

数据资产配置--组件

配置了所需要的各种功能组件，

目前逻辑是在角色捡到武器，会遍历这个资产的所有组件配置把他添加到角色上

Components to Initialize	12 数组元素
索引 [0]	BP_TempestTickComponent
索引 [1]	BP_TempestStateManagerComponent
索引 [2]	BP_TempestBaseCombatComponent
索引 [3]	BP_TempestAbilitySystemComponent
索引 [4]	BP_TempestAttributesComponents
索引 [5]	BP_TempestCollisionManager
索引 [6]	BP_TempestFeelComponent
索引 [7]	BP_TempestTargetingComponent
索引 [8]	BP_TempestPropertiesComponent
索引 [9]	BP_TempestCameraModeComponent
索引 [10]	BP_TempestStatisticsComponent
索引 [11]	BP_TempestInputComponent

BP_TempestStateManagerComponent

基类为C++中的UTempestBaseStateManagerComponent，主要负责状态的管理，类似于Unity中的状态机，但这里面只会记录当前状态，具体的实现则需要能力（Ability）驱动

BP_TempestAttributesComponents

基类为C++中的UTempestAttributesComponents，负责角色的基础属性，详情看
[UTempestAttributesComponents](#)

BP_TempestPropertiesComponent

基类为C++中的UTempestPropertiesComponent，负责角色所拥有的特性，详情看
[UTempestPropertiesComponent](#)

数据资产配置--特性 (GeneralProperty)

允许的状态输入特性

通过输入来驱动状态和能力

The screenshot shows the Unreal Engine's Properties panel for a 'BP States Allowed Inputs Property'. The panel is organized into sections:

- 索引 [0]**: A dropdown menu showing '1 个成员' (1 member) and a button to add a new element.
- Special Property**: A dropdown menu showing 'BP States Allowed Inputs Property'.
- 默认**: A section for default values.
- States & Inputs**: A section containing 11 array elements, each labeled '索引 [0]' through '索引 [10]'. Each element has a dropdown menu showing '2 个成员' (2 members).
- Special Property Base Variables**: A section for base variables.
- Property Tag**: A dropdown menu showing 'Property.Special_STATES & ALLOWED INPUTS'.

玩家速度特性

The screenshot shows the Unreal Engine's Properties panel for a 'BP Player Speeds Property'. The panel is organized into sections:

- 索引 [1]**: A dropdown menu showing '1 个成员' (1 member) and a button to add a new element.
- Special Property**: A dropdown menu showing 'BP Player Speeds Property'.
- 默认**: A section for default values, containing three speed settings:
 - Walking Movement Speed: 500.0
 - Sprinting Movement Speed: 700.0
 - Blocking Movement Speed: 200.0
- Special Property Base Variables**: A section for base variables.
- Property Tag**: A dropdown menu showing 'Property.Special_Speeds'.

特殊能力的蒙太奇配置

因为走的重定向逻辑，像基础的走跑跳就不用在配置了，但有些特殊动画比如死亡攻击之类的就需要特殊的蒙太奇，而且可能每个武器的这些动作都不一样都需要配置

▼ 紴引 [4]	1 个成员
▼ Special Property	BP Montages Per Ability Property
▼ 默认	
▼ Montages List Per Ability	9 贴图元素
▶ BP_PlayerNormalDeathAbility	1 个成员
▶ BP_PlayerNormalHitAbility	1 个成员
▶ BP_PlayerLightAttackAbility	1 个成员
▶ BP_CharacterEquipAbility	1 个成员
▶ BP_PlayerUnEquipAbility	1 个成员
▶ BP_PlayerBlockStartAbility	1 个成员
▶ BP_PlayerBlockEndAbility	1 个成员
▶ BP_CharacterNormalBlockHitAbility	1 个成员
▶ BP_PlayerSpecialAttackAbility	1 个成员
▼ Special Property Base Variables	
Property Tag	Property.Special.Montages Per Ability

状态和能力特性

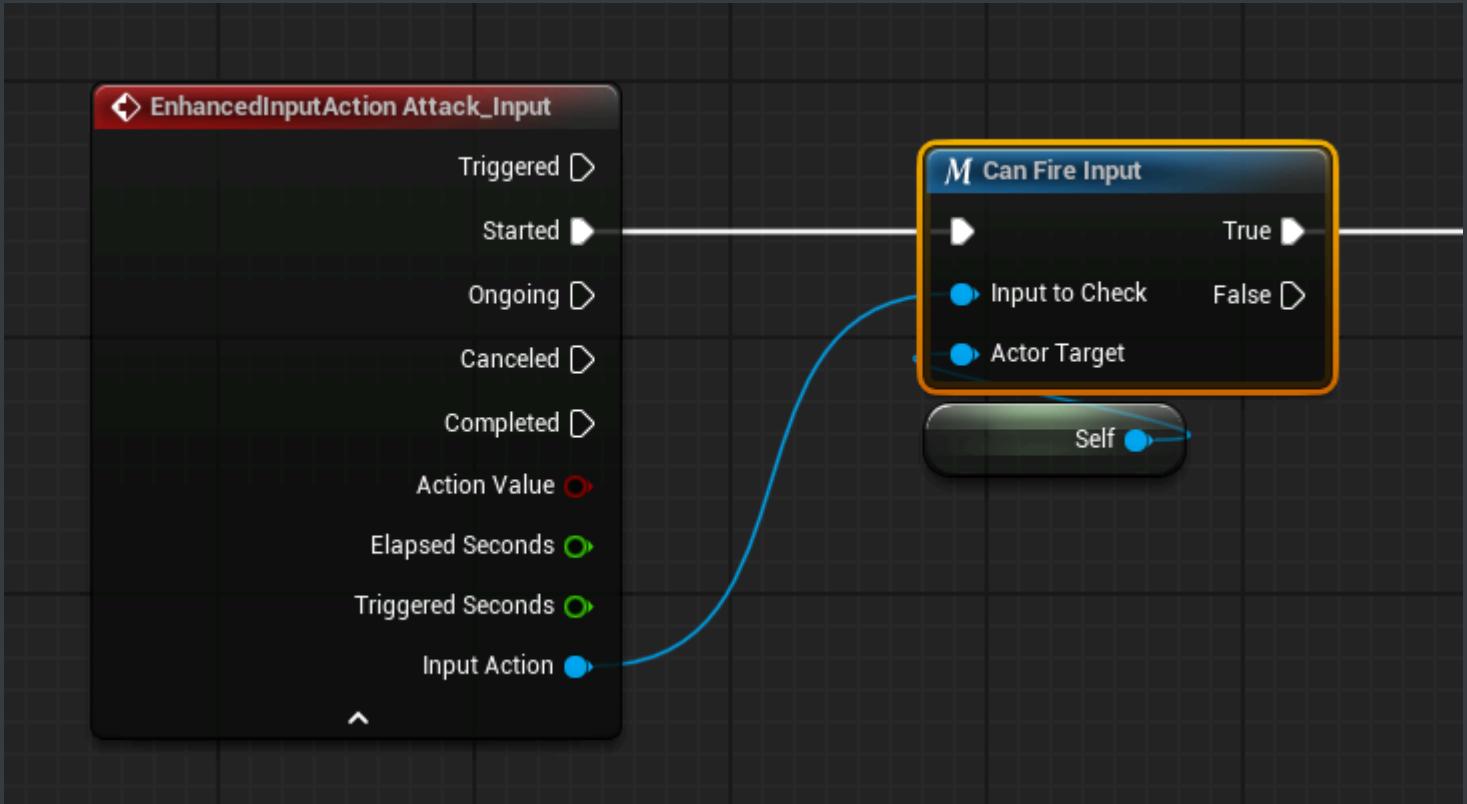
每一个这个武器或者角色可能拥有的状态都配置出来

▼ 紹引 [5]	1 个成员
▼ Special Property	BP States and Abilities Property
▼ 默认	
▼ Abilities Per State	12 贴图元素
▶ BP_CharacterIdleState	1 个成员
▶ BP_PlayerWalkingState	1 个成员
▶ BP_PlayerJumpingState	1 个成员
▶ BP_PlayerSprintingState	1 个成员
▶ BP_PlayerFallingState	1 个成员
▶ BP_PlayerDeathState	1 个成员
▶ BP_CharacterHitState	1 个成员
▶ BP_CharacterEquipState	1 个成员
▶ BP_CharacterAttackingState	1 个成员
▶ BP_CharacterUnEquipState	1 个成员
▶ BP_PlayerBlockState	1 个成员
▶ BP_CharacterBlockHitState	1 个成员
▼ Special Property Base Variables	
Property Tag	Property.Special.States & Abilities
Defense Property	BP_PlayerDefenseProperty

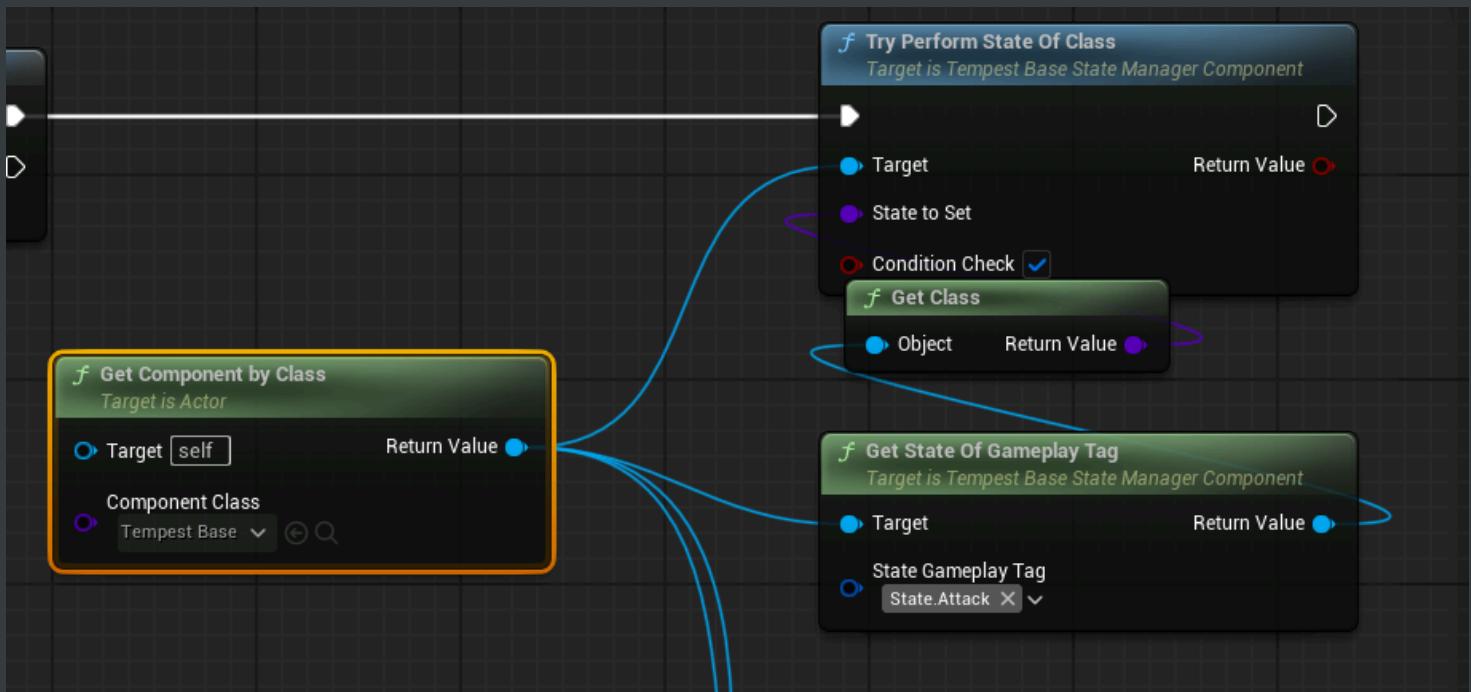
一个攻击的流程示意(将这几个组件串起来):

输入驱动状态,状态调用能力,一个状态可以拥有多种能力

在玩家的基类BP_ThirdPersonCharacterBasic中调用攻击,具体能不能广播事件见UTempestBaseInputComponent脚本,



之后将状态设置为攻击状态



这里解释一下切换到攻击状态的原理(后续会移动到UTempestBaseStateManagerComponent这个组件的讲解):

每个状态类的基类UTempestBaseStateManagerComponent

有个构造方法

使用 NewObject 动态创建一个新的状态对象实例

将新创建的状态对象添加到可激活状态列表(ActivatableStates)中

设置状态对象的执行者为当前组件的拥有者

这样状态对象知道是哪个Actor在执行它

```
void UTempestBaseStateManagerComponent::ConstructStateOfClass(TSubclassOf<UTempestBaseStateObject> StateToConstruct)
{
    ConstructedState = nullptr;
    if (StateToConstruct)
    {
        UTempestBaseStateObject* LocalNewState;
        LocalNewState = NewObject<UTempestBaseStateObject>(Outer: GetOwner(), StateToConstruct);

        ActivatableStates.AddUnique(LocalNewState);
        LocalNewState->SetPerformingActor(GetOwner());
    }
}
```

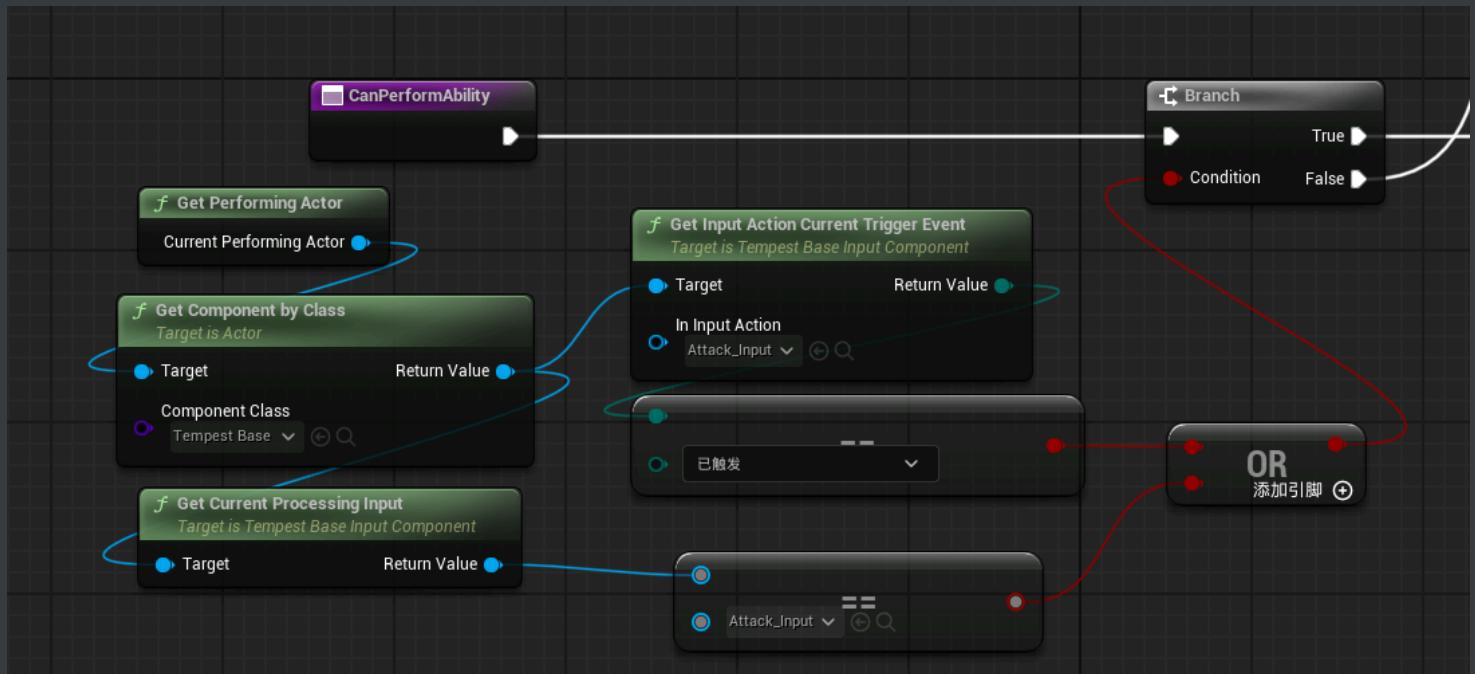
然后再 TryPerformStateOfClass 尝试执行新的状态的时候需要判断一下条件

```
bool UTempestBaseStateManagerComponent::TryPerformStateOfClass(TSubclassOf<UTempestBaseStateObject> StateToSet, bool bForce)
{
    if (StateToSet)
    {
        UTempestBaseStateObject* LocalState = nullptr;
        GetStateOfClass(StateToSet, &LocalState);

        if (LocalState)
        {
            if (ConditionCheck)
            {
                if (LocalState->CanPerformState())
                {
                    LocalState->PreStateActivation();
                    LocalState->StartState();
                    LocalState->PostStateActivation();
                    return true;
                }
            }
        }
    }
}
```

比如这个状态被初始化构造了,以及切换这个状态是否需要判断是否可以转化:判断是否可以转化的逻辑在BP_PlayerLightAttackAbility蓝图中编写,

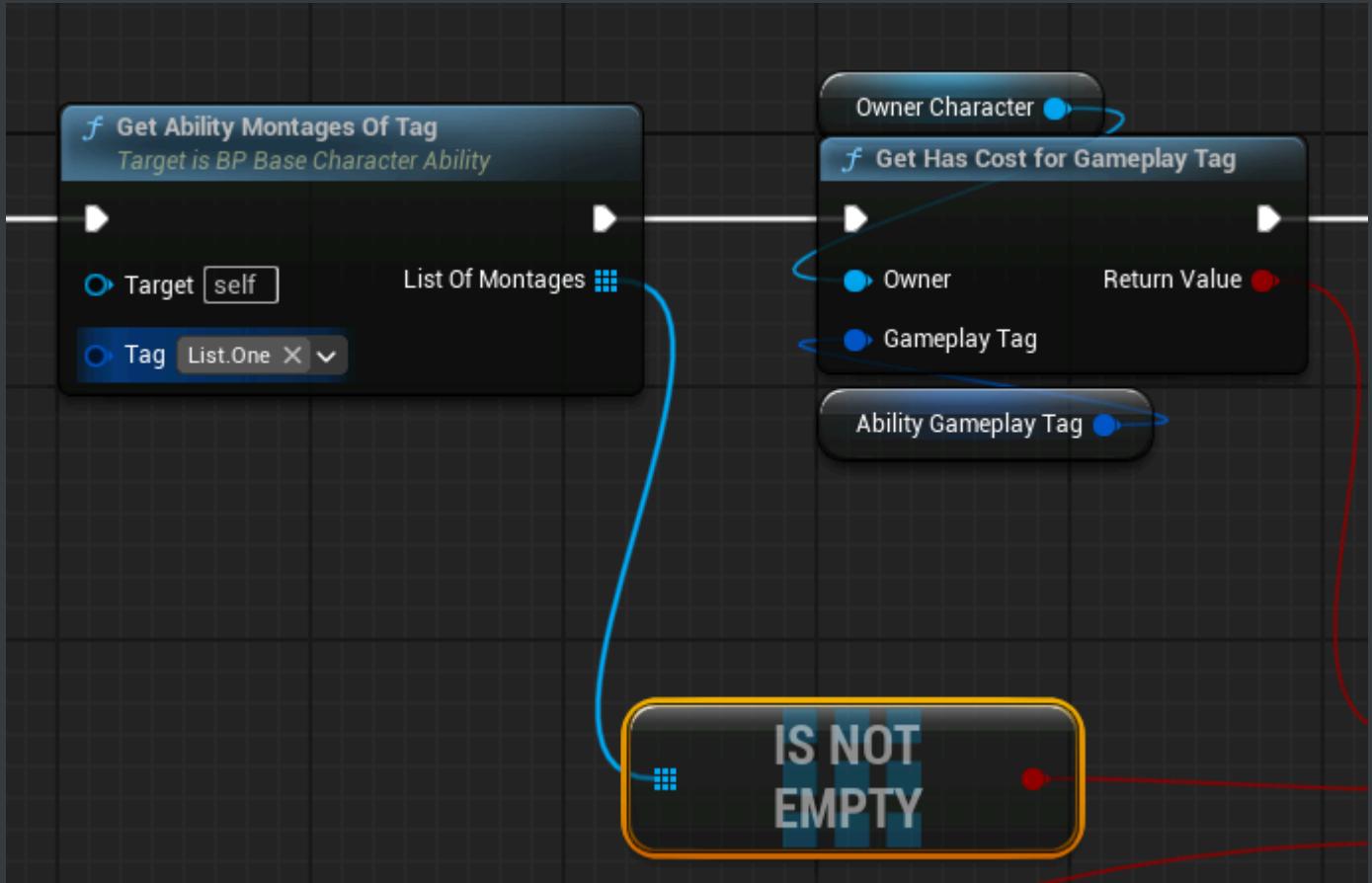
首先判断条件,是否触发了攻击按键



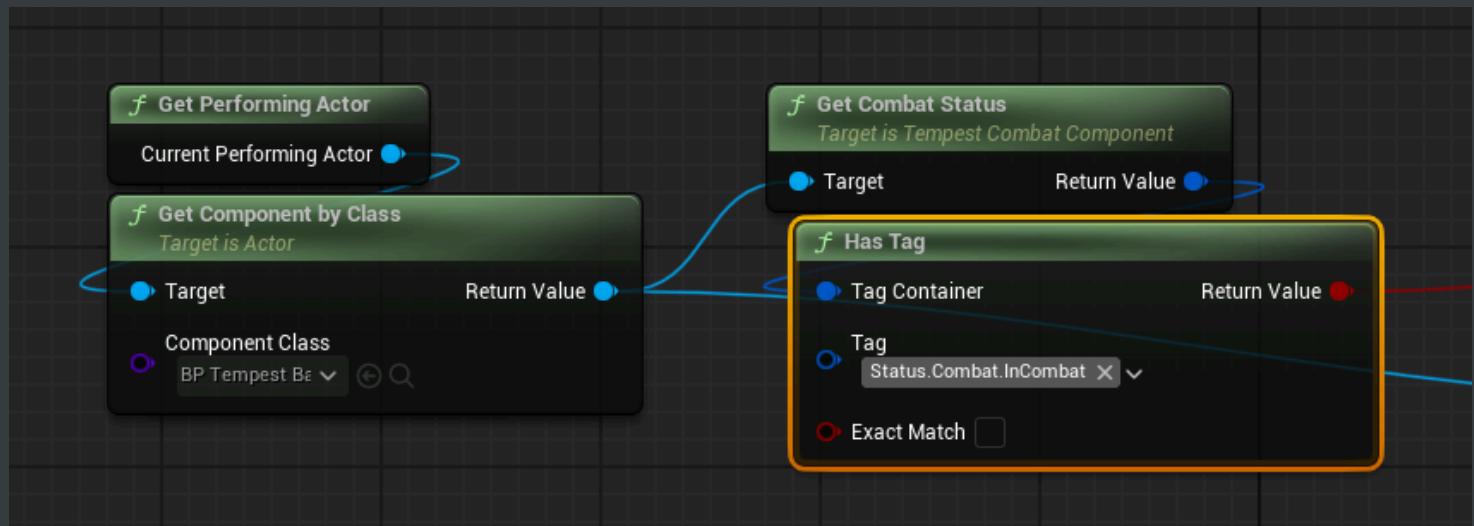
之后需要判断

1. 获取攻击动画,然后判断动画是否为空

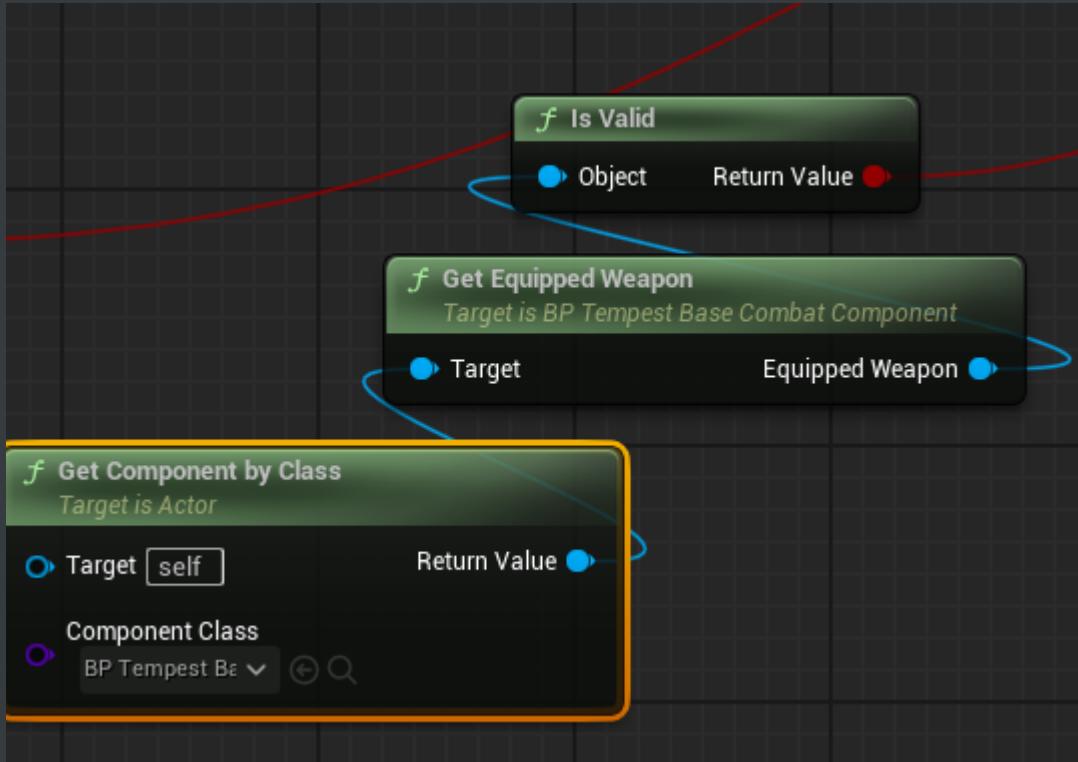
2/然后判断是否有足够的消耗值



3.判断是否处于战斗状态



4.是否持有武器



然后这四个条件都通过则可以通过判断条件来执行转换状态的逻辑:

```

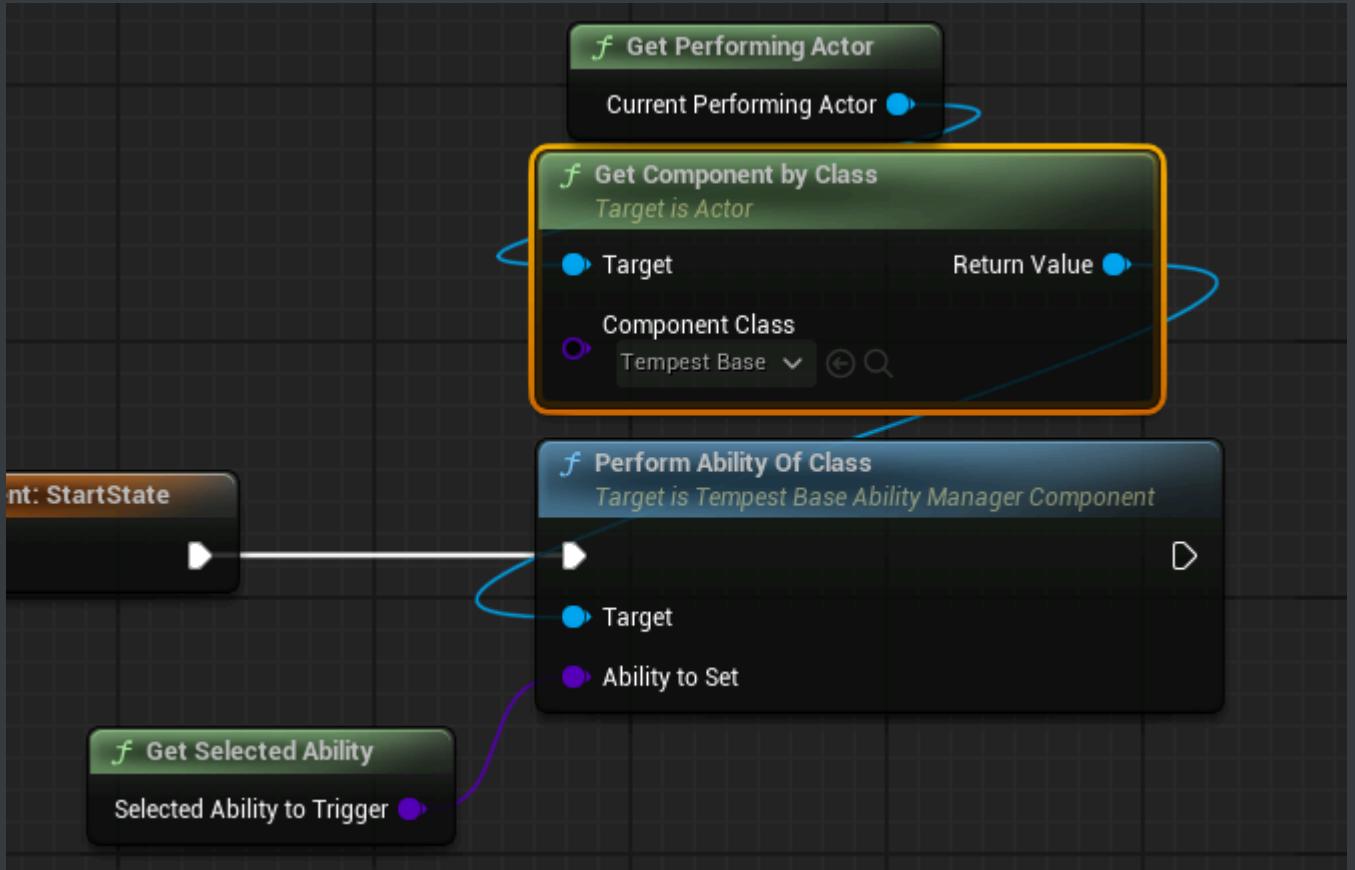
LocalState->PreStateActivation();
LocalState->StartState();
LocalState->PostStateActivation();
return true;

```

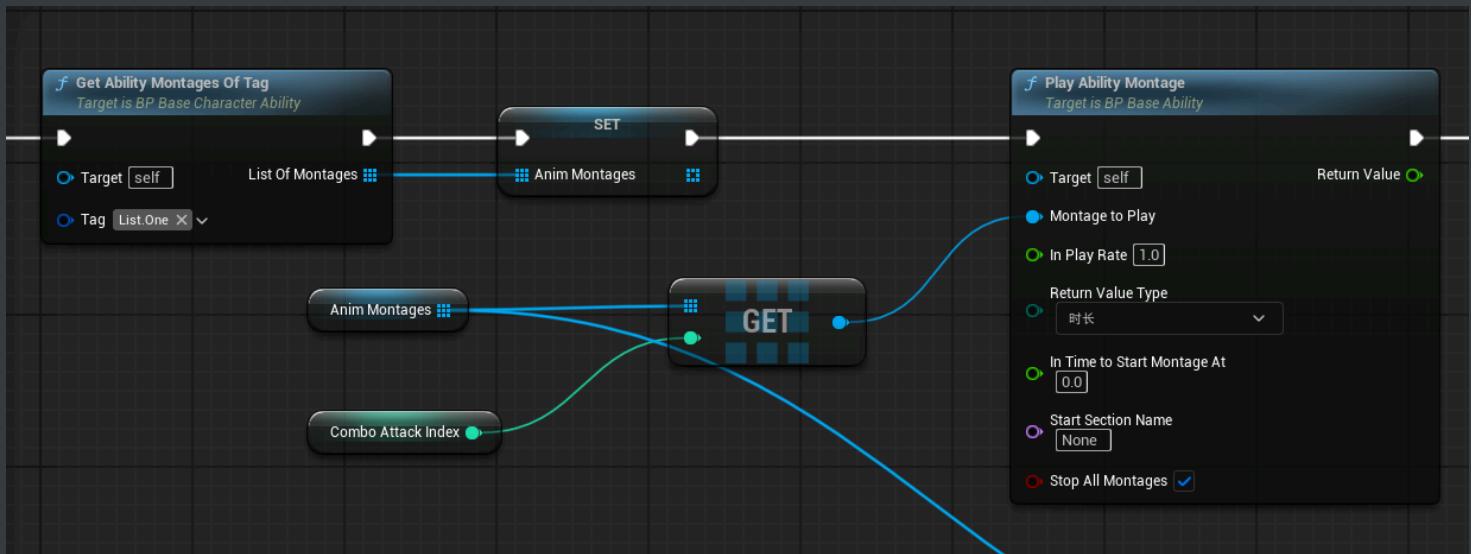
执行该状态激活前状态 -- 开始状态 -- 状态激活后状态

攻击只在自己的类写了开始状态:

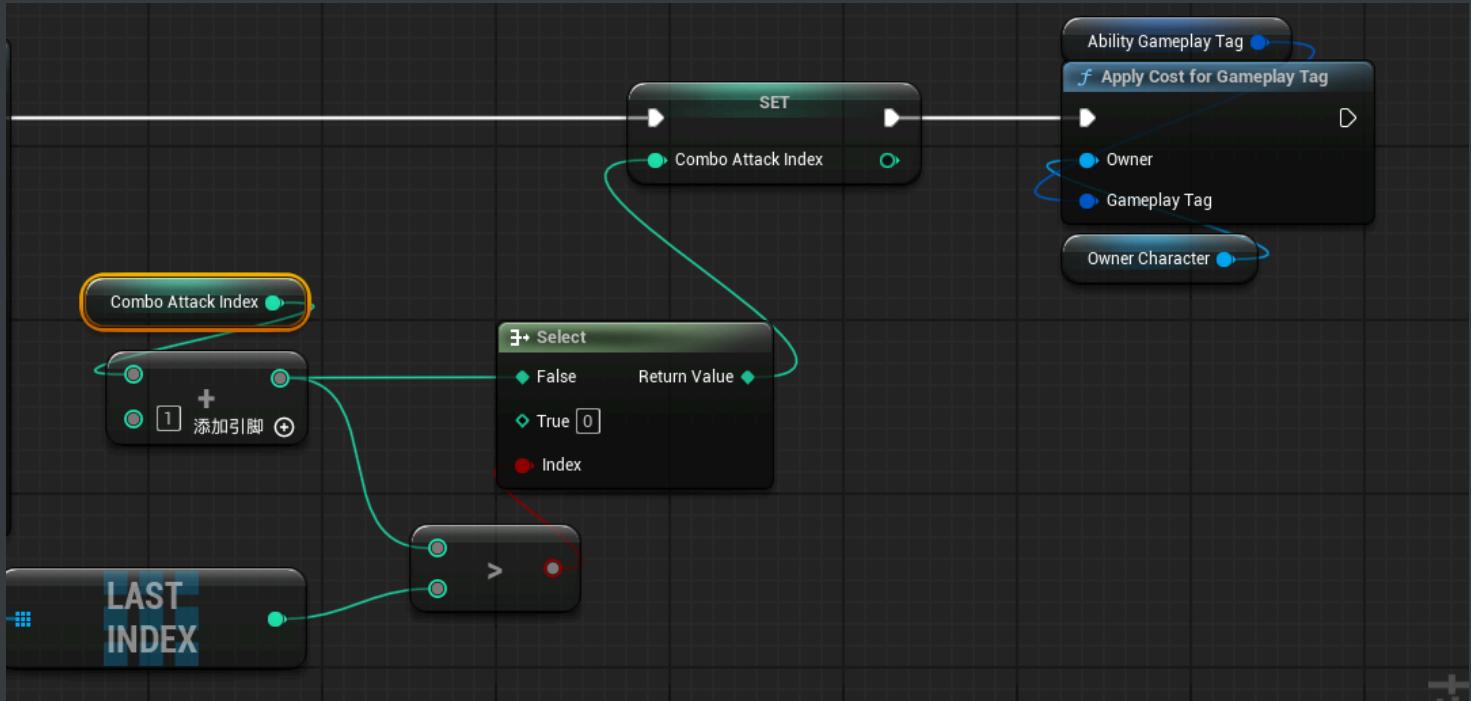
调用自己在数据资产里面的攻击能力BP_PlayerLightAttackAbility, 能力组件和状态组件类似



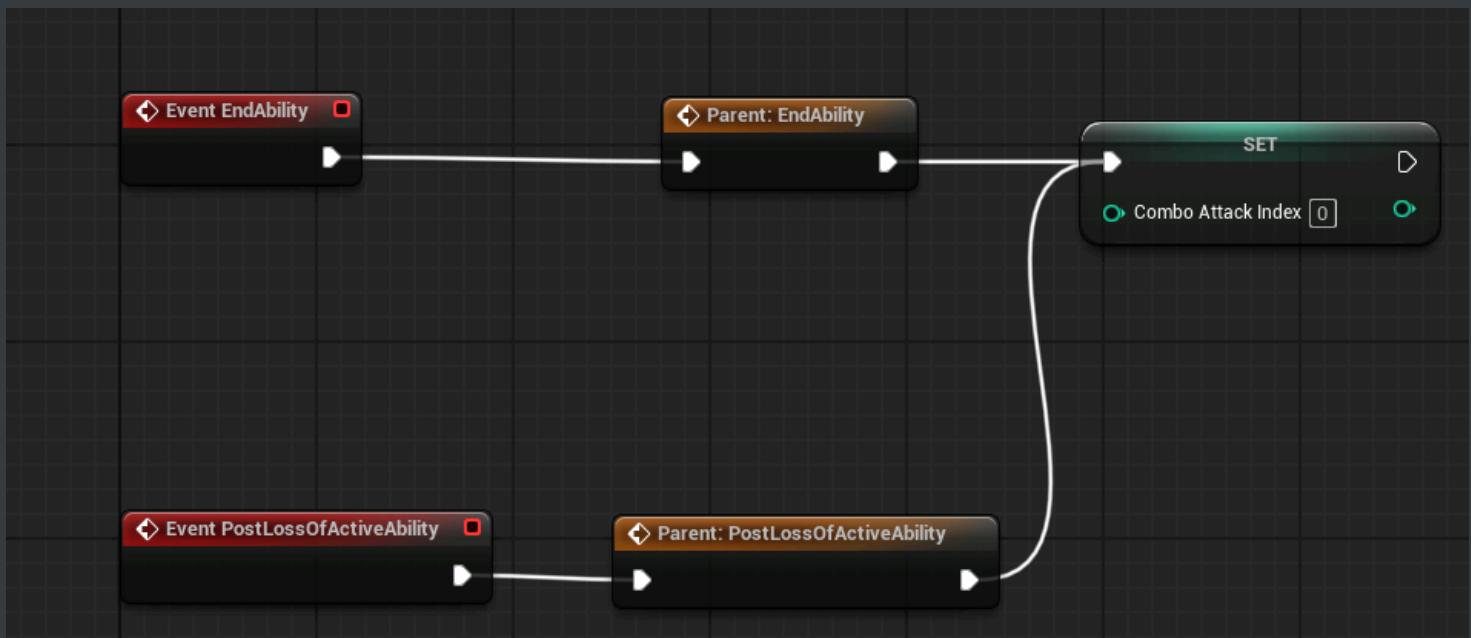
这个能力先获取要攻击的蒙太奇动画并播放



之后增加攻击索引以及计算攻击消耗



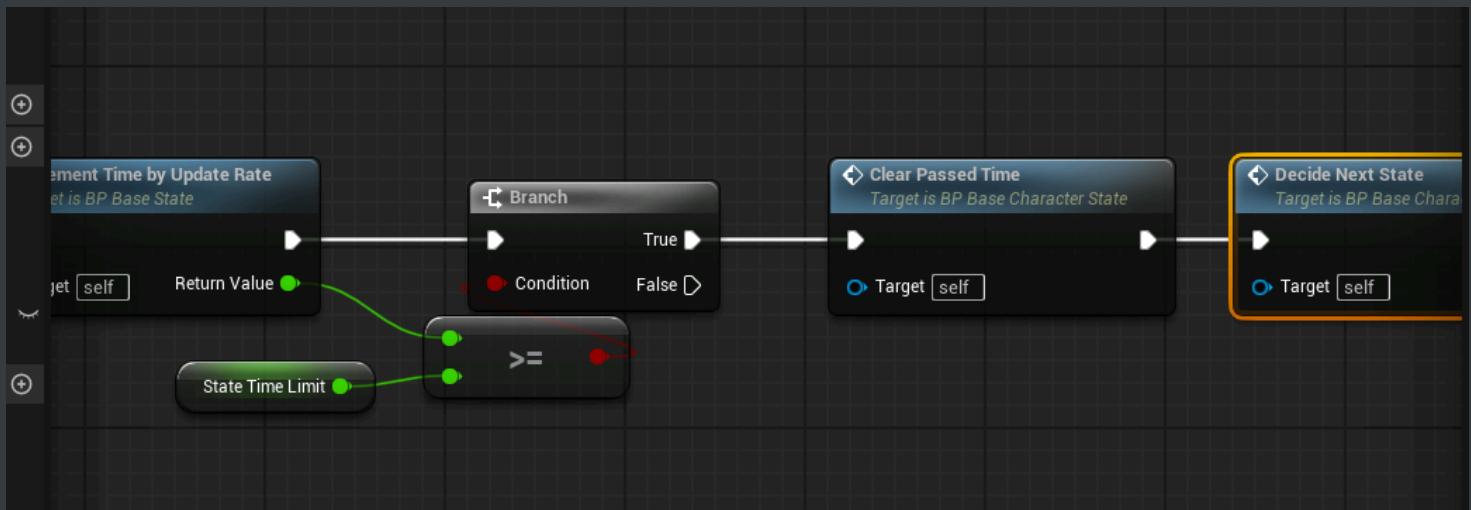
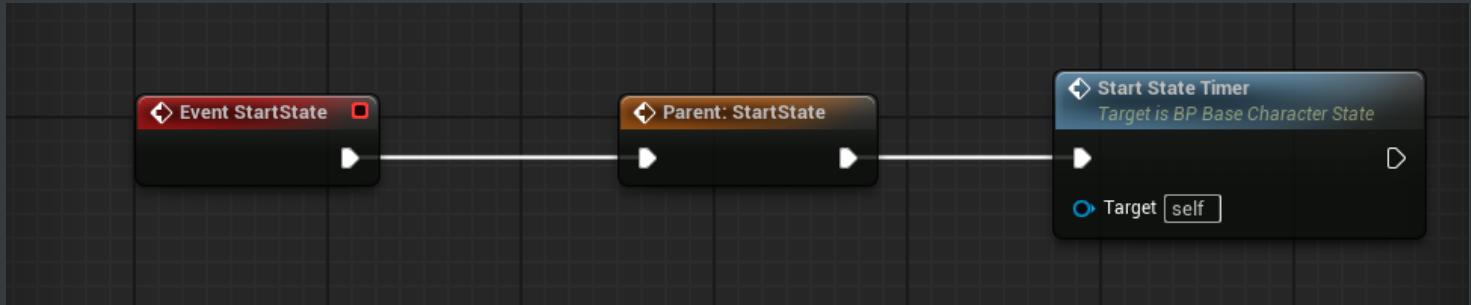
在攻击能力结束时重置索引



接下来只有最后一个问題,如何结束这个状态,在父类中

在开始状态是会启用一个计时,在时间超过StateTimeLimit之后就会调用状态的EndState方法

但目前StateTimeLimit是0秒也就是每次执行完就结束状态



功能组件

UTempestAttributesComponents

用于管理属性和属性修饰器的类

主要功能

- 属性管理**: 创建、存储和检索游戏中的各种属性对象
- 属性修饰器管理**: 管理影响属性的修饰器
- 生命周期管理**: 在所有者被销毁时清理资源

核心成员变量

- CreatedAttributes : 存储所有创建的属性对象
- CreatedAttributeModifiers : 存储所有创建的属性修饰器

属性业务类

有两种属性，一个就是Attribute另一个是AttributeModifier(属性修饰器)

两者之间的差异

维度	Attribute (属性)	ModifyAttribute (属性修饰)
本质	基础数据容器	对属性的操作或影响规则
可变性	存储基础值	定义如何改变属性值
生命周期	通常长期存在	可能是临时的或条件性的
功能	"是什么" - 存储状态	"如何变" - 定义变化规则

具体解释:

UTempestBaseAttributeObject

每个属性带有两个结构体用来完成业务

FAttributeProperties,用来存储属性的基本属性

```
USTRUCT(BlueprintType)
struct FAttributeProperties
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Properties")
    float AttributeValue;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Properties")
    float MinAttributeValue;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Properties")
    float MaxAttributeValue;

    FAttributeProperties() :AttributeValue(0.0f), MinAttributeValue(0.0f), MaxAttributeValue(0.0f) {};
    ~FAttributeProperties() {};
};
```

FInstancedAttributes,用来封装UTempestBaseAttributeObject*,为什么要这样,因为Instanced关键字以及UTempestBaseAttributeObject类中含有关键字EditInlineNew,代表这个类我是可以在外面直接选择实例化的,但是一般这个属性会含有多个所以会用数组,但是UE里面用数组直接操作指针不友好,所以需要包一层结构体,来在外面使用

```

USTRUCT(BlueprintType, meta = (DisplayName = "Instanced Attributes"))
struct FInstancedAttributes
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(BlueprintReadWrite, Instanced, EditAnywhere, Category = "Instanced Variables")
    class UTempestBaseAttributeObject* AttributeToCreate;

    FInstancedAttributes() :AttributeToCreate(nullptr) {};
    ~FInstancedAttributes() {};
};


```

UTempestBaseAttributeModifier

每个属性修饰器都会带有一个FAttributeModifierProperties结构体，用于存储这个修饰器的核心信息

属性的Tag、是否无限时间、持续时间、修饰器间隔、添加的数量

注意这里面的Tag是属性的Tag，AttributeModifierTag 这个变量表示的是修饰器的Tag

```

USTRUCT(BlueprintType)
struct FAttributeModifierProperties
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Modifier Properties", meta = FGameplayTag AttributeToModify);

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Modifier Properties")
    bool bIndefinite = false;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Modifier Properties", meta = float ModificationDuration = 0.f);

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Modifier Properties")
    float ModificationInterval = 0.2f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Base Attribute Modifier Properties")
    float AmountToAdd = 1.f;

};

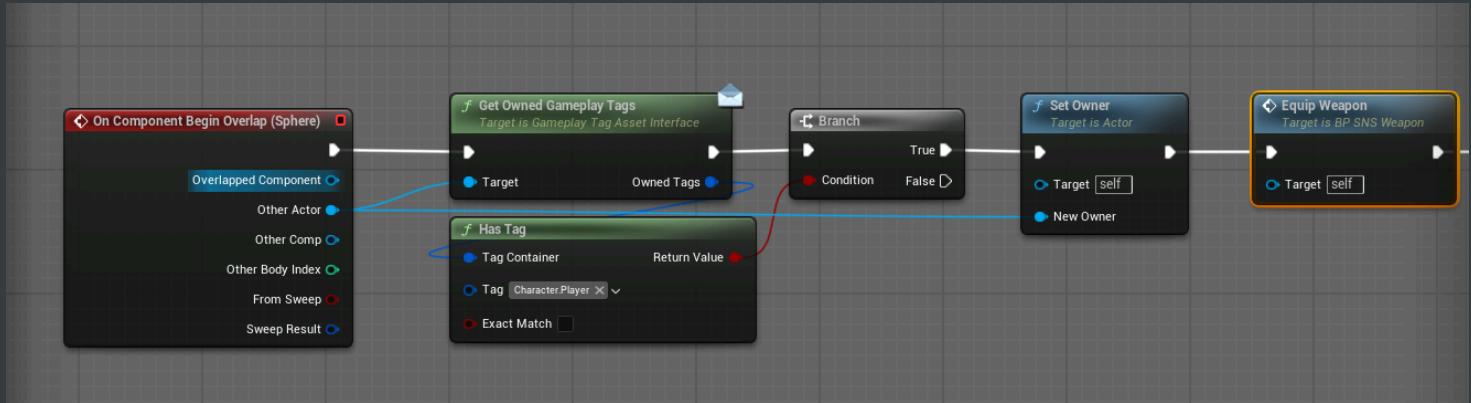
```

示例

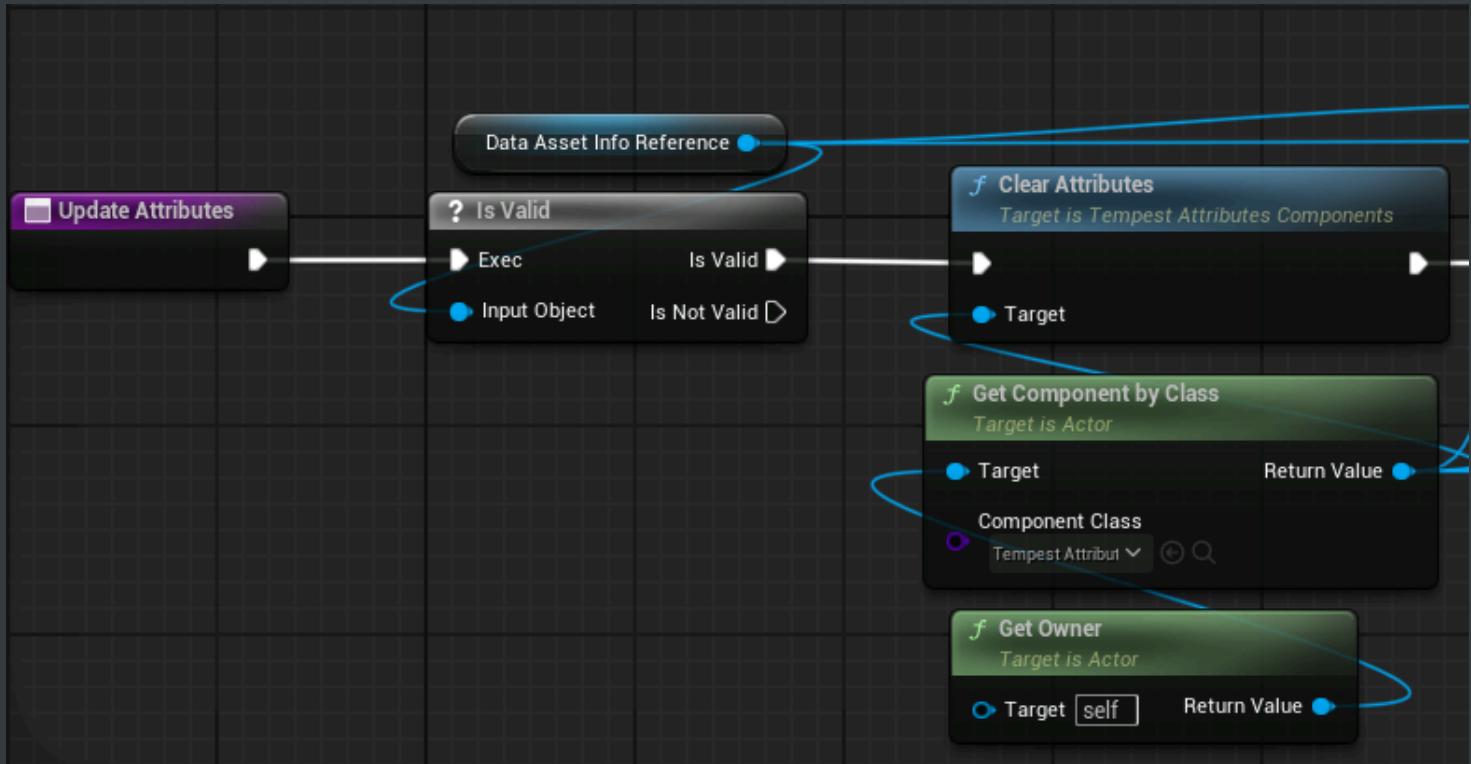
基础属性

装备武器的时候会更新这个武器的属性

在EquipWeapon中会更新这个武器的所有数据属性、特性、状态之类的



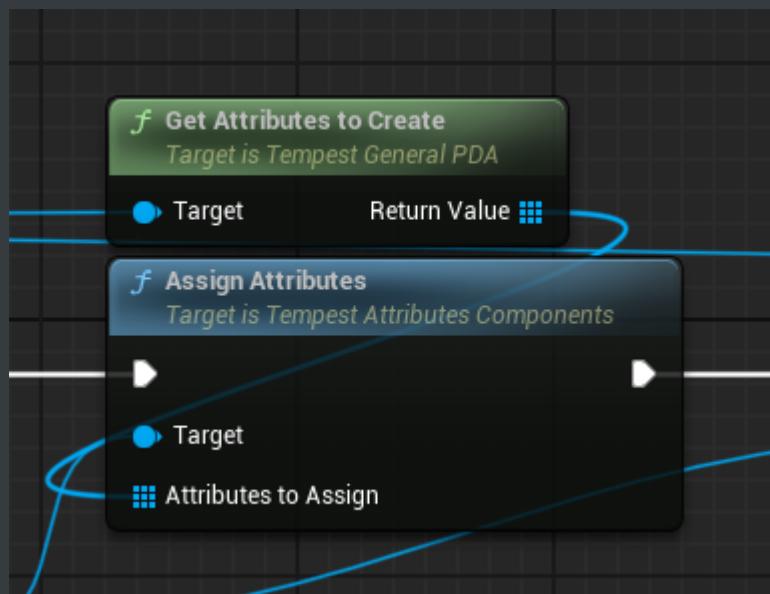
在更新属性首先判断，配置的数据存不存在，这个配置是这个武器对应的数据资产，



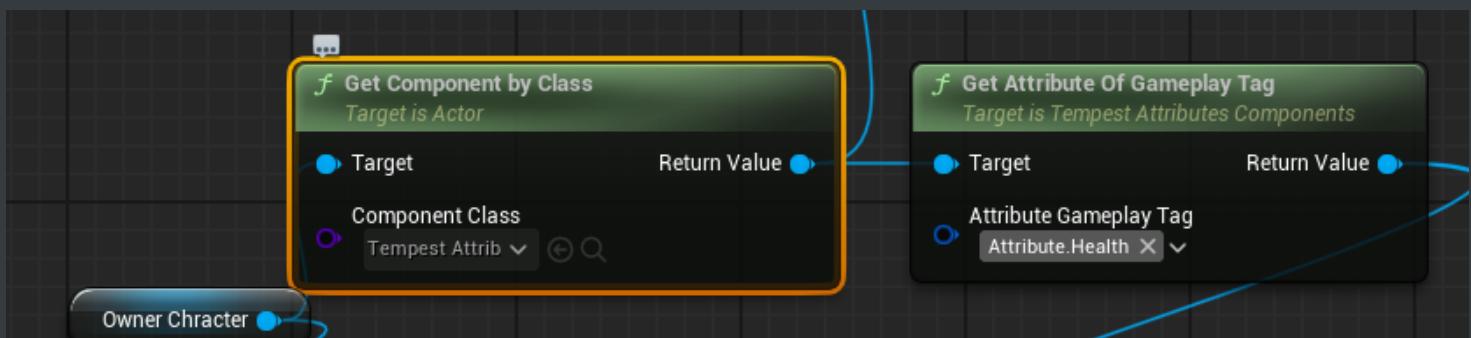
通过GetAttributesToCreate获取一个UTempestBaseAttributeObject*的数组，然后将这组数组添加到Created Attributes变量中存储起来

遍历的是这个AttributeToCreate里面的数据，自己配置的

▼ Attributes to Create	3 数组元素
▼ 索引 [0]	1 个成员
▼ Attribute to Create	BP Health Attribute
▼ Attribute Properties	
▼ Attribute Values	
Attribute Value	100.0
Min Attribute Value	0.0
Max Attribute Value	100.0
Attribute Gameplay Tag	Attribute.Health X
▼ 索引 [1]	1 个成员
▼ Attribute to Create	BP Stamina Attribute
▼ Attribute Properties	
▼ Attribute Values	
Attribute Value	100.0
Min Attribute Value	0.0
Max Attribute Value	100.0
Attribute Gameplay Tag	Attribute.Stamina X
▼ 索引 [2]	1 个成员
▼ Attribute to Create	BP Attack Power Attribute
▼ Attribute Properties	
▼ Attribute Values	
Attribute Value	2.0
Min Attribute Value	0.0
Max Attribute Value	10.0
Attribute Gameplay Tag	Attribute.Attack Power X



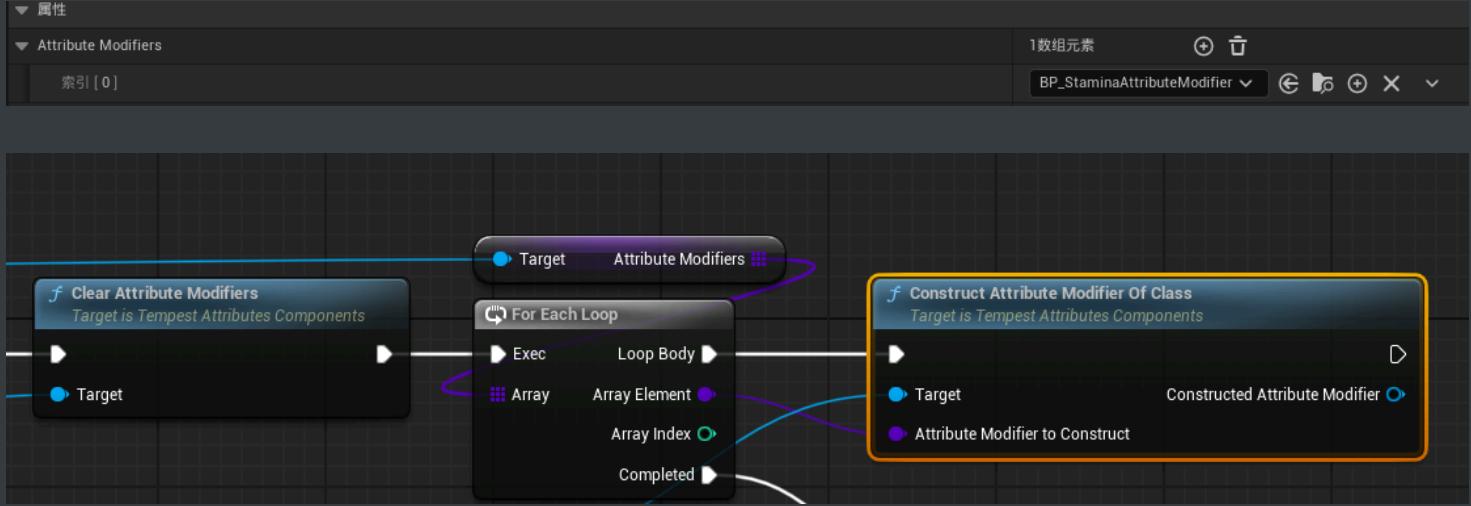
在需要这个地方可以通过GetAttributeOfGameplayTag这个方法来获取你需要的属性



目前版本角色属性完完全全是根据武器的配置来的，没有自身的基础属性，TODO需要修改

修饰器属性

依旧拿装备武器后更新数据举例，这个属性配置在AttributeModifiers中，通过类来构造属性

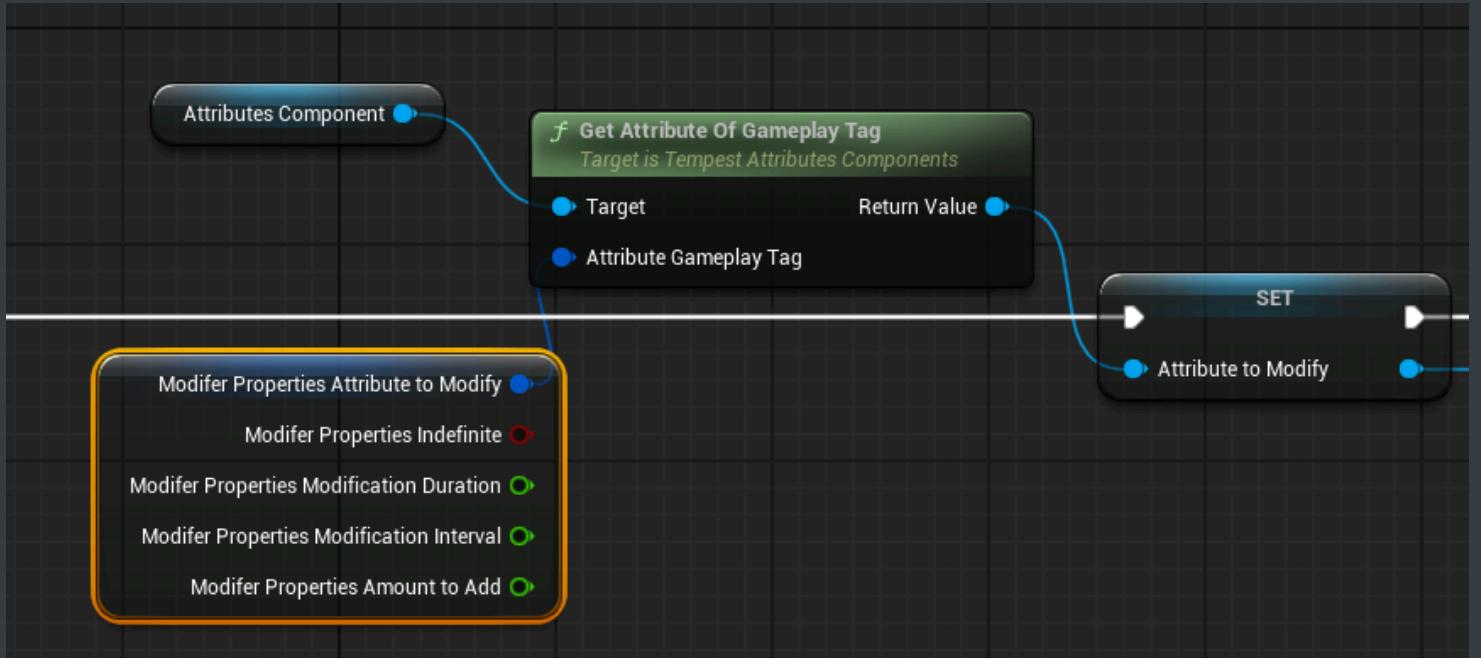


首先ConstructAttributeModifierOfClass方法会调用Modifier的ConstructAttributeModifier方法

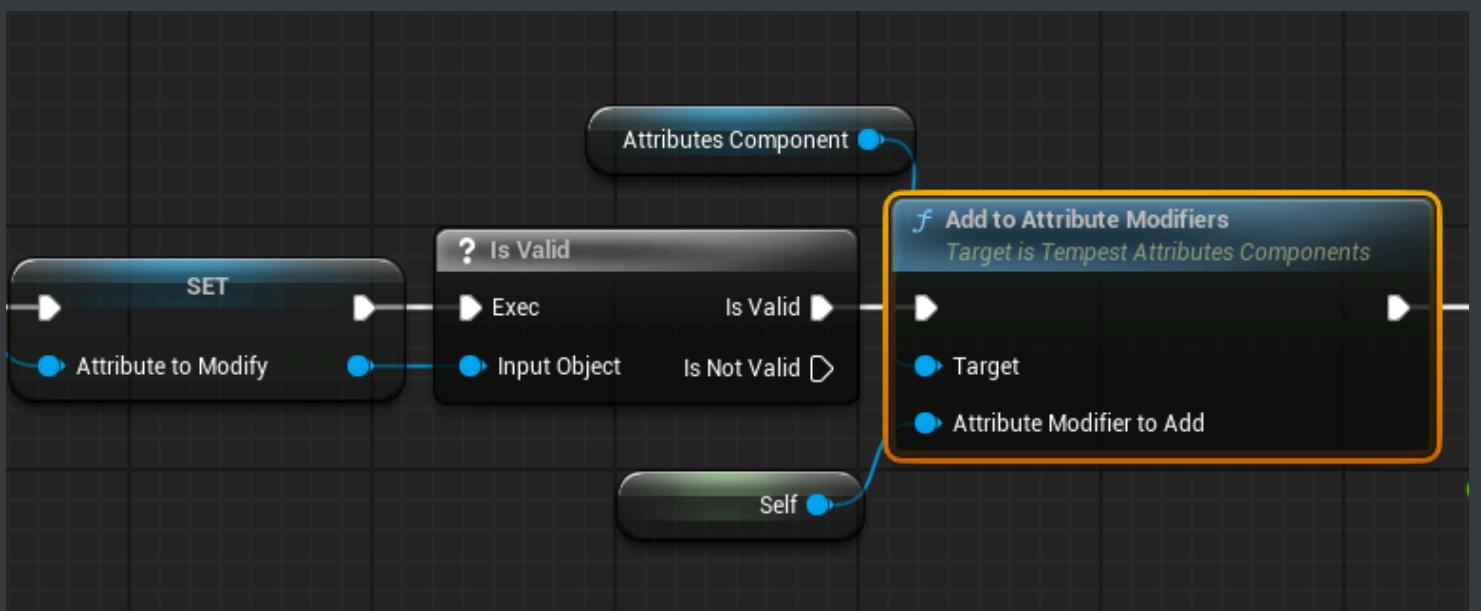
```
if (AttributeModifierToConstruct)
{
    UTempestBaseAttributeModifier* LocalNewAttributeModifier;
    LocalNewAttributeModifier = NewObject<UTempestBaseAttributeModifier>(this);
    LocalNewAttributeModifier->ConstructAttributeModifier();
    ConstructedAttributeModifier = LocalNewAttributeModifier;
}
```

然后再BP_BaseAttributeModifer蓝图类中

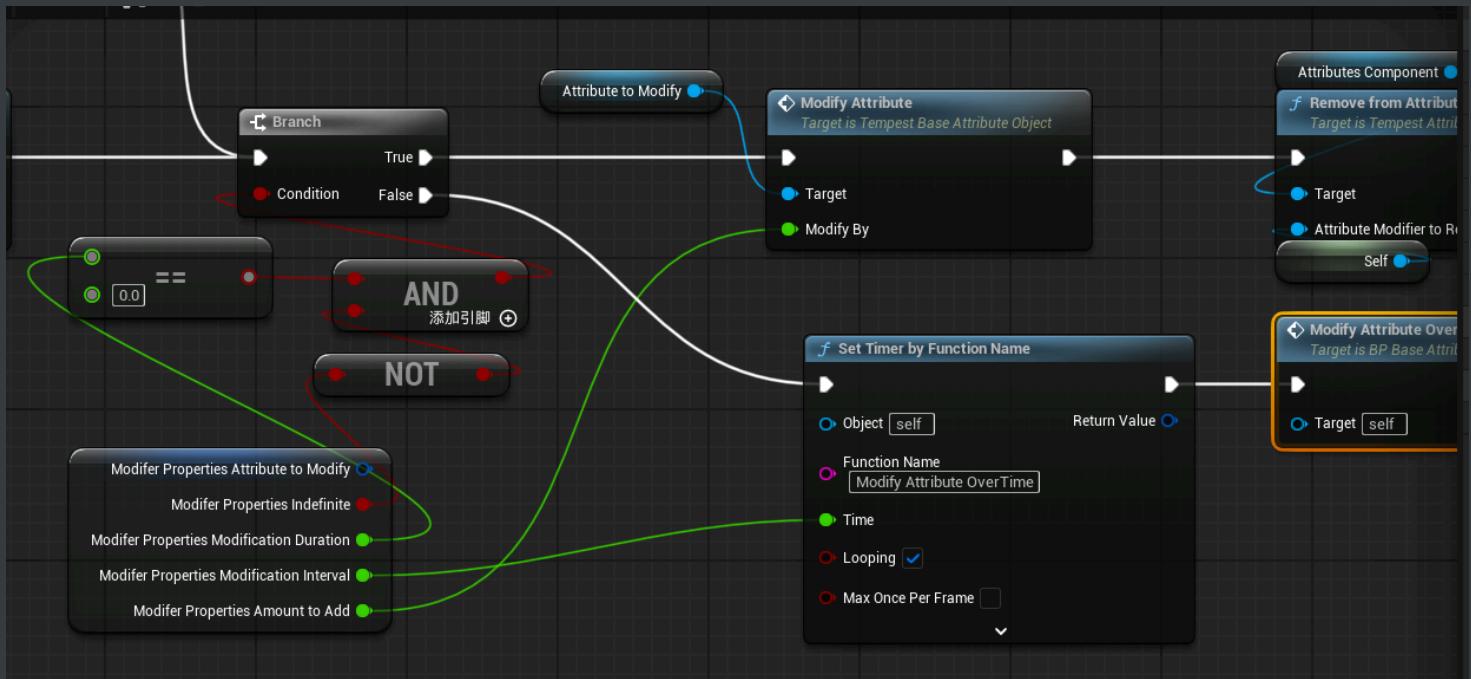
首先根据tag获取所有的Attribute



然后将自身存入到CreatedAttributeModifiers



然后根据结构体FAttributeModifierProperties来配置属性



UTempestPropertiesComponent

管理游戏实体的特殊属性（特性）实例性的类

Properties和Attribute的区别

1. Attributes :

- 通常表示实体的基础特性或数值状态
- 如: 生命值、攻击力、防御力、移动速度等（但目前角色的移动速度用的是Properties，不知道为什么）
- **往往是数值型的、可量化的**

2. Properties:

- 表示更复杂的行为或特殊能力
- 如: 技能消耗、允许通过按键通过的转态，以及状态和能力，以及动作的蒙太奇
- **通常是行为导向的、包含逻辑的**

一般Attributes只负责提供数据，Properties还会有能力实现

UTempestBaseStateManagerComponent

概述:

一个用于管理游戏对象状态的组件，实现了状态模式，允许游戏对象在不同的行为状态之间切换和管理。

只允许有一个主动状态,可以拥有多个被动状态

核心成员变量:

1. **ActivatableStates**: TArray<UTempestBaseStateObject*> - 存储所有可激活状态的数组
2. **QueuedStates**: TArray<TSubclassOf> - 存储排队等待执行的状态类
3. **PassiveStates**: TArray<UTempestBaseStateObject*> - 存储被动状态的数组
4. **CurrentActiveState**: UTempestBaseStateObject* - 当前激活的状态对象
5. **OnUpdatedCurrentActiveState**: 委托/事件，当当前激活状态更新时广播

工作流程

1. 状态执行流程:

- 检查状态是否存在 → 不存在则构造新状态
- 检查条件(可选) → 执行状态
- 触发PreStateActivation → Start State → PostStateActivation

2. 状态切换流程:

- 当前状态PreLossOfActiveState
- 设置新状态
- 广播OnUpdatedCurrentActiveState
- 原状态PostLossOfActiveState

特性业务类

示例

详情请见一个攻击的流程示意(将这几个组件串起来):

###

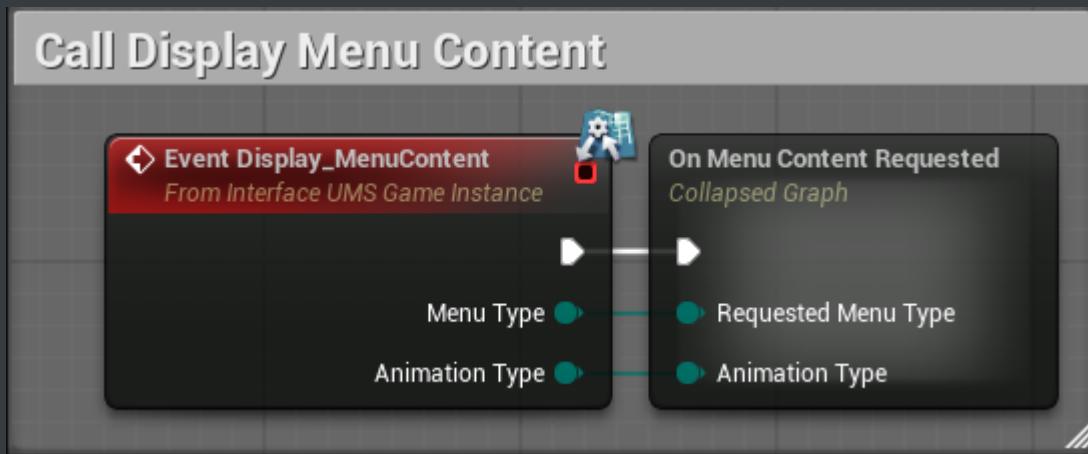
UI框架

大部分功能通过UMS_GameInstance以及WB_MenuMaster组合来实现

内容很多我用到哪写哪

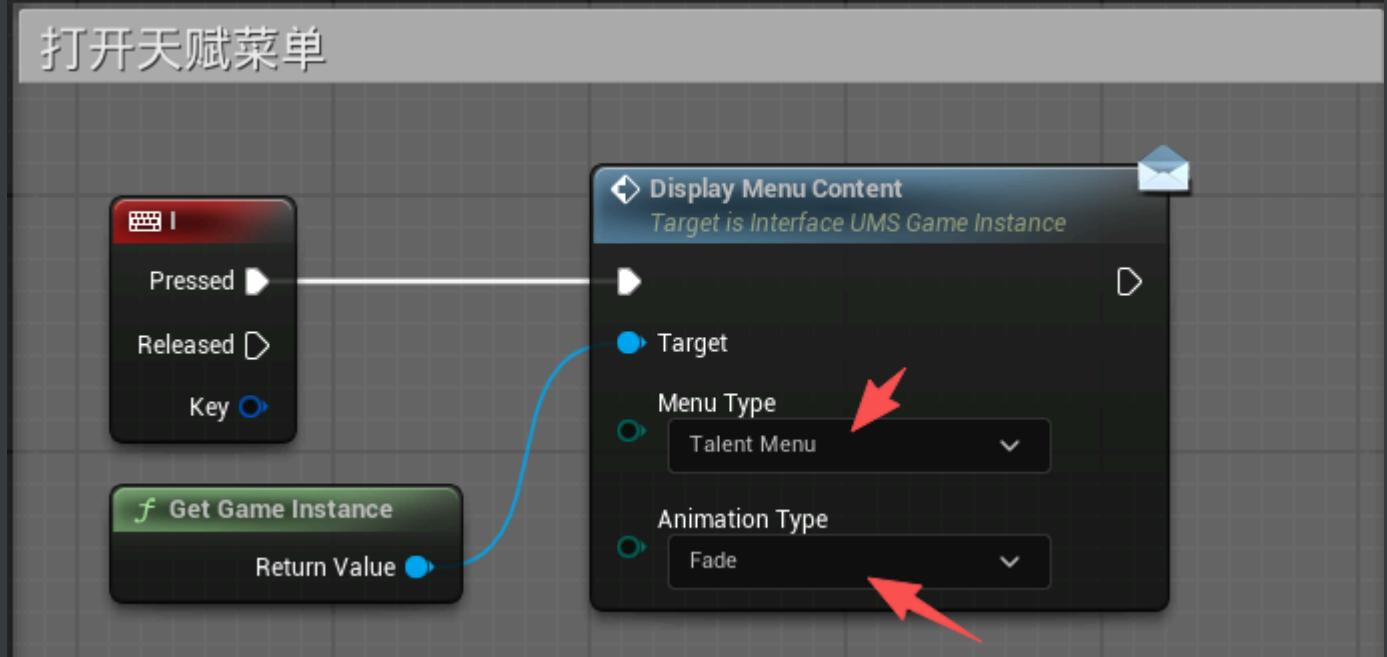
显示一个菜单内容

调用UMS_GameInstance， 中的方法



传入要打开的界面已经动画类型

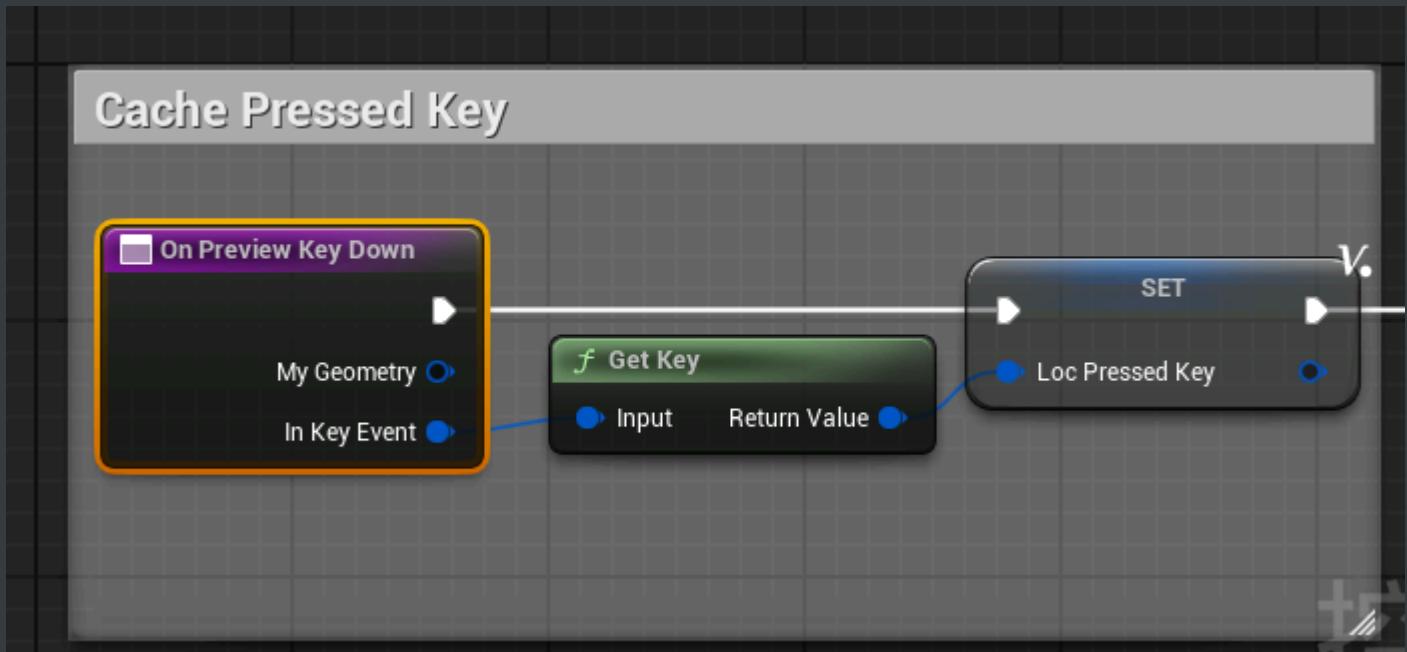
打开天赋菜单



然后去UMS_GameInstance添加一个新的UI界面，具体可以看里面的天赋界面

返回上一菜单ESC

在WB_MenuMaster中的这个方法，来检测可聚焦UI的按键



在判断回退按键的时候，判断自己的按键



WB_MenuMaster:

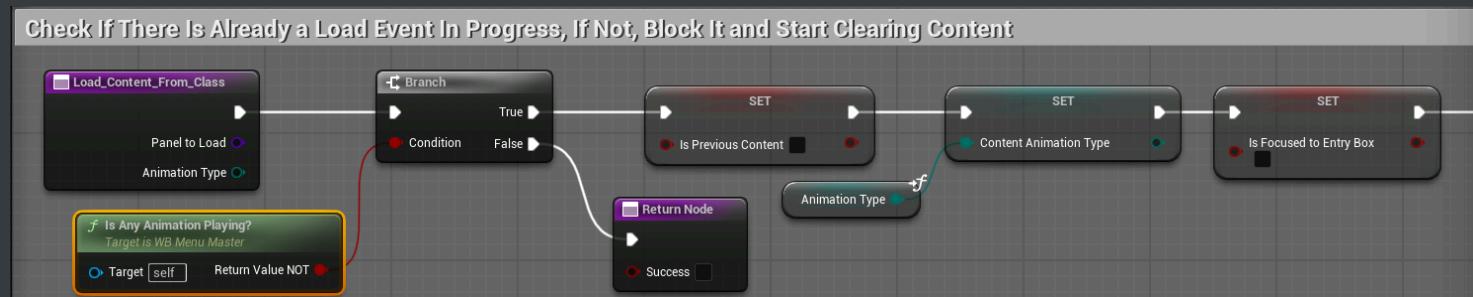
Load_Content_From_Class:

传入要加载的WBP类，加载这界面的动画类型

检查是否有加载事件正在进行，如果没有，则阻止它并开始清除内容

先检查是否有动画正在播放，返回的是个not play

设置是否是上一个节点，播放的动画类型，是否聚焦输入框

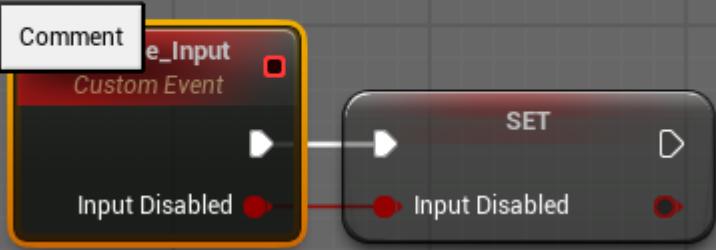


禁用所有输入，将Input Disabled设置为false

Disable All Inputs



Enable/Disable Input



设置为不可点击设置，然后将要加载的界面设置为Target Content Class

Set Not Hit-Testable To Avoid Click Operations | Set Target Class

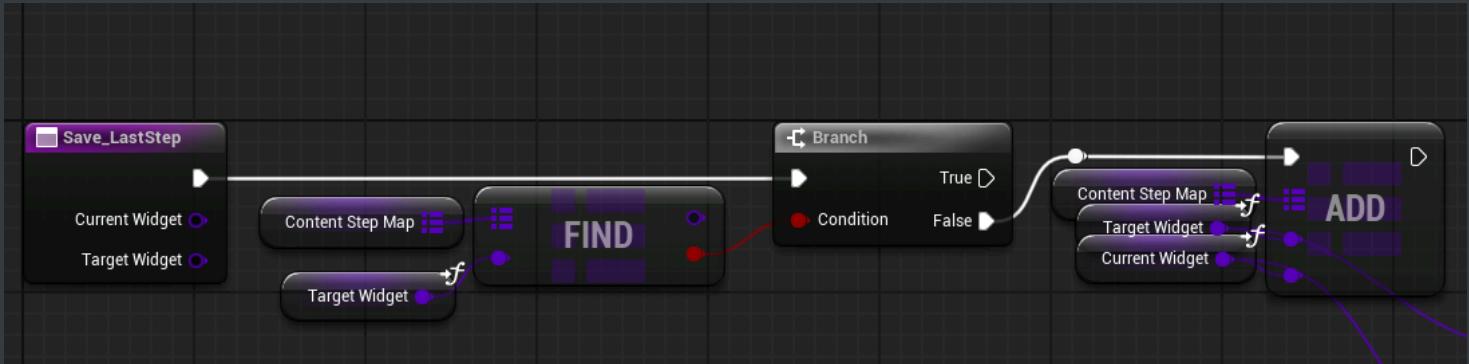


将最后一步保存起来，以便在返回菜单时使用。当前显示的就是最后一步存到Current Class

Save Last Step To Use When Going Back On menu



有个一个字典ContentStepMap,Key是目标页面, value是当前页面

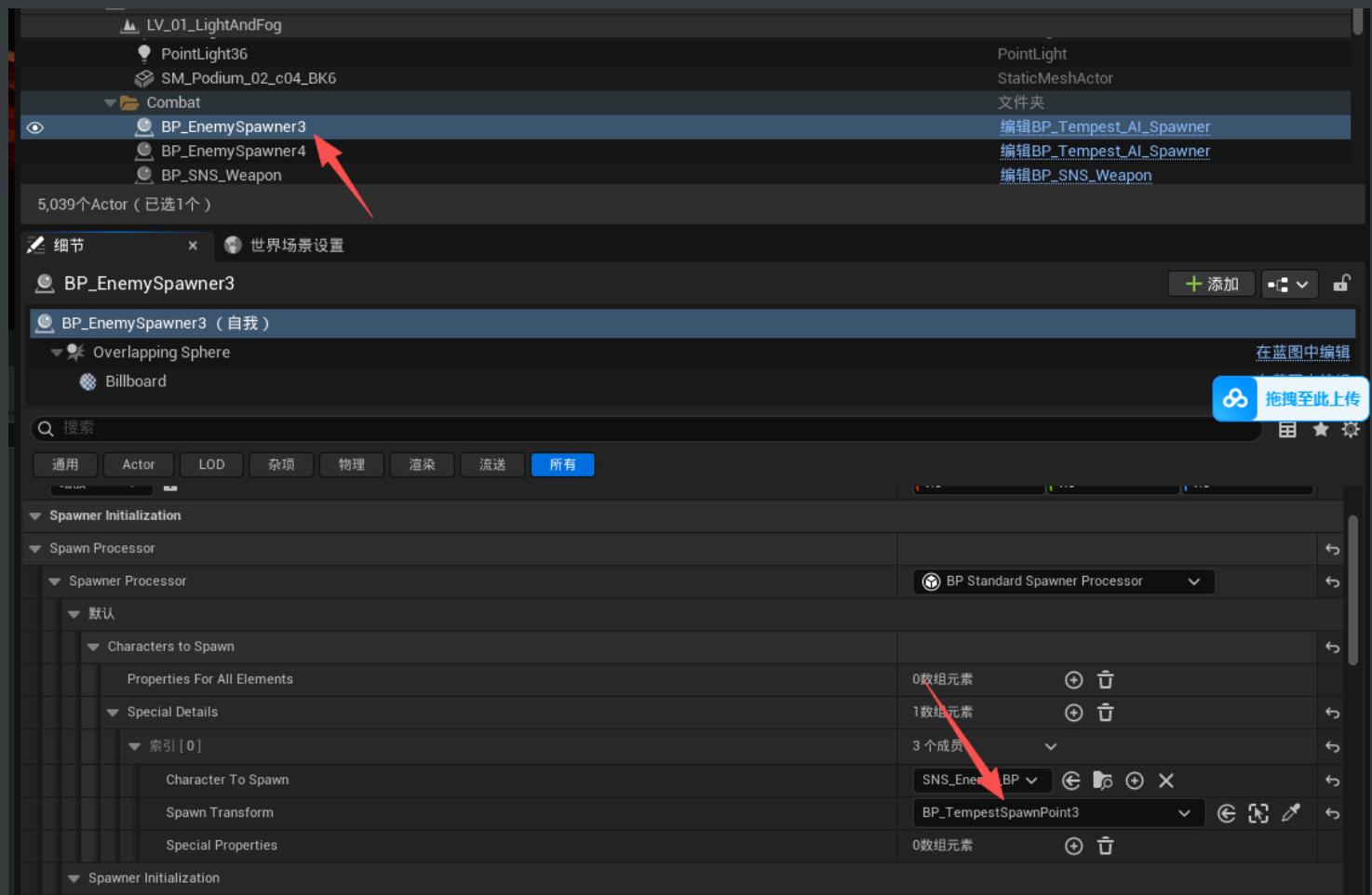


AI行为树的流程：

如何进入状态机：

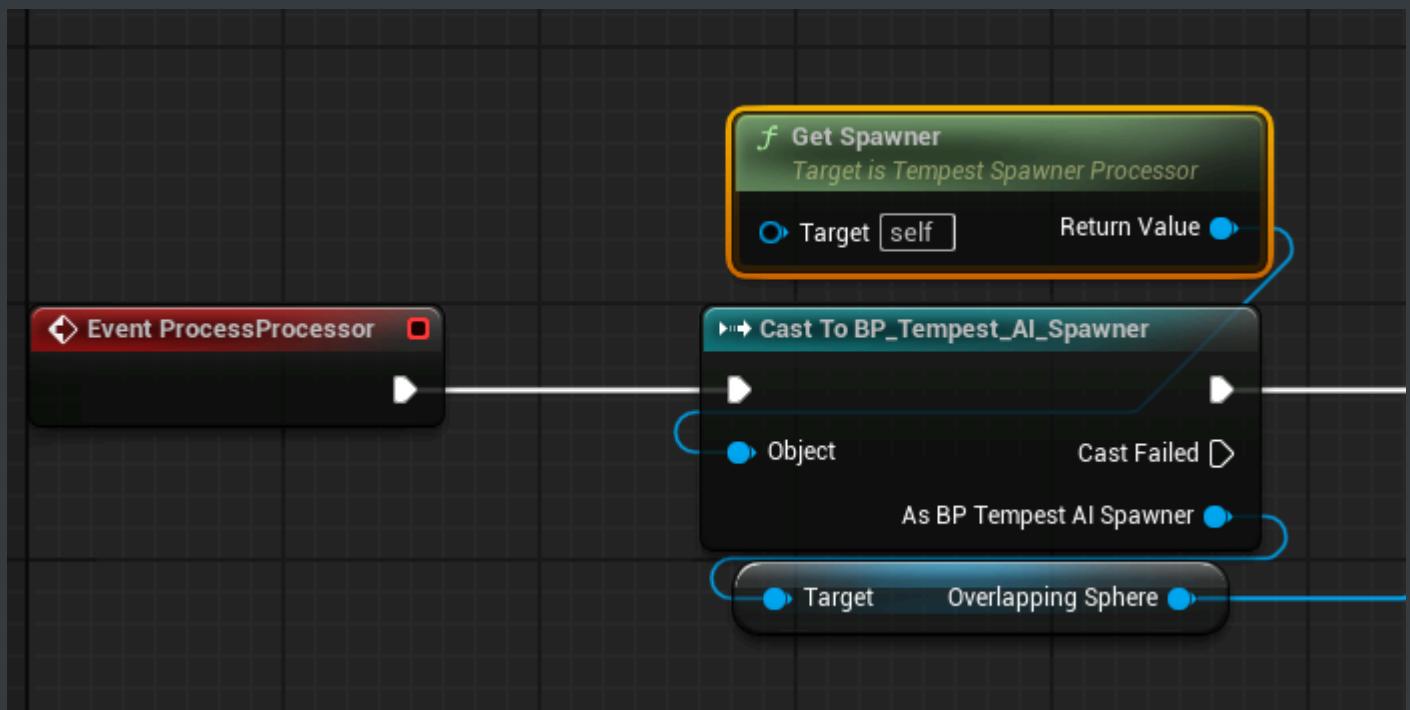
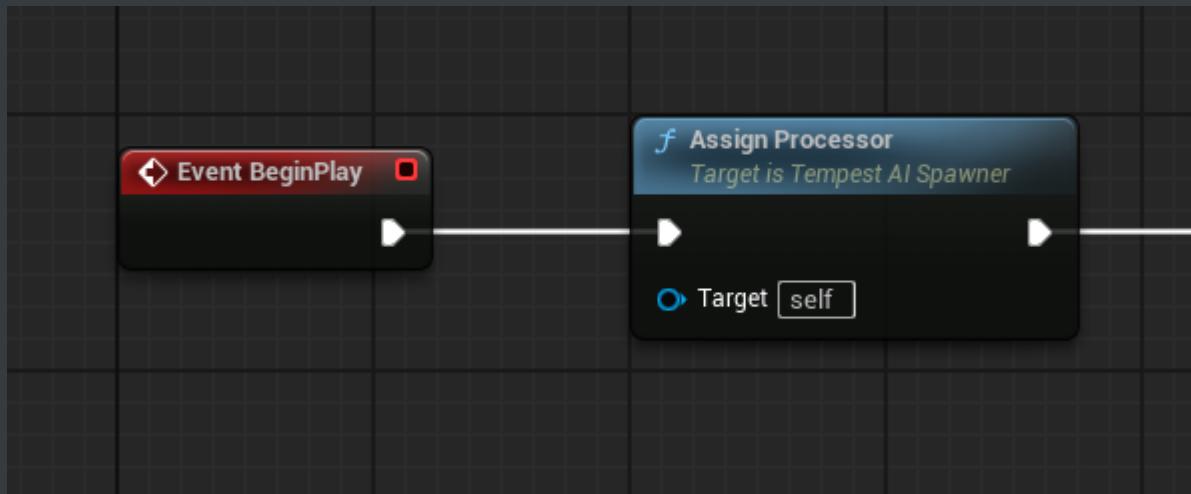
首先是生成NPC，简单过一下后面，可能不是这种方法生成

具体要生成的角色和位置在场景中的BP_Tempest_AI_Spawner配置

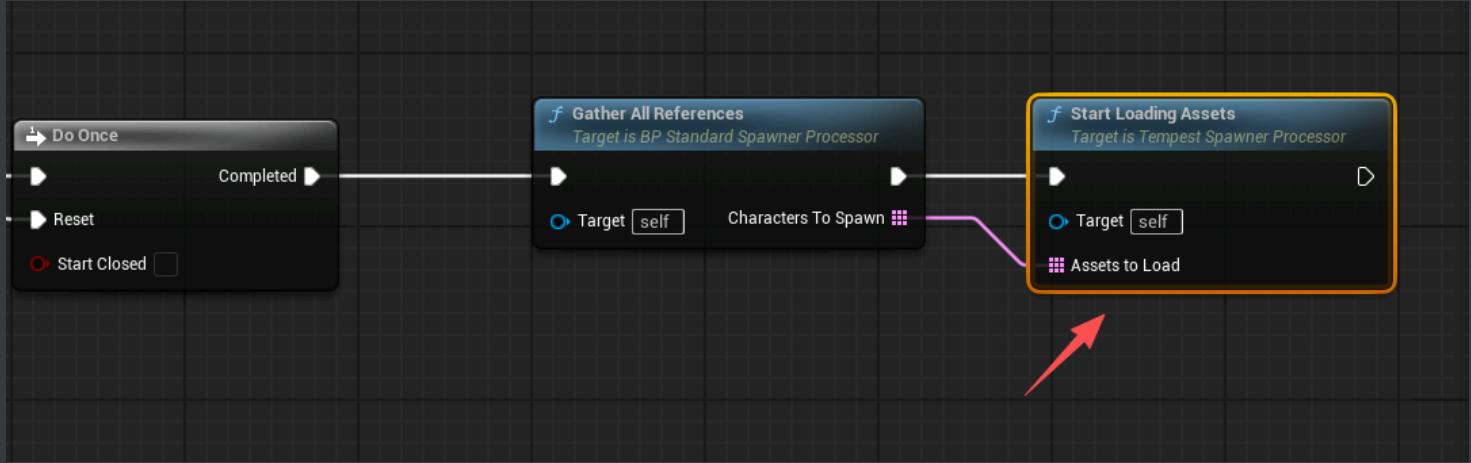


生成碰撞体用的是，BP_Tempest_AI_Spawner蓝图。他两之间创建起联系通过C++方法AssignProcessor和GetSpawner创建联系，

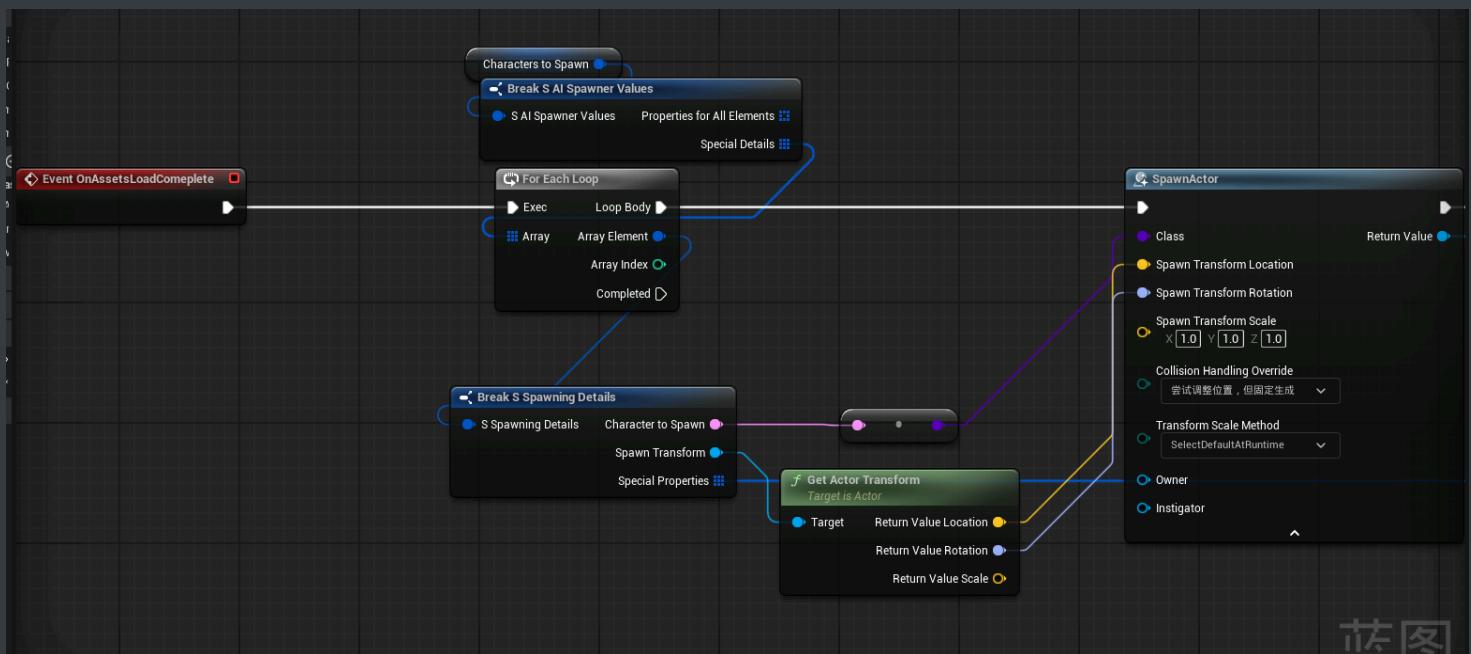
这几个触发器是在C++中完成父子关系的



BP_StandardSpawnerProcessor中根据碰撞检测生成角色，这个是C++中异步加载的方法

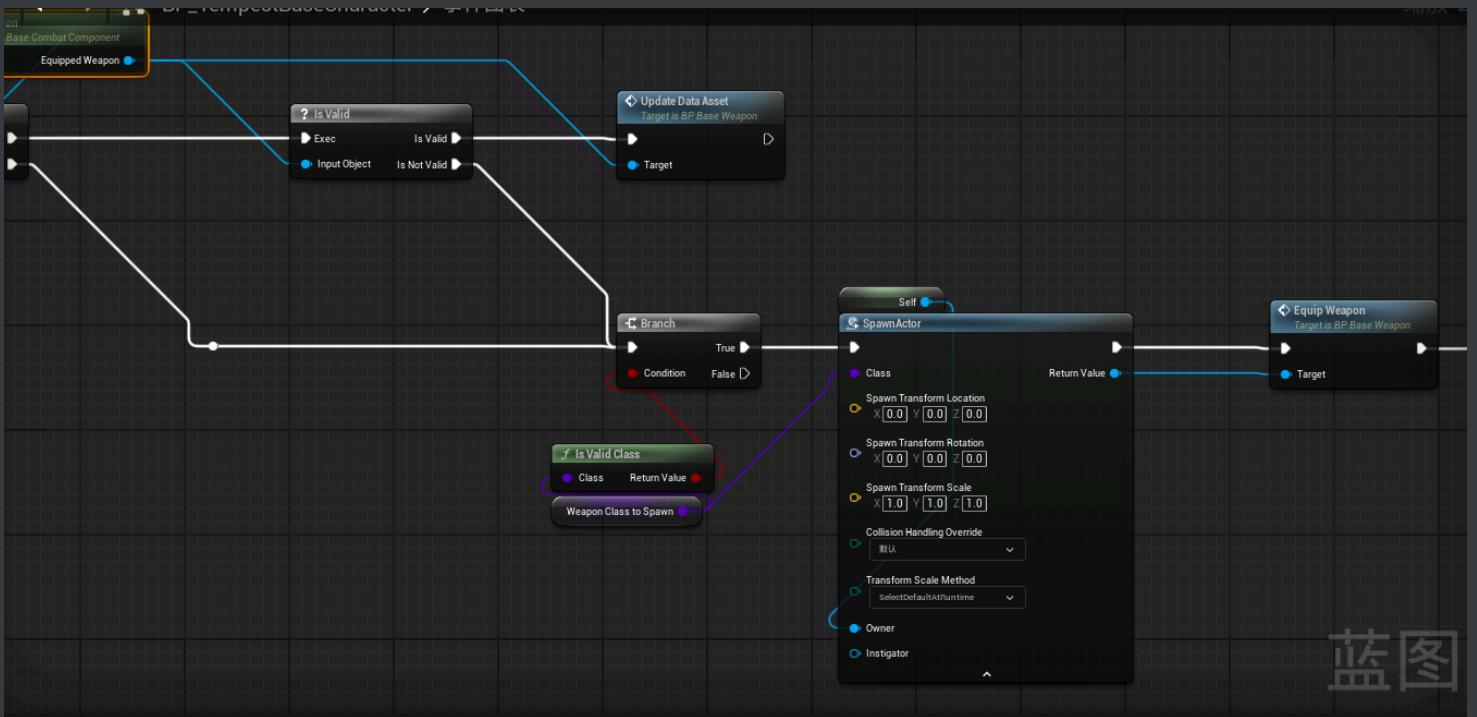


这里面会调用自身的生成物体方法



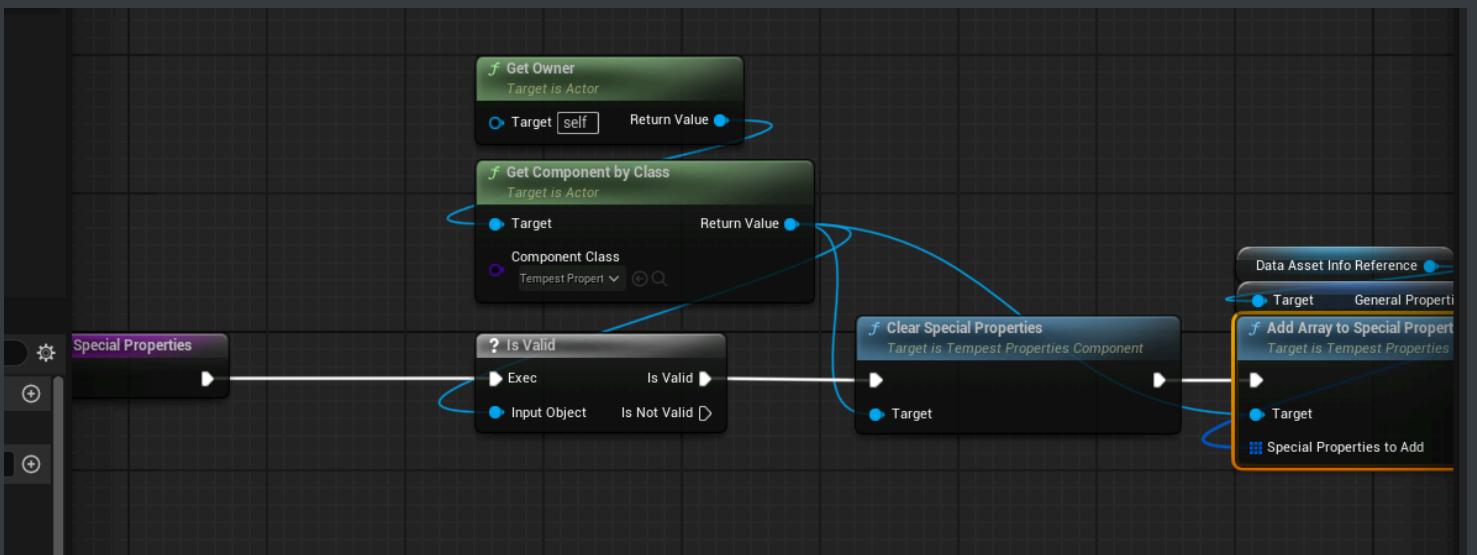
生成物体后，会调用角色基类BP_TempestBaseCharacter中的Possessed

这里面会检测角色有没有武器，没有武器的话会生成武器

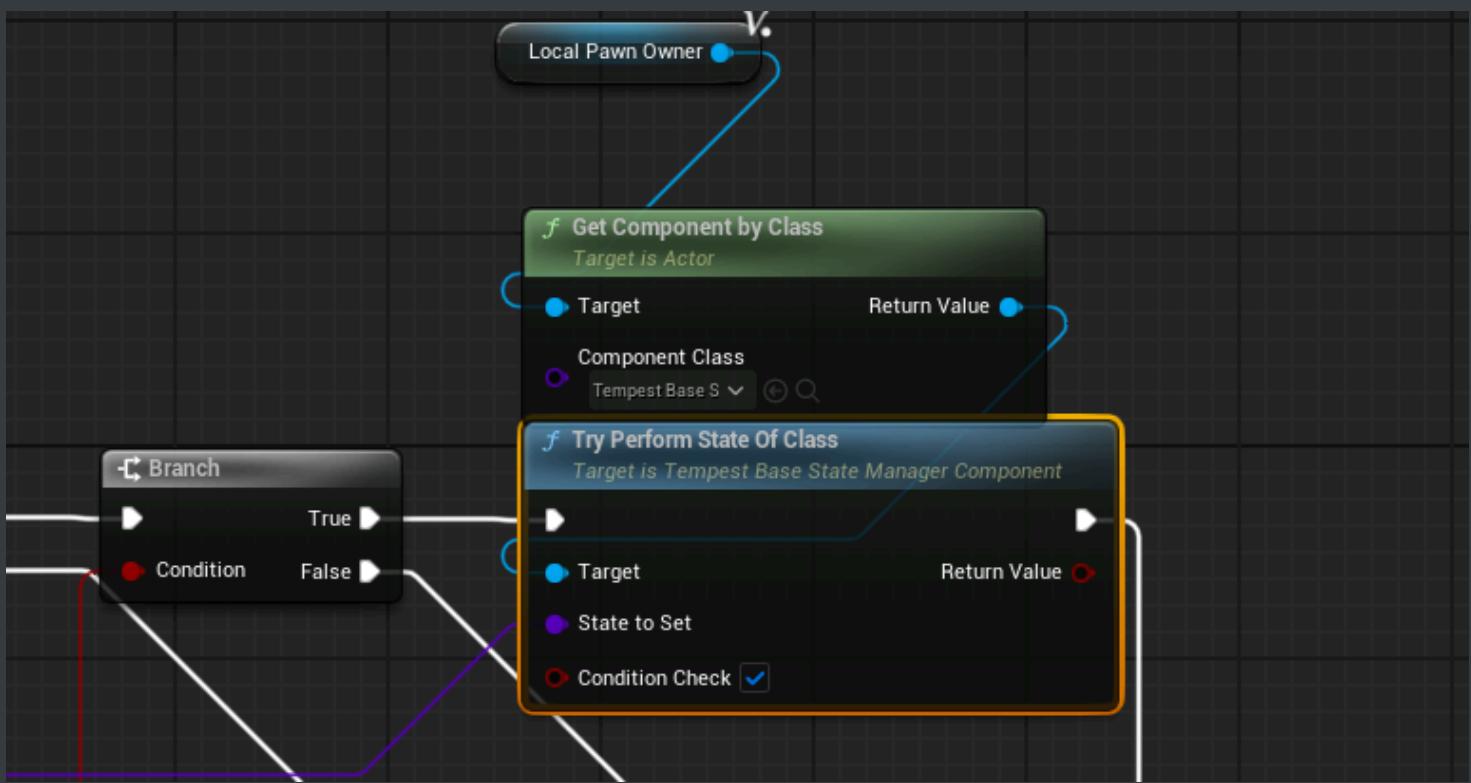
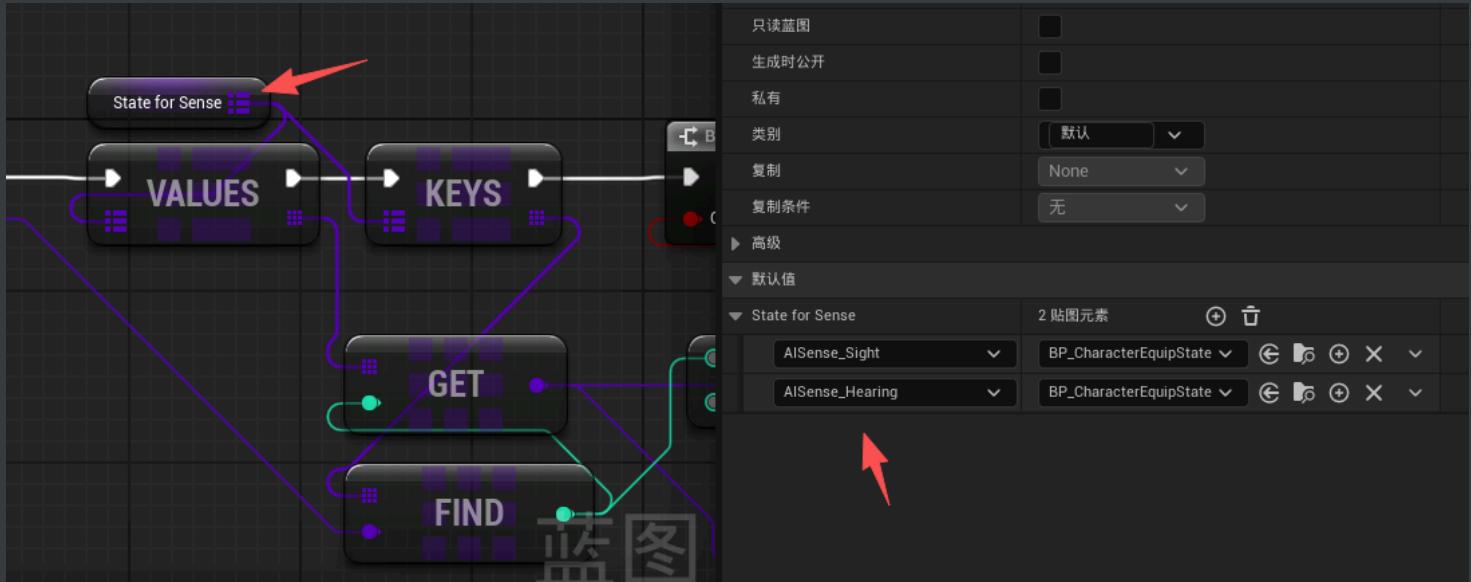


会走EquipWeapon的逻辑，这个逻辑见示例

这个方法会加载他所有特性



然后在BP_EventPerAISSense这个特性中会根据感官来,来尝试进入装备状态,也就进入了AI行为树第一个状态

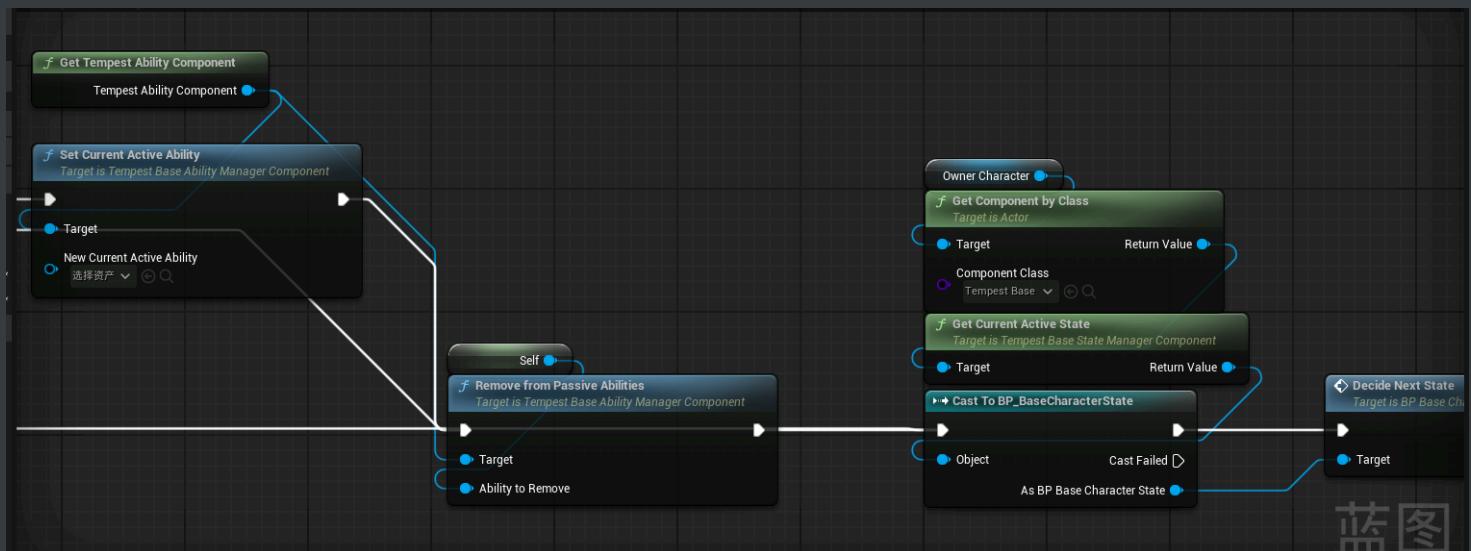


状态之间的切换:

装备能力的状态结束会有一个动画事件来通知结束当前能力结束

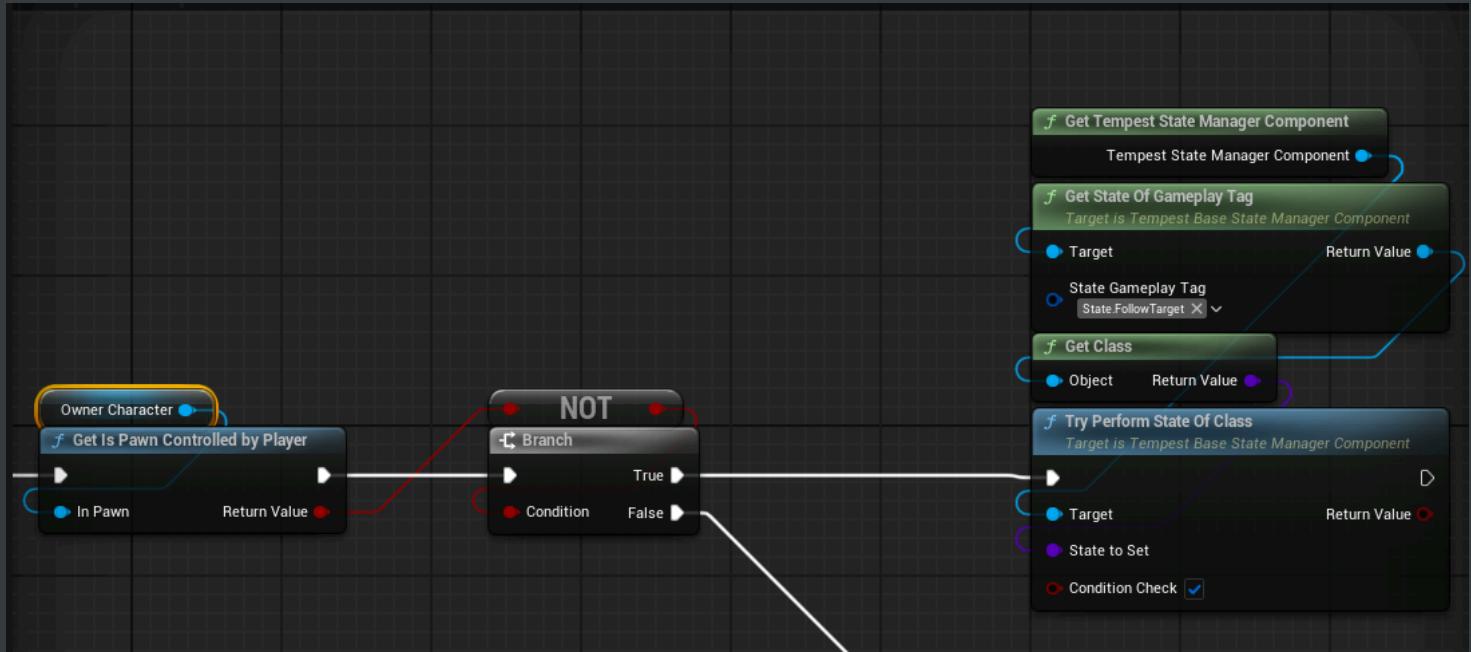


在结束通知里面,会清除当前的能力,然后进入下一个状态

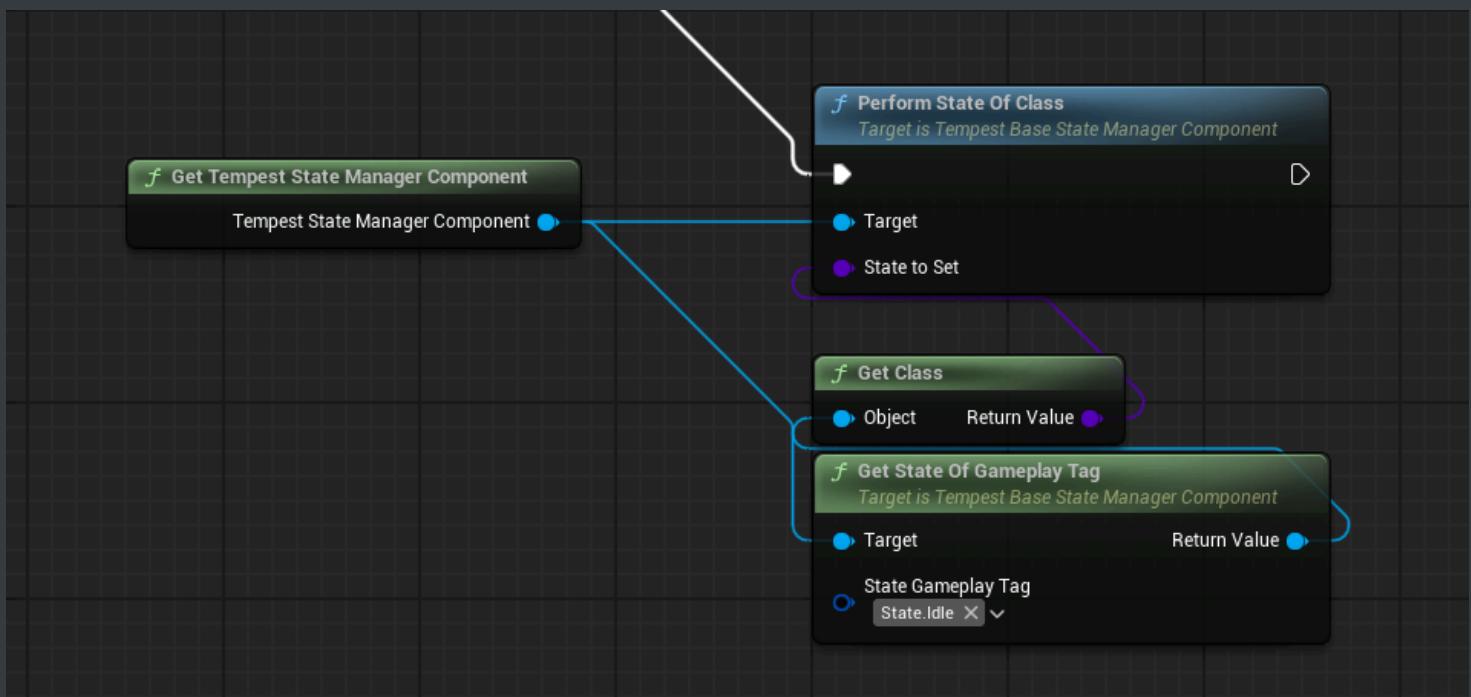


在装备状态的尝试下一状态

先判断有没有玩家控制的Pawn,有的话进入追寻状态

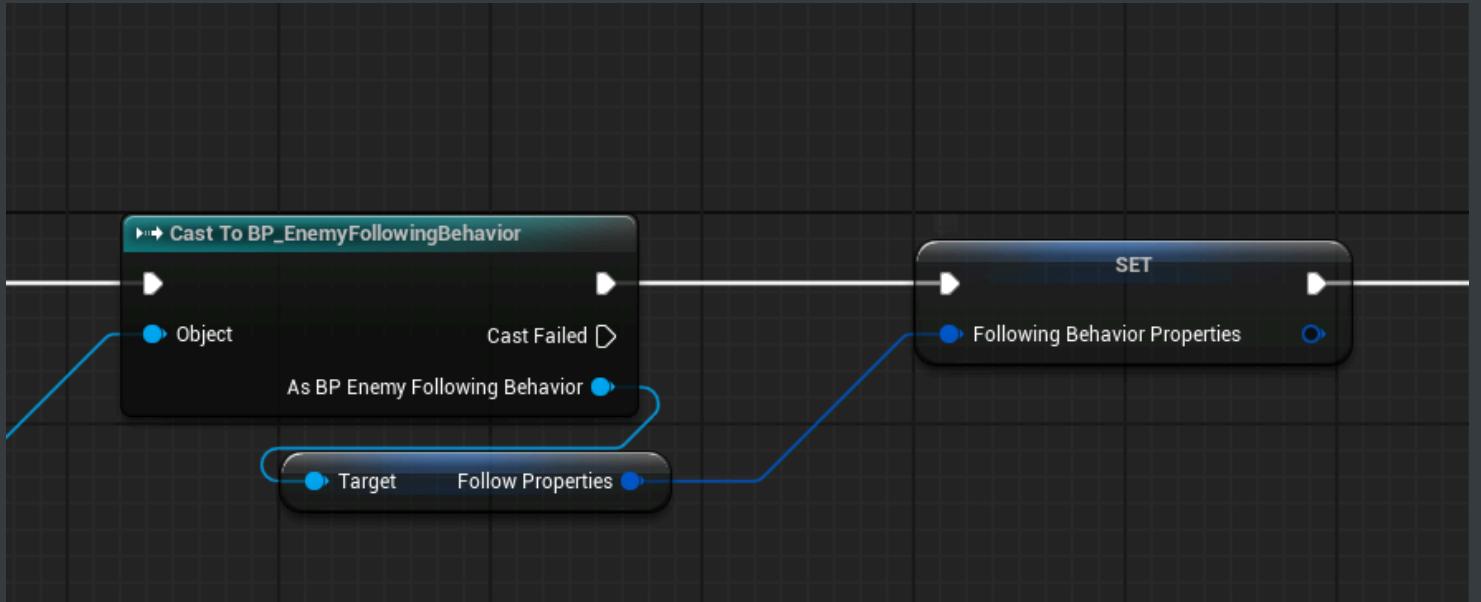


没有的话进入Idle状态



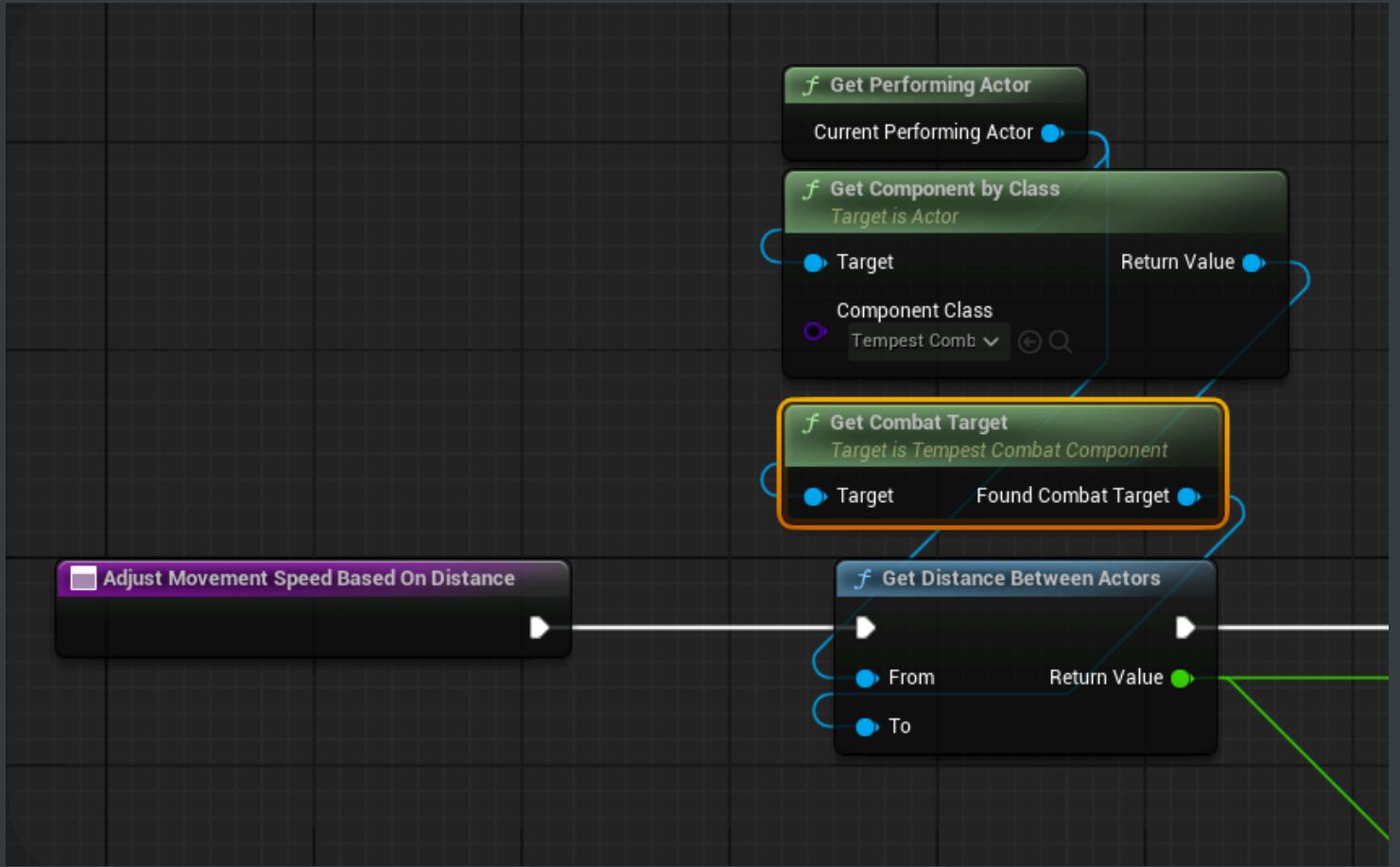
进入FollowingState,在PreStateActivation,

首先在UpDateFollowingBegavior中将配置的行为树数据绑定起来，这个数据主要判断距离以及各个档位的速度

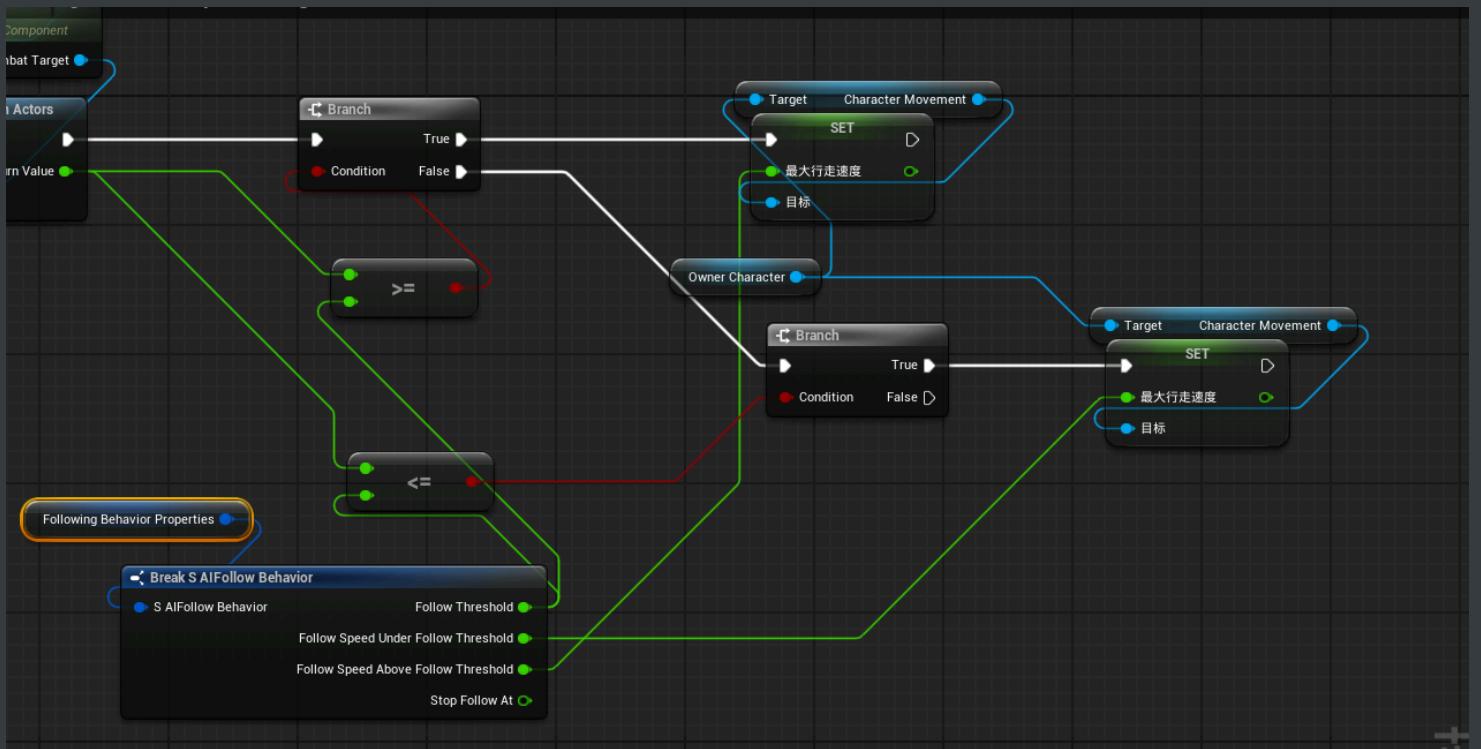


之后adjust movement speed based on distance,根据距离调整速度

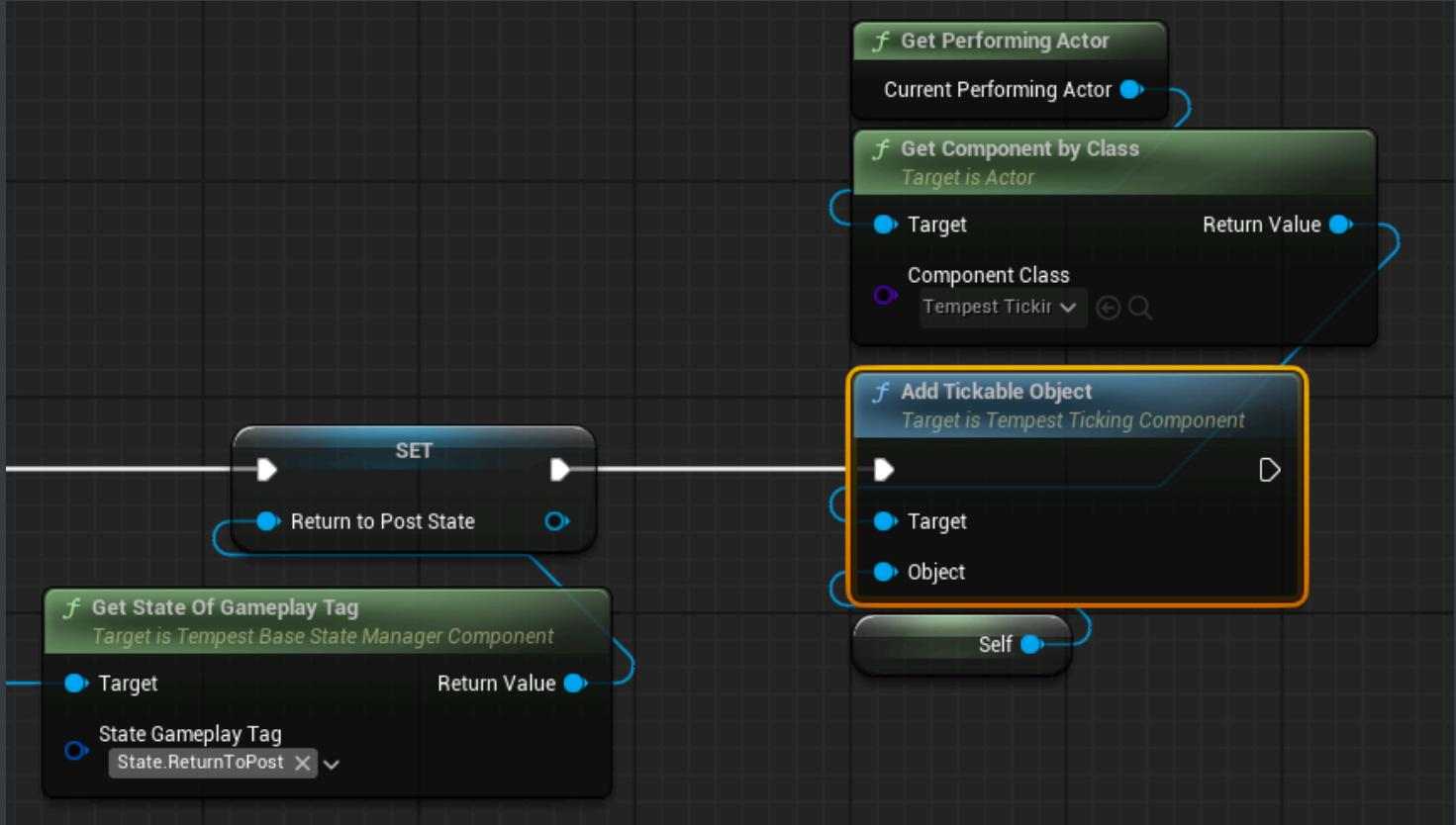
获取玩家和NPC之间的距离



然后根据距离来设置你的速度

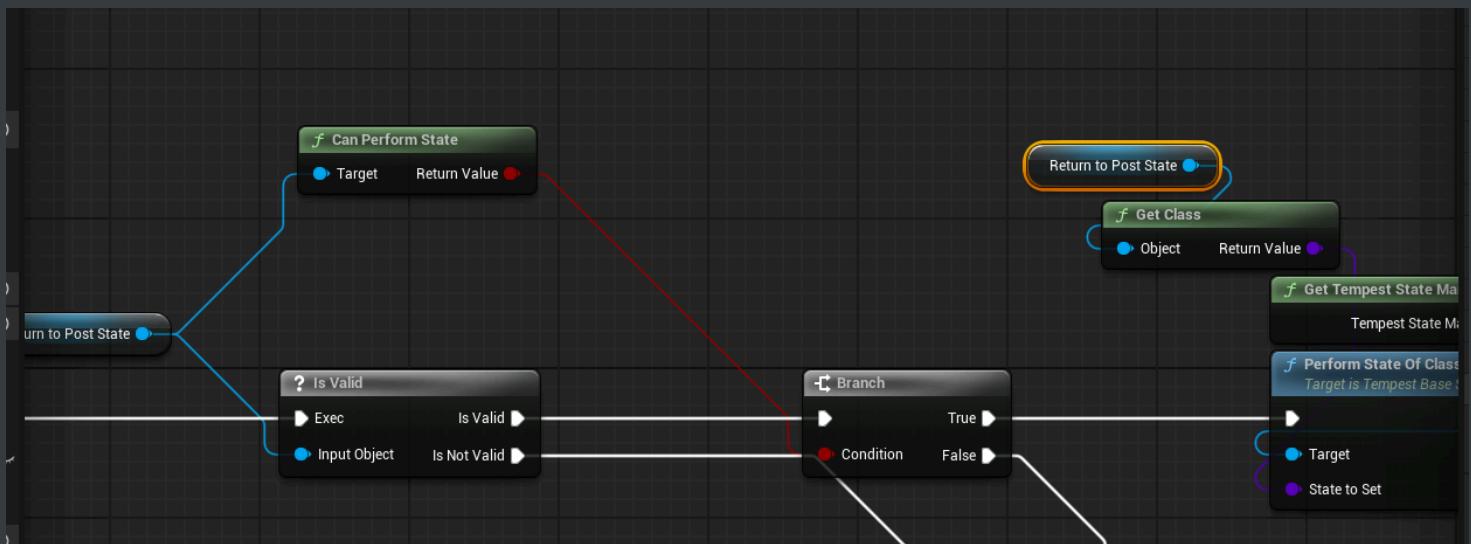


然后进入巡逻状态，里面就设置当前状态，然后将该状态设置为可Tick，然后再Tick里面判断逻辑



Tick的逻辑首先依旧是根据距离判断速度

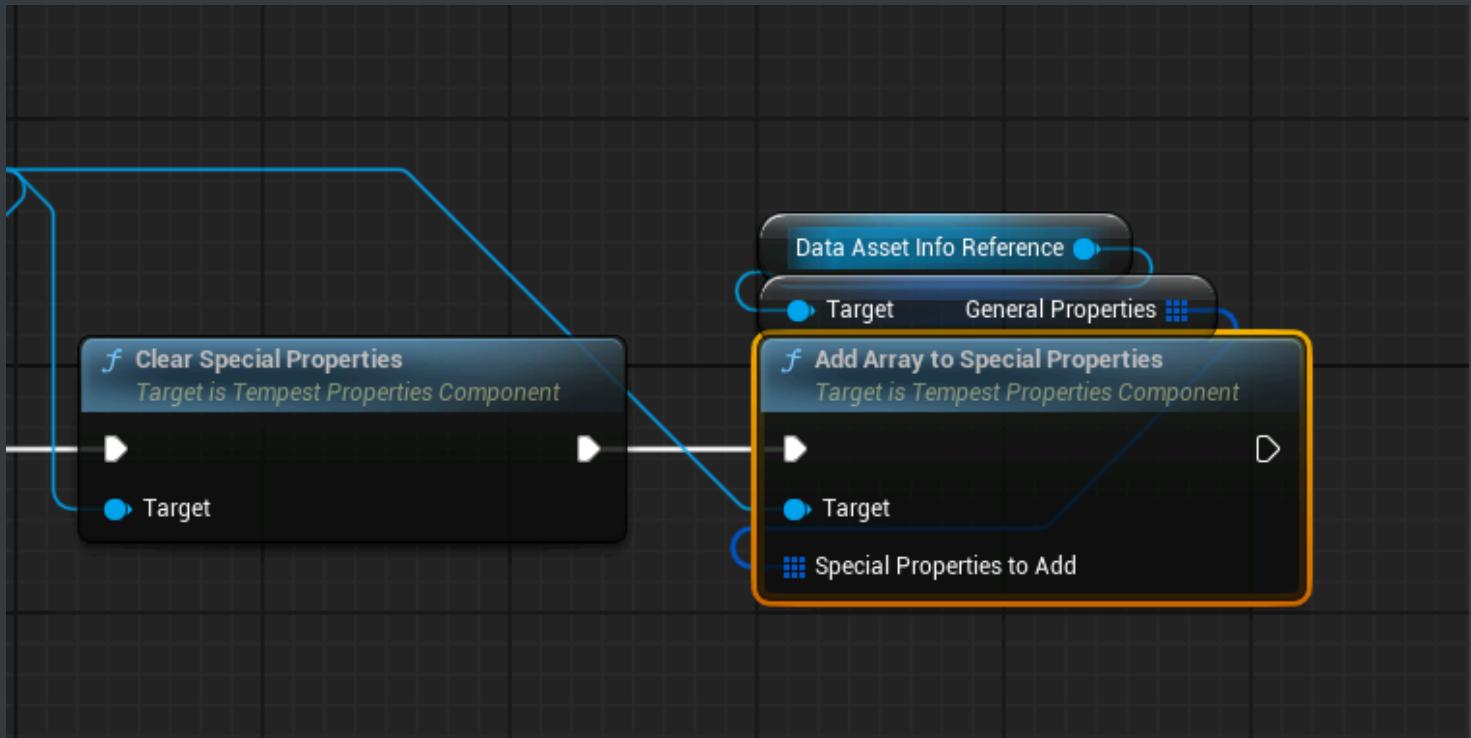
然后判断能不能继续following



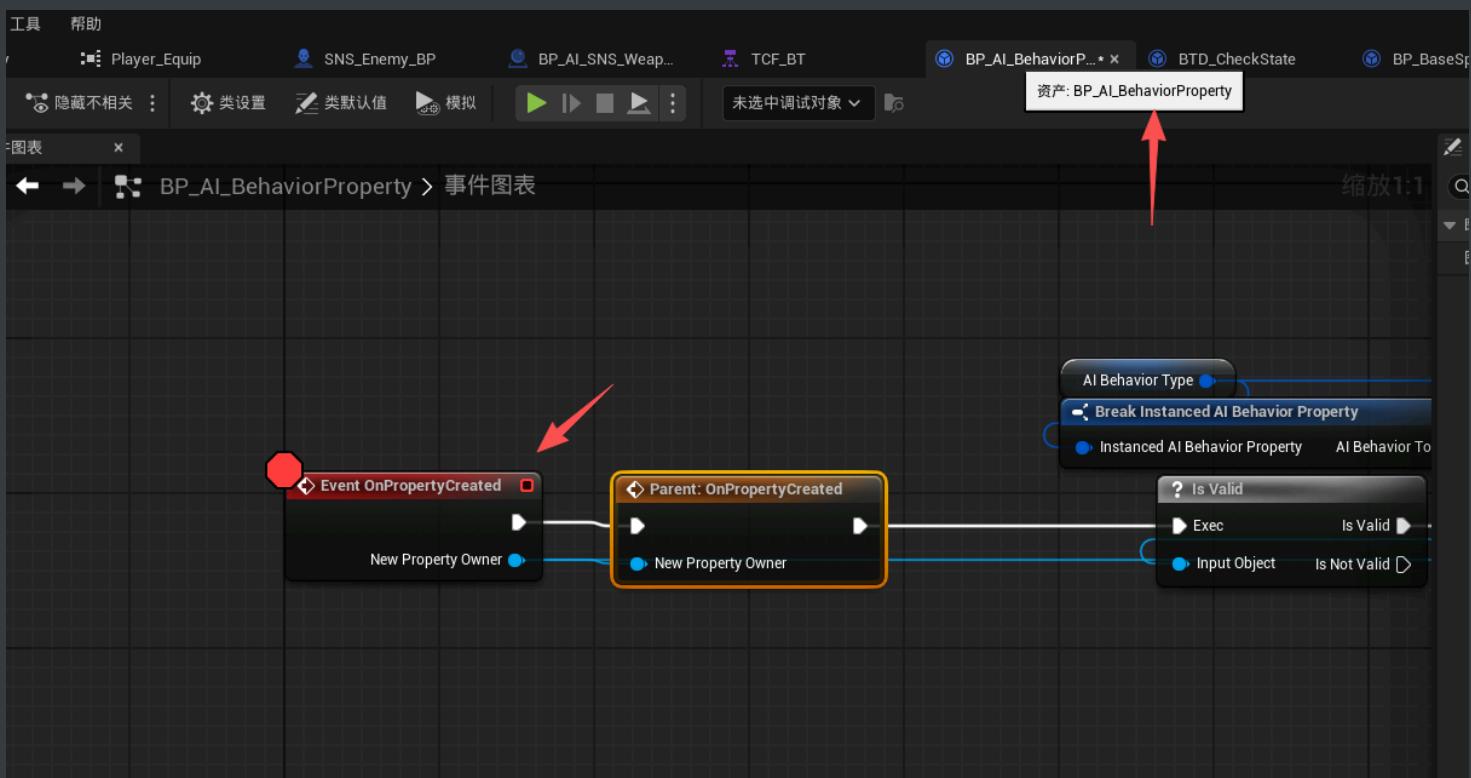
然后如果距离到了可以停止的距离则进入DecideNextSatem,这里面就是尝试进入攻击状态。如果不行就继续是追踪状态

追踪的行为树

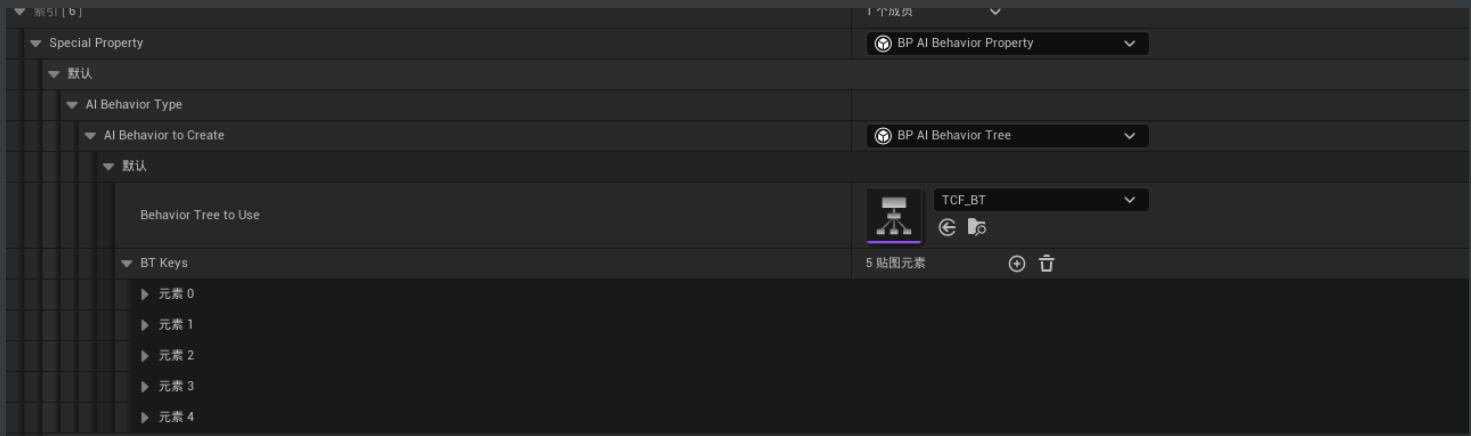
在捡到武器的时候，会将数据资产里面的所有配置创建出来



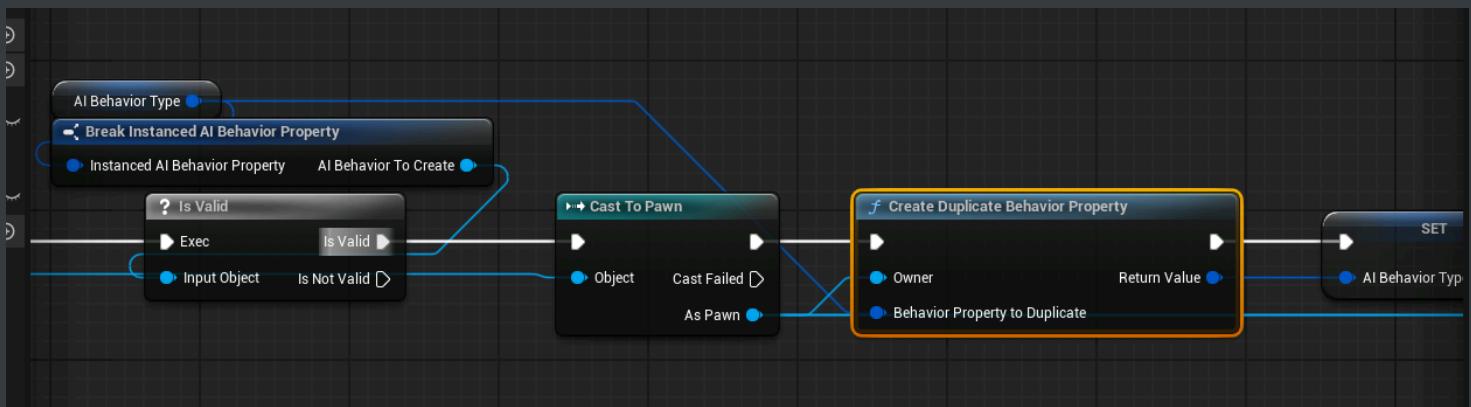
这个属性会在被创建的时候有特殊调用



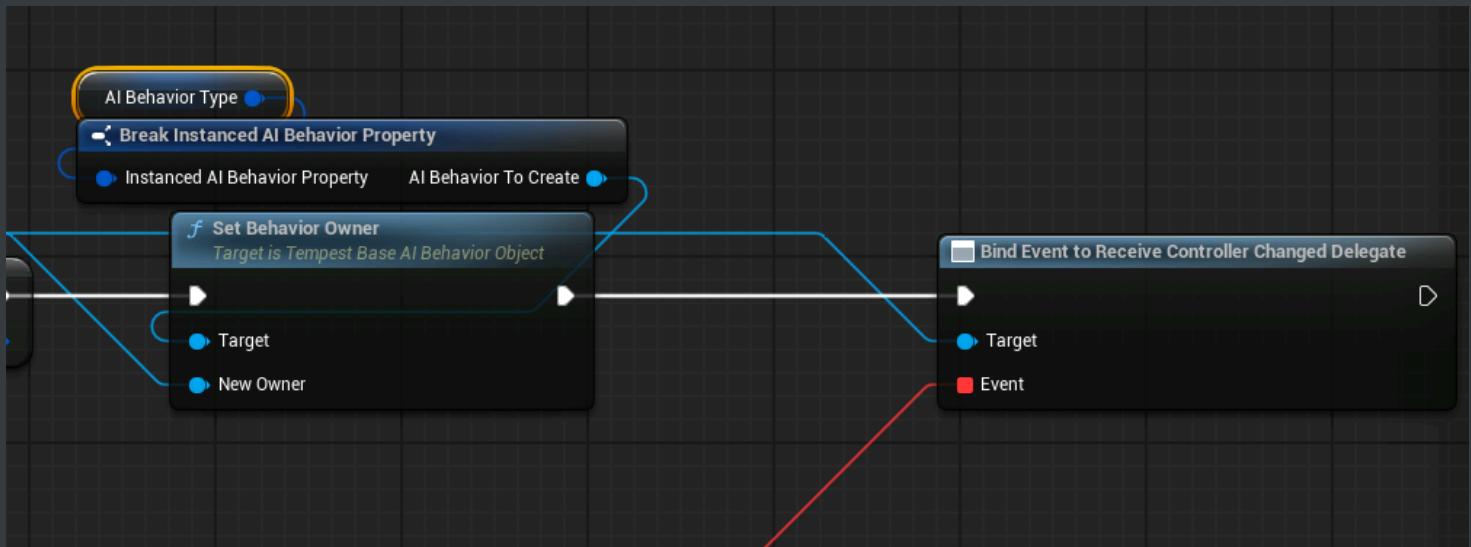
首先这个特性里面存了，一个行为树，以及一个行为树蓝图

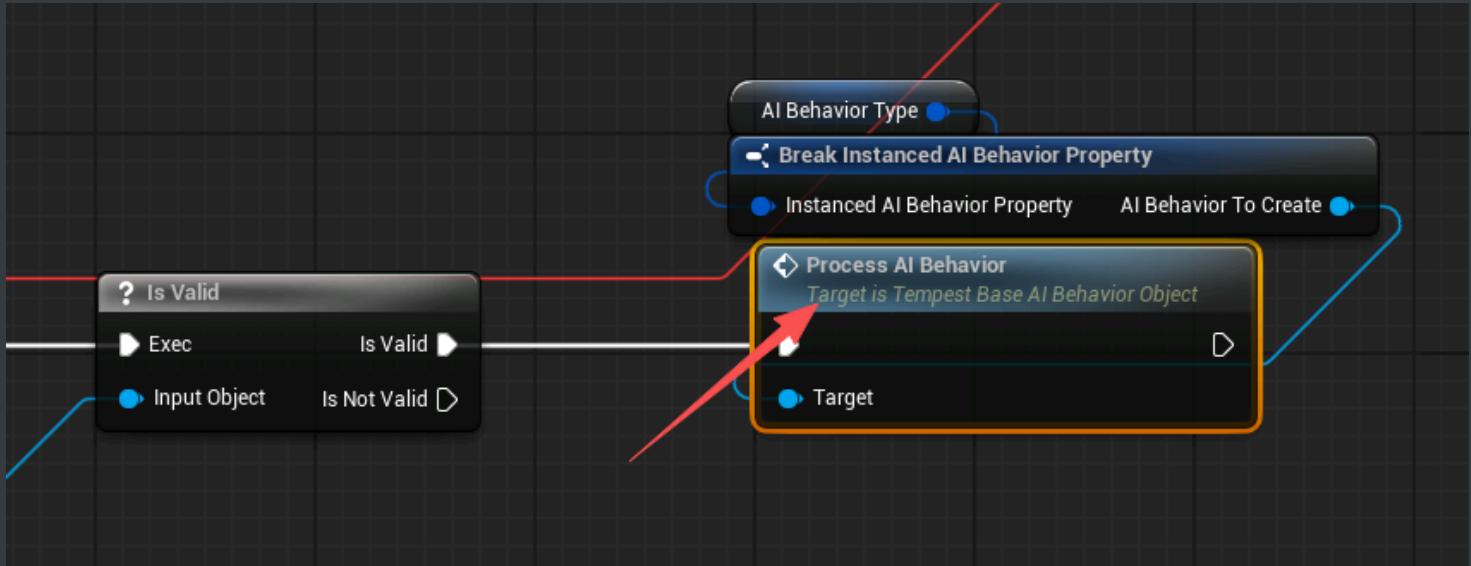


将这段数据深拷贝起来

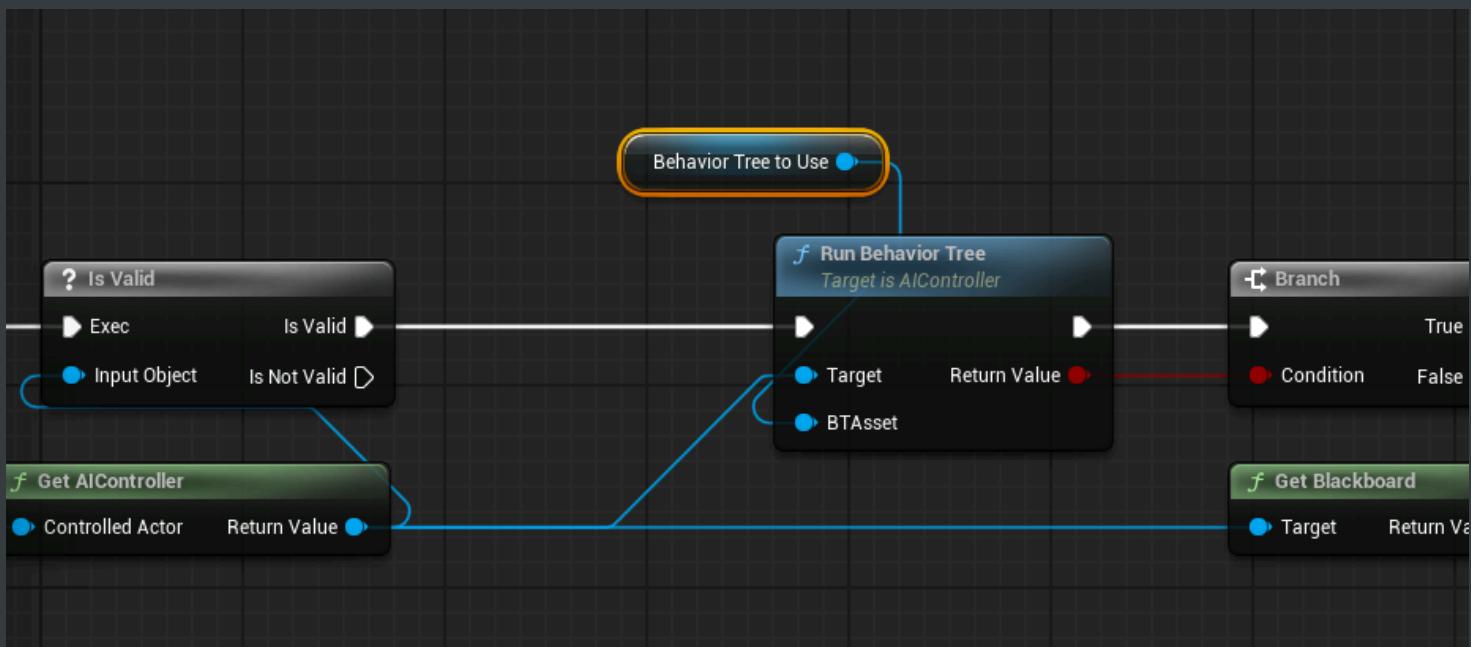


然后调用行为树蓝图的方法

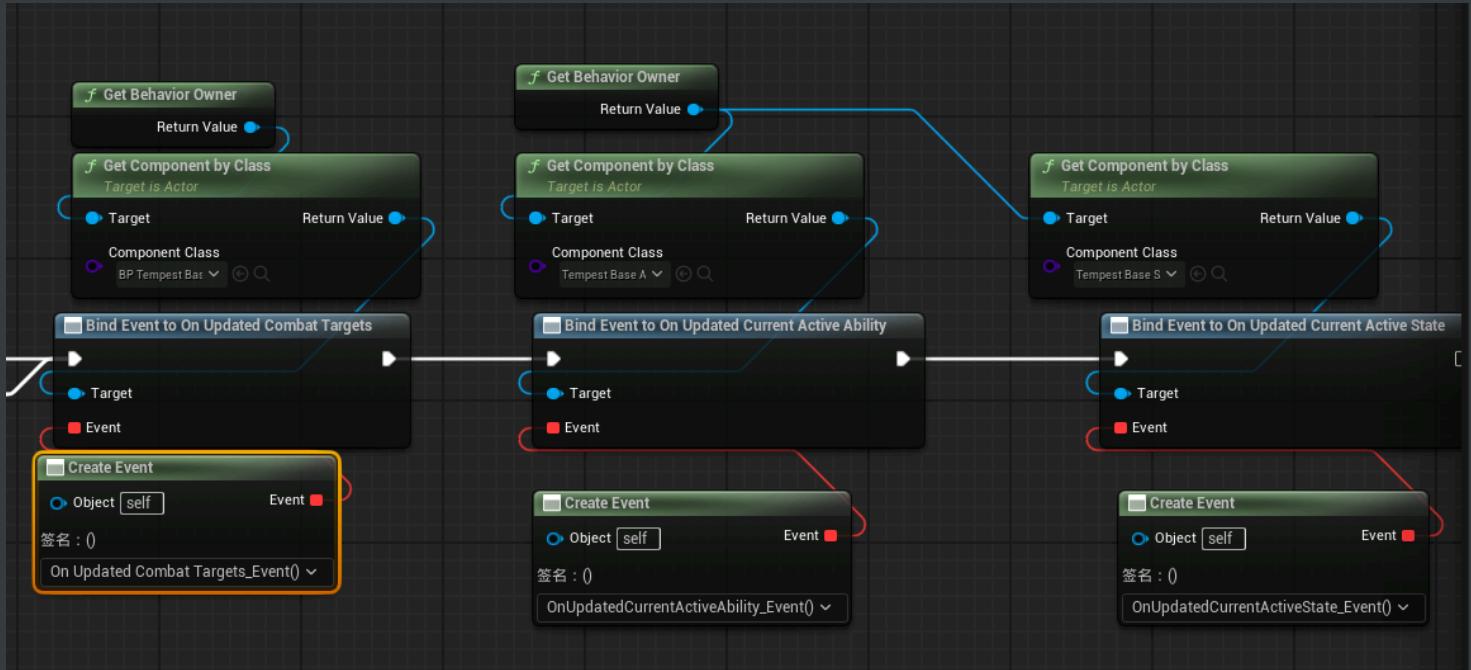




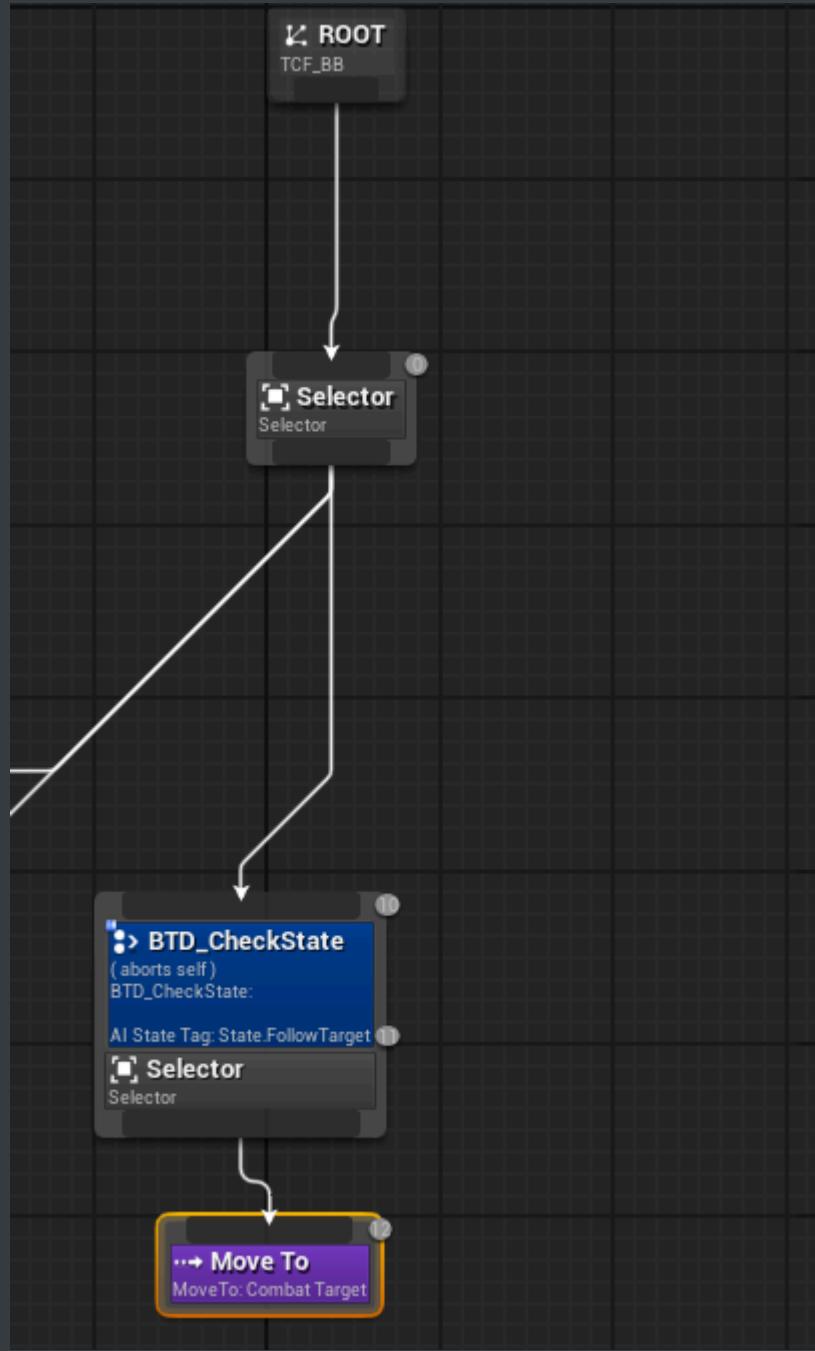
这个方法会运行行为树



然后黑板键的变化也在这里注册一个监听



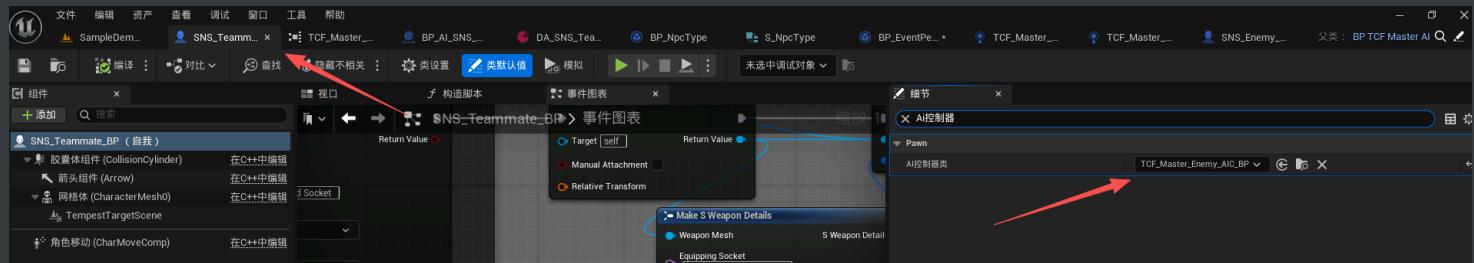
行为树里写的很简单，就是move to



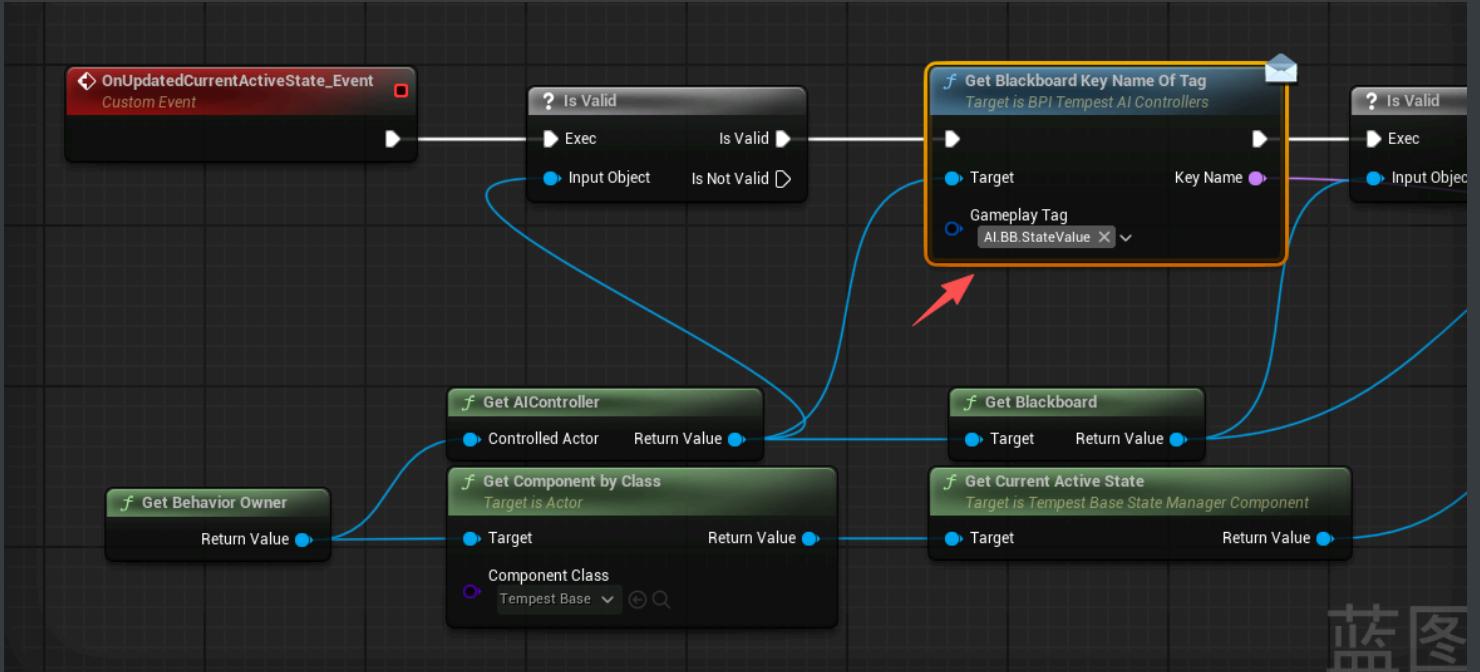
感知获取的实现

配置

每个角色有一个AI控制器类，这里面有有关于感知的操作



这个方法会将你传进去的Tag转化为黑板里面的键名

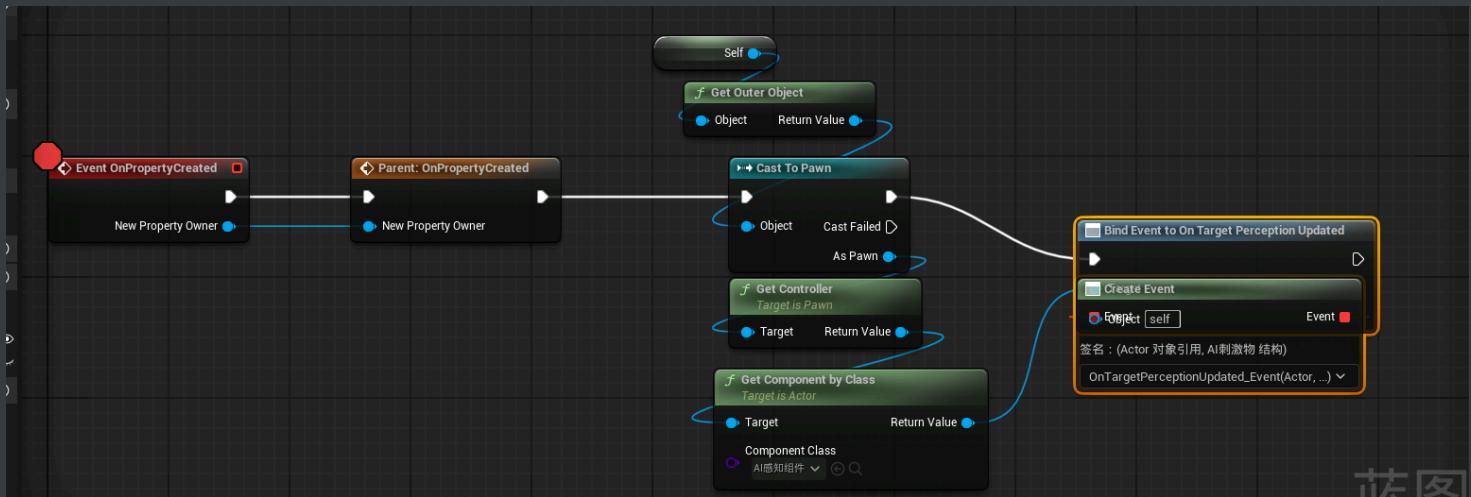


这个Tag也是在表里添的

索引 [7]	成员
Special Property	BP AI Behavior Property
默认	
AI Behavior Type	
AI Behavior to Create	BP AI Behavior Tree
默认	
Behavior Tree to Use	TCF_Teammate_BT
BT Keys	5 贴图元素
元素 0	键 (Gameplay Tag): AI.BB.SelfActor 值 (Name): SelfActor
元素 1	键 (Gameplay Tag): AI.BB.CombatTarget 值 (Name): Combat Target
元素 2	键 (Gameplay Tag): AI.BB.StateValue X 值 (Name): State Value
元素 3	键 (Gameplay Tag): AI.BB.AbilityValue X 值 (Name): Ability Value
元素 4	键 (Gameplay Tag): AI.BB.Destination X 值 (Name): Destination
Special Property Base Variables	

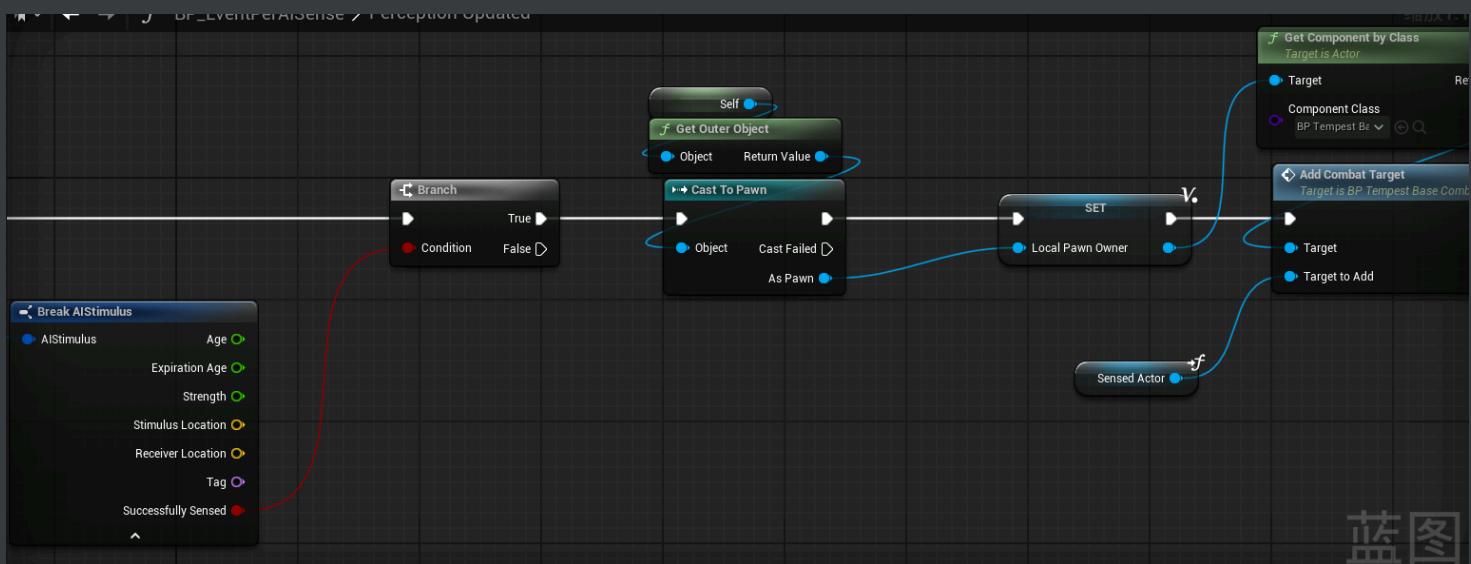
怎么获取感知内容

这里注册了一个感知事件，后面绑定的事件是Ai感知自带的接口，当目标感知信息更新时会广播



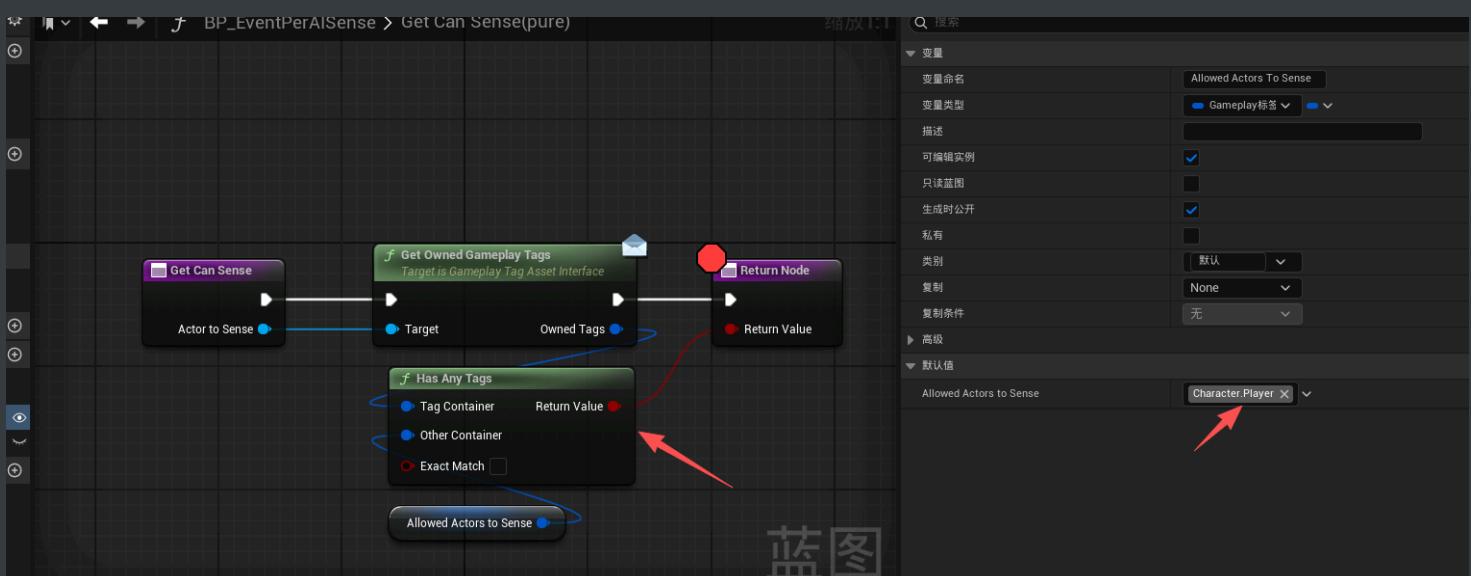
蓝图

然后将目标添加到战斗目标



蓝图

判断感知的类型实在这里，只有这个标签的才会被感知，有点变态了，要大修一下



蓝图

实际项目

天赋树

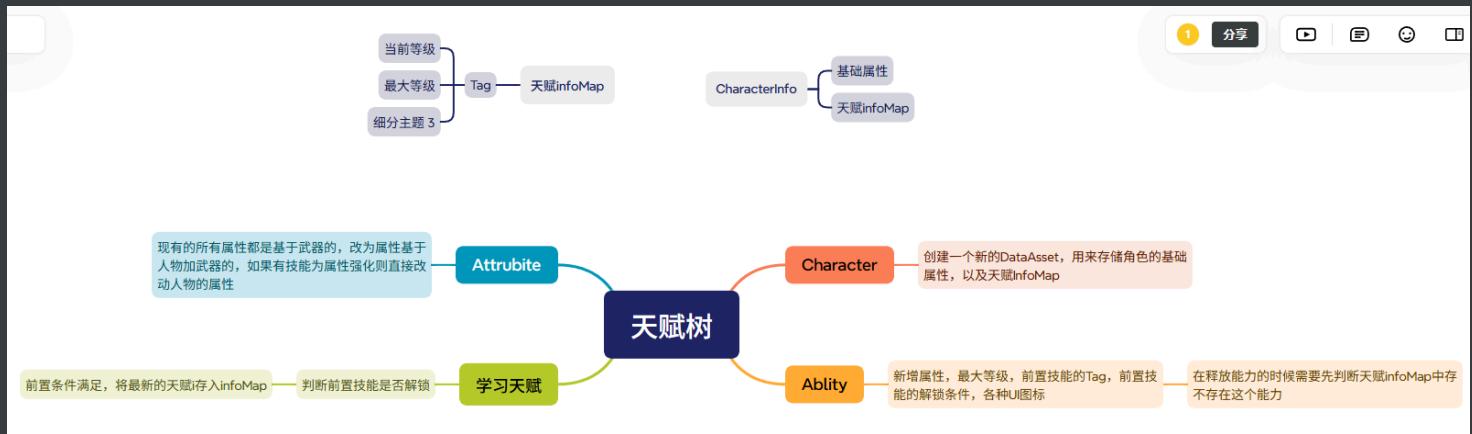
思维导图:

现在玩家属性和能力完全由武器提供,也就是不捡起武器,玩家将没有任何属性和能力,所以需要做出修改

优先看能不能通过覆写组件来实现,不行在新加

逻辑层面分为四个部分

- 角色:首先需要新增一个数据资产CharacterInfo,专门用来存储玩家的基础属性,以及一些特性,对于角色树来说,我需要存储一个字典Key为技能Tag,value为技能的等级.代表玩家当前拥有的技能.
- 能力:首先现在能力的属性比较少,需要新增技能的最大等级,前置技能的Tag以及等级,以及各种UI图标,以及在释放技能时需要判断天赋数中是否存在这个技能,其余不变.
- 属性:在UTempestBaseAttributeObject中添加新的结构体FAttributeToAbilities, 里面添加新的
- 学习天赋: 当天赋满足学习条件时, 更新Character Info



实现:

UI显示

美女系统

美女的AI比起敌人AI需要更多状态来驱动, 从逻辑上将美女就是一个不能操控的玩家, 也就是玩家有的状态, 美女理应也有这个状态

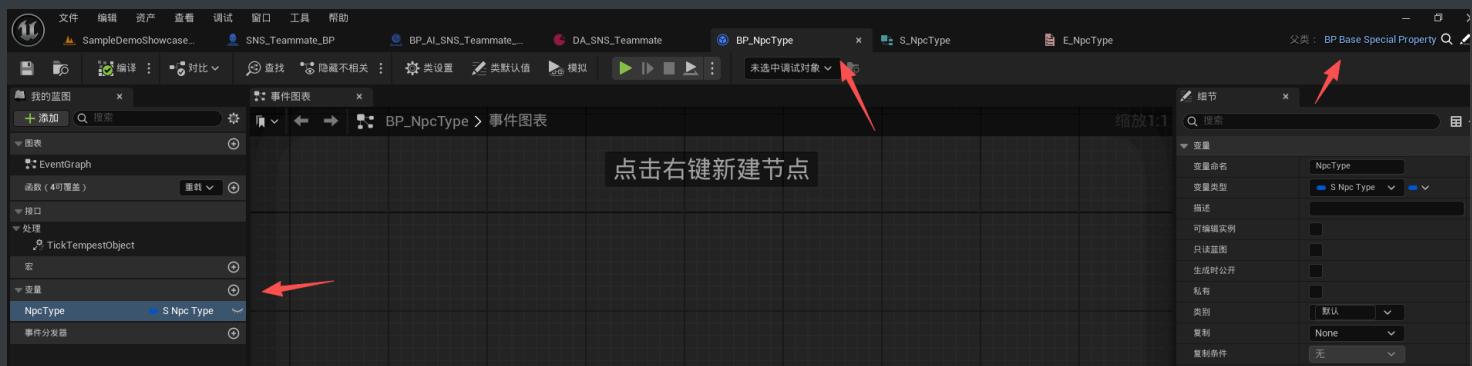
根据这套框架来写，可能写的位置和逻辑都不太对，但是功能的是要都实现的，可以适当借鉴一下吧

制作一个友军

目前没有友军设定，所以新建一个枚举代表队友敌人中立



在键一个特殊属性蓝图，添加到资产中，这样就有了基本的配置，具体实现可能还有在具体的功能蓝图里面实现



在数据资产中进行配置



美女属性行为

美女基础行为

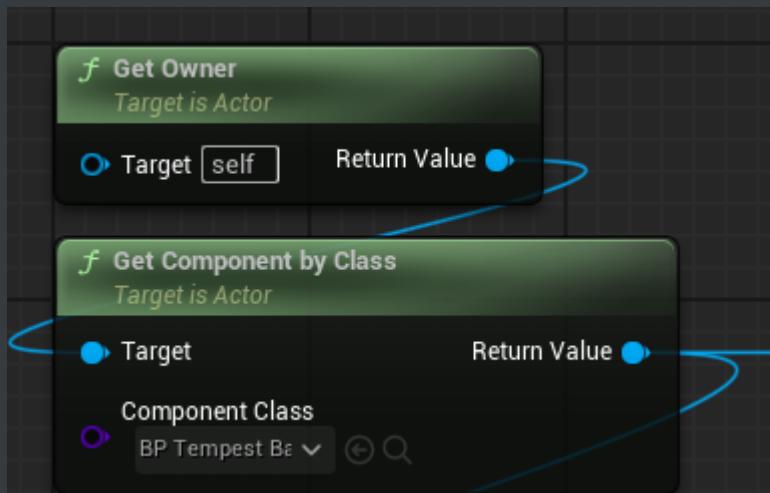
美女战斗行为

研究

武器系统集成大部分属性和能力的优缺点

优点:

- 获取组件和属性更加方便，因为所有的都集成在一起了你想获取某个组件，直接按照下面选择你想要获取的组件就可以获取



- 所有的数据也都在一起方便保存和读取

缺点:

多装备冲突

不支持多装备，因为他每次装备都会清除属性，比如左手剑右手盾这种情况会因为清除属性只会使用最后装备的属性，以及衣服防具这类装备也会和武器冲突，但如果像只狼那样就一把太刀一套衣服一直玩的话这个问题到不大

配置繁琐且有问题(这个问题可能是因为我自身)

配置很复杂很复杂，策划进行配置的话会有难度，因为我都要看晕了，就拿配置技能来举例，我要在众多没有提示的路径里面找到GeneralProperties的索引5然后在这个里面的众多状态找到玩家的攻击状态然后再里面的Abilities里面，添加自己的技能

Behavior Info

General Properties	
索引 [0]	1 个成员
索引 [1]	1 个成员
索引 [2]	1 个成员
索引 [3]	1 个成员
索引 [4]	1 个成员
索引 [5]	1 个成员
Special Property	
默认 Abilities Per State BP_CharacterIdleState BP_PlayerWalkingState BP_PlayerJumpingState BP_PlayerSprintingState BP_PlayerFallingState BP_PlayerDeathState BP_CharacterHitState BP_CharacterEquipState BP_CharacterAttackingState Abilities and Gameplay Tags Array 索引 [0] Abilities Tag Abilities 索引 [0]: BP_PlayerAssassinationAttackAbility 索引 [1]: BP_PlayerExecutionAttackAbility 索引 [2]: BP_PlayerSpecialAttackAbility 索引 [3]: BP_PlayerLightAttackAbility	

但是到这一步并不算完,因为你会发现你只是把这个技能配进来了,但技能的属性在哪里配置?

技能的基类里面有做出属性的声明

```

class TEMPESTCOMBATFRAMEWORK_API UTempestBaseAbilityObject : public UTempestBaseObject
{
    GENERATED_BODY()
public:
    UTempestBaseAbilityObject();

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ability Properties")
    FGameplayTag AbilityGameplayTag; ①已在 42 个蓝图中更改

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ability Properties")
    bool bHasCooldown; ②未在资源中更改

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ability Controls", meta = (EditCondition = "bHasCooldown"))
    float CooldownDuration = 0.f; ③未在资源中更改
  
```

```

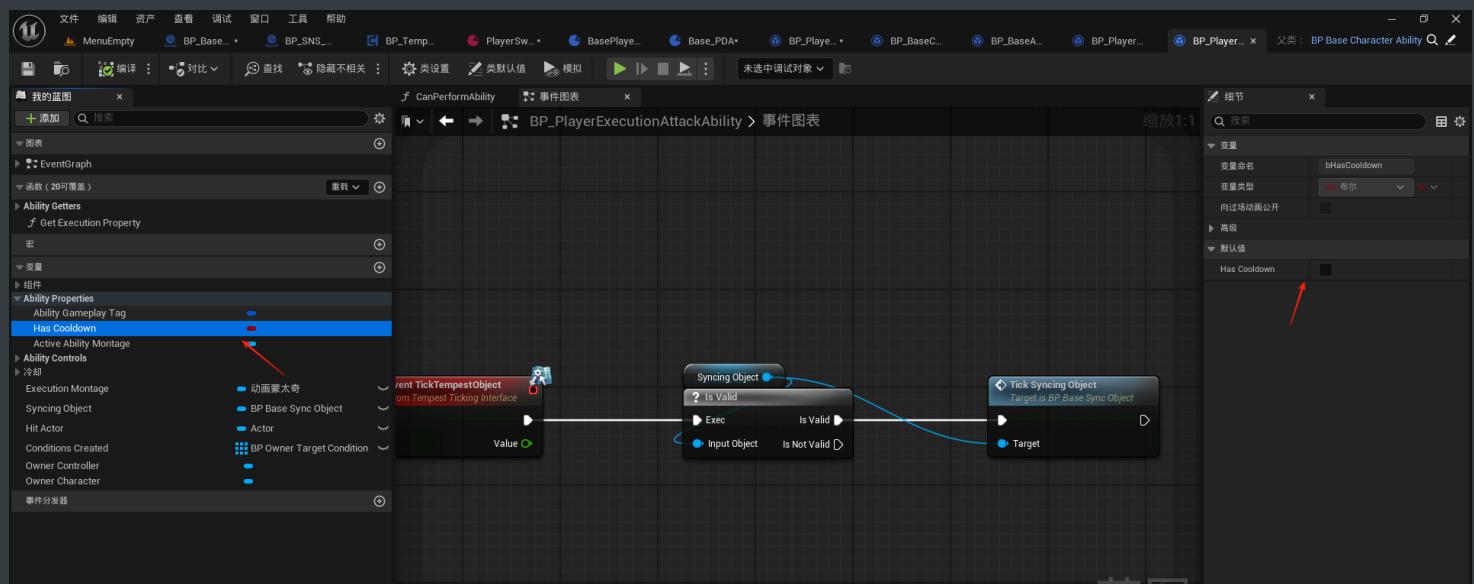
UPROPERTY(BlueprintReadWrite, Category = "Ability Properties")
class UAnimMontage* ActiveAbilityMontage; ④未在资源中更改

UPROPERTY()
AActor* PerformingActor;

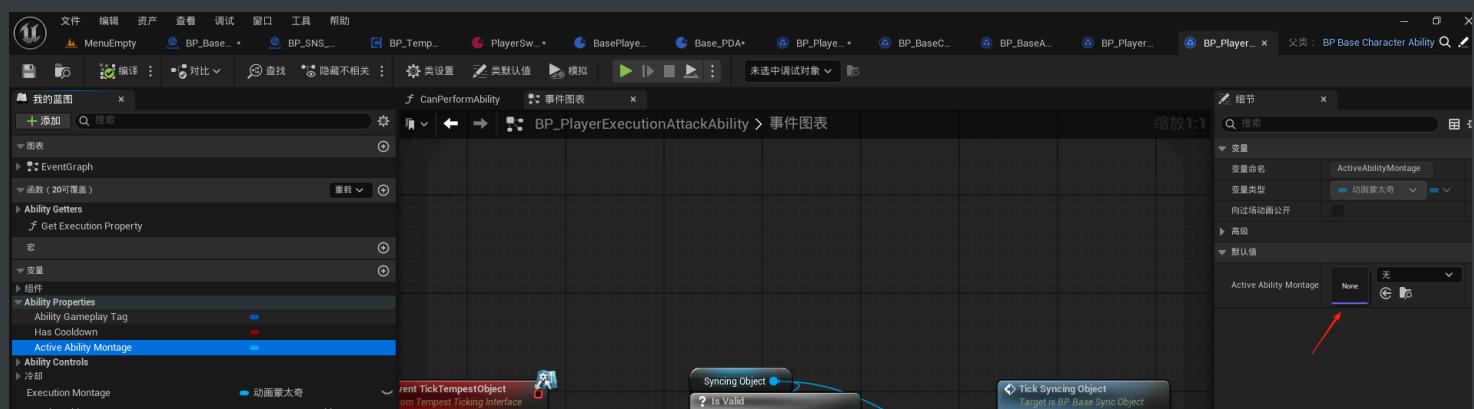
UPROPERTY(BlueprintReadWrite, Category = "Cooldown")
bool bAbilityOnCooldown; ④未在资源中更改

```

接下来才是变态的,技能配置,目前技能拥有的属性较少,后期可能还得加属性比如等级之类,目前的属性配置在两个地方配置第一个是在你的技能蓝图里面的变量配置,没有读表这一步,策划想改点cd什么的需要再程序的蓝图里面修改,应该是不合理的



其次他也不是全在蓝图里面配置,有些属性还是在数据资产里面配置



就比如你的蒙太奇动画，需要在索引四里面找到对应的进行配置

The screenshot shows the Behavior Editor's General Properties section for a skill node. It has five indices (0-4) each pointing to a BP Montages Per Ability Property. Index 4 is highlighted with a red arrow. The BP Montages Per Ability Property for index 4 points to the BP_MontagesListPerAbility class. Underneath, the Montages List Per Ability section shows three entries: BP_PlayerNormalDeathAbility, BP_PlayerNormalHitAbility, and BP_PlayerLightAttackAbility. The BP_PlayerLightAttackAbility entry is expanded, showing its Montages and Gameplay Tags Array. This array contains three indices (0-2), each pointing to a Montage asset. The Montage asset for index 0 is highlighted with a red arrow and is named swordandshieldslash_SNS_Montage. The Montage assets for indices 1 and 2 are also listed below it.

这整个技能的配置逻辑给我的感觉就是既想要解耦但是又耦合在一起，很奇怪。

对于技能树来说

这是我做的demo里面的属性配值，有些属性，比如技能所在的技能树节点，前置条件，等级，技能描述，

其中像前置条件，技能所在节点，技能的最大等级，数值，以及技能描述这些都是和策划相关的内容，

但现在我如果在旧的基础上改就会有两个疑问我是单启一个类还是就在技能的基础属性里面改

在基础里面改的话策划就配置不了这个技能，

我预想的改动是整个数据资产里面的数据是只有程序能修改，因为数据资产既可以读也可以保存，然后所以需要策划配置的根据不同的模块分成不同的表

结构

默认值

结构

提示文本

TalentID	整数	文本
TalentName	文本	整数
TalentMaxLevel	整数	文本
TalentTreeNode	文本	整数
TalentType	整数	文本
TalentDescription	文本	整数
TalentAbilityDescription	整数	文本
TalentCorrespondingToSkill	文本	整数
TalentLockImage	纹理2D	纹理2D
PreTalentNodeLevel	字符串	整数
TalentNonStudyTexture	纹理2D	纹理2D
TalentHasStudyTexture	纹理2D	纹理2D

行名	TalentID	TalentName	TalentMaxLevel	TalentTreeNode	TalentType	TalentDescription	TalentAbilityDescription	TalentCorrespondingToSkill	TalentLockImage	PreTalentNodeLevel	TalentNonStudyTexture
1	Attack111	小攻击	5	1,1,1	1	提升角色的攻击力	(1" = 提升攻击力 (1" = 一级提升攻击力, "2" = 二级提升攻击力, "3" = 三级提升攻击力)	810001	None	0	/Script/Engine.Texture2D/Game/Talent/UV/职业未选中_职业未选中
2	SecondLeft	小防御	3	1,1,2	2	提升角色的防御力	(1" = 一级提升防御力, "2" = 二级提升防御力, "3" = 三级提升防御力)	910001	None	("Attack111" + 5)	/Script/Engine.Texture2D/Game/Talent/UV/招格未选中_招格未选中
3	SecondRight	小生命	3	2,1,2	2	提升角色的生命值	(1" = 一级提升生命值, "2" = 二级提升生命值, "3" = 三级提升生命值)	910002	None	("Attack211" + 3)	/Script/Engine.Texture2D/Game/Talent/UV/柱石未选中_柱石未选中
4	LastTalent	最终技能	1	1,1,3	3	提升角色的攻击力	(5" = 获得大量攻击力提升)	910001	None	("SecondRight" + 3, "SecondLeft" + 3)	/Script/Engine.Texture2D/Game/Talent/UV/谋略未选中_谋略未选中
5	Attack211	中级攻击	5	2,1,1	1	中幅提升角色的攻击力	(1" = 一级提升攻击力, "3" = 三级提升攻击力, "5" = 五级提升攻击力)	810001	None	0	/Script/Engine.Texture2D/Game/Talent/UV/职业未选中_职业未选中
6	Attack213	大攻击	5	2,1,3	1	大幅提升角色的攻击力	(1" = 一级提升攻击力, "2" = 二级提升攻击力, "3" = 三级提升攻击力, "4" = 四级提升攻击力)	810001	None	("SecondRight" + 3)	/Script/Engine.Texture2D/Game/Talent/UV/职业未选中_职业未选中