设计方案与技术参考文档

# 目录

1. 文档简介
2. 模块总览
3. 核心模块设计参考
4. 交互系统
5. 剧本演出系统
6. 基于事件的GAS
7. 物件系统
8. 技能物件系统
9. 其他设计与技术参考
10. 碰撞预设与碰撞代理
11. 摄像机系统设计
12. 输入系统设计
13. UE单例模式设计
14. 处理模型遮挡视线
15. 限制玩家视野
16. 开发总结
17. 开发流程、蓝图与C++
18. UE插件开发经验

# 一．文档简介

此文档是对本人在2023.11-2023.12时间段内开发的ARPG游戏DEMO（后文统称为Demo）的设计方案与技术的总结，以供大家参考。

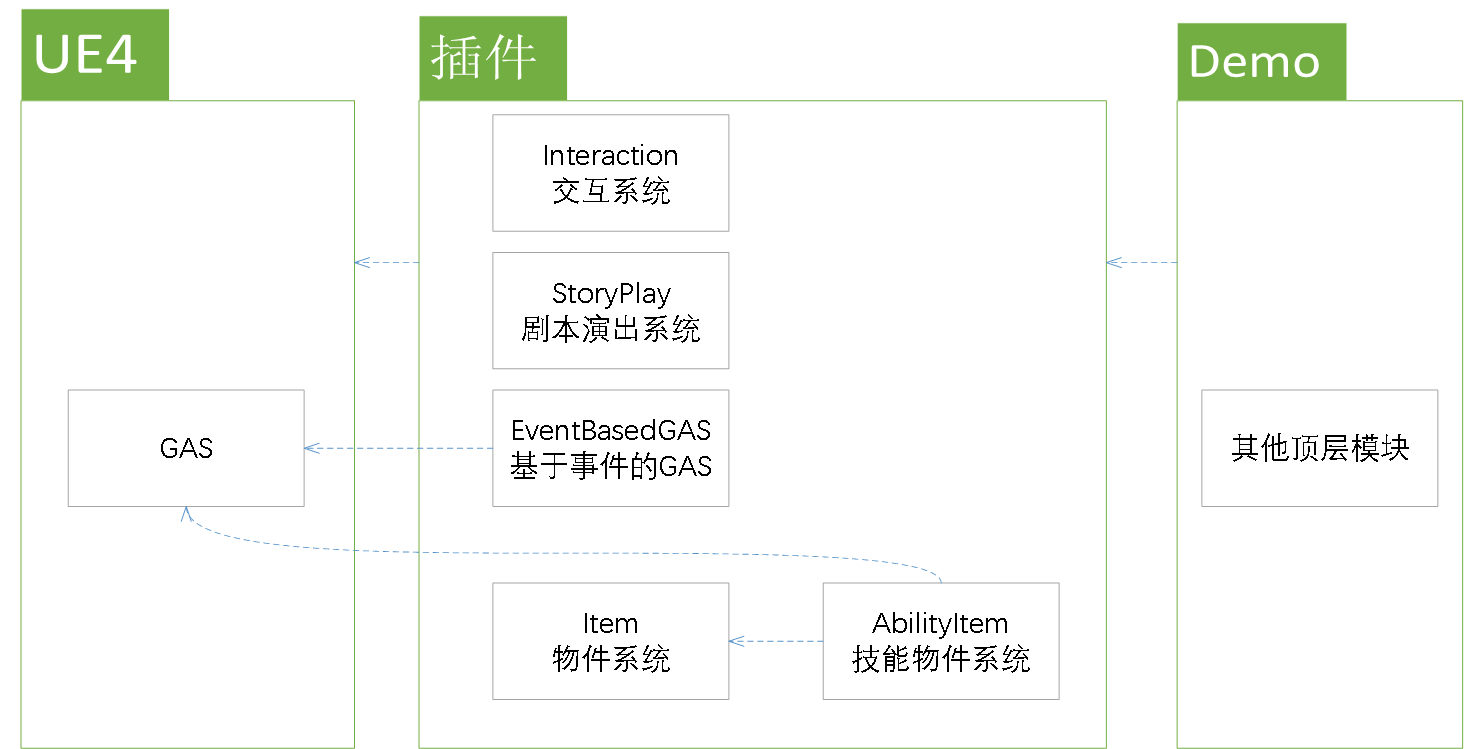
在撰写文档的期间，本人对整个Demo进行了设计模式的调整与重构，对高频逻辑进行了模块化与插件化，并完成了对外兼容与验证。

文档整理出游戏的五个核心模块系统，同时也对游戏使用的其他的设计方案与技术进行了总结。文档最后提出了一些本人在UE学习与Demo开发时总结出的一些经验，希望能为各位读者提供有效的参考意见，也欢迎各位提出自己的看法与建议。

文档中出现的架构图只包含对外接口的UObject与部分关键的关系。

# 二．模块总览

下图呈现了Demo中各模块之间的依赖关系：



五个核心模块：交互系统、剧本演出系统、基于事件的GAS、物件系统、技能物件系统，已完成对外兼容与解耦。其他顶层模块依赖于这些核心模块。

# 三．核心模块设计参考

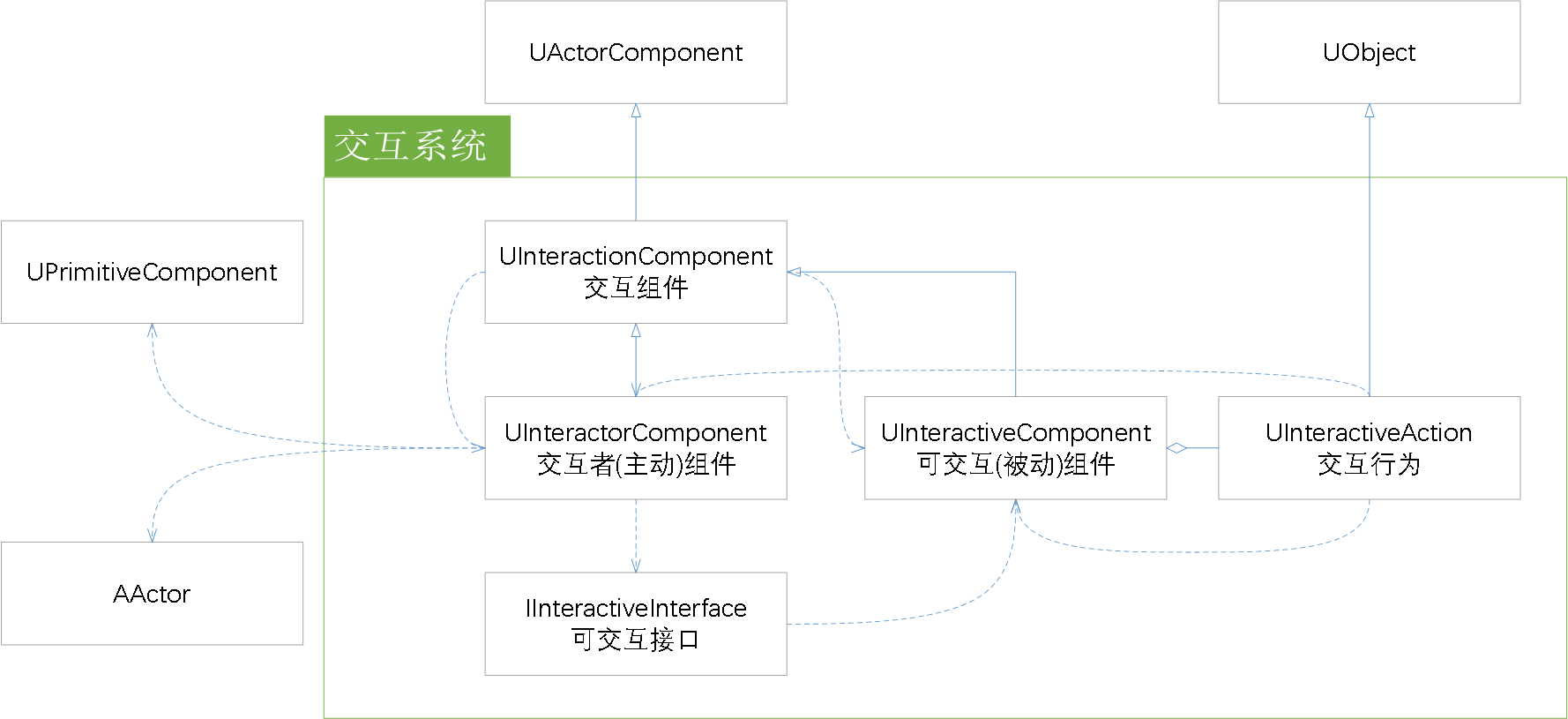
此节对上述的五个核心模块进行了简要说明，说明内容主要包括：设计背景，类架构图，对外接口与使用说明、内部逻辑流程。

## 交互系统

Ⅰ设计背景

在游戏中，玩家需要经常与场景中的某些道具或NPC进行交互。当玩家靠近可交互对象，系统会提醒玩家可以交互，玩家可以与对象主动交互，对象也会相应地响应交互。此系统即是对这一流程的抽象。

Ⅱ类架构图



Ⅲ对外接口与使用说明

* 1. UinteractorComponent

意为交互者组件，属于玩家。

为玩家Actor创建UInteractorComponent，并将之与负责检测交互范围的UPrimitiveComponent进行关联，可以直接绑定事件，也可以使用UInteractorComponent的BindPrimitive接口；

Actor中可以绑定UInteractorComponent提供的委托接口，事件主要包括：

1. 聚焦
2. 失焦
3. 交互开始
4. 交互结束
   1. UInteractiveComponent

意为可交互组件，属于可交互对象。

为可交互Actor创建UInteractiveComponent，并让Actor实现IInteractiveInterface，此接口使Actor可以返回一个UInteractiveComponent；

在Actor的UInteractiveComponent中指定UInteractiveAction的类；

Actor中可以绑定UInteractorComponent提供的委托接口，事件主要包括：

a. 聚焦

b. 失焦

c. 交互开始

d. 交互结束

* 1. UInteractiveAction

可交互行为。

通过继承UInteractiveAction，并根据需求重载其中的函数，为UInteractiveComponent提供实际的交互行为。

可编程逻辑主要包括：

1. 实时判断是否可以被Interactor聚焦与交互
2. 实时判断结束交互后是否自动释放Interactor的焦点
3. 处理被Interactor聚焦
4. 处理被Interactor失焦
5. 处理与Interactor开始交互
6. 处理与Interactor结束交互

Ⅳ内部逻辑流程

系统完成一次交互的逻辑流程大致依照以下顺序：

1. Primitive与OtherActor发生碰撞事件，Interactor处理碰撞事件；

②Interactor判断OtherActor是否实现了Interactive接口，若实现，则从接口获取Interactive组件；

③Interactive获取Action中的方法，判断Interactive组件与Interactor之间是否可交互，若是，则Interactor将Focus（聚焦）OtherActor；

Primitive与聚焦中的Actor碰撞结束，则会提醒Interactor对Actor解除聚焦。

碰撞中的Actor都会加入Interactor中保留，由Interactor选择聚焦（切换行动）

④Interactor可以选择正在聚焦的Actor进行主动交互，被交互的Interactive组件会获取Action中的方法进行调用。

## 剧本演出系统

Ⅰ设计背景

剧情演出是游戏中不可缺少的部分。目前流行的演出形式主要包括电影式演出与推进式文字冒险演出。

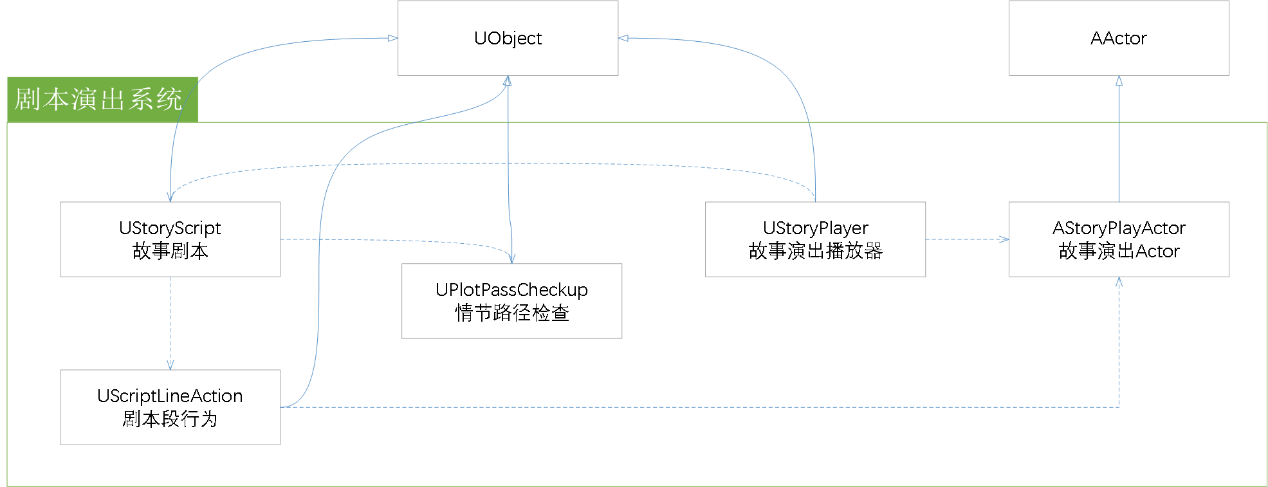
UE的LevelSequence能便利地构建电影式演出任务，但并不具有文字冒险式推进的功能，这些功能主要是：

进度控制

剧情分支

从结构上看，电影演出与推进演出分别是基于事件序列与文本序列。而本系统则是为了能兼容二者的优势而设计的。

Ⅱ类架构图



Ⅲ对外接口与使用说明

①UStoryPlayer

故事播放器，可类比ULevelSequencePlayer。

通过输入剧本UStoryScript与相关设定，创建AStoryPlayActor。

UStoryPlayer是AStoryPlayActor的对外接口，核心方法有：

1. Play 指演出开始
2. Forward 指演出推进，会要求输入选项索引

②AStoryPlayActor

故事演出Actor，一段故事演出的各类信息的物理上与逻辑上的载体，同时也可以视为演出的管理员。

UStoryPlayer无法控制演出进度，如果Actor标识处于正在推进状态（比如正在播放一段动画），UStoryPlayer将无法Forward到下一个剧情点，而使用AStoryPlayActor的接口则可以手动控制这些状态，因此AStoryPlayActor可以作为LevelSequence的Actor执行事件，比如让对话由LevelSequence内部时间序列进行推进。

②UStoryScript

剧本，驱动AStoryPlayActor的核心。

在创建剧本前，需要先认识几个结构体：

1. FScriptLine
2. FScriptLine:FTableRowBase
3. {
4. TSubclassOf<UScriptLineAction> LineAction;
5. };

FScriptLine指剧本中的一段，表示AStoryPlayActor中可以用Forward来分开的最小单位。

FScriptLine继承FTableRowBase，表示可以使用UDataTable来存储。

UScriptLineAction表示Story在Forward时需要执行的行为。

为Line创建Table时需要注意：

--单一的Action会产生闭包的不便利性，通常需要对Line进行拓展，模块提供了FScriptLine\_Complete，其结构供参考：

1. FScriptLine\_Complete :FScriptLineLine
2. {
3. ULevelSequence\* LineSequence;
4. FName ActorId;
5. FString LineText;
6. USoundCue\* LineSoundCue;
7. FName Flag;
8. };

--创建的DataTable需要满足RowName为Line的索引，即剧本以时间顺序排列，则Line的RowName为0-1-2-3-4-5-……，在使用表格工具编辑剧本Line时需要注意。这个顺序表示了情节的默认执行顺序。

1. FPlotPoint
2. FPlotPoint
3. {
4. UDataTable\* LinesDataTable;
5. FName TableKey;
6. };

FPlotPoint表示剧情点，标识了一个Table中的Line。

一张剧本中不能存在多个相同的剧情点。

默认创建的FPlotPoint会表示成一张剧本的起点。

1. FPlotPass
2. FPlotPass
3. {
4. TSubclassOf<UPlotPassCheckup> Checkup;
5. FName Flag;
6. FPlotPoint Endpoint;
7. };

FPlotPass意为剧情路线，Checkup用于实时判断该路线是否可以通过。

1. FPlotPipe
2. FPlotPipe
3. {
4. EPlotPipeType PipeType;
5. TArray<FPlotPass> Passes;
6. };

FPlotPipe意为剧情管线，目前支持两种PipeType，分别为Picker与Selector。

Picker管线表示对于Passes会顺序检测，在Forward时会选择第一个能通过的Pass进入；

Selector管线在Forward时，需要传入可通过Pass的参数，否则Forward会失败。

通过在蓝图中创建UStoryScript的子类，并为UStoryScript中的FPlotPoint创建管线，这样AStoryPlayActor在推进剧情时，会按照注册的管线的方式进行推进，而没有注册管线的FPlotPoint会按照默认方式（RowName的NextIndex）Forward下一个Point。

比如实现一个普通的顺序对话，只需要将对话整理成RowName为顺序的DataTable，然后在剧本中为起点创建一个指向该DataTable的FName=”0”的Pass即可。

剧本还提供允许向特殊的Point或全局委托进行绑定，比如可以为某一Point的Action注册Pre或Post事件，将特殊行为事件与较为一般的行为事件Action进行区分。

④UScriptLineAction

意为故事演出的行为，AStoryPlayActor在Forward时使用的行为。

可以在行为中播放LevelSequence。系统为LevelSequence提供一系列类似于LevelSequence中绑定Actor的方法，方便在LevelSequence绑定Actor。

若要实现在LevelSequence中播放文本的功能，可以让AStoryPlayActor加入LevelSequence的Actor，然后在LevelSequence中控制进度。

目前LevelSequence中绑定Actor无论是在C++还是在蓝图中都非常不方便，很难不为了每一个LevelSequence都创建对应的Action。

④UPlotPassCheckup

意为情节路线的检查点，会实时检查Pass能否通过。

比如：需要让玩家在对话中了解必要信息，NPC说完一段话后需要让玩家选择问问题，有三个问题选项，选择任意一个NPC都会回答，回答完后会回到这三个选项，回答过的问题选项会标灰色，必须三个问题选项都选择过了才会出现第四个选项允许玩家离开对话。

系统通过了与“剧情Flag”相关的接口，我们可以让Action中实现在Action执行后向系统注册一个“该选项已被选择过的剧情Flag”，我们在Checkup时就可以实现OpenIfFlagExist之类的操作了。

问题依旧在于闭包可能使得我们不得不创建非常多的UPlotPassCheckup类。后期或许需要通过更改结构取代Checkup的作用。

Ⅳ内部逻辑流程

开启一个剧本的故事演出的逻辑流程大致如下：

①输入UStoryScript与设置，创建UStoryPlayer与AStoryPlayActor，AStoryPlayActor加载相关数据；

1. UStoryPlayer调用Play方法，之后可以调用Forward方法
2. AStoryPlayActor内部执行Forward并依据UStoryPlayAction执行行为

在执行Forward前，Actor内存在CurrPoint，意为现在的剧情点，且在Forward前就会计算CurrPoint对应Pipe与其结果并保存，得到NextPoint或者Options；执行Forward（如果是Selector会先通过输入的OptionIndex获取NextPoint）会执行NextPoint的Action并计算Pipe，保存替换内存，如此完成了一次Forward迭代。

1. 当推进到不合法的Point会视为故事结束

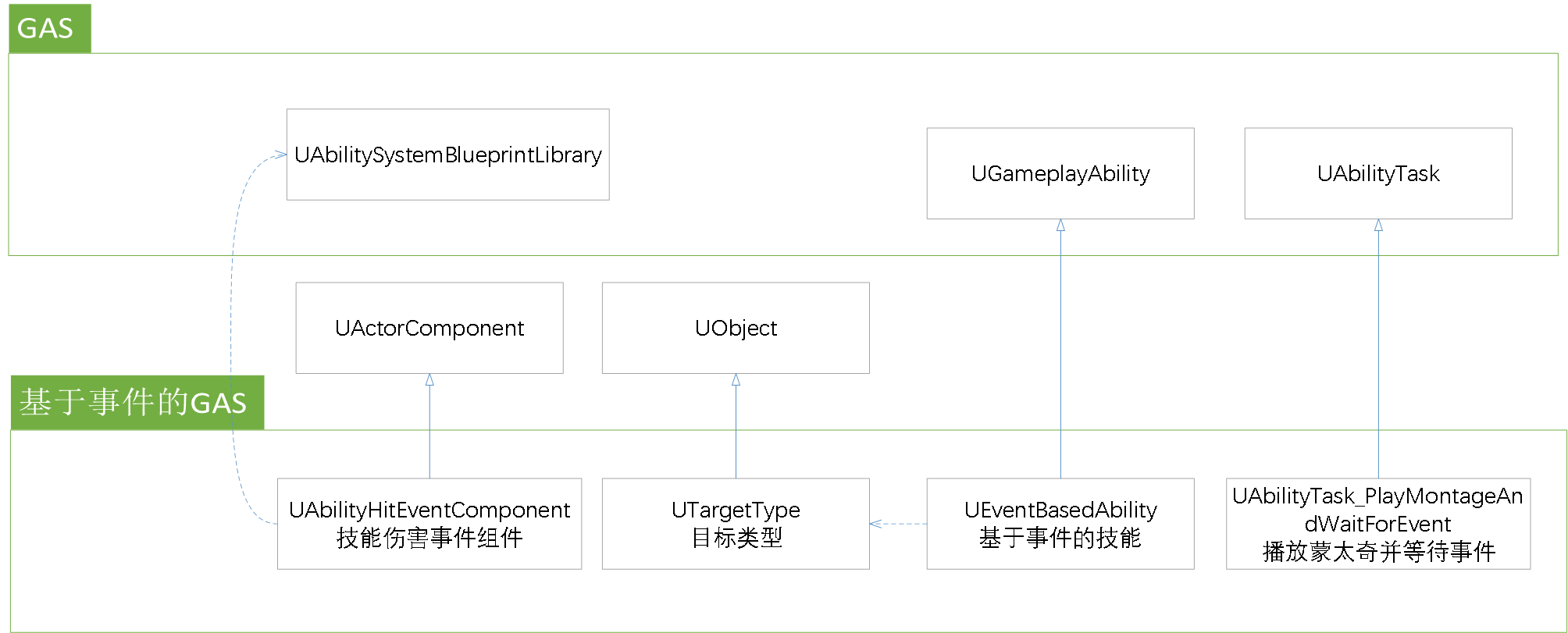
## 基于事件的GAS

Ⅰ设计背景

此模块是对UE的GAS系统进行的拓展，GAS本身依赖了GameplayTag，GameplayEvent，GameplayTask模块。此模块就是在借助这些模块的便利下对GameplayAbility的拓展。

部分设计参考官方项目《ActionRPG》。

Ⅱ类架构图



Ⅲ对外接口与使用说明

①UEventBasedAbility

内部搭载了

Map<Tag, {UTargetType,Array<UGameplayEffect>}>

容器的技能，施加GE时只需要输入事件的Tag即可；使得可以在一个技能中方便地处理多个事件（比如一个技能会包括命中加成与伤害等多个事件）

②UTargetType

通过继承UTargetType并实现GetTarget方法，以决定UEventBasedAbility在获得GE施加对象时的行为

③UAbilityHitEventComponent

主要功能为完成整个伤害的流程，并向Actor发送GameplayEvent

通常为具有伤害范围的Actor创建一个UAbilityHitEventComponent，将代表伤害范围的Primitive与之绑定并交于其管理，在需要时开关重叠检测，若产生碰撞事件，则向适宜的Actor发送GameplayEvent；

可在内部管理伤害的逻辑，比如两Actor之间可否造成伤害，伤害是群伤还是单伤，伤害攻击到Actor后会有多长时间的再伤害冷却；

可与GA的触发器配合。

④UAbilityTask\_PlayMontageAndWaitForEvent

继承自UAbilityTask，可允许Ability播放一个蒙太奇片段，并在播放阶段处理来自ASC的GameplayEvent。

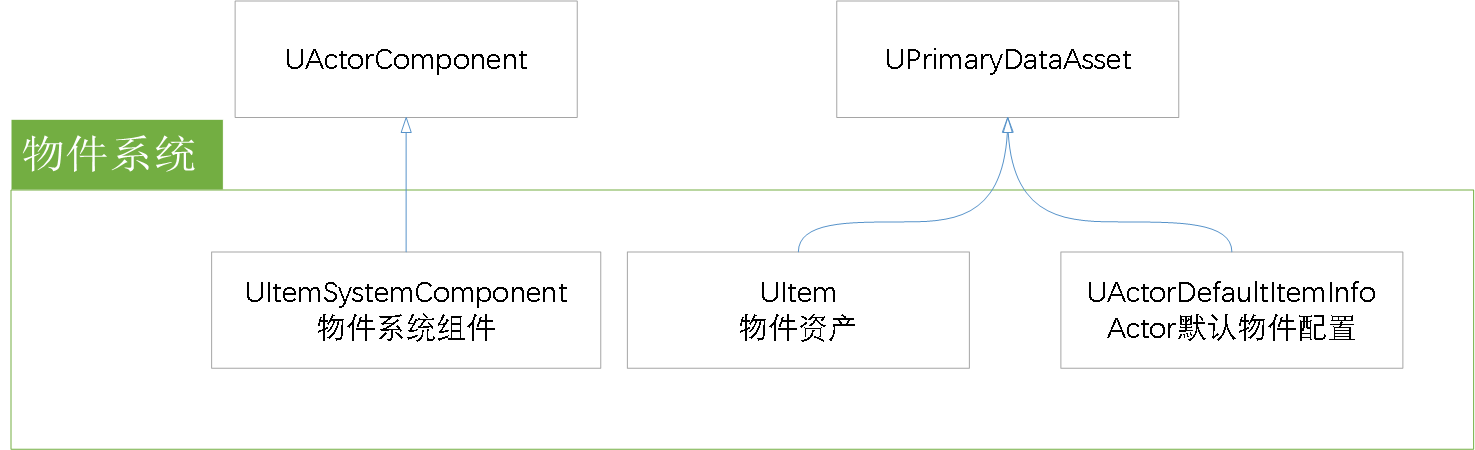
## 物件系统

Ⅰ设计背景

物件系统是几乎所有RPG游戏都拥有的系统。

在Demo设计的物品系统中，核心功能概念主要包括背包系统与装备栏系统；除了在UItemSystemComponent上负责承载相应的数据、行为、事件外，系统还提供了为Class注册默认物件数据与Item资源管理的相关设计方案。

Ⅱ类架构图



Ⅲ对外接口与使用说明

①UItemSystemComponent

为任何逻辑上可以持有物件的Actor创建该组件。

该组件从意义上继承了来自GAS中的Avatar与Owner的概念，在进行Actor通信时，需要Actor继承IItemSystemInterface并通过接口提供组件对象。

②UItem

若需要为自己的游戏创建独特的物件种类（如武器、护具），则需要继承UItem。

UItem本身继承自UPrimaryDataAsset，意味着可以为Item创建DataAsset，并交于AssetManager管理。

③UActorDefaultItemInfo

如果你想让Avatar在初始化时能使用一套默认的物件设置，则需要创建该类的资源，并在蓝图中填写结构

TMap<TSubclassOf<AActor>, FItemInfoDefault>；

FItemInfoDefault中包含静态的装备栏信息与背包信息，内部由FPrimaryAssetId驱使；

此对象也继承自UPrimaryDataAsset，所有能被AssetManager收集到的该类的资产信息都会在UItemSystem（实例子系统）初始化时被其收集起来。

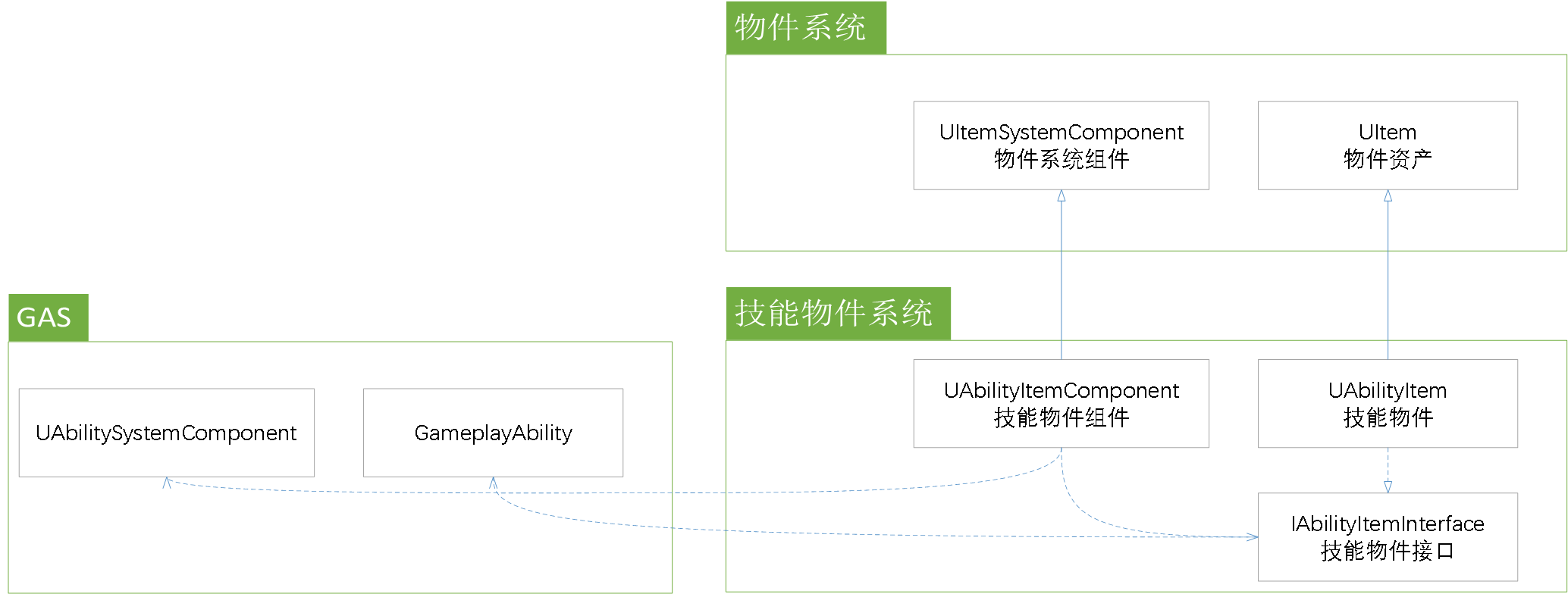
UItemSystemComponent可以为Avatar的类寻找对应的默认配置并加载到内存中。

## 技能物件系统

Ⅰ设计背景

游戏中一般会存在非常多的技能，难以管理。而技能物件系统就是使用物件去管理技能的手段，让物件承载技能，并让技能物件组件管理物件与技能。

Ⅱ类架构图



Ⅲ对外接口与使用说明

①UAbilityItem

该类继承自UItem，并实现了IAbilityItemInterface，系统内部实质通过该接口进行通信。

IAbilityItemInterface需要实现两类逻辑，分别是；

GetAbilitiesWhenHeld

GetAbilitiesWhenEquipped

它们分别对应了Item在Item组件中的两种状态，分别是持有但未装备/持有且装备，接口会返回对应状态下ASC能够使用的技能。

②UAbilityItemComponent

继承自UItemSystemComponent。

通过与ASC进行绑定，该组件会在Item变动下自行为ASC调整其内部的技能。也可以主动进行基于Item的技能调用。

# 四．其他设计与技术参考

此节总结了在Demo中使用的其他设计方案与技术。

## 碰撞预设与碰撞代理

就拿上文中的交互组件来说，这类逻辑在运作中往往存在两种碰撞身份，其一是发出者，其二是接收者。在伤害组件中也会出现伤害者与乘伤者。出于性能考虑，我们不会希望众多发出者之间或众多接收者之间进行碰撞检测，因此需要借助UE的碰撞预设对游戏中出现的碰撞角色设置碰撞通道。

考虑以下逻辑或需求：

1.出于便利或性能，同一碰撞实体往往会负责多种碰撞需求，这意味着碰撞组件会接收来自各种碰撞角色的事件，对于例如交互发起者来说，只希望处理来自交互接受者的事件，所以需要对所有的事件进行碰撞角色的过滤。

2.在战斗中，Actor可能要负责多个碰撞组件的事件，我们需要把这些碰撞组件收集起来统一转发事件。

基于上述需求，可以设计一种碰撞代理组件，来负责上述的功能以及后续类似的功能。

但如果要降低模块的耦合度，我们不希望其他模块对这种特殊的组件产生依赖，则可以让碰撞代理组件继承UPrimitiveComponent，这样碰撞代理组件就具有了UPrimitiveComponent的所有事件与行为，但并不具有碰撞实体。随后，其他模块的接口只需要对准UPrimitiveComponent即可。

## 摄像机系统设计

DEMO中较大程度地实现了视角自定义，对于摄像机的需求是俯视角但是只允许Pitch角可动。

UE的摄像机逻辑是链式的。PlayerController中的PlayerCameraManager将Actor设置为ViewTarget，ViewTarget会使用方法CalcCamera提供摄像机的位置与旋转。对于注册有CameraComponent的Actor，会默认使用CameraComponent的GetCameraView逻辑。这两种方法都非const，意味着我们可以将摄像机更新的逻辑放入其中。

要创建自定义的Camera逻辑，可以创建一个CameraComponent，再利用CameraComponent创建一个ViewTargetActor，让二者负责不同的逻辑。比如在Demo中，ATDSViewTargetActor包含了一个与视线平行的胶囊体碰撞，用于检测在俯视角中把视线遮挡的模型；此外，ATDSViewTargetActor中会使用多个位置，用于实现索敌，限定视角范围（固定在点或限制在一个体积内），混合动画等，因此需要确定好ViewTargetActor、视线中心、Camera等的变换方式。

## 3. 输入系统设计

在C++中，收集输入的方式是通过PlayerController对InputComponent进行出入栈管理。PlayerController也通常被视为需要进行输入管理的对象。

在游戏中，往往会出现多套控制方法，比如一般是Pawn在地图中自由移动，但是与解密物件进行交互后会进入解密界面，进入动画后会脱离一般控制等等。

在UE框架中比较贴合的逻辑应该是，对于每一套控制逻辑，应该由对应的Pawn进行负责，而PlayerController需要负责的则是在这些Pawn中进行控制交换，以及顶层输入的逻辑（如系统菜单）。所以对于成套的输入逻辑，理应是放在Pawn中。

## 4. UE单例模式设计

借助UE的框架，对于一个模块来说，我认为较好的单例可以使用Subsystem与CDO，且可以互相配合。

例如在Demo里，EventBasedAbility中建立GameInstanceSubsystem并完成了UAbilitySystemGlobals::Get().InitGlobalData()方法，相比于《ActionRPG》在AssertManager中初始化我认为是更好的方式（AssertManager相关的大多数功能在项目设置里面基本都能完成，继承成本较高）。 在物件系统模块中也使用了GameInstanceSubsystem来实现Item资产的异步加载

通过CDO可以很方便地收集类数据。在某个版本的物件模块中，曾使用CDO来完成Actor默认物件配置的注册，通过重写PostCDOContruct方法将数据加入类的静态数据段（后来因为数据量会比较大，转为对资产的管理会比较好，不过用AssertManager收集资产会稍微麻烦一点）。使用CDO的好处就是能与蓝图进行直接交互，当我们希望通过蓝图的方式管理C++中的单例，可以使用这种方式。

当我们希望一个类是纯方法类时，可以自由地使用CDO。是使用CDO还是NewObject，需要根据实际环境与设计需求来看，因为CDO不是WorldContextObject。

注意各对象的生命周期，CDO创建是在类型收集的PreInit阶段，而Subsystem的生命周期跟随其Outer，基本都在Init阶段。

## 5. 处理模型遮挡视线

开发中涉及到了因为使用俯视角导致部分模型遮挡的问题，产生了需要让模型淡化的需求。

解决方式是使用材质实例，为模型创建对应材质的材质实例，建立“透明遮蔽”参数（不使用透明度是考虑到性能），并由模型管理参数变化，混合方式。通过使用摄像机系统里建立的检测视线的碰撞体，当视线接触或离开时出发对应的动画。

游戏中使用的插值方式是线性，动画的实际效果一般，可以尝试其他插值方式。

## 6. 限制玩家视野

开发中涉及需要限制玩家视野的需求，这里的解决方案是使用后处理材质与后处理效果，通过修改材质参数集更新视野范围与玩家位置，在后处理材质节点图中对模型顶点进行颜色处理。

# 五．开发总结

此节总结了一些在开发Demo中的一些经验与感悟。

## 开发流程、蓝图与C++

为了提升开发效率，现今成熟的项目开发存在着一定的流程，这与UE的项目结构关系密切。

UE项目的可编程部分主要分为蓝图与C++。使用C++的好处是可迁移性与性能等，使用蓝图的好处是便利与可快速迭代。

项目初期需要对游戏的整体框架进行快速原型开发与游戏性验证，其中存在对逻辑的快速迭代，这个阶段使用蓝图比C++更能提升开发效率。UE的原生蓝图使用节点与连线，进一步提升逻辑可迁移性与可读性，可以使用unLua插件。

在完成了游戏原型的验证后，出于提升性能与逻辑复用率，应该对项目进行重构与整理，将各部分逻辑进行模块化插件化，此时显然使用C++会更有利。

在完成了游戏验证与重构后，剩下的阶段应该是对游戏细节逻辑的补完，这部分的逻辑并没有那么高的使用频率与调用频率，适宜使用蓝图。

## UE插件开发经验

优秀的插件除了能高效复现游戏原型的逻辑外，还应该具有对内对外的兼容性与接口使用的便利性。UE插件开发相关的参考意见，经本次开发经验后总结如下：

①对Gameplay设计

尽量使用原Gameplay框架提供的接口。GAS中若要从Actor中获取ASC，必须要让Actor实现对应的接口；在Demo中涉及到与碰撞组件绑定的相关操作时，都设计成了仅对一个UPrimitiveComponent对象设计接口，然后让碰撞代理去执行UPrimitiveComponent的全部功能。如此便实现了模块之间的解耦。

②对外接口设计

C++插件的对外接口除了服务C++调用者外，其实更应该考虑到蓝图使用者。蓝图的核心接口主要包括：

a. 蓝图类继承

b. 可修改蓝图类的与Level中Actor的属性默认值

c. 蓝图类的可重写与可调用的函数

d. 蓝图类的动态委托

③插件整体设计思路

在Demo开发的过程中，我感觉到对组件继承是一件相当麻烦的工作。为了避免继承组件等类似的较差的接口的诞生，尽量设计更加方便的接口，可以参照GAS的展现出来的设计思路：

a. 优先确定组件的逻辑流程；

b. 对于一个固定流程，会产生相关的属性（类成员）、行为（函数）、事件（委托），我们不需要继承组件也可以使用这些接口；

c. 对于流程中的可编程部分，按照结构将这些部分转化为UObject，我们在蓝图里继承UObject以让其参与组件的流程当中。如GAS的GA，GE。

④兼容性

在设计插件的逻辑流程时，要考虑到能具有一定的兼容性。这个兼容性首先是需要对内的，也就是尽量能满足后续游戏中可能涉及的需求，在这部分中，显然需要在原型开发迭代阶段就与需求端进行完整的需求拓展讨论。

插件开发也应该顾及到对外的兼容性，插件出了游戏能否在其他游戏中也能使用。但在项目开发中，对外的兼容性并非核心需求，对其的考量应该适度。

插件开发的流程设计是需要基于大量实践经验的，需要基于大量实际项目的考验与验证。

⑤生命周期

一般而言，类成员的生命周期由类本身的流程进行管理，但实际情况是，类成员经常会受到类外流程的影响。

在设计流程时，需要考虑到类成员的生命周期，并设置相应的检测方式（比如弱指针）。

⑥网络

插件设计需要考虑到网络相关因素。