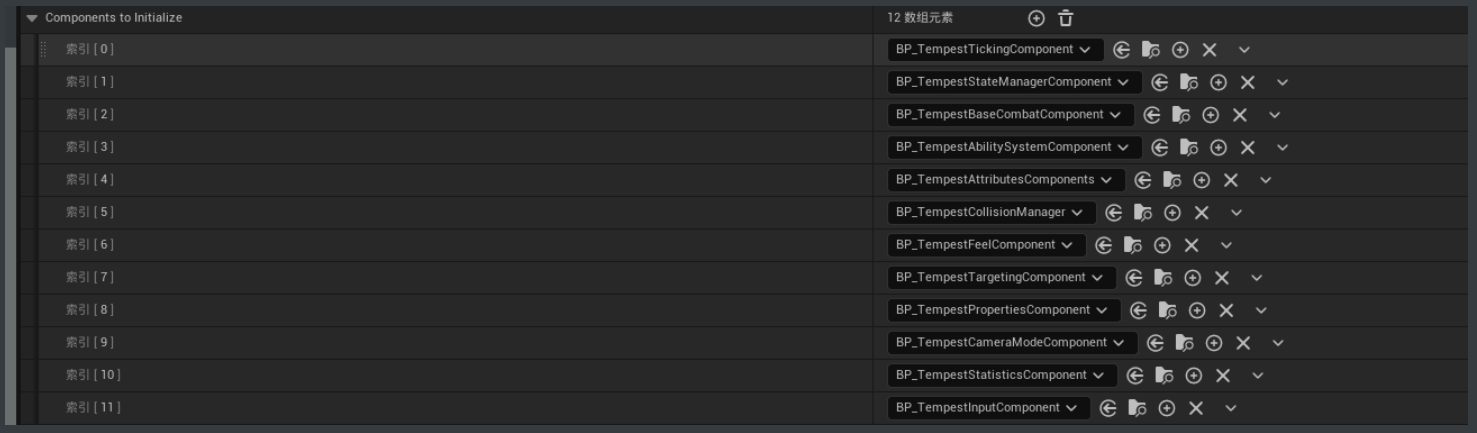
数据资产配置--组件

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



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

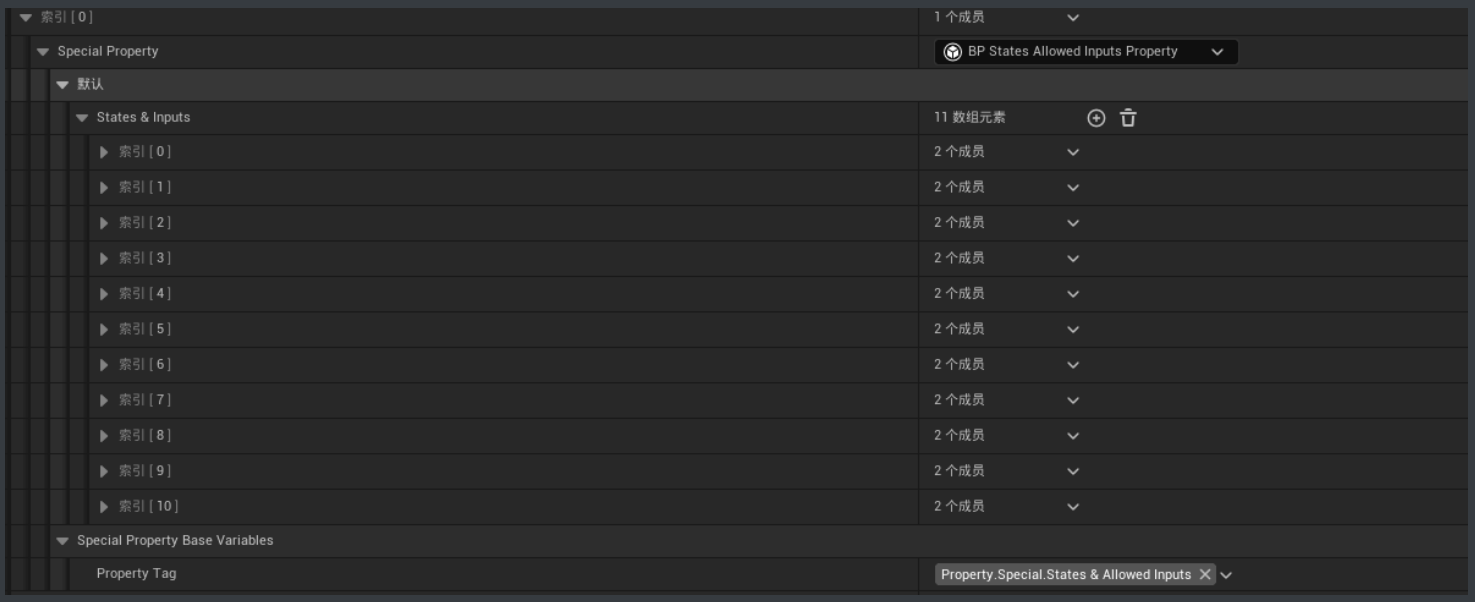
BP_TempestPropertiesComponent

基类为C++中的UTempestPropertiesComponent，负责角色所拥有的特性，详情看 UTempestPropertiesComponent

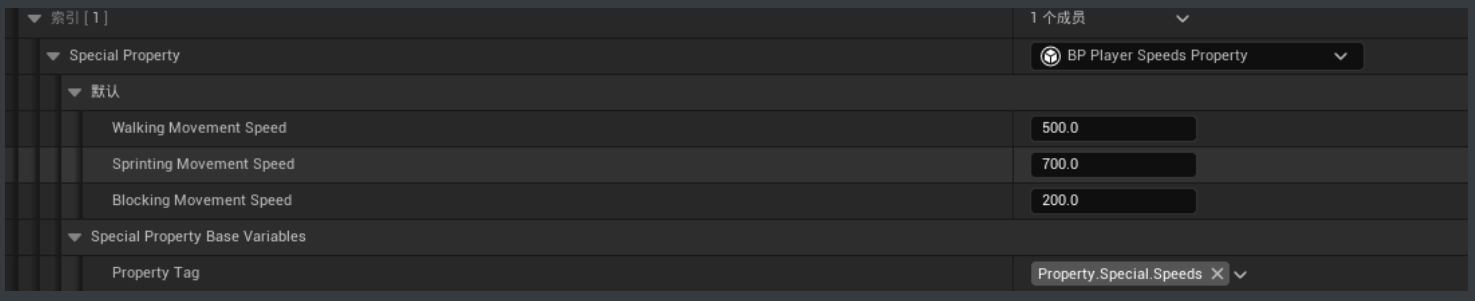
数据资产配置--特性（GeneralProperty）

允许的状态输入特性

通过输入来驱动状态和能力



玩家速度特性

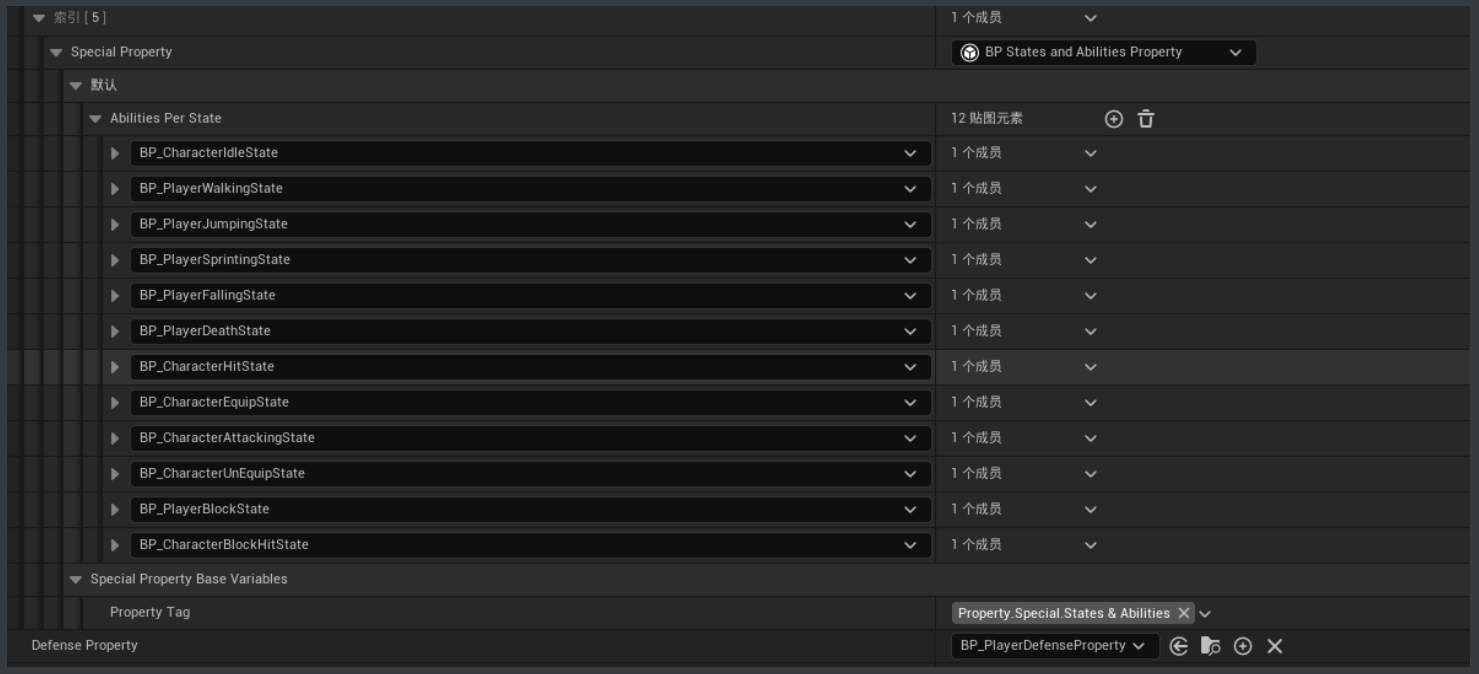


特殊能力的蒙太奇配置

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

状态和能力特性

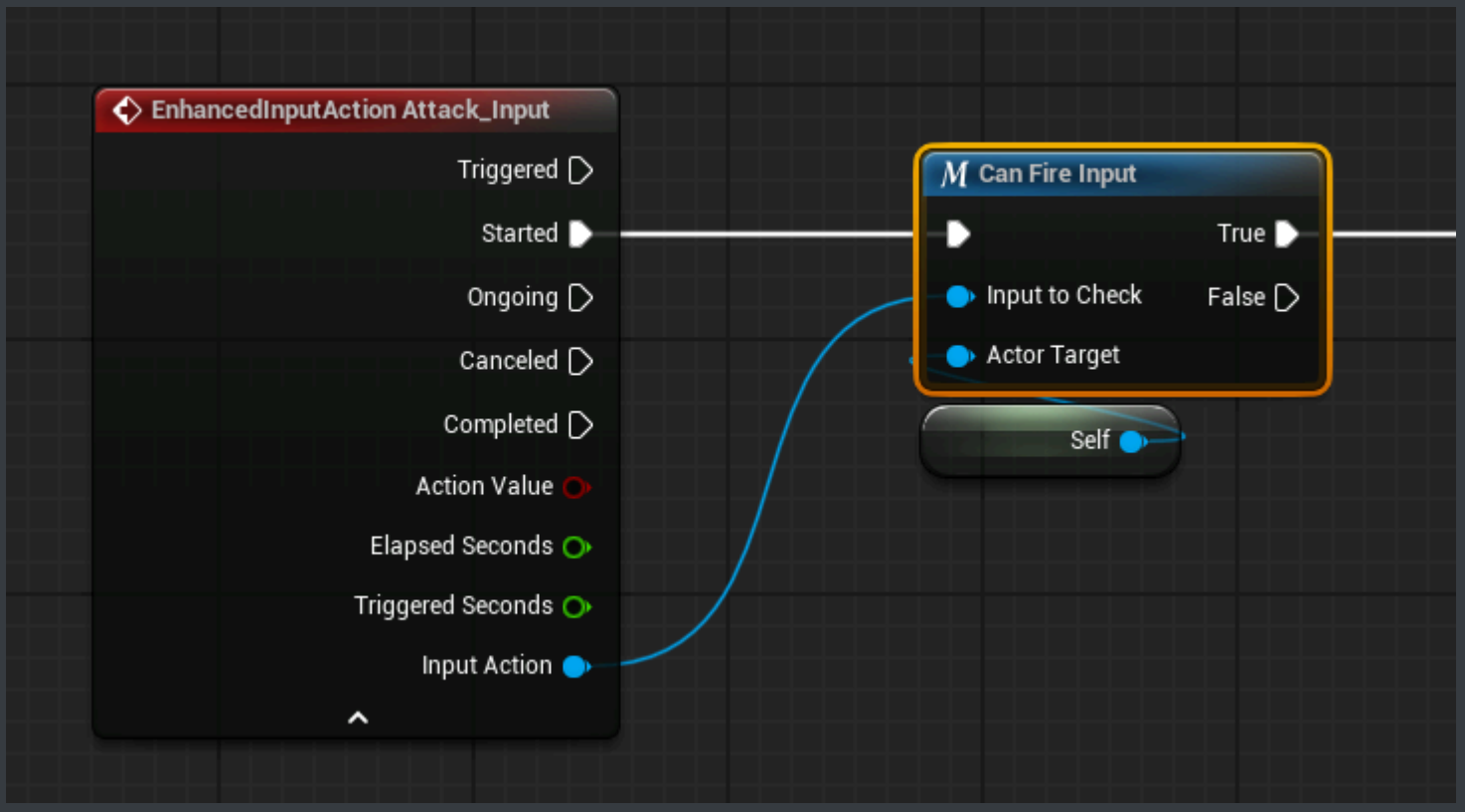
每一个这个武器或者角色可能拥有的状态都配置出来



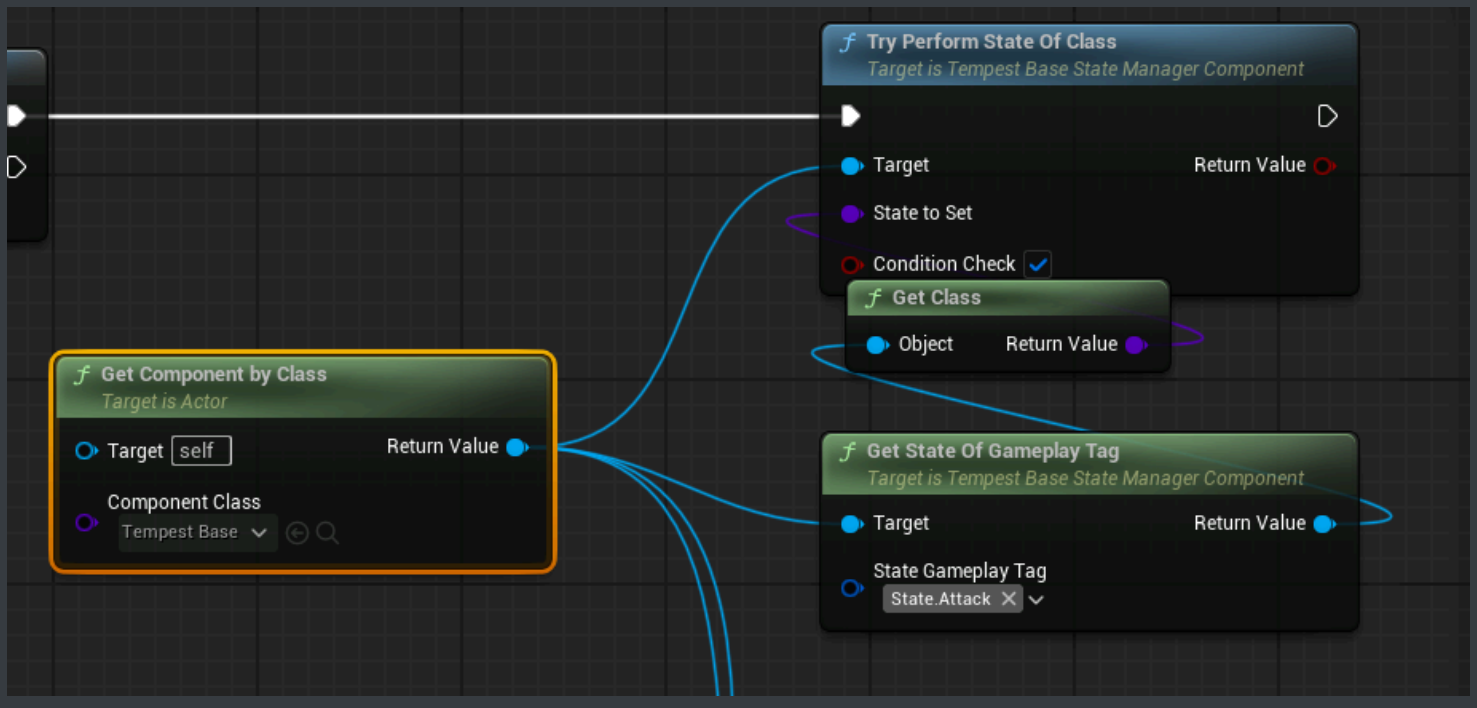
一个攻击的流程示意(将这几个组件串起来):

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

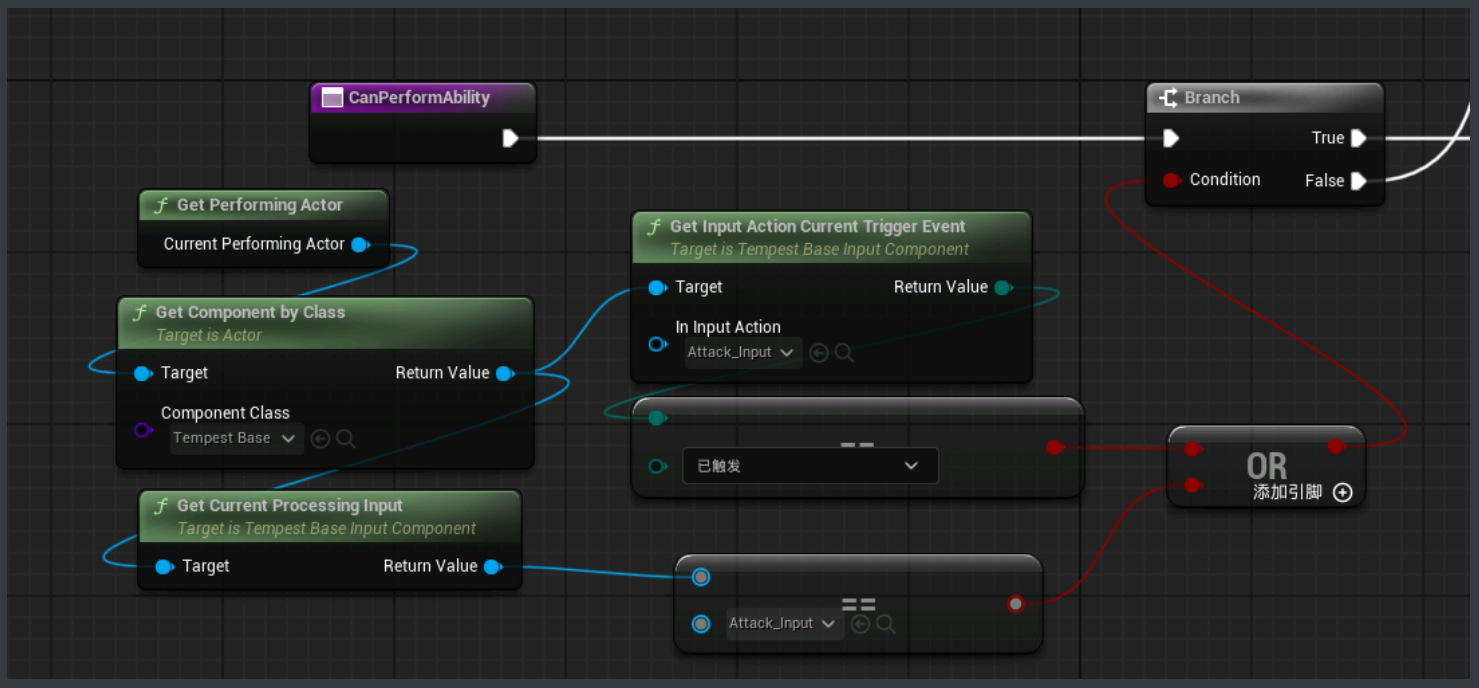
```
void UTempestBaseStateManagerComponent::ConstructStateOfClass(TSubclassOf<UTempestBaseStateObject> Sta
{
    ConstructedState = nullptr;
    if (StateToConstruct)
    {
        UTempestBaseStateObject* LocalNewState;
        LocalNewState = NewObject<UTempestBaseStateObject>( Outer: GetOwner(), StateToConstruct);

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

        LocalNewState->ConstructState();
        ConstructedState = LocalNewState;
    }
}
```

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
{
    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,

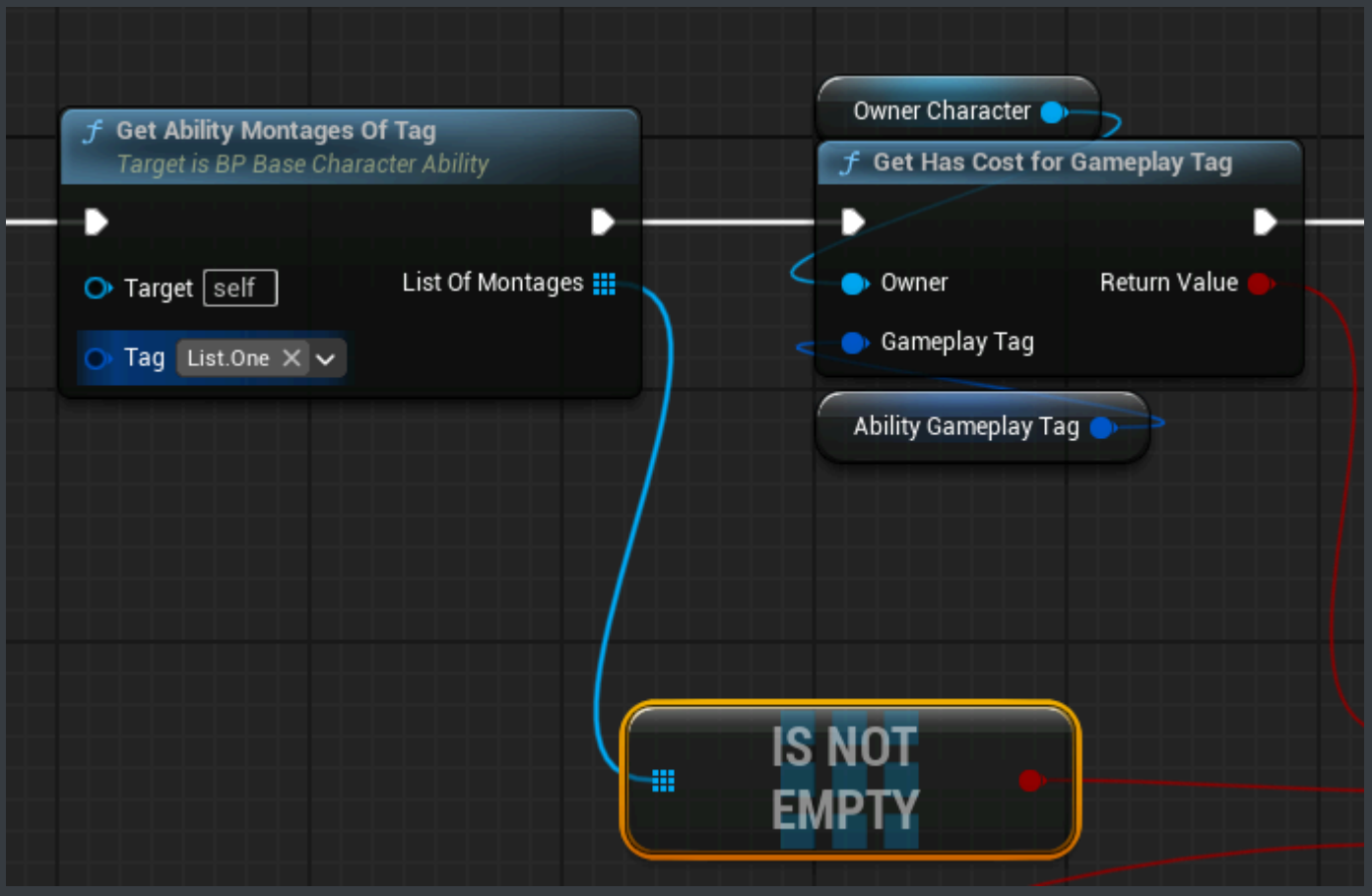
首先判断条件,是否触发了攻击按键



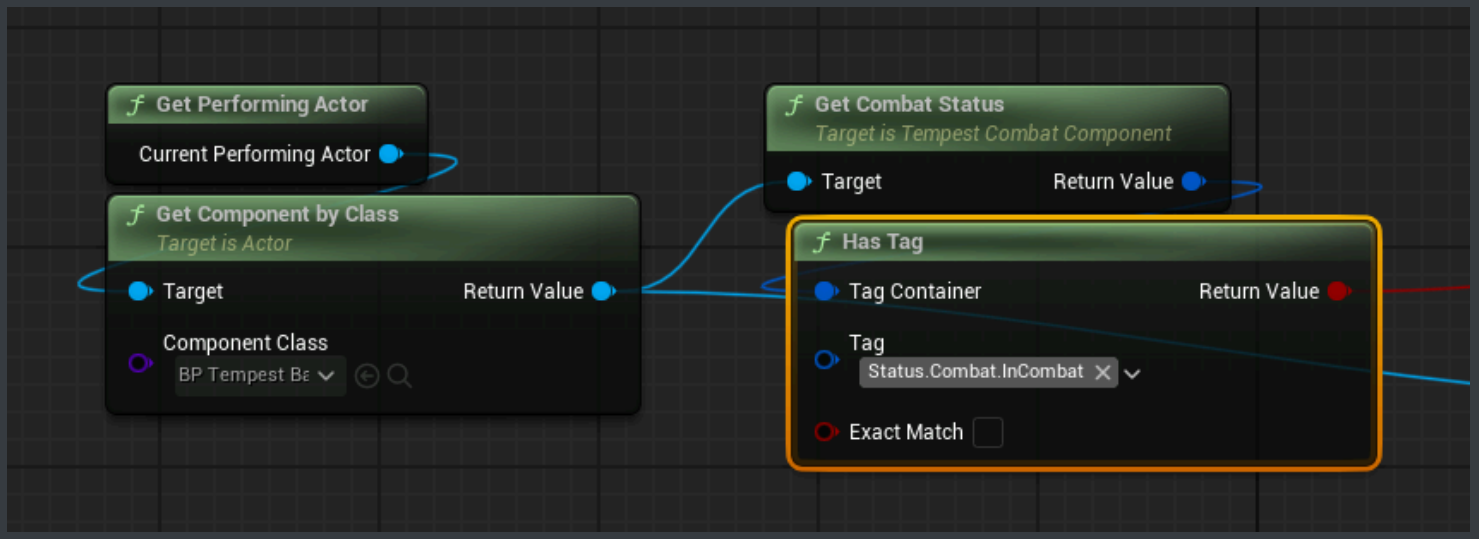
之后需要判断

1.获取攻击动画,然后判断动画是否为空

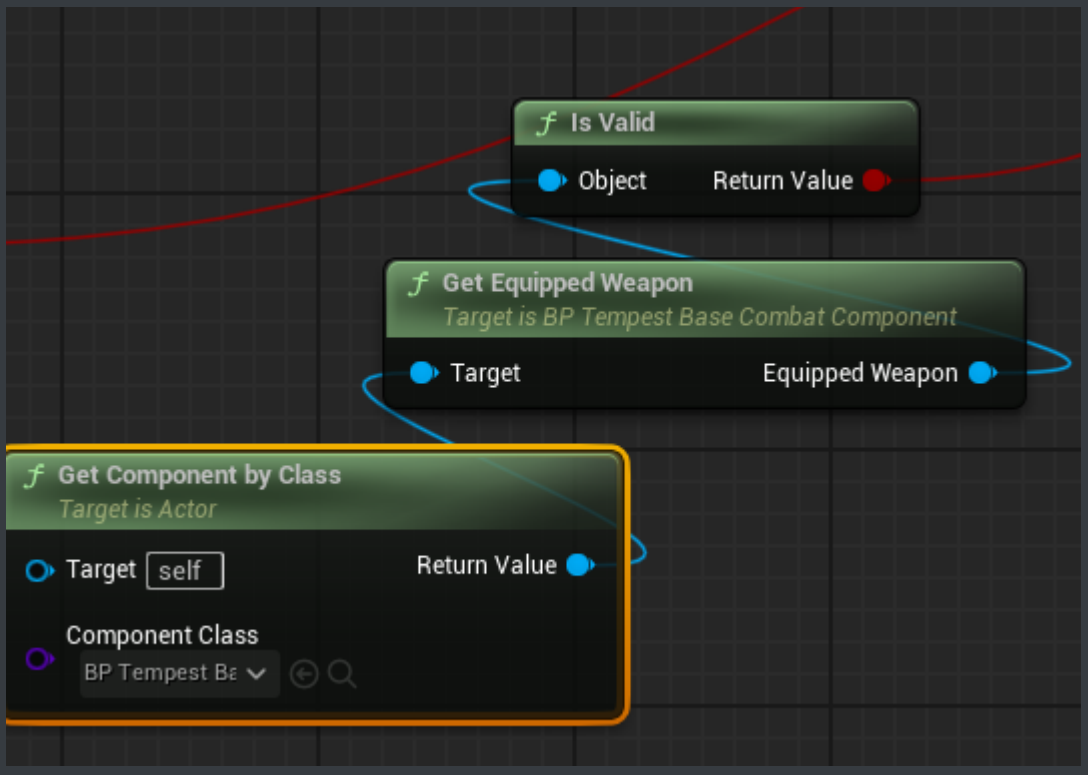
2/然后判断是否有足够的消耗值



3.判断是否处于战斗状态



4.是否持有武器



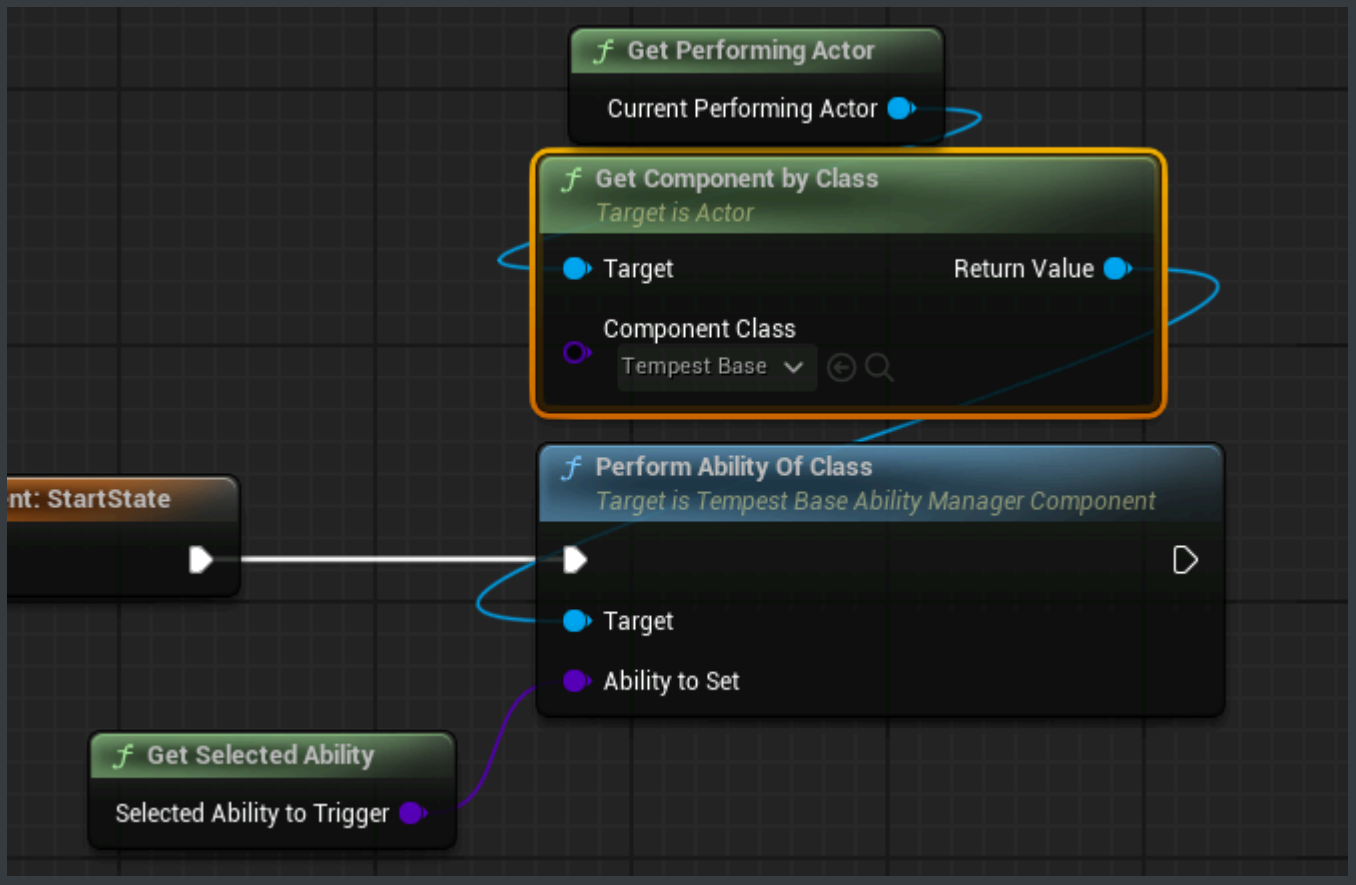
然后这四个条件都通过则可以成功通过判断条件来执行转换状态的逻辑:

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

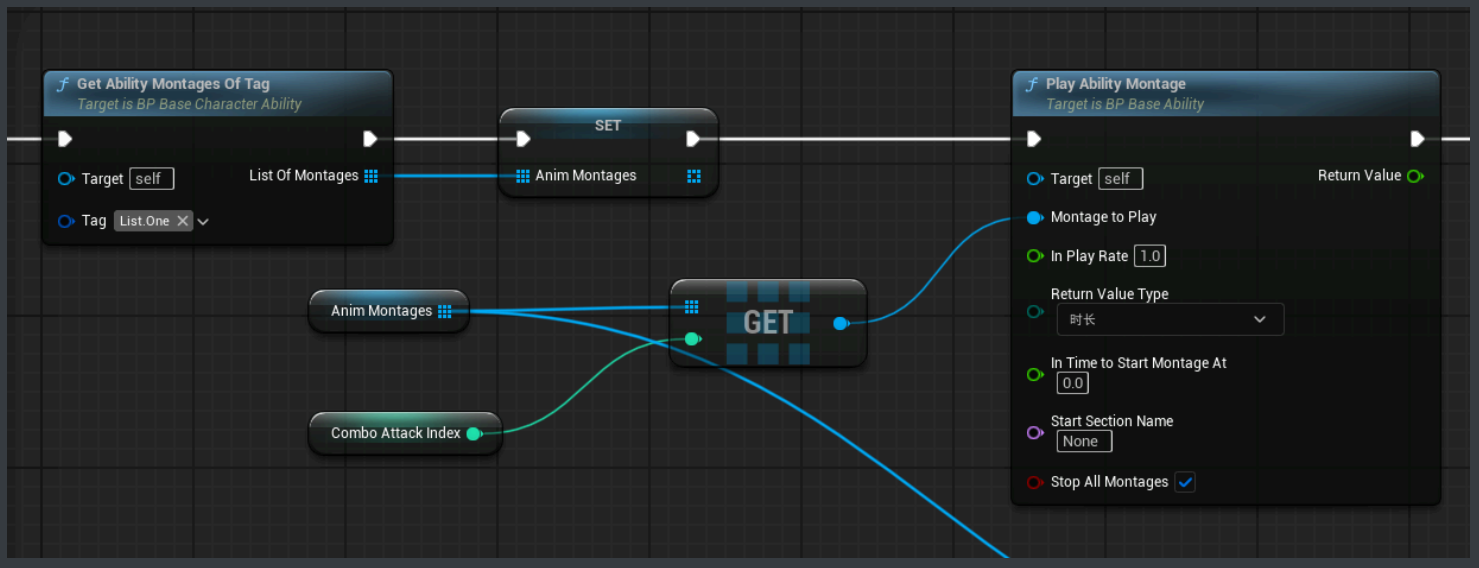
执行该状态激活前状态 -- 开始状态 --状态激活后状态

攻击只在自己的类写了开始状态:

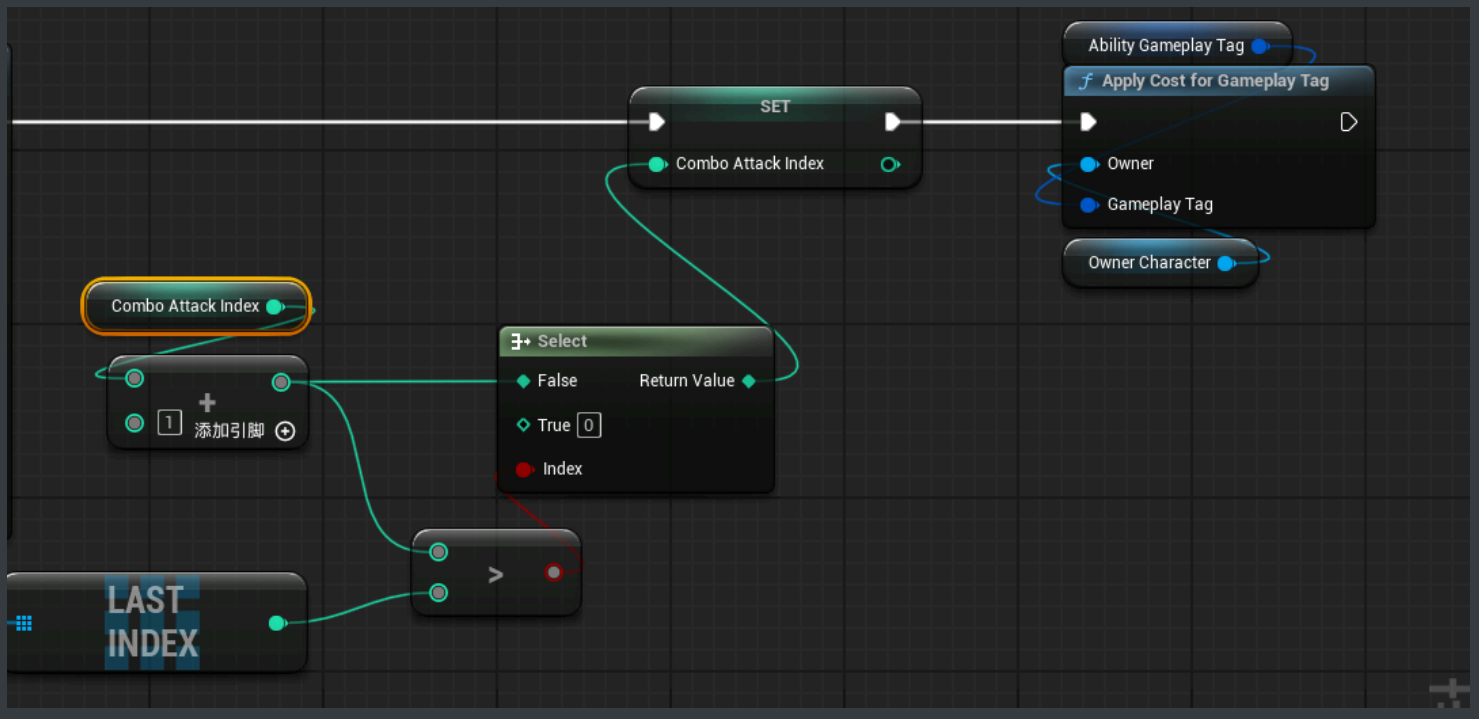
调用自己在数据资产里面的攻击能力BP_PlayerLightAttackAbility,能力组件和状态组件类似



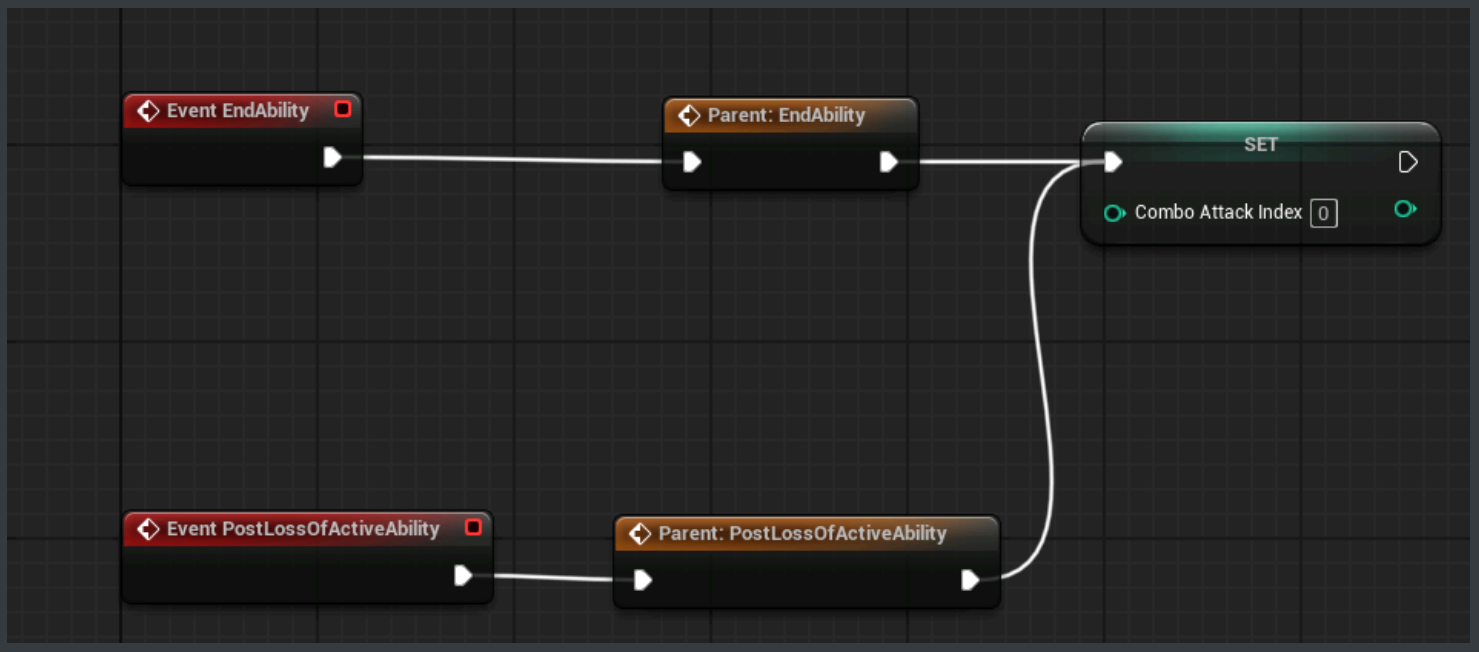
这个能力先获取要攻击的蒙太奇动画并播放



之后增加攻击索引以及计算攻击消耗



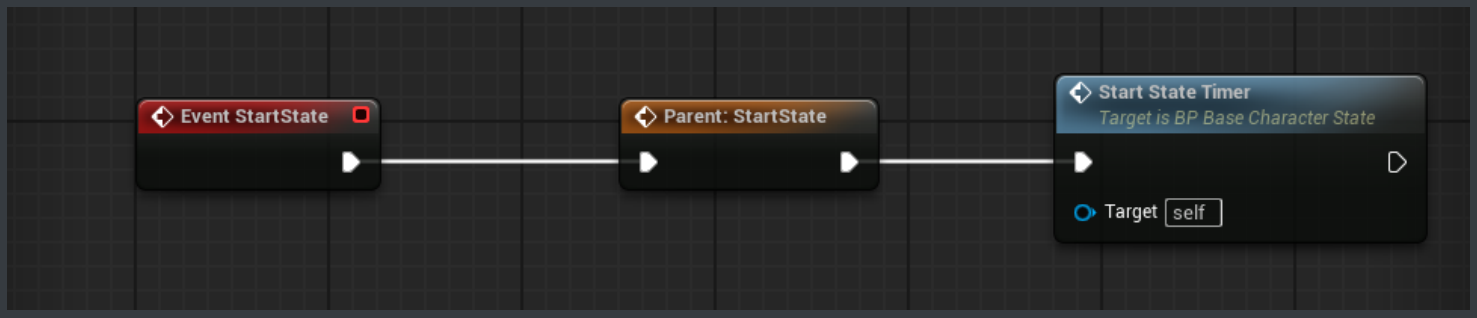
在攻击能力结束时重置索引

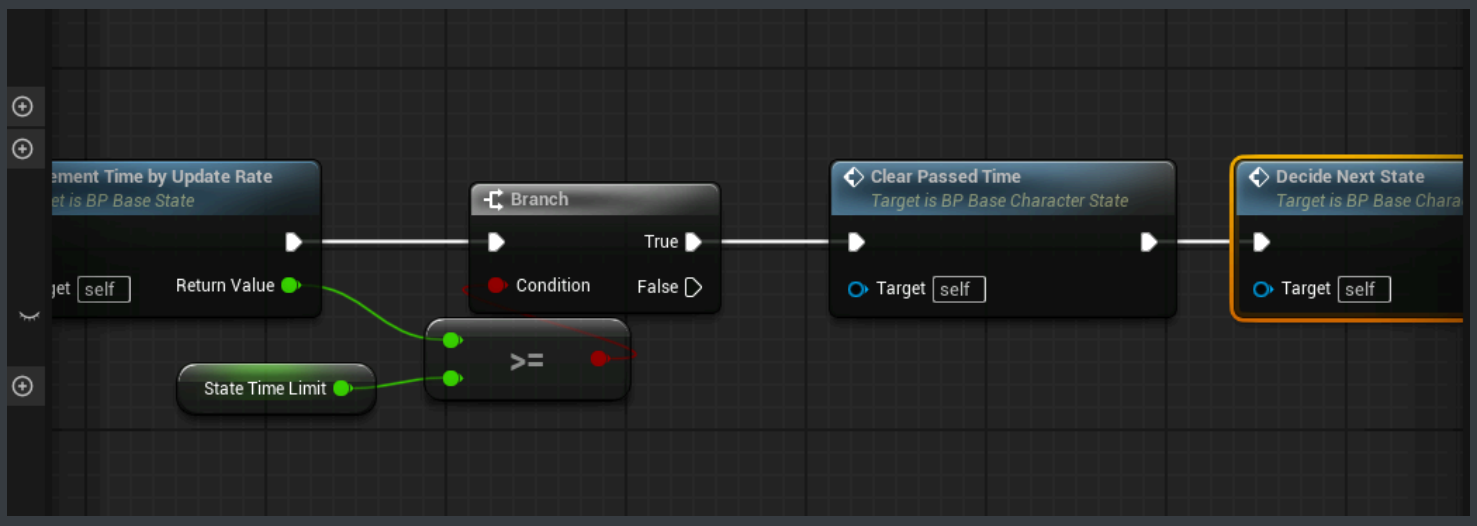


接下来只有最后一个问题,如何结束这个状态,在父类中

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

1. **属性管理**：创建、存储和检索游戏中的各种属性对象
2. **属性修饰器管理**：管理影响属性的修饰器
3. **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

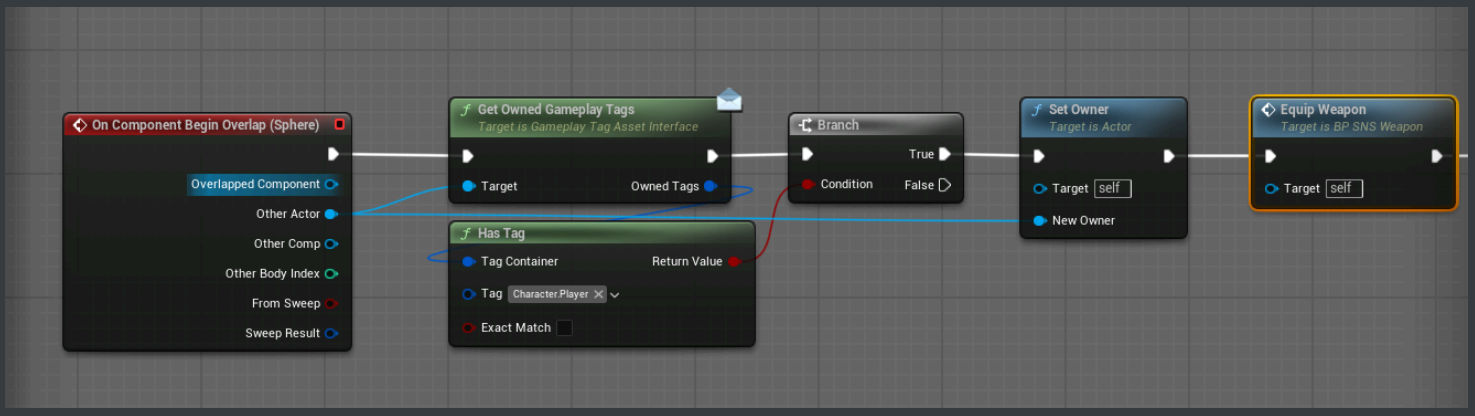
```
USTRUCT(BlueprintType)
struct FAttributeModifierProperties
{

    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

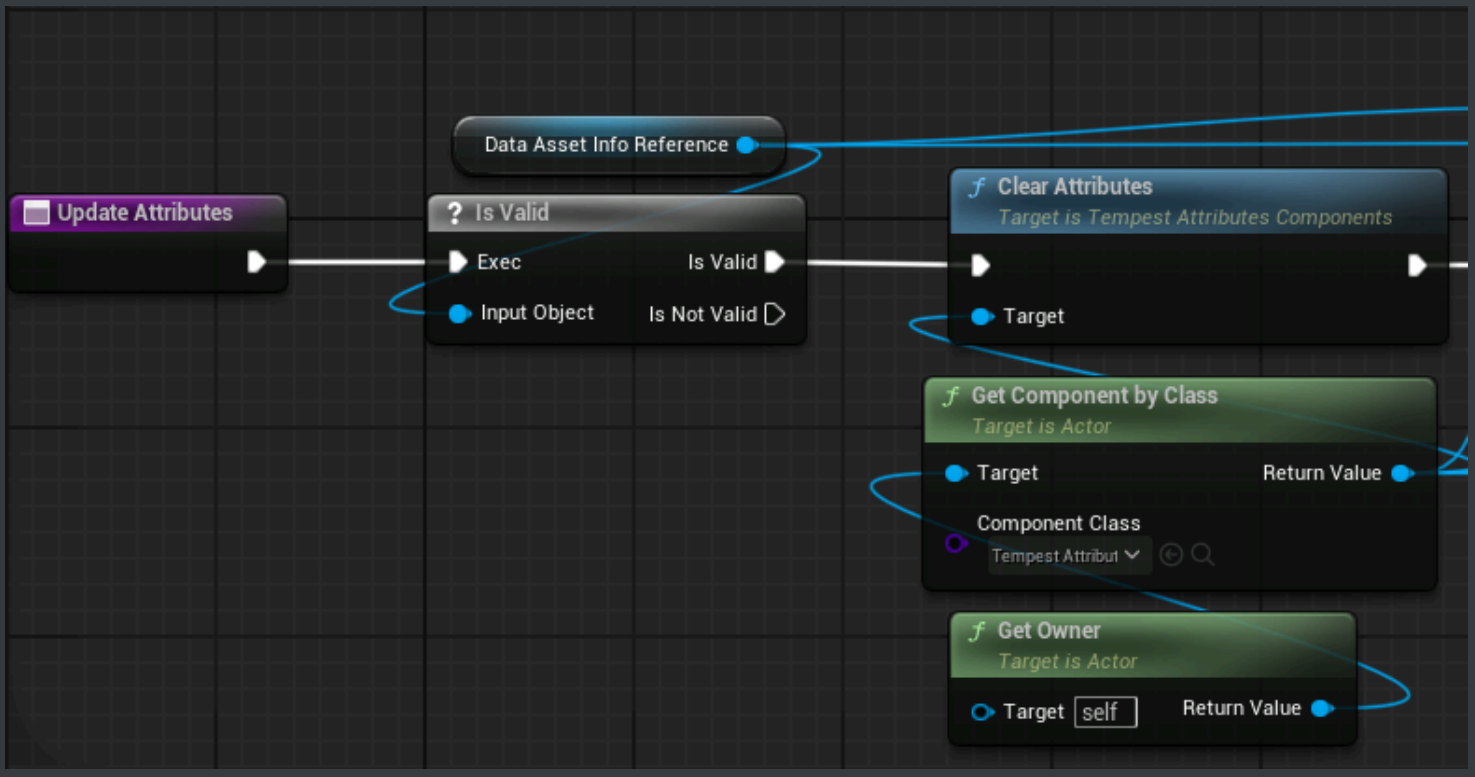
基础属性

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

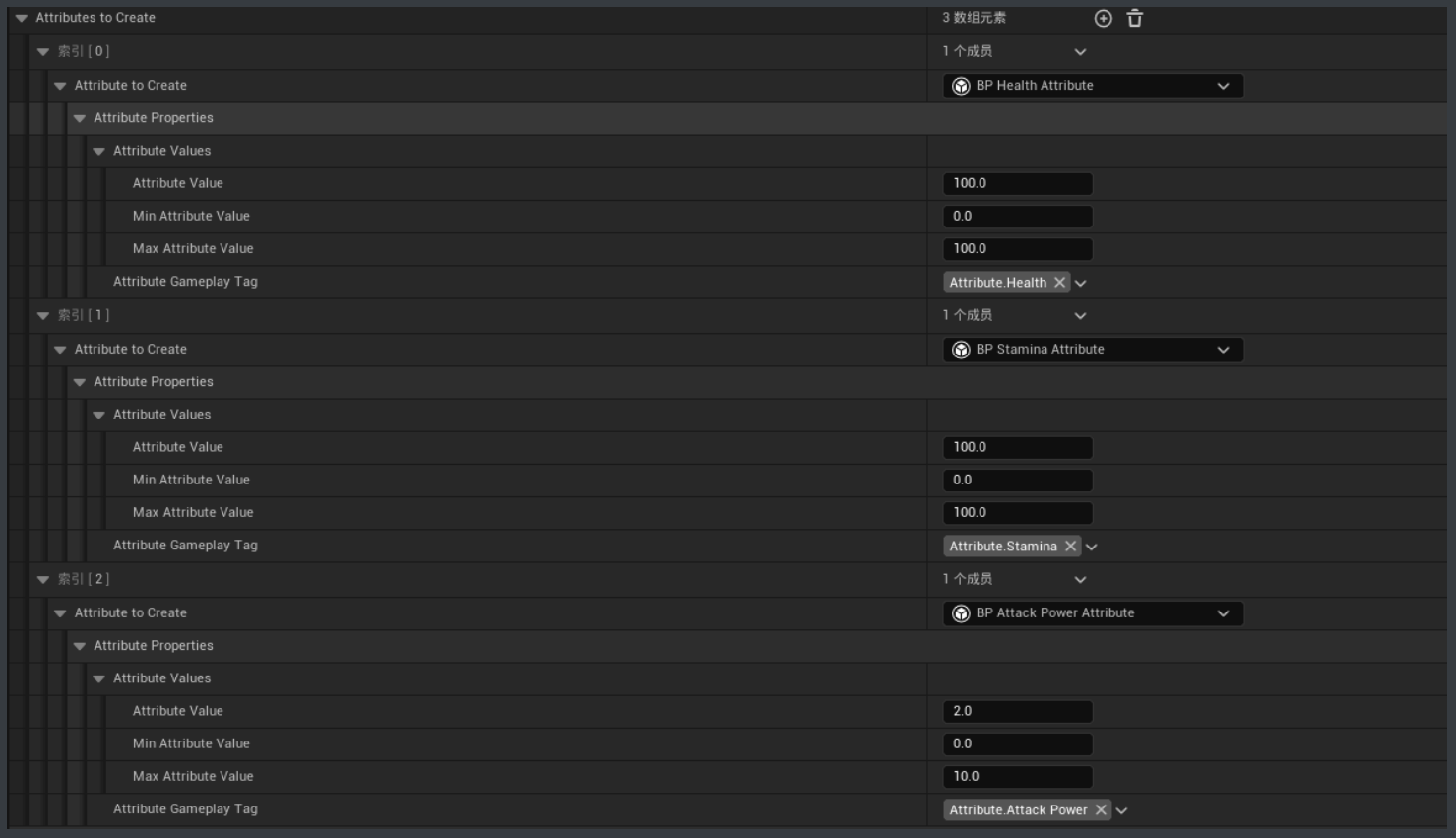


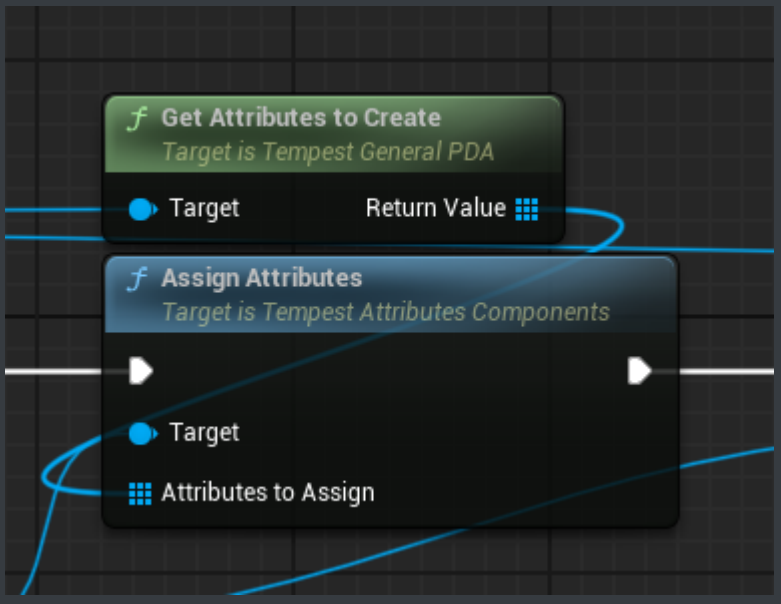
更新属性首先判断，配置的数据存不存在，这个配置是这个武器对应的数据资产，



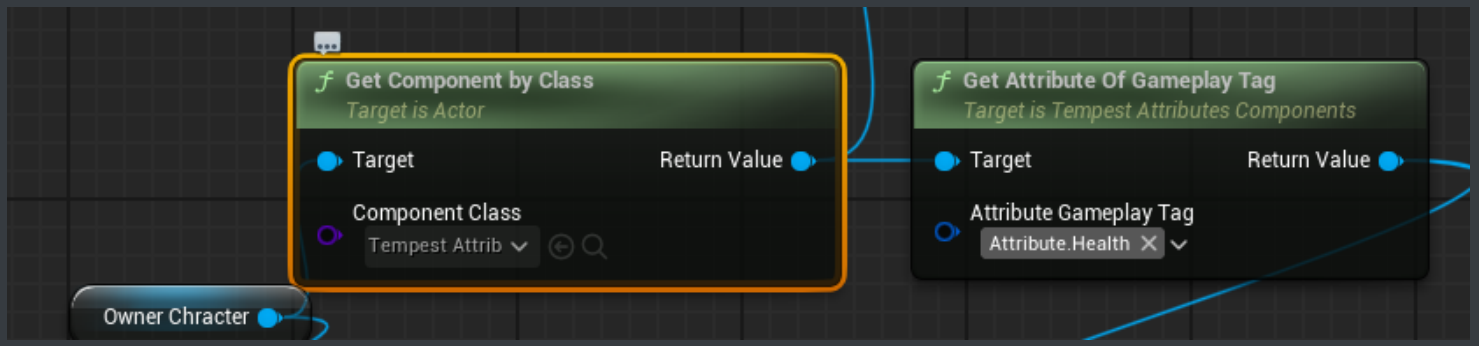
通过GetAttributesToCreate获取一个UTempestBaseAttributeObject*的数组，然后将这组数组添加到CreatedAttributes变量中存储起来

遍历的是这个AttributeToCreat里面的数据，自己配置的





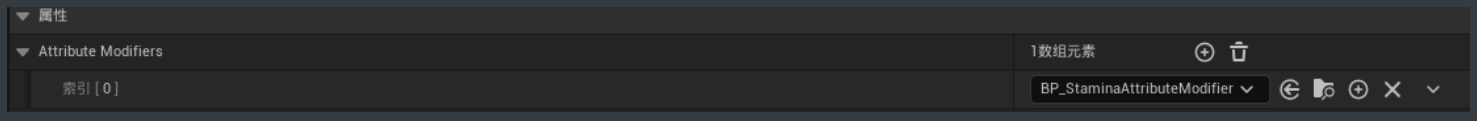
在需要这个属性的地方可以通过GetAttributeOfGameplayTag这个方法来获取你需要的属性

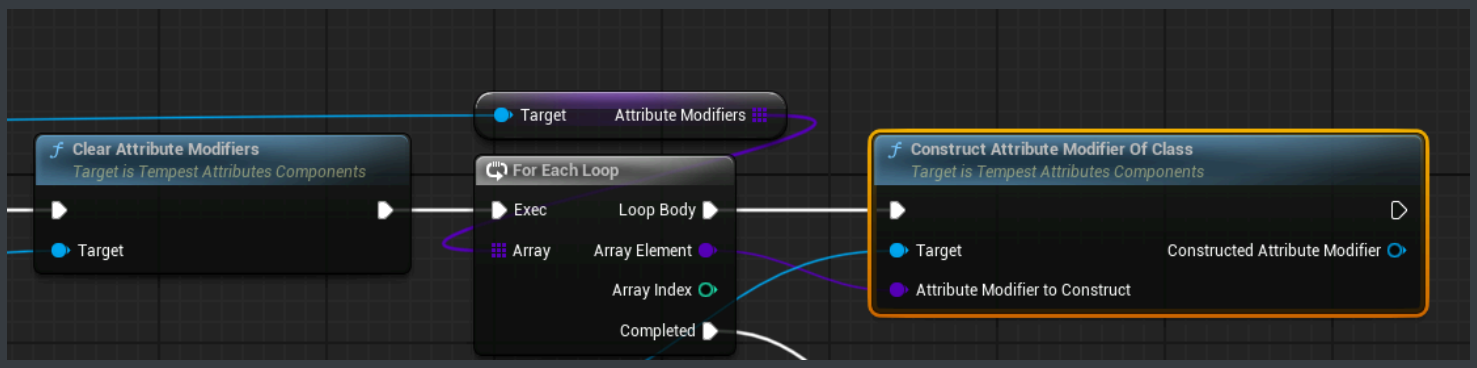


目前版本角色属性完完全全是根据武器的配置来的，没有自身的基础属性，TODO需要修改

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AttrubuteModifiers中，通过类来构造属性



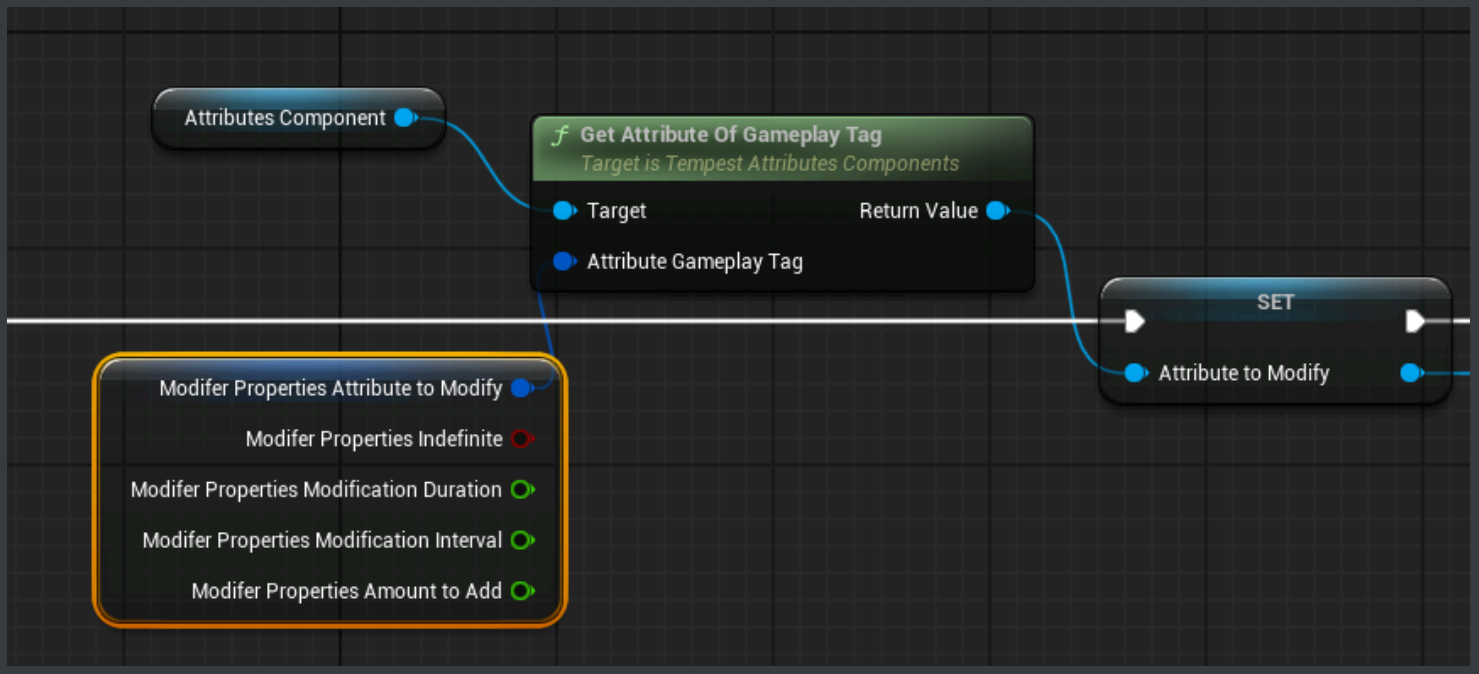


首先ConstructAttributeModifierOfClass方法会调用Modifier的ConstructAttributeModifier方法

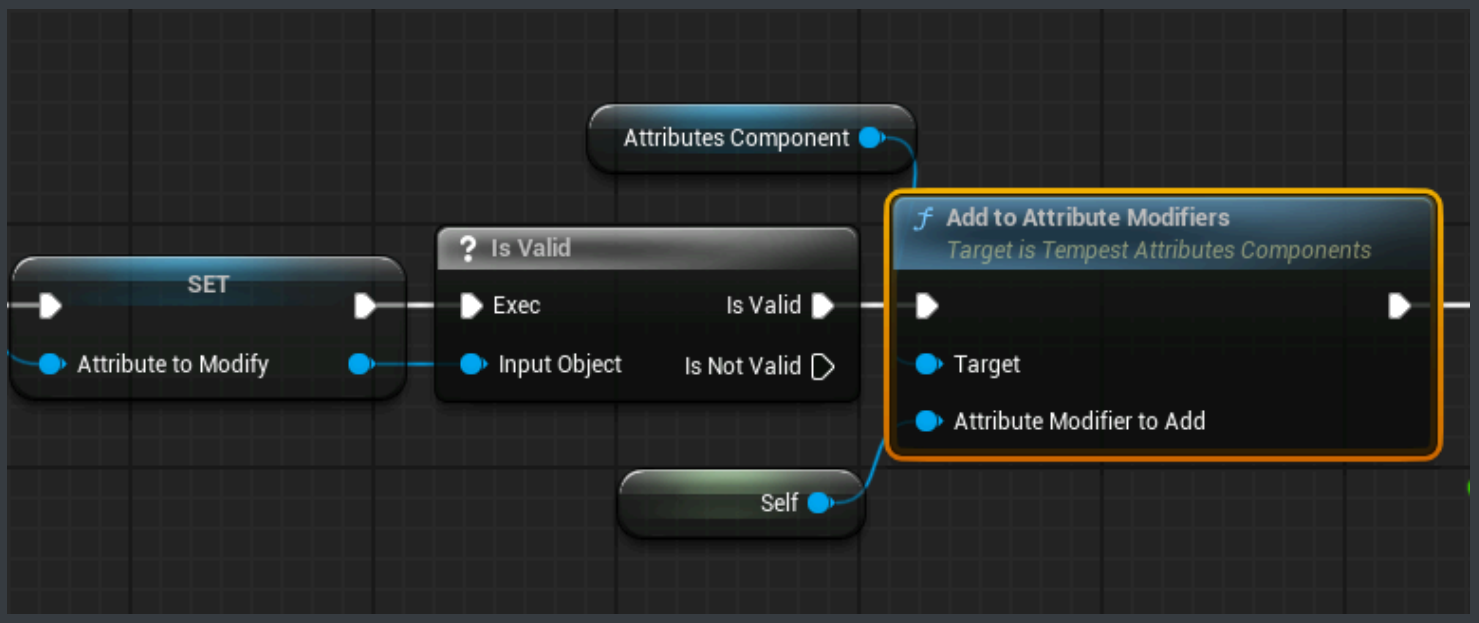
```
if (AttributeModifierToConstruct)
{
    UTempestBaseAttributeModifier* LocalNewAttributeModifier;
    LocalNewAttributeModifier = NewObject<UTempestBaseAttributeModifie
    LocalNewAttributeModifier->ConstructAttributeModifier();
    ConstructedAttributeModifier = LocalNewAttributeModifier;
}
```

然后再BP_BaseAttributeModifer蓝图类中

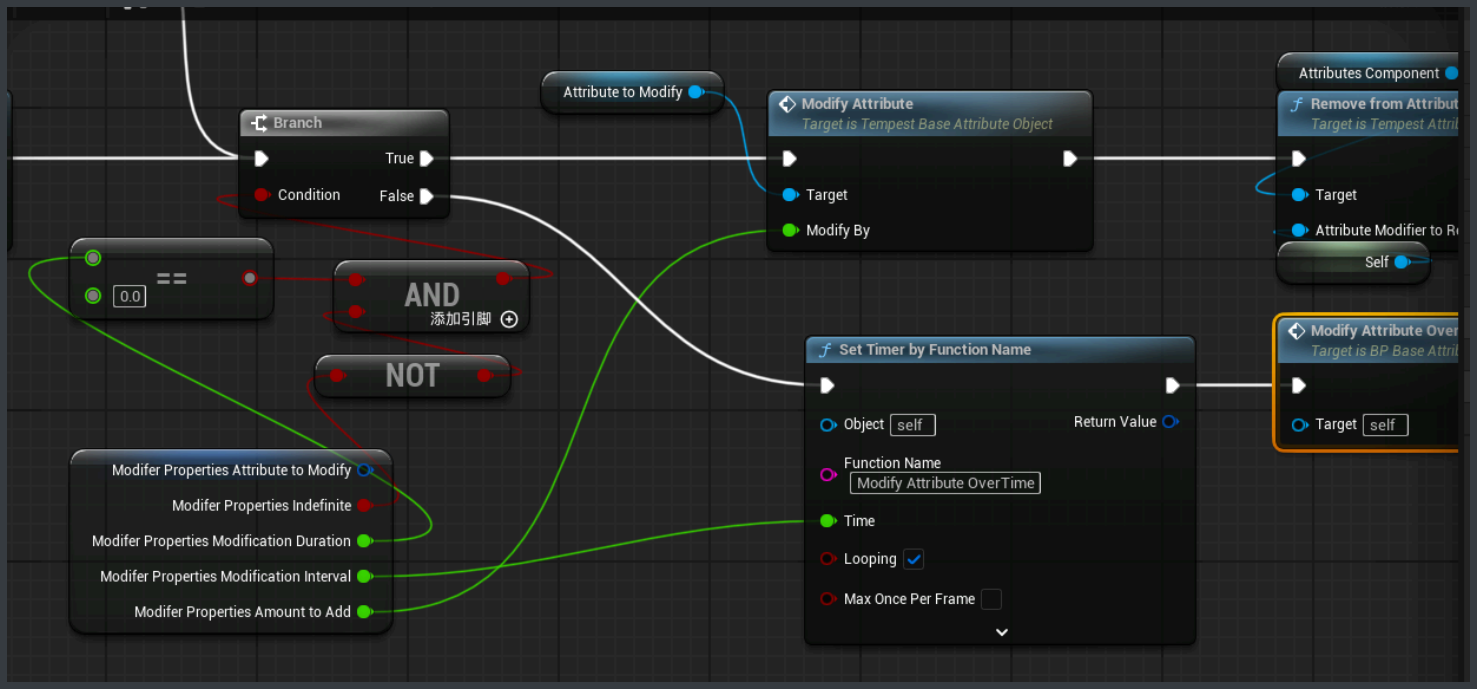
首先根据tag获取所有的Attribute



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 → StartState → 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

示例

详情请见一个攻击的流程示意(将这几个组件串起来):

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:现在人物的属性全根据武器来,拆分成两块角色+武器,如果有的天赋是增加属性的话,在点击完天赋就增加角色属性,保存到CharacterInfo中
- 学习天赋: