UE\_Ai行为树的源码分析

# run行为树的两种方法

## AIController中运行行为树

## 任务节点运行行为树

### 任务节点运行的时候，会做什么事情呢？

* + - * + 1， runBehaviortree这个任务节点中的executeTask中有UbehaviorTreeComponent::pushInstance()函数,这个关键函数的作用是加载推入，运行一个新的行为树
        + 2.判断是否可以推入一个新的行为树
        + 3. 如果可以，那就加载树，加载树有以下几个动作：初始化节点，初始化内存等
        + 4.将行为树推进栈中InstanceStack
        + 5.运行新的行为树

# 加载前的检查（蓝色泳道中的检查）

## 背景，有时候在链接runbehaviorTree任务节点的时候，明明行为树资源正确，甚至打断点命中RunBehavior这个任务节点了，但是仍没有运行，为什么？

### 那就有可能是在PushInstance之前的检查没有通过，导致RunBehavior失败了

### 那在加载行为树之前BehaviorTreeComponent都做了什么检查呢？

* + - * + 首先，就是检查子树和父树是不是使用了同一个黑板，如果不是，则不能运行
        + 其次，加载子树需要一个全局的UBehaviorTreeManager,如果获取不到，加载不了子树，返回失败
        + 最后，检查发起新子行为树的任务节点是否Push一个新的行为树，如果不愿意则返回失败，一般默认情况下是可以的，表示可以的函数是在，UBTCompositeNode::CanPushSubtree(),默认返回True

但是唯一一种情况是SimpleParallel(平行节点)会对这个函数重写，如果发起行为树的RunBehavior节点是MainTask，也就是第一个任务，那么是不允许执行，函数写的也很精妙bool UBTComposite\_SimpleParallel::CanPushSubTree(UBehaviorTreeComponent& OwnerComp,uint8\* NodeMemory,int32 ChildIdx)const { return (ChildIdx!=EBTParallelChild::MainTask) }

所以如果在SimpleParallel的第一个槽接一个RunBehavior节点，会直接导致执行节点失败。

# 树加载

## 其那面检查通过之后就正式执行树资源的加载了

## 首先是其有【缓存机制】，这可以避免不必要的重复加载，检查缓存中有没有资源。所有被加载完成的树模板都会被缓存到数组LoadedTemplates中

### 树模板是怎么存储呢

* + - * + 树模板是很简单的结构体，树资源Asset，输的根节点Template，树实例的总内存大小InstanceMemorySize

## 尝试读取缓存的时候，会将已缓存的模板进行便利操作，比较它们的Asset，如果和预加载的树的Asset一致那么直接返回该缓存过的一些信息即可

### 遍历对比，然后相同的话，使用缓存中的资源，代码如下：for(int32 TemplateIndex=0;TemplateIndex<LoadTemplates.Num();TemplateIndex++）{ FBehaviorTreeTemplateInfo& TemplateInfo=LoadTemplates[TemplateIndex] ;if(TemplateInfo.Asset==&Asset){ root=TemplateInfo.Template; InstanceMemorySize=TemplateInfo.InstanceMemorySize;return true;} }

## 如果不一致，那就是之前没有缓存过的，我们需要将新的Template加入缓存列表中，然后再进行后续操作（因为传来的是一份原始资源，不能进行修改，所以我们深拷贝一份即可，这里的拷贝函数是StaticDuplicateObject函数

### FBehaviorTreeTemplateInfo TemplateInfo; TemplateInfo.Asset =&Asset; TemplateInfo.Template=Cast<UBTCompositeNode>(StaticDuplicateObject(Asset.RootNode,this))

## 拷贝一份之后就可以进行后续操作了

### 这里先介绍一个数据结构FNodeInitializationData，这是一个用于存储节点初始化信息的结构



## 接下来运行的InitializeNodeHelper会返回一个数组的FNodeinitialzationData,代表行为树中所有节点的初始化信息

### 那这个InitializeNodeHelper的作用是干嘛了呢？我们知道他返回了一个节点数组，输入的是Template根节点，那就能推断出，这个函数作用是计算出每个节点的父节点，执行顺序，所需内存大小等

## 接着就是根据Helper遍历得的所有节点进行内存分配

### 每个节点所需要的内存大小知道了，然后就是根据每个节点的内存大小在List列表中进行排序

### 接着就是对排序之后的节点进行遍历，然后把每个节点所需的内存叠加，得到对应的内存偏移量MemoryOffset.遍历之后Memoryoffset就是这棵树所有节点需要的总内存大小

## 之后树加载完成，就是Template，将这个Template放置进templateInfo中供下次读取，然后将数据返回出去

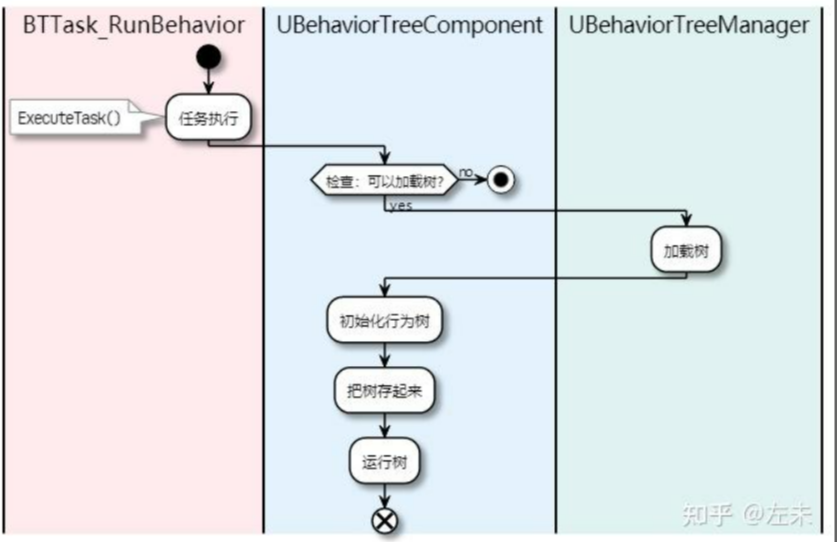
# 树加载之后初始化

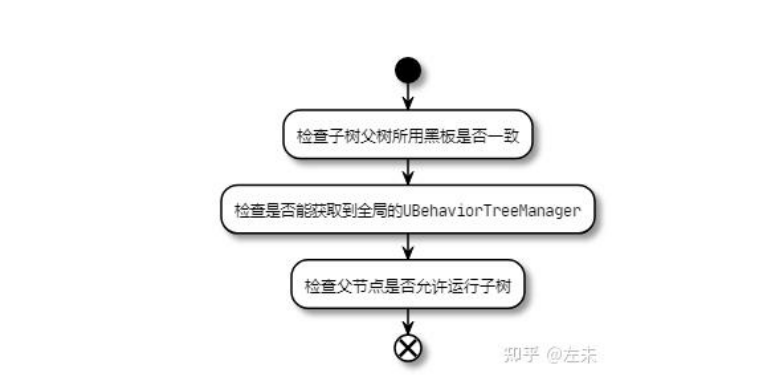
## 树资源在Manager中加载完之后，回到BehaviorTreeComponent泳道之后需要再UBehaviorTreeComponent::PushInstance中继续完成初始化

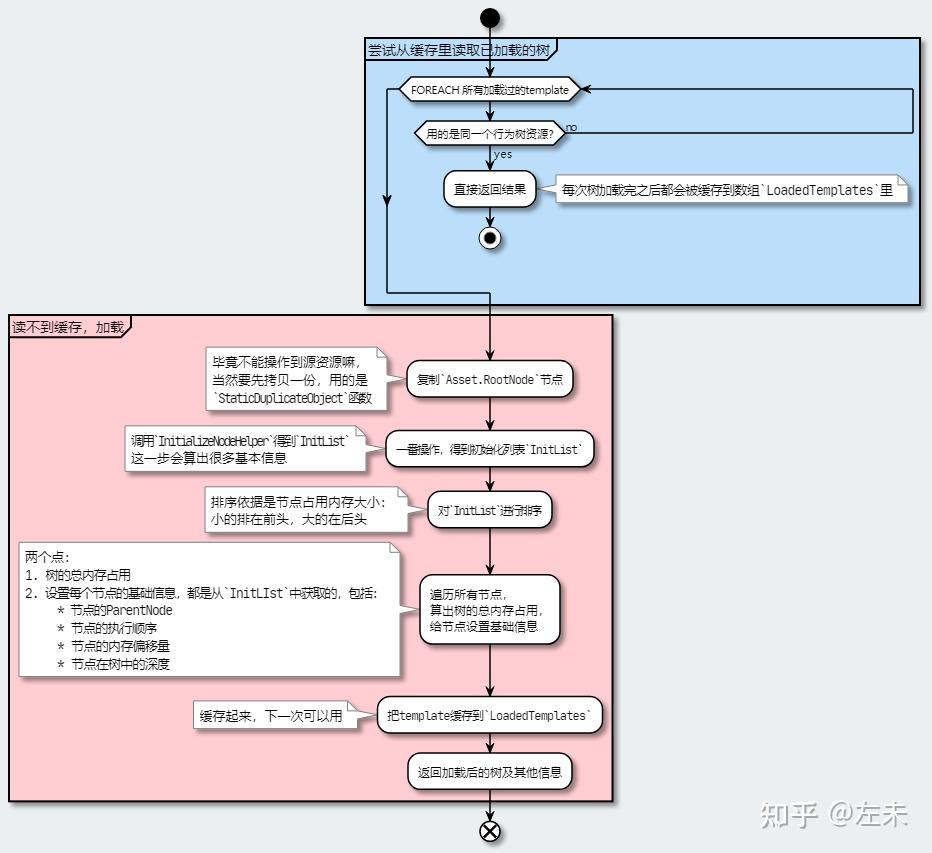
### 这里又有两个结构体

### FBehaviorTreeInstance

### FBehaviorTreeInstanceId







# 而现在我们研究的是两个东西，一个是FNodeInitializationData和FNodeInitializeNodeHelper

## InitializeNodeHelper做了什么呢？

### 当前节点处理

* + - * + 首先将一个FBehaviorTreeNodeInitializationData 对象添加进initList中，该对象存储1了当前节点NodeOb的信息，包括本身节点，父节点，执行索引，树的深度，节点实例内存大小和特殊内存大小
        + 然后进行日志记录，并初始化节点
        + ExecutionIndex++，可能代表处理完一个节点之后更新执行顺序或者计数
        + 然后判断当前节点是不是复合节点

通过Cast<UBTCompositeNode>(NodeOb)判断

* + - * + 遍历当前节点的每一个Service，如果为空节点，输出警告日志并从服务节点列表中移除该节点。
        + 不为空就将当前Service复制，然后添加到InitList数组中。然后输出日志。接着进行初始化，最后再进行ExecutionIndex++的操作

怎么Service也添加到数组中去了？我以为只有节点的，原来是所有的节点包含的结构体吗？猜想正确，下面子节点的Decorator也需要添加到数组中去

### 下层子节点处理

* + - * + 然后处理完该节点的Service节点就开始处理子节点，为啥不处理Decorate节点？我觉得是这不是这个函数的任务。这个函数只用保证跟此节点以及下层的节点相关的即可
        + 处理下层子节点，首先遍历子节点
        + 然后再循环嵌套遍历子节点的装饰器节点
        + 如果装饰器节点为空节点，输出警告日志，从装饰器节点中移除该节点
        + 如果下层节点有运行行为树的节点

将其注入到

## initializeNodeHelper主要作用就是初始化自身节点，以及下层的装饰器节点、服务节点。通过引用传递的FBehaviorTreeNodeInitializationData类型数组initList存储节点的初始化数据，通过修改ExecutionIndex进行记录执行顺序

### 然后对于复合节点会递归的处理子节点以及他们的装饰器节点，服务节点

### 最终的顺序就是父节点（Composite节点）本身 父节点（Composite节点）的所有service -> 子节点（非Composite节点）的decorator -> 子节点（非Composite节点）的service -> 子节点（非Composite节点）本身

# 这个就是相关代码

