



使用手册

目录

第一章 对话系统	4
第一节 基本概念	4
1.1 对话	4
1.2 对话结点	4
1.3 对话选项	4
第二节 对话	4
2.1 对话类	4
2.2 对话编辑器	5
第三节 基本结点类型	10
3.1 结点基类	10
3.2 语句结点	11
3.3 选项修饰结点	12
3.4 条件显示结点	13
3.5 拦截结点	14
3.6 后缀结点	15
3.7 外置选项结点	15
3.8 分叉结点	15
3.9 肖像语音覆盖结点	16
第四节 内置常用结点类型	16
4.1 根结点	16
4.2 结束结点	17
4.3 粗体选项结点	18
4.4 染色选项结点	19
4.5 斜体选项结点	19
4.6 按条件显示结点	19
4.7 完成前显示结点	20
4.8 按条件拦截结点	20
4.9 递归结点	21
4.10 随机顺序选项结点	21
4.11 还原选项结点	22
4.12 分支结点	23
4.13 事件结点	24
4.14 其它对话结点	24
4.15 按条件分叉结点	25
4.16 按条件覆盖肖像结点	25
4.17 按条件覆盖语音结点	26
第五节 结点用接口	26
5.1 单个主要选项结点接口	26
5.2 事件结点接口	26
5.3 可结束结点接口	26
5.4 手动处理结点接口	27
第六节 其它重要类型	27

6.1 对话选项	27
6.2 对话数据	27
6.3 对话事件	28
6.4 对话分组	29
6.5 对话管理器	30
6.6 对话处理器	31
6.7 对话人接口	33
第二章 条件系统	34
第一节 基本概念	34
第二节 组成要素	34
2.1 条件	34
2.2 条件组	35
第三章 关键字系统	36
第一节 基本概念	36
第二节 组成要素	36
2.1 关键字接口	36
2.2 关键字类	37
第三节 相关特性	37
3.1 关键字集合特性	37
3.2 关键字获取方法特性	38
3.3 运行时关键字获取方法特性	38
第四章 多语言系统	39
第一节 基本概念	39
第二节 组成要素	39
2.1 翻译映射	39
2.2 翻译包	39
2.3 本地化文件	41
2.4 本地化分组	42
第三节 翻译包的编辑	43
3.1 翻译包编辑器	43
3.2 用 Excel 编辑翻译映射	44
第五节 翻译器	45
第六节 多语言文本组件	46
6.1 静态多语言文本	46
6.2 多语言文本	47
第五章 存档系统	48
第一节 基本概念	48
第二节 组成要素	48
2.1 存档数据	48
2.2 存档管理器	48
第三节 相关特性	48
3.1 存档方法特性	48
3.2 读档方法特性	49
第六章 游戏界面	50

第一节 基本概念	50
第二节 窗口系统	50
2.1 窗口基类	50
2.2 窗口预制件集合	51
2.3 窗口管理器	51
第三节 通用窗口	53
第四节 对话框	54
第七章 交互系统	57
第一节 基本概念	57
第二节 组成要素	57
2.1 可交互接口	57
2.2 基本可交互对象	59
第三节 交互界面	60
3.1 交互面板	60
3.2 交互窗口	61
第八章 快速开始	62
第一节 使用交互系统进行对话	62
1.1 搭建 UI	62
1.2 添加角色	64
1.3 创建和编辑对话	68
第二节 其它方式进行对话	70
2.1 搭建 UI	70
2.2 创建和编辑对话	72
第三节 美化对话框	73
附录	77
编辑器设置	77
类	77
1. 轻量级通用型数据	77
2. 单例资源	79
3. 单例窗口	79
接口	79
1. 玩家名字持有者接口	79
2. 可复制接口	80
3. 可淡入淡出接口	80
4. 场景加载器接口	80
5. 消息显示器接口	81
特性	81
1. 初始化特性	81
2. 初始化方法特性	81

第一章 对话系统

第一节 基本概念

1.1 对话

[对话](#)是一种 ScriptableObject 资源文件，由对话结点和对话选项等基本内容构成。

1.2 对话结点

在对话系统中，对话结点是一个对话的基本构成要素。对话结点的基类是 [DialogueNode](#)，这是一个抽象类，需要用户自行继承并实现自己想要的结点功能，当然，本插件也准备了拥有基本功能的内置结点，比如[语句结点](#)，用于填写对话，又如[条件结点](#)，决定对话选项是否显示，等等。这些内容会在下文进行详细展开，此处仅做提及。

1.3 对话选项

既然上面提到了对话选项，这里就概述一下。对话选项，顾名思义，就是对话过程中向玩家弹出的选择项。选项又分为主要选项和普通选项，主要选项不是实际存在的选项（即不显示在 UI 上），用于连接下一句话，假设本句话没有需要显示的选项，但又要可以进入下一句，我们就需要一个中介连接本句和下一句，它就是主要选项；主要选项还可以用来连接一些特殊结点，比如[结束结点](#)，用于标志对话要在哪里结束，在哪里关闭[对话窗口](#)。普通选项则是显示在 UI 上面的选项，它们可被上文提到的条件结点修饰可见性。

第二节 对话

2.1 对话类

对话类是 Dialogue，继承自 ScriptableObject，它的成员如下：

运行时成员 (非方法成员仅显示名称)	说明
属性 ID	对话的唯一标识码，引用的是其 根结点 的 ID
字段 nodes	对话的所有结点，包括遍历不到的结点
属性 Nodes	nodes 的关联只读属性
属性 Entry	对话的根结点
属性 Exitable	用于判断对话是否有结束结点
构造函数 DialogueNode()	自动生成一个根结点，并加入 nodes
方法 bool Reachable(DialogueNode)	判断从根结点开始可否到达指定结点
静态方法 bool Reachable (DialogueNode, DialogueNode)	判断从指定结点开始可否到达另一个指定结点
静态方法 void Traverse (DialogueNode, Action<DialogueNode>)	从指定结点开始遍历它和它的子结点
静态方法 void Traverse (DialogueNode, Func<DialogueNode, bool>)	从指定结点开始遍历它和它的子结点，可被中断，返回值表示是否发生中断

由编辑器使用的成员如下：

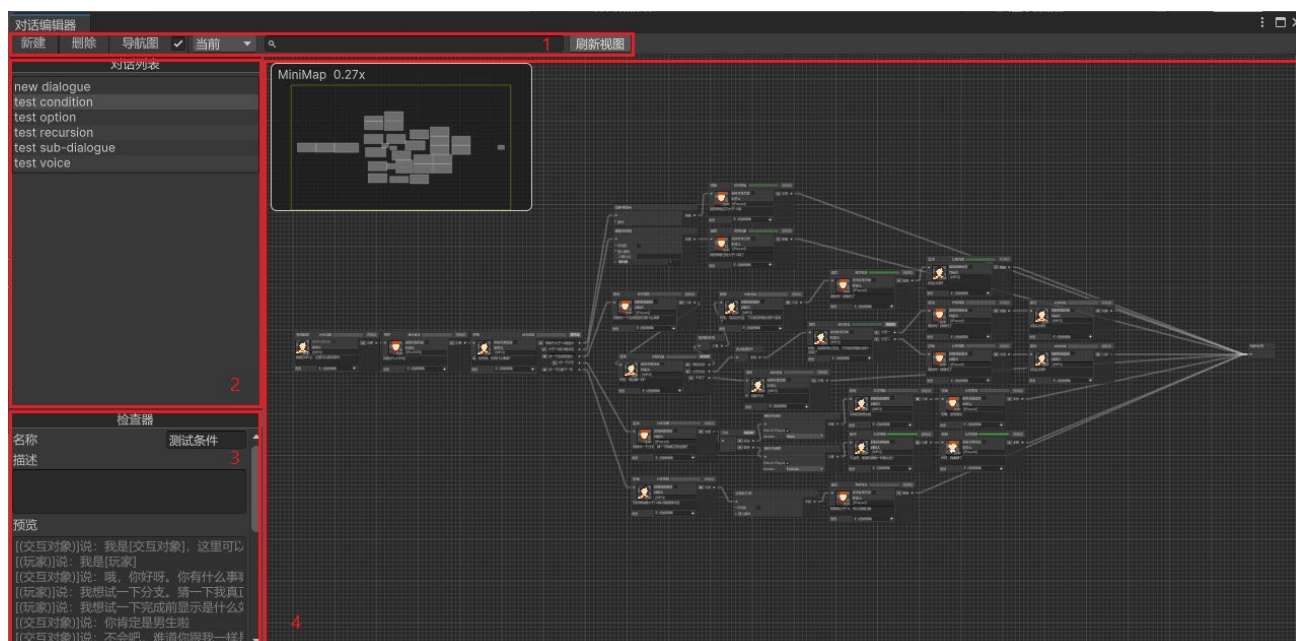
编辑器成员	说明
字段 <code>_exit</code>	对话的结束结点，无运行时功能，仅为了方便在编辑器中设置结点的 <code>exitHere</code> 属性
字段 <code>_name</code>	对话的名称
字段 <code>_description</code>	对话的描述，用于在编辑器中备注对话的用处
字段 <code>_groups</code>	对话中结点的分组，无运行时功能，仅为了方便在编辑器中成组管理结点

对话类还有一个供编辑器使用的内嵌静态类 `Editor`，它的成员如下：

编辑器成员	说明
静态方法 <code>DialogueNode AddNode (Dialogue, Type)</code>	在对话中新增一个指定类型的结点
静态方法 <code>void AddCopiedNode (Dialogue, DialogueNode)</code>	往对话中加入复制的结点
静态方法 <code>void RemoveNode (Dialogue, DialogueNode)</code>	从对话中移除指定结点

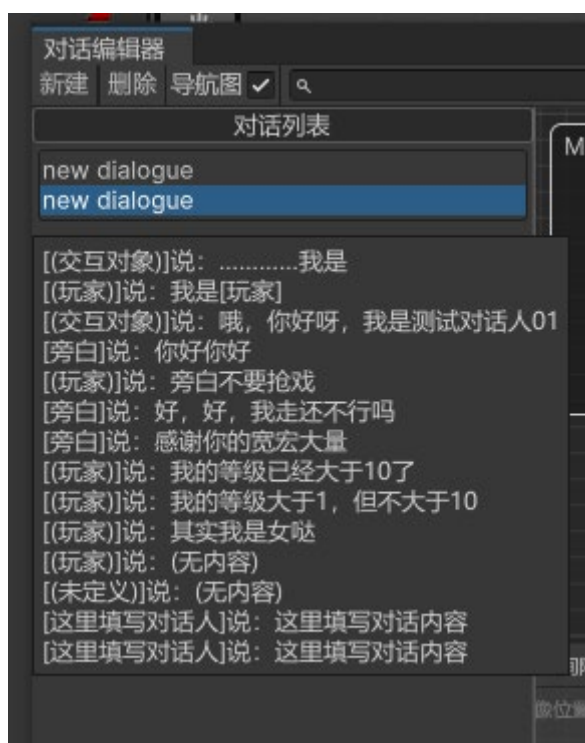
2.2 对话编辑器

对话编辑器是一个独立的编辑器窗口，它放弃传统的检查器（Inspector）形式，对对话资源文件进行图形化的结点到结点的连线编辑，如图所示：



区域 1 是工具栏区域，点击“新建”按钮可在指定路径新建一个对话，“删除”按钮可以删除选中的对话，“导航图”复选框可以切换区域 4 中左上角小地图的显示隐藏。搜索框可选择全局和当前局部搜索，当进行全局搜索时，会查找对话 ID、名称、描述以及结点的 ID、对话人、对话内容、选项标题、结点类型（在关键字前加“t:”前缀），进行局部搜索时，只会查找后面五种；全局搜索前三种将定位到对话，搜索后五种和局部搜索将定位到结点。要注意的是，无法检索富文本标签，进行搜索时富文本标签会被忽略。

区域 2 是项目中所有的对话资源文件（.asset）的列表，左键单击列表项可选中相应的对话，同时刷新区域 4 为选中对话的视图，而右键单击列表项则可以删除、定位或者重命名对话资源文件；将鼠标指针悬停在列表项上可预览对话中的语句结点，如图所示：

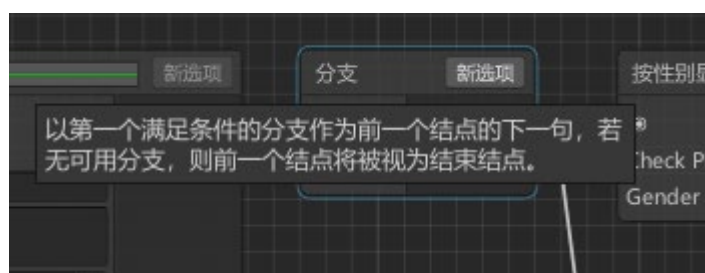


区域 3 是检查器区域，未选中结点时，显示的是对话的检查器，选中结点时，显示的是结点的检查器，可对结点进行更深入的设置（结点的一些字段不会显示在图形结点上面）。

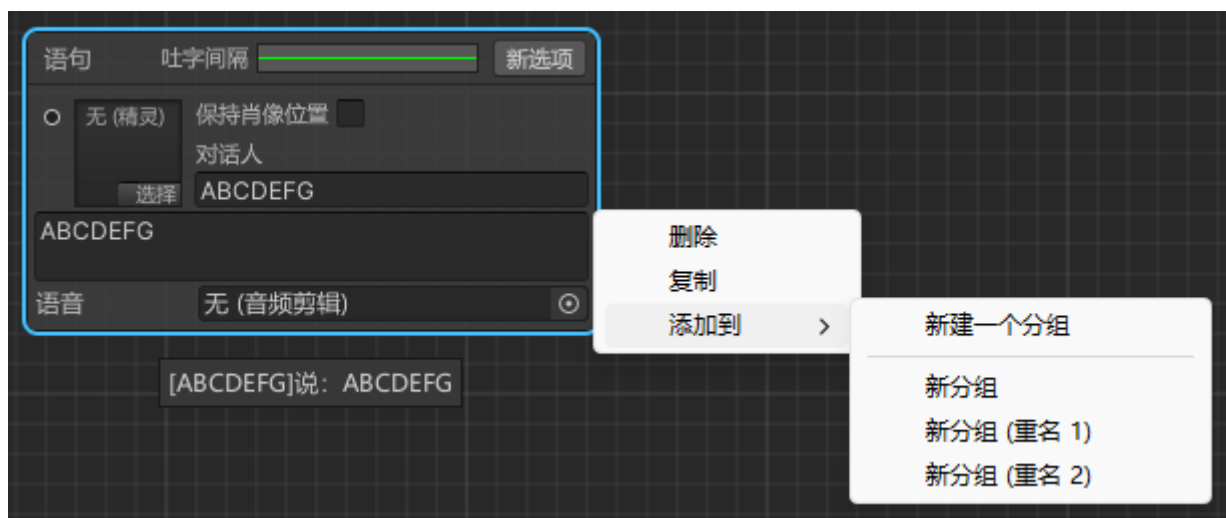
区域 4 则是对话编辑视图，以结点直接连线的方式构建对话。按住鼠标左键并移动鼠标可进行框选，按住鼠标中键可拖拽视图，鼠标滚轮可进行缩放，右键点击空白处则会弹出结点创建菜单，如图所示：



点击可创建相应的内容，其中的“粘贴”菜单项在复制结点后才会出现。把鼠标指针悬停在结点左上角的名称上可查看结点的功能描述，如图所示：




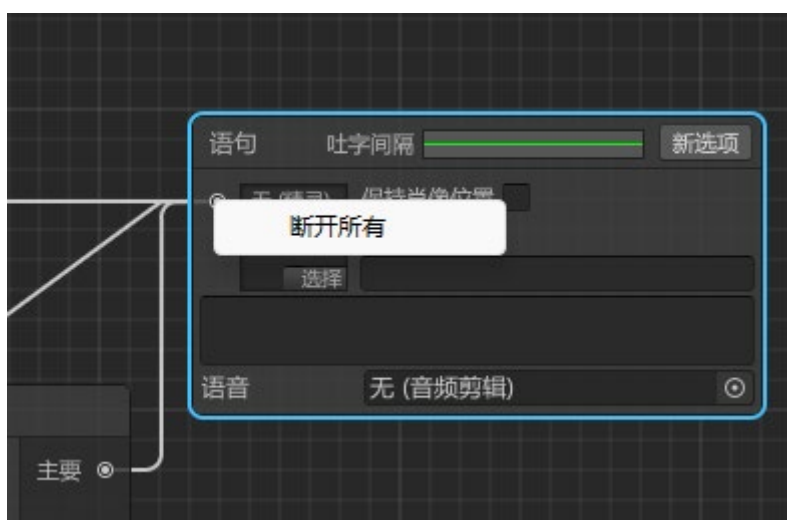
右键点击结点可对结点进行分组、复制、删除等操作，如图所示：




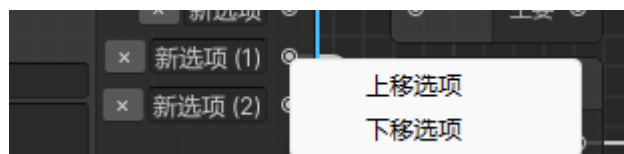
对于语句结点，把鼠标指针放在结点上，可在结点下方预览结点的文本，其中的[关键字](#) ID 字符串将被转换为相应的名称，如图所示：



可以使用快捷键“Ctrl + C”、“Ctrl + V”、“Ctrl + X”、“Delete”来复制、粘贴、删除结点，还可以使用“Ctrl + Z”、“Ctrl + Y”或者 Unity 右上角的  按钮对所做的修改进行撤销、重做。其中，粘贴结点时，将以鼠标指针所在的位置为中心放置粘贴的结点；
右键单击结点的入口端可以快速断开与之相连的所有连线，如图所示：



单击结点选项（即出口端）左侧的  按钮可以删除选项，右键单击选项则可以移动它的位置，如图所示：



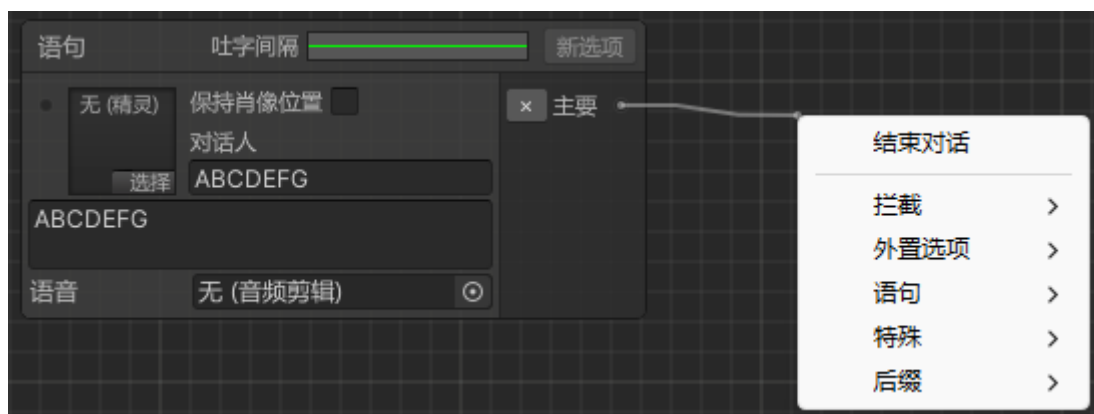
这里主要选项和普通选项的区别就是，主要选项的标题“主要”不可编辑，而普通选项的标题是一个输入框，可被同步修改，若此输入框中没有被选中的文字，还可以单击右键来快速插入关键字，如图所示：

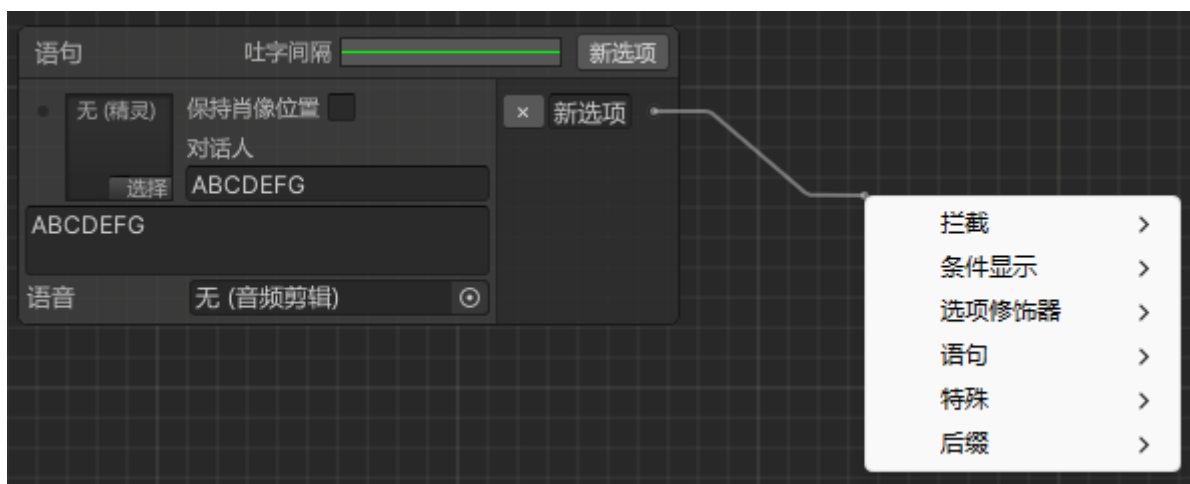


对于语句结点、[其它对话结点](#)，若它们的主要选项可被普通选项替代，则右键单击主要选项可将其转为普通选项，同理，只有单个普通选项时，也可以把它转为主要选项，如图所示：



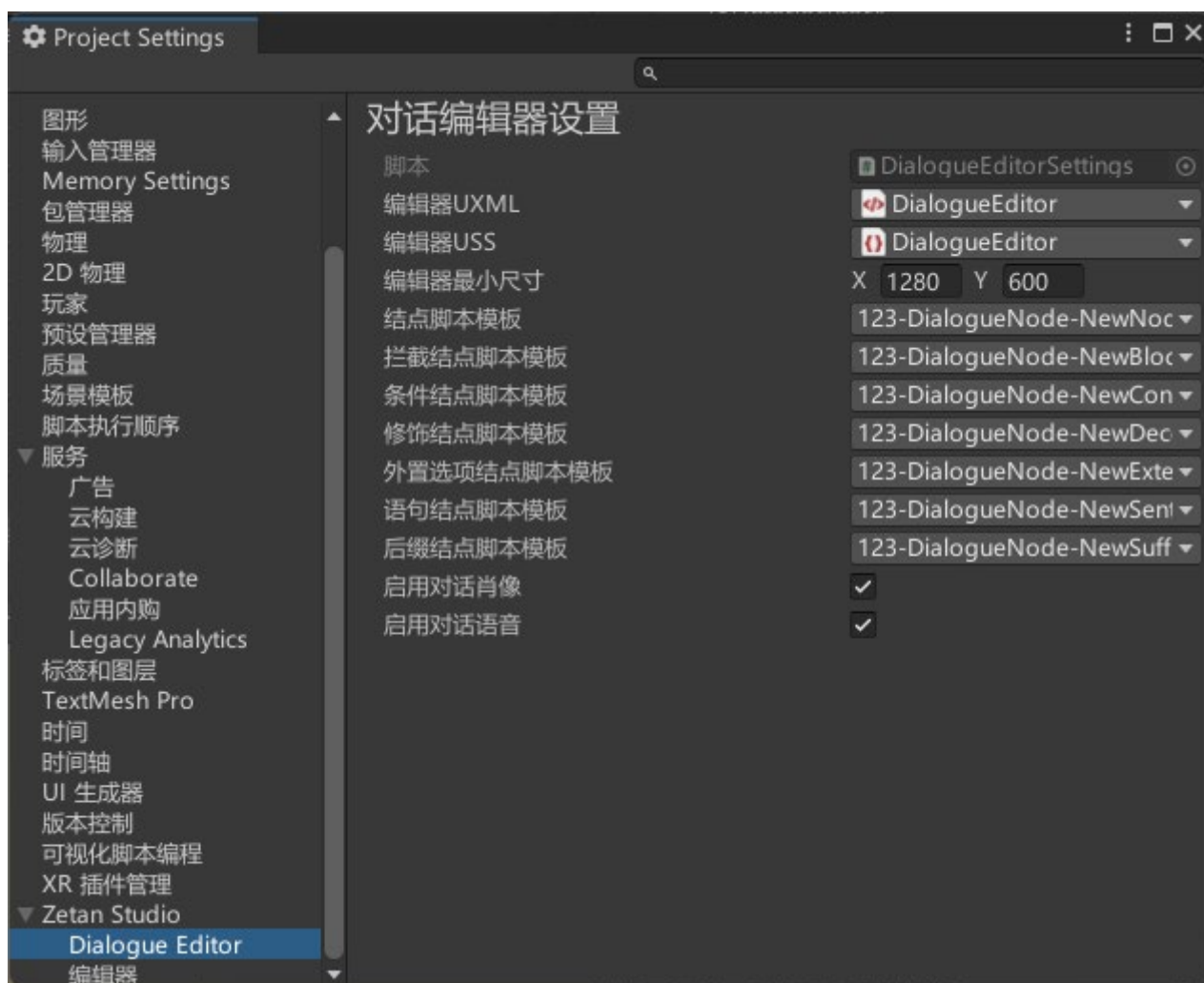
拖拽选项的连线到空白处释放鼠标，可弹出快速结点菜单，这个菜单罗列出可与当前选项连接的结点类型，点击后可快速创建，如图所示：





其中，“结束对话”可快速将选项连接到结束结点，并非创建一个结束结点。

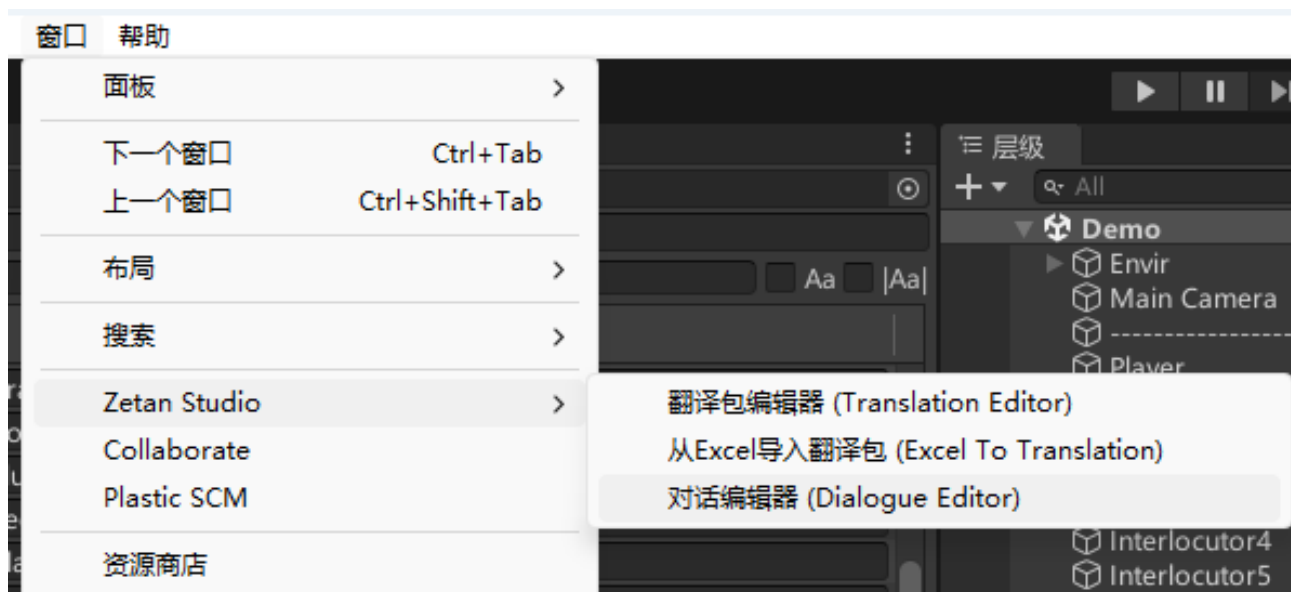
对话编辑器可在“编辑→项目设置...→Zetan Studio→对话编辑器”中进行设置，如图所示：



其中，请不要删除 UXML 和 USS 文件，这会导致对话编辑器无法使用；“启用对话肖像”和“启用对话语音”是默认勾选的，如果不需要这两个功能，请自行取消勾选，取消勾选后会重新编译项目代码，请耐心等待完成，不要进行反复勾选和取消勾选等多余操作。

如何打开对话编辑器：

双击对话资源文件，或点击它的检查器右上角的打开按钮，或从 Unity 上方的工具栏中打开，如图所示：



第三节 基本结点类型

3.1 结点基类

顾名思义，结点基类是所有结点类型的基类，上文也提到过，类型是 `DialogueNode`，它的成员如下：

运行时成员	说明
属性 ID	结点的唯一标识码，自动生成，也可以手动填写
字段 options	结点的内部选项数组，用来存储结点选项
属性 Options	options 的关联只读属性
字段 exitHere	用于标识对话是否在此结点结束内部字段
属性 ExitHere	exitHere 的关联只读属性，返回 True 的前提是结点必须继承接口 IExitableNode
属性 Exitable	用于标识从本结点出发，是否可以到达结束结点
属性 IsValid	用于检查结点的填写是否有效，否则结点不会参与到对话中
索引器 DialogueOption this[int index]	索引结点选项数组的一个捷径，是只读的
操作符 bool(DialogueNode)	用于对结点进行快速判空

由编辑器使用的成员如下：

编辑器成员	说明
字段 _position	用于在编辑器中记录结点位置
方法 string GetName()	获取此结点类型的名称（不是类型名称）
方法 bool CanConnectFrom (DialogueNode, DialogueOption)	用于判断其它某个结点的某个选项能否连接到此结点，可与下一个方法搭配使用
方法 bool CanConnectTo (DialogueNode, DialogueOption)	用于判断该结点的某个选项能否连接到其它某个结点，可与上一个方法搭配使用
方法 HashSet<string> GetHiddenFields()	用于在编辑器结点中筛选要显示的字段
方法 DialogueNode Copy()	用于在编辑器中复制结点

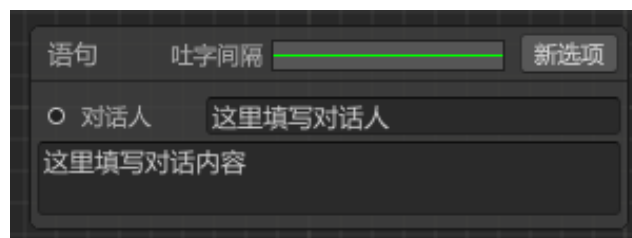
结点基类还有一个供编辑器使用的内嵌静态类 `Editor`，它的成员如下：

编辑器成员	说明
静态方法 string GetGroup(Type)	获取某类结点的分组
静态方法 string GetName(Type)	获取某类结点的名称
静态方法 string GetDescription(Type)	获取某类结点的描述
静态方法 float GetWidth(Type)	获取某类结点的图形结点宽度
静态方法 DialogueOption AddOption (DialogueNode, bool, string)	给结点增加选项
静态方法 void RemoveOption (DialogueNode, DialogueOption)	从结点中移除选项
静态方法 void MoveOptionUpward (DialogueNode, int)	将结点的指定选项上移一位
静态方法 void MoveOptionDownward (DialogueNode, int)	将结点的指定选项下移一位
静态方法 void SetAsExit(DialogueNode, bool)	将结点设置为结束结点

3.2 语句结点

语句结点是一个对话中最基本的单元，用于填写对话人和对话内容，以便在 UI 上面显示。它在对话编辑器中的图形结点如图所示：

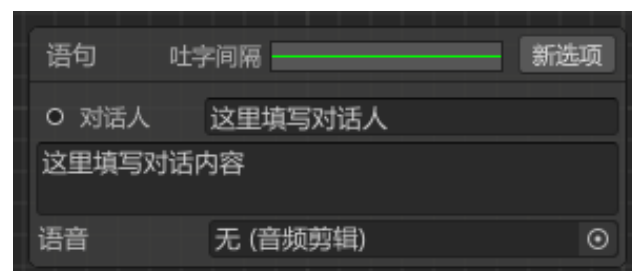
基本状态：



当启用了肖像功能：



当启用了语音功能：



当启用了肖像和语音功能：



语句结点类是 `SentenceNode`，继承自 `DialogueNode`，[IExitableNode](#)，[IEventNode](#)，它的成员如下：

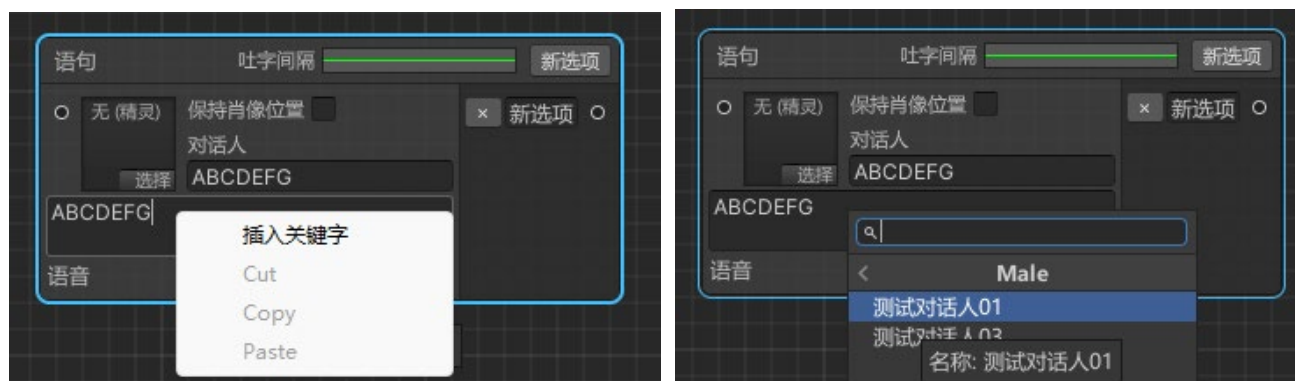
运行时成员	说明
属性 <code>Interlocutor</code>	对话人的名称
属性 <code>Content</code>	对话的内容
属性 <code>SpeakInterval</code>	吐字间隔，是对话窗口中播放打字动画时对话文本字符显示的间隔时间
属性 <code>Portrait</code>	对话人的肖像，当启用肖像功能时可用
属性 <code>KeepPortraitSide</code>	标志是否保持肖像位置，当启用肖像功能时可用。
属性 <code>Voice</code>	对话窗口显示本句话时要播放的语音，当启用语音功能时可用
属性 <code>VoiceOffset</code>	指定语音播放时从第几秒开始，当启用语音功能时可用
字段 <code>events</code>	本句话要触发的事件
属性 <code>Events</code>	<code>events</code> 的关联只读属性
属性 <code>IsValid</code>	当对话人、对话内容非空且对话事件都有效时，此语句结点的填写有效

对于属性 `KeepPortraitSide`，在对话窗口中设置肖像时，会先与上一句话对比对话人名称是否不同，然后自动选择是显示在左侧还是右侧，以形成对立效果，名字相同时位置则不变。如果对话人名称会因为某些原因而改变，但又不想改变其肖像位置，可以设置此属性为 `True`。

如何使用：

除分支结点外，其它类型结点的选项都可以直接连接到语句结点，形成下一句。

在对话人和对话内容的输入框内，若未选中文字，可单击右键打开关键字对象选择窗口，选中后会插入它的 ID 字符串，如图所示：



3.3 选项修饰结点

选项修饰结点是用来在对话窗口中用富文本修饰选项标题文本的，例如对选项的标题文本进行加粗、斜体、

随机颜色等。

选项修饰结点类是 `DecoratorNode`，继承自 `DialogueNode`，[ISoloMainOptionNode](#)，它的成员如下：

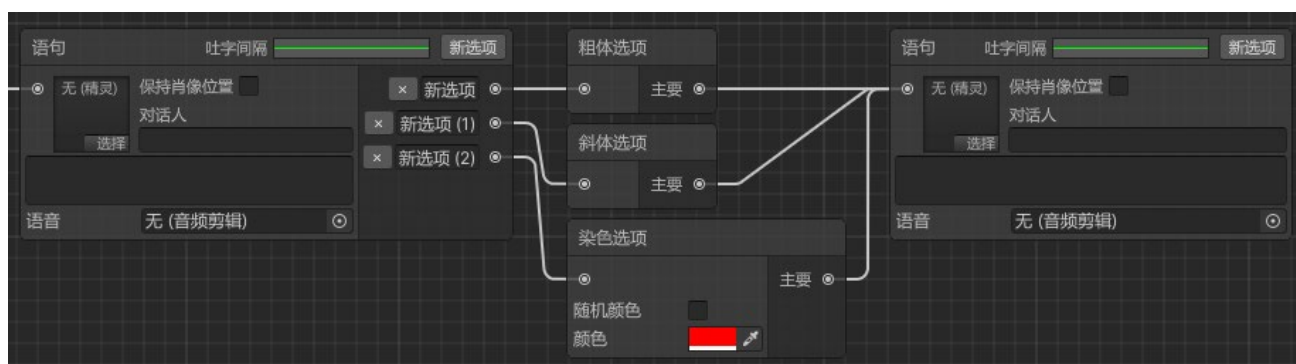
运行时成员	说明
构造函数 <code>DecoratorNode()</code>	自动生成一个主要选项
方法 <code>void Decorate(DialogueData, ref string)</code>	修饰选项标题文本的方法

由编辑器使用的成员如下：

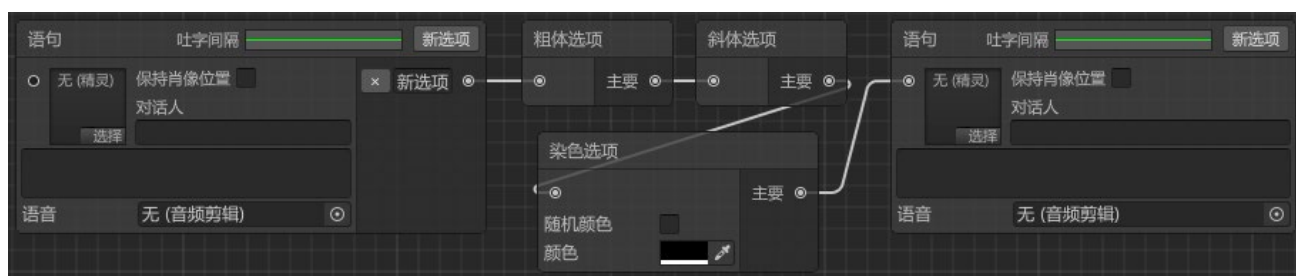
编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有普通选项或修饰结点可以连接到此类结点

如何使用：

将普通选项连接到对应的修饰结点即可，如图所示：



修饰结点之间还可以串联，形成混合修饰，如图所示：



3.4 条件显示结点

条件显示结点用于决定从它开始的分支是否要显示出来，当连接它的是选项时，会影响选项在对话窗口中是否显示出来，即是否剔除它；当连接它的是分支结点时，会影响出现的下一句话。

条件显示结点类是 `ConditionNode`，继承自 `DialogueNode`，[ISoloMainOptionNode](#)，它的成员如下：

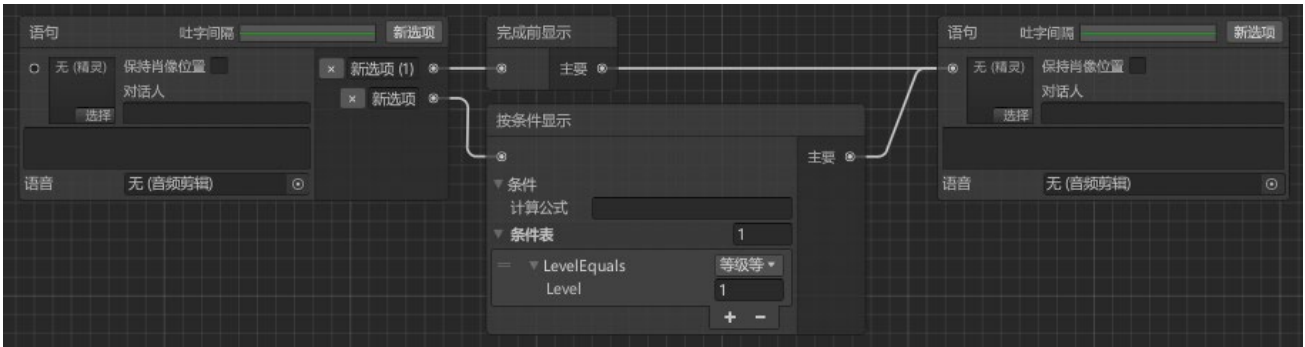
运行时成员	说明
构造函数 <code>ConditionNode()</code>	自动生成一个主要选项
方法 <code>bool Check(DialogueData)</code>	检查条件看是否符合，会一并判断串联的条件显示结点
方法 <code>bool CheckCondition(DialogueData)</code>	供子类重写的内部条件检查虚方法，由上一个方法使用

由编辑器使用的成员如下：

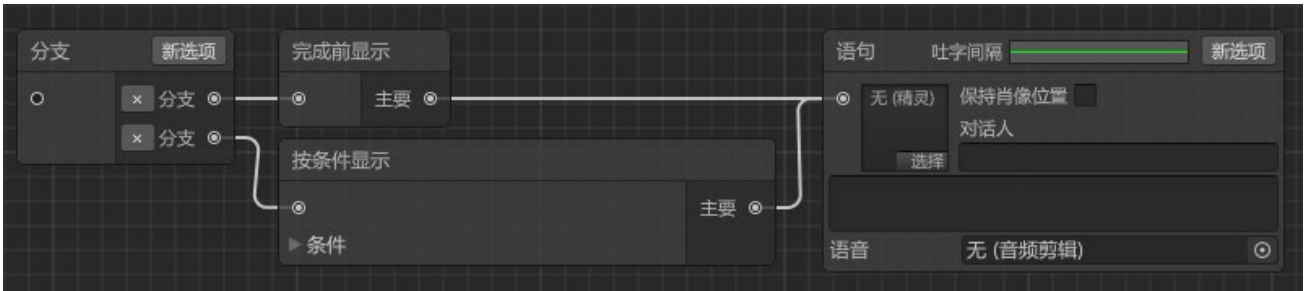
编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有普通选项或条件显示结点、分支结点可以连接到此类结点

如何使用：

将普通选项连接到对应的条件显示结点即可，如图所示：



还可以用分支结点的选项连接它们，如图所示：



当然，它们也可以像选项修饰结点那样串联起来，形成混合条件，如图所示：



要注意的是，当所有选项都能到达结束结点且都被剔除了，则会强制保留第一个选项来显示。

3.5 拦截结点

拦截结点用于判断在点击选项时对话窗口是否要切换到选项所连接的下一句，比如玩家持有某种道具时，才可以进入指定选项所在的分支。

拦截结点类是 `BlockNode`，继承自 `DialogueNode`，`IsoloMainOptionNode`，它的成员如下：

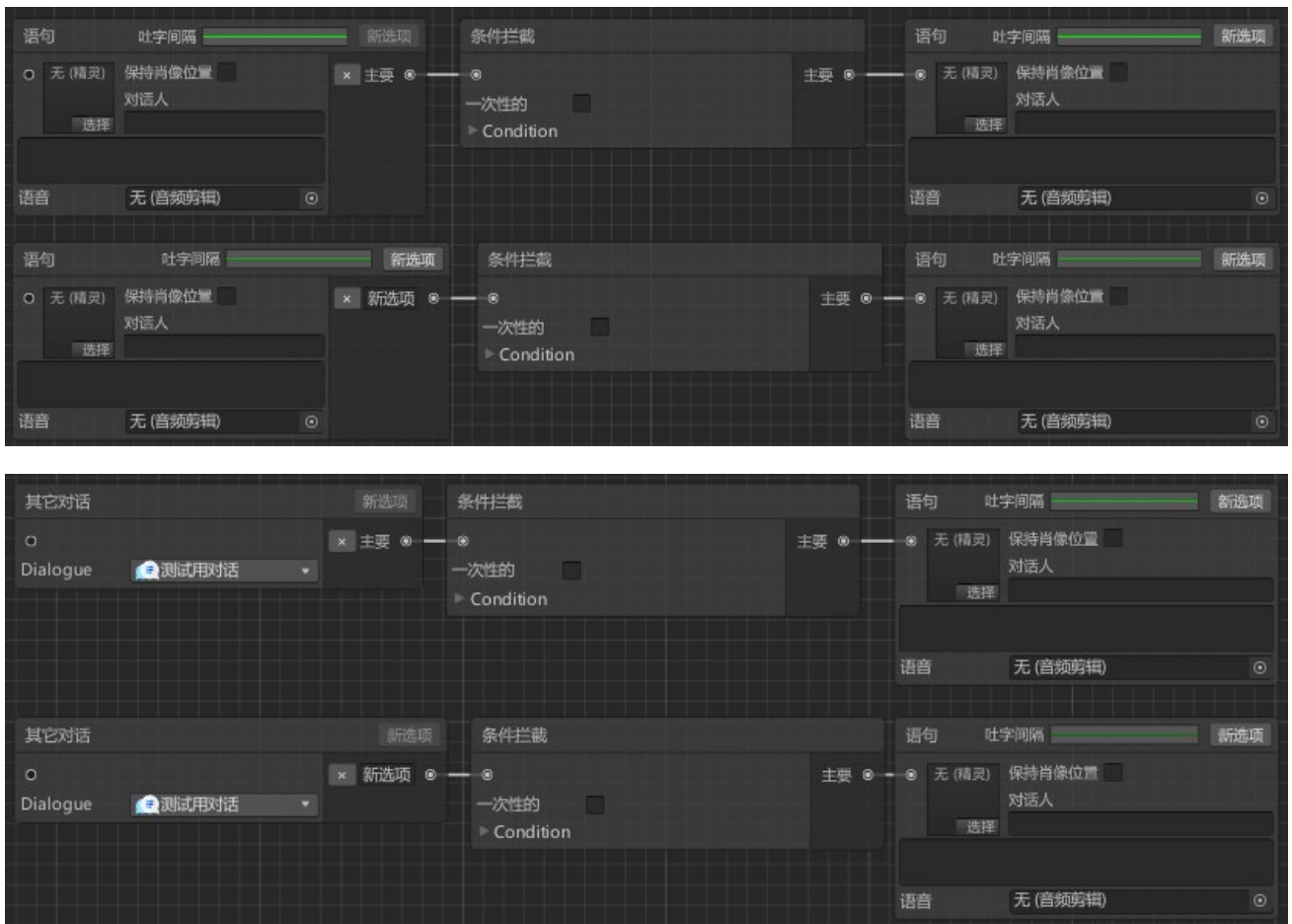
运行时成员	说明
属性 <code>OneTime</code>	是否让此条件结点在满足一次进入条件后，就不再拦截，即使这次条件不满足
构造函数 <code>BlockNode()</code>	自动生成一个主要选项
方法 <code>bool CanEnter(DialogueData, out string)</code>	检查是否能进入由此结点开始的分支
方法 <code>bool CheckCondition()</code>	检查可进入条件，返回 <code>True</code> 表示可进入
方法 <code>string GetBlockReason()</code>	获取为什么被拦截的通知

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有语句结点、其它对话结点和拦截结点可以连接到此类结点

如何使用：

把语句结点或其它对话结点的选项连接到对应的拦截结点即可，如图所示：



拦截结点也可以像条件显示结点那样串联起来，这里不再赘述。

3.6 后缀结点

后缀结点是没有选项的一类特殊结点。后缀结点类为 `SuffixNode`，继承自 `DialogueNode`。

3.7 外置选项结点

外置选项结点可以把上一个结点的普通选项外置在它这里，以实现额外的自定义的选项显示功能，比如按随机顺序显示等。

外置选项结点类是 `ExternalOptionsNode`，继承自 `DialogueNode`，它的成员如下：

运行时成员	说明
方法 <code>ReadOnlyCollection<DialogueOption> GetOptions(DialogueData, DialogueNode)</code>	获取实际的选项，如被打乱后的选项

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有语句结点、其它对话结点的主要选项可以连接到此类结点

3.8 分叉结点

分叉结点通常用于叉开对话流程，它只有两个固定的主要选项用来做两个分支。它也没有具体功能，它的分

支用来做什么完全看用户需求。

分叉结点类是 `BifurcationNode`，继承自 `DialogueNode`，它的成员如下：

运行时成员	说明
属性 <code>First</code>	第一个分支的起始结点
属性 <code>Second</code>	第二个分支的起始结点
构造函数 <code>BifurcationNode()</code>	自动生成两个主要选项
方法 <code>bool CheckIsDone(DialogueData)</code>	由于它的功能由用户定义，所以其完成状态也是

3.9 肖像语音覆盖结点

肖像语音覆盖结点可以将和它们连通的上一个最近的语句结点的肖像或语音替换成指定的肖像或语音，可用于根据玩家性别替换肖像和语音等功能。肖像语音覆盖结点之间也可以串联，当相互串联时，会从使用从左往右数第一个其覆盖用的肖像或语音非空的覆盖结点来进行覆盖。

肖像语音覆盖结点类是 `PortraitVoiceOverrideNode`，继承自 `DialogueNode`，`ISoloMainOptionNode`，它的成员如下：

运行时成员	说明
构造函数 <code>PortraitVoiceOverrideNode()</code>	自动生成一个主要选项

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有语句结点、肖像语音覆盖结点的主要选项可以连接到此类结点
方法 <code>bool CanConnectTo(DialogueNode, DialogueOption)</code>	用于规定此类结点只能用主选项连接到语句结点、肖像语音覆盖结点和 还原选项结点

肖像语音覆盖结点又可细分为肖像覆盖结点和语音覆盖结点，分别比它多了一个获取覆盖肖像和覆盖语音的方法。

肖像覆盖结点类是 `PortraitOverride`，继承自 `PortraitVoiceOverrideNode`，`IExitableNode`，它的成员如下：

运行时成员	说明
方法 <code>Sprite GetPortrait(DialogueData)</code>	获取覆盖肖像的抽象方法

语音覆盖结点类是 `VoiceOverride`，继承自 `PortraitVoiceOverrideNode`，它的成员如下：

运行时成员	说明
方法 <code>AudioClip GetVoice(DialogueData)</code>	获取覆盖语音的抽象方法

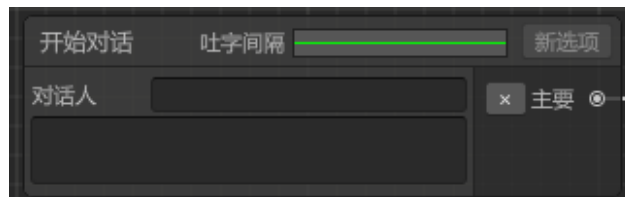
如何使用：

将语句结点的主选项连接到对应的覆盖结点即可。因为被覆盖的语句结点需要使用主选项来连接覆盖结点，所以当被覆盖的语句结点需要有普通选项的时候，可将尾部的覆盖结点连接至还原选项结点来设置还原语句结点应有的选项。

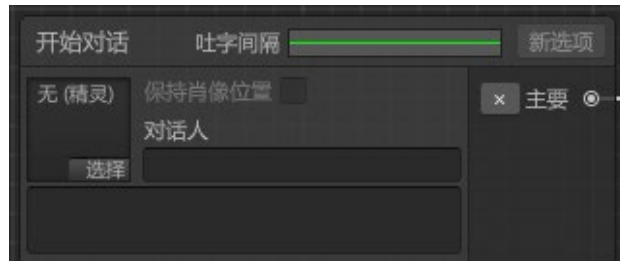
第四节 内置常用结点类型

4.1 根结点

根结点是一种语句结点，它是一个对话的初始结点，应该有且只能有一个，而且没有前置结点。对话窗口将从此类结点开始遍历处理对话结点，哪怕是通过传入对话人和对话内容字符串发起的临时对话，也会生成一个临时的根结点，所以，根结点默认将 `exitHere` 设置为 `True`。它在对话编辑器中的图形结点如图所示：基本状态：



当启用了肖像功能：



当启用了语音功能：



当启用了肖像和语音功能：



根结点类是 `EntryNode`，继承自 `SentenceNode`，它的成员如下：

运行时成员	说明
属性 <code>Interruptable</code>	用于标识是否可以关闭对话窗口以提前结束对话
构造函数 <code>EntryNode()</code>	赋予一个特殊 ID，还会自动生成一个主要选项，且默认把自身设置为结束结点
构造函数 <code>EntryNode(string, string, bool)</code>	在上一个构造函数的基础上设置对话人、对话内容以及 <code>Interruptable</code> 属性
构造函数 <code>EntryNode(string, string, string, bool)</code>	在上一个构造函数的基础上设置自定义的 ID

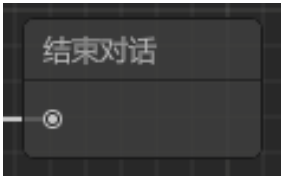
由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定任何结点都无法连接到此类结点

4.2 结束结点

结束结点是一种后缀结点，用于在编辑器中设置结束结点，仅存在于编辑器相关代码中，当游戏编译发布

后，将无法找到，所以请勿在游戏逻辑中使用。它在对话编辑器中的图形结点如图所示：

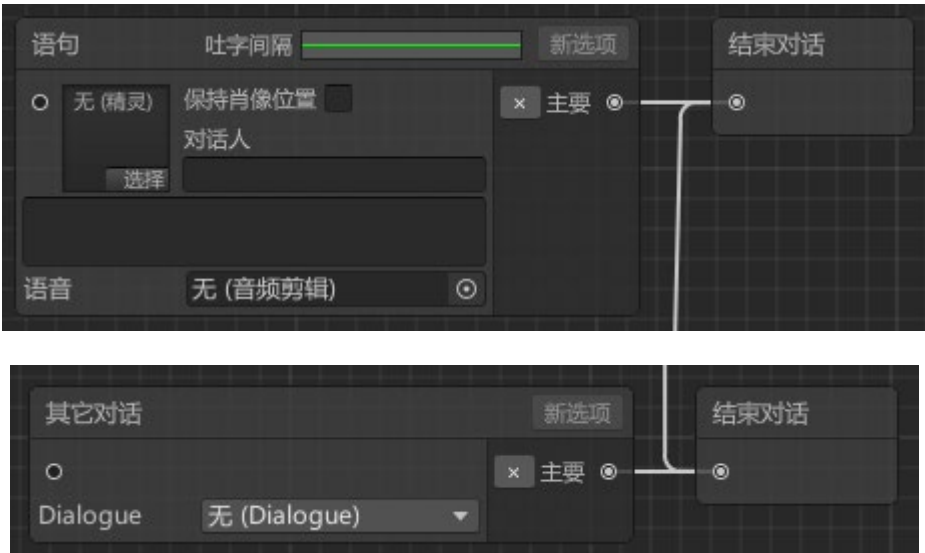


结束结点类是 `ExitNode`，继承自 `SuffixNode`，它由编辑器使用的成员如下：

编辑器成员	说明
属性 <code>IsValid</code>	返回 <code>True</code>
构造函数 <code>ExitNode()</code>	自动设置一个指定的结点位置，以免创建新对话时和根结点叠在一起
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有继承了 <code>IExitableNode</code> 接口的结点的主要选项才能连接到此类结点

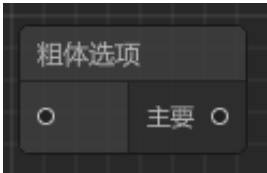
如何使用：

把继承自 `IExitableNode` 的结点的主要选项连接至结束结点即可，如图所示：



4.3 粗体选项结点

粗体选项结点是一种选项修饰结点，会以粗体显示选项的标题文本。它在对话编辑器中的图形结点如图所示：



粗体选项结点类是 `BoldDecorator`，继承自 `DecoratorNode`，它的成员如下：

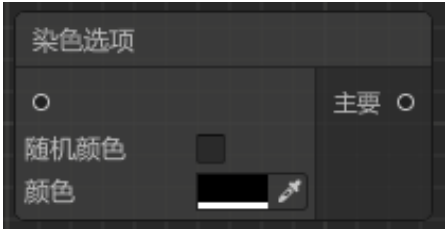
运行时成员	说明
属性 <code>IsValid</code>	返回 <code>True</code>
方法 <code>void Decorate(DialogueData, ref string)</code>	传出使用粗体富文本修饰过的选项标题字符串

使用方法见选项修饰结点类。

4.4 染色选项结点

染色选项结点是一种选项修饰结点，会以给定的颜色或一定范围内的随机颜色显示选项的标题文本。它在对话编辑器中的图形结点如图所示：

当未勾选随机颜色：



当勾选了随机颜色：



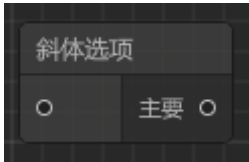
染色选项结点类是 `ColorfulDecorator`，继承自 `DecoratorNode`，它的成员如下：

运行时成员	说明
属性 <code>RandomColor</code>	是否使用一定范围内的随机颜色
属性 <code>Gradient</code>	富文本的随机颜色范围，当 <code>RandomColor</code> 为 <code>True</code> 时可见
属性 <code>Color</code>	富文本的颜色，让 <code>RandomColor</code> 为 <code>False</code> 时可见
属性 <code>IsValid</code>	如果使用随机颜色或非全透明颜色，返回 <code>True</code>
方法 <code>void Decorate(DialogueData, ref string)</code>	传出使用颜色富文本修饰过的选项标题字符串

使用方法见选项修饰结点类。

4.5 斜体选项结点

斜体选项结点是一种选项修饰结点，会以粗体显示选项的标题文本。它在对话编辑器中的图形结点如图所示：



斜体选项结点类是 `ItalicDecorator`，继承自 `DecoratorNode`，它的成员如下：

运行时成员	说明
属性 <code>IsValid</code>	返回 <code>True</code>
方法 <code>void Decorate(DialogueData, ref string)</code>	传出使用斜体富文本修饰过的选项标题字符串

使用方法见选项修饰结点类。

4.6 按条件显示结点

按条件显示结点是一种条件显示结点，可以按条件显示分支。它在对话编辑器中的图形结点如图所示：



按条件显示结点类是 `CommonCondition`，继承自 `ConditionNode`，它的成员如下：

运行时成员	说明
属性 <code>Condition</code>	需要满足的条件，类型是 ConditionGroup
属性 <code>IsValid</code>	<code>Condition</code> 填写有效时返回 <code>True</code>
方法 <code>bool CheckCondition(DialogueData)</code>	<code>Condition</code> 满足时返回 <code>True</code>

使用方法见条件显示结点类以及条件组类。

4.7 完成前显示结点

完成前显示结点是一种条件显示结点，可在所在分支完成前（[如何算完成？](#)）显示所在分支。它在对话编辑器中的图形结点如图所示：



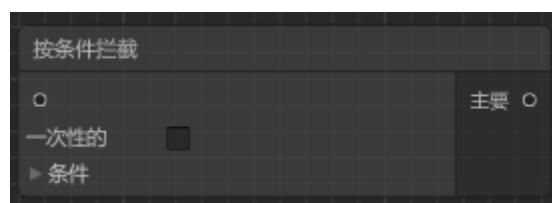
完成前显示结点类是 `DeleteOnDoneCondition`，继承自 `ConditionNode`，它的成员如下：

运行时成员	说明
属性 <code>IsValid</code>	返回 <code>True</code>
方法 <code>bool CheckCondition(DialogueData)</code>	它的主要选项所连接的结点完成时返回 <code>True</code>

使用方法见条件显示结点类。

4.8 按条件拦截结点

按条件拦截结点是一种拦截结点，它要求满足指定条件时才可进入从它开始的分支。它在对话编辑器中的图形结点如图所示：



按条件拦截结点类是 `ConditionalBlock`，继承自 `BlockNode`，它的成员如下：

运行时成员	说明
属性 <code>BlockReason</code>	被拦截时的提示语

属性 Condition	需要满足的进入条件，类型是 ConditionGroup
属性 IsValid	Condition 填写有效时返回 True
方法 bool CheckCondition()	Condition 满足时返回 True
方法 string GetBlockReason()	返回拦截提示语以提示玩家

使用方法见拦截结点类。

4.9 递归结点

递归结点是一种后缀结点，它也叫做返回结点，用于返回到前面指定的语句结点，由于返回后可能在某一时刻又返回到此结点，所以叫做递归结点。它在对话编辑器中的图形结点如图所示：



递归结点类是 RecursionSuffix，继承自 SuffixNode，[IManualNode](#)，它的成员如下：

运行时成员	说明
属性 Depth	返回深度，表示要返回到的是往前数第几个语句结点
属性 IsValid	返回深度大于 1 时返回 True
方法 DialogueNode FindRecursionPoint (EntryNode)	查找需要返回到的语句结点，深度至多到根结点
方法 void DoManual(DialogueWindow)	用上一个方法查找到的结点继续对话

由编辑器使用的成员如下：

编辑器成员	说明
方法 bool CanConnectFrom (DialogueNode, DialogueOption)	用于规定只有根结点以外的语句结点可以连接到此类结点

如何使用：

将语句结点的选项连接到对应的递归结点即可，如图所示：



注意，递归结点和要返回的语句结点之间至少要隔一个语句结点，即连接到递归结点的选项所在的语句结点不能作为要返回的结点，右键点击递归结点可定位返回到的结点。

4.10 随机顺序选项结点

随机顺序选项结点是一种外置选项结点，可以按随机顺序显示上一个结点外置在它这里的选项。它在对话编辑器中的图形结点如图所示：

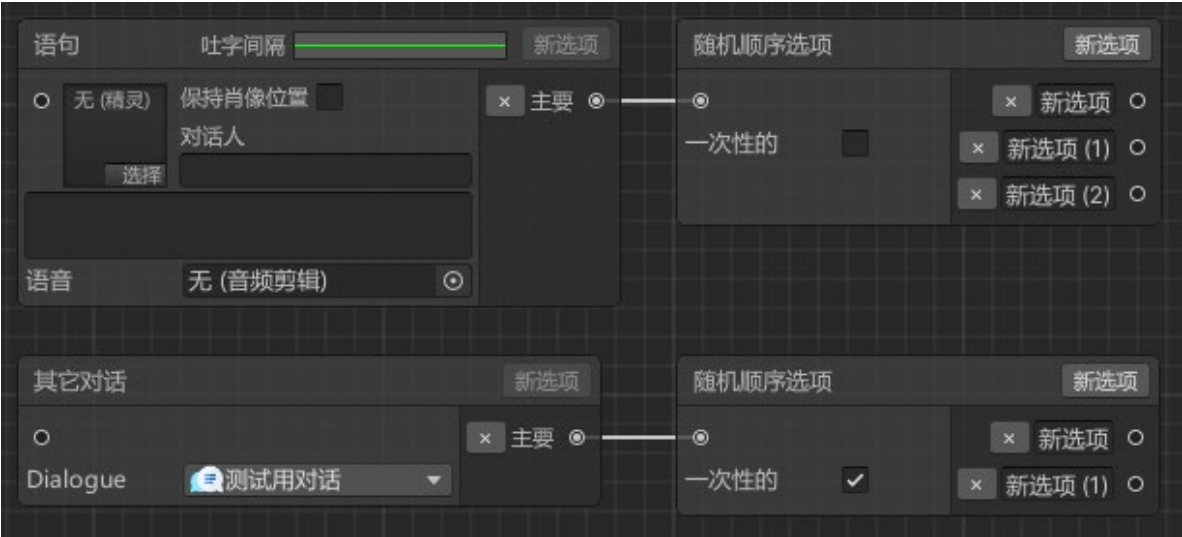


随机顺序选项结点类是 RandomOptions，继承自 ExternalOptionsNode，它的成员如下：

运行时成员	说明
属性 OneTime	是否仅在首次打乱选项的顺序
属性 IsValid	返回 True
方法 ReadOnlyCollection<DialogueOption> GetOptions(DialogueData, DialogueNode)	获取被打乱的选项

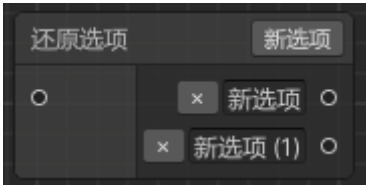
如何使用：

将语句结点或者其它对话结点的主要选项连接到对应的随机顺序选项结点即可，如图所示：



4.11 还原选项结点

还原选项结点是一种外置选项结点，可将因需要使用主选项来连接头像语音覆盖结点而无法使用普通选项的语句结点的普通选项外置在此处。它在对话编辑器中的图形结点如图所示：



还原选项结点类是 RevertOptions，继承自 ExternalOptionsNode，它的成员如下：

运行时成员	说明
属性 IsValid	返回 True
方法 ReadOnlyCollection<DialogueOption> GetOptions(DialogueData, DialogueNode)	获取该结点的选项，即返回 Options 属性

如何使用：

将覆盖结点的主选项连接到对应的还原选项结点即可，如图所示：



4.12 分支结点

分支结点可以将第一个满足显示条件的分支作为前一个结点的下一句，若无可用分支，则前一个结点将被视为结束结点。它在对话编辑器中的图形结点如图所示：



分支结点类是 `BranchNode`，继承自 `DialogueNode`，它的成员如下：

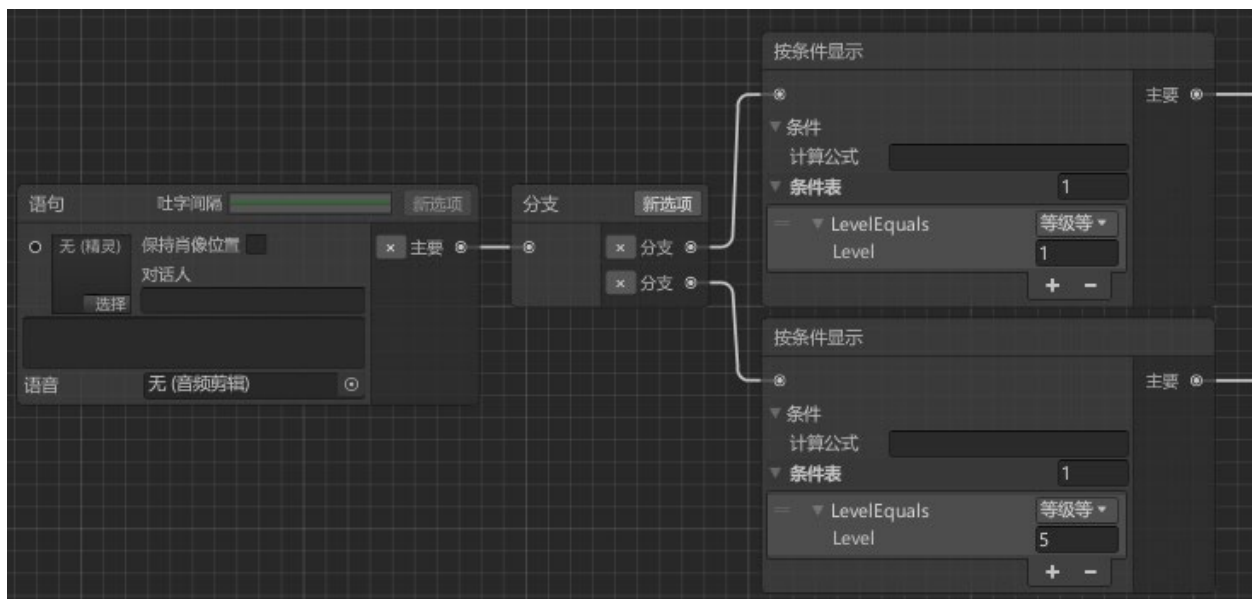
运行时成员	说明
属性 <code>IsValid</code>	当至少有一个选项且所有选项都是主要选项时返回 <code>True</code>
方法 <code>DialogueNode GetBranch(DialogueData)</code>	获取第一个满足显示条件的分支

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>bool CanConnectFrom(DialogueNode, DialogueOption)</code>	用于规定只有语句结点、其它对话结点的主要选项可以连接到此类结点

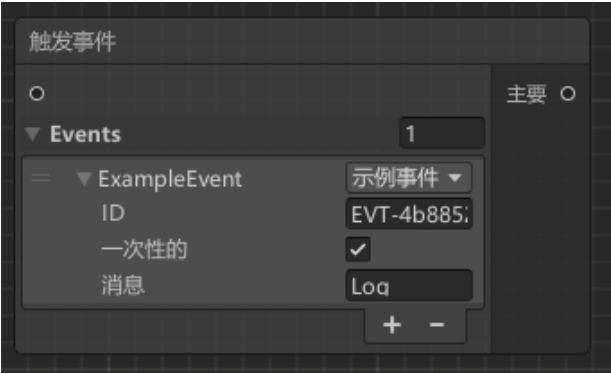
如何使用：

将语句结点或者其它对话结点的主要选项连接到对应的分支结点，然后分支结点的选项连接到条件显示结点即可，如图所示：



4.13 事件结点

进入事件结点时将触发给定的事件，然后以下一个结点继续对话。虽然语句结点也有事件，但它们都是在语句结点触发事件，而事件结点则可以分选项来触发不同的事件，还可以用于在进入目标结点之前触发事件。事件结点在对话编辑器中的图形结点如图所示：

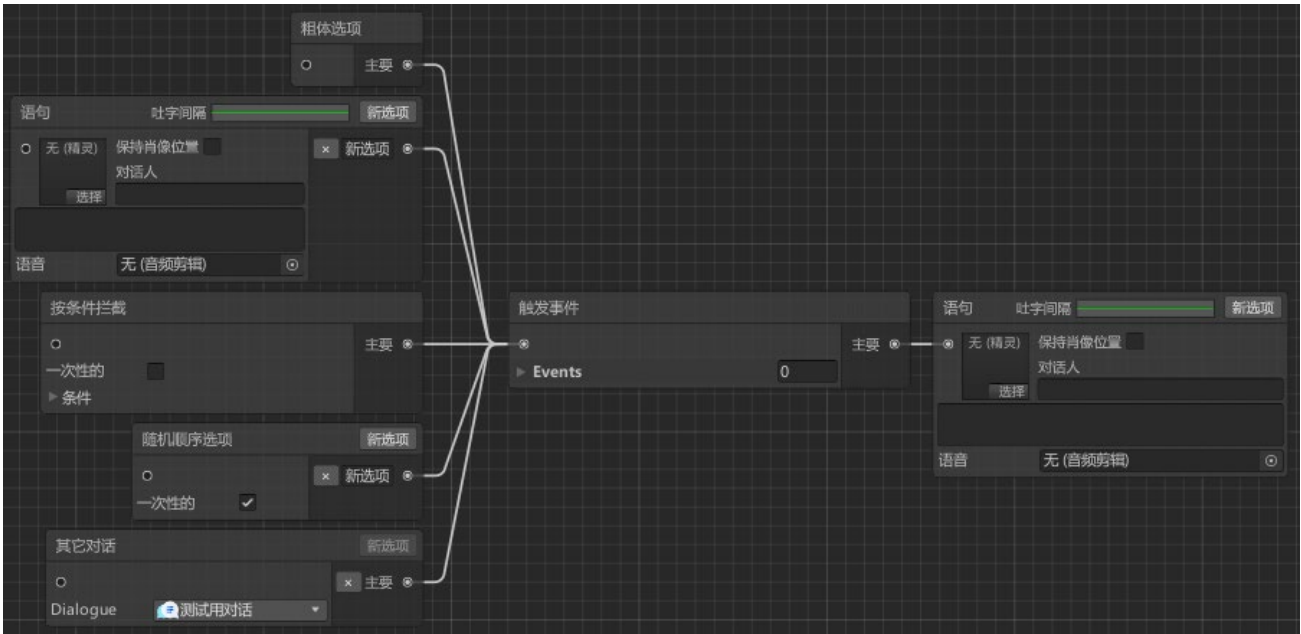


事件结点类是 `EventNode`，继承自 `DialogueNode`，`IEventNode`，`ISoloMainOptionNode`，`IManualNode`，它的成员如下：

运行时成员	说明
属性 <code>IsValid</code>	所有事件都有效时返回 <code>True</code>
字段 <code>events</code>	要触发的事件
属性 <code>Events</code>	<code>events</code> 的关联只读属性
构造函数 <code>EventNode()</code>	自动生成一个主要选项
方法 <code>void DoManual(DialogueWindow)</code>	触发事件后，以主要选项所连的结点继续对话

如何使用：

把任意非分支结点的选项连接到对应的事件结点即可，如图所示：



4.14 其它对话结点

其它对话结点会开始一段新的对话，但会保持首个对话的根结点作为首页（也就是点击对话窗口的“首页”

按钮时返回到的结点)，并在结束时返回原对话，适用于当前对话过于复杂，需要将部分结点分离到其它子对话的情景。它在对话编辑器中的图形结点如图所示：



其它对话结点类是 OtherDialogueNode，继承自 DialogueNode，IExitableNode，IManualNode，它的成员如下：

运行时成员	说明
属性 Dialogue	要进行的子对话
属性 IsValid	子对话非空且有结束结点时返回 True
方法 void DoManual(DialogueWindow)	开始一段新对话，并在结束后返回原对话

如何使用：

用法与语句结点一样，不再赘述。要注意的是，其它对话结点的选项相当于子对话最后一句话的选项，比如它的普通选项会作为子对话最后一句的选项显示出来。

4.15 按条件分叉结点

显而易见的，按条件分叉结点是一种分叉结点，当符合条件时，将使用第一个分支继续对话，否则使用第二个分支。它在对话编辑器中的图形结点如图所示：



按条件分叉结点类是 ConditionalBifurcation，继承自 BifurcationNode，IManualNode，它的成员如下：

运行时成员	说明
属性 Condition	需要满足的条件，类型是 ConditionGroup
属性 IsValid	Condition 填写有效时返回 True
方法 void DoManual(DialogueWindow)	当符合条件时，将使用第一个分支继续对话，否则使用第二个分支

4.16 按条件覆盖肖像结点

按条件覆盖肖像结点是一种肖像覆盖结点，在获取覆盖肖像时，若满足条件，则返回预留的肖像，否则，返回空值，以便被覆盖的语句结点保持原肖像。它在对话编辑器中的图形结点如图所示：

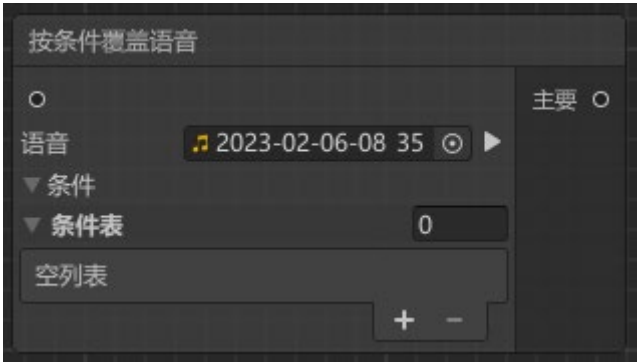


按条件覆盖肖像结点类是 ConditionalPortraitOverride，继承自 PortraitOverride，它的成员如下：

运行时成员	说明
属性 Portrait	用来替换的肖像
属性 Condition	需要满足的条件，类型是 ConditionGroup
属性 IsValid	替换肖像不为空且 Condition 填写有效时返回 True
方法 Sprite GetPortrait(DialogueData)	若满足条件，则返回用来替换的肖像，否则，返回空值

4.17 按条件覆盖语音结点

按条件覆盖语音结点是一种语音覆盖结点，在获取覆盖语音时，若满足条件，则返回预留的语音，否则，返回空值，以便被覆盖的语句结点保持原语音。它在对话编辑器中的图形结点如图所示：



按条件覆盖肖像结点类是 ConditionalPortraitOverride，继承自 PortraitOverride，它的成员如下：

运行时成员	说明
属性 Voice	用来替换的语音
属性 Condition	需要满足的条件，类型是 ConditionGroup
属性 IsValid	替换语音不为空且 Condition 填写有效时返回 True
方法 AudioClip GetVoice(DialogueData)	若满足条件，则返回用来替换的语音，否则，返回空值

第五节 结点用接口

5.1 单个主要选项结点接口

单个主要选项结点接口是 ISoloMainOptionNode，如果想让结点只有单个主要选项，则继承这个接口。要注意的是，结点类继承该接口后需要在构造函数自动生成一个主选项。

5.2 事件结点接口

事件结点接口是 IEventNode，如果想让结点在完成后调用事件，则继承这个接口。

它的成员如下：

运行时成员	说明
属性 Events	要触发的事件

5.3 可结束结点接口

可结束结点接口是 IExitableNode，如果想让结点可以被设置为退出结点，则继承这个接口。

5.4 手动处理结点接口

手动处理结点接口是 `IManualNode`，如果想不依赖对话框窗口用自定义行为处理某种结点，则继承这个接口。它的成员如下：

运行时成员	说明
方法 <code>void DoManual(DialogueHandler)</code>	处理结点的自定义行为

第六节 其它重要类型

6.1 对话选项

对话选项是非后缀结点的构成要素，在编辑器中的结点以出口端的方式展示。

对话选项类是 `DialogueOption`，它的成员如下：

运行时成员	说明
属性 <code>Title</code>	选项标题，即对话框窗口 UI 中选项按钮的显示文本
属性 <code>IsMain</code>	表示选项是否为主要选项
属性 <code>Next</code>	选项所连接的下一个结点
静态属性 <code>Main</code>	一个新建主要选项实例的快捷方式
构造函数 <code>DialogueOption()</code>	供序列化使用的无参构造函数
构造函数 <code>DialogueOption(bool, string)</code>	传入是否为主要选项和选项标题的构造函数
操作符 <code>bool(DialogueData)</code>	用于对对话选项进行快速判空

对话选项类还有一个供编辑器使用的内嵌静态类 `Editor`，它的成员如下：

编辑器成员	说明
静态方法 <code>void SetNext(DialogueOption, DialogueNode)</code>	设置选项所连接的下一个结点
静态方法 <code>void SetIsMain(DialogueOption, bool)</code>	设置选项是否为主要选项

6.2 对话数据

对话数据类是用于存储对话进度的类，比如要实现某个选项点击过一次后下一次不再出现这个选项之类的功能，就要借助这个对话数据类实现。

对话数据类是 `DialogueData`，它的成员如下：

运行时成员	说明
字段 <code>ID</code>	此数据的唯一标识码，即对应结点的 ID
字段 <code>accessed</code>	对应结点的访问状态，表示对应结点是否处理过了
属性 <code>Accessed</code>	<code>accessed</code> 的关联只读属性
字段 <code>children</code>	对应结点的选项所连结点的数据
属性 <code>Children</code>	<code>children</code> 的关联只读属性
字段 <code>family</code>	对应结点所在对话的所有结点的数据
属性 <code>Family</code>	<code>family</code> 的关联只读属性
字段 <code>eventStates</code>	对应结点的事件的触发状态
属性 <code>EventStates</code>	<code>eventStates</code> 的关联只读属性
属性 <code>IsDone</code>	对应结点是否标记为已完成，详见下文

属性 AdditionalData	供结点子类使用的 通用型数据
索引器 DialogueData this[DialogueNode]	快捷访问指定结点的数据
索引器 DialogueData this[string]	用 ID 快捷访问指定结点的数据
索引器 DialogueData this[int]	快捷访问 children 指定下标处的数据
构造函数 DialogueData(EntryNode)	传入根结点生成整个对话的结点的数据
构造函数 DialogueData(DialogueNode, Dictionary<string, DialogueData>)	生成指定结点的数据，并保存到所给的字典中，这个字典实际上就是它们的 family
构造函数 DialogueData(GenericData)	传入通用型数据生成整个对话的结点的数据
构造函数 DialogueData(string, Dictionary<string, DialogueData>)	生成指定 ID 的空数据，并保存到所给的字典中，这个字典实际上就是它们的 family
方法 void Refresh(EntryNode)	用于对话结构改变后覆盖刷新原数据，如结点增删、连线改变、条件改变等
方法 void Access()	将 accessed 设置为 True 表示对应结点已访问过
方法 void AccessEvent(string)	用于记录一次性事件的已执行状态
方法 GenericData GenerateSaveData()	获取用于存档的通用型数据
静态方法 void Traverse(DialogueData, Action<DialogueData>)	从指定结点数据开始深度优先遍历它和它的子结点数据
静态方法 bool Traverse(DialogueData, Func<DialogueData, bool>)	从指定结点数据开始遍历它和它的子结点数据，可被中断，返回值表示是否发生中断
操作符 bool(DialogueData)	用于对结点数据进行快速判空

什么时候结点数据被标记为已完成？本插件规定如下：

- 1、当结点没有后续结点或者它是结束结点时，它的完成状态就是它的访问状态；
- 2、当结点是拦截结点时，如果它可以进入它且下一个结点对应的数据已标记为完成，则标记为完成；
- 3、当结点是条件结点时，如果条件满足且下一个结点对应的数据也标记完成了，则标记为完成；
- 4、当结点是分支结点时，如果没有可用分支，则它的完成状态就是它的访问状态；如果有且那个分支的第一个结点的数据已标记完成，则它也标记完成；
- 5、当结点是外置选项结点时，如果它访问过了，而且它的实际选项中存在任意一个选项，它所连接的下一个结点可到达结束结点，那个结点的数据也已经标记为完成，则此结点数据也标记为完成。
- 6、除去以上情况，如果结点不是分叉结点（因为其完成状态是由用户定义的），它已经被访问过了，而且它所有选项所连接的下一个结点的数据也都已经标记为完成，则此结点数据标记为完成。

6.3 对话事件

对话事件用于在某些情况下触发事件，一般由继承了 IEventNode 的结点持有。对话进行到拥有对话事件的结点时会触发相应的对话事件，比如 Debug 事件、设置触发器事件等等，以便其它地方要用。

对话事件类是 DialogueEvent，在编辑器代码继承自 [ICopiable](#)，它的成员如下：

运行时成员	说明
属性 ID	对话事件的唯一标识码，自动生成，也可以手动填写
属性 OneTime	事件是否仅触发一次，默认为 True
属性 IsValid	用于检查事件的填写是否有效，否则不会触发
方法 void Invoke(DialogueData)	尝试触发事件
方法 void CancelInvoke(DialogueData)	取消触发，在关闭对话窗口或开始新对话时调用
方法 bool Invoke()	供子类重写的事件触发方法，返回值表示是否成功触发
静态方法 string GetName(Type)	获取事件的名称

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>object Copy()</code>	复制此事件，可被子类重写

如何扩展：

下面给出一个打印事件的示例代码：

[SerializeField, Name("示例事件")]

public sealed class ExampleEvent : DialogueEvent

```
{
    [field: SerializeField]
    public string Message { get; private set; }

    public override bool IsValid => !string.IsNullOrEmpty(Message);

    protected override bool Invoke()
    {
        Debug.Log(Message);
        return true;
    }
}
```

6.4 对话分组

对话分组用于在编辑器中给结点分组，不应在游戏逻辑中使用。它在对话编辑器中的图形分组如图所示：



对话分组类是 DialogueGroup，它的成员如下：

编辑器成员	说明
字段 <code>_name</code>	分组的名称
字段 <code>_nodes</code>	分组内的结点
字段 <code>_position</code>	分组的位置
构造函数 <code>DialogueGroup(string, Vector2)</code>	传入组名和位置的构造函数

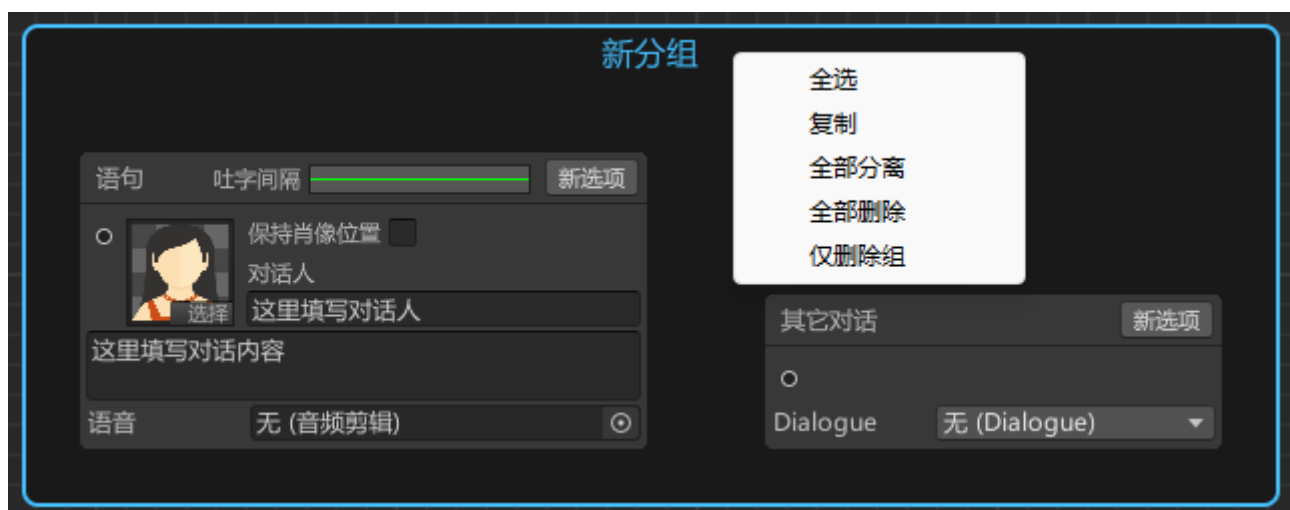
如何使用：

在编辑器对话视图中的空白处点击右键可创建分组，此时可以给分组命名，如图所示：

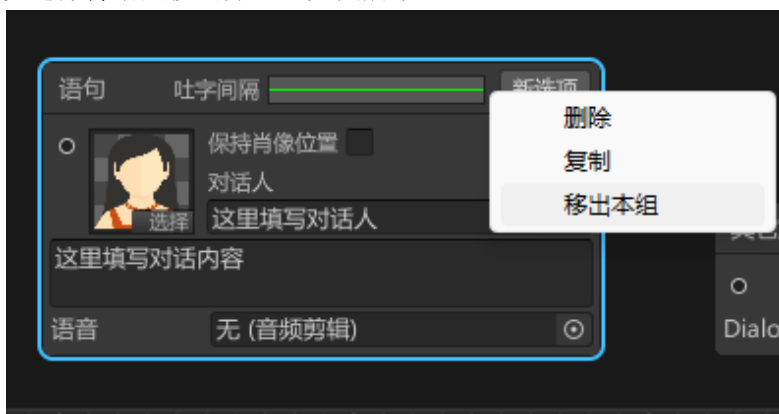


若忘记命名，双击分组头部可对分组进行重命名。

右键点击分组头部可对分组内容进行操作，如图所示：



其中，“全选”可以选中组内全部结点，“复制”可以对整组进行复制，“全部分离”则会把组内所有结点都取消分组，“全部删除”则会连着分组一起删除组内所有结点，“仅删除组”则是保留组内结点，但删除分组。右键单击组内结点可以选择将结点移出分组，如图所示：



6.5 对话管理器

对话管理器是用来管理对话数据的，比如对话数据的增删、存读档，是一个静态类。

对话管理器类是 `DialogueManager`，它的成员如下：

运行时成员	说明
静态字段 <code>data</code>	所有进行过的对话的根结点数据
静态方法 <code>DialogueData GetOrCreateData(EntryNode)</code>	根据所给根结点查找对话数据，没有则创建
静态方法 <code>void RemoveData(EntryNode)</code>	移除所给根结点的对话数据，可用于例如答题型对话，放弃答题后重置答题进度之类的功能。
静态方法 <code>void RemoveData(string)</code>	移除所给根结点 ID 对应的对话数据，用法同上
静态方法 <code>void SaveData(SaveData)</code>	将对话数据存入所给的存档
静态方法 <code>void LoadData(SaveData)</code>	从所给的存档中读取对话数据并覆盖

6.6 对话处理器

对话处理器是用来处理对话逻辑的，是对话系统的核心部分，由对话窗口使用。对话处理器要传入一些特定的回调才能正常发挥其功能，本插件把这些回调一起放在一个叫做对话处理器回调的类里面，方便初始化。对话处理器回调类是 `DialogueHandlerCallbacks`，它的成员如下：

运行时成员	说明
字段 <code>setName</code>	设置对话人名称文本的回调
字段 <code>setContent</code>	设置对话内容文本的回调
字段 <code>preprocessText</code>	文本预处理回调，对文本进行轻微加工，比如仅翻译文本而不处理文本中的关键字
字段 <code>processText</code>	文本处理回调，可在这个回调里进一步处理上个回调稍微处理过的文本
字段 <code>setPortrait</code>	设置对话人肖像的回调
字段 <code>setPortraitDark</code>	让对话人肖像变暗的回调
字段 <code>playVoice</code>	语音播放回调
字段 <code>stopVoice</code>	停止播放语音的回调
字段 <code>startCoroutine</code>	协程开始回调
字段 <code>stopCoroutine</code>	协程停止回调
字段 <code>onHandleNode</code>	结点处理完成时的回调
字段 <code>onEndWriting</code>	打字动画完成时的回调
字段 <code>onReachLastSentence</code>	到达最后一句时的回调
字段 <code>refreshNextAction</code>	刷新下一句的操作的回调
字段 <code>refreshOptions</code>	刷新选项栏的回调
字段 <code>sendMessage</code>	消息发送回调
字段 <code>getSkipDelay</code>	获取跳过间隔的回调，用来决定一句话至少显示多久才能跳过打字动画。为空时默认为 0.5 秒

对话处理器类是 `DialogueHandler`，它的成员如下：

运行时成员	说明
字段 <code>callbacks</code>	回调集合
字段 <code>window</code>	本处理器隶属于的对话窗口
属性 <code>Interlocutor</code>	发起本次对话的对话人
属性 <code>Home</code>	触发本次对话时遇到的第一个根结点
属性 <code>CurrentEntry</code>	当前根结点

属性 CurrentEntryData	当前根结点的数据
字段 currentNode	当前结点
属性 CurrentNodeData	当前结点的数据
字段 continueNodes	本次对话结束时用来继续对话的结点
字段 manualNodes	用于防止进行结点的自定义操作时出现死循环
属性 IsDoingManual	是否正在使用自定义操作处理结点
字段 isUsingLeftPortrait	是否正在使用左侧肖像，仅在启用肖像功能时可用
字段 hasLeftPortrait	左侧肖像是否非空，仅在启用肖像功能时可用
字段 hasRightPortrait	右侧肖像是否非空，仅在启用肖像功能时可用
字段 currentInterlocutor	当前语句的对话人，仅在启用肖像功能时可用
字段 portraitInterlocutor	当前侧肖像的对话人，仅在启用肖像功能时可用
字段 targetContent	要播放打字动画的文本
字段 writeCoroutine	播放打字动画的协程
字段 playTime	打字动画的播放时间
属性 IsPlaying	打字动画是否正在播放
字段 waitCoroutine	等待协程
字段 resumeOnWaitOver	是否在等待结束时刷新交互按钮
属性 IsWaiting	是否正在等待满足条件
构造函数 DialogueHandler (DialogueHandlerCallbacks, object)	传入回调和对话框口的构造函数
方法 void StartWith(IInterlocutor)	根据传入的对话人发起新的会话
方法 void StartWith(Dialogue)	根据传入的对话发起新的会话
方法 void StartWith(string, string)	根据传入的对话人和对话内容发起新的会话
方法 void StartWith(EntryNode)	根据传入的根结点发起新的会话
方法 void ContinueWith(DialogueNode)	从属于本次对话的指定结点继续会话
方法 void HandleNode(DialogueNode)	处理所给的结点在 UI 方面的效果
方法 bool HandleSpecialNodes (ref DialogueNode)	处理特殊结点，如跳过修饰结点、条件结点等
方法 void HandleLastSentence()	处理当前对话的最后一句话，例如 ExitHere 为 True 的结点、没有下一句的结点等
方法 void HandleInteraction()	处理对话框口的交互按钮
方法 void RefreshInteraction()	刷新交互按钮，比如设置 nextClicked、刷新选项等
方法 void ResetInteraction()	重置对话框口的交互按钮，如清除选项等
方法 void DisplaySentence(SentenceNode)	显示语句结点的对话内容
方法 IEnumerator Write(SentenceNode)	打字动画播放协程
方法 void Skip()	跳过打字动画
方法 void EndWriting()	打字动画收尾
方法 void StopWriting()	停止打字动画
方法 Coroutine StartCoroutine(IEnumerator)	开始协程
方法 void StopCoroutine(Coroutine)	停止协程
方法 string PreprocessText(string)	预处理文本
方法 string ProcessText(string)	先预处理文本，再进一步处理
方法 void RefreshNextAction(Action)	设置下一句的操作

方法 void HandleOptions (IList<DialogueOption>)	处理传入的选项，生成对应的按钮
方法 void RefreshOptions(List<OptionCallback>)	刷新选项列表
方法 void RefreshPortrait(SentenceNode)	刷新并显示给定语句的肖像，仅在启用肖像功能时有效
方法 void ResetPortraits()	重置肖像，并隐藏起来，仅在启用肖像功能时有效
方法 void SetPortraitDark(Image, bool)	让肖像变暗或取消变暗，仅在启用肖像功能时有效
方法 void PlayVoice(SentenceNode)	播放语句结点的语音，仅在启用语音功能时有效
方法 void StopVoice()	停止当前语音，仅在启用语音功能时有效
方法 void BackHome()	回到首页
方法 void BackHomeImmediate()	回到首页，但不播放打字动画
方法 void DoOption(DialogueOption)	执行选项所连结点的功能
方法 void DoManual(IManualNode)	执行结点的自定义操作
方法 void InvokeEvent(DialogueEvent)	触发所给事件
方法 void WaitUntil(Func<bool>)	等待直到满足给定条件才显示交互按钮
方法 void PushInteraction(DialogueNode)	将所给结点入栈，当本次对话结束时，将会显示栈顶结点的选项，一般由其它对话结点使用
方法 DialogueNode PopInteraction()	出栈上一个方法入栈的结点
方法 void Init()	初始化对对话处理器

6.7 对话人接口

如果想让某种可交互对象可触发对话，则需要继承对话人接口。

对话人接口是 `IInterlocutor`，继承自 [IInteractive](#)，它的成员如下：

运行时成员	说明
属性 Dialogue	用来开始交谈的对话

如何使用：

让触发对话的对象的类继承这个接口即可，如图所示：

```
[DisallowMultipleComponent, KeywordsSet("对话人")]
@ Unity 脚本 (3 个资产引用) | 3 个引用
public class Interlocutor : Character, IInterlocutor, IKeyword
{
    public Gender gender;

    2 个引用
    public bool IsInteractive...

    [field: SerializeField, SpriteSelector]
    2 个引用
    public Sprite Icon { get; private set; }

    [field: SerializeField]
    2 个引用
    public Dialogue Dialogue { get; private set; }

    5 个引用
    bool IInteractive.Interactable { get; set; }

    6 个引用
    string IKeyword.IDPrefix => "NPC";
}
```

第二章 条件系统

第一节 基本概念

本插件附带的条件系统是一个通用型的系统，可以用在很多地方，它由条件和条件组组成，其中，条件组又由条件计算公式和条件列表组成，计算公式可以让条件列表中的条件按给定的表达式进行混合判断，而不是单纯的逐个判断。

第二节 组成要素

2.1 条件

条件类是 `Condition`，它的成员如下：

运行时成员	说明
属性 <code>IsValid</code>	表示条件的填写是否有效
方法 <code>bool IsMet()</code>	条件是否满足
静态方法 <code>string GetGroup(Type)</code>	获取某类条件的分组名称
静态方法 <code>string GetName(Type)</code>	获取某类条件的名称
操作符 <code>bool(Condition)</code>	用于对条件进行快速判空

如何扩展：

下面给出等级条件的代码示例：

[Serializable]

```
public abstract class LevelCondition : Condition
```

```
{
```

```
    [field: SerializeField, Min(0)]
```

```
    public int Level { get; private set; } = 1;
```

```
    public override bool IsValid => Level >= 0;
```

```
}
```

```
[Serializable, Name("等级等于"), Group("等级相关")]
```

```
public sealed class LevelEquals : LevelCondition
```

```
{
```

```
    public override bool IsMet()
```

```
    {
```

```
        return PlayerManager.player.level == Level;
```

```
    }
```

```
}
```

```
[Serializable, Name("等级大于"), Group("等级相关")]
```

```
public sealed class LevelHigherThan : LevelCondition
```

```
{
```

```
    public override bool IsMet()
```

```
    {
```

```
        return PlayerManager.player.level > Level;
```

```

    }
}
[Serializable, Name("等级小于"), Group("等级相关")]
public sealed class LevelLowerThan : LevelCondition
{
    public override bool IsMet()
    {
        return PlayerManager.player.level < Level;
    }
}

```

2.2 条件组

条件组类是 `ConditionGroup`，它的成员如下：

运行时成员	说明
字段 <code>formula</code>	条件计算公式
属性 <code>Formula</code>	<code>formula</code> 的关联只读属性
字段 <code>conditions</code>	条件列表
属性 <code>Conditions</code>	<code>conditions</code> 的关联只读属性
属性 <code>IsValid</code>	表示条件列表中的条件填写是否都有效
方法 <code>bool IsMet()</code>	条件给出的条件是否满足
操作符 <code>bool(ConditionGroup)</code>	用于对条件组进行快速判空

其中，对于条件计算公式 `formula`，有如下规定：

- 1、操作数为条件列表中条件的下标；
- 2、运算符可使用 "("、")"、"|" (或)、"&" (且)、"!" (非)；
- 3、未对非法输入进行处理，需按规范填写；
- 4、例：(0 | 1) & !2 表示满足条件 0 或 1 且不满足条件 2；
- 5、为空时默认进行相互的“且”运算。

第三章 关键字系统

第一节 基本概念

本插件附带的关键字系统可以在字符串中插入给定格式的 ID 字符串，然后在处理该字符串时，把其中的 ID 字符串替换为 ID 所对应的对象的名称。如假设有 ID 为“NPC0001”、名字为“小明”的对话人，那么它的 ID 字符串就是“{[NPC]NPC0001}”，当输入字符串“我是{[NPC]NPC0001}”时，它会被处理成“我是小明”。

第二节 组成要素

2.1 关键字接口

如果想让某个类型的对象可以作为关键字对象，那么可以让那个类型继承关键字接口。

关键字接口是 `IKeyword`，它的成员如下：

运行时成员	说明
属性 ID	关键字对象的 ID
属性 IDPrefix	放在关键字 ID 字符串方括号内的前缀字符串，用于对关键字进行分类。例如 “[NPC]NPC001”，前缀是“NPC”
属性 Name	关键字对象的名称
属性 Color	在 UI 文本上显示时的富文本颜色

由编辑器使用的成员如下：

编辑器成员	说明
属性 Group	关键字对象的类间子分组

如何使用：

让关键字对象类继承这个接口，然后实现这个接口的成员即可，如图所示：

```
[DisallowMultipleComponent, KeywordsSet("对话人")]
☺ Unity 脚本 (3 个资产引用) | 3 个引用
public class Interlocutor : Character, IInterlocutor, IKeyword
{
    public Gender gender;

    2 个引用
    public bool IsInteractive...

    [field: SerializeField, SpriteSelector]
    2 个引用
    public Sprite Icon { get; private set; }

    [field: SerializeField]
    2 个引用
    public Dialogue Dialogue { get; private set; }

    5 个引用
    bool IInteractive.Interactable { get; set; }
```

2.2 关键字类

关键字类相当于关键字的管理器，是一个静态类，用于将字符串中的关键字 ID 字符串转成其对应的关键字对象的名称。要注意的是，为了关键字转换的准确性，每次转换之前都会搜集一次所有关键字对象，性能会有所影响，所以请尽量不要在 `void Update()` 之类的更新方法中转换关键字。

关键字类是 `Keyword`，它的成员如下：

运行时成员	说明
静态方法 <code>string Translate(string, bool)</code>	将给定的 ID 字符串转成名称，可选用富文本给名称染色
静态方法 <code>string Translate(string, bool, Dictionary<string, Dictionary<string, IKeyword>>)</code>	利用所给的关键字对象字典将给定的 ID 字符串转成名称，可选用富文本给名称染色
静态方法 <code>Dictionary<string, Dictionary<string, IKeyword>> CollectKeywords()</code>	用于在转换关键字之前搜集关键字对象
静态方法 <code>string HandleKeywords(string, bool)</code>	将所给字符串中的 ID 字符串转成名称，可选用富文本给名称染色
静态方法 <code>IEnumerable<KeyValuePair<string, string>> ExtractKeyWords(string)</code>	提取所给字符串中的 ID 字符串，转换成对应名称后以键值对列表的形式返回
静态方法 <code>string Generate(IKeyword)</code>	生成所给关键字对象的 ID 字符串
静态方法 <code>bool IsKeyword(string)</code>	所给字符串是否为 ID 字符串

关键字类还有一个供编辑器使用的内嵌静态类 `Editor`，它的成员如下：

编辑器成员	说明
静态方法 <code>string Translate(string)</code>	将给定的 ID 字符串转成名称
静态方法 <code>string Translate(string, Dictionary<string, Dictionary<string, IKeyword>>)</code>	利用所给的关键字对象字典将给定的 ID 字符串转成名称
静态方法 <code>Dictionary<string, Dictionary<string, IKeyword>> CollectKeywords()</code>	用于在转换关键字之前搜集关键字对象
静态方法 <code>string HandleKeywords(string)</code>	将所给字符串中的 ID 字符串转成名称
静态方法 <code>IEnumerable<KeyValuePair<string, string>> ExtractKeyWords(string)</code>	提取所给字符串中的 ID 字符串，转换成对应名称后以键值对列表的形式返回
静态方法 <code>void OpenKeywordsSelection(Vector2, Action<string>)</code>	打开关键字选取窗口，然后把选中的关键字对象以 ID 字符串的形式传入所给的回调中
静态方法 <code>void SetAsKeywordsField(TextField)</code>	将文本框设置为可右键点击插入关键字 ID 字符串

第三节 相关特性

3.1 关键字集合特性

关键字集合特性，是一个由编辑器使用的特性，用于在关键字选择窗口中对关键字进行根分组。

关键字集合特性类是 `KeywordsSetAttribute`，继承自 `Attribute`，它的成员如下：

运行时成员	说明
字段 <code>name</code>	集合的名称

如何使用：

加到关键字对象类上即可，如图所示：

```

[DisallowMultipleComponent, KeywordsSet("对话人")]
Unity 脚本 (3 个资产引用) | 3 个引用
public class Interlocutor : Character, IInterlocutor, IKeyword
{
    public Gender gender;

    2 个引用
    public bool IsInteractive...

    [field: SerializeField, SpriteSelector]
    2 个引用
    public Sprite Icon { get; private set; }

    [field: SerializeField]
    2 个引用
    public Dialogue Dialogue { get; private set; }

    5 个引用
    bool IInteractive.Interactable { get; set; }
}

```

3.2 关键字获取方法特性

关键字获取方法特性，也是一个由编辑器使用的特性，把它加到获取可用关键字的无参编辑器静态方法上，这样编辑器才能搜集关键字。要注意的是，那个静态方法要返回 `IEnumerable<IKeyword>`。

关键字获取方法特性类是 `GetKeywordsMethodAttribute`，继承自 `Attribute`。

如何使用：

加到可由编辑器使用的静态方法上即可，如图所示：

```

[RuntimeGetKeywordsMethod, GetKeywordsMethod]

public static IEnumerable<Interlocutor> GetInterlocutors()
{
    return FindObjectsOfType<Interlocutor>(true);
}

```

3.3 运行时关键字获取方法特性

运行时关键字获取方法特性，是一个由运行时代码使用的特性，把它加到获取可用关键字的无参运行时静态方法上，这样才能在游戏运行时搜集关键字。要注意的是，那个静态方法要返回 `IEnumerable<IKeyword>`。

运行时关键字获取方法特性类是 `RuntimeGetKeywordsMethodAttribute`，继承自 `Attribute`。

如何使用：

加到在游戏运行时代码中的静态方法上即可，如图所示：

```

[RuntimeGetKeywordsMethod, GetKeywordsMethod]
0 个引用
public static IEnumerable<Talker> GetTalkers()
{
    return FindObjectsOfType<Talker>(true);
}

```


第四章 多语言系统

第一节 基本概念

本插件附带的多语言系统可将游戏中的文本切换为多种语言，它由翻译映射、翻译包（即翻译映射表）、本地化文件组成。在翻译时，先使用一种名为“选择器”的索引在本地化文件中筛选要用到的翻译包，接着再使用翻译索引在这些翻译包中查找对应的翻译映射，找到后，再进一步取出当前语言的翻译内容（即用该语言书写的字符串），到此，翻译完成。为了翻译性能，多语言系统并不是实时更新的，仅在游戏启动和切换语言后刷新一次翻译缓存，这个缓存是以选择器为索引的字典，字典项是一个子字典，这个子字典才是真正的翻译映射表，它整合了选择器对应的所有翻译包中的翻译映射。因此，在游戏运行时，若翻译包的内容有所改变，它并不会实时反映到游戏中去。多语言系统也可以用在编辑器，对于编辑器，翻译方法有所改变：由于编辑器可以随意获取和引用翻译包，所以翻译时是直接使用所给翻译包的第一种语言的翻译内容。

第二节 组成要素

2.1 翻译映射

翻译映射是一种键值对，使用特定的键索引指定文本的翻译内容，翻译内容又细分为各个语言的翻译，它的结构可表示为：

翻译索引	语言 1 的翻译内容	语言 N 的翻译内容
“索引 01”	“Index 01”	“インデックス 01”

其中，可直接将默认语言（即开发者的母语）的源文本作为索引，这样会比较方便，不用反复查找某个索引所对应的源文本，有时候索引指向的项丢失了，就无法知道源文本了。

例如，某个按钮上的显示文本为“确定”，那么“确定”就是所谓的源文本，如果不使用源文本作为索引，那么假设它在翻译映射中的索引为“BUTTON_01”，第一语言的翻译是“确定”，第二语言是“OK”，假设现在这一条映射丢失了，会导致这个按钮直接显示索引“BUTTON_01”，但我们并不知道“BUTTON_01”的真正含义，就无法补充翻译了。本插件就采取源文本作为索引这种方式，不想这样的话可以把 [Language](#) 类的 [LanguageIndexOffset](#) 属性设置为 0 来取消。

如果有索引相同但翻译不同的情况，可以给重复的其它索引加上前缀“[DUP+数字]”，如重复的“索引 001”可以改为“[DUP02]索引 001”。当因找不到翻译而直接返回索引时，会自动去除这个前缀。

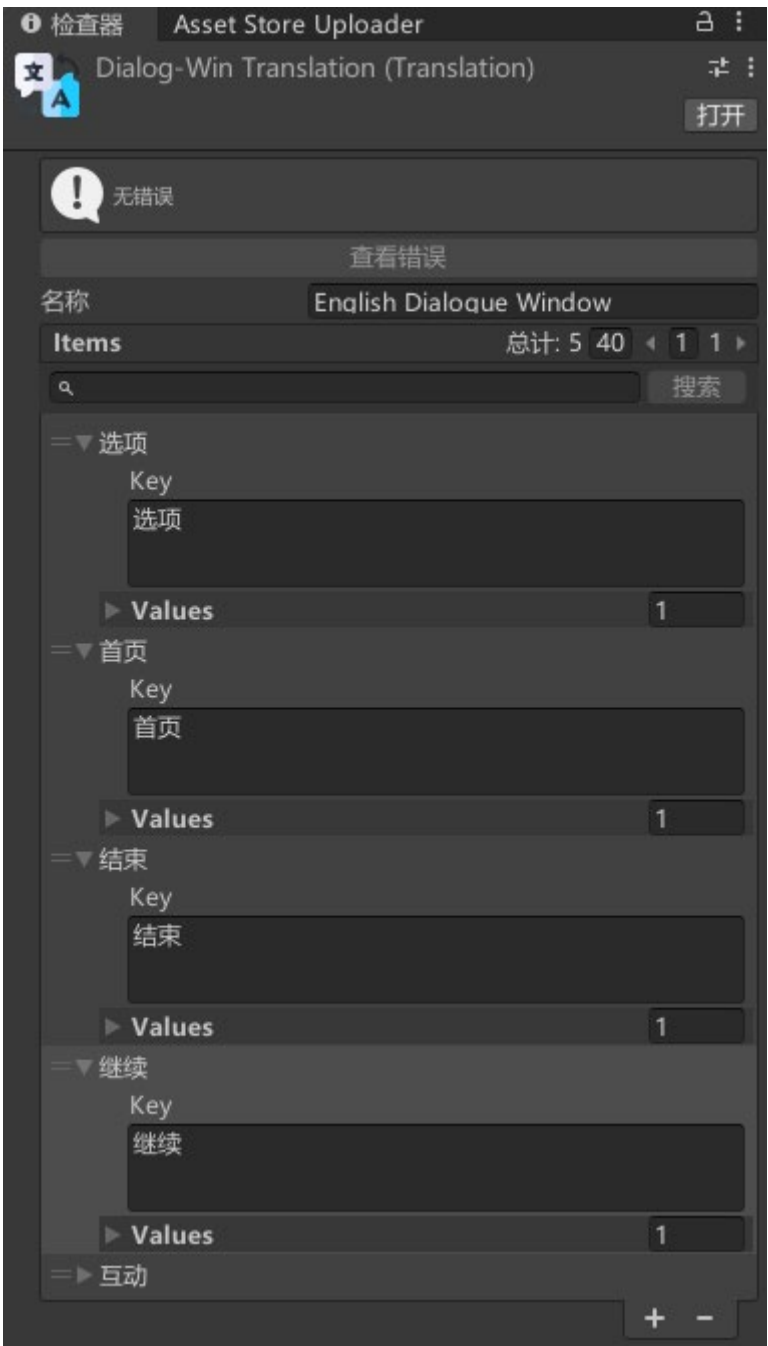
翻译映射类是 TranslationItem，它的成员如下：

运行时成员	说明
属性 Key	此项映射的索引
字段 values	此项映射各语言的翻译内容
属性 Values	values 的关联只读属性
构造函数 TranslationItem()	供序列化使用的无参构造函数
构造函数 TranslationItem (string, params string[])	传入索引和各语言翻译内容的构造函数

2.2 翻译包

翻译包是一种 ScriptableObject 资源文件，是翻译映射的一个集合，也就是说一个表。可以只使用一个全局翻译包，也可以每个系统分别使用一个翻译包，这样在处理翻译时可以进行分工。它在编辑器中的检查器界

面如下图所示：



翻译包类是 Translation，继承自 ScriptableObject，它的成员如下：

运行时成员	说明
字段 items	翻译映射列表
属性 Items	items 的关联只读属性
方法 Dictionary<string, string> AsDictionary(int)	把指定语言的翻译提取出来并生成字典
方法 string Tr(int, string)	用指定语言的翻译内容进行翻译
方法 string Tr(string)	默认使用第一个语言的翻译内容进行翻译

由编辑器使用的成员如下：

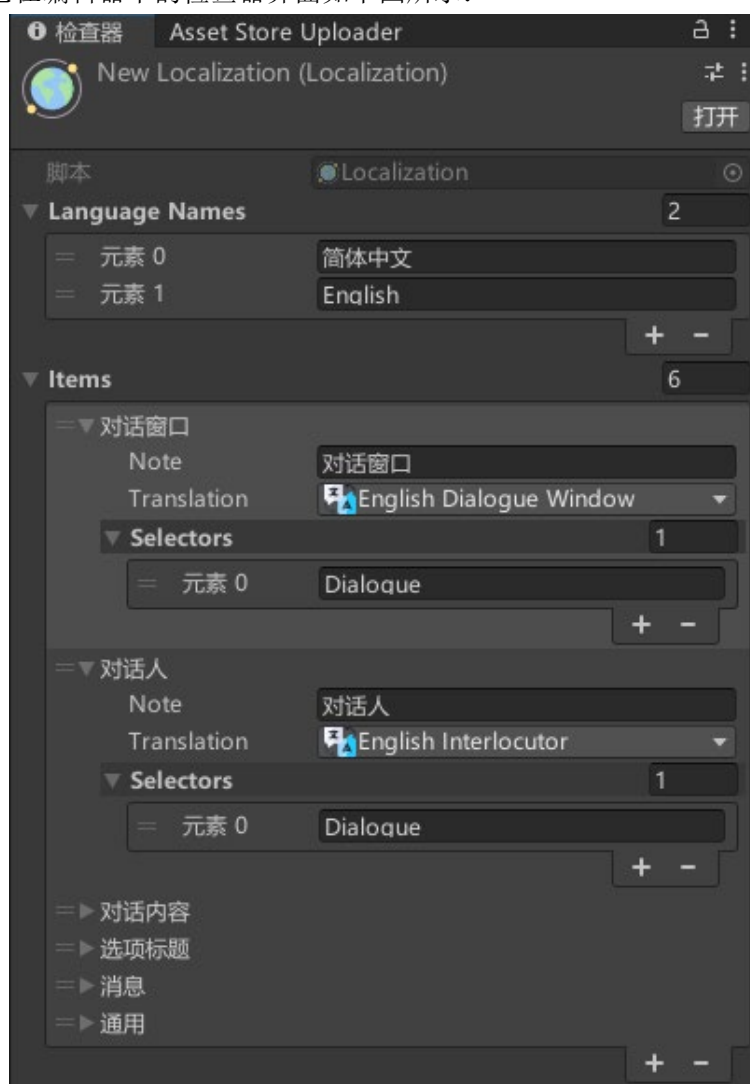
编辑器成员	说明
字段 _name	翻译包的名称，一般用于备注

翻译包类还有一个供编辑器使用的内嵌静态类 `Editor`，它的成员如下：

编辑器成员	说明
静态方法 <code>void SetName(Translation, string)</code>	设置所给翻译包的名称
静态方法 <code>void SetItems(Translation, TranslationItem[])</code>	设置所给翻译包的翻译映射列表

2.3 本地化文件

本地化文件是翻译包的进一步整合，是一种资源文件，它使用“选择器”来对翻译包进行分组。不过，选择器和翻译包并不是一对一的关系，一个选择器可以用在多个不同的翻译包上面，一个翻译包也可以被多个不同的选择器使用。它在编辑器中的检查器界面如下图所示：



本地化文件类是 `Localization`，继承自 `ScriptableObject`，它的成员如下：

运行时成员	说明
字段 <code>languageNames</code>	语言名称列表
属性 <code>LanguageNames</code>	<code>languageNames</code> 的关联只读属性
字段 <code>items</code>	翻译包和选择器的映射列表，项类型是 LocalizationItem
属性 <code>Items</code>	<code>items</code> 的关联只读属性
属性 <code>Instance</code>	本地化文件单例

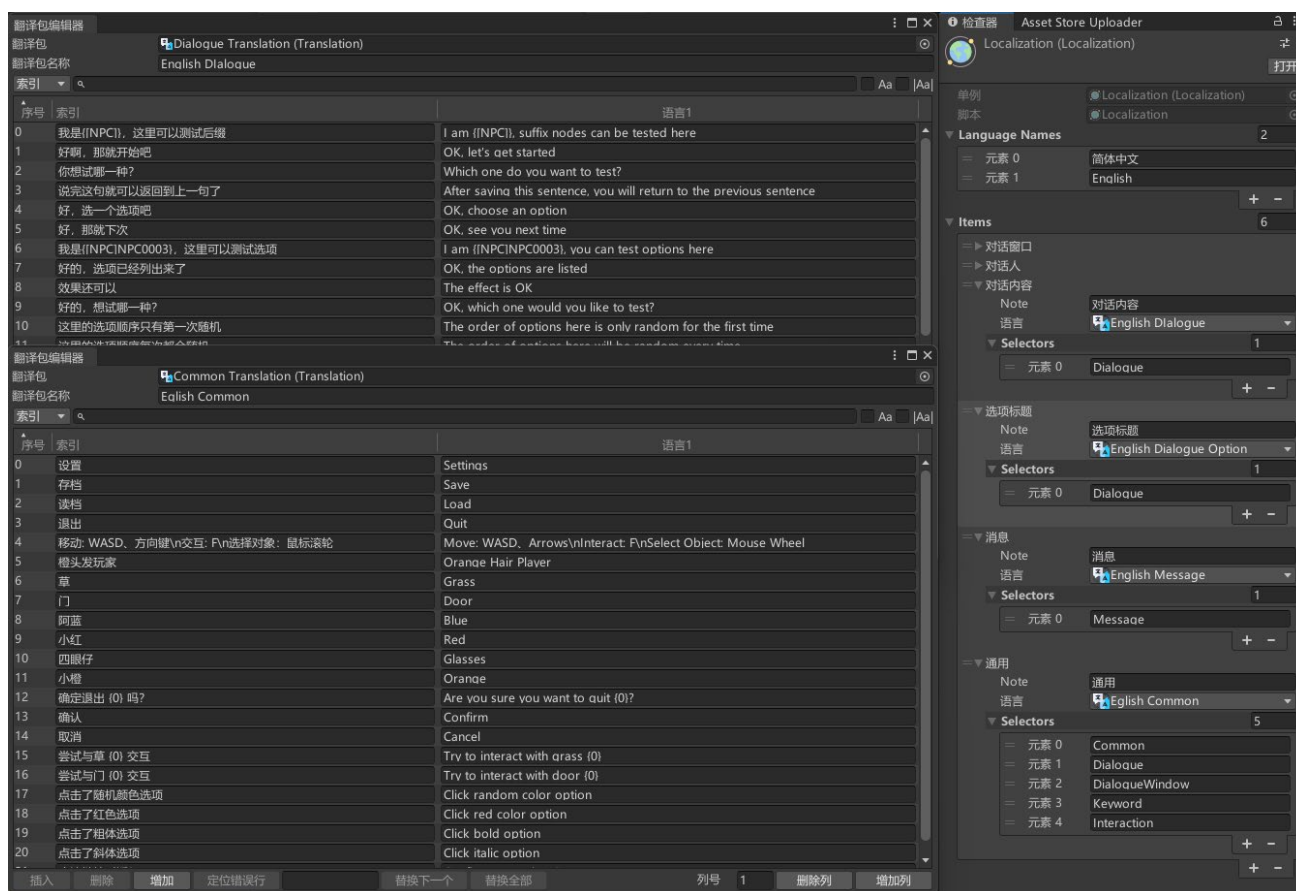
方法 Dictionary<string, List<Dictionary<string, string>>> AsDictionary(int)	生成此本地化文件的翻译缓存字典
方法 List<Dictionary<string, string>> FindDictionaries(string, int)	查找所给选择器对应的翻译字典集合

由编辑器使用的成员如下：

编辑器成员	说明
静态方法 void Create()	创建一个本地化文件单例

如何使用：

先在 Items 加入新的项，然后设置该项的翻译包，接着添加用来索引这个翻译包的选择器即可，也就是说在翻译时使用哪些选择器会选用这个翻译包来进行翻译，如图所示：



2.4 本地化分组

本地化分组是翻译包和选择器的映射。

本地化分组类是 LocalizationItem，它的成员如下：

运行时成员	说明
属性 Language	这个选择器分组所用的翻译包
字段 selectors	选择器列表
属性 Selectors	selectors 的关联只读属性

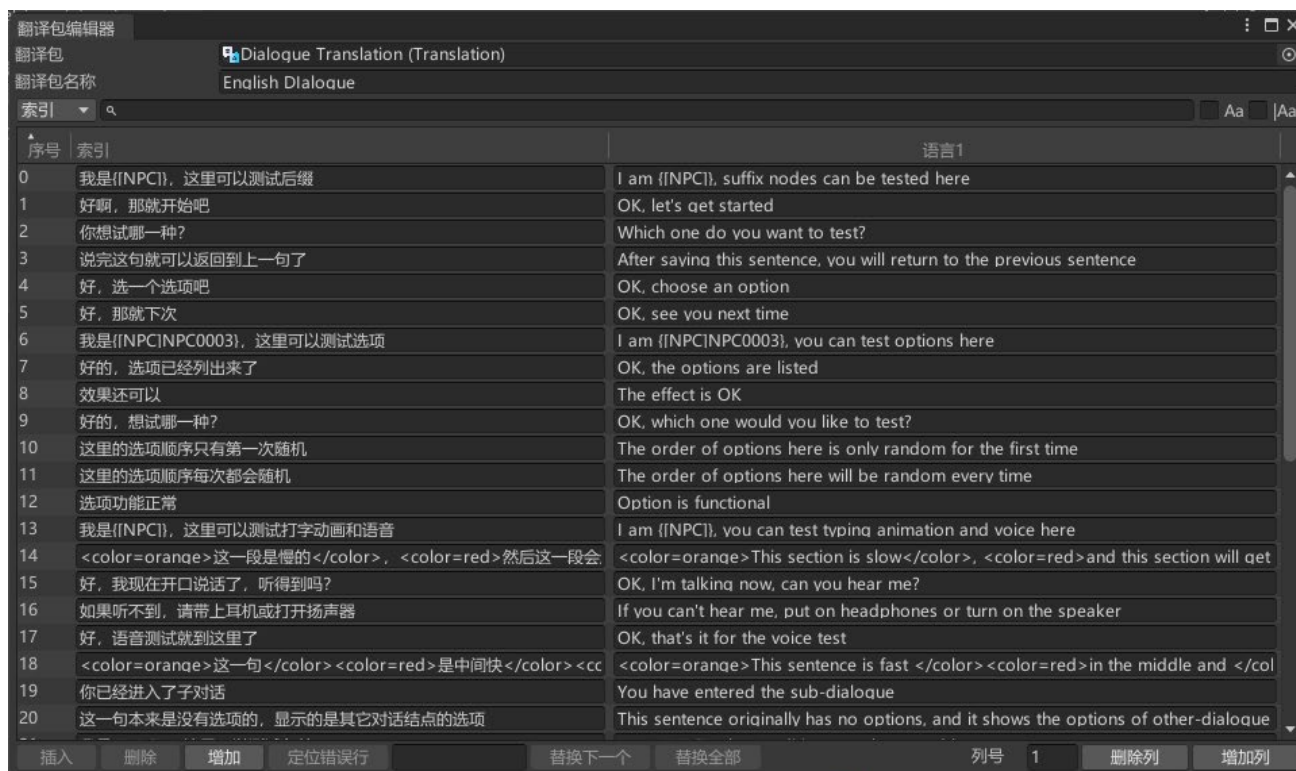
由编辑器使用的成员如下：

编辑器成员	说明
字段 note	这个分组的备注

第三节 翻译包的编辑

3.1 翻译包编辑器

翻译包编辑器可以像 Excel 表格那样编辑翻译包的映射，如下图所示：



其中，“插入”按钮会在当前选择的行下面插入新的一行，“增加”按钮则会在末尾追加新的一行；搜索框左侧的下拉菜单可以改变要检索的列，在进行搜索时，无法进行增删；右下角的“删除列”按钮可以删除它前面指出的列，要注意的是，列号从 0 开始，不包括序号列，比如此时的列号 1 代表的是“语言 1”这一列；拖拽列表项到新位置可以移动列表项；点击序号列的表头可切换升序或降序排序，当按降序排序时，无法使用拖拽功能。

如何打开翻译包编辑器：

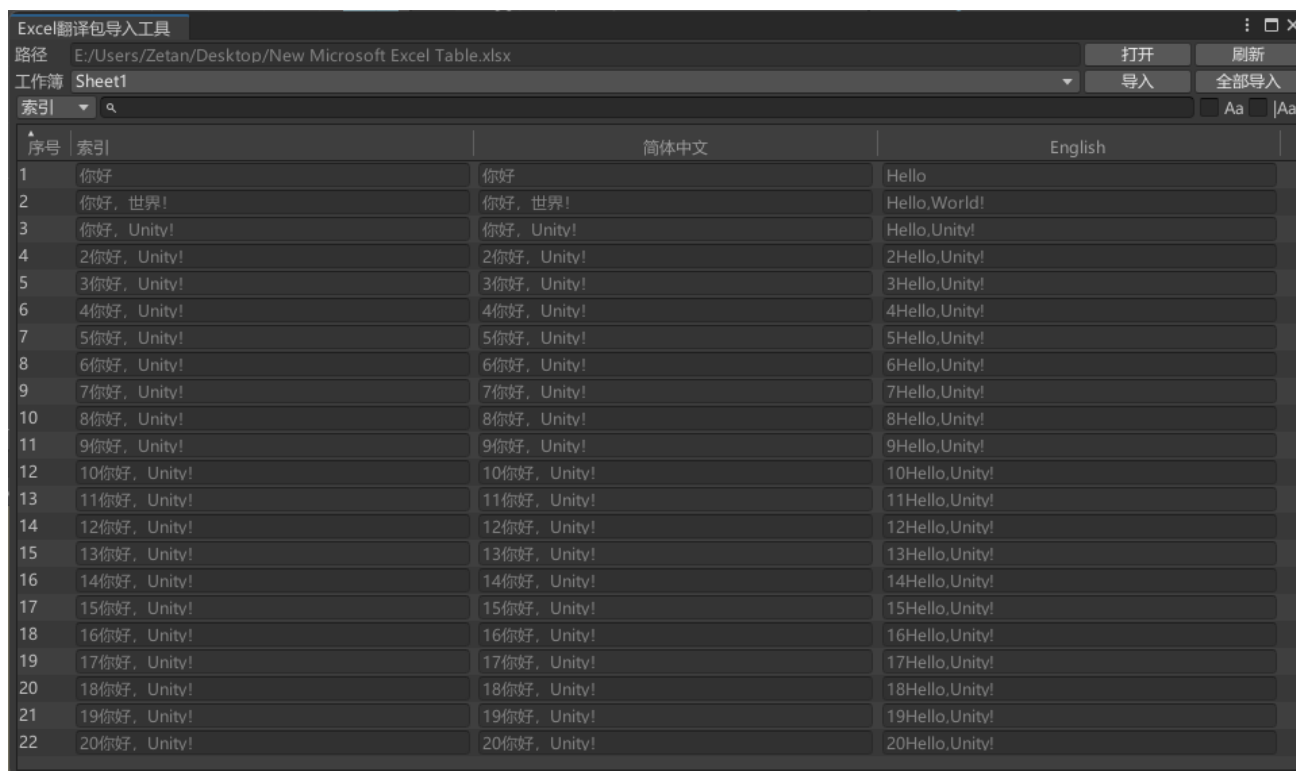
双击翻译包资源文件，或者点击它检查器右上角的打开按钮，或者从 Unity 上方的工具栏中打开，如图所示：



可同时打开多个翻译包编辑器。

3.2 用 Excel 编辑翻译映射

翻译包支持从 Excel 文件导入，在 Excel 中编辑翻译内容会更加方便。不过，目前暂时不支持把已有的翻译包导出到 Excel 文件中，用上面的翻译包编辑器还是用 Excel 来编辑翻译映射，请自行选择吧。在导入 Excel 文件时，会预览里面的内容，如图所示：



序号	索引	简体中文	English
1	你好	你好	Hello
2	你好，世界！	你好，世界！	Hello,World!
3	你好，Unity!	你好，Unity!	Hello,Unity!
4	2你好，Unity!	2你好，Unity!	2Hello,Unity!
5	3你好，Unity!	3你好，Unity!	3Hello,Unity!
6	4你好，Unity!	4你好，Unity!	4Hello,Unity!
7	5你好，Unity!	5你好，Unity!	5Hello,Unity!
8	6你好，Unity!	6你好，Unity!	6Hello,Unity!
9	7你好，Unity!	7你好，Unity!	7Hello,Unity!
10	8你好，Unity!	8你好，Unity!	8Hello,Unity!
11	9你好，Unity!	9你好，Unity!	9Hello,Unity!
12	10你好，Unity!	10你好，Unity!	10Hello,Unity!
13	11你好，Unity!	11你好，Unity!	11Hello,Unity!
14	12你好，Unity!	12你好，Unity!	12Hello,Unity!
15	13你好，Unity!	13你好，Unity!	13Hello,Unity!
16	14你好，Unity!	14你好，Unity!	14Hello,Unity!
17	15你好，Unity!	15你好，Unity!	15Hello,Unity!
18	16你好，Unity!	16你好，Unity!	16Hello,Unity!
19	17你好，Unity!	17你好，Unity!	17Hello,Unity!
20	18你好，Unity!	18你好，Unity!	18Hello,Unity!
21	19你好，Unity!	19你好，Unity!	19Hello,Unity!
22	20你好，Unity!	20你好，Unity!	20Hello,Unity!

其中，一个工作簿对应一个翻译包，“导入”按钮只会导入当前工作簿的内容，“全部导入”则会依次导入所有工作簿。

如何打开 Excel 翻译包导入工具：

需从 Unity 上方的工具栏中打开，如图所示：



Excel 工作簿的格式应该包含表头，如下图所示：

	A	B	C
1	索引	简体中文	English
2	你好	你好	Hello
3	你好，世界！	你好，世界！	Hello,World!
4	你好，Unity!	你好，Unity!	Hello,Unity!
5	从小到大	先插入指定位置	然后后面的依次后移一位
6	带娃哇额	打的滴滴	打撒大的滴滴

第五节 翻译器

翻译器，顾名思义，是把翻译映射中的索引翻译成对应语言的翻译内容的一个类，它是一个静态类，在载入游戏后或者切换语言时都会进行一次初始化，把对应语言的翻译内容缓存起来。所以翻译器并不是实时的，哪怕翻译包的内容修改了，也得等到下次切换语言才刷新，之所以这样做是因为我们一般不会在游戏运行时去对翻译包进行增删改，而且对翻译包的增删改是只能在编辑器模式完成的。

翻译器类是 `Language`，它的成员如下：

运行时成员	说明
静态字段 <code>cache</code>	翻译映射缓存
静态字段 <code>languageIndex</code>	当前语言的下标
静态属性 <code>LanguageIndex</code>	<code>languageIndex</code> 的关联属性
静态字段 <code>languageIndexOffset</code>	翻译器在索引 TranslationItem.Values 时下标的偏移量，即 <code>LanguageIndex</code> 应减去的数值
静态属性 <code>LanguageIndexOffset</code>	<code>languageIndexOffset</code> 的关联属性
静态事件 <code>Action OnLanguageChanged</code>	当 <code>languageIndex</code> 或 <code>languageIndexOffset</code> 的数值改变时调用
静态方法 <code>void Init()</code>	对翻译器进行初始化，生成新的缓存
静态方法 <code>string Translate (string, List<Dictionary<string, string>>)</code>	内部翻译方法，通过传入的映射集合逐个查找索引指向的翻译内容
静态方法 <code>string Tr(string, string)</code>	在给定选择器所对应的映射集合中查找翻译给定内容，其中 <code>Tr</code> 是对 <code>Translate</code> 的缩写
静态方法 <code>string Tr (string, string, params object[])</code>	按格式翻译，像 “ <code>string string.Format(string, string)</code> ” 那样
静态方法 <code>string[] TrM (string, string, params string[])</code>	批量翻译
静态方法 <code>string[] TrM(string, string[])</code>	也是批量翻译，解决上一个批量翻译方法无法传入数组的问题
静态方法 <code>string Tr(Translation, int, string)</code>	直接使用给定翻译包中的指定语言的翻译内容进行翻译
静态方法 <code>string Tr (Translation, int, string, params object[])</code>	参考上文
静态方法 <code>string[] TrM (Translation, int, string, params string[])</code>	参考上文
静态方法 <code>string[] TrM(Translation, int, string[])</code>	参考上文
静态方法 <code>string Tr(Translation, string)</code>	直接使用给定翻译包中的第一个语言的翻译内容进行翻译
静态方法 <code>string Tr (Translation, string, params object[])</code>	参考上文
静态方法 <code>string[] TrM (Translation, string, params string[])</code>	参考上文
静态方法 <code>string[] TrM(Translation, string[])</code>	参考上文

翻译器还有一个名称缩写的类，直接叫做 `L`，比如在进行翻译时，使用 “`L.Tr("选择器", "索引")`” 等同于 “`Language.Tr("选择器", "索引")`”，节省代码篇幅。

第六节 多语言文本组件

6.1 静态多语言文本

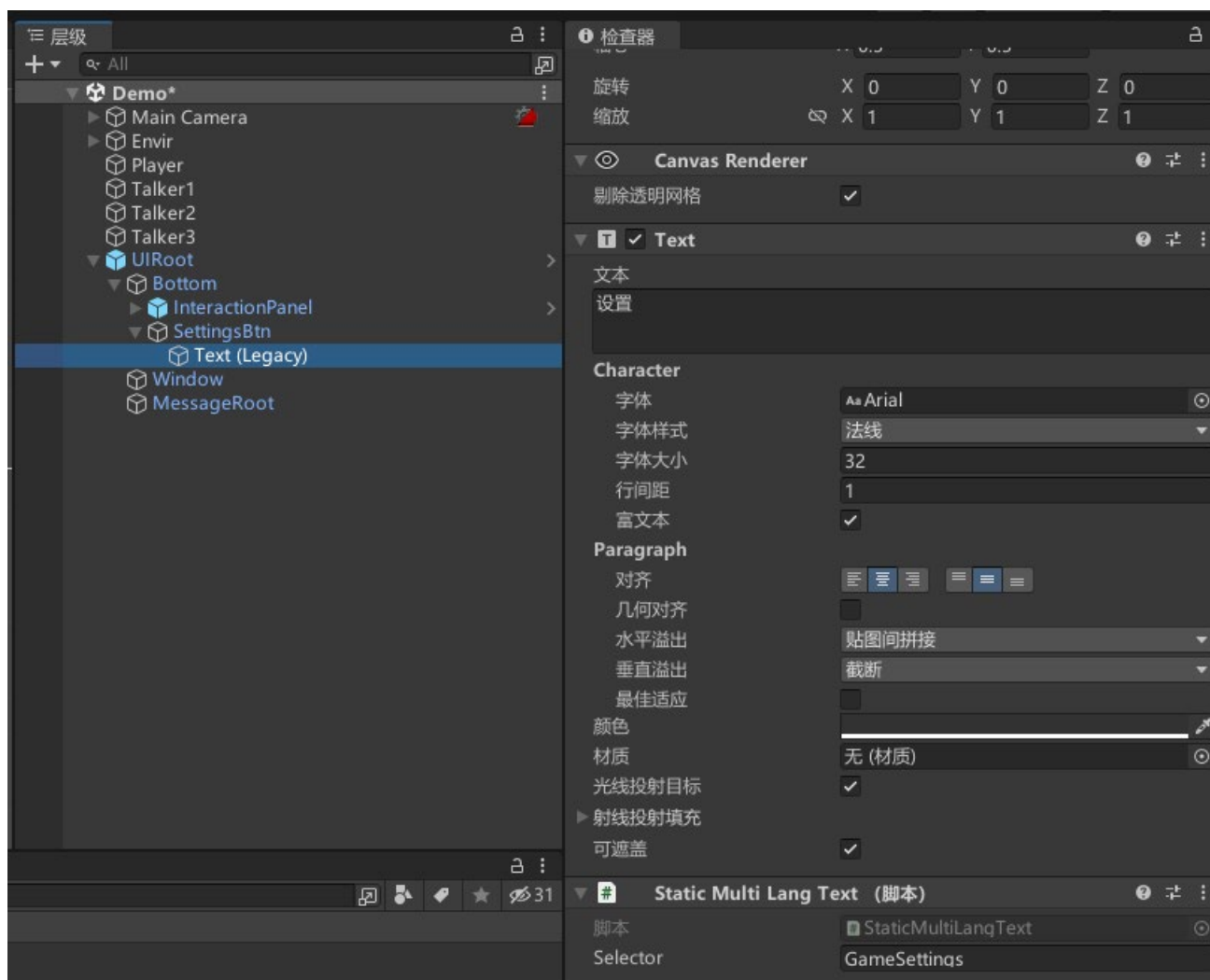
静态文本是指 UI 上面不会被其它代码修改的静态文本，比如背包窗口的标题“背包”之类的就是静态的，而非静态的就比如对话框的对话文本，那个会被频繁修改。静态多语言文本通过填写选择器来对静态的原始文本进行翻译，它需要 Text 组件的配合。

静态多语言文本类是 `StaticMultilingualText`，继承自 `MonoBehaviour`，它的成员为：

运行时成员	说明
字段 <code>selector</code>	翻译包选择器
字段 <code>component</code>	引用的 Text 组件
字段 <code>original</code>	原始文本
方法 <code>void Awake()</code>	由 Unity 调用，初始化文本，并监听语言变更事件
方法 <code>void SetText(string)</code>	更新原始文本并刷新翻译
方法 <code>void SetSelector(string)</code>	更新选择器并刷新翻译

如何使用：

把静态多语言文本组件添加到文本对象上然后设置选择器即可，如图所示：



6.2 多语言文本

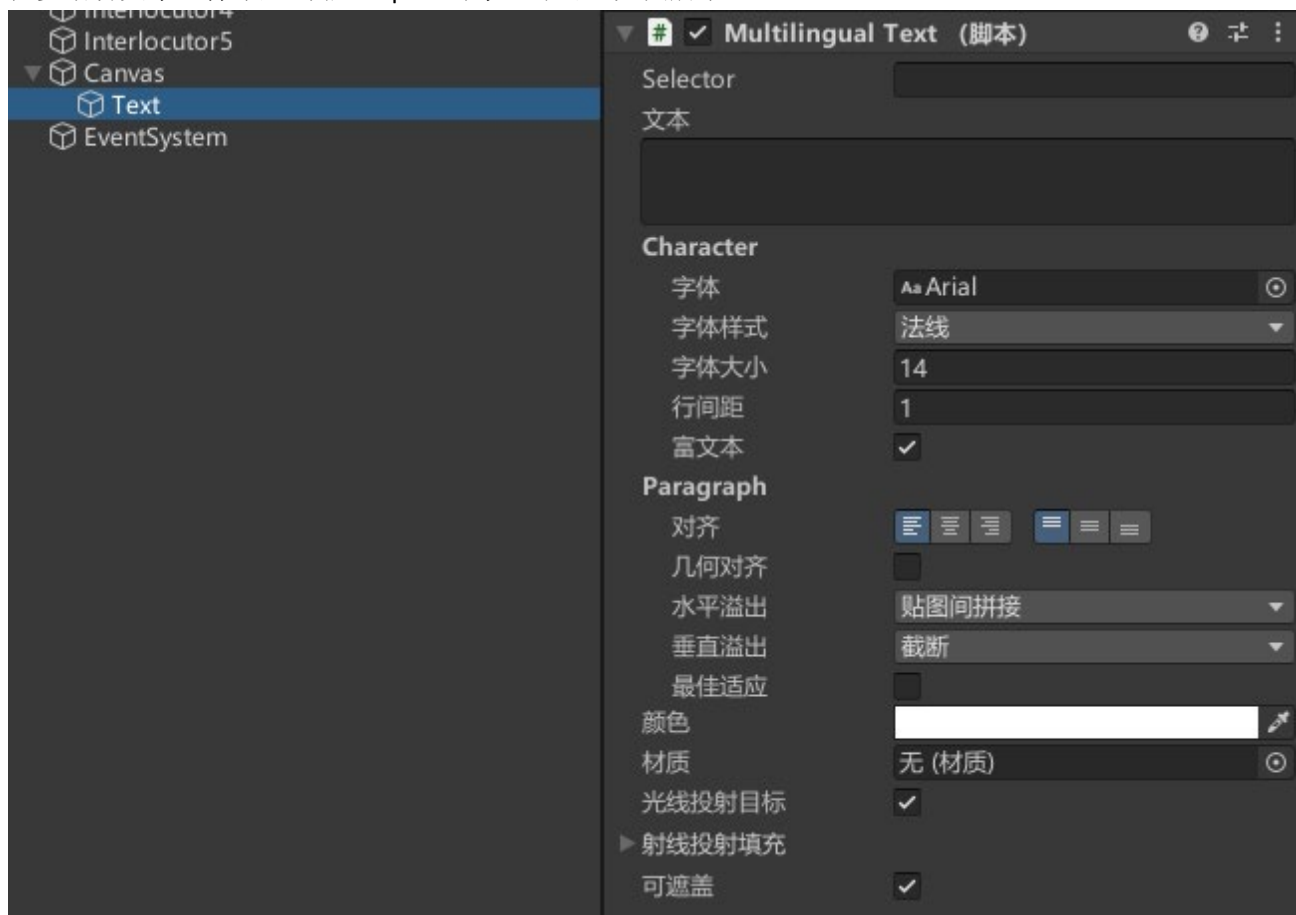
多语言文本，与上文的静态多语言文本功能类似，不过它是直接扩展 **Text** 组件，可以直接替代它来使用。要注意的是，多语言文本只适用于需要在设置原始文本时自动翻译的情景，而原始文本已经是翻译过的文本时，就没必要再使用这个组件了，这样就是多重翻译了。

多语言文本类是 **MultilingualText**，继承自 **Text**，它的成员如下：

运行时成员	说明
字段 m_selector	翻译包选择器
属性 selector	m_selector 的关联属性
属性 text	Text.text 属性的重写，写入时会先进行自动翻译才写入
字段 cache	文本缓存
字段 argsCache	按格式设置文本时，其参数的缓存
方法 void Awake()	由 Unity 调用，初始化文本，并监听语言变更事件
方法 void SetText(string)	设置文本
方法 void SetText(string, params object[])	按格式设置文本

如何使用：

把多语言文本组件添加到非 **Graphic** 对象上即可，如图所示：



第五章 存档系统

第一节 基本概念

存档是把游戏中需要存档的对象的重要数据转成可序列化的数据后保存到本地，然后在需要的时候重新读入并将这些数据还原到指定对象中。存档系统由存档数据和存档管理器组成。

第二节 组成要素

2.1 存档数据

存档数据是通用型数据的一个子类，无非就是预先存入了一些必要数据。

存档数据类是 `SaveData`，继承自 `GenericData`，它的成员如下：

运行时成员	说明
构造函数 <code>SaveData(string)</code>	传入存档版本的构造函数

2.2 存档管理器

存档管理器，顾名思义，就是用来对游戏进度进行存档和读档的，是一个静态类。如果需要的话，存档管理器还可以选择存档槽位。

存档管理器类是 `SaveManager`，它的成员如下：

运行时成员	说明
常量 <code>defaultDataName</code>	默认存档文件名
静态字段 <code>dataName</code>	用户存档文件名
静态属性 <code>DataName</code>	实际存档文件名，当用户文件名为空时使用默认的
常量 <code>defaultEncryptionKey</code>	默认存档加密密钥
静态字段 <code>encryptionKey</code>	用户存档加密密钥
静态属性 <code>EncryptionKey</code>	实际存档加密密钥，当用户密钥为空或长度不对时（16 位或 32 位）使用默认的
静态属性 <code>IsSaving</code>	是否正在存档
静态属性 <code>IsLoading</code>	是否正在读档
静态事件 <code>Action OnSaveCompletely</code>	存档成功时触发
静态事件 <code>Action OnLoadCompletely</code>	读档成功时触发
静态方法 <code>bool Save(int)</code>	将进度存入指定槽位的存档
静态方法 <code>bool Load(int)</code>	载入指定槽位的存档的进度

第三节 相关特性

3.1 存档方法特性

把存档方法特性加在静态方法上面，在存档时就会调用这个方法将进度写入存档数据，这个静态方法必须只有一个存档数据类参数。

存档方法特性类是 `SaveMethodAttribute`，继承自 `Attribute`，它的成员如下：

运行时成员	说明
字段 <code>priority</code>	调用优先级
构造函数 <code>SaveMethodAttribute(int)</code>	传入优先级的构造函数
静态方法 <code>void SaveAll(SaveData)</code>	按优先级依次调用带有此标签的存档方法，让它们将各自系统的进度写入存档数据，由存档管理器调用

3.2 读档方法特性

把读档方法特性加在静态方法上面，在读档时就会调用这个方法载入存档数据的进度，这个静态方法必须只有一个存档数据类参数。

读档方法特性类是 `LoadMethodAttribute`，继承自 `Attribute`，它的成员如下：

运行时成员	说明
字段 <code>priority</code>	调用优先级
构造函数 <code>LoadMethodAttribute(int)</code>	传入优先级的构造函数
静态方法 <code>void LoadAll(SaveData)</code>	按优先级依次调用带有此标签的度档方法，把存档数据的进度载入到各自的系统中，由存档管理器调用

第六章 游戏界面

第一节 基本概念

本插件附带的窗口系统是基于类型的，它以预制件为基础来管理游戏中的界面的生成，由各个窗口类和窗口管理器组成。因为是基于类型的，所以游戏窗口与它的窗口类是一一对应的关系，一个窗口类只对应一个窗口 `GameObject`，这样窗口管理器才能正确地打开、关闭或查找给定类型的游戏窗口。

第二节 窗口系统

2.1 窗口基类

窗口基类，顾名思义，是游戏中窗口界面的基类，它利用 `CanvasGroup` 组件的 `alpha` 和 `blocksRaycasts` 这两个属性来实现游戏窗口的打开和关闭，`alpha` 设为 1、`blocksRaycasts` 设为 `True` 表示打开，`alpha` 设为 0、`blocksRaycasts` 设为 `False` 表示关闭。

窗口类是 `Window`，继承自 `MonoBehaviour`，[IFadeAble](#)，它的成员如下：

运行时成员	说明
常量 <code>WindowStateChanged</code>	全局消息的类型，用于发布和监听消息
字段 <code>animated</code>	是否启用窗口开关时的淡入淡出动画，默认为 <code>True</code>
字段 <code>duration</code>	淡入淡出动画的持续时间，默认为 0.05 秒
字段 <code>content</code>	用来实现窗口打开和关闭的 <code>CanvasGroup</code> 组件
字段 <code>closeButton</code>	窗口关闭按钮，可为空
字段 <code>windowCanvas</code>	用来实现窗口排序的 <code>Canvas</code> 组件
事件 <code>OnClosed</code>	窗口关闭事件，每次绑定后，仅在此窗口下次关闭时触发一次
属性 <code>HideOnAwake</code>	窗口的默认初始状态是否为关闭，默认为 <code>True</code>
属性 <code>SortingOrder</code>	一个只写属性，用来设置窗口的显示顺序
属性 <code>IsOpen</code>	窗口是否处于打开状态
属性 <code>IsHidden</code>	窗口是否处于隐藏状态
属性 <code>IFadeAble.MonoBehaviour</code>	此属性引用窗口自身，用于 <code>IFadeAble</code> 接口
属性 <code>IFadeAble.FadeTarget</code>	此属性引用 <code>content</code> ，用于 <code>IFadeAble</code> 接口
属性 <code>IFadeAble.FadeCoroutine</code>	此属性用于缓存携程，用于 <code>IFadeAble</code> 接口
属性 <code>LanguageSelector</code>	由多语言系统使用的翻译包筛选器，详见多语言系统
方法 <code>bool Open(params object[])</code>	打开窗口
方法 <code>bool Close(params object[])</code>	关闭窗口
方法 <code>bool Hide(bool, params object[])</code>	设置窗口的隐藏状态，仅在 <code>IsOpen</code> 为 <code>True</code> 时有效
方法 <code>bool OnOpen(params object[])</code>	供子类重写的方法，在此处实现窗口打开逻辑
方法 <code>void OnCompletelyOpened()</code>	窗口完全淡入时调用此方法，若无动画则在打开时立即调用
方法 <code>bool OnClose(params object[])</code>	供子类重写的方法，在此处实现窗口关闭逻辑
方法 <code>void OnCompletelyClosed()</code>	窗口完全淡出时调用此方法，若无动画则在关闭时立即调用

方法 <code>bool OnHide(bool, params object[])</code>	供子类重写的方法，在此处实现窗口隐藏或取消隐藏的逻辑
方法 <code>void OnAwake()</code>	供子类重写的方法，默认为空
方法 <code>void OnDestroy_()</code>	供子类重写的方法，默认为空
方法 <code>void RegisterNotification()</code>	供子类重写的方法，用于订阅消息来刷新窗口内容，调用 <code>void Awake()</code> 时调用，默认为空
方法 <code>void UnregisterNotification()</code>	用于取消订阅消息
方法 <code>void Awake()</code>	由 Unity 调用，初始化窗口
方法 <code>void OnDestroy()</code>	由 Unity 调用，取消订阅消息
方法 <code>string Tr(string)</code>	多语言系统文本翻译的快捷方式
方法 <code>string Tr(string, params object[])</code>	多语言系统文本按格式翻译的快捷方式
方法 <code>string[] TrM(string, params string[])</code>	多语言系统文本批量翻译的快捷方式
方法 <code>string[] TrM(string[])</code>	多语言系统文本批量翻译的快捷方式
静态方法 <code>static bool IsName<T>(string)</code> where T : Window	判断所给的名称是不是 T 类窗口的名称
静态方法 <code>static bool IsType<T>(Type)</code> where T : Window	判断所给的类型是不是 T 类窗口的类型

由编辑器使用的成员如下：

编辑器成员	说明
方法 <code>void OnDrawGizmosSelected()</code>	用空心白色矩形绘制 <code>content</code> 的边界 Gizmos
静态方法 <code>void CreateUI()</code>	创建一个空的窗口 UI
静态方法 <code>bool CanCreateUI()</code>	判断可否创建空的窗口 UI

2.2 窗口预制件集合

游戏窗口可利用 Unity 的预制件系统实现按需生成，这就需要一个列表来存放这些窗口预制件，这就是窗口预制件集合类，它是一种资源文件。可利用这个类实现 UI 主题切换。

窗口预制件集合类是 `WindowPrefabs`，继承自 `ScriptableObject`，它的成员如下：

运行时成员	说明
字段 <code>windows</code>	窗口预制件列表
属性 <code>Windows</code>	<code>windows</code> 的关联只读属性
属性 <code>Instance</code>	窗口预制件集合单例
方法 <code>Window FindWindowOfType(Type)</code>	查找指定类型窗口的预制件

2.3 窗口管理器

窗口管理器是一个静态类，用来管理窗口的打开、关闭、隐藏等。

窗口管理器类是 `WindowManager`，它的成员如下：

运行时成员	说明
静态字段 <code>startSortingOrder</code>	窗口排序层的初始层数
静态属性 <code>StartSortingOrder</code>	<code>startSortingOrder</code> 的关联属性，写入此属性时将刷新已打开窗口的排序层
静态字段 <code>windowsContainer</code>	放置窗口的父对象
静态属性 <code>WindowsContainer</code>	<code>windowsContainer</code> 的关联属性

静态属性 Count	已打开的窗口的数量，不包括原本就是打开状态的
静态事件 Action OnCloseAll	关闭所有窗口时调用
静态事件 Action<bool> OnHideAll	隐藏或取消隐藏所有窗口时调用
静态字段 isContainerAutoCreated	windowsContainer 是否为自动创建的，如果是，则设置新 windowsContainer 的时候会销毁原来自动创建的
静态字段 isHidingAll	是否正在隐藏所有窗口，用于防止错误记录窗口真正的隐藏状态
静态字段 caches	缓存的窗口，减少查找窗口的开销
静态字段 openedWindows	已打开的窗口，是一个变相的栈
静态字段 hiddenStates	窗口的隐藏状态，由 void HideAll(bool)系列方法使用
静态方法 Window OpenWindow (string, params object[])	打开指定名称的窗口，成功打开时，返回该窗口
静态方法 Window OpenWindow (Type, params object[])	打开指定类型的窗口，成功打开时，返回该窗口
静态方法 T OpenWindow<T>(params object[]) where T : Window	打开 T 类型的窗口，成功打开时，返回该窗口
静态方法 void OpenOrCloseWindow(string)	打开或关闭指定名称的窗口
静态方法 void OpenOrCloseWindow(Type)	打开或关闭指定类型的窗口
静态方法 void OpenOrCloseWindow<T>() where T : Window	打开或关闭 T 类型的窗口
静态方法 Window OpenOrUnhideWindow(string)	打开或取消隐藏指定名称的窗口并返回该窗口
静态方法 Window OpenOrUnhideWindow(Type)	打开或取消隐藏指定类型的窗口并返回该窗口
静态方法 T OpenOrUnhideWindow<T>() where T : Window	打开或取消隐藏 T 类型的窗口并返回该窗口
静态方法 bool CloseWindow (string, params object[])	关闭指定名称的窗口，返回值表示是否成功关闭
静态方法 bool CloseWindow (Type, params object[])	关闭指定类型的窗口，返回值表示是否成功关闭
静态方法 bool CloseWindow<T> (params object[]) where T : Window	关闭 T 类型的窗口，返回值表示是否成功关闭
静态方法 void CloseTop()	关闭已打开的最顶层的可见窗口
静态方法 void CloseAll()	关闭所有已打开的窗口
静态方法 void CloseAllExceptName (params string[])	关闭除了指定名称之外的所有窗口
静态方法 void CloseAllExceptType (params Type[])	关闭除了指定类型之外的所有窗口
静态方法 void CloseAllExcept (params Window[])	关闭除了指定窗口之外的所有窗口
静态方法 void HideAll(bool)	隐藏或取消隐藏所有窗口
静态方法 void HideAllExceptName (bool, params string[])	隐藏或取消隐藏除了指定名称之外的所有窗口

静态方法 void HideAllExceptType (bool, params Type[])	隐藏或取消隐藏除了指定类型之外的所有窗口
静态方法 void HideAllExcept (bool, params Window[])	隐藏或取消隐藏除了指定窗口之外的所有窗口
静态方法 void RecordHiddenState(Window)	记录窗口的隐藏状态，正在隐藏所有窗口时，不记录
静态方法 Window FindWindow(string)	查找指定名称的窗口
静态方法 Window FindWindow(string, bool)	查找指定名称的窗口，可选找不到时创建
静态方法 Window FindWindow(Type)	查找指定类型的窗口
静态方法 Window FindWindow(Type, bool)	查找指定类型的窗口，可选找不到时创建
静态方法 T FindWindow<T>() where T : Window	查找 T 类型的窗口
静态方法 T FindWindow<T>(bool) where T : Window	查找 T 类型的窗口，可选找不到时创建
静态方法 void Cache(Window)	缓存指定窗口
静态方法 bool IsWindowOpen(string)	判断指定名称的窗口是否打开
静态方法 bool IsWindowOpen(Type)	判断指定类型的窗口是否打开
静态方法 bool IsWindowOpen<T>() where T : Window	判断 T 类型的窗口是否打开
静态方法 bool IsWindowOpen (string, out Window)	判断指定名称的窗口是否打开，并传出该窗口
静态方法 bool IsWindowOpen (Type, out Window)	判断指定类型的窗口是否打开，并传出该窗口
静态方法 bool IsWindowOpen<T>(out T) where T : Window	判断 T 类型的窗口是否打开，并传出该窗口
静态方法 bool IsWindowHidden(string)	与打开检查类似
静态方法 bool IsWindowHidden(Type)	与打开检查类似
静态方法 bool IsWindowHidden<T>() where T : Window	与打开检查类似
静态方法 bool IsWindowHidden (string, out Window)	与打开检查类似
静态方法 bool IsWindowHidden (Type, out Window)	与打开检查类似
静态方法 bool IsWindowHidden<T>(out T) where T : Window	与打开检查类似
静态方法 void Push(Window)	将窗口入栈，仅供窗口类的打开方法使用
静态方法 Window Peek()	返回已打开且未隐藏的最顶层窗口
静态方法 Window Top()	返回最顶层窗口
静态方法 void Remove(Window)	将窗口从栈中移除，仅供窗口类的关闭方法使用
静态方法 void Init()	关闭所有窗口，然后初始化窗口管理器

第三节 通用窗口

通用窗口是一种为热更新的设计的窗口，与原来的窗口系统设计理念有些出入，原窗口系统是用类型来识别一个窗口，而通用窗口则是用给定的窗口名称来识别窗口。由于本插件不实现热更新，所以窗口管理器中关于通用窗口的 API 没有列出来，通用窗口类也不会在此详细说明，但它的用法还是差不多的。

第四节 对话窗口

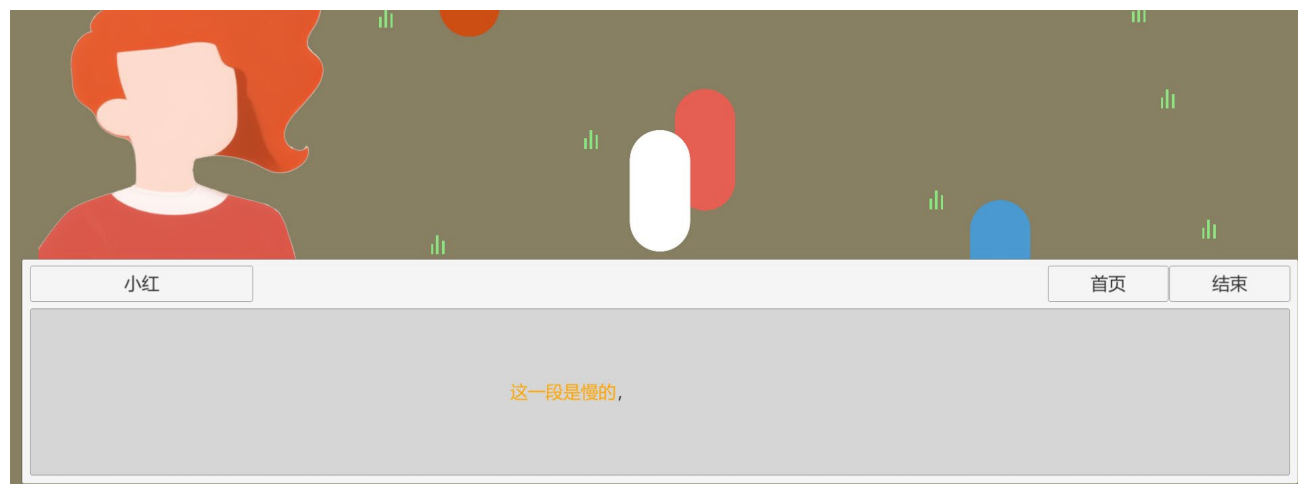
对话窗口把我们设计的对话展现给玩家，它主要由对话人文本区、对话文本区、选项区、肖像区、语音播放器等组成。默认的对对话窗口的皮肤如下图所示：

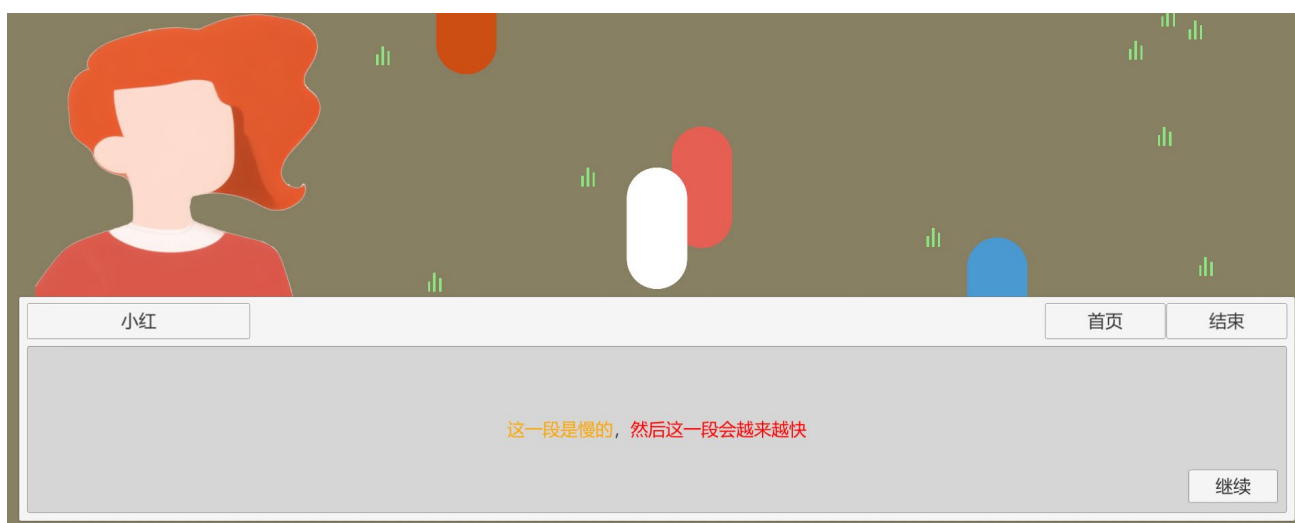


其中，右上角的“首页”按钮可以回到打开对话窗口时的第一句话，仅在通过和对话人交互打开窗口时才显示；“结束”按钮则可以关闭对话窗口，提前结束对话。它们都是可选的，如果不需要完全可以去掉，也不会因此报错。

选项上方的“互动”是选项区的标题，可修改为其它的文字，例如选项列表可作为任务列表，然后标题改为“任务”。

对话文本会有一个打字动画，动画播放结束后才会显示选项，如图所示：





打字动画的播放速度由语句结点的吐字间隔属性控制。还可以点击对话文本，用来跳过打字动画、切换下一句等。

对话窗口类是 `DialogueWindow`，继承自 [InteractionWindow<Interlocutor>](#)，它的成员如下：

运行时成员	说明
字段 <code>nameText</code>	对话人名称文本，请启用它的富文本
字段 <code>contentText</code>	对话内容文本，请启用它的富文本
字段 <code>contentButton</code>	对话文本的按钮组件，可选
字段 <code>homeButton</code>	首页按钮，可选
字段 <code>nextButton</code>	下一句按钮
字段 <code>nextButtonText</code>	下一句按钮的文本
字段 <code>optionArea</code>	选项区的 <code>GameObject</code>
字段 <code>optionTitle</code>	选项区标题的 <code>GameObject</code>
字段 <code>optionTitleText</code>	选项区标题文本
字段 <code>optionList</code>	选项列表
字段 <code>leftPortrait</code>	左侧肖像，当启用了肖像功能时可用
字段 <code>rightPortrait</code>	右侧肖像，当启用了肖像功能时可用
字段 <code>adaptiveSize</code>	肖像大小是否根据肖像 <code>Sprite</code> 进行自适应
字段 <code>voicePlayer</code>	语音片段播放器，当启用了语音功能时可用
字段 <code>skipDelay</code>	打字动画跳过延迟，即前句的打字动画至少显示多久才

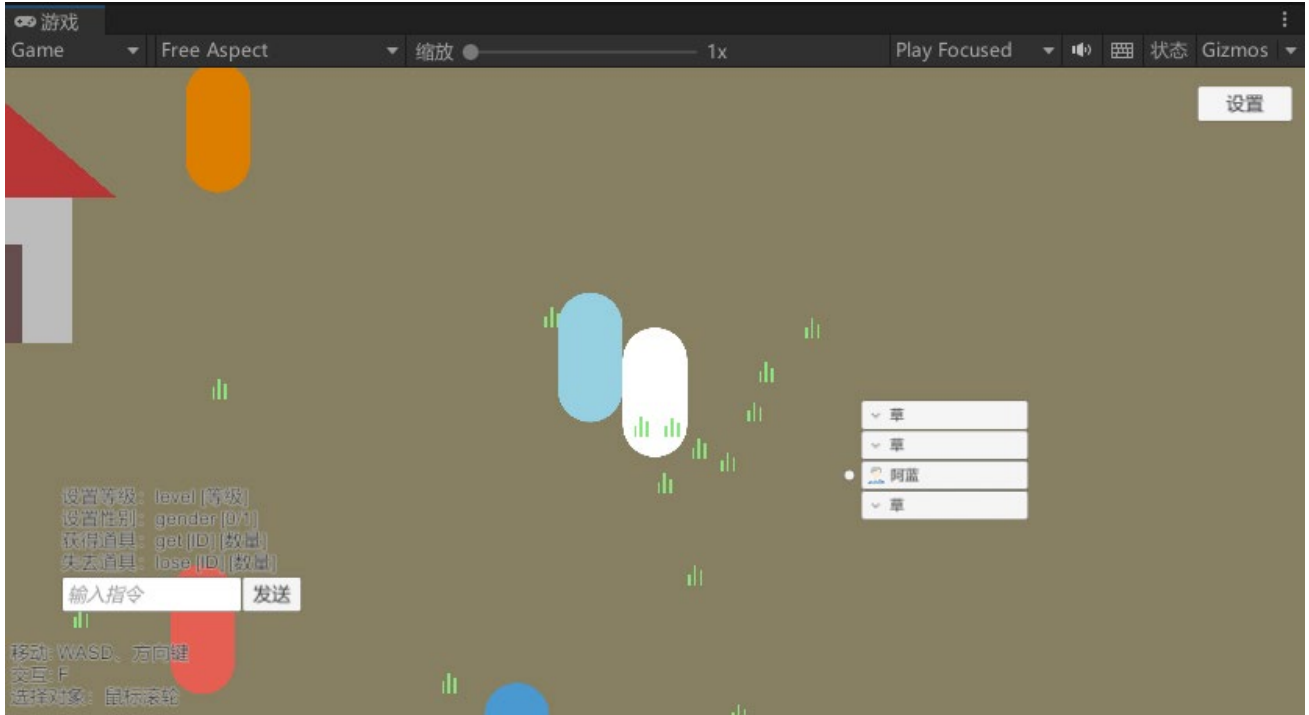
	可以跳过
字段 closeWhenExit	是否在对话结束时关闭对话窗口
字段 handler	对话处理器实例
字段 nextClicked	点击继续按钮时的操作，一般是切换至下一句
属性 Target	实现 InteractionWindow<IInterlocutor >抽象类的抽象属性，引用 handler 的 Interlocutor
属性 LanguageSelector	由多语言系统使用的选择器，这里是“Dialogue”
方法 void OnHandleNode(DialogueNode)	刷新主页按钮和关闭按钮的显隐状态
方法 void OnReachLastSentence()	到达最后一句时的操作
方法 string HandleKeywords(string)	处理所给字符串中的关键字
方法 void RefreshNextButton(Action, string)	设置下一句按钮的功能和标题
方法 void RefreshOptions (List<OptionCallback>, string)	刷新选项列表及选项区标题
方法 void SetPortrait(bool, Sprite)	刷新指定侧的肖像，仅在启用肖像功能时可用
方法 void SetPortraitDark(bool, bool)	让某侧肖像变暗或取消变暗，仅在启用肖像功能时可用
方法 void Next()	用于一键操作，比如只按确认键就能进行跳过打字动画、进入下一句、点击选中选项等操作
方法 void OnContentClick()	点击对话文本时的回调
方法 void NextOption()	选中下一个选项
方法 void PrevOption()	选中上一个选项
方法 void OnAwake()	生成对话处理器实例，设置按钮的点击回调
方法 bool OnInterrupt()	尝试打断以关闭窗口，前提是当前对话可被打断
方法 bool OnOpen_(params object[])	打开窗口之前处理传入的参数
方法 bool OnClose_(params object[])	关闭窗口之前执行清理

因为对话窗口是一种交互窗口，如果游戏中没有可交互对象且不需要交互面板，但仍需要进行对话，那就直接忽略所有交互相关的字段、属性和方法，然后把 Dialogue 资源文件或者 EntryNode 结点作为参数传入来打开对话窗口。

第七章 交互系统

第一节 基本概念

本插件附带的交互系统是和《原神》类似的一种列表式交互系统，由交互对象和交互面板组成。其中，交互面板会把玩家附件的交互对象以列表的形式罗列出来供玩家交互，如下图所示：



第二节 组成要素

2.1 可交互接口

可交互接口，顾名思义，如果希望某种对象可被加入交互面板，就得继承这个接口。

可交互接口是 `IInteractive`，它的成员如下：

运行时成员	说明
属性 <code>Name</code>	可交互对象的名称
属性 <code>Icon</code>	可交互对象的图标
属性 <code>IsInteractive</code>	可否加入交互面板（即交互列表）
属性 <code>Interactable</code>	是否已加入交互面板
方法 <code>bool DoInteract()</code>	在这里写交互逻辑，返回值表示交互是否成功，交互成功时，将从交互面板移出对象
方法 <code>void EndInteraction()</code>	这是一个有具体实现的方法，它用来结束交互，当对应的 <code>InteractionWindow<T></code> 关闭时被调用，也可以在需要结束交互的地方调用
方法 <code>void OnInteractable()</code>	这是有具体实现的方法，但默认为空，可以用显式实现来重写，加入到交互面板时调用

方法 <code>void OnNotInteractable()</code>	这是有具体实现的方法，但默认为空，可以用显式实现来重写，从交互面板移出时调用
方法 <code>void OnEndInteraction()</code>	这是有具体实现的方法，但默认为空，可以用显式实现来重写，调用 <code>void EndInteraction()</code> 时调用
静态方法 <code>void PushToPanel(IInteractive)</code>	尝试将可交互对象加入交互面板
静态方法 <code>void RemoveFromPanel(IInteractive)</code>	尝试将可交互对象从交互面板移出

如何使用：

下面给出一个基于 2D 触发器的可交互对象的示例代码：

```
public class Interactive : MonoBehaviour, IInteractive
{
    [field: SerializeField]
    public string Name { get; set; }

    [field: SerializeField]
    public Sprite Icon { get; set; }

    [field: SerializeField]
    public bool IsInteractive { get; set; }

    bool IInteractive.Interactable { get; set; }

    public bool DoInteract()
    {
        Debug.Log("Hello, world!");
        return false;
    }

    void IInteractive.OnInteractable() => Debug.Log("OnInteractable");
    void IInteractive.OnNotInteractable() => Debug.Log("OnNotInteractable");
    void IInteractive.OnEndInteraction() => Debug.Log("OnEndInteraction");

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player")) IInteractive.PushToPanel(this);
    }
    private void OnTriggerStay2D(Collider2D collision)
    {
        if (collision.CompareTag("Player")) IInteractive.PushToPanel(this);
    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player")) IInteractive.RemoveFromPanel(this);
    }
}
```

2.2 基本可交互对象

基本可交互对象，是一个 `MonoBehaviour` 对象，可用于那些没有父类的可交互 `MonoBehaviour` 对象，是可交互接口的一种具体化。

基本可交互对象类是 `InteractiveBase`，继承自 `MonoBehaviour`，`IInteractive`，它的成员如下：

运行时成员	说明
字段 <code>defaultName</code>	可交互对象的默认名称
属性 <code>Name</code>	引用默认名称，可重写为其它
字段 <code>defaultIcon</code>	可交互对象的默认图标
属性 <code>Icon</code>	引用默认图标，可重写为其它
属性 <code>IsInteractive</code>	供子类重写的抽象属性
属性 <code>IInteractive.Interactable</code>	接口属性的显式实现
方法 <code>bool DoInteract()</code>	供子类重写的抽象方法
方法 <code>void OnDestroy()</code>	由 <code>Unity</code> 调用，会从交互面板中移除自己
方法 <code>void OnDestroy_()</code>	供子类重写的销毁时回调，默认为空
方法 <code>void OnInteractable()</code>	供子类重写的方法
方法 <code>void OnNotInteractable()</code>	供子类重写的方法
方法 <code>void OnEndInteraction()</code>	供子类重写的方法
方法 <code>void IInteractive.OnInteractable()</code>	接口方法的显式实现，调用本身 <code>void OnInteractable()</code>
方法 <code>void IInteractive.OnNotInteractable()</code>	接口方法的显式实现，调用本身 <code>void OnNotInteractable()</code>
方法 <code>void IInteractive.OnEndInteraction()</code>	接口方法的显式实现，调用本身 <code>void OnEndInteraction()</code>

如何使用：

下面给出一个基于 2D 触发器的可交互对象基类的示例代码：

```
public abstract class Interactive2D : InteractiveBase
{
    public bool activated = true;

    protected virtual void OnTriggerEnter2D(Collider2D collision)
    {
        if (activated && collision.CompareTag("Player")) IInteractive.Push(this);
    }

    protected virtual void OnTriggerStay2D(Collider2D collision)
    {
        if (activated && collision.CompareTag("Player")) IInteractive.Push(this);
    }

    protected virtual void OnTriggerExit2D(Collider2D collision)
    {
        if (activated && collision.CompareTag("Player")) IInteractive.Remove(this);
    }
}
```

再给出一个它的子类对象的示例代码：

```

public class Grass : Interactive2D
{
    public override string Name => "Grass";

    public override bool IsInteractive => true;

    public override bool DoInteract()
    {
        Debug.Log(name);
        return false;
    }
}

```

第三节 交互界面

3.1 交互面板

交互面板也就是交互按钮列表，它会在列表下方插入新的交互对象，并保持当前所选的交互对象的选中状态，而不是像《原神》那样，插入新对象时就自动选中新对象，比如烦人的“烹饪”。下图所示就是所谓的交互面板：



交互面板类是 `InteractionPanel`，继承自 [SingletonWindow<InteractionPanel>](#)，它的成员如下：

运行时成员	说明
字段 <code>view</code>	列表视图
字段 <code>buttonPrefab</code>	交互按钮预制件
字段 <code>buttonParent</code>	放置交互按钮的父对象
字段 <code>buttonCacheParent</code>	放置交互按钮对象池缓存的父对象
属性 <code>IsOpen</code>	重写 <code>Window.IsOpen</code> ，返回 <code>True</code>
属性 <code>HideOnAwake</code>	重写 <code>Window.HideOnAwake</code> ，返回 <code>False</code>
属性 <code>CanInteract</code>	能否使用交互按钮（如 <code>F</code> 键）进行交互
属性 <code>CanScroll</code>	选择项能否滚动，比如《原神》那样用滚轮选择交互对象
字段 <code>objects</code>	已加入面板的交互对象
字段 <code>buttons</code>	已加入面板的交互按钮
字段 <code>showStates</code>	面板的隐藏状态栈，用栈结构防止取消隐藏的时机不对
字段 <code>selectedIndex</code>	选中的交互按钮的下标
字段 <code>pool</code>	交互按钮对象池
静态方法 <code>bool Push(IIInteractive)</code>	将交互对象加入交互列表
静态方法 <code>void HidePanelBy(IIInteractive, bool)</code>	通过交互对象隐藏或取消隐藏交互面板
静态方法 <code>bool Remove(IIInteractive)</code>	将交互对象从交换列表中移除

静态方法 void Interact()	与选中的交互对象进行交互
静态方法 void Next()	选择下一个交互对象
静态方法 void Prev()	选择是一个交互对象
方法 void UpdateSelected()	刷新交互按钮的选中状态
方法 void UpdateView()	刷新交互面板，防止选中的按钮被遮挡
方法 void OnAwake()	建立对象池
方法 void RegisterNotification()	监听语言变化事件
方法 void UnregisterNotification()	取消监听语言变化
方法 bool OnOpen(params object[])	交互面板是常开的，这里返回 False
方法 bool OnClose(params object[])	交互面板是常开的，这里返回 False

3.2 交互窗口

交互窗口是专门处理交互对象的窗口的基类，比如对话框就是跟对话人交互的一种交互窗口。

交互窗口类是 InteractionWindow<T>，继承自 Window，泛型参数 T 得是 IInteractive，它的成员如下：

运行时成员	说明
字段 hidePanelWhenInteracting	是否在交互时隐藏交互面板
属性 Target	当前交互对象
方法 void Interrupt()	中断交互
方法 bool OnInterrupt()	供子类重写的方法，尝试中断交互，由 void Interrupt()调用，返回值表示可否中断，默认返回 True
方法 bool OnOpen(params object[])	一个密封的重写方法，处理交互面板的隐藏，然后调用子类的 bool OnOpen_(params object[])
方法 bool OnClose(params object[])	一个密封的重写方法，处理交互面板的隐藏，然后调用子类的 bool OnClose_(params object[])
方法 bool OnOpen_(params object[])	供子类重写的方法，代替 bool OnOpen(params object[])
方法 bool OnClose_(params object[])	供子类重写的方法，代替 bool OnClose(params object[])

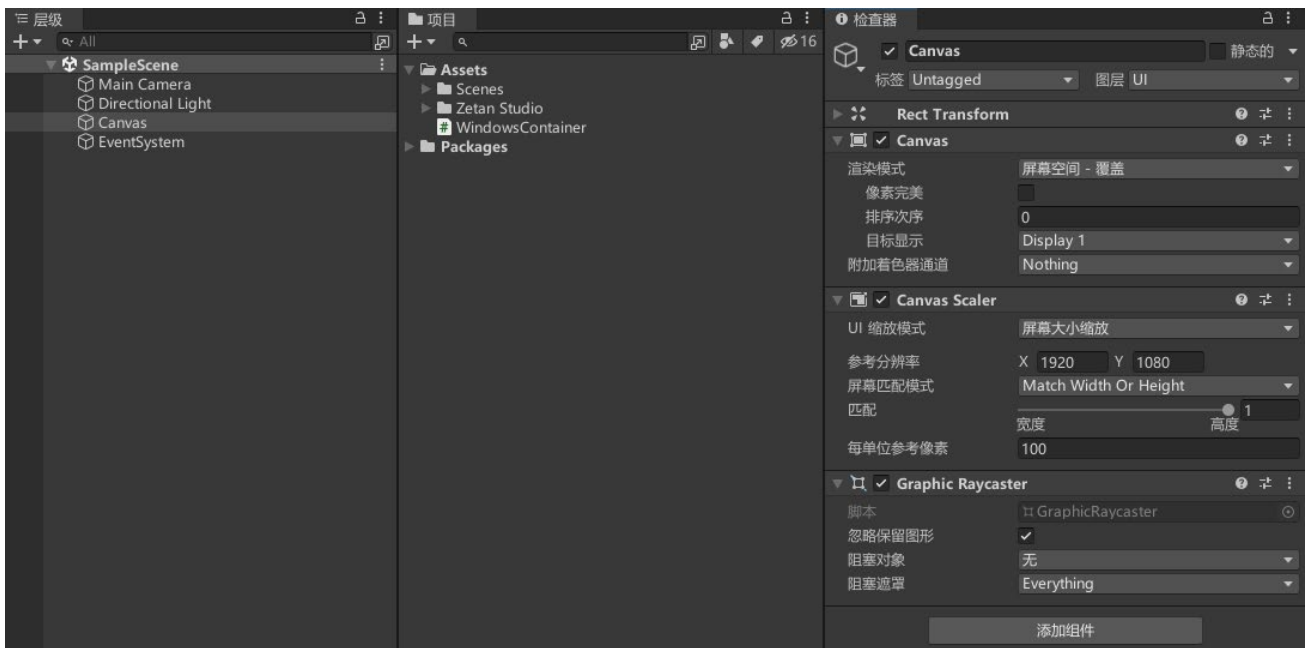
第八章 快速开始

第一节 使用交互系统进行对话

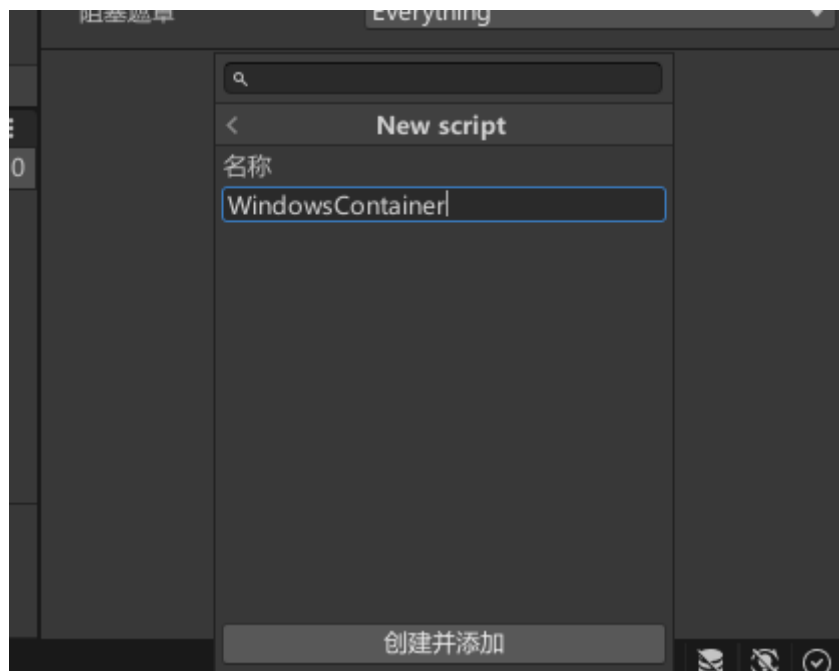
如果使用交互系统进行对话则会比较复杂，相比第二节的其它方式需要亲自编写更多的代码才能正常使用。

1.1 搭建 UI

- 1、首先，我们需要一个 UI 的容器来放置对话窗口，所以我们要先在场景中新建一个画布（Canvas），并调整一下它的 Canvas Scaler 组件的参数，如图所示：



- 2、然后，给画布写一个新脚本，名字随意，如图所示：

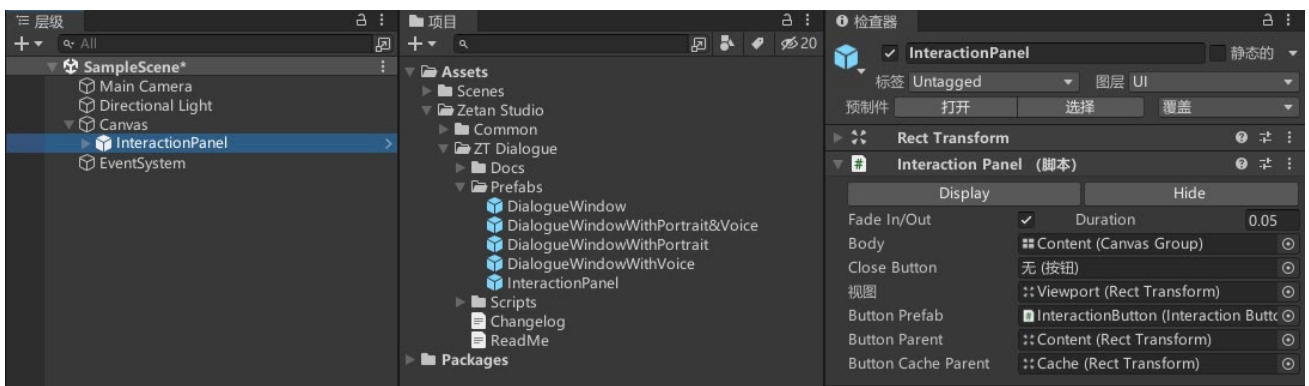


3、编辑上一步新建的脚本如下并保存：

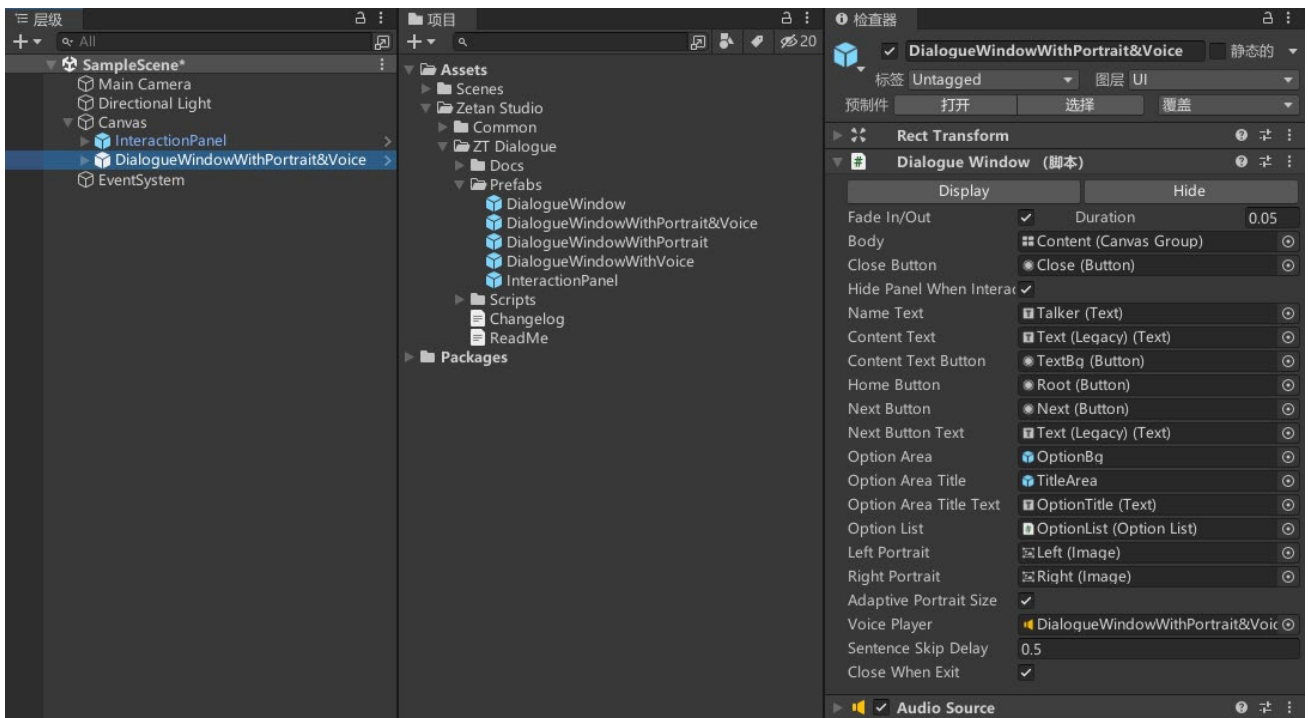
```
using UnityEngine;
using ZetanStudio.UI;

public class WindowsContainer : MonoBehaviour
{
    private void Awake()
    {
        //这一步用于设置 UI 管理器中放置 UI 的容器的单例
        WindowManager.WindowsContainer = transform;
    }
}
```

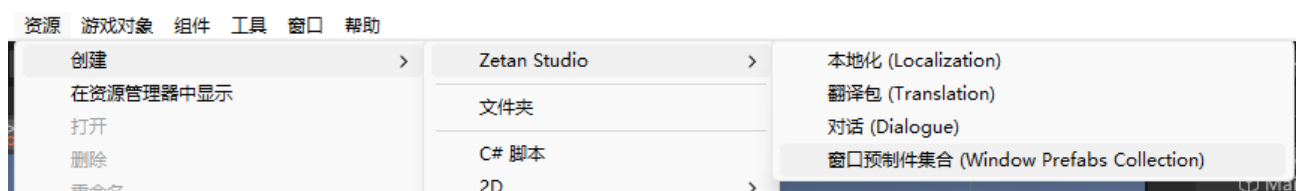
4、接着我们将交互面板预制件拖拽到画布上作为其子对象，如图所示：



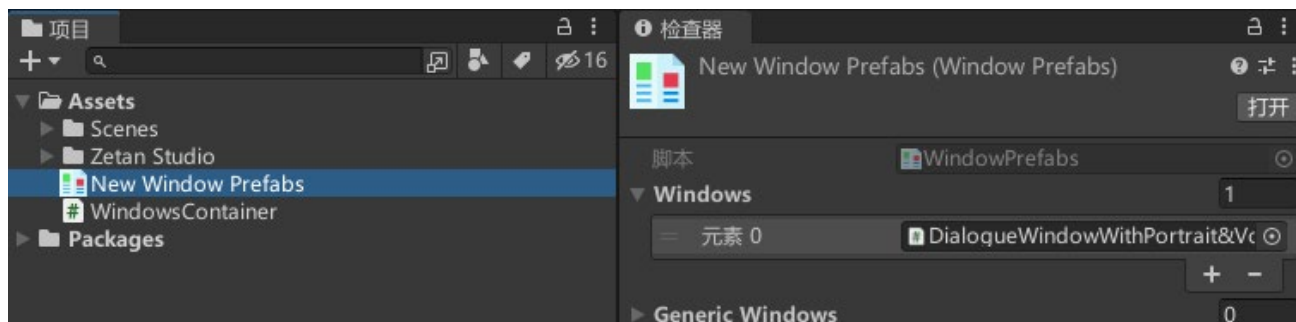
5、接下来我们要放置对话窗口，这里有两种方法。第一种方法是不通过预制件来生成对话窗口，而是直接在场景中放置对话窗口，这时我们只需要将“Zetan Studio/ZT Dialogue/Prefabs”路径下的 DialogueWindow 预制件（这里有多多个 DialogueWindow 预制件，请按需选择其中一个）直接拖拽到画布上作为其子对象，如图所示：



6、第二种方法比较复杂，需要先新建一个窗口预制件集合资源（WindowPrefabs 类资源），如图所示：



7、然后将对话窗口预制件添加到 Windows 列表中，如图所示：



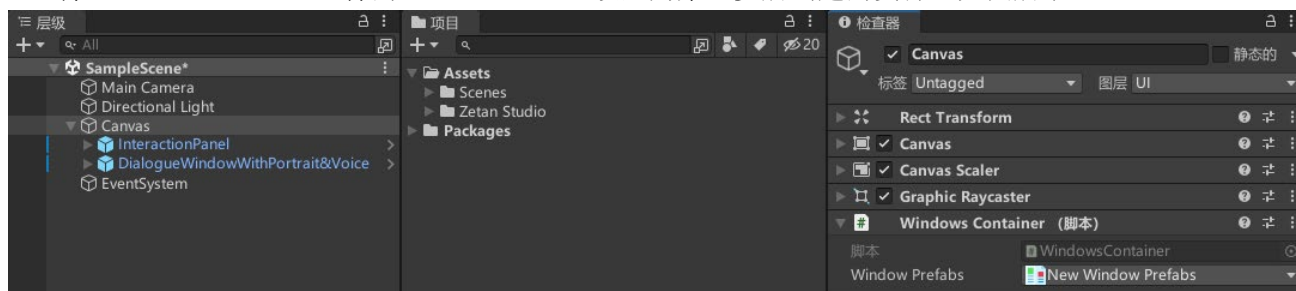
8、接着在第 2 步所建的脚本中加入一个窗口预制件集合参数，并在 Awake 函数中将其作为 WindowPrefabs 类的单例，如下所示：

```
using UnityEngine;
using ZetanStudio.UI;
```

```
public class WindowsContainer : MonoBehaviour
{
    [SerializeField]
    private WindowPrefabs windowPrefabs;

    private void Awake()
    {
        //这一步用于设置窗口预制件集合的单例
        WindowPrefabs.Instance = windowPrefabs;
        //这一步用于设置 UI 管理器中放置 UI 的容器的单例
        WindowManager.WindowsContainer = transform;
    }
}
```

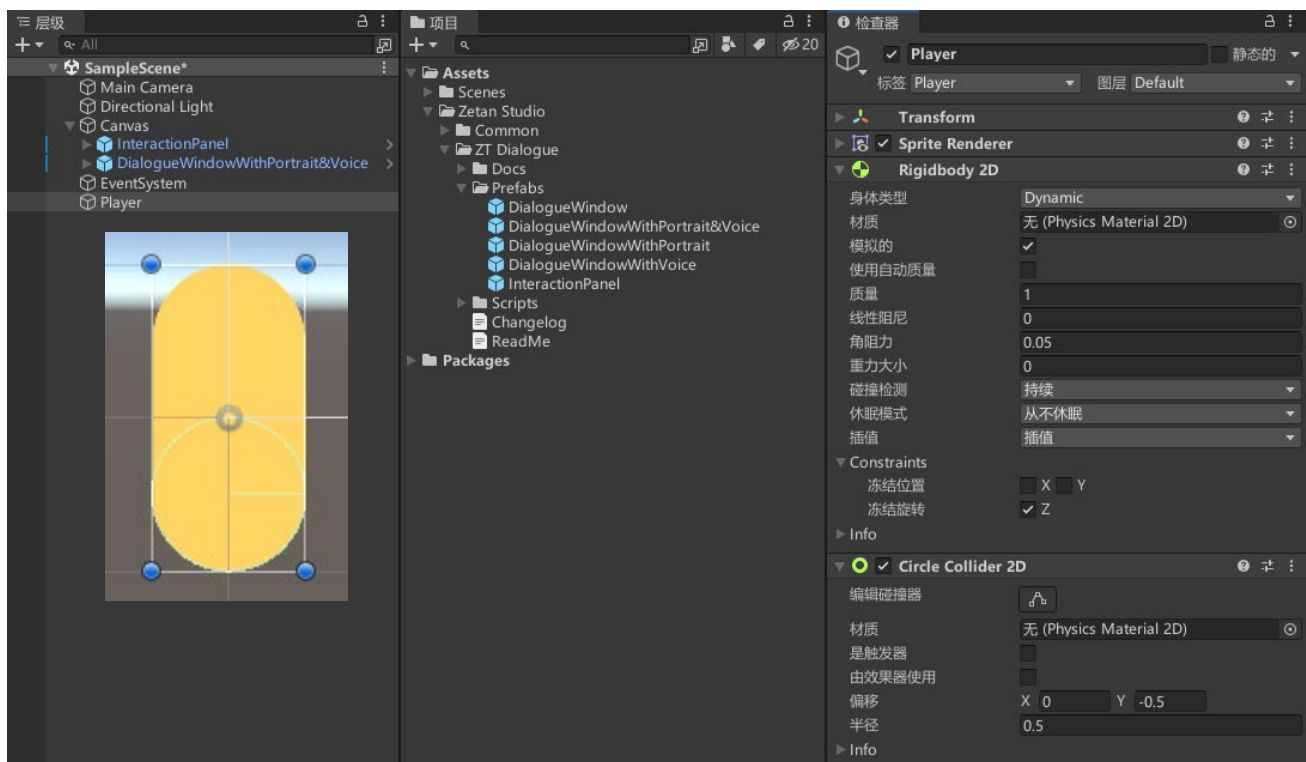
9、将 WindowsContainer 组件的 WindowPrefabs 设置为第 7 步所创建的资源，如图所示：



1.2 添加角色

1、接下来我们开始制作玩家角色。在场景中新建一个 2D 对象，这里我使用 2D 胶囊体，给它加上碰撞器和

2D 刚体，并调整刚体参数，然后将其标签（Tag）设置为 Player，如图所示：



2、接下来给玩家对象添加一个脚本来控制它，如下所示：

using UnityEngine;

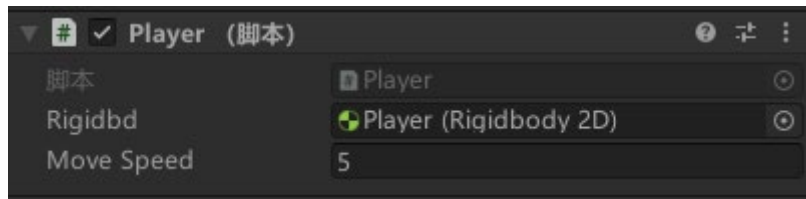
```
public class Player : MonoBehaviour
{
    [SerializeField]
    private Rigidbody2D rigidbd;
    [SerializeField]
    private float moveSpeed = 5f;

    private Vector3 movement;

    private void Update()
    {
        movement = Vector3.ClampMagnitude(new Vector3(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")), 1);
    }

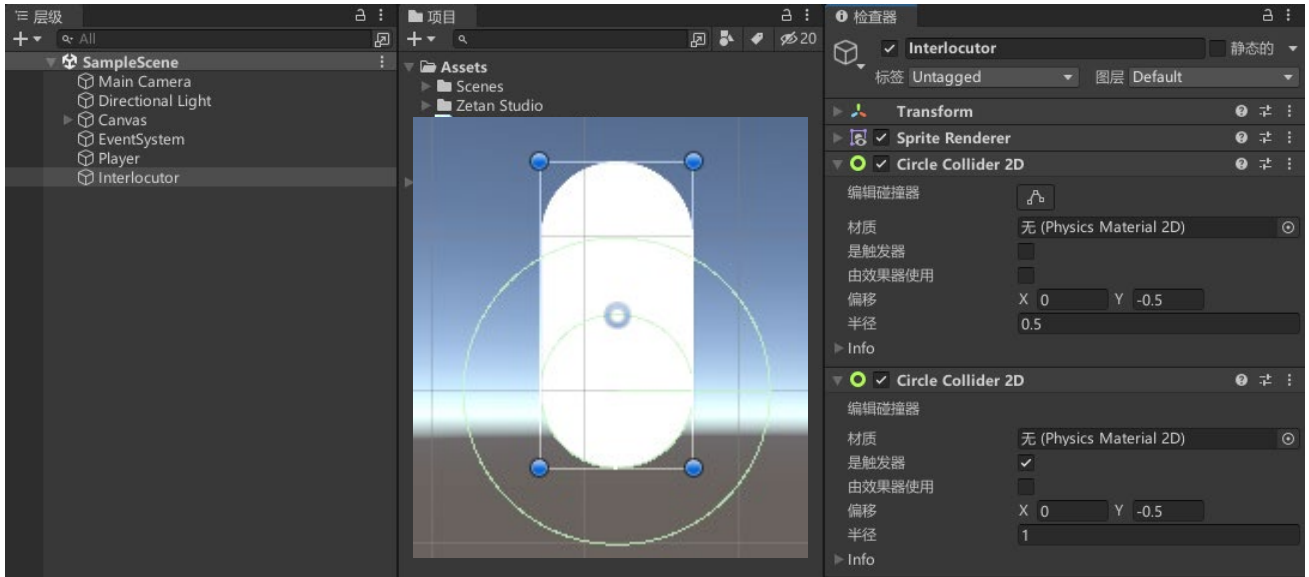
    private void FixedUpdate()
    {
        rigidbd.MovePosition(rigidbd.transform.position + moveSpeed * Time.deltaTime * movement);
    }
}
```

3、将玩家的刚体组件拖拽到上一步所创建组件的 Rigidbody 中来引用它，如图所示：



现在如果我们运行游戏，则会发现可以控制玩家移动了。

4、接下来我们开始制作对话人。继续在场景中新建一个 2D 对象，这里我仍然使用 2D 胶囊体，并让它稍微离玩家对象远一点。然后给对话人加上一个 2D 碰撞器和一个 2D 触发器，如图所示：



要注意点是，触发器的范围应大于碰撞器。

5、继续给对话人添加一个新脚本，如下所示：

```
using UnityEngine;
using ZetanStudio.DialogueSystem;
using ZetanStudio.DialogueSystem.UI;
using ZetanStudio.InteractionSystem;
using ZetanStudio.UI;

public class Interlocutor : MonoBehaviour, IInterlocutor
{
    [field: SerializeField]
    public string Name { get; private set; }

    [field: SerializeField]
    public Sprite Icon { get; private set; }

    [field: SerializeField]
    public Dialogue Dialogue { get; private set; }

    public bool IsInteractive => true;
}
```

```

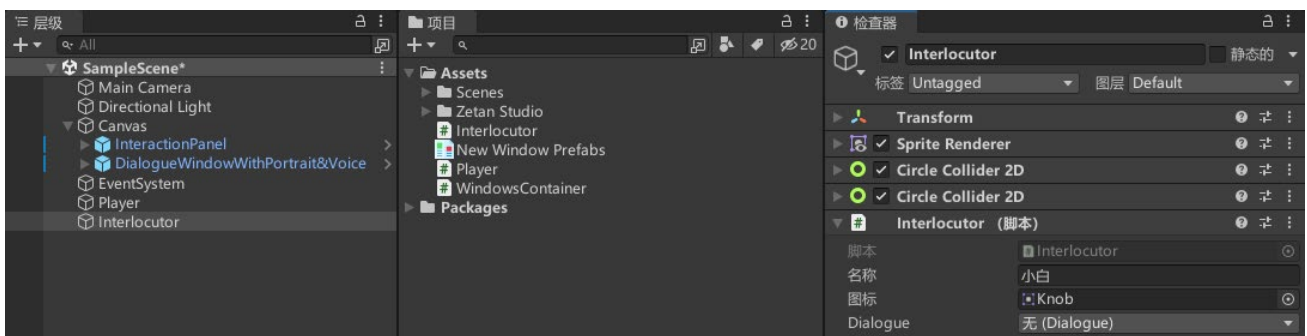
bool IInteractive.Interactable { get; set; }

bool IInteractive.DoInteract()
{
    return WindowManager.OpenWindow<DialogueWindow>(this);
}
//玩家离开可触发对话的区域范围时调用此方法
void IInteractive.OnNotInteractable()
{
    //若正在与此对话人进行对话
    if (WindowManager.IsWindowOpen<DialogueWindow>(out var window) && window.Target == this as
    Interlocutor)
        //则中断对话
        window.Interrupt();
}

private void OnTriggerEnter2D(Collider2D collision)
{
    //玩家进入触发区域，则将此对话人加入交互面板
    if (collision.CompareTag("Player")) IInteractive.PushToPanel(this);
}
private void OnTriggerStay2D(Collider2D collision)
{
    if (collision.CompareTag("Player")) IInteractive.PushToPanel(this);
}
private void OnTriggerExit2D(Collider2D collision)
{
    //玩家离开触发区域，则将此对话人移出交互面板
    if (collision.CompareTag("Player")) IInteractive.RemoveFromPanel(this);
}
private void OnDestroy()
{
    IInteractive.RemoveFromPanel(this);
}
}

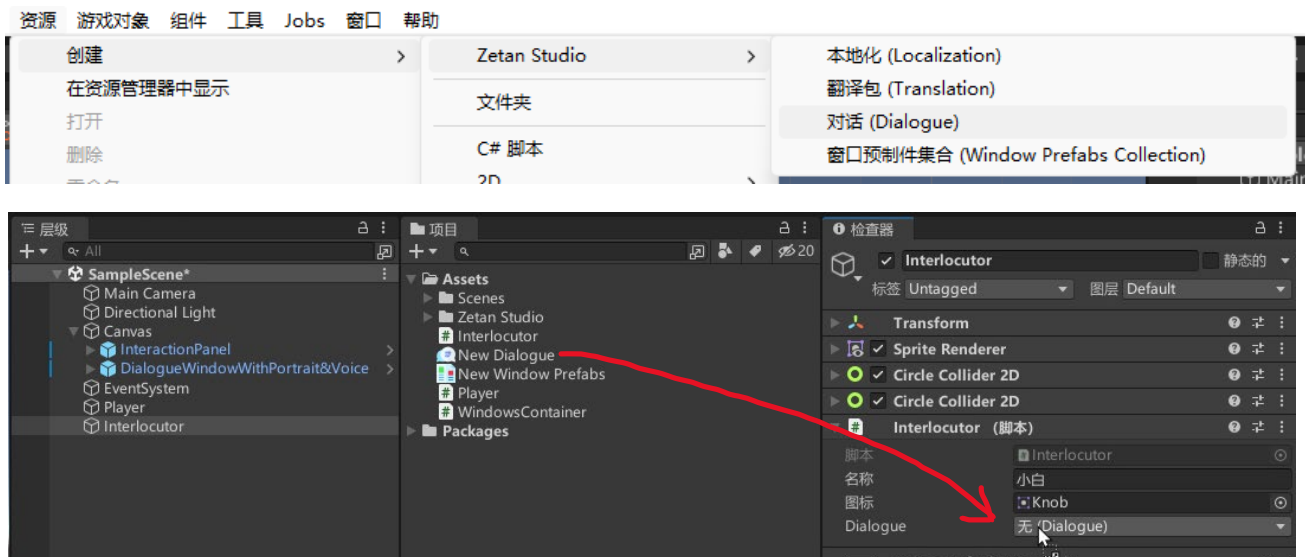
```

6、设置对话人的名字和图标，如图所示：

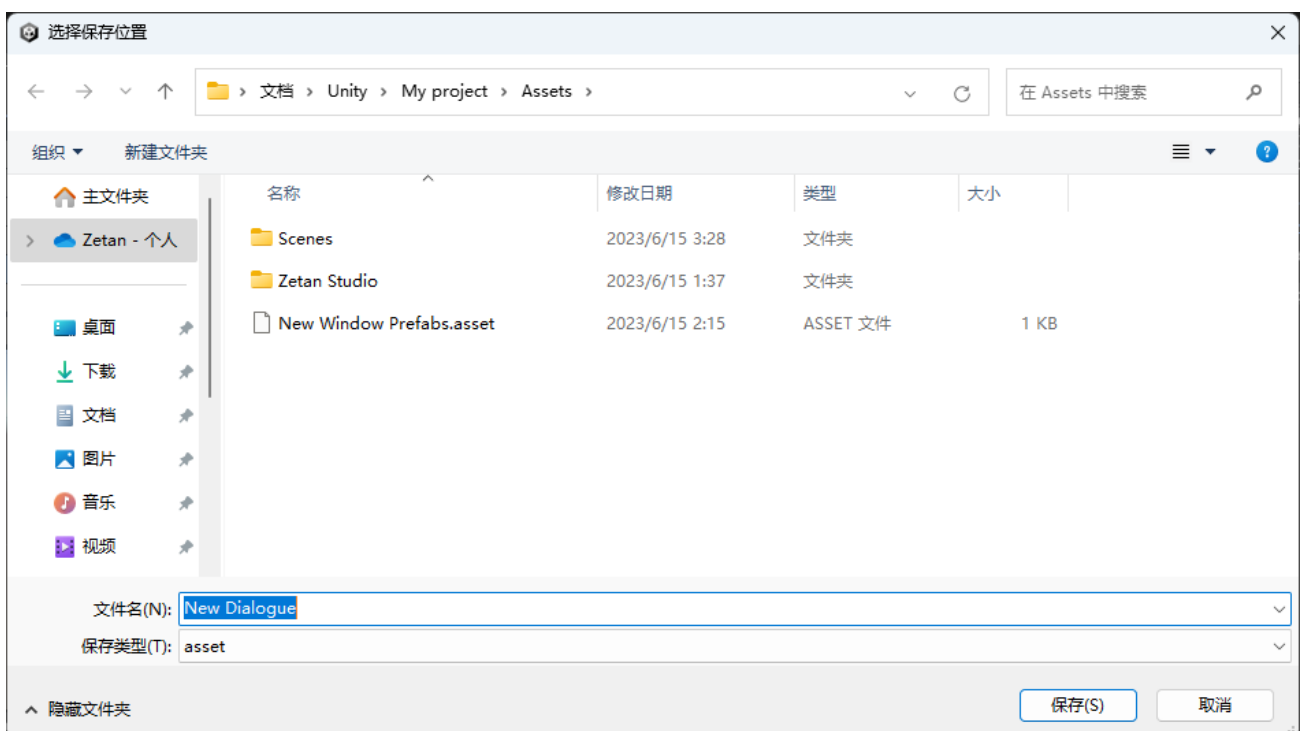
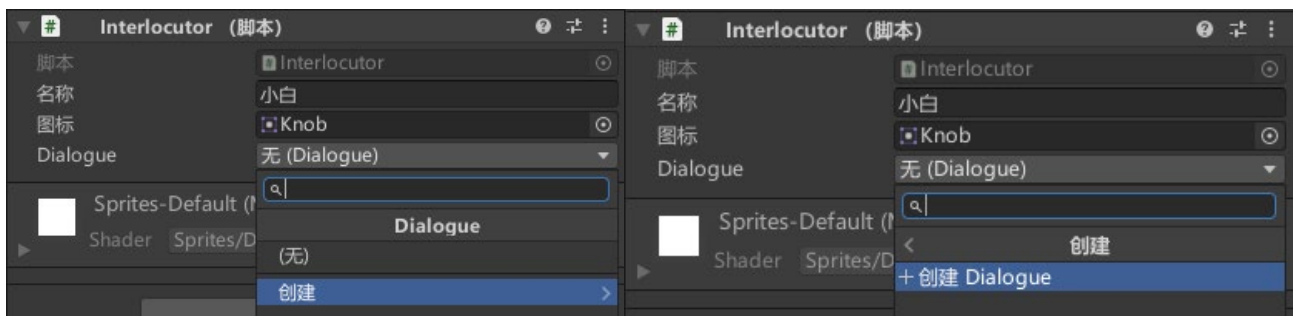


1.3 创建和编辑对话

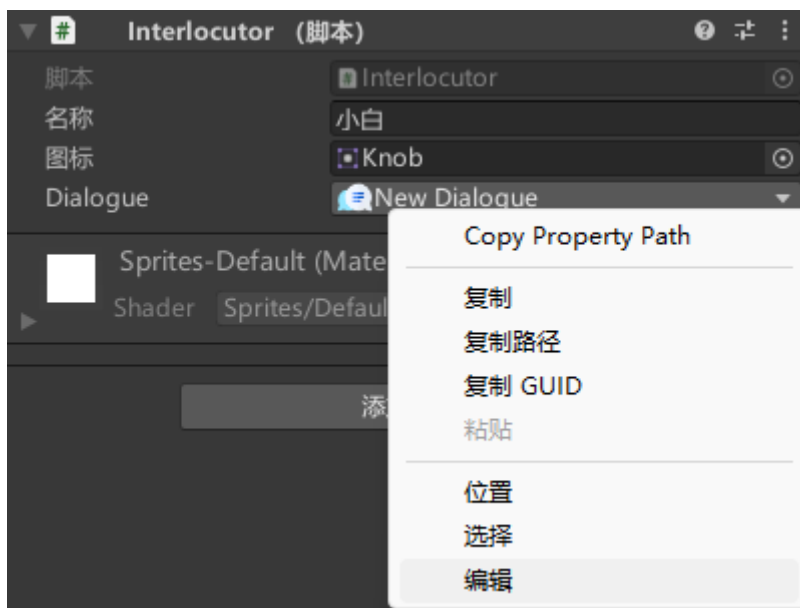
1、给此对话人新建一个对话，可通过工具栏创建并拖拽到对话人组件的 Dialogue 中来引用它，如图所示：



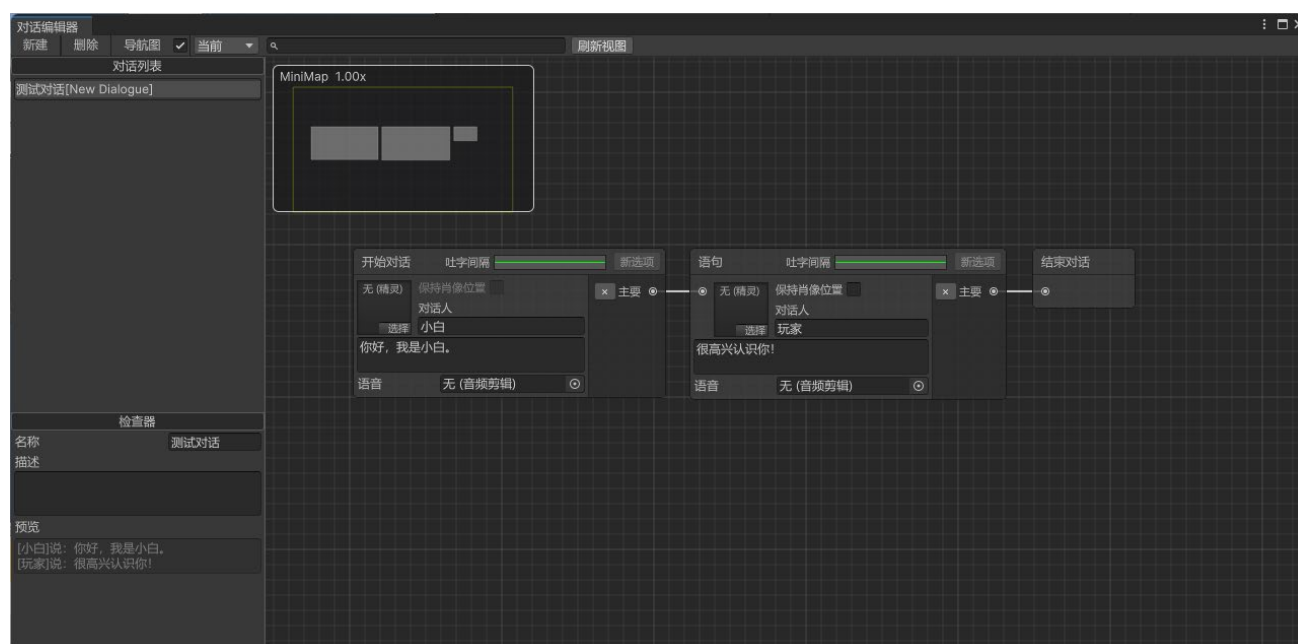
2、也可以直接在对话人组件的对话下拉窗口中创建，如图所示：



- 3、现在我们来编辑这段对话，可双击对话资源打开对话编辑器，或者右键点击对话人组件中的 Dialogue 下拉按钮来打开，如图所示：

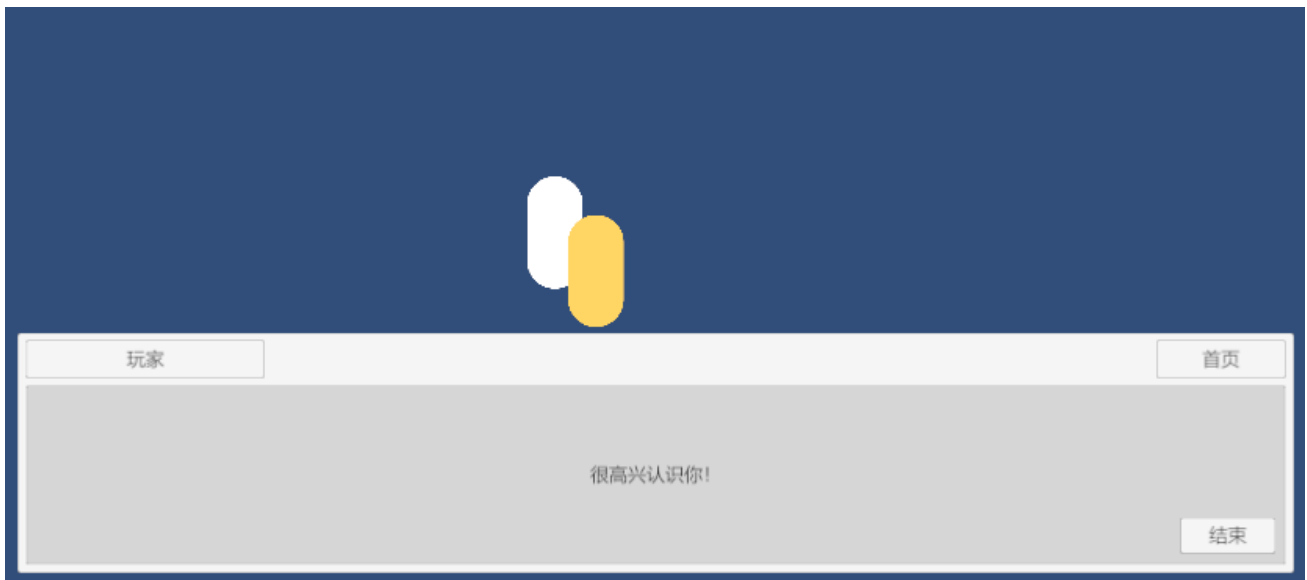
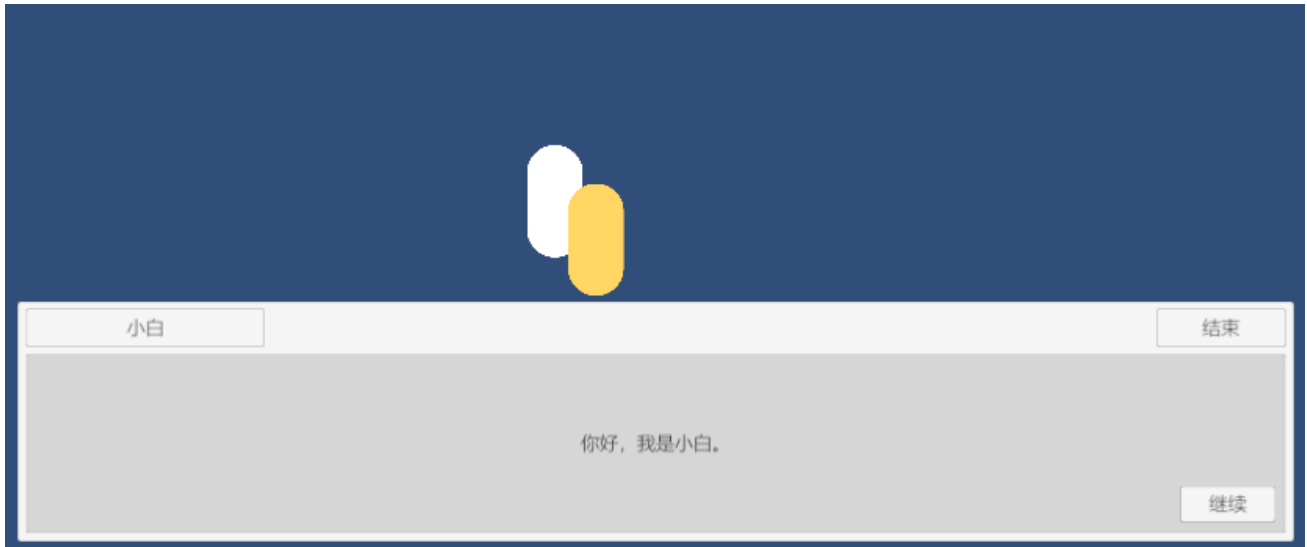


- 4、在打开的对话编辑器中，随意编辑一下当前对话，如图所示：



- 5、最后，运行游戏，将玩家移动到对话人旁边，即可通过交互面板中的按钮来触发对话，如图所示：



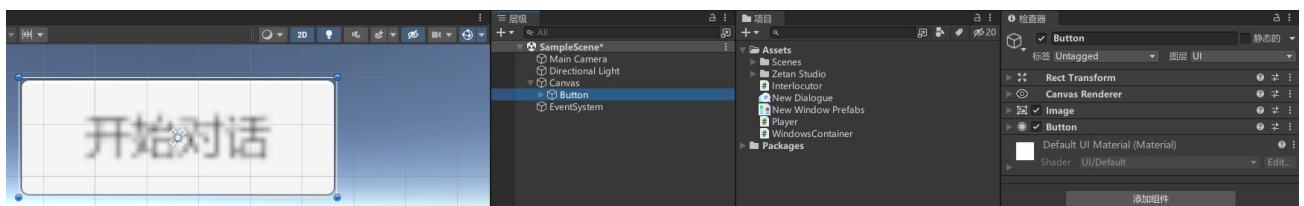


第二节 其它方式进行对话

如果项目中已存在现成的交互系统或者不需要交互功能，用不到本插件自带的交互系统，则可参考本节来了解如何触发对话。要注意的是，不通过交互系统打开对话窗口，对话窗口的“首页”按钮将不可用。

2.1 搭建 UI

- 1、搭建 UI 的操作除第 4 步的与交互面板相关的不做之外，和第一节步骤一致。
- 2、我们需要将对话传入对话窗口来打开它，这里我用一个简单的例子来说明如何操作。
- 3、假设现在我们用一个按钮来触发对话，则要先在第 1 步所创建的画布中添加一个按钮，如图所示：



- 4、给第 3 步创建的按钮添加一个新的脚本，如下所示：

```

using UnityEngine;
using ZetanStudio.DialogueSystem;
using ZetanStudio.DialogueSystem.UI;
using ZetanStudio.UI;

public class StartDialogue : MonoBehaviour
{
    public Dialogue dialogue;

    public void TriggerDialogue()
    {
        WindowManager.OpenWindow<DialogueWindow>(dialogue);
    }
}

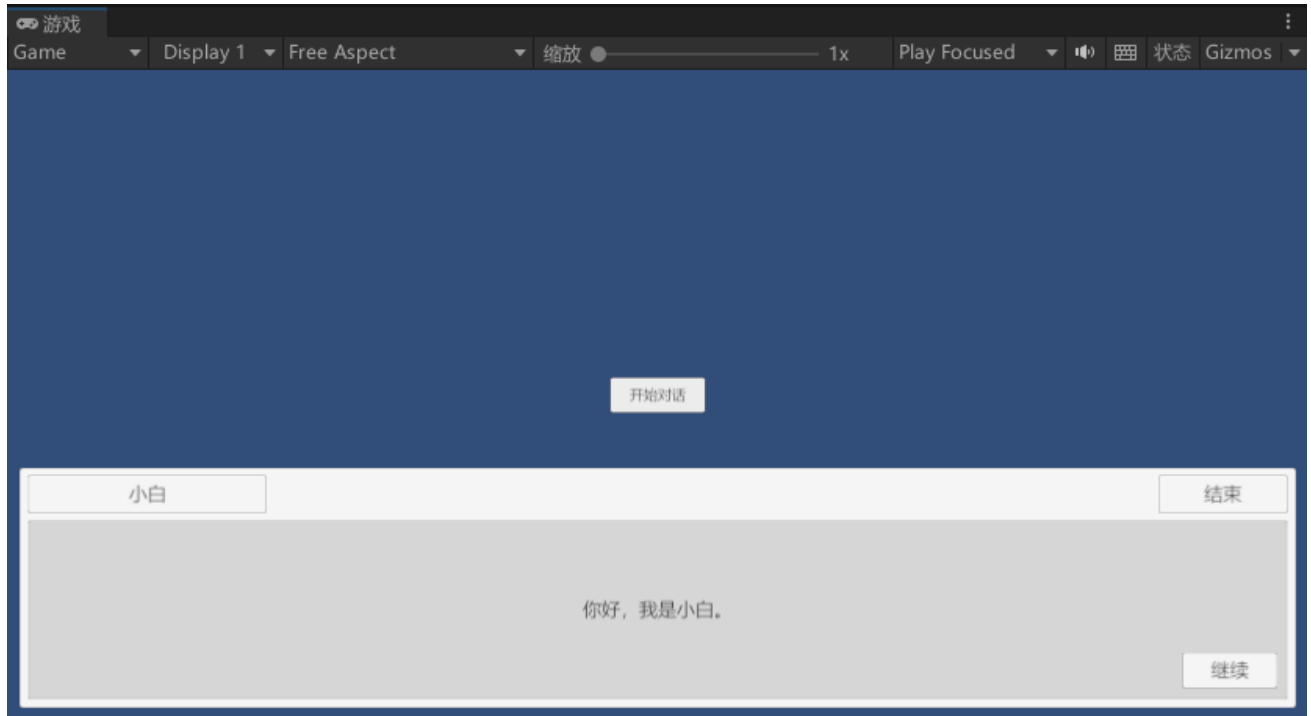
```

5、将此脚本的 `TriggerDialogue` 方法绑定到按钮的点击事件，如图所示：



2.2 创建和编辑对话

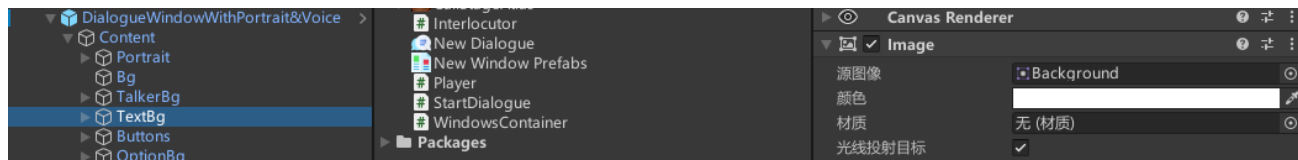
- 1、创建和编辑对话的方法与第一节并无二致，无非就是编辑的对象从对话人变成了按钮。
- 2、完成对话的创建和编辑后，点击运行游戏，即可通过点击“开始对话”按钮来触发对话了，如图所示：



第三节 美化对话窗口

本节我们将以 DialogueWindowWithPortrait&Voice 预制件为例来说明如何更换对话窗口的主题，即美化对话窗口。可进行美化的对象基本如下：

1、修改 Content/TextBg 的图片可更换对话窗口文本区的背景，如图所示：



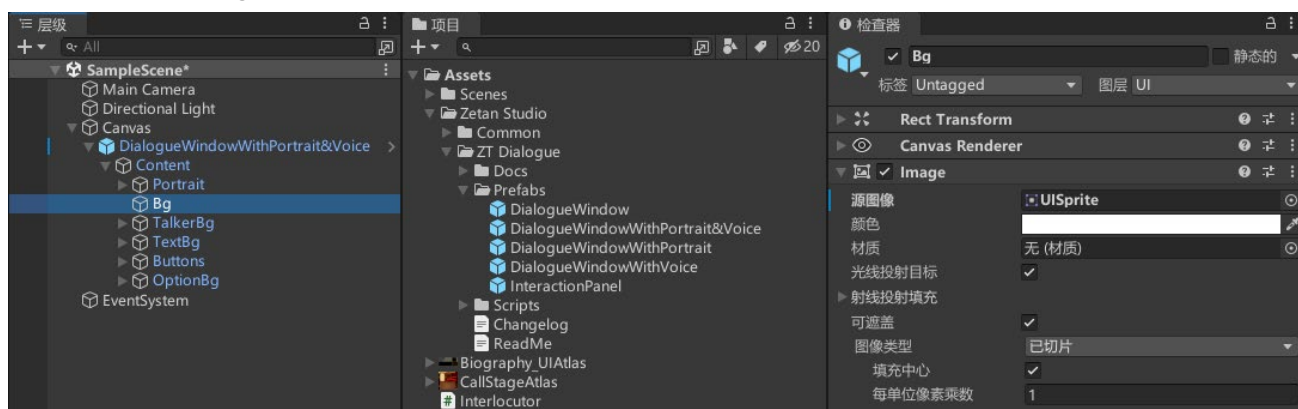
未修改：



修改后：



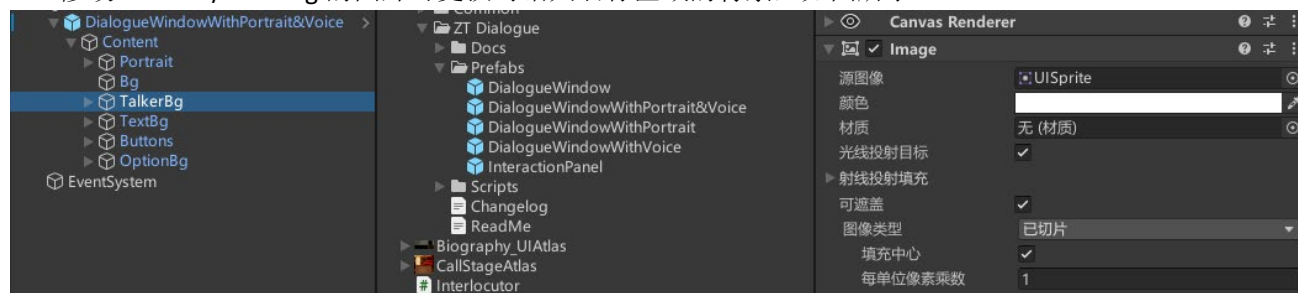
2、修改 Content/Bg 的图片可更换对话窗口的背景，如图所示：



修改后：



3、修改 Content/TalkerBg 的图片可更换对话人名称区域的背景，如图所示：



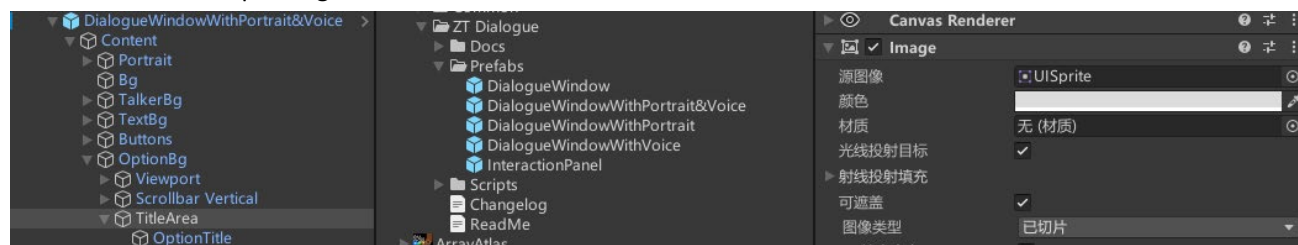
修改后：



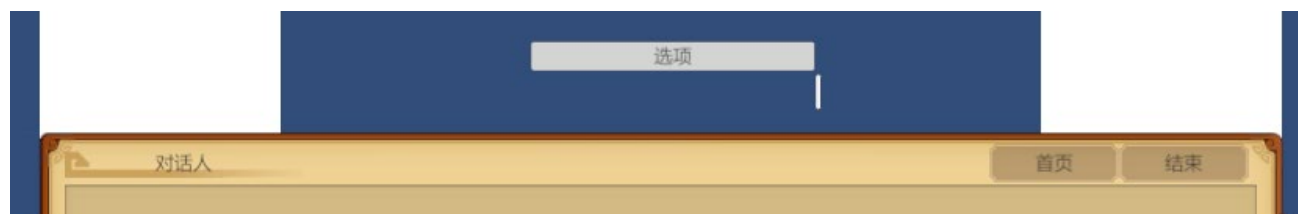
4、接着修改右侧三个按钮的图片，修改后的效果如下：



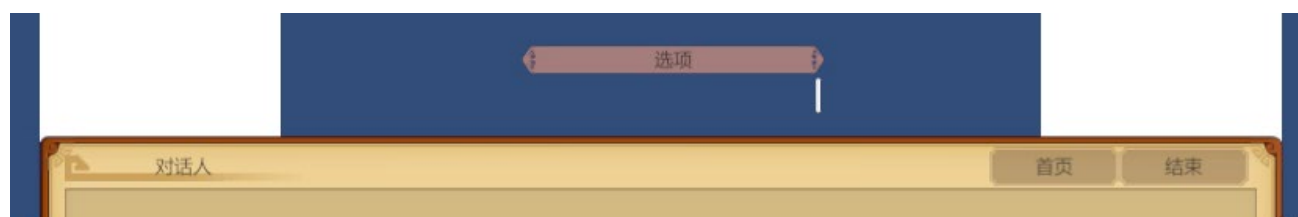
5、修改 Content/OptionBg/TitleArea 的图片可更换选项区标题的背景，如图所示：



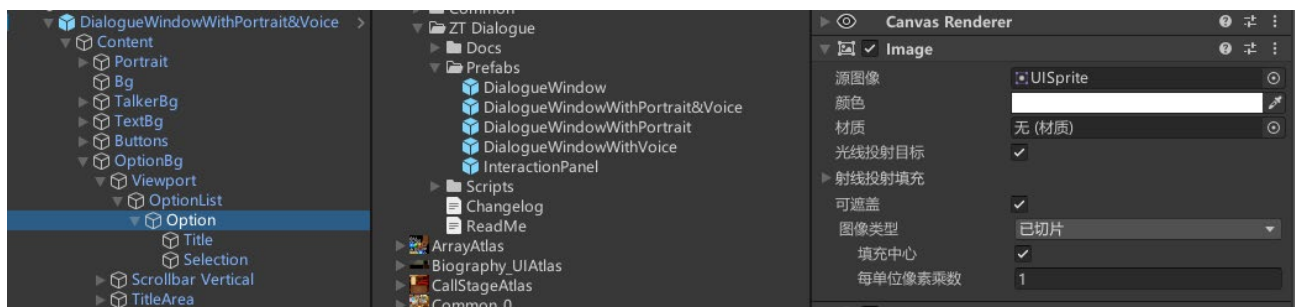
修改前：



修改后：



6、修改 Content/OptionBg/Viewport/OptionList/Option 的图片可更换选项按钮的背景，如图所示：



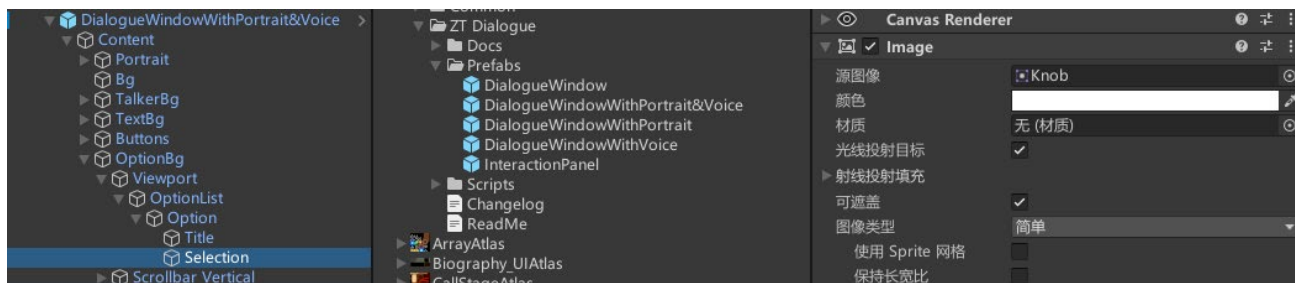
修改前：



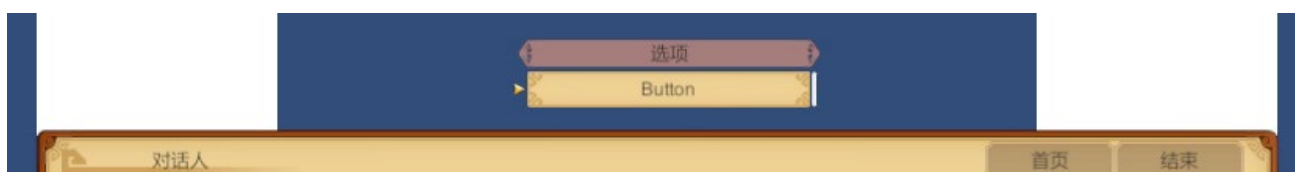
修改后：



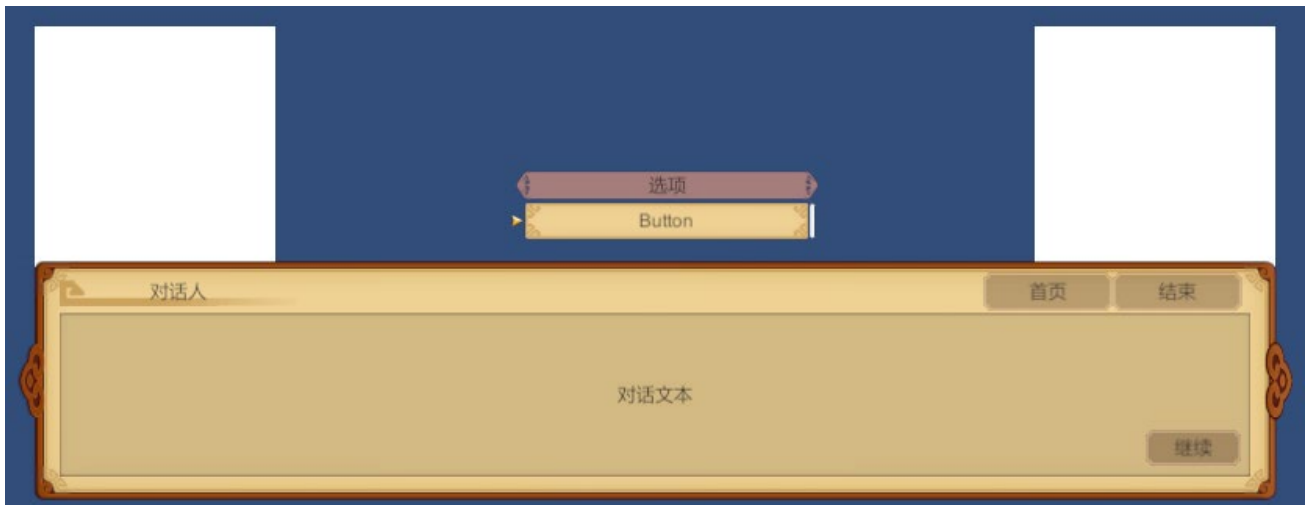
7、修改 Content/OptionBg/Viewport/OptionList/Option/Selection 的图片可更换选项按钮左侧的选中标识，如图所示：



修改后：



8、到此，我们就完成了对话窗口的简单美化，最终效果如图所示：

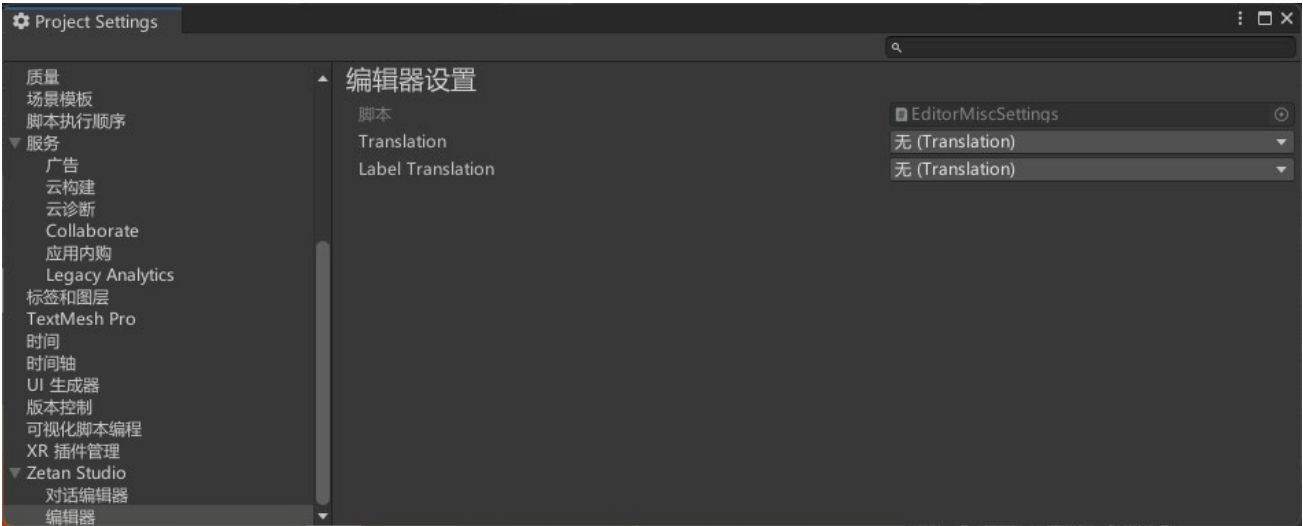


其中，左右两侧的肖像显示为白色的方块，这是正常现象，运行游戏后它们将会自动隐藏；选项区右侧滚动条的美化这里就不演示了，请自行处理。

附录

编辑器设置

可以在“编辑→项目设置...→Zetan Studio”中找到本插件的编辑器设置，如图所示：



其中，编辑器设置有两个翻译包（Translation）设置，第一个是编辑器全局的翻译，而第二个则是针对[Label]特性的翻译，可能要同时修改这两个才能得到想要的编辑器翻译效果，若将它们都置空则插件将以简体中文显示。

类

1. 轻量级通用型数据

轻量级通用型数据是一个用来存储简单数据的类，它无需增加额外字段就能存储用户指定的数据，这些数据将使用整型下标或字符串索引进行访问。

轻量级通用型数据是 GenericData，它的成员如下：

运行时成员	说明
字段 intList	下标索引的整型数据
字段 boolList	下标索引的布尔型数据
字段 floatList	下标索引的浮点型数据
字段 stringList	下标索引的字符串数据
字段 dataList	下标索引的通用型数据
字段 intDict	键索引的整型数据
字段 boolDict	键索引的布尔型数据
字段 floatDict	键索引的浮点型数据
字段 stringDict	键索引的字符串数据
字段 dataDict	键索引的通用型数据
索引器 object this[string key]	可快速读写键索引的类型受支持的数据
方法 int Write(int)	写入下标索引的整型数据，并返回源数据

方法 bool Write(bool)	写入下标索引的布尔型数据，并返回源数据
方法 float Write(float)	写入下标索引的浮点型数据，并返回源数据
方法 string Write(string)	写入下标索引的字符串数据，并返回源数据
方法 GenericData Write(GenericData)	写入下标索引的通用型数据，并返回源数据
方法 T WriteAll<T>(T)	批量写入指定类型数据，并返回源数据，泛型参数 T 得是 IEnumerable
方法 int Write(string, int)	写入键索引的整型数据，并返回源数据
方法 bool Write(string, bool)	写入键索引的布尔型数据，并返回源数据
方法 float Write(string, float)	写入键索引的浮点型数据，并返回源数据
方法 string Write(string, string)	写入键索引的字符串数据，并返回源数据
方法 GenericData Write(string, GenericData)	写入键索引的通用型数据，并返回源数据
方法 ReadOnlyCollection<int> ReadIntList()	读取所有下标索引的整型数据
方法 ReadOnlyCollection<bool> ReadBoolList()	读取所有下标索引的布尔型数据
方法 ReadOnlyCollection<float> ReadFloatList()	读取所有下标索引的浮点型数据
方法 ReadOnlyCollection<string> ReadStringList()	读取所有下标索引的字符串数据
方法 ReadOnlyCollection<GenericData> ReadDataList()	读取所有下标索引的通用型数据
方法 ReadOnlyDictionary<string, int> ReadIntDict()	读取所有键索引的整型数据
方法 ReadOnlyDictionary<string, bool> ReadBoolDict()	读取所有键索引的布尔型数据
方法 ReadOnlyDictionary<string, float> ReadFloatDict()	读取所有键索引的浮点型数据
方法 ReadOnlyDictionary<string, string> ReadStringDict()	读取所有键索引的字符串数据
方法 ReadOnlyDictionary<string, GenericData> ReadDataDict()	读取所有键索引的通用型数据
方法 bool TryReadInt(int, out int)	尝试用下标读取整型数据
方法 bool TryReadBool(int, out bool)	尝试用下标读取布尔型数据
方法 bool TryReadFloat(int, out float)	尝试用下标读取浮点型数据
方法 bool TryReadString(int, out string)	尝试用下标读取字符串数据
方法 bool TryReadData(int, out GenericData)	尝试用下标读取通用型数据
方法 bool ReadInt(int)	用下标读取整型数据
方法 bool ReadBool(int)	用下标读取布尔型数据
方法 bool ReadFloat(int)	用下标读取浮点型数据
方法 bool ReadString(int)	用下标读取字符串数据
方法 bool ReadData(int)	用下标读取通用型数据
方法 bool TryReadInt(string, out int)	尝试用键读取整型数据
方法 bool TryReadBool(string, out bool)	尝试用键读取布尔型数据
方法 bool TryReadFloat(string, out float)	尝试用键读取浮点型数据
方法 bool TryReadString(string, out string)	尝试用键读取字符串数据
方法 bool TryReadData(string, out GenericData)	尝试用键读取通用型数据

方法 <code>bool ReadInt(string)</code>	用键读取整型数据
方法 <code>bool ReadBool(string)</code>	用键读取布尔型数据
方法 <code>bool ReadFloat(string)</code>	用键读取浮点型数据
方法 <code>bool ReadString(string)</code>	用键读取字符串数据
方法 <code>bool ReadData(string)</code>	用键读取通用型数据

2. 单例资源

单例资源是指单例的 `ScriptableObject` 子类资源，得放在任意的“Resources”文件夹或其子文件夹中，虽然在编辑器中资源文件可以随意复制粘贴，但尽量不要对单例资源这么做。单例资源类是一个泛型类，但为了自定义单例资源的检查器，还得再套一层基类，这个基类并不是真正的单例资源，没有单例功能。

单例资源类的基类是 `SingletonScriptableObject`，继承自 `ScriptableObject`，它由编辑器使用的成员如下：

编辑器成员	说明
静态方法 <code>void FindEditorInstance()</code>	这是一个自动化方法，用于在打开编辑器时或重新编译后刷新单例资源的静态实例引用

单例资源类是 `SingletonScriptableObject<T>`，继承自上一个类，泛型参数 `T` 得是 `ScriptableObject`，它的成员如下：

运行时成员	说明
静态字段 <code>instance</code>	<code>T</code> 类的一个实例
静态属性 <code>Instance</code>	<code>instance</code> 的关联只读属性
构造函数 <code>SingletonScriptableObject()</code>	把自身引用给 <code>instance</code> 作为单例

由编辑器使用的成员如下：

编辑器成员	说明
静态方法 <code>T GetOrCreate()</code>	获取或创建实例
静态方法 <code>void CreateSingleton()</code>	尝试创建单例

3. 单例窗口

单例窗口，会有一个对指定类型窗口的静态引用，尽量不要在没有任何可用实例的情况下访问这个静态引用，因为如果该引用为空，那将会一直用 `T FindObjectOfType<T>(true)` 来查找，影响性能。

单例窗口类是 `SingletonWindow<T>`，继承自 `Window`，泛型参数 `T` 得是 `Window`，它的成员如下：

运行时成员	说明
静态字段 <code>instance</code>	<code>T</code> 类的一个实例
静态属性 <code>Instance</code>	<code>instance</code> 的关联只读属性

接口

1. 玩家名字持有者接口

玩家名字持有者接口用于关键字系统，用来把一些特殊的 ID 字符串替换为玩家名字。

玩家名字持有者接口是 `IPlayerNameHolder`，它的成员如下：

运行时成员	说明
静态属性 <code>Instance</code>	玩家名字持有者单例
属性 <code>Name</code>	玩家的名字

如何使用：

首先让玩家类继承这个接口，实现 **Name** 属性，然后把玩家类的实例赋给这个接口的静态属性 **Instance**。下面给出一个玩家类的示例代码：

```
public class Player : MonoBehaviour, IPlayerNameHolder
{
    [field: SerializeField]
    public string Name { get; set; }

    private void Awake()
    {
        IPlayerNameHolder.Instance = this;
    }
}
```

2. 可复制接口

可复制接口是一个由对话编辑器使用的接口，当然用在别的地方也行。如果想让某种对象可被复制，则继承这个接口。

可复制接口是 **ICopiable**，它的成员如下：

运行时成员	说明
方法 <code>object Copy();</code>	返回复制的对象

3. 可淡入淡出接口

可淡入淡出接口一般用在 UI 组件上面，例如窗口类，用来对 **CanvasGroup** 进行淡入淡出操作。

可淡入淡出接口是 **IFadeAble**，它的成员如下：

运行时成员	说明
属性 <code>MonoBehaviour</code>	用来执行淡入淡出协程的 MonoBehaviour
属性 <code>FadeTarget</code>	用来淡入淡出的 CanvasGroup
属性 <code>FadeCoroutine</code>	淡入淡出协程
静态方法 <code>void FadeTo(IFadeAble, float, float, Action)</code>	淡入淡出方法

4. 场景加载器接口

场景加载器接口用于存档系统加载场景，因为本插件不确定用户要如何实现场景加载，比如切换场景有没有画面的淡入淡出，有没有进度条，等等，所以用一个接口代替具体的场景加载器。

场景加载器接口是 **ISceneLoader**，它的成员如下：

运行时成员	说明
静态属性 <code>Instance</code>	场景加载器单例
方法 <code>void LoadScene(string, Action)</code>	加载指定名称的场景，并在加载完成时执行传入的回调

用法与玩家名字持有者接口差不多，无非是属性的实现变成了方法的实现。

5. 消息显示器接口

消息显示器用来推送游戏中来自本插件的消息，使用接口的理由同上。

消息显示器接口是 `IMessageDisplay`，它的成员如下：

运行时成员	说明
静态属性 <code>Instance</code>	消息显示器单例
方法 <code>void Push(string)</code>	向消息显示器中压入给定的消息

用法同上。

特性

1. 初始化特性

加了初始化特性的类会在读档时调用指定名称的初始化方法，当然也可以在其它需要初始化的地方调用。将这个特性加在特定类型上，对比把加初始化方法特性加在指定方法上，性能更好。

初始化特性类是 `InitAttribute`，继承自 `Attribute`，它的成员如下：

运行时成员	说明
字段 <code>method</code>	初始化方法的名称，这个方法应该无参数或可省略参数的方法
字段 <code>priority</code>	调用优先级
静态方法 <code>void InitAll()</code>	按优先级依次调用所有加了此特性的类的初始化方法

2. 初始化方法特性

加了初始化方法特性的方法会在读档时被调用，当然也可以在其它需要初始化的地方调用。将这个特性加在初始化方法上，对比把初始化特性加在特定类型上，更方便，不用填写方法名称，也不用担心初始化方法更名后忘记更新初始化特性里的方法名称，但是调用性能更差。

初始化方法特性类是 `InitMethodAttribute`，继承自 `Attribute`，它的成员如下：

运行时成员	说明
字段 <code>priority</code>	调用优先级
静态方法 <code>void InitAll()</code>	按优先级依次调用所有加了此特性的初始化方法