第5章

状态图和超级状态

5.1 状态

状态(State)用于告诉对象在特定情况下应该如何行动。它们经常被用于人工智能(AI)行为。例如,非玩家角色(None Player Character,NPC)的状态图可以有 4 个状态:巡逻、追赶、攻击和逃跑;再如,游戏中门锁的状态图可以有 3 个状态:锁定、解锁和打开。

在 Bolt 中,有如下两种状态图。

(1) 流状态图(Flow State):嵌套流图的状态图。这意味着可以在每个状态中使用所有单元和连接,一般创建的大多数状态图都是流状态图。

(2) 超状态图(Super State):嵌套另一个状态图的状态图。它们允许创建层次有限状态机,即状态机中的状态机。它们对于图的高级重用和组织非常有用。

这两种状态图都是 Nesters,这意味着它们的工作方式与机器完全一样:它们的子图可以嵌入,也可以从宏中引用。它们的检查器的外观和行为都是一样的。状态与转换连接在一起。

要显式地创建状态图,在项目视图中右击,在弹出的快捷菜单中选择 Create→Bolt→State Macro 命令,即可创建一个状态图,如图 5-1 所示。

可以通过选择一个或多个开始状态(Start State)来作为状态图的启动状态。要做到这一点,只需右击相应的状态并选择切换开始。启动状态用绿色高亮显示。与大多数有限状态机不同,Bolt 允许多种启动状态。这意味着可以让并行有限状态机在同一个图中运行,甚至在某个点加入。然而,在大多数情况下,只需要一个启动状态。

可以使用任意状态来触发到其他状态的转换,不管当前处于哪个状态,任意状态节点如图 5-2 所示。然而,此状态不能接收任何转换或执行任何操作。

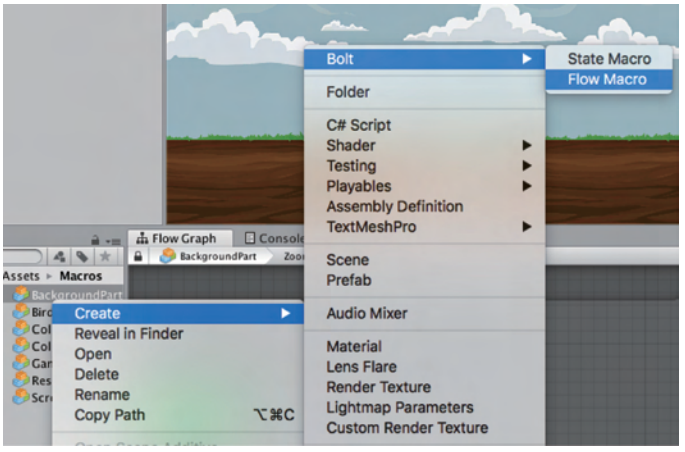

图 5-1 建立 State Macro

图 5-2 任意状态节点

状态图标的顶部是它的标题和摘要，如图 5-3 所示。这些对功能没有影响，只是一种识别状态的手段。可以选择状态嵌套图的源，并单击 Edit Graph 按钮打开它，还可以双击状态节点来打开它的嵌套图。

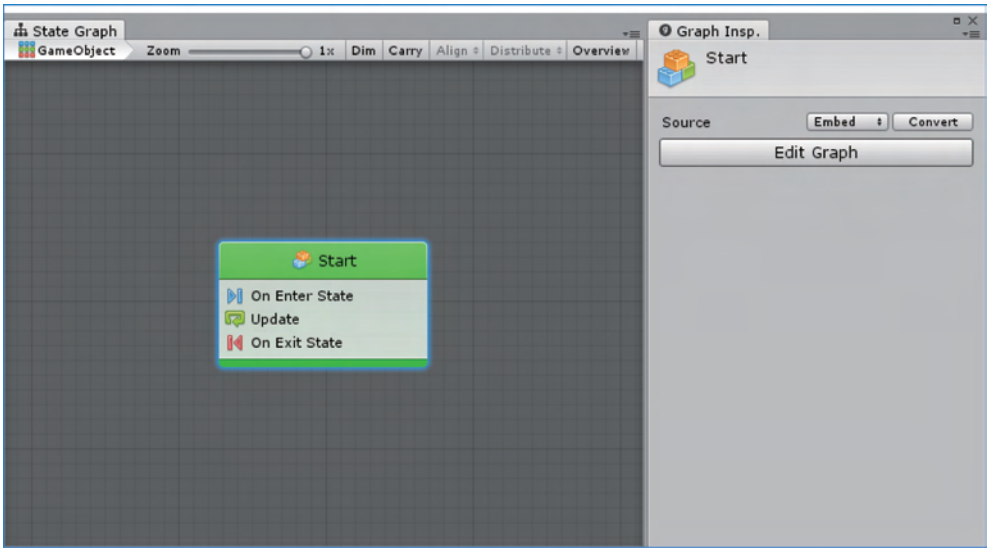

图 5-3 状态图标

5.2 流状态图

流状态的主体是嵌套流图中使用的所有事件的列表。默认情况下，流状态含有 On Enter State、Update 和 On Exit State 事件单元，如图 5-4 所示。但是如果不需要这些事件单元，则可以删除，并添加其他需要的事件单元。

在图 5-4 左上角的面包屑图中，可以看到目前在游戏对象的状态图中处于开始状态，如图 5-5 所示。

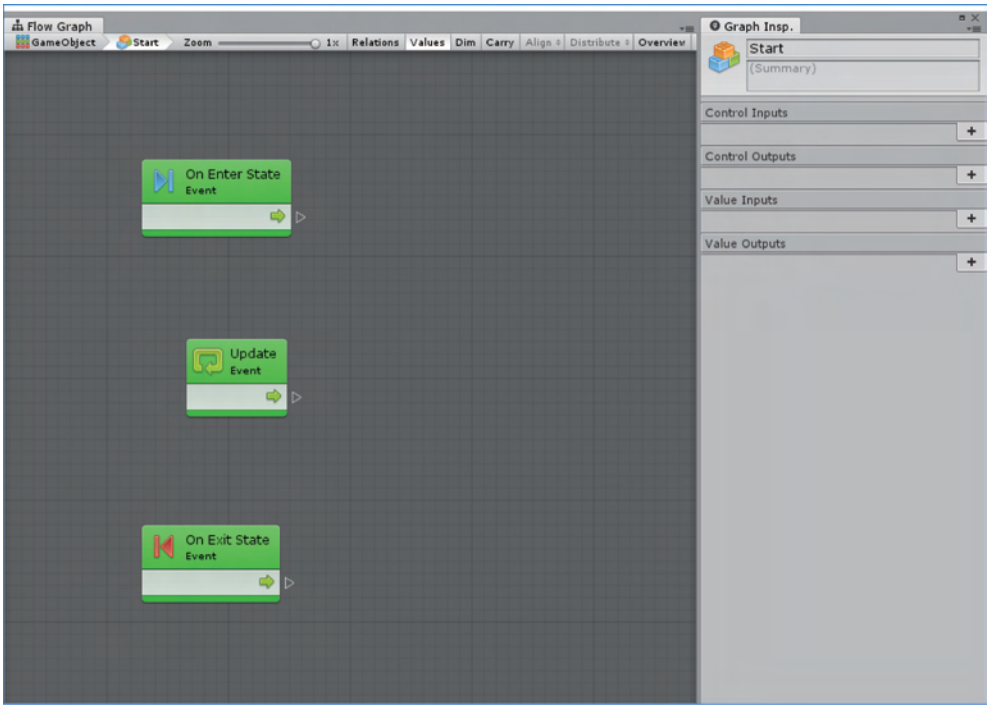

图 5-4　流状态默认含有的 On Enter State、Update 和 On Exit State 事件单元

可以使用工具栏的面包屑图随时导航回父图。在图形检查器的顶部，当没有选择节点时，可以编辑状态的标题和摘要，如图 5-6 所示。

图 5-5　面包屑图

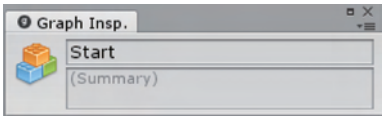

图 5-6　编辑状态的标题和摘要

可以忽略输入和输出端口定义，端口定义一般只用于超级单元，而不用于流状态。图中预先包含了三个事件。在进入状态(On Enter State)时，由传入转换导致父状态时调用；在退出状态(On Exit State)时，在状态被一个传出转换退出之前调用；当状态处于活动状态时，每个帧都会调用 Update。添加到流图中的每个事件都将只在父状态处于活动状态时侦听。这个图的其余部分与正常的流图完全相同。可以使用的单元没有限制。

超级状态的创建和编辑完全类似于流状态。当然，主要的区别在于，它不是一个流图，而是另一个状态图。当输入超级状态时，将输入嵌套图的所有开始状态。当超级状态退出时，嵌套图的每个状态和转换都将处于非活动状态。

5.3　状态转换

转换(Transition)是连接状态，以确定活动状态何时切换。要创建转换，右击源状态并在弹出的快捷菜单中选择 Make Transition 命令。然后，单击目标状态，建立一个转换，如

图 5-7 所示。

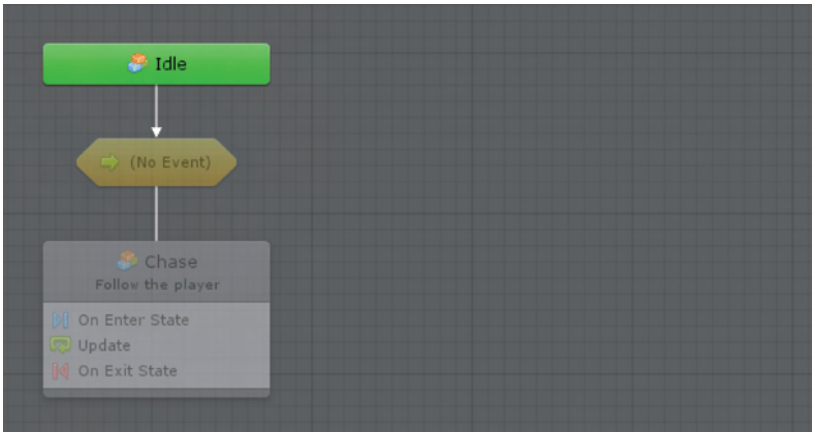

图 5-7 建立一个转换

作为快捷方式，可以在源节点上按住 Ctrl(Mac 下为⌘)键并拖动来创建转换。如同流状态，转换也是一个嵌套的流图。正如在图检查器中所看到的图，新转换存在一些问题，如图 5-8 所示。它从未被遍历过，因为还没有提供指定何时分支的事件。这就是该转换与目的状态一起变暗的原因。如果双击它的节点或单击 Edit Graph 按钮，就可以对转换图进行编辑。

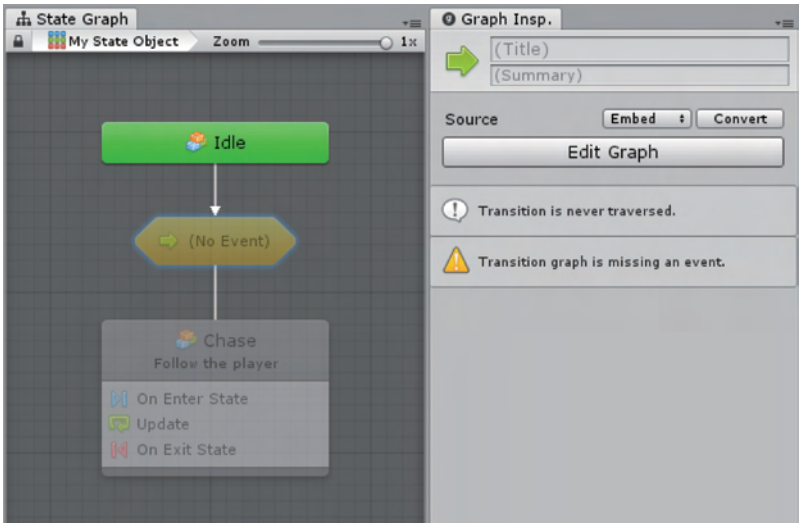

图 5-8 检查器中看到的图

默认情况下，转换图的配置如图 5-9 所示。

状态触发器转换(Trigger State Transition)单元是一个特殊单元，它告诉父状态应该通过当前转换进行分支。用于可以使用状态转换图中的任何单元，如事件或分支来触发此转换。例如，如果要转换到追逐状态，只有当带有 Player 标签的对象进入敌人的触发器时，可以有一个这样的转换图，如图 5-10 所示。

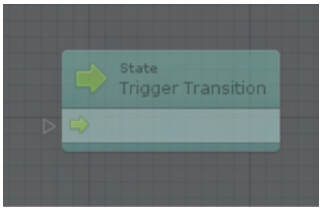

图 5-9 转换图的默认配置

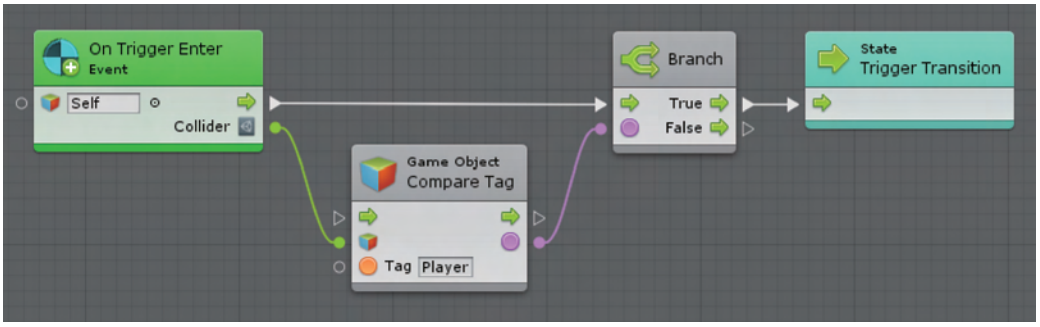

图 5-10　转换图示例

最后，如果想自定义转换在父状态图中的标签，可以取消选择所有单元，并在图检查器中编辑图的标题，如图 5-11 所示。

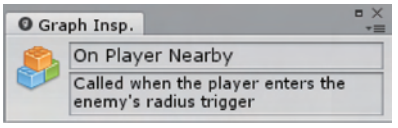

图 5-11　自定义转换在父状态图中的标签

当回到父状态时，在父状态图中转换的样子如图 5-12 所示。

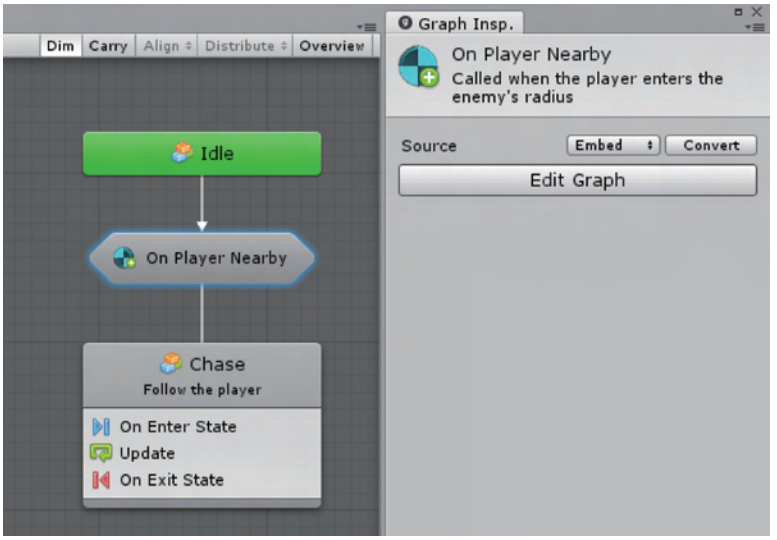

图 5-12　在父状态图中转换的样子

如果没有为转换分配自定义标题，则系统将使用事件的名称和描述作为显示标题。默认情况下，转换标签总是可见的。如果发现其在图中占用了太多的屏幕空间，可以在 Unity 编辑器的菜单栏中选择 Edit→Preferences 命令显示 Unity 编辑器的首选项窗口，在 Bolt 面板中的 State Graphs 中勾选 Transitions Reveal 复选框，更改其显示转换的模式，更改后的转换显示如图 5-13 所示。

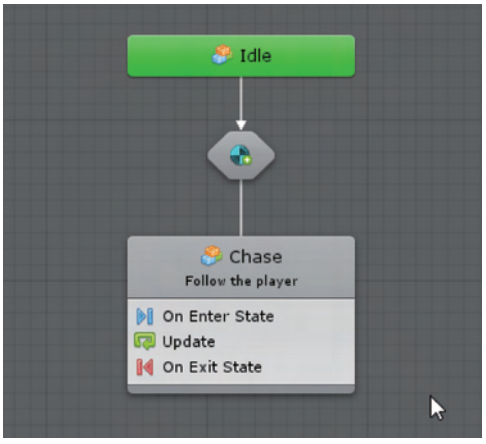

图 5-13 更改后的转换显示

有时候,状态向自身进行转换可能是有用的。右击状态并在弹出的快捷菜单中选择 Make Self Transition 命令。例如,假设想要一个敌人巡逻,每 3s 就把它的目标改变到一个随机的位置,其巡逻状态的流图如图 5-14 所示。

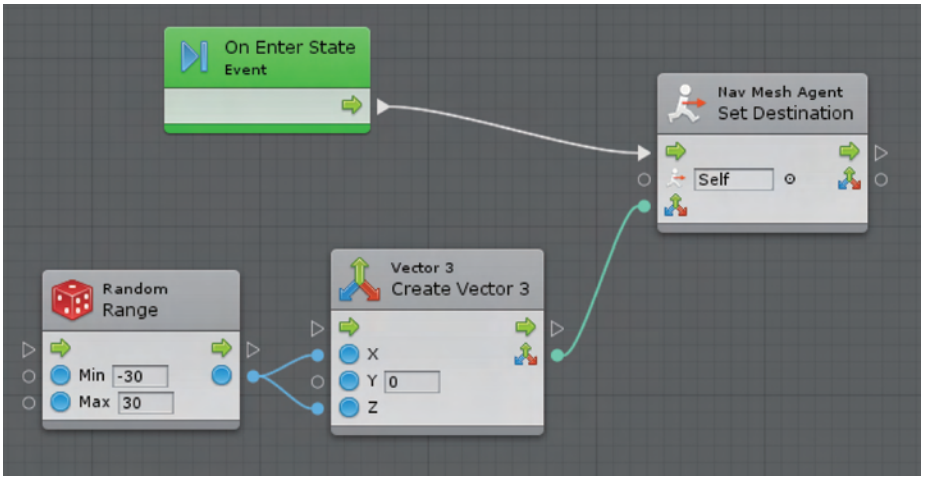

图 5-14 巡逻状态的流图

其自我状态转换的流图如图 5-15 所示。

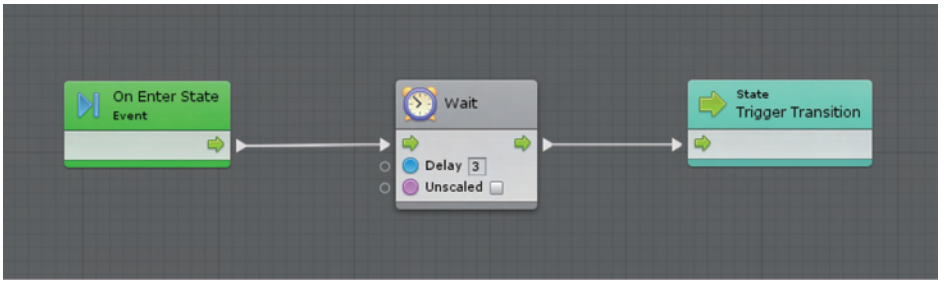

图 5-15 自我状态转换的流图

在父状态图中的自我状态转换图如图 5-16 所示。

可以向一个状态添加多少个转换是没有限制的，如图 5-17 所示。在转换之间没有优先级的概念，必须使用条件来确保选择了正确的转换。

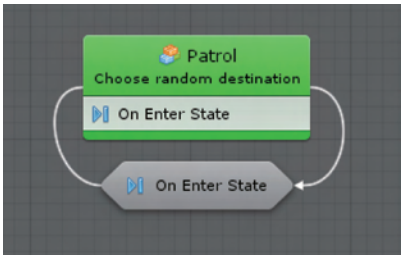

图 5-16　在父状态图中的自我状态转换图

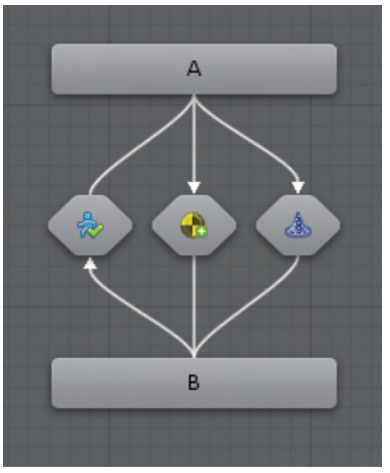

图 5-17　向一个状态添加多少个转换是没有限制的

5.4　状态单元

状态单元(State Unit)非常类似于超级单元，但是对于状态图而不是流图，它们允许将整个状态图嵌套到父流图中的单个单元中。

要创建空白状态单元，在模糊查找器中选择 Nesting→State Unit 命令。与往常一样，可以双击节点或单击检查器中的 Edit Graph 按钮打开嵌套图。要从宏(Macro)中创建状态单元，可以将宏资产拖动到图形中，也可以从模糊查找器中的宏类别中选择它。

状态单元如图 5-18 所示。有两个控制输入端口来指示何时启动和停止它，以及两个匹配的控制输出端口来指定之后要做什么。

启动状态单元时，将输入嵌套状态图中的所有开始状态。当它停止时，嵌套图中的每个状态和转换都将被标记为非活动的。

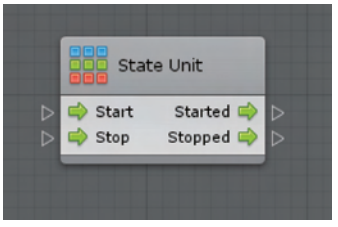

图 5-18　状态单元

第6章 和Unity的脚本协作

Bolt 支持 Unity 中的每个类和结构类型。默认情况下，模糊查找器中只包含最常见的类型以及从 Unity 对象派生的所有类型，如组件、Mono 行为和可编写脚本的对象。如果需要在图中使用非 Unity 类型，可以在 Unity 编辑器的菜单栏中选择 Tools→Bolt→Unit Options Wizard 命令添加它。例如，如果想使用低级图形 API 调用，可以添加 Unity Engine GL 类，然后单击 Generate 按钮。如果想使用来自自定义程序集的自定义类型（如第三方插件），需要首先在程序集选项（Assembly Options）中添加它。

Bolt 提供了一个简单的 API 来处理变量，允许获取或设置变量的值，并检查是否定义了变量。所有这些操作都可以从 Variables 类中获得。例如：

```
Variables.Application.Set("score",100);
```

6.1 变量作用域

要访问图上的变量，首先需要创建一个图引用。如果只想在机器上得到根图，可以使用以下代码：

```
var graphReference = GraphReference.New(flowMachine);
```

要访问嵌套图，需要将它们的父节点作为附加参数传递。例如：

```
var graphReference = GraphReference.New(flowMachine,
                     superUnit);
```

最后，只需传递图形的引用：

```
Variables.Graph(graphReference)
```

访问对象上的变量：

```
Variables.Object(gameObject)
```

访问场景变量：

```
Variables.Scene(scene)
```

或者

```
Variables.Scene(gameObjectInScene)
```

或者

```
Variables.ActiveScene
```

访问应用程序变量：

```
Variables.Application
```

访问存储级别的变量：

```
Variables.Saved
```

要获得变量的值，使用带有名称参数的 Get()方法：

```
scope.Get("name");
```

变量不是强类型的，因此需要手动转换它们。例如：

```
int health = (int)Variables.Object(player).Get("health")
```

要设置变量的值，使用具有名称和值参数的 Set()方法：

```
scope.Set("name",value);
```

因为变量不是强类型的，所以可以将任何值传递给第二个参数，即使当前变量是不同类型的。使用带有尚未存在的变量名的 Set()方法将定义一个新变量。

例如：

```
Variables.Object(player).Set("health",100);
```

要检查变量是否已定义，请使用带有名称参数的 IsDefined()方法：

```
scope.IsDefined("name");
```

例如：

```
if(Variables.Application.IsDefined("score"))
{
    //...
}
```

6.2 事件 API

Bolt 提供了一个简单的 API 来从脚本触发自定义事件。需要一个调用方法：

```
CustomEvent.Trigger(targetGameObject,argument1,
                    argument2,...)
```

该方法可以传递任意数量的参数(或者根本不传递事件),就好像这个自定义事件单元一样,如图 6-1 所示。

图 6-1 自定义事件单元

可以用以下代码触发:

```
CustomEvent.Trigger(enemy,"Damage",30);
```

6.2.1 重构

Bolt 可以从项目中的任何自定义脚本自动调用方法、字段和属性。例如,可以使用 TakeDamage()方法从自定义 Player 类中创建节点:

```
using UnityEngine;

public class Player:MonoBehaviour
{
    public void TakeDamage(int damage)
    {
        //...
    }
}
```

在图中,自定义 Player 类的单元如图 6-2 所示。

如果更改脚本并重命名或删除 TakeDamage()方法或 Player 类,则:

```
using UnityEngine;

public class Player:MonoBehaviour
{
    public void InflictDamage(int damage)
    {
        //...
    }
}
```

相关单元将在图形窗口中变为红色,Bolt 将向控制台记录一个警告,如图 6-3 所示。警告内容如下:

```
Failed to define Bolt.InvokeMember.
System.MissingMemberException:No matching member found: 'Player.TakeDamage'
```

图 6-2 自定义 Player 类的单元

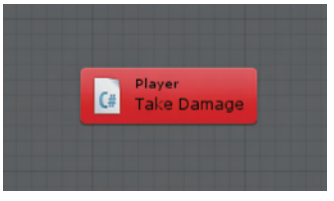

图 6-3 出错 Take Damage 单元