更新时间：2021/03/26(完结)
翻译此文时间: 2020.12.21
原文地址: tranek/GASDocumentation
翻译地址: BillEliot/GASDocumentation_Chinese
反馈: **github/PR** or **eliotwjz@gmail.com**

GASDocumentation

我使用一个简单的多人游戏模板项目来阐述对Unreal Engine 4中GameplayAbilitySystem(GAS)插件的理解. 这不是官方文档并且这个项目和我自己都不来自Epic Games. 我不能保证该文档的准确性.

该文档的目的是阐明GAS中的主要概念和相关类, 并结合我的经验提供一些附加说明. 在社区用户中, 已经形成了大量有关GAS的"部落知识", 而我致力于将我了解的全部在这里分享.

样例项目和文档目前基于`Unreal Engine 4.26`. 该文档拥有可用于旧版本Unreal Engine的分支, 但是它们不再受支持, 并且可能存在bug和过时信息.
GASShooter是该样例项目的姐妹项目, 其演示了基于多人FPS/TPS的高阶GAS技术.

最好的文档永远是该插件的代码.

目录

---

1. 步入GameplayAbilitySystem插件

摘自官方文档:

> Gameplay技能系统 是一个高度灵活的框架，可用于构建你可能会在RPG或MOBA游戏中看到的技能和属性类型。你可以构建可供游戏中的角色使用的动作或被动技能，使这些动作导致各种属性累积或损耗的状态效果，实现约束这些动作使用的"冷却"计时器或资源消耗，更改技能等级及每个技能等级的技能效果，激活粒子或音效，等等。简单来说，此系统可帮助你在任何现代RPG或MOBA游戏中设计、实现及高效关联各种游戏中的技能，既包括跳跃等简单技能，也包括你喜欢的角色的复杂技能集。

GameplayAbilitySystem插件由Epic Games开发, 随Unreal Engine 4 (UE4)发布. 它已经由3A商业游戏的严格测试, 例如帕拉贡(Paragon)和堡垒之夜(Fortnite)等等.

该插件对于单人和多人游戏提供了开箱即用的解决方案:

- 执行基于等级的角色能力(Ability)或技能(Skill), 该能力或技能可选花费和冷却时间. (GameplayAbility)
- 管理属于Actor的数值Attribute. (Attribute)
- 为Actor应用状态效果. (GameplayEffect)
- 为Actor应用GameplayTag. (GameplayTag)
- 生成视觉或声音效果. (GameplayCue)
- 为以上提到的所有应用同步(Replication).

在多人游戏中, GAS提供客户端预测(client-side prediction)支持:

- 能力激活.
- 播放蒙太奇.
- 对`Attribute`的修改.
- 应用`GameplayTag`.
- 生成`GameplayCue`.
- 通过连接于`CharacterMovementComponent`的`RootMotionSource`函数形成的移动.

GAS必须由C++创建, 但是`GameplayAbility`和`GameplayEffect`可由设计师在蓝图中创建.

GAS中的现存问题:

- `GameplayEffect`延迟调节(Latency Reconciliation).(不能预测能力冷却时间, 导致高延迟玩家相比低延迟玩家, 对于短冷却时间的能力有更低的激活速率.)
- 不能预测性地移除`GameplayEffect`. 然而我们可以反向预测性地添加`GameplayEffect`, 从而高效的移除它. 但是这不总是合适或者可行的, 因此这仍然是个问题.
- 缺乏样例模板项目, 多人联机样例和文档. 希望这篇文档会有所帮助.

↑ 返回目录

2. 样例项目

该文档包含一个支持多人联机的第三人称射击游戏模板项目, 其目标受众为初识`GameplayAbilitySystem`插件, 但并不是Unreal Engine 4新手. 用户应该了解C++, 蓝图, UMG, Replication和其他UE4的中间件. 该项目提供了一个样例, 其向你展示了如何使用`GameplayAbilitySystem`插件建立一个基础的支持多人联机的第三人称射击游戏, 其中`AbilitySystemComponent(ASC)`分别位于`PlayerState`类代表玩家/AI控制的人物和位于`Character`类代表AI控制的小兵.
我在保证展现GAS基础和带有完整注释的代码所表示的一些普遍技能的同时, 尽力使这个样例足够简单. 由于该文档专注于初学者, 因此该样例不包含像Predicting Projectiles这样的高阶技术.
概念说明:

- ASC位于PlayerState还是Character.
- 网络同步的Attribute.
- 网络同步的蒙太奇(Animation Montages).
- GameplayTag.
- 在GameplayAbility内部和外部应用和移除GameplayEffect.
- 应用被护甲防御后的伤害值来修改角色生命值.
- GameplayEffectExecutionCalculations.
- 眩晕效果.
- 死亡和重生.
- 在服务端上使用能力(Ability)生成抛射物(Projectile).
- 在瞄准和奔跑时, 预测性的修改本地玩家速度.
- 不断消耗耐力来奔跑.
- 消耗魔法值来使用能力(Ability).
- 被动能力(Ability).
- 堆栈GameplayEffect.
- 锁定Actor.
- 在蓝图中创建GameplayAbility.
- 在C++中创建GameplayAbility.
- 实例化每个Actor的GameplayAbility.
- 非实例化的GameplayAbility(Jump).
- 静态GameplayCue(子弹撞击粒子效果).
- Actor GameplayCue(奔跑和眩晕粒子效果).

角色类有如下能力:

能力	输入绑定	是否可预测	C++/Blueprint	描述
跳跃	空格键	Yes	C++	使角色跳跃.
枪	鼠标左键	No	C++	从角色的枪中发射投掷物, 发射动画是可预测的, 但是投掷物不能预测.
瞄准	鼠标右键	Yes	Blueprint	当按住鼠标右键时, 角色会走的更慢并且摄像机会拉近(zoom in)以获得更高的射击精度.
奔跑	左Shift	Yes	Blueprint	当按住左Shift时, 角色在消耗体力的同时跑得更快.
向前猛冲	Q	Yes	Blueprint	角色消耗体力的同时向前猛冲.
被动护盾叠加	被动	No	Blueprint	每过4s角色获得一个最大层数为4的护盾, 每次受到伤害时移除一层护盾.
陨石坠落	R	No	Blueprint	角色锁定一个敌人召唤一个陨石, 对其造成伤害和眩晕效果. 定位是可以预测的而陨石生成是不可预测的.

GameplayAbility无论是由蓝图还是C++创建都没关系. 这里我们使用蓝图和C++混合创建, 意在展示每种方式的使用方法.
AI控制的小兵没有预先定义的GameplayAbility. 红方小兵有较多的生命回复, 蓝方小兵有较多的初始生命.
对于GameplayAbility的命名, 我使用_BP后缀表示由蓝图创建的GameplayAbility逻辑, 没有后缀则表示由C++创建.

蓝图资源命名前缀

Prefix	Asset Type
GA_	GameplayAbility
GC_	GameplayCue
GE_	GameplayEffect

↑ 返回目录

3. 使用GAS创建一个项目

使用GAS建立一个项目的基本步骤:

1. 在编辑器中启用GameplayAbilitySystem插件.
2. 编辑 `YourProjectName.Build.cs`, 添加 `"GameplayAbilities"`, `"GameplayTags"`, `"GameplayTasks"` 到你的 `PrivateDependencyModuleNames`.
3. 刷新/重新生成Visual Studio项目文件.
4. 从4.24开始, 需要强制调用`UAbilitySystemGlobals::InitGlobalData()`来使用`TargetData`, 样例项目在`UEngineSubsystem::Initialize()`中调用该函数. 参阅`InitGlobalData()`获取更多信息.

这就是你启用GAS所需做的全部了. 从这里开始, 添加一个`ASC`和`AttributeSet`到你的`Character`或`PlayerState`, 并开始着手`GameplayAbility`和`GameplayEffect`!

↑ 返回目录

4. GAS概念

4.1 Ability System Component

`AbilitySystemComponent(ASC)` 是GAS的核心, 它是一个处理所有与该系统交互的`UActorComponent`(UAbilitySystemComponent), 所有期望使用GameplayAbility, 包含Attribute, 或者接受GameplayEffect的Actor都必须附加`ASC`. 这些对象都存于`ASC`并由其管理和同步(除了由AttributeSet同步的`Attribute`). 开发者最好但不强求继承该组件.

ASC附加的Actor被引用作为该ASC的OwnerActor, 该ASC的物理代表Actor被称为AvatarActor. OwnerActor和AvatarActor可以是同一个Actor, 比如MOBA游戏中的一个简单AI小兵; 它们也可以是不同的Actor, 比如MOBA游戏中玩家控制的英雄, 其中OwnerActor是PlayerState, AvatarActor是英雄的Character类. 绝大多数Actor的ASC都附加在其自身, 如果你的Actor会重生并且重生时需要持久化Attribute或GameplayEffect(比如MOBA中的英雄), 那么ASC理想的位置就是PlayerState.

Note: 如果ASC位于PlayerState, 那么你需要提高PlayerState的NetUpdateFrequency, 其默认是一个很低的值, 因此在客户端上发生Attribute和GameplayTag改变时会造成延迟或卡顿. 确保启用Adaptive Network Update Frequency, Fortnite就启用了该项.

OwnerActor需要继承并实现IAbilitySystemInterface, 如果AvatarActor和OwnerActor是不同的Actor, 那么AvatarActor也应该继承并实现IAbilitySystemInterface. 该接口有一个必须重写的函数, UAbilitySystemComponent* GetAbilitySystemComponent() const, 其返回一个指向ASC的指针, ASC通过寻找该接口函数来和系统内部进行交互.

ASC在FActiveGameplayEffectContainer ActiveGameplayEffect中保存其当前活跃的GameplayEffect.

ASC在FGameplayAbilitySpecContainer ActivatableAbility中保存其授予的GameplayAbility. 当你想遍历ActivatableAbility.Items时, 确保在循环体之上添加ABILITYLIST_SCOPE_LOCK();来锁定列表以防其改变(比如移除一个Ability). 每个域中的ABILITYLIST_SCOPE_LOCK();会增加AbilityScopeLockCount, 之后出域时会减量. 不要尝试在ABILITYLIST_SCOPE_LOCK();域中移除某个Ability(Ability删除函数会在内部检查AbilityScopeLockCount以防在列表锁定时移除Ability).

⬆ 返回目录

4.1.1 同步模式

ASC定义了三种不同的同步模式用于同步GameplayEffect, GameplayTag和GameplayCue - Full, Mixed和Minimal. Attribute由其AttributeSet同步.

同步模式	何时使用	描述
Full	单人	所有GameplayEffect都同步到客户端.
Mixed	多人, 玩家控制的Actor	GameplayEffect只同步到其所属客户端, 只有GameplayTag和GameplayCue同步到所有客户端.
Minimal	多人, AI控制的Actor	GameplayEffect从不同步到任何客户端, 只有GameplayTag和GameplayCue同步到所有客户端.

Note: Mixed同步模式需要OwnerActor的Owner是Controller. PlayerState的Owner默认是Controller但是Character不是. 如果OwnerActor不是PlayerState时使用Mixed同步模式, 那么需要在OwnerActor中调用SetOwner()设置Controller.

从4.24开始, 需要使用PossessedBy()设置新的Controller为Pawn的Owner.

⬆ 返回目录

4.1.2 设置和初始化

ASC一般在OwnerActor的构建函数中创建并且需要明确标记为Replicated. **这必须在C++中完成.**

```
AGDPlayerState::AGDPlayerState()
{
    // Create Ability system component, and set it to be explicitly replicated
    AbilitySystemComponent = CreateDefaultSubobject<UGDAbilitySystemComponent>
(TEXT("AbilitySystemComponent"));
    AbilitySystemComponent->SetIsReplicated(true);
    //...
}
```

OwnerActor和AvatarActor的ASC在服务端和客户端上均需初始化, 你应该在Pawn的Controller设置之后初始化(Possess之后), 单人游戏只需参考服务端的做法.

对于玩家控制的Character且ASC位于Pawn, 我一般在服务端Pawn的PossessedBy()函数中初始化, 在客户端PlayerController的AcknowledgePossession()函数中初始化.

```
void APACharacterBase::PossessedBy(AController * NewController)
{
    Super::PossessedBy(NewController);

    if (AbilitySystemComponent)
    {
        AbilitySystemComponent->InitAbilityActorInfo(this, this);
    }

    // ASC MixedMode replication requires that the ASC Owner's Owner be the Controller.
    SetOwner(NewController);
}
```

```
void APAPlayerControllerBase::AcknowledgePossession(APawn* P)
{
    Super::AcknowledgePossession(P);

    APACharacterBase* CharacterBase = Cast<APACharacterBase>(P);
    if (CharacterBase)
    {
        CharacterBase->GetAbilitySystemComponent()->InitAbilityActorInfo(CharacterBase,
CharacterBase);
    }

    //...
}
```

对于玩家控制的Character且ASC位于PlayerState, 我一般在服务端Pawn的PossessedBy()函数中初始化, 在客户端PlayerController的OnRep_PlayerState()函数中初始化, 这确保了PlayerState存在于客户端上.

```
void AGDHeroCharacter::PossessedBy(AController * NewController)
{
    Super::PossessedBy(NewController);

    AGDPlayerState* PS = GetPlayerState<AGDPlayerState>();
    if (PS)
    {
        // Set the ASC on the Server. Clients do this in OnRep_PlayerState()
        AbilitySystemComponent = Cast<UGDAbilitySystemComponent>(PS-
>GetAbilitySystemComponent());
```

```
        // AI won't have PlayerControllers so we can init again here just to be sure. No harm in
initing twice for heroes that have PlayerControllers.
        PS->GetAbilitySystemComponent()->InitAbilityActorInfo(PS, this);
    }

    //...
}
```

```
void AGDHeroCharacter::OnRep_PlayerState()
{
    Super::OnRep_PlayerState();

    AGDPlayerState* PS = GetPlayerState<AGDPlayerState>();
    if (PS)
    {
        // Set the ASC for clients. Server does this in PossessedBy.
        AbilitySystemComponent = Cast<UGDAbilitySystemComponent>(PS-
>GetAbilitySystemComponent());

        // Init ASC Actor Info for clients. Server will init its `ASC` when it possesses a new
Actor.
        AbilitySystemComponent->InitAbilityActorInfo(PS, this);
    }

    // ...
}
```

如果你遇到了错误消息`LogAbilitySystem: Warning: Can't activate LocalOnly or LocalPredicted Ability %s when not local!`, 那么就表明ASC没有在客户端中初始化.

⬆ 返回目录

4.2 Gameplay Tags

`FGameplayTag`是由`GameplayTagManager`注册的形似`Parent.Child.Grandchild...`的层级Name, 这些标签对于**分类和描述对象的状态**非常有用, 例如, 如果某个Character处于眩晕状态, 我们可以给一个`State.Debuff.Stun`的`GameplayTag`.

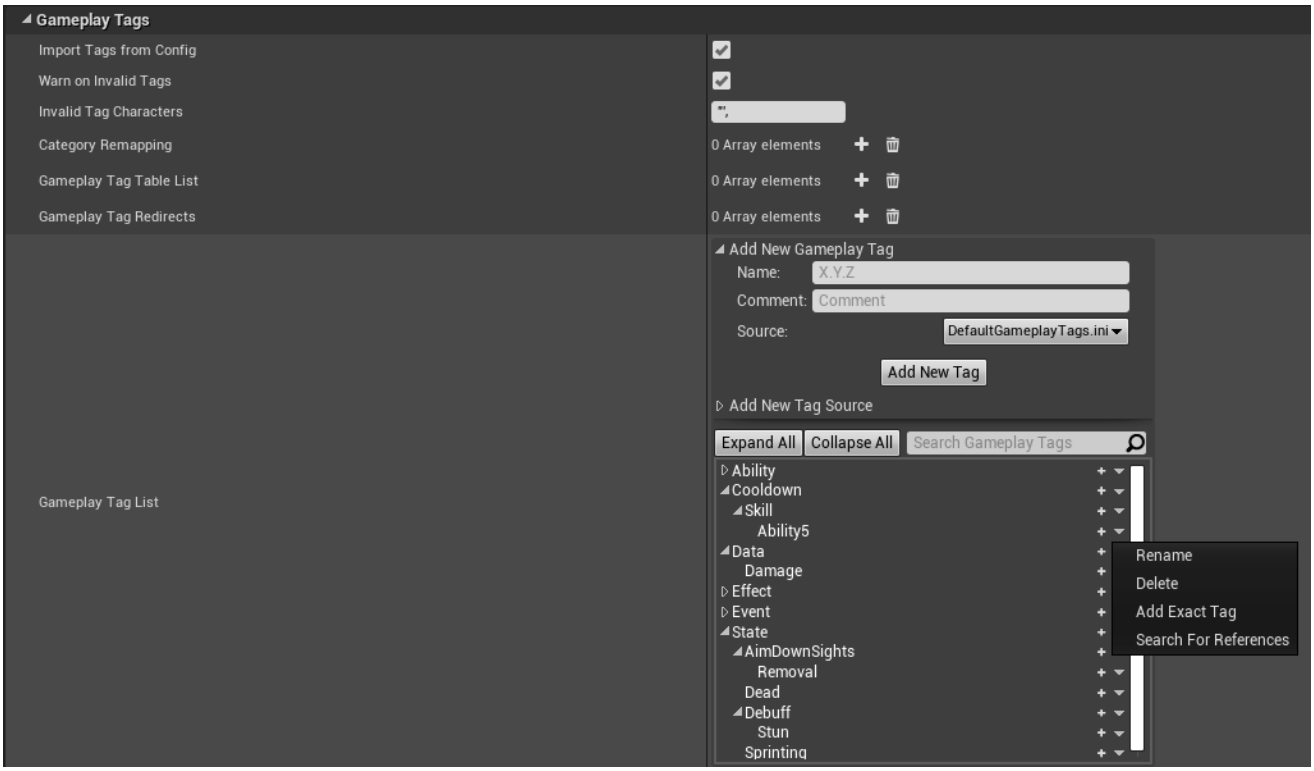
你会发现自己使用`GameplayTag`替换了过去使用布尔值或枚举值来编程, 并且需要对对象有无特定的`GameplayTag`做布尔逻辑判断.

当给某个对象设置标签时, 如果它有ASC的话, 我们一般添加标签到ASC以与其交互. UAbilitySystemComponent执行`IGameplayTagAssetInterface`接口的函数来访问其拥有的`GameplayTag`.

多个`GameplayTag`可被保存于一个`FGameplayTagContainer`中, 相比`TArray<FGameplayTag>`, 最好使用`GameplayTagContainer`, 因为`GameplayTagContainer`做了一些很有效率的优化. 因为标签是标准的`FName`, 所以当在项目设置(Project Setting)中启用`Fast Replication`后, 它们可以高效地打包进`FGameplayTagContainer`以用于同步. `Fast Replication`要求服务端和客户端有相同的`GameplayTag`列表, 这通常不是问题, 因此你应该启用该选项. `GameplayTagContainer`也可以返回`TArray<FGameplayTag>`以用于遍历.

保存于FGameplayTagCountContainer中的GameplayTag有保存该GameplayTag实例数的TagMap. FGameplayTagCountContainer可能存有TagMapCount为0的GameplayTag, 你可能在Debug时遇到这种情况. 任何HasTag()或HasMatchingTag()或其他相似的函数会检查TagMapCount, 如果GameplayTag不存在或者其TagMapCount为0就会返回false.

GameplayTag必须在DefaultGameplayTag.ini中提前定义, UE4编辑器在项目设置中提供了一个界面供开发者管理GameplayTag而无需手动编辑DefaultGameplayTag.ini, 该GameplayTag编辑器可以创建, 重命名, 搜索引用和删除GameplayTag.



搜索GameplayTag会弹出一个类似Reference Viewer的窗口来显示所有引用该GameplayTag的资源, 但这不会显示任何引用该GameplayTag的C++类.

重命名GameplayTag会创建重定向, 因此仍引用原来GameplayTag的资源会重定向到新的GameplayTag. 如果可以的话, 我更倾向于创建新的GameplayTag, 手动更新所有引用到新的GameplayTag, 之后删除旧的GameplayTag以避免创建新的重定向.

除了Fast Replication, GameplayTag编辑器可以选择填充普遍需要同步的GameplayTag以对其深度优化.

如果GameplayTag由GameplayEffect添加, 那么其就是可同步的. ASC允许你添加不可同步的LooseGameplayTag且必须手动管理. 样例项目对State.Dead使用了LooseGameplayTag, 因此当生命值降为0时, 其所属客户端会立即响应. 重生时需要手动将TagMapCount设置回0, 当使用LooseGameplayTag时只能手动调整TagMapCount, 相比纯手动调整TagMapCount, 最好使用UAbilitySystemComponent::AddLooseGameplayTag()和UAbilitySystemComponent::RemoveLooseGameplayTag().

C++中获取GameplayTag引用:

```
FGameplayTag::RequestGameplayTag(FName("Your.GameplayTag.Name"))
```

对于像获取父或子GameplayTag的高阶操作, 请查看GameplayTagManager提供的函数. 为了访问GameplayTagManager, 请引用GameplayTagManager.h并使用UGameplayTagManager::Get().FunctionName调用函数. 相比使用常量字符串进行操作和比较, GameplayTagManager实际上使用关系节点(父, 子等等)保存GameplayTag以获得更快的处理速度.

GameplayTag和GameplayTagContainer有可选UPROPERTY宏Meta = (Categories = "GameplayCue")用于在蓝图中过滤标签而只显示父标签为GameplayCue的GameplayTag, 当你知道GameplayTag或GameplayTagContainer变量应该只用于GameplayCue时, 这将是非常有用的.

作为选择, 有一单独的FGameplayCueTag结构体可以包裹FGameplayTag并且可以在蓝图中自动过滤GameplayTag而只显示父标签为GameplayCue的标签.

如果你想过滤函数中的GameplayTag参数, 使用UFUNCTION宏Meta = (GameplayTagFilter = "GameplayCue"). GameplayTagContainer参数不能过滤, 如果你想编辑引擎来允许过滤GameplayTagContainer参数, 查看SGameplayTagGraphPin::ParseDefaultValueData()是如何从Engine\Plugins\Editor\GameplayTagEditor\Source\GameplayTagEditor\Private\SGameplayTagGraphPin.cpp中调用FilterString = UGameplayTagManager::Get().GetCategoriesMetaFromField(PinStructType);的, 还有是如何在SGameplayTagGraphPin::GetListContent()中将FilterString传递给SGameplayTagWidget的, Engine\Plugins\Editor\GameplayTagEditor\Source\GameplayTagEditor\Private\SGameplayTagContainerGraphPin.cpp中这些函数的GameplayTagContainer版本并没有检查Meta域属性和传递过滤器.

样例项目广泛地使用了GameplayTag.

⬆ 返回目录

4.2.1 响应Gameplay Tags的变化

ASC提供了一个委托(Delegate)用于在GameplayTag添加或移除时触发, 其中EGameplayTagEventType参数可以明确是该GameplayTag添加/移除还是其TagMapCount发生变化时触发.

```
AbilitySystemComponent-
>RegisterGameplayTagEvent(FGameplayTag::RequestGameplayTag(FName("State.Debuff.Stun")),
EGameplayTagEventType::NewOrRemoved).AddUObject(this, &AGDPlayerState::StunTagChanged);
```

回调函数拥有变化的GameplayTag参数和新的TagCount参数.

```
virtual void StunTagChanged(const FGameplayTag CallbackTag, int32 NewCount);
```

⬆ 返回目录

4.3 Attribute

4.3.1 Attribute定义

Attribute是由FGameplayAttributeData结构体定义的浮点值, 其可以表示从**角色生命值到角色等级再到一瓶药水的剂量的任何事物, 如果某项数值是属于某个Actor且游戏相关的,** 你就应该考虑使用Attribute. Attribute一般应该只能由GameplayEffect修改, 这样ASC才能预测(Predict)其改变.

Attribute也可以由AttributeSet定义并存于其中. AttributeSet用于同步那些标记为replication的Attribute. 参阅AttributeSet部分来了解如何定义Attribute.

Tip: 如果你不想某个Attribute显示在编辑器的Attribute列表, 可以使用Meta = (HideInDetailsView)属性宏.

⬆ 返回目录

4.3.2 BaseValue vs. CurrentValue

一个Attribute是由两个值 —— 一个BaseValue和一个CurrentValue组成的, BaseValue是Attribute的永久值而CurrentValue是BaseValue加上GameplayEffect给的临时修改值后得到的. 例如, 你的Character可能有一个BaseValue为600u/s的移动速度Attribute, 因为还没有GameplayEffect修改移动速度, 所以CurrentValue也是600u/s, 如果Character获得了一个临时50u/s的移动速度加成, 那么BaseValue仍然是600u/s而CurrentValue是600+50=650u/s, 当该

移动速度加成消失后, CurrentValue就会变回BaseValue的600u/s.

初识GAS的新手经常将BaseValue误认为Attribute的最大值并以这样的认识去编程, 这是错误的, 可以修改或引用的Ability/UI中的Attribute最大值应该是另外单独的Attribute. 对于硬编码的最大值和最小值, 有一种方法是使用可以设置最大值和最小值的FAttributeMetaData定义一个DataTable, 但是Epic在该结构体上的注释称之为"work in progress", 详见AttributeSet.h. 为了避免这种疑惑, 我建议引用在Ability或UI中的最大值应该单独定义Attribute, 只用于限制(Clamp)Attribute大小的硬编码最大值和最小值应该在AttributeSet中定义为硬编码浮点值. 关于限制(Clamp)Attribute值的问题在PreAttributeChange()中讨论了CurrentValue的修改, 在PostGameplayEffectExecute()中讨论了GameplayEffect对BaseValue的修改.

即刻(Instant)GameplayEffect可以永久性的修改BaseValue, 而持续(Duration)和无限(Infinite)GameplayEffect可以修改CurrentValue. 周期性(Periodic)GameplayEffect被视为即刻(Instant)GameplayEffect并且可以修改BaseValue.

⬆ 返回目录

4.3.3 元(Meta)Attribute

一些Attribute被视为占位符, 其是用于预计和Attribute交互的临时值, 这些Attribute被叫做Meta Attribute. 例如, 我们通常定义伤害值为Meta Attribute, 使用伤害值Meta Attribute作为占位符, 而不是使用GameplayEffect直接修改生命值Attribute, 使用这种方法, 伤害值就可以在GameplayEffectExecutionCalculation中由buff和debuff修改, 并且可以在AttributeSet中进一步操作, 例如, 在最终将生命值减去伤害值之前, 要将伤害值减去当前的护盾值. 伤害值Meta Attribute在GameplayEffect之间不是持久化的, 并且可以被任何一方重写. Meta Attribute一般是不可同步的.

Meta Attribute对于在"我们应该造成多少伤害?"和"我们该如何处理伤害值?"这种问题之中的伤害值和治疗值做了很好的解构, 这种解构意味着GameplayEffect和ExecutionCalculation无需了解目标是如何处理伤害值的. 继续看伤害值的例子, GameplayEffect确定造成多少伤害, 之后AttributeSet决定如何使用该伤害值, 不是所有的Character都有相同的Attribute, 特别是使用了AttributeSet子类的话, AttributeSet基类可能只有一个生命值Attribute, 但是它的子类可能增加了一个护盾值Attribute, 拥有护盾值Attribute的子类AttributeSet可能会以不同于AttributeSet基类的方式分配收到的伤害.

尽管Meta Attribute是一个很好的设计模式, 但其并不是强制使用的. 如果你只有一个用于所有伤害实例的Execution Calculation和一个所有Character共用的AttributeSet类, 那么你就可以在Exeuction Calculation中分配伤害到生命, 护盾等等, 并直接修改那些Attribute, 这种方式你只会丢失灵活性, 但总体上并无大碍.

⬆ 返回目录

4.3.4 响应Attribute变化

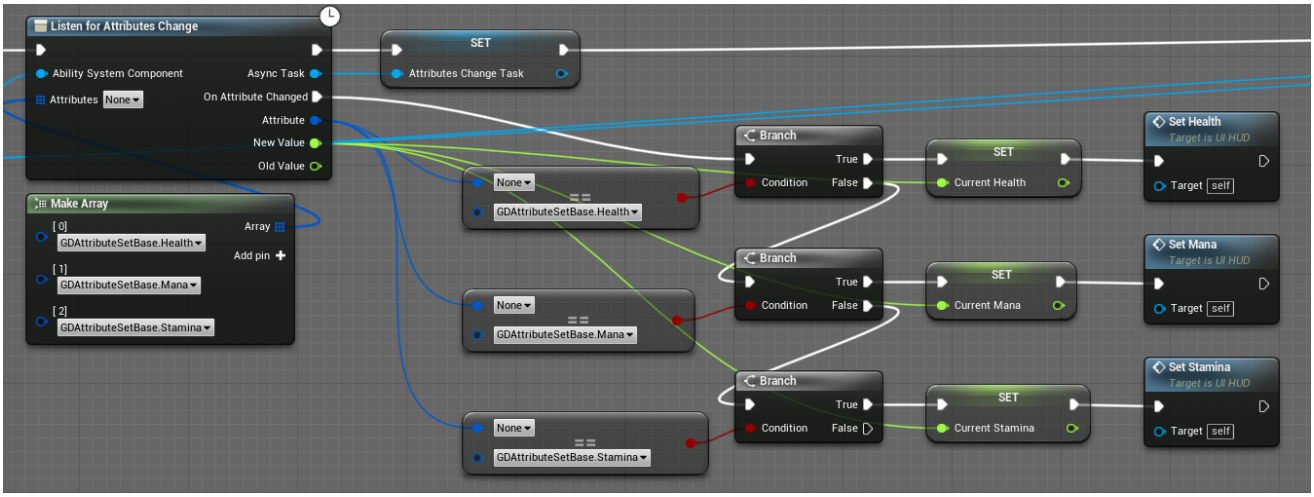
为了监听Attribute何时变化以便更新UI和其他游戏逻辑, 可以使用UAbilitySystemComponent::GetGameplayAttributeValueChangeDelegate(FGameplayAttributeAttribute), 该函数返回一个委托(Delegate), 你可以将其绑定一个当Attribute变化时需要自动调用的函数. 该委托提供一个FOnAttributeChangeData参数, 其中有NewValue, OldValue和FGameplayEffectModCallbackData. **Note**: FGameplayEffectModCallbackData只能在服务端上设置.

```
AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(AttributeSetBase-
>GetHealthAttribute()).AddUObject(this, &AGDPlayerState::HealthChanged);

virtual void HealthChanged(const FOnAttributeChangeData& Data);
```

样例项目将其绑定到了GDPlayerState用于更新HUD, 当生命值下降为0时, 也可以响应玩家死亡.

样例项目中有一个将上述逻辑包裹进ASyncTask的自定义蓝图节点, 其在UI_HUD(UMG Widget)中用于更新生命值, 魔法值和耐力值. 该AsyncTask会一直响应直到手动调用EndTask(), 就像在UMG Widget的Destruct事件中调用那样. 参阅AsyncTaskAttributeChanged.h/cpp.



⬆ 返回目录

4.3.5 自动推导Attribute

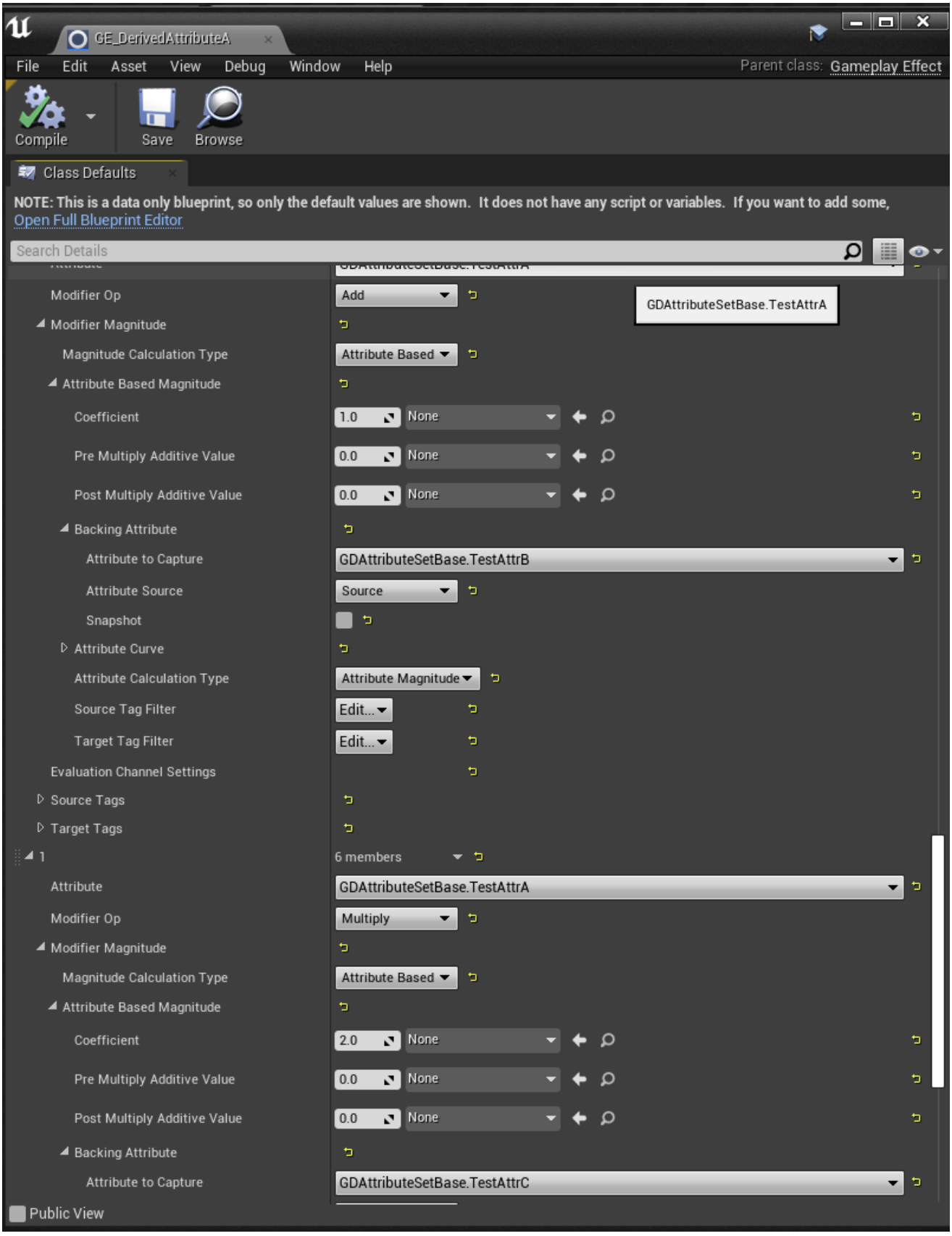
为了使一个`Attribute`的部分或全部值继承自一个或更多`Attribute`, 可以使用基于一个或多个`Attribute`或MMC Modifiers的无限(`Infinite`)`GameplayEffect`. 当自动推导`Attribute`依赖的某个`Attribute`更新时它也会自动更新.

在自动推导`Attribute`上的所有`Modifier`形成的最终公式和`Modifier Aggregators`的公式是一样的. 如果你需要计算式要按一定的顺序进行, 在`MMC`中做就是了.

```
((CurrentValue + Additive) * Multiplicitive) / Division
```

Note: 如果在PIE中打开多个窗口, 你需要在编辑器首选项中禁用`Run Under One Process`, 否则当自动推导`Attribute`所依赖的`Attribute`更新时, 除了第一个窗口外其不会更新.

在这个例子中, 我们有一个无限(`Infinite`)`GameplayEffect`, 其从TestAttrB和TestAttrC `Attribute`以`TestAttrA = (TestAttrA + TestAttrB) * ( 2 * TestAttrC)`公式继承得到TestAttrA, 每次TestAttrB和TestAttrC更新时, TestAttrA都会自动重新计算.


GE_DerivedAttributeA
File Edit Asset View Debug Window Help
Parent class: Gameplay Effect
Compile
Save
Browse
Class Defaults
NOTE: This is a data only blueprint, so only the default values are shown. It does not have any script or variables. If you want to add some,
Open Full Blueprint Editor
Search Details
Modifier Op
Add
GDAttributeSetBase.TestAttrA
Modifier Magnitude
Magnitude Calculation Type
Attribute Based
Attribute Based Magnitude
Coefficient
1.0
None
Pre Multiply Additive Value
0.0
None
Post Multiply Additive Value
0.0
None
Backing Attribute
Attribute to Capture
GDAttributeSetBase.TestAttrB
Attribute Source
Source
Snapshot
Attribute Curve
Attribute Calculation Type
Attribute Magnitude
Source Tag Filter
Edit...
Target Tag Filter
Edit...
Evaluation Channel Settings
Source Tags
Target Tags
1
6 members
Attribute
GDAttributeSetBase.TestAttrA
Modifier Op
Multiply
Modifier Magnitude
Magnitude Calculation Type
Attribute Based
Attribute Based Magnitude
Coefficient
2.0
None
Pre Multiply Additive Value
0.0
None
Post Multiply Additive Value
0.0
None
Backing Attribute
Attribute to Capture
GDAttributeSetBase.TestAttrC
Public View

↑ 返回目录

4.4 AttributeSet

4.4.1 定义AttributeSet

`AttributeSet`**用于定义, 保存以及管理对**`Attribute`**的修改.** 开发者应该继承UAttributeSet. 在OwnerActor的构造函数中创建`AttributeSet`会自动注册到其`ASC`. **这必须在C++中完成.**

⬆ 返回目录

4.4.2 设计AttributeSet

一个`ASC`可能有一个或多个`AttributeSet`, `AttributeSet`消耗的内存微不足道, 使用多少`AttributeSet`是留给开发人员决定的.

有种方案是设置一个单一且巨大的`AttributeSet`, 共享于游戏中的所有Actor, 并且只使用需要的`Attribute`, 忽略不用的`Attribute`.

作为选择, 你可以使用多个`AttributeSet`来表示按需添加到Actor的`Attribute`分组, 例如, 你可以有一个生命相关的`AttributeSet`, 一个魔法相关的`AttributeSet`等等. 在MOBA游戏中, 英雄可能需要魔法, 但是小兵并不需要, 因此英雄就需要魔法`AttributeSet`而小兵则不需要.

另外, 继承`AttributeSet`的另一种意义是可以选择一个Actor可以有哪些`Attribute`. `Attribute`在内部被引用为`AttributeSetClassName.AttributeName`, 当你继承`AttributeSet`时, 所有父类的`Attribute`仍将保留父类名作为前缀.

尽管可以拥有多个`AttributeSet`, 但是不应该在同一`ASC`中拥有多个同一类的`AttributeSet`, 如果在同一`ASC`中有多个同一类的`AttributeSet`, `ASC`就不知道该使用哪个`AttributeSet`而随机选择一个.

⬆ 返回目录

4.4.2.1 使用单独Attribute的子组件

假设你在某个Pawn上有多个可被损害的组件, 像可被独立损害的护甲片, 如果可以明确可被损害组件的最大数量, 我建议把多个生命值`Attribute`放到一个`AttributeSet`中 —— DamageableCompHealth0, DamageableCompHealth1, 等等, 以表示这些可被损害组件在逻辑上的"slot", 在可被损害组件的类实例中, 指定可以被`GameplayAbility`和Execution读取的带slot编号的`Attribute`来表明该应用伤害值到哪个`Attribute`. 如果Pawn当前拥有0个或少于最大数量的可损害组件也无妨, 因为`AttributeSet`拥有一个`Attribute`, 并不意味着必须要使用它, 未使用的`Attribute`只占用很少的内存.

如果每个子组件都需要很多`Attribute`且子组件的数量可以是无限的, 或者子组件可以分离被其他玩家使用(比如武器), 或者出于其他原因上述方法不适用于你, 那么我建议就不要使用`Attribute`, 而是在组件中保存普通的浮点数. 参阅Item Attribute.

⬆ 返回目录

4.4.2.2 运行时添加和移除AttributeSet

`AttributeSet`可以在运行时从`ASC`上添加和移除, 然而移除`AttributeSet`是很危险的, 例如, 如果某个`AttributeSet`在客户端上移除早于服务端, 而某个`Attribute`的变化又同步到了客户端, 那么`Attribute`就会因为找不到`AttributeSet`而使游戏崩溃.

武器添加到Inventory:

```
AbilitySystemComponent->SpawnedAttribute.AddUnique(WeaponAttributeSetPointer);
AbilitySystemComponent->ForceReplication();
```

武器从Inventory移除:

```
AbilitySystemComponent->SpawnedAttribute.Remove(WeaponAttributeSetPointer);
AbilitySystemComponent->ForceReplication();
```

⬆ 返回目录

4.4.2.3 Item Attribute(武器弹药)

有几种方法可以实现带有Attribute(武器弹药, 盔甲耐久等等)的可装备物品, 所有这些方法都直接在物品中存储数据, 这对于在生命周期中可以被多个玩家装备的物品来说是必须的.

1. 在物品中使用普通的浮点数(推荐).
2. 在物品中使用单独的AttributeSet.
3. 在物品中使用单独的ASC.

⬆ 返回目录

4.4.2.3.1 在物品中使用普通浮点数

在物品类实例中存储普通浮点数而不是Attribute, Fortnite和GASShooter就是这样处理枪械子弹的, 对于枪械, 在其实例中存储可同步的浮点数(COND_OwnerOnly), 比如最大弹匣量, 当前弹匣中弹药量, 剩余弹药量等等, 如果枪械需要共享剩余弹药量, 那么就将剩余弹药量移到Character中共享的弹药AttributeSet里作为一个Attribute(换弹Ability可以使用一个Cost GE从剩余弹药量中填充枪械的弹匣弹药量浮点). 因为没有为当前弹匣弹药量使用Attribute, 所以需要重写UGameplayAbility中的一些函数来检查和应用枪械中浮点数的花销(cost). 当授予Ability时将枪械在GameplayAbilitySpec中转换为SourceObject, 这意味着可以在Ability中访问授予Ability的枪械.

为了防止在全自动射击过程中枪械会反向同步弹药量并扰乱本地弹药量(译者注: 通俗解释就是因为存在同步延迟且在连续射击这一高同步过程中, 所以客户端的弹药量会来不及和服务端同步, 造成弹药量减少后又突然变多的现象.), 如果玩家拥有IsFiring的GameplayTag, 就在PreReplication()中禁用同步, 本质上是要在其中做自己的本地预测.

```
void AGSWeapon::PreReplication(IRepChangedPropertyTracker& ChangedPropertyTracker)
{
    Super::PreReplication(ChangedPropertyTracker);

    DOREPLIFETIME_ACTIVE_OVERRIDE(AGSWeapon, PrimaryClipAmmo, (IsValid(AbilitySystemComponent)
&& !AbilitySystemComponent->HasMatchingGameplayTag(WeaponIsFiringTag)));
    DOREPLIFETIME_ACTIVE_OVERRIDE(AGSWeapon, SecondaryClipAmmo, (IsValid(AbilitySystemComponent)
&& !AbilitySystemComponent->HasMatchingGameplayTag(WeaponIsFiringTag)));
}
```

好处:

1. 避免了使用AttributeSet的局限(见下).

局限:

1. 不能使用现有的GameplayEffect工作流(弹药使用的Cost GEs等等).
2. 要求重写UGameplayAbility中的关键函数来检查和应用枪械中浮点数的花销(Cost).

⬆ 返回目录

4.4.2.3.2 在物品中使用AttributeSet

在物品中使用单独的AttributeSet可以实现将其添加到玩家的Inventory, 但还是有一定的局限性. 较早版本的GASShooter中的武器弹药是使用的这种方法, 武器类在其自身存储诸如最大弹匣量, 当前弹匣弹药量, 剩余弹药量等等到一个AttributeSet, 如果枪械需要共享剩余弹药量, 那么就将剩余弹药量移到Character中共享的弹药AttributeSet里. 当服务端上某个武器添加到玩家的Inventory后, 该武器会将它的AttributeSet添加到玩家的

ASC::SpawnedAttribute, 之后服务端会将其同步下发到客户端, 如果该武器从Inventory中移除, 它也会将其AttributeSet从ASC::SpawnedAttribute中移除.

当AttributeSet存于除了OwnerActor之外的对象上时(对于某个武器来说), 会得到一些关于AttributeSet的编译错误, 解决办法是在BeginPlay()中构建AttributeSet而不是在构造函数中, 并在武器类中实现IAbilitySystemInterface(当你添加武器到玩家Inventory时设置ASC的指针).

```
void AGSWeapon::BeginPlay()
{
    if (!AttributeSet)
    {
        AttributeSet = NewObject<UGSWeaponAttributeSet>(this);
    }
    //...
}
```

你可以查看较早版本的GASShooter来实际地体会这种方案.

好处:

1. 可以使用已有的GameplayAbility和GameplayEffect工作流(弹药使用的Cost GEs等等).
2. 对于很小的物品集可以快速设置

局限:

1. 必须为每个武器类型创建新的AttributeSet类, ASC实际上只能有一个该类的AttributeSet实例, 因为对Attribute的修改会在ASC的SpawnedAttribute数组中寻找其第一个AttributeSet类实例, 其他相同的AttributeSet类实例则会被忽略.
2. 和第1条同样的原因(每个AttributeSet类一个AttributeSet实例), 在玩家的Inventory中每种武器类型只能有一个.
3. 移除AttributeSet是很危险的. 在GASShooter中, 如果玩家因为火箭弹而自杀, 玩家会立即从其Inventory中移除火箭弹发射器(包括其在ASC中的AttributeSet), 当服务端同步火箭弹发射器的弹药Attribute改变时, 由于AttributeSet在客户端ASC上不复存在而使游戏崩溃.

⬆ 返回目录

4.4.2.3.3 在物品中使用单独的ASC

在每个物品上都创建一个AbilitySystemComponent是种很极端的方案. 我还没有亲自做过这种方案, 在其他地方也没见过. 这种方案应该会花费相当的开发成本才能正常使用.

> Is it viable to have several AbilitySystemComponents which have the same owner but different avatars (e.g. on pawn and weapon/items/projectiles with Owner set to PlayerState)?

> The first problem I see there would be implementing the IGameplayTagAssetInterface and IAbilitySystemInterface on the owning Actor. The former may be possible: just aggregate the tags from all all ASCs (but watch out - HasAlMatchingGameplayTag may be met only via cross ASC aggregation. It wouldn't be enough to just forward that calls to each ASC and OR the results together). But the later is even trickier: which ASC is the authoritative one? If someone wants to apply a GE -which one should receive it? Maybe you can work these out but this side of the problem will be the hardest: owners will multiple ASCs beneath them.

> Separate ASCs on the pawn and the weapon can make sense on its own though. E.g, distinguishing between tags the describe the weapon vs those that describe the owning pawn. Maybe it does make sense that tags granted to the weapon also “apply” to the owner and nothing else (E.g, Attribute and GEs are independent but the owner will aggregate the owned tags like I describe above). This could work out, I am sure. But having multiple ASCs with the same owner may get dicey.

community questions #6中来自Epic的Dave Ratti的回答.

好处:

1. 可以使用已有的GameplayAbility和GameplayEffect工作流(弹药使用的Cost GEs等等).
2. 可以复用AttributeSet类(每个武器的ASC中各一个).

局限:

1. 未知的开发成本.
2. 甚至方案可行么?

⬆ 返回目录

4.4.3 定义Attribute

Attribute只能使用C++在AttributeSet头文件中定义. 建议把下面这个宏块加到每个AttributeSet头文件的顶部, 其会自动为每个Attribute生成getter和setter函数.

```
// Uses macros from AttributeSet.h
#define ATTRIBUTE_ACCESSORS(ClassName, PropertyName) \
    GAMEPLAYATTRIBUTE_PROPERTY_GETTER(ClassName, PropertyName) \
    GAMEPLAYATTRIBUTE_VALUE_GETTER(PropertyName) \
    GAMEPLAYATTRIBUTE_VALUE_SETTER(PropertyName) \
    GAMEPLAYATTRIBUTE_VALUE_INITTER(PropertyName)
```

一个可同步的生命值Attribute可能像下面这样定义:

```
UPROPERTY(BlueprintReadOnly, Category = "Health", ReplicatedUsing = OnRep_Health)
FGameplayAttributeData Health;
ATTRIBUTE_ACCESSORS(UGDAttributeSetBase, Health)
```

同样在头文件中定义OnRep函数:

```
UFUNCTION()
virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);
```

AttributeSet的.cpp文件应该用预测系统(Prediction System)使用的GAMEPLAYATTRIBUTE_REPNOTIFY宏填充OnRep函数:

```
void UGDAttributeSetBase::OnRep_Health(const FGameplayAttributeData& OldHealth)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UGDAttributeSetBase, Health, OldHealth);
}
```

最后, Attribute需要添加到GetLifetimeReplicatedProps:

```
void UGDAttributeSetBase::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME_CONDITION_NOTIFY(UGDAttributeSetBase, Health, COND_None, REPNOTIFY_Always);
}
```

REPTNOTIFY_Always用于设置OnRep函数在客户端值已经与服务端同步的值相同的情况下触发(因为有预测), 默认设置下, 客户端值与服务端同步的值相同时, OnRep函数是不会触发的.

如果Attribute无需像Meta Attribute那样同步, 那么OnRep和GetLifetimeReplicatedProps步骤可以跳过.

⬆返回目录

4.4.4 初始化Attribute

有多种方法可以初始化Attribute(将BaseValue和CurrentValue设置为某初始值). Epic建议使用即刻(Instant)GameplayEffect, 这也是样例项目使用的方法.

查看样例项目的GE_HeroAttribute蓝图来了解如何创建即刻(Instant)GameplayEffect以初始化Attribute, 该GameplayEffect应用是写在C++中的.

如果在定义Attribute时使用了ATTRIBUTE_ACCESSORS宏, 那么在AttributeSet中会自动为每个Attribute生成一个初始化函数.

```
// InitHealth(float InitialValue) is an automatically generated function for an Attribute
'Health' defined with the `ATTRIBUTE_ACCESSORS` macro
AttributeSet->InitHealth(100.0f);
```

查看AttributeSet.h获取更多初始化Attribute的方法.

Note: 4.24之前, FAttributeSetInitterDiscreteLevels不能和FGameplayAttributeData协同使用, 它在Attribute是原生浮点数时创建, 并且会和FGameplayAttributeData不是Plain Old Data(POD)时冲突. 该问题在4.24中修复(https://issues.unrealengine.com/issue/UE-76557.).

⬆返回目录

4.4.5 PreAttributeChange()

PreAttributeChange(const FGameplayAttribute& Attribute, float& NewValue)是AttributeSet中的主要函数之一, 其在修改发生前响应Attribute的CurrentValue变化, 其是通过引用参数NewValue限制(Clamp)CurrentValue即将进行的修改的理想位置.

例如像样例项目那样限制移动速度Modifier:

```
if (Attribute == GetMoveSpeedAttribute())
{
    // Cannot slow less than 150 units/s and cannot boost more than 1000 units/s
    NewValue = FMath::Clamp<float>(NewValue, 150, 1000);
}
```

GetMoveSpeedAttribute()函数是由我们在AttributeSet.h中添加的宏块创建的(定义Attribute).

PreAttributeChange()可以被Attribute的任何修改触发, 无论是使用Attribute的setter(由AttributeSet.h中的宏块定义(定义Attribute))还是使用GameplayEffect.

Note: 在这里做的任何限制都不会永久性地修改ASC中的Modifier, 只会修改查询Modifier的返回值, 这意味着像GameplayEffectExecutionCalculations和ModifierMagnitudeCalculations这种自所有Modifier重新计算CurrentValue的函数需要再次执行限制(Clamp)操作.

Note: Epic对于PreAttributeChange()的注释说明不要将该函数用于游戏逻辑事件, 而主要在其中做限制操作. 对于修改Attribute的游戏逻辑事件的建议位置是UAbilitySystemComponent::GetGameplayAttributeValueChangeDelegate(FGameplayAttribute Attribute)(响应Attribute变化).

⬆返回目录

4.4.6 PostGameplayEffectExecute()

PostGameplayEffectExecute(const FGameplayEffectModCallbackData & Data)仅在即刻(Instant)GameplayEffect对Attribute的BaseValue修改之后触发, 当GameplayEffect对其修改时, 这就是一个处理更多Attribute操作的有效位置.

例如, 在样例项目中, 我们在这里从生命值Attribute中减去了最终的伤害值Meta Attribute, 如果有护盾值Attribute的话, 我们也会在减除生命值之前从护盾值中减除伤害值. 样例项目也在这里应用了被击打反应动画, 显示浮动的伤害数值和为击杀者分配经验值和赏金. 通过设计, 伤害值Meta Attribute总是会传递给即刻(Instant)GameplayEffect而不是Attribute Setter.

其他只会由即刻(Instant)GameplayEffect修改BaseValue的Attribute, 像魔法值和耐力值, 也可以在这里被限制为其相应的最大值Attribute.

Note: 当PostGameplayEffectExecute()被调用时, 对Attribute的修改已经发生, 但是还没有被同步回客户端, 因此在这里限制值不会造成对客户端的二次同步, 客户端只会接收到限制后的值.

⬆ 返回目录

4.4.7 OnAttributeAggregatorCreated()

OnAttributeAggregatorCreated(const FGameplayAttribute& Attribute, FAggregator* NewAggregator)会在Aggregator为集合中的某个Attribute创建时触发, 它允许FAggregatorEvaluateMetaData的自定义设置, AggregatorEvaluateMetaData是Aggregator基于所有应用的Modifier(Modifier)评估Attribute的CurrentValue的. 默认情况下, AggregatorEvaluateMetaData只由Aggregator用于确定哪些Modifier是满足条件的, 以MostNegativeMod_AllPositiveMods为例, 其允许所有正(Positive)Modifier但是限制负(Negative)Modifier(仅最负的那一个), 这在Paragon中只允许将最负移动速度减速效果应用到玩家, 而不用管应用所有正移动速度buff时有多少负移动效果. 不满足条件的Modifier仍存于ASC中, 只是不被总合进最终的CurrentValue, 一旦条件改变, 它们之后就可能满足条件, 就像如果最负Modifier过期后, 下一个最负Modifier(如果存在的话)就是满足条件的.

为了在只允许最负Modifier和所有正Modifier的例子中使用AggregatorEvaluateMetaData:

```
virtual void OnAttributeAggregatorCreated(const FGameplayAttribute& Attribute, FAggregator*
NewAggregator) const override;

void UGSAttributeSetBase::OnAttributeAggregatorCreated(const FGameplayAttribute& Attribute,
FAggregator* NewAggregator) const
{
    Super::OnAttributeAggregatorCreated(Attribute, NewAggregator);

    if (!NewAggregator)
    {
        return;
    }

    if (Attribute == GetMoveSpeedAttribute())
    {
        NewAggregator->EvaluationMetaData =
&FAggregatorEvaluateMetaDataLibrary::MostNegativeMod_AllPositiveMods;
    }
}
```

你的自定义AggregatorEvaluateMetaData应该作为静态变量添加到FAggregatorEvaluateMetaDataLibrary.

⬆ 返回目录

4.5 Gameplay Effects

4.5.1 定义GameplayEffect

GameplayEffect(GE)是**Ability修改其自身和其他Attribute和GameplayTag的容器**, 其可以立即修改Attribute(像伤害或治疗)或应用长期的状态buff/debuff(像移动速度加速或眩晕). UGameplayEffect只是一个定义单一游戏效果的数据类, 不应该在其中添加额外的逻辑. 设计师一般会创建很多UGameplayEffect的子类蓝图.

GameplayEffect通过Modifier和Execution(GameplayEffectExecutionCalculation)修改Attribute.

GameplayEffect有三种持续类型: 即刻(Instant), 持续(Duration)和无限(Infinite).

额外地, GameplayEffect可以添加/执行GameplayCue, 即刻(Instant)GameplayEffect可以调用GameplayCue GameplayTag的Execute而持续(Duration)或无限(Infinite)可以调用GameplayCue GameplayTag的Add和Remove.

类型	GameplayCue事件	何时使用
即刻 (Instant)	Execute	对Attribute中BaseValue立即进行的永久性修改. 其不会应用GameplayTag, 哪怕是一帧.
持续 (Duration)	Add & Remove	对Attribute中CurrentValue的临时修改和当GameplayEffect过期或手动移除时, 应用将要被移除的GameplayTag. 持续时间是在UGameplayEffect类/蓝图中明确的.
无限 (Infinite)	Add & Remove	对Attribute中CurrentValue的临时修改和当GameplayEffect移除时, 应用将要被移除的GameplayTag. 该类型自身永不过期且必须由某个Ability或ASC手动移除.

持续(Duration)和无限(Infinite)GameplayEffect可以选择应用周期性的Effect, 其每过X秒(由周期定义)就应用一次Modifier和Execution, 当周期性的Effect修改Attribute的BaseValue和执行GameplayCue时就被视为即刻(Instant)GameplayEffect, 这种类型的Effect对于像随时间推移的持续伤害(damage over time, DOT)很有用. **Note**: 周期性的Effect不能被预测.

如果持续(Duration)和无限(Infinite)GameplayEffect的Ongoing Tag Requirements未满足/满足的话(Gameplay Effect Tags), 那么它们在应用后就可以被暂时的关闭和打开, 关闭GameplayEffect会移除其Modifier和已应用GameplayTag效果, 但是不会移除该GameplayEffect, 重新打开GameplayEffect会重新应用其Modifier和GameplayTag.

如果你需要手动重新计算某个持续(Duration)或无限(Infinite)GameplayEffect的Modifier(假设有一个使用非Attribute数据的MMC), 可以使用和UAbilitySystemComponent::ActiveGameplayEffect.GetActiveGameplayEffect(ActiveHandle).Spec.GetLevel()相同的Level调用UAbilitySystemComponent::ActiveGameplayEffect.SetActiveGameplayEffectLevel(FActiveGameplayEffectHandle ActiveHandle, int32 NewLevel). 当Backing Attribute更新时, 基于Backing Attribute的Modifier会自动更新. SetActiveGameplayEffectLevel()更新Modifier的关键函数是:

```
MarkItemDirty(Effect);
Effect.Spec.CalculateModifierMagnitudes();
// Private function otherwise we'd call these three functions without needing to set the level
to what it already is
UpdateAllAggregatorModMagnitudes(Effect);
```

GameplayEffect一般是不实例化的, 当Ability或ASC想要应用GameplayEffect时, 其会从GameplayEffect的ClassDefaultObject创建一个GameplayEffectSpec, 之后成功应用的GameplayEffectSpec会被添加到一个名为FActiveGameplayEffect的新结构体, 其是ASC在名为ActiveGameplayEffect的特殊结构体容器中追踪的内容.

↑ 返回目录

4.5.2 应用GameplayEffect

GameplayEffect可以被GameplayAbility和ASC中的多个函数应用, 其通常是ApplyGameplayEffectTo的形式, 不同的函数本质上都是最终在目标上调用UAbilitySystemComponent::ApplyGameplayEffectSpecToSelf()的方便函数.

为了在GameplayAbility之外应用GameplayEffect, 例如对于某个投掷物, 你就需要获取到目标的ASC并使用它的函数之一来ApplyGameplayEffectToSelf.

你可以绑定持续(Duration)或无限(Infinite)GameplayEffect的委托来监听其应用到ASC:

```
AbilitySystemComponent->OnActiveGameplayEffectAddedDelegateToSelf.AddUObject(this,
&APACharacterBase::OnActiveGameplayEffectAddedCallback);
```

回调函数:

```
virtual void OnActiveGameplayEffectAddedCallback(UAbilitySystemComponent* Target, const
FGameplayEffectSpec& SpecApplied, FActiveGameplayEffectHandle ActiveHandle);
```

服务端总是会调用该函数而不管同步模式是什么, Autonomous Proxy只会在Full和Mixed同步模式下对于同步的GameplayEffect调用该函数, Simulated Proxy只会在Full同步模式下调用该函数.

⬆ 返回目录

4.5.3 移除GameplayEffect

GameplayEffect可以被GameplayAbility和ASC中的多个函数移除, 其通常是RemoveActiveGameplayEffect的形式, 不同的函数本质上都是最终在目标上调用FActiveGameplayEffectContainer::RemoveActiveEffects()的方便函数.

为了在GameplayAbility之外移除GameplayEffect, 你就需要获取到该目标的ASC并使用它的函数之一来RemoveActiveGameplayEffect.

你可以绑定持续(Duration)或无限(Infinite)GameplayEffect的委托来监听其应用到ASC:

```
AbilitySystemComponent->OnAnyGameplayEffectRemovedDelegate().AddUObject(this,
&APACharacterBase::OnRemoveGameplayEffectCallback);
```

回调函数:

```
virtual void OnRemoveGameplayEffectCallback(const FActiveGameplayEffect& EffectRemoved);
```

服务端总是会调用该函数而不管同步模式是什么, Autonomous Proxy只会在Full和Mixed同步模式下对于可同步的GameplayEffect调用该函数, Simulated Proxy只会在Full同步模式下调用该函数.

⬆ 返回目录

4.5.4 GameplayEffectModifier

Modifier可以修改Attribute并且是唯一可以预测性修改Attribute的方法. 一个GameplayEffect可以有0个或多个Modifier, 每个Modifier通过某个指定的操作只能修改一个Attribute.

操作	描述
Add	将Modifier指定的Attribute加上计算结果. 使用负数以实现减法操作.

操作	描述
Multiply	将Modifier指定的Attribute乘以计算结果.
Divide	将Modifier指定的Attribute除以计算结果.
Override	使用计算结果覆盖Modifier指定的Attribute.

Attribute的CurrentValue是其所有Modifier与其BaseValue计算并总合后的结果, 像下面这样的Modifier总合公式被定义在GameplayEffectAggregator.cpp中的FAggregatorModChannel::EvaluateWithBase:

```
((InlineBaseValue + Additive) * Multiplicitive) / Division
```

OverrideModifier会优先覆盖最后应用的Modifier得出的最终值.

Note: 对于基于百分比的修改, 确保使用Multiply操作以使其在加法操作之后.

Note: 预测(Prediction)对于百分比修改有些问题.

有四种类型的Modifier: ScalableFloat, AttributeBased, CustomCalculationClass, 和 SetByCaller, 它们全都生成一些浮点数, 用于之后基于各自的操作修改指定Modifier的Attribute.

Modifier类型	描述
Scalable Float	FScalableFloat结构体可以指向某个横向为变量, 纵向为等级的Data Table, Scalable Float会以Ability的当前等级自动读取指定Data Table的某行值(或者在GameplayEffectSpec中重写的不同等级), 该值还可以进一步被系数处理, 如果没有指定Data Table/Row, 那么就会将其视为1, 因此该系数就可以在所有等级都硬编码为一个值.
Attribute Based	Attribute Based Modifier将Source(GameplayEffectSpec的创建者)或Target(GameplayEffectSpec的接收者)上的CurrentValue或BaseValue视为Backing Attribute, 可以使用系数和Pre与Post系数和来修改它. Snapshotting意味着当GameplayEffectSpec创建时捕获该Attribute, 而No Snapshotting意味着当GameplayEffectSpec应用时捕获该Attribute.
Custom Calculation Class	Custom Calculation Class为复杂的Modifier提供了最大的灵活性, 该Modifier使用了ModifierMagnitudeCalculation类, 且可以使用系数和Pre与Post系数和来处理浮点值结果.
Set By Caller	SetByCallerModifier是运行时由Ability或GameplayEffectSpec的创建者于GameplayEffect之外设置的值, 例如, 如果你想让伤害值随玩家蓄力技能的长短而变化, 那么就需要使用SetByCaller. SetByCaller本质上是存于GameplayEffectSpec中的TMap<FGameplayTag, float>, Modifier只是告知Aggregator去寻找与提供的GameplayTag相关联的SetByCaller值. Modifier使用的SetByCaller只能使用该概念的GameplayTag形式, FName形式在此处不适用. 如果Modifier被设置为SetByCaller, 但是带有正确GameplayTag的SetByCaller在GameplayEffectSpec中不存在, 那么游戏会抛出一个运行时错误并返回0, 这可能在Divide操作中造成问题. 参阅SetByCallers获取更多关于如何使用SetByCaller的信息.

↑ 返回目录

4.5.4.1 Multiply和Divide Modifier

默认情况下, 所有的Multiply和DivideModifier在对Attribute的BaseValue乘除前都会先加到一起.

```
float FAggregatorModChannel::EvaluateWithBase(float InlineBaseValue, const
FAggregatorEvaluateParameters& Parameters) const
{
    ...
    float Additive = SumMods(Mods[EGameplayModOp::Additive],
GameplayEffectUtilities::GetModifierBiasByModifierOp(EGameplayModOp::Additive), Parameters);
    float Multiplicitive = SumMods(Mods[EGameplayModOp::Multiplicitive],
GameplayEffectUtilities::GetModifierBiasByModifierOp(EGameplayModOp::Multiplicitive),
Parameters);
    float Division = SumMods(Mods[EGameplayModOp::Division],
GameplayEffectUtilities::GetModifierBiasByModifierOp(EGameplayModOp::Division), Parameters);
    ...
    return ((InlineBaseValue + Additive) * Multiplicitive) / Division;
    ...
}
```

```
float FAggregatorModChannel::SumMods(const TArray<FAggregatorMod>& InMods, float Bias, const
FAggregatorEvaluateParameters& Parameters)
{
    float Sum = Bias;

    for (const FAggregatorMod& Mod : InMods)
    {
        if (Mod.Qualifies())
        {
            Sum += (Mod.EvaluatedMagnitude - Bias);
        }
    }

    return Sum;
}
```

摘自GameplayEffectAggregator.cpp

在该公式中`Multiply`和`DivideModifier`都有一个值为1的`Bias`值(加法的`Bias`值为0), 因此它看起来像:

```
1 + (Mod1.Magnitude - 1) + (Mod2.Magnitude - 1) + ...
```

该公式会导致一些意料之外的结果, 首先, 它在对BaseValue乘除之前将所有的`Modifier`都加到了一起, 大部分人都期望将其乘或除在一起, 例如, 你有两个值为1.5的`MultiplyModifier`, 大部分人都期望将BaseValue乘上`1.5 x 1.5 = 2.25`, 然而, 这里是将两个1.5加在一起再乘以BaseValue(50%增量 + 另一个50%增量 = 100%增量). 拿`GameplayPrediction.h`中的一个例子来说, 给基值速度500加上10%的加速buff就是550, 再加上另一个10%的加速buff就是600.

其次, 该公式还有一些对于可以使用哪些值而未说明的的规则, 因为这是考虑Paragon的情况而设计的.

译者注: 说实话, 我没有搞懂下文中原文档作者的逻辑, 可能是没有充分了解项目的原因? 比如在样例项目中, 删除BP_DamageVolume的GameplayEffect中的Executions, 并按照下文例4添加两个Multiply Multipliers, Attribute均为GDAttributeSetBase.XP, Modifier Magnitude均为Scalable Float/5.0, 回到游戏, 击杀一个小兵使XP增加到1, 然后进入BP_DamageVolume, 会发现XP依次变为25, 625..., 进行调试也会发现是Modifier依次相乘的, 并不是作者所说的乘法分配律逻辑. 还有就是为什么符合公式规则的`1 + (0.5 - 1) + (1.1 - 1) = 0.6`是正确的而不符合公式规则的`1 + (0.5 - 1) + (0.5 - 1) = 0`和`1 + (5 - 1) + (5 - 1) = 9`就是错误预期? 这个正确和错误预期是以什么为评判标准的? 是否符合公式规则么? 如果各位明白其中道理, 还请不吝赐教, 在此感谢!

对于`Multiply`和`Divide`中乘法加法公式的规则:

- (最多不超过1个值 < 1) AND (任意数量值位于[1, 2))
- OR (一个值 >= 2)

公式中的Bias基本上都会减去`[1, 2)`区间中的整数位, 第一个Modifier的`Bias`会从最开始的`Sum`值减值(在循环体前设置Bias), 这就是为什么某个值它本身能起作用的原因以及某个小于1的值与`[1, 2)`区间中的值起作用的原因.

`Multiply`的一些例子:
Multipliers: 0.5
`1 + (0.5 - 1) = 0.5`, 正确.

Multipliers: 0.5, 0.5
`1 + (0.5 - 1) + (0.5 - 1) = 0`, 错误预期1? 多个小于1的值在`Modifier`相加中不起作用. Paragon这样设计只是为了使用Multiply Modifier的最负值, 因此最多只会有一个小于1的值乘到BaseValue.

Multipliers: 1.1, 0.5
`1 + (0.5 - 1) + (1.1 - 1) = 0.6`, 正确.

Multipliers: 5, 5
`1 + (5 - 1) + (5 - 1) = 9`, 错误预期10. 它总会是`Modifier`值的和 `- Modifier`个数 `+ 1`.

很多游戏会想要它们的`Modify`和`DivideModifier`在应用到BaseValue之前先乘或除到一起, 为了实现这种需求, 你需要修改`FAggregatorModChannel::EvaluateWithBase()`的引擎代码.

```
float FAggregatorModChannel::EvaluateWithBase(float InlineBaseValue, const
FAggregatorEvaluateParameters& Parameters) const
{
    ...
    float Multiplicitive = MultiplyMods(Mods[EGameplayModOp::Multiplicitive], Parameters);
    ...

    return ((InlineBaseValue + Additive) * Multiplicitive) / Division;
}
```

```
float FAggregatorModChannel::MultiplyMods(const TArray<FAggregatorMod>& InMods, const
FAggregatorEvaluateParameters& Parameters)
{
    float Multiplier = 1.0f;

    for (const FAggregatorMod& Mod : InMods)
    {
        if (Mod.Qualifies())
        {
            Multiplier *= Mod.EvaluatedMagnitude;
        }
    }

    return Multiplier;
}
```

↑ 返回目录

4.5.4.2 Modifier的GameplayTag

每个Modifier都可以设置SourceTag和TargetTag, 它们的作用就像GameplayEffect的Application Tag requirements, 因此只有当Effect应用后才会考虑标签, 对于周期性(Periodic)的无限(Infinite)Effect, 这些标签只会在第一次应用Effect时才会被考虑, 而不是在每次周期执行时.

Attribute Based Modifier也可以设置SourceTagFilter和TargetTagFilter. 当确定Attribute Based Modifier的源(Source)Attribute的Magnitude时, 这些过滤器就会用来将某些Modifier排除在该Attribute之外, 源(Source)或目标(Target)中没有过滤器所有标签的Modifier也会被排除在外.

这更详尽的意思是: 源(Source)ASC和目标(Target)ASC的标签都被GameplayEffect所捕获, 当GameplayEffectSpec创建时, 源(Source)ASC的标签被捕获, 当执行Effect时, 目标(Target)ASC的标签被捕获. 当确定无限(Infinite)或持续(Duration)Effect的Modifier是否满足条件可以被应用(也就是聚合器条件(Aggregator Qualify))并且过滤器已经设置时, 被捕获的标签就会和过滤器进行比对.

⬆ 返回目录

4.5.5 GameplayEffect堆栈

GameplayEffect默认会应用新的GameplayEffectSpec实例, 而不明确或不关心之前已经应用过的尚且存在的GameplayEffectSpec实例. GameplayEffect可以设置到堆栈中, 新的GameplayEffectSpec实例不会添加到堆栈中, 而是修改当前已经存在的GameplayEffectSpec堆栈数. 堆栈只适用于持续(Duration)和无限(Infinite)GameplayEffect.

有两种类型的堆栈: Aggregate by Source和Aggregate by Target.

堆栈类型	描述
Aggregate by Source	目标(Target)上的每个源(Source)ASC都有一个单独的堆栈实例, 每个源(Source)可以应用堆栈中的X个GameplayEffect.
Aggregate by Target	目标(Target)上只有一个堆栈实例而不管源(Source)如何, 每个源(Source)都可以在共享堆栈限制(Shared Stack Limit)内应用堆栈.

堆栈对过期, 持续刷新和周期性刷新也有一些处理策略, 这些在GameplayEffect蓝图中都有很友好的悬浮提示帮助.

样例项目包含一个用于监听GameplayEffect堆栈变化的自定义蓝图节点, HUD UMG Widget使用它来更新玩家拥有的被动护盾堆栈(层数). 该AsyncTask将会一直响应直到手动调用EndTask(), 就像在UMG Widget的Destruct事件中调用那样. 参阅AsyncTaskAttributeChanged.h/cpp.

⬆ 返回目录

4.5.6 授予Ability

GameplayEffect可以授予(Grant)新的GameplayAbility到ASC. 只有持续(Duration)和无限(Infinite)GameplayEffect可以授予Ability.

一个普遍用法是当想要强制另一个玩家做某些事的时候, 像击退或拉取时移动他们, 就会对它们应用一个GameplayEffect来授予其一个自动激活的Ability(查看被动Ability来了解如何在Ability被授予时自动激活它), 从而使其做出相应的动作.

设计师可以决定一个GameplayEffect能够授予哪些Ability, 授予的Ability等级, 将其绑定在什么输入键上以及该Ability的移除策略.

移除策略	描述

移除策略	描述
立即取消Ability	当授予Ability的GameplayEffect从目标移除时, 授予的Ability就会立即取消并移除.
结束时移除Ability	允许授予的Ability完成, 之后将其从目标移除.
无	授予的Ability不受从目标移除的授予GameplayEffect的影响, 目标将会一直拥有该Ability直到之后被手动移除.

⬆ 返回目录

4.5.7 GameplayEffect标签

GameplayEffect 可以带有多个 GameplayTagContainer, 设计师可以编辑每个类别的 Added 和 RemovedGameplayTagContainer, 结果会在编译后显示在 Combined GameplayTagContainer 中. Added 标签是该 GameplayEffect新增的之前其父类没有的标签, Removed标签是其父类拥有但该类没有的标签.

分类	描述
Gameplay Effect Asset Tags	GameplayEffect拥有的标签, 它们自身没有任何功能且只用于描述GameplayEffect.
Granted Tags	存于GameplayEffect中且又用于GameplayEffect所应用ASC的标签. 当GameplayEffect移除时它们也会从ASC中移除. 该标签只作用于持续(Duration)和无限(Infinite)GameplayEffect.
Ongoing Tag Requirements	一旦GameplayEffect应用后, 这些标签将决定GameplayEffect是开启还是关闭. GameplayEffect可以是关闭但仍然是应用的. 如果某个GameplayEffect由于不符合Ongoing Tag Requirements而关闭, 但是之后又满足需求了, 那么该GameplayEffect会重新打开并重新应用它的Modifier. 该标签只作用于持续(Duration)和无限(Infinite)GameplayEffect.
Application Tag Requirements	位于目标上决定某个GameplayEffect是否可以应用到该目标的标签, 如果不满足这些需求, 那么GameplayEffect就不可应用.
Remove Gameplay Effects with Tags	当GameplayEffect成功应用后, 如果位于目标上的该GameplayEffect在其Asset Tags或Granted Tags中有任意一个本标签的话, 其就会自目标上移除.

⬆ 返回目录

4.5.8 免疫

GameplayEffect可以基于GameplayTag实现免疫, 有效阻止其他GameplayEffect应用. 尽管免疫可以由Application Tag Requirements 等方式有效地实现, 但是使用该系统可以在 GameplayEffect 被免疫阻止时提供 UAbilitySystemComponent::OnImmunityBlockGameplayEffectDelegate委托(Delegate).

GrantedApplicationImmunityTags会检查源(Source)ASC(包括源Ability的AbilityTag, 如果有的话)是否包含特定的标签, 这是一种基于确定Character或源(Source)的标签对其所有GameplayEffect提供免疫的方法.

Granted Application Immunity Query会检查传入的GameplayEffectSpec是否与其查询条件相匹配, 从而阻止或允许其应用.

GameplayEffect蓝图中的查询条件都有友好的悬浮提示帮助.

↑ 返回目录

4.5.9 GameplayEffectSpec

GameplayEffectSpec(GESpec)可以看作是GameplayEffect的实例, 它保存了一个其所代表的GameplayEffect类引用, 创建时的等级和创建者, 它在应用之前可以在运行时(Runtime)自由的创建和修改, 不像GameplayEffect应该由设计师在运行前创建. 当应用GameplayEffect时, GameplayEffectSpec会自GameplayEffect创建并且会实际应用到目标(Target).

GameplayEffectSpec是由UAbilitySystemComponent::MakeOutgoingSpec()(BlueprintCallable)自GameplayEffect创建的. GameplayEffectSpec不必立即应用. 通常是将GameplayEffectSpec传递给创建自Ability的投掷物, 该投掷物可以应用到它之后击中的目标. 当GameplayEffectSpec成功应用后, 就会返回一个名为FActiveGameplayEffect的新结构体.

GameplayEffectSpec的重要内容:

- 创建该GameplayEffectSpec的GameplayEffect类.
- 该GameplayEffectSpec的等级. 通常和创建GameplayEffectSpec的Ability的等级一样, 但是可以是不同的.
- GameplayEffectSpec的持续时间. 默认是GameplayEffect的持续时间, 但是可以是不同的.
- 对于周期性Effect中GameplayEffectSpec的周期, 默认是GameplayEffect的周期, 但是可以是不同的.
- 该GameplayEffectSpec的当前堆栈数. 堆栈限制取决于GameplayEffect.
- GameplayEffectContextHandle表明该GameplayEffectSpec由谁创建.
- Attribute在GameplayEffectSpec创建时由Snapshot捕获.
- 除了GameplayEffect授予的GameplayTags, GameplayEffectSpec还会授予目标(Target)DynamicGrantedTags.
- 除了GameplayEffect拥有的AssetTags, GameplayEffectSpec还会拥有DynamicAssetTags.
- SetByCaller TMaps.

↑ 返回目录

4.5.9.1 SetByCaller

SetByCaller允许GameplayEffectSpec拥有和GameplayTag或FName相关联的浮点值, 它们存储在GameplayEffectSpec上其各自的TMaps: TMap<FGameplayTag, float>和TMap<FName, float>中, 可以作为GameplayEffect的Modifier或者传递浮点值的通用方法使用. 其普遍用法是经由SetByCaller传递某个Ability内部生成的数值数据到GameplayEffectExecutionCalculations或ModifierMagnitudeCalculations.

SetByCaller 使用	说明
Modifier	必须提前在GameplayEffect类中定义. 只能使用GameplayTag形式. 如果在GameplayEffect类中定义而GameplayEffectSpec中没有相应的标签/浮点值对, 那么游戏在GameplayEffectSpec应用时会抛出运行时错误并返回0, 这对于Divide操作是个潜在问题, 参阅Modifier.
其他	无需提前定义. 读取GameplayEffectSpec中不存在的SetByCaller会返回一个由开发者定义的可带有警告信息的默认值.

为了在蓝图中指定SetByCaller值, 请使用相应形式(GameplayTag或FName)的蓝图节点.

为了在蓝图中读取SetByCaller值, 需要在蓝图中创建自定义节点.

为了在C++中指定SetByCaller值, 需要使用相应形式的函数(GameplayTag或FName).

```
void FGameplayEffectSpec::SetSetByCallerMagnitude(FName DataName, float Magnitude);
```

```
void FGameplayEffectSpec::SetSetByCallerMagnitude(FGameplayTag DataTag, float Magnitude);
```

为了在C++中读取SetByCaller的值, 需要使用相应形式的函数(GameplayTag或FName).

```
float GetSetByCallerMagnitude(FName DataName, bool WarnIfNotFound = true, float
DefaultIfNotFound = 0.f) const;
```

```
float GetSetByCallerMagnitude(FGameplayTag DataTag, bool WarnIfNotFound = true, float
DefaultIfNotFound = 0.f) const;
```

我建议使用GameplayTag形式而不是FName形式, 这可以避免蓝图中的拼写错误, 并且当GameplayEffectSpec同步时, GameplayTag比FName在网络传输中更有效率, 因为TMap也会同步.

⬆ 返回目录

4.5.10 GameplayEffectContext

GameplayEffectContext结构体存有关于GameplayEffectSpec创建者(Instigator)和TargetData的信息, 这也是一个很好的可继承结构体以在ModifierMagnitudeCalculation/GameplayEffectExecutionCalculation, AttributeSet和GameplayCue之间传递任意数据.

继承GameplayEffectContext:

1. 继承FGameplayEffectContext.
2. 重写FGameplayEffectContext::GetScriptStruct().
3. 重写FGameplayEffectContext::Duplicate().
4. 如果新数据需要同步的话, 重写FGameplayEffectContext::NetSerialize().
5. 对子结构体实现TStructOpsTypeTraits, 就像父结构体FGameplayEffectContext有的那样.
6. 在AbilitySystemGlobals类中重写AllocGameplayEffectContext()以返回一个新的子结构体对象.

GASShooter使用了一个子结构体GameplayEffectContext来添加可以在GameplayCue中访问的TargetData, 特别是对于霰弹枪, 因为它可以击打多个敌人.

⬆ 返回目录

4.5.11 Modifier Magnitude Calculation

ModifierMagnitudeCalculations(ModMagcCalc或MMC)是在GameplayEffect中作为Modifier使用的强有力的类, 它的功能类似GameplayEffectExecutionCalculation但是要逊色一些, 最重要的是它是可预测的. 它唯一要做的就是自CalculateBaseMagnitude_Implementation()返回浮点值, 你可以在C++和蓝图中继承并重写该函数.

MMC可以用于各种持续时间的GameplayEffect - 即刻(Instant), 持续(Duration), 无限(Infinite)和周期性(Periodic).

MMC的优势在于能够完全访问GameplayEffectSpec来读取GameplayTag和SetByCaller, 从而能够捕获GameplayEffect的源(Source)或目标(Target)上任意数量的Attribute值. Attribute可以被Snapshot也可以不被Snapshot, Snapshotted Attribute在GameplayEffectSpec创建时被捕获而非Snapshotted Attribute在GameplayEffectSpec应用时被捕获并且该Attribute被无限(Infinite)或持续(Duration)GameplayEffect修改时会自动更新. 捕获Attribute会自ASC现有的Modifier重新计算它们的CurrentValue, 该重新计算**不会**执行AbilitySet中的PreAttributeChange(), 因此所有的限制操作(Clamp)必须在这里重新处理.

Snapshot	源(Source)或目标(Target)	在GameplayEffectSpec中捕获	Attribute被无限(Infinite)或持续(Duration)GameplayEffect修改时自动更新
是	Source	创建	否
是	Target	应用	否
否	Source	应用	是
否	Target	应用	是

MMC的结果浮点值可以进一步由系数和前后系数之和在GameplayEffect的Modifier中修改.

举一个MMC的例子, 该MMC会捕获目标的魔法值Attribute并因为毒药Effect而将其减少, 其减少量的变化取决于目标所拥有的魔法值和可能拥有的某个标签:

```
UPAMMC_PoisonMana::UPAMMC_PoisonMana()
{

    //ManaDef defined in header FGameplayEffectAttributeCaptureDefinition ManaDef;
    ManaDef.AttributeToCapture = UPAAttributeSetBase::GetManaAttribute();
    ManaDef.AttributeSource = EGameplayEffectAttributeCaptureSource::Target;
    ManaDef.bSnapshot = false;

    //MaxManaDef defined in header FGameplayEffectAttributeCaptureDefinition MaxManaDef;
    MaxManaDef.AttributeToCapture = UPAAttributeSetBase::GetMaxManaAttribute();
    MaxManaDef.AttributeSource = EGameplayEffectAttributeCaptureSource::Target;
    MaxManaDef.bSnapshot = false;

    RelevantAttributesToCapture.Add(ManaDef);
    RelevantAttributesToCapture.Add(MaxManaDef);
}

float UPAMMC_PoisonMana::CalculateBaseMagnitude_Implementation(const FGameplayEffectSpec & Spec)
const
{
    // Gather the tags from the source and target as that can affect which buffs should be used
    const FGameplayTagContainer* SourceTags = Spec.CapturedSourceTags.GetAggregatedTags();
    const FGameplayTagContainer* TargetTags = Spec.CapturedTargetTags.GetAggregatedTags();

    FAggregatorEvaluateParameters EvaluationParameters;
    EvaluationParameters.SourceTags = SourceTags;
    EvaluationParameters.TargetTags = TargetTags;

    float Mana = 0.f;
    GetCapturedAttributeMagnitude(ManaDef, Spec, EvaluationParameters, Mana);
    Mana = FMath::Max<float>(Mana, 0.0f);

    float MaxMana = 0.f;
    GetCapturedAttributeMagnitude(MaxManaDef, Spec, EvaluationParameters, MaxMana);
    MaxMana = FMath::Max<float>(MaxMana, 1.0f); // Avoid divide by zero

    float Reduction = -20.0f;
    if (Mana / MaxMana > 0.5f)
    {
        // Double the effect if the target has more than half their mana
        Reduction *= 2;
```

```
    }

    if (TargetTags-
>HasTagExact(FGameplayTag::RequestGameplayTag(FName("Status.WeakToPoisonMana"))))
    {
        // Double the effect if the target is weak to PoisonMana
        Reduction *= 2;
    }

    return Reduction;
}
```

如果你没有在MMC的构造函数中将FGameplayEffectAttributeCaptureDefinition添加到RelevantAttributesToCapture中并且尝试捕获Attribute, 那么将会得到一个关于捕获时缺失Spec的错误. 如果不需要捕获Attribute, 那么就不必添加什么到RelevantAttributesToCapture.

⬆ 返回目录

4.5.12 Gameplay Effect Execution Calculation

GameplayEffectExecutionCalculation(ExecutionCalculation, Execution(你会在插件代码里经常看到这个词)或ExecCalc)是GameplayEffect对ASC进行修改最强有力的方式. 像ModifierMagnitudeCalculation一样, 它也可以捕获Attribute并选择性地为其创建Snapshot, 和MMC不同的是, 它可以修改多个Attribute并且基本上可以处理程序员想要做的任何事. 这种强有力和灵活性的负面就是它是不可预测的且必须在C++中实现.

ExecutionCalculation只能由即刻(Instant)和周期性(Periodic)GameplayEffect使用, 插件中所有和"Execute"相关的一般都引用到这两种类型的GameplayEffect.

当GameplayEffectSpec创建时, Snapshot会捕获Attribute, 而当GameplayEffectSpec应用时, 非Snapshot会捕获Attribute. 捕获Attribute会自ASC现有的Modifier重新计算它们的CurrentValue, 该重新计算**不会**执行AbilitySet中的PreAttributeChange(), 因此所有的限制操作(Clamp)必须在这里重新处理.

快照	Source或Target	在GameplayEffectSpec中捕获
是	Source	创建
是	Target	应用
否	Source	应用
否	Target	应用

为了设置Attribute捕获, 我们采用Epic的ActionRPG样例项目使用的方式, 定义一个保存和声明如何捕获Attribute的结构体, 并在该结构体的构造函数中创建一个它的副本(Copy). 每个ExecCalc都需要有一个这样的结构体. **Note**: 每个结构体需要一个独一无二的名字, 因为它们共享同一个命名空间, 多个结构体使用相同名字在捕获Attribute时会造成错误(大多是捕获到错误的Attribute值).

对于Local Predicted, Server Only和Server Initiated的GameplayAbility, ExecCalc只在服务端调用.

ExecCalc最普遍的应用场景是计算一个来自很多源(Source)和目标(Target)中Attribute伤害值的复杂公式. 样例项目中有一个简单的ExecCalc用于计算伤害值, 其从GameplayEffectSpec的SetByCaller中读取伤害值, 之后基于从目标(Target)捕获的护盾Attribute来减少该伤害值. 参阅GDDamageExecCalculation.cpp/.h.

⬆ 返回目录

4.5.12.1 发送数据到Execution Calculation

除了捕获Attribute, 还有几种方法可以发送数据到ExecutionCalculation.

⬆ 返回目录

4.5.12.1.1 SetByCaller

任何设置在GameplayEffectSpec中的SetByCaller都可以直接在ExecutionCalculation中读取.

```
const FGameplayEffectSpec& Spec = ExecutionParams.GetOwningSpec();
float Damage = FMath::Max<float>
(Spec.GetSetByCallerMagnitude(FGameplayTag::RequestGameplayTag(FName("Data.Damage")), false,
-1.0f), 0.0f);
```

⬆ 返回目录

4.5.12.1.2 Backing数据Attribute计算Modifier

如果你想硬编码值到GameplayEffect, 可以使用CalculationModifier传递, 其使用捕获的Attribute之一作为Backing数据.

在这个截图例子中, 我们给捕获的伤害值Attribute增加了50, 你也可以将其设为Override来直接传入硬编码值.

当ExecutionCalculation捕获该Attribute时会读取这个值.

```
float Damage = 0.0f;
// Capture optional damage value set on the damage GE as a CalculationModifier under the
ExecutionCalculation
ExecutionParams.AttemptCalculateCapturedAttributeMagnitude(DamageStatics().DamageDef,
EvaluationParameters, Damage);
```

⬆ 返回目录

4.5.12.1.3 Backing数据临时变量计算Modifier

如果你想硬编码值到GameplayEffect, 可以在C++中使用CalculationModifier传递, 其使用一个临时变量或暂时聚合器(Transient Aggregator), 该临时变量与GameplayTag相关联.

在这个截图例子中, 我们使用Data.Damage GameplayTag增加50到一个临时变量.

添加Backing临时变量到你的ExecutionCalculation构造函数:

```
ValidTransientAggregatorIdentifiers.AddTag(FGameplayTag::RequestGameplayTag("Data.Damage"));
```

ExecutionCalculation会使用和Attribute捕获函数相似的特殊捕获函数来读取这个值.

```
float Damage = 0.0f;
ExecutionParams.AttemptCalculateTransientAggregatorMagnitude(FGameplayTag::RequestGameplayTag("D
ata.Damage"), EvaluationParameters, Damage);
```

⬆ 返回目录

4.5.12.1.4 GameplayEffectContext

你可以通过GameplayEffectSpec中的自定义GameplayEffectContext发送数据到ExecutionCalculation.

在ExecutionCalculation中, 你可以自FGameplayEffectCustomExecutionParameters访问EffectContext.

```
const FGameplayEffectSpec& Spec = ExecutionParams.GetOwningSpec();
FGSGameplayEffectContext* ContextHandle = static_cast<FGSGameplayEffectContext*>
(Spec.GetContext().Get());
```

如果你需要修改GameplayEffectSpec中的什么或者EffectContext:

```
FGameplayEffectSpec* MutableSpec = ExecutionParams.GetOwningSpecForPreExecuteMod();
FGSGameplayEffectContext* ContextHandle = static_cast<FGSGameplayEffectContext*>(MutableSpec-
>GetContext().Get());
```

在ExecutionCalculation中修改GameplayEffectSpec时要小心. 参看GetOwningSpecForPreExecuteMod()的注释.

```
/** Non const access. Be careful with this, especially when modifying a spec after attribute
capture. */
FGameplayEffectSpec* GetOwningSpecForPreExecuteMod() const;
```

⬆ 返回目录

4.5.13 自定义应用需求

CustomApplicationRequirement(CAR)类为设计师提供对于GameplayEffect是否可以应用的高阶控制, 而不是对GameplayEffect进行简单的GameplayTag检查. 这可以通过在蓝图中重写CanApplyGameplayEffect()和在C++中重写CanApplyGameplayEffect_Implementation()实现.

CAR的应用场景:

- 目标需要有一定数量的Attribute.
- 目标需要有一定数量的GameplayEffect堆栈.

CAR还有很多高阶功能, 像检查GameplayEffect实例是否已经位于目标上, 修改当前实例的持续时间而不是应用一个新实例(对于CanApplyGameplayEffect()返回false).

⬆ 返回目录

4.5.14 花费(Cost)GameplayEffect

GameplayAbility有一个特别设计用来作为Ability花费(Cost)的可选GameplayEffect. 花费(Cost)是指ASC激活GameplayAbility所必需的Attribute量. 如果某个GA不能提供Cost GE, 那么它就不能被激活. 该Cost GE应该是某个带有一个或多个自Attribute中减值Modifier的即刻(Instant)GameplayEffect. 默认情况下, Cost GE是可以被预测的, 建议保留该功能, 也就是不要使用ExecutionCalculations, MMC对于复杂的花费计算是完美适配并且鼓励使用的.

开始的时候, 你通常会为每个有花费的GA都设置一个独一无二的Cost GE, 一个更高阶的技巧是对多个GA复用一个Cost GE, 只需修改自GA的Cost GE创建的GameplayEffectSpec中指定的数据(花费值是在GA上定义的), **这只作用于**实例化**(Instanced)的Ability.**

复用Cost GE的两种技巧:

1. 使用MMC. 这是最简单的方式. 创建一个从GameplayAbility实例读取花费值的MMC, 你可以从GameplayEffectSpec中获取到该实例.

```
float UPGMMC_HeroAbilityCost::CalculateBaseMagnitude_Implementation(const FGameplayEffectSpec &
Spec) const
{
    const UPGGameplayAbility* Ability = Cast<UPGGameplayAbility>
(Spec.GetContext().GetAbilityInstance_NotReplicated());

    if (!Ability)
    {
        return 0.0f;
    }

    return Ability->Cost.GetValueAtLevel(Ability->GetAbilityLevel());
}
```

在这个例子中, 花费值是一个我添加到GameplayAbility子类上的FScalableFloat.

```
UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Cost")
FScalableFloat Cost;
```

2. **重写UGameplayAbility::GetCostGameplayEffect().** 重写该函数并在运行时创建一个用来读取GameplayAbility中花费值的GameplayEffect.

⬆ 返回目录

4.5.15 冷却(Cooldown)GameplayEffect

GameplayAbility有一个特别设计用来作为Ability冷却(Cooldown)的可选GameplayEffect. 冷却时间决定了激活Ability之后多久可以再次激活. 如果某个GA在冷却中, 那么它就不能被激活. 该Cooldown GE应该是一个不带有Modifier的持续(Duration)GameplayEffect, 并且在GameplayEffect的GrantedTags中每个GameplayAbility或Ability插槽(Slot)(如果你的游戏有分配到插槽的可交换Ability且共享同一个冷却)都有一个独一无二的GameplayTag(Cooldown Tag). GA实际上会检查Cooldown Tag的存在而不是Cooldown GE的存在, 默认情况下, Cooldown GE是可以被预测的, 建议保留该功能, 也就是不要使用ExecutionCalculations, MMC对于复杂的冷却计算是完美适配并且鼓励使用的.

开始的时候, 你通常会为每个有冷却的GA都设置一个独一无二的Cooldown GE, 一个更高阶的技巧是对多个GA复用一个Cooldown GE, 只需修改自GA的Cooldown GE创建的GameplayEffectSpec中指定的数据(冷却时间和Cooldown Tag是在GA上定义的), **这只作用于**实例化**(Instanced)的Ability.**

复用Cooldown GE的两种技巧:

1. 使用SetByCaller. 这是最简单的方式. 使用GameplayTag设置SetByCaller为共享Cooldown GE的持续时间. 在GameplayAbility子类中, 为持续时间定义一个浮点/FScalableFloat变量, 为独一无二的Cooldown Tag定义一个FGameplayTagContainer, 除此之外还要定义一个临时FGameplayTagContainer, 其用来作为Cooldown Tag与Cooldown GE标签并集的返回指针.

```
UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Cooldown")
FScalableFloat CooldownDuration;

UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Cooldown")
FGameplayTagContainer CooldownTags;

// Temp container that we will return the pointer to in GetCooldownTags().
// This will be a union of our CooldownTags and the Cooldown GE's cooldown tags.
UPROPERTY()
FGameplayTagContainer TempCooldownTags;
```

之后重写UGameplayAbility::GetCooldownTags()以返回Cooldown Tag和所有现有Cooldown GE标签的并集.

```
const FGameplayTagContainer * UPGGameplayAbility::GetCooldownTags() const
{
    FGameplayTagContainer* MutableTags = const_cast<FGameplayTagContainer*>(&TempCooldownTags);
    const FGameplayTagContainer* ParentTags = Super::GetCooldownTags();
    if (ParentTags)
    {
        MutableTags->AppendTags(*ParentTags);
    }
    MutableTags->AppendTags(CooldownTags);
    return MutableTags;
}
```

最后, 重写UGameplayAbility::ApplyCooldown()以注入我们自己的Cooldown Tag, 并将SetByCaller添加到Cooldown GameplayEffectSpec.

```
void UPGGameplayAbility::ApplyCooldown(const FGameplayAbilitySpecHandle Handle, const
FGameplayAbilityActorInfo * ActorInfo, const FGameplayAbilityActivationInfo ActivationInfo)
const
{
    UGameplayEffect* CooldownGE = GetCooldownGameplayEffect();
    if (CooldownGE)
    {
        FGameplayEffectSpecHandle SpecHandle = MakeOutgoingGameplayEffectSpec(CooldownGE-
>GetClass(), GetAbilityLevel());
        SpecHandle.Data.Get()->DynamicGrantedTags.AppendTags(CooldownTags);
        SpecHandle.Data.Get()->SetSetByCallerMagnitude(FGameplayTag::RequestGameplayTag(FName(
 OurSetByCallerTag  )), CooldownDuration.GetValueAtLevel(GetAbilityLevel()));
        ApplyGameplayEffectSpecToOwner(Handle, ActorInfo, ActivationInfo, SpecHandle);
    }
}
```

下面图片中, 冷却时间Modifier被设置为SetByCaller, 其Data Tag为Data.Cooldown. Data.Cooldown就是上面代码中的OurSetByCallerTag.

2. 使用MMC. 它的设置与上文所提的一致, 除了不需要在Cooldown GE和ApplyCost中设置SetByCaller作为持续时间, 相反, 我们需要将持续时间设置为Custom Calculation类并将其指向新创建的MMC.

```
UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Cooldown")
FScalableFloat CooldownDuration;

UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Cooldown")
FGameplayTagContainer CooldownTags;

// Temp container that we will return the pointer to in GetCooldownTags().
// This will be a union of our CooldownTags and the Cooldown GE's cooldown tags.
UPROPERTY()
FGameplayTagContainer TempCooldownTags;
```

之后重写UGameplayAbility::GetCooldownTags()以返回Cooldown Tag和所有现有Cooldown GE标签的并集.

```
const FGameplayTagContainer * UPGGameplayAbility::GetCooldownTags() const
{
    FGameplayTagContainer* MutableTags = const_cast<FGameplayTagContainer*>(&TempCooldownTags);
    const FGameplayTagContainer* ParentTags = Super::GetCooldownTags();
    if (ParentTags)
    {
        MutableTags->AppendTags(*ParentTags);
    }
    MutableTags->AppendTags(CooldownTags);
    return MutableTags;
}
```

最后, 重写UGameplayAbility::ApplyCooldown()以将我们的Cooldown Tag注入Cooldown GameplayEffectSpec.

```
void UPGGameplayAbility::ApplyCooldown(const FGameplayAbilitySpecHandle Handle, const
FGameplayAbilityActorInfo * ActorInfo, const FGameplayAbilityActivationInfo ActivationInfo)
const
{
    UGameplayEffect* CooldownGE = GetCooldownGameplayEffect();
    if (CooldownGE)
    {
        FGameplayEffectSpecHandle SpecHandle = MakeOutgoingGameplayEffectSpec(CooldownGE-
>GetClass(), GetAbilityLevel());
        SpecHandle.Data.Get()->DynamicGrantedTags.AppendTags(CooldownTags);
        ApplyGameplayEffectSpecToOwner(Handle, ActorInfo, ActivationInfo, SpecHandle);
    }
}
```

```
float UPGMMC_HeroAbilityCooldown::CalculateBaseMagnitude_Implementation(const
FGameplayEffectSpec & Spec) const
{
    const UPGGameplayAbility* Ability = Cast<UPGGameplayAbility>
(Spec.GetContext().GetAbilityInstance_NotReplicated());

    if (!Ability)
    {
        return 0.0f;
    }

    return Ability->CooldownDuration.GetValueAtLevel(Ability->GetAbilityLevel());
}
```

⬆ 返回目录

4.5.15.1 获取Cooldown GameplayEffect的剩余时间

```
bool APGPlayerState::GetCooldownRemainingForTag(FGameplayTagContainer CooldownTags, float &
TimeRemaining, float & CooldownDuration)
{
    if (AbilitySystemComponent && CooldownTags.Num() > 0)
    {
        TimeRemaining = 0.f;
        CooldownDuration = 0.f;

        FGameplayEffectQuery const Query =
FGameplayEffectQuery::MakeQuery_MatchAnyOwningTags(CooldownTags);
        TArray< TPair<float, float> > DurationAndTimeRemaining = AbilitySystemComponent-
>GetActiveEffectsTimeRemainingAndDuration(Query);
        if (DurationAndTimeRemaining.Num() > 0)
        {
            int32 BestIdx = 0;
            float LongestTime = DurationAndTimeRemaining[0].Key;
            for (int32 Idx = 1; Idx < DurationAndTimeRemaining.Num(); ++Idx)
            {
                if (DurationAndTimeRemaining[Idx].Key > LongestTime)
                {
                    LongestTime = DurationAndTimeRemaining[Idx].Key;
                    BestIdx = Idx;
                }
            }

            TimeRemaining = DurationAndTimeRemaining[BestIdx].Key;
            CooldownDuration = DurationAndTimeRemaining[BestIdx].Value;

            return true;
        }
    }

    return false;
}
```

Note: 在客户端上查询剩余冷却时间要求其可以接收同步的`GameplayEffect`, 这依赖于它们`ASC`的同步模式.

⬆ 返回目录

4.5.15.2 监听冷却开始和结束

为了监听某个冷却何时开始，你可以通过绑定`AbilitySystemComponent->OnActiveGameplayEffectAddedDelegateToSelf`或者`AbilitySystemComponent->RegisterGameplayTagEvent(CooldownTag, EGameplayTagEventType::NewOrRemoved)`分别在`Cooldown GE`应用和`Cooldown Tag`添加时作出响应. 我建议监听`Cooldown GE`何时应用, 因为这时还可以访问应用它的`GameplayEffectSpec`. 由此你可以确定当前`Cooldown GE`是客户端预测的还是由服务端校正的.

为了监听某个冷却何时结束, 你可以通过绑定AbilitySystemComponent->OnAnyGameplayEffectRemovedDelegate()或者 AbilitySystemComponent->RegisterGameplayTagEvent(CooldownTag， EGameplayTagEventType::NewOrRemoved)分别在Cooldown GE移除和Cooldown Tag移除时作出响应. 我建议监听Cooldown Tag何时移除, 因为当服务端校正的Cooldown GE到来时, 会移除客户端预测的Cooldown GE, 这会响应OnAnyGameplayEffectRemovedDelegate(), 即使仍处于冷却过程中. 预测的Cooldown GE在移除时和服务端校正的Cooldown GE在应用时Cooldown Tag都不会改变.

Note: 在客户端上监听某个GameplayEffect添加或移除要求其可以接收同步的GameplayEffect, 这依赖于它们ASC的同步模式.

样例项目包含一个用于监听冷却开始和结束的自定义蓝图节点, HUD UMG Widget使用它来更新陨石技能的剩余冷却时间, 该AsyncTask会一直响应直到手动调用EndTask(), 就像在UMG Widget的Destruct事件中调用那样. 参阅AsyncTaskAttributeChanged.h/cpp.

⬆ 返回目录

4.5.15.3 预测冷却时间

目前冷却时间不是真正可预测的. 我们可以在客户端预测的Cooldown GE应用时启动UI的冷却时间计时器, 但是GameplayAbility的实际冷却时间是由服务端的冷却时间剩余决定的. 取决于玩家的延迟情况, 可能客户端预测的冷却已经结束, 但是服务端上的GameplayAbility仍处于冷却过程, 这会阻止GameplayAbility的立刻再激活直到服务端冷却结束.

样例项目通过在客户端预测的冷却开始时灰化陨石技能的图标, 之后在服务端校正的Cooldown GE到来时启动冷却计时器处理该问题.

在实际游戏中导致的结果就是高延迟的玩家相比低延迟的玩家对冷却时间短的技能有更低的触发率, 从而处于劣势, Fortnite通过使其武器使用无需冷却GameplayEffect的自定义Bookkeeping而避免该现象.

Epic希望在未来的GAS迭代版本中实现真正的冷却预测(玩家可以激活一个在客户端冷却完成但服务端仍处于冷却过程的GameplayAbility).

⬆ 返回目录

4.5.16 修改已激活GameplayEffect的持续时间

为了修改Cooldown GE或其他任何持续(Duration)GameplayEffect的剩余时间, 我们需要修改GameplayEffectSpec的持续时间, 更新它的StartServerWorldTime, CachedStartServerWorldTime, StartWorldTime, 并且使用CheckDuration()重新检查持续时间. 在服务端上完成这些操作并将FActiveGameplayEffect标记为dirty, 其会将这些修改同步到客户端. **Note:** 该操作包含一个const_cast, 这可能不是Epic希望的修改持续时间的方法, 但是迄今为止它看起来运行得很好.

```
bool UPAAbilitySystemComponent::SetGameplayEffectDurationHandle(FActiveGameplayEffectHandle
Handle, float NewDuration)
{
    if (!Handle.IsValid())
    {
        return false;
    }

    const FActiveGameplayEffect* ActiveGameplayEffect = GetActiveGameplayEffect(Handle);
    if (!ActiveGameplayEffect)
    {
        return false;
    }
```

```
    FActiveGameplayEffect* AGE = const_cast<FActiveGameplayEffect*>(ActiveGameplayEffect);
    if (NewDuration > 0)
    {
        AGE->Spec.Duration = NewDuration;
    }
    else
    {
        AGE->Spec.Duration = 0.01f;
    }

    AGE->StartServerWorldTime = ActiveGameplayEffects.GetServerWorldTime();
    AGE->CachedStartServerWorldTime = AGE->StartServerWorldTime;
    AGE->StartWorldTime = ActiveGameplayEffects.GetWorldTime();
    ActiveGameplayEffects.MarkItemDirty(*AGE);
    ActiveGameplayEffects.CheckDuration(Handle);

    AGE->EventSet.OnTimeChanged.Broadcast(AGE->Handle, AGE->StartWorldTime, AGE->GetDuration());
    OnGameplayEffectDurationChange(*AGE);

    return true;
}
```

⬆ 返回目录

4.5.17 运行时创建动态GameplayEffect

在运行时创建动态GameplayEffect是一个高阶技术, 你不必经常使用它.

只有即刻(Instant)GameplayEffect可以在运行时由C++创建, 持续(Duration)和无限(Infinite)GameplayEffect不能在运行时动态创建, 因为它们在同步时会寻找并不存在的GameplayEffect类定义. 为了实现该功能, 你应该创建一个原型GameplayEffect类, 就像平时在编辑器中做的那样, 之后根据运行时所需来定制化GameplayEffectSpec.

运行时创建的即刻(Instant)GameplayEffect也可以在客户端预测的GameplayAbility中调用. 然而, 目前还不明确动态创建是否有副作用.

样例项目会在角色AttributeSet中的值受到致命一击时创建该GameplayEffect来将金币和经验点数返还给击杀者.

```
// Create a dynamic instant Gameplay Effect to give the bounties
UGameplayEffect* GEBounty = NewObject<UGameplayEffect>(GetTransientPackage(),
FName(TEXT("Bounty")));
GEBounty->DurationPolicy = EGameplayEffectDurationType::Instant;

int32 Idx = GEBounty->Modifiers.Num();
GEBounty->Modifiers.SetNum(Idx + 2);

FGameplayModifierInfo& InfoXP = GEBounty->Modifiers[Idx];
InfoXP.ModifierMagnitude = FScalableFloat(GetXPBounty());
InfoXP.ModifierOp = EGameplayModOp::Additive;
InfoXP.Attribute = UGDAttributeSetBase::GetXPAttribute();

FGameplayModifierInfo& InfoGold = GEBounty->Modifiers[Idx + 1];
InfoGold.ModifierMagnitude = FScalableFloat(GetGoldBounty());
InfoGold.ModifierOp = EGameplayModOp::Additive;
InfoGold.Attribute = UGDAttributeSetBase::GetGoldAttribute();

Source->ApplyGameplayEffectToSelf(GEBounty, 1.0f, Source->MakeEffectContext());
```

第二个样例展示了在一个客户端预测的GameplayAbility中创建运行时GameplayEffect, 使用风险自负(查看代码中的注释)!

```
UGameplayAbilityRuntimeGE::UGameplayAbilityRuntimeGE()
{
    NetExecutionPolicy = EGameplayAbilityNetExecutionPolicy::LocalPredicted;
}

void UGameplayAbilityRuntimeGE::ActivateAbility(const FGameplayAbilitySpecHandle Handle, const
FGameplayAbilityActorInfo* ActorInfo, const FGameplayAbilityActivationInfo ActivationInfo, const
FGameplayEventData* TriggerEventData)
{
    if (HasAuthorityOrPredictionKey(ActorInfo, &ActivationInfo))
    {
        if (!CommitAbility(Handle, ActorInfo, ActivationInfo))
        {
            EndAbility(Handle, ActorInfo, ActivationInfo, true, true);
        }

        // Create the GE at runtime.
        UGameplayEffect* GameplayEffect = NewObject<UGameplayEffect>(GetTransientPackage(),
TEXT("RuntimeInstantGE"));
        GameplayEffect->DurationPolicy = EGameplayEffectDurationType::Instant; // Only instant
works with runtime GE.

        // Add a simple scalable float modifier, which overrides MyAttribute with 42.
        // In real world applications, consume information passed via TriggerEventData.
        const int32 Idx = GameplayEffect->Modifiers.Num();
        GameplayEffect->Modifiers.SetNum(Idx + 1);
        FGameplayModifierInfo& ModifierInfo = GameplayEffect->Modifiers[Idx];
        ModifierInfo.Attribute.SetUProperty(UMyAttributeSet::GetMyModifiedAttribute());
        ModifierInfo.ModifierMagnitude = FScalableFloat(42.f);
        ModifierInfo.ModifierOp = EGameplayModOp::Override;

        // Apply the GE.

        // Create the GESpec here to avoid the behavior of ASC to create GESpecs from the GE
class default object.
        // Since we have a dynamic GE here, this would create a GESpec with the base
GameplayEffect class, so we
        // would lose our modifiers. Attention: It is unknown, if this "hack" done here can have
drawbacks!
        // The spec prevents the GE object being collected by the GarbageCollector, since the GE
is a UPROPERTY on the spec.
        FGameplayEffectSpec* GESpec = new FGameplayEffectSpec(GameplayEffect, {}, 0.f); //
"new", since lifetime is managed by a shared ptr within the handle
        ApplyGameplayEffectSpecToOwner(Handle, ActorInfo, ActivationInfo,
FGameplayEffectSpecHandle(GESpec));
    }
    EndAbility(Handle, ActorInfo, ActivationInfo, false, false);
}
```

⬆ 返回目录

4.5.18 GameplayEffect Containers

Epic的Action RPG样例项目实现了一个名为FGameplayEffectContainer的结构体, 它不属于原生GAS, 但是对于包含GameplayEffect和TargetData极其好用, 它会使一些过程自动化, 比如从GameplayEffect中创建GameplayEffectSpec并在其GameplayEffectContext中设置默认值. 在GameplayAbility中创建GameplayEffectContainer并将其传递给已生成的投掷物是非常简单和显而易见的, 然而我没有选择在样例项目中实现GameplayEffectContainer, 因为我想向你展示的是没有它的原生GAS, 但是我高度建议你研究一下它并将其纳入到你的项目中.

为了访问GameplayEffectContainer中的GESpec以求做一些诸如添加SetByCaller的操作, 请使用FGameplayEffectContainer结构体中的GESpec数组索引访问GESpec引用, 这要求你需要提前知道想要访问的GESpec的索引.

GameplayEffectContainer还包含一个可选的用于定位(Target)的高效方法.

⬆ 返回目录

4.6 Gameplay Abilities

4.6.1 GameplayAbility定义

GameplayAbility(GA)是Actor在游戏中可以触发的一切行为和技能. 多个GameplayAbility可以在同一时刻激活, 例如奔跑和射击. 其可由蓝图或C++完成.

GameplayAbility示例:

- 跳跃
- 奔跑
- 射击
- 每X秒被动地阻挡一次攻击
- 使用药剂
- 开门
- 收集资源
- 建造

不应该使用GameplayAbility的场景:

- 基础移动输入
- 一些与UI的交互 - 不要使用GameplayAbility从商店中购买物品

这些不是规定, 只是我的建议而已, 你的设计和实现可能是多样的.

GameplayAbility自带有根据等级修改Attribute变化量或者GameplayAbility作用的默认功能.

GameplayAbility运行在所属(Owning)客户端还是服务端取决于网络执行策略(Net Execution Policy)而不是Simulated Proxy. 网络执行策略(Net Execution Policy)决定某个GameplayAbility是否是客户端可预测的, 其对于可选的Cost和Cooldown GameplayEffect包含有默认行为. GameplayAbility使用AbilityTask用于随时间推移而发生的行为, 例如等待某个事件, 等待某个Attribute改变, 等待玩家选择一个目标或者使用Root Motion Source移动某个Character. **Simulated Client不会运行GameplayAbility,** 而是当服务端执行Ability时, 任何需要在Simulated Proxy上展现的视觉效果(像动画蒙太奇)将会被同步(Replicate)或者通过AbilityTask进行RPC或者对于像声音和粒子这样的装饰效果使用GameplayCue.

所有的GameplayAbility都会有它们各自由你的游戏逻辑重写的ActivateAbility()函数, 附加的逻辑可以添加到EndAbility(), 其会在GameplayAbility完成或取消时执行.

一个简单的GameplayAbility流程图:

一个更复杂GameplayAbility流程图:

复杂的Ability可以使用多个相互交互(激活, 取消等等)的GameplayAbility实现.

⬆ 返回目录

4.6.1.1 Replication Policy

不要使用该选项. 这个名字会误导你并且你并不需要它. GameplayAbilitySpec默认会从服务端向所属(Owning)客户端同步, 上文提到过, **GameplayAbility不会运行在Simulated Proxy上,** 其使用AbilityTask和GameplayCue来同步或者RPC视觉变化到Simulated Proxy. Epic的Dave Ratti已经表明要在未来移除该选项的意愿.

⬆ 返回目录

4.6.1.2 Server Respects Remote Ability Cancellation

这个选项往往会引起麻烦. 它的意思是如果客户端的GameplayAbility由于玩家取消或者自然完成时, 就会强制它的服务端版本结束而不管其是否完成. 最重要的是之后的问题, 特别是对于高延迟玩家所使用的客户端预测的GameplayAbility. 一般情况下禁用该选项.

⬆ 返回目录

4.6.1.3 Replicate Input Directly

设置该选项就会一直向服务端同步输入的按下(Press)和抬起(Release)事件. Epic不建议使用该选项而是依靠创建在已有输入相关的AbilityTask中的Generic Replicated Event(如果你的输入绑定在ASC).

Epic的注释:

```
/** Direct Input state replication. These will be called if bReplicateInputDirectly is true on
the ability and is generally not a good thing to use. (Instead, prefer to use Generic Replicated
Events). */
UAbilitySystemComponent::ServerSetInputPressed()
```

⬆ 返回目录

4.6.2 绑定输入到ASC

ASC允许你直接绑定输入事件并当你授予GameplayAbility时分配这些输入到GameplayAbility, 如果GameplayTag合乎要求, 当按下按键时, 分配到GameplayAbility的输入事件会自动激活各自的GameplayAbility. 分配的输入事件要求使用响应输入的内建AbilityTask.

除了分配的输入事件可以激活GameplayAbility, ASC也接受一般的Confirm和Cancel输入, 这些特殊输入被AbilityTask用来确定像Target Actor的对象或取消它们.

为了绑定输入到ASC, 你必须首先创建一个枚举来将输入事件名称转换为byte, 枚举名必须准确匹配项目设置中用于输入事件的名称, DisplayName就无所谓了.

样例项目中:

```
UENUM(BlueprintType)
enum class EGDAbilityInputID : uint8
{
```

```
    // 0 None
    None            UMETA(DisplayName = "None"),
    // 1 Confirm
    Confirm         UMETA(DisplayName = "Confirm"),
    // 2 Cancel
    Cancel          UMETA(DisplayName = "Cancel"),
    // 3 LMB
    Ability1        UMETA(DisplayName = "Ability1"),
    // 4 RMB
    Ability2        UMETA(DisplayName = "Ability2"),
    // 5 Q
    Ability3        UMETA(DisplayName = "Ability3"),
    // 6 E
    Ability4        UMETA(DisplayName = "Ability4"),
    // 7 R
    Ability5        UMETA(DisplayName = "Ability5"),
    // 8 Sprint
    Sprint          UMETA(DisplayName = "Sprint"),
    // 9 Jump
    Jump            UMETA(DisplayName = "Jump")
};
```

如果你的ASC位于Character, 那么就在SetupPlayerInputComponent()中包含用于绑定到ASC的函数.

```
// Bind to AbilitySystemComponent
AbilitySystemComponent->BindAbilityActivationToInputComponent(PlayerInputComponent,
FGameplayAbilityInputBinds(FString("ConfirmTarget"), FString("CancelTarget"),
FString("EGDAbilityInputID"), static_cast<int32>(EGDAbilityInputID::Confirm), static_cast<int32>
(EGDAbilityInputID::Cancel)));
```

如果你的ASC位于PlayerState, SetupPlayerInputComponent()中有一个潜在的竞争情况就是PlayerState还没有同步到客户端, 因此, 我建议尝试在SetupPlayerInputComponent()和OnRep_PlayerState()中绑定输入, 只有OnRep_PlayerState()自身是不充分的, 因为可能有种情况是当PlayerState在PlayerController告知客户端调用用于创建InputComponent的ClientRestart()前同步时, Actor的InputComponent可能为NULL. 样例项目演示了尝试使用一个布尔值控制流程从而在两个位置绑定, 这样实际上只绑定了一次.

Note: 样例项目枚举中的Confirm和Cancel没有匹配项目设置中的输入事件名称(ConfirmTarget和CancelTarget), 但是我们在BindAbilityActivationToInputComponent()中提供了它们之间的映射, 这是特殊的, 因为我们提供了映射并且它们无需匹配, 但是它们是可以匹配的. 枚举中的其他输入都必须匹配项目设置中的输入事件名称.

对于只能用一次输入激活的GameplayAbility(它们总是像MOBA游戏一样存在于相同的"槽"中), 我倾向在UGameplayAbility子类中添加一个变量, 这样我就可以定义他们的输入, 之后在授予Ability的时候可以从ClassDefaultObject中读取这个值.

⬆返回目录

4.6.2.1 绑定输入时不激活Ability

如果你不想你的GameplayAbility在按键按下时自动激活, 但是仍想将它们绑定到输入以与AbilityTask一起使用, 你可以在UGameplayAbility子类中添加一个新的布尔变量, bActivateOnInput, 其默认值为true并重写UAbilitySystemComponent::AbilityLocalInputPressed().

```
void UGSAbilitySystemComponent::AbilityLocalInputPressed(int32 InputID)
{
    // Consume the input if this InputID is overloaded with GenericConfirm/Cancel and the
GenericConfim/Cancel callback is bound
```

```
    if (IsGenericConfirmInputBound(InputID))
    {
        LocalInputConfirm();
        return;
    }

    if (IsGenericCancelInputBound(InputID))
    {
        LocalInputCancel();
        return;
    }

    // ------------------------------------------------------

    ABILITYLIST_SCOPE_LOCK();
    for (FGameplayAbilitySpec& Spec : ActivatableAbilities.Items)
    {
        if (Spec.InputID == InputID)
        {
            if (Spec.Ability)
            {
                Spec.InputPressed = true;
                if (Spec.IsActive())
                {
                    if (Spec.Ability->bReplicateInputDirectly && IsOwnerActorAuthoritative() ==
false)
                    {
                        ServerSetInputPressed(Spec.Handle);
                    }

                    AbilitySpecInputPressed(Spec);

                    // Invoke the InputPressed event. This is not replicated here. If someone is
listening, they may replicate the InputPressed event to the server.
                    InvokeReplicatedEvent(EAbilityGenericReplicatedEvent::InputPressed,
Spec.Handle, Spec.ActivationInfo.GetActivationPredictionKey());
                }
                else
                {
                    UGSGameplayAbility* GA = Cast<UGSGameplayAbility>(Spec.Ability);
                    if (GA && GA->bActivateOnInput)
                    {
                        // Ability is not active, so try to activate it
                        TryActivateAbility(Spec.Handle);
                    }
                }
            }
        }
    }
}
```

⬆返回目录

4.6.3 授予Ability

向ASC授予GameplayAbility会将其添加到ASC的ActivatableAbilities列表, 从而允许其在满足GameplayTag需求时激活该GameplayAbility.

我们在服务端授予GameplayAbility, 之后其会自动同步GameplayAbilitySpec到所属(Owning)客户端, 其他客户端/Simulated proxy不会接受到GameplayAbilitySpec.

样例项目在游戏开始时将TArray<TSubclassOf<UGDGameplayAbility>>保存在它读取和授予的Character类中.

```
void AGDCharacterBase::AddCharacterAbilities()
{
    // Grant abilities, but only on the server
    if (Role != ROLE_Authority || !AbilitySystemComponent.IsValid() || AbilitySystemComponent-
>CharacterAbilitiesGiven)
    {
        return;
    }

    for (TSubclassOf<UGDGameplayAbility>& StartupAbility : CharacterAbilities)
    {
        AbilitySystemComponent->GiveAbility(
            FGameplayAbilitySpec(StartupAbility,
GetAbilityLevel(StartupAbility.GetDefaultObject()->AbilityID), static_cast<int32>
(StartupAbility.GetDefaultObject()->AbilityInputID), this));
    }

    AbilitySystemComponent->CharacterAbilitiesGiven = true;
}
```

当授予这些GameplayAbility时, 我们就在使用UGameplayAbility类, Ability等级, 其绑定的输入和SourceObject或将该GameplayAbility设置到该ASC的源(Source)创建GameplayAbilitySpec.

⬆ 返回目录

4.6.4 激活Ability

如果某个GameplayAbility被分配给了一个输入事件, 那么当输入按键按下并且它的GameplayTag需求满足时, 它将会自动激活, 这可能并非总是激活GameplayAbility的期望方式. ASC提供了另外四种激活GameplayAbility的方法: 通过GameplayTag, GameplayAbility类, GameplayAbilitySpecHandle和Event, 通过Event激活GameplayAbility允许你传递一个该事件的数据负载(Payload).

```
UFUNCTION(BlueprintCallable, Category = "Abilities")
bool TryActivateAbilitiesByTag(const FGameplayTagContainer& GameplayTagContainer, bool
bAllowRemoteActivation = true);

UFUNCTION(BlueprintCallable, Category = "Abilities")
bool TryActivateAbilityByClass(TSubclassOf<UGameplayAbility> InAbilityToActivate, bool
bAllowRemoteActivation = true);

bool TryActivateAbility(FGameplayAbilitySpecHandle AbilityToActivate, bool
bAllowRemoteActivation = true);

bool TriggerAbilityFromGameplayEvent(FGameplayAbilitySpecHandle AbilityToTrigger,
FGameplayAbilityActorInfo* ActorInfo, FGameplayTag Tag, const FGameplayEventData* Payload,
UAbilitySystemComponent& Component);

FGameplayAbilitySpecHandle GiveAbilityAndActivateOnce(const FGameplayAbilitySpec& AbilitySpec);
```

想要通过Event激活GameplayAbility, GameplayAbility必须设置它的Trigger, 分配一个GameplayTag并为GameplayEvent选择一个选项. 想要发送Event, 就得使用UAbilitySystemBlueprintLibrary::SendGameplayEventToActor(AActor* Actor, FGameplayTag EventTag, FGameplayEventData Payload)函数. 通过Event激活GameplayAbility允许你传递一个数据负载(Payload).

GameplayAbility Trigger也允许你在某个GameplayTag添加或移除时激活该GameplayAbility.

Note: 当从蓝图中的Event激活GameplayAbility时, 你必须使用ActivateAbilityFromEvent节点, 并且标准的ActivateAbility节点不能出现在图表中, 如果ActivateAbility节点存在, 它就会一直被调用而不调用ActivateAbilityFromEvent节点.

Note: 不要忘记应该在GameplayAbility终止时调用EndAbility(), 除非你的GameplayAbility是像被动技能那样一直运行的GameplayAbility.

对于**客户端预测**GameplayAbility的激活序列:

1. **所属(Owning)客户端**调用TryActivateAbility()
2. 调用InternalTryActivateAbility()
3. 调用CanActivateAbility()并返回是否满足GameplayTag需求, ASC是否满足技能花费, GameplayAbility是否不在冷却期和当前是否没有其他实例被激活
4. 调用CallServerTryActivateAbility()并传入其生成的Prediction Key
5. 调用CallActivateAbility()
6. 调用PreActivate(), Epic称之为"boilerplate init stuff"
7. 调用ActivateAbility()最终激活Ability

服务端接收到CallServerTryActivateAbility()

1. 调用ServerTryActivateAbility()
2. 调用InternalServerTryActivateAbility()
3. 调用InternalTryActivateAbility()
4. 调用CanActivateAbility()并返回是否满足GameplayTag需求, ASC是否满足技能花费, GameplayAbility是否不在冷却期和当前是否没有其他实例被激活
5. 如果成功则调用ClientActivateAbilitySucceed()告知客户端更新它的ActivationInfo(即该激活已由服务端确认)并广播OnConfirmDelegate代理. 这和输入确认(Input Confirmation)不一样.
6. 调用CallActivateAbility()
7. 调用PreActivate(), Epic称之为"boilerplate init stuff"
8. 调用ActivateAbility()最终激活Ability

如果服务端在任意时刻激活失败, 就会调用ClientActivateAbilityFailed(), 立即终止客户端的GameplayAbility并撤销所有预测的修改.

⬆ 返回目录

4.6.4.1 被动Ability

为了实现自动激活和持续运行的被动GameplayAbility, 需要重写UGameplayAbility::OnAvatarSet(), 该函数在授予GameplayAbility并设置AvatarActor且调用TryActivateAbility()时自动调用.

我建议添加一个布尔值到你的自定义UGameplayAbility类来表明其在授予时是否应该被激活. 样例项目中的被动护甲叠层Ability是这样做的.

被动GameplayAbility一般有一个仅服务器(Server Only)的网络执行策略(Net Execution Policy).

```
void UGDGameplayAbility::OnAvatarSet(const FGameplayAbilityActorInfo * ActorInfo, const
FGameplayAbilitySpec & Spec)
{
    Super::OnAvatarSet(ActorInfo, Spec);

    if (ActivateAbilityOnGranted)
    {
        bool ActivatedAbility = ActorInfo->AbilitySystemComponent-
>TryActivateAbility(Spec.Handle, false);
    }
}
```

Epic描述该函数为初始化被动Ability的正确位置和应该做一些类似BeginPlay的事情.

⬆ 返回目录

4.6.5 取消Ability

为了从内部取消GameplayAbility, 可以调用CancelAbility(), 其会调用EndAbility()并设置它的WasCancelled参数为true.

为了从外部取消GameplayAbility, ASC提供了一些函数:

```
/** Cancels the specified ability CDO. */
void CancelAbility(UGameplayAbility* Ability);

/** Cancels the ability indicated by passed in spec handle. If handle is not found among
reactivated abilities nothing happens. */
void CancelAbilityHandle(const FGameplayAbilitySpecHandle& AbilityHandle);

/** Cancel all abilities with the specified tags. Will not cancel the Ignore instance */
void CancelAbilities(const FGameplayTagContainer* WithTags=nullptr, const FGameplayTagContainer*
WithoutTags=nullptr, UGameplayAbility* Ignore=nullptr);

/** Cancels all abilities regardless of tags. Will not cancel the ignore instance */
void CancelAllAbilities(UGameplayAbility* Ignore=nullptr);

/** Cancels all abilities and kills any remaining instanced abilities */
virtual void DestroyActiveState();
```

Note: 我发现如果存在一个非实例(Non-Instanced)GameplayAbility时, CancelAllAbilities似乎不能正常运行, 它似乎会命中这个非实例(Non-Instanced)GameplayAbility并放弃继续处理. CancelAbility可以更好地处理非实例(Non-Instanced)GameplayAbility, 样例项目就是这样使用的(跳跃是一个非实例(Non-Instanced)GameplayAbility), 因人而异.

⬆ 返回目录

4.6.6 获取激活的Ability

初学者经常会问"我怎样才能获取激活的Ability?", 也许是用来设置变量或取消它. 多个GameplayAbility可以在同一时刻激活, 因此没有"一个激活的Ability", 相反, 你必须搜索ASC的ActivatableAbility列表(ASC拥有的已授予GameplayAbility)并找到一个与你正在寻找的资源或授予的GameplayTag相匹配的Ability.

UAbilitySystemComponent::GetActivatableAbilities()会返回一个用于遍历的TArray<FGameplayAbilitySpec>.

ASC还有另一个有用的函数, 它将一个GameplayTagContainer作为参数来协助搜索, 而无需手动遍历GameplayAbilitySpec列表. bOnlyAbilitiesThatSatisfyTagRequirements参数只会返回那些GameplayTag满足需求且可以立刻激活的GameplayAbilitySpecs, 例如, 你可能有两个基本的攻击GameplayAbility, 一个使用武器, 另一个使用拳头, 正确的激活取决于武器是否装备并设置了GameplayTag需求. 详见Epic关于函数的注释.

```
UAbilitySystemComponent::GetActivatableGameplayAbilitySpecsByAllMatchingTags(const
FGameplayTagContainer& GameplayTagContainer, TArray < struct FGameplayAbilitySpec* >&
MatchingGameplayAbilities, bool bOnlyAbilitiesThatSatisfyTagRequirements = true)
```

一旦你获取到了寻找的FGameplayAbilitySpec, 那么就可以调用它的IsActive().

⬆ 返回目录

4.6.7 实例化策略

GameplayAbility的实例化策略决定了当GameplayAbility激活时是否和如何实例化.

实例化策略	描述	何时使用的例子
按Actor实例化 (Instanced Per Actor)	每个ASC只能有一个在激活之间复用的GameplayAbility实例.	这可能是你使用最频繁的实例化策略. 你可以对任一Ability使用并在激活之间提供持久化. 设计者可以在激活之间手动重设任意变量.
按操作实例化 (Instanced Per Execution)	每有一个GameplayAbility激活, 就有一个新的GameplayAbility实例创建.	这些GameplayAbility的好处是每次激活时变量都会重置, 其性能要比Instanced Per Actor差, 因为每次激活时都会生成新的GameplayAbility. 样例项目没有使用该方式.
非实例化 (Non-Instanced)	GameplayAbility操作其ClassDefaultObject, 没有实例创建.	它是三种方式中性能最好的, 但是使用它是最受限制的. 非实例化(Non-Instanced)GameplayAbility不能存储状态, 这意味着没有动态变量和不能绑定到AbilityTask委托. 使用它的最佳场景就是需要频繁使用的简单Ability, 像MOBA或RTS游戏中小兵的基础攻击. 样例项目中的跳跃GameplayAbility就是非实例化(Non-Instanced)的.

⬆ 返回目录

4.6.8 网络执行策略(Net Execution Policy)

GameplayAbility的网络执行策略(Net Execution Policy)决定了谁该以什么顺序运行该GameplayAbility.

网络执行策略(Net Execution Policy)	描述
Local Only	GameplayAbility只运行在所属(Owning)客户端. 这对那些只需做本地视觉变化的Ability很有用. 单人游戏应该使用Server Only.
Local Predicted	Local Predicted GameplayAbility首先在所属(Owning)客户端激活, 之后在服务端激活. 服务端版本会纠正客户端预测的所有不正确的地方. 参见Prediction.
Server Only	GameplayAbility只运行在服务端. 被动GameplayAbility一般是Server Only. 单人游戏应该使用该项.
Server Initiated	Server Initiated GameplayAbility首先在服务端激活, 之后在所属(Owning)客户端激活. 我个人使用的不多(如果有的话).

⬆ 返回目录

4.6.9 Ability标签

GameplayAbility自带有内建逻辑的GameplayTagContainer. 这些GameplayTag都不进行同步.

GameplayTagContainer	描述
Ability Tags	GameplayAbility拥有的GameplayTag, 这只是用来描述GameplayAbility的GameplayTag.
Cancel Abilities with Tag	当该GameplayAbility激活时, 其他Ability Tags中拥有这些GameplayTag的GameplayAbility将会被取消.
Block Abilities with Tag	当该GameplayAbility激活时, 其他Ability Tags中拥有这些GameplayTag的GameplayAbility将会阻塞激活.
Activation Owned Tags	当该GameplayAbility激活时, 这些GameplayTag会交给该GameplayAbility的拥有者.
Activation Required Tags	该GameplayAbility只有在其拥有者拥有所有这些GameplayTag时才会激活.
Activation Blocked Tags	该GameplayAbility在其拥有者拥有任意这些标签时不能被激活.
Source Required Tags	该GameplayAbility只有在Source拥有所有这些GameplayTag时才会激活. Source GameplayTag只有在该GameplayAbility由Event触发时设置.
Source Blocked Tags	该GameplayAbility在Source拥有任意这些标签时不能被激活. Source GameplayTag只有在该GameplayAbility由Event触发时设置.
Target Required Tags	该GameplayAbility只有在Target拥有所有这些GameplayTag时才会激活. Target GameplayTag只有在该GameplayAbility由Event触发时设置.
Target Blocked Tags	该GameplayAbility在Target拥有任意这些标签时不能被激活. Target GameplayTag只有在该GameplayAbility由Event触发时设置.

⬆ 返回目录

4.6.10 Gameplay Ability Spec

GameplayAbilitySpec会在GameplayAbility授予后存在于ASC中并定义可激活GameplayAbility - GameplayAbility类, 等级, 输入绑定和必须与GameplayAbility类分开保存的运行时状态.

当GameplayAbility在服务端授予时, 服务端会同步GameplayAbilitySpec到所属(Owning)客户端, 因此可以激活它.

激活GameplayAbilitySpec会根据它的实例化策略(Instancing Policy)创建一个GameplayAbility实例(Non-Instanced GameplayAbility除外).

⬆ 返回目录

4.6.11 传递数据到Ability

GameplayAbility的一般范式是Activate->Generate Data->Apply->End. 有时你需要调整现有数据, GAS提供了一些选项来获取外部数据到你的GameplayAbility.

方法	描述
通过Event激活GameplayAbility	使用含有数据负载(Payload)的Event激活GameplayAbility. 对于客户端预测的GameplayAbility, Event的负载(Payload)是由客户端同步到服务端的. 对于那些不适合任意现有变量的数据可以使用两个Optional Object或TargetData变量. 该方法的缺点是不能使用输入绑定激活Ability. 为了通过Event激活GameplayAbility, 该GameplayAbility必须设置其Trigger, 分配一个GameplayTag并选择一个GameplayEvent选项. 想要发送事件, 就得使用UAbilitySystemBlueprintLibrary::SendGameplayEventToActor(AActor* Actor, FGameplayTag EventTag, FGameplayEventData Payload)函数.
使用WaitGameplayEvent AbilityTask	在GameplayAbility激活后, 使用WaitGameplayEvent AbilityTask来告知其监听带有负载(Payload)的事件. Event负载(Payload)及其发送过程与通过Event激活GameplayAbility是一样的. 该方法的缺点是Event不能通过AbilityTask同步且只能用于Local Only和Server Only的GameplayAbility. 你可以编写自己的AbilityTask以支持同步Event负载(Payload).
使用TargetData	自定义TargetData结构体是一种在客户端和服务端之间传递任意数据的好方法.
保存数据在OwnerActor或者AvatarActor	使用保存于OwnerActor, AvatarActor或者其他任意你可以获取到引用的对象中的可同步变量. 这种方法最灵活且可以用于由输入绑定激活的GameplayAbility, 然而, 它不能保证在使用时数据同步, 你必须提前做好准备 - 这意味着如果你设置了一个可同步的变量, 之后立即激活该GameplayAbility, 那么因为存在潜在的丢包情况, 不能保证接收者上发生的顺序.

⬆ 返回目录

4.6.12 Ability花费(Cost)和冷却(Cooldown)

GameplayAbility默认带有对可选Cost和Cooldown的功能. Cost是ASC为了激活使用即刻(Instant)GameplayEffect(Cost GE)实现的GameplayAbility所必须具有的预先定义的大量Attribute. Cooldown是用于阻止GameplayAbility再次激活(直到冷却完成)的计时器, 其由持续(Duration)GameplayEffect(Cooldown GE)实现.

在GameplayAbility调用UGameplayAbility::Activate()之前, 其会调用UGameplayAbility::CanActivateAbility(), 该函数会检查所属ASC是否满足Cost(UGameplayAbility::CheckCost())并确保该GameplayAbility不在冷却期(UGameplayAbility::CheckCooldown()).

在GameplayAbility调用Activate()之后, 其可以选择性使用UGameplayAbility::CommitAbility()随时提交Cost和Cooldown, UGameplayAbility::CommitAbility()会调用UGameplayAbility::CommitCost()和UGameplayAbility::CommitCooldown(), 如果它们不需要同时提交, 设计师可以选择分别调用CommitCost()或CommitCooldown(). 提交Cost和Cooldown会多次调用CheckCost()和CheckCooldown(). 所属(Owning)ASC的Attribute在GameplayAbility激活后可能改变, 从而导致提交时无法满足Cost. 如果prediction key在提交时有效的话, 提交Cost和Cooldown是可以客户端预测的.

详见CostGE和CooldownGE.

⬆ 返回目录

4.6.13 升级Ability

有两个常见的方法用于升级Ability:

升级方法	描述
升级时取消授予和重新授予	自ASC取消授予(移除)GameplayAbility, 并在下一等级于服务端上重新授予. 如果此时GameplayAbility是激活的, 那么就会终止它.
增加GameplayAbilitySpec的等级	在服务端上找到GameplayAbilitySpec, 增加它的等级, 并将其标记为Dirty以同步到所属(Owning)客户端. 该方法不会在GameplayAbility激活时将其终止.

两种方法的主要区别在于你是否想要在升级时取消激活的GameplayAbility. 基于你的GameplayAbility, 你很可能需要同时使用两种方法. 我建议在UGameplayAbility子类中增加一个布尔值以明确使用哪种方法.

⬆ 返回目录

4.6.14 Ability集合

GameplayAbilitySet是很方便的UDataAsset类, 其用于保存输入绑定和初始GameplayAbility列表, 该GameplayAbility列表用于拥有授予GameplayAbility逻辑的Character. 子类也可以包含额外的逻辑和属性. Paragon中的每个英雄都拥有一个GameplayAbilitySet以包含其授予的所有GameplayAbility.

⬆ 返回目录

4.6.15 Ability批处理

一般GameplayAbility的生命周期最少涉及2到3个自客户端到服务端的RPC.

1. CallServerTryActivateAbility()
2. ServerSetReplicatedTargetData() (可选)
3. ServerEndAbility()

如果GameplayAbility在一帧同一原子(Atomic)组中执行这些操作, 我们就可以优化该工作流, 将所有2个或3个RPC批处理(整合)为1个RPC. GAS称这种RPC优化为Ability批处理. 常见的使用Ability批处理时机的例子是枪支命中判断. 枪支命中判断时, 会在一帧同一原子(Atomic)组中做射线检测, 发送TargetData到服务端并结束Ability. GASShooter样例项目对其枪支命中判断使用了该技术.

半自动枪支是最好的案例, 其将CallServerTryActivateAbility(), ServerSetReplicatedTargetData()(子弹命中结果)和ServerEndAbility()批处理成一个而不是三个RPC.

全自动/爆炸开火枪支会对第一发子弹将CallServerTryActivateAbility()和ServerSetReplicatedTargetData()批处理成一个而不是两个RPC, 接下来的每一发子弹都是它自己的ServerSetReplicatedTargetData()RPC, 最后, 当停止射击时, ServerEndAbility()会作为单独的RPC发送. 这是最坏的情况, 我们只保存了第一发子弹的一个RPC而不是两个. 这种情况也可以通过GameplayEvent激活Ability来实现, 该GameplayEvent会自客户端向服务端发送带有EventPayload的子弹TargetData. 后者的缺点是TargetData必须在Ability外部生成, 尽管批处理方法在Ability内部生成TargetData.

Ability批处理在ASC中是默认禁止的. 为了启用Ability批处理, 需要重写ShouldDoServerAbilityRPCBatch()以返回true:

```
virtual bool ShouldDoServerAbilityRPCBatch() const override { return true; }
```

现在Ability批处理已经启用，在激活你想使用批处理的Ability之前，必须提前创建一个FScopedServerAbilityRPCBatcher结构体，这个特殊的结构体会在其域内尝试批处理所有跟随它的Ability，一旦出了FScopedServerAbilityRPCBatcher域，所有已激活的Ability将不再尝试批处理. FScopedServerAbilityRPCBatcher通过在每个可以批处理的函数中设置特殊代码来运行，其会拦截RPC的发送并将消息打包进一个批处理结构体，当出了FScopedServerAbilityRPCBatcher后，它会在UAbilitySystemComponent::EndServerAbilityRPCBatch()中自动将该批处理结构体RPC到服务端，服务端在UAbilitySystemComponent::ServerAbilityRPCBatch_Internal(FServerAbilityRPCBatch& BatchInfo)中接收该批处理结构体，BatchInfo参数会包含几个flag，其包括该Ability是否应该结束，激活时输入是否按下和是否包含TargetData. 这是个可以设置断点以确认批处理是否正确运行的好函数. 作为选择，可以使用AbilitySystem.ServerRPCBatching.Log 1来启用特别的Ability批处理日志.

这个方法只能在C++中完成，并且只能通过FGameplayAbilitySpecHandle来激活Ability.

```
bool UGSAbilitySystemComponent::BatchRPCTryActivateAbility(FGameplayAbilitySpecHandle
InAbilityHandle, bool EndAbilityImmediately)
{
    bool AbilityActivated = false;
    if (InAbilityHandle.IsValid())
    {
        FScopedServerAbilityRPCBatcher GSAbilityRPCBatcher(this, InAbilityHandle);
        AbilityActivated = TryActivateAbility(InAbilityHandle, true);

        if (EndAbilityImmediately)
        {
            FGameplayAbilitySpec* AbilitySpec = FindAbilitySpecFromHandle(InAbilityHandle);
            if (AbilitySpec)
            {
                UGSGameplayAbility* GSAbility = Cast<UGSGameplayAbility>(AbilitySpec-
>GetPrimaryInstance());
                GSAbility->ExternalEndAbility();
            }
        }

        return AbilityActivated;
    }

    return AbilityActivated;
}
```

GASShooter对半自动和全自动枪支使用了相同的批处理GameplayAbility，并没有直接调用EndAbility()(它通过一个只能由客户端调用的Ability在该Ability外处理，该只能由客户端调用的Ability用于管理玩家输入和基于当前开火模式对批处理Ability的调用). 因为所有的RPC必须在FScopedServerAbilityRPCBatcher域中，所以我提供了EndAbilityImmediately参数以使仅客户端的控制/管理可以明确该Ability是否可以批处理EndAbility()(半自动)或不批处理EndAbility()(全自动)以及之后EndAbility()是否可以在其自己的RPC中调用.

GASShooter暴露了一个蓝图节点以允许上文提到的仅客户端调用的Ability所使用的批处理Ability来触发批处理Ability. (译者注: 此处相当拗口，但原文翻译确实如此，配合项目浏览也许会更容易明白些.)

⬆ 返回目录

4.6.16 网络安全策略(Net Security Policy)

GameplayAbility的网络安全策略决定了Ability应该在网络的何处执行. 它为尝试执行限制Ability的客户端提供了保护.

网络安全策略	描述
ClientOrServer	没有安全需求. 客户端或服务端可以自由地触发该Ability的执行和终止.
ServerOnlyExecution	客户端对该Ability请求的执行会被服务端忽略, 但客户端仍可以请求服务端取消或结束该Ability.
ServerOnlyTermination	客户端对该Ability请求的取消或结束会被服务端忽略, 但客户端仍可以请求执行该Ability.
ServerOnly	服务端控制该Ability的执行和终止, 客户端的任何请求都会被忽略.

4.7 Ability Tasks

4.7.1 AbilityTask定义

GameplayAbility只能在一帧中执行, 这本身并不能提供太多灵活性, 为了实现随时间推移而触发或响应一段时间后触发的委托操作, 我们需要使用AbilityTask.

GAS自带很多AbilityTask:

- 使用RootMotionSource移动Character的Task
- 播放动画蒙太奇的Task
- 响应Attribute变化的Task
- 响应GameplayEffect变化的Task
- 响应玩家输入的Task
- 更多

UAbilityTask的构造函数中强制硬编码允许最多1000个同时运行的AbilityTask, 当设计那些同时拥有数百个Character的游戏(像RTS)的GameplayAbility时要注意这一点.

⬆ 返回目录

4.7.2 自定义AbilityTask

通常你需要创建自己的自定义AbilityTask(C++中). 样例项目带有两个自定义AbilityTask:

1. PlayMontageAndWaitForEvent是默认PlayMontageAndWait和WaitGameplayEventAbilityTask的结合体, 其允许动画蒙太奇自AnimNotify发送GameplayEvent回到启动它的GameplayAbility, 可以使用该Task在动画蒙太奇的某个特定时刻来触发操作.
2. WaitReceiveDamage可以监听OwnerActor接收伤害. 当英雄接收到一个伤害实例时, 被动护甲层GameplayAbility就会移除一层护甲.

AbilityTask的组成:

- 创建新的AbilityTask实例的静态函数
- 当AbilityTask完成目的时分发的委托(Delegate)
- 进行主要工作的Activate()函数, 绑定到外部的委托等等

- 进行清理工作的OnDestroy()函数, 包括其绑定到外部的委托
- 所有绑定到外部委托的回调函数
- 成员变量和所有内部辅助函数

Note: AbilityTask只能声明一种类型的输出委托, 所有的输出委托都必须是该种类型, 不管它们是否使用参数. 对于未使用的委托参数会传递默认值.

AbilityTask只能运行在那些运行所属GameplayAbility的客户端或服务端, 然而, 可以通过设置bSimulatedTask = true 使 AbilityTask 运行在 Simulated Client 上, 在 AbilityTask 的构造函数中, 重写 virtual void InitSimulatedTask(UGameplayTasksComponent& InGameplayTasksComponent);并将所有成员变量设置为同步的, 这只在极少的情况下有用, 比如在移动AbilityTask中, 不想同步每次移动变化, 但是又需要模拟整个移动AbilityTask, 所有的RootMotionSource AbilityTask都是这样做的, 查看AbilityTask_MoveToLocation.h/.cpp以作为参考范例.

如果你在AbilityTask的构造函数中设置了bTickingTask = true;并重写了virtual void TickTask(float DeltaTime);, AbilityTask就可以使用Tick, 这在你需要根据帧率平滑线性插值的时候很有用. 查看AbilityTask_MoveToLocation.h/.cpp以作为参考范例.

⬆返回目录

4.7.3 使用AbilityTask

在C++中创建并激活AbilityTask(GDGA_FireGun.cpp):

```
UGDAT_PlayMontageAndWaitForEvent* Task =
UGDAT_PlayMontageAndWaitForEvent::PlayMontageAndWaitForEvent(this, NAME_None, MontageToPlay,
FGameplayTagContainer(), 1.0f, NAME_None, false, 1.0f);
Task->OnBlendOut.AddDynamic(this, &UGDGA_FireGun::OnCompleted);
Task->OnCompleted.AddDynamic(this, &UGDGA_FireGun::OnCompleted);
Task->OnInterrupted.AddDynamic(this, &UGDGA_FireGun::OnCancelled);
Task->OnCancelled.AddDynamic(this, &UGDGA_FireGun::OnCancelled);
Task->EventReceived.AddDynamic(this, &UGDGA_FireGun::EventReceived);
Task->ReadyForActivation();
```

在蓝图中, 我们只需使用为AbilityTask创建的蓝图节点, 不必调用ReadyForActivate(), 其由Engine/Source/Editor/GameplayTasksEditor/Private/K2Node_LatentGameplayTaskCall.cpp 自动调用. K2Node_LatentGameplayTaskCall也会自动调用BeginSpawningActor()和FinishSpawningActor()(如果它们存在于你的AbilityTask类中, 查看AbilityTask_WaitTargetData), 再强调一遍, K2Node_LatentGameplayTaskCall只会对蓝图做这些自动操作, 在C++中, 我们必须手动调用ReadyForActivation(), BeginSpawningActor()和FinishSpawningActor().

如果想要手动取消AbilityTask, 只需在蓝图(Async Task Proxy)或C++中对AbilityTask对象调用EndTask().

⬆返回目录

4.7.4 Root Motion Source Ability Task

GAS自带的AbilityTask可以使用挂载在CharacterMovementComponent中的Root Motion Source随时间推移而移动Character, 像击退, 复杂跳跃, 吸引和猛冲.

Note: RootMotionSource AbilityTask预测支持的引擎版本是4.19和4.25+, 该预测在引擎版本4.20-4.24中存在bug, 然而, AbilityTask仍然可以使用较小的网络修正在多人游戏中执行功能, 并且在单人游戏中完美运行. 可以将4.25中对预测的修复自定义到4.20~4.24引擎中.

⬆返回目录

4.8 Gameplay Cues

4.8.1 GameplayCue定义

GameplayCue(GC)执行非游戏逻辑相关的功能, 像音效, 粒子效果, 镜头抖动等等. GameplayCue一般是可同步(除非在客户端明确执行(Executed), 添加(Added)和移除(Removed))和可预测的.

我们可以在ASC中通过发送一个**强制带有"GameplayCue"父名**的相应GameplayTag和GameplayCueManager的事件类型(Execute, Add或Remove)来触发GameplayCue. GameplayCueNotify对象和其他实现IGameplayCueInterface的Actor可以基于GameplayCue的GameplayTag(GameplayCueTag)来订阅(Subscribe)这些事件.

Note: 再次强调, GameplayCue的GameplayTag需要以GameplayCue为开头, 举例来说, 一个有效的GameplayCue的GameplayTag可能是GameplayCue.A.B.C.

有两个GameplayCueNotify类, Static和Actor. 它们各自响应不同的事件, 并且不同的GameplayEffect类型可以触发它们. 根据你的逻辑重写相关的事件.

GameplayCue类	事件	GameplayEffect类型	描述
GameplayCueNotify_Static	Execute	Instant或Periodic	Static GameplayCueNotify直接操作ClassDefaultObject(意味着没有实例)并且对于一次性效果(像击打伤害)是极好的.
GameplayCueNotify_Actor	Add或Remove	Duration或Infinite	Actor GameplayCueNotify会在添加(Added)时生成一个新的实例, 因为其是实例化的, 所以可以随时间推移执行操作直到被移除(Removed). 这对循环的声音和粒子效果是很好的, 其会在持续(Duration)或无限(Infinite)GameplayEffect被移除或手动调用移除时移除. 其也自带选项来管理允许同时添加(Added)多少个, 因此多个相同效果的应用只启用一次声音或粒子效果.

GameplayCueNotify技术上可以响应任何事件, 但是这是我们一般使用它的方式.

Note: 当使用GameplayCueNotify_Actor时, 要勾选Auto Destroy on Remove, 否则之后对GameplayCueTag的添加(Add)调用将无法进行.

当使用除Full之外的ASC同步模式时, Add和RemoveGC事件会在服务端玩家中触发两次(Listen Server) - 一次是应用GE, 再一次是从"Minimal"NetMultiCast到客户端. 然而, WhileActive事件仍会触发一次. 所有的事件在客户端中只触发一次.

样例项目包含了一个GameplayCueNotify_Actor用于眩晕和奔跑效果. 其还含有一个GameplayCueNotify_Static用于枪支弹药伤害. 这些GC可以通过客户端触发来进行优化, 而不是通过GE同步. 我选择了在样例项目中展示使用它们的基本方法.

⬆ 返回目录

4.8.2 触发GameplayCue

GameplayEffect

在成功应用(未被Tag或Immunity阻塞)的GameplayEffect中填写所有应该触发的GameplayCue的GameplayTag.

手动调用

UGameplayAbility提供了蓝图节点来Execute, Add或Remove GameplayCue.

在C++中, 你可以在ASC中直接调用函数(或者在你的ASC子类中暴露它们到蓝图):

```
/** GameplayCues can also come on their own. These take an optional effect context to pass
through hit result, etc */
void ExecuteGameplayCue(const FGameplayTag GameplayCueTag, FGameplayEffectContextHandle
EffectContext = FGameplayEffectContextHandle());
void ExecuteGameplayCue(const FGameplayTag GameplayCueTag, const FGameplayCueParameters&
GameplayCueParameters);

/** Add a persistent gameplay cue */
void AddGameplayCue(const FGameplayTag GameplayCueTag, FGameplayEffectContextHandle
EffectContext = FGameplayEffectContextHandle());
void AddGameplayCue(const FGameplayTag GameplayCueTag, const FGameplayCueParameters&
GameplayCueParameters);

/** Remove a persistent gameplay cue */
void RemoveGameplayCue(const FGameplayTag GameplayCueTag);

/** Removes any GameplayCue added on its own, i.e. not as part of a GameplayEffect. */
void RemoveAllGameplayCues();
```

⬆ 返回目录

4.8.3 客户端GameplayCue

从GameplayAbility和ASC中暴露的用于触发GameplayCue的函数默认是可同步的. 每个GameplayCue事件都是一个多播(Multicast)RPC. 这会导致大量RPC. GAS也强制在每次网络更新中最多能有两个相同的GameplayCueRPC. 我们可以通过使用客户端GameplayCue来避免这个问题. 客户端GameplayCue只能在单独的客户端上Execute, Add或Remove.

可以使用客户端GameplayCue的场景:

- 抛射物伤害
- 近战碰撞伤害
- 动画蒙太奇触发的GameplayCue

你应该添加到ASC子类中的客户端GameplayCue函数:

```
UFUNCTION(BlueprintCallable, Category = "GameplayCue", Meta = (AutoCreateRefTerm =
"GameplayCueParameters", GameplayTagFilter = "GameplayCue"))
void ExecuteGameplayCueLocal(const FGameplayTag GameplayCueTag, const FGameplayCueParameters&
GameplayCueParameters);

UFUNCTION(BlueprintCallable, Category = "GameplayCue", Meta = (AutoCreateRefTerm =
"GameplayCueParameters", GameplayTagFilter = "GameplayCue"))
void AddGameplayCueLocal(const FGameplayTag GameplayCueTag, const FGameplayCueParameters&
GameplayCueParameters);

UFUNCTION(BlueprintCallable, Category = "GameplayCue", Meta = (AutoCreateRefTerm =
"GameplayCueParameters", GameplayTagFilter = "GameplayCue"))
void RemoveGameplayCueLocal(const FGameplayTag GameplayCueTag, const FGameplayCueParameters&
GameplayCueParameters);
```

```
void UPAAbilitySystemComponent::ExecuteGameplayCueLocal(const FGameplayTag GameplayCueTag, const
FGameplayCueParameters & GameplayCueParameters)
{
    UAbilitySystemGlobals::Get().GetGameplayCueManager()->HandleGameplayCue(GetOwner(),
GameplayCueTag, EGameplayCueEvent::Type::Executed, GameplayCueParameters);
}

void UPAAbilitySystemComponent::AddGameplayCueLocal(const FGameplayTag GameplayCueTag, const
FGameplayCueParameters & GameplayCueParameters)
{
    UAbilitySystemGlobals::Get().GetGameplayCueManager()->HandleGameplayCue(GetOwner(),
GameplayCueTag, EGameplayCueEvent::Type::OnActive, GameplayCueParameters);
    UAbilitySystemGlobals::Get().GetGameplayCueManager()->HandleGameplayCue(GetOwner(),
GameplayCueTag, EGameplayCueEvent::Type::WhileActive, GameplayCueParameters);
}

void UPAAbilitySystemComponent::RemoveGameplayCueLocal(const FGameplayTag GameplayCueTag, const
FGameplayCueParameters & GameplayCueParameters)
{
    UAbilitySystemGlobals::Get().GetGameplayCueManager()->HandleGameplayCue(GetOwner(),
GameplayCueTag, EGameplayCueEvent::Type::Removed, GameplayCueParameters);
}
```

如果某个GameplayCue是客户端添加的, 那么它也应该自客户端移除. 如果它是通过同步添加的, 那么它也应该通过同步移除.

⬆ 返回目录

4.8.4 GameplayCue参数

GameplayCue接受一个包含额外GameplayCue信息的FGameplayCueParameters结构体作为参数. 如果你在GameplayAbility或ASC中使用函数手动触发GameplayCue, 那么就必须手动填充传递给GameplayCue的GameplayCueParameters结构体. 如果GameplayCue由GameplayEffect触发, 那么下列的变量会自动填充到FGameplayCueParameters结构体中:

- AggregatedSourceTags
- AggregatedTargetTags
- GameplayEffectLevel
- AbilityLevel
- EffectContext
- Magnitude(如果GameplayEffect具有在GameplayCue标签容器(TagContainer)上方的下拉列表中选择的Magnitude Attribute和影响该Attribute的相应Modifier)

当手动触发GameplayCue时, GameplayCueParameters中的SourceObject变量似乎是一个传递任意数据到GameplayCue的好地方.

Note: 参数结构体中的某些变量, 像Instigator, 可能已经存在于EffectContext中. EffectContext也可以包含FHitResult用于存储GameplayCue在世界中生成的位置. 子类化EffectContext似乎是一个传递更多数据到GameplayCue的好方法, 特别是对于那些由GameplayEffect触发的GameplayCue.

查看UAbilitySystemGlobals中用于填充GameplayCueParameters结构体的三个函数以获得更多信息. 它们是虚函数, 因此你可以重写它们以自动填充更多信息.

```
/** Initialize GameplayCue Parameters */
virtual void InitGameplayCueParameters(FGameplayCueParameters& CueParameters, const
FGameplayEffectSpecForRPC &Spec);
virtual void InitGameplayCueParameters_GESpec(FGameplayCueParameters& CueParameters, const
FGameplayEffectSpec &Spec);
virtual void InitGameplayCueParameters(FGameplayCueParameters& CueParameters, const
FGameplayEffectContextHandle& EffectContext);
```

⬆ 返回目录

4.8.5 Gameplay Cue Manager

默认情况下, 游戏开始时GameplayCueManager会扫描游戏的全部目录以寻找GameplayCueNotify并将其加载进内存. 我们可以设置DefaultGame.ini来修改GameplayCueManager的扫描路径.

```
[/Script/GameplayAbilities.AbilitySystemGlobals]
GameplayCueNotifyPaths="/Game/GASDocumentation/Characters"
```

我们确实想要GameplayCueManager扫描并找到所有的GameplayCueNotify, 然而, 我们不想要它异步加载每一个, 因为这会将每个GameplayCueNotify和它们所引用的音效和粒子特效放入内存而不管它们是否在关卡中使用. 在像Paragon这样的大型游戏中, 内存中会放入数百兆的无用资源并造成卡顿和启动时无响应.

在启动时异步加载每个GameplayCue的一种可选方法是只异步加载那些会在游戏中触发的GameplayCue, 这会在异步加载每个GameplayCue时减少不必要的内存占用和潜在的游戏无响应几率, 从而避免特定GameplayCue在游戏中第一次触发时可能出现的延迟效果. SSD不存在这种潜在的延迟, 我还没有在HDD上测试过, 如果在UE编辑器中使用该选项并且编辑器需要编译粒子系统的话, 就可能会在GameplayCue首次加载时有轻微的卡顿或无响应, 这在构建版本中不是问题, 因为粒子系统肯定是编译好的.

首先我们必须继承UGameplayCueManager并告知AbilitySystemGlobals类在DefaultGame.ini中使用我们的UGameplayCueManager子类.

```
[/Script/GameplayAbilities.AbilitySystemGlobals]
GlobalGameplayCueManagerClass="/Script/ParagonAssets.PBGameplayCueManager"
```

在我们的UGameplayCueManager子类中, 重写ShouldAsyncLoadRuntimeObjectLibraries().

```
virtual bool ShouldAsyncLoadRuntimeObjectLibraries() const override
{
    return false;
}
```

⬆ 返回目录

4.8.6 阻止GameplayCue响应

有时我们不想响应GameplayCue, 例如我们阻止了一次攻击, 可能就不想播放附加在伤害GameplayEffect上的击打效果或者自定义的效果. 我们可以在 GameplayEffectExecutionCalculations 中调用OutExecutionOutput.MarkGameplayCuesHandledManually(), 之后手动发送我们的GameplayCue事件到Target或Source的ASC中.

如果你想某个特别指定ASC中的GameplayCue永不触发, 可以设置AbilitySystemComponent->bSuppressGameplayCues = true;.

⬆ 返回目录

4.8.7 GameplayCue批处理

每次GameplayCue触发都是一次不可靠的多播(NetMulticast)RPC. 在同一时刻触发多个GameplayCue的情况下, 有一些优化方法来将它们压缩成一个RPC或者通过发送更少的数据来节省带宽.

⬆ 返回目录

4.8.7.1 手动RPC

假设你有一个可以发射8枚弹丸的霰弹枪, 就会有8个轨迹和碰撞GameplayCue. GASShooter采用将它们联合成一个RPC的延迟(Lazy)方法, 其将所有的轨迹信息保存到EffectContext作为TargetData. 尽管其将RPC数量从8降为1, 然而还是在这一个RPC中通过网络发送大量数据(~500 bytes). 一个进一步优化的方法是使用一个自定义结构体发送RPC, 在该自定义RPC中你需要高效编码命中位置(Hit Location)或者给一个随机种子以在接收端重现/近似计算碰撞位置, 客户端之后需要解包该自定义结构体并重现客户端执行的GameplayCue.

运行机制:

1. 声明一个FScopedGameplayCueSendContext. 其会阻塞UGameplayCueManager::FlushPendingCues()直到其出域, 意味着所有GameplayCue都将会排队等候直到该FScopedGameplayCueSendContext出域.
2. 重写UGameplayCueManager::FlushPendingCues()以将那些可以基于一些自定义GameplayTag批处理的GameplayCue合并进自定义的结构体并将其RPC到客户端.
3. 客户端接收自定义结构体并将其解包进客户端执行的GameplayCue.

该方法也可以在你的GameplayCue需要特别指定的参数时使用, 这些需要特别指定的参数不能由GameplayCueParameter提供, 并且你不想将它们添加到EffectContext, 像伤害数值, 暴击标识, 破盾标识, 处决标识等等.

https://forums.unrealengine.com/development-discussion/c-gameplay-programming/1711546-fscopedgameplaycuesendcontext-gameplaycuemanager

⬆ 返回目录

4.8.7.2 GameplayEffect中的多个GameplayCue

一个GameplayEffect中的所有GameplayCue已经由一个RPC发送了. 默认情况下, UGameplayCueManager::InvokeGameplayCueAddedAndWhileActive_FromSpec()会在不可靠的多播(NetMulticast)RPC中发送整个GameplayEffectSpec(除了转换为FGameplayEffectSpecForRPC)而不管ASC的同步模式, 取决于GameplayEffectSpec的内容, 这可能会使用大量带宽, 我们可以通过设置AbilitySystem.AlwaysConvertGESpecToGCParams 1来将其优化, 这会将GameplayEffectSpec转换为FGameplayCueParameter结构体并且RPC它而不是整个FGameplayEffectSpecForRPC, 这会节省带宽但是只有较少的信息, 取决于GESpec如何转换为GameplayCueParameters和你的GameplayCue需要知道什么.

⬆ 返回目录

4.9 Ability System Globals

AbilitySystemGlobals类保存有关GAS的全局信息. 大多数变量可以在DefaultGame.ini中设置. 一般你不需要和该类互动, 但是应该知道它的存在. 如果你需要继承像GameplayCueManager或GameplayEffectContext这样的对象, 就必须通过AbilitySystemGlobals来做.

想要继承AbilitySystemGlobals, 需要在DefaultGame.ini中设置类名:

```
[/Script/GameplayAbilities.AbilitySystemGlobals]
AbilitySystemGlobalsClassName="/Script/ParagonAssets.PAAbilitySystemGlobals"
```

⬆ 返回目录

4.9.1 InitGlobalData()

从UE 4.24开始, 必须调用`UAbilitySystemGlobals::InitGlobalData()`来使用TargetData, 否则你会遇到关于`ScriptStructCache`的错误, 并且客户端会从服务端断开连接, 该函数只需要在项目中调用一次. Fortnite从AssetManager类的起始加载函数中调用该函数, Paragon是从UEngine::Init()中调用的. 我发现将其放到`UEngineSubsystem::Initialize()`是个好位置, 这也是样例项目中使用的. 我觉得你应该复制这段模板代码到你自己的项目中以避免出现`TargetData`的使用问题.

如果你在使用`AbilitySystemGlobals GlobalAttributeSetDefaultsTableNames`时发生崩溃, 可能之后需要像Fortnite一样在`AssetManager`或`GameInstance`中调用`UAbilitySystemGlobals::InitGlobalData()`而不是在`UEngineSubsystem::Initialize()`中. 该崩溃可能是由于`Subsystem`的加载顺序引发的, `GlobalAttributeDefaultsTables`需要加载`EditorSubsystem`来绑定`UAbilitySystemGlobals::InitGlobalData()`中的委托.

⬆ 返回目录

4.10 预测(Prediction)

GAS带有开箱即用的客户端预测功能, 然而, 它不能预测所有. GAS中客户端预测的意思是客户端无需等待服务端的许可而激活`GameplayAbility`和应用`GameplayEffect`. 它可以"预测"许可其可以这样做的服务端和其应用`GameplayEffect`的目标. 服务端在客户端激活之后运行`GameplayAbility`(网络延迟)并告知客户端它的预测是否正确, 如果客户端的预测出错, 那么它就会"回滚"其"错误预测"的修改以匹配服务端.

GAS相关预测的最佳源码是插件源码中的`GameplayPrediction.h`.

Epic的理念是只能预测"不受伤害(get away with)"的事情. 例如, Paragon和Fortnite不能预测伤害值, 它们很可能对伤害值使用了ExecutionCalculations, 而这无论如何是不能预测的. 这并不是说你不能试着预测像伤害值这样的事情, 无论如何, 如果你这样做了并且效果很好, 那就是极好的.

> ... we are also not all in on a "predict everything: seamlessly and automatically" solution. We still feel player prediction is best kept to a minimum (meaning: predict the minimum amount of stuff you can get away with).

来自Epic的Dave Ratti在新的网络预测插件中的注释.

什么是可预测的:

- Ability激活
- 触发事件
- GameplayEffect应用
  - Attribute修改(例外: 执行(Execution)目前无法预测, 只能是Attribute Modifiy)
  - GameplayTag修改
- GameplayCue事件(可预测GameplayEffect中的和它们自己)
- 蒙太奇
- 移动(内建在UE4 UCharacterMovement中)

什么是不可预测的:

- GameplayEffect移除
- GameplayEffect周期效果(dots ticking)

源自GameplayPrediction.h

尽管我们可以预测GameplayEffect的应用, 但是不能预测GameplayEffect的移除. 绕过这条限制的一个方法是当想要移除GameplayEffect时, 可以预测性地执行相反的效果, 假设我们要降低40%移动速度, 可以通过应用增加40%移动速度的buff来预测性地将其移除, 之后同时移除这两个GameplayEffect. 这并不是对每个场景都适用, 因此仍然需要对预测性地移除GameplayEffect的支持. 来自Epic的Dave Ratti已经表达过在GAS的迭代版本中增加它的期望.

因为我们不能预测GameplayEffect的移除, 所以就不能完全预测GameplayAbility的冷却时间, 并且也没有相反的GameplayEffect这种变通方案可供使用. 服务端同步的Cooldown GE将会存于客户端上, 并且任何对其绕过的尝试(例如使用Minimal同步模式)都会被服务端拒绝. 这意味着高延迟的客户端会花费较长事件来告知服务端开始冷却和接收到服务端Cooldown GE的移除. 这意味着高延迟的玩家会比低延迟的玩家有更低的触发率, 从而劣势于低延迟玩家. Fortnite通过使用自定义Bookkeeping而不是Cooldown GE的方法避免了该问题.

关于预测伤害值, 我个人不建议使用, 尽管它是大多数刚接触GAS的人最先做的事情之一, 我特别不建议尝试预测死亡, 虽然你可以预测伤害值, 但是这样做很棘手. 如果你错误预测地应用了伤害, 那么玩家将会看到敌人的生命值会跳动, 如果你尝试预测死亡, 那么这将会是特别尴尬和令人沮丧的, 假设你错误预测了某个Character的死亡, 那么它就会开启布娃娃模拟, 只有当服务端纠正后才会停止布娃娃模拟并继续向你射击.

Note: 修改Attribute的即刻(Instant)GameplayEffect(像Cost GE)在你自身可以无缝预测, 预测修改其他Character的即刻(Instant)Attribute会显示短暂的异常或者Attribute中的暂时性问题. 预测的即刻(Instant)GameplayEffect实际上被视为无限(Infinite)GameplayEffect, 因此如果错误预测的话还可以回滚. 当服务端的GameplayEffect被应用时, 其实是存在两个相同的GameplayEffect的, 会在短时间内造成Modifier应用两次或者不应用, 其最终会纠正自身, 但是有时候这个异常现象对玩家来说是显著的.

GAS的预测实现尝试解决的问题:

> 1. "我能这样做吗?"(Can I do this?) 预测的基本协议.
> 2. "撤销"(Undo) 当预测错误时如何消除其副作用.
> 3. "重现"(Redo) 如何避免已经在客户端预测但还是从服务端同步的重播副作用.
> 4. "完整"(Completeness) 如何确认我们真的预测了所有副作用.
> 5. "依赖"(Dependencies) 如何管理依赖预测和预测事件链.
> 6. "重写"(Override) 如何预测地重写由服务端同步/拥有的状态.

源自GameplayPrediction.h

↑ 返回目录

4.10.1 Prediction Key

GAS的预测建立在Prediction Key的概念上, 其是一个由客户端激活GameplayAbility时生成的整型标识符.

- 客户端激活GameplayAbility时生成Prediction Key, 这是Activation Prediction Key.
- 客户端使用CallServerTryActivateAbility()将该Prediction Key发送到服务端.
- 客户端在Prediction Key有效时将其添加到应用的所有GameplayEffect.
- 客户端的Prediction Key出域. 之后该GameplayAbility中的预测效果(Effect)需要一个新的Scoped Prediction Window.
- 服务端从客户端接收Prediction Key.
- 服务端将Prediction Key添加到其应用的所有GameplayEffect.
- 服务端同步该Prediction Key回客户端.
- 客户端使用Prediction Key从服务端接收同步的GameplayEffect, 该Prediction Key用于应用GameplayEffect. 如果任何同步的GameplayEffect与客户端使用相同Prediction Key应用的GameplayEffect相匹配, 那么其就是正确预测的. 目标上暂时会有GameplayEffect的两份拷贝直到客户端移除它预测的那一个.

- 客户端从服务端上接收回Prediction Key, 这是同步的Prediction Key, 该Prediction Key现在被标记为陈旧(Stale).
- 客户端移除所有由同步的陈旧(Stale)Prediction Key创建的GameplayEffect. 由服务端同步的GameplayEffect会持续存在. 任何客户端添加的和没有从服务端接收到一个匹配的同步版本的都被视为错误预测.

在源于Activation Prediction Key激活的GameplayAbility中的一个instruction "window"原子(Atomic)组期间, Prediction Key是保证有效的, 你可以理解为Prediction Key只在一帧期间是有效的. 任何潜在行为AbilityTask的回调函数将不再拥有一个有效的Prediction Key, 除非该AbilityTask有内建的可以生成新Scoped Prediction Window的同步点(Sync Point).

⬆ 返回目录

4.10.2 在Ability中创建新的预测窗口(Prediction Window)

为了在AbilityTask的回调函数中预测更多的行为, 我们需要使用新的Scoped Prediction Key创建Scoped Prediction Window, 有时这被视为客户端和服务端间的同步点(Sync Point). 一些AbilityTask, 像所有输入相关的AbilityTask, 带有创建新Scoped Prediction Window的内建功能, 意味着AbilityTask回调函数中的原子(Atomic)代码有一个有效的Scoped Prediction Key可供使用. 像WaitDelay的其他Task没有创建新Scoped Prediction Window以用于回调函数的内建代码, 如果你需要在WaitDelay这样的AbilityTask后预测行为, 就必须使用OnlyServerWait选项的WaitNetSync AbilityTask手动来做, 当客户端触发OnlyServerWait选项的WaitNetSync时, 它会生成一个新的基于GameplayAbility的Activation Prediction Key的Scoped Prediction Key, RPC其到服务端, 并将其添加到所有新应用的GameplayEffect. 当服务端触发OnlyServerWait选项的WaitNetSync时, 它会在继续前等待直到接收到客户端新的Scoped Prediction Key, 该Scoped Prediction Key会执行和Activation Prediction Key同样的操作 —— 应用到GameplayEffect并同步回客户端标记为陈旧(Stale). Scoped Prediction Key直到出域前都有效, 也就表示Scoped Prediction Window已经关闭了. 所以只有原子(Atomic)操作, nothing latent, 可以使用新的Scoped Prediction Key.

你可以根据需求创建任意数量的Scoped Prediction Window.

如果你想添加同步点(Sync Point)功能到自己的自定义AbilityTask, 请查看那些输入AbilityTask是如何从根本上注入WaitNetSync AbilityTask代码到自身的.

Note: 当使用WaitNetSync时, 会阻塞服务端GameplayAbility继续执行直到其接收到客户端的消息. 这可能会被破解游戏的恶意用户滥用以故意延迟发送新的Scoped Prediction Key, 尽管Epic很少使用WaitNetSync, 但如果你对此担忧的话, 其建议创建一个带有延迟的新AbilityTask, 它会自动继续运行而无需等待客户端消息.

样例项目在奔跑GameplayAbility中使用了WaitNetSync以在每次应用耐力花费时创建新的Scoped Prediction Window, 这样我们就可以进行预测. 理想上当应用花费和冷却时间时我们就想要一个有效的Prediction Key.

如果你有一个在所属(Owning)客户端执行两次的预测GameplayEffect, 那么你的Prediction Key就是陈旧(Stall)的, 并且正在经历"redo"问题. 你通常可以在应用GameplayEffect之前将OnlyServerWait选项的WaitNetSync AbilityTask放到正确的位置以创建新的Scoped Prediction Key来解决这个问题.

⬆ 返回目录

4.10.3 预测性地生成Actor

在客户端预测性地生成Actor是一项高级技术. GAS对此没有提供开箱即用的功能(SpawnActor AbilityTask只在服务端生成Actor). 其关键是在客户端和服务端都生成同步的Actor.

如果Actor只是用于场景装饰或者不服务于任何游戏逻辑, 简单的解决方案就是重写Actor的IsNetRelevantFor()函数以限制服务端同步到所属(Owning)客户端, 所属(Owning)客户端会拥有其本地生成的版本, 而服务端和其他客户端会拥有服务端同步的版本.

```
bool APAReplicatedActorExceptOwner::IsNetRelevantFor(const AActor * RealViewer, const AActor *
ViewTarget, const FVector & SrcLocation) const
{
    return !IsOwnedBy(ViewTarget);
}
```

如果生成的Actor影响了游戏逻辑, 像投掷物就需要预测伤害值, 那么你需要本文档范围之外的高级知识, 在Epic Games的Github上查看UnrealTournament是如何生成投掷物的, 它有一个只在所属(Owning)客户端生成且与服务端同步的投掷物.

⬆返回目录

4.10.4 GAS中预测的未来

`GameplayPrediction.h`说明了在未来可能会增加预测`GameplayEffect`移除和周期`GameplayEffect`的功能.

来自Epic的Dave Ratti已经表达过对其修复的兴趣, 包括预测冷却时间时的延迟问题和高延迟玩家对低延迟玩家的劣势.

来自Epic之手的新网络预测插件(Network Prediction plugin)期望能与GAS充分交互, 就像在次之前的`CharacterMovementComponent`.

⬆返回目录

4.10.5 网络预测插件(Network Prediction plugin)

Epic最近发起了一项倡议, 将使用新的网络预测插件替换`CharacterMovementComponent`, 该插件仍处于起步阶段, 但是在Unreal Engine Github上已经可以访问了, 现在说未来哪个引擎版本将首次搭载其试验版还为时尚早.

⬆返回目录

4.11 Targeting

4.11.1 Target Data

FGameplayAbilityTargetData是用于通过网络传输定位数据的通用结构体. `TargetData`一般用于保存AActor/UObject引用, FHitResult和其他通用的Location/Direction/Origin信息. 然而, 本质上你可以继承它以增添想要的任何数据, 其可以简单理解为在客户端和服务端的`GameplayAbility`中传递数据. 基础结构体`FGameplayAbilityTargetData`不能直接使用, 而是要继承它. GAS的`GameplayAbilityTargetTypes.h`中有一些开箱即用的派生`FGameplayAbilityTargetData`结构体.

`TargetData`一般由Target Actor或者手动创建, 供AbilityTask使用, 或者GameplayEffect通过EffectContext使用. 因为其位于`EffectContext`中, 所以Execution, MMC, GameplayCue和AttributeSet的后端函数可以访问该`TargetData`.

我们一般不直接传递`FGameplayAbilityTargetData`而是使用FGameplayAbilityTargetDataHandle, 其包含一个`FGameplayAbilityTargetData`指针类型的TArray, 这个中间结构体可以为`TargetData`的多态性提供支持.

⬆返回目录

4.11.2 Target Actor

GameplayAbility使用WaitTargetData AbilityTask生成TargetActor以在世界中可视化和获取定位信息. TargetActor可以选择使用GameplayAbilityWorldReticles显示当前目标. 经确认后, 定位信息将作为TargetData返回, 之后其可以传递给GameplayEffect.

TargetActor是基于AActor的, 因此它可以使用任意种类的可视组件来表示其在何处和如何定位的, 例如静态网格物(Static Mesh)和贴花(Decal). 静态网格物(Static Mesh)可以用来可视化角色将要建造的物体, 贴花(Decal)可以用来表现地面上的效果区域. 样例项目使用带有地面贴花的AGameplayAbilityTargetActor_GroundTrace表示陨石技能的伤害区域效果. 它也可以什么都不显示, 例如, GASShooter中的枪支命中判断, 要对目标的射线检测显示些什么就是没有意义的.

其使用基本的射线或者Overlap捕获定位信息, 并根据TargetActor的实现将FHitResult或AActor数组转换为TargetData. WaitTargetData AbilityTask 通过 TEnumAsByte<EGameplayTargetingConfirmation::Type> ConfirmationType 参数决定目标何时被确认, 当不使用TEnumAsByte<EGameplayTargetingConfirmation::Type::Instant 时, TargetActor 一般就在 Tick() 中执行射线/Overlap, 并根据它的实现来更新位置信息到FHitResult. 尽管是在Tick()中执行的射线/Overlap, 但是一般不用担心, 因为它是不同步的并且一般没有多个(尽管存在多个)TargetActor同时运行, 只是要留意它使用的是Tick(), 一些复杂的TargetActor可能会在其中做很多事情, 就像GASShooter中火箭筒的二技能. 当Tick()中的检测对客户端响应非常频繁时, 如果性能影响很大的话, 你可能就要考虑降低 TargetActor 的 Tick 频率. 对于TEnumAsByte<EGameplayTargetingConfirmation::Type::Instant, TargetActor会立即生成, 产生TargetData, 然后销毁, 并且从不会调用Tick().

EGameplayTargetingConfirmation::Type	何时确认Target
Instant	该定位无需特殊逻辑即可立即进行, 或者用户输入决定何时开始.
UserConfirmed	当Ability绑定到Confirm输入且用户确认或调用UAbilitySystemComponent::TargetConfirm()时触发该定位. TargetActor也会响应绑定的Cancel输入或者调用UAbilitySystemComponent::TargetCancel()来取消定位.
Custom	GameplayTargeting Ability负责调用UGameplayAbility::ConfirmTaskByInstanceName()来决定何时准备好定位数据. TargetActor也可以响应UGameplayAbility::CancelTaskByInstanceName()来取消定位.
CustomMulti	GameplayTargeting Ability负责调用UGameplayAbility::ConfirmTaskByInstanceName()来决定何时准备好定位数据. TargetActor也可以响应UGameplayAbility::CancelTaskByInstanceName()来取消定位. 不应在数据生成后就结束AbilityTask, 因为其允许多次确认.

并不是所有的 TargetActor 都支持每个 EGameplayTargetingConfirmation::Type, 例如, AGameplayAbilityTargetActor_GroundTrace就不支持Instant确认.

WaitTargetData AbilityTask将一个AGameplayAbilityTargetActor类作为参数, 其会在每次AbilityTask激活时生成一个实例并且在AbilityTask结束时销毁该TargetActor. WaitTargetDataUsingActor AbilityTask接受一个已经生成的TargetActor, 但是在该AbilityTask结束时仍会销毁它. 这两种AbilityTask都是低效的, 因为它们在每次使用时都要生成或需要一个新生成的TargetActor, 它们用于原型开发是很好的, 但是在实际发布版本中, 如果有持续产生TargetData 的场景, 像全自动开火, 你可能就要探索优化它的办法. GASShooter 有一个自定义的AGameplayAbilityTargetActor子类和一个完全重写的WaitTargetDataWithReusableActor AbilityTask, 其允许你复用TargetActor而无需将其销毁.

TargetActor默认是不可同步的. 然而, 如果在你的游戏中向其他玩家展示本地玩家正在定位的目标是有意义的, 那么它也可以被设计成可同步的, WaitTargetData AbilityTask也确实包含其通过RPC和服务端通信的默认功能. 如果TargetActor 的 ShouldProduceTargetDataOnServer 属性设置为 false, 那么客户端就会在确认时通过UAbilityTask_WaitTargetData::OnTargetDataReadyCallback()中的CallServerSetReplicatedTargetData()RPC它

的TargetData到服务端. 如果ShouldProduceTargetDataOnServer为true, 那么客户端就会发送一个通用确认事件, EAbilityGenericReplicatedEvent::GenericConfirm, 在UAbilityTask_WaitTargetData::OnTargetDataReadyCallback()中RPC到服务端, 服务端在接收到RPC时就会执行射线或者Overlap检测以生成数据. 如果客户端取消该定位, 它会发送一个通用取消事件, EAbilityGenericReplicatedEvent::GenericCancel, 在UAbilityTask_WaitTargetData::OnTargetDataCancelledCallback中RPC到服务端. 你可以看到, 在TargetActor和WaitTargetData AbilityTask中存在大量委托, TargetActor响应输入来产生广播TargetData的准备, 确认或者取消委托, WaitTargetData监听TargetActor的TargetData的准备, 确认和取消委托, 并将该信息转发回GameplayAbility和服务端. 如果是向服务端发送TargetData, 为了防止作弊, 可能需要在服务端做校验以保证该TargetData是合理的. 直接在服务端上产生TargetData可以完全避免这个问题, 但是可能会导致所属(Owning)客户端的错误预测.

根据使用的不同AGameplayAbilityTargetActor子类, WaitTargetData AbilityTask节点会暴露不同的ExposeOnSpawn参数, 一些常用的参数包括:

常用TargetActor参数	定义
Debug	如果为真, 每当非发行版本中的TargetActor执行射线检测时, 其会绘制debug射线/Overlap信息. 请记住, non-Instant TargetActor会在Tick()中执行射线检测, 因此这些debug绘制调用也会在Tick()中触发.
Filter	[可选]当射线/Overlap触发时, 用于从Target中过滤(移除)Actor的特殊结构体. 典型的使用案例是过滤玩家的Pawn, 其要求Target是特殊类. 查看Target Data Filters以获得更多高级使用案例.
Reticle Class	[可选]TargetActor生成的AGameplayAbilityWorldReticle子类.
Reticle Parameters	[可选]配置你的Reticle. 查看Reticles.
Start Location	用于设置射线检测应该从何处开始的特殊结构体. 一般这应该是玩家的视口(Viewport), 枪口或者Pawn的位置.

使用默认的TargetActor类时, Actor只有直接在射线/Overlap中时才是有效的, 如果它离开射线/Overlap(它移动开或你的视线转向别处), 就不再有效了. 如果你想TargetActor记住最后有效的Target, 就需要添加这项功能到一个自定义的TargetActor类. 我称之为持久化Target, 因为其会持续存在直到TargetActor接收到确认或取消消息, TargetActor会在它的射线/Overlap中找到一个新的有效Target, 或者Target不再有效(已销毁). GASShooter对火箭筒二技能的制导火箭定位使用了持久化Target.

⬆返回目录

4.11.3 TargetData过滤器

同时使用Make GameplayTargetDataFilter和Make Filter Handle节点, 你可以过滤玩家的Pawn或者只选择某个特定类. 如果需要更多高级过滤条件, 可以继承FGameplayTargetDataFilter并重写FilterPassesForActor函数.

```
USTRUCT(BlueprintType)
struct GASDOCUMENTATION_API FGDNameTargetDataFilter : public FGameplayTargetDataFilter
{
    GENERATED_BODY()

    /** Returns true if the actor passes the filter and will be targeted */
    virtual bool FilterPassesForActor(const AActor* ActorToBeFiltered) const override;
};
However, this will not work directly into the Wait Target Data node as it requires a
FGameplayTargetDataFilterHandle. A new custom Make Filter Handle must be made to accept the
subclass:
```

```
FGameplayTargetDataFilterHandle
UGDTargetDataFilterBlueprintLibrary::MakeGDNameFilterHandle(FGDNameTargetDataFilter Filter,
AActor* FilterActor)
{
    FGameplayTargetDataFilter* NewFilter = new FGDNameTargetDataFilter(Filter);
    NewFilter->InitializeFilterContext(FilterActor);

    FGameplayTargetDataFilterHandle FilterHandle;
    FilterHandle.Filter = TSharedPtr<FGameplayTargetDataFilter>(NewFilter);
    return FilterHandle;
}
```

⬆ 返回目录

4.11.4 Gameplay Ability World Reticles

当使用已确认的non-Instant TargetActor定位时, AGameplayAbilityWorldReticles(Reticles)可以可视化正在定位的目标. TargetActor负责所有Reticle生命周期的生成和销毁. Reticle是AActor, 因此其可以使用任意种类的可视组件作为表现形式. GASShooter中常见的一种实现方式是使用WidgetComponent在屏幕空间中显示UMG Widget(永远面对玩家的摄像机). Reticle不知道其正在定位的Actor, 但是你可以通过继承在自定义TargetActor中实现该功能. TargetActor一般在每次Tick()中更新Reticle的位置为Target的位置.

GASShooter对火箭筒二技能制导火箭锁定的目标使用了Reticle. 敌人身上的红色标识就是Reticle, 相似的白色图像是火箭筒的准星.

Reticle带有一些面向设计师的BlueprintImplementableEvents(它们被设计用来在蓝图中开发):

```
/** Called whenever bIsTargetValid changes value. */
UFUNCTION(BlueprintImplementableEvent, Category = Reticle)
void OnValidTargetChanged(bool bNewValue);

/** Called whenever bIsTargetAnActor changes value. */
UFUNCTION(BlueprintImplementableEvent, Category = Reticle)
void OnTargetingAnActor(bool bNewValue);

UFUNCTION(BlueprintImplementableEvent, Category = Reticle)
void OnParametersInitialized();

UFUNCTION(BlueprintImplementableEvent, Category = Reticle)
void SetReticleMaterialParamFloat(FName ParamName, float value);

UFUNCTION(BlueprintImplementableEvent, Category = Reticle)
void SetReticleMaterialParamVector(FName ParamName, FVector value);
```

Reticle可以选择使用TargetActor提供的FWorldReticleParameters来配置, 默认结构体只提供一个变量FVector AOEScale, 尽管你可以在技术上继承该结构体, 但是TargetActor只接受基类结构体, 不允许在默认TargetActor中子类化该结构体似乎有些短见, 然而, 如果你创建了自己的自定义TargetActor, 就可以提供自定义的Reticle参数结构体并在生成Reticle时手动传递它到AGameplayAbilityWorldReticles子类.

Reticle默认是不可同步的, 但是如果在你的游戏中向其他玩家展示本地玩家正在定位的目标是有意义的, 那么它也可以被设计成可同步的.

Reticle只会显示在默认TargetActor的当前有效Target上，例如，如果你正在使用AGameplayAbilityTargetActor_SingleLineTrace对目标执行射线检测，敌人只有直接处于射线路径上时Reticle才会显示，如果角色视线看向别处，那么该敌人就不再是一个有效Target，因此该Reticle就会消失. 如果你想Reticle保留在最后一个有效Target上，就需要自定义TargetActor来记住最后一个有效Target并使Reticle保留在其上. 我称之为持久化Target，因为其会持续存在直到TargetActor接收到确认或取消消息，TargetActor会在它的射线/Overlap中找到一个新的有效Target，或者Target不再有效(已销毁). GASShooter对火箭筒二技能的制导火箭定位使用了持久化Target.

⬆ 返回目录

4.11.5 Gameplay Effect Containers Targeting

GameplayEffectContainer提供了一个可选的产生TargetData的高效方法. 当EffectContainer在客户端和服务端上应用时，该定位会立即进行，它比TargetActor更有效率，因为它是运行在定位对象的CDO(Class Default Object)上的(没有Actor的生成和销毁)，但是它不支持用户输入，无需确认即可立即进行，不能取消，并且不能从客户端向服务端发送数据(在两者上产生数据)，它对即时射线检测和碰撞Overlap很有用. Epic的Action RPG Sample Project包含两种使用Container定位的样例 —— 定位Ability拥有者和从事件拉取TargetData，它还在蓝图中实现了在距玩家某个偏移处(由蓝图子类设置)做球形射线检测(Sphere Trace)，你可以在C++或蓝图中继承URPGTargetType以实现自己的定位类型.

⬆ 返回目录

5. 常用的Abilty和Effect

5.1 眩晕(Stun)

一般在眩晕状态时，我们就要取消Character所有已激活的GameplayAbility，阻止新的GameplayAbility激活，并在整个眩晕期间阻止移动. 样例项目的陨石GameplayAbility会在击中的目标上应用眩晕效果.

为了取消目标活跃的GameplayAbility，可以在眩晕GameplayTag添加时调用AbilitySystemComponent->CancelAbilities().

为了在眩晕时阻止新的GameplayAbility激活，可以在GameplayAbility的Activation Blocked Tags GameplayTagContainer中添加眩晕GameplayTag.

为了在眩晕时阻止移动，我们可以在拥有者拥有眩晕GameplayTag时重写CharacterMovementComponent的GetMaxSpeed()函数以返回0.

⬆ 返回目录

5.2 奔跑(Sprint)

样例项目提供了一个如何奔跑的例子 —— 按住左Shift时跑得更快.

更快的移动由CharacterMovementComponent通过网络发送flag到服务端预测性地处理. 详见GDCharacterMovementComponent.h/cpp.

GA处理响应左Shift输入，告知CMC开始和停止奔跑，并且在左Shift按下时预测性地消耗耐力. 详见GA_Sprint_BP.

⬆ 返回目录

5.3 瞄准(Aim Down Sight)

样例项目处理它和奔跑时完全一样, 除了降低移动速度而不是提高它.

详见GDCharacterMovementComponent.h/cpp是如何预测性地降低移动速度的.

详见GA_AimDownSight_BP是如何处理输入的. 瞄准时是不消耗耐力的.

⬆ 返回目录

5.4 生命偷取(Lifesteal)

我在伤害ExecutionCalculation中处理生命偷取. GameplayEffect会有一个像Effect.CanLifesteal样的GameplayTag, ExecutionCalculation会检查GameplayEffectSpec是否有Effect.CanLifesteal GameplayTag, 如果该GameplayTag存在, ExecutionCalculation会使用作为Modifer的生命值创建一个动态的即刻(Instant)GameplayEffect, 并将其应用回源(Source)ASC.

```
if (SpecAssetTags.HasTag(FGameplayTag::RequestGameplayTag(FName("Effect.Damage.CanLifesteal"))))
{
    float Lifesteal = Damage * LifestealPercent;

    UGameplayEffect* GELifesteal = NewObject<UGameplayEffect>(GetTransientPackage(),
FName(TEXT("Lifesteal")));
    GELifesteal->DurationPolicy = EGameplayEffectDurationType::Instant;

    int32 Idx = GELifesteal->Modifiers.Num();
    GELifesteal->Modifiers.SetNum(Idx + 1);
    FGameplayModifierInfo& Info = GELifesteal->Modifiers[Idx];
    Info.ModifierMagnitude = FScalableFloat(Lifesteal);
    Info.ModifierOp = EGameplayModOp::Additive;
    Info.Attribute = UPAAttributeSetBase::GetHealthAttribute();

    SourceAbilitySystemComponent->ApplyGameplayEffectToSelf(GELifesteal, 1.0f,
SourceAbilitySystemComponent->MakeEffectContext());
}
```

⬆ 返回目录

5.5 在客户端和服务端中生成随机数

有时你需要在GameplayAbility中生成随机数用于枪支后坐力或者子弹扩散, 客户端和服务端都需要生成相同的随机数, 要做到这一点, 我们必须在GameplayAbility激活时设置相同的随机数种子, 你需要在每次激活GameplayAbility时设置随机数种子, 以防客户端错误预测激活或者它的随机数列表与服务端的不同步.

随机种子设置方法	描述

随机种子设置方法	描述
使用Activation Prediction Key	GameplayAbility的Activation Prediction Key是一个确保同步并且在客户端和服务端的Activation()中都可用的int16类型值. 你可以在客户端和服务端中设置其作为随机数种子. 该方法的缺点是每次游戏开始时Prediction Key总是从0开始并且会在生成key之后持续增加, 这意味着每场游戏都有着及其相同的随机数序列, 这可能满足或不满足你的随机数需要.
激活GameplayAbility时通过事件负载(Event Payload)发送种子	通过事件激活GameplayAbility并通过可同步的事件负载(Event Payload)从客户端发送随机生成的种子到服务端, 这允许更高的随机性, 但是客户端也容易被破解而每次只发送相同的种子. 通过事件激活GameplayAbility也会阻止其从用户绑定激活.

如果你的随机偏差很小, 大多数玩家是不会注意到每次游戏的随机序列都是相同的, 那么使用Activation Prediction Key作为随机种子就应该适用于你. 如果你正在做一些更复杂的事, 需要防范破解者, 那么使用Server Initiated GameplayAbility会更好, 服务端可以创建Prediction Key或者生成随机数种子来通过事件负载(Event Payload)发送.

⬆ 返回目录

5.6 暴击(Critical Hits)

我在伤害ExecutionCalculation中处理暴击. GameplayEffect会有一个像Effect.CanCrit样的GameplayTag, ExecutionCalculation会检查GameplayEffectSpec是否有Effect.CanCrit GameplayTag, 如果该GameplayTag存在, ExecutionCalculation就会生成一个与暴击率相关联的随机数(从Source捕获的Attribute), 如果成功的话就会增加暴击伤害(另一个从Source捕获的Attribute). 因为我没有预测伤害, 所以不必担心在客户端和服务端上同步随机数生成器的问题(ExecutionCalculation只运行在服务端上). 如果你尝试使用MMC预测性地执行伤害计算, 就必须从GameplayEffectSpec->GameplayEffectContext->GameplayAbilityInstance中获取随机数种子的引用.

查看GASShooter是如何处理爆头问题的, 其概念是一样的, 除了不再依赖随机数作为概率而是检测FHitResult骨骼名.

⬆ 返回目录

5.7 非堆栈GameplayEffect, 但是只有其最高级(Greatest Magnitude)才能实际影响Target

Paragon中的Slow Effect是非堆栈的. 应用每个实例并且像平常一样跟踪其生命周期, 但是只有最高级(Greatest Magnitude)的Slow Effect才能实际影响Character. GAS为这种场景提供了开箱即用的AggregatorEvaluateMetaData, 详见AggregatorEvaluateMetaData()及其实现.

⬆ 返回目录

5.8 游戏暂停时生成TargetData

如果你需要在等待玩家从`WaitTargetData AbilityTask`生成TargetData时暂停游戏, 我建议使用`slomo 0`而不是暂停.

↑ 返回目录

5.9 按钮交互系统(Button Interaction System)

GASShooter实现了一个按钮交互系统, 玩家可以按下或按住'E'键来和可交互对象交互, 像复活玩家, 打开武器箱, 打开或关闭滑动门.

↑ 返回目录

6. 调试GAS

通常在调试GAS相关的问题时, 你感兴趣的事情像:

> "我的Attribute的值是多少?"
> "我有哪些GameplayTag?"
> "我现在有哪些GameplayEffect?"
> "我已经授予的Ability有哪些, 哪个正在运行, 哪个被堵止激活?"

GAS有两种技术可以在运行时解决这些问题 —— showdebug abilitysystem和在GameplayDebugger中Hook.

Tip: UE4倾向于优化C++代码, 这使得某些函数变得很难调试, 当深入追踪代码时很少遇到这种情况. 如果将Visual Studio的解决方案配置设置为`DebugGame Editor`仍然不能追踪代码或者监视变量, 可以使用`PRAGMA_DISABLE_OPTIMIZATION_ACTUAL`和`PRAGMA_ENABLE_OPTIMIZATION_ACTUAL`宏包裹优化函数来关闭优化, 这不能在插件代码中使用除非从源码重新编译插件. 这可以或不可以用于inline函数, 取决于它的作用和位置. 确保完成调试后移除这两个宏!

```
PRAGMA_DISABLE_OPTIMIZATION_ACTUAL
void MyClass::MyFunction(int32 MyIntParameter)
{
    // My code
}
PRAGMA_ENABLE_OPTIMIZATION_ACTUAL
```

↑ 返回目录

6.1 showdebug abilitysystem

在游戏中的控制台输入`showdebug abilitysystem`. 该特性被分为三"页", 三页都会显示当前拥有的`GameplayTag`, 在控制台输入`AbilitySystem.Debug.NextCategory`来换页.

第一页显示了所有`Attribute`的`CurrentValue`:

第二页显示了所有应用到你的持续(`Duration`)和无限(`Infinite`)`GameplayEffect`, 它们的堆栈数, 使用的`GameplayTag`和`Modifier`.

第三页显示了所有授予到你的GameplayAbility, 无论其是否正在运行, 无论其是否被阻止激活, 和当前正在运行的AbilityTask的状态.

你可以使用PageUp和PageDown切换Target, 页面只显示你本地控制的Character中的ASC数据, 然而, 使用AbilitySystem.Debug.NextTarget和AbilitySystem.Debug.PrevTarget可以显示其他ASC的数据, 但是不会显示调试信息的上半部分, 也不会更新绿色目标长方体, 因此无法知道当前定位的是哪个ASC, 该BUG已经被提交到https://issues.unrealengine.com/issue/UE-90437..

Note: 为了showdebug abilitysystem可以使用, 必须在GameMode中选择一个实际的HUD类, 否则就会找不到该命令并返回"Unknown Command".

⬆ 返回目录

6.2 Gameplay Debugger

GAS向Gameplay Debugger中添加了功能, 使用反引号(`)键以访问Gameplay Debugger. 按下小键盘的3键以启用Ability分类, 取决于你所拥有的插件, 分类可能是不同的. 如果你的键盘没有小键盘, 比如笔记本, 那么你可以在项目设置(Project Settings)里修改键盘绑定.

当你想要查看其他Character的GameplayTag, GameplayEffect和GameplayAbility时可以使用Gameplay Debugger, 可惜的是它不能显示Target的Attribute中的CurrentValue. 它会定位屏幕中央的任何Character, 你可以通过选择编辑器世界大纲(World Outliner)或者看向另一个不同的Character并再次按下反引号(`)键来修改Target. 当前监视的Character上方有最大的红色圆.

⬆ 返回目录

6.3 GAS日志(Logging)

GAS包含了大量不同详细级别的日志生成语句, 你很可能见到的是ABILITY_LOG()这样的语句. 默认的详细级别是Display, 更高的默认不会显示在控制台里.

为了修改某个日志分类的详细级别, 在控制台中输入:

```
log [category] [verbosity]
```

例如, 为了开启ABILITY_LOG()语句, 应该在控制台中输入:

```
log LogAbilitySystem VeryVerbose
```

为了恢复默认, 输入:

```
log LogAbilitySystem Display
```

为了显示所有的日志分类, 输入:

```
log list
```

值得注意的GAS相关的日志分类:

日志分类	默认详细级别
LogAbilitySystem	Display
LogAbilitySystemComponent	Log
LogGameplayCueDetails	Log
LogGameplayCueTranslator	Display
LogGameplayEffectDetails	Log
LogGameplayEffects	Display
LogGameplayTags	Log
LogGameplayTasks	Log
VLogAbilitySystem	Display

详情查看日志Wiki.

⬆ 返回目录

7. 优化

7.1 Ability批处理

GameplayAbility在一帧中的的激活、选择性地向服务器发送`TargetData`、结束可以被批处理而将2-3个RPC压缩成一个RPC. 这种RPC常用于枪支的命中判断.

⬆ 返回目录

7.2 GameplayCue批处理

如果你需要同时发送很多GameplayCue, 可以考虑将其批处理成一个RPC, 目标就是减少RPC数量(`GameplayCue`是不可靠的网络多播(NetMulticast))并以尽可能少的数据量发送.

⬆ 返回目录

7.3 AbilitySystemComponent同步模式(Replication Mode)

默认情况下, ASC处于Full Replication模式, 这会同步所有的GameplayEffect到每个客户端(对单人游戏来说很好). 在多人游戏中, 设置玩家拥有的`ASC`为`Mixed Replication`模式, AI控制的Character为`Minimal Replication`模式, 这会将应用到玩家Character上的`GE`仅同步到该Character的拥有者, 应用到AI控制的Character上的`GE`永远不会同步到客户端. 无论是什么同步模式, GameplayTag仍会进行同步, GameplayCue仍会以不可靠的网络多播(NetMulticast)到所有客户端. 当所有客户端都不需要查看这些数据时, 这会减少从`GE`同步的网络数据.

⬆ 返回目录

7.4 Attribute代理同步

在有很多玩家的大型游戏中, 像Fortnite大逃杀(Fortnite Battle Royale), 有大量存在于常相关`PlayerState`上的ASC, 其会同步大量Attribute. 为了优化该瓶颈, Fortnite禁止在`PlayerState::ReplicateSubobjects()`中完全同步`ASC`和它的AttributeSet到模拟玩家控制的代理(Simulated Player-Controlled Proxy)上. Autonomou代理和AI控制的`Pawn`仍然根据其同步模式来完全同步. Fortnite使用了玩家Pawn上的可同步代理结构体, 而不是同步常相关`PlayerState`上的`ASC`的`Attribute`. 当服务端ASC上的`Attribute`改变时, 其也会在代理结构体中改变, 客户端会从代理结构体接收同步的`Attribute`并将修改返回其本地`ASC`, 这允许`Attribute`同步使用`Pawn`的相关性(Relevancy)和`NetUpdateFrequency`, 该代理结构体也会使用位掩码(Bitmask)同步一个小的`GameplayTag`白名单集合. 该优化减少了网络数据量, 并允许我们利用Pawn相关性(Relevancy). AI控制的`Pawn`的`ASC`位于已经使用其相关性的`Pawn`上, 因此其并不需要该优化.

> I'm not sure if it is still necessary with other server side optimizations that have been done since then (Replication Graph, etc) and it is not the most maintainable pattern.

来自Epic的Dave Ratti回答社区问题#3.

⬆ 返回目录

7.5 ASC懒加载

Fortnite大逃杀(Fortnite Battle Royale)世界中有很多可损坏的`AActor`(树木, 建筑物等等), 每个都有一个`ASC`, 这会增加内存消耗, Fortnite通过只在需要时(当其第一次被玩家伤害时)懒加载`ASC`的方式优化该问题, 这会减少整体内存消耗, 因为某些`AActor`在一局比赛中可能永远不会被伤害.

⬆ 返回目录

8. Quality of Life Suggestions

8.1 Gameplay Effect Containers

GameplayEffectContainer将GameplayEffectSpec, TargetData, 简单定位和其他相关功能整合进简单易用的结构体, 这对转移`GameplayEffectSpec`到Ability生成的抛射物很有用, 该抛射物之后会在碰撞体上应用`GameplayEffectSpec`.

⬆ 返回目录

8.2 将蓝图AsyncTask绑定到ASC委托

为了增加设计师友好的迭代次数, 特别是在为UI设计UMG Widget时, 可以创建蓝图AsyncTask(在C++中)以直接在UMG蓝图图表中绑定到`ASC`中常见的修改委托, 唯一的告诫就是其必须手动销毁(比如在widget销毁时), 否则就会永远存于内存中.样例项目包含三个蓝图AsyncTask.

监听Attribute修改:

监听冷却时间修改:

监听GE堆栈修改:

⬆ 返回目录

9. 疑难解答

9.1 LogAbilitySystem: Warning: Can't activate LocalOnly or LocalPredicted ability %s when not local!

你需要在客户端初始化ASC.

⬆ 返回目录

9.2 ScriptStructCache错误

你需要调用UAbilitySystemGlobals::InitGlobalData().

⬆ 返回目录

9.3 动画蒙太奇不能同步到客户端

确保在GameplayAbility中正在使用的是PlayMontageAndWait节点而不是PlayMontage, 该AbilityTask可以通过ASC自动同步蒙太奇而PlayMontage节点不可以.

⬆ 返回目录

9.4 复制的蓝图Actor会将AttributeSet设置为nullptr

这是一个虚幻引擎的bug, 当使用从一个存在的蓝图Actor类复制的方式来创建新的类, 这会让这个类中将AttributeSet指针设置为空指针.

对此有一些变通的方法, 我已经成功地不在我的类内创建定制的AttributeSet指针(头文件中没有指针, 也不在构造函数中调用CreateDefaultSubobject),
而是直接在PostInitializeComponents()中向ASC添加AttributeSets(样本项目中没有显示).
复制的AttributeSets将仍然存在于ASC的SpawnedAttributes数组中. 它看起来像这样:

```
void AGDPlayerState::PostInitializeComponents()
{
    Super::PostInitializeComponents();

    if (AbilitySystemComponent)
    {
        AbilitySystemComponent->AddSet<UGDAttributeSetBase>();
        // ... 其他你可能拥有的属性集
    }
}
```

在这种情况下, 你想要读取或者修改AttributeSet的值, 就需要调用ASC实例中的函数, 而不是AttributeSet中定义的宏.

```
/** 返回当前(最终)属性值 */
float GetNumericAttribute(const FGameplayAttribute &Attribute) const;

/** 设置一个属性的基础值。 当前激活的修改器不会被清除并将在NewBaseValue上发挥作用 */
void SetNumericAttributeBase(const FGameplayAttribute &Attribute, float NewBaseValue);
```

所以GetHealth()的实现将会如下:

```
float AGDPlayerState::GetHealth() const
{
    if (AbilitySystemComponent)
    {
        return AbilitySystemComponent-
>GetNumericAttribute(UGDAttributeSetBase::GetHealthAttribute());
    }

    return 0.0f;
}
```

设置(初始化)生命值属性的实现将会是这样:

```
const float NewHealth = 100.0f;
if (AbilitySystemComponent)
{
    AbilitySystemComponent->SetNumericAttributeBase(UGDAttributeSetBase::GetHealthAttribute(),
NewHealth);
}
```

顺便提一下, 往ASC组件注册的每个AttributeSet类最多只有一个对象.

⬆ Back to Top

10. ASC常用术语缩略

术语	缩略
AbilitySystemComponent	ASC
AbilityTask	AT
Action RPG Sample Project by Epic	ARPG, ARPG Sample
CharacterMovementComponent	CMC
GameplayAbility	GA
GameplayAbilitySystem	GAS
GameplayCue	GC
GameplayEffect	GE
GameplayEffectExecutionCalculation	ExecCalc, Execution
GameplayTag	Tag, GT
ModiferMagnitudeCalculation	ModMagCalc, MMC

↑ 返回目录

11. 其他资源

官方文档

源代码! 特别是`GameplayPrediction.h`.

Epic的Action RPG样例项目

来自Epic的Dave Ratti回复社区关于GAS的问题

Unreal Slackers Discord有一个专注于GAS`#gameplay-abilities-plugin`的文字频道

Dan 'Pan'的Github库

SabreDartStudios的YouTube视频

↑ 返回目录

12. GAS更新日志

这是从Unreal Engine官方升级日志和我遇到的且未记录的升级中整理的一份值得一看的升级列表, 如果你发现某些没有列在其中, 请提issue或者PR.

⬆ 返回目录

4.26

- GAS plugin is no longer flagged as beta.
- Crash Fix: Fixed a crash when adding a gameplay tag without a valid tag source selection.
- Crash Fix: Added the path string arg to a message to fix a crash in UGameplayCueManager::VerifyNotifyAssetIsInValidPath.
- Crash Fix: Fixed an access violation crash in AbilitySystemComponent_Abilities when using a ptr without checking it.
- Bug Fix: Fixed a bug where stacking GEs that did not reset the duration on additional instances of the effect being applied.
- Bug Fix: Fixed an issue that caused CancelAllAbilities to only cancel non-instanced abilities.
- New: Added optional tag parameters to gameplay ability commit functions.
- New: Added StartTimeSeconds to PlayMontageAndWait ability task and improved comments.
- New: Added tag container "DynamicAbilityTags" to FGameplayAbilitySpec. These are optional ability tags that are replicated with the spec. They are also captured as source tags by applied gameplay effects.
- New: GameplayAbility IsLocallyControlled and HasAuthority functions are now callable from Blueprint.
- New: Visual logger will now only collect and store info about instant GEs if we're currently recording visual logging data.
- New: Added support for redirectors on gameplay attribute pins in blueprint nodes.
- New: Added new functionality for when root motion movement related ability tasks end they will return the movement component's movement mode to the movement mode it was in before the task started.

⬆ 返回目录

4.25.1

- Fixed! UE-92787 Crash saving blueprint with a Get Float Attribute node and the attribute pin is set inline
- Fixed! UE-92810 Crash spawning actor with instance editable gameplay tag property that was changed inline

⬆ 返回目录

4.25

- Fixed prediction of RootMotionSource AbilityTasks
- GAMEPLAYATTRIBUTE_REPNOTIFY() now additionally takes in the old Attribute value. We must supply that as the optional parameter to our OnRep functions. Previously, it was reading the attribute value to try to get the old value. However, if called from a replication function, the old value had already been discarded before reaching SetBaseAttributeValueFromReplication so we'd get the new value instead.
- Added NetSecurityPolicy to UGameplayAbility.
- Crash Fix: Fixed a crash when adding a gameplay tag without a valid tag source selection.
- Crash Fix: Removed a few ways for attackers to crash a server through the ability system.

- Crash Fix: We now make sure we have a GamplayEffect definition before checking tag requirements.
- Bug Fix: Fixed an issue with gameplay tag categories not applying to function parameters in Blueprints if they were part of a function terminator node.
- Bug Fix: Fixed an issue with gameplay effects' tags not being replicated with multiple viewports.
- Bug Fix: Fixed a bug where a gameplay ability spec could be invalidated by the InternalTryActivateAbility function while looping through triggered abilities.
- Bug Fix: Changed how we handle updating gameplay tags inside of tag count containers. When deferring the update of parent tags while removing gameplay tags, we will now call the change-related delegates after the parent tags have updated. This ensures that the tag table is in a consistent state when the delegates broadcast.
- Bug Fix: We now make a copy of the spawned target actor array before iterating over it inside when confirming targets because some callbacks may modify the array.
- Bug Fix: Fixed a bug where stacking GamplayEffects that did not reset the duration on additional instances of the effect being applied and with set by caller durations would only have the duration correctly set for the first instance on the stack. All other GE specs in the stack would have a duration of 1 second. Added automation tests to detect this case.
- Bug Fix: Fixed a bug that could occur if handling gameplay event delegates modified the list of gameplay event delegates.
- Bug Fix: Fixed a bug causing GiveAbilityAndActivateOnce to behave inconsistently.
- Bug Fix: Reordered some operations inside FGameplayEffectSpec::Initialize to deal with a potential ordering dependency.
- New: UGameplayAbility now has an OnRemoveAbility function. It follows the same pattern as OnGiveAbility and is only called on the primary instance of the ability or the class default object.
- New: When displaying blocked ability tags, the debug text now includes the total number of blocked tags.
- New: Renamed UAbilitySystemComponent::InternalServerTryActiveAbility to UAbilitySystemComponent::InternalServerTryActivateAbility.Code that was calling InternalServerTryActiveAbility should now call InternalServerTryActivateAbility.
- New: Continue to use the filter text for displaying gameplay tags when a tag is added or deleted. The previous behaviour cleared the filter.
- New: Don't reset the tag source when we add a new tag in the editor.
- New: Added the ability to query an ability system component for all active gameplay effects that have a specified set of tags. The new function is called GetActiveEffectsWithAllTags and can be accessed through code or blueprints.
- New: When root motion movement related ability tasks end they now return the movement component's movement mode to the movement mode it was in before the task started.
- New: Made SpawnedAttributes transient so it won't save data that can become stale and incorrect. Added null checks to prevent any currently saved stale data from propagating. This prevents problems related to bad data getting stored in SpawnedAttributes.
- API Change: AddDefaultSubobjectSet has been deprecated. AddAttributeSetSubobject should be used instead.
- New: Gameplay Abilities can now specify the Anim Instance on which to play a montage.

↑ 返回目录

4.24

- Fixed blueprint node Attribute variables resetting to None on compile.
- Need to call UAbilitySystemGlobals::InitGlobalData() to use TargetData otherwise you will get ScriptStructCache errors and clients will be disconnected from the server. My advice is to always call this in every project now whereas before 4.24 it was optional.
- Fixed crash when copying a GameplayTag setter to a blueprint that didn't have the variable previously defined.
- UGameplayAbility::MontageStop() function now properly uses the OverrideBlendOutTime parameter.
- Fixed GameplayTag query variables on components not being modified when edited.
- Added the ability for GameplayEffectExecutionCalculations to support scoped modifiers against "temporary variables" that aren't required to be backed by an attribute capture.

    - Implementation basically enables GameplayTag-identified aggregators to be created as a means for an execution to expose a temporary value to be manipulated with scoped modifiers; you can now build formulas that want manipulatable values that don't need to be captured from a source or target.
    - To use, an execution has to add a tag to the new member variable ValidTransientAggregatorIdentifiers; those tags will show up in the calculation modifier array of scoped mods at the bottom, marked as temporary variables—with updated details customizations accordingly to support feature
- Added restricted tag quality-of-life improvements. Removed the default option for restricted GameplayTag source. We no longer reset the source when adding restricted tags to make it easier to add several in a row.
- APawn::PossessedBy() now sets the owner of the Pawn to the new Controller. Useful because Mixed Replication Mode expects the owner of the Pawn to be the Controller if the ASC lives on the Pawn.
- Fixed bug with POD (Plain Old Data) in FAttributeSetInittterDiscreteLevels.

↑ 返回目录