**了解自动内存管理**

创建对象、字符串或数组时，用于存储它的内存是从称为**堆**的中央池分配的。当此项不再使用时，其先前占用的内存可被回收并用于其他目的。在过去，通常由程序员通过适当的函数调用显式地分配和释放这些堆内存块。如今，Unity 的 Mono 引擎等运行时系统会自动为您管理内存。自动内存管理比显式分配/释放的做法需要更少的编码工作，并且大大降低了内存泄漏的可能性（即分配了内存但后续从未释放的情况）。

**价值和引用类型**

调用某个函数时，其参数的值将复制到为该特定调用保留的内存区域。只占几个字节的数据类型可以非常快速和轻松地完成复制。但是，对象、字符串和数组通常要大得多，如果需要经常复制这些类型的数据，效率会非常低。幸运的是，并不是非要这样做；可从堆中分配大项的实际存储空间，并使用小“指针”值来记住它的位置。此后，在参数传递期间只需要复制指针。只要运行时系统能找到该指针所标识的项，就可以根据需要频繁使用该数据的同一个副本。

在参数传递期间直接存储和复制的类型称为值类型。这些类型包括整数、浮点数、布尔值和 Unity 的结构类型（例如，\_\_Color\_\_ 和 \_\_Vector3\_\_）。在堆上分配后再通过指针访问的类型称为引用类型，因为在变量中存储的值仅“引用”实际数据。引用类型的示例包括对象、字符串和数组。

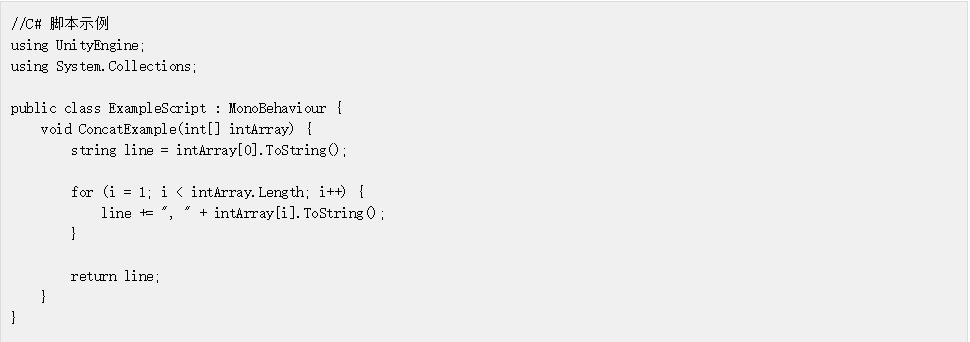
**分配和垃圾收集**

内存管理器跟踪已知未使用的堆区域。当请求新的内存块时（例如，当实例化对象时），管理器选择一个未使用的区域来分配内存块，然后从已知的未使用空间中移除分配的内存。后续请求以相同的方式处理，直到没有足够大的可用区域来分配所需的块大小。此时极不可能从堆中分配的所有内存都仍在使用中。若要访问堆上的引用项，前提是仍有引用变量可以定位到该项。如果对内存块的所有引用都消失（即，引用变量已被重新分配，或者引用变量是局部变量但现在已超出范围），则可安全地重新分配其占用的内存。

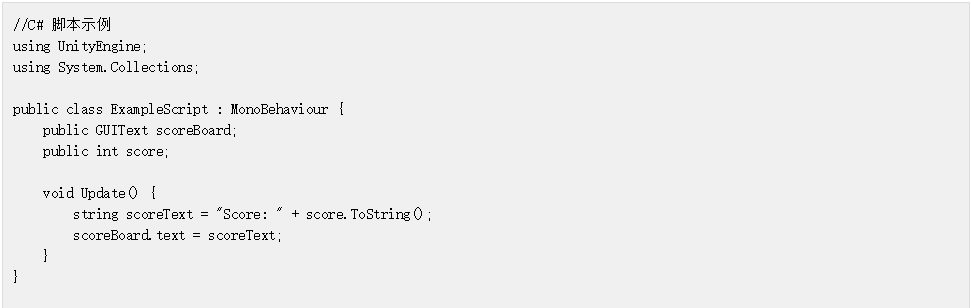
为确定哪些堆块已不再使用，内存管理器会搜索所有当前处于活动状态的引用变量，并将它们引用的块标记为“实时”。在搜索结束时，内存管理器会认为实时块之间的任何空间都是空的并可用于后续分配。由于显而易见的原因，定位和释放未使用的内存的过程称为垃圾收集（或简称 GC）。

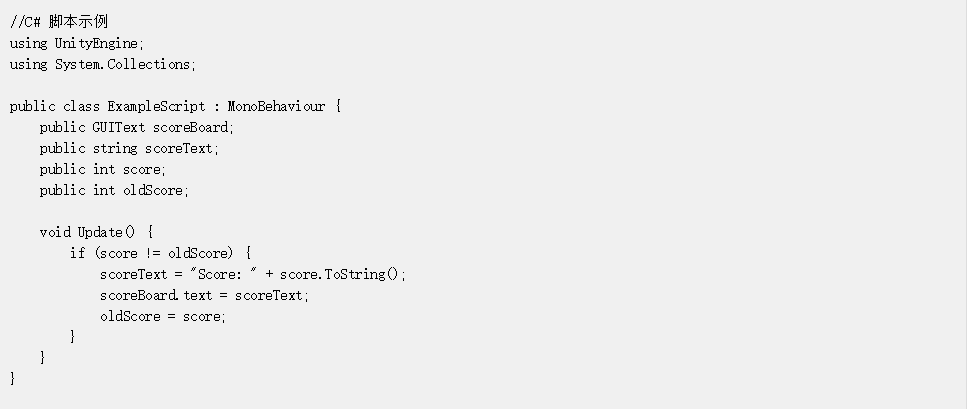
**优化**

垃圾收集是自动完成的，对程序员来说不可见，但收集过程实际上在后台需要耗费大量 CPU 时间。如果使用得当，自动内存管理通常在整体性能上能达到或超过手动分配。但是，程序员必须避免错误以免导致不必要的频繁触发垃圾回收器并在执行中引起暂停。

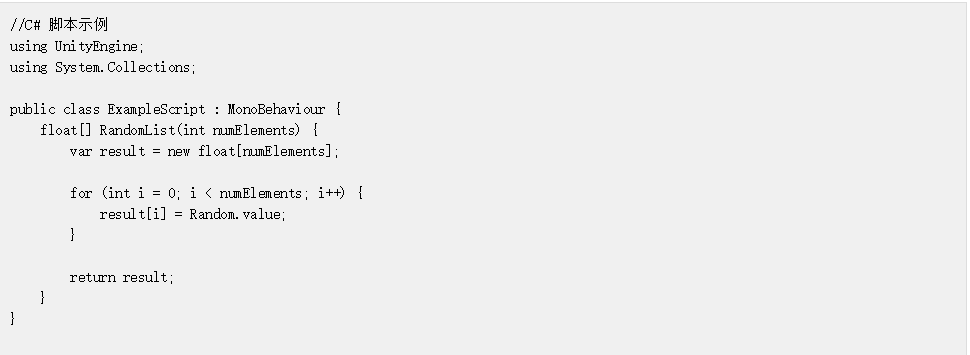
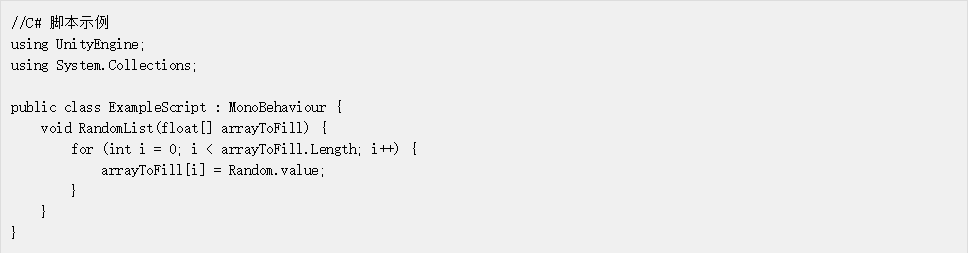
有一些臭名昭着的算法虽然一眼看上去好像是无辜的，但可能成为 GC 的噩梦。重复的字符串连接便是一个典型的例子：

此处的关键细节是新的部分不会逐一添加到字符串。实际情况的是，每次循环时，line 变量的先前内容变为死亡状态：分配的整个新字符串将包含原始部分加上末尾的新部分。由于字符串随着 i 值的增加而变长，因此消耗的堆空间量也会增加，所以每次调用此函数时都很容易用掉数百个字节的空闲堆空间。如果需要将大量字符串连接在一起，那么 Mono 库的 [System.Text.StringBuilder](http://msdn.microsoft.com/en-gb/library/system.text.stringbuilder.aspx) 类将是更好的选择。

但是，即使重复的连接也不会造成太大麻烦，除非频繁调用，而在 Unity 中这通常意味着帧更新。类似以下脚本：

…在每次调用 Update 时都会分配新的字符串，并生成源源不断的垃圾。通过仅在 score 发生变化时才更新 text，可避免大部分的垃圾：

另一个潜在问题是在函数返回数组值时出现的问题：

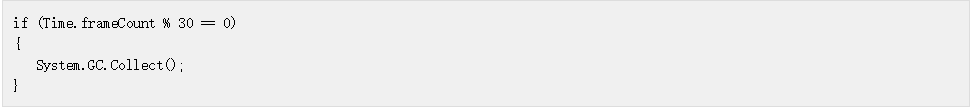
在新建包含值的数组时，这种类型的函数非常从容和方便。但是，如果重复调用这种函数，则每次都会分配全新的内存。由于数组可能非常大，因此空闲堆空间可能会迅速耗尽，导致频繁进行垃圾收集。避免此问题的一种方法是利用数组为引用类型这一特点。作为参数传入该函数的数组可在该函数内予以修改，且结果在函数返回后仍然保留。像上面这样的函数通常可替换为如下所示的函数：

此函数仅将数组的现有内容替换为新值。虽然这需要在调用代码中完成数组的初始分配（看起来有点不方便），但在调用该函数时不会产生任何新的垃圾。

## 请求进行垃圾收集

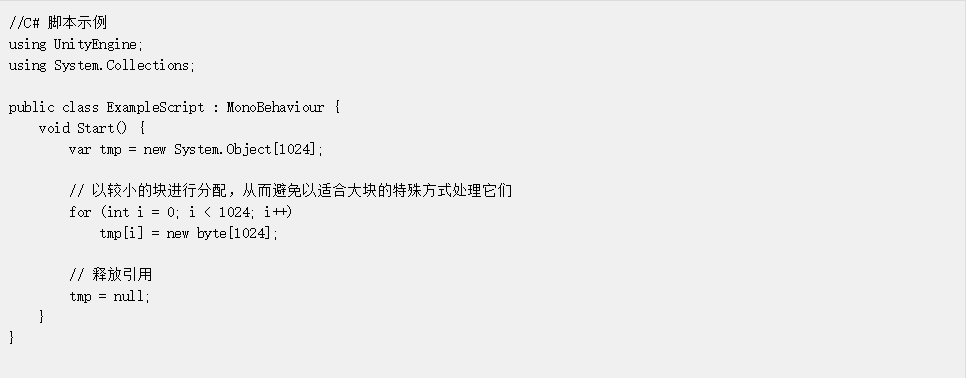
如上所述，最好尽量避免内存分配。但是，鉴于无法完全消除这些行为，可采用两种主要策略来最小化这些行为对游戏运行过程的干扰：

### 快速和频繁进行垃圾收集的小堆

这种策略通常最适合游戏运行过程很长且主要关注帧率平滑性的游戏。像这样的游戏通常会频繁分配小块，但这些块的使用时间很短暂。在 iOS 上使用此策略时的典型堆大小约为 200KB，在 iPhone 3G 上的垃圾收集时间大约需要 5ms。如果堆大小增加到 1MB，则收集时间将大约需要 7ms。因此，有时，以定期的帧间隔请求进行垃圾收集可能是有利的。这种情况下通常会使垃圾收集频率高于严格意义上的要求，但是这些行为将得到快速处理，并且对游戏运行过程的影响极小：

但是，应谨慎使用此技术并检查性能分析器的统计信息，以确保真正减少了游戏的垃圾收集时间。

### 慢速但不频繁进行垃圾收集的大堆

这种策略最适合内存分配（因此垃圾收集）相对不频繁并可在游戏运行过程的暂停期间进行处理的游戏。一种非常有用的方法是，尽可能增大堆的大小，但不至于因为系统内存不足而导致操作系统终止您的应用程序。但是，Mono 运行时会尽可能避免自动扩展堆。这种情况下，可通过在启动期间预先分配一些占位空间来手动扩展堆（即，实例化一个纯粹为了影响内存管理器而分配的“无用”对象）：

一个足够大的堆不应在游戏运行过程中配合进行垃圾收集的暂停期间完全耗尽。发生此类暂停时，可显式请求垃圾收集：

System.GC.Collect();

同样，在使用此策略时应谨慎，并注意性能分析器的统计信息，而不能仅仅期待其具有所需的效果。

## 可重用的对象池

在许多情况下，通过减少创建和销毁的对象数量即可避免生成垃圾。游戏中存在某些类型的对象，例如飞弹，可能会多次反复遇到，但是只有少数对象会同时处于游戏中。在这种情况下，通常可以重用对象，而不是销毁旧对象并替换为新对象。

## 其他信息

内存管理是一个微妙而复杂的主题，业界已投入了大量的学术努力。如果有兴趣了解这一主题，[memorymanagement.org](http://www.memorymanagement.org/) 将是极好的资源，其中列出了大量出版物和在线文章。如需了解对象池的更多信息，请访问 [Wikipedia 页面](http://en.wikipedia.org/wiki/Object_pool_pattern)以及 [Sourcemaking.com](http://sourcemaking.com/design_patterns/object_pool)。

# 特殊文件夹和脚本编译顺序

Unity 保留了一些项目文件夹名称来指示内容具有特殊用途。其中一些文件夹会影响脚本编译的顺序。这些文件夹名称为：

* Assets
* Editor
* Editor default resources
* Gizmos
* Plugins
* Resources
* Standard Assets
* StreamingAssets

请参阅[特殊文件夹名称](file:///E:\UnityDocumentation\Manual\SpecialFolders.html)了解这些文件夹的用途。

脚本编译过程有四个独立的阶段。编译脚本的阶段由其父文件夹确定。

在脚本必须引用其他脚本中定义的类的情况下，这很重要。基本规则是无法引用在当前阶段之后的阶段编译的任何内容。在当前阶段或早期阶段编译的所有内容则是完全可用的。

当用一种语言编写的脚本必须引用另一种语言定义的类（例如，UnityScript 文件声明 C# 脚本中定义的类的变量）时，会出现另一种情况。这里的规则是被引用的类必须在早期阶段完成编译。

编译阶段如下：

* **第 1 阶段：**名为 **Standard Assets**、**Pro Standard Assets** 和 **Plugins** 的文件夹中的运行时脚本。
* **第 2 阶段：**名为 **Editor** 的文件夹（位于名为 **Standard Assets**、**Pro Standard Assets** 和 **Plugins** 的顶层文件夹中的任意位置）中的 Editor 脚本。
* **第 3 阶段：**不在名为 **Editor** 的文件夹中的所有其他脚本。
* **第 4 阶段：**其余所有脚本（位于名为 **Editor** 的文件夹中的脚本）。

体现此顺序非常重要的一个常见示例是 UnityScript 文件需要引用 C# 文件中定义的类时。要实现此引用，必须将 C# 文件放在 Plugins 文件夹中，并将 UnityScript 文件放在非特殊文件夹中。如果不这样做，则会抛出错误，指出无法找到 C# 类。

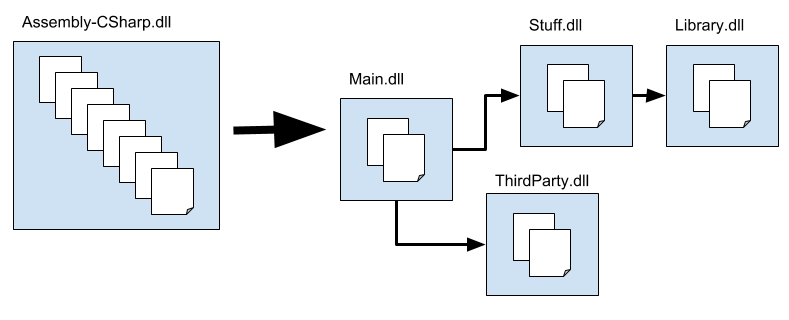
**注意**：Standard Assets 仅在 **Assets** 根文件夹中有效。

**脚本编译和程序集定义文件**

**关于**

Unity 自动定义脚本如何编译到托管程序集。通常情况下，随着向项目添加的脚本越来越多，Unity Editor 中用于迭代脚本更改的编译时间也会相应增加。

可使用程序集定义文件根据文件夹中的脚本来定义您自己的托管程序集。为此，请将项目脚本分成具有明确定义的依赖项的多个程序集，以确保在脚本中进行更改时仅重新构建所需的程序集。这样可以减少编译时间。应将每个托管程序集视为 Unity 项目中的单个库。

图 1 - 脚本编译

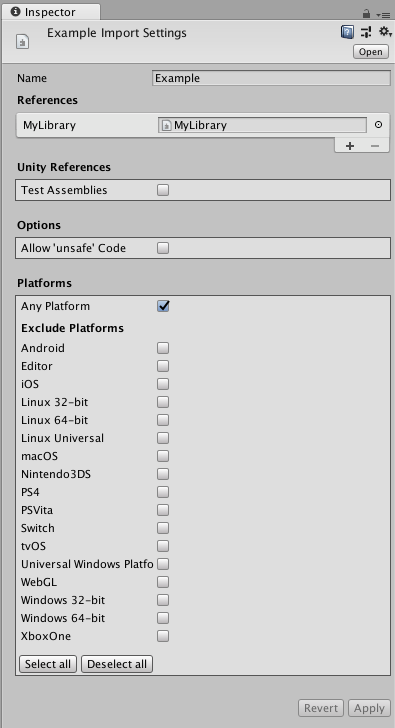
上面的**图 1** 说明了如何将项目脚本拆分为多个程序集。仅更改 Main.dll 中的脚本不会导致其他任何程序集重新编译。由于 Main.dll 包含的脚本较少，因此编译速度也比 CSharp.dll 快。 同样，只有 Stuff.dll 中的脚本更改才会导致 Main.dll 和 Stuff.dll 重新编译。

**如何使用程序集定义文件**

程序集定义文件是通过 **Assets** > **Create** > **Assembly Definition** 创建的资源文件。这些文件的扩展名为 .asmdef。

将一个程序集定义文件添加到 Unity 项目中的某个文件夹，即可将该文件夹中的所有脚本编译为一个程序集。可在 Inspector 中定义程序集的名称。

**注意**：程序集定义文件所在的文件夹的名称以及程序集定义文件的文件名对程序集的名称没有影响。

图 2 - Example Import Settings

添加对项目其他程序集定义文件的引用也需要使用 Inspector。要查看 Inspector，请单击程序集定义文件，随后应该会出现该面板。要添加引用，请单击 **References** 部分下的 **+** 图标，然后选择您的文件。

Unity 使用引用来编译程序集，还定义程序集之间的依赖关系。

要标记该程序集以进行测试，请在 Inspector 中启用 Test Assemblies。这样会在程序集定义文件中添加对 unit.framework.dll 和 UnityEngine.TestRunner.dll 的引用。

在标记程序集以进行测试时，请确保：

* 预定义的程序集（Assembly-CSharp.dll 等）不会自动引用标记为进行测试的程序集定义文件。
* 该程序集未包含在正常构建中。要将程序集包含在播放器构建中，请在构建脚本中使用 [BuildOptions.IncludeTestAssemblies](file:///E:\\UnityDocumentation\\ScriptReference\\BuildOptions.IncludeTestAssemblies.html)。请注意，这样只会在构建中包含程序集，而不会执行任何测试。

**注意**：如果在程序集内的脚本中使用 *unsafe* 关键字，必须在 Inspector 中启用 **Allow ‘unsafe’ Code** 选项。这样将在编译程序集时将 /unsafe 选项传递给 C# 编译器。

可在 Inspector 中为程序集定义文件设置平台兼容性。在此处可选择排除或包含特定平台。

**文件夹层级视图中的多个程序集定义文件**

在文件夹层级视图中保留多个程序集定义文件（扩展名：.asmdef）会使每个脚本添加到具有最短路径距离的程序集定义文件。

示例：

如果有一个 *Assets/ExampleFolder/MyLibrary.asmdef* 文件和一个 *Assets/ExampleFolder/ExampleFolder2/Utility.asmdef* 文件， 则：

* 位于 **Assets** > **ExampleFolder** > **ExampleFolder2** 文件夹内的任何脚本都将编译为 Assets/ExampleFolder/ExampleFolder2/Utility.asmdef 定义的程序集。
* 位于 **Assets** > **ExampleFolder** 文件夹内而不在 **Assets** > **ExampleFolder** > **ExampleFolder2** 文件夹内的任何文件都将编译为 Assets/ExampleFolder/MyLibrary.asmdef 定义的程序集。

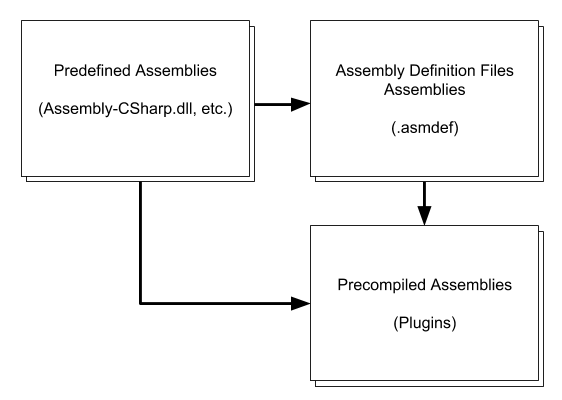
**程序集定义文件不是构建系统文件**

**注意**：程序集定义文件不是程序集构建文件。它们不支持构建系统中常见的条件构建规则。

也正因为如此，程序集定义文件不支持预处理器指令（define 指令）的设置，因为它们始终为静态。

**向后兼容性和隐式依赖关系**

程序集定义文件向后兼容 Unity 中的现有[预定义编译系统](file:///E:\\UnityDocumentation\\Manual\\ScriptCompileOrderFolders.html)。这意味着预定义的程序集始终依赖于每个程序集定义文件的程序集。 这类似于所有脚本都依赖于与 Unity 中的活动构建目标兼容的所有预编译程序集（插件/.dll）。

图 3 - 程序集依赖关系

**图 3** 中的图表说明了预定义程序集、程序集定义文件程序集和预编译程序集之间的依赖关系。

Unity 为程序集定义文件设定的优先级高于[预定义编译系统](file:///E:\\UnityDocumentation\\Manual\\ScriptCompileOrderFolders.html)。 这意味着，在程序集定义文件的文件夹中具有来自预定义编译的任何特殊文件夹名称对编译没有任何影响。Unity 将这些文件夹视为没有任何特殊含义的常规文件夹。

强烈建议要么对项目中的所有脚本使用程序集定义文件，要么根本不使用。否则，程序集定义文件每次重新编译时，不使用程序集定义文件的脚本也会始终重新编译。这样就弱化了使用程序集定义文件的优势。

**API**

在 UnityEditor.Compilation 命名空间中，有一个静态的 [CompilationPipeline](file:///E:\\UnityDocumentation\\ScriptReference\\Compilation.CompilationPipeline.html) 类用于检索关于程序集定义文件和 Unity 构建的所有程序集的信息。

**文件格式**

程序集定义文件是 JSON 文件。它们具有以下字段：

| **字段** | **类型** |
| --- | --- |
| **name** | string |
| **references（可选）** | 字符串数组 |
| **includePlatforms（可选）** | 字符串数组 |
| **excludePlatforms（可选）** | 字符串数组 |

不要在同一程序集定义文件中同时使用 **includePlatforms** 和 **excludePlatforms** 字段。

可使用 CompilationPipeline.GetAssemblyDefinitionPlatforms 检索平台名称。

**示例**

MyLibrary.asmdef

{

"name": "MyLibrary",

"references": [ "Utility" ],

"includePlatforms": ["Android", "iOS"]

}

MyLibrary2.asmdef

{

"name": "MyLibrary2",

"references": [ "Utility" ],

"excludePlatforms": ["WebGL"]

}

**.NET 配置文件支持**

Unity 支持许多 .NET 配置文件。每个配置文件为 C# 代码提供不同的 API 表面来与 .NET 类库交互。可在 Player Settings（选择 **Edit** > **Project Settings** > **Player\_\_）中更改 .NET 配置文件，为此需要使用**Other Settings\_\_ 部分的 **Api Compatibility Level** 选项。

**旧版脚本运行时**

旧版脚本运行时支持两种不同的配置文件：.NET 2.0 Subset 和 .NET 2.0。这两者都与 Microsoft 的 .NET 2.0 配置文件密切相关。.NET 2.0 Subset 配置文件小于 .NET 4.x 配置文件，可用于访问大多数 Unity 项目使用的类库 API。此配置文件是大小受限平台（比如移动平台）的理想选择，并提供了一组可移植的 API 来实现多平台支持。默认情况下，大多数 Unity 项目应使用 .NET Standard 2.0 配置文件。

**稳定脚本运行时**

稳定脚本运行时支持两种不同的配置文件：.NET Standard 2.0 和 .NET 4.x。 .NET Standard 2.0 配置文件的名称可能有点误导，因为该配置文件与来自旧版脚本运行时的 .NET 2.0 和 .NET 2.0 Subset 配置文件无关。相反，Unity 支持的 .NET Standard 2.0 配置文件对应于 .NET Foundation 发布的同名配置文件。Unity 中的 .NET 4.x 配置文件对应于来自 .NET Framework 的 .NET 4 系列（.NET 4.5、.NET 4.6、.NET 4.7 等等）配置文件。

仅当需要确保与外部库的兼容性时，或者需要的功能在 .NET Standard 2.0 中不可用时，才应使用 .NET 4.x 配置文件。

**跨平台兼容性**

Unity 致力于在所有平台上支持 .NET Standard 2.0 配置文件中的绝大多数 API。虽然并非所有平台都完全支持 .NET Standard，但是旨在实现跨平台兼容性的库应指向 .NET Standard 2.0 配置文件。.NET 4.x 配置文件包含的 API 表面要大得多，包括可能在很少平台上运行甚至无法在任何平台上运行的部分。

**托管插件**

在 Unity 外部编译的托管代码插件可使用 Unity 中的 .NET Standard 2.0 配置文件或 .NET 4.x 配置文件。下表描述了 Unity 支持的配置：

|  | API Compatibility Level: | |
| --- | --- | --- |
|  | **.NET Standard 2.0** | **.NET 4.x** |
| Managed plugin compiled against: |  |  |
| **.NET Standard** | 受支持 | 受支持 |
| **.NET Framework** | 受限 | 受支持 |
| **.NET Core** | Not Supported | Not Supported |

**注意**：

* 根据任何 .NET Standard 版本编译的托管插件均可用于 Unity。
* 有限支持表示，如果使用的来自 .NET Framework 的所有 API 都存在于 .NET Standard 2.0 配置文件中，则 Unity 支持该配置。但是，.NET Framework API 是 .NET Standard 2.0 配置文件的超集，因此有些 API 不可用。

**引用其他类库程序集**

如果 Unity 项目需要访问默认情况下未编译的 .NET 类库 API 的一部分，则项目可以通知 Unity 中的 C# 编译器。此行为取决于项目使用的 .NET 配置文件。

**.NET Standard 2.0 配置文件**

如果项目使用 .NET Standard 2.0 **API 兼容性级别\_\_，应该不需要采取任何其他步骤来使用 .NET 类库 API 的一部分。如果此 API 的一部分似乎丢失，可能是 .NET Standard 2.0 未随附此部分。项目可能需要改用 .NET 4.x**API 兼容性级别\_\_。

**.NET 4.x 配置文件**

默认情况下，Unity 在使用 .NET 4.x \_\_API 兼容性级别\_\_时引用以下程序集：

* mscorlib.dll
* System.dll
* System.Core.dll
* System.Runtime.Serialization.dll
* System.Xml.dll
* System.Xml.Linq.dll

应使用 *mcs.rsp* 文件来引用所有其他类库程序集。可将此文件添加到 Unity 项目的 Assets 目录，然后使用该文件将其他命令行参数传递到 C# 编译器。例如，如果项目使用 HttpClient 类（在 *System.Net.Http.dll* 程序集中定义），C# 编译器可能生成以下初始错误消息：

The type `HttpClient` is defined in an assembly that is not referenced.You must add a reference to assembly 'System.Net.Http, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a'.

可通过将以下 mcs.rsp 文件添加到项目来解决此错误：

-r:System.Net.Http.dll

应按照以上示例中所述引用类库程序集。请勿将这些程序集复制到 Project 目录中。

**切换配置文件**

使用 *mcs.rsp* 文件来引用类库程序集时，务必谨慎。如果将 \_*API 兼容性级别\_\_从 .NET 4.x 更改为 .NET Standard 2.0，而 Project 中存在类似于以上示例的*mcs.rsp\_，则 C# 编译会失败。*System.Net.Http.dll* 程序集并未存在于 .NET Standard 2.0 配置文件中，因此 C# 编译器无法找到该程序集。

*mcs.rsp* 文件可能具有特定于当前 .NET 配置文件的部分。如果更改此配置文件，则需要修改 *mcs.rsp* 文件。

# 稳定脚本运行时：已知限制

Unity 支持新版的 .NET 运行时。使用 .NET 运行时的时候，可能会遇到以下问题：

## 代码大小

稳定脚本运行时随附比旧版脚本运行时更大的 .NET 类库 API。这意味着代码大小通常更大。这种大小增加可能很明显，特别是在大小受限和提前 (AOT) 平台上。

要减少代码大小的增加幅度，请执行以下操作：

1.尽可能选择最小的 .NET 配置文件（请参阅 [.NET 配置文件支持](file:///E:\UnityDocumentation\Manual\dotnetProfileSupport.html)）。.NET Standard 2.0 配置文件大小大约只有 .NET 4.x 配置文件的一半，因此请尽可能使用 .NET Standard 2.0 配置文件。 1.在 Unity Editor 的 Player Settings（选择 **Edit** > **Project Settings** > **Player\_\_）中，启用**Strip Engine Code\_\_。此选项会静态分析项目中的托管代码，并删除所有未使用的代码。**注意**：只有 IL2CPP 脚本后端才附带此选项。

## 传输层安全性 (TLS)

新版 Mono 具有针对若干平台的 TLS 1.2 支持。Unity 支持 .NET 类库中的 TLS 1.0，而 TLS 仅对 Mac 独立平台播放器有效。在需要全面 TLS 支持的情况下，请使用 [UnityWebRequest](file:///E:\\UnityDocumentation\\Manual\\UnityWebRequest.html) 或特定于平台的本机解决方案。Unity 正积极致力于在 Unity 支持的所有平台上为所有 .NET 类库 API 添加 TLS 1.2 支持

# 通用函数

脚本参考中列出的某些函数（例如，各种 GetComponent 函数）附带有一个变体， 该变体在函数名称后的尖括号中具有字母 T 或类型名称：

//C#

void FuncName<T>();

这些函数称为通用函数。这些函数对脚本的重要性在于， 可以在调用函数时指定参数类型和/或返回类型。在 JavaScript 中， 这种做法可以规避动态输入的限制：

// 可正确推断出类型，因为已在函数调用中定义该类型。

//在 C# 中

var obj = GetComponent<Rigidbody>();

在 C# 中，这样可以节省很多击键和转换：

Rigidbody rb = go.GetComponent<Rigidbody>();

// ...相较于：

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));

在脚本参考页面上列出了通用变体的所有函数都会允许这种 特殊调用语法。

# 脚本限制

我们致力于在 Unity 支持的所有平台之间提供通用的脚本 API 和体验。但是，有些平台存在固有的限制。为帮助您了解这些限制并支持跨平台代码，下表描述了每个平台和脚本后端适用的限制：

## .NET 4.x 等效脚本运行时

|  |
| --- |
|  |
| ***平台（脚本后端）*** | ***提前编译*** | ***无线程*** | ***.NET Core 类库子集*** | |
| Android (IL2CPP) | √ |  |  | |
| Android (Mono) |  |  |  | |
| iOS (IL2CPP) | √ |  |  | |
| PlayStation 4 (IL2CPP) | √ |  |  | |
| PlayStation Vita (IL2CPP) | √ |  |  | |
| 独立平台 (IL2CPP) | √ |  |  | |
| 独立平台 (Mono) |  |  |  | |
| Switch (IL2CPP) | √ |  |  | |
| 通用 Windows 平台 (IL2CPP) | √ |  |  | |
| 通用 Windows 平台 (.NET) |  |  | √ | |
| WebGL (IL2CPP) | √ | √ |  |
| WiiU (Mono) |  |  |  | |  |
| XBox One (IL2CPP) | √ |  |  | |  |

## .NET 3.5 等效脚本运行时

|  |
| --- |
|  |
| ***平台（脚本后端）*** | ***提前编译*** | ***无线程*** | ***.NET Core 类库子集*** | |
| Android (IL2CPP) | √ |  |  | |
| Android (Mono) |  |  |  | |
| iOS (IL2CPP) | √ |  |  | |
| PlayStation 4 (IL2CPP) | √ |  |  | |
| PlayStation 4 (Mono) | √ |  |  | |
| PlayStation Vita (IL2CPP) | √ |  |  | |
| PlayStation Vita (Mono) | √ |  |  | |
| 独立平台 (IL2CPP) | √ |  |  | |
| 独立平台 (Mono) |  |  |  | |
| Switch (IL2CPP) | √ |  |  | |
| 通用 Windows 平台 (IL2CPP) | √ |  |  | |
| 通用 Windows 平台 (.NET) |  |  | √ | |
| WebGL (IL2CPP) | √ | √ |  |
| WiiU (Mono) |  |  |  | |  |
| XBox One (IL2CPP) | √ |  |  | |  |
| XBox One (Mono) | √ |  |  | |  |

## 提前编译

有些平台不允许生成运行时代码。因此，任何依赖于在目标设备上即时 (JIT) 编译的托管代码都将失败。相反，我们需要提前 (AOT) 编译所有托管代码。通常，这种区别并不重要，但在少数特定情况下，AOT 平台需要额外注意。

### System.Reflection.Emit

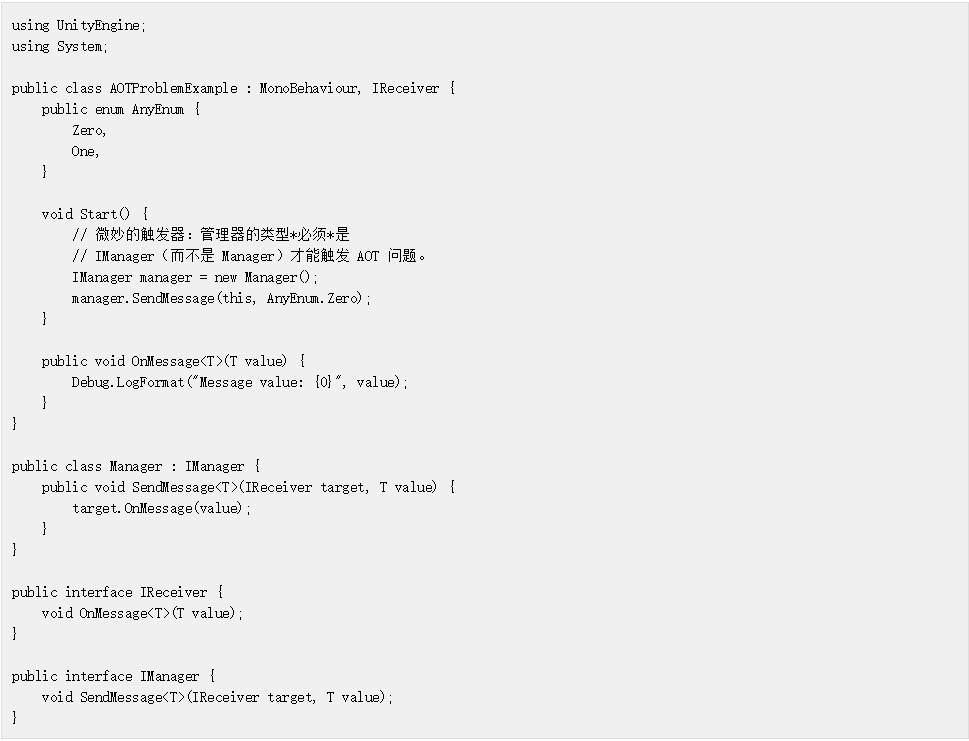
AOT 平台无法实现 **System.Reflection.Emit** 命名空间中的任何方法。请注意，**System.Reflection** 的其余部分是可接受的，只要 编译器 可以推断通过反射 使用的代码需要在运行时存在。

### 序列化

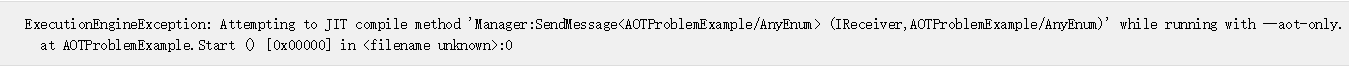
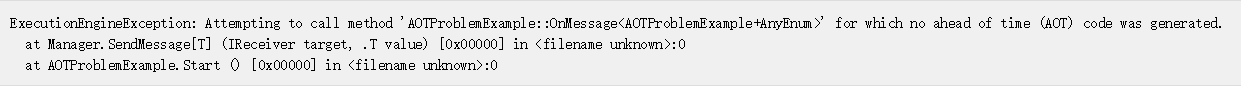
AOT 平台可能会由于使用了反射而遇到序列化和反序列化问题。如果仅通过反射将某个类型或方法作为序列化或反序列化的一部分使用，则 AOT 编译器无法检测到需要为该类型或方法生成代码。

### 通用虚拟方法

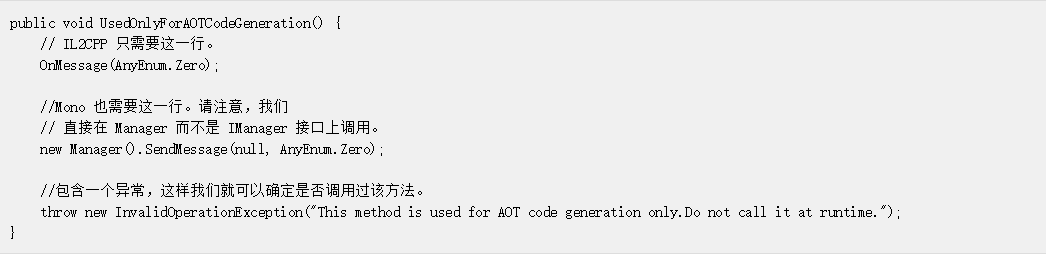
通用方法要求编译器做一些额外的工作，将开发人员编写的代码扩展到设备上实际执行的代码。例如，对于具有 **int** 或 **double** 类型的 **List**，我们需要不同代码。虚拟方法在运行时而不是编译时确定行为，存在虚拟方法时，编译器可在不完全明显的地方轻松地要求从源代码生成运行时代码。

假设有以下代码，此代码在 JIT 平台上完全按预期工作（向控制台输出一次“Message value: Zero”）：

使用 IL2CPP 脚本后端在 AOT 平台上执行此代码时，发生以下异常：

同样，Mono 脚本后端提供以下类似的异常：

AOT 编译器不会意识到自己应该为 **T** 为 **AnyEnum** 的泛型方法 **OnMessage** 生成代码，所以它直接继续往下，跳过该方法。调用该方法时，运行时无法找到要执行的正确代码，因此抛出此错误消息。

要解决像这样的 AOT 问题，我们通常可以强制编译器为我们生成适当的代码。如果我们向 **AOTProblemExample** 类添加如下所示的方法：

当编译器遇到 **T** 为 **AnyEnum** 的 **OnMessage** 的显式调用时，它会生成运行时执行的正确代码。不需要调用 **UsedOnlyForAOTCodeGeneration** 方法；该方法的存在只是为了让编译器看到而已。

## 无线程

有些平台不支持使用线程，因此任何使用 **System.Threading** 命名空间的托管代码都将在运行时失败。此外，.NET 类库的某些部分存在对线程的隐式依赖。一个常用的例子是 **System.Timers.Timer** 类，它依赖于对线程的支持