInterfaces	Blueprint	列出与蓝图类不兼容的接口，阻止实现	★★
HiddenNode	Blueprint	把指定的UBTNode隐藏不在右键菜单中显示。	★
HideFunctions	Blueprint	在属性查看器中不显示指定类别中的所有函数。	★★★
ExposedAsyncProxy	Blueprint	在 Async Task 节点中公开此类的一个代理对象。	★★★
NotInputConfigurable	Blueprint	让一些UInputModifier和UInputTrigger不能在ProjectSettings里配置。	★
BlueprintThreadSafe	Blueprint	用在类上或函数上，标记类里的函数都是线程安全的。这样就可以在动画蓝图等非游戏线程被调用了。	★★★
RestrictedToClasses	Blueprint	限制蓝图函数库下的函数只能在 RestrictedToClasses 指定的类蓝图中右键创建出来	★★★
DontUseGenericSpawnObject	Blueprint	阻止使用蓝图中的 Generic Create Object 节点来生成本类的对象。	★★
ObjectSetType	Blueprint	指定统计页面的对象集合类型。	★
SparseClassDataTypes	Blueprint		★★★
KismetHideOverrides	Blueprint	不允许被覆盖的蓝图事件的列表。	💀
BlueprintType	Blueprint	表明可以作为一个蓝图变量	★★★★★
IsConversionRoot	Blueprint	允许Actor在自身以及子类之间做转换	★★★
BlueprintInternalUseOnlyHierarchical	Blueprint	标明该结构及其子类都不暴露给用户定义和使用，均只能在蓝图系统内部使用	★
BlueprintSetter	Blueprint	采用一个自定义的set函数来读取。会默认设置 BlueprintReadWrite.	★★★
DisplayName	Blueprint	此节点在蓝图中的命名将被此处提供的值所取代，而非代码生成的命名。	★★★★★
ExposeOnSpawn	Blueprint	使该属性在 ConstructObject 或 SpawnActor 等创建对象的时候暴露出来。	★★★★★
NativeConst	Blueprint	指定有C++里的const标志	★
CPP_Default_XXX	Blueprint	XXX=参数名字	★★★★★
BlueprintGetter	Blueprint	采用一个自定义的get函数来读取。如果没有设置 BlueprintSetter 或 BlueprintReadWrite，则会默认设置 BlueprintReadOnly.	★★★
IsBlueprintBase	Blueprint	说明此类是否为创建蓝图的一个可接受基类，与 UCLASS 说明符、 Blueprintable 或 'NotBlueprintable' 相似。	★★★★★

Name	引擎模块	功能描述	常用程度
BlueprintInternalUseOnly	Blueprint	标明该元素是作为蓝图系统的内部调用或使用，不暴露出来在用户层面直接定义或使用。	★★★

Component

Name	引擎模块	功能描述	常用程度
UseComponentPicker	Component	用在ComponentReference属性上，使得选取器的列表里展示出Actor属下的Component以便选择。	★★
AllowAnyActor	Component	用在ComponentReference属性上，在UseComponentPicker的情况下使得组件选取器扩大到场景里其他Actor下的其他组件。	★★
BlueprintSpawnableComponent	Component	允许该组件出现在Actor蓝图里Add组件的面板里。	★★★★

Config

Name	引擎模块	功能描述	常用程度
ConsoleVariable	Config	把一个Config属性的值同步到一个同名的控制台变量。	★★★★★
EditorConfig	Config	保存编辑器的配置	★★★
ConfigHierarchyEditable	Config	使得一个属性可以在Config的各个层级配置。	★★★
ConfigRestartRequired	Config	使属性在设置里改变后弹出重启编辑器的对话框。	★★★

Container

Name	引擎模块	功能描述	常用程度
ReadOnlyKeys	Container	使TMap属性的Key不能编辑。	★★
ArraySizeEnum	Container	为固定数组提供一个枚举，使得数组元素按照枚举值来作为索引和显示。	★★★
TitleProperty	Container	指定结构数组里的结构成员属性内容来作为结构数组元素的显示标题。	★★

Name	引擎模块	功能描述	常用程度
EditFixedOrder	Container	使数组的元素无法通过拖拽来重新排序。	★★
NoElementDuplicate	Container	去除TArray属性里数据项的Duplicate菜单项按钮。	★

Debug

Name	引擎模块	功能描述	常用程度
DebugTreeLeaf	Debug	阻止BlueprintDebugger展开该类的属性以加速编辑器里调试器的性能	★

DetailsPanel

Name	引擎模块	功能描述	常用程度
HideInDetailPanel	DetailsPanel	在Actor的事件面板里隐藏该动态多播委托属性。	★★
DisplayAfter	DetailsPanel	使本属性在指定的属性之后显示。	★★★
EditCondition	DetailsPanel	给一个属性指定另外一个属性或者表达式来作为是否可编辑的条件。	★★★★★
DisplayPriority	DetailsPanel	指定本属性在细节面板的显示顺序优先级，越小的优先级越高。	★★★
AdvancedClassDisplay	DetailsPanel	指定该类型的变量在高级显示里显示	★★★
bShowOnlyWhenTrue	DetailsPanel	根据编辑器config配置文件里字段值来决定当前属性是否显示。	★
PrioritizeCategories	DetailsPanel	把指定的属性目录优先显示在前面	★★★
AutoExpandCategories	DetailsPanel	指定类内部的属性目录自动展开起来	★★★
AutoCollapseCategories	DetailsPanel	指定类内部的属性目录自动折叠起来	★★★
HideEditConditionToggle	DetailsPanel	用在使用EditCondition的属性上，表示该属性不想要其EditCondition用到的属性被隐藏起来。	★★★★★
ClassGroupNames	DetailsPanel	指定ClassGroup的名字	★★★

Name	引擎模块	功能描述	常用程度
MaxPropertyDepth	DetailsPanel	指定对象或结构在细节面板里展开的层数。	★
DeprecatedNode	DetailsPanel	用于BehaviorTreeNode或EnvQueryNode，说明该类已废弃，在编辑器中红色错误展示并有错误ToolTip提示	★★
UsesHierarchy	DetailsPanel	说明类使用层级数据。用于实例化“细节”面板中的层级编辑功能。	💀
IgnoreCategoryKeywordsInSubclasses	DetailsPanel	用于让一个类的首个子类忽略所有继承的ShowCategories和HideCategories说明符。	★
NoResetToDefault	DetailsPanel	禁用和隐藏属性在细节面板上的“重置”功能。	★★★
ReapplyCondition	DetailsPanel	// Properties that have a ReapplyCondition should be disabled behind the specified property when in reapply mode	★
HideBehind	DetailsPanel	只在指定的属性为true或不为空的时候本属性才显示	★
Category	DetailsPanel	指定属性在细节面板中的分类	★★★★★
HideCategories	DetailsPanel	隐藏的类别	★★★
EditInline	DetailsPanel	为对象属性创建一个实例，并作为子对象。	★★★

Development

Name	引擎模块	功能描述	常用程度
DeprecatedProperty	Development	标记弃用，引用到该属性的蓝图会触发一个警告	★
Deprecated	Development	指定该元素要废弃的引擎版本号。	★
DevelopmentOnly	Development	使得一个函数变为DevelopmentOnly，意味着只会在Development模式中运行。适用于调试输出之类的功能，但在最终发布版中会跳过。	★
DeprecationMessage	Development	定义弃用的消息	★
DeprecatedFunction	Development	标明一个函数已经被弃用	★

Name	引擎模块	功能描述	常用程度
Comment	Development	用来记录注释的内容	★★★
FriendlyName	Development	和DisplayName一样?	💀
DevelopmentStatus	Development	标明开发状态	★
ToolTip	Development	在Meta里提供一个提示文本，覆盖代码注释里的文本	★★★

Enum

Name	引擎模块	功能描述	常用程度
Enum	Enum	给一个String指定以枚举里值的名称作为选项	★★★
GetRestrictedEnumValues	Enum	指定一个函数来指定枚举属性值的哪些枚举选项是禁用的	★★★
EnumValueDisplayNameOverrides	Enum	改变枚举属性值上的显示名字	★★
EnumDisplayNameFn	Enum	在Runtime下为枚举字段提供自定义名称的函数回调	★★
Bitflags	Enum	设定一个枚举支持采用位标记赋值，从而在蓝图中可以识别出来是BitMask	★★★★★
Spacer	Enum	隐藏UENUM的某个值	★★★★★
InvalidEnumValues	Enum	指定枚举属性值上不可选的枚举值选项，用以排除一些选项	★★★
ValidEnumValues	Enum	指定枚举属性值上可选的枚举值选项	★★★
<u>DisplayName</u>	Enum	改变枚举值的显示名称	★★★★★
<u>Hidden</u>	Enum	隐藏UENUM的某个值	★★★★★
DisplayValue	Enum	Enum /Script/Engine.AnimPhysCollisionType	💀
Grouping	Enum	Enum /Script/Engine.EAlphaBlendOption	💀
TraceQuery	Enum	Enum /Script/Engine.ECollisionChannel	💀
Bitmask	Enum	设定一个属性采用Bitmask赋值	★★★★★

FieldNotify

Name	引擎模块	功能描述	常用程度
FieldNotifyInterfaceParam	FieldNotify	指定函数的某个参数提供FieldNotify的ViewModel信息。	★★★

GAS

Name	引擎模块	功能描述	常用程度
HideInDetailsView	GAS	把该UAttributeSet子类里的属性隐藏在FGameplayAttribute的选项列表里。	★★★
SystemGameplayAttribute	GAS	把UAbilitySystemComponent子类里面的属性暴露到FGameplayAttribute 选项框里。	★★★
HideFromModifiers	GAS	指定AttributeSet下的某属性不出现在GameplayEffect下的Modifiers的Attribute选择里。	★★★

Material

Name	引擎模块	功能描述	常用程度
MaterialParameterCollectionFunction	Material	指定该函数是用于操作UMaterialParameterCollection，从而支持ParameterName的提取和验证	★★★
MaterialNewHSLGenerator	Material	标识该UMaterialExpression为采用新HSL生成器的节点，当前在材质蓝图右键菜单中隐藏。	★
ShowAsInputPin	Material	使得UMaterialExpression里的一些基础类型属性变成材质节点上的引脚。	★★★
RequiredInput	Material	在UMaterialExpression中指定FExpressionInput属性是否要求输入，引脚显示白色或灰色。	💀
MaterialControlFlow	Material	标识该UMaterialExpression为一个控制流节点，当前在材质蓝图右键菜单中隐藏。	★
OverridingInputProperty	Material	在UMaterialExpression中指定本float要覆盖的其他FExpressionInput 属性。	★★★
Private	Material	标识该UMaterialExpression为私有节点，当前在材质蓝图右键菜单中隐藏。	★

Niagara

Name	引擎模块	功能描述	常用程度
NiagaraClearEachFrame	Niagara	ScriptStruct /Script/Niagara.NiagaraSpawnInfo	💀
NiagaralInternalType	Niagara	指定该结构的类型为Niagara的内部类型。	💀

Numeric

Name	引擎模块	功能描述	常用程度
CtrlMultiplier	Numeric	指定数字输入框在Ctrl按下时鼠标轮滚动和鼠标拖动改变值的倍率。	★★
ShiftMultiplier	Numeric	指定数字输入框在Shift按下时鼠标轮滚动和鼠标拖动改变值的倍率。	★★
SliderExponent	Numeric	指定数字输入框上滚动条拖动的变化指数分布	★★★★★
Multiple	Numeric	指定数字的值必须是Mutliple提供的值的整数倍。	★★★
ForceUnits	Numeric	固定设定属性值的单位保持不变，不根据数值动态调整显示单位。	★★★
Units	Numeric	设定属性值的单位，支持实时根据数值不同动态改变显示的单位。	★★★
LinearDeltaSensitivity	Numeric	在设定Delta后，进一步设定数字输入框变成线性改变以及改变的敏感度（值越大越不敏感）	★★★
Delta	Numeric	设定数字输入框值改变的幅度为Delta的倍数	★★★
UIMax	Numeric	指定数字输入框上滚动条拖动的最大范围值	★★★★★
UIMin	Numeric	指定数字输入框上滚动条拖动的最小范围值	★★★★★
SupportDynamicSlider.MaxValue	Numeric	支持数字输入框上滚动条的最大范围值在Alt按下时被动态改变	★
ClampMax	Numeric	指定数字输入框实际接受的最大值	★★★★★
ClampMin	Numeric	指定数字输入框实际接受的最小值	★★★★★
ArrayClamp	Numeric	限定整数属性的值必须在指定数组的合法下标范围内， [0,ArrayClamp.Size()-1]	★★★
HideAlphaChannel	Numeric	使FColor或FLinearColor属性在编辑的时候隐藏Alpha通道。	★★★

Name	引擎模块	功能描述	常用程度
AllowPreserveRatio	Numeric	在细节面板上为FVector属性添加一个比率锁。	★★★
NoSpinbox	Numeric	使数值属性禁止默认的拖放和滚轮的UI编辑功能，数值属性包括int系列以及float系列。	★★
SupportDynamicSliderMinValue	Numeric	支持数字输入框上滚动条的最小范围值在Alt按下时被动态改变	★
sRGB	Numeric	使FColor或FLinearColor属性在编辑的时候采用sRGB方式。	💀
WheelStep	Numeric	指定数字输入框上鼠标轮上下滚动产生的变化值	★★★
InlineColorPicker	Numeric	使FColor或FLinearColor属性在编辑的时候直接内联一个颜色选择器。	★★
ShowNormalize	Numeric	使得FVector变量在细节面板出现一个正规化的按钮。	★★★
ColorGradingMode	Numeric	使得一个FVector4属性成为颜色显示	★★

Object

Name	引擎模块	功能描述	常用程度
ThumbnailSize	Object	改变缩略图的大小。	💀
LoadBehavior	Object	用在UCLASS上标记这个类的加载行为，使得相应的TObjectPtr属性支持延迟加载。可选的加载行为默认为Eager，可改为LazyOnDemand。	★
DisplayThumbnail	Object	指定是否在该属性左侧显示一个缩略图。	★★★
ShowInnerProperties	Object	在属性细节面板中显示对象引用的内部属性	★★★★★
Untracked	Object	使得TSOFTObjectPtr和FSOFTObjectPath的软对象引用类型的属性，不跟踪记录资产的。	★
HideAssetPicker	Object	隐藏Object类型引脚上的AssetPicker的选择列表	★★

Name	引擎模块	功能描述	常用程度
AssetBundles	Object	标明该属性其引用的资产属于哪一些AssetBundles。	★★★
MustBeLevelActor	Object		
ExposeFunctionCategories	Object	指定该Object属性所属于的类里的某些目录下的函数可以直接在本类上暴露出来。	★★★
IncludeAssetBundles	Object	用于UPrimaryDataAsset的子对象属性，指定应该继续递归到孩子对象里去探测AssetBundle数据。	★★

Path

Name	引擎模块	功能描述	常用程度
RelativeToGameContentDir	Path	使得系统目录选择对话框的结果为相对Content的相对路径。	💀
ContentDir	Path	使用UE的风格来选择Content以及子目录。	★★★
LongPackageName	Path	使用UE的风格来选择Content以及子目录，或者把文件路径转换为长包名。	★★★
FilePathFilter	Path	设定文件选择器的扩展名，规则符合系统对话框的格式规范，可以填写多个扩展名。	★★★
RelativePath	Path	使得系统目录选择对话框的结果为当前运行exe的相对路径。	💀
RelativeToGameDir	Path	如果系统目录选择框的结果为Project的子目录，则转换为相对路径，否则返回绝对路径。	★★★

Pin

Name	引擎模块	功能描述	常用程度
HidePin	Pin	用在函数调用上，指定要隐藏的参数名称，也可以隐藏返回值。可以隐藏多个参数	★★
InternalUseParam	Pin	用在函数调用上，指定要隐藏的参数名称，也可以隐藏返回值。可以隐藏多个参数	★★

Name	引擎模块	功能描述	常用程度
HideSelfPin	Pin	用在函数调用上，隐藏默认的SelfPin，也就是Target，导致该函数只能在OwnerClass内调用。	★★
DataTablePin	Pin	指定一个函数参数为DataTable或CurveTable类型，以便为FName的其他参数提供RowNameList的选择。	★★
DisableSplitPin	Pin	禁用Struct的split功能	★★
HiddenByDefault	Pin	Struct的Make Struct和Break Struct节点中的引脚默认为隐藏状态	★
AlwaysAsPin	Pin	在动画蓝图中使得动画节点的某个属性总是暴露出来成为引脚	★★★
NeverAsPin	Pin	在动画蓝图中使得动画节点的某个属性总是不暴露出来成为引脚	★★★
PinHiddenByDefault	Pin	使得这个结构里的属性在蓝图里作为引脚时默认是隐藏的。	★★

RigVMStruct

Name	引擎模块	功能描述	常用程度
Hidden	RigVMStruct	指定FRigUnit下的该属性隐藏	★★★
Input	RigVMStruct	指定FRigUnit下的该属性作为输入引脚。	★★★★★
TemplateName	RigVMStruct	指定该FRigUnit成为一个泛型模板节点。	★★★
CustomWidget	RigVMStruct	指定该FRigUnit里的属性要用自定义的控件来编辑。	★★
ExpandByDefault	RigVMStruct	把FRigUnit里的属性引脚默认展开。	★★★
Aggregate	RigVMStruct	指定FRigUnit里的属性引脚为可扩展连续二元运算符的运算数。	★★★
Visible	RigVMStruct	指定FRigUnit下的该属性为常量引脚，无法连接变量。	★★★
Output	RigVMStruct	指定FRigUnit下的该属性作为输出引脚。	★★★★★
DetailsOnly	RigVMStruct	指定FRigUnit下的该属性只在细节面板中显示。	★★★
Varying	RigVMStruct	ScriptStruct /Script/RigVM.RigVMFunction_GetDeltaTime	💀
MenuDescSuffix	RigVMStruct	标识FRigUnit在蓝图右键菜单项的名字后缀。	★★★
NodeColor	RigVMStruct	指定FRigUnit蓝图节点的RGB颜色值。	★★
Icon	RigVMStruct	设定FRigUnit蓝图节点的图标。	★★
<u>Deprecated</u>	RigVMStruct	标识该FRigUnit为弃用状态，不在蓝图右键菜单中显示。	★★
Abstract	RigVMStruct	标识该FRigUnit为抽象类，不用实现Execute。	★★

Name	引擎模块	功能描述	常用程度
Constant	RigVMStruct	标识一个属性成为一个常量的引脚。	★★★
RigVMTypAllowed	RigVMStruct	指定一个UENUM可以在FRigUnit的UEnum*属性中被选择。	★★
Keywords	RigVMStruct	设定FRigUnit蓝图节点在右键菜单中的关键字，方便输入查找。	★★★

Scene

Name	引擎模块	功能描述	常用程度
MakeEditWidget	Scene	使FVector和FTransform在场景编辑器里出现一个可移动的控件。	★★★
ValidateWidgetUsing	Scene	提供一个函数来验证当前属性值是否合法。	★★★
AllowedLocators	Scene	用来给Sequencer定位可绑定的对象	★

Script

Name	引擎模块	功能描述	常用程度
ScriptNoExport	Script	不导出该函数或属性到脚本。	★★★
ScriptConstant	Script	把一个静态函数的返回值包装成为一个常量值。	★★★
ScriptMethodMutable	Script	把ScriptMethod的第一个const结构参数在调用上改成引用参数，函数内修改的值会保存下来。	★★
ScriptName	Script	在导出到脚本里时使用的名字	★★★
ScriptConstantHost	Script	在ScriptConstant的基础上，指定常量生成的所在类型。	★
ScriptMethodSelfReturn	Script	在ScriptMethod的情况下，指定把这个函数的返回值要去覆盖该函数的第一个参数。	★★
ScriptMethod	Script	把静态函数导出变成第一个参数的成员函数。	★★★
ScriptDefaultBreak	Script		★
ScriptOperator	Script	把第一个参数为结构的静态函数包装成结构的运算符。	★★★
ScriptDefaultMake	Script	禁用结构上的HasNativeMake，在脚本里构造的时候不调用C++里的NativeMake函数，而采用脚本内建的默认初始化方式。	★

Sequencer

Name	引擎模块	功能描述	常用程度
TakeRecorderDisplayName	Sequencer	指定UTakeRecorderSource的显示名字。	★★
SequencerBindingResolverLibrary	Sequencer	把具有SequencerBindingResolverLibrary标记的UBlueprintFunctionLibrary作为动态绑定的类。	★★
CommandLineID	Sequencer	标记UMovieSceneCaptureProtocolBase的子类的协议类型。	★★

Serialization

Name	引擎模块	功能描述	常用程度
SkipUCSModifiedProperties	Serialization	跳过序列化Component里某个属性	💀
MatchedSerializers	Serialization	只在NoExportTypes.h中使用，标明采用结构序列化器。是否支持文本导入导出	💀

SparseDataType

Name	引擎模块	功能描述	常用程度
NoGetter	SparseDataType	阻止UHT为该属性生成一个C++的get函数，只对稀疏类的结构数据里的属性生效。	★

String/Text

Name	引擎模块	功能描述	常用程度
PasswordField	String/Text	使得文本属性显示为密码框	★★★★★
PropertyValidator	String/Text	用名字指定一个UFUNCTION函数来进行文本的验证	★★★
MultiLine	String/Text	使得文本属性编辑框支持换行。	★★★★★
AllowedCharacters	String/Text	只允许文本框里可以输入这些字符。	★★★
GetValueOptions	String/Text	为TMap里的FName/FString作Value提供细节面板里选项框的选项值	💀

Name	引擎模块	功能描述	常用程度
GetOptions	String/Text	指定一个外部类的函数提供选项给FName或 FString属性在细节面板中下拉选项框提供值列表。	★★★★★
GetKeyOptions	String/Text	为TMap里的FName/FString作为Key提供细节面板里选项框的选项值	💀
MaxLength	String/Text	在文本编辑框里限制文本的最大长度	★★★★★

Struct

Name	引擎模块	功能描述	常用程度
MakeStructureDefaultValue	Struct	存储BP中自定义结构里的属性的默认值。	★
IgnoreForMemberInitializationTest	Struct	使得该属性忽略结构的未初始验证。	★★
HasNativeBreak	Struct	为该结构指定一个C++内的UFunction函数作为Break节点的实现	★★★★★
DataflowFlesh	Struct	ScriptStruct /Script/DataflowNodes.FloatOverrideDataflowNode	💀

TypePicker

Name	引擎模块	功能描述	常用程度
AllowedTypes	TypePicker	为FPrimaryAssetId可以指定允许的资产类型。	★★★
BaseClass	TypePicker	只在StateTree模块中使用，限制FStateTreeEditorNode选择的基类类型。	★
GetDisallowedClasses	TypePicker	用在类选择器上，通过一个函数来指定选择的类型列表中排除掉某一些类型基类。	★★
GetAllowedClasses	TypePicker	用在类或对象选择器上，通过一个函数来指定选择的对象必须属于某一些类型基类。	★★
AllowedClasses	TypePicker	用在类或对象选择器上，指定选择的对象必须属于某一些类型基类。	★★★
DisallowedClasses	TypePicker	用在类或对象选择器上，指定选择的对象排除掉某一些类型基类。	★★★
BaseStruct	TypePicker	指定FInstancedStruct属性选项列表选择的结构都必须继承于BaseStruct指向的结构。	★★★
MetaStruct	TypePicker	设定到UScriptStruct*属性上，指定选择的类型的父结构。	★★★

Name	引擎模块	功能描述	常用程度
ExactClass	TypePicker	在同时设置AllowedClasses和GetAllowedClasses的时候，ExactClass指定只取这两个集合中类型完全一致的类型交集，否则取一致的交集再加上其子类。	★
ShowDisplayNames	TypePicker	在Class和Struct属性上，指定类选择器显示另外的显示名称而不是类原始的名字。	★
DisallowedsStructs	TypePicker	只在SmartObject模块中应用，用以在类选择器中排除掉某个类以及子类。	★
ExcludeBaseStruct	TypePicker	在使用BaseStruct的FInstancedStruct属性上忽略BaseStruct指向的结构基类。	★★★
RowType	TypePicker	指定FDataTableRowHandle 属性的可选行类型的基类。	★★★
MustImplement	TypePicker	指定TSubClassOf或FSoftClassPath属性选择的类必须实现该接口	★★★
ShowTreeView	TypePicker	用于选择Class或Struct的属性上，使得在类选取器中显示为树形而不是列表。	★★
BlueprintBaseOnly	TypePicker	用于类属性，指定是否只接受可创建蓝图子类的基类	★★
MetaClass	TypePicker	用在软引用属性上，限定要选择的对象的基类	★★
StructTypeConst	TypePicker	指定FInstancedStruct属性的类型不能在编辑器被选择。	★
AllowAbstract	TypePicker	用于类属性，指定是否接受抽象类。	★★
HideViewOptions	TypePicker	用于选择Class或Struct的属性上，隐藏在类选取器中修改显示选项的功能。	★
OnlyPlaceable	TypePicker	用在类属性上，指定是否只接受可被放置到场景里的Actor	★★

UHT

Name	引擎模块	功能描述	常用程度
DocumentationPolicy	UHT	指定文档验证的规则，当前只能设为Strict	★
GetByRef	UHT	指定UHT为该属性生成返回引用的C++代码	💀
CustomThunk	UHT	指定UHT不为该函数生成蓝图调用的辅助函数，而需要用户自定义编写。	★★★★★

Name	引擎模块	功能描述	常用程度
NativeConstTemplateArg	UHT	指定该属性是一个const的模板参数。	💀
CppFromBpEvent	UHT		💀
IncludePath	UHT	记录UClass的引用路径	💀
ModuleRelativePath	UHT	记录类型定义的头文件路径，为其处于模块的内部相对路径。	💀

Widget

Name	引擎模块	功能描述	常用程度
DisableNativeTick	Widget	禁用该UserWidget的NativeTick。	★★★
ViewmodelBlueprintWidgetExtension	Widget	用来验证InListItems的Object类型是否符合EntryWidgetClass的MVVM绑定的ViewModelProperty。	💀
DesignerRebuild	Widget	指定Widget里的某个属性值改变后应该重新刷新UMG的预览界面。	★
DefaultGraphNode	Widget	标记引擎默认创建的蓝图节点。	💀
BindWidget	Widget	指定在C++类中该Widget属性一定要绑定到UMG的某个同名控件。	★★★★★
BindWidgetOptional	Widget	指定在C++类中该Widget属性可以绑定到UMG的某个同名控件，也可以不绑定。	★★★
OptionalWidget	Widget	指定在C++类中该Widget属性可以绑定到UMG的某个同名控件，也可以不绑定。	★★★
BindWidgetAnimOptional	Widget	指定在C++类中该UWidgetAnimation属性可以要绑定到UMG下的某个动画，也可以不绑定。	★★★
IsBindableEvent	Widget	把一个动态单播委托暴露到UMG蓝图里以绑定相应事件。	★★★
EntryInterface	Widget	限定EntryWidgetClass属性上可选类必须实现的接口，用在DynamicEntryBox和ListView这两个Widget上。	★★★

Name	引擎模块	功能描述	常用程度
BindWidgetAnim	Widget	指定在C++类中该UWidgetAnimation属性一定要绑定到UMG下的某个动画	★★★★★

ClassFlags :

Name	Feature	Trait	Value	Description	UCLASS	Related to UPROPERTY
CLASS_Abstract	Blueprint		0x00000001	指定这个类是抽象基类，不可实例化	Abstract (Specifier/UCLASS/Abstract.md)	
CLASS_Const	Blueprint	Inherit	0x00010000	该类的所有属性和函数都是const的，也应该被暴露为const	Const (Specifier/UCLASS/Const.md)	
CLASS_CompiledFromBlueprint	Blueprint		0x00040000u	指定该类从蓝图的编译中创建		
CLASS_NewerVersionExists	Blueprint		0x80000000u			
CLASS_NoExport	UHT		0x00000100u	不暴露到C++头文件，不生成注册代码	NoExport (Specifier/UCLASS/NoExport.md)	
CLASS_CustomConstructor	UHT		0x000008000u	不创建一个默认构造函数，只在C++环境下使用	CustomConstructor (Specifier/UCLASS/CustomConstructor.md)	
CLASS_Deprecated	Editor	Inherit	0x02000000u	显示废弃警告	Deprecated (Specifier/UCLASS/Deprecated.md)	
CLASS_HideDropDown	Editor		0x04000000u	类不在右键选择框中显示	HideDropDown (Specifier/UCLASS/HideDropDown.md)	
CLASS_EditInlineNew	Editor		0x00001000u	对象可以通过EditinlineNew按钮构造	EditinlineNew (Specifier/UCLASS/EditinlineNew.md), NotEditinlineNew (Specifier/UCLASS/NotEditinlineNew.md)	
CLASS_Hidden	Editor		0x01000000u	不在编辑器的类浏览器和edit inline new中显示		
CLASS_CollapseCategories	Editor		0x00002000u	属性在展示时不分目录	CollapseCategories (Specifier/UCLASS/CollapseCategories.md), DontCollapseCategories (Specifier/UCLASS/DontCollapseCategories.md)	
CLASS_NotPlaceable	Behavior	Inherit	0x00000200u	不能被放置在场景中	Deprecated (Specifier/UCLASS/Deprecated.md), NotPlaceable (Specifier/UCLASS/NotPlaceable.md), Placeable (Specifier/UCLASS/Placeable.md)	
CLASS_ReplicationDataIsSetUp	Behavior		0x00000800u	是否在该类仍然需要调用 SetUpRuntimeReplicationData		
CLASS_MinimalAPI	DllExport		0x00080000u	指定该类的最小导出，只导出获得类指针的函数	MinimalAPI (Specifier/UCLASS/MinimalAPI.md)	
CLASS_RequiredAPI	DllExport	DefaultC++, Internal	0x00100000u	指定该类必须具有DLL导出，导出所有函数和属性	UCLASS() (Specifier/UCLASS/UCLASS().md)	
	DllExport					
CLASS_DefaultToInstanced	LoadConstruct	Inherit	0x02000000u	指定引用到该类的所有引用都默认创建个实例对象	DefaultToInstanced (Specifier/UCLASS/DefaultToInstanced.md)	
CLASS_HasInstancedReference	LoadConstruct	Inherit	0x00800000u	类拥有组件属性		
CLASS_Parsed	LoadConstruct		0x00000001u	成功解析完成		
CLASS_TokenStreamAssembled	LoadConstruct	DefaultC++	0x00400000u	指定父类的TokenStream已经被成功合并到自身类上	UCLASS() (Specifier/UCLASS/UCLASS().md)	
CLASS_LayoutChanging	LoadConstruct			指定该类的内存布局已经被改变，因此目前还不能创建CDO		
CLASS_Constructed	LoadConstruct	DefaultC++	0x20000000u	类已经被构造完成	UCLASS() (Specifier/UCLASS/UCLASS().md)	
CLASS_NeedsDeferredDependencyLoading	LoadConstruct	Inherit		指定该类需要延迟依赖加载	NeedsDeferredDependencyLoading (Specifier/UCLASS/NeedsDeferredDependencyLoading.md)	
CLASS_Transient	LoadConstruct	Inherit	0x00000008u	透明的，在序列化的时候被跳过	Transient (Specifier/UCLASS/Transient.md), NonTransient (Specifier/UCLASS/NonTransient.md)	
CLASS_MatchedSerializers	LoadConstruct	DefaultC++, Internal	0x00000002u		UCLASS() (Specifier/UCLASS/UCLASS().md), MatchedSerializers (Specifier/UCLASS/MatchedSerializers.md)	
CLASS_Native	Traits	DefaultC++	0x00000008u	指定为原生类，C++里创建的类	UCLASS() (Specifier/UCLASS/UCLASS().md)	
CLASS_Intrinsic	Traits	DefaultC++	0x10000000u	类在C++中定义，且没有UHT生成的代码	Intrinsic (Specifier/UCLASS/Intrinsic.md), UCLASS() (Specifier/UCLASS/UCLASS().md)	
CLASS_Interface	Traits		0x00004000u	该类是一个接口	Interface (Specifier/UCLASS/Interface.md)	
CLASS_Optional	Traits	Inherit	0x00000001u	This object type may not be available in certain context. (i.e. game runtime or in certain configuration). Optional class data is saved separately to other object types. (i.e. might use sidecar files)	Optional (Specifier/UCLASS/Optional.md)	
CLASS_Config	Config	Inherit	0x00000004u	在构造的时候载入对象的config配置		
CLASS_DefaultConfig	Config	Inherit	0x00000002u	保存对象配置到DefaultXXX.ini，而不是Local，必须和CLASS_Config选用	DefaultConfig (Specifier/UCLASS/DefaultConfig.md)	
CLASS_ProjectUserConfig	Config	Inherit	0x000000040u	指定settings.config文件保存在Project/User*.ini 和 CLASS_GlobalUserConfig类似	ProjectUserConfig (Specifier/UCLASS/ProjectUserConfig.md)	
CLASS_PerObjectConfig	Config	Inherit	0x000000400u	对每个对象进行配置，而不是在类级别	PerObjectConfig (Specifier/UCLASS/PerObjectConfig.md)	
CLASS_GlobalUserConfig	Config	Inherit	0x08000000u	类Settings被保存到....Blah.ini	GlobalUserConfig (Specifier/UCLASS/GlobalUserConfig.md)	
CLASS_ConfigDoNotCheckDefaults	Config	Inherit	0x40000000u	指定对象配置将不会检查base/defaults ini	ConfigDoNotCheckDefaults (Specifier/UCLASS/ConfigDoNotCheckDefaults.md)	
HasCustomFieldNotify					CustomFieldNotify (Specifier/UCLASS/CustomFieldNotify.md)	

StructFlags :

Name	Value	Description	USTRUCT
STRUCT_NoFlags	0x00000000		
STRUCT_Native	0x00000001		
STRUCT_IdenticalNative	0x00000002	If set, this struct will be compared using native code	
STRUCT_HasInstancedReference	0x00000004		
STRUCT_NoExport	0x00000008		
STRUCT_Atomic	0x00000010	Indicates that this struct should always be serialized as a single unit	Atomic (Specifier/USTRUCT/Atomic.md)
STRUCT.Immutable	0x00000020	Indicates that this struct uses binary serialization; it is unsafe to add/remove members from this struct without incrementing the package version	immutable (Specifier/USTRUCT/immutable.md)
STRUCT.AddStructReferencedObjects	0x00000040	If set, native code needs to be run to find referenced objects	
STRUCT.RequiredAPI	0x00000200	Indicates that this struct should be exportable/importable at the DLL layer. Base structs must also be exportable for this to work.	
STRUCT.NetSerializeNative	0x00000400	If set, this struct will be serialized using the CPP net serializer	
STRUCT.SerializeNative	0x00000800	If set, this struct will be serialized using the CPP serializer	
STRUCT.CopyNative	0x00001000	If set, this struct will be copied using the CPP operator=	
STRUCT.IsPlainOldData	0x00002000	If set, this struct will be copied using memcpy	
STRUCT.NoDestructor	0x00004000	If set, this struct has no destructor and non will be called. STRUCT_IsPlainOldData implies STRUCT_NoDestructor	
STRUCT.ZeroConstructor	0x00008000	If set, this struct will not be constructed because it is assumed that memory is zero before construction.	
STRUCT.ExportTextItemNative	0x00010000	If set, native code will be used to export text	
STRUCT.ImportTextItemNative	0x00020000	If set, native code will be used to export text	
STRUCT.PostSerializeNative	0x00040000	If set, this struct will have PostSerialize called on it after CPP serializer or tagged property serialization is complete	
STRUCT.SerializeFromMismatchedTag	0x00080000	If set, this struct will have SerializeFromMismatchedTag called on it if a mismatched tag is encountered.	
STRUCT.NetDeltaSerializeNative	0x00100000	If set, this struct will be serialized using the CPP net delta serializer	
STRUCT.PostScriptConstruct	0x00200000	If set, this struct will have PostScriptConstruct called on it after a temporary object is constructed in a running blueprint	
STRUCT.NetSharedSerialization	0x00400000	If set, this struct can share net serialization state across connections	
STRUCT.Trashed	0x00800000	If set, this struct has been cleaned and sanitized (trashed) and should not be used	

Name	Value	Description	USTRUCT
STRUCT_NewerVersionExists	0x01000000	If set, this structure has been replaced via reinstancing	
STRUCT_CanEditChange	0x02000000	If set, this struct will have CanEditChange on it in the editor to determine if a child property can be edited	

EnumFlags :

Name	Feature	Value	Description	UENUM	UENUM 1
Flags	Trait	0x00000001	Whether the UEnum represents a set of flags		Flags (Specifier/UENUM/Flags.md)
NewerVersionExists	Trait	0x00000002	If set, this UEnum has been replaced by a newer version		

FunctionFlags :

Name	Feature	Value	Description	UFUNCTION/UDELEGATE	UFUNCTION/UDELEGATE 1	USTRUCT
FUNC_Final	Trait	0x00000001	Function is final (prebindable, non-overridable function).	SealedEvent (Specifier/UFUNCTION/SealedEvent.md)		
FUNC_RequiredAPI	Dll	0x00000002	Indicates this function is DLL exported/imported.			
FUNC_BlueprintAuthorityOnly	Network	0x00000004	Function will only run if the object has network authority	BlueprintAuthorityOnly (Specifier/UFUNCTION/BlueprintAuthorityOnly.md)		
FUNC_BlueprintCosmetic	Network	0x00000008	Function is cosmetic in nature and should not be invoked on dedicated servers	BlueprintCosmetic (Specifier/UFUNCTION/BlueprintCosmetic.md)		
FUNC_Net	Network	0x00000040	Function is network-replicated.	Client (Specifier/UFUNCTION/Client.md), NetMulticast (Specifier/UFUNCTION/NetMulticast.md), Server (Specifier/UFUNCTION/Server.md), ServiceRequest (Specifier/UFUNCTION/ServiceRequest.md), ServiceResponse (Specifier/UFUNCTION/ServiceResponse.md)		
FUNC_NetReliable	Network	0x00000080	Function should be sent reliably on the network.	Reliable (Specifier/UFUNCTION/Reliable.md), ServiceRequest (Specifier/UFUNCTION/ServiceRequest.md), ServiceResponse (Specifier/UFUNCTION/ServiceResponse.md)		
FUNC_NetRequest	Network	0x00000100	Function is sent to a net service	ServiceRequest (Specifier/UFUNCTION/ServiceRequest.md)		
FUNC_Exec	Trait	0x00000200	Executable from command line.	Exec (Specifier/UFUNCTION/Exec.md)		
FUNC_Native	Trait	0x00000400	Native function.	BlueprintImplementableEvent (Specifier/UFUNCTION/BlueprintImplementableEvent.md)		
FUNC_Event	Trait	0x00000800	Event function.	BlueprintImplementableEvent (Specifier/UFUNCTION/BlueprintImplementableEvent.md), BlueprintNativeEvent (Specifier/UFUNCTION/BlueprintNativeEvent.md), ServiceRequest (Specifier/UFUNCTION/ServiceRequest.md), ServiceResponse (Specifier/UFUNCTION/ServiceResponse.md)		
FUNC_NetResponse	Network	0x00001000	Function response from a net service	ServiceResponse (Specifier/UFUNCTION/ServiceResponse.md)		
FUNC_Static		0x00002000	Static function.			
FUNC_NetMulticast	Network	0x00004000	Function is networked multicast Server -> All Clients	NetMulticast (Specifier/UFUNCTION/NetMulticast.md)		
FUNC_UbergraphFunction	Blueprint	0x00008000	Function is used as the merge 'ubergraph' for a blueprint, only assigned when using the persistent 'ubergraph' frame			
FUNC_MulticastDelegate	Trait	0x00010000	Function is a multi-cast delegate signature (also requires FUNC_Delegate to be set!)			
FUNC_Public	Trait	0x00020000	Function is accessible in all classes (if overridden, parameters must remain unchanged).			
FUNC_Private	Trait	0x00040000	Function is accessible only in the class it is defined in (cannot be overridden, but function name may be reused in subclasses. IOW: if overridden, parameters don't need to match, and Super.Fun() cannot be accessed since it's private.)			
FUNC_Protected	Trait	0x00080000	Function is accessible only in the class it is defined in and subclasses (if overridden, parameters must remain unchanged).			
FUNC_Delegate	Trait	0x00100000	Function is delegate signature (either single-cast or multi-cast, depending on whether FUNC_MulticastDelegate is set.)			
FUNC_NetServer	Network	0x00200000	Function is executed on servers (set by replication code if passes check)	Server (Specifier/UFUNCTION/Server.md)		
FUNC_HasOutParms	Trait	0x00400000	function has out (pass by reference) parameters			

Name	Feature	Value	Description	UFUNCTION/UDELEGATE	UFUNCTION/UDELEGATE 1	USTRUCT
FUNC_HasDefaults	Trait	0x00800000	function has structs that contain defaults			HasDefaults (Specifier/USTRUCT/HasDefaults.md)
FUNC_NetClient	Network	0x01000000	function is executed on clients	Client (Specifier/UFUNCTION/Client.md)		
FUNC_DLLImport	Dll	0x02000000	function is imported from a DLL			
FUNC_BlueprintCallable	Blueprint	0x04000000	function can be called from blueprint code	BlueprintGetter (Specifier/UFUNCTION/BlueprintGetter.md), BlueprintPure (Specifier/UFUNCTION/BlueprintPure.md), BlueprintSetter (Specifier/UFUNCTION/BlueprintSetter.md), BlueprintCallable (Specifier/UFUNCTION/BlueprintCallable.md)		
FUNC_BlueprintEvent	Blueprint	0x08000000	function can be overridden/implemented from a blueprint	BlueprintImplementableEvent (Specifier/UFUNCTION/BlueprintImplementableEvent.md), BlueprintNativeEvent (Specifier/UFUNCTION/BlueprintNativeEvent.md)		
FUNC_BlueprintPure	Blueprint	0x10000000	function can be called from blueprint code, and is also pure (produces no side effects). If you set this, you should set FUNC_BlueprintCallable as well.	BlueprintGetter (Specifier/UFUNCTION/BlueprintGetter.md), BlueprintPure (Specifier/UFUNCTION/BlueprintPure.md)		
FUNC_EditorOnly	Trait	0x20000000	function can only be called from an editor script.			
FUNC_Const	Trait	0x40000000	function can be called from blueprint code, and only reads state (never writes state)			
FUNC_NetValidate	Network	0x80000000	function must supply a _Validate implementation	WithValidation (Specifier/UFUNCTION/WithValidation.md)		

PropertyFlags :

Name	Feature	Value	Description	UPARAM	UPROPERTY
CPF_Edit	Editor	0x0000000000000001	Property is user-settable in the editor.		EditAnywhere (Specifier/UPROPERTY/EditAnywhere.md), EditDefaultsOnly (Specifier/UPROPERTY/EditDefaultsOnly.md), EditInstanceOnly (Specifier/UPROPERTY/EditInstanceOnly.md), VisibleAnywhere (Specifier/UPROPERTY/VisibleAnywhere.md), VisibleDefaultsOnly (Specifier/UPROPERTY/VisibleDefaultsOnly.md), VisibleInstanceOnly (Specifier/UPROPERTY/VisibleInstanceOnly.md), Interp (Specifier/UPROPERTY/Interp.md)
CPF_ConstParm	Trait	0x0000000000000002	This is a constant function parameter	Const (Specifier/UPARAM/Const.md)	
CPF_BlueprintVisible	Blueprint	0x0000000000000004	This property can be read by blueprint code		BlueprintReadWrite (Specifier/UPROPERTY/BlueprintReadWrite.md), BlueprintReadOnly (Specifier/UPROPERTY/BlueprintReadOnly.md), BlueprintSetter (Specifier/UPROPERTY/BlueprintSetter.md), BlueprintGetter (Specifier/UPROPERTY/BlueprintGetter.md), Interp (Specifier/UPROPERTY/Interp.md)
CPF_ExportObject	Serialization	0x0000000000000008	Object can be exported with actor.		Instanced (Specifier/UPROPERTY/Instanced.md), Export (Specifier/UPROPERTY/Export.md)
CPF_BlueprintReadOnly	Blueprint	0x0000000000000010	This property cannot be modified by blueprint code		BlueprintReadOnly (Specifier/UPROPERTY/BlueprintReadOnly.md), BlueprintGetter (Specifier/UPROPERTY/BlueprintGetter.md)
CPF_Net	Network	0x0000000000000020	Property is relevant to network replication.		Replicated (Specifier/UPROPERTY/Replicated.md), ReplicatedUsing (Specifier/UPROPERTY/ReplicatedUsing.md)
CPF_EditFixedSize	Editor	0x0000000000000040	Indicates that elements of an array can be modified, but its size cannot be changed.		EditFixedSize (Specifier/UPROPERTY/EditFixedSize.md)
CPF_Parm	Function	0x0000000000000080	Function/When call parameter.		
CPF_OutParm	Function	0x0000000000000100	Value is copied out after function call.		
CPF_ZeroConstructor	Trait	0x0000000000000200	memset is fine for construction		
CPF_ReturnParm	Function	0x0000000000000400	Return value.		
CPF_DisableEditOnTemplate	Editor	0x0000000000000800	Disable editing of this property on an archetype/sub-blueprint		EditInstanceOnly (Specifier/UPROPERTY/EditInstanceOnly.md), VisibleInstanceOnly (Specifier/UPROPERTY/VisibleInstanceOnly.md)
CPF_NONNULLable	Trait	0x0000000000001000	Object property can never be null		
CPF_Transient	Serialization	0x0000000000002000	Property is transient: shouldn't be saved or loaded, except for Blueprint CDOs.		Transient (Specifier/UPROPERTY/Transient.md)
CPF_Config	Config	0x0000000000004000	Property should be loaded/saved as permanent profile.		Config (Specifier/UPROPERTY/Config.md)
CPF_RequiredParm	Editor	0x0000000000008000	Parameter must be linked explicitly in blueprint. Leaving the parameter out results in a compile error.	Required (Specifier/UPARAM/Required.md)	
CPF_DisableEditOnInstance	Editor	0x00000000000010000	Disable editing on an instance of this class		EditDefaultsOnly (Specifier/UPROPERTY/EditDefaultsOnly.md), VisibleDefaultsOnly (Specifier/UPROPERTY/VisibleDefaultsOnly.md)
CPF_EditConst	Editor	0x0000000000020000	Property is uneditable in the editor.		VisibleAnywhere (Specifier/UPROPERTY/VisibleAnywhere.md)
CPF_GlobalConfig	Config	0x0000000000040000	Load config from base class, not subclass.		GlobalConfig (Specifier/UPROPERTY/GlobalConfig.md)
CPF_InstancedReference	Trait	0x0000000000080000	Property is a component references.		Instanced (Specifier/UPROPERTY/Instanced.md)
CPF_DuplicateTransient	Serialization	0x0000000000200000	Property should always be reset to the default value during any type of duplication (copy/paste, binary duplication, etc.)		DuplicateTransient (Specifier/UPROPERTY/DuplicateTransient.md)
CPF_SaveGame	Serialization	0x0000000010000000	Property should be serialized for save games, this is only checked for game-specific archives with ArlsSaveGame		

Name	Feature	Value	Description	UPARAM	UPROPERTY
CPF_NoClear	Editor	0x0000000000000000	Hide clear button.		NoClear (Specifier/UPROPERTY/NoClear.md)
CPF_ReferenceParm	Function	0x0000000000000000	Value is passed by reference; CPF_OutParam and CPF_Param should also be set.	ref (Specifier/UPARAM/ref.md)	
CPF_BlueprintAssignable	Blueprint	0x0000000000000000	MC Delegates only. Property should be exposed for assigning in blueprint code		BlueprintAssignable (Specifier/UPROPERTY/BlueprintAssignable.md)
CPF_Deprecated	Trait	0x0000000000000000	Property is deprecated. Read it from an archive, but don't save it.		
CPF_IsPlainOldData	Trait	0x0000000040000000	If this is set, then the property can be memcopied instead of CopyCompleteValue / CopySingleValue		
CPF_RepSkip	Network	0x0000000080000000	Not replicated. For non replicated properties in replicated structs	NotReplicated (Specifier/UPARAM/NotReplicated.md)	NotReplicated (Specifier/UPROPERTY/NotReplicated.md)
CPF_RepNotify	Network	0x0000000010000000	Notify actors when a property is replicated		ReplicatedUsing (Specifier/UPROPERTY/ReplicatedUsing.md)
CPF_Interp	Editor	0x0000000020000000	interpolatable property for use with cinematics		Interp (Specifier/UPROPERTY/Interp.md)
CPF_NonTransactional	Editor	0x0000000040000000	Property isn't transacted		NonTransactional (Specifier/UPROPERTY/NonTransactional.md)
CPF_EditorOnly	Editor	0x0000000080000000	Property should only be loaded in the editor		
CPF_NoDestructor	Trait	0x0000001000000000	No destructor		
CPF_AutoWeak	Trait	0x0000004000000000	Only used for weak pointers, means the export type is autoweak		
CPF_ContainsInstancedReference	Trait	0x0000008000000000	Property contains component references.		
CPF_AssetRegistrySearchable	Editor	0x0000010000000000	asset instances will add properties with this flag to the asset registry automatically		AssetRegistrySearchable (Specifier/UPROPERTY/AssetRegistrySearchable.md)
CPF_SimpleDisplay	Editor	0x0000020000000000	The property is visible by default in the editor details view		SimpleDisplay (Specifier/UPROPERTY/SimpleDisplay.md)
CPF_AdvancedDisplay	Editor	0x0000040000000000	The property is advanced and not visible by default in the editor details view		AdvancedDisplay (Specifier/UPROPERTY/AdvancedDisplay.md)
CPF_Protected	Editor	0x0000080000000000	property is protected from the perspective of script		
CPF_BlueprintCallable	Blueprint	0x0000100000000000	MC Delegates only. Property should be exposed for calling in blueprint code		BlueprintCallable (Specifier/UPROPERTY/BlueprintCallable.md)
CPF_BlueprintAuthorityOnly	Network	0x0000200000000000	MC Delegates only. This delegate accepts (only in blueprint) only events with BlueprintAuthorityOnly.		BlueprintAuthorityOnly (Specifier/UPROPERTY/BlueprintAuthorityOnly.md)
CPF_TextExportTransient	Serialization	0x0000400000000000	Property shouldn't be exported to text format (e.g. copy/paste)		TextExportTransient (Specifier/UPROPERTY/TextExportTransient.md)
CPF_NonPIEDuplicateTransient	Serialization	0x0000800000000000	Property should only be copied in PIE		NonPIEDuplicateTransient (Specifier/UPROPERTY/NonPIEDuplicateTransient.md)
CPF_ExposeOnSpawn	Trait	0x0001000000000000	Property is exposed on spawn		
CPF_PersistentInstance	Serialization	0x0002000000000000	A object referenced by the property is duplicated like a component. (Each actor should have an own instance.)		Instanced (Specifier/UPROPERTY/Instanced.md)
CPF_UObjectWrapper	Trait	0x0004000000000000	Property was parsed as a wrapper class like TSubclassOf, FScriptInterface etc., rather than a USomething*		
CPF_HasGetValueTypeHash	Trait	0x0008000000000000	This property can generate a meaningful hash value.		
CPF_NativeAccessSpecifierPublic	Trait	0x0010000000000000	Public native access specifier		
CPF_NativeAccessSpecifierProtected	Trait	0x0020000000000000	Protected native access specifier		
CPF_NativeAccessSpecifierPrivate	Trait	0x0040000000000000	Private native access specifier		
CPF_SkipSerialization	Serialization	0x0080000000000000	Property shouldn't be serialized, can still be exported to text		SkipSerialization (Specifier/UPROPERTY/SkipSerialization.md)

ChildCannotTick

- 功能描述:** 用于Actor或ActorComponent子类，标记允许其蓝图子类不可以接受响应Tick事件，哪怕父类可以Tick
- 使用位置:** UCLASS

- **元数据类型:** bool
- **限制类型:** Actor类
- **关联项:** ChildCanTick
- **常用程度:** ★★★

ChildCanTick

- **功能描述:** 标记允许其蓝图子类可以接受响应Tick事件
- **使用位置:** UCLASS
- **引擎模块:** Actor
- **元数据类型:** bool
- **限制类型:** Actor或ActorComponent子类
- **关联项:** ChildCannotTick
- **常用程度:** ★★★

要在蓝图中重载Tick事件函数并只会在编译的时候触发判断。

```
//(BlueprintType = true, ChildCannotTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCanTick))
class INSIDER_API AMyActor_ChildCanTick : public AActor
{
    GENERATED_BODY()

public:
    AMyActor_ChildCanTick()
    {
        PrimaryActorTick.bCanEverTick = false;
    }
};

//(BlueprintType = true, ChildCanTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCanTick))
class INSIDER_API UMyActorComponent_ChildCanTick : public UActorComponent
{
    GENERATED_BODY()

public:
};

//(BlueprintType = true, ChildCannotTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCannotTick))
class INSIDER_API AMyActor_ChildCannotTick : public AActor
{
    GENERATED_BODY()

public:
};
```

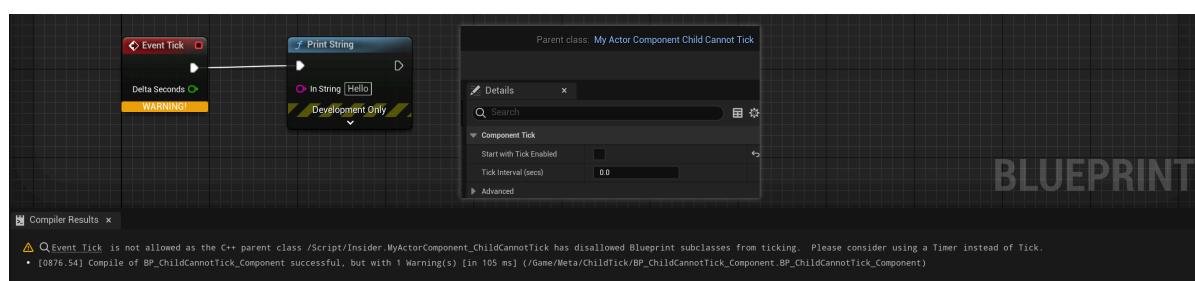
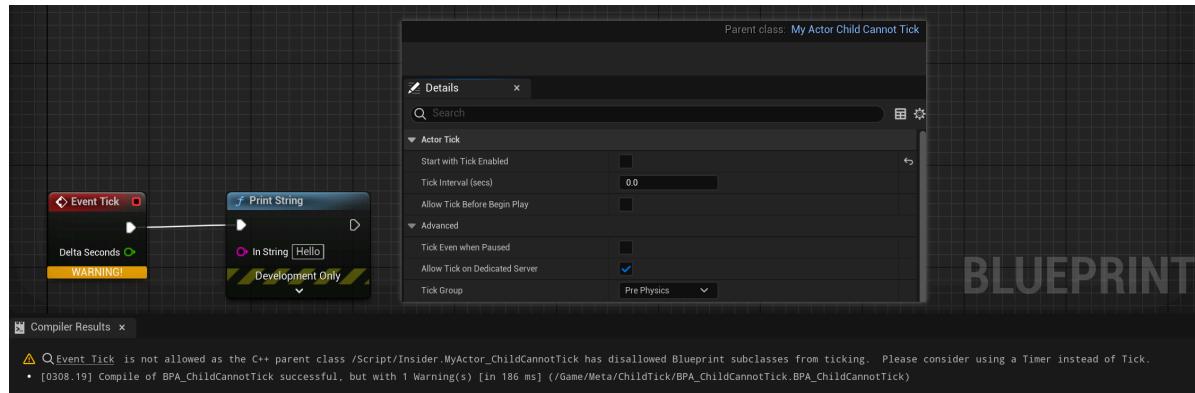
```

//(BlueprintType = true, ChildCannotTick = , IncludePath =
class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCannotTick))
class INSIDER_API UMyActorComponent_ChildCannotTick : public UActorComponent
{
    GENERATED_BODY()
public:
};

```

蓝图Actor或ActorComponent里测试：

也注意到这个判断跟蓝图中是否开启Tick并没有关系。



而AMyActor_ChildCanTick类里虽然已经手动关闭了PrimaryActorTick.bCanEverTick，但是在子类里依然可以正常的Tick（在编译的时候内部可以正常的再重新开启bCanEverTick）。



源码里判断的逻辑：

开启bCanEverTick=true的条件有3，一是EngineSettings->bCanBlueprintsTickByDefault，二是父类是AActor或UActorComponent本身，三是C++基类上有ChildCanTick的标记。

```

void FKismetCompilerContext::SetCanEverTick() const
{
    // RECEIVE TICK
    if (!TickFunction->bCanEverTick)
    {

```

```

    // Make sure that both AActor and UActorComponent have the same name for
    their tick method
    static FName ReceiveTickName(GET_FUNCTION_NAME_CHECKED(AActor, ReceiveTick));
    static FName
ComponentReceiveTickName(GET_FUNCTION_NAME_CHECKED(UActorComponent,
ReceiveTick));

    if (const UFunction* ReceiveTickEvent =
FKismetCompilerUtilities::FindOverriddenImplementableEvent(ReceiveTickName,
NewClass))
    {
        // We have a tick node, but are we allowed to?

        const UEngine* EngineSettings = GetDefault<UEngine>();
        const bool bAllowTickingByDefault = EngineSettings-
>bCanBlueprintsTickByDefault;

        const UClass* FirstNativeClass =
FBlueprintEditorUtils::FindFirstNativeClass(NewClass);
        const bool bHasCanTickMetadata = (FirstNativeClass != nullptr) &&
FirstNativeClass->HasMetaData(FBlueprintMetadata::MD_ChildCanTick);
        const bool bHasCannotTickMetadata = (FirstNativeClass != nullptr) &&
FirstNativeClass->HasMetaData(FBlueprintMetadata::MD_ChildCannotTick);
        const bool bHasUniversalParent = (FirstNativeClass != nullptr) &&
((AActor::StaticClass() == FirstNativeClass) || (UActorComponent::StaticClass()
== FirstNativeClass));

        if (bHasCanTickMetadata && bHasCannotTickMetadata)
        {
            // User error: The C++ class has conflicting metadata
            const FString ConlictingMetadataWarning = FText::Format(
                LOCTEXT("HasBothCanAndCannotMetadataFmt", "Native class %s has
both '{0}' and '{1}' metadata specified, they are mutually exclusive and '{1}' will
win."),
                FText::FromString(FirstNativeClass->GetPathName()),
                FText::FromName(FBlueprintMetadata::MD_ChildCanTick),
                FText::FromName(FBlueprintMetadata::MD_ChildCannotTick)
            ).ToString();
            MessageLog.Warning(*ConlictingMetadataWarning);
        }

        if (bHasCannotTickMetadata)
        {
            // This could only happen if someone adds bad metadata to AActor or
UActorComponent directly
            check(!bHasUniversalParent);

            // Parent class has forbidden us to tick
            const FString NativeClassSaidNo = FText::Format(
                LOCTEXT("NativeClassProhibitsTickingFmt", "@@ is not allowed as
the C++ parent class {0} has disallowed Blueprint subclasses from ticking.
Please consider using a Timer instead of Tick."),
                FText::FromString(FirstNativeClass->GetPathName())
            ).ToString();
            MessageLog.Warning(*NativeClassSaidNo,
FindLocalEntryPoint(ReceiveTickEvent));
        }
    }
}

```

```

    }
    else
    {
        if (bAllowTickingByDefault || bHasUniversalParent || bHasCanTickMetadata)
        {
            // We're allowed to tick for one reason or another
            TickFunction->bCanEverTick = true;
        }
        else
        {
            // Nothing allowing us to tick
            const FString ReceiveTickEventWarning = FText::Format(
                LOCTEXT("ReceiveTick_CanNeverTickFmt", "@@ is not allowed for
Blueprints based on the C++ parent class {0}, so it will never Tick!"),
                FText::FromString(FirstNativeClass ? *FirstNativeClass-
>GetPathName() : TEXT("<null>"))
            ).ToString();
            MessageLog.Warning(*ReceiveTickEventWarning,
                FindLocalEntryPoint(ReceiveTickEvent));
        }
    }
}
}

```

AllowedParamType

- **使用位置:** UFUNCTION
- **引擎模块:** AnimationGraph
- **元数据类型:** string="abc"

```

// Sets a parameter's value in the supplied scope.
// @param Scope Scopes corresponding to an existing scope in a schedule, or
// "None". Passing "None" will apply the parameter to the whole schedule.
// @param Ordering where to apply the parameter in relation to the supplied
// scope. Ignored for scope "None".
// @param Name The name of the parameter to apply
// @param Value The value to set the parameter to
UFUNCTION(BlueprintCallable, Category = "AnimNext", CustomThunk, meta =
(CustomStructureParam = Value, UnsafeDuringActorConstruction))
void SetParameterInScope(UPARAM(meta = (CustomWidget = "ParamName",
AllowedParamType = "FAnimNextScope")) FName Scope,
EAnimNextParameterScopeOrdering Ordering, UPARAM(meta = (CustomWidget =
"ParamName")) FName Name, int32 Value);

```

查了一下，只在AnimNext中用到。

AlwaysAsPin

- 功能描述：**在动画蓝图中使得动画节点的某个属性总是暴露出来成为引脚
- 使用位置：** UPROPERTY
- 引擎模块：** Pin
- 元数据类型：** bool
- 限制类型：** FAnimNode_Base
- 关联项：** PinShownByDefault
- 常用程度：** ★★★

和PinShownByDefault的区别是前者会导致只能一直显示为引脚。而PinShownByDefault默认显示为引脚，当之后也可以改变。

测试代码：

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (AlwaysAsPin))
    int32 MyInt_AlwaysAsPin = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (NeverAsPin))
    int32 MyInt_NeverAsPin = 123;
};

```

测试效果：



原理：

根据源码里的逻辑可见，`bAlwaysShow` 会导致`bShowPin`，和`PinShownByDefault`的区别是前者会导致只能一直显示为引脚。而`PinShownByDefault`默认显示为引脚，当之后也可以改变。

```
void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();

    // Determine if this is a pose or array of poses
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ? ArrayProp->Inner : TestProperty);
    const bool bIsPoseInput = (StructProp && StructProp->Struct->IsChildOf(FPoseLinkBase::StaticStruct()));

    // @TODO: Error if they specified two or more of these flags
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
        || bIsPoseInput;
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema->NAME_PinShownByDefault);
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema->NAME_PinHiddenByDefault);
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema->NAME_CustomizeProperty);
    const bool bCanTreatPropertyAsOptional =
        CanTreatPropertyAsOptional(TestProperty);

    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
        (bOptional_ShowByDefault || bOptional_HideByDefault);
    Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
    Record.bPropertyIsCustomized = bPropertyIsCustomized;
}
```

AnimBlueprintFunction

- **功能描述：**标明是动画蓝图里的内部纯存根函数，只在动画蓝图编译时设置
- **使用位置：** UFUNCTION
- **引擎模块：** AnimationGraph
- **元数据类型：** bool
- **限制类型：** Anim BP

只是在内部使用，在动画蓝图编译的时候设置。但是没有在代码里显式的编写。

AnimGetter

- **功能描述:** 指定UAnimInstance及子类的该函数成为一个AnimGetter函数。
- **使用位置:** UFUNCTION
- **引擎模块:** AnimationGraph
- **元数据类型:** bool
- **限制类型:** UAnimInstance及子类的函数
- **关联项:** GetterContext
- **常用程度:** ★★★

指定UAnimInstance及子类的该函数成为一个AnimGetter函数。

- 在一些情况下会继承UAnimInstance创建自己的动画蓝图子类，然后里面可以自己做一些优化，或者添加一些自己的功能函数。
- 所谓的AnimGetter，其实就是会被UK2Node_AnimGetter识别并包装成该蓝图节点的函数。识别的范围是在UAnimInstance及子类（就是动画蓝图）的C++函数。
- AnimGetter还有两个额外功能：一是会自动根据当前上下文填充函数里的AssetPlayerIndex, MachineIndex, StateIndex, TransitionIndex和参数。二是会根据GetterContext把该函数限定只能在某些蓝图里调用。普通的蓝图函数不具有这些便利的功能和检查，用起来就不够智能。
- 要成为AnimGetter还必须具有：
 - AnimGetter，自然不必说
 - BlueprintThreadSafe，才能在动画蓝图里调用，多线程安全
 - BlueprintPure，成为一个存获取值的函数
 - BlueprintInternalUseOnly = "true"，避免再生成一个默认的蓝图节点，只用UK2Node_AnimGetter包装而成的那个。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyAnimInstance :public UAnimInstance
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintInternalUseOnly = "true", AnimGetter, BlueprintThreadsafe))
        float MyGetAnimationLength_AnimGetter(int32 AssetPlayerIndex);

    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintThreadsafe))
        float MyGetAnimationLength(int32 AssetPlayerIndex);
public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintInternaluseOnly = "true", AnimGetter, BlueprintThreadSafe))
        float MyGetStateweight_AnimGetter(int32 MachineIndex, int32 StateIndex);

    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintThreadSafe))
        float MyGetStateweight(int32 MachineIndex, int32 StateIndex);
```

```

public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
(BlueprintInternalUseOnly = "true", AnimGetter, BlueprintThreadSafe))
        float MyGetTransitionTimeElapsed_AnimGetter(int32 MachineIndex, int32
TransitionIndex);

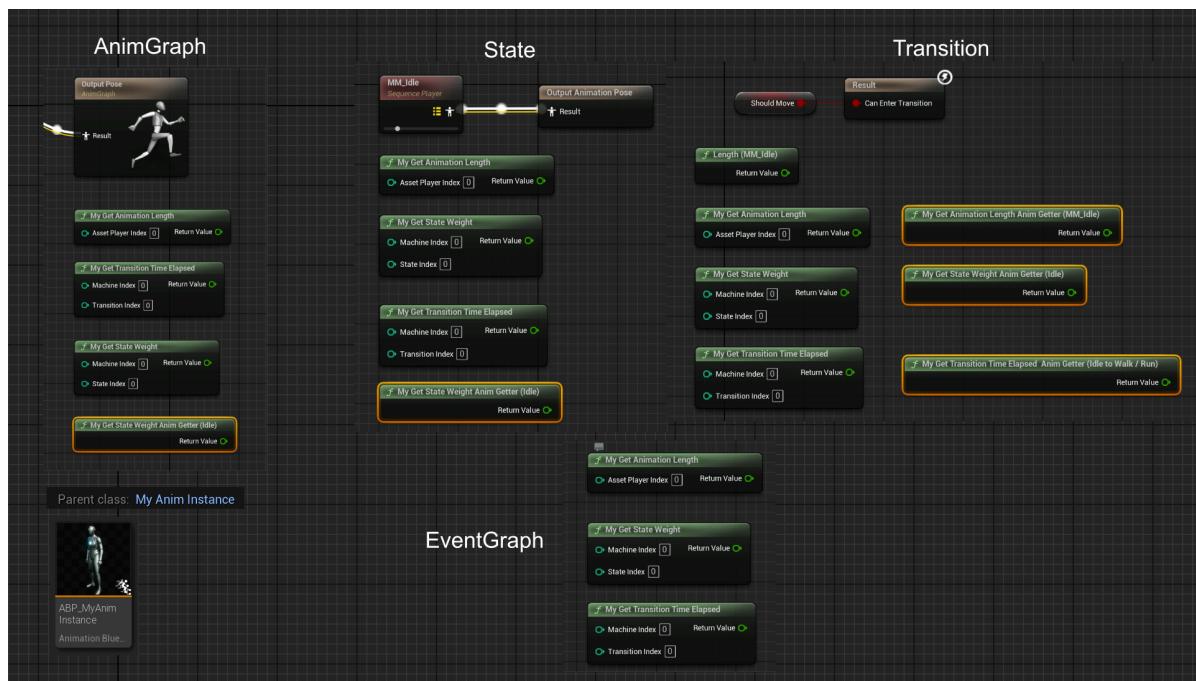
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
(BlueprintThreadSafe))
        float MyGetTransitionTimeElapsed(int32 MachineIndex, int32 TransitionIndex);
};

```

测试效果：

分别定义使用了AssetPlayerIndex, MachineIndex, StateIndex, TransitionIndex的AnimGetter函数以及普通蓝图函数作为对比。分别查看在动画蓝图里几个作用域里的用法。

- 可见在不管什么作用域，普通蓝图函数都可以调用（毕竟没有做Context的检查）。另外AssetPlayerIndex等参数都没有被自动填充，这几乎是没法用的，因为用户其实并不太懂如何去手填这些Index，最好是交给编译器来填充。
- 图里高亮的是可以调用的AnimGetter函数。细看的话，可以分析发现规则是只有能正确填充AssetPlayerIndex等参数的才能调用。因此在Transition里能调用的最多，因为这个时候最叶子节点，有动画，又有状态机和Transition节点。



原理：

分析函数上的AnimGetter标记并且生成蓝图节点的功能基本都在UK2Node_AnimGetter这个类里。大家可自行查看。

```

void UK2Node_AnimGetter::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    TArray<UFunction*> AnimGetters;
    for(TFieldIterator<UFunction> FuncIter(BPClass) ; FuncIter ;
++FuncIter)
    {
        UFunction* Func = *FuncIter;

        if(Func->HasMetaData(TEXT("AnimGetter")) && Func-
>HasAnyFunctionFlags(FUNC_Native))
        {
            AnimGetters.Add(Func);
        }
    }
}

```

AnimNotifyBoneName

- 功能描述:** 使得UAnimNotify或UAnimNotifyState下的FName属性作为BoneName的作用。
- 使用位置:** UPROPERTY
- 引擎模块:** AnimationGraph
- 元数据类型:** bool
- 限制类型:** UAnimNotify或UAnimNotifyState子类下的FName属性
- 常用程度:** ★★

使得UAnimNotify或UAnimNotifyState下的FName属性作为BoneName的作用。

在动画通知的时候，也常常需要一个传递骨骼名字参数，用普通的字符串参数显然不够定制化。因此给一个FName属性标上AnimNotifyBoneName就可以在配合的细节面板定制化里为它创建专门的更便于填写BoneName的UI。

源码中例子：

```

UCLASS(const, hidecategories = Object, collapsecategories, meta = (DisplayName =
"Play Niagara Particle Effect"), MinimalAPI)
class UAnimNotify_PlayNiagaraEffect : public UAnimNotify
{
    // SocketName to attach to
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AnimNotify", meta =
(AnimNotifyBoneName = "true"))
    FName SocketName;
}

UCLASS(Blueprintable, meta = (DisplayName = "Timed Niagara Effect"), MinimalAPI)
class UAnimNotifyState_TimedNiagaraEffect : public UAnimNotifyState
{
    // The socket within our mesh component to attach to when we spawn the
    Niagara component
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NiagaraSystem, meta =
(ToolTip = "The socket or bone to attach the system to", AnimNotifyBoneName =
"true"))
}
```

```
    FName SocketName;  
}
```

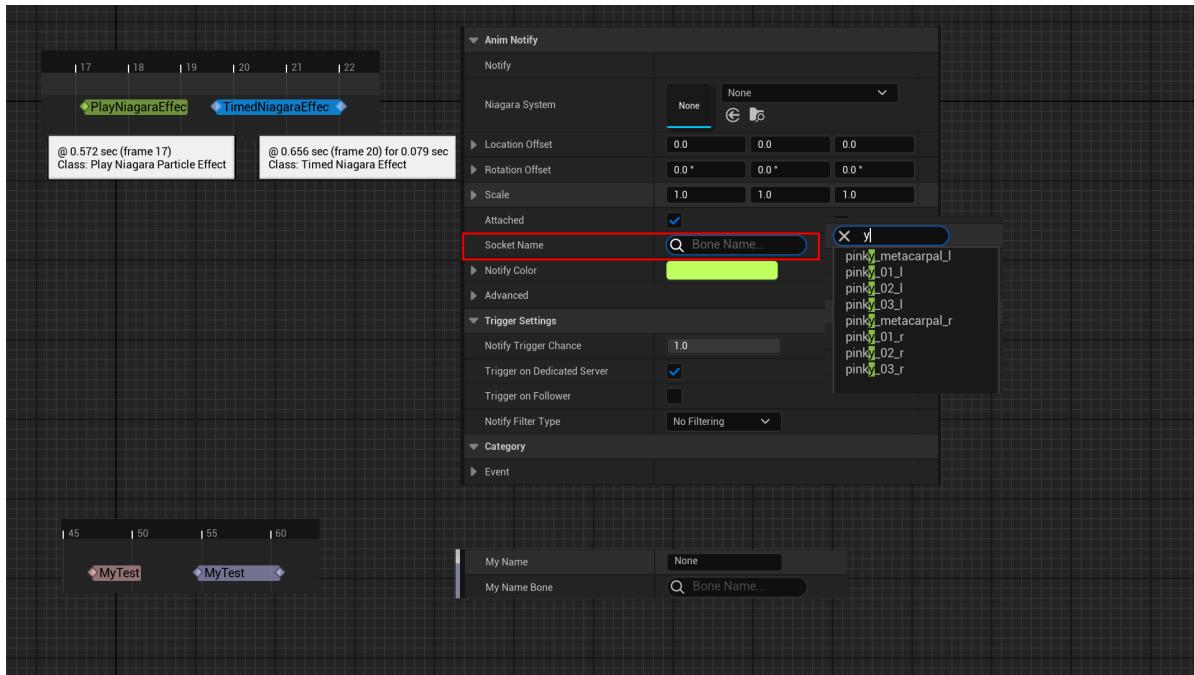
测试代码：

```
UCLASS(BlueprintType)  
class INSIDER_API UAnimNotify_MyTest:public UAnimNotify  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    FName MyName;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(AnimNotifyBoneName="true"))  
    FName MyName_Bone;  
};  
  
UCLASS(BlueprintType)  
class INSIDER_API UAnimNotifyState_MyTest:public UAnimNotifyState  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    FName MyName;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(AnimNotifyBoneName="true"))  
    FName MyName_Bone;  
};
```

测试效果：

在一个动画序列里加上动画通知，可以加两种：UAnimNotify或UAnimNotifyState。首先引擎里的自带例子UAnimNotify_PlayNiagaraEffect 和UAnimNotifyState_TimedNiagaraEffect，可以看见在右侧的细节面板上的SocketName的UI不是普通的字符串。

我们自己定义的MyBoneName的动画通知，也可以达成同样的效果。MyName_Bone因为加了AnimNotifyBoneName，就和普通的MyName不一样了。



原理：

在定制化的时候，根据AnimNotify下的属性是否有这个标记，生成专门的UI。

```
bool FAnimNotifyDetails::CustomizeProperty(IDetailCategoryBuilder&
CategoryBuilder, UObject* Notify, TSharedPtr<IPropertyHandle> Property)
{
    else if (InPropertyParams->GetBoolMetaData(TEXT("AnimNotifyBoneName")))
    {
        // Convert this property to a bone name property
        AddBoneNameProperty(CategoryBuilder, Notify, InPropertyParams);
    }

    if (bIsBoneName)
    {
        AddBoneNameProperty(CategoryBuilder, Notify,PropertyParams);
        return true;
    }
}
```

AnimNotifyExpand

- 功能描述：**使得UAnimNotify或UAnimNotifyState下的属性直接展开到细节面板里。
- 使用位置：**UPROPERTY
- 引擎模块：**AnimationGraph
- 元数据类型：**bool
- 限制类型：**UAnimNotify或UAnimNotifyState子类下的FName属性

使得UAnimNotify或UAnimNotifyState下的属性直接展开到细节面板里。

在源码里也没有找到应用的例子。

原理：

看源码，里面写死了只对有限的引擎自带的几个类起效，因此自己的测试代码并不能生效。

这种写法确实不应该这么写死，希望以后改进吧。到时应该就有源码里的例子了。

```
PropertyModule.RegisterCustomClassLayout( "EditorNotifyObject",
FOnGetDetailCustomizationInstance::CreateStatic(&FAnimNotifyDetails::MakeInstance
));

bool FAnimNotifyDetails::CustomizeProperty(IDetailCategoryBuilder&
CategoryBuilder, UObject* Notify, TSharedPtr<IPROPERTYHandle> Property)
{
    if(Notify && Notify->GetClass() && Property->IsValidHandle())
    {
        FString ClassName = Notify->GetClass()->GetName();
        FStringPropertyName = Property->GetProperty()->GetName();
        bool bIsBoneName = Property->GetBoolMetaData(TEXT("AnimNotifyBoneName"));

        if(ClassName.Find(TEXT("AnimNotify_PlayParticleEffect")) != INDEX_NONE &&
PropertyName == TEXT("SocketName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if(ClassName.Find(TEXT("AnimNotifyState_TimedParticleEffect")) != INDEX_NONE && PropertyName == TEXT("SocketName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if(ClassName.Find(TEXT("AnimNotify_PlaySound")) != INDEX_NONE && PropertyName == TEXT("AttachName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if (ClassName.Find(TEXT("_SoundLibrary")) != INDEX_NONE && PropertyName == TEXT("SoundContext"))
        {
            CategoryBuilder.AddProperty(Property);
            FixBoneNamePropertyRecurse(Property);
            return true;
        }
        else if (ClassName.Find(TEXT("AnimNotifyState_Trail")) != INDEX_NONE)
        {
            if(PropertyName == TEXT("FirstSocketName") || PropertyName ==
TEXT("SecondSocketName"))
            {
                AddBoneNameProperty(CategoryBuilder, Notify, Property);
                return true;
            }
            else if(PropertyName == TEXT("WidthScaleCurve"))
            {

```

```

        AddCurveNameProperty(CategoryBuilder, Notify, Property);
        return true;
    }
}
else if (bIsBoneName)
{
    AddBoneNameProperty(CategoryBuilder, Notify, Property);
    return true;
}
}
}

```

BlueprintCompilerGeneratedDefaults

- 功能描述:** 指定该属性的值是编译器生成的，因此在编译后无需复制，可以加速一些编译性能。
- 使用位置:** UPROPERTY
- 引擎模块:** AnimationGraph
- 元数据类型:** bool
- 限制类型:** FAnimNode里的属性

指定该属性的值是编译器生成的，因此在编译后无需复制，可以加速一些编译性能。

在源码里寻找例子，可以看到基本是在FAnimNode下的属性在使用。在动画蓝图编译后，会调用 UEngine::CopyPropertiesForUnrelatedObjects来把之前编译的旧对象里的值复制到新对象，其中 FCopyPropertiesForUnrelatedObjectsParams的bSkipCompilerGeneratedDefaults决定是否要赋值这个属性的值。如果有标上这个值，就说明不要复制。这个值会在别的地方由编译器来填充值。

UAnimGraphNode_Base::OnProcessDuringCompilation函数就是编译后回调的函数。

测试代码：

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDER_API FAnimNode_MyCompilerDefaults : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Links)
    FPoseLink Source;
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CompilerDefaultsTest)
    FString MyString;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CompilerDefaultsTest,
meta = (BlueprintCompilerGeneratedDefaults))
    FString MyString_CompilerDefaults;
};

UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyCompilerDefaults : public
UAnimGraphNode_Base
{
    GENERATED_UCLASS_BODY()

```

```

public:
~UAnimGraphNode_MyCompilerDefaults();

UPROPERTY(EditAnywhere, Category = Settings)
FAnimNode_MyCompilerDefaults Node;

protected:
    virtual void OnProcessDuringCompilation(IAnimBlueprintCompilationContext&
InCompilationContext, IAnimBlueprintGeneratedClassCompiledData& OutCompiledData)
{
    Node.MyString=TEXT("This is generated by compiler.");
    Node.MyString_CompilerDefaults=TEXT("This is generated by compiler.");
}
};


```

测试效果：

这个因为是序列化的过程，因此并没有直观的示意图。

可验证的结果是在FCPFUOWriter::ShouldSkipProperty可以见到MyString_CompilerDefaults属性跳过了复制。

原理：

蓝图编译的过程，核心思想是生成一个新的Graph对象，然后把上一次编译的结果对象里的属性和只对对象复制到这个新的对象里去。这一步操作是用UEngine::CopyPropertiesForUnrelatedObjects来完成的，再内部会继续用FCPFUOWriter::ShouldSkipProperty来判断是否该复制某个属性。而对于一些属性的值只是由编译器生成的临时值，反正下一次编译也会再生成，因此就不需要复制了，标上之后可以略微加速一些性能，虽然其实也不多。

```

void
UK2Node_PropertyAccess::CreateClassVariablesFromBlueprint(IAnimBlueprintVariableC
reationContext& InCreationContext)
{
    GeneratedPropertyName = NAME_None;

    const bool bRequiresCachedVariable = !bWasResolvedThreadSafe ||
UAnimBlueprintExtension_PropertyAccess::ContextRequiresCachedVariable(ContextId);

    if(ResolvedPinType != FEdGraphPinType() && ResolvedPinType.PinCategory !=
UEdGraphSchema_K2::PC_Wildcard && bRequiresCachedVariable)
    {
        // Create internal generated destination property (only if we were not
        // identified as thread safe)
        if(FProperty* DestProperty = InCreationContext.CreateUniqueVariable(this,
ResolvedPinType))
        {
            GeneratedPropertyName = DestProperty->GetFName();
            DestProperty->SetMetaData(TEXT("BlueprintCompilerGeneratedDefaults"),
TEXT("true"));
        }
    }
}


```

```

/* Serializes and stores property data from a specified 'source' object. Only
stores data compatible with a target destination object. */
struct FCPFUOWriter : public FObjectWriter, public FCPFUOArchive
{
    #if WITH_EDITOR
    virtual bool ShouldSkipProperty(const class FProperty* InProperty) const
override
    {
        return (bSkipCompilerGeneratedDefaults && InProperty->HasMetaData(BlueprintCompilerGeneratedDefaultsName));
    }
#endif
};

```

CustomizeProperty

- 功能描述:** 使用在FAnimNode的成员属性上，告诉编辑器不要为它生成默认Details面板控件，后续会在DetailsCustomization里自定义创建相应的编辑控件。
- 使用位置:** UPROPERTY
- 引擎模块:** AnimationGraph
- 元数据类型:** bool
- 限制类型:** FAnimNode里的属性
- 常用程度:** ★

使用在FAnimNode的成员属性上，告诉编辑器不要为它生成默认Details面板控件，后续会在DetailsCustomization里自定义创建相应的编辑控件。

和AllowEditInlineCustomization的作用有点像，都只是做个标记提示编辑器会在别的地方进行自定义，不用为它生成默认Details面板控件。

源码中例子：

在源码里能见到挺多例子，常见的就是在AnimBP中的节点上的属性，其在细节面板需要专门的定制化编辑。最常见的例子是Slot这个节点，其SlotName只是个FString类型，但是在细节面板里显示的却是个ComboString。这是因为它标上了CustomizeProperty，告知默认的动画节点细节面板生成器*(FAnimGraphNodeDetails)先不要为这个属性创建编辑控件，之后会在自己的定制化(FAnimGraphNodeSlotDetails)里为SlotName再创建自定义UI。

```

struct FAnimNode_Slot : public FAnimNode_Base
{
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Settings, meta=
(CustomizeProperty))
    FName SlotName;
}

void FPersonaModule::CustomizeBlueprintEditorDetails(const TSharedRef<class
IDetailsView>& InDetailsView, FOnInvokeTab InOnInvokeTab)
{
    InDetailsView-
>RegisterInstancedCustomPropertyLayout(UAnimGraphNode_Slot::StaticClass(),

```

```

FOnGetDetailCustomizationInstance::CreateStatic(&FAnimGraphNodeSlotDetails::MakeInstance, InOnInvokeTab));

    InDetailsView->SetExtensionHandler(MakeShared<FAAnimGraphNodeBindingExtension>()
);
}

```

测试代码：

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDER_API FAnimNode_MyCustomProperty : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty)
        FString MyString_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty, meta = (CustomizeProperty))
        FString MyString_CustomizeProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty, meta = (CustomizeProperty))
        FString MyString_CustomizeProperty_Other;
};

UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyCustomProperty : public UAnimGraphNode_Base
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(EditAnywhere, Category = Settings)
        FAnimNode_MyCustomProperty Node;
};

//再创建一个定制化，生成自定义UI
void FMyAnimNode_MyCustomPropertyCustomization::CustomizeDetails(class IDetailLayoutBuilder& DetailBuilder)
{
    TSharedPtr<IPropertyHandle> PropertyHandle =
        DetailBuilder.GetProperty(TEXT("Node.MyString_CustomProperty"));

    //Just for test
    ComboListItems.Empty();
    ComboListItems.Add(MakeShareable(new FString(TEXT("First"))));
    ComboListItems.Add(MakeShareable(new FString(TEXT("Second"))));
    ComboListItems.Add(MakeShareable(new FString(TEXT("Third"))));
    const TSharedPtr< FString > ComboBoxSelectedItem = ComboListItems[0];

    IDetailCategoryBuilder& Group =
        DetailBuilder.EditCategory(TEXT("CustomProperty"));
    Group.AddCustomRow(INVTEXT("CustomProperty"))
        .NameContent()
        [

```

```

PropertyHandle->CreatePropertyNameWidget()
]
.valueContent()
[
SNew(STextComboBox)
.OptionsSource(&ComboListItems)
.InitiallySelectedItem(ComboBoxSelectedItem)
.ContentPadding(2, 4)
.ToolTipText(FText::FromString(*ComboBoxSelectedItem))
];
}

//注册定制化
PropertyModule.RegisterCustomClassLayout(TEXT("AnimGraphNode_MyCustomProperty"),
FOnGetDetailCustomizationInstance::CreateStatic(&FMyAnimNode_MyCustomPropertyCustomization::MakeInstance));

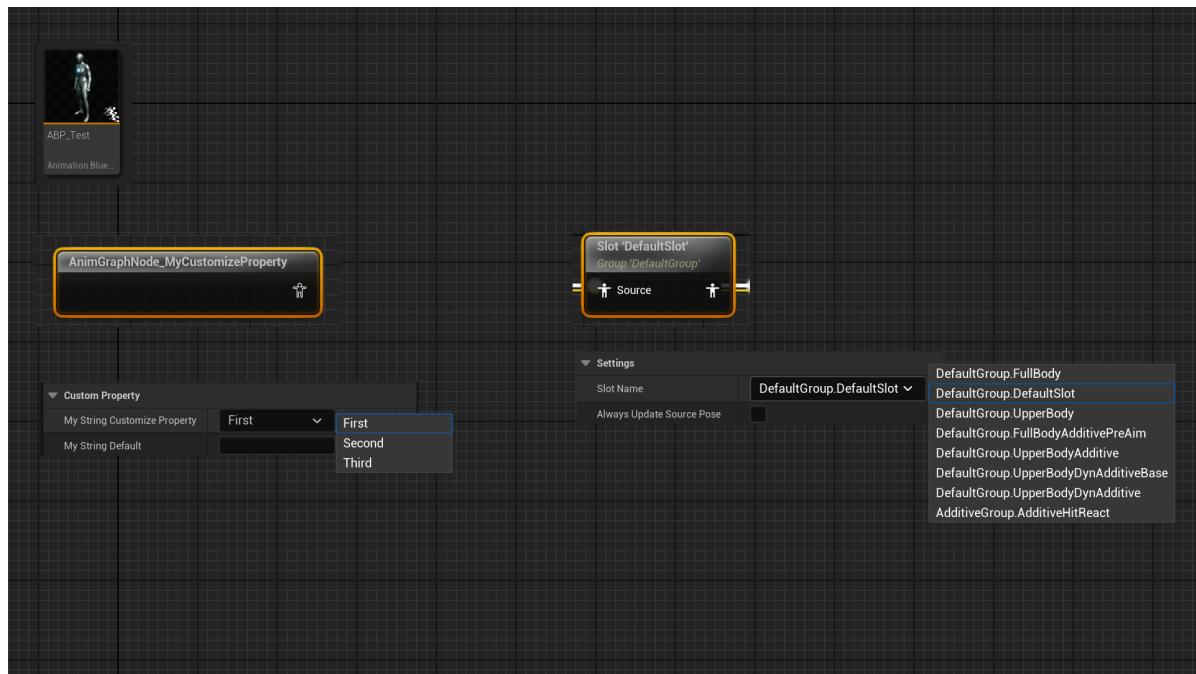
```

测试效果：

SlotName的效果如右侧所示。

我们自己模仿的例子可见MyString_Default依然只是个默认String，而MyString_CustomizeProperty则为它创建了自定义UI。

作为对比，MyString_CustomizeProperty_Other我们标上了CustomizeProperty但是没有为它创建UI，则没有显示出来，说明引擎默认的机制因此就把它的UI默认创建流程给跳过了。



原理：

CustomizeProperty其实会改变Pin的bPropertyIsCustomized 属性 (GetRecordDefaults中体现)，然后在创建流程的过程中不创建默认的widget，这个可见CustomizeDetails中的bPropertyIsCustomized判断得知。

```

void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{

```

```

const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();

// Determine if this is a pose or array of poses
FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ?
ArrayProp->Inner : TestProperty);
const bool bIsPoseInput = (StructProp && StructProp->Struct-
>IsChildOf(FPoseLinkBase::StaticStruct()));

//@TODO: Error if they specified two or more of these flags
const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
|| bIsPoseInput;
const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinShownByDefault);
const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinHiddenByDefault);
const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema-
>NAME_CustomizeProperty);
const bool bCanTreatPropertyAsOptional =
CanTreatPropertyAsOptional(TestProperty);

Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
(bOptional_ShowByDefault || bOptional_HideByDefault);
Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
Record.bPropertyIsCustomized = bPropertyIsCustomized;
}

//这个是在AnimBP中选中一个节点然后在右侧细节面板中的属性
void FAnimGraphNodeDetails::CustomizeDetails(class IDetailLayoutBuilder&
DetailBuilder)
{
    // sometimes because of order of customization
    // this gets called first for the node you'd like to customize
    // then the above statement won't work
    // so you can mark certain property to have meta data "CustomizeProperty"
    // which will trigger below statement
    if (OptionalPin.bPropertyIsCustomized)
    {
        continue;
    }
    TSharedRef<swidget> InternalCustomWidget =
CreatePropertyWidget(TargetProperty, TargetPropertyHandle.ToSharedRef(),
AnimGraphNode->GetClass());
}

```

CustomWidget

- **使用位置:** UFUNCTION, UPROPERTY
- **引擎模块:** AnimationGraph
- **元数据类型:** string="abc"

也可以放在属性上

```

// @param Scope Scopes corresponding to an existing scope in a schedule, or
// "None". Passing "None" will apply the parameter to the whole schedule.
// @param Ordering where to apply the parameter in relation to the supplied
// scope. Ignored for scope "None".
// @param Name The name of the parameter to apply
// @param Value The value to set the parameter to
UFUNCTION(BlueprintCallable, Category = "AnimNext", CustomThunk, meta =
(CustomStructureParam = Value, UnsafeDuringActorConstruction))
void SetParameterInScope(UPARAM(meta = (CustomWidget = "ParamName",
AllowedParamType = "FAnimNextScope")) FName Scope,
EAnimNextParameterScopeOrdering Ordering, UPARAM(meta = (CustomWidget =
"ParamName")) FName Name, int32 Value);

```

只在AnimNext和RigVM里用到。

FoldProperty

- 功能描述:** 在动画蓝图中使得动画节点的某个属性成为FoldProperty。
- 使用位置:** UPROPERTY
- 引擎模块:** AnimationGraph
- 元数据类型:** bool
- 限制类型:** FAnimNode_Base下的属性
- 常用程度:** ★

在动画蓝图中使得动画节点的某个属性成为FoldProperty。

- 在UI表现上PinHiddenByDefault也有同样的效果，但是FoldProperty在行为上有别的不同。
- 所谓FoldProperty，指的是这些属性往往被WITH_EDITORONLY_DATA包起来的。记录编辑器状况下的信息。比如FAnimNode_SequencePlayer下的PlayRate数据，其就是在编辑器状态的下数据。又或者只是动画蓝图本身的信息，不管动画蓝图的多少个实例，这些属性的值其实都是同样的。这些属性就适合成为FoldProperty。
- 这些属性需要在节点上编辑，但又不想暴露成引脚，因此就在形式上和PinHiddenByDefault一样放到右侧的细节面板里。

在FAnimNodeData* FAnimNode_Base::NodeData里存储着该动画节点的所有实例所用到的“Constant/Fold”属性信息。该动画蓝图在游戏里可能有多个实例，在这些实例之间都只存一份动画节点的常量信息，也只存一份FoldProperty的信息。因此用FoldProperty标记的属性的真实数据是存在TArray UAnimBlueprintGeneratedClass::AnimNodeData中的。存在Class中，其实就是类似CDO的意思了。这么做的显然好处之一是节省内存。

自然的，不同的存储方式，自然要采用不同的访问方式。因此这些FoldProperty都是采用GET_ANIM_NODE_DATA来访问该数据。

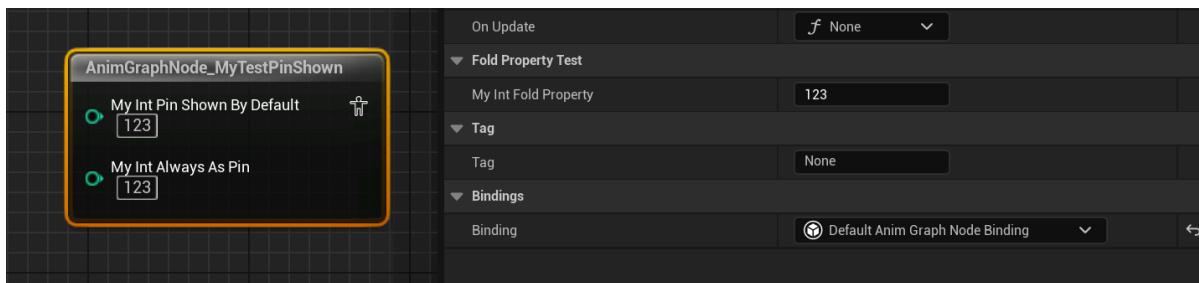
测试代码：

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FoldPropertyTest, meta
= (FoldProperty))
    int32 MyInt_FoldProperty = 123;

```

测试结果：



原理：

编译的时候会把该FoldProperty添加到FoldRecords里。如果这个属性不是动态，也没有暴露成引脚连接，则会被当作常量。

```
void FAnimBlueprintCompilerContext::GatherFoldRecordsForAnimationNode(const UScriptStruct* InNodeType, FStructProperty* InNodeProperty, UAnimGraphNode_Base* InVisualAnimNode)
{
    if(SubProperty->HasMetaData(NAME_FoldProperty))
    {
        // Add folding candidate
        AddFoldedPropertyRecord(InVisualAnimNode, InNodeProperty, SubProperty,
        bAllPinsExposed, !bAllPinsDisconnected, bAlwaysDynamic);
    }
}

void FAnimBlueprintCompilerContext::AddFoldedPropertyRecord(UAnimGraphNode_Base* InAnimGraphNode, FStructProperty* InAnimNodeProperty, FProperty* InProperty, bool bInExposedOnPin, bool bInPinConnected, bool bInAlwaysDynamic)
{
    const bool bConstant = !bInAlwaysDynamic && (!bInExposedOnPin ||
    (bInExposedOnPin && !bInPinConnected));

    if(!InProperty->HasAnyPropertyParams(CPF_EditorOnly))
    {
        MessageLog.Warning(*FString::Printf(TEXT("Property %s on @@ is foldable, but not editor only"), *InProperty->GetName()), InAnimGraphNode);
    }

    // Create record and add it our lookup map
    TSharedRef<IAnimBlueprintCompilationContext::FFoldedPropertyRecord> Record =
    MakeShared<IAnimBlueprintCompilationContext::FFoldedPropertyRecord>
    (InAnimGraphNode, InAnimNodeProperty, InProperty, bConstant);
    TArray<TSharedRef<IAnimBlueprintCompilationContext::FFoldedPropertyRecord>>&
    Array = NodeToFoldedPropertyRecordMap.FindOrAdd(InAnimGraphNode);
    Array.Add(Record);

    // Record it in the appropriate data area
    if(bConstant)
    {
        ConstantPropertyParams.Add(Record);
    }
    else

```

```
{  
    MutablePropertyRecords.Add(Record);  
}  
}
```

GetterContext

- **功能描述:** 继续限定AnimGetter函数在哪个地方才可以使用，如果不填，则默认都可以用。
- **使用位置:** UFUNCTION
- **引擎模块:** AnimationGraph
- **元数据类型:** string="abc"
- **限制类型:** UAnimInstance及子类的AnimGetter函数
- **关联项:** AnimGetter
- **常用程度:** ★★

继续限定AnimGetter函数在哪个地方才可以使用，如果不填，则默认都可以用。

选项有：Transition, CustomBlend, AnimGraph。

源码注释：

```
* A context string can be provided in the GetterContext metadata and can  
contain any (or none) of the  
* following entries separated by a pipe (|)  
* Transition - only available in a transition rule  
* AnimGraph - only available in an animgraph (also covers state anim  
graphs)  
* CustomBlend - only available in a custom blend graph
```

测试代码：

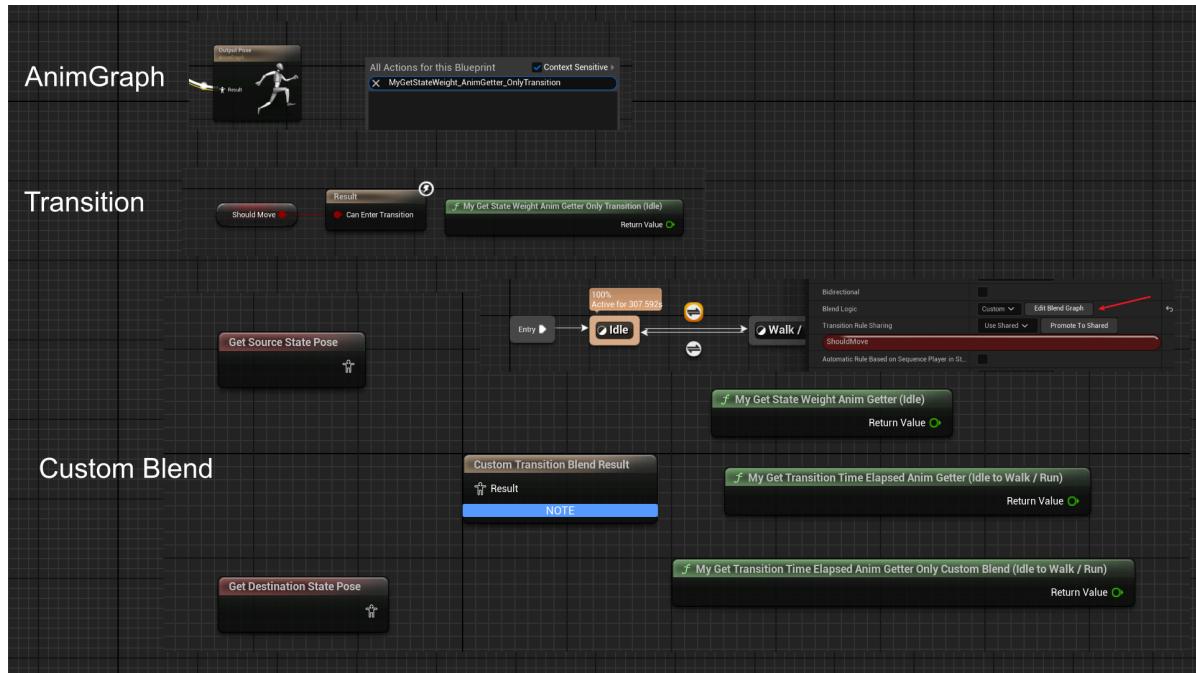
```
UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =  
(BlueprintThreadSafe))  
float MyGetStateweight(int32 MachineIndex, int32 StateIndex);  
public:  
UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =  
(BlueprintInternalUseOnly = "true", AnimGetter, GetterContext = "Transition",  
BlueprintThreadSafe))  
float MyGetStateweight_AnimGetter_OnlyTransition(int32 MachineIndex, int32  
StateIndex);  
  
UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =  
(BlueprintInternalUseOnly = "true", AnimGetter, GetterContext = "CustomBlend",  
BlueprintThreadSafe))  
float MyGetTransitionTimeElapsed_AnimGetter_OnlyCustomBlend(int32 MachineIndex,  
int32 TransitionIndex);
```

测试效果：

这个图要对比AnimGetter里的图来查看。

关注点一是在AnimGraph里的MyGetStateWeight_AnimGetter_OnlyTransition，如果不标GetterContext则是可以调用的，但标上就只能在Transition里调用。同时也发现该函数不能在CustomBlend里调用。

二是在CustomBlend里。操作步骤是在Rule上右侧细节面板改为Custom然后进入CustomBlend的蓝图。在该蓝图下，MyGetStateWeight可以调用，因为并没有填写GetterContext。而MyGetTransitionTimeElapsed_AnimGetter_OnlyCustomBlend可以开始调用了。



原理：

判断能否调用的函数如下。

```
bool UK2Node_AnimGetter::IsContextValidForSchema(const UEdGraphSchema* Schema)
{
    if(Contexts.Num() == 0)
    {
        // Valid in all graphs
        return true;
    }

    for(const FString& Context : Contexts)
    {
        UClass* ClassToCheck = nullptr;
        if(Context == TEXT("CustomBlend"))
        {
            ClassToCheck = UAnimationCustomTransitionSchema::StaticClass();
        }

        if(Context == TEXT("Transition"))
        {
            ClassToCheck = UAnimationTransitionSchema::StaticClass();
        }

        if(Context == TEXT("AnimGraph"))
        {
            ClassToCheck = UAnimationGraphSchema::StaticClass();
        }
    }
}
```

```

    }

    return Schema->GetClass() == ClassToCheck;
}

return false;
}

```

NeverAsPin

- 功能描述:** 在动画蓝图中使得动画节点的某个属性总是不暴露出来成为引脚
- 使用位置:** UPROPERTY
- 引擎模块:** Pin
- 元数据类型:** bool
- 限制类型:** FAnimNode_Base
- 关联项:** PinShownByDefault
- 常用程度:** ★★★

NeverAsPin源码中并没有用到，因为默认情况下就是不支持为引脚，所以不加也都一样。

测试代码：

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (AlwaysAsPin))
    int32 MyInt_AlwaysAsPin = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (NeverAsPin))
    int32 MyInt_NeverAsPin = 123;
};

```

测试效果：

MyInt_NeverAsPin只能和右边和默认的属性一样，不能显示为引脚。



原理：

发现bNeverShow并没有用到，因为默认情况下就是不支持为引脚。

```

void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();

    // Determine if this is a pose or array of poses
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ? ArrayProp->Inner : TestProperty);
    const bool bIsPoseInput = (StructProp && StructProp->Struct->IsChildOf(FPoseLinkBase::StaticStruct()));

    // TODO: Error if they specified two or more of these flags
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
    || bIsPoseInput;
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema->NAME_PinShownByDefault);
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema->NAME_PinHiddenByDefault);
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema->NAME_CustomizeProperty);
    const bool bCanTreatPropertyAsOptional =
        CanTreatPropertyAsOptional(TestProperty);

    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
        (bOptional_ShowByDefault || bOptional_HideByDefault);
    Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
    Record.bPropertyIsCustomized = bPropertyIsCustomized;
}

```

OnEvaluate

- **使用位置:** UPROPERTY
- **引擎模块:** AnimationGraph

原理:

在源码中发现，说明OnEvaluate已经放弃了。

```

// Dynamic value that needs to be wired up and evaluated each frame
const FString& EvaluationHandlerStr = SourcePinProperty->GetMetaData(AnimGraphDefaultSchema->NAME_OnEvaluate);
FName EvaluationHandlerName(*EvaluationHandlerStr);
if (EvaluationHandlerName != NAME_None)
{
    // warn that NAME_OnEvaluate is deprecated:
    InCompilationContext.GetMessageLog().Warning(*LOCTEXT("OnEvaluateDeprecated",
    "OnEvaluate meta data is deprecated, found on @@").ToString(),
    SourcePinProperty);
}

```

PinShownByDefault

- **功能描述:** 在动画蓝图中使得动画节点的某个属性一开始就暴露出来成为引脚，但也可以改变。
- **使用位置:** UPROPERTY
- **引擎模块:** AnimationGraph
- **元数据类型:** bool
- **限制类型:** FAnimNode_Base
- **关联项:** AlwaysAsPin, NeverAsPin
- **常用程度:** ★★★

在动画蓝图中使得动画节点的某个属性一开始就暴露出来成为引脚。

和常规的蓝图不同，FAnimNode_Base里面的属性默认是不在节点上显示出来的。因此才需要这个meta显式的指定哪些需要显式。

PinShownByDefault目前只在动画蓝图节点上应用。

相反的，可以用PinHiddenByDefault来隐藏属性成为引脚。

测试代码：

```
USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
    meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
    meta = (AlwaysAsPin))
    int32 MyInt_AlwaysAsPin = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
    meta = (NeverAsPin))
    int32 MyInt_NeverAsPin = 123;
};

UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyTestPinshown : public
UAnimGraphNode_Base
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(EditAnywhere, Category = Settings)
    FAnimNode_MyTestPinShown Node;
};
```

测试效果：

可见，同样的两个属性，MyInt_NotShown 默认情况不显示成节点，只能在细节面板里编辑。而 MyInt_PinShownByDefault默认情况下成为引脚。当PinShownByDefault还可以改变去掉Pin的功能。



原理：

源码里唯一用的地方就是在FAnimBlueprintNodeOptionalPinManager，其实质就是处理动画蓝图节点的Pin如何显示。

```
void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();

    // Determine if this is a pose or array of poses
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ? ArrayProp->Inner : TestProperty);
    const bool bIsPoseInput = (StructProp && StructProp->Struct->IsChildOf(FPoseLinkBase::StaticStruct()));

    // @TODO: Error if they specified two or more of these flags
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
        || bIsPoseInput;
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema->NAME_PinShownByDefault);
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema->NAME_PinHiddenByDefault);
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema->NAME_CustomizeProperty);
    const bool bCanTreatPropertyAsOptional =
        CanTreatPropertyAsOptional(TestProperty);

    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
        (bOptional_ShowByDefault || bOptional_HideByDefault);
    Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
    Record.bPropertyIsCustomized = bPropertyIsCustomized;
}
```

DisallowedAssetDataTags

- 功能描述：**在UObject*属性上指定Tags来进行过滤，必须没有拥有该Tags才可以被选择。
- 使用位置：**UPROPERTY
- 引擎模块：**Asset Property
- 元数据类型：**strings="a=b, c=d, e=f"
- 限制类型：**UObject*
- 关联项：**RequiredAssetDataTags, AssetRegistrySearchable

- 常用程度： ★★

ForceShowEngineContent

- **功能描述：** 指定UObject*属性的资源可选列表里强制可选引擎的内建资源
- **使用位置：** UPROPERTY
- **引擎模块：** Asset Property
- **元数据类型：** bool
- **限制类型：** UObject*
- **关联项：** ForceShowPluginContent
- **常用程度：** ★★

指定UObject*属性的资源可选列表里强制可选引擎的内建资源。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_ShowContent :public UDataAsset
{
public:
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object)
    TObjectPtr<UObject> MyAsset_Default;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(ForceShowEngineContent))
    TObjectPtr<UObject> MyAsset_ForceShowEngineContent;

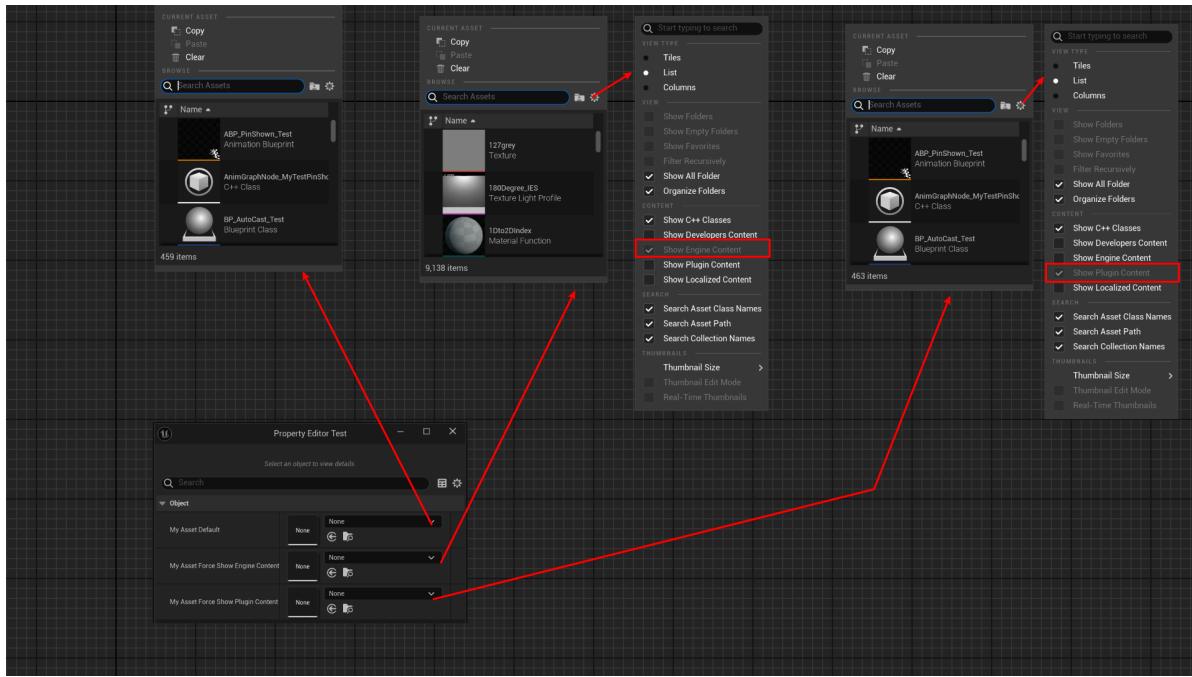
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(ForceShowPluginContent))
    TObjectPtr<UObject> MyAsset_ForceShowPluginContent;
};
```

测试结果：

可见MyAsset_Default默认是只包含本项目的资源。

MyAsset_ForceShowEngineContent的作用其实就是在选项卡里勾选ShowEngineContent，因此结果上会发现多了非常多的可选资源。

MyAsset_ForceShowPluginContent的作用同样是在选项卡里勾选ShowPluginContent，可以选择别的插件里的资源。



原理：

在属性的资源选择器里会尝试寻找ForceShowEngineContent和ForceShowPluginContent，然后设置到AssetPickerConfig里，从而改变资源的可选类型。

```

void SPropertyMenuAssetPicker::Construct( const FArguments& InArgs )
{
    const bool bForceShowEngineContent = PropertyHandle ? PropertyHandle->HasMetaData(TEXT("ForceShowEngineContent")) : false;
    const bool bForceShowPluginContent = PropertyHandle ? PropertyHandle->HasMetaData(TEXT("ForceshowPluginContent")) : false;

    FAssetPickerConfig AssetPickerConfig;
    // Force show engine content if meta data says so
    AssetPickerConfig.bForceShowEngineContent = bForceShowEngineContent;
    // Force show plugin content if meta data says so
    AssetPickerConfig.bForceShowPluginContent = bForceShowPluginContent;
}

```

ForceShowPluginContent

- 功能描述：**指定UObject*属性的资源可选列表里强制可选其他插件里的内建资源
- 使用位置：**UPROPERTY
- 引擎模块：**Asset Property
- 元数据类型：**bool
- 限制类型：**UObject*
- 关联项：**ForceShowEngineContent

GetAssetFilter

- 功能描述：**指定一个UFUNCTION来对UObject*属性的可选资源进行排除过滤。
- 使用位置：**UPROPERTY

- **引擎模块:** Asset Property
- **元数据类型:** string="abc"
- **限制类型:** UObject*
- **常用程度:** ★★★

指定一个UFUNCTION来对UObject*属性的可选资源进行排除过滤。

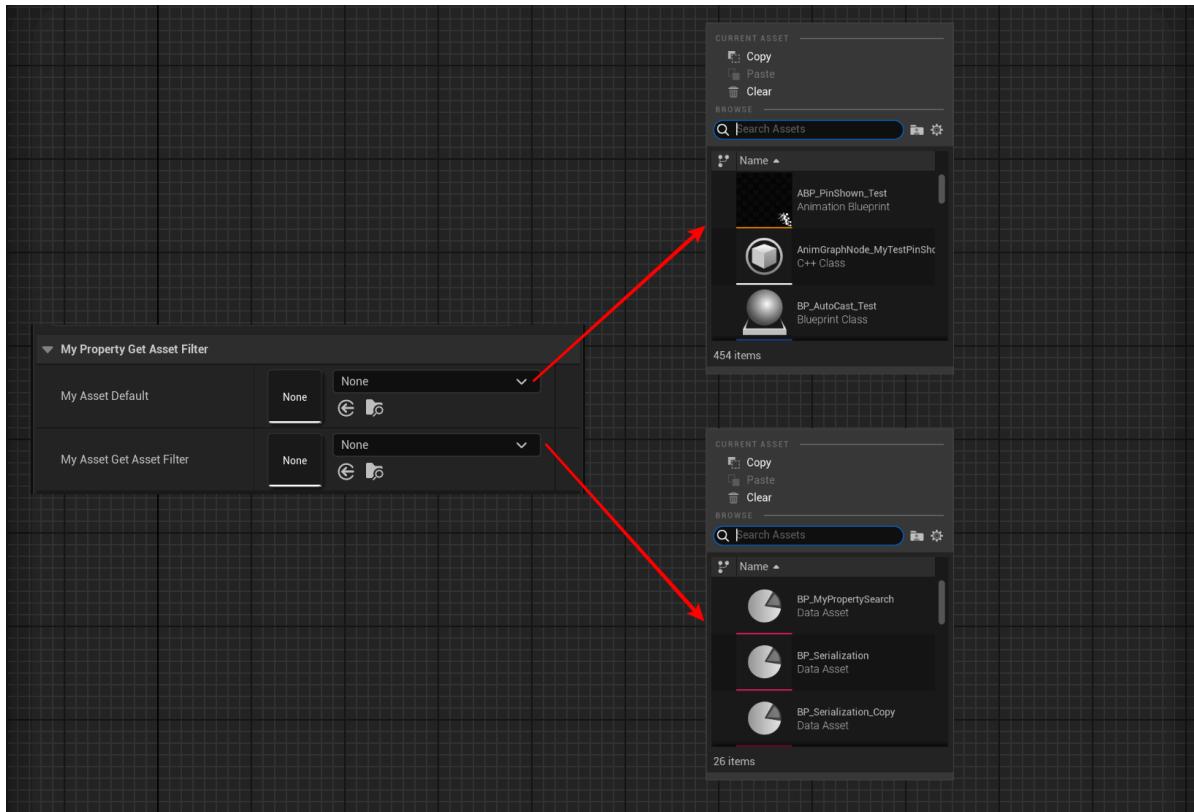
- 指定的函数名字必须是UFUNCTION，在本类中定义。
- 函数的原型是bool FuncName(const FAssetData& AssetData) const;，返回true来排除掉该资产。
- 这是一种让用户自定义资产过滤的方式。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_GetAssetFilter :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UObject* MyAsset_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (GetAssetFilter =
"IsShouldFilterAsset"))
    UObject* MyAsset_GetAssetFilter;
public:
    UFUNCTION()
    bool IsShouldFilterAsset(const FAssetData& AssetData)
    {
        return !AssetData.IsInstanceOf<UDataAsset>();
    }
};
```

测试效果：

可以见到，MyAsset_GetAssetFilter进行过滤后只允许DataAsset类型的资产。



原理：

在SPropertyEditorAsset（对应UObject类型属性）中有判断GetAssetFilter的meta，得到函数并附加到资产排除的回调里去。

```

void SPropertyEditorAsset::Construct(const FArguments& InArgs, const
TSharedPtr<FPropertyEditor>& InPropertyEditor)
{
    if (Property && Property->GetOwnerProperty()->HasMetaData("GetAssetFilter"))
    {
        // Add MetaData asset filter
        const FString GetAssetFilterFunctionName = Property->GetOwnerProperty()->
GetMetaData("GetAssetFilter");
        if (!GetAssetFilterFunctionName.IsEmpty())
        {
            TArray<UObject*> ObjectList;
            if (PropertyEditor.IsValid())
            {
                PropertyEditor->GetPropertyHandle()->GetOuterObjects(ObjectList);
            }
            else if (PropertyHandle.IsValid())
            {
                PropertyHandle->GetOuterObjects(ObjectList);
            }
            for (UObject* object : ObjectList)
            {
                const UFunction* GetAssetFilterFunction = object ? object-
>FindFunction(*GetAssetFilterFunctionName) : nullptr;
                if (GetAssetFilterFunction)
                {

```

```

AppendOnShouldFilterAssetCallback(FOnShouldFilterAsset::Create(object,
GetAssetFilterFunction->GetFName())));

```

```
        }
    }
}
}
```

RequiredAssetDataTags

- **功能描述:** 在UObject*属性上指定Tags来进行过滤，必须拥有该Tags才可以被选择。
- **使用位置:** UPROPERTY
- **引擎模块:** Asset Property
- **元数据类型:** strings="a=b, c=d, e=f"
- **限制类型:** UObject*
- **关联项:** DisallowedAssetDataTags, AssetRegistrySearchable
- **常用程度:** ★★

在UObject*属性上指定Tags来进行过滤，必须拥有该Tags才可以被选择。

相关联的可参考AssetRegistrySearchable标识符和GetAssetRegistryTags 方法。

测试代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyTableRow_Required :public FTableRowBase
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyTableRow_Disallowed :public FTableRowBase
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat = 123.f;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UTexture2D* MyTexture;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_AssetDataTags :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object)
    TObjectPtr<UObject> MyAsset_Default;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(RequiredAssetDataTags = "MyIdForSearch=MyId456"))
TObjectPtr<UObject> MyAsset_RequiredAssetDataTags;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(DisallowedAssetDataTags = "MyOtherId=MyOtherId789"))
TObjectPtr<UObject> MyAsset_DisallowedAssetDataTags;
public:
UPROPERTY(EditAnywhere, Category = DataTable)
TObjectPtr<class UDataTable> MyDataTable_Default;

UPROPERTY(EditAnywhere, Category = DataTable, meta = (RequiredAssetDataTags =
"RowStructure=/Script/Insider.MyTableRow_Required"))
TObjectPtr<class UDataTable> MyDataTable_RequiredAssetDataTags;

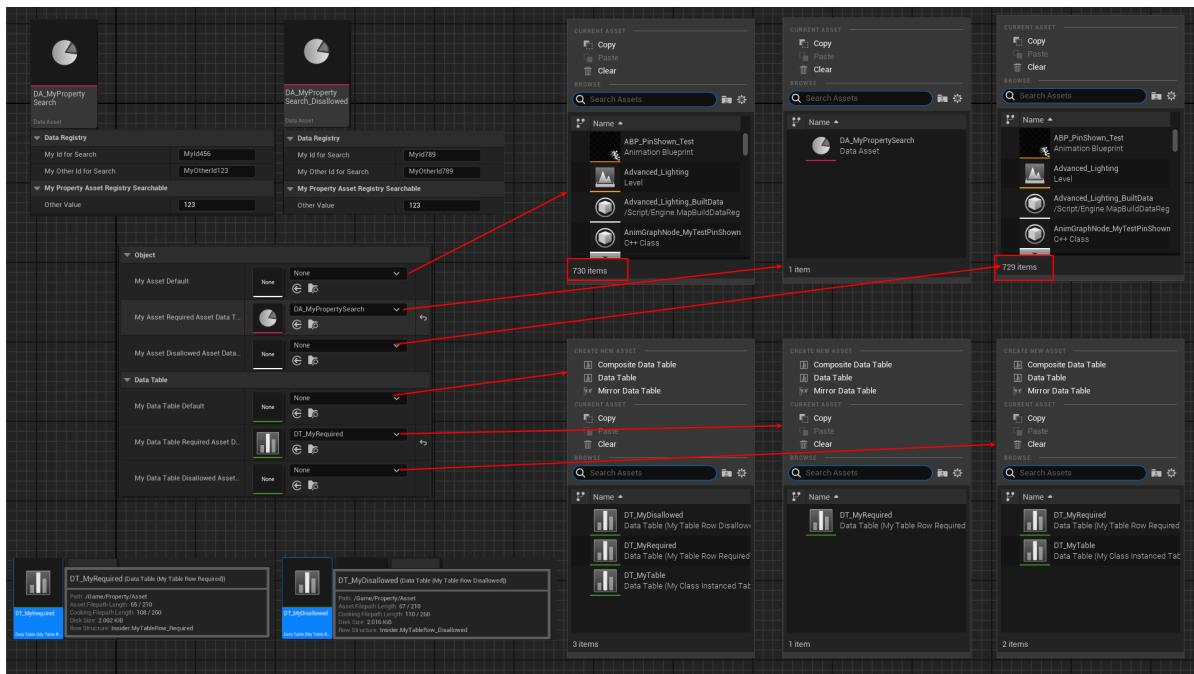
UPROPERTY(EditAnywhere, Category = DataTable, meta = (DisallowedAssetDataTags =
"RowStructure=/Script/Insider.MyTableRow_Disallowed"))
TObjectPtr<class UDataTable> MyDataTable_DisallowedAssetDataTags;
};

```

测试效果：

如上面代码所见，定义了两个不同类型的FTableRowBase，并且也创建了两个DataTable。同时也有两个DataAsset（AssetRegistrySearchable的例子里定义的结构）都有MyIdForSearch和MyOtherId的Tag，但是有不同的值，以此来进行区分。

- MyAsset_Default，可以筛选出所有的对象，图中示例有730个。
- MyAsset_RequiredAssetDataTags，因为加了RequiredAssetDataTags，只有DA_MyPropertySearch符合，因为MyIdForSearch=MyId456。
- MyAsset_DisallowedAssetDataTags，把DA_MyPropertySearch_Disallowed过滤掉了，因为我配置的MyOtherId=MyOtherId789，因此只剩下729个。
- 关于DataTable也是同理。MyDataTable_Default可以获取所有的DataTable（有3个），而MyDataTable_RequiredAssetDataTags限制了RowStructure只能是FMyTableRow_Required（因此只能筛选出一个）。MyDataTable_DisallowedAssetDataTags排除掉一个RowStructure为FMyTableRow_Disallowed的，因此就剩下2个。



源码中例子：

```

UPROPERTY(Category="StateTree", EditAnywhere, meta=
(RequiredAssetDataTags="Schema=/Script/MassAIBehavior.MassStateTreeSchema"))
TObjectPtr<UStateTree> StateTree;

UPROPERTY(EditAnywhere, Category=Appearance, meta = (RequiredAssetDataTags =
"RowStructure=/Script/UMG.RichImageRow"))
TObjectPtr<class UDataTable> ImageSet;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Compositing, meta =
(AllowPrivateAccess, RequiredAssetDataTags = "IsSourceValid=True"), Setter =
SetCompositeTexture, Getter = GetCompositeTexture)
TObjectPtr<class UTexture> CompositeTexture;

```

原理：

在UObject*属性上RequiredAssetDataTags和DisallowedAssetDataTags的配置，会在这个属性的编辑器 (SPropertyEditorAsset) 初始化的时候解析提取到其成员变量RequiredAssetDataTags和 DisallowedAssetDataTags里，本质就是个键值对。而后续在进行Asset过滤的时候 (IsAssetFiltered的调用)，就会开始把FAssetData里的Tags去匹配该属性的Tags需求。Disallowed的出现就排除掉， Required的必须拥有才不会被过滤，最终实现了过滤效果。

关于FAssetData里的Tags，可以参考AssetRegistrySearchable标识符和GetAssetRegistryTags 方法的调用和实现，简单来说就是在对象上会有个方式主动提供Tags给AssetRegistry。

关于DataTable为何可以通过RowStructure过滤，通过查看DataTable里的GetAssetRegistryTags方法就可以知道它主动提供了RowStructure的Tags注册。

```

FAssetDataTagMapBase=TSortedMap< FName, FString, FDefaultAllocator,
FNameFastLess>;

SPropertyEditorAsset::
/** Tags (and eventually values) that can NOT be used with this property */
TSharedPtr<FAssetDataTagMap> DisallowedAssetDataTags;

```

```


    /** Tags and values that must be present for an asset to be used with this
property */

    TSharedPtr<FAssetDataTagMap> RequiredAssetDataTags;

void SPropertyEditorAsset::InitializeAssetDataTags(const FProperty* Property)
{
    if (Property == nullptr)
    {
        return;
    }

const FProperty* MetadataProperty = GetActualMetadataProperty(Property);
const FString DisallowedAssetDataTagsFilterString = MetadataProperty-
>GetMetaData("DisallowedAssetDataTags");
if (!DisallowedAssetDataTagsFilterString.IsEmpty())
{
    TArray< FString> DisallowedAssetDataTagsAndValues;

    DisallowedAssetDataTagsFilterString.ParseIntoArray(DisallowedAssetDataTagsAndValue
es, TEXT(","), true);

    for (const FString& TagAndOptionalValueString :
DisallowedAssetDataTagsAndValues)
    {
        TArray< FString> TagAndOptionalValue;
        TagAndOptionalValueString.ParseIntoArray(TagAndOptionalValue, TEXT("="),
true);
        size_t NumStrings = TagAndOptionalValue.Num();
        check((NumStrings == 1) || (NumStrings == 2)); // there should be a
single '=' within a tag/value pair

        if (!DisallowedAssetDataTags.IsValid())
        {
            DisallowedAssetDataTags = MakeShared<FAssetDataTagMap>();
        }
        DisallowedAssetDataTags->Add(FName(*TagAndOptionalValue[0]), (NumStrings
> 1) ? TagAndOptionalValue[1] : FString());
    }
}

const FString RequiredAssetDataTagsFilterString = MetadataProperty-
>GetMetaData("RequiredAssetDataTags");
if (!RequiredAssetDataTagsFilterString.IsEmpty())
{
    TArray< FString> RequiredAssetDataTagsAndValues;

    RequiredAssetDataTagsFilterString.ParseIntoArray(RequiredAssetDataTagsAndValues,
TEXT(","), true);

    for (const FString& TagAndOptionalValueString :
RequiredAssetDataTagsAndValues)
    {
        TArray< FString> TagAndOptionalValue;
        TagAndOptionalValueString.ParseIntoArray(TagAndOptionalValue, TEXT("="),
true);
    }
}


```

```

        size_t NumStrings = TagAndOptionalValue.Num();
        check((NumStrings == 1) || (NumStrings == 2)); // there should be a
single '=' within a tag/value pair

        if (!RequiredAssetDataTags.IsValid())
        {
            RequiredAssetDataTags = MakeShared<FAssetDataTagMap>();
        }
        RequiredAssetDataTags->Add(FName(*TagAndOptionalValue[0]), (NumStrings >
1) ? TagAndOptionalValue[1] : FString());
    }
}

bool SPropertyEditorAsset::IsAssetFiltered(const FAssetData& InAssetData)
{
//判断只要包含就不符合
    if (DisallowedAssetDataTags.IsValid())
    {
        for (const auto& DisallowedTagAndValue : *DisallowedAssetDataTags.Get())
        {
            if
(InAssetData.TagsAndValues.ContainsKeyValue(DisallowedTagAndValue.Key,
DisallowedTagAndValue.Value))
            {
                return true;
            }
        }
    }
//判断必须包含才不会被过滤掉
    if (RequiredAssetDataTags.IsValid())
    {
        for (const auto& RequiredTagAndValue : *RequiredAssetDataTags.Get())
        {
            if
(!InAssetData.TagsAndValues.ContainsKeyValue(RequiredTagAndValue.Key,
RequiredTagAndValue.Value))
            {
                // For backwards compatibility compare against short name version
of the tag value.
                if (!FPackageName::IsShortPackageName(RequiredTagAndValue.Value))
&&

InAssetData.TagsAndValues.ContainsKeyValue(RequiredTagAndValue.Key,
FPackageName::ObjectPathToObjectName(RequiredTagAndValue.Value)))
                {
                    continue;
                }
                return true;
            }
        }
    }
    return false;
}

void UDataTable::GetAssetRegistryTags(FAssetRegistryTagsContext Context) const

```

```

{
    if (AssetImportData)
    {
        Context.AddTag( FAssetRegistryTag(SourceFileName(), AssetImportData-
>GetSourceData().ToJson(), FAssetRegistryTag::TT_Hidden) );
    }

    // Add the row structure tag
    {
        static const FName RowStructureTag = "RowStructure";
        Context.AddTag( FAssetRegistryTag(RowStructureTag,
GetRowStructPathName().ToString(), FAssetRegistryTag::TT_Alphabetical) );
    }

    Super::GetAssetRegistryTags(Context);
}

```

AdvancedDisplay

- 功能描述:** 把函数的一些参数折叠起来不显示，需要手动点开下拉箭头来展开编辑。
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** strings="a, b, c"
- 常用程度:** ★★★★☆

把函数的一些参数折叠起来不显示，需要手动点开下拉箭头来展开编辑。

AdvancedDisplay同时支持两种格式，一是用"Parameter1, Parameter2, .."来显式的指定需要折叠的参数名字，适用于要折叠的参数不连续或者处在函数参数列表中中央的情况下。二是"N"指定一个数字序号，第N之后的所有参数将显示为高级引脚。

测试代码：

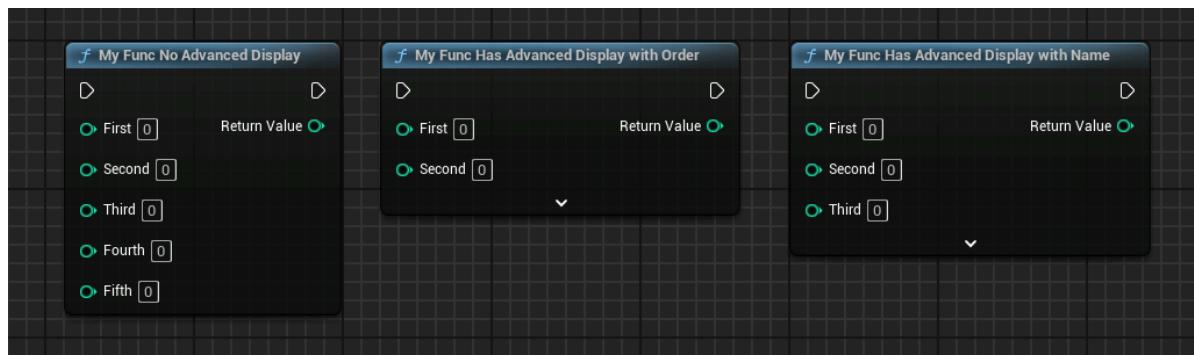
```

UFUNCTION(BlueprintCallable, meta = (AdvancedDisplay = "2"))
static int32 MyFunc_HasAdvancedDisplay_withOrder(int32 First, int32 Second,
int32 Third, int32 Fourth, int32 Fifth) { return 0; }
UFUNCTION(BlueprintCallable, meta = (AdvancedDisplay = "Fourth,Fifth"))
static int32 MyFunc_HasAdvancedDisplayWithName(int32 First, int32 Second,
int32 Third, int32 Fourth, int32 Fifth) { return 0; }

UFUNCTION(BlueprintCallable, meta = ())
static int32 MyFunc_NoAdvancedDisplay(int32 First, int32 Second, int32 Third,
int32 Fourth, int32 Fifth) { return 0; }

```

蓝图效果：



源码中典型的例子是PrintString，在第2个参数后的其他参数就都折叠了起来。

```
UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
CallableWithoutWorldContext, Keywords = "log print", AdvancedDisplay = "2",
DevelopmentOnly), Category="Development")
static ENGINE_API void PrintString(const UObject* WorldContextObject, const
FString& InString = FString(TEXT("Hello")), bool bPrintToScreen = true, bool
bPrintToLog = true, FLinearColor TextColor = FLinearColor(0.0f, 0.66f, 1.0f),
float Duration = 2.f, const FName Key = NAME_None);
```

原理：

AdvancedDisplay使得被标注的函数参数增加EPropertyFlags.AdvancedDisplay的标记，从而使得其被折叠起来。这个逻辑是在UHT对函数进行解析的时候设置的。

```
//支持参数名字和数字序号两种模式
if (_metaData.TryGetValue(UhtNames.AdvancedDisplay, out string? foundString))
{
    _parameterNames = foundString.ToString().Split(',',',
StringSplitOptions.RemoveEmptyEntries);
    for (int index = 0, endIndex = _parameterNames.Length; index < endIndex;
++index)
    {
        _parameterNames[index] = _parameterNames[index].Trim();
    }
    if (_parameterNames.Length == 1)
    {
        _bUseNumber = Int32.TryParse(_parameterNames[0], out
_numberLeaveUnmarked);
    }
}

//设置EPropertyFlags.AdvancedDisplay
private static void UhtFunctionParser::ParseParameterList(UhtParsingScope
topScope, UhtPropertyParseOptions options)
{
    UhtAdvancedDisplayParameterHandler advancedDisplay =
new(topScope.ScopeType.MetaData);

    topScope.TokenReader.RequireList(')', ',', false, () =>
{
```

```

topScope.HeaderParser.GetCachedPropertyParser().Parse(topScope, disallowFlags,
options, propertyCategory,
    (UhtParsingScope topScope, UhtProperty property, ref
UhtToken nameToken, UhtLayoutMacroType layoutMacroType) =>
{
    property.PropertyFlags |= EPropertyFlags.Parm;
    if (advancedDisplay.CanMarkMore() &&
advancedDisplay.ShouldMarkParameter(property.EngineName))
    {
        property.PropertyFlags |=
EPropertyFlags.AdvancedDisplay;
    }
}

```

AllowPrivateAccess

- 功能描述:** 允许一个在C++中private的属性，可以在蓝图中访问。
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 关联项:** BlueprintProtected
- 常用程度:** ★★★★☆

允许一个在C++中private的属性，可以在蓝图中访问。

AllowPrivateAccess的意义是允许这个属性在C++是private的，不允许C++子类访问，但又允许其暴露到蓝图中访问。

测试代码：

```

public:
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite)
    int32 MyNativeInt_NativePublic;

private:
    //CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    //error : BlueprintReadWrite should not be used on private members
    UPROPERTY()
    int32 MyNativeInt_NativePrivate;

    // (AllowPrivateAccess = TRUE, Category = MyFunction_Access,
ModuleRelativePath = Function/MyFunction_Access.h)
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintReadWrite, meta = (AllowPrivateAccess = true))
    int32 MyNativeInt_NativePrivate_AllowPrivateAccess;

```

在MyNativeInt_NativePrivate上如果尝试加上BlueprintReadWrite或BlueprintReadOnly都会触发UHT编译报错。

蓝图里的效果：

默认情况下MyNativeInt_NativePrivate_AllowPrivateAccess在蓝图里的访问权限和MyNativeInt_NativePublic一致。

如果读者想要修改改属性在蓝图中的访问权限，则可以配合加上BlueprintProtected和BlueprintPrivate。



原理：

UHT在识别属性的BlueprintReadWrite或BlueprintReadOnly标识符的时候，会同时检测是否有AllowPrivateAccess，没有的话会触发报错。

因此AllowPrivateAccess的意义其实只是在阻止UHT的报错，这层检测报错过了之后，属性上的BlueprintReadWrite或BlueprintReadOnly就会被识别并发挥作用，从而可以在蓝图中访问。

```
private static void BlueprintReadWriteSpecifier(UhtSpecifierContext specifierContext)
{
    bool allowPrivateAccess =
context.MetaData.TryGetValue(UhtNames.AllowPrivateAccess, out string? privateAccessMD) && !privateAccessMD.Equals("false",
StringComparison.OrdinalIgnoreCase);
    if (specifierContext.AccessSpecifier == UhtAccessSpecifier.Private &&
!allowPrivateAccess)
    {
        context.MessageSite.LogError("BlueprintReadWrite should not be
used on private members");
    }
}

private static void BlueprintReadOnlySpecifier(UhtSpecifierContext specifierContext)
{
    bool allowPrivateAccess =
context.MetaData.TryGetValue(UhtNames.AllowPrivateAccess, out string? privateAccessMD) && !privateAccessMD.Equals("false",
StringComparison.OrdinalIgnoreCase);
    if (specifierContext.AccessSpecifier == UhtAccessSpecifier.Private &&
!allowPrivateAccess)
    {
        context.MessageSite.LogError("BlueprintReadOnly should not be
used on private members");
    }
}
```

BlueprintAutocast

- **功能描述:** 告诉蓝图系统这个函数是用来支持从A类型到B类型的自动转换。
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **常用程度:** ★

告诉蓝图系统这个函数是用来支持从A类型到B类型的自动转换。

所谓自动转换指的是从A类型的Pin拖拉到B类型的Pin时，蓝图会在其中自动的生成类型转换节点。

这种转换函数必须是BlueprintPure，因为其实是被动调用的，不带主动执行节点。

测试代码：

```
USTRUCT(BlueprintType)
struct FAutoCastFrom
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 X = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Y = 0;
};

USTRUCT(BlueprintType)
struct FAutoCastTo
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Sum = 0;
};

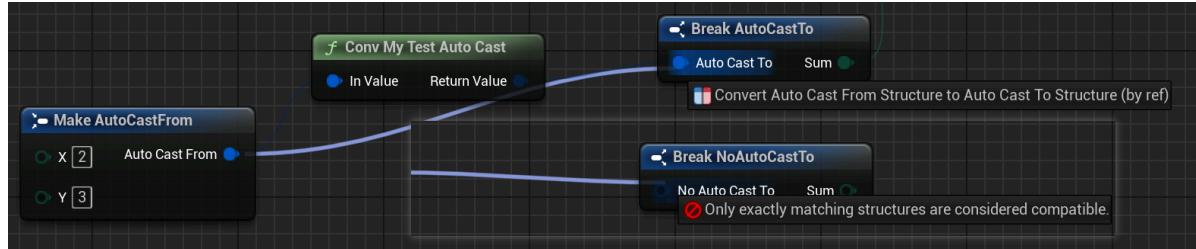
USTRUCT(BlueprintType)
struct FNoAutoCastTo
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Sum = 0;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_AutoCast :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = (BlueprintAutocast))
    static FAutoCastTo Conv_MyTestAutoCast(const FAutoCastFrom& InValue);
};
```

```
//转换函数也经常配合CompactNodeTitle使用。
UFUNCTION(BlueprintPure, Category="Widget", meta = (CompactNodeTitle = "->", BlueprintAutocast))
static UMG_API FInputEvent GetInputEventFromKeyEvent(const FKeyEvent& Event);
```

示例效果：

支持自动转换的FAutoCastTo就在拖拉连线的时候就会自动生成节点，而没有自动转换函数的FNoAutoCastTo就会产生报错。



原理代码：

从这可以看出，该函数必须是static，C++中的Public函数，标上BlueprintPure，拥有返回值，且有一个输入参数。引擎里类型的自动转换关系是靠FAutocastFunctionMap来维护的。

```
static bool IsAutocastFunction(const UFunction* Function)
{
    const FName BlueprintAutocast(TEXT("BlueprintAutocast"));
    return Function
        && Function->HasMetaData(BlueprintAutocast)
        && Function->HasAllFunctionFlags(FUNC_Static | FUNC_Native | FUNC_Public
| FUNC_Blueprint)
        && Function->GetReturnProperty()
        && GetFirstInputProperty(Function);
}
```

BlueprintGetter

- 功能描述：**采用一个自定义的get函数来读取。
如果没有设置BlueprintSetter或BlueprintReadWrite，则会默认设置BlueprintReadOnly.
- 使用位置：**UFUNCTION, UPROPERTY
- 引擎模块：**Blueprint
- 元数据类型：**string="abc"
- 关联项：**
 - UFUNCTION: BlueprintGetter
 - UPROPERTY: BlueprintGetter
- 常用程度：**★★★

BlueprintInternalUseOnly

- **功能描述:** 标明该元素是作为蓝图系统的内部调用或使用，不暴露出来在用户层面直接定义或使用。
- **使用位置:** UFUNCTION, USTRUCT
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **关联项:**

Meta: BlueprintType, BlueprintInternalUseOnlyHierarchical

UFUNCTION: BlueprintInternalUseOnly

USTRUCT: BlueprintInternalUseOnly

- **常用程度:** ★★★

也可以用在USTRUCT上，标明该结构不可用来定义新BP变量，但可作为别的类的成员变量暴露和变量传递。

用在UFUNCTION上时：此函数是一个内部实现细节，用于实现另一个函数或节点。其从未直接在蓝图图表中公开。

BlueprintInternalUseOnlyHierarchical

- **功能描述:** 标明该结构及其子类都不暴露给用户定义和使用，均只能在蓝图系统内部使用
- **使用位置:** USTRUCT
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **关联项:**

Meta: BlueprintInternalUseOnly, BlueprintType

USTRUCT: BlueprintInternalUseOnlyHierarchical

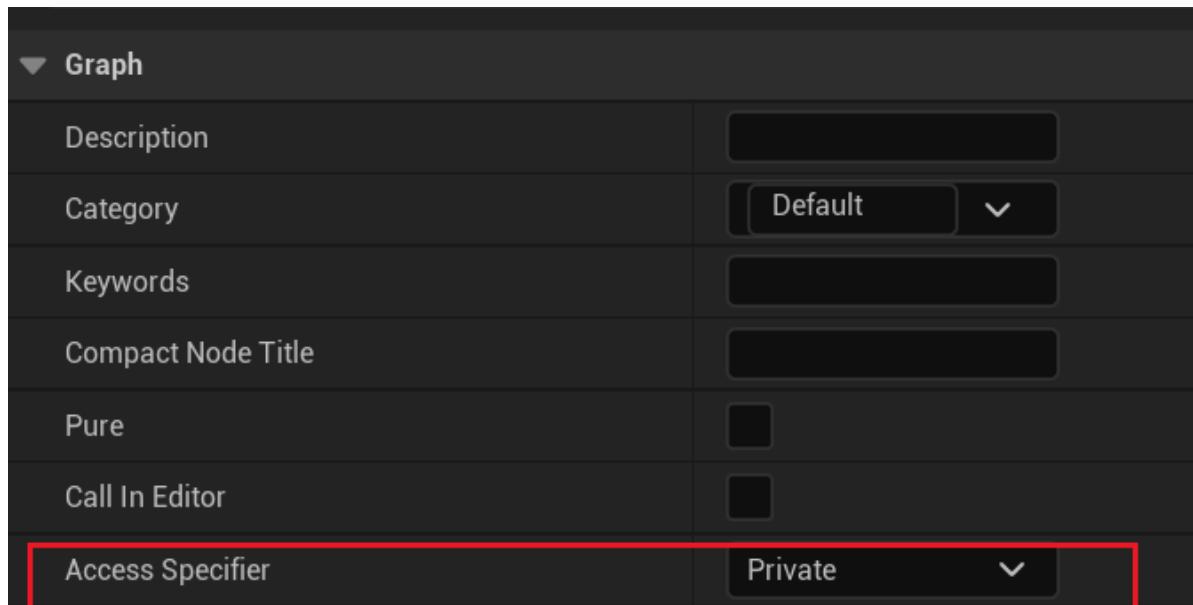
- **常用程度:** ★

指明一个不向最终用户公开的BlueprintType类型的结构以及其派生的结构。

BlueprintPrivate

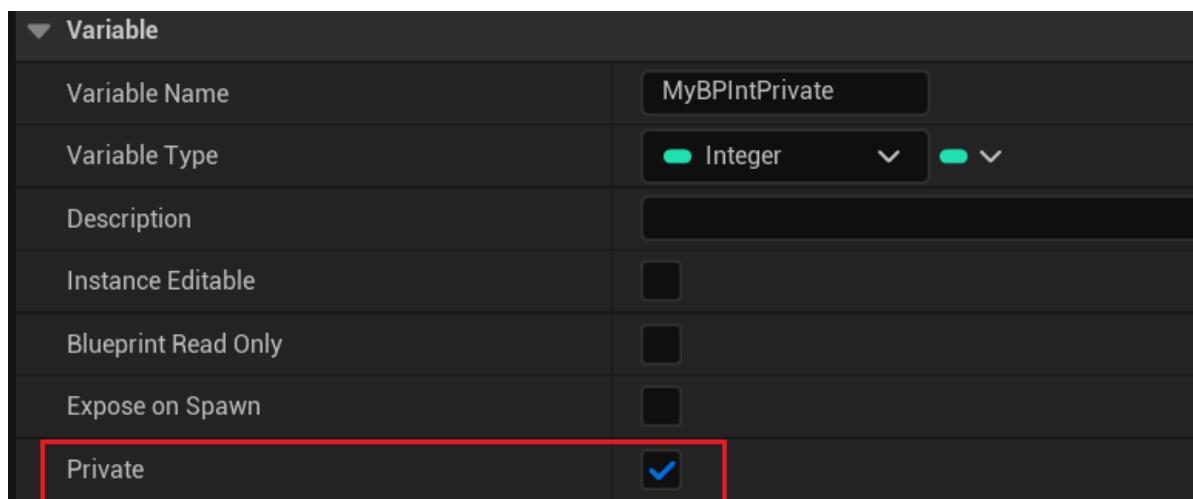
- **功能描述:** 指定该函数或属性只能在本类中被调用或读写，类似C++中的private的作用域限制。不可在别的蓝图类里访问。
- **使用位置:** UFUNCTION, UPROPERTY
- **元数据类型:** bool
- **关联项:** BlueprintProtected
- **常用程度:** ★★

在函数细节面板上可以设置函数的访问权限：



造成的结果就是在函数上增加BlueprintPrivate="true"

在细节面板上可以设置属性的



结果也是在属性上增加BlueprintPrivate="true"

BlueprintProtected

- 功能描述:** 指定该函数或属性只能在本类以及子类中被调用或读写，类似C++中的protected作用域限制。不可在别的蓝图类里访问。
- 使用位置:** UFUNCTION, UPROPERTY
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 关联项:** BlueprintPrivate, AllowPrivateAccess
- 常用程度:** ★★★

作用在函数上：

标记该函数只能在本类以及子类中被调用，类似C++中的protected函数的作用域限制。不可在别的蓝图类里调用。

作用在属性上时，标明该属性只能在本类或派生类里进行读写，但不能在别的蓝图类里访问。

指定该函数或属性只能在本类以及子类中被调用或读写，类似C++中的protected函数的作用域限制。不可在别的蓝图类里访问。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Access :public AActor
{
public:
    GENERATED_BODY()

public:
    //BlueprintProtected = true, ModuleRelativePath =
    //Function/MyFunction_Access.h
    //FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable, meta = (BlueprintProtected = "true"))
    void MyNative_HasProtected() {}

    //BlueprintPrivate = true, ModuleRelativePath =
    //Function/MyFunction_Access.h
    //FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable, meta = (BlueprintPrivate = "true"))
    void MyNative_HasPrivate() {}

public:
    //FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyNative_NativePublic() {}

protected:
    //FUNC_Final | FUNC_Native | FUNC_Protected | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyNative_NativeProtected() {}

private:
    //FUNC_Final | FUNC_Native | FUNC_Private | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyNative_NativePrivate() {}

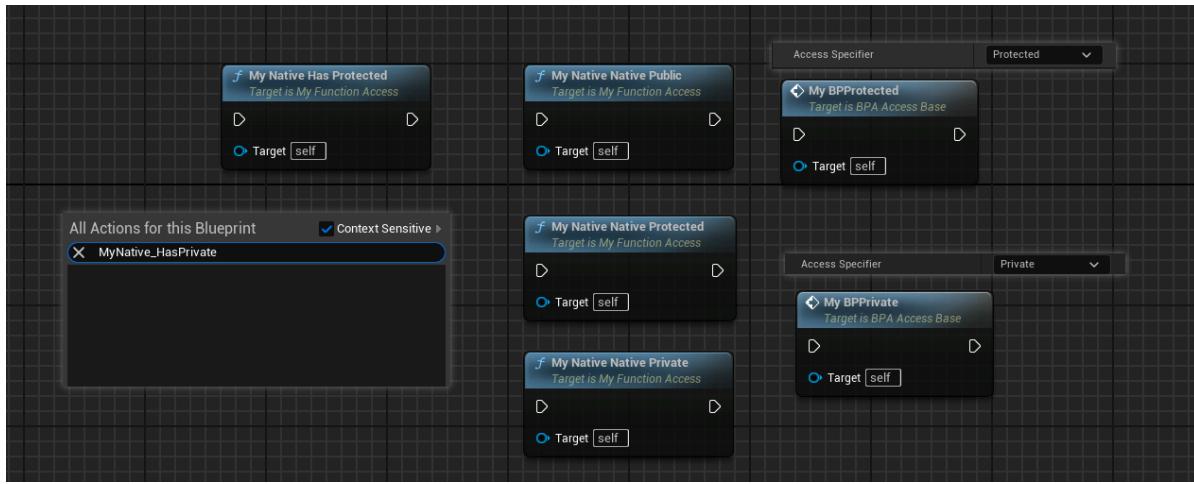
};
```

测试效果：

蓝图中的子类 (BPA_Access_Base继承自AMyFunction_Access) 效果：

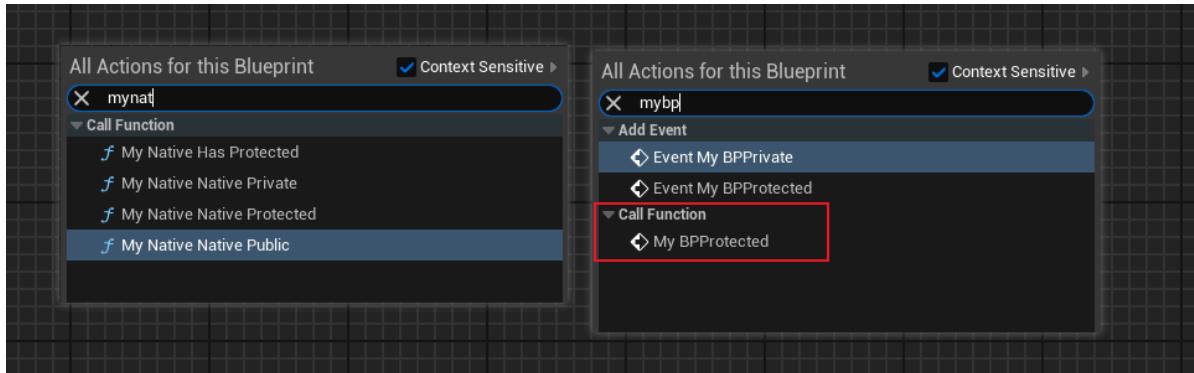
可见BlueprintProtected可以被子类调用，但是BlueprintPrivate只能在本类 (C++类中定义的只能在C++中调用，蓝图中定义的只能在蓝图本类中调用)。而在C++中用protected或private修饰的函数会相应的增加FUNC_Protected和FUNC_Private，但是实际上并不会发生作用。因为机制的设计目的就是如此 (详见后文解释)。

而在BPA_Access_Base中直接定义的MyBPProtected和MyBPPrivate通过在函数细节面板上直接设置AccessSpecifier，可以在本类都可以调用，但是MyBPPrivate在更加的蓝图子类无法被调用。

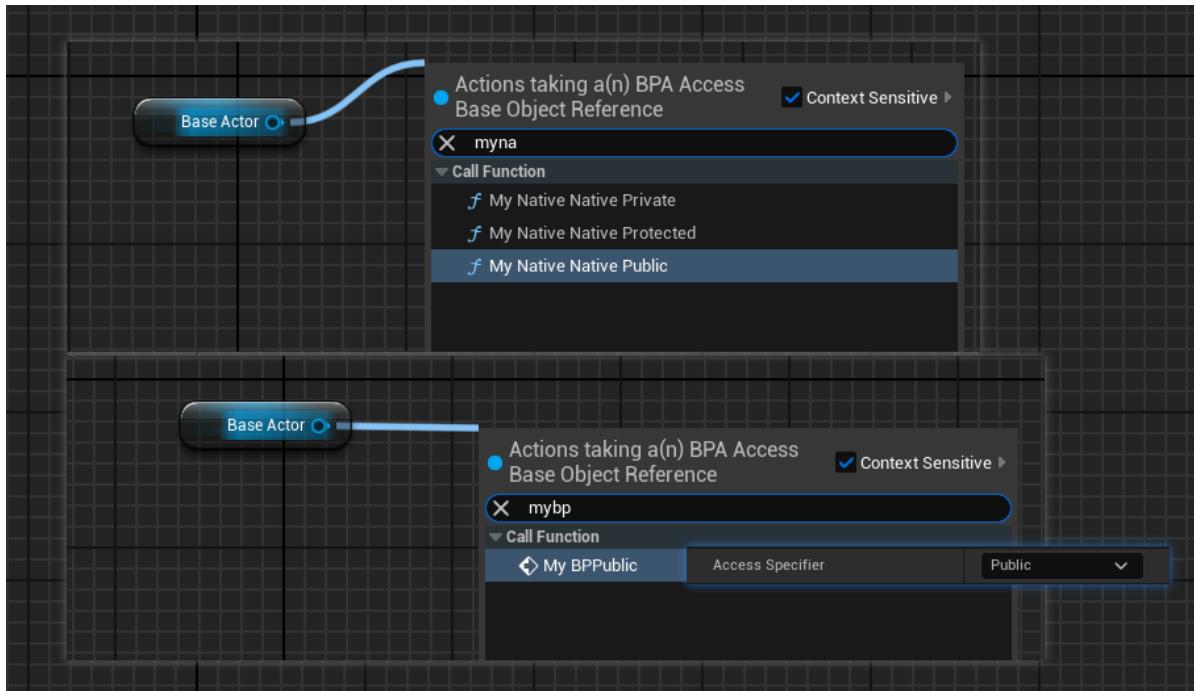


蓝图中的子类 (BPA_Access_Child继承自BPA_Access_Base) 效果：

可见MyNative函数的访问一样。而MyBPPrivate则不能被调用了，这和我们预想的规则一样。



而在外部类中(BPA_Access_Other, 继承自Actor), 通过BPA_Access_Base或BPA_Access_Child对象实例访问函数的时候, 发现带有BlueprintProtected和BlueprintPrivate都不能被调用。BP的函数也只有AccessSpecifier为默认Public的可以调用。这个规则也很符合预期。



原理：

在蓝图右键上是否可以选择该函数的过滤逻辑：

如果是static函数，则总是可以。否则必须没有BlueprintProtected或BlueprintPrivate才可以是Public可以被选择出来的。

如果是Private，则外部类必须是定义的类本身。

如果是Protected，则外部类只需要是定义的类或子类。

```
static bool BlueprintActionFilterImpl::IsFieldInaccessible(FBlueprintActionFilter
const& Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool const bIsProtected =
Field.HasMetaData(FBlueprintMetadata::MD_Protected);
    bool const bIsPrivate =
Field.HasMetaData(FBlueprintMetadata::MD_Private);
    bool const bIsPublic = !bIsPrivate && !bIsProtected;

    if( !bIsPublic )
    {
        UClass const* ActionOwner = BlueprintAction.GetOwnerClass();
        for (UBlueprint const* Blueprint : FilterContext.Blueprints)
        {
            UClass const* BpClass =
GetAuthoritativeBlueprintClass(Blueprint);
            if (!ensureMsgf(BpClass != nullptr
                , TEXT("Unable to resolve IsFieldInaccessible() - Blueprint
(%s) missing an authoritative class (skel: %s, generated: %s, parent: %s)")
                , *Blueprint->GetName()
                , Blueprint->SkeletonGeneratedClass ? *Blueprint-
>SkeletonGeneratedClass->GetName() : TEXT("[NULL]")
                , Blueprint->GeneratedClass ? *Blueprint->GeneratedClass-
>GetName() : TEXT("[NULL]")
                , Blueprint->ParentClass ? *Blueprint->ParentClass->GetName()
: TEXT("[NULL]")))
            {
                continue;
            }

            // private functions are only accessible from the class they
belong to
            if (bIsPrivate && !IsClassOfType(BpClass, ActionOwner,
/*bNeedsExactMatch =*/true))
            {
                bIsFilteredOut = true;
                break;
            }
            else if (bIsProtected && !IsClassOfType(BpClass, ActionOwner))
            {
                bIsFilteredOut = true;
                break;
            }
        }
    }

bool UEdGraphSchema_K2::ClassHasBlueprintAccessibleMembers(const UClass* InClass)
const
```

```

{
    // @TODO Don't show other blueprints yet...
    UBlueprint* ClassBlueprint = UBlueprint::GetBlueprintFromClass(InClass);
    if (!InClass->HasAnyClassFlags(CLASS_Deprecated | CLASS_NewerVersionExists)
&& (ClassBlueprint == NULL))
    {
        // Find functions
        for (TFieldIterator<UFunction> FunctionIt(InClass,
EFieldIteratorFlags::IncludeSuper); FunctionIt; ++FunctionIt)
        {
            UFunction* Function = *FunctionIt;
            const bool bIsBlueprintProtected = Function-
>GetBoolMetaData(FBlueprintMetadata::MD_Protected);
            const bool bHidden =
FObjectEditorUtils::IsFunctionHiddenFromClass(Function, InClass);
            if (UEdGraphSchema_K2::CanUserKismetCallFunction(Function) &&
!bIsBlueprintProtected && !bHidden)
            {
                return true;
            }
        }

        // Find vars
        for (TFieldIterator<FProperty> PropertyIt(InClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;
            if (CanUserKismetAccessVariable(Property, InClass, CannotBeDelegate))
            {
                return true;
            }
        }
    }

    return false;
}

```

在BP中定义的函数如果通过AccessSpecifier设置为Protected或Private，也会相应把该函数加上FUNC_Protected或FUNC_Private。从而实际上影响该函数的作用域。但源码中很多判断会先判断是否Native函数，如果是就不继续做限制。因此我们可以理解这是UE机制的有意为之，故意不把C++里的protected和private作用域算进去，而要求你必须自己手动显式的写上BlueprintProtected或BlueprintPrivate，这样避免有时的模糊不清。

```

bool UEdGraphSchema_K2::CanFunctionBeUsedInGraph(const UClass* InClass, const
UFunction* InFunction, const UEdGraph* InDestGraph, uint32
InAllowedFunctionTypes, bool bInCalledForEach, FText* OutReason) const
{
    const bool bIsNotNative =
!FBlueprintEditorUtils::IsNativeSignature(InFunction);
    if(bIsNotNative)
    {
        // Blueprint functions visibility flags can be enforced in blueprints -
native functions
        // are often using these flags to only hide functionality from other
native functions:
    }
}

```

```

        const bool bIsProtected = (InFunction->FunctionFlags & FUNC_Protected) != 0;
    }

bool UK2Node_CallFunction::IsActionFilteredOut(FBlueprintActionFilter const& Filter)
{
    bool bIsFilteredOut = false;
    for(UEdGraph* TargetGraph : Filter.Context.Graphs)
    {
        bIsFilteredOut |= !CanPasteHere(TargetGraph);
    }

    if(const UFunction* TargetFunction = GetTargetFunction())
    {
        const bool bIsProtected = (TargetFunction->FunctionFlags & FUNC_Protected) != 0;
        const bool bIsPrivate = (TargetFunction->FunctionFlags & FUNC_Private) != 0;
        const UClass* OwningClass = TargetFunction->GetOwnerClass();
        if( (bIsProtected || bIsPrivate) && !FBlueprintEditorUtils::IsNativeSignature(TargetFunction) && OwningClass)
        {
            OwningClass = OwningClass->GetAuthoritativeClass();
            // we can filter private and protected blueprints that are unrelated:
            bool bAccessibleInAll = true;
            for (const UBlueprint* Blueprint : Filter.Context.Blueprints)
            {
                UClass* AuthoritativeClass = Blueprint->GeneratedClass;
                if(!AuthoritativeClass)
                {
                    continue;
                }

                if(bIsPrivate)
                {
                    bAccessibleInAll = bAccessibleInAll && AuthoritativeClass == OwningClass;
                }
                else if(bIsProtected)
                {
                    bAccessibleInAll = bAccessibleInAll && AuthoritativeClass->IsChildOf(OwningClass);
                }
            }

            if(!bAccessibleInAll)
            {
                bIsFilteredOut = true;
            }
        }
    }

    return bIsFilteredOut;
}

```

作用在属性上：

作用在属性上时，标明该属性只能在本类或派生类里进行读写，但不能在别的蓝图类里访问。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Access :public AActor
{
public:
    GENERATED_BODY()
public:
    //BlueprintProtected = true, Category = MyFunction_Access,
    ModuleRelativePath = Function/MyFunction_Access.h
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, meta = (BlueprintProtected = "true"))
    int32 MyNativeInt_HasProtected;

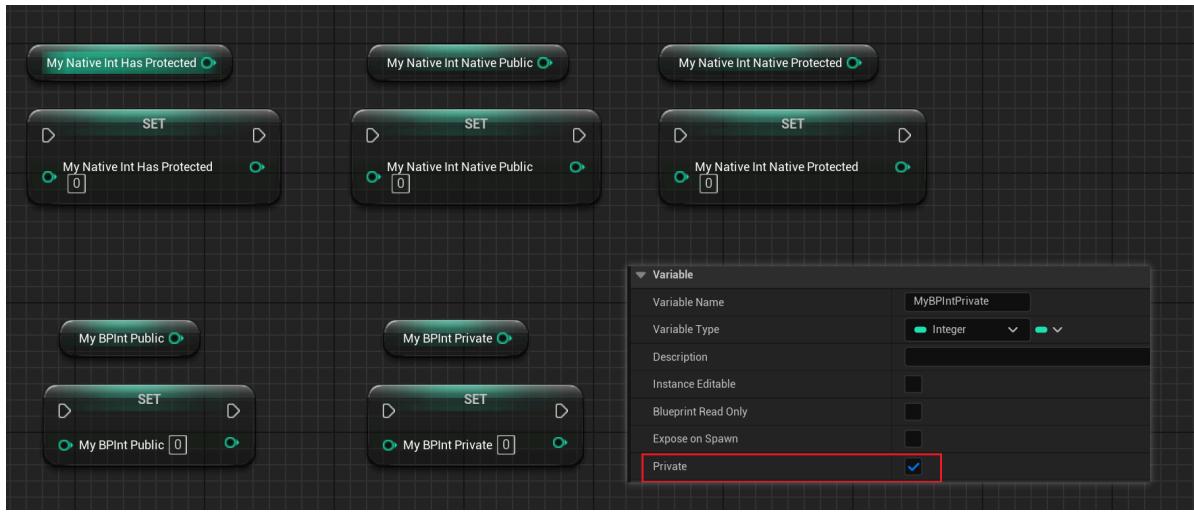
    //BlueprintPrivate = true, Category = MyFunction_Access, ModuleRelativePath
    = Function/MyFunction_Access.h
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, meta = (BlueprintPrivate = "true"))
    int32 MyNativeInt_HasPrivate;

public:
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite)
    int32 MyNativeInt_NativePublic;
protected:
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_Protected | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierProtected
    UPROPERTY(BlueprintReadOnly)
    int32 MyNativeInt_NativeProtected;
private:
    //CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    //error : BlueprintReadWrite should not be used on private members
    UPROPERTY(EditAnywhere)
    int32 MyNativeInt_NativePrivate;
};
```

蓝图效果：

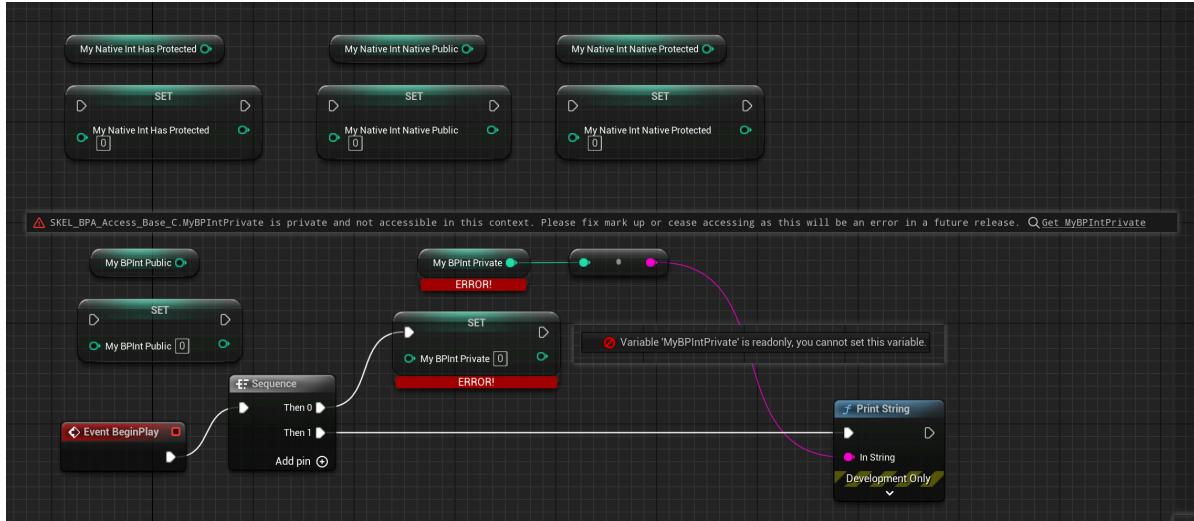
在其子类BPA_Access_Base测试，发现除了MyNativeInt_HasPrivate都可以访问。这符合逻辑，毕竟Private的含义就是只有在本类才可以访问。

而在本蓝图类定义的MyBPIIntPrivate因为勾上了Private，会导致该属性增加了BlueprintPrivate = true的meta，但因为是本类里定义的，所以在本类里也依然可以读写访问。



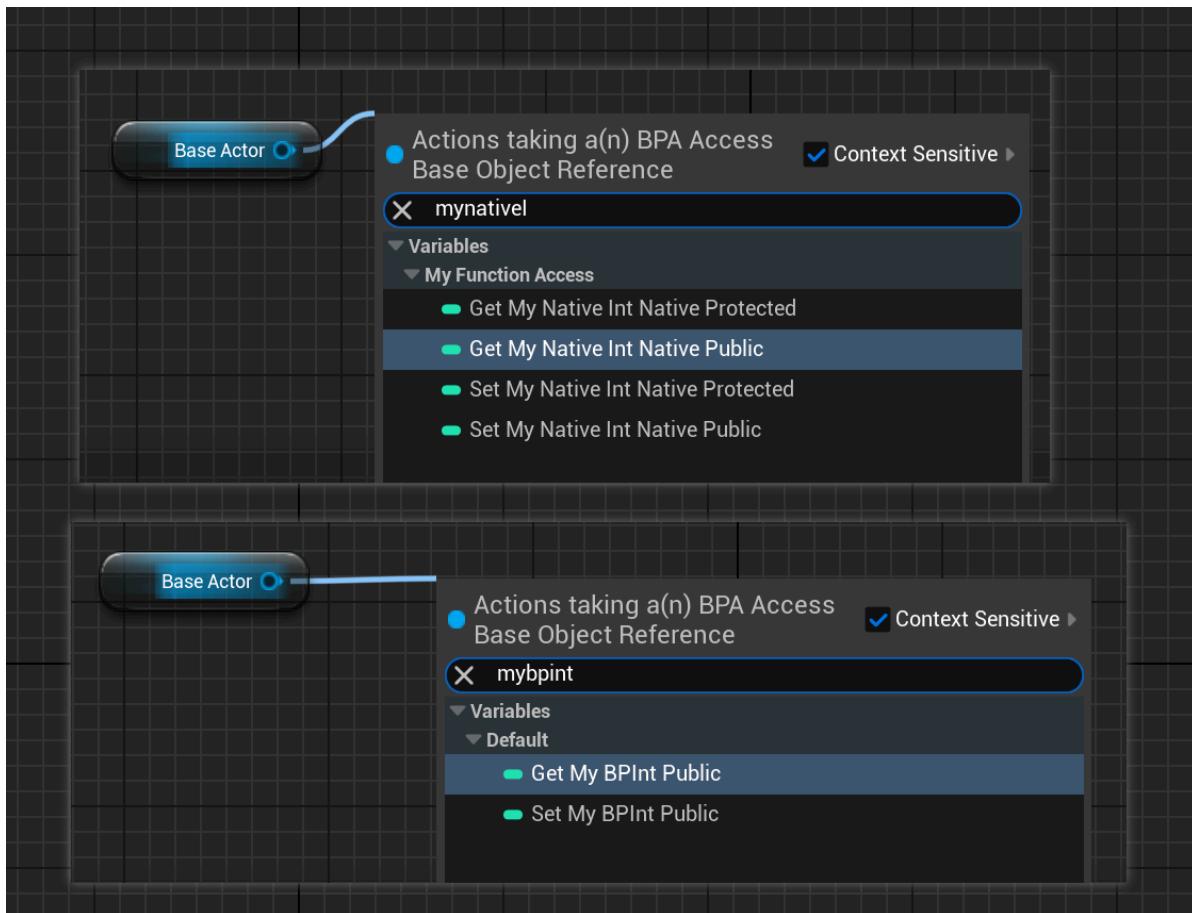
继续在蓝图中的子类 (BPA_Access_Child继承自BPA_Access_Base) 效果：

Protected的属性依然都可以访问，但是MyBPIntPrivate属性因为是Private的，因此都不能读写，如果强制粘贴节点，会在编译的时候报错。Private的含义是只在本类中才可以访问。



而在外部类中(BPA_Access_Other, 继承自Actor), 通过BPA_Access_Base或BPA_Access_Child对象实例访问属性的时候：带有BlueprintProtected和BlueprintPrivate都不能访问。而C++中的protected修饰并无影响。

而MyBPIntPrivate因为是Private所以不能访问。



原理：

在源码里搜索CPF_NativeAccessSpecifierProtected，发现并无使用的地方。

而CPF_NativeAccessSpecifierPrivate只在IsPropertyParams中引用，后者也只在蓝图编译检测线程安全的时候被检测到。因此CPF_NativeAccessSpecifierPrivate也实际上并无真正的被用来做作用域的限制。

综合二者，这也是在C++中protected和private并不在蓝图中造成影响的原因。但UHT会阻止private变量上的BlueprintReadWrite或BlueprintReadOnly，造成事实上的无法在蓝图中访问，达成了无法在蓝图子类里访问C++基类private变量的效果。

因此实际上在蓝图中的变量作用域控制，采用的元数据BlueprintProtected 和BlueprintPrivate，在蓝图右键能否创建属性读写节点的逻辑在上面的BlueprintActionFilterImpl::IsFieldInaccessible函数中体现。而编译的时候判断一个属性是否可读写的逻辑在IsPropertyParamsWritableInBlueprint和 IsPropertyParamsReadableInBlueprint这两个函数，如果最终的状态结果是Private，则说明不可访问。在 UK2Node_VariableGet和UK2Node_VariableSet的ValidateNodeDuringCompilation，会检测出来并报错。

```

bool FBlueprintEditorUtils::IsPropertyParamsPrivate(const FPropertyParams*PropertyParams)
{
    returnPropertyParams->HasAnyPropertyParams(CPF_NativeAccessSpecifierPrivate) ||
PropertyParams->GetBoolMetaData(FBlueprintMetadata::MD_Private);
}

FBlueprintEditorUtils::EPropertyParamsWritableState
FBlueprintEditorUtils::IsPropertyParamsWritableInBlueprint(const UBlueprint* Blueprint,
const FPropertyParams*PropertyParams)
{
    if(Properties)
    {
}

```

```

    if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
    {
        return EPropertyWritableState::NotBlueprintVisible;
    }
    if (Property->HasAnyPropertyFlags(CPF_BlueprintReadOnly))
    {
        return EPropertyWritableState::BlueprintReadOnly;
    }
    if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
    {
        const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
        if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
        {
            return EPropertyWritableState::Private;
        }
    }
}
return EPropertyWritableState::Writable;
}

FBlueprintEditorUtils::EPropertyReadableState
FBlueprintEditorUtils::IsPropertyReadableInBlueprint(const UBlueprint* Blueprint,
const FProperty* Property)
{
    if (Property)
    {
        if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
        {
            return EPropertyReadableState::NotBlueprintVisible;
        }
        if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
        {
            const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
            if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
            {
                return EPropertyReadableState::Private;
            }
        }
    }
}
return EPropertyReadableState::Readable;
}

```

BlueprintSetter

- 功能描述:** 采用一个自定义的set函数来读取。
会默认设置BlueprintReadWrite.
- 使用位置:** UFUNCTION, UPROPERTY
- 引擎模块:** Blueprint
- 元数据类型:** string="abc"
- 关联项:**

UFUNCTION: BlueprintSetter

UPROPERTY: BlueprintSetter

- 常用程度： ★★★

BlueprintThreadSafe

- **功能描述：** 用在类上或函数上，标记类里的函数都是线程安全的。
这样就可以在动画蓝图等非游戏线程被调用了。
- **使用位置：** UCLASS, UFUNCTION
- **引擎模块：** Blueprint
- **元数据类型：** bool
- **限制类型：** 从实践上，类一般是BlueprintFunctionLibrary
- **关联项：** NotBlueprintThreadSafe
- **常用程度：** ★★★

动画蓝图的AimGraph默认是开启线程安全Update的。设置在ClassSettings里（默认是打开的）

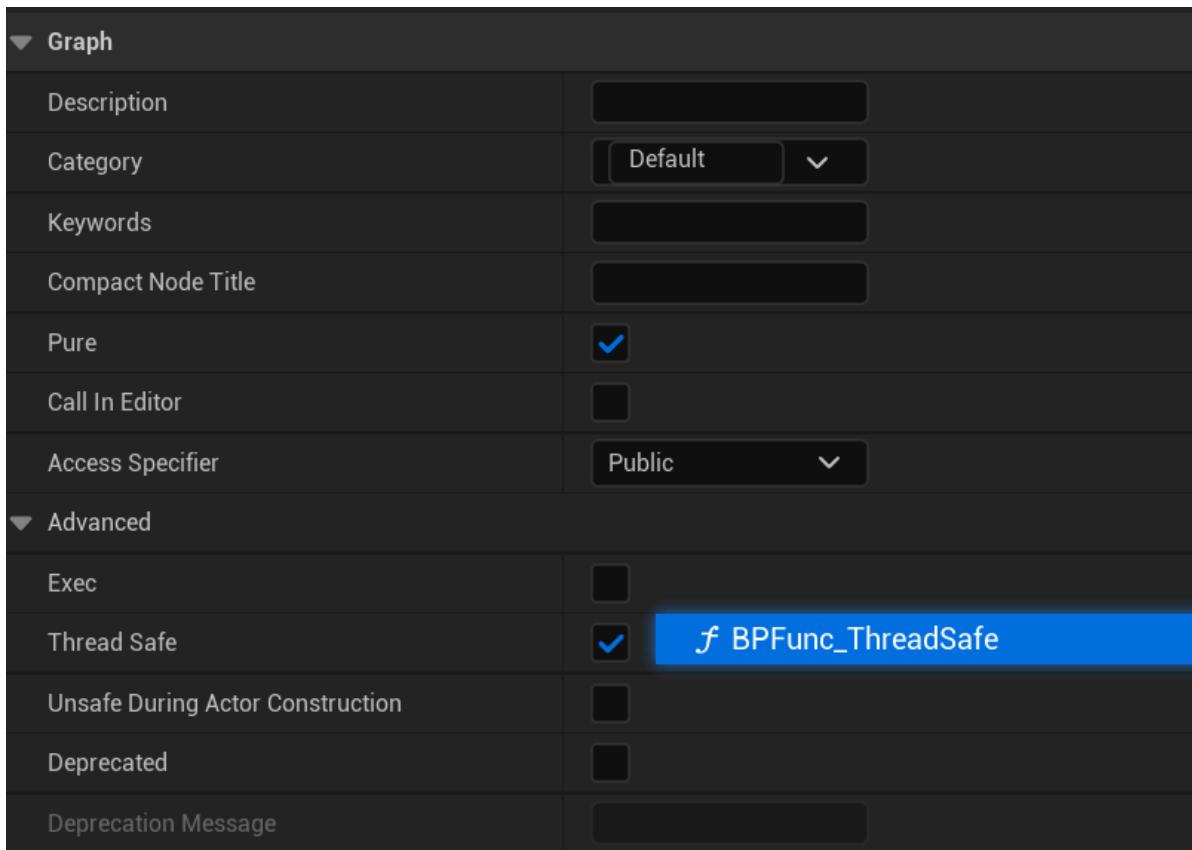


可参考官方文档的**CPU Thread Usage and Performance**这一节

[Graphing in Animation Blueprints](#)

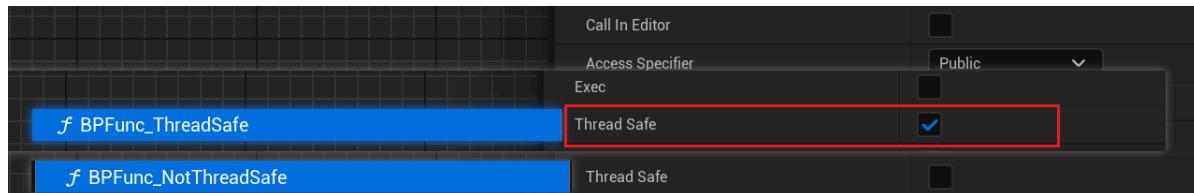
因此AimGraph里的函数要求都得是线程安全的。你的C++函数或者是蓝图里蓝图库里的函数都需要手动标记为ThreadSafe，默认不带ThreadSafe标记的都是不线程安全的。

在蓝图里，如果在蓝图函数面板中勾上ThreadSafe，这个函数的对象会设置bThreadSafe=True，从而在编译生成的BlueprintGeneratedClass上面设置(BlueprintThreadSafe = true)

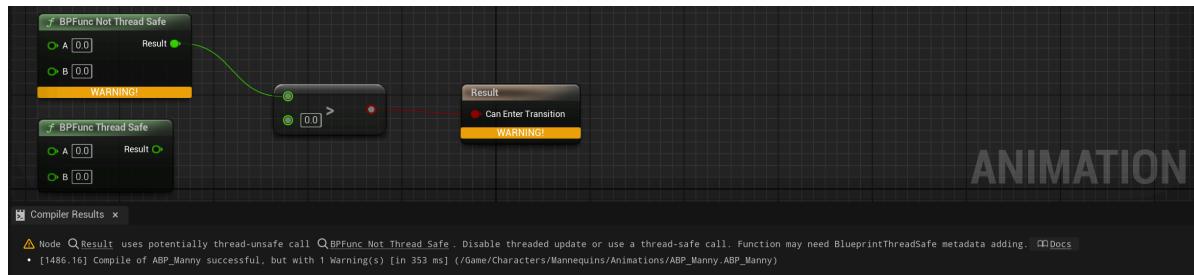


测试蓝图函数库：

同样的函数，一个打开ThreadSafe，一个没有。没有的那个函数在动画蓝图的AnimGraph里使用的时候，在编译的时候就会触发警告。



测试结果：



在C++里，C++的测试代码：

```
//(BlueprintThreadSafe = , IncludePath = Class/Blueprint/MyClass_Threadsafe.h,
ModuleRelativePath = Class/Blueprint/MyClass_Threadsafe.h)
UCLASS(meta=(BlueprintThreadSafe))
class INSIDER_API UMyBlueprintFunctionLibrary_Threadsafe : public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure)
    static float MyFunc_ClassThreadsafe_Default(float value) {return value+100;}

    //(ModuleRelativePath = Class/Blueprint/MyClass_ThreadSafe.h,
NotBlueprintThreadSafe = )
    UFUNCTION(BlueprintPure,meta=(NotBlueprintThreadSafe))
    static float MyFunc_ClassThreadSafe_FuncNotThreadSafe(float value) {return
value+100;}
};

UCLASS()
class INSIDER_API UMyBlueprintFunctionLibrary_NoThreadSafe : public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    //(BlueprintThreadSafe = , ModuleRelativePath =
Class/Blueprint/MyClass_ThreadSafe.h)
    UFUNCTION(BlueprintPure,meta=(BlueprintThreadSafe))
    static float MyFunc_ClassDefault_FuncThreadSafe(float value) {return
value+100;}

    //(ModuleRelativePath = Class/Blueprint/MyClass_ThreadSafe.h,
NotBlueprintThreadSafe = )
```

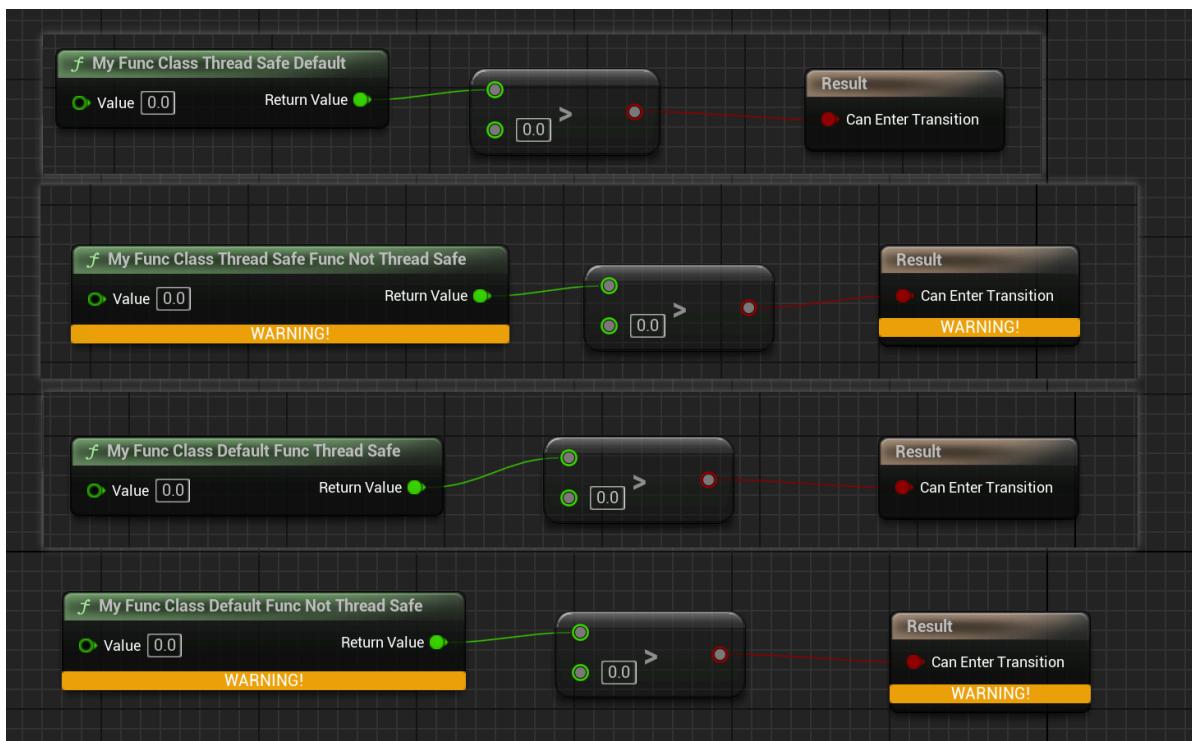
```

UFUNCTION(BlueprintPure, meta=(NotBlueprintThreadSafe))
static float MyFunc_ClassDefault_FuncNotThreadSafe(float value) {return
value+100;}
};

UCLASS()
class INSIDER_API UMyBlueprintFunctionLibrary_Default : public
UBlueprintFunctionLibrary
{
GENERATED_BODY()
public:
UFUNCTION(BlueprintPure)
static float MyFunc_ClassDefault_FuncDefault(float value) {return value+100;}
};

```

动画蓝图的测试效果：



解析原理：

```

bool FBlueprintEditorUtils::HasFunctionBlueprintThreadSafeMetaData(const
UFunction* InFunction)
{
    if(InFunction)
    {
        const bool bHasThreadSafeMetaData = InFunction-
>HasMetaData(FBlueprintMetadata::MD_ThreadSafe);
        const bool bHasNotThreadSafeMetaData = InFunction-
>HasMetaData(FBlueprintMetadata::MD_NotThreadSafe);
        const bool bClassHasThreadSafeMetaData = InFunction->GetOwnerClass() &&
InFunction->GetOwnerClass()->HasMetaData(FBlueprintMetadata::MD_ThreadSafe);

        // Native functions need to just have the correct class/function metadata
    }
}

```

```

        const bool bThreadsafeNative = InFunction-
>HasAnyFunctionFlags(FUNC_Native) && (bHasThreadSafeMetaData ||
(bClassHasThreadSafeMetaData && !bHasNotThreadSafeMetaData));

        // Script functions get their flag propagated from their entry point, and
        // don't pay heed to class metadata
        const bool bThreadsafeScript = !InFunction-
>HasAnyFunctionFlags(FUNC_Native) && bHasThreadSafeMetaData;

        return bThreadsafeNative || bThreadsafeScript;
    }

    return false;
}

```

可以从逻辑上看出，如果在UCLASS上带上了BlueprintThreadSafe，则其内部的函数就默认是线程安全，除非特意手动加上NotBlueprintThreadSafe来排除。而如果UCLASS上没有标记，则需一个个手动的在UFUNCTION上标记BlueprintThreadSafe。两种方式都可以。

注意UCLASS(meta=(NotBlueprintThreadSafe))这种是没有被识别判断的，因此并没有什么意义。

BlueprintType

- 功能描述：**表明可以作为一个蓝图变量
- 使用位置：** UCLASS, UENUM, UINTERFACE, USTRUCT
- 引擎模块：** Blueprint
- 元数据类型：** bool
- 关联项：**

UCLASS: Blueprintable, NotBlueprintable, BlueprintType, NotBlueprintType

Meta: BlueprintInternalUseOnly, BlueprintInternalUseOnlyHierarchical

UENUM: BlueprintType

UFUNCTION: BlueprintInternalUseOnly

UINTERFACE: Blueprintable, NotBlueprintable

USTRUCT: BlueprintInternalUseOnly, BlueprintType

- 常用程度：** ★★★★☆

CallableWithoutWorldContext

- 功能描述：**让函数也可以脱离WorldContextObject而使用
- 使用位置：** UFUNCTION
- 元数据类型：** bool
- 关联项：** WorldContext

让函数也可以脱离WorldContextObject而使用。

CallableWithoutWorldContext 是配合WorldContext或DefaultToSelf来使用，这二者会使得一个函数会要求外部传入一个WorldContext对象才能调用。因此这种函数在没有实现GetWorld的Object子类里就不能调用。但有时某些函数又不一定必须得有WorldContextObject才能工作，比如PrintString或VisualLogger里的函数。

测试代码：

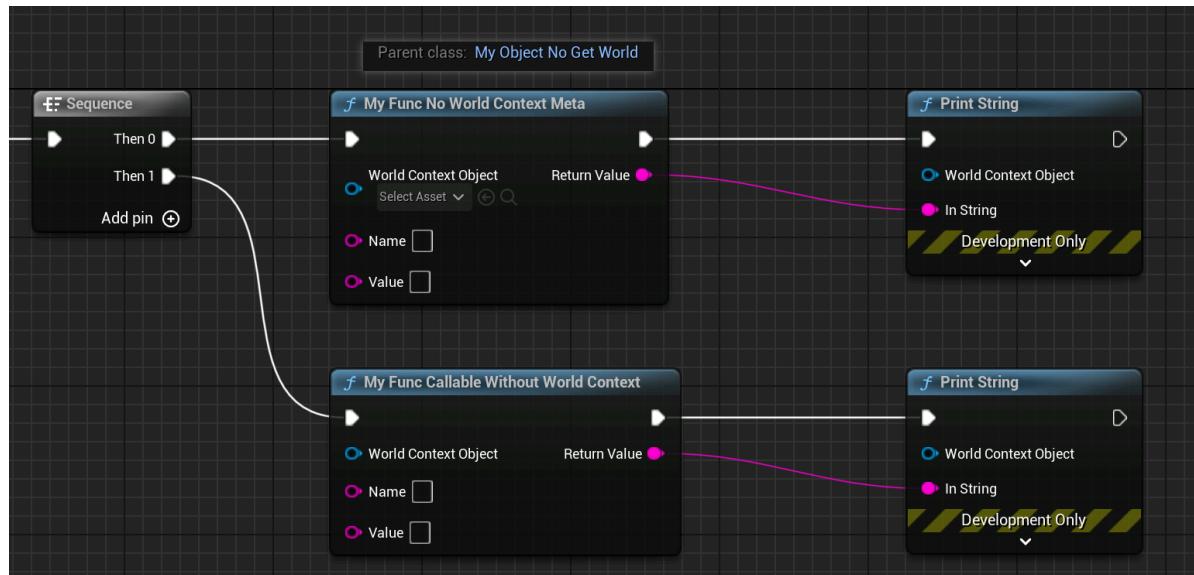
```
UFUNCTION(BlueprintPure, meta = (WorldContext = "WorldContextObject"))
static FString MyFunc_HasWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

UFUNCTION(BlueprintCallable, meta = (WorldContext =
"WorldContextObject", CallablewithoutWorldContext))
static FString MyFunc_CallablewithoutWorldContext(const UObject*
WorldContextObject, FString name, FString value);

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyObject_NoGetWorld :public UObject
{
    GENERATED_BODY()
};
```

蓝图测试效果：

在UMyObject_NoGetWorld 的子类内，MyFunc_HasWorldContextMeta不能调用，因为其外部类必须提供WorldContextObject。而MyFunc_CallableWithoutWorldContext可以调用，可以接受不提供WorldContextObject。



源码里典型的应用是：

```
UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
CallablewithoutWorldContext, Keywords = "log print", AdvancedDisplay = "2",
DevelopmentOnly), Category="Development")
static ENGINE_API void PrintString(const UObject* WorldContextObject, const
FString& InString = FString(TEXT("Hello")), bool bPrintToScreen = true, bool
bPrintToLog = true, FLinearColor TextColor = FLinearColor(0.0f, 0.66f, 1.0f),
float Duration = 2.f, const FName Key = NAME_None);
```

CallInEditor

- **功能描述:** 可以在Actor的细节面板上作为一个按钮来调用该函数。
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **关联项:**
UFUNCTION: CallInEditor
- **常用程度:** ★★★★☆

CannotImplementInterfaceInBlueprint

- **功能描述:** 指定该接口不能在蓝图中实现
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **关联项:**
UINTERFACE: NotBlueprintable
- **常用程度:** ★★★

和UINTERFACE(NotBlueprintable)的效果一样，指定不能在蓝图中继承

CommutativeAssociativeBinaryOperator

- **功能描述:** 标记一个二元运算函数的运算支持交换律和结合律，在蓝图节点上增加一个“+”引脚，允许动态的直接添加多个输入值。
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **常用程度:** ★★★★☆

标记一个二元运算函数的运算支持交换律和结合律，在蓝图节点上增加一个“+”引脚，允许动态的直接添加多个输入值。而不需要自己手动创建多个本函数节点来运算，这是蓝图提供的便利功能之一。

CommutativeAssociativeBinaryOperator的限制是函数必须是BlueprintPure并且有两个参数。否则会产生编译报错或功能失效。

测试代码：

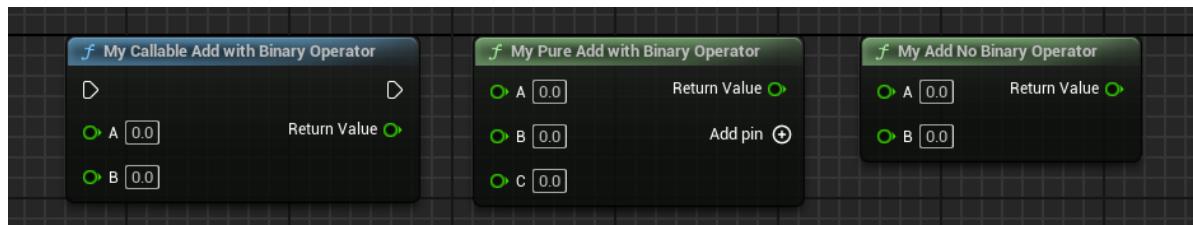
```
UFUNCTION(BlueprintCallable, meta = (CommutativeAssociativeBinaryOperator))
static float My_CallableAdd_WithBinaryOperator(float A, float B) { return A +
B; }

UFUNCTION(BlueprintPure, meta = (CommutativeAssociativeBinaryOperator))
static float My_PureAdd_WithBinaryOperator(float A, float B) { return A + B;
}

UFUNCTION(BlueprintPure, meta = ())
static float My_Add_NoBinaryOperator(float A, float B) { return A + B; }

// error : Commutative associative binary operators must have exactly 2
parameters of the same type and a return value.
//UFUNCTION(BlueprintPure, meta = (CommutativeAssociativeBinaryOperator))
// static float My_PureAdd3_WithBinaryOperator(float A, float B, float C) {
return A + B+C; }
```

蓝图效果：



原理：

标记CommutativeAssociativeBinaryOperator的函数会采用UK2Node_CollectiveAssociativeBinaryOperator来生成节点。这个二元运算满足交换率和结合律，因此可以通过多次的调用本函数来支持多个输入值的运算。在UK2Node_CollectiveAssociativeBinaryOperator展开的时候，会创建中间的多个UK2Node_CollectiveAssociativeBinaryOperator来形成调用序列。

在源码中的应用是一些二元运算，在UKismetMathLibrary中有大量的运用，典型的比如FVector的互相运算。

```
void
UK2Node_CollectiveAssociativeBinaryOperator::ExpandNode(FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
    Super::ExpandNode(CompilerContext, SourceGraph);

    if (NumAdditionalInputs > 0)
    {
        const UEdGraphSchema_K2* Schema = CompilerContext.GetSchema();

        UEdGraphPin* LastOutPin = NULL;
        const UFunction* const Function = GetTargetFunction();

        const UEdGraphPin* SrcOutPin = FindOutPin();
        const UEdGraphPin* SrcSelfPin = FindSelfPin();
```

```

        UEdGraphPin* SrcFirstInput = GetInputPin(0);
        check(SrcFirstInput);

        for(int32 PinIndex = 0; PinIndex < Pins.Num(); PinIndex++)
        {
            UEdGraphPin* CurrentPin = Pins[PinIndex];
            if( (CurrentPin == SrcFirstInput) || (CurrentPin == SrcOutPin) ||
            (srcSelfPin == CurrentPin) )
            {
                continue;
            }

            UK2Node_CollectiveAssociativeBinaryOperator* NewOperator =
            SourceGraph->CreateIntermediateNode<UK2Node_CollectiveAssociativeBinaryOperator>()
            NewOperator->SetFromFunction(Function);
            NewOperator->AllocateDefaultPins();

            CompilerContext.MessageLog.NotifyIntermediateObjectCreation(NewOperator, this);

            UEdGraphPin* NewOperatorInputA = NewOperator->GetInputPin(0);
            check(NewOperatorInputA);
            if(LastOutPin)
            {
                Schema->TryCreateConnection(LastOutPin, NewOperatorInputA);
            }
            else
            {
                // handle first created node (SrcFirstInput is skipped, and has
                no own node).
                CompilerContext.MovePinLinksToIntermediate(*SrcFirstInput,
                *NewOperatorInputA);
            }

            UEdGraphPin* NewOperatorInputB = NewOperator->GetInputPin(1);
            check(NewOperatorInputB);
            CompilerContext.MovePinLinksToIntermediate(*CurrentPin,
            *NewOperatorInputB);

            LastOutPin = NewOperator->FindOutPin();
        }

        check(LastOutPin);

        UEdGraphPin* TrueOutPin = FindOutPin();
        check(TrueOutPin);
        CompilerContext.MovePinLinksToIntermediate(*TrueOutPin, *LastOutPin);

        BreakAllNodeLinks();
    }
}

```

CompactNodeTitle

- **功能描述:** 使得函数的展示形式变成精简压缩模式，同时指定一个新的精简的名字
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **常用程度:** ★★★

使得函数的展示形式变成精简压缩模式，同时指定一个新的精简的名字。注意到该模式下就会忽略 DisplayName的数据。

测试代码：

```
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle =
"MyCompact", DisplayName="AnotherName"))
static int32 MyFunc_HasCompactNodeTitle(FString Name) {return 0;}

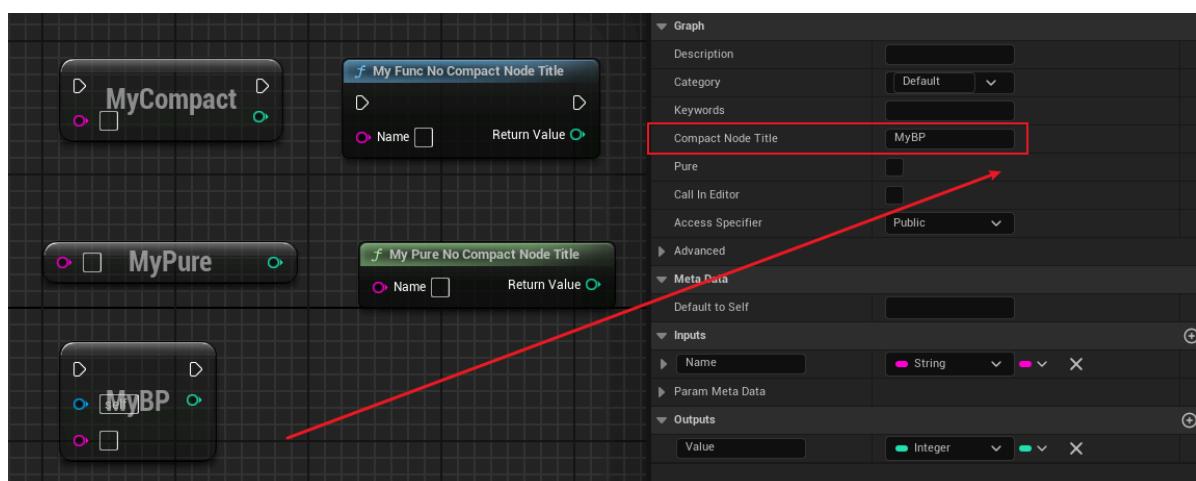
UFUNCTION(BlueprintCallable, meta = ())
static int32 MyFunc_NoCompactNodeTitle(FString Name) {return 0;}

UFUNCTION(BlueprintPure, meta = (CompactNodeTitle =
"MyPure", DisplayName="AnotherName"))
static int32 MyPure_HasCompactNodeTitle(FString Name) {return 0;}

UFUNCTION(BlueprintPure, meta = ())
static int32 MyPure_NoCompactNodeTitle(FString Name) {return 0;}
```

蓝图效果：

显示效果明显发生了变化。同时我们在蓝图里定义的函数也可以通过这个细节面板上的设置变成压缩模式展示。



原理：

```
bool UK2Node_CallFunction::ShouldDrawCompact(const UFunction* Function)
{
    return (Function != NULL) && Function->HasMetaData(FBlueprintMetadata::MD_CompactNodeTitle);
```

```

}

FString UK2Node_CallFunction::GetCompactNodeTitle(const UFunction* Function)
{
    static const FString ProgrammerMultiplicationSymbol = TEXT("*");
    static const FString CommonMultiplicationSymbol = TEXT("\u00d7");

    static const FString ProgrammerDivisionSymbol = TEXT("/");
    static const FString CommonDivisionSymbol = TEXT("\u2077");

    static const FString ProgrammerConversionSymbol = TEXT("->");
    static const FString CommonConversionSymbol = TEXT("\u2022");

    const FString& OperatorTitle = Function-
>GetMetaData(FBlueprintMetadata::MD_CompactNodeTitle);
    if (!OperatorTitle.IsEmpty())
    {
        if (OperatorTitle == ProgrammerMultiplicationSymbol)
        {
            return CommonMultiplicationSymbol;
        }
        else if (OperatorTitle == ProgrammerDivisionSymbol)
        {
            return CommonDivisionSymbol;
        }
        else if (OperatorTitle == ProgrammerConversionSymbol)
        {
            return CommonConversionSymbol;
        }
        else
        {
            return OperatorTitle;
        }
    }

    return Function->GetName();
}

```

CPP_Default_XXX

- **功能描述:** XXX=参数名字
- **使用位置:** UPARAM
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **常用程度:** ★★★★★

在UFUNCTION的meta上保存参数的默认值。

测试代码：

```
//(CPP_Default_intValue = 123, CPP_Default_intValue2 = 456, ModuleRelativePath =
Function/Param/MyFunction_TestParam.h)
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_DefaultInt2(int intValue=123,int intValue2=456);
```

在Meta里也可以直接写属性的默认值，如Duration。

```
UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=(Latent,
worldContext="WorldContextObject", LatentInfo="LatentInfo", Duration="0.2",
Keywords="sleep"))
static ENGINE_API void MyDelay(const UObject* WorldContextObject, float
Duration, struct FLatentActionInfo LatentInfo );
```

原理代码：

在UEdGraphSchema_K2::FindFunctionParameterDefaultValue里会尝试找该参数名称对应的Meta。如果找不到则会继续找CPP_Default_ParamName这个名称。然后设置到Pin->AutogeneratedDefaultValue

```
bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    FString ParamValue;
    if (K2Schema->FindFunctionParameterDefaultValue(Function, Param,
ParamValue))
    {
        K2Schema->SetPinAutogeneratedDefaultValue(Pin, ParamValue);
    }
    else
    {
        K2Schema->SetPinAutogeneratedDefaultValueBasedOnType(Pin);
    }
}
```

DefaultToSelf

- 功能描述：**用在函数上，指定一个参数的默认值为Self值
- 使用位置：**UFUNCTION
- 引擎模块：**Blueprint
- 元数据类型：**bool
- 常用程度：**★★★★★

使得在蓝图调用的时候更加的便利，当然也要根据这个函数的需要。

测试代码：

```
UCLASS()
class INSIDER_API UMyFunctionLibrary_SelfPinTest :public
UBlueprintFunctionLibrary
```

```

{
GENERATED_BODY()

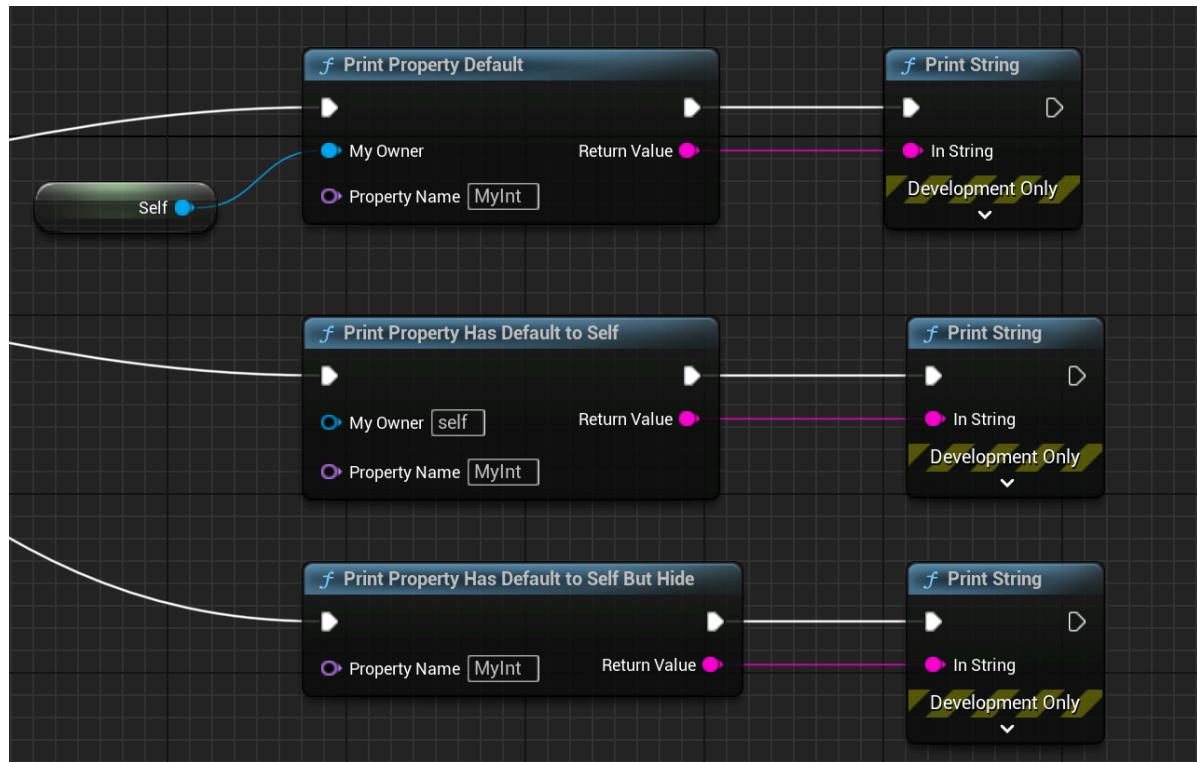
public:
UFUNCTION(BlueprintCallable)
static FString PrintProperty_Default(UObject* myOwner, FName propertyName);

UFUNCTION(BlueprintCallable, meta=(DefaultToSelf="myOwner"))
static FString PrintProperty_HasDefaultToSelf(UObject* myOwner, FName
propertyName);

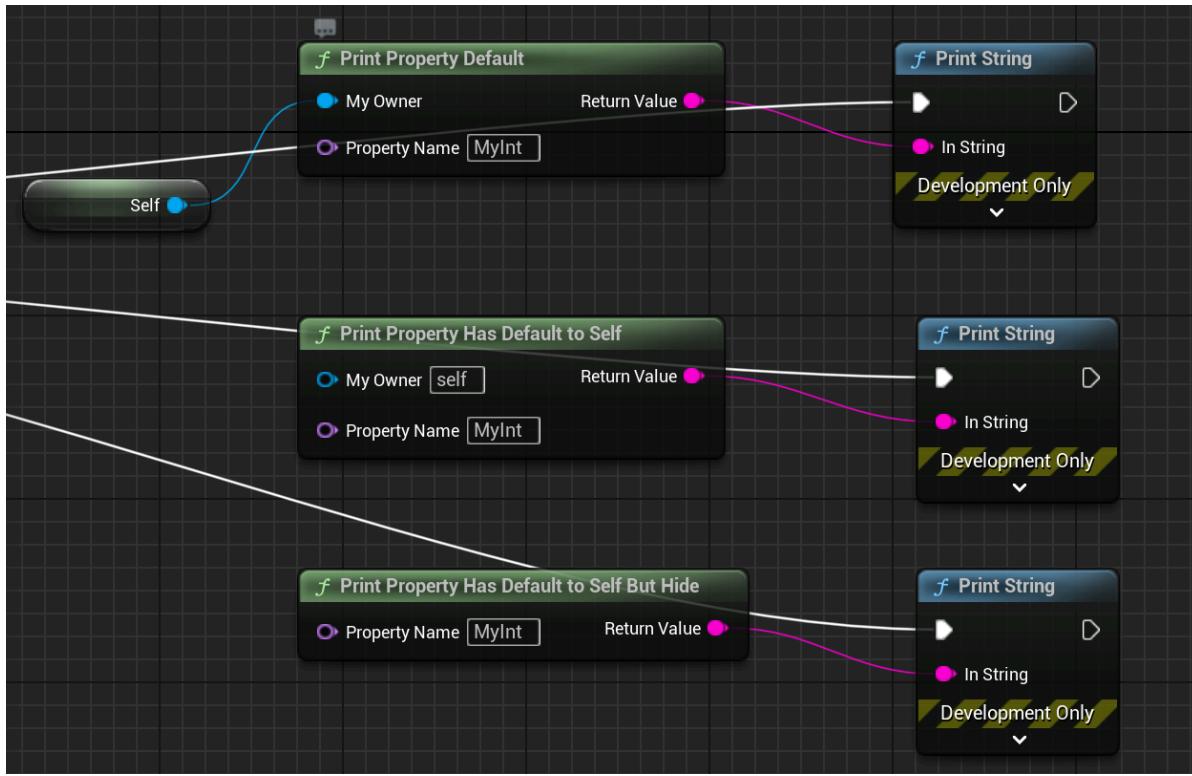
UFUNCTION(BlueprintCallable, meta=(DefaultToSelf="myOwner", hidePin="myOwner"))
static FString PrintProperty_HasDefaultToSelf_ButHide(UObject* myOwner, FName
propertyName);
};

```

蓝图里的节点，可以看出蓝图编译器会自动的把DefaultToSelf指定的函数参数，自动的赋值到Self，当然这个和手动的连到self本质是一样的。额外一点，可以通过HidePin再隐藏掉这个函数参数，这样就默认把该蓝图节点所在的蓝图对象（Self）当作第一个函数参数，显得更加简洁一些。



如果是BlueprintPure也是可以的：



原理：

蓝图中的函数调用在编译的时候，会自动的创建SelfPin（名字为Target）。如果该函数是静态函数或 HideSelfPin的标记，则会把SelfPin隐藏起来。其SelfPin的值就是当前蓝图运行时对象的值。因此 DefaultToSelf的效果就是蓝图系统会自动的把这个参数的值赋值为被调用处的蓝图运行时对象，相当于 C++ this指针的效果

```

bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    UEdGraphPin* SelfPin = CreateSelfPin(Function);
    // Renamed self pin to target
    SelfPin->PinFriendlyName = LOCTEXT("Target", "Target");
}

UEdGraphPin* FBlueprintNodeStatics::CreateSelfPin(UK2Node* Node, const UFunction* Function)
{
    // Chase up the function's Super chain, the function can be called on any
    // object that is at least that specific
    const UFunction* FirstDeclaredFunction = Function;
    while (FirstDeclaredFunction->GetSuperFunction() != nullptr)
    {
        FirstDeclaredFunction = FirstDeclaredFunction->GetSuperFunction();
    }

    // Create the self pin
    UClass* FunctionClass = CastChecked<UClass>(FirstDeclaredFunction-
>GetOuter());
    // we don't want blueprint-function target pins to be formed from the
    // skeleton class (otherwise, they could be incompatible with other pins
    // that represent the same type)... this here could lead to a compiler
    // warning (the GeneratedClass could not have the function yet), but in
    // that, the user would be reminded to compile the other blueprint
}

```

```

    if (FunctionClass->ClassGeneratedBy)
    {
        FunctionClass = FunctionClass->GetAuthoritativeClass();
    }

    UEdGraphPin* SelfPin = NULL;
    if (FunctionClass == Node->GetBlueprint()->GeneratedClass)
    {
        // This means the function is defined within the blueprint, so the pin
        // should be a true "self" pin
        SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Object,
        UEdGraphSchema_K2::PSC_Self, nullptr, UEdGraphSchema_K2::PN_Self);
    }
    else if (FunctionClass->IsChildOf(UInterface::StaticClass()))
    {
        SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Interface,
        FunctionClass, UEdGraphSchema_K2::PN_Self);
    }
    else
    {
        // This means that the function is declared in an external class, and
        // should reference that class
        SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Object,
        FunctionClass, UEdGraphSchema_K2::PN_Self);
    }
    check(SelfPin != nullptr);

    return SelfPin;
}

```

DisplayName

- 功能描述:** 此节点在蓝图中的命名将被此处提供的值所取代，而非代码生成的命名。
- 使用位置:** UCLASS, UENUM::UMETA, UFUNCTION, UPARAM, UPROPERTY
- 引擎模块:** Blueprint
- 元数据类型:** string="abc"
- 关联项:**
UPARAM: DisplayName
- 常用程度:** ★★★★☆

DontUseGenericSpawnObject

- 功能描述:** 阻止使用蓝图中的Generic Create Object节点来生成本类的对象。
- 使用位置:** UCLASS
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 限制类型:** 既非Actor又非ActorComponent的BlueprintType类时
- 常用程度:** ★★

用于阻止该类被通用的ConstructObject蓝图节点所构造出来。在源码里典型里使用例子是UDragDropOperation和UUserWidget，前者由UK2Node_CreateDragDropOperation这个专门的节点建出来（内部调用UWidgetBlueprintLibrary::CreateDragDropOperation），后者由CreateWidget创建。因此这种的典型用法是你自己再创建一个New的函数来自己创建该Object。

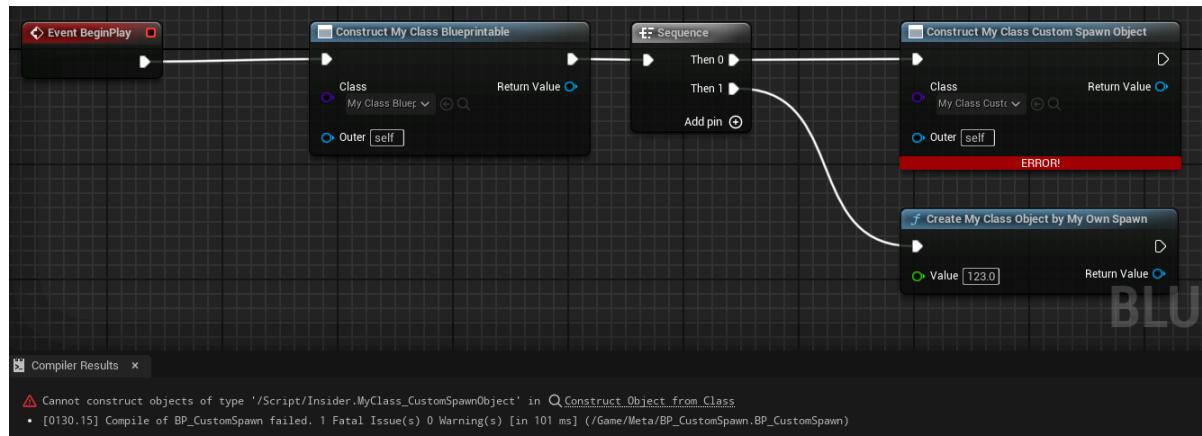
测试代码：

```
UCLASS(Blueprintable, meta=(DontUseGenericSpawnObject="true"))
class INSIDER_API UMyClass_CustomSpawnObject : public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;

    UFUNCTION(BlueprintCallable)
    static UMyClass_CustomSpawnObject* Create MyClassObjectByMyOwnSpawn(float value)
    {
        UMyClass_CustomSpawnObject* obj = NewObject<UMyClass_CustomSpawnObject>();
        obj->MyFloat = value;
        return obj;
    }
};
```

测试效果：



原理：

会提前验证是否包含DontUseGenericSpawnObject元数据，因为是采用GetBoolMetaData，因此必须写上="true"

```
struct FK2Node_GenericCreateObject_Utils
{
    static bool CanSpawnObjectOfClass(TSubclassOf<UObject> ObjectClass, bool bAllowAbstract)
    {
        // Initially include types that meet the basic requirements.
        // Note: CLASS_Deprecated is an inherited class flag, so any subclass of
        // an explicitly-deprecated class also cannot be spawned.
        bool bCanSpawnObject = (nullptr != *ObjectClass)
```

```

    && (bAllowAbstract || !ObjectClass->HasAnyClassFlags(CLASS_Abstract))
    && !ObjectClass->HasAnyClassFlags(CLASS_Deprecated |
CLASS_NewerVersionExists);

        // UObject is a special case where if we are allowing abstract we are
        // going to allow it through even though it doesn't have BlueprintType on it
        if (bCanSpawnObject && (!bAllowAbstract || (*ObjectClass != 
UObject::StaticClass())))
        {
            static const FName BlueprintTypeName(TEXT("BlueprintType"));
            static const FName NotBlueprintTypeName(TEXT("NotBlueprintType"));
            static const FName
DontUseGenericSpawnObjectName(TEXT("DontUseGenericSpawnObject"));

            auto IsClassAllowedLambda = [](const UClass* InClass)
            {
                return InClass != AActor::StaticClass()
                    && InClass != UActorComponent::StaticClass();
            };

            // Exclude all types in the initial set by default.
            bCanSpawnObject = false;
            const UClass* CurrentClass = ObjectClass;

            // Climb up the class hierarchy and look for "BlueprintType." If
            // "NotBlueprintType" is seen first, or if the class is not allowed, then stop
            // searching.
            while (!bCanSpawnObject && CurrentClass != nullptr && !CurrentClass-
>GetBoolMetaData(NotBlueprintTypeName) && IsClassAllowedLambda(CurrentClass))
            {
                // Include any type that either includes or inherits
                // 'BlueprintType'
                bCanSpawnObject = CurrentClass-
>GetBoolMetaData(BlueprintTypeName);

                // Stop searching if we encounter 'BlueprintType' with
                // 'DontUseGenericSpawnObject'
                if (bCanSpawnObject && CurrentClass-
>GetBoolMetaData(DontUseGenericSpawnObjectName))
                {
                    bCanSpawnObject = false;
                    break;
                }

                CurrentClass = CurrentClass->GetSuperclass();
            }

            // If we validated the given class, continue walking up the hierarchy
            // to make sure we exclude it if it's an Actor or ActorComponent derivative.
            while (bCanSpawnObject && CurrentClass != nullptr)
            {
                bCanSpawnObject &= IsClassAllowedLambda(CurrentClass);

                CurrentClass = CurrentClass->GetSuperclass();
            }
        }
    }
}

```

```
        return bCanSpawnObject;
    }
};
```

ExpandBoolAsExecs

- **功能描述:** 是ExpandEnumAsExecs的别名，完全等价其功能。
- **使用位置:** UFUNCTION
- **元数据类型:** string="abc"
- **关联项:** ExpandEnumAsExecs
- **常用程度:** ★★★★☆

ExpandEnumAsExecs

- **功能描述:** 指定多个enum或bool类型的函数参数，自动根据条目生成相应的多个输入或输出执行引脚，并根据实参值不同来相应改变控制流。
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** strings="a, b, c"
- **关联项:** ExpandBoolAsExecs
- **常用程度:** ★★★★☆

指定多个enum或bool类型的函数参数，自动根据条目生成相应的多个输入或输出执行引脚，并根据实参值不同来相应改变控制流。

支持改变输入和输出的Exec，输入Exec只可以一个，但是输出ExecEnum Pin可以多个。但是不能用在 BlueprintPure上（都没Exec引脚了）。

也可以通过' | '来分隔。

支持3种参数类型，enum class, TEnumAsByte[EMyExecPins2::Type](#)和bool，enum必须用UENUM标记。

引用类型的参数和返回值用作输出Pin，值类型的参数用作输入Pin。

可以用"ReturnValue"这个名字来指定使用返回值参数。

如果有多个输出Enum参数，会在函数的调用之后排成Sequecene来——分别根据输出Enum的值来触发输出Exec。

测试代码：

```
UENUM(BlueprintType)
enum class EMyExecPins : uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType)
```

```

namespace EMyExecPins2
{
    enum Type : int
    {
        Found,
        NotFound,
    };
}

UENUM(BlueprintType)
enum class EMyExecAnimalPins : uint8
{
    Cat,
    Dog,
};

public:
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyEnumAsExec_Output(FString Name, EMyExecPins& Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyEnumAsExec_Input(FString Name, TEnumAsByte<EMyExecPins2::Type> Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "ReturnValue"))
    static EMyExecPins MyEnumAsExec_Return(FString Name) { return EMyExecPins::First; }

public:
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyBoolAsExec_Output(FString Name, bool& Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyBoolAsExec_Input(FString Name, bool Pins) { return 0; }

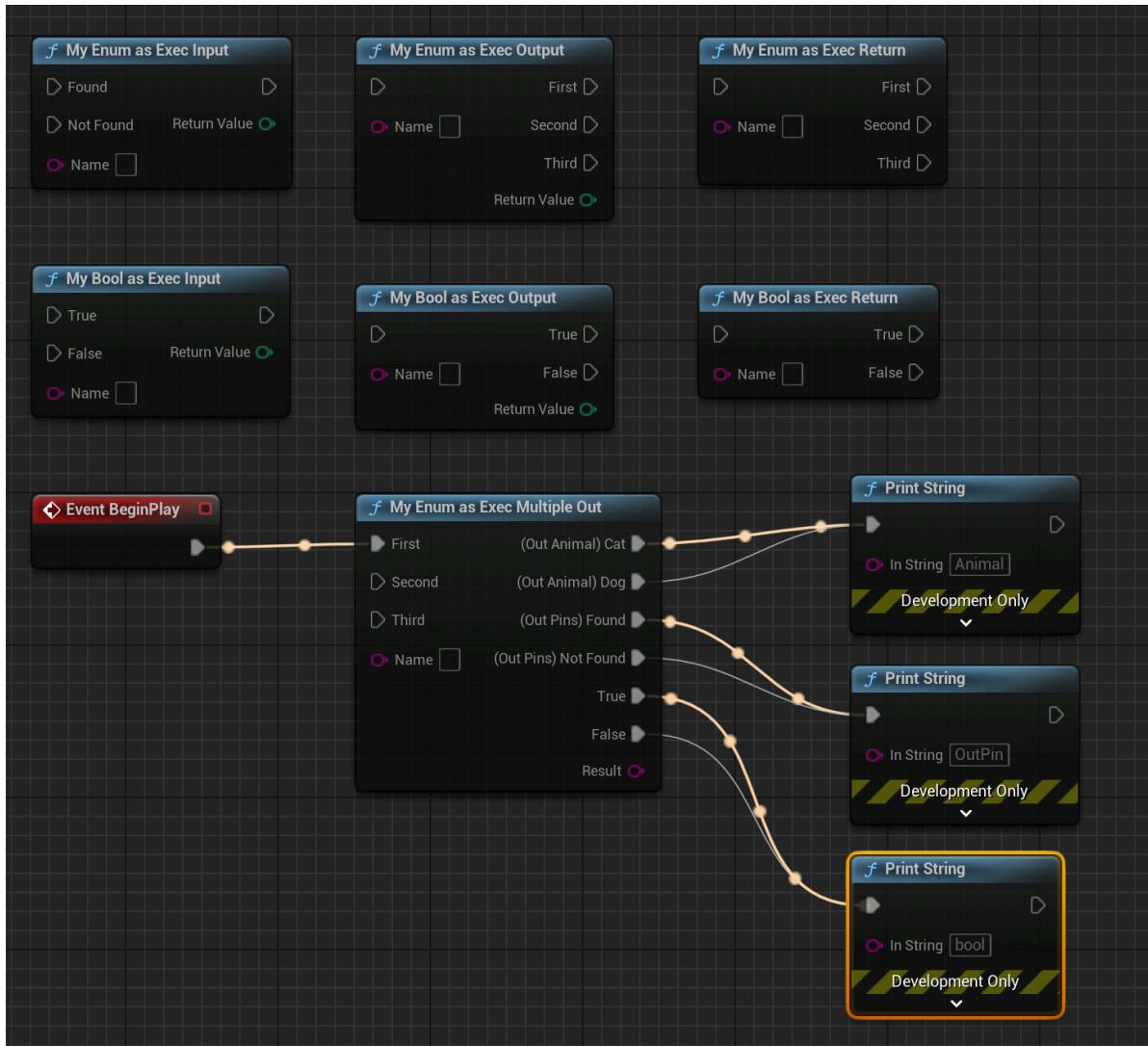
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "ReturnValue"))
    static bool MyBoolAsExec_Return(FString Name) { return false; }

public:
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs =
"InPins|OutAnimal|OutPins|ReturnValue"))
    static bool MyEnumAsExec_MultipleOut(FString Name, EMyExecPins InPins,
EMyExecAnimalPins& OutAnimal, TEnumAsByte<EMyExecPins2::Type>& OutPins, FString& Result);

```

蓝图效果：

可以对照上述上述的函数原型和蓝图节点，可以发现ExpandEnumAsExecs执行3种参数类型。同时也验证了在同时拥有多个输出Enum参数的时候(代码里是OutAnimal | OutPins | ReturnValue)，会按顺序执行3次输出，就像用Sequence节点连接在一起一样。



原理：

真正的创建Pin是在void UK2Node_CallFunction::CreateExecPinsForFunctionCall(const UFunction* Function)，然后这些新的ExecPin和配套的赋值输入参数值，以及根据输出参数执行不同输出ExecPin的逻辑在UK2Node_CallFunction::ExpandNode中。代码太多就不贴出来了。

是如何控制Exec流向的？在函数的实现里，只要把相应的引用输出参数赋值，就自然会流向不同的Pin。这部分逻辑是在UK2Node_CallFunction::ExpandNode中实现。大概逻辑是针对Input引脚，会在中间插入UK2Node_AssignmentStatement，执行不同输入Pin，会相应的设置输入enum参数的值。而针对Output引脚，会在中间插入UK2Node_SwitchEnum，这样当我们在函数中设置引用输出enum参数的值后，就可以根据enum的值，流向不同的输出Pin节点。而对bool参数，也会创建相应的中间蓝图节点来获取和设置bool参数。

函数原始的参数Pin会被隐藏起来，从而只暴露生成后的Exec Pin。

ExposedAsyncProxy

- 功能描述：**在 Async Task 节点中公开此类的一个代理对象。
- 使用位置：** UCLASS
- 引擎模块：** Blueprint
- 元数据类型：** string="abc"
- 限制类型：** Async Blueprint node
- 关联项：** HideSpawnParms, HasDedicatedAsyncNode, HideThen

- 常用程度：★★★

在UK2Node_BaseAsyncTask中使用，用来为蓝图异步节点暴露一个异步对象引脚，以支持对这个异步行为的进一步操作。

在源码里的用处一是UBlueprintAsyncActionBase的子类，二是UGameplayTask子类，皆是分别会有另外的UK2Node_BaseAsyncTask以及UK2Node_LatentGameplayTaskCall来解析类的声明定义并包装生成相应的异步蓝图节点。

基类都是继承自UBlueprintAsyncActionBase。利用ExposedAsyncProxy 指定异步任务对象的名字。在异步蓝图节点上继续返回异步对象，可以在之后支持取消该异步操作。

测试代码：

UCancellableAsyncAction是引擎提供的继承自UBlueprintAsyncActionBase的一个便利的子类。
UMyFunction_Async 定义了一个蓝图异步节点DelayLoop。

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FDelayOutputPin);

UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async :public UBlueprintAsyncActionBase
{
public:
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Loop;

    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Complete;

    UFUNCTION(BlueprintCallable, meta = (BlueprintInternalUseOnly = "true",
    WorldContext = "WorldContextObject"), Category = "Flow Control")
        static UMyFunction_Async* DelayLoop(const UObject* WorldContextObject, const
    float DelayInSeconds, const int Iterations);

    virtual void Activate() override;

    UFUNCTION()
    static void Test();

private:
    const UObject* WorldContextObject = nullptr;
    float MyDelay = 0.f;
    int MyIterations = 0;
    bool Active = false;

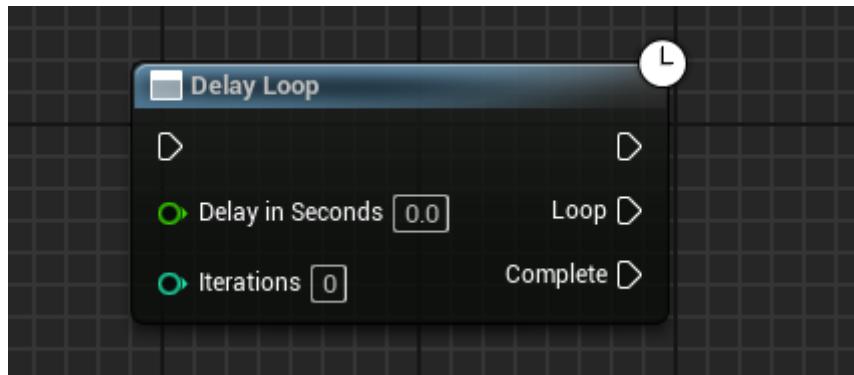
    UFUNCTION()
    void ExecuteLoop();

    UFUNCTION()
    void ExecuteComplete();
};

};
```

默认的蓝图节点是：

如果UMyFunction_Async直接继承自UBlueprintAsyncActionBase，并且没有设置ExposedAsyncProxy，则生成的蓝图异步节点为下图。

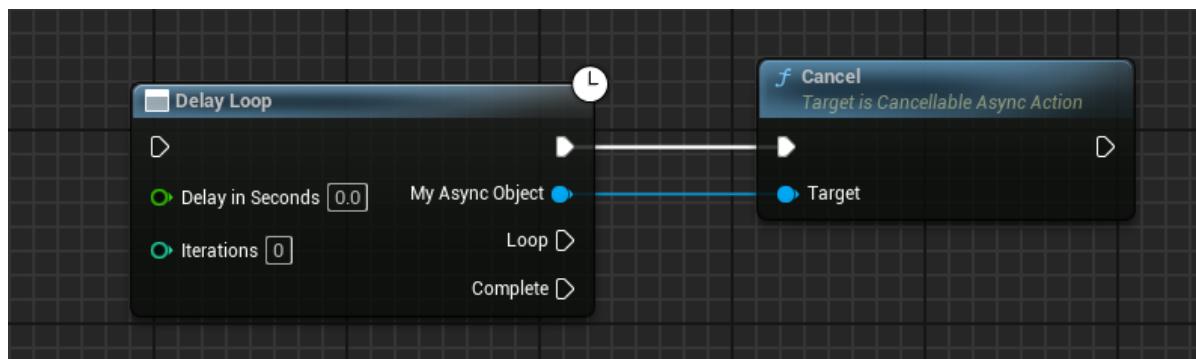


而如果继承自UCancellableAsyncAction(提供了Cancel方法)，并且设置ExposedAsyncProxy为自己想要的AsyncObject引脚名称。

```
UCLASS(Abstract, BlueprintType, meta = (ExposedAsyncProxy = AsyncAction), MinimalAPI)
class UCancellableAsyncAction : public UBlueprintAsyncActionBase
{
    UFUNCTION(BlueprintCallable, Category = "Async Action")
    ENGINE_API virtual void Cancel();
}

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FDelayOutputPin);
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async : public UCancellableAsyncAction
{}
```

修改后的效果如下图：



该Meta在源码中发生的位置：

```
void UK2Node_BaseAsyncTask::AllocateDefaultPins()
{
    bool bExposeProxy = false;
    bool bHideThen = false;
    FText ExposeProxyDisplayName;
    for (const UStruct* TestStruct = ProxyClass; TestStruct; TestStruct =
        TestStruct->GetSuperStruct())
    {
```

```

bExposeProxy |= TestStruct->HasMetaData(TEXT("ExposedAsyncProxy"));
bHideThen |= TestStruct->HasMetaData(TEXT("HideThen"));
if (ExposeProxyDisplayName.IsEmpty())
{
    ExposeProxyDisplayName = TestStruct-
>GetMetaDataText(TEXT("ExposedAsyncProxy"));
}

if (bExposeProxy)
{
    UEdGraphPin* ProxyPin = CreatePin(EGPD_Output,
UEdGraphSchema_K2::PC_Object, ProxyClass,
FBaseAsyncTaskHelper::GetAsyncTaskProxyName());
    if (!ExposeProxyDisplayName.IsEmpty())
    {
        ProxyPin->PinFriendlyName = ExposeProxyDisplayName;
    }
}

}

```

ExposeOnSpawn

- 功能描述:** 使该属性在ContractObject或SpawnActor等创建对象的时候暴露出来。
- 使用位置:** UPROPERTY
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 常用程度:** ★★★★☆

使该属性在ContractObject或SpawnActor等创建对象的时候暴露出来。

- 具体来说，通过在源码搜索，这个标记在UK2Node.AddComponent, UK2Node_ContractObjectFromClass, UK2Node_SpawnActor, UK2Node_LatentGameplayTaskCall的时候用到。
- 在C++里设置的效果等同于在蓝图里勾上ExposeOnSpawn。
- 该meta的设置也会同时设置到PropertyFlags里的CPF_ExposeOnSpawn

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeOnSpawn :public UObject
{
GENERATED_BODY()
public:
// (Category = MyProperty_ExposeOnSpawn, ModuleRelativePath =
Property/Blueprint/MyProperty_ExposeOnSpawn.h)
// PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic

UPROPERTY(EditAnywhere, BlueprintReadWrite)

```

```

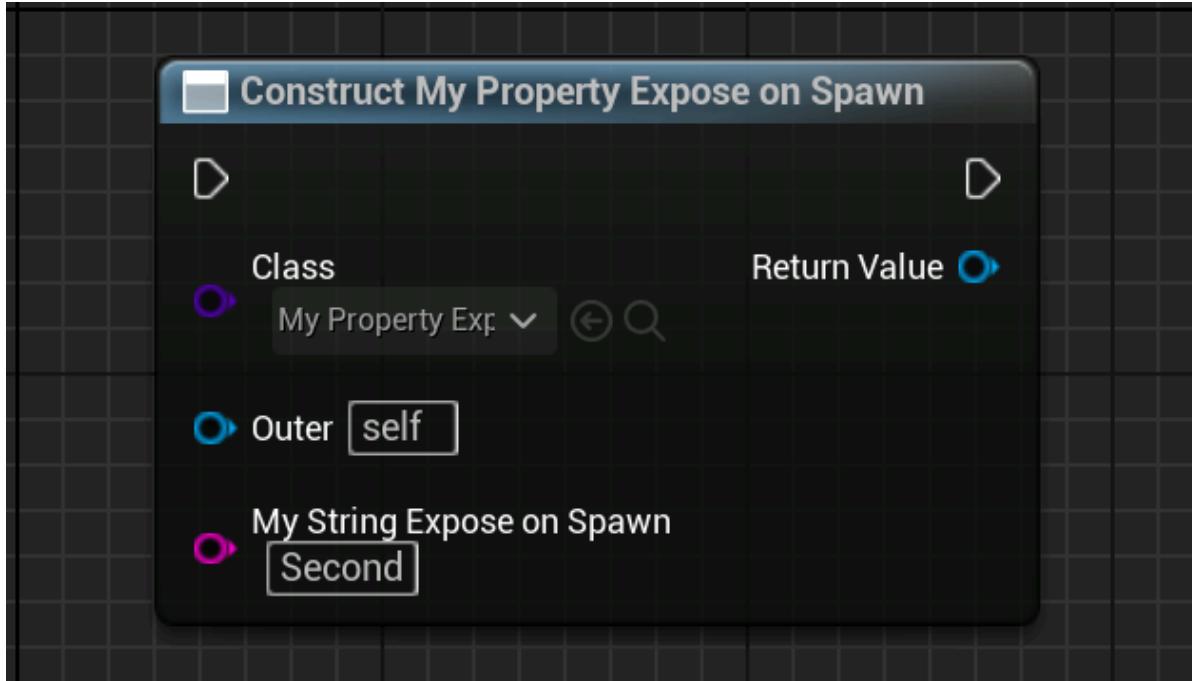
FString MyString = TEXT("First");

// (Category = MyProperty_ExposeOnSpawn, ExposeOnSpawn = ,
ModuleRelativePath = Property/Blueprint/MyProperty_ExposeOnSpawn.h)
// PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor | 
CPF_ExposeOnSpawn | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ExposeOnSpawn))
FString MyString_ExposeOnSpawn = TEXT("Second");
};

```

测试效果：

可见MyString_ExposeOnSpawn 暴露了出来，而MyString 没有。



原理：

在UHT的时候会分析如果包含ExposeOnSpawn就会同步设置CPF_ExposeOnSpawn。

而在IsPropertyExposedOnSpawn这个函数里具体判断是否要暴露，这个函数被上述的4个函数节点引用。源码里举UK2Node_ConstructObjectFromClass里的CreatePinsForClass作为例子，可见只有bIsExposedToSpawn 的时候才会为蓝图节点开始创建额外的Pin引脚。

```

if (propertySettings.MetaData.ContainsKey(UhtNames.ExposeOnSpawn))
{
    propertySettings.PropertyFlags |= EPropertyFlags.ExposeOnSpawn;
}

bool UEdGraphSchema_K2::IsPropertyExposedOnSpawn(const FPropertyParams*PropertyParams)
{
   PropertyParams = FBlueprintEditorUtils::GetMostUpToDatePropertyParams(PropertyParams);
    if (PropertyParams)
    {
        const bool bMeta =PropertyParams->HasMetaData(FBlueprintMetadata::MD_ExposeOnSpawn);
        const bool bFlag =PropertyParams->HasAllPropertyParams(CPF_ExposeOnSpawn);
        if (bMeta != bFlag)
    }
}

```

```

    {
        const FCoreTexts& CoreTexts = FCoreTexts::Get();

        UE_LOG(LogBlueprint, Warning
            , TEXT("ExposeOnSpawn ambiguity. Property '%s', MetaData '%s',
Flag '%s'"))
            , *Property->GetFullName()
            , bMeta ? *CoreTexts.True.ToString() :
*CoreTexts.False.ToString()
            , bFlag ? *CoreTexts.True.ToString() :
*CoreTexts.False.ToString());
    }

    return bMeta || bFlag;
}

return false;
}

void UK2Node_ConstructObjectFromClass::CreatePinsForClass(UClass* InClass,
TArray<UEdGraphPin*>* OutClassPins)
{
    for (TFieldIterator<FProperty> PropertyIt(InClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
    {
        FProperty* Property = *PropertyIt;
        UClass* PropertyClass = CastChecked<UClass>(Property->GetOwner<UObject>());
        const bool bIsDelegate = Property-
>IsA(FMulticastDelegateProperty::StaticClass());
        const bool bIsExposedToSpawn =
UEdGraphSchema_K2::IsPropertyExposedOnSpawn(Property);
        const bool bIsSettableExternally = !Property-
>HasAnyPropertyFlags(CPF_DisableEditOnInstance);

        if( bIsExposedToSpawn &&
            !Property->HasAnyPropertyFlags(CPF_Parm) &&
            bIsSettableExternally &&
            Property->HasAllPropertyFlags(CPF_BlueprintVisible) &&
            !bIsDelegate &&
            (nullptr == FindPin(Property->GetFName()) ) &&
            FBlueprintEditorUtils::PropertyStillExists(Property))
        {
            if (UEdGraphPin* Pin = CreatePin(EGPD_Input, NAME_None, Property-
>GetFName()))
            {
        }
    }
}

```

ForceAsFunction

- 功能描述:** 把C++里用BlueprintImplementableEvent或NativeEvent定义的事件强制改为函数在子类中覆写。
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 常用程度:** ★★★

把C++里用BlueprintImplementableEvent或NativeEvent定义的事件强制改为函数在子类中覆写。

什么情况下需要把Event改成函数?

- 变成函数后，在实现的时候就可以定义内部的局部变量。当然也就失去了调用Delay等延时函数的能力。
- 事件不能有输出的参数，但是如果想要一个有输出的函数在蓝图类里覆写（得 BlueprintImplementableEvent或NativeEvent），则默认的以事件方式重载是不行的。因此这个时候把这个事件强迫改为函数的形式，就可以正常的覆写。
- 带有输出或返回参数的Event会默认被改为function，即使没有加上ForceAsFunction。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_ForceAsFunction :public AActor
{
public:
    GENERATED_BODY()
public:
//FUNC_Native | FUNC_Event | FUNC_Public | FUNC_BlueprintCallable |
FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    void MyNativeEvent_Default(const FString& name);

//FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    void MyImplementableEvent_Default(const FString& name);

public:
    //((ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
    //FUNC_Native | FUNC_Event | FUNC_Public | FUNC_BlueprintCallable |
FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintNativeEvent, meta = (ForceAsFunction))
    void MyNativeEvent_ForceAsFunction(const FString& name);

    ////(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
    //FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, meta =
(ForceAsFunction))
    void MyImplementableEvent_ForceAsFunction(const FString& name);

public:
    //FUNC_Native | FUNC_Event | FUNC_Public | FUNC_HasOutParams |
FUNC_BlueprintCallable | FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    bool MyNativeEvent_Output(const FString& name, int32& outValue);

//FUNC_Event | FUNC_Public | FUNC_HasOutParams | FUNC_BlueprintCallable |
FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    bool MyImplementableEvent_Output(const FString& name, int32& outValue);
```

```

//(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
//FUNC_Native | FUNC_Event | FUNC_Public | FUNC_HasOutParms |
FUNC_BlueprintCallable | FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, meta = (ForceAsFunction))
bool MyNativeEvent_Output_ForceAsFunction(const FString& name, int32&
outValue);

//(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
//FUNC_Event | FUNC_Public | FUNC_HasOutParms | FUNC_BlueprintCallable |
FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, meta =
(ForceAsFunction))
bool MyImplementableEvent_Output_ForceAsFunction(const FString& name, int32&
outValue);
};

```

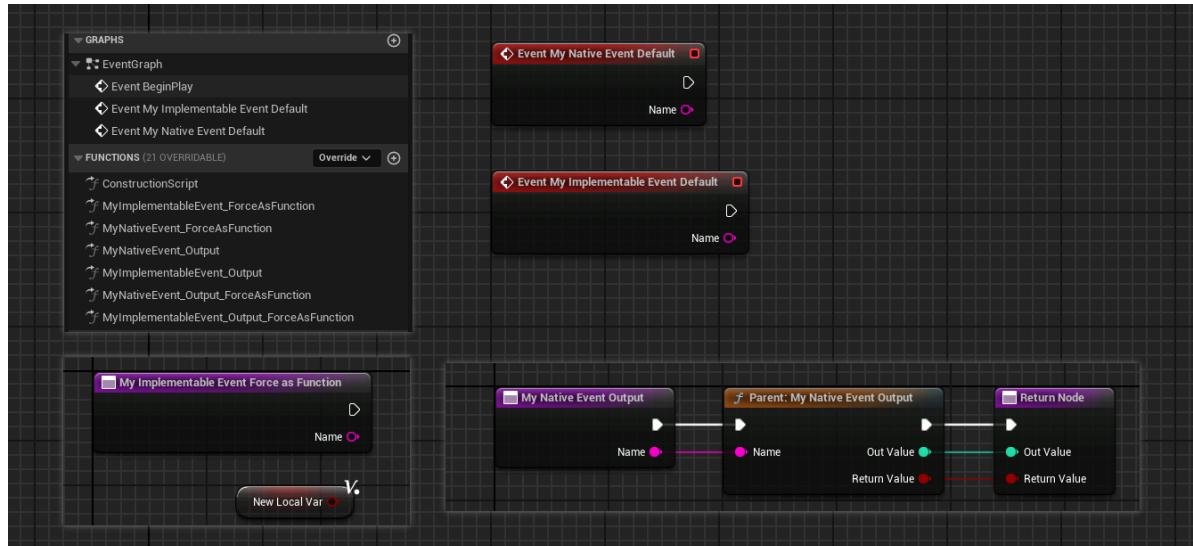
蓝图中效果：

在函数上覆写的时候，会发现只有MyNativeEvent_Default和MyImplementableEvent_Default被默认覆盖为事件，其他都以函数的方式被覆写。

图里展示了MyImplementableEvent_ForceAsFunction被改为函数后，可以在内部定义局部变量。

也展示了MyNativeEvent_Output这种拥有输出参数的事件被覆写成函数后的函数体。

但无论是覆盖为事件还是函数，被调用的时候用法并无区别。



原理：

判断一个函数是否是事件的逻辑为以下函数：

主要看第二if和最后判断，BlueprintImplementableEvent或NativeEvent的函数上都会加上FUNC_BlueprintEvent标签，因此如果带有ForceAsFunction元数据或者有输出参数（返回值也算），就只能显示为函数。

```

bool UEdGraphSchema_K2::FunctionCanBePlacedAsEvent(const UFunction* InFunction)
{

```

```

    // First check we are override-able, non-static, non-const and not marked
    thread safe
    if (!InFunction || !CanKismetOverrideFunction(InFunction) || InFunction-
>HasAnyFunctionFlags(FUNC_Static|FUNC_Const) ||
FBPBlueprintEditorUtils::HasFunctionBlueprintThreadSafeMetaData(InFunction))
{
    return false;
}

    // Check if meta data has been set to force this to appear as blueprint
    function even if it doesn't return a value.
    if (InFunction->HasAllFunctionFlags(FUNC_BlueprintEvent) && InFunction-
>HasMetaData(FBlueprintMetadata::MD_ForceAsFunction))
{
    return false;
}

    // Then look to see if we have any output, return, or reference params
    return !HasFunctionAnyOutputParameter(InFunction);
}

```

GetByRef

- 功能描述:** 指定UHT为该属性生成返回引用的C++代码
- 使用位置:** UPROPERTY
- 引擎模块:** UHT
- 元数据类型:** bool
- 限制类型:** 只用在SparseClassDataTypes 指定的结构里的属性。
- 关联项:** SparseClassDataTypes

指定UHT为该属性生成返回引用的C++代码。

只用在SparseClassDataTypes 指定的结构里的属性。

代码例子:

```

USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    FString MyString_EditDefault = TEXT("MyName");
    //FString GetMyString_EditDefault() const { return
    GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault; } \

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");

```

```

    //const FString& GetMyString_EditDefault_ReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault_ReadOnly; }
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes = MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
    GENERATED_BODY()
}

```

生成的代码：

可见，后者生成返回值是const FString&而不是FString。

```

#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_36_SPARSE_DAT
A_PROPERTY_ACCESSORS \
FString GetMyString_EditDefault() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault; } \
const FString& GetMyString_EditDefault_ReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault_ReadOnly; }

```

原理：

UHT中为SparseDataType生成代码的时候会判断GetByRef来分别生成不同的格式代码。

```

private StringBuilder AppendSparseDeclarations(StringBuilder builder, uhtClass
classObj, IEnumerable<UhtScriptStruct> sparseScriptStructs,
uhtUsedDefineScopes<UhtProperty> sparseProperties)
{
    if (property.MetaData.ContainsKey(UhtNames.GetByRef()))
    {
        builder.Append("const ").AppendSparse(property).Append("&
Get").Append(cleanPropertyName).Append("( ) const");
    }
    else
    {
        builder.AppendSparse(property).Append(""
Get").Append(cleanPropertyName).Append("( ) const");
    }
}

```

HasDedicatedAsyncNode

- **使用位置:** UCLASS
- **元数据类型:** bool
- **关联项:** ExposedAsyncProxy

隐藏UBlueprintAsyncActionBase子类里工厂方法自动生成的蓝图异步节点，以便自己可以手动自定义创建一个相应的UK2Node_XXX。

```
/***
 * BlueprintCallable factory functions for classes which inherit from
 UBlueprintAsyncActionBase will have a special blueprint node created for it:
 UK2Node_AsyncAction
 * You can stop this node spawning and create a more specific one by adding the
 UCLASS metadata "HasDedicatedAsyncNode"
 */

UCLASS(MinimalAPI)
class UBlueprintAsyncActionBase : public UObject
{}
```

测试代码：

```
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy =
MyAsyncObject, HasDedicatedAsyncNode))
class INSIDER_API UMyFunction_Async : public UCancellableAsyncAction
{};

//可以自定义一个K2Node
UCLASS()
class INSIDER_API UK2Node_MyFunctionAsyncAction : public UK2Node_AsyncAction
{
    GENERATED_BODY()

    // UK2Node interface
    virtual void GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const override;
    virtual void AllocateDefaultPins() override;
    // End of UK2Node interface

protected:
    virtual bool HandleDelegates(
        const TArray<FBaseAsyncTaskHelper::FOutputPinAndLocalVariable>&
variableOutputs, UEdGraphPin* ProxyObjectPin,
        UEdGraphPin*& InoutLastThenPin, UEdGraph* SourceGraph,
        FKismetCompilerContext& CompilerContext) override;
};

void
UK2Node_MyFunctionAsyncAction::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    struct GetMenuActions_Utils
    {
        static void SetNodeFunc(UEdGraphNode* NewNode, bool /*bIsTemplateNode*/,
        TweakObjectPtr<UFunction> FunctionPtr)
        {
            UK2Node_MyFunctionAsyncAction* AsyncTaskNode =
CastChecked<UK2Node_MyFunctionAsyncAction>(NewNode);
            if (FunctionPtr.IsValid())

```

```

    {
        UFunction* Func = FunctionPtr.Get();
        FObjectProperty* ReturnProp = CastFieldChecked<FObjectProperty>
        (Func->GetReturnProperty());

        AsyncTaskNode->ProxyFactoryFunctionName = Func->GetFName();
        AsyncTaskNode->ProxyFactoryClass = Func->GetOuterUClass();
        AsyncTaskNode->ProxyClass = ReturnProp->PropertyClass;
        AsyncTaskNode->NodeComment = TEXT("This is MyCustomK2Node");
    }
}

};

UClass* NodeClass = GetClass();
ActionRegistrar.RegisterClassFactoryActions<UMyFunction_Async>
(FBlueprintActionDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeC
lass](const UFunction* FactoryFunc)->UBlueprintNodeSpawner*
{
    UBlueprintNodeSpawner* NodeSpawner =
    UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
    check(NodeSpawner != nullptr);
    NodeSpawner->NodeClass = NodeClass;

    TWeakObjectPtr<UFunction> FunctionPtr =
    MakeWeakObjectPtr(const_cast<UFunction*>(FactoryFunc));
    NodeSpawner->CustomizeNodeDelegate =
    UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(GetMenuActions_Utils
::SetNodeFunc, FunctionPtr);

    return NodeSpawner;
}));
```

}

```

void UK2Node_MyFunctionAsyncAction::AllocateDefaultPins()
{
    Super::AllocateDefaultPins();
}
```

```

bool UK2Node_MyFunctionAsyncAction::HandleDelegates(const
TArray<FBaseAsyncTaskHelper::FOutputPinAndLocalVariable>& VariableOutputs,
UEdGraphPin* ProxyObjectPin, UEdGraphPin*& InOutLastThenPin, UEdGraph*
SourceGraph, FKismetCompilerContext& CompilerContext)
{
    bool bIsErrorFree = true;

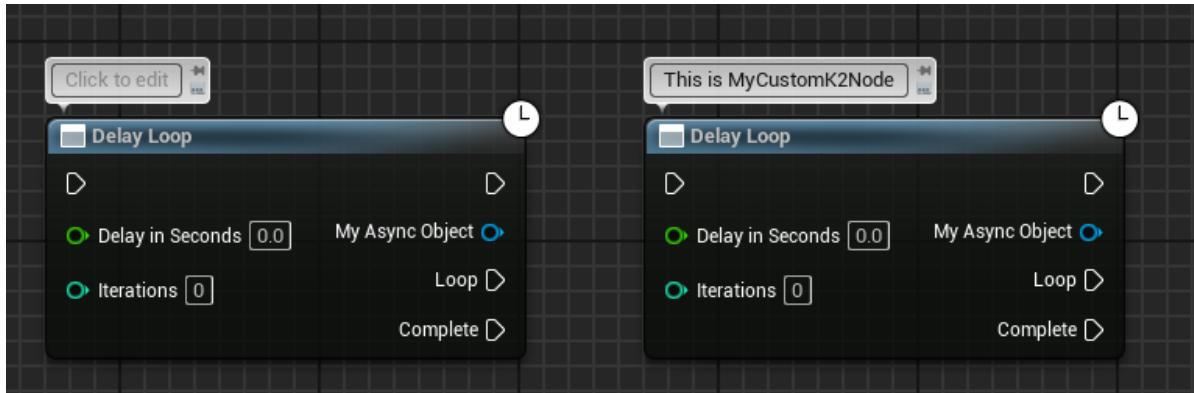
    for (TFieldIterator<FMulticastDelegateProperty> PropertyIt(ProxyClass);
PropertyIt && bIsErrorFree; ++PropertyIt)
    {
        UEdGraphPin* LastActivatedThenPin = nullptr;
        bIsErrorFree &=
FBaseAsyncTaskHelper::HandleDelegateImplementation(*PropertyIt, VariableOutputs,
ProxyObjectPin, InOutLastThenPin, LastActivatedThenPin, this, SourceGraph,
CompilerContext);
    }

    return bIsErrorFree;
}
```

```
}
```

蓝图效果：

左侧是引擎自带的UK2Node_AsyncAction生成节点，右边是自定义的UK2Node_MyFunctionAsyncAction生成的蓝图节点，虽然功能一致，但是右边额外加了个注释以便区分。有了这个基础，你也可以在其中继续重载方法进一步自定义。



当前在源码里有两处地方使用：

```
UCLASS(BlueprintType, meta = (ExposedAsyncProxy = "AsyncTask",
HasDedicatedAsyncNode))
class GAMEPLAYMESSAGES_API UAsyncAction_RegisterGameplayMessageReceiver : public
UBlueprintAsyncActionBase
{
    UFUNCTION(BlueprintCallable, Category = Messaging, meta=
(WorldContext="WorldContextObject", BlueprintInternalUseOnly="true"))
    static UAsyncAction_RegisterGameplayMessageReceiver*
RegisterGameplayMessageReceiver(UObject* WorldContextObject, FEventMessageTag
Channel, UScriptStruct* PayloadType, EGameplayMessageMatchType MatchType =
EGameplayMessageMatchType::ExactMatch, AActor* ActorContext = nullptr);

}

//由UK2Node_GameplayMessageAsyncAction来负责创建
void
UK2Node_GameplayMessageAsyncAction::GetMenuActions(FBlueprintDatabaseRegistrar& ActionRegistrar) const
{
    //...
    UClass* NodeClass = GetClass();

    ActionRegistrar.RegisterClassFactoryActions<UAsyncAction_RegisterGameplayMessageR
eceiver>
    (FBlueprintDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeC
lass](const UFunction* FactoryFunc) -> UBlueprintNodeSpawner*
    {
        UBlueprintNodeSpawner* NodeSpawner =
        UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
        check(NodeSpawner != nullptr);
        NodeSpawner->NodeClass = NodeClass;
    })
}
```

```

        TweakObjectPtr<UFunction> FunctionPtr =
    MakeweakObjectPtr(const_cast<UFunction*>(FactoryFunc));
        NodeSpawner->CustomizeNodeDelegate =
    UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(OptionsMenu_Utils:
    :SetNodeFunc, FunctionPtr);

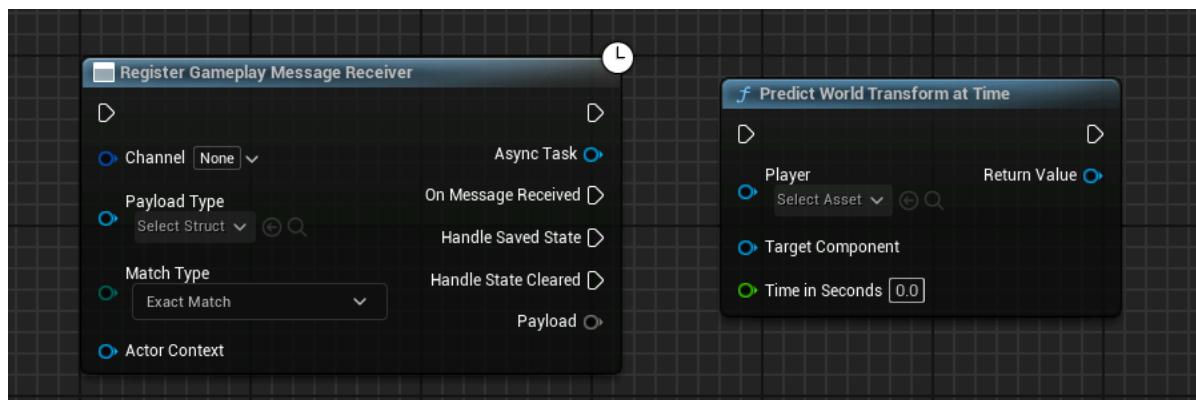
        return NodeSpawner;
    }) );
}

UCLASS(BlueprintType, meta=(ExposedAsyncProxy = "AsyncTask",
HasDedicatedAsyncNode))
class UMovieSceneAsyncAction_SequencePrediction : public
UBlueprintAsyncActionBase
{
    UFUNCTION(BlueprintCallable, Category=Cinematics)
    static UMovieSceneAsyncAction_SequencePrediction*
PredictWorldTransformAtTime(UMovieSceneSequencePlayer* Player, USceneComponent* TargetComponent, float TimeInSeconds);
}

```

生成的蓝图：

UAsyncAction_RegisterGameplayMessageReceiver由自定义的UK2Node_GameplayMessageAsyncAction来创建蓝图节点，从而提供了一个泛型的Payload输出引脚。而UMovieSceneAsyncAction_SequencePrediction里的工厂方法PredictWorldTransformAtTime，由于隐藏了自动生成的版本，又没有加上BlueprintInternalUseOnly来抑制UHT生成的版本，因此最终呈现的是普通版本的静态函数蓝图节点。



源码里的作用机制：

可以看到，如果在类上有找到HasDedicatedAsyncNode，直接就返回nullptr，不再生成NodeSpawner，因此就阻止了蓝图节点的生成。

```

void UK2Node_AsyncAction::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    ActionRegistrar.RegisterClassFactoryActions<UBlueprintAsyncActionBase>
(FBlueprintActionDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeC
lass](const UFunction* FactoryFunc)->UBlueprintNodeSpawner*
{
    UClass* FactoryClass = FactoryFunc ? FactoryFunc->GetOwnerClass() :
    nullptr;

```

```

    if (FactoryClass && FactoryClass-
>HasMetaData(TEXT("HasDedicatedAsyncNode")))
{
    // Wants to use a more specific blueprint node to handle the async
action
    return nullptr;
}

UBlueprintNodeSpawner* NodeSpawner =
UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
check(NodeSpawner != nullptr);
NodeSpawner->NodeClass = NodeClass;

TweakObjectPtr<UFunction> FunctionPtr =
MakeWeakObjectPtr(const_cast<UFunction*>(FactoryFunc));
NodeSpawner->CustomizeNodeDelegate =
UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(GetMenuActions_Utils:
:SetNodeFunc, FunctionPtr);

return NodeSpawner;
}) );
}

```

HiddenNode

- 功能描述:** 把指定的UBTNode隐藏不在右键菜单中显示。
- 使用位置:** UCLASS
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 限制类型:** UBTNode
- 常用程度:** ★

把指定的UBTNode隐藏不在右键菜单中显示。

测试代码:

```

UCLASS(MinimalAPI,meta = ())
class UMyBT_NotHiddenNode : public UBTDecorator
{
GENERATED_UCLASS_BODY()

UPROPERTY(Category = Node, EditAnywhere)
float MyFloat;
};

UCLASS(MinimalAPI,meta = (HiddenNode))
class UMyBT_HiddenNode : public UBTDecorator
{
GENERATED_UCLASS_BODY()

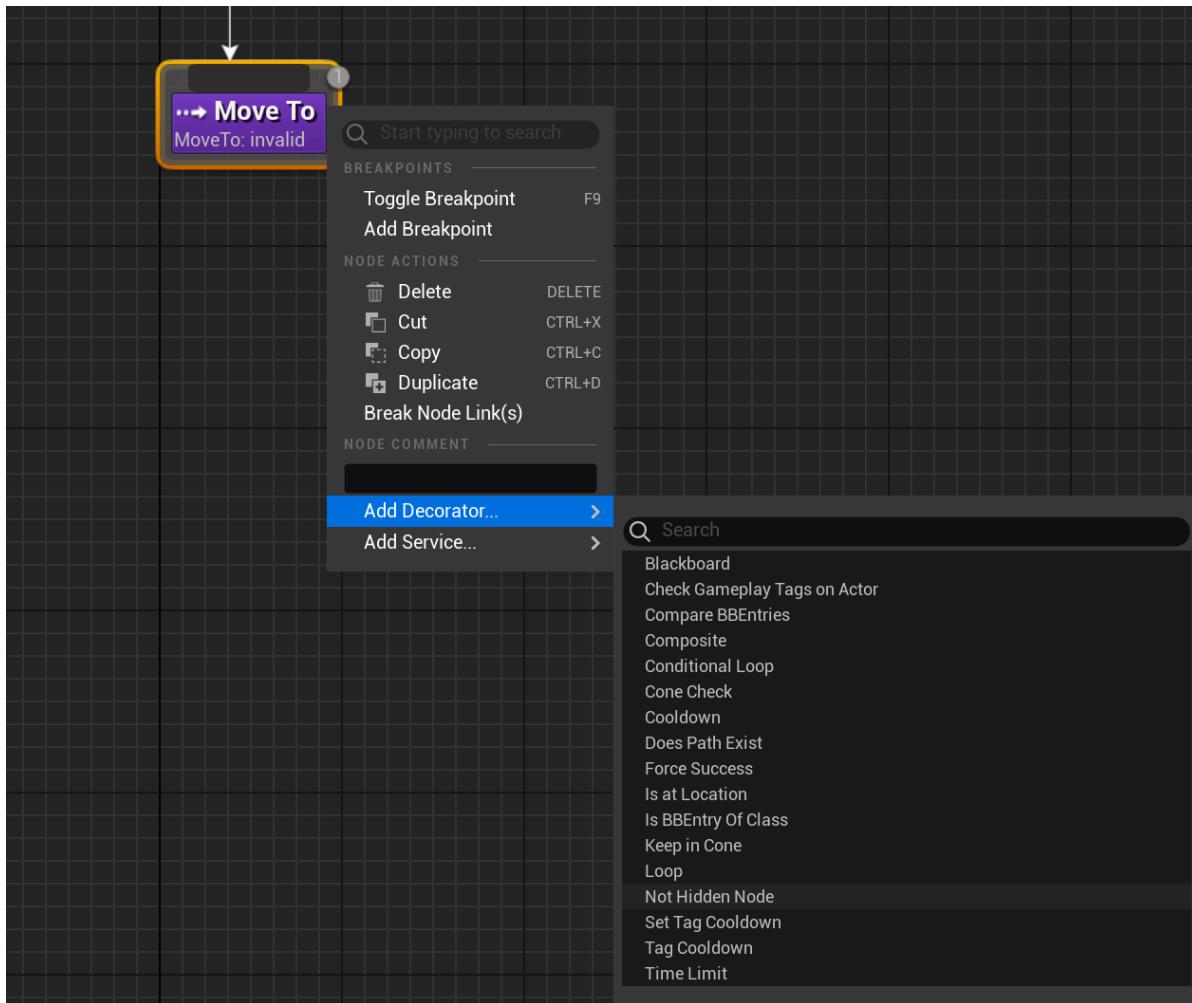
UPROPERTY(Category = Node, EditAnywhere)

```

```
float MyFloat;  
};
```

测试结果：

可见只有UMyBT_NotHiddenNode 显示了出来，而UMyBT_HiddenNode 被隐藏了。



原理：

原理比较简单，就是坚持元数据标记，然后设置blsHidden。

```
bool FGraphNodeClassHelper::IsHidingClass(UClass* Class)  
{  
    static FName MetaHideInEditor = TEXT("HiddenNode");  
  
    return  
        Class &&  
        ((Class->HasAnyClassFlags(CLASS_Native) && Class-  
        >HasMetaData(MetaHideInEditor))  
        || ForcedHiddenClasses.Contains(Class));  
}  
  
//D:\github\UnrealEngine\Engine\Source\Editor\AIGraph\Private\AIGraphTypes.cpp  
void FGraphNodeClassHelper::BuildClassGraph()  
{  
    for (TObjectIterator<UClass> It; It; ++It)
```

```

{
    UClass* TestClass = *It;
    if (TestClass->HasAnyClassFlags(CLASS_Native) && TestClass-
>IsChildOf(RootNodeClass))
    {

        NewData.bIsHidden = IsHidingClass(TestClass);

        NewNode->Data = NewData;

        if (TestClass == RootNodeClass)
        {
            RootNode = NewNode;
        }

        NodeList.Add(NewNode);
    }
}
}

```

HideFunctions

- 功能描述:** 在属性查看器中不显示指定类别中的所有函数。
- 使用位置:** UClass
- 引擎模块:** Blueprint
- 元数据类型:** strings="a, b, c"
- 关联项:**
UCLASS: HideFunctions, ShowFunctions
- 常用程度:** ★★★

HideThen

- 功能描述:** 隐藏异步蓝图节点的Then引脚
- 使用位置:** UClass
- 元数据类型:** bool
- 限制类型:** 蓝图异步节点
- 关联项:** ExposedAsyncProxy

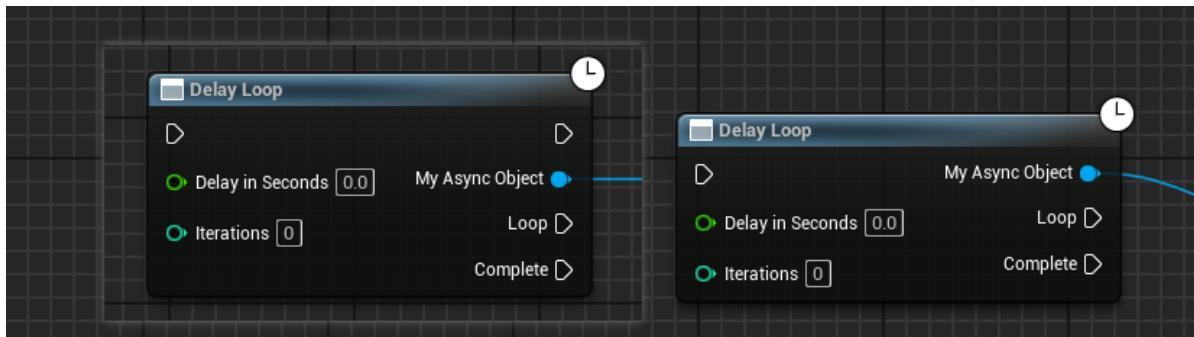
在源码中HideThen只在UK2Node_BaseAsyncTask中判断，因此这个标签只作用于蓝图异步节点。

测试代码：

```
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{};

UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy =
MyAsyncObject, HideThen))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{}
```

使用HideThen前后对比：



源码位置：

```
void UK2Node_BaseAsyncTask::AllocateDefaultPins()
{
    bool bExposeProxy = false;
    bool bHideThen = false;
    FText ExposeProxyDisplayName;
    for (const UStruct* TestStruct = ProxyClass; TestStruct; TestStruct =
TestStruct->GetSuperStruct())
    {
        bExposeProxy |= TestStruct->HasMetaData(TEXT("ExposedAsyncProxy"));
        bHideThen |= TestStruct->HasMetaData(TEXT("HideThen"));
        if (ExposeProxyDisplayName.IsEmpty())
        {
            ExposeProxyDisplayName = TestStruct-
>GetMetaDataText(TEXT("ExposedAsyncProxy"));
        }
    }

    if (!bHideThen)
    {
        CreatePin(EGPD_Output, UEdGraphSchema_K2::PC_Exec,
UEdGraphSchema_K2::PN_Then);
    }
}
```

IgnoreTypePromotion

- 功能描述：标记该函数不收录进类型提升函数库

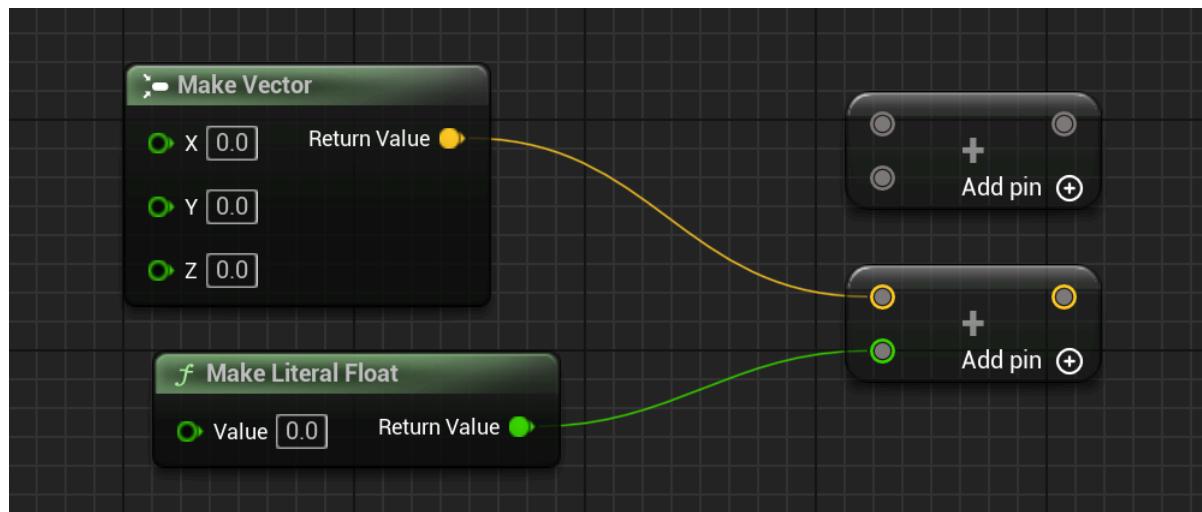
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **限制类型:** UBlueprintFunctionLibrary内的BlueprintPure，以OP_XXX形式的函数
- **常用程度:** ★★

标记该函数不收录进类型提升函数库。

这里有三个关键点：

一是该函数是什么类型？或者说一个类型提升函数是什么类型？根据IsPromotableFunction源码定义，该函数必须定义在UBlueprintFunctionLibrary中，必须是BlueprintPure，且是以操作符“OP_XXX”这种名字格式的函数，其中OP的名字在OperatorNames这个命名空间中可见。示例可见KismetMathLibrary中有大量的这种类型函数。

二是什么是类型提升函数库？源码中有FTypePromotion的类，里面的OperatorTable记录了从OP名字到函数列表的一个Map映射，比如支持Add(+)的有多个Add_Vector, Add_Float等。当我们在蓝图中右键输入+或Add节点的时候，出现的第一个是一个泛型的+节点。然后再连接到具体的变量类型，蓝图系统根据Pin类型会在FTypePromotion::OperatorTable里找到最匹配的Func来最终调用，或者自动的在内部做类型提升。比如下图的+最终调用的就是UKismetMathLibrary::Add_VectorFloat。这种泛型的运算符调用，使得各种基本类型之间的基本运算在蓝图节点创建上更加的便利和统一，也方便直接Add Pin和在Pin上直接Convert到可兼容的其他Pin类型。



三是为什么有些函数不想被收录进FTypePromotion里？在源码中搜索，在KismetMathLibrary中发现只有FDateTime加上了IgnoreTypePromotion标记。虽然FDateTime也定义了一系列的各种运算符函数，比如Add, Subtract和其他各种比较运算符，但是FDateTime在意义上和其他的基本类型可互相运算不同，FDateTime+float或FDateTime+vector并无什么意义。FDateTime只允许+FDateTime或+FTimeSpan。因此类似FDateTime这种并不想参与到其他类型的类型提升转换关系中，只想安静的自成一派在自己小范围内运算，就可以加上IgnoreTypePromotion，不参与进FTypePromotion这个体系。

测试代码：

假设我们有个FGameProp结构，定义了游戏里的战斗属性 (HP, Attack, Defense) 这些，然后游戏中通常要穿装备和加Buff等等操作会计算个最终的属性。这种FGameProp结构我们就可以为之定义一系列的基本运算函数。并加上IgnoreTypePromotion，因为肯定不想参与进TypePromotion，与别的基本类型直接运算 (float, vector等这些)。

为了对比，代码里也定义一模一样FGameProp2和一样的运算函数，唯一区别是不加IgnoreTypePromotion，然后观察最终的蓝图节点上的差异。

```

USTRUCT(BlueprintType)
struct FGameProp
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double HP;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double Attack;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double Defense;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_IgnoreTypePromotion :public
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    /** Makes a GameProp struct */
    UFUNCTION(BlueprintPure, Category = "Math|GameProp", meta =
(IgnoreTypePromotion, NativeMakeFunc))
    static FGameProp MakeGameProp(double HP, double Attack, double Defense) {
        return FGameProp();
    }

    /** Breaks a GameProp into its components */
    UFUNCTION(BlueprintPure, Category = "Math|GameProp", meta =
(IgnoreTypePromotion, NativeBreakFunc))
    static void BreakGameProp(FGameProp InGameProp, double& HP, double& Attack,
double& Defense) {}

    /** Addition (A + B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
+ GameProp", CompactNodeTitle = "+", Keywords = "+ add plus"), Category =
"Math|GameProp")
    static FGameProp Add_GameProp(FGameProp A, FGameProp B);

    /** Subtraction (A - B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
- GameProp", CompactNodeTitle = "-", Keywords = "- subtract minus"), Category =
"Math|GameProp")
    static FGameProp Subtract_GameProp(FGameProp A, FGameProp B) { return
        FGameProp(); }

    /** Returns true if the values are equal (A == B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "Equal
(GameProp)", CompactNodeTitle = "==", Keywords = "== equal"), Category =
"Math|GameProp")
    static bool EqualEqual_GameProp(FGameProp A, FGameProp B) { return true; }

    /** Returns true if the values are not equal (A != B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "Not
Equal (GameProp)", CompactNodeTitle = "!=", Keywords = "!= not equal"), Category
= "Math|GameProp")

```

```

static bool NotEqual_GameProp(FGameProp A, FGameProp B) { return true; }

/** Returns true if A is greater than B (A > B) */
UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
> GameProp", CompactNodeTitle = ">", Keywords = "> greater"), Category =
"Math|GameProp")
static bool Greater_GameProp(FGameProp A, FGameProp B) { return true; }

/** Returns true if A is greater than or equal to B (A >= B) */
UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
>= GameProp", CompactNodeTitle = ">=", Keywords = ">= greater"), Category =
"Math|GameProp")
static bool GreaterEqual_GameProp(FGameProp A, FGameProp B) { return true; }

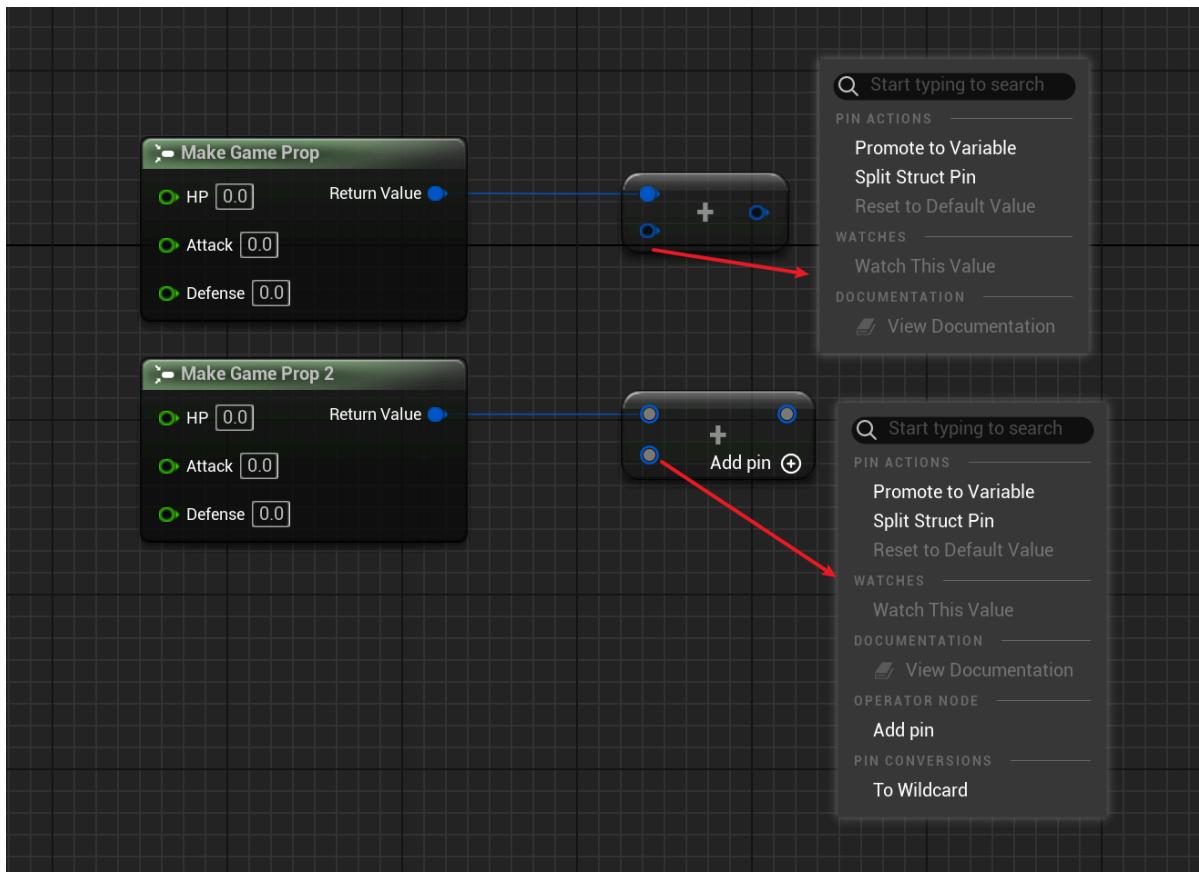
/** Returns true if A is less than B (A < B) */
UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
< GameProp", CompactNodeTitle = "<", Keywords = "< less"), Category =
"Math|GameProp")
static bool Less_GameProp(FGameProp A, FGameProp B) { return true; }

/** Returns true if A is less than or equal to B (A <= B) */
UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
<= GameProp", CompactNodeTitle = "<=", Keywords = "<= less"), Category =
"Math|GameProp")
static bool LessEqual_GameProp(FGameProp A, FGameProp B) { return true; }
};

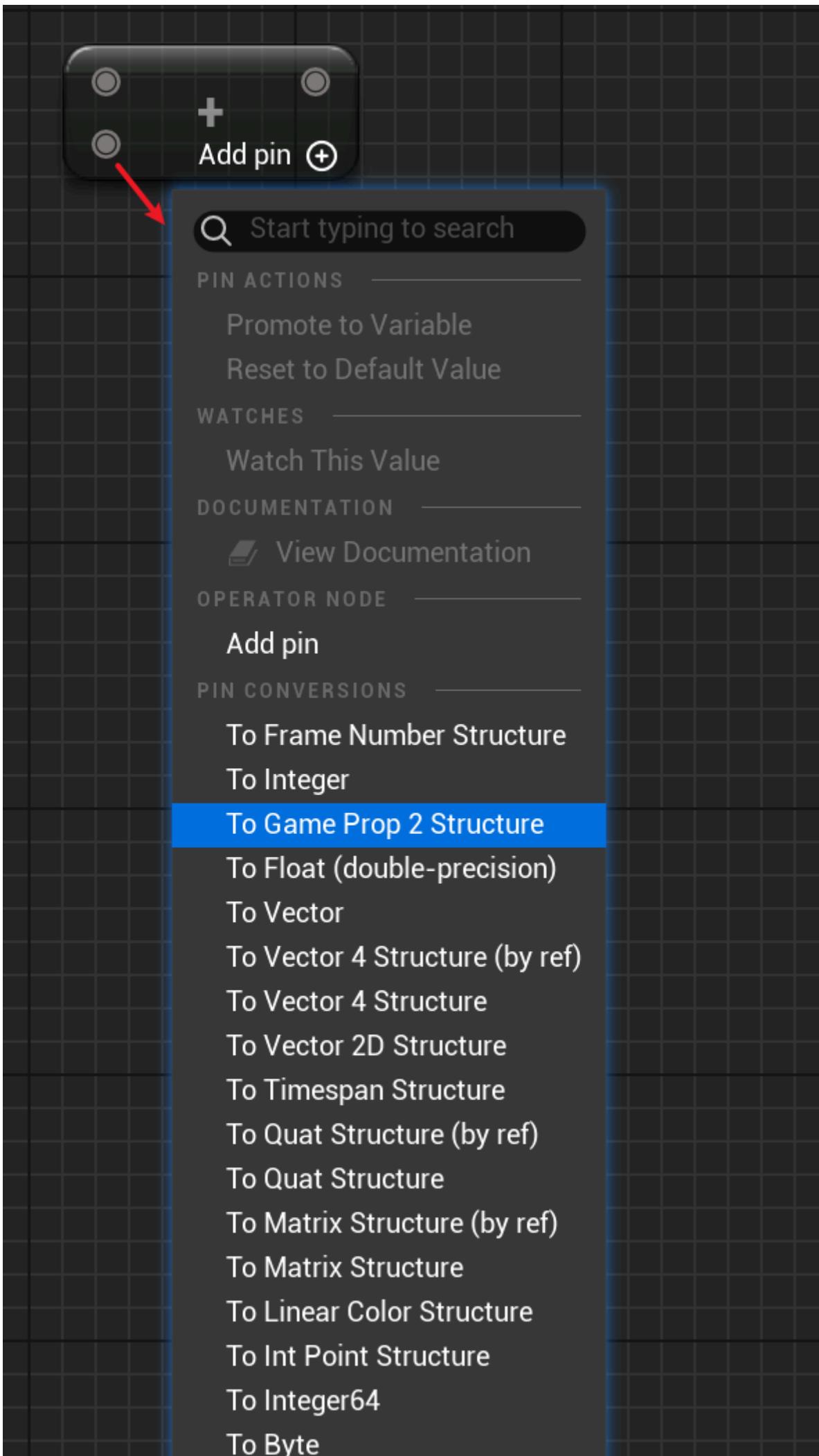
```

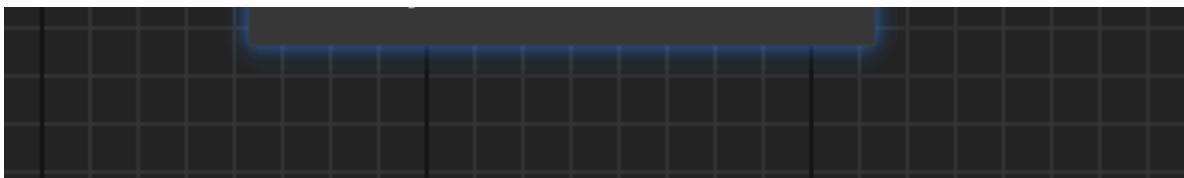
蓝图效果：

加了IgnoreTypePromotion的FGameProp, Add的时候就是直接最原始的Add_GameProp节点。而不加IgnoreTypePromotion的FGameProp2, Add的时候产生的节点是泛型的+, 可以继续AddPin, 甚至在Pin上右键还会尝试寻找向其他类型的转换（虽然这里结果找不到，是因为我们没有定义FGameProp2和其他类型的运算函数）。



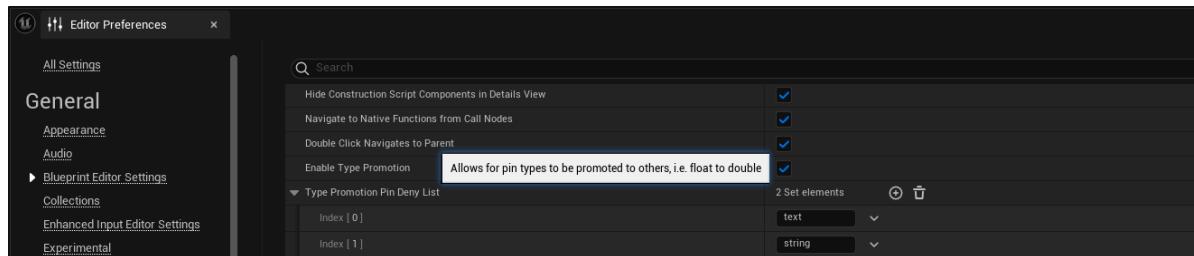
另外一点是，如果是在一个空的泛型Add节点上右键，会发现出现转换到FGameProp2的选项（但是FGameProp并没有）。这也是标明FGameProp2存在于TypePromotion这个体系里。但是实际上我们并不希望FGameProp2出现这里，还是那句话，这种玩法的战斗属性，有自己的运算规则，并不想掺和进基本类型的数学运算里。





原理：

在编辑器设置中，有个选项EnableTypePromotion打开后，就会使得FTypePromotion开始收集引擎内定义的所有函数，并判断其是否是个类型提升函数。



一个函数名如果前面包含运算符前缀（OperatorNames里定义的这些），例如Add_XXX，则会被提取操作符。被注册加入到这个FTypePromotion::OperatorTable映射表里的函数，这样在蓝图里右键一些操作符的时候（比如+），就会在这个映射表里找到最匹配的函数。

```
namespace OperatorNames
{
    static const FName NoOp = TEXT("NO_OP");

    static const FName Add = TEXT("Add");
    static const FName Multiply = TEXT("Multiply");
    static const FName Subtract = TEXT("Subtract");
    static const FName Divide = TEXT("Divide");

    static const FName Greater = TEXT("Greater");
    static const FName GreaterEq = TEXT("GreaterEqual");
    static const FName Less = TEXT("Less");
    static const FName LessEq = TEXT("LessEqual");
    static const FName NotEq = TEXT("NotEqual");
    static const FName Equal = TEXT("EqualEqual");
}

bool const bIsPromotableFunction = TypePromoDebug::IsTypePromoEnabled() &&
FTypePromotion::IsFunctionPromotionReady(Function);
if (bIsPromotableFunction)
{
    NodeClass = UK2Node_PromotableOperator::StaticClass();
}

bool FTypePromotion::IsPromotableFunction(const UFunction* Function)
{
    TRACE_CUPROFILER_EVENT_SCOPE(FTypePromotion::IsPromotableFunction);

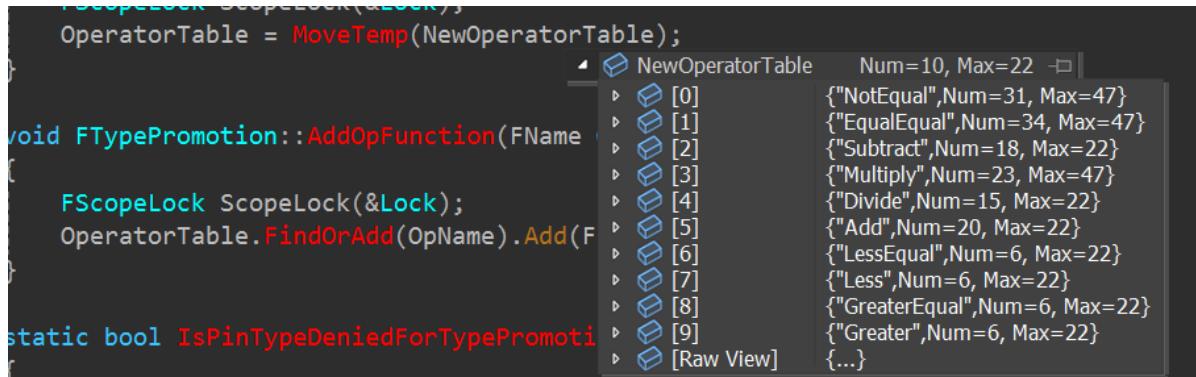
    // Ensure that we don't have an invalid OpName as well for extra safety when
    // this function
    // is called outside of this class, not during the opTable creation process
    FName OpName = GetOpNameFromFunction(Function);
    return Function &&
```

```

Function->HasAnyFunctionFlags(FUNC_BlueprintPure) &&
Function->GetReturnProperty() &&
OpName != OperatorNames::NoOp &&
!IsPinTypeDeniedForTypePromotion(Function) &&
// Users can deny specific functions from being considered for type
promotion
!Function->HasMetaData(FBlueprintMetadata::MD_IgnoreTypePromotion);
}

```

FTypePromotion收集的OperatorTable里面内容：



一个函数如果IsPromotableFunction，在调用的时候就会用UK2Node_PromotableOperator来作为蓝图节点（默认是UK2Node_CallFunction），UK2Node_PromotableOperator是典型的用于Wildcard泛型的二元运算符。如下图的Add(+)。在这种Add 的引脚上右键可以弹出Pin的类型转换从Wildcard到特定的类型，因为该结构有定义Add_XXX的函数，并且没有IgnoreTypePromotion，因此就被包含进了TypePromotion的映射表里。

上面的这个Pin转换菜单就是在UK2Node_PromotableOperator::CreateConversionMenu里收集的。

IsBlueprintBase

- 功能描述：**说明此类是否为创建蓝图的一个可接受基类，与 UCLASS 说明符、 Blueprintable 或 'NotBlueprintable` 相似。
- 使用位置：** UCLASS, UINTERFACE
- 引擎模块：** Blueprint
- 元数据类型：** bool
- 关联项：**
 - UCLASS: Blueprintable, NotBlueprintable
 - UINTERFACE: Blueprintable, NotBlueprintable
- 常用程度：** ★★★★☆

IsConversionRoot

- 功能描述：**允许Actor在自身以及子类之间做转换
- 使用位置：** UCLASS, UINTERFACE
- 引擎模块：** Blueprint
- 元数据类型：** bool
- 关联项：**
 - UCLASS: ConversionRoot

- 常用程度: ★★★

Keywords

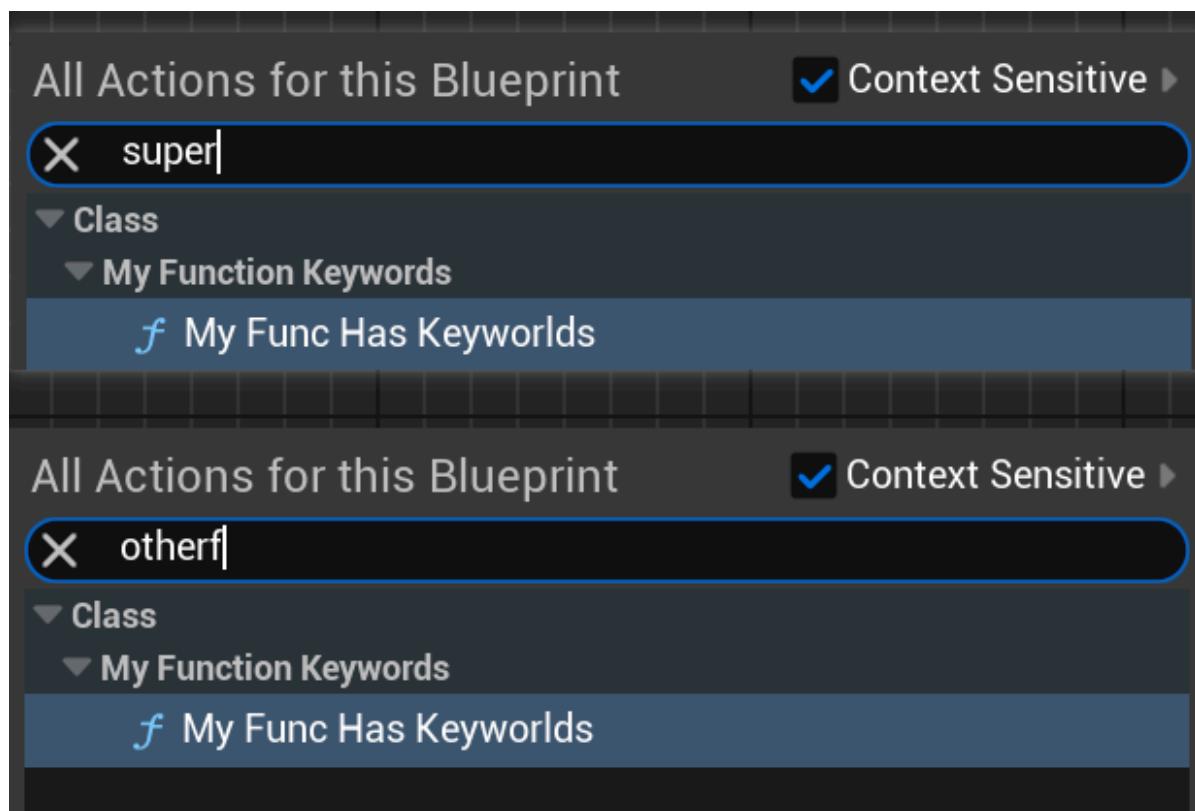
- **功能描述:** 指定一系列关键字用于在蓝图内右键找到该函数
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **常用程度:** ★★★★★

Keywords里的单词可以用空格隔开，也可以逗号隔开。这里面的文本是会被进行字符串匹配搜索。

测试代码:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Keywords :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta=(Keywords="This is a SuperFunc,OtherFunc"))
    static void MyFunc_HasKeyworlds();
};
```

蓝图效果:



原理:

该Keywords的内容，最终会被FEdGraphSchemaAction所应用，用于蓝图内右键菜单的文本搜索。

另外每个K2Node都可以返回一个Keywords。效果应该跟函数上的Keywords一样。

```
FText UEdGraphNode::GetKeywords() const
{
    return GetClass()->GetMetaDataText(TEXT("Keywords"), TEXT("UObjectKeywords"),
GetClass()->GetFullGroupName(false));
}
```

KismetHideOverrides

- **功能描述:** 不允许被覆盖的蓝图事件的列表。
- **使用位置:** UCLASS
- **引擎模块:** Blueprint
- **元数据类型:** strings="a, b, c"

在源码中发现ALevelScriptActor上面定义了很多，用来阻止被覆盖。

样例：

```
UCLASS(notplaceable, meta=(ChildCanTick, KismetHideOverrides =
"ReceiveAnyDamage,ReceivePointDamage,ReceiveRadialDamage,ReceiveActorBeginOverlap
,ReceiveActorEndOverlap,ReceiveHit,ReceiveDestroyed,ReceiveActorBeginCursorOver,R
eceiveActorEndCursorOver,ReceiveActorOnClicked,ReceiveActorOnReleased,ReceiveActo
rOnInputTouchBegin,ReceiveActorOnInputTouchEnd,ReceiveActorOnInputTouchEnter,Rece
iveActorOnInputTouchLeave"), HideCategories=(Collision,Rendering,Transformation),
MinimalAPI)
class ALevelScriptActor : public AActor
{}
```

但是实际在LevelScriptActor的子类中依然可以覆盖该事件。有一些被隐藏的Event是其实通过 HideCategories来做到的。因此该Meta其实并没有实现，如果要达到该效果，还是要通过 HideFunctions或HideCategories来达成。

OVERRIDE FUNCTION	
ActorBeginCursorOver	Actor
ActorEndCursorOver	Actor
ActorOnClicked	Actor
ActorOnReleased	Actor
AnyDamage	Actor
Async Physics Tick	Actor
BeginInputTouch	Actor
Destroyed	Actor
End Play	Actor
EndInputTouch	Actor
Level Reset	Level Script Actor
OnBecomeViewTarget	Actor
OnEndViewTarget	Actor
OnReset	Actor
PointDamage	Actor
RadialDamage	Actor
TouchEnter	Actor
TouchLeave	Actor
World Origin Location Changed	Level Script Actor

原理：

可以看到这里面的判断，并没有用到该Meta

```
void SMyBlueprint::CollectAllActions(FGraphActionListBuilderBase& OutAllActions)
{
    // Cache potentially overridable functions
    UClass* ParentClass = BlueprintObj->SkeletonGeneratedClass ? BlueprintObj-
>SkeletonGeneratedClass->GetSuperClass() : *BlueprintObj->ParentClass;
    for (TFieldIterator FunctionIt(ParentClass,
        EFieldIteratorFlags::IncludeSuper); FunctionIt; ++FunctionIt )
    {
        const* Function = *FunctionIt;
        const FName& FunctionName = Function->GetFName();

        UClass* OuterClass = CastChecked<UClass>(Function->GetOuter());
        // ignore skeleton classes and convert them into their "authoritative" types
        so they
        // can be found in the graph
        if(UBlueprintGeneratedClass* GeneratedOuterClass =
            Cast<UBlueprintGeneratedClass>(OuterClass))
        {
            OuterClass = GeneratedOuterClass->GetAuthoritativeClass();
        }

        if (UEdGraphSchema_K2::CanKismetOverrideFunction(Function)
            && !OverridableFunctionNames.Contains(FunctionName)
            && !ImplementedFunctionCache.Contains(FunctionName)
            && !ObjectEditorUtils::IsFunctionHiddenFromClass(Function, ParentClass)
            && !FBlueprintEditorUtils::FindOverrideForFunction(BlueprintObj,
            OuterClass, Function->GetFName())
    }
}
```

```

        && Blueprint->AllowFunctionOverride(Function)
    )
{
    FText FunctionTooltip =
FText::FromString(UK2Node_CallFunction::GetDefaultTooltipForFunction(Function));
    FText FunctionDesc = K2Schema->GetFriendlySignatureName(Function);
    if (FunctionDesc.IsEmpty())
    {
        FunctionDesc = FText::FromString(Function->GetName());
    }

    if (Function->HasMetaData(FBlueprintMetadata::MD_DeprecatedFunction))
    {
        FunctionDesc =
FBlueprintEditorUtils::GetDeprecatedMemberMenuItemName(FunctionDesc);
    }

    FText FunctionCategory = FObjectEditorUtils::GetCategoryText(Function);

    TSharedPtr<FEEdGraphSchemaAction_K2Graph> NewFuncAction =
MakeShareable(new
FEEdGraphSchemaAction_K2Graph(EEdGraphSchemaAction_K2Graph::Function,
FunctionCategory, FunctionDesc, FunctionTooltip, 1,
NodeSectionID::FUNCTION_OVERRIDABLE));
    NewFuncAction->FuncName = FunctionName;

    OverridableFunctionActions.Add(NewFuncAction);
    OverridableFunctionNames.Add(ClassName);
}
}
}

```

Latent

- 功能描述:** 标明一个函数是一个延迟异步操作
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 关联项:** LatentInfo, NeedsLatentFixup, LatentCallbackTarget
- 常用程度:** ★★★★☆

标明一个函数是一个延迟异步操作，需要配合LatentInfo来使用。

会导致在逻辑执行上Then（也叫Complete）引脚需要手动触发（引擎内部触发），且函数右上角增加时钟的图标。

测试代码：

```

class FMySleepAction : public FPendingLatentAction
{
public:
    float TimeRemaining;
}

```

```

FName ExecutionFunction;
int32 OutputLink;
FWeakObjectPtr CallbackTarget;

FMySleepAction(float Duration, const FLatentActionInfo& LatentInfo)
    : TimeRemaining(Duration)
    , ExecutionFunction(LatentInfo.ExecutionFunction)
    , OutputLink(LatentInfo.Linkage)
    , CallbackTarget(LatentInfo.CallbackTarget)
{
}

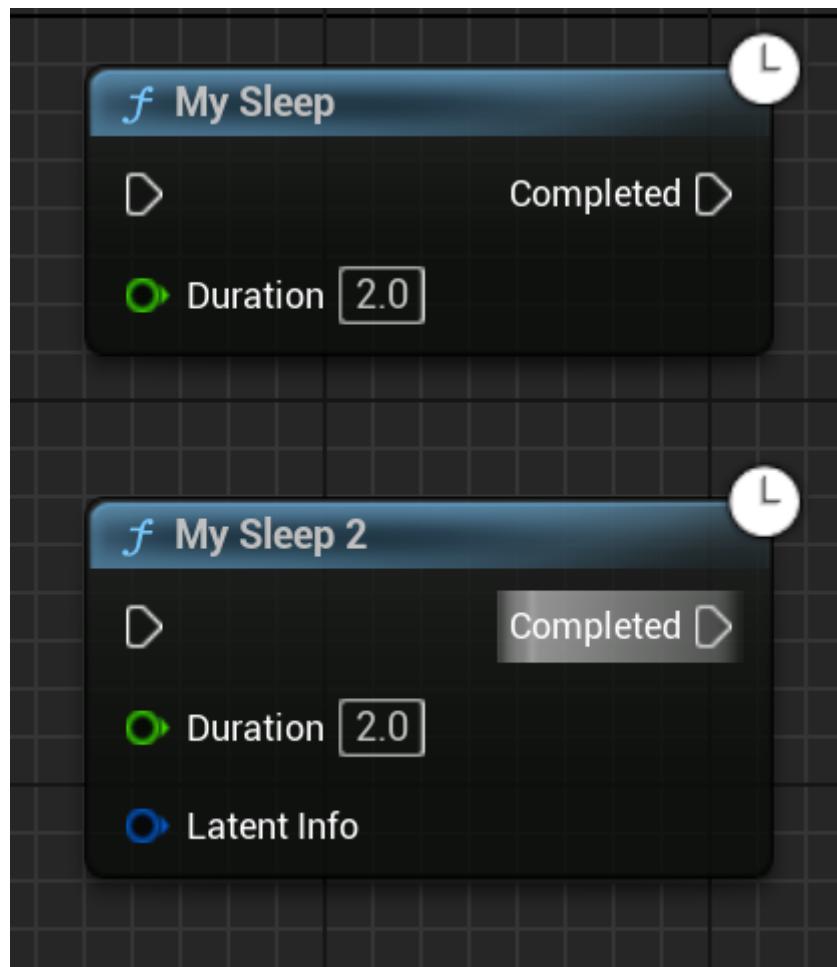
virtual void UpdateOperation(FLatentResponse& Response) override
{
    TimeRemaining -= Response.ElapsedTime();
    Response.FinishAndTriggerIf(TimeRemaining <= 0.0f, ExecutionFunction,
OutputLink, CallbackTarget);
}
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Latent :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (Latent, WorldContext =
"WorldContextObject", LatentInfo = "LatentInfo", Duration = "5"))
        static void MySleep(const UObject* WorldContextObject, float Duration,
FLatentActionInfo LatentInfo)
    {
        if (UWorld* World = GEngine-
>GetWorldFromContextObject(WorldContextObject,
EGWorldErrorMode::LogAndReturnNull))
        {
            FLatentActionManager& LatentActionManager = World-
>GetLatentActionManager();
            if (LatentActionManager.FindExistingAction<FMySleepAction>
(LatentInfo.CallbackTarget, LatentInfo.UUID) == NULL)
            {
                LatentActionManager.AddNewAction(LatentInfo.CallbackTarget,
LatentInfo.UUID, new FMySleepAction(Duration, LatentInfo));
            }
        }
    }

    UFUNCTION(BlueprintCallable, meta = (Latent, WorldContext =
"WorldContextObject", Duration = "5"))
        static void MySleep2(const UObject* WorldContextObject, float Duration,
FLatentActionInfo LatentInfo);
};

```

蓝图效果：



MySleep可以像Delay一样正常工作。但是MySleep2因为没有标明LatentInfo，因此LatentInfo函数参数没有被蓝图系统赋值，导致无法工作。

在源码里Latent使用的非常频繁，最常见的例子：

```
UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject", Latent = "", LatentInfo = "LatentInfo", DisplayName = "Load Stream Level (by Name)", Category="Game")
static ENGINE_API void LoadStreamLevel(const UObject* WorldContextObject, FName LevelName, bool bMakeVisibleAfterLoad, bool bShouldBlockOnLoad, FLatentActionInfo LatentInfo);

UFUNCTION(BlueprintCallable, meta = (Latent, LatentInfo = "LatentInfo", WorldContext = "WorldContextObject", BlueprintInternalUseOnly = "true"), Category = "Utilities")
static ENGINE_API void LoadAsset(const UObject* WorldContextObject, TSoftObjectPtr<UObject> Asset, FOnAssetLoaded OnLoaded, FLatentActionInfo LatentInfo);

UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=(Latent, WorldContext="WorldContextObject", LatentInfo="LatentInfo", Duration="0.2", Keywords="sleep"))
static ENGINE_API void Delay(const UObject* WorldContextObject, float Duration, struct FLatentActionInfo LatentInfo);
```

关于使用Latent还是继承自UBlueprintAsyncActionBase来创建蓝图异步节点的差异，可以在网上别的文章查看。

LatentCallbackTarget

- **功能描述:** 用在FLatentActionInfo::CallbackTarget属性上，告诉蓝图VM在哪个对象上调用函数。
- **使用位置:** UPROPERTY
- **元数据类型:** bool
- **关联项:** Latent
- **常用程度:** ★

用在FLatentActionInfo::CallbackTarget属性上，告诉蓝图VM在哪个对象上调用函数。

```
USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{
    GENERATED_USTRUCT_BODY()

    /** Object to execute the function on. */
    UPROPERTY(meta=(LatentCallbackTarget = true))
    TObjectPtr<UObject> CallbackTarget;

    //...
};
```

源码里作用的地方：

```
void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    // property for fixup
    FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
    (LatentInfoProperty);
    check(StructProperty->Struct == LatentInfoStruct);

    int32 StructSize = LatentInfoStruct->GetStructureSize();
    uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
    StructProperty->InitializeValue(StructData);

    // Assume that any errors on the import of the name string have been caught
    // in the function call generation
    StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

    Writer << EX_StructConst;
    Writer << LatentInfoStruct;
    Writer << StructSize;

    checksSlow(Schema);
    for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
>PropertyLinkNext)
    {
        if (TargetLabel && Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
```

```

    {
        // Emit the literal and queue a fixup to correct it once the address
        is known
        writer << EX_SkipOffsetConst;
        CodeskipSizeType PatchUpNeededAtOffset =
writer.EmitPlaceholderskip();
        JumpTargetFixupMap.Add(PatchUpNeededAtOffset,
FCodeSkipInfo(FCodeSkipInfo::Fixup, TargetLabel));
    }
    else if (Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
{
    FBPTerminal CallbackTargetTerm;
    CallbackTargetTerm.bIsLiteral = true;
    CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
    EmitTermExpr(&CallbackTargetTerm, Prop);
}
else
{
    // Create a new term for each property, and serialize it out
    FBPTerminal NewTerm;
    if(Schema->ConvertPropertyToPinType(Prop, NewTerm.Type))
    {
        NewTerm.bIsLiteral = true;
        Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

        EmitTermExpr(&NewTerm, Prop);
    }
    else
    {
        // Do nothing for unsupported/unhandled property types. This will
        leave the value unchanged from its constructed default.
        writer << EX_Nothing;
    }
}
}

writer << EX_EndStructConst;
}

```

LatentInfo

- **功能描述:** 和Latent配合，指明哪个函数参数是LatentInfo参数。
- **使用位置:** UFUNCTION
- **元数据类型:** string="abc"
- **关联项:** Latent
- **常用程度:** ★★★

Latent的函数需要FLatentActionInfo才能工作。FLatentActionInfo里记录着这个延迟操作的ID以及下一步要执行的函数名称等。在蓝图的虚拟机运行环境下，一个Latent函数执行的时候，蓝图VM会收集当前的函数上下文信息（典型的比如下Latent函数连接的下一个节点），然后继续赋值到Latent函数的FLatentActionInfo参数上，再配合FPendingLatentAction注册到FLatentActionManager里面去。等时

间到达或者触发条件达成后，FLatentActionManager会触发CallbackTarget->ProcessEvent(ExecutionFunction, &(LinkInfo.LinkID))，从而继续执行下去。

如果没有用LatentInfo来指定函数参数，则因为断了LatentInfo的赋值操作，因此就无法正常工作，蓝图效果图见Latent页面。

LatentInfo值就像WorldContext一样，会被蓝图VM系统自动的填充值。填充值的操作是在EmitLatentInfoTerm里执行的。把LatentInfoStruct的值填充到LatentInfo的函数参数里去。LatentInfo的参数位置并不重要。LatentInfo指定的函数参数Pin会被隐藏。

```
void EmitFunctionCall(FKismetCompilerContext& CompilerContext,
FKismetFunctionContext& FunctionContext, FBlueprintCompiledStatement& Statement,
UEdGraphNode* SourceNode)
{
    if (bIsUbergraph && FuncParamProperty->GetName() == FunctionToCall-
>GetMetaData(FBlueprintMetadata::MD_LatentInfo))
    {
        EmitLatentInfoTerm(Term, FuncParamProperty,
Statement.TargetLabel);
    }
}

void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    // property for fixup
    FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
(LatentInfoProperty);
    check(StructProperty->Struct == LatentInfoStruct);

    int32 StructSize = LatentInfoStruct->GetStructureSize();
    uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
    StructProperty->InitializeValue(StructData);

    // Assume that any errors on the import of the name string have been caught
    // in the function call generation
    StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

    Writer << EX_StructConst;
    Writer << LatentInfoStruct;
    Writer << StructSize;

    checksSlow(Schema);
    for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
>PropertyLinkNext)
    {
        if (TargetLabel && Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
        {
            // Emit the literal and queue a fixup to correct it once the address
            // is known
            Writer << EX_SkipOffsetConst;
            CodeSkipSizeType PatchUpNeededAtOffset =
Writer.EmitPlaceholderskip();
        }
    }
}
```

```

        JumpTargetFixupMap.Add(PatchupNeededAtOffset,
FCodeSkipInfo(FCodeSkipInfo::Fixup, TargetLabel));
    }
    else if (Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
{
    FBPTerminal CallbackTargetTerm;
    CallbackTargetTerm.bIsLiteral = true;
    CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
    EmitTermExpr(&CallbackTargetTerm, Prop);
}
else
{
    // Create a new term for each property, and serialize it out
    FBPTerminal NewTerm;
    if(Schema->ConvertPropertyToPinType(Prop, NewTerm.Type))
    {
        NewTerm.bIsLiteral = true;
        Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

        EmitTermExpr(&NewTerm, Prop);
    }
    else
    {
        // Do nothing for unsupported/unhandled property types. This will
        leave the value unchanged from its constructed default.
        Writer << EX_Nothing;
    }
}

Writer << EX_EndStructConst;
}

```

LatentInfo信息的收集是在FKCHandler_CallFunction::CreateFunctionCallStatement里

NeedsLatentFixup

- 功能描述:** 用在FLatentActionInfo::Linkage属性上，告诉蓝图VM生成跳转信息
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 关联项:** Latent
- 常用程度:** ★

在源码里找到的用处：

```
USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{
    GENERATED_USTRUCT_BODY()

    /** The resume point within the function to execute */
    UPROPERTY(meta=(NeedsLatentFixup = true))
    int32 Linkage;

    //...
};
```

源码里发挥作用的地方：

看着就是把Linkage这个属性进行单独的处理。用来在JumpTargetFixupMap里进行专门的跳转

```
void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    // property for fixup
    FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
    (LatentInfoProperty);
    check(StructProperty->Struct == LatentInfoStruct);

    int32 StructSize = LatentInfoStruct->GetStructSize();
    uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
    StructProperty->InitializeValue(StructData);

    // Assume that any errors on the import of the name string have been caught
    // in the function call generation
    StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

    Writer << EX_StructConst;
    Writer << LatentInfoStruct;
    Writer << StructSize;

    checksSlow(Schema);
    for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
    >PropertyLinkNext)
    {
        if (TargetLabel && Prop-
        >GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
        {
            // Emit the literal and queue a fixup to correct it once the address
            // is known
            Writer << EX_SkipOffsetConst;
            CodeSkipSizeType PatchUpNeededAtOffset =
            Writer.EmitPlaceholderskip();
            JumpTargetFixupMap.Add(PatchUpNeededAtOffset,
            FCodeSkipInfo(FCodeSkipInfo::Fixup, TargetLabel));
        }
    }
}
```

```

    }
    else if (Prop->GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
    {
        FBPTerminal CallbackTargetTerm;
        CallbackTargetTerm.bIsLiteral = true;
        CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
        EmitTermExpr(&CallbackTargetTerm, Prop);
    }
    else
    {
        // Create a new term for each property, and serialize it out
        FBPTerminal NewTerm;
        if(Schema->ConvertPropertyToPinType(Prop, NewTerm.Type))
        {
            NewTerm.bIsLiteral = true;
            Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

            EmitTermExpr(&NewTerm, Prop);
        }
        else
        {
            // Do nothing for unsupported/unhandled property types. This will
leave the value unchanged from its constructed default.
            writer << EX_Nothing;
        }
    }
}

writer << EX_EndStructConst;
}

```

NativeBreakFunc

- 功能描述:** 指定一个函数采用BreakStruct的图标。
- 使用位置:** UFUNCTION
- 元数据类型:** bool
- 关联项:** NativeMakeFunc
- 常用程度:** ★

其功能在NativeMakeFunc里已经说明

NativeConst

- 功能描述:** 指定有C++里的const标志
- 使用位置:** UPARAM
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 关联项:**

UPARAM: Const

- 常用程度: ★

NativeMakeFunc

- **功能描述:** 指定一个函数采用MakeStruct的图标
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **关联项:** NativeBreakFunc
- **常用程度:** ★

指定一个函数采用MakeStruct的图标。

这个函数的实际逻辑是否符合MakeStruct的规范并没有做检测，该标记只是造成显示图标的不同。因此虽然正常情况下都是搭配BlueprintPure，但是BlueprintCallable也无所谓。甚至MakeMyStructNative_Wrong函数的版本没有返回值也可以编译通过。

测试代码：

```
USTRUCT(BlueprintType)
struct FMyStruct_ForNative
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 X = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Y = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Z = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_NativeMakeBreak :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = (NativeMakeFunc))
    static FMyStruct_ForNative MakeMyStructNative(FString ValueString);

    UFUNCTION(BlueprintPure)
    static FMyStruct_ForNative MakeMyStructNative_NoMeta(FString ValueString);

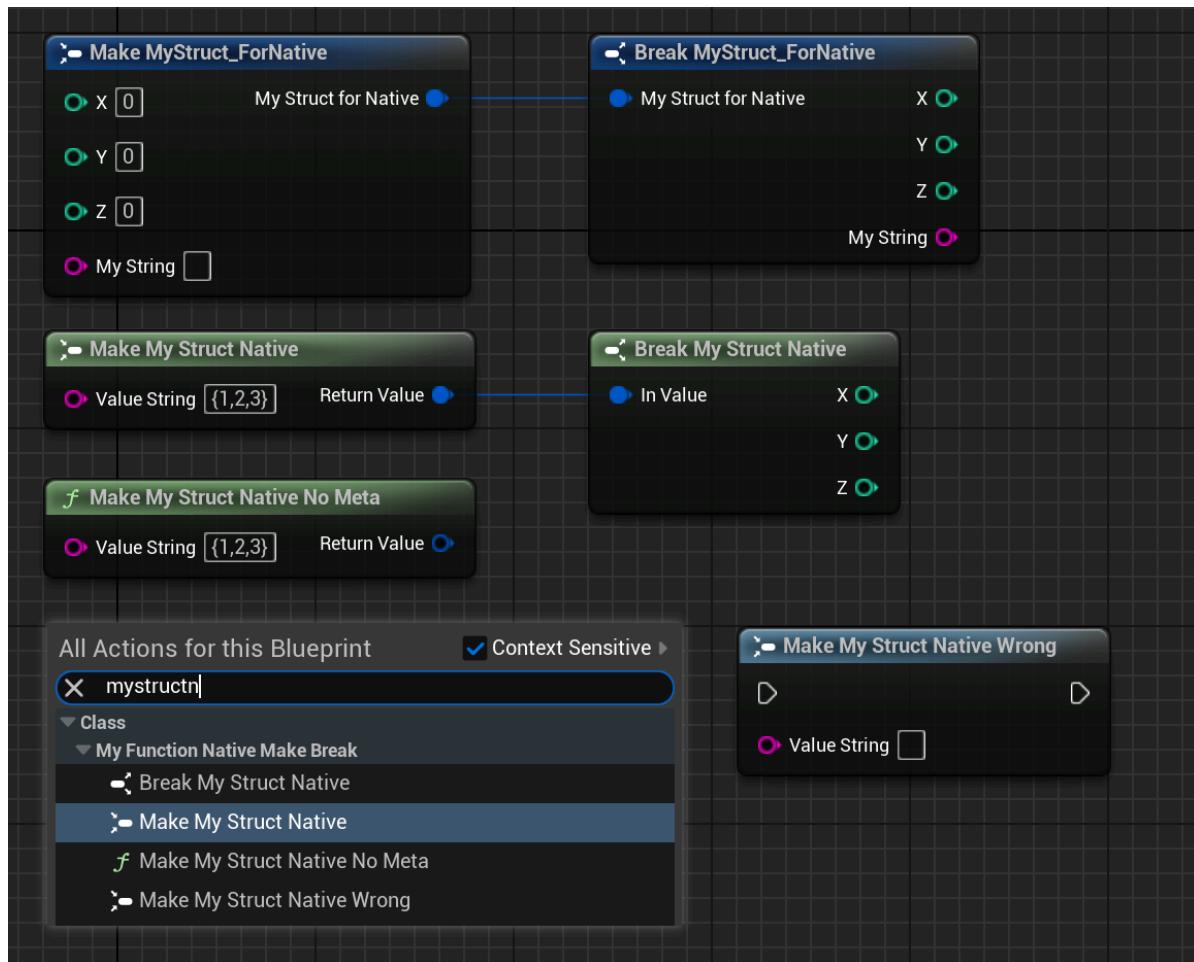
    UFUNCTION(BlueprintPure, meta = (NativeBreakFunc))
    static void BreakMyStructNative(const FMyStruct_ForNative& InValue, int32& X,
int32& Y, int32& Z);

    UFUNCTION(BlueprintCallable, meta = (NativeMakeFunc))
    static void MakeMyStructNative_Wrong(FString ValueString);
```

```
};
```

蓝图里效果：

可以看到如果是NoMeta，则函数的图标就是标准的f图标，否则则是另外的图标。同时也注意到Struct可以有多个Make和Break函数，都可以同时正常使用。



原理：

在引擎源码里唯一找到的地方是如下代码。因此该标记实际上并没有逻辑上的差别，但是在显示上会有差别。

可以看见针对NativeMakeFunc和NativeBrakeFunc采用了不同的图标。

```
FSlateIcon UK2Node_CallFunction::GetPaletteIconForFunction(UFunction const* Function, FLinearColor& OutColor)
{
    static const FName NativeMakeFunc(TEXT("NativeMakeFunc"));
    static const FName NativeBrakeFunc(TEXT("NativeBreakFunc"));

    if (Function && Function->HasMetaData(NativeMakeFunc))
    {
        static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.MakeStruct_16x");
        return Icon;
    }
    else if (Function && Function->HasMetaData(NativeBrakeFunc))
    {
```

```

        static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.BreakStruct_16x");
        return Icon;
    }

    // Check to see if the function is calling an function that could be an
    event, display the event icon instead.
    else if (Function && UEdGraphSchema_K2::FunctionCanBePlacedAsEvent(Function))
    {

        static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.Event_16x");
        return Icon;
    }
    else
    {
        outColor = GetPaletteIconColor(Function);

        static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"Kismet.AllClasses.FunctionIcon");
        return Icon;
    }
}

```

NotBlueprintThreadSafe

- 功能描述:** 用在函数上，标记这个函数是不线程安全的
- 使用位置:** UFUNCTION
- 元数据类型:** bool
- 关联项:** BlueprintThreadSafe
- 常用程度:** ★

NotInputConfigurable

- 功能描述:** 让一些UInputModifier和UInputTrigger不能在ProjectSettings里配置。
- 使用位置:** UCLASS
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 限制类型:** UInputModifier和UInputTrigger的子类
- 常用程度:** ★

让一些UInputModifier和UInputTrigger不能在ProjectSettings里配置。

源码例子：

```
UCLASS(NotBlueprintable, meta = (DisplayName = "Chorded Action",
NotInputConfigurable = "true"))
class ENHANCEDINPUT_API UInputTriggerChordAction : public UInputTrigger
{};

UCLASS(NotBlueprintable, meta = (DisplayName = "Combo (Beta)",
NotInputConfigurable = "true"))
class ENHANCEDINPUT_API UInputTriggerCombo : public UInputTrigger
{}
```

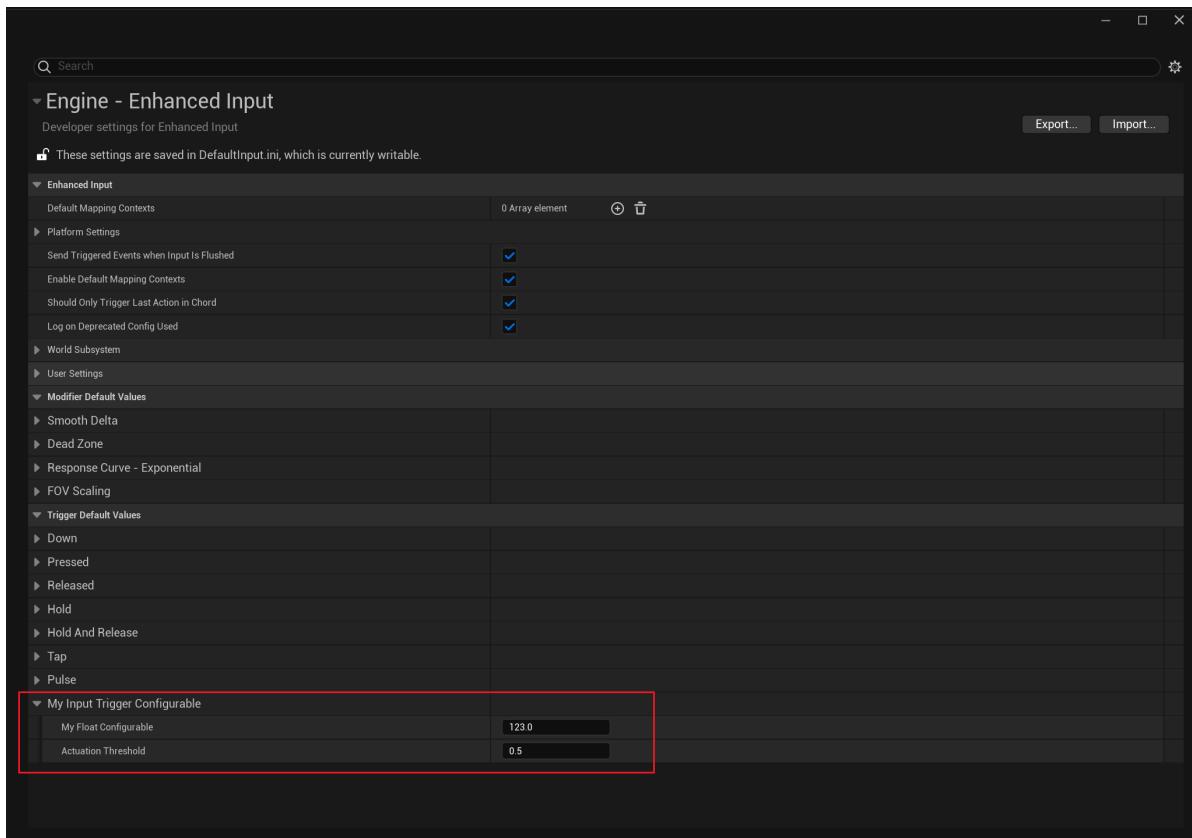
测试代码：

```
UCLASS( meta = (NotInputConfigurable = "true"))
class INSIDER_API UMyInputTrigger_NotInputConfigurable :public UInputTrigger
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    float MyFloat = 123;
};

UCLASS( meta = ())
class INSIDER_API UMyInputTrigger_Configurable :public UInputTrigger
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    float MyFloatConfigurable = 123;
};
```

测试效果：

可见只有UMyInputTrigger_Configurable 可以编辑默认值。



原理：

UEnhancedInputDeveloperSettings的UI定制化会收集UInputModifier和UInputTrigger的CDO对象，然后根据NotInputConfigurable过滤掉一些不能配置的。

```

GatherNativeClassDetailsCDOS(UInputModifier::StaticClass(), ModifierCDOS);
GatherNativeClassDetailsCDOS(UInputTrigger::StaticClass(), TriggerCDOS);

void
FEnhancedInputDeveloperSettingsCustomization::GatherNativeClassDetailsCDOS(UClass
* Class, TArray<UObject*>& CDOS)
{
    // Strip objects with no config stored properties
    CDOS.RemoveAll([Class](UObject* Object) {
        UClass* ObjectClass = Object->GetClass();
        if (ObjectClass->GetMetaData(TEXT("NotInputConfigurable")).ToBool())
        {
            return true;
        }
        while (ObjectClass)
        {
            for (FProperty* Property : TFieldRange<FProperty>(ObjectClass,
EFieldIteratorFlags::ExcludeSuper, EFieldIteratorFlags::ExcludeDeprecated))
            {
                if (Property->HasAnyPropertyFlags(CPF_Config))
                {
                    return false;
                }
            }
        }
    });
}

```

```

        // Stop searching at the base type. We don't care about
configurable properties lower than that.
        objectClass = objectClass != class ? objectClass->GetSuperClass()
: nullptr;
    }
    return true;
);
}

```

ObjectType

- 功能描述:** 指定统计页面的对象集合类型。
- 使用位置:** UClass
- 引擎模块:** Blueprint
- 元数据类型:** string="abc"
- 常用程度:** ★

指定统计页面的对象集合类型。

属于StatViewer模块，只在固定的内部几个类上使用。

源码例子：

```

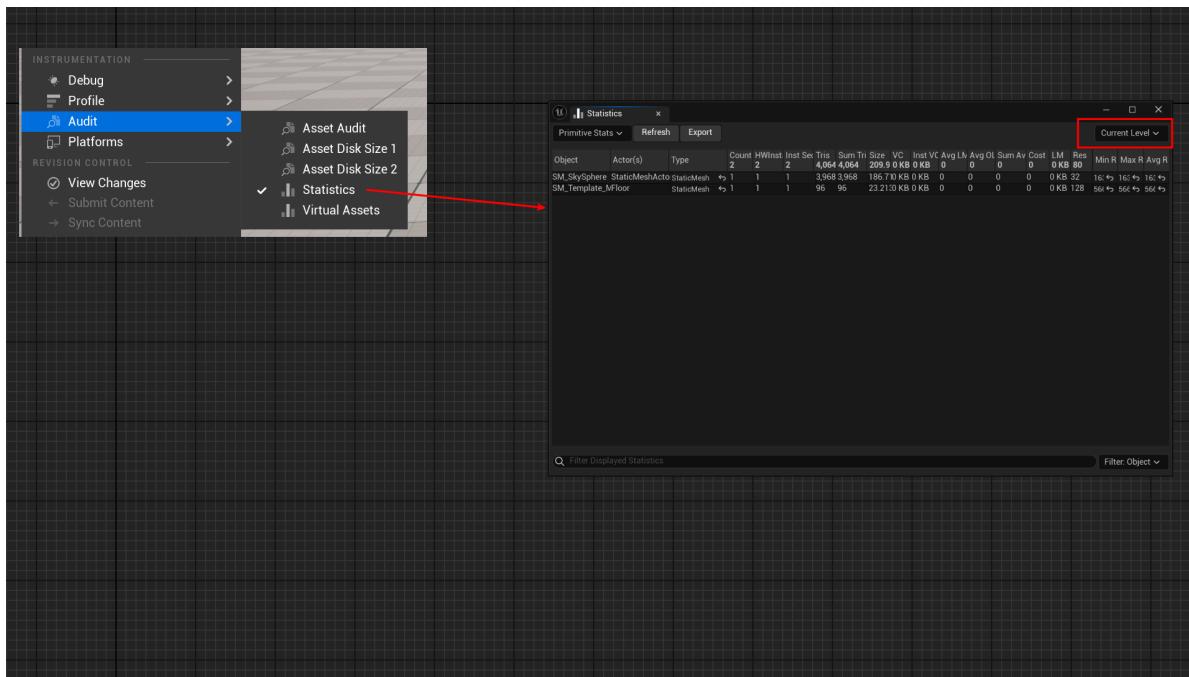
/** Enum defining the object sets for this stats object */
UENUM()
enum EPrimitiveObjectSets : int
{
    PrimitiveObjectSets_AllObjects           UMETA( DisplayName = "All objects" ,
ToolTip = "View primitive statistics for all objects in all levels" ),
    PrimitiveObjectSets_CurrentLevel         UMETA( DisplayName = "Current Level"
, ToolTip = "View primitive statistics for objects in the current level" ),
    PrimitiveObjectSets_SelectedObjects      UMETA( DisplayName = "Selected
Objects" , ToolTip = "View primitive statistics for selected objects" ),
};

/** Statistics page for primitives. */
UCLASS(Transient, MinimalAPI, meta=( DisplayName = "Primitive Stats",
ObjectSetType = "EPrimitiveObjectSets" ) )
class UPrimitiveStats : public UObject
{}

```

相应效果：

在统计页面，可见右上角的类型。



原理：

```

template <typename Entry>
class FStatsPage : public IStatsPage
{
public:
    FStatsPage()
    {
        FString EnumName = Entry::StaticClass()->GetName();
        EnumName += TEXT(".");
        EnumName += Entry::StaticClass()->GetMetaData( TEXT("ObjectType") );
        ObjectSetEnum = FindObject<UEnum>( nullptr, *EnumName );
        bRefresh = false;
        bShow = false;
        ObjectSetIndex = 0;
    }
};

```

ArrayParm

- 功能描述：**指定一个函数为使用Array<*>的函数，数组元素类型为通配符的泛型。
- 使用位置：** UFUNCTION
- 引擎模块：** Blueprint
- 元数据类型：** strings="a, b, c"
- 关联项：** ArrayTypeDependentParams
- 常用程度：** ★★★

指定一个函数为使用Array<*>的函数，数组元素类型为通配符的泛型。

在内部逻辑上的处理区别是有ArrayParm的会采用UK2Node_CallArrayFunction来生成节点，而不是UK2Node_CallFunction。

ArrayParam可以指定多个，用逗号分隔开。

在源码里只在UKismetArrayLibrary里使用，但如果自己也想顶一个数组的操作，则也可以加上ArrayParam。

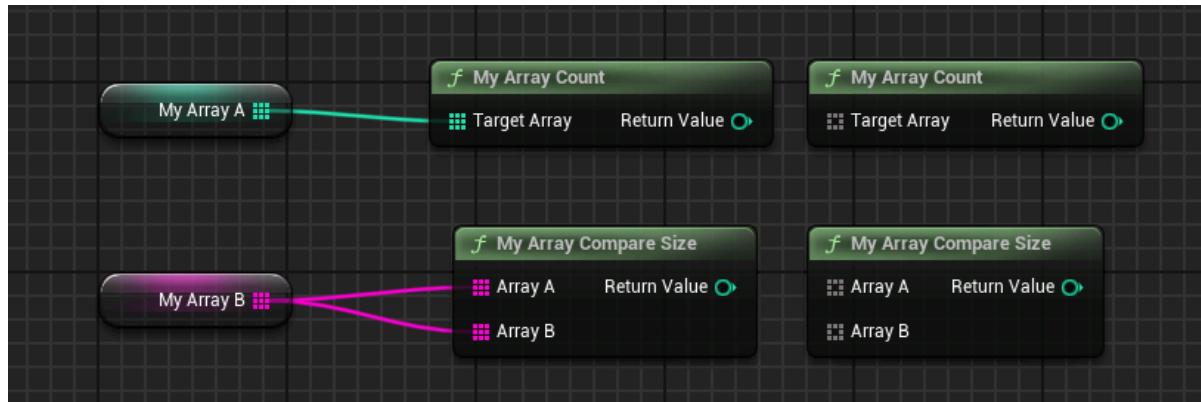
因为数组元素类型为通配符的泛型，因此在C++中实现的时候，要配合CustomThunk来自己写一些蓝图逻辑胶水代码才好正确处理不同的数组类型。这部分可以参照源码里UKismetArrayLibrary的样例模仿。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Param :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
//Array
    UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "TargetArray"))
        static int32 MyArray_Count(const TArray<int32>& TargetArray);
    static int32 GenericMyArray_Count(const void* TargetArray, const
FArrayProperty* ArrayProp);
    DECLARE_FUNCTION(execMyArray_Count);

    UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "ArrayA,ArrayB",
ArrayTypeDependentParams = "ArrayB"))
        static int32 MyArray_CompareSize(const TArray<int32>& ArrayA, const
TArray<int32>& ArrayB);
    static int32 GenericMyArray_CompareSize(void* ArrayA, const FArrayProperty*
ArrayAProp, void* ArrayB, const FArrayProperty* ArrayBProp);
    DECLARE_FUNCTION(execMyArray_CompareSize);
};
```

蓝图效果：



可以看到，在没有连接具体数组类型的时候，Array是灰色的通配符类型。而连接上不同的数组类型，Array参数引脚就会自动变成相应的类型，这些逻辑是在UK2Node_CallArrayFunction中实现的，有兴趣的去自行翻阅。

ArrayTypeDependentParams

- 功能描述：**当ArryParam指定的函数拥有两个或以上Array参数的时候，指定哪些数组参数的类型也应该相应的被更新改变。
- 使用位置：** UFUNCTION
- 元数据类型：** string="abc"

- **关联项:** ArrayParm

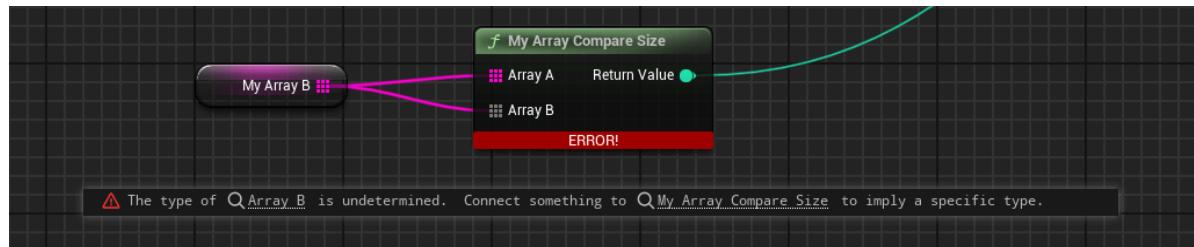
当ArryParam指定的函数拥有两个或以上Array参数的时候，指定哪些数组参数的类型也应该相应的被更新改变。

指明一个参数的类型，用于确定ArrayParam的值类型

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Param :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
//Array

    UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "ArrayA,ArrayB",
    ArrayTypeDependentParams = "ArrayB"))
        static int32 MyArray_CompareSize(const TArray<int32>& ArrayA, const
    TArray<int32>& ArrayB);
        static int32 GenericMyArray_CompareSize(void* ArrayA, const FArrayProperty* ArrayAProp, void* ArrayB, const FArrayProperty* ArrayBProp);
    DECLARE_FUNCTION(execMyArray_CompareSize);
};
```

如果没有ArrayTypeDependentParams，在连接ArrayA后，ArrayB的类型依然没有确定，即使连接上了也是如此，这应该是引擎的实现所限制。编译会造成编译错误。



因此ArrayTypeDependentParams可以指定另外的数组参数，其类型会由别的（第一个）数组实际参数所决定，即typeof(ArrayB)=typeof(ArrayA)。在示例代码里所示加上ArrayB作为ArrayTypeDependentParams之后，MyArrayB无论是先连接到ArrayA还是ArrayB都可以触发二者改变为一致的数组类型。这是因为ArrayA作为第一个参数，天生在引擎内已经实现了第一个参数的动态类型实时变化。因此我们只要再加上ArrayB就好了。

原理：

引擎内已经实现了第一个参数的动态类型实时变化：

```
void UK2Node_CallArrayFunction::AllocateDefaultPins()
{
    Super::AllocateDefaultPins();

    UEdGraphPin* TargetArrayPin = GetTargetArrayPin();
    if (ensureMsgf(TargetArrayPin, TEXT("%s"), *GetFullName()))
    {
        TargetArrayPin->PinType.ContainerType = EPinContainerType::Array;
        TargetArrayPin->PinType.bIsReference = true;
        TargetArrayPin->PinType.PinCategory = UEdGraphSchema_K2::PC_Wildcard;
        TargetArrayPin->PinType.PinSubCategory = NAME_None;
```

```

    TargetArrayPin->PinType.PinSubCategoryObject = nullptr;
}

TArray< FArrayPropertyPinCombo > ArrayPins;
GetArrayPins(ArrayPins);
for(auto Iter = ArrayPins.CreateConstIterator(); Iter; ++Iter)
{
    if(Iter->ArrayPropPin)
    {
        Iter->ArrayPropPin->bHidden = true;
        Iter->ArrayPropPin->bNotConnectable = true;
        Iter->ArrayPropPin->bDefaultValueIsReadOnly = true;
    }
}

PropagateArrayTypeInfo(TargetArrayPin);
}

```

关于ArrayDependentParam的作用机制，可以参照UK2Node_CallArrayFunction里的NotifyPinConnectionListChanged和PropagateArrayTypeInfo这两个函数的实现，可以看到其他的数组参数Pin类型被动态的修改为SourcePin的类型。

AutoCreateRefTerm

- 功能描述：**指定函数的多个输入引用参数在没有连接的时候自动为其创建默认值
- 使用位置：** UFUNCTION
- 引擎模块：** Blueprint
- 元数据类型：** strings="a, b, c"
- 常用程度：** ★★★★☆

指定函数的多个输入引用参数在没有连接的时候自动为其创建默认值。

要注意“输入”+“引用”，这两个前提项。

当有些情况你想把函数的参数采用引用来传递，但是又不想每次都得必须连接一个变量，想在不连接的时候提供一个内联编辑的默认值，因此蓝图系统就提供了这么一个便利功能。

测试代码：

```

UFUNCTION(BlueprintCallable, meta = (AutoCreateRefTerm = "Location,Value"))
static bool MyFunc_HasAutoCreateRefTerm(const FVector& Location, const int32& value) { return false; }

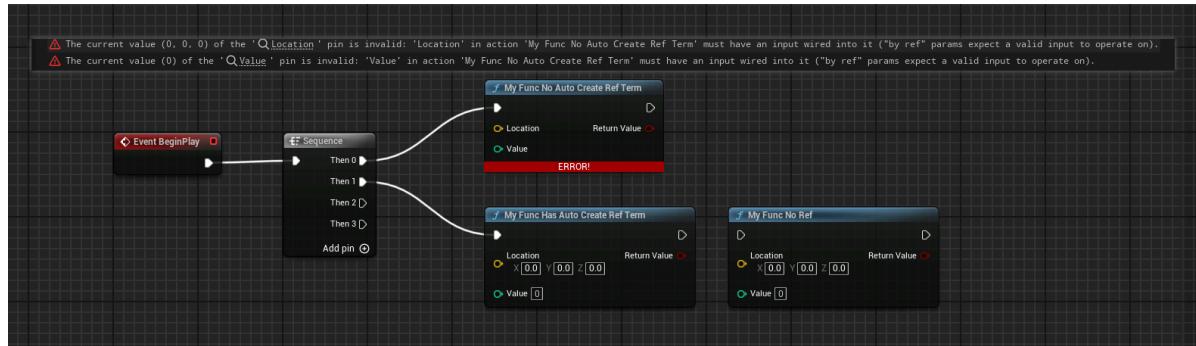
UFUNCTION(BlueprintCallable)
static bool MyFunc_NoAutoCreateRefTerm(const FVector& Location, const int32& value) { return false; }

UFUNCTION(BlueprintCallable)
static bool MyFunc_NoRef(FVector Location, int32 Value) { return false; }

```

蓝图效果：

可以见到MyFunc_NoAutoCreateRefTerm的函数会产生编译的报错，因为是引用参数但是却没有连接，导致引用缺少实参。



原理代码：

```
void UEdGraphSchema_K2::GetAutoEmitTermParameters(const UFunction* Function,
TArray< FString>& AutoEmitParameterNames)
{
    AutoEmitParameterNames.Reset();

    const FString& MetaData = Function->GetMetaData(FBlueprintMetadata::MD_AutoCreateRefTerm);
    if (!MetaData.IsEmpty())
    {
        MetaData.ParseIntoArray(AutoEmitParameterNames, TEXT(","), true);

        for (int32 NameIndex = 0; NameIndex < AutoEmitParameterNames.Num();)
        {
            FString& ParameterName = AutoEmitParameterNames[NameIndex];
            ParameterName.TrimStartAndEndInline();
            if (ParameterName.IsEmpty())
            {
                AutoEmitParameterNames.RemoveAtSwap(NameIndex);
            }
            else
            {
                ++NameIndex;
            }
        }
    }

    // Allow any params that are blueprint defined to be autocreated:
    if (!FBpBlueprintEditorUtils::IsNativeSignature(Function))
    {
        for (TFieldIterator< FProperty > ParamIter(Function,
EFieldIterationFlags::Default);
             ParamIter && (ParamIter->PropertyFlags & CPF_Parm);
             ++ParamIter)
        {
            FProperty* Param = *ParamIter;
            if(Param->HasAnyPropertyParams(CPF_ReferenceParm))
            {
                AutoEmitParameterNames.Add(Param->GetName());
            }
        }
    }
}
```

```

        }
    }
}

还有在
void UK2Node_CallFunction::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
if ( Function )
{
    TArray< FString> AutoCreateRefTermPinNames;
    CompilerContext.GetSchema()->GetAutoEmitTermParameters(Function,
AutoCreateRefTermPinNames);
    const bool bHasAutoCreateRefTerms = AutoCreateRefTermPinNames.Num() != 0;

    for (UEdGraphPin* Pin : Pins)
    {
        const bool bIsRefInputParam = Pin && Pin->PinType.bIsReference && (Pin-
>Direction == EGPD_Input) && !CompilerContext.GetSchema()->IsMetaPin(*Pin);
        if (!bIsRefInputParam)
        {
            continue;
        }

        const bool bHasConnections = Pin->LinkedTo.Num() > 0;
        const bool bCreateDefaultValRefTerm = bHasAutoCreateRefTerms &&
            !bHasConnections && AutoCreateRefTermPinNames.Contains(Pin-
>PinName.ToString());

        if (bCreateDefaultValRefTerm)
        {
            const bool bHasDefaultValue = !Pin->DefaultValue.IsEmpty() || Pin-
>DefaultObject || !Pin->DefaultTextValue.IsEmpty();

            // copy defaults as default values can be reset when the pin is
connected
            const FString DefaultValue = Pin->DefaultValue;
            Uobject* DefaultObject = Pin->DefaultObject;
            const FText DefaultTextValue = Pin->DefaultTextValue;
            bool bMatchesDefaults = Pin->DoesDefaultValueMatchAutogenerated();

            UEdGraphPin* ValuePin = InnerHandleAutoCreateRef(this, Pin,
CompilerContext, SourceGraph, bHasDefaultValue);
            if ( ValuePin )
            {
                if (bMatchesDefaults)
                {
                    // Use the latest code to set default value
                    Schema->SetPinAutogeneratedDefaultValueBasedOnType(ValuePin);
                }
                else
                {
                    ValuePin->DefaultValue = DefaultValue;
                    ValuePin->DefaultObject = DefaultObject;
                    ValuePin->DefaultTextValue = DefaultTextValue;
                }
            }
        }
    }
}

```

```

        }
    }

    // since EX_Self does not produce an addressable (referenceable)
FProperty, we need to shim
    // in a "auto-ref" term in its place (this emulates how UHT generates a
local value for
    // native functions; hence the IsNative() check)
    else if (bHasConnections && Pin->LinkedTo[0]->PinType.PinSubCategory ==
UEdGraphSchema_K2::PSC_Self && Pin->PinType.bIsConst && !Function->IsNative())
    {
        InnerHandleAutoCreateRef(this, Pin, CompilerContext, SourceGraph,
/*bForceAssignment =*/true);
    }
}
}

```

CustomStructureParam

- 功能描述:** 被CustomStructureParam标记的函数参数会变成Wildcard的通配符参数，其引脚的类型会等于连接的变量类型。
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** strings="a, b, c"
- 常用程度:** ★★★★☆

被CustomStructureParam标记的多个函数参数会变成Wildcard的通配符参数，其引脚的类型会等于连接的变量类型。

CustomStructureParam总是和CustomThunk一起配合使用，这样才能在自己的函数体内来处理泛型的参数类型。

```

UFUNCTION(BlueprintCallable, CustomThunk, meta = (DisplayName =
"PrintStructFields", CustomStructureParam = "inputStruct"))
static FString PrintStructFields(const int32& inputStruct) { return TEXT(""); }

DECLARE_FUNCTION(execPrintStructFields);
static FString Generic_PrintStructFields(const UScriptStruct* ScriptStruct, const
void* StructData);

DEFINE_FUNCTION(UMyFunction_Custom::execPrintStructFields)
{
    FString result;
    Stack.MostRecentPropertyAddress = nullptr;
    Stack.StepCompiledIn<FStructProperty>(nullptr);

    void* StructData = Stack.MostRecentPropertyAddress;
    FStructProperty* StructProperty = CastField<FStructProperty>
(Stack.MostRecentProperty);
    UScriptStruct* ScriptStruct = StructProperty->Struct;
    P_FINISH;
    P_NATIVE_BEGIN;
}

```

```

        result = Generic_PrintStructFields(ScriptStruct, StructData);

    P_NATIVE_END;
    *(FString*)RESULT_PARAM = result;
}

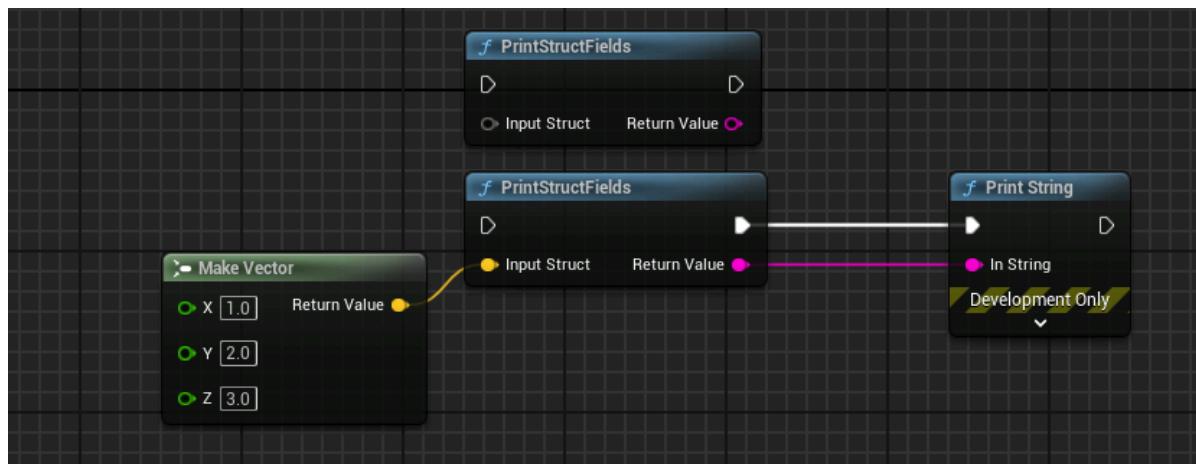
FString UMyFunction_Custom::Generic_PrintStructFields(const UScriptStruct*
ScriptStruct, const void* StructData)
{
    FString str;
    for (TFieldIterator<FProperty> i(ScriptStruct); i; ++i)
    {
        FString propertyValueString;
        const void* PropertyValuePtr = i->ContainerPtrToValuePtr<const void*>
(StructData);
        i->ExportTextItem_Direct(propertyValueString, PropertyValuePtr, nullptr,
(UObject*)ScriptStruct, PPF_None);

        str += FString::Printf(TEXT("%s:%s\n"), *i->GetFName().ToString(),
*propertyValueString);
    }

    return str;
}

```

蓝图中的效果：



可以看到定义了一个接受通用结构参数的节点，然后打印出内部所有的属性。其中 CustomStructureParam 指定函数的参数是自定义的类型。

源码中的典型例子是

```

UFUNCTION(BlueprintCallable, CustomThunk, Category = "DataTable", meta=
(CustomStructureParam = "OutRow", BlueprintInternalUseOnly="true"))
static ENGINE_API bool GetDataTableRowFromName(UDataTable* Table, FName RowName,
FTableRowBase& OutRow);

```

原理：

首先拥有CustomStructureParam的参数会被识别为Wildcard属性。然后通过FCustomStructureParamHelper来控制Pin->PinType = LinkedTo->PinType;，从而改变Pin的实际类型。

```
bool UEdGraphSchema_K2::IsWildcardProperty(const FPropertyParams*PropertyParams)
{
    UFunction* Function =PropertyParams->GetOwner() ;

    return Function && ( UK2Node_CallArrayFunction::IsWildcardPropertyParams(Function,
PropertyParams)
        || UK2Node_CallFunction::IsStructureWildcardPropertyParams(Function,PropertyParams-
>GetFName())
        || UK2Node_CallFunction::IsWildcardPropertyParams(Function,PropertyParams)
        || FEdGraphUtilities::IsArrayDependentPropertyParams(Function,PropertyParams-
>GetFName()) );
}

static void FCustomStructureParamHelper::HandlesinglePin(UEdGraphPin* Pin)
{
    if (Pin)
    {
        if (Pin->LinkedTo.Num() > 0)
        {
            UEdGraphPin* LinkedTo = Pin->LinkedTo[0];
            check(LinkedTo);

            if (UK2Node* Node = Cast<UK2Node>(Pin->GetOwningNode()))
            {
                ensure(
                    !LinkedTo->PinType.IsContainer() ||
                    Node->DoesWildcardPinAcceptContainer(Pin)
                );
            }
            else
            {
                ensure( !LinkedTo->PinType.IsContainer() );
            }

            Pin->PinType = LinkedTo->PinType;
        }
        else
        {
            // constness and refness are controlled by our declaration
            // but everything else needs to be reset to default wildcard:
            const bool bWasRef = Pin->PinType.bIsReference;
            const bool bWasConst = Pin->PinType.bIsConst;

            Pin->PinType = FEdGraphPinType();
            Pin->PinType.bIsReference = bWasRef;
            Pin->PinType.bIsConst = bWasConst;
            Pin->PinType.PinCategory = UEdGraphSchema_K2::PC_Wildcard;
            Pin->PinType.PinSubCategory = NAME_None;
            Pin->PinType.PinSubCategoryObject = nullptr;
        }
    }
}
```

```
        }
    }
}
```

DeterminesOutputType

- **功能描述:** 指定一个参数的类型作为函数动态调整输出参数类型的参考类型
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **限制类型:** Class或Object指针类型，或容器类型
- **关联项:** DynamicOutputParam
- **常用程度:** ★★★

指定一个参数的类型作为函数输出参数的类型。

假定这么一个函数原型：

```
UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"A",DynamicOutputParam="P1,P2"))
TypeR MyFunc(TypeA A,Type1 P1,Type2 P2,Type3 P3);
```

DeterminesOutputType的值指定了一个函数参数名称，即A。其TypeA的类型必须是Class或Object，一般是TSubClassOf 或者XXX*，当然也可以是TArray<XXX>，还可以是指向参数结构里的某个属性。如Args_ActorClassType。TSoftObjectPtr也是可以的，指向一个子类Asset对象，然后输出的基类Asset就可以相应改变。

所谓输出参数包括返回值和函数的输出参数，因此上述函数原型里的TypeR,P1,P2都是输出参数。为了让输出参数的类型也相应变化，TypeR、Type1和Type2的类型也得是Class或Object类型，且A参数在蓝图节点上实际选择的类型必须是输出参数类型的子类，这样才能自动转换过去。

如果没有P1和P2，只把返回值当作TypeR，则可以不指定DynamicOutputParam也可以自动默认把返回值当作动态的输出参数。否则则需要手动书写DynamicOutputParam来指定哪些函数参数来支持动态类型。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyAnimalActor :public AActor
{
public:
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyCatActor :public AMyAnimalActor
{
public:
    GENERATED_BODY()
};
```

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyDogActor :public AMyAnimalActor
{
public:
    GENERATED_BODY()
};

USTRUCT(BlueprintType)
struct FMyOutputTypeArgs
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 1;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TSubclassOf ActorClassType;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_OutputTypeTest :public
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    //class
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"ActorClassType"))
    static TArray<AActor*> MyGetAnimals(TSubclassOf

```

```

    static TArray<AActor*> MyGetAnimalsWithStructProperty(const
FMyOutputTypeArgs& Args);
}

```

蓝图中效果：

用返回值当作输出参数的例子，注意到返回值类型实际变成了TArray<AMyCatActor*>。



也可以加上DynamicOutputParam来指定输出参数作为动态类型参数：



DynamicOutputParam可以指定多个参数



DeterminesOutputType 的参数类型也可以是Object或者Object的容器：



DeterminesOutputType 的参数甚至可以是结构里的某个属性，但是只有SplitStruct的时候才生效，因为这个时候结构的属性变量才变成函数的Pin，才可以进行DeterminesOutputType的名称比对。这个时候要书写成“`A_B`”，而不是“`A.B`”。



原理：

DeterminesOutputType的作用机制是根据这个名称去函数蓝图节点上查找Pin，这个Pin得是Class或Object类型的（容器也行），因为必须是这二者才支持指针类型的转换。这个Pin在蓝图节点上是会由各种TypePicker来实际指定值，比如ClassPicker或ObjectPicker。之后根据TypePicker选择的值，就可以相应的调整DynamicOutputParam指定的参数的类型（或返回参数），真正发挥类型改变的是

`Pin->PinType.PinSubCategoryObject = PickedClass;`这一句。

```

void FDynamicOutputHelper::ConformOutputType() const
{
    if (IsTypePickerPin(MutatingPin))
    {
        UClass* PickedClass = GetPinClass(MutatingPin);
        UK2Node_CallFunction* FuncNode = GetFunctionNode();

        // See if there is any dynamic output pins
        TArray<UEdGraphPin*> DynamicPins;
        GetDynamicOutPins(FuncNode, DynamicPins);

        // Set the pins class
        for (UEdGraphPin* Pin : DynamicPins)
        {
            if (ensure(Pin != nullptr))
            {
                Pin->PinType.PinSubCategoryObject = PickedClass; // 设定每个动态
参数的子类型
            }
        }
    }
}

```

```
    }  
}
```

DynamicOutputParam

- **功能描述:** 配合DeterminesOutputType，指定多个支持动态类型的输出参数。
- **使用位置:** UFUNCTION
- **元数据类型:** strings="a, b, c"
- **限制类型:** Class或Object指针类型，或容器类型
- **关联项:** DeterminesOutputType

常常和DeterminesOutputType一起配合。动态参数的数量可以为多个。

HideSpawnParms

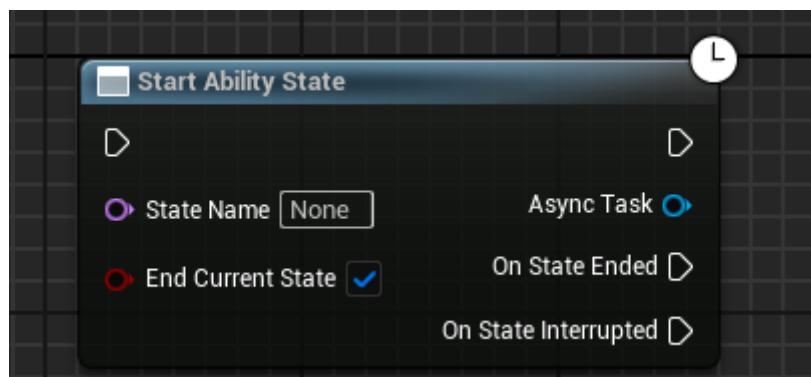
- **功能描述:** 在UGamelayTask子类生成的蓝图异步节点上隐藏UGamelayTask子类继承链中某些属性。
- **使用位置:** UFUNCTION
- **元数据类型:** strings="a, b, c"
- **关联项:** ExposedAsyncProxy

在UGamelayTask子类生成的蓝图异步节点上隐藏UGamelayTask子类继承链中某些属性。

HideSpawnParms 只在UK2Node_LatentGameplayTaskCall中判断，因此只作用于UGameplayTask的子类。在源码中找到的唯一用法是 HideSpawnParms = "Instigator"，但是其UGamelayTask子类继承链中并无该属性，因此其实是不发挥作用的。

```
UFUNCTION(BlueprintCallable, Meta = (HidePin = "OwningAbility", DefaultToSelf  
= "OwningAbility", BlueprintInternalUseOnly = "true", HideSpawnParms =  
"Instigator"), Category = "Ability|Tasks")  
    static UAbilityTask_StartAbilityState* StartAbilityState(UGameplayAbility*  
OwningAbility, FName StateName, bool bEndCurrentState = true);
```

保留和去掉HideSpawnParms 的蓝图的节点都为：



源码里发生的位置：

```
void UK2Node_LatentGameplayTaskCall::CreatePinsForClass(UClass* InClass)  
{
```

```

// Tasks can hide spawn parameters by doing meta =
(HideSpawnParms="PropertyA,PropertyB")
// (For example, hide Instigator in situations where instigator is not
relevant to your task)

TArray< FString> IgnorePropertyList;
{
    UFunction* ProxyFunction = ProxyFactoryClass-
>FindFunctionByName(ProxyFactoryFunctionName);

    const FString& IgnorePropertyListStr = ProxyFunction-
>GetMetaData(FName(TEXT("HideSpawnParms")));

    if (!IgnorePropertyListStr.IsEmpty())
    {
        IgnorePropertyListStr.ParseIntoArray(IgnorePropertyList, TEXT(","),
true);
    }
}
}

```

MapKeyParam

- 功能描述:** 指定一个函数参数为Map的Key，其根据MapParam指定的实际Map参数的Key类型而相应改变。
- 使用位置:** UFUNCTION
- 元数据类型:** string="abc"
- 限制类型:** TMap
- 关联项:** MapParam
- 常用程度:** ★★★

MapParam

- 功能描述:** 指定一个函数为使用TMap< TKey, TValue>的函数，元素类型为通配符的泛型。
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** string="abc"
- 限制类型:** TMap
- 关联项:** MapKeyParam, MapValueParam
- 常用程度:** ★★★

指定一个函数为使用TMap< TKey, TValue>的函数，元素类型为通配符的泛型。

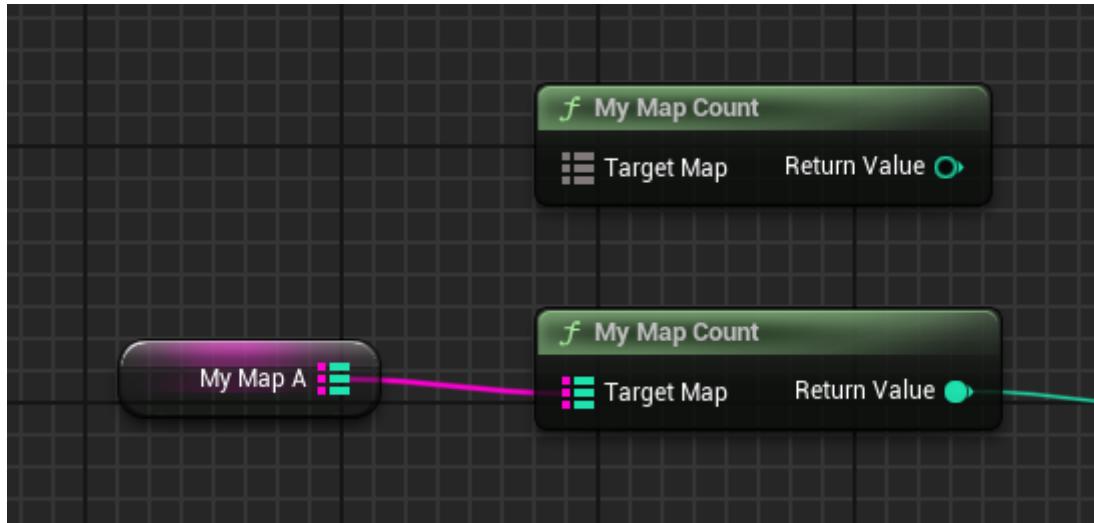
只能支持一个MapParam，源码中的实现是只根据一个名字来FindPin。

在源码中，例子都是在UBlueprintMapLibrary中使用。

测试代码1：

```
UFUNCTION(BlueprintPure, CustomThunk, meta = (MapParam = "TargetMap"))
static int32 MyMap_Count(const TMap<int32, int32>& TargetMap);
static int32 GenericMyMap_Count(const void* TargetMap, const FMapProperty* MapProperty);
DECLARE_FUNCTION(execMyMap_Count);
```

蓝图中效果1：

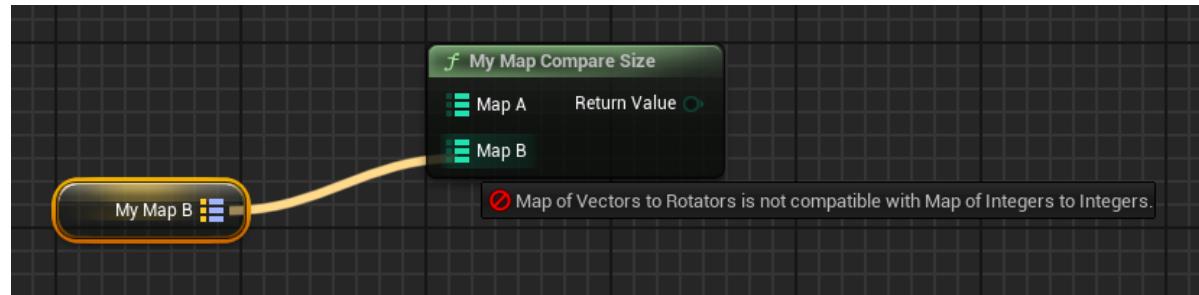


因为只支持一个MapParam，因此如果你书写这种代码。

测试代码2：

```
UFUNCTION(BlueprintPure, CustomThunk, meta = (MapParam = "MapA,MapB"))
static int32 MyMap_CompareSize(const TMap<int32, int32>& MapA, const TMap<int32, int32>& MapB);
static int32 GenericMyMap_CompareSize(void* MapA, const FMapProperty* MapAProp, void* MapB, const FMapProperty* MapBProp);
DECLARE_FUNCTION(execMyMap_CompareSize);
```

会导致MapParam搜索不到Pin，从而失去通配符的功能。



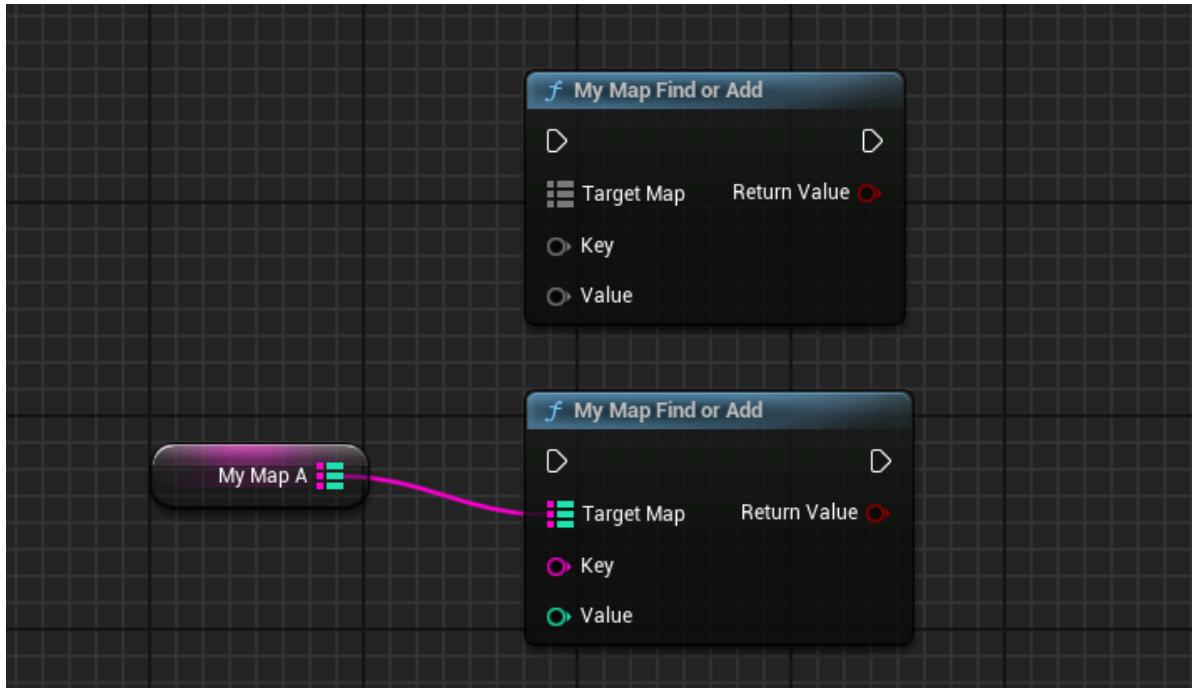
而如果要实现类似Add的功能，达到Key和Value的Pin类型也可以动态的根据Map的类型而自动的改变。则需要加上MapKeyParam 和MapViewParam 分别的指定另外的函数参数以便能找到正确的Pin，从而实现动态的根据Map类型而更改KeyValue Pin类型。MapKeyParam 和MapViewParam 指定的参数也可以为数组等容器，可以参照UBlueprintMapLibrary中的Keys和Values参数。

```

UFUNCTION(BlueprintCallable, CustomThunk, meta = (MapParam =
"TargetMap", MapKeyParam = "Key", MapValueParam = "Value"))
    static bool MyMap_FindOrAdd(const TMap<int32, int32>& TargetMap, const int32&
Key, const int32& value);
    static bool GenericMyMap_FindOrAdd(const void* TargetMap, const FMapProperty*&
MapProperty, const void* KeyPtr, const void* ValuePtr);
DECLARE_FUNCTION(execMyMap_FindOrAdd);

```

蓝图中的效果2：



原理代码：

```

void UK2Node_CallFunction::ConformContainerPins()
{
    //在这其中检测容器Pin
    const FString& MapPinMetaData = TargetFunction-
>GetMetaData(FBlueprintMetadata::MD_MapParam);
    const FString& MapKeyPinMetaData = TargetFunction-
>GetMetaData(FBlueprintMetadata::MD_MapKeyParam);
    const FString& MapValuePinMetaData = TargetFunction-
>GetMetaData(FBlueprintMetadata::MD_MapValueParam);

    if(!MapPinMetaData.IsEmpty() || !MapKeyPinMetaData.IsEmpty() ||
!MapValuePinMetaData.IsEmpty())
    {
        // if the map pin has a connection infer from that, otherwise use the
        information on the key param and value param:
        bool bReadyToPropagateKeyType = false;
        FEdGraphTerminalType KeyTypeToPropagate;
        bool bReadyToPropagateValueType = false;
        FEdGraphTerminalType ValueTypeToPropagate;

        UEdGraphPin* MapPin = MapPinMetaData.IsEmpty() ? nullptr :
FindPin(MapPinMetaData);

```

```

        UEdGraphPin* MapKeyPin = MapKeyPinMetaData.IsEmpty() ? nullptr :
FindPin(MapKeyPinMetaData);
        UEdGraphPin* MapValuePin = MapValuePinMetaData.IsEmpty() ? nullptr :
FindPin(MapValuePinMetaData);

        TryReadTypeToPropagate(MapPin, bReadyToPropagateKeyType,
KeyTypeToPropagate); //读取MapPin的Key连接类型
        TryReadValueTypeToPropagate(MapPin, bReadyToPropagateValueType,
ValueTypeToPropagate); //读取MapPin上连接的Map Value类型
        TryReadTypeToPropagate(MapKeyPin, bReadyToPropagateKeyType,
KeyTypeToPropagate); //读取KeyPin上的连接类型
        TryReadTypeToPropagate(MapValuePin, bReadyToPropagateValueType,
ValueTypeToPropagate); //读取valuePin上的连接类型

        TryPropagateType(MapPin, KeyTypeToPropagate,
bReadyToPropagateKeyType); //改变MapPin的Key当前类型
        TryPropagateType(MapKeyPin, KeyTypeToPropagate,
bReadyToPropagateKeyType); //改变KeyPin的当前类型

        TryPropagateValueType(MapPin, ValueTypeToPropagate,
bReadyToPropagateValueType); //改变MapPin的Value当前类型
        TryPropagateType(MapValuePin, ValueTypeToPropagate,
bReadyToPropagateValueType); //改变valuePin的当前类型
    }
}

```

MapViewParam

- 功能描述:** 指定一个函数参数为Map的Value，其根据MapParam指定的实际Map参数的Value类型而相应改变。
- 使用位置:** UFUNCTION
- 元数据类型:** string="abc"
- 限制类型:** TMap
- 关联项:** MapParam
- 常用程度:** ★★★

ProhibitedInterfaces

- 功能描述:** 列出与蓝图类不兼容的接口，阻止实现
- 使用位置:** UCLASS
- 引擎模块:** Blueprint
- 元数据类型:** strings="a, b, c"
- 常用程度:** ★★

测试代码:

```

UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_First:public UInterface
{
    GENERATED_UINTERFACE_BODY()
}

```

```

};

class INSIDER_API IMyInterface_First
{
    GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    void FirstFunc() const;
};

UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_Second:public UIInterface
{
    GENERATED_UINTERFACE_BODY()
};

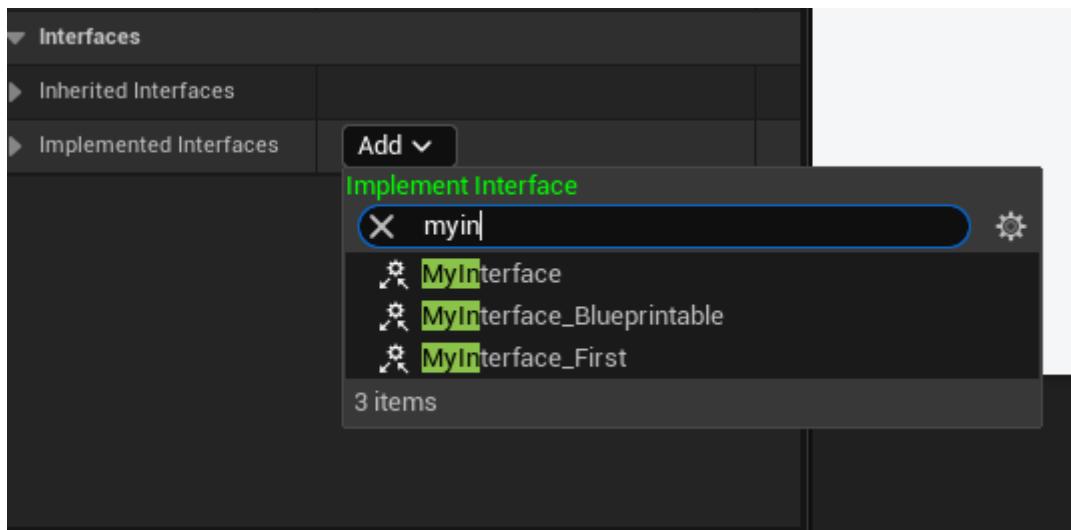
class INSIDER_API IMyInterface_Second
{
    GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    void SecondFunc() const;
};

UCLASS(Blueprintable,meta=(ProhibitedInterfaces="UMyInterface_Second"))
class INSIDER_API UMyClass_ProhibitedInterfaces :public UObject
{
    GENERATED_BODY()
public:
};

```

测试结果：

发现UMyInterface_Second被阻止实现了，但是UMyInterface_First依然可以被实现



原理代码：

可以看到在构造列表的时候，进行了过滤筛选。同时发现了.RightChop(1);的使用，因此填的接口名称，要加上U的前缀。如果UMyInterface_Second

```

TSharedRef<SWidget>
FBlueprintEditorUtils::ConstructBlueprintInterfaceClassPicker( const TArray<
UBlueprint*> &Blueprints, const FOnClassPicked& OnPicked)
{
//...

    UClass const* const ParentClass = Blueprint->ParentClass;
    // see if the parent class has any prohibited interfaces
    if ((ParentClass != nullptr) && ParentClass-
>HasMetaData(FBlueprintMetadata::MD_ProhibitedInterfaces))
    {
        FString const& ProhibitedList = Blueprint->ParentClass-
>GetMetaData(FBlueprintMetadata::MD_ProhibitedInterfaces);

        TArray< FString> ProhibitedInterfaceNames;
        ProhibitedList.ParseIntoArray(ProhibitedInterfaceNames, TEXT(","),
true);

        // Loop over all the prohibited interfaces
        for (int32 ExclusionIndex = 0; ExclusionIndex <
ProhibitedInterfaceNames.Num(); ++ExclusionIndex)
        {
            ProhibitedInterfaceNames[ExclusionIndex].TrimStartInline();
            FString const& ProhibitedInterfaceName =
ProhibitedInterfaceNames[ExclusionIndex].RightChop(1);
            UClass* ProhibitedInterface = UClass::TryFindTypeSlow< UClass>
(ProhibitedInterfaceName);
            if(ProhibitedInterface)
            {
                Filter->DisallowedClasses.Add(ProhibitedInterface);
                Filter->DisallowedChildrenOfClasses.Add(ProhibitedInterface);
            }
        }
    }

    // Do not allow adding interfaces that are already added to the Blueprint
    TArray< UClass*> InterfaceClasses;
    FindImplementedInterfaces(Blueprint, true, InterfaceClasses);
    for(UClass* InterfaceClass : InterfaceClasses)
    {
        Filter->DisallowedClasses.Add(InterfaceClass);
    }

    // Include a class viewer filter for imported namespaces if the class
    // picker is being hosted in an editor context
    TSharedPtr< IToolkit > AssetEditor =
FToolkitManager::Get().FindEditorForAsset(Blueprint);
    if (AssetEditor.IsValid() && AssetEditor->IsBlueprintEditor())
    {
        TSharedPtr< IBlueprintEditor > BlueprintEditor =
StaticCastSharedPtr< IBlueprintEditor >(AssetEditor);
        TSharedPtr< IClassViewerFilter > ImportedClassViewerFilter =
BlueprintEditor->GetImportedClassViewerFilter();
        if (ImportedClassViewerFilter.IsValid())
        {

```

```

        Options.ClassFilters.AddUnique(ImportedClassViewerFilter.ToSharedRef());
    }
}

// never allow parenting to children of itself
for (UClass* BPClass : BlueprintClasses)
{
    Filter->DisallowedChildrenOfClasses.Add(BPClass);
}

return FModuleManager::LoadModuleChecked<FCClassViewerModule>
("ClassViewer").CreateClassViewer(Options, OnPicked);
}

```

RestrictedToClasses

- 功能描述:** 限制蓝图函数库下的函数只能在RestrictedToClasses指定的类蓝图中右键创建出来
- 使用位置:** UClass
- 引擎模块:** Blueprint
- 元数据类型:** strings="a, b, c"
- 限制类型:** BlueprintFunctionLibrary
- 常用程度:** ★★★

在蓝图函数库上使用，指定该函数库中的函数只能用在RestrictedToClasses指定的类的蓝图中，不能在别的蓝图类中被右键出来。

测试代码：

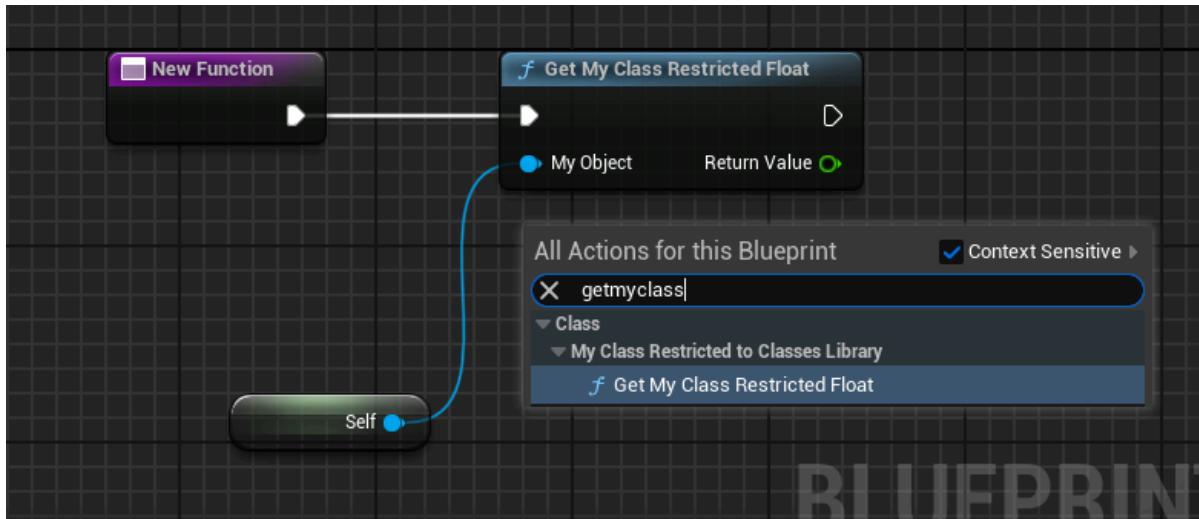
```

UCLASS(Blueprintable)
class INSIDER_API UMyClass_RestrictedToClasses : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
};

UCLASS(meta=(RestrictedToClasses="MyClass_RestrictedToClasses"))
class INSIDER_API UMyClass_RestrictedToClassesLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static float Get MyClassRestrictedFloat(UMyClass_RestrictedToClasses*
myObject) {return myObject->MyFloat;}
};

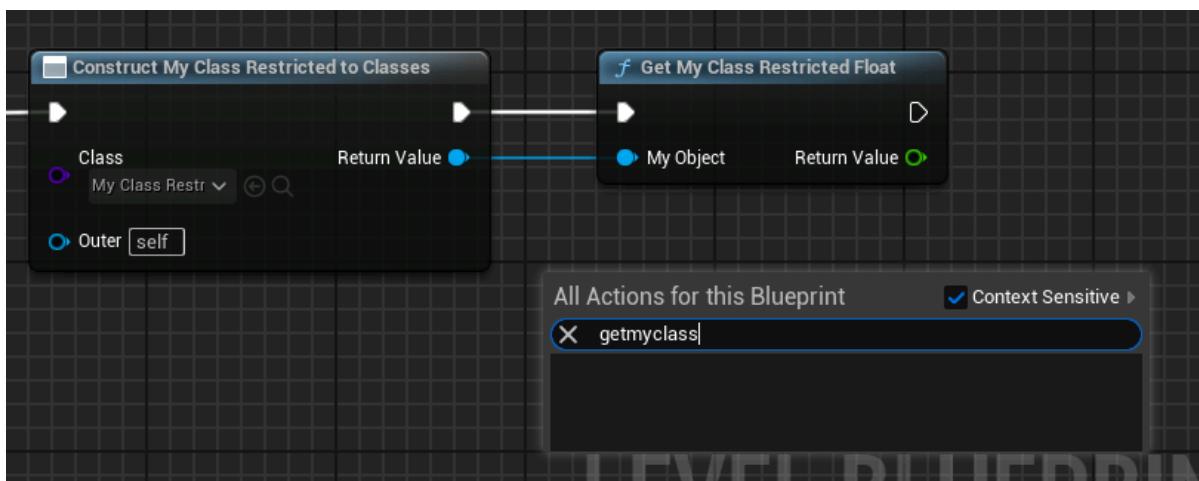
```

在UMyClass_RestrictedToClasses 的子类蓝图中测试效果：



在别的地方，比如关卡蓝图中测试效果：

因此右键创建不出来，但是直接粘贴节点其实还是可以调用的。



源码中的例子：

指定UBTFunctionLibrary中的节点，只能在BTNode中使用，否则没有意义。

```
UCLASS(meta=(RestrictedToClasses="BTNode"), MinimalAPI)
class UBTFunctionLibrary : public UBlueprintFunctionLibrary
{
    UFUNCTION(BlueprintPure, Category="AI|BehaviorTree", Meta=(HidePin="NodeOwner",
DefaultToSelf="NodeOwner"))
        static AIMODULE_API UBlackboardComponent* GetOwnersBlackboard(UBTNode*
NodeOwner);
    //...
}
```

原理：

```
static bool
BlueprintActionFilterImpl::IsRestrictedClassMember(FBlueprintActionFilter const&
Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool bIsFilteredOut = false;
    FBlueprintActionContext const& FilterContext = Filter.Context;
```

```

if (UClass const* ActionClass = BlueprintAction.GetOwnerClass())
{
    if (ActionClass->HasMetaData(FBlueprintMetadata::MD_RestrictedToClasses))
    {
        FString const& ClassRestrictions = ActionClass-
>GetMetaData(FBlueprintMetadata::MD_RestrictedToClasses);

        // Parse the the metadata into an array that is delimited by ',' and
        trim whitespace
        TArray<FString> ParsedClassRestrictions;
        ClassRestrictions.ParseIntoArray(ParsedClassRestrictions, TEXT(","));
        for (FString& ValidClassName : ParsedClassRestrictions)
        {
            ValidClassName = ValidClassName.TrimStartAndEnd();
        }

        for (UBlueprint const* TargetContext : FilterContext.Blueprints)
        {
            UClass* TargetClass = TargetContext->GeneratedClass;
            if (!TargetClass)
            {
                // skip possible null classes (e.g. macros, etc)
                continue;
            }

            bool bIsClassListed = false;

            UClass const* QueryClass = TargetClass;
            // walk the class inheritance chain to see if this class is one
            // of the allowed
            while (!bIsClassListed && (QueryClass != nullptr))
            {
                FString const ClassName = QueryClass->GetName();
                // If this class is on the list of valid classes
                for (const FString& ValidClassName : ParsedClassRestrictions)
                {
                    bIsClassListed = (ClassName == ValidClassName);
                    if (bIsClassListed)
                    {
                        break;
                    }
                }

                QueryClass = QueryClass->GetSuperclass();
            }

            // if the blueprint class wasn't listed as one of the few
            // classes that this can be accessed from, then filter it out
            if (!bIsClassListed)
            {
                bIsFilteredOut = true;
                break;
            }
        }
    }
}

```

```
    return bIsFilteredout;
}
```

ReturnDisplayName

- **功能描述:** 改变函数返回值的名字， 默认是ReturnValue
- **使用位置:** UFUNCTION
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **常用程度:** ★★★★☆

函数的返回值引脚名字默认是ReturnValue，如果想自己提供一个更有意义的名字，则可以用ReturnDisplayName来自定义一个名字。

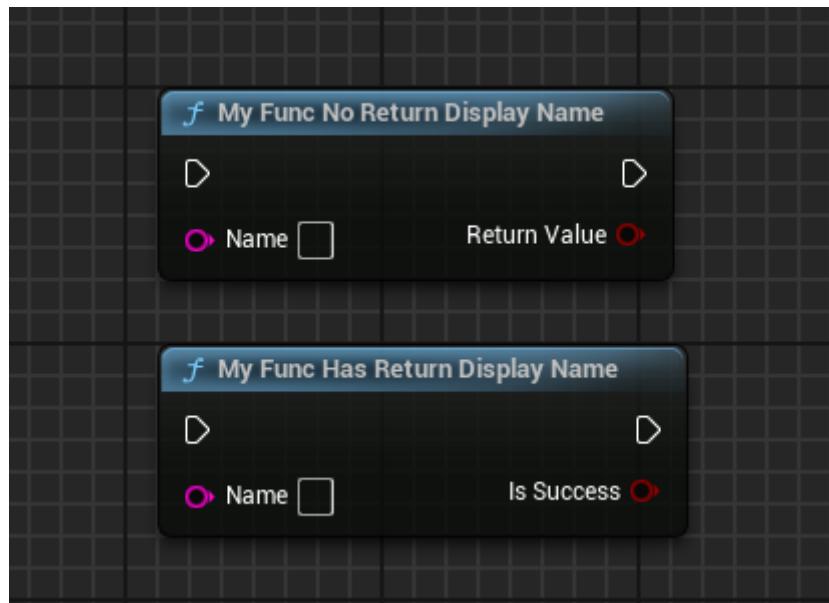
测试代码：

```
UFUNCTION(BlueprintCallable, meta = (ReturnDisplayName = "IsSuccess"))
static bool MyFunc_HasReturnDisplayName(FString Name) { return true; }

UFUNCTION(BlueprintCallable, meta = ())
static bool MyFunc_NoReturnDisplayName(FString Name) { return true; }
```

蓝图效果：

对比返回值的名字可以验证效果。



原理：

原理也很简单，在Pin上判断Meta并设置PinFriendlyName

```

if (Function->GetReturnProperty() == Param && Function-
>HasMetaData(FBlueprintMetadata::MD_ReturnDisplayName))
{
    Pin->PinFriendlyName = Function-
>GetMetaDataText(FBlueprintMetadata::MD_ReturnDisplayName);
}

```

SetParam

- 功能描述:** 指定一个函数为使用Set的函数，元素类型为通配符的泛型。
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** string="A | B | C"
- 限制类型:** TSet
- 常用程度:** ★★★

源码在UBlueprintSetLibrary。

SetParam支持多个Set和元素参数，以'/'分隔开，然后Pin的引脚可以通过'/'继续分隔，形成"SetA | ItemA, SetB | ItemB"的多组数据。

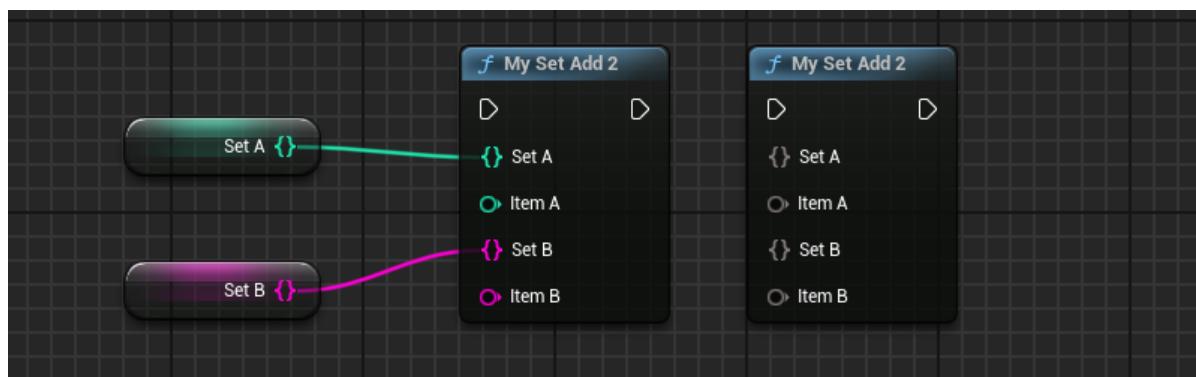
测试代码：

```

UFUNCTION(BlueprintCallable, CustomThunk, meta = (SetParam =
"SetA|ItemA,SetB|ItemB"))
    static void MySet_Add2(const TSet<int32>& SetA, const int32& ItemA, const
TSet<int32>& SetB, const int32& ItemB);
    static void GenericMySet_Add2(const void* TargetSet, const FSetPropertyParams*
SetA, const void* ItemA, const FSetPropertyParams* SetB, const void* ItemB);
DECLARE_FUNCTION(execMySet_Add2);

```

蓝图里效果：



原理：

用'/'分隔的是组，组内用'|'分隔的参数Pin都是同一种类型的。

```

void UK2Node_CallFunction::ConformContainerPins()
{
    // find any pins marked as SetParam
}

```

```

        const FString& SetPinMetaData = TargetFunction-
>GetMetaData(FBlueprintMetadata::MD_SetParam);

        // useless copies/allocates in this code, could be an optimization
target...
        TArray<FString> SetParamPinGroups;
        {
            SetPinMetaData.ParseIntoArray(SetParamPinGroups, TEXT(","),
true);
        }

        for (FString& Entry : SetParamPinGroups)
        {
            // split the group:
            TArray<FString> GroupEntries;
            Entry.ParseIntoArray(GroupEntries, TEXT(" | "),
true);
            // resolve pins
            TArray<UEdGraphPin*> ResolvedPins;
            for(UEdGraphPin* Pin : Pins)
            {
                if (GroupEntries.Contains(Pin->GetName()))
                {
                    ResolvedPins.Add(Pin);
                }
            }

            // if nothing is connected (or non-default), reset to wildcard
            // else, find the first type and propagate to everyone else::
            bool bReadyToPropagatSetType = false;
            FEdGraphTerminalType TypeToPropagate;
            for (UEdGraphPin* Pin : ResolvedPins)
            {
                TryReadTypeToPropagate(Pin, bReadyToPropagatSetType,
TypeToPropagate);
                if(bReadyToPropagatSetType)
                {
                    break;
                }
            }

            for (UEdGraphPin* Pin : ResolvedPins)
            {
                TryPropagateType( Pin, TypeToPropagate, bReadyToPropagatSetType
);
            }
        }
    }
}

```

ShowWorldContextPin

- 功能描述:** 放在UCLASS上，指定本类里的函数调用都必须显示WorldContext引脚，无论其本来是否默认隐藏
- 使用位置:** UCLASS
- 元数据类型:** bool

- **关联项:** WorldContext

放在UCLASS上，指定本类里的函数调用都必须显示WorldContext引脚，无论其本来是否默认隐藏，因为本Object类无法被当作WorldContextObject，即使实现了GetWorld()也要当作无法自动获得以此来让用户必须手动指定WorldContextObject。

一般放在UObject上，但在源码里发现在AGameplayCueNotify_Actor， AEeditorUtilityActor上也有。AEeditorUtilityActor是因为本身不会在Runtime里运行，因此没有World。AGameplayCueNotify_Actor有可能在CDO上被使用和Recycle，因此可也不能假定必须有WorldContext。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_WorldContextTest : public
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable)
    static FString MyFunc_NoWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

    UFUNCTION(BlueprintCallable, meta = (WorldContext = "WorldContextObject"))
    static FString MyFunc_HasWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);
};

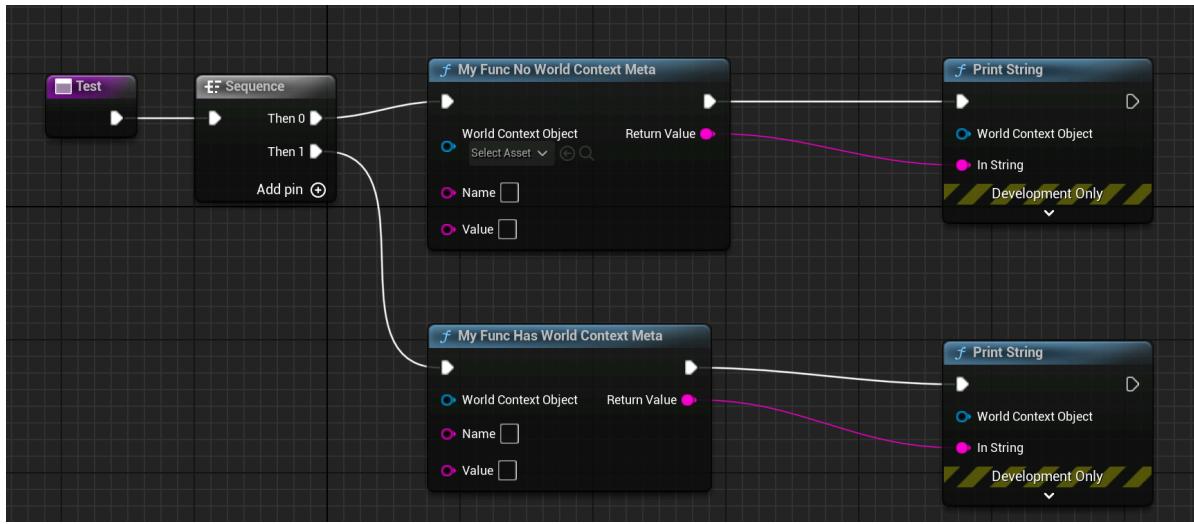
UCLASS(Blueprintable, BlueprintType, meta = (ShowWorldContextPin = "true"))
class INSIDER_API UMyObject_ShowWorldContextPin : public UObject
{
    GENERATED_BODY()
    UWorld* WorldPrivate = nullptr;

public:
    UFUNCTION(BlueprintCallable)
    void RegisterwithOuter()
    {
        if (UObject* outer = GetOuter())
        {
            WorldPrivate = outer->GetWorld();
        }
    }

    virtual UWorld* GetWorld() const override final { return WorldPrivate; }
};
```

蓝图测试效果：

可以见到虽然UMyObject_ShowWorldContextPin类实现了GetWorld()方法，但是即使是MyFunc_HasWorldContextMeta，WorldContextObject本来应该被自动赋值且隐藏的，但是在本类里也显式显示了出来。同时注意到PrintString也显示出了WorldContextObject。



原理：

在CallFunction的蓝图节点上，如果有bShowWorldContextPin，则不隐藏WorldContextMetaValue或DefaultToSelfMetaValue指定的函数参数。

```

bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    const bool bshowworldContextPin = ((PinsToHide.Num() > 0) && BP->BP->ParentClass && BP->ParentClass->HasMetaDataHierarchical(FBlueprintMetadata::MD_ShowworldContextPin));
    //...
    if (PinsToHide.Contains(Pin->PinName))
    {
        const FString PinNameStr = Pin->PinName.ToString();
        const FString& DefaultToSelfMetaValue = Function->GetMetaData(FBlueprintMetadata::MD_DefaultToSelf);
        const FString& WorldContextMetaValue = Function->GetMetaData(FBlueprintMetadata::MD_WorldContext);
        bool bIsSelfPin = ((PinNameStr == DefaultToSelfMetaValue) || (PinNameStr == WorldContextMetaValue));

        if (!bshowworldContextPin || !bIsSelfPin)
        {
            Pin->bHidden = true;
            Pin->bNotConnectable = InternalPins.Contains(Pin->PinName);
        }
    }
}

```

SparseClassDataTypes

- **使用位置:** UCLASS
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **关联项:** GetByRef

UCLASS: SparseClassDataType

- 常用程度： ★★

UnsafeDuringActorConstruction

- **功能描述：** 标明该函数不能在Actor的构造函数里调用
- **使用位置：** UFUNCTION
- **引擎模块：** Blueprint
- **元数据类型：** bool
- **常用程度：** ★★

标明该函数不能在Actor的构造函数里调用。一般是涉及渲染及物理的函数，在Actor的构造期间不允许被调用。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Unsafe :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static void MySafeFunction();
    UFUNCTION(BlueprintCallable, meta=(UnsafeDuringActorConstruction = "true"))
    static void MyUnsafeFunction();
};
```

在蓝图里函数的细节面板上也可以设置该Meta: UnsafeDuringActorConstruction，和在C++里设置是一样的效果。

可以发现MyUnsafeFunction函数不能在Actor构造函数里被调用出来，而蓝图里自定义的函数加上UnsafeDuringActorConstruction 标志后也会生成相应的警告和编译错误信息。



原理：

在蓝图中有该系列的判断，一目了然。

```

bool UEdGraphSchema_K2::CanFunctionBeUsedInGraph(const UClass* InClass, const
UFunction* InFunction, const UEdGraph* InDestGraph, uint32
InAllowedFunctionTypes, bool bIsCalledForEach, FText* OutReason) const
{
    const bool bIsUnsafeForConstruction = InFunction-
>GetBoolMetaData(FBlueprintMetadata::MD_UnsafeForConstructionScripts);
    if (bIsUnsafeForConstruction && bIsConstructionScript)
    {
        if(OutReason != nullptr)
        {
            *OutReason = LOCTEXT("FunctionUnsafeForConstructionScript", "Function
cannot be used in a Construction Script.");
        }
    }

    return false;
}
}

```

Variadic

- 功能描述:** 指定该函数接受多个参数
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 关联项:**
UFUNCTION: Variadic
- 常用程度:** ★★★

WorldContext

- 功能描述:** 指定函数的一个参数自动接收WorldContext对象，以便确定当前运行所处的World
- 使用位置:** UFUNCTION
- 引擎模块:** Blueprint
- 元数据类型:** string="abc"
- 关联项:** CallableWithoutWorldContext, ShowWorldContextPin
- 常用程度:** ★★★★☆

指定函数的一个参数自动接收WorldContext对象，以便确定当前运行所处的World。函数是 BlueprintCallable或BlueprintPure都可以，静态函数或成员函数也都可以。一般情况下是用于函数库里的静态函数，典型的例子是UGameplayStatics中的众多static函数。

```

UFUNCTION(BlueprintPure, Category="Game", meta=
(WorldContext="WorldContextObject"))
static ENGINE_API class UGameInstance* GetGameInstance(const UObject* 
worldContextObject)
{
    UWorld* World = GEngine->GetWorldFromContextObject(worldContextObject,
EGWorldErrorMode::LogAndReturnNull);
    return World ? World->GetGameInstance() : nullptr;
}

//在Runtime下获得World的方式一般是:
UWorld* World = GEngine->GetWorldFromContextObject(worldContextObject,
EGWorldErrorMode::ReturnNull);
//在Editor下(如CallInEditor函数)获得World的方式一般是:
UObject* worldContextObject = EditorEngine->GetEditorWorldContext().World();

```

测试代码:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_WorldContextTest :public 
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable)
    static FString MyFunc_NoWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

    UFUNCTION(BlueprintCallable, meta = (WorldContext = "WorldContextObject"))
    static FString MyFunc_HasWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

    UFUNCTION(BlueprintPure)
    static FString MyPure_NoWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

    UFUNCTION(BlueprintPure, meta = (WorldContext = "WorldContextObject"))
    static FString MyPure_HasWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyObject_NoGetWorld :public UObject
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyObject_HasGetWorld :public UObject
{
    GENERATED_BODY()

    UWorld* WorldPrivate = nullptr;
};

```

```

public:
    UFUNCTION(BlueprintCallable)
    void Registerwithouter()
    {
        if (UObject* outer = GetOuter())
        {
            WorldPrivate = outer->GetWorld();
        }
    }

    virtual UWorld* GetWorld() const override final { return WorldPrivate; }
};

//.cpp

FString UMyFunctionLibrary_WorldContextTest::MyFunc_HasWorldContextMeta(const UObject* WorldContextObject, FString name, FString value)
{
    UWorld* World = GEngine->GetWorldFromContextObject(WorldContextObject,
    EGetWorldErrorMode::LogAndReturnNull);
    if (World != nullptr)
    {
        return WorldContextObject->GetName();
    }
    return TEXT("None");
}

FString UMyFunctionLibrary_WorldContextTest::MyFunc_NoWorldContextMeta(const UObject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

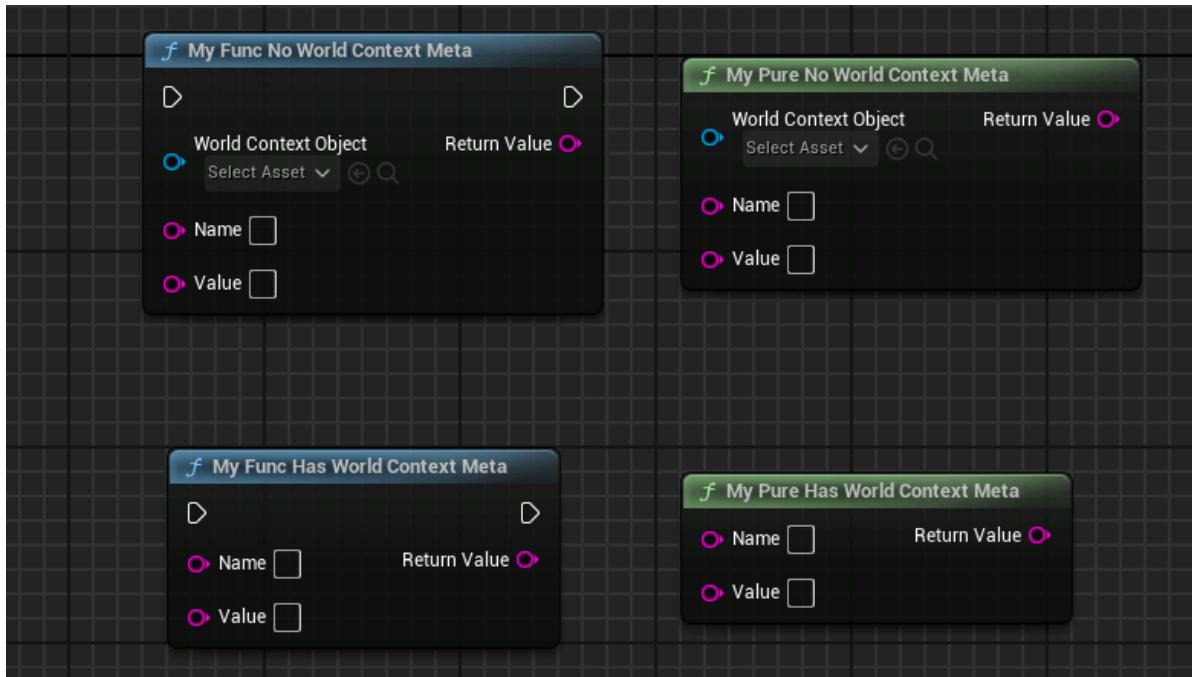
FString UMyFunctionLibrary_WorldContextTest::MyPure_NoWorldContextMeta(const UObject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

FString UMyFunctionLibrary_WorldContextTest::MyPure_HasWorldContextMeta(const UObject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

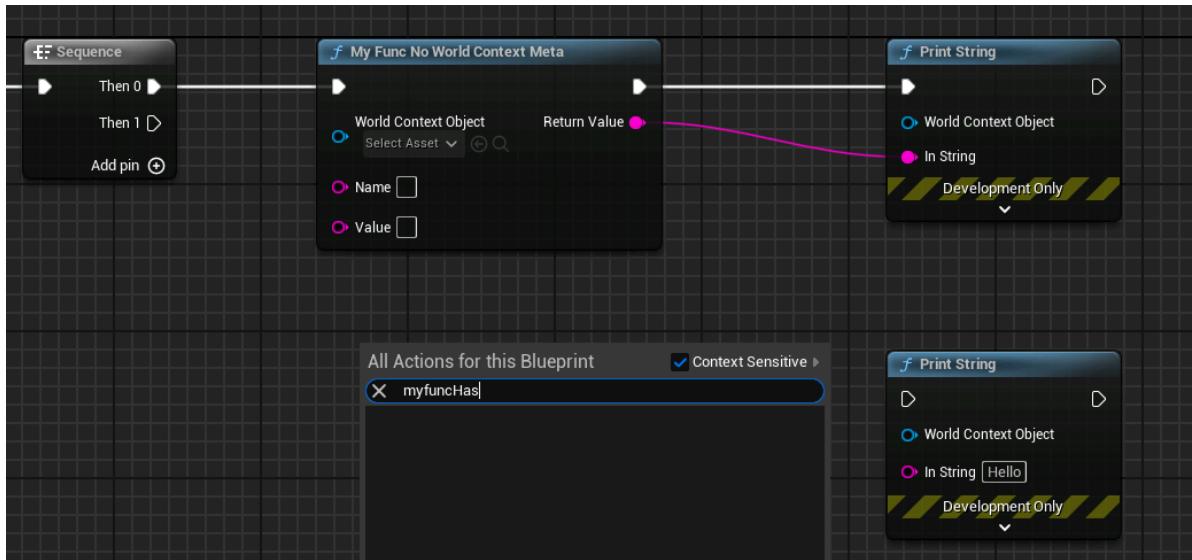
```

蓝图中的测试效果：

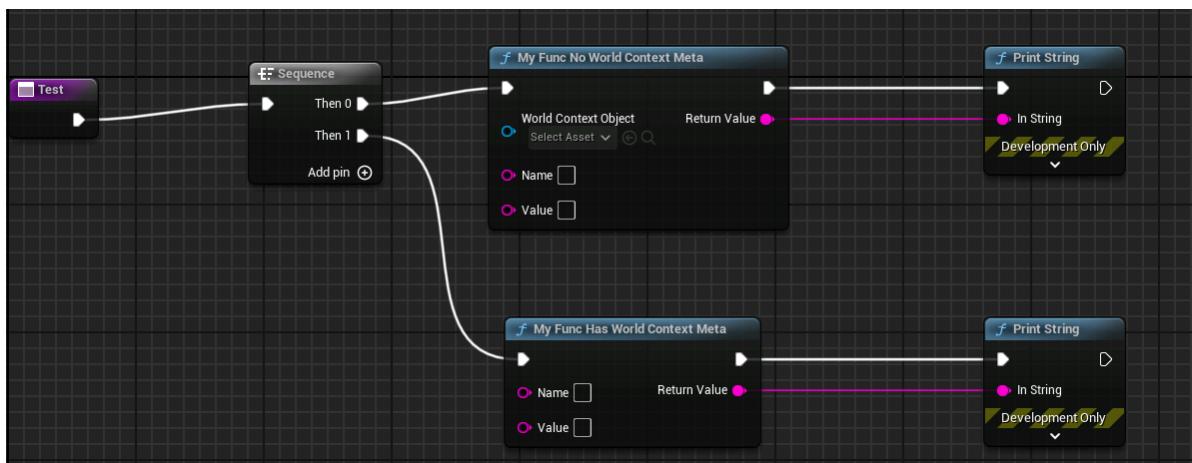
在Actor中调用，可以发现没指定WorldContext 的函数，会暴露出这个Object参数，让你必须手动指定。而带上WorldContext 的函数，则默认隐藏了起来WorldContextObject参数，因为WorldContextObject对象在Actor中可以自动被赋值（其值就是当前Actor）。



在UMyObject_NoGetWorld的子类里，因为并没有实现GetWorld，因此无法获得World，从而没办法自动赋值WorldContextObject，所以并不能调用出MyFunc_HasWorldContextMeta。



而在UMyObject_HasGetWorld的子类中调用，因为UMyObject_HasGetWorld实现了GetWorld，因此就可以允许调用MyFunc_HasWorldContextMeta，其WorldContextObject的值为UMyObject_HasGetWorld子类对象，在其身上会调用GetWorld()，从而获得之前注册进去的WorldPrivate对象。



原理：

当一个函数需要和World产生交互，而这个函数（通常是static函数）又无法直接寻找到World对象的时候，需要在函数的参数上手动从外部传入一个额外参数，根据这个参数来顺藤摸瓜寻到OuterWorld。这个参数就叫做WorldContextObject，一般是UObject*类型，方便传入各种类型的对象。

这个WorldContextObject你可以手动传入。也可以自动被赋值，只要这个Object类实现了virutal GetWorld()接口并且不返回nullptr，就可以正常的获得World对象，从而可以同runtime游戏世界产生交互。

在平常的使用过程中，我们的蓝图对象大部分已经是知道自身处于哪个World的，比如Actor肯定就知道自己属于哪个World。

```
Uworld* AActor::GetWorld() const
{
    // CDO objects do not belong to a world
    // If the actors outer is destroyed or unreachable we are shutting down and
    // the world should be nullptr
    if (!HasAnyFlags(RF_ClassDefaultObject) && ensureMsgf(GetOuter(),
        TEXT("Actor: %s has a null OuterPrivate in AActor::GetWorld()"), *GetFullName())
        && !GetOuter()->HasAnyFlags(RF_BeginDestroyed) && !GetOuter()-
    >IsUnreachable())
    {
        if (ULevel* Level = GetLevel())
        {
            return Level->OwningWorld;
        }
    }
    return nullptr;
}
```

在这种Actor内部调用static函数，如果每次还得手动设置WorldContextObject就显得麻烦又有点明知故问了。因此WorldContext这个meta就指定蓝图系统自动为我们的WorldContextObject赋值，其值就是该Actor本身，省去了我们手动传参的麻烦，又隐藏了这个与业务逻辑无关的功能胶水参数，看起来优雅一些。

而如果是在普通的Object对象内部调用函数，则不知道其所属于哪个World。这个时候就需要这个Object类实现了GetWorld()。编辑器下用bGetWorldOverridden这个变量来判断一个UObject子类是否已经覆盖了GetWorld。如果子类有覆盖，在探测的时候，只要在CDO上调用一下ImplementsGetWorld就可以获得bGetWorldOverridden==true的结果，结果就可以允许自动指定WorldContextObject的函数版本被调用。稍微多提一下，bGetWorldOverridden 不是UObject成员变量，其只是在 ImplementsGetWorld()调用时用到的临时变量，因此不需要保存。

```
#if DO_CHECK || WITH_EDITOR
// Used to check to see if a derived class actually implemented GetWorld() or not
thread_local bool bGetWorldOverridden = false;
#endif // #if DO_CHECK || WITH_EDITOR

class Uworld* UObject::GetWorld() const
{
    if (UObject* Outer = GetOuter())
    {
        return Outer->GetWorld();
    }
}
```

```

#if DO_CHECK || WITH_EDITOR
    bGetworldoverridden = false;
#endif
    return nullptr;
}

#if WITH_EDITOR

bool Uobject::ImplementsGetWorld() const
{
    bGetworldoverridden = true;
    Getworld();
    return bGetworldoverridden;
}

#endif // #if WITH_EDITOR

```

更进一步解释，在静态蓝图函数上（BPTYPE_FunctionLibrary里的函数，或者为FUNC_Static的函数）都有一个隐藏的参数“**WorldContext**”，在**UK2Node_CallFunction**的**ExpandNode**的时候，会把WorldContext上的值，赋值给DefaultToSelf或WorldContext则两个Meta标签指定的函数参数，从而实现自动把Self赋值到DefaultToSelf和WorldContext。

```

void UK2Node_CallFunction::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)

else if (UEdGraphPin* BetterSelfPin = EntryPoints[0]->GetAutoWorldContextPin())
{
    const FString& DefaultToSelfMetaValue = Function-
>GetMetaData(FBlueprintMetadata::MD_DefaultToSelf);
    const FString& WorldContextMetaValue = Function-
>GetMetaData(FBlueprintMetadata::MD_WorldContext);

    struct FStructConnectHelper
    {
        static void Connect(const FString& PinName, UK2Node* Node, UEdGraphPin* BetterSelf, const UEdGraphSchema_K2* InSchema, FCompilerResultsLog& MessageLog)
        {
            UEdGraphPin* Pin = Node->FindPin(PinName);
            if (!PinName.IsEmpty() && Pin && !Pin->LinkedTo.Num())
            {
                const bool bConnected = InSchema->TryCreateConnection(Pin,
BetterSelf);
                if (!bConnected)
                {
                    MessageLog.Warning(*LOCTEXT("DefaultToSelfNotConnected",
"DefaultToSelf pin @@ from node @@ cannot be connected to @@").ToString(), Pin,
Node, BetterSelf);
                }
            }
        };
        FStructConnectHelper::Connect(DefaultToSelfMetaValue, this, BetterSelfPin,
Schema, CompilerContext.MessageLog);
    };
}

```

```

if (!Function->HasMetaData(FBlueprintMetadata::MD_CallableWithoutWorldContext))
{
    FStructConnectHelper::Connect(WorldContextMetaValue, this, BetterSelfPin,
Schema, CompilerContext.MessageLog);
}
}

```

AllowAnyActor

- 功能描述:** 用在ComponentReference属性上，在UseComponentPicker的情况下使得组件选取器扩大到场景里其他Actor下的其他组件。
- 使用位置:** UPROPERTY
- 引擎模块:** Component Property
- 元数据类型:** bool
- 限制类型:** FComponentReference, FSoftComponentReference
- 关联项:** UseComponentPicker
- 常用程度:** ★★

用在ComponentReference属性上，在UseComponentPicker的情况下使得组件选取器扩大到场景里其他Actor下的其他组件。

- 也要注意到，这个AllowAnyActor影响的只是UI上的组件选择。一个ComponentReference即使不加AllowAnyActor，也可以通过ReferencedActor引用到别的Actor，然后手填其属下的组件名字。然后可以正常的在C++里GetComponent里出来正确的组件对象。因此AllowAnyActor跟逻辑无关。

测试代码和效果见UseComponentPicker。

原理：

主要是FComponentReferenceCustomization。根据源码查看，bAllowAnyActor 只在已经有 bUseComponentPicker的情况下生效，且用来对Actor列表进行过滤。

```

void
FComponentReferenceCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
CustomizationUtils)
{
    PropertyHandle = InPropertyHandle;

    CachedComponent.Reset();
    CachedFirstOuterActor.Reset();
    CachedPropertyAccess = FPropertyAccess::Fail;

    bAllowClear = false;
    bAllowAnyActor = false;
    bUseComponentPicker = PropertyHandle->HasMetaData(NAME_UseComponentPicker);
    bIsSoftReference = false;

    if (bUseComponentPicker)
    {

```

```

        FProperty* Property = InPropertyParams->GetProperty();
        check(CastField<FStructPropertyParams>(Property) &&
              (FComponentReference::StaticStruct() == CastFieldChecked<const
FStructPropertyParams>(Property)->Struct ||
               FSoftComponentReference::StaticStruct() == CastFieldChecked<const
FStructPropertyParams>(Property)->Struct));

        bAllowClear = !(InPropertyParams->GetMetaDataProperty()->PropertyFlags &
CPF_NoClear);
        bAllowAnyActor = InPropertyParams->HasMetaData(NAME_AllowAnyActor);
        bIsSoftReference = FSoftComponentReference::StaticStruct() ==
CastFieldChecked<const FStructPropertyParams>(Property)->Struct;

        BuildClassFilters();
        BuildComboBox();

        InPropertyParams-
>SetOnPropertyValueChanged(FSimpleDelegate::CreateSP(this,
&FComponentReferenceCustomization::OnPropertyValueChanged));

        // set cached values
        {
            CachedComponent.Reset();
            CachedFirstOuterActor = GetFirstOuterActor();

            FComponentReference TmpComponentReference;
            CachedPropertyAccess = GetValue(TmpComponentReference);
            if (CachedPropertyAccess == FPropertyAccess::Success)
            {
                CachedComponent =
TmpComponentReference.GetComponent(CachedFirstOuterActor.Get());
                if (!IsComponentReferenceValid(TmpComponentReference))
                {
                    CachedComponent.Reset();
                }
            }
        }

        HeaderRow.NameContent()
        [
            InPropertyParams->CreatePropertyNameWidget()
        ]
        .valueContent()
        [
            ComponentComboBox.ToSharedRef()
        ]
        .IsEnabled(MakeAttributesSP(this,
&FComponentReferenceCustomization::CanEdit));
    }
    else
    {
        HeaderRow.NameContent()
        [
            InPropertyParams->CreatePropertyNameWidget()
        ]
        .valueContent()
    }
}

```

```

        [
            InPropertyParams->CreatePropertyValueWidget()
        ]
        .IsEnabled(MakeAttributesP(this,
&FComponentReferenceCustomization::CanEdit));
    }
}

bool FComponentReferenceCustomization::IsFilteredActor(const AActor* const Actor)
const
{
    return bAllowAnyActor || Actor == CachedFirstOuterActor.Get();
}

```

BlueprintSpawnableComponent

- 功能描述:** 允许该组件出现在Actor蓝图里Add组件的面板里。
- 使用位置:** UCLASS
- 引擎模块:** Component Property
- 元数据类型:** bool
- 限制类型:** Component类
- 常用程度:** ★★★★

允许该组件出现在Actor蓝图里Add组件的面板里。

在蓝图节点上，不管有没有BlueprintSpawnableComponent则都是可以添加该组件的。

测试代码：

```

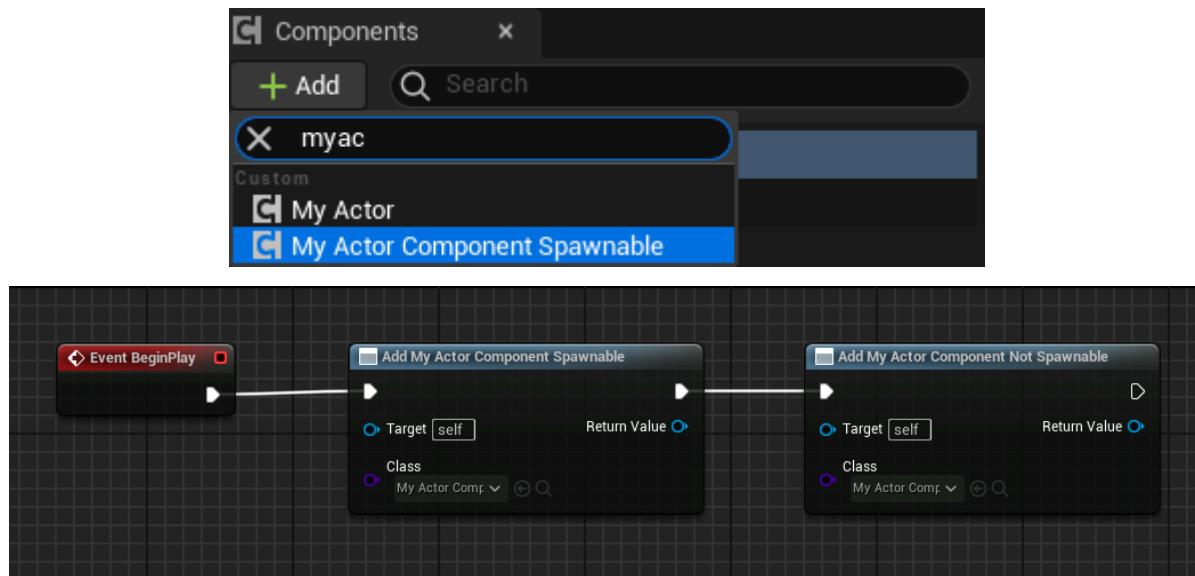
UCLASS(Blueprintable, meta = (BlueprintSpawnableComponent))
class INSIDER_API UMyActorComponent_Spawnable : public UActorComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
};

UCLASS(Blueprintable)
class INSIDER_API UMyActorComponent_NotSpawnable : public UActorComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
};

```

蓝图中效果：

可以看到，在Actor的左边Add的按钮下，UMyActorComponent_Spawnable 可以被添加进去，但是 UMyActorComponent_NotSpawnable 被阻止了。但同时也要注意到如果在蓝图中AddComponent节点则是都可以的。



原理：

```
bool FKismetEditorUtilities::IsClassABlueprintSpawnableComponent(const uclass* class)
{
    // @fixme: Cooked packages don't have any metadata (yet; they might become available via the sidecar editor data)
    // However, all uncooked BPs that derive from ActorComponent have the BlueprintSpawnableComponent metadata set on them
    // (see FBlueprintEditorUtils::RecreateClassMetaData), so include any ActorComponent BP that comes from a cooked package
    return (!class->HasAnyClassFlags(CLASS_Abstract) &&
            class->IsChildof<UActorComponent>() &&
            (class->HasMetaData(FBlueprintMetadata::MD_BlueprintSpawnableComponent) || class->GetPackage()->bIsCookedForEditor));
}
```

UseComponentPicker

- 功能描述：**用在ComponentReference属性上，使得选取器的列表里展示出Actor属下的Component以便选择。
- 使用位置：**UPROPERTY
- 引擎模块：**Component Property
- 元数据类型：**bool
- 限制类型：**FComponentReference, FSoftComponentReference
- 关联项：**AllowAnyActor
- 常用程度：**★★

用在ComponentReference属性上，使得选取器的列表里展示出Actor属下的Component以便选择。

- 默认情况下，FComponentReference的Referenced Actor属性展开的选择器列表是让你选择场景里的Actor，因此并不会把该Actor下的组件也都显示出来。而ComponentReference下的ComponentName属性需要玩家手动填写。这种方式比较原始，也容易出错。
- 因此加上UseComponentPicker后，就可以显示出组件列表来选择。但是又默认限制是当前Actor属下的所有组件，不包括场景里其他Actor里的组件。
- 如果想要进一步把场景里所有Actor下的所有组件都列出来，则需要进一步加上AllowAnyActor，以扩大筛选范围。
- ComponentReference的属性类型有两种，FComponentReference和FSoftComponentReference，二者都对应了FComponentReferenceCustomization。测试代码为简洁就没有列出FSoftComponentReference。

测试代码：

```
UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category = "UseComponentPickerTest")
FComponentReference MyComponentReference_NoUseComponentPicker;

UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category = "UseComponentPickerTest", meta = (UseComponentPicker))
FComponentReference MyComponentReference_UseComponentPicker;

UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category = "UseComponentPicker_AllowAnyActor_Test", meta = (UseComponentPicker,AllowAnyActor))
FComponentReference MyComponentReference_UseComponentPicker_AllowAnyActor;
```

测试效果：

- 可见默认的第一个列出了所有Actor，但是ComponentName需要手写。
- 第二个加上UseComponentPicker后，列出了当前Actor下的所有组件，但是不能选择到其他Actor的组件。
- 第三个继续加上AllowAnyActor后，列出了所有Actor的所有组件。

 F:\UnrealSpecifiers\Doc\Meta\Component\UseComponentPicker\UseComponentPicker.jpg

原理：

FComponentReference和FSoftComponentReference，二者都对应了FComponentReferenceCustomization。看源码可发现用上bUseComponentPicker后，会专门创建ClassFilters和ComboBox，就是采用不同的类型过滤器和不同的UI来选择组件。否则else分支就是很朴素的结构属性展开编辑。

```
void FComponentReferenceCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle> InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils& CustomizationUtils)
{
    PropertyHandle = InPropertyHandle;
```

```

CachedComponent.Reset();
CachedFirstOuterActor.Reset();
CachedPropertyAccess = FPropertyAccess::Fail;

bAllowClear = false;
bAllowAnyActor = false;
bUseComponentPicker = PropertyHandle->HasMetaData(NAME_UseComponentPicker);
bIsSoftReference = false;

if (bUseComponentPicker)
{
    FProperty* Property = InPropertyParams->GetProperty();
    check(CastField<FStructPropertyParams>(Property) &&
        (FComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct ||
        FSoftComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct));

    bAllowClear = !(InPropertyParams->GetMetaDataProperty()->PropertyFlags &
CPF_NoClear);
    bAllowAnyActor = InPropertyParams->HasMetaData(NAME_AllowAnyActor);
    bIsSoftReference = FSoftComponentReference::StaticStruct() ==
CastFieldChecked<const FPropertyParams>(Property)->Struct;

    BuildClassFilters();
    BuildComboBox();
}

InPropertyParams-
>SetOnPropertyValuechanged(FSimpleDelegate::CreateSP(this,
&FComponentReferenceCustomization::OnPropertyValuechanged));

// set cached values
{
    CachedComponent.Reset();
    CachedFirstOuterActor = GetFirstOuterActor();

    FComponentReference TmpComponentReference;
    CachedPropertyAccess = GetValue(TmpComponentReference);
    if (CachedPropertyAccess == FPropertyAccess::Success)
    {
        CachedComponent =
TmpComponentReference.GetComponent(CachedFirstOuterActor.Get());
        if (!IsComponentReferenceValid(TmpComponentReference))
        {
            CachedComponent.Reset();
        }
    }
}

HeaderRow.NameContent()
[
    InPropertyParams->CreatePropertyNameWidget()
]
.valueContent()
[
    ComponentComboBox.ToSharedRef()
]

```

```

        ]
        .IsEnabled(MakeAttributesP(this,
&FComponentReferenceCustomization::CanEdit));
    }
    else
    {
        HeaderRow.NameContent()
        [
            InPropertyParams->CreatePropertyNameWidget()
        ]
        .ValueContent()
        [
            InPropertyParams->CreatePropertyValueWidget()
        ]
        .IsEnabled(MakeAttributesP(this,
&FComponentReferenceCustomization::CanEdit));
    }
}

```

ConfigHierarchyEditable

- **功能描述:** 使得一个属性可以在Config的各个层级配置。
- **使用位置:** UPROPERTY
- **引擎模块:** Config
- **元数据类型:** bool
- **常用程度:** ★★★

使得一个属性可以在Config的各个层级配置。

- 所谓Config的层级，指的是Base, ProjectDefault, EnginePlatform, ProjectPlatform这些逐级被更高优先级的覆盖。这部分知识大家可以参考网上其他的config详解文章。
- 一般的带有Config的属性，其配置值只存在于UCLASS上的config标识符指定的config文件中。但如果该属性加上ConfigHierarchyEditable这个标记，就允许它在各个层级都可以进行不同的配置。这种属性一般是有根据不同平台而要配置不同的值的需求，比如一些平台相关的性能参数等。

测试例子：

```

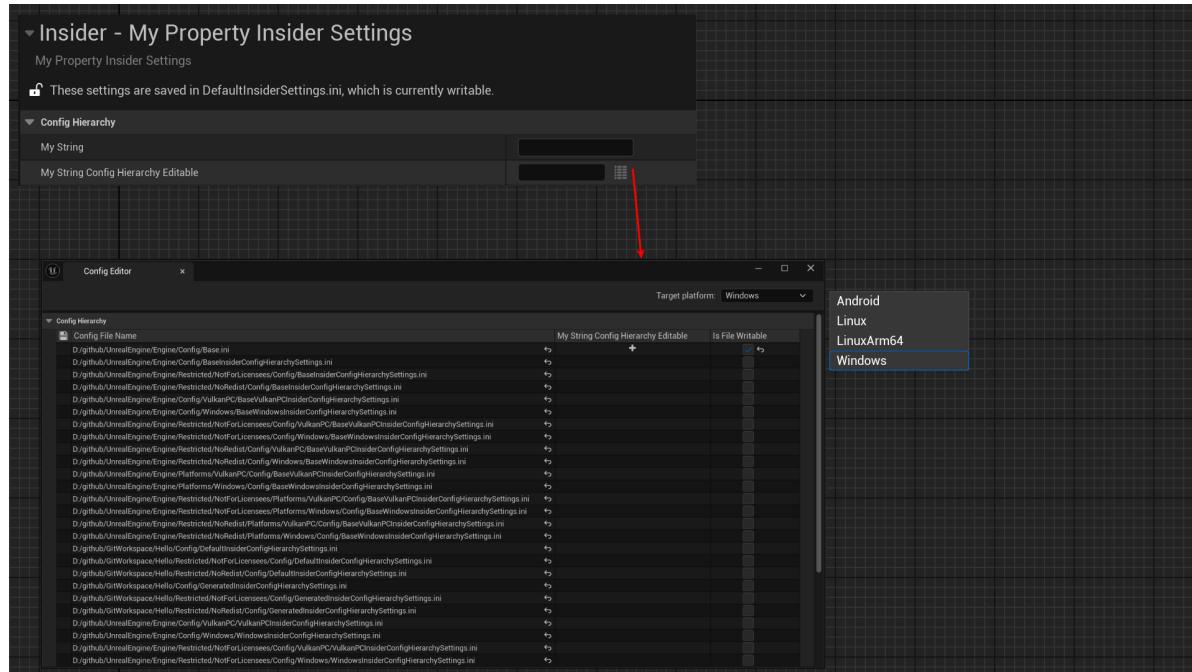
UCLASS(config = InsiderSettings, defaultconfig)
class UMyProperty_InsiderSettings :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigHierarchy)
    FString MyString;

    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigHierarchy, meta = (ConfigHierarchyEditable))
    FString MyString_ConfigHierarchyEditable;
};

```

测试效果：

可以见到MyString_ConfigHierarchyEditable输入框的右边出现了个层级按钮，可打开一个专门的ConfigEditor，方便你分别在不同的平台和不同的层级配置不同的值。



源码例子：

```
UCLASS(config = Game, defaultconfig)
class COMMONUI_API UCommonUISettings : public uobject
{
    /** The set of traits defined per-platform (e.g., the default input mode,
    whether or not you can exit the application, etc...) */
    UPROPERTY(config, EditAnywhere, Category = "Visibility", meta=
(Categories="Platform.Trait", ConfigHierarchyEditable))
        TArray<FGameplayTag> PlatformTraits;
}
```

原理：

逻辑很简单，就是在细节面板生成ValueWidget的时候，根据ConfigHierarchyEditable配置额外再生成一个层级配置按钮。

```
void FDetailPropertyRow::MakeValueWidget( FDetailWidgetRow& Row, const
TSharedPtr< FDetailWidgetRow> InCustomRow, bool bAddWidgetDecoration ) const
{
    // Don't add config hierarchy to container children, can't edit child
    properties at the hierarchy's per file level
    TSharedPtr< IPropertyHandle> ParentHandle = PropertyHandle->GetParentHandle();
    bool bIsChildProperty = ParentHandle && (ParentHandle->AsArray() ||
    ParentHandle->AsMap() || ParentHandle->AsSet());

    if (!bIsChildProperty && PropertyHandle-
>HasMetaData(TEXT("ConfigHierarchyEditable")))
    {
```

```

valueWidget->AddSlot()
    .AutoWidth()
    .VAlign(VAlign_Center)
    .HAlign(HAlign_Left)
    .Padding(4.0f, 0.0f, 4.0f, 0.0f)
    [
        PropertyCustomizationHelpers::MakeEditConfigHierarchyButton(FSimpleDelegate::CreateSP(PropertyEditor.ToSharedRef(), &FPropertyEditor::EditConfigHierarchy))
    ];
}
}

```

ConfigRestartRequired

- 功能描述:** 使属性在设置里改变后弹出重启编辑器的对话框。
- 使用位置:** UPROPERTY
- 引擎模块:** Config
- 元数据类型:** bool
- 常用程度:** ★★★

使属性在设置里改变后弹出重启编辑器的对话框。

自然的，一般是用于真的需要重启编辑器的设置。

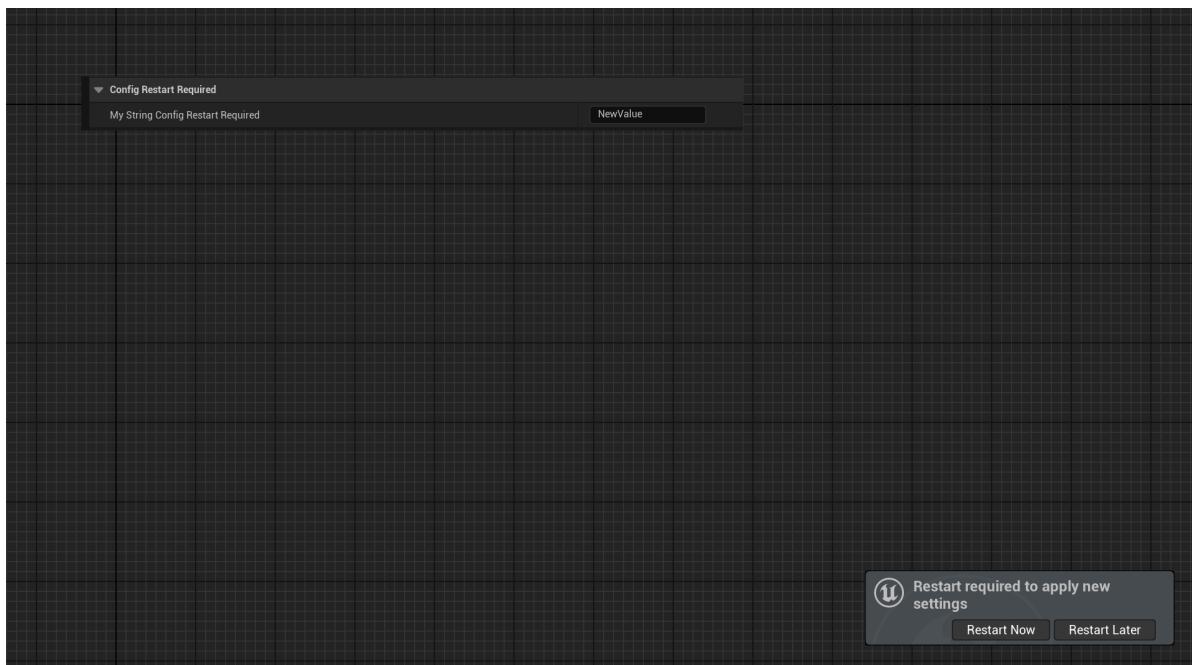
测试代码：

```

public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigRestartRequired, meta = (ConfigRestartRequired="true"))
    FString MyString_ConfigRestartRequired;

```

测试效果：



原理：

在SSettingsEditor生效，可见得是在UI窗口发生改变。然后弹出对话框。

```
void SSettingsEditor::NotifyPostChange( const FPropertyChangedEvent&
PropertyChangedEvent, class FEditPropertyChain* PropertyThatChanged )
{
    static const FName ConfigRestartRequiredKey = "ConfigRestartRequired";
    if (PropertyChangedEvent.Property->GetBoolMetaData(ConfigRestartRequiredKey) ||
PropertyChangedEvent.MemberProperty->GetBoolMetaData(ConfigRestartRequiredKey))
    {
        OnApplicationRestartRequiredDelegate.ExecuteIfBound();
    }
}
```

ConsoleVariable

- 功能描述：**把一个Config属性的值同步到一个同名的控制台变量。
- 使用位置：**UPROPERTY
- 引擎模块：**Config
- 元数据类型：**string="abc"
- 常用程度：**★★★★★

把一个Config属性的值同步到一个同名的控制台变量。

- Config值往往也确实需要在控制台（按~）中更改，这种需求挺常见的，因此就有了这个标记。典型的例子是源码中的URendererSettings有相匹配的一系列“r.”开头的控制变量。
- Config文件中的值也会变成这个ConsoleVariable的名字（一般形如 r.XXX.XX之类的格式），而不是属性名。
- 但单单加上这个标记是不够的，这个控制台变量是不会被自动创建出来的。因此需要自己再用代码创建，用类似TAutoConsoleVariable这种注册同名的控制台变量。
- 有了控制台变量之后，也需要专门的代码来对二者的值进行同步。见下述测试代码 ImportConsoleVariableValues和ExportValuesToConsoleVariables的调用。
- 另外要格外注意，ConsoleVariable的设置是有优先级的。Console的优先级比ProjectSettings高，因此如果在Console中改变后再尝试在ProjectSettings中更改值，就会报错。

测试代码：

```
UCLASS(config = InsiderSettings, defaultconfig)
class UMyProperty_InsiderSettings :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category = Console, meta
= (Consolevariable = "i.Insider.MyStringConsole"))
        FString MyString_Consolevariable;
public:
    virtual void PostInitProperties() override;
#endif WITH_EDITOR
```

```

    virtual void PostEditChangeProperty(FPropertyChangedEvent&
PropertyChangedEvent) override;
#endif
};

//.cpp
static TAutoConsolevariable<FString> CVarInsiderMyStringConsole(
TEXT("i.Insider.MyStringConsole"),
TEXT("Hello"),
TEXT("Insider test config to set MyString."));

void UMyProperty_InsiderSettings::PostInitProperties()
{
    Super::PostInitProperties();

#if WITH_EDITOR
    if (IsTemplate())
    {
        ImportConsolevariablevalues();
    }
#endif // #if WITH_EDITOR
}

#if WITH_EDITOR
void UMyProperty_InsiderSettings::PostEditChangeProperty(FPropertyChangedEvent&
PropertyChangedEvent)
{
    Super::PostEditChangeProperty(PropertyChangedEvent);

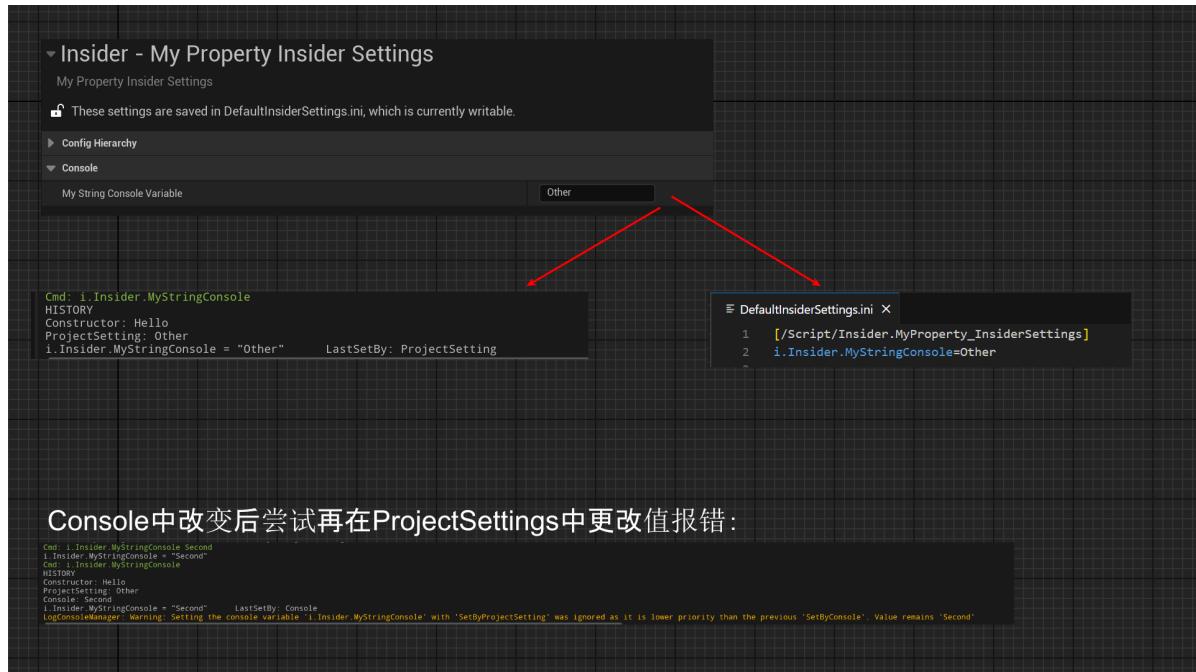
    if (PropertyChangedEvent.Property)
    {
        ExportValuesToConsolevariables(PropertyChangedEvent.Property);
    }
}
#endif // #if WITH_EDITOR

```

测试结果：

可见一开始的时候控制台和配置文件的值都是和ProjectSettings中的值同步。

如果在Console中改变后再尝试在ProjectSettings中更改值，就会报错。



原理：

具体的值同步逻辑可见以下两个函数就知道了，无非是根据名字去寻找相应的ConsoleVariable 然后get/set值。

```

void UDeveloperSettings::ImportConsoleVariableValues()
{}

void UDeveloperSettings::ExportValuesToConsoleVariables(FProperty*
PropertyThatChanged)
{}

```

EditorConfig

- 功能描述：**保存编辑器的配置
- 使用位置：** UCLASS
- 引擎模块：** Config
- 元数据类型：** string="abc"
- 关联项：**

UCLASS: EditorConfig

- 常用程度：** ★★★

ArraySizeEnum

- 功能描述：**为固定数组提供一个枚举，使得数组元素按照枚举值来作为索引和显示。
- 使用位置：** UPROPERTY
- 引擎模块：** Container Property
- 元数据类型：** string="abc"
- 限制类型：** T Array[Size]
- 常用程度：** ★★★

为固定数组提供一个枚举，使得数组元素按照枚举值来作为索引和显示。

- 所谓固定数组，是区别于TArray这种可以动态改变大小的数组，而是平凡的用[size]直接定义的数组。这种固定数组（static array）因为不会增删，因此才有时适合用枚举里的所有值用作下标，达成更高的便利性。
- 枚举里一般会把最后一个枚举项（一般叫做Max或者Size，count之类）作为数据大小值。
- 枚举里不想显示的枚举值可以用Hidden隐藏起来，但因为数组下标对应的是枚举项的下标（就是第几个枚举值）而不是枚举项的值，因此会发现数组的实际显示项目比定义的Size要小。

测试代码：

```
UENUM(BlueprintType)
enum class EMyArrayEnumNormal : uint8
{
    First,
    Second,
    Third,
    Max,
};

UENUM(BlueprintType)
enum class EMyArrayEnumHidden : uint8
{
    First,
    Second,
    Cat = 5 UMETA(Hidden),
    Third = 2,
    Max = 3,
};

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest)
int32 MyIntArray_NoArraySizeEnum[3];

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest, meta = (ArraySizeEnum =
"MyArrayEnumNormal"))
int32 MyIntArray_Normal_HasArraySizeEnum[(int)EMyArrayEnumNormal::Max];

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest, meta = (ArraySizeEnum =
"MyArrayEnumHidden"))
int32 MyIntArray_Hidden_HasArraySizeEnum[(int)EMyArrayEnumHidden::Max];
```

测试效果：

都是大小为3的固定数组。

- MyIntArray_NoArraySizeEnum，是最普通的数组模样。
- MyIntArray_Normal_HasArraySizeEnum，正统的使用枚举项来当数组下标的例子。可以发现下标名字不是012，而是枚举项名称了。
- MyIntArray_Hidden_HasArraySizeEnum采用的枚举项里有隐藏的一项Cat，但它的下标是2（因为定义的顺序），因此数组的第3个被隐藏了起来。

▼ Array Size Enum Test	
My Int Array No Array Size Enum	3 Array elements
Index [0]	0
Index [1]	0
Index [2]	0
My Int Array Normal Has Array Size Enum	3 Array elements
First	0
Second	0
Third	0
My Int Array Hidden Has Array Size Enum	2 Array elements
First	0
Second	0

原理：

可以发现一开始判断是否是固定数组 (ArrayDim>1其实就是固定数组了) , 然后通过名字找枚举, 以及为数组的每一项去枚举里找枚举项从而生成细节面板里的子行。

```
void FItemPropertyParams::InitChildNodes()
{
    if( MyPropertyParams->ArrayDim > 1 && ArrayIndex == -1 )
    {
        // Do not add array children which are defined by an enum but the
        // enum at the array index is hidden
        // This only applies to static arrays
        static const FName NAME_ArraySizeEnum("ArraySizeEnum");
        UEnum* ArraySizeEnum = NULL;
        if (MyPropertyParams->HasMetaData(NAME_ArraySizeEnum))
        {
            ArraySizeEnum = FindObject<UEnum>(NULL, *MyPropertyParams-
>GetMetaData(NAME_ArraySizeEnum));
        }

        // Expand array.
        for( int32 Index = 0 ; Index < MyPropertyParams->ArrayDim ; Index++ )
        {
            bool bShouldBeHidden = false;
            if( ArraySizeEnum )
            {
                // The enum at this array index is hidden
                bShouldBeHidden = ArraySizeEnum->HasMetaData(TEXT("Hidden"),
Index );
            }

            if( !bShouldBeHidden )
            {
                TSharedPtr<FItemPropertyParams> NewItemNode( new
FItemPropertyParams );
                FPropertyParamsInitParams InitParams;
```

```

        InitParams.ParentNode = SharedThis(this);
        InitParams.Property = MyProperty;
        InitParams.ArrayOffset = Index*MyProperty->ElementSize;
        InitParams.ArrayIndex = Index;
        InitParams.bAllowChildren = true;
        InitParams.bForceHiddenPropertyVisibility =
bShouldShowHiddenProperties;
        InitParams.bCreateDisableEditOnInstanceNodes =
bShouldShowDisableEditOnInstance;

        NewItemNode->InitNode( InitParams );
        AddChildNode(NewItemNode);
    }
}
}

}

```

EditFixedOrder

- 功能描述:** 使数组的元素无法通过拖拽来重新排序。
- 使用位置:** UPROPERTY
- 引擎模块:** Container Property
- 元数据类型:** bool
- 限制类型:** TArray
- 常用程度:** ★★

测试代码:

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TArray<int32> MyIntArray_NoEditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedOrder))
TArray<int32> MyIntArray_EditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TSet<int32> MyIntSet_NoEditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedOrder))
TSet<int32> MyIntSet_EditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TMap<int32,FString> MyIntMap_NoEditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedOrder))
TMap<int32,FString> MyIntMap_EditFixedOrder;

```

测试效果：

- 可见只有第一个MyIntArray_NoEditFixedOrder，在数组元素上出现可拖拽的标记，然后可以改变顺序。
- 加上EditFixedOrder的TArray就无法改变顺序了。
- 其他TSet, TMap是不支持该meta的，因为其内部本身顺序也无关。

 F:\UnrealSpecifiers\Doc\Meta\Container>EditFixedOrder>EditFixedOrder.gif

原理：

可以看见，细节面板里一个属性行是否可重排序的判断是外部是个数组，且没有EditFixedOrder和ArraySizeEnum（固定数组）。当然本身这个属性也要在可编辑状态（比如被禁用灰掉就显然不可编辑了）

```
bool FPropertyNode::IsReorderable()
{
    FProperty* NodeProperty = GetProperty();
    if (NodeProperty == nullptr)
    {
        return false;
    }
    // It is reorderable if the parent is an array and metadata doesn't prohibit it
    const FArrayProperty* OuterArrayProp = NodeProperty->GetOwner<FArrayProperty>();
    static const FName Name_DisableReordering("EditFixedOrder");
    static const FName NAME_ArraySizeEnum("ArraySizeEnum");
    return OuterArrayProp != nullptr
        && !OuterArrayProp->HasMetaData(Name_DisableReordering)
        && !IsEditConst()
        && !OuterArrayProp->HasMetaData(NAME_ArraySizeEnum)
        && !FApp::IsGame();
}
```

NoElementDuplicate

- **功能描述：**去除TArray属性里数据项的Duplicate菜单项按钮。
- **使用位置：** UPROPERTY
- **引擎模块：** Container Property
- **元数据类型：** bool
- **限制类型：** TArray
- **常用程度：** ★

去除TArray属性里数据项的Duplicate菜单项按钮。

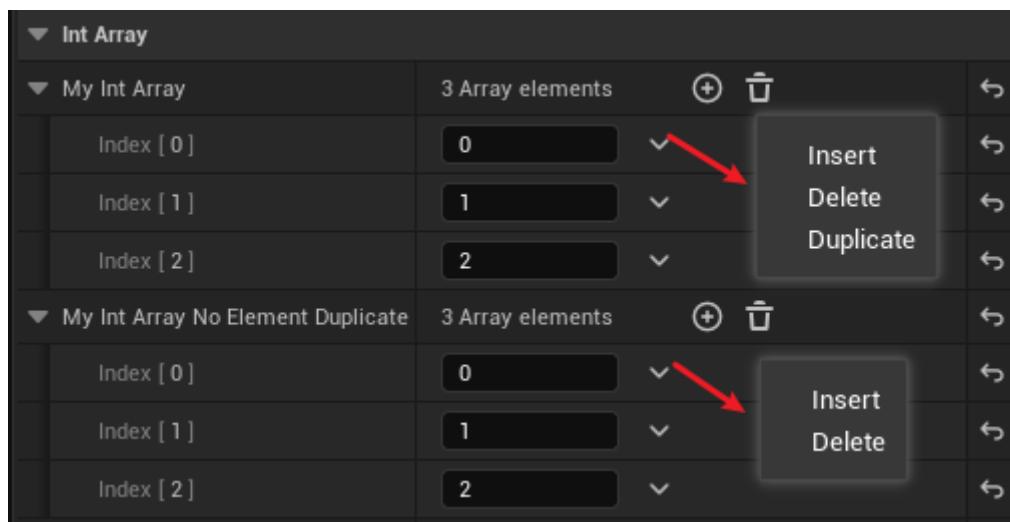
用于TArray属性，值可以是任何类型，可以是数值，结构，也可以是Object*。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = IntArray)  
    TArray<int32> MyIntArray;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = IntArray, meta =  
(NoElementDuplicate))  
    TArray<int32> MyIntArray_NoElementDuplicate;
```

效果：

可以看到带有NoElementDuplicate的数组，在值的右侧下拉箭头的菜单项里只有两项。



原理：

判断如有NoElementDuplicate，则只生成Insert_Delete菜单，否则是默认的Insert_Delete_Duplicate。当然要求当前属性是数组属性，且不是EditFixedSize固定的。

```
void GetRequiredPropertyButtons( TSharedRef<FPropertyName> PropertyNode,  
TArray<EPropertyButton::Type>& OutRequiredButtons, bool bUsingAssetPicker )  
{  
    const FArrayProperty* OuterArrayProp = NodeProperty->  
GetOwner<FArrayProperty>();  
  
    if( OuterArrayProp )  
    {  
        if( PropertyNode->HasNodeFlags(EPropertyNodeFlags::singleSelectOnly)  
&& !(OuterArrayProp->PropertyFlags & CPF_EditFixedSize) )  
        {  
            if (OuterArrayProp->HasMetaData(TEXT("NoElementDuplicate")))  
            {  
                OutRequiredButtons.Add( EPropertyButton::Insert_Delete );  
            }  
            else  
            {  
                OutRequiredButtons.Add( EPropertyButton::Insert_Delete_Duplicate );  
            }  
        }  
    }  
}
```

```
    }
}
}
```

ReadOnlyKeys

- **功能描述:** 使TMap属性的Key不能编辑。
- **使用位置:** UPROPERTY
- **引擎模块:** Container Property
- **元数据类型:** bool
- **限制类型:** TMap属性
- **常用程度:** ★★

使TMap属性的Key不能编辑。

意味着这个TMap里的元素是在这之前（构造函数里初始化等）就设置好的，但我们只希望用户更改值的内容，而不改Key的名字。这在某些情况下比较有用，比如以Platform作为Key，这样Platform的列表是固定的就不希望用户更改了。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ReadonlyKeysTest)
TMap<int32, FString> MyIntMap_NoReadOnlyKeys;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ReadonlyKeysTest, meta
= (ReadOnlyKeys))
TMap<int32, FString> MyIntMap_ReadOnlyKeys;
```

测试结果：

可见MyIntMap_ReadOnlyKeys的Key是灰色的，不可编辑。

▼ Read Only Keys Test	
▼ My Int Map No Read Only Keys	
0	Hello
1	Hello
2	Hello
▼ My Int Map Read Only Keys	
0	Hello
1	Hello
2	Hello

源码里搜到：

```
void FDetailPropertyRow::MakeNameOrKeyWidget( FDetailWidgetRow& Row, const  
TSharedPtr< FDetailWidgetRow> InCustomRow ) const  
{  
    if (PropertyHandle->HasMetaData(TEXT("ReadOnlyKeys")))  
    {  
        PropertyKeyEditor->GetPropertyNode()->  
        >SetNodeFlags(EPropertyNodeFlags::IsReadOnly, true);  
    }  
}
```

TitleProperty

- **功能描述：** 指定结构数组里的结构成员属性内容来作为结构数组元素的显示标题。
- **使用位置：** UPROPERTY
- **引擎模块：** Container Property
- **元数据类型：** string="abc"
- **限制类型：** TArray
- **常用程度：** ★★

指定结构数组里的结构成员属性内容来作为结构数组元素的显示标题。

重点是：

- 作用目标为结构数组TArray，其他的TSet, TMap不支持。
- TitleProperty顾名思义是用作标题的属性。但要更明确一些，标题指的是结构数组元素在细节面板里显示的标题。属性指的是结构数组里的结构里面的属性。
- 然后下一步是TitleProperty的格式要怎么写。根据引擎源码，TitleProperty最后是用FTitleMetadataFormatter来解析计算内容。通过查看其内部代码，可知其TitleProperty格式可以为：
 - 如果TitleProperty里包含“{”，则会把里面的字符串当作一个FTextFormat（以“{ArgName}...”组织的格式字符串）。最终格式是以“{PropertyName}...”组织的字符串去找多个对应的属性。
 - 否则就会把TitleProperty的全部当作PropertyName，直接去找对应的属性名称。

测试代码：

```
USTRUCT(BlueprintType)  
struct INSIDER_API FMyArrayTitleStruct  
{  
    GENERATED_BODY()  
public:  
    FMyArrayTitleStruct() = default;  
    FMyArrayTitleStruct(int32 id) : MyInt(id) {}  
    UPROPERTY(BlueprintReadWrite, EditAnywhere)  
    int32 MyInt = 123;  
    UPROPERTY(BlueprintReadWrite, EditAnywhere)  
    FString MyString=TEXT("Hello");  
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
```

```

float MyFloat=456.f;
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TitlePropertyTest)
TArray<FMyArrayTitleStruct> MyStructArray_NoTitleProperty;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TitlePropertyTest, meta =
(TitleProperty="{MyString}[{MyInt}]"))
TArray<FMyArrayTitleStruct> MyStructArray_HasTitleProperty;

```

测试效果：

可以发现，下面的数组元素的标题变为了“Hello[x]”，而不是默认的“3 members”。

▼ Title Property Test			
▼ My Struct Array No Title Property		3 Array elements	⊕ □
▶ Index [0]		3 members	▼
▶ Index [1]		3 members	▼
▶ Index [2]		3 members	▼
▼ My Struct Array Has Title Prop...		3 Array elements	⊕ □
⋮▶ Index [0]		Hello[0]	▼
▶ Index [1]		Hello[1]	▼
▶ Index [2]		Hello[2]	▼

原理：

属性编辑器里属性节点如果发现有TitleProperty，则会生成一个FTitleMetadataFormatter的TitlePropertyFormatter来进行字符串格式的解析和输出结果内容。其内部其实真正用的又是FTextFormat，这样才可以把多个属性的内容拼接成一个目标字符串。

当然SPropertyEditorTitle还注意到了如果有TitleProperty，可能会实时的改变，因此绑定了一个函数来进行Tick更新。

```

//绑定一个函数来每tick获取名字
void SPropertyEditorTitle::Construct( const FArguments& InArgs, const
TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    // If our property has title support we want to fetch the value every tick,
    // otherwise we can just use a static value
    static const FName NAME_TitleProperty = FName(TEXT("TitleProperty"));
    const bool bHasTitleProperty = InPropertyEditor->GetProperty() &&
InPropertyEditor->GetProperty()->HasMetaData(NAME_TitleProperty);
    if (bHasTitleProperty)
    {
        NameTextBlock =
            SNew(STextBlock)
            .Text(InPropertyEditor, &FPropertyEditor::GetDisplayName)
            .Font(NameFont);
    }
    else

```

```

    }

    NameTextBlock =
        SNew(STextBlock)
        .Text(InPropertyEditor->GetDisplayName())
        .Font(NameFont);
    }
}

FText FPropertyEditor::GetDisplayName() const
{
    FItemPropertyNode* ItemPropertyNode = PropertyNode->AsItemPropertyNode();

    if ( ItemPropertyNode != NULL )
    {
        return ItemPropertyNode->GetDisplayName();
    }

    if (const FComplexPropertyNode* ComplexPropertyName = PropertyNode-
>AsComplexNode())
    {
        const FText DisplayName = ComplexPropertyName->GetDisplayName();

        // Does this property define its own name?
        if (!DisplayName.IsEmpty())
        {
            return DisplayName;
        }
    }

    FString DisplayName;
    PropertyNode->GetQualifiedName( DisplayName, true );
    return FText::FromString(DisplayName);
}

//生成TitleFormatter来解析TitleProperty里面的内容，最后得出文字。发现不支持Map, Set，因此
只支持array。签名还有个判断ArrayIndex()==1的分支，走进普通属性
FText FItemPropertyNode::GetDisplayName() const
{
    if (CastField<F SetProperty>(ParentProperty) == nullptr &&
        CastField<F MapProperty>(ParentProperty) == nullptr)
    {
        // Check if this property has Title Property Meta
        static const FName NAME_TitleProperty = FName(TEXT("TitleProperty"));
        FString TitleProperty = PropertyPtr->GetMetaData(NAME_TitleProperty);
        if (!TitleProperty.IsEmpty())
        {
            // Find the property and get the right property handle
            if (PropertyStruct != nullptr)
            {
                const TSharedPtr<IPROPERTYHandle> ThisAsHandle =
                    PropertyEditorHelpers::GetPropertyHandle(NonConstThis->AsShared(), nullptr,
                    nullptr);
                TSharedPtr<FTitleMetadataFormatter> TitleFormatter =
                    FTitleMetadataFormatter::TryParse(ThisAsHandle, TitleProperty);
                if (TitleFormatter)
                {

```

```

        TitleFormatter->GetDisplayText(FinalDisplayName);
    }
}
}
}

//生成一个TitlePropertyFormatter
void SPropertyEditorArrayItem::Construct( const FArguments& InArgs, const
TSharedRef< class FPropertyEditor>& InPropertyEditor )
{
    static const FName TitlePropertyName = FName(TEXT("TitleProperty"));

    // if this is a struct property, try to find a representative element to use
    // as our stand in
    if (PropertyEditor->PropertyIsA( FStructProperty::StaticClass() ))
    {
        const FProperty* MainProperty = PropertyEditor->GetProperty();
        const FProperty* ArrayProperty = MainProperty ? MainProperty-
>GetOwner<const FProperty>() : nullptr;
        if (ArrayProperty) // should always be true
        {
            TitlePropertyFormatter =
FTitleMetadataFormatter::TryParse(PropertyEditor->GetPropertyHandle(),
ArrayProperty->GetMetaData(TitlePropertyName));
        }
    }
}
}

```

源码中例子：

读者还可以在UPropertyEditorTestObject里找到应用的例子。用testprops命令行就可以打开。

```

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=
(TitleProperty=IntPropertyInsideAStruct))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithTitle;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=(TitleProperty=""
{IntPropertyInsideAStruct} + {FloatPropertyInsideAStruct}"))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithFormattedTitle;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=
(TitleProperty=ErrorProperty))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithTitleError;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=(TitleProperty=""
{ErrorProperty}"))
TArray<FPropertyEditorTestBasicStruct>
StructPropertyArrayWithFormattedTitleError;

```

DebugTreeLeaf

- 功能描述：**阻止BlueprintDebugger展开该类的属性以加速编辑器里调试器的性能
- 使用位置：** UCLASS

- **引擎模块:** Debug
- **元数据类型:** bool
- **常用程度:** ★

阻止BlueprintDebugger展开该类的属性以加速编辑器里调试器的性能。当一个类拥有过多的属性（或递归嵌套太多属性）的时候，BlueprintDebugger在展示该类的属性数据的时候便会消耗过多的性能，造成编辑器卡顿。因此对于这种类，我们可以手动的加上该标志来阻止继续展开属性树，只到此为止。因此顾名思义，本类变成了调试时属性树的叶子。

在源码中只有UAnimDataModel用到了该标记，不过我们也可以在自己的类上加上该标记，当它拥有非常多的属性并且又不想调试它的数据的时候。

测试代码：

```
UCLASS(BlueprintType, meta = (DebugTreeLeaf))
class INSIDER_API UMyClass_DebugTreeLeaf :public UObject
{
    GENERATED_BODY()
    UMyClass_DebugTreeLeaf();

public:
    UPROPERTY(BlueprintReadWrite)
    TArray<int32> IntArray;
    UPROPERTY(BlueprintReadWrite)
    TMap<int32, FString> IntStringMap;
    UPROPERTY(BlueprintReadWrite)
    TSet<int32> IntSet;
};
```

蓝图中的效果：

UMyClass_DebugTreeLeaf对象作为一个类的成员变量（或者其他），在蓝图中调试查看变量，开启BlueprintDebugger查看变量属性时。如果没有加上DebugTreeLeaf，则会默认的展开所有内部属性。而如果加上DebugTreeLeaf标志，则会停止递归，阻止属性变量的展开。



AdvancedClassDisplay

- **功能描述:** 指定该类型的变量在高级显示里显示
- **使用位置:** UCLASS
- **引擎模块:** DetailsPanel
- **元数据类型:** bool
- **关联项:**
UCLASS: AdvancedClassDisplay
- **常用程度:** ★★★

AllowEditInlineCustomization

- **功能描述:** 允许EditInline的对象属性可以自定义属性细节面板来编辑该对象内的数据。
- **使用位置:** UPROPERTY
- **元数据类型:** string="abc"
- **关联项:** EditInline
- **常用程度:** ★

允许EditInline的对象属性可以自定义属性细节面板来编辑该对象内的数据。

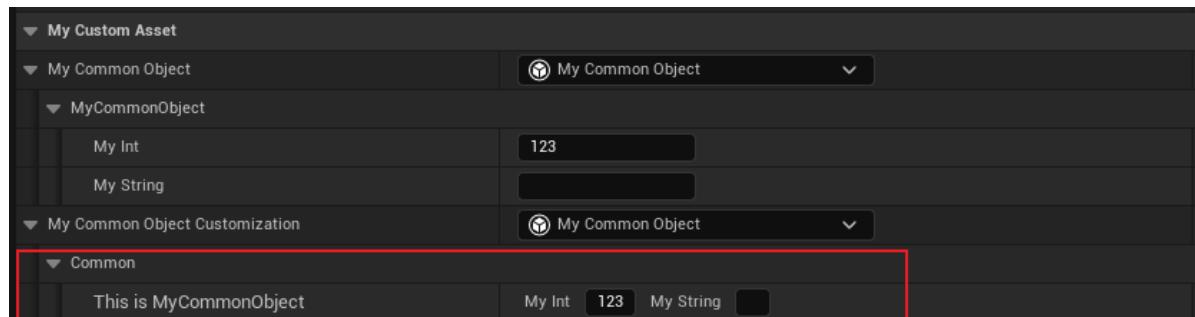
测试代码:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCustomAsset :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
    UMyCommonObject* MyCommonObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta =
    (EditInline, AllowEditInlineCustomization))
    UMyCommonObject* MyCommonObject_Customization;
};
```

效果:



要做到自定义EditInline的效果，采用自定义的IPropertyTypeCustomization和RegisterCustomPropertyParamsLayout也能做到。区别是，正如上面代码里的UMyCustomAsset里面有两个同类型的UMyCommonObject*对象，假如用IPropertyTypeCustomization的方式，就会导致两个变量都变成自定义的UI模式。而用AllowEditInlineCustomization就可以使得其中你想要的那个变成自定义方式，而其他的不做改变。

在用法上，AllowEditInlineCustomization必须配合自定义的FAssetEditorToolkit来自己定义一个DetailView（而不是只自定义某个类型在引擎统一的DetailView的显示），然后再自定义IDetailCustomization来提供具体的Widget，最后用RegisterInstancedCustomPropertyParamsLayout来关联起来。

```
Detailsview->RegisterInstancedCustomPropertyParamsLayout(UMyCommonObject::StaticClass(), FOnGetDetailCustomizationInstance::CreateStatic(&FMyCommonObjectDetailsCustomization::MakeInstance));
```

(这部分代码可参考MyCustomAsset的相关实现)

源码：

```
FDetailPropertyRow::FDetailPropertyRow(TSharedPtr<FPropertyNode> InPropertyNode,
TSharedRef<FDetailCategoryImpl> InParentCategory,
TSharedPtr<FComplexPropertyNode> InExternalRootNode)
{
    static FName InlineCustomizationKeyMeta("AllowEditInlineCustomization");
    if (PropertyNode->AsComplexNode() && ExternalRootNode.IsValid()) // AsComplexNode works both for objects and structs
    {
        // We are showing an entirely different object inline. Generate a layout for it now.
        if (IDetailsViewPrivate* DetailsView = InParentCategory->GetDetailsView())
        {
            ExternalObjectLayout = MakeShared<FDetailLayoutData>();
            DetailsView->UpdateSinglePropertyMap(InExternalRootNode,
*ExternalObjectLayout, true);
        }
    }
    else if (PropertyNode->HasNodeFlags(EPropertyNodeFlags::EditInlineNew) &&
PropertyNode->GetProperty()->HasMetaData(InlineCustomizationKeyMeta))
    {
        // Allow customization of 'edit inline new' objects if the metadata key has been specified.
        // The child of this node, if set, will be an object node that we will want to treat as an 'external object layout'
        TSharedPtr<FPropertyNode> ChildNode = PropertyNode-
>GetNumChildNodes() > 0 ? PropertyNode->GetChildNode(0) : nullptr;
        TSharedPtr<FComplexPropertyNode> ComplexChildNode =
StaticCastSharedPtr<FComplexPropertyNode>(ChildNode);
        if (ComplexChildNode.IsValid())
        {
            // We are showing an entirely different object inline. Generate a layout for it now.
        }
    }
}
```

```

        if (IDetailsViewPrivate* DetailsView = InParentCategory-
>GetDetailsView())
    {
        ExternalObjectLayout = MakeShared<FDetailLayoutData>();
        DetailsView->UpdatesinglePropertyMap(ComplexChildNode,
*ExternalObjectLayout, true);
    }
}
}

```

作用的原理是在创建FDetailPropertyRow的时候，即一个属性的在细节面板里的一行，如果有AllowEditInlineCustomization，就会创建ExternalObjectLayout，之后在FDetailPropertyRow的创建孩子的时候，就会判断是否有ExternalObjectLayout，如果有就可以应用上我们之前的Customization，如果没有就会应用默认的设置。如下是使用ExternalObjectLayout的代码：

```

void FDetailPropertyRow::GenerateChildrenForPropertyNode(
TSharedPtr<FPropertyNode>& RootPropertyNode, FDetailNodeList& OutChildren )
{
    // Children should be disabled if we are disabled
    TAttribute<bool> ParentEnabledState = TAttribute<bool>::CreateSP(this,
&FDetailPropertyRow::GetEnabledState);

    if( PropertyTypeLayoutBuilder.IsValid() && bShowCustomPropertyChildren )
    {
        const TArray< FDetailLayoutCustomization >& ChildRows =
PropertyTypeLayoutBuilder->GetChildCustomizations();

        for( int32 ChildIndex = 0; ChildIndex < ChildRows.Num(); ++ChildIndex )
        {
            TsharedRef<FDetailItemNode> ChildNodeItem =
MakeShared<FDetailItemNode>(ChildRows[ChildIndex],
ParentCategory.Pin().ToSharedRef(), ParentEnabledState);
            ChildNodeItem->Initialize();
            OutChildren.Add( ChildNodeItem );
        }
    }
    else if (ExternalObjectLayout.IsValid() && ExternalObjectLayout-
>DetailLayout->HasDetails())
    {
        OutChildren.Append(ExternalObjectLayout->DetailLayout-
>GetAllRootTreeNodes());
        //自定义的面板
    }
    else if ((bShowCustomPropertyChildren || !CustomPropertyWidget.IsValid()) &&
RootPropertyNode->GetNumChildNodes() > 0)
    {
        //正常的默认创建孩子
    }
}

```

源码里使用的一个例子是LevelSequence上Bind Actor上的Binding Property的细节面板。

假如我们采用一些代码去掉

```

USTRUCT()
struct FMovieSceneBindingPropertyInfo
{
    GENERATED_BODY()

    // Locator for the entry
    UPROPERTY(EditAnywhere, Category = "Default", meta=(AllowedLocators="Actor",
    DisplayName="Actor"))
    FUniversalObjectLocator Locator;

    // Flags for how to resolve the locator
    UPROPERTY()
    ELocatorResolveFlags ResolveFlags = ELocatorResolveFlags::None;

    UPROPERTY(Instanced, VisibleAnywhere, Category = "Default", meta=(EditInline,
    AllowEditInlineCustomization, DisplayName="Custom Binding Type"))
    UMovieSceneCustomBinding* CustomBinding = nullptr;
};

//自己Hack 代码
UObject* obj =
UInsiderLibrary::FindObjectByNameSmart(TEXT("MovieSceneBindingPropertyInfo"));
UScriptStruct* ss = Cast<UScriptStruct>(obj);
FProperty* prop = ss->FindPropertyByName(TEXT("CustomBinding"));
prop->RemoveMetaData(TEXT("AllowEditInlineCustomization"));

```

效果就会从左变到右边：



注册的方式也不同：

```

void
ULevelSequenceEditorSubsystem::AddBindingDetailCustomizations(TSharedRef<IDetails
View> DetailsView, TSharedPtr<ISequencer> ActiveSequencer, FGuid BindingGuid)
{
    // TODO: Do we want to create a generalized way for folks to add instanced
    // property layouts for other custom binding types so they can have access to
    // sequencer context?
    if (ActiveSequencer.IsValid())
    {
        UMovieSceneSequence* Sequence = ActiveSequencer-
>GetFocusedMovieSceneSequence();
        UMovieScene* MovieScene = Sequence ? Sequence->GetMovieScene() : nullptr;
        if (MovieScene)
        {
            FPropertyEditorModule& PropertyEditor =
FModuleManager::Get().LoadModuleChecked<FPropertyEditorModule>
(TEXT("PropertyEditor"));
            DetailsView-
>RegisterInstancedCustomPropertyTypeLayout(FMovieSceneBindingPropertyInfo::Static
Struct()->GetFName(), FOnGetPropertyTypeCustomizationInstance::CreateLambda([]
(TweakPtr<ISequencer> InSequencer, UMovieScene* InMovieScene, FGuid
InBindingGuid, ULevelSequenceEditorSubsystem* LevelSequenceEditorSubsystem)
{

```

```

        return
MakeShared<FMovieSceneBindingPropertyInfoDetailCustomization>(InSequencer,
InMovieScene, InBindingGuid, LevelSequenceEditorSubsystem);
}, ActiveSequencer.ToWeakPtr(), MovieScene, BindingGuid, this));

    DetailsView-
>RegisterInstancedCustomPropertyLayout(UMovieSceneSpawnableActorBinding::StaticClass(),
FOnGetDetailCustomizationInstance::CreateStatic(&FMovieSceneSpawnableActorBinding
BaseCustomization::GetInstance, ActiveSequencer.ToWeakPtr(), MovieScene,
BindingGuid));
}

}
}

```

AutoCollapseCategories

- 功能描述:** 指定类内部的属性目录自动折叠起来
- 使用位置:** UCLASS
- 引擎模块:** DetailsPanel
- 元数据类型:** strings="a, b, c"
- 关联项:**
UCLASS: AutoCollapseCategories, DontAutoCollapseCategories, AutoExpandCategories
- 常用程度:** ★★★

AutoExpandCategories

- 功能描述:** 指定类内部的属性目录自动展开起来
- 使用位置:** UCLASS
- 引擎模块:** DetailsPanel
- 元数据类型:** strings="a, b, c"
- 关联项:**
UCLASS: AutoExpandCategories, AutoCollapseCategories
- 常用程度:** ★★★

bShowOnlyWhenTrue

- 功能描述:** 根据编辑器config配置文件里字段值来决定当前属性是否显示。
- 使用位置:** UPROPERTY
- 引擎模块:** DetailsPanel
- 元数据类型:** string="abc"
- 常用程度:** ★

根据编辑器config配置文件里字段值来决定当前属性是否显示。

- 这个编辑器config配置文件，指的是GEditorPerProjectIni，因此一般是Config\DefaultEditorPerProjectUserSettings.ini

- 其中Section的名字是“UnrealEd.PropertyFilters”
- 然后Key的值就可以定了。

在源码里没有找到使用的例子，但这依然可以工作的。

测试代码：

```
D:\github\Gitworkspace\Hello\Config\DefaultEditorPerProjectUserSettings.ini
[UnrealEd.PropertyFilters]
ShowMyInt=true
ShowMyString=false

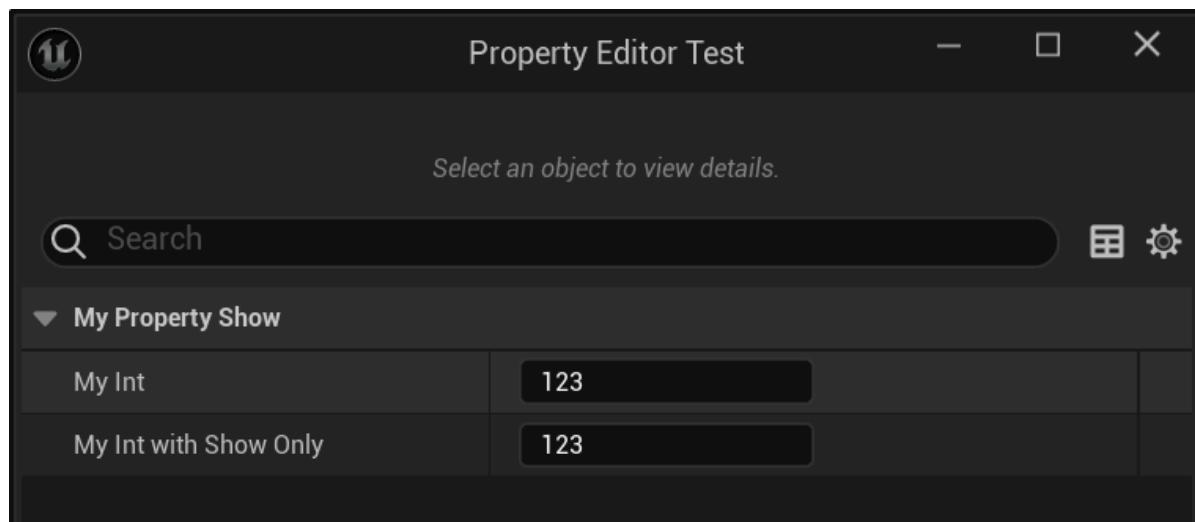
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Show :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (bShowOnlyWhenTrue =
"ShowMyInt"))
    int32 MyInt_WithShowOnly = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (bShowOnlyWhenTrue =
"ShowMyString"))
    FString MyString_WithShowOnly;
};
```

测试结果：

可见MyString_WithShowOnly就没有显示出来，因为我们在DefaultEditorPerProjectUserSettings中配置了ShowMyString=false。



原理：

就是取得config中的值用来决定属性框是否显示。

```

void FObjectPropertyNode::GetCategoryProperties(const TSet<UClass*>&
classesToConsider, const FProperty* CurrentProperty, bool
bShouldShowDisableEditOnInstance, bool bShouldShowHiddenProperties,
const TSet<FName>& CategoriesFromBlueprints, TSet<FName>&
CategoriesFromProperties, TArray<FName>& SortedCategories)
{
    bool bMetaDataAllowVisible = true;
    const FString& ShowOnlyWhenTrueString = CurrentProperty-
>GetMetaData(Name_bShowOnlyWhenTrue);
    if (ShowOnlyWhenTrueString.Len())
    {
        //ensure that the metadata visibility string is actually set to true in
        order to show this property
        GConfig->GetBool(TEXT("UnrealEd.PropertyFilters"),
*ShowOnlyWhenTrueString, bMetaDataAllowVisible, GEditorPerProjectIni);
    }

    if (bMetaDataAllowVisible)
    {
        if (PropertyEditorHelpers::ShouldBeVisible(*this, CurrentProperty) &&
!HiddenCategories.Contains(CategoryName))
        {
            if (!CategoriesFromBlueprints.Contains(CategoryName) &&
CategoriesFromProperties.Contains(CategoryName))
            {
                SortedCategories.AddUnique(CategoryName);
            }
            CategoriesFromProperties.Add(CategoryName);
        }
    }
}

void FCategoryPropertyNode::InitChildNodes()
{
    bool bMetaDataAllowVisible = true;
    if (!bShowHiddenProperties)
    {
        static const FName
Name_bShowOnlyWhenTrue("bShowOnlyWhenTrue");
        const FString& MetaDataVisibilityCheckString = It-
>GetMetaData(Name_bShowOnlyWhenTrue);
        if (MetaDataVisibilityCheckString.Len())
        {
            //ensure that the metadata visibility string is
            actually set to true in order to show this property
            // @todo Remove this
            GConfig->GetBool(TEXT("UnrealEd.PropertyFilters"),
*MetaDataVisibilityCheckString, bMetaDataAllowVisible, GEditorPerProjectIni);
        }
    }
}

```

Category

- **功能描述:** 指定属性在细节面板中的分类
- **使用位置:** UFUNCTION, UPROPERTY
- **引擎模块:** DetailsPanel
- **元数据类型:** string="A | B | C"
- **关联项:**
 UFUNCTION: Category
 UPROPERTY: Category
- **常用程度:** ★★★★☆

ClassGroupNames

- **功能描述:** 指定ClassGroup的名字
- **使用位置:** UCLASS
- **引擎模块:** DetailsPanel
- **元数据类型:** strings="a, b, c"
- **限制类型:** TArray
- **关联项:**
 UCLASS: ClassGroup
- **常用程度:** ★★★

DeprecatedNode

- **功能描述:** 用于BehaviorTreeNode或EnvQueryNode，说明该类已废弃，在编辑器中红色错误展示并有错误ToolTip提示
- **使用位置:** UCLASS
- **引擎模块:** DetailsPanel
- **元数据类型:** bool
- **限制类型:** BehaviorTreeNode, EnvQueryNode
- **常用程度:** ★★

在AI行为树或EQS的节点上设置，标记该节点已经弃用。

源码中的例子：

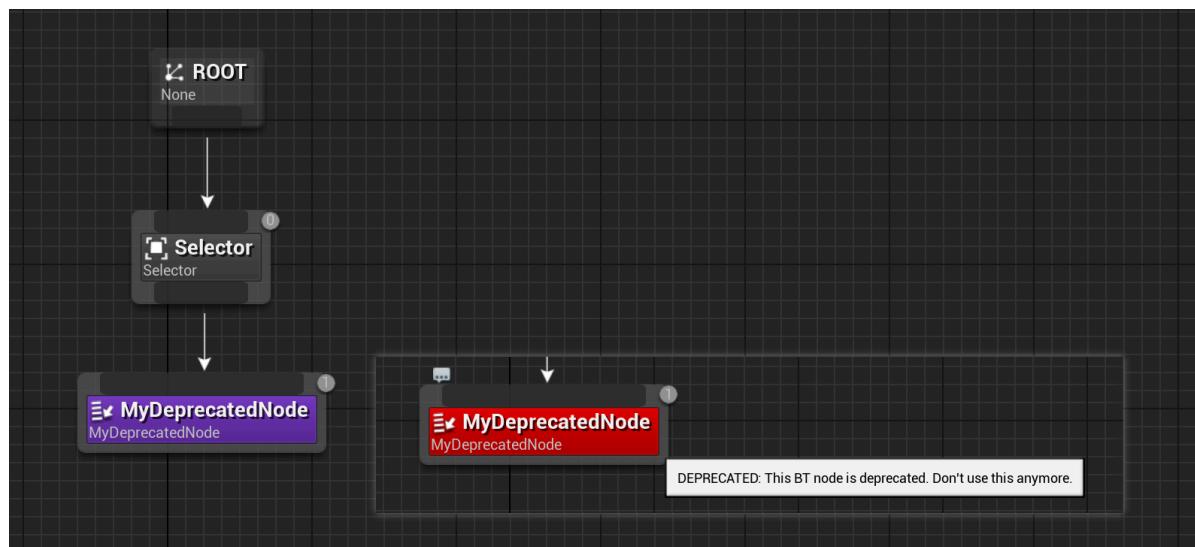
```
UCLASS(meta = (DeprecatedNode, DeprecationMessage = "Please use IsAtLocation
decorator instead."), MinimalAPI)
class UBTDecorator_ReachedMoveGoal : public UBTDecorator
{
    GENERATED_UCLASS_BODY()
};

UCLASS(MinimalAPI, meta=(DeprecatedNode, DeprecationMessage = "This class is now
deprecated, please use RunMode supporting random results instead."))
class UEnvQueryTest_Random : public UEnvQueryTest
{
    GENERATED_UCLASS_BODY()
};
```

C++测试代码：

```
UCLASS(meta = (DeprecatedNode, DeprecationMessage = "This BT node is deprecated.
Don't use this anymore."), MinimalAPI)
class UBTTTask_MyDeprecatedNode : public UBTTTaskNode
{
    GENERATED_UCLASS_BODY()
};
```

行为树里的结果，如果加上DeprecatedNode，就会红色显示，并提示错误信息。



源码里测试的代码：

```
FString FGraphNodeClassHelper::GetDeprecationMessage(const UClass* Class)
{
    static FName MetaDeprecated = TEXT("DeprecatedNode");
    static FName MetaDeprecatedMessage = TEXT("DeprecationMessage");
    FString DefDeprecatedMessage("Please remove it!");
    FString DeprecatedPrefix("DEPRECATED");
    FString DeprecatedMessage;
```

```

    if (Class && Class->HasAnyClassFlags(CLASS_Native) && Class-
>HasMetaData(MetaDeprecated))
{
    DeprecatedMessage = DeprecatedPrefix + TEXT(": ");
    DeprecatedMessage += Class->HasMetaData(MetaDeprecatedMessage) ? Class-
>GetMetaData(MetaDeprecatedMessage) : DefDeprecatedMessage;
}

return DeprecatedMessage;
}

```

DisplayAfter

- 功能描述:** 使本属性在指定的属性之后显示。
- 使用位置:** UPROPERTY
- 引擎模块:** DetailsPanel
- 元数据类型:** string="abc"
- 常用程度:** ★★★

使本属性在指定的属性之后显示。

- 默认情况下，属性在细节面板中的顺序是依照头文件中的定义顺序。但如果我们要自己调节这个顺序，就可以用该标记。
- 限制条件是这两个属性必须得是在同一个Category下。这也很好理解，Category组织的优先级肯定更大。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Priority :public UObject
{
GENERATED_BODY()
public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest)
int32 MyInt = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest)
FString MyString;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest, meta =
(DisplayAfter = "MyInt"))
int32 MyInt_After = 123;
public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest2, meta =
(DisplayAfter = "MyInt"))
int32 MyInt_After2 = 123;

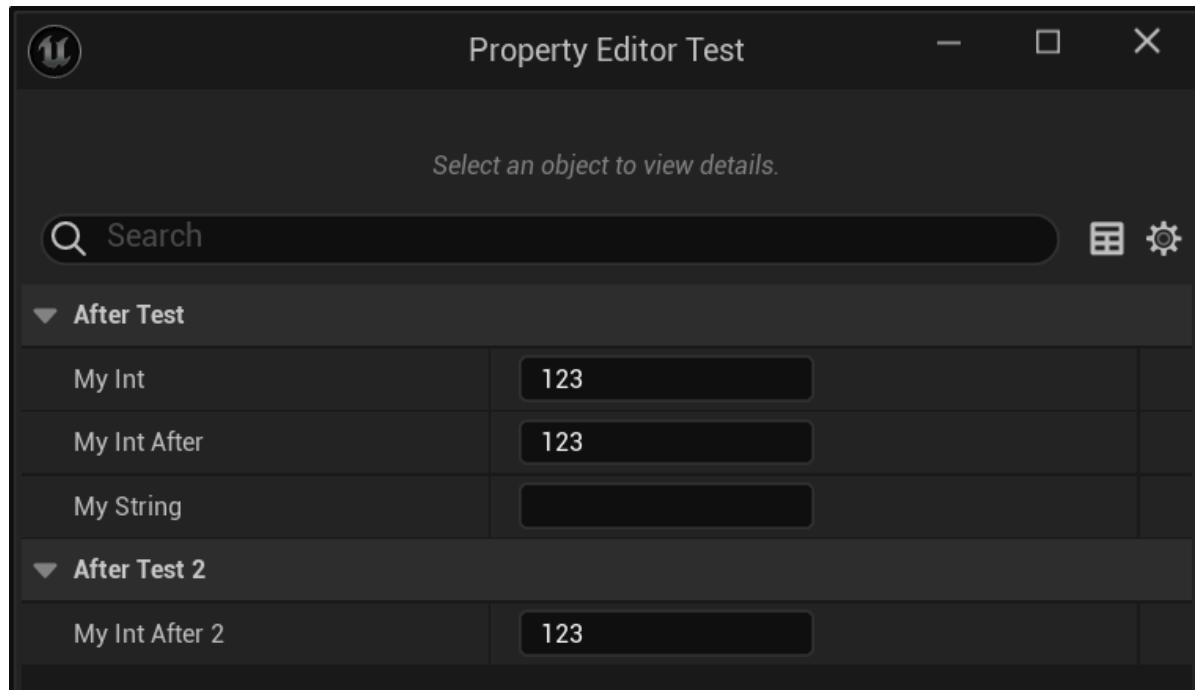
};

```

测试效果：

可见MyInt_After直接在Int后显示。

而MyInt_After2 因为在不同的Category下，因此就保留原样。



原理：

检查该属性如果有DisplayAfter，就把它插入在指定的属性之后。

```
void PropertyEditorHelpers::OrderPropertiesFromMetadata(TArray<FProperty*>& Properties)
{
    const FString& DisplayAfterPropertyName = Prop->GetMetaData(NAME_DisplayAfter);
    if (DisplayAfterPropertyName.IsEmpty())
    {
        InsertProperty(OrderedProperties);
    }
    else
    {
        TArray<TPair<FProperty*, int32>>& DisplayAfterProperties =
        DisplayAfterPropertyMap.FindOrAdd(FName(*DisplayAfterPropertyName));
        InsertProperty(DisplayAfterProperties);
    }
}
```

DisplayPriority

- 功能描述：**指定本属性在细节面板的显示顺序优先级，越小的优先级越高。
- 使用位置：** UPROPERTY
- 引擎模块：** DetailsPanel
- 元数据类型：** int32
- 常用程度：** ★★★

指定本属性在细节面板的显示顺序优先级，越小的优先级越高。

- 如果有DisplayAfter的设置，则DisplayAfter的优先级更高。

- 同样的限制得是在同Category里。

测试代码：

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =
(DisplayPriority = 3))
    int32 MyInt_P3 = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =
(DisplayPriority = 1))
    int32 MyInt_P1 = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =
(DisplayPriority = 2))
    int32 MyInt_P2 = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =
(DisplayPriority = 4,DisplayAfter="MyInt_P1"))
    int32 MyInt_P4 = 123;
```

测试结果：

P4即使优先级比较低，但因为DisplayAfter也仍然排在了P1之后。

▼ Priority Test	
My Int P1	123
My Int P4	123
My Int P2	123
My Int P3	123

原理：

排序的逻辑在这个函数内，自行查看就好。一个简单的插入排序算法。

```
void PropertyEditorHelpers::OrderPropertiesFromMetadata(TArray<FProperty*>&
Properties)
{}
```

EditCondition

- 功能描述：**给一个属性指定另外一个属性或者表达式来作为是否可编辑的条件。
- 使用位置：**UPROPERTY
- 引擎模块：**DetailsPanel
- 元数据类型：**string="abc"
- 关联项：**EditConditionHides, InlineEditConditionToggle, HideEditConditionToggle
- 常用程度：**★★★★★

给一个属性指定另外一个属性或者表达式来作为是否可编辑的条件。

- 表达式里引用的属性必须得是同一个类或结构范围内的。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_EditCondition_Test :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
    bool MyBool;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
    (EditCondition = "MyBool"))
    int32 MyInt_EditCondition = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
    (EditCondition = "!MyBool"))
    int32 MyInt_EditCondition_Not = 123;

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MyFirstInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MySecondInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
    meta = (EditCondition = "(MyFirstInt+MySecondInt)==500"))
    int32 MyInt_EditConditionExpression = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
    meta = (EditCondition = "!((MyFirstInt+MySecondInt)==500"))
    int32 MyInt_EditConditionExpression_Not = 123;
};

```

测试结果：

- 可以通过bool单个属性来控制其他属性是否可以编辑
- 也可以通过一个表达式引入更复杂的计算机制来决定是否来编辑。



原理：

在细节面板的属性初始化的时候，会判断该属性EditCondition设置，如果有值，会创建FEditConditionParser来解析表达式然后求值。

```

void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{

```

```

        const FString& EditConditionString = MyProperty-
>GetMetaData(TEXT("EditCondition"));

        // see if the property supports some kind of edit condition and this isn't
        // the "parent" property of a static array
        const bool bIsStaticArrayParent = MyProperty->ArrayDim > 1 && GetArrayIndex()
!= -1;
        if (!EditConditionString.IsEmpty() && !bIsStaticArrayParent)
{
    EditConditionExpression = EditConditionParser.Parse(EditConditionString);
    if (EditConditionExpression.IsValid())
    {
        EditConditionContext = MakeShareable(new
FEditConditionContext(*this));
    }
}

}

```

EditConditionHides

- 功能描述:** 在已经有EditCondition的情况下，指定该属性在EditCondition不满足的情况下隐藏起来。
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 关联项:** EditCondition
- 常用程度:** ★★★★☆

在已经有EditCondition的情况下，指定该属性在EditCondition不满足的情况下隐藏起来。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_EditCondition_Test :public UObject
{
GENERATED_BODY()

public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
bool MyBool;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
(EditConditionHides, EditCondition = "MyBool"))
int32 MyInt_EditCondition_Hides = 123;

public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
int32 MyFirstInt = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
int32 MySecondInt = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
meta = (EditConditionHides, EditCondition = "(MyFirstInt+MySecondInt)==500"))

```

```
int32 MyInt_EditConditionExpression_Hides = 123;  
};
```

测试效果：

下面的图中可以明显见到两个属性随着条件的满足显示了出来。



原理：

其实就是加了个是否显示的判断。

```
bool FPropertyNode::IsOnlyVisibleWhenEditConditionMet() const  
{  
    static const FName Name_EditConditionHides("EditConditionHides");  
    if (Property.IsValid() && Property->HasMetaData(Name_EditConditionHides))  
    {  
        return HasEditCondition();  
    }  
  
    return false;  
}
```

EditInline

- 功能描述：**为对象属性创建一个实例，并作为子对象。
- 使用位置：**UPROPERTY
- 引擎模块：**DetailsPanel
- 元数据类型：**bool
- 关联项：**NoEditInline, AllowEditInlineCustomization, ForceInlineRow
UPROPERTY: Instanced
- 常用程度：**★★★

为对象属性创建一个实例，并作为子对象。

也可以手动设置。如果UClass上有EditInlineNew，但是属性上没有Instanced，这个时候可以手动的设置EditInline然后通过自己手动赋值对象引用属性来使得这个对象可以直接编辑。

和ShowInnerProperties是否等价？EditInline在对象容器（Array, Map, Set）的情况下，也可以使用。但ShowInnerProperties只能在单个对象属性上生效。

可以设置在Struct上？看源码里也有该设置。但其实并没有效果。

示例代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite)  
UMyProperty_EditInline_Sub* MyObject;  
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))  
UMyProperty_EditInline_Sub* MyObject_EditInline;  
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (NoEditInline))  
UMyProperty_EditInline_Sub* MyObject_NoEditInline;
```

```

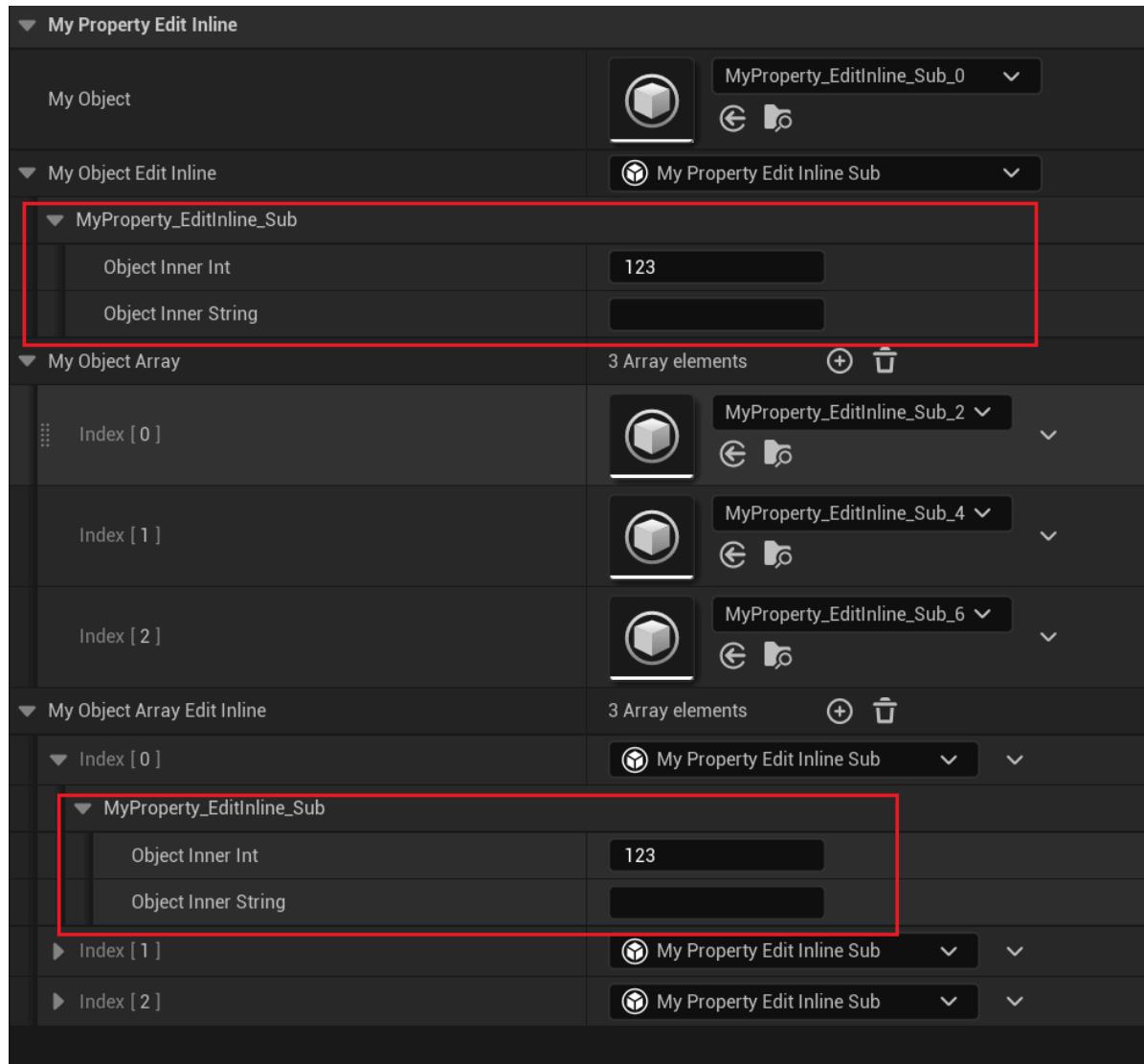
UPROPERTY(EditAnywhere, BlueprintReadWrite)
TArray<UMyProperty_EditInline_Sub*> MyObjectArray;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
TArray<UMyProperty_EditInline_Sub*> MyObjectArray_EditInline;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (NoEditInline))
TArray<UMyProperty_EditInline_Sub*> MyObjectArray_NoEditInline;

```

蓝图效果：



原理：

会相应的设置EPropertyNodeFlags::EditInlineNew。

```

void FPropertyNode::InitNode(const FPropertyParamsInitParams& InitParams)
{
    // we are EditInlineNew if this property has the flag, or if inside a
    // container that has the flag.
    bIsEditInlineNew = GotReadAddresses && bIsObjectOrInterface &&
        !MyProperty->HasMetaData(Name_NoEditInline) &&
        (MyProperty->HasMetaData(Name>EditInline) || (bIsInsideContainer &&
        OwnerProperty->HasMetaData(Name>EditInline)));
}

```

```

        bShowInnerObjectProperties = bIsObjectOrInterface && MyProperty-
>HasMetaData(Name_ShowInnerProperties);

    if (bIsEditInlineNew)
    {
        SetNodeFlags(EPropertyNodeFlags::EditInlineNew, true);
    }
    else if (bShowInnerObjectProperties)
    {
        SetNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties, true);
    }
}

bool SPropertyEditorEditInline::Supports( const FPropertyName* InTreeNode, int32
InArrayIdx )
{
    return InTreeNode
        && InTreeNode->HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
        && InTreeNode->FindObjectItemParent()
        && !InTreeNode->IsPropertyConst();
}

void FItemPropertyNode::InitExpansionFlags(void)
{
    FProperty* MyProperty = GetProperty();

    if (TSharedPtr<FPropertyName>& valueNode = GetOrCreateOptionalValueNode())
    {
        // This is a set optional, so check its SetValue instead.
        MyProperty = valueNode->GetProperty();
    }

    bool bExpandableType = CastField<FStructProperty>(MyProperty)
        || (CastField<FArrayProperty>(MyProperty) || CastField<FSetProperty>
        (MyProperty) || CastField<FMapProperty>(MyProperty));

    if (bExpandableType
        || HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
        || HasNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties)
        || (MyProperty->ArrayDim > 1 && ArrayIndex == -1))
    {
        SetNodeFlags(EPropertyNodeFlags::CanBeExpanded, true);
    }
}

```

ForceInlineRow

- **功能描述:** 强制TMap属性里的结构key和其他Value合并到同一行来显示
- **使用位置:** UPROPERTY
- **元数据类型:** bool
- **关联项:** EditInline
- **常用程度:** ★

强制TMap属性里的结构key和其他Value合并到同一行来显示。这里要注意的点是：

- 本属性是TMap属性，这样才有Key。 TArray或TSet是没有用的。
- FStruct作为Key，这样源码里的机制才能生效，因为判断的就是Key Property
- 该FStruct有注册相关的IPropertyTypeCustomization，这样才能自定义该结构的显示UI
- 该IPropertyTypeCustomization的ShouldInlineKey返回false（默认就是），否则true的话则不管有没有标ForceInlineRow，都会合并成一行

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    TMap<FMyCommonStruct, int32> MyStructMap;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ForceInlineRow))  
    TMap<FMyCommonStruct, int32> MyStructMap_ForceInlineRow;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    TMap<int32, FMyCommonStruct> MyStructMap2;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ForceInlineRow))  
    TMap<int32, FMyCommonStruct> MyStructMap_ForceInlineRow2;  
  
  
  
void FMyCommonStructCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>  
PropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationutils&  
CustomizationUtils)  
{  
    HeaderRow.NameContent() [SNew(STextBlock).Text(INVTEXT("This is  
MyCommonStruct"))];  
  
    TSharedPtr<IPropertyHandle> IntPropertyHandle = PropertyHandle->  
GetChildHandle(GET_MEMBER_NAME_CHECKED(FMyCommonStruct, MyInt));  
    TSharedPtr<IPropertyHandle> StringPropertyHandle = PropertyHandle->  
GetChildHandle(GET_MEMBER_NAME_CHECKED(FMyCommonStruct, MyString));  
  
    HeaderRow.ValueContent()  
    [  
        SNew(SHorizontalBox)  
        + SHorizontalBox::Slot()  
        .Padding(5.0f, 0.0f).Autowidth()  
        [  
            IntPropertyHandle->CreatePropertyNameWidget()  
        ]  
        + SHorizontalBox::Slot()  
        .Padding(5.0f, 0.0f).Autowidth()  
        [  
            IntPropertyHandle->CreatePropertyValueWidget()  
        ]  
        + SHorizontalBox::Slot()  
        .Padding(5.0f, 0.0f).Autowidth()  
        [  
            StringPropertyHandle->CreatePropertyNameWidget()  
        ]  
    ]  
}
```

```

        ]
        + SHorizontalBox::Slot()
        .Padding(5.0f, 0.0f).AutoWidth()
        [
            StringPropertyHandle->CreatePropertyValueWidget()
        ]
    ];

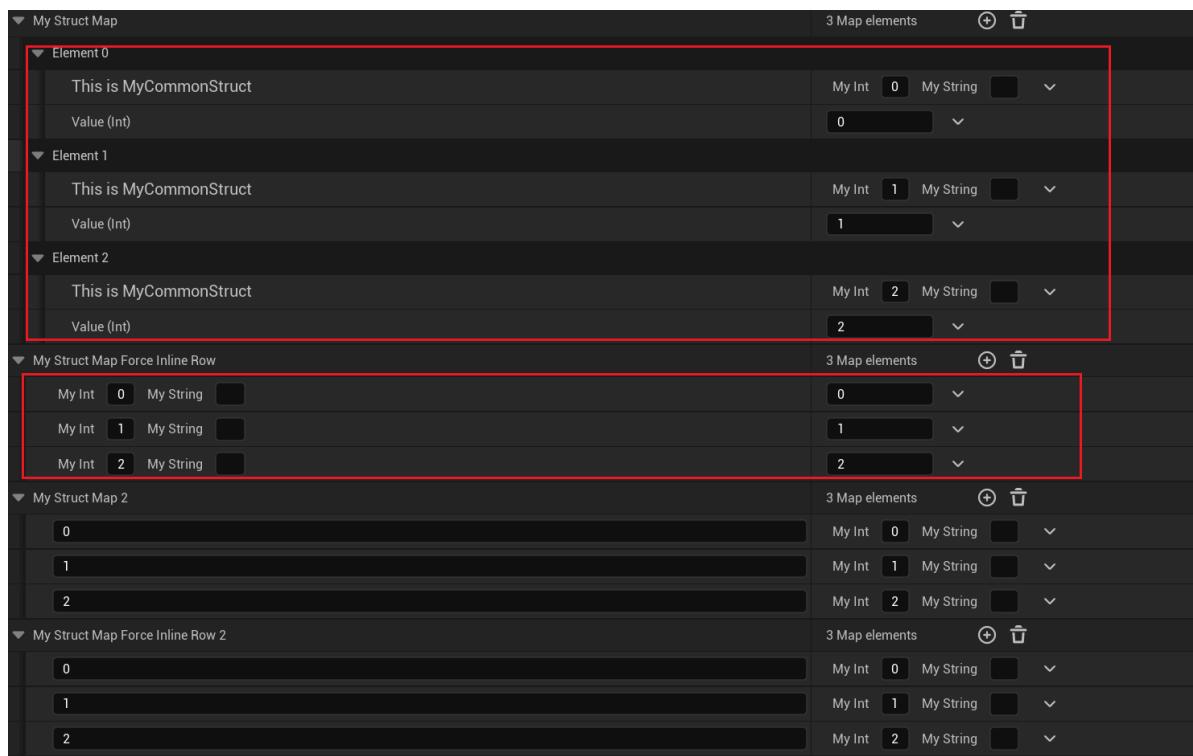
}

```

测试效果：

可以见到MyStructMap的数据项展示就分为了两行。而带有ForceInlineRow之后，数据项UI就合并为一行，显得更加的简洁。

在下面也特别观察到如果把FStruct作为Value，则是没有这个区别的。



假如不注册FMyCommonStruct相应的IPropertyTypeCustomization的话，则结构的属性UI采用默认方式显示，则都是分为两行。

My Struct Map		3 Map elements		
▼ Element 0				
▼ Key (My Common Struct)		2 members		
My Int	0			
My String				
Value (Int)	0			
▼ Element 1				
► Key (My Common Struct)		2 members		
Value (Int)	1			
▼ Element 2				
► Key (My Common Struct)		2 members		
Value (Int)	2			
▼ My Struct Map Force Inline Row		3 Map elements		
▼ Element 0				
► Key (My Common Struct)		2 members		
Value (Int)	0			
▼ Element 1				
► Key (My Common Struct)		2 members		
Value (Int)	1			
▼ Element 2				
► Key (My Common Struct)		2 members		
Value (Int)	2			
▼ My Struct Map 2		3 Map elements		
▼ 0		2 members		
My Int	0			
My String				
► 1		2 members		
► 2		2 members		
▼ My Struct Map Force Inline Row 2		3 Map elements		
► 0		2 members		
► 1		2 members		
► 2		2 members		

而假如FMyCommonStruct的I.PropertyTypeCustomization的ShouldInlineKey返回true，则会导致即使没有ForceInlineRow也会把该拥有该结构作为Key的属性给都合并为一行显示，这个时候就失去ForceInlineRow的作用和区别了。

My Struct Map				3 Map elements	
My Int	0	My String		0	
My Int	1	My String		1	
My Int	2	My String		2	
My Struct Map Force Inline Row				3 Map elements	
My Int	0	My String		0	
My Int	1	My String		1	
My Int	2	My String		2	
My Struct Map 2				3 Map elements	
0				My Int	0 My String
1				My Int	1 My String
2				My Int	2 My String
My Struct Map Force Inline Row 2				3 Map elements	
0				My Int	0 My String
1				My Int	1 My String
2				My Int	2 My String

原理：

该部分逻辑也同样处于在FDetailPropertyRow的构造函数创建过程中，判断是否有GetPropertyKeyNode，则其实是在要求TMap属性。

接着作为Key的类型，如果是UObject*，则因为NeedsKeyNode一直返回false，则无论如何都会进入MakePropertyEditor的分支。

因此此项测试的类型其实是Struct，这样就必须依赖bInlineRow 和FoundPropertyCustomisation 的配合。这个时候就必须有I.PropertyTypeCustomization才会进入分支，而且如果I.PropertyTypeCustomization::ShouldInlineKey()返回true，则就不管属性上的ForceInlineRow如何，都会进入分支。否则就靠属性上的ForceInlineRow，这个时候才是这个Meta发挥作用的时候。

```

FDetailPropertyRow::FDetailPropertyRow(TSharedPtr<FPropertyNode> InPropertyName,
TSharedRef<FDetailCategoryImpl> InParentCategory,
TSharedPtr<FComplexPropertyNode> InExternalRootNode)
{
    if (PropertyName->GetPropertyKeyNode().IsValid())
    {
        TSharedPtr<I.PropertyTypeCustomization> FoundPropertyCustomisation =
GetPropertyParams(PropertyName->GetPropertyKeyNode().ToSharedRef(),
ParentCategory.Pin().ToSharedRef());

        bool bInlineRow = FoundPropertyParams->ShouldInlineKey() : false;

        static FName InlineKeyMeta("ForceInlineRow");
        bInlineRow |= InPropertyName->GetParentNode()->GetProperty()-
>HasMetaData(InlineKeyMeta);

        // Only create the property editor if it's not a struct or if it requires
        // to be inlined (and has customization)
        if (!NeedsKeyNode(PropertyNameRef, InParentCategory) || (bInlineRow &&
FoundPropertyParams != nullptr))
        {
            CachedKeyCustomTypeInterface = FoundPropertyParams;

            MakePropertyEditor(PropertyName->GetPropertyKeyNode().ToSharedRef(),
utilities, PropertyKeyEditor);
        }
    }
}

```

```

        }
    }

bool FDetailPropertyRow::NeedsKeyNode(TSharedRef<FPropertyNode> InPropertyName,
TSharedRef<FDetailCategoryImpl> InParentCategory)
{
    FStructProperty* KeyStructProp = CastField<FStructProperty>(InPropertyName->GetPropertyKeyNode() ->GetProperty());
    return KeyStructProp != nullptr;
}

```

源码里使用的例子：

在源码里搜索发现到该例子，但实际上其实这里HLODSetups上的ForceInlineRow并不能起作用。

```

USTRUCT()
struct FRuntimePartitionDesc
{
    GENERATED_USTRUCT_BODY()

#if WITH_EDITORONLY_DATA
    /** Partition class */
    UPROPERTY(EditAnywhere, Category = RuntimeSettings)
    TSubclassOf<URuntimePartition> Class;

    /** Name for this partition, used to map actors to it through the
Actor.RuntimeGrid property */
    UPROPERTY(EditAnywhere, Category = RuntimeSettings, Meta = (EditCondition =
"Class != nullptr", HideEditConditionToggle))
    FName Name;

    /** Main partition object */
    UPROPERTY(VisibleAnywhere, Category = RuntimeSettings, Instanced, Meta =
(EditCondition = "Class != nullptr", HideEditConditionToggle, NoResetToDefault,
TitleProperty = "Name"))
    TObjectPtr<URuntimePartition> MainLayer;

    /** HLOD setups used by this partition, one for each layers in the hierarchy
*/
    UPROPERTY(EditAnywhere, Category = RuntimeSettings, Meta = (EditCondition =
"Class != nullptr", HideEditConditionToggle, ForceInlineRow))
    TArray<FRuntimePartitionHLODSetup> HLODSetups;
#endif

#if WITH_EDITOR
    void UpdateHLODPartitionLayers();
#endif
};

```

HideBehind

- **功能描述：**只在指定的属性为true或不为空的时候本属性才显示
- **使用位置：** UPROPERTY

- **引擎模块:** DetailsPanel
- **元数据类型:** string="abc"
- **限制类型:** Foliage模块中
- **常用程度:** ★

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Placement, meta=(UIMin = 0, ClampMin = 0, UIMax = 359, ClampMax = 359, HideBehind="AlignToNormal"))
float AlignMaxAngle;
```

只在Foliage里用到，其实用EditCondition就可以达到同样的效果了。

HideCategories

- **功能描述:** 隐藏的类别
- **使用位置:** UCLASS
- **引擎模块:** DetailsPanel
- **元数据类型:** strings="a, b, c"
Related To UCLASS: ShowCategories (../../Specifier/UCLASS>ShowCategories.md)
- **关联项:** ShowCategories (ShowCategories.md)
- **常用程度:** ★★★

HideEditConditionToggle

- **功能描述:** 用在使用EditCondition的属性上，表示该属性不想要其EditCondition用到的属性被隐藏起来。
- **使用位置:** UPROPERTY
- **引擎模块:** DetailsPanel
- **元数据类型:** bool
- **限制类型:** bool
- **关联项:** EditCondition
- **常用程度:** ★★★★☆

用在使用EditCondition的属性上，表示该属性不想要其EditCondition用到的属性被隐藏起来。和InlineEditConditionToggle是有相反的作用。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (InlineEditConditionToggle))  
    bool MyBool_Inline;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (EditCondition = "MyBool_Inline"))  
    int32 MyInt_EditCondition_UseInline = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (HideEditConditionToggle, EditCondition =  
    "MyBool_Inline"))  
    int32 MyInt_EditCondition_UseInline_Hide = 123;  
};
```

测试效果：

 F:\UnrealSpecifiers\Doc\Meta\DetailsPanel\HideEditConditionToggle\HideEditConditionToggle.g
if

原理：

判断如果有HideEditConditionToggle，就支持不支持当前行有单选框的按钮。

```
bool FPropertyNode::SupportsEditConditionToggle() const  
{  
    if (!Property.IsValid())  
    {  
        return false;  
    }  
  
    FProperty* MyProperty = Property.Get();  
  
    static const FName Name_HideEditConditionToggle("HideEditConditionToggle");  
    if (EditConditionExpression.IsValid() && !Property->HasMetaData(Name_HideEditConditionToggle))  
    {  
        const FBoolProperty* ConditionalProperty = EditConditionContext->GetSingleBoolProperty(EditConditionExpression);  
        if (ConditionalProperty != nullptr)  
        {  
            // There are 2 valid states for inline edit conditions:  
            // 1. The property is marked as editable and has  
            InlineEditConditionToggle set.  
            // 2. The property is not marked as editable and does not have  
            InlineEditConditionToggle set.  
            // In both cases, the original property will be hidden and only show  
            up as a toggle.  
        }  
    }  
}
```

```

        static const FName
Name_InlineEditConditionToggle("InlineEditConditionToggle");
        const bool bIsInlineEditCondition = ConditionalProperty-
>HasMetaData(Name_InlineEditConditionToggle);
        const bool bIsEditable = ConditionalProperty-
>HasAllPropertyFlags(CPF_Edit);

        if (bIsInlineEditCondition == bIsEditable)
{
    return true;
}

        if (bIsInlineEditCondition && !bIsEditable)
{
    UE_LOG(LogPropertyName, Warning, TEXT("Property being used as
inline edit condition is not editable, but has redundant
InlineEditConditionToggle flag. Field \\\"%s\\\" in class \\\"%s\\\"."),
*ConditionalProperty->GetNameCPP(), *Property->GetOwnerStruct()->GetName());
    return true;
}

// The property is already shown, and not marked as inline edit
condition.
        if (!bIsInlineEditCondition && bIsEditable)
{
    return false;
}
}

return false;
}

```

HideInDetailPanel

- 功能描述:** 在Actor的事件面板里隐藏该动态多播委托属性。
- 使用位置:** UPROPERTY
- 引擎模块:** DetailsPanel
- 元数据类型:** bool
- 限制类型:** Actor里的动态多播委托
- 常用程度:** ★★

在Actor的事件面板里隐藏该动态多播委托属性。

测试代码:

```

UCLASS(BlueprintType, Blueprintable)
class INSIDER_API AMyProperty_HideInDetailPanel :public AActor
{
GENERATED_BODY()
public:
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnMyHideTestEvent);

```

```

UPROPERTY(BlueprintAssignable, Category = "Event")
FOnMyHideTestEvent MyEvent;

UPROPERTY(BlueprintAssignable, Category = "Event", meta =
(HideInDetailPanel))
FOnMyHideTestEvent MyEvent_HideInDetailPanel;
};

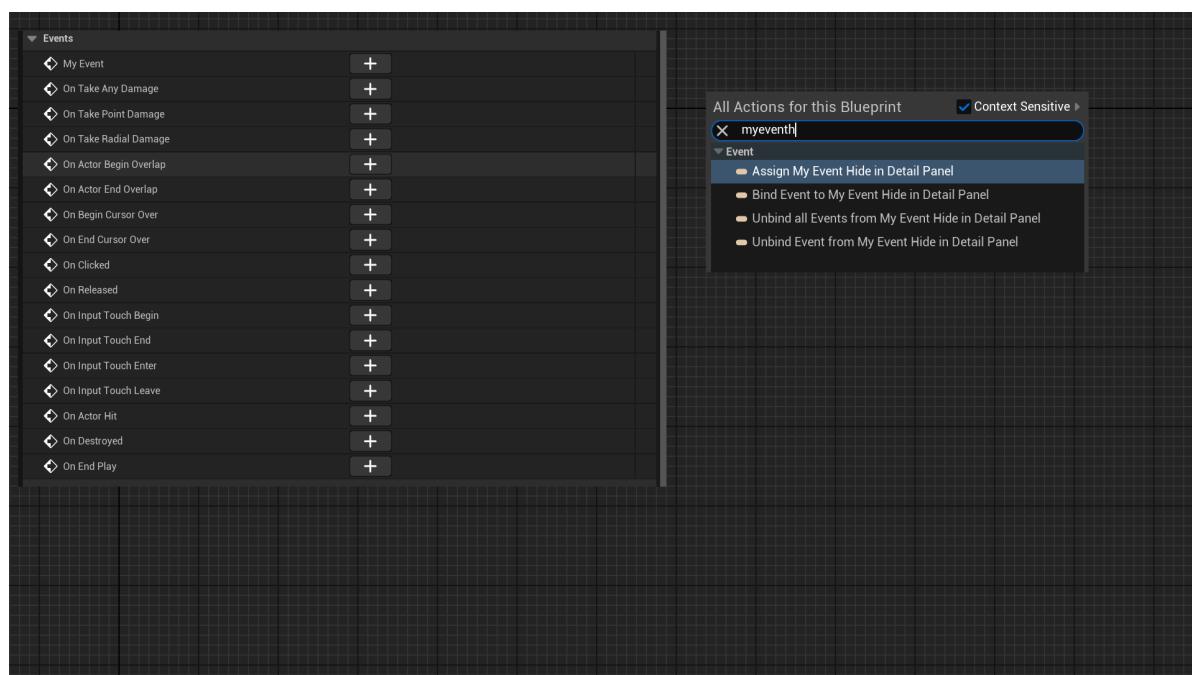
```

测试效果：

测试步骤是在蓝图里创建AMyProperty_HideInDetailPanel 的子类，然后观察Event的显示情况。

可见MyEvent会显示在Class Defaults里的Events，而MyEvent_HideInDetailPanel则没有显示。

不过MyEvent_HideInDetailPanel依然是可以在蓝图里进行绑定，只不过默认没显示在UI上而已。



原理：

先判断没有这个标记，然后创建相应的UI控件。

```

void FActorDetails::AddEventsCategory(IDetailLayoutBuilder& DetailBuilder)
{
    IDetailCategoryBuilder& EventsCategory =
    DetailBuilder.EditCategory("Events", FText::GetEmpty(),
    ECategoryPriority::Uncommon);
    static const FName HideInDetailPanelName("HideInDetailPanel");

    // Find all the Multicast delegate properties and give a binding button
    // for them
    for (TFieldIterator<FMulticastDelegateProperty> PropertyIt(Actor-
    >GetClass(), EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
    {
        FMulticastDelegateProperty* Property = *PropertyIt;

        // Only show BP assiangular, non-hidden delegates
    }
}

```

```

        if (!Property->HasAnyPropertyFlags(CPF_Parm) && Property-
>HasAllPropertyFlags(CPF_BlueprintAssignable) && !Property-
>HasMetaData(HideInDetailPanelName))
    {}
}

void FBlueprintDetails::AddEventsCategory(IDetailLayoutBuilder& DetailBuilder,
FName PropertyName, UClass* PropertyClass)
{
    static const FName HideInDetailPanelName("HideInDetailPanel");
// Check for multicast delegates that we can safely assign
if ( !Property->HasAnyPropertyFlags(CPF_Parm) && Property-
>HasAllPropertyFlags(CPF_BlueprintAssignable) &&
    !Property->HasMetaData(HideInDetailPanelName) )
}

```

IgnoreCategoryKeywordsInSubclasses

- 功能描述:** 用于让一个类的首个子类忽略所有继承的 ShowCategories 和 HideCategories 说明符。
- 使用位置:** UClass
- 引擎模块:** DetailsPanel
- 元数据类型:** bool
Related To UClass: ComponentWrapperClass
(../../Specifier/UCLASS/ComponentWrapperClass.md)
- 常用程度:** ★

和ComponentWrapperClass相互关联

InlineEditConditionToggle

- 功能描述:** 使这个bool属性在被用作EditCondition的时候内联到对方的属性行里成为一个单选框，而不是自己成为一个编辑行。
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 限制类型:** bool
- 关联项:** EditCondition
- 常用程度:** ★★★★☆

使这个bool属性在被用作EditCondition的时候内联到对方的属性行里成为一个单选框，而不是自己成为一个编辑行。

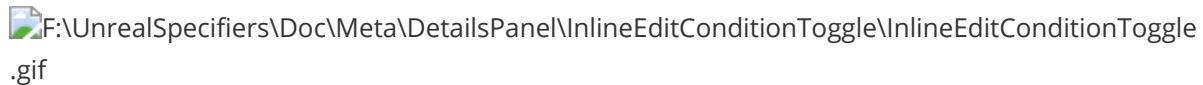
虽然EditCondition支持别的类型属性或者是表达式，但是这个InlineEditConditionToggle只支持bool属性。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (InlineEditConditionToggle))  
    bool MyBool_Inline;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (EditCondition = "MyBool_Inline"))  
    int32 MyInt_EditCondition_UseInline = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle)  
    int32 MyThirdInt_Inline = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
    InlineEditConditionToggle, meta = (EditCondition = "MyThirdInt_Inline>200"))  
    int32 MyInt_EditConditionExpression_UseInline = 123;
```

测试效果：

可见MyBool_Inline变成了单选框。而MyThirdInt_Inline就没有被隐藏掉。



原理：

可以看到用这个判断是否支持出现单选框。

```
bool FPropertyNode::SupportsEditConditionToggle() const  
{  
    if (!Property.IsValid())  
    {  
        return false;  
    }  
  
    FProperty* MyProperty = Property.Get();  
  
    static const FName Name_HideEditConditionToggle("HideEditConditionToggle");  
    if (EditConditionExpression.IsValid() && !Property->HasMetaData(Name_HideEditConditionToggle))  
    {  
        const FBoolProperty* ConditionalProperty = EditConditionContext->GetSingleBoolProperty(EditConditionExpression);  
        if (ConditionalProperty != nullptr)  
        {  
            // There are 2 valid states for inline edit conditions:  
            // 1. The property is marked as editable and has  
            InlineEditConditionToggle set.  
            // 2. The property is not marked as editable and does not have  
            InlineEditConditionToggle set.  
        }  
    }  
}
```

```

        // In both cases, the original property will be hidden and only show
        up as a toggle.

        static const FName
Name_InlineEditConditionToggle("InlineEditConditionToggle");
        const bool bIsInlineEditCondition = ConditionalProperty-
>HasMetaData(Name_InlineEditConditionToggle);
        const bool bIsEditable = ConditionalProperty-
>HasAllPropertyFlags(CPF_Edit);

        if (bIsInlineEditCondition == bIsEditable)
{
    return true;
}

        if (bIsInlineEditCondition && !bIsEditable)
{
    UE_LOG(LogPropertyName, Warning, TEXT("Property being used as
inline edit condition is not editable, but has redundant
InlineEditConditionToggle flag. Field \\\"%s\\\" in class \\\"%s\\\"."),
*ConditionalProperty->GetNameCPP(), *Property->GetOwnerStruct()->GetName());
    return true;
}

        // The property is already shown, and not marked as inline edit
condition.
        if (!bIsInlineEditCondition && bIsEditable)
{
    return false;
}
}

return false;
}

```

MaxPropertyDepth

- 功能描述:** 指定对象或结构在细节面板里展开的层数。
- 使用位置:** UPROPERTY
- 引擎模块:** DetailsPanel
- 元数据类型:** int32
- 限制类型:** 对象或结构属性
- 常用程度:** ★

指定对象或结构在细节面板里展开的层数。

- 默认是没有限制的，可以一直递归展开到最深层次字段。
- 如果对象的子对象再有子对象，这样递归很多层级，可能我们会想要限制不想展开太深，因此我们可以指定一个层级限制。
- 取值-1表示没有限制，0表示完全不展开，>0表示限制的层数。
- 源码里没有找到例子，但却是可以工作的。

测试代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth1
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt1 = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString1;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth2
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStructDepth1 MyStruct1;
};

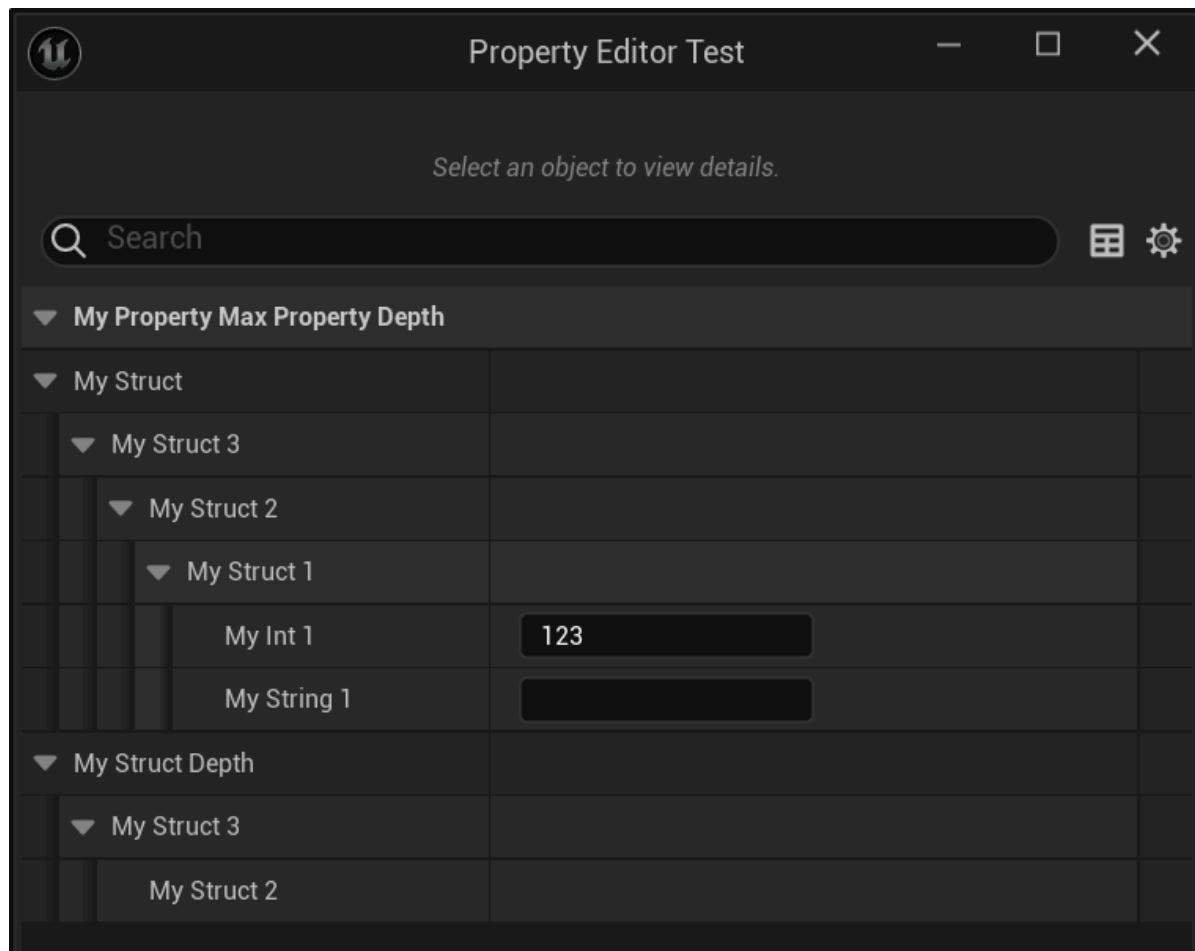
USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth3
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStructDepth2 MyStruct2;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth4
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStructDepth3 MyStruct3;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_MaxPropertyDepth :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FMyStructDepth4 MyStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(MaxPropertyDepth=2))
    FMyStructDepth4 MyStruct_Depth;
};
```

测试效果：



原理：

在每个FPropertyNode构建子节点的时候，检查一下当前的MaxChildDepthAllowed，超过了就不继续往下构建。

```
/** Safety value representing Depth in the property tree used to stop diabolical
 * topology cases
 * -1 = No limit on children
 * 0 = No more children are allowed. Do not process child nodes
 * >0 = A limit has been set by the property and will tick down for successive
 * children
 */
int32 MaxChildDepthAllowed;

void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    //Get the property max child depth
    static const FName Name_MaxPropertyDepth("MaxPropertyDepth");
    if (Property->HasMetaData(Name_MaxPropertyDepth))
    {
        int32 NewMaxChildDepthAllowed = Property-
>GetIntMetaData(Name_MaxPropertyDepth);
        //Ensure new depth is valid. Otherwise just let the parent specified
        //value stand
        if (NewMaxChildDepthAllowed > 0)
        {
            MaxChildDepthAllowed = NewMaxChildDepthAllowed;
        }
    }
}
```

```

        //if there is already a limit on the depth allowed, take the
        minimum of the allowable depths
        if (MaxChildDepthAllowed >= 0)
        {
            MaxChildDepthAllowed = FMath::Min(MaxChildDepthAllowed,
NewMaxChildDepthAllowed);
        }
        else
        {
            //no current limit, go ahead and take the new limit
            MaxChildDepthAllowed = NewMaxChildDepthAllowed;
        }
    }
}

void FPropertyNode::RebuildChildren()
{
    if (MaxChildDepthAllowed != 0)
    {
        //the case where we don't want init child nodes is when an Item has children
        //that we don't want to display
        //the other option would be to make each node "Read only" under that item.
        //The example is a material assigned to a static mesh.
        if (HasNodeFlags(EPropertyNodeFlags::CanBeExpanded) && (childNodes.Num() ==
0))
        {
            InitChildNodes();
            if (ExpandedPropertyItemSet.size() > 0)
            {
                FPropertyNodeUtils::SetExpandedItems(ThisAsSharedRef,
ExpandedPropertyItemSet);
            }
        }
    }
}

```

NoEditInline

- 功能描述:** Object properties pointing to an UObject instance whose class is marked editinline will not show their properties inline in property windows. Useful for getting actor components to appear in the component tree but not inline in the root actor details panel.
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 限制类型:** UObject*
- 关联项:** EditInline (EditInline.md)

对象引用默认就不能EditInline，因此也不需要额外再加上这个。除非Instanced之后？

结构属性默认就可以EditInline，加上这个后也没有作用，因此也不需要加上这个。

在源码中只找到：

```
UPROPERTY(VisibleAnywhere, Category = "Connection Point", meta =
(NoEditInline))
FLinearColor color = FLinearColor::Black;
```

NoResetToDefault

- **功能描述:** 禁用和隐藏属性在细节面板上的“重置”功能。
- **使用位置:** UPROPERTY
- **引擎模块:** DetailsPanel
- **元数据类型:** bool
- **常用程度:** ★★★

禁用和隐藏属性在细节面板上的“重置”功能。

测试代码:

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=ResetToDefaultTest)
    int32 MyInt_Default = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=ResetToDefaultTest, meta
    = (NoResetToDefault))
    int32 MyInt_NoResetToDefault = 123;
```

测试效果:

可以发现默认的属性在改变值后，右侧会出现一个重置按钮，以便让属性重置回默认值。
NoResetToDefault的作用就是去除这个功能。



原理:

编辑器里会判断这个meta，如果没有则创建SResetToDefaultPropertyEditor。

```
bool SsingleProperty::GeneratePropertyCustomization()
{
    if (!PropertyEditor->GetPropertyHandle()-
>HasMetaData(TEXT("NoResetToDefault")) && !bShouldHideResetToDefault)
    {
        HorizontalBox->AddSlot()
        .Padding( 2.0f )
        .Autowidth()
        .VAlign( VAlign_Center )
        [
            SNew( SResetToDefaultPropertyEditor, PropertyEditor-
>GetPropertyHandle() )
        ];
    }
}
```

PrioritizeCategories

- **功能描述:** 把指定的属性目录优先显示在前面
- **使用位置:** UCLASS
- **引擎模块:** DetailsPanel
- **元数据类型:** strings="a, b, c"
- **关联项:**
UCLASS: PrioritizeCategories
- **常用程度:** ★★★

ReapplyCondition

- **功能描述:** // Properties that have a ReapplyCondition should be disabled behind the specified property when in reapply mode
- **使用位置:** UPROPERTY
- **引擎模块:** DetailsPanel
- **元数据类型:** string="abc"
- **常用程度:** ★

代码:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Placement, meta=(UIMin = 0, ClampMin = 0, UIMax = 359, ClampMax = 359,  
ReapplyCondition="ReapplyRandomPitchAngle"))  
float RandomPitchAngle;
```

也只在Foliage中用到。

ShowCategories

- **功能描述:** 显示类别
- **使用位置:** UCLASS
- **元数据类型:** strings="a, b, c"
- **关联项:** HideCategories

在类上面标记的ShowCategories，并不会保存到meta中去，只是用来抹除基类HideCategories的设置。因此meta里的ShowCategories是没有用到的。

```

//(BlueprintType = true, IncludePath = Class/Display/MyClass_HideCategories.h,
IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_HideCategories.h)
UCLASS(Blueprintable, ShowCategories = MyGroup1)
class INSIDER_API UMyClass_ShowCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | "
MyGroup2")
        int Property_Group222;
};

```

UsesHierarchy

- 功能描述:** 说明类使用层级数据。用于实例化“细节”面板中的层级编辑功能。
- 使用位置:** UCLASS
- 引擎模块:** DetailsPanel
- 元数据类型:** bool
- 常用程度:** 0

Comment

- 功能描述:** 用来记录注释的内容
- 使用位置:** Any
- 引擎模块:** Development
- 元数据类型:** string="abc"
- 常用程度:** ★★★

Comment跟ToolTip不同，后者是用户鼠标悬停上的提示，前者只是简单的代码中的注释的记录。但是
一般我们在代码里写上的注释，会自动也加到ToolTip 上，因此我们往往也会看到UI界面上的提示。

但如果不需要ToolTip，想只有Comment，则也可以自己手动在meta里添加。

测试代码：

```

//(BlueprintType = true, Comment = //This is a comment on class, IncludePath =
Property/Development/MyProperty_Development.h, ModuleRelativePath =
Property/Development/MyProperty_Development.h, ToolTip = This is a comment on
class)

//This is a comment on class
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Development :public UObject
{
    GENERATED_BODY()

```

```

public:
    // (Comment = //This is a comment on function, ModuleRelativePath =
    Property/Development/MyProperty_Development.h, ToolTip = This is a comment on
    function)

    //This is a comment on function
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}

    // (Category = MyProperty_Development, Comment = //This is a comment on
    property, ModuleRelativePath = Property/Development/MyProperty_Development.h,
    ToolTip = This is a comment on property)

    //This is a comment on property
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;

    // (Category = MyProperty_Development, Comment = This is my other property.,
    ModuleRelativePath = Property/Development/MyProperty_Development.h)

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(Comment="This is my other
    property."))
    int32 MyProperty_WithComment = 123;
};
```

测试结果：

MyProperty_WithComment是单独只加的Comment，就没有了鼠标悬停效果。



Deprecated

- 功能描述：**指定该元素要废弃的引擎版本号。
- 使用位置：**Any
- 引擎模块：**Development
- 元数据类型：**string="abc"
- 常用程度：**★

指定该元素要废弃的引擎版本号。

这个值只是单纯的在C++代码中记录一下信息，并不会真正的使得一个元素变成废弃。这个值也没有在别的地方UI使用和显示出来。

要废弃一个元素，还是要用别的标记，如**DeprecatedProperty**, **DeprecatedFunction**等。

DeprecatedFunction

- 功能描述：**标明一个函数已经被弃用
- 使用位置：**UFUNCTION
- 引擎模块：**Development

- 元数据类型: bool

- 常用程度: ★

Any Blueprint references to this function will cause compilation warnings telling the user that the function is deprecated. You can add to the deprecation warning message (for example, to provide instructions on replacing the deprecated function) using the DeprecationMessage metadata specifier.

DeprecatedProperty

- 功能描述: 标记弃用, 引用到该属性的蓝图会触发一个警告

- 使用位置: UPROPERTY

- 引擎模块: Development

- 元数据类型: bool

- 关联项:

UCLASS: Deprecated

- 常用程度: ★

标记弃用, 引用到该属性的蓝图会触发一个警告

示例代码:

```
// Simple
UPROPERTY(BlueprintReadWrite, meta=(DeprecatedProperty, DeprecationMessage="This
is deprecated"))
FString PlantName;

// Better
UPROPERTY(BlueprintReadWrite, meta=(DisplayName="PlantName", DeprecatedProperty,
DeprecationMessage="PlantName is deprecated, instead use PlantDisplayName."))
FString DEPRECATED_PlantName;
```

DeprecationMessage

- 功能描述: 定义弃用的消息

- 使用位置: UCLASS, UFUNCTION, UPROPERTY

- 引擎模块: Development

- 元数据类型: string="abc"

- 关联项:

UCLASS: Deprecated

- 常用程度: ★

例子：

```
UFUNCTION(meta=(DeprecatedFunction,DeprecationMessage="This function is  
deprecated, please use OtherFunctionName instead."))  
ReturnType FunctionName([Parameter, Parameter, ...])  
  
UPROPERTY(BlueprintReadWrite, meta=(DeprecatedProperty, DeprecationMessage="This  
is deprecated"))  
FString PlantName;
```

DevelopmentOnly

- 功能描述：**使得一个函数变为DevelopmentOnly，意味着只会在Development模式中运行。适用于调试输出之类的功能，但在最终发布版中会跳过。
- 使用位置：** UFUNCTION
- 引擎模块：** Development
- 元数据类型：** bool
- 常用程度：** ★

使得一个函数变为DevelopmentOnly，意味着只会在Development模式中运行。适用于调试输出之类的功能，但在最终发布版中会跳过。

源码中最典型的例子就是PrintString。

测试代码：

```
UFUNCTION(BlueprintCallable,meta=(Developmentonly))  
static void MyFunc_DevelopmentOnly(){}  
  
UFUNCTION(BlueprintCallable,meta=())  
static void MyFunc_NotDevelopmentOnly(){}  
 
```

蓝图效果：

原理：

其会改变这个函数蓝图节点的状态为DevelopmentOnly，从而最终导致该node在shipping模式下被pass through。

```
void UK2Node_CallFunction::Serialize(FArchive& Ar)  
{  
    if (const UFunction* Function = GetTargetFunction())  
    {  
        // Enable as development-only if specified in metadata. This  
        // way existing functions that have the metadata added to them will get their  
        // enabled state fixed up on load.  
        if (GetDesiredEnabledState() == ENodeEnabledState::Enabled &&  
            Function->HasMetaData(FBlueprintMetadata::MD_Developmentonly))  
        {  
 
```

```

        SetEnabledState(ENodeEnabledState::DevelopmentOnly,
/*buserAction=*/ false);
    }
        // Ensure that if the metadata is removed, we also fix up the
enabled state to avoid leaving it set as development-only in that case.
    else if (GetDesiredEnabledState() ==
ENodeEnabledState::DevelopmentOnly && !Function-
>HasMetaData(FBlueprintMetadata::MD_DevelopmentOnly))
{
    SetEnabledState(ENodeEnabledState::Enabled,
/*buserAction=*/ false);
}
}

```

DevelopmentStatus

- 功能描述:** 标明开发状态
- 使用位置:** UCLASS
- 引擎模块:** Development
- 元数据类型:** string="abc"
- 关联项:**
UCLASS: Experimental, EarlyAccessPreview
- 常用程度:** ★

DevelopmentStatus=Experimental
DevelopmentStatus=EarlyAccess

FriendlyName

- 功能描述:** 和DisplayName一样?
- 使用位置:** Any
- 引擎模块:** Development
- 元数据类型:** string="abc"

ShortTooltip

- 功能描述:** 提供一个更简洁版本的提示文本，例如在类型选择器的时候显示
- 使用位置:** Any
- 元数据类型:** string="abc"
- 关联项:** ToolTip

ToolTip

- 功能描述:** 在Meta里提供一个提示文本，覆盖代码注释里的文本
- 使用位置:** Any
- 引擎模块:** Development

- 元数据类型: string="abc"

- 关联项: ShortTooltip

- 常用程度: ★★★

测试代码:

```
// This is a ToolTip out of Class. There're so so so so so so so many words I want
to say, but here's too narrow.
UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within
Class. There're so so so so so so so many words I want to say, but here's too
narrow."))
class INSIDER_API UMyClass_ToolTip :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a
ToolTip within Property. There're so so so so so so so many words I want to say,
but here's too narrow."))
    float MyFloat_WithToolTip;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;

    UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within
Function. There're so so so so so so so many words I want to say, but here's too
narrow."))
    void MyFunc_WithToolTip() {}

    UFUNCTION(BlueprintCallable)
    void MyFunc() {}
};

// This is a ToolTip out of Class. There're so so so so so so so many words I want
to say, but here's too narrow.
UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within
Class. There're so so so so so so so many words I want to say, but here's too
narrow.", ShortToolTip = "This is a ShortToolTip within Class."))
class INSIDER_API UMyClass_WithAllToolTip :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a
ToolTip within Property."))
    float MyFloat_WithToolTip;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a ToolTip
within Property. There're so so so so so so so many words I want to say, but
here's too narrow.\nThis is a new line.", ShortToolTip = "This is a ShortToolTip
within Property."))
    float MyFloat_WithAllToolTip;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
```

```

UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within
Function. There're so so so so so so so many words I want to say, but here's too
narrow.",ShortToolTip = "This is a ShortToolTip within Function."))
void MyFunc_WithAllToolTip() {}

UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within
Function."))
void MyFunc_WithToolTip() {}
};

// This is a ToolTip out of Class. There're so so so so so so so many words I want
// to say, but here's too narrow.
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeA :public UObject
{
    GENERATED_BODY()
};

/***
 *  This is a ToolTip out of Class.
 *  There're so so so so so so so many words I want to say, but here's too
narrow.
 *  Please read this tooltip before you use this class.
 */
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeB :public UObject
{
    GENERATED_BODY()
};

```

测试效果：

在选取父类时候的提示：

可以发现，如果提供了ToolTip，就会覆盖掉代码注释里的注释。同时也可以发现在下图中，提示的文本过长超过了选项框。这时如果提供了ShortToolTip，则会在父类选取器中显示ShortToolTip中的文本，从而简洁一点显示。在很多别的地方也同样应用这个规则，比如（该类型）变量上的提示，或者在选择变量类型的时候。

原理代码：

在源码里有FField和UField，普通的属性是FField，而像UClass是继承于UField，因此要注意
FField::GetToolTipText的bShortTooltip一直是false，而UField::GetToolTipText则会传true。

```

FText FField::GetToolTipText(bool bShortTooltip) const
{
    bool bFoundShortTooltip = false;
    static const FName NAME_Tooltip(TEXT("Tooltip"));
    static const FName NAME_ShortTooltip(TEXT("ShortTooltip"));
    FText LocalizedToolTip;
    FString NativeToolTip;

    if (bshortTooltip)

```

```

{
    NativeToolTip = GetMetaData(NAME_ShortTooltip);
    if (NativeToolTip.IsEmpty())
    {
        NativeToolTip = GetMetaData(NAME_Tooltip);
    }
    else
    {
        bFoundShortTooltip = true;
    }
}
else
{
    NativeToolTip = GetMetaData(NAME_Tooltip);
}

const FString Namespace = bFoundShortTooltip ? TEXT("UObjectShortTooltips") :
TEXT("UObjectToolTips");
const FString Key = GetFullGroupName(false);
if (!FText::FindText(Namespace, Key, /*OUT*/&LocalizedToolTip,
&NativeToolTip))
{
    if (!NativeToolTip.IsEmpty())
    {
        static const FString DoxygenSee(TEXT("@see"));
        static const FString TooltipSee(TEXT("See:"));
        if (NativeToolTip.ReplaceInline(*DoxygenSee, *TooltipSee) > 0)
        {
            NativeToolTip.TrimEndInline();
        }
    }
    LocalizedToolTip = FText::FromString(NativeToolTip);
}

return LocalizedToolTip;
}

FText UField::GetToolTipText(bool bShortTooltip) const
{
    bool bFoundShortTooltip = false;
    static const FName NAME_Tooltip(TEXT("Tooltip"));
    static const FName NAME_ShortTooltip(TEXT("ShortTooltip"));
    FText LocalizedToolTip;
    FString NativeToolTip;

    if (bShortTooltip)
    {
        NativeToolTip = GetMetaData(NAME_ShortTooltip);
        if (NativeToolTip.IsEmpty())
        {
            NativeToolTip = GetMetaData(NAME_Tooltip);
        }
        else
        {
            bFoundShortTooltip = true;
        }
    }
}

```

```

    }

    else
    {
        NativeToolTip = GetMetaData(NAME_Tooltip);
    }

    const FString Namespace = bFoundShortTooltip ? TEXT("UObjectShortTooltips") :
TEXT("UObjectToolTips");
    const FString Key = GetFullGroupName(false);
    if ( !FText::FindText( Namespace, Key, /*OUT*/LocalizedToolTip,
&NativeToolTip ) )
    {
        if (NativeToolTip.IsEmpty())
        {
            NativeToolTip =
FName::NameToDisplayString(FDisplayNameHelper::Get(*this), false);
        }
        else if (!bShortTooltip && IsNative())
        {
            FormatNativeToolTip(NativeToolTip, true);
        }
        LocalizedToolTip = FText::FromString(NativeToolTip);
    }

    return LocalizedToolTip;
}

//在类型选择器中优先选择ShortTooltip
FText FClassPickerDefaults::GetDescription() const
{
    FText Result = LOCTEXT("NullClass", "(null class)");

    if (UClass* ItemClass = LoadClass<UObject>(NULL, *ClassName, NULL, LOAD_None,
NULL))
    {
        Result = ItemClass->GetToolTipText(/*bShortTooltip= */ true);
    }

    return Result;
}

```

但对于Property和Function，在显示的时候，都只会显示ToolTip，并不会应用ShortToolTip

变量和函数：



其他需要注意的是，代码里注释的文本也会当作ToolTip。支持//和/**/这两种格式。如果在ToolTip中想换行，可以直接加/n就可以。

```

/*
(BlueprintType = true, Comment = // This is a Tooltip out of class. There're so so
so so so so many words I want to say, but here's too narrow.

```

```

, IncludePath = Any/ToolTip_Test.h, IsBlueprintBase = true, ModuleRelativePath =
Any/ToolTip_Test.h, ToolTip = This is a ToolTip out of Class. There're so so so so
so so so many words I want to say, but here's too narrow.)
*/



// This is a ToolTip out of Class. There're so so so so so so so many words I want
to say, but here's too narrow.
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeA :public UObject
{
    GENERATED_BODY()
};

// [MyClass_ToolTip_TypeB   Class->Struct->Field->Object
/Script/Insider.MyClass_ToolTip_TypeB]
//(BlueprintType = true, Comment = /**
/* This is a ToolTip out of Class.
/* There're so so so so so so so many words I want to say, but here's too
narrow.
/* Please read this tooltip before you use this class.
// */, IncludePath = Any/ToolTip_Test.h, IsBlueprintBase = true,
ModuleRelativePath = Any/ToolTip_Test.h, ToolTip = This is a ToolTip out of
class.
//There're so so so so so so so many words I want to say, but here's too narrow.
//Please read this tooltip before you use this class.)


/**
*   This is a ToolTip out of Class.
*   There're so so so so so so so many words I want to say, but here's too
narrow.
*   Please read this tooltip before you use this class.
*/
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeB :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within
Class. There're so so so so so so so many words I want to say, but here's too
narrow.\nThis is a new Line.", ShortToolTip = "This is a ShortToolTip within
Class."))

```

Bitflags

- 功能描述:** 设定一个枚举支持采用位标记赋值，从而在蓝图中可以识别出来是BitMask
- 使用位置:** UENUM
- 引擎模块:** Enum Property
- 元数据类型:** bool
- 关联项:** UseEnumValuesAsMaskValuesInEditor
- 常用程度:** ★★★★☆

常常和UPROPERTY上的bitmask一起配合使用。

注意这个和UENUM(flags)的区别，后者是影响C++里字符串输出函数。

这个是指定该枚举支持位标记，从而在蓝图中可以被选择出来。

```
UENUM(BlueprintType, Flags)
enum class EMyEnum_Flags:uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType, Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First,
    Second,
    Third,
};

//源码中的例子:
UENUM(Meta = (Bitflags))
enum class EColorBits
{
    ECB_Red,
    ECB_Green,
    ECB_Blue
};
UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EColorBits"))
int32 ColorFlags;
```

如下图所示：EMyEnum_Flags就不会被列在选项里。而EMyEnum_BitFlags就可以被列进来。

如果没有UPROPERTY(bitmask)的配合使用，则蓝图里还是只能单项选择

Bitmask

- **功能描述：**设定一个属性采用Bitmask赋值
- **使用位置：** UPROPERTY
- **引擎模块：** Enum Property
- **元数据类型：** bool
- **限制类型：** 用来表示枚举值的int32
- **关联项：** BitmaskEnum
- **常用程度：** ★★★★☆

这个标记和enum身上的定义并没有一定的关系，因此可以单独定义。

```
UENUM(BlueprintType)
enum class EMyEnum_Normal:uint8
{
```

```

        First,
        Second,
        Third,
    };

UENUM(BlueprintType,Flags)
enum class EMyEnum_Flags:uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType,Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First,
    Second,
    Third,
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_EnumBitFlags_Test:public AActor
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_Normal"))
    int32 MyNormal;

    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_Flags"))
    int32 MyFlags;

    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_BitFlags"))
    int32 MyBitFlags;
};

```

都是可以在蓝图中用标记来定义

可以用BitmaskEnum进一步提供枚举值

BitmaskEnum

- 功能描述:** 使用位标记后采用的枚举名字
- 使用位置:** UPROPERTY
- 元数据类型:** string="abc"
- 限制类型:** 用来表示枚举值的int32
- 关联项:** Bitmask
- 常用程度:** ★★★★☆

如果没有标上BitmaskEnum，则无法提供标记的的名称值

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_EnumBitFlags_Test : public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, Meta = (Bitmask))
    int32 MyNormalWithoutEnum;
};

```

如果没有标上BitmaskEnum，则无法提供标记的的名称值

Enum

- 功能描述:** 给一个String指定以枚举里值的名称作为选项
- 使用位置:** UPROPERTY
- 引擎模块:** Enum Property
- 元数据类型:** string="abc"
- 限制类型:** FString
- 关联项:** ValidEnumValues
- 常用程度:** ★★★

EnumDisplayNameFn

- 功能描述:** 在Runtime下为枚举字段提供自定义名称的函数回调
- 使用位置:** UENUM
- 引擎模块:** Enum Property
- 元数据类型:** string="abc"
- 常用程度:** ★★

只在Runtime下生效，在Editor下依然不起作用。

测试代码：

```

// [EMyEnum_CustomDisplay   Enum->Field->Object
/Script/Insider.EMyEnum_CustomDisplay]
//(BlueprintType = true, EnumDisplayNameFn = GetMyEnumCustomDisplayName,
First.Name = EMyEnum_CustomDisplay::First, IsBlueprintBase = true,
ModuleRelativePath = Enum/MyEnum_Test.h, Second.Name =
EMyEnum_CustomDisplay::Second, Third.Name = EMyEnum_CustomDisplay::Third)
// ObjectFlags: RF_Public | RF_Transient
// Outer: Package /Script/Insider
// EnumFlags: EEnumFlags::None
// EnumDisplayNameFn: 6adb4804
// CppType: EMyEnum_CustomDisplay
// CppForm: EnumClass
//{
// First = 0,
// Second = 1,

```

```

// Third = 2,
// EMyEnum_MAX = 3
//};

UENUM(Blueprintable, meta = (EnumDisplayNameFn = "GetMyEnumCustomDisplayName"))
enum class EMyEnum_CustomDisplay :uint8
{
    First,
    Second,
    Third,
};

extern FText GetMyEnumCustomDisplayName(int32 val);

FText GetMyEnumCustomDisplayName(int32 val)
{
    EMyEnum_CustomDisplay enumValue = (EMyEnum_CustomDisplay)val;
    switch (enumValue)
    {
        case EMyEnum_CustomDisplay::First:
            return FText::FromString(TEXT("My_First"));
        case EMyEnum_CustomDisplay::Second:
            return FText::FromString(TEXT("My_Second"));
        case EMyEnum_CustomDisplay::Third:
            return FText::FromString(TEXT("My_Third"));
        default:
            return FText::FromString(TEXT("Invalid MyEnum"));
    }
}

```

测试蓝图：

EnumDisplayNameFn 的函数设置是在gen.cpp中完成的，因此并不需要成为UFUNCTION。

```

const UECodeGen_Private::FEnumParams
Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::EnumParams = {
    (UObject*)(*)() Z_Construct_UPackage_Script_Insider,
    GetMyEnumCustomDisplayName, //这里！！！
    "EMyEnum_CustomDisplay",
    "EMyEnum_CustomDisplay",
    Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enumerators,
    RF_Public|RF_Transient|RF_MarkAsNative,

    UE_ARRAY_COUNT(Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enumerators),
    EEnumFlags::None,
    (uint8)UEnum::ECppForm::EnumClass,

    METADATA_PARAMS(UE_ARRAY_COUNT(Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enum_MetaDataParams),
    Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enum_MetaDataParams)
};

```

原理代码：

```
/FText UEnum::GetDisplayNameTextByIndex(int32 NameIndex) const
{
    FString RawName = GetNameStringByIndex(NameIndex);

    if (RawName.IsEmpty())
    {
        return FText::GetEmpty();
    }

#if WITH_EDITOR
    FText LocalizedDisplayName;
    // In the editor, use metadata and localization to look up names
    static const FString Namespace = TEXT("UObjectDisplayNames");
    const FString Key = GetFullGroupName(false) + TEXT(".") + RawName;

    FString NativeDisplayName;
    if (HasMetaData(TEXT("DisplayName"), NameIndex))
    {
        NativeDisplayName = GetMetaData(TEXT("DisplayName"), NameIndex);
    }
    else
    {
        NativeDisplayName = FName::NameToDisplayString(RawName, false);
    }

    if (!(FText::FindText(Namespace, Key, /*OUT*/&LocalizedDisplayName,
&NativeDisplayName)))
    {
        LocalizedDisplayName = FText::FromString(NativeDisplayName);
    }

    if (!LocalizedDisplayName.IsEmpty())
    {
        return LocalizedDisplayName;
    }
#endif
//Runtime下到这里
    if (EnumDisplayNameFn)
    {
        return (*EnumDisplayNameFn)(NameIndex);
    }

    return FText::FromString(GetNameStringByIndex(NameIndex));
}
```

EnumValueDisplayNameOverrides

- **功能描述：** 改变枚举属性值上的显示名字
- **使用位置：** UPROPERTY
- **引擎模块：** Enum Property

- **元数据类型:** string="abc"
- **关联项:** ValidEnumValues
- **常用程度:** ★★

给枚举属性上的枚举值进行一些改名，可以改变多个，按照“A=B;C=D”的格式列出即可。收集到的信息是TMap< FName, FText>映射，因此要同时提供原枚举值名称和新的显示名称配对。

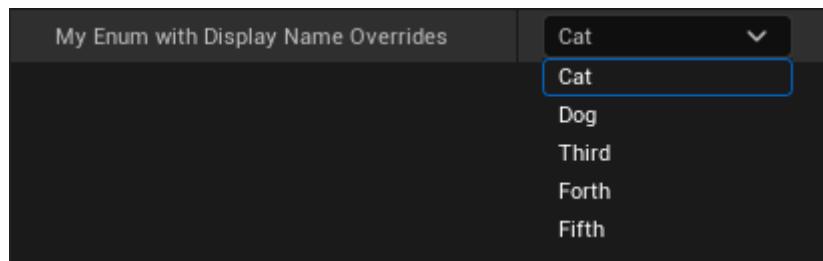
测试代码：

```
UENUM(BlueprintType)
enum class EMyPropertyTestEnum : uint8
{
    First,
    Second,
    Third,
    Forth,
    Fifth,
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EnumValueDisplayNameOverrides
= "First=Cat;Second=Dog"))
EMyPropertyTestEnum MyEnumwithDisplayNameOverrides;
```

蓝图效果：

可见实际上改变了First、Second的显示名字。



原理代码见ValidEnumValues上的代码

GetRestrictedEnumValues

- **功能描述:** 指定一个函数来指定枚举属性值的哪些枚举选项是禁用的
- **使用位置:** UPROPERTY
- **引擎模块:** Enum Property
- **元数据类型:** string="abc"
- **限制类型:** TArray FuncName() const;
- **关联项:** ValidEnumValues
- **常用程度:** ★★★

Restricted和Invalid的区别是：

Invalid会隐藏掉该选项值

Restricted依然会显示该选项值，只是会灰调不可选。

指定的函数名字必须是一个UFUNCTION函数，这样才能通过名字找到该函数。

InvalidEnumValues

- **功能描述:** 指定枚举属性值上不可选的枚举值选项，用以排除一些选项
- **使用位置:** UPROPERTY
- **引擎模块:** Enum Property
- **元数据类型:** strings="a, b, c"
- **限制类型:** 枚举属性值
- **关联项:** ValidEnumValues
- **常用程度:** ★★★

如果同时指定了InvalidEnumValues和ValidEnumValues，且里面的值有重叠，则还是以InvalidEnumValues的为准：这项枚举值就是非法的。

DisplayName

- **功能描述:** 改变枚举值的显示名称
- **使用位置:** UENUM::UMETA
- **引擎模块:** Enum Property
- **元数据类型:** string="abc"
- **常用程度:** ★★★★★

改变枚举值的显示名称

示例代码：

```
/*
[enum 602d0d4e680 EMyEnum_HasDisplayName      Enum->Field->Object
/Script/Insider.EMyEnum_HasDisplayName]
(BlueprintType = true, First.DisplayName = Dog, First.Name =
EMyEnum_HasDisplayName::First, IsBlueprintBase = true, ModuleRelativePath =
Enum/MyEnum_Test.h, Second.DisplayName = Cat, Second.Name =
EMyEnum_HasDisplayName::Second, Third.DisplayName = Pig, Third.Name =
EMyEnum_HasDisplayName::Third)
    ObjectFlags:    RF_Public | RF_Transient
    Outer:    Package /Script/Insider
    EnumFlags:    None
    EnumDisplayNameFn: 0
    CppType:    EMyEnum_HasDisplayName
    CppForm:    EnumClass
{
    First = 0,
    Second = 1,
    Third = 2,
    EMyEnum_MAX = 3
};
*/
UENUM(Blueprintable)
enum class EMyEnum_HasDisplayName : uint8
```

```

{
    First UMETA(DisplayName="Dog"),
    Second UMETA(DisplayName="Cat"),
    Third UMETA(DisplayName="Pig"),
};

UCLASS(BlueprintType)
class INSIDER_API UMyEnum_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EMyEnum_HasDisplayName MyEnum_HasDisplayName;
}

```

示例效果：

可见改变了名称。



DisplayValue

- 功能描述:** Enum /Script/Engine.AnimPhysCollisionType
- 使用位置:** UENUM::UMETA
- 引擎模块:** Enum Property
- 常用程度:** 0

源码例子：

```

UENUM()
enum class AnimPhysCollisionType : uint8
{
    CoM UMETA(DisplayName="CoM", DisplayValue="CoM", ToolTip="Only limit the
center of mass from crossing planes."),
    CustomSphere UMETA(ToolTip="Use the specified sphere radius to collide with
planes."),
    Innersphere UMETA(ToolTip="Use the largest sphere that fits entirely within
the body extents to collide with planes."),
    OuterSphere UMETA(ToolTip="Use the smallest sphere that wholly contains the
body extents to collide with planes.")
};

```

Grouping

- 功能描述:** Enum /Script/Engine.EAlphaBlendOption
- 使用位置:** UENUM::UMETA
- 引擎模块:** Enum Property
- 元数据类型:** bool
- 常用程度:** 0

感觉是用在Sequencer里面的，只用在SEasingFunctionGridWidget里面。

源码例子：

```
UENUM()
enum class EAlphaBlendOption : uint8
{
    Linear = 0 UMETA(Grouping = Linear, DisplayName = "Linear", ToolTip = "Linear
interpolation"),
    Cubic UMETA(Grouping = Cubic, DisplayName = "Cubic In", ToolTip = "Cubic-in
interpolation"),
    HermiteCubic UMETA(Grouping = Cubic, DisplayName = "Hermite-Cubic InOut",
ToolTip = "Hermite-Cubic"),
    Sinusoidal UMETA(Grouping = Sinusoidal, DisplayName = "Sinusoidal", ToolTip =
"Sinusoidal interpolation"),
    QuadraticInOut UMETA(Grouping = Quadratic, DisplayName = "Quadratic InOut",
ToolTip = "Quadratic in-out interpolation"),
    CubicInOut UMETA(Grouping = Cubic, DisplayName = "Cubic InOut", ToolTip =
"Cubic in-out interpolation"),
    QuarticInOut UMETA(Grouping = Quartic, DisplayName = "Quartic InOut", ToolTip
= "Quartic in-out interpolation"),
    QuinticInOut UMETA(Grouping = Quintic, DisplayName = "Quintic InOut", ToolTip
= "Quintic in-out interpolation"),
    CircularIn UMETA(Grouping = Circular, DisplayName = "Circular In", ToolTip =
"Circular-in interpolation"),
    CircularOut UMETA(Grouping = Circular, DisplayName = "Circular Out", ToolTip
= "Circular-out interpolation"),
    CircularInOut UMETA(Grouping = Circular, DisplayName = "Circular InOut",
ToolTip = "Circular in-out interpolation"),
    ExpIn UMETA(Grouping = Exponential, DisplayName = "Exponential In", ToolTip =
"Exponential-in interpolation"),
    ExpOut UMETA(Grouping = Exponential, DisplayName = "Exponential Out", ToolTip
= "Exponential-Out interpolation"),
    ExpInOut UMETA(Grouping = Exponential, DisplayName = "Exponential InOut",
ToolTip = "Exponential in-out interpolation"),
    Custom UMETA(Grouping = Custom, DisplayName = "Custom", ToolTip = "Custom
interpolation, will use custom curve inside an FAlphaBlend or linear if none has
been set"),
};

UENUM()
enum class EMovieSceneBuiltInEasing : uint8
{
    // Linear easing
    Linear UMETA(Grouping=Linear,DisplayName="Linear"),
    // Sinusoidal easing
    SinIn UMETA(Grouping=Sinusoidal,DisplayName="Sinusoidal In"), SinOut
UMETA(Grouping=Sinusoidal,DisplayName="Sinusoidal Out"), SinInOut
UMETA(Grouping=Sinusoidal,DisplayName="sinusoidal InOut"),
    // Quadratic easing
    QuadIn UMETA(Grouping=Quadratic,DisplayName="Quadratic In"), QuadOut
UMETA(Grouping=Quadratic,DisplayName="Quadratic Out"), QuadInOut
UMETA(Grouping=Quadratic,DisplayName="Quadratic InOut"),
    // Cubic easing
    CubicIn UMETA(Grouping=Cubic,DisplayName="Cubic In"), CubicOut
UMETA(Grouping=Cubic,DisplayName="Cubic Out"), CubicInOut
UMETA(Grouping=Cubic,DisplayName="Cubic InOut"),
    // Sine easing
    SineIn UMETA(Grouping=Sine,DisplayName="Sine In"), SineOut
UMETA(Grouping=Sine,DisplayName="Sine Out"), SineInOut
UMETA(Grouping=Sine,DisplayName="Sine InOut"),
    // Exponential easing
    ExpIn UMETA(Grouping=Exponential,DisplayName="Exponential In"), ExpOut
UMETA(Grouping=Exponential,DisplayName="Exponential Out"), ExpInOut
UMETA(Grouping=Exponential,DisplayName="Exponential InOut"),
    // Circular easing
    CircularIn UMETA(Grouping=Circular,DisplayName="Circular In"), CircularOut
UMETA(Grouping=Circular,DisplayName="Circular Out"), CircularInOut
UMETA(Grouping=Circular,DisplayName="Circular InOut"),
    // Elastic easing
    ElasticIn UMETA(Grouping=Elastic,DisplayName="Elastic In"), ElasticOut
UMETA(Grouping=Elastic,DisplayName="Elastic Out"), ElasticInOut
UMETA(Grouping=Elastic,DisplayName="Elastic InOut"),
    // Back easing
    BackIn UMETA(Grouping=Back,DisplayName="Back In"), BackOut
UMETA(Grouping=Back,DisplayName="Back Out"), BackInOut
UMETA(Grouping=Back,DisplayName="Back InOut"),
    // Bounce easing
    BounceIn UMETA(Grouping=Bounce,DisplayName="Bounce In"), BounceOut
UMETA(Grouping=Bounce,DisplayName="Bounce Out"), BounceInOut
UMETA(Grouping=Bounce,DisplayName="Bounce InOut"),
    // Ease-in easing
    EaseIn UMETA(Grouping=EaseIn,DisplayName="Ease In"), EaseOut
UMETA(Grouping=EaseIn,DisplayName="Ease Out"), EaseInOut
UMETA(Grouping=EaseIn,DisplayName="Ease InOut"),
    // Ease-out easing
    EaseOut UMETA(Grouping=EaseOut,DisplayName="Ease Out"), EaseIn
UMETA(Grouping=EaseOut,DisplayName="Ease In"), EaseInOut
UMETA(Grouping=EaseOut,DisplayName="Ease InOut"),
    // Ease-in-out easing
    EaseInOut UMETA(Grouping=EaseInOut,DisplayName="Ease InOut"),
};
```

```

        Cubic UMETA(Grouping = Cubic, DisplayName = "Cubic"), CubicIn
UMETA(Grouping=Cubic,DisplayName="Cubic In"), CubicOut
UMETA(Grouping=Cubic,DisplayName="Cubic Out"), CubicInOut
UMETA(Grouping=Cubic,DisplayName="Cubic InOut"), HermiteCubicInOut UMETA(Grouping
= Cubic, DisplayName = "Hermite-Cubic InOut"),
    // Quartic easing
    QuartIn UMETA(Grouping=Quartic,DisplayName="Quartic In"), QuartOut
UMETA(Grouping=Quartic,DisplayName="Quartic Out"), QuartInOut
UMETA(Grouping=Quartic,DisplayName="Quartic InOut"),
    // Quintic easing
    QuintIn UMETA(Grouping=Quintic,DisplayName="Quintic In"), QuintOut
UMETA(Grouping=Quintic,DisplayName="Quintic Out"), QuintInOut
UMETA(Grouping=Quintic,DisplayName="Quintic InOut"),
    // Exponential easing
    ExpoIn UMETA(Grouping=Exponential,DisplayName="Exponential In"), ExpoOut
UMETA(Grouping=Exponential,DisplayName="Exponential Out"), ExpoInOut
UMETA(Grouping=Exponential,DisplayName="Exponential InOut"),
    // Circular easing
    CircIn UMETA(Grouping=Circular,DisplayName="Circular In"), CircOut
UMETA(Grouping=Circular,DisplayName="Circular Out"), CircInOut
UMETA(Grouping=Circular,DisplayName="Circular InOut"),
    // Custom
    Custom UMETA(Grouping = Custom, DisplayName = "Custom"),
};


```

原理：

```

TArray<SEasingFunctionGridWidget::FGroup>
SEasingFunctionGridWidget::ConstructGroups(const TSet<EMovieSceneBuiltInEasing>&
FilterExclude)
{
    const UEnum* EasingEnum = StaticEnum<EMovieSceneBuiltInEasing>();
    check(EasingEnum)

    TArray<FGroup> Groups;

    for (int32 NameIndex = 0; NameIndex < EasingEnum->NumEnums() - 1;
++NameIndex)
    {
        const FString& Grouping = EasingEnum->GetMetaData(TEXT("Grouping"),
NameIndex);
        EMovieSceneBuiltInEasing Value = (EMovieSceneBuiltInEasing)EasingEnum-
>GetValueByIndex(NameIndex);

        if (FilterExclude.IsEmpty() || FilterExclude.Find(Value) == nullptr)
        {
            FindOrAddGroup(Groups, Grouping).Values.Add(Value);
        }
    }

    return Groups;
}

```

Hidden

- **功能描述:** 隐藏UENUM的某个值
- **使用位置:** UENUM::UMETA
- **引擎模块:** Enum Property
- **元数据类型:** bool
- **限制类型:** UENUM的值
- **常用程度:** ★★★★★

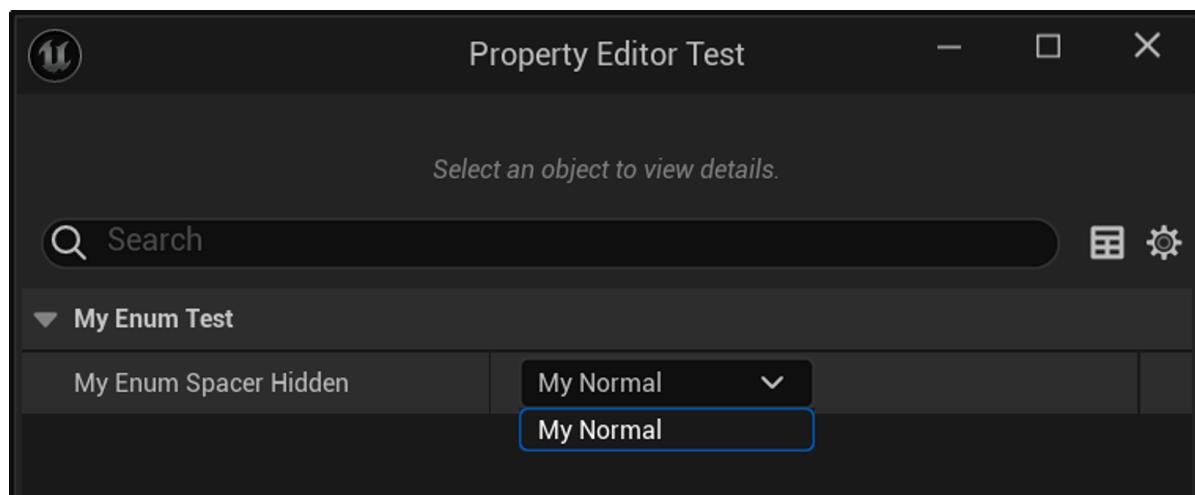
隐藏UENUM的某个值。

测试代码：

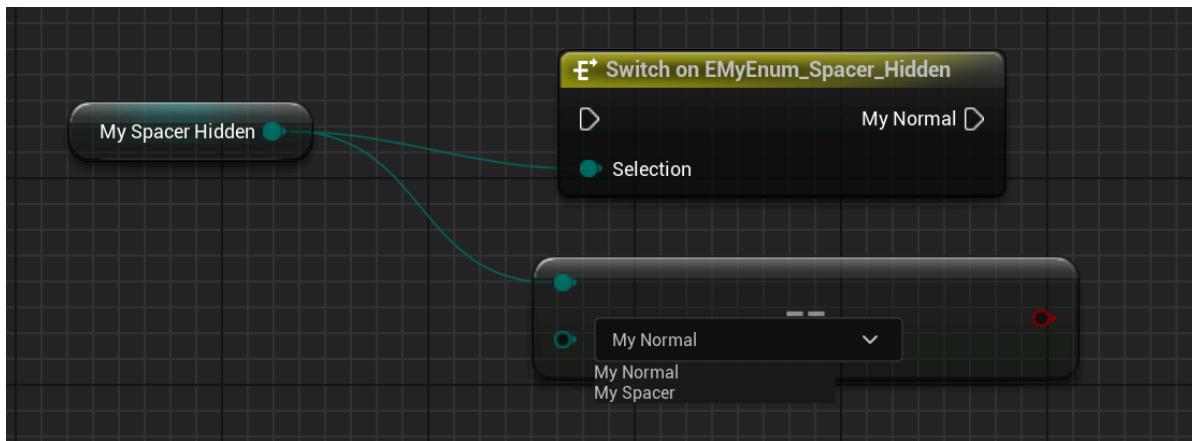
```
UENUM(Blueprintable, BlueprintType)
enum class EMyEnum_Spacer_Hidden : uint8
{
    MyNormal,
    MySpacer UMETA(Spacer),
    MyHidden UMETA(Hidden),
};

UCLASS(BlueprintType)
class INSIDER_API UMyEnum_Test : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EMyEnum_Spacer_Hidden MyEnum_Spacer_Hidden;
};
```

测试效果：



但是蓝图里访问：



原理：

在属性细节面板里生成枚举可能的值的时候，会判断Hidden和Spacer选项来隐藏。

但是在SEnumComboBox和SGraphPinEnum这种在蓝图里显示的时候，只会判断Hidden，而没有（忘了？）判断Spacer，因此MySpacer是依然会被显示出来。

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText *>* OutDisplayNames)
{
    // Ignore hidden enums
    bool bShouldBeHidden = Enum->HasMetaData(TEXT("Hidden"), EnumIndex) || Enum-
>HasMetaData(TEXT("Spacer"), EnumIndex);
    if (!bShouldBeHidden)
    {
        if (ValidEnumValues.Num() > 0)
        {
            bShouldBeHidden = !ValidEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
        }
        // If both are specified, InvalidEnumValues takes precedence
        else if (InvalidEnumValues.Num() > 0)
        {
            bShouldBeHidden = InvalidEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
        }
    }
}

void SEnumComboBox::Construct(const FArguments& InArgs, const UEnum* InEnum)
{
    if (Enum->HasMetaData(TEXT("Hidden"), Index) == false)
    {
        VisibleEnums.Emplace(Index, Enum->GetValueByIndex(Index), Enum-
>GetDisplayNameTextByIndex(Index), Enum->GetToolTipTextByIndex(Index));
    }
}

void SGraphPinEnum::GenerateComboBoxIndexes( TArray< TSharedPtr< int32 > >&
OutComboBoxIndexes )
{
    // Ignore hidden enum values
}

```

```

if( !EnumPtr->HasMetaData(TEXT("Hidden"), EnumIndex) )
{
    TSharedPtr<int32> EnumIdxPtr(new int32(EnumIndex));
    OutComboBoxIndexes.Add(EnumIdxPtr);
}
}

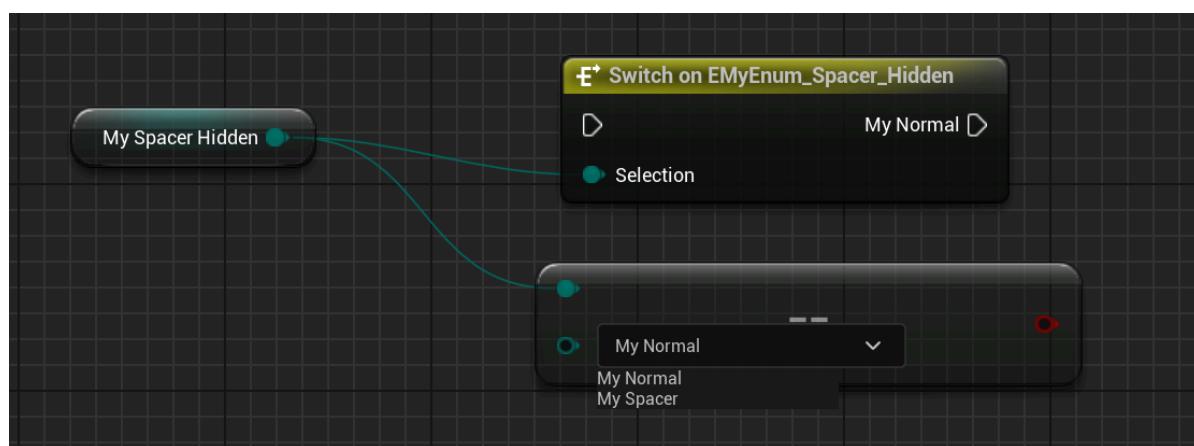
```

Spacer

- 功能描述:** 隐藏UENUM的某个值
- 使用位置:** UENUM::UMETA
- 引擎模块:** Enum Property
- 元数据类型:** bool
- 限制类型:** UENUM
- 常用程度:** ★★★★☆

Spacer和Hidden的功能大体是一致的。唯一区别是Spacer在蓝图里==的时候还是会显示出来。

因此还是建议如果要隐藏枚举值，还是要尽量都用Hidden。



其他示例代码见Hidden

TraceQuery

- 功能描述:** Enum /Script/Engine.ECollisionChannel
- 使用位置:** UENUM::UMETA
- 引擎模块:** Enum Property
- 元数据类型:** bool
- 常用程度:** 0

只在ECollisionChannel 上使用，指定哪些通道是用来Trace的。

源码例子：

```

UENUM(BlueprintType)
enum ECollisionChannel : int
{
    ECC_WorldStatic UMETA(DisplayName="WorldStatic"),
}

```

```

ECC_WorldDynamic UMETA(DisplayName="worldDynamic"),
ECC_Pawn UMETA(DisplayName="Pawn"),
ECC_Visibility UMETA(DisplayName="Visibility" , TraceQuery="1"),
ECC_Camera UMETA(DisplayName="Camera" , TraceQuery="1"),
ECC_PhysicsBody UMETA(DisplayName="PhysicsBody"),
ECC_Vehicle UMETA(DisplayName="Vehicle"),
ECC_Destructible UMETA(DisplayName="Destructible"),

/** Reserved for gizmo collision */
ECC_EngineTraceChannel1 UMETA(Hidden),

ECC_EngineTraceChannel2 UMETA(Hidden),
ECC_EngineTraceChannel3 UMETA(Hidden),
ECC_EngineTraceChannel4 UMETA(Hidden),
ECC_EngineTraceChannel5 UMETA(Hidden),
ECC_EngineTraceChannel6 UMETA(Hidden),

ECC_GameTraceChannel1 UMETA(Hidden),
ECC_GameTraceChannel2 UMETA(Hidden),
ECC_GameTraceChannel3 UMETA(Hidden),
ECC_GameTraceChannel4 UMETA(Hidden),
ECC_GameTraceChannel5 UMETA(Hidden),
ECC_GameTraceChannel6 UMETA(Hidden),
ECC_GameTraceChannel7 UMETA(Hidden),
ECC_GameTraceChannel8 UMETA(Hidden),
ECC_GameTraceChannel9 UMETA(Hidden),
ECC_GameTraceChannel10 UMETA(Hidden),
ECC_GameTraceChannel11 UMETA(Hidden),
ECC_GameTraceChannel12 UMETA(Hidden),
ECC_GameTraceChannel13 UMETA(Hidden),
ECC_GameTraceChannel14 UMETA(Hidden),
ECC_GameTraceChannel15 UMETA(Hidden),
ECC_GameTraceChannel16 UMETA(Hidden),
ECC_GameTraceChannel17 UMETA(Hidden),
ECC_GameTraceChannel18 UMETA(Hidden),

/** Add new serializeable channels above here (i.e. entries that exist in
FCollisionResponseContainer) */
/** Add only nonserialized/transient flags below */

// NOTE!!!! THESE ARE BEING DEPRECATED BUT STILL THERE FOR BLUEPRINT. PLEASE
DO NOT USE THEM IN CODE

ECC_OverlapAll_Deprecated UMETA(Hidden),
ECC_MAX,
};


```

原理:

```

void UCollisionProfile::LoadProfileConfig(bool bForceInit)
{
    static const FString TraceType = TEXT("TraceQuery");
}

```

UseEnumValuesAsMaskValuesInEditor

- **功能描述:** 指定枚举值已经是位移后的值，而不是位标记的索引下标。
- **使用位置:** UENUM
- **元数据类型:** bool
- **关联项:** Bitflags
- **常用程度:** ★★

指定在编辑器里的枚举值直接就是位标记的最终值，而不是标记位。但是注意在C++里的定义是一样的形式：

```
UENUM(BlueprintType, meta = (Bitflags, UseEnumValuesAsMaskValuesInEditor =
"true"))
enum class EMotionExtractor_MotionType : uint8
{
    None          = 0 | UMETA(Hidden),
    Translation   = 1 << 0,
    Rotation      = 1 << 1,
    Scale         = 1 << 2,
    TranslationSpeed = 1 << 3,
    RotationSpeed   = 1 << 4,
};

UENUM(meta = (Bitflags))
enum class EPCGChangeType : uint8
{
    None = 0,
    Cosmetic = 1 << 0,
    Settings = 1 << 1,
    Input = 1 << 2,
    Edge = 1 << 3,
    Node = 1 << 4,
    Structural = 1 << 5
};
```

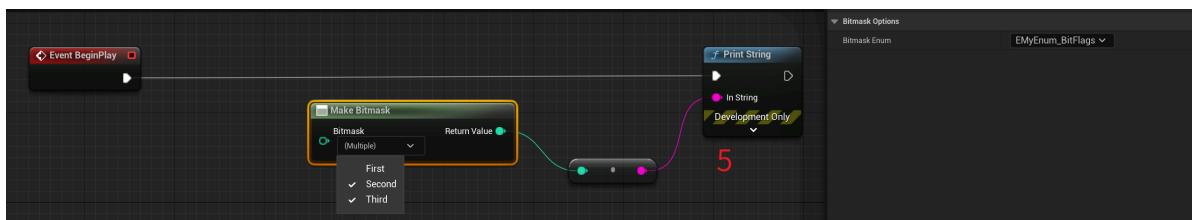
测试的代码：

```
UENUM(BlueprintType,Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First, //0
    Second, //1
    Third, //2
};

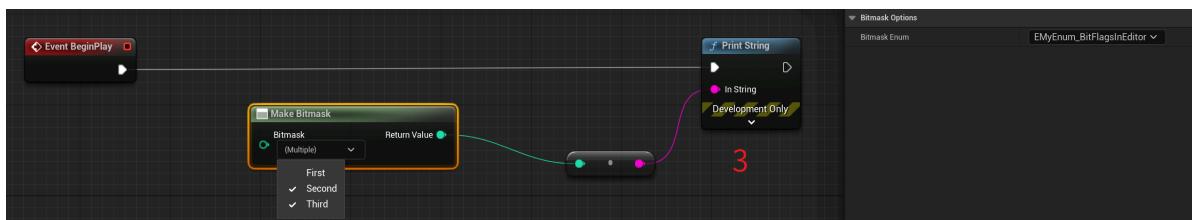
UENUM(BlueprintType, meta = (Bitflags, UseEnumValuesAsMaskValuesInEditor =
"true"))
enum class EMyEnum_BitFlagsInEditor:uint8
{
    First, //0
    Second, //1
    Third, //2
};
```

```
};
```

测试的蓝图1：



测试的蓝图2：



因此可以看出前者是 $1 << 2 + 1 << 2$, 而后者是 $1 | 2$, 因此后者是直接把枚举值作为已经位移后的值

ValidEnumValues

- 功能描述:** 指定枚举属性值上可选的枚举值选项
- 使用位置:** UPROPERTY
- 引擎模块:** Enum Property
- 元数据类型:** strings="a, b, c"
- 限制类型:** 枚举属性值
- 关联项:** InvalidEnumValues, GetRestrictedEnumValues, EnumValueDisplayNameOverrides, Enum
- 常用程度:** ★★★

指定枚举属性值上可选的枚举值选项，默认情况下。枚举属性在细节面板上可选项为全部的枚举值，但我们可以**通过ValidEnumValues来限制只展示这些值。**

枚举属性的写法有3种，分别是enum class, TEnumAsByte和FString叠加enum meta的写法，这3种写法都会被视为一个枚举属性然后尝试产生combo list来让用户选择属性值。

示例代码：

```
UENUM(BlueprintType)
enum class EMyPropertyTestEnum : uint8
{
    First,
    Second,
    Third,
    Forth,
    Fifth,
};

UENUM(BlueprintType)
namespace EMyPropertyTestEnum
```

```

{
    enum Type : int
    {
        First,
        Second,
        Third,
        Forth,
        Fifth,
    };
}

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Enum :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EMyPropertyTestEnum MyEnum;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ValidEnumValues =
"First,Second,Third"))
    EMyPropertyTestEnum MyEnumWithValid;      // Type 1

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ValidEnumValues =
"First,Second,Third"))
    TEnumAsByte<EMyPropertyTestEnum2::Type> MyAnotherEnumWithValid; //Type 2

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (enum =
"EMyPropertyTestEnum"))
    FString MyStringWithEnum;    //Type 3

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (InvalidEnumValues =
"First,Second,Third"))
    EMyPropertyTestEnum MyEnumWithInvalid = EMyPropertyTestEnum::Forth;

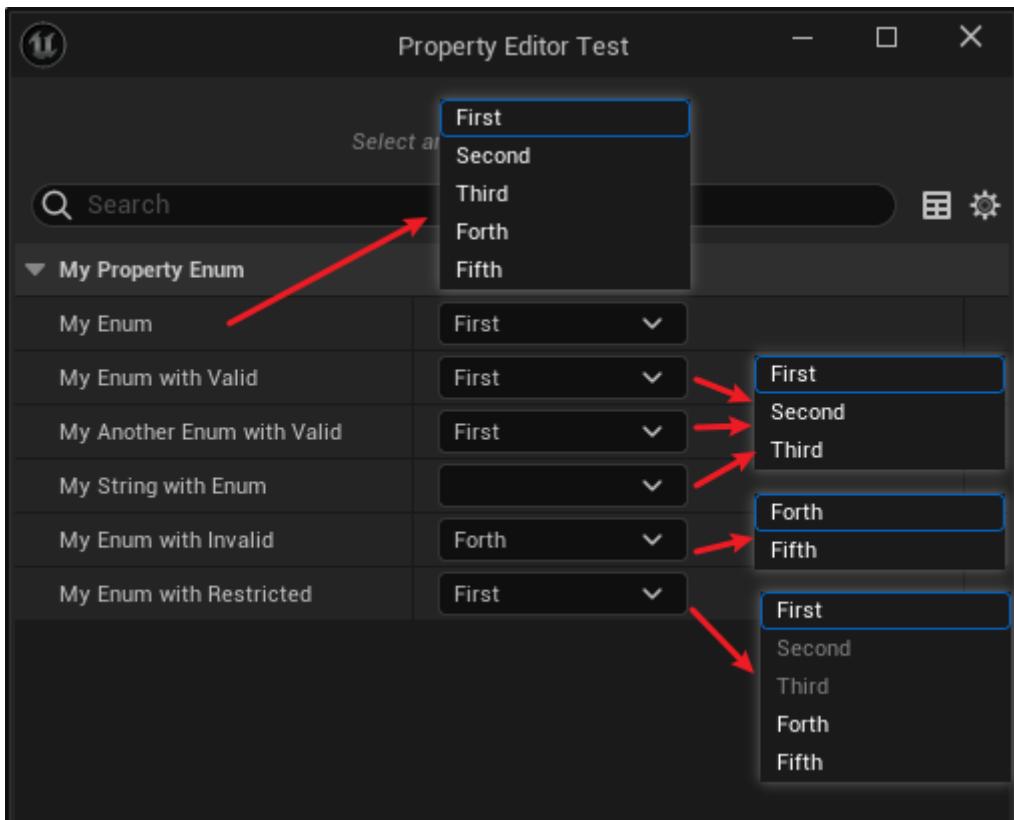
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (GetRestrictedEnumValues =
"MyGetRestrictedEnumValues"))
    EMyPropertyTestEnum MyEnumWithRestricted;

public:
    UFUNCTION(BlueprintInternalUseOnly)
    TArray< FString > MyGetRestrictedEnumValues() const{   return TArray< FString >
{ "Second", "Third" }; }
};

```

蓝图效果：

可见默认情况下枚举属性会显示全部5个枚举值，但其他3个枚举属性值的可选列表被限制到了3个。



原理：

下述的代码，

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< TSharedPtr< FString>> &
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems)
{
    TArray< UObject*> OuterObjects;
    GetOuterObjects(OuterObjects);

    const TArray< FName > ValidEnumValues =
        PropertyEditorHelpers::GetValidEnumsFromPropertyOverride(Property, Enum);
    const TArray< FName > InvalidEnumValues =
        PropertyEditorHelpers::GetInvalidEnumsFromPropertyOverride(Property, Enum);
    const TArray< FName > RestrictedEnumValues =
        PropertyEditorHelpers::GetRestrictedEnumsFromPropertyOverride(OuterObjects,
            Property, Enum);

    const TMap< FName, FText > EnumValueDisplayNameOverrides =
        PropertyEditorHelpers::GetEnumValueDisplayNamesFromPropertyOverride(Property,
            Enum);

    //NumEnums() - 1, because the last item in an enum is the _MAX item
    for( int32 EnumIndex = 0; EnumIndex < Enum->NumEnums() - 1; ++EnumIndex )
    {
        // Ignore hidden enums
        bool bShouldBeHidden = Enum->HasMetaData(TEXT("Hidden"), EnumIndex ) ||
            Enum->HasMetaData(TEXT("Spacer"), EnumIndex );
        if (!bShouldBeHidden)
        {
            if( validEnumValues.Num() > 0)
            {

```

```

        bShouldBeHidden = !ValidEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
    }
    // If both are specified, InvalidEnumValues takes precedence
    else if(InvalidEnumValues.Num() > 0)
    {
        bShouldBeHidden = InvalidEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
    }
}

if (!bShouldBeHidden)
{
    bShouldBeHidden = IsHidden(Enum->GetNameStringByIndex(EnumIndex));
}

if( !bShouldBeHidden )
{
    // See if we specified an alternate name for this value using
metadata
    FString EnumName = Enum->GetNameStringByIndex(EnumIndex);
    FString EnumDisplayName = EnumValueDisplayNameOverrides.FindRef(Enum-
>GetNameByIndex(EnumIndex)).ToString();
    if (EnumDisplayName.IsEmpty())
    {
        EnumDisplayName = Enum-
>GetDisplayNameTextByIndex(EnumIndex).ToString();
    }

    FText RestrictionTooltip;
    const bool bIsRestricted = GenerateRestrictionToolTip(EnumName,
RestrictionTooltip) || RestrictedEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
    OutRestrictedItems.Add(bIsRestricted);

    if (EnumDisplayName.Len() == 0)
    {
        EnumDisplayName = MoveTemp(EnumName);
    }
    else
    {
        bUsesAlternateDisplayValues = true;
    }

    TSharedPtr< FString > EnumStr(new FString(EnumDisplayName));
    OutOptionStrings.Add(EnumStr);

    FText EnumValueToolTip = bIsRestricted ? RestrictionTooltip : Enum-
>GetToolTipTextByIndex(EnumIndex);
    OutToolTips.Add(MoveTemp(EnumValueToolTip));
}
else
{
    OutToolTips.Add(FText());
}
}

```

```
}
```

HideFromModifiers

- 功能描述:** 指定AttributeSet下的某属性不出现在GameplayEffect下的Modifiers的Attribute选择里。
- 使用位置:** UPROPERTY
- 引擎模块:** GAS
- 元数据类型:** bool
- 限制类型:** UAttributeSet下的属性
- 关联项:** HideInDetailsView
- 常用程度:** ★★★

指定AttributeSet下的某属性不出现在GameplayEffect下的Modifiers的Attribute选择里。

测试代码:

```
UCLASS()
class UMyAttributeSet : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideInDetailsView))
    float HP_HideInDetailsView = 100.f;

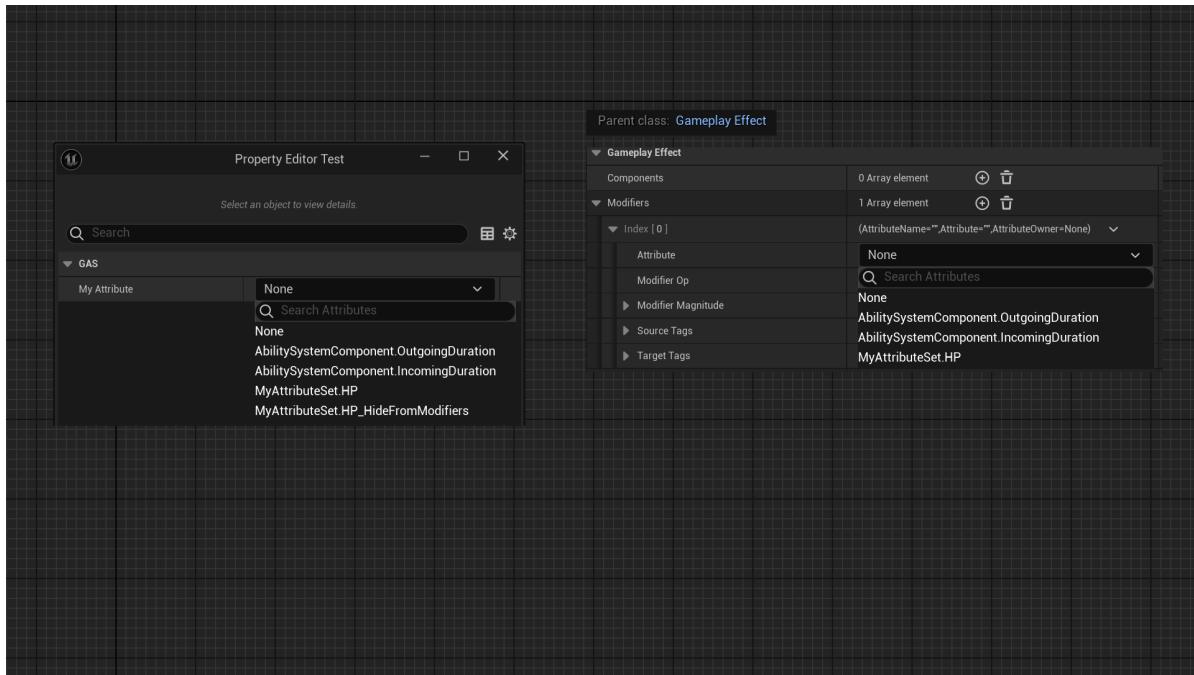
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideFromModifiers))
    float HP_HideFromModifiers = 100.f;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    FGameplayAttribute MyAttribute;
};
```

测试结果:

在蓝图中创建一个GameplayEffect，然后观察其Modifiers下的Attribute选择。

发现HP_HideFromModifiers 可以出现在正常的FGameplayAttribute 选项卡中，但不能出现在Modifiers下的Attribute选项卡里。这就是这里的作用。



原理：

在FGameplayModifierInfo的Attribute属性上有FilterMetaTag的元数据，然后取出其里面的值，最终还是传到GetAllAttributeProperties作为FilterMetaStr来过滤。因此属性上出现HideFromModifiers就不能出现。

```

USTRUCT(BlueprintType)
struct GAMEPLAYABILITIES_API FGameplayModifierInfo
{
    GENERATED_USTRUCT_BODY()

    /** The Attribute we modify or the GE we modify modifies. */
    UPROPERTY(EditDefaultsOnly, Category=GameplayModifier, meta=(FilterMetaTag="HideFromModifiers"))
        FGameplayAttribute Attribute;
    };

    void FAttributePropertyDetails::CustomizeHeader( TSharedRef<IPropertyHandle> StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
    IPropertyTypeCustomizationUtils& StructCustomizationUtils )
    {
        const FString& FilterMetaStr = StructPropertyHandle->GetProperty()->GetMetaData(TEXT("FilterMetaTag"));
        SNew(SGameplayAttributeWidget)
            .OnAttributeChanged(this, &FAttributePropertyDetails::OnAttributeChanged)
            .DefaultProperty(PropertyValue)
            .FilterMetaData(FilterMetaStr)
    }
    void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>& OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
    {}
}

```

HideInDetailsView

- 功能描述：**把该UAttributeSet子类里的属性隐藏在FGameplayAttribute的选项列表里。

- **使用位置:** UCLASS, UPROPERTY
- **引擎模块:** GAS
- **元数据类型:** bool
- **限制类型:** UAttributeSet
- **关联项:** HideFromModifiers, SystemGameplayAttribute
- **常用程度:** ★★★

把该UAttributeSet子类里的属性隐藏在FGameplayAttribute的选项列表里。

可用在UCLASS上隐藏掉整个类里的所有属性，也可以用在某个特定属性上只隐藏该属性。

在源码里用到的例子是UAbilitySystemTestAttributeSet，因为其就是一个专门用来测试的AS，因此希望不影响正常的选项列表。

测试代码：

```
UCLASS()
class UMyAttributeSet : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;

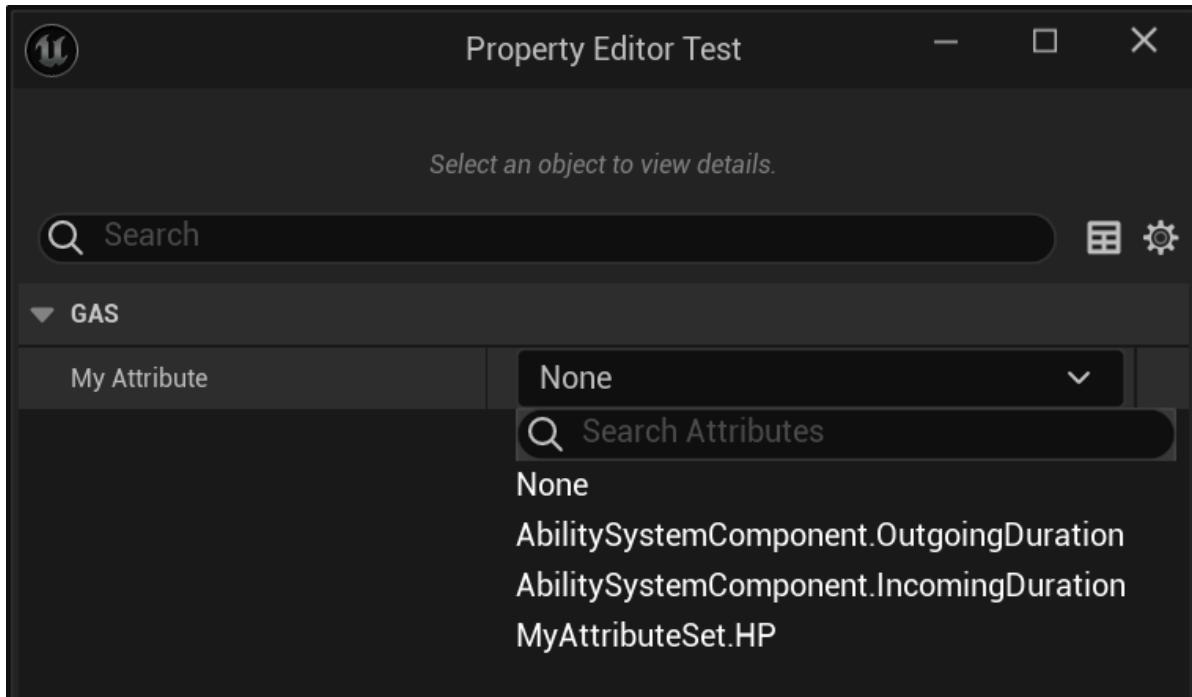
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideInDetailsView))
    float HP_HideInDetailsView = 100.f;
};

UCLASS(meta = (HideInDetailsView))
class UMyAttributeSet_Hide : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    FGameplayAttribute MyAttribute;
};
```

测试效果：

可见只能选到UMyAttributeSet.HP属性，无法选择到UMyAttributeSet_Hide里的选项。



原理：

```
PropertyModule.RegisterCustomPropertyTypeLayout( "GameplayAttribute",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(
&FAttributePropertyParams::MakeInstance ) );

void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>&
OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
{
    for (TObjectIterator<UClass> ClassIt; ClassIt; ++ClassIt)
    {
        if (UseEditorOnlyData)
        {
            #if WITH_EDITOR
            // Allow entire classes to be filtered globally
            if (Class->HasMetaData(TEXT("HideInDetailsView")))
            {
                continue;
            }
            #endif
        }

        for (TFieldIterator<FProperty> PropertyIt(Class,
EFieldIteratorFlags::ExcludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;

            if (UseEditorOnlyData)
            {
                // Allow properties to be filtered globally (never show up)
                if (Property->HasMetaData(TEXT("HideInDetailsView")))
                {
                    continue;
                }
            }
        }
    }
}
```

```

        }

        OutProperties.Add(Property);
    }
}

```

SystemGameplayAttribute

- 功能描述:** 把UAbilitySystemComponent子类里面的属性暴露到FGameplayAttribute 选项框里。
- 使用位置:** UPROPERTY
- 引擎模块:** GAS
- 元数据类型:** bool
- 限制类型:** UAbilitySystemComponent子类里面的属性
- 关联项:** HideInDetailsView
- 常用程度:** ★★★

把UAbilitySystemComponent子类里面的属性暴露到FGameplayAttribute 选项框里。

源码例子:

```

UCLASS(ClassGroup=AbilitySystem, hidecategories=
(Object, LOD, Lighting, Transform, Sockets, TextureStreaming), editinlinenew, meta=
(BlueprintSpawnableComponent))
class GAMEPLAYABILITIES_API UAbilitySystemComponent : public
UGameplayTasksComponent, public IGameplayTagAssetInterface, public
IAbilitySystemReplicationProxyInterface
{
    /** Internal Attribute that modifies the duration of gameplay effects created
     * by this component */
    UPROPERTY(meta=(SystemGameplayAttribute="true"))
    float OutgoingDuration;

    /** Internal Attribute that modifies the duration of gameplay effects applied
     * to this component */
    UPROPERTY(meta = (SystemGameplayAttribute = "true"))
    float IncomingDuration;
}

```

测试代码:

```

UCLASS(Blueprintable, BlueprintType)
class UMyAbilitySystemComponent : public UAbilitySystemComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    float MyFloat;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS", meta =
    (SystemGameplayAttribute))

```

```

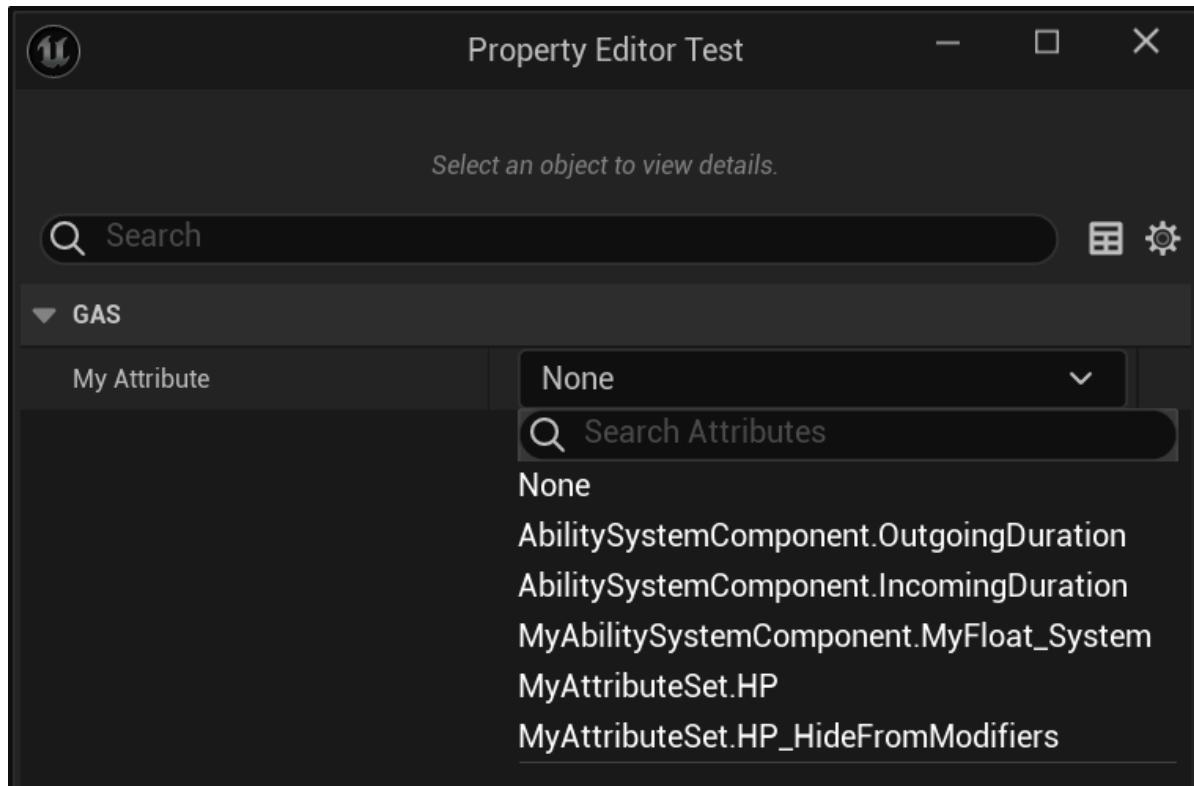
    float MyFloat_System;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    FGameplayAttribute MyAttribute;
};

```

测试效果：

可见MyFloat_System可以暴露到选项列表里去。



原理：

也是在 GetAllAttributeProperties 判断属性上的 SystemGameplayAttribute 标记。

```

void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>& OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
{
    // UAbilitySystemComponent can add 'system' attributes
    if (Class->IsChildOf(UAbilitySystemComponent::StaticClass()) && !Class->ClassGeneratedBy)
    {
        for (TFieldIterator<FProperty> PropertyIt(Class,
EFIELDITERATORFLAGS::ExcludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;

```

```

// SystemAttributes have to be explicitly tagged
if (Property->HasMetaData(TEXT("SystemGameplayAttribute")) == false)
{
    continue;
}
OutProperties.Add(Property);
}
}

```

MaterialControlFlow

- 功能描述:** 标识该UMaterialExpression为一个控制流节点，当前在材质蓝图右键菜单中隐藏。
- 使用位置:** UCLASS
- 引擎模块:** Material
- 元数据类型:** bool
- 限制类型:** UMaterialExpression子类
- 常用程度:** ★

标识该UMaterialExpression为一个控制流节点，当前在材质蓝图右键菜单中隐藏。

一般是在IfThenElse, ForLoop这种节点上，当前引擎实现还未完善，因此该功能禁用。

源码例子：

```

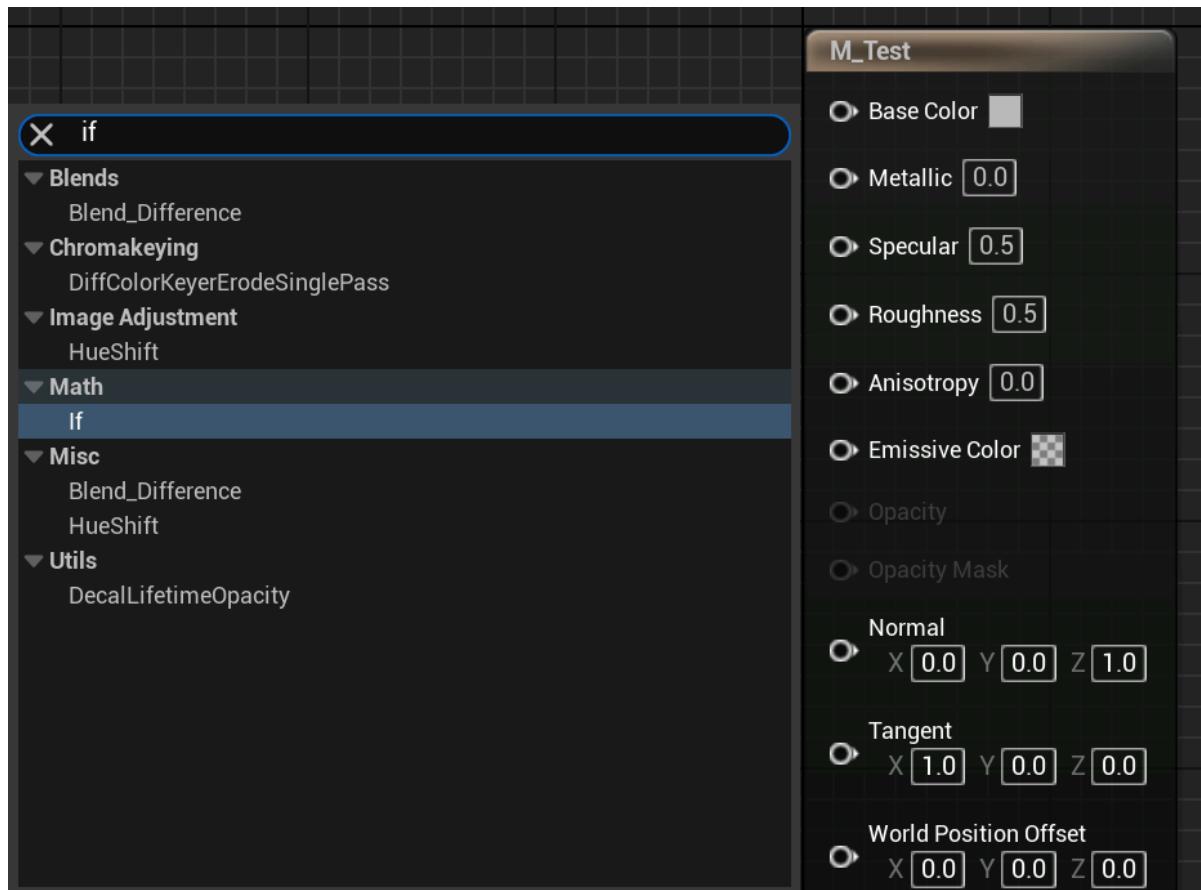
UCLASS(collapsecategories, hidecategories=Object, MinimalAPI)
class UMaterialExpressionIf : public UMaterialExpression
{};

UCLASS(collapsecategories, hidecategories = Object, MinimalAPI, meta=
(MaterialControlFlow))
class UMaterialExpressionIfThenElse : public UMaterialExpression
{};

```

测试效果：

可以找到If节点，但是无法调用IfThenElse节点。



原理：

在遍历所有可用FMaterialExpression的时候，如果有MaterialControlFlow则略过。当前引擎下AllowMaterialControlFlow一直未false，未实现。

```

// r.MaterialEnableControlFlow is removed and the feature is forced disabled as
// how control flow should be
// implemented in the material editor is still under discussion.
inline bool AllowMaterialControlFlow()
{
    return false;
}

void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHLSLGenerator =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.MaterialEnableNewHLSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
    const bool bEnableNewHLSLGenerator = CVarMaterialEnableNewHLSLGenerator-
>GetValueOnAnyThread() != 0;

    for( TObjectIterator<UClass> It ; It ; ++It )
    {
        if( class->IsChildOf(UMaterialExpression::StaticClass()) )
        {
            // Hide node types related to control flow, unless it's
enabled
    }
}

```

```

        if (!bEnableControlFlow && Class-
>HasMetaData("MaterialControlFlow"))
{
    continue;
}

if (!bEnableNewHLSLGenerator && Class-
>HasMetaData("MaterialNewHLSLGenerator"))
{
    continue;
}

// Hide node types that are tagged private
if (Class->HasMetaData(TEXT("Private")))
{
    continue;
}
AllExpressionClasses.Add(MaterialExpression);
}
}
}

```

MaterialNewHLSLGenerator

- 功能描述:** 标识该UMaterialExpression为采用新HLSL生成器的节点，当前在材质蓝图右键菜单中隐藏。
- 使用位置:** UCLASS
- 引擎模块:** Material
- 元数据类型:** bool
- 限制类型:** UMaterialExpression子类
- 常用程度:** ★

标识该UMaterialExpression为采用新HLSL生成器的节点，当前在材质蓝图右键菜单中隐藏。

源码例子：

```

UCLASS(MinimalAPI, meta = (MaterialNewHLSLGenerator))
class UMaterialExpressionLess : public UMaterialExpressionBinaryOp
{
    GENERATED_UCLASS_BODY()
#if WITH_EDITOR
    virtual UE::HLSLTree::EOperation GetBinaryOp() const override { return
UE::HLSLTree::EOperation::Less; }
#endif // WITH_EDITOR
};

```

测试效果：

材质蓝图里无法调用Less。

原理：

在遍历所有可用FMaterialExpression的时候，如果有MaterialEnableNewHLSLGenerator则略过。当前引擎下r.MaterialEnableNewHLSLGenerator是只读的，且实现未完全。

```
static TAutoConsoleVariable<int32> CVarMaterialEnableNewHLSLGenerator(
    TEXT("r.MaterialEnableNewHLSLGenerator"),
    0,
    TEXT("Enables the new (WIP) material HLSL generator.\n")
    TEXT("0 - Don't allow\n")
    TEXT("1 - Allow if enabled by material\n")
    TEXT("2 - Force all materials to use new generator\n"),
    ECVF_RenderThreadSafe | ECVF_ReadOnly);
```

```
void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHLSLGenerator =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.MaterialEnableNewHLSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
    const bool bEnableNewHLSLGenerator = CVarMaterialEnableNewHLSLGenerator->GetValueOnAnyThread() != 0;

    for( TObjectIterator<UClass> It ; It ; ++It )
    {
        if( class->IsChildOf(UMaterialExpression::StaticClass()) )
        {
            // Hide node types related to control flow, unless it's
enabled
            if (!bEnableControlFlow && class->HasMetaData("MaterialControlFlow"))
            {
                continue;
            }

            if (!bEnableNewHLSLGenerator && class->HasMetaData("MaterialNewHLSLGenerator"))
            {
                continue;
            }

            // Hide node types that are tagged private
            if(class->HasMetaData(TEXT("Private")))
            {
                continue;
            }
            AllExpressionClasses.Add(MaterialExpression);
        }
    }
}
```

MaterialParameterCollectionFunction

- **功能描述:** 指定该函数是用于操作UMaterialParameterCollection，从而支持ParameterName的提取和验证
- **使用位置:** UFUNCTION
- **引擎模块:** Material
- **元数据类型:** bool
- **限制类型:** 带有UMaterialParameterCollection参数的函数
- **常用程度:** ★★★

指定该函数是用于操作UMaterialParameterCollection，从而支持ParameterName的提取和验证。

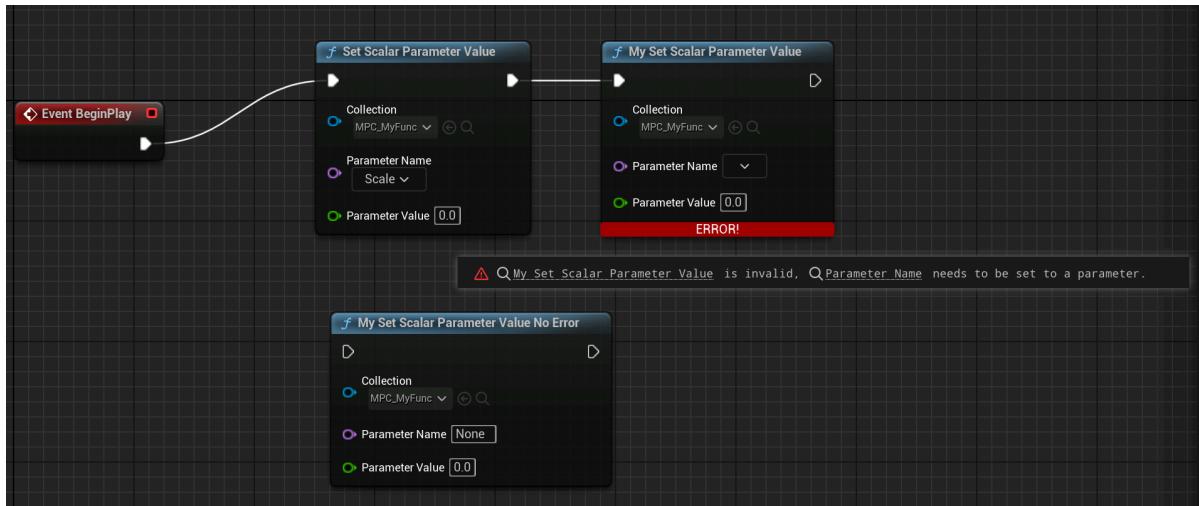
测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Material :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
MaterialParameterCollectionFunction))
        static void MySetScalarParameterValue(UObject* WorldContextObject,
UMaterialParameterCollection* Collection, FName ParameterName, float
ParameterValue);

    UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject"))
        static void MySetScalarParameterValue_NoError(UObject* WorldContextObject,
UMaterialParameterCollection* Collection, FName ParameterName, float
ParameterValue);
};
```

蓝图中效果：

引擎自带的UKismetMaterialLibrary::SetScalarParameterValue和我们自己手动编写的MySetScalarParameterValue，会触发材质参数集合的蓝图节点验证检测。如果没有指定ParameterName，则会产生编译错误。而没有MaterialParameterCollectionFunction标记的MySetScalarParameterValue_NoError函数版本则只是当作一个普通的函数一样，一是不会自动提取MPC中的Parameters集合来选择，二是没有ParameterName为空的错误验证。



原理：

```

UBlueprintFunctionNodeSpawner* UBlueprintFunctionNodeSpawner::Create(UFunction
const* const Function, UObject* Outer/* = nullptr*/)
{
    bool const bIsMaterialParamCollectionFunc = Function-
>HasMetaData(FBlueprintMetadata::MD_MaterialParameterCollectionFunction);
    if (bIsMaterialParamCollectionFunc)
    {
        NodeClass =
UK2Node_CallMaterialParameterCollectionFunction::StaticClass();
    }
    else
    {
        NodeClass = UK2Node_CallFunction::StaticClass();
    }
}

TSharedPtr<SGraphNode> FNodeFactory::CreateNodeWidget(UEdGraphNode* InNode)
{
    if (UK2Node_CallMaterialParameterCollectionFunction* CallFunctionNode =
Cast<UK2Node_CallMaterialParameterCollectionFunction>(InNode))
    {
        return SNew(SGraphNodeCallParameterCollectionFunction,
CallFunctionNode);
    }
}
TSharedPtr<SGraphPin>
SGraphNodeCallParameterCollectionFunction::CreatePinWidget(UEdGraphPin* Pin)
const
{
    // 提取MPC中参数列表等操作
    if (Collection)
    {
        // Populate the ParameterName pin combobox with valid options from
        // the Collection
        const bool bVectorParameters = CallFunctionNode-
>FunctionReference.GetMemberName().ToString().Contains(TEXT("Vector"));
        Collection->GetParameterNames(NameList, bVectorParameters);
    }
}

```

```
}
```

MaterialParameterCollectionFunction这个标记的，会采用UK2Node_CallMaterialParameterCollectionFunction来验证材质函数的正确书写与否。同时UK2Node_CallMaterialParameterCollectionFunction这个蓝图节点也在引擎内部继续被识别以进一步定制化ParameterName的Pin节点。

引擎源码内采用MaterialParameterCollectionFunction标记的函数只有UKismetMaterialLibrary里的函数。

OverridingInputProperty

- **功能描述:** 在UMaterialExpression中指定本float要覆盖的其他FExpressionInput 属性。
- **使用位置:** UPROPERTY
- **引擎模块:** Material
- **元数据类型:** bool
- **限制类型:** UMaterialExpression::float
- **关联项:** RequiredInput
- **常用程度:** ★★★

在UMaterialExpression中指定本float要覆盖的其他FExpressionInput 属性。

- 在材质表达式里有些属性，当用户没连接的时候，我们想提供一个默认值。这个时候这个Float属性的指就可以当作默认值。
- 而当用户连接的时候，这个引脚又要变成一个正常的输入，因此得有另一个FExpressionInput 属性，所以才需要用OverridingInputProperty 指定另一个属性。
- 被OverridingInputProperty 指定的FExpressionInput 属性一般是RequiredInput = "false"，因为正常的逻辑是你都提供默认值了，那当然用户就不一定非得输入这个值了。当然也可以RequiredInput = "true"，提醒用户这个引脚最好要有个输入，但如果真没有，也可以用默认值。
- 输出节点上的很多BaseColor之类的引脚就是又RequiredInput又提供默认值的。

测试代码：

在Compile函数中模仿源码给大家示范如何正确处理这种属性来获得值。大家也可以参照源码中别的例子来参考。

```
UCLASS()
class UMyProperty_MyMaterialExpression : public UMaterialExpression
{
    GENERATED_UCLASS_BODY()

public:
    UPROPERTY()
    FExpressionInput MyInput_Default;

    UPROPERTY(meta = (RequiredInput = "true"))
    FExpressionInput MyInput_Required;

    UPROPERTY(meta = (RequiredInput = "false"))
    FExpressionInput MyInput_NotRequired;
}
```

```

    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(RequiredInput = "false"))
    FExpressionInput MyAlpha;

    /** only used if MyAlpha is not hooked up */
    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(OverridingInputProperty = "MyAlpha"))
    float ConstAlpha = 1.f;

    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(RequiredInput = "true"))
    FExpressionInput MyAlpha2;

    /** only used if MyAlpha is not hooked up */
    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(OverridingInputProperty = "MyAlpha2"))
    float ConstAlpha2 = 1.f;
public:

    //~ Begin UMaterialExpression Interface
#if WITH_EDITOR
    virtual int32 Compile(class FMaterialCompiler* Compiler, int32 OutputIndex)
override
{
    int32 IndexAlpha = MyAlpha.GetTracedInput().Expression ?
MyAlpha.Compile(Compiler) : Compiler->Constant(ConstAlpha);
    return 0;
}
    virtual void GetCaption(TArray<FString>& OutCaptions) const override;

    virtual bool GenerateHLSLExpression(FMaterialHSLGenerator& Generator,
UE::HLSLTree::FScope& Scope, int32 OutputIndex, UE::HLSLTree::FExpression const*&
OutExpression) const override;
#endif
    //~ End UMaterialExpression Interface
};

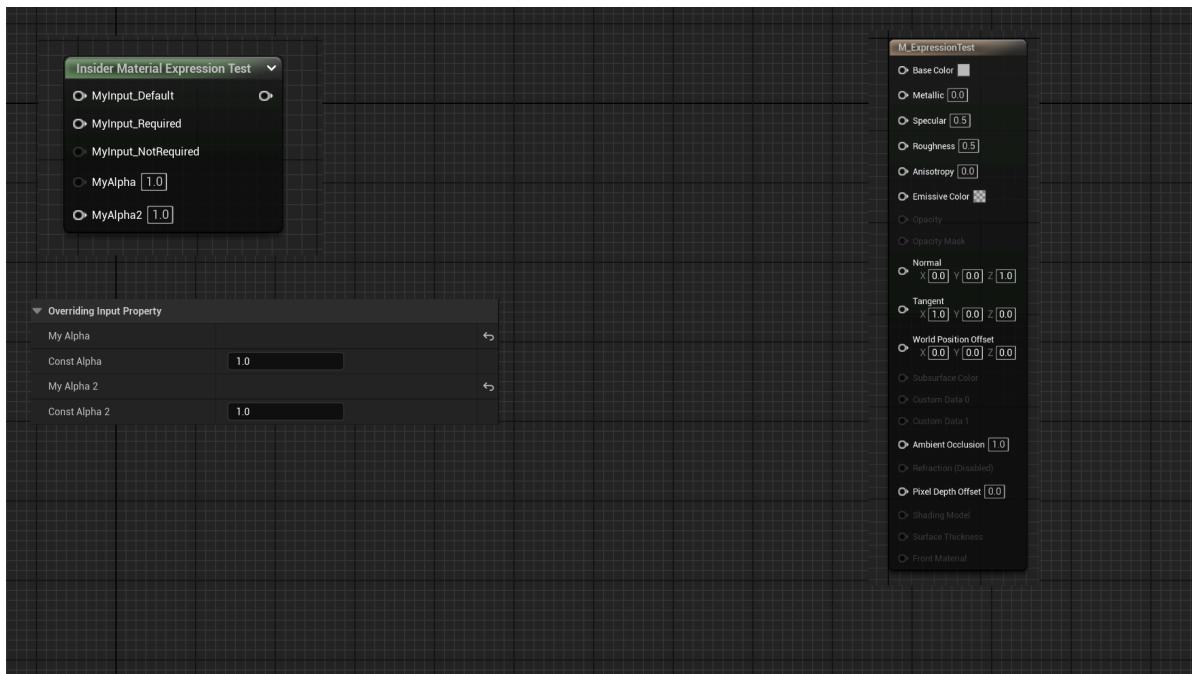
```

测试结果：

可见MyAlpha和MyAlpha2的右边都有个默认值，二者又因为RequiredInput的不同而一个灰色一个白色。

其他3个MyInput用来给大家对比。

右侧的材质最终输出表达式上的各个引脚更是有各种情况来让大家参考。



原理：

其实也是根据Meta的标记，来改变引脚的可编辑状态以及输入框。

```

bool UMaterialExpression::CanEditChange(const FProperty* InProperty) const
{
    bool bIsEditable = Super::CanEditChange(InProperty);
    if (bIsEditable && InProperty != nullptr)
    {
        // Automatically set property as non-editable if it has
        OverridingInputProperty metadata
        // pointing to an FExpressionInput property which is hooked up as an
        input.
        //
        // e.g. in the below snippet, meta=(OverridingInputProperty = "A")
        indicates that ConstA will
        // be overridden by an FExpressionInput property named 'A' if one is
        connected, and will thereby
        // be set as non-editable.
        //
        // UPROPERTY(meta = (RequiredInput = "false", ToolTip = "Defaults to
        'ConstA' if not specified"))
        // FExpressionInput A;
        //
        // UPROPERTY(EditAnywhere, Category = MaterialExpressionAdd, meta =
        OverridingInputProperty = "A"))
        // float ConstA;
        //

        static FName
        OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));

        if (InProperty->HasMetaData(OverridingInputPropertyMetaData))
        {
    
```

```

        const FString& OverridingPropertyName = InProperty-
>GetMetaData(OverridingInputPropertyMetaData);

        FStructProperty* StructProp = FindFProperty<FStructProperty>
(GetClass(), *OverridingPropertyName);
        if (ensure(StructProp != nullptr))
        {
            static FName RequiredInputMetaData(TEXT("RequiredInput"));

            // Must be a single FExpressionInput member, not an array, and
            must be tagged with metadata RequiredInput="false"
            if (ensure( StructProp->Struct->GetFName() ==
NAME_ExpressionInput &&
                StructProp->ArrayDim == 1 &&
                StructProp->HasMetaData(RequiredInputMetaData) &&
                !StructProp->GetBoolMetaData(RequiredInputMetaData)))
            {
                const FExpressionInput* Input = StructProp-
>ContainerPtrToValuePtr<FExpressionInput>(this);

                if (Input->Expression != nullptr && Input-
>GetTracedInput().Expression != nullptr)
                {
                    bIsEditable = false;
                }
            }
        }

        if (bIsEditable)
        {
            // If the property has EditCondition metadata, then whether it's
            editable depends on the other EditCondition property
            const FString EditConditionPropertyName = InProperty-
>GetMetaData(TEXT("EditCondition"));
            if (!EditConditionPropertyName.IsEmpty())
            {
                FBoolProperty* EditConditionProperty =
FindFProperty<FBoolProperty>(GetClass(), *EditConditionPropertyName);
                {
                    bIsEditable = *EditConditionProperty-
>ContainerPtrToValuePtr<bool>(this);
                }
            }
        }

        return bIsEditable;
    }
}

```

Private

- **功能描述:** 标识该UMaterialExpression为私有节点，当前在材质蓝图右键菜单中隐藏。

- **使用位置:** UCLASS
- **引擎模块:** Material
- **元数据类型:** bool
- **限制类型:** UMaterialExpression子类
- **常用程度:** ★

标识该UMaterialExpression为私有节点，当前在材质蓝图右键菜单中隐藏。

在MaterialX模块中使用，目前是把里面的Expression暂时先都隐藏起来。

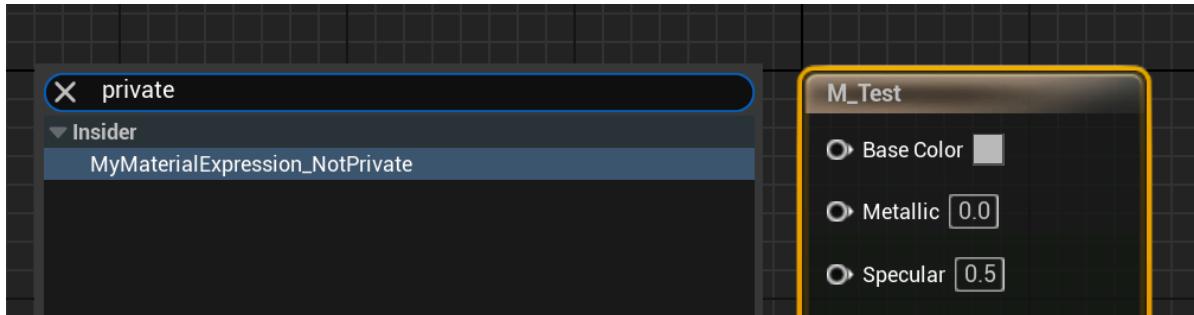
源码例子：

```
UCLASS()
class UMyMaterialExpression_NotPrivate : public UMaterialExpression
{};

UCLASS(meta=(Private))
class UMyMaterialExpression_Private : public UMaterialExpression
{};
```

测试效果：

材质蓝图里只能调用UMyMaterialExpression_NotPrivate。



原理：

在遍历所有可用FMaterialExpression的时候，如果有Private则略过。

```
static TAutoConsoleVariable<int32> CVarMaterialEnableNewHLSLGenerator(
    TEXT("r.MaterialEnableNewHLSLGenerator"),
    0,
    TEXT("Enables the new (WIP) material HLSL generator.\n")
    TEXT("0 - Don't allow\n")
    TEXT("1 - Allow if enabled by material\n")
    TEXT("2 - Force all materials to use new generator\n"),
    ECVF_RenderThreadSafe | ECVF_ReadOnly);

void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHLSLGenerator =
    IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.MaterialEnableNewHLSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
```

```

        const bool bEnableNewHLSLGenerator = CVarMaterialEnableNewHLSLGenerator-
>GetValueOnAnyThread() != 0;

        for( TobjectIterator<UClass> It ; It ; ++It )
        {
            if( Class->IsChildOf(UMaterialExpression::StaticClass()) )
            {
                // Hide node types related to control flow, unless it's
enabled
                if( !bEnableControlFlow && Class-
>HasMetaData("MaterialControlFlow") )
                {
                    continue;
                }

                if( !bEnableNewHLSLGenerator && Class-
>HasMetaData("MaterialNewHLSLGenerator") )
                {
                    continue;
                }

                // Hide node types that are tagged private
                if( Class->HasMetaData(TEXT("Private")) )
                {
                    continue;
                }
                AllExpressionClasses.Add(MaterialExpression);
            }
        }
    }
}

```

RequiredInput

- 功能描述:** 在UMaterialExpression中指定FExpressionInput属性是否要求输入，引脚显示白色或灰色。
- 使用位置:** UPROPERTY
- 引擎模块:** Material
- 元数据类型:** bool
- 限制类型:** UMaterialExpression::FExpressionInput
- 关联项:** OverridingInputProperty

在UMaterialExpression中指定FExpressionInput属性是否要求输入，引脚显示白色或灰色。

一般都配合OverridingInputProperty使用。

代码和效果参见OverridingInputProperty

原理:

```

bool UMaterialExpression::CanEditChange(const FProperty* InProperty) const
{
    bool bIsEditable = Super::CanEditChange(InProperty);
}

```

```

    if (bIsEditable && InProperty != nullptr)
    {
        // Automatically set property as non-editable if it has
        OverridingInputProperty metadata
            // pointing to an FExpressionInput property which is hooked up as an
            input.
            //
            // e.g. in the below snippet, meta=(OverridingInputProperty = "A")
            indicates that ConstA will
                // be overridden by an FExpressionInput property named 'A' if one is
                connected, and will thereby
                    // be set as non-editable.
                    //
                    // UPROPERTY(meta = (RequiredInput = "false", ToolTip = "Defaults to
                    'ConstA' if not specified"))
                    // FExpressionInput A;
                    //
                    // UPROPERTY(EditAnywhere, Category = MaterialExpressionAdd, meta =
                    OverridingInputProperty = "A"))
                    // float ConstA;
                    //

        static FName
        OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));

        if (InProperty->HasMetaData(OverridingInputPropertyMetaData))
        {
            const FString& OverridingPropertyName = InProperty-
            >GetMetaData(OverridingInputPropertyMetaData);

            FStructProperty* StructProp = FindFProperty<FStructProperty>
            (GetClass(), *OverridingPropertyName);
            if (ensure(StructProp != nullptr))
            {
                static FName RequiredInputMetaData(TEXT("RequiredInput"));

                // Must be a single FExpressionInput member, not an array, and
                must be tagged with metadata RequiredInput="false"
                if (ensure( StructProp->Struct->GetFName() ==
                NAME_ExpressionInput &&
                    StructProp->ArrayDim == 1 &&
                    StructProp->HasMetaData(RequiredInputMetaData) &&
                    !StructProp->GetBoolMetaData(RequiredInputMetaData)))
                {
                    const FExpressionInput* Input = StructProp-
                    >ContainerPtrToValuePtr<FExpressionInput>(this);

                    if (Input->Expression != nullptr && Input-
                    >GetTracedInput().Expression != nullptr)
                    {
                        bIsEditable = false;
                    }
                }
            }
        }
    }
}

```

```

if (bIsEditable)
{
    // If the property has EditCondition metadata, then whether it's
    // editable depends on the other EditCondition property
    const FString EditConditionPropertyName = InProperty-
>GetMetaData(TEXT("EditCondition"));
    if (!EditConditionPropertyName.IsEmpty())
    {
        FBoolProperty* EditConditionProperty =
FindFProperty<FBoolProperty>(GetClass(), *EditConditionPropertyName);
        {
            bIsEditable = *EditConditionProperty-
>ContainerPtrToValuePtr<bool>(this);
        }
    }
}

return bIsEditable;
}

```

ShowAsInputPin

- 功能描述:** 使得UMaterialExpression里的一些基础类型属性变成材质节点上的引脚。
- 使用位置:** UPROPERTY
- 引擎模块:** Material
- 元数据类型:** bool
- 限制类型:** UMaterialExpression里的属性
- 常用程度:** ★★★

使得UMaterialExpression里的一些基础类型属性变成材质节点上的引脚。

- 所谓基础类型，指的是float, FVector这种常用的类型。
- 默认这些基础类型属性是不显示为引脚的。
- ShowAsInputPin 值有两个选项，Primary可以直接显示出来，Advanced需要展开箭头后显示。

测试代码：

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest)
    float MyFloat;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest, meta =
(ShowAsInputPin = "Primary"))
    float MyFloat_Primary;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest, meta =
(ShowAsInputPin = "Advanced"))
    float MyFloat_Advanced;

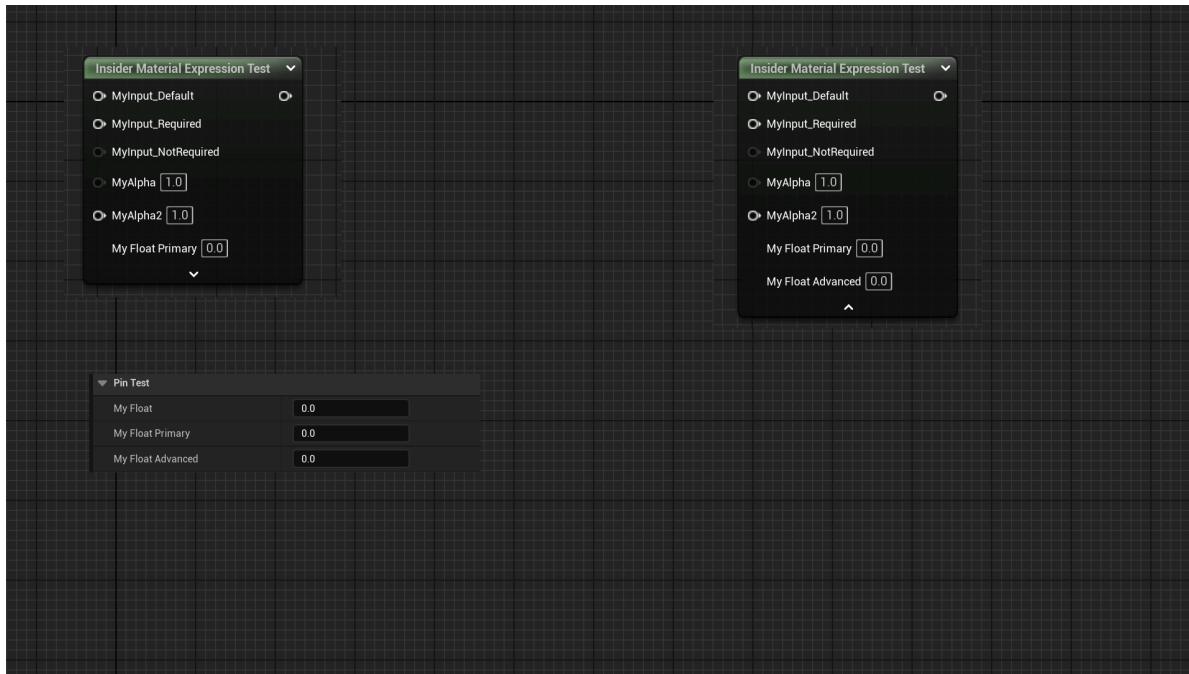
```

测试结果：

可见MyFloat没有显示在节点上。

MyFloat_Primary显示在节点上。

MyFloat_Advanced需要展开箭头后才显示出来。



原理：

遍历UMaterialExpression里的属性，根据含有ShowAsInputPin的类型来生成引脚。

```
TArray<FProperty*> UMaterialExpression::GetPropertyInputs() const
{
    TArray<FProperty*> PropertyInputs;

    static FName OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));
    static FName ShowAsInputPinMetaData(TEXT("ShowAsInputPin"));
    for (TFieldIterator<FProperty> InputIt(GetClass(),
        EFieldIteratorFlags::IncludeSuper, EFieldIteratorFlags::ExcludeDeprecated);
        InputIt; ++InputIt)
    {
        bool bCreateInput = false;
        FProperty* Property = *InputIt;
        // Don't create an expression input if the property is already associated
        // with one explicitly declared
        bool bOverridingInputProperty = Property->
            HasMetaData(OverridingInputPropertyMetaData);
        // It needs to have the 'EditAnywhere' specifier
        const bool bEditAnywhere = Property->HasAnyPropertyParams(CPF_Edit);
        // It needs to be marked with a valid pin category meta data
        const FString ShowAsInputPin = Property->
            GetMetaData>ShowAsInputPinMetaData();
        const bool bShowAsInputPin = ShowAsInputPin == TEXT("Primary") ||
            ShowAsInputPin == TEXT("Advanced");
    }
}
```

```

    if (!bOverridingInputProperty && bEditAnywhere && bShowAsInputPin)
    {
        // Check if the property type fits within the allowed widget types
        FFieldClass* PropertyClass = Property->GetClass();
        if (PropertyClass == FFloatProperty::StaticClass()
            || PropertyClass == FDoubleProperty::StaticClass()
            || PropertyClass == FIntProperty::StaticClass()
            || PropertyClass == FUInt32Property::StaticClass()
            || PropertyClass == FByteProperty::StaticClass()
            || PropertyClass == FBoolProperty::StaticClass()
        )
    }
    {
        bCreateInput = true;
    }
    else if (PropertyClass == FStructProperty::StaticClass())
    {
        FStructProperty* StructProperty = CastField<FStructProperty>
        (Property);
        UScriptStruct* Struct = StructProperty->Struct;
        if (Struct == TBaseStructure<FLinearColor>::Get()
            || Struct == TBaseStructure< FVector4>::Get()
            || Struct == TVariantStructure< FVector4d>::Get()
            || Struct == TBaseStructure< FVector>::Get()
            || Struct == TVariantStructure< FVector3f>::Get()
            || Struct == TBaseStructure< FVector2D>::Get()
        )
    }
    {
        bCreateInput = true;
    }
}
if (bCreateInput)
{
    PropertyInputs.Add(Property);
}
}

return PropertyInputs;
}

```

NiagaraClearEachFrame

- 功能描述:** ScriptStruct /Script/Niagara.NiagaraSpawnInfo
- 使用位置:** USTRUCT
- 引擎模块:** Niagara
- 元数据类型:** bool
- 常用程度:** 0

指定某结构的数据在Niagara后续每一帧不应该读取，只作为初始数据。

当前只用在FNiagaraSpawnInfo上，仅仅内部用。

源码例子：

```
/** Data controlling the spawning of particles */
USTRUCT(BlueprintType, meta = (DisplayName = "Spawn Info", NiagaraClearEachFrame
= "true"))
struct FNiagaraSpawnInfo
{
```

原理：

```
// If the NiagaraClearEachFrame value is set on the data set, we don't bother
// reading it in each frame as we know that it is invalid. However,
// this is only used for the base data set. Other reads are potentially from
// events and are therefore perfectly valid.
if (DataSetIndex == 0 && Var.GetType().GetScriptStruct() != nullptr &&
Var.GetType().GetScriptStruct()-
>GetMetaData(TEXT("NiagaraClearEachFrame")).Equals(TEXT("true"),
ESearchCase::IgnoreCase))
{
    Fmt = VariableName + TEXT("{0} = {4};\n");
}
```

NiagaraInternalType

- 功能描述：**指定该结构的类型为Niagara的内部类型。
- 使用位置：**USTRUCT
- 引擎模块：**Niagara
- 元数据类型：**bool
- 常用程度：**0

指定该结构的类型为Niagara的内部类型。

用来和用户自定义的类型区分。用户不需要自己使用该元数据。

源码例子：

```
USTRUCT(meta = (DisplayName = "Half", NiagaraInternalType = "true"))
struct FNiagaraHalf
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 value = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector2", NiagaraInternalType = "true"))
struct FNiagaraHalfVector2
{
    GENERATED_USTRUCT_BODY()
```

```

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 x = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 y = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector3", NiagaraInternalType = "true"))
struct FNiagaraHalfVector3
{
GENERATED_USTRUCT_BODY()

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 x = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 y = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 z = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector4", NiagaraInternalType = "true"))
struct FNiagaraHalfVector4
{
GENERATED_USTRUCT_BODY()

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 x = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 y = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 z = 0;

UPROPERTY(EditAnywhere, Category = Parameters)
uint16 w = 0;
};

```

原理：

```

#if WITH_EDITORONLY_DATA
bool FNiagaraTypeDefinition::IsInternalType() const
{
    if (const UScriptStruct* ScriptStruct = GetScriptStruct())
    {
        return ScriptStruct->GetBoolMetaData(TEXT("NiagaraInternalType"));
    }

    return false;
}
#endif

```

AllowPreserveRatio

- **功能描述:** 在细节面板上为FVector属性添加一个比率锁。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FVector
- **常用程度:** ★★★

在细节面板上为FVector属性添加一个比率锁。

测试代码:

```
public:  
    UPROPERTY(EditAnywhere, Category = VectorTest)  
    FVector MyVector_Default;  
  
    UPROPERTY(EditAnywhere, Category = VectorTest, meta = (AllowPreserveRatio))  
    FVector MyVector_AllowPreserveRatio;  
  
    UPROPERTY(EditAnywhere, Category = VectorTest, meta = (ShowNormalize))  
    FVector MyVector_ShowNormalize;
```

测试结果:

可见MyVector_AllowPreserveRatio的值在锁上之后可以形成固定的比率。



原理:

其实就是UI定制化的时候检测出AllowPreserveRatio就创建单独的UI。

```
void FMathStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>&  
StructPropertyHandle, FDetailWidgetRow& Row)  
{  
    if (StructPropertyHandle->HasMetaData("AllowPreserveRatio"))  
    {  
        if (!GConfig->GetBool(TEXT("SelectionDetails"), *  
(StructPropertyHandle->GetProperty()->GetName() + TEXT("_PreserveScaleRatio")),  
bPreserveScaleRatio, GEditorPerProjectIni))  
        {  
            bPreserveScaleRatio = true;  
        }  
  
        HorizontalBox->AddSlot()  
        .Autowidth()  
        .Maxwidth(18.0f)  
        .VAlign(VAlign_Center)  
        [  
            // Add a checkbox to toggle between preserving the ratio of x,y,z  
            components of scale when a value is entered
```

```

        SNew(SCheckBox)
            .IsChecked(this,
&FMathStructCustomization::IsPreserveScaleRatioChecked)
            .OnCheckStateChanged(this,
&FMathStructCustomization::OnPreserveScaleRatioToggled, StructWeakHandlePtr)
                .Style(FAppStyle::Get(), "TransparentCheckBox")
                .ToolTipText(LOCTEXT("PreserveScaleToolTip", "When locked, scales
uniformly based on the current xyz scale values so the object maintains its shape
in each direction when scaled"))
            [
                SNew(SImage)
                    .Image(this,
&FMathStructCustomization::GetPreserveScaleRatioImage)
                    .ColorAndOpacity(FSlateColor::UseForeground())
            ]
        ];
    }

}

```

ArrayClamp

- 功能描述:** 限定整数属性的值必须在指定数组的合法下标范围内, [0,ArrayClamp.Size()-1]
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** int32
- 限制类型:** int32
- 常用程度:** ★★★

限定整数属性的值必须在指定数组的合法下标范围内, [0,ArrayClamp.Size()-1]

测试代码:

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest)
    int32 MyInt_NoArrayClamp = 0;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest, meta =
(ArrayClamp = "MyIntArray"))
    int32 MyInt_HasArrayClamp = 0;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest)
    TArray<int32> MyIntArray;

```

测试效果:

可见拥有ArrayClamp的整数值被限制在数组的下标中。



原理：

根据指定的数组名称在本类里寻找到Array属性，然后把本整数属性的值Clamp在该数组的下标范围内。

```
template <typename Type>
static Type ClampIntegerValueFromMetaData(Type InValue, FPropertyHandleBase&
InPropertyHandle, FPropertyNode& InPropertyName)
{
    Type RetVal = ClampValueFromMetaData<Type>(InValue, InPropertyHandle);

    //enforce array bounds
    const FString& ArrayClampString =
InPropertyHandle.GetMetaData(TEXT("ArrayClamp"));
    if (ArrayClampString.Len())
    {
        FObjectPropertyNode* ObjectPropertyNode =
InPropertyName.FindObjectItemParent();
        if (ObjectPropertyNode && ObjectPropertyNode->GetNumObjects() == 1)
        {
            Type LastValidIndex = static_cast<Type>
(GetArrayPropertyLastValidIndex(ObjectPropertyNode, ArrayClampString));
            RetVal = FMath::Clamp<Type>(RetVal, 0, LastValidIndex);
        }
        else
        {
            UE_LOG(LogPropertyName, Warning, TEXT("Array Clamping isn't supported
in multi-select (Param Name: %s)"), *InPropertyHandle.GetProperty()->GetName());
        }
    }

    return RetVal;
}
```

ClampMax

- **功能描述:** 指定数字输入框实际接受的最大值
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** float/int
- **限制类型:** float,int32
- **关联项:** UIMin
- **常用程度:** ★★★★☆

ClampMin

- **功能描述:** 指定数字输入框实际接受的最小值
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** float/int

- **限制类型:** float,int32
- **关联项:** UIMin
- **常用程度:** ★★★★☆

ColorGradingMode

- **功能描述:** 使得一个FVector4属性成为颜色显示
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** string="abc"
- **限制类型:** FVector4
- **常用程度:** ★★

使得一个FVector4属性成为颜色显示。因为FVector4和RGBA正好对应。

必须配合UIMin, UIMax才能使用，否则会崩溃，因为FColorGradingVectorCustomization里直接对UIMinValue直接取值。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = ())
FVector4 MyVector4_NotColor;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "saturation"))
FVector4 MyVector4_Saturation;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "contrast"))
FVector4 MyVector4_Contrast;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "gamma"))
FVector4 MyVector4_Gamma;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "gain"))
FVector4 MyVector4_Gain;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "offset"))
FVector4 MyVector4_Offset;
```

测试效果：

可以发现没有ColorGradingMode 的依然是普通的FVector4，否则就会用颜色转盘来显示编辑。

▼ Color Grading Mode Test

▼ My Vector 4 Not Color

	0.0	0.0	0.0	1.0
X	0.0			
Y	0.0			
Z	0.0			
W	1.0			

▼ My Vector 4 Saturation



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Contrast



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Gamma



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Gain



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Offset



0.0

R	0.0	G 0.0	B 0.0	Y 1.0
---	-----	-------	-------	-------

R	0.0			
G	0.0			
B	0.0			
Y	1.0			

原理：

如果是FVector4属性，再判断如果有ColorGradingMode，则创建FColorGradingVectorCustomization来把FVector定制化会颜色显示。再根据字符串判断EColorGradingModes，最后创建相应的具体UI控件。

```
void FVector4StructCustomization::CustomizeChildren(TSharedRef<IPropertyHandle>
StructPropertyHandle, IDetailChildrenBuilder& StructBuilder,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    FProperty* Property = StructPropertyHandle->GetProperty();
    if (Property)
    {
        const FString& ColorGradingModeString = Property-
>GetMetaData(TEXT("ColorGradingMode"));
        if (!ColorGradingModeString.IsEmpty())
        {
            //Create our color grading customization shared pointer
            TSharedPtr<FColorGradingVectorCustomization>
ColorGradingCustomization =
GetOrCreateColorGradingVectorCustomization(StructPropertyHandle);

            //Customize the childrens
            ColorGradingVectorCustomization->CustomizeChildren(StructBuilder,
StructCustomizationUtils);

            // we handle the customize children so just return here
            return;
        }
    }

    //Use the base class customize children
    FMathStructCustomization::CustomizeChildren(StructPropertyHandle,
StructBuilder, StructCustomizationUtils);
}

EColorGradingModes FColorGradingVectorCustomizationBase::GetColorGradingMode()
const
{
    EColorGradingModes ColorGradingMode = EColorGradingModes::Invalid;

    if (ColorGradingPropertyHandle.IsValid())
    {
        //Query all meta data we need
        FProperty* Property = ColorGradingPropertyHandle.Pin()->GetProperty();
        const FString& ColorGradingModeString = Property-
>GetMetaData(TEXT("ColorGradingMode"));

        if (ColorGradingModeString.Len() > 0)
        {
            if (ColorGradingModeString.Compare(TEXT("saturation")) == 0)
            {
                ColorGradingMode = EColorGradingModes::Saturation;
            }
        }
    }
}
```

```

        }
        else if (ColorGradingModeString.Compare(TEXT("contrast")) == 0)
        {
            ColorGradingMode = EColorGradingModes::Contrast;
        }
        else if (ColorGradingModeString.Compare(TEXT("gamma")) == 0)
        {
            ColorGradingMode = EColorGradingModes::Gamma;
        }
        else if (ColorGradingModeString.Compare(TEXT("gain")) == 0)
        {
            ColorGradingMode = EColorGradingModes::Gain;
        }
        else if (ColorGradingModeString.Compare(TEXT("offset")) == 0)
        {
            ColorGradingMode = EColorGradingModes::Offset;
        }
    }

    return ColorGradingMode;
}

```

CtrlMultiplier

- 功能描述:** 指定数字输入框在Ctrl按下时鼠标轮滚动和鼠标拖动改变值的倍率。
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 限制类型:** 数据结构: FVector, FRotator, FColor
- 关联项:** ShiftMultiplier
- 常用程度:** ★★

指定数字输入框在Ctrl按下时鼠标轮滚动和鼠标拖动改变值的倍率。

- CtrlMultiplier的默认值是0.1f，一般作为一种精调模式。
- 直接设置到float属性上并无效果。默认情况下，属性上的CtrlMultiplier并不会设置到SNumericEntryBox和SSpinBox上，因为这二者并不会直接从property上提取meta来设置到其本身的Multiplier值上。
- 在源码里发现FMathStructCustomization里会提取CtrlMultiplier和ShiftMultiplier的值，因此我们可以在一些数学结构上设置，如FVector, FRotator, FColor
- 如果自己定义Customization和创建SSpinBox，则可以自己提取Multiplier的值自己设置到控件里去。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (CtrlMultiplier = "5"))
    float MyFloat_HasCtrlMultiplier = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (ShiftMultiplier = "100"))
    float MyFloat_HasShiftMultiplier = 100;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest)
FVector MyVector_NoMultiplier;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (CtrlMultiplier = "5"))
    FVector MyVector_HasCtrlMultiplier;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (ShiftMultiplier = "100"))
    FVector MyVector_HasShiftMultiplier;
```

测试效果：

- 发现普通float属性测试并无效果，按下Ctrl和Shift依然是默认的改变值0.1和10.f
- 普通的默认FVector，按下Ctrl和Shift也是默认的改变值0.1和10.f
- MyVector_HasCtrlMultiplier，发现按下Ctrl，一下子改变幅度是5
- MyVector_HasShiftMultiplier，发现按下Shift，一下子改变幅度是100
- 当然用鼠标拖动也是一样的效果，只是那样改变太过剧烈，演示效果不明显

原理：

SNumericEntryBox的构造函数里：

```
, _ShiftMultiplier(10.f)
, _CtrlMultiplier(0.1f)
```

```
void FMathStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>& StructPropertyHandle, FDetailWidgetRow& Row)
{
    for (int32 ChildIndex = 0; ChildIndex < SortedChildHandles.Num(); ++ChildIndex)
    {
        TSharedRef<IPROPERTYHANDLE> ChildHandle = SortedChildHandles[ChildIndex];

        // Propagate metadata to child properties so that it's reflected in the nested, individual spin boxes
        ChildHandle->SetInstanceMetaData(TEXT("UIMin"), StructPropertyHandle->GetMetaData(TEXT("UIMin")));
        ChildHandle->SetInstanceMetaData(TEXT("UIMax"), StructPropertyHandle->GetMetaData(TEXT("UIMax")));
        ChildHandle->SetInstanceMetaData(TEXT("SliderExponent"),
StructPropertyHandle->GetMetaData(TEXT("SliderExponent")));
        ChildHandle->SetInstanceMetaData(TEXT("Delta"), StructPropertyHandle->GetMetaData(TEXT("Delta")));
```

```

    ChildHandle->SetInstanceMetaData(TEXT("LinearDeltaSensitivity"),
StructPropertyHandle->GetMetaData(TEXT("LinearDeltaSensitivity")));
    ChildHandle->SetInstanceMetaData(TEXT("ShiftMultiplier"),
StructPropertyHandle->GetMetaData(TEXT("ShiftMultiplier")));
    ChildHandle->SetInstanceMetaData(TEXT("CtrlMultiplier"),
StructPropertyHandle->GetMetaData(TEXT("CtrlMultiplier")));
    ChildHandle->SetInstanceMetaData(TEXT("SupportDynamicsSlider.MaxValue"),
StructPropertyHandle->GetMetaData(TEXT("SupportDynamicsSlider.MaxValue")));
    ChildHandle->SetInstanceMetaData(TEXT("SupportDynamicsSlider.MinValue"),
StructPropertyHandle->GetMetaData(TEXT("SupportDynamicsSlider.MinValue")));
    ChildHandle->SetInstanceMetaData(TEXT("ClampMin"), StructPropertyHandle-
>GetMetaData(TEXT("ClampMin")));
    ChildHandle->SetInstanceMetaData(TEXT("ClampMax"), StructPropertyHandle-
>GetMetaData(TEXT("ClampMax")));
}
}

```

Delta

- 功能描述:** 设定数字输入框值改变的幅度为Delta的倍数
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 限制类型:** float,int32
- 关联项:** LinearDeltaSensitivity
- 常用程度:** ★★★

设定数字输入框值改变的幅度为Delta的倍数。

注意的事项是：

1. 在全局使得变化值成为Delta的倍数
2. Delta默认值是0，这个时候代表没有设置，数字会指数改变。
3. 注意和WheelStep的区别是，Delta在鼠标左右拖动和按下键盘方向键的时候生效，WheelStep是只在鼠标滚轮变化的时候生效。二者虽然都是用来控制变化幅度，但作用范围不同。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest)
float MyFloat_DefaultDelta = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10))
float MyFloat_Delta10 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10))
float MyFloat_Delta10_UIMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50_UIMinMax = 100;
```

测试效果：

- MyFloat_DefaultDelta， 默认情况下鼠标往右拖动变化很剧烈，变化呈指数上升。
- MyFloat_Delta10， 鼠标往右拖动变化也很剧烈（最终到达的值也很巨大），但是变化始终以Delta为步幅。
- MyFloat_Delta10_UIMinMax， 限定了UIMinMax，导致最大值受限，但是变化其实是按照比例值线性（SliderExponent默认是1没有改变）。
- MyFloat_Delta10_LinearDeltaSensitivity50，在没有UIMinMax的情况下，且同时设置了LinearDeltaSensitivity，会导致鼠标往右拖动的整个过程中变化值始终是线性。LinearDeltaSensitivity越大越不敏感。因此一次一次缓慢的改变10
- MyFloat_Delta10_LinearDeltaSensitivity50_UIMinMax， 如果上面一个再加上UIMinMax，则发现又失去了LinearDeltaSensitivity的效果。因为LinearDeltaSensitivity不能在有滚动条的情况下生效。



原理：

- 在方向键上下左右按键的时候每次变化正负Delta，但是OnKeyDown没有直接绑定，所以默认情况下按键只会触发焦点的转移。
- 基础step是0.1或1，如果没受到CtrlMultiplier和ShiftMultiplier的影响。
- 默认情况下，鼠标左右移动改变的差量中的指数部分是FMath::Pow((double)CurrentValue, (double)SliderExponent.Get())，SliderExponent默认是1，也就是越往左右两端改变的幅度越大。
- 在同时设置LinearDeltaSensitivity和Delta的情况下，鼠标左右移动改变的差量中的指数部分是FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get())，即改变的差量在左右整个数轴上是线性一致的。
- 在最后提交值的时候，改变的差量值会被规范到Delta的倍数上来。

```
, _Delta(0)
virtual FReply SSpinBox<NumericType>::OnMouseMove(const FGeometry& MyGeometry,
const FPointerEvent& MouseEvent) override
{
```

```

if (bUnlimitedSpinRange)
{
    // If this control has a specified delta and sensitivity then we
    // use that instead of the current value for determining how much to change.
    const double Sign = (MouseEvent.GetCursorDelta().X > 0) ? 1.0 :
-1.0;

    if (LinearDeltaSensitivity.IsSet() &&
LinearDeltaSensitivity.Get() != 0 && Delta.IsSet() && Delta.Get() > 0)
    {
        const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / (float)LinearDeltaSensitivity.Get());
        NewValue = InternalValue + (Sign * MouseDelta * FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get())) * Step;
    }
    else
    {
        const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / SliderWidthInSlateUnits);
        const double CurrentValue = FMath::Clamp<double>
(FMath::Abs(InternalValue), 1.0,
(double)std::numeric_limits<NumericType>::max());
        NewValue = InternalValue + (Sign * MouseDelta * FMath::Pow((double)CurrentValue, (double)SliderExponent.Get())) * Step;
    }
}
}

virtual FReply SSpinBox<NumericType>::OnKeyDown(const FGeometry& MyGeometry,
const FKeyEvent& InKeyEvent) override
{
    else if (Key == EKeys::Up || Key == EKeys::Right)
    {
        const NumericType LocalValueAttribute = ValueAttribute.Get();
        const NumericType LocalDelta = Delta.Get();
        InternalValue = (double)LocalValueAttribute;
        CommitValue(LocalValueAttribute + LocalDelta, Internalvalue +
(double)LocalDelta, CommittedViaArrowKey, ETextCommit::OnEnter);
        ExitTextMode();
        return FReply::Handled();
    }
    else if (Key == EKeys::Down || Key == EKeys::Left)
    {
        const NumericType LocalValueAttribute = ValueAttribute.Get();
        const NumericType LocalDelta = Delta.Get();
        Internalvalue = (double)LocalValueAttribute;
        CommitValue(LocalValueAttribute - LocalDelta, Internalvalue +
(double)LocalDelta, CommittedViaArrowKey, ETextCommit::OnEnter);
        ExitTextMode();
        return FReply::Handled();
    }
}

void SSpinBox<NumericType>::CommitValue(NumericType NewValue, double
NewSpinValue, ECommitMethod CommitMethod, ETextCommit::Type OriginalCommitInfo)
{

```

```

    // If needed, round this value to the delta. Internally the value is not held
    // to the delta but externally it appears to be.
    if (CommitMethod == CommittedViaSpin || CommitMethod == CommittedViaArrowKey
    || bAlwaysUsesDeltaSnap)
    {
        NumericType CurrentDelta = Delta.Get();
        if (CurrentDelta != NumericType())
        {
            NewValue = FMath::GridSnap<NumericType>(NewValue, CurrentDelta); ///
            // snap numeric point value to nearest Delta
        }
    }
}

```

ForceUnits

- **功能描述:** 固定设定属性值的单位保持不变，不根据数值动态调整显示单位。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** string="abc"
- **限制类型:** float,int32
- **关联项:** Units
- **常用程度:** ★★★

HideAlphaChannel

- **功能描述:** 使FColor或FLinearColor属性在编辑的时候隐藏Alpha通道。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FColor , FLinearColor
- **常用程度:** ★★★

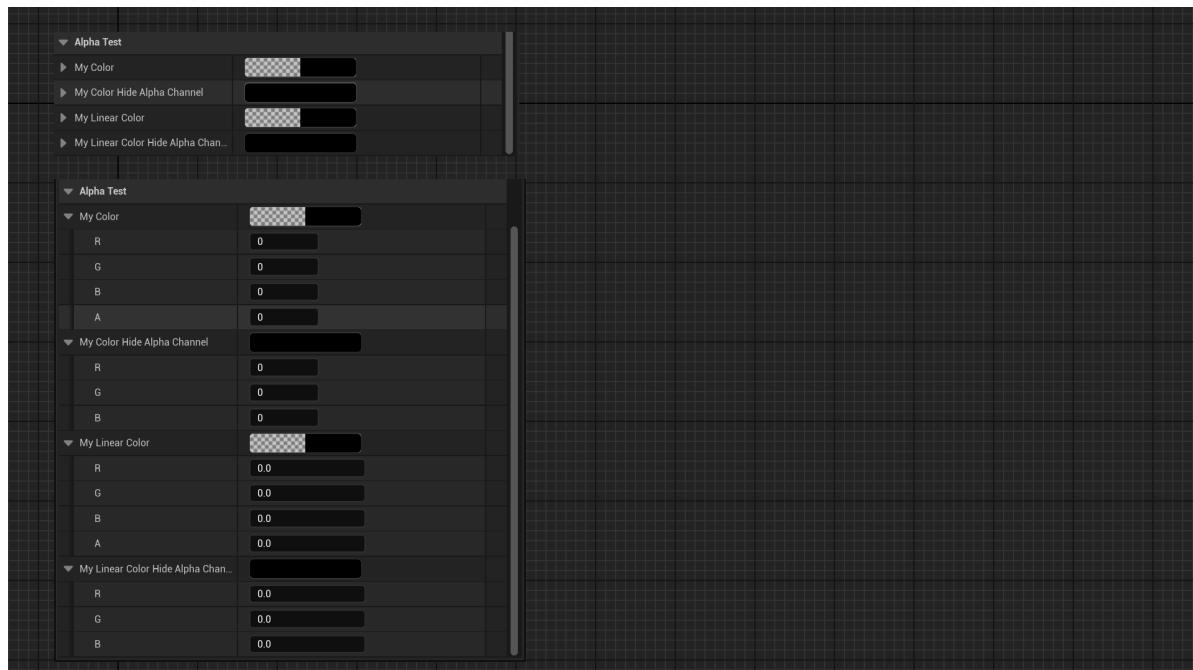
使FColor或FLinearColor属性在编辑的时候隐藏Alpha通道。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, Category = AlphaTest)  
    FColor MyColor;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest, meta = (HideAlphaChannel))  
    FColor MyColor_HideAlphaChannel;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest)  
    FLinearColor MyLinearColor;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest, meta = (HideAlphaChannel))  
    FLinearColor MyLinearColor_HideAlphaChannel;
```

测试结果：

可见带有HideAlphaChannel的属性就没有了Alpha通道。



原理：

```
void FColorStructCustomization::CustomizeHeader(TSharedRef<class IPropertyHandle>  
InStructPropertyHandle, class FDetailWidgetRow& InHeaderRow,  
IPropertyTypeCustomizationUtils& StructCustomizationUtils)  
{  
    bIgnoreAlpha = TypeSupportsAlpha() == false || StructPropertyHandle->  
GetProperty()->HasMetaData(TEXT("HideAlphaChannel"));  
}  
  
.AlphaDisplayMode(bIgnoreAlpha ? EColorBlockAlphaDisplayMode::Ignore :  
EColorBlockAlphaDisplayMode::Separate)
```

InlineColorPicker

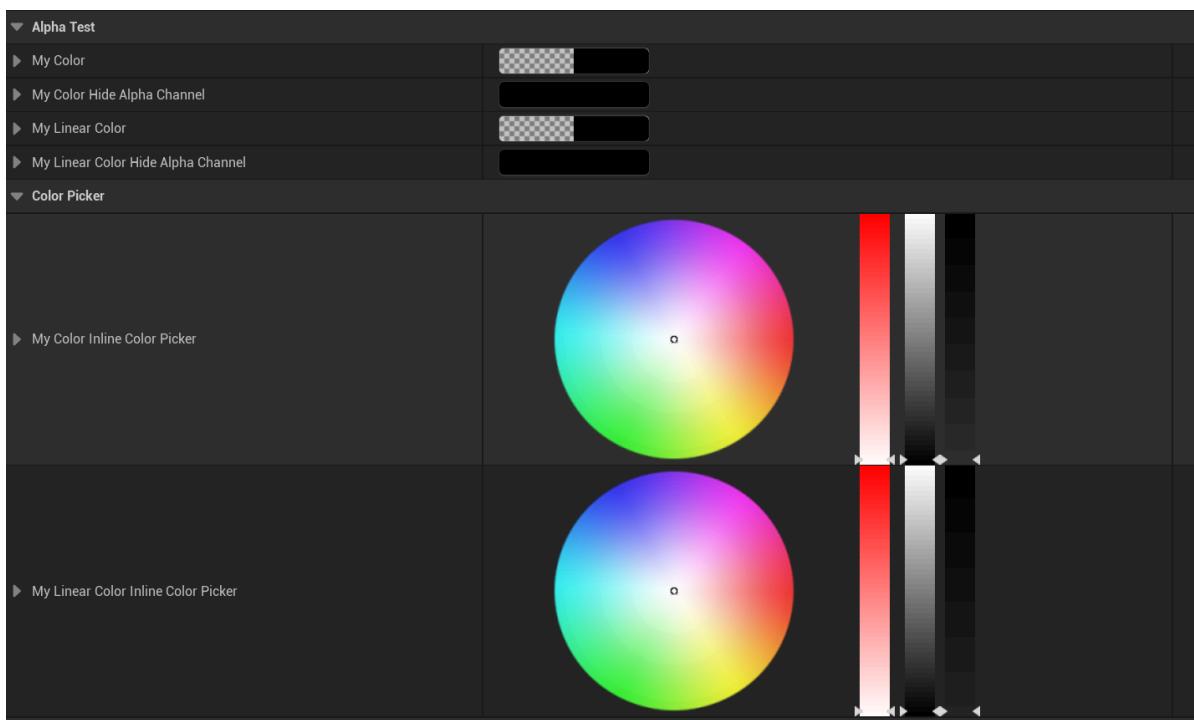
- **功能描述:** 使FColor或FLinearColor属性在编辑的时候直接内联一个颜色选择器。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FColor , FLinearColor
- **常用程度:** ★★

使FColor或FLinearColor属性在编辑的时候直接内联一个颜色选择器。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, Category = ColorPicker, meta = (InlineColorPicker))  
    FColor MyColor_InlineColorPicker;  
    UPROPERTY(EditAnywhere, Category = ColorPicker, meta = (InlineColorPicker))  
    FLinearColor MyLinearColor_InlineColorPicker;
```

测试结果：



原理：

根据不同的标记创建不同的的ColorWidget。

```

void FColorStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>&
InStructPropertyHandle, FDetailWidgetRow& Row)
{
    if (InStructPropertyHandle->HasMetaData("InlineColorPicker"))
    {
        ColorWidget = CreateInlineColorPicker(StructWeakHandlePtr);
        ContentWidth = 384.0f;
    }
    else
    {
        ColorWidget = CreateColorWidget(StructWeakHandlePtr);
    }
}

```

LinearDeltaSensitivity

- 功能描述:** 在设定Delta后，进一步设定数字输入框变成线性改变以及改变的敏感度（值越大越不敏感）
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 限制类型:** float,int32
- 关联项:** Delta
- 常用程度:** ★★★

生效的条件：

1. 先设置Delta>0
2. 不设置UIMin, UIMax
3. 设定LinearDeltaSensitivity >0

测试代码：

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10))
float MyFloat_Delta10_UIMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50_UIMinMax = 100;

```

测试效果：

效果解析请参见：Delta的解析

▼ Delta Test	
My Float Default Delta	100.0
My Float Delta 10	100.0
My Float Delta 10 UIMin Max	100.0
My Float Delta 10 Linear Delta Sensitivity 50	100.0
My Float Delta 10 Linear Delta Sensitivity 50 UIMin Max	100.0

原理：

可见只有没有UIMinMax且已经设置Delta后才能走进线性改变的代码分支。

```
, _Delta(0)
virtual FReply SSpinBox<NumericType>::OnMouseMove(const FGeometry& MyGeometry,
const FPointerEvent& MouseEvent) override
{
    if (bUnlimitedSpinRange)
    {
        // If this control has a specified delta and sensitivity then we
        // use that instead of the current value for determining how much to change.
        const double sign = (MouseEvent.GetCursorDelta().X > 0) ? 1.0 :
-1.0;

        if (LinearDeltaSensitivity.IsSet() &&
LinearDeltaSensitivity.Get() != 0 && Delta.IsSet() && Delta.Get() > 0)
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / (float)LinearDeltaSensitivity.Get());
            NewValue = InternalValue + (Sign * MouseDelta *
FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get())) * Step;
        }
        else
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / SliderWidthInlateUnits);
            const double currentValue = FMath::Clamp<double>
(FMath::Abs(InternalValue), 1.0,
(double)std::numeric_limits<NumericType>::max());
            NewValue = InternalValue + (Sign * MouseDelta *
FMath::Pow((double)currentValue, (double)SliderExponent.Get())) * Step;
        }
    }
}
```

Multiple

- **功能描述:** 指定数字的值必须是Mutliple提供的值的整数倍。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** int32

- **限制类型:** int32
- **常用程度:** ★★★

指定数字的值必须是Mutliple提供的值的整数倍。

测试代码:

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultipleTest)
    int32 MyInt_NoMultiple = 100;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultipleTest, meta =
(Multiple = 5))
    int32 MyInt_HasMultiple = 100;
```

蓝图效果:

可以看到，拥有Multiple 的只能按照5的倍数来增长。



原理:

```
template <typename Type>
static Type ClampIntegerValueFromMetaData(Type InValue, FPropertyHandleBase&
InPropertyParams, FPropertyParams& InPropertyParams)
{
    Type RetVal = ClampValueFromMetaData<Type>(InValue, InPropertyParams);

    //if there is "Multiple" meta data, the selected number is a multiple
    const FString& MultipleString =
InPropertyParams.GetMetaData(TEXT("Multiple"));
    if (MultipleString.Len())
    {
        check(MultipleString.IsNumeric());
        Type MultipleValue;
        TTTypeFromString<Type>::FromString(MultipleValue, *MultipleString);
        if (MultipleValue != 0)
        {
            RetVal -= Type(RetVal) % MultipleValue;
        }
    }

    return RetVal;
}
```

NoSpinbox

- **功能描述:** 使数值属性禁止默认的拖放和滚轮的UI编辑功能，数值属性包括int系列以及float系列。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property

- **元数据类型:** bool
- **限制类型:** Numeric Type, int / float
- **常用程度:** ★★

使数值属性禁止默认的拖放和滚轮的UI编辑功能，数值属性包括int系列以及float系列。

测试代码：

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest, meta =
    (NoSpinbox = true))
    int32 MyInt_NoSpinbox = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest)
    float MyFloat = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest, meta =
    (NoSpinbox = true))
    float MyFloat_NoSpinbox = 123;
```

测试效果：

发现带有NoSpinbox 的属性不能用鼠标左右拖动改变数值，也不能用鼠标滚轮改变数值。



原理：

可以看到针对数值的UI， bAllowSpin的功能直接决定了Widget的AllowWheel和AllowSpin功能。

```
virtual TSharedRef<SWidget> GetDefaultValueWidget() override
{
    const typename TNumericPropertyParams<NumericType>::FMetaDataGetter
    MetaDataGetter =
    TNumericPropertyParams<NumericType>::FMetaDataGetter::CreateLambda([&] (const
    FName& Key)
    {
        return (PinProperty) ? PinProperty->GetMetaData(Key) : FString();
    });

    TNumericPropertyParams<NumericType> NumericPropertyParams(PinProperty,
    PinProperty ? MetaDataGetter : nullptr);

    const bool bAllowSpin = !(PinProperty && PinProperty-
    >GetBoolMetaData("NoSpinbox"));

    // Save last committed value to compare when value changes
    LastSliderCommittedValue = GetNumericValue().GetValue();

    return SNew(SBox)
        .MinDesiredWidth(MinDesiredBoxWidth)
```

```

    .MaxDesiredWidth(400)
    [
        SNew(SNumericEntryBox<NumericType>)
        .EditableTextBoxStyle(FAppStyle::Get(), "Graph.EditableTextBox")
        .BorderForegroundColor(FSlateColor::UseForeground())
        .Visibility(this, &SGraphPinNumSlider::GetDefaultValueVisibility)
        .IsEnabled(this, &SGraphPinNumSlider::GetDefaultValueIsEditable)
        .value(this, &SGraphPinNumSlider::GetNumericValue)
        .MinValue(NumericPropertyParams.MinValue)
        .MaxValue(NumericPropertyParams.MaxValue)
        .MinSliderValue(NumericPropertyParams.MinSliderValue)
        .MaxSliderValue(NumericPropertyParams.MaxSliderValue)
        .SliderExponent(NumericPropertyParams.SliderExponent)
        .Delta(NumericPropertyParams.Delta)

        .LinearDeltaSensitivity(NumericPropertyParams.GetLinearDeltaSensitivityAttribute())
    )
        .AllowWheel(bAllowSpin)
        .WheelStep(NumericPropertyParams.WheelStep)
        .AllowSpin(bAllowSpin)
        .OnValueCommitted(this, &SGraphPinNumSlider::OnValueCommitted)
        .OnValueChanged(this, &SGraphPinNumSlider::OnValueChanged)
        .OnBeginSliderMovement(this,
        &SGraphPinNumSlider::OnBeginSliderMovement)
        .OnEndSliderMovement(this, &SGraphPinNumSlider::OnEndSliderMovement)
    ];
}

```

ShiftMultiplier

- 功能描述:** 指定数字输入框在Shift按下时鼠标轮滚动和鼠标拖动改变值的倍率。
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 限制类型:** 数据结构: FVector, FRotator, FColor
- 关联项:** CtrlMultiplier
- 常用程度:** ★★

默认值是10.f

Shift的模式可以认为是一种快调模式，可以快速的改变值。

ShowNormalize

- 功能描述:** 使得FVector变量在细节面板出现一个正规化的按钮。
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** bool
- 限制类型:** FVector
- 常用程度:** ★★★

使得FVector变量在细节面板出现一个正规化的按钮。

测试代码：

```
UPROPERTY(EditAnywhere, Category = VectorTest)
FVector MyVector_Default;

UPROPERTY(EditAnywhere, Category = VectorTest, meta = (AllowPreserveRatio))
FVector MyVector_AllowPreserveRatio;

UPROPERTY(EditAnywhere, Category = VectorTest, meta = (ShowNormalize))
FVector MyVector_ShowNormalize;
```

测试结果：

MyVector_ShowNormalize右侧的按钮可以把值正规化。



原理：

其实就是UI定制化的时候检测出ShowNormalize就创建单独的UI。

```
if (StructPropertyHandle->HasMetaData("ShowNormalize") &&
MathStructCustomization::IsFloatVector(StructPropertyHandle))
{
    HorizontalBox->AddSlot()
        .AutoWidth()
        .MaxWidth(18.0f)
        .VAlign(VAlign_Center)
        [
            // Add a button to scale the vector uniformly to achieve a unit
vector
            SNew(SButton)
                .OnClicked(this, &FMathStructCustomization::OnNormalizeClicked,
structweakHandlePtr)
                    .ButtonStyle(FAppStyle::Get(), "NoBorder")
                    .ToolTipText(LOCTEXT("NormalizeToolTip", "when clicked, if the
vector is large enough, it scales the vector uniformly to achieve a unit vector
(vector with a length of 1)"))
            [
                SNew(SImage)
                    .ColorAndOpacity(FSlateColor::UseForeground())
                    .Image(FAppStyle::GetBrush(TEXT("Icons.Normalize")))
            ]
        ];
}
```

SliderExponent

- 功能描述：**指定数字输入框上滚动条拖动的变化指数分布
- 使用位置：** UPROPERTY
- 引擎模块：** Numeric Property

- **元数据类型:** float/int
- **限制类型:** float,int32
- **常用程度:** ★★★★★

指定数字输入框上滚动条拖动的变化指数分布。默认值是1。

该值必须配合Min, Max使用

所谓指数分布指的是在UIMin和Max的范围内，当滚动的文本值变化的时候，滚动条的百分比值如何变化。默认情况下，中间点的值就是在50%。但我们也同样可以指定一个指数，形成一条指数分布曲线，在数轴左侧一开始的时候比较平缓变动比较缓慢拥有更高的调整精度，在数轴的右侧结束的部分曲线变得陡峭变动剧烈就失去了精度。

测试代码：

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = sliderTest, meta =
(UIMin = "0", UIMax = "1000"))
    float MyFloat_DefaultSliderExponent = 100;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = sliderTest, meta =
(UIMin = "0", UIMax = "1000", SliderExponent = 5))
    float MyFloat_HasSliderExponent = 100;
```

测试效果：

可见SliderExponent=5的效果导致100的文本值落在UI 1000的范围内一开始处于快0.3的位置，且变动的范围在500前比较精细，500后比较迅速。跟前者形成对比。



原理：

默认的值是1，如果不是，则采用SpinBoxComputeExponentSliderFraction来计算新的百分比，读者可自行观察SpinBoxComputeExponentSliderFraction函数来了解指数分布的情况。

```
const float CachedSliderExponent = SliderExponent.Get();
if (!FMath::IsNearlyEqual(CachedSliderExponent, 1.f))
{
    if (SliderExponentNeutralValue.IsSet() && SliderExponentNeutralValue.Get() >
GetMinSliderValue() && SliderExponentNeutralValue.Get() < GetMaxSliderValue())
    {
        //Compute a log curve on both side of the neutral value
        float StartFractionFilled =
Fraction((double)SliderExponentNeutralValue.Get(), (double)GetMinSliderValue(),
(double)GetMaxSliderValue());
        FractionFilled = SpinBoxComputeExponentSliderFraction(FractionFilled,
StartFractionFilled, CachedSliderExponent);
    }
    else
    {
        FractionFilled = 1.0f - FMath::Pow(1.0f - FractionFilled,
CachedSliderExponent);
    }
}
```

sRGB

- **功能描述:** 使FColor或FLinearColor属性在编辑的时候采用sRGB方式。
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FColor , FLinearColor

使FColor或FLinearColor属性在编辑的时候采用sRGB方式。

但是在测试的时候并无法工作。

原理:

```
void FColorStructCustomization::CustomizeHeader(TSharedRef<class IPropertyHandle>
InStructPropertyHandle, class FDetailWidgetRow& InHeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    if (StructPropertyHandle->GetProperty()->HasMetaData(TEXT("sRGB")))
    {
        SRGBOverride = StructPropertyHandle->GetProperty()->
GetBoolMetaData(TEXT("sRGB"));
    }
}
```

SupportDynamicSlider.MaxValue

- **功能描述:** 支持数字输入框上滚动条的最大范围值在Alt按下时被动态改变
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FVector4
- **关联项:** SupportDynamicSliderMinValue
- **常用程度:** ★

SupportDynamicSlider.MinValue

- **功能描述:** 支持数字输入框上滚动条的最小范围值在Alt按下时被动态改变
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** bool
- **限制类型:** FVector4
- **关联项:** SupportDynamicSlider.MaxValue

- 常用程度：★

支持数字输入框上滚动条的最小范围值在Alt按下时被动态改变。

- 必须配合UIMin, UIMax使用，因为这样才有滚动条UI
- 一般情况下滚动条范围是初始设置好的，但该设置支持动态改变。方法是按下Alt拖动鼠标。
- 普通的float属性等是不支持该meta的，因为普通自动生成的的SPropertyEditorNumeric，没有从property meta中提取SupportDynamicSliderMinValue，因此即使设置了也不会生效。
- 目前源码中只有继承了FMathStructCustomization的FColorGradingVectorCustomizationBase (对应 FVector4) 才提取SupportDynamicSliderMinValue，然后自己创建SNumericEntryBox，从而正确的设置SupportDynamicSliderMinValue的值。
- 因此如果你也想使得自己的结构里的数字属性支持该功能，也需要手动创建Customization，然后在里面手动创建SNumericEntryBox，从而自己设置SupportDynamicSliderMinValue的值。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DynamicsliderTest,
meta = (UIMin = "0", UIMax = "1"))
FVector4 MyVector4_NoDynamicSlider;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DynamicsliderTest,
meta = (UIMin = "0", UIMax = "1", SupportDynamicSliderMinValue = "true",
SupportDynamicSliderMaxValue = "true"))
FVector4 MyVector4_SupportDynamicSlider;
```

测试结果：

可以看见MyVector4_NoDynamicSlider并无法更改0-1的滚动条范围。而
MyVector4_SupportDynamicSlider在按下Alt同时拖动鼠标后可以改变最小和最大的UI范围。



原理：

在SPropertyEditorNumeric里并不会去提取SupportDynamicSliderMinValue该属性，因此默认情况下该值是不生效的。

```
void SPropertyEditorNumeric<NumericType>::Construct( const FArguments& InArgs,
const TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    TNumericPropertyParams<NumericType> NumericPropertyParams(Property,
MetaDataTable);
    ChildSlot
    [
        SAssignNew(PrimaryWidget, SNumericEntryBox<NumericType>)
        // Only allow spinning if we have a single value
        .AllowSpin(bAllowSpin)
        .value(this, &SPropertyEditorNumeric<NumericType>::OnGetValue)
        .Font(InArgs._Font)
        .MinValue(NumericPropertyParams.MinValue)
        .MaxValue(NumericPropertyParams.MaxValue)
        .MinSliderValue(NumericPropertyParams.MinSliderValue)
        .MaxSliderValue(NumericPropertyParams.MaxSliderValue)
    ];
}
```

```

        .SliderExponent(NumericPropertyParams.SliderExponent)
        .Delta(NumericPropertyParams.Delta)
        // LinearDeltaSensitivity needs to be left unset if not provided, rather
        // than being set to some default

    .LinearDeltaSensitivity(NumericPropertyParams.GetLinearDeltaSensitivityAttribute(
    ))
        .Allowwheel(bAllowSpin)
        .wheelStep(NumericPropertyParams.WheelStep)
        .UndeterminedString(PropertyEditorConstants::DefaultUndeterminedText)
        .OnValueChanged(this,
&SPropertyEditorNumeric<NumericType>::OnValueChanged)
        .OnValueCommitted(this,
&SPropertyEditorNumeric<NumericType>::OnValueCommitted)
        .OnUndeterminedValueCommitted(this,
&SPropertyEditorNumeric<NumericType>::OnUndeterminedValueCommitted)
        .OnBeginSliderMovement(this,
&SPropertyEditorNumeric<NumericType>::OnBeginSliderMovement)
        .OnEndSliderMovement(this,
&SPropertyEditorNumeric<NumericType>::OnEndSliderMovement)
        .TypeInterface(TypeInterface)

    ];
}

virtual FReply OnMouseMove(const FGeometry& MyGeometry, const FPointerEvent&
MouseEvent) override
{
    if (MouseEvent.IsAltDown())
    {
        float DeltaToAdd =
(float)MouseEvent.GetCursorDelta().X / SliderWidthInSlateUnits;

        if (SupportDynamicSlider.MaxValue.Get() &&
(NumericType)InternalValue == GetMaxSliderValue())
        {
            ApplySlider.MaxValueChanged(DeltaToAdd, false);
        }
        else if (SupportDynamicSlider.MinValue.Get() &&
(NumericType)InternalValue == GetMinSliderValue())
        {
            ApplySlider.MinValueChanged(DeltaToAdd, false);
        }
    }
}

```

UIMax

- 功能描述:** 指定数字输入框上滚动条拖动的最大范围值
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 限制类型:** float,int32

- **关联项:** UIMin
- **常用程度:** ★★★★☆

UIMin

- **功能描述:** 指定数字输入框上滚动条拖动的最小范围值
- **使用位置:** UPROPERTY
- **引擎模块:** Numeric Property
- **元数据类型:** float/int
- **限制类型:** float,int32
- **关联项:** UIMax, ClampMin, ClampMax
- **常用程度:** ★★★★☆

UIMin-UIMax和ClampMin-ClampMax的区别是，UI系列阻止用户在拖动鼠标的时候把值超过某个范围，但是用户依然可以手动输入超过这个范围的值。而Clamp系列是实际的值的范围限制，用户拖动或者手动输入值都不允许超过这个范围。

这两个限制都无法限制蓝图下直接修改值。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest)
float MyFloat_NoMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(UIMin = "0", UIMax = "100"))
float MyFloat_HasMinMax_UI = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "0", ClampMax = "100"))
float MyFloat_HasMinMax_Clamp = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "0", ClampMax = "100", UIMin = "20", UIMax = "50"))
float MyFloat_HasMinMax_ClampAndUI = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "20", ClampMax = "50", UIMin = "0", UIMax = "100"))
float MyFloat_HasMinMax_ClampAndUI2 = 100;
```

测试效果：

- 从MyFloat_HasMinMax_UI发现，UIMin, UIMax限制数字输入框滚动条的范围，但依然可以手动输入超过的值999
- 从MyFloat_HasMinMax_Clamp发现，ClampMin, ClampMax会同时限制UI和手动输入的范围。
- 从MyFloat_HasMinMax_ClampAndUI和MyFloat_HasMinMax_ClampAndUI2发现，UI的滚动条会取UI的限制和Clamp限制的更窄范围，而实际输入值也是会被限制在更窄的范围内。

原理：

TNumericPropertyParams在构造的时候就会取得一些meta来初始化这些变量。否则就会成为默认值。

数值类型有实际的最小最大值 (MinValue-MaxValue) , 是由ClampMin和ClampMax提供的。也有UI上的最小最大值 (MinSliderValue-MaxSliderValue) , 是由Max(UImin,ClampMin)和Min(UIMax,ClampMax)提供的，即取最小的范围来保障合法输入。

```
template<typename NumericType>
struct TNumericPropertyParams
{
    if (MetaDataGetter.IsBound())
    {
        UIMinString = MetaDataGetter.Execute("UIMin");
        UIMaxString = MetaDataGetter.Execute("UIMax");
        SliderExponentString = MetaDataGetter.Execute("sliderExponent");
        LinearDeltaSensitivityString =
            MetaDataGetter.Execute("LinearDeltaSensitivity");
        DeltaString = MetaDataGetter.Execute("Delta");
        ClampMinString = MetaDataGetter.Execute("ClampMin");
        ClampMaxString = MetaDataGetter.Execute("ClampMax");
        ForcedUnits = MetaDataGetter.Execute("ForceUnits");
        wheelStepString = MetaDataGetter.Execute("wheelStep");
    }

    TOptional<NumericType> minValue;
    TOptional<NumericType> maxValue;
    TOptional<NumericType> minSliderValue;
    TOptional<NumericType> maxSliderValue;
    NumericType sliderExponent;
    NumericType Delta;
    int32 LinearDeltaSensitivity;
    TOptional<NumericType> wheelStep;
}

//最终这些值会传输给
SAssignNew(SpinBox, SSpinBox<NumericType>)
    .Style(InArgs._SpinBoxStyle)
    .Font(InArgs._Font.IsTrue() ? InArgs._Font : InArgs._EditableTextBoxStyle->TextStyle.Font)
    .value(this, &SNumericEntryBox<NumericType>::OnGetValueForSpinBox)
    .Delta(InArgs._Delta)
    .ShiftMultiplier(InArgs._ShiftMultiplier)
    .CtrlMultiplier(InArgs._CtrlMultiplier)
    .LinearDeltaSensitivity(InArgs._LinearDeltaSensitivity)
    .SupportDynamicslider.MaxValue(InArgs._SupportDynamicslider.MaxValue)
    .SupportDynamicslider.MinValue(InArgs._SupportDynamicslider.MinValue)
    .OnDynamicslider.MaxValueChanged(InArgs._OnDynamicslider.MaxValueChanged)
    .OnDynamicslider.MinValueChanged(InArgs._OnDynamicslider.MinValueChanged)
    .OnValueChanged(OnValueChanged)
    .OnValueCommitted(OnValueCommitted)
    .MinFractionalDigits(MinFractionalDigits)
    .MaxFractionalDigits(MaxFractionalDigits)
    .MinSliderValue(InArgs._MinSliderValue)
    .MaxSliderValue(InArgs._MaxSliderValue)
    .MaxValue(InArgs._MaxValue)
    .MinValue(InArgs._MinValue)
    .SliderExponent(InArgs._SliderExponent)
    .SliderExponentNeutralValue(InArgs._SliderExponentNeutralValue)
    .Enablewheel(InArgs._Allowwheel)
```

```

.BroadcastValueChangesPerKey(InArgs._BroadcastValueChangesPerKey)
.WheelStep(InArgs._WheelStep)
.OnBeginInitSliderMovement(InArgs._OnBeginInitSliderMovement)
.OnEndsSliderMovement(InArgs._OnEndsSliderMovement)
.MinDesiredWidth(InArgs._MinDesiredValueWidth)
.TypeInterface(Interface)
.ToolTipText(this, &SNumericEntryBox<NumericType>::GetValueAsText);

//最后
void SSpinBox<NumericType>::CommitValue(NumericType NewValue, double
NewSpinValue, ECommitMethod CommitMethod, ETextCommit::Type OriginalCommitInfo)
{
    if (CommitMethod == CommittedViaSpin || CommitMethod == CommittedViaArrowKey)
    {
        const NumericType LocalMinSliderValue = GetMinSliderValue();
        const NumericType LocalMaxSliderValue = GetMaxSliderValue();
        NewValue = FMath::Clamp<NumericType>(NewValue, LocalMinSliderValue,
LocalMaxSliderValue);
        NewSpinValue = FMath::Clamp<double>(NewSpinValue,
(double)LocalMinSliderValue, (double)LocalMaxSliderValue);
    }

    {
        const NumericType LocalMinValue = GetMinValue();
        const NumericType LocalMaxValue = GetMaxValue();
        NewValue = FMath::Clamp<NumericType>(NewValue, LocalMinValue,
LocalMaxValue);
        NewSpinValue = FMath::Clamp<double>(NewSpinValue, (double)LocalMinValue,
(double)LocalMaxValue);
    }

    // Update the internal value, this needs to be done before rounding.
    InternalValue = NewSpinValue;

    const bool bAlwaysUsesDeltaSnap = GetAlwaysUsesDeltaSnap();
    // If needed, round this value to the delta. Internally the value is not held
    to the Delta but externally it appears to be.
    if (CommitMethod == CommittedViaSpin || CommitMethod == CommittedViaArrowKey
|| bAlwaysUsesDeltaSnap)
    {
        NumericType CurrentDelta = Delta.Get();
        if (CurrentDelta != NumericType())
        {
            NewValue = FMath::GridSnap<NumericType>(NewValue, CurrentDelta); ///
            snap numeric point value to nearest Delta
        }
    }

    // Update the max slider value based on the current value if we're in dynamic
mode
    if (SupportDynamicsSlider.MaxValue.Get() && ValueAttribute.Get() >
GetMaxSliderValue())
    {
        ApplySlider.MaxValueChanged(Float(ValueAttribute.Get() -
GetMaxSliderValue()), true);
    }
}

```

```

        }
        else if (SupportDynamicsSliderMinValue.Get() && valueAttribute.Get() <
GetMinSliderValue())
        {
            ApplySliderMinValueChanged(float(ValueAttribute.Get() -
GetMinSliderValue()), true);
        }
    }
}

```

Units

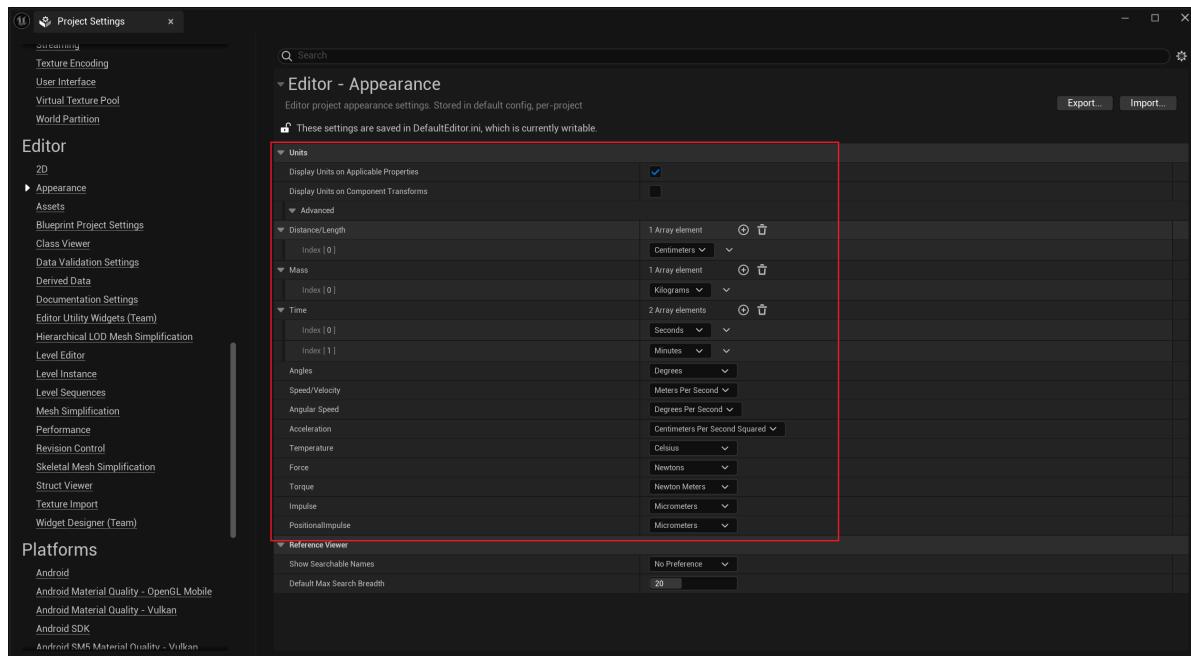
- 功能描述:** 设定属性值的单位，支持实时根据数值不同动态改变显示的单位。
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** string="abc"
- 限制类型:** float,int32
- 关联项:** ForceUnits
- 常用程度:** ★★★

设定属性值的单位。一个单位有多个别名，如Kilograms和kg, Centimeters和cm，都是可以的。

Units的作用不光是设定单位，而且隐含着这个显示的单位字符串可以根据用户输入的数值自动的进行调整适应。比如100cm其实就是1m，0.5km就是500m。

另外设定了单位之后，还可以接受用户直接在数字框中输入数字和单位的组合，比如直接敲入1km就可以设置值为1，单位为km。或者1ft为1英尺=30.84cm。

要实现自动的调整显示单位的功能，首先需要在项目设置里设置单位系列。比如下图就在距离上设置了厘米，米，千米，毫米的单位（顺序不重要）。然后之后数字框显示距离的单位时就可以在这4者之间转换。



测试代码：

```
UPROPERTY(EditAnywhere, Category = UnitsTest)
float MyFloat_NoUnits = 0.0;

UPROPERTY(EditAnywhere, Category = UnitsTest, Meta = (Units = "cm"))
float MyFloat_HasUnits_Distance = 100.f;

UPROPERTY(EditAnywhere, Category = UnitsTest, Meta = (ForceUnits = "cm"))
float MyFloat_HasForceUnits_Distance = 100.f;
```

测试效果：

- 在项目设置里填入4个单位：cm, m, km, mm后开始测试。
- 发现采用Units的属性，会自动的根据值的不同调整单位。而且也接受数字+单位的输入。
- 发现采用ForceUnits的属性，也可以接受数字+单位的输入，但是在显示上却始终显示为cm，不会调整到别的单位。



原理：

- 如果ForcedUnits有设置，则会同时设置UnderlyingUnits（基础单位），UserDisplayUnits 和 FixedDisplayUnits。
- 否则如果设置Units，则会只设置到UnderlyingUnits和FixedDisplayUnits
- 最后在显示单位的时候，如果有UserDisplayUnits，则优先采用。之后才是FixedDisplayUnits。
- ToString的时候，是把UnderlyingUnits转到UserDisplayUnits或FixedDisplayUnits。
- 在数字输入框改变的时候，会触发SetupFixedDisplay，从而在内部每次重新计算合适的单位赋值到FixedDisplayUnits。因此如上面说的，如果没有设置UserDisplayUnits（没有ForceUnits），则每次都会调整到新的合适显示单位。否则就会因为UserDisplayUnits优先级最高且一直有值，导致总是以UserDisplayUnits显示保持不变。

```
void SPropertyEditorNumeric<NumericType>::Construct( const FArguments& InArgs,
const TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    // First off, check for ForceUnits= meta data. This meta tag tells us to
    // interpret, and always display the value in these units.
    FUnitConversion::Settings().ShouldDisplayUnits does not apply to such properties
    const FString& ForcedUnits = MetaDataGetter.Execute("ForceUnits");
    TOptional<EUnit> PropertyUnits =
    FUnitConversion::UnitFromString(*ForcedUnits);
    if (PropertyUnits.IsSet())
    {
        // Create the type interface and set up the default input units if
        // they are compatible
        TypeInterface = MakeShareable(new
TNumericUnitTypeInterface<NumericType>(PropertyUnits.GetValue()));
        TypeInterface->UserDisplayUnits = TypeInterface->FixedDisplayUnits =
        PropertyUnits.GetValue();
    }
}
```

```

        // If that's not set, we fall back to Units=xxx which calculates the most
        // appropriate unit to display in
    else
    {
        if (FUnitConversion::Settings().ShouldDisplayUnits())
        {
            const FString& DynamicUnits = PropertyHandle-
>GetMetaData(TEXT("Units"));
            if (!DynamicUnits.IsEmpty())
            {
                PropertyUnits =
FUnitConversion::UnitFromString(*Dynamicunits);
            }
            else
            {
                PropertyUnits =
FUnitConversion::UnitFromString(*MetaDataGetter.Execute("Units"));
            }
        }

        if (!PropertyUnits.IsSet())
        {
            PropertyUnits = EUnit::Unspecified;
        }
    }

}

void SPropertyEditorNumeric<NumericType>::OnValueCommitted( NumericType NewValue,
ETextCommit::Type CommitInfo )
{
    if (TypeInterface.IsValid())
    {
        TypeInterface->SetupFixedDisplay(NewValue);
    }
}

template<typename NumericType>
void TNumericUnitTypeInterface<NumericType>::SetupFixedDisplay(const NumericType&
InValue)
{
    // We calculate this regardless of whether FixedDisplayUnits is used, so that
    // the moment it is used, it's correct
    EUnit DisplayUnit = FUnitConversion::CalculateDisplayUnit(InValue,
UnderlyingUnits);
    if (DisplayUnit != EUnit::Unspecified)
    {
        FixedDisplayUnits = DisplayUnit;
    }
}

//在转换的时候
 FString TNumericUnitTypeInterface<NumericType>::ToString(const NumericType&
value) const
{
    if (UserDisplayUnits.IsSet())

```

```

    {
        auto Converted = FinalValue.ConvertTo(UserDisplayUnits.GetValue());
        if (Converted.IsSet())
        {
            return ToUnitString(Converted.GetValue());
        }
    }

    if (FixedDisplayUnits.IsSet())
    {
        auto Converted = FinalValue.ConvertTo(FixedDisplayUnits.GetValue());
        if (Converted.IsSet())
        {
            return ToUnitString(Converted.GetValue());
        }
    }
}

```

WheelStep

- 功能描述:** 指定数字输入框上鼠标轮上下滚动产生的变化值
- 使用位置:** UPROPERTY
- 引擎模块:** Numeric Property
- 元数据类型:** float/int
- 常用程度:** ★★★

指定数字输入框上鼠标轮上下滚动产生的变化值。

默认值的规则:

如果属性是浮点数且UI滚动条范围小于10，则WheelStep=0.1，否则为1

如果属性是整数，则WheelStep=1

测试代码:

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest)
float MyFloat_DefaultWheelStep = 50;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest, meta =
(UIMin = "0", UIMax = "10"))
float MyFloat_SmallWheelStep = 1;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest, meta =
(wheelStep = 10))
float MyFloat_HaswheelStep = 50;

```

效果图:

默认值不指定UIMin, UIMax也可以鼠标轮滚动变化值。默认为1

MyFloat_SmallWheelStep的UI范围只有10，则默认改变幅度0.1

指定WheelStep =10，则一下子变化10

F:\UnrealSpecifiers\Doc\Meta\Numeric\WheelStep\WheelStep2.gif

原理：

通过代码可知，如果设置了WheelStep则用该值。

否则如果是浮点数且UI滚动条范围小于10，则WheelStep=0.1，否则为1

否则如果是整数，则WheelStep=1

```
virtual FReply SSpinBox<NumericType>::OnMousewheel(const FGeometry&
MyGeometry, const FPointerEvent& MouseEvent) override
{
    if (bEnableWheel && PointerDraggingSliderIndex == INDEX_NONE &&
HasKeyboardFocus())
    {
        // If there is no wheelStep defined, we use 1.0 (or 0.1 if slider
range is <= 10)
        constexpr bool bIsIntegral = TIsIntegral<NumericType>::Value;
        const bool bIsSmallStep = !bIsIntegral && (GetMaxSliderValue() -
GetMinSliderValue()) <= 10.0;
        double Step = WheelStep.IsSet() && wheelStep.Get().IsSet() ?
wheelStep.Get().GetValue() : (bIsSmallStep ? 0.1 : 1.0);

        if (MouseEvent.IsControlDown())
        {
            // If no value is set for wheelSmallStep, we use the DefaultStep
multiplied by the CtrlMultiplier
            Step *= CtrlMultiplier.Get();
        }
        else if (MouseEvent.IsShiftDown())
        {
            // If no value is set for wheelBigStep, we use the DefaultStep
multiplied by the ShiftMultiplier
            Step *= ShiftMultiplier.Get();
        }

        const double Sign = (MouseEvent.GetWheelDelta() > 0) ? 1.0 : -1.0;
        const double NewValue = InternalValue + (Sign * Step);
        const NumericType RoundedNewValue = RoundIfIntegerValue(NewValue);

        return FReply::Handled();
    }

    return FReply::Unhandled();
}
```

AssetBundles

- 功能描述：**标明该属性其引用的资产属于哪一些AssetBundles。
- 使用位置：** UPROPERTY
- 引擎模块：** Object Property

- **元数据类型:** strings="a, b, c"
- **限制类型:** UPrimaryDataAsset内部的FS softObjectPtr, FS osoftObjectPath
- **关联项:** IncludeAssetBundles
- **常用程度:** ★★★

用于UPrimaryDataAsset内部的 SoftObjectPtr 或 SoftObjectPath 属性，标明其引用的资产属于哪一些 AssetBundles。

要理解这个的作用，需要先理解一些基本概念：

- PrimaryAsset指的是在游戏中可以进行手动载入/释放的东西。包括关卡文件 (.umap) 以及一些游戏相关的物件，例如角色或者背包里的物品。顾名思义，主要资产指的是游戏里的主要根部资产，其引用树下有一大堆其他资产。另一方面，我们往往会有主动加载或释放这些主要资产，比如加载关卡，加载怪物角色，加载掉落道具。但我们一般不太会直接去加载材质贴图声音这种资产，因为它们绝大多数是被主要资产引用着。我们在加载主要资产的时候，就会自带的加载这些次要资产了。
- SecondaryAsset指的是其他的那些Assets了，例如贴图和声音等。这一类型的assets是根据 PrimaryAsset来自动进行载入的。我们一般来说不太需要对次要资产进行管理，其会被主要资产根据引用关系来自动的加载。
- AssetBundle可以叫做资产包，其实就是一个Asset的列表，我们对每个资产包起个名字来区分，比如UI, Game，这样其实也是对一些资产进行标签分类。这里的Asset我们不区分是PrimaryAsset还是SecondaryAsset，因为这是从用途上进行区分的，而不是加载方式。AssetBundle的作用是当我们加载PrimaryAsset的时候，这个PrimaryAsset本身可能引用着另外一些SecondaryAsset资产，各自有不同的用途。我们就可以把这些SecondaryAsset资产划分到不同的AssetBundle里，这样我们在加载PrimaryAsset的时候，可以通过额外提供AssetBundleName来更加精细化的控制 SecondaryAsset资产的加载。
- PrimaryAsset里的指定AssetBundle的Asset属性必须是软引用，否则是硬引用的话无论如何也会被加载进来。软引用的Asset在默认时候需要我们手动的进行加载，通过附加AssetBundle，就可以在加载PrimaryAsset的时候，附带的加载该软引用资产。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_Item :public UPrimaryDataAsset
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
        FString Name;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta=(AssetBundles="UI,Game"))
        TSoftObjectPtr<UTexture2D> Icon;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta=(AssetBundles="Game"))
        TSoftObjectPtr<UStaticMesh> Mesh;

public:
    virtual FPrimaryAssetId GetPrimaryAssetId() const override;
};
```

测试效果：

- 首先我们在BP里定义了UMyProperty_Asset_Item 的资产，并相应的配置上了引用的对象。如图所示，有一个图标是给UI和Game用的，有一个Mesh是专门给Game用的。大概设想一下在一些界面我们只需要道具的图标就可以了。

- 然后在LoadPrimaryAsset的时候可以指定LoadBundles的名字，从而只加载特定的Bundle。如下图所示。
- 当指定Bundle为UI的时候，可以看见Mesh并没有加载进来。
- 当指定Bundle为Game的时候，可以看见Icon和Mesh都加载了进来。
- 要注意在编辑器下测试时候，如果之前已经加载了Mesh，因为还常驻在编辑器内存里。因此即使是使用名字UI，也仍然会发现Mesh可以被引用到。



原理：

首先UPrimaryDataAsset里有一个AssetBundleData保存着当前引用的AssetBundle的信息，这个信息是在编辑器环境下PreSave的时候保存的，会在UAssetManager::InitializeAssetBundlesFromMetadata里进行meta的分析和映射。之后在UAssetManager的LoadPrimaryAsset时内部调用ChangeBundleStateForPrimaryAssets，然后检查AssetBundle把其他额外要一并加载的Asset添加到PathsToLoad，从而最终完成一并加载的这个逻辑。

```

void UAssetManager::InitializeAssetBundlesFromMetadata_Recursive(const UStruct* Struct, const void* StructValue, FAssetBundleData& AssetBundle, FName DebugName, TSet<const void*>& AllVisitedStructValues) const
{
    static FName AssetBundlesName = TEXT("AssetBundles");
    static FName IncludeAssetBundlesName = TEXT("IncludeAssetBundles");

    //根据当前对象的值，搜索拥有AssetBundles的属性的值，最后AddBundleAsset，BundleName就是设置的值，而FoundRef是引用的对象的资产路径
    TSet<FName> BundleSet;
    TArray<const FProperty*> PropertyChain;
    It.GetPropertyChain(PropertyChain);

    for (const FProperty* PropertyToSearch : PropertyChain)
    {
        if (PropertyToSearch->HasMetaData(AssetBundlesName))
        {
            TSet<FName> LocalBundleSet;
            TArray< FString > BundleList;
            const FString& BundleString = PropertyToSearch-
>GetMetaData(AssetBundlesName);
            BundleString.ParseIntoArrayWS(BundleList, TEXT(", "));

            for (const FString& BundleNameString : BundleList)
            {
                LocalBundleSet.Add(FName(*BundleNameString));
            }

            // If Set is empty, initialize. otherwise intersect
            if (BundleSet.Num() == 0)
            {
                BundleSet = LocalBundleSet;
            }
            else
            {
                BundleSet = BundleSet.Intersect(LocalBundleSet);
            }
        }
    }
}

```

```

        }

    }

    for (const FName& BundleName : BundleSet)
    {
        AssetBundle.AddBundleAsset(BundleName,
        FoundRef.GetAssetPath());
    }
}

#if WITH_EDITORONLY_DATA
void UPrimaryDataAsset::UpdateAssetBundleData()
{
    // By default parse the metadata
    if (UAssetManager::IsInitialized())
    {
        AssetBundleData.Reset();
        UAssetManager::Get().InitializeAssetBundlesFromMetadata(this,
        AssetBundleData);
    }
}

void UPrimaryDataAsset::PreSave(FObjectPreSaveContext ObjectSaveContext)
{
    Super::PreSave(ObjectSaveContext);

    UpdateAssetBundleData();

    if (UAssetManager::IsInitialized())
    {
        // Bundles may have changed, refresh
        UAssetManager::Get().RefreshAssetData(this);
    }
}
#endif

void UPrimaryDataAsset::PostLoad()
{
    Super::PostLoad();

#if WITH_EDITORONLY_DATA
    FAssetBundleData OldData = AssetBundleData;

    UpdateAssetBundleData();

    if (UAssetManager::IsInitialized() && OldData != AssetBundleData)
    {
        // Bundles changed, refresh
        UAssetManager::Get().RefreshAssetData(this);
    }
#endif
}

//加载asset的时候，如果有FAssetBundleEntry，则一起加到PathsToLoad里

```

```

TSharedPtr<FStreamableHandle>
UAssetManager::ChangeBundleStateForPrimaryAssets(const TArray<FPrimaryAssetId>&
AssetsToChange, const TArray< FName >& AddBundles, const TArray< FName >&
RemoveBundles, bool bRemoveAllBundles, FStreamableDelegate DelegateToCall,
TAsyncLoadPriority Priority)
{
    if (!AssetPath.IsNull())
    {
        // Dynamic types can have no base asset path
        PathsToLoad.Add(AssetPath);
    }

    for (const FName& BundleName : NewBundleState)
    {
        FAssetBundleEntry Entry = GetAssetBundleEntry(PrimaryAssetId,
        BundleName);

        if (Entry.IsValid())
        {
            for (const FTopLevelAssetPath & Path : Entry.AssetPaths)
            {
                PathsToLoad.AddUnique(FSoftObjectPath(Path));
            }
        }
        else
        {
            UE_LOG(LogAssetManager, Verbose,
TEXT("ChangeBundleStateForPrimaryAssets: No assets for bundle %s::%s"),
*PrimaryAssetId.ToString(), *BundleName.ToString());
        }
    }
}

```

参考文档: https://dev.epicgames.com/documentation/en-us/unreal-engine/asset-management-in-unreal-engine?application_version=5.4

CollapsibleChildProperties

- 功能描述:** 在TextureGraph模块中新增加的meta。用于折叠一个结构的内部属性。
- 使用位置:** UPROPERTY
- 元数据类型:** bool
- 限制类型:** TextureGraph插件内使用
- 关联项:** ShowInnerProperties
- 常用程度:** 0

在TextureGraph模块中新增加的meta。用于折叠一个结构的内部属性。

源码:

```

bool STG_GraphPinOutputSettings::collapsibleChildProperties() const
{
    FProperty* Property = GetPinProperty();
}

```

```

        bool Collapsible = false;
        // check if there is a display name defined for the property, we use that as
        // the Pin Name
        if (Property && Property->HasMetaData("CollapsibleChildProperties"))
        {
            Collapsible = true;
        }
        return Collapsible;
    }

    UPROPERTY(EditAnywhere, Category = NoCategory, meta = (TGType = "TG_Input",
    CollapsibleChildProperties, ShowOnlyInnerProperties, FullyExpand,
    NoResetToDefault, PinDisplayName = "Settings") )
    FTG_OutputSettings OutputSettings;

```

DisplayThumbnail

- 功能描述:** 指定是否在该属性左侧显示一个缩略图。
- 使用位置:** UPROPERTY
- 引擎模块:** Object Property
- 元数据类型:** bool
- 限制类型:** UObject*
- 关联项:** ThumbnailSize
- 常用程度:** ★★★

指定是否在该属性左侧显示一个缩略图。

测试代码:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Thumbnail_Test :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"false"))
    UObject* MyObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"true"))
    UObject* MyObject_DisplayThumbnail;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    AActor* MyActor;

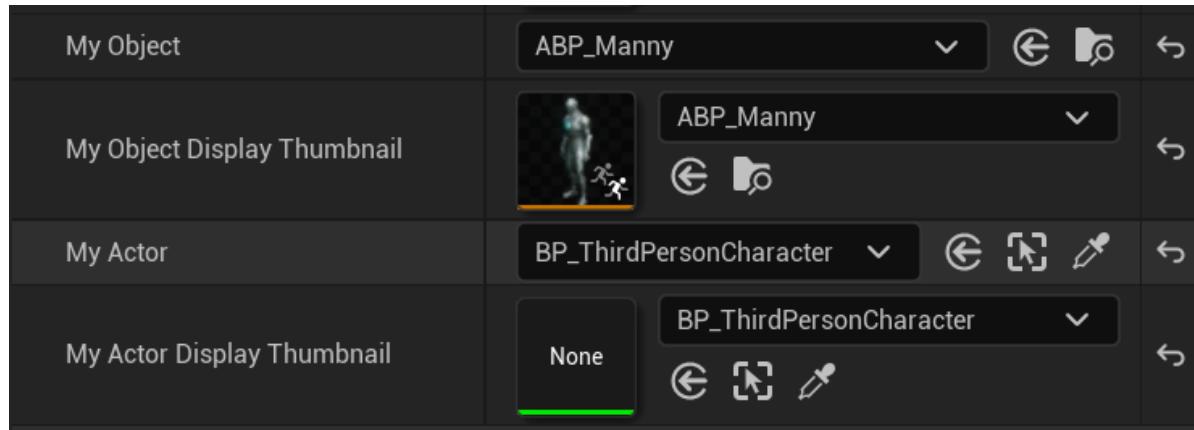
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"true"))
    AActor* MyActor_DisplayThumbnail;
};

```

测试效果：

可见MyObject_DisplayThumbnail的左侧显示出了所选择资产的缩略图，而MyObject因为设置了false因此是没有的。如果不设置DisplayThumbnail =false，则默认也是会显示缩略图的。

MyActor_DisplayThumbnail出现了缩略图的图标，但是发现并没有显示出正确的说了图。AActor在默认情况下是不显示缩略图的。



原理：

判断是否要显示缩略图就在这个函数。

默认情况下非Actor类型才会显示。另外SPropertyEditorAsset是用在资产类型属性上的，其实质是Object属性。

```
bool SPropertyEditorAsset::ShouldDisplayThumbnail(const FArguments& InArgs, const UClass* InObjectClass) const
{
    if (!InArgs._DisplayThumbnail || !InArgs._ThumbnailPool.IsValid())
    {
        return false;
    }

    bool bShowThumbnail = InObjectClass == nullptr || !InObjectClass->IsChildOf(AActor::StaticClass());

    // also check metadata for thumbnail & text display
    const FPropertyParams*PropertyParamsToCheck = nullptr;
    if (PropertyParamsEditor.IsValid())
    {
       PropertyParamsToCheck =PropertyParamsEditor->GetProperty();
    }
    else if (PropertyParamsHandle.IsValid())
    {
       PropertyParamsToCheck =PropertyParamsHandle->GetProperty();
    }

    if (PropertyParamsToCheck != nullptr)
    {
       PropertyParamsToCheck = GetActualMetadataPropertyParamsPropertyParamsToCheck);
    }

    return GetTagOrBoolMetadataPropertyParamsToCheck, TEXT("DisplayThumbnail"),
bShowThumbnail;
```

```

    }

    return bShowThumbnail;
}

```

ExposeFunctionCategories

- 功能描述:** 指定该Object属性所属于的类里的某些目录下的函数可以直接在本类上暴露出来。
- 使用位置:** UPROPERTY
- 引擎模块:** Object Property
- 元数据类型:** strings="a, b, c"
- 限制类型:** UObject*
- 常用程度:** ★★★

指定该Object属性所属于的类里的某些目录下的函数可以直接在本类上暴露出来。

一开始直接还挺难理解的其含义和作用的，但这其实是一个便利性的功能而已。比如有类A里面定义一些函数，然后类B里有个A的对象。这个时候如果想在B对象身上去调用A的函数，就得手动先拖拉出B.ObjA然后再拖拉出其内部的函数。我们希望把当前B的应用上下文场景下，可以把A里的某些函数直接比较方便的暴露到B里来调用。

其实就是引擎帮我们自动的拖拉出B.ObjA这一步操作而已。你如果想要调用A里的更多的函数，也可以自己手动在B.ObjA身上拖拉右键出更多的函数。

源码里这种应用也比较多，比较方便的例子是以下源码例子，这样当前在ASkeletalMeshActor 身上就可以直接拖拉出USkeletalMeshComponent里ExposeFunctionCategories 所定义的目录的函数。

```

UCLASS(ClassGroup=ISkeletalMeshes, Blueprintable, ComponentWrapperClass,
ConversionRoot, meta=(ChildCanTick), MinimalAPI)
class ASkeletalMeshActor : public AActor
{
private:
    UPROPERTY(Category = SkeletalMeshActor, VisibleAnywhere, BlueprintReadOnly,
meta = (ExposeFunctionCategories =
"Mesh,Components|SkeletalMesh,Animation,Physics", AllowPrivateAccess = "true"))
    TobjectPtr<class USkeletalMeshComponent> SkeletalMeshComponent;
}

```

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeFunctionCategories :public UObject
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, Category = "FirstFunc")
    void MyExposeFunc1() {}

    UFUNCTION(BlueprintCallable, Category = "SecondFunc")
    void MyExposeFunc2() {}

    UFUNCTION(BlueprintCallable, Category = "ThirdFunc")
    void MyExposeFunc3() {}

};

```

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeFunctionCategories_Test :public UObject
{
    GENERATED_BODY()

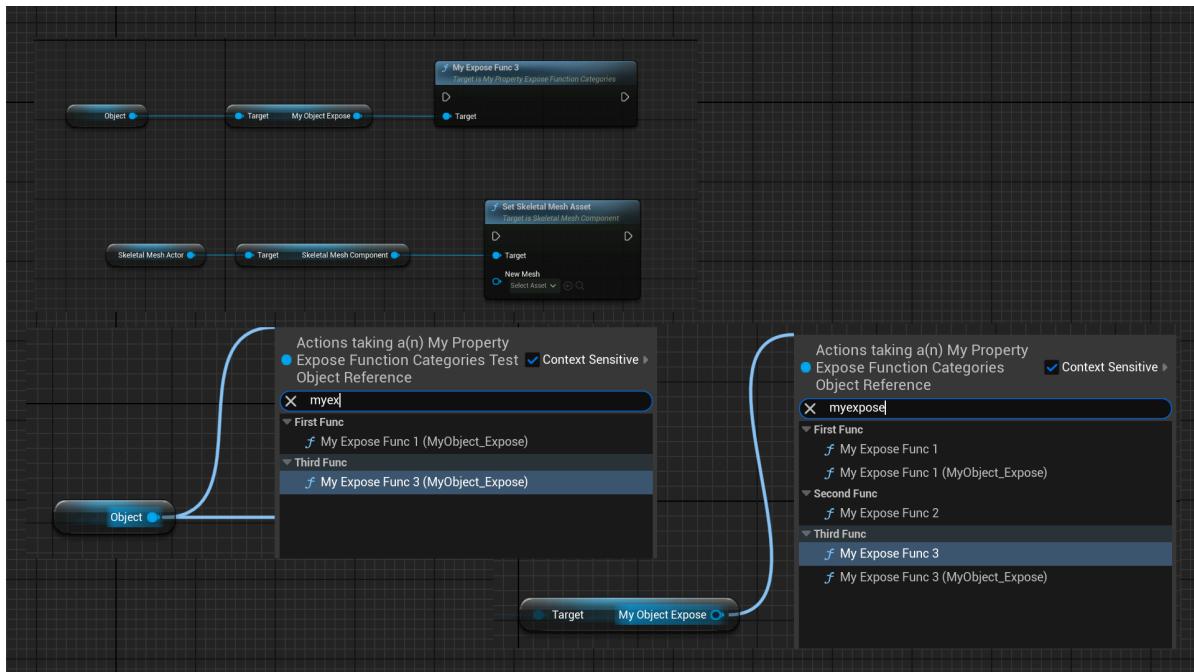
public:
    UPROPERTY(BlueprintReadOnly, meta = (ExposeFunctionCategories =
"FirstFunc,ThirdFunc"))
    UMyProperty_ExposeFunctionCategories* MyObject_Expose;
};

```

测试效果：

可以见到在UMyProperty_ExposeFunctionCategories_Test类型的Object身上，我直接输入MyExposeFunc就可以弹出“FirstFunc, ThirdFunc”这两个目录里的这两个函数，而不会直接弹出MyExposeFunc2，因为其没有直接被暴露出来。

而如果在MyObject_Expose这种内部对象上直接拖拉右键，则可以见到所有内部定义的函数。注意这里虽然有两个条目的MyExposeFunc1，但其实调用出来的是同一个函数，实际并没有影响。



原理：

在蓝图右键菜单的构建过程中，会判断某个操作是否要过滤掉。这里的IsUnexposedMemberAction就是判断这个函数是否应该被过滤掉。大致的逻辑是取得其函数对应的属性，比如在UMyProperty_ExposeFunctionCategories_Test这个Object身上，递归到子对象，其实有3个函数会进来参加测试。这3个函数（MyExposeFunc 1 2 3）各自有自己的Category，但都对应MyObject_Expose这个Property，因此其AllExposedCategories的值是我们定义的“FirstFunc,ThirdFunc”数组，最终只有两个函数通过测试，因此最后显示1, 3两个函数。

```

static bool
BlueprintActionMenuUtilsImpl::IsUnexposedMemberAction(FBlueprintActionFilter
const& Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool bIsFilteredout = false;
}

```

```

if (UFunction const* Function = BlueprintAction.GetAssociatedFunction())
{
    TArray< FString> AllExposedCategories;
    for (FBindingObject Binding : BlueprintAction.GetBindings())
    {
        if (FProperty* Property = Binding.Get< FProperty>())
        {
            const FString& ExposedCategoryMetadata = Property-
>GetMetaData(FBlueprintMetadata::MD_ExposeFunctionCategories);
            if (ExposedCategoryMetadata.IsEmpty())
            {
                continue;
            }

            TArray< FString> PropertyExposedCategories;
            ExposedCategoryMetadata.ParseIntoArray(PropertyExposedCategories,
TEXT(","),
true);
            AllExposedCategories.Append(PropertyExposedCategories);
        }
    }
}

const FString& FunctionCategory = Function-
>GetMetaData(FBlueprintMetadata::MD_FunctionCategory);
bIsFilteredout = !AllExposedCategories.Contains(FunctionCategory);
}
return bIsFilteredout;
}

```

FullyExpand

- **使用位置:** UPROPERTY
- **元数据类型:** bool
- **关联项:** ShowInnerProperties

但是没有发现该Meta被使用的原理代码。

在源码中搜索发现有多处应用，但实际上没有原理代码。

```

/** The options that are available on the node. */
UPROPERTY(EditAnywhere, Instanced, Category = "Options", meta=
(ShowInnerProperties, FullyExpand="true"))
TObjectPtr<UMovieGraphValueContainer> SelectOptions;

/** The currently selected option. */
UPROPERTY(EditAnywhere, Instanced, Category = "Options", meta=
(ShowInnerProperties, FullyExpand="true"))
TObjectPtr<UMovieGraphValueContainer> SelectedOption;

```

HideAssetPicker

- **功能描述:** 隐藏Object类型引脚上的AssetPicker的选择列表
- **使用位置:** UFUNCTION

- **引擎模块:** Object Property
- **元数据类型:** strings="a, b, c"
- **限制类型:** UObject*
- **常用程度:** ★★

隐藏Object类型引脚上的AssetPicker的选择列表。这在有时我们只是想要自己传递Object引用，不希望用户选择到引擎里别的资产的时候会比较有用。因为Asset类型其实也是Object，因此对于Object引用类型的参数叫做HideAssetPicker。

在源码里并没有找到有使用的地方，但是这个功能是可用的。

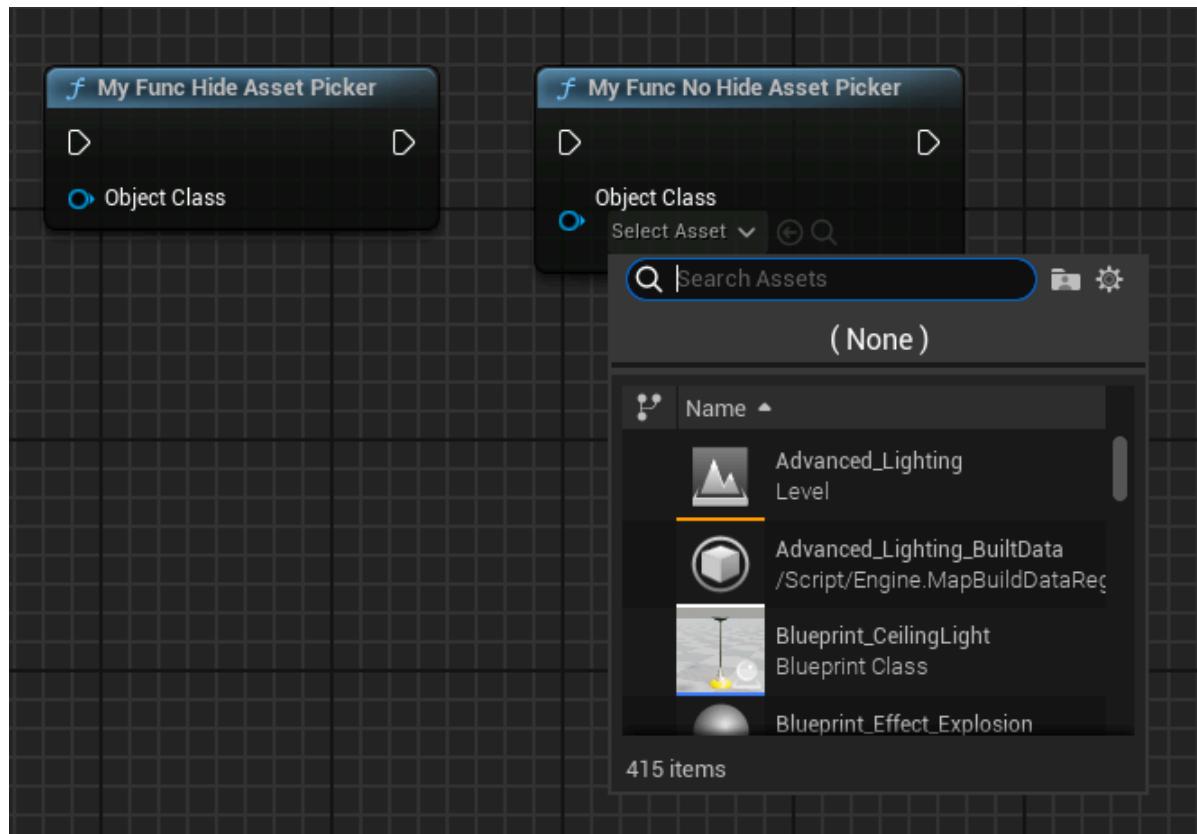
测试代码：

```
UFUNCTION(BlueprintCallable)
static void MyFunc_NoHideAssetPicker(UObject* ObjectClass) {}

UFUNCTION(BlueprintCallable, meta = (HideAssetPicker = "ObjectClass"))
static void MyFunc_HideAssetPicker(UObject* ObjectClass) {}
```

蓝图效果：

默认的情况MyFunc_NoHideAssetPicker是可以弹出选择列表的。而MyFunc_HideAssetPicker则就隐藏了起来。



原理：

判断一个函数引脚是否允许打开AssetPicker的逻辑是：

- 必须是个object 类型
- 如果是UActorComponent则不显示

- 如果是Actor类型，那么得在关卡蓝图中，且该Actor是placeable的才显示。
- 如果用HideAssetPicker显式指定了该参数，则该参数也不显示。

```

bool UEdGraphSchema_K2::ShouldShowAssetPickerForPin(UEdGraphPin* Pin) const
{
    bool bShow = true;
    if (Pin->PinType.PinCategory == PC_Object)
    {
        UClass* ObjectClass = Cast<UClass>(Pin-
>PinType.PinSubCategoryObject.Get());
        if (ObjectClass)
        {
            // Don't show literal buttons for component type objects
            bShow = !ObjectClass->IsChildOf(UActorComponent::StaticClass());

            if (bShow && ObjectClass->IsChildOf(AActor::StaticClass()))
            {
                // Only show the picker for Actor classes if the class is
                placeable and we are in the level script
                bShow = !ObjectClass->HasAllClassFlags(CLASS_NotPlaceable)
                    &&
FBlueprintEditorUtils::IsLevelScriptBlueprint(FBlueprintEditorUtils::FindBlueprint
tForNode(Pin->GetOwningNode()));
            }
        }
    }

    if (bShow)
    {
        if (UK2Node_CallFunction* CallFunctionNode =
Cast<UK2Node_CallFunction>(Pin->GetOwningNode()))
        {
            if (UFunction* FunctionRef = CallFunctionNode-
>GetTargetFunction())
            {
                const UEdGraphPin* WorldContextPin = CallFunctionNode-
>FindPin(FunctionRef->GetMetaData(FBlueprintMetadata::MD_WorldContext));
                bShow = (WorldContextPin != Pin);

                // Check if we have explicitly marked this pin as hiding
                the asset picker
                const FString& HideAssetPickerMetaData = FunctionRef-
>GetMetaData(FBlueprintMetadata::MD_HideAssetPicker);
                if (!HideAssetPickerMetaData.IsEmpty())
                {
                    TArray< FString> PinNames;
                    HideAssetPickerMetaData.ParseIntoArray(PinNames,
TEXT("",), true);
                    const FString PinName = Pin->GetName();
                    for (FString& ParamNameToHide : PinNames)
                    {
                        ParamNameToHide.TrimStartAndEndInline();
                        if (ParamNameToHide == PinName)
                        {
                            bShow = false;
                            break;
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
else if (Cast<UK2Node_CreateDelegate>( Pin->GetOwningNode()))
{
    bShow = false;
}
}
}

return bShow;
}

```

IncludeAssetBundles

- 功能描述:** 用于UPrimaryDataAsset的子对象属性，指定应该继续递归到孩子对象里去探测AssetBundle数据。
- 使用位置:** UPROPERTY
- 引擎模块:** Object Property
- 元数据类型:** string="abc"
- 限制类型:** UPrimaryDataAsset内部的ObjectPtr属性
- 关联项:** AssetBundles
- 常用程度:** ★★

用于UPrimaryDataAsset的子对象属性，指定应该继续递归到孩子对象里去探测AssetBundle数据。

这样这些指对象内部的 FSoftObjectPtr 或 FSoftObjectPath 属性，其上面标明的AssetBundle的数据才会被解析添加到UPrimaryDataAsset的AssetBundleData里。

- 默认情况下，InitializeAssetBundlesFromMetadata_Recursive只会分析到UPrimaryDataAsset的本身这一层级的属性，比如下面的Icon和Mesh属性。
- 而如果再嵌套了一层，就是UPrimaryDataAsset下面拥有只对象，UMyProperty_Asset_ChildObject，而UMyProperty_Asset_ChildObject 里面又包含 FSoftObjectPath，希望它被属于AssetBundles 的一部分，在加载UPrimaryDataAsset的时候同时一并加载。这个时候就需要告诉引擎需要继续分析这个子对象。
- 注意到UMyProperty_Asset_ChildObject我都是用TObjectPtr，是个硬引用，该对象在UMyProperty_Asset_Item 被加载的时候也会自然被加载进来。因此无论如何，UMyProperty_Asset_ChildObject 都会被加载进来。但是UMyProperty_Asset_ChildObject 内部的 ChildIcon是用TSoftObjectPtr，是软引用，因此必须依赖AssetBundle的机制才会被加载。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_ChildObject :public UDataAsset
{
GENERATED_BODY()
public:
UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles = "Client"))
TSoftObjectPtr<UTexture2D> ChildIcon;

```

```

};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_Item :public UPrimaryDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString Name;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles =
"UI,Game"))
    TSoftObjectPtr<UTexture2D> Icon;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles = "Game"))
    TSoftObjectPtr<UStaticMesh> Mesh;

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyProperty_Asset_ChildObject>
    MyChildObject_NotIncludeAssetBundles;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (IncludeAssetBundles))
    TObjectPtr<UMyProperty_Asset_ChildObject> MyChildObject_IncludeAssetBundles;
public:
    virtual FPrimaryAssetId GetPrimaryAssetId() const override;
};

```

测试效果：

配置的数据图的下部分，分别配置了两张图片。但在LoadPrimaryAsset后，只有MyChildObject_IncludeAssetBundles内部的ChildIcon才被加载进来。



如果分析UMyProperty_Asset_Item 的AssetBunbleData数据，会发现其Client只包含第二张Stone图片的路径。这是因为只有第二张图片才被分析到并包含进来。

```

{
    BundleName = "Client";
    BundleAssets =
    {
        {
            AssetPath =
            {
                PackageName = "/Game/Asset/Image/T_Shop_Stone";
                AssetName = "T_Shop_Stone";
            };
            SubPathString = "";
        };
    },
    AssetPaths =
    {
        {
            PackageName = "/Game/Asset/Image/T_Shop_Stone";
            AssetName = "T_Shop_Stone";
        };
    };
}
```

```
    },  
};
```

原理：

UPrimaryDataAsset下的属性如果是个Object属性，只当有IncludeAssetBundles的时候，才会进一步递归向下探测。

```
void UAssetManager::InitializeAssetBundlesFromMetadata_Recursive(const UStruct*  
Struct, const void* StructValue, FAssetBundleData& AssetBundle, FName DebugName,  
TSet<const void*>& AllVisitedStructValues) const  
{  
    static FName AssetBundlesName = TEXT("AssetBundles");  
    static FName IncludeAssetBundlesName = TEXT("IncludeAssetBundles");  
  
    //根据当前对象的值，搜索拥有AssetBundles的属性的值，最后AddBundleAsset，BundleName就是设置的值，而FoundRef是引用的对象的资产路径  
    else if (const FObjectProperty* ObjectProperty = CastField<FOBJECTPROPERTY>  
(Property))  
    {  
        if (ObjectProperty->PropertyFlags & CPF_InstancedReference ||  
ObjectProperty->GetOwnerProperty()->HasMetaData(IncludeAssetBundlesName))  
        {  
            const UObject* Object = ObjectProperty->  
GetObjectPropertyValue(PropertyValue);  
            if (Object != nullptr)  
            {  
                InitializeAssetBundlesFromMetadata_Recursive(Object->GetClass(),  
Object, AssetBundle, Object->GetFName(), AllVisitedStructValues);  
            }  
        }  
    }  
}
```

LoadBehavior

- 功能描述：**用在UCLASS上标记这个类的加载行为，使得相应的TObjectPtr属性支持延迟加载。可选的加载行为默认为Eager，可改为LazyOnDemand。
- 使用位置：** UCLASS
- 引擎模块：** Object Property
- 元数据类型：** string="abc"
- 限制类型：** TObjectPtr
- 常用程度：** ★

用在UCLASS上标记这个类的加载行为，使得相应的TObjectPtr属性支持延迟加载。可选的加载行为默认为Eager，可改为LazyOnDemand。

- 默认Eager的逻辑和我们常见的资源硬引用的逻辑相同，就是如果A硬引用了B，在加载A的时候就会递归把B也加载进来。

- 改为LazyOnDemand的逻辑是只有在这个资源真正被需要（触发Get）的时候才会去加载该资源。这也是种硬引用，但是是延迟加载的。同样A硬引用了B，在加载A的时候，不会直接立马就加载B，而是先记录下来这个引用关系信息（B的ObjectPath）。在A里真正需要访问B的时候，这个时候因为已经事先记录知道了B在哪里，因此就可以在这个时候再去把B加载进来。如果加载的够快，对用户是无感的。LazyOnDemand只在编辑器下生效，这么做的好处是可以尽快的打开编辑器，而不是等待所有资源都加载进来。因为其实并不是所有资源都要第一时间加载解析进来访问。
- 同FS soft ObjectPtr的区别是后者是软引用，需要用户手动的自己判断时机加载。而LazyOnDemand是自动的延迟加载，用户是无感的，不需要做额外的操作。
- LoadBehavior只作用于TObjectPtr属性，UObject属性总是直接加载的。因为只有TObjectPtr里实现了UObject的引用路径信息编码，才可以支持延迟加载。
- LoadBehavior也只支持在编辑器环境。因为在Runtime，TObjectPtr会退化成UObject*，也就全部都是直接加载了。
- LoadBehavior一般标记在资产类型的类上。源码里标记的类有：
DataAsset, DataTable, CurveTable, SoundCue, SoundWave, DialogueWave, AnimMontage。因此假如你自己自定义了资产类，也包含了许多数据，就可以用LazyOnDemand来优化在编辑器下的加载速度。
- 要测试LoadBehavior，要打开引擎的LazyLoadImports功能，默认情况下是关闭的。打开的方式可在DefaultEngine.ini下增加Core.System.Experimental节下LazyLoadImports=True的设置。源码可参考IsImportLazyLoadEnabled这个方法。
- 在测试的时候，要小心如果是双击打开DataAsset资产，因为在属性细节面板里要展示属性的值，在属性上会调用GetObjectPropertyValue_InContainer，因此会触发ObjectHandleResolve，导致TObjectPtr的Resolve。

测试代码：

如下专门定义了UMyDataAsset_Eager 和UMyDataAsset_LazyOnDemand 两种DataAsset，并标注了不同的LoadBehavior 以做对比。

```
//(BlueprintType = true, IncludePath = Property/MyProperty_Asset.h,
IsBlueprintBase = true, LoadBehavior = Eager, ModuleRelativePath =
Property/MyProperty_Asset.h)
UCLASS(Blueprintable, Blueprintable, meta = (LoadBehavior = "Eager"))
class INSIDER_API UMyDataAsset_Eager :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

//(BlueprintType = true, IncludePath = Property/MyProperty_Asset.h,
IsBlueprintBase = true, LoadBehavior = LazyOnDemand, ModuleRelativePath =
Property/MyProperty_Asset.h)
UCLASS(Blueprintable, Blueprintable, meta = (LoadBehavior = "LazyOnDemand"))
class INSIDER_API UMyDataAsset_LazyOnDemand :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};
```

```

UCLASS(BlueprintType)
class INSIDER_API UMyClass_LoadBehaviorTest :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyDataAsset_LazyOnDemand> MyLazyOnDemand_AssetPtr;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyDataAsset_Eager> MyEager_AssetPtr;

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (LoadBehavior = "Eager"))
    TObjectPtr<UMyDataAsset_LazyOnDemand>
    MyLazyOnDemand_AssetPtr_EagerOnProperty;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (LoadBehavior =
"LazyOnDemand"))
    TObjectPtr<UMyDataAsset_Eager> MyEager_AssetProperty_LazyOnDemandOnProperty;

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UMyDataAsset_LazyOnDemand* MyLazyOnDemand_Asset;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UMyDataAsset_Eager* MyEager_Asset;

public:
    UFUNCTION(BlueprintCallable)
    static void LoadBehaviorTest();
};

void UMyClass_LoadBehaviorTest::LoadBehaviorTest()
{
    UPackage* pk = LoadPackage(nullptr,
    TEXT("/Game/Class/Behavior/LoadBehavior/DA_LoadBehaviorTest"), 0);
    UMyClass_LoadBehaviorTest* obj = LoadObject<UMyClass_LoadBehaviorTest>(pk,
    TEXT("DA_LoadBehaviorTest"));
}

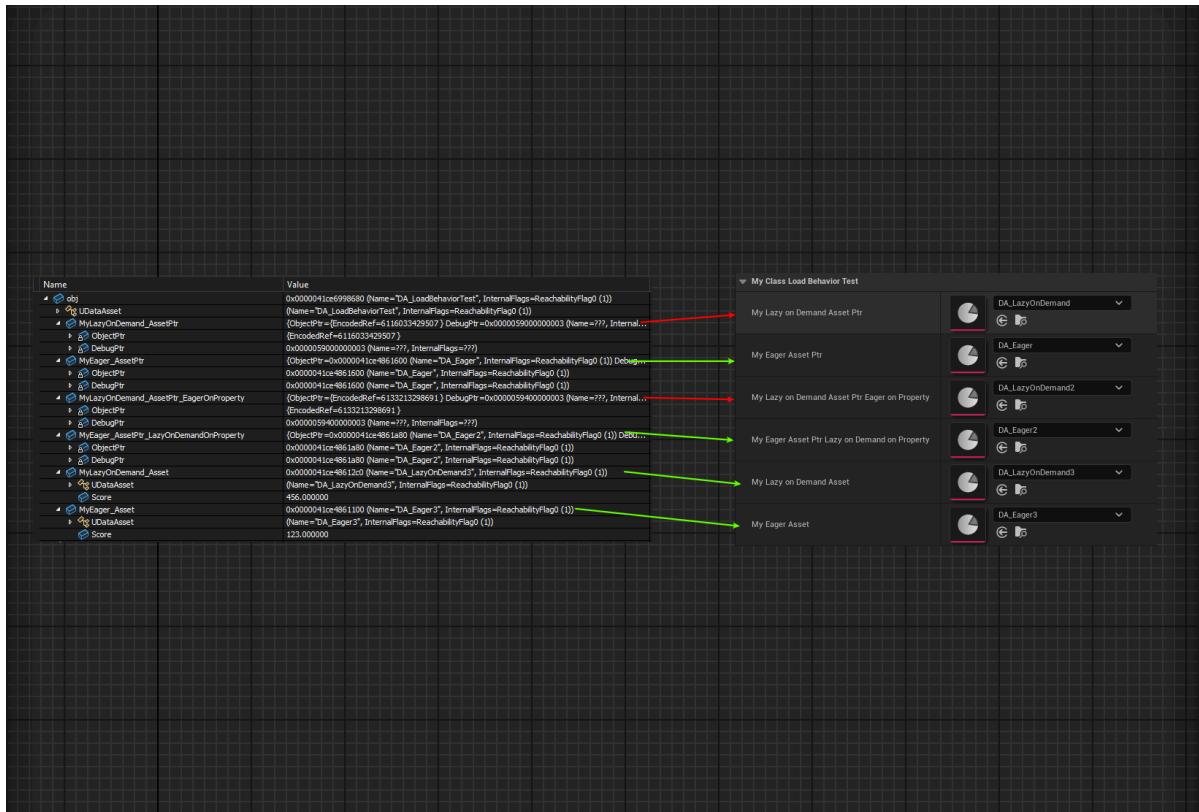
//开启功能
DefaultEngine.ini
[Core.System.Experimental]
LazyLoadImports=True

```

测试结果：

在编辑器运行起来之后，手动调用LoadBehaviorTest来加载这个UMyClass_LoadBehaviorTest 的 DataAsset。查看不同类型属性的对象值。可以发现：

- 其中MyLazyOnDemand_AssetPtr和MyLazyOnDemand_AssetProperty_LazyOnDemandOnProperty的ObjectPtr 的值是还没有Resolved的，其他的都可以查看到直接对象的值。
- 可以得出的结论有，只有在UCLASS上标记LazyOnDemand才可以使得延迟加载生效。在属性上标记LoadBehavior 并不会起作用。直接UObject*的属性统统都会直接加载。



原理：

在LinkerLoadImportBehavior.cpp里可看见判断LoadBehavior的FindLoadBehavior方法，因此发现其只作用在UCLASS上。

另外也可在TObjectPtr的Get函数里发现ResolveObjectHandle的调用。这是触发Resolve的地方。

也注意到UE_WITH_OBJECT_HANDLE_LATE_RESOLVE 的定义是WITH_EDITORONLY_DATA，因此是在编辑器环境下生效。

```

//D:\github\UnrealEngine\Engine\Source\Runtime\CoreUObject\Private\Uobject\Linker
LoadImportBehavior.cpp

enum class EImportBehavior : uint8
{
    Eager = 0,
    // @TODO: OBJPTR: we want to permit lazy background loading in the future
    //LazyBackground,
    LazyOnDemand,
};

EImportBehavior FindLoadBehavior(const UClass& Class)
{
    //Package class can't have meta data because of UHT
    if (&Class == UPackage::StaticClass())
    {
        return EImportBehavior::LazyOnDemand;
    }

    static const FName Name_LoadBehavior(TEXT("LoadBehavior"));
    if (const FString* LoadBehaviorMeta = Class.FindMetaData(Name_LoadBehavior))
    {
        if (*LoadBehaviorMeta == TEXT("LazyOnDemand"))
        {
            return EImportBehavior::LazyOnDemand;
        }
    }
}

```

```

        return EImportBehavior::LazyOnDemand;
    }
    return EImportBehavior::Eager;
}
else
{
    //look in super class to see if it has lazy load on
    const UClass* Super = Class.GetSuperClass();
    if (Super != nullptr)
    {
        return FindLoadBehavior(*Super);
    }
    return EImportBehavior::Eager;
}
}

#define UE_WITH_OBJECT_HANDLE_LATE_RESOLVE WITH_EDITORONLY_DATA

inline UObject* ResolveObjectHandle(FObjectHandle& Handle)
{
#if UE_WITH_OBJECT_HANDLE_LATE_RESOLVE || UE_WITH_OBJECT_HANDLE_TRACKING
    UObject* ResolvedObject = ResolveObjectHandleNoRead(Handle);
    UE::CoreUObject::Private::OnHandleRead(ResolvedObject);
    return ResolvedObject;
#else
    return ReadObjectHandlePointerNoCheck(Handle);
#endif
}

```

MustBeLevelActor

- **使用位置:** UPROPERTY
- **引擎模块:** Object Property
- **元数据类型:** bool

意思是这个必须是场景里的Actor，而不是LevelScriptActor的意思。

触发时机在用箭头选择的当前选中actor的时候。

源码中遇见：

```

if (FObjectPropertyBase* ObjectProperty = CastField< FObjectPropertyBase>
(Property))
{
    ObjectClass = ObjectProperty->PropertyClass;
    bMustBeLevelActor = ObjectProperty->GetOwnerProperty()-
>GetBoolMetaData(TEXT("MustBeLevelActor"));
    RequiredInterface = ObjectProperty->GetOwnerProperty()-
>GetClassMetaData(TEXT("MustImplement"));
}

```

ShowInnerProperties

- **功能描述:** 在属性细节面板中显示对象引用的内部属性
- **使用位置:** UPROPERTY
- **引擎模块:** Object Property
- **元数据类型:** bool
- **限制类型:** UObject*
- **关联项:** ShowOnlyInnerProperties, FullyExpand, CollapsableChildProperties
- **常用程度:** ★★★★☆

在属性细节面板中显示对象引用的内部属性。

默认情况下，对象引用属性的内部属性在细节面板里是不会显示出来的，只是孤零零的显示一个对象名字。但你如果想直接显示出其内部属性然后可以编辑的话，就需要ShowInnerProperties这个meta的作用。

但ShowInnerProperties作用有两个限定条件，一是这个属性得是UObject*，二是这个属性不是个容器。

同时也注意到，Struct属性是默认就会显示内部属性的，因此也不需要再设置ShowInnerProperties。

和EditInlineNew的区别是什么？

这种效果，和在UCLASS上设置EditInlineNew配合其对象引用属性上设置Instanced，达成的效果很相似。区别是UCLASS上设置EditInlineNew会使得一个类的对象属性引用可以在属性面板里创建对象，而UPROPERTY上的Instanced，会使得这个属性自动的增加EditInline的meta，因此也会产生显示内部属性的同样效果。因此结论上来说，和ShowInnerProperties像的是本质是EditInline这个meta。但EditInline的效果多了一层是它支持对象容器，而ShowInnerProperties只支持单个对象引用属性。

测试代码：

```
USTRUCT(BlueprintType)
struct FMyPropertyInner
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 StructInnerInt = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString StructInnerString;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_InnerSub :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 ObjectInnerInt = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString ObjectInnerString;
};
```

```

UCLASS(BlueprintType, EditInlineNew)
class INSIDER_API UMyProperty_InnerSub_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 ObjectInnerInt = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString ObjectInnerString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Inner :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FMyPropertyInner InnerStruct;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowInnerProperties))
    FMyPropertyInner InnerStruct_ShowInnerProperties;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UMyProperty_InnerSub* InnerObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowInnerProperties))
    UMyProperty_InnerSub* InnerObject_ShowInnerProperties;

    //((Category = MyProperty_Inner, EditInline = , ModuleRelativePath =
    Property/MyProperty_Inner.h)
    //CPF>Edit | CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
    UMyProperty_InnerSub* InnerObject_EditInline;

    //((Category = MyProperty_Inner, EditInline = true, ModuleRelativePath =
    Property/MyProperty_Inner.h)
    //CPF>Edit | CPF_BlueprintVisible | CPF_ExportObject | CPF_ZeroConstructor |
    CPF_InstancedReference | CPF_NoDestructor | CPF_PersistentInstance |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
    UMyProperty_InnerSub* InnerObject_Instanced;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UMyProperty_InnerSub_EditInlineNew* InnerObject_EditInlineNewClass;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
    UMyProperty_InnerSub_EditInlineNew*
    InnerObject_EditInlineNewClass_EditInline;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
    UMyProperty_InnerSub_EditInlineNew* InnerObject_EditInlineNewClass_Instanced;

public:
    UFUNCTION(CallInEditor)
    void ClearInnerObject();
    UFUNCTION(CallInEditor)
    void InitInnerObject();
};

```

```

void UMyProperty_Inner::ClearInnerObject()
{
    InnerObject = nullptr;
    InnerObject_ShowInnerProperties = nullptr;
    InnerObject_EditInline = nullptr;
    InnerObject_Instanced = nullptr;

    InnerObject_EditInlineNewClass = nullptr;
    InnerObject_EditInlineNewClass_EditInline = nullptr;
    InnerObject_EditInlineNewClass_Instanced = nullptr;

    Modify();

    FPropertyEditorModule& PropertyEditorModule =
        FModuleManager::GetModuleChecked<FPropertyEditorModule>("PropertyEditor");
    PropertyEditorModule.NotifyCustomizationModuleChanged();
}

void UMyProperty_Inner::InitInnerObject()
{
    InnerObject = NewObject<UMyProperty_InnerSub>(this);
    InnerObject_ShowInnerProperties = NewObject<UMyProperty_InnerSub>(this);
    InnerObject_EditInline = NewObject<UMyProperty_InnerSub>(this);
    InnerObject_Instanced = NewObject<UMyProperty_InnerSub>(this);

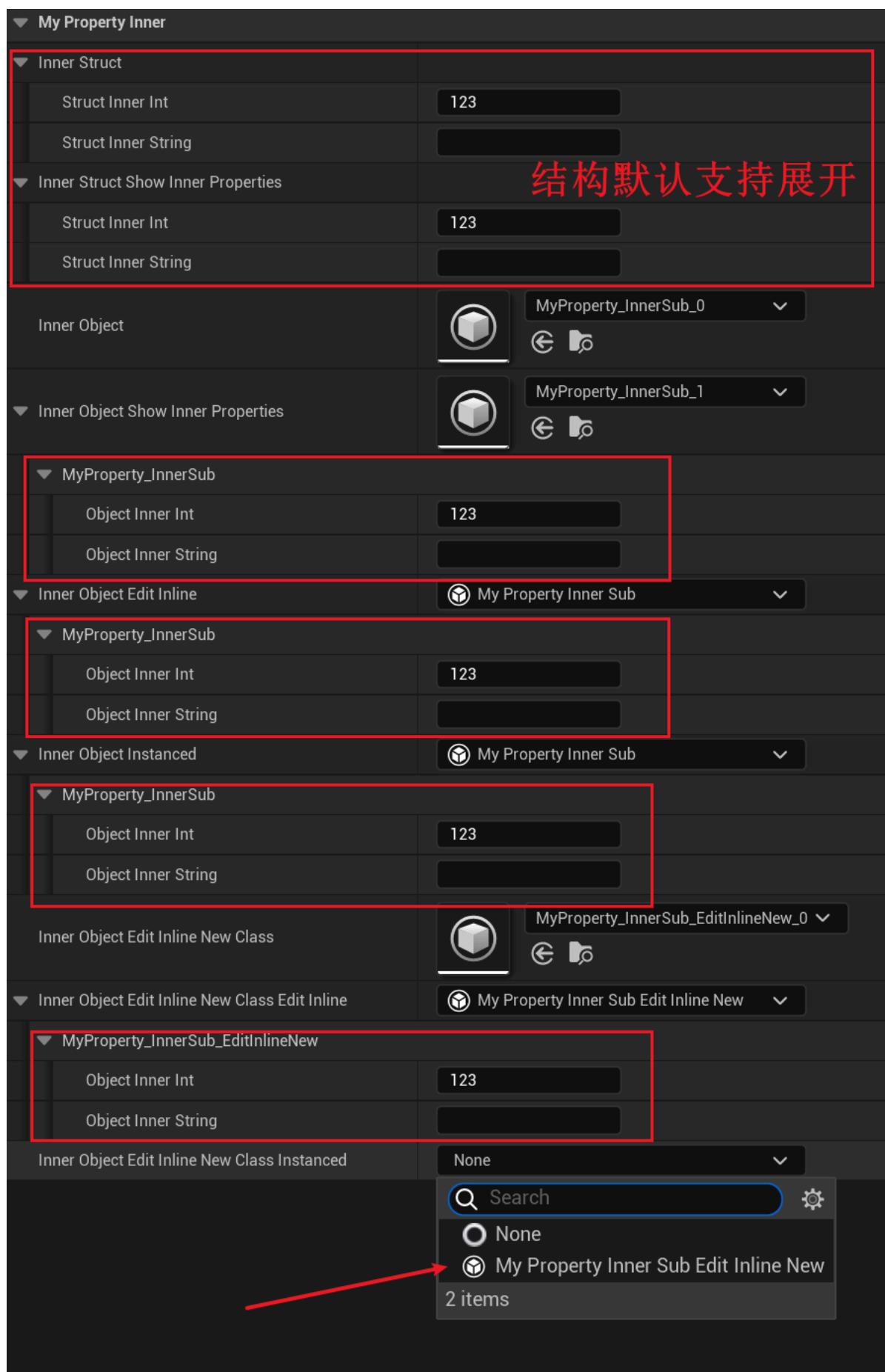
    InnerObject_EditInlineNewClass =
        NewObject<UMyProperty_InnerSub_EditInlineNew>(this);
    InnerObject_EditInlineNewClass_EditInline =
        NewObject<UMyProperty_InnerSub_EditInlineNew>(this);
    //InnerObject_EditInlineNewClass_Instanced =
    NewObject<UMyProperty_InnerSub_EditInlineNew>(this);

    Modify();

    FPropertyEditorModule& PropertyEditorModule =
        FModuleManager::GetModuleChecked<FPropertyEditorModule>("PropertyEditor");
    PropertyEditorModule.NotifyCustomizationModuleChanged();
}

```

蓝图效果：



可以观察到：

- 结构的属性默认支持展开内部属性

- 带有ShowInnerProperties的UMyProperty_InnerSub* InnerObject_ShowInnerProperties;支持展开属性
- 带有EditInline和Instanced的UMyProperty_InnerSub* 也都支持展开内部属性，也可以观察到他们的meta是一致的，都带有EditInline=true
- 只有EditInlineNew的UCLASS的UMyProperty_InnerSub_EditInlineNew* InnerObject_EditInlineNewClass;其对象引用不支持展开属性，说明在类上设置EditInlineNew并没有作用。
- 但是我们也观察到InnerObject_EditInlineNewClass_Instanced的设置里支持直接创建对象，因为其类上有EditInlineNew。而InnerObject_Instanced上并不支持直接创建对象，因为其类UMyProperty_InnerSub上并没有EditInlineNew，因此不会出现在可选框里。

扩展例子：

在源码中搜索观察到UChildActorComponent::ChildActorTemplate上也会带有ShowInnerProperties，则就是一个典型的应用，以便让我们直接在熟悉细节面板里直接编辑ChildActor的属性数据。

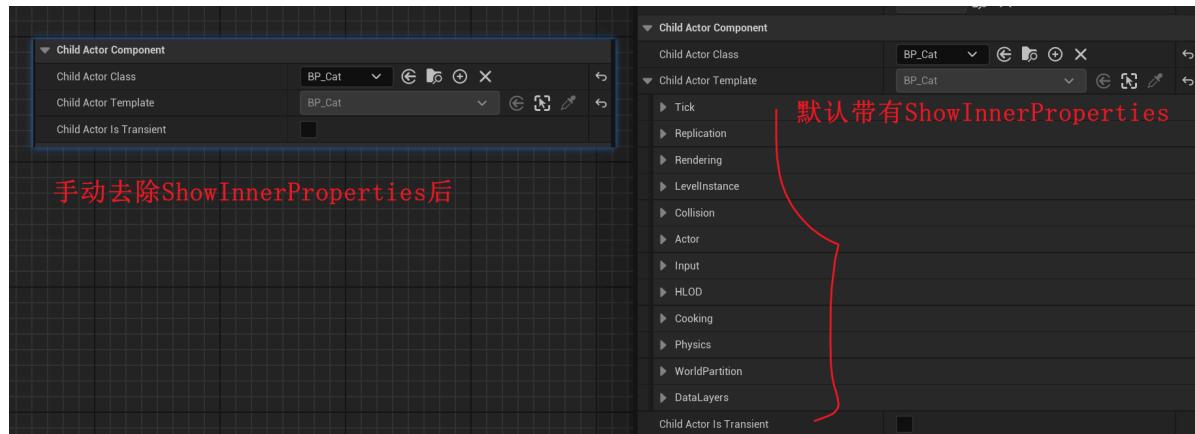
但假如我们去掉这个ShowInnerProperties，我们可以来前后对比一下效果：

```
class UChildActorComponent : public USceneComponent
{
    UPROPERTY(VisibleDefaultsOnly, DuplicateTransient,
    Category=ChildActorComponent, meta=(ShowInnerProperties))
    TObjectPtr<AActor> ChildActorTemplate;
}

void UMyProperty_Inner::RemoveActorMeta()
{
    FProperty* prop = UChildActorComponent::StaticClass()-
>FindPropertyByName(TEXT("ChildActorTemplate"));
    prop->RemoveMetaData(TEXT("ShowInnerProperties"));
}

void UMyProperty_Inner::AddActorMeta()
{
    FProperty* prop = UChildActorComponent::StaticClass()-
>FindPropertyByName(TEXT("ChildActorTemplate"));
    prop->SetMetaData(TEXT("ShowInnerProperties"), TEXT(""));
}
```

对比效果：



可以发现，去除ShowInnerProperties后，ChildActorTemplate属性退化成一个普通的对象引用，我们无法在上面直接编辑对象的内部属性。

原理：

源码里最典型的例子是ChildActorTemplate，这样就可以直接显示出内部的属性。

```
class UChildActorComponent : public USceneComponent
{
    UPROPERTY(VisibleDefaultsOnly, DuplicateTransient,
    Category=ChildActorComponent, meta=(ShowInnerProperties))
    TObjectPtr<AActor> ChildActorTemplate;
}
```

作用的源码：

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    const bool bIsObjectOrInterface = CastField<FOBJECTPROPERTYBASE>(MyProperty) || CastField<FINTERFACEPROPERTY>(MyProperty);
    // we are EditInlineNew if this property has the flag, or if inside a
    // container that has the flag.
    bIsEditInlineNew = GotReadAddresses && bIsObjectOrInterface &&
    !MyProperty->HasMetaData(NAME_NoEditInline) &&
    (MyProperty->HasMetaData(NAME_EditInline) || (bIsInsideContainer &&
    OwnerProperty->HasMetaData(NAME_EditInline)));
    bShowInnerObjectProperties = bIsObjectOrInterface && MyProperty-
    >HasMetaData(NAME_ShowInnerProperties);

    if (bIsEditInlineNew)
    {
        SetNodeFlags(EPropertyNodeFlags::EditInlineNew, true);
    }
    else if (bShowInnerObjectProperties)
    {
        SetNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties, true);
    }
}

void FItemPropertyNode::InitExpansionFlags(void)
```

```

{
    FProperty* MyProperty = GetProperty();

    if (TSharedPtr<FPropertyParams>& valueNode = GetOrCreateOptionalValueNode())
    {
        // This is a set optional, so check its SetValue instead.
        MyProperty = valueNode->GetProperty();
    }

    bool bExpandableType = CastField<FStructProperty>(MyProperty)
        || (CastField<FArrayProperty>(MyProperty) || CastField<FSetProperty>(MyProperty) || CastField<FMapProperty>(MyProperty));

    if (bExpandableType
        || HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
        || HasNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties)
        || (MyProperty->ArrayDim > 1 && ArrayIndex == -1))
    {
        SetNodeFlags(EPropertyNodeFlags::CanBeExpanded, true);
    }
}

void FPropertyParams::RebuildChildren()
{
    if (HasNodeFlags(EPropertyNodeFlags::CanBeExpanded) && (childNodes.Num() == 0))
    {
        InitChildNodes();
        if (ExpandedPropertyItemSet.size() > 0)
        {
            FPropertyParamsUtils::SetExpandedItems(ThisAsSharedRef,
ExpandedPropertyItemSet);
        }
    }
}

```

特别注意，这里的bShowInnerObjectProperties的判断条件是bIsObjectOrInterface 且有meta，因此该特性只作用于对象引用上。然后如果判断有EPropertyNodeFlags::ShowInnerObjectProperties，则继续设置EPropertyNodeFlags::CanBeExpanded，最后导致展开对象的属性。

ShowOnlyInnerProperties

- 功能描述：**把结构属性的内部属性直接上提一个层级直接展示
- 使用位置：** UPROPERTY
- 元数据类型：** bool
- 限制类型：** FStruct属性
- 关联项：** ShowInnerProperties
- 常用程度：** ★★★

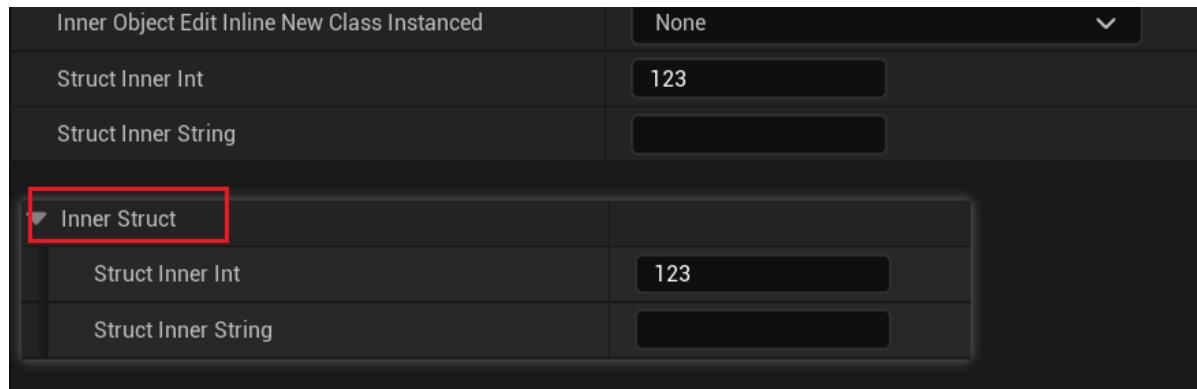
把结构属性的内部属性直接上提一个层级直接展示，而不是像默认一样归属于一个可展开的父级结构。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyPropertyInner InnerStruct;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowOnlyInnerProperties))
FMyPropertyInner InnerStruct_ShowOnlyInnerProperties;
```

效果对比：



可以发现InnerStruct_ShowOnlyInnerProperties的内部属性直接就显示在对象的当前层级上，而InnerStruct的属性有一个结构名称作为Category来展开。

原理：

在遇见FStructPropertyParams的时候，会开始判断ShowOnlyInnerProperties来决定是否要创建一个可展开的Category，或者还是直接把内部属性展示出来。有了ShowOnlyInnerProperties，就会直接递归迭代到其内部属性。

```
void DetailLayoutHelpers::updateSinglePropertyMapRecursive(FPropertyNode& InNode,
FName CurCategory, FComplexPropertyNode* CurObjectNode, FUpdatePropertyParams&
InUpdateArgs)
{
    static FName ShowOnlyInners("ShowOnlyInnerProperties");
    // whether or not to push out struct properties to their own categories
    // or show them inside an expandable struct
    // This recursively applies for any nested structs that have the
    // ShowOnlyInners metadata
    const bool bPushoutStructProps = bIsStruct && !bIsCustomizedStruct &&
    Property->HasMetaData(ShowOnlyInners);

    if (bRecurseIntoChildren || LocalUpdateFavoriteSystemOnly)
    {
        // Built in struct properties or children of arrays
        updateSinglePropertyMapRecursive(ChildNode, CurCategory,
        CurObjectNode, ChildArgs);
    }
}

void FObjectPropertyNode::GetCategoryProperties(const TSet<UClass*>&
ClassesToConsider, const FProperty* CurrentProperty, bool
bShouldShowDisableEditOnInstance, bool bShouldShowHiddenProperties,
```

```

        const TSet<FName>& CategoriesFromBlueprints, TSet<FName>&
CategoriesFromProperties, TArray<FName>& SortedCategories)
{
    if (CurrentProperty->HasMetaData(Name_ShowOnlyInnerProperties))
    {
        const FStructProperty* StructProperty = CastField<const
FStructProperty>(CurrentProperty);
        if (StructProperty)
        {
            for (TFieldIterator<FProperty> It(StructProperty->Struct);
It; ++It)
            {
                GetCategoryProperties(ClassesToConsider, *It,
bShouldShowDisableEditOnInstance, bShouldShowHiddenProperties,
CategoriesFromBlueprints, CategoriesFromProperties, SortedCategories);
            }
        }
    }
}

```

ThumbnailSize

- 功能描述:** 改变缩略图的大小。
- 使用位置:** UCLASS, UPROPERTY
- 引擎模块:** Object Property
- 元数据类型:** bool
- 关联项:** DisplayThumbnail

改变缩略图的大小。但发现并不会起作用。

原理:

```

void SObjectPropertyEntryBox::Construct( const FArguments& InArgs )
{
    // check if the property metadata wants us to display a thumbnail
    const FString& DisplayThumbnailString = PropertyHandle->GetProperty()->GetMetaData(TEXT("DisplayThumbnail"));
    if(DisplayThumbnailString.Len() > 0)
    {
        bDisplayThumbnail = DisplayThumbnailString == TEXT("true");
    }

    // check if the property metadata has an override to the thumbnail size
    const FString& ThumbnailSizeString = PropertyHandle->GetProperty()->GetMetaData(TEXT("ThumbnailSize"));
    if (ThumbnailSizeString.Len() > 0 )
    {
        FVector2D ParsedVector;
        if (ParsedVector.InitFromString(ThumbnailSizeString) )
        {
            ThumbnailSize.X = (int32)ParsedVector.X;
            ThumbnailSize.Y = (int32)ParsedVector.Y;
        }
    }
}

```

```
    }
}
}
```

Untracked

- **功能描述:** 使得TSofObjectPtr和FSofObjectPath的软对象引用类型的属性，不跟踪记录资产的。
- **使用位置:** UPROPERTY
- **引擎模块:** Object Property
- **元数据类型:** bool
- **限制类型:** TSofObjectPtr, FSofObjectPath
- **常用程度:** ★

使得TSofObjectPtr和FSofObjectPath的软对象引用类型的属性，不跟踪记录资产的。

一般默认情况，属性上的软对象引用也是会产生资产的引用依赖，虽然在Load本身的时候，不会像硬引用一样也去加载其他软引用对象。但是因为引用关系依然存在，因此在cook的时候，或者资产重定向的时候都会去检查这些软引用对象，确保其也会被cook进去，或者正常的处理。

而当你想在属性上记录“引用”一些资产，以便之后加载使用，但是又不想产生真正的资产引用依赖，这个时候就可以用untracked。源码中应用的不多，这是比较稀少的情况下。

和transient标记的区别是，transient属性在序列化的时候也不会序列化，因为其ctrl+S保存后重启编辑器会丢失值。transient属性既不产生资产引用关系也序列化保存值，Untracked属性会序列化保存值但不产生资产引用关系。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Soft :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TSofObjectPtr<UStaticMesh> MyStaticMesh;

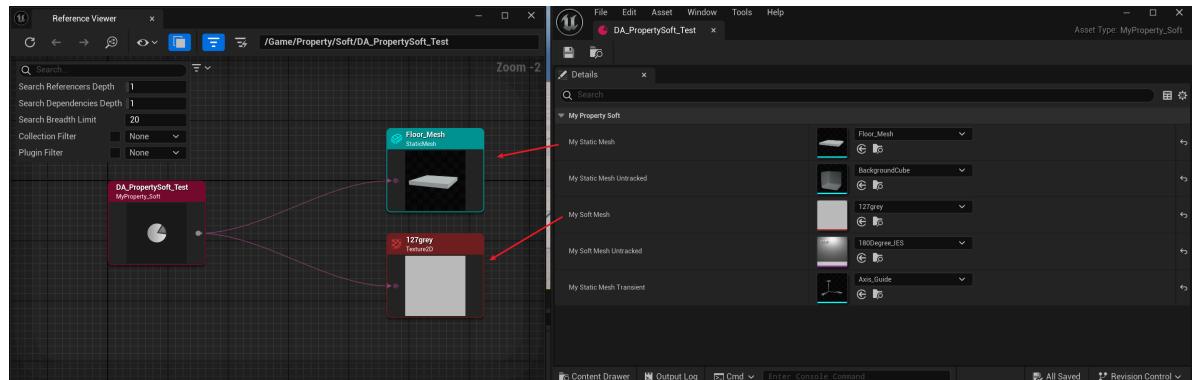
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (Untracked))
    TSofObjectPtr<UStaticMesh> MyStaticMeshUntracked;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FSofObjectPath MySoftMesh;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (Untracked))
    FSofObjectPath MySoftMeshUntracked;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
    TSofObjectPtr<UStaticMesh> MyStaticMeshTransient;
};
```

蓝图效果：

在蓝图中建立一个UMyProperty_Soft DataAsset资产，然后设置其属性值。然后查看其引用的资源，会发现Untracked的属性，其设置的资产并没有出现在引用关系中。当然Transient的属性也不在引用关系中。



原理：

Untracked元数据，会设置为ESoftObjectPathCollectType::NeverCollect的选项。继续搜索会发现带有NeverCollect的FSoftObjectPath，其上面的资产package 不会被算到资产引用里，从而不会带到upackage Import表里。源码中有多处地方带有这个NeverCollect 的类似判断。

```
bool FSoftObjectPathThreadContext::GetSerializationOptions(FName& OutPackageName,
FName& OutPropertyName, ESoftObjectPathCollectType& OutCollectType,
ESoftObjectPathSerializeType& OutSerializeType, FArchive* Archive) const
{
#if WITH_EDITOR
    bEditorOnly = Archive->IsEditorOnlyPropertyOnTheStack();

    static FName UntrackedName = TEXT("Untracked");
    if (CurrentProperty && CurrentProperty->GetOwnerProperty()->HasMetaData(UntrackedName))
    {
        // Property has the Untracked metadata, so set to never collect
        // references if it's higher than NeverCollect
        CurrentCollectType =
FMath::Min(ESoftObjectPathCollectType::NeverCollect, CurrentCollectType);
    }
#endif
}

FArchive& FImportExportCollector::operator<<(FSoftObjectPath& Value)
{
    FName CurrentPackage;
    FNamePropertyName;
    ESoftObjectPathCollectType CollectType;
    ESoftObjectPathSerializeType SerializeType;
    FSoftObjectPathThreadContext& ThreadContext =
FSoftObjectPathThreadContext::Get();
    ThreadContext.GetSerializationOptions(CurrentPackage, PropertyName,
CollectType, SerializeType, this);

    if (CollectType != ESoftObjectPathCollectType::NeverCollect && CollectType != ESoftObjectPathCollectType::NonPackage)
```

```

    {
        FName PackageName = value.GetLongPackageName();
        if (PackageName != RootPackageName && !PackageName.IsNone())
        {
            AddImport(value, CollectType);
        }
    }
    return *this;
}

```

ContentDir

- **功能描述:** 使用UE的风格来选择Content以及子目录。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** bool
- **限制类型:** FDirectoryPath
- **关联项:** RelativePath, RelativeToGameContentDir
- **常用程度:** ★★★

使用UE的风格来选择Content以及子目录。

默认情况下，选择一个目录，会弹出windows默认的选择目录对话框，因为FDirectoryPath 你确实可以用来选择到windows系统里任意的目录（可能你的项目就是这么需求的）。但如果你确实是想要选择一个UE Content下目录，这个时候你指定ContentDir，UE就可以为你弹出一个专门的UE选择目录对话框，更加的便利也避免出错。

在使用FDirectoryPath的时候，ContentDir和LongPackageName是等价的。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Path :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath)
    FDirectoryPath MyDirectory_Default;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (ContentDir))
    FDirectoryPath MyDirectory_ContentDir;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (LongPackageName))
    FDirectoryPath MyDirectory_LongPackageName;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (RelativeToGameContentDir))
    FDirectoryPath MyDirectory_RelativeToGameContentDir;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (RelativePath))

```

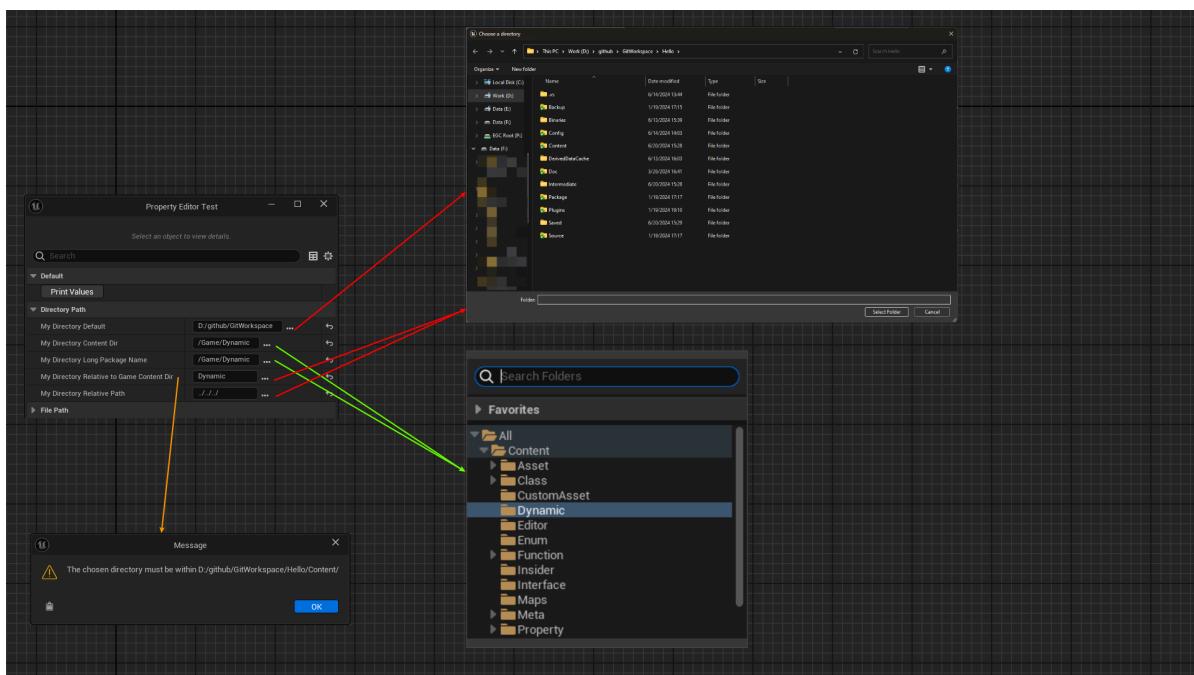
```

    FDirectoryPath MyDirectory_RelativePath;
};


```

测试结果：

- 默认的MyDirectory_Default会打开系统对话框，可以选择任何目录。
- MyDirectory_ContentDir和MyDirectory_LongPackageName，会如图所示弹出UE风格的对话框来选择目录。
- MyDirectory_RelativeToGameContentDir和MyDirectory_RelativePath都会弹出系统对话框，不同的是MyDirectory_RelativeToGameContentDir最终的目录会限制在Content目录下（如果选择别的目录，会弹出错误警告），结果是个相对路径。MyDirectory_RelativePath的结果也是个相对路径，但是可以选择任意目录。



原理：

FDirectoryPath的编辑有FDirectoryPathStructCustomization来定制化。根据其代码可见，如果有ContentDir或LongPackageName，则则是个ContentDir，则会采用OnPickContent来选择目录。内部再用ContentBrowserModule.Get().CreatePathPicker(PathPickerConfig)来创建专门的目录选择菜单。

否则走到OnPickDirectory分支，会采用DesktopPlatform->OpenDirectoryDialog来打开系统的对话框。

从源码也可以看出：

bRelativeToGameContentDir会导致Directory.RightChopInline(AbsoluteGameContentDir.Len(), EAllowShrinking::No);，把Content路径的左边部分切掉。

bUseRelativePath会触发Directory = IFileManager::Get().ConvertToRelativePath(*Directory);，把路径转换成相对路径。

```

/** Structure for directory paths that are displayed in the editor with a picker
UI. */
USTRUCT(BlueprintType)
struct FDirectoryPath
{

```

```

GENERATED_BODY()


    /**
     * The path to the directory.
     */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Path)
    FString Path;
};

RegisterCustomPropertyTypeLayout("DirectoryPath",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FDirectoryPathStructCustomization::MakeInstance));
RegisterCustomPropertyTypeLayout("FilePath",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FFilePathStructCustomization::MakeInstance));

void FDIRECTORYPATHSTRUCTCUSTOMIZATION::CustomizeHeader(
TSharedRef<IPropertyHandle> StructPropertyHandle, class FDetailWidgetRow&
HeaderRow, IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    TSharedPtr<IPropertyHandle> PathProperty = StructPropertyHandle-
>GetChildHandle("Path");

    const bool bRelativeToGameContentDir = StructPropertyHandle->HasMetaData(
TEXT("RelativeToGameContentDir") );
    const bool bUseRelativePath = StructPropertyHandle->HasMetaData(
TEXT("RelativePath") );
    const bool bContentDir = StructPropertyHandle->HasMetaData(
TEXT("ContentDir") ) || StructPropertyHandle-
>HasMetaData(TEXT("LongPackageName"));

    AbsoluteGameContentDir =
FPaths::ConvertRelativePathToFull(FPaths::ProjectContentDir());

    if(bContentDir)
    {
        PickerWidget = SAssignNew(PickerButton, SButton)
            .ButtonStyle( FAppStyle::Get(), "HoverHintOnly" )
            .ToolTipText( LOCTEXT( "FolderComboToolTipText", "Choose a content
directory" ) )
            .OnClicked( this, &FDIRECTORYPATHSTRUCTCUSTOMIZATION::OnPickContent,
PathProperty.ToSharedRef() )
            .ContentPadding(2.0f)
            .ForegroundColor( FSlateColor::UseForeground() )
            .IsFocusable(false)
            .IsEnabled(this, &FDIRECTORYPATHSTRUCTCUSTOMIZATION::IsBrowseEnabled,
StructPropertyHandle)
        [
            SNew(SImage)
                .Image(FAppStyle::GetBrush("Propertywindow.Button_Ellipsis"))
                .ColorAndOpacity(FSlateColor::UseForeground())
        ];
    }
    else
    {

}

```

```

        PickerWidget = SAssignNew(BrowseButton, SButton)
            .ButtonStyle( FAppStyle::Get(), "HoverHintOnly" )
            .ToolTipText( LOCTEXT( "FolderButtonToolTipText", "Choose a directory
from this computer" ) )
            .OnClicked( this, &FDirectoryPathStructCustomization::OnPickDirectory,
PathProperty.ToSharedRef(), bRelativeToGameContentDir, bUseRelativePath )
            .ContentPadding( 2.0f )
            .ForegroundColor( FSlateColor::UseForeground() )
            .IsFocusable( false )
            .IsEnabled( this, &FDirectoryPathStructCustomization::IsBrowseEnabled,
StructPropertyHandle )
    [
        SNew( SImage )
            .Image( FAppStyle::GetBrush("Propertywindow.Button_Ellipsis") )
            .ColorAndOpacity( FSlateColor::UseForeground() )
    ];
}
}

FReply
FDirectoryPathStructCustomization::OnPickContent(TSharedRef<IPropertyHandle>
PropertyHandle)
{
    FContentBrowserModule& ContentBrowserModule =
FModuleManager::LoadModuleChecked<FContentBrowserModule>("ContentBrowser");
    FPathPickerConfig PathPickerConfig;
    PropertyHandle->GetValue(PathPickerConfig.DefaultPath);
    PathPickerConfig.bAllowContextMenu = false;
    PathPickerConfig.OnPathSelected = FOnPathSelected::CreateSP(this,
&FDirectoryPathStructCustomization::OnPathPicked, PropertyHandle);

    FMenuBuilder MenuBuilder(true, NULL);
    MenuBuilder.AddWidget(SNew(SBox)
        .WidthOverride(300.0f)
        .HeightOverride(300.0f)
    [
        ContentBrowserModule.Get().CreatePathPicker(PathPickerConfig)
    ], FText());
}

PickerMenu = FSlateApplication::Get().PushMenu(PickerButton.ToSharedRef(),
FWidgetPath(),
MenuBuilder.MakeWidget(),
FSlateApplication::Get().GetCursorPos(),
FPopupTransitionEffect(FPopupTransitionEffect::ContextMenu)
);

return FReply::Handled();
}

FReply
FDirectoryPathStructCustomization::OnPickDirectory(TSharedRef<IPropertyHandle>
PropertyHandle, const bool bRelativeToGameContentDir, const bool
bUseRelativePath) const
{
    FString Directory;
    IDesktopPlatform* DesktopPlatform = FDesktopPlatformModule::Get();

```

```

if (DesktopPlatform)
{
    TSharedPtr<SWindow> ParentWindow =
        FSlateApplication::Get().FindWidgetWindow(BrowseButton.ToSharedRef());
    void* ParentwindowHandle = (ParentWindow.IsValid() && ParentWindow->GetNativeWindow().IsValid()) ? ParentWindow->GetNativeWindow() ->GetOSWindowHandle() : nullptr;

    FString StartDirectory =
        FEditorDirectories::Get().GetLastDirectory(ELastDirectory::GENERIC_IMPORT);
    if (bRelativeToGameContentDir && !IsValidPath(StartDirectory,
        bRelativeToGameContentDir))
    {
        StartDirectory = AbsoluteGameContentDir;
    }

    // Loop until; a) the user cancels (OpenDirectoryDialog returns false),
    or, b) the chosen path is valid (IsValidPath returns true)
    for (;;)
    {
        if (DesktopPlatform->OpenDirectoryDialog(ParentwindowHandle,
            LOCTEXT("FolderDialogTitle", "Choose a directory").ToString(), StartDirectory,
            Directory))
        {
            FText FailureReason;
            if (IsValidPath(Directory, bRelativeToGameContentDir,
                &FailureReason))
            {
                FEditorDirectories::Get().SetLastDirectory(ELastDirectory::GENERIC_IMPORT,
                    Directory);

                if (bRelativeToGameContentDir)
                {
                    Directory.RightChopInline(AbsoluteGameContentDir.Len(),
                        EAllowShrinking::No);
                }
                else if (bUseRelativePath)
                {
                    Directory =
                        IFileManager::Get().ConvertToRelativePath(*Directory);
                }

                PropertyHandle->SetValue(Directory);
            }
            else
            {
                StartDirectory = Directory;
                FMessageDialog::Open(EAppMsgType::Ok, FailureReason);
                continue;
            }
        }
        break;
    }
}

```

```
    return FReply::Handled();
}
```

FilePathFilter

- **功能描述:** 设定文件选择器的扩展名，规则符合系统对话框的格式规范，可以填写多个扩展名。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** string="abc"
- **限制类型:** FFilePath
- **常用程度:** ★★★

一般常见的用法是".umap", ".uasset"之类的。但也可以支持采用“描述 | *.后缀名”的格式自己书写过滤方式，规则同windows系统选取规则一样，也可以同时写多个后缀名。

LongPackageName

- **功能描述:** 使用UE的风格来选择Content以及子目录，或者把文件路径转换为长包名。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** bool
- **限制类型:** FDirectoryPath, FFilePath
- **常用程度:** ★★★

LongPackageName可以同时用在FDirectoryPath和FFilePath，都限制选取范围在Content目录内。

用在FDirectoryPath上的时候，限制目录为Content或其子目录。

用在FFilePath的时候，限制选择范围为Content里的资产，最终把选取的文件路径转换为"/Game/ObjectPath"这种对象的路径名。

RelativePath

- **功能描述:** 使得系统目录选择对话框的结果为当前运行exe的相对路径。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** bool
- **限制类型:** FDirectoryPath
- **关联项:** ContentDir

当前目录为：D:\github\GitWorkspace\Hello\Binaries\Win64，就是exe所在的工作目录。选择的目录会被转换为相对路径。

```
Directory = IFileManager::Get().ConvertToRelativePath(*Directory);
```

RelativeToGameContentDir

- **功能描述:** 使得系统目录选择对话框的结果为相对Content的相对路径。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** bool
- **限制类型:** FDirectoryPath
- **关联项:** ContentDir

限制目录选择的结果必须是Content目录或其子目录，否则会弹出报错信息。转换的逻辑为把左侧的Content路径裁切掉。

```
Directory.RightChopInline(AbsoluteGameContentDir.Len(), EAllowShrinking::No);
```

RelativeToGameDir

- **功能描述:** 如果系统目录选择框的结果为Project的子目录，则转换为相对路径，否则返回绝对路径。
- **使用位置:** UPROPERTY
- **引擎模块:** Path Property
- **元数据类型:** bool
- **限制类型:** FFilePath
- **常用程度:** ★★★

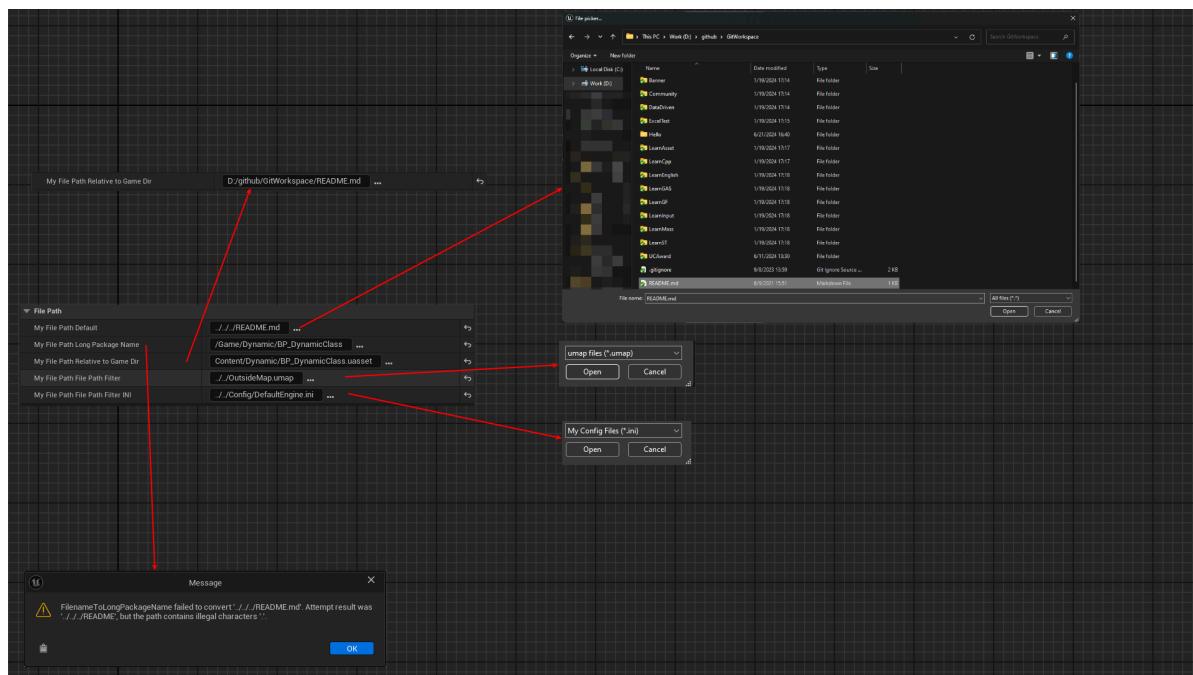
如果系统目录选择框的结果为Project的子目录，则转换为相对路径，否则返回绝对路径。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath)  
    FFilePath MyFilePath_Default;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(LongPackageName))  
    FFilePath MyFilePath_LongPackageName;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(RelativeToGameDir))  
    FFilePath MyFilePath_RelativeToGameDir;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(FilePathFilter = "umap"))  
    FFilePath MyFilePath_FilePathFilter;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(FilePathFilter = "My Config Files|*.ini"))  
    FFilePath MyFilePath_FilePathFilter_INI;
```

测试结果：

- FFilePath 弹出的都为 Windows 系统的默认文件选择对话框。
- MyFilePath_Default，弹出默认的系统文件选择对话框，可以选择任何路径的任何文件。
- MyFilePath_LongPackageName，限制选择范围为 Content 下的资产，否则会弹出报错。结果路径会被转换为 /Game/ObjectPath 这种长的包名。
- MyFilePath_RelativeToGameDir，如果选择结果为 Project 目录（uproject 所在的目录）下的子文件，则返回结果会相对路径。否则直接返回绝对路径。
- MyFilePath_FilePathFilter，可以选择任何目录下的指定后缀名的文件。代码里示例为 umap，则只能选择关卡文件。
- MyFilePath_FilePathFilter_INI，演示了只能选取 ini 文件。FilePathFilter 支持我们采用“描述 | *.后缀名”的格式自己书写过滤方式，规则同 windows 系统选取规则一样，也可以同时写多个后缀名。



原理：

在下面代码就可以看见 FilePathFilter, bLongPackageName, bRelativeToGameDir 的处理逻辑。

- FileTypeFilter 设定扩展名到 SFilePathPicker 里
- bLongPackageName 触发 TryConvertFilenameToLongPackageName 来转换路径。
- bRelativeToGameDir 触发 AbsolutePickedPath.RightChop(AbsoluteProjectDir.Len()); 来变成相对路径。

```
USTRUCT(BlueprintType)
struct FFilePath
{
GENERATED_BODY()

/**
 * The path to the file.
 */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath)
 FString FilePath;
};
```

```

void FFilePathStructCustomization::CustomizeHeader( TSharedRef<IPROPERTYHANDLE>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    const FString& MetaData = StructPropertyHandle-
>GetMetaData(TEXT("FilePathFilter"));
    bLongPackageName = StructPropertyHandle-
>HasMetaData(TEXT("LongPackageName"));
    bRelativeToGameDir = StructPropertyHandle-
>HasMetaData(TEXT("RelativeToGameDir"));

    if (MetaData.IsEmpty())
    {
        FileTypeFilter = TEXT("All files (*.*)|*.*");
    }
    else
    {
        if (MetaData.Contains(TEXT("|")))// If MetaData follows the
Description|ExtensionList format, use it as is
        {
            FileTypeFilter = MetaData;
        }
        else
        {
            FileTypeFilter = FString::Printf(TEXT("%s files (*.%s)|*.%s"),
*MetaData, *MetaData, *MetaData);
        }
    }
}

void FFilePathStructCustomization::HandleFilePathPickerPathPicked( const FString&
PickedPath )
{
    FString FinalPath = PickedPath;
    if (bLongPackageName)
    {
        FString LongPackageName;
        FString StringFailureReason;
        if (FPackageName::TryConvertFilenameToLongPackageName(PickedPath,
LongPackageName, &StringFailureReason) == false)
        {
            FMessageDialog::Open(EAppMsgType::Ok,
FText::FromString(StringFailureReason));
        }
        FinalPath = LongPackageName;
    }
    else if (bRelativeToGameDir && !PickedPath.IsEmpty())
    {
        //A filepath under the project directory will be made relative to the
project directory
        //Otherwise, the absolute path will be returned unless it doesn't exist,
the current path will
        //be kept. This can happen if it's already relative to project dir
(tabbing when selected)

```

```

        const FString ProjectDir = FPaths::ProjectDir();
        const FString AbsoluteProjectDir =
FPaths::ConvertRelativePathToFull(ProjectDir);
        const FString AbsolutePickedPath =
FPaths::ConvertRelativePathToFull(PickedPath);

        //verify if absolute path to file exists. If it was already relative to
content directory
        //the absolute will be to binaries and will possibly be garbage
        if (FPaths::FileExists(AbsolutePickedPath))
{
    //If file is part of the project dir, chop the project dir part
    //Otherwise, use the absolute path
    if (AbsolutePickedPath.StartsWith(AbsoluteProjectDir))
    {
        FinalPath =
AbsolutePickedPath.RightChop(AbsoluteProjectDir.Len());
    }
    else
    {
        FinalPath = AbsolutePickedPath;
    }
}
else
{
    //If absolute file doesn't exist, it might already be relative to
project dir
    //If not, then it might be a manual entry, so keep it untouched
either way
    FinalPath = PickedPath;
}
}

PathStringProperty->SetValue(FinalPath);
FEditorDirectories::Get().SetLastDirectory(ELastDirectory::GENERIC_OPEN,
FPaths::GetPath(PickedPath));
}

```

DataTablePin

- 功能描述:** 指定一个函数参数为DataTable或CurveTable类型，以便为FName的其他参数提供RowNameList的选择。
- 使用位置:** UFUNCTION
- 引擎模块:** Pin
- 元数据类型:** string="abc"
- 限制类型:** DataTable, CurveTable
- 常用程度:** ★★

指定一个函数参数为UDataTable类型，这样就可以加载根据DataTable里的数据，为RowName提供一个List来选择，而不是手填。支持的类型是DataTable和CurveTable

如果是DataTablePin，则会采用UK2Node_CallDataTableFunction来生成蓝图节点，以便在DataTable Pin连接参数改变的时候，触发RowNameList的刷新。

UK2Node_GetDataRow也是一个单独的蓝图节点。

但是发现RowName并没有用meta指定？但是有判断函数的参数如果类型是FName，就会把它当作RowName来看待。（这么简单粗糙的判断，也是因为其并不打算把这个Meta给用户使用）。因此实际上函数里可以有多个FName的参数，都会被自动的赋予RowNameList

在源码中用的地方只有UDataTableFunctionLibrary

```
UFUNCTION(BlueprintCallable, Category = "DataTable", meta =
(ExpandEnumAsExecs="OutResult", DataTablePin="CurveTable"))
static ENGINE_API void EvaluateCurveTableRow(UCurveTable* CurveTable, FName
RowName, float InXY, TEnumAsByte<EEvaluateCurveTableResult::Type>& OutResult,
float& OutXY, const FString& ContextString);
```

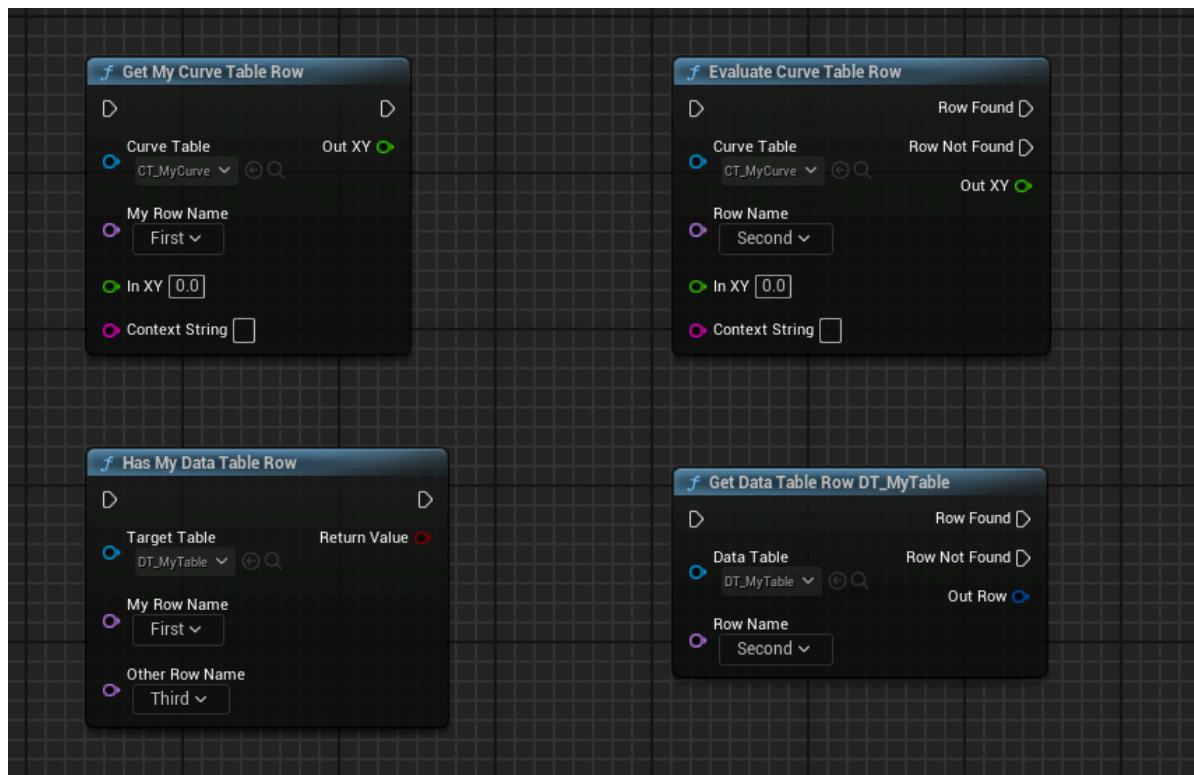
测试代码：

```
UFUNCTION(BlueprintCallable, meta = (DataTablePin="CurveTable"))
static void GetMyCurveTableRow(UCurveTable* CurveTable, FName MyRowName,
float InXY, float& OutXY, const FString& ContextString){}

UFUNCTION(BlueprintCallable, meta = (DataTablePin="TargetTable"))
static bool HasMyDataTableRow(UDataTable* TargetTable, FName MyRowName, FName
OtherRowName){return false;}
```

蓝图效果：

左侧为自己定义的函数节点，可以看见左侧的蓝图节点上的Name都变成了CurveTable和DataTable里的RowNameList，即使这些FName参数并没有什么特殊指定，但是蓝图系统里自动识别到FName类型并改变了实际的Pin Widget.



原理：

如果发现FName节点，会去尝试找DataTablePin，然后根据DataTablePin的类型来改变FName类型的Pin类型。

```
TSharedPtr<class SGraphPin> FBlueprintGraphPanelPinFactory::CreatePin(class  
UEdGraphPin* InPin) const  
{  
    if (InPin->PinType.PinCategory == UEdGraphSchema_K2::PC_Name)  
    {  
        UObject* Outer = InPin->GetOuter();  
  
        // Create drop down combo boxes for DataTable and CurveTable RowName pins  
        const UEdGraphPin* DataTablePin = nullptr;  
        if (Outer->IsA(UK2Node_CallFunction::StaticClass()))  
        {  
            const UK2Node_CallFunction* CallFunctionNode =  
            CastChecked<UK2Node_CallFunction>(Outer);  
            if (CallFunctionNode)  
            {  
                const UFunction* FunctionToCall = CallFunctionNode->  
                    GetTargetFunction();  
                if (FunctionToCall)  
                {  
                    const FString& DataTablePinName = FunctionToCall->  
                        GetMetaData(FBlueprintMetadata::MD_DataTablePin);  
                    DataTablePin = CallFunctionNode->FindPin(DataTablePinName);  
                }  
            }  
        }  
        else if (Outer->IsA(UK2Node_GetDataTableRow::StaticClass()))  
        {  
            const UK2Node_GetDataTableRow* GetDataTableRowNode =  
            CastChecked<UK2Node_GetDataTableRow>(Outer);  
            DataTablePin = GetDataTableRowNode->GetDataTablePin();  
        }  
  
        if (DataTablePin)  
        {  
            if (DataTablePin->DefaultObject != nullptr && DataTablePin->  
                LinkedTo.Num() == 0)  
            {  
                if (auto DataTable = Cast<UDataTable>(DataTablePin->  
                    DefaultObject))  
                {  
                    return SNew(SGraphPinDataTableRowName, InPin, DataTable);  
                }  
                if (DataTablePin->DefaultObject->IsA(UCurveTable::StaticClass()))  
                {  
                    UCurveTable* CurveTable = (UCurveTable*)DataTablePin->  
                        DefaultObject;  
                    if (CurveTable)  
                    {  
                        TArray<TSharedPtr<FName>> RowNames;  
                        /* Extract all the row names from the RowMap */  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        for (TMap< FName, FRealCurve*>::TConstIterator
Iterator(CurveTable->GetRowMap()); Iterator; ++Iterator)
{
    /** Create a simple array of the row names */
    TSharedPtr< FName> RowNameItem = MakeShareable(new
FName(Iterator.Key()));

    RowNames.Add(RowNameItem);
}
return SNew(SGraphPinNameList, InPin, RowNames);
}
}

return nullptr;
}

```

DisableSplitPin

- 功能描述:** 禁用Struct的split功能
- 使用位置:** USTRUCT
- 引擎模块:** Pin
- 元数据类型:** bool
- 常用程度:** ★★

对于某些Struct，特别是只有一个成员变量的结构，有时候如果按照默认的展开则会显得很怪。这个时候就希望能够禁用掉这个功能。但是注意依然可以手动在蓝图里Break来访问成员变量。如果在蓝图里也不想暴露成员变量访问，那应该在UPROPERTY上不能加BlueprintReadWrite/BlueprintReadOnly

在源码里搜索，如FGameplayTag, FPostProcessSettings, FSlatePostSettings

测试代码：

```

USTRUCT(BlueprintType, meta = (DisableSplitPin))
struct INSIDER_API FMyStruct_DisableSplitPin
{
GENERATED_BODY()

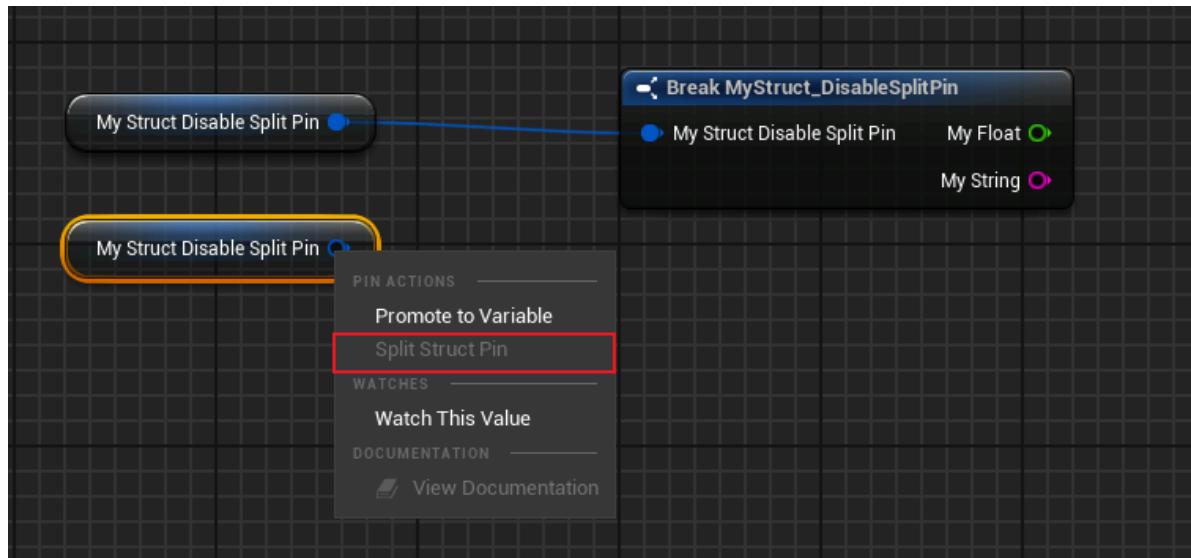
UPROPERTY(BlueprintReadWrite, EditAnywhere)
float MyFloat;
UPROPERTY(BlueprintReadWrite, EditAnywhere)
 FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_WithSplitPin
{
GENERATED_BODY()

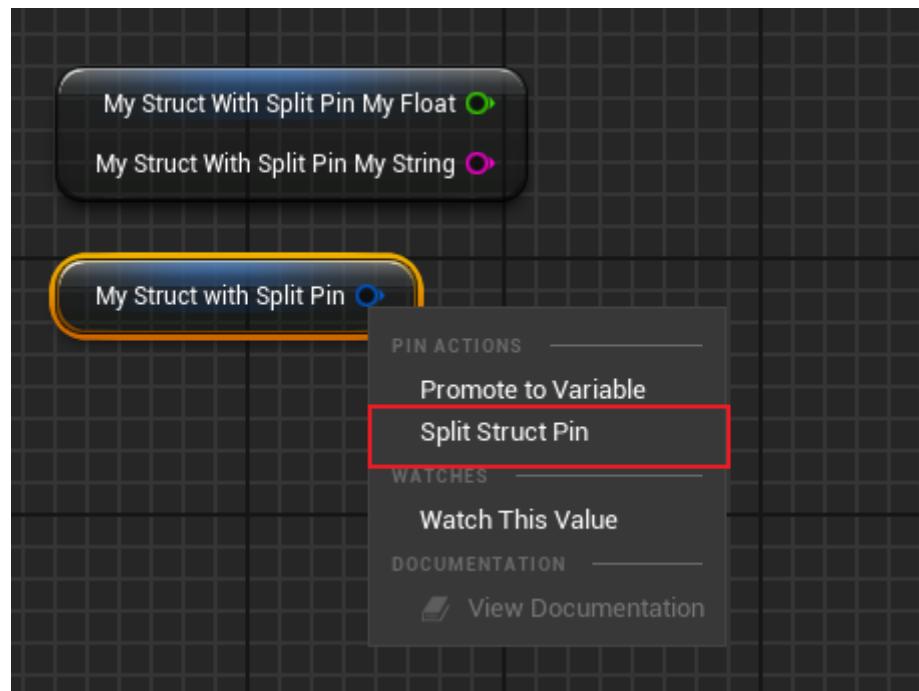
UPROPERTY(BlueprintReadWrite, EditAnywhere)
float MyFloat;

```

```
UPROPERTY(BlueprintReadWrite, EditAnywhere)
FString MyString;
};
```



允许的对比



HiddenByDefault

- **功能描述:** Struct的Make Struct和Break Struct节点中的引脚默认为隐藏状态
- **使用位置:** USTRUCT
- **引擎模块:** Pin
- **元数据类型:** bool
- **常用程度:** ★

测试代码:

```
//(BlueprintType = true, HiddenByDefault = , ModuleRelativePath =
Struct/MyStruct_HiddenByDefault.h)
```

```

USTRUCT(BlueprintType, meta = (HiddenByDefault))
struct INSIDER_API FMyStruct_HiddenByDefault
{
    GENERATED_BODY()

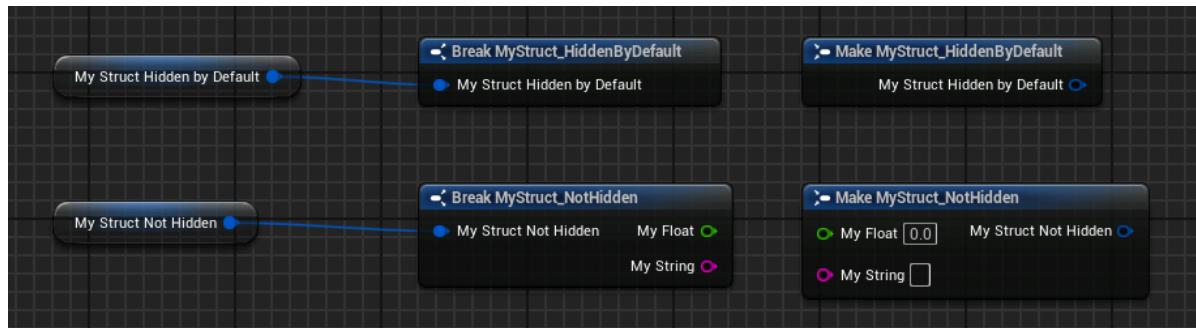
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_NotHidden
{
    GENERATED_BODY()

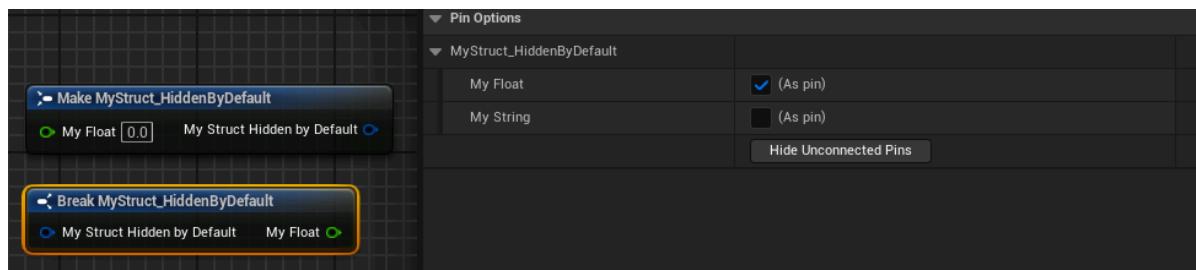
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

```

蓝图结果：



所谓隐藏，指的是在节点的细节面板里需要手动选择某些属性。而不是像默认的一样一开始就全部自动打开。



HidePin

- 功能描述:** 用在函数调用上，指定要隐藏的参数名称，也可以隐藏返回值。可以隐藏多个参数
- 使用位置:** UFUNCTION
- 引擎模块:** Pin
- 元数据类型:** strings="a, b, c"
- 关联项:** InternalUseParam
- 常用程度:** ★★

源码里倒是经常发现和DefaultToSelf比较合同。既隐藏的同时，又有默认值。一起合并的效果是把一个静态函数调用，限制在一个参数直接为外部调用环境对象。

HidePin的值也经常是WorldContextObject，

```
meta = (HidePin = "WorldContextObject", DefaultToSelf = "WorldContextObject")
```

C++测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HidePinTest :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int MyFunc_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "options"))
    int MyFunc_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_HidePin2(FName name, float value, FString options, FString comment)
    { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options"))
    int MyFunc_InternalUseParam(FName name, float value, FString options) {
    return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "ReturnValue"))
    int MyFunc_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintPure)
    int MyPure_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintPure, meta = (HidePin = "options"))
    int MyPure_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintPure, meta = (InternalUseParam = "options"))
    int MyPure_InternalUseParam(FName name, float value, FString options) {
    return 0; }

    UFUNCTION(BlueprintPure, meta = (HidePin = "ReturnValue"))
    int MyPure_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_InternalUseParams2(FName name, float value, FString
options, FString comment) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam =
"options,comment,ReturnValue"))
```

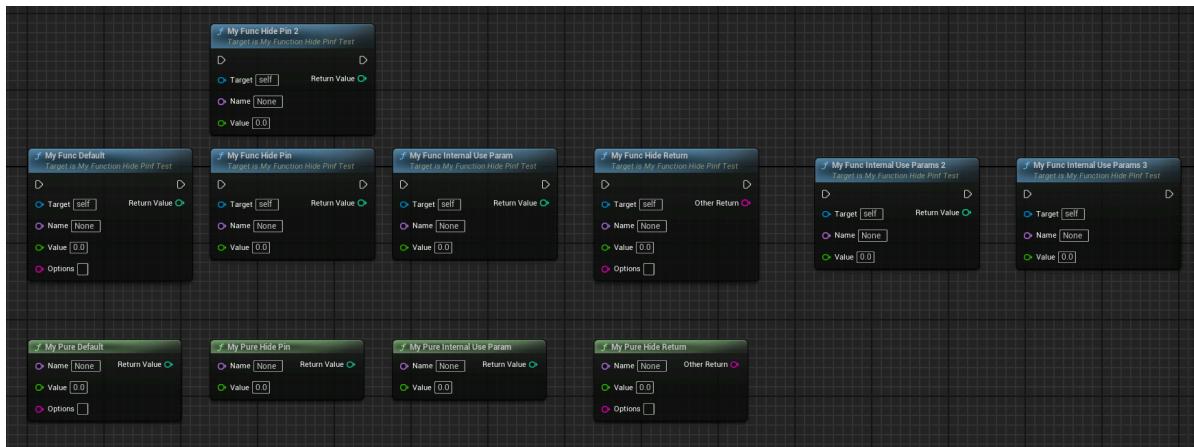
```

int MyFunc_InternalUseParams3(FName name, float value, FString
options, FString comment) { return 0; }

};

```

蓝图测试结果：



可以看出BlueprintCallable和BlueprintPure其实都可以。另外ReturnValue是默认的返回值的名字，也可以通过这个来隐藏掉。

原理：

在源码里搜索，唯一用到的是：

可以发现以下这些情况的Pin会自动被隐藏：

- LatentInfo="ParameterName"
- HidePin="ParameterName"
- InternalUseParam="ParameterName1, ParameterName2", 可以多个
- ExpandEnumAsExecs或ExpandBoolAsExecs里面指定的PinNames
- WorldContext="ParameterName", 成员函数被调用, 且C++基类有实现GetWorld, 这时WorldContext可以自动被赋予正确的World值, 就不需要显示出来了。

这个函数是被CreatePinsForFunctionCall所调用, 用来对Function的内部Property进行筛选, 也就是Params+ReturnValue, 因此是HidePin是不能用来隐藏Target这种Pin的, 这种需求应该是HideSelfPin。

```

// Gets a list of pins that should be hidden for a given function
void FBlueprintEditorUtils::GetHiddenPinsForFunction(UEdGraph const* Graph,
UFunction const* Function, TSet<FName>& HiddenPins, TSet<FName>* OutInternalPins)
{
    check(Function != nullptr);
    TMap<FName, FString>* MetaData = UMetaData::GetMapForObject(Function);
    if (MetaData != nullptr)
    {
        for (TMap<FName, FString>::TConstIterator It(*MetaData); It; ++It)
        {
            const FName& Key = It.Key();

            if (Key == FBlueprintMetadata::MD_LatentInfo)
            {

```

```

        HiddenPins.Add(*It.Value());
    }
    else if (Key == FBlueprintMetadata::MD_HidePin)
    {
        TArray< FString> HiddenPinNames;
        It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
        for ( FString& HiddenPinName : HiddenPinNames)
        {
            HiddenPinName.TrimStartAndEndInline();
            HiddenPins.Add(*HiddenPinName);
        }
    }
    else if (Key == FBlueprintMetadata::MD_ExpandEnumAsExecs || Key == FBlueprintMetadata::MD_ExpandBoolAsExecs)
    {
        TArray< FName> EnumPinNames;
        UK2Node_CallFunction::GetExpandEnumPinNames(Function, EnumPinNames);

        for (const FName& EnumName : EnumPinNames)
        {
            HiddenPins.Add(EnumName);
        }
    }
    else if (Key == FBlueprintMetadata::MD_InternalUseParam)
    {
        TArray< FString> HiddenPinNames;
        It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
        for ( FString& HiddenPinName : HiddenPinNames)
        {
            HiddenPinName.TrimStartAndEndInline();

            FName HiddenPinFName(*HiddenPinName);
            HiddenPins.Add(HiddenPinFName);

            if (OutInternalPins)
            {
                OutInternalPins->Add(HiddenPinFName);
            }
        }
    }
    else if (Key == FBlueprintMetadata::MD_WorldContext)
    {
        const UEdGraphSchema_K2* K2Schema = GetDefault<UEdGraphSchema_K2>();
        if(!K2Schema->IsStaticFunctionGraph(Graph))
        {
            bool bHasIntrinsicWorldContext = false;

            UBlueprint const* CallingContext =
FindBlueprintForGraph(Graph);
            if (CallingContext && CallingContext->ParentClass)
            {
                UClass* NativeOwner = CallingContext->ParentClass;
                while(NativeOwner && !NativeOwner->IsNative())
                {

```

HideSelfPin

- **功能描述:** 用在函数调用上，隐藏默认的SelfPin，也就是Target，导致该函数只能在OwnerClass内调用。
 - **使用位置:** UFUNCTION
 - **引擎模块:** Pin
 - **元数据类型:** bool
 - **常用程度:** ★★

用在函数调用上，隐藏默认的SelfPin，也就是Target，导致该函数只能在OwnerClass内调用。

注释里说通常与 DefaultToSelf 说明符共用。但是实际上在源码里没找到例子。

和HidePin以及InternalUseParam比较类似，不过后者可以隐藏其他参数， HideSelfPin只能隐藏SelfPin

逻辑代码：

可以看出SelfPin的bHidden 与否，受到一些情况的影响：

1. 如果是Static 函数（蓝图函数库里的函数，或者C++里的静态函数），因为默认不需要Target来调用，则默认就隐藏起SelfPin。
 2. 如果函数上带有HideSelfPin标记，则默认也隐藏且不可在外部连接，导致该函数只能在本类内调用。
 3. 如果函数是BlueprintPure函数，且当前是在OwnerClass内调用，则不需要显示SelfPin。
 4. 在源码里只找到这一个地方的判断和应用。因此可以认为HideSelfPin只隐藏Self，意思是只隐藏类成员函数被调用时候的This指针（就是Self，也就是Target），但不会隐藏Static函数被调用时候的参数，哪怕这个参数被DefaultToSelf标识。被DefaultToSelf标志只意味着这个参数的默认值为外部调用环境的Self值，并不是指这个函数节点上的SelfPin，静态函数的SelfPin总是隐藏的。被DefaultToSelf标识的那个参数虽然值等于Self，但并不是SelfPin。

```

bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
//...
    if (bIsStaticFunc)
    {
        // For static methods, wire up the self to the CDO of the class if
        it's not us
        if (!bIsFunctionCompatibleWithSelf)
        {
            UClass* AuthoritativeClass = FunctionOwnerClass-
>GetAuthoritativeClass();
            SelfPin->DefaultObject = AuthoritativeClass->GetDefaultObject();
        }

        // Purity doesn't matter with a static function, we can always hide
        the self pin since we know how to call the method
        SelfPin->bHidden = true;
    }
    else
    {
        if (Function->GetBoolMetaData(FBlueprintMetadata::MD_HideSelfPin))
        {
            SelfPin->bHidden = true;
            SelfPin->bNotConnectable = true;
        }
        else
        {
            // Hide the self pin if the function is compatible with the
            blueprint class and pure (the !bIsConstFunc portion should be going away soon too
            hopefully)
            SelfPin->bHidden = (bIsFunctionCompatibleWithSelf && bIsPureFunc
&& !bIsConstFunc);
        }
    }
}

```

C++测试代码：

```

UCLASS()
class INSIDER_API UMyFunctionLibrary_SelfPinTest :public
UBlueprintFunctionLibrary
{
GENERATED_BODY()

UFUNCTION(BlueprintPure,meta=(DefaultToSelf="myOwner",hidePin="myOwner"))
static FString PrintProperty_HasDefaultToSelf_ButHide(UObject* myOwner,FName
propertyName);

UFUNCTION(BlueprintPure,meta=(DefaultToSelf="myOwner",HideSelfPin="true"))
static FString PrintProperty_HasDefaultToSelf_AndHideSelf(UObject*
myOwner,FName propertyName);

UFUNCTION(BlueprintPure,meta=
(DefaultToSelf="myOwner",InternalUseParam="myOwner"))

```

```

static FString PrintProperty_HasDefaultToSelf_ButInternal(Uobject* myOwner, FName propertyName);
};

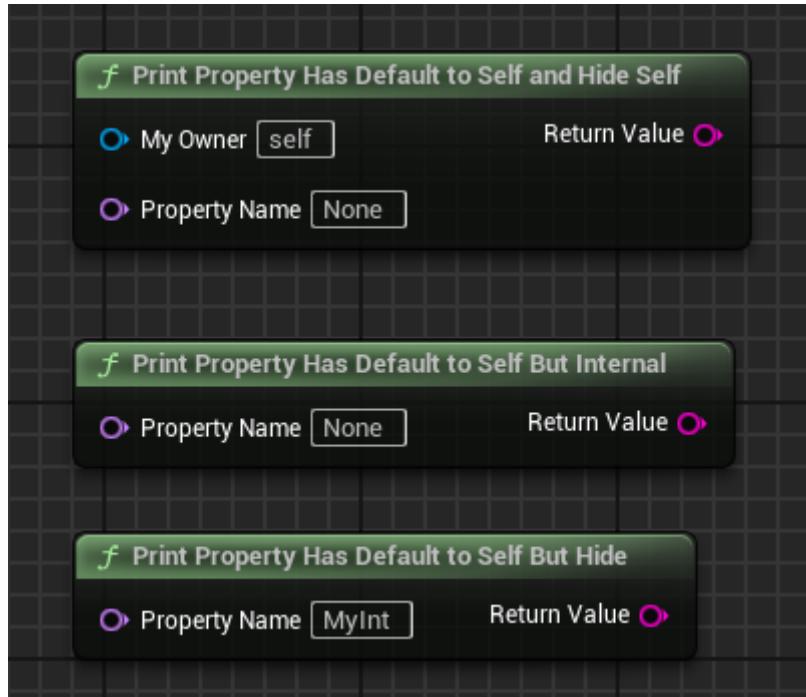
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HideSelfTest :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default(int value){}

    UFUNCTION(BlueprintCallable, meta=(HideSelfPin="true"))
    void MyFunc_HideSelfPin(int value){}
};

```

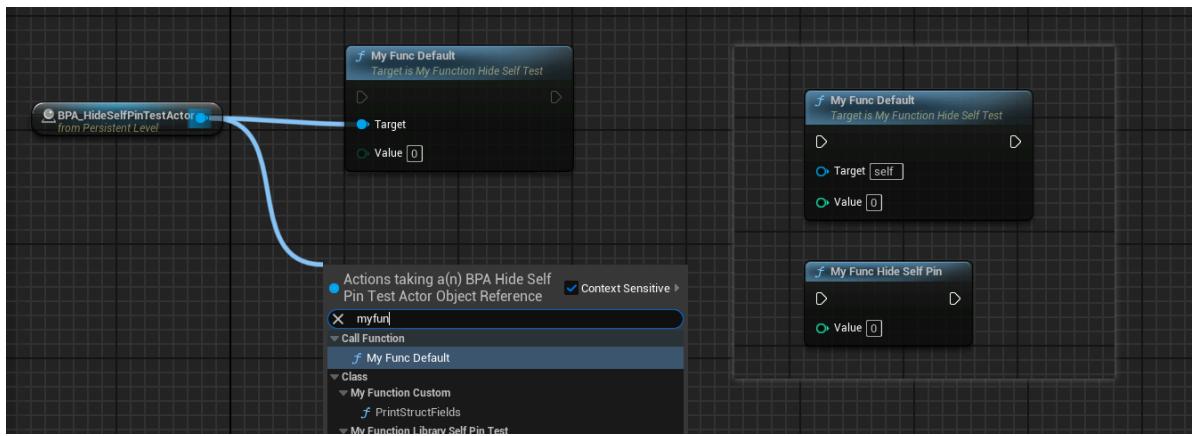
蓝图里测试效果：

从第一个图可以看到， HideSelfPin用在Static函数并无效果。而InternalUseParam可以隐藏引脚。



而对于类成员函数的测试结果可以看出：

- 在类内部调用的时候， self可以被隐藏起来，但都可以调用。二者的区别是， MyFunc_Default这个默认的版本，也可以接受同类型的AMyFunction_HideSelfTest 不同对象实例来调用。而 MyFunc_HideSelfPin只能被当前的对象来调用。
- 而在左侧关卡蓝图中，通过一个AMyFunction_HideSelfTest 对象尝试调用这两个函数，可以发现 MyFunc_Default可以调用，而MyFunc_HideSelfPin这个函数节点就无法被创建出来。就算用复制粘贴的方法硬创造出来，也因为失去了Self这个Target Pin而无法连接，从而无法调用。



InternalUseParam

- 功能描述:** 用在函数调用上，指定要隐藏的参数名称，也可以隐藏返回值。可以隐藏多个参数
- 使用位置:** UFUNCTION
- 引擎模块:** Pin
- 元数据类型:** strings="a, b, c"
- 关联项:** HidePin
- 常用程度:** ★★

该元数据和HidePin是等价的。

C++测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HidePinTest :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int MyFunc_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "options"))
    int MyFunc_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_HidePin2(FName name, float value, FString options,FString comment)
    { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options"))
    int MyFunc_InternaluseParam(FName name, float value, FString options) {
    return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "ReturnValue"))
    int MyFunc_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintPure)
    int MyPure_Default(FName name, float value, FString options) { return 0; }

```

```

UFUNCTION(BlueprintPure, meta = (HidePin = "options"))
int MyPure_HidePin(FName name, float value, FString options) { return 0; }

UFUNCTION(BlueprintPure, meta = (InternalUseParam = "options"))
int MyPure_InternalUseParam(FName name, float value, FString options) {
return 0; }

UFUNCTION(BlueprintPure, meta = (HidePin = "ReturnValue"))
int MyPure_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

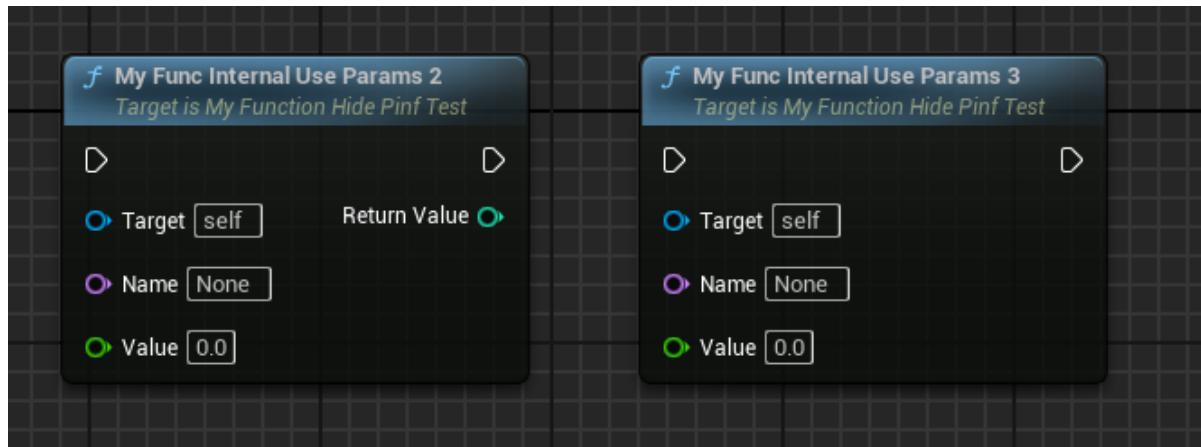
public:
UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
int MyFunc_InternalUseParams2(FName name, float value, FString
options,FString comment) { return 0; }

UFUNCTION(BlueprintCallable, meta = (InternalUseParam =
"options,comment,ReturnValue"))
int MyFunc_InternalUseParams3(FName name, float value, FString
options,FString comment) { return 0; }

};

```

蓝图测试结果：



可以看出BlueprintCallable和BlueprintPure其实都可以。另外ReturnValue是默认的返回值的名字，也可以通过这个来隐藏掉。

原理：

可见MD_InternalUseParam的使用也是在隐藏引脚。

```

// Gets a list of pins that should hidden for a given function
void FBlueprintEditorUtils::GetHiddenPinsForFunction(UEdGraph const* Graph,
UFunction const* Function, TSet<FName>& HiddenPins, TSet<FName>* OutInternalPins)
{
    check(Function != nullptr);
    TMap<FName, FString>* MetaData = UMetaData::GetMapForObject(Function);
    if (MetaData != nullptr)
    {
        for (TMap<FName, FString>::TConstIterator It(*MetaData); It; ++It)

```

```

{
    const FName& Key = It.Key();

    if (Key == FBlueprintMetadata::MD_LatentInfo)
    {
        HiddenPins.Add(*It.Value());
    }
    else if (Key == FBlueprintMetadata::MD_HidePin)
    {
        TArray< FString> HiddenPinNames;
        It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
        for (FString& HiddenPinName : HiddenPinNames)
        {
            HiddenPinName.TrimStartAndEndInline();
            HiddenPins.Add(*HiddenPinName);
        }
    }
    else if (Key == FBlueprintMetadata::MD_ExpandEnumAsExecs || 
              Key == FBlueprintMetadata::MD_ExpandBoolAsExecs)
    {
        TArray< FName> EnumPinNames;
        UK2Node_CallFunction::GetExpandEnumPinNames(Function,
                                                     EnumPinNames);

        for (const FName& EnumName : EnumPinNames)
        {
            HiddenPins.Add(EnumName);
        }
    }
    else if (Key == FBlueprintMetadata::MD_InternalUseParam)
    {
        TArray< FString> HiddenPinNames;
        It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
        for (FString& HiddenPinName : HiddenPinNames)
        {
            HiddenPinName.TrimStartAndEndInline();

            FName HiddenPinFName(*HiddenPinName);
            HiddenPins.Add(HiddenPinFName);

            if (OutInternalPins)
            {
                OutInternalPins->Add(HiddenPinFName);
            }
        }
    }
    else if (Key == FBlueprintMetadata::MD_WorldContext)
    {
        const UEdGraphSchema_K2* K2Schema = GetDefault<UEdGraphSchema_K2>
();
        if(!K2Schema->IsStaticFunctionGraph(Graph))
        {
            bool bHasIntrinsicWorldContext = false;

            UBlueprint const* CallingContext =
FindBlueprintForGraph(Graph);

```

```

        if (CallingContext && CallingContext->ParentClass)
        {
            UClass* NativeOwner = CallingContext->ParentClass;
            while(NativeOwner && !NativeOwner->IsNative())
            {
                NativeOwner = NativeOwner->GetSuperclass();
            }

            if(Nativeowner)
            {
                bHasIntrinsicworldContext = NativeOwner-
>GetDefaultObject()->ImplementsGetWorld();
            }
        }

        // if the blueprint has world context that we can lookup with
        "self",
        // then we can hide this pin (and default it to self)
        if (bHasIntrinsicworldContext)
        {
            HiddenPins.Add(*It.Value());
        }
    }
}
}
}

```

PinHiddenByDefault

- 功能描述:** 使得这个结构里的属性在蓝图里作为引脚时默认是隐藏的。
- 使用位置:** UPROPERTY
- 引擎模块:** Pin
- 元数据类型:** bool
- 限制类型:** struct member property
- 常用程度:** ★★

使得这个结构里的属性在蓝图里作为引脚时默认是隐藏的。

需要注意的是这个meta只作用于结构的成员属性，且只作用在蓝图里的节点。在有些时候一个结构里拥有多个属性，但不是希望一下子暴露所有属性来让用户编辑，有些属性可能是高级属性需要一开始隐藏起来。

该标记也可以在动画蓝图中，让动画节点的某些属性不暴露成引脚。

测试代码：

```

USTRUCT(BlueprintType)
struct FMyPinHiddenTestStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=PinHiddenByDefaultTest)

```

```

int32 MyInt_NotHidden = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=PinHiddenByDefaultTest,
meta = (PinHiddenByDefault))
    int32 MyInt_PinHiddenByDefault = 123;
};

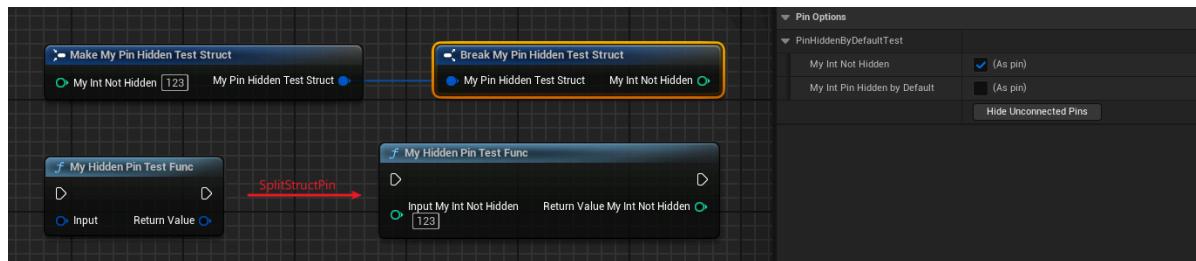
UFUNCTION(BlueprintCallable)
static FMyPinHiddenTestStruct MyHiddenPinTestFunc(FMyPinHiddenTestStruct
Input);

```

测试结果：

可以发现MakeStruct和BreakStruct的节点，默认只有MyInt_NotHidden。当选择该蓝图节点时，可以在右侧细节面板看到MyInt_PinHiddenByDefault 的显示默认没有选中，这就是区别。

同样的，当该结构当作函数输入和输出参数的时候，当用SplitStructPin展开结构节点，会发现MyInt_PinHiddenByDefault 也被隐藏了起来。



原理：

通过源码可以发现，FStructOperationOptionalPinManager 里使用了该meta，然后 FMakeStructPinManager 和 FBreakStructPinManager 继承了，从而使得PinHiddenByDefault的Pin一开始不显示。

```

struct FStructOperationOptionalPinManager : public FOptionalPinManager
{
    //~ Begin FOptionalPinsUpdater Interface
    virtual void GetRecordDefaults(FProperty* TestProperty,
    FOptionalPinFromProperty& Record) const override
    {
        Record.bCanToggleVisibility = true;
        Record.bShowPin = true;
        if (TestProperty)
        {
            Record.bShowPin = !TestProperty->HasMetaData(TEXT("PinHiddenByDefault"));
            if (Record.bShowPin)
            {
                if (UStruct* OwnerStruct = TestProperty->GetOwnerStruct())
                {
                    Record.bShowPin = !OwnerStruct->HasMetaData(TEXT("HiddenByDefault"));
                }
            }
        }
    }
}

```

```

    virtual void CustomizePinData(UEdGraphPin* Pin, FName SourcePropertyName,
int32 ArrayIndex, FProperty* Property) const override;
    // End of FOptional PinsUpdater interface
};

struct FMakeStructPinManager : public FStructOperationOptionalPinManager
{}
struct FBreakStructPinManager : public FStructOperationOptionalPinManager
{}

```

Abstract

- 功能描述:** 标识该FRigUnit为抽象类，不用实现Execute。
- 使用位置:** USTRUCT
- 引擎模块:** RigVMStruct
- 元数据类型:** bool
- 限制类型:** FRigUnit类型上
- 常用程度:** ★★

标识该FRigUnit为抽象类，不用实现Execute，常常用作别的FRigUnit类的基类使用。

但如果还是实现了Execute，其实也还是可以在蓝图中调用的。

测试代码：

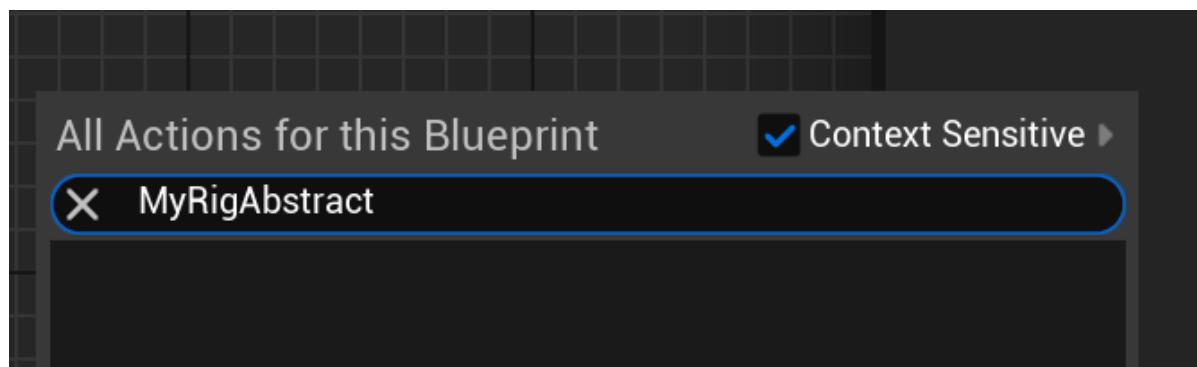
```

USTRUCT(meta = (DisplayName = "MyRigAbstract", Abstract))
struct INSIDER_API FRigUnit_MyRigAbstract : public FRigUnit
{
    GENERATED_BODY()
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};

```

测试效果：



原理：

在一些内部处理的时候，当然会略过这种抽象基类。

```
void FRigVMBuildingBlocks::ForAllRigVMStructs(TFunction<void(UScriptStruct*)> InFunction)
{
    // Run over all unit types
    for(TObjectIterator<UObject> StructIt; StructIt; ++StructIt)
    {
        if (*StructIt)
        {
            if(StructIt->IsChildOf(FRigVMStruct::StaticStruct()) && !StructIt->HasMetaData(FRigVMStruct::AbstractMetaName))
            {
                if (UScriptStruct* ScriptStruct = Cast<UScriptStruct>(*StructIt))
                {
                    InFunction(ScriptStruct);
                }
            }
        }
    }
}
```

Aggregate

- **功能描述：**指定FRigUnit里的属性引脚为可扩展连续二元运算符的运算数。
- **使用位置：** UPROPERTY
- **引擎模块：** RigVMStruct
- **元数据类型：** bool
- **限制类型：** FRigUnit下的属性
- **常用程度：** ★★★

指定FRigUnit里的属性引脚为可扩展连续二元运算符的运算数。

记得在Input和Output上都加上Aggregate。

测试代码：

```
USTRUCT(meta = (DisplayName = "MyRigAggregate"))
struct INSIDER_API FRigUnit_MyRigAggregate : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input, Aggregate))
    float A = 0.f;

    UPROPERTY(meta = (Input, Aggregate))
    float B = 0.f;
```

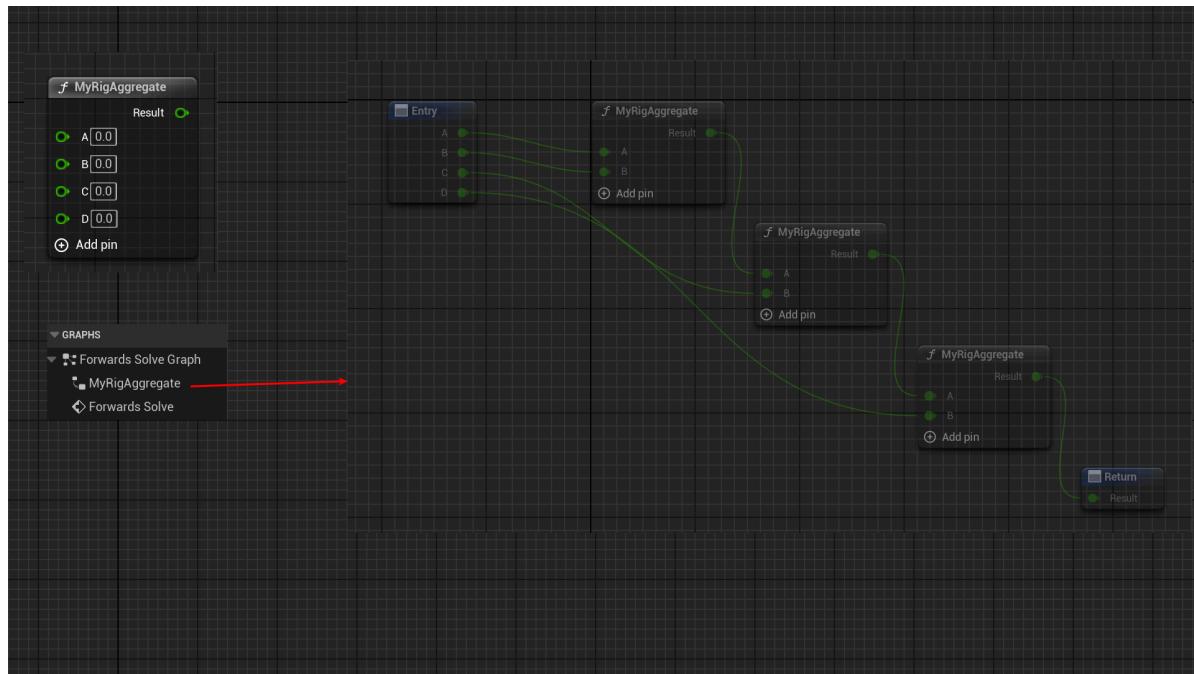
```

UPROPERTY(meta = (Output, Aggregate))
float Result = 0.f;
};

```

测试效果：

可见加了Aggregate之后，在蓝图节点上就可以继续动态AddPin。在左侧的Graph上也会创建中间MyRigAggregate节点。点开后，可以看见，其实就是继续组装原始的二元运算来达成继续AddPin的效果。



原理：

识别该Meta然后然后把引脚加到AggregateInputs和AggregateOutputs里。

```

TArray<URigVMPin*> URigVMUnitNode::GetAggregateInputs() const
{
    TArray<URigVMPin*> AggregateInputs;
#if UE_RIGVM_AGGREGATE_NODES_ENABLED
    if (const UScriptStruct* Struct = GetScriptStruct())
    {
        for (URigVMPin* Pin : GetPins())
        {
            if (Pin->GetDirection() == ERigVMPinDirection::Input)
            {
                if (const FProperty* Property = Struct->FindPropertyByName(Pin->GetFName()))
                {
                    if (Property->HasMetaData(FRigVMStruct::AggregateMetaName))
                    {
                        AggregateInputs.Add(Pin);
                    }
                }
            }
        }
    }
}

```

```

    else
    {
        return Super::GetAggregateInputs();
    }
#endif
    return AggregateInputs;
}

```

Constant

- 功能描述:** 标识一个属性成为一个常量的引脚。
- 使用位置:** UPROPERTY, USTRUCT
- 引擎模块:** RigVMStruct
- 元数据类型:** bool
- 关联项:** Input
- 常用程度:** ★★★

放在UPROPERTY上的时候，和Visible一样，标识一个属性成为一个常量的引脚。

放在USTRUCT上的时候，发现用在IsDefinedAsConstant这种函数上，但是F5没有发现调用的地方。

```

USTRUCT(meta = (DisplayName = "Rotation order", Category = "Math|Quaternion",
Constant))
struct RIGVM_API FRigVMFunction_MathQuaternionRotationOrder : public
FRigVMFunction_MathBase
{
}

```

CustomWidget

- 功能描述:** 指定该FRigUnit里的属性要用自定义的控件来编辑。
- 使用位置:** UPROPERTY
- 引擎模块:** RigVMStruct
- 元数据类型:** string="abc"
- 限制类型:** FRigUnit中的属性
- 常用程度:** ★★

指定该FRigUnit里的属性要用自定义的控件来编辑。

CustomWidget的值是在一些选项中选择的，这些自定义控件是已经在引擎中实现的。

可用的列表为：BoneName, ControlName, SpaceName/NullName, CurveName, ElementName, ConnectorName, DrawingName, ShapeName, AnimationChannelName, MetadataName, MetadataTagName。

测试代码里就只用BoneName作为测试展示：

```

USTRUCT(meta = (DisplayName = "MyRigCustomWidget"))
struct INSIDER_API FRigUnit_MyRigCustomWidget : public FRigUnit
{
}

```

```
GENERATED_BODY()

RIGVM_METHOD()
    virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    FString MyString;

    UPROPERTY(meta = (Input, CustomWidget = "BoneName"))
    FString MyString_Custom;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

测试效果：

可见MyString_Custom的Pin类型变成可选BoneName的列表。

f MyRigCustomWidget

My Float Output 

 My String

 My String Custom   

Search

- None
- ball_l
- ball_r
- calf_l
- calf_r
- calf_twist_01_l
- calf_twist_01_r
- clavicle_l
- clavicle_r
- foot_l
- foot_r
- hand_l
- hand_r
- head
- ik_foot_l
- ik_foot_r
- ik_foot_root
- ik_hand_gun
- ik_hand_l
- ik_hand_r
- ik_hand_root
- index_01_l
- index_01_r
- index_02_l
- index_02_r
- index_03_l
- index_03_r
- lowerarm_l
- lowerarm_r
- lowerarm_twist_01_l

原理：

```
TSharedPtr<SGraphPin>
FControlRigGraphPanelPinFactory::CreatePin_Internal(UEdGraphPin* InPin) const
{
    if (CustomWidgetName == TEXT("BoneName"))
    {
        return SNew(SRigVMGraphPinNameList, InPin)
            .ModelPin(ModelPin)
            .OnGetNameFromSelection_UObject(RigGraph,
&UControlRigGraph::GetSelectedElementsNameList)
            .OnGetNameListContent_UObject(RigGraph,
&UControlRigGraph::GetBoneNameList)
            .OnGetSelectedClicked_UObject(RigGraph,
&UControlRigGraph::HandleGetSelectedClicked)
            .OnBrowseClicked_UObject(RigGraph,
&UControlRigGraph::HandleBrowseClicked);
    }
    //等等其他
}
```

Deprecated

- 功能描述：**标识该FRigUnit为弃用状态，不在蓝图右键菜单中显示。
- 使用位置：** USTRUCT
- 引擎模块：** RigVMStruct
- 元数据类型：** bool
- 限制类型：** FRigUnit类型上
- 常用程度：** ★★

标识该FRigUnit为弃用状态，不在蓝图右键菜单中显示。

但如果之前已经在蓝图中使用了，则还是可以继续使用。

注意这个时候要相应的实现GetUpgradeInfo()，否则会报错。

测试代码：

```
USTRUCT(meta = (DisplayName = "MyRigDeprecated", Deprecated))
struct INSIDER_API FRigUnit_MyRigDeprecated : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
    virtual void Execute() override;

    RIGVM_METHOD()
    virtual FRigVMStructUpgradeInfo GetUpgradeInfo() const override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;
```

```
UPROPERTY(meta = (Output))
float MyFloat_Output = 123.f;
};
```

测试效果：



原理：

在构建菜单项的时候略过Deprecated的节点。

```
void FRigVMEditorModule::GetTypeActions(URigVMBLueprint* RigVMBLueprint,
FBBlueprintActionDatabaseRegistrar& ActionRegistrar)
{
    // Add all rig units
    for(const FRigVMFunction& Function : Registry.GetFunctions())
    {
        // skip deprecated units
        if(Function.Struct->HasMetaData(FRigVMStruct::DeprecatedMetaName))
        {
            continue;
        }
    }
}
```

DetailsOnly

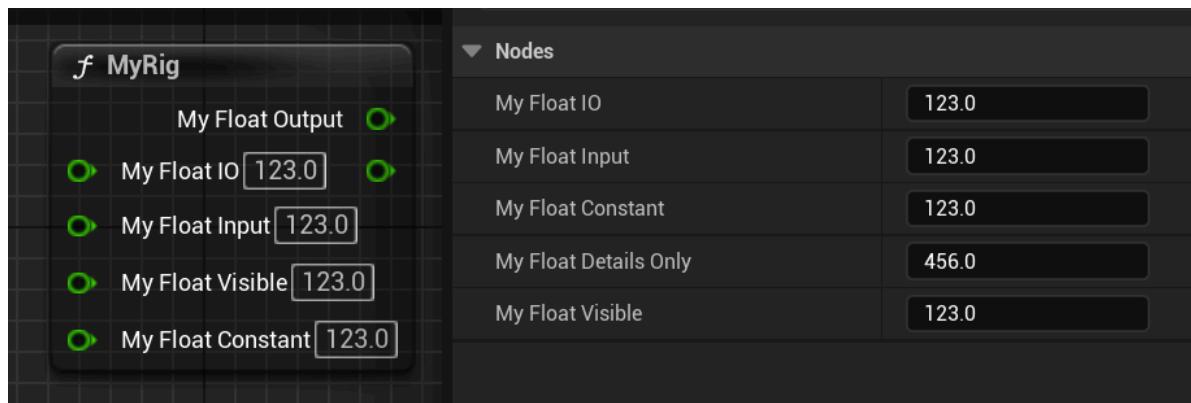
- 功能描述：**指定FRigUnit下的该属性只在细节面板中显示。
- 使用位置：** UPROPERTY
- 引擎模块：** RigVMStruct
- 元数据类型：** bool
- 限制类型：** FRigUnit下的属性
- 关联项：** Input
- 常用程度：** ★★★

指定FRigUnit下的该属性只在细节面板中显示。

测试代码：

```
UPROPERTY(meta = (Input, DetailsOnly))
float MyFloat_DetailsOnly = 456.f;
```

测试效果：



原理：

根据DetailsOnly判断返回是否ShowInDetailsPanelOnly。

```
bool URigVMPin::ShowInDetailsPanelOnly() const
{
#if WITH_EDITOR
    if (GetParentPin() == nullptr)
    {
        if (URigVMUnitNode* UnitNode = Cast<URigVMUnitNode>(GetNode()))
        {
            if (UScriptStruct* ScriptStruct = UnitNode->GetScriptStruct())
            {
                if (FProperty* Property = ScriptStruct-
>FindPropertyByName(Get FName()))
                {
                    if (Property->HasMetaData(FRigVMStruct::DetailsOnlyMetaName))
                    {
                        return true;
                    }
                }
            }
        }
        else if (const URigVMTemplateNode* TemplateNode = Cast<URigVMTemplateNode>(GetNode()))
        {
            if (const FRigVMTemplate* Template = TemplateNode->GetTemplate())
            {
                return !Template->GetArgumentMetaData(Get FName(),
FRigVMStruct::DetailsOnlyMetaName).IsEmpty();
            }
        }
    }
#endif
    return false;
}
```

ExpandByDefault

- **功能描述：**把FRigUnit里的属性引脚默认展开。

- **使用位置:** UPROPERTY
- **引擎模块:** RigVMStruct
- **元数据类型:** bool
- **常用程度:** ★★★

把FRigUnit里的属性引脚默认展开。

测试代码：

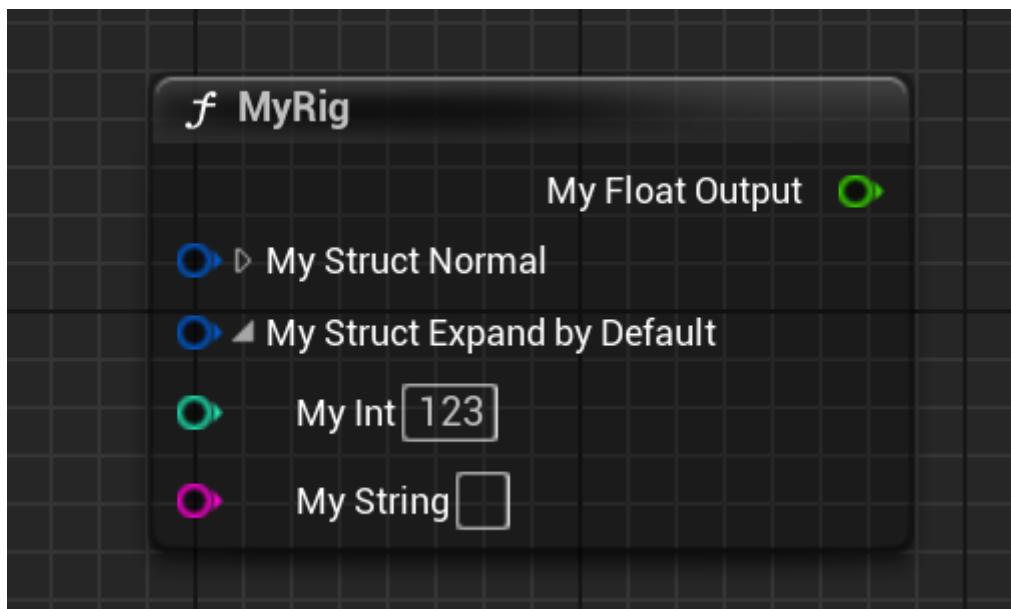
```
USTRUCT(meta = (DisplayName = "MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    UPROPERTY(meta = (Input))
    FMyCommonStruct MyStruct_Normal;

    UPROPERTY(meta = (Input, ExpandByDefault))
    FMyCommonStruct MyStruct_ExpandByDefault;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
}
```

测试效果：

可见MyStruct_ExpandByDefault默认状态下就把该结构展开。



原理：

识别该Meta然后设定该引脚的bIsExpanded状态。

```

FRigVMPinInfo::FRigVMPinInfo(FProperty* InProperty, ERigVMPinDirection
InDirection, int32 InParentIndex, const uint8* InDefaultValueMemory)
{
    if (InProperty->HasMetaData(FRigVMStruct::ExpandPinByDefaultMetaName))
    {
        bIsExpanded = true;
    }
}

```

Hidden

- 功能描述:** 指定FRigUnit下的该属性隐藏
- 使用位置:** UPROPERTY
- 引擎模块:** RigVMStruct
- 元数据类型:** bool
- 限制类型:** FRigUnit中属性
- 关联项:** Input
- 常用程度:** ★★★

Icon

- 功能描述:** 设定FRigUnit蓝图节点的图标。
- 使用位置:** USTRUCT
- 引擎模块:** RigVMStruct
- 元数据类型:** string="abc"
- 限制类型:** FRigUnit
- 常用程度:** ★★

设定FRigUnit蓝图节点的图标。

根据源码中的注释得知，Icon的格式是

"StyleSetName | StyleName | SmallStyleName | StatusOverlayStyleName"，最后两项是可选的，可参考FSlateIcon的更多介绍。

测试代码：

```

USTRUCT(meta = (DisplayName =
"MyRigIcon", Icon="EditorStyle|GraphEditor.Macro.ForEach_16x"))
struct INSIDER_API FRigUnit_MyRigIcon: public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

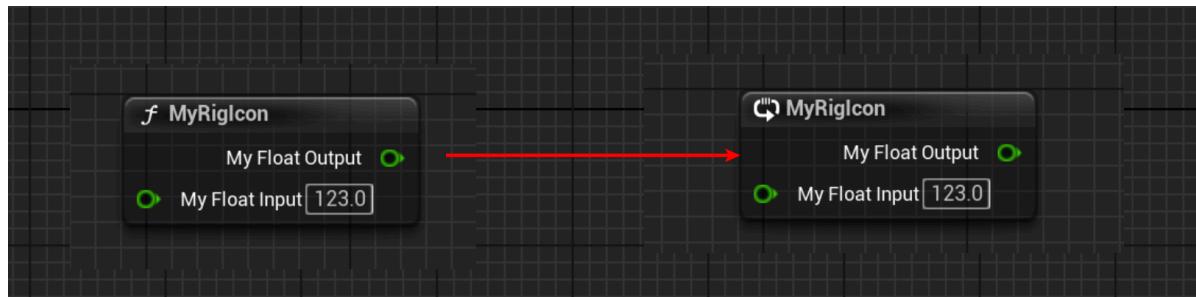
    UPROPERTY(meta = (Output))

```

```
float MyFloat_Output = 123.f;  
};
```

测试效果：

可见加了Icon之后，左上角图标变成了其他，不是默认的f函数目标。



原理：

```
FSlaterIcon URigVMEdGraphNode::GetIconAndTint(FLinearColor& OutColor) const  
{  
  
    if(MetadataScriptStruct && MetadataScriptStruct->HasMetaDataHierarchical(FRigVMStruct::IconMetaName))  
    {  
        FString IconPath;  
        const int32 NumOfIconPathNames = 4;  
  
        FName IconPathNames[NumOfIconPathNames] = {  
            NAME_None, // StyleSetName  
            NAME_None, // StyleName  
            NAME_None, // SmallStyleName  
            NAME_None // StatusOverlayStyleName  
        };  
  
        // icon path format:  
        styleSetName|styleName|smallStyleName|statusOverlayStyleName  
        // the last two names are optional, see FSlaterIcon() for reference  
        MetadataScriptStruct->GetStringMetaDataHierarchical(FRigVMStruct::IconMetaName, &IconPath);  
        return FSlaterIcon(IconPathNames[0], IconPathNames[1],  
                           IconPathNames[2], IconPathNames[3]);  
    }  
}
```

Input

- **功能描述:** 指定FRigUnit下的该属性作为输入引脚。
- **使用位置:** UPROPERTY
- **引擎模块:** RigVMStruct
- **元数据类型:** bool
- **限制类型:** FRigUnit中属性
- **关联项:** Output, Visible, Hidden, DetailsOnly, Constant

- 常用程度：★★★★★

指定FRigUnit下的该属性作为输入引脚。

值得注意的是，一个引脚如果同时加上Input和Output，那它就变成IO引脚，同时可作为输入和输出。

测试代码：

```
USTRUCT(meta = (DisplayName="MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
    virtual void Execute() override;

public:
    UPROPERTY()
    float MyFloat_Normal;

    UPROPERTY(meta = (Input))
    float MyFloat_Input;

    UPROPERTY(meta = (Output))
    float MyFloat_Output;

    UPROPERTY(meta = (Input, Output))
    float MyFloat_IO;

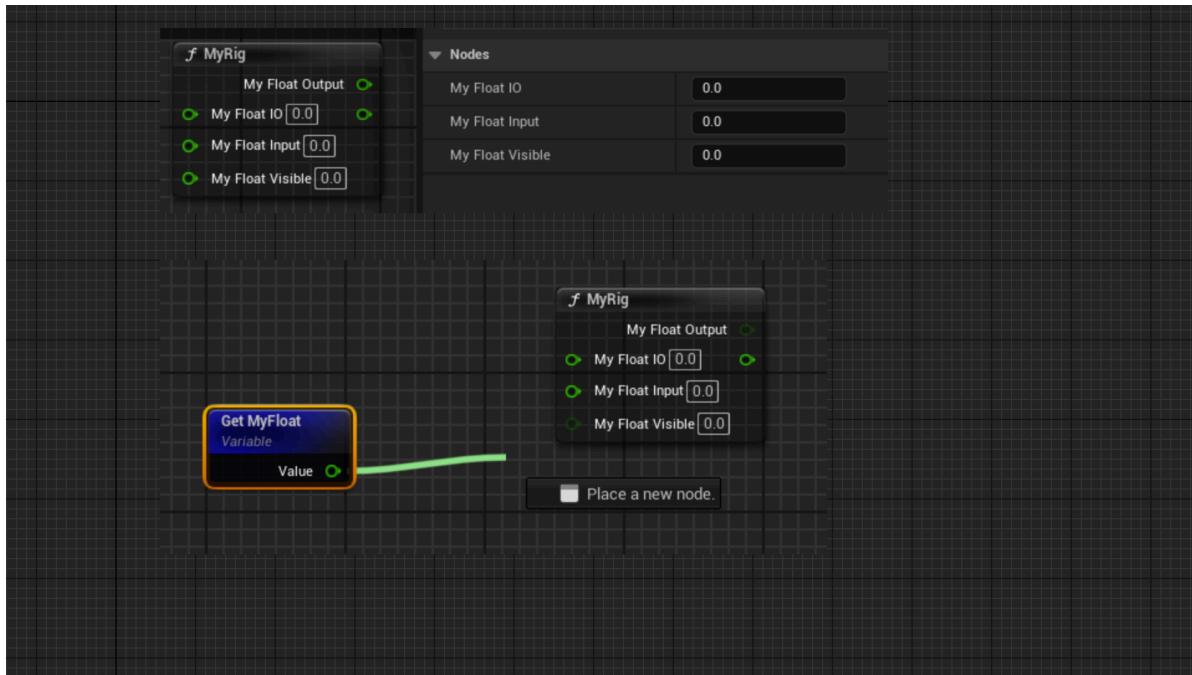
    UPROPERTY(meta = (Visible))
    float MyFloat_Visible;

    UPROPERTY(meta = (Hidden))
    float MyFloat_Hidden;
};
```

测试效果：

在ControlRig蓝图里就可以调用MyRig节点，注意观察属性在蓝图节点上的引脚表现以及在右侧细节面板的显示。

- MyFloat_Normal不标meta，在两个地方都没有显示。
- MyFloat_Input，作为输入引脚，且在右侧细节面板也显示。
- MyFloat_Output，作为输出引脚，右侧细节面板不显示。
- MyFloat_IO，可以同时作为输入和输出引脚，右侧细节面板会显示。
- MyFloat_Visible，可以作为输入引脚显示，右侧细节面板会显示。但是无法连接变量，意思是只能作为常量使用。
- MyFloat_Hidden，如同MyFloat_Normal一样，在蓝图节点和细节面板都隐藏起来，只是作为自己的内部值使用。



原理：

根据属性上的Meta标记来区分引脚的方向。可以在源码里查看ERigVMPinDirection 的各个类型。

```

UENUM(BlueprintType)
enum class ERigVMPinDirection : uint8
{
    Input, // A const input value
    Output, // A mutable output value
    IO, // A mutable input and output value
    Visible, // A const value that cannot be connected to
    Hidden, // A mutable hidden value (used for internal state)
    Invalid // The max value for this enum - used for guarding.
};

ERigVMPinDirection FRigVMStruct::GetPinDirectionFromProperty(FProperty* InProperty)
{
    bool bIsInput = InProperty->HasMetaData(InputMetaName);
    bool bIsOutput = InProperty->HasMetaData(OutputMetaName);
    bool bIsVisible = InProperty->HasMetaData(VisibleMetaName);

    if (bIsVisible)
    {
        return ERigVMPinDirection::Visible;
    }

    if (bIsInput)
    {
        return bIsOutput ? ERigVMPinDirection::IO : ERigVMPinDirection::Input;
    }

    if(bIsOutput)
    {
        return ERigVMPinDirection::Output;
    }
}

```

```
    return ERigVMPinDirection::Hidden;
}
```

Keywords

- **功能描述:** 设定FRigUnit蓝图节点在右键菜单中的关键字，方便输入查找。
- **使用位置:** USTRUCT
- **引擎模块:** RigVMStruct
- **元数据类型:** strings="a, b, c"
- **限制类型:** FRigUnit
- **常用程度:** ★★★

设定FRigUnit蓝图节点在右键菜单中的关键字，方便输入查找。

同Function上的Keywords有一样的作用。

测试代码：

```
USTRUCT(meta = (DisplayName = "MyRigKeywords", Keywords="MyKey,Otherword"))
struct INSIDER_API FRigUnit_MyRigKeywords: public FRigUnit
{
    GENERATED_BODY()

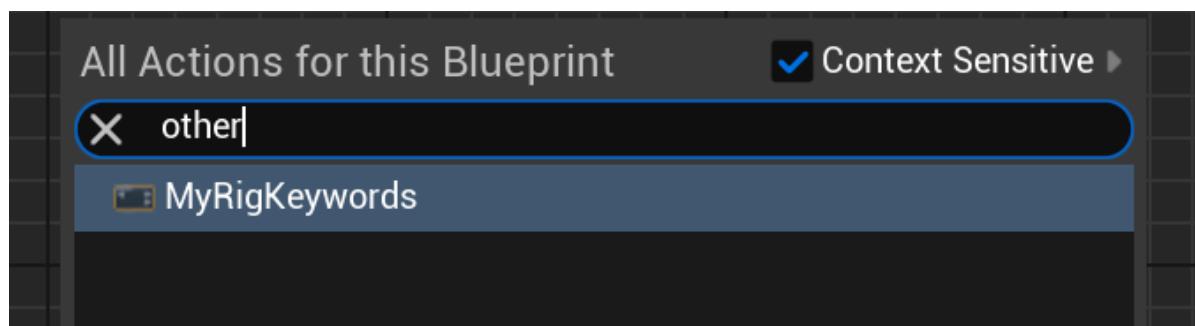
    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

测试效果：

在输入Keywords中的字符的时候，也可以找到该节点。



原理：

```

URigVMEdGraphUnitNodeSpawner*
URigVMEdGraphUnitNodeSpawner::CreateFromStruct(UScriptStruct* InStruct, const
FName& InMethodName, const FText& InMenuDesc, const FText& InCategory, const
FText& InTooltip)
{
    FString KeywordsMetadata, TemplateNameMetadata;
    InStruct->GetStringMetaDataHierarchical(FRigVMStruct::KeywordsMetaName,
    &KeywordsMetadata);
    if(!TemplateNameMetadata.IsEmpty())
    {
        if(KeywordsMetadata.IsEmpty())
        {
            KeywordsMetadata = TemplateNameMetadata;
        }
        else
        {
            KeywordsMetadata = KeywordsMetadata + TEXT(",") +
TemplateNameMetadata;
        }
    }
    MenuSignature.Keywords = FText::FromString(KeywordsMetadata);
}

```

MenuDescSuffix

- 功能描述:** 标识FRigUnit在蓝图右键菜单项的名字后缀。
- 使用位置:** USTRUCT
- 引擎模块:** RigVMStruct
- 元数据类型:** bool
- 限制类型:** FRigUnit类型上
- 常用程度:** ★★★

标识FRigUnit在蓝图右键菜单项的名字后缀。

测试代码:

```

USTRUCT(meta = (DisplayName = "MyRigSuffix", MenuDescSuffix = "(MyVector)"))
struct INSIDER_API FRigUnit_MyRigSuffix: public FRigUnit
{
    GENERATED_BODY()

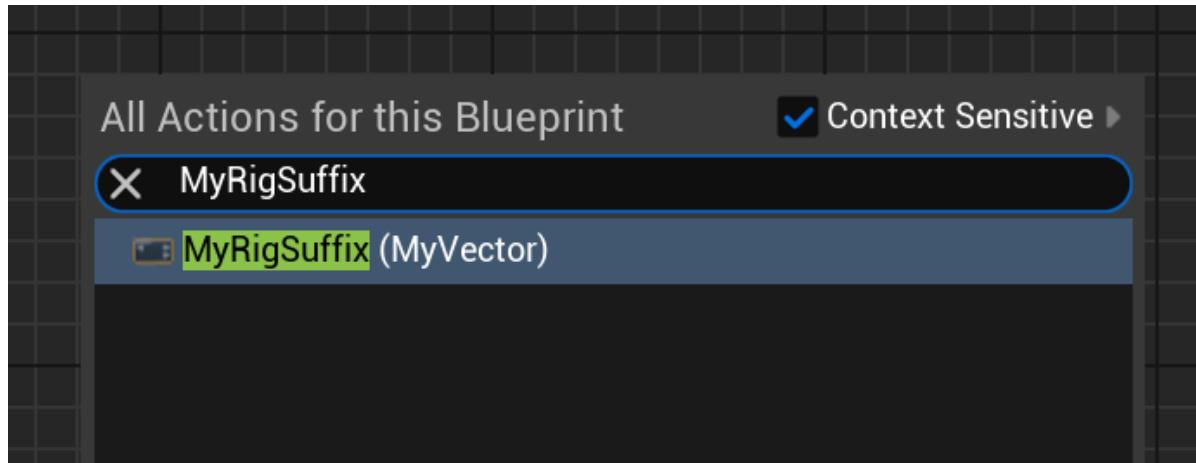
    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};

```

测试效果：

可见出现了"(MyVector)"的后缀。



原理：

得到该数据，然后添加到DisplayName后面。

```
FString CategoryMetadata, DisplayNameMetadata, MenuDescSuffixMetadata;
Struct->GetStringMetaDataHierarchical(FRigVMStruct::CategoryMetaName,
&CategoryMetadata);
Struct->GetStringMetaDataHierarchical(FRigVMStruct::DisplayNameMetaName,
&DisplayNameMetadata);
Struct->GetStringMetaDataHierarchical(FRigVMStruct::MenuDescSuffixMetaName,
&MenuDescSuffixMetadata);

if(DisplayNameMetadata.IsEmpty())
{
    DisplayNameMetadata = Struct->GetDisplayNameText().ToString();
}
if (!MenuDescSuffixMetadata.IsEmpty())
{
    MenuDescSuffixMetadata = TEXT(" ") + MenuDescSuffixMetadata;
}

FText MenuDesc = FText::FromString(DisplayNameMetadata + MenuDescSuffixMetadata);
```

NodeColor

- 功能描述：**指定FRigUnit蓝图节点的RGB颜色值。
- 使用位置：**USTRUCT
- 引擎模块：**RigVMStruct
- 元数据类型：**string="abc"
- 限制类型：**FRigUnit
- 常用程度：**★★

指定FRigUnit蓝图节点的RGB颜色值。

测试代码：

```
USTRUCT(meta = (DisplayName = "MyRigColor", NodeColor="1 0 0"))
struct INSIDER_API FRigUnit_MyRigColor : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

测试效果：

加上NodeColor之后，颜色从左变成右。



原理：

从Meta中获取颜色值。

```
FLinearColor FRigVMDispatchFactory::GetNodeColor() const
{
    if(const UScriptStruct* ScriptStruct = GetScriptStruct())
    {
        FString NodeColor;
        if (ScriptStruct->GetStringMetaDataHierarchical(FRigVMStruct::NodeColorMetaName, &NodeColor))
        {
            return FRigVMTemplate::GetColorFromMetadata(NodeColor);
        }
    }
    return FLinearColor::white;
}
```

Output

- **功能描述：**指定FRigUnit下的该属性作为输出引脚。
- **使用位置：** UPROPERTY
- **引擎模块：** RigVMStruct

- 元数据类型: bool
- 限制类型: FRigUnit中属性
- 关联项: Input
- 常用程度: ★★★★☆

指定FRigUnit下的该属性作为输出引脚。

RigVMTypAllowed

- 功能描述: 指定一个UENUM可以在FRigUnit的UEnum*属性中被选择。
- 使用位置: UENUM
- 引擎模块: RigVMStruct
- 元数据类型: bool
- 常用程度: ★★

指定一个UENUM可以在FRigUnit的UEnum*属性中被选择。

测试代码:

```
UENUM(BlueprintType)
enum class ERigMyEnum : uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType, meta = (RigVMTypAllowed))
enum class ERigMyEnumAllowed : uint8
{
    Cat,
    Dog,
    Tiger,
};

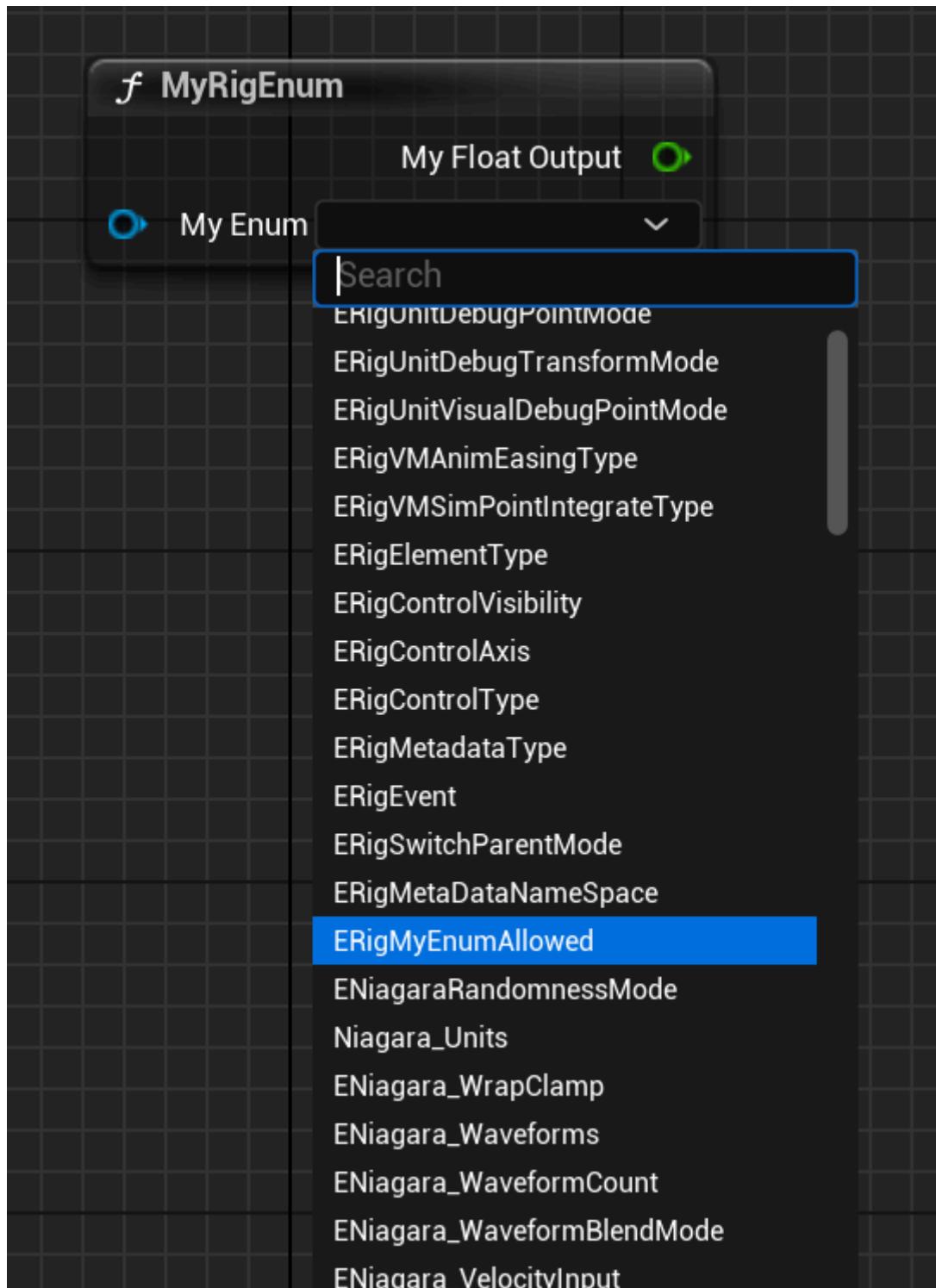
USTRUCT(meta = (DisplayName = "MyRigEnum"))
struct INSIDER_API FRigUnit_MyRigEnum : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
    virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    UEnum* MyEnum;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

测试效果：

可见在选项列表中只有ERigMyEnumAllowed，没有ERigMyEnum。



原理：

在生成选项的时候，判断 !Enum->IsAsset()说明是C++里的枚举，然后必须有RigVMTypeAllowed。

```
void SRigVMEnumPicker::PopulateEnumOptions()
{
    EnumOptions.Reset();
    EnumOptions.Add(MakeShareable(new FString(TEXT("None"))));
    for (TObjectIterator<UEnum> EnumIt; EnumIt; ++EnumIt)
    {
        if (!EnumIt->IsAsset())
        {
            if (EnumIt->RigVMTypeAllowed())
            {
                EnumOptions.Add(MakeShareable(new FString(EnumIt->GetName())));
            }
        }
    }
}
```

```

UEnum* Enum = *EnumIt;

if (Enum->HasAnyFlags(RF_BeginDestroyed | RF_FinishDestroyed) || !Enum->HasAllFlags(RF_Public))
{
    continue;
}

// Any asset based enum is valid
if (!Enum->IsAsset())
{
    // Native enums only allowed if contain RigVMTypAllowed metadata
    if (!Enum->HasMetaData(TEXT("RigVMTypAllowed")))
    {
        continue;
    }
}

EnumOptions.Add(MakeShareable(new FString(Enum->GetPathName())));
}
}

```

TemplateName

- 功能描述:** 指定该FRigUnit成为一个泛型模板节点。
- 使用位置:** USTRUCT
- 引擎模块:** RigVMStruct
- 元数据类型:** string="abc"
- 限制类型:** FRigUnit
- 常用程度:** ★★★

指定该FRigUnit成为一个泛型模板节点。

不同的FRigUnit在设置到同一个TemplateName之后，会分析其Input和Output的属性的整个函数签名，最后分析出哪些属性是泛型引脚（即同名不同类型的属性）。在调用的时候，输入的是TemplateNode，即TemplateName形成的节点。然后再手动连接引脚来确定最后的函数类型，从而最后再完全确定应该实际应用到哪一个FRigUnit节点。

这个功能在实现一些逻辑相同，但是参数类型稍微不同的时候，会比较便利。往往FRigUnit_MyTemplate_Float和FRigUnit_MyTemplate_Int会继承于一个基类（但不是强制），在里面实现公用的逻辑或属性。

测试代码：

```

USTRUCT(meta = (DisplayName = "Set My float", TemplateName = "SetMyTemplate"))
struct INSIDER_API FRigUnit_MyTemplate_Float : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:

```

```

UPROPERTY(meta = (Input))
float MyValue;

UPROPERTY(meta = (Output))
FString MyStringResult;
};

USTRUCT(meta = (DisplayName = "Set My int", TemplateName = "SetMyTemplate"))
struct INSIDER_API FRigUnit_MyTemplate_Int : public FRigUnit
{
GENERATED_BODY()

RIGVM_METHOD()
    virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    int32 MyValue;

    UPROPERTY(meta = (Output))
    FString MyStringResult;
};

```

测试效果：

可见一开始的节点是SetMyTemplate，然后根据引脚类型的不同，再实际Resolve成FRigUnit_MyTemplate_Float 或者是FRigUnit_MyTemplate_Int。因为我没有实现SetMyString，所以 FString类型的是不能连接到引脚的。



原理：

源码里涉及到这一块的代码比较多。大致逻辑是FRigUnit在初始化的时候注册到FRigVMRegistry里，如果有TempalteName则创建一个FRigTemplate，之后蓝图右键创建的时候实际创建的是URigTemplateNameNode，然后再由FRigDispatch来分发到实际的最终节点。

```

void FRigVMRegistry::Register(const TCHAR* InName, FRigVMFunctionPtr
InFunctionPtr, UScriptStruct* InStruct, const TArray<FRigVMFunctionArgument>&
InArguments)
{
    FString TemplateMetadata;
    if (InStruct->GetStringMetaDataHierarchical(TemplateNameMetaName,
&TemplateMetadata))
    {
    }
}

```

Varying

- 功能描述：** ScriptStruct /Script/RigVM.RigVMFunction_GetDeltaTime
- 使用位置：** UCLASS
- 引擎模块：** RigVMStruct

- **元数据类型:** bool

- **常用程度:** 0

放在USTRUCT上的时候，发现用在IsDefinedAsVarying这种函数上，但是F5没有发现调用的地方。

Visible

- **功能描述:** 指定FRigUnit下的该属性为常量引脚，无法连接变量。

- **使用位置:** UPROPERTY

- **引擎模块:** RigVMStruct

- **元数据类型:** bool

- **限制类型:** FRigUnit中属性

- **关联项:** Input

- **常用程度:** ★★★

指定FRigUnit下的该属性为常量引脚，无法连接变量。

Visible和Input+Constant的效果是一致的。

测试代码：

```
USTRUCT(meta = (DisplayName = "MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;

    public:
        UPROPERTY()
        float MyFloat_Normal;

        UPROPERTY(meta = (Input))
        float MyFloat_Input;

        UPROPERTY(meta = (Output))
        float MyFloat_Output;

        UPROPERTY(meta = (Input, Output))
        float MyFloat_IO;

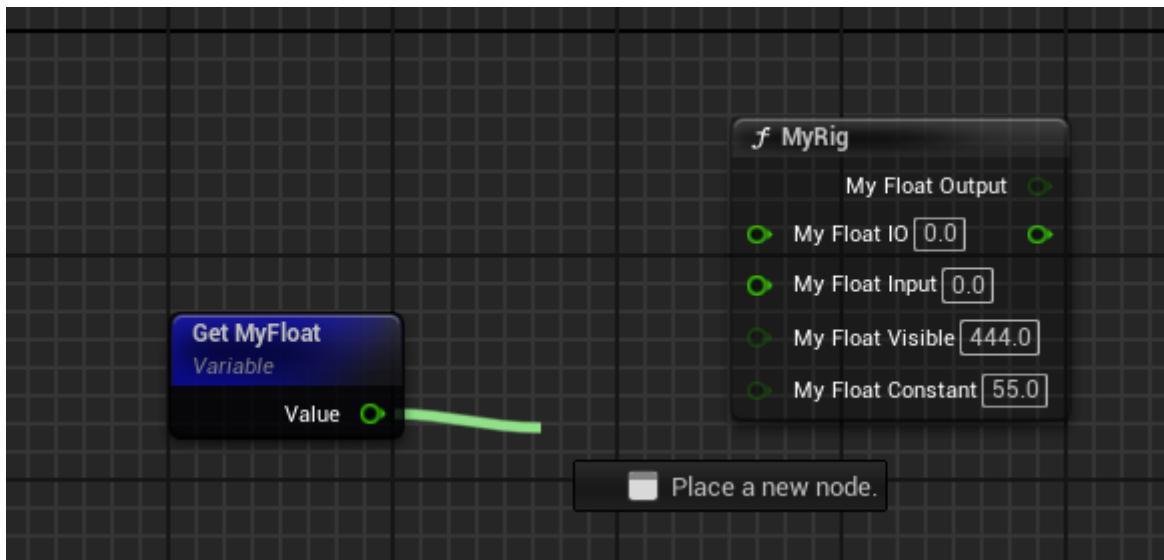
        UPROPERTY(meta = (Hidden))
        float MyFloat_Hidden;

        UPROPERTY(meta = (Visible))
        float MyFloat_Visible;

        UPROPERTY(meta = (Input, Constant))
        float MyFloat_Constant;
};
```

测试效果：

Visible和Input+Constant的效果是一致的，也是成为一个常量。



原理：

```
UENUM(BlueprintType)
enum class ERigVMPinDirection : uint8
{
    Input, // A const input value
    Output, // A mutable output value
    IO, // A mutable input and output value
    Visible, // A const value that cannot be connected to
    Hidden, // A mutable hidden value (used for internal state)
    Invalid // The max value for this enum - used for guarding.
};

FRigVMPinInfo::FRigVMPinInfo(FProperty* InProperty, ERigVMPinDirection InDirection, int32 InParentIndex, const uint8* InDefaultValueMemory)
{
    bIsConstant = InProperty->HasMetaData(TEXT("Constant"));
}

void URigVMController::ConfigurePinFromProperty(FProperty* InProperty, URigVMPin* InOutPin, ERigVMPinDirection InPinDirection) const
{
    InOutPin->bIsConstant = InProperty->HasMetaData(TEXT("Constant"));
}

bool URigVMPin::CanBeBoundToVariable(const FRigVMEExternalVariable& InExternalVariable, const FString& InSegmentPath) const
{
    if (bIsConstant)
    {
        return false;
    }
}
```

AllowedLocators

- **功能描述:** 用来给Sequencer定位可绑定的对象
- **使用位置:** UPROPERTY
- **引擎模块:** Scene
- **元数据类型:** string="abc"
- **限制类型:** FUniversalObjectLocator
- **常用程度:** ★

用来给Sequencer定位可绑定的对象。

看起来是Sequencer用来定位Actor做属性绑定的辅助定位器。只用在FUniversalObjectLocator 这个写好的属性里，一般我们用不到去扩展这部分，因此只是OnlyInternal。

源码中搜索得到：

```
// Helper struct for Binding Properties UI for locators.  
USTRUCT()  
struct FMovieSceneUniversalLocatorInfo  
{  
    GENERATED_BODY()  
  
    // Locator for the entry  
    UPROPERTY(EditAnywhere, Category = "Default", meta=(AllowedLocators="Actor"))  
    FUniversalObjectLocator Locator;  
  
    // Flags for how to resolve the locator  
    UPROPERTY()  
    ELocatorResolveFlags ResolveFlags = ELocatorResolveFlags::None;  
};
```

看起来是允许定位的对象类型。

原理：

```
TMap<FName, TSharedPtr<ILocatorEditor>> ApplicableLocators;  
  
void  
FUniversalObjectLocatorCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle>  
StructPropertyHandle, FDetailWidgetRow& HeaderRow,  
IPropertyTypeCustomizationUtils& StructCustomizationUtils)  
{  
    TArray< FString > AllowedTypes;  
    if (PropertyHandle->HasMetaData("AllowedLocators"))  
    {  
        PropertyHandle->  
        GetMetaData("AllowedLocators").ParseIntoArray(AllowedTypes, TEXT(","));  
    }  
}
```

```

FUniversalObjectLocatorEditorModule& Module =
FModuleManager::Get().LoadModuleChecked< FUniversalObjectLocatorEditorModule>
("UniversalObjectLocatorEditor");
for (TPair< FName, TSharedPtr< ILocatorEditor>> Pair : Module.LocatorEditors)
{
    if (AllowedTypes.Num() == 0 || AllowedTypes.Contains(Pair.Key.ToString()))
    {
        ApplicableLocators.Add(Pair.Key, Pair.Value);
    }
}

```

MakeEditWidget

- 功能描述:** 使FVector和FTransform在场景编辑器里出现一个可移动的控件。
- 使用位置:** UPROPERTY
- 引擎模块:** Scene
- 元数据类型:** bool
- 限制类型:** FVector, FTransform
- 关联项:** ValidateWidgetUsing
- 常用程度:** ★★★

使FVector和FTransform在场景编辑器里出现一个可移动的控件。

这样相比直接的数值编辑更加的直观一些。

测试代码：

```

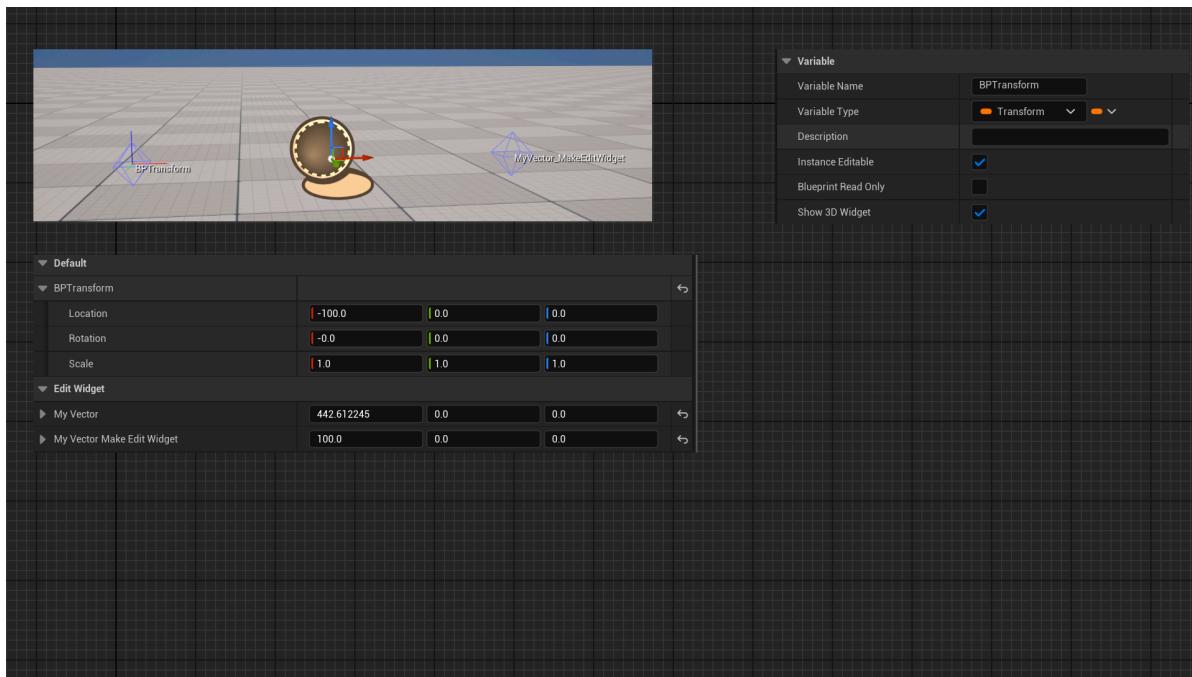
UCLASS(BlueprintType)
class INSIDER_API AMyActor_EditWidget :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Editwidget")
    FVector MyVector;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Editwidget", meta=(MakeEditWidget))
    FVector MyVector_MakeEditWidget;
};

```

测试结果：

在蓝图里继承的AMyActor_EditWidget 子类里加上另外一个FTransform变量，可以看见“Show 3D Widget”的选项，这个和MyVector_MakeEditWidget都在场景里出现了可移动的控件。



原理：

判断如果是FVector或FTransform，并且有MakeEditWidget属性，则可以创建控件。

```
/** Value of UPROPERTY can be edited with a widget in the editor (translation,
 * rotation) */
static UNREALED_API const FName MD_MakeEditWidget;
/** Specifies a function used for validation of the current value of a property.
 * The function returns a string that is empty if the value is valid, or contains an
 * error description if the value is invalid */
static UNREALED_API const FName MD_ValidatewidgetUsing;

bool FLegacyEdModeWidgetHelper::CanCreateWidgetForStructure(const UStruct*
InPropStruct)
{
    return InPropStruct && (InPropStruct->GetFName() == NAME_Vector || 
InPropStruct->GetFName() == NAME_Transform);
}

bool FLegacyEdModeWidgetHelper::ShouldCreateWidgetForProperty(FProperty* InProp)
{
    return CanCreateWidgetForProperty(InProp) && InProp-
>HasMetaData(MD_MakeEditWidget);
}
```

ValidateWidgetUsing

- 功能描述：** 提供一个函数来验证当前属性值是否合法。
- 使用位置：** UPROPERTY
- 引擎模块：** Scene
- 元数据类型：** bool
- 限制类型：** 带有MakeEditWidget的FVector, FTransform

- **关联项:** MakeEditWidget

- **常用程度:** ★★★

ValidateWidgetUsing提供一个函数来验证当前属性值是否合法。

- 当前属性要有MakeEditWidget的标记
- 函数的原型是FString MyFunc(), 返回非空表示错误信息。

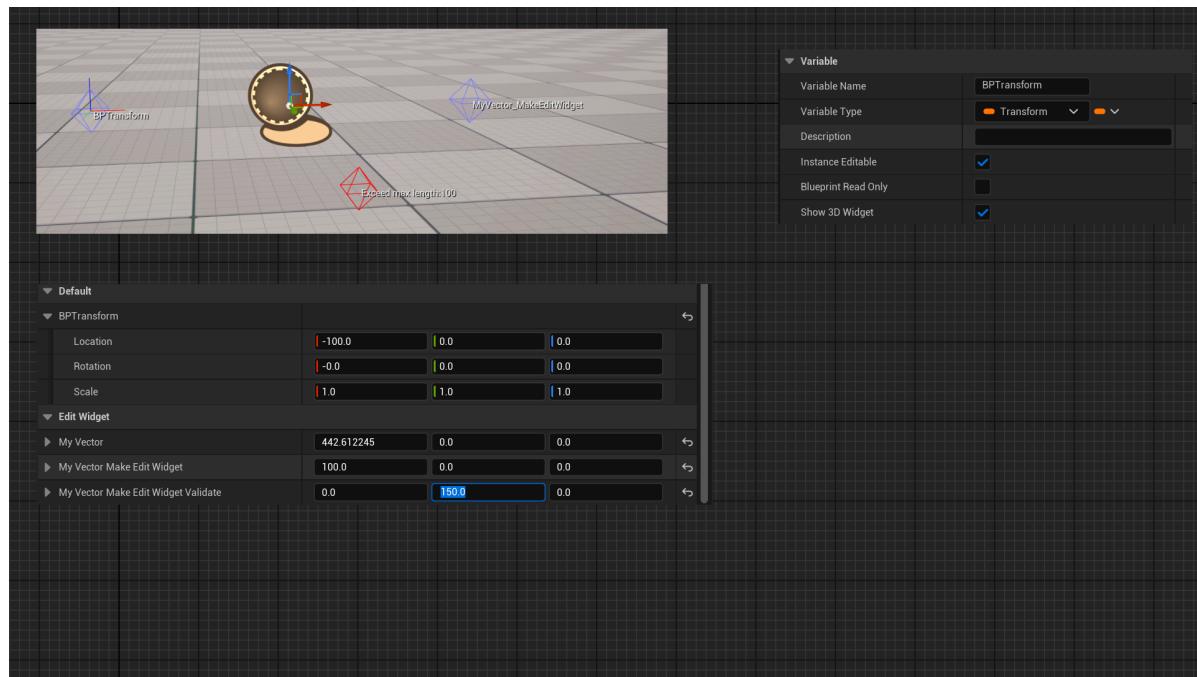
测试代码:

```
UFUNCTION()
FString validateMyvector()
{
    if (MyVector_MakeEditWidget_validate.Length()>100.f)
    {
        return TEXT("Exceed max length:100");
    }
    return TEXT("");
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Editwidget", meta =
(MakeEditwidget, ValidatewidgetUsing = "ValidateMyVector"))
FVector MyVector_MakeEditWidget_validate;
```

测试结果:

可见MyVector_MakeEditWidget_Validate长度超过100之后，控件颜色变成红色，并且显示出错误的信息在控件上。



原理:

逻辑比较简单。发现有验证函数，就调用验证函数来验证。如果有错误信息，就一起改变最终输出的颜色和显示文字。

```

        static FLegacyEdModeWidgetHelper::FPropertyWidgetInfo CreateWidgetInfo(const
TArray<FPropertyWidgetInfoChainElement>& Chain, bool bIsTransform, FProperty* CurrentProp, int32 Index = INDEX_NONE)
{
    check(CurrentProp);
    FEdMode::FPropertyWidgetInfo WidgetInfo;
    WidgetInfo.PropertyValidationName = FName(*CurrentProp->GetMetaData(FEdMode::MD_validateWidgetUsing));

    return WidgetInfo;
}

void FLegacyEdModeWidgetHelper::FPropertyWidgetInfo::GetTransformAndColor(UObject* BestSelectedItem, bool bisSelected, FTransform& OutLocalTransform, FString& OutValidationMessage, FColor& OutDrawColor) const
{
    // Determine the desired color
    if (PropertyValidationName != NAME_None)
    {
        if (UFunction* ValidateFunc = BestSelectedItem->FindFunction(PropertyValidationName))
        {
            BestSelectedItem->ProcessEvent(ValidateFunc, &OutValidationMessage);

            // If we have a negative result, the widget color is red.
            OutDrawColor = OutValidationMessage.IsEmpty() ? OutDrawColor : FColor::Red;
        }
    }
}

void FLegacyEdModeWidgetHelper::DrawHUD(FEditorViewportClient* ViewportClient,
FViewport* Viewport, const FSceneView* View, FCanvas* Canvas)
{
    FTransform LocalWidgetTransform;
    FString ValidationMessage;
    FColor WidgetColor;
    WidgetInfo.GetTransformAndColor(BestSelectedItem, bSelected, /*out*/ LocalWidgetTransform, /*out*/ ValidationMessage, /*out*/ WidgetColor);

    Canvas->DrawItem(TextItem);
}

```

ScriptConstant

- 功能描述:** 把一个静态函数的返回值包装成为一个常量值。
- 使用位置:** UFUNCTION
- 引擎模块:** Script
- 元数据类型:** string="abc"
- 关联项:** ScriptConstantHost
- 常用程度:** ★★★

把一个静态函数的返回值包装成为一个常量值。

- 函数的名字即为常量的默认名称，但ScriptConstant也可以额外提供一个自定义名称。
- 常量作用域默认存在于该静态函数的外部类中，但也可以通过ScriptConstantHost来指定到另外一个类型中。

测试代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonConstantStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ConstantOwner :public UObject
{
    GENERATED_BODY()
public:
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_Constant_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = (ScriptConstant))
    static int32 MyIntConst() { return 123; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant = "MyOtherIntConst"))
    static int32 MyIntConst2() { return 456; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant))
    static FMyPythonConstantStruct MyStructConst() { return
        FMyPythonConstantStruct{ TEXT("Hello") }; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant = "MyOtherStructConst"))
    static FMyPythonConstantStruct MyStructConst2() { return
        FMyPythonConstantStruct{ TEXT("World") }; }

public:
    UFUNCTION(BlueprintPure, meta = (ScriptConstant="FirstString",
        ScriptConstantHost = "/Script/Insider.MyPython_ConstantOwner"))
        static FString MyStringConst() { return TEXT("First"); }
    ****};
}
```

生成的Py代码：

```
class MyPython_Constant_Test(Object):
    r"""

```

```

My Python Constant Test

**C++ Source:**

- Module: Insider
- File: MyPython_ScriptConstant.h

"""

MY_OTHER_STRUCT_CONST: MyPythonConstantStruct #: (MyPythonConstantStruct): My
Struct Const 2
MY_STRUCT_CONST: MyPythonConstantStruct #: (MyPythonConstantStruct): My
Struct Const
MY_OTHER_INT_CONST: int #: (int32): My Int Const 2
MY_INT_CONST: int #: (int32): My Int Const

class MyPython_ConstantOwner(Object):
    r"""
    **My Python Constant Owner

**C++ Source:**

- Module: Insider
- File: MyPython_ScriptConstant.h

"""

FIRST_STRING: str #: (str): My String Const

```

运行的结果：

可见在类中生成了相应的常量。而MyStringConst因为指定了ScriptConstantHost 而生成在别的类中。

```

LogPython: print(unreal.MyPython_Constant_Test.MY_INT_CONST)
LogPython: 123
LogPython: print(unreal.MyPython_Constant_Test.MY_OTHER_INT_CONST)
LogPython: 456
LogPython: print(unreal.MyPython_Constant_Test.MY_OTHER_STRUCT_CONST)
LogPython: <Struct 'MyPythonConstantStruct' (0x00000A0FC4051F00) {my_string:
"world"}>
LogPython: print(unreal.MyPython_Constant_Test.MY_STRUCT_CONST)
LogPython: <Struct 'MyPythonConstantStruct' (0x00000A0FC4051EA0) {my_string:
"Hello"}>
LogPython: print(unreal.MyPython_ConstantOwner.FIRST_STRING)
LogPython: First

```

原理：

生成的逻辑在这个GenerateWrappedConstant 函数里。

```

auto GenerateWrappedConstant = [this, &GeneratedWrappedType,
&OutGeneratedWrappedTypeReferences, &OutDirtyModules](const UFunction* InFunc)
{ }

```

ScriptConstantHost

- **功能描述:** 在ScriptConstant的基础上，指定常量生成的所在类型。
- **使用位置:** UFUNCTION
- **引擎模块:** Script
- **元数据类型:** string="abc"
- **关联项:** ScriptConstant
- **常用程度:** ★

在ScriptConstant的基础上，指定常量生成的所在类型。

测试代码见ScriptConstant。ScriptConstantHost指定的字符串应该是个对象路径。

```
UFUNCTION(BlueprintPure, meta = (ScriptConstant="FirstString",
ScriptConstantHost = "/Script/Insider.MyPython_ConstantOwner"))
static FString MyStringConst() { return TEXT("First"); }
```

ScriptDefaultBreak

- **使用位置:** USTRUCT
- **引擎模块:** Script
- **元数据类型:** bool
- **关联项:** ScriptDefaultMake
- **常用程度:** ★

见ScriptDefaultMake的原理和测试代码。

ScriptDefaultMake

- **功能描述:** 禁用结构上的HasNativeMake，在脚本里构造的时候不调用C++里的NativeMake函数，而采用脚本内建的默认初始化方式。
- **使用位置:** USTRUCT
- **引擎模块:** Script
- **元数据类型:** bool
- **关联项:** ScriptDefaultBreak
- **常用程度:** ★

禁用结构上的HasNativeMake，在脚本里构造的时候不调用C++里的NativeMake函数，而采用脚本内建的默认初始化方式。

ScriptDefaultBreak也是同理。

测试代码：

```

USTRUCT(BlueprintType, meta = (ScriptDefaultMake,
ScriptDefaultBreak, HasNativeMake =
"/script/Insider.MyPython_MakeBreak_Test.MyNativeMake", HasNativeBreak =
"/script/Insider.MyPython_MakeBreak_Test.MyNativeBreak"))
struct INSIDER_API FMyPythonMBStructNative
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 0;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_MakeBreak_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = ())
    static FMyPythonMBStructNative MyNativeMake(int32 InInt) { return
FMyPythonMBStructNative{ InInt, TEXT("Hello") }; }

    UFUNCTION(BlueprintPure, meta = ())
    static void MyNativeBreak(const FMyPythonMBStructNative& InStruct, int&
outInt) { outInt = InStruct.MyInt + 123; }
};

```

生成的py代码：

无论有没有加ScriptDefaultMake, ScriptDefaultBreak, MyPythonMBStructNative生成的py代码其实是一样的。不同点在于构成和to_tuple时候的结果不同。

```

class MyPythonMBStructNative(StructBase):
    r"""
    My Python MBStruct Native

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMakeBreak.h

    **Editor Properties:** (see get_editor_property/set_editor_property)

    - ``my_int`` (int32): [Read-write]
    - ``my_string`` (str): [Read-write]
    """

    def __init__(self, int: int = 0) -> None:
        ...
    @property
    def my_int(self) -> int:
        r"""
        (int32): [Read-write]

```

```

    ...
    @my_int.setter
    def my_int(self, value: int) -> None:
        ...

    @property
    def my_string(self) -> str:
        r"""
        (str): [Read-write]
        """

    ...
    @my_string.setter
    def my_string(self, value: str) -> None:
        ...

```

运行的结果：

- 第二段是加了ScriptDefaultMake, ScriptDefaultBreak后的效果。我故意在C++的Make和Break函数里做了一些不一样，可以观察到调用到C++里的函数。
- 第一段是在代码里加上ScriptDefaultMake, ScriptDefaultBreak后（保留HasNativeMake, HasNativeBreak）调用的结果，可见C++里的Make/Break函数就没有再被调用到了。

```

LogPython: b=unreal.MyPythonMBStructNative()
LogPython: print(b)
LogPython: <Struct 'MyPythonMBStructNative' (0x0000085F2EE9E680) {my_int: 0,
my_string: "Hello"}>
LogPython: print(b.to_tuple())
LogPython: (123,)

LogPython: b=unreal.MyPythonMBStructNative()
LogPython: print(b)
LogPython: <Struct 'MyPythonMBStructNative' (0x000005E6C3AAFDC0) {my_int: 0,
my_string: ""}>
LogPython: print(b.to_tuple())
LogPython: (0, '')

```

原理：

在FindMakeBreakFunction函数里，如果发现有ScriptDefaultMake或ScriptDefaultBreak标记，就不去使用C++里由HasNativeMake, HasNativeBreak指定的函数。

另外py里的结构初始化会调用到默认的init或者结构的make函数，而to_tuple就相当于break的作用，会调用到默认的每个属性to_tuple或者是结构的自定义break函数。

```

const FName ScriptDefaultMakeMetaDataKey = TEXT("ScriptDefaultMake");
const FName ScriptDefaultBreakMetaDataKey = TEXT("ScriptDefaultBreak");

namespace UE::Python
{
    /**
     * Finds the UFunction corresponding to the name specified by 'HasNativeMake' or
     'HasNativeBreak' meta data key.

```

```

    * @param The structure to inspect for the 'HasNativeMake' or 'HasNativeBreak'
meta data keys.
    * @param InNativeMetaDataKey The native meta data key name. Can only be
'HasNativeMake' or 'HasNativeBreak'.
    * @param InScriptDefaultMetaDataKey The script default meta data key name. Can
only be 'ScriptDefaultMake' or 'ScriptDefaultBreak'.
    * @param NotFoundFn Function invoked if the structure specifies as Make or Break
function, but the function couldn't be found.
    * @return The function, if the struct has the meta key and if the function was
found. Null otherwise.
*/
template<typename NotFoundFuncT>
UFunction* FindMakeBreakFunction(const UScriptStruct* InStruct, const FName&
InNativeMetaDataKey, const FName& InScriptDefaultMetaDataKey, const
NotFoundFuncT& NotFoundFn)
{
    check(InNativeMetaDataKey == PyGenUtil::HasNativeMakeMetaDataKey ||
InNativeMetaDataKey == PyGenUtil::HasNativeBreakMetaDataKey);
    check(InScriptDefaultMetaDataKey == PyGenUtil::ScriptDefaultMakeMetaDataKey
|| InScriptDefaultMetaDataKey == PyGenUtil::ScriptDefaultBreakMetaDataKey);

    UFunction* MakeBreakFunc = nullptr;
    if (!InStruct->HasMetaData(InScriptDefaultMetaDataKey)) // <-- 有了default,
会直接返回null
    {
        const FString MakeBreakFunctionName = InStruct-
>GetMetaData(InNativeMetaDataKey);
        if (!MakeBreakFunctionName.IsEmpty())
        {
            // Find the function.
            MakeBreakFunc = FindObject<UFunction>(*Outer*/nullptr,
*MakeBreakFunctionName, /*ExactClass*/true);
            if (!MakeBreakFunc)
            {
                NotFoundFn(MakeBreakFunctionName);
            }
        }
    }
    return MakeBreakFunc;
}

struct FFuncs
{
    static int Init(FPyWrapperStruct* InSelf, PyObject* InArgs, PyObject* InKwds)
    {
        const int SuperResult = PyWrapperStructType.tp_init((PyObject*)InSelf,
InArgs, InKwds);
        if (SuperResult != 0)
        {
            return SuperResult;
        }

        return FPyWrapperStruct::MakeStruct(InSelf, InArgs, InKwds);
    }
};

```

```

GeneratedwrappedType->PyType.tp_init = (initproc)&FFuncs::Init;

// python wrapper 给每个类型都映射了 to_tuple 函数，会调用类型的 break 函数转换为 tuple
static PyObject* ToTuple(FPyWrapperStruct* InSelf)
{
    return FPyWrapperStruct::BreakStruct(InSelf);
}

.....
{ "to_tuple", PyCFunctionCast(&FMethods::ToTuple), METH_NOARGS,
"to_tuple(self) -> Tuple[object, ...] -- break this Unreal struct into a tuple of
its properties" },

```

ScriptMethod

- 功能描述：**把静态函数导出变成第一个参数的成员函数。
- 使用位置：** UFUNCTION
- 引擎模块：** Script
- 元数据类型：** string="a;b;c"
- 限制类型：** static function
- 关联项：** ScriptMethodMutable, ScriptMethodSelfReturn
- 常用程度：** ★★★

把静态函数导出变成第一个参数的成员函数。

- 把func(A, B)变成A.func(B)，这样就可以给A对象添加成员函数方法。有点像C#里的扩展方法。
- 也可以直接再提供一个名字来改变包装后的成员函数的名称。注意与ScriptName区分，
ScriptName改变的是本身导出到脚本的名字，而ScriptMethod改变的是结果成员函数的名字。把
func(A, B)改成A.OtherFunc(B)。

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod :public UObject
{
    GENERATED_BODY()
public:
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonStruct_ScriptMethod
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod_Test :public UObject
{
    GENERATED_BODY()
public:
};

```

```

UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
static void MyFuncOnobject(UMyPython_ScriptMethod* obj, FString val);

UFUNCTION(BlueprintCallable, meta = (ScriptMethod =
"MySuperFunOnObject;MyOtherFunOnObject"))
static void MyFuncOnobject2(UMyPython_ScriptMethod* obj, FString val);

public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static void MyFuncOnStruct(const FMyPythonStruct_ScriptMethod& myStruct,
    FString val);;
};

```

测试效果：

可见在MyPythonStruct_ScriptMethod里增加了my_func_on_struct的方法，而MyPython_ScriptMethod里增加了my_func_on_object的方法。因此如果在py里你就可以把这两个函数当作成员函数一样调用。

另外MyFuncOnObject2上面设置了两个ScriptMethod 别称，也可以在MyPython_ScriptMethod里见到。

```

class MyPythonStruct_ScriptMethod(StructBase):
    r"""
    My Python Struct Script Method

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    ...
    def __init__(self) -> None:
        ...
    def my_func_on_struct(self, val: str) -> None:
        r"""
        x.my_func_on_struct(val) -> None
        My Func on Struct

    Args:
        val (str):
    ...

class MyPython_ScriptMethod(Object):
    r"""
    My Python Script Method

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    ...

```

```
def my_super_func_on_object(self, val: str) -> None:
    r"""
        x.my_super_func_on_object(val) -> None
        My Func on Object 2

    Args:
        val (str):
            """
    ...

def my_other_func_on_object(self, val: str) -> None:
    r"""
        deprecated: 'my_other_func_on_object' was renamed to
        'my_super_func_on_object'.
    """

    ...

def my_func_on_object(self, val: str) -> None:
    r"""
        x.my_func_on_object(val) -> None
        My Func on Object

    Args:
        val (str):
            """
    ...

class MyPython_ScriptMethod_Test(Object):
    r"""
        My Python Script Method Test

    **C++ Source:**
    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    """
    @classmethod
    def my_func_on_struct(cls, my_struct: MyPythonStruct_ScriptMethod, val: str)
    -> None:
        r"""
            X.my_func_on_struct(my_struct, val) -> None
            My Func on Struct

        Args:
            my_struct (MyPythonStruct_ScriptMethod):
                val (str):
                    """
    ...

    @classmethod
    def my_func_on_object2(cls, obj: MyPython_ScriptMethod, val: str) -> None:
        r"""
            X.my_func_on_object2(obj, val) -> None
            My Func on Object 2

        Args:
            obj (MyPython_ScriptMethod):
                val (str):
                    """
```

```

...
@classmethod
def my_func_on_object(cls, obj: MyPython_ScriptMethod, val: str) -> None:
    r"""
    X.my_func_on_object(obj, val) -> None
    My Func on Object

    Args:
        obj (MyPython_ScriptMethod):
        val (str):
    """
    ...

```

原理：

在GenerateWrappedDynamicMethod中有详细的如何把静态函数包装成成员函数的过程。感兴趣的可以去细看。

```

PyTypeObject* FPywrapperTypeRegistry::GeneratewrappedClassType(const UClass*
InClass, FGeneratedwrappedTypeReferences& OutGeneratedwrappedTypeReferences,
TSet<FName>& OutDirtyModules, const EPyTypeGenerationFlags InGenerationFlags)
{
    // Should this function also be hoisted as a struct method or operator?
    if (InFunc->HasMetaData(PyGenUtil::ScriptMethodMetaDataKey))
    {
        GeneratewrappedDynamicMethod(InFunc, GeneratedwrappedMethodCopy);
    }
}

```

ScriptMethodMutable

- 功能描述：**把ScriptMethod的第一个const结构参数在调用上改成引用参数，函数内修改的值会保存下来。
- 使用位置：** UFUNCTION
- 引擎模块：** Script
- 元数据类型：** bool
- 限制类型：** 第一个参数是结构类型
- 关联项：** ScriptMethod
- 常用程度：** ★★

把ScriptMethod的第一个const结构参数在调用上改成引用参数，函数内修改的值会保存下来。

- 在const参数上如果想改变值，依然要标记c++里的mutable。
- 虽然py生成的代码一模一样，但实际调用上ScriptMethodMutable会真正改变参数的值，而没有ScriptMethodMutable的函数并不会改变参数的原始值。
- ScriptMethodMutable和UPARAM(ref) 在调用效果上，都可以改变参数的值。但区别是UPARAM(ref)生成的py代码会返回第一个参数作为返回值。

```

USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonStruct_ScriptMethod

```

```

{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    mutable FString MyString;

};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static void SetStringOnStruct(const FMyPythonStruct_ScriptMethod& myStruct,
FString val);

    UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))
    static void SetStringOnStructMutable(const FMyPythonStruct_ScriptMethod&
myStruct, FString val);

    UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))
    static void SetStringOnStructViaRef(UPARAM(ref) FMyPythonStruct_ScriptMethod&
myStruct, FString val);
};

```

测试效果：

看py里生成的代码是一致的，如果用UPARAM(ref)，则在MyPython_ScriptMethod_Test里面生成的my_func_on_struct_via_ref会返回结构MyPythonStruct_ScriptMethod来达成引用的效果。

然而my_func_on_struct Mutable返回的是None，同不加ScriptMethodMutable的my_func_on_struct并没有区别。但是实际上在真正调用的时候会真正有区别。

```

class MyPythonStruct_ScriptMethod(StructBase):
    r"""
    My Python Struct Script Method

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    """

    def __init__(self) -> None:
        ...

    def my_func_on_struct_via_ref(self, val: str) -> None:
        r"""
        x.my_func_on_struct_via_ref(val) -> None
        My Func on Struct Via Ref

        Args:
            val (str):
        """

        ...

```

```

def my_func_on_struct mutable(self, val: str) -> None:
    r"""
    x.my_func_on_struct mutable(val) -> None
    My Func on Struct Mutable

    Args:
        val (str):
            ...
            ...
            ...

def my_func_on_struct(self, val: str) -> None:
    r"""
    x.my_func_on_struct(val) -> None
    My Func on Struct

    Args:
        val (str):
            ...
            ...
            ...

class MyPython_ScriptMethod_Test(Object):
    r"""
    My Python Script Method Test

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    ...
    @classmethod
    def my_func_on_struct_via_ref(cls, my_struct: MyPythonStruct_ScriptMethod,
val: str) -> MyPythonStruct_ScriptMethod:
        r"""
        X.my_func_on_struct_via_ref(my_struct, val) ->
MyPythonStruct_ScriptMethod
        My Func on Struct Via Ref

    Args:
        my_struct (MyPythonStruct_ScriptMethod):
            val (str):
                ...

    Returns:
        MyPythonStruct_ScriptMethod:

        my_struct (MyPythonStruct_ScriptMethod):
            ...
            ...
            ...

    @classmethod
    def my_func_on_struct mutable(cls, my_struct: MyPythonStruct_ScriptMethod,
val: str) -> None:
        r"""
        X.my_func_on_struct mutable(my_struct, val) -> None
        My Func on Struct Mutable

    Args:

```

```

my_struct (MyPythonStruct_ScriptMethod):
    val (str):
        ...
    ...

@classmethod
def my_func_on_struct(cls, my_struct: MyPythonStruct_ScriptMethod, val: str)
-> None:
    r"""
    X.my_func_on_struct(my_struct, val) -> None
    My Func on Struct

Args:
    my_struct (MyPythonStruct_ScriptMethod):
        val (str):
            ...
        ...
    ...

```

在UE Python控制台里调用的记录，分析调用顺序：

- 一开始调用set_string_on_struct mutable，再print(a)，可以打印出Hello，说明值真正的设置到了a结构里。
- 再尝试set_string_on_struct，再print(a)，无法打印出FFF，说明值并没有设置到a结构里。说明py在调用的时候很可能构造了一个临时值来当作调对象，调用完成的新值并没有设置到a对象上。
- 再尝试set_string_on_struct_via_ref，再print(a)，可以打印出First，说明用UPARAM(ref)可以也达成改变参数的效果。

```

LogPython: a=unreal.MyPythonStruct_ScriptMethod()
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
""}>
LogPython: a.set_string_on_struct mutable("Hello")
LogBlueprintUserMessages: [None]
UMyPython_ScriptMethod_Test::SetStringOnStructMutable
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"Hello"}>
LogPython: a.set_string_on_struct("FFF")
LogBlueprintUserMessages: [None] UMyPython_ScriptMethod_Test::SetStringOnStruct
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"Hello"}>
LogPython: a.set_string_on_struct_via_ref("First")
LogBlueprintUserMessages: [None]
UMyPython_ScriptMethod_Test::SetStringOnStructViaRef
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"First"}>

```

原理：

判断如果有ScriptMethodMutable，会设置SelfReturn，从而再最后把函数调用中的临时值复制给原本的参数值，达成可变引用调用的效果。

```

// The function may have been flagged as mutable, in which case we always
// consider it to need a 'self' return
if (!GeneratedwrappedDynamicMethod.SelfReturn.ParamProp && InFunc-
>HasMetaData(PyGenUtil::ScriptMethodMutableMetaDataKey))
{
    if (!SelfParam.ParamProp->IsA<FStructProperty>())
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodMutable' but the 'self' argument is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    GeneratedwrappedDynamicMethod.SelfReturn = SelfParam;
}

```

ScriptMethodSelfReturn

- 功能描述:** 在ScriptMethod的情况下，指定把这个函数的返回值要去覆盖该函数的第一个参数。
- 使用位置:** UFUNCTION
- 引擎模块:** Script
- 元数据类型:** bool
- 关联项:** ScriptMethod
- 常用程度:** ★★

在ScriptMethod的情况下，指定把这个函数的返回值要去覆盖该函数的第一个参数。

这种情况下，原本的函数就没有返回值返回了。效果上形如：

```

C Func(A,B) -> void A::Func2(B)
调用的时候：
从 C=A.Func(B) ->
void A::Func2(B)
{
    A=A.Func(B)
}

```

测试代码：

注意因为AppendStringOnStructViaRef参数是引用参数，所以为了结果应用到myStruct，在函数体内就不需要创建临时值，可以直接在myStruct上面修改。如果也用临时值的话，myStruct就无法得到修改，也就失去了ref参数的意义。

```

public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static FMyPythonStruct_ScriptMethod AppendStringOnStruct(const
FMyPythonStruct_ScriptMethod& myStruct, FString val)
    {
        FMyPythonStruct_ScriptMethod Result = myStruct;
        Result.MyString += val;
        return Result;
    }

```

```

UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodSelfReturn))
static FMyPythonStruct_ScriptMethod AppendStringOnStructReturn(const
FMyPythonStruct_ScriptMethod& myStruct, FString val)
{
    FMyPythonStruct_ScriptMethod Result = myStruct;
    Result.MyString += val;
    return Result;
}
UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))
static FMyPythonStruct_ScriptMethod AppendStringOnStructViaRef(UPARAM(ref)
FMyPythonStruct_ScriptMethod& myStruct, FString val)
{
    myStruct.MyString += val;
    return myStruct;
}

//LogPython: Error: Function
'MyPython_ScriptMethod_Test.AppendStringOnStructViaRefReturn' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is also marked as UPARAM(ref).
This is not allowed.
//UFUNCTION(BlueprintCallable, meta = (ScriptMethod,
ScriptMethodMutable, ScriptMethodSelfReturn))
//static FMyPythonStruct_ScriptMethod
AppendStringOnStructViaRefReturn(UPARAM(ref) FMyPythonStruct_ScriptMethod&
myStruct, FString val);

```

生成的py代码：

可见append_string_on_struct_return是没有返回值了。而append_string_on_struct有返回值。
append_string_on_struct_via_ref也有返回值。

```

class MyPythonStruct_ScriptMethod(StructBase):
    def append_string_on_struct_return(self, val: str) -> None:
        r"""
        x.append_string_on_struct_return(val) -> None
        Append String on Struct Return

        Args:
            val (str):

        Returns:
            MyPythonStruct_ScriptMethod:
        """

        ...
    def append_string_on_struct(self, val: str) -> MyPythonStruct_ScriptMethod:
        r"""
        x.append_string_on_struct(val) -> MyPythonStruct_ScriptMethod
        Append String on Struct

        Args:
            val (str):

        Returns:
            MyPythonStruct_ScriptMethod:
        """

```

```

    """
    def append_string_on_struct_via_ref(self, val: str) ->
MyPythonStruct_ScriptMethod:
    r"""
        x.append_string_on_struct_via_ref(val) -> MyPythonStruct_ScriptMethod
        Append String on Struct Via Ref

    Args:
        val (str):

    Returns:
        MyPythonStruct_ScriptMethod:
    """
    ...

```

测试代码：观察运行的结果以及对象的内存地址。

- 可以看出append_string_on_struct是有返回值的，但是改变的结果没有应用到参数a上。
- append_string_on_struct_return可以应用到参数a上，但是没有返回值。
- append_string_on_struct_via_ref可以应用到参数a上，同时也有返回值。但是注意返回值和a其实并不是同一个对象，因为内存地址不同。
- 但是注意 ScriptMethodSelfReturn和UPARAM(ref)不能混用，否则会报错： LogPython: Error: Function 'MyPython_ScriptMethod_Test.AppendStringOnStructViaRefReturn' is marked as 'ScriptMethodSelfReturn' but the 'self' argument is also marked as UPARAM(ref). This is not allowed.

```

LogPython: a=unreal.MyPythonStruct_ScriptMethod()
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB08ED0) {my_string:
""}>
LogPython: b=a.append_string_on_struct("Hello")
LogPython: print(b)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB04010) {my_string:
"Hello"}>
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB08ED0) {my_string:
""}>
LogPython: c=a.append_string_on_struct_return("Hello")
LogPython: print(c)
LogPython: None
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB08ED0) {my_string:
"Hello"}>
LogPython: d=a.append_string_on_struct_via_ref("world")
LogPython: print(d)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB06110) {my_string:
"HelloWorld"}>
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEB08ED0) {my_string:
"HelloWorld"}>

```

原理：

把输出参数的第一个当作返回参数。输出参数其实就是函数里的返回值。SelfReturn的意思是这个值之后要去覆盖掉调用对象的值，也就是发生调用的对象。

```
// The function may also have been flagged as having a 'self' return
if (InFunc->HasMetaData(PyGenUtil::ScriptMethodSelfReturnMetaDataKey))
{
    if (GeneratedwrappedDynamicMethod.SelfReturn.ParamProp)
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is also marked as UPARAM(ref).
This is not allowed."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    else if (GeneratedwrappedDynamicMethod.MethodFunc.OutputParams.Num() == 0 ||
GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp-
>HasAnyPropertyFlags(CPF_ReturnParm))
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but has no return value."), *InFunc->GetOwnerClass()-
>GetName(), *InFunc->GetName());
        return;
    }
    else if (!SelfParam.ParamProp->IsA<FStructProperty>())
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    else if (!GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp-
>IsA<FStructProperty>())
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the return value is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    else if (CastFieldChecked<const FStructProperty>
(GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp)->Struct !=
CastFieldChecked<const FStructProperty>(SelfParam.ParamProp)->Struct)
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the return value is not the same type as the 'self'
argument."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    else
    {
        GeneratedwrappedDynamicMethod.SelfReturn =
MoveTemp(GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0]);
        GeneratedwrappedDynamicMethod.MethodFunc.OutputParams.RemoveAt(0, 1,
EAllowShrinking::No);
    }
}
```

```
}
```

ScriptName

- **功能描述:** 在导出到脚本里时使用的名称
- **使用位置:** Any
- **引擎模块:** Script
- **元数据类型:** string="abc"
- **常用程度:** ★★★

指定导出到脚本中的名字。

- 可以使用在UCLASS, USTRUCT, UENUM, UFUNCTION, UPROPERTY上使用，改变其导出到脚本的名字。
- 如果没有使用ScriptName自定义名字，则导出的名字未默认的python化的名字。如MyFunc()变成my_func()。

在测试Python的时候，记得打开python插件。

可在

\UnrealEngine\Engine\Plugins\Experimental\PythonScriptPlugin\Source\PythonScriptPlugin\Private\PyTest.h里见到大量写好的测试用例。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPythonTestLibrary2 :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType, meta=(ScriptName="MyPythonLib"))
class INSIDER_API UMyPythonTestLibrary :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    //unreal.MyPythonLib.my_script_func_default()
    UFUNCTION(BlueprintCallable, meta=())
    static void MyScriptFuncDefault()
    {
        UInsiderSubsystem::Get().PrintStringEx(nullptr,
TEXT("MyScriptFuncDefault"));
    }

    //unreal.MyPythonLib.my_script_func()
    UFUNCTION(BlueprintCallable, meta=(ScriptName="MyScriptFunc"))
    static void MyScriptFunc_ScriptName()
    {
        UInsiderSubsystem::Get().PrintStringEx(nullptr,
TEXT("MyScriptFunc_ScriptName"));
    }
};
```

测试效果：

开启编辑器后，引擎会自动根据类型数据信息反射生成向相应的导出到py的胶水代码，我们在C++中定义的类就可以在\Intermediate\PythonStub[unreal.py](<http://unreal.py/>)里查看其导出的脚本代码。

如上的类，在unreal.py生成的py代码如下：

- 可见UMyPythonTestLibrary2 没有加ScriptName就是默认的名字，而UMyPythonTestLibrary 的名字变成了MyPythonLib。
- MyScriptFuncDefault的导出脚本名字是my_script_func_default，而MyScriptFunc_ScriptName 因为写了ScriptName变成了MyScriptFunc

```
class MyPythonTestLibrary2(BlueprintFunctionLibrary):
    r"""
    My Python Test Library 2

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPythonTest.h

    """
    ...

class MyPythonLib(BlueprintFunctionLibrary):
    r"""
    My Python Test Library

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_Test.h

    """
    @classmethod
    def my_script_func_default(cls) -> None:
        r"""
        X.my_script_func_default() -> None
        My Script Func Default
        """

    ...
    @classmethod
    def my_script_func(cls) -> None:
        r"""
        X.my_script_func() -> None
        My Script Func Script Name
        """

    ...
```

原理：

在获取各个类型名字的时候，会先判断ScriptName，如果获得，就使用该名字。否则在GetFieldPythonNameImpl里会对名字进行python化处理。

```
\Engine\Plugins\Experimental\PythonScriptPlugin\Source\PythonScriptPlugin\Private
\PyGenUtil.cpp

const FName ScriptNameMetaDataKey = TEXT("ScriptName");

FString GetClassPythonName(const UClass* InClass)
{
    return GetFieldPythonNameImpl(InClass, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedClassPythonNames(const UClass* InClass)
{
    return GetDeprecatedFieldPythonNamesImpl(InClass, ScriptNameMetaDataKey);
}

FString GetStructPythonName(const UScriptStruct* InStruct)
{
    return GetFieldPythonNameImpl(InStruct, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedStructPythonNames(const UScriptStruct* InStruct)
{
    return GetDeprecatedFieldPythonNamesImpl(InStruct, ScriptNameMetaDataKey);
}

FString GetEnumPythonName(const UEnum* InEnum)
{
    return GetFieldPythonNameImpl(InEnum, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedEnumPythonNames(const UEnum* InEnum)
{
    return GetDeprecatedFieldPythonNamesImpl(InEnum, ScriptNameMetaDataKey);
}

FString GetFieldPythonNameImpl(const FFieldVariant& InField, const FName
InMetaDataKey)
{
    FString FieldName;

    // First see if we have a name override in the meta-data
    if (GetFieldPythonNameFromMetaDataImpl(InField, InMetaDataKey, FieldName))
    {
        return FieldName;
    }

    //...
}
```

ScriptNoExport

- **功能描述:** 不导出该函数或属性到脚本。
- **使用位置:** UFUNCTION, UPROPERTY
- **引擎模块:** Script
- **元数据类型:** bool
- **常用程度:** ★★★

不导出该函数或属性到脚本。

测试代码:

```
UCLASS(Blueprintable, BlueprintType, meta = (ScriptName = "MyPythonLib"))
class INSIDER_API UMyPythonTestLibrary :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static void MyScriptFunc_None();

    UFUNCTION(BlueprintCallable, meta = (ScriptNoExport))
    static void MyScriptFunc_NoExport();

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat = 123.f;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ScriptNoExport))
    float MyFloat_NoExport = 123.f;
};
```

测试效果py代码:

可见默认的函数和属性都会导出到脚本里。而MyScriptFunc_NoExport和MyFloat_NoExport在py里并没有。

```
class MyPythonLib(BlueprintFunctionLibrary):
    r"""
    My Python Test Library

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPythonTest.h

    ...
    @property
    def my_float(self) -> float:
        r"""
        (float): [Read-write]
        ...
        ...
        @my_float.setter
```

```

def my_float(self, value: float) -> None:
    ...
    @classmethod
    def my_script_func_none(cls) -> None:
        r"""
        X.my_script_func_none() -> None
        My Script Func None
        """
        ...


```

原理：

根据这个ScriptNoExport来判断一个属性或函数是否导出。

```

bool IsScriptExposedProperty(const FProperty* InProp)
{
    return !InProp->HasMetaData(ScriptNoExportMetaDataKey)
        && InProp->HasAnyPropertyParams(CPF_BlueprintVisible | CPF_BlueprintAssignable);
}

bool IsScriptExposedFunction(const UFunction* InFunc)
{
    return !InFunc->HasMetaData(ScriptNoExportMetaDataKey)
        && InFunc->HasAnyFunctionFlags(FUNC_BlueprintCallable | FUNC_BlueprintEvent)
        && !InFunc->HasMetaData(BlueprintGetterMetaDataKey)
        && !InFunc->HasMetaData(BlueprintSetterMetaDataKey)
        && !InFunc->HasMetaData(BlueprintInternalUseOnlyMetaDataKey)
        && !InFunc->HasMetaData(CustomThunkMetaDataKey)
        && !InFunc->HasMetaData(NativeBreakFuncMetaDataKey)
        && !InFunc->HasMetaData(NativeMakeFuncMetaDataKey);
}

```

ScriptOperator

- 功能描述：**把第一个参数为结构的静态函数包装成结构的运算符。
- 使用位置：**UFUNCTION
- 引擎模块：**Script
- 元数据类型：**string="a;b;c"
- 常用程度：**★★★

把第一个参数为结构的静态函数包装成结构的运算符。

- 可以包含多个运算符。

不同的运算符需要匹配不同的函数签名。规则见如下：

- bool运算符：bool
 - bool FuncName(const FMyStruct& Value); //Value的类型可以是const FMyStruct&或者直接FMyStruct
- 一元运算符：neg (取负)

- FMyStruct FuncName(const FMyStruct&);
- 比较运算符: (==, !=, <, <=, >, >=)
 - bool FuncName(const FMyStruct, OtherType); //OtherType可以是其他任何类型
- 数学运算符: (+, -, *, /, %, &, |, ^, >>, <<)
 - ReturnType FuncName(const FMyStruct&, OtherType); //ReturnType 和 OtherType可以是其他任何类型
- 数学赋值运算符: (+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=)
 - FMyStruct FuncName(const FMyStruct&, OtherType); //OtherType可以是其他任何类型

可见，如果想一个函数同时支持普通数学运算符和赋值运算符，函数签名可以是：

FMyStruct FuncName(const FMyStruct&, OtherType); //这里OtherType可以是任何类型，也可以是FMyStruct

这个也常常一起配合ScriptMethod使用，这样就可以在结构中一起提供一个运算成员函数，这个函数的名字还可以通过ScriptMethod来自定义。

源码里也常见到和ScriptMethodSelfReturn使用的例子，如+=运算符上。但其实ScriptMethodSelfReturn不是必须的，在+=的时候，自然会把返回值应用到第一个参数上。

测试代码：

```

USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonMathStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 value = 0;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_Operator_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod=HasValue, ScriptOperator =
"bool"))
    static bool IsValid(const FMyPythonMathStruct& InStruct) { return
InStruct.value != 0; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "neg"))
    static FMyPythonMathStruct Neg(const FMyPythonMathStruct& InStruct) { return
{ -InStruct.value }; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "=="))
    static bool IsEqual(const FMyPythonMathStruct& A, const FMyPythonMathStruct&
B) { return A.Value == B.Value; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "+;+="))
    static FMyPythonMathStruct AddInt(FMyPythonMathStruct InStruct, const int32
InValue) { InStruct.value += InValue; return InStruct; }
};

```

生成的py代码：

可见，在py里生成了**bool**, **eq**, **add**, **iadd**, **neg**的函数。同时isValid加上了ScriptMethod，就有了另一个has_value函数。

```
class MyPythonMathStruct(structBase):
    r"""
    My Python Math Struct

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptOperator.h

    **Editor Properties:** (see get_editor_property/set_editor_property)

    - ``value`` (int32): [Read-Write]
    .....

    def __init__(self, value: int = 0) -> None:
        ...
        @property
        def value(self) -> int:
            r"""
            (int32): [Read-Write]
            .....

            ...
            @value.setter
            def value(self, value: int) -> None:
                ...
                def has_value(self) -> bool:
                    r"""
                    x.has_value() -> bool
                    Is valid

                    Returns:
                        bool:
                    .....

                    ...
                    def __bool__(self) -> bool:
                        r"""
                        Is valid
                        .....

                        ...
                        def __eq__(self, other: object) -> bool:
                            r"""
                            **Overloads:**

                            - ``MyPythonMathStruct`` Is Equal
                            .....

                            ...
                            def __add__(self, other: MyPythonMathStruct) -> None:
                                r"""
                                **Overloads:**
```

```

- ``int32`` Add Int
"""

...
def __iadd__(self, other: MyPythonMathStruct) -> None:
    r"""
    **Overloads:**

    - ``int32`` Add Int
"""

...
def __neg__(self) -> None:
    r"""
    Neg
"""

...

```

进行运行的测试：

可见确实支持了数学+=运算符和bool的比较。

```

LogPython: a=unreal.MyPythonMathStruct(3)
LogPython: print(a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: 3}>
LogPython: print(not a)
LogPython: False
LogPython: a+=3
LogPython: print(a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: 6}>
LogPython: print(-a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: -6}>

```

原理：

具体的包装函数都在GenerateWrappedOperator 里，具体想了解的可细看这里。

```

auto GenerateWrappedOperator = [this, &OutGeneratedWrappedTypeReferences,
&OutDirtyModules](const UFunction* InFunc, const
PyGenUtil::FGeneratedwrappedMethod& InTypeMethod)
{
    // only static functions can be hoisted onto other types
    if (!InFunc->HasAnyFunctionFlags(FUNC_Static))
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Non-static function '%s.%s' is marked as 'scriptoperator' but only static functions can be hoisted."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }

    // Get the list of operators to apply this function to
    TArray< FString> ScriptOperators;
    {
        const FString& ScriptOperatorsStr = InFunc->GetMetaData(PyGenUtil::ScriptOperatorMetaDataKey);

```

```

        ScriptOperatorsStr.ParseIntoArray(ScriptOperators, TEXT(";"));

    }

    // Go through and try and create a function for each operator, validating
    // that the signature matches what the operator expects
    for (const FString& ScriptOperator : ScriptOperators)
    {
        PyGenUtil::FGeneratedWrappedOperatorSignature OpSignature;
        if (!PyGenUtil::FGeneratedWrappedOperatorSignature::StringToSignature(*ScriptOperator, OpSignature))
        {
            REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is
marked as 'ScriptOperator' but uses an unknown operator type '%s'."),
                *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName(), *ScriptOperator);
            continue;
        }

        PyGenUtil::FGeneratedWrappedOperatorFunction OpFunc;
        {
            FString SignatureError;
            if (!OpFunc.SetFunction(InTypeMethod.MethodFunc, OpSignature,
&SignatureError))
            {
                REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is
marked as 'ScriptOperator' but has an invalid signature for the '%s' operator:
%s."),
                *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName(), *ScriptOperator,
*SignatureError);
                continue;
            }
        }

        // Ensure that we've generated a finalized Python type for this struct
        // since we'll be adding this function as a operator on that type
        const UScriptStruct* HostedStruct = CastFieldChecked<const
FStructProperty>(OpFunc.SelfParam.ParamProp)->Struct;
        if (GenerateWrappedStructType(HostedStruct,
OutGeneratedWrappedTypeReferences, OutDirtyModules,
EPyTypeGenerationFlags::ForceShouldExport))
        {
            // Find the wrapped type for the struct as that's what we'll actually
            // add the operator to (via its meta-data)
            TSharedPtr<PyGenUtil::FGeneratedWrappedStructType>
HostedStructGeneratedWrappedType =
StaticCastSharedPtr<PyGenUtil::FGeneratedWrappedStructType>
(GeneratedWrappedTypes.FindRef(PyGenUtil::GetTypeRegistryName(HostedStruct)));
            check(HostedStructGeneratedWrappedType.IsValid());
            StaticCastSharedPtr<FPyWrapperStructMetaData>
(HostedStructGeneratedWrappedType->MetaData)-
>OpStacks[(int32)OpSignature.OpType].Funcs.Add(MoveTemp(OpFunc));
        }
    }
};


```

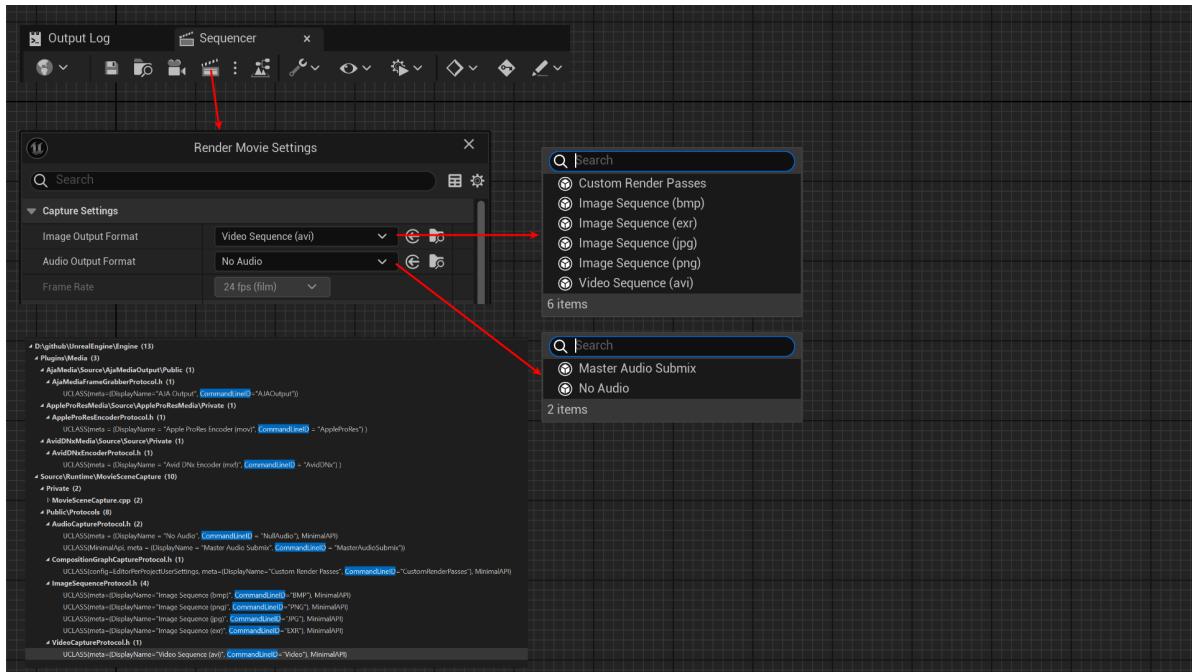
CommandLineID

- **功能描述:** 标记UMovieSceneCaptureProtocolBase的子类的协议类型。
- **使用位置:** UCCLASS
- **引擎模块:** Sequencer
- **元数据类型:** string="abc"
- **限制类型:** UMovieSceneCaptureProtocolBase的子类上
- **常用程度:** ★★

标记UMovieSceneCaptureProtocolBase的子类的协议类型。

用来在Sequencer渲染导出的时候选择到正确的处理类。一般也就引擎内部使用，除非想自己自定义渲染输出的格式协议类。

测试效果：



原理：

简单来说就是通过选择的格式名字来找到相关的ProtocolType Class

```
void UMovieSceneCapture::Initialize(TSharedPtr<FSceneViewport> InSceneViewport,
int32 PIEInstance)
{
    FString ImageProtocoloverrideString;
    if ( FParse::Value( FCommandLine::Get(), TEXT( "-MovieFormat=" ),
ImageProtocoloverrideString )
        || FParse::Value( FCommandLine::Get(), TEXT( "-ImageCaptureProtocol=" ),
), ImageProtocoloverrideString ) )
    {
        static const TCHAR* const CommandLineIDString =
TEXT("CommandLineID");
        TArray<UClass*> AllProtocolTypes =
FindAllCaptureProtocolClasses();
        for ( UClass* Class : AllProtocolTypes)
```

```

    {
        bool bMetaDataMatch = Class->GetMetaData(CommandLineIDString)
== ImageProtocolOverrideString;
        if ( bMetaDataMatch || Class->GetName() ==
ImageProtocolOverrideString )
        {
            overrideClass = Class;
        }
    }
    ImageCaptureProtocolType = overrideClass;
}

if (FParse::Value( FCommandLine::Get(), TEXT( "-AudioCaptureProtocol=" ),
AudioProtocolOverrideString ) )
{
    static const TCHAR* const CommandLineIDString =
TEXT("CommandLineID");
}
}

```

SequencerBindingResolverLibrary

- 功能描述:** 把具有SequencerBindingResolverLibrary标记的UBlueprintFunctionLibrary作为动态绑定的类。
- 使用位置:** UCLASS
- 引擎模块:** Sequencer
- 元数据类型:** bool
- 限制类型:** UClass上，但一般是UBlueprintFunctionLibrary
- 常用程度:** ★★

把具有SequencerBindingResolverLibrary标记的UBlueprintFunctionLibrary作为动态绑定的类。只把它里面的函数添加到右键菜单里。

动态绑定是Sequencer的一个新功能，简单来说就是允许设定好的轨迹变化动态应用到运行时的其他Actor上，用来做Gameplay和Sequence的过度切换会很有用。更细致用法可以参考官方文档：
<https://dev.epicgames.com/documentation/zh-cn/unreal-engine/dynamic-binding-in-sequencer>

测试代码：

```

UCLASS(meta=(SequencerBindingResolverLibrary), MinimalAPI)
class UMySequencerBindingResolverLibrary : public UBlueprintFunctionLibrary
{
GENERATED_BODY()

public:

    /** Resolve the bound object to the player's pawn */
    UFUNCTION(BlueprintPure, Category="Sequencer|Insider", meta=
(WorldContext="WorldContextObject"))
        static FMovieSceneDynamicBindingResolveResult ResolveToMyActor(UObject*
WorldContextObject, FString ActorTag);
};

```

源码：

```
UCLASS(meta=(SequencerBindingResolverLibrary), MinimalAPI)
class UBuiltInDynamicBindingResolverLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

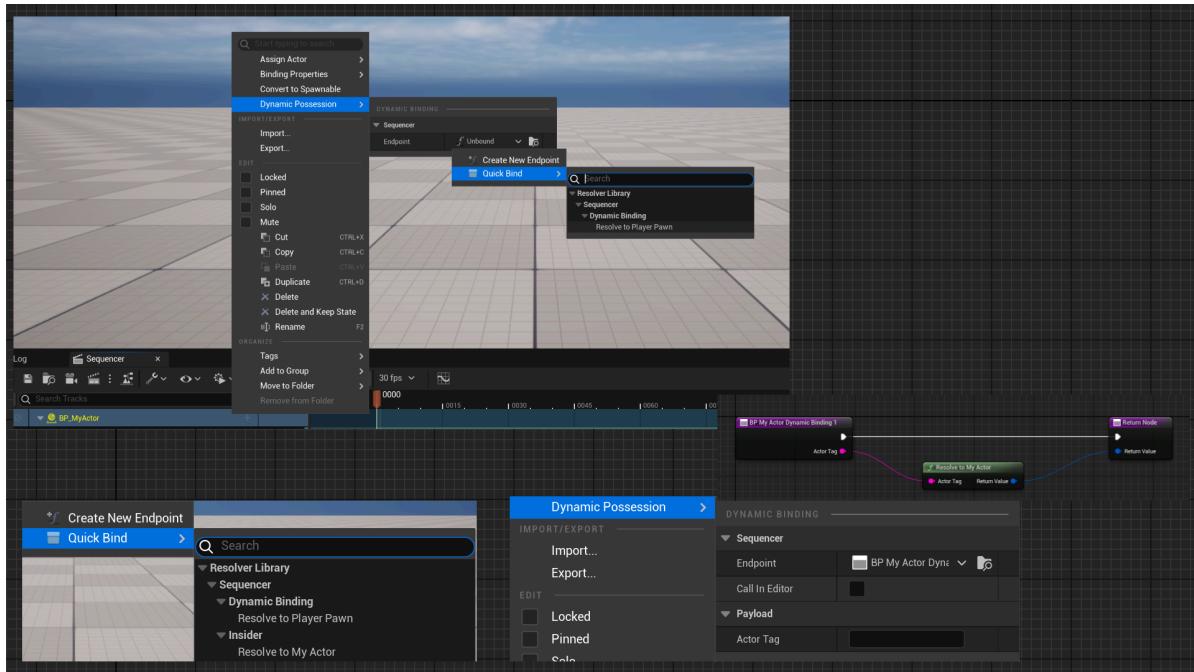
public:

    /** Resolve the bound object to the player's pawn */
    UFUNCTION(BlueprintPure, Category="Sequencer|Dynamic Binding", meta=
    (WorldContext="WorldContextObject"))
    static MOVIESCENE_API FMovieSceneDynamicBindingResolveResult
    ResolveToPlayerPawn(UObject* WorldContextObject, int32 PlayerControllerIndex =
    0);
};
```

测试结果：

在没有定义UMySequencerBindingResolverLibrary之前，引擎里有个内建的ResolveToPlayerPawn，可以把PlayerControllerIndex解析为Pawn来动态绑定到玩家的Pawn。

因此我们也可以定义自己的动态绑定函数，来解析一个FString为一个Actor，如代码里ResolveToMyActor所示。



原理：

FMovieSceneDynamicBindingCustomization会搜索引擎里的所有类，但为了缩减范围，因此只有在标有SequencerBindingResolverLibrary的类下面才去发现Resolver函数。

```
void
FMovieSceneDynamicBindingCustomization::collectResolverLibraryBindActions(UBlueprint
int* Blueprint, FBlueprintActionMenuBuilder& MenuBuilder, bool bIsRebinding)
{
```

```

    // Add any class that has the "SequencerBindingResolverLibrary" meta as a
    target class.
    //
    // We don't consider *all* blueprint function libraries because there are many,
    many of them that expose
    // functions that are, technically speaking, compatible with bound object
    resolution (i.e. they return
    // a UObject pointer) but that are completely non-sensical in this context.
    const static FName
    SequencerBindingResolverLibraryMeta("SequencerBindingResolverLibrary");
    for (TObjectIterator<UClass> ClassIt; ClassIt; ++ClassIt)
    {
        UClass* CurrentClass = *ClassIt;
        if (CurrentClass->HasMetaData(SequencerBindingResolverLibraryMeta))
        {
            FBlueprintActionFilter::Add(MenuFilter.TargetClasses, CurrentClass);
        }
    }
}

```

TakeRecorderDisplayName

- 功能描述:** 指定UTakeRecorderSource的显示名字。
- 使用位置:** UCLASS
- 引擎模块:** Sequencer
- 元数据类型:** string="abc"
- 限制类型:** UTakeRecorderSource的子类上
- 常用程度:** ★★

指定UTakeRecorderSource的显示名字。

这个一般是引擎内部自己用，除非想自己自定义UTakeRecorderSource才会派上用场。因为原理和展示过于简单，因此就不自己构建测试代码。

源码例子：

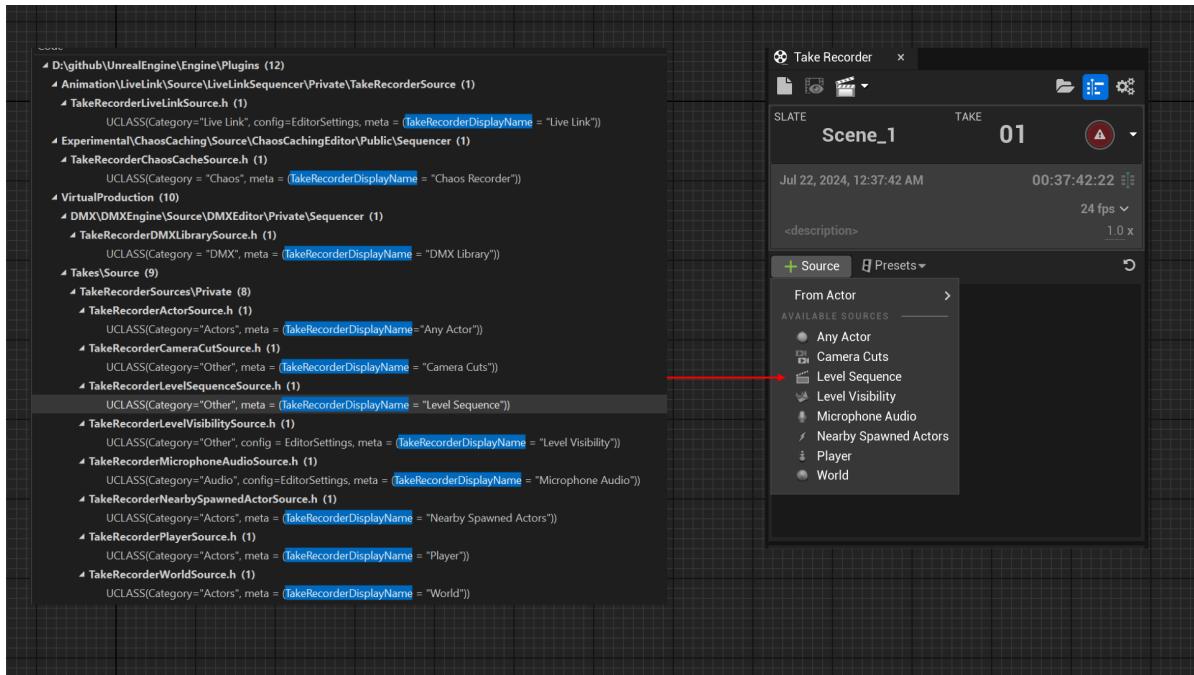
```

UCLASS(Category="Actors", meta = (TakeRecorderDisplayName = "Player"))
class UTakeRecorderPlayerSource : public UTakeRecorderSource
{}

```

测试效果：

在引擎源码中可见有多个UTakeRecorderSource，其上都标了名字。



原理：

用TakeRecorderDisplayName指定的名字来作为菜单项的名字。

```
TSharedRef<SWidget> SLevelSequenceTakeEditor::OnGenerateSourcesMenu()
{
    for (UClass* class : SourceClasses)
    {
        TSubclassOf<UTakeRecorderSource> subclassof = class;

        MenuBuilder.AddMenuEntry(
            FText::FromString(class-
>GetMetaData(TEXT("TakeRecorderDisplayName"))),
            class->GetToolTipText(true),
            FSlateIconFinder::FindIconForClass(class),
            FUIAction(
                FExecuteAction::CreateSP(this,
&SLevelSequenceTakeEditor::AddSourceFromClass, subclassof),
                FCanExecuteAction::CreateSP(this,
&SLevelSequenceTakeEditor::CanAddSourceFromClass, subclassof)
            )
        );
    }
}
```

MatchedSerializers

- 功能描述：**只在NoExportTypes.h中使用，标明采用结构序列化器。是否支持文本导入导出
- 使用位置：** UClass
- 引擎模块：** Serialization
- 元数据类型：** bool
- 关联项：**

UCLASS: MatchedSerializers

- 常用程度: 0

```
if (!GetUnrealSourceFile().IsNoExportTypes())
{
    LogError(TEXT("The 'MatchedSerializers' class specifier is only valid in the
NoExportTypes.h file"));
}
ParsedClassFlags |= CLASS_MatchedSerializers;
```

跟在Class中标记MatchedSerializers是等价的

SkipUCSModifiedProperties

- **功能描述:** 跳过序列化Component里某个属性
- **使用位置:** UPROPERTY
- **引擎模块:** Serialization
- **元数据类型:** bool
- **限制类型:** ActorComponent下的属性
- **常用程度:** 0

原理:

只在ActorComponent.cpp里用到，感觉是用于跳过序列化某个属性。

也只在UPrimitiveComponent 里的BodyInstance用到。物理的表示信息是运行时生成的，确实不需要序列化。但其实标一个Transient也就可以了，只能说是混乱的用法了。

```
UCLASS(Abstract, HideCategories=(Mobility, VirtualTexture), ShowCategories=
(PhysicsVolume), MinimalAPI)
class UPrimitiveComponent : public USceneComponent, public INavRelevantInterface,
public IInterface_AsyncCompilation, public IPhysicsComponent
{
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Collision, meta=
>ShowOnlyInnerProperties, SkipUCSModifiedProperties)
    FBodyInstance BodyInstance;
}

class FComponentPropertySkipper : public FArchive
{
public:
    FComponentPropertySkipper()
        : FArchive()
    {
        this->SetIsSaving(true);

        // Include properties that would normally skip tagged
        serialization (e.g. bulk serialization of array properties).
        ArPortFlags |= PPF_ForceTaggedSerialization;
    }

    virtual bool ShouldSkipProperty(const FProperty* InProperty) const
override
{
```

```

static const FName
MD_SkipUCSModifiedProperties(TEXT("SkipUCSModifiedProperties"));
    return (InProperty->HasAnyPropertyParams(CPF_Transient)
        || !InProperty->HasAnyPropertyParams(CPF_Edit | CPF_Interp)
        || InProperty->IsA<FMulticastDelegatePropertyParams>()

#if WITH_EDITOR
    || InProperty->HasMetaData(MD_SkipUCSModifiedProperties)
#endif
);
}

} PropertySkipper;

```

NoGetter

- 功能描述:** 阻止UHT为该属性生成一个C++的get函数，只对稀疏类的结构数据里的属性生效。
- 使用位置:** UPROPERTY
- 引擎模块:** SparseDataType
- 元数据类型:** bool
- 关联项:**
UCLASS: SparseClassDataType
- 常用程度:** ★

阻止UHT为该属性生成一个C++的get函数，只对稀疏类的结构数据里的属性生效。

这个要和SparseClassDataTypes的用法一起配合看，且NoGetter不影响蓝图里对该属性的访问。

测试代码：

```

USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    // copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (NoGetter))
    FString MyString_EditDefault_NoGetter = TEXT("MyName");
};

```

测试结果：

在生成的.generated.h中，会发现MyString_EditDefault_NoGetter 没有生成相应的C++ get函数。

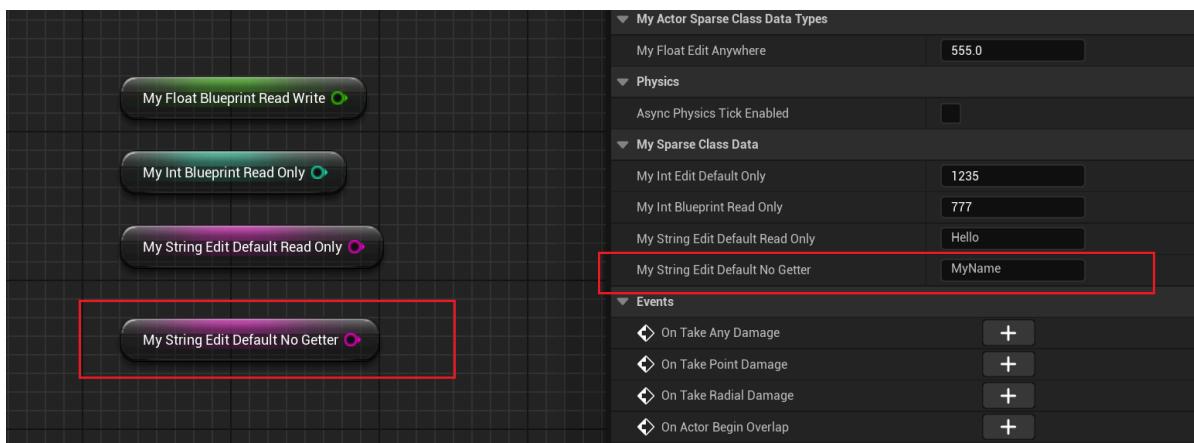
```

// "MyClass_SparseClassDataTypes.generated.h"

#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_33_SPARSE_DATA_PROPERTY_ACCESSORS \
int32 GetMyInt_EditDefaultOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)->MyInt_EditDefaultOnly; } \
int32 GetMyInt_BlueprintReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)->MyInt_BlueprintReadOnly; } \
const FString& GetMyString_EditDefault_ReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)->MyString_EditDefault_ReadOnly; }

```

而蓝图里是依然可以访问到的：



原理：

UHT在识别到SparseDataStruct后，会为其内部的属性调用AppendSparseDeclarations生成相应的C++属性Get函数（就是FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_33_SPARSE_DATA_PROPERTY_ACCESSORS那些）。而如果属性上标上NoGetter，就把该属性从SparseDataStruct里过滤掉。

```

private static IEnumerable<UhtProperty>
EnumerateSparseDataStructProperties(IEnumerable<UhtScriptStruct>
sparseScriptStructs)
{
    foreach (UhtScriptStruct sparseScriptStruct in sparseScriptStructs)
    {
        foreach (UhtProperty property in sparseScriptStruct.Properties)
        {
            if (!property.MetaData.ContainsKey(UhtNames.NoGetter))
            {
                yield return property;
            }
        }
    }
}

```

但蓝图里的属性细节面板的属性还是存在的，因为蓝图系统会分析SparseDataStruct里的所有属性并加到细节面板里去。这部分逻辑并没有判断NoGetter，因此NoGetter不影响属性在蓝图里的访问，只影响C++里的get函数。

```
if (UObject* SparseClassDataStruct = ResolvedBaseClass->GetSparseClassDataStruct())
{
    SparseClassDataInstances.Add(ResolvedBaseClass, TTuple<UObject*, void*>(SparseClassDataStruct, ResolvedBaseClass->GetOrCreateSparseClassData()));

    for (TFieldIterator<FProperty> It(SparseClassDataStruct); It; ++It)
    {
        GetCategoryProperties(classesToConsider, *It,
bShouldShowDisableEditOnInstance, bShouldShowHiddenProperties,
CategoriesFromBlueprints, CategoriesFromProperties, SortedCategories);
    }
}
```

AllowedCharacters

- 功能描述：**只允许文本框里可以输入这些字符。
- 使用位置：**UPROPERTY
- 引擎模块：**String/Text Property
- 元数据类型：**string="abc"
- 限制类型：**FName/FString/FText
- 常用程度：**★★★

只允许文本框里可以输入这些字符。

测试代码：

```
public:
    UPROPERTY(EditAnywhere, Category = AllowedCharactersTest, meta =
(AllowedCharacters = "abcde"))
    FString MyString_AllowedCharacters;
    UPROPERTY(EditAnywhere, Category = AllowedCharactersTest, meta =
(AllowedCharacters = "你好"))
    FString MyString_AllowedCharacters_Chinese;
```

测试效果：

可见第一个只能输入abcde，而fgh产生了报错。在测试中文的时候，如果粘贴进去对应的中文，则是OK的。否则也会产生报错，不允许输入进去。



原理：

SPropertyEditorText里实际保存了FCharRangeList 的AllowedCharacters用来限制字符。同样在字符串改变的时候，验证字符是否合法。

```
FCharRangeList AllowedCharacters;

AllowedCharacters.InitializeFromString(PropertyHandle-
>GetMetaData(NAME_AllowedCharacters));

bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)
{
    const FString& TextString = Text.ToString();

    if (MaxLength > 0 && TextString.Len() > MaxLength)
    {
        OutError = FText::Format(LOCTEXT("PropertyTextTooLongError", "This value
is too long ({0}/{1} characters)", TextString.Len(), MaxLength);
        return false;
    }

    if (!AllowedCharacters.IsEmpty())
    {
        if (!TextString.IsEmpty() &&
!AllowedCharacters.AreAllCharsIncluded(TextString))
        {
            TSet<TCHAR> InvalidCharacters =
AllowedCharacters.FindCharsNotIncluded(TextString);
            FString InvalidCharactersString;
            for (TCHAR Char : InvalidCharacters)
            {
                if (!InvalidCharactersString.IsEmpty())
                {
                    InvalidCharactersString.AppendChar(TEXT(' '));
                }
                InvalidCharactersString.AppendChar(Char);
            }
            OutError =
FText::Format(LOCTEXT("PropertyTextCharactersNotAllowedError", "The value may not
contain the following characters: {0}"),
FText::FromString(InvalidCharactersString));
            return false;
        }
    }

    if (PropertyValidatorFunc.IsBound())
    {
        FText Result = PropertyValidatorFunc.Execute(TextString);
        if (!Result.IsEmpty())
        {
            OutError = Result;
            return false;
        }
    }
}
```

```
    return true;  
}
```

根据FCharRangeList的定义，限制字符的格式是：

```
/** Initializes this instance with the character ranges represented by the passed  
definition string.  
* A definition string contains characters and ranges of characters, one after  
another with no special separators between them.  
* Characters - and \ must be escaped like this: \- and \\  
*  
* Examples:  
*     "aT._" <-- Letters 'a' and 'T', dot and underscore.  
*     "a-zA-Z0-9._" <-- All letters from 'a' to 'z', letter 'T', dot and underscore.  
*     "a-zA-Z0-9\-\\\._" <-- All lowercase and uppercase letters, all digits, dot  
and underscore.  
*     "a-zA-Z0-9\-\\\._" <-- All lowercase and uppercase letters, all digits,  
minus sign, backslash, dot and underscore.  
*/
```

GetKeyOptions

- 功能描述：**为TMap里的FName/FString作为Key提供细节面板里选项框的选项值
- 使用位置：**UPROPERTY
- 引擎模块：**String/Text Property
- 元数据类型：**string="abc"
- 限制类型：**TMap里FName/FString作为Key
- 关联项：**GetOptions

GetOptions

- 功能描述：**指定一个外部类的函数提供选项给FName或FString属性在细节面板中下拉选项框提供值列表。
- 使用位置：**UPARAM, UPROPERTY
- 引擎模块：**String/Text Property
- 元数据类型：**string="abc"
- 限制类型：**FString,FName
- 关联项：**GetKeyOptions, GetValueOptions
- 常用程度：**★★★★★

指定一个外部类的函数提供选项给FName或FString属性在细节面板中下拉选项框提供值列表。

- 只作用于FName或FString属性， FText不支持。
- 也可以用在容器上，比如TArray, TMap, TSet。

- 也可用在内部结构的变量上。这里的关键点是在寻找函数的时候，是通过找到OuterObject::Function来的，因此即使是内部结构的变量，也可以找到外部class里的函数。但如果另外一个不相关的类，就必须用“Module.Class.Function”这种方式才能找到，否则只能返回空。
- 函数的原型是TArray FuncName()，返回一个字符串类型，即使类型是FName，因为引擎内部会自己做转换。
- 函数可以是成员函数，有可以是静态函数。

测试代码：

```

USTRUCT(BlueprintType)
struct INSIDER_API FMyOptionsTest
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
    FString MyString_GetOptions;

    UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
    TArray<FString> MyArray_GetOptions;

    UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
    TSet<FString> MySet_GetOptions;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Text :public UObject
{
public:
    UPROPERTY(EditAnywhere, Category = GetOptions)
    FString MyString_NoOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FString MyString_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FName MyName_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FText MyText_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    TArray<FString> MyArray_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    TSet<FString> MySet_GetOptions;
};

```

```

UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
TMap<FString, int32> MyMap_GetOptions;

UFUNCTION()
static TArray<FString> MyGetOptions_Static() { return TArray<FString>{"Cat",
"Dog"}; }

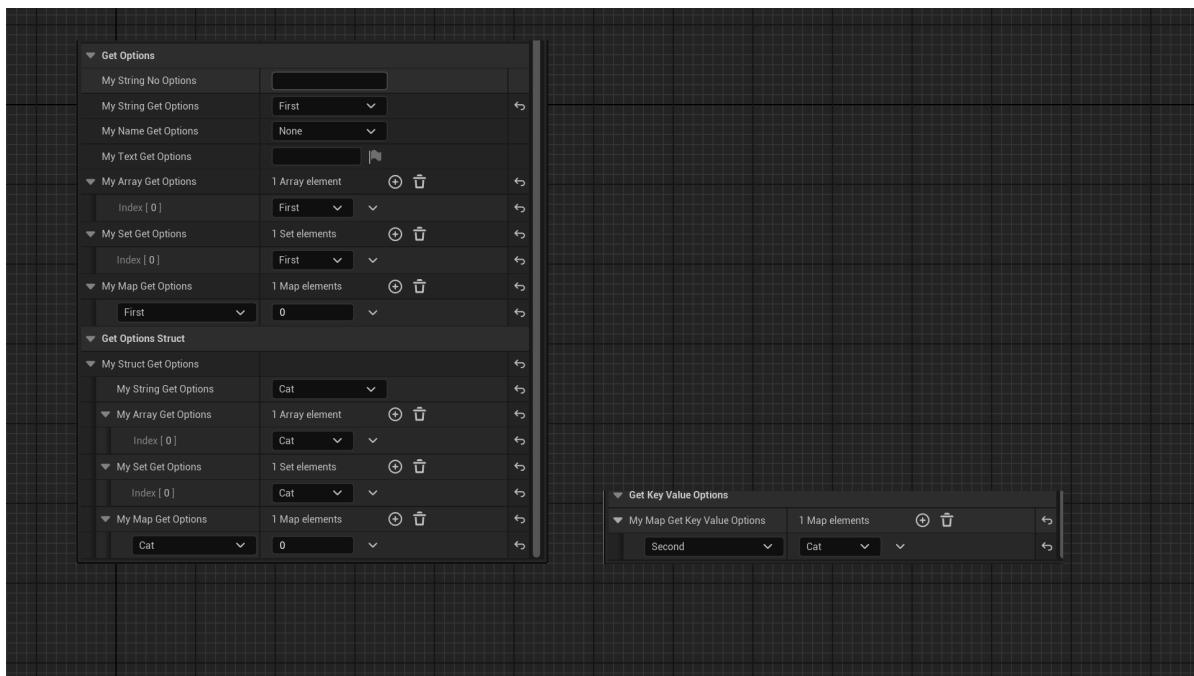
UFUNCTION()
TArray<FString> MyGetOptions() { return TArray<FString>{"First", "Second",
"Third"}; }
public:
UPROPERTY(EditAnywhere, Category = GetOptionsStruct)
FMyOptionsTest MyStruct_GetOptions;
public:
UPROPERTY(EditAnywhere, Category = GetKeyValueOptions, meta = (GetKeyOptions
= "MyGetOptions", GetValueOptions="MyGetOptions_Static"))
TMap<FString, FName> MyMap_GetKeyValueOptions;
}

```

测试效果：

根据下图，可见FText并没有起作用。其他带有GetOptions标记的在细节面板上都有一个下拉选项框。

而另外当使用TMap的时候，还可以用GetKeyOptions 和GetValueOptions来分别单独为Key和Value提供不一样的选项列表，见MyMap_GetKeyValueOptions。



原理：

大致流程是用GetPropertyOptionsMetaDataTable来判断一个属性是否支持选项框编辑，然后通过GetPropertyOptions调用指定的函数来获得选项列表，最后根据这个列表的值BuildComboBoxWidget。

```

void PropertyEditorUtils::GetPropertyOptions(TArray<UObject*>&
InOutContainers, FString& InOutPropertyName,

```

```

TArray<TSharedPtr<FString>>& InOutOptions)
{
    // Check for external function references
    if (InOutPropertyParams.Contains(TEXT(".")))
    {
        InOutContainers.Empty();
        UFunction* GetOptionsFunction = FindObject<UFunction>(nullptr,
*InOutPropertyParams, true);

        if (ensureMsgf(GetOptionsFunction && GetOptionsFunction-
>HasAnyFunctionFlags(EFunctionFlags::FUNC_Static), TEXT("Invalid GetOptions:
%s"), *InOutPropertyParams))
        {
            Uobject* GetOptionsCDO = GetOptionsFunction->GetOuterUClass()->GetDefaultObject();
            GetOptionsFunction->GetName(InOutPropertyParams);
            InOutContainers.Add(GetOptionsCDO);
        }
    }

    if (InOutContainers.Num() > 0)
    {
        TArray<FString> OptionIntersection;
        TSet<FString> OptionIntersectionSet;

        for (Uobject* Target : InOutContainers)
        {
            TArray<FString> StringOptions;
            {
                FEditorScriptExecutionGuard ScriptExecutionGuard;

                FCachedPropertyPath Path(InOutPropertyParams);
                if (!PropertyPathHelpers::GetPropertyValue(Target, Path,
StringOptions))
                {
                    TArray< FName> NameOptions;
                    if (PropertyPathHelpers::GetPropertyValue(Target, Path,
NameOptions))
                    {
                        Algo::Transform(NameOptions, StringOptions, [] (const
FName& InName) { return InName.ToString(); });
                    }
                }
            }

            // If this is the first time there won't be any options.
            if (OptionIntersection.Num() == 0)
            {
                OptionIntersection = StringOptions;
                OptionIntersectionSet = TSet< FString>(StringOptions);
            }
            else
            {
                TSet< FString> StringOptionsSet(StringOptions);
                OptionIntersectionSet =
StringOptionsSet.Intersect(OptionIntersectionSet);
            }
        }
    }
}

```

```

        OptionIntersection.RemoveAll([&OptionIntersectionSet](const
FString& Option){ return !OptionIntersectionSet.Contains(Option); });

// If we're out of possible intersected options, we can stop.
if (OptionIntersection.Num() == 0)
{
    break;
}
}

Algo::Transform(OptionIntersection, InOutOptions, [](const FString&
InString) { return MakeShared<FString>(InString); });
}

FName GetPropertyOptionsMetaDataKey(const FProperty* Property)
{
    // only string and name properties can have options
if (Property->IsA(FStrPropertyParams::StaticClass()) || Property-
>IsA(FNamePropertyParams::StaticClass()))
{
    const FProperty* OwnerProperty = Property->GetOwnerProperty();
    static const FName GetOptionsName("GetOptions");
    if (OwnerProperty->HasMetaData(GetOptionsName))
    {
        return GetOptionsName;
    }

    // Map properties can have separate options for keys and values
    const FMapPropertyParams* MapPropertyParams = CastField<FMapPropertyParams>(OwnerProperty);
    if (MapPropertyParams)
    {
        static const FName GetKeyOptionsName("GetKeyOptions");
        if (MapPropertyParams->HasMetaData(GetKeyOptionsName) && MapPropertyParams-
>GetKeyPropertyParams() == Property)
        {
            return GetKeyOptionsName;
        }

        static const FName GetValueOptionsName("GetValueOptions");
        if (MapPropertyParams->HasMetaData(GetValueOptionsName) && MapPropertyParams-
>GetValuePropertyParams() == Property)
        {
            return GetValueOptionsName;
        }
    }
}

return NAME_None;
}

TSharedPtr<SWidget> SGraphPinString::TryBuildComboBoxWidget()
{
    PropertyEditorUtils::GetPropertyOptions(PropertyContainers,
GetOptionsFunctionName, ComboBoxOptions);
}

```

```
}
```

GetValueOptions

- **功能描述:** 为TMap里的FName/FString作Value提供细节面板里选项框的选项值
- **使用位置:** UPROPERTY
- **引擎模块:** String/Text Property
- **元数据类型:** string="abc"
- **限制类型:** TMap里FName/FString作为Value
- **关联项:** GetOptions

MaxLength

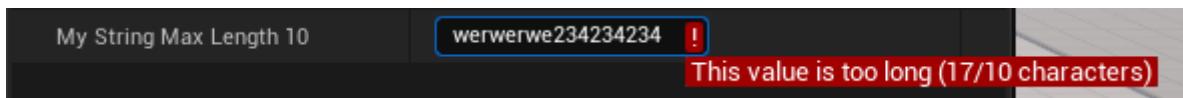
- **功能描述:** 在文本编辑框里限制文本的最大长度
- **使用位置:** UPROPERTY
- **引擎模块:** String/Text Property
- **元数据类型:** int32
- **限制类型:** FName/FString/Fext
- **常用程度:** ★★★★☆

在文本编辑框里限制文本的最大长度。但在C++或蓝图层面还是可以自己任意写入值的。

测试代码:

```
UPROPERTY(EditAnywhere, Category = MaxLengthTest, meta = (MaxLength = 10))
FString MyString_MaxLength10 = TEXT("Hello");
```

测试效果:



原理:

在文本框里字符串改变的时候，检查当前的长度，超过则报错。

如果是FName属性，则会把MaxLength继续限制在NAME_SIZE(1024)以内。

```
MaxLength = PropertyHandle->GetIntMetaData(NAME_MaxLength);
if (InPropertyEditor->PropertyIsA(FNameProperty::StaticClass()))
{
    MaxLength = MaxLength <= 0 ? NAME_SIZE - 1 : FMath::Min(MaxLength, NAME_SIZE
    - 1);
}

bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)
{
    const FString& TextString = Text.ToString();
```

```
if (MaxLength > 0 && TextString.Len() > MaxLength)
{
    OutError = FText::Format(LOCTEXT("PropertyTextTooLongError", "This
value is too long ({0}/{1} characters)", TextString.Len(), MaxLength);
    return false;
}
}
```

MultiLine

- **功能描述:** 使得文本属性编辑框支持换行。
- **使用位置:** UPROPERTY
- **引擎模块:** String/Text Property
- **元数据类型:** bool
- **限制类型:** FName/FString/Fext
- **常用程度:** ★★★★★

使得文本属性编辑框支持换行。换行后的字符串以"\r\n"分隔换行。

测试代码：

```
UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true))
FString MyString_MultiLine = TEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true))
FText MyText_MultiLine = INVTEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true,
PasswordField = true))
FString MyString_MultiLine_Password = TEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true,
PasswordField = true))
FText MyText_MultiLine_Password = INVTEXT("Hello");
```

测试结果：

按住Shift+Enter回车换行。

Password Test			
My String Password		
My Text Password	⚠	
Multi Line Test			
My String Multi Line	Hello fer		↶
My Text Multi Line	Hello erer	⚠	↶
My String Multi Line Password	Hello ewrer		↶
My Text Multi Line Password	Hello werert	⚠	↶

原理：

原理也很简单，根据bIsMultiLine 创建特定的多行编辑控件SMultiLineEditableTextBox。

```

void SPropertyEditorText::Construct( const FArguments& InArgs, const TSharedRef<
class FPropertyEditor >& InPropertyEditor )
{
    bIsMultiLine = PropertyHandle->GetBoolMetaData(NAME_MultiLine);

    if(bIsMultiLine)
    {
        childslot
        [
            SAssignNew(HorizontalBox, SHorizontalBox)
                +SHorizontalBox::slot()
                .Fillwidth(1.0f)
                [
                    SAssignNew(MultiLineWidget, SMultiLineEditableTextBox)
                        .Text(InPropertyEditor, &FPropertyEditor::GetValueAsText)
                        .Font(InArgs._Font)
                        .SelectAllTextWhenFocused(false)
                        .ClearKeyboardFocusOnCommit(false)
                        .OnTextCommitted(this, &SPropertyEditorText::OnTextCommitted)
                        .OnVerifyTextChanged(this,
                            &SPropertyEditorText::OnVerifyTextChanged)
                            .SelectAllTextOnCommit(false)
                            .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)
                            .AutoWrapText(true)
                            .ModifierKeyForNewLine(EModifierKey::Shift)
                            // .IsPassword( bIsPassword )
                ]
        ];
        PrimaryWidget = MultiLineWidget;
    }
}

```

PasswordField

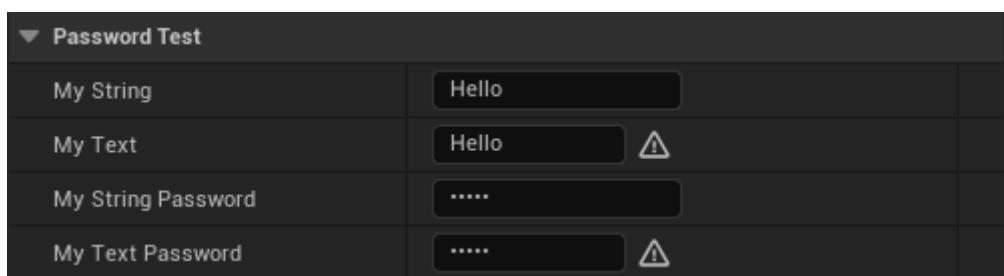
- **功能描述:** 使得文本属性显示为密码框
- **使用位置:** UPROPERTY
- **引擎模块:** String/Text Property
- **元数据类型:** bool
- **限制类型:** FName/FString/Fext
- **常用程度:** ★★★★★

使得文本属性显示为密码框。注意该文本的值在内存里依然是直接保存的，并没有加密，因此要自己来处理安全问题。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, Category = PasswordTest)  
    FString MyString = TEXT("Hello");  
  
    UPROPERTY(EditAnywhere, Category = PasswordTest)  
    FText MyText = INVTEXT("Hello");  
  
public:  
    UPROPERTY(EditAnywhere, Category = PasswordTest, meta = (PasswordField =  
true))  
    FString MyString_Password = TEXT("Hello");  
  
    UPROPERTY(EditAnywhere, Category = PasswordTest, meta = (PasswordField =  
true))  
    FText MyText_Password = INVTEXT("Hello");
```

测试效果：



原理：

该属性会导致Widget的IsPassword为true，从源码也可以得知需要注意PasswordField不能和MultiLine共用，会导致PasswordField功能失效。

```
void SPropertyEditorText::Construct( const FArguments& InArgs, const TSharedRef<  
class FPropertyEditor >& InPropertyEditor )  
{  
    const bool bIsPassword = PropertyHandle-  
>GetBoolMetaData(NAME_PasswordField);  
  
    if(bIsMultiLine)
```

```

{
    Childslot
    [
        SAssignNew(HorizontalBox, SHorizontalBox)
        +SHorizontalBox::Slot()
        .Fillwidth(1.0f)
        [
            SAssignNew(MultiLineWidget, SMultiLineEditableTextBox)
            .Text(InPropertyEditor, &FPropertyEditor::GetValueAsText)
            .Font(InArgs._Font)
            .SelectAllTextWhenFocused(false)
            .ClearKeyboardFocusOnCommit(false)
            .OnTextCommitted(this, &SPropertyEditorText::OnTextCommitted)
            .OnVerifyTextChanged(this,
                &SPropertyEditorText::OnVerifyTextChanged)
                .SelectAllTextOnCommit(false)
                .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)
                .AutoWrapText(true)
                .ModifierKeyForNewLine(EModifierKey::Shift)
                // .IsPassword( bIsPassword )
            ]
        ];
    ];

    PrimaryWidget = MultiLineWidget;
}
else
{
    Childslot
    [
        SAssignNew(HorizontalBox, SHorizontalBox)
        +SHorizontalBox::Slot()
        .Fillwidth(1.0f)
        [
            SAssignNew( SingleLineWidget, SEditableTextBox )
            .Text( InPropertyEditor, &FPropertyEditor::GetValueAsText )
            .Font( InArgs._Font )
            .SelectAllTextWhenFocused( true )
            .ClearKeyboardFocusOnCommit(false)
            .OnTextCommitted( this, &SPropertyEditorText::OnTextCommitted
)
            .OnVerifyTextChanged( this,
                &SPropertyEditorText::OnVerifyTextChanged )
                .SelectAllTextOnCommit( true )
                .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)
                .IsPassword( bIsPassword )
            ]
        ];
    };

    PrimaryWidget = SingleLineWidget;
}
}

```

PropertyValidator

- **功能描述:** 用名字指定一个UFUNCTION函数来进行文本的验证

- **使用位置:** UPROPERTY
- **引擎模块:** String/Text Property
- **元数据类型:** string="abc"
- **限制类型:** FName/FString/Fext
- **常用程度:** ★★★

用名字指定一个UFUNCTION函数来进行文本的验证。

这个函数必须是用UFUNCTION修饰的，这样才能通过名字找到。且因为搜索的范围在本类中，因此该函数必须定义在本类中。否则会报错：“LogPropertyNode: Warning: PropertyValidator ufunction 'MyValidateMyString' on UMyProperty_Text::MyString_PropertyValidator not found.”

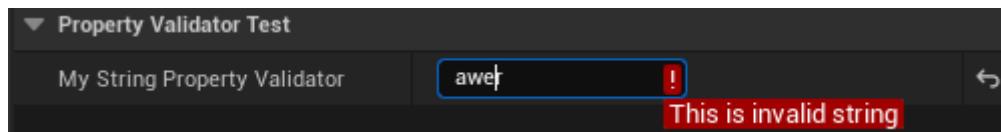
函数的签名见如下代码。返回空的FText代表没有错误，否则即是错误信息。

测试代码：

```
UPROPERTY(EditAnywhere, Category = PropertyValidatorTest, meta =
(PropertyValidator = "MyValidateMyString"))
FString MyString_PropertyValidator;

UFUNCTION()
static FText MyValidateMyString(const FString& value)
{
    if (value.Len() <= 5 && value.Contains("A"))
    {
        return FText();
    }
    return INVTEXT("This is invalid string");
}
```

测试效果：



原理：

原理也比较简单，分为两部分。一是如何找到并创建该UFunction，二是调用该函数来验证字符串。

```
const FString PropertyValidatorFunctionName = PropertyHandle-
>GetMetaData(NAME_PropertyValidator);
const UClass* OuterBaseClass = PropertyHandle->GetOuterBaseClass();
if (!PropertyValidatorFunctionName.IsEmpty() && OuterBaseClass)
{
    static TSet<FString> LoggedWarnings;

    UObject* ValidatorObject = OuterBaseClass->GetDefaultObject<UObject>();
    const UFunction* PropertyValidatorFunction = ValidatorObject-
>FindFunction(*PropertyValidatorFunctionName);
    if (PropertyValidatorFunction)
    {
        if (PropertyValidatorFunction->FunctionFlags & FUNC_Static)
```

```

    {
        PropertyValidatorFunc =
FPropertyValidatorFunc::CreateUFunction(ValidatorObject,
PropertyValidatorFunction->GetFName());
    }

bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)
{
    const FString& TextString = Text.ToString();

    if (PropertyValidatorFunc.IsBound())
    {
        FText Result = PropertyValidatorFunc.Execute(TextString);
        if (!Result.IsEmpty())
        {
            OutError = Result;
            return false;
        }
    }

    return true;
}

```

DataflowFlesh

- 功能描述:** ScriptStruct /Script/DataflowNodes.FloatOverrideDataflowNode
- 使用位置:** USTRUCT
- 引擎模块:** Struct
- 元数据类型:** bool
- 常用程度:** 0

没有在源码里找到应用的例子

HasNativeBreak

- 功能描述:** 为该结构指定一个C++内的UFunction函数作为Break节点的实现
- 使用位置:** USTRUCT
- 引擎模块:** Struct
- 元数据类型:** string="abc"
- 关联项:** HasNativeMake
- 常用程度:** ★★★★★

为该结构指定一个C++内的UFunction函数作为Break节点的实现

指定一个static UFunction函数的完整路径值，一般是"/Script/Module.Class.Function"

这个函数一般是BlueprintThreadSafe，因为这种纯Make和Break函数一般不带副作用，因此可以随便的多线程调用。

测试代码：

```

//(BlueprintType = true, HasNativeBreak =
/Script/Insider.MyHasNativeStructHelperLibrary.BreakMyHasNativeStruct,
HasNativeMake =
/Script/Insider.MyHasNativeStructHelperLibrary.MakeMyHasNativeStruct,
ModuleRelativePath = Struct/MyStruct_NativeMakeBreak.h)
USTRUCT(BlueprintType, meta = (HasNativeBreak =
"/Script/Insider.MyHasNativeStructHelperLibrary.BreakMyHasNativeStruct",
HasNativeMake =
"/Script/Insider.MyHasNativeStructHelperLibrary.MakeMyHasNativeStruct"))
struct INSIDER_API FMyStruct_HasNative
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyReadWrite;
    UPROPERTY(BlueprintReadOnly, EditAnywhere)
    float MyReadOnly;
    UPROPERTY(EditAnywhere)
    float MyNotBlueprint;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_HasDefaultMakeBreak
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyReadWrite;
    UPROPERTY(BlueprintReadOnly, EditAnywhere)
    float MyReadOnly;
    UPROPERTY(EditAnywhere)
    float MyNotBlueprint;
};

UCLASS()
class UMyHasNativeStructHelperLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

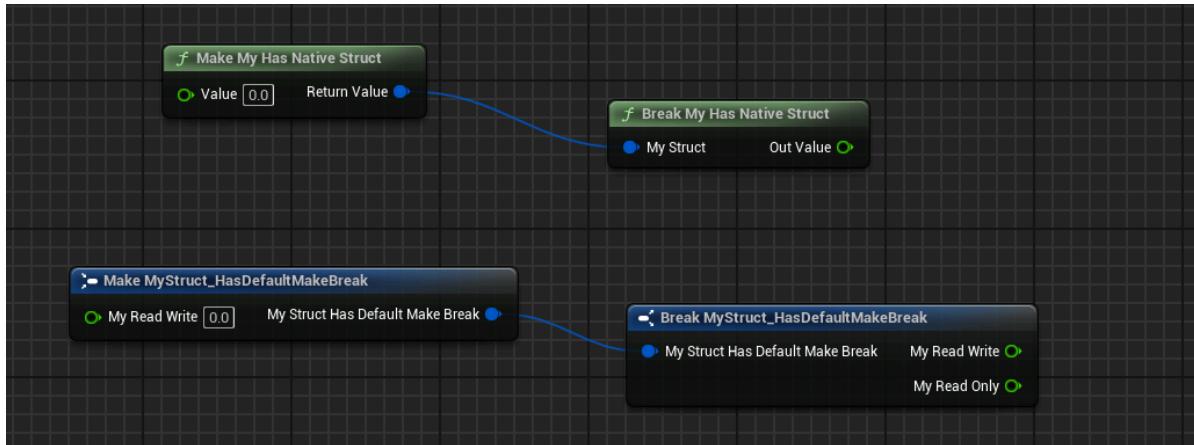
public:
    UFUNCTION(BlueprintPure, meta = (BlueprintThreadSafe))
    static void BreakMyHasNativeStruct(const FMyStruct_HasNative& myStruct,
float& outValue)
    {
        outValue = myStruct.MyReadWrite + myStruct.MyReadOnly +
myStruct.MyNotBlueprint;
    }

    UFUNCTION(BlueprintPure, meta = (BlueprintThreadSafe))
    static FMyStruct_HasNative MakeMyHasNativeStruct(float value)
    {
        FMyStruct_HasNative result;
        result.MyReadWrite = value;
        result.MyReadOnly = value;
        result.MyNotBlueprint = value;
        return result;
    }
}

```

```
};
```

蓝图节点：



原理是：

通过Meta里配置的值去找UFunction函数，因此我们配置的时候需要提供的是能找到UFunction的完整路径值。这个函数的签名会自动的被反射提取信息到UK2Node_CallFunction节点上，从而构造出不同样式的Make和Break蓝图节点。

```
E:\P4V\Engine\Source\Editor\BlueprintGraph\Private\EdGraphSchema_K2.cpp

const FString& MetaData = StructType->GetMetaData(FBlueprintMetadata::MD_NativeMakeFunction);
const UFunction* Function = FindObject<UFunction>(nullptr, *MetaData, true);

UK2Node_CallFunction* CallFunctionNode;

if (Params.bTransient || Params.CompilerContext)
{
    CallFunctionNode = (Params.bTransient ? NewObject<UK2Node_CallFunction>(Graph) : Params.CompilerContext->SpawnIntermediateNode<UK2Node_CallFunction>(GraphNode, Params.SourceGraph));
    CallFunctionNode->SetFromFunction(Function);
    CallFunctionNode->AllocateDefaultPins();
}
else
{
    FGraphNodeCreator<UK2Node_CallFunction> MakeStructCreator(*Graph);
    CallFunctionNode = MakeStructCreator.CreateNode(false);
    CallFunctionNode->SetFromFunction(Function);
    MakeStructCreator.Finalize();
}

SplitPinNode = CallFunctionNode;
```

HasNativeMake

- **功能描述：**为该结构指定一个C++内的UFunction函数作为Break节点的实现
- **使用位置：** USTRUCT

- **元数据类型:** string="abc"
- **关联项:** HasNativeBreak
- **常用程度:** ★★★★☆

IgnoreForMemberInitializationTest

- **功能描述:** 使得该属性忽略结构的未初始验证。
- **使用位置:** UPROPERTY
- **引擎模块:** Struct
- **元数据类型:** bool
- **限制类型:** C++结构下的属性
- **常用程度:** ★★

使得该属性忽略结构的未初始验证。

- 所谓未初始化，指的是C++结构里的变量没有在构造函数里初始化，也没有直接写上初始值
- 结构未初始验证指的是引擎提供的验证工具，可采用控制台命令“CoreUObject.AttemptToFindUninitializedScriptStructMembers”调用。然后会输出引擎内所有未初始化的变量信息。
- UE里的USTRUCT只是一个纯C++结构，不像用UCLASS定义的类都是一个UObject，后者的UPROPERTY属性会自动的都初始化为0，而结构里的UPROPERTY并不会自动的初始化，因此就需要我们手动的初始化。
- 从实践来说，如果开发者清楚知道某变量未初始化不会影响逻辑，那即使未初始化也并没有关系。但现实是往往大家绝大多数情况下只是单纯的犯懒或者遗忘给属性初始化。因此还是建议大家尽量给结构里的所有属性都初始化值。但一些特殊情况下，某些属性确实不适合初始化，比如源码例子里一些FGuid变量，只在真正用到的时候才手动赋值，在这之前初始化什么值其实都意义不大。在这种情况下，就可以用IgnoreForMemberInitializationTest来使该属性忽略这个验证，避免输出报错信息。

测试代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_InitTest
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=IgnoreForMemberInitializationTest)
    int32 MyProperty_IgnoreTest;
};
```

测试结果：

可见MyProperty因为没有IgnoreForMemberInitializationTest就报错了。

在控制台调用CoreUObject::AttemptToFindUninitializedScriptStructMembers后：

```
LogClass: Error: IntProperty FMyStruct_InitTest::MyProperty is not initialized  
properly. Module:Insider File:Property/Struct/MyProperty_Struct.h
```

原理：

这个命令行调用AttemptToFindUninitializedScriptStructMembers，继而继续调用FindUninitializedScriptStructMembers来查找出UScriptStruct中的未初始化变量。具体的找出方式可以在该函数中细看。

```
static void FindUninitializedScriptStructMembers(UScriptStruct* ScriptStruct,  
EScriptStructTestCtorSyntax ConstructorSyntax, TSet<const FProperty*>&  
OutUninitializedProperties)  
{  
  
    for (const FProperty* Property : TFieldRange<FProperty>(ScriptStruct,  
EFieldIteratorFlags::ExcludeSuper))  
    {  
        #if WITH_EDITORONLY_DATA  
            static const FName  
NAME_IgnoreForMemberInitializationTest(TEXT("IgnoreForMemberInitializationTest"))  
;  
            if (Property->HasMetaData(NAME_IgnoreForMemberInitializationTest))  
            {  
                continue;  
            }  
        #endif // WITH_EDITORONLY_DATA  
  
    }  
  
    //由这个调用  
FStructUtils::AttemptToFindUninitializedScriptStructMembers();  
  
    // 命令行调用  
CoreUObject::AttemptToFindUninitializedScriptStructMembers
```

MakeStructureDefaultValue

- 功能描述：**存储BP中自定义结构里的属性的默认值。
- 使用位置：**UPROPERTY
- 引擎模块：**Struct
- 元数据类型：**bool
- 限制类型：**BP里的用户自定义Struct
- 常用程度：**★

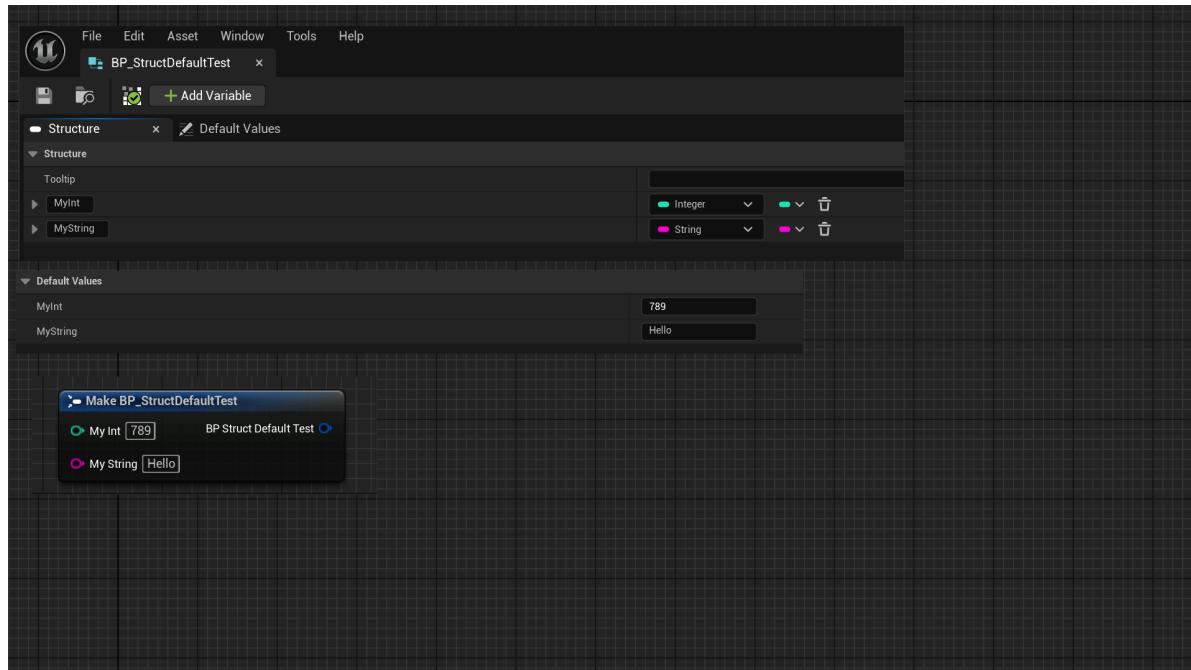
存储BP中自定义结构里的属性的默认值。

- 在C++中我们写的USTRUCT的结构里的属性默认值并不需要存储在元数据中，因为在创建该结构实例的时候，就自然会调用该结构的构造函数，从而正确初始化值。

- 而在蓝图中的用户自定义结构，并没有构造函数之类的机制。因此我们需要一个专门的Tab来填写属性的默认值。这些默认值就会存储在属性的元数据中。
-

测试代码：

在蓝图中定义一个结构BP_StructDefaultTest，并填上默认值。



测试结果：

用测试命令行打印出其相关的信息，可以见到MyInt和MyString的真正属性名以及 MakeStructureDefaultValue 的值。

```
[struct BP_StructDefaultTest   UserDefinedStruct->ScriptStruct->Struct->Field->Object /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest]
(BlueprintType = true, Tooltip = )
ObjectFlags: RF_Public | RF_Standalone | RF_Transactional | RF_WasLoaded |
RF_LoadCompleted
Outer: Package /Game/Struct/BP_StructDefaultTest
StructFlags: STRUCT_NoFlags
Size: 24
{
  (DisplayName = MyInt, Tooltip = , MakeStructureDefaultValue = 789)
  0-[4] int32 MyInt_3_CC664A574A072369083883B38EA2F129;
  PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
  ObjectFlags: RF_Public | RF_LoadCompleted
  Outer: UserDefinedStruct
/Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest
  Path: IntProperty
/Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest:MyInt_3_CC664A574A07236908
3883B38EA2F129
  (DisplayName = MyString, Tooltip = , MakeStructureDefaultValue = Hello)
  8-[16] FString MyString_6_D8FAF5D6454C781C2D5175ACF266C394;
  PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_HasGetValueTypeHash
```

```

ObjectFlags:    RF_Public | RF_LoadCompleted
Outer:  UserDefinedStruct
/Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest
    Path:  StrProperty
/Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest:MyString_6_D8FAF5D6454C781
C2D5175ACF266C394
};

```

原理：

在BP中结构创建变量保存的时候，如果发现默认值不为空，则设置到MakeStructureDefaultValue中去。之后再MakeStruct的时候就可以用上了。

```

void UK2Node_MakeStruct::FMakeStructPinManager::CustomizePinData(UEdGraphPin*
Pin, FName SourcePropertyName, int32 ArrayIndex, FProperty* Property) const
{
    const FString& MetadataDefaultValue = Property-
>GetMetaData(TEXT("MakeStructureDefaultValue"));
    if (!MetadataDefaultValue.IsEmpty())
    {
        Schema->SetPinAutogeneratedDefaultValue(Pin, MetadataDefaultValue);
        return;
    }
}

static void
FUserDefinedStructureCompilerInner::CreateVariables(UUserDefinedStruct* Struct,
const class UEdGraphSchema_K2* Schema, FCompilerResultsLog& MessageLog)
{
    if (!VarDesc.DefaultValue.IsEmpty())
    {
        VarProperty->SetMetaData(TEXT("MakeStructureDefaultValue"),
*VarDesc.DefaultValue);
    }
}

```

AllowAbstract

- 功能描述：**用于类属性，指定是否接受抽象类。
- 使用位置：**UPARAM, UPROPERTY
- 引擎模块：**TypePicker
- 元数据类型：**bool
- 限制类型：**TSubClassOf, FSoftClassPath, UClass*
- 常用程度：**★★

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
GENERATED_BODY()

```

```

};

UCLASS(BlueprintType)
class INSIDER_API UMyCommonObjectChild :public UMyCommonObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, Abstract)
class INSIDER_API UMyCommonObjectChildAbstract :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AllowAbstractTest")
    TSubclassOf<UMyCommonObject> MyClass_NotAllowAbstract;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AllowAbstractTest",
meta = (AllowAbstract))
    TSubclassOf<UMyCommonObject> MyClass_AllowAbstract;

```

测试效果：

可见增加了AllowAbstract属性的类选择器里增加了UMyCommonObjectChildAbstract这个抽象类。



原理：

其中的一个判断是IsEditInlineClassAllowed的判断，其中有bAllowAbstract。

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText *> OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property-
>IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty >(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty >(Property)->MetaClass;

        FString NoneStr( TEXT("None") );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add( FText::FromString( NoneStr ) );
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
    }
}

```

```

        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf<AActor>())
        {
            for (TObjectIterator<UClass> It; It; ++It)
            {
                if (It->IsChildOf(MetaClass)
                    && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract)
                    && (!bBlueprintBaseOnly ||
FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                    && (!InterfaceThatMustBeImplemented || It->ImplementsInterface(InterfaceThatMustBeImplemented))
                    && (!bAllowOnlyPlaceable || !It->HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
                {
                    OutOptionStrings.Add(It->GetName());
                    if (OutDisplayNames)
                    {
                        OutDisplayNames->Add(FText::FromString(It->GetName()));
                    }
                }
            }
        }

bool IsEditInlineClassAllowed( UClass* CheckClass, bool bAllowAbstract )
{
    return !CheckClass->HasAnyClassFlags(CLASS_Hidden|CLASS_HideDropDown|CLASS_Deprecated)
        && (bAllowAbstract || !CheckClass->HasAnyClassFlags(CLASS_Abstract));
}

```

AllowedClasses

- 功能描述:** 用在类或对象选择器上，指定选择的对象必须属于某一些类型基类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** strings="a, b, c"
- 限制类型:** TSubClassOf, UClass, FSoftClassPath, UObject, FSoftObjectPath, FPrimaryAssetId, FComponentReference,
- 关联项:** ExactClass, DisallowedClasses, GetAllowedClasses, GetDisallowedClasses
- 常用程度:** ★★★

用在类或对象选择器上，指定选择的对象必须属于某一些类型基类。

- 类选择器的应用属性是：TSubClassOf, UClass, FSoftClassPath。不能应用的属性是：*UScriptStruct*

- 对象选择器的应用属性是：UObject*, FSoftObjectPath, FPrimaryAssetId, FComponentReference
- 这些选择器往往会展现出一大串对象资源来供选择，因此就可以用AllowedClasses来限定其必须属于的类型。
- AllowedClasses里可以用逗号隔开多个类型，因此可以同时支持多个类型筛选。

测试代码：

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|TSubclassOf")
    TSubclassOf<UObject> MyClass_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|TSubclassOf", meta = (AllowedClasses = "MyCommonObject"))
    TSubclassOf<UObject> MyClass_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|UClass*")
    UClass* MyClassPtr_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|UClass*", meta = (AllowedClasses = "MyCommonObject"))
    UClass* MyClassPtr_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftClassPath")
    FSoftClassPath MyClass_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftClassPath", meta = (AllowedClasses = "MyCommonObject"))
    FSoftClassPath MyClass_AllowedClasses;

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftObjectPath")
    UObject* MyObject_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftObjectPath", meta = (AllowedClasses =
"/Script/Engine.Texture2D"))
    UObject* MyObject_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftObjectPath")
    FSoftObjectPath MySoftObject_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftObjectPath", meta = (AllowedClasses =
"/Script/Engine.Texture2D"))
    FSoftObjectPath MySoftObject_AllowedClasses;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FPrimaryAssetId")
FPrimaryAssetId MyPrimaryAsset_NoAllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FPrimaryAssetId", meta = (AllowedClasses =
"MyPrimaryDataAsset"))
FPrimaryAssetId MyPrimaryAsset_AllowedClasses;

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Class :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"AllowedClassesTest|FComponentReference", meta = (UseComponentPicker))
FComponentReference MyComponentReference_NoAllowedClasses;

    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"AllowedClassesTest|FComponentReference", meta =
(UseComponentPicker, AllowedClasses = "MyActorComponent"))
FComponentReference MyComponentReference_AllowedClasses;
};

UCLASS(BlueprintType)
class INSIDER_API UMyPrimaryDataAsset :public UPrimaryDataAsset
{}

```

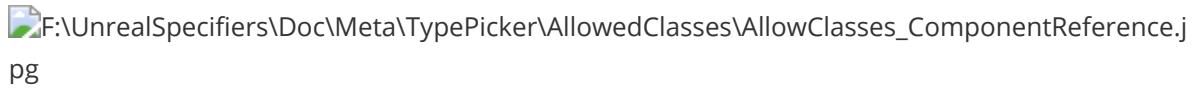
测试结果：

- 在类选择器上，可见加了AllowedClasses 之后，就限定到MyCommonObject的子类上。
- 在对象选择器上，加上了AllowedClasses = "/Script/Engine.Texture2D"之后，就把类型限定到纹理上。
- 在FPrimaryAssetId 属性的资产筛选上，加了AllowedClasses 之后，可以限定到 MyPrimaryDataAsset类型，图上是BP_MyPrimaryAsset。注意一下UMyPrimaryDataAsset 需要在ProjectSettings里设置上。



测试FComponentReference的效果：

结合上述代码，可见默认情况下， FComponentReference可选择的范围是当前Actor下所有 Component。而加上AllowedClasses后，可以把选择的范围限定到代码里描述的 MyActorComponent。



原理：

在源码里搜索可见，往往在各种类型的Customization或SPropertyEditorXXX上，会进行 AllowedClasses 和DisallowedClasses 的判断，之后再对类型进行IsChildOf的Filter筛选。

```

void FPrimaryAssetIdCustomization::CustomizeHeader(TSharedRef<class
IPropertyHandle> InStructPropertyParams, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(StructPropertyParams-
>GetMetaData("AllowedClasses"));

    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(StructPropertyParams-
>GetMetaData("DisallowedClasses"));
}

void FComponentReferenceCustomization::BuildClassFilters()
{
    const FString& AllowedClassesFilterString =PropertyParams-
>GetMetaData(NAME_AllowedClasses);

    ParseClassFilters(AllowedClassesFilterString, AllowedActorClassFilters,
AllowedComponentClassFilters);

    const FString& DisallowedClassesFilterString =PropertyParams-
>GetMetaData(NAME_DisallowedClasses);

    ParseClassFilters(DisallowedClassesFilterString,
DisallowedActorClassFilters, DisallowedComponentClassFilters);
}

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyParams, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    TArray<const UClass*> AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyParams-
>GetMetaData("AllowedClasses"));

    TArray<const UClass*> DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyParams-
>GetMetaData("DisallowedClasses"));
}

void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&
ObjectList, const FPropertyParams& MetadataProperty, TArray<const UClass*>&
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const
UClass* ObjectClass)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwner-
Property()->GetMetaData("AllowedClasses"));

    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwner-
Property()->GetMetaData("DisallowedClasses"));

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetAllowedClasses"))
    {
        const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetAllowedClasses");
    }

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetDisallowedClasses"))
    {
}

```

```

        const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetDisallowedClasses");
    }

void SPropertyEditorAsset::InitializeClassFilters(const FPropertyParams*PropertyParams)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyParams >& InPropertyParams)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*PropertyParams, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<SWidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*PropertyParams, AllowedClassFilters, DisallowedClassFilters, false);
}

```

AllowedTypes

- 功能描述:** 为FPrimaryAssetId可以指定允许的资产类型。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** strings="a, b, c"
- 限制类型:** FPrimaryAssetId
- 常用程度:** ★★★

为FPrimaryAssetId可以指定允许的资产类型。

测试代码:

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_PrimaryAsset :public UObject
{
GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "PrimaryAsset")
    FPrimaryAssetId MyPrimaryAsset;

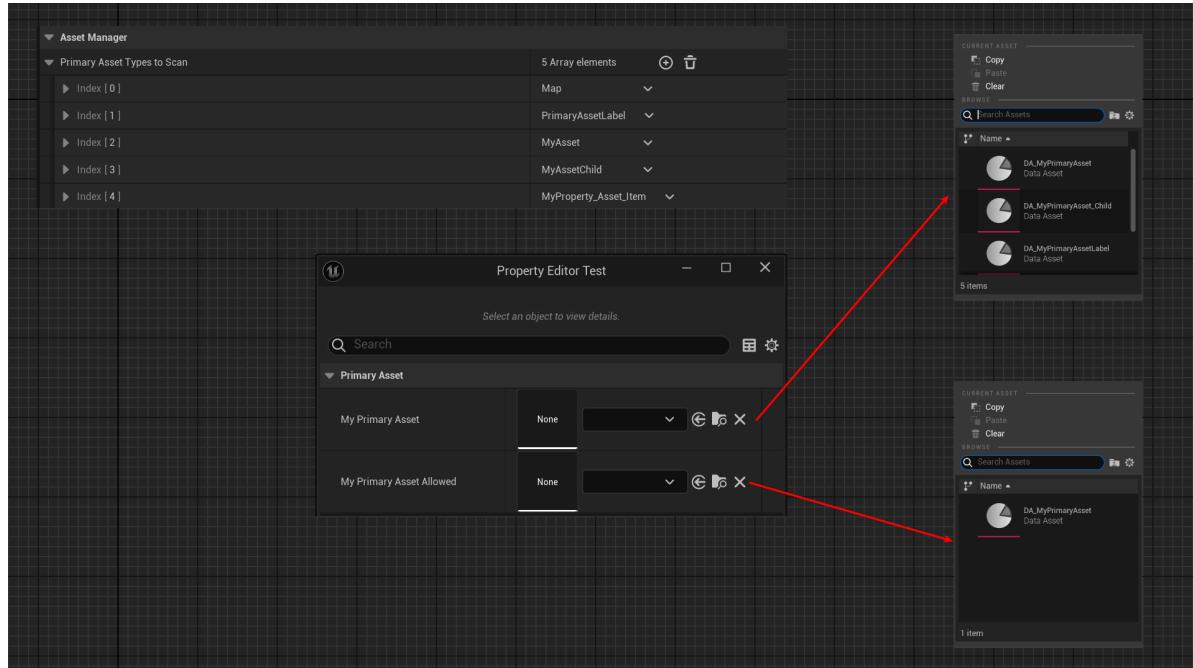
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "PrimaryAsset", meta=
(AllowedTypes="MyAsset"))
    FPrimaryAssetId MyPrimaryAsset_Allowed;
};

```

测试结果：

在项目中已经事先定义了多个UPrimaryDataSet，也在ProjectSettings里设置了。（如何定义请查看别的文章详解）。

可见MyPrimaryAsset_Allowed的选项只有一个了，说明受到了限制。



原理：

在FPrimaryAssetId 的定制化FPrimaryAssetIdCustomization中，会查看该标记，解析AllowedTypes的值并把它设置到成员变量AllowedTypes中去，最终达成筛选。

```
TArray<FPrimaryAssetType> FPrimaryAssetIdCustomization::AllowedTypes;

void FPrimaryAssetIdCustomization::CustomizeHeader(TSharedRef<class IPropertyHandle> InStructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    check(UAssetManager::IsInitialized());

    StructPropertyHandle = InStructPropertyHandle;

    const FString& TypeFilterString = StructPropertyHandle->GetMetaData("AllowedTypes");
    if( !TypeFilterString.IsEmpty() )
    {
        TArray< FString > CustomTypeFilterNames;
        TypeFilterString.ParseIntoArray(CustomTypeFilterNames, TEXT(","), true);

        for(auto It = CustomTypeFilterNames.CreateConstIterator(); It; ++It)
        {
            const FString& TypeName = *It;

            AllowedTypes.Add(*TypeName);
        }
    }
}
```

```
IAssetManagerEditorModule::MakePrimaryAssetIdSelector(
    FOnGetPrimaryAssetDisplayText::CreateSP(this,
    &FPrimaryAssetIdCustomization::GetDisplayText),
    FOnSetPrimaryAssetId::CreateSP(this,
    &FPrimaryAssetIdCustomization::OnIdSelected),
    bAllowClear, AllowedTypes, AllowedClasses, DisallowedClasses)

}
```

BaseClass

- **功能描述:** 只在StateTree模块中使用，限制FStateTreeEditorNode选择的基类类型。
- **使用位置:** UPROPERTY
- **引擎模块:** TypePicker
- **元数据类型:** bool
- **限制类型:** FStateTreeEditorNode属性
- **常用程度:** ★

只在StateTree模块中使用，限制FStateTreeEditorNode选择的基类类型。

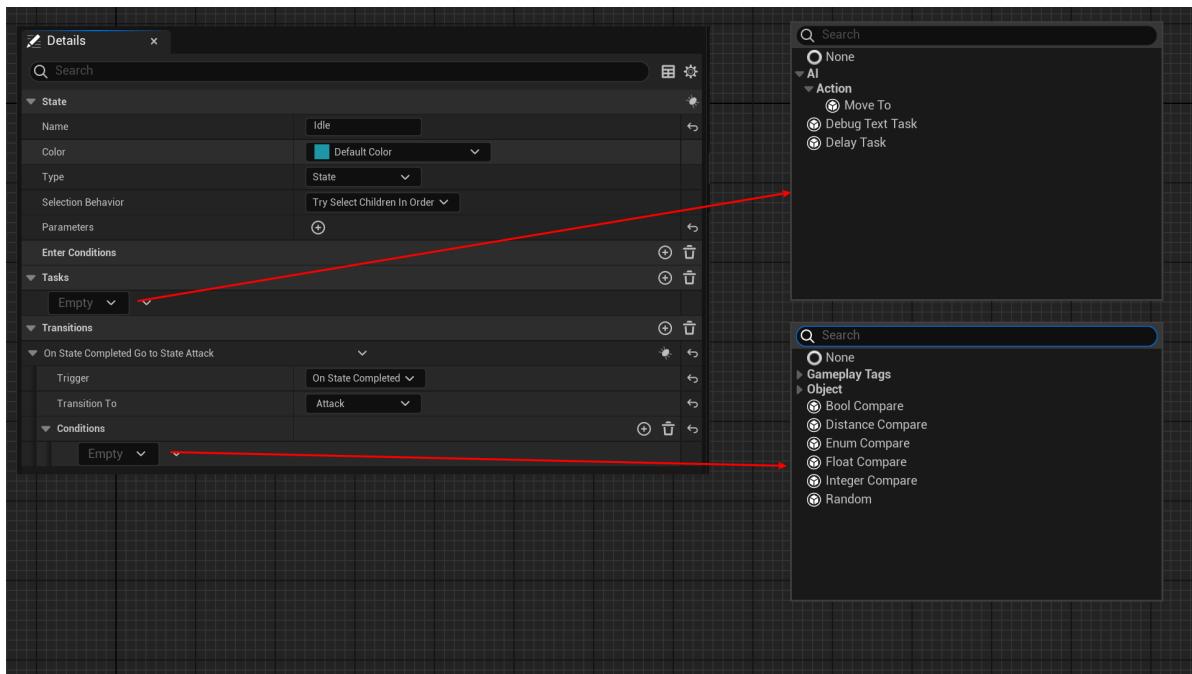
源码例子：

```
USTRUCT()
struct STATETREEEDITORMODULE_API FStateTreeTransition
{
    /** Conditions that must pass so that the transition can be triggered. */
    UPROPERTY(EditDefaultsOnly, Category = "Transition", meta = (BaseStruct =
"/Script/StateTreeModule.StateTreeConditionBase", BaseClass =
"/Script/StateTreeModule.StateTreeConditionBlueprintBase"))
    TArray<FStateTreeEditorNode> Conditions;

    UPROPERTY(EditDefaultsOnly, Category = "Tasks", meta = (BaseStruct =
"/Script/StateTreeModule.StateTreeTaskBase", BaseClass =
"/Script/StateTreeModule.StateTreeTaskBlueprintBase"))
    TArray<FStateTreeEditorNode> Tasks;
}
```

测试结果：

可见，虽然Conditions和Tasks的类型都是FStateTreeEditorNode，但是选项列表里的内容是不同的。这是由于其上面的BaseStruct和BaseClass不同，分别限定了结构的基类类型以及蓝图类的基类。



原理：

在FStateTreeNodeDetails的UI定制化上获取该属性，然后用来过滤可用的节点类型。

```
void FStateTreeNodeDetails::CustomizeHeader(TSharedRef<class IPropertyHandle> StructPropertyParamsHandle, class FDetailWidgetRow& HeaderRow, I.PropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName BaseClassMetaName(TEXT("Baseclass")); // TODO: move these names into one central place.
    const FString BaseClassName = StructPropertyParamsHandle->GetMetaData(BaseClassMetaName);
    UClass* BaseClass = UClass::TryFindTypeSlow<UClass>(BaseClassName);
}
```

BaseStruct

- 功能描述：**指定FInstancedStruct属性选项列表选择的结构都必须继承于BaseStruct指向的结构。
- 使用位置：**UPROPERTY
- 引擎模块：**TypePicker
- 元数据类型：**bool
- 限制类型：**FInstancedStruct
- 关联项：**ExcludeBaseStruct, StructTypeConst
- 常用程度：**★★★

指定FInstancedStruct属性选项列表选择的结构都必须继承于BaseStruct指向的结构。

测试代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyCommonStruct
```

```

{

};

USTRUCT(BlueprintType)
struct INSIDER_API FMyCommonStructChild:public FMyCommonStruct
{
    GENERATED_BODY()
};

USTRUCT(BlueprintType, DisplayName = "This is MyCommonStructChild")
struct INSIDER_API FMyCommonStructChild_HasDisplayName :public FMyCommonStruct
{
    GENERATED_BODY()
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Struct :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct")
    FInstancedStruct MyStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (BaseStruct = "/Script/Insider.MyCommonStruct"))
    FInstancedStruct MyStruct_BaseStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (ExcludeBaseStruct, BaseStruct = "/Script/Insider.MyCommonStruct"))
    FInstancedStruct MyStruct_ExcludeBaseStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (StructTypeConst))
    FInstancedStruct MyStruct_Const;
};

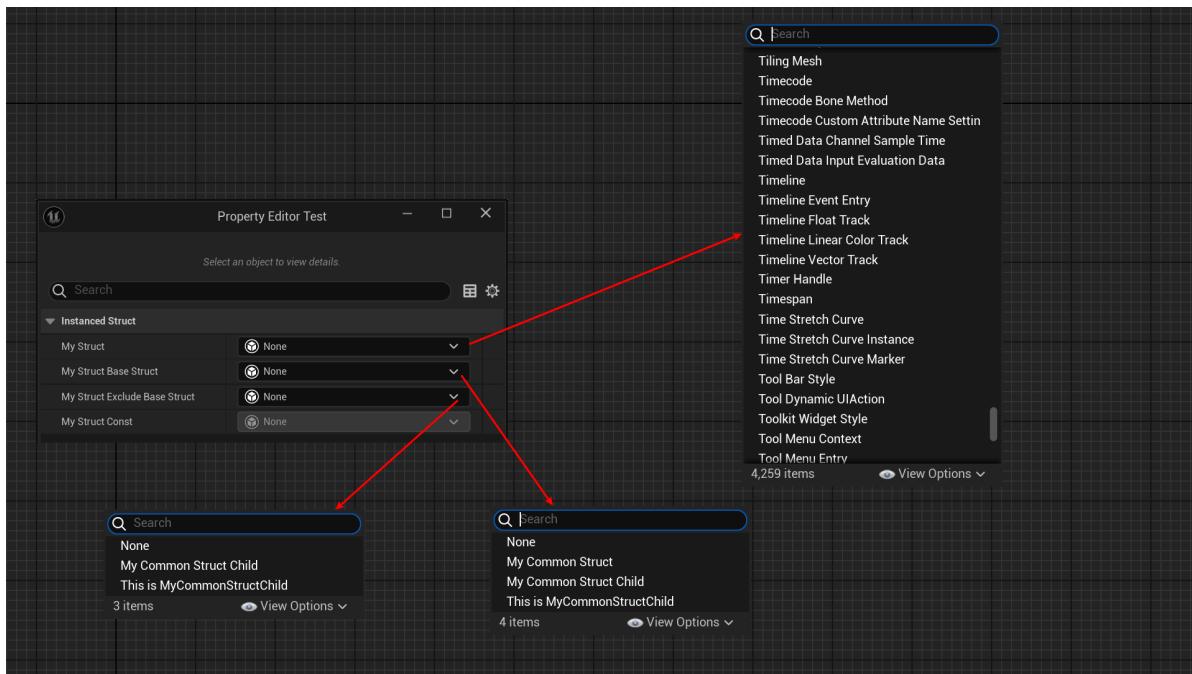
```

测试结果：

可见MyStruct_BaseStruct的选择限制在FMyCommonStruct的自己以及子类。

而如果不想要基类本身，则加上ExcludeBaseStruct的MyStruct_ExcludeBaseStruct就不包含FMyCommonStruct了。

加上StructTypeConst的MyStruct_Const就不能编辑了。



原理：

抽取BaseStruct的元信息以填充到StructFilter中去。

```

void FInstancedStructDetails::CustomizeHeader(TSharedRef<class IPropertyHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName NAME_BaseStruct = "BaseStruct";
    const FString& BaseStructName = StructProperty->GetMetaData(NAME_BaseStruct);
    if (!BaseStructName.IsEmpty())
    {
        BaseScriptStruct = UClass::TryFindTypesSlow<UScriptStruct>
(BaseStructName);
        if (!BaseScriptStruct)
        {
            BaseScriptStruct = LoadObject<UScriptStruct>(nullptr,
*BaseStructName);
        }
    }
}

TSharedRef<SWidget> FInstancedStructDetails::GenerateStructPicker()
{
    static const FName NAME_ExcludeBaseStruct = "ExcludeBaseStruct";
    static const FName NAME_HideViewOptions = "HideViewOptions";
    static const FName NAME_ShowTreeView = "ShowTreeView";

    const bool bExcludeBaseStruct = StructProperty-
>HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bHideViewOptions = StructProperty-
>HasMetaData(NAME_HideViewOptions);
}

```

```
const bool bShowTreeView = StructProperty->HasMetaData(NAME_ShowTreeView);

StructFilter->BaseStruct = BaseScriptStruct;
StructFilter->bAllowBaseStruct = !bExcludeBaseStruct;

}
```

BlueprintBaseOnly

- **功能描述:** 用于类属性，指定是否只接受可创建蓝图子类的基类
- **使用位置:** UPROPERTY
- **引擎模块:** TypePicker
- **元数据类型:** bool
- **限制类型:** TSubClassOf, FSoftClassPath, UClass*
- **常用程度:** ★★

这个限定在只想要可当作蓝图基类的类时会比较有用。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType)
class INSIDER_API UMyCommonObjectChild :public UMyCommonObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, NotBlueprintable)
class INSIDER_API UMyCommonObjectChild_NotBlueprintable :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"BlueprintBaseOnlyTest")
    TSubclassOf<UMyCommonObject> MyClass_NotBlueprintBaseOnly;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"BlueprintBaseOnlyTest", meta = (BlueprintBaseOnly))
    TSubclassOf<UMyCommonObject> MyClass_BlueprintBaseOnly;
```

测试效果：

加了BlueprintBaseOnly的限定后，UMyCommonObjectChild_NotBlueprintable 这个类因为NotBlueprintable就不能被选择了。



原理：

如果bBlueprintBaseOnly 则需要进一步判断CanCreateBlueprintOfClass，后者判断一个类是否可以创建蓝图子类。

通常情况下，如果一个C++类，没有定义Blueprintable，则无法创建蓝图子类，则不会被该属性选择到。

本身是蓝图类的都是可以再创建蓝图子类的。

```
bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>& OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems, TArray< FText *> OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property->IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty >(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty >(Property)->MetaClass;

        FString NoneStr( TEXT("None" ) );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(NoneStr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf< AActor >())
        {
            for (TObjectIterator< UClass > It; It; ++It)
            {
                if (It->IsChildOf(MetaClass)
                    && PropertyEditorHelpers::IsEditInlineClassAllowed(*It, bAllowAbstract)
                    && (!bBlueprintBaseOnly ||
                         FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                    && (!InterfaceThatMustBeImplemented || It->ImplementsInterface(InterfaceThatMustBeImplemented)))
                {
                    OutOptionStrings.Add( FText::FromName(It->GetName()) );
                    OutToolTips.Add( FText::FromName(It->GetName()) );
                    OutDisplayNames->Add( FText::FromName(It->GetName()) );
                }
            }
        }
    }
}
```

```
&& (!bAllowOnlyPlaceable || !It->
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
{
    OutOptionStrings.Add(It->GetName());
    if (OutDisplayNames)
    {
        OutDisplayNames->Add(FText::FromString(It->GetName()));
    }
}
}
}
}
```

DisallowedClasses

- **功能描述:** 用在类或对象选择器上，指定选择的对象排除掉某一些类型基类。
 - **使用位置:** UPROPERTY
 - **引擎模块:** TypePicker
 - **元数据类型:** strings="a, b, c"
 - **限制类型:** TSubClassOf, UClass*, FSoftClassPath, FComponentReference
 - **关联项:** AllowedClasses
 - **常用程度:** ★★★

用在类或对象选择器上，指定选择的对象排除掉某一些类型基类。

- 类选择器的应用属性是：TSubClassOf, UClass, FSoftClassPath。不能应用的属性是：
UScriptStruct
 - 对象选择器的应用属性是：FComponentReference。会发现相比AllowedClasses少了“UObject*”，
FSoftObjectPath, FPrimaryAssetId”，这是因为这3个对应的UI是SAssetPicker，而这个里面并没有应用FARFilter.RecursiveClassPathsExclusionSet来进行排除。

测试代码：

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|TSubclassof")  
    TSubclassOf<UObject> MyClass_NoDisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|TSubclassof", meta = (DisallowedClasses =  
"/Script/GameplayAbilities.AbilityAsync"))  
    TSubclassOf<UObject> MyClass_DisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|UClass*")  
    UClass* MyClassPtr_NoDisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|UClass*", meta = (DisallowedClasses =  
"/Script/GameplayAbilities.AbilityAsync"))
```

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Class :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"DisallowedClassesTest|FComponentReference", meta = (UseComponentPicker))
        FComponentReference MyComponentReference_NoDisallowedClasses;

    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"DisallowedClassesTest|FComponentReference", meta = (UseComponentPicker,
DisallowedClasses = "MyActorComponent"))
        FComponentReference MyComponentReference_DisallowedClasses;
};

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftClassPath")
    FSoftClassPath MySoftClass_NoDisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftClassPath", meta = (DisallowedClasses =
"/script/GameplayAbilities.AbilityAsync"))
    FSoftClassPath MySoftClass_DisallowedClasses;
public://Not work
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath")
    UObject* MyObject_NoDisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath", meta = (DisallowedClasses =
"/script/Engine.Texture2D"))
    UObject* MyObject_DisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath")
    FSoftObjectPath MySoftobject_NoDisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath", meta = (DisallowedClasses =
"/script/Engine.Texture2D"))
    FSoftObjectPath MySoftObject_DisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FPrimaryAssetId")
    FPrimaryAssetId MyPrimaryAsset_NoDisallowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FPrimaryAssetId", meta = (DisallowedClasses =
"MyPrimaryDataAsset"))
    FPrimaryAssetId MyPrimaryAsset_DisallowedClasses;
}

```

测试效果：

- 在类选择器上，可见加了DisallowedClasses之后，就排除掉了AbilityAsync类。
- 而在对象选择器上，却没必要发生作用。二者的可选对象列表是一样的。原因是因为SAssetPicker并没有实际上应用DisallowedClasses。



而在FComponentReference上的测试效果是：

DisallowedClasses可以排除掉MyActorComponent。



原理：

主要的原理已经在AllowedClasses上展示出来。DisallowedClasses的数据主要是设置到DisallowedClassFilters里面去。之后在创建ClassViewer的时候设定到ClassFilter上，最后其实还是IsChildOf的判断。但也要注意并不是所有的类和对象选择器都有使用DisallowedClasses，AssetPicker上就并没有实现。

```
    TSharedRef<FPropertyEditorClassFilter> PropEdClassFilter =
MakeShared<FPropertyEditorClassFilter>();
    PropEdClassFilter->ClassPropertyMetaClass = MetaClass;
    PropEdClassFilter->InterfaceThatMustBeImplemented = RequiredInterface;
    PropEdClassFilter->bAllowAbstract = bAllowAbstract;
    PropEdClassFilter->AllowedClassFilters = AllowedClassFilters;
    PropEdClassFilter->DisallowedClassFilters = DisallowedClassFilters;

    ClassViewerOptions.ClassFilters.Add(PropEdClassFilter);

    ClassFilter = FModuleManager::LoadModuleChecked<FClassViewerModule>
("ClassViewer").CreateClassFilter(ClassViewerOptions);

template <typename TClass>
bool FPropertyEditorClassFilter::IsClassAllowedHelper(TClass InClass)
{
    bool bMatchesFlags = !InClass->HasAnyClassFlags(CLASS_Hidden |
CLASS_HideDropDown | CLASS_Deprecated) &&
        (bAllowAbstract || !InClass->HasAnyClassFlags(CLASS_Abstract));

    if (bMatchesFlags && InClass->IsChildOf(classPropertyMetaClass) &&
        (!InterfaceThatMustBeImplemented || InClass-
>ImplementsInterface(InterfaceThatMustBeImplemented)))
    {
        auto PredicateFn = [InClass](const UClass* Class)
        {
            return InClass->IsChildOf(Class);
        };

        if (DisallowedClassFilters.FindByPredicate(PredicateFn) == nullptr &&
            (AllowedClassFilters.Num() == 0 ||
AllowedClassFilters.FindByPredicate(PredicateFn) != nullptr))
    }
```

```

    {
        return true;
    }

}

return false;
}

void SAssetPicker::Construct( const FArguments& InArgs )
{
    if (InArgs._AssetPickerConfig.bAddFilterUI)
    {
        // We create available classes here. These are used to hide away the type
        // filters in the filter list that don't match this list of classes
        TArray<UClass*> FilterclassList;
        for(auto Iter = CurrentBackendFilter.ClassPaths.CreateIterator(); Iter;
        ++Iter)
        {
            FTopLevelAssetPath ClassName = (*Iter);
            UClass* FilterClass = FindObject<UClass>(ClassName);
            if(FilterClass)
            {
                FilterclassList.AddUnique(FilterClass);
            }
        }
    }
}

```

DisallowedStructs

- 功能描述:** 只在SmartObject模块中应用，用以在类选择器中排除掉某个类以及子类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** string="abc"
- 常用程度:** ★

只在SmartObject模块中应用，用以在类选择器中排除掉某个类以及子类。

源码：

```

UPROPERTY(EditDefaultsonly, Category = "SmartObject", meta=
(DisallowedStructs="/Script/SmartObjectsModule.SmartObjectsSlotAnnotation"))
TArray<FSmartObjectDefinitionDataProxy> DefinitionData;

```

ExactClass

- 功能描述:** 在同时设置AllowedClasses和GetAllowedClasses的时候，ExactClass指定只取这两个集合中类型完全一致的类型交集，否则取一致的交集再加上其子类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool

- **限制类型:** FSoftObjectPath, UObject*
- **关联项:** AllowedClasses
- **常用程度:** ★

在同时设置AllowedClasses和GetAllowedClasses的时候，ExactClass指定只取这两个集合中类型完全一致的类型交集，否则取一致的交集再加上其子类。

- 只作用于FSoftObjectPath和UObject*，因为目前只有SPropertyEditorAsset才用到这个Meta。
- 在二者区分度上很不容易理解。因为如果一致的类型刚好是其他类型的基类，则有没有取到“其子类”本身也没有关系。
- 在测试代码里，构建了一个测试例子，AllowedClasses 里取Texture2D和TextureCube，GetAllowedClasses 里取TextureLightProfile和TextureCube。我们知道TextureLightProfile是继承于Texture2D的。因此如果ExactClass==true，则最后的筛选类型是TextureCube，因为要类型完全一致。而如果ExactClass==false，则最后的筛选类型是TextureCube和TextureLightProfile，因为TextureLightProfile是继承于Texture2D，因此TextureLightProfile会被选中。

测试代码：

```

UFUNCTION()
TArray<UClass*> MyGetAllowedClassesFunc()
{
    TArray<UClass*> classes;
    classes.Add(UTextureLightProfile::StaticClass());
    classes.Add(UTextureCube::StaticClass());
    return classes;
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|UObject*", meta = (AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
UObject* MyObject_NoExactClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|UObject*", meta = (ExactClass, AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
UObject* MyObject_ExactClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|FSoftObjectPath", meta = (AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
FSoftObjectPath MySoftObject_NoExactClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|FSoftObjectPath", meta = (ExactClass, AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
FSoftObjectPath MySoftObject_ExactClass;

```

测试效果：

- 可见没有ExactClass的时候，筛选类型是TextureCube和TextureLightProfile，总共有18项。
- 而有ExactClass后，筛选类型是TextureCube，总共只有12项。

 F:\UnrealSpecifiers\Doc\Meta\TypePicker\ExactClass\ExactClass.jpg

原理：

经过测试和源码逻辑查看，确定ExactClass必须配合GetAllowedClasses来用，而且还必须 AllowedClasses同时有。因为“ExactClass”属性传递到GetAllowedAndDisallowedClasses里。在整个源码中只找到这个地方使用。而继续根据GetAllowedAndDisallowedClasses里面的逻辑，当 AllowedClasses和GetAllowedClasses里都有值的情况下，才会进bExactClass 的判断。

- 如果bExactClass ==false，则AllowedClasses和GetAllowedClasses里的值，要相等或者是对方的子类才可以，意思就是得是一个继承树里面的，然后最终取值的时候会取子类，而不是基类。
- 如果bExactClass ==true，则只有AllowedClasses和GetAllowedClasses里一致相等的才可以。

```
void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<  
FPropertyEditor>& InPropertyEditor)  
{  
    //默认就是false，因此没用到ExactClass  
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList, *Property,  
    AllowedClassFilters, DisallowedClassFilters, false);  
}  
  
void SPropertyEditorAsset::InitializeClassFilters(const FPropertyParams*PropertyParams)  
{  
    if (PropertyParams == nullptr)  
    {  
        AllowedClassFilters.Add(ObjectClass);  
        return;  
    }  
  
    // Account for the allowed classes specified in the property metadata  
    const FPropertyParams* MetadataPropertyParams = GetActualMetadataPropertyParams(Property);  
  
    bExactClass = GetTagOrBoolMetadata(MetadataPropertyParams, "ExactClass", false);  
  
    TArray<UObject*> ObjectList;  
    if (PropertyEditor && PropertyEditor->GetPropertyHandle()->IsValidHandle())  
    {  
        PropertyEditor->GetPropertyHandle()->GetOuterObjects(ObjectList);  
    }  
    else if (PropertyHandle.IsValid())  
    {  
        PropertyHandle->GetOuterObjects(ObjectList);  
    }  
  
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,  
    *PropertyParams, AllowedClassFilters, DisallowedClassFilters, bExactClass,  
    ObjectClass);  
  
    if (AllowedClassFilters.Num() == 0)  
    {
```

```

        // always add the object class to the filters
        AllowedClassFilters.Add(ObjectClass);
    }

}

void GetAllowedAndDisallowedClasses(const TArray<UObject*>& ObjectList, const FPropertyParams& MetadataProperty, TArray<const UClass*>& AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const UClass* ObjectClass)
{
    TArray<const UClass*> CurrentAllowedClassFilters =
    MoveTemp(AllowedClasses);
    ensure(AllowedClasses.IsEmpty());
    for (const UClass* MergedClass : MergedClasses)
    {
        // Keep classes that match both allow list
        for (const UClass* CurrentClass : CurrentAllowedClassFilters)
        {
            if (CurrentClass == MergedClass || (!bExactClass &&
CurrentClass->IsChildOf(MergedClass)))
            {
                AllowedClasses.Add(CurrentClass);
                break;
            }
            if (!bExactClass && MergedClass->IsChildOf(CurrentClass))
            {
                AllowedClasses.Add(MergedClass);
                break;
            }
        }
    }
}

```

ExcludeBaseStruct

- 功能描述:** 在使用BaseStruct的FInstancedStruct属性上忽略BaseStruct指向的结构基类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** FInstancedStruct
- 关联项:** BaseStruct
- 常用程度:** ★★★

在使用BaseStruct的FInstancedStruct属性上忽略BaseStruct指向的结构基类。

GetAllowedClasses

- 功能描述:** 用在类或对象选择器上，通过一个函数来指定选择的对象必须属于某一些类型基类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker

- **元数据类型:** string="abc"
- **限制类型:** TSubClassOf, UClass, UObject, FSoftObjectPath
Code: TArray<UClass*> FuncName() const;
- **关联项:** AllowedClasses
- **常用程度:** ★★

AllowedClass是用直接指定类名字字符串的方式来限定基类。而GetAllowedClasses就更进一步，允许通过一个函数来返回筛选的基类。动态和自定义的灵活性就更高了。

当然GetAllowedClasses不如AllowedClass支持那么多属性类型，只支持：TSubClassOf, UClass, UObject, FSoftObjectPath

测试代码：

```

public:
    UFUNCTION()
    TArray<UClass*> MyGetAllowedClassesFunc()
    {
        TArray<UClass*> classes;
        classes.Add(UMyCommonObject::StaticClass());
        classes.Add(UTexture2D::StaticClass());
        classes.Add(UMyPrimaryDataAsset::StaticClass());

        return classes;
    }

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|TSubclassOf")
    TSubclassOf<UObject> MyClass_NoGetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|TSubclassOf", meta = (GetAllowedClasses =
    "MyGetAllowedClassesFunc"))
    TSubclassOf<UObject> MyClass_GetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|UClass*")
    UClass* MyClassPtr_NoGetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|UClass*", meta = (GetAllowedClasses =
    "MyGetAllowedClassesFunc"))
    UClass* MyClassPtr_GetAllowedClasses;
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|FSoftObjectPath")
    UObject* MyObject_NoGetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    "GetAllowedClassesTest|FSoftObjectPath", meta = (GetAllowedClasses =
    "MyGetAllowedClassesFunc"))
    UObject* MyObject_GetAllowedClasses;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath")
FSoftObjectPath MySoftObject_NoGetAllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath", meta = (GetAllowedClasses =
"MyGetAllowedClassesFunc"))
FSoftObjectPath MySoftObject_GetAllowedClasses;

```

测试效果：

可见Class选择器把可选范围限定到了设定的3个基类上。而对象选择器也把对象限定到了这3个基类。



原理：

通过源码发现，可以应用的属性只有SPropertyEditorAsset和SPropertyEditorClass，而这其实对应着Class类型（没有FSoftClassPath）和对应UObject的Asset类型（FSoftObjectPath对应SPropertyEditorAsset）

```

void SPropertyEditorAsset::InitializeClassFilters(const FProperty* Property)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<SWidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&
ObjectList, const FProperty& MetadataProperty, TArray<const UClass*>&
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const
UClass* ObjectClass)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw
nerProperty() -> GetMetaData("AllowedClasses"));

    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw
nerProperty() -> GetMetaData("DisallowedClasses"));

    bool bMergeAllowedClasses = !AllowedClasses.IsEmpty();
}

```

```

if (MetadataProperty.GetOwnerProperty()>>HasMetaData("GetAllowedClasses"))
{
    const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()>>GetMetaData("GetAllowedClasses");
}

if (MetadataProperty.GetOwnerProperty()>>HasMetaData("GetDisallowedClasses"))
{
    const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()>>GetMetaData("GetDisallowedClasses");
    if (!GetDisallowedClassesFunctionName.IsEmpty())
    {
        for (UObject* Object : ObjectList)
        {
            const UFunction* GetDisallowedClassesFunction = Object ? Object-
>FindFunction(*GetDisallowedClassesFunctionName) : nullptr;
            if (GetDisallowedClassesFunction)
            {
                DECLARE_DELEGATE_RetVal(TArray<UClass*>,
FGetDisallowedClasses);
                DisallowedClasses.Append(FGetDisallowedClasses::Create(Object,
GetDisallowedClassesFunction->GetFName()).Execute());
            }
        }
    }
}
}

```

GetDisallowedClasses

- 功能描述:** 用在类选择器上，通过一个函数来指定选择的类型列表中排除掉某一些类型基类。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** string="abc"
- 限制类型:** TSubClassOf, UClass*
- Code: TArray<UClass*> FuncName() const;
- 关联项:** AllowedClasses
- 常用程度:** ★★

大体和GetAllowedClasses相同，只是相反的作用。

但作用的属性类型和DisallowedClasses相似，只能作用在类选择器上。因此经过测试下来，只能作用在TSubClassOf, UClass*。

测试代码：

```

UFUNCTION()
TArray<UClass*> MyGetDisallowedClassesFunc()
{
    TArray<UClass*> classes;
    classes.Add(UAbilityAsync::StaticClass());
}

```

```

        classes.Add(UTexture2D::StaticClass());
    return classes;
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|TSubclassOf")
TSubclassof<UObject> MyClass_NoGetDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|TSubclassOf", meta = (GetDisallowedClasses =
"MyGetDisallowedClassesFunc"))
TSubclassof<UObject> MyClass_GetDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|UClass*")
UClass* MyClassPtr_NoGetDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|UClass*", meta = (GetDisallowedClasses =
"MyGetDisallowedClassesFunc"))
UClass* MyClassPtr_GetDisallowedClasses;

```

测试效果：

可以发现加了GetDisallowedClasses之后，选择列表上少了一些类型。



原理：

虽然SPropertyEditorAsset和SPropertyEditorClass都用上了GetAllowedAndDisallowedClasses，因此可以利用GetDisallowedClasses。但是SPropertyEditorAsset后面是用的SAssetPicker，它里面不会用DisallowedClasses，因此实际上SPropertyEditorAsset是不支持GetDisallowedClasses的，因此UObject*类型的属性不支持GetDisallowedClasses。

```

void SPropertyEditorAsset::InitializeClassFilters(const FProperty* Property)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
    *MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
    ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
    *Property, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<SWidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
    *Property, AllowedClassFilters, DisallowedClassFilters, false);
}

```

```

void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&
ObjectList, const FProperty& MetadataProperty, TArray<const UClass*>&
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const
UClass* ObjectClass)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwnerProperty()->GetMetaData("AllowedClasses"));
    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwnerProperty()->GetMetaData("DisallowedClasses"));

    bool bMergeAllowedClasses = !AllowedClasses.IsEmpty();

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetAllowedClasses"))
    {
        const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetAllowedClasses");
    }

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetDisallowedClasses"))
    {
        const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetDisallowedClasses");
        if (!GetDisallowedClassesFunctionName.IsEmpty())
        {
            for (UObject* Object : ObjectList)
            {
                const UFunction* GetDisallowedClassesFunction = Object ? Object-
>FindFunction(*GetDisallowedClassesFunctionName) : nullptr;
                if (GetDisallowedClassesFunction)
                {
                    DECLARE_DELEGATE_RetVal(TArray<UClass*>,
FGetDisallowedClasses);
                    DisallowedClasses.Append(FGetDisallowedClasses::Create(Object,
GetDisallowedClassesFunction->GetFName()).Execute());
                }
            }
        }
    }
}

```

HideViewOptions

- 功能描述:** 用于选择Class或Struct的属性上，隐藏在类选取器中修改显示选项的功能。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** TSubClassOf, FSoftClassPath, UClass, UScriptStruct, FInstancedStruct
- 常用程度:** ★

用于选择Class或Struct的属性上，隐藏在类选取器中修改显示选项的功能。

应用的属性类型有TSubClassOf, FSoftClassPath, UClass, UScriptStruct, FInstancedStruct 这种用于选择类型的属性。如果是FSoftObjectPtr或者FSoftObjectPath这种用于选择对象的则不起作用。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|TSubclassOf", meta = (HideviewOptions))
TSubclassOf<UObject> MyClass_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|UClass*")
UClass* MyClassPtr_NotHideviewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|UClass*", meta = (HideViewOptions))
UClass* MyClassPtr_HideviewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|FSoftClassPath")
FSoftClassPath MySoftClass_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|FSoftClassPath", meta = (HideViewOptions))
FSoftClassPath MySoftClass_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|UScriptStruct*")
UScriptStruct* MyStructPtr_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|UScriptStruct*", meta = (HideViewOptions))
UScriptStruct* MyStructPtr_HideviewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|FInstancedStruct")
FInstancedStruct MyInstancedStruct_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HideViewOptionsTest|FInstancedStruct", meta = (HideViewOptions))
FInstancedStruct MyInstancedStruct_HideViewOptions;
```

测试效果：

可见如果没有HideViewOptions，则在弹出框的角落有个齿轮或者眼睛用于修改显示选项。



原理：

在源码中对于TSubClassOf和UClass有SPropertyEditorClass，对于FSOFTCLASSPATH有FSOFTCLASSPATHCustomization，对于USTRUCT有SPropertyEditorStruct，对于FINSTANCEDSTRUCT有FINSTANCEDSTRUCTDETAILS来进行UI定制化。

```
void FSOFTCLASSPATHCUSTOMIZATION::CustomizeHeader(TSharedRef<IPROPERTYHANDLE>
INPROPERTYHANDLE, FDETAILWIDGETROW& HEADERROW, IPROPERTYTYPECUSTOMIZATIONUTILS&
STRUCTCUSTOMIZATIONUTILS)
{
    const bool bShowTreeView = PropertyHandle->HasMetaData("ShowTreeView");
    const bool bHideViewOptions = PropertyHandle-
>HasMetaData("HideViewOptions");

    SNEW(SCLASSPROPERTYENTRYBOX)
        .ShowTreeView(bShowTreeView)
        .HideViewOptions(bHideViewOptions)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPROPERTYEDITORCLASS::Construct(const FArguments& InArgs, const TSharedPtr<
FPROPERTYEDITOR &> InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

void SPROPERTYEDITORSTRUCT::Construct(const FArguments& InArgs, const TSharedPtr<
class FPROPERTYEDITOR &> InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

TSharedRef<SWIDGET> FINSTANCEDSTRUCTDETAILS::GenerateStructPicker()
{
    const bool bExcludeBaseStruct = StructProperty->HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetPropertyFlags() & CPF_NoClear);
    const bool bHideViewOptions = StructProperty->HasMetaData(NAME_HideViewOptions);
    const bool bShowTreeView = StructProperty->HasMetaData(NAME_ShowTreeView);
}
```

MetaClass

- **功能描述:** 用在软引用属性上，限定要选择的对象的基类
- **使用位置:** UPROPERTY
- **引擎模块:** TypePicker
- **元数据类型:** string="abc"
- **限制类型:** FSoftClassPath, FSoftObjectPath
- **常用程度:** ★★

用在软引用属性上，限定要选择的资源的基类。

软引用属性指的是FSoftClassPath和FSoftObjectPath，这类属性本身并没有像TSubClassOf一样本身的类型限制，因此可以额外的加MetaClass来限制要选择的对象的所属于的基类。

MetaClass里的值也可以是"/Script/Engine.Actor"这种ObjectPath。

测试代码：

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|TSubclassOf", meta = (MetaClass = "MyCommonObject"))
TSubclassOf<UObject> MyClass_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath")
FSoftClassPath MySoftClass_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath", meta = (MetaClass = "MyCommonObject"))
FSoftClassPath MySoftClass_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath", meta = (MetaClass = "MyCommonObject"))
TSoftClassPtr<UObject> MySoftClassPtrT_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MetaClassTest|UClass*")
UClass* MyClassPtr_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MetaClassTest|UClass*", 
meta = (MetaClass = "MyCommonObject"))
UClass* MyClassPtr_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath")
FSoftObjectPath MySoftObject_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath", meta = (MetaClass = "MyCustomAsset"))
FSoftObjectPath MySoftObject_MetaClass;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath", meta = (MetaClass = "MyCustomAsset"))
TSoftObjectPtr<UObject> MySoftObjectPtr_T_MetaData;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|UScriptStruct*")
UScriptStruct* MyStructPtr_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|UScriptStruct*", meta = (MetaClass="MyCommonStruct"))
UScriptStruct* MyStructPtr_MetaData;

```

测试效果：

测试效果，只有MySoftClass_MetaData和MySoftObject_MetaData的选择列表里进行了筛选。



原理：

在源码里搜索，发现FSoftClassPath和FSoftObjectPath都有进行类型定制化，根据MetaClass分别设定到SClassPropertyEntryBox和SObjectPropertyEntryBox的MetaClass和AllowedClass上。

发现TSoftObjectPtr和TSoftClassPtr并没有进行定制化，因此没有支持该功能。UScriptStruct*本身也不支持该功能，虽然也是选择类型。

```

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationutils&
StructCustomizationUtils)
{
    const FString& MetaClassName = PropertyHandle->GetMetaData("MetaClass");
    const FString& RequiredInterfaceName = PropertyHandle-
>GetMetaData("RequiredInterface"); // This was the old name, switch to
MustImplement to synchronize with class property
    const FString& MustImplementName = PropertyHandle-
>GetMetaData("MustImplement");
    TArray<const UClass*> AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("AllowedClasses"));
    TArray<const UClass*> DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("DisallowedClasses"));
    const bool bAllowAbstract = PropertyHandle->HasMetaData("AllowAbstract");
    const bool bIsBlueprintBaseOnly = PropertyHandle-
>HasMetaData("IsBlueprintBaseOnly") || PropertyHandle-
>HasMetaData("BlueprintBaseOnly");
    const bool bAllowNone = !(PropertyHandle->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bShowTreeView = PropertyHandle->HasMetaData("ShowTreeView");
    const bool bHideViewOptions = PropertyHandle->HasMetaData("HideViewOptions");
    const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");

    const UClass* const MetaClass = !MetaClassName.IsEmpty() ?
        FEditorClassUtils::GetClassFromString(MetaClassName)

```

```

: UObject::StaticClass();

SNew(SClassPropertyEntryBox)
.MetaClass(MetaClass)

}

void FSoftObjectPathCustomization::CustomizeHeader( TSharedRef<IPROPERTYHandle>
InStructPropertyHandle, FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    const FString& MetaClassName = InStructPropertyHandle->GetMetaData("MetaClass");
    UClass* MetaClass = !MetaClassName.IsEmpty() ? FEditorClassUtils::GetClassFromString(MetaClassName) : UObject::StaticClass();
    TSharedRef<SOBJECTPROPERTYENTRYBOX> ObjectPropertyEntryBox = SNew(SObjectPropertyEntryBox)
        .AllowedClass(MetaClass)
        .PropertyHandle(InStructPropertyHandle)
        .ThumbnailPool(StructCustomizationUtils.GetThumbnailPool());
}

```

MetaStruct

- 功能描述:** 设定到UScriptStruct*属性上，指定选择的类型的父结构。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** string="abc"
- 限制类型:** UScriptStruct*
- 常用程度:** ★★★

设定到UScriptStruct*属性上，指定选择的类型的父结构。

测试代码：

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaStructTest|UScriptStruct*", meta = ())
UScriptStruct* MyStructPtr_NoMetaStruct;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaStructTest|UScriptStruct*", meta = (MetaStruct = "MyCommonStruct"))
UScriptStruct* MyStructPtr_MetaData;

```

测试结果：

拥有MetaStruct 的可以把类型列表筛选到MyCommonStruct的子类上。



原理：

找到MetaStruct后设置到StructFilter上的MetaStruct，最后进行筛选。定义了选择结构的基类。

```
void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<class FPropertyEditor>& InPropertyEditor)
{
    const FString& MetaStructName = Property->GetOwnerProperty()-
>GetMetaData(TEXT("MetaStruct"));
    if (!MetaStructName.IsEmpty())
    {
        MetaStruct = UClass::TryFindTypes<UScriptStruct>
(MetaStructName, EFindFirstObjectOptions::EnsureIfAmbiguous);
        if (!MetaStruct)
        {
            MetaStruct = LoadObject<UScriptStruct>(nullptr,
*MetaStructName);
        }
    }
}

virtual bool FPropertyEditorStructFilter::IsStructAllowed(const
FStructViewerInitializationOptions& InInitOptions, const UScriptStruct* InStruct,
TSharedRef<FStructViewerFilterFuncs> InFilterFuncs) override
{
    if (InStruct->IsA<UUserDefinedStruct>())
    {
        // User Defined Structs don't support inheritance, so only include them
        // if we have don't a MetaStruct set
        return MetaStruct == nullptr;
    }

    // query the native struct to see if it has the correct parent type (if any)
    return !MetaStruct || InStruct->IsChildOf(MetaStruct);
}
```

MustImplement

- **功能描述：**指定TSubClassOf或FSOFTCLASSPATH属性选择的类必须实现该接口
- **使用位置：**UPROPERTY
- **引擎模块：**TypePicker
- **元数据类型：**string="abc"
- **限制类型：**TSubClassOf, FSOFTCLASSPATH, UClass*
- **常用程度：**★★★

指定TSubClassOf或FSOFTCLASSPATH属性选择的类必须实现该接口。

- TSubClassOf, FSOFTCLASSPATH, 还有原始的UClass属性都可以用来找到一个UClass，区别是UClass是硬引用到一个具体的类对象，而FSOFTCLASSPATH是软引用到类对象的路径，不过这二者都是泛泛的UClass，并没有对子类型进行约束。而TSubClassOf虽然也是硬引用类对象，但是进一步把类型的选择范围限制到了T的子类上，在很多时候会更加的便利，特别是你已经知道你的子类范围。比如TSubClassOf或TSubClassOf。

- 在这种用于选择Class的属性上，如果不进行限制则会把引擎里的所有类都找出来让你选择，不是那么便利。
- 因此引擎里增加了一些进一步筛选的机制。MustImplement就是用于筛选指定class属性必须实现某个接口。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyCommonInterfaceChild :public UObject, public
IMyCommonInterface
{
GENERATED_BODY()
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf")
TSubclassof<UObject> MyClass_NoMustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf", meta = (MustImplement = "MyCommonInterface"))
TSubclassof<UObject> MyClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
TSubclassof<UUserWidget> MyWidgetClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath")
FSoftClassPath MySoftClass_NoMustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath", meta = (MustImplement = "MyCommonInterface"))
FSoftClassPath MySoftClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
FSoftClassPath MySoftWidgetClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*")
UClass* MyClassStar_NoMustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*", meta = (MustImplement = "MyCommonInterface"))
UClass* MyClassStar_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
UClass* MyWidgetClassStar_MustImplement;

UFUNCTION(BlueprintCallable, meta=(Category="MustImplementTest|TSubclassOf"))

```

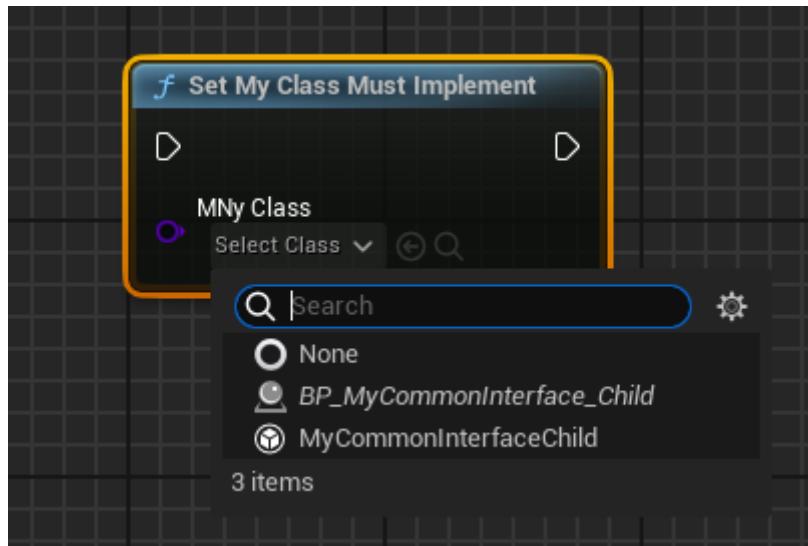
```
static void SetMyClassMustImplement(UPARAM(meta=MustImplement="MyCommonInterface")) TSubclassOf<UObject> MNyClass){}
```

测试效果：

可以发现第一个没有筛选的结果，第二和第三个有了筛选后的结果。



也可以放在在函数里作为参数：



原理：

在FPropertyHandleBase生成可能值的时候，可以看到做了一系列的筛选。

```
void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle> InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    PropertyHandle = InPropertyHandle;

    const FString& MetaClassName = PropertyHandle->GetMetaData("MetaClass");
    const FString& RequiredInterfaceName = PropertyHandle-
>GetMetaData("RequiredInterface"); // This was the old name, switch to
MustImplement to synchronize with class property
    const FString& MustImplementName = PropertyHandle-
>GetMetaData("MustImplement");
    TArray<const UCLASS*> AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("AllowedClasses"));
    TArray<const UCLASS*> DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("DisallowedClasses"));
    const bool bAllowAbstract = PropertyHandle->HasMetaData("AllowAbstract");
    const bool bIsBlueprintBaseOnly = PropertyHandle-
>HasMetaData("IsBlueprintBaseOnly") || PropertyHandle-
>HasMetaData("BlueprintBaseOnly");
    const bool bAllowNone = !(PropertyHandle->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
```

```

const bool bShowTreeView = PropertyHandle->HasMetaData("ShowTreeView");
const bool bHideViewOptions = PropertyHandle->HasMetaData("HideViewOptions");
const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");

const UClass* const MetaClass = !MetaClassName.IsEmpty()
    ? FEditorClassUtils::GetClassFromString(MetaClassName)
    : UObject::StaticClass();
const UClass* const RequiredInterface = !RequiredInterfaceName.IsEmpty()
    ? FEditorClassUtils::GetClassFromString(RequiredInterfaceName)
    : FEditorClassUtils::GetClassFromString(MustImplementName);
}

TSharedRef<SWidget> SGraphPinClass::GenerateAssetPicker()
{
    if (UEdGraphNode* ParentNode = GraphPinObj->GetOwningNode())
    {
        FString PossibleInterface = ParentNode->GetPinMetaData(GraphPinObj-
>PinName, TEXT("MustImplement"));
        if (!PossibleInterface.IsEmpty())
        {
            Filter->RequiredInterface = UClass::TryFindTypesIn

```

```

    {
        for (TObjectIterator<UClass> It; It; ++It)
        {
            if (It->IsChildOf(MetaClass)
                && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract)
                && (!bBlueprintBaseOnly ||

FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                && (!InterfaceThatMustBeImplemented || It-
>ImplementsInterface(InterfaceThatMustBeImplemented))
                && (!bAllowOnlyPlaceable || !It-
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
            {
                OutOptionStrings.Add(It->GetName());
                if (OutDisplayNames)
                {
                    OutDisplayNames->Add(FText::FromString(It->GetName()));
                }
            }
        }
    }
}

```

OnlyPlaceable

- 功能描述:** 用在类属性上，指定是否只接受可被放置到场景里的Actor
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** TSubClassOf, FSoftClassPath, UClass*
- 常用程度:** ★★

可以排除掉一些AInfo等不能放进场景里的Actor类。

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor :public AActor
{
GENERATED_BODY()

};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActorChild_Placeable :public AMyActor
{
GENERATED_BODY()

};

UCLASS(Blueprintable, BlueprintType, NotPlaceable)
class INSIDER_API AMyActorChild_NotPlaceable :public AMyActor

```

```

{
    GENERATED_BODY()
};

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "OnlyPlaceableTest")
    TSubclassOf<AMyActor> MyActor_NotOnlyPlaceable;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "OnlyPlaceableTest",
    meta = (OnlyPlaceable))
    TSubclassOf<AMyActor> MyActor_OnlyPlaceable;

```

测试效果：

可见AMyActorChild_NotPlaceable 类因为加了NotPlaceable标记，就不能被MyActor_OnlyPlaceable属性选择上。



原理：

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText>*& OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property-
>IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty>(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty>(Property)->MetaClass;

        FString NoneStr( TEXT("None" ) );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(NoneStr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf< AActor >())
        {
            for (TObjectIterator< UClass > It; It; ++It)
            {
                if (It->IsChildOf(MetaClass)
                    && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract))

```

```
&& (!bBlueprintBaseonly ||  
FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))  
&& (!InterfaceThatMustBeImplemented || It->  
>ImplementsInterface(InterfaceThatMustBeImplemented))  
&& (!bAllowOnlyPlaceable || !It->  
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))  
{  
    OutOptionStrings.Add(It->GetName());  
    if (OutDisplayNames)  
    {  
        OutDisplayNames->Add(FText::FromString(It->GetName()));  
    }  
}  
}  
}  
}
```

RowType

- **功能描述:** 指定FDataTableRowHandle 属性的可选行类型的基类。
 - **使用位置:** UPROPERTY
 - **引擎模块:** TypePicker
 - **元数据类型:** string="abc"
 - **限制类型:** FDataTableRowHandle
 - **常用程度:** ★★★

指定FDataTableRowHandle 属性的可选行类型的基类。

测试代码：

```
USTRUCT(BlueprintType)
struct FMyCommonRow : public FTableRowBase
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString MyString;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FVector MyVector;
};

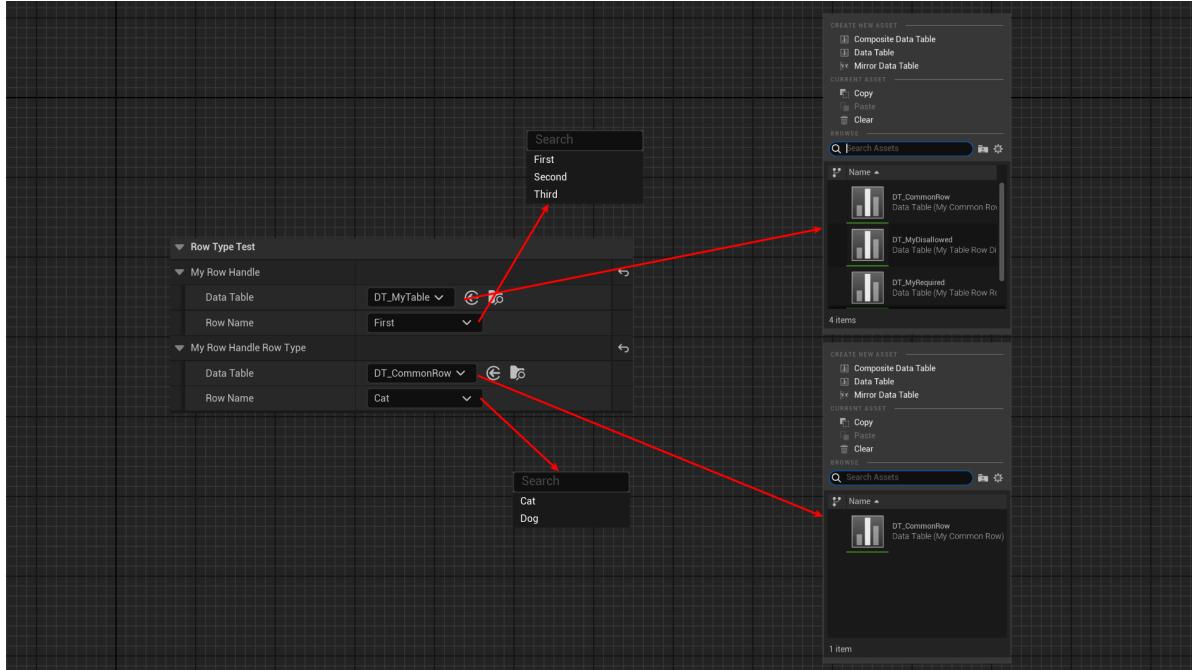
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_RowType :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "RowTypeTest")
    FDataTableRowHandle MyRowHandle;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "RowTypeTest", meta =
    (RowType = "/Script/Insider.MyCommonRow"))
    FDataTableRowHandle MyRowHandle RowType;
```

```
};
```

测试结果：

在编辑器中创建基于FMyCommonRow 的DataTable，即DT_MyCommonRow。当然项目里也有别的RowStruct的DataTable。

可以见到MyRowHandle_RowType的选项被限定到了DT_MyCommonRow，而且RowName也正确的显示了出来。



原理：

也是针对于FDataTableRowHandle这个类型进行UI的定制化，如果有该RowType数据，则赋值到RowFilterStruct，从而完成筛选。

```
void FDataTableCustomizationLayout::CustomizeHeader(TSharedRef<class IPropertyHandle> InStructPropertyParams, class FDetailWidgetRow& HeaderRow, I.PropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    if (StructPropertyParams->HasMetaData(TEXT("RowType")))
    {
        const FString& RowType = StructPropertyParams-
>GetMetaData(TEXT("RowType"));
        RowTypeFilter = FName(*RowType);
        RowFilterStruct = UClass::TryFindTypesInUObjectStruct<UScriptStruct>(RowType);
    }
}

bool FDataTableCustomizationLayout::ShouldFilterAsset(const struct FAssetData& AssetData)
{
    if (!RowTypeFilter.IsNone())
    {
        static const FName RowStructureTagName("RowStructure");
    }
}
```

```

FString RowStructure;
if (AssetData.GetTagValue<FString>(RowStructureTagName, RowStructure))
{
    if (RowStructure == RowTypeFilter.ToString())
    {
        return false;
    }

    // This is slow, but at the moment we don't have an alternative to
    // the short struct name search
    UScriptStruct* RowStruct = UClass::TryFindTypeslow<UScriptStruct>
    (RowStructure);
    if (RowStruct && RowFilterStruct && RowStruct-
>IsChildof(RowFilterStruct))
    {
        return false;
    }
}

return true;
}
return false;
}

RegisterCustomPropertyTypeLayout("DataTableRowHandle",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FDataTableCustomizationLay
out::MakeInstance));

```

ShowDisplayNames

- 功能描述:** 在Class和Struct属性上，指定类选择器显示另外的显示名称而不是类原始的名字。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** TSubClassOf, FSoftClassPath, UClass, UScriptStruct
- 常用程度:** ★

在Class和Struct属性上，指定类选择器显示另外的显示名称而不是类原始的名字。

类的显示名称指的是加在UCLASS或USTUCT上的DisplayName上的名字，这往往是对用户更友好的名字。类的原始名字就是类的类型名。

测试代码：

```

UCLASS(BlueprintType, NotBlueprintable, DisplayName="This is MyCommonObjectChild")
class INSIDER_API UMyCommonObjectChild_HasDisplayName :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

USTRUCT(BlueprintType, DisplayName="This is MyCommonStructChild")

```

```

struct INSIDER_API FMyCommonStructChild_HasDisplayName : public FMyCommonStruct
{
    GENERATED_BODY()
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|TSubclassOf", meta = ())
TSubclassof<UMyCommonObjectChild_HasDisplayName> MyClass_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|TSubclassOf", meta = (ShowDisplayNames))
TSubclassof<UMyCommonObjectChild_HasDisplayName> MyClass_ShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|UClass*", meta = (AllowedClasses =
"MyCommonObjectChild_HasDisplayName"))
UClass* MyClassPtr_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|UClass*", meta = (AllowedClasses =
"MyCommonObjectChild_HasDisplayName", ShowDisplayNames))
UClass* MyClassPtr_ShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|FSoftClassPath", meta = (MetaClass =
"MyCommonObjectChild_HasDisplayName"))
FSoftClassPath MySoftClass_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|FSoftClassPath", meta = (MetaClass =
"MyCommonObjectChild_HasDisplayName", ShowDisplayNames))
FSoftClassPath MySoftClass_ShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|UScriptStruct*", meta = (MetaStruct =
"MyCommonStructChild_HasDisplayName"))
UScriptStruct* MyStructPtr_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowDisplayNamesTest|UScriptStruct*", meta = (MetaStruct =
"MyCommonStructChild_HasDisplayName", ShowDisplayNames))
UScriptStruct* MyStructPtr_ShowDisplayNames;

```

测试结果：

可见加上ShowDisplayNames后，显示在列表里的是“This is XXX”的更友好的名字，否则就是直接的类名。

为了让效果更加直观，上面的测试代码里也加上了MetaClass, MetaStruct, AllowedClasses 用来限定选择范围。



原理：

在源码中可见，如果打开了bShowDisplayNames，则最后显示的是
(Class,Struct)→GetDisplayNameText而不是(Class,Struct)→GetName。

因为FInstancedStructDetails里并没有使用这个Meta，因此并不支持该选项。

```
void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");
    SNew(SClassPropertyEntryBox)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

static FText GetClassDisplayName(const UObject* Object, bool bShowDisplayNames)
{
    const UClass* Class = Cast<UClass>(Object);
    if (Class != nullptr)
    {
        if (bShowDisplayNames)
        {
            return Class->GetDisplayNameText();
        }

        UBlueprint* BP = UBlueprint::GetBlueprintFromClass(Class);
        if (BP != nullptr)
        {
            return FText::FromString(BP->GetName());
        }
    }
}
```

```

        return (Object) ? FText::Fromstring(Object->GetName()) :
LOCTEXT("Invalidobject", "None");
}

FText SPropertyEditorStruct::GetDisplayValue() const
{
    static bool bIsReentrant = false;

    auto GetStructDisplayName = [this](const Uobject* Inobject) -> FText
    {
        if (const UScriptStruct* Struct = Cast<UScriptStruct>(Inobject))
        {
            return bShowDisplayNames
                ? Struct->GetDisplayNameText()
                : FText::AsCultureInvariant(Struct->GetName());
        }
        return LOCTEXT("None", "None");
    };
}

```

ShowTreeView

- 功能描述:** 用于选择Class或Struct的属性上，使得在类选取器中显示为树形而不是列表。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** TSubClassOf, FSoftClassPath, UClass, UScriptStruct, FInstancedStruct
- 常用程度:** ★★

用于选择Class或Struct的属性上，使得在类选取器中显示为树形而不是列表。

应用的属性类型有TSubClassOf, FSoftClassPath, UClass, UScriptStruct, FInstancedStruct 这种用于选择类型的属性。如果是TSoftObjectPtr或者FSoftObjectPath这种用于选择对象的则不起作用。

测试代码：

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|TSubclassOf)
TSubclassof<UObject> MyClass_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|TSubclassOf", meta = (ShowTreeView))
TSubclassof<UObject> MyClass_ShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|UClass*)
UClass* MyClassPtr_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|UClass*", meta = (ShowTreeView))
UClass* MyClassPtr_ShowTreeView;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|FSoftClassPath")
FSoftClassPath MySoftClass_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|FSoftClassPath", meta = (ShowTreeView))
FSoftClassPath MySoftClass_ShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|UScriptStruct*")
UScriptStruct* MyStructPtr_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|UScriptStruct*", meta = (ShowTreeView))
UScriptStruct* MyStructPtr_ShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|FInstancedStruct")
FInstancedStruct MyInstancedStruct_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ShowTreeViewTest|FInstancedStruct", meta = (ShowTreeView))
FInstancedStruct MyInstancedStruct_ShowTreeView;

```

测试结果：

可见带有ShowTreeView的属性，在弹出的选择框上显示的是树形而不是列表。



原理：

在源码中对于TSubClassOf和 UClass 有 SPropertyEditorClass，对于 FSoftClassPath 有 FSoftClassPathCustomization，对于 UScriptStruct 有 SPropertyEditorStruct，对于 FInstancedStruct 有 FInstancedStructDetails 来进行 UI 定制化。

```

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyParams, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    const bool bShowTreeView =PropertyParams->HasMetaData("ShowTreeView");
    const bool bHideviewOptions =PropertyParams-
>HasMetaData("HideViewOptions");

    SNew(SClassPropertyEntryBox)
        .ShowTreeview(bShowTreeView)
        .HideViewOptions(bHideviewOptions)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyParams)
{
    bShowViewOptions =PropertyParams->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
}

```

```

        bShowTree = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowTreeView"));
        bShowDisplayNames = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowDisplayNames"));
    }
void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()-
>HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowDisplayNames"));
}
TSharedRef<SWidget> FInstancedStructDetails::GenerateStructPicker()
{
    const bool bExcludeBaseStruct = StructProperty-
>HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bHideViewOptions = StructProperty-
>HasMetaData(NAME_HideViewOptions);
    const bool bShowTreeView = StructProperty-
>HasMetaData(NAME_ShowTreeView);
}

```

StructTypeConst

- 功能描述:** 指定FInstancedStruct属性的类型不能在编辑器被选择。
- 使用位置:** UPROPERTY
- 引擎模块:** TypePicker
- 元数据类型:** bool
- 限制类型:** FInstancedStruct
- 关联项:** BaseStruct
- 常用程度:** ★

指定FInstancedStruct属性的类型不能在编辑器被选择。

用处往往是在之后交给用户在代码里初始化。

原理:

如果有该标记，就禁用编辑的控件。

```

void FInstancedStructDetails::CustomizeHeader(TSharedRef<class IPropertyHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName NAME_StructTypeConst = "StructTypeConst";
    const bool bEnableStructSelection = !StructProperty-
>HasMetaData(NAME_StructTypeConst);

    .Enabled(bEnableStructSelection)

}

```

CppFromBpEvent

- **使用位置:** Todo
- **引擎模块:** UHT
- **元数据类型:** bool
- **常用程度:** 0

指定这是个在C++中定义的蓝图事件。

早期的UHT会使用这个元数据，不过现在的引擎版本已经不用这个了。

原理代码：

```

public static class UhtFunctionParser
{
    private static UhtParseResult ParseUFunction(UhtParsingScope parentScope,
UhtToken token)
    {
        if (function.MetaData.ContainsKey(UhtNames.CppFromBpEvent))
        {
            function.FunctionFlags |= EFunctionFlags.Event;
        }
    }
}

```

CustomThunk

- **功能描述:** 指定UHT不为该函数生成蓝图调用的辅助函数，而需要用户自定义编写。
- **使用位置:** UFUNCTION
- **引擎模块:** UHT
- **元数据类型:** bool
- **关联项:**

UFUNCTION: ServiceRequest, CustomThunk

- **常用程度:** ★★★★☆

DocumentationPolicy

- **功能描述:** 指定文档验证的规则, 当前只能设为Strict
- **使用位置:** Any
- **引擎模块:** UHT
- **元数据类型:** string="abc"
- **常用程度:** ★

在UHT的ValidateDocumentationPolicy函数里, 会发现这个值主要是用来判断类型或字段上是否有提供Comment或Tooltip, 或者Float变量是否配了对应的“UIMin / UIMax”, 以便提取出来这些信息生成对应的文档。

当前只有一个配置是Strict, 里面默认是开启了所有的检查。所有可以理解为在C++源码里配置上DocumentationPolicy=Strict, 就意味着想要引擎来检查文档配置。

```
_documentationPolicies["Strict"] = new()
{
    ClassOrStructCommentRequired = true,
    FunctionToolTipsRequired = true,
    MemberToolTipsRequired = true,
    ParameterToolTipsRequired = true,
    FloatRangesRequired = true,
};

protected override void validateDocumentationPolicy(UhtDocumentationPolicy
policy)
{
    if (policy.classorstructcommentRequired)
    {
        string classTooltip = MetaData.GetValueOrDefault(UhtNames.Tooltip);
        if (classTooltip.Length == 0 || classTooltip.Equals(EngineName,
StringComparison.OrdinalIgnoreCase))
        {
            this.LogError($"{EngineType.CapitalizedText()} '{SourceName}'"
does not provide a tooltip / comment (DocumentationPolicy).");
        }
    }
//...
}
```

源码中的类似例子:

```
USTRUCT(meta=(DisplayName="Set Transform", Category="Transforms", TemplateName =
"Set Transform", DocumentationPolicy = "Strict",
Keywords="SetBoneTransform,SetControlTransform,SetInitialTransform,SetSpaceTransf
orm", NodeColor="0, 0.364706, 1.0", Varying))
struct CONTROLRIG_API FRigUnit_SetTransform : public FRigUnitMutable
{
```

自己的测试代码：

```
UCLASS(BlueprintType, meta = (DocumentationPolicy=Strict))
class INSIDER_API UMyClass_DocumentationPolicy :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;

    //This is a float
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (UIMin = "0.0", UIMax =
    "100.0"))
    float MyFloat_WithValidate;

    UFUNCTION(meta = (DocumentationPolicy=Strict))
    void MyFunc() {}

    /**
     * Test Func for validate param
     * @param keyOtherName The name of Key
     * @param keyValue
     */
    UFUNCTION(BlueprintCallable, meta = (DocumentationPolicy=Strict))
    int MyFunc_ValidateParamFailed(FString keyName,int keyValue){return 0;}//必须至少有一个@param才会开启参数注释的验证

    /**
     * Test Func for validate param
     *
     * @param keyName The name of key
     * @param keyValue The value of key
     * @return Return operation result
     */
    UFUNCTION(meta = (DocumentationPolicy=Strict))
    int MyFunc_ValidateParam(FString keyName,int keyValue){return 0;}
};

USTRUCT(BlueprintType, meta = (DocumentationPolicy=Strict))
struct INSIDER_API FMyStruct_DocumentationPolicy
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UENUM(BlueprintType, meta = (DocumentationPolicy=Strict))
enum class EMyEnum_DocumentationPolicy :uint8
{
```

```

        First,
        Second,
        Third,
    };

// This a tooltip / comment
UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict))
class INSIDER_API UMyClass_DocumentationPolicy_TypeA :public UObject
{
    GENERATED_BODY()
};

/***
 *  This a tooltip / comment
 *
 */
UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict))
class INSIDER_API UMyClass_DocumentationPolicy_TypeB :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict,ToolTip="This a
tooltip")) //Cannot use ShortToolTip
class INSIDER_API UMyClass_DocumentationPolicy_TypeC :public UObject
{
    GENERATED_BODY()
};

```

产生的UHT编译报错：

```

error : class 'UMyClass_DocumentationPolicy' does not provide a tooltip /
comment(DocumentationPolicy).
error : Property 'UMyClass_DocumentationPolicy::MyFloat' does not provide a
tooltip / comment(DocumentationPolicy).
error : Property 'UMyClass_DocumentationPolicy::MyString' does not provide a
tooltip / comment(DocumentationPolicy).
error : Property 'UMyClass_DocumentationPolicy::MyFloat' does not provide a valid
UIMin / UIMax(DocumentationPolicy).
error : Function 'UMyClass_DocumentationPolicy::MyFunc' does not provide a
tooltip / comment(DocumentationPolicy).
error : Function 'UMyClass_DocumentationPolicy::MyFunc' does not provide a
comment(DocumentationPolicy).
error : Function 'UMyClass_DocumentationPolicy::MyFunc_validateParamFailed'
doesn't provide a tooltip for parameter 'keyName' (DocumentationPolicy).
error : Function 'UMyClass_DocumentationPolicy::MyFunc_validateParamFailed'
doesn't provide a tooltip for parameter 'keyValue' (DocumentationPolicy).
error : Function 'UMyClass_DocumentationPolicy::MyFunc_validateParamFailed'
provides a tooltip for an unknown parameter 'keyOtherName'
error : Struct 'FMyStruct_DocumentationPolicy' does not provide a tooltip /
comment(DocumentationPolicy).
error : Property 'FMyStruct_DocumentationPolicy::MyFloat' does not provide a
tooltip / comment(DocumentationPolicy).
error : Property 'FMyStruct_DocumentationPolicy::MyString' does not provide a
tooltip / comment(DocumentationPolicy).

```

```
error : Property 'FMyStruct_DocumentationPolicy::MyFloat' does not provide a
valid UIMin / UIMax(DocumentationPolicy).
error : Enum 'EMyEnum_DocumentationPolicy' does not provide a tooltip /
comment(DocumentationPolicy)
error : Enum entry
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::First' does not
provide a tooltip / comment(DocumentationPolicy)
error : Enum entry
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::Second' does not
provide a tooltip / comment(DocumentationPolicy)
error: Enum entry
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::Third' does not
provide a tooltip / comment(DocumentationPolicy)
```

FieldNotifyInterfaceParam

- 功能描述:** 指定函数的某个参数提供FieldNotify的ViewModel信息。
- 使用位置:** UFUNCTION
- 引擎模块:** FieldNotify
- 元数据类型:** string="abc"
- 限制类型:** 函数里有其他FFieldNotificationId 参数
- 常用程度:** ★★★

指定函数的某个参数提供FieldNotify的ViewModel信息。

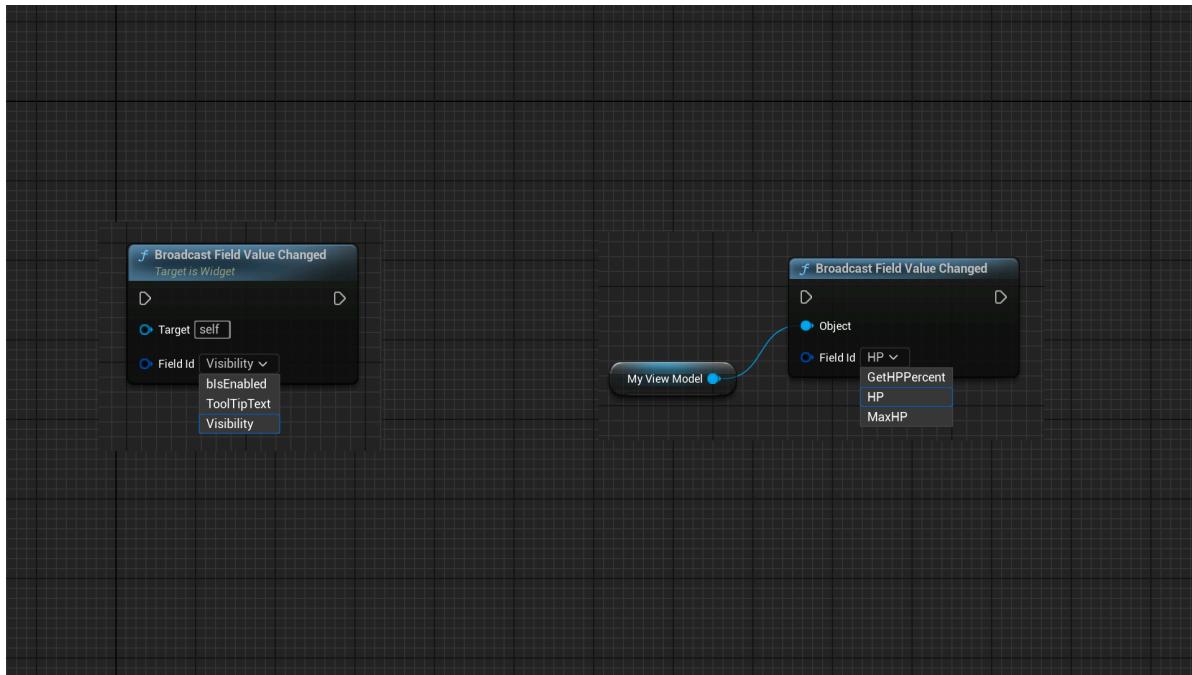
该参数为之后的FFieldNotificationId参数的提供上下文信息，这样FieldId的选项框才知道有哪些可选值。

源码例子：

```
/** Broadcast that the Field value changed. */
UFUNCTION(BlueprintCallable, Category = "FieldNotification", meta =
(FieldNotifyInterfaceParam="Object", DisplayName = "Broadcast Field value
Changed"))
    static void BroadcastFieldValueChanged(UObject* Object, FFieldNotificationID
FieldID);
```

蓝图效果：

在UserWidget里测试，可见没有连接到参数的Target默认为当前的UserWidget，则FieldId是3个值。而连接到我们自定义的ViewModel后，则改变为我们下面定义的值。



原理：

```

TSharedRef<SWidget> SFieldNotificationGraphPin::GetDefaultValueWidget()
{
    UEdGraphPin* SelfPin = GraphPinObj->GetOwningNode()-
>FindPin(UEdGraphSchema_K2::PSC_self);
    if (UK2Node_CallFunction* CallFunction = Cast<UK2Node_CallFunction>
(GraphPinObj->GetOwningNode()))
    {
        if (UFunction* Function = CallFunction->GetTargetFunction())
        {
            const FString& PinName = Function-
>GetMetaData("FieldNotifyInterfaceParam");
            if (PinName.Len() != 0)
            {
                SelfPin = GraphPinObj->GetOwningNode()->FindPin(*PinName);
            }
        }
    }

    return SNew(SFieldNotificationPicker)
        .value(this, &SFieldNotificationGraphPin::GetValue)
        .onValueChanged(this, &SFieldNotificationGraphPin::SetValue)
        .fromClass_Static(Private::GetPinClass, SelfPin)
        .visibility(this, &SGraphPin::GetDefaultValueVisibility);
}

```

IncludePath

- **功能描述：**记录 UClass 的引用路径
- **使用位置：** UClass
- **引擎模块：** UHT
- **元数据类型：** string="abc"

- **限制类型:** UClass上的信息

- **常用程度:** 0

记录UClass的引用路径。

开发者一般也不用管这个值。

有一个作用是在UHT生成.gen.cpp的时候，在头文件部分方便引用到该类的头文件。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Template :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};
```

其类型信息：

```
[class MyProperty_Template Class->Struct->Field->Object
/Script/Insider.MyProperty_Template]
(BlueprintType = true, IncludePath = Property/MyProperty_Template.h,
ModuleRelativePath = Property/MyProperty_Template.h)
ObjectFlags: RF_Public | RF_Standalone | RF_Transient
Outer: Package /Script/Insider
ClassHierarchy: MyProperty_Template:Object
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size: 56
Within: Object
ClassConfigName: Engine
{
    (Category = MyProperty_Template, ModuleRelativePath =
Property/MyProperty_Template.h)
    48-[4] int32 MyProperty;
    PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    Path: IntProperty /Script/Insider.MyProperty_Template:MyProperty
    [func MyFunc Function->Struct->Field->Object
/Script/Insider.MyProperty_Template:MyFunc]
    (ModuleRelativePath = Property/MyProperty_Template.h)
    ObjectFlags: RF_Public | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public | 
FUNC_BlueprintCallable
    NumParms: 2
    ParmSize: 20
```

```

ReturnValueOffset: 16
RPCId: 0
RPCResponseId: 0
public int32 MyFunc(FString str)final;
{
    0-[16] FString str;
    PropertyFlags: CPF_Parm | CPF_ZeroConstructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: StrProperty /Script/Insider.MyProperty_Template:MyFunc:str
    16-[4] int32 ReturnValue;
    PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |
    CPF_ReturnParm | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
    | CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: IntProperty
/Script/Insider.MyProperty_Template:MyFunc:ReturnValue
};
}

```

原理：

同样也是在UHT中分析后添加的。具体的逻辑值请见ModuleRelativePath中的原理代码部分。

```

protected override void UhtClass::Resolvesuper(UhtResolvePhase resolvePhase)
{
    switch (classType)
    {
        case UhtClassType::Class:
        {
            MetaData.Add(UhtNames::IncludePath,
HeaderFile.IncludeFilePath);
        }
    }
}

```

ModuleRelativePath

- 功能描述:** 记录类型定义的头文件路径，为其处于模块的内部相对路径。
- 使用位置:** Any
- 引擎模块:** UHT
- 元数据类型:** string="abc"
- 常用程度:** 0

记录当前元类型定义的头文件路径，为相对模块的相对路径。

对于开发者来说一般不用管，但是引擎编辑器会用它来定位某个类型是在哪个.h里定义的，从而在你双击类型的时候，可以为你在VS里打开相应的头文件。具体的逻辑可以去FSourceCodeNavigation里查看。

和IncludePath的区别是，ModuleRelativePath 在各种类型信息上都有，而IncludePath只用于UCLASS上。另外ModuleRelativePath 的值可以包含“Classes/Public/Internal/Private”这4个以开头，我们一般也确实会建议把.h.cpp划分到这4个文件夹里。而IncludeFilePath 的值就会去掉这个头。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Template :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};
```

其元类型信息打印：

可以发现ModuleRelativePath 在类，属性和函数上都有该信息。

而IncludePath只有在UCLASS上才有。

```
[class MyProperty_Template Class->Struct->Field->Object
/Script/Insider.MyProperty_Template]
(BlueprintType = true, IncludePath = Property/MyProperty_Template.h,
ModuleRelativePath = Property/MyProperty_Template.h)
ObjectFlags: RF_Public | RF_Standalone | RF_Transient
Outer: Package /Script/Insider
ClassHierarchy: MyProperty_Template:Object
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size: 56
within: Object
ClassConfigName: Engine
{
    (Category = MyProperty_Template, ModuleRelativePath =
Property/MyProperty_Template.h)
    48-[4] int32 MyProperty;
    PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    Path: IntProperty /Script/Insider.MyProperty_Template:MyProperty
    [func MyFunc Function->Struct->Field->Object
/Script/Insider.MyProperty_Template:MyFunc]
    (ModuleRelativePath = Property/MyProperty_Template.h)
    ObjectFlags: RF_Public | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public | 
FUNC_BlueprintCallable
    NumParms: 2
    ParmSize: 20
```

```

ReturnValueOffset: 16
RPCId: 0
RPCResponseId: 0
public int32 MyFunc(FString str)final;
{
    0-[16] FString str;
    PropertyFlags: CPF_Parm | CPF_ZeroConstructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: StrProperty /Script/Insider.MyProperty_Template:MyFunc:str
    16-[4] int32 ReturnValue;
    PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |
    CPF_ReturnParm | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
    | CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: IntProperty
/Script/Insider.MyProperty_Template:MyFunc:ReturnValue
};
}

```

原理：

在UHT分析的时候，自动的为类型加上头文件的路径信息。

从源码逻辑可以看出，ModuleRelativePath 的值可以包含“Classes/Public/Internal/Private”这4个以开头，我们一般也确实会建议把.h.cpp划分到这4个文件夹里。而IncludeFilePath 的值就会去掉这个头。

```

public enum UhtHeaderFileType
{
    /// <summary>
    /// Classes folder
    /// </summary>
    Classes,

    /// <summary>
    /// Public folder
    /// </summary>
    Public,

    /// <summary>
    /// Internal folder
    /// </summary>
    Internal,

    /// <summary>
    /// Private folder
    /// </summary>
    Private,
}

```

```

public static void AddModuleRelativePathToMetaData(UhtMetaData metaData,
UhtHeaderFile headerFile)
{
    metaData.Add(UhtNames.ModuleRelativePath, headerFile.ModuleRelativeFilePath);
}

//分析文件路径
private void StepPrepareHeaders(UhtPackage package, IEnumerable<string>
headerFiles, UhtHeaderFileType headerFileType)
{
    string typeDirectory = headerFileType.ToString() + '/';

    headerFile.ModuleRelativeFilePath = normalizedFullPath[stripLength..];
    if (normalizedFullPath[stripLength..].StartsWith(typeDirectory, true,
null))
    {
        stripLength += typeDirectory.Length;
    }
    headerFile.IncludeFilePath = normalizedFullPath[stripLength..];
}

```

NativeConstTemplateArg

- 功能描述:** 指定该属性是一个const的模板参数。
- 使用位置:** UPROPERTY
- 引擎模块:** UHT
- 元数据类型:** bool
- 常用程度:** 0

指定该属性是一个const的模板参数。

在源码里并没有找到使用的地方。只有在UHT中用到。

在UHT中查看主要是UhtArrayProperty和UhtObjectPropertyBase, UhtOptionalProperty。

BindWidget

- 功能描述:** 指定在C++类中该Widget属性一定要绑定到UMG的某个同名控件。
- 使用位置:** UPROPERTY
- 引擎模块:** Widget Property
- 元数据类型:** bool
- 限制类型:** UUserWidget子类里属性
- 关联项:** BindWidgetOptional, OptionalWidget
- 常用程度:** ★★★★☆

指定在C++类中该Widget属性一定要绑定到UMG的某个同名控件。

一种平常通用的编程范式是在C++中定义一个UUserWidget子类，然后再在UMG中继承于这个C++类，这样就能把一些逻辑放在C++中实现，而在UMG中排布控件。这个时候常常就会有个需求：需要在C++中用属性引用到UMG中定义的具体控件。

- 在C++里常见的作法是用WidgetTree->FindWidget来通过控件名字查找。但如果类里定义有几十个控件，——这么做就很繁琐。
- 因此更便利的方式是在C++里定义同名的控件属性，这样就会自动的关联起来，UMG蓝图对象在创建后引擎会自动的给C++中的Widget属性自动赋值到同名的控件。
- 必须要指出：BindWidget只是用作UMG编辑器的编辑和编译提示，让你记得要——把名字关联上。在C++里定义的该属性，要记得在UMG里也创建同名控件。在UMG中创建或更改的控件名字时，知道在C++中有一个同名属性来关联接收，就不会报错，否则会提示和C++定义的名字冲突。
- 总结BindWidget的作用有二：一是提醒UMG一定要相应的创建同名控件，否则编译抱错误。二是在定义同C++里属性同名的控件的时候，让UMG不会报错。
- 用法建议是为所有你想要绑定的同名属性都显式的加上BindWidget，不要依赖含糊默认的自动同名绑定机制。

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_NotFound;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_SameName;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidget))
    class UTextBlock* MyTextBlock_Bind;
};

void UMyProperty_Bindwidget::RunTest()
{
    //C++里查找Widget的方式
    UTextBlock* bindwidget= WidgetTree->FindWidget(TEXT("MyTextBlock_Bind"));
    check(bindwidget==MyTextBlock_Bind);
}

```

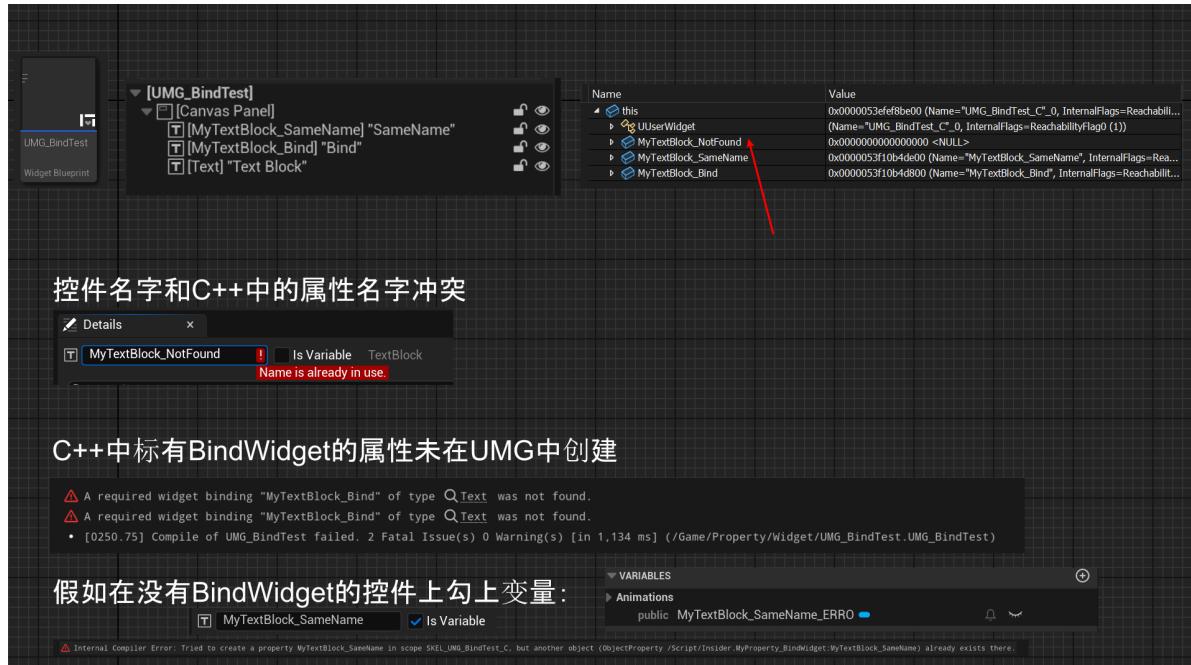
测试效果：

测试操作是在C++中定义如上图的UUserWidget基类，然后在UMG中创建蓝图子类。控件的列表如下图所示。

- 为了对比验证，分别在C++和蓝图中定义3个控件，有同名的和非同名的。然后在CreateWidet后在C++中调试查看这3个属性值。
- 可以发现MyTextBlock_Bind和MyTextBlock_SameName都自动的关联上了值，发现关联属性值的逻辑其跟有没有标上BindWidget并没有关系。但是如果在MyTextBlock_SameName勾上变量，也会报名字冲突的错。这是因为勾上变量，会在蓝图中创建一个属性，这样自然就和C++里的冲突。而没有勾上变量的时候，MyTextBlock_SameName本质上只是一个在WidgetTree下的对象，编辑器可以提示同名冲突（C++里先定义然后UMG里再定义），也可以选择不提示（BindWidget的作用了其实）。但如果是也要相应创建BP里的MyTextBlock_SameName变量，这个冲突就是必然存在了。这个时候如果没有加上BindWidget，引擎就会认为这是两个独立的不同的属性（假如你在C++里明明没写BindWidget而引擎自作主张给你BindWidget了，实际可能反而出现更多莫名其妙的错误）。

误）。只有显式的加上BindWidget，这个时候我们为MyTextBlock_Bind勾上变量，引擎知道C++里已经有个C++属性了，就没必要再创建一个蓝图属性了（这个时候BP面板里没有）。

- MyTextBlock_NotFound并没有值，这很符合逻辑，因为我们也没有在UMG中定义该控件。但是值得注意的是假如我们尝试在UMG中定义该名字的控件，会报错提示名字已经被占用。也很正常，因为这就像C++类的子类里定义成员变量，肯定不能出现成员变量冲突。但假如我们定义MyTextBlock_Bind就不会报这个“名字占用”的错，因为引擎知道C++里有一个同名属性是要用来引用该控件。因此这才是BindWidget的精确作用含义，只是作为提示。这个时候可能有人会问那我的UMG里的MyTextBlock_SameName是怎么创建上去的？不是会报错吗？答案是先在UMG里定义好，然后再在C++里定义，这样就不会报错了。
- 假如最后MyTextBlock_Bind没有在UMG中定义，那么UMG在编译的时候会报想要绑定的控件找不到，提醒你自己说想要BindWidget结果你又不创建。



原理：

判断一个属性是否是BindWidget的函数是IsBindWidgetProperty这个函数。

在控件改名或编译的时候，用来判断是否要生成错误提示的操作在FinishCompilingClass，大致逻辑就是根据IsBindWidgetProperty判断该控件是否想要绑定，然后根据当前情况，生成提示。

而因为同名而自动关联值的逻辑操作在UWidgetBlueprintGeneratedClass::InitializeWidgetStatic，逻辑其实是遍历WidgetTree下的控件，根据其名字去C++中查找，如果找到就自动的赋值。

```
void UWidgetBlueprintGeneratedClass::InitializeWidgetStatic()
{
    // Find property with the same name as the template and assign the new widget
    // to it.
    if (FObjectPropertyBase** PropPtr = ObjectPropertiesMap.Find(Widget-
>GetFName()))
    {
        FObjectPropertyBase* Prop = *PropPtr;
        check(Prop);
        Prop->SetObjectPropertyValue_InContainer(UserWidget, Widget);
        UObject* Value = Prop->GetObjectPropertyValue_InContainer(UserWidget);
        check(Value == Widget);
    }
}
```

```

}

void FWidgetBlueprintCompilerContext::FinishCompilingClass(UClass* Class)
{
    // Check that all BindWidget properties are present and of the appropriate
    // type
    for (TObjectPropertyBase<UWidget*>* WidgetProperty :
        TFieldRange<TObjectPropertyBase<UWidget*>>(ParentClass))
    {
        bool bIsOptional = false;

        if (FWidgetBlueprintEditorUtils::IsBindWidgetProperty(WidgetProperty,
            bIsOptional))
        {}
    }

}

bool FWidgetBlueprintEditorUtils::IsBindWidgetProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if (InProperty)
    {
        bool bIsBindWidget = InProperty->HasMetaData("BindWidget") || InProperty-
            >HasMetaData("BindWidgetOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetOptional") || (
            InProperty->HasMetaData("OptionalWidget") || InProperty-
            >GetBoolMetaData("OptionalWidget"));
    }

    return bIsBindWidget;
}

return false;
}

```

BindWidgetAnim

- 功能描述:** 指定在C++类中该UWidgetAnimation属性一定要绑定到UMG下的某个动画
- 使用位置:** UPROPERTY
- 引擎模块:** Widget Property
- 元数据类型:** bool
- 限制类型:** UWidget子类里UWidgetAnimation属性
- 关联项:** BindWidgetAnimOptional
- 常用程度:** ★★★★☆

指定在C++类中该UWidgetAnimation属性一定要绑定到UMG下的某个动画。

作用同BindWidget类似，都是用来把C++的属性和BP里的控件或动画赋值绑定起来。但又有一些区别：

- UWidgetAnimation和Widget不同，Widget的属性和控件只要同名就可以自动绑定起来，而UWidgetAnimation就不允许不加BindWidgetAnim而同名，否则会名字冲突报错。这是由于UMG里创建的Widget默认是不创建BP变量的，子控件只是WidgetTree下的一个对象，但是动画是默认会创建BP变量的。因此即使是UMG里先定义动画，然后C++里再定义同名属性，也是会过不了编译

的。

- UWidgetAnimation属性必须得是Transient，否则也会报错。我想这是因为UWidgetAnimation自然会在BP里作为子对象序列化，而不需要在C++序列化的时候访问到该属性，因此强制Transient以免不小心序列化它。另外UWidgetAnimation只是用作表现，因此其实也会自动的加上CPF_RepSkip，跳过网络复制。

测试代码：

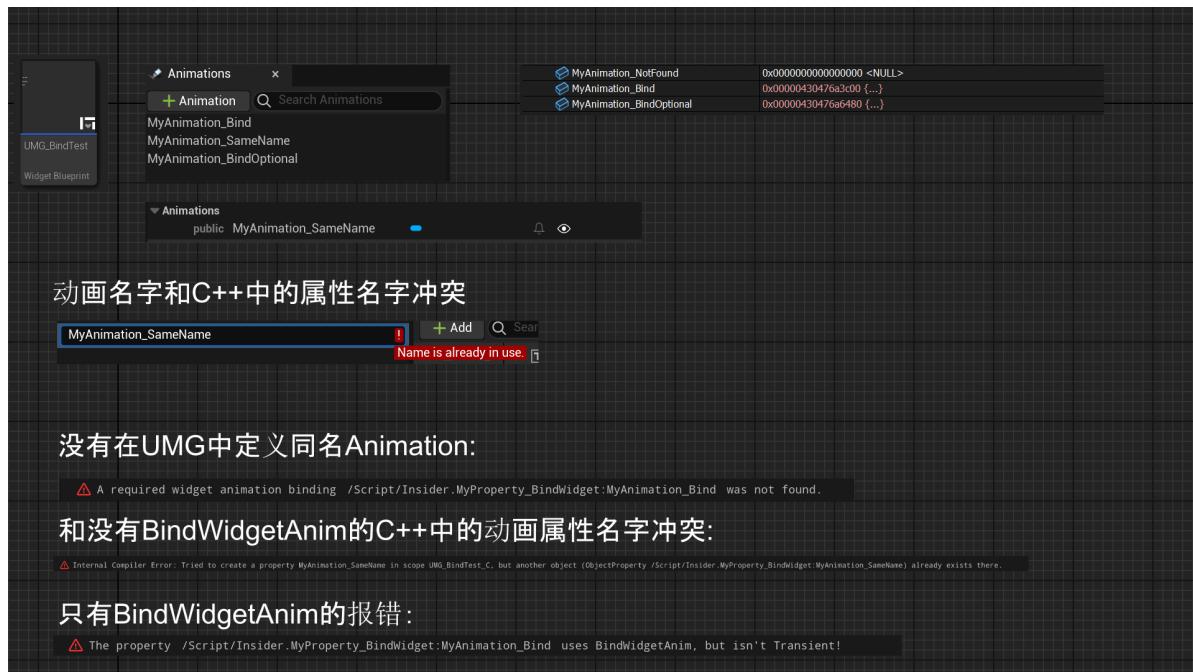
```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_BindWidget :public UUserWidget
{
    GENERATED_BODY()
    UMyProperty_BindWidget(const FObjectInitializer& ObjectInitializer);

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UWidgetAnimation* MyAnimation_NotFound;
    //UPROPERTY(EditAnywhere, BlueprintReadWrite)
    //class UWidgetAnimation* MyAnimation_SameName;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient, meta =
    (BindWidgetAnim))
    class UWidgetAnimation* MyAnimation_Bind;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient, meta =
    (BindWidgetAnimOptional))
    class UWidgetAnimation* MyAnimation_BindOptional;
};
```

测试效果：

测试过程和BindWidget类似，在C++和UMG中定义不同类型属性和动画对象。可以根据VS里实际对象的值发现：

- MyAnimation_Bind和MyAnimation_BindOptional都自动的绑定了正确的动画对象。
- 没有加BindWidgetAnim的MyAnimation_SameName必须注释掉，否则会和UMG里的MyAnimation_SameName名字冲突。
- 再提一下，不能像Widget里一样先UMG里定义动画，然后再C++定义同名属性，因为WidgetAnimation是一定会创建BP变量的，这是关键不同。



没有在UMG中定义同名Animation:

`A required widget animation binding /Script/Insider.MyProperty_BindWidget:MyAnimation_Bind was not found.`

和没有BindWidgetAnim的C++中的动画属性名字冲突:

`Internal Compiler Error: Tried to create a property MyAnimation_SameName in scope UMG_BindTest_C, but another object (ObjectProperty /Script/Insider.MyProperty_BindWidget:MyAnimation_SameName) already exists there.`

只有BindWidgetAnim的报错:

`The property /Script/Insider.MyProperty_BindWidget:MyAnimation_Bind uses BindWidgetAnim, but isn't Transient!`

原理:

大致逻辑和BindWidget类似，都是判断属性是否BindWidgetAnim。然后相应的在编译和改名的时候判断。

关于动画变量设置PropertyFlags的逻辑在CreateClassVariablesFromBlueprint里，可以看见加上了4个属性，明确了不要序列化该属性。

而为UWidgetAnimation*属性自动绑定赋值的逻辑在BindAnimationsStatic，一眼就懂。

```
bool FwidgetBlueprintEditorUtils::IsBindWidgetAnimProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if (InProperty)
    {
        bool bIsBindWidgetAnim = InProperty->HasMetaData("BindWidgetAnim") ||
        InProperty->HasMetaData("BindWidgetAnimOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetAnimOptional");

        return bIsBindWidgetAnim;
    }

    return false;
}

void FwidgetBlueprintCompilerContext::CreateClassVariablesFromBlueprint()
{
    for (UWidgetAnimation* Animation : WidgetBP->Animations)
    {
        FEdGraphPinType WidgetPinType(UEdGraphSchema_K2::PC_Object, NAME_None,
        Animation->GetClass(), EPinContainerType::None, true, FEdGraphTerminalType());
        FProperty* AnimationProperty = CreateVariable(Animation->GetFName(),
        WidgetPinType);

        if (AnimationProperty != nullptr)
        {
            const FString DisplayName = Animation->GetDisplayName().ToString();
        }
    }
}
```

```

        AnimationProperty->SetMetaData(TEXT("DisplayName"), *DisplayName);

        AnimationProperty->SetMetaData(TEXT("Category"), TEXT("Animations"));

        AnimationProperty->SetPropertyFlags(CPF_Transient);
        AnimationProperty->SetPropertyFlags(CPF_BlueprintVisible);
        AnimationProperty->SetPropertyFlags(CPF_BlueprintReadOnly);
        AnimationProperty->SetPropertyFlags(CPF_Repskip);

        WidgetAnimToMemberVariableMap.Add(Animation, AnimationProperty);
    }
}

void FWidgetBlueprintCompilerContext::FinishCompilingClass(UClass* Class)
{
    if (!WidgetAnimProperty->HasAnyPropertyFlags(CPF_Transient))
    {
        const FText BindWidgetAnimTransientError =
LOCTEXT("BindWidgetAnimTransient", "The property @@ uses BindWidgetAnim, but isn't Transient!");
        MessageLog.Error(*BindWidgetAnimTransientError.ToString(),
WidgetAnimProperty);
    }
}

void UWidgetBlueprintGeneratedClass::BindAnimationsStatic(UUserWidget* Instance,
const TArrayView<UWidgetAnimation*> InAnimations, const TMap< FName,
FObjectPropertyBase*>& InPropertyMap)
{
    // Note: It's not safe to assume here that the UserWidget class type is a UWidgetBlueprintGeneratedClass!
    // - @see InitializeWidgetStatic()

    for (UWidgetAnimation* Animation : InAnimations)
    {
        if (Animation->GetMovieScene())
        {
            // Find property with the same name as the animation and assign the animation to it.
            if (FObjectPropertyBase* const* PropPtr =
InPropertyMap.Find(Animation->GetMovieScene()->GetFName())))
            {
                check(*PropPtr);
                (*PropPtr)->SetObjectPropertyValue_InContainer(Instance,
Animation);
            }
        }
    }
}

```

BindWidgetAnimOptional

- 功能描述:** 指定在C++类中该UWidgetAnimation属性可以要绑定到UMG下的某个动画，也可以不绑定。

- **使用位置:** UPROPERTY
- **引擎模块:** Widget Property
- **元数据类型:** bool
- **限制类型:** UWidget子类里UWidgetAnimation属性
- **关联项:** BindWidgetAnim
- **常用程度:** ★★★

同BindWidgetOptional作用也类似，在不绑定的时候在编译结果里会有一个提示，而不是像BindWidget一样强制的错误。

- An optional widget animation binding /Script/Insider.MyProperty_BindWidget:MyAnimation_BindOptional2 is available.

自然的也说过不能像Widget一样，不加BindWidget就自动默认绑定。

因此用法上要嘛加BindWidgetAnim，要嘛加BindWidgetAnimOptional。

BindWidgetOptional

- **功能描述:** 指定在C++类中该Widget属性可以绑定到UMG的某个同名控件，也可以不绑定。
- **使用位置:** UPROPERTY
- **引擎模块:** Widget Property
- **元数据类型:** bool
- **限制类型:** UWidget子类里属性
- **关联项:** BindWidget
- **常用程度:** ★★★

指定在C++类中该Widget属性可以绑定到UMG的某个同名控件，也可以不绑定。

大致作用和BindWidget一样，区别是：

- BindWidgetOptional顾名思义是可选的，意思是UMG里即使不定义该控件在编译的时候也不会报错。编译会通过，但是会提示警告缺少控件。
-
- 和不加BindWidgetOptional的控件同名属性的区别是，前者在UMG里定义同名控件的时候不会报错，但后者是会提示同名冲突报错。

BindWidgetOptional的写法有两种：

BindWidgetOptional可以看作是BindWidget和OptionalWidget的合并版。

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_BindWidget :public UUserWidget
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_SameName;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidgetOptional))
    class UTextBlock* MyTextBlock_Optional1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidget,
OptionalWidget))
    class UTextBlock* MyTextBlock_Optional2;
};

```

测试效果：

- An optional widget binding "MyTextBlock_Optional2" of type `QText` is available.

原理：

```

bool FWidgetBlueprintEditorUtils::IsBindWidgetProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if ( InProperty )
    {
        bool bIsBindWidget = InProperty->HasMetaData("BindWidget") || InProperty->HasMetaData("BindWidgetOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetOptional") || (InProperty->HasMetaData("OptionalWidget") || InProperty->GetBoolMetaData("OptionalWidget") );

        return bIsBindWidget;
    }

    return false;
}

```

DefaultGraphNode

- 功能描述：**标记引擎默认创建的蓝图节点。
- 使用位置：**UCLASS
- 引擎模块：**Widget Property
- 元数据类型：**bool
- 常用程度：**0

标记引擎默认创建的蓝图节点。

这样就可以在判断蓝图内是否有用户手动创建的节点时，过滤掉引擎自动创建的那些。

只在内部使用，用户不需要自己使用。

原理：

```
static bool BlueprintEditorImpl::GraphHasUserPlacedNodes(UEdGraph const* InGraph)
{
    bool bHasUserPlacedNodes = false;

    for (UEdGraphNode const* Node : InGraph->Nodes)
    {
        if (Node == nullptr)
        {
            continue;
        }

        if (!Node->GetOutermost()->GetMetaData()->HasValue(Node,
FNodeMetadata::DefaultGraphNode))
        {
            bHasUserPlacedNodes = true;
            break;
        }
    }

    return bHasUserPlacedNodes;
}
```

DesignerRebuild

- **功能描述：**指定Widget里的某个属性值改变后应该重新刷新UMG的预览界面。
- **使用位置：** UPROPERTY
- **引擎模块：** Widget Property
- **元数据类型：** bool
- **限制类型：** UWidget子类里属性
- **常用程度：** ★

指定Widget里的某个属性值改变后应该重新刷新UMG的预览界面。

首先想到的问题是，哪种属性需要用到该DesignerRebuild标记？

这个属性很少需要用到，一般Widget里的属性在更新后也只需要更新自己的显示，不需要刷新整个界面，比如字号。需要用到的情况想来有二：

1. 一些属性的改变会大大的改变其控件样式，当然也可以做到精细化的只重绘自己，但干脆整个预览界面刷新一下得了，反正是编辑器环境。比如UTextBlock 的bSimpleTextMode，和UListViewBase 下的EntryWidgetClass，都会大大的改变自己。
2. 一些属性可能影响到整个界面别的东西的时候，这个时候也时候干脆全部刷新一下。没找到恰当的例子，但如果用户自己的控件有这个需求，就可以标上。

源码里的例子是：

```
UCLASS(meta=(DisplayName="Text"), MinimalAPI)
```

```

class UTextBlock : public UTextLayoutWidget
{
    /**
     * If this is enabled, text shaping, wrapping, justification are disabled in
     favor of much faster text layout and measurement.
     * This feature is only suitable for "simple" text (ie, text containing only
     numbers or basic ASCII) as it disables the complex text rendering support
     required for certain languages (such as Arabic and Thai).
     * It is significantly faster for text that can take advantage of it
     (particularly if that text changes frequently), but shouldn't be used for
     localized user-facing text.
    */
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Performance,
AdvancedDisplay, meta=(AllowPrivateAccess = "true", DesignerRebuild))
    bool bSimpleTextMode;
}

UCLASS(Abstract, NotBlueprintable, hidedropdown, meta = (EntryInterface =
UserListEntry), MinimalAPI)
class UListViewBase : public UWidget
{
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = ListEntries, meta =
(DesignerRebuild, AllowPrivateAccess = true, MustImplement =
"/script/UMG.UserListEntry"))
    TSubclassOf<UUserWidget> EntrywidgetClass;
}

```

UTextBlock的测试效果：

可以发现在改变bSimpleTextMode的时候，左侧预览界面会一下下的在跳动刷新。而在点击改变别的按钮的时候就没有该效果。



测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
public:
    UPROPERTY(EditAnywhere, Category = Design)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, Category = Design, meta = (DesignerRebuild))
    int32 MyInt_DesignerRebuild = 123;
}

```

测试效果：

可见在改变普通的属性MyInt 的时候，界面并不会刷新。而在改变MyInt_DesignerRebuild 的时候，界面左上角的数字在跳动（虽然整个界面其实并没有什么实质变化）。



原理：

在Widget里的带有DesignerRebuild的某属性改变之后，会通知InvalidatePreview以便更新编辑器里的预览窗口。

```
void SWidgetDetailsView::NotifyPostChange(const FPropertyChangedEvent&
                                         PropertyChangedEvent, FEditPropertyChain* PropertyThatChanged)
{
    const static FName DesignerRebuildName("DesignerRebuild");

    //...
    // If the property that changed is marked as "DesignerRebuild" we invalidate
    // the preview.
    if ( PropertyChangedEvent.Property->HasMetaData(DesignerRebuildName) ||
         PropertyThatChanged->GetActiveMemberNode()->GetValue()-
         >HasMetaData(DesignerRebuildName) )
    {
        const bool bViewOnly = true;
        BlueprintEditor.Pin()->InvalidatePreview(bViewOnly);
    }
}
```

DisableNativeTick

- **功能描述：**禁用该UserWidget的NativeTick。
- **使用位置：** UCLASS
- **引擎模块：** Widget Property
- **元数据类型：** bool
- **限制类型：** UserWidget的子类
- **常用程度：** ★★★

禁用该UserWidget的NativeTick。

如果只有C++类则不起作用，因为纯C++的Widget没有WidgetBPClass。

而且BP的子类要删除Tick蓝图节点。

测试代码：

```
UCLASS(BlueprintType, meta=())
class INSIDER_API UMyWidget_WithNativeTick :public UUserWidget
{
    GENERATED_BODY()
public:
    virtual void NativeTick(const FGeometry& MyGeometry, float InDeltaTime)
override
    {
        Super::NativeTick(MyGeometry, InDeltaTime);
        UKismetSystemLibrary::PrintString(nullptr, TEXT("WithNativeTick"), true);
    }
};
```

```

UCLASS(BlueprintType, meta=(DisableNativeTick))
class INSIDER_API UMyWidget_DisableNativeTick :public UUserWidget
{
    GENERATED_BODY()

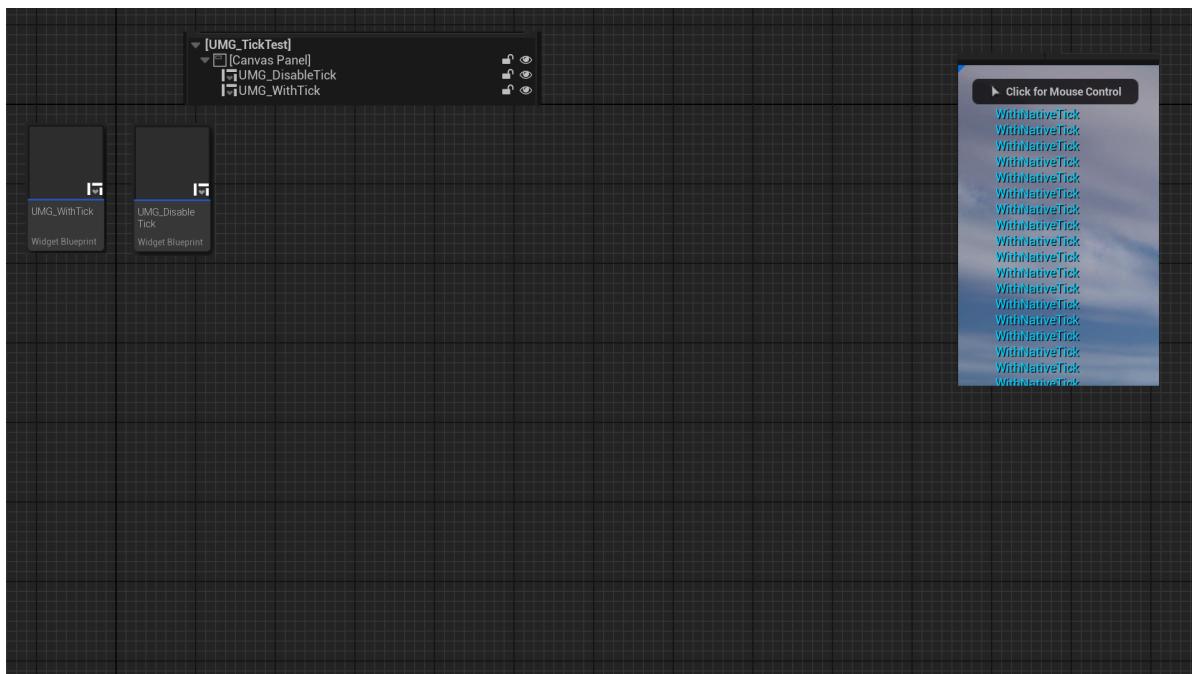
public:
    virtual void NativeTick(const FGeometry& MyGeometry, float InDeltaTime)
override
    {
        Super::NativeTick(MyGeometry, InDeltaTime);
        UKismetSystemLibrary::PrintString(nullptr, TEXT("DisableNativeTick")),
true);
    }
};

```

测试效果：

在蓝图中分别创建UMyWidget_WithNativeTick 和UMyWidget_DisableNativeTick 的子类 UMG_WithTick和UMG_DisableTick。然后把他们都添加到一个UMG里，添加到屏幕上后观察 NativeTick的调用情况。

可见只有WithNativeTick调用。



原理：

在UCLASS上标记会导致UMG蓝图的bClassRequiresNativeTick=false。继而在UUserWidget的 UpdateCanTick里判断。如果WidgetBPClass不为空（是蓝图子类）且ClassRequiresNativeTick为 false，bCanTick 才一开始为false。然后判断bHasScriptImplementedTick则要求蓝图中没有EventTick （默认会创建，自己要手动删掉）。然后后面继续判断要没有延迟蓝图节点，没有动画。总之就是这个 Widget要真的没有Tick的需求，则可以真的最后bCanTick=false。

```

void UwidgetBlueprint::UpdateTickabilityStats(bool& OutHasLatentActions, bool&
OutHasAnimations, bool& OutClassRequiresNativeTick)
{
    static const FName DisableNativeTickMetaTag("DisableNativeTick");
}

```

```

        const bool bClassRequiresNativeTick = !NativeParent-
>HasMetaData(DisableNativeTickMetaTag);
        OutClassRequiresNativeTick = bClassRequiresNativeTick;

    }

void FWidgetBlueprintCompilerContext::CopyTermDefaultsToDefaultObject(UObject*
DefaultObject)
{
    WidgetBP->UpdateTickabilityStats(bClassOrParentsHaveLatentActions,
bClassOrParentsHaveAnimations, bClassRequiresNativeTick);
    WidgetClass->SetClassRequiresNativeTick(bClassRequiresNativeTick);
}

void UUserWidget::UpdateCanTick()
{
    UWidgetBlueprintGeneratedClass* WidgetBPClass =
Cast<UWidgetBlueprintGeneratedClass>(GetClass());
    bCanTick |= !WidgetBPClass || WidgetBPClass->ClassRequiresNativeTick();
    bCanTick |= bHasScriptImplementedTick;
    bCanTick |= World->GetLatentActionManager().GetNumActionsForObject(this)
!= 0;
    bCanTick |= ActiveSequencePlayers.Num() > 0;
    bCanTick |= QueuedWidgetAnimationTransitions.Num() > 0;
    SafeGCWidget->SetCanTick(bCanTick);
}

```

EntryClass

- 功能描述:** 限定EntryWidgetClass属性上可选类必须继承自的基类，用在DynamicEntryBox和ListView这两个Widget上。
- 使用位置:** UCLASS, UPROPERTY
- 元数据类型:** string="abc"
- 限制类型:** UWidget子类
- 关联项:** EntryInterface
- 常用程度:** ★★★

EntryInterface

- 功能描述:** 限定EntryWidgetClass属性上可选类必须实现的接口，用在DynamicEntryBox和ListView这两个Widget上。
- 使用位置:** UCLASS, UPROPERTY
- 引擎模块:** Widget Property
- 元数据类型:** string="abc"
- 限制类型:** UWidget子类
- 关联项:** EntryClass
- 常用程度:** ★★★

限定EntryWidgetClass属性上可选类必须实现的接口，用在DynamicEntryBox和ListView这两个Widget上。

以ListView为例，术语Entry指的是列表中显示的子控件，而Item指的是列表背后的数据元素。比如列表背包有1000个元素(Item)，但是同时只能呈现10个控件(Entry)在界面上。

因此EntryInterface和EntryClass，顾名思义，指的是EntryWidget上要实现的接口和其基类。

用法展示，以下都用ListView举例，DynamicBox同理。

```
//1. ListView作为别的widget的属性，因此会在Property上进行Meta的提取判断。  
//该属性必须是BindWidget，才能自动绑定到UMG里的控件，同时作为C++ property才能被枚举到。  
class UMyUserWidget : public UUserWidget  
{  
    UPROPERTY(BindWidget, meta = (EntryClass =  
        MyListEntryWidget, EntryInterface = MyUserListEntry ))  
    UListViewBase* MyListView;  
}  
  
//2. 如果在Property上没有找到改Meta，也会尝试在widget class身上直接找  
UCLASS(meta = (EntryClass = MyListEntryWidget, EntryInterface =  
    "/Script/UMG.UserObjectListEntry"))  
class UMyListView : public UListViewBase, public ITypedUMGListView<UObject*>  
{  
  
//3. 之后在ClassPicker的时候，EntryClass指定其父类，EntryInterface指定类必须实现的接口
```

源码中的用法：

```
UCLASS(Abstract, NotBlueprintable, hidedropdown, meta = (EntryInterface =  
    UserListEntry), MinimalAPI)  
class UListViewBase : public UWidget  
{  
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = ListEntries, meta =  
        (DesignerRebuild, AllowPrivateAccess = true, MustImplement =  
            "/Script/UMG.UserListEntry"))  
    TSubclassOf<UUserWidget> EntryWidgetClass;  
}  
  
UCLASS(meta = (EntryInterface = "/Script/UMG.UserObjectListEntry"), MinimalAPI)  
class UListView : public UListViewBase, public ITypedUMGListView<UObject*>  
{}  
  
//其中UserObjectListEntry接口继承自UserListEntry，Entry Widget都得继承自该接口。  
UINTERFACE(MinimalAPI)  
class UUserObjectListEntry : public UUserListEntry  
{}  
  
SNew(SClassPropertyParams)  
.AllowNone(false)  
.IsBlueprintBaseOnly(true)  
.RequiredInterface(RequiredEntryInterface)  
.MetaClass(EntryBaseClass ? EntryBaseClass : UUserWidget::StaticClass())  
.SelectedClass(this, &FDynamicEntryWidgetDetailsBase::GetSelectedEntryClass)  
.OnSetClass(this, &FDynamicEntryWidgetDetailsBase::HandleNewEntryClassSelected)
```

在FDynamicEntryWidgetDetailsBase中判断EntryInterface和EntryClass，然后在SClassPropertyEntryBox中限定属性细节面板ClassPicker的可选类。
FDynamicEntryWidgetDetailsBase是FListViewBaseDetails和FDynamicEntryBoxDetails的基类，因此ListView和DynamicBox的属性细节面板都由它进行定制化。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyEntryWidget :public UUserWidget, public IUserObjectListEntry
{
    GENERATED_BODY()
public:
    virtual void NativeOnListItemObjectSet(UObject* ListItemObject) override;
public:
    UPROPERTY(meta = (BindWidget))
    class UTextBlock* ValueTextBlock;
};

////////////////////////////

UINTERFACE(MinimalAPI)
class UMyCustomListEntry : public IUserObjectListEntry
{
    GENERATED_UINTERFACE_BODY()
};

class IMyCustomListEntry : public IUserObjectListEntry
{
    GENERATED_IINTERFACE_BODY()
};

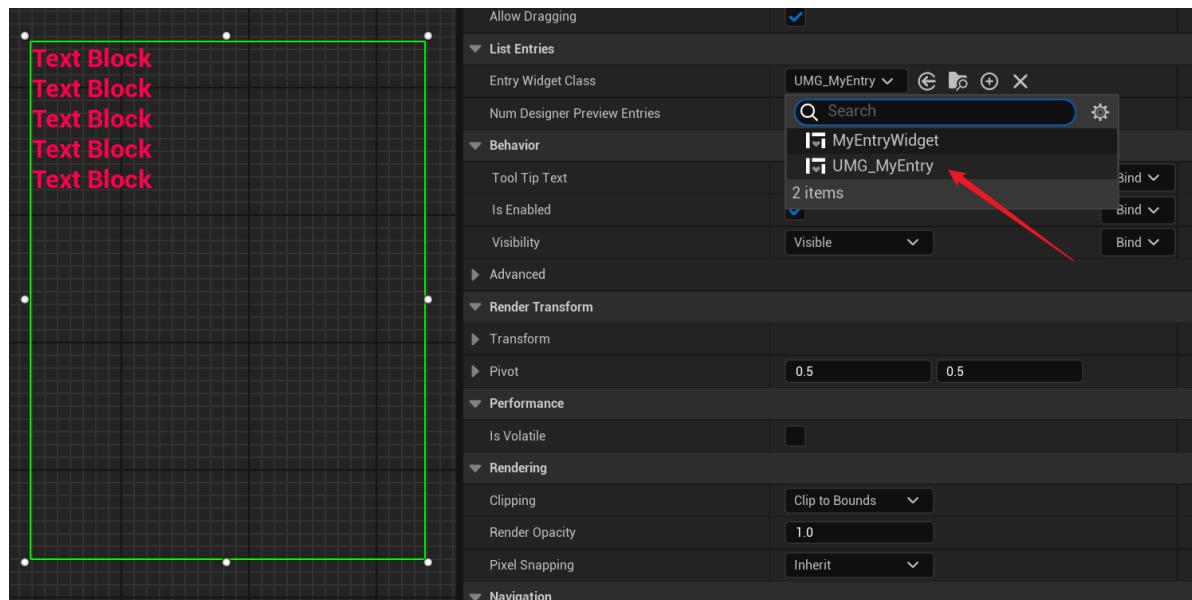
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCustomEntryWidget :public UUserWidget, public IMyCustomListEntry
{
    GENERATED_BODY()
public:
    virtual void NativeOnListItemObjectSet(UObject* ListItemObject) override;
public:
    UPROPERTY(meta = (BindWidget))
    class UTextBlock* ValueTextBlock;
};

////////////////////////////

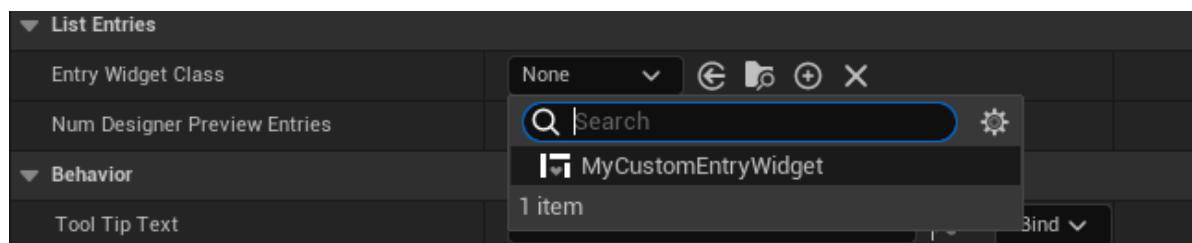
UCLASS()
class INSIDER_API UMyListContainerWidget :public UUserWidget
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (BindWidget, EntryClass = MyCustomEntryWidget, EntryInterface = MyCustomListEntry))
    class UListView* MyListView;
};
```

蓝图中的效果：

如果MyListView上没有指定EntryClass或EntryInterface，则在ListView的EntryWidgetClass属性上可以选择蓝图创建的UMG_MyEntry(继承自C++的UMyEntryWidget)。



如果如上面代码中所示，我们新创建一个接口为MyCustomListEntry，并且也新建一个新的MyCustomEntryWidget，然后在MyListView属性上指定EntryClass或EntryInterface（可以一起也可以单个），则ListView的EntryWidgetClass属性可选的类就被限制住了。



还有一种用法是当你想自定义一个ListView，可以选择继承自ListViewBase，然后在这个子类上直接限定EntryClass或EntryInterface，效果和上图是一样的。

```
UCLASS(meta = (EntryClass = MyCustomEntryWidget, EntryInterface = MyCustomListEntry))
class UMyListView : public UListViewBase, public ITypedUMGListView<UObject*>
{}
```

IsBindableEvent

- 功能描述：**把一个动态单播委托暴露到UMG蓝图里以绑定相应事件。
- 使用位置：**UPROPERTY
- 引擎模块：**Widget Property
- 元数据类型：**bool
- 限制类型：**UWidget子类里动态单播属性
- 常用程度：**★★★

把一个动态单播委托暴露到UMG蓝图里以绑定相应事件。

需要注意的点是：

- 必须是动态委托，就是DYNAMIC的那些，这样才可以在蓝图里序列化。
- 动态多播委托 (DECLARE_DYNAMIC_MULTICAST_DELEGATE) 默认就可以在UMG里绑定事件，因此没有必要加IsBindableEvent。往往也配合加上BlueprintAssignable以便也可以在蓝图里手动绑定。
- 动态单播委托 (DECLARE_DYNAMIC_DELEGATE) 默认是不在UMG里暴露的。但可以加上IsBindableEvent以便可以在其实例的细节面板上绑定。
- UMG里的控件事件为什么要有多播和单播？其实多播和单播除了数量不同以外，最大的不同是多播没有返回值。这个例子可以对比UButton下的OnClicked多播事件和UIImage下的OnMouseButtonEvent单播委托，前者是点击的事件，已经是个“结果”事件了，点击事件可能被多个地方响应，因此要设计成多播。而后的OnMouseButtonEvent是鼠标按下的事件，有一个重要的逻辑是会根据返回值FReply的不同而决定该事件是否继续路由上去，因此只能用单播，只能绑定一个。

源码例子：

```

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnButtonClickedEvent);
class UButton : public UContentWidget
{
    UPROPERTY(BlueprintAssignable, Category="Button|Event")
    FOnButtonClickedEvent OnClicked;
}

DECLARE_DYNAMIC_DELEGATE_RetVal_TwoParams(FReply, FOnPointerEvent,
FGeometry, MyGeometry, const FPointerEvent&, MouseEvent);
class UIImage : public UWidget
{
    UPROPERTY(EditAnywhere, Category=Events, meta=( IsBindableEvent="True" ))
    FOnPointerEvent OnMouseButtonEvent;
}

```

测试代码：

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
public:
    DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnMyClickedMulticastDelegate);

    UPROPERTY(EditAnywhere, BlueprintAssignable, Category = MyEvent)
    FOnMyClickedMulticastDelegate MyClickedMulticastDelegate;

public:

    DECLARE_DYNAMIC_DELEGATE_RetVal_OneParam(FString, FOnMyClickedDelegate, int32, MyValue);

    UPROPERTY(EditAnywhere, Category = MyEvent)
    FOnMyClickedDelegate MyClickedDelegate_Default;

    UPROPERTY(EditAnywhere, Category = MyEvent)
    FOnMyClickedDelegate MyClickedEvent;

```

```

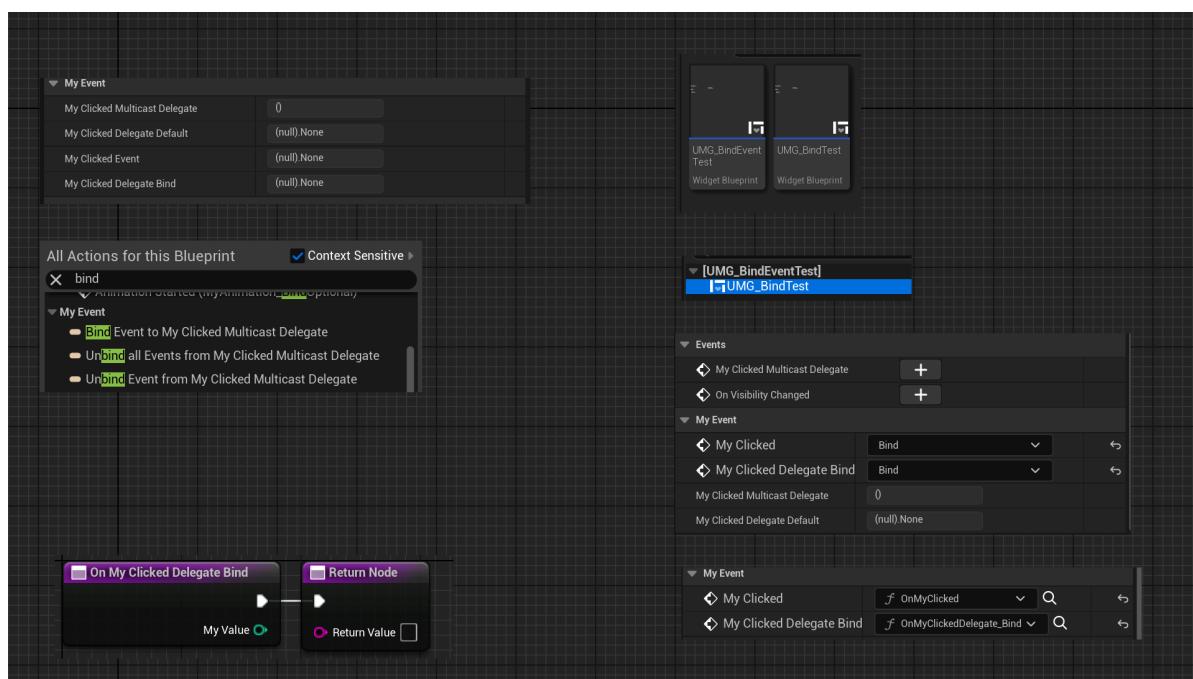
UPROPERTY(EditAnywhere, Category = MyEvent, meta = (IsBindableEvent =
"True"))
FOnMyClickedDelegate MyclickedDelegate_Bind;
}

```

测试结果：

操作步骤是在UMG_BindTest外再创建一个UMG，然后让UMG_BindTest成为子控件，然后观察其实例上的事件绑定，如下图右侧所示。

- 可以发现动态多播委托默认就会出现可以绑定的+定制化按钮，如MyClickedMulticastDelegate。
- 动态多播委托加上BlueprintAssignable（不能加在单播委托上）了之后，就可以在蓝图里绑定事件，如左下侧图。
- 加了IsBindableEvent 的MyClickedDelegate_Bind，可以看见出现了可以Bind的下拉按钮，绑定之后可以显示函数名字，也可以清除。
- 没有加IsBindableEvent 的MyClickedDelegate_Default就没有出现在可绑定的按钮，你只能在C++里自己绑定了。
- 没有加IsBindableEvent 的MyClickedEvent因为名字以Event结尾也出现了可绑定的按钮，这只能说是当前的一个潜规则。源码注释也说以后会去除。
- 另外这些委托我虽然都加上了EditAnywhere，但其实你也知道这并没办法编辑。



原理：

对于Widget的细节面板，引擎也定义了各种Customization。其中对应的就是FBlueprintWidgetCustomization。其针对绑定的部分的代码如下。

代码也很容易理解，动态多播委托默认都出现绑定，动态单播委托有加IsBindableEvent或者名字以Event结尾就也创建绑定按钮。

```

PropertyView->RegisterInstancedCustomPropertyParams(UWidget::StaticClass(),
FOnGetDetailCustomizationInstance::CreateStatic(&FBlueprintWidgetCustomization::MakeInstance, BlueprintEditorRef, BlueprintEditorRef->GetBlueprintObj()));

```

```

void BlueprintWidgetCustomization::PerformBindingCustomization(IDetailLayoutBuilder&
DetailLayout, const TArrayView<UWidget*> Widgets)
{
    static const FName IsBindableEventName(TEXT("IsBindableEvent"));

    bCreateMulticastEventCustomizationErrorAdded = false;
    if (Widgets.Num() == 1)
    {
        UWidget* Widget = Widgets[0];
        UClass* PropertyClass = Widget->GetClass();

        for (TFieldIterator<FProperty> PropertyIt(PropertyClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;

            if (FDelegateProperty* DelegateProperty =
CastField<FDelegateProperty>(*PropertyIt))
            {
                //TODO Remove the code to use ones that end with "Event". Prefer
metadata flag.
                if (DelegateProperty->HasMetaData(IsBindableEventName) ||
DelegateProperty->GetName().Endswith(TEXT("Event")))
                {
                    CreateEventCustomization(DetailLayout, DelegateProperty,
Widget);
                }
            }
            else if (FMulticastDelegateProperty* MulticastDelegateProperty =
CastField<FMulticastDelegateProperty>(Property))
            {
                CreateMulticastEventCustomization(DetailLayout, Widget-
>GetFName(), PropertyClass, MulticastDelegateProperty);
            }
        }
    }
}

```

OptionalWidget

- 功能描述:** 指定在C++类中该Widget属性可以绑定到UMG的某个同名控件，也可以不绑定。
- 使用位置:** UPROPERTY
- 引擎模块:** Widget Property
- 元数据类型:** bool
- 限制类型:** UWidget子类里属性
- 关联项:** BindWidget
- 常用程度:** ★★★

必须配合BindWidget使用。

BindWidget+OptionalWidget=BindWidgetOptional

ViewmodelBlueprintWidgetExtension

- **功能描述:** 用来验证InListItems的Object类型是否符合EntryWidgetClass的MVVM绑定的ViewModelProperty。
- **使用位置:** UFUNCTION
- **引擎模块:** Widget Property
- **元数据类型:** string="abc"
- **常用程度:** 0

用来验证InListItems的Object类型是否符合EntryWidgetClass的MVVM绑定的ViewModelProperty。

当前只在ListView里该函数使用。

原理:

```
UCLASS(meta = (EntryInterface = "/Script/UMG.UserObjectListEntry"), MinimalAPI)
class UListView : public UListViewBase, public ITypedUMGListView<UObject*>
{
    UFUNCTION(BlueprintCallable, Category = ListView, meta = (AllowPrivateAccess
= true, DisplayName = "Set List Items", ViewmodelBlueprintWidgetExtension =
"EntryViewModel"))
    UMG_API void BP_SetListItems(const TArray<UObject*>& InListItems);
}

void
UMVVMViewBlueprintListViewBaseExtension::Precompile(UE::MVVM::Compiler::IMVVMBlueprintViewPrecompile* Compiler, UWidgetBlueprintGeneratedClass* Class)
{}
```

Abstract

- **功能描述:** 指定此类为抽象基类。可被继承，但不可生成对象。
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中添加CLASS_Abstract
- **常用程度:** ★★★★★

指定此类为抽象基类。可被继承，但不可生成对象。

一般是用在XXXBase基类。

示例代码：

```
/*
    ClassFlags: CLASS_Abstract | CLASS_MatchedSerializers | CLASS_Native | 
    CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic | 
    CLASS_Constructed
*/
UCLASS(Blueprintable, abstract)
class INSIDER_API UMyClass_Abstract :public UObject
{
    GENERATED_BODY()
};

//测试语句：
UMyClass_Abstract* obj=NewObject<UMyClass_Abstract>();
```

示例效果：

在蓝图中的ConstructObject不会出现该类。同时在C++中NewObject也会报错。



原理：

在NewObject的时候会进行Abstract的判断。

```
bool StaticAllocateObjectErrorTests( const UClass* InClass, UObject* InOuter,
FName InName, EObjectFlags InFlags)
{
    // validation checks.
    if( !InClass )
    {
        UE_LOG(LogUObjectGlobals, Fatal, TEXT("Empty class for object %s"),
*InName.ToString() );
        return true;
    }

    // for abstract classes that are being loaded NOT in the editor we want to
    // error. If they are in the editor we do not want to have an error
    if (FScopedAllowAbstractClassAllocation::IsDisallowedAbstractClass(InClass,
InFlags))
    {
        if ( GIsEditor )
        {
            const FString ErrorMsg = FString::Printf(TEXT("Class which was marked
abstract was trying to be loaded in outer %. It will be nulled out on save. %s
%s"), *GetPathNameSafe(InOuter), *InName.ToString(), *InClass->GetName());
            // if we are trying instantiate an abstract class in the editor we'll
            // warn the user that it will be nulled out on save
            UE_LOG(LogUObjectGlobals, Warning, TEXT("%s"), *ErrorMsg);
            ensureMsgf(false, TEXT("%s"), *ErrorMsg);
        }
    }
}
```

```

        UE_LOG(LogUObjectGlobals, Fatal, TEXT("%s"), *FString::Printf(
TEXT("Can't create object %s in outer %s: class %s is abstract"),
*InName.ToString(), *GetPathNameSafe(Outer), *InClass->GetName()));
    return true;
}
}

bool FScopedAllowAbstractClassAllocation::IsDisallowedAbstractClass(const
UClass* InClass, EObjectFlags InFlags)
{
    if (((InFlags & RF_ClassDefaultObject) == 0) && InClass-
>HasAnyClassFlags(CLASS_Abstract))
    {
        if (AllowAbstractCount == 0)
        {
            return true;
        }
    }

    return false;
}

```

Blueprintable

- 功能描述:** 可以在蓝图里被继承，隐含的作用也可当变量类型
- 引擎模块:** Blueprint
- 元数据类型:** bool
- 作用机制:** 在Meta添加IsBlueprintBase和BlueprintType
- 关联项:** NotBlueprintable
- 常用程度:** ★★★★☆

可以在蓝图里被继承，隐含的作用也可当变量类型。

当设置Blueprintable标记的时候，会隐含的设置上BlueprintType = true的metadata。去除的时候，也会相应的去除掉BlueprintType = true。

示例代码：

```

/*
(BlueprintType = true, IncludePath = Class/MyClass_Blueprintable.h,
IsBlueprintBase = true, ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(Blueprintable)
class INSIDER_API UMyClass_Blueprintable :public UObject
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_Blueprintable.h, IsBlueprintBase = false,
ModuleRelativePath = Class/MyClass_Blueprintable.h)

```

```

*/
UCLASS(NotBlueprintable)
class INSIDER_API UMyClass_NotBlueprintable :public UObject
{
    GENERATED_BODY()
};

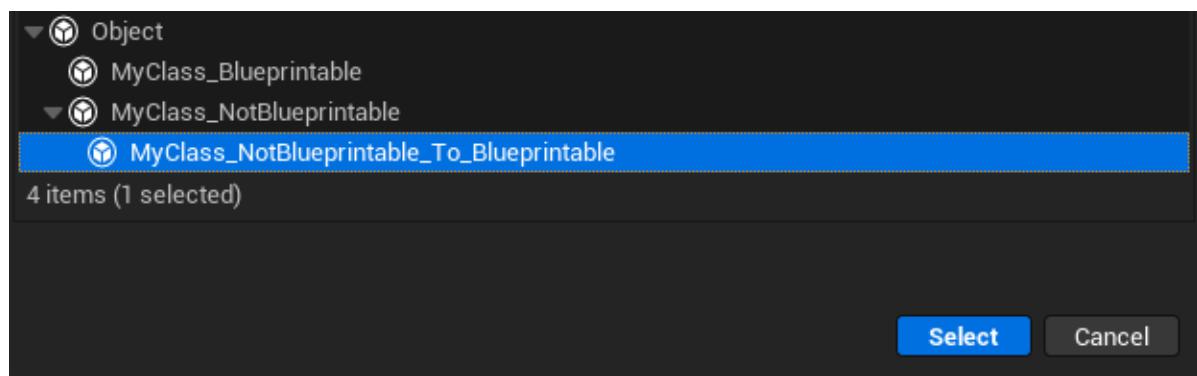
/*
(BlueprintType = true, IncludePath = Class/MyClass_Blueprintable.h,
IsBlueprintBase = true, ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(Blueprintable)
class INSIDER_API UMyClass_NotBlueprintable_To_Blueprintable :public UMyClass_NotBlueprintable
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_Blueprintable.h, IsBlueprintBase = false,
ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(NotBlueprintable)
class INSIDER_API UMyClass_Blueprintable_To_NotBlueprintable :public UMyClass_Blueprintable
{
    GENERATED_BODY()
};

```

示例效果：

只有带有Blueprintable才可以被选做基类。



不过是否能够当做变量的规则，还是会依赖父类的Blueprint标记。因此以下这3个都是可以当做变量的。

其中UMyClass_Blueprintable_To_NotBlueprintable可以当做变量是因为父类UMyClass_Blueprintable可以当做变量，因此就继承了下来。



原理：

可见MD_IsBlueprintBase的判断用来决定是否能创建子类

```
bool FKismetEditorUtilities::CanCreateBlueprintOfClass(const UClass* Class)
{
    bool bCanCreateBlueprint = false;

    if (Class)
    {
        bool bAllowDerivedBlueprints = false;
        GConfig->GetBool(TEXT("Kismet"), TEXT("AllowDerivedBlueprints"), /*out*/
bAllowDerivedBlueprints, GEngineIni);

        bCanCreateBlueprint = !Class->HasAnyClassFlags(CLASS_Deprecated)
            && !Class->HasAnyClassFlags(CLASS_NewerVersionExists)
            && (!Class->ClassGeneratedBy || (bAllowDerivedBlueprints &&
!IsClassABlueprintSkeleton(Class)));

        const bool bIsBPGC = (Cast<UBlueprintGeneratedClass>(Class) != nullptr);

        const bool bIsValidClass = Class-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_IsBlueprintBase)
            || (Class == UObject::StaticClass())
            || (Class == USceneComponent::StaticClass() || Class ==
UActorComponent::StaticClass())
            || bIsBPGC; // BPs are always considered inheritable

        bCanCreateBlueprint &= bIsValidClass;
    }

    return bCanCreateBlueprint;
}
```

BlueprintType

- **功能描述:** 可当做变量类型
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **作用机制:** Meta增加BlueprintType
- **关联项:** NotBlueprintType
- **常用程度:** ★★★★☆

可当做变量类型。

关键是设置BlueprintType和NotBlueprintType这两个metadata.

示例代码：

```
/*
```

```

(BlueprintType = true, IncludePath = Class/MyClass_BlueprintType.h,
ModuleRelativePath = Class/MyClass_BlueprintType.h)
*/
UCLASS(BlueprintType)
class INSIDER_API UMyClass_BlueprintType :public UObject
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h)
*/
UCLASS()
class INSIDER_API UMyClass_BlueprintType_Child :public UMyClass_BlueprintType
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h, NotBlueprintType = true)
*/
UCLASS(NotBlueprintType)
class INSIDER_API UMyClass_NotBlueprintType :public UObject
{
    GENERATED_BODY()
};

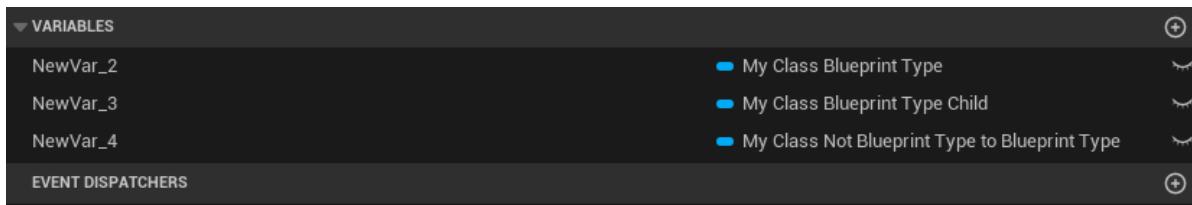
/*
(BlueprintType = true, IncludePath = Class/MyClass_BlueprintType.h,
ModuleRelativePath = Class/MyClass_BlueprintType.h)
*/
UCLASS(BlueprintType)
class INSIDER_API UMyClass_NotBlueprintType_To_BlueprintType:public
UMyClass_NotBlueprintType
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h, NotBlueprintType = true)
*/
UCLASS(NotBlueprintType)
class INSIDER_API UMyClass_BlueprintType_To_NotBlueprintType:public
UMyClass_BlueprintType
{
    GENERATED_BODY()
};

```

示例结果：

带有BlueprintType =true的才可以当作变量



原理：

在UEdGraphSchema_K2::IsAllowableBlueprintVariableType的3个重载函数分别判断UEnum, UClass, UScriptStruct能否当作变量。

用UEdGraphSchema_K2::IsAllowableBlueprintVariableType来判断

```
const UClass* ParentClass = InClass;
while(ParentClass)
{
    // Climb up the class hierarchy and look for "BlueprintType" and
    "NotBlueprintType" to see if this class is allowed.
    if(ParentClass-
>GetBoolMetaData(FBlueprintMetadata::MD_AllowableBlueprintVariableType)
        ||
        ParentClass-
>HasMetaData(FBlueprintMetadata::MD_BlueprintSpawnableComponent))
    {
        return true;
    }
    else if(ParentClass-
>GetBoolMetaData(FBlueprintMetadata::MD_NotAllowableBlueprintVariableType))
    {
        return false;
    }
    ParentClass = ParentClass->GetSuperClass();
}
```

Const

- **功能描述：** 表示本类的内部属性不可在蓝图中被修改，只读不可写。
- **引擎模块：** Blueprint
- **元数据类型：** bool
- **作用机制：** 在ClassFlags中添加CLASS_Abstract
- **常用程度：** ★★★

表示本类的内部属性不可在蓝图中被修改，只读不可写。

继承的蓝图类也是如此。其实就是自动的给本类和子类上添加const的标志。注意只是在蓝图里检查，C++依然可以随意改变，遵循C++的规则。所以这个const是只给蓝图用的，在蓝图里检查。函数依然可以随便调用，只是没有属性的Set方法了，也不能改变了。

示例代码：

```
/*
    ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_Const |
    CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic |
    CLASS_Constructed
*/
UCLASS(Blueprintable, Const)
class INSIDER_API UMyClass_Const :public UObject
{
    GENERATED_BODY()

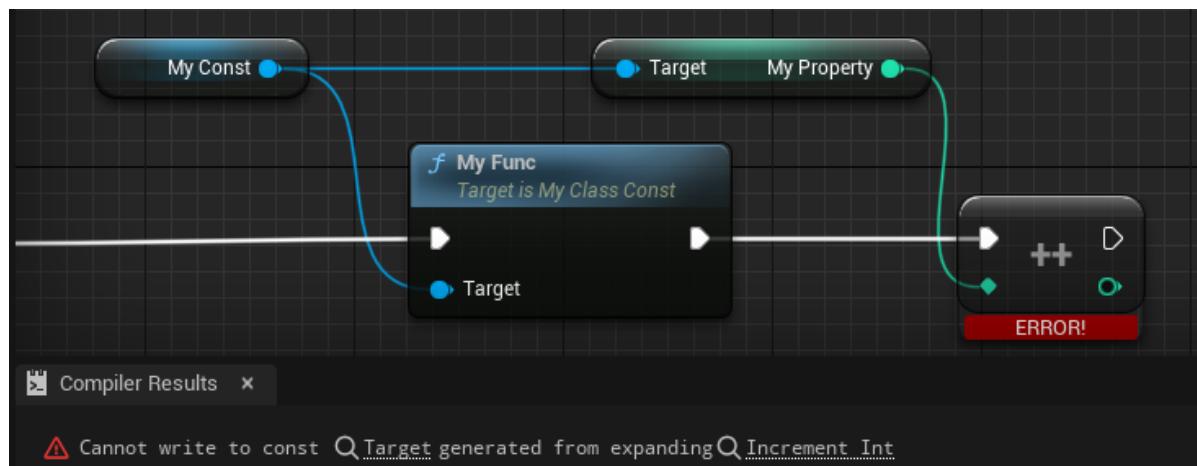
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UFUNCTION(BlueprintCallable)
    void MyFunc() { ++MyProperty; }
};
```

示例效果：

在蓝图子类中尝试修改属性会报错。



跟蓝图Class Settings里打开这个开关设定的一样



原理：

Const类生成的实例属性对带有const的标记，从而阻止修改自身的属性。

```

void FKCHandler_VariableSet::InnerAssignment(FKismetFunctionContext& Context,
UEdGraphNode* Node, UEdGraphPin* VariablePin, UEdGraphPin* ValuePin)
{
    if (!(*VariableTerm)->IsTermWritable())
    {
        CompilerContext.MessageLog.Error(*LOCTEXT("WriteConst_Error", "Cannot
write to const @@").ToString(), VariablePin);
    }
}

bool FBPTerminal::IsTermWritable() const
{
    return !bIsLiteral && !bIsConst;
}

```

HideFunctions

- 功能描述:** 在子类的函数覆盖列表里隐藏掉某些函数。
- 引擎模块:** Blueprint
- 元数据类型:** strings=(abc, "d|e", "x|y|z")
- 作用机制:** 在Meta中增加HideFunctions
- 关联项:** ShowFunctions
- 常用程度:** ★★

在子类的函数覆盖列表里隐藏掉某些函数。

- 在蓝图中鼠标右键依然可以查看到该类下BlueprintCallable的函数，依然可以调用，本标记只是用在类的函数覆盖列表上。
- HideFunctions其实只能填函数名字，想要隐藏一个目录下的函数，是需要HideCategories再额外定义的。

源码中只有一个地方用到，一个很好的示例是UCameraComponent中定义的SetFieldOfView和SetAspectRatio，对UCineCameraComponent来说是无意义的，因此隐藏掉会更好。

```

class ENGINE_API UCameraComponent : public USceneComponent
{
UFUNCTION(BlueprintCallable, Category = Camera)
    virtual void SetFieldofview(float InFieldofview) { Fieldofview =
InFieldofview; }
UFUNCTION(BlueprintCallable, Category = Camera)
    void SetAspectRatio(float InAspectRatio) { AspectRatio = InAspectRatio; }

UCLASS(HideCategories = (CameraSettings), HideFunctions = (SetFieldofview,
SetAspectRatio), Blueprintable, ClassGroup = Camera, meta =
(BlueprintSpawnableComponent), Config = Engine)
class CINEMATICCAMERA_API UCineCameraComponent : public UCameraComponent

```

示例代码：

```

UCLASS(Blueprintable, HideFunctions = (MyFunc1, MyEvent2), hideCategories=
EventCategory2)
class INSIDER_API AMyClass_HideFunctions :public AActor
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    void MyFunc1() {}

    UFUNCTION(BlueprintCallable)
    void MyFunc2() {}

    UFUNCTION(BlueprintCallable, Category = "FuncCategory1")
    void MyFuncInCategory1() {}

    UFUNCTION(BlueprintCallable, Category = "FuncCategory2")
    void MyFuncInCategory2() {}

public:
    UFUNCTION(BlueprintImplementableEvent)
    void MyEvent1();

    UFUNCTION(BlueprintImplementableEvent)
    void MyEvent2();

    UFUNCTION(BlueprintImplementableEvent, Category = "EventCategory1")
    void MyEventInCategory1();

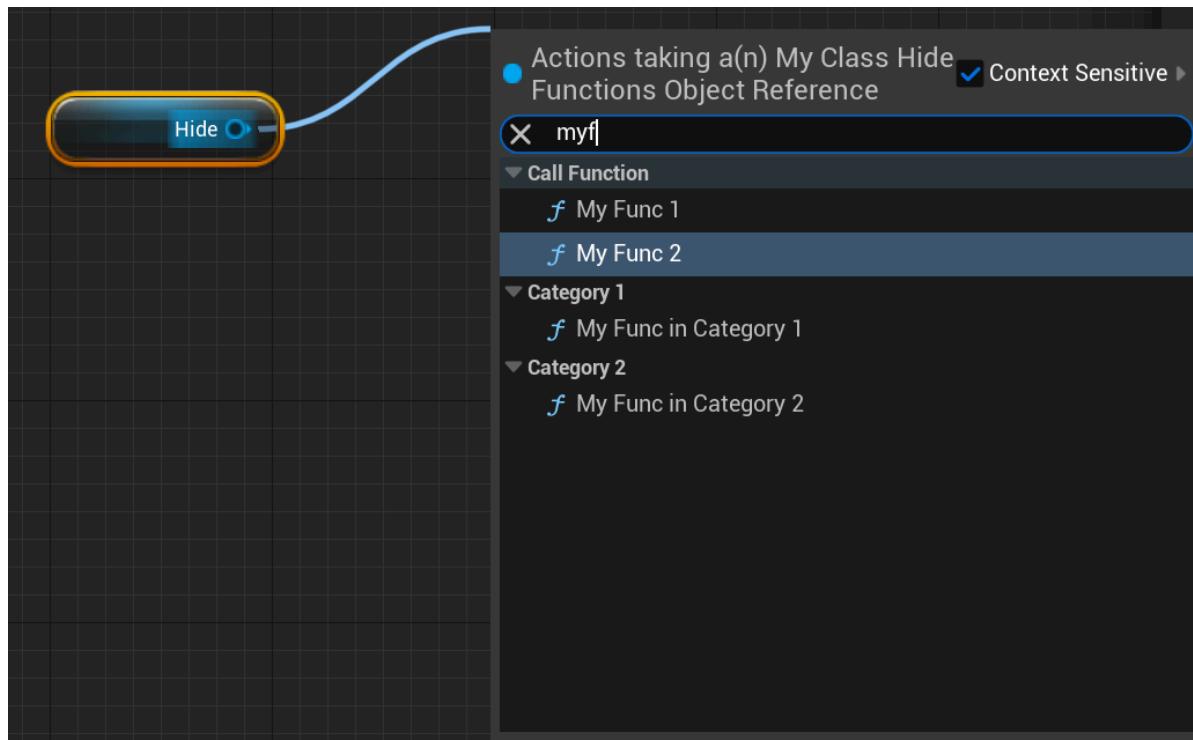
    UFUNCTION(BlueprintImplementableEvent, Category = "EventCategory2")
    void MyEventInCategory2();
};

UCLASS(Blueprintable, ShowFunctions = (MyEvent2), showCategories= EventCategory2)
class INSIDER_API AMyClass_ShowFunctions :public AMyClass_HideFunctions
{
    GENERATED_BODY()
public:
};

```

示例效果：

发现Callable的函数是依然可以调用的。



在HideFunction子类里，函数重载会发现少两个

▼ FUNCTIONS (21 OVERRIDA...

Override ▾ +

ConstructionScript

MACROS

VARIABLES

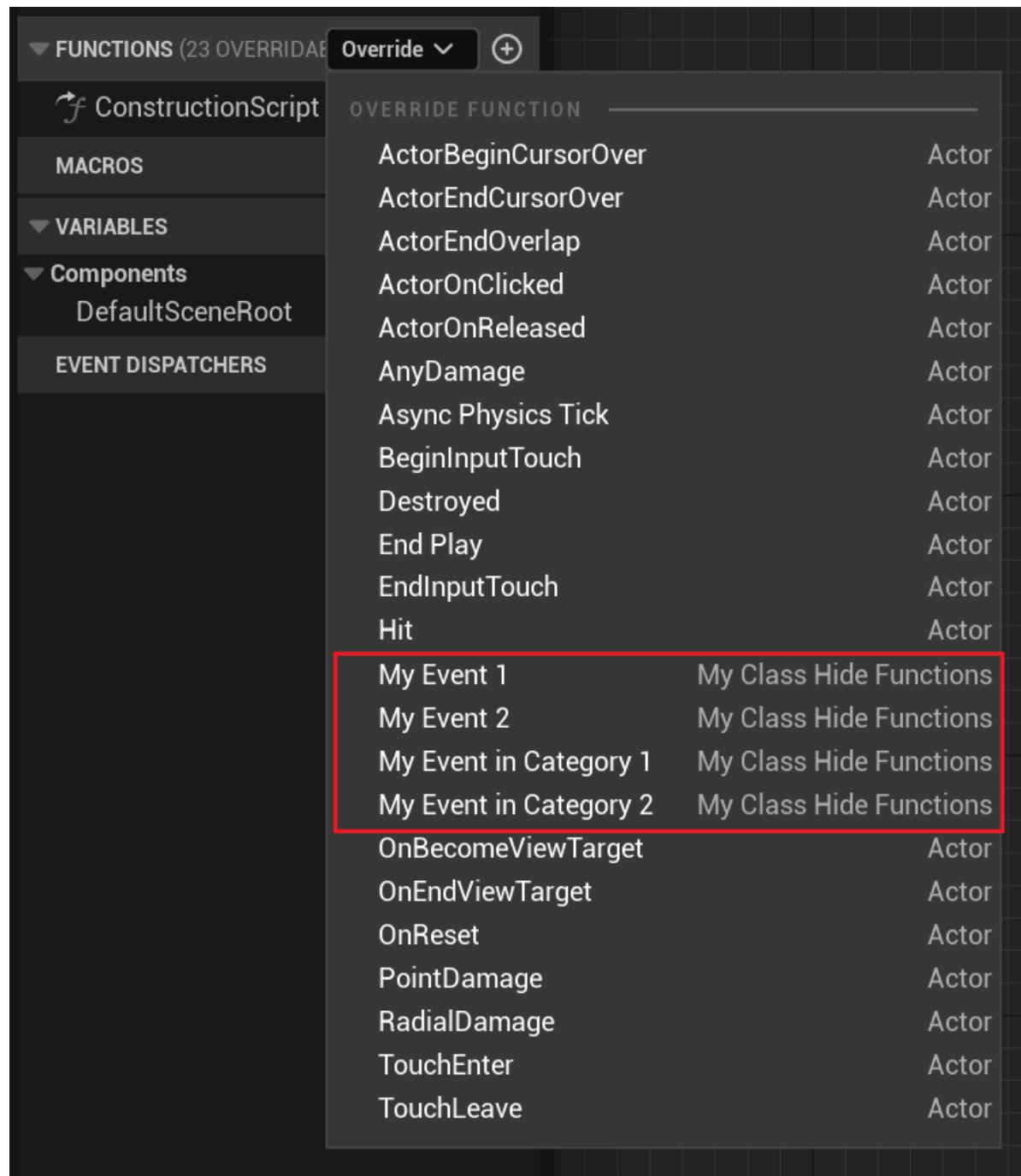
Components

DefaultSceneRoot

EVENT DISPATCHERS

Override Function	Type
ActorBeginCursorOver	Actor
ActorEndCursorOver	Actor
ActorEndOverlap	Actor
ActorOnClicked	Actor
ActorOnReleased	Actor
AnyDamage	Actor
Async Physics Tick	Actor
BeginInputTouch	Actor
Destroyed	Actor
End Play	Actor
EndInputTouch	Actor
Hit	Actor
My Event 1	My Class Hide Functions
My Event in Category 1	My Class Hide Functions
OnBecomeViewTarget	Actor
OnEndViewTarget	Actor
OnReset	Actor
PointDamage	Actor
RadialDamage	Actor
TouchEnter	Actor
TouchLeave	Actor

在ShowFunction的子类里可以重新打开Event2和EventCategory2



原理：

原理显示，HideFunctions其实只能填函数名字，想要隐藏一个目录下的函数，是需要HideCategories再额外定义的。

```
bool IsFunctionHiddenFromClass( const UFunction* InFunction, const UClass* Class )
{
    bool bResult = false;
    if( InFunction )
    {
        bResult = Class->IsFunctionHidden( *InFunction->GetName() );

        static const FName FunctionCategory(TEXT("Category")); // BlueprintMetadata::MD_FunctionCategory
        if( !bResult && InFunction->HasMetaData( FunctionCategory ) )
        {
            ...
        }
    }
}
```

```

        FString const& FuncCategory = InFunction-
>GetMetaData(FunctionCategory);
        bResult = FEditorCategoryUtils::IsCategoryHiddenFromClass(Class,
FuncCategory);
    }
}
return bResult;
}

```

NeedsDeferredDependencyLoading

- **引擎模块:** Blueprint
- **元数据类型:** bool
- **作用机制:** 在ClassFlags增加CLASS_NeedsDeferredDependencyLoading

源码例子:

```

UCLASS(NeedsDeferredDependencyLoading, MinimalAPI)
class UBlueprintGeneratedClass : public UClass, public
IBlueprintPropertyGuidProvider
{
}

```

原理:

```

if (ClassFlags.HasAnyFlags(EClassFlags::NeedsDeferredDependencyLoading) &&
!IsChildOf(Session.UClass))
{
    // CLASS_NeedsDeferredDependencyLoading can only be set on classes derived
    // from UClass
    this.LogError($"'NeedsDeferredDependencyLoading' is set on '{SourceName}' but
    the flag can only be used with classes derived from UClass.");
}

```

NotBlueprintable

- **功能描述:** 不可在蓝图里继承，隐含作用也不可当作变量
- **引擎模块:** Blueprint
- **元数据类型:** bool
- **作用机制:** 在Meta去除IsBlueprintBase和BlueprintType
- **关联项:** Blueprintable
- **常用程度:** ★★★★

NotBlueprintType

- **功能描述:** 不可当做变量类型

- **引擎模块:** Blueprint
- **元数据类型:** bool
- **作用机制:** Meta移除BlueprintType
- **关联项:** BlueprintType
- **常用程度:** ★★★★

ShowFunctions

- **功能描述:** 在子类的函数覆盖列表里重新打开某些函数。
- **引擎模块:** Blueprint
- **元数据类型:** strings=(abc, "d|e", "x|y|z")
- **作用机制:** 在Meta中去除HideFunctions
- **关联项:** HideFunctions
- **常用程度:** ★★

在子类的函数覆盖列表里重新打开某些函数。

测试代码和效果图见HideFunctions。

原理:

UHT中的代码，可见ShowFunctions的作用就是去除掉之前设置的HideFunctions。

```
private void MergeCategories()
{
    MergeShowCategories();

    // Merge ShowFunctions and HideFunctions
    AppendStringListMetaData(SuperClass, UhtNames.HideFunctions, HideFunctions);
    foreach (string value in ShowFunctions)
    {
        HideFunctions.RemoveSwap(value);
    }
    ShowFunctions.Clear();
}
```

SparseClassDataType

- **功能描述:** 让Actor的一些重复不变的数据存放在一个共同的结构里，以达到减少内容使用量的目的
- **引擎模块:** Blueprint
- **元数据类型:** string="abc"
- **作用机制:** 在Meta中增加SparseClassDataTypes
- **关联项:** NoGetter
- **常用程度:** ★★★

这是个重构和性能优化的点。在使用SparseClassDataType的时候，分为两种情况，一是以前的Actor想利用这个特性来优化，二是新创建的Actor一开始就想使用这个特性。

示例用法：

分为两部分：

一，旧的Actor存在冗余属性

简而言之是那些不会在BP改变的属性。C++方面，如果有修改这些属性，也要修改为使用Get函数来获得，从而转到SparseDataStruct里去。

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
    GENERATED_BODY()

public:
    UPROPERTY(EditDefaultsOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");

    UPROPERTY(EditAnywhere)
    float MyFloat_EditAnywhere = 555.f;

    UPROPERTY(BlueprintReadWrite)
    float MyFloat_BlueprintReadWrite = 666.f;
};
```

改为以下的代码。把属性用WITH_EDITORONLY_DATA包起来，以示意只在editor下做操作，在runtime是已经消除的。加上_DEPRECATED后缀标记也是为了进一步提醒原先BP里的访问要去除。重载MoveDataToSparseClassDataStruct以便把现在BP Class Defaults里配置的值拷贝给新的FMySparseClassData结构数值。

```
USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta=(GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes= MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
```

```

{
    GENERATED_BODY()

public:
#if WITH_EDITOR
    // ~ This function transfers existing data into FMySparseClassData.
    virtual void MoveDataToSparseClassDataStruct() const override;
#endif // WITH_EDITOR
public:
#if WITH_EDITORONLY_DATA
    UPROPERTY()
        int32 MyInt_EditDefaultOnly_DEPRECATED = 123;

    UPROPERTY()
        int32 MyInt_BlueprintReadOnly_DEPRECATED = 1024;

    UPROPERTY()
        FString MyString>EditDefault_ReadOnly_DEPRECATED = TEXT("MyName");
#endif // WITH_EDITORONLY_DATA
public:
    UPROPERTY(EditAnywhere)
        float MyFloat_EditAnywhere = 555.f;

    UPROPERTY(BlueprintReadWrite)
        float MyFloat_BlueprintReadWrite = 666.f;
};

//cpp
#if WITH_EDITOR
void AMyActor_SparseClassDataTypes::MoveDataToSparseClassDataStruct() const
{
    // make sure we don't overwrite the sparse data if it has been saved already
    UBlueprintGeneratedClass* BPClass = Cast<UBlueprintGeneratedClass>
    (GetClass());
    if (BPClass == nullptr || BPClass->bIsSparseClassDataSerializable == true)
    {
        return;
    }

    Super::MoveDataToSparseClassDataStruct();

#if WITH_EDITORONLY_DATA
    // Unreal Header Tool (UHT) will create GetMySparseClassData automatically.
    FMySparseClassData* SparseclassData = GetMySparseClassData();

    // Modify these lines to include all Sparse Class Data properties.
    SparseclassData->MyInt_EditDefaultOnly = MyInt_EditDefaultOnly_DEPRECATED;
    SparseclassData->MyInt_BlueprintReadOnly =
    MyInt_BlueprintReadOnly_DEPRECATED;
    SparseclassData->MyString>EditDefault_ReadOnly_DEPRECATED =
    MyString>EditDefault_ReadOnly_DEPRECATED;
#endif // WITH_EDITORONLY_DATA
}
#endif // WITH_EDITOR

```

在BP的PostLoad加载之后，会自动的调用MoveDataToSparseClassDataStruct，所以要在内部检测bIsSparseClassSerializable.

```
void UBlueprintGeneratedClass::PostLoadDefaultObject(UObject* object)
{
    FScopeLock SerializeAndPostLoadLock(&SerializeAndPostLoadCritical);

    Super::PostLoadDefaultObject(Object);

    if (Object == ClassDefaultObject)
    {
        // Rebuild the custom property list used in post-construct initialization
        // logic. Note that PostLoad() may have altered some serialized properties.
        UpdateCustomPropertyListForPostConstruction();

        // Restore any property values from config file
        if (HasAnyClassFlags(CLASS_Config))
        {
            ClassDefaultObject->LoadConfig();
        }
    }

#if WITH_EDITOR
    Object->MoveDataToSparseClassDataStruct();

    if (Object->GetSparseClassDataStruct())
    {
        // now that any data has been moved into the sparse data structure we can
        // safely serialize it
        bIsSparseClassSerializable = true;
    }

    ConformSparseClassData(Object);
#endif
}
```

在UClass下

```
protected:
    /** This is where we store the data that is only changed per class instead of
     * per instance */
    void* SparseClassData;

    /** The struct used to store sparse class data. */
    UScriptStruct* SparseClassDataStruct;
```

在构造UClass的时候，会SetSparseClassDataStruct来把结构传进去，因此就把结构关联起来。

```
UClass* Z_Construct_UClass_AMyActor_SparseClassDataTypes()
{
    if (!Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton)
    {

UECodeGen_Private::ConstructUClass(Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes,
                                    Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton,
                                    Z_Construct_UClass_AMyActor_SparseClassDataTypes_Statics::ClassParams);
```

```

    Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.outersingleton-
>SetSparseClassDataStruct(AMyActor_SparseClassDataTypes::StaticGetMySparseClassDa
taScriptStruct());
}

return
Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.outersingleton;
}

```

注意此时BP里没法blueprint get 那些ReadOnly的变量的，因为有_DEPRECATED在占用着。一种方法是自己再额外定义Gettter方法：

```

UFUNCTION(BlueprintPure)
int32 GetMyMyInt_BlueprintReadOnly() const
{
    return GetMySparseClassData()->MyInt_BlueprintReadOnly;
}

```

二，另一种方法是在MoveDataToSparseClassDataStruct之后（记得要打开编辑器，并且打开子类BP蓝图后保存）就干脆删除掉AMyActor_SparseClassDataTypes里的冗余属性，全部使用FMySparseClassData中的值。从而变成：

```

USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta=(GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes= MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere)
    float MyFloat_EditAnywhere = 555.f;

    UPROPERTY(BlueprintReadWrite)
    float MyFloat_BlueprintReadWrite = 666.f;
};

```

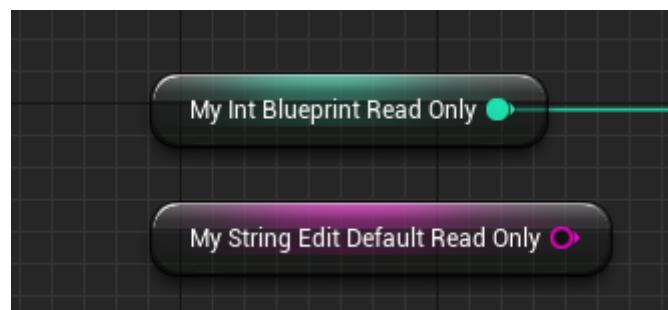
这样就达到了最终的效果，这个效果也对新的Actor要采用冗余属性的结果也是一样的。注意此时，在BP里是依然可以访问BlueprintReadOnly属性的，因为UHT和BP系统已经帮我们加了一层访问方便的控制。

示例效果：

UHT会帮我们生成C++访问函数：

```
#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_30_SPARSE_DAT
A \
FMySparseClassData* GetMySparseClassData(); \
FMySparseClassData* GetMySparseClassData() const; \
const FMySparseClassData* GetMySparseClassData(EGetSparseClassDataMethod
GetMethod) const; \
static UScriptStruct* StaticGetMySparseClassDataScriptStruct(); \
int32 GetMyInt_EditDefaultOnly() \
{ \
    return GetMySparseClassData()->MyInt_EditDefaultOnly; \
} \
int32 GetMyInt_EditDefaultOnly() const \
{ \
    return GetMySparseClassData()->MyInt_EditDefaultOnly; \
} \
int32 GetMyInt_BlueprintReadOnly() \
{ \
    return GetMySparseClassData()->MyInt_BlueprintReadOnly; \
} \
int32 GetMyInt_BlueprintReadOnly() const \
{ \
    return GetMySparseClassData()->MyInt_BlueprintReadOnly; \
} \
const FString& GetMyString_EditDefault_ReadOnly() \
{ \
    return GetMySparseClassData()->MyString_EditDefault_ReadOnly; \
} \
const FString& GetMyString_EditDefault_ReadOnly() const \
{ \
    return GetMySparseClassData()->MyString_EditDefault_ReadOnly; \
}
```

在BP中依然可以访问：



在Class Defaults里也可以改变值：

▼ My Sparse Class Data		
My Int Edit Default Only	1235	↶
My Int Blueprint Read...	777	↶
My String Edit Default...	Hello	↶

AdvancedClassDisplay

- **功能描述:** 把该类下的所有属性都默认显示在高级目录下
- **引擎模块:** Category
- **元数据类型:** bool
- **作用机制:** 在Meta增加AdvancedClassDisplay
- **常用程度:** ★★★★

让这个类的所有属性显示在本身类的Detail面板的“高级”栏目下显示。

但是可以通过在单个属性上使用SimpleDisplay来重载掉。在搜索了一番源码后，发现使用AdvancedClassDisplay的只有3个Actor，且这3个Actor里都没有再定义属性。

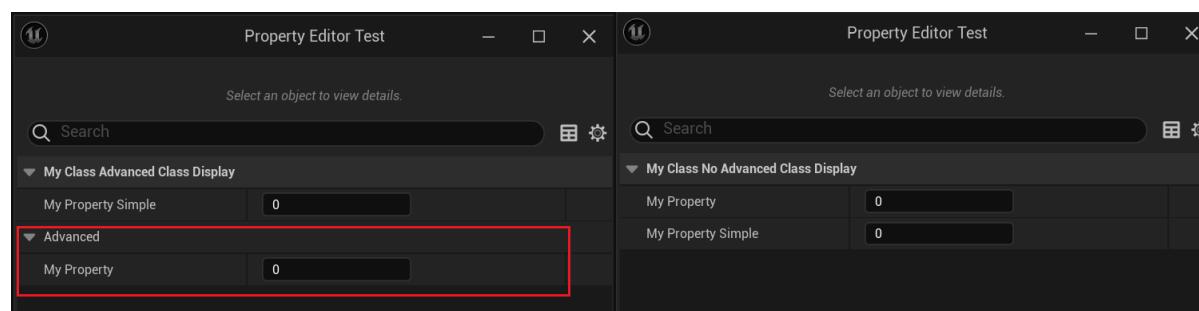
示例代码：

```
UCLASS(Blueprintable, AdvancedClassDisplay)
class INSIDER_API UMyClass_AdvancedClassDisplay :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SimpleDisplay)
    int32 MyProperty_Simple;
};

UCLASS(Blueprintable)
class INSIDER_API UMyClass_NoAdvancedClassDisplay :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SimpleDisplay)
    int32 MyProperty_Simple;
};
```

示例效果：

MyProperty_Simple即使在AdvancedClassDisplay的类中也依然是简单的显示。



原理：

```
// Property is advanced if it is marked advanced or the entire class is advanced
// and the property not marked as simple
static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
bool bAdvanced = Property.IsValid() ? (Property-
>HasAnyPropertyParams(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyParams(
CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()-
>GetBoolMetaData(Name_AdvancedClassDisplay) ) ) : false;
```

AutoCollapseCategories

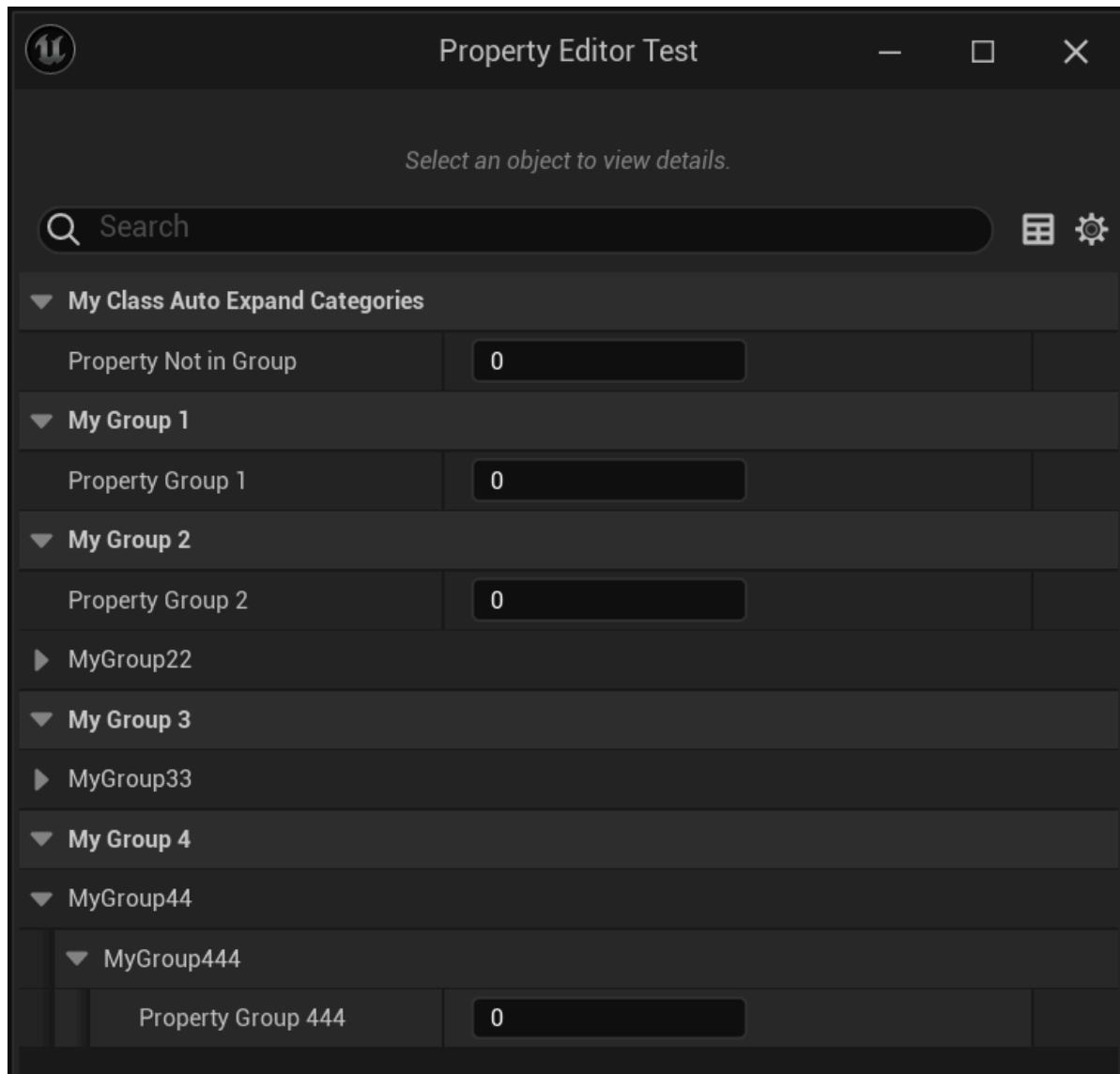
- **功能描述：** AutoCollapseCategories说明符使父类上的 AutoExpandCategories 说明符的列出类别的效果无效。
- **引擎模块：** Category
- **元数据类型：** strings=(abc, "d|e", "x|y|z")
- **作用机制：** 在Meta中增加AutoCollapseCategories，去除AutoExpandCategories
- **关联项：** DontAutoCollapseCategories、AutoExpandCategories
- **常用程度：** ★

示例代码：

```
UCLASS(Blueprintable, AutoCollapseCategories = ("MyGroup2|MyGroup22"))
class INSIDER_API UMyClass_AutoCollapseCategories : public
UMyClass_AutoExpandCategories
{
    GENERATED_BODY()
public:
};
```

示例结果：

关闭了Group22的展开，但是444的展开依然继承了



AutoExpandCategories

- 功能描述:** 指定此类的对象在细节面板中应该自动展开的Category。
- 引擎模块:** Category
- 元数据类型:** strings=(abc, "d|e", "x|y|z")
- 作用机制:** 在Meta中去除AutoCollapseCategories，增加AutoExpandCategories
- 关联项:** AutoCollapseCategories
- 常用程度:** ★

指定此类的对象在细节面板中应该自动展开的Category。

- 这里面的Category可以填多个，对应本类中属性身上定义的Category。
- 值得注意的是，编辑器会自动的保存属性目录的展开关闭状态。影响属性是否展开，还会受到DetailPropertyExpansion的配置的影响，在打开窗口后，SDetailsViewBase::UpdateFilteredDetails()会保存当前展开的属性项目，应该是为了下次打开的时候自动展开。保存的代码为GConfig->SetSingleLineArray(TEXT("DetailPropertyExpansion"), *Struct->GetName(), ExpandedPropertyItems, GEditorPerProjectIni); 从而在\Hello\Saved\Config\WindowsEditor\EditorPerProjectUserSettings.ini下保存。因此为了更好的测试该元数据的作用状态。应该手动先清除一下ini中的保存值后再测试。

[DetailCategories]

```

MyClass_AutoExpandCategories.MyClass_AutoExpandCategories=False
MyClass_AutoExpandCategories.MyGroup1=False
MyClass_AutoExpandCategories.MyGroup2=False
MyClass_AutoExpandCategories.MyGroup3=True
MyClass_AutoExpandCategories.MyGroup4=True

[DetailPropertyExpansion]
GeometryCache="\\"Object.GeometryCache.Materials\\" \\"Object.GeometryCache.Tracks\""
"
Object="\\"Object.MyGroup2.MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\" "
GeometryCacheCodecV1="\\"Object.GeometryCache.TopologyRanges\" "
GeometryCacheCodecBase="\\"Object.GeometryCache.TopologyRanges\" "
MassSettings="\\"Object.Mass\" "
DeveloperSettings=
SmartObjectSettings="\\"Object.SmartObject\" "
MyClass_ShowCategories=
MyClass_ShowCategoriesChild=
MyClass_DontCollapseCategories="\\"Object.MyGroup2.MyGroup22|MyGroup22\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33.MyGroup3|MyGroup33|MyGroup333\\" "
MyClass_CollapseCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33.MyGroup3|MyGroup33|MyGroup333\\" "
MyClass_AutoExpandCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\" "
MyClass_AutoExpandCategoriesCompare=
MyClass_AutoCollapseCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\" "

```

根据代码搜索规则，AutoExpandCategories 和AutoCollapseCategories的值要用空格隔开。顶层目录一开始默认就是打开的，所以AutoExpandCategories 一般用在子层目录。而且还有个限制是必须一级一级都打开。直接打开最子目录还不行。因此在示例代码里必须要把中间的二级目录"MyGroup4|MyGroup44"也都得写上。

示例代码：

```

UCLASS(Blueprintable, AutoExpandCategories = ("MyGroup2|MyGroup22",
"MyGroup4|MyGroup44", "MyGroup4|MyGroup44|MyGroup444"))
class INSIDER_API UMyClass_AutoExpandCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")
        int Property_Group2;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
        int Property_Group22;
}

```

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3|MyGroup33")
    int Property_Group33;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup4|MyGroup44|MyGroup44")
    int Property_Group44;
};
```

源码里最复杂的样例：

```
UCLASS(Config = Engine, PerObjectConfig, BlueprintType, AutoCollapseCategories =
("Data Layer|Advanced"), AutoExpandCategories = ("Data Layer|Editor", "Data
Layer|Advanced|Runtime"))
class ENGINE_API UDataLayerInstance : public UObject
```

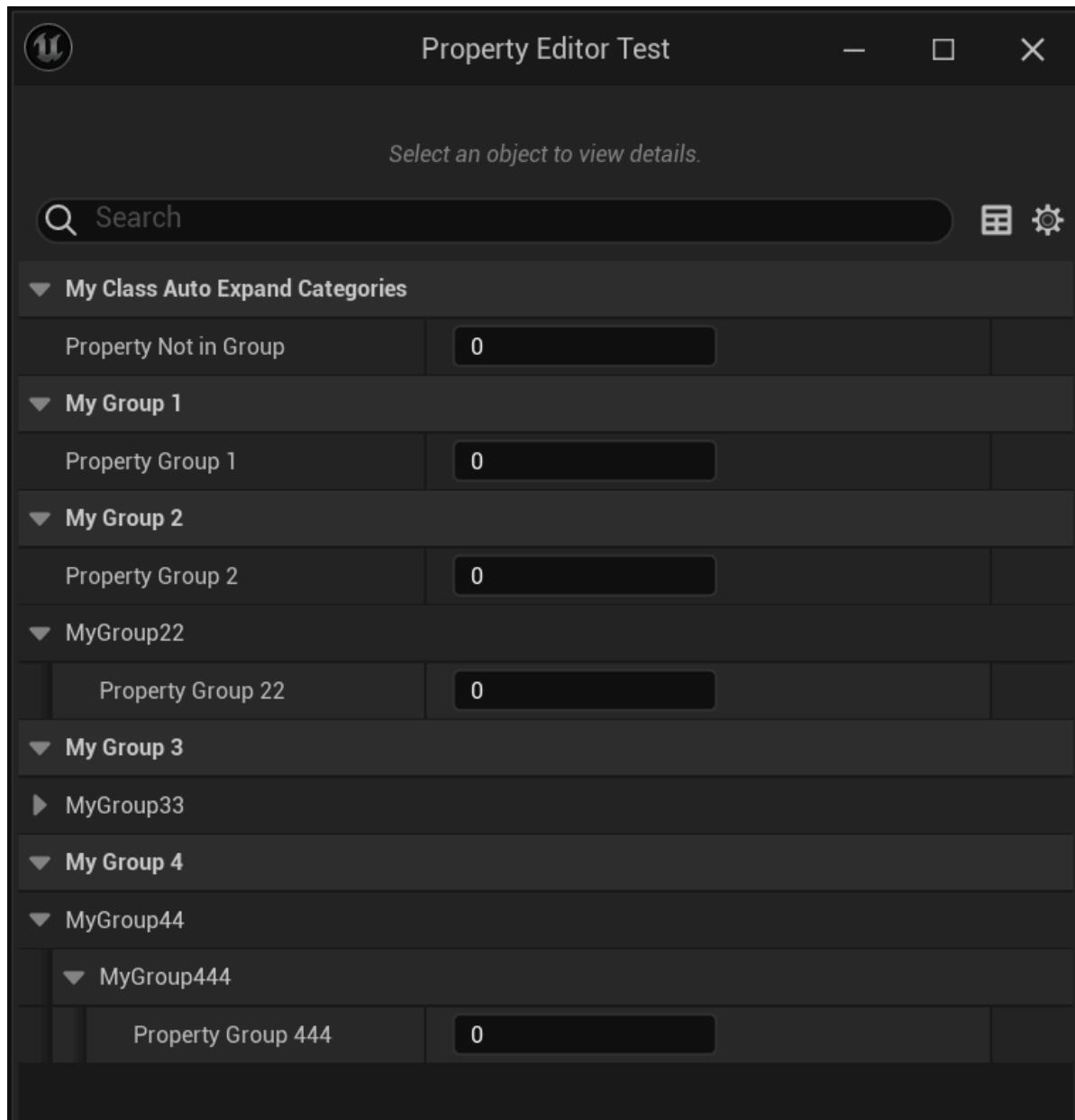
可以打开子目录： UCLASS(Blueprintable, AutoExpandCategories = ("MyGroup2|MyGroup22",
"MyGroup4|MyGroup44", "MyGroup4|MyGroup44|MyGroup44"))

不可以打开子目录： UCLASS(Blueprintable, AutoExpandCategories =
("MyGroup2|MyGroup22", "MyGroup4|MyGroup44|MyGroup44"))

示例效果：

在Saved\EditorPerProjectUserSettings中删除掉DetailCategories以及DetailPropertyExpansion下的 MyClass_AutoCollapseCategories值之后再用testprops class=MyClass_AutoExpandCategories来打开该窗口：

通过对比可以看出Expand确实可以自动展开子目录方便立马编辑。要求是AutoExpandCategories 里填的目录要和属性上的Category匹配



原理：

UClass里提取AutoExpandCategories和AutoCollapseCategories的元数据来判断Category是否应该显示。

```
if (BaseClass->IsAutoExpandCategory(*CategoryName.ToString()) &&
!BaseClass->IsAutoCollapseCategory(*CategoryName.ToString()))
{
    NewCategoryNode->SetNodeFlags(EPropertyNodeFlags::Expanded, true);
}

bool UClass::IsAutoExpandCategory(const TCHAR* InCategory) const
{
    static const FName NAME_AutoExpandCategories(TEXT("AutoExpandCategories"));
    if (const FString* AutoExpandCategories =
FindMetaData(NAME_AutoExpandCategories))
    {
        return !FCString::StrfindDelim(**AutoExpandCategories, InCategory,
TEXT(" "));
    }
    return false;
}
```

```

}

bool UClass::IsAutoCollapseCategory(const TCHAR* InCategory) const
{
    static const FName
NAME_AutoCollapseCategories(TEXT("AutoCollapseCategories"));
    if (const FString* AutoCollapseCategories =
FindMetaData(NAME_AutoCollapseCategories))
    {
        return !FCString::StrfindDelim(**AutoCollapseCategories, InCategory,
TEXT(" "));
    }
    return false;
}

```

ClassGroup

- 功能描述:** 指定组件在Actor的AddComponent面板里的分组，以及在蓝图右键菜单中的分组。
- 引擎模块:** Category, Editor
- 元数据类型:** string="a|b|c"
- 作用机制:** 在Meta中增加ClassGroupNames
- 常用程度:** ★★★

指定组件在Actor的AddComponent面板里的分组，以及在蓝图右键菜单中的分组。

示例代码：

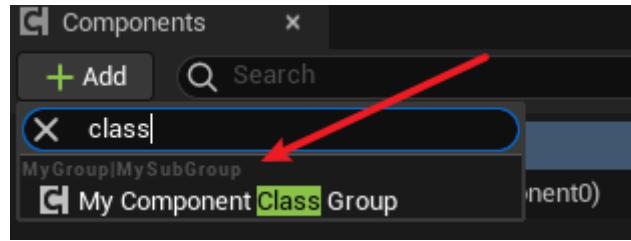
```

//ClassGroup 必须是BlueprintSpawnableComponent才有效
/*
(BlueprintSpawnableComponent = , BlueprintType = true, ClassGroupNames =
MyGroup|MySubGroup, IncludePath = Class/MyComponent_ClassGroup.h, IsBlueprintBase
= true, ModuleRelativePath = Class/MyComponent_ClassGroup.h)
*/
UCLASS(Blueprintable, ClassGroup="MyGroup|MySubGroup", meta =
(BlueprintSpawnableComponent))
class INSIDER_API UMyComponent_ClassGroup:public UActorComponent
{
    GENERATED_BODY()
public:
};

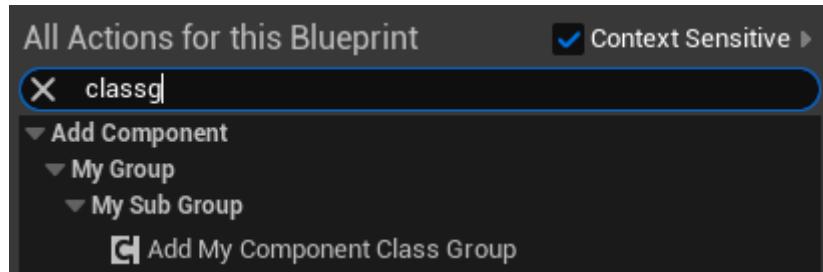
```

示例效果：

在添加组件的时候：



在蓝图中右键AddComponent，该测试只对带有BlueprintSpawnableComponent的UActorComponent起作用，因为只有BlueprintSpawnableComponent才可在蓝图中动态添加组件。



原理：

Metadata中的ClassGroupNames，被使用方法是 UClass::GetClassGroupNames，这个又是在 BlueprintComponentNodeSpawner中被使用。还有一个使用地方是ComponentTypeRegistry.cpp 中，也是在判断Component。因此这个ClassGroup确实是只被Component使用的。

```
static FText GetDefaultMenuCategory(const TSubclassOf<UActorComponent>
ComponentClass)
{
    TArray< FString> ClassGroupNames;
    ComponentClass->GetClassGroupNames(ClassGroupNames);

    if (FKismetEditorUtilities::IsClassABlueprintSpawnableComponent(Class))
    {
        TArray< FString> ClassGroupNames;
        Class->GetClassGroupNames(ClassGroupNames);
    }
}
```

CollapseCategories

- 功能描述：** 在类的属性面板里隐藏所有带Category的属性，但是只对带有多个嵌套Category的属性才起作用。
- 引擎模块：** Category
- 元数据类型：** bool
- 作用机制：** 在ClassFlags中添加CLASS_CollapseCategories
- 关联项：** DontCollapseCategories
- 常用程度：** ★★

在类的属性面板里隐藏所有带Category的属性，但是只对带有多个嵌套Category的属性才起作用。

示例代码：

```
/*
```

```

ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_CollapseCategories | 
CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic | 
CLASS_Constructed
*/
UCLASS(Blueprintable, CollapseCategories)
class INSIDER_API UMyClass_CollapseCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
        int Property_Group22;

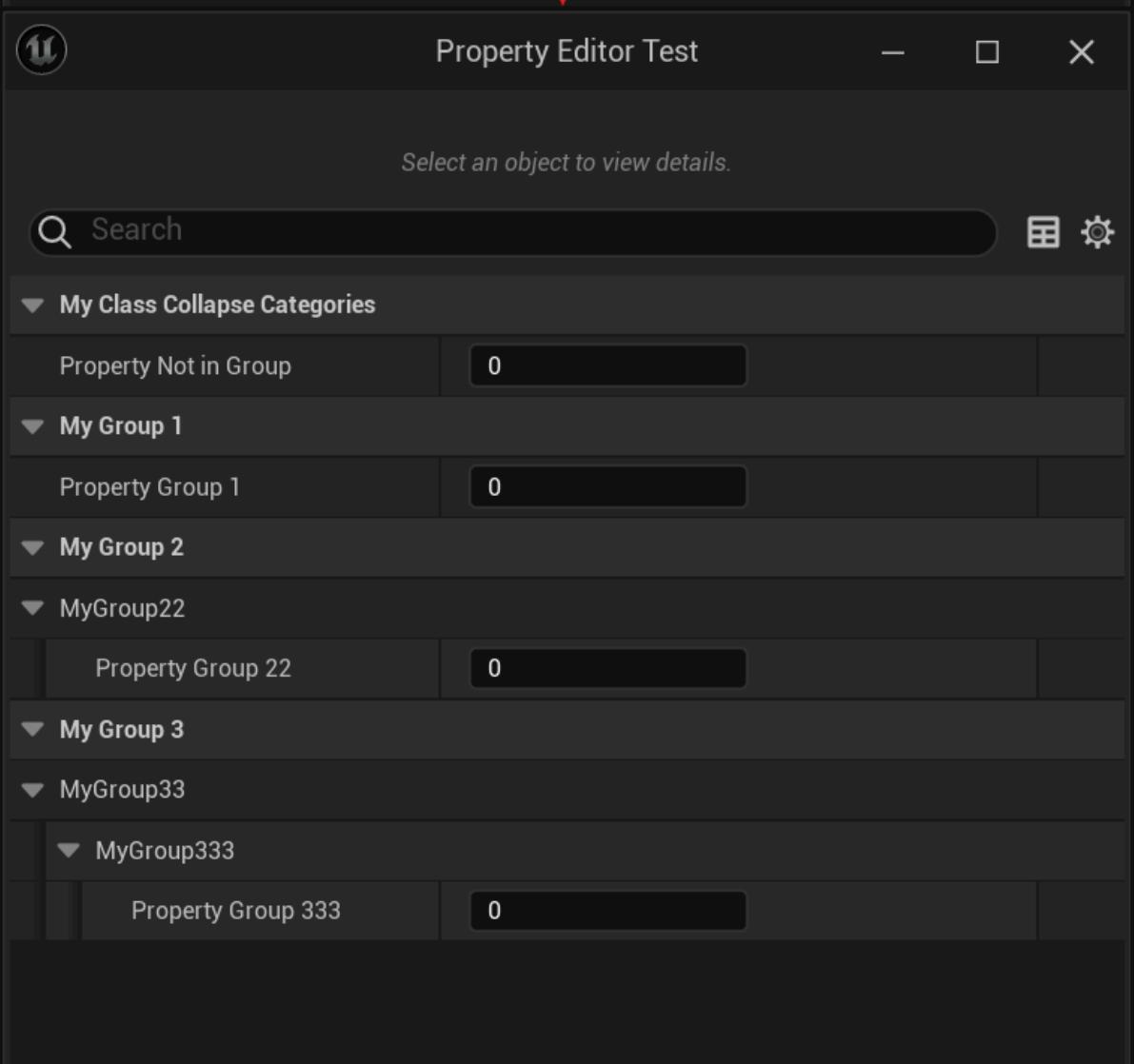
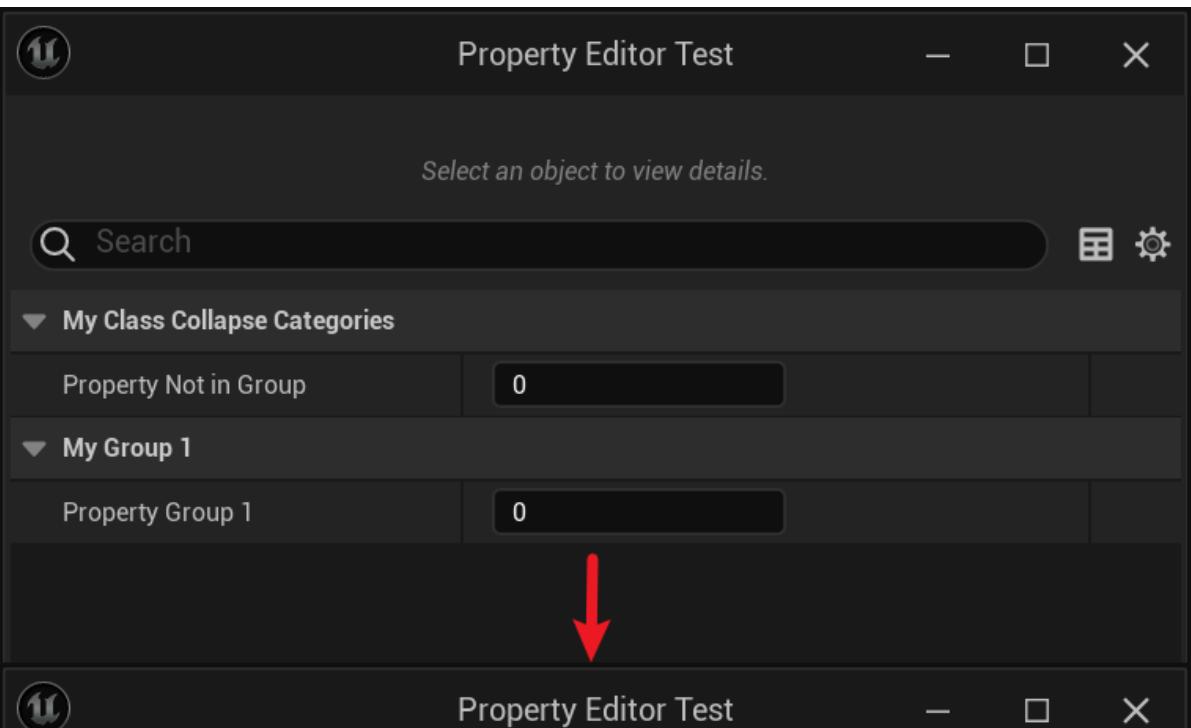
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup3|MyGroup33|MyGroup333")
        int Property_Group33;
};

/*
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI | 
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
*/
UCLASS(Blueprintable, dontCollapseCategories)
class INSIDER_API UMyClass_DontCollapseCategories :public
UMyClass_CollapseCategories
{
    GENERATED_BODY()
public:
};

```

示例效果：

第一个是UMyClass_CollapseCategories 的效果，第二个是UMyClass_DontCollapseCategories 的效果，可见一些属性被隐藏了起来。



原理：

```
if (Specifier == TEXT("collapseCategories"))
{
    // Class' properties should not be shown categorized in the editor.
    ClassFlags |= CLASS_CollapseCategories;
}
else if (Specifier == TEXT("dontCollapseCategories"))
{
    // Class' properties should be shown categorized in the editor.
    ClassFlags &= ~CLASS_CollapseCategories;
}
```

ComponentWrapperClass

- 功能描述：** 指定该类为一个简单的封装类，忽略掉子类的Category相关设置。
- 引擎模块：** Category
- 元数据类型：** bool
- 作用机制：** 在Meta中增加IgnoreCategoryKeywordsInSubclasses
- 常用程度：** ★★

指定该类为一个简单的封装类，忽略掉子类的Category相关设置。

如名字所说，为一个组件的包装类，其实就是一个Actor简单的只包含一个Component。这种简单的包装关系，典型的例子是ALight包装ULightComponent， ASkeletalMeshActor包装USkeletalMeshComponent。

控制子类上面定义的hideCategories和showCategories都被忽略，而直接采用基类上的目录定义，也就是本组件包装类上的目录设置。当前源码里只有BlueprintEditorUtils.cpp在用，而这是蓝图打开的过程，因此这个只有在双击打开一个蓝图的时候才起作用。普通的UObject类，直接用testprops创建的窗口，因为不是双击打开蓝图，因此是无法生效的。

ComponentWrapperClass在源码里搜了一下，只有一些Actor在用。

示例代码：

```
UCLASS(Blueprintable, BlueprintType, ComponentWrapperClass, hideCategories =
MyGroup3) //依然会显示出Property_Group3
class AMyActor_ComponentwrapperClass : public AActor
{
    GENERATED_UCLASS_BODY()
public:
    UPROPERTY(BlueprintReadOnly, VisibleAnywhere)
        class UPointLightComponent* PointLightComponent;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup3)
        int Property_Group3;
};

UCLASS(Blueprintable, BlueprintType, hideCategories = MyGroup3)
class AMyActor_NoComponentwrapperClass : public AActor //Property_Group3会被隐藏
{
```

```

GENERATED_UCLASS_BODY()

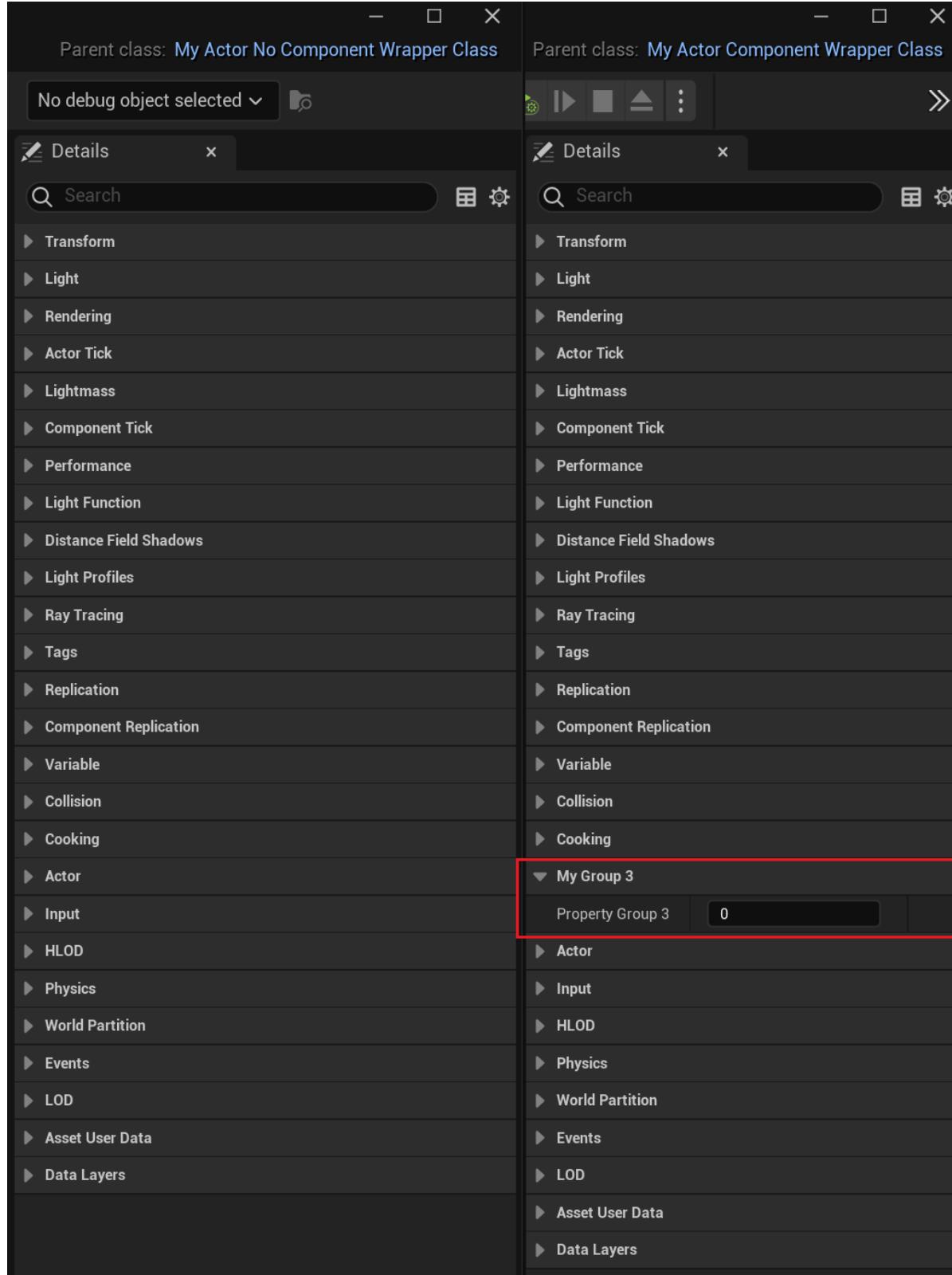
public:
    UPROPERTY(BlueprintReadOnly, VisibleAnywhere)
        class UPointLightComponent* PointLightComponent;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup3)
        int Property_Group3;
};

```

子类的实际作用效果：

子类里的MyGroup3即使被隐藏了起来，也还是显示了出来。



原理：

ComponentWrapperClass实际会造成IgnoreCategoryKeywordsInSubclasses=true的元数据添加。因此在有了IgnoreCategoryKeywordsInSubclasses元数据之后，就不会判断之后的ShowCategories等设置了。

当前源码里只有BlueprintEditorUtils.cpp在用，而这是蓝图打开的过程，因此这个只有在双击打开一个蓝图的时候才起作用。普通的UObject类，直接用testprops创建的窗口，因为不是双击打开蓝图，因此是无法生效的。

```
case EClassMetadataSpecifier::Componentwrapperclass:
    MetaData.Add(NAME_IgnoreCategoryKeywordsInSubclasses, TEXT("true"));
    // "IgnoreCategoryKeywordsInSubclasses"
    break;
///////////////////////////////////////////////////////////////////////////////
E:\P4V\Engine\Source\Editor\UnrealEd\Private\Kismet2\BlueprintEditorUtils.cpp
void FBlueprintEditorUtils::RecreateClassMetaData(UBlueprint* Blueprint, UClass* Class, bool bRemoveExistingMetaData)

if (!ParentClass-
>HasMetaData(FBlueprintMetadata::MD_IgnoreCategoryKeywordsInSubclasses)) //如果没有这个设置
{
    // we want the categories just as they appear in the parent class
    // (set bHomogenize to false) - especially since homogenization
    // could inject spaces

    //以下这些操作是当没有这个设置的时候，子类会继承父类的目录设置。
    FEditorCategoryUtils::GetClassHideCategories(ParentClass,
AllHideCategories, /*bHomogenize =*/false);
    if (ParentClass->HasMetaData(TEXT("ShowCategories")))
    {
        Class->SetMetaData(TEXT("ShowCategories"), *ParentClass-
>GetMetaData("ShowCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("AutoExpandCategories")))
    {
        Class->SetMetaData(TEXT("AutoExpandCategories"), *ParentClass-
>GetMetaData("AutoExpandCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("AutoCollapseCategories")))
    {
        Class->SetMetaData(TEXT("AutoCollapseCategories"), *ParentClass-
>GetMetaData("AutoCollapseCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("PrioritizeCategories")))
    {
        Class->SetMetaData(TEXT("PrioritizeCategories"), *ParentClass-
>GetMetaData("PrioritizeCategories"));
    }
}
```

DontAutoCollapseCategories

- **功能描述:** 使列出的类别的继承自父类的AutoCollapseCategories说明符无效。
- **引擎模块:** Category
- **元数据类型:** strings="a, b, c"
- **作用机制:** 在Meta中去除AutoCollapseCategories
- **关联项:** AutoCollapseCategories
- **常用程度:** ★

根据代码，只是简单的移除AutoCollapseCategories，和AutoExpandCategories的区别就是不会自动加一个展开。在源码里搜了一下，并没有使用到。而且当前的源码实现有bug，做不到去除。

```
case EClassMetadataSpecifier::AutoExpandCategories:  
  
    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);  
  
    for (FString& value : PropSpecifier.Values)  
    {  
        AutoCollapseCategories.RemoveSwap(value);  
        AutoExpandCategories.AddUnique(MoveTemp(value));  
    }  
    break;  
  
case EClassMetadataSpecifier::AutoCollapseCategories:  
  
    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);  
  
    for (FString& value : PropSpecifier.Values)  
    {  
        AutoExpandCategories.RemoveSwap(value);  
        AutoCollapseCategories.AddUnique(MoveTemp(value));  
    }  
    break;  
case EClassMetadataSpecifier::DontAutoCollapseCategories:  
  
    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);  
  
    for (const FString& value : PropSpecifier.Values)  
    {  
        AutoCollapseCategories.RemoveSwap(value); //当前AutoCollapseCategories  
        的值还是空的。去除是没有用的  
    }  
    break;
```

改动：

```
FUnrealClassDefinitionInfo::MergeClassCategories()放最后：  
// Merge DontAutoCollapseCategories and AutoCollapseCategories  
    for (const FString& value : DontAutoCollapseCategories)  
    {  
        AutoCollapseCategories.RemoveSwap(value);  
    }  
    DontAutoCollapseCategories.Empty();
```

```

改为:

case EClassMetadataSpecifier::DontAutoCollapseCategories:

    FHeaderParser::RequiresSpecifierValue(*this, PropSpecifier);

    for (FString& value : PropSpecifier.Values)
    {
        DontAutoCollapseCategories.AddUnique(MoveTemp(value));
        //AutoCollapseCategories.RemoveSwap(value);
    }
    break;

```

DontCollapseCategories

- 功能描述:** 使继承自基类的CollapseCategories说明符无效。
- 引擎模块:** Category
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中去除CLASS_CollapseCategories
- 关联项:** CollapseCategories
- 常用程度:** ★★

理论上是去除类标志上的CLASS_CollapseCategories标志。可以重新打开所有的属性显示。

HideCategories

- 功能描述:** 在类的ClassDefaults属性面板里隐藏某些Category的属性。
- 引擎模块:** Category
- 元数据类型:** strings=(abc, "d|e", "x|y|z")
- 关联项:** ShowCategories
- 常用程度:** ★★★★

在类的ClassDefaults属性面板里隐藏某些Category的属性。

注意，要先在类里定义属性然后设置它的Category。HideCategories的信息会被UHT分析，并保存到 UClass的元数据里去。HideCategories的信息可以被子类继承下来。

示例代码：

```

UCLASS(Blueprintable, hideCategories = MyGroup1)
class INSIDER_API UMyClass_HideCategories :public UObject
{
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | MyGroup3")
        int Property_Group23;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
};

```

```
/*
(BlueprintType = true, HideCategories = MyGroup2 | MyGroup3, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/

UCLASS(Blueprintable, showCategories = MyGroup1, hideCategories = "MyGroup2 |
MyGroup3")
class INSIDER_API UMyClass_HideCategoriesChild :public UMyClass_ShowCategories
{
    GENERATED_BODY()
public:

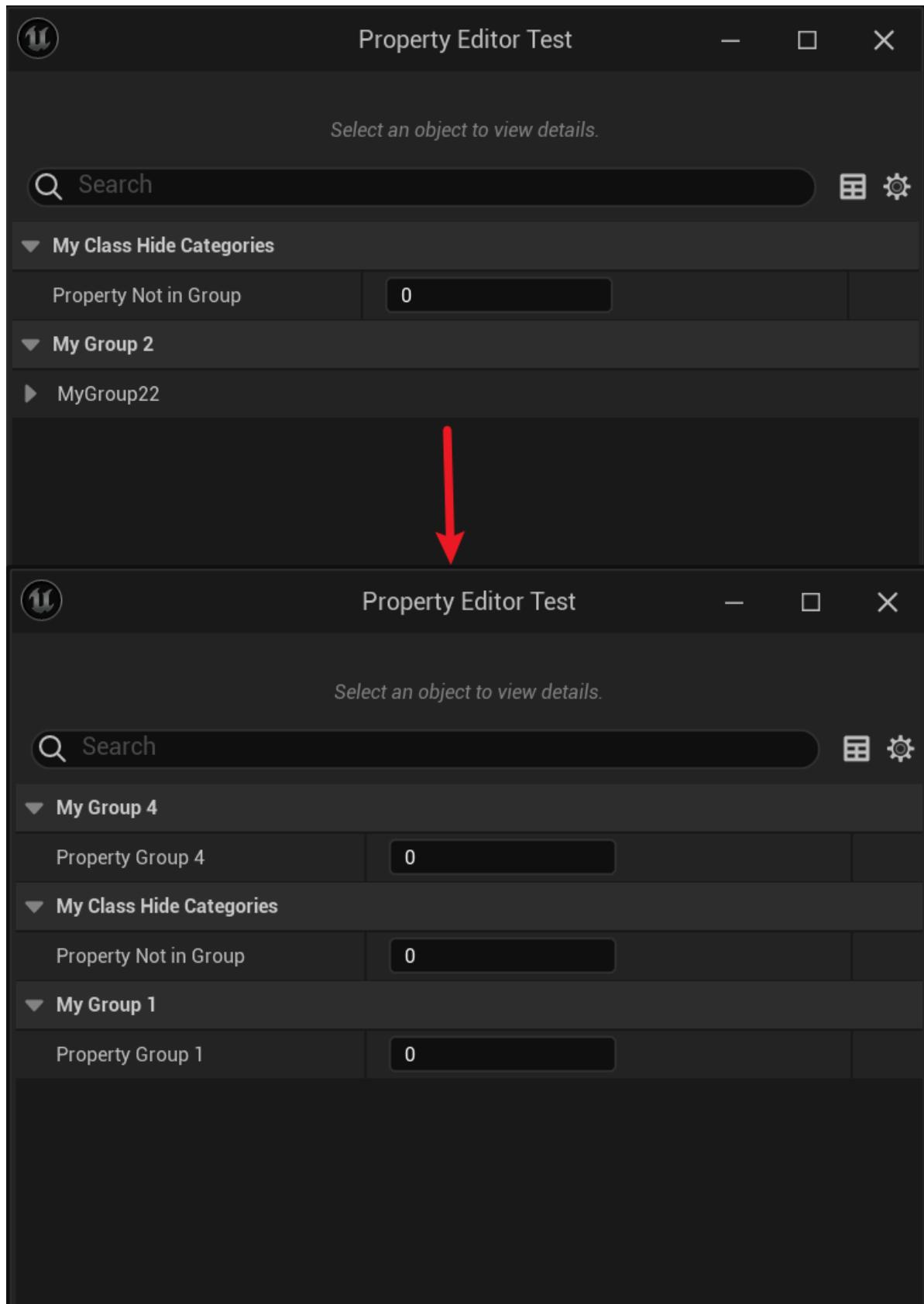
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")
        int Property_Group2;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3")
        int Property_Group3;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup4)
        int Property_Group4;
};
```

示例效果：

注意这里，单独的MyGroup2和MyGroup3也都没有显示。所以判断的标准只要目录符合某个目录名字匹配就行。



原理：

在GetClassHideCategories中检查ClassHideCategoriesMetaKey元数据。

```
void FEditorCategoryUtils::GetClassShowCategories(const UStruct* class,
TArray< FString>& CategoriesOut)
{
    CategoriesOut.Empty();
}
```

```

using namespace FEditorCategoryUtilsImpl;
if (Class->HasMetaData(ClassShowCategoriesMetaKey))
{
    const FString& ShowCategories = Class-
>GetMetaData(ClassShowCategoriesMetaKey);
    ShowCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

    for (FString& Category : CategoriesOut)
    {
        Category =
GetCategoryDisplayString(FText::FromString(Category)).ToString();
    }
}
}

void FEditorCategoryUtils::GetClassHideCategories(const UStruct* Class,
TArray<FString>& CategoriesOut, bool bHomogenize)
{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassHideCategoriesMetaKey))
    {
        const FString& HideCategories = Class-
>GetMetaData(ClassHideCategoriesMetaKey);

        HideCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        if (bHomogenize)
        {
            for (FString& Category : CategoriesOut)
            {
                Category = GetCategoryDisplayString(Category);
            }
        }
    }
}

```

PrioritizeCategories

- 功能描述:** 把指定的属性目录优先显示在细节面板的前面。
- 引擎模块:** Category
- 元数据类型:** strings=(abc, "d|e", "x|y|z")
- 作用机制:** 在Meta中增加PrioritizeCategories
- 常用程度:** ★★★

把指定的属性目录优先显示在细节面板的前面。

示例代码:

```

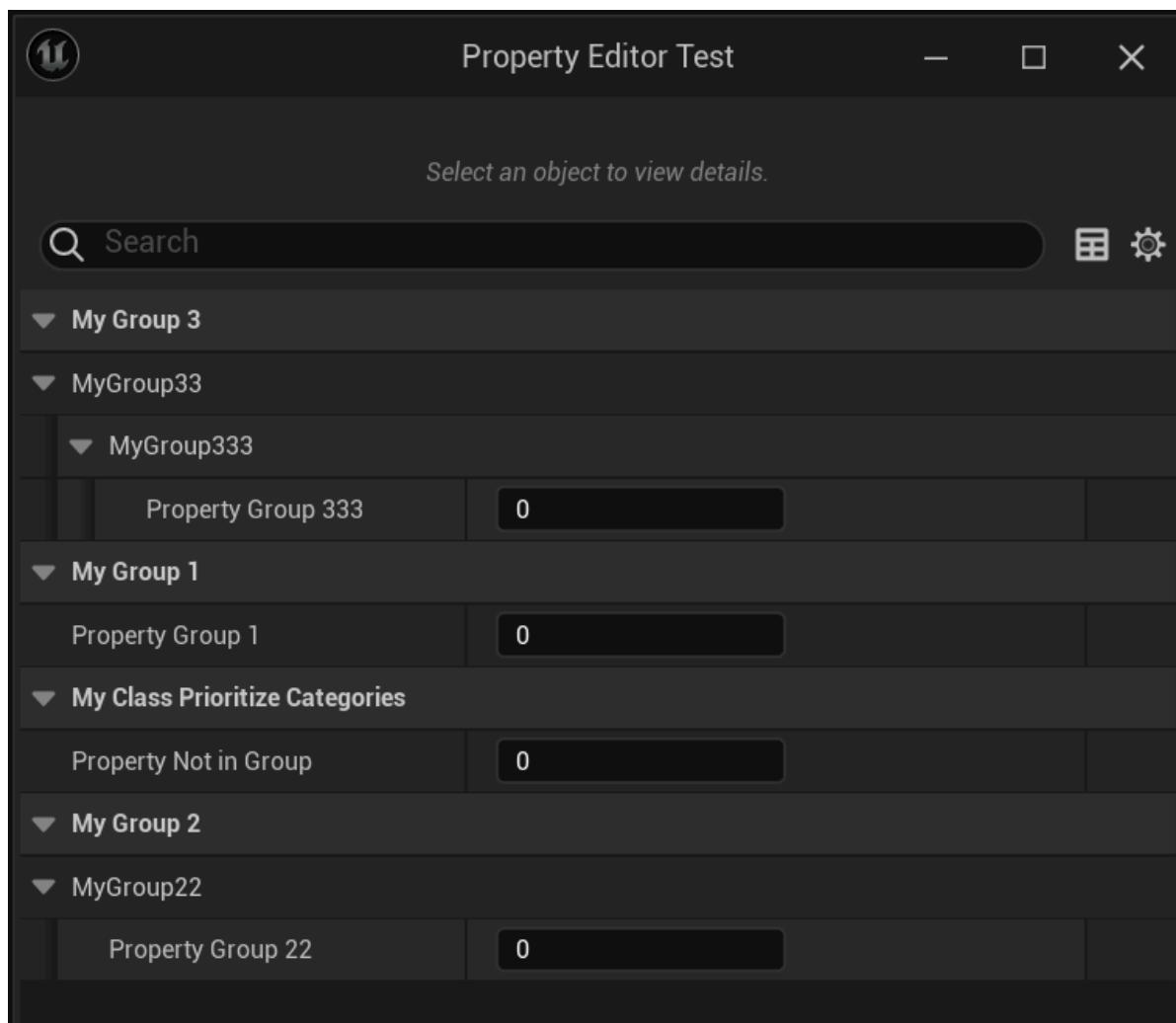
UCLASS(Blueprintable, PrioritizeCategories=
("MyGroup3|MyGroup33|MyGroup333", "MyGroup1"))
class INSIDER_API UMyClass_PrioritizeCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
        int Property_Group22;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup3|MyGroup33|MyGroup333")
        int Property_Group333;
};

```

示例结果：

可见Property_Group333排到了最前面。



原理：

在UClass::GetPrioritizeCategories(TArray& OutPrioritizedCategories)中获取优先级目录。原理是按照指定的顺序放到SortedCategories里，所以就会被首先创建出属性目录来。

```

TArray<FString> ClassPrioritizeCategories;
Class->GetPrioritizeCategories(ClassPrioritizeCategories);
for (const FString& ClassPrioritizeCategory : ClassPrioritizeCategories)
{
    FName PrioritizeCategoryName = FName(ClassPrioritizeCategory);
    SortedCategories.AddUnique(PrioritizeCategoryName);
    PrioritizeCategories.AddUnique(PrioritizeCategoryName);
}

```

ShowCategories

- 功能描述:** 在类的ClassDefaults属性面板里显示某些Category的属性。
- 引擎模块:** Category
- 元数据类型:** strings=(abc, "d|e", "x|y|z")
- 作用机制:** 在Meta中增加HideCategories
- 关联项:** HideCategories
- 常用程度:** ★★★

在类的ClassDefaults属性面板里显示某些Category的属性。使列出的类别的继承自基类的 HideCategories说明符无效。

ShowCategories会被UHT分析，但不会被保存到UClass的元数据里去。它作用的方式是可以抹去之前基类设置的HideCategories的属性。ShowCategories可以被子类继承下来。

示例代码：

```

/*
(BlueprintType = true, HideCategories = MyGroup1, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/
UCLASS(Blueprintable, hideCategories = MyGroup1)
class INSIDER_API UMyClass_ShowCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | MyGroup3")
        int Property_Group2;

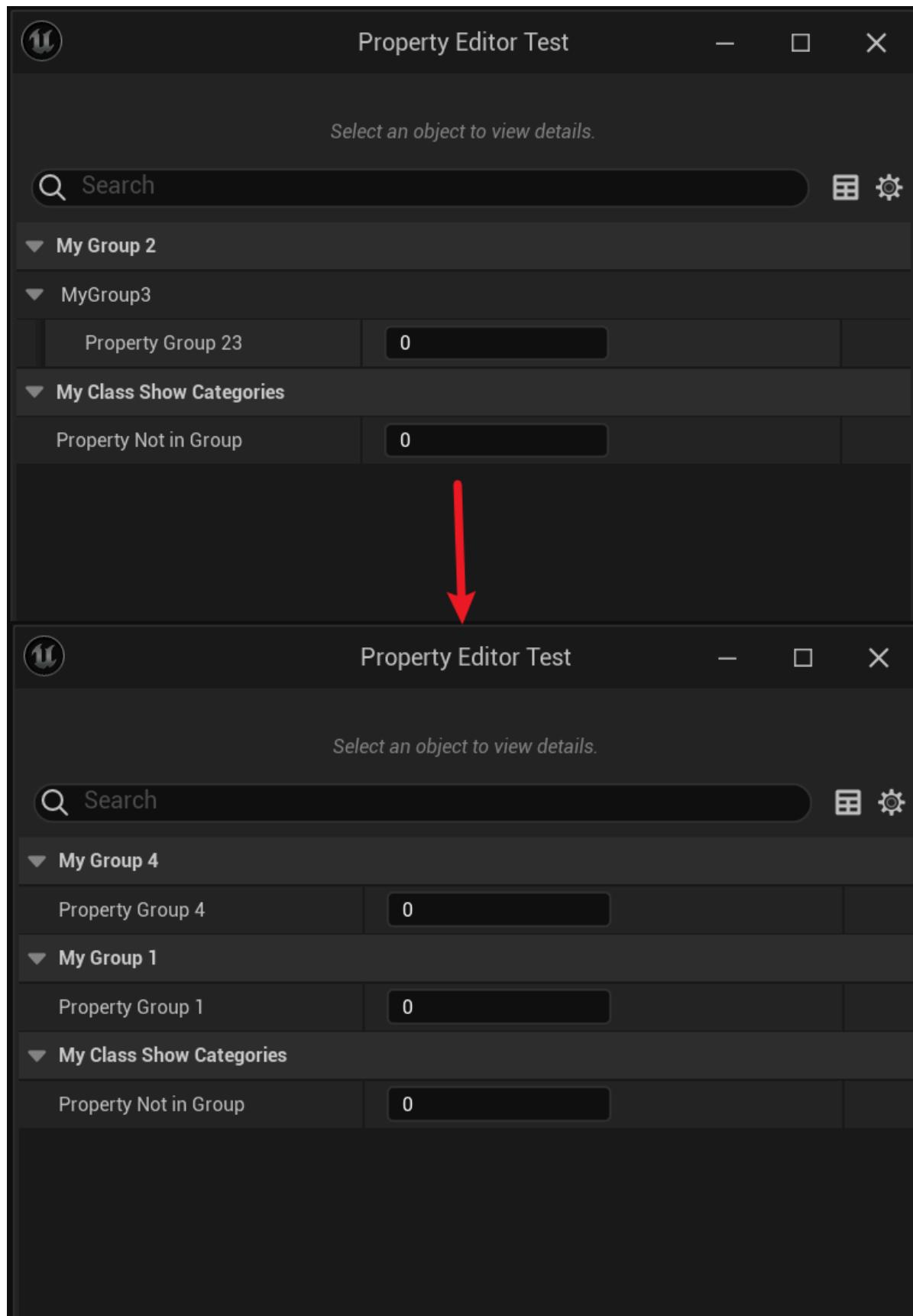
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
};

/*
(BlueprintType = true, HideCategories = MyGroup2 | MyGroup3, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/

```

```
UCLASS(Blueprintable, showCategories = MyGroup1, hideCategories = "MyGroup2 |  
MyGroup3")  
class INSIDER_API UMyClass_ShowCategoriesChild :public UMyClass_ShowCategories  
{  
    GENERATED_BODY()  
public:  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")  
        int Property_Group2;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3")  
        int Property_Group3;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup4)  
        int Property_Group4;  
};
```

示例效果：



原理：

其实实际上UHT保存的只在HideCategories里，这点通过对类的元数据查看就可知。

```
void FEditorCategoryUtils::GetClassShowCategories(const UStruct* Class,  
TArray< FString>& CategoriesOut)
```

```

{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassShowCategoriesMetaKey))
    {
        const FString& ShowCategories = Class-
>GetMetaData(ClassShowCategoriesMetaKey);
        ShowCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        for (FString& Category : CategoriesOut)
        {
            Category =
GetCategoryDisplayString(FText::FromString(Category)).ToString();
        }
    }
}

void FEditorCategoryUtils::GetClassHideCategories(const UStruct* Class,
TArray<FString>& CategoriesOut, bool bHomogenize)
{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassHideCategoriesMetaKey))
    {
        const FString& HideCategories = Class-
>GetMetaData(ClassHideCategoriesMetaKey);

        HideCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        if (bHomogenize)
        {
            for (FString& Category : CategoriesOut)
            {
                Category = GetCategoryDisplayString(Category);
            }
        }
    }
}

```

Config

- **功能描述:** 指定配置文件的名字，把该对象的值保存到ini配置文件中。
- **引擎模块:** Config
- **元数据类型:** string="abc"
- **作用机制:** Config文件名存在FName UClass::ClassConfigName这个参数里
- **关联项:** PerObjectConfig、ConfigDoNotCheckDefaults、DefaultConfig、GlobalUserConfig、ProjectUserConfig
- **常用程度:** ★★★★☆

指定配置文件的名字，把该对象的值保存到ini配置文件中。

- 一整个类在ini中只有一个节的值，因此一般是保存的CDO对象，但也可以用普通对象。
- Config文件名称的元数据值保存在FName UClass::ClassName。
- 默认是保存在Saved/XXX.ini的Local文件中。
- 此说明符会传播到所有子类并且无法使此说明符无效，但是子类可通过重新声明config说明符并提供不同的ConfigName来更改配置文件。
- 常见的ConfigName值是“Engine”、“Editor”、“Input”和“Game”。
- 可以自己手动调用SaveConfig和LoadConfig来读写配置值。CDO的值会被引擎自己的从配置中读取而更新。
- 想保存到配置文件里的属性要相应的用UPROPERTY(config)修饰。

示例代码：

```
UCLASS(Config = Game)
class INSIDER_API UMyClass_Config :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertywithConfig = 123;
};

//测试代码
UMyClass_Config* testObject = NewObject<UMyClass_Config>
(GetTransientPackage(), TEXT("testObject"));
testObject->SaveConfig();

//生成
\Hello\Saved\Config\windowsEditor\Game.ini
[/Script/Insider.MyClass_Config]
MyPropertywithConfig=123
```

原理：

在引擎启动的时候UObjectLoadAllCompiledInDefaultProperties会加载所有Class的CDO，在多个调用链条之后会自动的调用CDO的LoadConfig来初始化CDO的值。

```

static void UObjectLoadAllCompiledInDefaultProperties(TArray<UClass*>&
OutAllNewClasses)
{
    for (UClass* Class : NewClasses)
    {
        UE_LOG(Log UObjectBootstrap, Verbose, TEXT("GetDefaultObject Begin %s
%s"), *Class->GetOutermost()->GetName(), *Class->GetName());
        Class->GetDefaultObject();
        UE_LOG(Log UObjectBootstrap, Verbose, TEXT("GetDefaultObject End %s %s"),
*Class->GetOutermost()->GetName(), *Class->GetName());
    }
}

```

ConfigDoNotCheckDefaults

- 功能描述:** 指定在保存配置值的时候忽略上一级的配置值的一致性检查。
- 引擎模块:** Config
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中增加CLASS_ConfigDoNotCheckDefaults
- 关联项:** Config
- 常用程度:** ★

指定在保存配置值的时候忽略上一级的配置值的一致性检查。

- 在保存配置的时候，决定是否要先根据Base或Default的配置来检查属性是否一致，如果一致就不用序列化写入下来。但加上这个标志后，即使同上一个层级的配置值相同也无论如何都要保存下来。

UCLASS(config=XXX, configdonotcheckdefaults): 表示这个类对应的配置文件不会检查XXX层级上层的DefaultXXX配置文件是否有该信息（后面会解释层级），就直接存储到Saved目录下。

示例代码：

```

UCLASS(Config = Game)
class INSIDER_API UMyClass_Config :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
};

UCLASS(Config = Game, configdonotcheckdefaults)
class INSIDER_API UMyClass_ConfigDoNotCheckDefaults :public UMyClass_Config
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfigSub = 123;
};

```

```

UCLASS(Config = Game)
class INSIDER_API UMyClass_ConfigDefaultChild :public UMyClass_Config
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
        int32 MyPropertyWithConfigSub = 123;
};

```

示例效果：

```

void UMyClass_Config_Test::TestConfigCheckDefaultSave()
{
    auto* testObject = NewObject<UMyClass_ConfigDoNotCheckDefaults>
    (GetTransientPackage(), TEXT("testObjectCheckDefault"));
    auto* testObject2 = NewObject<UMyClass_ConfigDefaultChild>
    (GetTransientPackage(), TEXT("testObjectDefaultChild"));

    testObject->SaveConfig();
    testObject2->SaveConfig();
}

生成:
[/Script/Insider.MyClass_Config]
MyPropertyWithConfig=777

[/Script/Insider.MyClass_ConfigDoNotCheckDefaults]
MyPropertyWithConfigSub=123
MyPropertyWithConfig=777

[/Script/Insider.MyClass_ConfigDefaultChild]
MyPropertyWithConfigSub=123

```

由此可见， MyClass_ConfigDoNotCheckDefaults中的MyPropertyWithConfig的值默认跟 UMyClass_Config中的777值一致，但是依然会写入进来。在MyClass_ConfigDefaultChild类中， MyPropertyWithConfig的值因为没有改变，就会被略过。

在源码里搜configdonotcheckdefaults的时候发现常常和defaultconfig配合使用。什么时候应该使用 configdonotcheckdefaults？感觉是为了保持自己的完整性，无论如何都要全部写入进去。在 defaultConfig的时候，就可以不管Base里的值，都写入一份到Default配置里，这样在编辑上更加的完整。

原理：

```
const bool bShouldCheckIfIdenticalBeforeAdding = !GetClass()->HasAnyClassFlags(CLASS_ConfigDoNotCheckDefaults) && !bPerObject && bIsPropertyInherited;  
//简单的示例判断  
if (!bPropDeprecated && (!bShouldCheckIfIdenticalBeforeAdding || !Property->Identical_InContainer(this, SuperClassDefaultToObject, Index)))  
{  
    FString Value;  
    Property->ExportText_InContainer( Index, Value, this, this, this, PortFlags );  
    Config->SetString( *Section, *Key, *Value, PropFileName );  
}  
else  
{  
    // If we are not writing it to config above, we should make sure that this  
    // property isn't stagnant in the cache.  
    Config->RemoveKey( *Section, *Key, PropFileName );  
}
```

DefaultConfig

- **功能描述：** 指定保存到的配置文件层级是Project/Config/DefaultXXX.ini。
- **引擎模块：** Config
- **元数据类型：** bool
- **作用机制：** 在ClassFlags中增加CLASS_DefaultConfig
- **关联项：** Config
- **常用程度：** ★★★

指定保存到的配置文件层级是Project/Config/DefaultXXX.ini。

- 而不是默认的Saved/XXX.ini
- 一般用在编辑器里把Settings自动保存到Project/Config/DefaultXXX.ini里去

示例代码：

```
UCLASS(Config = MyGame, DefaultConfig)  
class INSIDER_API UMyClass_DefaultConfig :public UDeveloperSettings  
{  
    GENERATED_BODY()  
public:  
    /** Gets the settings container name for the settings, either Project or  
     * Editor */  
    virtual FName GetContainerName() const override { return TEXT("Project"); }  
    /** Gets the category for the settings, some high level grouping like,  
     * Editor, Engine, Game...etc. */  
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }  
    /** The unique name for your section of settings, uses the class's FName. */  
    virtual FName GetSectionName() const override { return TEXT("MyGame"); }  
public:
```

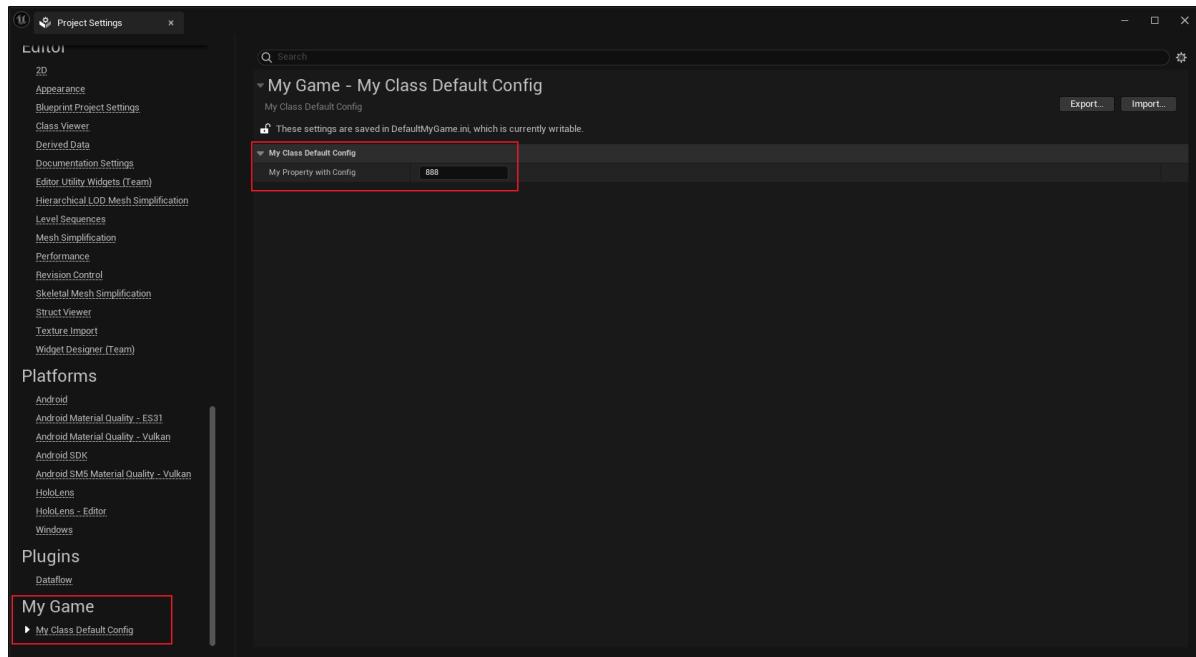
```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
};

//保存的结果:
//Config/DefaultMyGame.ini
[/Script/Insider.MyClass_DefaultConfig]
MyPropertyWithConfig=888

```

示例结果:



原理:

代码里要使用Settings->TryUpdateDefaultConfigFile();，但发现TryUpdateDefaultConfigFile不管有没有DefaultConfig都可以调用，都可以保存到Default里。因此应该调用哪个SaveConfig

(TryUpdateDefaultConfigFile, UpdateGlobalUserConfigFile, UpdateProjectUserConfigFile) 是可以手动指定的。

但是在编辑器里编辑的时候，则可以通过写好的代码来处理好逻辑。如SSettingsEditor.cpp里NotifyPostChange中调用Section->Save();则可以在内部再调用如下代码：

```

bool FSettingsSection::Save()
{
    if (ModifiedDelegate.IsBound() && !ModifiedDelegate.Execute())
    {
        return false;
    }

    if (SaveDelegate.IsBound())
    {
        return SaveDelegate.Execute();
    }

    //更新到正确的文件里
    if (SettingsObject.IsValid())

```

```

{
    if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_DefaultConfig))
    {
        SettingsObject->TryUpdateDefaultConfigFile();
    }
    else if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_GlobalUserConfig))
    {
        SettingsObject->UpdateGlobalUserConfigFile();
    }
    else if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_ProjectUserConfig))
    {
        SettingsObject->UpdateProjectUserConfigFile();
    }
    else
    {
        SettingsObject->SaveConfig();
    }

    return true;
}

return false;
}

```

EditorConfig

- 功能描述:** 用来在编辑器状态下保存信息。
- 引擎模块:** Config, Editor
- 元数据类型:** string="abc"
- 作用机制:** 在Meta中增加EditorConfig
- 常用程度:** ★

用来在编辑器状态下保存信息。

一般用在EditorTarget的Module里，用于配置相应编辑器的信息，比如列宽，收藏夹之类的，用json保存。

保存在：C:\Users\{user name}\AppData\Local\UnrealEngine\Editor。当前有：

The screenshot shows the Visual Studio Code interface. In the left pane (EXPLORER), there are two sections: OPEN EDITORS and EDITOR. Under OPEN EDITORS, 'ContentBrowser.json' is listed. Under EDITOR, 'ContentBrowser.json' is also listed. In the right pane, the file 'ContentBrowser.json' is open, displaying its JSON content. The code is as follows:

```
1  {
2      "$type": "ContentBrowserConfig",
3      "Instances":
4      [
5          "ContentBrowserTab1":
6          {
7              "$type": "ContentBrowserInstanceConfig",
8              "PathView":
9              {
10                 "$type": "PathViewConfig",
11                 "SelectedPaths": [
12                     "/Game/Class/Serialization"
13                 ]
14             },
15             "bFavoritesExpanded": false,
16             "bFilterRecursively": true,
17             "bShowCppClassFolders": true,
18             "bSearchClasses": true,
19             "bSearchAssetPaths": true,
20             "bSearchCollections": true
21         },
22         "ContentBrowserDrawer":
23         {
24             "$type": "ContentBrowserInstanceConfig",
25             "bFavoritesExpanded": false,
26             "bFilterRecursively": true,
27             "bShowCppClassFolders": true,
28             "bSearchClasses": true,
29             "bSearchAssetPaths": true,
30             "bSearchCollections": true
31         }
32     }
33 }
```

在源码里搜索后，使用的时候必须继承于基类：

```
/** Inherit from this class to simplify saving and loading properties from editor
configs. */
UCLASS()
class EDITORCONFIG_API UEditorConfigBase : public UObject
{
    GENERATED_BODY()

public:

    /** Load any properties of this class into properties marked with metadata
tag "EditorConfig" from the class's EditorConfig */
    bool LoadEditorConfig();

    /** Save any properties of this class in properties marked with metadata tag
"EditorConfig" into the class's EditorConfig. */
    bool SaveEditorConfig() const;
};
```

示例代码：

```
UCLASS(EditorConfig = "MyEditorGame")
class INSIDER_API UMyClass_EditorConfig : public UEditorConfigBase
{
public:
    GENERATED_BODY()
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditorConfig))
int32 MyPropertywithConfig = 123;
};

void UMyClass_EditorConfig_Test::TestConfigSave()
{
    //must run after editor initialization
    auto* testObject = NewObject<UMyClass_EditorConfig>(GetTransientPackage(),
TEXT("testObject_EditorConfig"));
    testObject->MyPropertywithConfig = 777;
    testObject->SaveEditorConfig();

}

void UMyClass_EditorConfig_Test::TestConfigLoad()
{
    auto* testObject = NewObject<UMyClass_EditorConfig>(GetTransientPackage(),
TEXT("testObject_EditorConfig"));
    testObject->LoadEditorConfig();
}

//运行Save后的保存结果:
C:\Users\jack.fu\AppData\Local\UnrealEngine\Editor\MyEditorGame.json

{
    "$type": "MyClass_EditorConfig",
    "MyPropertyWithConfig": 777
}

```

GlobalUserConfig

- 功能描述:** 指定保存到的配置文件层级是全局用户设置 Engine/Config/UserXXX.ini。
- 引擎模块:** Config
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中增加CLASS_GlobalUserConfig
- 关联项:** Config
- 常用程度:** ★★★

指定保存到的配置文件层级是全局用户设置 Engine/Config/UserXXX.ini。

示例代码:

属性用Config或者GlobalConfig都是可以的。

```

UCLASS(Config = MyGame, GlobalUserConfig)
class INSIDER_API UMyClass_GlobalUserConfig:public UDeveloperSettings
{
    GENERATED_BODY()
public:
    /** Gets the settings container name for the settings, either Project or
Editor */

```

```

    virtual FName GetContainerName() const override { return TEXT("Project"); }
    /** Gets the category for the settings, some high level grouping like,
Editor, Engine, Game...etc. */
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }
    /** The unique name for your section of settings, uses the class's FName. */
    virtual FName GetSectionName() const override { return TEXT("MyGlobalGame"); }
}

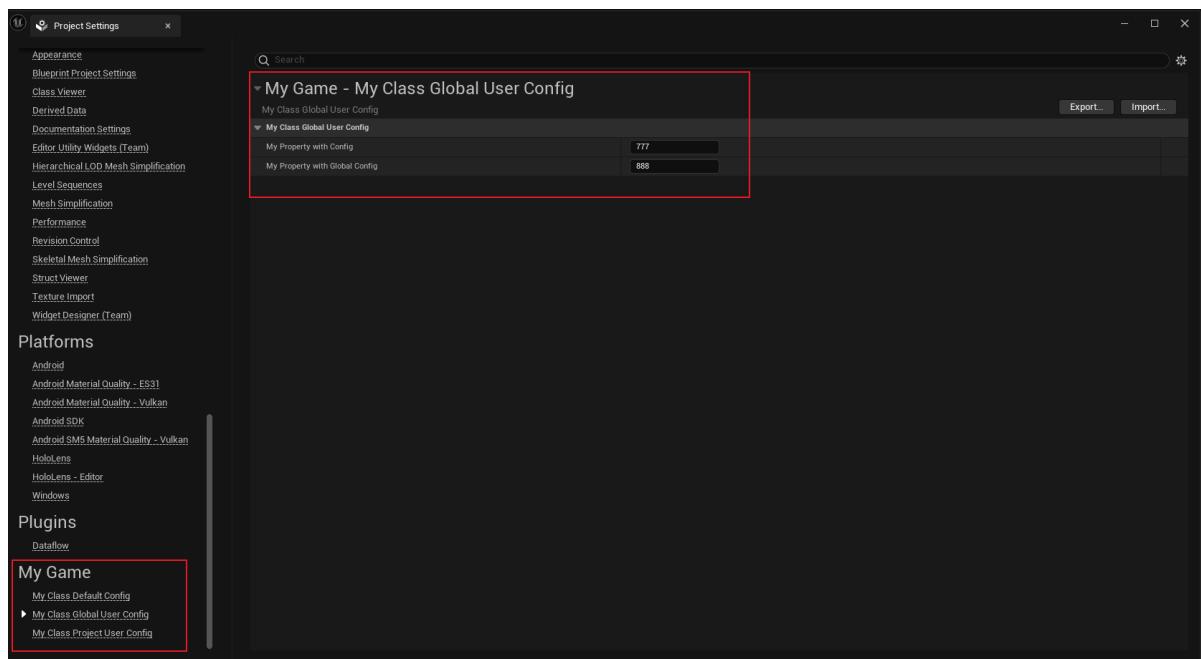
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
    int32 MyPropertyWithGlobalConfig = 456;
};

保存到C:\Users\jack.fu\AppData\Local\Unreal Engine\Engine\Config\UserMyGame.ini
[/Script/Insider.UMyClass_GlobalUserConfig]
MyPropertyWithGlobalConfig=999

```

示例效果：



源码例子：

```

UCLASS(config=Engine, globaluserconfig)
class ANDROIDPLATFORMEDITOR_API UAndroidSDKSettings : public uobject
{
public:
    GENERATED_UCLASS_BODY()

    // Location on disk of the Android SDK (falls back to ANDROID_HOME
    environment variable if this is left blank)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
    (DisplayName = "Location of Android SDK (the directory usually contains 'android-
    sdk-')"))
    FDirectoryPath SDKPath;
}

```

```

    // Location on disk of the Android NDK (falls back to NDKROOT environment
    variable if this is left blank)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "Location of Android NDK (the directory usually contains 'android-
ndk-')"))
    FDirectoryPath NDKPath;

    // Location on disk of Java (falls back to JAVA_HOME environment variable if
    this is left blank)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "Location of JAVA (the directory usually contains 'jdk')"))
    FDirectoryPath JavaPath;

    // Which SDK to package and compile Java with (a specific version or (without
    quotes) 'latest' for latest version on disk, or 'matchndk' to match the NDK API
    Level)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "SDK API Level (specific version, 'latest', or 'matchndk' - see
tooltip")))
    FString SDKAPILevel;

    // Which NDK to compile with (a specific version or (without quotes) 'latest'
    for latest version on disk). Note that choosing android-21 or later won't run on
    pre-5.0 devices.
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "NDK API Level (specific version or 'latest' - see tooltip")))
    FString NDKAPILevel;
};

```

PerObjectConfig

- 功能描述:** 在已经有config配置文件名字的情况下，指定应该按每个对象实例来存储值，而不是一个类一个存储值。
- 引擎模块:** Config
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中增加CLASS_PerObjectConfig
- 关联项:** Config
- 常用程度:** ★★★★☆

在已经有config配置文件名字的情况下，指定应该按每个对象实例来存储值，而不是一个类一个存储值。

- 此类的配置信息将按对象存储，在.ini文件中，每个对象都有一个分段，根据对象命名，格式为 [ObjectName ClassName]。
- 此说明符会传播到子类。指定该配置是对每个对象都单独保存。

示例代码：

注意ObjectName必须一致

```

UCLASS(Config = Game, PerobjectConfig)
class INSIDER_API UMyClass_PerObjectConfig :public UObject
{

```

```

GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
        int32 MyPropertywithConfig = 123;
};

void UMyClass_Config_Test::TestPerObjectConfigSave()
{
    UMyClass_PerobjectConfig* testObject1 = NewObject<UMyClass_PerobjectConfig>(GetTransientPackage(), TEXT("testObject1"));
    testObject1->MyPropertywithConfig = 456;
    testObject1->SaveConfig();

    UMyClass_PerobjectConfig* testObject2 = NewObject<UMyClass_PerobjectConfig>(GetTransientPackage(), TEXT("testObject2"));
    testObject2->MyPropertywithConfig = 789;
    testObject2->SaveConfig();

}

void UMyClass_Config_Test::TestPerObjectConfigLoad()
{
    UMyClass_PerobjectConfig* testObject1 = NewObject<UMyClass_PerobjectConfig>(GetTransientPackage(), TEXT("testObject1"));
    //testObject1->LoadConfig();      //不需要显式调用LoadConfig

    UMyClass_PerobjectConfig* testObject2 = NewObject<UMyClass_PerobjectConfig>(GetTransientPackage(), TEXT("testObject2"));
    //testObject2->LoadConfig();
}

//\Saved\Config\WindowsEditor\Game.ini
[testObject1 MyClass_PerobjectConfig]
MyPropertywithConfig=456

[testObject2 MyClass_PerobjectConfig]
MyPropertywithConfig=789

```

原理：

对象构造的末期会尝试去读取配置。

```

void FObjectInitializer::PostConstructInit()
{
    //在NewObject构造中后面会调用
    if (bIsCDO || Class->HasAnyClassFlags(CLASS_PerObjectConfig))
    {
        Obj->LoadConfig(NULL, NULL, bIsCDO ? UE::LCPF_ReadParentSections :
UE::LCPF_None);
    }
}

```

ProjectUserConfig

- **功能描述:** 指定保存到的配置文件层级是项目用户设置 Project/Config/UserXXX.ini。
- **引擎模块:** Config
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中增加CLASS_ProjectUserConfig
- **关联项:** Config
- **常用程度:** ★★★

指定保存到的配置文件层级是项目用户设置 Project/Config/UserXXX.ini。

示例代码：

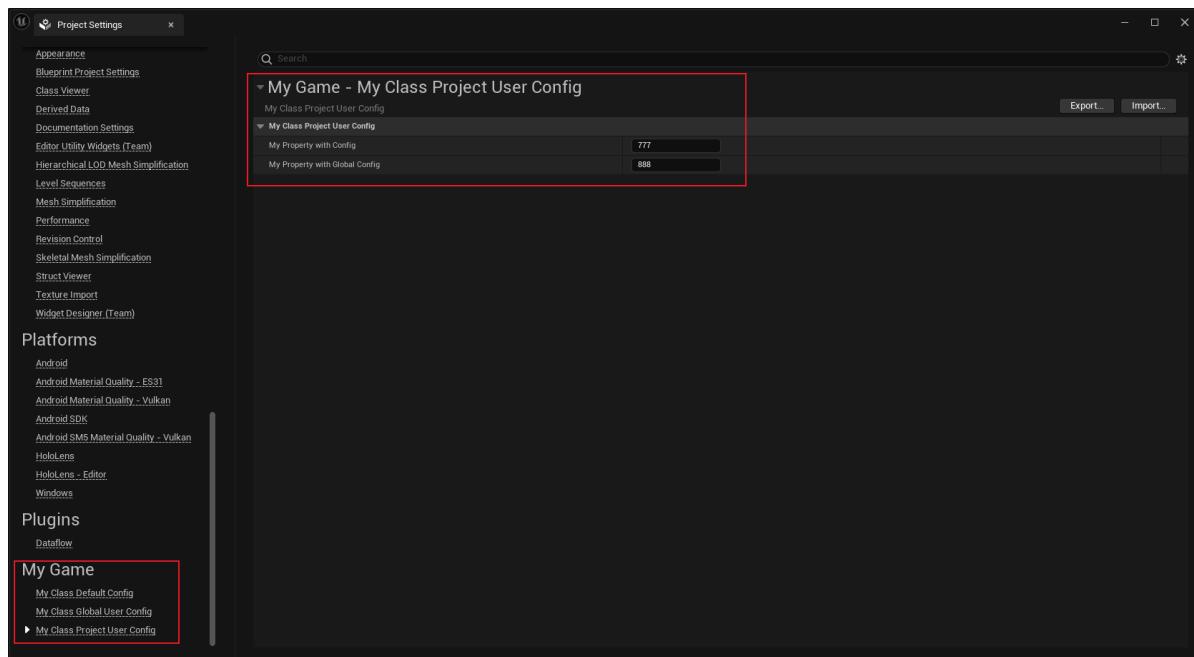
保存的目录是\Hello\Config\UserMyGame.ini

```
UCLASS(Config = MyGame, ProjectUserConfig)
class INSIDER_API UMyClass_ProjectUserConfig :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    /** Gets the settings container name for the settings, either Project or
Editor */
    virtual FName GetContainerName() const override { return TEXT("Project"); }
    /** Gets the category for the settings, some high level grouping like,
Editor, Engine, Game...etc. */
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }
    /** The unique name for your section of settings, uses the class's FName. */
    virtual FName GetSectionName() const override { return TEXT("MyProjectGame"); }
}
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
        int32 MyPropertyWithConfig = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
        int32 MyPropertyWithGlobalConfig = 456;
};

//结果: \Hello\Config\UserMyGame.ini
[/Script/Insider.MyClass_ProjectUserConfig]
MyPropertyWithConfig=777
MyPropertyWithGlobalConfig=888
```

示例效果：



在源码中搜索：

```
UCLASS(config = Engine, projectuserconfig, meta = (DisplayName = "Rendering  
Overrides (Local)"))  
class ENGINE_API URendererOverrideSettings : public UDeveloperSettings  
{  
}
```

Deprecated

- **功能描述:** 标明该类已经弃用。
- **引擎模块:** Development
- **元数据类型:** bool
- **作用机制:** 在ClassFlags添加CLASS_Deprecated、CLASS_NotPlaceable，在Meta添加DeprecationMessage、DeprecatedProperty
- **常用程度:** ★★★

标明该类已经弃用。

弃用会导致：不可被创建，不可被序列化保存，在继承列表里被过滤掉。此说明符子类会继承下来，标明子类也是废弃的。标上Deprecated 的类需要在类名前加上UDEPRECATED的显眼前缀，但是类名不会变，Actor加ADEPRECATED， UObject加UDEPRECATED_。ClassFlags里会标上CLASS_Deprecated和CLASS_NotPlaceable。注意还是可以正常NewObject使用的。而SpawnActor会失败，报错： failed because class %s is deprecated。EditInline也都会被禁止。

示例代码1：

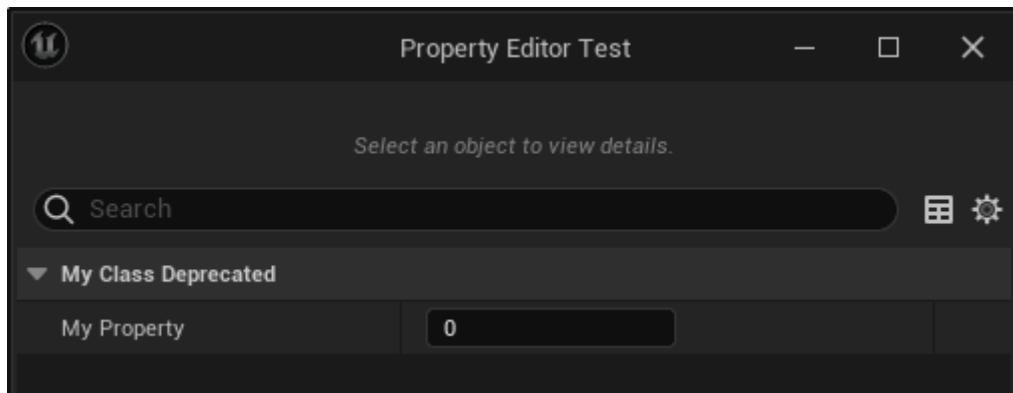
```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_Deprecated :public UObject
{
    GENERATED_BODY()
};

//改为:

UCLASS(Blueprintable, Deprecated)
class INSIDER_API UDEPRECATED_MyClass_Deprecated :public UObject
{
    GENERATED_BODY()
};
```

示例效果1：

依然可以NewObject。



示例代码2：

但要注意这个是UE的标记。源码里还看见很多UE_DEPRECATED宏的使用，则是在VS编译器级别的标记，会根据使用引用情况在编译的步骤中生成警告。

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Deprecated_Test :public UObject
{
    GENERATED_BODY()
public:

    UE_DEPRECATED(5.2, "MyClass_Deprecated has been deprecated, please remove it.")
    UDEPRECATED_MyClass_Deprecated* MyProperty_Deprecated;

    UE_DEPRECATED(5.2, "MyIntProperty has been deprecated, please remove it.")
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(DeprecatedProperty,
    DeprecationMessage = "MyIntProperty has been deprecated."))
    int MyIntProperty;

    UE_DEPRECATED(5.2, "MyClass_Deprecated has been deprecated, please remove it.")
    void MyFunc(UDEPRECATED_MyClass_Deprecated* obj){}
```

```

UFUNCTION(BlueprintCallable, meta = (DeprecatedProperty,
DeprecationMessage="MyVoidFunc has been deprecated."))
    void MyVoidFunc(){}
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Deprecated_Usage :public UObject
{
    GENERATED_BODY()
public:

    void MyFunc()
    {
        UMyClass_Deprecated_Test* obj=NewObject<UMyClass_Deprecated_Test>();
        UDEPRECATED_MyClass_Deprecated* obj2 =
        NewObject<UDEPRECATED_MyClass_Deprecated>();
        obj->MyProperty_Deprecated= obj2;
        obj->MyProperty_Deprecated->MyFunc();

        obj->MyIntProperty++;
        obj->MyFunc(obj2);
        obj->MyVoidFunc();
    }
};

```

编译警告:

```

warning C4996: 'UMyClass_Deprecated_Test::MyProperty_Deprecated':
MyClass_Deprecated has been deprecated, please remove it. Please update your code
to the new API before upgrading to the next release, otherwise your project will
no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyProperty_Deprecated':
MyClass_Deprecated has been deprecated, please remove it. Please update your code
to the new API before upgrading to the next release, otherwise your project will
no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyIntProperty': MyIntProperty has been
deprecated, please remove it. Please update your code to the new API before
upgrading to the next release, otherwise your project will no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyFunc': MyClass_Deprecated has been
deprecated, please remove it. Please update your code to the new API before
upgrading to the next release, otherwise your project will no longer compile.

```

注意如果没有UE_DEPRECATED标记，则不会生成编译警告。

UPROPERTY(EditAnywhere, BlueprintReadWrite) int MyInt2Property_DEPRECATED;
会触发:

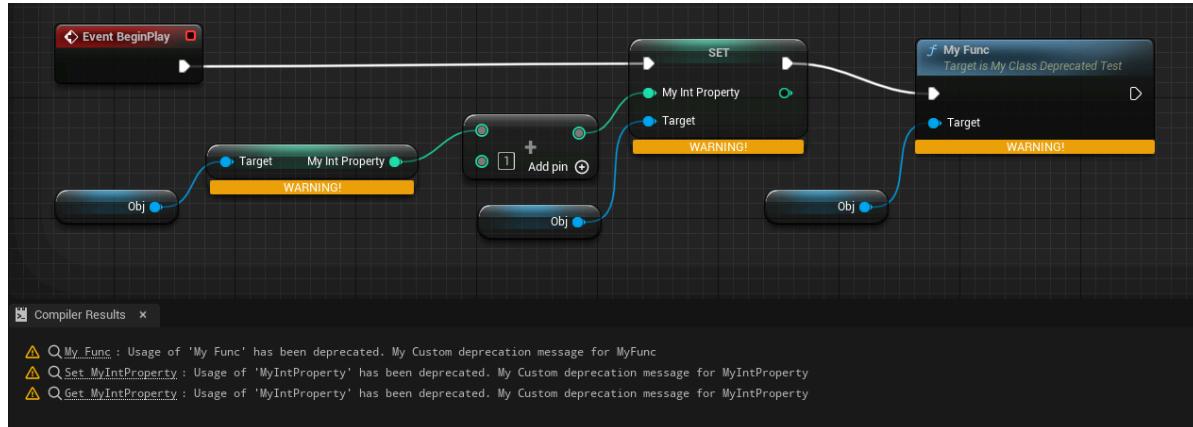
```

warning : Member variable declaration: Deprecated property
'MyInt2Property_DEPRECATED' should not be marked as blueprint visible without
having a BlueprintGetter
warning : Member variable declaration: Deprecated property
'MyInt2Property_DEPRECATED' should not be marked as blueprint writable without
having a BlueprintSetter
warning : Member variable declaration: Deprecated property
'MyInt2Property_DEPRECATED' should not be marked as visible or editable
因此只能改成:
UPROPERTY() int MyInt2Property_DEPRECATED;

```

示例效果2：

属性和函数上加上Deprecated标记后，会在BP编译的时候生成警告。注意函数是先有一个正常的函数，在BP里连接完成之后再在C++里标记DeprecatedFunction才会生成警告，否则已经Deprecated的函数是无法再在BP里调用的。



原理：

源码中有众多CLASS_Deprecated的判断，比如SpawnActor：

```
AActor* Uworld::SpawnActor( uclass* Class, FTransform const* UserTransformPtr,
const FActorSpawnParameters& SpawnParameters )
{
    if( Class->HasAnyClassFlags(CLASS_Deprecated) )
    {
        UE_LOG(LogSpawn, Warning, TEXT("SpawnActor failed because class %s is
deprecated"), *Class->GetName() );
        return NULL;
    }
}
```

EarlyAccessPreview

- 功能描述：** 标明该类是早期预览版，比试验版要更完善一些，但还是没到产品级。
- 引擎模块：** Development
- 元数据类型：** bool
- 作用机制：** 在Meta中添加DevelopmentStatus，将类标记为EarlyAccess
- 常用程度：** ★★★

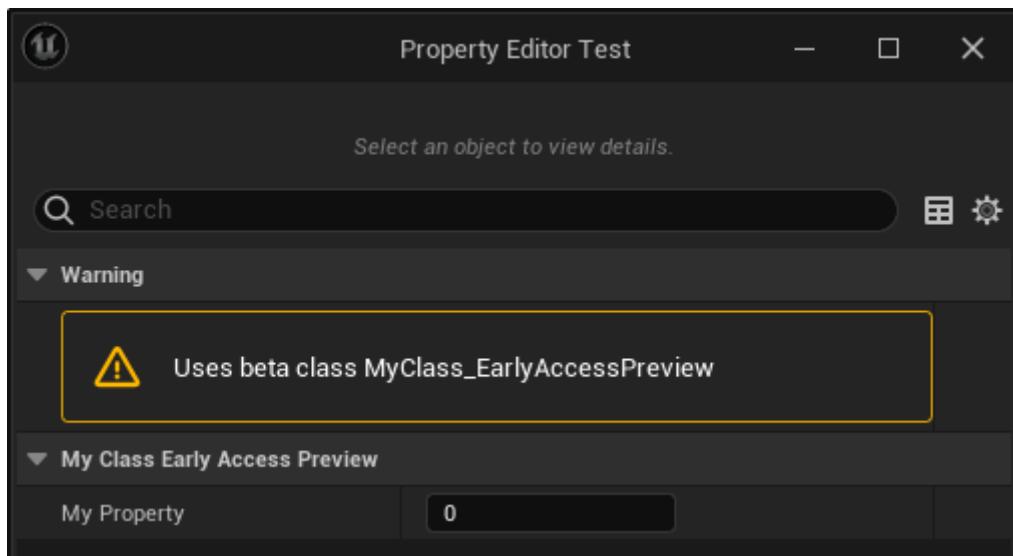
标明该类是早期预览版，比试验版要更完善一些，但还是没到产品级。

这个标记会在类的元数据上加上{ "DevelopmentStatus", "EarlyAccess" }。

示例代码：

```
//(BlueprintType = true, DevelopmentStatus = EarlyAccess, IncludePath =
Class/Display/MyClass_Deprecated.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_Deprecated.h)
UCLASS(Blueprintable, EarlyAccessPreview)
class INSIDER_API UMyClass_EarlyAccessPreview :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
    UFUNCTION(BlueprintCallable)
        void MyFunc() {}
};
```

示例结果：



Experimental

- **功能描述:** 标明该类是试验性版本，当前没有文档描述，之后有可能废弃掉。
- **引擎模块:** Development
- **元数据类型:** bool
- **作用机制:** 在Meta中添加DevelopmentStatus，将类标记为Experimental
- **常用程度:** ★★★

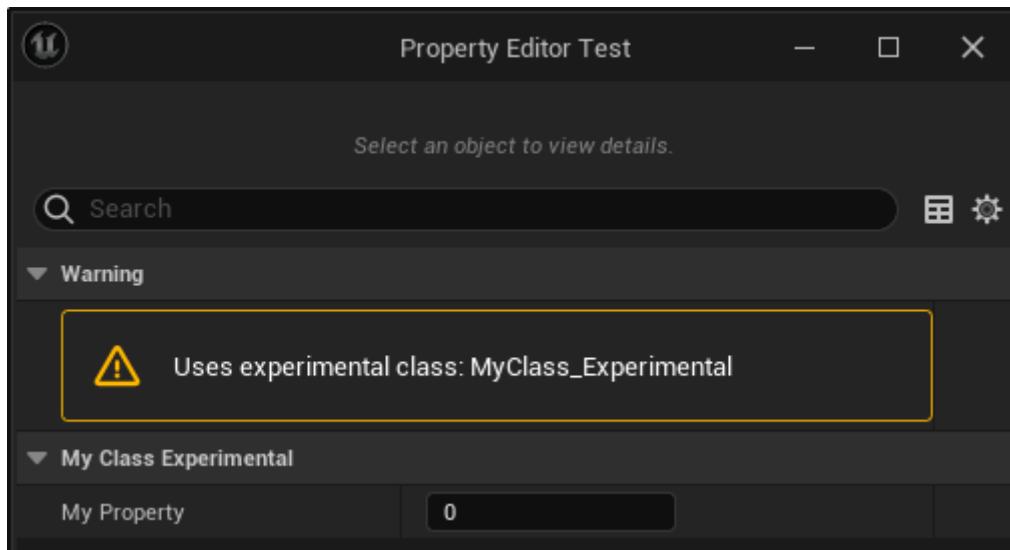
标明该类是试验性版本，当前没有文档描述，之后有可能废弃掉。

源码里的例子是Paper2D的类。这个标记会在类的元数据上加上{ "DevelopmentStatus", "Experimental" }。

示例代码：

```
/*
(BlueprintType = true, DevelopmentStatus = Experimental, IncludePath =
Class/Display/MyClass_Deprecated.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_Deprecated.h)
*/
UCLASS(Blueprintable, Experimental)
class INSIDER_API UMyClass_Experimental :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
    UFUNCTION(BlueprintCallable)
        void MyFunc();
};
```

示例效果：



DefaultToInstanced

- 功能描述：**指定该类的所有实例属性都默认是UPROPERTY(instanced)，即都默认创建新的实例，而不是对对象的引用。
- 引擎模块：** Instance
- 元数据类型：** bool
- 作用机制：** 在ClassFlags中添加CLASS_DefaultToInstanced
- 常用程度：** ★★★★

指定该类的所有实例属性都默认是UPROPERTY(instanced)，即都默认创建新的实例，而不是对对象的引用。

UPROPERTY(instanced)的含义是造成Property的CPF_InstancedReference，即为该属性创建对象实例。

所谓实例指的是为该UObject指针创建一个对象，而不是默认的去找到引擎内已有的对象的来引用。

也常常和EditInlineNew配合使用，以便在细节面板中可以创建对象实例。

UActorComponent本身就是带有DefaultToInstanced的。

示例代码：

```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_NotDefaultToInstanced :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

// ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
// CLASS_DefaultToInstanced | CLASS_TokenStreamAssembled | CLASS_Intrinsic |
// CLASS_Constructed
UCLASS(Blueprintable, DefaultToInstanced)
class INSIDER_API UMyClass_DefaultToInstanced :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

// ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_EditInlineNew |
// CLASS_RequiredAPI | CLASS_DefaultToInstanced | CLASS_TokenStreamAssembled |
// CLASS_Intrinsic | CLASS_Constructed
UCLASS(Blueprintable, DefaultToInstanced, EditInlineNew)
class INSIDER_API UMyClass_DefaultToInstanced_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

UCLASS(Blueprintable, EditInlineNew)
class INSIDER_API UMyClass_NotDefaultToInstanced_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_DefaultToInstanced_Test :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "NormalProperty")
```

```

UMyClass_NotDefaultToInstanced* MyObject_NotDefaultToInstanced;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "NormalProperty")
UMyClass_DefaultToInstanced* MyObject_DefaultToInstanced;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
"NormalProperty | Instanced")
UMyClass_NotDefaultToInstanced* MyObject_NotDefaultToInstanced_Instanced;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
"NormalProperty | Instanced")
UMyClass_DefaultToInstanced* MyObject_DefaultToInstanced_Instanced;

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "EditInlineNew")
    UMyClass_NotDefaultToInstanced_EditInlineNew*
MyObject_NotDefaultToInstanced_EditInlineNew;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "EditInlineNew")
    UMyClass_DefaultToInstanced_EditInlineNew*
MyObject_DefaultToInstanced_EditInlineNew;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
>EditInlineNew | Instanced")
    UMyClass_NotDefaultToInstanced_EditInlineNew*
MyObject_NotDefaultToInstanced_EditInlineNew_Instanced;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
>EditInlineNew | Instanced")
    UMyClass_DefaultToInstanced_EditInlineNew*
MyObject_DefaultToInstanced_EditInlineNew_Instanced;
};

```

示例效果：

- MyObject_NotDefaultToInstanced和MyObject_NotDefaultToInstanced_EditInlineNew因为属性没有Instanced的标记，因此打开是一个选择对象引用的列表。
- MyObject_DefaultToInstanced因为类上有DefaultToInstanced，因此该属性是Instanced。当然我们也可以手动给属性加上Instanced标记，正如MyObject_NotDefaultToInstanced_Instanced和MyObject_DefaultToInstanced_Instanced。出现了创建实例的窗口，但是还不能创建在细节面板里直接创建对象。
- MyObject_DefaultToInstanced_EditInlineNew,
MyObject_NotDefaultToInstanced_EditInlineNew_Instanced,
MyObject_DefaultToInstanced_EditInlineNew_Instanced这3个都可以直接在细节面板创建对象实例。是因为这个类本身要有EditInlineNew，另外这个属性要有Instanced（要嘛在该类上设置DefaultToInstanced以此该类的所有属性都自动是Instanced，或者在属性上单个设置Instanced）



原理：

```
UObject* FObjectInstancingGraph::InstancePropertyValue(UObject* SubObjectTemplate, UObject* CurrentValue, UObject* Owner, EInstancePropertyValueFlags Flags)
{
    if (CurrentValue->GetClass()->HasAnyClassFlags(CLASS_DefaultToInstanced))
    {
        bCausesInstancing = true; // these are always instanced no matter what
    }
}
```

EditInlineNew

- **功能描述：** 指定该类的对象可以在属性细节面板里直接内联创建，要和属性的Instanced配合。
- **引擎模块：** Instance
- **元数据类型：** bool
- **作用机制：** 在ClassFlags中添加CLASS_EditInlineNew
- **关联项：** NotEditInlineNew (NotEditInlineNew.md)
- **常用程度：** ★★★★☆

指定该类的对象可以在属性细节面板里直接内联创建。

如果想在细节面板里直接创建对象，属性上也必须先标记Instanced或ShowInnerProperties。

EditInlineNew主要是用在UObject的子类上，一般不标EditInlineNew的是用在Actor或资产的引用上。注意EditInlineNew是表明增加从属性细节面板里直接创建对象实例的能力，而非限制只能在属性细节面板里创建，当然也可以自己手动NewObject再赋值给对象引用属性。

这个跟UPROPERTY上的Instanced能力是独立的。如果UCLASS上不加EditInlineNew，但是属性上加上Instanced，则在手动NewObject赋值该属性后，该属性也会展开内部属性来提供编辑功能。因为Instanced的属性会自动的在property上加上EditInline的meta。

此说明符会传播到所有子类；子类可通过 NotEditInlineNew 说明符覆盖它。

示例代码：

```
UCLASS(Blueprintable, EditInlineNew)
class INSIDER_API UMyClass_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

UCLASS(Blueprintable, NotEditInlineNew)
class INSIDER_API UMyClass_NotEditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
```

```

        int32 MyProperty;
    };

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Edit_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = InstancedProperty)
        UMyClass_EditInlineNew* MyEditInlineNew;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = InstancedProperty)
        UMyClass_NotEditInlineNew* MyNotEditInlineNew;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NormalProperty)
        UMyClass_EditInlineNew* MyEditInlineNew_NotInstanced;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NormalProperty)
        UMyClass_NotEditInlineNew* MyNotEditInlineNew_NotInstanced;
};

```

示例效果：

EditInlineNew支持直接C++或BP子类创建对象实例，然后在上面编辑实例。

而NotEditInlineNew的属性则无法找到支持的类来创建对象。

如果属性上没有Instanced则只能尝试去引用（找不到对象）。



原理：

判断该类是否有CLASS_EditInlineNew来决定是否可内联创建编辑。

```

template <typename TClass, typename TIsChildOfFunction>
bool FPropertyEditorInlineClassFilter::IsClassAllowedHelper(TClass InClass,
TIsChildOfFunction IsClassChildOf, TSharedRef< FClassViewerFilterFuncs >
InFilterFuncs)
{
    const bool bMatchesFlags = InClass->HasAnyClassFlags(CLASS_EditInlineNew) &&
        !InClass->HasAnyClassFlags(CLASS_Hidden | CLASS_HideDropDown |
CLASS_Deprecated) &&
        (bAllowAbstract || !InClass->HasAnyClassFlags(CLASS_Abstract));
}

```

NotEditInlineNew

- 功能描述：**不能通过Editinline按钮创建
- 引擎模块：**Instance
- 元数据类型：**bool
- 作用机制：**在ClassFlags中移除CLASS_EditInlineNew
- 关联项：**EditInlineNew (EditInlineNew.md)

- 常用程度：★

Within

- **功能描述：** 指定对象创建的时候必须依赖于OuterClassName的对象作为Outer。
- **引擎模块：** Instance
- **元数据类型：** string="abc"
- **作用机制：** 保存在UClass* UClass::ClassWithin=XXX的XXX中
- **常用程度：** ★★★

指定对象创建的时候必须依赖于OuterClassName的对象作为Outer。

此类的对象无法在OuterClassName对象的实例之外存在。这意味着，要创建此类的对象，需要提供OuterClassName的一个实例作为其Outer对象。

本类在这种情况下是用来当做子对象来使用的。

示例代码：

```
UCLASS(Within= MyClass_Within_Outer)
class INSIDER_API UMyClass_Within :public UObject
{
    GENERATED_BODY()
};

UCLASS()
class INSIDER_API UMyClass_Within_Outer :public UObject
{
    GENERATED_BODY()
public:
};
```

示例结果：

```
//错误! Fatal error: Object MyClass_Within None created in Package instead of
MyClass_Within_Outer
UMyClass_Within* obj=NewObject<UMyClass_Within>();

//正确:
UMyClass_Within_Outer* objouter = NewObject<UMyClass_Within_Outer>();
UMyClass_Within* obj=NewObject<UMyClass_Within>(objouter);
```

原理：

生成的UClass的字段： UClass* ClassWithin会保存这个信息，然后在创建的时候StaticAllocateObject会测试 check(bCreatingCDO || !InOuter || InOuter->IsA(InClass->ClassWithin))。因此需要先创建Within的对象。

```

bool staticAllocateObjectErrorTests( const UClass* InClass, UObject* InOuter,
FName InName, EObjectFlags InFlags)
{
    if ( (InFlags & (RF_ClassDefaultObject|RF_ArchetypeObject)) == 0 )
    {
        if ( InOuter != NULL && !InOuter->IsA(InClass->ClassWithin) )
        {
            UE_LOG(Log UObject Globals, Fatal, TEXT("Object %s %s created in %s instead of %s"), *InClass->GetName(),
*InName.ToString(), *InOuter->GetClass()->GetName(), *InClass->ClassWithin-
>GetName() );
            return true;
        }
    }
}

```

在源码里可以搜索到很多Within的用法

```

UCLASS(Within=Engine, config=Engine, transient)
class ENGINE_API ULocalPlayer

UCLASS(Abstract, DefaultToInstanced, Within=UserWidget)
class UMG_API UUserWidgetExtension : public UObject
{

```

ConversionRoot

- 功能描述:** 在场景编辑器里允许Actor在自身以及子类之间做转换
- 引擎模块:** Scene
- 元数据类型:** bool
- 作用机制:** 在Meta中增加IsConversionRoot
- 常用程度:** ★

一般是用在Actor上，在Actor转换的时候用来限制转换的级别。比如ASkeletalMeshActor, AStaticMeshActor等。

常常ComponentWrapperClass一起出现。

根据代码来说，meta中的IsConversionRoot会限制只传达到这一层，不继续往根上查找。

只有配有ConversionRoot的Actor才会允许Convert Actor，否则是禁用的。

示例代码：

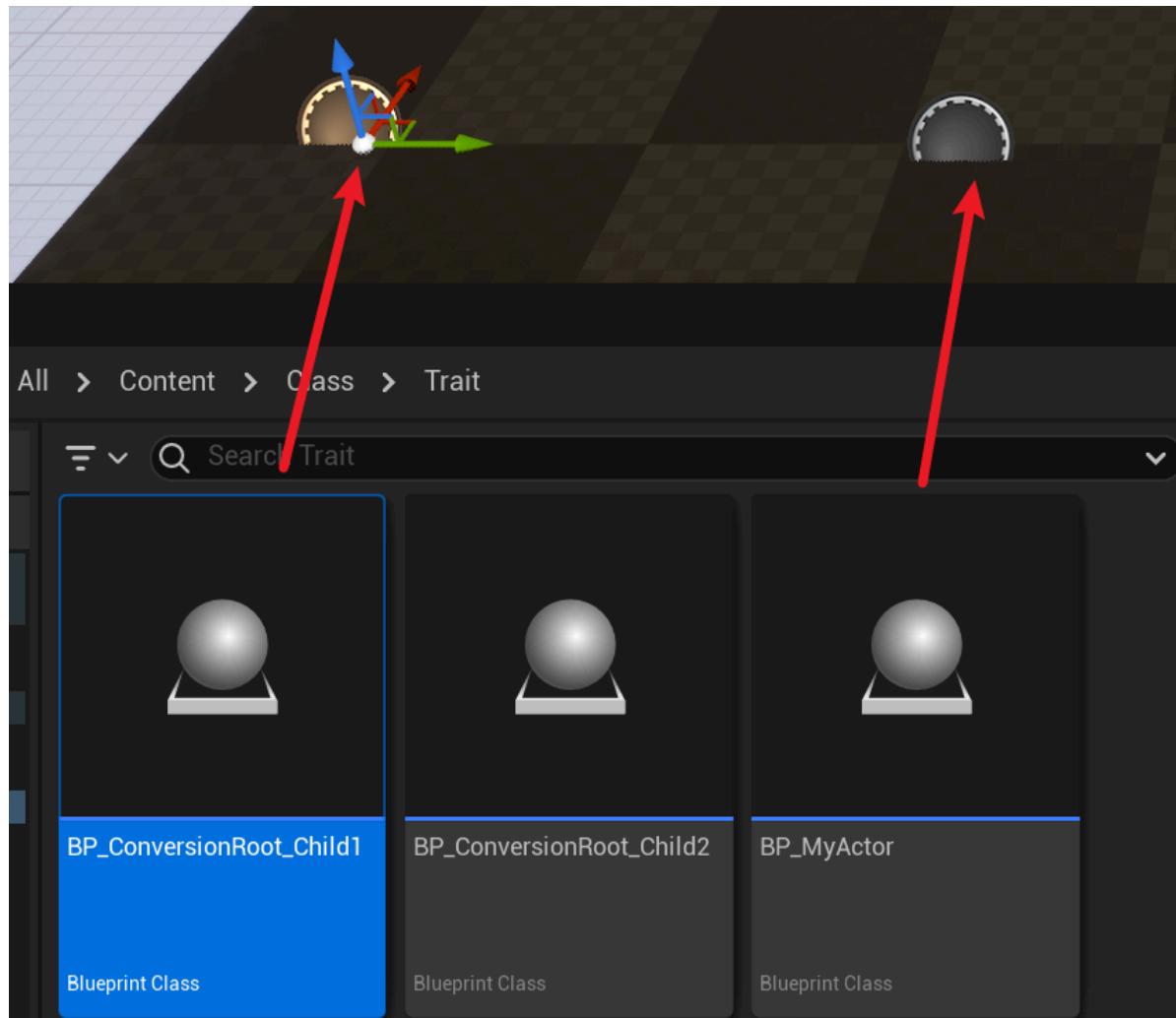
```

//(BlueprintType = true, IncludePath = Class/Trait/MyClass_ConversionRoot.h,
IsBlueprintBase = true, IsConversionRoot = true, ModuleRelativePath =
Class/Trait/MyClass_ConversionRoot.h)
UCLASS(Blueprintable, BlueprintType, ConversionRoot)
class INSIDER_API AMyActor_ConversionRoot :public AActor
{
    GENERATED_BODY()
};

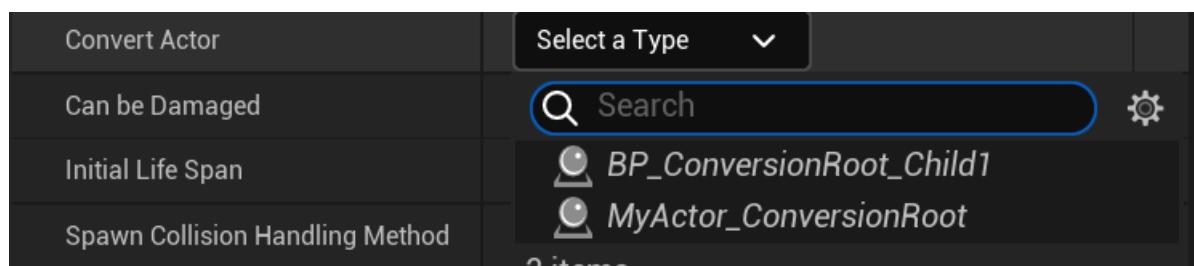
```

示例效果：

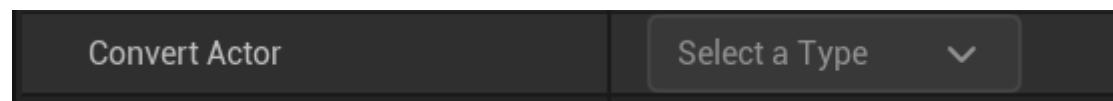
在蓝图中创建其子类BP_ConversionRoot_Child1和BP_ConversionRoot_Child2。然后把BP_ConversionRoot_Child1拖放进场景里创建个Actor，也创建个普通的蓝图Actor作为对比。



在关卡中选择Child1，会允许ConvertActor，在ConversionRoot的自身以及所有子类之间做转换。



如果是普通的Actor，因为没有定义ConversionRoot，则不能做转换。



原理：

在关卡中的Actor选择：关卡中选择一个Actor，然后DetailsPanel里会显示ConverActor属性栏，可以选择另外一个Actor来进行改变。

TSharedRef FActorDetails::MakeConvertMenu(const FSelectedActorInfo& SelectedActorInfo)

这个函数就是用来创建Select Type的Combo Button的菜单的。内部会调用

CreateClassPickerConvertActorFilter:

```

uclass* FActorDetails::GetConversionRoot( uclass* InCurrentClass ) const
{
    uclass* ParentClass = InCurrentClass;

    while(ParentClass)
    {
        if( ParentClass->GetBoolMetaData(FName(TEXT("IsConversionRoot")))) )
        {
            break;
        }
        ParentClass = ParentClass->GetSuperClass();
    }

    return ParentClass;
}

void FActorDetails::CreateClassPickerConvertActorFilter(const
TweakObjectPtr<AActor> ConvertActor, class FClassViewerInitializationOptions*
ClassPickerOptions)
Filter->AllowedChildofRelationship.Add(RootConversionClass); //限定这个基类以下的其他
子类

```

NotPlaceable

- **功能描述:** 标明该Actor不可被放置在关卡里
- **引擎模块:** Behavior
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中添加CLASS_NotPlaceable
- **关联项:** Placeable (Placeable.md)
- **常用程度:** ★★★

标明该Actor不可被放置在关卡里，没法拖放到场景里。使继承自基类的Placeable说明符无效。会在ClassFlagss里标记上CLASS_NotPlaceable，这个标记是可以继承的，意味着其所有的子类默认都不可放置。例如AWorldSettings其实就是一个notplaceable的Actor。

但是注意该类依然可以通过SpawnActor动态生成到关卡中。

NotPlaceable的类是不出现在PlaceMode的类选择里去的。

示例代码:

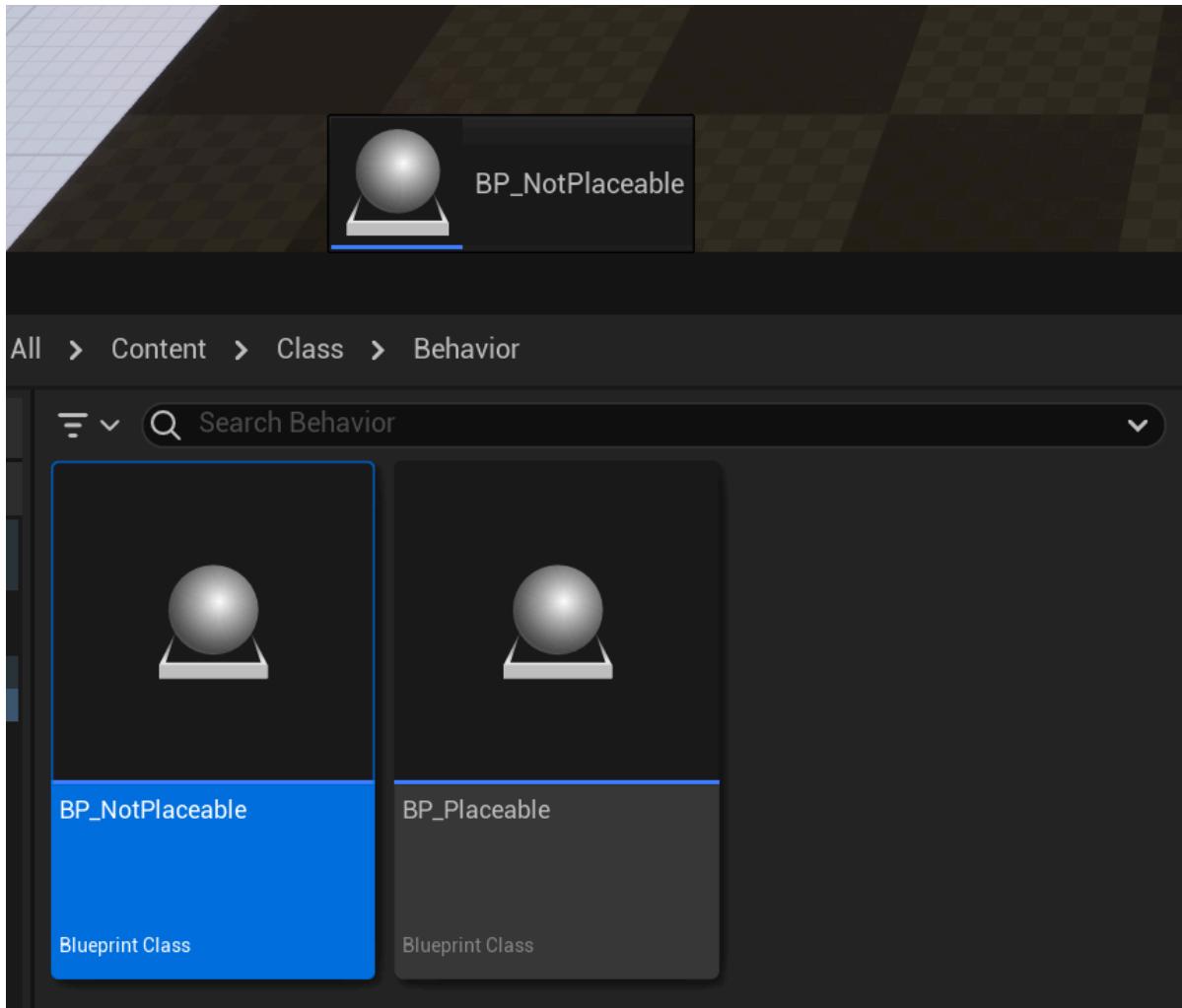
```

UCLASS(Blueprintable, BlueprintType, NotPlaceable)
class INSIDER_API AMyActor_NotPlaceable :public AActor
{
    GENERATED_BODY()
};

```

示例效果:

拖动到场景里会发现不能创建Actor。



原理：

如果直接是C++类AMyActor_NotPlaceable，是可以直接从ContentBrowser拖到场景里去的。看源码可知，只有BP继承下来的子类才有受到这个限制。

```
TArray<AActor*> FLevelEditorViewportClient::TryPlacingActorFromObject( ULevel* InLevel, UObject* ObjToUse, bool bSelectActors, EObjectFlags ObjectFlags, UActorFactory* FactoryToUse, const FName Name, const FViewportCursorLocation* Cursor )  
{  
  
    bool bPlace = true;  
    if (ObjectClass->IsChildOf(UBlueprint::StaticClass()))  
    {  
        UBlueprint* BlueprintObj = StaticCast<UBlueprint*>(ObjToUse);  
        bPlace = BlueprintObj->GeneratedClass != NULL;  
        if(bPlace)  
        {  
            check(BlueprintObj->ParentClass == BlueprintObj->GeneratedClass->GetSuperClass());  
            if (BlueprintObj->GeneratedClass->HasAnyClassFlags(CLASS_NotPlaceable  
| CLASS_Abstract))  
            {  
                bPlace = false;  
            }  
        }  
    }  
}
```

```

if (bPlace)
{
    PlacedActor = FActorFactoryAssetProxy::AddActorForAsset( ObjToUse,
    bSelectActors, ObjectFlags, FactoryToUse, Name );
    if ( PlacedActor != NULL )
    {
        PlacedActors.Add(PlacedActor);
        PlacedActor->PostEditMove(true);
    }
}
}

```

Placeable

- **功能描述:** 标明该Actor可以放置在关卡里。
- **引擎模块:** Scene
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中移除CLASS_NotPlaceable
- **关联项:** NotPlaceable
- **常用程度:** ★★★

标明该Actor可以放置在关卡里。

默认情况下是placeable的，因此源码里目前没有用到Placeable的地方。

子类可使用NotPlaceable说明符覆盖此标志，正如AInfo之类的上面自己设置NotPlaceable。

指示可在编辑器中创建此类，而且可将此类放置到关卡、UI场景或蓝图（取决于类类型）中。此标志会传播到所有子类；

placeable没法清除父类的notplaceable标记。

示例代码：

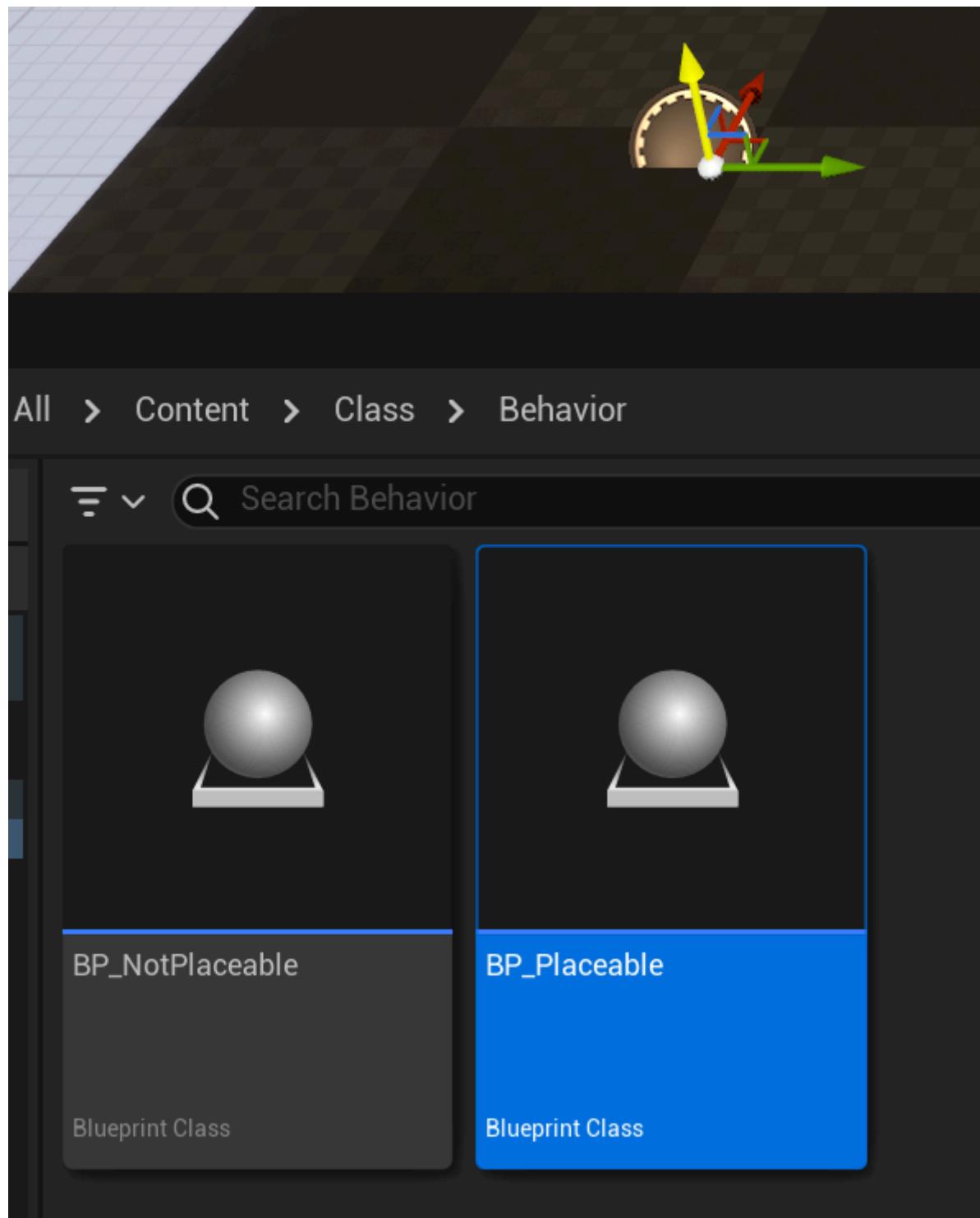
```

UCLASS(Blueprintable, BlueprintType, placeable)
class INSIDER_API AMyActor_Placeable :public AMyActor_NotPlaceable
{
    GENERATED_BODY()
};

error : The 'placeable' specifier cannot override a 'nonplaceable' base class.
Classes are assumed to be placeable by default. Consider whether using the
'abstract' specifier on the base class would work.

```

示例效果：



MatchedSerializers

- **功能描述:** 指定类支持文本结构序列化
- **引擎模块:** Serialization
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中增加CLASS_MatchedSerializers，在Meta中添加MatchedSerializers
- **常用程度:** 0

该标识符只允许在NoExportTypes.h中使用，属于是引擎自用的内部标识符。

基本上大部分的类都拥有该标记，除了自身不导出的类，一般包括NoExportTypes.h定义的（除非手动加上MatchedSerializers，比如UObject），或者靠DECLARE_CLASS_INTRINSIC直接在源码里定义的元数据。

因此实际上大部分的类都拥有该标记。因为在UHT中只要不是NoExport的，就会自动的加上这个标记。

```
// Force the MatchedSerializers on for anything being exported
if (!ClassExportFlags.HasAnyFlags(UhtClassExportFlags.NoExport))
{
    ClassFlags |= EClassFlags::MatchedSerializers;
}
```

结构化序列化器：

如果一个类支持文本格式，则StructuredArchive的结构的意思是会把类里的字段树形展开来序列化展示出来，从而方便人类理解。而如果不支持文本格式，则会把所有的字段值压进一个二进制buffer里（Data字段），这也是runtime时候用的方式。

测试代码：

```
UCLASS(Blueprintable, BlueprintType, editinlinenew)
class INSIDER_API UMyClass_MatchedSerializersSub :public UObject
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_MatchedSerializersTestAsset:public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
    UMyClass_MatchedSerializersSub* SubObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UStruct* MyStructType;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UClass* MyClassType;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UEnum* MyEnumType;
};

void UMyClass_MatchedSerializers_Test::ApplyClassFlag()
{
```

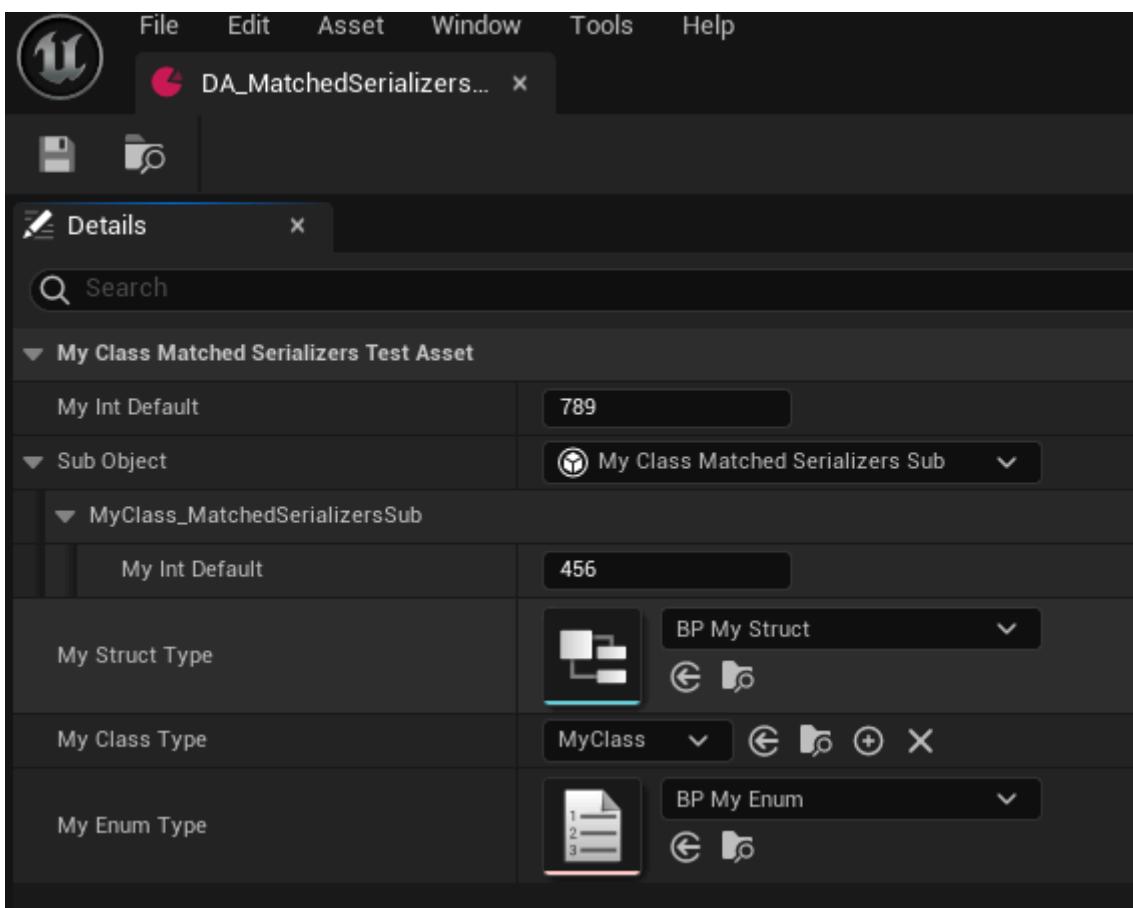
```

UMyClass_MatchedSerializersTestAsset::StaticClass()>ClassFlags |=
CLASS_MatchedSerializers;
UMyClass_MatchedSerializersSub::StaticClass()>ClassFlags |=
CLASS_MatchedSerializers;
}

void UMyClass_MatchedSerializers_Test::RemoveClassFlag()
{
    UMyClass_MatchedSerializersTestAsset::StaticClass()>ClassFlags &=
~CLASS_MatchedSerializers;
    UMyClass_MatchedSerializersSub::StaticClass()>ClassFlags &=
~CLASS_MatchedSerializers;
}

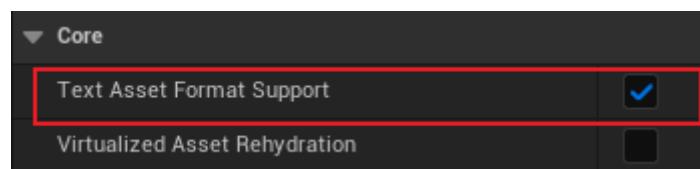
```

在编辑器中创建测试数据Asset

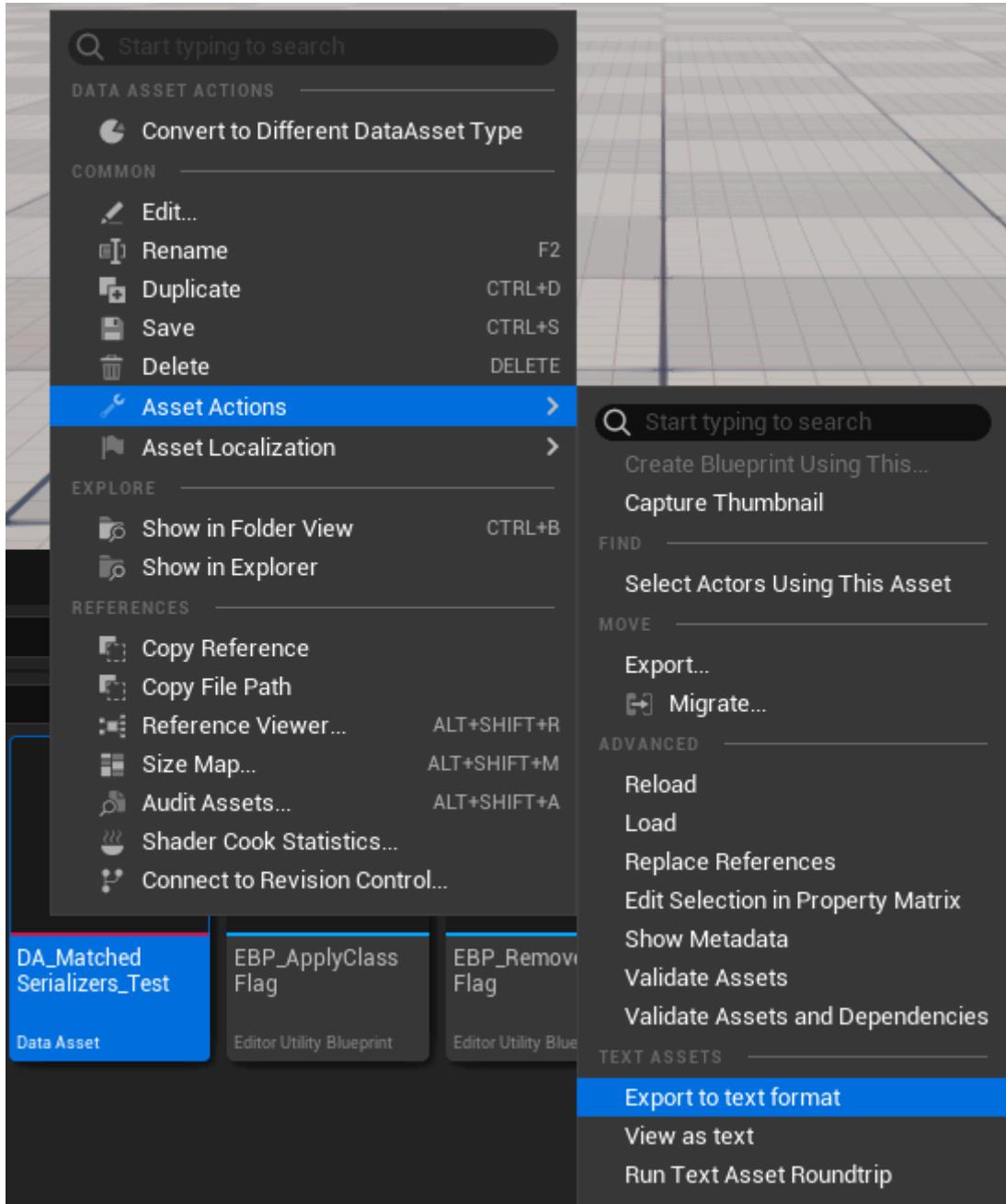


然后在Editor选项里打开

TextAssetFormatSupport(UEditorExperimentalSettings::bTextAssetFormatSupport)



然后在资产上就出现3个菜单支持把资产导出为文本。



ExportToTextFormat会在蓝图资产的同目录生成一个.utxt的文件，格式为json。通过动态的增删CLASS_MatchedSerializers这个标记来对比这个标记产生的差异：

```
"RootMetaDataMember": [
    {
        "Namespace": "PackageLocalizationNamespace",
        "B292B0ECCCFA8D5473A903D7AEE63B37"
    }
],
"DA_MatchedSerializers_Test: MyClass_MatchedSerializersSub_0": {
    "Properties": [
        {
            "SerializationControlExtensions": 0,
            "Value": {
                "MyInt_Default": {
                    "Type": "Int32Property",
                    "PropertyExtensions": 0,
                    "Value": 456
                }
            }
        },
        {
            "Name": "DA_MatchedSerializers_Test",
            "Properties": [
                {
                    "SerializationControlExtensions": 0,
                    "Value": {
                        "MyInt_Default": {
                            "Type": "Int32Property",
                            "PropertyExtensions": 0,
                            "Value": 789
                        }
                    }
                },
                {
                    "SubObject": {
                        "Type": "ObjectProperty",
                        "PropertyExtensions": 0,
                        "Value": "2"
                    }
                },
                {
                    "MyStructType": {
                        "Type": "ObjectProperty",
                        "PropertyExtensions": 0,
                        "Value": "-11"
                    }
                },
                {
                    "MyClassType": {
                        "Type": "ObjectProperty",
                        "PropertyExtensions": 0,
                        "Value": "4"
                    }
                },
                {
                    "MyEnumType": {
                        "Type": "ObjectProperty",
                        "PropertyExtensions": 0,
                        "Value": "-10"
                    }
                }
            ]
        }
    ],
    "Summary": {
        "RootMetaDataMember": [
            {
                "Namespace": "PackageLocalizationNamespace",
                "B292B0ECCCFA8D5473A903D7AEE63B37"
            }
        ],
        "DA_MatchedSerializers_Test: MyClass_MatchedSerializersSub_0": {
            "Name": "DA_MatchedSerializers_Test",
            "Properties": [
                {
                    "Digest": "f49b6e5b3eff8d749743b49b447842bb36fd1",
                    "Base": [
                        "2",
                        "-11",
                        "4",
                        "-10"
                    ]
                },
                {
                    "Name": "DA_MatchedSerializers_Test",
                    "Properties": [
                        {
                            "MyInt_Default": {
                                "Type": "Int32Property",
                                "PropertyExtensions": 0,
                                "Value": 789
                            }
                        },
                        {
                            "SubObject": {
                                "Type": "ObjectProperty",
                                "PropertyExtensions": 0,
                                "Value": "2"
                            }
                        },
                        {
                            "MyStructType": {
                                "Type": "ObjectProperty",
                                "PropertyExtensions": 0,
                                "Value": "-11"
                            }
                        },
                        {
                            "MyClassType": {
                                "Type": "ObjectProperty",
                                "PropertyExtensions": 0,
                                "Value": "4"
                            }
                        },
                        {
                            "MyEnumType": {
                                "Type": "ObjectProperty",
                                "PropertyExtensions": 0,
                                "Value": "-10"
                            }
                        }
                    ]
                }
            ],
            "Summary": {
                "RootMetaDataMember": [
                    {
                        "Namespace": "PackageLocalizationNamespace",
                        "B292B0ECCCFA8D5473A903D7AEE63B37"
                    }
                ]
            }
        }
    }
}
```

可以发现，序列化出来的内容有明显的差异，不带有CLASS_MatchedSerializers标记的产生的右侧结果，把所有的字段值压进一个二进制buffer里（Data字段）。

内部机制原理：

CLASS_MatchedSerializers这个标记在UClass::IsSafeToSerializeToStructuredArchives中被使用，标明采用结构序列化器。是否支持文本导入导出，只在编辑器情况下使用。

在发生作用的只有SavePackage2.cpp和LinkerLoad.cpp，因此是只发生在保存UPackage的时候，作为子类对象。所以不能用简单的内存里Archive序列化来进行测试。

```
bool UClass::IsSafeToSerializeToStructuredArchives(UClass* InClass)
{
    while (InClass)
    {
        if (!InClass->HasAnyClassFlags(CLASS_MatchedSerializers))
        {
            return false;
        }
        InClass = InClass->GetSuperClass();
    }
    return true;
}

//LinkerLoad.cpp
bool bClassSupportsTextFormat =
UClass::IsSafeToSerializeToStructuredArchives(Object->GetClass());
if (IsTextFormat()) //如果Ar序列化是文本格式
{
    FStructuredArchiveslot Exportslot = GetExportslot(Export.ThisIndex);

    if (bClassSupportsTextFormat) //如果类本身支持文本格式
    {
        Object->GetClass()->SerializeDefaultObject(Object, Exportslot);
    }
    else
    {
        FStructuredArchiveChildReader ChildReader(Exportslot);
        FArchiveUObjectFromStructuredArchive
Adapter(ChildReader.GetRoot());
        Object->GetClass()->SerializeDefaultObject(Object,
Adapter.GetArchive());
    }
}

//SavePackage2.cpp
#if WITH_EDITOR
    bool bSupportsText =
UClass::IsSafeToSerializeToStructuredArchives(Export.Object->GetClass());
#else
    bool bSupportsText = false;
#endif

if (bSupportsText)
{
    Export.Object->GetClass()->SerializeDefaultObject(Export.Object,
Exportslot);
}
```

```

else
{
    FArchiveUObjectFromStructuredArchive Adapter(ExportSlot);
    Export.Object->GetClass()->SerializeDefaultObject(Export.Object,
Adapter.GetArchive());
    Adapter.Close();
}

```

文本格式只在编辑器环境下生效。

可以从源码看到，如果类本身支持文本格式序列化，则在Ar是文本格式的时候，直接可以序列化，采用默认的SerializeTaggedProperties。否则得采用FArchiveUObjectFromStructuredArchive 来适配一下，把对象指针转换为object path+ int32 Index的组合。

在引擎中打印出所有包含或不包含CLASS_MatchedSerializers的类，发现UStruct继承链下面的类开始包含（但是UClass却不包含），而上面UField的类则不包含，比如各种Property。类列表见Doc下txt文件。

NonTransient

- 功能描述：**使继承自基类的Transient说明符无效。
- 引擎模块：**Serialization
- 元数据类型：**bool
- 作用机制：**在ClassFlags中移除CLASS_Transient
- 关联项：**Transient
- 常用程度：**★★★

Optional

- 功能描述：**标记该类的对象是可选的，在Cooking的时候可以选择是否要忽略保存它们。
- 引擎模块：**Serialization
- 作用机制：**在ClassFlags中添加CLASS_Optional
- 常用程度：**★

标记该类的对象是可选的，在Cooking的时候可以选择是否要忽略保存它们。

- 一般为EditorOnly的数据，如MetaData等，在游戏运行时不存在，保存在其他的特定文件中。
- Optional的对象一般也包在WITH_EDITORONLY_DATA宏里，只在编辑器下使用。
- 引擎在cook的时候，会根据EDITOROPTIONAL的配置来加上SAVE_Optional，从而选择是否一起序列化该对象值，比如metadata。

示例代码：

```

//ClassFlags: CLASS_Optional | CLASS_MatchedSerializers | CLASS_Native |
CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic |
CLASS_Constructed
UCLASS(Optional)
class INSIDER_API UMyClass_Optional :public UObject
{
    GENERATED_BODY()
}

```

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty = 123;
};

UCLASS()
class INSIDER_API UMyClass_NotOptional :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty = 123;
};

UCLASS()
class INSIDER_API UMyClass_Optional_Test :public UObject
{
    GENERATED_BODY()
public:

#if WITH_EDITORONLY_DATA
    UPROPERTY()
        UMyClass_Optional* MyOptionalObject;

#endif // WITH_EDITORONLY_DATA

public:
    UPROPERTY()
        UMyClass_NotOptional* MyNotOptionalObject;
public:
    static void CreatePackageAndSave();
    static void LoadPackageAndTest();
};

void UMyClass_Optional_Test::CreatePackageAndSave()
{
    FString packageName = TEXT("/Game/MyOptionTestPackage");
    FString assetPath = FPackageName::LongPackageNameToFilename(packageName,
FPackageName::GetAssetPackageExtension());

    IFileManager::Get().Delete(*assetPath, false, true);

    UPackage* package = CreatePackage(*packageName);
    FSavePackageArgs saveArgs{};
    //saveArgs.TopLevelFlags = EObjectFlags::RF_Public | 
EObjectFlags::RF_Standalone;
    saveArgs.Error = GError;
    saveArgs.SaveFlags=SAVE_NoError;

    //SAVE_Optional = 0x00008000, //< Indicate that we want to save optional
exports. This flag is only valid while cooking. Optional exports are filtered if
not specified during cooking.

    UMyClass_Optional_Test* testObject = NewObject<UMyClass_Optional_Test>
(package, TEXT("testObject"));
}

```

```

#if WITH_EDITORONLY_DATA
    testObject->MyOptionalObject = NewObject<UMyClass_Optional>(testObject,
TEXT("MyOptionalObject"));
    testObject->MyOptionalObject->MyProperty = 456;
#endif

    testObject->MyNotOptionalObject = NewObject<UMyClass_NotOptional>(testObject,
TEXT("MyNotOptionalObject"));

    testObject->MyNotOptionalObject->MyProperty = 456;

    FString str = UIInsiderSubsystem::Get().PrintObject(package,
EInsiderPrintFlags::All);
    FString str2 = UIInsiderSubsystem::Get().PrintObject(testObject,
EInsiderPrintFlags::All);
    FString str3 =
UIInsiderSubsystem::Get().PrintObject(UMyClass_Optional::StaticClass(),
EInsiderPrintFlags::All);
    FString str4 =
UIInsiderSubsystem::Get().PrintObject(UMyClass_NotOptional::StaticClass(),
EInsiderPrintFlags::All);

    bool result = UPackage::SavePackage(package, testObject, *assetPath,
saveArgs);

}

void UMyClass_Optional_Test::LoadPackageAndTest()
{
    FString packageName = TEXT("/Game/MyOptionTestPackage");
    FString assetPath = FPackageName::LongPackageNameToFilename(packageName,
FPackageName::GetAssetPackageExtension());

    UPackage* package = LoadPackage(nullptr, *assetPath, LOAD_None);
    package->FullyLoad();

    UMyClass_Optional_Test* newTestObject = LoadObject<UMyClass_Optional_Test>
(package, TEXT("testObject"), *assetPath);
    //UMyClass_Transient_Test* newTestObject = nullptr;

    /*const TArray<FObjectExport>& exportMap = package->GetLinker()->ExportMap;
for (const auto& objExport : exportMap)
{
    if (objExport.ObjectName == TEXT("testObject"))
    {
        newTestObject = Cast<UMyClass_Transient_Test>(objExport.Object);
        break;
    }
}*/
    FString str = UIInsiderSubsystem::Get().PrintObject(package,
EInsiderPrintFlags::All);
}

```

示例效果：

正常的SavePackage发现是没有作用的，依然会序列化保存。特殊的保存方式在Cook阶段，本例就没有专门测试了。

newTestObject	0x0000095c3e7ec640 (Name="testObject") (Name="testObject")	UMyClass_Optional_Test *
↳ UObject	0x0000095c3e9750a0 (Name="MyOptionalObject") (Name="MyOptionalObject")	UObject
↳ MyOptionalObject	456	TObjectPtr<UMyClass_Optional>
↳ UObject	0x0000095c3e9750a0 (Name="MyOptionalObject") (Name="MyOptionalObject")	UObject
↳ MyProperty	456	int
↳ Raw View	{ObjectPtr=0x0000095c3e9750a0 (Name="MyOptionalObject") DebugPtr=0x0000095c3e9750a0 (Name="MyOptio...") ObjectPtr=0x0000095c3e9750a0 (Name="MyOptionalObject")}	TObjectPtr<UMyClass_Optional>
↳ MyNotOptionalObject	0x0000095c3e7ec5c0 (Name="MyNotOptionalObject") (Name="MyNotOptionalObject")	UMyClass_NonOptional *
↳ UObject	456	UObject
↳ MyProperty	456	int

在源码里搜索Optional，可以看到一般是EditorOnlyData和CookedMetaData类在使用。

```
UCLASS(Optional, within=Enum)
class ENGINE_API UEnumCookedMetaData : public UObject
UCLASS(Optional, within=ScriptStruct)
class ENGINE_API UStructCookedMetaData : public UObject
UCLASS(Optional, within=Class)
class ENGINE_API UClassCookedMetaData : public UObject

UMaterialInterfaceEditorOnlyData* UMaterialInterface::CreateEditorOnlyData()
{
    const UClass* EditorOnlyClass = GetEditorOnlyDataClass();
    check(EditorOnlyClass);
    check(EditorOnlyClass->HasAllClassFlags(CLASS_Optional));

    const FString EditorOnlyName =
    MaterialInterface::GetEditorOnlyDataName(*GetName());
    const EObjectFlags EditorOnlyFlags =
    GetMaskedFlags(RF_PropagateToSubObjects);
    return NewObject<UMaterialInterfaceEditorOnlyData>(this, EditorOnlyClass,
*EditorOnlyName, EditorOnlyFlags);
}
```

引擎里也有一些验证：

```
UnrealTypeDefinitionInfo.cpp:
// Validate if we are using editor only data in a class or struct definition
if (HasAnyClassFlags(CLASS_Optional))
{
    for (TSharedRef<FUnrealPropertyDefinitionInfo> PropertyDef : GetProperties())
    {
        if (PropertyDef->GetPropertyBase().IsEditorOnlyProperty())
        {
            PropertyDef->LogError(TEXT("Cannot specify editor only property
inside an optional class."));
        }
        else if (PropertyDef->GetPropertyBase().ContainsEditorOnlyProperties())
        {
            PropertyDef->LogError(TEXT("Do not specify struct property containing
editor only properties inside an optional class."));
        }
    }
}
```

通过源码发现：

```
//SAVE_Optional = 0x00008000,    //< Indicate that we want to save optional exports. This flag is only valid while cooking. Optional exports are filtered if not specified during cooking.
```

这个SAVE_Optional 作用于UUserDefinedEnum, UUserDefinedStruct, UBlueprintGeneratedClass的MetaData对象上。

```
void UUserDefinedStruct::PreSaveRoot(FObjectPreSaveRootContext ObjectSaveContext)
{
    Super::PreSaveRoot(ObjectSaveContext);

    if (ObjectSaveContext.IsCooking() && (ObjectSaveContext.GetSaveFlags() & SAVE_Optional))
    {
        //这个对象是以this为Outer的，标记RF_Standalone | RF_Public，会造成孩子对象被序列化下来
        UStructCookedMetaData* CookedMetaData = NewCookedMetaData();
        CookedMetaData->CacheMetaData(this);

        if (!CookedMetaData->HasMetaData())
        {
            PurgeCookedMetaData(); //清理掉这个CookedMetaData对象
        }
    }
    else
    {
        PurgeCookedMetaData();
    }
}
```

另外，在cook的时候，如果指定

```
bCookEditorOptional = Switches.Contains(TEXT("EDITOROPTIONAL")); // Produce the optional editor package data alongside the cooked data.
```

则会加上CookEditorOptional 的标识

```
CookFlags |= bCookEditorOptional ? ECookInitializationFlags::CookEditorOptional : ECookInitializationFlags::None;
```

再之后则会传达SAVE_Optional 给Package的SaveFlags

```
SaveFlags |= COTFS.IsCookFlagSet(ECookInitializationFlags::CookEditorOptional) ? SAVE_Optional : SAVE_None;
```

从而在包括Package的时候，IsSaveOptional()的判断会造成是否创建Optional的Realm，

```
TArray<ESaveRealm> FSaveContext::GetHarvestedRealmsToSave()
{
    TArray<ESaveRealm> HarvestedContextsToSave;
    if (IsCooking())
    {
        HarvestedContextsToSave.Add(ESaveRealm::Game);
        if (IsSaveOptional())
        {
            HarvestedContextsToSave.Add(ESaveRealm::Optional);
        }
    }
}
```

```

        }
    }
    else
    {
        HarvestedContextsToSave.Add(ESaveRealm::Editor);
    }
    return HarvestedContextsToSave;
}

```

还有如果发现Object有CLASS_Optional，则不把它当做Export(子对象)，而是当做Import(引用的对象)，Optional对象有可能放在外部独立的文件中。

```

ESavePackageResult HarvestPackage(FSaveContext& SaveContext)
{
    // If we have a valid optional context and we are saving it,
    // transform any harvested non optional export into imports
    // Mark other optional import package as well
    if (!SaveContext.IsSaveAutoOptional() &&
        SaveContext.IsSaveOptional() &&
        SaveContext.IsCooking() &&
        SaveContext.GetHarvestedRealm(ESaveRealm::Optional).GetExports().Num() &&
        SaveContext.GetHarvestedRealm(ESaveRealm::Game).GetExports().Num())
    {
        bool bHasNonOptionalSelfReference = false;
        FHarvestedRealm& OptionalContext =
        SaveContext.GetHarvestedRealm(ESaveRealm::Optional);
        for (auto It = OptionalContext.GetExports().CreateIterator(); It; ++It)
        {
            if (!It->Obj->GetClass()->HasAnyClassFlags(CLASS_Optional))
            {
                // Make sure the export is found in the game context as well
                if (FTaggedExport* GameExport =
                    SaveContext.GetHarvestedRealm(ESaveRealm::Game).GetExports().Find(It->Obj))
                {
                    // Flag the export in the game context to generate it's
                    public hash
                    GameExport->bGeneratePublicHash = true;
                    // Transform the export as an import
                    OptionalContext.AddImport(It->Obj);
                    // Flag the package itself to be an import
                    bHasNonOptionalSelfReference = true;
                }
                // if not found in the game context and the reference directly
                // came from an optional object, record an illegal reference
                else if (It->bFromOptionalReference)
                {
                    SaveContext.RecordIllegalReference(nullptr, It->Obj,
EILlegalRefReason::ReferenceFromOptionalToMissingGameExport);
                }
                It.RemoveCurrent();
            }
        }
        // Also add the current package itself as an import if we are referencing
        // any non optional export
        if (bHasNonOptionalSelfReference)

```

```

    {
        OptionalContext.AddImport(SaveContext.GetPackage());
    }
}

```

Transient

- 功能描述:** 指定该类的所有对象都略过序列化。
- 引擎模块:** Serialization
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中添加CLASS_Transient
- 关联项:** NonTransient
- 常用程度:** ★★★

指定该类的所有对象都略过序列化。

- 从不将属于此类的对象保存到磁盘。此说明符会传播到子类，但是可由NonTransient说明符覆盖。可以在子类被改写。
- 会造成相应Object的RF_Transient标记。
- 注意：UPROPERTY(Transient)只是指定这一个特定的属性不序列化。而UCLASS(Transient)是作用于该类的所有对象。

示例代码：

```

UCLASS(Blueprintable, Transient)
class INSIDER_API UMyClass_Transient :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};

UCLASS(Blueprintable, NonTransient)
class INSIDER_API UMyClass_NonTransient :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Transient_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UMyClass_Transient* MyTransientObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)

```

```

UMyClass_NonTransient* MyNonTransientObject;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyInt_Normal=123;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
int32 MyInt_Transient =123;
};

```

序列化对象指针关键是采用FObjectProperty里的SerializeItem方法Slot << ObjectValue;

不能采用Actor的Class Default来测试，因为：

Transient properties are serialized for (Blueprint) Class Default Objects but should not be serialized for any 'normal' instances of classes.

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Transient_Test :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UMyClass_Transient* MyTransientObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UMyClass_NonTransient* MyNonTransientObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Normal=123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
    int32 MyInt_Transient =123;
};

```

也不能用FObjectAndNameAsStringProxyArchive测试，因为FObjectAndNameAsStringProxyArchive 内部在序列化对象的时候，会先查找名字。

```

FArchive& FObjectAndNameAsStringProxyArchive::operator<<(uobject*& Obj)
{
    if (IsLoading())
    {
        // Load the path name to the object
        FString LoadedString;
        InnerArchive << LoadedString;
        // Look up the object by fully qualified pathname
        Obj = Findobject<uobject>(nullptr, *LoadedString, false);
        // If we couldn't find it, and we want to load it, do that
        if(!Obj && bLoadIfFindFails)
        {
            Obj = LoadObject<uobject>(nullptr, *LoadedString);
        }
    }
    else
    {
        // Save out the fully qualified object name
        FString SavedString(Obj->GetPathName());
        InnerArchive << SavedString;
    }
}

```

```
    return *this;  
}
```

因此采用Package的测试：

```
FString packageName = TEXT("/Game/MyTestPackage");  
FString assetPath = FPackageName::LongPackageNameToFilename(packageName,  
FPackageName::GetAssetPackageExtension());  
UPackage* package = CreatePackage(*packageName);  
FSavePackageArgs saveArgs{};  
saveArgs.Error = GError;  
  
//ObjectFlags: RF_NoFlags  
UMyClass_Transient_Test* testObject = NewObject<UMyClass_Transient_Test>(package,  
TEXT("testObject"));  
//ObjectFlags: RF_Transient  
testObject->MyTransientObject = NewObject<UMyClass_Transient>(testObject,  
TEXT("MyTransientObject"));  
//ObjectFlags: RF_NoFlags  
testObject->MyNonTransientObject = NewObject<UMyClass_NonTransient>(testObject,  
TEXT("MyNonTransientObject"));  
  
testObject->MyTransientObject->MyProperty = 456;  
testObject->MyNonTransientObject->MyProperty = 456;  
testObject->MyInt_Normal = 456;  
testObject->MyInt_Transient = 456;  
  
bool result = UPackage::SavePackage(package, testObject, *assetPath, saveArgs);
```

在保存完成之后，重新加载Package：

```
FString packageName = TEXT("/Game/MyTestPackage");  
FString assetPath = FPackageName::LongPackageNameToFilename(packageName,  
FPackageName::GetAssetPackageExtension());  
  
UPackage* package = LoadPackage(nullptr, *assetPath, LOAD_None);  
package->FullyLoad();  
  
UMyClass_Transient_Test* newTestObject=LoadObject<UMyClass_Transient_Test>  
(package, TEXT("testObject"), *assetPath);
```

示例效果：

可以看到MyTransientObject 并没有被序列化到磁盘上，因此不会加载出来。

▲ ⚒ newTestObject	0x0000075046120f00 (Name="testObject")
▶ ⚒ UObject	(Name="testObject")
▶ ⚒ MyTransientObject	0x0000000000000000 <NULL>
▲ ⚒ MyNonTransientObject	0x000007503f7ff7d0 (Name="MyNonTransientObject")
▶ ⚒ UObject	(Name="MyNonTransientObject")
▶ ⚒ MyProperty	456
▶ ⚒ MyInt_Normal	456
▶ ⚒ MyInt_Transient	123

原理：

在SavePackage的时候：RF_Transient会导致这个对象不会被HarvestExport，不会被放进SaveContext里

```
void FPackageHarvester::TryHarvestExport(UObject* Inobject)
{
    // Those should have been already validated
    check(Inobject && Inobject->IsInPackage(SaveContext.GetPackage()));

    // Get the realm in which we should harvest this export
    EIllegalRefReason Reason = EIllegalRefReason::None;
    ESaveRealm HarvestContext = GetObjectHarvestingRealm(Inobject, Reason);
    if (!SaveContext.GetHarvestedRealm(HarvestContext).IsExport(Inobject))
    {
        SaveContext.MarkUnsaveable(Inobject);
        bool bExcluded = false;
        if (!Inobject->HasAnyFlags(RF_Transient))
        {
            bExcluded = ConditionallyExcludeObjectForTarget(SaveContext,
Inobject, HarvestContext);
        }
        if (!Inobject->HasAnyFlags(RF_Transient) && !bExcluded)
        {
            // It passed filtering so mark as export
            HarvestExport(Inobject, HarvestContext);
        }
    }

    // If we have a illegal ref reason, record it
    if (Reason != EIllegalRefReason::None)
    {

        SaveContext.RecordIllegalReference(CurrentExportDependencies.CurrentExport,
Inobject, Reason);
    }
}
```

HideDropDown

- **功能描述：** 在类选择器中隐藏此类
- **引擎模块：** TypePicker
- **元数据类型：** bool
- **作用机制：** 在ClassFlags中添加CLASS_HideDropDown
- **常用程度：** ★★

在类选择器中隐藏此类，通常是TSubClassOf触发，或者Class变量触发的类选择窗口。这个时候，这个标识符可以阻止其出现。在源码里的使用，通常是一些旧的废弃的类或者Test类，Abstract类和基类。

示例代码：

```
UCLASS(Blueprintable)
```

```

class INSIDER_API UMyClass_HideDropDownBase :public UObject
{
    GENERATED_BODY()
public:
};

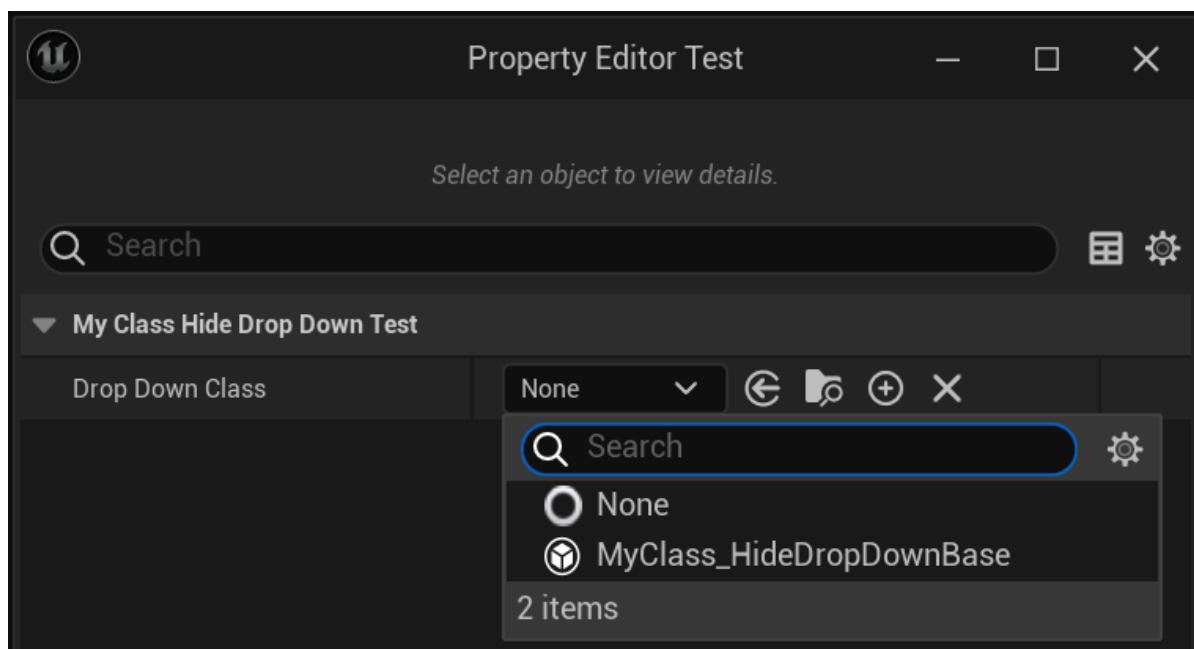
UCLASS(Blueprintable, hidedropdown)
class INSIDER_API UMyClass_HideDropDown :public UMyClass_HideDropDownBase
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable, hidedropdown)
class INSIDER_API UMyClass_NoHideDropDown :public UMyClass_HideDropDownBase
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable)
class INSIDER_API UMyClass_HideDropDown_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TSubclassOf<UMyClass_HideDropDownBase> DropDownClass;
};

```

示例结果：



原理：

HideDropDown会造成CLASS_HideDropDown标记，从而在类型UI定制化的列表里把该类剔除出去。

```
template <typename TClass>
bool FPropertyEditorClassFilter::IsClassAllowedHelper(TClass InClass)
{
    bool bMatchesFlags = !InClass->HasAnyClassFlags(CLASS_Hidden |
CLASS_HideDropDown | CLASS_Deprecated) &&
(bAllowAbstract || !InClass->HasAnyClassFlags(CLASS_Abstract));

    if (bMatchesFlags && InClass->IsChildOf(ClassPropertyMetaClass)
        && (!InterfaceThatMustBeImplemented || InClass-
>ImplementsInterface(InterfaceThatMustBeImplemented)))
    {
        auto PredicateFn = [InClass](const UClass* Class)
        {
            return InClass->IsChildOf(Class);
        };

        if (DisallowedClassFilters.FindByPredicate(PredicateFn) == nullptr &&
            (AllowedClassFilters.Num() == 0 ||
AllowedClassFilters.FindByPredicate(PredicateFn) != nullptr))
        {
            return true;
        }
    }

    return false;
}
```

CustomConstructor

- **功能描述：**阻止构造函数声明自动生成。
- **引擎模块：**UHT
- **元数据类型：**bool
- **作用机制：**在ClassFlags中添加CLASS_CustomConstructor

UHT不会生成NO_API UMyClass_ModuleAPI(const FObjectInitializer& ObjectInitializer = FObjectInitializer::Get());的默认构造函数。但是这个一般都是配合GENERATED_UCLASS_BODY使用的，因为GENERATED_BODY会自动生成默认构造函数。一般在自己需要自定义这个函数的时候使用。(但其实用GENERATED_BODY也行)

当前已经弃用：

```
CLASS_CustomConstructor UE_DEPRECATED(5.1, "CLASS_CustomConstructor should no
longer be used. It is no longer being set by engine code.") = 0x00008000u,
```

CustomFieldNotify

- **功能描述：**阻止UHT为该类生成FieldNotify的相关代码。

- **引擎模块:** UHT
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中增加HasCustomFieldNotify
- **常用程度:** 0

阻止UHT为该类生成FieldNotify的相关代码。

在源码里只在UWidget上使用，例如该类里面的bIsEnabled是FieldNotify，正常来说UHT要为其生成代码。但如果该类想自己手动书写这些UHT代码，则可以加上CustomFieldNotify来阻止UHT生成。UWidget的cpp里因为要用别的方式UE_FIELD_NOTIFICATION_IMPLEMENT_CLASS_DESCRIPTOR，因此要拒绝UHT生成。

如果自己的类也要自己UE_FIELD_NOTIFICATION_IMPLEMENT_CLASS_DESCRIPTOR，则可以用上CustomFieldNotify。

源码例子：

```
//E:\P4V\Engine\Source\Runtime\UMG\Public\FieldNotification\FieldNotificationDeclaration.h
UCLASS(Abstract, BlueprintType, Blueprintable, CustomFieldNotify)
class UMG_API UWidget : public UVisual, public INotifyFieldValueChanged
{
    GENERATED_UCLASS_BODY()
public:
    UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BASE_BEGIN(UMG_API)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(ToolTipText)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(Visibility)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(bIsEnabled)
        UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(ToolTipText)
        UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(Visibility)
        UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(bIsEnabled)
        UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END()
    UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BASE_END();

    UPROPERTY(EditAnywhere, BlueprintReadWrite, FieldNotify,
    Getter="GetIsEnabled", Setter="SetIsEnabled", BlueprintGetter="GetIsEnabled",
    BlueprintSetter="SetIsEnabled", Category="Behavior")
    uint8 bIsEnabled:1;

//cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_CLASS_DESCRIPTOR_ThreeFields(UWidget,
    ToolTipText, Visibility, bIsEnabled);
```

原理：

在判断条件上可见HasCustomFieldNotify的判断。

```

protected static bool NeedFieldNotifyCodeGen(UhtClass classObj)
{
    return

!classObj.ClassExportFlags.HasAnyFlags(UhtClassExportFlags.HasCustomFieldNotify)
&&

classObj.ClassExportFlags.HasAnyFlags(UhtClassExportFlags.HasFieldNotify);
}

```

CustomThunkTemplates

- 功能描述:** Specifies the struct that contains the CustomThunk implementations
- 引擎模块:** UHT
- 元数据类型:** bool

在源码里找不到引用的地方

Interface

- 功能描述:** 标识这个Class是个Interface。
- 引擎模块:** UHT
- 元数据类型:** bool
- 作用机制:** 在ClassFlags中添加CLASS_Interface
- 常用程度:** 0

标识这个Class是个Interface。

只用在NoExportTypes.h中，我们自己的UIinterface不需要手动设置。

是UHT在为UIinterface生成的时候，设置在.generated.h里的。

源码例子：

```

UCLASS(abstract, noexport, intrinsic, interface, config = Engine)
class UInterface : public UObject
{}

```

原理：

```

bool FKismetEditorUtilities::IsClassABlueprintInterface(const UClass* Class)
{
    if (Class->HasAnyClassFlags(CLASS_Interface) && !Class-
>HasAnyClassFlags(CLASS_NewerVersionExists))
    {
        return true;
    }
    return false;
}

```

Intrinsic

- **功能描述:** 指定UHT完全不为此类生成代码，需要自己手写。
- **引擎模块:** UHT
- **元数据类型:** bool
- **作用机制:** 在ClassFlags中增加CLASS_Intrinsic
- **常用程度:** 0

指定UHT完全不为此类生成代码，需要自己手写。

只在C++直接设定，一般新类不设定这个，标记这个的都是UE4内部原生的那些类，相当于已经在源码中手写了元数据代码。

noexport至少还会解析生成元数据，只是缺少注册。因此intrinsic类的所有元数据flags要自己手动标记。但是intrinsic完全不生成代码。其generated.h和.gen.cpp里面都是空的。noexporttyps.h里的目前采用intrinsic的类只有UCLASS(noexport, Intrinsic)class UModel{}，这还是被cpp不编译的。

```
//UCLASS(Intrinsic)
//class INSIDER_API UMyClass_Intrinsic :public UObject //syntax error: missing
';' before '<class-head>'
//{
//  GENERATED_BODY()
//
//};

//.h
class INSIDER_API UMyClass_Intrinsic :public UObject
{
    DECLARE_CLASS_INTRINSIC(UMyClass_Intrinsic, UObject,
CLASS_MatchedSerializers, TEXT("/Script/Insider"))
};

//.cpp
IMPLEMENT_INTRINSIC_CLASS(UMyClass_Intrinsic, INSIDER_API, UObject, INSIDER_API,
"/Script/Insider", {})

class COREUOBJECT_API UIInterface : public UObject
{
    DECLARE_CLASS_INTRINSIC(UIInterface, UObject, CLASS_Interface |
CLASS_Abstract, TEXT("/Script/CoreUObject"))
};
```

MinimalAPI

- **功能描述:** 不dll导出该类的函数，只导出类型信息当作变量。
- **引擎模块:** DllExport
- **元数据类型:** bool
- **作用机制:** 在ClassFlags增加CLASS_MinimalAPI
- **常用程度:** ★★★

不dll导出该类的函数，只导出类型信息当作变量。

- 其他引用的模块可以利用指针来做转换，但是不能调用上面的函数。但是蓝图里依然可以访问。
- 好处是可以缩短编译信息和加快链接速度，因为没有了那么多dllexport函数。
- 注意MinimalAPI不能和MODULENAME_API一起使用，因为MinimalAPI就是用来不导出的，而MODULENAME_API就是用来导出的。但是MinimalAPI的效果并不等价于不写MODULENAME_API的效果，因为MinimalAPI还会导出GetPrivateStaticClass用来允许NewObject。所以如果一个类完全不想让另一个模块知道，则不需要写任何导出。而如果想让另一个模块知道类型，但是完全不能调用函数，则可以用MinimalAPI来防止。
- 游戏的模块推荐不导出。插件的模块外部的推荐导出，内部的基类可以考虑MinimalAPI，私有类则可以完全不导出。引擎里使用MinimalAPI还是非常多的，生成的效果是这些类可以作为变量使用，但不能继承和调用方法。
- 一般是配合BlueprintType使用，这样就可以在蓝图中作为变量。
- 可以正常在蓝图中调用函数和属性。因为蓝图调用是只需要反射信息就可以的，因为是自己模块把函数和属性的指针注册到系统里。

示例代码：

```

UCLASS()
class UMyClass_NotMinimalAPI :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UFUNCTION(BlueprintCallable)
    void MyFunc();
};

UCLASS(MinimalAPI)
class UMyClass_MinimalAPI :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UFUNCTION(BlueprintCallable)
    void MyFunc();
};

UCLASS(MinimalAPI, BlueprintType)
class UMyClass_MinimalAPI_BlueprintType :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UFUNCTION(BlueprintCallable)
    void MyFunc() {}
};

UCLASS(MinimalAPI)
class UMyClass_MinimalAPI_BlueprintFunctionLibrary :public BlueprintFunctionLibrary

```

```

{
    GENERATED_BODY()
public:

    UFUNCTION(BlueprintCallable)
    static void MyFuncInMinimalAPI();

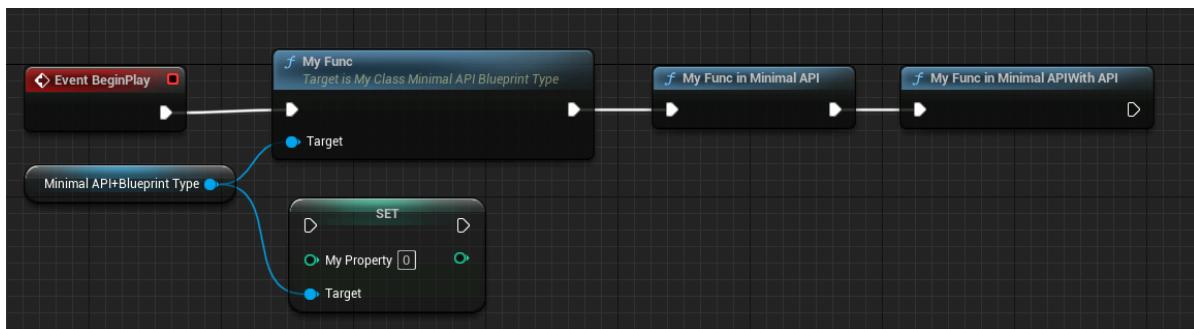
    UFUNCTION(BlueprintCallable)
    static INSIDER_API void MyFuncInMinimalAPIWithAPI();
};

}

```

示例效果：

可以正常在蓝图中调用函数和属性。蓝图函数库中的方法也可以调用，说明UHT对MinimalAPI还是依然生成反射的调用信息的，蓝图调用是只需要反射信息就可以的，因为是自己模块把函数和属性的指针注册到系统里，因此并不需要dll导出。只不过在dll导出工具里查看dll导出的函数列表并没有该函数。



查看dll导出函数列表：

```

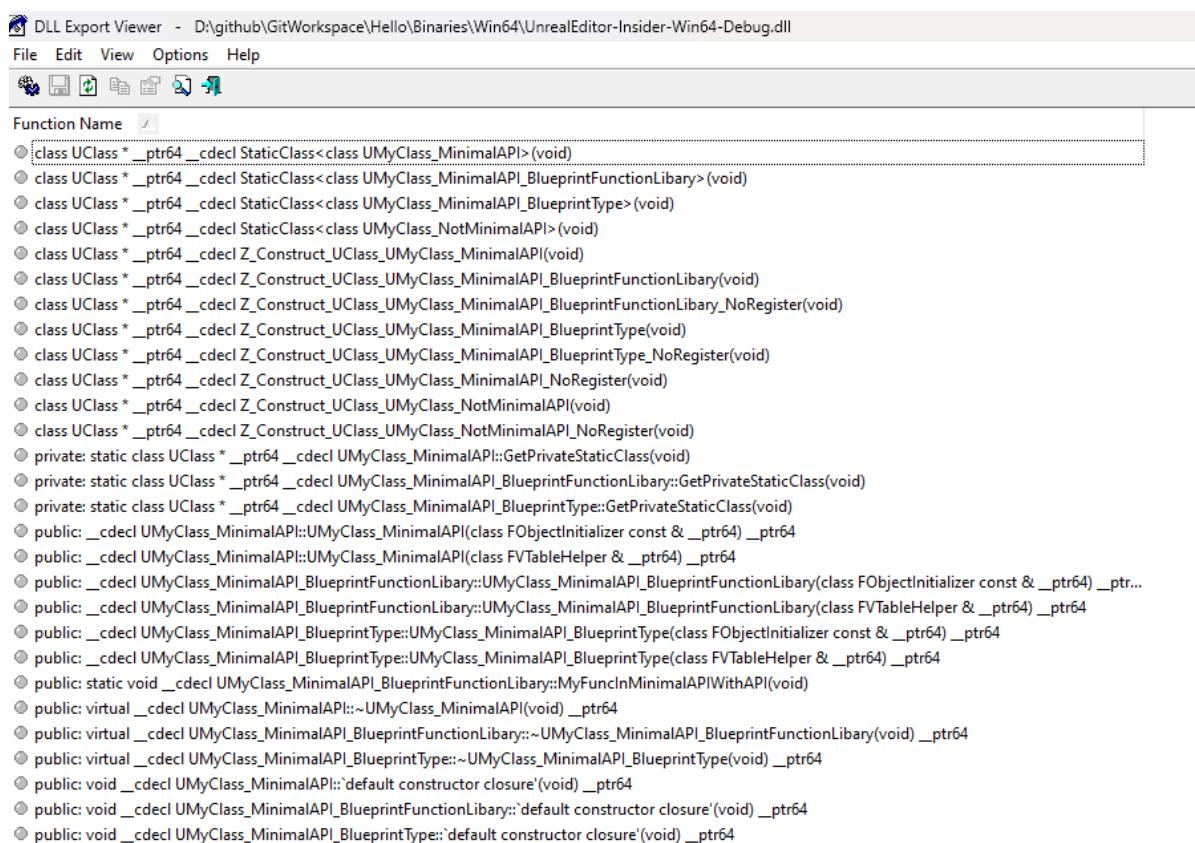
class UCClass * __ptr64 __cdecl StaticClass<class UMyClass_MinimalAPI>(void)
class UCClass * __ptr64 __cdecl StaticClass<class
UMyClass_MinimalAPI_BlueprintFunctionLibrary>(void)
class UCClass * __ptr64 __cdecl StaticClass<class
UMyClass_MinimalAPI_BlueprintType>(void)
class UCClass * __ptr64 __cdecl StaticClass<class UMyClass_NotMinimalAPI>(void)
class UCClass * __ptr64 __cdecl Z_Construct_UClass_UMyClass_MinimalAPI(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintFunctionLibrary(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintFunctionLibrary_NoRegister(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintType(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintType_NoRegister(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_NoRegister(void)
class UCClass * __ptr64 __cdecl Z_Construct_UClass_UMyClass_NotMinimalAPI(void)
class UCClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_NotMinimalAPI_NoRegister(void)
private: static class UCClass * __ptr64 __cdecl
UMyClass_MinimalAPI::GetPrivateStaticClass(void)
private: static class UCClass * __ptr64 __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::GetPrivateStaticClass(void)
private: static class UCClass * __ptr64 __cdecl
UMyClass_MinimalAPI_BlueprintType::GetPrivateStaticClass(void)

```

```

public: __cdecl UMyClass_MinimalAPI::UMyClass_MinimalAPI(class FObjectInitializer
const & __ptr64) __ptr64
public: __cdecl UMyClass_MinimalAPI::UMyClass_MinimalAPI(class FVTableHelper &
__ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::UMyClass_MinimalAPI_BlueprintFunctionLibrary(class FObjectInitializer const & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::UMyClass_MinimalAPI_BlueprintFunctionLibrary(class FVTableHelper & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintType::UMyClass_MinimalAPI_BlueprintType(class FObjectInitializer const & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintType::UMyClass_MinimalAPI_BlueprintType(class FVTableHelper & __ptr64) __ptr64
public: static void __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPIwithAPI(void)
public: virtual __cdecl UMyClass_MinimalAPI::~UMyClass_MinimalAPI(void) __ptr64
public: virtual __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::~UMyClass_MinimalAPI_BlueprintFunctionLibrary(void) __ptr64
public: virtual __cdecl
UMyClass_MinimalAPI_BlueprintType::~UMyClass_MinimalAPI_BlueprintType(void)
__ptr64
public: void __cdecl UMyClass_MinimalAPI::`default constructor closure'(void)
__ptr64
public: void __cdecl UMyClass_MinimalAPI_BlueprintFunctionLibrary::`default
constructor closure'(void) __ptr64
public: void __cdecl UMyClass_MinimalAPI_BlueprintType::`default constructor
closure'(void) __ptr64

```



在跨模块调用的时候，因为没有dll导出，因此会触发链接错误。

```
UMyClass_MinimalAPI* a = NewObject<UMyClass_MinimalAPI>();  
  
//第一种错误  
//error LNK2019: unresolved external symbol "public: void __cdecl  
UMyClass_MinimalAPI::MyFunc(void)" (?MyFunc@UMyClass_MinimalAPI@@QEAXXZ)  
referenced in function "public: void __cdecl  
UMyClass_UseMinimalAPI::TestFunc(void)" (?  
TestFunc@UMyClass_UseMinimalAPI@@QEAXXZ)  
//a->MyFunc();  
  
a->MyProperty++;  
  
//第二种错误  
//error LNK2019: unresolved external symbol "private: static class UClass *  
__cdecl UMyClass_NotMinimalAPI::GetPrivateStaticClass(void)" (?  
GetPrivateStaticClass@UMyClass_NotMinimalAPI@@CAPEAVUClass@@XZ)  
//referenced in function "class UClass * __cdecl NewObject<class  
UMyClass_NotMinimalAPI>(class UObject *)" (??  
$NewObject@VUMyClass_NotMinimalAPI@@@YAPEAVUMyClass_NotMinimalAPI@@PEAVUObject@@  
@Z)  
auto* a = NewObject<UMyClass_NotMinimalAPI>();  
  
//第三种错误  
//error LNK2019: unresolved external symbol "public: static void __cdecl  
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPI(void)" (?  
MyFuncInMinimalAPI@UMyClass_MinimalAPI_BlueprintFunctionLibrary@@SAXXZ)  
//referenced in function "public: void __cdecl  
UMyClass_UseMinimalAPI::TestFunc(void)" (?  
TestFunc@UMyClass_UseMinimalAPI@@QEAXXZ)  
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPI();  
  
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPIWithAPI();
```

NoExport

- 功能描述：**指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。
- 引擎模块：** UHT
- 元数据类型：** bool
- 作用机制：** 在ClassFlags中增加EClassFlags: CLASS_NoExport
- 常用程度：** 0

指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。

引擎的NoExportTypes.h里大量都是这种类型，专门提供给UHT来提取信息的。一般会用#ifndef !CPP
//noexport class来包裹，来避免编译。同时在另一个地方会定义这个类。因为
StaticRegisterNatives##TClass没有生成，所以GetPrivateStaticClass不能调用成功，所以不能
NewObject。一般noexport和Intrinsic都是配合使用的。因为DECLARE_CLASS_INTRINSIC内部会声明
static void StaticRegisterNatives##TClass() {} 来允许成功调用。

引擎里的结构倒是经常用noexport来阻止生成UHT注册。因为结构其实不需要调用GetPrivateStaticClass来创建元数据。只要有Z_Construct_UScriptStruct_XXX来生成构造相应的UScriptStruct对象就行。

测试代码：

```
UCLASS(noexport)
class INSIDER_API UMyClass_NoExport :public UObject
{
    GENERATED_BODY()
public:
};
```

测试结果：

编译的时候生成错误：

```
error LNK2019: unresolved external symbol "private: static void __cdecl
UMyClass_NoExport::StaticRegisterNativesUMyClass_NoExport(void)" (??
StaticRegisterNativesUMyClass_NoExport@UMyClass_NoExport@@CAXXZ) referenced in
function "private: static class UClass * __cdecl
UMyClass_NoExport::GetPrivateStaticClass(void)" (??
GetPrivateStaticClass@UMyClass_NoExport@@CAPEAVUClass@@XZ)
```

UCLASS()

- **功能描述：** 留空的默认行为是不能在蓝图中被继承，不能在蓝图中定义变量，但拥有反射的功能。
- **引擎模块：** UHT
- **元数据类型：** bool
- **作用机制：** 在ClassFlags中增加CLASS_MatchedSerializers, CLASS_Native, CLASS_RequiredAPI, CLASS_TokenStreamAssembled, CLASS_Intrinsic, CLASS_Constructed
- **关联项：** 不写UCLASS()
- **常用程度：** ★★★★☆

不能在蓝图中被继承，不能在蓝图中定义变量。

但依然都可以通过蓝图ConstructObject创建出来。对于想要拥有反射功能，但是并不想在蓝图中被使用会挺适合。

示例代码：

```
/*
[MyClass_Default    Class->Struct->Field->Object
/Script/Insider.MyClass_Default] [IncludePath = Class/MyClass_Default.h,
ModuleRelativePath = Class/MyClass_Default.h]
ObjectFlags:    RF_Public | RF_Standalone | RF_Transient
Outer:    Package /Script/Insider
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size:    48
{
```

```

public: void ExecuteUbergraph(int32 EntryPoint);
};

*/



UCLASS()
class INSIDER_API UMyClass_Default :public UObject
{
    GENERATED_BODY()
public:
};

```

默认的拥有这些标志: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed

不写UCLASS()

- 功能描述:** 只是作为一个普通的C++对象，没有反射功能。
- 引擎模块:** UHT
- 元数据类型:** bool
- 关联项:** UCLASS()
- 常用程度:** ★

只是作为一个普通的C++对象，没有反射功能。

一般情况继承自UObject的最少也会有一个UCLASS()，这样才有反射功能。但是注意，如果调用 UMyClass_NoUCLASS::StaticClass()会返回基类UObject的Class，因为子类没有覆盖。因此也可以说本类是没有生成自己的UClass元数据对象。

```

class INSIDER_API UMyClass_NoUCLASS :public UObject
{
};


```

UObject的Class默认的标记是: CLASS_Abstract | CLASS_MatchedSerializers | CLASS_Native | CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed。因此不能被NewObject生成对象。在手动去掉CLASS_Abstract后可以正常new，但是对象的名称依然是Object，显然这是因为使用的就是Object的Class。

Blueprintable

- 功能描述:** 可以在蓝图中实现
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 作用机制:** 在Meta中加入IsBlueprintBase, BlueprintType
- 关联项:** NotBlueprintable
- 常用程度:** ★★★★☆

是否可以在蓝图中实现。

示例代码：

```
UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_Blueprintable:public UInterface
{
    GENERATED_UINTERFACE_BODY()
};

class INSIDER_API IMyInterface_Blueprintable
{
    GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    void Func_ImplementableEvent() const;

    UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    void Func_NativeEvent() const;
};

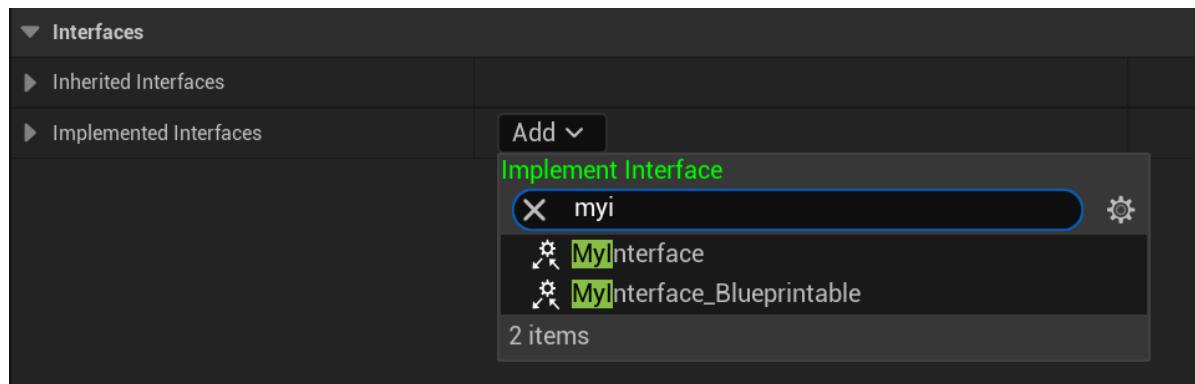
UINTERFACE(NotBlueprintable,MinimalAPI)
class UMyInterface_NotBlueprintable:public UInterface
{
    GENERATED_UINTERFACE_BODY()
};

class INSIDER_API IMyInterface_NotBlueprintable
{
    GENERATED_IINTERFACE_BODY()
public:
    //也不得定义蓝图函数，因为已经不能在蓝图中实现了
    //UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    //void Func_ImplementableEvent() const;

    // UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    // void Func_NativeEvent() const;
};
```

示例效果：

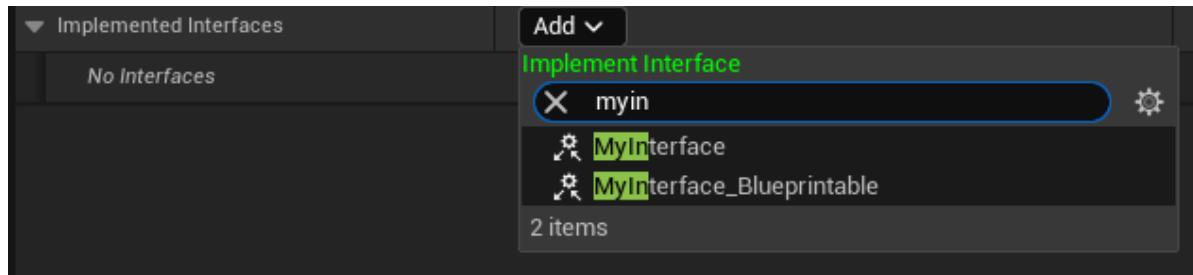
在蓝图中测试，发现UMyInterface_NotBlueprintable并不能找到。



NotBlueprintable

- **功能描述:** 指定不可以再蓝图中实现
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在Meta中去除IsBlueprintBase、 BlueprintType，等价于 CannotImplementInterfaceInBlueprint
- **关联项:** Blueprintable
- **常用程度:** ★★★

在Class Settings里的Interface里找不到不允许实现的接口。



什么情况下需要用到该标记？虽然不能在蓝图中实现，但是依然可以在C++里实现，也可以通过反射判断一个对象是否实现该接口。

ConversionRoot

- **功能描述:** Sets IsConversionRoot metadata flag for this interface.
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在Meta中加入IsConversionRoot

在源码中并不能找到该使用示例

MinimalAPI

- **功能描述:** 指定该UIInterface对象不导出到别的模块
- **元数据类型:** bool
- **引擎模块:** DllExport
- **常用程度:** ★

可以参照UCLASS里的MinimalAPI的解释。

简单来说UIInterface对象，只是作为接口的辅助对象，因此本身并没有什么可值得暴露出来的函数。因此源码里的大部分UIInterface对象都被标记成了MinimalAPI，以加快编译同时隔绝别的模块使用。

```
UINTERFACE(MinimalAPI, BlueprintType)
class USoundLibraryProviderInterface : public UIInterface
{
    GENERATED_BODY()
};
```

BlueprintInternalUseOnly

- **功能描述:** 不可定义新BP变量，但可作为别的类的成员变量暴露和变量传递
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在Meta中加入BlueprintInternalUseOnly, BlueprintType
- **常用程度:** ★★

指明这个STRUCT会是个BlueprintType，但在蓝图编辑器中又不能声明新变量，但是可以作为别的类的成员变量暴露到蓝图中。

和不写BlueprintType的差别是什么？

不写BlueprintType则完全不能作为别的类的成员变量。BlueprintInternalUseOnly抑制了定义新变量的能力，但是可以作为变量传递。比如在C++中定义变量，然后在蓝图中传递。

如FTableRowBase本身并不能定义新变量，但是其子类（要加上BlueprintType）是可以定义新变量的，正常被使用。

示例代码：

```
//(BlueprintInternalUseOnly = true, BlueprintType = true, ModuleRelativePath =
Struct/MyStruct_BlueprintInternalUseOnly.h)
USTRUCT(BlueprintInternalUseOnly)
struct INSIDER_API FMyStruct_BlueprintInternalUseOnly
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score=0.f;
};

USTRUCT()
struct INSIDER_API FMyStruct_NoBlueprintInternalUseOnly
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float Score=0.f;
};

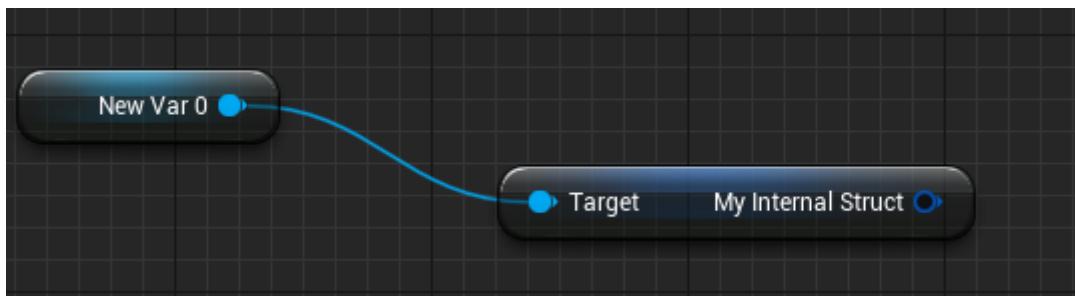
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_BlueprintInternalUseOnlyTest :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStruct_BlueprintInternalUseOnly MyInternalStruct;

    /*UPROPERTY(BlueprintReadWrite, EditAnywhere)      //no supported by BP
    FMyStruct_NoBlueprintInternalUseOnly MyStruct;*/}

};
```

示例效果：

NewVar是UMyClass_BlueprintInternalUseOnlyTest 类型的，依然可以访问内部的MyInternalStruct变量。



源码里可以找到：

```
USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{};

USTRUCT(BlueprintInternalUseOnly)
struct FTableRowBase
{}
```

原理：

```
bool UEdGraphSchema_K2::IsAllowableBlueprintVariableType(const UScriptStruct* InStruct, const bool bForInternalUse)
{
    if (const UUserDefinedStruct* UDStruct = Cast<const UUserDefinedStruct>(InStruct))
    {
        if (EUserDefinedStructureStatus::UDSS_UpToDate != UDStruct->Status.GetValue())
        {
            return false;
        }

        // User-defined structs are always allowed as BP variable types.
        return true;
    }

    // struct needs to be marked as BP type
    if (InStruct && InStruct->GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_AllowableBlueprintVariableType))
    {
        // for internal use, all BP types are allowed
        if (bForInternalUse)
        {
            return true;
        }

        // for user-facing use case, only allow structs that don't have the
        internal-use-only tag
```

```

// struct itself should not be tagged
if (!InStruct-
>GetBoolMetaData(FBlueprintMetadata::MD_BlueprintInternalUseOnly))
{
    // struct's base structs should not be tagged
    if (!InStruct-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_BlueprintInternalUseOnlyHierarchical))
    {
        return true;
    }
}

return false;
}

//Node->IsIntermediateNode()如果为true，则是作为中间节点使用，true会导致bForInternalUse
为true
if (!UK2Node_MakeStruct::CanBeMade(Node->StructType, Node->IsIntermediateNode()))

```

BlueprintInternalUseOnlyHierarchical

- 功能描述：** 在BlueprintInternalUseOnly的基础上，增加了子类也不能定义新BP变量的限制。
- 元数据类型：** bool
- 引擎模块：** Blueprint
- 作用机制：** 在Meta中加入BlueprintInternalUseOnlyHierarchical
- 常用程度：** ★

在BlueprintInternalUseOnly的基础上，增加了子类也不能定义新BP变量的限制。

目前只找到一个用处，但是也依然没有子类。如果我们在C++中定义新的子类，则所有的子类都不能定义变量。注意和FTableRowBase的区别是，FTableRowBase的子类依然可以定义新变量，因为FTableRowBase的BlueprintInternalUseOnly标记只作用于自己。

示例代码：

```

USTRUCT(BlueprintInternalUseOnlyHierarchical)
struct GAMEPLAYABILITIESEDITOR_API FGameplayAbilityAuditRow : public
FTableRowBase
{};

USTRUCT(BlueprintInternalUseOnly)
struct FTableRowBase
{};

```

原理：

只在这个地方用到，GetBoolMetaDataHierarchical会检查结构的所有父类测试是否含有某个标记。所以只要有一个父类有一个这个标记，就不能定义新变量。

```

bool UEdGraphSchema_K2::IsAllowableBlueprintVariableType(const UScriptStruct* InStruct, const bool bForInternalUse)
{
    if (const UUserDefinedStruct* UDStruct = Cast<const UUserDefinedStruct>(InStruct))
    {
        if (EUserDefinedStructureStatus::UDSS_UpToDate != UDStruct->Status.GetValue())
        {
            return false;
        }

        // User-defined structs are always allowed as BP variable types.
        return true;
    }

    // struct needs to be marked as BP type
    if (InStruct && InStruct->GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_AllowableBlueprintVariableType))
    {
        // for internal use, all BP types are allowed
        if (bForInternalUse)
        {
            return true;
        }

        // for user-facing use case, only allow structs that don't have the
        // internal-use-only tag
        // struct itself should not be tagged
        if (!InStruct->GetBoolMetaData(FBlueprintMetadata::MD_BlueprintInternalUseOnly))
        {
            // struct's base structs should not be tagged
            if (!InStruct->GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_BlueprintInternalUseOnlyHierarchical))
            {
                return true;
            }
        }
    }

    return false;
}

```

BlueprintType

- **功能描述:** 允许这个结构在蓝图中声明变量
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在Meta中加入BlueprintType

- 常用程度：★★★★★

和UCLASS里的一样，可以允许这个结构在蓝图中声明变量

示例代码：

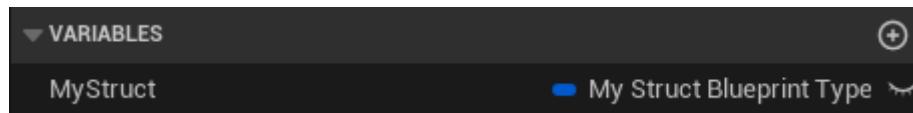
```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_BlueprintType
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

USTRUCT()
struct INSIDER_API FMyStruct_NoBlueprintType
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float Score;
};
```

测试蓝图：



immutable

- **功能描述：** Immutable is only legal in Object.h and is being phased out, do not use on new structs!
- **元数据类型：** bool
- **引擎模块：** Serialization
- **作用机制：** 在StructFlags中加入STRUCT_Immutable

当前只在noexporttypes.h里找到一堆Struct

指定这个结构的字段已经定义完毕，以后不会再修改，因此可以UseBinarySerialization来序列化，不需要支持字段的增删。

```
//USTRUCT(BlueprintType, Immutable) //error : Immutable is being phased out in
//favor of SerializeNative, and is only legal on the mirror structs declared in
//UObject
//struct INSIDER_API FMyStruct_Immutable
//{
//    GENERATED_BODY()
//
//    UPROPERTY(BlueprintReadWrite, EditAnywhere)
//    float Score;
//}
```

```
//};

Struct[67] WithFlags:STRUCT_Immutable
Struct: ScriptStruct /Script/CoreUObject.Guid
Struct: ScriptStruct /Script/CoreUObject.DateTime
Struct: ScriptStruct /Script/CoreUObject.Box
Struct: ScriptStruct /Script/CoreUObject.Vector
Struct: ScriptStruct /Script/CoreUObject.Box2D
Struct: ScriptStruct /Script/CoreUObject.Vector2D
Struct: ScriptStruct /Script/CoreUObject.Box2F
Struct: ScriptStruct /Script/CoreUObject.Vector2F
Struct: ScriptStruct /Script/CoreUObject.Box3D
Struct: ScriptStruct /Script/CoreUObject.Vector3D
Struct: ScriptStruct /Script/CoreUObject.Box3F
Struct: ScriptStruct /Script/CoreUObject.Vector3F
Struct: ScriptStruct /Script/CoreUObject.Color
Struct: ScriptStruct /Script/CoreUObject.Int32Point
Struct: ScriptStruct /Script/CoreUObject.Int32Vector
Struct: ScriptStruct /Script/CoreUObject.Int32Vector2
Struct: ScriptStruct /Script/CoreUObject.Int32Vector4
Struct: ScriptStruct /Script/CoreUObject.Int64Point
Struct: ScriptStruct /Script/CoreUObject.Int64Vector
Struct: ScriptStruct /Script/CoreUObject.Int64Vector2
Struct: ScriptStruct /Script/CoreUObject.Int64Vector4
Struct: ScriptStruct /Script/CoreUObject.LinearColor
Struct: ScriptStruct /Script/CoreUObject.Quat
Struct: ScriptStruct /Script/CoreUObject.TwoVectors
Struct: ScriptStruct /Script/CoreUObject.IntPoint
Struct: ScriptStruct /Script/CoreUObject.IntVector
Struct: ScriptStruct /Script/CoreUObject.IntVector2
Struct: ScriptStruct /Script/CoreUObject.IntVector4
Struct: ScriptStruct /Script/CoreUObject.Matrix
Struct: ScriptStruct /Script/CoreUObject.Plane
Struct: ScriptStruct /Script/CoreUObject.Matrix4D
Struct: ScriptStruct /Script/CoreUObject.Plane4D
Struct: ScriptStruct /Script/CoreUObject.Matrix44F
Struct: ScriptStruct /Script/CoreUObject.Plane4F
Struct: ScriptStruct /Script/CoreUObject.OrientedBox
Struct: ScriptStruct /Script/CoreUObject.PackedNormal
Struct: ScriptStruct /Script/CoreUObject.PackedRGB10A2N
Struct: ScriptStruct /Script/CoreUObject.PackedRGBA16N
Struct: ScriptStruct /Script/CoreUObject.Quat4D
Struct: ScriptStruct /Script/CoreUObject.Quat4F
Struct: ScriptStruct /Script/CoreUObject.Ray
Struct: ScriptStruct /Script/CoreUObject.Ray3D
Struct: ScriptStruct /Script/CoreUObject.Ray3F
Struct: ScriptStruct /Script/CoreUObject.Rotator
Struct: ScriptStruct /Script/CoreUObject.Rotator3D
Struct: ScriptStruct /Script/CoreUObject.Rotator3F
Struct: ScriptStruct /Script/CoreUObject.Sphere
Struct: ScriptStruct /Script/CoreUObject.Sphere3D
Struct: ScriptStruct /Script/CoreUObject.Sphere3F
Struct: ScriptStruct /Script/CoreUObject.Timespan
Struct: ScriptStruct /Script/CoreUObject.Transform3D
Struct: ScriptStruct /Script/CoreUObject.Transform3F
Struct: ScriptStruct /Script/CoreUObject.Uint32Point
```

```
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector2
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector4
Struct: ScriptStruct /Script/CoreUObject.Uint64Point
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector2
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector4
Struct: ScriptStruct /Script/CoreUObject.UintPoint
Struct: ScriptStruct /Script/CoreUObject.UintVector
Struct: ScriptStruct /Script/CoreUObject.UintVector2
Struct: ScriptStruct /Script/CoreUObject.UintVector4
Struct: ScriptStruct /Script/CoreUObject.Vector4
Struct: ScriptStruct /Script/CoreUObject.Vector4d
Struct: ScriptStruct /Script/CoreUObject.Vector4f
```

Atomic

- **功能描述:** 指定该结构在序列化的时候总是一整个输出全部属性，而不是只输出改变的属性。
- **元数据类型:** bool
- **引擎模块:** UHT
- **作用机制:** 在StructFlags中加入STRUCT_Atomic
- **常用程度:** ★

指定该结构在序列化的时候总是一整个输出全部属性，而不是只输出改变的属性。

所谓的原子化序列化指的是如果该结构的某个字段属性同默认值不同，但是其他字段相同，也要一次性地序列化整个结构，而不是拆开。注意这个只在普通的SerializeVersionedTaggedProperties下有效，因为是对比默认值。在Bin下无效。其实作用机理是当采用原子化序列化的时候，就不检查内部属性的默认值，从而无论什么情况都会序列化进整个属性。

UE的noexporttype.h中有大量的atomic的基础结构，如FVector，因为Immutable也会同时设置STRUCT_Atomic，但是没有发现单独设置Atomic的地方。

示例代码：

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_InnerItem
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 A = 1;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 B = 2;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 C = 3;

    bool operator==(const FMyStruct_InnerItem& other) const
    {
        return A == other.A;
    }
}
```

```

};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_NoAtomic
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStruct_InnerItem Item;
};

USTRUCT(Atomic, BlueprintType)
struct INSIDER_API FMyStruct_Atomic
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStruct_InnerItem Item;
};

template<>
struct TStructOpsTypeTraits<FMyStruct_InnerItem> : public
TStructOpsTypeTraitsBase2<FMyStruct_InnerItem>
{
    enum
    {
        withIdenticalViaEquality = true,
    };
};

void USerializationLibrary::SaveStructToMemory(UScriptStruct* structClass, void*
structObject, const void* structDefaults, TArray<uint8>& outSaveData,
EInsiderSerializationFlags flags/*=EInsiderSerializationFlags::None*/)
{
    FMemoryWriter MemoryWriter(outSaveData, false);
    MemoryWriter.SetWantBinaryPropertySerialization(EnumHasAnyFlags(flags,
EInsiderSerializationFlags::UseBinary));
    if (!EnumHasAnyFlags(flags, EInsiderSerializationFlags::CheckDefaults))
    {
        structDefaults=nullptr;
    }
    structClass->SerializeItem(MemoryWriter, structObject, structDefaults);
}

```

测试代码:

```

FMyStruct_NoAtomic NoAtomicStruct;
NoAtomicStruct.Item.A=3;

FMyStruct_Atomic AtomicStruct;
AtomicStruct.Item.A=3;

TArray<uint8> NoAtomicMemoryChanged;
USerializationLibrary::SaveStructToMemory(NoAtomicStruct, NoAtomicMemoryChanged, EInsiderSerializationFlags::CheckDefaults);

TArray<uint8> AtomicMemoryChanged;

```

```
userSerializationLibrary::SaveStructToMemory(AtomicStruct, AtomicMemoryChanged, EInside  
erSerializationFlags::CheckDefaults);
```

示例效果：

可见AtomicMemoryChanged的占用内存大小比AtomicMemoryChanged多，因为这两个结构的属性虽然都改变了，但是AtomicStruct总是会把所有的属性都序列化出来。

原理：

作用的机理是，一个外部结构是Atomic的，其内部的属性如果发现有改变，这个时候内部属性得是另一个结构，因为如果只是Int属性，则不会对比内部属性默认值。如果是内部结构属性的话，因为其中一个ID字段不一样，就在对比的时候导致整个结构不等（但同时该内部结构又有其他属性是相同的，所以上面示例代码只改了A，且提供了==的比较函数）。默认的方式是依然会在内部结构的内部属性上继续对比默认值，但原子化后就截断了默认值为null，从而导致孙子属性没有默认值可对比，从而就把整个内部属性就都输出出来。因此Atomic是用在外部结构上的，用在FVector这种不太会继续拆开的结构其实没什么意义。

```
void USTRUCT::SERIALIZEVERSIONEDTAGGEDPROPERTIES(FSTRUCTUREDArchive::FSLOT Slot,
uint8* Data, USTRUCT* DefaultsStruct, uint8* Defaults, const UObject* BreakRecursionIfFullyLoad) const
{
//.....
/** If true, it means that we want to serialize all properties of this struct if
any properties differ from defaults */
    bool bUseAtomicSerialization = false;
    if (DefaultsscriptStruct)
    {
        bUseAtomicSerialization = DefaultsscriptStruct-
>ShouldSerializeAtomically(UnderlyingArchive);
    }

    if (bUseAtomicSerialization)
    {
        DefaultValue = NULL;
    }
}
```

HasDefaults

- **功能描述:** 指定该结构的字段拥有默认值。这样如果本结构作为函数参数或返回值时候，函数则可以为其提供默认值。
 - **元数据类型:** bool
 - **引擎模块:** UHT
 - **限制类型:** 只在NoExportTypes.h供UHT使用
 - **作用机制:** 在FunctionFlags中加入FUNC_HasDefaults

- **常用程度:** 0

指定该结构的字段拥有默认值。

不是指的是NoExportTypes.h的声明上是否写有默认值，而是指其真正的声明之处，其内部的属性都有初始值。这样如果本结构作为函数参数或返回值时候，函数则可以为其提供默认值。

NoExportTypes.h里的大部分结构都拥有该结构（88/135），没有的是像FPackedXXX的。

原理：

如果是一个class中的函数且参数用到了结构，如果该结构拥有HasDefaults，则会造成EFunctionFlags.HasDefaults

```
// The following code is only performed on functions in a class.
if (Outer is UhtClass)
{
    foreach (UhtType type in Children)
    {
        if (type is UhtProperty property)
        {
            if (property.PropertyFlags.HasExactFlags(EPropertyFlags.OutParm | EPropertyFlags.ReturnParm, EPropertyFlags.OutParm))
            {
                FunctionFlags |= EFunctionFlags.HasOutParms;
            }
            if (property is UhtStructProperty structProperty)
            {
                if (structProperty.ScriptStruct.HasDefaults)
                {
                    FunctionFlags |= EFunctionFlags.HasDefaults;
                }
            }
        }
    }
}
```

HasNoOpConstructor

- **功能描述:** 指定该结构拥有ForceInit的构造函数，这样在作为BP function返回值的时候，可以调用来初始化
- **元数据类型:** bool
- **引擎模块:** UHT
- **限制类型:** 只在NoExportTypes.h供UHT使用
- **常用程度:** 0

指定该结构拥有ForceInit的构造函数，这样在作为BP Function返回值或参数的时候，引擎就知道这个结构有这么一个构造函数可以调用来初始化。

作用的地方是UhtHeaderCodeGenerator中的AppendEventParameter，为了这样的代码，这是一个暴露到BP中的Event，要为它生成一些胶水代码。这里FLinearColor就是HasNoOpConstructor。例如以下这个函数：

```

UFUNCTION(BlueprintNativeEvent, Category = "Modifier")
FLinearColor GetVisualizationColor(FInputActionValue Samplevalue,
FInputActionValue Finalvalue) const;

```

生成的代码：

```

struct InputModifier_eventGetVisualizationColor_Parms
{
    FInputActionValue Samplevalue;
    FInputActionValue Finalvalue;
    FLinearColor ReturnValue;

    /** Constructor, initializes return property only */
    InputModifier_eventGetVisualizationColor_Parms()
        : ReturnValue(ForceInit)//强制初始化
    {
    }
};

static FName NAME_UInputModifier_GetVisualizationColor =
FName(TEXT("GetVisualizationColor"));
FLinearColor UInputModifier::GetVisualizationColor(FInputActionValue
SampleValue, FInputActionValue FinalValue) const
{
    InputModifier_eventGetVisualizationColor_ParmsParms;
   Parms.Samplevalue=Samplevalue;
   Parms.Finalvalue=Finalvalue;
    const_cast<UInputModifier*>(this)->ProcessEvent(FindFunctionChecked(NAME_UInputModifier_GetVisualizationColor),&Parms);
    returnParms.Returnvalue;
}

```

因此要求该结构拥有ForceInit的构造函数

```

FORCEINLINE explicit FLinearColor(EForceInit)
: R(0), G(0), B(0), A(0)
{}

```

原理：

```

if (ScriptStruct.HasNoOpConstructor)
{
    //If true, the an argument will need to be added to the constructor
    PropertyCaps |= UhtPropertyCaps.RequiresNullConstructorArg;
}

```

IsAlwaysAccessible

- 功能描述：**指定UHT在生成文件的时候总是可以访问到改结构的声明，否则要在gen.cpp里生成镜像结构定义

- **元数据类型:** bool
- **引擎模块:** UHT
- **限制类型:** 只在NoExportTypes.h供UHT使用
- **常用程度:** 0

指定该结构的声明是否在UHT为NoExportTypes.h生成的gen.cpp里总是可以访问到。

换句话说其实就是是否这些结构在GeneratedCppIncludes.h的声明里可以找到。如果不可以找到，那在后面生成Z_Construct_UScriptStruct_FMatrix44d_Statics这种类似的时候就得再自己定义一个镜像结构定义。如果可以找到，比如FGuid，则就不需要。

因此这只是一个手动的内部标记，帮助UHT程序识别哪些结构要再创建镜像结构定义。

在NoExportTypes.h查看各个结构的时候，会发现有些结构（85/135）会标上IsAlwaysAccessible，而有些没有。这是因为UHT在为NoExportTypes.h生成gen.cpp的时候，

```
\unrealEngine\Engine\Source\Runtime\coreUObject\Public\UObject\GeneratedCppIncludes.h
#include "UObject/Object.h"
#include "UObject/UObjectGlobals.h"
#include "UObject/CoreNative.h"
#include "UObject/Class.h"
#include "UObject/MetaData.h"
#include "UObject/UnrealType.h"
#include "UObject/EnumProperty.h"
#include "UObject/TextProperty.h"
#include "UObject/FieldPathProperty.h"

#if UE_ENABLE_INCLUDE_ORDER_DEPRECATED_IN_5_2
#include "CoreMinimal.h"
#endif

\Hello\Intermediate\Build\win64\HelloEditor\Inc\CoreUObject\UHT\NoExportTypes.gen
.cpp:
// Copyright Epic Games, Inc. All Rights Reserved.
/*=====
   Generated code exported from UnrealHeaderTool.
   DO NOT modify this manually! Edit the corresponding .h files instead!
=====*/
//以下这两行
#include "UObject/GeneratedCppIncludes.h"//A
#include "UObject/NoExportTypes.h"//B
PRAGMA_DISABLE_DEPRECATED_WARNINGS
void EmptyLinkFunctionForGeneratedCodeNoExportTypes() {}
```

在最开头的两个include里如果可以直接找到该struct的定义，则在gen.cpp中的A和B处需要结构定义的时候，就不需要再额外去找定义了。

```

const UECodeGen_Private::FStructParams
Z_Construct_UScriptStruct_FMatrix44f_Statics::ReturnStructParams = {
    (UObject* (*)())Z_Construct_UPackage__Script_CoreUObject,
    nullptr,
    nullptr,
    "Matrix44f",
    Z_Construct_UScriptStruct_FMatrix44f_Statics::PropPointers,
    UE_ARRAY_COUNT(Z_Construct_UScriptStruct_FMatrix44f_Statics::PropPointers),
    sizeof(FMatrix44f), //这个A
    alignof(FMatrix44f), //这个B
    RF_Public|RF_Transient|RF_MarkAsNative,
    EStructFlags(0x00000038),
};

METADATA_PARAMS(UE_ARRAY_COUNT(Z_Construct_UScriptStruct_FMatrix44f_Statics::Struct_MetaDataParams),
Z_Construct_UScriptStruct_FMatrix44f_Statics::Struct_MetaDataParams)
};

```

如果找不到，比如FMatrix44f，是定义在Engine\Source\Runtime\Core\Public\Math\Matrix.h，则必须为它生成一个一模一样的定义（不include的作用是加快编译）：

```

struct Z_Construct_UScriptStruct_FMatrix44f_Statics
{
    struct FMatrix44f //内存布局一致的定义
    {
        FPlane4f XPlane;
        FPlane4f YPlane;
        FPlane4f ZPlane;
        FPlane4f WPlane;
    };

    static_assert(sizeof(FMatrix44f) < MAX_uint16);
    static_assert(alignof(FMatrix44f) < MAX_uint8);
};

```

当然如果子字段或者父类也找不到定义，则只需要把父定义先写在前面就可以了。因此cs里的FindNoExportStructsRecursive就是为了找到其相关的结构。没有标IsAlwaysAccessible则意味着要生成假的结构定义

```

private static void FindNoExportStructsRecursive(List<UhtScriptStruct>
outScriptStructs, UhtStruct structObj)
{
    for (UhtStruct? current = structObj; current != null; current =
current.SuperStruct)
    {
        // Is isn't true for noexport structs
        if (current is UhtScriptStruct scriptStruct)
        {
            if
(scriptStruct.ScriptStructFlags.HasAnyFlags(EstructFlags.Native))
            {
                break;
            }
        }
    }
}

```

```

        // these are a special cases that already exists and if wrong
if exported naively
{
    if (!scriptStruct.IsAlwaysAccessible)
    {
        outScriptStructs.Remove(scriptStruct);
        outScriptStructs.Add(scriptStruct);
    }
}

foreach (UhtType type in current.Children)
{
    if (type is UhtProperty property)
    {
        foreach (UhtType referenceType in
property.EnumerateReferencedTypes())
        {
            if (referenceType is UhtScriptStruct
propertyScriptStruct)
            {
                FindNoExportStructsRecursive(outScriptStructs,
propertyScriptStruct);
            }
        }
    }
}
}

```

IsCoreType

- 功能描述:** 指定该结构是核心类，UHT在用它的时候不需要前向声明。
- 元数据类型:** bool
- 引擎模块:** UHT
- 限制类型:** 只在NoExportTypes.h供UHT使用
- 常用程度:** 0

指定该结构是核心类，UHT在用它的时候不需要前向声明。

原理：

看UHT源码是把struct用在参数或属性等被引用的时候。

```

public override string? UhtStructProperty::GetForwardDeclarations()
{
    if (ScriptStruct.IsCoreType)
    {
        return null;
    }

    if (TemplateWrapper != null)
    {
        StringBuilder builder = new();
        TemplateWrapper.AppendForwardDeclarations(builder);
    }
}

```

```

        return builder.ToString();
    }

    return $"struct {ScriptStruct.SourceName};";
}

```

NoExport

- **功能描述:** 指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。
- **元数据类型:** bool
- **引擎模块:** UHT
- **常用程度:** ★

指定UHT不要用来自动生成注册的代码，而只是进行词法分析提取元数据。

NoExportTypes.h里使用了很多该例子。定义的结构常常用!CPP宏包起来以不在C++中参与编译。因此一般是只给引擎内部使用的。

实际上我们想使用也可以，只要保持C++中内存布局一样，就可以自己多定义。使用场景：想自己定义一个UHT头喂给UHT分析，然后自己在别处定义实际的C++。一种典型用途是C++里的实际多个类继承于一个模板基类，如FVector2MaterialInput，这样可以每个特化子类定一个UHT类型别名。另一种目的是把UHT要分析的头文件都放在一个文件里，加速UHT分析生成，不用分析多个文件，反正只要UHT信息和内存布局对就行。

```

#ifndef !CPP // begin noexport class
USTRUCT(noexport, BlueprintType) //如果不写noexport，会报错: Expected a
GENERATED_BODY() at the start of the struct.
struct FFloatRK4SpringInterpolator
{

    UPROPERTY(EditAnywhere, Category = "FloatRK4SpringInterpolator")
    float StiffnessConstant;

    /** 0 = Undamped, <1 = Underdamped, 1 = Critically damped, >1 = Over damped */
    UPROPERTY(EditAnywhere, Category = "FloatRK4SpringInterpolator")
    float DampeningRatio;

    bool bIsInitialized;
    bool bIsInMotion;
    float TimeRemaining;
    FRK4SpringConstants SpringConstants;

    float LastPosition;
    RK4Integrator::FRK4State<float> State;
};

#endif // end noexport class

//实际应用:
template <typename T>
struct FRK4SpringInterpolator
{
protected:

```

```

float StiffnessConstant;
float DampeningRatio;

bool bIsInitialized;
bool bIsInMotion;
float TimeRemaining;
FRK4SpringConstants SpringConstants;

T LastPosition;
RK4Integrator::FRK4State<T> State;
}

struct FFloatRK4SpringInterpolator : FRK4SpringInterpolator<float>
struct FVectorRK4SpringInterpolator : FRK4SpringInterpolator<FVector>

```

不生成的代码包括：

```

USTRUCT(BlueprintType,noexport)
struct INSIDER_API FMyStruct_NoExport
{
    //抑制: GENERATED_BODY() 解释生成的:
    //static class UScriptStruct* StaticStruct();

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

//抑制:
//template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>()

```

.h里不会生成，因此不会在别的模块里使用

```
template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>();
```

但是依然会在Module.init.gen.cpp里生成Z_Construct_UScriptStruct_FMyStruct_NoExport的调用，因此还是会在蓝图里暴露出来。

```

#include "UObject/GeneratedCppIncludes.h"
PRAGMA_DISABLE_DEPRECATED_WARNINGS
void EmptyLinkFunctionForGeneratedCodeInsider_init() {}
INSIDER_API UScriptStruct* Z_Construct_UScriptStruct_FMyStruct_NoExport();
static FPackageRegistrationInfo Z_Registration_Info_UPackage__Script_Insider;
FORCEINLINE UPackage* Z_Construct_UPackage__Script_Insider()
{
    if (!Z_Registration_Info_UPackage__Script_Insider.OuterSingleton)
    {
        static UObject* (*const SingletonFuncArray[])() = {
            (uobject* (*)()
        })();
    }
    Z_Construct_UScriptStruct_FMyStruct_NoExport, //这里注入调用
}
static const UECodeGen_Private::FPackageParams PackageParams = {
    "/Script/Insider",
    SingletonFuncArray,

```

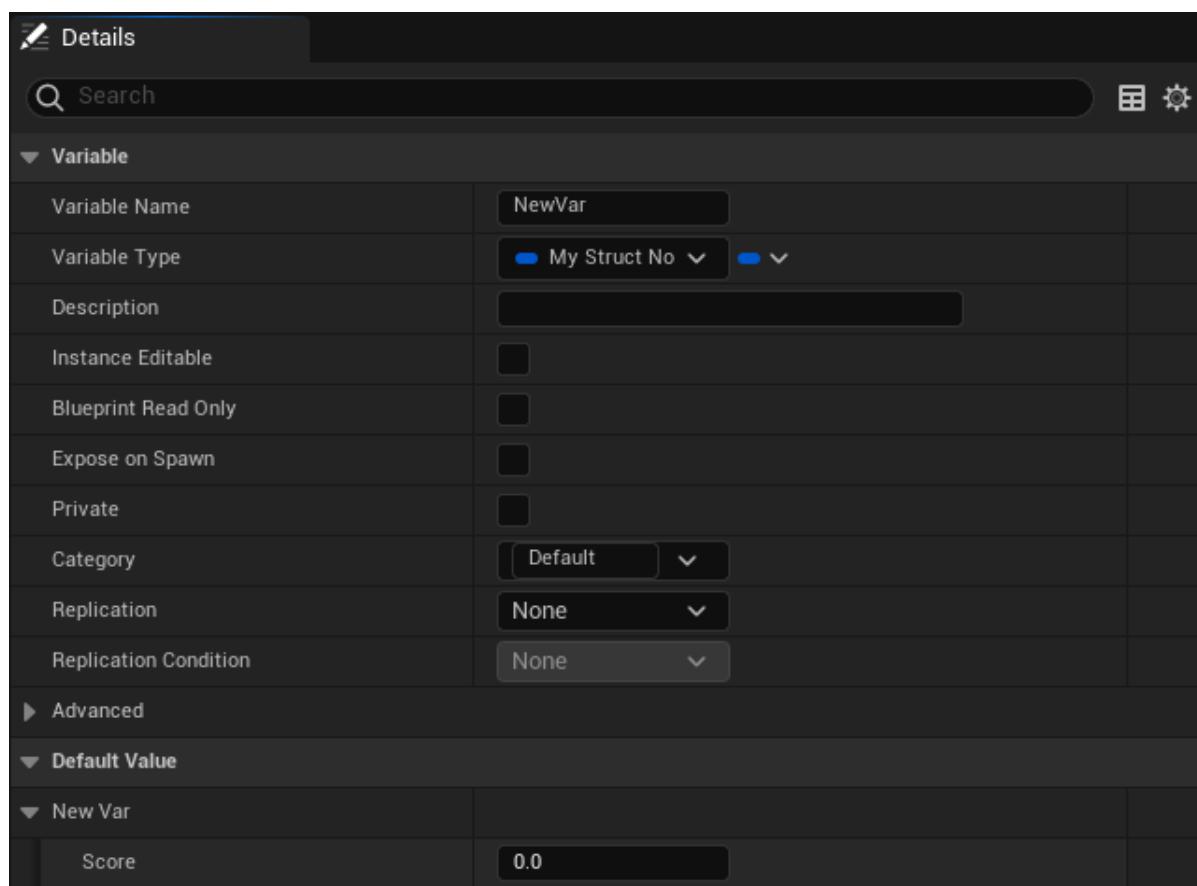
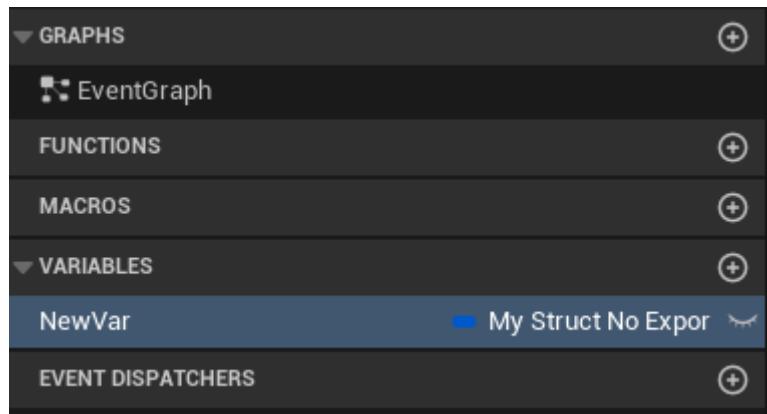
```

        UE_ARRAY_COUNT(SingletonFuncArray),
        PKG_CompiledIn | 0x00000000,
        0x02A7B98C,
        0xFA17C3C4,
        METADATA_PARAMS(0, nullptr)
    };
}

UECodeGen_Private::ConstructUPackage(Z_Registration_Info_UPackage__Script_Insider
.OuterSingleton, PackageParams);
}
return Z_Registration_Info_UPackage__Script_Insider.OuterSingleton;
}
static FRegisterCompiledInInfo
Z_CompiledInDeferPackage_UPackage__Script_Insider(Z_Construct_UPackage__Script_In
sider, TEXT("/Script/Insider"), Z_Registration_Info_UPackage__Script_Insider,
CONSTRUCT_RELOAD_VERSION_INFO(FPackageReloadVersionInfo, 0x02A7B98C,
0xFA17C3C4));
PRAGMA_ENABLE_DEPRECATED_WARNINGS

```

蓝图里的效果：依然可以当作变量。



加上noexport的区别是不能用StaticStruct和没了TCppStructOps，不能做一些优化。其他还是可以正常使用，就像FVector一样。

缺失的代码，也可以通过手动添加代码来获得。

```
USTRUCT(BlueprintType,noexport)
struct INSIDER_API FMyStruct_NoExport
{
    //GENERATED_BODY() //missing type specifier - int assumed, ..generated.h里只是定一个StaticStruct()函数

    static class UScriptStruct* StaticStruct(); //可以自己定义

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;

};

template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>
() //可以自己定义

//.cpp
//链入函数声明，在其他的cpp里已经有实现，所以可以正常调用到
INSIDER_API UScriptStruct* Z_Construct_UScriptStruct_FMyStruct_NoExport();
UPackage* Z_Construct_UPackage__Script_Insider();

static FStructRegistrationInfo
Z_Registration_Info_UScriptStruct_MyStruct_NoExport;

class UScriptStruct* FMyStruct_NoExport::StaticStruct()
{
    if (!Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton)
    {
        Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton =
GetStaticStruct(Z_Construct_UScriptStruct_FMyStruct_NoExport,
Z_Construct_UPackage__Script_Insider(), TEXT("MyStruct_NoExport"));
    }
    return Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton;
}

template<> INSIDER_API UScriptStruct* StaticStruct<FMyStruct_NoExport>()
{
    return FMyStruct_NoExport::StaticStruct();
}
```

BlueprintType

- **功能描述：** 可以作为蓝图变量
- **元数据类型：** bool
- **引擎模块：** Blueprint
- **作用机制：** 在Meta中增加BlueprintType
- **常用程度：** ★★★★☆

和其他地方的BlueprintType用法一样。

Flags

- **功能描述:** 把该枚举的值作为一个标志来拼接字符串输出。
- **元数据类型:** bool
- **引擎模块:** Trait
- **作用机制:** 在EnumFlags中添加Flags
- **常用程度:** ★★★★☆

把该枚举的值作为一个标志来拼接字符串输出。

指定作用的地方是当把一个值输出为字符串输出的时候。转换成字符串的方式：一种是直接查找精确相等的值，然后查找特定的枚举值。第二种就是把它当做标志，输出符合相应标记的“A | B | C”这种格式的值。Flags就是指定采用第二种方式。

但是要注意其枚举值是完全没有影响变化的。跟直接把枚举值定义为标记是不同的。

注意和meta(bitflags)的区别，后者是标记该枚举可以作为一个标记，可以作为Bitmask被筛选中。

示例代码：

```
UENUM(BlueprintType)
enum class EMyEnum_Normal: uint8
{
    First,
    Second,
    Third,
};

/*
[EMyEnum_Flags Enum->Field->Object /Script/Insider.EMyEnum_Flags]
(BlueprintType = true, First.Name = EMyEnum_Flags::First, ModuleRelativePath =
Enum/MyEnum_Flags.h, Second.Name = EMyEnum_Flags::Second, Third.Name =
EMyEnum_Flags::Third)
    ObjectFlags:    RF_Public | RF_Transient
    Outer:    Package /Script/Insider
    EnumFlags:  EEnumFlags::Flags
    EnumDisplayNameFn: 0
    CppType:   EMyEnum_Flags
    CppForm:   EnumClass
{
    First = 0,
    Second = 1,
    Third = 2,
    EMyEnum_MAX = 3
};
*/
UENUM(BlueprintType, Flags)
enum class EMyEnum_Flags: uint8
{
    First,
    Second,
    Third,
```

```

};

void UMyActor_EnumBitFlags_Test::TestFlags()
{
    int value = 3;

    FString outStr_Normal = StaticEnum<EMyEnum_Normal>()-
>GetValueOrBitfieldAsString(value);
    FString outStr_Flags = StaticEnum<EMyEnum_Flags>()-
>GetValueOrBitfieldAsString(value);
    FString outStr_BitFlags = StaticEnum<EMyEnum_BitFlags>()-
>GetValueOrBitfieldAsString(value);

}

```

示例效果：

蓝图中的表示，依然只能选择单项。



而测试代码里打印出来的字符串：

可见outStr_Flags 的打印是字符串拼接的。



原理：

只在GetValueOrBitfieldAsString这个函数中生效，所以要用这个方法测试才生效。

```

FString UEnum::GetValueOrBitfieldAsString(int64 InValue) const
{
    if (!HasAnyEnumFlags(EEEnumFlags::Flags) || InValue == 0)
    {
        return GetNameStringByValue(InValue);
    }
    else
    {
        FString BitfieldString;
        bool WroteFirstFlag = false;
        while (InValue != 0)
        {
            int64 NextValue = 111 << FMath::CountTrailingZeros64(InValue);
            InValue = InValue & ~NextValue;
            if (WroteFirstFlag)
            {
                // We don't just want to use the NameValuePair.Key because we
                // want to strip enum class prefixes
                BitfieldString.Appendf(TEXT(" | %s"),
*GetNameStringByValue(NextValue));
            }
            else
            {
                // We don't just want to use the NameValuePair.Key because we
                // want to strip enum class prefixes
            }
        }
    }
}

```

```

        BitfieldString.Appendf(TEXT("%s"),
*GetNameStringByValue(NextValue));
        WroteFirstFlag = true;
    }
}
return BitfieldString;
}
}

```

BlueprintCallable

- 功能描述:** 暴露到蓝图中可被调用
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 作用机制:** 在FunctionFlags增加FUNC_BlueprintCallable
- 常用程度:** ★★★★☆

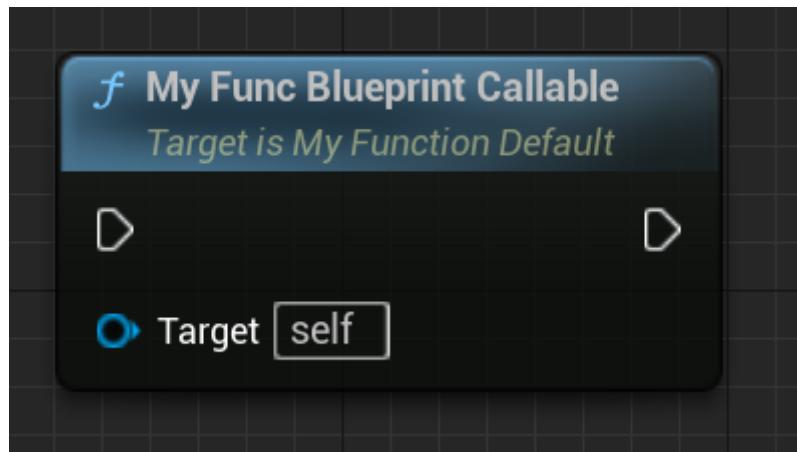
测试代码:

```

UFUNCTION(BlueprintCallable)
void MyFunc_BlueprintCallable() {}

```

效果展示:



BlueprintGetter

- 功能描述:** 指定该函数作为属性的自定义Get函数。
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 作用机制:** 在Meta中加入BlueprintGetter, 在FunctionFlags加入FUNC_BlueprintCallable, FUNC_BlueprintPure
- 常用程度:** ★★

指定该函数作为属性的自定义Get函数。

此说明符隐含BlueprintPure和BlueprintCallable。

更多可以参考UPROPERTY的BlueprintGetter

BlueprintImplementableEvent

- **功能描述:** 指定一个函数调用点，可以在蓝图中重载实现。
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在FunctionFlags中增加FUNC_Event、FUNC_Native、FUNC_BlueprintEvent
- **常用程度:** ★★★★☆

指定一个函数调用点，可以在蓝图中重载实现。是一种方便的用来实现C++来调用蓝图函数的方式。

蓝图中如果没提供实现，调用的话相当于调用空函数。

BlueprintImplementableEvent也要配合BlueprintCallable使用，如果没加BlueprintCallable的话就只能在CPP里调用，在蓝图会发现找不到Call Function的节点。

测试代码：

```
//FunctionFlags: FUNC_Event | FUNC_Public | FUNC_BlueprintCallable |  
FUNC_BlueprintEvent  
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)  
void MyFunc_ImplementableEvent();
```

效果展示：

右键可添加自定义实现



原理：

在C++里调用的时候，里面会FindFunctionChecked根据名字寻找。如果蓝图中有找到的话，则会调用。如果在蓝图中直接调用，则其实是会直接FindFunctionChecked查找，蓝图中有定义的话则会被直接找到。

```
void AMyFunction_Default::MyFunc_ImplementableEvent()  
{  
  
ProcessEvent(FindFunctionChecked(NAME_AMyFunction_Default_MyFunc_ImplementableEvent),NULL);  
}
```

BlueprintNativeEvent

- **功能描述:** 可以在蓝图总覆盖实现，但是也在C++中提供一个默认实现。

- 元数据类型: bool
- 引擎模块: Blueprint
- 作用机制: 在FunctionFlags中增加FUNC_Event、FUNC_BlueprintEvent
- 常用程度: ★★★★☆

可以在蓝图总覆盖实现，但是也在C++中提供一个默认实现。

需要在CPP中声明名称与主函数相同的附加函数，但是末尾添加了*Implementation*。如果未找到任何蓝图覆盖，该自动生成的代码将调用“*[FunctionName]Implementation*”方法。一般用在OnXXX之类的函数上，在C++提供实现，这样如果蓝图中没有覆盖的时候，就可以默认调用C++中默认实现版本。

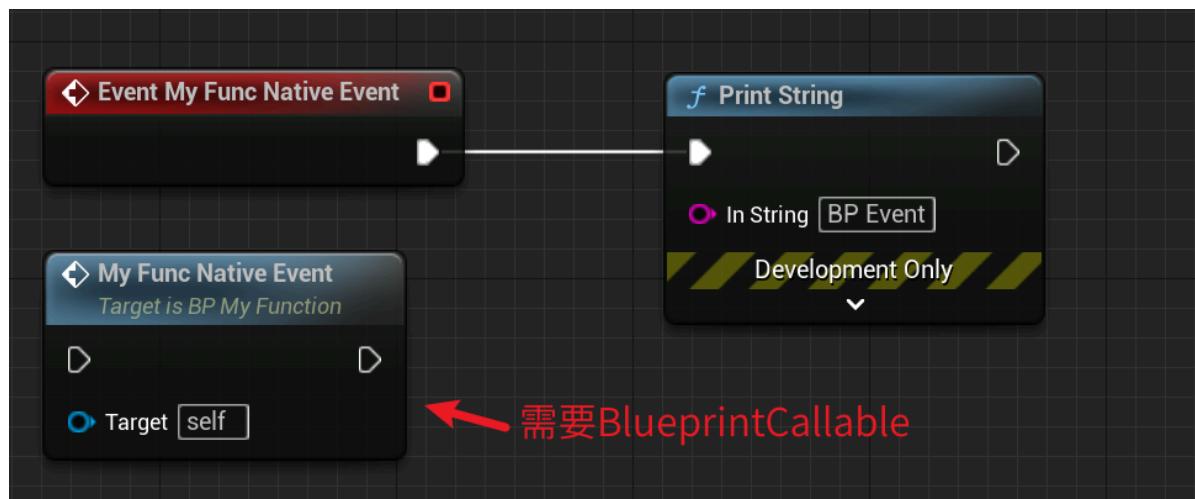
BlueprintNativeEvent，没加BlueprintCallable的话就只能在CPP里调用，因此一般也要配合加上BlueprintCallable。

测试代码：

```
//FunctionFlags: FUNC_Native | FUNC_Event | FUNC_Public |
FUNC_BlueprintCallable | FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
void MyFunc_NativeEvent();

void AMyFunction_Default::MyFunc_NativeEvent_Implementation()
{
    GEngine->AddOnScreenDebugMessage(-1, 3.f, FColor::Red,
"MyFunc_NativeEvent_Implementation");
}
```

效果展示：



原理：

在调用MyFunc_NativeEvent的时候，内部FindFunctionChecked会根据名字查找，如果在蓝图中有定义，则会找到蓝图中的实现版本。否则的话，则会找到execMyFunc_NativeEvent这个实现版本，从而调用MyFunc_NativeEvent_Implementation。

```

DEFINE_FUNCTION(AMyFunction_Default::execMyFunc_NativeEvent)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    P_THIS->MyFunc_NativeEvent_Implementation();
    P_NATIVE_END;
}

void AMyFunction_Default::MyFunc_NativeEvent()
{
    ProcessEvent(FindFunctionChecked(NAME_AMyFunction_Default_MyFunc_NativeEvent), NULL);
}

```

BlueprintPure

- 功能描述:** 指定作为一个纯函数，一般用于Get函数用来返回值。
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 作用机制:** 在FunctionFlags增加FUNC_BlueprintCallable、FUNC_BlueprintPure
- 常用程度:** ★★★★☆

指定作为一个纯函数，一般用于Get函数用来返回值。

- 纯函数是指没有执行引脚的函数，不是指const函数。
- 纯函数可以有多个返回值，用引用参数加到函数里就行。
- 不能用于void函数，否则会报错“error : BlueprintPure specifier is not allowed for functions with no return value and no output parameters.”

测试代码：

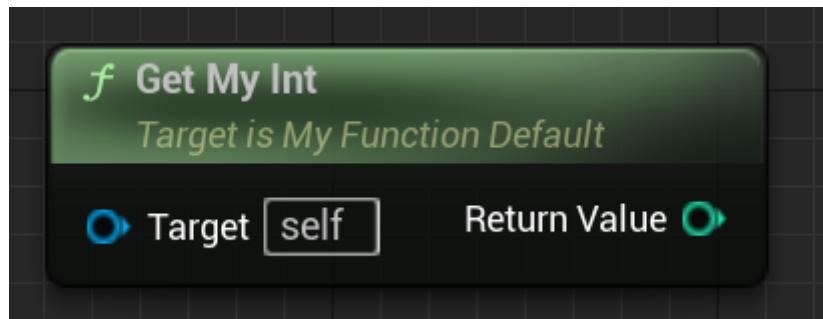
```

UFUNCTION(BlueprintPure)
    int32 GetMyInt() const { return MyInt; }

private:
    int32 MyInt;

```

效果展示：



BlueprintSetter

- **功能描述:** 指定该函数作为属性的自定义Set函数。
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **作用机制:** 在Meta中加入BlueprintSetter，在FunctionFlags中加入FUNC_BlueprintCallable
- **常用程度:** ★★

指定该函数作为属性的自定义Set函数。

此说明符隐含BlueprintCallable。

更多可以参考UPROPERTY的BlueprintSetter

CallInEditor

- **功能描述:** 可以在属性细节面板上作为一个按钮来调用该函数。
- **元数据类型:** bool
- **引擎模块:** Editor
- **作用机制:** 在Meta中增加CallInEditor
- **常用程度:** ★★★★★

可以在属性细节面板上作为一个按钮来调用该函数。

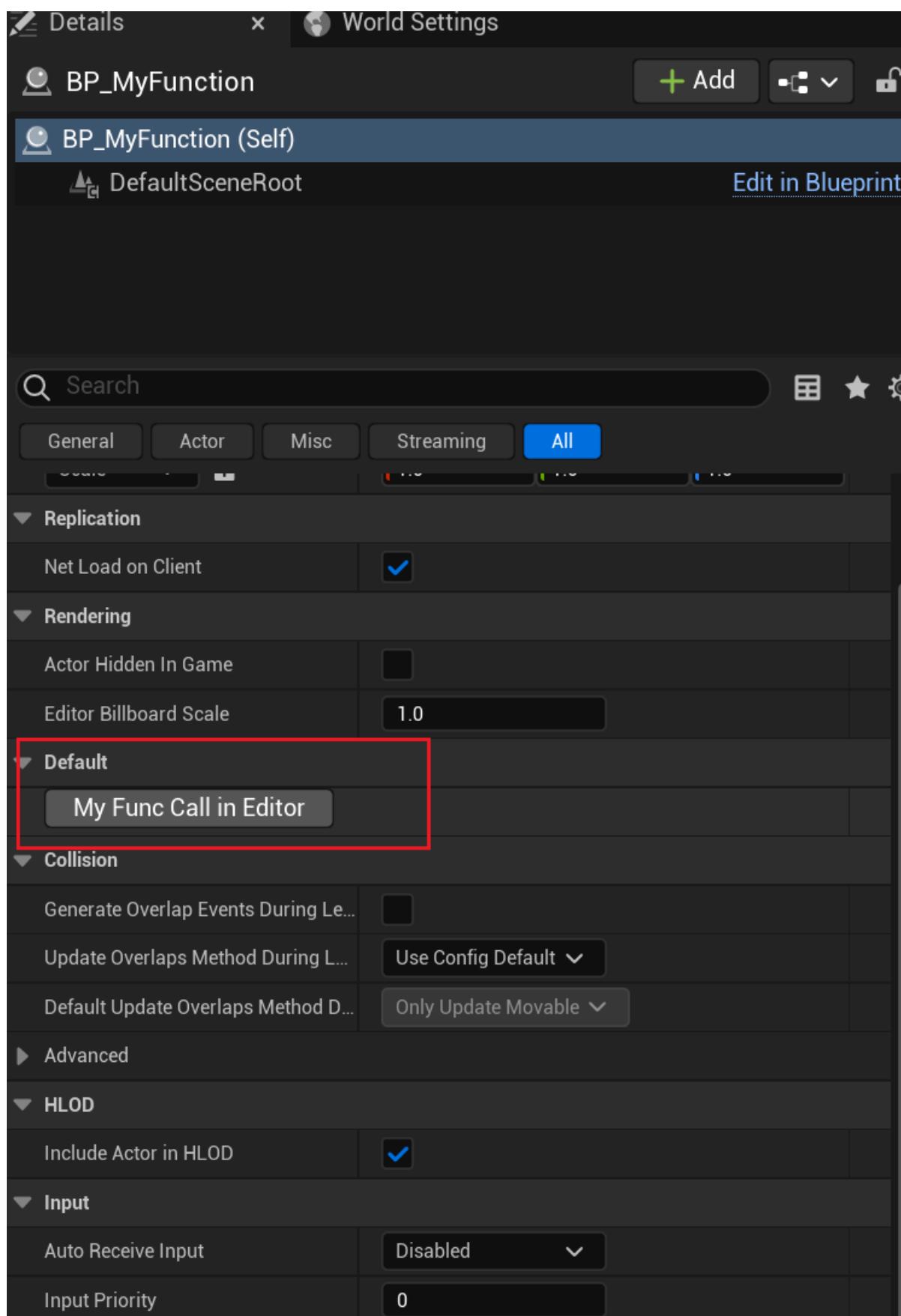
该函数写在AActor或UObject子类里都是可以的，只要有对应的属性细节面板。

注意这一般是处于Editor运行环境的。典型的例子是ASkyLight的Recapture按钮。因此函数里有时会调用编辑器环境下函数。但也要注意不要在runtime下混用了，比较容易出错。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Default :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(CallInEditor)
    void MyFunc_CallInEditor(){}
};
```

蓝图展示：



SealedEvent

- 功能描述:** 无法在子类中覆盖此函数。SealedEvent关键词只能用于事件。对于非事件函数，请将它们声明为static或final，以密封它们。
- 元数据类型:** bool

- 引擎模块: Behavior
- 作用机制: 在FunctionFlags中添加FUNC_Final

在源码里搜索: 发现都是用在网络的函数上

```
▲ Engine\Classes\GameFramework (8)
  ▲ HUD.h (3)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)

  ▲ PlayerController.h (5)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)
```

UHT中的处理:

```
//先识别符号
[UhtSpecifier(Extends = UhtTableNames.Function, valueType =
UhtSpecifierValueType.Legacy)]
private static void SealedEventSpecifier(UhtSpecifierContext specifierContext)
{
    UhtFunction function = (UhtFunction)specifierContext.Type;
    function.FunctionExportFlags |= UhtFunctionExportFlags.SealedEvent;
}

//再设置标记
// Handle the initial implicit/explicit final
// A user can still specify an explicit final after the parameter list as well.
if (automaticallyFinal ||
function.FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.SealedEvent))
{
    function.FunctionFlags |= EFunctionFlags.Final;
    function.FunctionExportFlags |= UhtFunctionExportFlags.Final | 
UhtFunctionExportFlags.AutoFinal;
}
```

再验证: 限定只能用在Event上。

```
if (FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.SealedEvent) &&
!FunctionFlags.HasAnyFlags(EFunctionFlags.Event))
{
    this.LogError("SealedEvent may only be used on events");
}

if (FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.SealedEvent) &&
FunctionFlags.HasAnyFlags(EFunctionFlags.BlueprintEvent))
{
    this.LogError("SealedEvent cannot be used on Blueprint events");
}
```

测试代码：

```
//Error: "SealedEvent may only be used on events"
UFUNCTION(SealedEvent)
void MyFunc_SealedEvent() {}

//Error: "SealedEvent cannot be used on Blueprint events"
UFUNCTION(BlueprintCallable,BlueprintImplementableEvent,SealedEvent)
void MyFunc_ImplementableEvent();

//Error: "SealedEvent cannot be used on Blueprint events"
UFUNCTION(BlueprintCallable,BlueprintNativeEvent,SealedEvent)
void MyFunc_NativeEvent();
```

因此无法用于普通的函数，又无法用于蓝图中的Event。所以既是Event又不是BlueprintEvent的是什么？看源码是只有网络的一些函数。

通过对比，发现Sealed函数的区别是多了FUNC_Final的标记。但FUNC_Final又不一定必须要以SealedEvent才能添加，exec或普通的BlueprintCallable函数都会添加。但是如果是virtual的函数就不会添加。在UHT中的原理是：

```
private static UhtParseResult ParseUFunction(UhtParsingscope parentScope,
UhtToken token)
{
    if (function.FunctionFlags.HasAnyFlags(EFunctionFlags.Net))
    {
        // Network replicated functions are always events, and
        // are only final if sealed
        scopeName = "event";
        tokenContext.Reset(scopeName);
        automaticallyFinal = false;
    }

    // If virtual, remove the implicit final, the user can still specifying
    // an explicit final at the end of the declaration
    if
(function.FunctionExportFlags.HasAnyFlags(UhtFunctionFlags.Virtual))
    {
        automaticallyFinal = false;
    }
    // Handle the initial implicit/explicit final
    // A user can still specify an explicit final after the parameter list as
    // well.
    if (automaticallyFinal ||
function.FunctionExportFlags.HasAnyFlags(UhtFunctionFlags.SealedEvent))
    {
        function.FunctionFlags |= EFunctionFlags.Final;
        function.FunctionExportFlags |=
UhtFunctionFlags.Final | UhtFunctionFlags.AutoFinal;
    }

}
```

在自己的C++代码中测试，发现在C++中怎么继承都不会触发编译错误。因此如果想拒绝被继承，还是用C++标准是final关键字。在函数末尾加final。

E:\P4V\Engine\Source\Editor\KismetCompiler\Private\KismetCompiler.cpp

```
const uint32 overrideFlagsToCheck = (FUNC_FuncOverrideMatch &
~FUNC_AccessSpecifiers);
if ((Context.Function->FunctionFlags & overrideFlagsToCheck) !=
(OverriddenFunction->FunctionFlags & overrideFlagsToCheck))
{
    MessageLog.Error(*LOCTEXT("IncompatibleOverrideFlags_Error", "Overridden
function is not compatible with the parent function @@. Check flags: Exec, Final,
Static.").ToString(), Context.EntryPoint);
}
```

在编译的时候检测的是是否是重载父类的函数，但因为SealedEvent不作用于普通函数，也不作用于BlueprintEvent，因此感觉只能在C++中继承。

Category

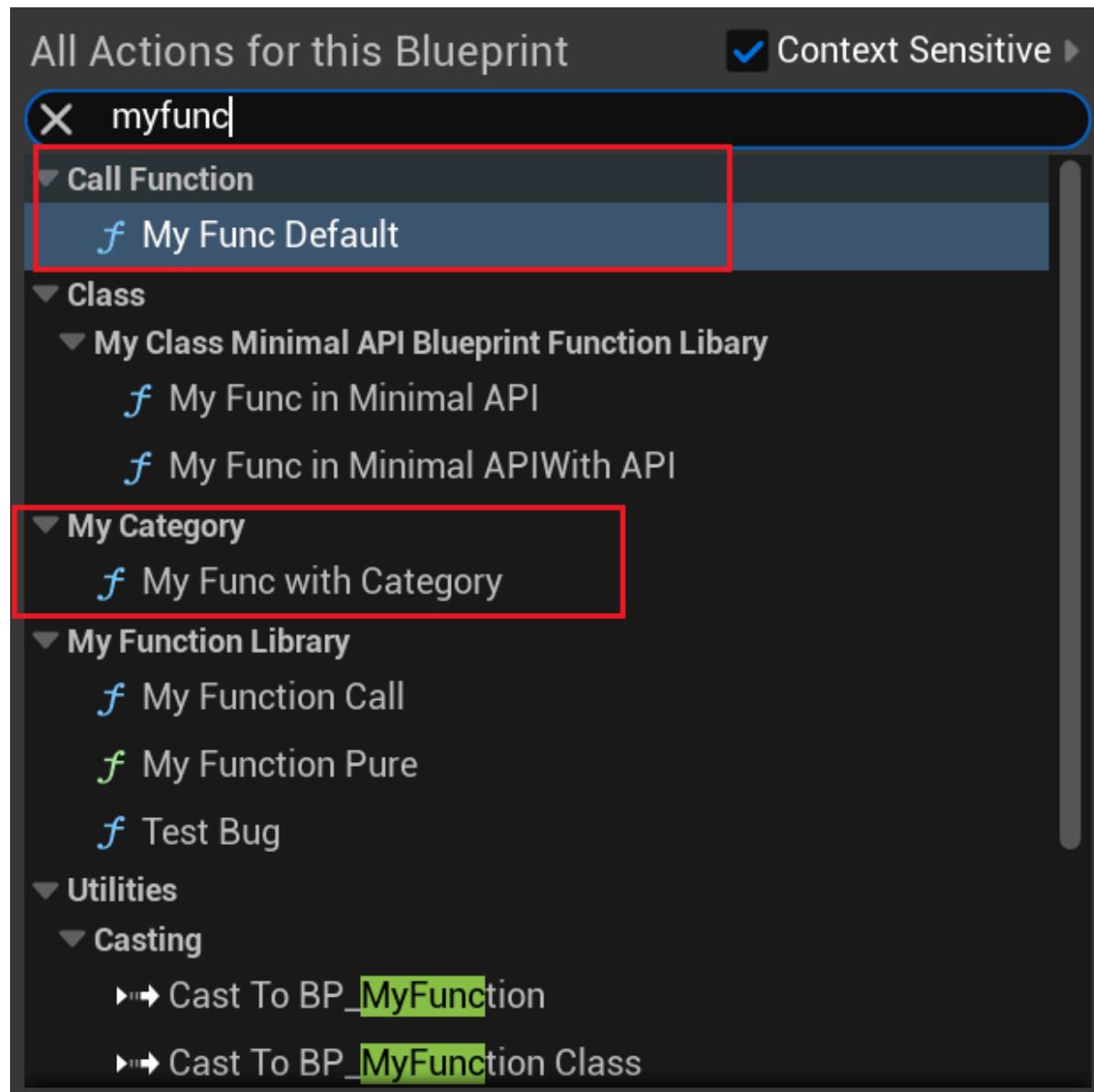
- **功能描述：** 在蓝图的右键菜单中为该函数指定类别分组，可以嵌套多级
- **元数据类型：** strings="a|b|c"
- **引擎模块：** Editor
- **作用机制：** 在Meta中加入Category
- **常用程度：** ★★★★☆

在蓝图的右键菜单中为该函数指定类别分组。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Default :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, Category = MyCategory)
    void MyFunc_WithCategory(){}
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default(){}
};
```

蓝图中的展示：



Exec

- **功能描述:** 在特定类里注册一个函数为作为控制台命令，允许接受参数。
- **元数据类型:** bool
- **引擎模块:** Behavior
- **限制类型:** 特定的几个类
- **作用机制:** 在FunctionFlags中加入FUNC_Exec
- **常用程度:** ★★★

一般特定的几个类是：UPlayerInput, APlayerController, APawn, AHUD, AGameModeBase, ACheatManager, AGameStateBase, APlayerCameraManager的子类。

当在视口中输入控制台命令后，首先执行到的是UConsole::ConsoleCommand，然后是APlayerController::ConsoleCommand，然后是UPlayer::ConsoleCommand，中间先尝试ViewportClient->Exec（可能处理一些编辑器命令），然后到达ULocalPlayer::Exec（已经处理一些自定义命令了）。

UGameViewportClient, UGameInstance, UPlayer是继承于FExec的，因此本身含有一些Exec, Exec_Runtime, Exec_Dev, Exec_Editor的4个虚函数重载。

其中UEngine::Exec, 内部会转发给各个模块来尝试。其中重要的是StaticExec, 最后会 FSelfRegisteringExec::StaticExec(InWorld, Cmd, Ar)来调用自注册的Exec。

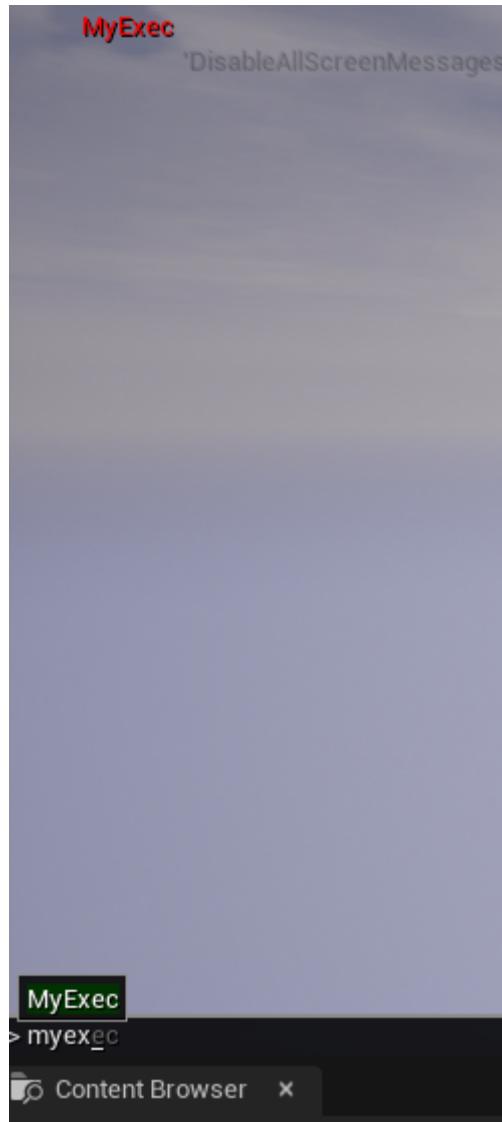
如果是在编辑器中~执行命令, FConsoleCommandExecutor::ExecInternal, 最后也会到 ULocalPlayer::Exec。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Exec :public APawn
{
public:
    GENERATED_BODY()
public:
    //FunctionFlags: FUNC_Final | FUNC_Exec | FUNC_Native | FUNC_Public
    UFUNCTION(exec)
    void MyExec();
};

void AMyFunction_Exec::MyExec()
{
    GEngine->AddOnScreenDebugMessage(-1, 3.f, FColor::Red, "MyExec");
}
```

在PIE的时候~打开控制台运行结果：



原理：

根据源码中的流程：

```
bool UGameViewportClient::Exec(Uworld* InWorld, const TCHAR* Cmd, FOutputDevice& Ar)
{
    //按顺序ULocalPlayer::Exec_Editor, Exec_Dev, Exec_Runtime, 各自判断是否是一些命令
    if (FExec::Exec(InWorld, Cmd, Ar))
    {
        return true;
    }
    else if (ProcessConsoleExec(Cmd, Ar, NULL))
    {
        return true;
    }
    else if (GameInstance && (GameInstance->Exec(InWorld, Cmd, Ar) ||
GameInstance->ProcessConsoleExec(Cmd, Ar, nullptr)))
    {
        return true;
    }
    else if (GEngine->Exec(InWorld, Cmd, Ar))
    {
        return true;
    }
}
```

```

    }

    else
    {
        return false;
    }
}

bool UPlayer::Exec( Uworld* InWorld, const TCHAR* Cmd, FOutputDevice& Ar)
{
    // Route through Exec_Dev and Exec_Editor first
    //按顺序ULocalPlayer::Exec_Editor, Exec_Dev, Exec_Runtime, 各自判断是否是一些命令
    if (FExec::Exec(InWorld, Cmd, Ar))
    {
        return true;
    }

    AActor* ExecActor = PlayerController;
    if (!ExecActor)
    {
        UNetConnection* NetConn = Cast<UNetConnection>(this);
        ExecActor = (NetConn && NetConn->OwningActor) ? ToRawPtr(NetConn-
>OwningActor) : nullptr;
    }

    if (ExecActor)
    {
        // Since UGameViewportClient calls Exec on Uworld, we only need to
        // explicitly
        // call Uworld::Exec if we either have a null GEngine or a null
        ViewportClient
        Uworld* World = ExecActor->GetWorld();
        check(World);
        check(InWorld == nullptr || InWorld == World);
        const bool bWorldNeedsExec = GEngine == nullptr || Cast<ULocalPlayer>
(this) == nullptr || static_cast<ULocalPlayer*>(this)->ViewportClient == nullptr;
        APawn* PCPawn = PlayerController ? PlayerController->GetPawnOrSpectator()
: nullptr;
        if (bWorldNeedsExec && World->Exec(World, Cmd, Ar))
        {
            return true;
        }
        else if (PlayerController && PlayerController->PlayerInput &&
PlayerController->PlayerInput->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
        else if (ExecActor->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
        else if (PCPawn && PCPawn->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
        else if (PlayerController && PlayerController->MyHUD && PlayerController-
>MyHUD->ProcessConsoleExec(Cmd, Ar, PCPawn))
    }
}

```

```

    {
        return true;
    }
    else if (World->GetAuthGameMode() && World->GetAuthGameMode()-
>ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (PlayerController && PlayerController->CheatManager &&
PlayerController->CheatManager->ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (World->GetGameState() && World->GetGameState()-
>ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (PlayerController && PlayerController->PlayerCameraManager &&
PlayerController->PlayerCameraManager->ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
}
return false;
}

```

查找Exec的顺序应该是：

- UGameInstance::Exec, UGameInstance::ProcessConsoleExec
- GEngine->Exec(InWorld, Cmd, Ar)
- UWorld::Exec, 在没有LocalPlayer处理的情况下
- UPlayerInput::ProcessConsoleExec
- APlayerController::ProcessConsoleExec
- APawn::ProcessConsoleExec
- AHUD::ProcessConsoleExec
- AGameModeBase::ProcessConsoleExec
- ACheatManager::ProcessConsoleExec
- AGameStateBase::ProcessConsoleExec
- APlayerCameraManager::ProcessConsoleExec

ProcessConsoleExec内部会调用CallFunctionByNameWithArguments代码：因此确实会限制这种方式声明的Exec只能在以上几个类里面

```

bool UObject::CallFunctionByNameWithArguments(const TCHAR* Str, FOutputDevice&
Ar, UObject* Executor, bool bForceCallWithNonExec/*=false*/)
{
    UFunction* Function = FindFunction(Message); //寻找函数
}

```

BlueprintAuthorityOnly

- **功能描述:** 这个函数只能在拥有网络权限的端上运行。
- **元数据类型:** bool
- **引擎模块:** Network
- **作用机制:** 在FunctionFlags中添加FUNC_BlueprintAuthorityOnly
- **常用程度:** ★★★

这个函数只能在拥有网络权限的端上运行。 HasAuthority:: (GetLocalRole() == ROLE_Authority) 。共有4种NetRole: ROLE_None (不复制) , ROLE_SimulatedProxy (在客户端上模拟的代理) , ROLE_AutonomousProxy (在客户端上的匿名代理, 接收玩家输入) , ROLE_Authority (服务器拥有权限的) 。

因此BlueprintAuthorityOnly限定这个函数只能在服务器上运行, 这个“服务器”可以是LS服务器, DS服务器, 单机 (可以看作没有客户端的服务器) 。

注意在测试的时候需要把该Actor设置为Replicates。

测试代码:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()
public:
    //FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
    FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default();

    //FunctionFlags: FUNC_Final | FUNC_BlueprintAuthorityOnly | FUNC_Native | 
    FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable, BlueprintAuthorityOnly)
    void MyFunc_BlueprintAuthorityOnly();

    static void PrintFuncStatus(AActor* actor, FString funcName);
};

void AMyFunction_Network::MyFunc_Default()
{
    PrintFuncStatus(this, TEXT("MyFunc_Default"));
}

void AMyFunction_Network::MyFunc_BlueprintAuthorityOnly()
{
    PrintFuncStatus(this, TEXT("MyFunc_BlueprintAuthorityOnly"));
}

void AMyFunction_Network::PrintFuncStatus(AActor* actor, FString funcName)
{
    FString actorName = actor->GetName();
```

```

FString localRoleStr;
UEnum::GetValueAsString(actor->GetLocalRole(), localRoleStr);

FString remoteRoleStr;
UEnum::GetValueAsString(actor->GetRemoteRole(), remoteRoleStr);

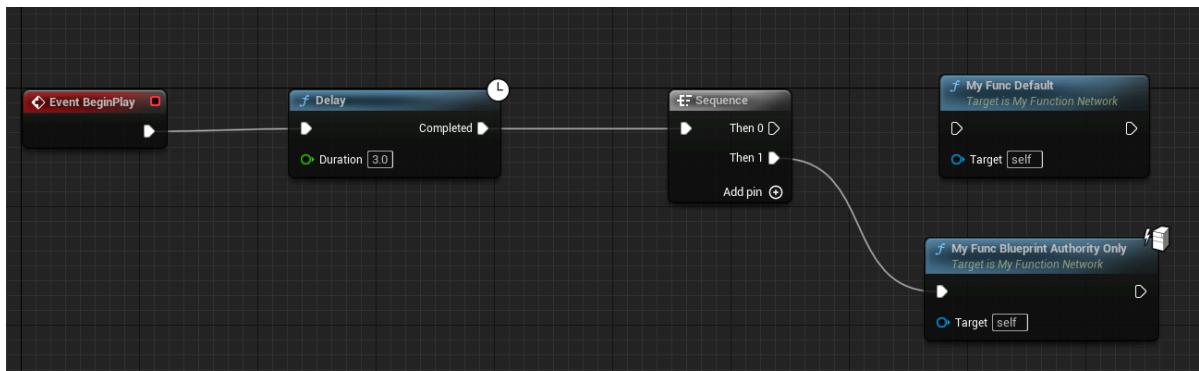
FString netModeStr = Insider::NetModeToString(actor->GetNetMode());

FString str = FString::Printf(TEXT("%s\t%s\t%s\tLocal:%s\tRemote:%s"),
*funcName, *actorName, *netModeStr, *localRoleStr, *remoteRoleStr);
GEngine->AddOnScreenDebugMessage(-1, 20.f, FColor::Red, str);

UE_LOG(LogInsider, Display, TEXT("%s"), *str);
}

```

蓝图代码：



对于不Replicated的Actor：

```

MyFunc_Default BP_Network_C_1 NM_ListenServer Local:ROLE_Authority
Remote:ROLE_None
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_None Remote:ROLE_Authority
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_None Remote:ROLE_Authority

```

而对于Replicated的Actor，同时有1个S和两个C，运行普通的函数：

```

MyFunc_Default BP_Network_C_1 NM_ListenServer Local:ROLE_Authority
Remote:ROLE_SimulatedProxy
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy
Remote:ROLE_Authority
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy
Remote:ROLE_Authority

```

如果允许的BlueprintAuthorityOnly函数：

```

MyFunc_BlueprintAuthorityOnly BP_Network_C_1 NM_ListenServer
Local:ROLE_Authority Remote:ROLE_SimulatedProxy

```

结果可见，Default的函数在3个端上都可以运行，而BlueprintAuthorityOnly只能在服务器上运行。而Client上无法运行。

原理：

```
int32 AActor::GetFunctionCallspace( UFunction* Function, FFrame* Stack )
{
    FunctionCallspace::Type Callspace = (LocalRole < ROLE_Authority) && Function->HasAllFunctionFlags(FUNC_BlueprintAuthorityOnly) ? FunctionCallspace::Absorbed : FunctionCallspace::Local;
}
```

BlueprintCosmetic

- **功能描述：**此函数为修饰性的，无法在DS上运行。
- **元数据类型：**bool
- **引擎模块：**Network
- **作用机制：**在FunctionFlags中加入FUNC_BlueprintCosmetic
- **常用程度：**★★★

这个函数是修饰性的，所谓修饰性是指这个函数的内容是为了展现一些与逻辑无关的内容，比如动画音效特效等。因为DS并没有实际的画面输出，因此这些修饰性的函数是对DS无意义的。因此这些修饰性函数会被无视掉。

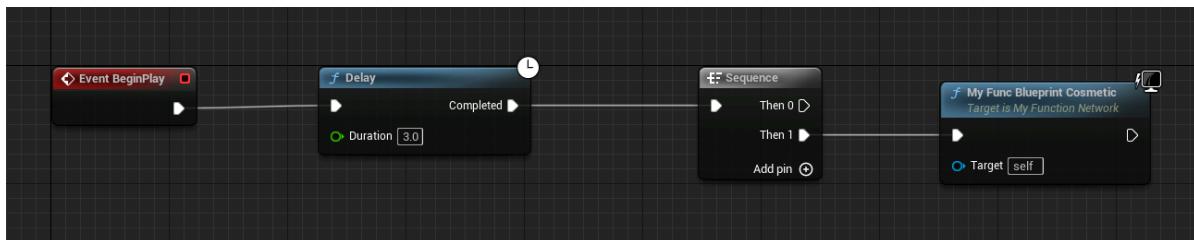
但是也注意在ListenServer或Client上，这二者都会允许运行。因为这两个端都需要画面展示。

测试代码：

```
UFUNCTION(BlueprintCallable, BlueprintCosmetic)
void MyFunc_BlueprintCosmetic();
```

测试蓝图：

节点上的电脑标记就是意味着只在客户端上运行。



结果输出

```
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_ListenServer  Local:ROLE_Authority
                               Remote:ROLE_SimulatedProxy
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_Client       Local:ROLE_SimulatedProxy
                               Remote:ROLE_Authority
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_Client       Local:ROLE_SimulatedProxy
                               Remote:ROLE_Authority
```

原理：

```
int32 AActor::GetFunctionCallspace( UFunction* Function, FFrame* Stack )
{
// Dedicated servers don't care about "cosmetic" functions.
if (NetMode == NM_DedicatedServer && Function->HasAllFunctionFlags(FUNC_BlueprintCosmetic))
{
    DEBUG_CALLSPACE(TEXT("GetFunctionCallspace Blueprint Cosmetic Absorbed: %s"),
*Function->GetName());
    return FunctionCallspace::Absorbed;
}
}
```

Client

- 功能描述：**在Client-owned的Actor上（PlayerController或Pawn）执行一个RPC函数，只运行在客户端上。对应的实现函数会添加_Implementation后缀。
- 元数据类型：**bool
- 引擎模块：**Network
- 作用机制：**在FunctionFlags加入FUNC_Net、FUNC_NetClient
- 常用程度：**★★★★★

在Client-owned的Actor上（PlayerController或Pawn）执行一个RPC函数，只运行在客户端上。对应的实现函数会添加_Implementation后缀。

一般用于从Server发送一个RPC到Client。和蓝图里RunOnClient的效果一样。

所谓Client-owned，参考文档：<https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>

从服务器调用的 RPC

Actor 所有权	未复制	NetMulticast	Server	Client
Client-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在 actor 的所属客户端上运行
Server-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行
Unowned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行

从客户端调用的 RPC

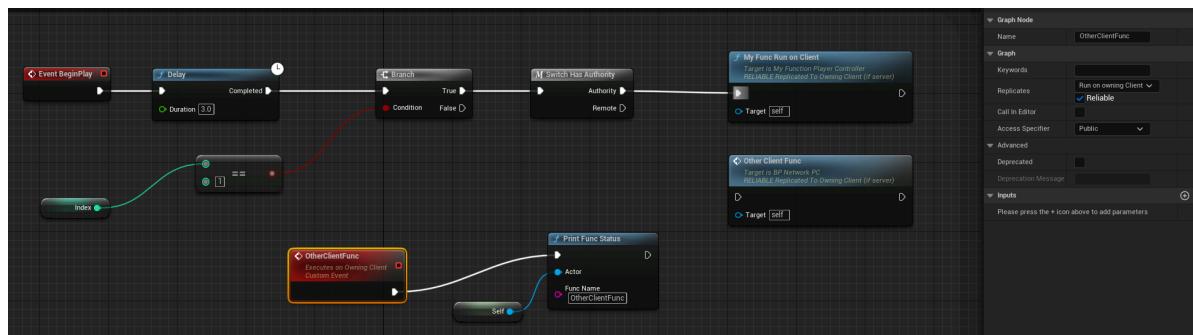
Actor 所有权	未复制	NetMulticast	Server	Client
Owned by invoking client	在执行调用的客户端上运行	在执行调用的客户端上运行	在服务器上运行	在执行调用的客户端上运行
Owned by a different client	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Server-owned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Unowned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, Client, Reliable)
    void MyFunc_RunOnClient();
};

void AMyFunction_PlayerController::MyFunc_RunOnClient_Implementation()
{
    UInsiderLibrary::PrintFuncStatus(this,
TEXT("MyFunc_RunOnClient_Implementation"));
}
```

测试蓝图：PIE模式，一个ListenServer+2Client



测试输出结果：

```
MyFunc_Client_Implementation    BP_NetworkPC_C_0    NM_Client
Local:ROLE_AutonomousProxy    Remote:ROLE_Authority
OtherClientFunc    BP_NetworkPC_C_0    NM_Client    Local:ROLE_AutonomousProxy
Remote:ROLE_Authority
```

可见，测试代码中取第2个PC，发出一个Run on Client的RPC调用，最终在Client上成功触发。C++定义的函数和蓝图中添加的自定义RunOnClient事件效果是等价的。

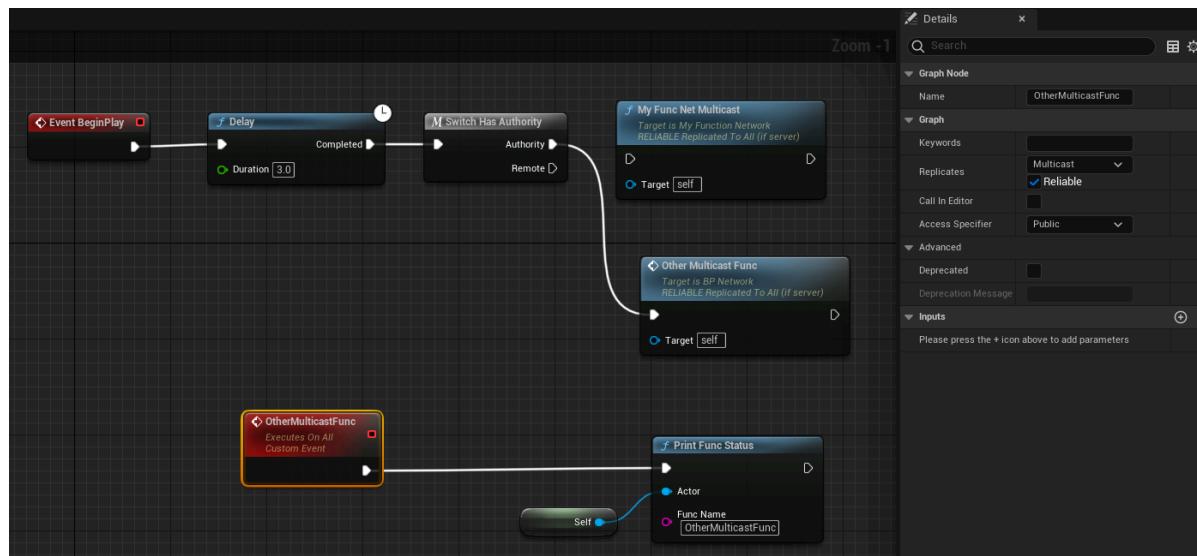
而如果这个函数在Server owned Actor上执行，则只会在运行在服务器上，不会传递到客户端。

NetMulticast

- 功能描述：** 定义一个多播RPC函数在服务器和客户端上都执行。对应的实现函数会添加`_Implementation`后缀。
- 元数据类型：** bool
- 引擎模块：** Network
- 作用机制：** 在FunctionFlags中加入FUNC_Net、FUNC_NetMulticast
- 常用程度：** ★★★★☆

定义一个多播RPC函数在服务器和客户端上都执行。对应的实现函数会添加`_Implementation`后缀。

RPC执行的规则，参考文档：<https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>



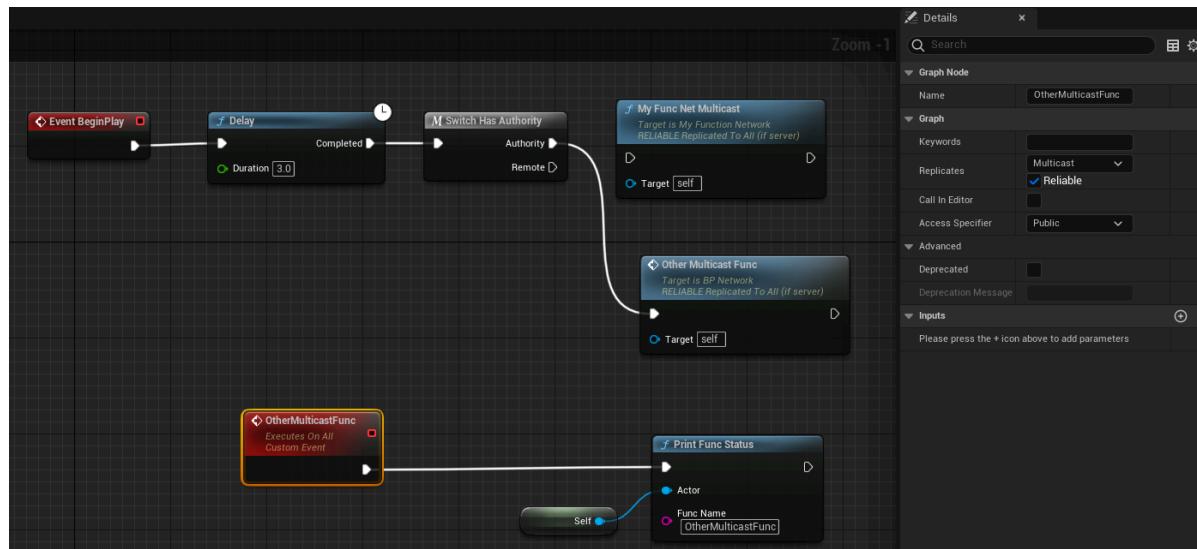
测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, NetMulticast, Reliable)
    void MyFunc_NetMulticast();

void AMyFunction_Network::MyFunc_NetMulticast_Implementation()
{
    UIInsiderLibrary::PrintFuncStatus(this,
    TEXT("MyFunc_NetMulticast_Implementation"));
}
```

测试蓝图：PIE模式，一个ListenServer+2Client



测试输出结果：

```
LogInsider: Display: 46715a00    MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_ListenServer Local:ROLE_Authority
Remote:ROLE_SimulatedProxy
LogInsider: Display: 46e65000    MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_Client     Local:ROLE_SimulatedProxy  Remote:ROLE_Authority
LogInsider: Display: 29aaaa00    MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_Client     Local:ROLE_SimulatedProxy  Remote:ROLE_Authority

LogInsider: Display: 4ff44600    OtherMulticastFunc  BP_Network_C_1
NM_ListenServer Local:ROLE_Authority  Remote:ROLE_SimulatedProxy
LogInsider: Display: 3bf89b00    OtherMulticastFunc  BP_Network_C_1  NM_Client
Local:ROLE_SimulatedProxy  Remote:ROLE_Authority
LogInsider: Display: 29d68700    OtherMulticastFunc  BP_Network_C_1  NM_Client
Local:ROLE_SimulatedProxy  Remote:ROLE_Authority
```

在一个Server Owned的Actor上，发出Multicast RPC事件调用，可以见到在3个端都得到了调用。

Reliable

- **功能描述：**指定一个RPC函数为“可靠的”，当遇见网络错误时会重发以保证到达。一般用在逻辑关键的函数上。
- **元数据类型：** bool
- **引擎模块：** Network
- **作用机制：** 在FunctionFlags加入FUNC_NetReliable
- **常用程度：** ★★★★☆

指定一个RPC函数为“可靠的”，当遇见网络错误时会重发以保证到达。一般用在逻辑关键的函数上。

具体的原理涉及到了重发信息包的逻辑。

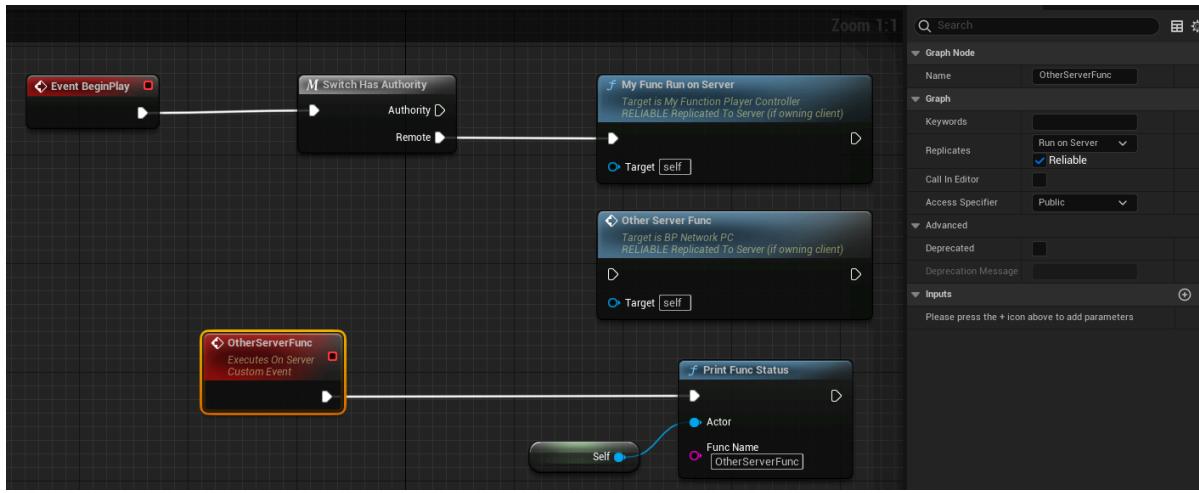
Server

- **功能描述：** 在Client-owned的Actor上（PlayerController或Pawn）执行一个RPC函数，只运行在服务器上。对应的实现函数会添加_Implementation后缀
- **元数据类型：** bool
- **引擎模块：** Network
- **作用机制：** 在FunctionFlags中加入FUNC_Net、FUNC_NetServer
- **常用程度：** ★★★★☆

在Client-owned的Actor上（PlayerController或Pawn）执行一个RPC函数，只运行在服务器上。对应的实现函数会添加_Implementation后缀。

和RunOnServer的效果一样。

所谓Client-owned，参考文档：<https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>



测试代码：

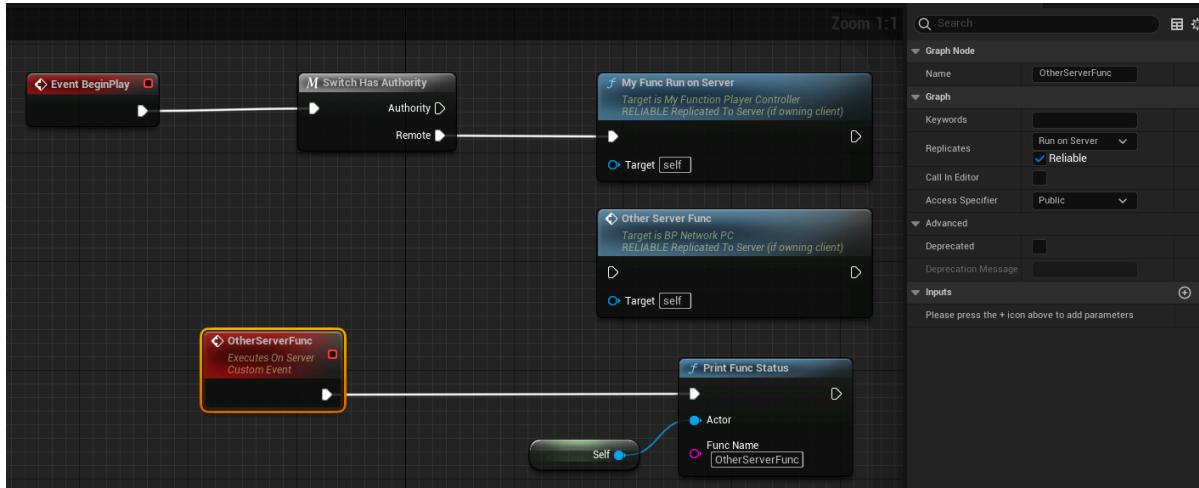
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, server, Reliable)
        void MyFunc_RunOnServer();
};

void AMyFunction_PlayerController::MyFunc_RunOnServer_Implementation()
{
    UInsiderLibrary::PrintFuncStatus(this,
TEXT("MyFunc_RunOnServer_Implementation"));
}

```

测试蓝图：PIE模式，一个ListenServer+2Client



测试输出结果：

```
LogInsider: Display: 5118b400    MyFunc_RunOnServer_Implementation
BP_NetworkPC_C_1      NM_ListenServer Local:ROLE_Authority
Remote:ROLE_AutonomousProxy
LogInsider: Display: 44ec3c00    MyFunc_RunOnServer_Implementation
BP_NetworkPC_C_2      NM_ListenServer Local:ROLE_Authority
Remote:ROLE_AutonomousProxy

LogInsider: Display: 49999000    OtherServerFunc BP_NetworkPC_C_1
NM_ListenServer Local:ROLE_Authority    Remote:ROLE_AutonomousProxy
LogInsider: Display: 4bcbd800    OtherServerFunc BP_NetworkPC_C_2
NM_ListenServer Local:ROLE_Authority    Remote:ROLE_AutonomousProxy
```

可见，测试代码中取第2个PC，发出一个Run on Server的RPC调用，最终在Server上成功触发。C++定义的函数和蓝图中添加的自定义RunOnServer事件效果是等价的。

而如果这个函数在Server owned Actor上执行，则只会在运行在服务器上，不会传递到客户端。

ServiceRequest

- **功能描述：**此函数为RPC（远程过程调用）服务请求。rpc服务请求
- **元数据类型：**bool
- **引擎模块：**Network
- **作用机制：**在Meta中加入CustomThunk，在FunctionFlags加入FUNC_Net、FUNC_Event、FUNC_NetReliable、FUNC_NetRequest

在源码里都没看到使用，只搜到

```
UCLASS()
class
UTestReplicationStateDescriptor_TestFunctionWithNotReplicatedNonPODParameters :
public Uobject
{
GENERATED_BODY()

protected:
// Currently some features such as not replicating all parameters isn't
allowed on regular RPCs
UFUNCTION(ServiceRequest(Iris))
void FunctionWithNotReplicatedNonPODParameters(int Param0, bool Param1, int
Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
void FunctionWithNotReplicatedNonPODParameters_Implementation(int Param0,
bool Param1, int Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
};
```

UDN回答：

Alex: Those specifiers were added quite a while ago as a way to mark functions as RPC requests/responses to and from a backend service, the name of which would be given as part of the specifier: UFUNCTION(ServiceRequest()). However, the feature was never fully implemented, and since then the specifiers have only been used internally (and even then, I don't believe "ServiceResponse" is used at all anymore). This is why there isn't any public documentation or examples available, as they're not formally supported in the engine. You can check out ServiceRequestSpecifier and ServiceResponseSpecifier in UhtFunctionSpecifiers.cs to see how UHT handles these specifiers.

Mi: 这两个标记是我们用来自由扩展和自己的服务器通信的（例如http request），譬如可以提供自己的NetDriver处理特定标记的ServiceRequest的RPC，自己序列化对应参数发给自己的服务。

“意思是如果使用引擎的默认实现的话，使用这两个标记是无效的吗？我尝试在服务器或者客户端发起对ServiceRequest标记的ufunction的调用，结果都是会打印错误日志”

是的，默认的UE client和DS通信的NetDriver的RPC不需要这两个关键字，用了之后会找不到相应处理的NetDriver的实现。

在Server Owned Actor上调用会出错： LogNet: Warning: UNetDriver::ProcessRemoteFunction: No owning connection for actor BP_Network_C_1. Function MyFunc_ServiceRequest will not be processed.

在PC上Server调用也会：

```
LogRep: Error: Rejected RPC function due to access rights. Object: BP_NetworkPC_C
/Game/UEDPIE_0_StartMap.StartMap:PersistentLevel.BP_NetworkPC_C_1, Function:
MyFunc_ServiceRequest
LogNet: Error: UActorChannel::ProcessBunch: Replicator.ReceivedBunch failed. Closing
connection. RepObj: BP_NetworkPC_C
/Game/UEDPIE_0_StartMap.StartMap:PersistentLevel.BP_NetworkPC_C_1, Channel: 3
```

ServiceResponse

- **功能描述：**此函数为RPC服务响应。rpc服务回复
- **元数据类型：**bool
- **引擎模块：**Network
- **作用机制：**在FunctionFlags加入FUNC_Net、FUNC_Event、FUNC_NetReliable、FUNC_NetResponse

在源码里一个也没看到使用。

Unreliable

- **功能描述：**指定一个RPC函数为“不可靠的”，当遇见网络错误时就会被丢弃。一般用在传播效果表现的函数上，就算漏掉也没有关系。
- **元数据类型：**bool
- **引擎模块：**Network
- **常用程度：**★★★★★

指定一个RPC函数为“不可靠的”，当遇见网络错误时就会被丢弃。一般用在传播效果表现的函数上，就算漏掉也没有关系。

WithValidation

- **功能描述：** 指定一个RPC函数在执行前需要验证，只有验证通过才可以执行。
- **元数据类型：** bool
- **引擎模块：** Network
- **作用机制：** 在FunctionFlags中加入FUNC_NetValidate
- **常用程度：** ★★★★☆

指定一个RPC函数在执行前需要验证，只有验证通过才可以执行。

WithValidation实际上可以用于Client, Server, NetMulticast的RPC函数，但一般来说还是用在Server的最多，因为一般是Server的数据最权威可以进行数据合法性校验。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, Client, Reliable, withValidation)
    void MyFunc2_RunOnClient();

    UFUNCTION(BlueprintCallable, Server, Reliable, withValidation)
    void MyFunc2_RunOnServer();
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()
    UFUNCTION(BlueprintCallable, NetMulticast, Reliable, withValidation)
    void MyFunc2_NetMulticast();
};

void AMyFunction_PlayerController::MyFunc2_RunOnServer_Implementation()
{
    UInsiderLibrary::PrintFuncStatus(this,
    TEXT("MyFunc2_RunOnServer_Implementation"));
}

bool AMyFunction_PlayerController::MyFunc2_RunOnServer_Validate()
{
    UInsiderLibrary::PrintFuncStatus(this, TEXT("MyFunc2_RunOnServer_Validate"));
    return true;
}

bool AMyFunction_Network::MyFunc2_NetMulticast_Validate()
{
```

```

    UIInsiderLibrary::PrintFuncStatus(this,
TEXT("MyFunc2_NetMulticast_validate"));
    return true;
}

```

测试结果：

```

RunOnClient:
LogInsider: Display: 815f7800 MyFunc2_RunOnClient_Validate BP_NetworkPC_C_0
    NM_Client Local:ROLE_AutonomousProxy Remote:ROLE_Authority
LogInsider: Display: 815f7800 MyFunc2_RunOnClient_Implementation
    BP_NetworkPC_C_0 NM_Client Local:ROLE_AutonomousProxy
    Remote:ROLE_Authority

RunOnServer:
LogInsider: Display: 7fd11800 MyFunc2_RunOnServer_Validate BP_NetworkPC_C_1
    NM_ListenServer Local:ROLE_Authority Remote:ROLE_AutonomousProxy
LogInsider: Display: 7fd11800 MyFunc2_RunOnServer_Implementation
    BP_NetworkPC_C_1 NM_ListenServer Local:ROLE_Authority
    Remote:ROLE_AutonomousProxy

Multicast: ServerOwned
LogInsider: Display: 947e6400 MyFunc2_NetMulticast_Validate BP_Network_C_1
    NM_ListenServer Local:ROLE_Authority Remote:ROLE_SimulatedProxy
LogInsider: Display: 947e6400 MyFunc2_NetMulticast_Implementation
    BP_Network_C_1 NM_ListenServer Local:ROLE_Authority
    Remote:ROLE_SimulatedProxy
LogInsider: Display: 8795eb00 MyFunc2_NetMulticast_Validate BP_Network_C_1
    NM_Client Local:ROLE_SimulatedProxy Remote:ROLE_Authority
LogInsider: Display: 8795eb00 MyFunc2_NetMulticast_Implementation
    BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy Remote:ROLE_Authority
LogInsider: Display: 8f6a3700 MyFunc2_NetMulticast_Validate BP_Network_C_1
    NM_Client Local:ROLE_SimulatedProxy Remote:ROLE_Authority
LogInsider: Display: 8f6a3700 MyFunc2_NetMulticast_Implementation
    BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy Remote:ROLE_Authority

```

原理：

如果加上WithValidation标记，在UHT生成代码的时候就会：

```

DEFINE_FUNCTION(AMyFunction_PlayerController::execMyFunc2_RunOnServer)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_RunOnServer_Validate())
    {
        RPC_ValidateFailed(TEXT("MyFunc2_RunOnServer_Validate"));
        return;
    }
    P_THIS->MyFunc2_RunOnServer_Implementation();
    P_NATIVE_END;
}

DEFINE_FUNCTION(AMyFunction_PlayerController::execMyFunc2_RunOnClient)

```

```

{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_RunOnClient_validate())
    {
        RPC_ValidateFailed(TEXT("MyFunc2_RunOnClient_validate"));
        return;
    }
    P_THIS->MyFunc2_RunOnClient_Implementation();
    P_NATIVE_END;
}

DEFINE_FUNCTION(AMyFunction_Network::execMyFunc2_NetMulticast)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_NetMulticast_validate())
    {
        RPC_ValidateFailed(TEXT("MyFunc2_NetMulticast_validate"));
        return;
    }
    P_THIS->MyFunc2_NetMulticast_Implementation();
    P_NATIVE_END;
}

```

BlueprintInternalUseOnly

- 功能描述:** 指示不应向最终用户公开此函数。蓝图内部调用，不暴露给用户。
- 元数据类型:** bool
- 引擎模块:** Blueprint, UHT
- 作用机制:** 在Meta中加入BlueprintInternalUseOnly、BlueprintType
- 常用程度:** ★★★

指示不应向最终用户公开此函数。蓝图内部调用，不暴露给用户。

等价于meta里加上BlueprintInternalUseOnly = true。默认情况下，BlueprintCallable/Pure的函数会生成UK2Node_CallFunction来调用。但BlueprintInternalUseOnly阻止了这一部分。

典型的用处有二：

一是在蓝图中隐藏该函数，但因为该函数依然有UFUNCTION，因此可以通过名字来反射调用该函数。虽然该用法比较稀少，但也算是一种用处。

二是引擎在别的地方会为该函数声明去按照特定的规则创建另一个蓝图函数节点，因此要隐藏掉按照默认规则创建的这个。这种用法就是引擎源码里大量在使用的用法。

示例代码1：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Internal :public AActor
{
public:
    GENERATED_BODY()
public:

```

```

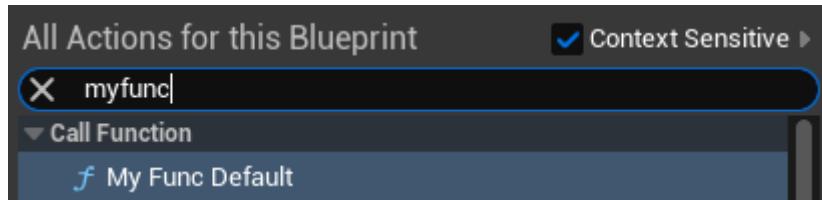
//(BlueprintInternalUseOnly = true, BlueprintType = true, ModuleRelativePath
= Function/MyFunction_Internal.h)
//FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable, BlueprintInternalUseOnly)
void MyFunc_InternalOnly()

//FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable)
void MyFunc_Default()

};


```

在蓝图中只有MyFunc_Default是可以调用的。因此可以理解为这个函数依然暴露到蓝图，但是却又被隐藏起来了。不能让用户自己直接调用，但是可以在代码里通过查找函数名之类的间接可以调用到。



在源码里找到一个示例，因此这个GetLevelScriptActor函数，可以不在蓝图中被调用，但是有可以通过名字查找到。方便生成一个UFunction以被注入到别的地方作为callback

```

ULevelStreaming:
UFUNCTION(BlueprintPure, meta = (BlueprintInternalUseOnly = "true"))
ENGINE_API ALevelScriptActor* GetLevelScriptActor();

然后发现:
GetLevelScriptActorNode->SetFromFunction(ULevelStreaming::StaticClass()-
>FindFunctionByName(GET_FUNCTION_NAME_CHECKED(ULevelStreaming,
GetLevelScriptActor)));

```

示例代码2：

实现代码就不贴了，可以自己去项目里查看。

```

UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy =
MyAsyncObject, HasDedicatedAsyncNode))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Loop;

    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Complete;

    UFUNCTION(BlueprintCallable, meta = (BlueprintInternalUseOnly = "true",
worldContext = "WorldContextObject"), category = "Flow Control")

```

```

static UMyFunction_Async* DelayLoop(const UObject* WorldContextObject, const
float DelayInSeconds, const int Iterations);

virtual void Activate() override;

UFUNCTION()
static void Test();

private:
const UObject* WorldContextObject = nullptr;
float MyDelay = 0.f;
int MyIterations = 0;
bool Active = false;

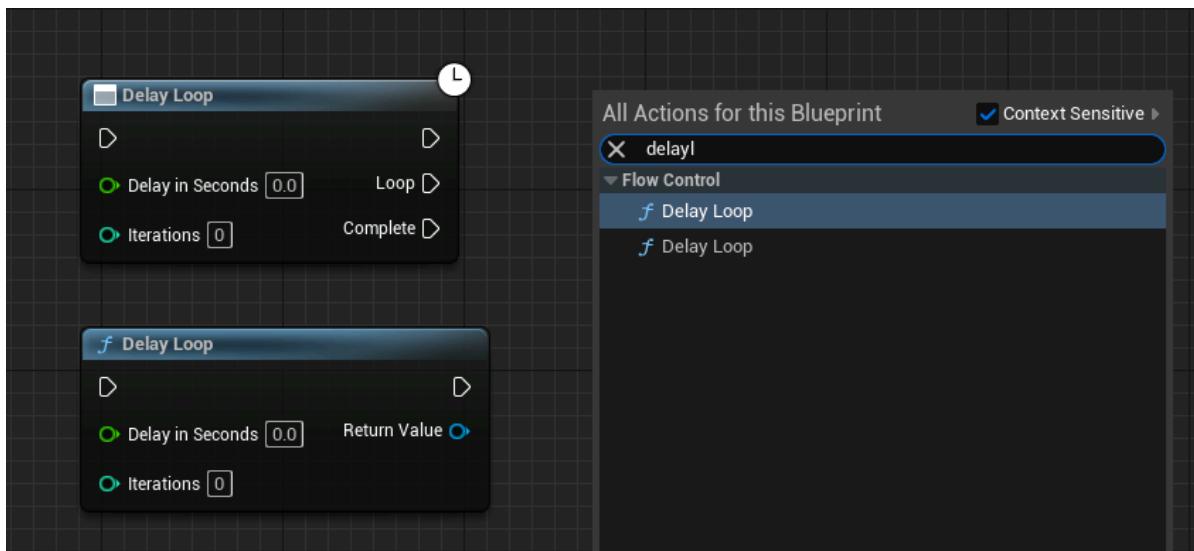
UFUNCTION()
void ExecuteLoop();

UFUNCTION()
void ExecuteComplete();
};


```

示例效果：

假如注释掉上述源码的BlueprintInternalUseOnly，会发现在蓝图里可以有两个DelayLoop。上面的一个是按UBlueprintAsyncActionBase规则生成的，第二个是按普通的蓝图函数规则生成的。明显这种情况下我们并不想同时出现两个来给用户造成困惑。因此要加上BlueprintInternalUseOnly来阻止生成默认的蓝图节点。



原理：

关于UBlueprintAsyncActionBase的使用，UK2Node_BaseAsyncTask的函数实现里体现了书写继承于UBlueprintAsyncActionBase的规则，简单来说就是通过static函数来当作Factory function，然后分析这个Proxy类的Delegate property来当作Pin。

如果不加BlueprintInternalUseOnly = "true"，则会生成两个函数。下面那个是普通static函数的生成。上面那个是分析UBlueprintAsyncActionBase生成的函数。

其中识别UBlueprintAsyncActionBase里面static函数作为FactoryFunction的流程是， BlueprintActionDatabaseImpl::GetNodeSpecificActions会触发 UK2Node_AsyncAction::GetMenuActions，从而ActionRegistrar.RegisterClassFactoryActions，内部再判断RegisterClassFactoryActions_Utils::IsFactoryMethod(Function, UBlueprintAsyncActionBase) 会通过（判断是static函数，并且返回类型是UBlueprintAsyncActionBase的子类对象），继而继续通过回调UBlueprintFunctionNodeSpawner::Create(FactoryFunc);创建一个工厂方法的nodeSpawner。

因此总结，此时的BlueprintInternalUseOnly 就是隐藏掉默认生成的那个。

CustomThunk

- 功能描述：**指定UHT不为该函数生成蓝图调用的辅助函数，而需要用户自定义编写。
- 元数据类型：** bool
- 引擎模块：** UHT
- 作用机制：** 在Meta中加入CustomThunk
- 常用程度：** ★★★

指定UHT不为该函数生成蓝图调用的辅助函数，而需要用户自定义编写。

这里Thunk的意思就是类似execFoo的函数，需要用户自己定义。

CustomThunk一般是用于配合函数参数不定的情况，如各种通配符，或者需要自己更细致的自定义的逻辑处理。

测试代码：

```
UFUNCTION(BlueprintPure, CustomThunk)
static int32 MyFunc_CustomDivide(int32 A, int32 B = 1);

DECLARE_FUNCTION(execMyFunc_CustomDivide);

int32 UMyFunction_Custom::MyFunc_CustomDivide(int32 A, int32 B /*= 1*/)
{
    return 1;
}

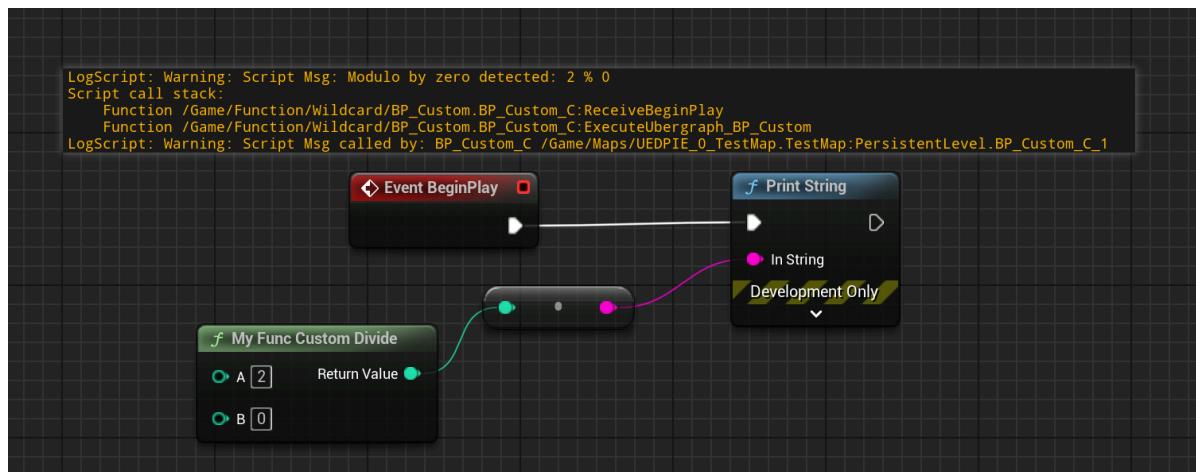
DEFINE_FUNCTION(UMyFunction_Custom::execMyFunc_CustomDivide)
{
    P_GET_PROPERTY(FIntProperty, A);
    P_GET_PROPERTY(FIntProperty, B);

    P_FINISH;

    if (B == 0)
    {
        FFrame::KismetExecutionMessage(*FString::Printf(TEXT("Modulo by zero
detected: %d %0\n%s")), A, *Stack.GetStackTrace()), ELogVerbosity::Warning);
        *(int32*)RESULT_PARAM = 0;
        return;
    }

    *(int32*)RESULT_PARAM = A/B;
}
```

蓝图效果：



可以看到，即使是用除以0，可以自定义报错信息。

最重要的是如果观察.gen.cpp，可以对比发现内部不再生成execFoo的函数。

FieldNotify

- 功能描述：**为该函数创建一个FieldNotify的绑定点。
- 元数据类型：**bool
- 引擎模块：**UHT
- 限制类型：**ViewModel里的函数
- 常用程度：**★★★

为该函数创建一个FieldNotify的绑定点。

需要注意的是，如果是Get函数则其返回值改变的时候，需要在别的触发改变的地方手动广播事件。正如下面的代码UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);所做的。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyViewModel : public UMVVMViewModelBase
{
    GENERATED_BODY()

protected:
    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetHP)
    float HP = 1.f;

    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetMaxHP)
    float MaxHP = 100.f;

public:
    float GetHP() const { return HP; }
    UFUNCTION(BlueprintSetter)
    void SetHP(float val)
    {
        if (UE_MVVM_SET_PROPERTY_VALUE(HP, val))
        {
    }
```

```

        UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
    }

}

float GetMaxHP() const { return MaxHP; }
UFUNCTION(BlueprintSetter)
void SetMaxHP(float val)
{
    if (UE_MVVM_SET_PROPERTY_VALUE(MaxHP, val))
    {
        UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
    }
}

//You need to manually notify that GetHealthPercent changed when
CurrentHealth or MaxHealth changed.
UFUNCTION(BlueprintPure, FieldNotify)
float GetHPPPercent() const
{
    return (MaxHP != 0.f) ? HP / MaxHP : 0.f;
}
;

```

测试效果：

可见GetHPPPercent有生成一个FIELD。

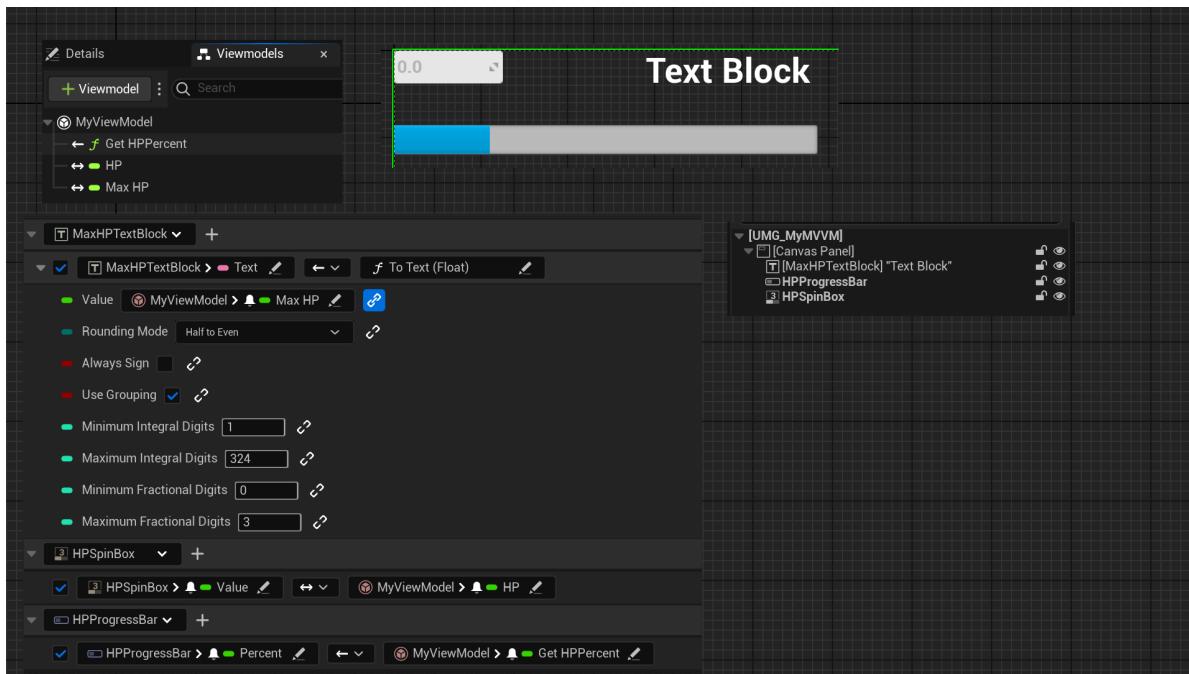
```

//MyViewModel.generated.h
#define
FID_Gitworkspace_Hello_Source_Insider_Property_MVVM_MyViewModel_h_12_FIELDNOTIFY \
\
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BEGIN(INSIDER_API ) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(HP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(HP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END() \
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_END(); \
//MyViewModel.gen.cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_BEGIN(UMyViewModel)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_END(UMyViewModel);

```

蓝图效果：

进度条可以绑定到GetHPPPercent。



Variadic

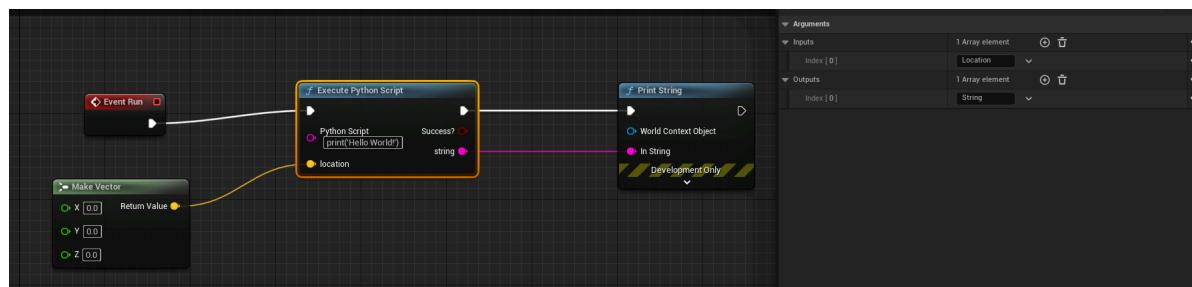
- 功能描述:** 标识一个函数可以接受任意类型的多个参数 (包括input/output).
- 元数据类型:** bool
- 引擎模块:** Blueprint, UHT
- 作用机制:** 在Meta中加入Variadic
- 常用程度:** ★★★

标识一个函数可以接受任意类型的多个参数 (包括input/output).

在源码中搜索应用：然后配合UK2Node_ExecutePythonScript

```
UFUNCTION(BlueprintCallable, CustomThunk, Category = "Python|Execution", meta=(Variadic, BlueprintInternalUseOnly="true"))
    static bool ExecutePythonScript(UPARAM(meta=(MultiLine=True)) const FString&
PythonScript, const TArray<FString>& PythonInputs, const TArray<FString>&
PythonOutputs);
    DECLARE_FUNCTION(execExecutePythonScript);
```

蓝图的效果：



示例代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Variadic : public UBlueprintFunctionLibrary
{
```

```

public:
    GENERATED_BODY()
public:
/*
    [PrintVariadicFields    Function->Struct->Field->Object
 /script/Insider.MyFunction_Variadic:PrintVariadicFields]
    (BlueprintInternalUseOnly = true, BlueprintType = true, CustomThunk =
 true, ModuleRelativePath = Function/Variadic/MyFunction_Variadic.h, variadic = )
*/
    UFUNCTION(BlueprintCallable, CustomThunk, BlueprintInternalUseOnly, meta =
(Variadic))
    static FString PrintVariadicFields(const TArray<FString>& Inputs, const
TArray<FString>& Outputs);
    DECLARE_FUNCTION(execPrintVariadicFields);
};

FString UMyFunction_Variadic::PrintVariadicFields(const TArray<FString>& Inputs,
const TArray<FString>& Outputs)
{
    check(0);
    return TEXT("");
}

DEFINE_FUNCTION(UMyFunction_Variadic::execPrintVariadicFields)
{
    FString str;

    P_GET_TARRAY_REF(FString, Inputs);
    P_GET_TARRAY_REF(FString, Outputs);

    for (const FString& PythonInput : Inputs)
    {
        Stack.MostRecentPropertyAddress = nullptr;
        Stack.MostRecentProperty = nullptr;
        Stack.StepCompiledIn<FProperty>(nullptr);
        check(Stack.MostRecentProperty && Stack.MostRecentPropertyAddress);

        FProperty* p = CastField<FProperty>(Stack.MostRecentProperty);

        FString PropertyValueString;
        const void* PropertyValuePtr = p->ContainerPtrToValuePtr<const void*>
(Stack.MostRecentPropertyContainer);

        p->ExportTextItem_Direct(PropertyValueString, PropertyValuePtr, nullptr,
nullptr, PPF_None);

        str += FString::Printf(TEXT("%s:%s\n"), *p->GetFName().ToString(),
*PropertyValueString);

    }
    P_FINISH;

    *(FString*)RESULT_PARAM = str;
}

```

示例效果：



打印：

CallFunc_MakeVector_ReturnValue:(X=1.000000,Y=2.000000,Z=3.000000)

CallFunc_MakeLiteralDouble_ReturnValue:456.000000

原理：

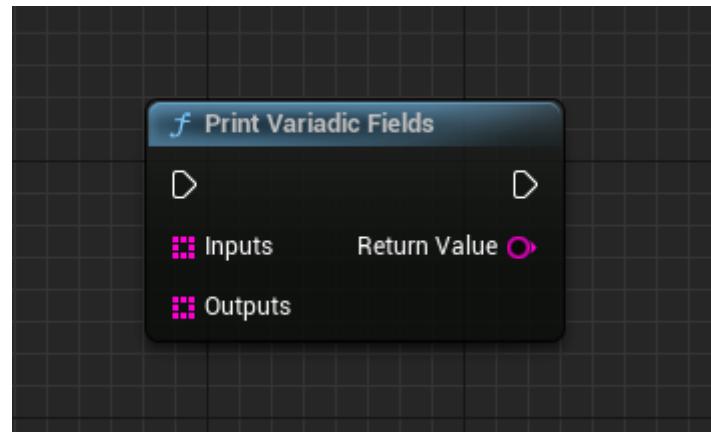
普通的CustomThunk函数还有一些限制，参数名字和个数是在UFunction里写死的，不能支持动态的个数。

目前，想使用**Variadic**功能，需要自定义蓝图节点用C++来为**K2Node_CallFunction**添加引脚。

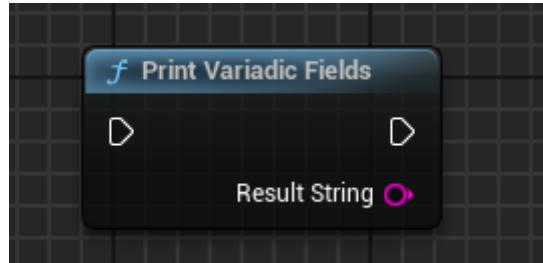
想必是想要开发来同时实现**K2Node**以及对应的**CustomThunk+Variadic**方法，来保证使用上的安全性。

BlueprintInternalUseOnly也要加上，否则会自动生成普通的蓝图函数，达不到variadic的效果。

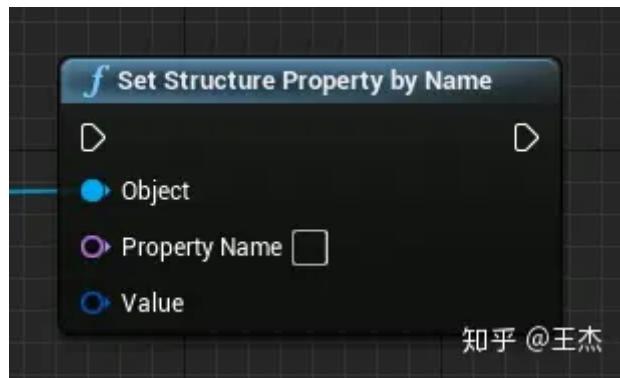
以下是不加BlueprintInternalUseOnly自动生成的版本：



实际应该是：然后再手动添加参数。



和Wildcard的区别是，Wildcard的参数是任意类型的，但个数是固定好的。



官方添加的和Python交互的功能 [Added a Blueprint node for calling Python with args](#)

官方的提交:

<https://github.com/EpicGames/UnrealEngine/commit/61d0f65e1cded45ed94f0422eb931f44688e972>

注释:

Implemented variadic function support for Blueprints

Variadic functions are required to be a CustomThunk marked with the "Variadic" meta-data. They can then be used from a custom Blueprint node to accept arbitrary arguments at the end of their parameter list (any extra pins added to the node that aren't part of the main function definition will become the variadic payload).

Variadic arguments aren't type checked, so you need other function input to tell you how many to expect, and for a nativized function, also what type of arguments you're dealing with.

#jira UE-84932

#rb Dan.OConnor

[CL 10421401 by Jamie Dale in Dev-Editor branch]`

Const

- **功能描述:** 指定该函数参数不可更改
- **元数据类型:** bool
- **引擎模块:** Blueprint, Parameter
- **作用机制:** 在PropertyFlags中加入CPF_ConstParm, 在Meta中加入NativeConst
- **常用程度:** ★

指定该函数参数不可更改。

如果在C++代码的参数上直接加const, 则会自动的被UHT识别并添加CPF_ConstParm 标志, 以及 NativeConst元数据。但也可以手动加上UPARAM(const)来强制UHT添加CPF_ConstParm, 效果见下面蓝图中的Out节点, 把输出参数变成了输入参数。

虽然不知道什么情况下需要手动添加, 因此在源码中没有找到实际的用例。能想到的用处是在蓝图层面使它变成const输入参数, 但是在C++层面依然是可变的引用参数, 方便在C++里调用一些非const的方法。

测试代码:

```

//PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_ZeroConstructor | 
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_ConstInt(UPARAM(const) int value);

    //PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm | 
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_ConstIntOut(UPARAM(const) int& value);

    //NativeConst = 
    //PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm | 
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_ConstIntRef(UPARAM(const) const int& value);

    //PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_NoConstInt(int value);

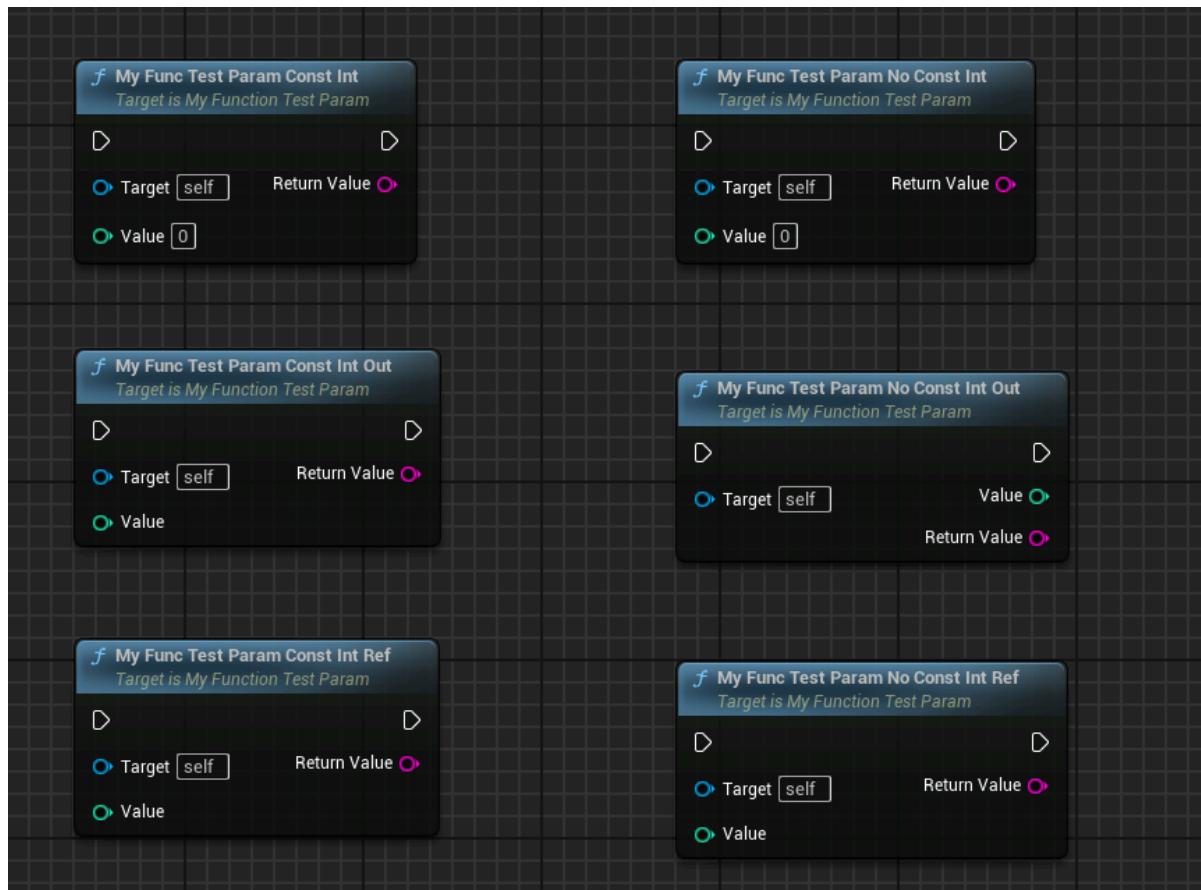
    //PropertyFlags:    CPF_Parm | CPF_OutParm | CPF_ZeroConstructor | 
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_NoConstIntOut(int& value);

    //NativeConst = 
    //PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm | 
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UFUNCTION(BlueprintCallable)
    FString MyFuncTestParam_NoConstIntRef(const int& value);

```

蓝图节点:

MyFuncTestParam_ConstIntOut的输出Value变成了输入的Value，因为不能改变。



原理代码：

在代码中实际出现const，就会增加CPF_ConstParam Flag。

```
\Engine\Source\Programs\Shared\EpicGames.UHT\Parsers\UhtPropertyParser.cs 1030

if (propertySettings.PropertyCategory != UhtPropertyCategory.Member &&
!isTemplateArgument)
{
    // const before the variable type support (only for params)
    if (tokenReader.TryOptional("const"))
    {
        propertySettings.PropertyFlags |= EPropertyFlags.ConstParm;
        propertySettings.MetaData.Add(UhtNames.NativeConst, "");
    }
}
```

DisplayName

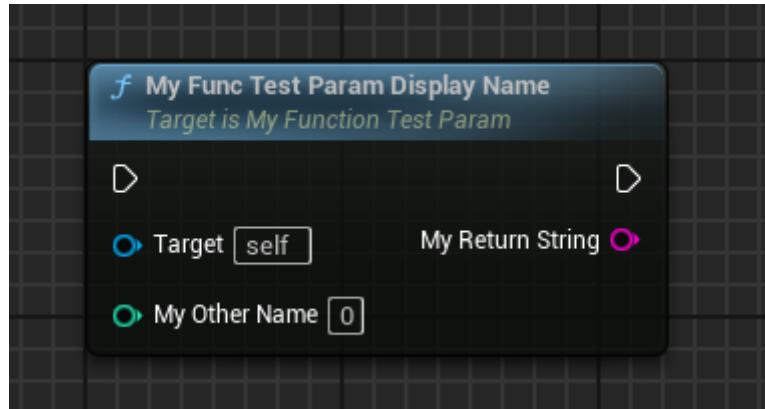
- 功能描述：**更改函数参数在蓝图节点上的显示名字
- 元数据类型：** string="abc"
- 引擎模块：** Blueprint, Parameter
- 作用机制：** 在Meta中加入DisplayName
- 常用程度：** ★★★★☆

注意：UPARAM也可以用在返回值上，默认值是ReturnValue。

测试代码：

```
//(DisplayName = My Other Name)
UFUNCTION(BlueprintCallable)
UPARAM(DisplayName = "My Return String") FString
MyFuncTestParam_DisplayName(UPARAM(DisplayName = "My Other Name") int value);
```

蓝图节点：



ref

- **功能描述：**使得函数的参数变成引用类型
- **元数据类型：**bool
- **引擎模块：**Blueprint, Parameter
- **作用机制：**在PropertyFlags中加入CPF_ReferenceParm
- **常用程度：**★★★★★

普通参数和引用参数的区别是，在获取参数的时候，Ref类型会直接获得实参的引用，而不是拷贝。这样就可以避免拷贝，保存修改。

单纯的&参数是会被解析成输出返回参数，因此要用ref再继续标明。

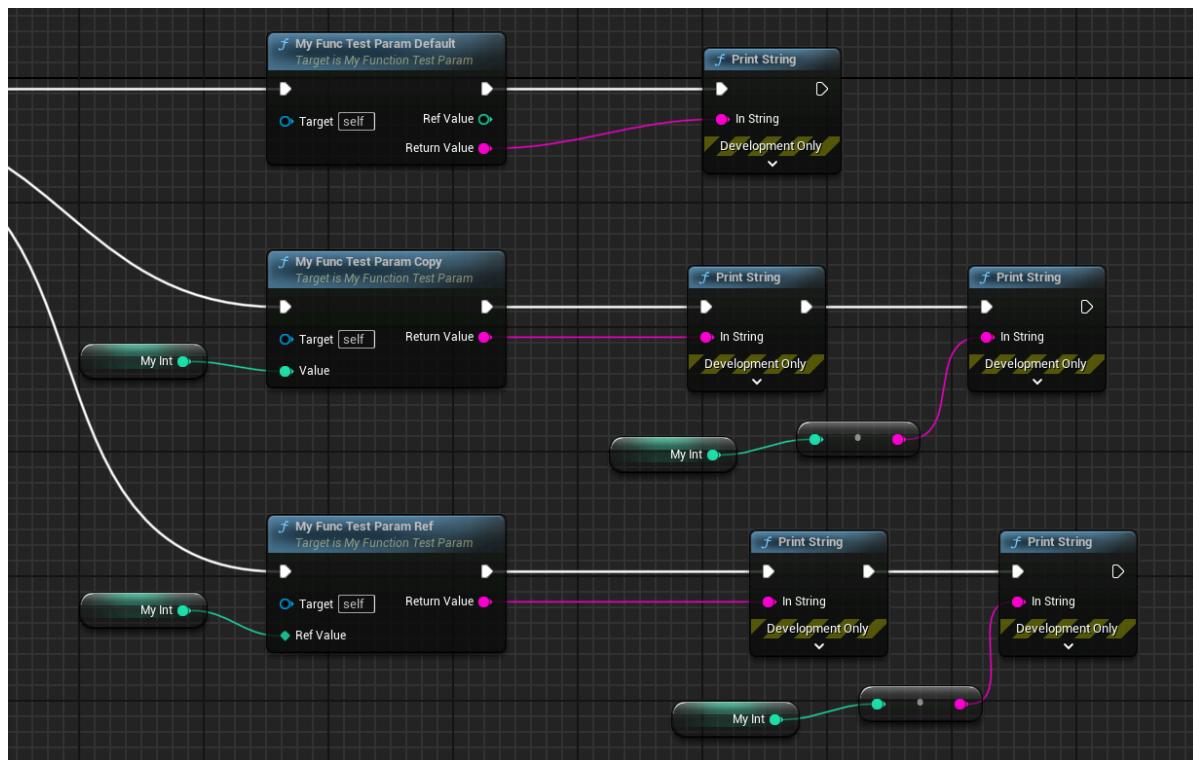
测试代码：

```
//PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPublic
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_Default(int& refValue);

//PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |
CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_Ref(UPARAM(ref) int& refValue);

UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_Copy(int value);
```

蓝图的代码：



原理：

ref参数在UHT生成时会用P_GET_PROPERTY_REF来获得

```
#define P_GET_PROPERTY(PropertyType, ParamName)
\
    PropertyType::TCppType ParamName = PropertyType::GetDefaultValue();
\
    Stack.StepCompiledIn<PropertyType>(&ParamName);

#define P_GET_PROPERTY_REF(PropertyType, ParamName)
\
    PropertyType::TCppType ParamName##Temp =
PropertyType::GetDefaultValue(); \
    PropertyType::TCppType& ParamName = Stack.StepCompiledInRef<PropertyType,
PropertyType::TCppType>(&ParamName##Temp);
```

Required

- **功能描述:** 指定函数的参数节点必须连接提供一个值
 - **元数据类型:** bool
 - **引擎模块:** Blueprint, Parameter
 - **作用机制:** 在PropertyFlags中加入CPF_RequiredParm
 - **常用程度:** ★★

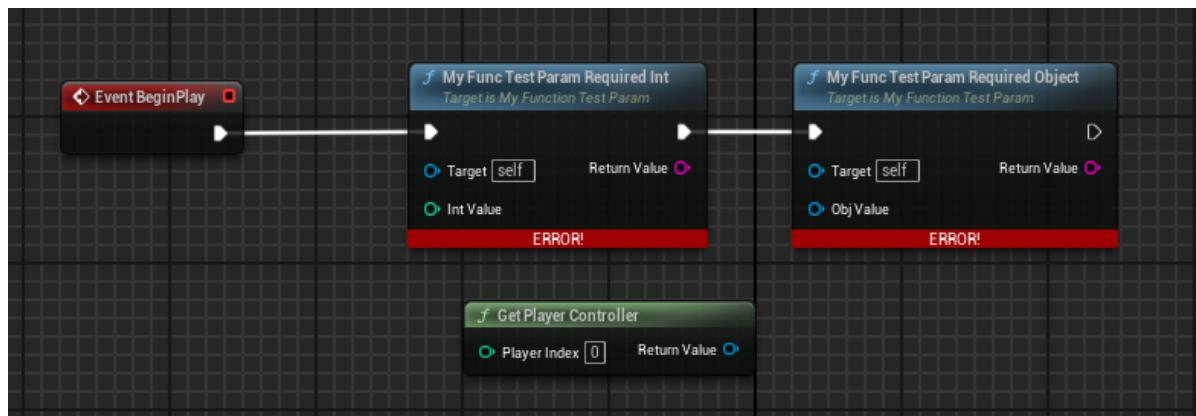
指定函数的参数节点必须连接一个变量来提供一个值。

如果参数上有提供默认值，该标志依然会忽略默认值，认为还是没提供值。还是要连接变量。

测试代码：

```
//PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_RequiredParm |  
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_RequiredObject(UPARAM(Required) UObject* objValue);  
  
//(CPP_Default_intValue = 123, ModuleRelativePath =  
Function/Param/MyFunction_TestParam.h)  
//PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_RequiredParm |  
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_RequiredInt(UPARAM(Required) int intValue=123);
```

蓝图节点：



如果不连一个节点，编译时会报错：

Pin Int Value must be linked to another node (in My Func Test Param Required Int)
Pin Obj Value must be linked to another node (in My Func Test Param Required Object)

原理：

根据这个标记来判断。

```
const bool bIsRequiredParam = Param->HasAnyPropertyFlags(CPF_RequiredParm);  
// Don't let the user edit the default value if the parameter is required to  
be explicit.  
Pin->bDefaultValueIsIgnored |= bIsRequiredParam;
```

NotReplicated

- **引擎模块：** Blueprint, Network, Parameter
- **作用机制：** 在PropertyFlags中加入CPF_RepSkip

参照UFUNCTION的ServiceRequest，该标识符弃用。

“Only parameters in service request functions can be marked NotReplicated”

```

if (context.PropertySettings.PropertyCategory ==
UhtPropertyCategory.ReplicatedParameter)
{
    context.PropertySettings.PropertyCategory =
UhtPropertyCategory.RegularParameter;
    context.PropertySettings.PropertyFlags |= EPropertyFlags.RepSkip;
}
else
{
    context.MessageSite.LogError("Only parameters in service request
functions can be marked NotReplicated");
}

```

源码里只知道

```

// Currently some features such as not replicating all parameters isn't allowed
on regular RPCs
UFUNCTION(ServiceRequest(Iris))
void FunctionWithNotReplicatedNonPODParameters(int Param0, bool Param1, int
Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
void FunctionWithNotReplicatedNonPODParameters_Implementation(int Param0, bool
Param1, int Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);

```

AssetRegistrySearchable

- 功能描述:** 标记该属性可以作为AssetRegistry的Tag和Value值来进行资产的过滤搜索
- 元数据类型:** bool
- 引擎模块:** Asset
- 作用机制:** 在PropertyFlags中加入CPF_AssetRegistrySearchable，在Meta中加入RequiredAssetDataTags、DisallowedAssetDataTags
- 常用程度:** ★★★

不能用在结构属性上。

子类也可以重载GetAssetRegistryTags以提供自定义的Tag。

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_AssetRegistrySearchable :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, AssetRegistrySearchable, Category
= DataRegistry)
        FString MyIdForSearch;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)

```

```

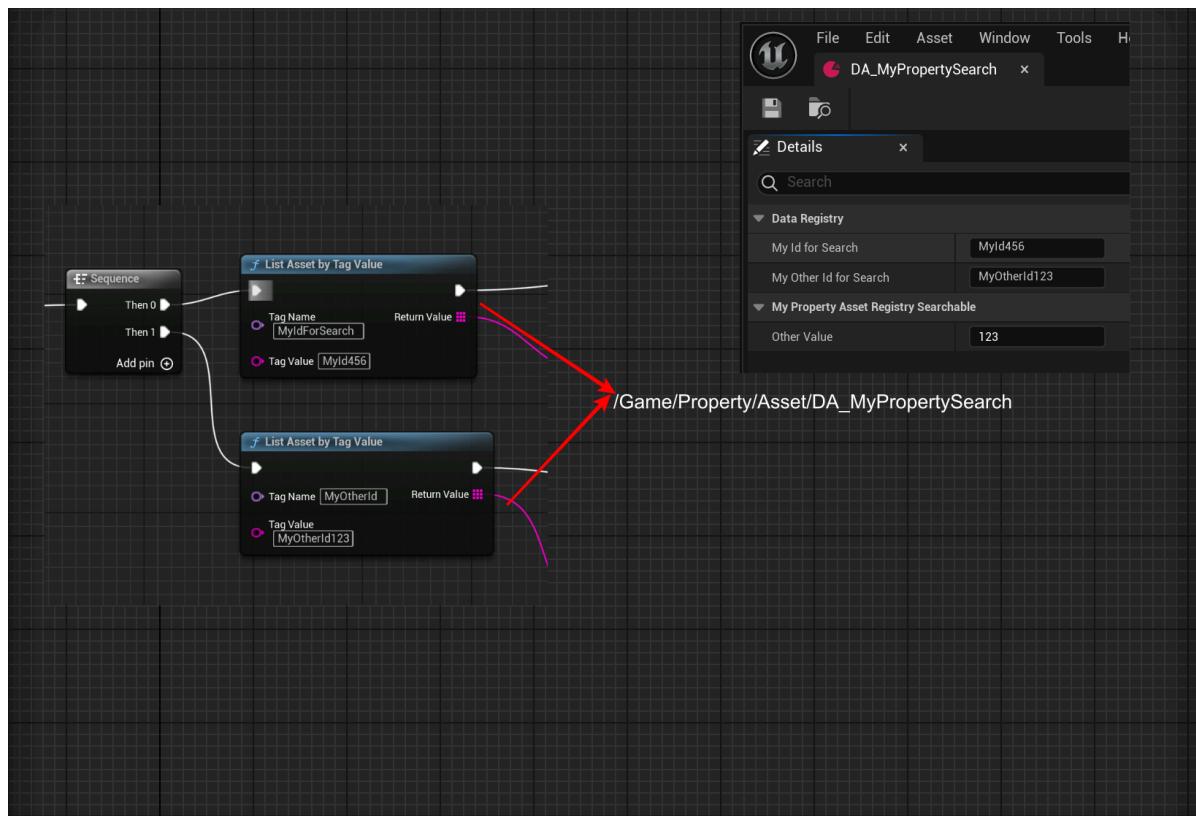
int32 OtherValue = 123;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DataRegistry)
FString MyOtherIdForSearch;

public:
    virtual void GetAssetRegistryTags(FAssetRegistryTagsContext Context) const
override
{
    //called on CDO and instances
    Super::GetAssetRegistryTags(Context);
    Context.AddTag(FAssetRegistryTag(TEXT("MyOtherId")), MyOtherIdForSearch,
UObject::FAssetRegistryTag::TT_Alphabetical);
}
};


```

测试结果：

在EditorUtilityWidget中测试，可见ListAssetByTagValue都可以搜索找到该Asset。



测试的蓝图代码，也可用IAssetRegistry::Get()->GetAssetsByTagValues(tagValues,outAssets);来进行搜索，不过要注意搜索的时机要在AssetRegistry加载之后，AssetRegistry如果是Runtime要记得序列化到磁盘

```

//DefaultEngine.ini
[AssetRegistry]
bSerializeAssetRegistry=true

```

原理：

可查看GetAssetRegistryTags的函数的实现和调用。在UObject::GetAssetRegistryTags中调用使用，把该属性的值作为AssetData的Tag供给AssetRegistry

Localized

- **功能描述:** 此属性的值将拥有一个定义的本地化值。多用于字符串。暗示为 ReadOnly。该值有一个本地化值。最常标记在string上
- **元数据类型:** bool
- **引擎模块:** Behavior
- **限制类型:** FString

BlueprintAssignable

- **功能描述:** 在蓝图中可以为这个多播委托绑定事件
- **元数据类型:** bool
- **引擎模块:** Blueprint
- **限制类型:** Multicast Delegates
- **作用机制:** 在PropertyFlags中加入CPF_BlueprintAssignable
- **常用程度:** ★★★

C++的测试代码:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FMyDynamicMulticastDelegate_One,
int32, value);

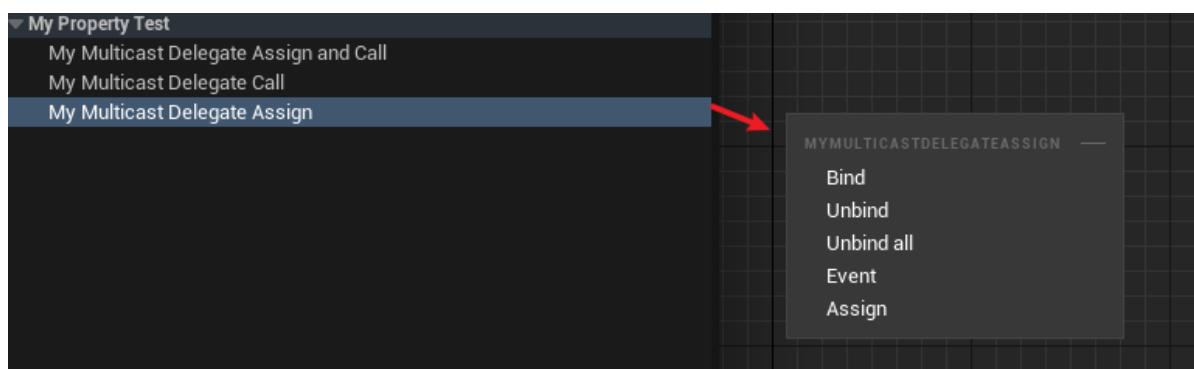
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateCall;

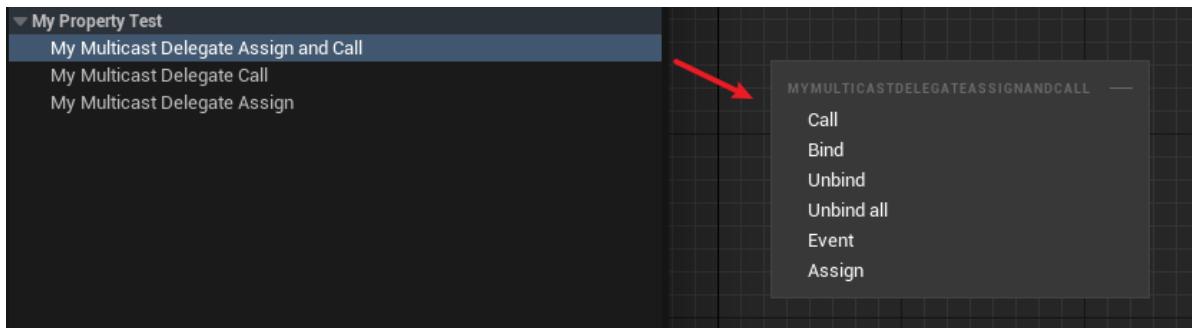
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssign;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyDynamicMulticastDelegate_One MyMulticastDelegate;
```

蓝图中的表现:



因此一般建议二者标记都加上:



BlueprintAuthorityOnly

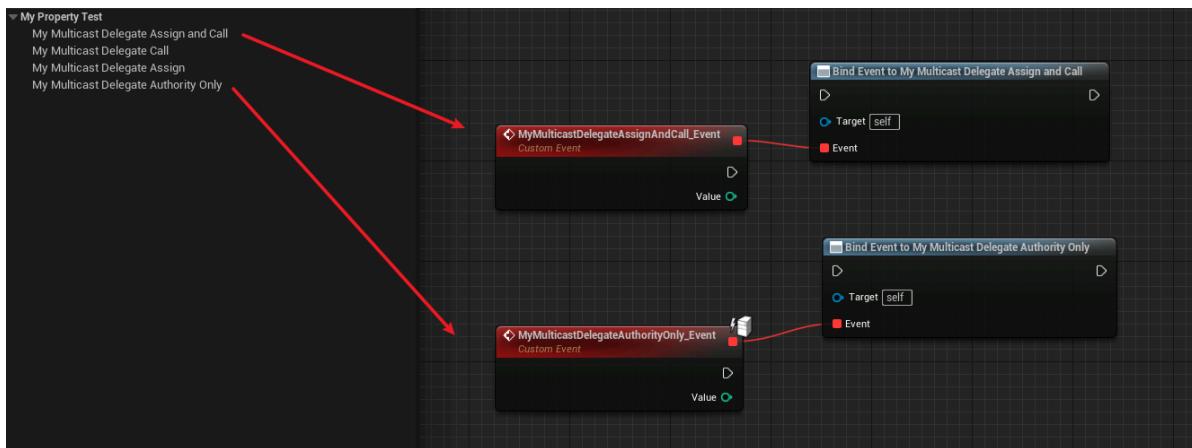
- 功能描述:** 只能绑定为BlueprintAuthorityOnly的事件，让该多播委托只接受在服务端运行的事件
- 元数据类型:** bool
- 引擎模块:** Blueprint, Network
- 限制类型:** Multicast Delegates
- 作用机制:** 在PropertyFlags中加入CPF_BlueprintAuthorityOnly
- 常用程度:** ★★★

测试代码:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable, BlueprintAuthorityOnly)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAuthorityOnly;
```

蓝图中表现:



BlueprintCallable

- 功能描述:** 在蓝图中可以调用这个多播委托
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 限制类型:** Multicast Delegates

- **作用机制:** 在PropertyFlags中加入CPF_BlueprintCallable

- **常用程度:** ★★★

在蓝图中可以调用这个多播委托。

示例代码:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FMyDynamicMulticastDelegate_One,
int32, Value);

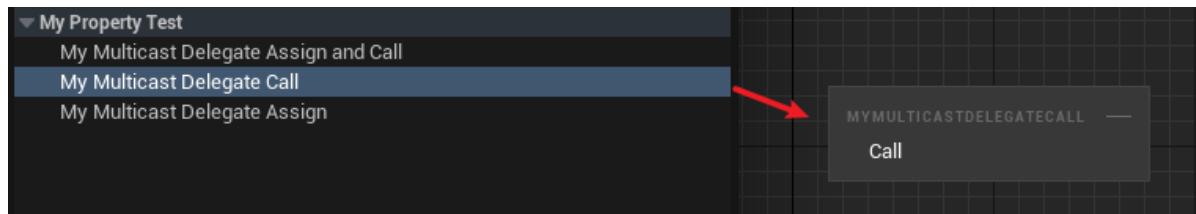
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssign;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyDynamicMulticastDelegate_One MyMulticastDelegate;
```

示例效果:



注意BlueprintAssignable和BlueprintCallable只能用于多播委托:

```
DECLARE_DYNAMIC_DELEGATE_OneParam(FMyDynamicSinglecastDelegate_One, int32,
Value);

//编译报错: 'BlueprintCallable' is only allowed on a property when it is a
multicast delegate
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicSinglecastDelegate_One MyMyDelegate4;

//编译报错: 'BlueprintAssignable' is only allowed on a property when it is a
multicast delegate
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicSinglecastDelegate_One MyMyDelegate5;
```

BlueprintGetter

- **功能描述:** 为属性定义一个自定义的Get函数来读取。
- **元数据类型:** string="abc"
- **引擎模块:** Blueprint

- **作用机制:** 在PropertyFlags中加入CPF_BlueprintReadOnly、CPF_BlueprintVisible，在Meta中加入BlueprintGetter
- **常用程度:** ★★★

为属性定义一个自定义的Get函数来读取。

如果没有设置BlueprintSetter或BlueprintReadWrite，则会默认设置BlueprintReadOnly，这个属性变成只读的。

示例代码:

```
public:
    // Blueprint Getter = , Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h
    UFUNCTION(BlueprintGetter, Category = Blueprint)      // or BlueprintPure
    int32 MyInt_Getter() const { return MyInt_WithGetter * 2; }

    // Blueprint Setter = , Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h
    UFUNCTION(BlueprintSetter, Category = Blueprint)      // or BlueprintCallable
    void MyInt_Setter(int NewValue) { MyInt_WithSetter = NewValue / 4; }

private:
    // Blueprint Getter = MyInt_Getter, Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h
    // PropertyFlags: CPF_BlueprintVisible | CPF_BlueprintReadOnly |
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintGetter = MyInt_Getter, Category = Blueprint)
    int32 MyInt_WithGetter = 123;

    // Blueprint Setter = MyInt_Setter, Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h
    // PropertyFlags: CPF_BlueprintVisible | CPF_ZeroConstructor |
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
    CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintSetter = MyInt_Setter, Category = Blueprint)
    int32 MyInt_WithSetter = 123;
```

示例效果:

可见MyInt_WithGetter是只读的。

而MyInt_WithSetter 是可读写的。



BlueprintReadOnly

- **功能描述:** 此属性可由蓝图读取，但不能被修改。
- **元数据类型:** bool

- **引擎模块:** Blueprint
- **作用机制:** 在PropertyFlags中加入CPF_BlueprintVisible, CPF_BlueprintReadOnly
- **常用程度:** ★★★★★

此属性可由蓝图读取，但不能被修改。此说明符与 BlueprintReadWrite 说明符不兼容。

示例代码:

```
public:
    //PropertyFlags:    CPF_BlueprintVisible | CPF_ZeroConstructor |
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, Category = Blueprint)
    int32 MyInt_ReadWrite = 123;
    //PropertyFlags:    CPF_BlueprintVisible | CPF_BlueprintReadOnly |
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadOnly, Category = Blueprint)
    int32 MyInt_ReadOnly = 123;
```

示例效果:

指定蓝图中只读：



原理:

有CPF_BlueprintVisible 就可以Get

加上CPF_BlueprintReadOnly 后就不能修改。

```
EPropertyAccessResultFlags FPropertyAccessUtil::CanGetPropertyValue(const
FProperty* InProp)
{
    if (!InProp->HasAnyPropertyFlags(CPF_Edit | CPF_BlueprintVisible |
CPF_BlueprintAssignable))
    {
        return EPropertyAccessResultFlags::PermissionDenied |
EPropertyAccessResultFlags::AccessProtected;
    }

    return EPropertyAccessResultFlags::Success;
}

FBlueprintEditorUtils::EPropertyWritabilityState
FBlueprintEditorUtils::IsPropertyWritableInBlueprint(const UBlueprint* Blueprint,
const FProperty* Property)
{
    if (Property)
```

```

{
    if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
    {
        return EPropertyWritableState::NotBlueprintVisible;
    }
    if (Property->HasAnyPropertyFlags(CPF_BlueprintReadOnly))
    {
        return EPropertyWritableState::BlueprintReadOnly;
    }
    if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
    {
        const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
        if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
        {
            return EPropertyWritableState::Private;
        }
    }
}
return EPropertyWritableState::Writable;
}

```

BlueprintReadWrite

- 功能描述:** 可从蓝图读取或写入此属性。
- 元数据类型:** bool
- 引擎模块:** Blueprint
- 作用机制:** 在PropertyFlags中加入CPF_BlueprintVisible
- 常用程度:** ★★★★☆

可从蓝图读取或写入此属性。

此说明符与 BlueprintReadOnly 说明符不兼容。

示例代码：

```

public:
    //PropertyFlags: CPF_BlueprintVisible | CPF_ZeroConstructor |
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, Category = Blueprint)
        int32 MyInt_Readwrite = 123;
    //PropertyFlags: CPF_BlueprintVisible | CPF_BlueprintReadOnly |
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadOnly, Category = Blueprint)
        int32 MyInt_ReadOnly = 123;

```

示例效果：

蓝图中可读写：



原理：

如果有CPF_Edit | CPF_BlueprintVisible | CPF_BlueprintAssignable之一，则可以Get属性。

```
EPropertyAccessResultFlags PropertyAccessUtil::CanGetPropertyValue(const
FProperty* InProp)
{
    if (!InProp->HasAnyPropertyFlags(CPF_Edit | CPF_BlueprintVisible |
CPF_BlueprintAssignable))
    {
        return EPropertyAccessResultFlags::PermissionDenied |
EPropertyAccessResultFlags::AccessProtected;
    }

    return EPropertyAccessResultFlags::Success;
}
```

BlueprintSetter

- 功能描述：**采用一个自定义的set函数来读取。
- 元数据类型：**string="abc"
- 引擎模块：**Blueprint
- 作用机制：**在PropertyFlags中加入CPF_BlueprintVisible，在Meta中加入BlueprintSetter
- 常用程度：**★★★

采用一个自定义的set函数来读取。

会默认设置BlueprintReadWrite。

测试代码：

```
public:
    // (BlueprintGetter = , Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h)
    UFUNCTION(BlueprintGetter, Category = Blueprint)      // or BlueprintPure
    int32 MyInt_Getter() const { return MyInt_WithGetter * 2; }

    // (BlueprintSetter = , Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h)
    UFUNCTION(BlueprintSetter, Category = Blueprint)      // or BlueprintCallable
    void MyInt_Setter(int NewValue) { MyInt_WithSetter = NewValue / 4; }

private:
    // (BlueprintGetter = MyInt_Getter, Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h)
```

```

    //PropertyFlags:    CPF_BlueprintVisible | CPF_BlueprintReadOnly | 
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintGetter = MyInt_Getter, Category = Blueprint)
        int32 MyInt_WithGetter = 123;

    //((BlueprintSetter = MyInt_Setter, Category = Blueprint, ModuleRelativePath =
    Property/MyProperty_Test.h)
    //PropertyFlags:    CPF_BlueprintVisible | CPF_ZeroConstructor | 
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintSetter = MyInt_Setter, Category = Blueprint)
        int32 MyInt_WithSetter = 123;

```

蓝图表现：



原理：

如果有MD_PropertySetFunction则用它来作为Set的调用。

```

void UK2Node_VariableSet::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
    // If property has a BlueprintSetter accessor, then replace the variable
    // get node with a call function
    if (VariableProperty)
    {
        // todo check with BP team if we need to test if the variable has
        // native Setter
        const FString& SetFunctionName = VariableProperty-
>GetMetaData(FBlueprintMetadata::MD_PropertySetFunction);
        if (!SetFunctionName.IsEmpty())
        {
        }
    }
}

```

Getter

- 功能描述：**为属性增加一个C++的Get函数，只在C++层面应用。
- 元数据类型：**string="abc"
- 引擎模块：** Blueprint
- 常用程度：** ★★★

为属性增加一个C++的Get函数，只在C++层面应用。

- Getter上如不提供函数名，那就用默认的GetXXX的名字。也可以提供另外一个函数名。

- 这些Getter函数是不加UFUNCTION的，这点要和BlueprintGetter区分。
- 感觉更好的名字是NativeGetter。
- GetXXX的函数必须自己手写，否则UHT会报错。
- 我们当然也可以自己写GetSet函数，不需要写Getter和Setter的元数据。但写上Getter和Setter的好处是，万一在项目里别的地方，用到了反射来获取和设置值，这个时候如果没有标上Getter和Setter，就会直接从属性上获取值，从而跳过我们想要的自定义Get/Set流程。

测试代码：

```

public:
    //GetterFunc: Has Native Getter
    UPROPERTY(BlueprintReadWrite, Getter)
    float MyFloat = 1.0f;

    //GetterFunc: Has Native Getter
    UPROPERTY(BlueprintReadWrite, Getter = GetMyCustomFloat)
    float MyFloat2 = 1.0f;
public:
    float GetMyFloat()const { return MyFloat + 100.f; }

    float GetMyCustomFloat()const { return MyFloat2 + 100.f; }

void UMyProperty_Get::RunTest()
{
    float value1=MyFloat;

    FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));
    float value2=0.f;

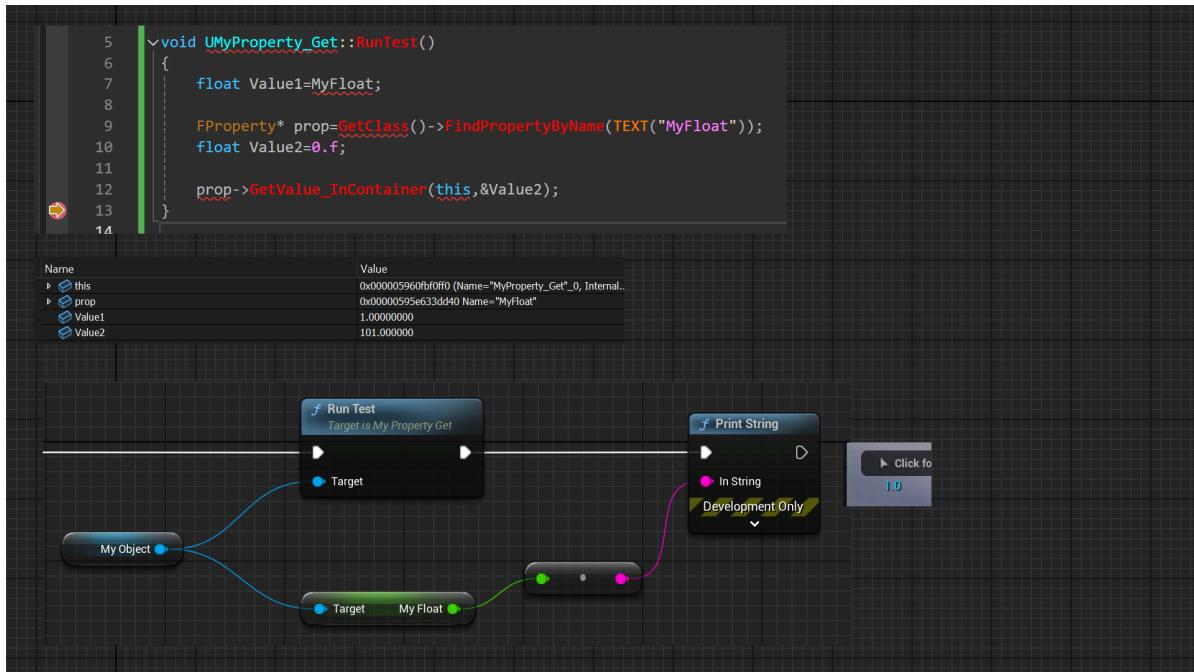
    prop->GetValue_InContainer(this,&value2);
}

```

蓝图表现：

在测试的时候，可见如果是用GetValue_InContainer这种反射的方式来获取值，就会自动的调用到GetMyFloat，从而返回不同的值。

在蓝图里直接Get MyFloat 是依然是1.



原理：

UHT在分析Getter标记后，会在gen.cpp里生成相应的函数包装。在构建FProperty的时候，就会创建TPropertyWithSetterAndGetter，之后在GetSingleValue_InContainer的时候就会调用到CallGetter。

```

void UMyProperty_Get::GetMyFloat_WrapperImpl(const void* Object, void* OutValue)
{
    const UMyProperty_Get* Obj = (const UMyProperty_Get*)Object;
    float& Result = *(float*)OutValue;
    Result = (float)Obj->GetMyFloat();
}

const UECodeGen_Private::FFloatPropertyParams
Z_Construct_UClass_UMyProperty_Get_Statics::NewProp_MyFloat = { "MyFloat",
    nullptr, (EPropertyFlags)0x0100000000000004,
    UECodeGen_Private::EPropertyGenFlags::Float,
    RF_Public|RF_Transient|RF_MarkAsNative, nullptr,
    &UMyProperty_Get::GetMyFloat_WrapperImpl, 1, STRUCT_OFFSET(UMyProperty_Get,
    MyFloat), METADATA_PARAMS(UE_ARRAY_COUNT(NewProp_MyFloat_MetaData),
    NewProp_MyFloat_MetaData) };

template <typename PropertyType, typenamePropertyParamsType>
PropertyParams* NewFProperty(FFieldVariant Outer, const FPropertyParamsBase&
PropBase)
{
    constPropertyParamsType& Prop = (constPropertyParamsType&)PropBase;
   PropertyParams* NewProp = nullptr;

    if (Prop.SetterFunc || Prop.GetterFunc)
    {
        NewProp = new TPropertyWithSetterAndGetter<PropertyType>(Outer, Prop);
    }
    else
    {
        NewProp = new PropertyType(Outer, Prop);
    }
}

```

```

}

void FProperty::GetSingleValue_InContainer(const void* InContainer, void*
OutValue, int32 ArrayIndex) const
{
    checkf(ArrayIndex <= ArrayDim, TEXT("ArrayIndex (%d) must be less than the
property %s array size (%d)"), ArrayIndex, *GetFullName(), ArrayDim);
    if (!HasGetter())
    {
        // Fast path - direct memory access
        CopySingleValue(OutValue,
ContainerVoidPtrToValuePtrInternal((void*)InContainer, ArrayIndex));
    }
    else
    {
        if (ArrayDim == 1)
        {
            // Slower but no mallocs. We can copy the value directly to the
resulting param
            CallGetter(InContainer, OutValue);
        }
        else
        {
            // Malloc a temp value that is the size of the array. Getter will
then copy the entire array to the temp value
            uint8* ValueArray = (uint8*)AllocateAndInitializeValue();
            GetValue_InContainer(InContainer, ValueArray);
            // Copy the item we care about and free the temp array
            CopySingleValue(OutValue, ValueArray + ArrayIndex * ElementSize);
            DestroyAndFreeValue(ValueArray);
        }
    }
}
}

```

Setter

- 功能描述:** 为属性增加一个C++的Set函数，只在C++层面应用。
- 元数据类型:** string="abc"
- 引擎模块:** Blueprint
- 关联项:** Getter
- 常用程度:** ★★★

为属性增加一个C++的Set函数，只在C++层面应用。

- Getter上如不提供函数名，那就用默认的SetXXX的名字。也可以提供另外一个函数名。
- 这些Getter函数是不加UFUNCTION的，这点要和BlueprintGetter区分。
- 感觉更好的名字是NativeSetter。
- SetXXX的函数必须自己手写，否则UHT会报错。

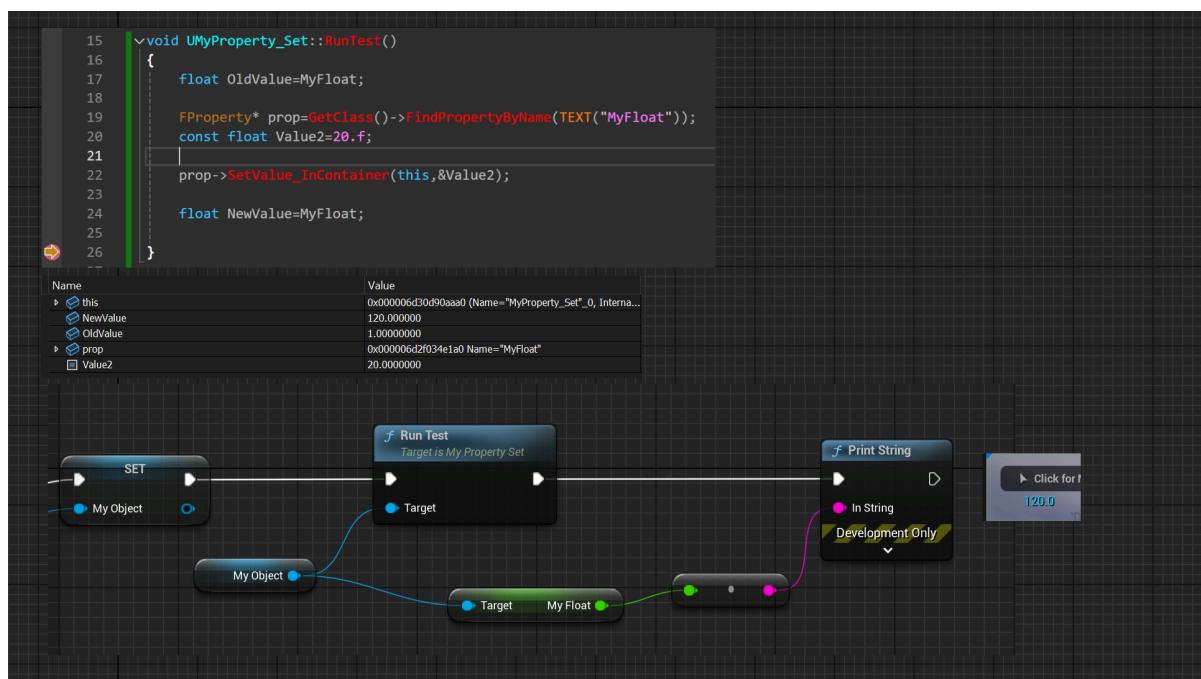
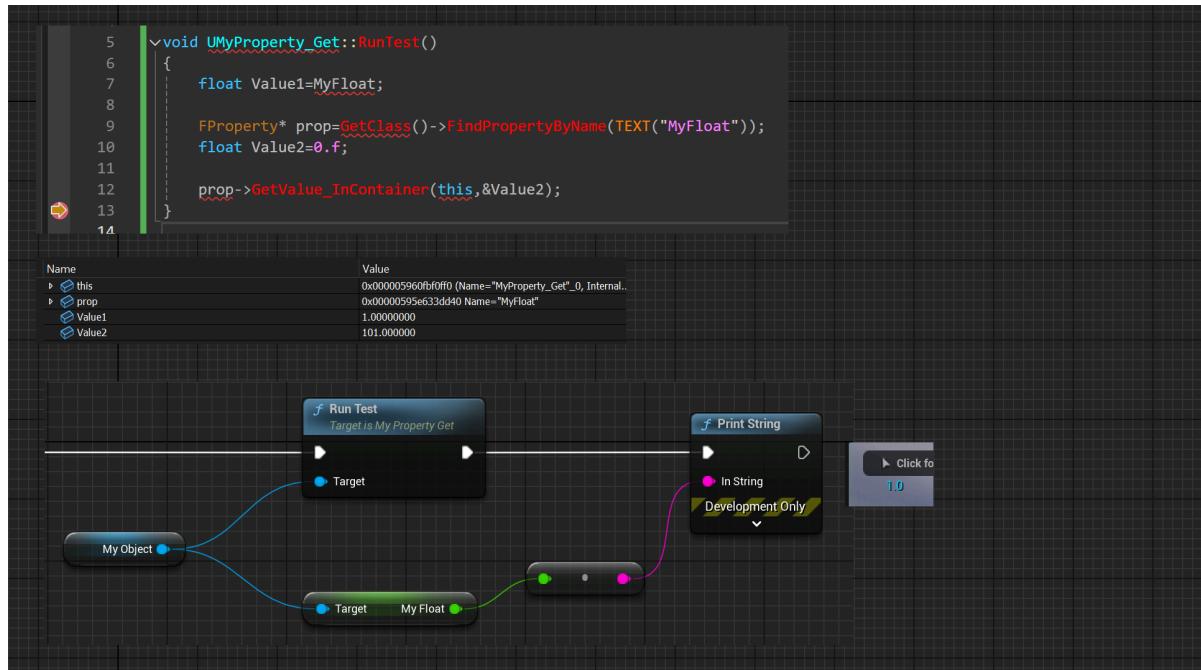
- 我们当然也可以自己写GetSet函数，不需要写Getter和Setter的元数据。但写上Getter和Setter的好处是，万一在项目里别的地方，用到了反射来获取和设置值，这个时候如果没有标上Getter和Setter，就会直接从属性上获取值，从而跳过我们想要的自定义Get/Set流程。

测试代码：

```
public:  
    UPROPERTY(BlueprintReadWrite, Setter)  
    float MyFloat = 1.0f;  
  
    UPROPERTY(BlueprintReadWrite, Setter = SetMyCustomFloat)  
    float MyFloat2 = 1.0f;  
public:  
    void SetMyFloat(float val) { MyFloat = val + 100.f; }  
    void SetMyCustomFloat(float val) { MyFloat2 = val + 100.f; }  
  
public:  
    UFUNCTION(BlueprintCallable)  
    void RunTest();  
};  
  
void UMyProperty_Set::RunTest()  
{  
    float oldValue=MyFloat;  
  
    FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));  
    const float value2=20.f;  
  
    prop->SetValue_InContainer(this,&value2);  
  
    float newValue=MyFloat;  
}
```

蓝图表现：

在测试的时候，可见如果是用SetValue_InContainer这种反射的方式来获取值，就会自动的调用到SetMyFloat，从而实际上设置到不同的值。



原理：

UHT在分析Setter标记后，会在gen.cpp里生成相应的函数包装。在构建FProperty的时候，就会创建TPropertyWithSetterAndGetter，之后在GetSingleValue_InContainer的时候就会调用到CallGetter。

```

void UMyProperty_Set::SetMyFloat_WrapperImpl(void* Object, const void* InValue)
{
    UMyProperty_Set* Obj = (UMyProperty_Set*)Object;
    float& Value = *(float*)InValue;
    Obj->SetMyFloat(value);
}

```

```

const UECodeGen_Private::FFloatPropertyParams
Z_Construct_UClass_UMyProperty_Set_Statics::NewProp_MyFloat = { "MyFloat",
nullptr, (EPropertyFlags)0x0010000000000004,
UECodeGen_Private::EPropertyGenFlags::Float,
RF_Public|RF_Transient|RF_MarkAsNative, &UMyProperty_Set::SetMyFloat_WrapperImpl,
nullptr, 1, STRUCT_OFFSET(UMyProperty_Set, MyFloat),
METADATA_PARAMS(UE_ARRAY_COUNT(NewProp_MyFloat_MetaData),
NewProp_MyFloat_MetaData) };

template <typename PropertyType, typenamePropertyParamsType>
.PropertyType* NewFProperty(FFieldVariant Outer, const FPropertyParamsBase&
PropBase)
{
    constPropertyParamsType& Prop = (constPropertyParamsType&)PropBase;
    PropertyType* NewProp = nullptr;

    if (Prop.SetterFunc || Prop.GetterFunc)
    {
        NewProp = new TPropertyWithSetterAndGetter<.PropertyType>(Outer, Prop);
    }
    else
    {
        NewProp = new PropertyType(Outer, Prop);
    }
}

void FProperty::setSingleValue_InContainer(void* OutContainer, const void*
InValue, int32 ArrayIndex) const
{
    checkf(ArrayIndex <= ArrayDim, TEXT("ArrayIndex (%d) must be less than the
property %s array size (%d)"), ArrayIndex, *GetFullName(), ArrayDim);
    if (!HasSetter())
    {
        // Fast path - direct memory access
        CopySinglevalue(ContainerVoidPtrToValuePtrInternal((void*)OutContainer,
ArrayIndex), InValue);
    }
    else
    {
        if (ArrayDim == 1)
        {
            // Slower but no mallocs. We can copy the value directly to the
resulting param
            CallSetter(OutContainer, InValue);
        }
        else
        {
            // Malloc a temp value that is the size of the array. We will then
copy the entire array to the temp value
            uint8* ValueArray = (uint8*)AllocateAndInitializeValue();
            GetValue_InContainer(OutContainer, ValueArray);
            // Replace the value at the specified index in the temp array with
the InValue
            Copysinglevalue(ValueArray + ArrayIndex * ElementSize, InValue);
            // Now call a setter to replace the entire array and then destroy the
temp value
        }
    }
}

```

```
        CallSetter(OutContainer, ValueArray);
        DestroyAndFreeValue(ValueArray);
    }
}
```

Config

- **功能描述:** 指定该属性是一个配置属性，该属性可以被序列化读写到ini文件（路径由uclass的config标签指定）中。
- **元数据类型:** bool
- **引擎模块:** Config
- **作用机制:** CPF_Config
- **常用程度:** ★★★

指定该属性是一个配置属性，该属性可以被序列化读写到ini文件（路径由uclass的config标签指定）中。

在载入的时候会自动从ini中加载。如果没再加写标记，则会隐含该属性为ReadOnly。

参见UCLASS中的config标记的示例代码和效果。

GlobalConfig

- **功能描述:** 和Config一样指定该属性可作为配置读取和写入ini中，但只会读取写入到配置文件里基类的值，而不会使用配置文件里子类里的值。
- **元数据类型:** bool
- **引擎模块:** Config
- **作用机制:** 在PropertyFlags中加入CPF_GlobalConfig
- **常用程度:** ★★★

和Config一样指定该属性可作为配置读取和写入ini中，但只会读取写入到配置文件里基类的值，而不会使用配置文件里子类里的值。

但是不同点在于，该属性在LoadConfig的时候，只会读取基类的ini，而不会去读取子类的ini。因为只有基类里的Ini设置在生效，相当于全局只有一个配置在生效，因此名字叫做GlobalConfig。

示例代码：

```
UCLASS(Config = MyOtherGame)
class INSIDER_API UMyProperty_Config :public UObject
{
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
    int32 MyPropertywithGlobalConfig = 123;
};
```

```

UCLASS(Config = MyOtherGame)
class INSIDER_API UMyProperty_Config_Child :public UMyProperty_Config
{
    GENERATED_BODY()
public:
};

void UMyProperty_Config_Test::TestConfigSave()
{
    FString fileName = FPaths::ProjectConfigDir() / TEXT("MyOtherGame.ini");
    fileName = FConfigCacheIni::NormalizeConfigIniPath(fileName);

    {
        UMyProperty_Config* testObject = NewObject<UMyProperty_Config>
        (GetTransientPackage(), TEXT("testObject"));

        testObject->MyProperty = 777;
        testObject->MyPropertyWithConfig = 777;
        testObject->MyPropertyWithGlobalConfig = 777;

        testObject->SaveConfig(CPF_Config, *fileName);
    }

    {
        UMyProperty_Config_Child* testObject =
        NewObject<UMyProperty_Config_Child>(GetTransientPackage(),
        TEXT("testObjectChild"));

        testObject->MyProperty = 888;
        testObject->MyPropertyWithConfig = 888;
        testObject->MyPropertyWithGlobalConfig = 888;

        testObject->SaveConfig(CPF_Config, *fileName);
    }
}

void UMyProperty_Config_Test::TestConfigLoad()
{
    FString fileName = FPaths::ProjectConfigDir() / TEXT("MyOtherGame.ini");
    fileName = FConfigCacheIni::NormalizeConfigIniPath(fileName);

    UMyProperty_Config* testObject = NewObject<UMyProperty_Config>
    (GetTransientPackage(), TEXT("testObject"));
    testObject->LoadConfig(nullptr, *fileName);

    UMyProperty_Config_Child* testObjectChild =
    NewObject<UMyProperty_Config_Child>(GetTransientPackage(),
    TEXT("testObjectChild"));
    testObjectChild->LoadConfig(nullptr, *fileName);
}

```

示例效果：

TestConfigSave之后，MyPropertyWithGlobalConfig=888，可见保存的时候也只会保存在基类上。

```

[ /Script/Insider.MyProperty_Config]
MyPropertywithConfig=777
MyPropertywithGlobalConfig=888

[ /Script/Insider.MyProperty_Config_Child]
MyPropertywithConfig=888

```

为了测试，假如手动把配置里的值改为：然后再进行TestConfigLoad测试

```

[ /Script/Insider.MyProperty_Config]
MyPropertywithConfig=777
MyPropertywithGlobalConfig=888

[ /Script/Insider.MyProperty_Config_Child]
MyPropertywithConfig=888
MyPropertywithGlobalConfig=999

```

显示效果：

可见testObjectChild 的值并没有使用ini里MyProperty_Config_Child下的999的值，而是同样的888。



原理：

如果是bGlobalConfig，会采用基类。

```

void Uobject::LoadConfig( UClass* ConfigClass/*=NULL*/ , const TCHAR*
InFilename/*=NULL*/ , uint32 PropagationFlags/*=LCFNone*/ , FProperty*
PropertyToLoad/*=NULL*/ )
{
    const bool bGlobalConfig = (Property->PropertyFlags&CPF_GlobalConfig) != 0;
    UClass* OwnerClass = Property->GetOwnerClass();

    UClass* BaseClass = bGlobalConfig ? OwnerClass : ConfigClass;
    if ( !bPerObject )
    {
        ClassSection = BaseClass->GetPathName();
        LongCommitName = BaseClass->GetOutermost()->GetFName();

        // allow the class to override the expected section name
        overrideConfigSection(ClassSection);
    }

    // globalconfig properties should always use the owning class's config
    file
    // specifying a value for InFilename will override this behavior (as it
    does with normal properties)
    const FString& PropFileName = (bGlobalConfig && InFilename == NULL) ?
    OwnerClass->GetConfigName() : filename;
}

```

AdvancedDisplay

- **功能描述:** 被折叠到高级栏下，要手动打开。一般用在不太常用的属性上面。
- **元数据类型:** bool
- **引擎模块:** DetailsPanel, Editor
- **作用机制:** 在PropertyFlags中加入CPF_AdvancedDisplay
- **常用程度:** ★★★★☆

被折叠到高级栏下，要手动打开。一般用在不太常用的属性上面。

示例代码:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
    int32 MyInt_SimpleDisplay = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
    int32 MyInt_AdvancedDisplay = 123;
}
```

示例效果:



原理:

如果CPF_AdvancedDisplay, bAdvanced =true

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    // Property is advanced if it is marked advanced or the entire class is
    // advanced and the property not marked as simple
    static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
    bool bAdvanced = Property.IsValid() ? (Property-
>HasAnyPropertyParams(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyParams(
    CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()-
>GetBoolMetaData(Name_AdvancedClassDisplay) ) ) : false;
}
```

Category

- **功能描述:** 指定属性的类别，使用 | 运算符定义嵌套类目。
- **元数据类型:** strings="a|b|c"
- **引擎模块:** DetailsPanel, Editor
- **作用机制:** 在Meta中加入Category
- **常用程度:** ★★★★☆

指定属性的类别，使用 | 运算符定义嵌套类目。

示例代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
        int32 MyInt_SimpleDisplay = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
        int32 MyInt_AdvancedDisplay = 123;
public:
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, Category = Edit)
        int32 MyInt_EditAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
    CPF_DisableEditOnInstance | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditDefaultsOnly, Category = Edit)
        int32 MyInt_EditDefaultsOnly = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
    CPF_DisableEditOnTemplate | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditInstanceOnly, Category = Edit)
        int32 MyInt_EditInstanceOnly = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_EditConst | 
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleAnywhere, Category = Edit)
        int32 MyInt_VisibleAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
    CPF_DisableEditOnInstance | CPF_EditConst | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
```

```
UPROPERTY(VisibleDefaultsOnly, Category = Edit)
    int32 MyInt_VisibleDefaultsonly = 123;
}
```

示例效果：



原理：

比较简单，把值设置到meta里的Category，之后读取出来使用。

EditAnywhere

- 功能描述：**在默认值和实例的细节面板上均可编辑
- 元数据类型：**bool
- 引擎模块：**DetailsPanel, Editor
- 作用机制：**在PropertyFlags中加入CPF_Edit
- 常用程度：**★★★★★

在默认值和实例的细节面板上均可编辑。

示例代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
public:
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, Category = Edit)
    int32 MyInt_EditAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
    CPF_DisableEditOnInstance | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditDefaultsonly, Category = Edit)
    int32 MyInt_EditDefaultsonly = 123;
```

```

        //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
CPF_DisableEditOnTemplate | CPF_IsPlainOldDataOldData | CPF_NoDestructor |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditInstanceOnly, Category = Edit)
        int32 MyInt_EditInstanceOnly = 123;

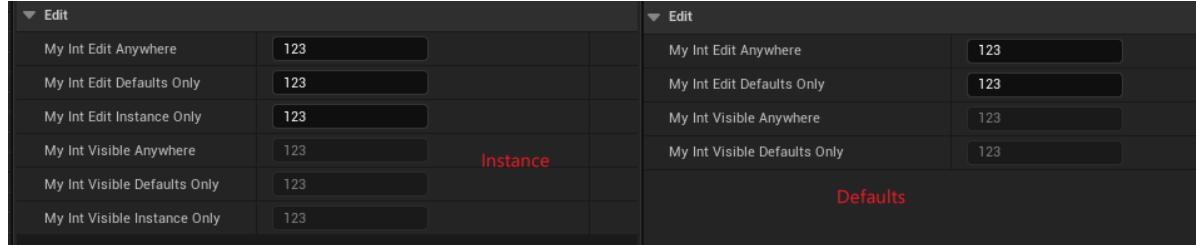
        //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_EditConst |
CPF_IsPlainOldDataOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleAnywhere, Category = Edit)
        int32 MyInt_VisibleAnywhere = 123;

        //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
CPF_DisableEditOnInstance | CPF_EditConst | CPF_IsPlainOldDataOldData | CPF_NoDestructor |
| CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleDefaultsOnly, Category = Edit)
        int32 MyInt_VisibleDefaultsOnly = 123;

        //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
CPF_DisableEditOnTemplate | CPF_EditConst | CPF_IsPlainOldDataOldData | CPF_NoDestructor |
| CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleInstanceOnly, Category = Edit)
        int32 MyInt_VisibleInstanceOnly = 123;
}

```

示例效果：



原理：

CPF_Edit在源码里有非常多的使用，决定了很多地方属性是否可以显示和编辑。有兴趣可以自己去搜搜CPF_Edit的使用。

EditDefaultsOnly

- 功能描述：**只能在默认值面板里编辑
- 元数据类型：**bool
- 引擎模块：**DetailsPanel, Editor
- 作用机制：**在PropertyFlags中加入CPF_Edit, CPF_DisableEditOnInstance
- 常用程度：**★★★★★

一并参见EditAnywhere里的示例代码和效果。

EditFixedSize

- 功能描述：**在细节面板上不允许改变该容器的元素个数。

- **元数据类型:** bool
- **引擎模块:** DetailsPanel, Editor
- **限制类型:** TArray, TSet, TMap
- **作用机制:** 在PropertyFlags中加入CPF_EditFixedSize
- **常用程度:** ★★★

在细节面板上不允许改变该容器的元素个数。

只适用于容器。这能防止用户通过虚幻编辑器属性窗口修改容器的元素个数。

但在C++代码和蓝图中依然可以修改的。

示例代码：

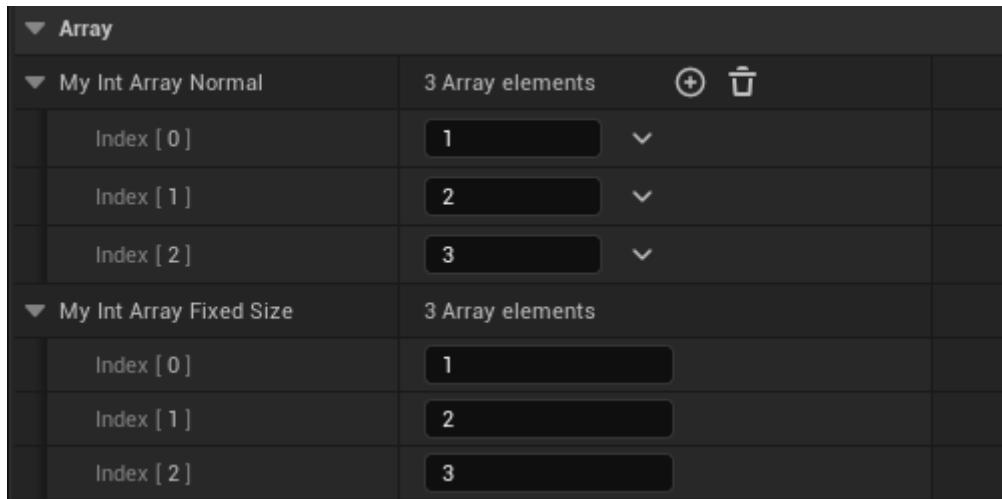
以TArray为例，其他同理。

```
UPROPERTY(EditAnywhere, Category = Array)
TArray<int32> MyIntArray_Normal{1,2,3};

UPROPERTY(EditAnywhere, EditFixedSize,Category = Array)
TArray<int32> MyIntArray_FixedSize{1,2,3};
```

示例效果：

蓝图中的表现，前者可以动态再添加元素。后者不可。



原理：

如果有CPF_EditFixedSize，则不会添加+和清空的按钮。

```

void PropertyEditorHelpers::GetRequiredPropertyButtons( TSharedRef< FPropertyNode>
PropertyNode, TArray< EPropertyButton::Type>& OutRequiredButtons, bool
bUsingAssetPicker )
{
    // Handle a container property.
    if( NodeProperty->IsA(FArrayProperty::StaticClass()) || NodeProperty-
>IsA(FSetProperty::StaticClass()) || NodeProperty-
>IsA(FMapProperty::StaticClass()) )
    {
        if( !(NodeProperty->PropertyFlags & CPF_EditFixedSize) )
        {
            OutRequiredButtons.Add( EPropertyButton::Add );
            OutRequiredButtons.Add( EPropertyButton::Empty );
        }
    }
}

```

EditInstanceOnly

- 功能描述:** 只能在实例的细节面板上编辑该属性
- 元数据类型:** bool
- 引擎模块:** DetailsPanel, Editor
- 作用机制:** 在PropertyFlags中加入CPF_Edit, CPF_DisableEditOnTemplate
- 常用程度:** ★★★★☆

一并参见EditAnywhere里的示例代码和效果。

Interp

- 功能描述:** 指定该属性值可暴露到时间轴里编辑，在平常的Timeline或UMG的动画里使用。
- 元数据类型:** bool
- 引擎模块:** Sequencer
- 作用机制:** 在PropertyFlags中加入CPF_Edit, CPF_BlueprintVisible, CPF_Interp
- 常用程度:** ★★★

该属性可以暴露到时间轴里，一般用来编辑动画。

示例代码：

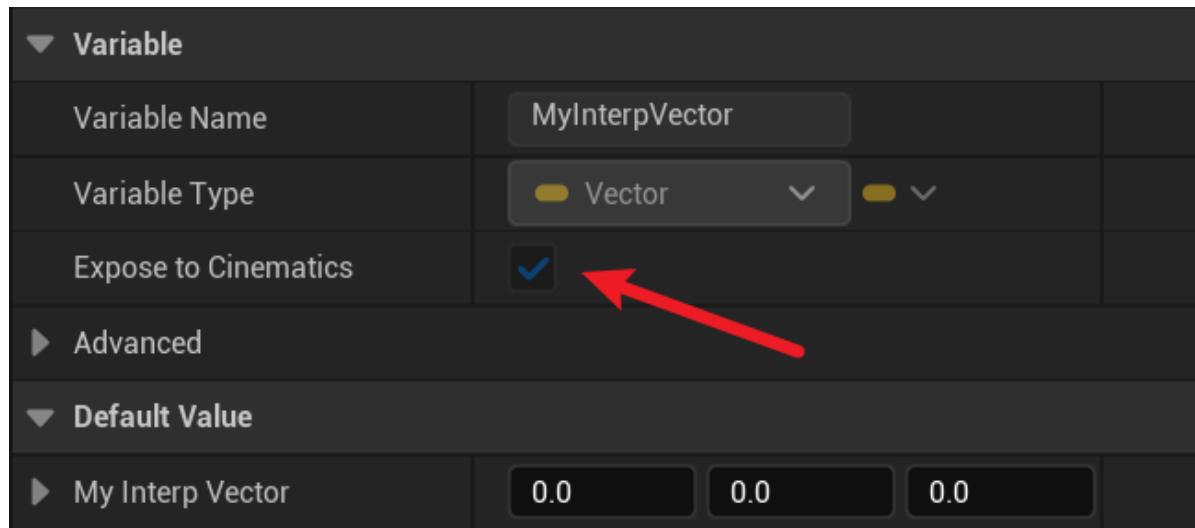
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Interp :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Interp, Category = Animation)
        FVector MyInterpVector;
};

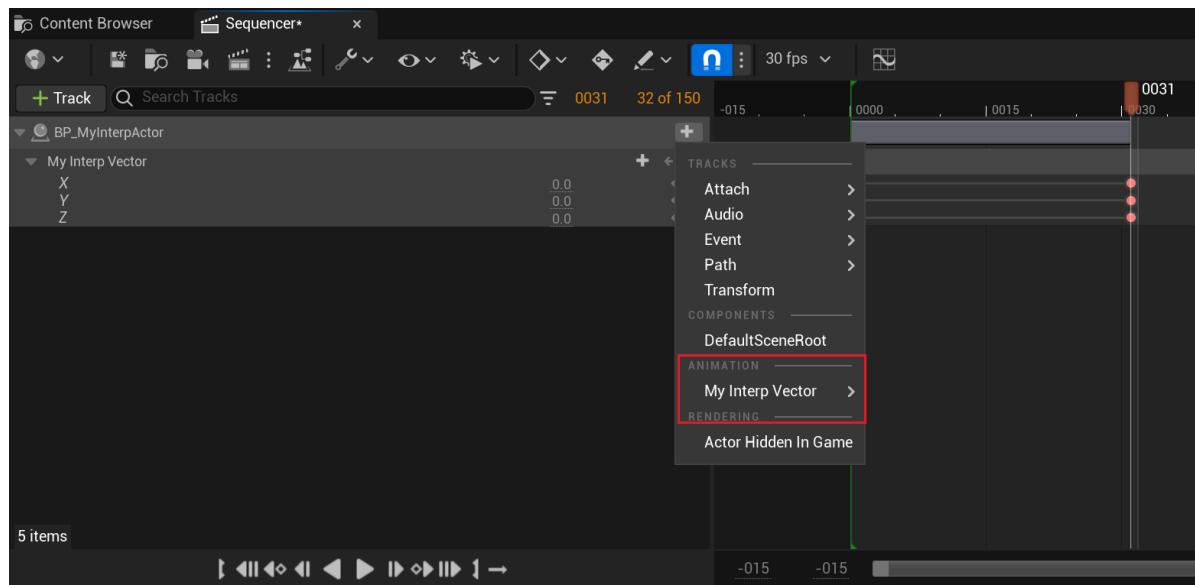
```

示例效果：

影响的是属性上的该标志



从而可以在Sequencer里对该属性添加Track



NoClear

- 功能描述:** 指定该属性的编辑选项中不出现Clear按钮，不允许置null。
- 元数据类型:** bool
- 引擎模块:** DetailsPanel, Editor
- 限制类型:** 引用类型
- 作用机制:** 在PropertyFlags中加入CPF_NoClear
- 常用程度:** ★★★

指定该属性的编辑选项中不出现Clear按钮。

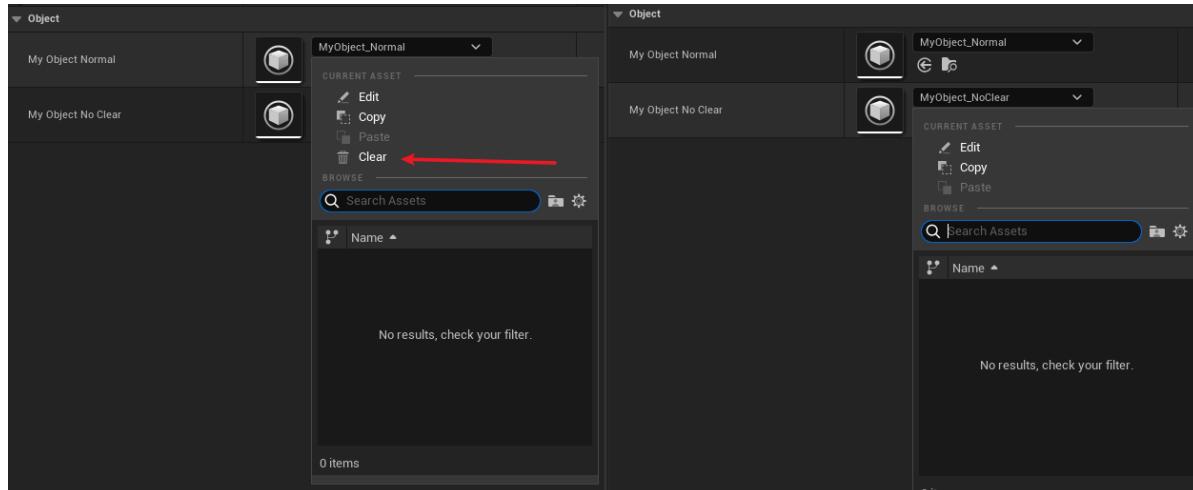
作用是阻止用户在编辑器面板上将此Object引用设为null。但其实也可用在其他表示一个引用类型的结构上，比如FPrimaryAssetId, FInstancedStruct, FDataRegistryType等。

示例代码：

```
UPROPERTY(EditAnywhere, Category = Object)
class UMyClass_Default* MyObject_Normal;
//PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_NoClear |
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, NoClear, Category = Object)
class UMyClass_Default* MyObject_NoClear;

//构造函数赋值:
MyObject_Normal = CreateDefaultSubobject<UMyClass_Default>("MyObject_Normal");
MyObject_NoClear = CreateDefaultSubobject<UMyClass_Default>("MyObject_NoClear");
```

示例效果：



原理：

CPF_NoClear在引擎里有挺多使用。

```
const bool bAllowClear = !StructPropertyHandle->GetMetaDataProperty()-
>HasAnyPropertyParams(CPF_NoClear);
```

NonTransactional

- 功能描述：**对该属性的改变操作，不会被包含进编辑器的Undo/Redo命令中。
- 元数据类型：**bool
- 引擎模块：**Editor
- 作用机制：**在PropertyParams中加入CPF_NonTransactional
- 常用程度：**★★

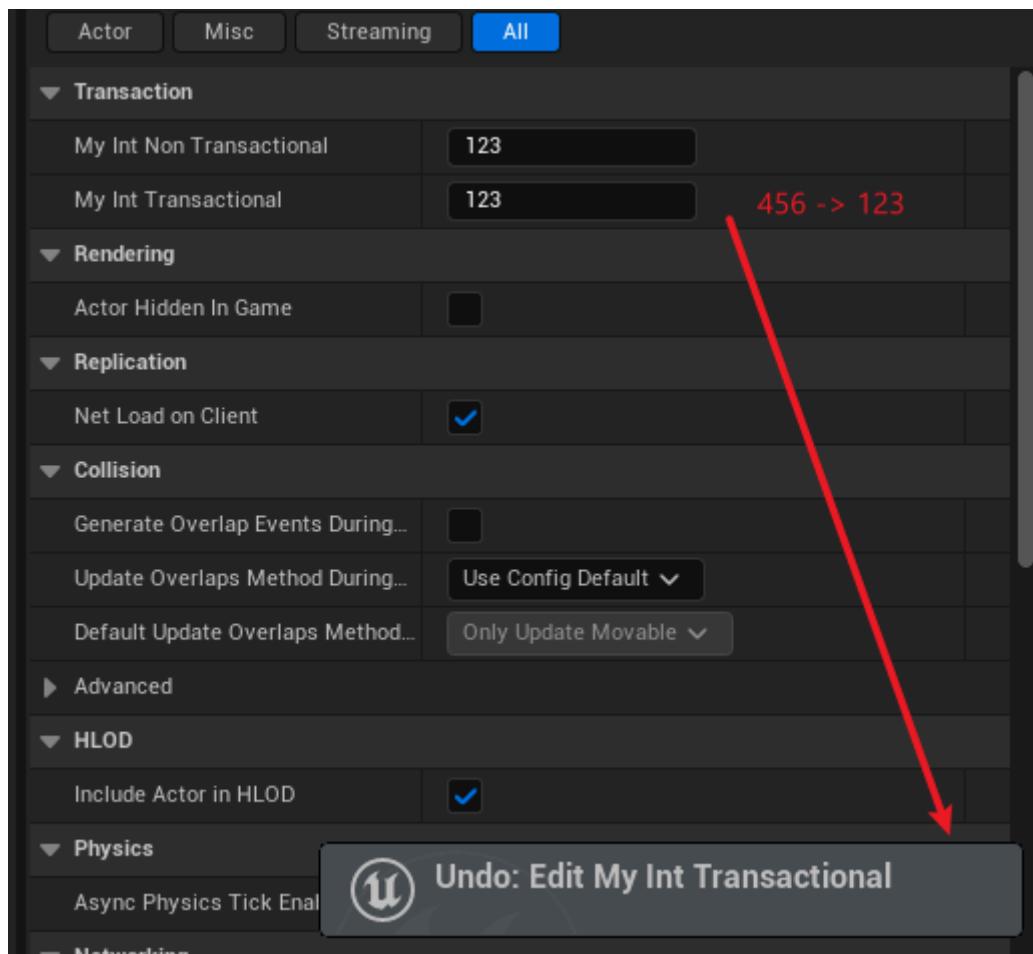
指定该属性的改变，不能在编辑器中通过Ctrl+Z来撤销或Ctrl+Y来重做。在Actor或在BP的Class Defaults都可以生效。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Transaction :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonTransactional, Category = Transaction)
        int32 MyInt_NonTransactional = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Transaction)
        int32 MyInt_Transactional = 123;
};
```

蓝图表现：

在MyInt_Transactional 上可以撤销之前的输入，而MyInt_NonTransactional上的输入无法用Ctrl+Z撤销。



SimpleDisplay

- 功能描述：**在细节面板中直接可见，不折叠到高级中。
- 元数据类型：**bool
- 引擎模块：**DetailsPanel, Editor
- 作用机制：**在PropertyFlags中加入CPF_SimpleDisplay

- 常用程度：★★★

在细节面板中直接可见，不折叠到高级中。

默认情况下本身就是不折叠，但可以用来覆盖掉类上的AdvancedClassDisplay的设置。具体可参见AdvancedClassDisplay的代码和效果。

示例代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
        int32 MyInt_SimpleDisplay = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
        int32 MyInt_AdvancedDisplay = 123;
}
```

示例效果：



原理：

如果有CPF_SimpleDisplay，则bAdvanced =false

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    // Property is advanced if it is marked advanced or the entire class is
    // advanced and the property not marked as simple
    static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
    bool bAdvanced = Property.IsValid() ? (Property-
>HasAnyPropertyParams(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyParams(
    CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()-
>GetBoolMetaData(Name_AdvancedClassDisplay) ) ) : false;
}
```

VisibleAnywhere

- 功能描述：在默认值和实例细节面板均可见，但不可编辑

- **元数据类型:** bool
- **引擎模块:** DetailsPanel, Editor
- **作用机制:** 在PropertyFlags中加入CPF_Edit, CPF_EditConst
- **常用程度:** ★★★★☆

—并参见EditAnywhere里的示例代码和效果。

VisibleDefaultsOnly

- **功能描述:** 在默认值细节面板可见，但不可编辑
- **元数据类型:** bool
- **引擎模块:** DetailsPanel, Editor
- **作用机制:** 在PropertyFlags中加入CPF_Edit, CPF_DisableEditOnInstance
- **常用程度:** ★★★★☆

—并参见EditAnywhere里的示例代码和效果。

VisibleInstanceOnly

- **功能描述:** 在实例细节面板可见，但不可编辑
- **元数据类型:** bool
- **引擎模块:** DetailsPanel, Editor
- **作用机制:** 在PropertyFlags中加入CPF_Edit, CPF_DisableEditOnTemplate
- **常用程度:** ★★★★☆

—并参见EditAnywhere里的示例代码和效果。

Instanced

- **功能描述:** 指定对该对象属性的编辑赋值应该新创建一个实例并作为子对象，而不是寻找一个对象引用。
- **元数据类型:** bool
- **引擎模块:** Instance
- **限制类型:** UObject*
- **作用机制:** 在PropertyFlags中加入CPF_PersistentInstance, CPF_ExportObject, CPF_InstancedReference，在Meta中加入EditInline
- **常用程度:** ★★★

指定对该对象属性的编辑赋值应该新创建一个实例并作为子对象，而不是寻找一个对象引用。

- 单个属性上的Instanced和UCLASS上的DefaultToInstanced作用有点类似，区别是前者只作用于单个属性，后者作用于该类类型的所有属性。
- 常常和EditInlineNew一起使用，在细节面板上可以为对象属性新创建实例并编辑。
- Instanced隐含了EditInline and Export.

在Object*属性上设置值的时候，如果不标Instanced，则只能为其设置一个对象引用。而如果想在编辑器里为其真正的创建一个对象实例并赋予给这个属性，则需要加上Instanced标记。但光有Instanced还不够，这个Class还需要加上EditInlineNew，才能让该类出现在可新创建类实例的列表里。

当然，在C++里手动设置对象给这个属性的话还是都可以的。也要注意和UCLASS(DefaultToInstanced)区分，DefaultToInstanced是表明这个类的所有属性都默认的加上Instanced的意思，避免了对该类的所有属性每次都要手动设置。

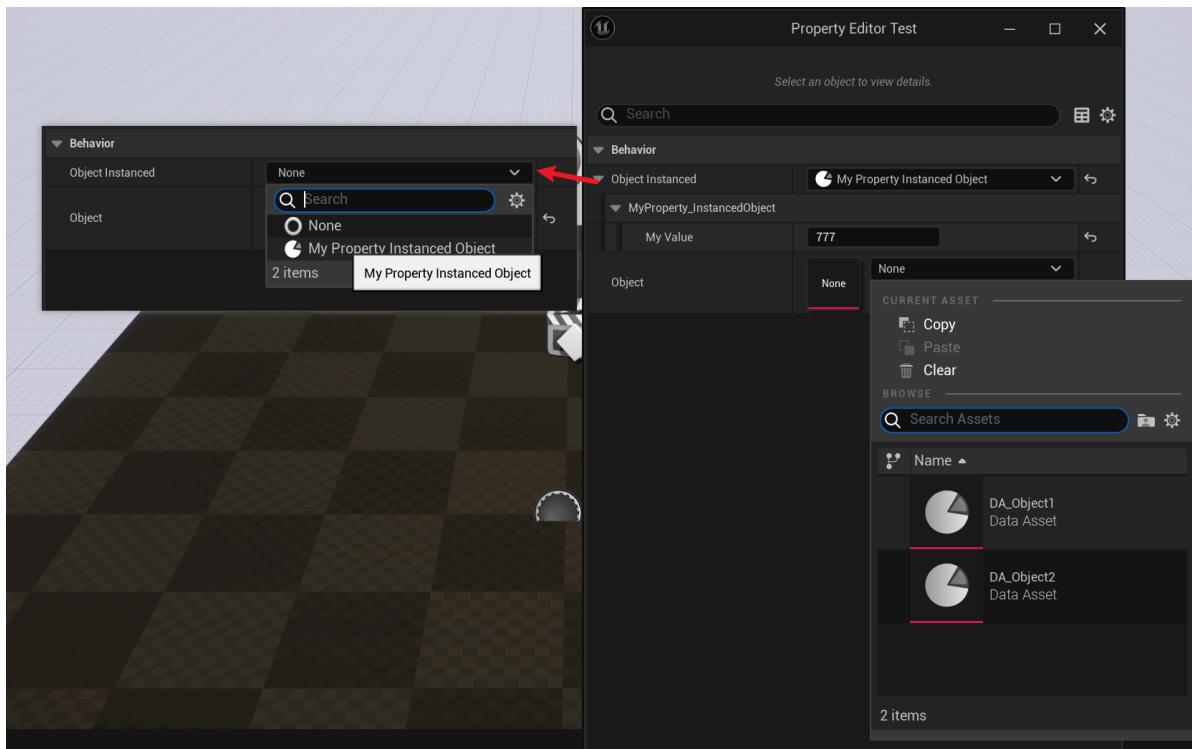
示例代码：

```
UCLASS(Blueprintable, BlueprintType, editinlinenew)
class INSIDER_API UMyProperty_InstancedObject :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyValue = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Instanced :public UObject
{
public:
    GENERATED_BODY()
    UMyProperty_Instanced(const FObjectInitializer& ObjectInitializer =
FObjectInitializer::Get());
public:
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ExportObject |
    CPF_ZeroConstructor | CPF_InstancedReference | CPF_NoDestructor |
    CPF_PersistentInstance | CPF_HasGetValueTypeHash |
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = Behavior)
        UMyProperty_InstancedObject* ObjectInstanced;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Behavior)
        UMyProperty_InstancedObject* Object;
};
```

示例效果：

可见ObjectInstanced和Object弹出的编辑框是不同的。



NotReplicated

- 功能描述:** 跳过复制。这只会应用到服务请求函数中的结构体成员和参数。
- 元数据类型:** bool
- 引擎模块:** Network
- 限制类型:** Struct members
- 作用机制:** 在PropertyFlags中加入CPF_RepSkip
- 常用程度:** ★★★

只用在结构成员中，指定struct中的某个属性不复制，否则默认就都会复制。这个用于排除掉结构中的某属性。

示例代码：

```

USTRUCT(BlueprintType)
struct FMyReplicatedStruct
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FString MyString_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NotReplicated)
        FString MyString_NotReplicated;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
        FMyReplicatedStruct MyStruct_Replicated;
};

```

```
};
```

其中MyStruct_Replicated会复制，但是其中的MyString_NotReplicated不会复制。

Replicated

- **功能描述：**指定该属性应随网络进行复制。
- **元数据类型：**bool
- **引擎模块：**Network
- **作用机制：**在PropertyFlags中加入CPF_Net
- **常用程度：**★★★★★

示例代码：

记得要在cpp代码中相应添加GetLifetimeReplicatedProps函数

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
    int32 MyInt_Replicated = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
    FMyReplicatedStruct MyStruct_Replicated;
};

void AMyProperty_Network::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(AMyProperty_Network, MyInt_Replicated);
    DOREPLIFETIME(AMyProperty_Network, MyStruct_Replicated);
}
```

示例效果就不发了，这个是基本的网络标记。

ReplicatedUsing

- **功能描述：**指定一个通知回调函数，在属性通过网络更新后执行。
- **元数据类型：**string="abc"
- **引擎模块：**Network
- **作用机制：**在PropertyFlags中加入CPF_Net, CPF_RepNotify
- **常用程度：**★★★★★

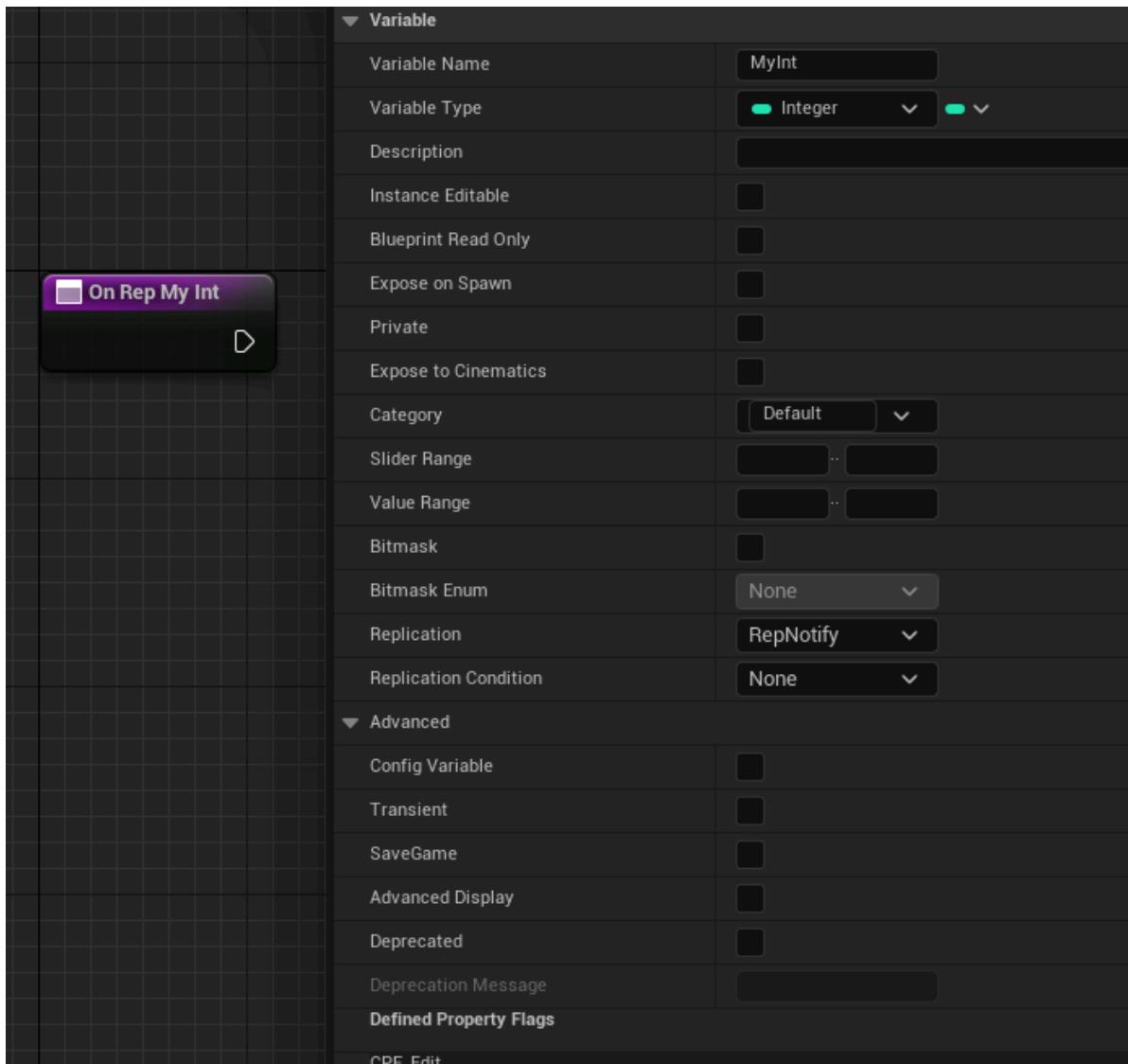
ReplicatedUsing 可以接受无参数的函数，或是带一个参数的函数携带旧值。一般在OnRep函数里，做一些开启关闭的相应操作，比如enabled的复制就会触发相应的后续逻辑。

测试代码：

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
protected:
    UFUNCTION()
        void OnRep_MyInt(int32 oldValue);
UPROPERTY(EditAnywhere, BlueprintReadWrite, ReplicatedUsing = OnRep_MyInt)
    int32 MyInt_ReplicatedUsing = 123;
};

void AMyProperty_Network::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(AMyProperty_Network, MyInt_ReplicatedUsing);
}
```

在蓝图中等价于RepNotify的作用。



RepRetry

- 功能描述:** 只适用于结构体属性。如果此属性未能完全发送（举例而言：Object引用尚无法通过网络进行序列化），则重新尝试对其的复制。对简单引用而言，这是默认选择；但对结构体而言，这会产生带宽开销，并非优选项。因此在指定此标签之前其均为禁用状态。
- 元数据类型:** bool
- 引擎模块:** Network

DuplicateTransient

- 功能描述:** 在对象复制或COPY格式导出的时候，忽略该属性。
- 元数据类型:** bool
- 引擎模块:** Serialization
- 作用机制:** 在PropertyFlags中加入CPF_DuplicateTransient
- 常用程度:** ★★

在对象复制或COPY格式导出的时候，忽略该属性。

示例代码：

```
UCLASS(Blueprintable, BlueprintType)
```

```

class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
        | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
        int32 MyInt_DuplicateTransient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
        int32 MyInt_NonPIEDuplicateTransient = 123;
};

void UMyProperty_Serialization_Test::RunTest()
{
    UMyProperty_Serialization* obj = NewObject<UMyProperty_Serialization>
    (GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_Transient = 456;
    obj->MyInt_DuplicateTransient = 456;
    obj->MyInt_NonPIEDuplicateTransient = 456;

    UMyProperty_Serialization* obj3= DuplicateObject<UMyProperty_Serialization>
    (obj,GetTransientPackage());
}

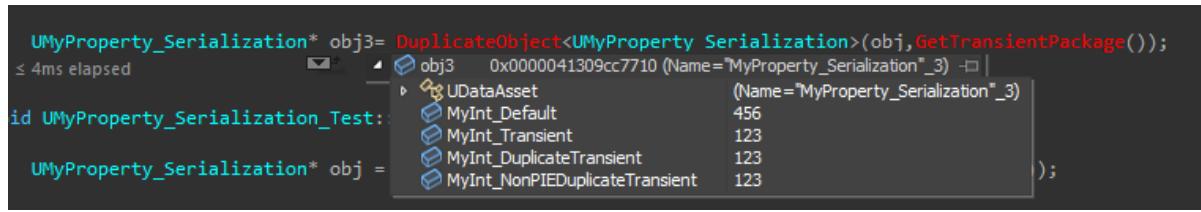
```

示例效果：

复制蓝图，可以看到DuplicateTransient并不会被复制



在采用C++复制的时候：也看到MyInt_DuplicateTransient 并不会产生复制，还是123而不是456。



原理：

在文本导出的时候，如果是T3D格式，则依然会导出。如果是COPY格式，则不导出。

```
bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyFlags(CPF_DuplicateTransient |
CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties |
PPF_IncludeTransient)))
    {
        return false;
    }
}
```

在二进制序列化的时候：

只有在PPF_Duplicate生效的时候，(DuplicateObject或者资产复制)，才会跳过该属性

```
bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
    // Skip properties marked DuplicateTransient when duplicating
    if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
PPF_Duplicate))
    {
        return false;
    }
}
```

在对资产进行Duplicate的时候，发生DuplicateAsset然后DuplicateObject，这个时候PortFlags=PPF_Duplicate，然后会触发ShouldSerializeValue进行判断。这个时候会跳过该属性

Export

- 功能描述：**在对Asset导出的时候，决定该类的对象应该导出内部的属性值，而是对象的路径。
- 元数据类型：**bool
- 引擎模块：**Serialization
- 限制类型：**Object属性，或Object数组
- 作用机制：**在PropertyFlags中加入CPF_ExportObject
- 常用程度：**★

在对Asset导出的时候，决定该类的对象应该导出内部的属性值，而是对象的路径。

- 说明Object被复制时（例如复制/粘贴操作）指定到此属性的Object应整体导出为一个子Object块（后文的例子里会看到，其实也是输出内部属性的值），而非只是输出Object引用本身。
- 只适用于Object属性（或Object数组），因为是用在对象的导出的上的。

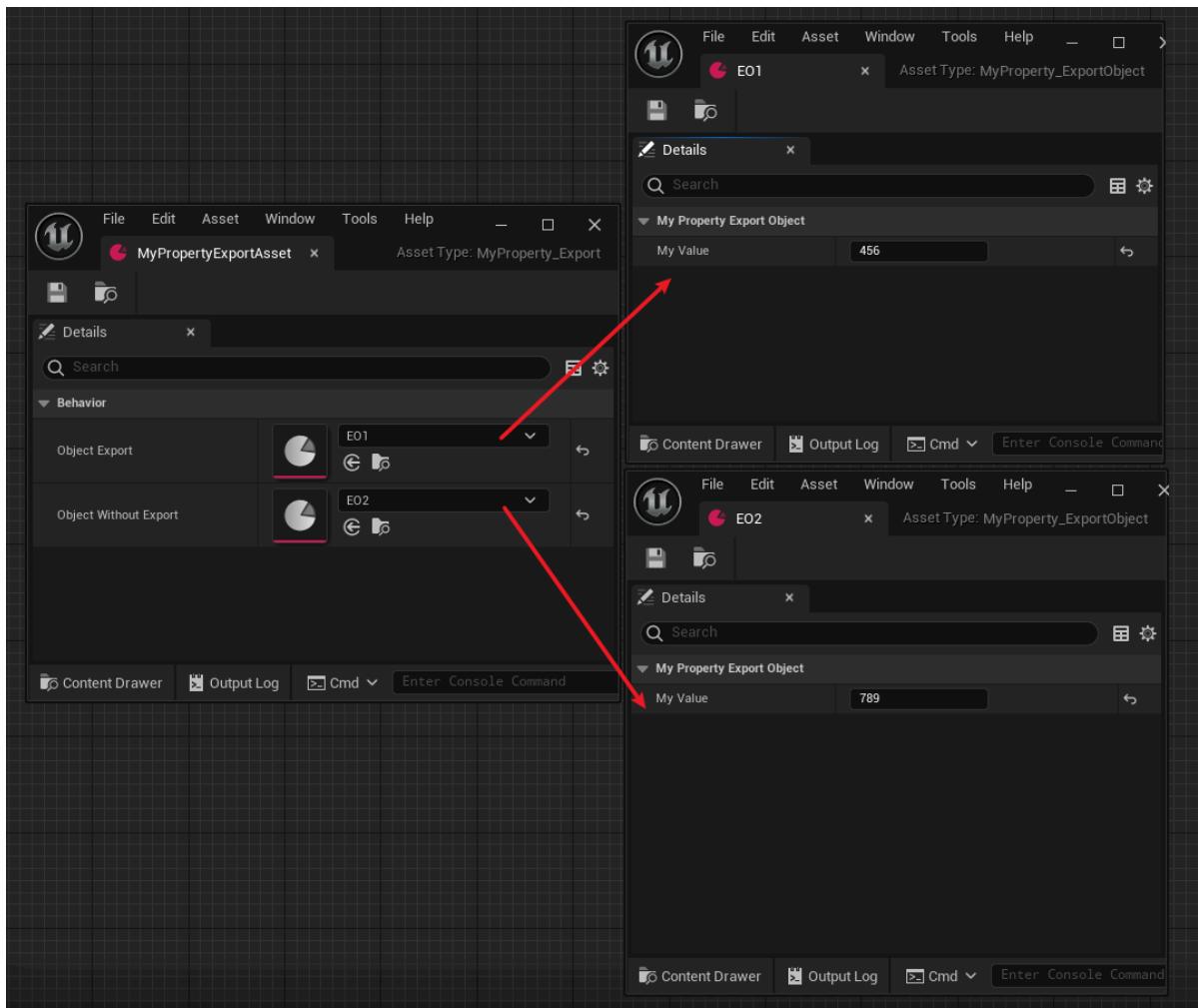
- 其实就是浅复制和深复制的区别。不标Export就是浅复制，只输出对象路径。标上Export后是深复制，也输出对象的内部属性。

示例代码：

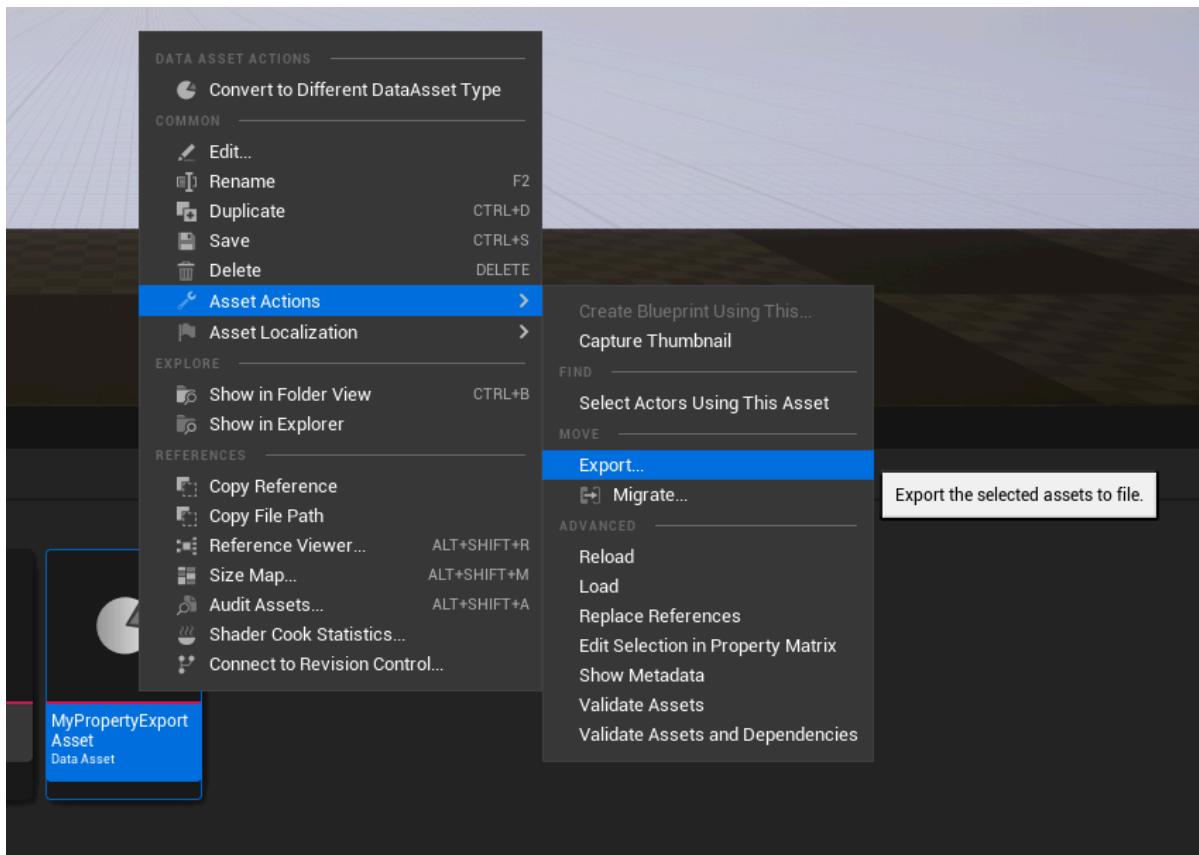
```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_ExportObject :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyValue = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Export :public UDataAsset
{
public:
public:
    GENERATED_BODY()
    UMyProperty_Export(const FObjectInitializer& ObjectInitializer =
FObjectInitializer::Get());
public:
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ExportObject | 
CPF_ZeroConstructor | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Export, Category = Behavior)
        UMyProperty_ExportObject* ObjectExport;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Behavior)
        UMyProperty_ExportObject* ObjectwithoutExport;
};
```

配置的对象值：



主要是用在Export操作的时候，用来决定如何导出Object*属性的内容。NoExport的话是只输出对象引用的路径，而Export的话会输出这个对象其再内部的的属性值。



导出的文本：

```

Begin Object Class=/Script/Insider.MyProperty_Export Name="MyPropertyExportAsset"
ExportPath=/Script/Insider.MyProperty_Export"/Game/Property/MyPropertyExportAsset.MyPropertyExportAsset"
    Begin Object Class=/Script/Insider.MyProperty_ExportObject Name="EO1"
    ExportPath=/Script/Insider.MyProperty_ExportObject"/Game/Property/EO1.EO1"
        "MyValue"=456
    End Object

    "ObjectExport"=/Script/Insider.MyProperty_ExportObject"/Game/Property/EO1.EO1"

    "ObjectWithoutExport"=/Script/Insider.MyProperty_ExportObject"/Game/Property/EO2
        .EO2"
    End Object

```

可以看到ObjectExport的对象也导出的字段值，但是ObjectWithoutExport只有路径。

原理：

源码内作用的函数，要注意一点的是，要让Export标记在ExportProperties起作用，export标记不能用在对象的sub object上，否则会走ExportInnerObjects的调用路线。上面例子中ObjectExport和ObjectWithoutExport都是指向了外部的另外一个对象，所以用DataAsset来产生资产。

```

void ExportProperties()
{
    FObjectPropertyBase* ExportObjectProp = (Property->PropertyFlags &
CPF_ExportObject) != 0 ? CastField<FObjectPropertyBase>(Property) : NULL;
}

```

NonPIEDuplicateTransient

- 功能描述：**在对象复制的时候，且在不是PIE的场合，忽略该属性。
- 元数据类型：**bool
- 引擎模块：**Serialization
- 作用机制：**在PropertyFlags中加入CPF_NonPIEDuplicateTransient
- 常用程度：**★

在对象复制的时候，且在不是PIE的场合，忽略该属性。

- DuplicateTransient和NonPIEDuplicateTransient的区别是，前者在任何情况的对象复制时都忽略该属性，而后者在PIE的时候（也是在发生对象复制过程）依然会复制该属性，其他情况下的复制和前者行为一致。
- PIE的时候本质就是把当前的编辑World里Actor复制一份到PIE的世界里，会触发Actor的复制。

示例代码：

准备了一份DataAsset和Actor来分别验证复制行为的不同。

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:

```

```

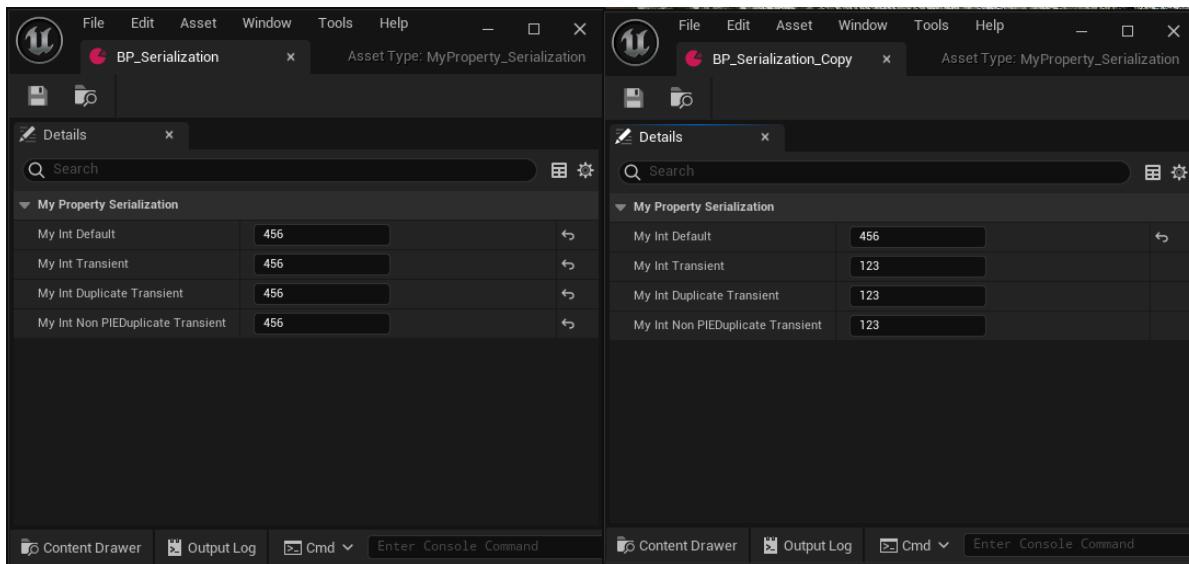
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
        | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
        int32 MyInt_DuplicateTransient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
        int32 MyInt_NonPIEDuplicateTransient = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Serialization_TestActor :public AActor
{
public:
    GENERATED_BODY()
protected:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
        | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
        int32 MyInt_DuplicateTransient = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
        int32 MyInt_NonPIEDuplicateTransient = 123;
};

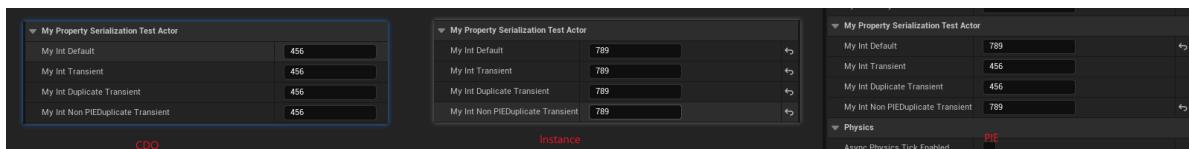
```

示例效果：

在对资产进行Duplicate的时候，发生DuplicateAsset然后DuplicateObject，这个时候PortFlags=PPF_Duplicate，然后会触发ShouldSerializeValue进行判断。这个时候会跳过该属性。
可以看到NonPIEDuplicateTransient并不会被复制。



在点击PIE的时候，可以看到NonPIEDuplicateTransient这个时候却是会复制值过去了。这是因为这个时候PortFlags=PPF_DuplicateForPIE&PPF_Duplicate



结论是用于一些Cache数据，在复制的时候并不需要序列化复制，这样可以阻止两个不同的Actor采用同一份计算后的临时数据。但是又可以在PIE的时候，让Actor各自采用自己的一份数据，因为PIE的时候，本质就是把当前的编辑World里Actor复制一份到PIE的世界里，会触发Actor的复制。

原理：

在文本导出的时候，在不是PIE里的复制的时候，不序列化该属性。

```
bool FProperty::ShouldPort( uint32 PortFlags /*=0*/ ) const
{
    // if we're not copying for PIE and NonPIETransient is set, don't export
    if (!(PortFlags & PPF_DuplicateForPIE) &&
        HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
    {
        return false;
    }
}
```

在二进制序列化的时候：

只有在PPF_Duplicate生效的时候，(DuplicateObject?或者资产复制)，才会跳过该属性。但是在PIE的时候，又要继续序列化。

```

bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
// Skip properties marked NonPIEDuplicateTransient when duplicating, but not when
we're duplicating for PIE
    if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
    {
        return false;
    }
}

```

SaveGame

- 功能描述:** 在SaveGame存档的时候，只序列化有SaveGame标记的属性，而不序列化别的属性。
- 元数据类型:** bool
- 引擎模块:** Serialization
- 常用程度:** ★★★★☆

在SaveGame存档的时候，只序列化有SaveGame标记的属性，而不序列化别的属性。

特别的标识哪些属性是用于存档保存的。

对于子结构或子对象属性，也必须要加上SaveGame标记。

NoExportTypes.h里的很多基础结构内部属性都被标上了SaveGame。

测试代码：

```

struct FMySaveGameArchive : public FObjectAndNameAsStringProxyArchive
{
    FMySaveGameArchive (FArchive& InInnerArchive)
        :   FObjectAndNameAsStringProxyArchive(InInnerArchive)
    {
        ArIsSaveGame = true;
    }
};

USTRUCT(BlueprintType)
struct FMySaveGameStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FString MyString_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        FString MyString_SaveGame;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SaveGame :public USaveGame
{
public:
    GENERATED_BODY()
public:

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
//PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor | 
CPF_SaveGame | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        int32 MyInt_SaveGame = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        FMySaveGameStruct MyStruct;
};

UMyProperty_SaveGame* UMyProperty_SaveGame_Test::LoadGameFromMemory(const
TArray<uint8>& InSaveData)
{
    FMemoryReader MemoryReader(InSaveData, true);

    FObjectAndNameAsStringProxyArchive Ar(MemoryReader, true);
    Ar.ArIsSaveGame = true;//必须手动加上这个标记

    UMyProperty_SaveGame* OutSaveGameObject = NewObject<UMyProperty_SaveGame>
(GetTransientPackage(), UMyProperty_SaveGame::StaticClass());
    OutSaveGameObject->Serialize(Ar);

    return OutSaveGameObject;
}

bool UMyProperty_SaveGame_Test::SaveGameToMemory(UMyProperty_SaveGame*
SaveGameObject, TArray<uint8>& OutSaveData)
{
    FMemoryWriter MemoryWriter(OutSaveData, true);

    // Then save the object state, replacing object refs and names with strings
    FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
    Ar.ArIsSaveGame = true;//必须手动加上这个标记
    SaveGameObject->Serialize(Ar);

    return true; // Not sure if there's a failure case here.
}

void UMyProperty_SaveGame_Test::RunTest()
{
    UMyProperty_SaveGame* saveGame = Cast<UMyProperty_SaveGame>
(UGameplayStatics::CreateSaveGameObject(UMyProperty_SaveGame::StaticClass()));
    saveGame->MyInt_Default = 456;
    saveGame->MyInt_SaveGame = 456;
    saveGame->MyStruct.MyString_Default = TEXT("Hello");
    saveGame->MyStruct.MyString_SaveGame = TEXT("Hello");

    TArray<uint8> outBytes;
    UMyProperty_SaveGame_Test::SaveGameToMemory(saveGame, outBytes);

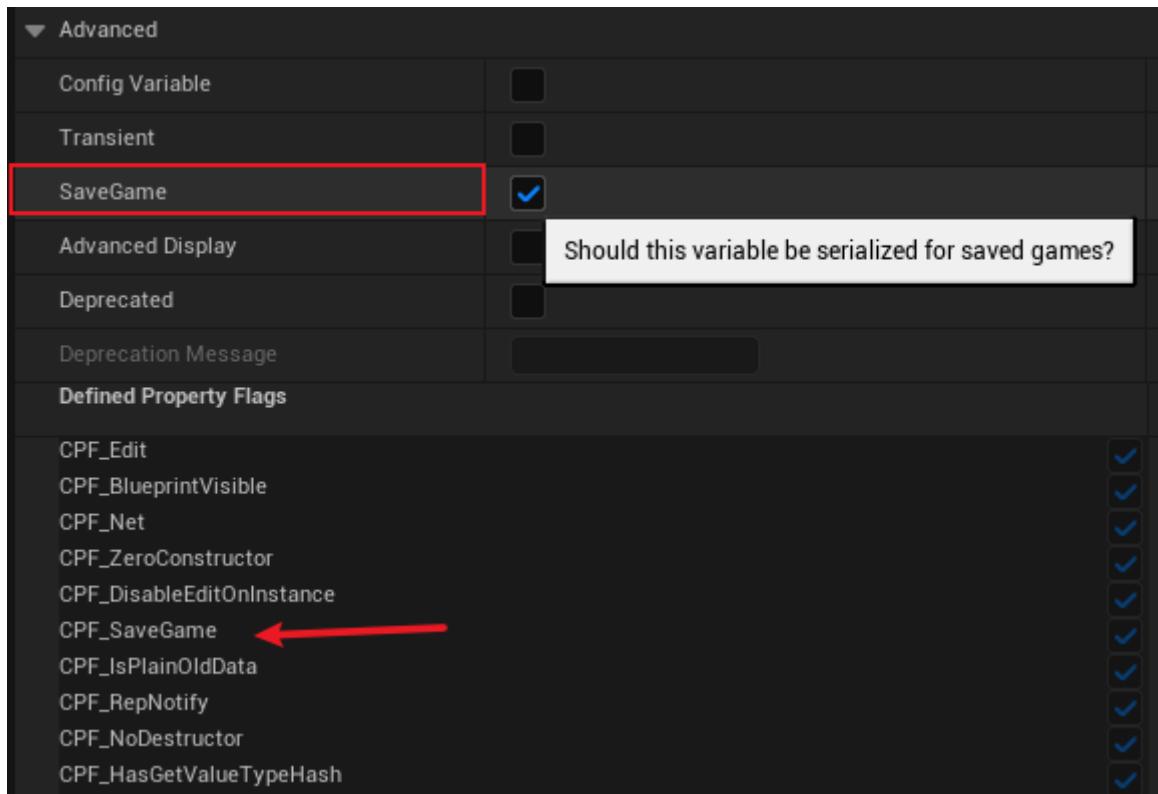
    UMyProperty_SaveGame* saveGame2 =
    UMyProperty_SaveGame_Test::LoadGameFromMemory(outBytes);
}

```

测试结果，只有SaveGame标记的属性这个值才序列化进去。

Name	Value	Type
UMyProperty_SaveGame_Test::LoadGam...	0x00000760e109cbe0 (Name="MyProperty_SaveGame"_1)	UMyProperty_SaveGa...
outBytes	Num=205, Max=285 "\xf\0\0\0MyInt_SaveGame\0\f\0\0\0int..."	TArray<unsigned char...
saveGame	0x00000760e119d2c0 (Name="MyProperty_SaveGame"_0)	UMyProperty_SaveGa...
saveGame2	0x00000760e109cbe0 (Name="MyProperty_SaveGame"_1) (Name="MyProperty_SaveGame"_1)	UMyProperty_SaveGa...
USaveGame	123	int
MyInt_Default	123	int
MyInt_SaveGame	456	int
MyStruct	{MyString_Default=Empty MyString_SaveGame=L"Hello"}	FMySaveGameStruct
MyString_Default	Empty	FString
MyString_SaveGame	L"Hello"	FString

等价于在蓝图的细节面板里表示：



原理：

只在ArIsSaveGame的时候检测这个标记，意味着这个标记只在检测USaveGame的对象的子对象结构属性的时候才用。但是ArIsSaveGame需要自己手动设置为true，否则默认这个机制是不工作的。实现的一种方式是自己手动加上一句Ar.ArIsSaveGame = true;，或者自己自定义一个FMySaveGameArchive来进行序列化。

在源码里发现UEnhancedInputUserSettings也是继承于USaveGame，采用存档的方式保存的。

```
bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
    // Skip the property if the archive says we should
    if (Ar.ShouldSkipProperty(this))
    {
        return false;
    }

    // Skip non-SaveGame properties if we're saving game state
    if (!(PropertyFlags & CPF_SaveGame) && Ar.IsSaveGame())
    {
```

```

        return false;
    }

    const uint64 SkipFlags = CPF_Transient | CPF_DuplicateTransient |
    CPF_NonPIEDuplicateTransient | CPF_NonTransactional | CPF_Deprecated |
    CPF_DevelopmentAssets | CPF_SkipSerialization;
    if (!(PropertyFlags & SkipFlags))
    {
        return true;
    }

    // Skip properties marked Transient when persisting an object, unless we're
    // saving an archetype
    if ((PropertyFlags & CPF_Transient) && Ar.IsPersistent() &&
    !Ar.IsSerializingDefaults())
    {
        return false;
    }

    // Skip properties marked DuplicateTransient when duplicating
    if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
    PPF_Duplicate))
    {
        return false;
    }

    // Skip properties marked NonPIEDuplicateTransient when duplicating, but not
    // when we're duplicating for PIE
    if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
    PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
    {
        return false;
    }

    // Skip properties marked NonTransactional when transacting
    if ((PropertyFlags & CPF_NonTransactional) && Ar.IsTransacting())
    {
        return false;
    }

    // Skip deprecated properties when saving or transacting, unless the archive
    // has explicitly requested them
    if ((PropertyFlags & CPF_Deprecated) &&
    !Ar.HasAllPortFlags(PPF_UseDeprecatedProperties) && (Ar.Issaving() ||
    Ar.IsTransacting() || Ar.WantBinaryPropertySerialization()))
    {
        return false;
    }

    // Skip properties marked SkipSerialization, unless the archive is forcing
    // them
    if ((PropertyFlags & CPF_SkipSerialization) &&
    (Ar.WantBinaryPropertySerialization() ||
    !Ar.HasAllPortFlags(PPF_ForceTaggedSerialization())))
    {
        return false;
    }
}

```

```

    }

    // Skip editor-only properties when the archive is rejecting them
    if (IsEditorOnlyProperty() && Ar.IsFilterEditorOnly())
    {
        return false;
    }

    // otherwise serialize!
    return true;
}

```

SkipSerialization

- 功能描述:** 二进制序列化时跳过该属性，但在ExportText的时候依然可以导出。
- 元数据类型:** bool
- 引擎模块:** Serialization
- 作用机制:** 在PropertyFlags中加入CPF_SkipSerialization
- 常用程度:** ★★★

在进行普通的二进制序列化的时候，这个标记会阻止序列化。作用和Transient一样。但如果是ExportText，则依然可以把该属性导出。其内部用的ExportProperties。

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SerializationText :public UObject
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
        CPF_NativeAccessSpecifierPublic | CPF_SkipSerialization
    UPROPERTY(EditAnywhere, BlueprintReadWrite,SkipSerialization)
        int32 MyInt_SkipSerialization = 123;
};

void UMyProperty_SerializationText_Test::RunTest()
{
    UMyProperty_SerializationText* obj = NewObject<UMyProperty_SerializationText>
    (GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_SkipSerialization = 456;

    //save obj
    TArray<uint8> outBytes;
    FMemoryWriter MemoryWriter(outBytes, true);
    FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
    obj->Serialize(Ar);
}

```

```

//load
FMemoryReader MemoryReader(outBytes, true);

FOBJECTANDNAMEASSTRINGPROXYARCHIVE Ar2(MemoryReader, true);

UMYPROPERTY_SERIALIZATIONTEXT* obj2 =
NewObject<UMYPROPERTY_SERIALIZATIONTEXT>(GetTransientPackage());
obj2->Serialize(Ar2);
}

```

此时可见测试结果，该属性并没有被序列化进去。

obj	0x000004b88ea5e800 (Name="MyProperty_SerializationText_0") (Name="MyProperty_SerializationText_0")	UMYPROPERTY_SERIALIZA...
UObject		UObject
MyInt_Default	456	int
MyInt_SkipSerialization	456	int
obj2	0x000004b88ea5e9c0 (Name="MyProperty_SerializationText_1") (Name="MyProperty_SerializationText_1")	UMYPROPERTY_SERIALIZA...
UObject		UObject
MyInt_Default	456	int
MyInt_SkipSerialization	123	int
outBytes	Num=60, Max=73 "\xe\0\0\0MyInt_Default\f\0\0\0IntPrope... \ View ▾ TArray<unsigned char...	

如果采用ExportText导出：T3D或COPY格式都行

```

UMYPROPERTY_SERIALIZATIONTEXT* obj = NewObject<UMYPROPERTY_SERIALIZATIONTEXT>
(GetTransientPackage());

obj->MyInt_Default = 456;
obj->MyInt_SkipSerialization = 456;

FStringOutputDevice Ar;
UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar, TEXT("T3D"), 3);

```

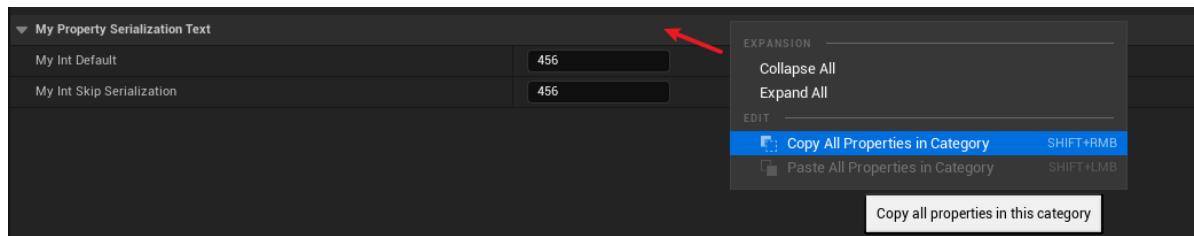
则输出结果为：

```

Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="MyProperty_SerializationText_0"
ExportPath=/Script/Insider.MyProperty_SerializationText"/Engine/Transient.MyProperty_SerializationText_0"
MyInt_Default=456
MyInt_SkipSerialization=456
End Object

```

另外如果在编辑器里右击复制



也可以产生文本的导出：

```

{
    "Tagged": [
        [
            "MyInt_Default",
            "456"
        ],
        [
            "MyInt_SkipSerialization",
            "456"
        ]
    ]
}

```

原理：

注意在判断一个Property是否应该序列化的时候，ShouldSerializeValue函数是用在普通的序列化的时候用来判断的。而在ExportText的时候，是用ShouldPort判断的。

TextExportTransient

- 功能描述：** 在ExportText导出为.COPY格式的时候，忽略该属性。
- 元数据类型：** bool
- 引擎模块：** Serialization
- 作用机制：** 在PropertyFlags中加入CPF_TextExportTransient
- 常用程度：** ★

在ExportText导出为.COPY格式的时候，忽略该属性。

但鼠标复制拷贝属性依然会有文本导出生效。

测试代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SerializationText :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
    CPF_NativeAccessSpecifierPublic | CPF_SkipSerialization
    UPROPERTY(EditAnywhere, BlueprintReadWrite,SkipSerialization)
    int32 MyInt_SkipSerialization = 123;
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_IsPlainOldData | CPF_NoDestructor | CPF_TextExportTransient |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite,TextExportTransient)
    int32 MyInt_TextExportTransient = 123;

};

```

```

void UMyProperty_SerializationText_Test::RunExportTest()
{
    UMyProperty_SerializationText* obj = NewObject<UMyProperty_SerializationText>
    (GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_SkipSerialization = 456;
    obj->MyInt_TextExportTransient = 456;

    FStringOutputDevice Ar;
    UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar, TEXT("T3D"), 3);

    FStringOutputDevice Ar2;
    UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar2, TEXT("COPY"), 3);

    FString str=Ar;
}

```

导出的结果：

```

T3D格式:
Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="BP_SerializationText"
ExportPath="/Script/Insider.MyProperty_SerializationText'/Game/Property/BP_SerializationText.BP_SerializationText'"
    MyInt_Default=456
    MyInt_SkipSerialization=456
    MyInt_TextExportTransient=456
End Object

COPY格式:
Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="BP_SerializationText"
ExportPath="/Script/Insider.MyProperty_SerializationText'/Game/Property/BP_SerializationText.BP_SerializationText'"
    MyInt_Default=456
    MyInt_SkipSerialization=456
End Object

```

复制拷贝依然会有文本生效：

```

{
    "Tagged": [
        [
            "MyInt_Default",
            "456"
        ],
        [
            "MyInt_SkipSerialization",
            "456"
        ],
        [
            "MyInt_TextExportTransient",

```

```

        "456"
    ]
}

```

因此可以发现在COPY格式的时候，MyInt_TextExportTransient并没有被导出。

原理：

注意在判断一个Property是否应该序列化的时候，ShouldSerializeValue函数是用在普通的序列化的时候用来判断的。而在ExportText的时候，是用ShouldPort判断的。

但是如果序列化出的格式是COPY，在设置PortFlags的时候，会额外的加上PPF_Copy。因此在后续的判断里才会生效对CPF_TextExportTransient的判断。

```

if ( FCString::Stricmp(FileType, TEXT("COPY")) == 0 )
{
    // some code which doesn't have access to the exporter's file type needs
    // to handle copy/paste differently than exporting to file,
    // so set the export flag accordingly
    PortFlags |= PPF_Copy;
}

//
// Return whether the property should be exported.
//
bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if no size, don't export
    if (GetSize() <= 0)
    {
        return false;
    }

    if (HasAnyPropertyFlags(CPF_Deprecated) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_UseDeprecatedProperties)))
    {
        return false;
    }

    // if we're parsing default properties or the user indicated that transient
    // properties should be included
    if (HasAnyPropertyFlags(CPF_Transient) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyFlags(CPF_DuplicateTransient |
CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties |
PPF_IncludeTransient)))
    {
        return false;
    }
}

```

```

// if we're not copying for PIE and NonPIETransient is set, don't export
if (!(PortFlags & PPF_DuplicateForPIE) &&
HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
{
    return false;
}

// if we're only supposed to export components and this isn't a component
property, don't export
if ((PortFlags & PPF_SubobjectsOnly) && !ContainsInstancedObjectProperty())
{
    return false;
}

// hide non-Edit properties when we're exporting for the property window
if ((PortFlags & PPF_Propertywindow) && !(PropertyFlags & CPF_Edit))
{
    return false;
}

return true;
}

```

Transient

- 功能描述:** 不序列化该属性，该属性初始化时候会被0填充。
- 元数据类型:** bool
- 引擎模块:** Serialization
- 作用机制:** 在PropertyFlags中加入CPF_Transient
- 常用程度:** ★★★★☆

序列化的时候略过该属性，用0来填充默认值。

二进制和文本都不序列化该属性。

一般用于一些临时中间变量或计算后的结果变量。

示例代码：

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
        | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient = 123;

```

```

//PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
int32 MyInt_DuplicateTransient = 123;
//PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_IsPlainOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
int32 MyInt_NonPIEDuplicateTransient = 123;
};

void UMyProperty_Serialization_Test::RunTest()
{
    UMyProperty_Serialization* obj = NewObject<UMyProperty_Serialization>
(GetTransientPackage());

obj->MyInt_Default = 456;
obj->MyInt_Transient = 456;
obj->MyInt_DuplicateTransient = 456;
obj->MyInt_NonPIEDuplicateTransient = 456;

//save obj
TArray<uint8> outBytes;
FMemoryWriter MemoryWriter(outBytes, true);
FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
obj->Serialize(Ar);

//load
FMemoryReader MemoryReader(outBytes, true);

FObjectAndNameAsStringProxyArchive Ar2(MemoryReader, true);

UMyProperty_Serialization* obj2 = NewObject<UMyProperty_Serialization>
(GetTransientPackage());
obj2->Serialize(Ar2);
}

```

对这么一个BP DataAsset进行AssetActions→Export,

T3D格式:

```

Begin Object Class=/Script/Insider.MyProperty_Serialization
Name="BP_Serialization"
ExportPath="/Script/Insider.MyProperty_Serialization'/Game/Property/BP_Serialization.BP_Serialization"
MyInt_Default=456
MyInt_DuplicateTransient=456
End Object

```

COPY格式:

```

Begin Object Class=/Script/Insider.MyProperty_Serialization
Name="BP_Serialization"
ExportPath="/Script/Insider.MyProperty_Serialization'/Game/Property/BP_Serialization'BP_Serialization'"
    MyInt_Default=456
End Object

```

如果是普通的序列化：

可见obj2的MyInt_Transient 属性并没有从序列化中获得新值456.

obj	0x000005965862a820 (Name="MyProperty_Serialization"_1)	UMyProperty_Serializa...
UDataSet	(Name="MyProperty_Serialization"_1)	UDataSet
MyInt_Default	456	int
MyInt_Transient	456	int
MyInt_DuplicateTransient	456	int
MyInt_NonPIEDuplicateTransient	456	int
obj2	0x000005965862a910 (Name="MyProperty_Serialization"_2)	UMyProperty_Serializa...
UDataSet	(Name="MyProperty_Serialization"_2)	UDataSet
MyInt_Default	456	int
MyInt_Transient	123	int
MyInt_DuplicateTransient	456	int
MyInt_NonPIEDuplicateTransient	456	int
outBytes	Num=182, Max=208 "xe\0\0\0MyInt_Default\0\f\0\0\0IntPro..."	TArray<unsigned char...

原理代码：

判断CPF_Transient的生效，只有在IsPersistent()的时候，并且不是在保存CDO。SetIsPersistent()的调用在很多地方都出现，比如在MemoryReader/MemoryWriter都是IsPersistent。

因此Transient是在序列化的时候会被忽略。

在ExportText的时候发现会进行CPF_Transient的判断，除非强制进行包括PPF_IncludeTransient

```

bool FProperty::shouldSerializeValue(FArchive& Ar) const
{
    // skip the property if the archive says we should
    if (Ar.ShouldSkipProperty(this))
    {
        return false;
    }

    // skip non-SaveGame properties if we're saving game state
    if (!(PropertyFlags & CPF_SaveGame) && Ar.IsSaveGame())
    {
        return false;
    }

    const uint64 SkipFlags = CPF_Transient | CPF_DuplicateTransient |
    CPF_NonPIEDuplicateTransient | CPF_NonTransactional | CPF_Deprecated |
    CPF_DevelopmentAssets | CPF_SkipSerialization;
    if (!(PropertyFlags & SkipFlags))
    {
        return true;
    }

    // skip properties marked Transient when persisting an object, unless we're
    saving an archetype
}

```

```

        if ((PropertyFlags & CPF_Transient) && Ar.IsPersistent() &&
!Ar.IsSerializingDefaults())
{
    return false;
}

// Skip properties marked DuplicateTransient when duplicating
if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
PPF_Duplicate))
{
    return false;
}

// Skip properties marked NonPIEDuplicateTransient when duplicating, but not
when we're duplicating for PIE
if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
{
    return false;
}

// Skip properties marked NonTransactional when transacting
if ((PropertyFlags & CPF_NonTransactional) && Ar.IsTransacting())
{
    return false;
}

// Skip deprecated properties when saving or transacting, unless the archive
has explicitly requested them
if ((PropertyFlags & CPF_Deprecated) &&
!Ar.HasAllPortFlags(PPF_UseDeprecatedProperties) && (Ar.Issaving() ||
Ar.IsTransacting() || Ar.WantBinaryPropertySerialization()))
{
    return false;
}

// Skip properties marked SkipSerialization, unless the archive is forcing
them
if ((PropertyFlags & CPF_SkipSerialization) &&
(Ar.WantBinaryPropertySerialization() ||
!Ar.HasAllPortFlags(PPF_ForceTaggedSerialization())))
{
    return false;
}

// Skip editor-only properties when the archive is rejecting them
if (IsEditorOnlyProperty() && Ar.IsFilterEditorOnly())
{
    return false;
}

// otherwise serialize!
return true;
}

```

```

///////////////////////////////
bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if no size, don't export
    if (GetSize() <= 0)
    {
        return false;
    }

    if (HasAnyPropertyFlags(CPF_Deprecated) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_UseDeprecatedProperties)))
    {
        return false;
    }

    // if we're parsing default properties or the user indicated that transient
    // properties should be included
    if (HasAnyPropertyFlags(CPF_Transient) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyFlags(CPF_DuplicateTransient | 
CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties | 
PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're not copying for PIE and NonPIETransient is set, don't export
    if (!(PortFlags & PPF_DuplicateForPIE) &&
HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
    {
        return false;
    }

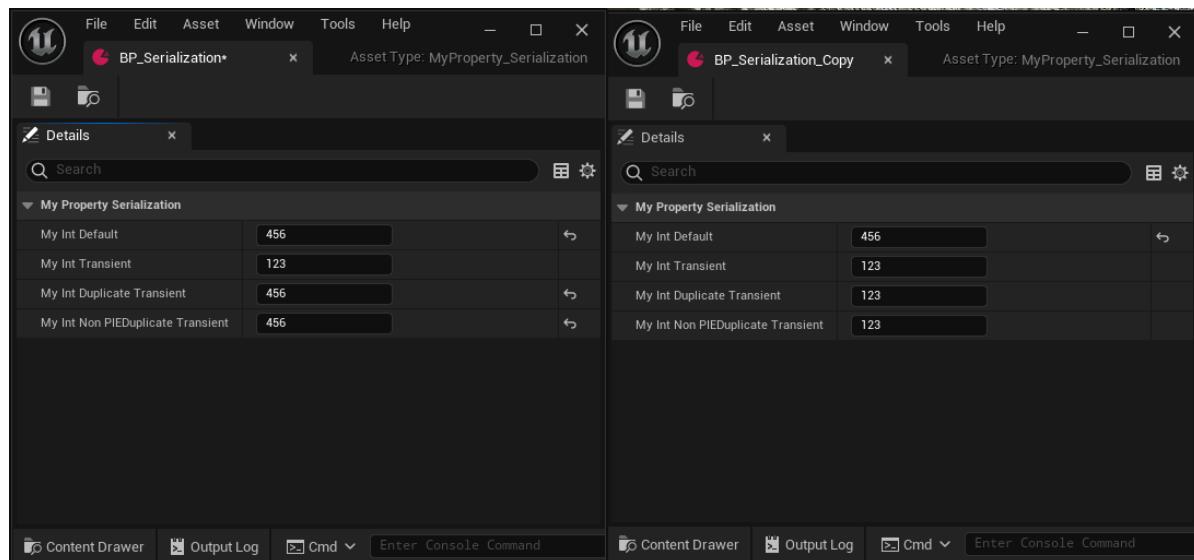
    // if we're only supposed to export components and this isn't a component
    // property, don't export
    if ((PortFlags & PPF_SubobjectsOnly) && !ContainsInstancedObjectProperty())
    {
        return false;
    }

    // hide non-Edit properties when we're exporting for the property window
    if ((PortFlags & PPF_Propertywindow) && !(PropertyFlags & CPF_Edit))
    {
        return false;
    }

    return true;
}

```

因为不序列化Transient属性，因此该属性修改值也并不会被保存起来。打开Asset的时候依然会是默认值，也并不会被复制。



FieldNotify

- 功能描述：**在打开MVVM插件后，使得该属性变成支持FieldNotify的属性。
- 元数据类型：** bool
- 引擎模块：** MVVM, UHT
- 限制类型：** ViewModel里的属性
- 常用程度：** ★★★★

在打开MVVM插件后，使得该属性变成支持FieldNotify的属性。

测试代码：

```
UCLASS(BlueprintType)
class INSIDER_API UMyViewModel : public UMVVMViewModelBase
{
    GENERATED_BODY()
protected:
    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetHP)
        float HP = 1.f;

    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetMaxHP)
        float MaxHP = 100.f;
public:
    float GetHP() const { return HP; }
    UFUNCTION(BlueprintSetter)
    void SetHP(float val)
    {
        if (UE_MVVM_SET_PROPERTY_VALUE(HP, val))
        {
            UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
        }
    }
}
```

```

float GetMaxHP() const { return MaxHP; }
UFUNCTION(BlueprintSetter)
void SetMaxHP(float val)
{
    if (UE_MVVM_SET_PROPERTY_VALUE(MaxHP, val))
    {
        UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
    }
}

//You need to manually notify that GetHealthPercent changed when
CurrentHealth or MaxHealth changed.
UFUNCTION(BlueprintPure, FieldNotify)
float GetHPPPercent() const
{
    return (MaxHP != 0.f) ? HP / MaxHP : 0.f;
}
;
```

测试效果：

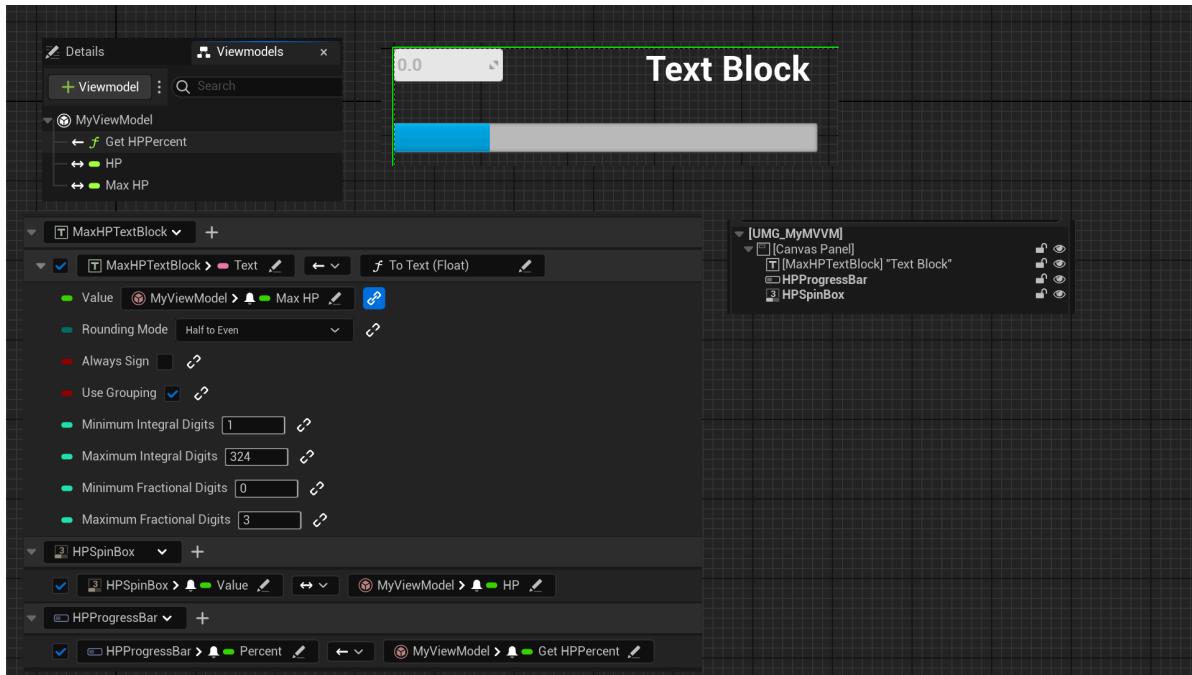
效果一方面是UHT生成的.generated.h里和.gen.cpp里的代码，具体的宏作用就不展开了，大家可查看其他更细致的MVVM相关文章。这里只要知道UHT会为标了FieldNotify的属性定义FFieldId以及FFieldNotificationClassDescriptor来标识一个接受可通知的属性。

```

//MyViewModel.generated.h
#define
FID_GitWorkspace_Hello_Source_Insider_Property_MVVM_MyViewModel_h_12_FIELDNOTIFY \
\
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BEGIN(INSIDER_API ) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(HP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(HP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END() \
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_END(); \
\
//MyViewModel.gen.cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_BEGIN(UMyViewModel)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_END(UMyViewModel);
```

蓝图效果：

这些控件的属性就可以和ViewModel里的属性绑定起来。



Native

- **功能描述:** 属性为本地: C++代码负责对其进行序列化并公开到垃圾回收。
- **元数据类型:** bool
- **引擎模块:** Behavior

已被删除