声明：本文档由(炼金师-金奇 译）文档内容并没有进行严格的校验可能存在错别字等，如果文档中任何有异议的地方，请参见官方文档！如果你觉得本插件好用，请支持正版！请支持正版！请支持正版！NodeCanvas 交流 QQ 群:117182310

NodeCanvas 离线帮助文档

1：入门

1.1 框架概念

本节将介绍 NodeCanvas 框架的核心概念，并尽可能详细的介绍工作原理以及背后相关的知识点，这里介绍的内容适用于所有 NC 创建的子系统（行为树 BT，状态机 FSM，对话树 DT）

Agents（代理，就是 GraphOwners 所挂载的 gameobject）

Agent（代理）是一个核心术语，用来描述某些操作（Action）所依赖的‘object’对象！例如“谁”将被某个操作处理！或者“谁”将被某个判断条件所检查！在 Nodecanvas 中，Agent 是一个 component 对象的引用（GraphOwner？），而不是一个 GameObject 的引用。这样做的目的是能够提供更强大的灵活性。只有当使用到 Agent 时才会把 Agent 的引用传递到 Nodecanvas 中去！

Blackboards（黑板）

黑板是用来存储/获取数据的对象！它被用于在不同的节点或 Task（任务）间进行数据共享操作。你也可以通过一些简单的 API 对黑板进行数据访问操作！Blackboard（黑板）的引用还有在被使用的时候才会传入到 Nodecanvas 中。你可以在 GraphOwner 中使用黑板中的这些数据！

GraphOwners（图形拥有者）

顾名思义，图形拥有者就是你所创建的图形（行为树图，状态机图，对话书图）的拥有者！它是一个 component 组件！它包含了一个可执行的图，以及一些资深的状态设置！想要使用 NodeCanvas 提供的这些图，去使用它们对应的 GraphOwner 是最方便的方式！当然这不是强制的，你可以在不使用这些 Owner 的情况下去使用这些 Graph 图！

绑定了 GraphOwner 的 GameObject 在 Hierarchy 中的显示会在最后面显示一个小图标！

Graphs（图）

Graph（图）包括了对应系统（行为树，状态机，对话树）的所有节点以及功能。当运行某个图的时候，它会给某个 Agent（代理）提供服务，并使用特定的 Blackboard（黑板）。在 Graph 中同一时间只能处理一个 Agent(代理)。如果你使用 Graph（图）对应的 Owner 的话那么这些处理代理的操作将会是自动执行和维护的！（具体参见:GraphOwner.cs 中的相关定义）

Graph（图）可以绑定到一个代理上，如果不绑定也可以把图保存为独立的 Asset 序列化文件！绑定后的图会绑定到 GraphOwner 所挂载的 GameObject 中，最后连同这个 GameObject 一同保存到本地！如果保存成 Asset 序列化文件的话，那么这个图将可以作为共享资源在不同的 GraphOwner 间进行使用！使用资源文件的好处是可以动态更新图！

Nodes（节点）

节点依托于图（行为树图，状态机图，对话树图），根据图的类型不同，可以使用的节点也有所不同！在 NodeCavnas 中 像“what”和“if”这种功能多数情况下不会包含在 Node 节点中，而是在一个 Task 中！换句话说，Node 节点不包含任何 Action（操作），只是一个容器，用来存储所有其他的 Action（操作）。这么做能最大限度的解耦掉 Task，让 Task 的实现保持相对独立！

Tasks（任务）

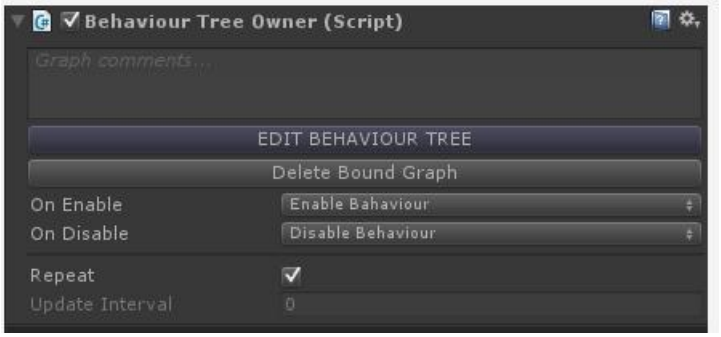
Task 可以是 Actions（行为/操作）也可以是 Conditions（条件判断）。Task 会绑定到 Node 节点上，并最终被执行！当某个 Task 被执行或判断的时候，它总是被某个 Agent（代理）和 Blackboard（黑板）使

用！默认情况下，这个 Agent（代理）和 Blackboard（黑板）只被当前的 Graph（图）使用，但是你可以在别的图中通过 overriden（覆盖）代理来使用这个代理和黑板，同样的你可以在当前的 Task 中通过 overriden（覆盖）来使用别的 Agent（代理）和 Blackboard（黑板）.在不同的项目中，你需要为你的项目创建不同的 Task 来实现你自己所需要的功能！

1.2 The GraphOwner Component （GraphOwner 组件）

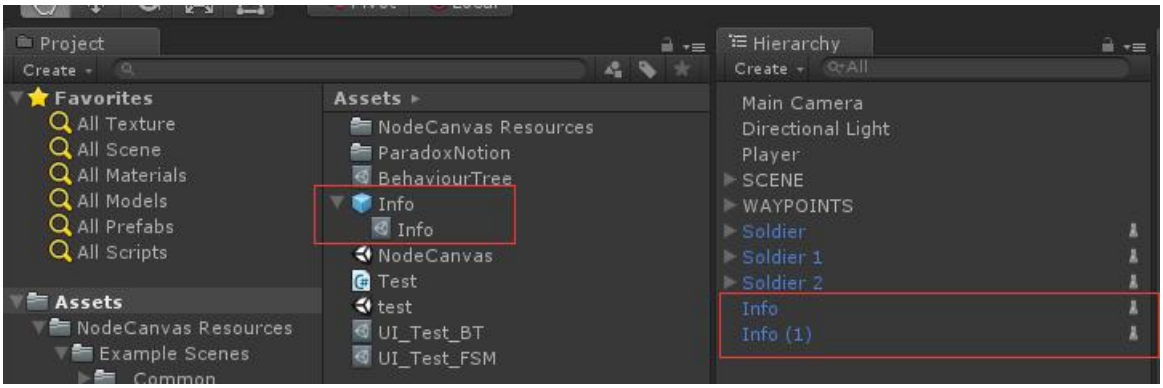
GraphOwner 组件负责创建子行为系统，包括 BehaviorTreeOwner 的行为树，FSMOwner 的状态机.这些组件并不包含 graph 图，它们只会被分配一个图！

* 想要创建一个带有 BehaviorTree 的 gameobject，你需要给这个 gameobject 挂上一个 BehaviorTrreOwner 组件。
* 想要创建一个带有 FSM 的 gameobject，你需给这个 gameobject 挂上一个 FSMOwner 组件。
* 你可以同时绑定两个类型的组件，也可以绑定多个同类型的 GraphOwner 组件在同一个 gameobject 上！



一旦你添加了 GraphOwner 组件，你就需要给它指定一个 Graph。这个指定的 Graph 可以是 Bound（绑定的）也可以是 Asset 资源（非绑定）。

*** A Bound Graph** （绑定的图）会和 gameobject 一同保存。如下图：



这个绑定的图只在 Preject 中能够看到，在 Hierarchy 中是看不到的！

*** An Asset Graph**（独立资源的图）是一个图例的磁盘文件，例如上图中的“UI_Test_BT”和“UI_Test_FSM”.这个资源文件无法直接拖拽到场景中进行使用！这么做的好处是，你可以把同一个图分配到多个的 Owner 上！

通过上图编辑器的 UI,你可以随时切换到绑定或者是独立资源，即便是已经有了图！

所有的 GraphOwner 组件，都有两个按钮给你设置（OnEnabel/OnDisable），用来控制当 gameobject（enabled/disabled）的时候 GraphOwner 如何处理！你可以点击查看所有选项！

最后，每当你绑定一个 GraphOwner 组件的时候，一个 Blackboard 组件会被自动添加到 gameobject 上！这个 Blackboard 就是用来共享数据用的！每个 gameobject 上只应该有一个 Blackboard 组件！

1.3 The “Self” Parameter （self 参数）

默认情况下 Task（任务）会使用当前图所在的 gameobject 对象作为处理操作的代理对象。你也可以通过 override（覆盖/重新指定）这个参数，从而指定另一个对象（也就是另一个代理）！当你要重新指定代理的时候，你有两种方式可选：

* 直接指定一个对象（代理）
* 从 Blackboard（黑板）选择一个对象，作为代理对象！

下图所示，注意 这里的 Transform 是类定义中使用的类型，uese self 也可以是别的类型，根据具体 Task 定义不同而不同，下图中使用的 DebugLogText 类定义 `public class DebugLogText : ActionTask<Transform>{.....}`

在上图的这个 Task(DebugLogText)的属性面板中 括号中的红色（Transform）是需要的类型，只有类型赋值正确才行，否则会报错并一直显示成红色！

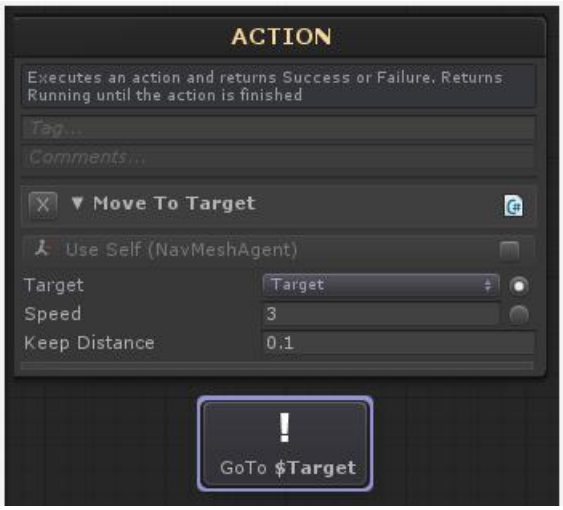
1.4 BBParameters （BB 类型参数）

在 Nodecanvas 中大多数设置参数的地方你会经常遇到这种叫做 BBParameters 类型的参数！这种类型的参数是一种访问 BlackBoard 的参数（也就是这种类型的参数和 Blackboard 中的数据是绑定关系）！这个类型很重要，应用面很广，希望大家记住！

想要使用这种类型的参数，你只需要点击相应参数后面的圆形单选按钮即可，点击后会有一个下拉菜单可供选择，下拉菜单中提供了当前 Blackboard 中可用的数据！如上面第二张图和第三张图中的 Transform！

通常这个下拉菜单提供了如下三种选项：

* 从 Blackboard 黑板中选择一个变量进行绑定！这也是最常用的选项！
* 在运行时定义一个“Dynamic Variable”变量
* 从 Global Blackboard（全局的 Blackboard，这个全局黑板是单独创建的，绑定在当前场景全局唯一的一个 gameobject 上，用于全局数据共享）中选择一个变量进行绑定。



当你使用上面的任意一种方式选择了某个变量后，你会注意到在 Graph 图中你的 Node 节点中会显示这个 Task 的相关信息，并且使用$符号进行了字符串链接，这个信息会自动生成，使用了节点名称，以及 Task 功能！明确的告诉使用者，这个 Node 节点当前有哪些 Task，以及这些 Task 的具体操作信息，很方便！

1.5 Visual Debugging（可视化调试）

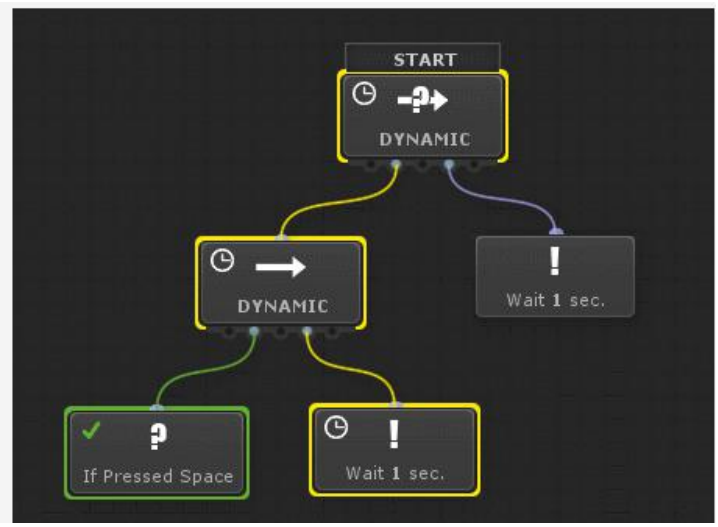
Nodecanvas 拥有很棒的信息提示功能以及运行时可视化调试功能！在 play mode（播放状态）下运行某个行为树，你会看到在 Graph 编辑器窗口中究竟发生了什么，不同的 Node 节点在不同状态下会显示不同的颜色！不同颜色定义如下：

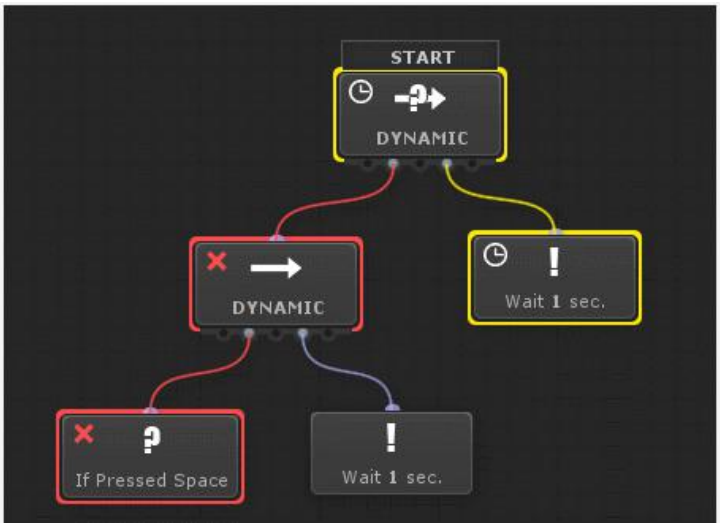
* Blue（蓝色）表示静止状态！

* Yellow（黄色）表示正在运行，沿着时钟图标  执行

* Green（绿色）表示执行成功，沿着绿色的勾执行

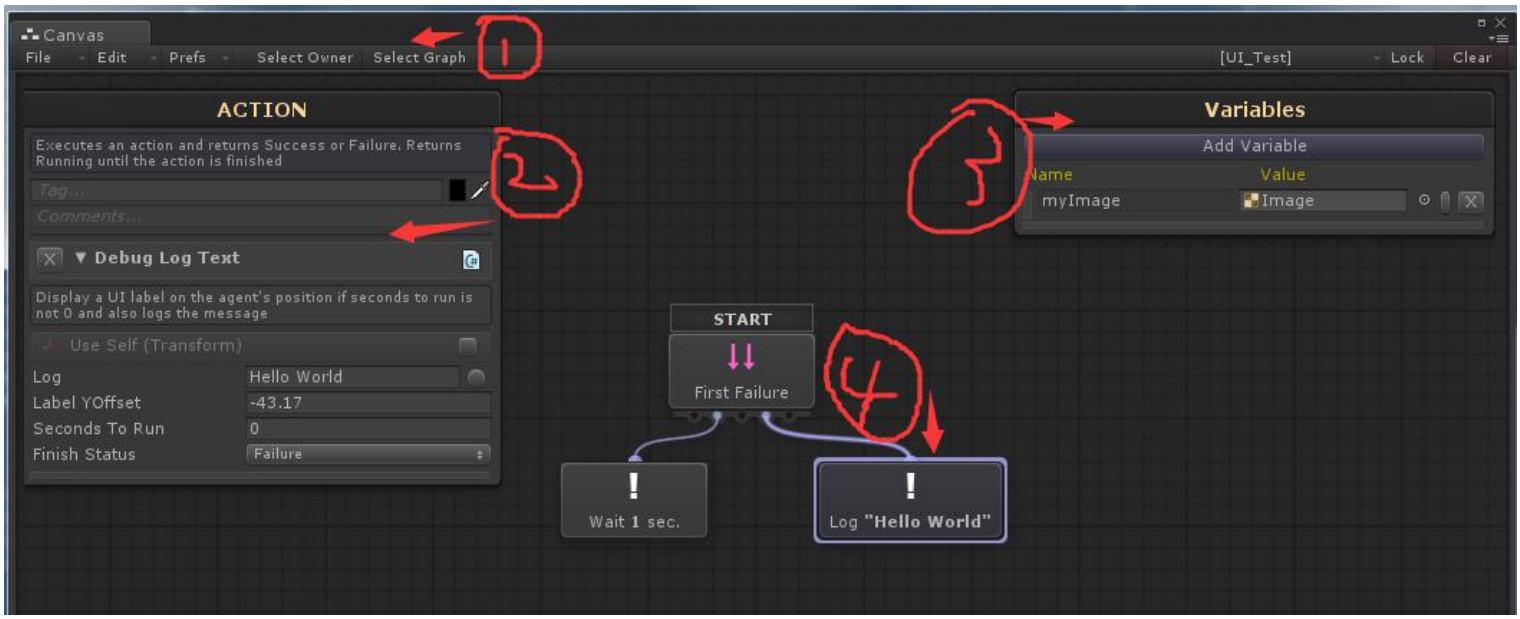
* Red（红色）表示执行失败，沿着红色的执行





还有一点要知道，就是在运行时你可以随意改变节点顺序，增加/删除节点，修改连线顺序，等等所有操作！

2.The Editor（编辑器介绍）



1：Toolbar （工具栏）

File (文件)

* ImportJSON：你可以导入一个保存为 json 格式的 graph！

* ExportJSON：把当前的图导出到外部的一个 json 文件中，方便别的地方导入使用！

Edit (编辑)

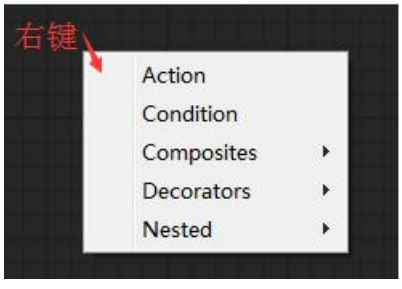
* Bind To Owner：把当前的 graph 图绑定到 GraphOwner 上

* Save To Asset：把当前的 graph 图保存到独立的.asset 文件中！

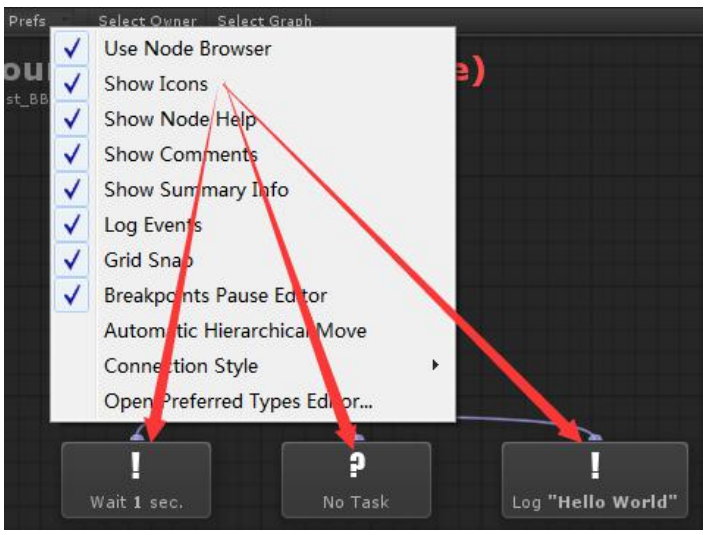
* Create Defined Variables：把当前图中使用了但是未定义的变量直接在 Blackboard 中创建出来！

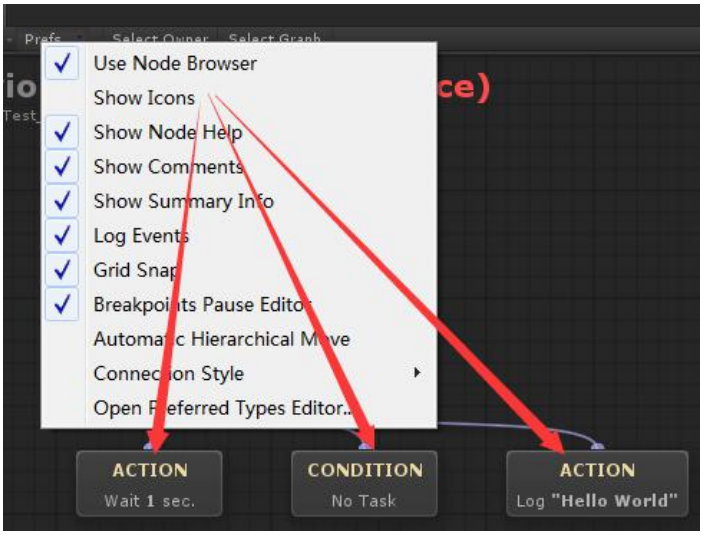
Prefs（首选项）

* Use Node Browser：在图中空白区域鼠标右键的显示方式（菜单/对话框）如下图：

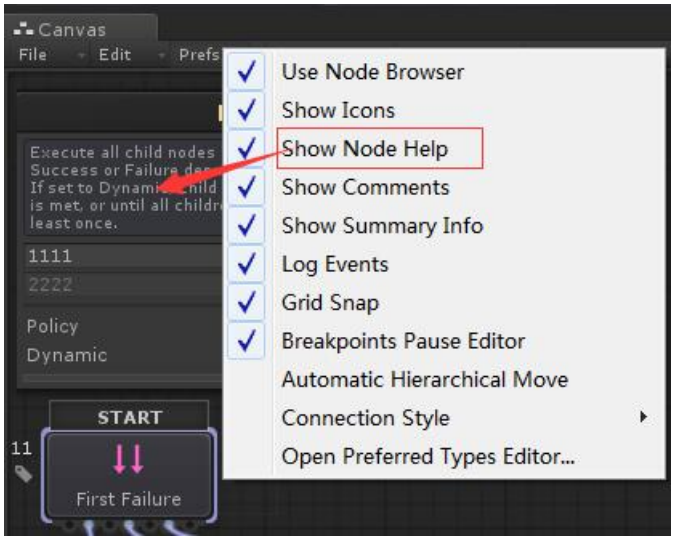


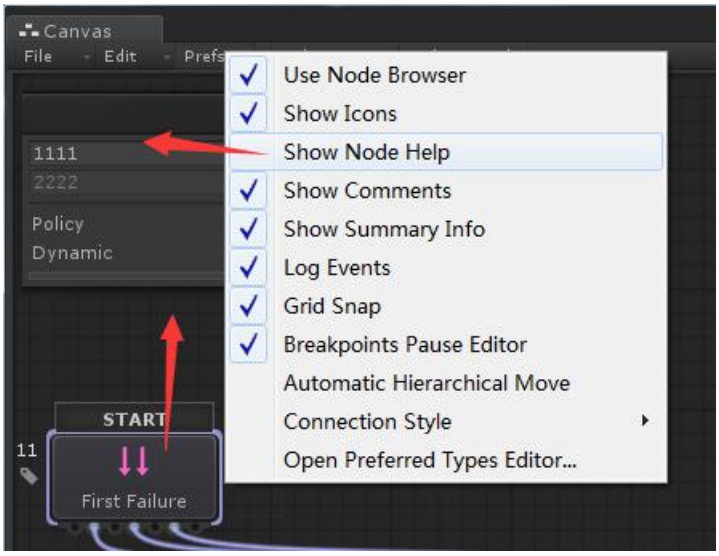
* Shwo Icons：是否显示节点中的图标，如上图中的“叹号图标”！如果不显示图标则显示他们的类型，如下图：



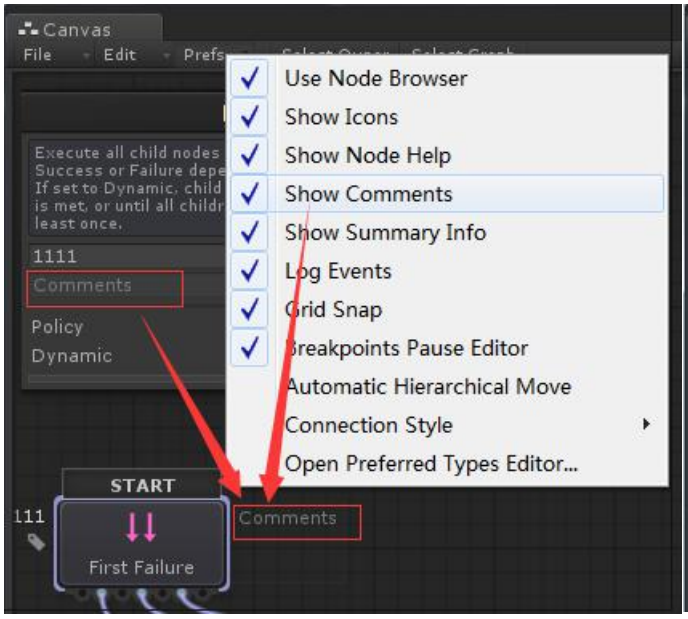


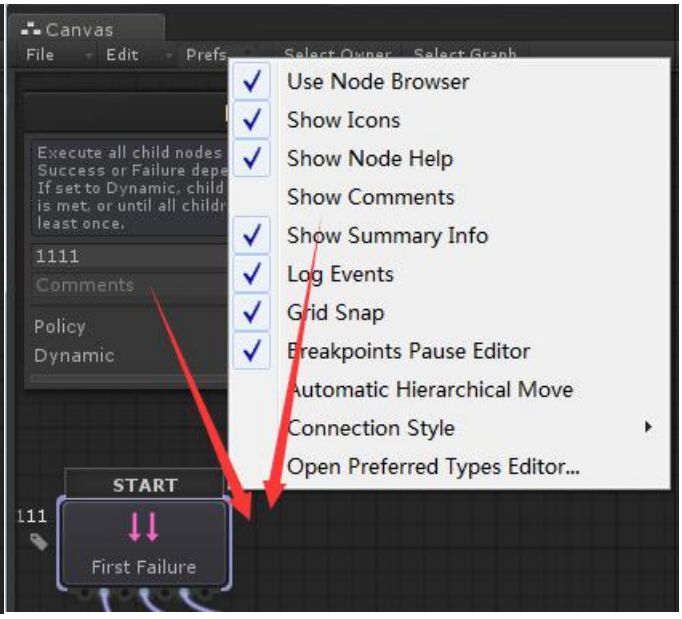
*** Show Node Help：是否显示节点的帮助信息**



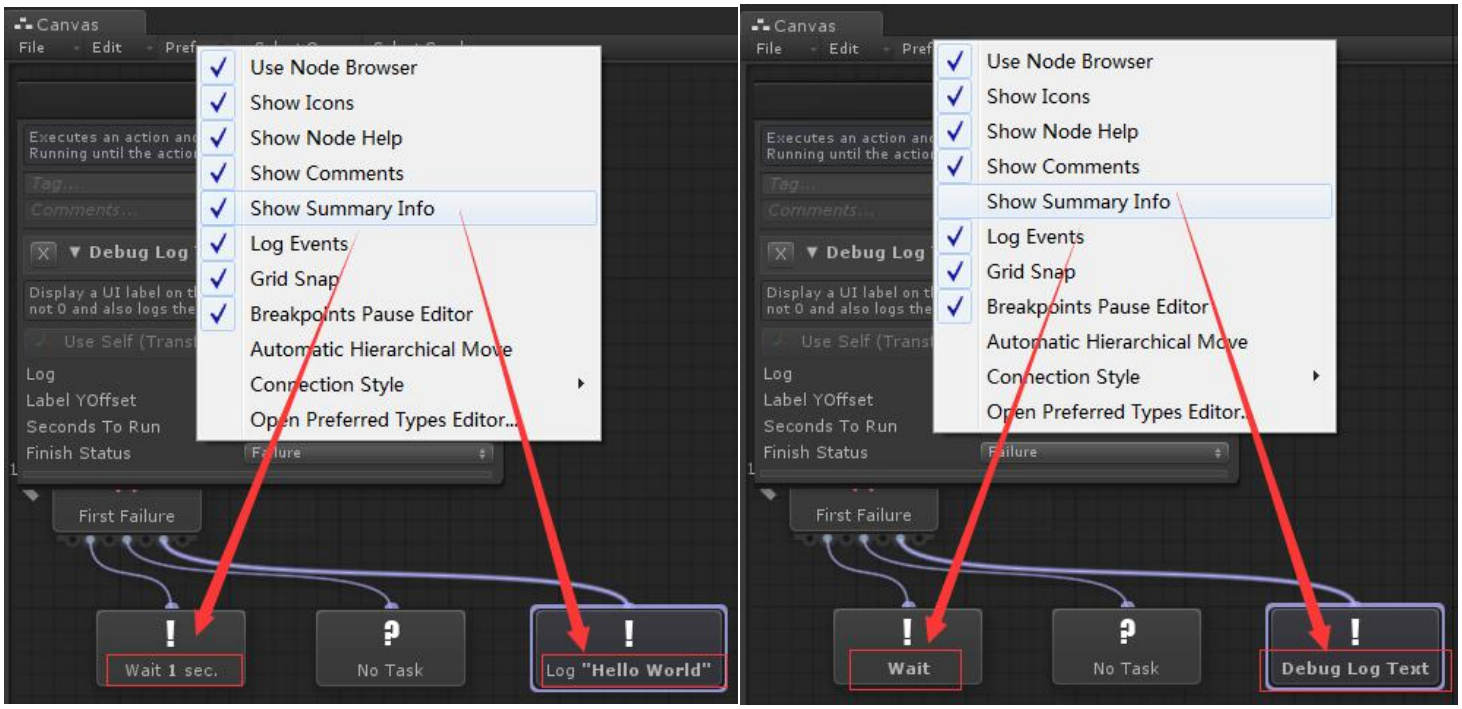


***Show Comments：是否显示注释**

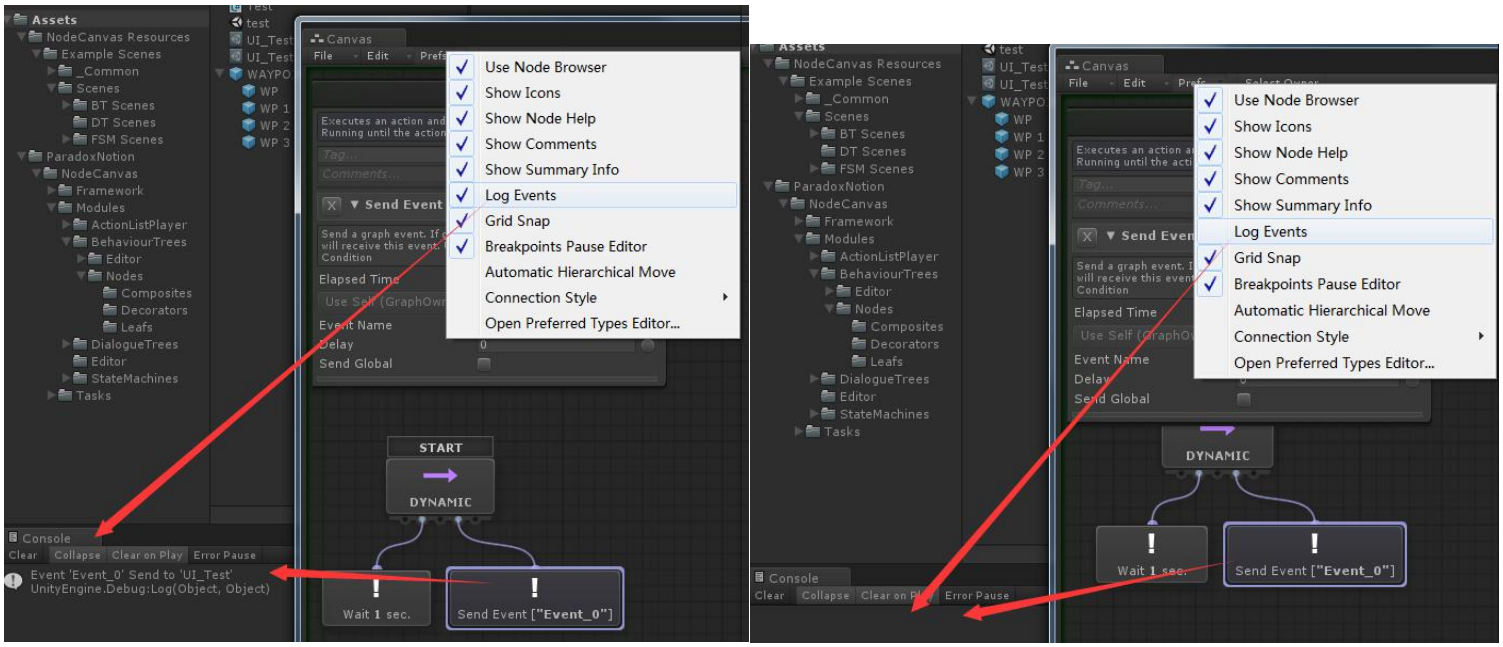




***Show Summary Info 显示摘要信息**



*** Log Events 是否打印 Event 消息（只控制跟 Event 相关的消息，其他 Log 信息正常打印）**



*** Grid snap：在编辑（拖放）Node 节点的时候是否自动对齐到网格**

*** Breakpoints Pause Editor: 断点触发时是否暂定游戏！**

*** Automatic Hierarchical Move：移动某个节点时，这个节点下的子节点是否跟随移动！**

*** Connection Style：节点链接线条样式（弧线，垂直线，直线）**

*** Open Preferred types Editor....：打开数据类型管理器**

2.Node Inspector Panel （节点属性面板）

当选中某个节点后，在编辑界面左上角会显示一个面板，这个面板中包含了当前节点的所有信息，包括节点下的 Task 属性！

3.BlackBoard Variables Panel (黑板属性面板)

在编辑界面的右上角是当前 graph（图）可用的数据，这些数据就是 Blackboard 中的数据！点击 Add 按钮可进行添加变量操作！

4：Canvas 画布

这里说的 Cavas 画布就是主要编辑区域，用来摆放所有的 Node 节点！右键点击画布空白区域可以添加需要的节点！你可以通过右键某个节点进行节点操作（包括添加断点，禁用/启用，设置为 Start 等等）

2.1 Controls &Shortcuts （快捷键和常用操作）

* 右键点击 Canvas（画布）添加需要的 node（节点）。
* 可以选中并拖拽节点
* 鼠标中间+拖拽可以移动 Canvas（画布）
* Alt + 鼠标左键，然后进行过拽，也可以移动 Canvas（画布），没有鼠标中键的时候可以使用这个命令
* Shift + 鼠标左键，然后拖拽某个 Node（节点），会连同这个节点下的所有子节点一起移动！
* 拖拽某个连接端口到另一个端口时，将会链接这两个端口！右键点击某个连接端口，可以取消连接操作！
* 拖拽某个连接端口到空白区域时，弹出对话框进行选择（可连接的 Node 节点）
* 点击已经连接的端口，可以选中这根连接线并查看相关属性！
* 删除某个节点是，可以通过键盘的 Delete 键，也可以通过右键点击节点，从弹出菜单中选择删除！
* 复制节点使用 Ctrl +D.
* 双击某个节点，会在 IDE 中打开这个节点的代码！如果是嵌套节点的话会打开所嵌套的节点树！
* Relink a connection by click and dragging it over a new node.
* “F”键会把显示位置定位到你当前图中所有节点占用区域的中心点！
* 鼠标滚轮可以用来缩放 Canvas 画布！
* 按住左键拖拽可以选择多个节点
* 按住 Ctrl 然后再按住鼠标左键进行拖拽会创建一个 Group 组（一个新的节点），用来管理这个组里的所有节点！

右键某个节点后会有一个弹出菜单，包含的功能选项如下：

* Delete：删除当前节点
* Copy Node：拷贝一份当前节点，（支持多选拷贝）
* Copy Assigned Task: 拷贝节点下所有的 Task（任务）
* Paste Assigned Task: 粘贴拷贝的所有 Task 到当前节点中

拷贝/黏贴 Task 的操作可以在不同的 graph 图中使用。例：把图 A 中某个 Node 上的 Task 拷贝到图 B 中某个 Node 上！

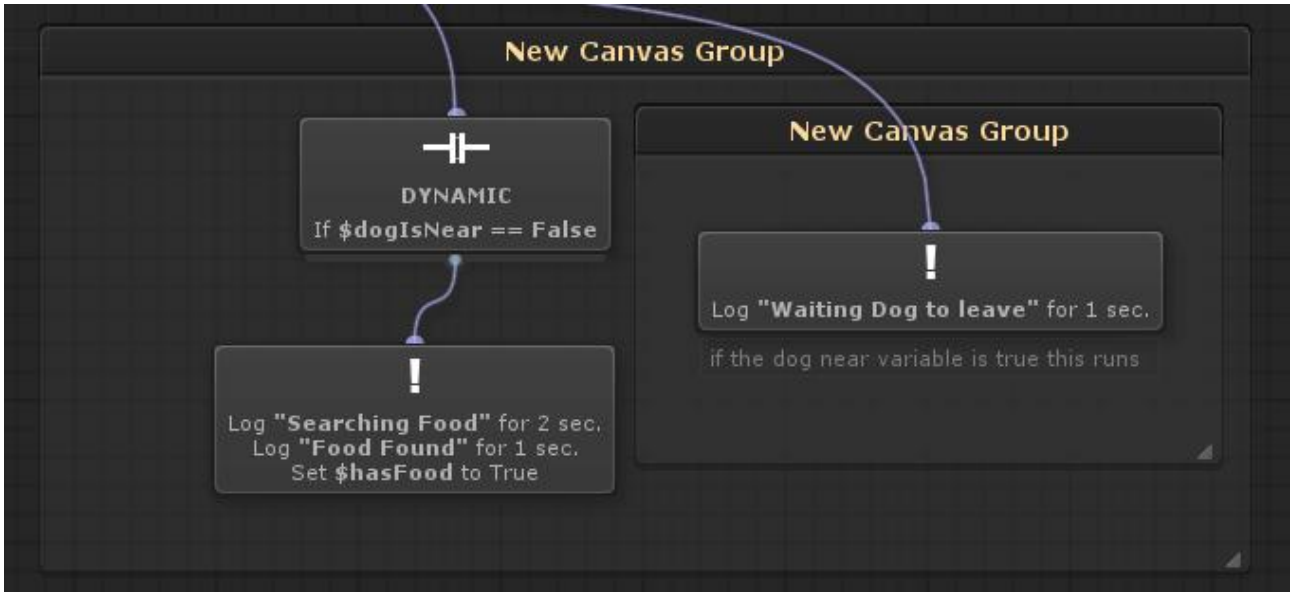
2.2 Assigning Tasks（添加 Task 任务）

所有的 Node 节点都要分配一个 Action 或者是一个 Condition，所有这些添加的 Action 和 Condition 统称为 Task（Node 节点只是一个空壳，需要添加不同的 Task 来确定这个节点具体是个什么节点）。当节点下是空的没有任何 Task 的时候会有一个 Add 按钮让你添加可选的 Task。

当你添加了某个 Task 后，就会显示这个添加的 Task 的相关属性，如上面的第二张图！

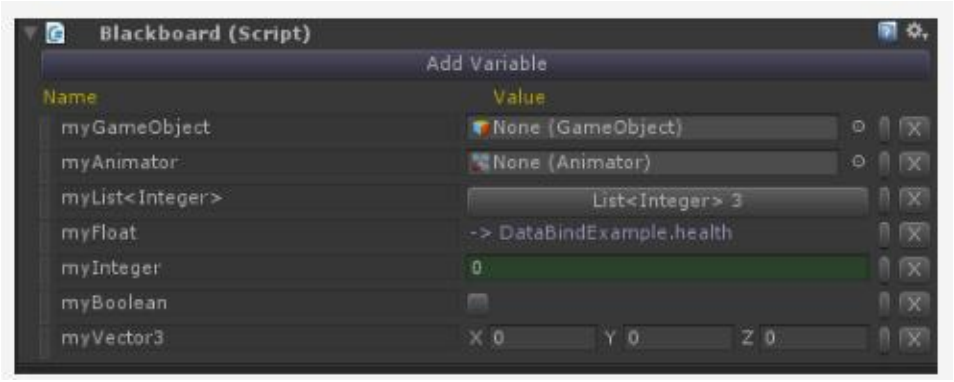
2.3 Canvas Group(Canvas 组）

在上面的快捷键命令章节讲过了这个组的创建方法，这里就不在阐述了！最终效果如下图：



按住 shift 拖拽 group 可以调整 group 的位置，group 之间也可以嵌套使用

3：The Blackboard (黑板)



黑板的作用就是用来交互数据用的！如果一个 gamobject 对象上拥有多个 GraphOwner，那么这个黑板是可以被共用的！你除了在 graph 图中使用黑板中的变量外，还可以通过简单的 API 在代码中进行变量的读写操作！

黑板可以存储任何数据类型的变量！在编辑器中你点击“Add Variable”后你会看到类型列表，如果没有需要的类型你只需要添加进去就好了！通过类型编辑器（Preferred Types Editor）！

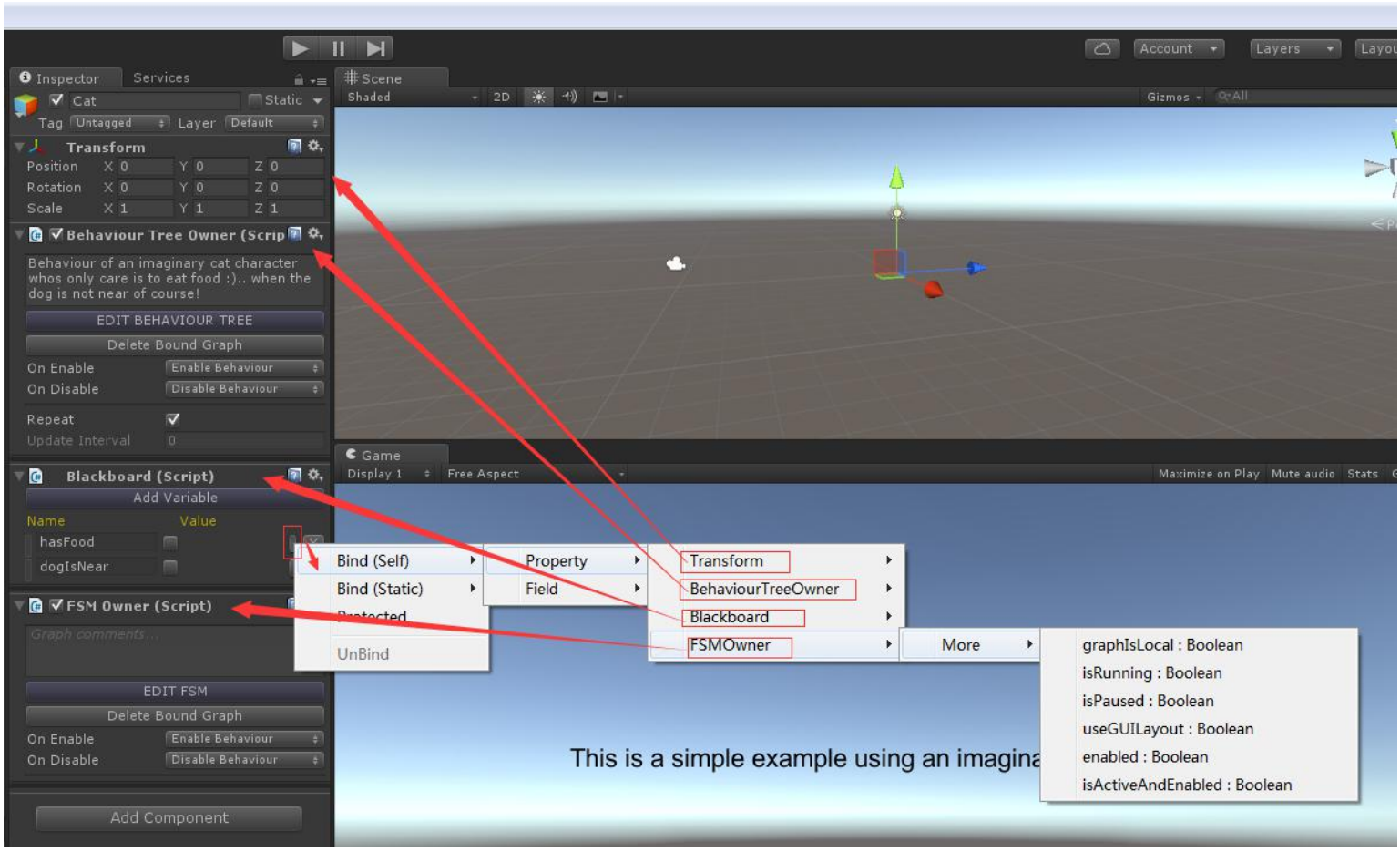
黑板中的数据可以被保存/加载。可以在运行时跨场景使用，甚至是保存到本地文件中！

黑板中的变量支持 DataBoud（数据绑定）操作。可以和当前对象身上的任何组件中的 public 属性进行绑定。这极大的方便了数据访问和操作！（译者：是否是线程安全的，有待测试！）

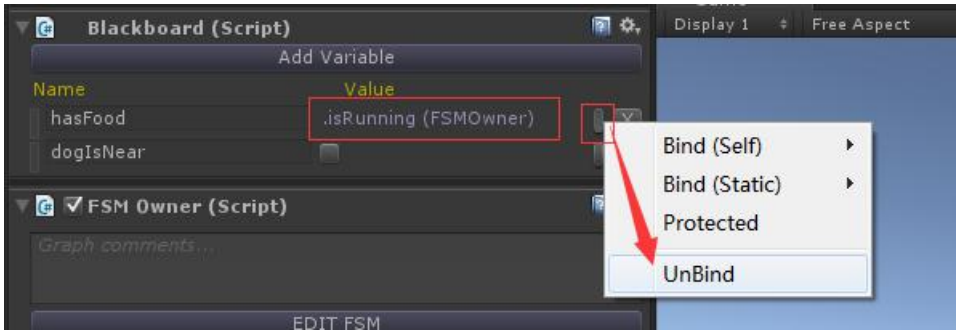
3.1 Data Binding Variables（数据绑定）

当某个变量被添加到黑板中后，就可以和当前 gameobjct 身上其他的 componets 上的 public 属性进行绑定操作了！绑定后的数据访问可能是单向的 get 也可能是双向的 get/set，这取决于绑定的属性是只读的，还是可读写的！通过属性的 Get/Set 操作对数据进行读写操作！通过黑板来绑定其他组件中的属性并进行访问，这个功能很强大！你可以直接在当前 graph 图中对当前 gameobject 身上其他的组件中的 public 属性进行访问/设置！（译者：这里你需要区分 Property 属性和 Field 字段之前的区别，C#中声明的变量都叫做 Field 字段，使用 get/set 的叫做 Property 属性）

想要绑定某个属性，你只需要在黑板中点击变量后面的一个按钮，然后通过下拉菜单进行选择即可，如下图：



绑定后的效果如下图，点击 UnBind 取消绑定



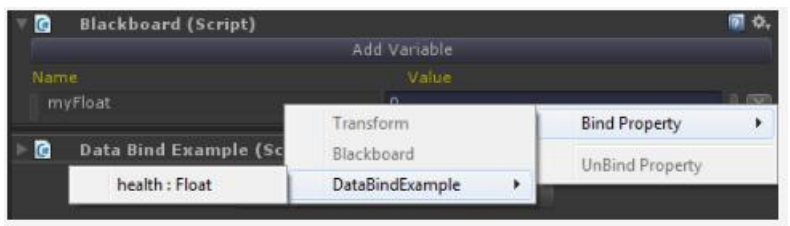
为了测试，我们自己创建一个脚本，挂到当前物体上

```
using UnityEngine;

public class DataBindExample : MonoBehaviour {

    [SerializeField]
    private float _health;

    public float health{
        get {return _health;}
        set {_health = value;}
    }
}
```

字段

属性

把他添加到 Blackboard 所在的 gameobject 上以后，我们就可以绑定这个 health 属性了



3.2 Save & Load （保存/加载）

黑板中的数据可以通过 PlayerPrefs 进行保存和加载。这意味着你可以保存整个行为树的状态。

通过两个简单的 API 我们就可以保存和加载这些数据了！

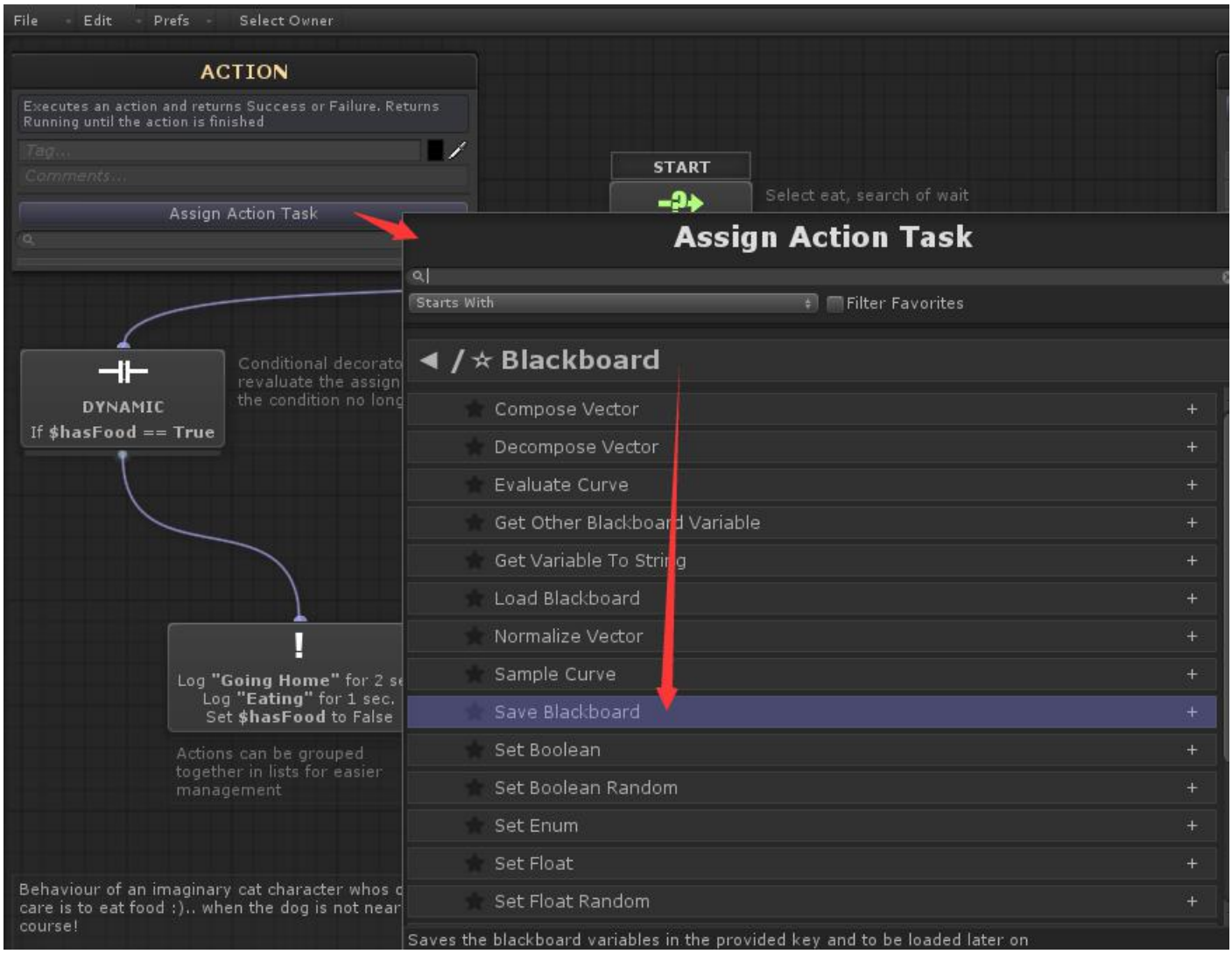
string Svae（string savekey）

使用 savekey 把黑板中的数据保存到哦 PlayerPrefs 中。返回值是序列化后的 json 字符串！

bool Load（string savekey）

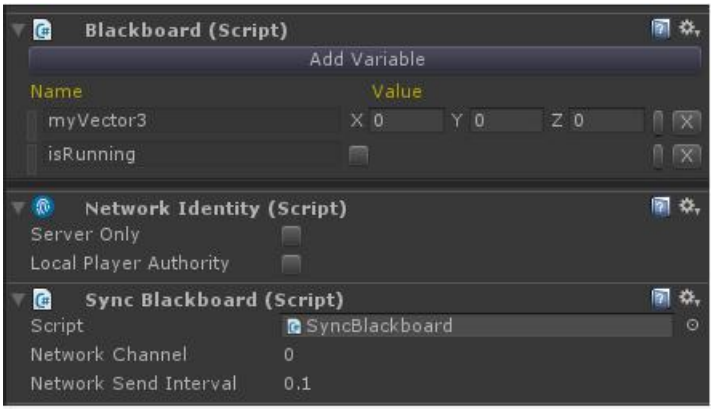
通过 savekey 从 playerPrefs 中加载保存的黑板数据。

为了方便操作，在 Action 的可选 Task 中提供了这两个接口，在 BlackBoard 菜单下！

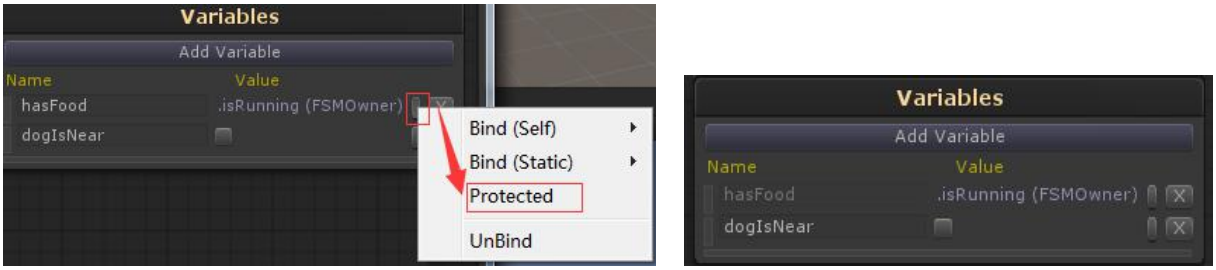


3.3 Network sync

黑板中的这些数据同样可以在 Unity5 中通过 UNet 来进行网络数据同步！你要做的只是在已经包含 Blackboard 的 gameobject 身上在添加一个“SyncBlackboard”脚本即可，如下图：

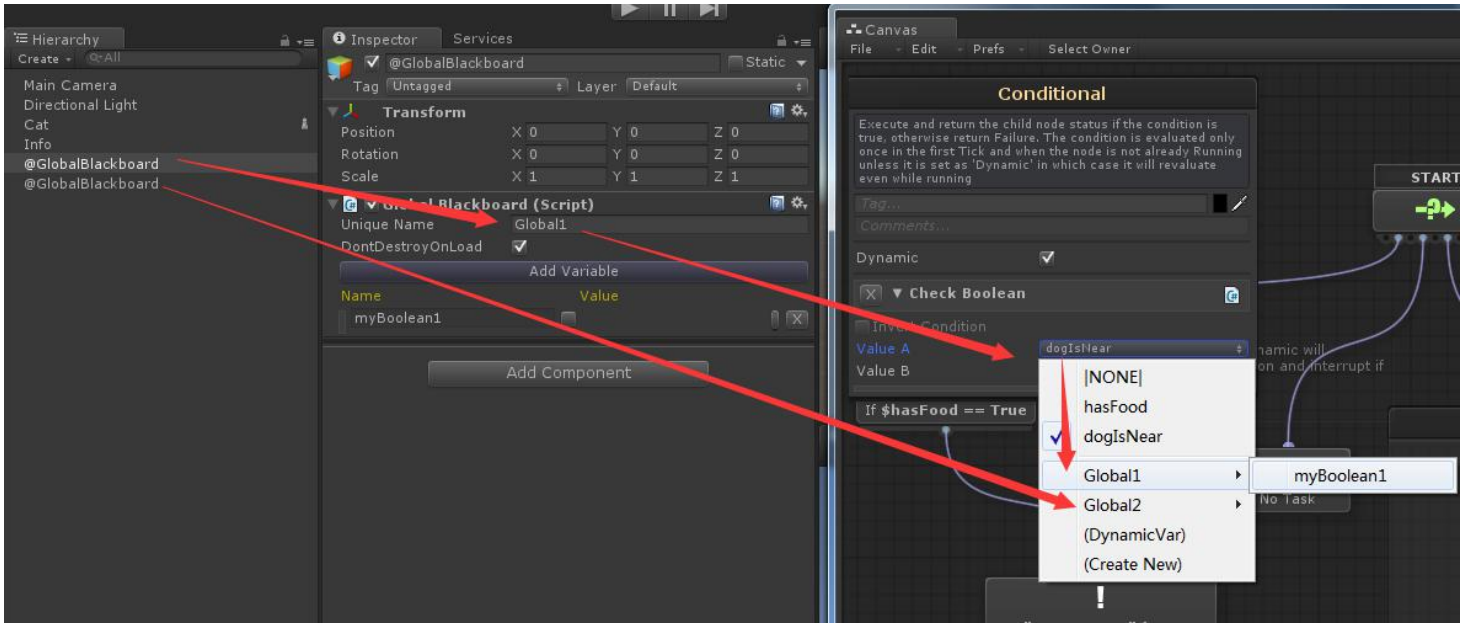


当你添加完毕后，黑板中没有被标记为 protected 的数据以及被 UNET 系统支持的数据都可以在网络上进行同步！不需要额外的操作。如果有些变量你不想被同步，你只需要把它标记成 protected 的就行了！通过变量后面的按钮打开的菜单选项进行选择即可完成标记，标记过后会在属性面板中显示成灰色！如下图：



3.4 Global Variables（全局变量）

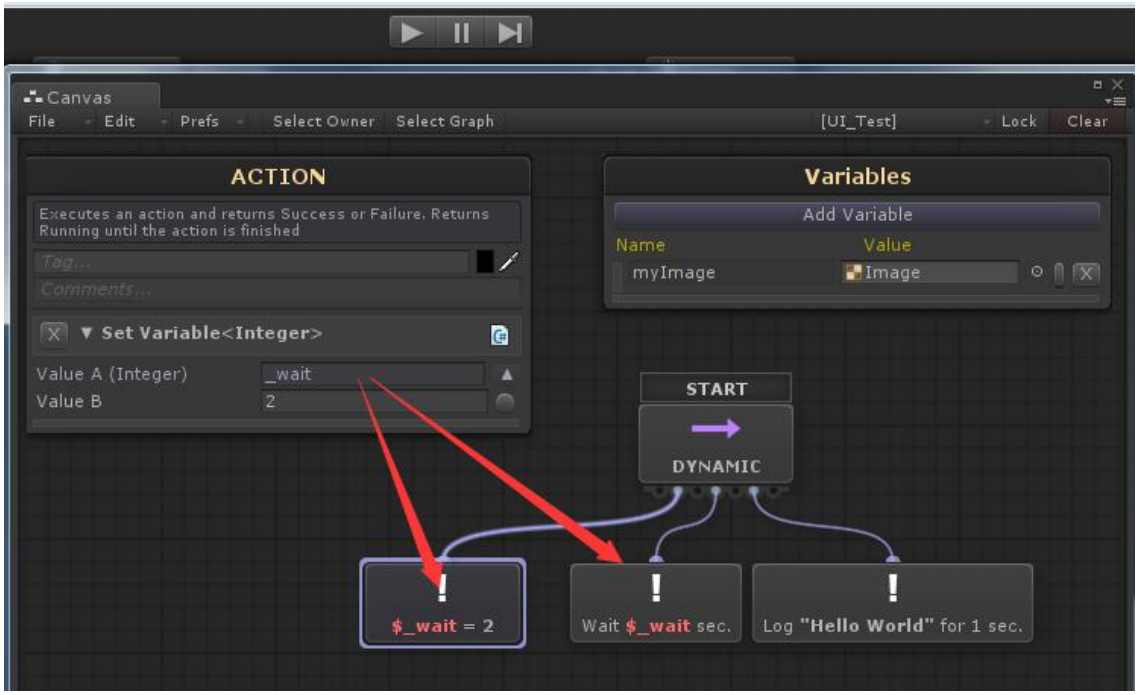
你可以通过菜单栏的“Tools/ParadoxNotion/NodeCanvas/Create/SceneGlobalBlackboard”创建全局的黑板（Global Blackboards）.全局黑板会被添加到当前场景中，这样你可以当前场景中的某个图中引用这个全局变量！你可以拥有多个 GlobalBlackboard，并且这些 GlobalBlackboard 是 DontDestroyOnLoad 的，如下图：

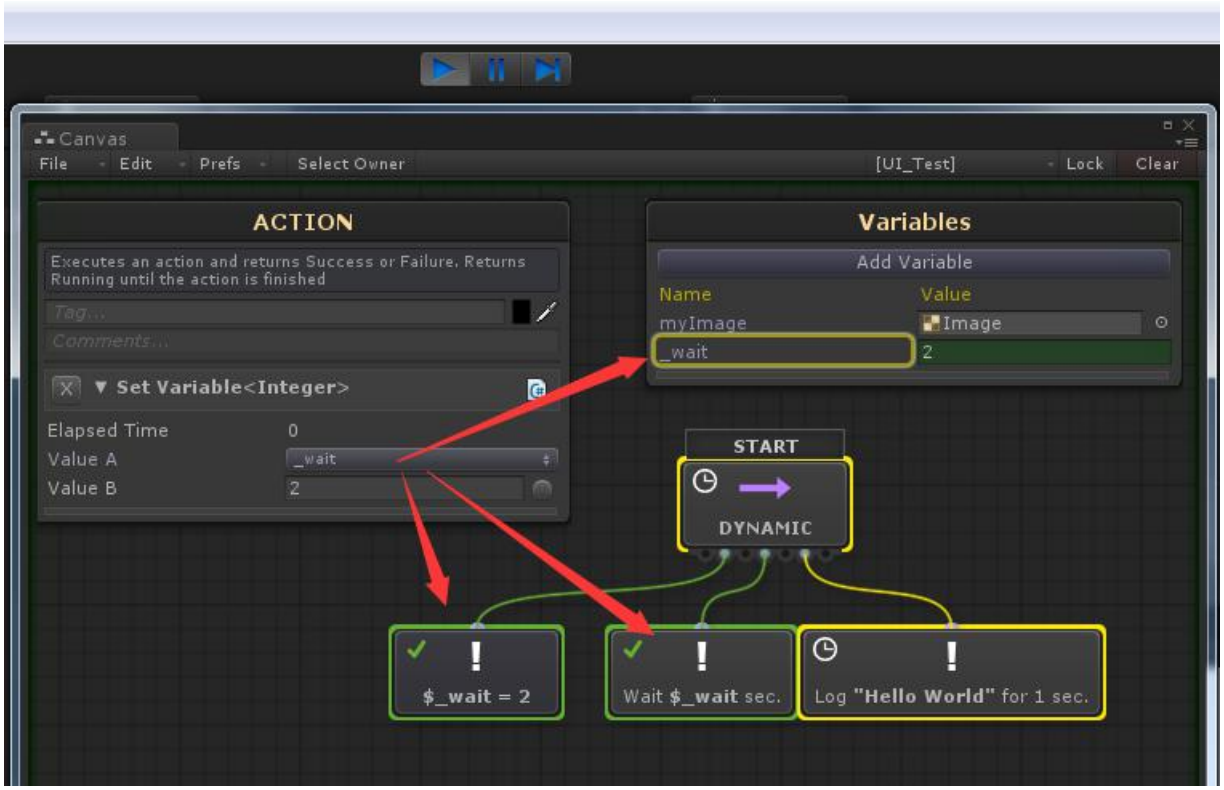


3.5 Dynamic Variables（动态变量，也可以叫运行时变量）

到目前为止你可能已经注意到了，当你在 Task 中操作某些变量并引用黑板中的变量时，你会发现有一个（DynamicVar）的选项，就如上图中显示的那样！这个所谓的动态变量的作用类似于临时变量，只为在两个 Task 之间临时使用一下，这种情况会很多，你不会希望在你的黑板中充斥着各种临时变量，而且它们只在很少时候才会被用到！

例如下图中所展示的这样，先对这个不存在的动态变量_wait 进行赋值 然后取出这个值进行 Wait 等待操作，之后输出 Log，可以看到当前的黑板中是没有_wait 这个变量的，后面这张图是运行时的效果，运行起来以后这个动态变量就被创建了，这个动态变量虽然很好用，但是作为用户来说你需要有自己一套合理的命名规则否则运行时如果发生名称冲突就很难找到问题了。（译者：这个变量在创建后不会被销毁，即便是整个树执行完毕也不销毁）





4：Tasks（Actions & Conditions） 任务（操作和条件判断）

任务功能包括具体的操作和条件判断；也就是 Task 能干什么以及能否这么干！在 Nodecanvas 中，Task 被添加到 Node 节点中。正如之前说的 Node 节点只是一个空壳，具体的作用要看添加进来的 Task 是什么！这样做的好处是保证了足够的灵活性以及低耦合！确保 Task 之间低耦合并且可以组合出不同的 Node 节点，这是 Nodecanvas 的核心思想！

Nodecanvas 默认包含了很多 Task 功能，当然你可以通过一些简单的 API 自定义你自己的 Task。

在 NodeCanvas 中你主要通过两种方式实现自己的功能：

1：创建自己的 Task

2：通过 ScripControl 这个 Task 来调用你在 MonoBehaviors 中的功能函数！

4.1 Creating Custom Tasks （创建自定义 Task）

NodeCanvas 在设计之初就考虑到要让 node 节点只负责管理它下面的工作流，而不用去执行具体的操作！Task 才是真正意义上的执行者，而且 Task 被设计成只有两种类型，一种是 Action（行为），一种是 Condition（条件判断）。这样 Task 就和 Node 脱离了依赖关系，可以在不同的 Node 中使用相同的 Task。这种设计概念避免了在图形化工具中因为图形节点而打乱工作流的问题，也就是具体的操作和图形节点分离！

Action Tasks (行为功能）

简单来说，想要创建一个 Action 的 Task 你必须继承自 ActionTask 这个类并且要覆盖掉一些虚函数。当这个 Action 执行完毕你必须调用 EndAction(bool)，并通过这个 bool 值确定这个 Action 是执行成功，还是执行失败！

下面列出一些主要的 API 用于创建你自己的 Action 类型的 Task：

```
1  //This is the agent for whom the action will take place.
2  public Component agent {get;}
3
4  //This is the blackboard that you can use to read/write variab
5  public IBlackboard blackboard {get;}
6
7  //This is the time in seconds the action is running.
8  public float elapsedTime {get;}
9
10 //Called only the first time the action is executed and before
11 virtual protected string OnInit()
12
13 //Called once when the action is executed.
14 virtual protected void OnExecute()
15
16 //Called every frame while the action is running.
17 virtual protected void OnUpdate()
18
19 //Called when the action stops for any reason.
20 //Either because you called EndAction or cause the action was
21 virtual protected void OnStop()
```

```
22
23 //Called when the action is paused comonly when it was still r
24 virtual protected void OnPause()
25
26 //Send an event to the behaviour graph. Use this along with th
27 protected void SendEvent(string)
28
29 //Similar to above, but sends a value along with the event.
30 protected void SendEvent<T>(string, T)
31
32 //You can use coroutines from within action task as normal.
33 protected Coroutine StartCoroutine(IEnumerator)
34
35 //You must call this to end the action. You can call this from
36 //although typicaly is done in either OnExecute or OnUpdate
37 public void EndAction(bool)
```

示例：

一个简单的 Action 会是这个样子的：

```
using UnityEngine;
using NodeCanvas.Framework;

public class SimpleAction : ActionTask{

    protected override void OnExecute(){
        Debug.Log("My agent is " + agent.name);
        EndAction(true);
    }
}
```

一个简单的“等待”操作，会是这个样子的：

```
using UnityEngine;
using NodeCanvas.Framework;

public class WaitAction : ActionTask {

    public float timeToWait;

    protected override void OnUpdate(){
        if (elapsedTime > timeToWait)
            EndAction(true);
    }
}
```

Condition Tasks （条件判断）

简单来说，想要创建一个 Condition 的 Task 你需要继承自 ConditionTask,并重载 OnoCheck 这个函数并返回 check 的结果！下面列出来一些重要的 API 接口，用于创建一个 Condition 类型的 Task：

```
//This is the agent for whom the action will take place.
public Component agent {get;}

//This is the blackboard that you can use to read/write variab
public IBlackboard blackboard {get;}

//Same as action task.
virtual protected string OnInit()

//This is called when the condition is checked. Override and r
virtual protected bool OnCheck()

//This is a helper method to make the condition return true or
protected void YieldReturn(bool)
```

例子：一个简单的判断事例

```
using UnityEngine;
using NodeCanvas.Framework;

public class EmptyCondition : ConditionTask{

    protected override bool OnCheck(){
        return true;
    }
}
```

4.2 Using Task Attributes （使用 Task 的 Attributes 属性）

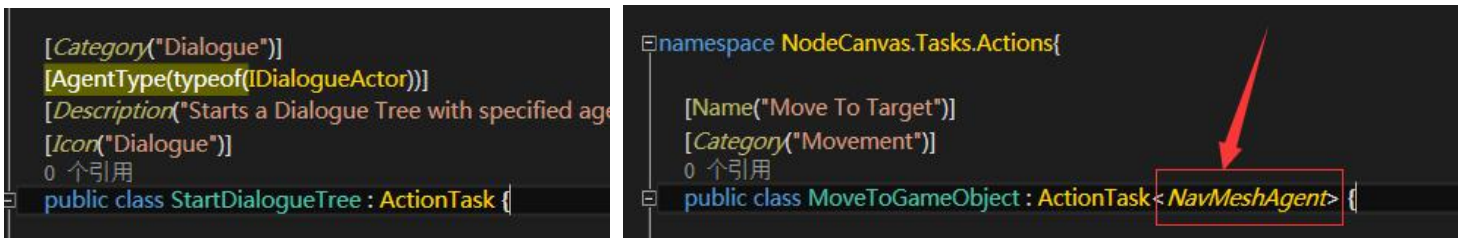
为了使用起来更加方便，NodeCanvas 给大家提供了一些很方便的 Attribute 属性来使用，这些 Attribute 属性既可以在运行时使用也可以在编辑器下使用。下面来一起看一下！

Runtime Attributes 运行时 Attributes 属性

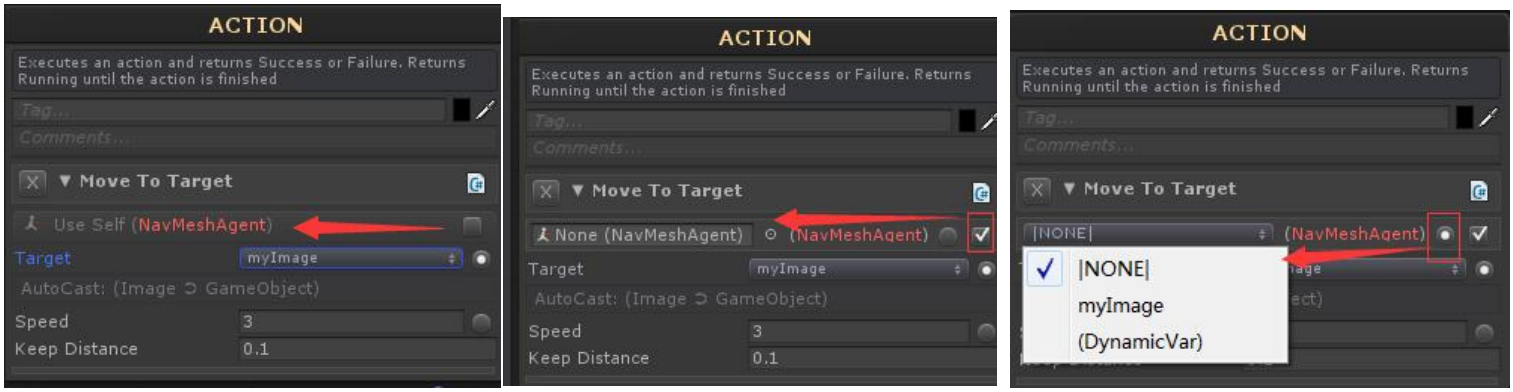
[AgentType(System.Type)]

你可以在 Action 或者 Condition 类型的 Task 上使用这个 Attribute 属性。这将要求 agent（代理）对象所在的 gameobject 上必须有所需要的组件类型。如过 gameobject 身上没有这个类型的组件，则 Task 将在初始化的时候失败！（译者：当前版本中

只发现在 DT 对话树系统中使用了这个 Attribute 属性，其他行为树中不在使用这个属性，代理对象的类型在创建 Action 或者 Condition 的时候已经通过模板进行创建了）如下图：



使用这个属性后，会在属性面板中显示出来，并可以修改当前代理对象（默认是 self 自己的代理，你也可以指定到别的对象上作为代理，如下图）



注意 1：可以给 AgentType 定义一个 interface 类型

注意 2：作为替代这个 Attribute 的操作，你可以在创建 Action 或 Condition 的时候通过模板类型<T>来设置类型，从而不必使用这里的 Attribute！

[Eventreceiver(params stringp[])]

先来看下使用的地方：

```
Assets\ParadoxNotion\NodeCanvas\Tasks\Actions\Animator\MecanimSetIK.cs(10):  [EventReceiver("OnAnimatorIK")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Actions\Animator\MecanimSetLookAt.cs(10):  [EventReceiver("OnAnimatorIK")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckCollision.cs(10): [EventReceiver("OnCollisionEnter", "OnCollisionExit")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckCollision2D.cs(11):  [EventReceiver("OnCollisionEnter2D", "OnCollisionExit2D")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckMouse.cs(10): [EventReceiver("OnMouseEnter", "OnMouseExit", "OnMouseOver")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckMouse2D.cs(11):  [EventReceiver("OnMouseEnter", "OnMouseExit", "OnMouseOver")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckMouseClick.cs(10):  [EventReceiver("OnMouseDown", "OnMouseUp")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckMouseClick2D.cs(11):  [EventReceiver("OnMouseDown", "OnMouseUp")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckTrigger.cs(10):  [EventReceiver("OnTriggerEnter", "OnTriggerExit")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\SystemEvents\CheckTrigger2D.cs(11): [EventReceiver("OnTriggerEnter2D", "OnTriggerExit2D")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\Utility\CheckEvent.cs(12):  [EventReceiver("OnCustomEvent")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\Utility\CheckEvent.cs(37):  [EventReceiver("OnCustomEvent")]
Assets\ParadoxNotion\NodeCanvas\Tasks\Conditions\Utility\CheckEventValue.cs(11): [EventReceiver("OnCustomEvent")]
: 14  已搜索文件总数：363
```

当你想在你的 Task 中获取 Unity 系统时间时，例如 OnTriggerEnter，OnTriggerExit 等等，此时请使用这个 Attribute 属性！添加你想监听的事件，这样你就可以在 Task 中使用它们了！下面列举出了当前可以被使用的事件：

- OnTriggerEnter
- OnTriggerExit
- OnTriggerStay
- OnCollisionEnter
- OnCollisionExit
- OnCollisionStay
- OnTriggerEnter2D
- OnTriggerExit2D
- OnTriggerStay2D
- OnCollisionEnter2D
- OnCollisionExit2D
- OnCollisionStay2D
- OnMouseDown
- OnMouseDrag
- OnMouseEnter
- OnMouseExit
- OnMouseOver
- OnMouseUp
- OnBecameVisible
- OnBecameInvisible
- OnAnimatorIK

注意：Nodecanvas 默认已经包含了 Triggers，Collisions 和 mouse 这些事件的 Condition 类型的 Task！

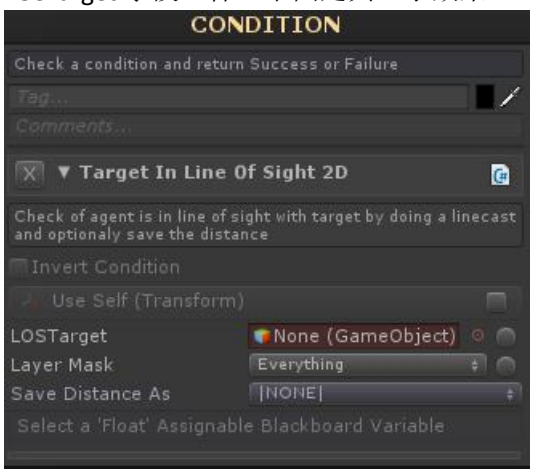
[GetFromAgent]

这个属性用来标记字段的，标记过已有这个字段的值会从当前的代理身上通过 Getcomponent 自动获取。如果没有对应组件则会在初始化时失败，这个属性用来帮助你省区每次都手动赋值的操作！如下图所示，Collider2D 组件每次都自动赋值，并且是 protected 属性的，不在属性面板中显示！

```
public class CheckLOS2D : ConditionTask<Transform> {

    [RequiredField]
    public BBParameter<GameObject> LOSTarget;
    public BBParameter<LayerMask> layerMask = (LayerMask)(-1);
    [BlackboardOnly]
    public BBParameter<float> saveDistanceAs;

    [GetFromAgent]
    protected Collider2D agentCollider;
    private RaycastHit2D[] hits;
```

[RequiredField]

在 nullable 的 public 字段上使用这个属性，在属性面板中会在字段为空的时候显示为醒目的红色提示用户！如同上图中的 LOSTarget 字段一样：下图是其显示效果



Design Attributes (Editor 编辑器下使用的字段)

无论是 Actoion 还是 Condition，它们所有的 public 字段默认情况下都会显示在 Nodecavas 的编辑器属性面板中（Inspector ）！下面列举出一些你可以用来控制这些字段的显示方式的 Attribute 属性！

[SliderField(float min,float max)]

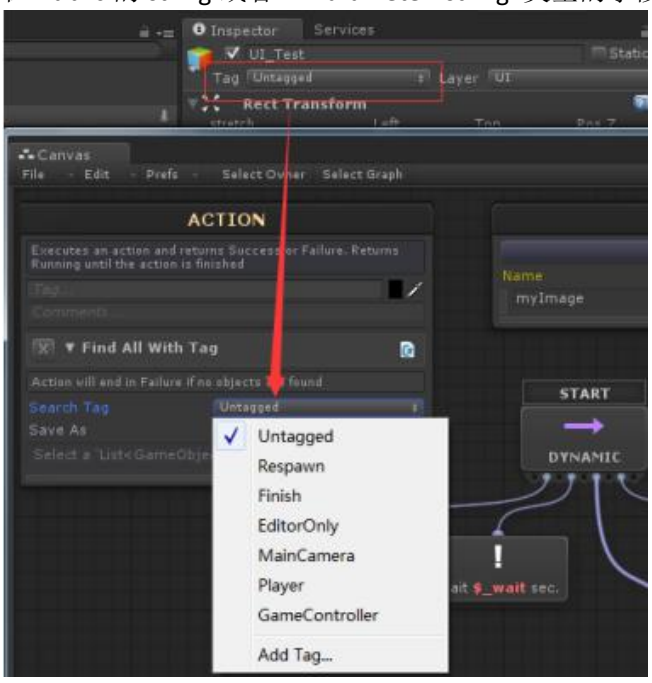
在 public 的 float/int 类型或者 BBParameter<float/int>类型的字段上使用这个属性，将把这些数值类型显示为滑动条！

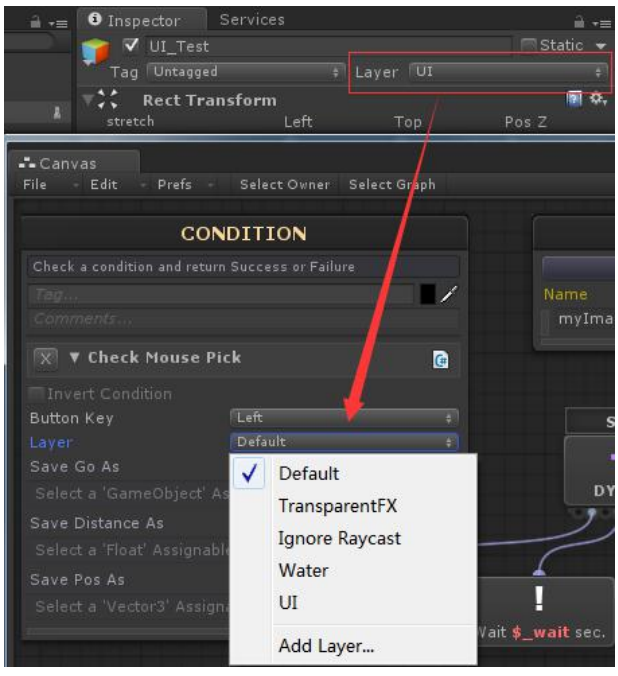
[TextAreaField(float height)]

在 public 的 string 或者 BBParameter<string>类型的字段上使用这个属性，将会把它们显示为一个很大的文本区域

[TagField]

在 Public 的 string 或者 BBParameter<string>类型的字段上使用这个属性，将会把它显示为一个 Tag 下拉菜单！如下图 1





[LayerField]

在 Public 的 int 或者 BBParameter<int>类型的字段上使用这个属性，将会把它显示为一个 Layer 下拉菜单！如上图 2

注意 1：从父类继承的 public 字段也会显示在 inspection（属性面板中）

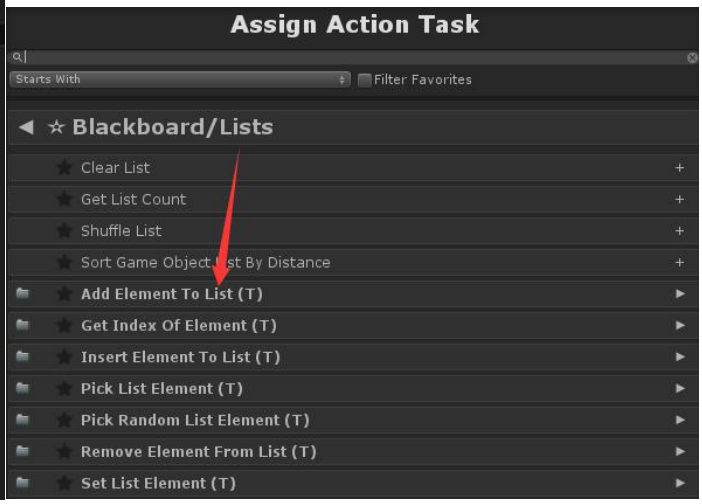
注意 2：如果你需要其他的特许显示样式或者效果，你可以 override 虚函数 virtual protected void OnTaskInspectorGUI(),记得把它添加到#if UNITY_EDITOR 下就行了！

Meta-information 元信息相关属性

[Category(string)]

用来标记当前 Task 所要显示的类型路径，你可以使用“/”来划分层级

```
namespace NodeCanvas.Tasks.Actions{

    [Category("☆ Blackboard/Lists")]
    0 个引用
    public class AddElementToList<T> : ActionTask{
        [RequiredField] [BlackboardOnly]
        public BBParameter<List<T>> targetList;
        public BBParameter<T> targetElement;

        99+ references
        protected override string info{
            get {return string.Format("Add {0} In {1}", targetElement, targetList);}
        }

        99+ references
        protected override void OnExecute(){
            targetList.value.Add(targetElement.value);
            EndAction();
        }
    }
}
```



[Name(string)]

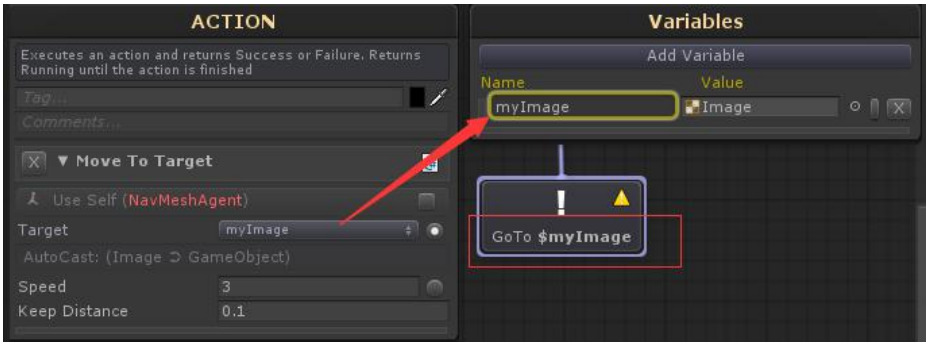
当你的类名太长或者不适合显示的时候，你可以用这个属性显示你想要显示的名称！

```
[Name("Conditional")]
[Category("Decorators")]
[Description("Execute and return the child node status if the condition is true, otherwis
[Icon("Accessor")]
0 个引用
public class ConditionalEvaluator : BTDecorator, ITaskAssignable<ConditionTask> {
```

（译者：还有其他一些属性，例如：[Description(string)] , [Icon(string)] 等等，请自行查看，官方帮助文档中并未提及！）

默认情况下显示在 Node 节点中的 Task 信息你可以通过覆盖虚函数“virtual string info()”来进行重写！例如下图：

```
[Name("Move To Target")]
[Category("Movement")]
0 个引用
public class MoveToGameObject : ActionTask<NavMeshAgent> {

    [RequiredField]
    public BBParameter<GameObject> target;
    public BBParameter<float> speed = 3;
    public float keepDistance = 0.1f;

    private Vector3? lastRequest;

    99+ references
    protected override string info{
        get {return "GoTo " + target.ToString();}
    }
```

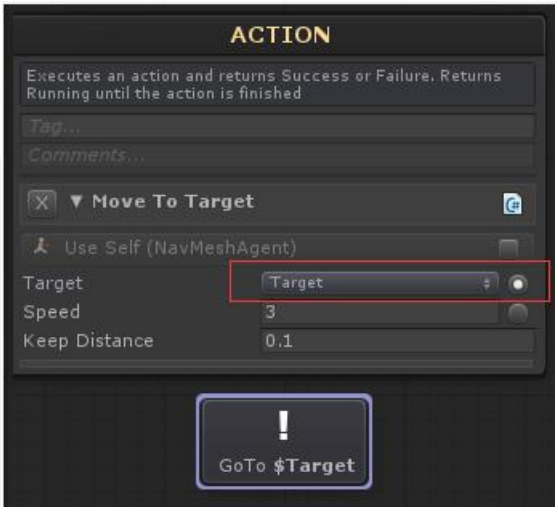


如果你没有重载这个属性，那么默认将显示 Task 的 Name

4.3 Using BBParameters in Tasks Task 中的 BB 参数

BB 参数能够让你很方便的和黑板中的数据进行交互。简单来说，你需要使用 BBParameter<T>来声明你的字段。下面是个简单的例子：

```csharp
using UnityEngine;
using NodeCanvas.Framework;

public class WaitAction : ActionTask {

    public BBParameter<float> timeToWait;

    protected override void OnUpdate(){
        if (elapsedTime > timeToWait.value)
            EndAction(true);
    }
}
```

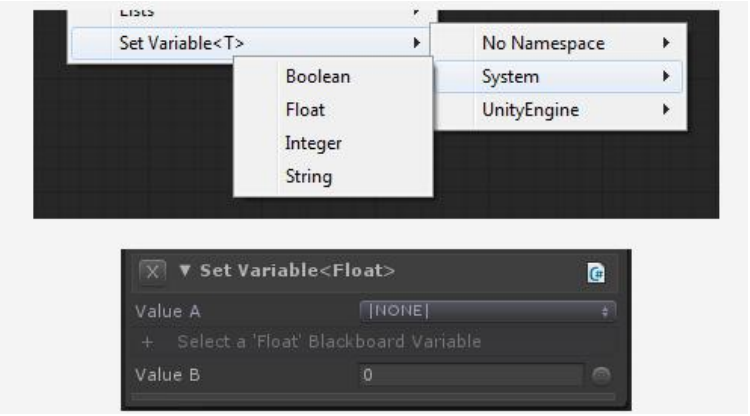


记住，你可以直接在继承自 Task 类的子类中直接使用.blackboard 进行对黑板的读写操作！

4.4 Creating Generic Tasks 创建一个通用的 Task

在 NodeCanvas 中你提供了各种泛型模板让你创建各种 Action 或 Condition，而不用担心因为类型的不同而重复编写操作相同的 Task！这也是 BBParameter 使用泛型模板的原因！下面就是使用泛型模板的例子，这个类已经包含在 Nodecanvas 中了：

```csharp
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.Tasks.Actions{

    [Category("☆ Blackboard")]
    public class SetVariable<T> : ActionTask {

        [BlackboardOnly] [RequiredField]
        public BBParameter<T> valueA;
        public BBParameter<T> valueB;

        protected override string info{
            get {return valueA + " = " + valueB;}
        }

        protected override void OnExecute(){
            valueA.value = valueB.value;
            EndAction();
        }
    }
}
```



接下来你就可以在编辑器中选中某个 Action 节点然后添加这个 Action，并通过泛型模板<T>来选择你所需要的类型。如上图 2

最后这里再看一个 AddElementToList<T>的 Action 类型的 Task

```csharp
using System.Collections.Generic;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.Tasks.Actions{

    [Category("☆ Blackboard/Lists")]
    public class AddElementToList<T> : ActionTask{

        [RequiredField] [BlackboardOnly]
        public BBParameter<List<T>> targetList;
        public BBParameter<T> targetElement;

        protected override void OnExecute(){
            targetList.value.Add(targetElement.value);
            EndAction();
        }
    }
}
```

4.5: The Script Control Tasks 脚本控制器 Task

NodeCanvas 中提供了很多有用的 Task 来帮助用户们直接使用现有的代码或者组件。

通过这些 Task 可以干很多事情，下面就列举一些，保证让你眼前一亮：

* Translate an object （对一个 object 进行类型转换操作）
* Change an object's name（修改一个 object 的名称）
* Check if a CharacterController isGrounded (检查某个 CharacterController 角色控制器是否站在地面上）
* Check if a RigidBody isKinematic（检查某个 RigidBody 缸体是否是 isKinematic 运动的）
* Send a Message to a game object (向某个 gameobject 发送一个消息)
* Stop an AudioSource（停止一个音频的播放）
* Set an AudioSource's volume (设置一个音频的音量)
*and so on.... 等等。。。

此外，你可以把你的逻辑直接写在自己的脚本文件中而不需要把它们写成 Action 或 Condition，然后再 Nodecanvas 中通过这些 ScriptControl 的 Task 去访问你的脚本（获取数据/调用方法）。下面列举出这些 ScriptControl 的 Task。

Actions

*** Execute Function （执行某个方法）**


```
//do it by calling delegate or invoking method
99+ references
protected override void OnExecute(){

    for (var i = 0; i < parameters.Count; i++){
        args[i] = parameters[i].value;
    }

    if (targetMethod.ReturnType == typeof(IEnumerator)){
        StartCoroutine( InternalCoroutine( (IEnumerator)targetMethod.Invoke(agent, args) ));
        return;
    }

    returnValue.value = targetMethod.Invoke(agent, args);
    EndAction();
}
```

这个 Action 允许你执行获取某个 Animator 中的某个 Float 变量，通过一个 Name 参数获取以及把返回值保存到某个黑板变量中！这里的代理是 Self，因为当前的对象身上没有 Animator 组件所以这里显示为红色！
默认情况下，执行完方法后直接返回 true，如果函数返回值是一个 IEnumerator 的话，那么将会创建一个 Coroutine 携程并执行，直到携程结束才会返回 true. 如上图！EndAction 定义如下

```
///Ends the action either in success or failure. Ending with null means that it's a cancel/interrupt.
///Null is used by the external system. You should use true or false when calling EndAction within it.
79 个引用
public void EndAction(){ EndAction(true); }
92 个引用
public void EndAction(bool? success){

    latch = success != null? true : false;

    if (status != Status.Running)
        return;

    isPaused = false;
    status = success == true? Status.Success : Status.Failure;
    OnStop();
}
```

*** Implemented Action （应用一个 Action）**
使用这个 Implemented Action 你同样是可以调用一个在其他脚本中的方法，但是这个方法的返回值是有状态，

上面的 ExecuteFunction 返回的状态永远是 True，而这里的返回值是 Statues 状态，不在是某个具体的数据类型。例如下面这样：

```
using NodeCanvas.Framework;

public class Example : MonoBehaviour {

    public Status SomeAction(string text){

        Debug.Log(text);
        return Status.Success;
    }
}
```

*** Get Property（获取属性中的数据） Set Porperty（设置属性中数据）**

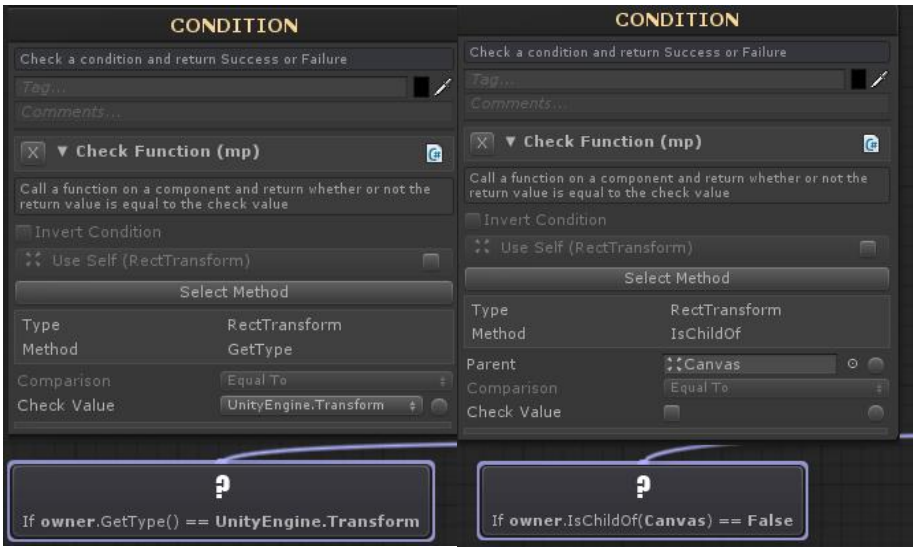
这两个个 Action 允许你从别的脚本中获取某/设置某个属性的值 ， 并保存到黑板中或是从黑板中读取数据！

*** Get Field（获取字段中的数据） Set Field（设置字段中数据）**

获取/设置 字段中的数据。

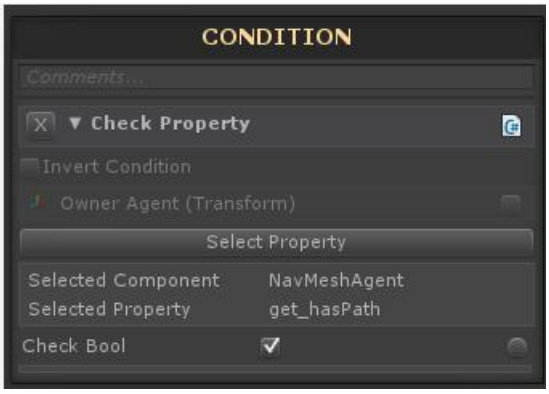
Conditions

*** Check Function （方法返回值进行比较检查）**



这个 Condition 允许你获取某个类中的带有返回值的方法，并把返回值和你指定的值进行比较！

*** Check Property（属性值进行比较检查）**



这个 Condition 允许你对属性值进行获取并和自己指定的值进行比较判断！

*** Check Csharp Event（对 C#事件进行比较检查）**

这个 Condition 用来监听 EventHandler 类型或者是用户自定义类型（0 个参数，void 返回值）的事件！当事件被触发的时候这个 Condition 会返回 true！

4.6 Using FixedUpdate and OnGUI

默认情况下所有的 Action 类型的 Task 都是在 Update 下进行更新的，但是有些时候你可能希望使用 fixedUpdate 或者是 OnGUI。从而避免掉频繁的 Update，虽然没有可以覆盖 Update 接口的虚函数，但是你可以通过 MonoManager 类中的 AddFixedUpdateMethod/AddOnGUIMethod 等相关的函数 在 onFixedUpdate 或 onGUI 回调事件中进行 处理，事件定义如下图：

```
//These are used internaly instead of the events above for performance reasons.
private List<Action> onUpdateCalls     = new List<Action>();
private List<Action> onLateUpdateCalls  = new List<Action>();
private List<Action> onFixedUpdateCalls = new List<Action>();
private List<Action> onGUICalls        = new List<Action>();
```

```
1 个引用
public static void AddUpdateMethod(Action method) { current.onUpdateCalls.Add(method); }
2 个引用
public static void RemoveUpdateMethod(Action method) { current.onUpdateCalls.Remove(method); i--; }

0 个引用
public static void AddLateUpdateMethod(Action method) { current.onLateUpdateCalls.Add(method); }
0 个引用
public static void RemoveLateUpdateMethod(Action method) { current.onLateUpdateCalls.Remove(method); i--; }

0 个引用
public static void AddFixedUpdateMethod(Action method) { current.onFixedUpdateCalls.Add(method); }
0 个引用
public static void RemoveFixedUpdateMethod(Action method) { current.onFixedUpdateCalls.Remove(method); i--; }

1 个引用
public static void AddOnGUIMethod(Action method) { current.onGUICalls.Add(method); }
1 个引用
public static void RemoveOnGUIMethod(Action method) { current.onGUICalls.Remove(method); i--; }
```

（译者：游戏中不建议大家使用 OnGUI 这个接口，因为每次调用这个更新都会产生 400B 的内存开销，建议大家屏蔽(public void OnGUI(){...}这个接口)

5：Behaviour Trees （行为树）

Overview 预览

BT 行为树现在应用非常广泛，最常见的是被用在 AI 人工智能当中！行为树可以被当做一个简化的编程语言，你可以通过一些简单的节点，例如判断，循环，执行队列，选择等等。来组织你的逻辑！Nodecanvas 可以让你方便的去编辑行为树！
行为树在每一次 Update（Nocecanvas 中叫做 Tick）的时候都会从第一个 Node 节点开始执行。之后这个节点会执行其内部的 Task，并返回一个 Status 状态（成功/失败），或者是运行中状态！节点的功能是由内部具体的 Task 决定的，但是节点本身可以大致分为以下几个类型：

Leaf Nodes （叶子节点）
叶子节点没有子节点，当一次 Update（Tick）执行到这个叶子节点的时候，就表示这个分支执行结束了！一般的叶子节点的作用要么是一个 Condition 作为条件判断，要么就是个 Action 执行某些操作，或者是改变某些游戏状态等！

Composite Nodes（复合节点）
复合节点的作用则是用于叶子节点的执行顺序。例如：复合节点下有很多个叶子节点，那么这个复合节点就可以有选择性的执行下面的叶子节点（顺序执行，随机执行）。最基本的复合节点就是 Sequencer 顺序执行节点，它的功能就是顺序执行下面所有的叶子节点，直到遇到一个返回错误的叶子节点才会停止！而 selector 复合节点的作用是执行下面所哟叶子节点，直到遇到一个返回正确的叶子节点才会停止！就像逻辑预算符“And”和“OR”.

Decorator Nodes (修饰节点或者叫控制器节点更形象)
每个修饰节点的作用都有所不同，但是整体来说它们就是以某种方式来改变它们的子节点的功能或者说是如何去执行它们的子节点！常见的 Decorator 装饰节点包括：子节点循环控制，子节点访问限制，打断子节点当前的执行顺序等等！Nodecanvas 提供了很多有用的适配器给大家使用！

Further Reading （延伸阅读）
如果你想更深入的了解 Behavior trees ，请查看一下链接：
http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php
http://aigamedev.com/open/article/behavior-trees-part1/
http://gamedev.stackexchange.com/questions/51693/decision-tree-vs-behavior-tree
http://www.cse.scu.edu/~tschwarz/COEN196_13/PPT/Decision%20Making.pdf

5.1 BT Nodes Reference （BT 节点介绍）

Leafs （叶子节点：包括 Action 行为节点和 Condition 条件节点）

叶子节点没有子节点，它是一个分支的末尾最后的节点！
Action （行为节点）

Action 节点将会执行分配到其中的所有 Action 类型的 Task。只有 Action 节点中的所有 ActionTask 执行完毕并返回 Success 或者 Failure ，否则这个 Action 节点将会一直返回 Running！ActionTask 包含的状态如下：

* Success：当添加的 ActionTask 完成时返回 Success
* Failure： 当添加的 ActionTask 失败时返回 Failure
* Running：当添加的 ActionTask 正在运行时返回 Running

Condition （条件节点）

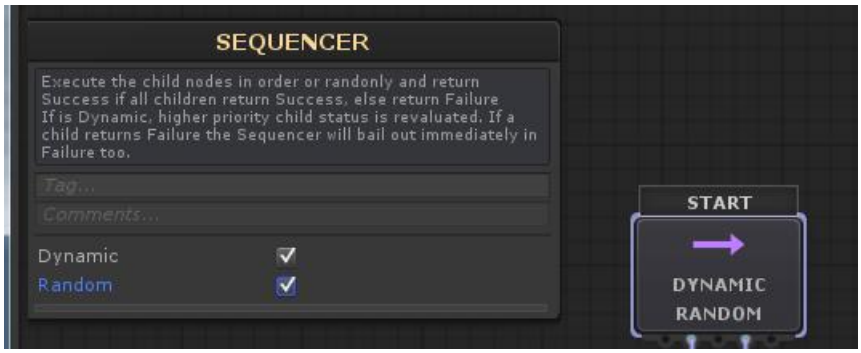
条件节点用来处理各种条件判断的 ConditionTask，并返回 success 或者 Failure！ConditionTask 包含的状态如下：

* Success ：如果 ConditionTask 返回 true
* Failure：如果 ConditionTask 返回 false
* Running：这个状态不可用于 ConditionTask

Composites (复合节点)

常用的复合节点有如下几个：

1：Sequencer（顺序节点）



顺序节点将从高到低（从左到右）执行下面的所有节点，当所有子节点返回 success 的时候返回 Success，如果有任何一个子节点返回 Failure 则它返回 Failure。

Dynamic Sequencer 会实时监测优先级比较高的子节点的状态变化（优先级从左往右）。所以如果优先级比较高的子节点被执行过了，但是它的状态发生变化比如返回 Failure，那么这个 Sequencer 会打断当前正在运行的子节点单并返回 Failure！这个功能对于希望实时响应某些优先级比较高的子节点状态变化的应用场景很有帮助！（译者：读起来比较拗口）！

Random Sequencer (随机序列)会在每次执行到这个序列的时候都会打乱所有子节点的执行顺序，进行随机的执行！

Sequencer 包含三个返回状态：

*** Success：** 当所有子节点都 Success

*** Failure：** 当任意一个子节点 Failure

*** Running：** 当正在处理某个子节点的时候返回 Running

2：Selector（选择节点）

Selector 选择节点将从左往右的执行它下面的子节点，只要任意一个子节点返回 Success 则直接返回 Success 不在执行后续子节点，如果 Selector 的所有子节点都返回 Failure，那么 Selector 才会返回 Failure！

Dynamic Selector 会实时监测高优先级子节点的状态变化（优先级从左往右），如果某个已经执行过的高优先级的子节点状态发生了变化，那么这个 Selector 会打断当前正在执行的操作！如果这个高优先级的子节点状态变为 Success，那么 Selector 就会直接返回 success 不在继续执行后续子节点！这个 Dynamic 的作用跟 Sequencer 的一样！

Random Selector 会随机执行 Selector 节点下的所有子节点。

Selector 包含三个返回状态：

*** Success：**当任意一个子节点返回 Success

*** Failure：** 当所有子节点返回 Failure

*** Running：**当正在处理某个子节点的时候返回 Running

3：Probability Selector（概率选择节点）

Probability Selector 概率选择节点会基于一定的百分比的命中率去执行下面的某个子节点！如果选中的节点返回 Success，那么 Probability Selector 概率选择节点就返回 Success，如果选中的的子节点返回 Failure,那么它会重新选择一个新的子节点进行执行。如果没有其他的子节点了或者是所有的子节点都返回 Failure 那么它将返回 Failure！如果随机选择过程中选到了“Direct Failure Chance”就像上图中那样，那么也会返回 Failure。

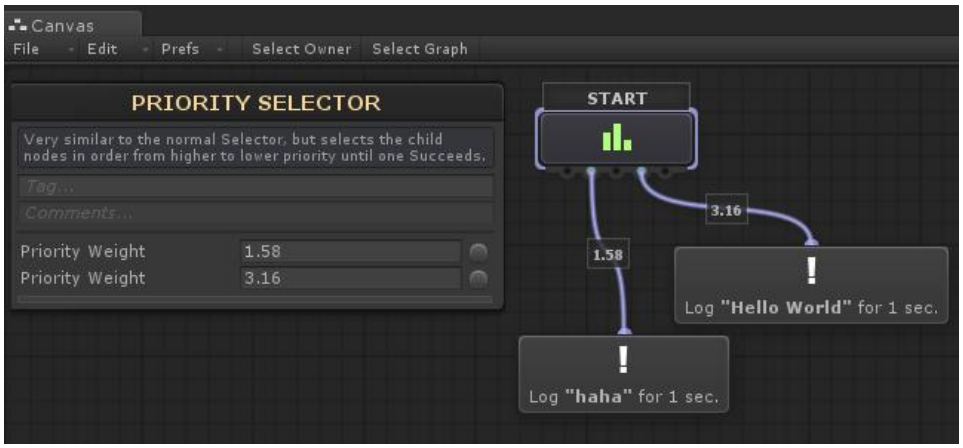
*** Success：**当选中的子节点返回 Success 时

*** Failure：** 当选中的子节点返回 Failure 或者选中了“Direct Failure Chance”的时候

*** Running：**当选中的子节点正在 Running 的时候

注意：Probability 属性面板中的的随机数值可以使用 Blackboard 黑板中的变量，这样你可以在运行时动态修改这些概率！

4：Priority Selector（优先选择节点）



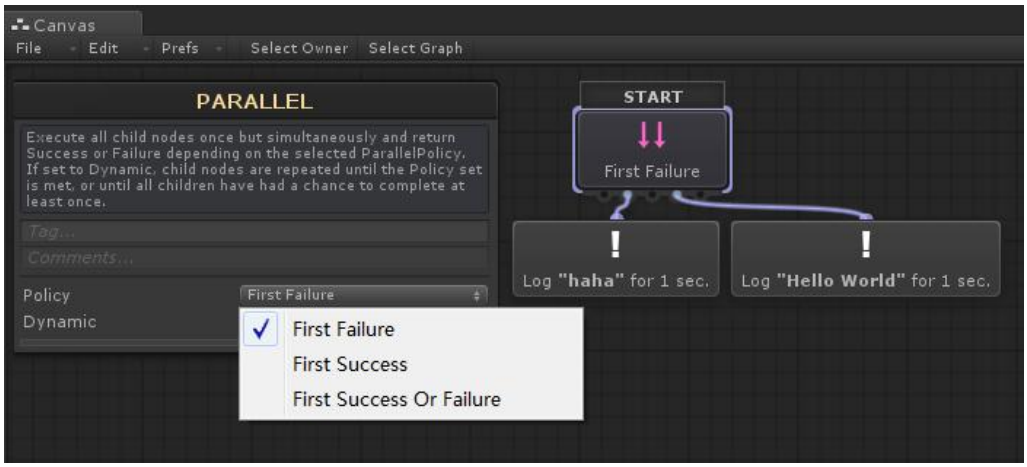
优先选择节点，如上图这样，谁的 Weight 权重高就先执行谁！上图中右边的 Log 操作权重比左边的高所以会先执行右边的！这些权重数值同样可以绑定到黑板中的变量，然后在运行时进行修改！

*** Success：**当选中的子节点返回 Success 的时候

*** Failure：** 当所有子节点都返回 Failure 的时候

*** Running：**当选中的子节点正在 Running 的时候

5：Parallel（平行节点）



平行节点，如上图所示。平行节点会同时执行下面的所有子节点，至于返回值则由 Policy 参数控制。

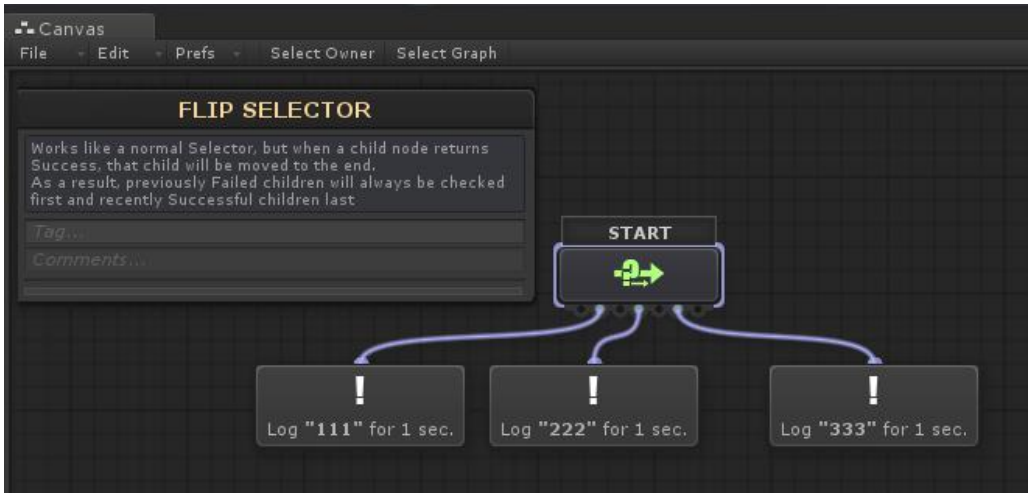
*** Failure：**如果 Policy 设置成 Firs Failure，那么只要有一个节点返回 Failure 此时会先 reset 重置所有正在运行的子节点然后返回 Failure。只有当所有子节点都返回 Success 的时候 Parallel 平行节点才会返回 Success！

*** Success：**如果 Policy 设置成 First Success，那么只要有一个节点返回 Success，此时会先 reset 重置所有正在运行的子节点然后返回 Success。只有当所有子节点都返回 Failure 的时候 Parallel 平行节点才会返回 Failure！

*** Success Or Failure** 如果 Policy 设置成 First Success Or Failure，那么只要有任意一个节点返回，Parallel 都会使用这个返回值返回，无论是 Success 还是 Failure！

*** Running：** 当选中的子节点正在 Running 的时候

6：Flip Selector（翻转选择节点）

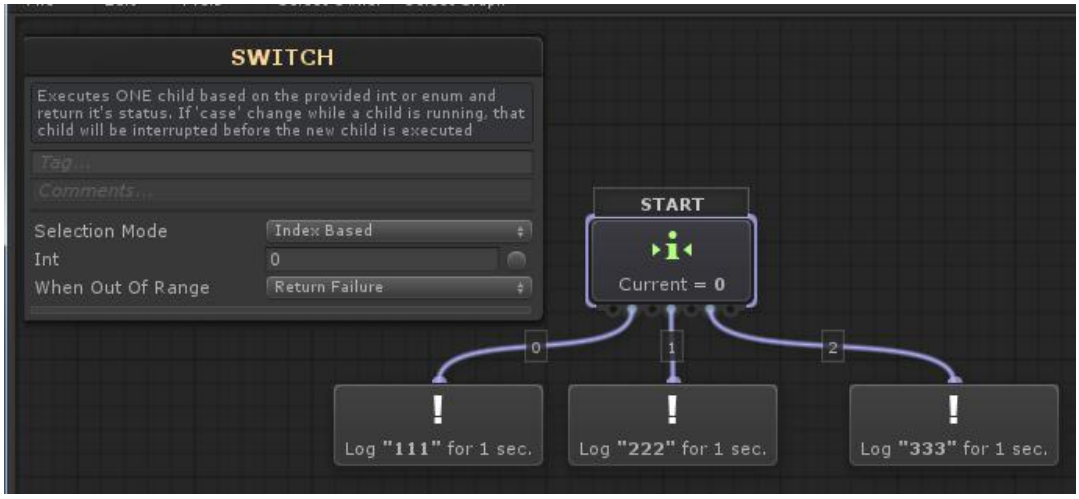


FlipSelector 工作原理类似于 Selector，不同的是当一个子节点返回 Success 的时候，这个子节点会被移动到子节点顺序的结尾（最右边）。这样能保证下次执行这个 FlipSelector 的子节点的时候，子节点中返回 Failure 的节点总是最先被检查！

*** Success：** 当选中的子节点返回 Success 的时候
*** Failure：** 当所有子节点都返回 Failure 的时候
*** Running：** 当选中的子节点正在 Running 的时候

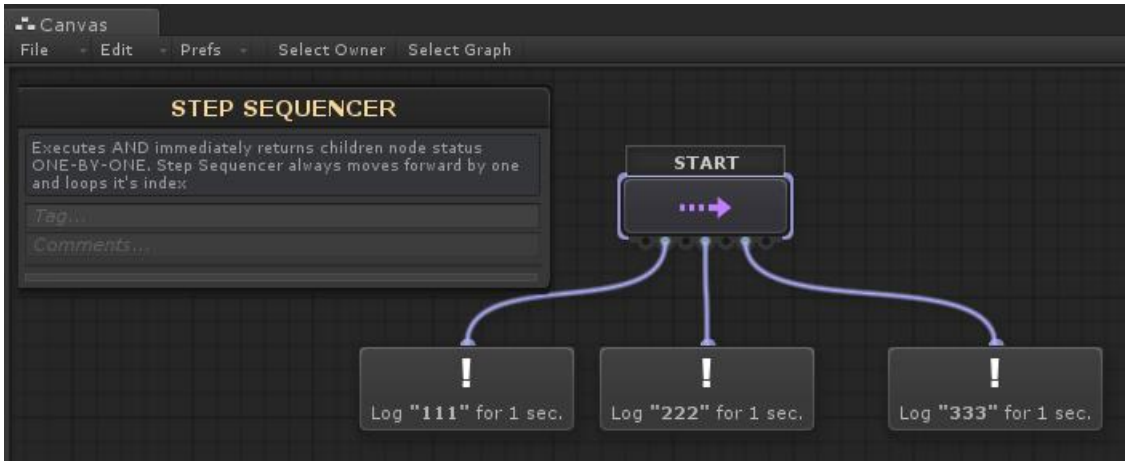
7：Switch（选择节点）

Switch 节点既能选择整型值，也可以选择枚举值。通过 Selection Mode 修改数据类型！它会通过当前值来确定需要执行的子节点究竟是哪个，这个值你可以绑定到黑板进行动态修改！如果某个节点正在执行时你修改了执行索引值，那么 Switch 会暂停当前正在执行的子节点并重新执行新的索引值对应的子节点！如下图所示：



*** Success：** 当选中的子节点返回 Success 的时候，或者索引值超出了范围并且 When Out Of Range 设置成 Success 的时候
*** Failure：** 当选中的子节点返回 Failure 的时候，或者索引值超出了范围并且 When Out Of Range 设置成 Failure 的时候
*** Running：** 当选中的子节点正在 Running 的时候

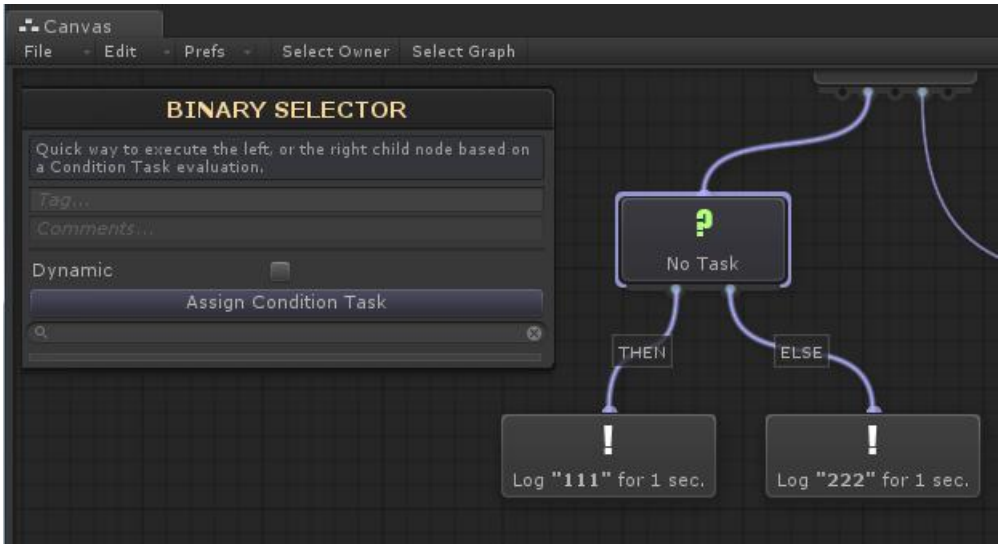
8：Step Sequencer（顺序选择节点）



顺序执行节点会一个接着一个的执行下面的子节点，每次 Update(Tick)只执行一个节点,再次进入此节点才会执行下一个子节点！执行到最后一个后会从头继续执行！

*** Success：** 当选中的子节点返回 Success 的时候
*** Failure：** 当所有子节点都返回 Failure 的时候
*** Running：** 当选中的子节点正在 Running 的时候

9：Binary Selector（更准确的来讲应该叫 bool 选择器）

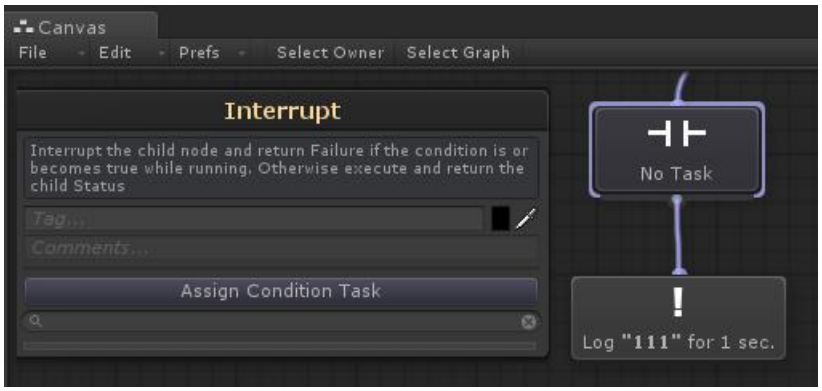


看图就明白了，添加一个 ConditionTask 然后根据比较结果选择执行哪个节点，返回值就是选择的节点的返回值！

Decarators(修饰节点)

修饰节点的子节点只能有一个。修饰节点给子节点提供了额外的功能，过滤或者修改！

1：Interrupt（打断节点）



Interrupt 打断节点需要一个 Condition Task。如果这个 Condition 判断节点为 true 并且子节点在执行状态的话，那么会打断子节点的运行并返回 Failure。否则 interrupt 将返回子节点的执行结果！

*** Success：** 当子节点返回 Success 的时候

*** Failure：** 当子节点返回 Failure 的时候，或者 condition 判断条件为 true 的时候（即使是在运行状态）

*** Running：** 当子节点正在 Running 的时候

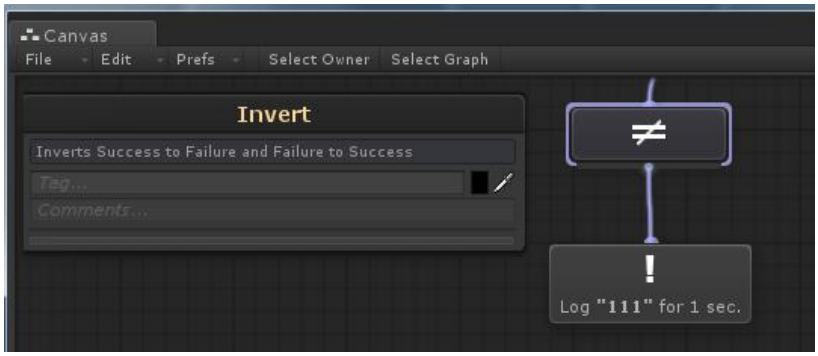
注意：interrupt 节点可以被添加任何类型的 Condition Task！

2：Conditional（条件节点）

只有在 Conditional 节点中的 Condition 为 true 的时候才会执行子节点，然后返回子节点的返回值（Success 或者是 Failure）！只有当子节点不在运行的时候并且 condition 是 false 的时候返回 Failure.换句话说就是如果 Conditional 节点判断为 true 正在执行子节点，此时 conditional 判断即便变为 false 也不会打断正在执行的子节点！只在执行子节点前判断，执行中不在受其影响！

*** Success：** 当子节点返回 Success 的时候
*** Failure：** 当子节点返回 Failure 的时候，或者 condition 判断条件为 false 的时候（并且子节点不在运行状态）
*** Running：** 当子节点正在 Running 的时候

3：Inverter（取反节点）

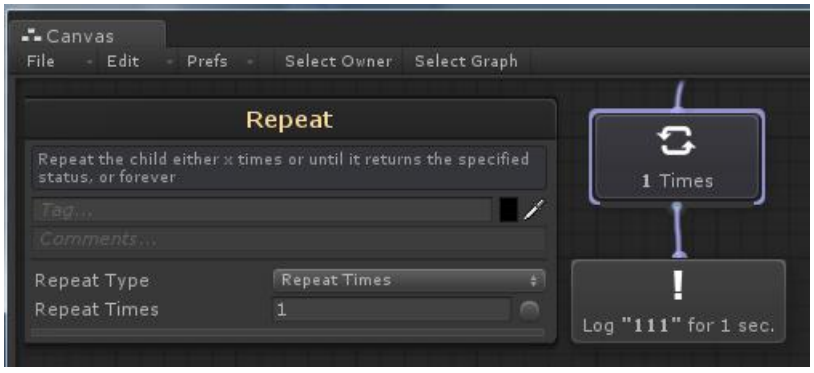


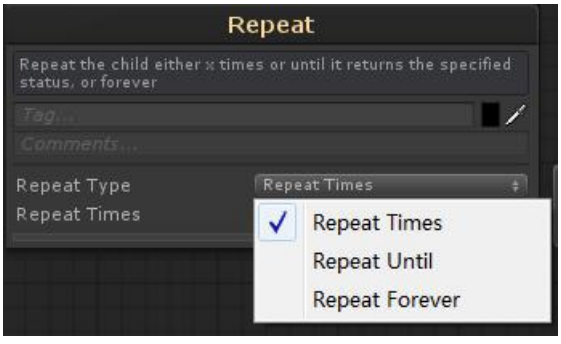
如果子节点返回 Success，Inverter 则返回 Failure，如果子节点返回 failure,Inverter 则返回 Success！总是相反结果！
*** Success：** 当子节点返回 Failure 的时候
*** Failure：** 当子节点返回 Success 的时候
*** Running：** 当子节点正在 Running 的时候

4：Remap（重映射节点）

对子节点返回值进行重映射，具体因设置通过属性面板自行设置。类似于 Inverter 节点！

5：Repeater（重复执行节点）





更具 Repeat 类型 来确定 Repeat 数值！
*** Success：** 1：如果设置为“Repeat Times”，当子节点返回 Success 时。
2：如果设置为“Repeat Until”，当子节点返回 Success，并且 Repeat Until 设置为“Success”
*** Failure：** 1：如果设置为“Repeat Times”，当子节点返回 Failure 时。
2：如果设置为“Repeat Until”，当子节点返回 Failure，并且 Repeat Until 设置为“Failure”
*** Running：** 只要设置为重复并满足重复条件，则一直保持运行状态！
注意：Repeat Time 可以绑定到黑板数据，从而动态修改 Repeat Time！

6：Filter（过滤节点）

如上图所示有两个模式选择：1：限制次数模式，执行时判断次数限制，2：倒计时模式，时间到就激活一次子节点

Inactive when Limited 用来控制 Filter 节点初始化时是 Active 还是 Inactive 的。

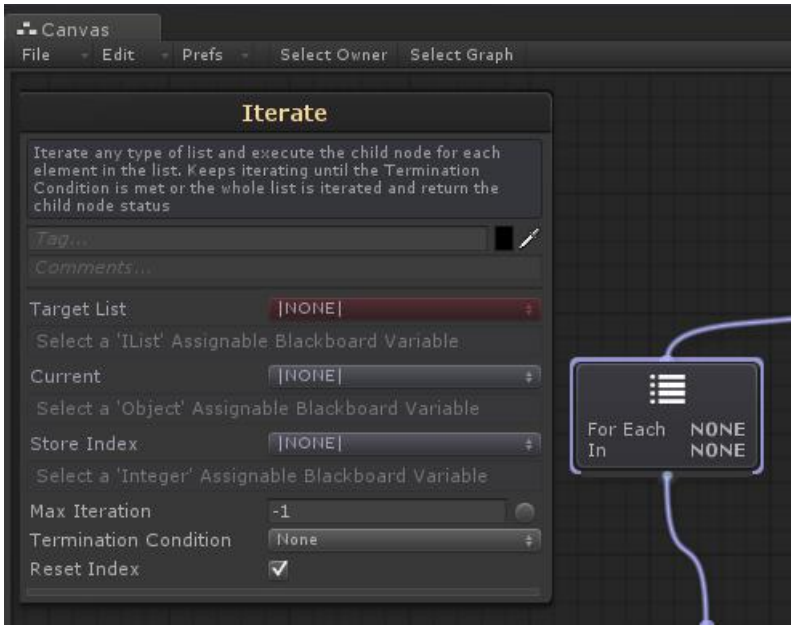
*** Success：**当子节点被访问过了并且返回的是 Success

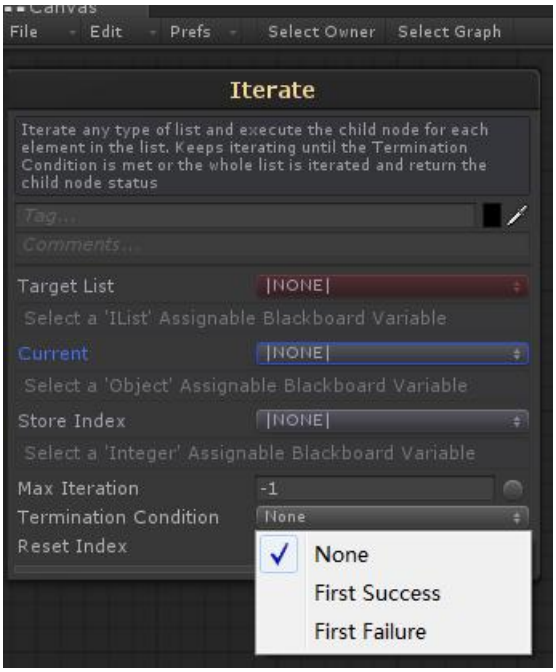
*** Failure：** 当子节点被访问过并且返回 Failure，或者关闭了“Inactive When Limited”

*** Running：**当子节点被访问过了，并返回 Running 的时候

注意：变量值可以绑定黑板中的数据，以便动态修改！

7：Iterator（迭代器节点）





Iterator 迭代器会遍历一个 Blackboard 的 list。并把当前值和索引值保存到黑板变量中！ The Iterator can optionaly be set to terminate iteration as soon as the decorated node returns either Success or Failure. 如果没有设置“Termination Condition(终止判断)”或者是 List 遍历完了但是并没触发 Termination conditions(终止判断)，那么 Iterator 迭代器节点将返回最后一个迭代的子节点执行的结果！

如果开启了 ResetIndex，那么当这个 Iterator 节点 Reset 重置的时候，迭代器的索引会被重置回 0。

如果没开启 ResetIndex，那么会保存当前索引位置,再次执行到这个迭代器节点的时候会使用上次的索引位置！如果遍历到 List 末尾那么每次重新执行这个迭代器节点都会使用最后的这一个迭代器索引位置！ 把它看成是“for each”吧！

*** Success：**1：当 List 被遍历完了并且所有子节点都返回 Success.

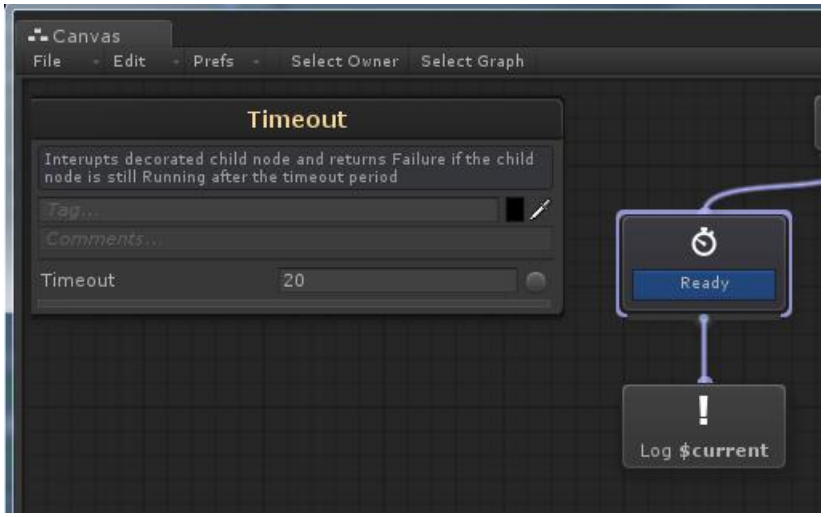
2：当设置为“First Success”的时候，遍历时只要遇到一个返回 Success 的子节点就返回！

*** Failure：** 1：当 List 遍历完了并且所有节点返回 Failure.

2：当设置为“First Failure”的时候，便利士只要遇到一个返回 Failure 的子节点就返回！

*** Running：**处于遍历中的时候一直是 Running 状态！

8：Timeout（超时节点）



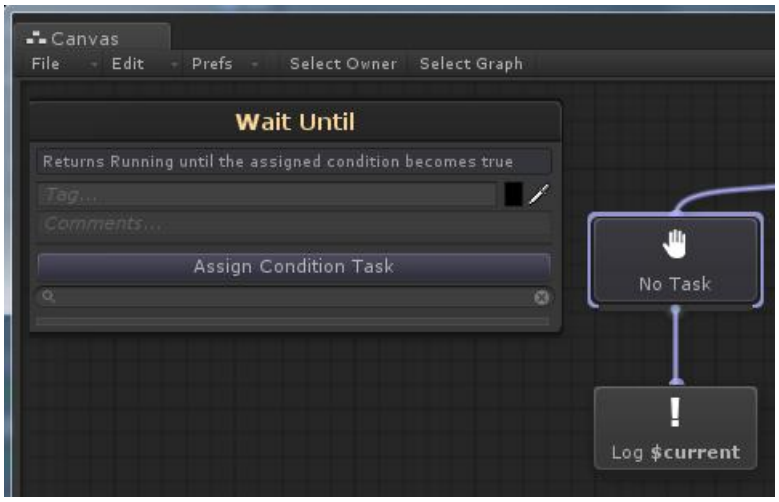
顾名思义，计时在 Timeout 时间内一直执行子节点，直到时间结束

*** Success：** 当子节点返回 Success 的时候

*** Failure：** 当子节点返回 Failure 的时候，或者倒计时结束时

*** Running：** 当子节点正在 Running 的时候

9：Wait Until（等待节点）



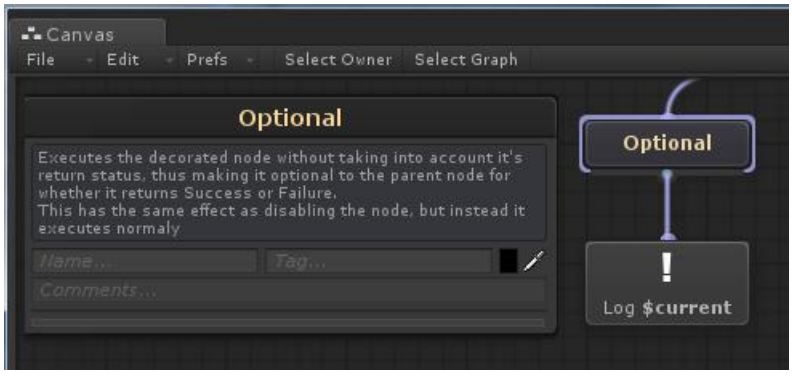
添加一个 Condition Task 就可以进行判断操作了，直到判断条件返回 true 才会执行子节点

*** Success：** 当子节点返回 Success 的时候

*** Failure：** 当子节点返回 Failure 的时候

*** Running：** 当子节点正在 Running 的时候或者 Condition 是 false 的

10：Optional（配置节点）



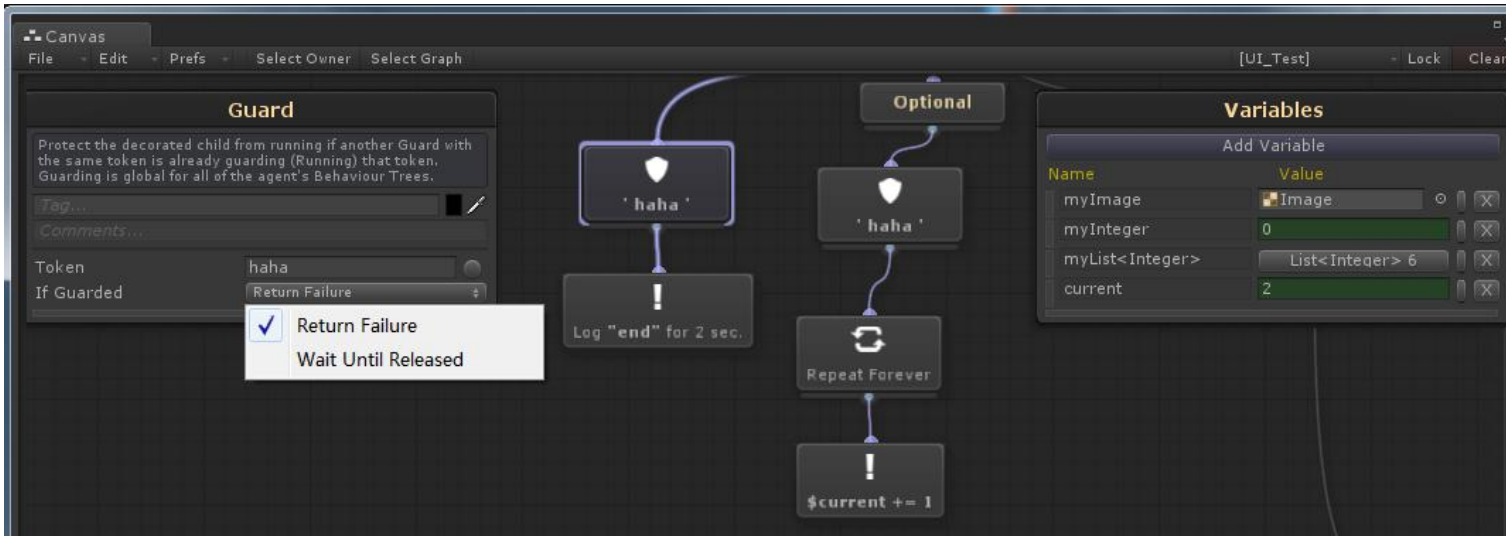
既不返回 Success 也不返回 Failure，只是执行了一次子节点而已，而不必在意子节的返回状态！

*** Success：** 从不

*** Failure：** 从不

*** Running：** 当子节点 Running 时

11：Guard（守卫节点）



守护节点主要是用来起到互斥作用的，避免同一时间对某个数据或者属性进行操作！当一个 Token 在运行的时候，其他所有相同 Token 的 Guard 要么返回 Failure，要么 Wait Until Release，Guard 的作用域是整个行为树！

*** Success：当子节点返回 Success**

*** Failure： 当子节点返回 Failure，或者保护功能被激活并且设置为保护时返回 Failure。**

*** Running：当子节点运行时或者保护功能被激活并且设置为 Wait Until Released**

12：Override Agent（覆盖 Agent 节点）

覆盖 Agent 代理将会使用另外一个 Agent 代理替换子节点中当前的 Agent。这样你可以通过绑定黑板变量来动态修改这个代理对象！

*** Success：**当子节点返回 Success 的时候

*** Failure：** 当子节点返回 Failure 的时候

*** Running：**当子节点正在 Running 的时候或

Sub - Behaviours (子行为树)

行为树是可以嵌套使用的，被嵌套的字树叫做 Sub-Behavior，这样可以提供方便的模块化处理操作！

- SubTree

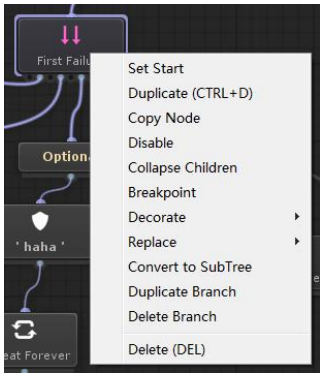
你可以创建一个新的树或者使用现有的树 Asset 资源！双击节点进入子树，子树和主树共享使用同一个黑板！

*** Success：** 当子节点返回 Success 的时候

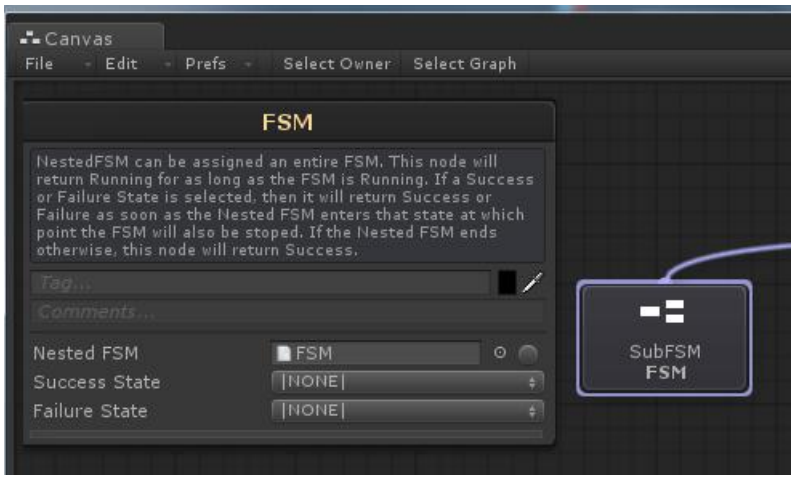
*** Failure：** 当子节点返回 Failure 的时候

*** Running：** 当子节点正在 Running 的时候或

注意：你可以通过右键点击任意一个 composite 复合节点选择 conver to subTree 来转换成子行为树！



- NestedFsm 嵌套的 FSM



嵌套 FSM 用来在行为树 BT 中嵌套 FSM 状态机。你可以定义一个变量作为 Success 状态返回，另一个作为 Failure 状态返回！只要在 FSM 切换到这个两个状态即可返回响应状态，另外 FSM 和 BT 之间共享 Blackboard！

*** Success：** 当嵌套 FSM 节点进入 Success 状态的时候或者 FSM 完成（finished）

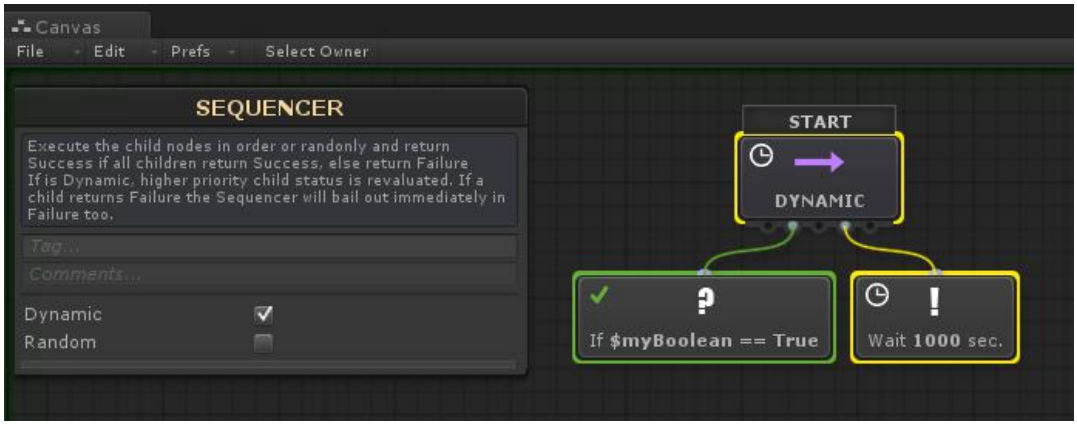
*** Failure：** 当嵌套 FSM 节点返回 Failure 的时候

*** Running：** 当嵌套 FSM 节点正在 Running 的时候或

5.2 Reactive Evaluation （节点的响应策略）

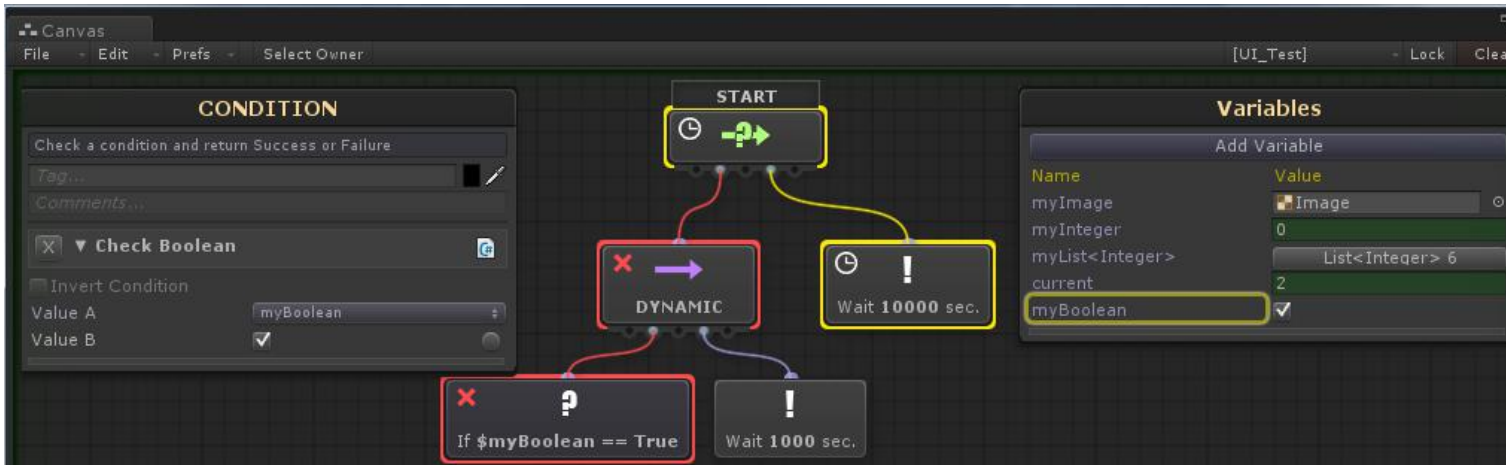
Behavior Tree 在 NodeCanvas 中被设计成能够立刻响应的，这是为了区别于以往的行为树（轮询操作，无法及时响应）也是当前主流游戏所需要的功能！

有很多行为树的节点提供了一个叫做“Dynamic”的设置项。开启这个设置后行为树节点能够立即对子节点的状态变化做出反应，而无需等到下一个更新操作（Update 或 Tick）。下面我们来看个例子：

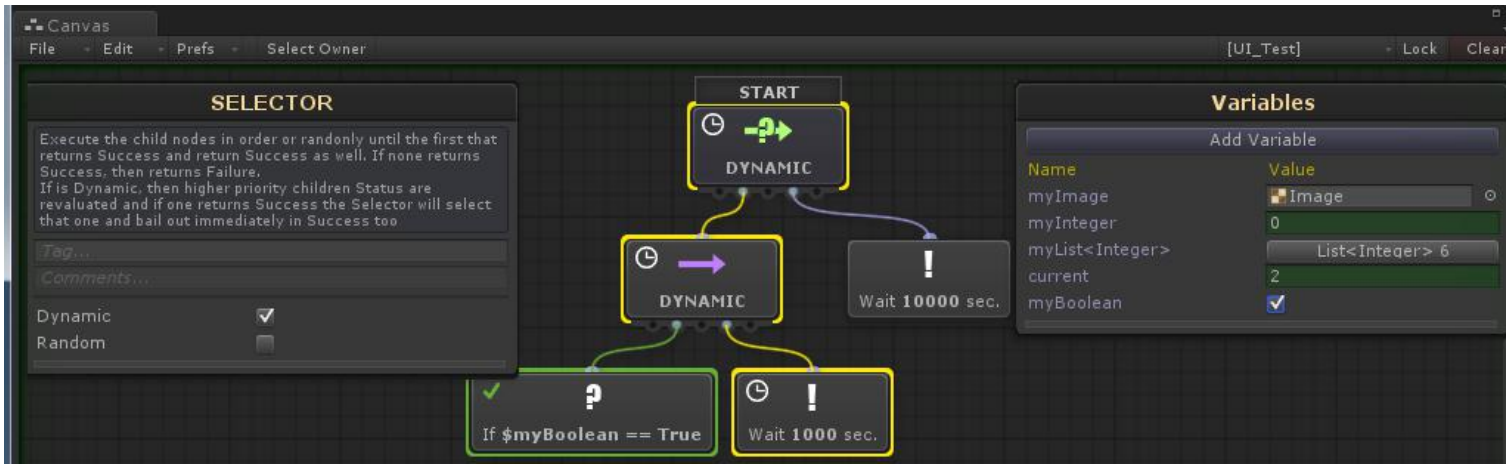


图中假设我们有一棵行为树，其中“myBoolen”变量值为 true,接着我们等待 1000 秒（夸张一点）。如果判断条件为 false 的话则不会执行后面的 wait 操作！如果是 true 的话则会等待后面的 1000 秒。此时如果把 Sequencer 设置成 Dynamic 的，如上图那样，那么当在执行 Wait 的时候此时 myboolean 变成了 false，那么会立刻中断 wait 的操作并返回 Failure。这就是 Dynamic 的作用，如果 Sequencer 不是 Dynamic 的，当执行到 wait 的时候，无论 myboolean 是否为 true 都不会打断 wait 操作了！

Recursive Reactive Evaluation（递归子节点的响应策略）



例如上图这样，当执行到 Selector 节点后面的 wait10000 的时候，此时即便前一个 Sequencer 是 Dynamic 的，无论如何修改 myboolean 的值都无法打断这个 10000 的 wait 操作。此时你需要把这个 Selector 也设置成 Dynamic 的才能够打断并重新执行前一个 wait1000 的操作！如下图



如果你正在制作一款即时的战略游戏或者动作游戏，那么你一定需要这个即时响应的 Dynamic 功能！

Dynamic 选项可以在一下 BT 行为树的节点中被找到：

*** Sequencer**

*** Selector**

*** Parallel**

*** conditional Decorator**

5.3 Creating Custom BT Nodes （创建自定义 BT 节点）

为了能够在 NodeCanvas 中更加合理的实现自己的游戏逻辑，你可以定制自己的 BT 节点，包括 Leafs 叶子节点，Composites 复合节点，Decorators 修饰节点。需要注意的是，你需要继承自它们各自的基类，并覆盖一些必要的函数！

下面是一些重要的属性：

Status status

Node 节点当前的状态，你还可以用来存储临时状态以便后续查看和使用！

Component graphAgent

BT 当前的 Agent 代理对象

IBlackboard graphBlackboard

BT 当前使用的黑板

List<Connection> outConnections

在 Nodecanvas 中，这个连接对象指的是连接节点自己。你可以通过它们的链接来访问子节点。这个对象提供一个方法去获取子节点的执行状态！

下面列出一些必要的函数，用于自定义实现：

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

    [Category("My Nodes")]
    [Icon("SomeIcon")]
    [Description("Some description")]
    public class NodeName : BTNode {

        //When the BT starts
        public override void OnGraphStarted(){

        }

        //When the BT stops
        public override void OnGraphStoped(){

        }

        //When the BT pauses
        public override void OnGraphPaused(){

        }

        //When the node is Ticked
        protected override Status OnExecute(Component agent, IBlackboa

            return Status.Success;
        }

        //When the node resets (start of graph, interrupted, new tree
        protected override void OnReset(){

        }

        #if UNITY_EDITOR

        //This GUI is shown IN the node IF you want
        protected override void OnNodeGUI(){

        }

        //This GUI is shown when the node is selected IF you want
        protected override void OnNodeInspectorGUI(){

            DrawDefaultInspector(); //this is done when you dont overr
        }

        #endif
    }
}
```

Creating a Leaf node（创建叶子节点）

BT 的叶子节点不再包含任何子节点。要创建一个自定义的叶子节点，需要继承自 BTNode，下面是一个简单的 Action 节点的演示：

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

    [Category("My Nodes")]
    [Icon("SomeIcon")]
    [Description("Wait for an ammount of seconds then return Success")]
    public class SimpleDelay : BTNode {

        public BBParameter<float> waitTime;
        private float timer;

        protected override Status OnExecute(Component agent, IBlackboa

            timer += Time.deltaTime;
            if (timer > waitTime.value)
                return Status.Success;

            return Status.Running;
        }

        protected override void OnReset(){

            timer = 0;
        }
    }
}
```

Nodecanvas 推荐大家不要直接创建叶子节点，而是去创建 Action 或 Condition 的 Task，节点就直接使用默认提供的 Action 节点或 Condition 节点就好了！（译者：注意区分 Node 节点和 Task 是不同的，Node 是容器，Task 是具体任务）

Creating a Decorator node（创建修饰节点）

创建修饰节点，你需要继承自 BTDecrator 基类。Decorator 修饰节点只能有一个子节点。你可以通过 connection object 来访问子节点。下面是一个 Inverter 取反修饰节点的示例：

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

    [Category("Decorators")]
    [Icon("SomeIcon")]
    [Description("Invert Success to Failure and Failure to Success")]
    public class Inverter : BTDecorator{

        protected override Status OnExecute(Component agent, IBlackboa

            status = decoratedConnection.Execute();

            if (status == Status.Success)
                return Status.Failure;

            if (status == Status.Failure)
                return Status.Success;

            return status;
        }
    }
}
```

你可以通过 connection object 访问所修饰的子节点。在 Nodecanvas 中 connections 总是只想它们自身并且他们提供了接口去访问它们的执行状态！

Creating a Composite node（创建复合节点）

去创建一个复合节点，你需要继承自 BTComposite 基类。复合节点可以包含任意多的子节点。和 Decorator 修饰节点一样，你也可以通过 connections 访问子节点或者执行子节点！

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

    [Category("Composites")]
    [Icon("Sequencer")]
    [Description("Execute the child nodes in order from left to right"
    public class SimpleSequencer : BTComposite {

        private int lastRunningNodeIndex;

        protected override Status OnExecute(Component agent, IBlackboa

            for (int i = lastRunningNodeIndex; i < outConnections.Coun

                status = outConnections<em class="d4pbbc-italic"></em>

                if (status == Status.Running){
                    lastRunningNodeIndex = i;
                    return Status.Running;
                }

                if (status == Status.Failure)
                    return Status.Failure;
            }

            return Status.Success;
        }

        protected override void OnReset(){

            lastRunningNodeIndex = 0;
        }
    }
}
```

就是这些，记住双击某个 Node 节点，通过 IDE 打开他们的代码！

6. State Machines（状态机 FSM）

States 什么是状态

FSM 节点当中大部分都称之为 State。一个 State 在 Entered 进入之后会一直保持 Running 运行状态，知道 Exited 退出！State 之间可以互相连接。当 state 的判断条件满足的时候可以跳转到另一个 state。当触发跳转后当天的 state 会执行 Exit 操作，跳转到的 state 会执行 Enter 操作。当 state 的内部任务完成之后他会检查所有的跳转操作，如果存在某个跳转操作则直接执行跳转，这个跳转操作不受 Condition 判断的影响！

Transitions（跳转条件）

除了 Concurrent state（并发状态），其他所有 FSM 中的 node 节点都拥有 Transitions（跳转条件）。一个 Transition 绑定一个 Condition task（条件判断任务）。每当某个 State 被激活的时候，这个 state 下面的所有 Transitions 都会被检查！如果有判断是 true 的那么就会触发这个 Transition 并且当前 state 会执行 Exit 退出，目标 state 会执行 Enter 并切换成为整个状态机 FSM 的当前正在执行的 State。一个没有 condition 判断的 Transition 跳转操作，当其父 State 完成（finished）的时候会被执行！注意当你拥有多个这种没有 Condition 的 Transition 的时候，只有第一个 Transition 会被执行！

你可以通过属性面板设置多个 Transition 之间的优先级，一次来确定它们的优先顺序！

Nodecanvas 中的 FSM 系统是基于 Condition 条件判断的，不是基于 Event 事件的！ 如果你想使用 Event 那么可以使用 Condition 去封装一个 event 的检查器！

Further Reading（扩展阅读）

https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

6.1 FSM Node reference (FSM 节点参考)

Action State（节点）

Action State 节点中包含了一个 Action 的 List，当 Enter 的时候，这个 List 中的 Action 会被执行，执行的顺序可以是顺序的也可以是并行的！可以通过 inspector 面板进行设置！所有 Action 执行完毕后 State 会切换到 Finished 状态。运行的时候正在被执行的 Action 会被添加一个播放图标！

*** OnEnter：** 所有 list 中的 Action 会被执行(OnExecute)

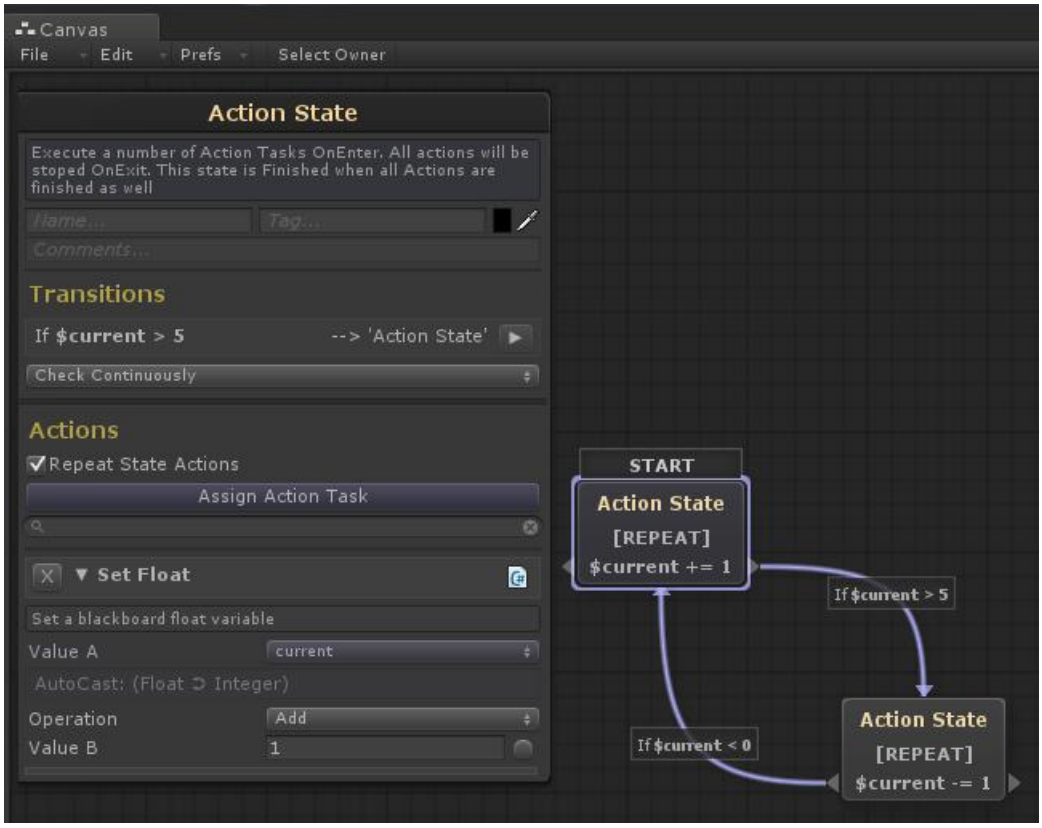
*** OnExit：** 所有 list 中的 Action 会被暂停(OnStop)

*** Finished:** 当所有 List 中的 Action 都执行完毕(EndAction)

你可以通过开关“Repeat State Actions”去设置是否重复 List 中的 Action

你也可以通过 Inspector 设置 List 的执行顺序“Sequencer”顺序执行 还是“Parallel”并行执行!

例如下图：current+=1，并且 repeat，满足条件后切换到 current-=1，满足条件后再切回 current+=1;



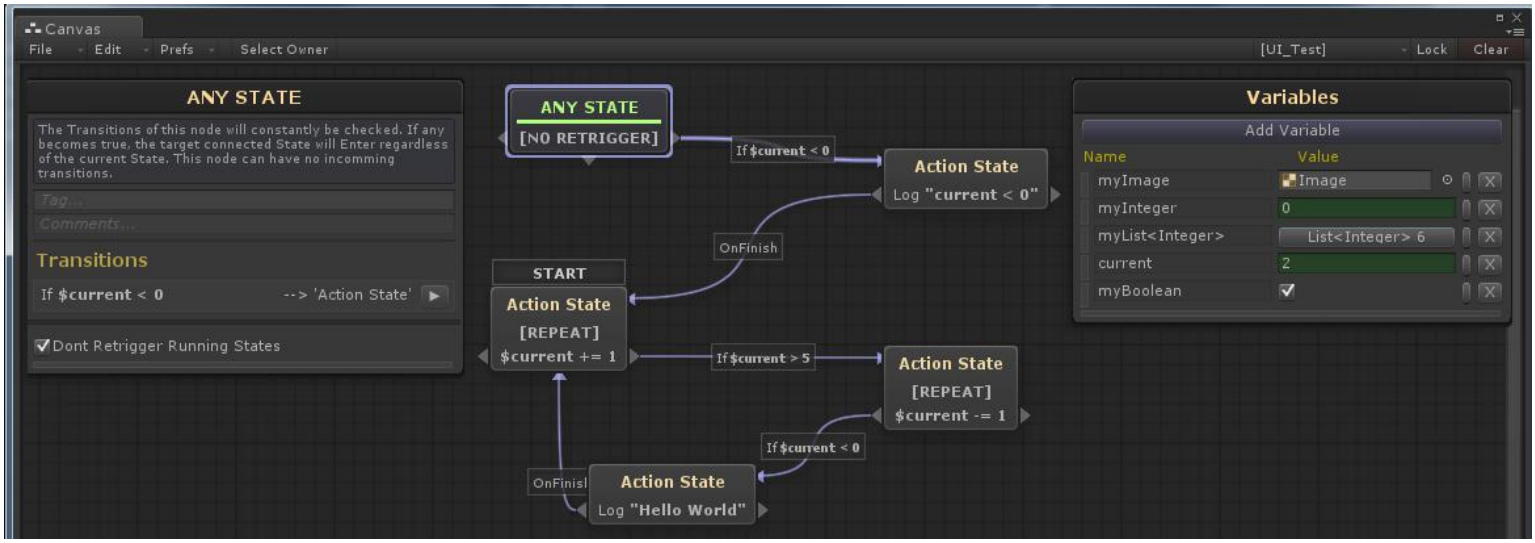
Any State（节点）

这个 Any State 节点的作用就是通过一个 Condition 判断从任何一个节点跳转到另一个节点！它可以包含任意多个的 Transition 连接！如果任意一个 Transition 的 Condition 判断变为 true，那么 FSM 当前正在执行的 State 都会执行 Exit 并跳转到目

标 State 并执行 Enter 操作！（译者：类似于动画状态机中的 Any State）

你可以通过标记“Don’t Reenter Active States”（不要重复进入已激活的 State）来禁止从 Any State 切换到已经在激活中的 State。

例如下图，会在两个 Action 节点之间来回循环执行！当 current<0 时会触发 Any State，不再触发 log“Hello world ”的 state，并输出 log ”current<0”，输出完会再次执行+=操作。



Concurrent State （并发状态节点）

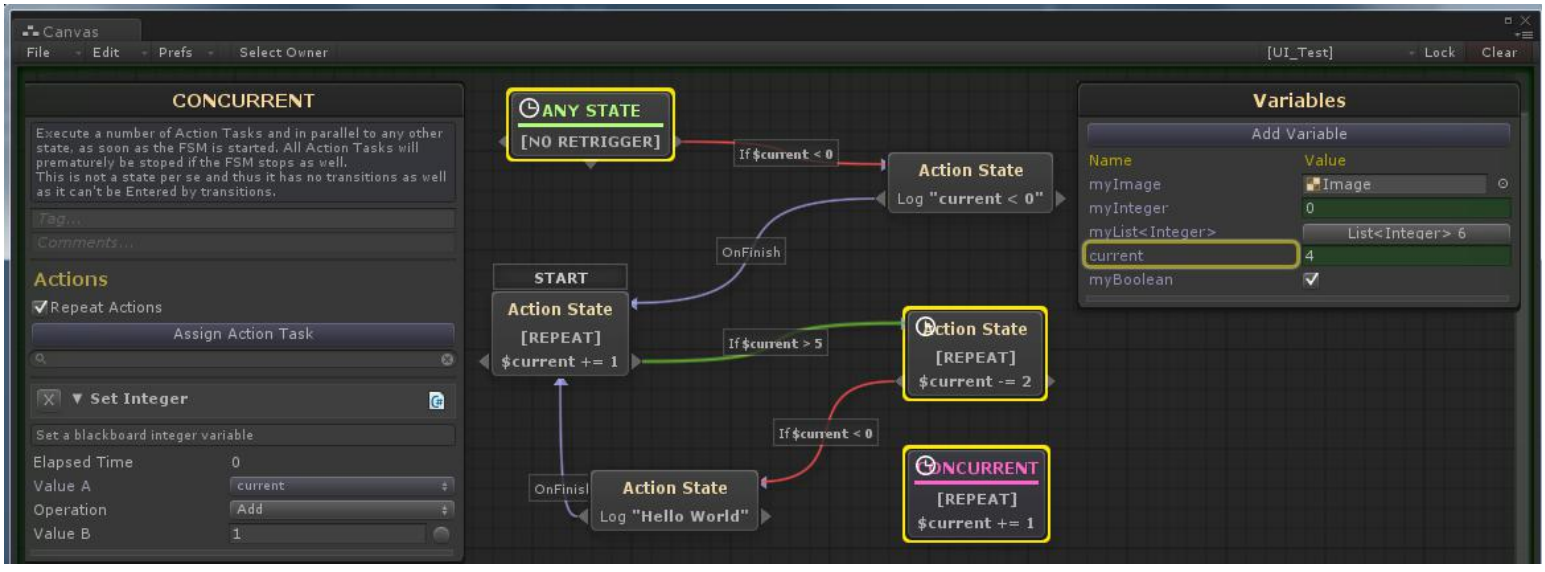
并发状态节点表示并行去执行多个 State。它本身并不是一个 State 状态，独立运行于 FSM 中。因此他既没有输出 Transition 也没有输入 Transition！它下面的 Action 会在 FSM 状态为 finished 的时候停止。（译者:可以理解为 FSM 全局一直在运行的 Action 队列，确保勾选 Repeat）

*** OnEnter：** 所有 list 中的 Action 会被执行(OnExecute)

*** OnExit：** 所有 list 中的 Action 会被暂停(OnStop)

*** Finished:** 当所有 List 中的 Action 都执行完毕(EndAction)

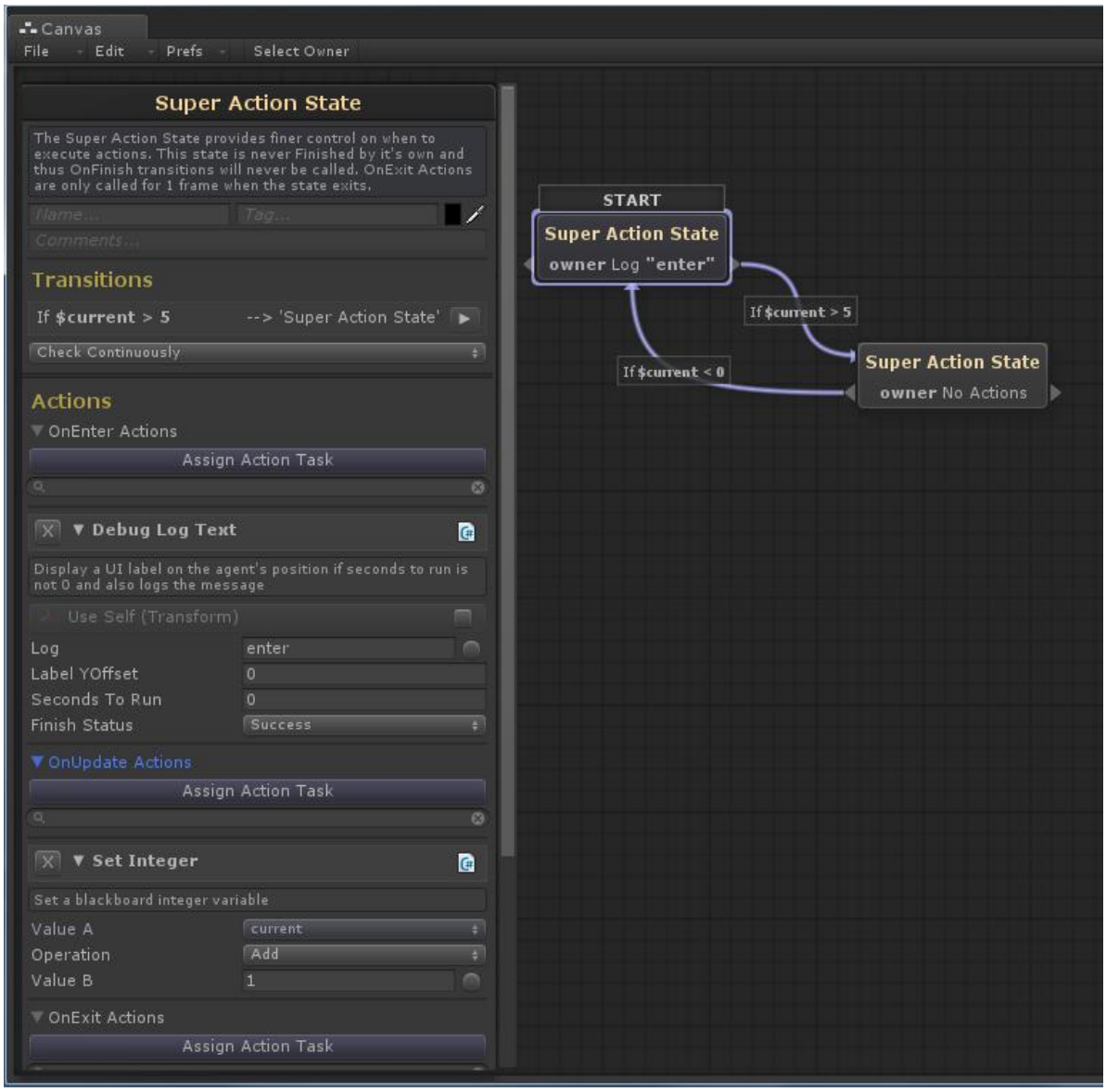
如下图：并发节点开启 Repeat Action 后 上面的 current 需要每次-=2 才行，否则 current 值一直是 5.因为在并发节点中没个 Update（Tick）都在 current+=1；



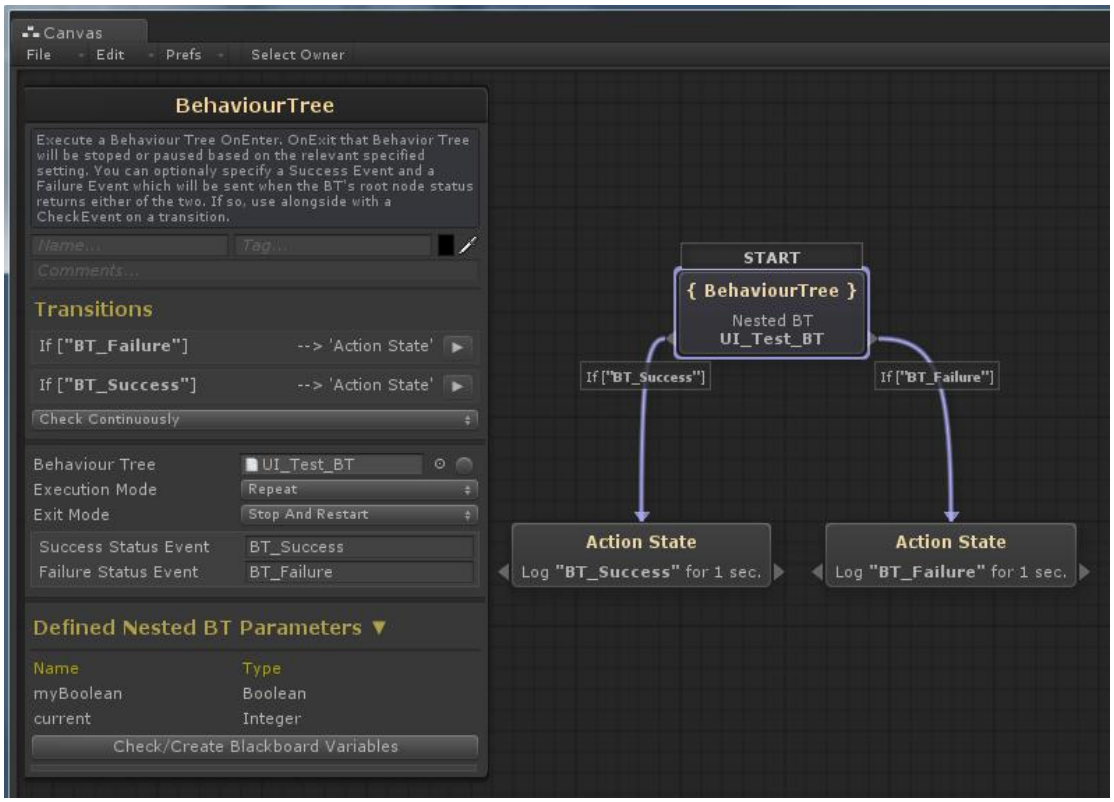
Super Action （超级 Action 节点）

Super Action 提供了对于 State 节点中 Action 的完整控制能力。提供了 OnEnter，OnUpdate，OnExit 三个 Action 的 list，不在提供单独的”Repeat State Actions”。这个节点自己不会执行 finished 操作，也就是说 OnFinish 这个 Transition 永远不会起作用！OnExit 的 Action 只在退出时被执行一次！

如下图：



Nested Behaviour Tree （嵌套的 BT 节点）



嵌套的 BT 节点用来绑定一颗行为树。你可以在 inspector 中的设置行为树是执行一次，还是循环执行！如果设置成 once

则行为树只执行一个循环（不是一个 update 而是保证整棵树执行完为一个循环）！你有两个 event 事件可以定义，一个是行为树 Success 的事件另一个是 Failure 事件。无论哪个被触发对应的事件都将会被发送出去。使用“Check Event”的 condition 来判断行为树是返回 Success 还是 Failure！

嵌套 BT 这个功能很强大，你可以在逻辑上次使用状态机，而在具体状态中使用行为树！

*** OnEnter：BT 树将执行**

*** OnExit：BT 树将停止**

*** Finished：当 BT 树结束时**

当 FSM 被暂停时，嵌套的 BT 树如果在执行中那么也会被暂停掉！

嵌套的 BT 树可以绑定到黑板中的变量！这样可以动态修改所使用的 BT 树！

Nested FSM （嵌套的 FSM）

FSM 可以嵌套 FSM，OnEnter 时执行嵌套的 FSM，这个节点会在嵌套的 fsm 变为 finished 的时候返回 finished，同样如果这个节点的 Transition 判断为 true 的话则会打断嵌套 fsm 的执行。

*** OnEnter：FSM 树将执行**

*** OnExit：FSM 树将停止**

*** Finished：从不 finished，除非嵌套的 FSM 被 stop 了！**

当 FSM 停止后，如果嵌套 FSM 正在运行，那么也会停止！

嵌套 FSM 可以绑定到黑板中的变量，这样你可以动态修改使用的 FSM！

6.2 FSM Callbacks （FSM 回调）

如果你在 FSOwner 所在的 gameobject 上添加了自己的脚本组件，那么当 FSM 在执行的时候，无论是 Enter，Update，还是 Exit，你都可以在你的脚本中监听到这些状态变换的回调事件：

* OnStateEnter(Istate)

* OnStateUpdate(Istate)

* OnStateExit(IState)

下面是个事例：Istate 是一个 Interface 接口，它包含了 current State 的所有有用信息！

```
using UnityEngine;

public class StateCallbacksExample : MonoBehaviour {

    public void OnStateEnter(IState state){

    }

    public void OnStateUpdate(IState state){

    }

    public void OnStateExit(IState state){

    }
}
```

```
public interface IState{

    //The name of the state
    string name{get;}

    //The tag of the state
    string tag{get;}

    //The elapsed time of the state
    float elapsedTime{get;}

    //The FSM this state belongs to
    FSM FSM{get;}

    //An array of the state's transition connections
    FSMConnection[] GetTransitions();

    //Evaluates the state's transitions and returns true if a tra
    bool CheckTransitions();
}
```

这些 callback 的主要作用不是实现整个状态的逻辑，而是给你一个处理额外操作的机会！

6.3 Creating Custom FSM Nodes（创建自定义 FSM 节点）

Nodecanvas 建议大家不要直接创建节点，而是创建对应的 ActionTask 或者是 ConditionTask!

要创建自定义节点，你需要继承自 FSMState 并 override 一些必要的函数，最后调用 Finish().当节点创建完毕后你可以在 FSM 的 Canvas 中右键菜单中查看到，你甚至可以自定义一些图标用于节点的显示！下面是个事例：

```
using UnityEngine;
using NodeCanvas.Framework;

namespace NodeCanvas.StateMachines{

    [Category("My States")]
    [Icon("SomeIconName")] //Icon must be in a Resources folder
    public class SampleState : FSMState {

        public BBParameter<float> timeout;
        private float timer;

        //When the FSM itself starts
        protected override void Init(){
            Debug.Log("Init");
        }

        //When the state is entered. Not when it is resumed though
        protected override void Enter(){
            Debug.Log("Enter");
        }

        //As long as the state is active
        protected override void Stay(){
            timer += Time.deltaTime;
            if (timer >= timeout.value)
                Finish();
        }

        //When the state was active and another state entered thus thi
        protected override void Exit(){
            Debug.Log("Exit");
            timer = 0;
        }

        //When the state was active and FSM paused
        protected override void Pause(){
            Debug.Log("Pause");
        }
    }
}
```

就是这样，记住双击可以在 IDE 中打开并编辑这个节点的脚本！

最后，讲一下 FSMState 中的重要属性和方法：

*** FSM FSM**

当前 State 的上一个 FSM graph；

*** Component graphAgent**

FSM 的代理

*** Blackboard graphBlackboard**

黑板

***float elapsedTime**

当前 state 已经运行的时间

***void Finish()**

结束当前 State 时调用这个接口

***void SendEvent(string name)**

用来向 FSM 图发送事件

7. Dialogue Trees （对话树 DT）

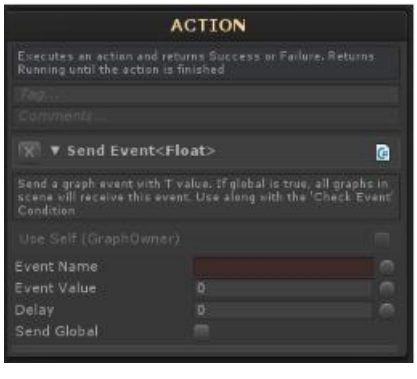
略。。。。。。

8. Using Graph Events(在 graph 图中使用 Event)

Graph Event 用来在整个行为树图形中分发和处理事件。Nodecanvas 拥有一套内置的事件系统。事件可以传递/接收任何已经添加的数据类型！

Sending an Event

你可以通过内置的 Action Task 去发送事件（SendEvent 或 SendEvent<T>）。当使用 SendEvent<T>这个泛型版本的时候你会在 inspector 中看到这个类型所对应的变量的值。下面以 SendEvent<float>为例：



你也可以通过代码来发送事件。例如下图：

```
public class Example : MonoBehaviour{

    public BehaviourTreeOwner owner;

    public void SendNormalEvent(){
        owner.SendEvent( "MyEventName" );
    }

    public void SendValueEvent(){
        owner.SendEvent<float>( "MyEventName", 1.2f );
    }
}
```

Checking an Event

要监听/获取某个发送的事件，你可以使用下图的 Condition task。

获取的事件数据你可以存储到黑板中。在同一帧中或者说是在同一个 Update/Tick 中所有对同一个事件的 check 都会被执行，下一帧事件就会无效！在同一帧中发送事件给自己是无法监听到事件的只能等到下一帧才能监听到事件。

在 FSM 中使用事件系统并不是很方便，在 BT 中使用事件系统则能够带来极大的便利，举个例子:

* 你可以在 BT 中的 interrupt 打断节点中使用事件判断，如果收到事件则打断子节点操作！
* 你也可以用事件作为判断条件去触发某些子节点的操作！