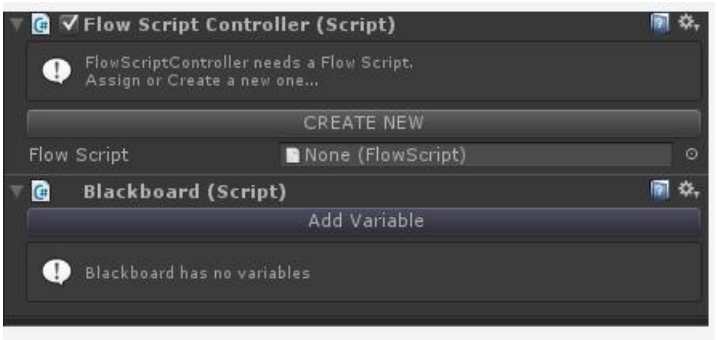
声明：本文档由(炼金师-金奇 译）文档内容并没有进行严格的校验可能存在错别字等，当前翻译版本为 v1.3.0b 其中关于 Network Sync 的部分没有翻译。如果文档中任何有异议的地方，请参见官方文档！如果你觉得本插件好用，请支持正版！请支持正版!请支持正版!（补充说明：FlowCanvas 的扩展阅读：http://www.jinqi166.com/2016/11/09/unity_flowcanvas_ugui_3/）FlowCanvas 交流 QQ 群:117182310

FlowCanvas 离线帮助文档

1：开始学习

1.1 关于 FlowScript Controller

FlowCanvas 中最重要的一个 component 组件就是这个 **FlowScriptController**。从它的命名就能看出，它负责了管理和控制 flowscript（FlowCanvas 的功能脚本）. 一般来说从这个组件开始学习是比较简单的，当然你也可以从其他组件开始学习！当你在某个 gameObject 上添加了这个 FlowScriptController 组件之后你会发现另外一个叫做“Blackboard”的组件也被自动添加了！这个组件负责创建和保存上面这个 Controller 中需要的变量，更多关于这个 Blackboard 的介绍将在后面进行阐述！

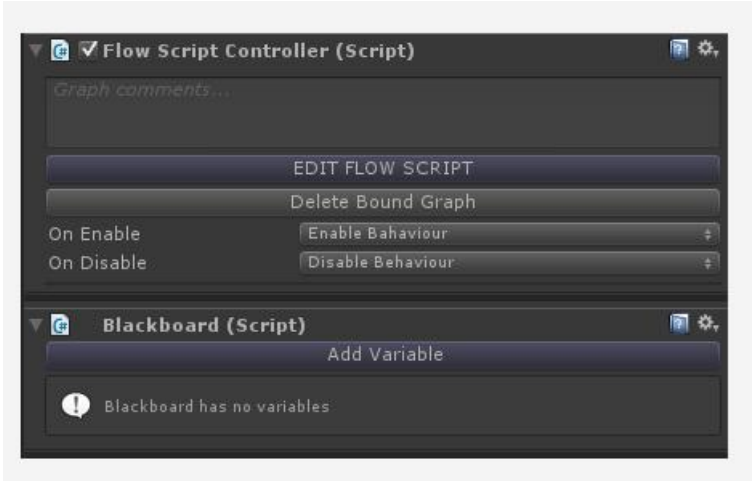


FlowScriptController 需要一个 FlowScript，你可以拖拽一个上去，或者点击“Create New”按钮创建一个！在创建的时候 会弹出一个对话框用于确认即将创建的这个 FlowScript 是直接绑定到当前的 Controller 上（Bound）还是作为一个独立的 资源存在（Asset graph）！

A：当选择了 Bound 的时候,创建的 FlowScript 直接绑定到当前 gameobject 上的 FlowScriptController 中！并且可以对当前绑定的 GameObject 进行引用（获取/设置所在物体的一些属性或方法）

B：当选择了 Asset graph 的时候，将在你的当前的项目工程中创建一个后缀名为.asset 的文件。这么做的好处就是可以在游戏内其他任何 FlowScriptController 上共享使用这个 FlowScript。缺点就是独立的 asset 文件，所以不存在绑定的 GameObject 的引用！

创建一个 Bound 是默认推荐的方式，不过不用担心，当你创建完 FlowScript 之后可以随时切换到 Asset 资源！会有对应的按钮进行删除操作，如下图：



1.2 了解 Flow 原理，即工作流程！

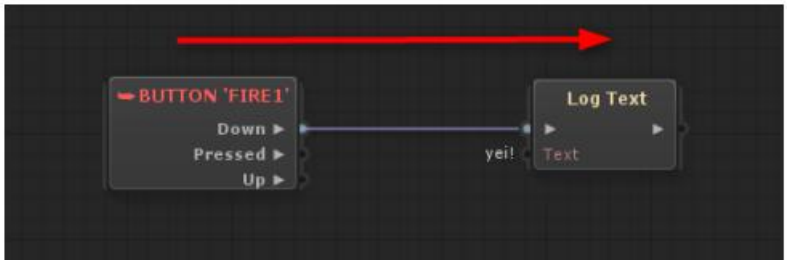
在开始学习之前有必要了解下什么是 FlowCanvas，以及它的工作原理！它是如何实现数据衔接以及如何在 node（节点）

之间来回跳转的！下面就来介绍两种不同的不同的衔接端！

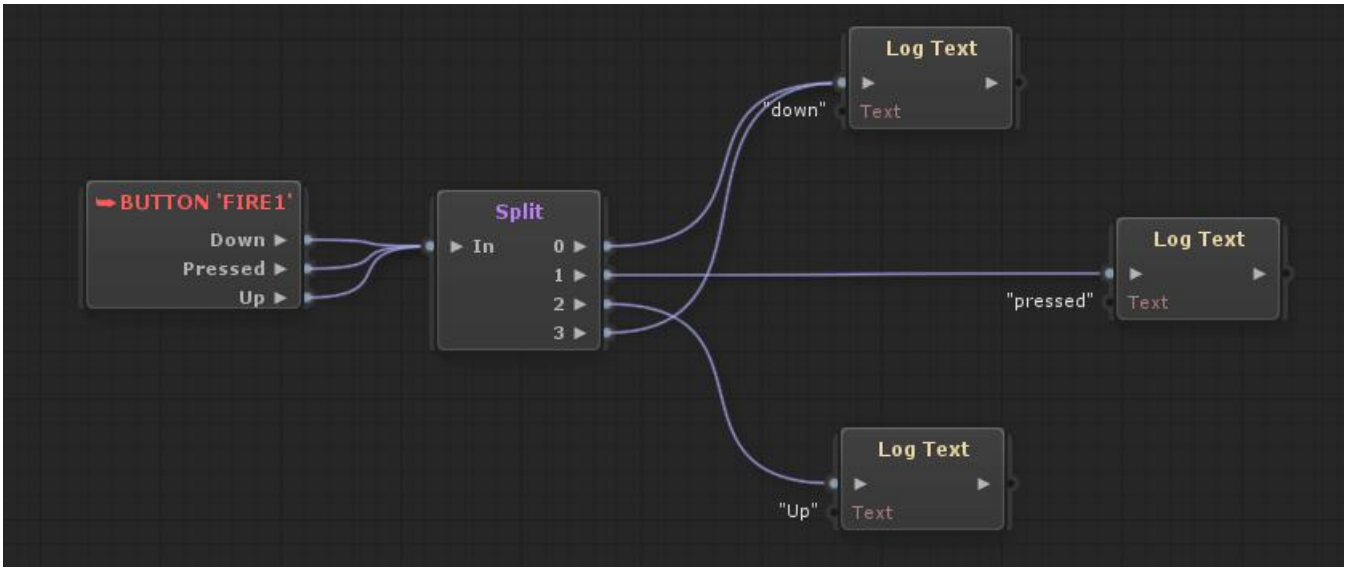
Flow Ports(流程端，流程端是一个统称，每个流程端都是一个 node 节点)

流程端是一些能够处理单独操作的（函数）。相当于调用一次某个函数完成某项任务。Flow 的执行顺序是从左到右的，类似于代码中调用函数时从上到下一个接一个的调用。所以，当你链接多个 flow ports 节点的时候相当于设置了函数的调用顺序一个接一个的调用！

在接下来的例子中，事件节点会在用户按下“Fire1”的时候执行一次 flow（流程），并触发后续的 Log 操作！



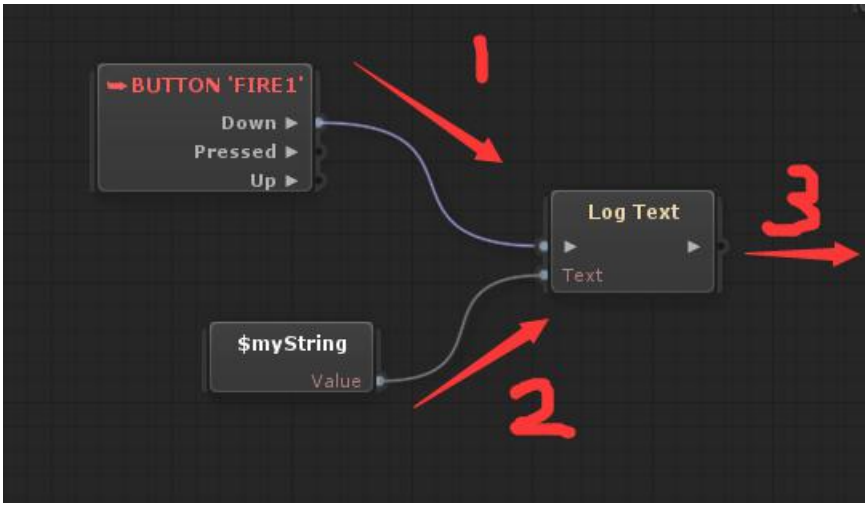
* Flow 的 outputs 输出端只能有一个输出链接（如果你想在这里产生分支，请使用 Split 节点）
* Flow 的 inputs 输入端则可以拥有多个输入链接。
* Flow ports（流程端）只能链接到其他的流程端，不能链接到 value 端



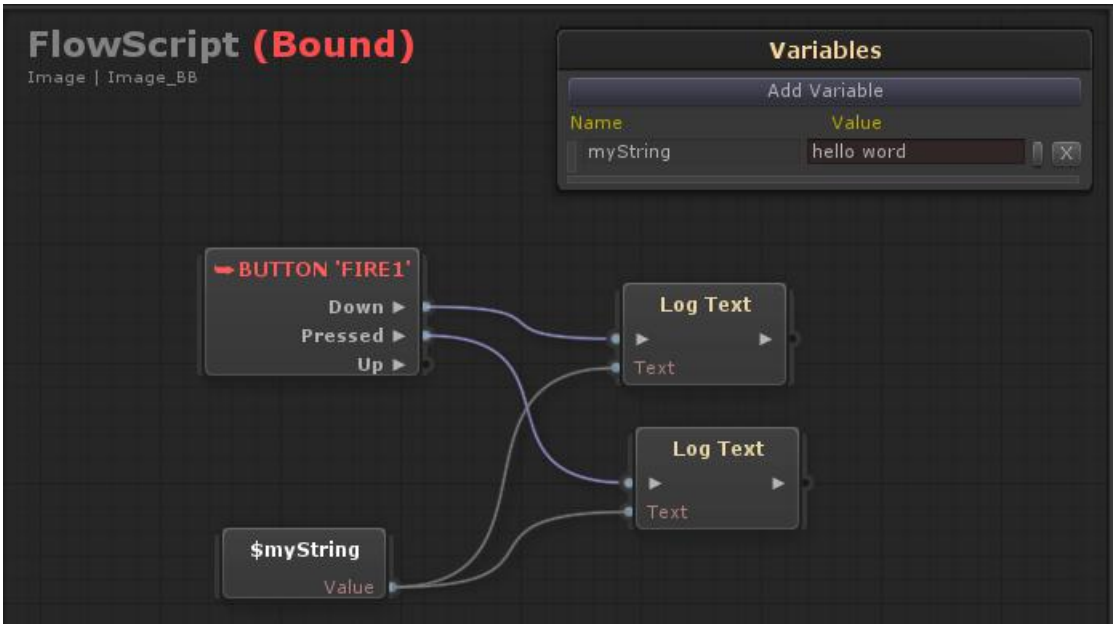
Value ports (数值端)

顾名思义，数值端就是一些数值，例如 int，bool，string，GameObject，以及其他任何数据类型！数值端的功能相当于代码中的数据获取操作。不同于流程路径，数值端只有输出端，没有输入端。所以数值端相当于 Get 操作！

在下面的事例中，在 log 被显示之前“Text”属性会被“hello”这个数值端的数据赋值并最终输出！



* Value ports（数值端）的 outputs 输出端口可以链接到多个输入端。
* Value ports（数值端）的 inputs 输入端 无！（在属性面板进行添加）
* Value ports（数值端）只能被链接到相同数据类型的流程端节点上的对应属性或者可隐式转换数据类型的属性！

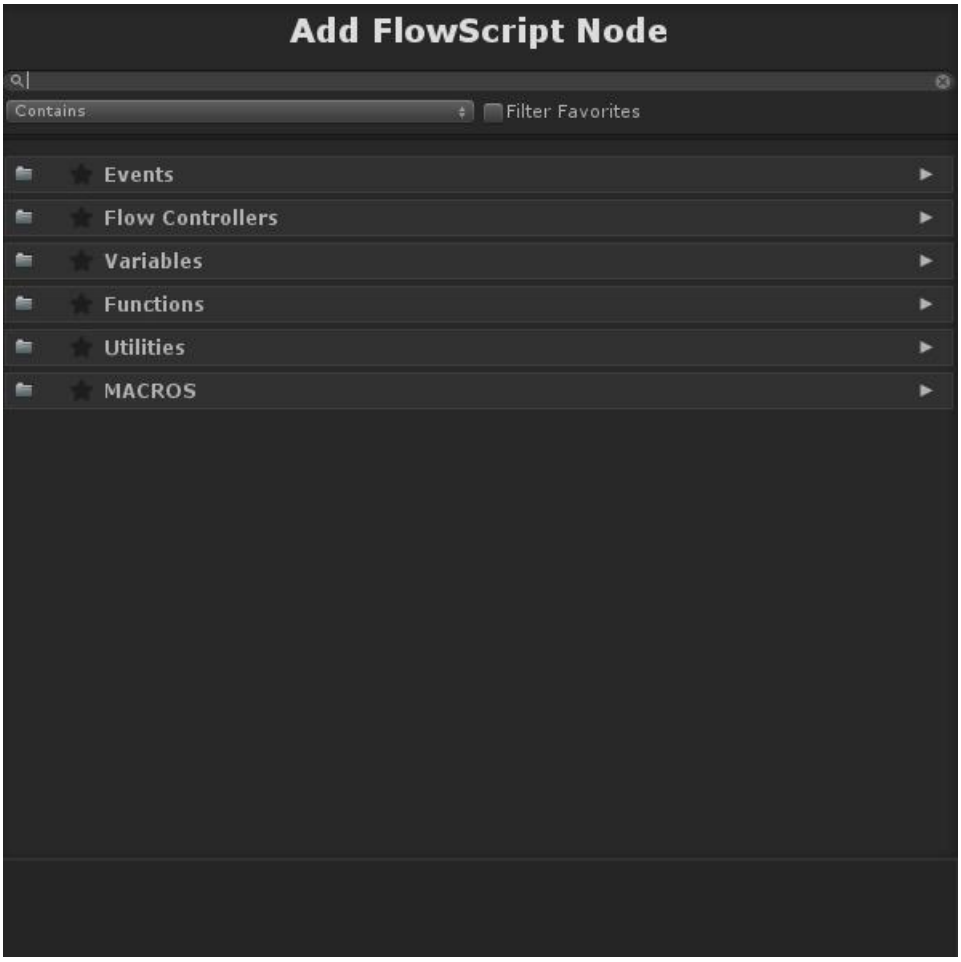


1.3: 添加 Node 节点

有很多种方法可以在 flowscript 中添加节点。下面来逐一进行介绍！

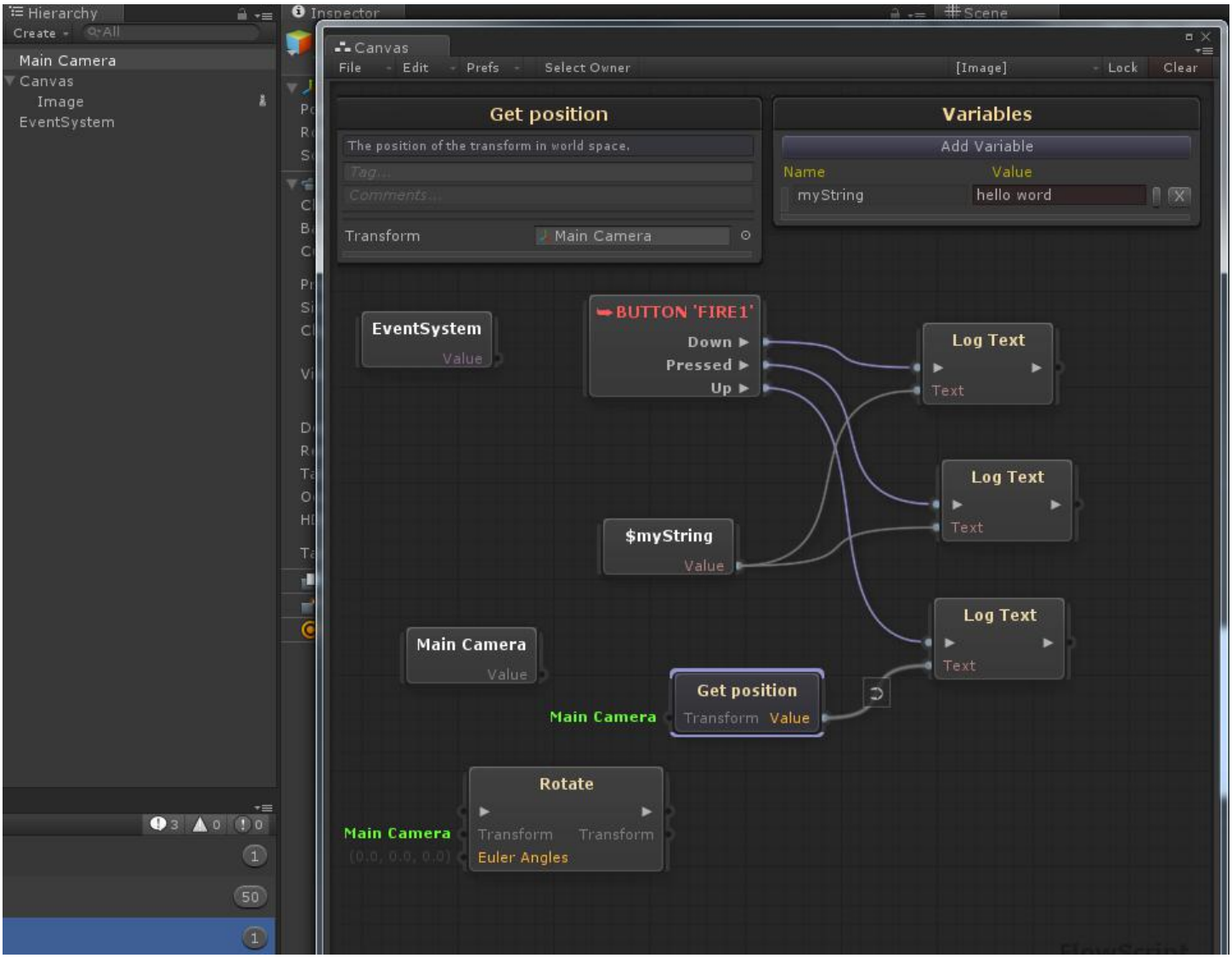
Context Menu & Browser （右键菜单和功能查看器）

最快捷的创建方式是使用鼠标右键菜单，或者是通过点击键盘上的“空格”键来打开功能查看器！上面这两种方法都会打开功能查看器！如下图：



Drag &Drop Objects （拖拽对象）

另一种添加节点的方式就是拖拽，你可以拖拽任何类型的资源文件或是实例化在场景中的对象（场景文件，文件目录，脚本，纹理）并创建对应的节点，根据拖拽类型，你可以在创建时选择是创建流程端，还是创建数值端。对于流程端而言你可以选择拖拽对象上能访问到的任何方法。对于数值端你可以选择任何对象上的属性（get/set），而对于字段你无法直接访问，即便是 public 的字段！（译者：建议大家都把需要访问的字段添加一个属性进行访问！）



Drag&Drop Port Connections (拖拽输入输出端的端口)

直接拖拽端口处的点，进行操作是最省时也是最方便的。从端口处拖拽所弹出来的界面中根据上下文以及端口类型自动过滤掉了不可用的节点，这样你可以更加快速的找到你想要的节点进行创建,创建完毕后自动进行连接操作！

1.4 Connecting Nodes 节点的连接操作

最直接的方法就是点击节点两边的端口（圆点）进行拖拽操作。连接操作是双向的，即你可以从当前节点拖拽连接到目标节点，也可以从目标节点拖拽连接到当前节点！

删除操作就是选中某个节点，从右键菜单中选择删除！或者选中后直接 delete 删除！

你也可以通过拖拽端口到空白区域进行节点创建操作，创建完后将自动连接！

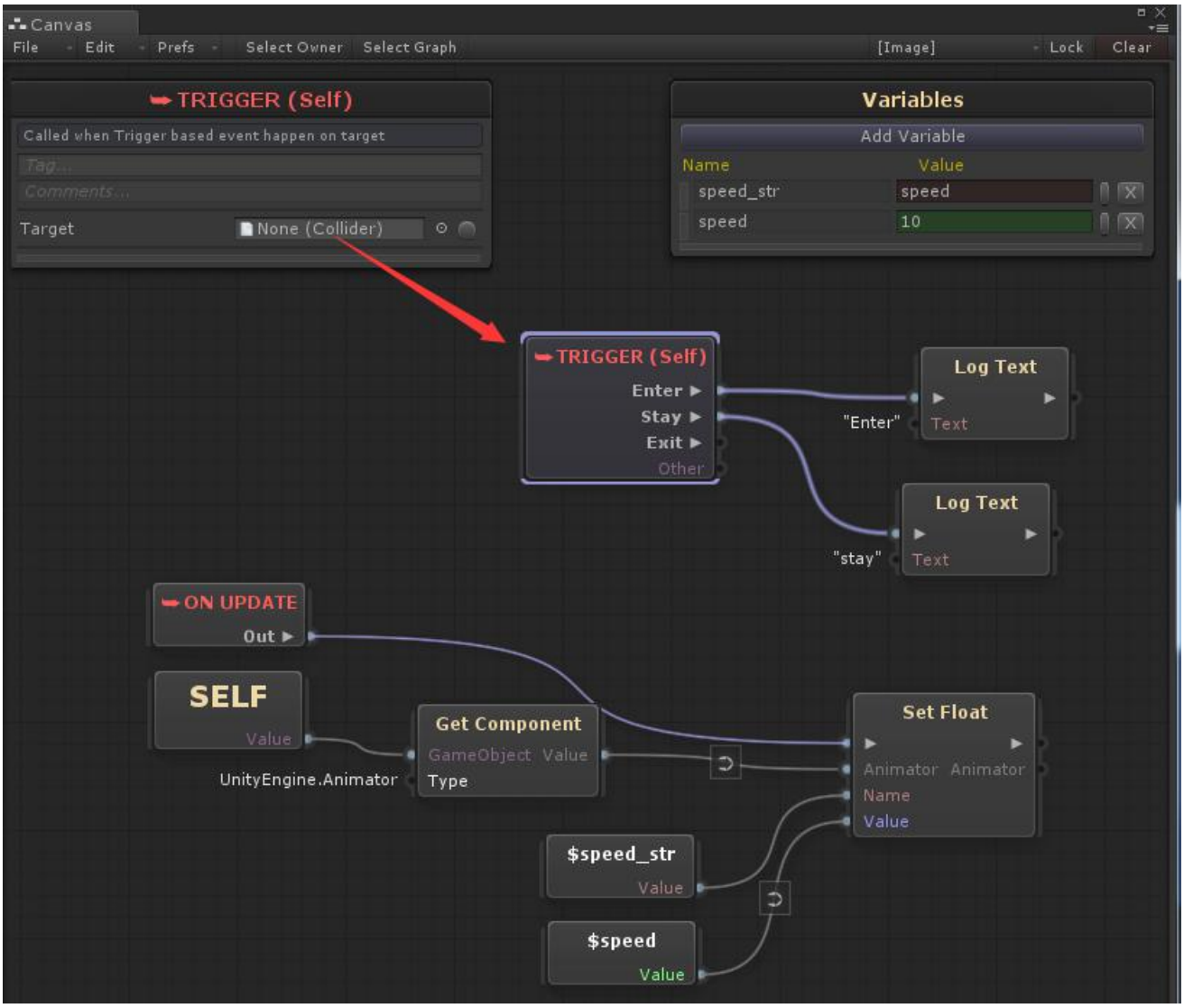
删除连线的操作是鼠标右键点击小圆点（端口）

1.5 The “Self” Parameter （self 参数）

在 Flowcanvas 内部，self 参数代表着当前 FlowScript 的拥有者，也就是 FlowScriptController 所绑定的 GameObject 对象！如果某个节点中包含了属性是 GameObject 或者是 component 并且引用的对象是自己的话则可以看到这个 self 的选项！

例如在一个 Gameobject 对象上有一个 FlowScriptController,并且包含一个 Animator 组件，添加如下节点的时候则能看到 Animator 组件引用的是 self 自身的这个 Animator 组件！

另外一个使用“Self”的地方是 Event nodes(事件节点)。默认使用 self 自身的碰撞盒，你也可以指定其他的碰撞盒！在下面这个示例中，将通过检查是否和自身的碰撞盒或者指定的碰撞盒发生碰撞，以便做后续操作！



1.6 Visual Debugging 帮助信息

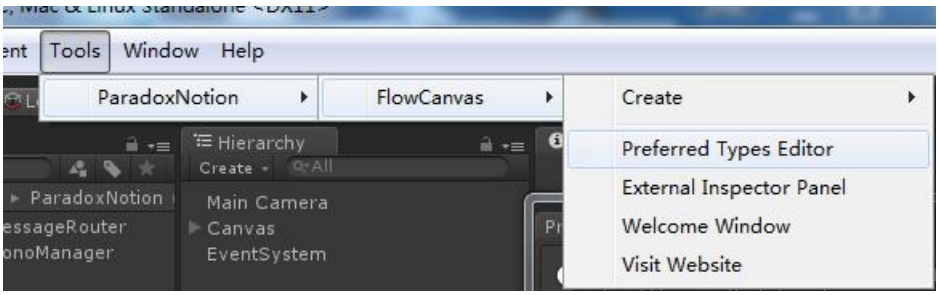
Flowcanvas 提供了有用的 Debugger。在运行时你可以看到整个 flow 工作流的执行顺序。并且可以打断点。在节点两端的端口上提供了备注以及一些必要的提示信息，方便预览（点击节点可以在左上角进行编辑）如上图中的 Self.Getcomponent 的 type 类型！

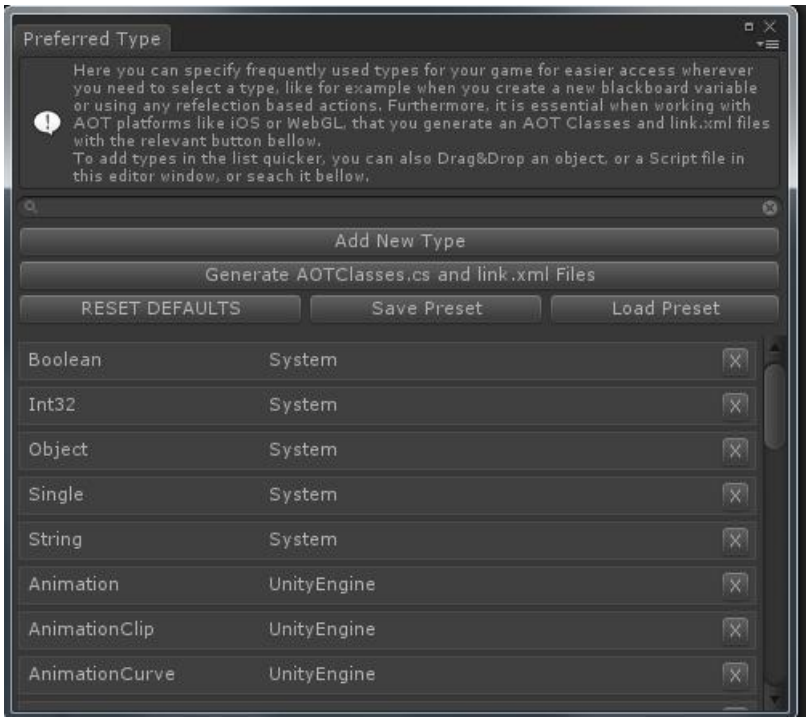
2 ： Working with Types (类型的使用）

这是一个很重要的窗口！

FlowCanvas 的目标是能够处理任何的数据类型，任何类型都能在节点中进行操作！包括自定义类型！

打开类型编辑器





这个界面是从 NodeCanvas 插件集成过来的并扩展的！

这个列表里的类型有以下几个特点：

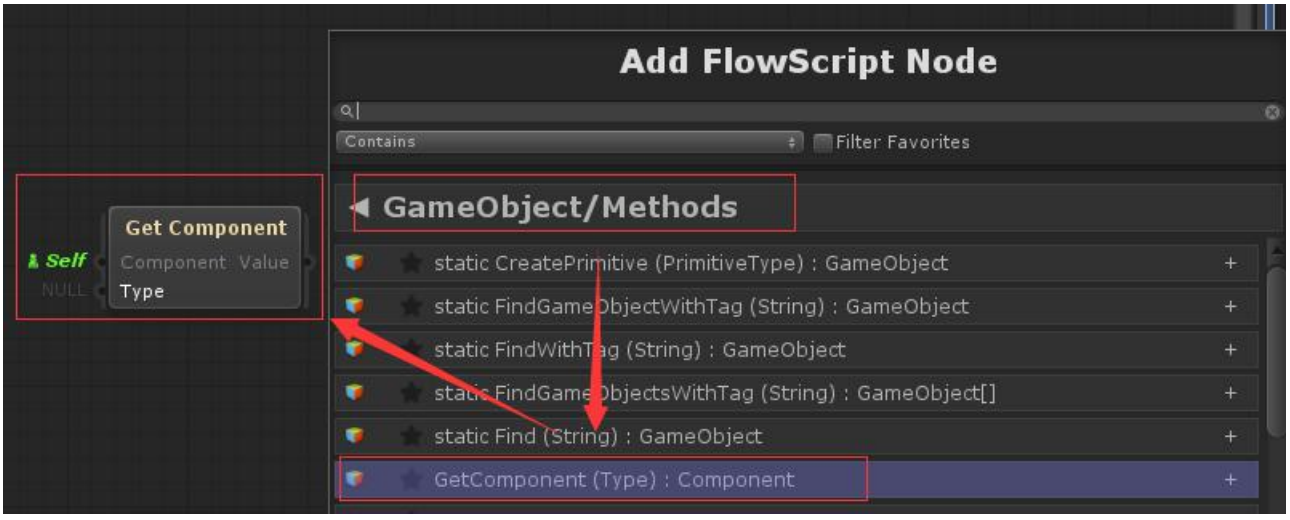
1：列表里的数据类型，你可以通过 Blackboard(黑板)进行数据类型的访问和使用！

2：自动生成反射节点

3：所有泛型节点和菜单都是以（T）结尾！

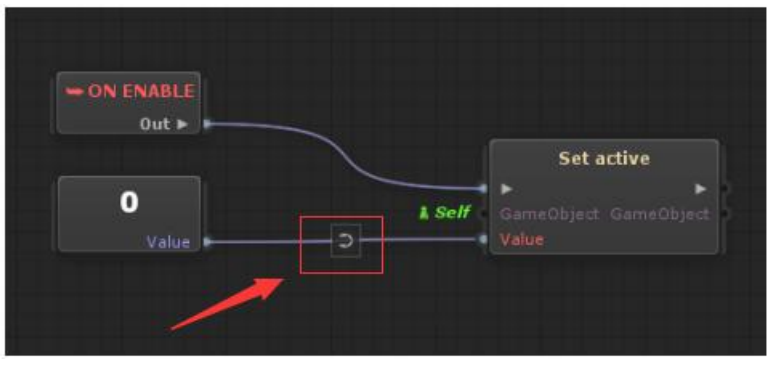
2.1. Generic Type Nodes（泛型类型节点）

当你选择某个带有泛型的节点进行操作时，你会在选择面板中看到（Type）的标记。创建完节点后你可以选择 Type 类型，这些 Type 类型就是你在上方介绍的类型面板里添加的类型！



2.2. Connecting Assignable Types(可连接的类型)

正如之前说过的，变量类型的节点只能连接到可连接或可转换的类型节点！意思就是说类型不匹配的时候回进行强转操作，当然无法强转的则不会转换！这个转换过程是自动的，无需手动参与！如果发生了类型转换操作，则在连线的中间会显示一个转换的标记如下图：

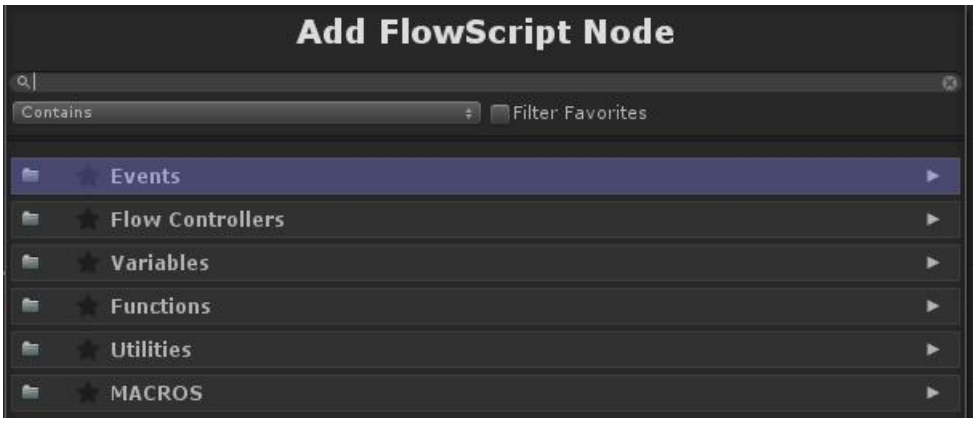


下面列出了一些 FlowCanvas 默认的类型转换：

源数据类型	目标数据类型	转换方式
Any Primitive	Any Primitive	.ConvertTo()
GameObject	Any Component	.GetComponent
GameObject	Transform	.transform
Any Component	GameObject	.gameObject
GameObject	Vector3	.transform.position
AnyComponent	Vector3	.transform.position
A Child Type	Base Type	upcasting
A Base Type	Child Type	downcasting
Anything	String	.ToString()

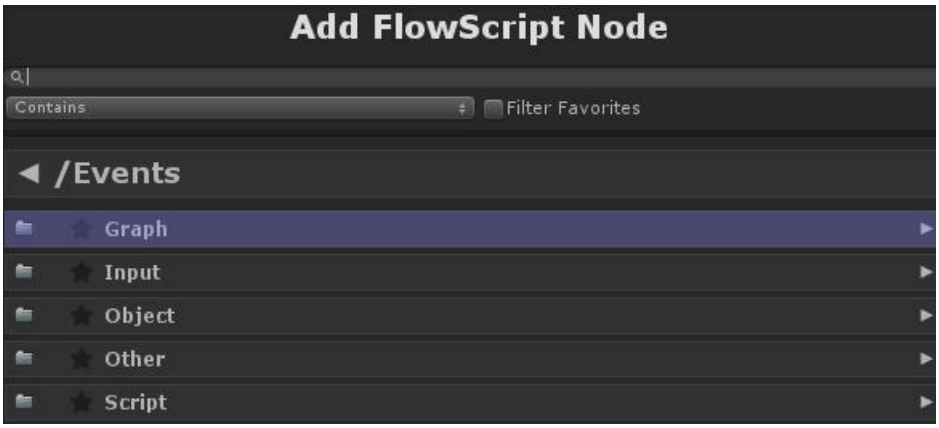
3. Node Categories(节点类型)

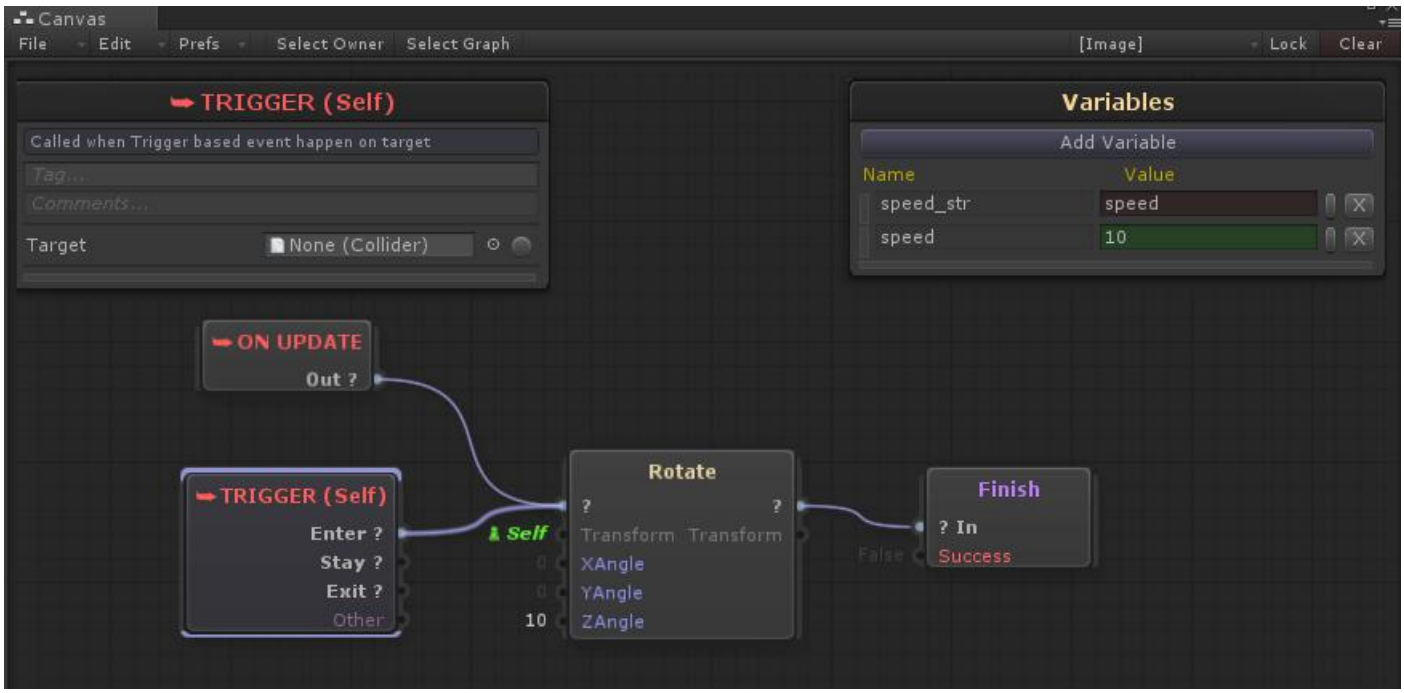
所有 FlowCanvas 支持的 node 节点类型,都可以统计到以下这几种类型中：



3.1 Event（事件）

Event 事件作为整个 Flow 工作流的起点，用来负责触发和执行后续的 Flow 操作！没有事件节点，你的后续 Flow 节点的操作不会被执行！你需要添加一个事件节点来触发事件，例如碰撞事件，输入事件，Update 事件，自定义事件等等。。



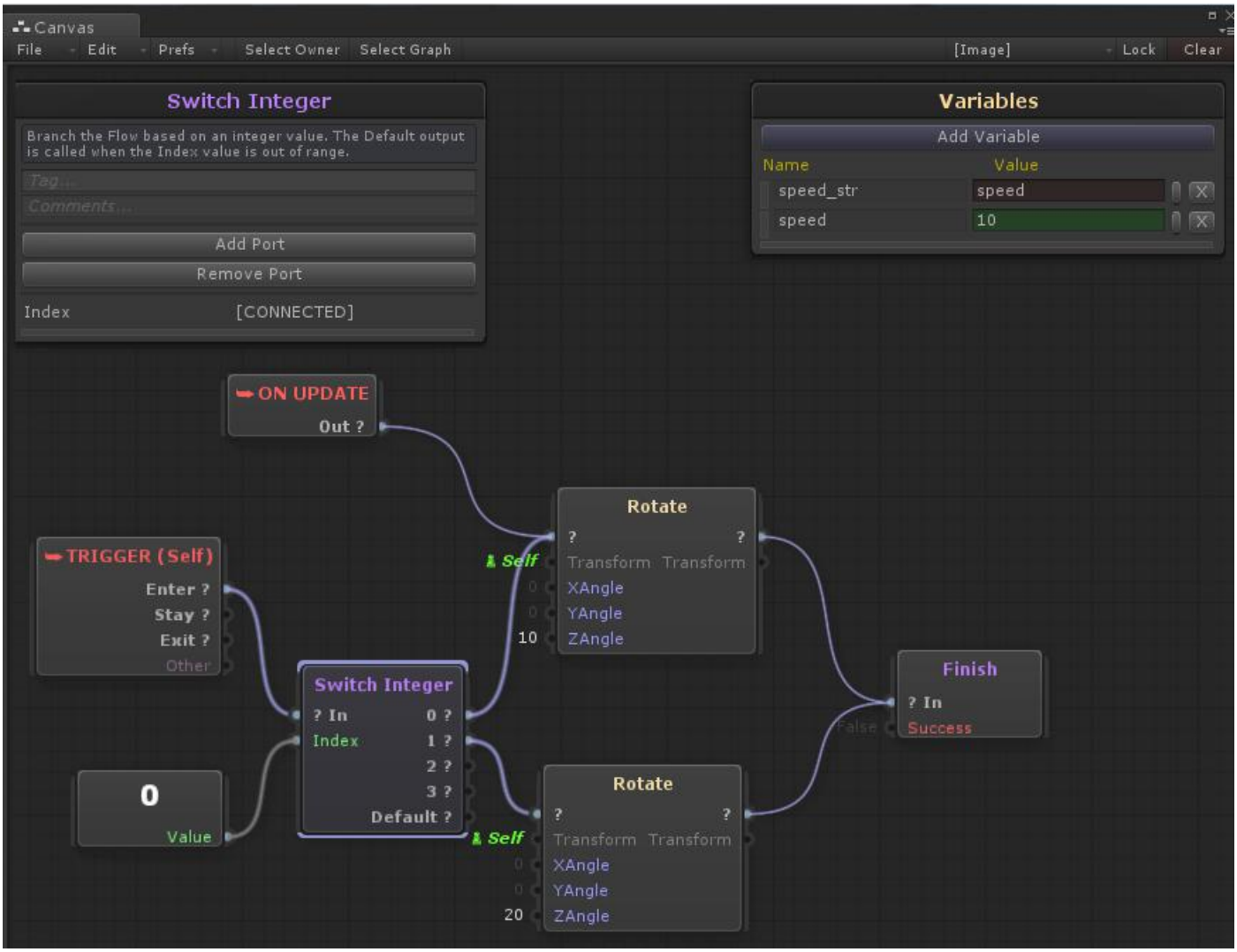


有些事件节点需要一个 Target 目标的参数，这个 Target 目标用来标记事件的发起者！例如上图中的 Trigger 事件节点就需要一个 Target 目标对象的引用,以便确定是谁触发了 Trigger(默认情况下是指向 Self 的也就是指向了当前 FlowScriptController 脚本绑定的 GameObject),当然你也可以指定其他的 GameObject。而上图中的 Update 则不需要 Target 目标参数，因为不可以指定其他 GameObject 的 Update 事件，只能使用 self 自己的 Update 事件！Update 的调用是 Unity 系统自己触发的，作为我们用户来说是不能触发的，我们最多是调用这个 Update 函数而无法触发 Update 事件！

所有事件节点的标题颜色都是红色的，并且没有输入端和任意多个输出端。输出端通常是事件类型或一些事件相关的信息参数！

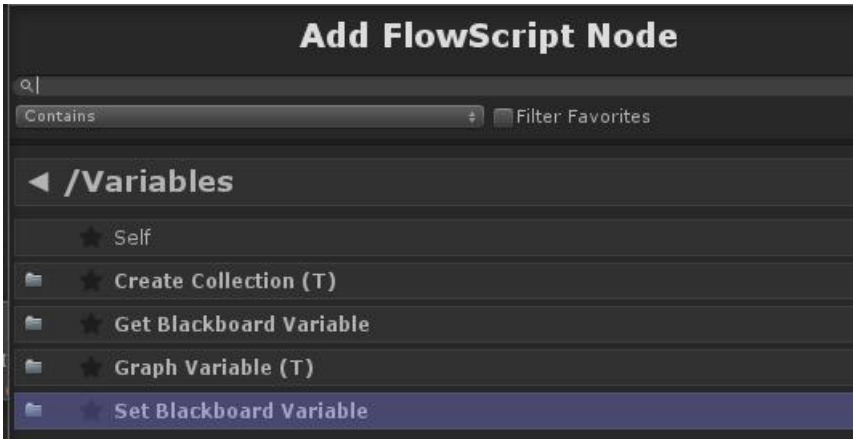
3.2 Flow Controllers

流控制器的作用就是控制整个 Flow 流的走向也就是流程控制，例如：过滤，合并，分支，等等！对应于代码中的流程控制（if-then-else，while，for-each）



上图中的 Switch Integer 就是一个流程控制器，流程控制器的标题使用的是 blue-ish 类似于蓝色的颜色，并且至少有一个输入端和一个输出端！

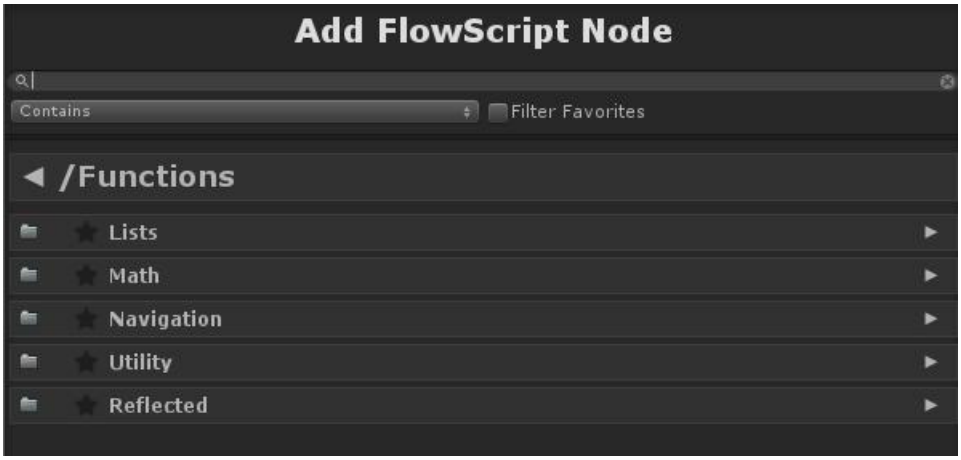
3.3 Variables （变量相关）



如上图所示，合理创建的变量节点包括 self 节点，集合节点，List 节点，引用对象节点，等等。一系列的数据类型都能单独创建一个变量节点！并且能从 BlackBoard 中获取/设置已有的变量！

变量类型的标题文字颜色都是白色的！

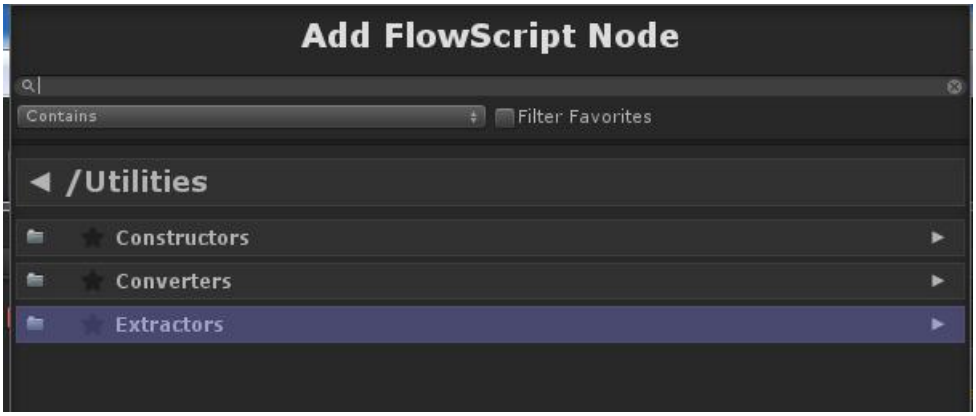
3.4 Function (功能节点）



提供了常用的功能函数，以及对应 Unity 类型的所有操作！默认情况下，功能节点没有没有 Flow 端口，只有数据输入端口和输入输出端口！有些节点拥有 flow 端口，另外还有些端口没有 flow 端口但是可以通过属性设置中的 Callable 开启 Flow 端口！

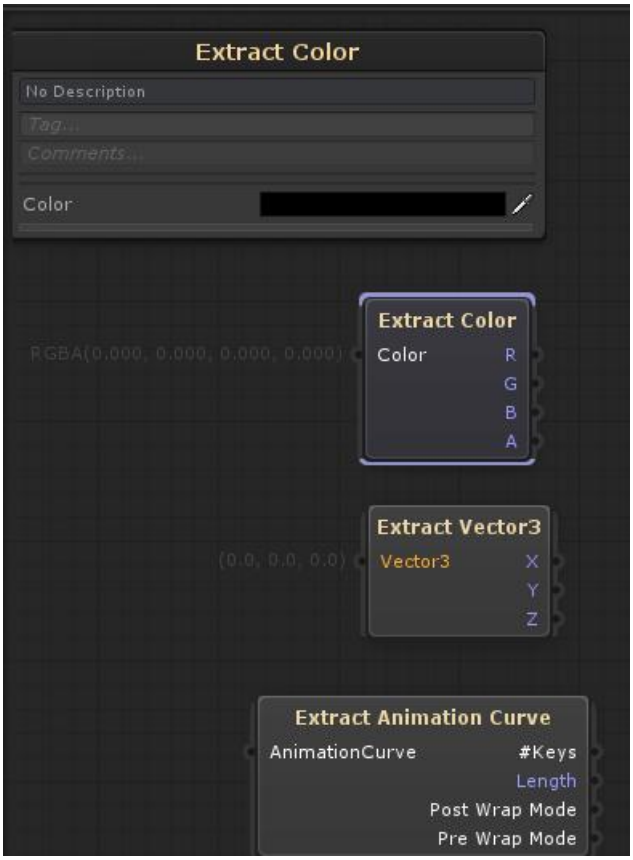
功能节点都是黄色标题颜色！

3.5 Constructors & converters & Extractors (辅助功能：new 操作，转换操作，)

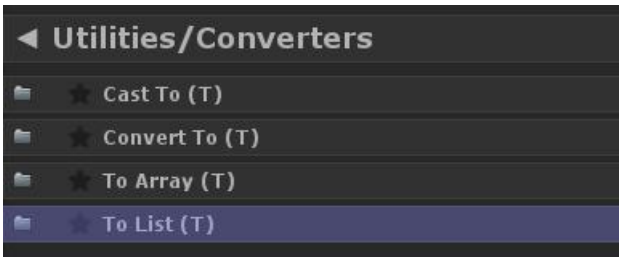


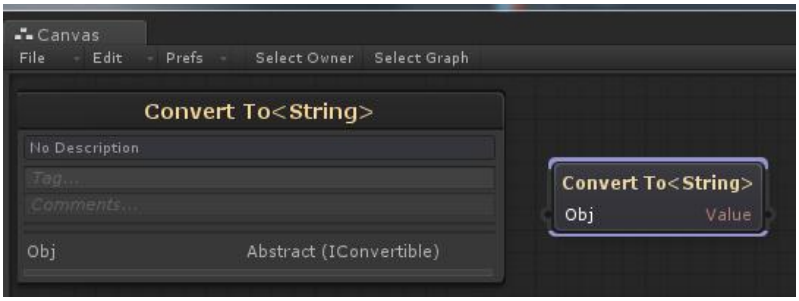
Constructors （new 操作）

Extractors 用于分解数据结构作为输出端，如下图：



Converters 转换

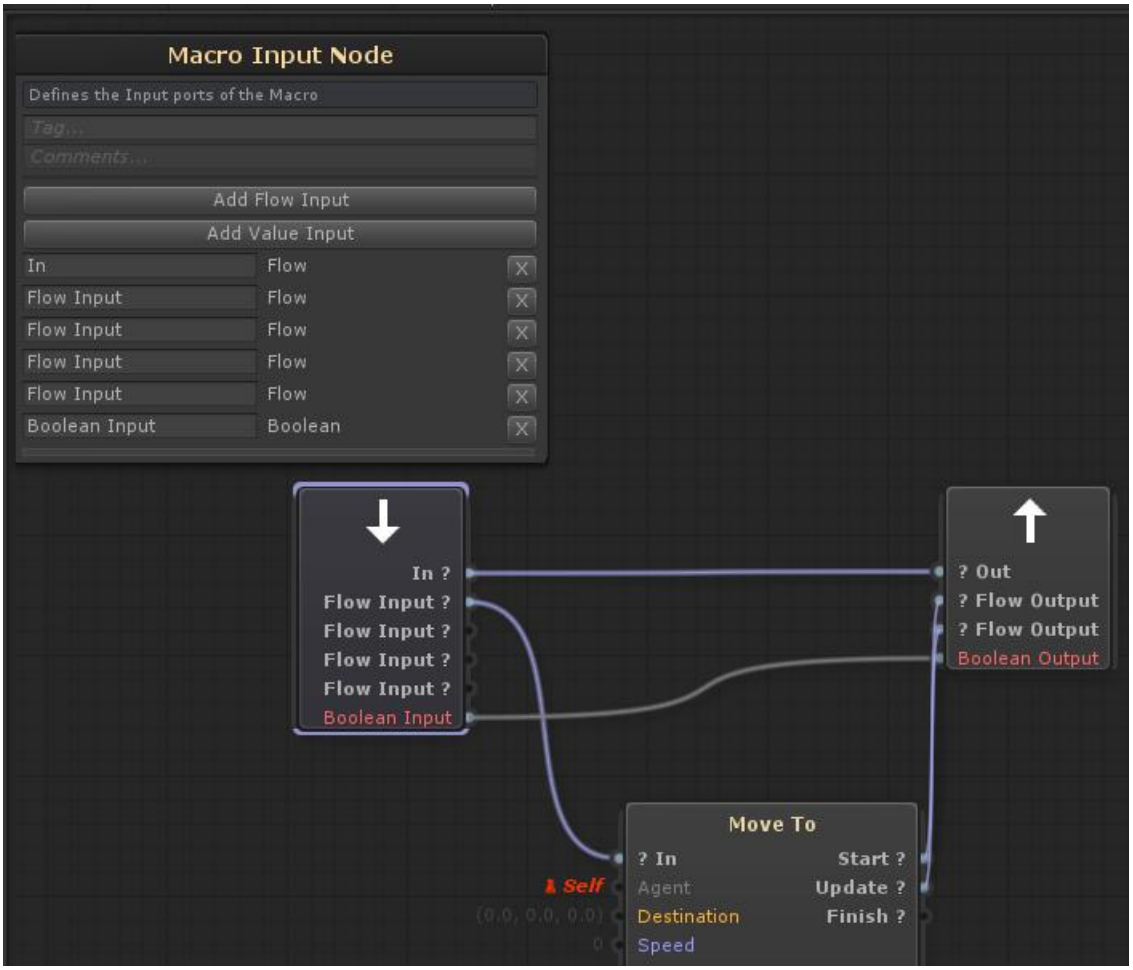




4：variable and Binding 变量和属性绑定

5:Macros（宏命令）

Macros 是一个 node 节点的混合绑定资源文件（以独立的文件形式存放和使用），作为独立资源在其他 FlowScript 中使用！可以设置输入端的 flow 数量和属性，以及输出端的数量和属性，并可以在中间添加想要的任意操作节点！双击进入编辑，编辑完后通过左上角的按钮返回主操作界面，会看到变化！



6：Createing custom nodes（创建自定义节点）

在 FlowCanvas 中有三种方式创建自定义节点！

* Automaticaly through reflection. 通过自动反射
* Coding Simplex Nodes. 代码编写简单节点
* Coding Full Nodes. 打开吗编写完整节点

6.1 Creating Simplex Nodes（代码编写简单节点）

在 Flowcanvas 创建一个简单节点是很容易的！你只需要一个继承自 node 基类即可！

CallableActionNode

Callable 节点的作用就是可以开启 flow 控制流，之前也介绍过了。这里继承后你只需要覆盖 Invoke 接口即可。模板类型需要统一！例如下面这样：

```
[Category("Actions/Utility")]
public class LogValue : CallableActionNode<object>{
 public override void Invoke(object obj){
  Debug.Log(obj);
 }
}
```

CallableFunctionNode

类似于上面的类型，但是这个是有返回值的！

```
[Category("Examples")]
public class FindTransform : CallableFunctionNode<Transform, Transform, string>{
 public override Transform Invoke(Transform root, string name){
  return root != null? root.Find(name) : null;
 }
}
```

LatentActionNode

类似于上面，只是这里使用的是携程

```
[Category("Actions/Utility")]
public class Wait : LatentActionNode<float>{
 public override IEnumerator Invoke(float time){
  var timer = time;
  while (timer > 0){
   timer -= Time.deltaTime;
   yield return null;
  }
 }
}
```

PureFunctionNode

这个节点表示不需要任何 flow 操作，只是一个 function 节点返回一个值！第一个参数是返回值！

```
[Category("Functions/Math/Floats")]
[Name("!=")]
public class FloatNotEqual : PureFunctionNode<bool, float, float>{
 public override bool Invoke(float a, float b){
  return a != b;
 }
}
```

6.2. Creating Full Nodes

你可以创建任意类型的节点和样式！

我们来尝试重写“SwitchBool”这个节点，首先创建一个类并继承自 FlowControlNode，重写方法 RegisterPorts 并注册进去！注册方法如下：

FlowInput : AddFlowInput(string name, Action pointer)
name: The name of the port.
pointer: A delegate that will be called once this flow input is called.

FlowOutput: AddFlowOutput(string name)
name: The name of the port.
To execute the port, you have to call, Call() on the FlowOutput object returned, along with the current Flow object received.

ValueInput<T> : AddValueInput<T>(string name)
name: The name of the port.
To get the value connected to the port, you simply do so by calling the .value property of the object returned.

ValueOutput<T> : AddValueOutput<T>(string name, Func<T> getter)
name: The name of the port.
getter: A delegate that will be called to get the value of type T.

完整示例代码：

```
using ParadoxNotion.Design;

namespace FlowCanvas.Nodes{

 [Name("Switch Condition")]
 [Category("Flow Controllers/Switchers")]
 [Description("Branch the Flow based on a conditional boolean value")]
 public class SwitchBool : FlowControlNode {
  protected override void RegisterPorts(){
   var condition = AddValueInput<bool>("Condition");
   var trueOut = AddFlowOutput("True");
   var falseOut = AddFlowOutput("False");

   AddFlowInput("In", (f)=>
   {
    if (condition.value){
     trueOut.Call(f);
    } else {
     falseOut.Call(f);
    }
   });
  }
 }
}
```

6.3. Creating Event Nodes

有时你可能需要自定义事件来触发真个后续的 Flow 流！虽然你可以使用 c#事件系统。当你使用 c#事件并通过 CodeEvent 节点进行监听的时候通过反射进行操作，但是有时并不能满足你的需求，这样只能使用 delegate 和 System.Action .因此你可以创建自定义事件节点获取你想要的一切。下面是一个使用静态 c#事件的例子！

```
namespace FlowCanvas.Nodes{

 public class CustomEventNodeExample : EventNode {

  private FlowOutput raised;

  public override void OnGraphStarted(){
   //Subscribe to the event here. For example:
   MyClass.MyEvent += EventRaised;
  }

  public override void OnGraphStoped(){
   //Unsubscribe here. For example:
   MyClass.MyEvent -= EventRaised;
```

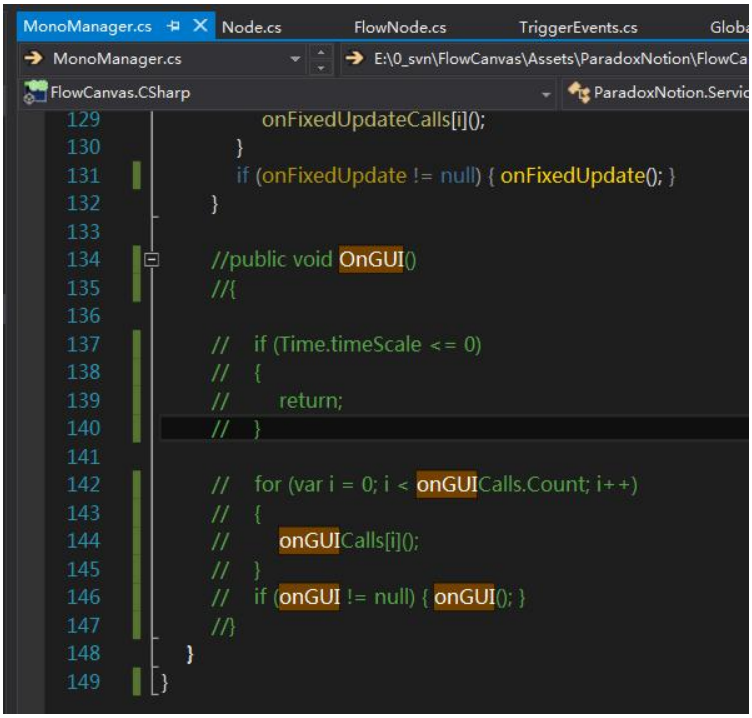
```
    }

    //Register the output flow port or any other port
    protected override void RegisterPorts(){
     raised = AddFlowOutput("Out");
    }

    //Fire output flow
    void EventRaised(){
     raised.Call(new Flow());
    }
   }
  }
```

如果 c#的 event 不是一个静态对象，只是一个 monobehavior 的对象你可以创建一个 EventNode 的通用版本！（泛型版本）在泛型版本中你会获得一个这个类型的引用，类型对象引用通过 target.value 进行传递。target 默认情况下使用的是“self”，当然你也可以从属性面板中进行指定 target！

```
namespace FlowCanvas.Nodes{

 public class CustomEventNodeExample : EventNode<MyComponent> {

  private FlowOutput raised;

  public override void OnGraphStarted(){
   base.OnGraphStarted(); //make sure to call base.
   //Subscribe to the event here. For example:
   target.value.MyEvent += EventRaised;
  }

  public override void OnGraphStoped(){
   base.OnGraphStoped(); //make sure to call base.
   //Unsubscribe here. For example:
   target.value.MyEvent -= EventRaised;
  }

  //Register the output flow port or any other port
  protected override void RegisterPorts(){
   raised = AddFlowOutput("Out");
  }

  //Fire output flow
  void EventRaised(){
   raised.Call(new Flow());
  }
 }
}
```

运行时发现的问题：

1：如下图 MonoManager 中的 OnGUI 不需要，否则每帧 400B 内存开销，节点中也不会使用跟 GUI 相关的类型或接口！



2: SendEvent 的时候因为使用了字符串事件名称，所以会有一定的内存开销！