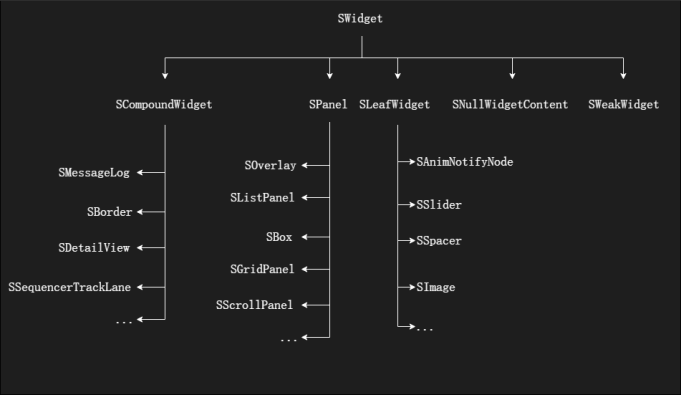
14.承接上文，讲完 Slate 的槽点以后，简单介绍⼀下 Slate 的⼀些基础知识：



|  |  |  |
| --- | --- | --- |
| 名称 | 作⽤ | 派⽣类型 |
| SWidget | 所有 Slate Widget 的基类，UE 不建议直接继承这个类型，建议如果没有⼦节点，可以继承 SLeafWidget，有⼦节点则继承 SPanel | SCompoundWidget 、 SPanel 、  SLeafWidget 、  SNullWidgetContent 、 SWeakWidget |
| SCompoundWidget | ⼤部分只有⼀个⼦节点的 Widget 都派⽣⾃这个类型，比较常⽤的有 SBorder (带有 BorderImage 属性的单⼦节点 Widget )、 SDetailView ( UObject 反射属性查看使⽤的 Widget )、 SMessageLog (消息提⽰) | SBorder 、  SDetailView 、  SMessageLog 、  SSequencerTrackLane 等 |
| SPanel | SPanel 中有个关键函数 OnArrangeChildren 其参数 AllottedGeometry (代表 SPanel 的 Geometry 信息)、ArrangedChildren ( SPanel 的 Child Widget 都需要通过 AddWidget 添加到这个结构中)， SPanel 的⼦类通过重载这个函数，使其具有不同的分配⼦节点⽅式。 | SListPanel 、 SBox 、  SGridPanel 、SScrollPanel 、SOverlay |
| SLeafWidget | 该 Widget 不能添加⼦节点，比较简单，只做最终⼀个区域的绘制⽤ | SSlider 、 SSpacer 、 SImage 、  SAnimNotifyNode |
| SNullWidgetContent | 没有任何⼤⼩的空 Widget，也不能够添加⼦节点，⼀般使⽤ SNullWidget::NullWidget 静态字段来获取，并作为参数给需要 SWidget 的函数占位⽤ |  |
| SWeakWidget | ⼀般来说⽗节点对⼦节点的引⽤都是强引⽤，保证⼦节点不会被 GC ，若⼦节点是 SWeakWidget ，则⼦节点⽣命周期不受⽗节点影响，官⽅的例⼦是 Hover ⼀个 Widget 显⽰ tooltip 时，弹出的 Window 对于这个被 Hover 的 Widget ⽽⾔必须是 SWeakWidget |  |

15.在掌握了 Slate 定义了哪些 Widget 后，可以开始进⼀步了解如何写⾃定义的 Slate Widget，⾸先挑选⼀个上文提到的类型作为基类，当然如果⻓相⼀致也可以考虑继承那些由上文类型派⽣的类型，例如想快 速建立⼀个树状结构的列表，则直接继承⾃ STreeView ，下⾯就拿这个来举例：

|  |
| --- |
| class SActImageTreeView : public STreeView<TSharedRef<SActImageTreeViewTableRow>> |

STreeView 模板要求 ItemType 是⼀个指针类型，这⾥统⼀使⽤ SharedRef 的⽅式，来确保这个指针的内容不会为空。接下来是具体类型，这⾥例⼦中选择的是派⽣⾃ SMultiColumnTableRow ，也可以派⽣⾃其他更简单的 SWidget ，例如每个 Item 只需要⼀列的话可以直接派⽣⾃ STableRow 等。因为例子中界⾯设计上需要兼容⼀个 TreeView 的 Item 就将多列的内容包揽在内，所以⽤ SMultiColumnTableRow ，通过重载 GenerateWidgetForColumn 能够根据参数 InColumnName 来区分每列需要创建的内容 Widget。注意到模板也是需要传入 SharedRef ，不过这次指针对象类型可以是⾃⾝，这样表⽰对应的 Model 数据也来⾃ Widget。

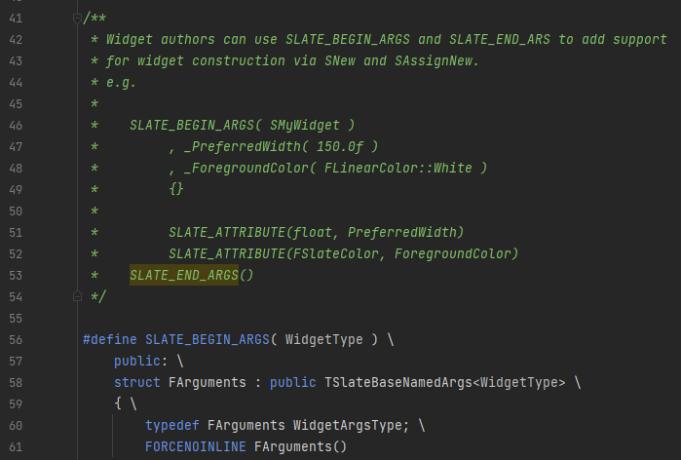
|  |
| --- |
| class SActImageTreeViewTableRow : public SMultiColumnTableRow<TSharedRef<SActImageTreeViewTableRow>> |

这⾥就顺带说⼀下，看到很多 Widget 代码中都记录了 SharedPtr<XXXModel> 有些甚⾄还有辅助的 class 来记录或转发控制信息，起初以为是 MVC 的设计，但是后来仔细看了更多同类代码后，发现并不是这样，⼤部分 Model 出现的地⽅都是做 Widget 的接⼝，实际上 Widget 所控制的 Model 会是这个 XXXModel 的派⽣类，然后派⽣类再去实现⼀些具体的设计，这样可以把⼀些互相之间⽆关的类型放到⼀个 Widget 中做显⽰和数据传递。例如： FAnimTimelineTrack 、 FAnimModel 这样的，实际上它们的派⽣类会有很多不同的数据需要处理，不同的显⽰绘制⽅式，但是 Widget 可以只使⽤ FAnimModel 就完成所有的事情。 UE 做了很多接⼝和实现分离的事情，有些看起来是必须的，但是有些却显得有点多余（猜测可能是为了将来的扩展考虑，或是旧代码没有人整理），⼤⾯积的这样做存在的⼀个问题，就导致了函数的参数列表分给不同的实现去使⽤的情况，例如：参数1给 A 实现⽤，B 实现只⽤参数2，C 实现⼜⽤到参数1和3，进而导致⼀些⽆谓的传参和看函数参数无法快速理解其实现效果。在 Delegate 中这种情况就更是屡⻅不鲜了，因为之前说过的 Widget 将⾃⼰的参数不断往下层 Widget 传递，这些定义好的 Delegate 函数也在往下传，如果中间有任何 Widget 需要往 Delegate 中添加参数使之往上层传，都要加到 Delegate 定义的参数列表中，这样传了⼏次后，每个 Widget 都可能加了⼏个不同的参数，就导致了参数多⽽实际⽆⽤的情况出现。

16.刚才也算是看代码后的⼀段题外话，还是回到 Slate Widget 的写法来，在继承了理想的基类后，需要明确定义⼀个 FArguments 。这个结构⽬前有两种⽅式来定义，常⻅的是使⽤宏来定义，通过查看

DeclarativeSyntaxSupport.h 文件中宏定义可以看到，宏的作⽤是定义⼀个 class 内部的 struct FArguments ，同时给这个参数列表加了很多默认的字段上去，这些个默认字段是支持传参给 SWidget 基类使⽤。

|  |
| --- |
| ...  SLATE\_BEGIN\_ARGS(SActImageTreeView) { }  ...  SLATE\_END\_ARGS()  // #define SLATE\_END\_ARGS() \  // };  ... |



~~那么顺⽔推⾈，另⼀种⽅式就是直接将宏的内容给复制出来，~~并不是/dodge。另⼀种⽅式之前也提到过，就是直接顺⼿牵⽺，巧夺名义。 typedef 其他 Widget 的参数作为⾃⼰的参数，其实这样如果刚好也是继承⾃那个 Widget 的话，就基本⼀模⼀样了。那么想要不⼀样的话，就得从构造开始⼀步步改变⾃⾝。 Slate Widget 很少⽤到构造函数，所谓的 Slate 构造都是以名为 Construct 的函数开始。这个函数也必须将第⼀个参数定义为 const FArguments& ，原因就在 Slate 的构造宏 SNew 和 SAssignNew 中。这些宏改造了第⼀个参数的传入过程 (也是前文提到奇怪的连续赋值的始作俑者)，使得可以直接在构造宏后边接 Slate 声明在参数列表中的参数，修改这些参数的值，还可以直接操作这个 Widget 的 Slot 添加⼦节点（因为 Slot 其实也是这个参数列表的参数之⼀）。

|  |
| --- |
| ...  public:  typedef STreeView::FArguments FArguments;  void Construct(const FArguments& InArgs, const TSharedRef<SActImagePoolAreaPanel>& InActImageTrackAreaPanel);  ... |
| // SNew 简单写法示例  SNew(SBorder)  .BorderImage(FEditorStyle::GetBrush("ToolPanel.GroupBorder")) // SBorder -> SLATE\_ATTRIBUTE( const FSlateBrush\*, BorderImage )  .Padding(FMargin(0.6f, 0.3f)) // SBorder -> SLATE\_ATTRIBUTE( FMargin, Padding )  .Clipping(EWidgetClipping::ClipToBounds) // FArguments 基类 TSlateBaseNamedArgs -> SLATE\_PRIVATE\_ARGUMENT\_FUNCTION(EWidgetClipping, Clipping)  [  SNew(SHorizontalBox) // 单个子节点的情况，直接使用中括号添加内容  ] |

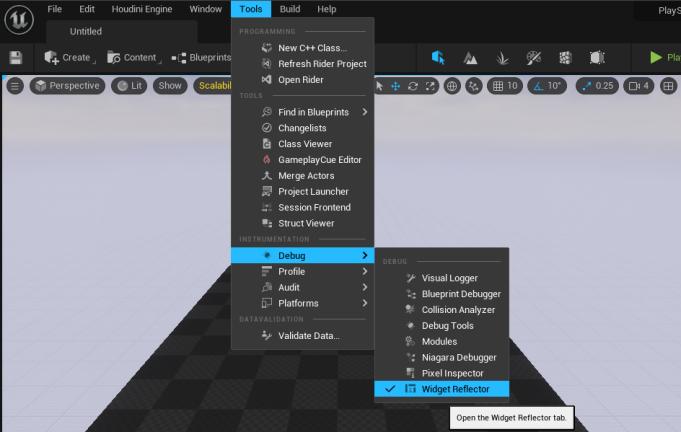
|  |
| --- |
| // SAssignNew 简单写法⽰例  TSharedPtr<SGridPanel> GridPanel = nullptr;  SAssignNew(GridPanel, SGridPanel)  .FillRow(1, 1.0f) // SGridPanel -> FArguments& FillRow( int32  RowId, const TAttribute<float>& Coefficient )  .FillColumn(0, ...) // SGridPanel -> FArguments& FillColumn( int32  ColumnId, const TAttribute<float>& Coefficient )  .FillColumn(1, ...) // SGridPanel -> FArguments& FillColumn( int32  ColumnId, const TAttribute<float>& Coefficient )  + SGridPanel::Slot(0, 0) // 多个⼦节点的情况，先通过 + XXX::Slot() 添加⼦  节点 Slot ，后中括号接⼦节点内容  [  SNew(SBorder)  ]  + SGridPanel::Slot(...)  [  ...  ] |

上⾯列举了 SNew 和 SAssignNew 的基本⽤法，以及⼀些奇怪的 Slate 语法规则，在 UE 文档中看到 Slate 的作者们信誓旦旦的说“ Slate's composition framework makes it simple to re-arrange UI elements quickly for prototyping and iteration.”，感觉说的大约是跟纯 C++ 代码去撸来比较的。实际上拆开这些宏看，跟纯 C++ 也没啥⼤区别，就是定义了些代码格式化⼯具⽆法识别的符号，代替了⼿写⼀⼤堆要靠⾃动补全才敲得出来的代码。总结⼀句话，也没好到哪⾥去。借助从参考知乎文章中看的原话：“UE4 的 UI 解决⽅案⼀直都是 Slate 表⽰的，只是上层封了层 UMG 之后感觉 UE4 的团队就很少有精⼒扩展 Slate 的东⻄了。许多 UI 层的组件需求和效果需求反应都很迟钝。”。⽽且关键的关键！这个⽅式做手机游戏那些复杂 UI 是要坑死人的，每改⼀个布局⼀个属性都需要重新编译⼀次代码，就算有 HotReload ，迭代起来也比那些脚本语⾔⽀持的 UI 框架不知道要慢到哪去了。或许这也是为啥 Slate 没有后续扩展的原因之一吧（端游的话 UI 不重度都可以另说）。

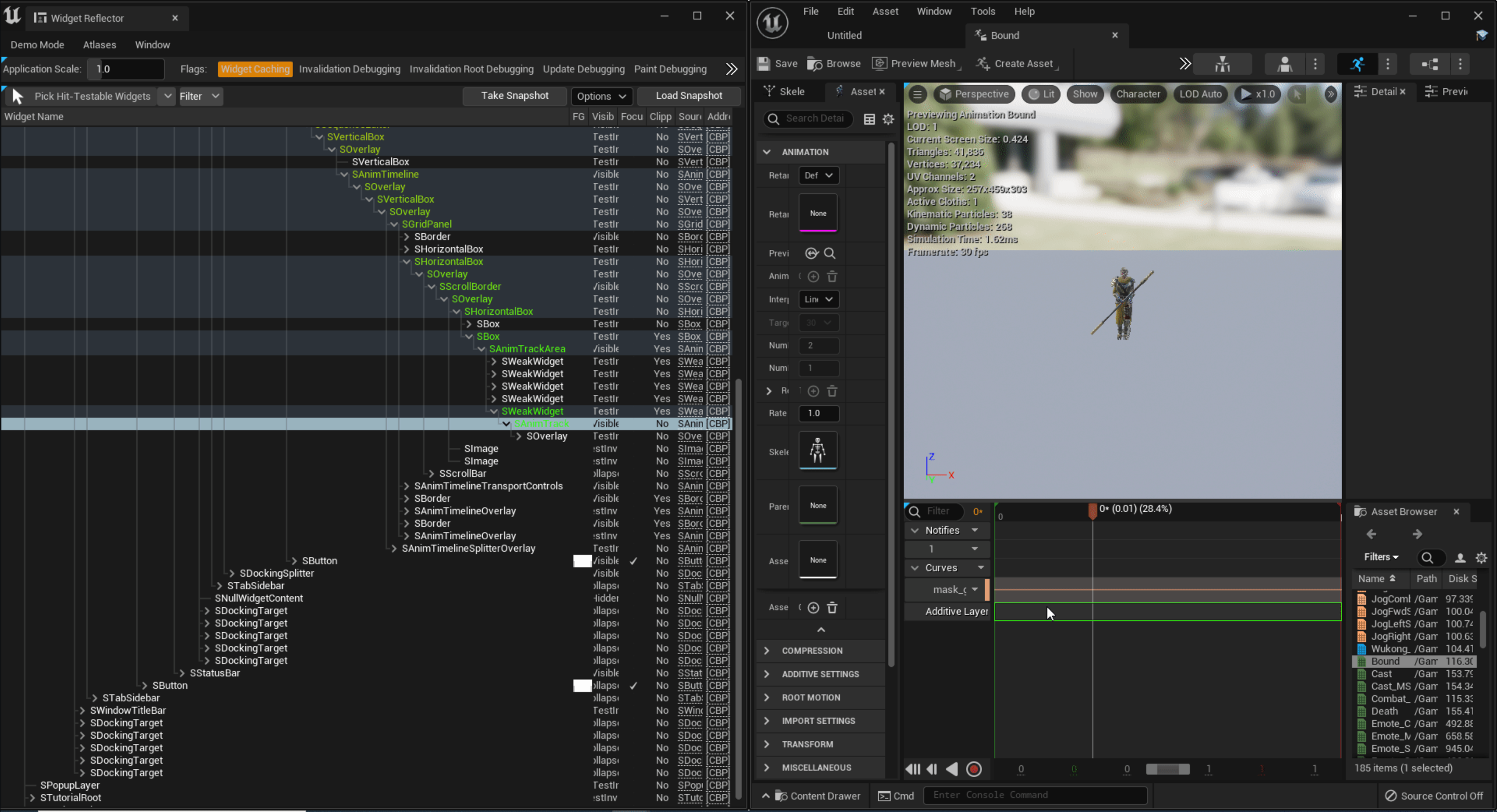
17. Construct 函数通过构造宏来调⽤，除了第⼀个参数必须满⾜规则以外，其他参数倒是可以随便传， 终归也没太搞懂⽤ FArguments 传参的好处在哪，或许是写法上的统⼀和解决构造时需要修改基类参数的问题？不论如何，写在 Construct 的参数列表后⾯，和预先定义在 Widget SLATE\_BEGIN\_ARGS ... SLATE\_END\_ARGS 之间，效果是差不多的。这两种⽅式都没有把你定义的参数替你保存在构造的 Widget 中，就还是需要你在 Widget 中再定义⼀个对应的字段去保存这个参数（除了那些只在构造时使⽤的参数/dodge)。这⾥提⼀下如何在 SLATE\_BEGIN\_ARGS ... SLATE\_END\_ARGS 之间定义参数，因为非强制的情况下个⼈不⽤这种⽅式。这些宏也都在 DeclarativeSyntaxSupport.h 文件中定义。

|  |
| --- |
| SLATE\_BEGIN\_ARGS(SBorder) { }  ...  SLATE\_ARGUMENT(EHorizontalAlignment, HAlign) // 声明字段参数，一般就是 Enum struct class 基本数据类型 这些  ...  SLATE\_ATTRIBUTE(FMargin, Padding) // 比较特殊的字段参数，会用 TAttribute 包装给的类型  ...  SLATE\_EVENT(FPointerEventHandler, OnMouseButtonDown) // 声明 Delegate 参数  ...  SLATE\_END\_ARGS() |

18. 有了构造和参数，剩下的只有怎么绘制的问题了。绘制的代码基本也是抄⼀抄，要不然没有设计的情况下也不知道这个 Widget 什么样比较合适使用者（或者从 Unity 借鉴 or ⾃⼰设计/dodge）。说到学（抄）代码，不得不介绍⼀个利器，这是个⼈认为 UE 的 Slate 做得最好的地⽅，就是下图所⽰的⼯具 Widget Reflector。



⽤法也很简单，⼀种是使⽤ Pick 功能正向查找想要学习的 Widget 。另⼀种是反向查找，点击树状结构中的节点，通过查看对应框中（绿⾊/⽩⾊）区域来确定是不是要的那个区域。这⾥只演⽰正向查找。



先确定当前区域是想要学习的区域，并且编辑器也正确的链接到 IDE 后，就可以通过 Widget Reflector 中的 Source 列快速定位 Widget 的代码了！

19. 恭喜你！已经⼤致掌握了⾃定义 UE 资源和 Slate 界⾯，接下来就是 UMG 的学习或者深入到 SlateCore 中看代码。说起 UMG ， puerts 的作者提出⽤ react 绑定 UMG 对象的想法还挺不错的，⼤致上就是让 UMG UI 布局对应套⽤成熟的 react 布局写法，核⼼靠映射的 UMG UI 组件来完成，感觉⾄少比写坑人的 C++ Slate 要好（不过这也是游戏内 UI 的后话了）。

参考文献

[[1] Slate UI Framework](https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Slate/)

[[2] UE4中UI解决⽅案Slate](https://zhuanlan.zhihu.com/p/24726208)

[[3] UE4下玩转react](https://zhuanlan.zhihu.com/p/240617365)