**脚本序列化**

序列化是将数据结构或对象状态转换为 Unity 可存储并在以后可重构的格式的自动过程。Unity 的一些内置功能会使用序列化，比如保存和加载、Inspector 窗口、实例化和预制件等功能。请参阅有关[内置序列化使用情况](file:///E:\\UnityDocumentation\\Manual\\script-Serialization-BuiltInUse.html)的文档，了解所有相关的背景详情。

数据在 Unity 项目中的组织方式会影响 Unity 序列化该数据的方式，并会对项目的性能产生重大影响。以下是有关 Unity 中的序列化以及如何针对其进行项目优化的一些准则。

另请参阅以下相关文档：[序列化错误](file:///E:\UnityDocumentation\Manual\script-Serialization-Errors.html)、[自定义序列化](file:///E:\UnityDocumentation\Manual\script-Serialization-Custom.html)和[内置序列化](file:///E:\UnityDocumentation\Manual\script-Serialization-BuiltInUse.html)。

**了解热重载**

**热重载**

热重载是在 Editor 打开的状态下创建或编辑脚本并立即应用脚本行为的过程。无需重新启动游戏和 Editor 即可使更改生效。

更改并保存脚本时，Unity 会热重载所有当前加载的脚本数据。它首先将所有可序列化变量存储在所有加载的脚本中，并在加载脚本后恢复它们。热重载后，所有不可序列化的数据都将丢失。

**保存和加载**

Unity 使用序列化技术从 计算机的硬盘驱动器 加载和保存 [场景](file:///E:\UnityDocumentation\Manual\CreatingScenes.html)、[资源](file:///E:\UnityDocumentation\Manual\AssetWorkflow.html)和 [AssetBundle](file:///E:\UnityDocumentation\Manual\AssetBundlesIntro.html)。这包括保存在您自己的脚本 API 对象（如 [MonoBehaviour](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.html) 组件和 [ScriptableObject](file:///E:\\UnityDocumentation\\Manual\\class-ScriptableObject.html)）中的数据。

Unity Editor 中的许多功能都建立在核心序列化系统之上。对于序列化要特别注意的两点是 [Inspector 窗口](file:///E:\UnityDocumentation\Manual\UsingTheInspector.html)和热重载。

**Inspector 窗口**

在 [Inspector 窗口](file:///E:\UnityDocumentation\Manual\UsingTheInspector.html)中查看或更改游戏对象的组件字段的值时，Unity 会序列化此数据，然后在 Inspector 窗口中显示数据。Inspector 窗口在显示字段值时不与 Unity Scripting API 通信。

如果在脚本中使用属性，则在 Inspector 窗口中查看或更改值时，绝不会调用任何属性 getter 和 setter，因为 Unity 会直接序列化 Inspector 窗口字段。这意味着：当 Inspector 窗口中的字段值表示脚本属性时，对 Inspector 窗口中值的更改不会调用脚本中的任何属性 getter 和 setter

**序列化规则**

Unity 中的序列化程序在实时游戏环境中运行。这对性能有重大影响。因此，Unity 中的序列化与其他编程环境中的序列化具有不同的行为。下面列出了一些关于如何在 Unity 中使用序列化的技巧。

**如何确保脚本中的字段被序列化**

确保其符合以下条件：

* 为 public，或具有 [SerializeField](file:///E:\\UnityDocumentation\\ScriptReference\\SerializeField.html) 属性
* 非 static
* 非 const
* 非 readonly
* 具有可序列化的 fieldtype  
  （请参阅下面的[可序列化的简单字段类型](file:///E:\\UnityDocumentation\\Manual\\script-Serialization.html" \l "FieldSerliaized2)。）

**可序列化的简单字段类型**

* 具有 Serializable 属性的自定义非抽象、非泛型类  
  （请参阅下面的[如何确保自定义类可序列化](file:///E:\UnityDocumentation\Manual\script-Serialization.html#ClassSerialized)。）
* 具有 Serializable 属性的自定义结构
* 对从 [UnityEngine.Object](file:///E:\\UnityDocumentation\\ScriptReference\\Object.html) 派生的对象的引用
* 原始数据类型（int、float、double、bool、string 等）
* 枚举类型
* 某些 Unity 内置类型：Vector2、Vector3、Vector4、Rect、Quaternion、Matrix4x4、Color、Color32、LayerMask、AnimationCurve、Gradient、RectOffset、GUIStyle

**可序列化的容器字段类型**

* 可序列化的简单字段类型的数组
* 可序列化的简单字段类型的 List<T>

**注意**：Unity 不支持多级类型（多维数组、交错数组和嵌套容器类型）的序列化。  
如果要序列化这些类型，可使用两种方法：将嵌套类型包装在类或结构中，或使用序列化回调 [ISerializationCallbackReceiver](file:///E:\\UnityDocumentation\\ScriptReference\\ISerializationCallbackReceiver.html) 执行自定义序列化。有关更多信息，请参阅[自定义序列化](file:///E:\\UnityDocumentation\\Manual\\script-Serialization-Custom.html)的文档。

**如何确保自定义类可序列化**

确保其符合以下条件：

* 具有 [Serializable](file:///E:\UnityDocumentation\ScriptReference\Serializable.html) 属性
* 非抽象
* 非静态
* 非泛型（但可继承自泛型类）

要确保自定义类或结构的字段被序列化，请参阅上面的[如何确保脚本中的字段被序列化](file:///E:\UnityDocumentation\Manual\script-Serialization.html#FieldSerliaized1)。

**序列化程序何时可能出现意外行为？**

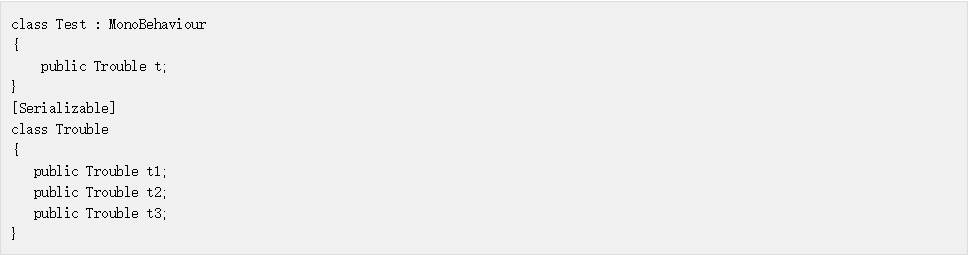
**自定义类的行为类似于结构**

对于不是从 [UnityEngine.Object](file:///E:\\UnityDocumentation\\ScriptReference\\Object.html) 派生的自定义类，Unity 以内联方式按值对它们进行序列化，类似于结构的序列化方式。如果在多个不同的字段中存储对自定义类的实例的引用，则在序列化时它们将成为单独的对象。然后，当 Unity 反序列化这些字段时，它们将包含具有相同数据的不同对象。

需要序列化具有引用的复杂对象图时，不要让 Unity 自动序列化对象。相反，应使用 ISerializationCallbackReceiver 手动序列化它们。这样可以防止 Unity 从对象引用创建多个对象。有关更多信息，请参阅 [ISerializationCallbackReceiver](file:///E:\\UnityDocumentation\\ScriptReference\\ISerializationCallbackReceiver.html) 的文档。

这种情况仅适用于自定义类。Unity 以“内联”方式对自定义类进行序列化，因为这些类的数据会成为使用这些类的 [MonoBehaviour](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.html) 或 [ScriptableObject](file:///E:\\UnityDocumentation\\ScriptReference\\ScriptableObject.html) 的完整序列化数据的一部分。当字段引用 [UnityEngine.Object](file:///E:\\UnityDocumentation\\ScriptReference\\Object.html) 派生的某个类（例如 public Camera myCamera）时，Unity 会序列化对摄像机 UnityEngine.Object的实际引用。如果脚本派生自 MonoBehaviour 或 ScriptableObject（二者均从 UnityEngine.Object 派生而来），在脚本的实例中也适用相同的序列化原则。

#### 自定义类不支持 null

思考在反序列化一个使用以下脚本的 MonoBehaviour 时进行了多少次分配。

出现一次以下分配并不奇怪：Test 对象的一次分配。另外，出现两次以下分配也不奇怪：Test 对象的一次分配和 Trouble 对象的一次分配。

但是，Unity 实际上进行超过一千次的分配。序列化程序不支持 null。如果序列化程序对某个对象进行序列化，并且一个字段为 null，则 Unity 将实例化该类型的新对象，并对该对象进行序列化。显然，这可能导致无限循环，因此存在七级的深度限制。达到该限制时，Unity 停止序列化具有自定义类、结构、列表或数组类型的字段。

由于 Unity 的子系统很多都是在序列化系统之上构建的，因此 Test MonoBehaviour 的这种异常大型的序列化流会导致所有这些子系统的执行速度低于必要速度。

#### 不支持多态

如果具有 public Animal[] animals 并输入一个 Dog、一个 Cat 和一个 Giraffe 的实例，则在序列化之后有三个 Animal 实例。

解决此限制的一种方法是认识到它仅适用于内联序列化的自定义类。对其他 UnityEngine.Objects 的引用被序列化为实际引用，对于这些情况，多态确实有效。可创建一个 ScriptableObject 派生类或另一个 MonoBehaviour 派生类，并引用该类。这样做的缺点是需要在某处存储该 Monobehaviour 或脚本化对象，并且无法有效地对其进行内联序列化。

设置这些限制的原因是序列化系统的核心基础之一是提前知道对象的数据流布局；它取决于类的字段类型，而不是存储在字段内的具体内容。

### 提示

#### 序列化的优化应用

您可以组织数据来确保从 Unity 的序列化获得最佳使用效果。

* 组织数据的目的是让 Unity 序列化尽可能小的数据集。这样做的主要目的不是为了节省计算机硬盘驱动器上的空间，而是为了确保您可以保持与项目以前版本的向后兼容性。如果使用大型的序列化数据集，那么在开发后期保持向后兼容性会变得更加困难。
* 组织数据时确保绝不会让 Unity 序列化重复的数据或缓存的数据。否则会导致向后兼容性出现严重问题：存在着很高的错误风险，因为数据太容易失去同步。
* 避免使用嵌套的递归结构引用其他类。序列化结构的布局总是必须相同；独立于数据，仅依赖于脚本中公开的内容。引用其他对象的唯一方法是通过 [UnityEngine.Object](file:///E:\\UnityDocumentation\\ScriptReference\\Object.html) 派生的类。这些类是完全独立的；它们只互相引用，没有嵌入内容。

#### 使 Editor 代码可热重载

重新加载脚本时，Unity 会在所有加载的脚本中序列化并存储所有变量。重新加载脚本后，Unity 会将它们恢复为序列化前的原始值。

重新加载脚本时，Unity 会恢复满足序列化要求的所有变量（包括私有变量），即使变量没有 [SerializeField](file:///E:\\UnityDocumentation\\ScriptReference\\SerializeField.html) 属性也是如此。在某些情况下，需要特意防止恢复私有变量：例如，如果您希望从脚本重新加载后引用为 null。在这种情况下，请使用 [NonSerializable](file:///E:\\UnityDocumentation\\ScriptReference\\NonSerializable.html) 属性。

Unity 绝不会恢复静态变量，因此对于重新加载脚本后需要保留的状态，不要使用静态变量。

# 内置序列化

Unity 的一些内置功能自动使用序列化技术。这些功能如下所述。

请参阅有关[脚本序列化](file:///E:\UnityDocumentation\Manual\script-Serialization.html)的文档以了解更多信息。

#### 保存和加载

Unity 使用序列化技术从计算机的硬盘驱动器加载和保存[场景](file:///E:\\UnityDocumentation\\Manual\\CreatingScenes.html)、[资源](file:///E:\UnityDocumentation\Manual\AssetWorkflow.html)和 [AssetBundle](file:///E:\\UnityDocumentation\\Manual\\AssetBundlesIntro.html)。这包括保存在您自己的脚本 API 对象（如 [MonoBehaviour](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.html) 组件和 [ScriptableObject](file:///E:\\UnityDocumentation\\Manual\\class-ScriptableObject.html)）中的数据。

这发生在 Editor 的 Play 模式和 Edit 模式中。

#### Inspector 窗口

在 [Inspector 窗口](file:///E:\UnityDocumentation\Manual\UsingTheInspector.html)中查看或更改游戏对象的组件字段的值时，Unity 会序列化此数据，然后在 Inspector 窗口中显示数据。Inspector 窗口在显示字段值时不与 Unity Scripting API 通信。如果在脚本中使用属性，则在 Inspector 窗口中查看或更改值时，绝不会调用任何属性 getter 和 setter，因为 Unity 会直接序列化 Inspector 窗口字段。

#### 在 Unity Editor 中重新加载脚本

更改并保存脚本时，Unity 会重新加载所有当前加载的脚本数据。它首先将所有可序列化变量存储在所有加载的脚本中，并在加载脚本后恢复它们。重新加载脚本后，所有不可序列化的数据都将丢失。

这会影响所有 Editor 窗口以及项目中的所有 MonoBehaviour。与 Unity 中的其他序列化情况不同，私有字段在重新加载时会默认序列化，即使它们没有“SerializeField”属性也是如此。

#### 预制件

在序列化背景下，[预制件](file:///E:\UnityDocumentation\Manual\Prefabs.html)是一个或多个[游戏对象](file:///E:\UnityDocumentation\Manual\GameObjects.html)和[组件](file:///E:\UnityDocumentation\Manual\Components.html)的序列化的数据。预制件实例包含对预制件源及其一系列修改的引用。这些修改是 Unity 为了创建特定预制件实例而需要对预制件源进行的操作。

只有在 Unity Editor 中编辑项目时，预制件实例才存在。在项目构建期间，Unity Editor 从两组序列化数据（预制件源和预制件实例的修改）进行游戏对象的实例化。

#### 实例化

对[场景](file:///E:\\UnityDocumentation\\Manual\\CreatingScenes.html)中存在的任何对象（例如[预制件](file:///E:\\UnityDocumentation\\Manual\\Prefabs.html)或[游戏对象](file:///E:\UnityDocumentation\Manual\GameObjects.html)）调用 [Instantiate](file:///E:\UnityDocumentation\ScriptReference\Object.Instantiate.html) 时，Unity 都会对其进行序列化。在运行时和在 Editor 中都是如此。从 [UnityEngine.Object](file:///E:\\UnityDocumentation\\ScriptReference\\Object.html) 派生的一切均可序列化。

Unity 随后创建一个新的游戏对象并将数据反序列化到新的游戏对象上。接下来，Unity 在另一个不同的变体中运行相同的序列化代码，以报告正在引用的其他 UnityEngine.Objects。它会检查所有引用的 UnityEngine.Objects 以查看它们是否是要实例化的数据的一部分。如果引用指向“外部”，例如纹理，则 Unity 会保持该引用不变。如果引用指向“内部”，例如子游戏对象，则 Unity 会修补对相应副本的引用。

#### 卸载未使用的资源

Resource.GarbageCollectSharedAssets() 是本机 Unity 垃圾回收器，与标准 C# 垃圾回收器执行不同的功能。该垃圾回收器在您加载场景之后运行，检查不再引用的对象（如纹理），并安全卸载它们。本机 Unity 垃圾回收器在一个变体中运行序列化程序，而在此变体中，对象会报告对外部 UnityEngine.Objects 的所有引用。通过此方式即可将一个场景使用过的纹理在下一个场景中卸载。

# 自定义序列化

序列化是将数据结构或对象状态转换为 Unity 可存储并在以后可重构的格式的自动过程。（请参阅有关[脚本序列化](file:///E:\UnityDocumentation\Manual\script-Serialization.html)的文档以了解关于 Unity 序列化的更多信息。）

有时可能希望序列化 Unity 的序列化程序不支持的内容。在许多情况下，最好的办法是使用序列化回调。（请参阅 Unity 的脚本 API 参考：[ISerializationCallbackReceiver](file:///E:\UnityDocumentation\ScriptReference\ISerializationCallbackReceiver.html)，了解关于使用序列化回调进行自定义序列化的更多信息。）

序列化回调可让您在序列化程序从字段中读取数据之前以及在完成对字段的写入之后收到通知。使用序列化回调可在运行时为难以序列化的数据赋予不同于实际序列化时的表示形式。

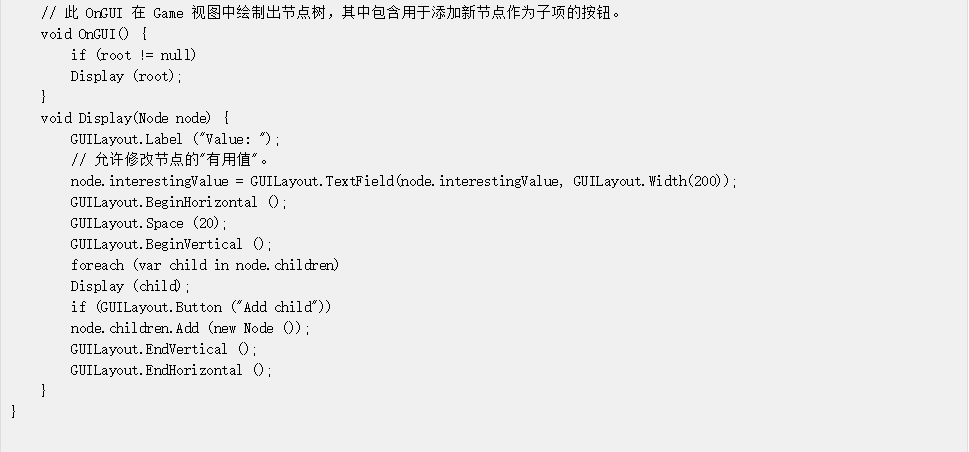
为实现此目的，请在 Unity 要序列化数据之前将数据转换为 Unity 理解的表示形式。然后，在 Unity 将数据写入字段之后，可将序列化的形式转换回在运行时需要的数据形式。

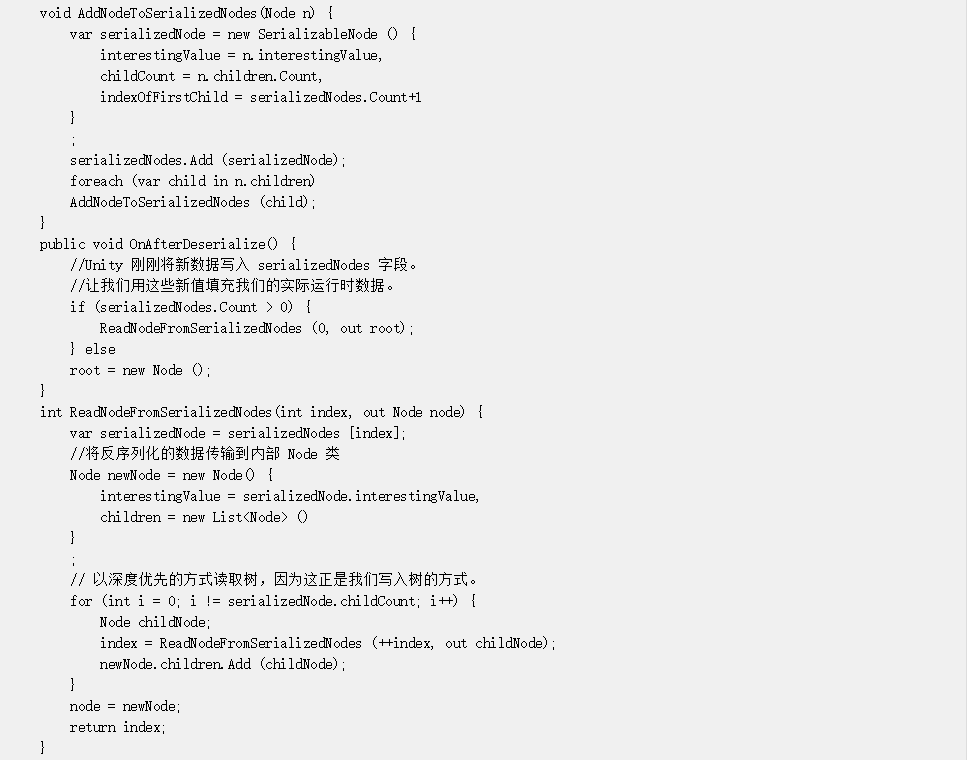
例如：您需要一个树数据结构。如果让 Unity 直接序列化该数据结构，“不支持 null”限制将导致数据流变得非常大，从而在许多系统中引起性能下降。下面的示例 1 中显示了这一情况。

相反，您告诉 Unity 不要直接序列化树，并创建单独的字段来以序列化的格式（适用于 Unity 序列化程序）存储树。下面的示例 2 中显示了这一情况。

**示例 2**：避免 Unity 直接序列化，并避免性能问题

L





# 脚本序列化错误

序列化是将数据结构或对象状态转换为 Unity 可存储并在以后可重构的格式的自动过程。（请参阅有关[脚本序列化](file:///E:\UnityDocumentation\Manual\script-Serialization.html)的文档以了解更多信息。）

在某些情况下，脚本序列化可能会导致错误。下面列出了其中一些错误的修复方法。

#### 从构造函数或字段初始化函数调用 Unity Scripting API

在 [MonoBehaviour](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.html) 构造函数或字段初始化函数中调用诸如 [GameObject.Find](file:///E:\\UnityDocumentation\\ScriptReference\\GameObject.Find.html) 之类的脚本 API 会触发错误：“Find is not allowed to be called from a MonoBehaviour constructor (or instance field initializer), call in in Awake or Start instead.”

解决此问题的方法是在 [MonoBehaviour.Start](file:///E:\\UnityDocumentation\\ScriptReference\\MonoBehaviour.Start.html) 中而不是在构造函数中调用脚本 API。

#### 在反序列化期间调用 Unity Scripting API

从标有 System.Serializable 的类的构造函数中调用诸如 [GameObject.Find](file:///E:\\UnityDocumentation\\ScriptReference\\GameObject.Find.html) 之类的脚本 API 会触发错误：“Find is not allowed to be called during serialization, call it from Awake or Start instead.”

要解决此问题，请编辑代码，确保不会在任何序列化对象的构造函数中调用任何脚本 API。

#### 线程安全的 Unity Scripting API

大多数脚本 API 都受上面列出的限制所影响。只有部分 Unity Scripting API 免受影响，可从任何位置接受调用。它们如下：

* [Debug.Log](file:///E:\UnityDocumentation\ScriptReference\Debug.Log.html)
* [Mathf](file:///E:\UnityDocumentation\ScriptReference\Mathf.html) 函数
* 简单的独立结构；例如，像 [Vector3](file:///E:\UnityDocumentation\ScriptReference\Vector3.html) 和 [Quaternion](file:///E:\UnityDocumentation\ScriptReference\Quaternion.html) 这样的数学结构

为了降低序列化过程中出错的风险，只能调用不需要在 Unity 本身中获取或设置数据的独立 API 方法。仅在没有其他选择的情况下才调用它们。