Unreal4 笔记

郑华

2021 年 12 月 14 日

目录

第一章　GamePlay

1.1　基础世界观

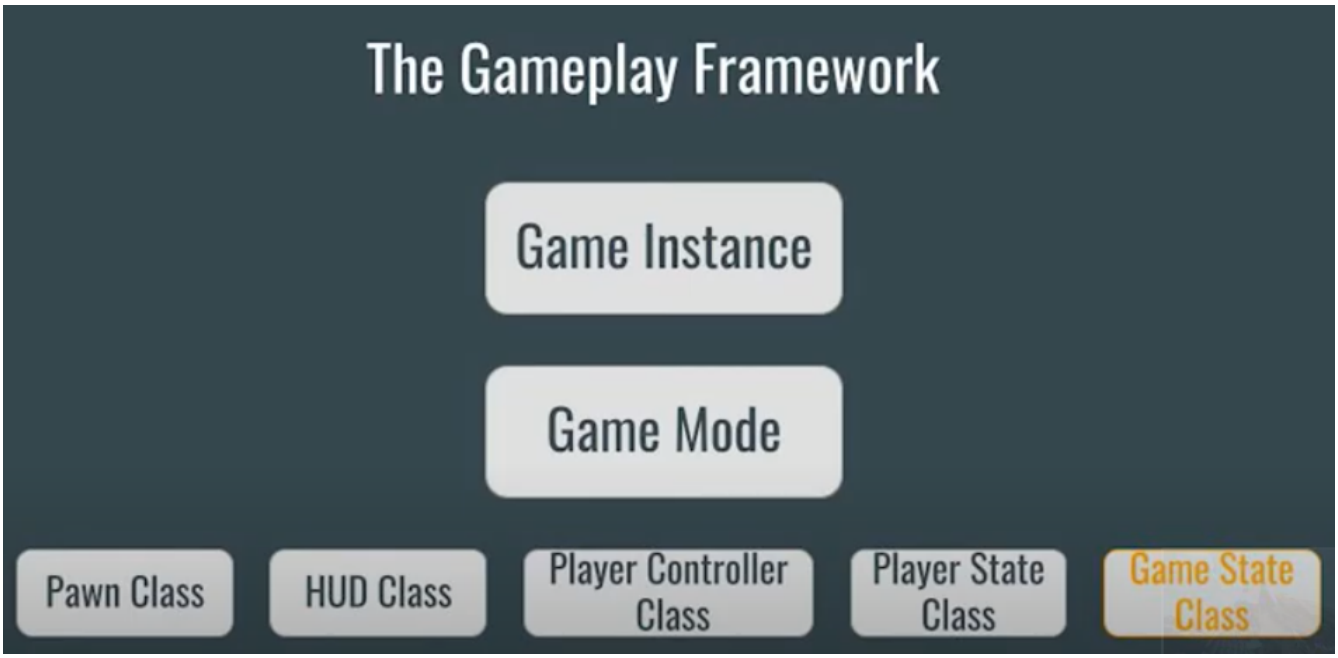

图 1.1: Gameplay Framwork

- Game Instance: 从游戏开始到游戏结束，可保存跨场景数据等。
- Game Mode: 储存游戏中不经常被修改的数据
- Pawn Class: 人物、AI 载体实体
- HUD Class: 二维显示信息
- Player Controller Class; 代表玩家的控制：包括输入操作、AI 控制等
- Player State Class: 玩家的状态信息
- Game State Class: 任务状态等

1.2 基础元素与组件

1.2.1 UObject

UObject 类提供了以下功能：

- 垃圾收集
- 引用自动更新
- 反射
- 序列化
- 自动检测默认变量的更改
- 自动变量初始化
- 和虚幻引擎编辑器的自动交互
- 运行时类型识别
- 网络复制

1.2.2 Actor

Actor 类在场景中拥有一个位置坐标和旋转量。

Actor 类拥有这样的能力：**它能够被挂载组件**。

1.2.3 Pawn

现 Pawn 类提供了被“操作”的特性。

它能够被一个 Controller 操纵。这个 Controller 可以是玩家，当然也可以是 AI（人工智能）。这就像是一个棋手，操作着这个棋子。

这就是 Pawn 类，一个被操纵的兵或卒，一个一旦脱离棋手就无法自主行动的、悲哀的肉体。

1.2.4 Controller

既然是灵魂，那么肉体就不唯一，因此灵魂（Controller）可以通过 Possess/UnPossess 来控制一个肉体，或者从一个肉体上离开。

1.2.5 Character

Character 类代表一个角色，它继承自 Pawn 类。

Character 类提供了一个特殊的组件，Character Movement。这个组件提供了一个基础的、基于胶囊体的角色移动功能。包括移动和跳跃，以及如果你需要，还能扩展出更多，例如蹲伏和爬行。

1.3 控制

1.4 迭代

1.5 命名规则

常用的前缀如下：

- F 纯 C++ 类
- U 继承自 UObject，但不继承自 Actor
- A 继承自 Actor
- S Slate 控件相关类
- H HitResult 相关类

1.6 对象

1.6.1 生成

在标准 C++ 中，一个类产生一个对象，被称为“实例化”。实例化对象的方法是通过 new 关键字。而在虚幻引擎中，这一个问题变得略微复杂。对于某些类型，我们不得不通过调用某些

函数来产生出对象。具体而言：

- 如果你的类是一个**纯 C++ 类**型（F 开头），你可以通过 **new** 来产生对象。
- 如果你的类**继承自 UObject** 但不继承自 Actor，你需要通过 **NewObject 函数**来产生出对象。`NewObject<T>()`
- 如果你的类**继承自 AActor**,你需要通过 **SpawnActor 函数**来产生出对象。`GetWorld()-> SpawnAct`
- 如果你需要产生出一个 **Slate 类**，需要使用 **SNew 函数**。

1.6.2 获取

：

1 • 通过 new 获取引用或者指针。

2 • 间接的通过遍历对象获取。

```
for(TActorIterator <AActor> Iterator(GetWorld()); Iterator; ++Iterator)
{ ...//do something }
```

1.6.3 销毁

- 纯 C++ 类：符合 C++ 基本语法
- UObject 类：UObject 采用自动垃圾回收机制。当一个**类的成员**变量包含指向 *UObject* 的对象，同时又带有 *UPROPERTY* 宏定义，那么这个成员变量将会触发引用计数机制。垃圾回收器会定期从根节点 Root 开始检查，当一个 UObject 没有被别的任何 UObject 引用，就会被垃圾回收。你可以通过 AddToRoot 函数来让一个 UObject 一直不被回收。
- Actor 类：可以通过**调用 Destory 函数**来请求销毁，这样的销毁意味着将当前 Actor 从所属的世界中“摧毁”。但是对象对应内存的回收依然是由系统决定。

1.7 subsystem

Subsystems 是一套可以定义、自动实例化和释放的类的框架。可以理解为后台管理程序，或者游戏运行支持系统。

1.7.1 概述

更易管理的生命周期 引擎只支持一个 GameInstance ，运行周期是整个引擎的生命周期

自定义 ManagerActor，生命周期一般为当前 level 的生命周期

Subsystems 的生命周期可以依存于 Engine，Editor，World，LocalPlayer

更简单易生成 GameInstance 或者自定义 ManagerActor，需要手动维控制创建、释放

Subsystems 自动创建、释放，提供 Initialize()、Deinitialize()，并且可重载

模块化、代码更易读

1.7.2 类关系

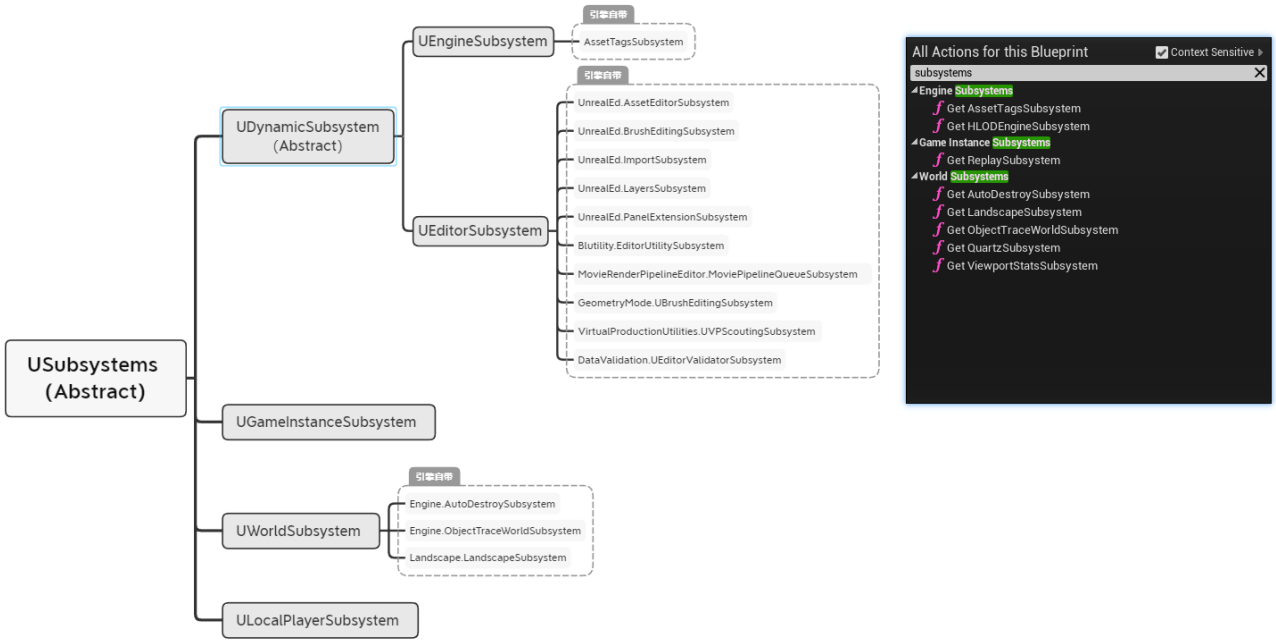

图 1.2: 子系统类型及类图

1.7.3 生命周期

如上：主要描述各种 subsystem 的启动时机与销毁时机。

- UDynamicSubsystem
  - UEngineSubsystem：从进程启动开始创建，进程退出时销毁。

  - UEditorSubsystem：从编辑器启动开始创建，到编辑器退出时销毁。

- UGameInstanceSubsystem：从游戏的启动开始创建，游戏退出时销毁。这里的一场游戏指的是 Runtime 或 PIE 模式的运行的都算，一场游戏里可能会创建多个 World 切换。
- UWorldSubsystem：其生命周期，跟 GameMode 是一起的。(EWorldType:Game，Editor，PIE，EditorPreview，GamePreview 等)
- ULocalPlayerSubsystem：LocalPlayer 虽然往往跟 PlayerController 一起访问，但是其生命周期其实是跟 UGameInstance 一起的 (默认一开始的时候就创建好一定数量的本地玩家)，或者更准确的说是跟 LocalPlayer 的具体数量挂钩（当然你也可以运行时动态调用 AddLocalPlayer)。

1.7.4 创建示例 C++

示例代码：

```
// UMyEngineSubsystem 获取
UMyEngineSubsystem* MyEngineSubsystem = GEngine->GetEngineSubsystem<
    UMyEngineSubsystem>();

// UMyEditorSubsystem 获取
UMyEditorSubsystem* MyEditorSubsystem = GEditor->GetEditorSubsystem<
    UMyEditorSubsystem>();

// UMyGameInstanceSubsystem 获取
//UGameInstance* GameInstance = GetWorld()->GetGameInstance();
UGameInstance* GameInstance = UGameplayStatics::GetGameInstance();
UMyGameInstanceSubsystem* MyGameInstanceSubsystem = GameInstance->GetSubsystem<
    UMyGameInstanceSubsystem>();

// UMyWorldSubsystem 获取
UMyWorldSubsystem* MyWorldSubsystem = GetWorld()->GetSubsystem<UMyWorldSubsystem>();

// UMyLocalPlayerSubsystem 获取
ULocalPlayer* LocalPlayer = UGameplayStatics::GetPlayerController()->GetLocalPlayer
    ();
UMyLocalPlayerSubsystem* MyLocalPlayerSubsystem = LocalPlayer->GetSubsystem<
    UMyLocalPlayerSubsystem>();
```

第二章　蓝图

2.1　基本概念

蓝图类，类似于 prefab，定义一种**基于事件**的预工作流。

2.1.1　分类

- 类蓝图 Blueprint Class：是一种允许内容创建者轻松地基于现有游戏性类添加功能的资源。
- 纯数据蓝图 Data-Only Blueprint：是指仅包含代码 (以节点图表的形式)、**变量**及**从其父类继承的组件**的类蓝图。
- 关卡蓝图 Level Blueprint：用作关卡范围的**全局事件**图。

2.1.2　蓝图接口

蓝图接口（Blueprint Interface）是一个或多个函数的集合 - 只有名称，没有实施-可以添加到其他蓝图中。

2.1.3　蓝图宏库

蓝图宏库（Blueprint Macro Library）是一个容器，它包含一组宏或自包含的图表，这些图表可以作为节点放置在其他蓝图中。它们可以节省时间，因为它们可以存储常用的节点序列，包括执行和数据传输所需的输入和输出。

2.1.4 蓝图工具

蓝图编辑器（Blueprint Editor）中的 组件（Components）窗口允许您将组件添加到蓝图。这提供了以下方法：通过胶囊组件（CapsuleComponent）、盒体组件（BoxComponent）或球体组件（SphereComponent）添加碰撞几何体，以静态网格体组件（StaticMeshComponent）或金属网格体组件（SkeletalMeshComponent）形式添加渲染几何体，使用移动组件（MovementComponent）控制移动。还可以将组件（Components）列表中添加的组件指定给实例变量，以便您在此蓝图或其他蓝图的图表中访问它们。

2.1.5 核心组件

- 构造函数
- 事件图表：事件驱动
- 函数
- 变量

2.2 使用规则

基于事件。

控制流、数据流。

2.3 调用 C++

如何让蓝图能够调用我的 C++ 类中的函数呢？

- `UPROPERTY`宏: 注册一个变量到蓝图中`UPROPERTY(BlueprintReadWrite,VisibleAnywhere,Categor`
- `UFUNCTION`宏: 注册函数到蓝图中 `UFUNCTION(BlueprintCallable,Category="Test")`

第三章　事件

第四章　委托

4.1　概述

是 UE4 中一种自定义的消息机制。基本上可以将 delegate 与 event 相等同。

4.2　使用

4.2.1　单个消息监听

1. 申明一个代理（消息）类型

```
DECLARE_DELEGATE(FStandardDelegateSignature)
```

FStandardDelegateSignature 就是我们自己定义代理类型，可以将 FStandardDelegateSignature 看成使用 UE 宏 DECLARE_DELEGATE 声明的一个类 (Class)

2. 定义一个代理（消息）

```
FStandardDelegateSignature MyStandardDelegate;
```

3. 绑定响应函数

```
MyStandardDelegate.BindUObject(this, &ADelegateListener::EnableLight)
```

4. 触发代理（消息）

```
MyStandardDelegate.ExeIfBound();
```

5. 解除绑定响应函数

```
MyStandardDelegate.Unbind();
```

4.2.2 多消息监听

1. 声明委托

```
DECLARE_MULTICAST_DELEGATE_OneParam(FMuitiDelagateWithOneParam, FString);
```

2. 定义委托变量

```
FMuitiDelagateWithOneParam MuitiDelagateWithOneParam;
```

3. 绑定函数指针 AddUObject

```
fdele01 = MuitiDelagateWithNoParam.AddUObject(this, &ThisClass::Func1);
```

4. 执行委托，触发函数

```
MuitiDelagateWithOneParam.Broadcast(FString("PerformMultiDelagateWithOneParam"
    ));
```

5. Remove 和 RemoveAll:Remove 的参数为委托 AdddUObject 返回的句柄 FDelegateHandle

```
MuitiDelagateWithOneParam.Remove(fdele_01) ;
```

4.3 原理

第五章　引擎系统

5.1　正则

```
#include"Regex.h"
```

5.2　FPaths

：

- 具体路径类，如：FPaths::GameDir() 可以获取到游戏根目录。
- 工具类，如：FPaths::FileExists() 用于判断一个文件是否存在。
- 路径转换类，如：FPaths::ConvertRelativePathToFull() 用于将相对路径转换为绝对路径。

5.3　XML、Json

```
Include"Json.h"
```

5.4　GConfig

写配置、读配置

```
GConfig->SetString( TEXT("MySection"), TEXT("Name"), TEXT("李白"), FPaths::GameDir()
    / "MyConfig.ini");

FString Result;
GConfig->GetString( TEXT("MySection"), TEXT("Name"), Result, FPaths::GameDir() / "
    MyConfig.ini");
```

5.5 File

虚幻引擎提供了与平台无关的文件读写与访问接口，即 FPlatformFileManager。

5.6 UE_LOG

5.6.1 查看 log

: Log 窗口（Window->DeveloperTools- >OutputLog）

5.6.2 log 类型

：UE_LOG 宏输出 Log，第一个参数为 Log 的分类（需要预先定义）。第二个参数为类型，有 Log、Warning、Error 三种类型。

自定义 Category:DEFINE_LOG_CATEGORY_STATIC(LogMyCategory,Warning,All);

5.6.3 打印 log

:

```
UE_LOG(LogMy, Warning, TEXT("Hell␣World"));
UE_LOG(LogMy, Warning, TEXT("Show␣a␣String␣%s"),*FString("Hello"));
UE_LOG(LogMy, Warning, TEXT("Show␣a␣Int␣%d"),100);
```

5.7 string

“文字”类型其实是一组类型：*FName*，*FText* 和 *FString*。这**三种类型**可以互相转换。

5.7.1 FName

FName 是无法被修改的字符串，大小写不敏感。

5.7.2 FText

FText 表示一个“被显示的字符串”。所有你希望“显示”的字符串都应该是 FText。因为 FText 提供了内置的本地化支持，也通过一张查找表来支持运行时本地化。FText 不提供任何的更改操作，对于被显示的字符串来说，“修改”是一个非常不安全的操作。

5.7.3 FString

FString 是唯一提供修改操作的字符串类。同时也意味着 FString 的消耗要高于 FName 和 FText。

5.8 Images

5.9 编译器相关

第六章　AI-行为树

6.1　行为-基础元素

6.1.1　流程控制

:

- `Selector`: 选择器 IF

  Selector 节点会从左到右逐个执行下面的子树，如果有一个子树返回 true，它就会返回 true，只有所有的子树均返回 false，它才会返回 false。这就类似于日常生活中“几个方案都试一试”的概念。

- `Sequence`: 顺序执行器 STATEMENT

  Sequence 节点会按顺序执行自己的子树，只有当前子树返回 true，才会去执行下一个子树，直到全部执行完毕，才会向上一级返回 true。任何一个子树返回了 false，它就会停止执行，返回 false。类似于日常生活中“依次执行”的概念。

- `Parallel`: 并行执行 THREAD

6.1.2　装饰器

：对子树的返回结果进行处理的节点。

6.1.3　执行节点

：执行节点必然是叶子节点，执行具体的任务，并在任务执行一段时间后，根据任务执行成功与否，返回 *true* 或者 *false*。

6.2 行为-细节

6.2.1 Selector

终止条件：当某一个执行成功返回 True。

优先级：

6.2.2 Sequence

6.2.3 Parallel

第七章　AI-强化学习

第八章　编译

8.1　模块、UnrealBuildTool

虚幻引擎的源码目录，会发现其按照四大部分：Runtime、Development、Editor、Plugin 来进行规划，而每个部分内部，则包含了一个一个的小文件夹。每个文件夹即对应一个模块。

- Public 文件夹
- Private 文件夹
- .build.cs 文件

8.2　编译问题

第九章　光照

第十章　寻路

第十一章　UMG

第十二章　物理

第十三章　动画

13.1　骨骼动画

基础概念：ref:https://blog.csdn.net/shenshen211/article/details/72779882

蒙皮动画：https://blog.csdn.net/shenshen211/article/details/72779917

https://blog.csdn.net/shenshen211/article/details/72779933

汇总：https://blog.csdn.net/qq_23030843/article/details/109103433

官方:https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/

顶点动画需要你移动每个顶点，而骨骼动画使你只需移动模型中的一根骨头，顶点将能随之而变。

13.1.1　概述

骨骼动画是使用“骨头”来运动一个模型而不是通过手动编辑和移动每个顶点或面而实现的动画。

每个顶点被附着到一根骨头 (或有时是多根骨头)。

一根骨头或一个关节只是一组顶点的一个控制点。

当骨头运动时，每个**附着在它上的顶点**也跟着运动，甚至骨头自身的运动也会导致其它骨头的运动。

优点、区别　区别：顶点动画需要移动每个顶点，而骨骼动画只需移动模型中的一根骨头，顶点将能随之而变。

在传统的关键帧动画中，游戏会在两个位置中间进行线性插值。

与帧动画不同的是：骨骼动画每帧都存储一组新的顶点，所有我们需要存储的只是骨头的**旋转和平移**信息。这可以节省巨大的存储空间，你仅仅只要加入骨头和顶点—骨头附着信息。

因为骨骼的自由性，这使得你可以随心所欲地实时定位它们，也可以实现在运行时创建动画。就相当于提供了一个更多样化的动画库。你甚至能够让游戏控制当身体碰到一个物体时的动作，或从一个斜坡上滑下来。这种技术是在玩的时候即时产生的。

13.1.2 骨骼动画原理

仿生现象 来看看你的手臂。将你的手臂在你胸前展开并观察它。你的手臂有许多骨头，两根主要的大骨头 (two main ones)，还有手掌和手指上的一些骨头。

移动手指，只会移动到手指，而手臂上的其它部分并不会因此而移动。

弯曲肘关节，不仅手臂移动了，而且手指和手掌也同样跟着移动了。

当移动手臂“上游”的一根骨头时，任何在这根骨头的下游部分也都会跟着移动。这就是**骨骼动画最基本的概念**。

这样就有一个美妙之处是: 它可以允许 你**移动模型上的任何一根骨头，并渗透到下面的移动**，应用到以这个移动为原点的下面的任何事物。例如，这允许你只需移动角色的肩膀，不需担心肘和手的移动位置，因为它们会自动移到正确位置。你也能够通过复位以确保它们会**被自动更新**。

骨头、关节 关节、骨头

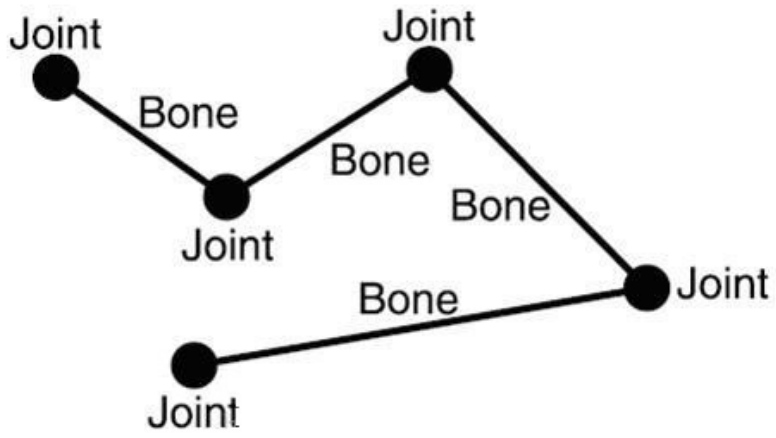

图 13.1: 骨头、关节

当执行骨骼动画时，不必担心关节，或骨头之间的结合点。每个顶点实际上是附着 (关联) 到

关节点的，而不是骨头。

根关节　根关节是一个模型中的**终端关节**。任何其它关节都以自己的路径最终关联到根关节。

任何对**根关节**的操作，如不论是平移或是旋转，都会影响到模型中的每个顶点。旋转或平移根关节将影响模型中其他所有的关节和顶点。

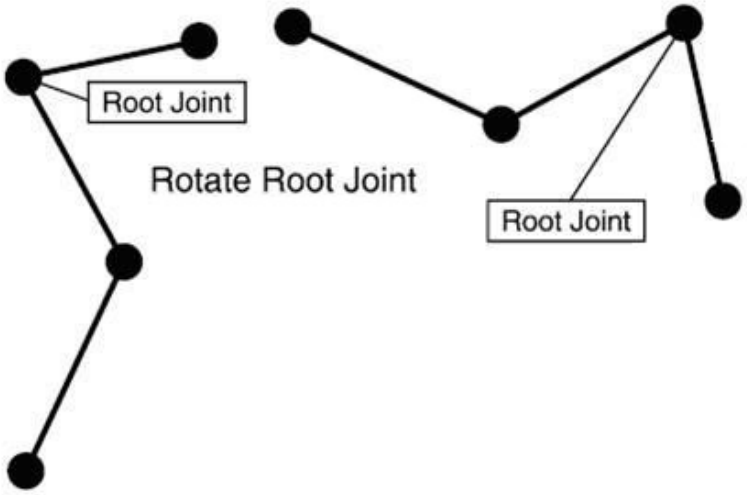

图 13.2: 根关节

在每个模型里**只有一个根关节**，它没有父关节。

根关节通常是许多骨头连接的地方，而不是需要一个小动画的地方。例如包括中部和下后部，但没有明确要求根关节一定要在模型中的某个准确位置。

父关节和子关节　一个关节可以有父关节和子关节。

父关节的旋转和平移会影响 计算当前关节的新位置。

再拿手臂的例子来说，肘关节是手掌的父关节。移动肘关节则影响手掌。在简单的骨骼动画里，每个关节只有一个父关节，如果有的话。

一个关节可以有许多子关节。任何你对父关节做的事都会渗透到子关节。

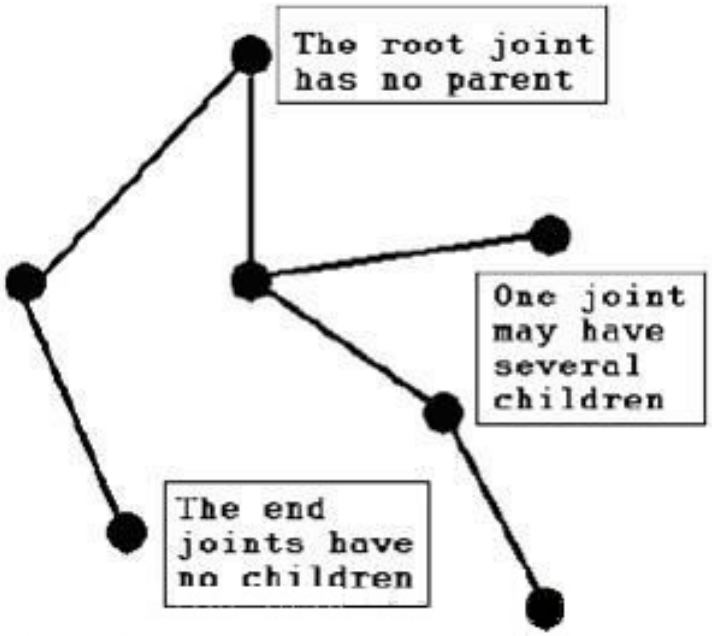

图 13.3: 父子关节

关键帧　关键帧是一个模型位置的瞬时状态。

不同于每个关键帧都包含其自身所有顶点的**拷贝的方式**，骨骼动画的关键帧或叫**骨骼帧 (boneframe)** 包含了旋转和平移的变换信息，一般平移是一个 X，Y，Z 值的形式，和三个分别包含了按 X，Y，Z 轴旋转的值。

如常规顶点关键帧一样，**这些骨骼帧必须被插值来提供平滑的动画效果**。

对弧形路径按两端点的直线方式而不是沿弧的路径方式进行插值就会产行“渗出”效果。解决这个问题的最好方法就是**采用四元数**来对旋转进行插值。

计算位置　如何更改它以使关节之间能正常运动。

首先你要做的就是创建一个**变换矩阵**，该矩阵用于每一个使用了不同旋转和平移关键帧数据的 (关键帧) 点。这个变换矩阵可以通过先**生成三个旋转矩阵** (译注:x，y，z 三个方向的旋转矩阵) 和**平移矩阵**。再将这三个矩阵相乘就会生成了最终的**变换矩阵**。

你也可以使用 matrix 类的 SetRotation 和 SetTranslation 函数来避免你自己做矩阵乘法。这个生成的矩阵叫做**相对矩阵 (relativematrix)**。

下一步需要计算出一个**绝对矩阵 (absolute matrix)**。绝对矩阵是关节的相对矩阵乘上 它的父关节的绝对矩阵得到的。绝对矩阵告诉你关节的绝对变换 (译注: 就是将关节的本地坐标变换为世界坐标)。这包括了自身的相对变换，除此之外，层次结构中所有在它之前的关节的变换

都已计算完成。这允许其他关节的移动作为移动关节链上游关节的结果。细想一下，例如，当你移动肩膀时肘又是怎么移动的，这引出了一个问题: 你如何计算最初的绝对矩阵? 根关节是没有父关节的，因此，它的绝对矩阵就是它的相对矩阵。

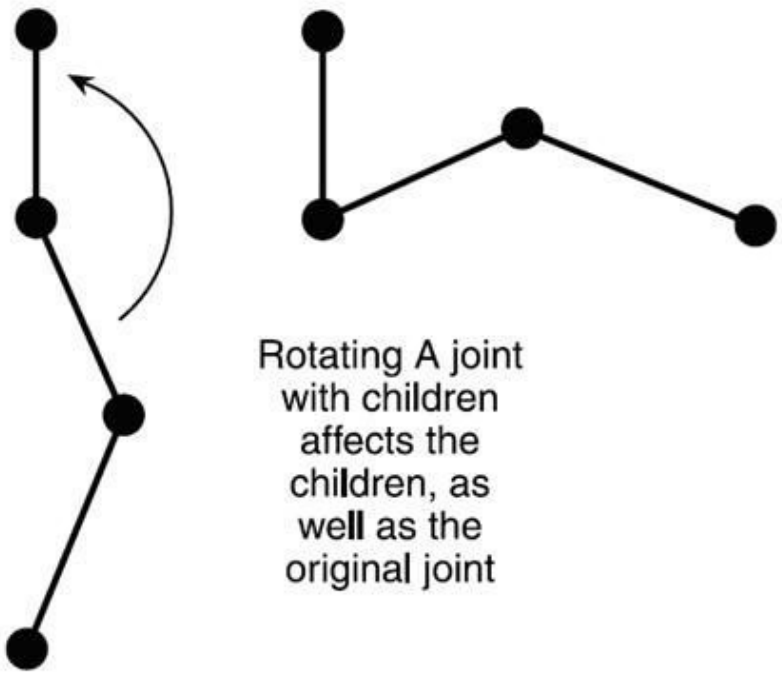

图 13.4: 旋转一个关节并影响其子关节群

如上图所述：遍历各关节，会顾及其所有前面的变换。哪怕只有一个关节要移动，在它下面的关节也得跟着移动，这很像移动你的臀部，则你的膝和踝也得跟着移动一样。

骨骼帧不会被累积，每帧保存了从起点起的特定关节的旋转和平移的信息。如果你不回退到原始顶点，那么当每次计算新顶点位置时，模型将会飘乎不定得运动。

将网格附着到骨骼 当关节已能平滑运动的时候，也是该将网格附着到骨骼的时候。

网格 (mesh) 组成了模型的形状，它是由一组使模型具有立体感和细节的顶点和三角形组成，没有网格，基于骨骼运动的模型只是一个简单的骨架。每个网格的顶点存储了一个指向关节数组的索引，用于指示它被附着到某根骨头。

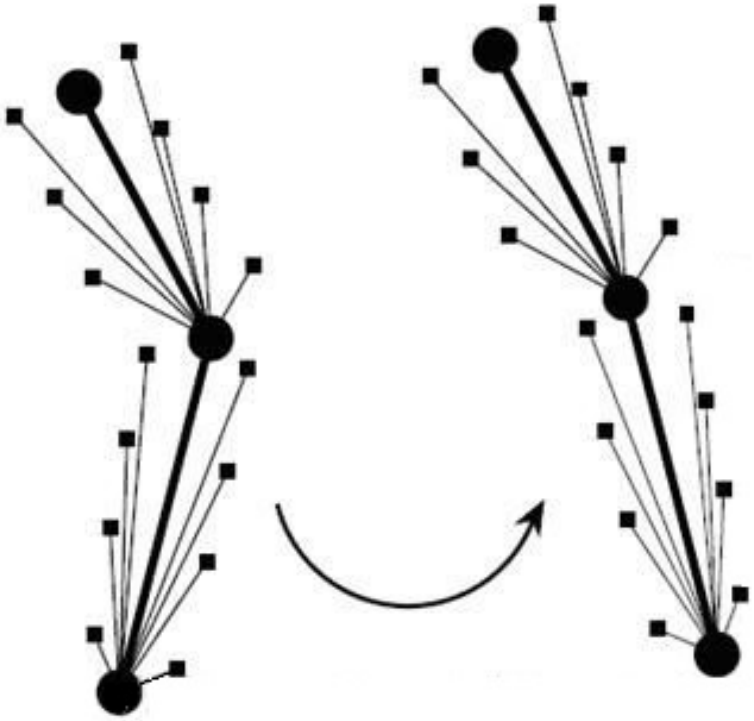

图 13.5: 顶点附着于关节

每个模型包含了一个**纹理坐标的集合**和一个**三角形信息的集合**。仅因为**顶点的位置变更**并不意味着**三角形顶点索引**和**纹理坐标**也会变化。这意思是说，当第一次设置好它们之后就根本不需要再担心它们。

法线则是另一回事了，因为多边形的朝向和顶点的位置更改了，因此法线也相应更改了。如果你正在使用面法线，那么你在每一帧将他们传给渲染器之前都需要重新计算。然而，如果你一开始就计算了顶点法线，那么你很幸运，顶点法线不必在变换后都完全重新计算，它们能利用和顶点相同的变换矩阵来进行变换。唯一不同的是，你不需要考虑平移。

另一个需要考虑的问题是，顶点如何附着到多于一根的骨头。此时，每根骨头将被赋予一个权重，这个权重决定它影响关节点的比重。最终的变换是所有这些被附着的骨头的变换的加权平均。

13.2 骨骼动画 UE4 基本概念

动画系统概念介绍：ref url:https://zhuanlan.zhihu.com/p/62401630

创建人物角色动画蓝图:ref url:https://blog.csdn.net/alzzw/article/details/104727955/

13.2.1 概述

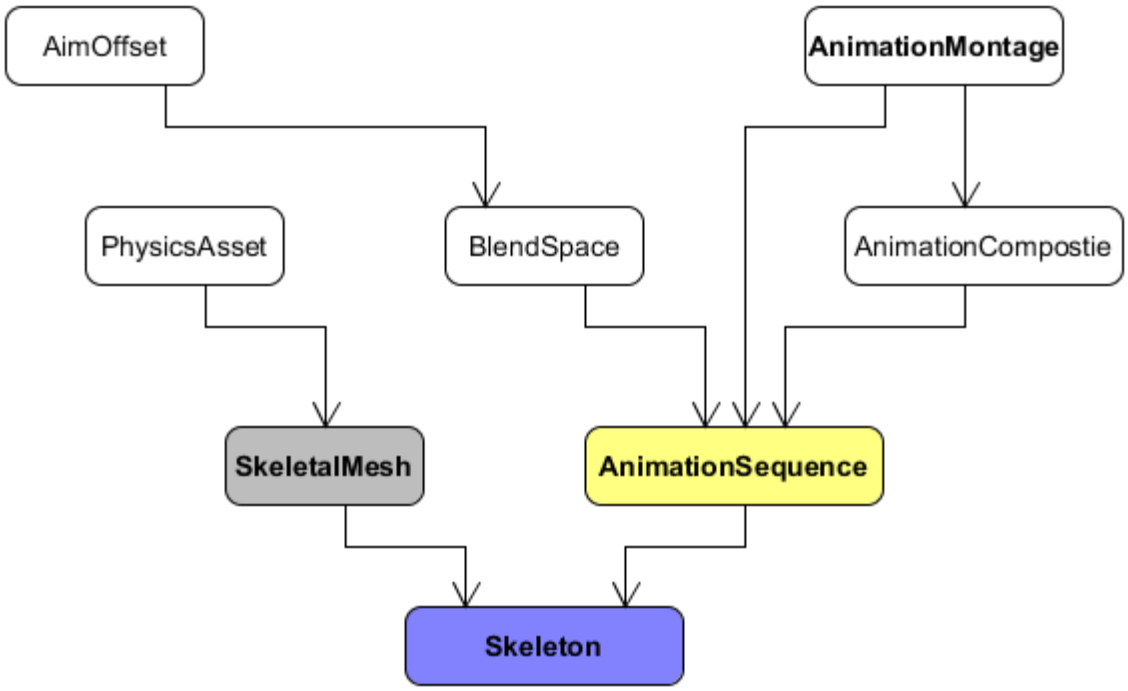

图 13.6: UE4 常见动画资源关系图

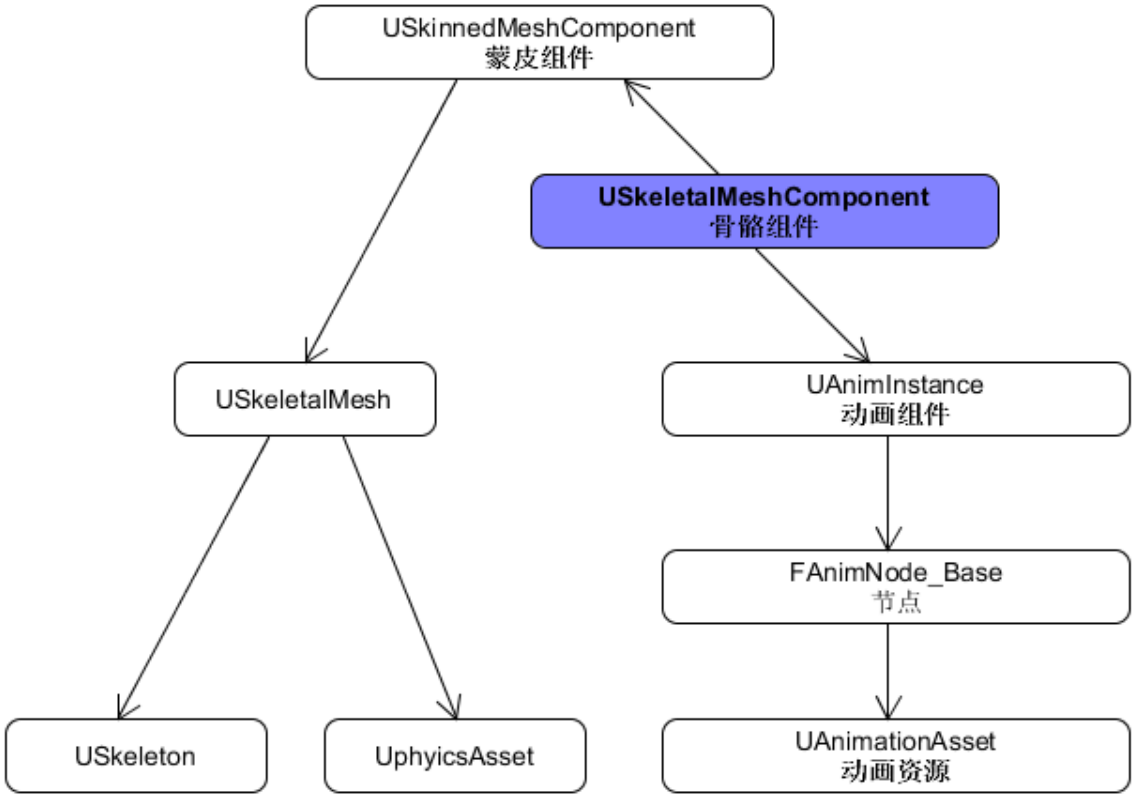

图 13.7: UE4 骨骼动画资源关系图

Skeleton 骨骼 是整个**动画系统的基础**，其主要作用是记录了骨架以下信息：

- USkeleton：`USkeleton` 并不会直接使用自身的数据而是会生成一个`FReferenceSkeleton` 来提供给 mesh 来使用。
- 骨骼层级信息:
- 参考姿势信息:
- 曲线（Curve）信息
- 动画通知（Animation Notify）信息
- 插槽数据（插槽名称、所属骨骼、Transform 等）
- 骨骼名称 Index 映射表、其他骨骼设置信息 (位移重定向（Translation Retarget）设置，LOD 设置等)

Skeletal Mesh 骨骼模型 骨骼模型是**在骨骼基础之上**的模型，通俗来说就是绑定骨骼后的网格体，也就是我们在游戏中能够看到的角色的**“肉身”**。

- USkeletalMesh: 链接 Mesh 和动画的桥梁
- 模型的顶、线、面信息
- 顶点的骨骼蒙皮权重
- 模型的材质信息
- 所属骨架
- Morph Target、Physics Asset、布料系统等设置

Animation Sequence 动画序列 是用来记录骨骼运动状态的资源，也是让动画动起来的关键之一，其主要包含了一下信息：

- 动画**关键帧**信息
- 包含运动的骨骼信息
- **每帧骨骼的 Transform 信息**：包含了骨骼的旋转、位移和缩放
- 动画通知信息：记录了触发通知的类型和时间
- 动画曲线信息：记录了随时间轴变化的曲线信息

- 其他基础信息：包括了叠加动画设置、根骨骼位移设置等信息

BlendSpace 混合空间

AimOffset AimOffset 是 *BlendSpace* 的一种特殊形式，区别在于其内部的所有采样点都是叠加动画。主要用在游戏过程中叠加角色瞄准和看的方向。

动作融合

叠加动画

AnimationMontage AnimationMontage（简称 Montage）是对Animation Sequence 等资源的扩充，可以方便的使用蓝图、代码控制动画资源的播放，并且可以方便的实现根骨骼动画和动画播放结束后的回调。

要使用Montage 就需要将相应的动画拖入其中，在Montage 编辑器中可以对动画进行编辑、裁剪、拼接等操作（详情请见官方文档）。

Montage 编辑器中有一个 Slot（插槽）的东西，通俗的解释就是，插槽是对应在动画蓝图中播放Montage 的**播放点**，同一个Montage 可以有多个不同的Slot。另外 Montage 的使用上还有 Section 等概念，具体使用方式在未来讲到动画系统搭建的时候会具体解释。

动画节点

分类

- FAnimNode_Base: 运行时的行为节点
- UAnimGraphNode_Base : 编辑器的图表节点
- FAnimNode_AssetPlayerBase: 播放动画的节点父类, 大部分工作都是对于UAnimSequence 资源类型的更新

核心函数

- Initialize_AnyThread：当节点第一次运行时，会运行Initialize_AnyThread。
- CacheBones_AnyThread：CacheBones 用于刷新该节点所引用的骨骼索引。

- `Update_AnyThread`：更新
- `Update_AnyThread`：调用以根据 Update() 中设置的权重计算局部空间骨骼变换。
- `FPoseLink SourcePose`; 这是在节点上的 pose 连结点，会串联出来基本的运行逻辑。基本上每个 animNode 都会。

连接点

- FPoseLink
- FPoseLinkBase

Animation Instance 这是所有逻辑的控制中心。

Animation Blueprint

13.2.2 角色动画：骨骼动画系统

大流程 大致如下：

1. **计算骨骼 Pose**

   每一根骨骼可以看做一个点，而 Pose 就是所有骨骼 Transform（位移 + 旋转 + 缩放）的集合，注意，一般来说，这个 Pose 是基于骨架参考姿势（Reference Pose）的**变换矩阵**。

2. **蒙皮**

   将 3D 美术制作好的网格体的顶点按照骨骼 *Pose* 进行变化。这里变换的依据是顶点的蒙皮权重和参考姿势的信息（一个顶点可能受到多个骨骼的影响）。

更新流程 UE4 里主要承担这两步职责的是`Skeletal Mesh Component`。

`SkeletalMeshComponent` 继承自 `SkinnedMeshComponent`。在`SkeletalMeshComponent` 创建时，会创建一个`Animation Instance`, **这就是主要负责计算最终 Pose 的对象**，而我们制作的 `AnimationBlueprint` 也是基于 `UAnimationInstance` 这个类的。

在`SkeletalMeshComponent` 进行 `Tick()` 时，会调用 `TickAnimation()` 方法，然后会调用 `Animation Instance` 的 `UpdateAnimation()` 方法，此方法会调用一遍所有动画蓝图中连接的节点的 `Update_AnyThread()` 方法，用来**更新所有节点的状态**。

然后后续 根据设置的不同会从 Tick() 函数或者 Work 线程中调用 SkeletalMeshComponent 的 RefreshBoneTransforms() 方法,此方法进而会调用动画蓝图所有节点的 Evaluate_AnyThread() 方法。

Evaluate 的含义就是指 根据特定的条件（从 *Update()* 时获得的参数）计算出动画**所有骨骼的 Transform 信息**，最后 **输出一个 Pose 给到渲染线程**并存在本地 Component 上。

简化下的流程如下：

- Init() 初始化
- Update() 动画蓝图从游戏逻辑中收集状态变量并更新骨骼位置
- Evaluate() 根据骨骼位置对动画进行解压和混合
- Complete() 将运算后的顶点数据推送到渲染现场，更新物体位置和动画通知

AnimInstanceProxy 属于多线程优化动画系统的核心对象。可以分担动画蓝图的更新工作，**将部分动画蓝图的任务分配到其他线程去做**。

13.2.3 IK

13.2.4 其他

13.3 流程

13.4 深入

第十四章　粒子系统

第十五章　资源管理

第十六章　Editor 扩展

第十七章　跨平台发布 apk

第十八章　Profile

第十九章　UE4 优化

第二十章　调试技巧