Ctrl+K, O

Dante\Character\

InputDefine中，每个FInputKey都有个Tick ，在UCharacterInputBufferComp中Tick

GameEventManager

template<typename T>

struct GlobalEvent

{

GlobalEvent()

: type(static\_cast<T>(INVALID\_EVENT\_TYPE))

{ }

T type;

typedef TArray<uint8> TDataBuffer;

TDataBuffer paramBuffer;

};

定义的全局事件，看起来是通过发送Array

#define BUILD\_GLOBAL\_EVENT\_PARAMS(EventObj, EventType, ...)\

do {\

EventObj.*type* = EventType;\

FMemoryWriter writer(EventObj.paramBuffer);\

VORTEX\_FOREACH\_ASSIGN(writer, FOREACH\_DO\_SERIAILIZE\_VAR, \_\_VA\_ARGS\_\_);\

} while (0)

这俩宏，创建以及发送全局Event,下面的宏是带参数的类型，还有不带的

#define EMIT\_GLOBAL\_EVENT\_PARAMS(EventType, ...)\

do {\

FGameEvent eventObj;\

BUILD\_GLOBAL\_EVENT\_PARAMS(eventObj, EventType, \_\_VA\_ARGS\_\_);\

FGameEventManager::Singleton().EmitEvent(eventObj);\

} while (0)

typedef GoodGame::GlobalEvent<EGameEvent> FGameEvent; 一个事件的Event

EGameEvent 就是一堆的事件类型枚举

FGameEventManager 全局事件管理器

GlobalEventManager.hpp 会将要Emit的Event记录在m\_needEmitEventArray中，在Update的时候，去进行统一发送，而不是当即帧马上发送（避免卡顿？同帧执行过多？）

SendEvent还是使用的委托多播

FGameEventManager::Singleton().ConnectEvent( 此方法来注册事件

单例：通过GoodGame里面的Singleton的DECLARE\_SINGLETON在声明函数时来进行创建。

CharacterInputBufferComp：

ActionParam : 里面包含了

struct ActionParam

{

ECharacterPlayerInputKey Action{ ECharacterPlayerInputKey::Empty };

ECharacterAttackComboKey AttackComboKey{ ECharacterAttackComboKey::AttackCombo\_None };

ECharacterSkillType SkillType{ ECharacterSkillType::Empty };

};

FCharacterStateMachine::State\_ChargedAttack\_Begin() 在此使用技能

右键点击后跳转到

Ability是但丁的Ability : public UObject

void ADanteCharacter::Tick(float DeltaTime)

{

Super::Tick(DeltaTime);

UpdateActorRotation();

CharacterInputBufferComp->UpdateInputKey(EInputKey::Move, IE\_Axis);

StateMachine->UpdateState(DeltaTime);

if (bIsTrain)

{

TickChekAiParamChange();

AiMoveUpdate(DeltaTime);

}

}

在dante的tick里面，去进行切换state，攻击就是在这里分发的。

在GoodGame的State中，声明了很多State

在Machine的PreStateUpdate中，直接通过const TArray<EAbilityType>& ActivedAbilitys = CharacterInstance->AbilityComponent->GetActivedAbilitys()

获取能够运行的ability

再根据ECharacterState 获取对应的状态机状态，进而使用SWITCH\_STATE\_AND\_RETURN(StateID);切换状态。

## 状态机研究：

状态：通过三个无参数的委托发哦是哪个

FActionDelegate BeginAction;

FActionDelegate EndAction;

FActionDelegate UpdateAction;

复合状态：就是会在状态中在设置一个parentState ， 自身Update与End等的时候，调用Parent状态。

在CharacterStateMachineBase中通过DECLARE\_STATE\_AND\_FUNCTION\_VIRTUAL 去定义状态以及声明状态的开始函数结束函数更新函数。

#define DECLARE\_ALL\_STATE\_FUNCTION\_VIRTUAL(stateName) \

virtual void State\_##stateName##\_Begin(); \

virtual void State\_##stateName##\_Update(); \

virtual void State\_##stateName##\_End()

BIND\_ALL\_STATE\_FUNCTION 通过该函数去设定给定StateName的名字与ID并且，在该State上的委托上绑定由之前声明的Begin() , Update(),End()函数。

**声明了之后，在CharacterStateMachineBase.hpp中，对每个函数进行了定义。**

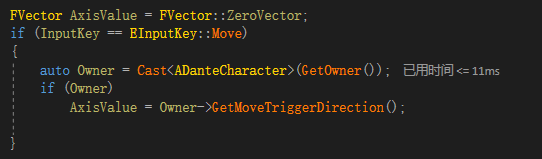
状态机运行流程：

ADanteCharacter上绑定按键回调。

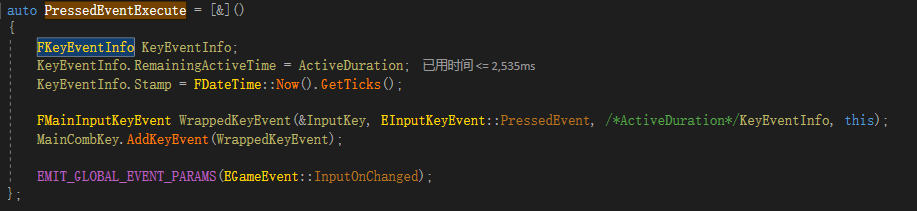
ADanteCharacter：：Tick 会从ChkaracterInputBufferComp的UpdateInputKey中先更新

UpdateInputKey(EInputKey::Move, IE\_Axis);

就是说，Character在每帧会主动更新Move的移动输入，其余都是等待按键之后的绑定调用去运行，就是你啥也不干，他也会每帧在UpdateInputKey中去



如果是InputEvent的事件调用，会根据键按下或是释放，生成FKeyEventInfo去推送全局事件



再将当前的按键事件包装一个MainInputKeyEvent，再加入到MainCombKey中。

同样的按键抬起事件，也会放到一个FMainInputKeyEvent中，ReleasedEventMainKey，用作最后一个的按键抬起保存。

///////////////////////////////插一个段：

这两个事件都是触发OnInputChanged，

**这个MainCombKey ，是一个FMainInputCombKey：**

自身有Tick，应该是用来运行当前有多少Combo按键事件的？因为会判断他们的持续时间并一直减少。

////////////////插一个段完了

StateMachine->UpdateState(DeltaTime); 状态机的Update，

其中的PreStateUpdate会去取身上的AbilityComponent中的所有激活的能力，并且去判断是否可用，Attack也是一个能力Ability。

示例：

点击一下攻击，调用了State\_Attack\_Begin 这个状态函数的begin，跟踪代码运行发现，是在StateMachine的PreStateUpdate中就发现了攻击这个能力变为了可用，进而调用了SWITCH\_STATE\_AND\_RETURN(StateID); 切换到了攻击状态。那么跟进下是为何攻击的Ability变为可用的。（在下两行写出了）

**在UCharacterInputBufferComp的OnInputChanged中，通过Match函数，将EActionFlags设置为当前按键对应的Ability状态。**

**之后调用SetActionTriggers 通过StrategyInputActionDelegate 将当前的ActionFlags发送出去, 之后在后续的代码中会根据输入的TriggerAction去找到对应的Ability并且最终调用DanteCharacterBase中的TriggerAbility 将Ability的bBeTriggered设为True,之后在Ability的CheckAny中，就可以检测判断能力可用了。**

**从State\_Attack中来分析状态机的运行：**

State\_Skill\_Bgein和State\_Attack\_Begin都是调用CharacterStateMachine的AbilityComponent的UseAbility来进行攻击和技能的开始。

这个AbilityData->Data 好像是这么用的，AbilityData->Data[0] 取他的ComboType,因为如果不是上一个状态不是攻击状态就会Add（0），上一个状态是攻击状态，就会Add（1）

之后就是applyAttack，去设置当前需要播放的动画蒙太奇，关闭rootMotion，然后PlayMontage

State\_Attack\_Update的时候，会去将Actor旋转到配置好的目标角度，并实时判断蒙太奇是否结束，结束会return True并切换状态机。

void UAttack::End()

{

Caster->CharacterWeaponComp->GetCurrentWeapon()->EnableCollision(false);

if (bDisableAttackMontageRootMotion)

{

Caster->GetAnimOperator()->EnableRootMotion(AttackMontage);

}

EndRotate();

Clear();

}

最后End即可，

## AttackHandlerComponent：

void UAttackHandlerComponent::Reset()

{

FAttackUtility::ResetAttackData(AttackData);

}

主要为在里面的这个方法, 这个方法就是把传入的AttackData设置为初始值 0啊之类的。

相当于，在Character身上挂一个这个Component，负责提供一个AttackData

## AnimationOperator

相当于包含了一对功能的，组件，在Character初始化的时候，创建爱你，并且设置初始参数。

AnimOperator = MakeShared<GoodGame::FAnimationOperator>();

TArray<USkeletalMeshComponent\*> SkeMeshComps;

GetComponents<USkeletalMeshComponent>(SkeMeshComps);

check(SkeMeshComps.Num() > 0);

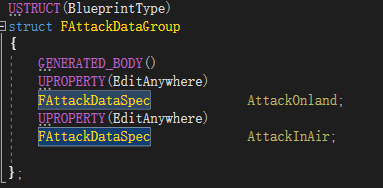
AnimOperator->Init(SkeMeshComps[0]->GetAnimInstance());

## UAttack：

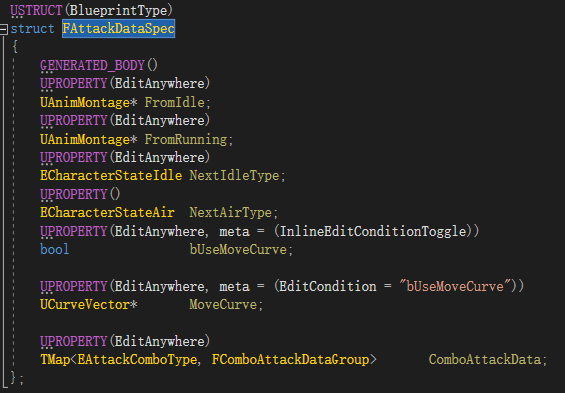
FEnemyUtility::FindEnemyInRange查找范围内敌人

在UAttack的ApplyAttack中，ApplyFirstAttack ， 其中有一个

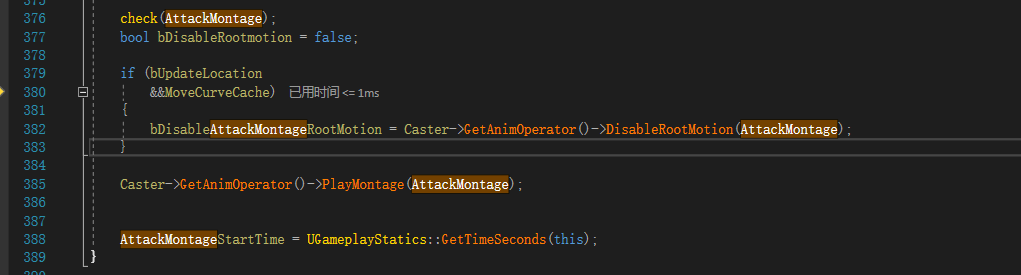
FAttackDataGroup类型，内部的变量有



这就是与状态机联系的部分了，



里面包含了对应的Montage，在ApplyFirstAttack中，



正式播放蒙太奇就在这里了。

**CharacterStateDefind.h 记录了玩家的各种数据信息。**

## 看一看Attack这个模块（文件夹）应该是包含了被攻击的流程

UAttackEffectHandlerComponent：

会先创建两个RenderTarget，一个是UVToWorldPos 一个是HitMask

USceneCaptureComponent2D 小摄像机，将图像做另外的输出, ShowOnlyActors 将玩家自身加入进去之后，令该摄像机只补捉玩家图像。

嗯？？是在，DanteEnemyBase中进的BeginPlay进行的初始化，但是却并没有绑在Enemy身上。从引用上看，只有武器weapon.cpp会去调一下这个，但是并没有使用到目前，也许是有什么特殊技能把AWeapon::PlayScratchEffect()

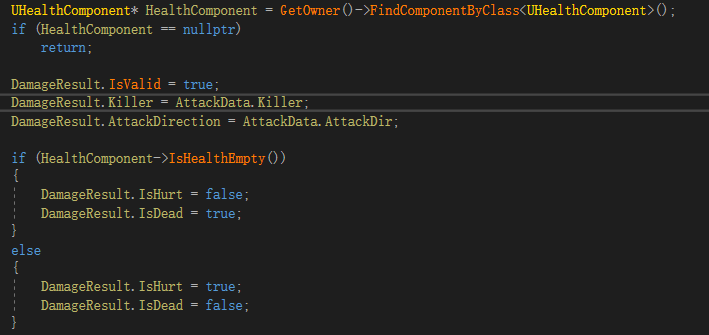
DamageHandlerComponent:

每次收到攻击，会调用AddDamage函数（由HitBoxCollisionOperator的OnCollisionOverBegin 函数中调用发送,owner是被攻击者），在AttackPool中添加一次攻击data，该Data包含了攻击者与被害者。

每个DamageHandlerComponent身上都会创建一个FAttackPool。

AttackPool会把身上的攻击广播到DamageHandlerComponent 的 OnDamage中，通过FAttackDispatcher的Dispatch去分发.（比如玩家受到攻击，调用ADanteCharacterBase的OnDamage）

耶啊分发后又回来啦，调用被攻击者自身的UDamageHandlerComponent::ApplyDamage ，（攻击结果是使用FDamageResult进行传递的），之后在ApplyPendingDamage中，会去判断



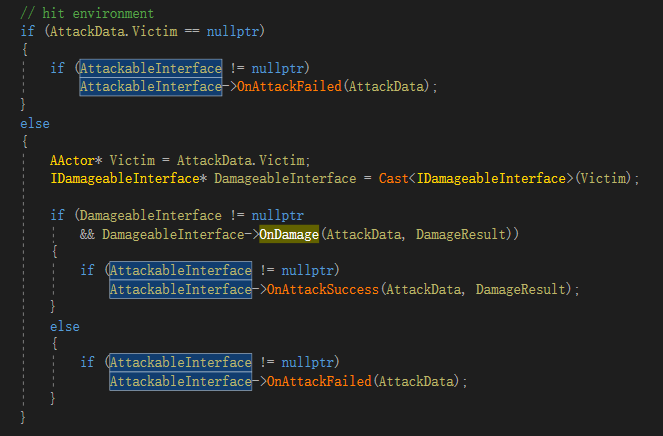
最后计算出一个确定的伤害结果，返回到Actor身上的HealthComponent进行计算。

AttackPool：

Update时会遍历身上所有的攻击Data，再把事件根据注册的委托广播出去。

AttackDispatcher:

分发是通过，获取AttackData 的Killer和Victim去Cast成IDamageableInterface，根据攻击逻辑去调用对应的函数。



## Ability

Ability已abilityComponent作为Outer进行创建，Ability自身的Data由Component的GetAblilityData去创建。

USTRUCT(BlueprintType)

struct FAbilityDataRow :public FDanteTableRowBase

配置按键，Ability类型，Category等，其中的

FPatternInputCombKey ActiveKey;

就是用来记录，按什么建以及按键的方式等等，这个还有个很重要的功能，多个键同时按下的情况！

是放在CharacterStateDefind中的，存放于Ability身上的，能力数据

struct FAbilityData

{

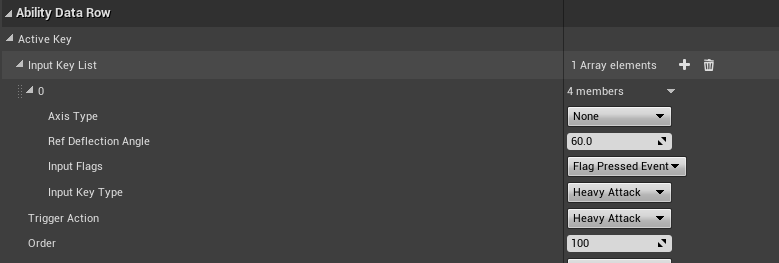
GENERATED\_BODY()

EAbilityType AbilityType= EAbilityType::Empty;

bool bBeTriggered=false;

TArray<uint8> Data;

};

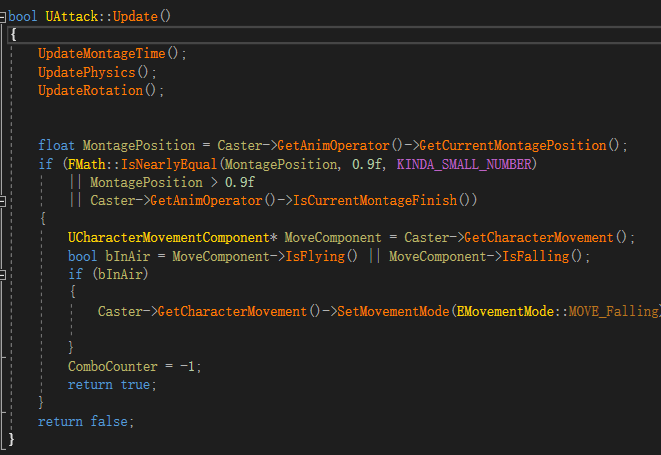


FAttackDataRow 用来记录Ability的数据, 在空中和在地面的蒙太奇，

AbilityComponent中，TickAbilitys去tick身上所有的Ability

在Ability中，也会更新蒙太奇等数据的，例如在UAttack中的Update，可以看到更新

AbilityData->bBeTriggered 是在CheckAny中进行判断，当此值为true时，会进入流程判断该Ability是否可用



## DataManager

所有数据都是继承自FDataTableRowBase

TMap<const UScriptStruct\*, FDanteDataTableContainer> DataContainer;

通过：

template<class T>

TArray<T\*> FAssetUtility::LoadAssetsByPath(const FString& Path)

{

TArray<T\*> Out;

for (FAssetData AD : GetAssetsInPath(Path))

{

T\* OutAsset = Cast<T>(AD.GetAsset());

if (OutAsset)

{

Out.Add(OutAsset);

}

}

return Out;

}

将所有的Data文件一次性载入，再根据配置表的UScriptStruct做Key存放在DataManager中。

## Input相关

FPatternInputCombKey 组合键相关

## 碰撞相关

HitBoxCollisionOperator :