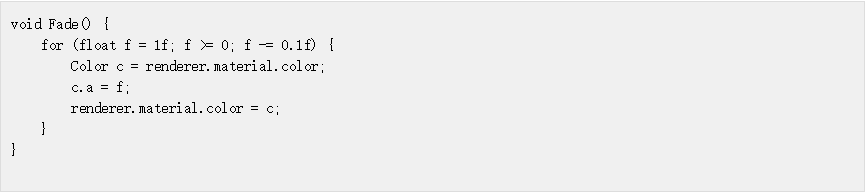
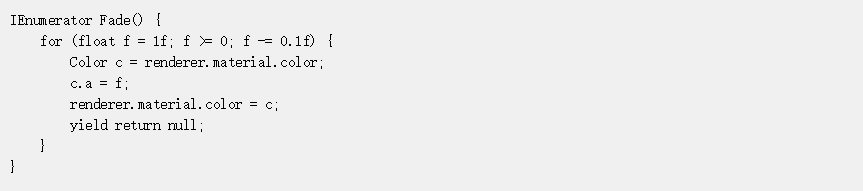
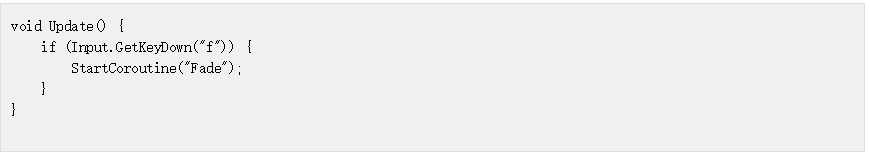
**协程**

调用函数时，函数将运行到完成状态，然后返回。这实际上意味着在函数中发生的任何动作都必须在单帧更新内发生；函数调用不能用于包含程序性动画或随时间推移的一系列事件。例如，假设需要逐渐减少对象的 Alpha（不透明度）值，直至对象变得完全不可见。

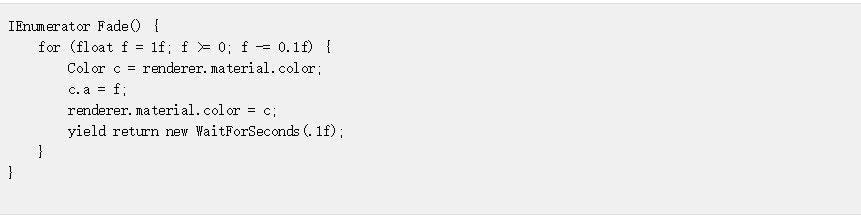
就目前而言，Fade 函数不会产生期望的效果。为了使淡入淡出过程可见，必须通过一系列帧降低 Alpha 以显示正在渲染的中间值。但是，该函数将完全在单个帧更新中执行。这种情况下，永远不会看到中间值，对象会立即消失。

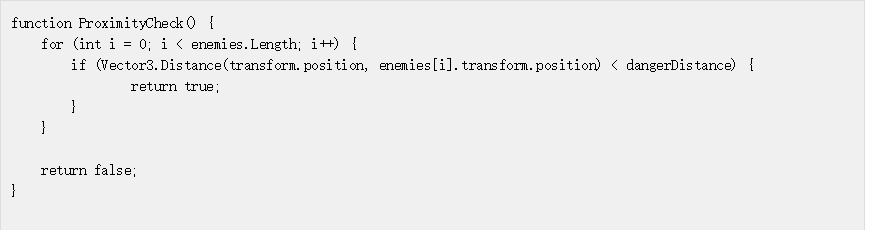
可以通过向 Update 函数添加代码（此代码逐帧执行淡入淡出）来处理此类情况。但是，使用协程来执行此类任务通常会更方便。

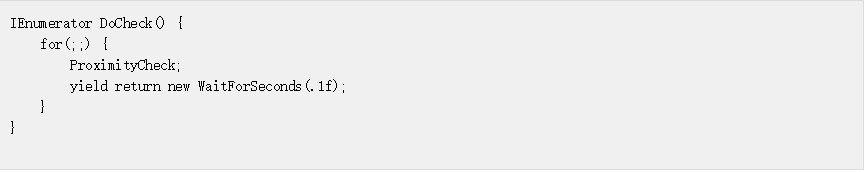
协程就像一个函数，能够暂停执行并将控制权返还给 Unity，然后在下一帧继续执行。在 C# 中，声明协程的方式如下：

此协程本质上是一个用返回类型 IEnumerator 声明的函数，并在主体中的某个位置包含 yield return 语句。yield return 行是暂停执行并随后在下一帧恢复的点。要将协程设置为运行状态，必须使用 [StartCoroutine](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.StartCoroutine.html) 函数：

您会注意到 Fade 函数中的循环计数器能够在协程的生命周期内保持正确值。实际上，在 yield 语句之间可以正确保留任何变量或参数。

默认情况下，协程将在执行 yield 后的帧上恢复，但也可以使用 [WaitForSeconds](file:///E:\\UnityDocumentation\\ScriptReference\\WaitForSeconds.html) 来引入时间延迟：

可通过这样的协程在一段时间内传播效果，但也可将其作为一种有用的优化。游戏中的许多任务需要定期执行，最容易想到的方法是将任务包含在 Update 函数中。但是，通常情况下，每秒将多次调用该函数。不需要以这样的频繁程度重复任务时，可以将其放在协程中来进行定期更新，而不是每一帧都更新。这方面的一个示例可能是在附近有敌人时向玩家发出的警报。此代码可能如下所示：

如果有很多敌人，那么每帧都调用此函数可能会带来很大开销。但是，可以使用协程，每十分之一秒调用一次：

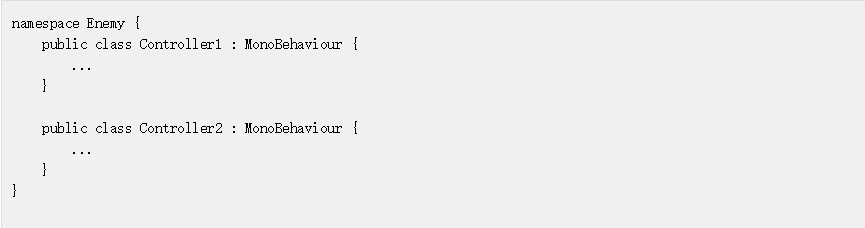
这将大大减少所进行的检查次数，而不会对游戏运行过程产生任何明显影响。

注意：禁用 MonoBehaviour 时，不会停止协程，仅在明确销毁 MonoBehaviour 时才会停止协程。可以使用 MonoBehaviour.StopCoroutine 和 MonoBehaviour.StopAllCoroutines 来停止协程。销毁 MonoBehaviour 时，也会停止协程。

# 命名空间

随着项目变得越来越大并且脚本数量增加，脚本类名称之间发生冲突的可能性也越来越大。当几个程序员分别处理游戏的不同方面并最终将他们的工作结合在一个项目中时，尤其如此。例如，一个程序员可能正在编写代码来控制主要玩家角色，而另一个程序员则为敌人编写等效的代码。两个程序员都可能选择调用他们的主脚本类 \_Controller\_，但这样会在他们的项目组合在一起时引起冲突。

在某种程度上，可通过采用命名约定或通过在发现冲突时对类重命名来避免此问题（例如，上面的类可命名为类似 PlayerController 和 EnemyController 的名称）。但是，当存在多个具有冲突名称的类或当使用这些名称来声明变量时，这很麻烦：必须替换每次出现的旧类名才能编译代码。

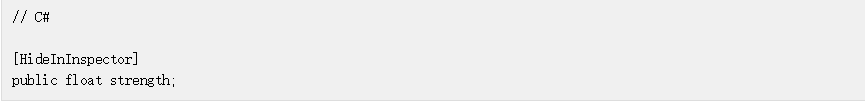
C# 语言提供了称为**命名空间**的功能，以一种可靠的方式解决了该问题。命名空间就是类的集合；引用集合中的类时需要在类名中使用所选的前缀。在下面的示例中，类 Controller1 和 Controller2 是名为 Enemy 的命名空间的成员：

在代码中，这些类分别以 Enemy.Controller1 和 Enemy.Controller2 的形式引用。就目前而言，这种解决方案比重命名类更好，因为命名空间声明可在现有类声明外层括起来（即，没有必要单独更改所有类的名称）。此外，无论类在任何位置，均可使用多个带括号的命名空间部分将类包裹起来，即使这些类位于不同的源文件中也是如此。

通过在文件顶部添加 **using** 指令可避免重复输入命名空间前缀。

此行表示，在出现类名 Controller1 和 Controller2 的位置，它们应分别表示 Enemy.Controller1 和 Enemy.Controller2。如果脚本还需要引用不同命名空间中同名的类（比如说名为 Player 的命名空间），那么仍然可以使用前缀。如果同时使用 using 指令导入两个包含冲突类名的命名空间，则编译器将报错。

# 特性

**特性 (Attribute)** 是可以放在脚本中的类、属性或函数上方来指示特殊行为的标记。例如，\_\_HideInInspector\_\_ 特性可以添加到属性声明中，从而防止该属性（即使是公共属性）显示在检视面板中。在 JavaScript 中，特性名称以“@”符号开头，而在 C# 中，它包含在方括号内：

Unity 提供了若干特性；请查看脚本参考（从侧栏的弹出菜单中选择 Editor 或 Runtime Attributes 部分）了解这些属性。.NET 库中还定义了一些有时可能在 Unity 代码中很有用的特性。

**注意：**不应使用 .NET 库中定义的 [ThreadStatic](http://msdn.microsoft.com/en-us/library/system.threadstaticattribute.aspx) 特性，因为如果将其添加到 Unity 脚本，会导致崩溃。

# 事件函数执行顺序

在 Unity 脚本中，有许多事件函数在脚本执行时按预定顺序执行。此执行顺序如下所述：

## 加载第一个场景

场景开始时将调用以下函数（为场景中的每个对象调用一次）。

* **Awake：**始终在任何 Start 函数之前并在实例化预制件之后调用此函数。（如果游戏对象在启动期间处于非活动状态，则在激活之后才会调用 Awake。）
* **OnEnable：**（仅在对象处于激活状态时调用）在启用对象后立即调用此函数。在创建 MonoBehaviour 实例时（例如加载关卡或实例化具有脚本组件的游戏对象时）会执行此调用。
* **OnLevelWasLoaded：**执行此函数可告知游戏已加载新关卡。

请注意，对于添加到场景中的对象，在为任何对象调用 Start 和 Update 等函数之前，会为\_所有\_脚本调用 Awake 和 OnEnable 函数。当然，在游戏运行过程中实例化对象时，不能强制执行此调用。

## Editor

* **Reset：**调用 Reset 可以在脚本首次附加到对象时以及使用 Reset 命令时初始化脚本的属性。

## 在第一次帧更新之前

* **Start：**仅当启用脚本实例后，才会在第一次帧更新之前调用 Start。

对于添加到场景中的对象，在为任何脚本调用 Update 等函数之前，将在所有脚本上调用 Start 函数。当然，在游戏运行过程中实例化对象时，不能强制执行此调用。

## 帧之间

* **OnApplicationPause：**在帧的结尾处调用此函数（在正常帧更新之间有效检测到暂停）。在调用 **OnApplicationPause** 之后，将发出一个额外帧，从而允许游戏显示图形来指示暂停状态。

## 更新顺序

跟踪游戏逻辑和交互、动画、摄像机位置等的时候，可以使用一些不同事件。常见方案是在 **Update** 函数中执行大多数任务，但是也可以使用其他函数。

* **FixedUpdate：**调用 **FixedUpdate** 的频度常常超过 **Update**。如果帧率很低，可以每帧调用该函数多次；如果帧率很高，可能在帧之间完全不调用该函数。在 **FixedUpdate** 之后将立即进行所有物理计算和更新。在 **FixedUpdate** 内应用运动计算时，无需将值乘以 **Time.deltaTime**。这是因为 **FixedUpdate** 的调用基于可靠的计时器（独立于帧率）。
* **Update：**每帧调用一次 **Update**。这是用于帧更新的主要函数。
* **LateUpdate：**每帧调用一次 **LateUpdate\_\_（在**Update\_\_ 完成后）。**LateUpdate** 开始时，在 **Update** 中执行的所有计算便已完成。**LateUpdate** 的常见用途是跟随第三人称摄像机。如果在 **Update** 内让角色移动和转向，可以在 **LateUpdate** 中执行所有摄像机移动和旋转计算。这样可以确保角色在摄像机跟踪其位置之前已完全移动。

## 渲染

* **OnPreCull：**在摄像机剔除场景之前调用。剔除操作将确定摄像机可以看到哪些对象。正好在进行剔除之前调用 OnPreCull。
* **OnBecameVisible/OnBecameInvisible：**对象变为对任何摄像机可见/不可见时调用。
* **OnWillRenderObject：**如果对象可见，则为每个摄像机调用**一次**。
* **OnPreRender：**在摄像机开始渲染场景之前调用。
* **OnRenderObject：**所有常规场景渲染完成之后调用。此时，可以使用 [GL](file:///E:\UnityDocumentation\ScriptReference\GL.html) 类或 [Graphics.DrawMeshNow](file:///E:\\UnityDocumentation\\ScriptReference\\Graphics.DrawMeshNow.html) 来绘制自定义几何形状。
* **OnPostRender：**在摄像机完成场景渲染后调用。
* **OnRenderImage：**在场景渲染完成后调用以允许对图像进行后处理，请参阅[后期处理效果](file:///E:\UnityDocumentation\Manual\PostProcessingOverview.html)。
* **OnGUI：**每帧调用多次以响应 GUI 事件。首先处理布局和重新绘制事件，然后为每个输入事件处理布局和键盘/鼠标事件。
* **OnDrawGizmos** 用于在场景视图中绘制辅助图标以实现可视化。

## 协程

Update 函数返回后将运行正常协程更新。协程是一个可暂停执行 (yield) 直到给定的 YieldInstruction 达到完成状态的函数。 协程的不同用法：

* **yield** 在下一帧上调用所有 Update 函数后，协程将继续。
* **yield WaitForSeconds** 在为帧调用所有 Update 函数后，在指定的时间延迟后继续协程
* **yield WaitForFixedUpdate** 在所有脚本上调用所有 FixedUpdate 后继续协程
* **yield WWW** 在 WWW 下载完成后继续。
* **yield StartCoroutine** 将协程链接起来，并会等待 MyFunc 协程先完成。

## 销毁游戏对象时

* **OnDestroy：**对象存在的最后一帧完成所有帧更新之后，调用此函数（可能应 Object.Destroy 要求或在场景关闭时销毁该对象）。

## 退出时

在场景中的所有活动对象上调用以下函数：

* **OnApplicationQuit：**在退出应用程序之前在所有游戏对象上调用此函数。在编辑器中，用户停止播放模式时，调用函数。
* **OnDisable：**行为被禁用或处于非活动状态时，调用此函数。

