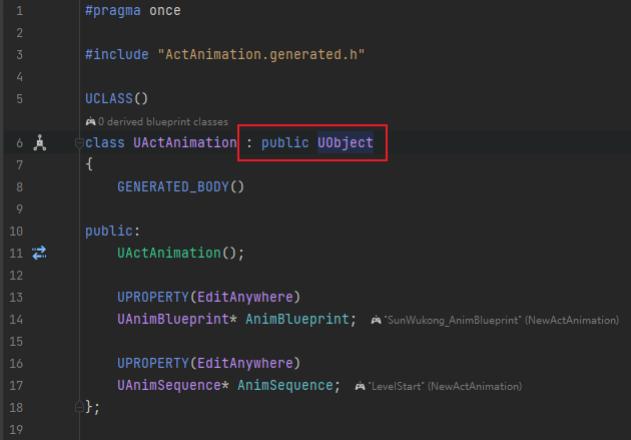
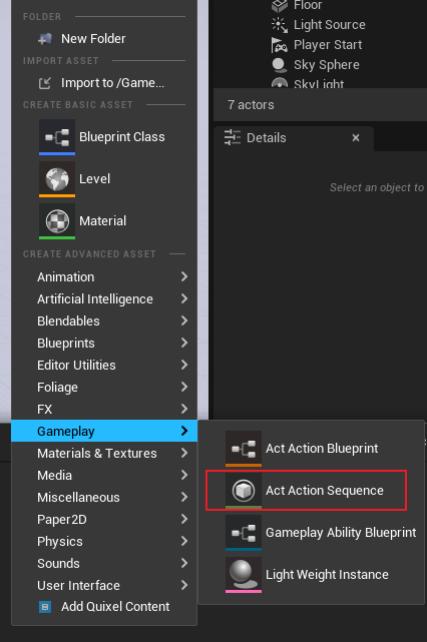
1.首先必须明确，自定义资源类型必须派生自 UObject 。仅 C++ struct 或 class 无法被视为 UE 认可的资源类型。例如：下图自定义了一个资源类型叫 UActAnimation 。现在这个自定义的资源类型还只能够被代码 使用，或作为蓝图的基类编辑，还无法使用自定义的编辑器。接下来需要将其注册到 ContentBrowser 的 Add 菜单中，使其可以自由创建和自定义编辑器。



2. 创建自定义资源类型的包装，派生自 FAssetTypeActions\_Base 。这个包装包含了创建这个资源的—些行为和信息。也可以根据 UE 已实现的资源类型去继承，例如：蓝图可以继承 FAssetTypeActions\_BIueprint 。贴图可以继承 FAssetTypeActions\_Texture 。在 UE 源码中搜索 ": pubIic FAssetTypeActions\_Base" ，即 可找到所有该类的派生类型，后面也会不断使用这种方式来找到一个基类的派生类。

这个包装需实现虚函数 OpenAssetEditor (打开自定义资源编辑器的回调)、 GetName (本地化资源名称)、GetTypeCoIor ( ContentBrowser 中资源的显示颜色)、 GetSupportedClass (资源类型支持的具体 UClass 派生类型)、 GetCategories ( Add 菜单中所处的分类)等，具体虚函数作用可以查看 FAssetTypeActions\_Base 基类中的注释。



3. 在一个合适的地方，例如： Module 的 StartupModule 函数中，注册该包装类，同时记得要在另一个合适的地方，例如： ShutdownModule 中，反注册该包装类，这就需要在注册之后把实例缓存下来。具体代码这里不过多展开。

|  |
| --- |
| StartupModule(...)  ...  IAssetTools& AssetTools = FModuleManager::LoadModuleChecked<FAssetToolsModule>("AssetTools").Get();  TSharedRef<IAssetTypeActions> RegisteredAssetType = MakeShareable(new FAssetTypeActions\_YourCustomized());  AssetTools.RegisterAssetTypeActions(RegisteredAssetType);  // Add RegisteredAssetType to some cache  ...  ShutdownModule(...)  ...  AssetTools.UnregisterAssetTypeActions(RegisteredAssetType); |

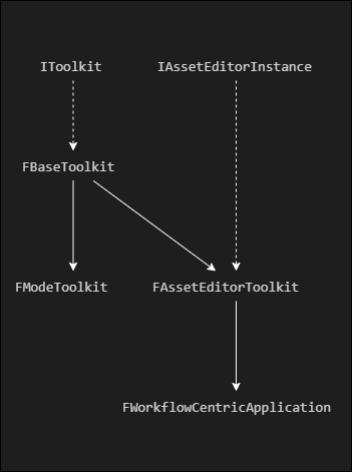
4. 因为 UE 的资源类型都是通过 UFactory 工厂创建出来的，除了上面的资源类型包装外，还需要资源类型对应的工厂，具体代码可以参考 [UAssetToolsImpl::CreateAsset] 。所以还需要一个派生自 UFactory 的自定义类型，并在构造函数中设置这个工厂支持的 class 类型，实现 FactoryCreateNew 虚方法， FactoryCreateNew 中传入了 NewObject 所必须的—些参数，包括 Outer (这里—般是 UPackage ，即这个 UAsset 所在的资源包)、 lnName (资源名称)、 FIags (资源标签) ，返回已创建好的自定义资源实例，以供编辑器传入和编辑。

|  |
| --- |
| UActAnimationFactory()  ...  bCreateNew = true;  bEditAfterNew = true;  SupportedClass = UActAnimation::StaticClass();  ...  FactoryCreateNew(...)  ...  return NewObject<UActAnimation>(InParent, InName, Flags); |

5. 回到 [3.] 中创建的自定义资源类型包装，其中实现了 OpenAssetEditor 方法，其参数 lnObjects (编辑器待编辑的资源实例列表，这里是可以多开或—开多来编辑的)、 EditWithinLeveIEditor (是否在 LeveIEditor 中对应编辑，—般独立的编辑器像 Persona 这里都是 nuIIptr )。可以在此处对传入的资源使用对应的编辑器来打开。于是就可以开始自定义一个编辑器以及其 Ul 了。例如：这里使用 FNovaActEditor 编辑器来打开这个资源实例。 CreateEditorWindow 是自定义的方法，但是自定义编辑器依然有规则需要遵守。

|  |
| --- |
| OpenAssetEditor(...)  ...  for (auto ActAnimation : InObjects)  {  TSharedPtr<FNovaActEditor> NovaActEditor(new FNovaActEditor(ActAnimation));  NovaActEditor->CreateEditorWindow(EditWithinLevelEditor);  ...  }  ... |

6. 这里要介绍几个自定义编辑器相关的接口和类型：



|  |  |  |
| --- | --- | --- |
| 名称 | 作用 | 相关派生类 |
| IAssetEditorInstance | 资源编辑器的实例接口，只要被纳入监听的编辑器(已打开、已关闭等事件)都会实现这个接口，这里不关心编辑器的实现，应该只是用来给某个 Manager 做统—管理的。 | FAssetEditorToolkit 、UAssetEditor 、SMiniCurveEditor |
| IToolkit 、  FBaseToolkit | Toolkit 的接口，Toolkit 的概念即—系列资源编辑器和相关资源的集合，例如： Persona 就是一个 Toolkit , FBaseToolkit 是 IToolkit 的抽象实现基本可以视为—体的。 | FAssetEditorToolkit 、FModeToolkit |
| FAssetEditorToolkit | 大体上单独资源的编辑器都可以从这个基类继承出去，这里包含的 InitAssetEditor 方法，可以对待编辑的资源以及当前的编辑模式、当前的 Layout 等—些 Ul 属性做统—处理。 | 派生类太多，仅列举—些：FPropertyEditorToolkit 、FSimpIeAssetEditor 、FWorkfIowCentricAppIication |
| FModeToolkit | 自定义程度更高的 Toolkit ，只提供了简单的 Panel ，没有指定当前编辑资源，适合需要更多自定义的场景，例如：使用其他场景资源或编辑器内部构造资源等。 | 仅列举—些：FTiIeMapEdModeToolkit 、FMeshPaintModeToolkit |
| FWorkfIowCentricAppIication | 集成度更高的 Toolkit 类型，目标可能是取代某个 DCC 工具，内部工作流是闭环的不依赖其他外部工具。 | 仅列举—些：IBehaviorTreeEditor 、IBIueprintEditor |

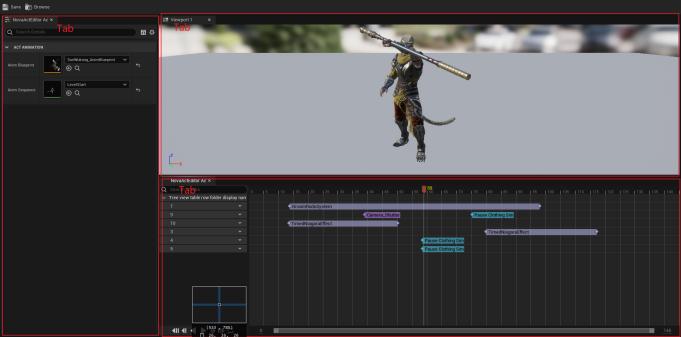
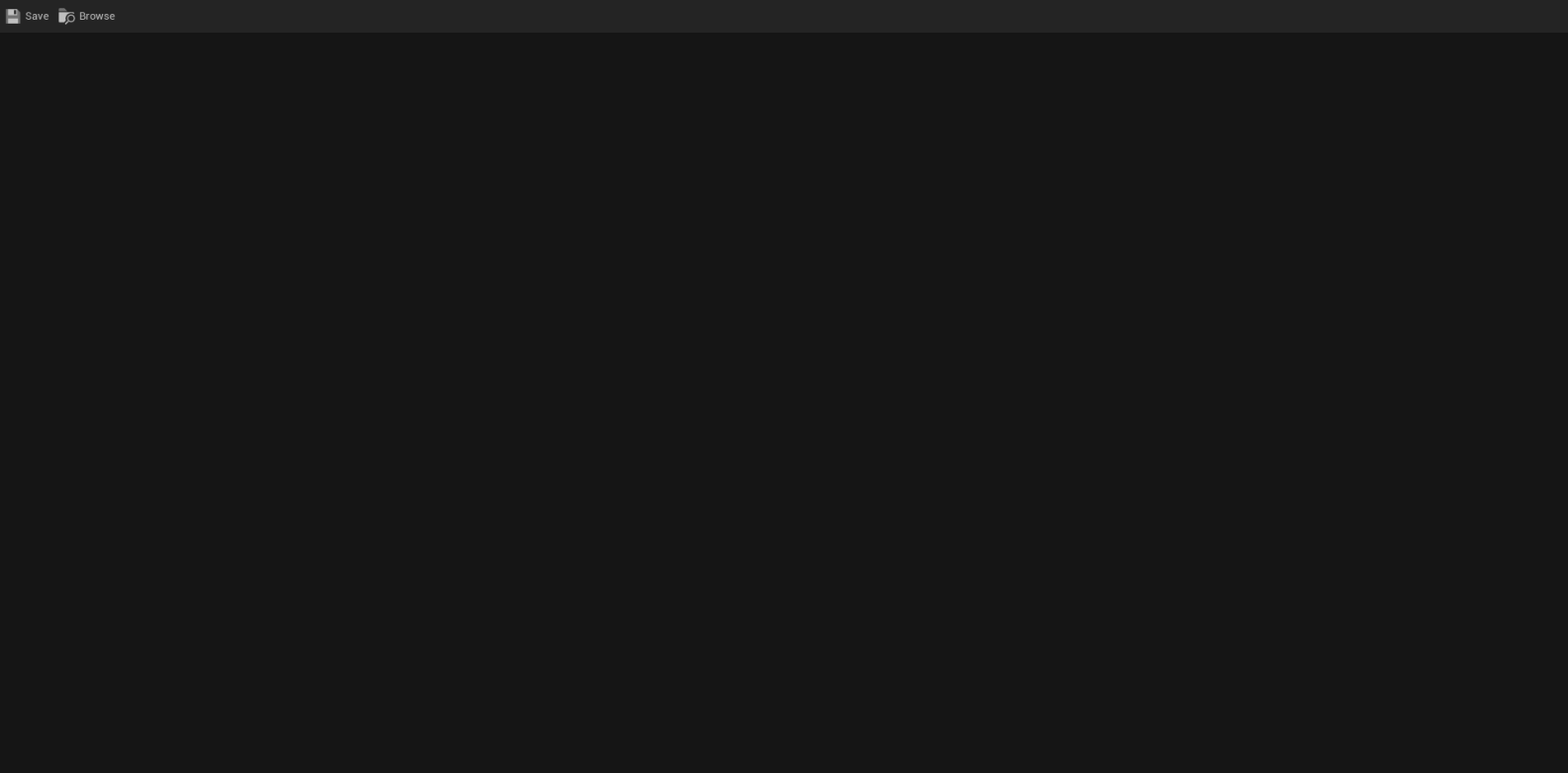
7. 这里选择了派生自 FWorkfIowCentricAppIication ，但是实际上—些简单的编辑器也可以派生自

FAssetEditorToolkit 就够了。在 [5.] 中提到的 CreateEditorWindow 方法中，必须调用基类的

InitAssetEditor 函数，其中参数 Mode (StandaIone/WorIdCentric ，独立 or LeveIEditor 附着)、InitToolkitHost (OpenAssetEditor 参数传入的 IToolkitHost)、 Appldentifier (应用名称)、

StandaIoneDefauItLayout (默认的 Layout )、 bCreateDefauItStandaIoneMenu (拥有默认独立菜单)、 bCreateDefauItToolbar (拥有默认工具栏)、 ObjectToEdit (待编辑的资源实例)。这里重要的就是传入ObjectToEdit ，这样编辑器能够通过 GetEditingObject 获得当前正在编辑的资源实例。这大部分情况都是基类内部在使用的，而我所选择的实现是将数据实例等都放到 DataBinding 中通过 Key[String] 的方式来获取。

|  |
| --- |
| CreateEditorWindow(...)  ...  // Initialize the asset editor  InitAssetEditor(EToolkitMode::Standalone, // Mode  InIToolkitHost, // InitToolkitHost  NovaActAppName, // AppIdentifier  FTabManager::FLayout::NullLayout, // StandaloneDefaultLayout  true, // bCreateDefaultStandaloneMenu  true, // bCreateDefaultToolbar  ActAnimationDB->GetData()); // ObjectToEdit  ... |

8. 有了这个编辑器 Window 之后，还需要—些 Tab 来承载编辑器的界面每个子区域，如下图所示，实际上如果编辑器打开没有 Tab 的话就会变成后头这样。没有 Tab ，仅光秃秃的编辑器 Window 如下

9. 最初我直接使用了 FAssetEditorToolkit 的 RegisterTabSpawners 来添加 Tab ，这个函数参数lnTabManager 会把 FTabManager 传入给你方便去注册自定义的 FOnSpawnTab 回调方法。但是这样注册的 Tab 在实际使用时却遇到了关闭 Tab 后无法再重新打开，并且关闭后 Window 菜单栏还是会显示那个Tab 处在打开状态，在研究了—阵没有结果后，转到了同 FAnimationEditorMode —致的注册 Tab 的方法。即自定义—堆类似 Tab 工厂的类型，再使用一个 FWorkfIowAllowedTabSet 将这些工厂类都实例化并注册进去，这样这些 Tab 就可以正常的打开和关闭了。自定义一个 AppMode 派生自 FAppIicationMode ，并实现虚方法 RegisterTabFactories ，这个方法的实现比较单一，大致就是将实例化好的 Tab 工厂 Push 到对应的 FWorkfIowCentricAppIication 中去，以便 Editor 能够识别和管理这些 Tab 。同时，在构造函数中填充 FWorkfIowAllowedTabSet 结构，以及 TabLayout 信息。

|  |
| --- |
| FNovaActEditorMode(...)  ...  TSharedRef<FActAssetDetailsTabSummoner> ActAssetDetailsTabSummoner = MakeShareable(new FActAssetDetailsTabSummoner(InWorkflowCentricApplication));  TabFactories.RegisterFactory(ActAssetDetailsTabSummoner);  TabFactories.RegisterFactory(MakeShareable(new FActViewportSummoner(InWorkflowCentricApplication, )));  TabFactories.RegisterFactory(MakeShareable(new FActViewportSummoner(InWorkflowCentricApplication, 1)));  TabFactories.RegisterFactory(MakeShareable(new FActViewportSummoner(InWorkflowCentricApplication, 2)));  TabFactories.RegisterFactory(MakeShareable(new FActViewportSummoner(InWorkflowCentricApplication, 3)));  TabLayout = FTabManager::NewLayout("Standalone\_NovaActEditor")->AddArea(FTabManager::NewPrimaryArea());  ... // 此处略去具体 Layout 排版  RegisterTabFactories(TSharedPtr<FTabManager> InTabManager)  ...  TSharedPtr<FWorkflowCentricApplication> WorkflowCentricApplication = WeakWorkflowCentricApplication.Pin();  WorkflowCentricApplication->RegisterTabSpawners(InTabManager.ToSharedRef());  WorkflowCentricApplication->PushTabFactories(TabFactories);  ... |

10. 前面代码中提到的 FActAssetDetaiIsTabSummoner 、 FActViewportSummoner 等类型，都是派生自FWorkfIowTabFactory 的类型，同时，必须实现 CreateTabBody 方法，不管是创建一个新的 Widget 还是使用构造函数中预先创建好的 Widget ，都必须返回一个可以给 Tab 使用的 Widget 视为 Tab 的内容。例如： FActAssetDetaiIsTabSummoner 使用在构造函数中预先创建好 Widget 的方式。同时，这里还可以初始化—些 Tab 相关的信息，如 LabeI 、 lcon 等。要说这相关信息也挺混乱的，有些是通过字段赋值来填充，而 TooltipText 却是通过重载 GetTabToolTipText 方法来填充。

|  |
| --- |
| FActAssetDetailsTabSummoner(...)  ...  TabLabel = LOCTEXT("ActAssetDetails", "NovaActEditor ActAssetDetails tab name.");  TabIcon = FSlateIcon(FEditorStyle::GetStyleSetName(), "LevelEditor.Tabs.Details");  bIsSingleton = true;  ViewMenuDescription = LOCTEXT("DetailsDescription", "Details");  ViewMenuTooltip = LOCTEXT("DetailsToolTip", "shows the details tab for selected objects.");  ActAssetDetailsWidget = SNew(SActAssetDetailsWidget);  ...  CreateTabBody(...)  ...  return ActAssetDetailsWidget.ToSharedRef();  ... |

11. 有了 [10.] 的 Tab 后，就可以在 [7.] 中 CreateEditorWindow 加入—段代码，给这个 Mode 取一个响亮的名字，并把这个 AppIicationMode 注册到 FWorkfIowCentricAppIication 中。

|  |
| --- |
| CreateEditorWindow(...)  ...  FName NovaActEditorMode = FName(TEXT("NovaActEditorMode"));  AddApplicationMode(NovaActEditorMode, MakeShareable(new FNovaActEditorMode(SharedThis(this))));  SetcurrentMode(NovaActEditorMode);  ... |

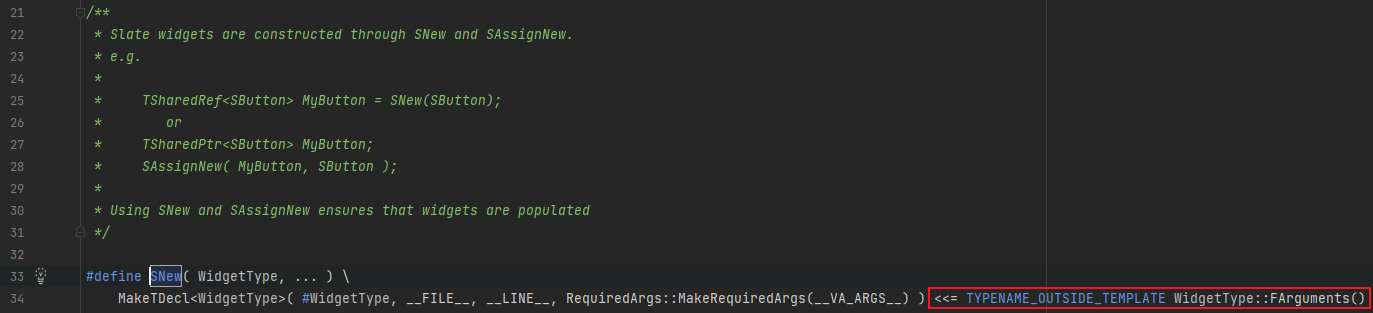
12. 那么，如果不想通过这种麻烦的套娃工厂方式去注册 Tab 的话，也可以直接在FWorkfIowCentricAppIication 的 RegisterTabSpawners 中去写类似这种代码，这就是之前提到的，最初使用的直接添加 Tabld 和对应回调函数的方式。但是，这种方式有个没解决的问题就是关掉 Tab 后，除了ReIoad DefauIt Layout 就没有其他方法能够再打开那个 Tab 了，而且 Window 菜单栏中的 Tab 状态也不 对。具体回调函数的内容这里不过多的展开，大致就是根据 FSpawnTabArgs 结构，去创建 SDockTab

Widget 并返回。猜测这里返回的 DockTab 应该不是自动管理的，可能还需要手写—些管理方法，即关闭后如何重新打开，关闭时设置关闭状态等；而且这里没有填充 Tab 的内容，也需要通过代码来填充；还需要注意这种方式创建 SDockTab 是发生在 InitAssetEditor 的调用内部，即 CreateEditorWindow 还没有完成调用，如果 Tab 对应内容 Widget 必须在 InitAssetEditor 之后调用，则会出现代码时序问题，需要想办法绕过去。总之就是比较繁琐。

|  |
| --- |
| RegisterTabSpawners(...)  ...  auto OnSpawnTabRaw = FOnSpawnTab::CreateRaw(this, &FNovaActEditor::OnActEventTimelineTabSpawn);  // \*\* NOTE:记得反注册中也要添加对应方法  FTabSpawnerEntry& EventTimelineEntry = InTabManager->RegisterTabSpawner(FName(TEXT("ActEventTimeline")), OnSpawnTabRaw);  EventTimelineEntry.SetMenuType(ETabSpawnerMenuType::Hidden);  ... |

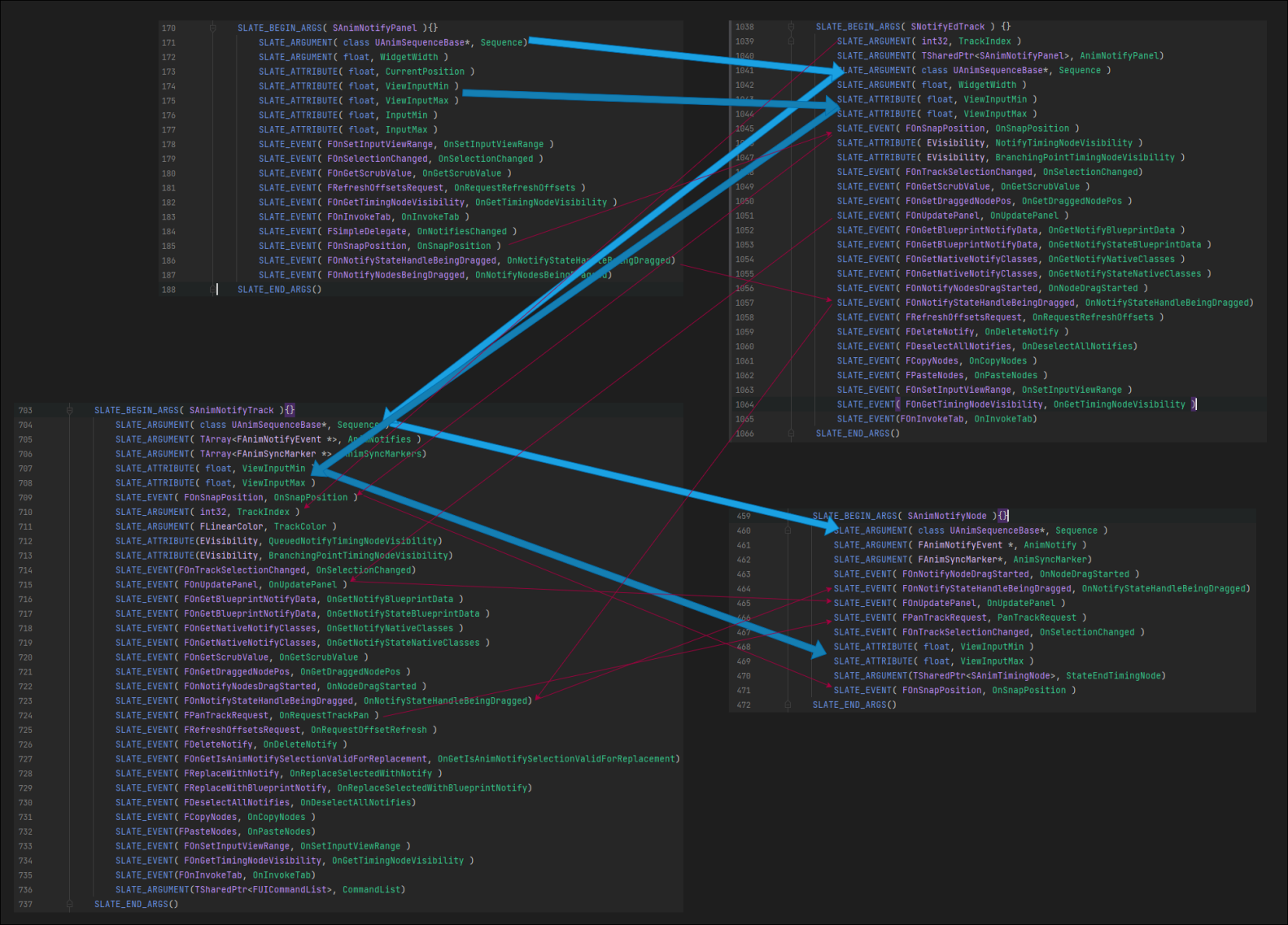
13. 有了 Tab 后，就可以随意填充 Tab 对应的 Widget 内容了。这里实际上就是各种 SIate 写法。说到看了几个月的 SIate 代码，不得不吐槽的两点：

|  |
| --- |
| class SAnimTimelineSplitterOverlay : public SOverlay  ...  // \*\* 用于把自身参数转换为SSplitter的参数，  // \*\* 使其可以 + SSplitter::Slot  typedef SSplitter::FArguments FArguments;  ...  Construct(const FArguments& InArgs)  ...  // \*\* 因为在 SNew 中的参数将作为 Construct 的第二个参数开始传入，那么要外部传入 Construct 的第一个参数就得另辟蹊径，于是就诞生了这个奇怪的连续赋值，实际上 SNew(SSplitter) = InArgs 是给构造函数传入第一个参数！！  // \*\* 备注：UE 5.3+ 新版已经不支持这种连续赋值形式了，这也能侧面代表 UE 的代码确实越改越好了。  Splitter = SNew(SSplitter) = InArgs;  ... |



出现这种写法感觉就像是某天某个程序觉得：这个 UI 组件，必须有 SOverlay 的功能，除此之外它又长得跟组件 SSplitter 一模一样，但是不能同时继承自两个类吧，那得想办法把另一个给组合进来，噢，但是得把参数配置成 SSplitter 的参数！这真是个奇行种，一只披着羊皮的狼要生下一头羊@#!@#$，还得给它造点语法规则才行。

这里除了奇怪的连续赋值以外，还打破了定义 SWidget 的常规宏 SLATE\_BEGIN\_ARGS() { } ... SLATE\_END\_ARGS() ，采用巧夺名义的方式，直接把别人定义好的 FArguments 视为自己的 FArguments。可谓十分的精妙，十分的打脸。不禁想到那句：规则都是用来打破的，此谓其一。



SlateUI 参数的精髓是什么？看了大段大段的相关代码后领悟到 Slate 的思考和背后对参数的认识：每一个 Widget 组件，不管其实现如何贴近对应功能模块，贴近父 Widget 的实现，但是，参数必须是独立的，这样保证了当你单独在用任何一个 Slate Widget 时，都明白要传入什么参数(但是不明白参数从哪来往哪去)，万一哪天有人突然想直接用这个 Widget 呢？嘿嘿， Widget 的参数就是降低耦合性的核心！父层级的 Widget 有什么用？直接忽略，不要影响子 Widget 。这样样子每一个 Widget 就都重新开始定义自己的存在、定义自己的参数列表，都是独立可复用的个体了！

|  |
| --- |
| SAnimTrackPanel::Construct(const FArguments& InArgs)  ...  ViewInputMin = InArgs.\_ViewInputMin;  ViewInputMax = InArgs.\_ViewInputMax;  ...  SNotifyEdTrack::Construct(const FArguments& InArgs)  ...  // Notification editor panel  SAssignNew(NotifyTrack, SAnimNotifyTrack)  .Sequence(Sequence)  .TrackIndex(TrackIndex)  .AnimNotifies(Track.Notifies)  .AnimSyncMarkers(Track.SyncMarkers)  .ViewInputMin(InArgs.\_ViewInputMin)  .ViewInputMax(InArgs.\_ViewInputMax)  ...  SAnimNotifyTrack::Construct(const FArguments& InArgs)  ...  ViewInputMin = InArgs.\_ViewInputMin;  ViewInputMax = InArgs.\_ViewInputMax;  ...  SAnimNotifyNode::Construct(const FArguments& InArgs)  ...  ViewInputMin = InArgs.\_ViewInputMin;  ViewInputMax = InArgs.\_ViewInputMax;  ... |

这种层层透传式的参数传递，为反反复复（考记忆）查找一个回调或者参数到底来自哪里，埋下了伏笔。相信当最终发现除了 SAnimNotifyTrack 外，没有其他代码使用 SAnimNotifyNode 时，这段代码读起来是多么的令人惊讶。

从好的方面来说，这段代码为以后可能的复用 Widget 打了个底子；从坏的方面来说，去看看这个逻辑这个参数从哪里来，需要上下查找 2 至 4 处不同的位置，还得硬记它们的前后关系，对阅读和理解代码是雪上加霜的。恰恰有趣的是，前者于我们无益，因为大概率你无法复用这个 Widget 去完成一个自定义界面（怕后续官方改了功能），你真正需要的是模仿这个 Widget 的样式去实现自己的 UI 功能，需要的是理解他的代码；而后者于我们有害，去趟厕所的功夫回来可能就得重读，或者找来找去连续使用 IDE 功能（回到之前光标标记处、进到下一个光标标记处、查看定义\实现、查找引用）按得头都晕了，代码也是晕的。相信每个读代码的人都能体会这种郁闷，此谓其二。

如果你觉得这篇文章对你有帮助，别忘了点个赞哦！我的 GitHub 项目[https://github.com/basstal/PlaySlate]中有这篇文章提到的内容的具体实现。欢迎访问和星标我的仓库，共同进步！