前置条件：有使用过事件分发器、蓝图接口、GAS的AttributeSet的创建与维护

为什么用事件分发器而不是接口。或者说，什么时候用事件分发器什么时候用接口。

两者的区别、使用时机、为什么使用。

在游戏开发中，构建模块化、灵活的系统、可扩展性和可维护性非常关键。

接口和事件分发器都能实现游戏各模块之间的通讯，来避免产生复杂的依赖关系。

## 接口

接口可以定义一组函数，这些函数描述，类应该如何行为，但又不需要实现这些函数。使用这个接口的类会提供自己对于该函数的独特实现。

接口并不是用于共享函数的具体实现（共享函数的实现有其他方式），这是一种在不同类之间共享结构的方式

举个例子

假设正在开发一款游戏，游戏中可以与不同对象互动，比如，和NPC互动、和物品互动。我们就需要一个互动功能，可以在按下特地按键时与特定对象进行互动。

但是NPC和物品的交互方式是不一样的，他们有各自独特的互动功能实现方式。在NPC和物品交互之间的相同点，就是他们都必须具备这个功能。这个功能就是接口中的函数。

为了确保他们都包含这个功能，这个函数接收特定输入并产生相应输出，你需要创建一个可交互接口，并在接口中定义该函数。这样，任何实现该接口的类，比如NPC和物品，都将要提供自己独特的交互函数的实现。

这样做的目的是：能够让不同的类实现相同的接口，并确保这些类行为的一致性。

我们可以做一个实验

创建BPI\_Interface，创建两个函数：

Interact，没有任何参数和返回值

CanInteract，返回值为Bool类型，用来判断是否可以互动，比如npc正在繁忙无法互动

现在我们只关注Interact函数。

接下来创建两个类，NPC类和PickUp类。他们目前还只是空类，他们的内部只包含一个网格体的空Actor。

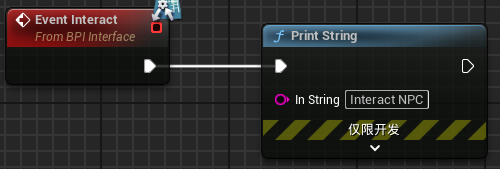
进入NPC类，在类设置--已实现的接口中添加BPI\_Interface

这时，就可以在我的蓝图中看到Interact函数。同时提示我需要实现这个函数。

此时双击函数名，就会自动创建这个事件。（如果函数有输出，引擎就会创建一个普通函数，而不是事件。）



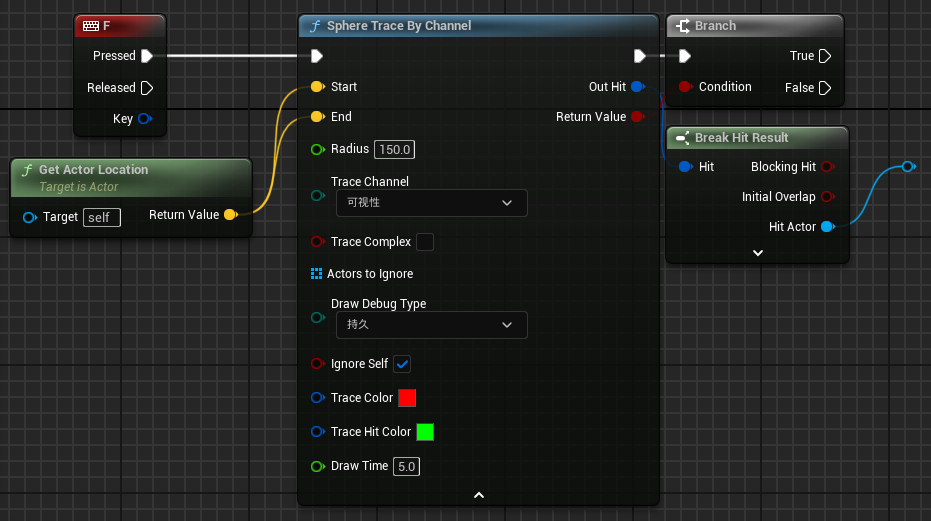
接下来就是定义域NPC交互的具体含义。因为我们正在处理NPC类，我们可以让NPC朝向玩家（或者打开任务菜单之类的）。在这个示例项目中，我们只需要Print就可以



对PickUp这个Actor同理

此时我们就在两个类里面实现了这个接口。我们还没有在任何地方用到接口的函数。这也是接口发挥作用的地方：接口能够实现不同类之间的通信。

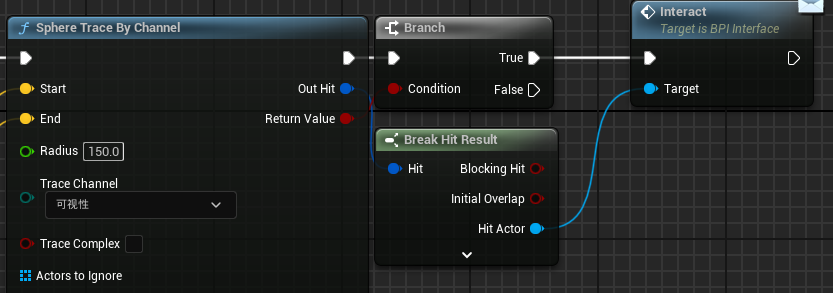
现在进入到Player类中，当摁下某个按键时，进行一次球体追踪检测



此时我们只是检测到了一个Actor。我想试试能不能和碰到的这个Actor进行互动，

进行互动的方式非常简单粗暴，直接调用蓝图的函数就可以

当然，如果目标没有实现所需接口，则什么都不做



Tip：使用接口可以避免CastTo。CastTo也并不是总是不好的。

这里解释一下CastTo

我们在内容管理器中右键一个资产，选择“尺寸贴图Size Map”



我们可以在右上角选择内存大小，他会显示这个对象在内存中加载时占用了多少RAM空间。



可以看到这里大部分是贴图、纹理、骨骼网格体等数据。

现在如果我进行任意的类型转换，仅在Player中加一个CastTo节点，就可以在尺寸贴图里看到内存空间加了很多。如果我CastToNPC，我会在加载Player的时候同时加载NPC。这是因为当进行类型转换的时候，是在告诉引擎Player的存在依赖于NPC，所以两个会同时加载。

所以我们并不需要过分的避免CastTo节点。对于那些明确依赖于其他的类，就可以对该类进行CastTo。比如角色本身携带Axe作为武器，当在角色蓝图内时，就可以很放心的CastToAxe，因为他本身也会随着Character加载到内存中。

同时，如果你对AIController进行类型转换，转换成它所控制的敌人的AI，也是没问题的，因为他们总是一起加载的。

总的来说，就是尽可能的避免互不依赖的类进行类型转换。

总结：当你需要在多个不同的类中强制实现一组相同的函数时，就应该使用接口。这样其他类就能统一处理这些类，而无需直接引用他们。这样就能给游戏增加结构性，还能实现职责分离，减少依赖，降低内存占用。

## 事件分发器

事件分发器的作用是在对象之间广播事件，用来通知某件事情已经发生。也就是说，当你需要等待某个时间发生时，这个方法特别有用。

比如等待一个对象，然后另一个对象执行操作。

比如，想在玩家受伤时更新血条界面，就可以在TakenDamage的时候CallOnEventHappen，然后在别的地方，比如在UI这里绑定事件，在事件发生的时候SetPercent。

这是事件分发器最常用的场景之一。

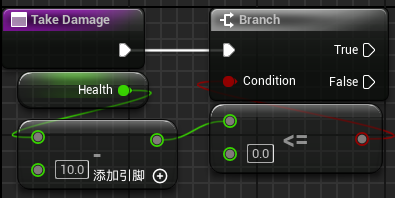
还有一个用处，是使用事件分发器构建通用的角色组件。这些组件可以应用于任何角色。

假设想创建一个可以添加给任何角色的生命值组件，使其管理生命值、受伤和死亡，在这种情况下，事件分发器会很有用。

创建ActorComponent类蓝图，命名为HealthComponent

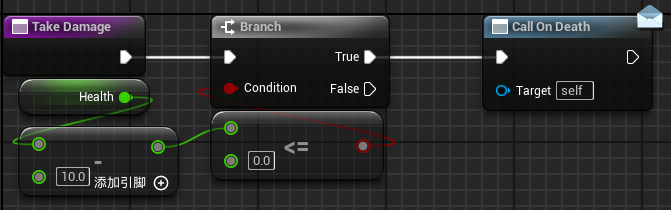
创建变量：Health

创建函数：TakeDamage



如果此时生命值小于0，拥有HealthComponent的这个对象应该死亡。但我们此时并不想实现死亡的具体逻辑，因为这个组件可能被NPC调用也可能被Player调用，而且他们有各自独特的死亡行为。

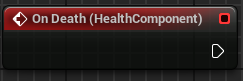
所以Actor组件并不应该预设死亡时的具体情况，相反，他只需要分发一个事件即可



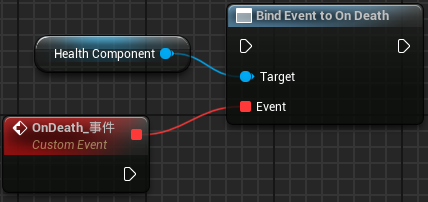
现在进入Player蓝图，添加这个组件

此时就可以在这个组件的细节中找到OnDeath事件

此时我们就可以创建相应的事件。



当然，我们也可以使用Assign来绑定事件



他们是等价的。区别就是提供了解绑的选项和手动绑定。

总的来说，事件分发器能让游戏的一部分向其他部分发送信号，从而触发相应的反应。这在当游戏的某部分需要等待特定事件发生后再采取行动时相当好用。比如玩家拾取某个物品后才开启一扇门，这样就不用反复检查玩家是否拾取物品，一遍又一遍的问这个问题。你只需等待事件发生，然后再采取相应行动即可。

## 混淆

基于以上内容，两者可以实现相同功能。

我们在BPI里创建Death函数，在HealthComponent的函数中，当生命值小于0的时候，GetOwner调用Death函数，也能实现功能。

那么为什么前面不用这个方法？

首先说接口方法。我们假设一种情况，我们在HealthComponent中有一个接口Death函数被调用，所以如果我用的是BPI而不是EventDispatchers，HealthComponent仍然会做出一个假定：它假设GetOwner的返回值实现了这个接口，且拥有Death函数，同时也不允许所有者执行其他任何操作，你无法对其进行扩展。

但是如果使用事件分发器，任何角色都可以使用这个组件。所以就算是PickUp蓝图，虽然没有死亡的方法，但是也可以处于某种原因实现生命值组件

## 区别

EventDispatcher能够提高代码的灵活性，更易于管理。因为它实现了游戏各模块之间的相互通信，而无需预设个模块的具体功能。

BPI则在不同类之间强制实施特定结构方面非常有用。他确保了一旦某个类实现了接口，就相当于签订了一份合同，这意味着该类承诺遵守特定的结构，或实现规定的一系列功能。这样一来，你就能用同意的方式处理不同的类，而不必担心每个类的具体实现。