设 计 模 式

同乐技术

目 录

[设计模式 3](#_Toc51)

[组件模式 4](#_Toc21655)

[对象池模式 7](#_Toc6839)

[对象池模式 7](#_Toc1187)

[开放思维 9](#_Toc16232)

[观察者模式 11](#_Toc23851)

[观察者模式（发布者/订阅者模式） 11](#_Toc12959)

[发布者 11](#_Toc6976)

[订阅者 11](#_Toc1748)

[订阅者（开门） 12](#_Toc29280)

[中介者模式 14](#_Toc2260)

[中介者模式 15](#_Toc20067)

[战斗管理器案例 15](#_Toc26215)

[事件管理器 19](#_Toc6646)

[事件分发器与蓝图接口 24](#_Toc11111)

[接口 24](#_Toc21992)

[事件分发器 27](#_Toc20571)

[混淆 29](#_Toc28619)

[区别 29](#_Toc12047)

[性能优化 30](#_Toc31420)

[参考文档 31](#_Toc5309)

# 设计模式

设计模式的核心目的就是优化游戏，增加游戏帧率，让运行更流畅。

CastTo节点要尽可能地少用，会增加内存占用

大型程序项目设计中，尽可能的解耦

# 组件模式

Actor组件，对应普通组件，

举个例子，在角色类蓝图的组件中，横线下面的就是Actor组件



Actor组件我们是可以自定义的

创建蓝图类，选择Actor组件

既然是组件，我们可以选择一个比较通用的组件

比如选择健康组件

健康组件自己有一个变量，命名为健康值，float型，可编辑实例，生成时公开。

然后我们要给健康组件定义一个函数，命名为“触发伤害”。函数需要一个形式参数，来传递伤害值



然后我们需要创建一个事件分发器，对事件分发器进行调用。

Actor组件和使用该组件的Actor之间通信的方式：事件分发器

在Actor组件中调用一个事件分发器就是再说：

“嘿，我要让所有对这件事感兴趣的人直到角色正在触发伤害”。但是Actor组件本身并不决定在受到伤害时除了减少生命值之外应该做什么。（组件本身并不关心事件被调用时会发生什么，只是通知使用组件的部分：事件发生了）

然后我们可以再创建一个事件分发器，命名为死亡。用于在角色死亡时发出通知，



待会就会看到如何在玩家角色中配置事件。

然后我们需要在一个蓝图中使用这个组件

创建一个角色蓝图，在组件中添加刚创建的健康组件，然后我们就可以在细节中看到我们刚创建的变量和事件分发器。

进入事件图表，创建该组件的节点，可以在细节的事件中看到我们刚创建的两个事件分发器。同时也可以通过组件节点引出得到健康值



同样的也可以通过组件节点使用“触发伤害”函数



现在想在玩家角色的头顶上添加一个血条Widget，我们可以在Actor组件内部来完成这项工作。

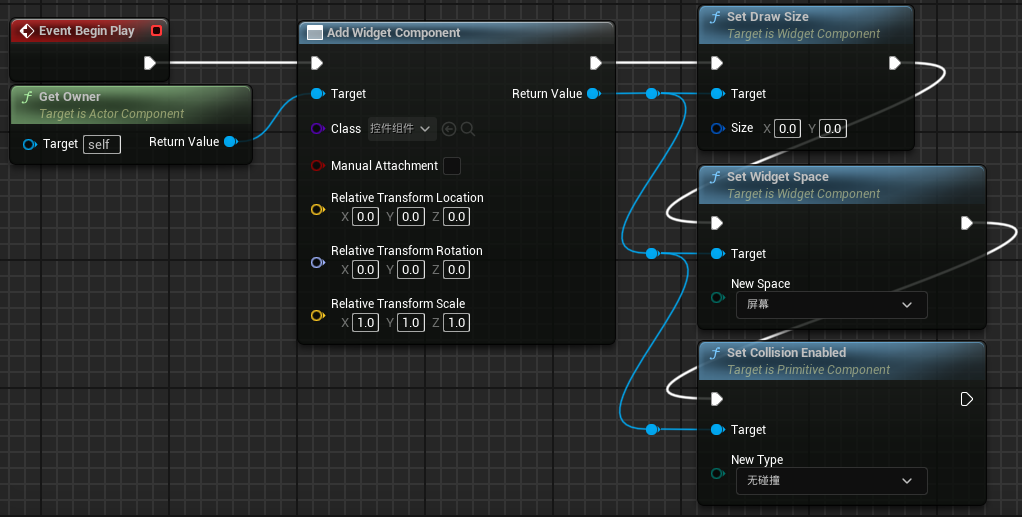
你可以使用一个Actor组件向Actor添加这个组建所依赖的其他组件。

如果你想在角色的头上添加一个血条，通常的做法是给Mesh组件添加一个Widget组件，然后把widget放到头上

现在我们用Health组件来完成这一要求。

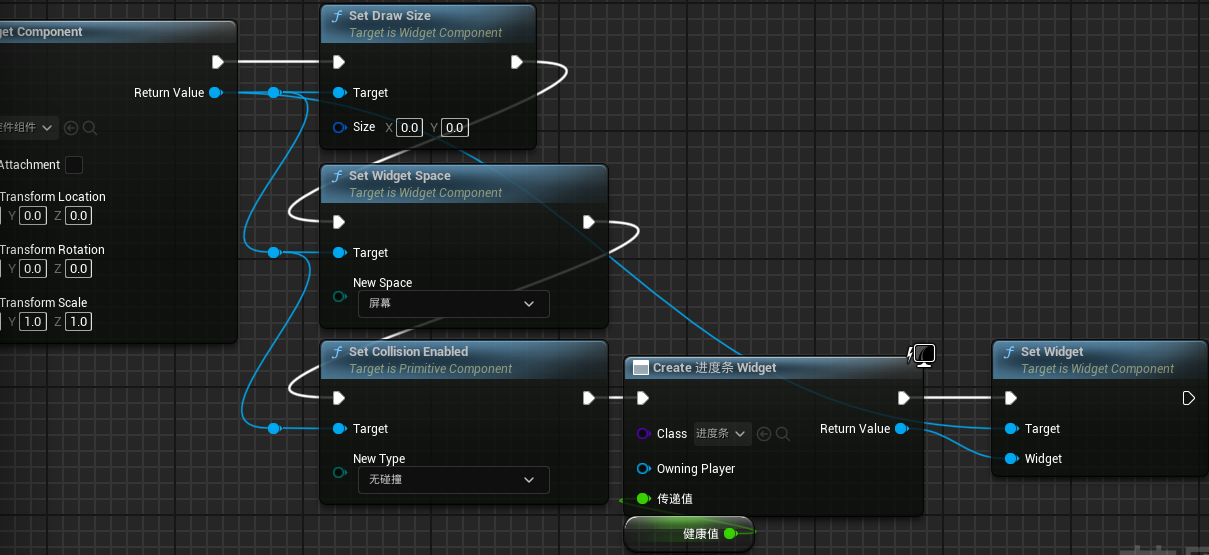
在健康组件中GetOwner，然后AddComponentByClass，选择UE自带的WidgetComponent。

然后对WidgetComponent进行SetDrawSize、SetWidgetSpace、SetCollisionEnabled



现在我们创建一个控件蓝图，蓝图中在画布面板下只有一个进度条。将进度条设置为变量后进入事件图表，创建一个float型参数，命名为“传递值”。我们要通过传递的值（当前生命值）与最大生命值（10）进行计算求出进度条的位置。

然后回到健康组件，在前面的节点完成之后CreateWidget。Create之后不用add to viewport，而是SetWidget。这个函数接受一个Widget引用作为参数，这个引用是来自于Add Widget Component节点。



这样就可以实现。

Attack组件同理

https://www.bilibili.com/video/BV1ygHeeSEXv/?spm\_id\_from=333.788.recommend\_more\_video.1&vd\_source=8dc4bce5a7680a01e3e5813461aae91c

# 对象池模式

## 对象池模式

对象池模式有两个核心：AC\_ObjectPool和BP\_PooledActor。

PooledActor的核心就是，这个Actor是否正在使用，以及他的使用条件和停用条件。

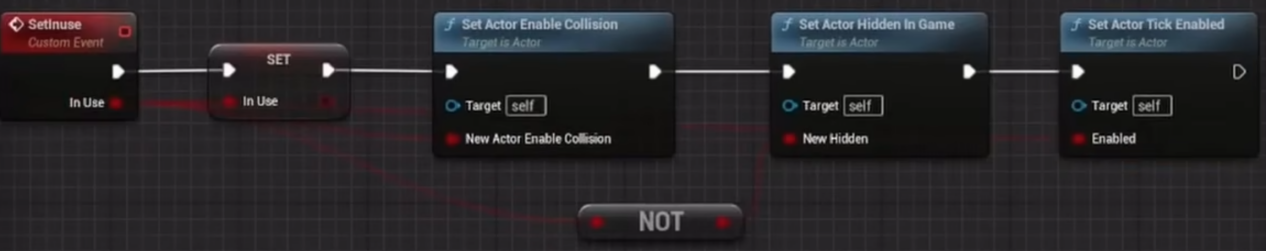
在Ali Elzoheiry的视频中，停用条件是弹道的存活时间。

在视频中，创建了两个变量：Bool的InUse和Float的LifeTime

其中的LifeTime是可编辑实例生成时公开。

在PooledActor中创建函数SetInUse。

SetInUse和核心是，当Actor没有使用时，设置碰撞和打开隐藏Actor，并且关闭Tick

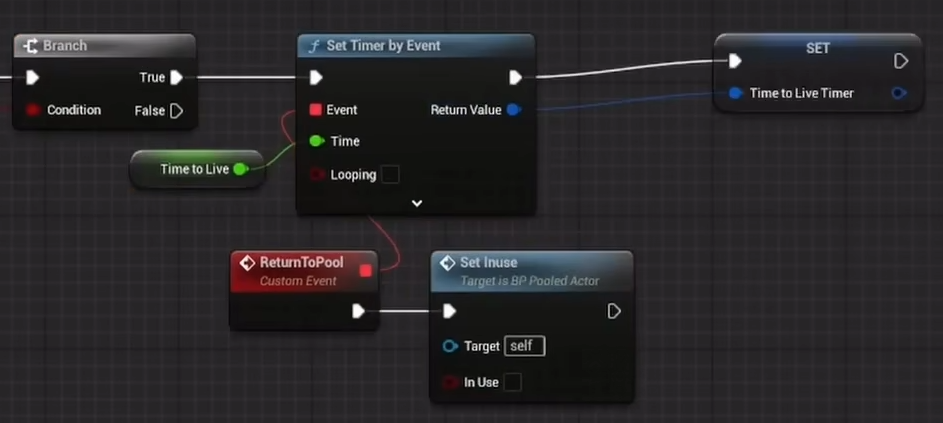


SetActorEnableCollision（true则有碰撞，false则无碰撞）

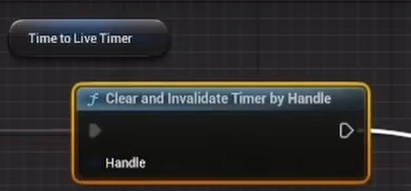
SetActorHiddenInGame（true为隐藏，false为实体）

SetActorTickEnable

然后定时检查一下我们是否将对象设置为激活状态



在Branch之前，将Timer的变量进行清空



最后将InUse设置为默认False。无论是设置变量的默认值，还是在构造函数或EventBeginPlay中实现都可以。

然后就是AC\_ObjectPool

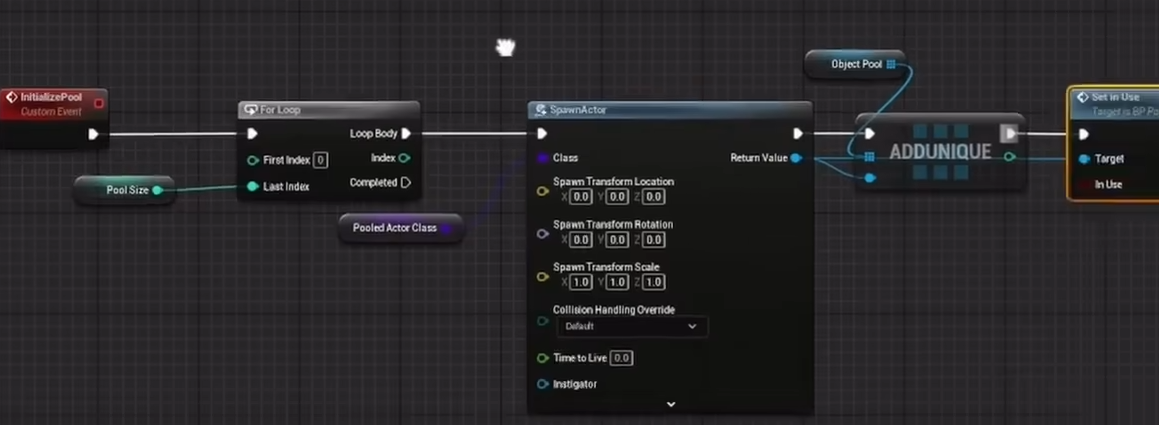
AC需要知道对象池的对象的类，创建变量，变量类型为PooledActor的类引用，不是数组。设置为可编辑实例。

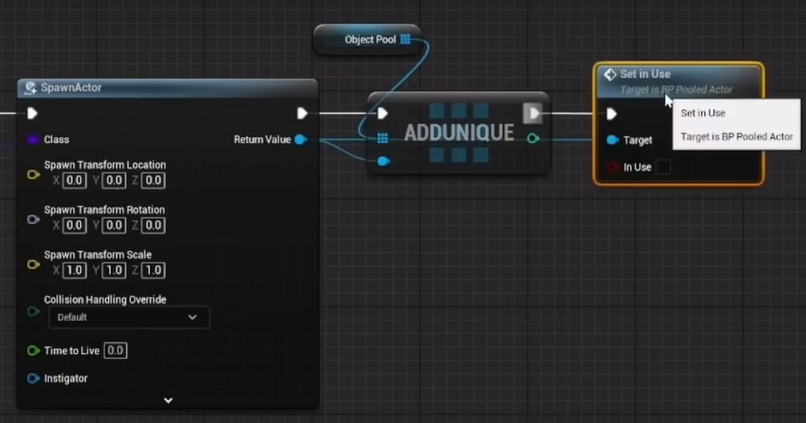
还有一个变量PoolSize，int型，用于描述对象池初始化的时候会生成多少对象实例。同样，可编辑实例。

最后一个变量就是PooledActor，是BP\_PooledActor的数组类对象引用。

然后就是实现函数。

InitializePool



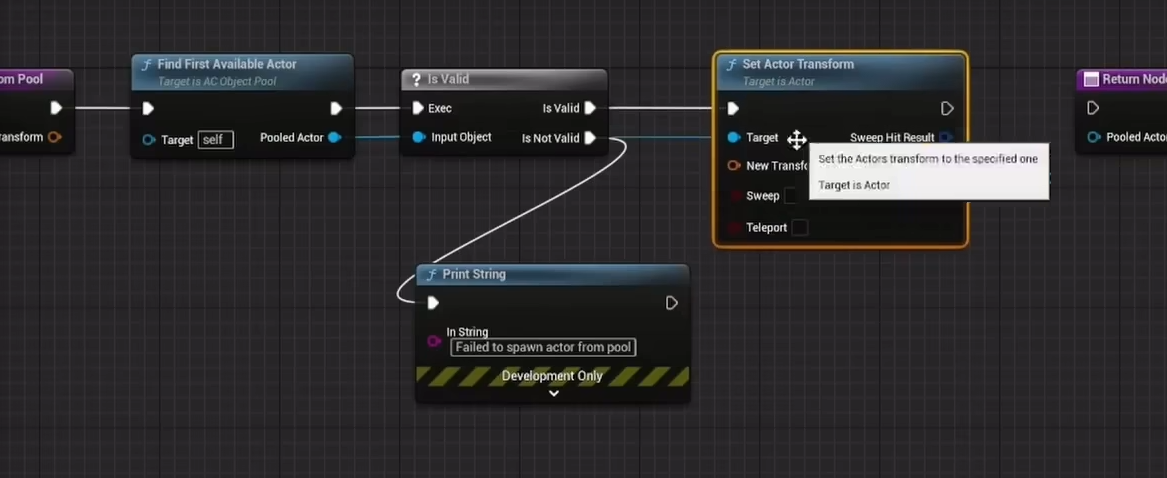


可以在EventBeginPlay中调用该函数

然后我们需要两个函数帮我们在对象池中检索对象

FindFirstAvailableActor，返回数组中第一个可以使用的Actor。

SpawnFromPool，用于将Actor的状态设置为激活，然后把他放置在世界场景中的一个location。





## 开放思维

有两种类型：个体对象池和全局对象池

如果是敌人特别的场景，比如 雷电这种类型的游戏，使用全局[对象池](https://search.bilibili.com/all?from_source=webcommentline_search&keyword=%E5%AF%B9%E8%B1%A1%E6%B1%A0&seid=5241250486286361902" \t "https://www.bilibili.com/video/BV1wz421h7iR/_blank)比较靠谱

但一般来说，使用个体对象池就可以，而且对于一个角色which can spawn a lot of Actors，使用个体对象池是非常合适的。

对象池模式是一种设计思想，怎么实现并不重要。

在Ali Elzoheiry的视频中，对象池是通过Actor组件来完成的。这自然可以成为一种思路。Actor组件类型的对象池的一个好处就是可以代码复用，当有别的对象池的时候不用复写代码。

但是这种对象池只适用于子弹类型不多的，如果游戏中出现多种类型的子弹，也不能真的去把每种子弹都生成上百个

所以Ali Elzoheiry视频里的是初级版，有的做的自适应对象池，就可以实现说的这种多类型子弹。举个例子，你不开枪时，没有对象池，当你开枪之后，新建一个对象池，将你发射（生成）的子弹一个一个存入到对象池之中，然后重复利用。根据射击频率进行适配上限，比如你第一次打个人，开了五枪，对象池存入了五发，【假如1、你下一次射击的时候<=五发，这五发就可以重复利用射出去了】，【假如二、你下一次射击的时候超过了五发，比如十发，那么之前保存的五发先射出去，之后生成新的五发子弹跟着射出去，而这一次有十颗子弹保存到对象池中】。你不可能按死扳机一直打吧，就算按死也是有限制的，弹匣容量就那么多，如果是机枪一类的大容量弹匣，可以设置过热，亦或者计算射速、射程、弹速设定一个值，比如每分钟1000发射速的冲锋枪，最大射程200米，弹速600米/秒，你的子弹不会停留超过一秒，那每秒最多17发子弹，取个容错值，17+5=22吧，如果玩家按死扳机，对象池最多保存22发子弹，就足够使用了。 另外给对象池一个寿命变量，可以是指定的时间（比如60秒），这个在你第一次开枪之后生成的对象池在60秒之后自动销毁，或是进行判断，超过特定时间（比如60秒）没有再开枪，则销毁对象池。直到你再次开火才会生成新的对象池来保存你的子弹。

# 观察者模式

有这样一种情况：有一个Widget显示剩余角色数量，每当杀死敌人时更新数据。还有一个门，只有当所有敌人都死亡时他才会打开。所有对象之间都需要互相通信：Widget、门、Level，甚至在Enemy和Player之间也需要发送。如何正确的实现这一切呢？

## 观察者模式（发布者/订阅者模式）

问题的重点就是，游戏中不同对象之间的通信与信息共享。我们该如何创建一种可扩展的、易于维护的方式去实现。

就从上面说的这个例子，我们该如何实现角色死亡时动态更新剩余角色数量？

一种错误的方法是直接获取对Widget的引用。虽然能实现功能，但是这样就耦合了角色和Widget。

观察者模式是基于事件发生时进行信号广播的模式。游戏中其他接收到广播信号的对象，就会对这个事件进行反应。这意味着你可以拥有一个对象，在某处广播一个信号，却不需要考虑会发生什么。游戏中的其他对象可以监听这个事件，并对其作出不同的反应。

在蓝图中，这是非常容易实现的，只需要定义一个事件分发器EventDispatcher就可以实现。

在C++中，则需要定义Delegate。

这里，我们只关注蓝图中的实现。C++中的思路类似，只是实现方式略微不同。

通常情况下，事件分发器会命名为OnXXX，因为是一个触发器。

## 发布者

打开Enemy蓝图。现在我们想把Enemy类作为发布者来广播一个信号，来广播已经死亡。

在Enemy类中创建事件分发器命名为OnDeath，创建自定义事件Death。

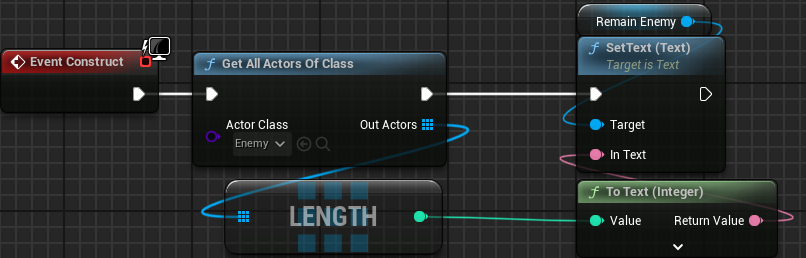


如果想通过事件分发器传递参数，在事件分发器的细节面板最上面的输入中加上参数。这里我们并不需要参数。

发布者的行为已完成

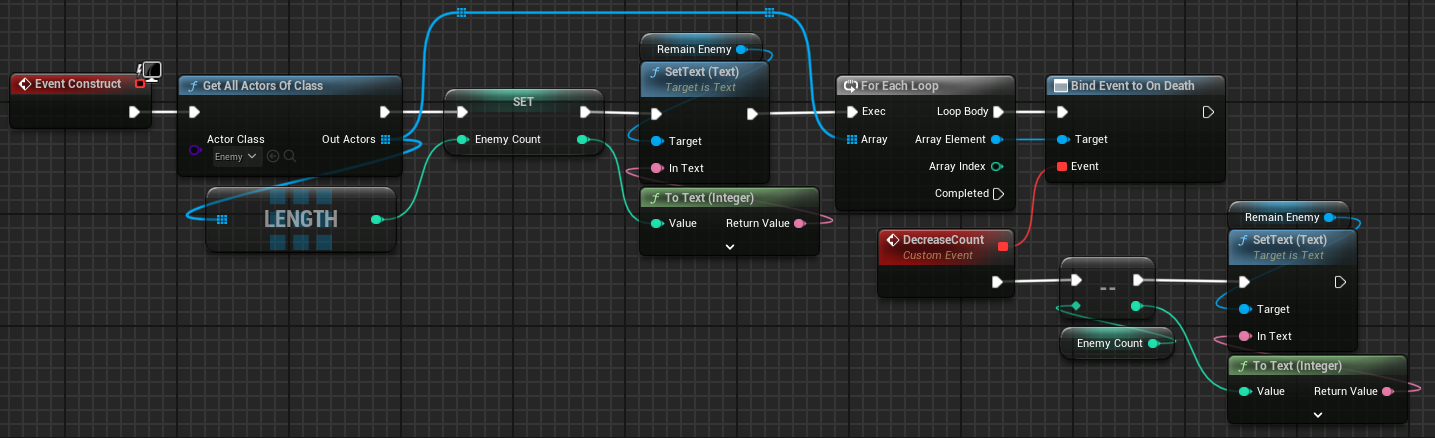
## 订阅者

在WBP\_ReaminEnemy中，



接下来我们需要在WBP中订阅事件

为了订阅事件，需要使用bind关键字



完成功能

观察者模式的好处就是可以用多个订阅者订阅同一个事件去完成不同的任务。

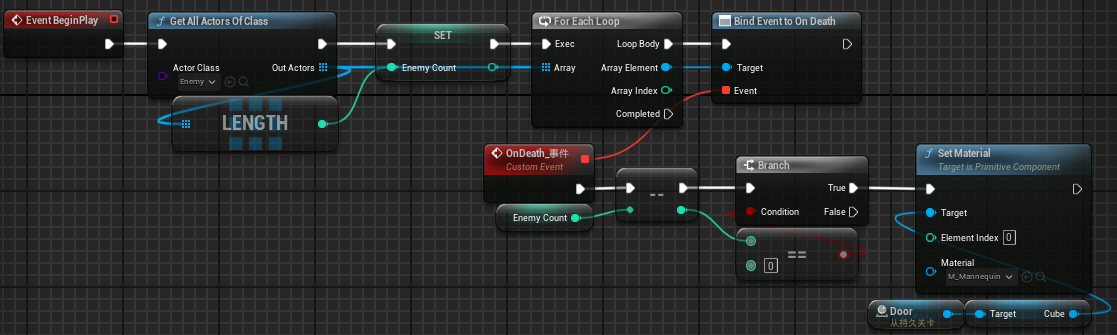
## 订阅者（开门）

游戏中有很多门，他们的开关原因可能是不同的。所以我们可以在关卡蓝图中实现。

在关卡蓝图中添加门对象的引用



我们设计：当所有敌人都死亡的时候，设置门的材质，代替开门的功能。



完成功能。

C++项目下的观察者模式：

Ali Elzoheiry 的虚幻引擎中的设计模式系列教程：观察者模式 https://www.youtube.com/watch?v=YFtLd-bKl-U

--------------------------------------------------------------------------------

设计模式就是如何编写良好代码的指南，无论您使用蓝图还是 C++，了解良好的软件实践都是必须的！

我希望本系列的第一个视频能够介绍何为设计模式，如何使用观察者模式。

这会通过「事件分发器」的案例来完成，虚幻引擎中的事件分发器使用Delegate，是观察者模式的实现。

通过蓝图，可以调用事件分发器，但有某些限制，本系列将讨论这些限制。

下载项目文件并在 Patreon 上支持我：

https://www.patreon.com/c/Alizoh/home

加入我的 Discord 服务器： https://discord.com/invite/AEnM4wvYzN

额外资源：

https://refactoring.guru/design-patterns/observer https://gameprogrammingpatterns.com/observer.html https://dev.epicgames.com/documentation/en-us/unreal-engine/delegates-and-lamba-functions-in-unreal-engine?application\_version=5.2

# 中介者模式

前置：观察者模式

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

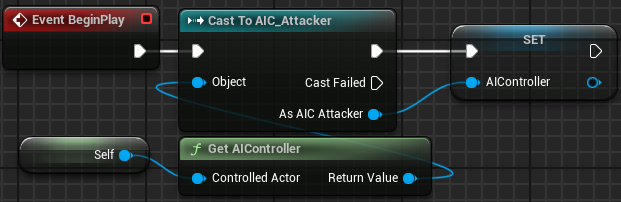
这就是我们要在告诉Enemy去Attack或Retreat或Wait时做的，要通过AIController的函数，来修改这个变量。

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。

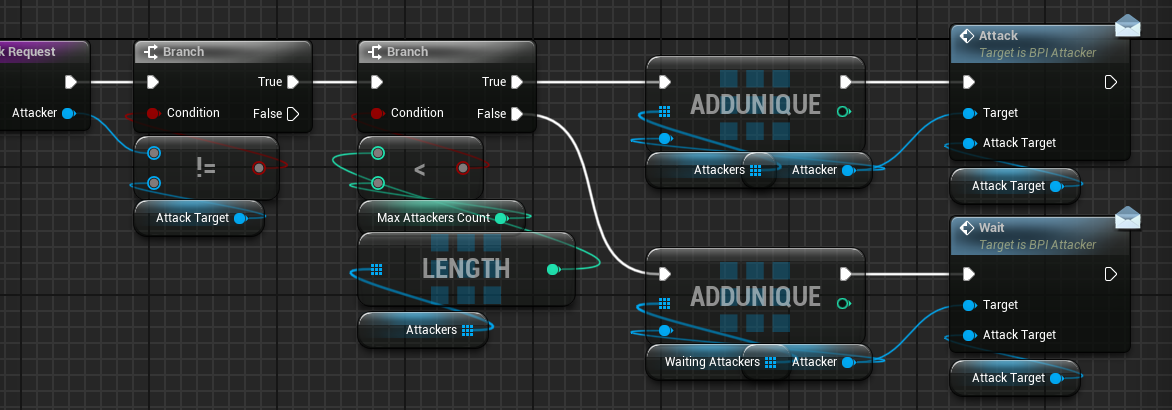
接下来关注战斗管理器如何使用他们

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。

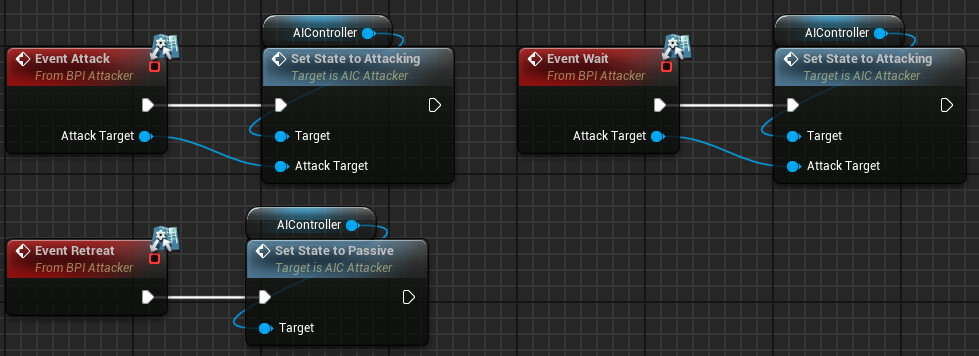
接下来在Enemy和Player中实现接口并调用函数

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

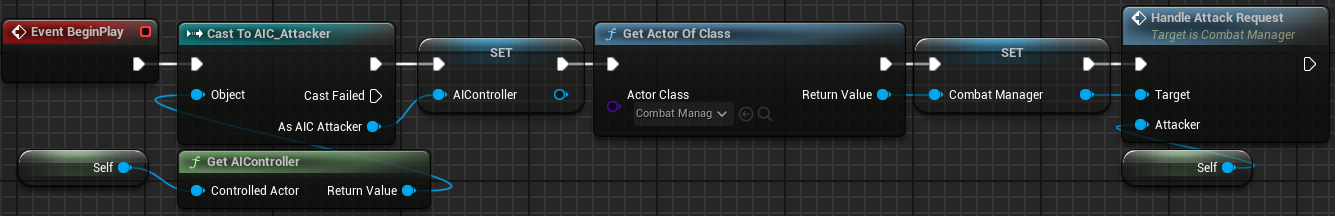


在Enemy类中，类似。

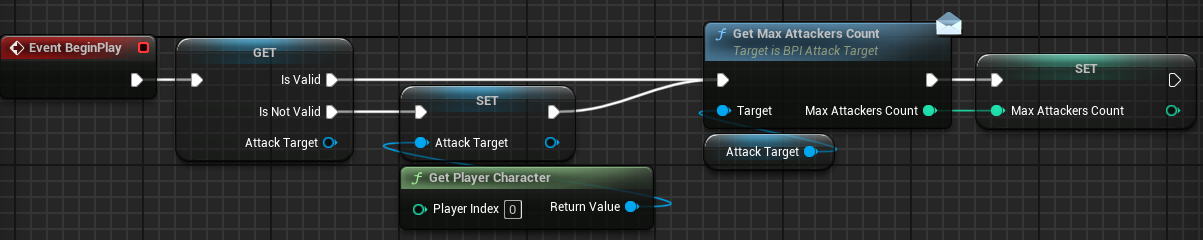
创建接口的实现事件



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。



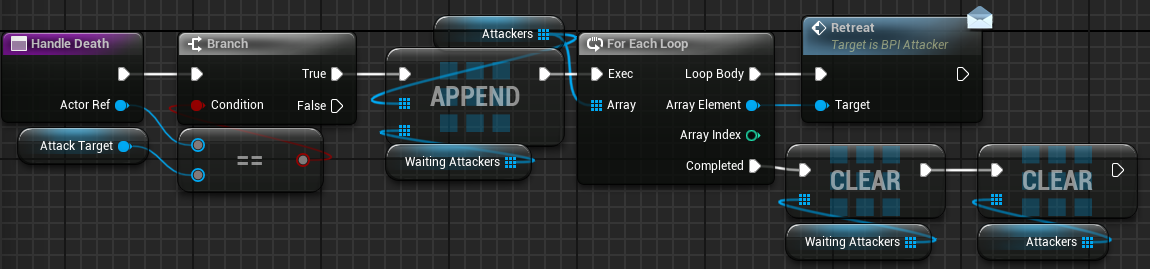
现在就可以开启游戏观察到有3个Enemy移动到自己的位置

（记得把CombatManager在场景中创建一个实例）

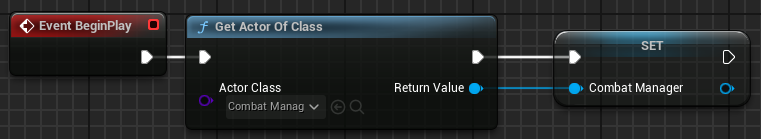
改GetMAxAttackersCount函数的返回值也可以正常运行。

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

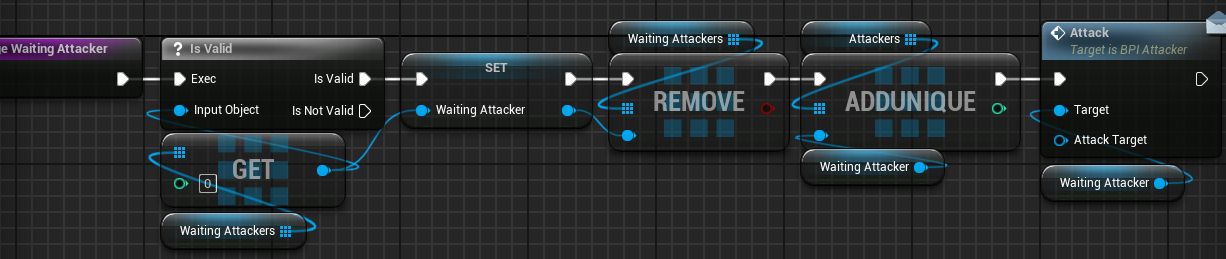




之后，当Player死亡时，Enemy就会回到原点。

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。

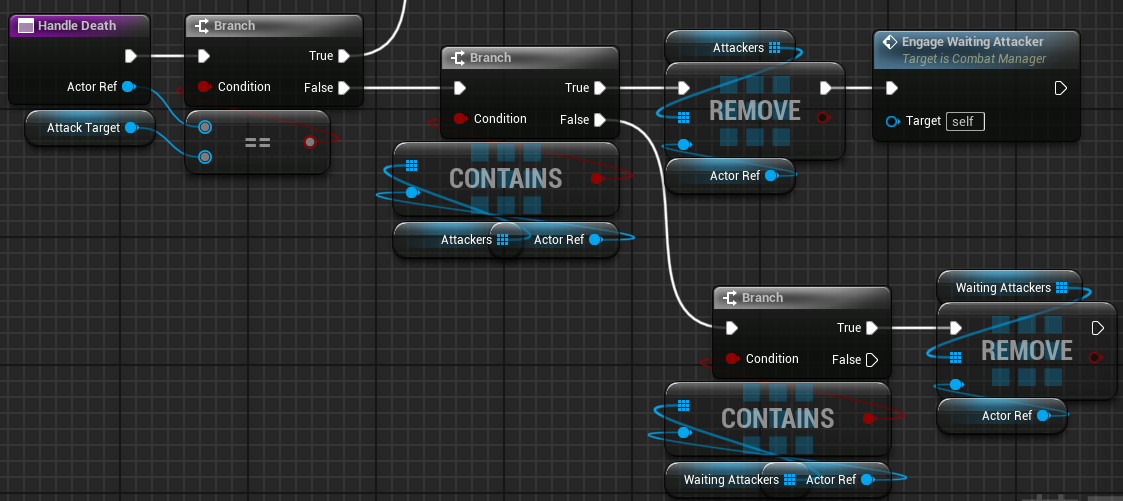
在ConbatManager中创建函数EngageWaitingAttacker



这里的WaitingAttacker是局部变量

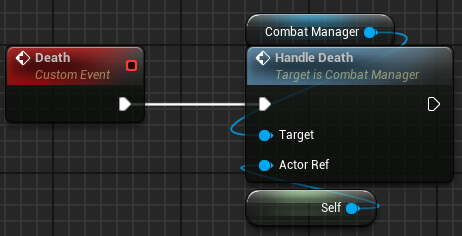
我们在Enemy死亡时调用这个函数

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。

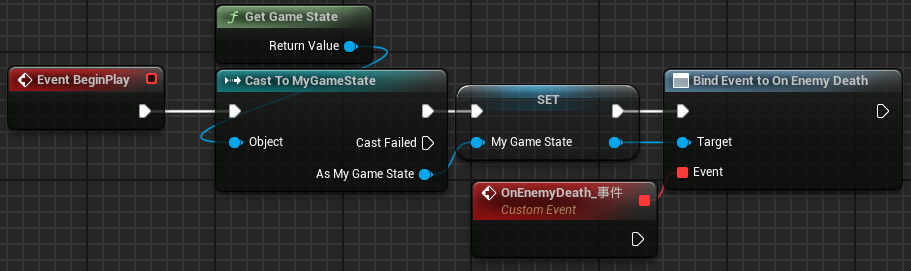
现在，这个GameState对象将会使我们的事件管理器，他的职责是，分发我们游戏中所有的事件。

项目用到观察者模式的项目

回到Enemy中，删掉OnDeath事件分发器

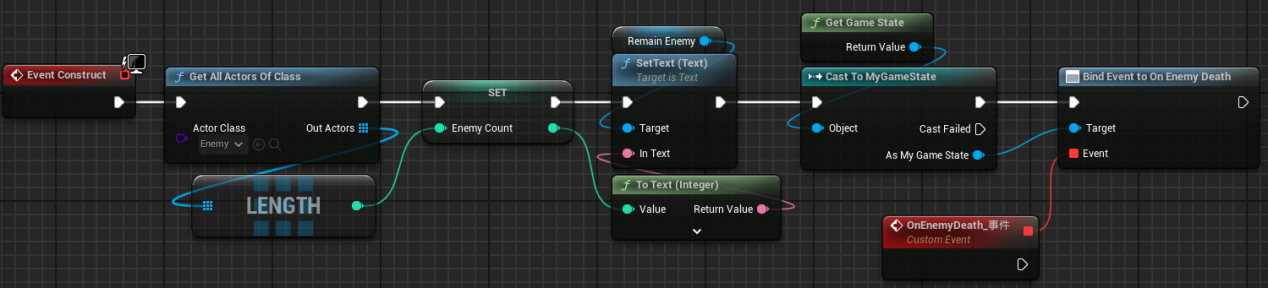
然后进入GameState，添加一个新的事件分发器，命名为OnEnemyDeath。

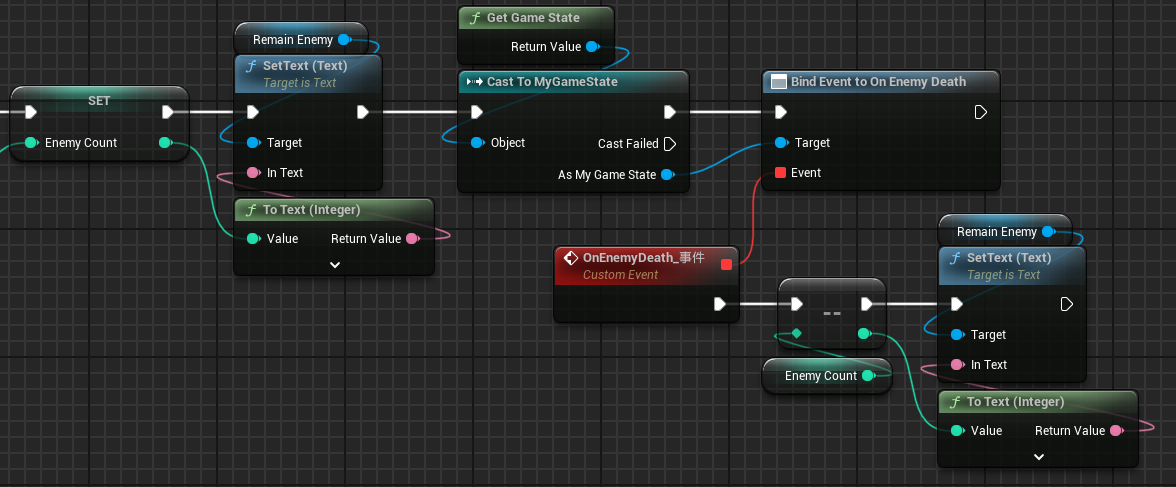
现在回到Enemy蓝图，取而代之的是调用GameState中的事件分发器



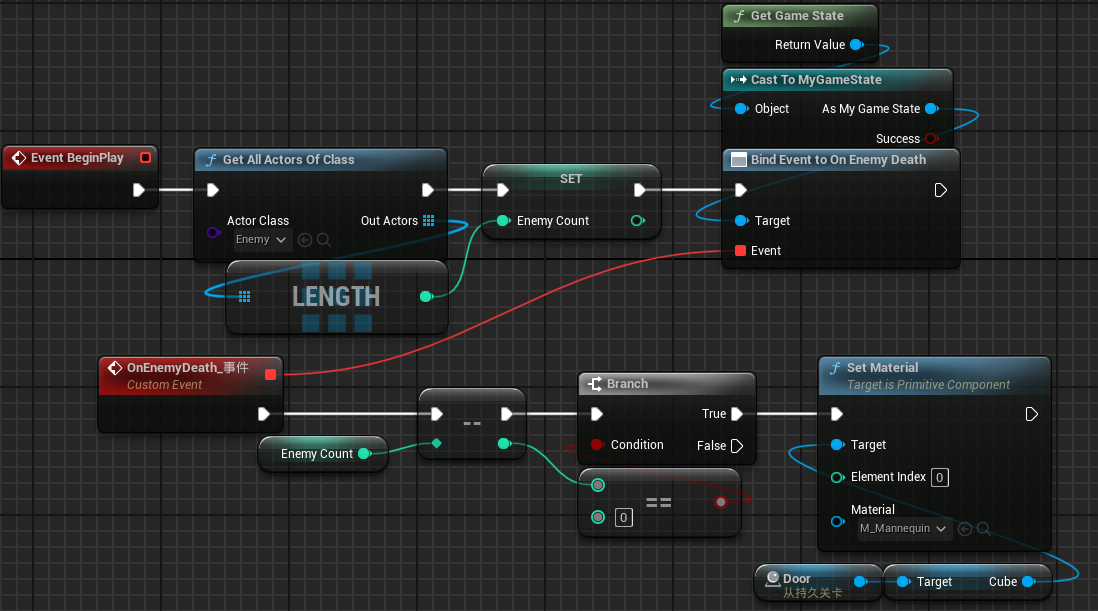
（后面会移除CastTo节点）

然后进入Widget WBP\_RemainEnemy，





然后进入关卡蓝图做一遍相同的事情。



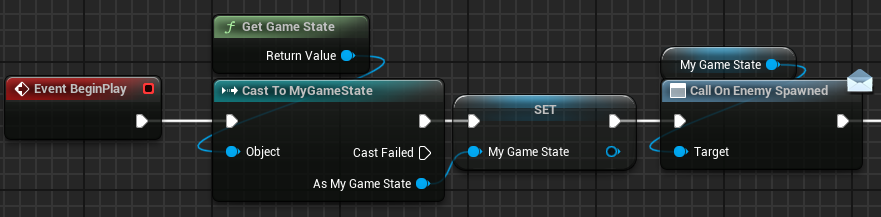
（记得在Enemy类中调用事件分发器）

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。

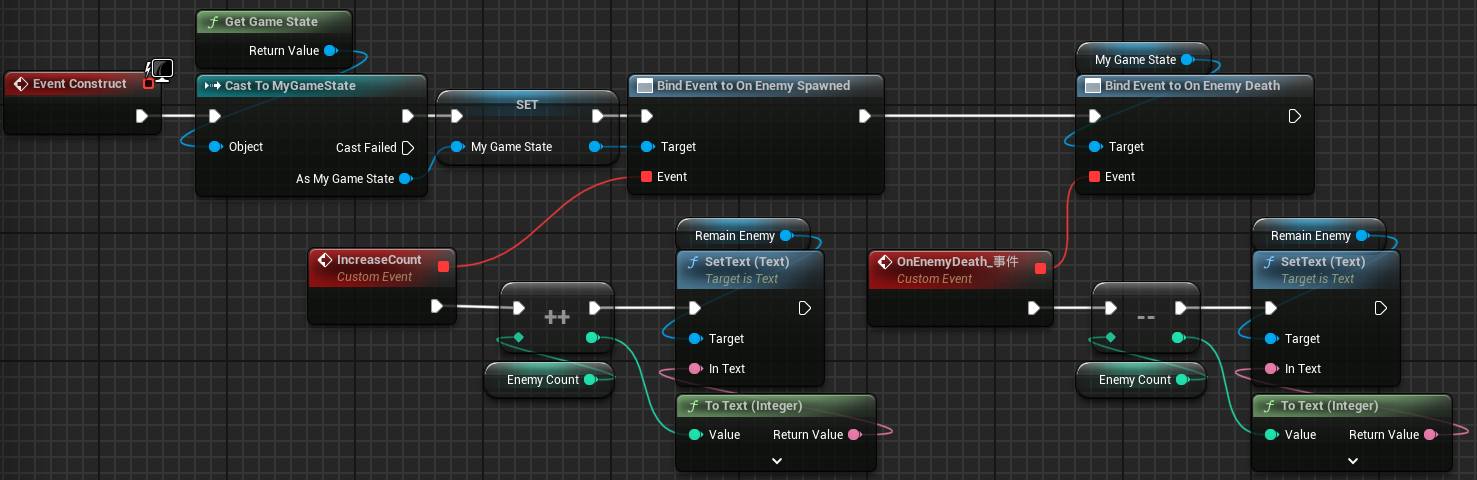
在Enemy的BeginPlay中



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。

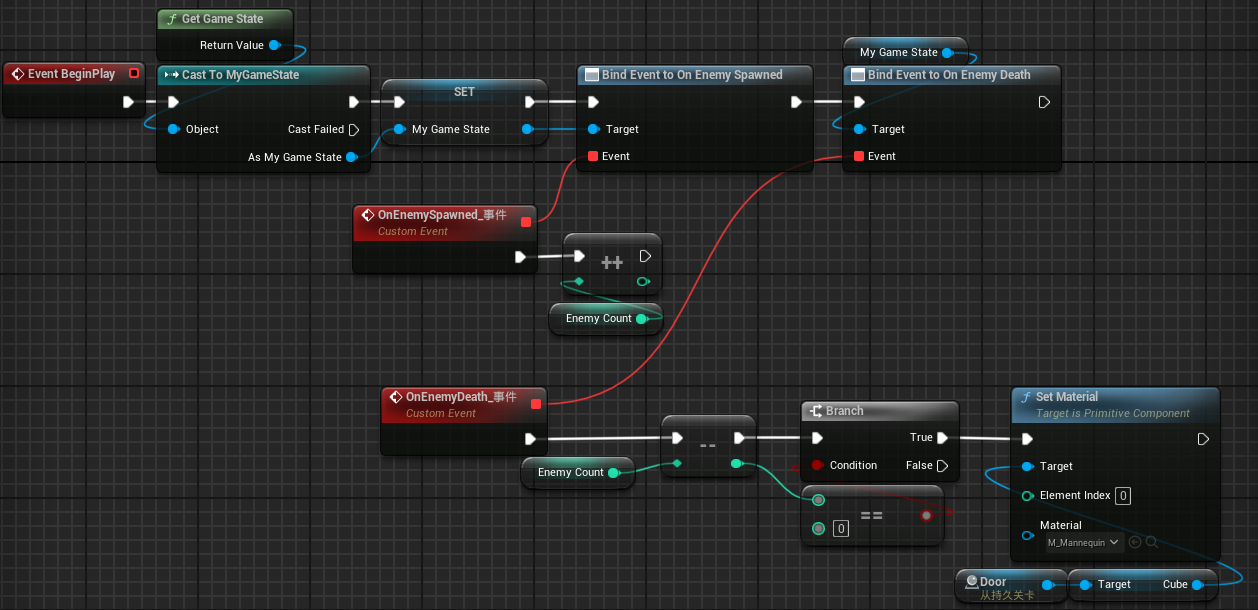
回到Widget

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。

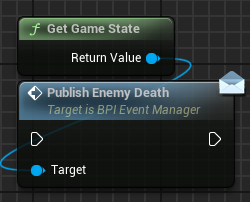
下面我们通过接口来将所有的CastTo节点消掉

创建BPI\_EventManager

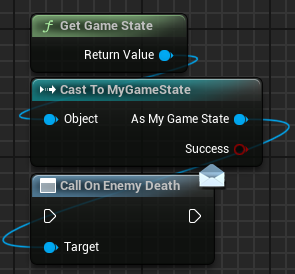
创建函数PublishEnemyDeath和PublishEnemySpawn

此时回到Enemy蓝图，我们就再也不用CastTo了

这时，我们可以用



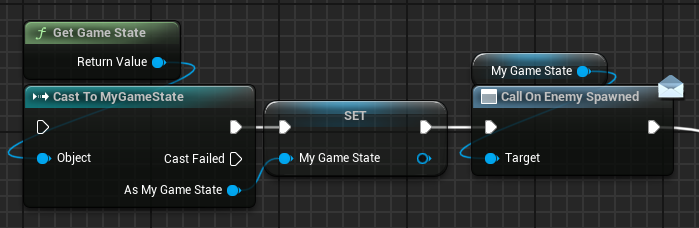
替换



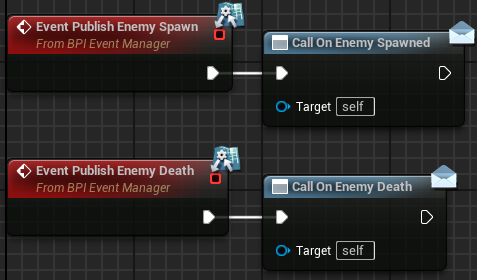
用



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。

# 事件分发器与蓝图接口

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。

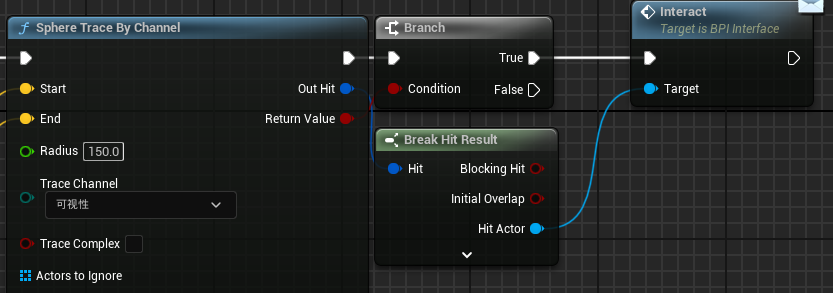
现在进入到Player类中，当摁下某个按键时，进行一次球体追踪检测



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。

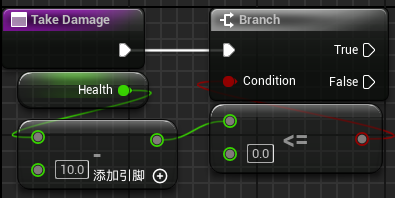
还有一个用处，是使用事件分发器构建通用的角色组件。这些组件可以应用于任何角色。

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

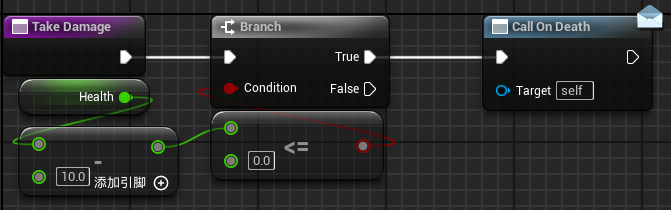
创建变量：Health

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



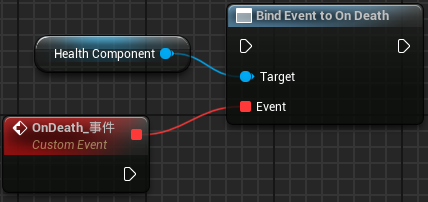
现在进入Player蓝图，添加这个组件

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。

# 性能优化

CastTo的性能优化方法：

1. 使用蓝图接口
2. 使用Tag，当
3. 使用IsA函数
4. 使用C++的CastTo

IsA是有一定开销的，如果想减少IsA的开销可以先使用Tag过滤一部分。

但是，无论怎么搞，性能最好的是C++

# 参考文档

组件模式：<https://www.bilibili.com/video/BV1TnmUYrEMW/>

对象池模式：<https://www.bilibili.com/video/BV13cmkYFEvv/>

观察者模式：<https://www.bilibili.com/video/BV1bDy5Y3Exk/>

中介者模式：<https://www.bilibili.com/video/BV1eSS7YTEgR/>