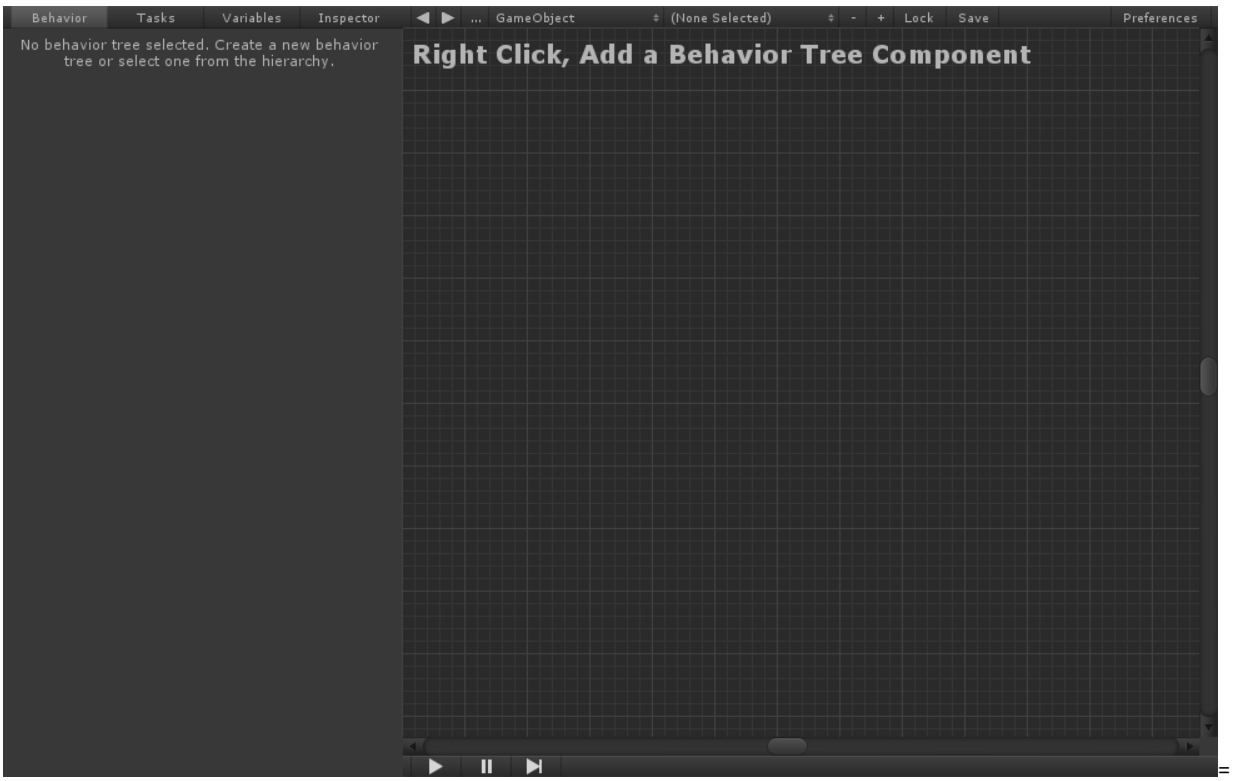
(译者：金奇 QQ 群: 475453656 下面的内容算是对官方文档的一个简单翻译和理解外加一些笔记，有一些简单的部分和章节我直接跳过了！需要查看的话去官网查看！另外没有核对过下面的翻译内容，可能存在错别字之类的地方，敬请谅解！)

Behavior Designer 概述：

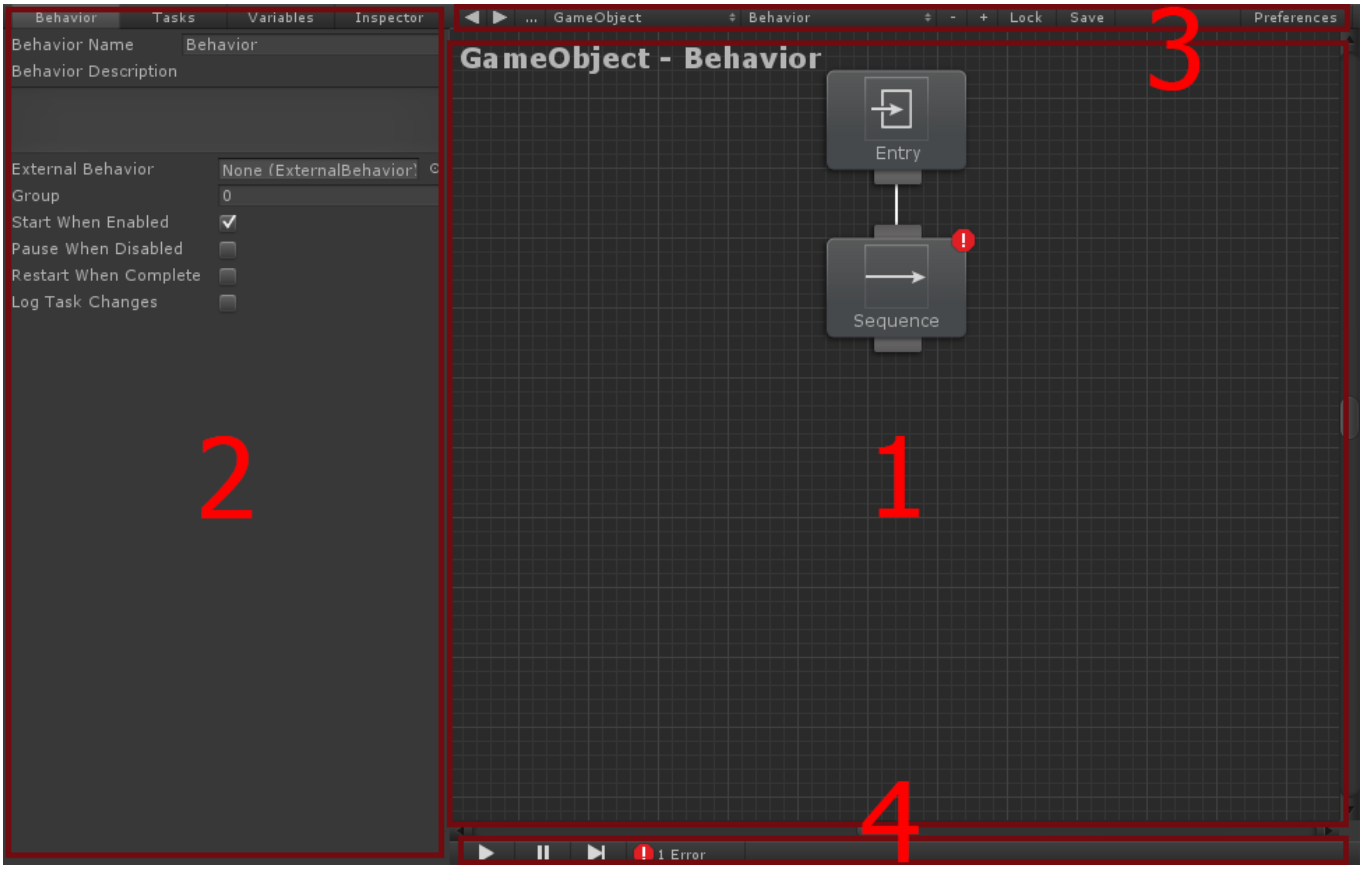
Behavior Designer 是一个行为树插件！是为了让么个设计师，程序员，美术人员方便使用的可视化编辑器！Behavior Designer 提供了强大的 API 可以让你轻松的创建 tasks（任务），配合 uScript 和 PlayMaker 这样的插件，可以不费吹灰之力就能够创建出强大的 AI 系统，而无需写一行代码！

本指南将介绍所有 Behavior Designer 的功能特性！如果你还不了解什么是行为树（behavior trees）请查看“行为树的概述”！依赖于 Behavior Designer 你完全可以不用关心什么是行为树（behavior trees）但是，如果你了解了一些 behavior trees！这将有助于你使用 Behavior Designer，包括一些常用组件，例如：tasks（任务），action（行为），composite（复合），conditional（条件），decorator（修饰符）！

当你第一次打开 Behavior Designer ，会出现下面的窗口：

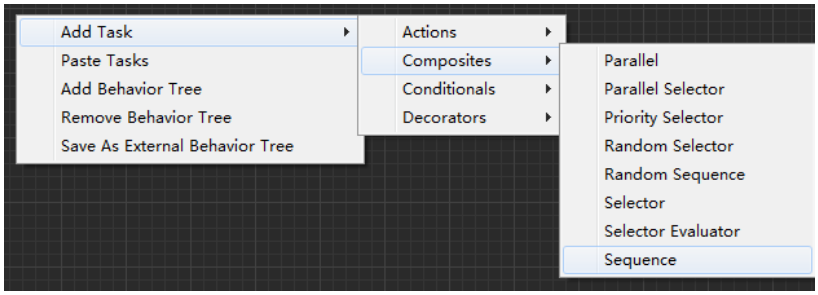


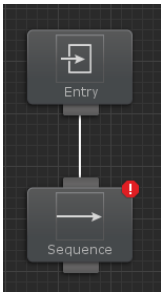
这里一共分为四大部分：下图中的第一个部分是主要操作区，用来创建你的行为树！第二部分是面板属性区，这里可以编辑一个行为树的特定属性，添加新任务，创建新的变量，或者编辑 tasks（任务）的参数。第三部分是工具栏，你可以添加/删除行为树，锁定当前行为树，查看所有行为树，等等！第四部分是调试工具栏。你可以启动/停止，逐步调试，暂停，查看错误！



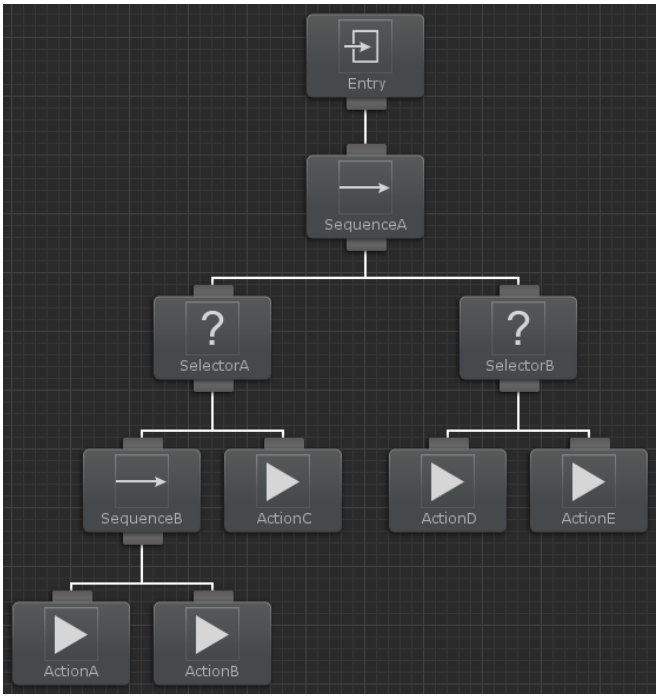
第一部分是主要的设计工作的区域！在这个区域，你可以创建新的 Task（任务）和设计这些 Task（任务）的行为树。第一步要做的就是创建一个行为树（behavior tree），通过右键选择“Add Behavior Tree”可以创建新的 behavior tree 行为树，或者通过上方的工具栏 Lock（锁定）旁边的加号添加一个新的行为树！

一旦行为树创建完毕，你就可以开始添加 tasks（任务）了。添加一个 task 可以通过右键点击空白区，或者左侧 2 区中的 Tasks 标签来创建！一旦你的 task 创建完毕，你将看到类似于下面图中的效果！





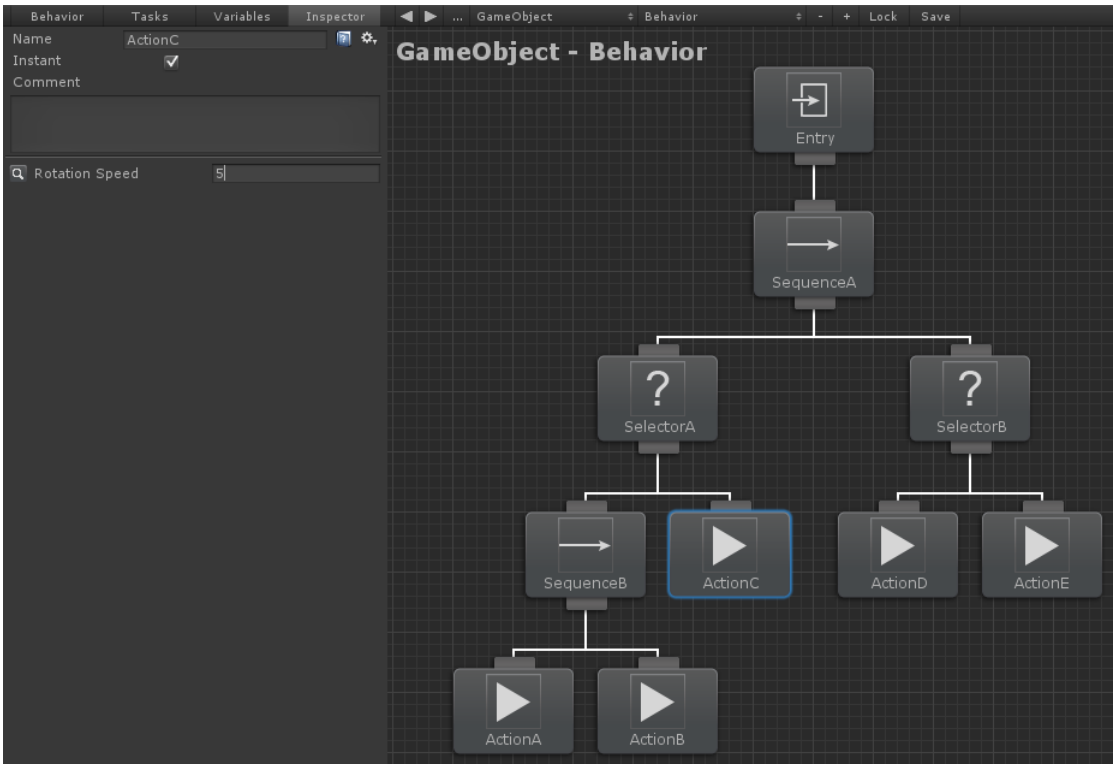
这里的 Entry task 是默认添加的，作为根节点进行后续的 Task 添加！这里能看到添加的 Sequence（队列）节点有错误，这是因为这个 Sequence 必须有后续子节点，只要添加了子节点，这个错误就会消失了！现在我们已经有了第一个 Task，接下来让我们多添加几个：



你可以创建 Sequence（序列）节点，或者 selector（选择器）节点，他们都是 task（任务）。通过多个这种节点的组合，你可以创建很深的层次结构！如果在创建过程中你不小心搞错了什么，那么选择有错误的节点，delete 删除掉即可！

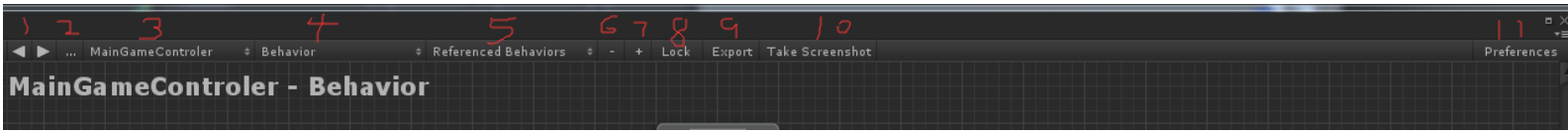
Behavior Designer 的执行顺序是从左到右的顺序执行，并且是深度优先。上图中的执行顺序如下：

SequenceA,SelectorA、SequenceB ActionA、ActionB ActionC,SelectorB,ActionD ActionE



现在我们已经有了一个基础的行为树，让我们改改参数看看！选择 ActionC 节点然后查看左侧的属性面板，选到 Inspector 视图！你可以再这里重命名这个 Task（任务）名称，设置参数，或者输入一个注释在（Comment）

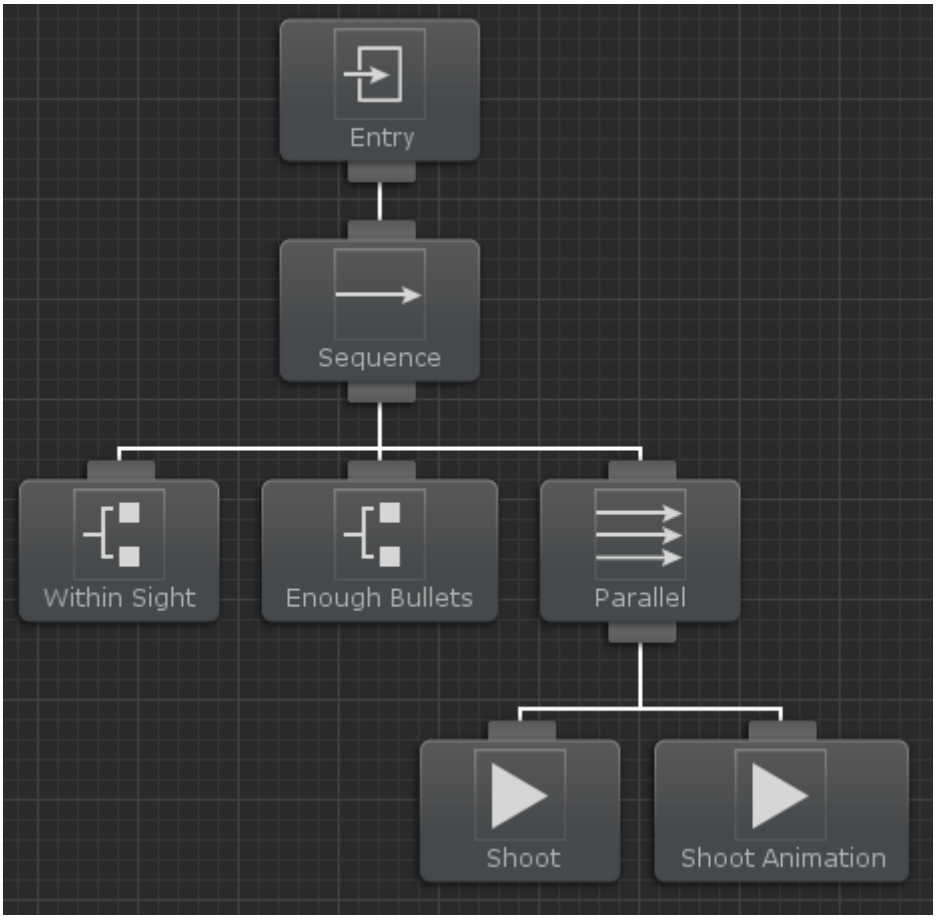
在左侧的界面中有出了 Inspector 还有其他三个标签（Behavior，Tasks，Variables）Variables 面板允许你创建共享变量，并改变变量值！关于这个面板的具体信息，在后面章节介绍！Tasks 面板列出了所有可用的 tasks（任务）这里的 tasks 和你在空白区右键弹出的菜单中的 tasks 是一样的！这个列表可以通过上方的搜索框输入，快速定位到你想要创建的 tasks 任务，包括 action（行为），composite（复合），conditional（条件），decorator（修饰符）！类型！最后一个面板，behavior 面板，显示的是当前行为树 Behavior Tree 组件的属性，这个属性也会出现在所捆绑的 Gameobject 的属性面板上！各个面板具体介绍请看后续章节介绍！



最后的是工具栏，工具栏提供了例如添加/删除等基础的行为树操作的工具！上图中 1 号位置的箭头是用来预览不同行为树，如果在同一个 Gameobject 上添加了多个行为树的话！2 号位置用来显示所有行为树，并可以选择某个行为树，3 号位置是当前场景中拥有行为树的 Gameobject，可以展开下拉菜单并查看和选择！4 号位置也是用来选择当前 Gameobject 上的不同行为树，如果有多个的话，5 号是删除当前行为树，6 号增加一个行为树，7 号锁定当前行为树试图，不过因为在场景中点击其他资源导致行为树设计面板中的内容切换！8 号是保存当前行为树，9 号是导出当前行为树作为一个 Scriptable 资源以便其他行为树调用，10 号截图，11 号偏好设置，包括了一些基础设置！

什么是行为树 Behavior Tree ？

行为树在人工智能游戏中很受欢迎。像《光晕 2》就是一个使用行为树并火起来的游戏！行为树的有点就是很容易理解并且是可视化的编辑！



下面来了解一下行为树：有四种不同类型的 task(任务): 包括 action（行为），composite（复合），conditional（条件），decorator（修饰符）！
action（行为）可能很容易理解，因为他们在某种程度上改变游戏的状态和结果。

conditional（条件）用来判断某些游戏属性是否合适！例如：在上图中的行为树中，有两个 conditional（条件）节点，两个 action（行为）节点
前两个 conditional（条件）用来检查是否需有敌人，并确保是否有足够的子弹。如果这些条件都是真的，纳闷这两个 task(任务)将被执行，并执行后续任务，如果有 conditional（条件）不满足，则不会执行后续操作，直接返回上层的 Sequence，并结束本次行为树的执行！之后的是一个并行队列（parallel），下面的两个 action（行为）第一个负责计算设计伤害，第二个负责播放射击动画，他们是同事发生的！这里你完全可以把后面的两个 action（行为）作为单独的一个行为树！以此类推，编辑出负责的，嵌套的行为树！

composite（复合）：从上图中可以看出，Sequence 和 parallel 属于 composite（复合）节点。一个是顺序执行，一个是并列执行！

decorator（修饰符）：这个类型的节点只能有一个子节点。它的功能是修改子任务的行为。在上面的例子中，我们没有使用 decorator（修饰符），如果你需要类似于打断操作的话会用得到这个 decorator（修饰符）类型！举个例子：一个收集资源的操作，它可能有一个中断节点，这个节点判断是否被攻击，如果被攻击则中断收集资源操作！decorator（修饰符）的另一个应用场合是重复执行子任务 X 次，或者执行子任务直到完成！

行为树还有一个重要话题，那就是返回状态！有时候一个 task（任务）需要多帧才能完成。例如，大多数动画不会在一帧开始并结束。此外有 conditional（条件）的任务需要一种方法来告诉他们的父任务条件是否正确，以便让父节点确定子节点的执行顺序。这两个问题都可以使用 status（状态）来解决。一个任务有三种不同状态：运行，成功或者失败。在第一个例子中，射击动画的 task 任务只有一个 status 状态，而确定敌人的条件是在 Within Sight 任务中返回的，如果返回失败，也就是不在视野中则不会执行到后面的任务！

究竟该用行为树还是有限状态机？Behavior Trees or Finite State Machines

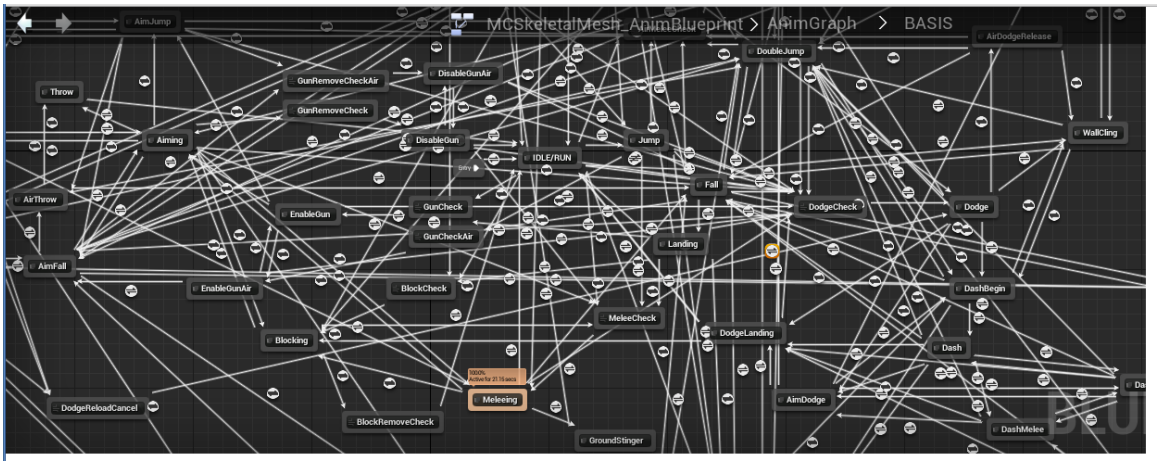
在什么情况下会需要用行为树（BehaviorTree）而不用有限状态机（FiniteStateMachines）例如 PlayMaker 这种插件？从更高级的逻辑层面来讲，行为树常常用于复杂的人工智能，而有限状态机（FSMs）则用于一般的可视化变成。当然你也可以用有限状态机（FSMs）来写 AI，或者使用行为树（BehaviorTree）来进行可视化编程，工具的使用因人而异！《人工智能开发》《AI Game Development》的作者 Alex J. Champandard 在 2007 年 12 月 28 日的一篇博客中提到：有限状态机（FSMs）时代已经结束了的 10 大原因！原文地址：http://aigamedev.com/open/article/fsm-age-is-over/ 。（译者：当然万事无绝对，有限状态机还是有他的用武之地的！因人而异，因游戏而异！）虽然行为树不至于走到这一步，但是可以肯定行为树在 AI 上比状态机有着绝对的优势！

行为树比有限状态的几个优势：行为树提供了强大的灵活性，非常强大，并且很容易更改行为树结构！

让我们先来看第一个优势：灵活性！在使用状态机 FSM 时，你要如何同时执行两个状态呢？你只能去创建两个状态机（FSM）！但是如果你使用行为树的话，你只需要添加一个并行节点（Parallel）即可，所有子节点都将并行执行！使用 Behavior Designer，这些子节点可以是 PlayMaker 的 FSM，并且这些 FSMs 将被并行触发！

另一个关于灵活性的例子就是 guard task（监控任务）。比如你有两个不同的 task（任务）一个播放声音，一个播放特效。这两个任务在行为树里是两个不同的分支，所以他们之间互相并不知道对方的状态，有可能同一时间这两个任务被同时执行！你可能不希望这种情况发生。在这种情况下，你可以添加一个 semaphore task（在 Behavior Designer 中被称为 Task Guard 监控任务）这样就可以在行为树中保证当前要么播放音效，要么播放特效！只有当第一个播放完毕，才会播放第二个！

行为树另一个有点：行为树的结构很健壮很清晰！这并不是说 FSM 结构并不够健壮不够清晰，只是他们的实现方式不同！在我看来，行为树让 AI 实现比有限制状态机更加方便！行为树能更好的去表达和实现复杂的 AI，而如果使用 FSM 去实现则会很复杂！为了达到同样的效果的 FSM 连接线最终可能开上去就像面条！

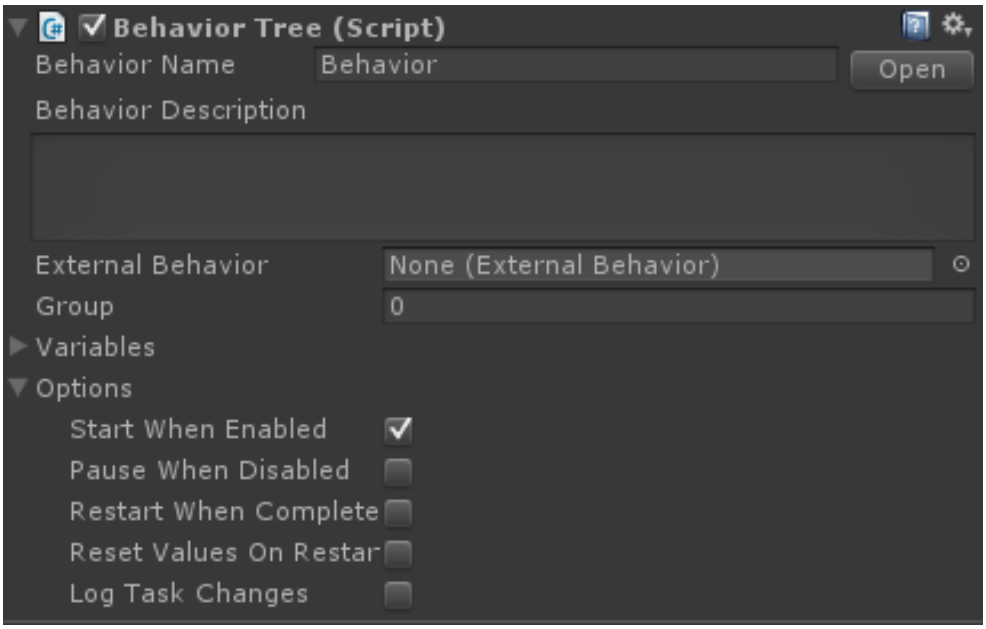


最后一个行为树优点：方便修改！行为树边的如此受欢迎原因之一就是很容易创建可视化的编辑器！在 FSM 中你如果想改变执行顺序，你必须在状态之间进行切换操作，改变各种连线！而在行为树中你不必这么麻烦！而且添加删除节点也很方便！

说了这么多，行为树和 FSM 并不一定是互斥的！他们可以相互配合使用已达到更好的效果！

行为树组件：Behavior Tree Component

如下图：



这个组件记录了你的行为树的结构以及一些 BehaviorDesigner 配置信息！下面的 API 用来启动和停止你的行为树！

```
public void EnableBehavior();
public void DisableBehavior(bool pause = false);
```

你可以通过下面的这些方法查找行为树中的相关节点 task 任务

```
TaskType FindTask< TaskType >();
List< TaskType > FindTasks< TaskType >();
Task FindTaskWithName(string taskName);
List< Task > FindTasksWithName(string taskName);
```

行为树当前的执行状态可以像下面这样获取：

```
behaviorTree.ExecutionStatus;
```

当行为树运行结束后会有一个状态被返回，返回的接口可能是 Success 成功或者是 Failure 失败，这个结构依赖于行为树中的各个子节点 Task 任务的返回值！

你可以对行为树监听以下事件：

```
OnBehaviorStart
OnBehaviorRestart
OnBehaviorEnd
```

行为树组件包含以下几个属性：

Behavior Name

行为树的名称

Behavior Description

行为树的描述信息

External Behavior

一个外部行为树的资源引用，行为树可以被导出成外部序列化文件（ScriptableObject 文件）单独存储，并被其他行为树引用，或者作为子节点任务而使用！方便了行为树的共用！

Group

行为树的分组编号，用来将行为树分组！可以用来方便的查找到特定的行为树！

Start When Enabled

如果设置为 true，那么当这个行为树组件 enabled 的时候，这个行为树就会被执行！

Pause When Disabled

如果设置为 true，那么当这个行为树组件 disabled 的时候，这个行为树就会被暂停！

Restart When Complete

如果设置为 true，那么当这个行为树组件执行结束的时候，这个行为树就会被重新执行！

Reset Values On Restart

如果设置为 true，那么当这个行为树组件 reset 的时候，这个行为树就会被重新执行！

Log Task Changes

当设置为 true 是，这个行为树下只要 task 流程发生变化就会打印一条 log 日志到控制台中！

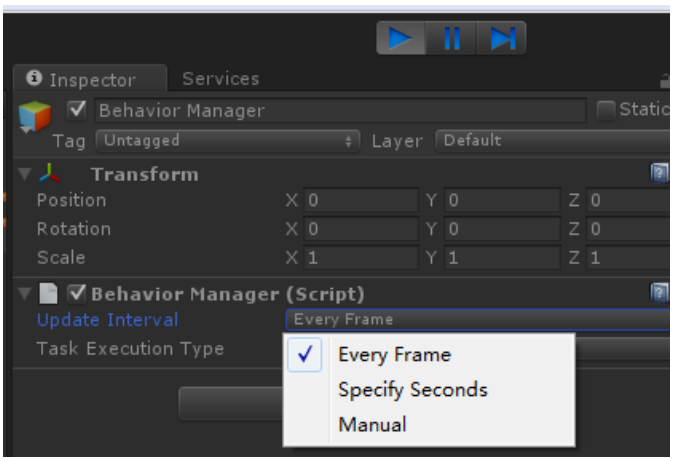
用脚本创建一个行为树

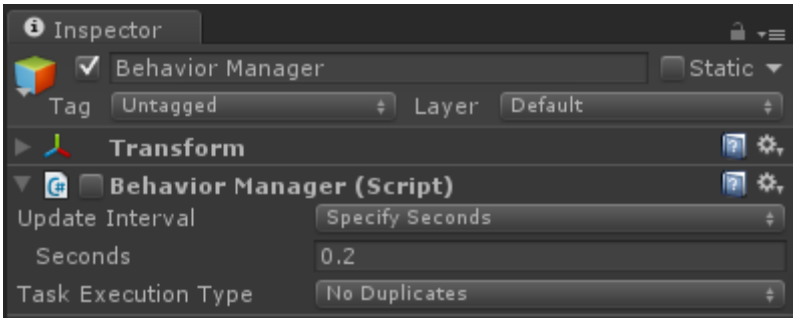
在某些情况下，你可能想要通过脚本在运行时创建一个行为树，而不是直接使用拖拽或者面板操作去创建！例如：如果你已经导出了一个外部行为树，并想通过脚本创建它的话，可以如下这么做：

```
using UnityEngine;
using BehaviorDesigner.Runtime;

0 个引用
public class main : MonoBehaviour
{
    public ExternalBehaviorTree behaviorTree;

    0 个引用
    void Start () {
        var bt = gameObject.AddComponent < Behavior>();
        bt.ExternalBehavior = behaviorTree;
        bt.StartWhenEnabled = false;
    }
}
```

在这个例子中公共变量 behaviorTree 包含你引用的外部行为树。新创建的行为树在创建时将自动加载所有子节点任务。通过设置 startWhenEnabled 为 false 来阻止行为树在创建后立刻被执行！可以通过 bt.enabledBehavior()来开启行为树！

Behavior Manager 行为管理器、





当运行一个行为树的时候，会在场景中自动创建一个名称为 BehaviorManager 的 GameObject，并添加了组件脚本 BehaviorManage.cs! 这个脚本用来管理所有场景中的行为树！

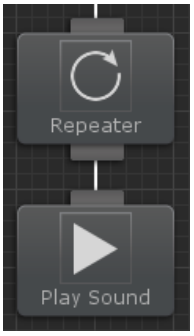
你可以控制行为树的更新类型，以及更新时间等等！"Every Frame"是每帧都更新行为树！"Specify Seconds"定义个一个更新间隔时间！"Manual"是手动调用更新，选择这个后需要通过脚本来调用行为树的更新，例如下面这样：

```
BehaviorManager.instance.Tick();
```

此外，如果你想让不同的行为树都有各自独立的更新间隔的话，可以这样：

```
BehaviorManager.instance.Tick(BehaviorTree);
```

Task Execution Type（任务执行类型）允许你指定行为树行为树的执行次数，默认是“No Duplicates”（不复制，不重复）像下图中的这种循环操作可以简单的通过这类设置执行次数来实现！



Repeater Task（重复任务节点）设置成 5 次。如果 Task Execution Type（任务执行类型）被设置为“No Duplicates”（不复制，不重复），那么 Play Sound task(播放音乐任务节点)则会被每帧执行一次。如果 Task Execution Type（任务执行类型）被设置为 5，那么那么 Play Sound task(播放音乐任务节点)会在每帧被执行 5 次！

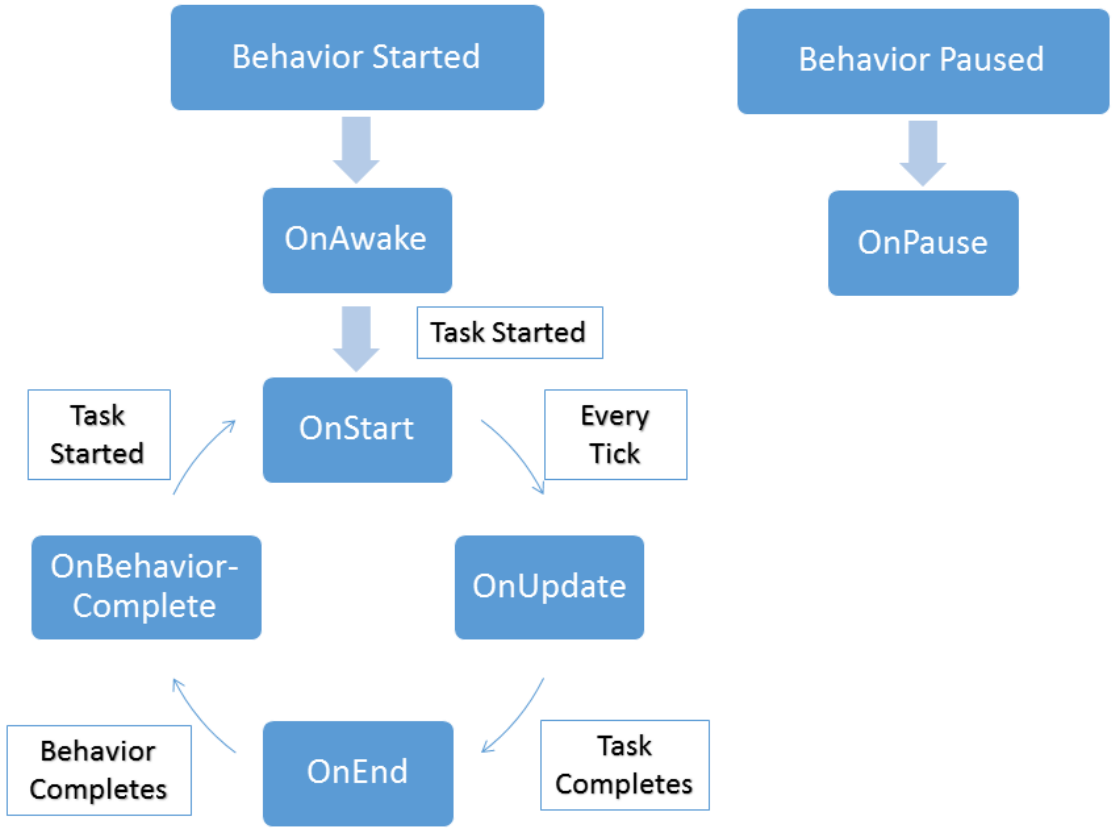
Tasks（任务）

在整个任务树的最高层的节点我们称之为 Task（任务）。这些 task 任务拥有类似于 MonoBehavior 那样的接口用于实现和扩展，如下：

```
// OnAwake is called once when the behavior tree is enabled. Think of it as a constructor
public virtual void OnAwake();
// OnStart is called immediately before execution. It is used to setup any variables that need to be reset from the previous run
public virtual void OnStart();
// OnUpdate runs the actual task
public virtual TaskStatus OnUpdate();
// OnEnd is called after execution on a success or failure.
public virtual void OnEnd();
// OnPause is called when the behavior is paused and resumed
public virtual void OnPause(bool paused);
// The priority select will need to know this tasks priority of running
public virtual float GetPriority();
// OnBehaviorComplete is called after the behavior tree finishes executing
public virtual void OnBehaviorComplete();
// OnReset is called by the inspector to reset the public properties
public virtual void OnReset();
// Allow OnDrawGizmos to be called from the tasks
public virtual void OnDrawGizmos();
// Keep a reference to the behavior that owns this task
public Behavior Owner;
```

task 任务有三个基础的公共属性：name, comment, instant（名称，简介，立刻）。这里的 instant 立刻，并不好容易理解！行为树中，当一个 task 任务返回成功或者失败后，行为树会在同一帧中立刻移动到下一个 task 任务。如果你没有选择 instant 选项，那么在当前 task 任务执行完毕后，都会停留在当前节点中，直到收到了下一个 tick，才会移动到下一个 task 任务！

下面是执行的顺序的流程图：



父任务 Parent Tasks

behavior tree 行为树中的父任务 task 包括：composite（复合），decorator（修饰符）！虽然 Monobehaviour 没有类似的 API，但是并不难去理解这些功能：

```
// The maximum number of children a parent task can have. Will usually be 1 or int.MaxValue
public virtual int MaxChildren();
// Boolean value to determine if the current task is a parallel task
public virtual bool CanRunParallelChildren();
// The index of the currently active child
public virtual int CurrentChildIndex();
// Boolean value to determine if the current task can execute
public virtual bool CanExecute();
// Apply a decorator to the executed status
public virtual TaskStatus Decorate(TaskStatus status);
// Notifies the parent task that the child has been executed and has a status of childStatus
public virtual void OnChildExecuted(TaskStatus childStatus);
// Notifies the parent task that the child at index childIndex has been executed and has a status of childStatus
public virtual void OnChildExecuted(int childIndex, TaskStatus childStatus);
// Notifies the task that the child has started to run
public virtual void OnChildStarted();
// Notifies the parallel task that the child at index childIndex has started to run
public virtual void OnChildStarted(int childIndex);
// Some parent tasks need to be able to override the status, such as parallel tasks
public virtual TaskStatus OverrideStatus(TaskStatus status);
// The interrupt node will override the status if it has been interrupted.
public virtual TaskStatus OverrideStatus();
// Notifies the composite task that an conditional abort has been triggered and the child index should reset
public virtual void OnConditionalAbort(int childIndex);
```

编写自定义的条件任务节点：Writing a New Conditional Task

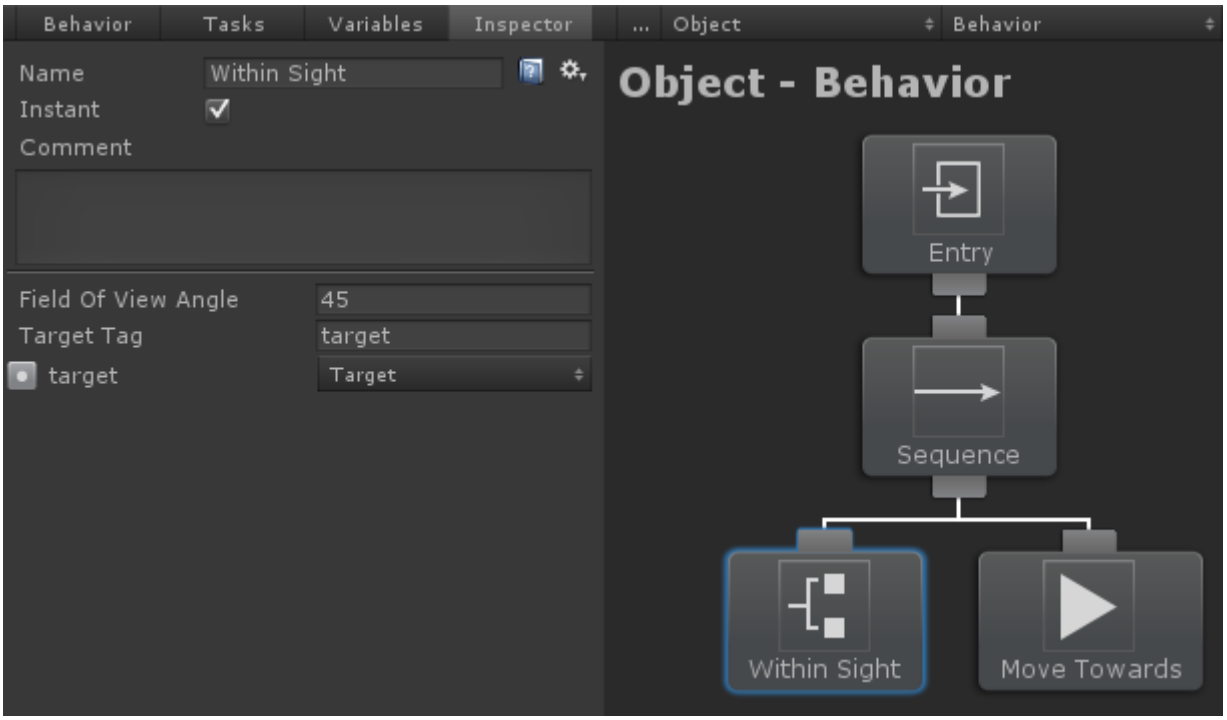
这个主题包含两个部分。第一部分介绍如何编写新的条件任务节点 conditional task，第二个部分介绍如何编写行为任务 action task conditional task（条件任务节点）用来判断某些变量和条件，而 action task（行为任务节点）则负责执行某些具体的逻辑操作！下面举例来写一个判断是否在视野距离中的条件任务节点（WithinSight）以及一个朝目标移动的（action task）(译者：具体步骤略，直接上最终完整代码)

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

public class WithinSight : Conditional
{
	// How wide of an angle the object can see
	public float fieldOfViewAngle;
	// The tag of the targets
	public string targetTag;
	// Set the target variable when a target has been found so the subsequent tasks know which object is the target
	public SharedTransform target;

	// A cache of all of the possible targets
	private Transform[] possibleTargets;

	public override void OnAwake()
	{
		// Cache all of the transforms that have a tag of targetTag
		var targets = GameObject.FindGameObjectsWithTag(targetTag);
		possibleTargets = new Transform[targets.Length];
		for (int i = 0; i < targets.Length; ++i) {
			possibleTargets[i] = targets[i].transform;
		}
	}

	public override TaskStatus OnUpdate()
	{
		// Return success if a target is within sight
		for (int i = 0; i < possibleTargets.Length; ++i) {
			if (withinSight(possibleTargets[i], fieldOfViewAngle)) {
				// Set the target so other tasks will know which transform is within sight
				target.Value = possibleTargets[i];
				return TaskStatus.Success;
			}
		}
		return TaskStatus.Failure;
	}

	// Returns true if targetTransform is within sight of current transform
	public bool withinSight(Transform targetTransform, float fieldOfViewAngle)
	{
		Vector3 direction = targetTransform.position - transform.position;
		// An object is within sight if the angle is less than field of view
		return Vector3.Angle(direction, transform.forward) < fieldOfViewAngle;
	}
}
```

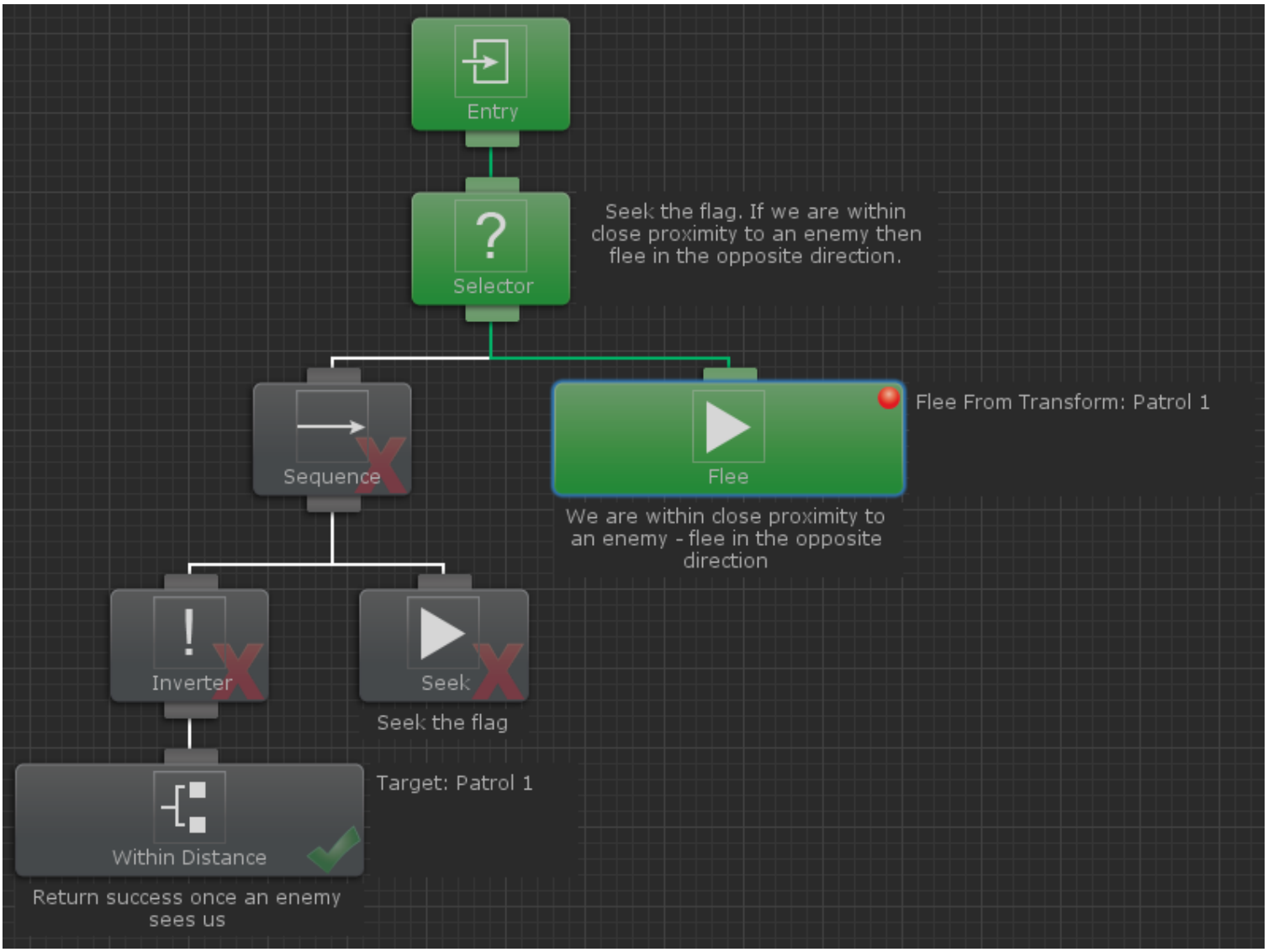
编写自定义行为任务节点 Writing a New Action Task

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

public class MoveTowards : Action
{
	// The speed of the object
	public float speed = 0;
	// The transform that the object is moving towards
	public SharedTransform target;

	public override TaskStatus OnUpdate()
	{
		// Return a task status of success once we've reached the target
		if (Vector3.SqrMagnitude(transform.position - target.Value.position) < 0.1f) {
			return TaskStatus.Success;
		}
		// We haven't reached the target yet so keep moving towards it
		transform.position = Vector3.MoveTowards(transform.position, target.Value.position, speed * Time.deltaTime);
		return TaskStatus.Running;
	}
}
```

最终在编辑器中连接起来后是这个样子！



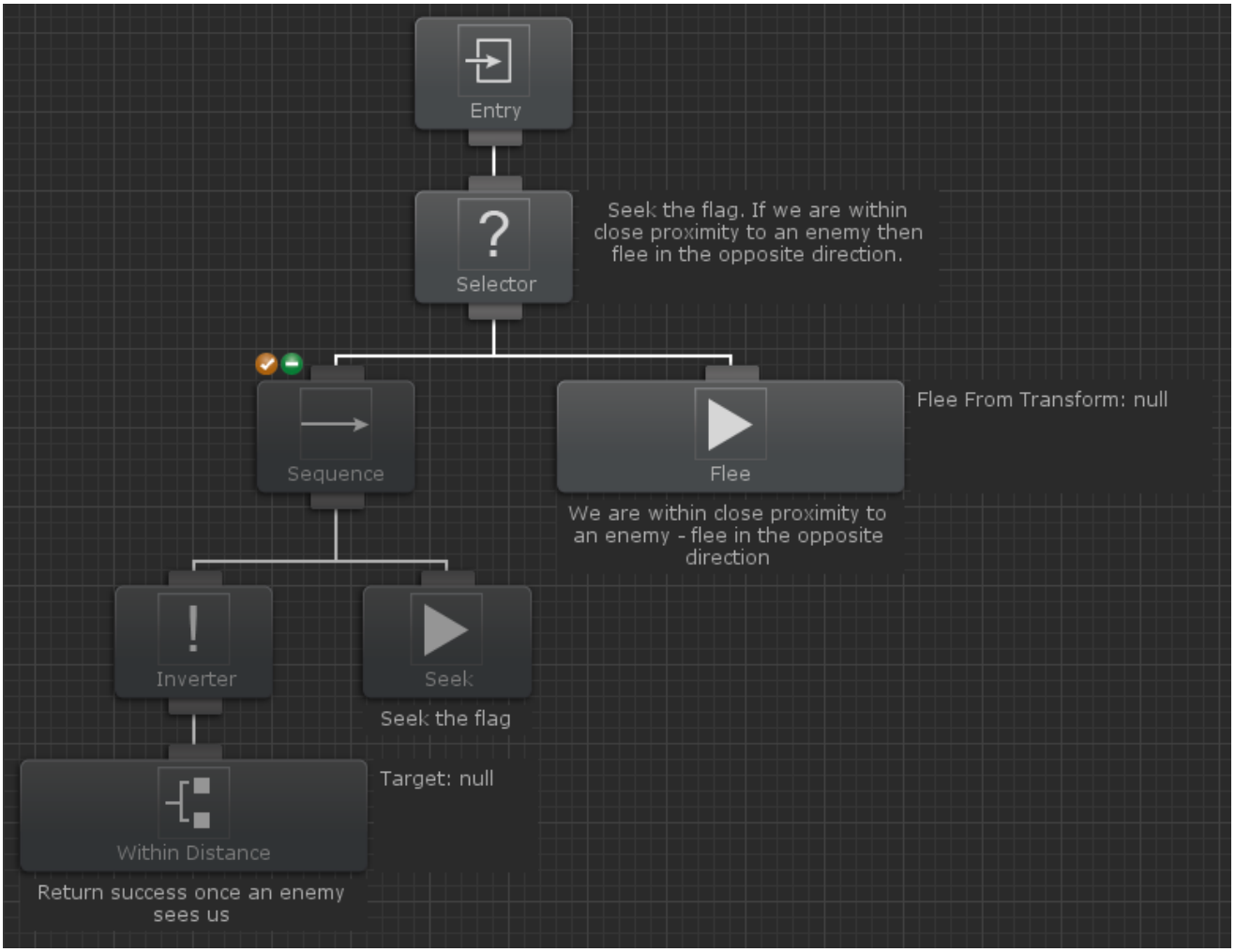
调试



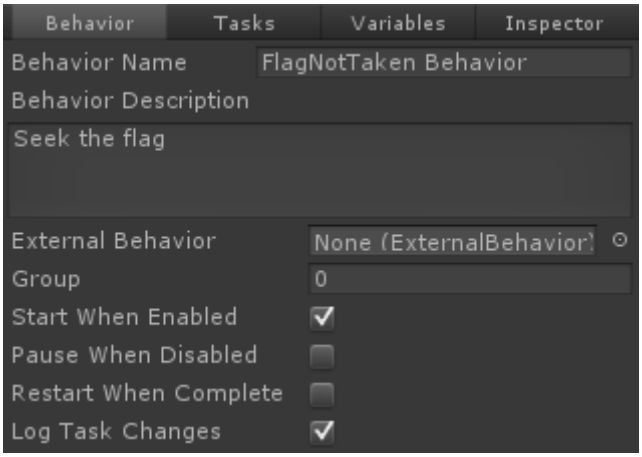
当行为树在执行的过程中，你会看到类似上图的效果，绿色的是真在执行的部分，灰色的是没有执行或者执行过的部分！部分节点的右下角  ，或者  表示这个节点的返回值是成功，还是失败！任务运行时仍然可以通过属性面板来改变数值并查看数值改变后的游戏表现！

通过鼠标右键点击某个任务节点，可以给这个节点添加一个断掉，这样在运行到这个节点的时候会中断，你可以查看节点的状态和属性等等！如上图；

当你选中某个任务节点后，可以通过左侧的 Inspector 面板来查看具体的变量，并通过变量掐面的按钮，在设计区域查看变量具体的值！如上图！



有时候你只希望执行行为树的一部分而不是全部，那么你可以禁用某些节点极其子节点，只需选中某个节点然后选择左上角的 X 号即可！



另外通过打开 Behavior 的 LogTaskchanges 也可以打印行为树的执行顺序，类似于下面的输出

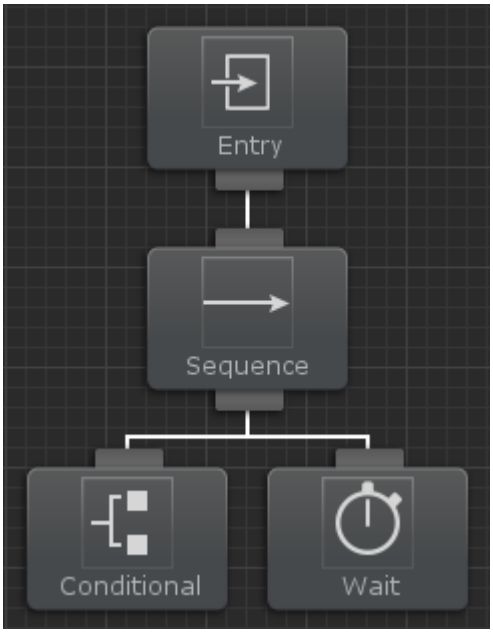
```
GameObject - Behavior: Push task Sequence (index 0) at stack index 0
GameObject - Behavior: Push task Wait (index 1) at stack index 0
GameObject - Behavior: Pop task Wait (index 1) at stack index 0 with status Success
GameObject - Behavior: Push task Wait (index 2) at stack index 0
GameObject - Behavior: Pop task Wait (index 2) at stack index 0 with status Success
GameObject - Behavior: Pop task Sequence (index 0) at stack index 0 with status Success
Disabling GameObject - Behavior
```

这些消息可以分成以下部分:

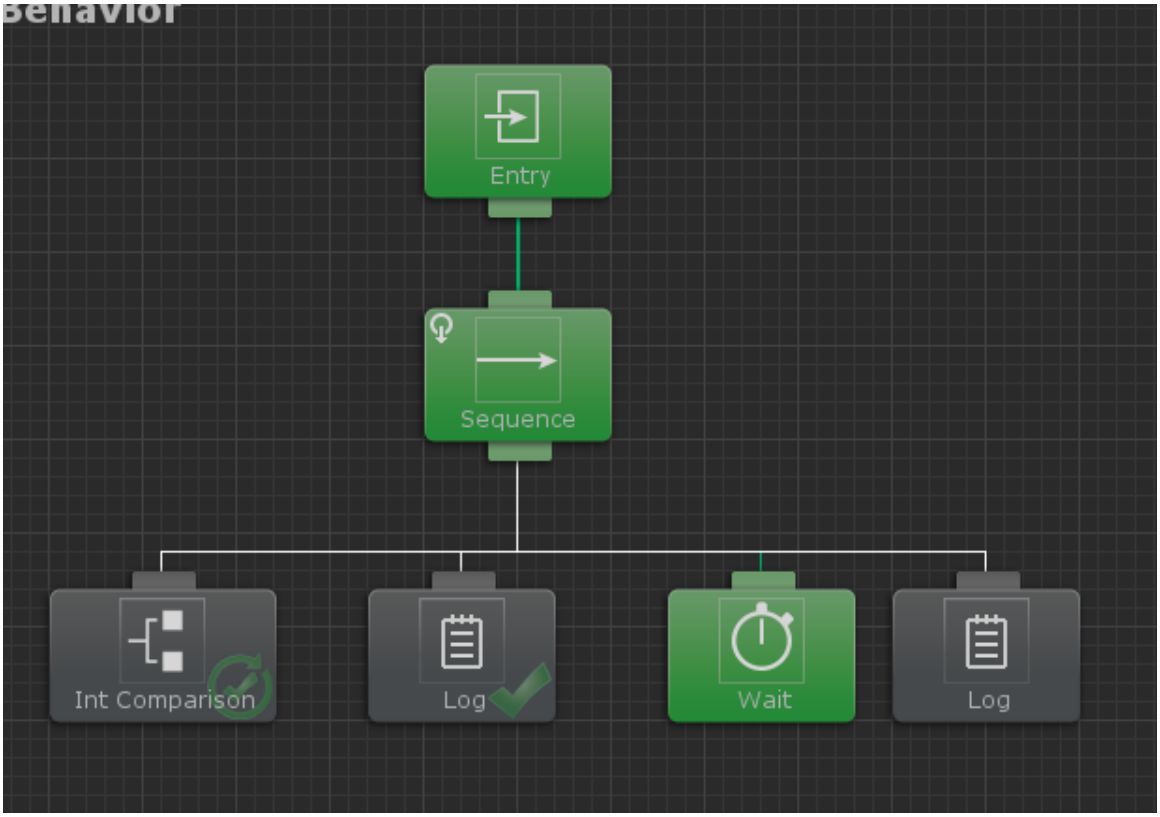
{game object name } – {behavior name}: {task change} {task type} (index {task index}) at stack index {stack index} {optional status}

条件节点的终止 Conditional Aborts

Conditional aborts（条件 终止）允许你的行为树动态的改变，而无需使用很多的类似于 打断/执行打断（Interrupt/Perform Interrupt） 等等类似的任务。这个特性类似于虚幻 4 中的观察者中止。大多数的其他行为树工具在处理类似问题的时候都需要重新遍历一次行为树。而这里的 Conditional aborts （条件终止）可以避免这种重新遍历的情况！以下图为例来说明下它的用法:

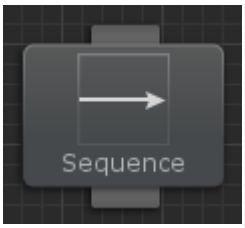


当这个行为树运行的时候，先执行 Conditional 判断，如果返回正确，则到 Wait 节点等待，这里 Wait 节点等待 10 秒！假设在等待过程中 conditional 节点的判断条件发生变化，返回 failure。如果 conditional 的 aborts（打断）被开启了的话，conditional 节点会触发一个打断操作并停止掉 Wait 节点的任务！conditional 节点任务会根据之前的逻辑重新评估并返回是否成功！conditional 节点的 aborts 可以被任何 composite 复合节点(上图中的 Sequence 节点)访问到.如下图这样：如果第一个 Int 的判断条件不满足，则会被重新执行一遍 Sequence



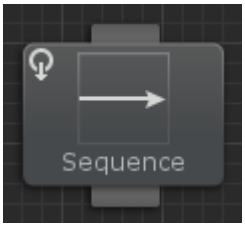
一共有四种中断类型的 abort types: None, Self, Lower Priority, and Both.

None



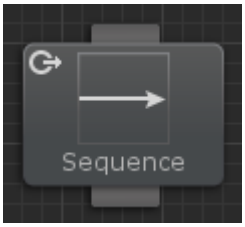
这种是默认的中断类型！

Self



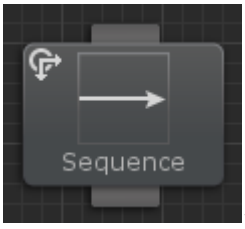
这是一种自包含中断类型。也就是会检测此节点下所有条件判断节点，即便是被执行过的节点，如果判断条件不满足则打断当前执行顺序从新回到判断节点判断，并返回判断结果！

Lower Priority



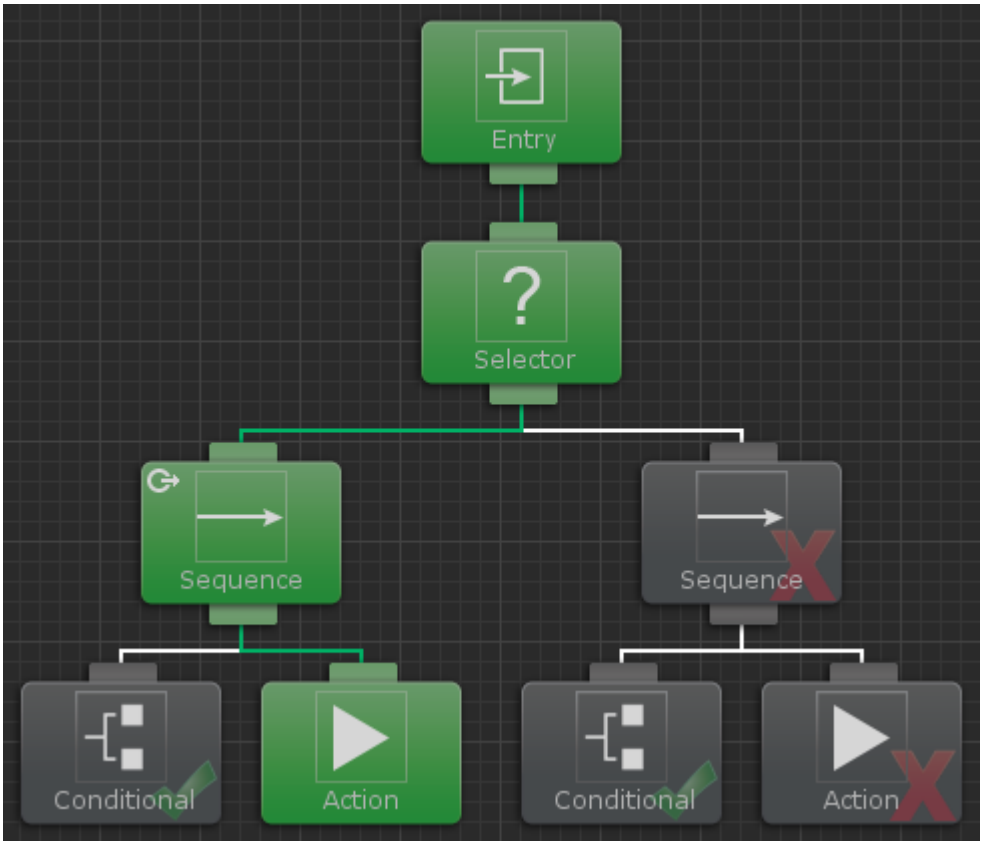
当运行到后续节点时，本节点的判断生效了的话则打断当前执行顺序，返回本节点执行！

Both

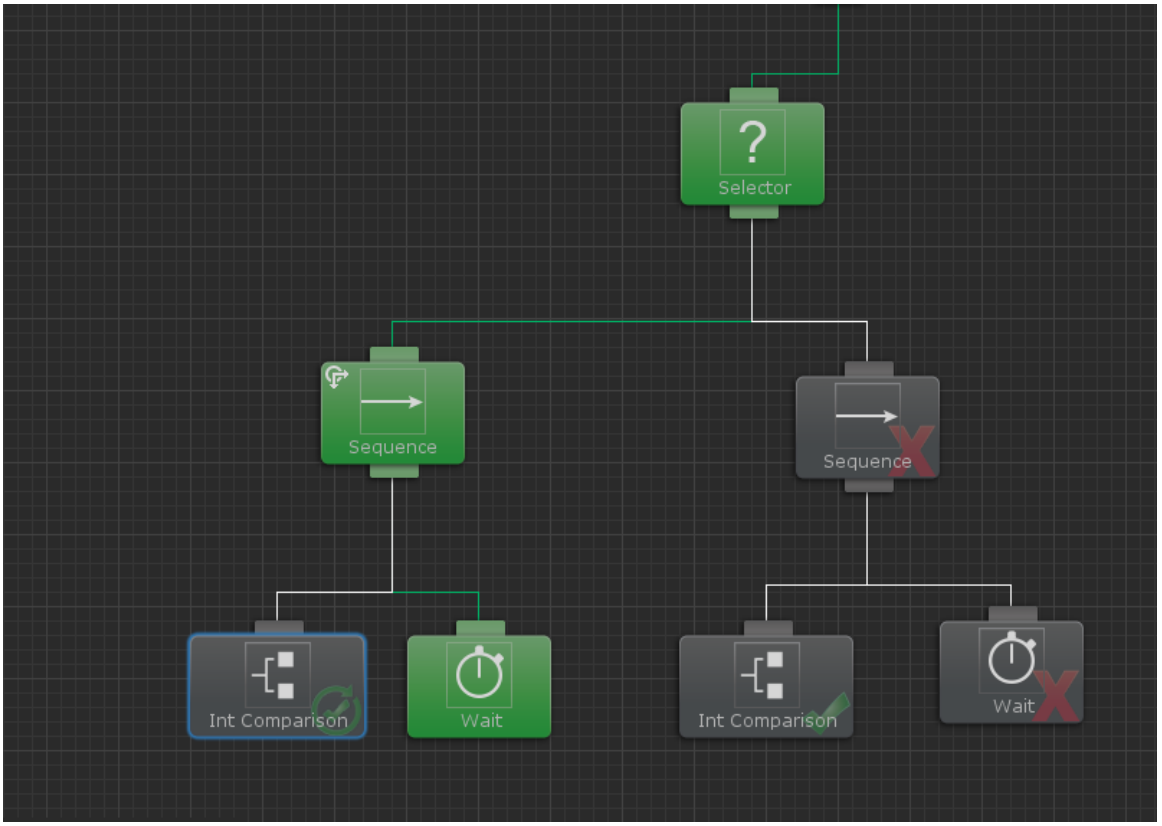


包含了 上面的两个类型！

下面的示例将使用低优先级的中断类型 **Lower Priority**:

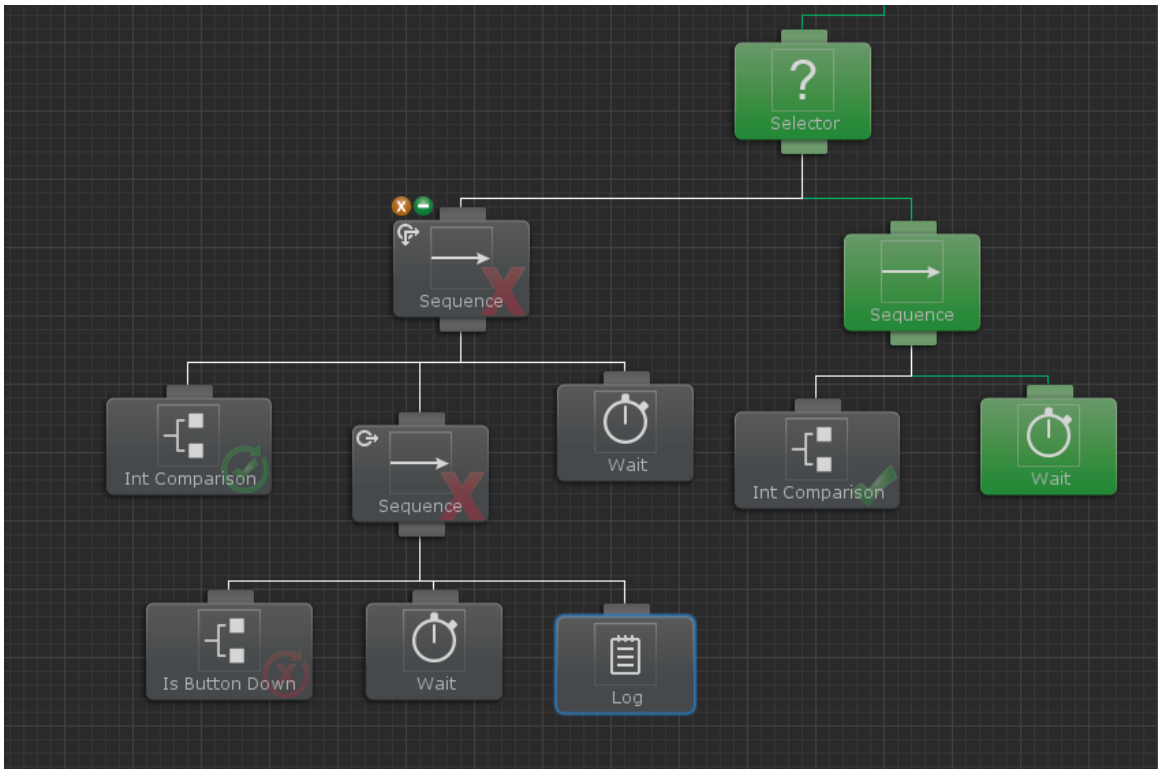


在这个例子中左边的 Sequence 的中断类型为 **Lower Priority**，假设左侧分支返回错误，行为树将跳转到右侧分支！当右侧分支运行时，第一个节点的判断条件变成了 **success**。这时因为判断结果发生变化并且设置了 **Lower Priority**，所以会打断当前正在执行的 **Action** 并返回去执行第一个 **Action**。(译者：如果要自行测试上图效果，建议把 **Action** 换成 **wait** 节点，我在测试的时候用的 **log** 结果 **log** 太快，这个 **LowerPriority** 测试了很久也没搞明白，原来是我的 **Action** 太快，导致行为树结束)；如下图

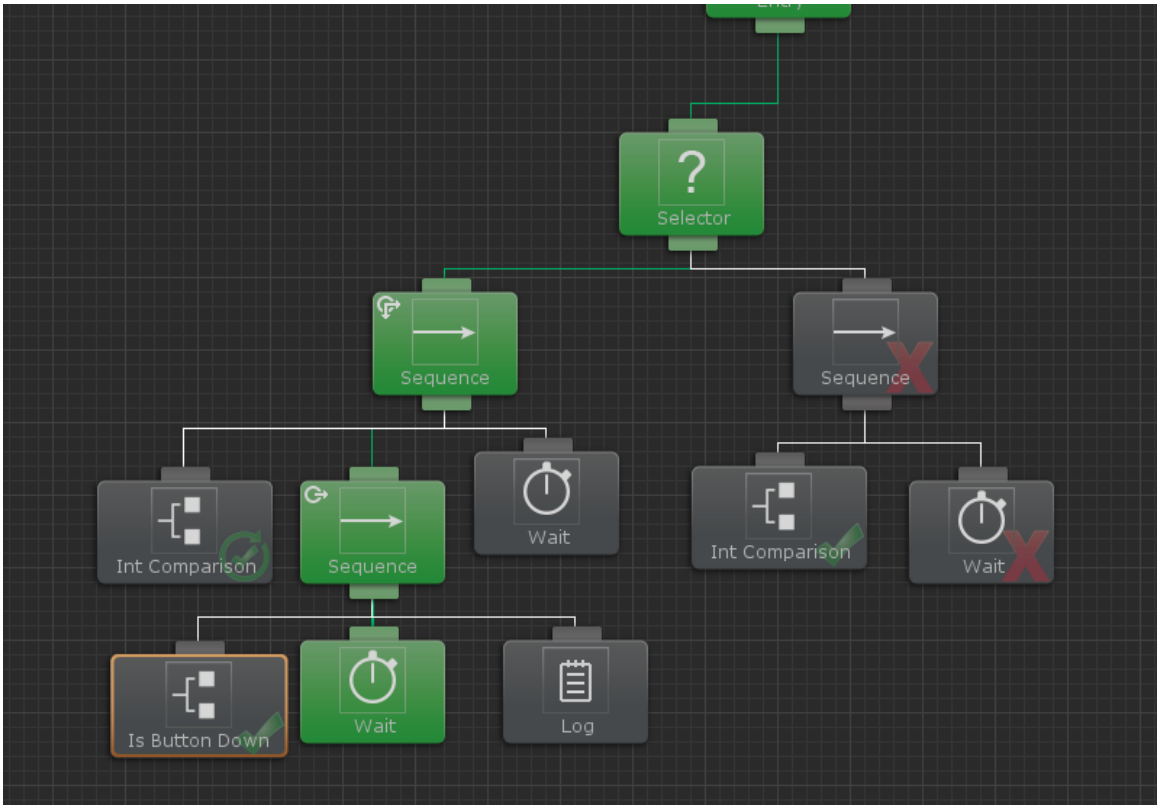


条件在被检测时会有一个图标来标记，表示这个判断节点当前被检测中，当状态变化后会根据打断类型打断行为树当前的执行顺序！上图中左下角的 Int Comparison(整形数值判断)节点，如果判断返回 false 则会打断 wait，并跳转到后一个 Sequence 队列，如果此时判断有变成了有效值则又会跳回来执行第一个 Wait。这是因为第一个 Sequence 选择了 Both 的打断类型！

另外有打断的条件节点可以嵌套。例如下图！



如果按下 Fire1（is Buttondown 监听的是 Fire1），则会跳回到左侧 Sequence 下的 Wait



Event 事件

Behavior Designer 中的 Event 事件系统可以让你很容易的使用！你可以通过代码触发一个 event 事件，也可以通过行为树的节点来触发一个事件！

这些事件可以通过行为树的 SendEvent 节点和 HasRecivedEvent 节点来触发和监听事件！当一个事件要被发送时使用 SendeEvnet 节点。HasRecivedEvent 节点是一个条件节点，当接收到注册的事件后会返回 success。可以通过事件名称的定义来触发和监听一个事件！

出了通过行为树节点来触发事件，还可以通过代码来触发事件！ BehaviorTree.SendEvent 函数就是用来干这个的：

```
var behaviorTree = GetComponent< BehaviorTree >();
behaviorTree.SendEvent< object >("MyEvent", Vector3.zero);
```

上面这个例子就是通过代码，将事件“MyEvent”发送到行为树，并带有参数（Vector3.zero），如果行为树中有监听器，则监听器位置会返回 success！

Task 的引用，任务节点之间的引用！

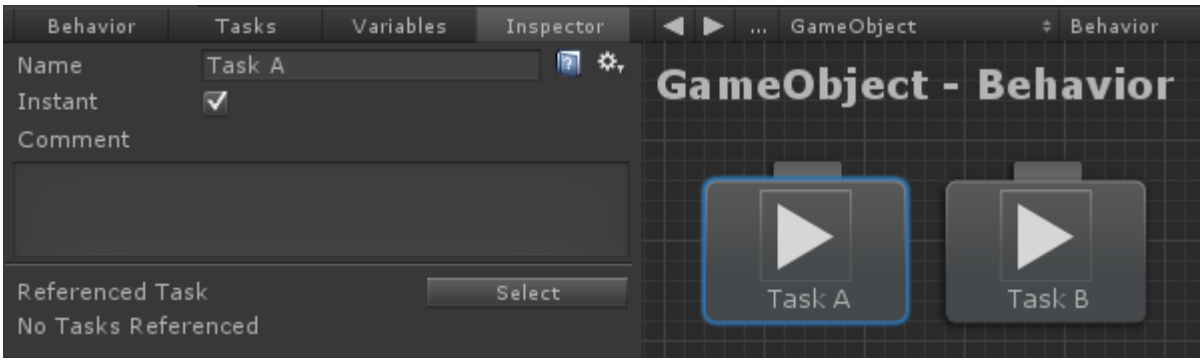
在编写一个 Task 任务节点的时候可能需要访问另外一个 Task 任务。例如 TaskA 想访问 TaskB 的某个属性数值！例如下面这样的 TaskA 和 TaskB

```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;
public class TaskA : Action
{
   public TaskB referencedTask;
   public void OnAwake()
   {
      Debug.Log(referencedTask.SomeFloat);
   }
}
```

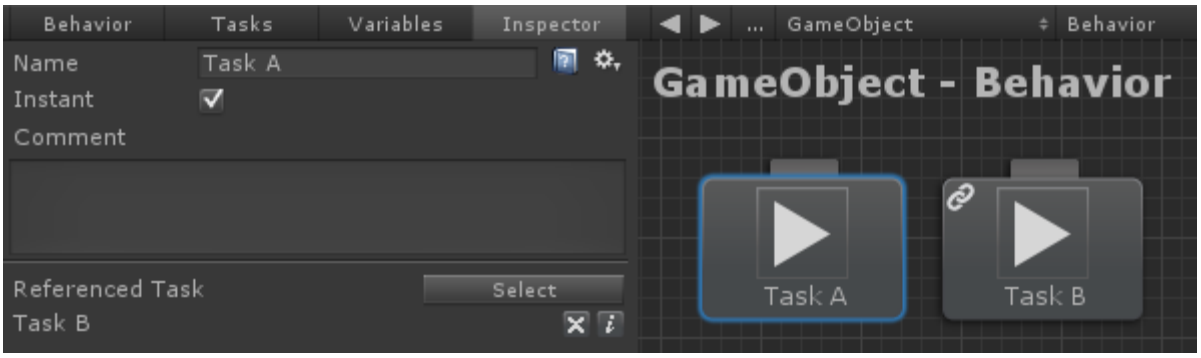
TaskB 然后看起来像：

```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;
public class TaskB : Action
{
   public float SomeFloat;
}
```

将这两个任务添加到行为树编辑器中：



选中 TaskA，你会在 Inspector 面板中看到变量 referencedTask 他是个 Task 类型，这时你可以选择 Select 按钮进行选择，最终像下图这样

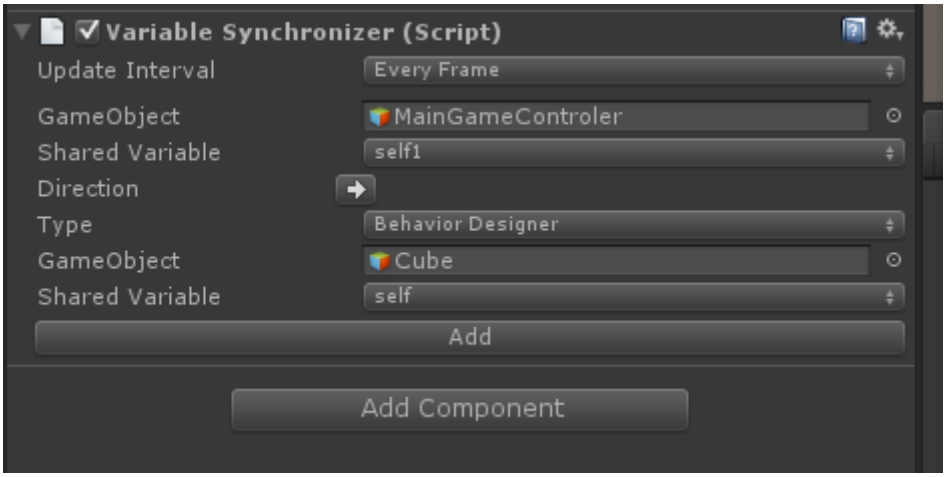


你可以通过点击“X”号来取消引用关系！这样配置后，在执行 TaskA 的时候就会显示 TaskB 的属性，像下图这样

Task 的引用也可以是数组引用，像下面这样！

```
public class TaskA : Action
{
   public TaskB[] referencedTasks;
}
```

变量同步器 Variable Synchronizer



在 GameObject 上挂在脚本，并设置同步的源，以及同步目标，中间的箭头表示同步方向，向右表示上面的变量同步给下面的，向左表示下面的变量同步给上面的！

Task 任务的可用属性

HelpURL：web 连接

```
[HelpURL("http://www.opsive.com/assets/BehaviorDesigner/documentation.php?id=27")]
public class Parallel : Composite
{
////////////////////////////////////////////////////////////////////////////////
```

TaskIcon ： 任务的图标

```
[TaskIcon("Assets/Path/To/{SkinColor}Icon.png")]
public class MyTask : Action
{
////////////////////////////////////////////////////////////////////////////////
```

TaskCategory：任务的显示位置（在 Task 任务面板中的显示位置）

```
[TaskCategory("Common")]
public class Seek : Action
{
[TaskCategory("RTS/Harvester")]
public class HarvestGold : Action
{
////////////////////////////////////////////////////////////////////////////////
```

TaskDescription：功能描述的文本内容，显示在编辑器布局区域的左下角

```
[TaskDescription("The sequence task is similar to an \"and\" operation. ...")]
public class Sequence : Composite
{
////////////////////////////////////////////////////////////////////////////////
```

LinkedTask：应用其他的 Task 任务

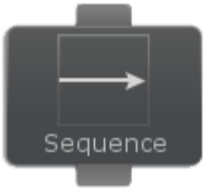
```
[LinkedTask]
public TaskGuard[] linkedTaskGuards = null;
////////////////////////////////////////////////////////////////////////////////
```

InheritedField ： 继承属性

```
[InheritedField]
public float moveSpeed;
```

Composites （复合）节点

Sequence（序列）节点

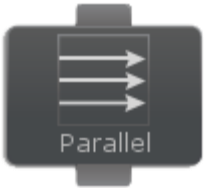


这个节点是一个“和”的关系，也就是他下面的子节点的执行顺序是一个接着一个的！如果其中一个返回 false。那么后续的子节点不会被执行，这个序列节点返回 false。只有当所有子节点全部完成并返回 success 的时候，这个 Sequence（序列）节点才会返回 success;

Selector（选择）节点

这个节点是“或”的关系，也就是他下面的子节点的执行顺序是一个或另一个的！只有所有子节点返回 false 才会返回 false。只要有一个子节点返回 success，那么这个 Selector 节点就会返回 success，后续的节点不会被执行！

Parallel（并行）节点



这个节点类似于 Sequence（序列）节点。不同的是，Parallel（并行）节点会在同一时间执行所有子节点而不是一个一个的去执行！如果子节点中有任意一个返回 false，则停掉所有子节点并返回 false。只有所有子节点全部返回 success 的时候，才会返回 success。

Parallel Selector（并行选择）节点

类似于 Selector（选择）节点，ParallelSelector(并行选择)节点只要有一个子节点返回 success，那么他就会返回 success！不同于 Selector 的一点就是 ParalleSelector（并行选择）节点会在同一时间执行下面的所有子节点，如果有一个节点返回 success，则会停止掉其他所有子节点并返回 success。只有当所有子节点全部 false 的时候才会返回 false！

Priority Selector（优先选择）节点

类似于 Selector（选择）节点，PrioritySelector(并行选择)节点只要有一个子节点返回 success，那么他就会返回 success！不同点在于，子节点的执行顺序不是从左到右的，而是通过优先级来确定的执行顺序！较高的优先级的子节点会被先执行！（译者：优先级在哪里设置的，没有搞清楚，目前测试结果同 Selector 节点，后来还是用我大 Google 搜索到的解决办法！百度就是个垃圾站）
需要在 Task 类中覆盖函数，来设置不同的 Priority,原文地址：

http://forum.unity3d.com/threads/behavior-designer-behavior-trees-for-everyone.227497/page-4

```
// The priority select will need to know this tasks priority of running
public virtual float GetPriority();
```

Random Selector（随机选择）节点

这个节点的特点是：随机的执行子节点，只要有一个子节点返回成功，它就会返回成功，不再执行后续节点。如果所有子节点都返回 false 则它也返回 false！
在这个节点的属性面板中有：seed（随机种子）的设置，自行使用！

Random Sequence（随机序列）节点

类似于 Sequence（序列）节点，只是他的执行顺序是随机的！只要遇到一个子节点返回 false，RandomSequence（随机序列）就返回错误，直到全部子节点都返回 success，它才会返回 success！
在这个节点的属性面板中有：seed（随机种子）的设置，自行使用！

Selector Evaluator（重复判断选择）节点

这个节点每帧都会去重新评估子节点的执行状态并选择。它会执行子节点中优先级最低的子节点！每帧都会这么干！如果当前一个高优先级的节点在运行并且下一帧要执行的子节点优先级比较低，那么它会打断高优先级的节点，去执行优先级低的子节点！ Selector Evaluator（重复判断选择）节点会从低到高的去遍历执行所有子节点，直到遇到一个返回 success 的！如果所有子节点都返回 false，那么它就返回 false！否则只要有一个返回 success，它就会返回 success！这个节点模拟了条件打断功能，如果子节点没有条件节点的话！

Conditionals（条件判断）节点

条件节点的任务是判断游戏的一些属性，比如玩家是否活着，怪物是否在视野距离内！

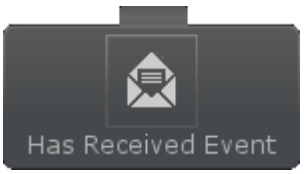
Random Probability (随机概率)节点

通过设置 *successProbability* 属性来控制返回 success 的几率（默认 0.5，也就是 50%几率）！另外还有 seed 随机种子的设置等！

Compare Field Value（字段比较）节点

比较指定的值的字段值。 返回成功如果值是相同的。

Has Received Event（是否接收到事件）



（译者：还有很多条件节点，这里就忽略了！）

Decorators（修饰器）节点

这种节点的功能是用来包装另一个节点！（只能有一个子节点）。Decorators（修饰节点）将改变节点的行为！例如：修饰节点可以再运行时控制子节点直到返回某个特定状态（success 或者是 false）。后者是对子节点返回结果取反（即：success 返回 false，false，返回 success）；下面来一一介 BehaviorDesigner 默认自带的几个 Decorator（装饰器节点）

Conditional Evaluator (条件节点的评估) 装饰节点

参数设置：

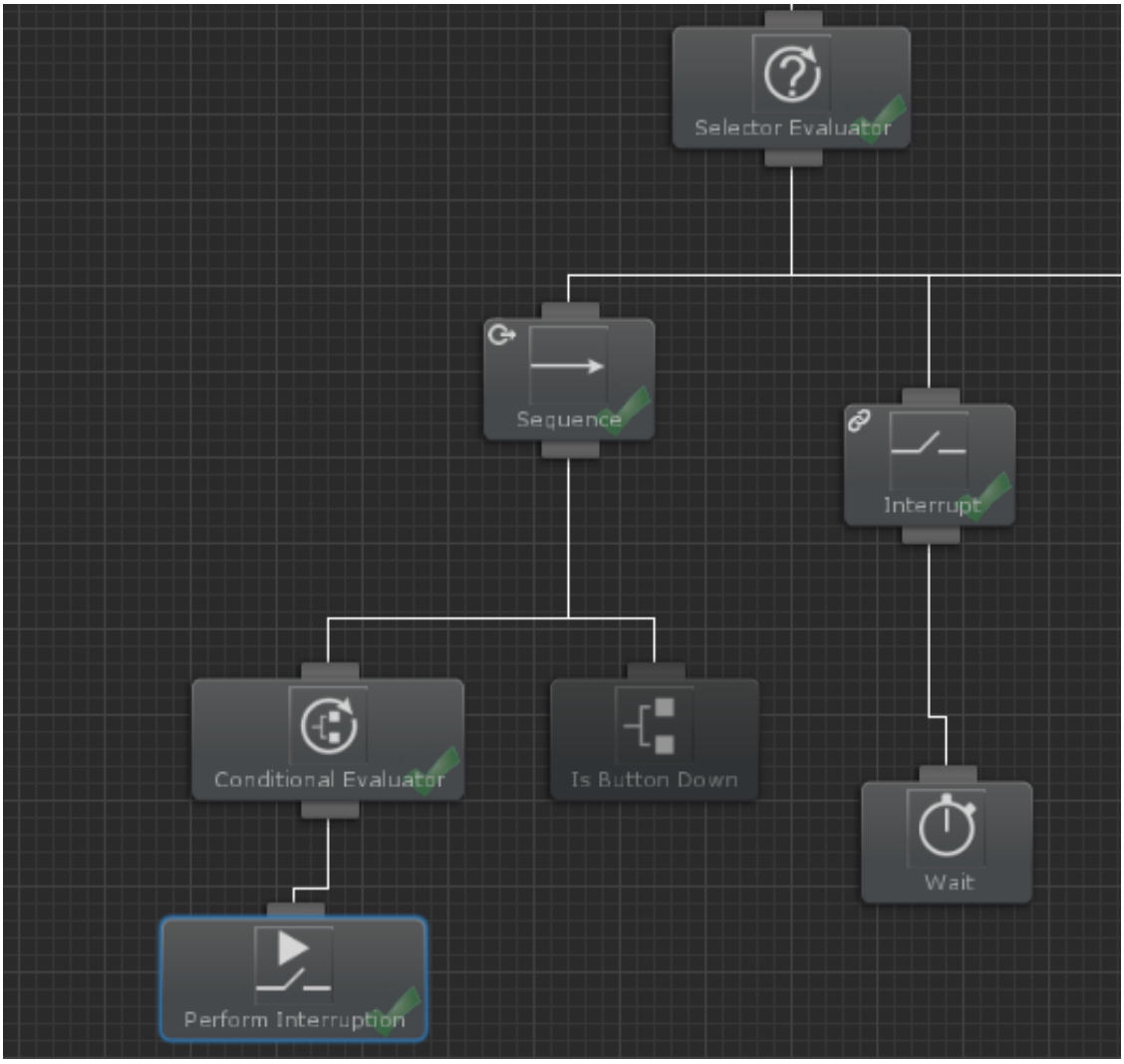
1：reevaluate ：条件节点是否需要每帧都重新评估一次

2：*conditionalTask*：要被评估的条件节点，注意：这个节点本身就是个条件节点！

对设置的条件节点进行评估，如果条件节点返回 success，那么运行子节点并返回子节点的运行结果！如果条件节点没有返回 success 那么子节点不会被运行，并且立刻返回 failure！条件节点只会在开始运行的时候被评估一次！

Interrupt（打断）装饰节点

如果打断节点被触发，则打断下面的所有子节点任务的执行！打断命令可以被 Perform interruption（执行打断）节点发起！打断节点在收到打断命令前，不会打断他下面的子节点的执行状态！如果子节点执行完毕还没有收到打断命令，则直接返回子节点的执行结果！例如下图这样：

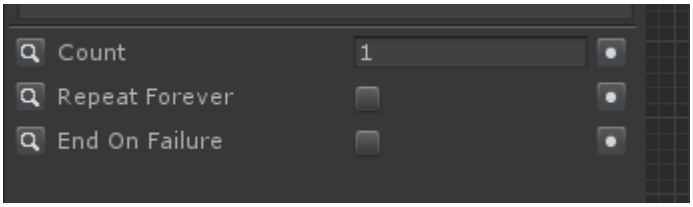


Inverter（取反）装饰节点

子节点的任务完成后返回值，在这个节点会被取反并传递到上一级中！

Repeater（重复/循环）装饰节点

有三个属性设置：执行次数，是否一直重复，运行直到返回错误！



Return Failure （返回失败）装饰节点

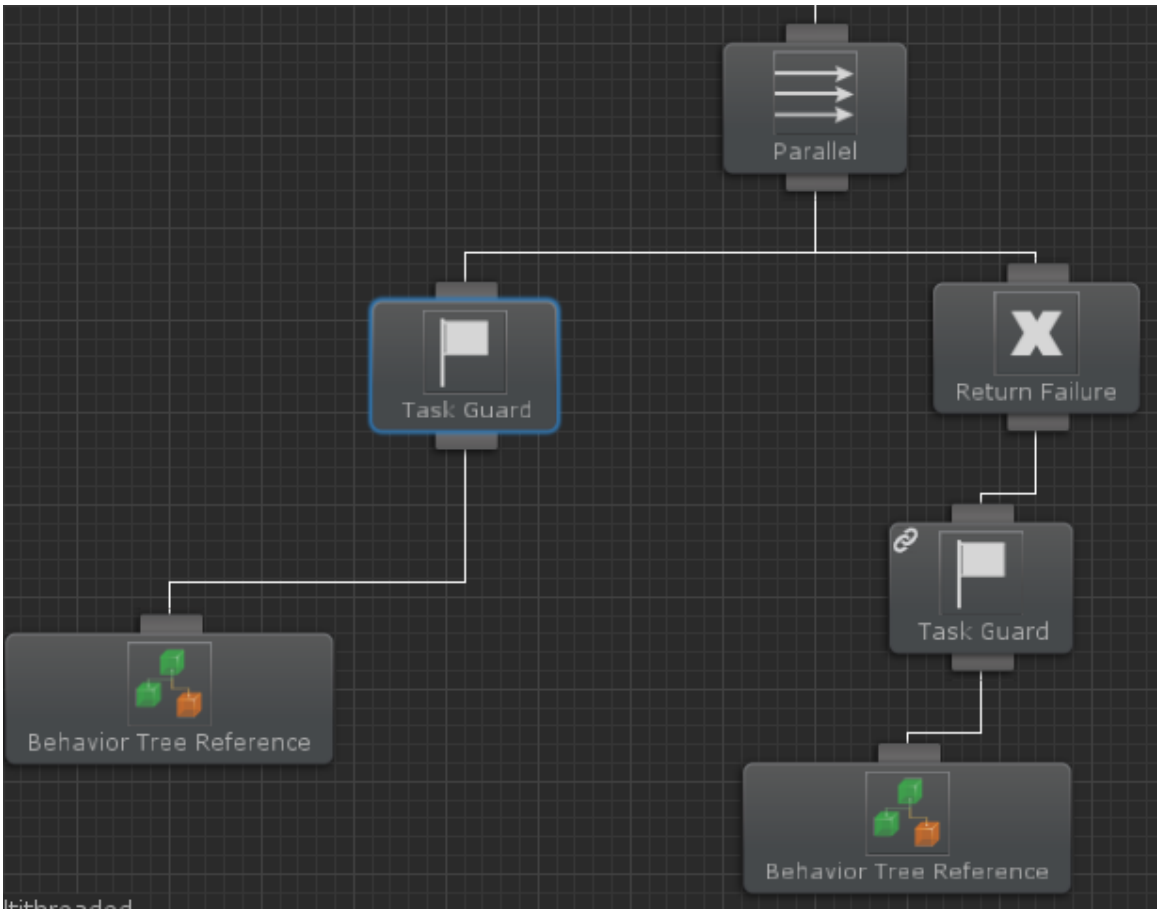
只要子节点当前的状态不是 running，也就是子节点执行结果无论是 success 还是 failure，都返回 failure！如果子节点状态是 running 的话则返回 running！

Return Success（返回正确）装饰节点

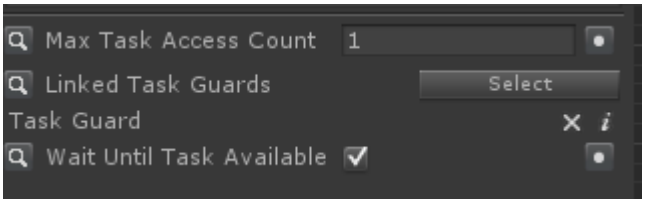
只要子节点当前的状态不是 running，也就是子节点执行结果无论是 success 还是 failure，都返回 success！如果子节点状态是 running 的话则返回 running！

Task Guard（任务守卫）装饰节点

类似于多线程互斥操作中使用的 Lock 标记，为了避免公共数据被多次引用！下图以外部行为树为例进行演示！这里使用并行触发两个外部行为树，如果不加上 TaskGuard，那么两边都会去并行执行外部行为树，现在加上 TaskGuard 后同一时间只能执行一个，而另一个要等待执行完毕才能执行

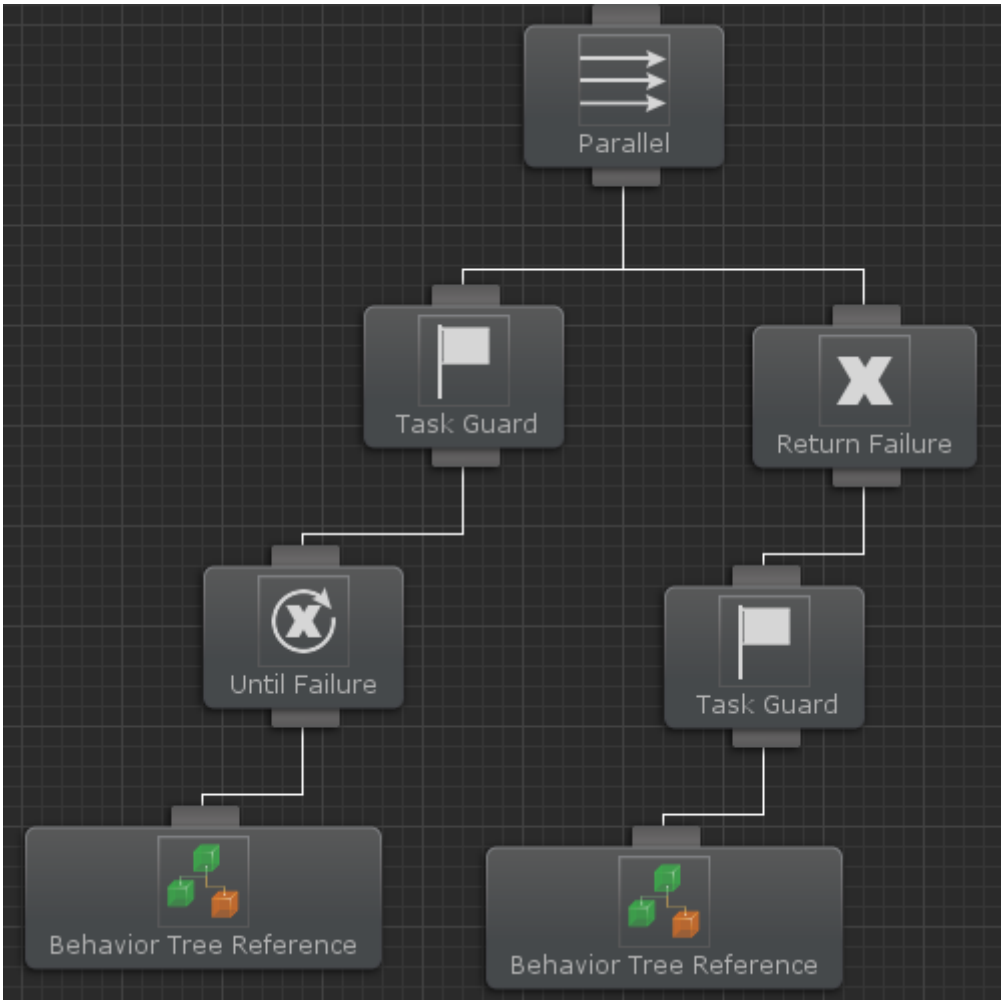


上图 TaskGard 配置如下图：



Until Failure（直到失败）装饰节点

直到子节点返回 failure，否则一直循环执行子节点， 如下图：



Until Success（直到成功）装饰节点

直到子节点返回 success，否则一直循环执行子节点！