前置：

创建一个简单的行为树

Enemy有四种行为：

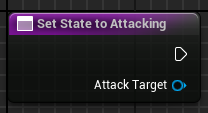
未激活的、等待的、攻击、死亡

这些状态都由AIController中的一个变量State来控制

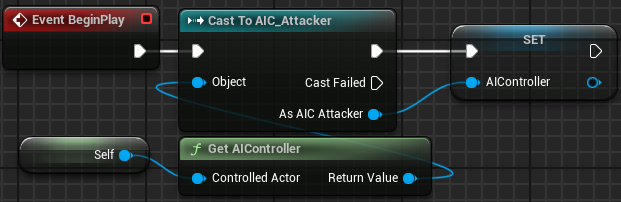
这就是我们要在告诉Enemy去Attack或Retreat或Wait时做的，要通过AIController的函数，来修改这个变量。

创建AIController蓝图，命名为AIC\_Attacker

创建变量State，实现函数SetStateToAttacking和SetStateToPassive



在Enemy中



假设一下，所有的敌人都需要攻击玩家，但是我们需要限制能够同时攻击的敌人数量。

一旦Enemy死亡，另一个Enemy会取代他的位置。如果玩家死亡，所有的攻击者应该回到他们的起始位置

我们该怎么做才能使Enemy之间不需要互相配合来跟踪：谁在攻击？谁在等待？和Player通信来获取当前状态？从而避免复杂的依赖关系。

## 中介者模式

两个案例

战斗管理器：前面介绍的战斗场景逻辑

事件管理器：解决观察者模式的一些缺点

中介者模式aims to answer the question how to coordinate communication between different actors in my game,without having direct references or dependencies on each other.

对于战斗管理器的案例，这些敌人如何才能知道有多少攻击者正在攻击玩家、他们现在是否能够进行攻击。

一个容易犯的错就是在Enemy和Player之间，以及在Enemy和Enemy之间建立直接通讯。这样会创建硬引用，对内存使用不利，还会再游戏中的不同角色之间产生依赖关系。

中介者模式要做的就是让对象之间相互独立，所有的对象都只和的中介者进行通信。中介者再将消息重定位给对应的其他对象。这样，你的所有对象都依赖于中介者，而不再依赖其他对象。

## 战斗管理器案例

问题是，所有的Enemy都想攻击玩家，但是最多只有3个能同时攻击到。一旦一个死亡后另一个会上场。Player死亡后所有的攻击者能够回到他们的出发点。与其把这些逻辑放到Player和Enemy类中，不如创建一个战斗管理器作为中介者，来处理所有的战斗逻辑。

创建一个Actor类对象，命名为CombatManager。我们可以把这个Actor直接放置在场景中。

现在这个战斗管理器可以跟踪一些对象，来实现我们想要的行为。

在CombatManager蓝图中，创建Actor对象引用类变量AttackTarget，创建攻击者的列表Attackers，类型为Actor类数组。创建WaitingAttackers列表，这里装的是那些想要进行攻击，但是已经达到最大攻击数目限度的不能攻击的Enemy。

这些变量都是Actor对象引用，我们并不需要指定攻击目标和攻击者的类。

首先，这些Actor该如何和中介者进行通信呢？当攻击者进入战斗时，他们会询问中介者能否进行攻击，然后中介者会检查目前有多少个攻击者正在攻击目标，然后告诉Enemy能否进行攻击。所以攻击者Enemy应该暴露一个Attack函数和一个Wait函数给中介者CombatManager，然后将攻击的实现留给Enemy自己。这就是接口的作用。

创建BPI\_Attacker。在这里实现中介者关于攻击者的所有的函数。

在BPI中创建Attack函数和Wait函数，同时创建一个Retreat函数，这个函数将在攻击目标（也就是Player）死亡后被调用。

Attack和Wait函数都有一个Actor类型的参数作为形式参数，命名为AttackTarget。

然后给攻击目标创建接口

创建BPI\_AttackTarget。

中介者只需要从攻击目标身上知道允许同时和之多多少个攻击者战斗。在BPI中创建函数GetMaxAttackersCount。这个函数会返回一个int型变量，命名为MaxAttackersCount。

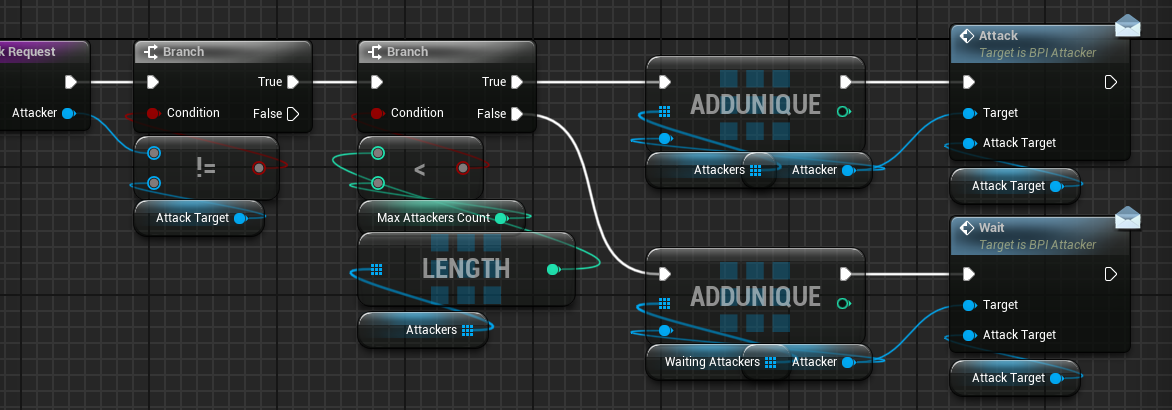
接下来关注战斗管理器如何使用他们

在CombatManager蓝图中，在EventBeginPlay时，调用GetMaxAttackersCount函数，获取允许的最大攻击数量，然后将它提升为变量，命名为MaxAttackersCount。



然后我们需要一个方式让攻击者可以询问中介者是否能够进行攻击。

创建函数命名为HandleAttackRequest



这里我们并没有CastTo，因为这些函数时接口函数，可以在任何对象上调用。如果对象没有实现接口，就不会做任何事。

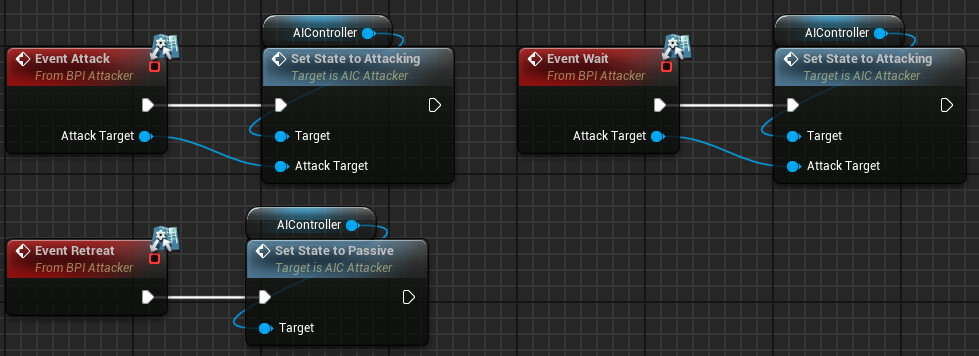
接下来在Enemy和Player中实现接口并调用函数

在Player类中，加上BPI\_AttackTarger接口。双击接口函数，添加一个简单的静态数字就可以了

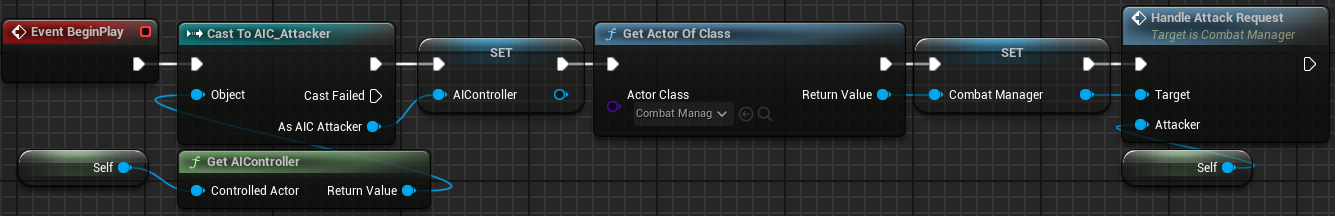


在Enemy类中，类似。

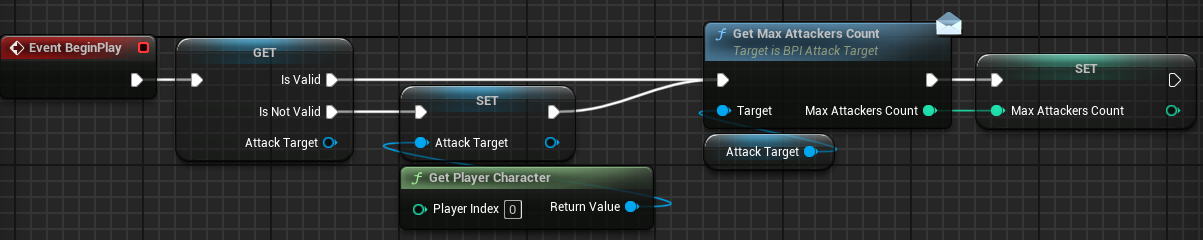
创建接口的实现事件



最后处理AttackRequest



回到CombatManager，我们要告诉它谁是攻击目标。将AttackTarget变量设置为可编辑实例，并在EventBeginPlay时检查它是否合法。



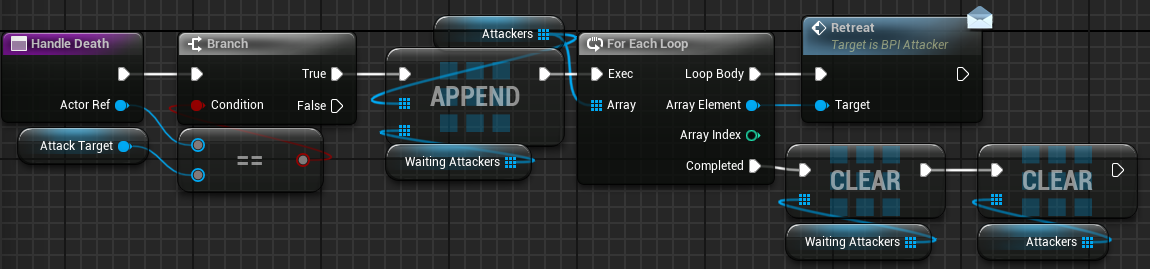
现在就可以开启游戏观察到有3个Enemy移动到自己的位置

（记得把CombatManager在场景中创建一个实例）

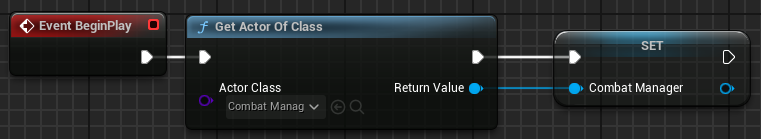
改GetMAxAttackersCount函数的返回值也可以正常运行。

现在我们来处理当Player死亡的时候的情况。

回到CombatManager，创建函数HandleDeath



然后我们需要在Player死亡时调用这个函数

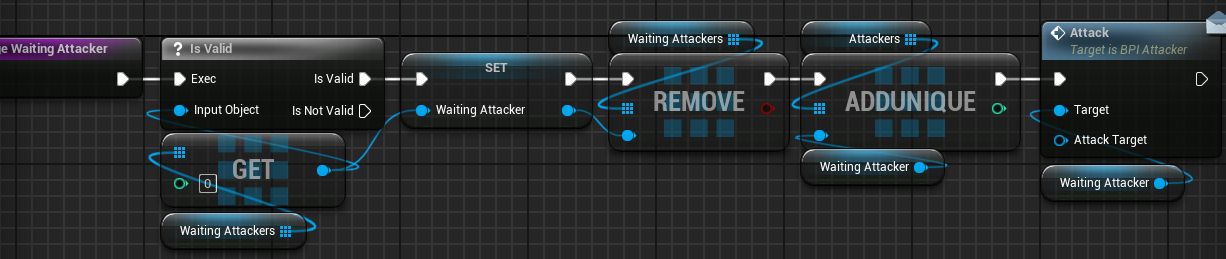




之后，当Player死亡时，Enemy就会回到原点。

最后处理，当一个Enemy死亡时，让另一个等待攻击的Enemy开始攻击。

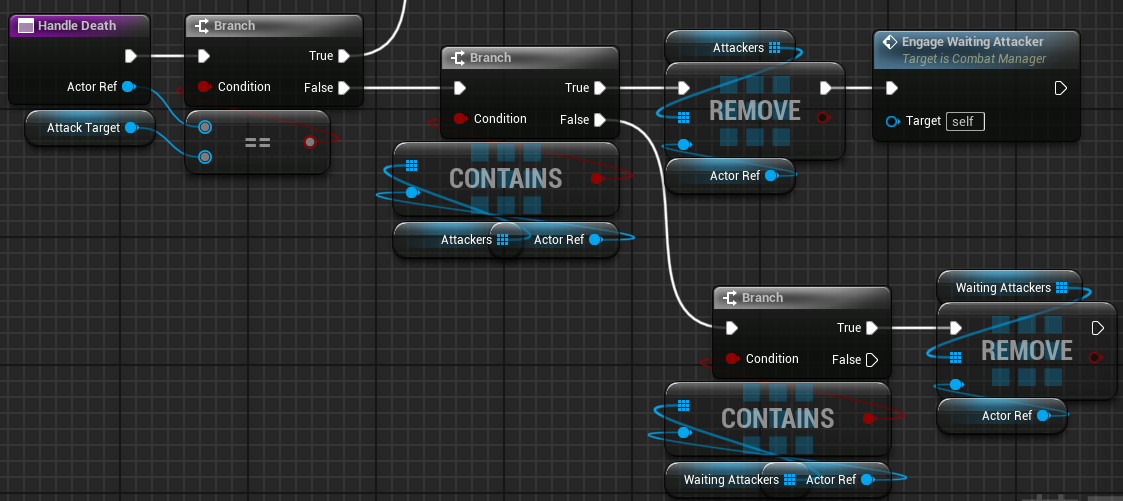
在ConbatManager中创建函数EngageWaitingAttacker



这里的WaitingAttacker是局部变量

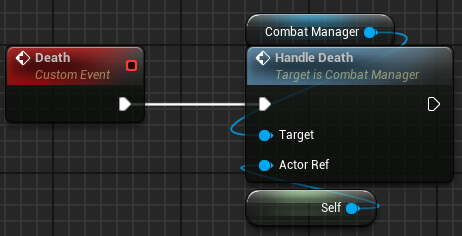
我们在Enemy死亡时调用这个函数

进入HandleDeath函数



这里的ActorRef节点就搜GetActorRef就可以

最后，在Enemy蓝图中，



这样就完成了所有功能。

这时，我甚至可以将一个Enemy，或者任何一个Actor，作为攻击的目标。

方法就是在场景中选择CombatManager，在细节中给AttackTarget赋一个场景中实例化的值。

另外要做的，就是在Enemy中实现BPI\_AttackTarget的接口并实现函数。

## 事件管理器

这里将中介者模式和观察者模式合作，创建一个事件管理器。

观察者模式的缺点是，我们依然需要Enemy对象的引用，来绑定到他们的OnDeath事件。无论是Widget还是Door。这意味着游戏的任何一个部分中，如果你想知道敌人的死亡信息，必须首先持有关卡中所有敌人的引用。这很糟糕，因为作为事件的订阅者，我并不关心是谁发布了它。我只想在事件发生的时候被通知到。

这就是需要引入中介者模式的地方，我们是用一个事件管理器作为中介来处理所有发布和订阅的调用，而不是让Widget小部件或者关卡蓝图和Enemy进行绑定。

创建事件管理器，我们首先需要创建GameState对象。因为GameState对象是一个全局可访问的对象，而且在客户端和服务器上都会被同步。这是用来实现事件管理器的最好的地方。

创建GameStateBase类蓝图，命名为MyGameState。将这个GameState赋值给GameMode。

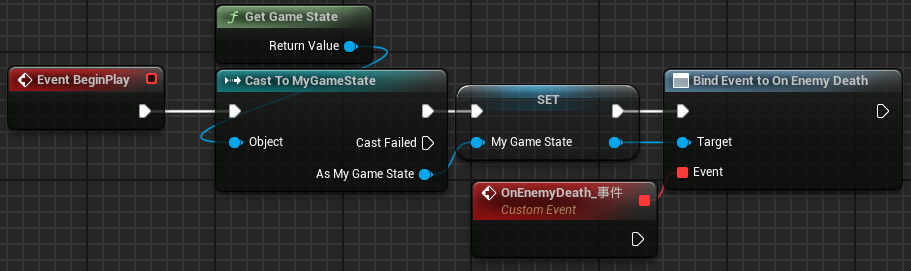
现在，这个GameState对象将会使我们的事件管理器，他的职责是，分发我们游戏中所有的事件。

项目用到观察者模式的项目

回到Enemy中，删掉OnDeath事件分发器

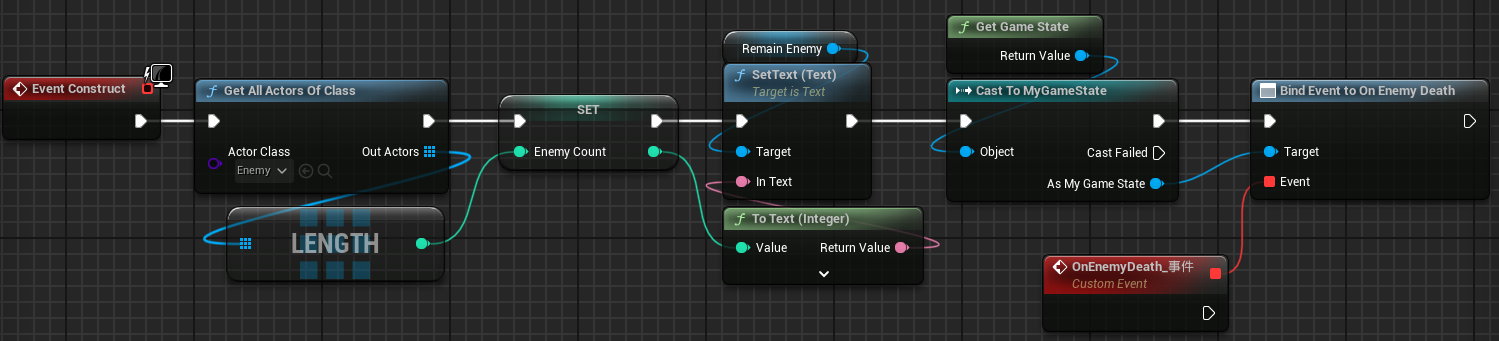
然后进入GameState，添加一个新的事件分发器，命名为OnEnemyDeath。

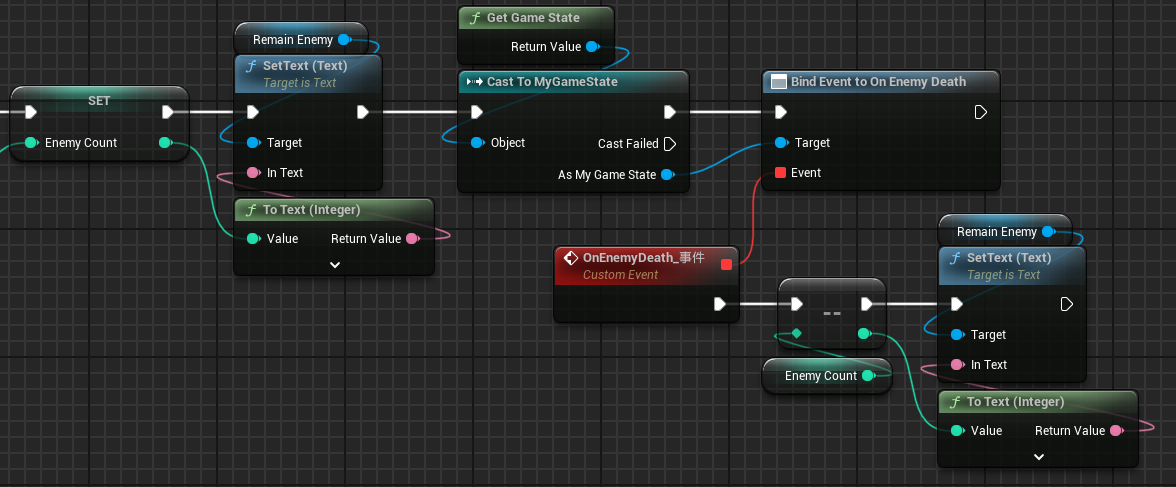
现在回到Enemy蓝图，取而代之的是调用GameState中的事件分发器



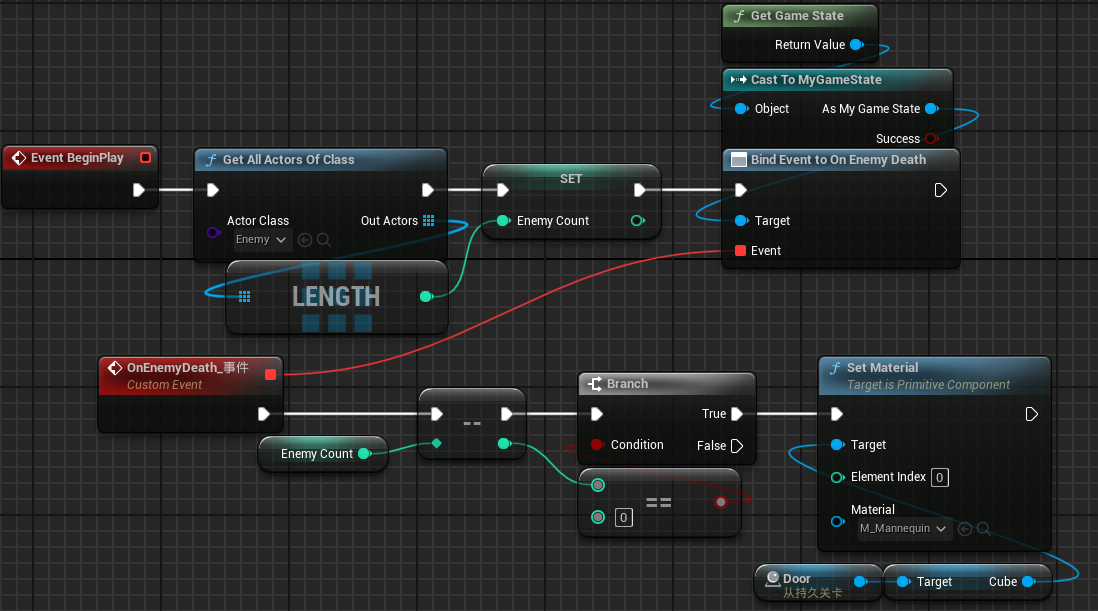
（后面会移除CastTo节点）

然后进入Widget WBP\_RemainEnemy，





然后进入关卡蓝图做一遍相同的事情。



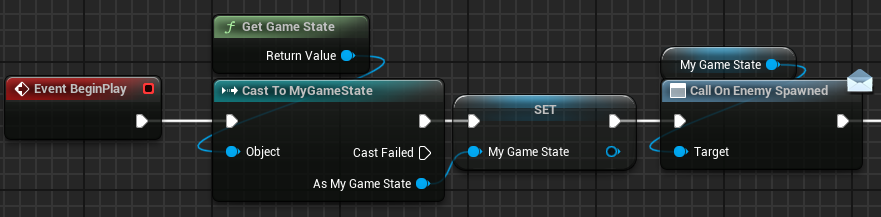
（记得在Enemy类中调用事件分发器）

完成

我们还要继续摆脱这些初始引用，不要循环遍历敌人来获得初始数量。

方法就是在GameState事件管理器中添加另外一个事件分发器，命名为OnEnemySpawned。这样我们就可以在SpawnEnemy的时候发布这个事件，然后在SpawnEnemy实例的时候进行计数。这样就可以不是以一个默认的数字开始。

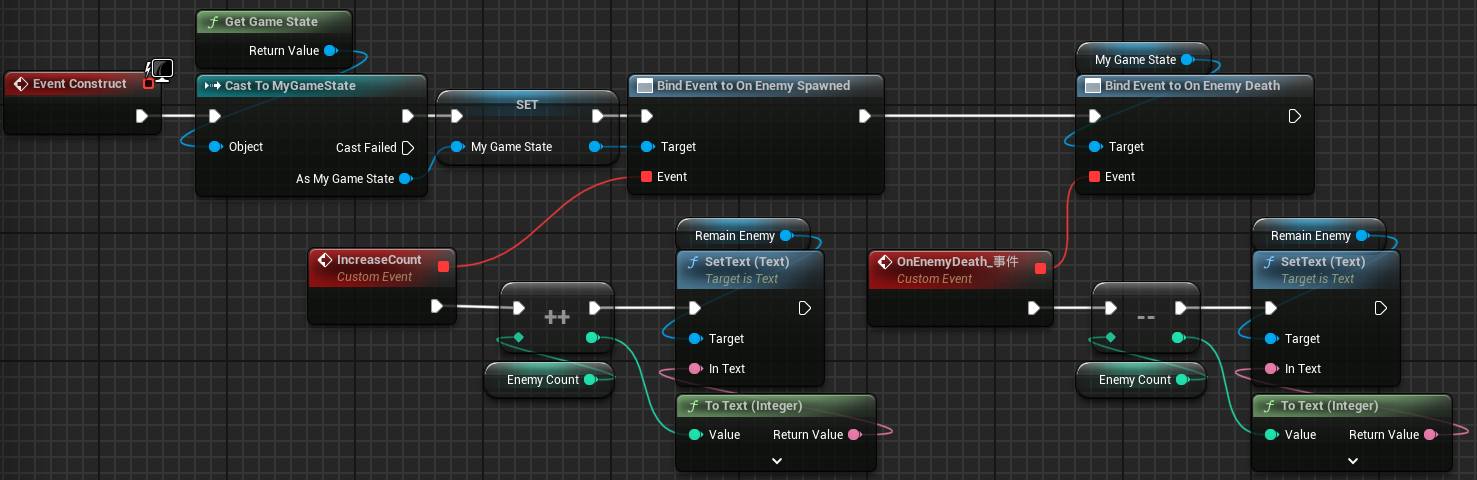
在Enemy的BeginPlay中



注意，在EventBeginPlay的时候发布事件时一定要小心。订阅这个事件的Actor可能还没有完成初始化。所以最好在前面加上一段延迟。

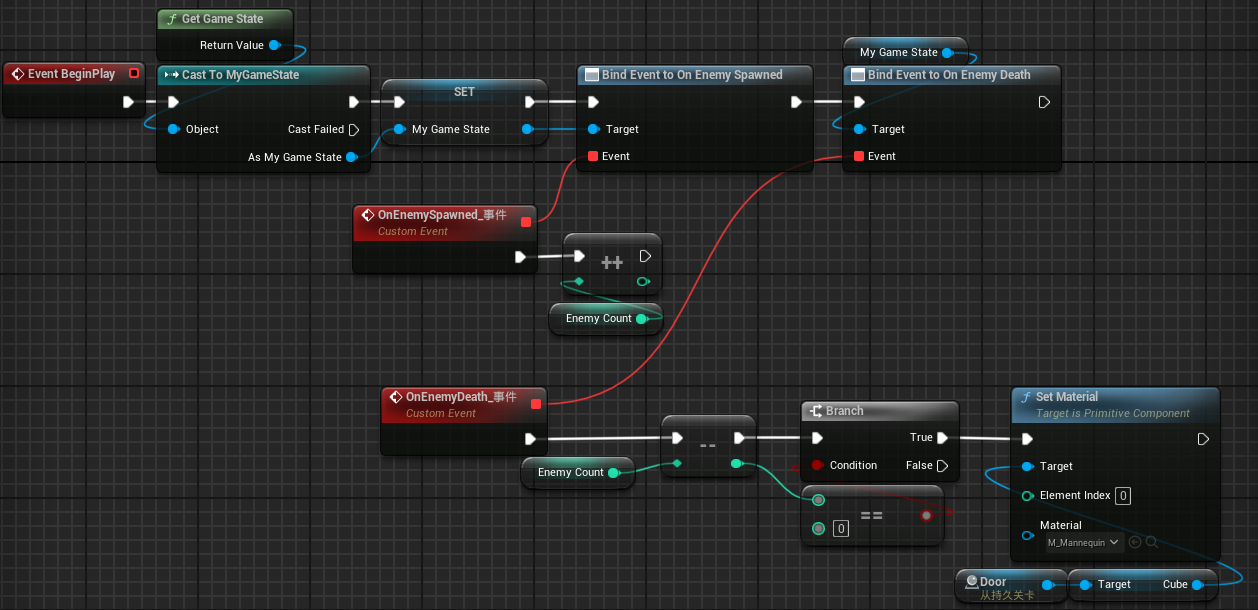
回到Widget

截止到现在我们还在遍历所有敌人来获取列表长度，来初始化敌人的数量。现在用GameState来实现



此时，我们甚至可以动态生成Enemy，同时让Widget动态修改数值。

现在处理Door蓝图。在关卡蓝图中，



完成。

截至目前，我们在GameState中发布事件，这就意味着敌人需要知道一个叫做Call OnEnemyDeath的事件，所以我们需要将GetGameState强转成MyGameState。

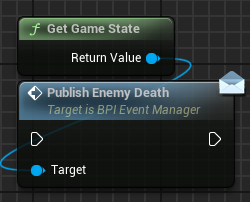
下面我们通过接口来将所有的CastTo节点消掉

创建BPI\_EventManager

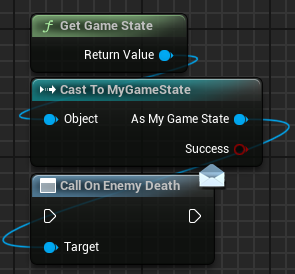
创建函数PublishEnemyDeath和PublishEnemySpawn

此时回到Enemy蓝图，我们就再也不用CastTo了

这时，我们可以用



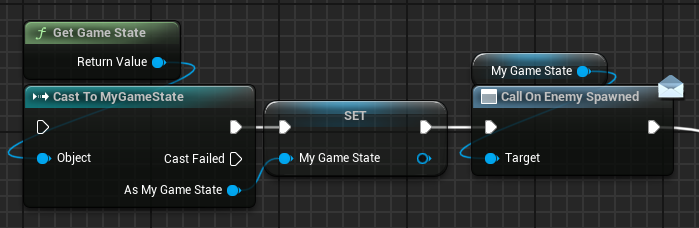
替换



用



替换



当然，回到EventManager，实现这个接口



这么做的唯一缺点就是你需要承担这部分调用事件分发器的开销，但这能让发布者更干净。而对于订阅者，依然需要CastTo。因为你需要一个被发布事件的引用，而且你不能把他添加到蓝图接口中。