

4.6.2 压缩格式

压缩格式是AudioClip本身的压缩格式。



图4.18 AudioClip压缩格式

脉冲编码调制

它是未经压缩的，消耗大量的内存。除非你想要非常高质量的声音，否则不要设置这个。

自适应差分脉冲编码调制

它比PCM少用70%的内存，但质量较低，其特点是CPU负载比Vorbis低很多。这意味着它解压速度快，适合立即播放，也适合大音量播放的声音。对于嘈杂的声音，如脚步声、碰撞声、武器声等，尤其如此，这些声音需要快速、大量地回放。

Vorbis

作为一种有损压缩格式，质量比PCM低，但文件大小更小。只有质量可以设置，允许进行微调。所有声音（背景音乐、声音效果、声音）最常用的压缩格式。

▼ 表4.3 压缩方法和主要用途摘要

类型。	使用
脉冲编码 调制	未使用
自适应差 分脉冲编	音效

码调制	
Vorbis	背景音乐、声音效果、声音。

4.6.3 采样率规格

可以通过指定采样率来调整质量。所有的压缩格式都被支持，并且可以从采样率设置中选择三种不同的方法。

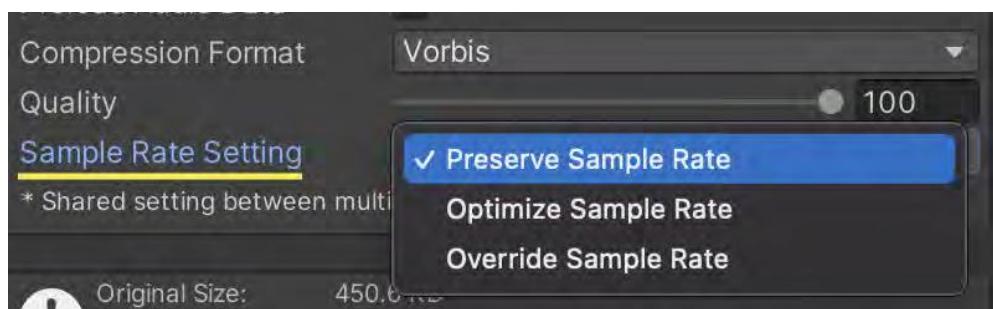


图4.19 采样率设置

保存采样率

默认设置。使用原始声源的采样率。

优化采样率

它在Unity一侧进行分析，并根据最高频率成分自动优化。

覆盖采样率

覆盖原始声源的采样率，可以指定8,000到192,000赫兹。如果采样率高于原始音源，质量就不会提高。当你想降低原始音源的采样率时使用。

4.6.4 对于声音效果，设置为单声道。

Unity默认播放的是立体声，但你可以通过启用Force To Mono来启用单声道播放。强

制执行单声道播放可以使文件大小和内存大小减半，因为你不必为每一面都准备数据。

4.7 资源 / 流媒体资产

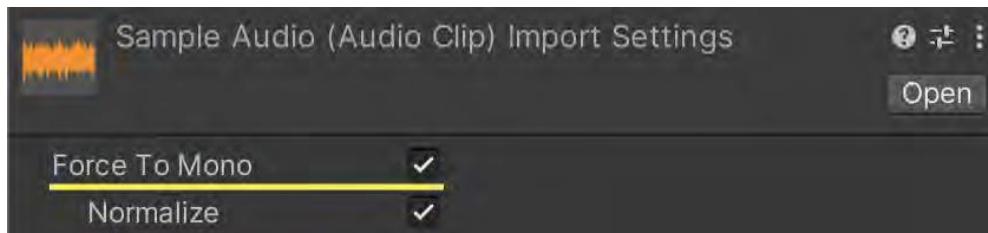


图4.20 AudioClip强制转单声道

单声道播放通常对声音效果很好。3D声音在单声道播放时也可能更好。经过仔细考虑，建议启用Force To Mono。性能调整的效果是山穷水尽疑无路柳暗花明。如果你对单声道播放没有问题，可以积极地使用强制转为单声道。

这与性能调整的话题无关，但未压缩的音频文件应该被导入Unity中。如果你导入压缩的音频文件，由于它们在Unity那边被解码和重新压缩，质量会有所下降。

4.7 资源 / 流媒体资产

在Unity中有一些特殊的文件夹。从性能的角度来看，以下三个方面尤其需要注意

- 资源文件夹
- StreamingAssets文件夹。

Unity通常在构建中包括从场景、材料、脚本等引用的对象。

▼清单4.7。脚本中引用的对象的例子

```
1: [SerializeField] GameObject sample; // 引用的对象被包含在构建中。
```

对于前面的特殊折页，规则是不同的。存储的文件被包含在构建中。这意味着在发布时不需要的文件也会包括在构建中，如果它们被存储和

导致建设规模的扩大。

问题是不能从程序中检查。你必须目视检查不需要的文件，这很耗费时间。小心地将文件添加到这些文件夹。

然而，随着项目的进展，存储文件的数量将不可避免地增加。这些文件中的一些可能与不再使用的不必要的文件混在一起。总而言之，我们建议定期审查存储的文件。

4.7.1 减慢启动时间 资源文件夹

在Resources文件夹中存储大量的对象会增加应用程序的启动时间；Resources文件夹是一个老式的便利功能，允许通过字符串引用加载对象。

▼ 清单4.8 脚本中引用的对象的例子

```
1: var object = Resources.Load("aa/bb/cc/obj");
```

你可以用这样的代码加载对象，这常常被过度使用，因为把它们存储在资源文件夹中，你就可以从脚本中访问它们。然而，如上所述，过度装载资源文件夹会增加应用程序的启动时间。其原因是，当Unity启动时，它分析了所有资源文件夹中的结构，并创建一个查找表。最好是尽可能地减少对资源文件夹的使用。

4.8 脚本对象

ScriptableObjects是YAML资产，许多项目都可能将其文件作为文本格式管理。你可以明确地指定[**PreferBinarySerialisation**]属性来改变保存格式为二进制。主要是针对有大量数据的资产，二进制格式提高了写和读的性能。

然而，二进制格式自然更难用合并工具处理。对于那些只需要被覆盖并且不需要检查文本变化差异的资产，或者对于那些在游戏开发完成后不再需要改变数据的资产，建议指定[**PreferBinarySerialisation**]。

4.8 脚本对象

在使用ScriptableObjects时，一个常见的错误是类的名称和源代码的文件名不匹配。类和文件必须有相同的名字。当创建一个类时，要注意命名，并确保.NET技术的使用。资产文件被正确序列化并以二进制格式保存。

▼ 清单4.9：一个ScriptableObject的示例实现。

```
1: /*
2: *当源代码文件名是ScriptableObjectSample.cs时
3: */
4:
5: // 成功实现序列化
6: [SerializeField]。
7: public sealed class ScriptableObjectSample : ScriptableObject 8:
{
9:     ...
10: }
11:
12: // 串行化失败
13: [SerializeField]。
14: public sealed class MyScriptableObject : ScriptableObject
15: {
16:     ...
17: }
```




统一

性能调整

CHAPTER

0

第五章。

调试实践--资产捆绑

AssetBundle配置的问题会导致一系列的问题，如浪费用户宝贵的通信和存储空间，以及阻碍舒适的游戏体验。本章介绍AssetBundle的配置和实施策略。

5.1 资产包的颗粒度

由于依赖性问题，需要仔细考虑AssetBundle的颗粒度应该有多细小。在极端的情况下，有两种方法：把所有的资产放在一个AssetBundle中，或者为每个资产设置一个AssetBundle。这两种方法都很简单，但前一种方法有一个关键问题。这是因为即使你只添加资产或更新一个资产，你也必须重建整个文件并分发。如果资产总量为GB，更新负荷可能非常高。

因此，应尽可能选择划分AssetBundle的方法，但如果划分得太细，会造成各方面的开销。因此，基本上建议根据以下策略来创建AssetBundle。

- 将同时使用的前提资产合并为一个AssetBundle。
- 被一个以上的资产引用的资产应该在一个单独的资产包中。

这很难完美地控制，但在项目中设置一些关于颗粒度的规则是个好主意。

5.2 资产包加载API

有三个API用于从AssetBundle加载资产。

AssetBundle.LoadFromFile

通过指定存储中存在的文件路径来加载。最快和更多。

也能节省内存，因此通常使用。

资产包从内存中加载

加载已经加载到内存中的指定的AssetBundle数据；在使用AssetBundle时，需要在内存中保持非常大的数据，内存负荷非常高。因此，通常不使用它。

AssetBundle.LoadFromStream

通过指定一个返回AssetBundle数据的流来加载。当加载一个加密的AssetBundle同时解密时，考虑到内存负载，使用这个API。然而，流必须是可寻的，所以必须注意不要使用不可寻的密码算法。

5.3 资产包的卸货策略。

`AssetBundle.Unload(bool unloadAllLoadedObjects)` 的参数
unloadAllLoadedObjects是用于在不再需要时卸载AssetBundle的API，这个参数非常重要，应该在开发初期设置。是非常重要的，应该在开发过程的早期就决定如何设置它。如果这个参数为真，当卸载一个资产包时，所有从该资产包加载的资产也将被卸载；如果为假，没有资产将被卸载。

换句话说，真正的情况是，当资产被使用时，AssetBundle也必须继续被加载，它的内存负载更高，但也更安全，因为它确保了资产可以被销毁。另一方面，如果是假的，内存负载就会很低，因为当资产加载完毕后，AssetBundle可以被卸载，但忘记卸载已使用的资产会导致内存泄漏或导致同一资产在内存中被多次加载。需要适当的内存管理。一般来说，严格的内存管理是严厉的，所以如果有足够的内存负载，建议使用`AssetBundle.Unload(true)`。

5.4 优化同时加载的资产包的数量

`Unload(true)`，资产包在使用中时不能被卸载。因此，根据资产包的颗粒度，可能会出现同时加载100多个资产包的情况。在这种情况下，重要的是要意识到文件描述符和持久性文件的限制。

第5章 调试实践 - 资产捆绑

Manager.Remapper的内存用量。

文件描述符是操作系统在读或写文件时分配的一个操作ID；读或写一个文件需要一个文件描述符，文件操作完成后，文件描述符被释放。一个进程可以拥有的这些文件描述符的数量是有上限的，所以不可能同时打开超过这个限制。“如果你看到错误“太多打开的文件”，这表明已经达到了这个限制。因此，同时加载AssetBundle的数量会受到这个限制的影响，Unity也会打开一些文件，所以你需要对这个限制保持一定的余量。这个限制因操作系统和版本而异，所以有必要事先调查目标平台的数值，但作为一个例子，在iOS和macOS上有限制为256的版本。即使达到了限制，也有可能根据操作系统^{*1}暂时提高限制，所以必要时考虑实施。

同时加载许多AssetBundles的第二个问题是，Unity的PersistentManager.Remapper的存在，简单地说，它是管理Unity内部对象和数据之间的映射关系的函数。问题在于，即使AssetBundle被释放，所使用的内存空间也没有被释放，而是被汇集起来。由于这种性质，内存被挤压的程度与并发负载的数量成正比，因此减少并发负载的数量非常重要。

从上面来看，在AssetBundle.Unload(true)策略下操作时，有必要同时加载目标是最多加载150-200个资产包，当使用AssetBundle.Unload(false)策略操作时，最多150个或更少。

^{*1} 在Linux/Unix环境下，可以在运行时使用setrlimit函数改变极限值。



统一

性能调整

CHAPTER

0

第六章。

调音实践--物理学

这一章介绍了物理学的优化。

这里的物理学是指PhysX的物理学操作，它不包括ECS中的Unity物理学。

另外，尽管本章主要关注三维物理学，但在二维物理学中也有许多值得参考的地方。

6.1 物理学开 - 关

在Unity标准中，物理引擎总是每一帧都进行物理计算，即使在场景中没有放置与物理有关的组件。因此，如果你的游戏中不需要物理学，你应该关闭物理学引擎。

物理引擎的处理是通过对Physics.autoSimulation设置一个值来打开/关闭的。

你可以切换物理学。例如，如果你只在游戏中使用物理，而在其他情况下不使用，你应该只在游戏中把这个值设置为真。

6.2 固定时间步长和固定更新频率的优化

MonoBehaviour中的 FixedUpdate与Update不同，它在一个固定的时间运行。物理引擎将在一帧内复制一个固定更新，相对于前一帧的经过时间。

通过多次调用它，游戏世界中的经过时间与物理引擎世界中的时间是一致的。因此，Fixed Timestep的值越小，**Fixed Update**被调用的次数就越多，这就会造成负载。

这个时间可以在项目设置中的固定时间步长中设置，如图6.1所示。这个值的单位是秒。默认情况下，指定0.02，即20毫秒。

6.2 固定时间步长和固定更新频率的优化

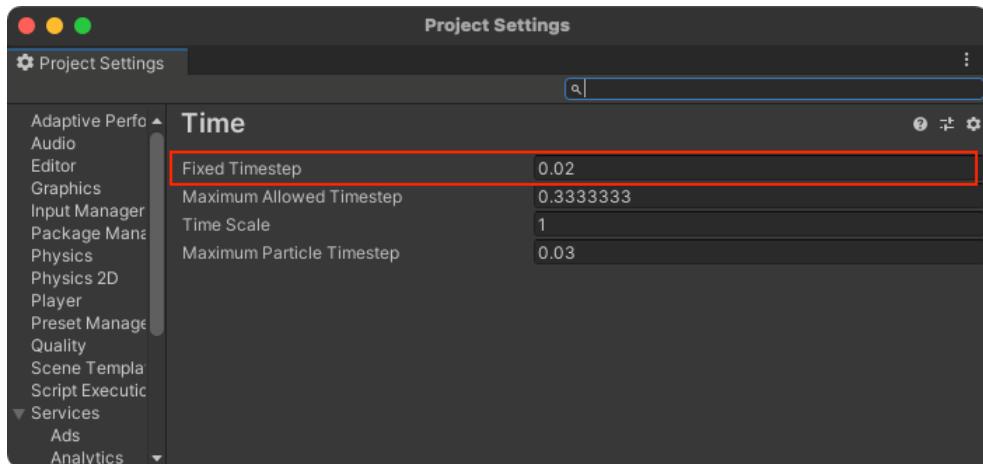


图6.1 项目设置中的固定时间段项目

它也可以在脚本中通过操作Time.fixedDeltaTime来改变。

一般来说，固定时间步长越小，物理计算就越精确，也就越不可能发生诸如错过碰撞的问题。因此，尽管在精确度和负载之间有一个权衡，但只要不导致游戏行为问题，建议将这个值设置为接近目标FPS。

6.2.1 允许的最大时间步长

正如上一节所提到的，固定更新是根据自上一帧以来所经过的时间多次调用的。

如果前一帧的耗时增加，例如，由于“一帧中的大量渲染处理”，固定更新将在该帧中被比平时更频繁地调用。

例如，如果固定时间步长为20毫秒，而前一帧用了200毫秒，则固定更新将被调用10次。

换句话说，如果某一帧超出了流程，那么下一帧的物理操作的成本就会更高。这增加了该帧也被放弃的风险，这反过来又使下一帧的物理操作更加沉重，这种现象在物理引擎的世界中被称为导致负螺旋。

为了解决这个问题，Unity允许用户选择“项目设置”页面，如图6.2所示。

最大允许时间步长, 这是物理学操作在一帧内使用的最大时间量。

第6章调谐练习--物理学

可以设置一个值。这个值默认设置为0.33秒，但你可能希望把它设置得更接近目标FPS，以限制固定更新调用的数量并稳定帧率。

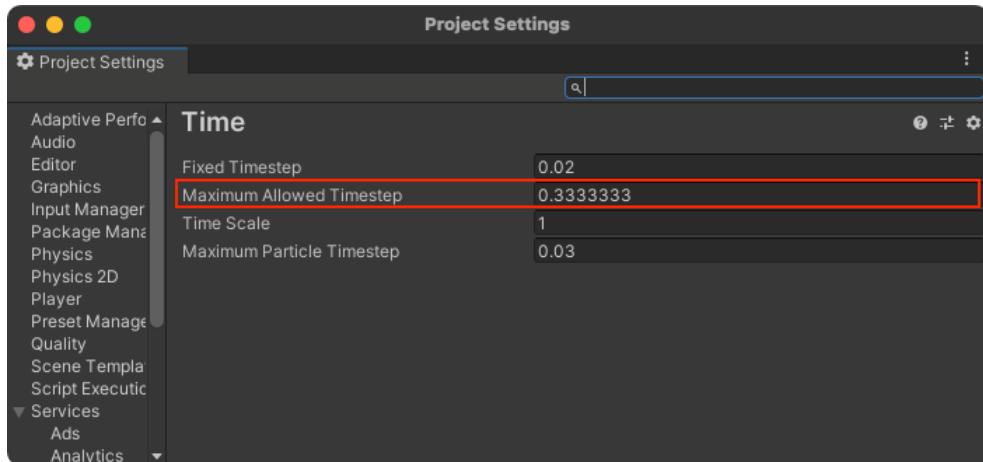


图6.2 项目设置中的最大允许时间步长项目

6.3 碰撞几何的选择。

一般来说，复杂的碰撞几何形状的撞击成本更高。因此，在优化物理学性能时，对撞机的几何形状尽可能地简单是很重要的。

例如，胶囊碰撞器经常被用来近似人形角色的形状，但如果高度不影响作为游戏的规范，用球体碰撞器代替它们，对打击的成本较低。

形状的选择很难一概而论，因为成本取决于击中的目标和情况，但值得记住的是，球体对撞机、胶囊对撞机、箱体对撞机和网格对撞机，决策成本依次递减。

6.4 碰撞矩阵和层的优化

物理学有一个叫做“碰撞矩阵”的设置，定义了哪些游戏对象层可以相互碰撞。这个

设置如图6.3所示，用于定义项目的碰撞矩阵。

6.5 射影优化

这可以在设置中的物理学>层碰撞矩阵中改变。

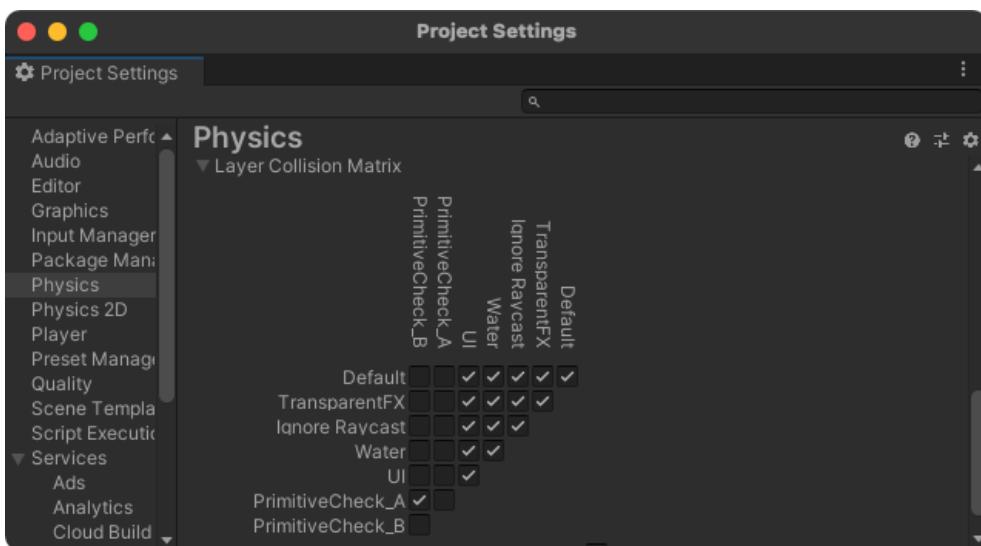


图6.3 项目设置中的图层碰撞矩阵项目

碰撞矩阵表明，如果在两个层的交叉点上的复选框被勾选，它们将发生碰撞。

适当地设置这一点是消除不需要碰撞的物体之间的计算的最有效方法，因为不相互碰撞的层也被排除在对物体进行粗略打击的预算之外，这被称为广义阶段。

出于性能方面的考虑，建议为物理操作设置专门的层，并取消不需要碰撞的层之间的所有复选框。

6.5 射影优化

Raycasting是一个有用的功能，它允许获得关于飞行的射线和碰撞的碰撞器的碰撞信息，但另一方面，它也可能是一个负载源。

6.5.1 铺设石膏的类型

除了Physics.Raycast与线段进行碰撞判定外，还有其他方法与其他形状进行判定，如Physics.SphereCast。

然而，作出决定的几何形状越复杂，负荷就越大。出于性能方面的考虑，最好是尽可能地只使用Physics.Raycast。

6.5.2 优化光线投射的参数。

Physics.Raycast除了有两个参数，即射线传输的起点和方向之外，还有maxDistance和layerMask作为与性能优化有关的参数。

maxDistance指定做出射线传输决定的最大长度，即射线的长度。

如果省略这个参数，Mathf.Infinity被作为默认值传递，它试图用很长的射线进行决策。不要指定超过必要的距离，因为这样的射线会对广义阶段产生负面影响，并可能击中首先不需要被击中的物体。

layerMask还能确保不需要打的层中的位不被立起来。
是的。

与碰撞矩阵一样，没有驻留位的层也被排除在广义阶段之外，从而降低了计算成本。如果省略了这个参数，默认值是Physics.DefaultRaycastLayers，它与所有的层相撞，除了忽略Raycast，所以还必须指定它。

6.5.3 RaycastAll和RaycastNonAlloc

Physics.Raycast返回其中一个碰撞器的碰撞信息，但多个碰撞信息可以用Physics.RaycastAll方法来检索。Physics.RaycastAll通过动态分配一个RaycastHit结构的数组来返回碰撞信息。因此，每次调用这个方法时，都会产生一个GC Alloc，而且GC不允许返回任何碰撞信息。
这可能会导致尖峰。

为了解决这个问题，有一个名为Physics.RaycastNonAlloc的方法，当把一个已分配的数组作为参数传递给它时，它把结果写入该数组并返回。

出于性能考虑，GC Alloc尽可能在 FixedUpdate内生成。
不应允许它这样做。

除了在数组初始化过程中，GC Alloc可以被避免，方法是将结果写入的数组保存在一个类域、池或其他机制中，并将数组传递给Physics.RaycastNonAlloc，如清单

6.1中所示。

6.6 对撞机和刚体

▼清单6.1。 Physics.RaycastAllNonAlloc的用法

```
1: // 飞行射线的起点
2: var origin = transform.origin;
3: // 射线飞行的方向。
4: var direction = Vector3.forward;
5: // Ray的长度
6: var maxDistance = 3.0f;
7: // 射线碰撞的图层
8: var layerMask = 1 << LayerMask.NameToLayer("Player");
9:
10: // 用来存储射线投射碰撞结果的数组
11: // 这个数组可以在初始化时预分配，也可以在初始化时预分配。
12: // 使用池中保留的东西
13: // 需要事先确定射线投射结果的最大数量
14: // private const int kMaxResultCount = 100;
15: // private readonly RaycastHit[] _results = new RaycastHit[kMaxResultCount];
16:
17: // 所有碰撞信息都以数组形式返回
18: // 碰撞数在返回值中返回，应与返回值一起使用。
19: var hitCount = Physics.RaycastNonAlloc(
20:     原产地。
21:     导演：
22:     导致的结果。
23:     层面具。
24:     询问
25: );
26: 如果(hitCount > 0)
27: {
28:     Debug.Log($"{hitCount}与玩家相撞")。
29:
30:     // _results包含碰撞信息，从数组31中的第0个开始。
31:     var firstHit = _results[0];
32:
33:     // 注意，如果指定的指数超过一定数量，就是无效的碰撞信息。
34: }
```

6.6 对撞机和刚体

Unity中的物理学有处理碰撞的Collider组件，如Sphere Collider和Mesh Collider，以及用于基于刚体物理学模拟的Rigidbody组件。根据这些组件的组合和它们的设置，它们被分为三种碰撞器。

一个连接了碰撞器组件而没有连接刚性体组件的对象被称为静态碰撞器。

优化是基于这样的假设，即该对撞机只用于始终保持在同一位置而不移动的几何体。

因此，在游戏过程中可以启用和禁用静态碰撞器，并且可以移动和缩放。

第6章调谐练习--物理学

不应进行。这样做将导致由于内部数据结构的变化而导致重新计算，这可能会大大降低性能。

对撞机和刚体组件都已连接。

被称为动态碰撞器的对象被称为动态碰撞器。

这个碰撞器可以通过物理引擎与其他物体发生碰撞。它还可以对碰撞和通过操纵脚本中的Rigidbody组件施加的力做出反应。

这使得它成为需要物理学的游戏中最常用的碰撞器。

对撞机和刚体组件都可以连接和使用。

Rigidbody的isKinematic属性被激活的组件被归类为Kinematic动态碰撞器。

运动学动态碰撞器可以通过直接操纵变换组件来移动，但不能像普通动态碰撞器那样通过操纵刚性体组件来施加碰撞或力。

当你想切换物理操作的执行时，或者当你想把这个碰撞器用于你想偶尔移动但不是大部分时间的障碍物，如门，这个碰撞器可以用来优化物理操作的执行。

6.6.1 刚体和睡眠状态

作为优化的一部分，如果一个刚体组件所连接的物体在一定时间内没有移动，物理引擎就会认为该物体处于休眠状态，并将其内部状态改为睡眠。只要物体不因外力或碰撞等事件而移动，移动到睡眠状态就能将物体的计算成本降到最低。

因此，Rigidbody组件不需要移动它所连接的任何物体。

通过尽可能地将不需要的项目过渡到睡眠状态，可以减少物理学操作的计算成本。

用于确定一个Rigidbody组件是否应该进入睡眠状态。

阈值可以通过项目设置中物理学里面的睡眠阈值来设置，如图6.4所示。另外，如果你想为单个对象指定阈值，你可以通过Rigidbody.sleepThreshold属性来设置它。

6.6 对撞机和刚体

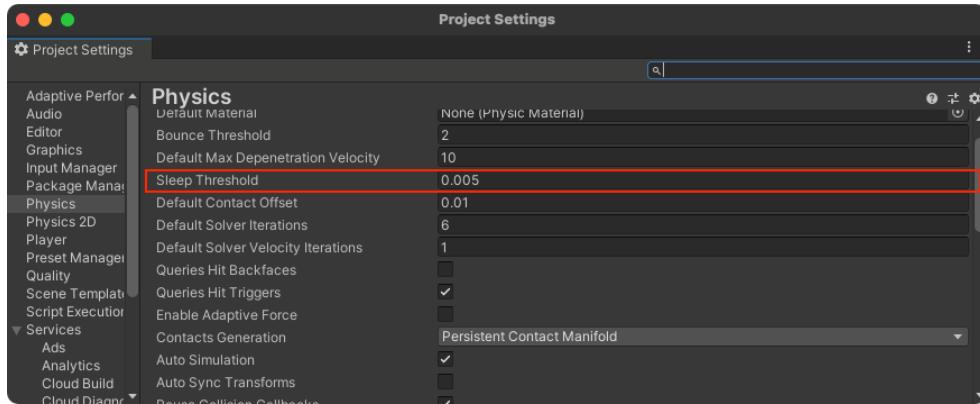


图6.4 项目设置中的睡眠阈值项目

睡眠阈值表示进入睡眠状态时按质量归一化的动能。

如果这个值增加，对象将更快地进入睡眠状态，从而保持低计算成本。然而，该物体可能会出现突然停止的情况，因为它即使在缓慢移动时也往往进入睡眠状态。如果这个值降低，上述现象就不容易发生，但另一方面，对象更难进入睡眠状态，这往往会使计算成本降低。

Rigidbody.IsSleeping属性可以检查Rigidbody是否正在睡觉。
场景中活动的Rigidbody组件的数量可以在Profiler的物理条目中找到。场景中活动的Rigidbody组件的总数可以在Profiler的Physics项目中找到，如图6.5所示。

第6章调谐练习--物理学

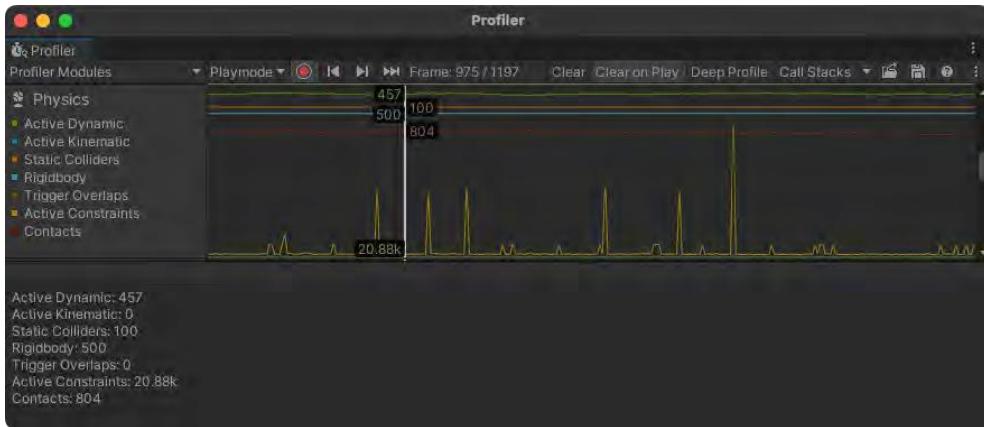


图6.5. 物理学条目在Profiler中。你可以看到活动的刚体的数量，以及物理引擎上每个元素的数量。

你也可以使用物理调试器来查看场景中哪些物体是活动的。

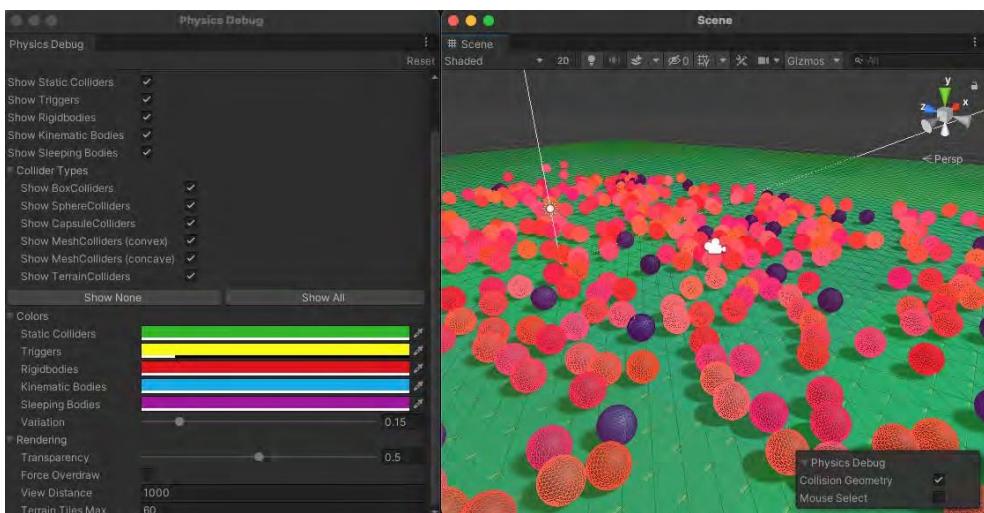


图6.6. 物理调试器，显示场景中的物体在物理引擎方面的状态，用颜色编码。

6.7 优化碰撞检测

刚体组件可以在“碰撞检测”项目中选择用于碰撞检测的算法。

从Unity 2020.3开始，有四种碰撞确定方式

- 离散性
- 连续的
- 连续动态
- 连续投机性

这些算法大致可分为两类：离散和连续碰撞测定。离散属于离散碰撞测定，其他属于连续碰撞测定。离散碰撞检测，顾名思义，在每次模拟中离散地传送物体，碰撞检测在所有物体被移动后进行。这可能导致错过碰撞，特别是当物体高速移动时，导致它们滑过。

有可能发生摩擦。

另一方面，连续碰撞确定法可以防止快速移动的物体滑过，因为它考虑到了物体移动前后的碰撞。这使得它比离散碰撞检测的计算成本更高。为了优化性能，创建游戏行为，以便尽可能地选择离散。

如果有任何不便，可以考虑连续碰撞的确定：连续只对动态和静态碰撞器组合有效，而连续动态也对动态碰撞器有效。连续动态的计算成本较高。

因此，如果角色绕场运行，即你只想考虑动态和静态碰撞器之间的碰撞检测，则选择连续动态，如果你也想考虑移动碰撞器之间的碰撞，则选择连续动态。

尽管动态碰撞器之间的连续碰撞确定是有效的，但连续投机性的计算成本比连续动态低，但应谨慎引入，因为它可能导致一种被称为**幽灵碰撞（Ghost Collision）**的现象，即多个碰撞器靠近并相互碰撞。然而，在引入这种方法时应谨慎行事，因为它可能导致一种被称为“幽灵碰撞”的现象，即多个碰撞器在很近的地方发生碰撞。

6.8 其他项目设置的优化

除了到目前为止介绍的设置外，这里还有一些项目设置中特别影响性能优化的项目。

6.8.1 Physics.autoSyncTransforms

在Unity 2018.3之前的版本中，每当调用与物理操作相关的API（如Physics.Raycast）时，Transform和物理引擎的位置会自动同步。

这个过程相对较重，在调用物理学API时可能会引起尖峰。

为了解决这个问题，从Unity 2018.3开始，Physics.autoSyncTransforms和该设置已被添加。如果这个值被设置为false，当调用物理学API时，将不执行上述的Transform同步过程。

变形同步发生在 FixedUpdate被调用之后，在物理学模拟过程中。这意味着，如果你移动一个碰撞器，然后在碰撞器的新位置上执行光线投射，光线投射将不会击中碰撞器。

6.8.2 Physics.reuseCollisionCallbacks

在Unity 2018.3之前的版本中，每次调用一个事件来接收 Collider组件的碰撞决定，例如OnCollisionEnter，都会创建并传递一个新的 Collision实例的参数，导致GC Alloc GC分配。

这种行为会对游戏性能产生负面影响，这取决于事件被调用的频率，所以从2018.3开始，一个名为Physics.reuseCollisionCallbacks的新属性已经暴露出来。

将此值设置为 "true "可以抑制GC Alloc，因为在事件调用过程中传递的 Collision实例被内部使用。

从2018.3开始，这个设置的默认值是true，如果你用相对较新的Unity版本创建你的项目，这很好，但如果你用早于2018.3的版本创建你的项目，这个值可能被设置为false。如果这个设置被禁用，你应该启用它，然后使用游戏应该修改代码，使其正确工作。



统一

性能调整

CHAPTER

0

第七章。

调音实践--图形

本章介绍了围绕Unity的图形功能的调整技术。

7.1 分辨率调整

在渲染管线中，碎片着色器的成本与渲染的分辨率成比例地增加。特别是在今天的移动设备的高显示分辨率下，渲染分辨率需要调整到一个合适的值。

7.1.1 DPI设置

将分辨率缩放模式设置为**固定DPI**，包含在移动平台的播放器设置中与分辨率相关的部分，可以将分辨率降低到针对一个特定的**DPI**（每英寸点数）。

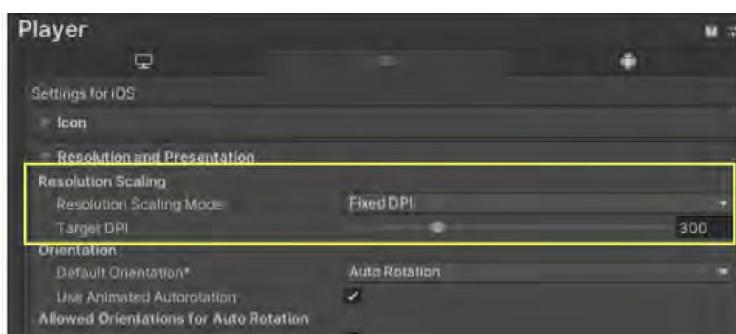


图7.1 分辨率缩放模式。

最终的绘图分辨率是由目标DPI值乘以质量设置中的分辨率缩放DPI比例因子值来决定

的。

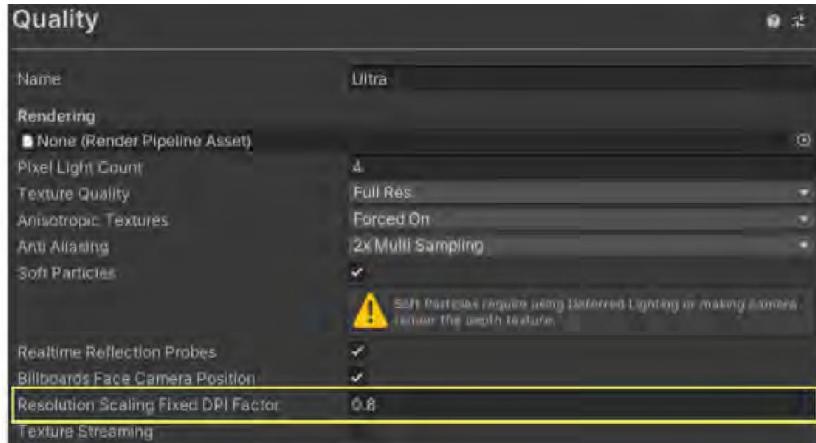


图7.2 分辨率比例 DPI比例系数

7.1.2 脚本化的分辨率设置

要从脚本中动态地改变绘图分辨率，请调用Screen.SetResolution。

当前的分辨率可以通过Screen.width或Screen.height获得，而DPI可以通过Screen.dpi获得。

▼列表7.1 屏幕.SetResolution

```
1: public void SetupResolution()
2: {
3:     var factor = 0.8f;
4:
5:     //用Screen.width, Screen.height获取当前分辨率
6:     var width = (int)(Screen.width * factor);
7:     var height = (int)(Screen.height * factor);
8:
9:     //设置分辨率。
10:    Screen.SetResolution(width, height, true);
11: }
```

Screen.SetResolution中的分辨率设置只反映在实际设备上。

请注意，变化不会反映在编辑器中。

7.2 半透明性和过度拉伸

半透明材料的使用是增加超量的一个主要原因。过度绘制是指在每个像素的屏幕上多次绘制一个片段，这对性能的影响与片段着色器的负载成正比。

特别是当大量的半透明粒子由粒子系统等生成时，往往会出现大量的过度拉伸。

可以通过以下方式减少因过度拉伸而导致的拉丝负荷增加

- 减少不必要的绘图区域
 - 尽可能减少纹理完全透明的区域数量，因为它们也会被渲染。
- 对于可能发生超绘的对象，尽可能使用轻量级着色器。
- 尽量少用半透明的材料。
 - 还可以考虑使用抖动技术，它可以用不透明的材料再现伪透明的外观。

在内置渲染管道的编辑器中，通过将场景视图模式改为超绘，可以将超绘可视化，这对于调整超绘的基础是非常有用的。

7.3 减少平局次数

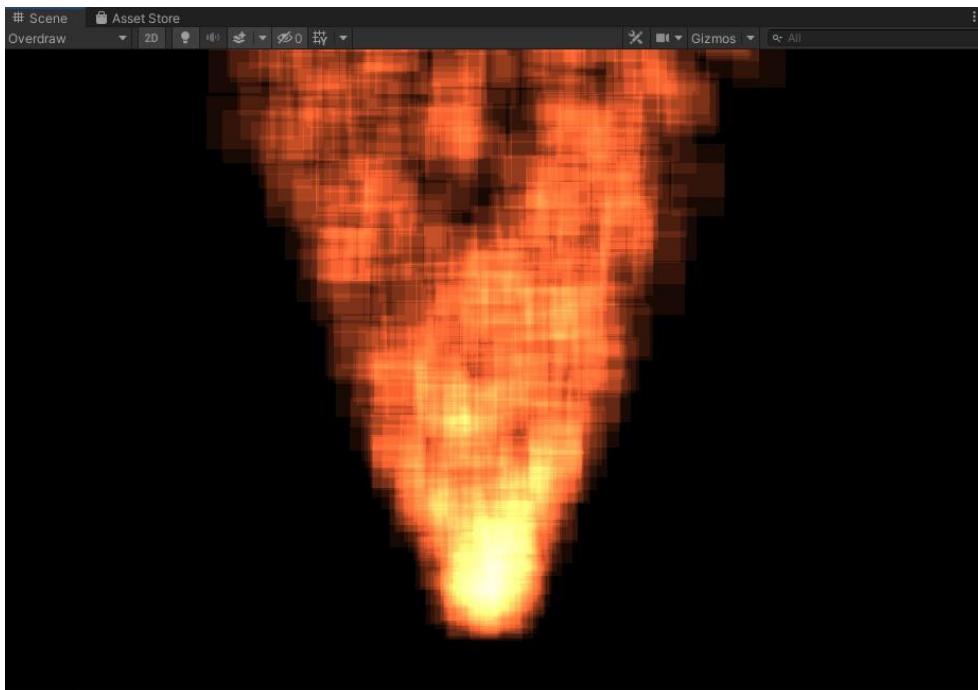


图7.3 超拔模式。

通用渲染管线可以通过场景调试视图模式将超绘可视化，该模式从Unity 2021.2开始实施。

7.3 减少平局次数

绘图调用的增加往往会影响到CPU的负载，但Unity中存在一些功能来减少绘图调用的数量。

7.3.1 动态配料

动态批处理是一个在运行时对动态对象进行批处理的功能。利用这一功能，使用相同

材料的动态对象

第7章 调试实践--图形

抽奖活动可以整合并减少。

要使用它，在播放器设置中启用动态批处理项目。另外，在Universal Render Pipeline中，你可以启用Universal Render Pipeline Asset，**Dy-中的Dynamic Batching**项目。

必须启用**Dynamic Batching**项目。只有通用渲染管道。

动态批处理的使用在下列情况下被废弃



▲ 图7.4 动态配料设置

由于动态批处理是一个耗费CPU成本的过程，在对一个对象应用之前，需要满足一些条件。主要条件列举如下。

- 指的是同一材料。
- MeshRenderer或Particle System用于绘图。
 - 其他组件，如SkinnedMeshRenderer，不受动态批处理的影响。
- 网格中的顶点数量少于300。
- 没有使用多路径。
- 不受实时阴影的影响。

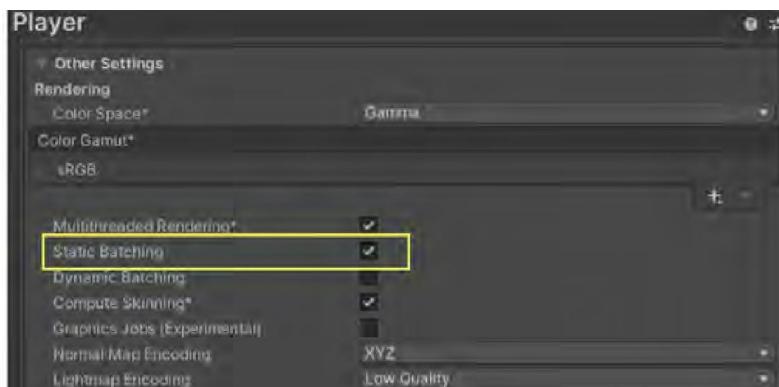
动态批处理可能不被推荐，因为它影响到CPU的稳态负载。下面介绍的**SRP**

Batcher, 可以用来实现类似于动态配料的效果。

7.3.2 静态配料

静态批处理是一个用于批处理在场景中不移动的对象的功能。这个功能可以用来减少对使用相同材质的静态对象的绘制调用。

与动态批处理一样，它可以通过从播放器设置中启用静态批处理来使用。



▲ 图7.5 静态配料设置。

为了使一个对象有资格进行静态批处理，必须启用该对象的静态标志。具体来说，静态标志中的子标志**Batching Static**必须被启用。

第7章 调试实践--图形

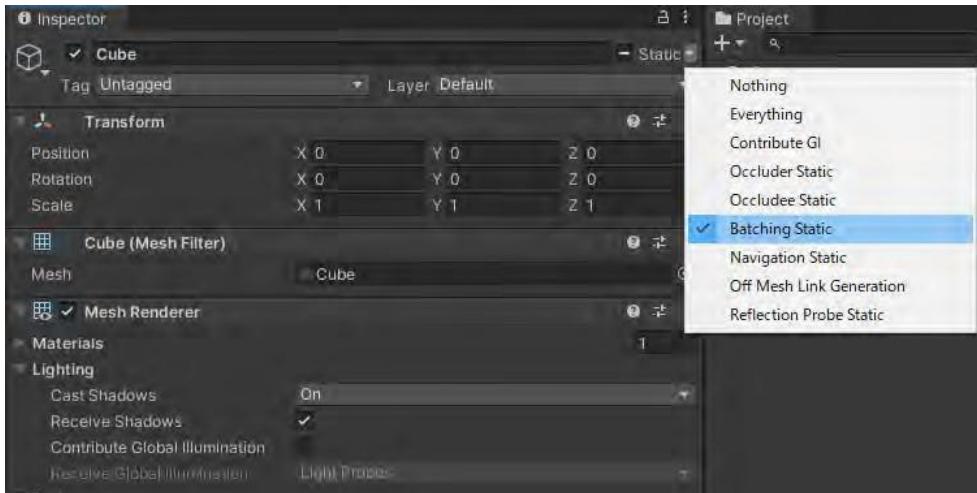


图7.6 分批静止。

与动态批处理不同，静态批处理不涉及运行时的顶点转换处理，因此可以在低负载的情况下进行。然而，应该注意的是，批量处理会消耗大量的内存来存储组合的网格信息。

7.3.3 GPU实例化。

GPU实例化是一个高效绘制相同网格和材料的对象的功能。它有望在多次绘制同一网格时减少绘制调用，如草或树。

要使用GPU实例化，请在材料的检查器中启用启用实例化。

7.3 减少平局次数



图7.7 启用实例。

创建可以使用GPU实例化的着色器需要一些专门的支持。下面是一个示例的着色器代码，它是在内置渲染管道中使用GPU实例化的最小实现。

▼ 清单7.2 支持GPU实例化的着色器。

```
1 : 着色器 "SimpleInstancing"
2: { 属性。
3: {
4:     _Colour ("Colour", Color) = (1, 1, 1, 1)
5: }
6:
7: CGINCLUDE
8:
9:
10: #include "UnityCG.cginc"
11:
12: struct appdata
13: {
14:     float4 vertex : POSITION;
15:     UNITY_VERTEX_INPUT_INSTANCE_ID
16: };
17:
18: 结构 v2f
19: {
20:     float4 vertex : SV_POSITION;
21:     // 只有当你想在片段着色器中访问INSTANCED_PROP时才需要。
22:     unity_vertex_input_instance_id
23: };
24:
25: UNITY_INSTANCING_BUFFER_START(Props)
26:     UNITY_DEFINE_INSTANCED_PROP(float4,
27:     _Colour) UNITY_INSTANCING_BUFFER_END(Props)
28:
29: v2f vert(appdata v)
```

第7章 调试实践--图形

```
30:    {
31:        v2f o;
32:
33:        UNITY_SETUP_INSTANCE_ID(v);
34:
35:        // 只有当你想在片段着色器中访问INSTANCED_PROP时才需要。
36:        UNITY_TRANSFER_INSTANCE_ID(v, o);
37:
38:        o.vertex = UnityObjectToClipPos(v.vertex);
39:        返回o。
40:    }
41:
42:    fixed4 frag(v2f i) : SV_Target
43:    {
44:        // 只有当你想在片段着色器中访问INSTANCED_PROP时才需要。
45:        unity_setup_instance_id(i);
46:
47:        float4 color = UNITY_ACCESS_INSTANCED_PROP(Props, _Colour);
48:        返回颜色。
49:    }
50:
51: ENDCG
52:
53: 子着色器
54: {
55:     标签 { "RenderType"="Opaque" }
56:     LOD 100
57:
58:     通过
59:     {
60:         CGPROGRAM
61:         #pragma vertex vert
62:         #pragma fragment frag
63:         #pragma multi_compile_instancing
64:         ENDCG
65:     }
66: }
67: }
```

GPU实例化只影响引用相同材质的对象，但你可以为每个实例设置属性。目标属性可以使用UNITY_INSTANCING_BUFFER_START(Props)和UNITY_INSTANCING_BUFFER_END(Props) 将其作为一个属性来单独改变。
你可以确定。

这个属性可以在C#中使用**MaterialPropertyBlock** API修改，以设置个别颜色和其他属性。只是要注意不要将 MaterialPropertyBlock 用于太多实例，因为访问 MaterialPropertyBlock 可能会影响CPU性能。

7.3.4 SRP Batcher。

SRP Batcher是一个减少绘图的CPU成本的功能，只在可脚本的渲染管道中可用。这个功能允许使用同一着色器变体的多个着色器设置通过调用被一起处理。

要使用SRP Batcher，请在**Scriptable Render Pipeline**资产的检查器中启用**SRP Batcher**项目。



图7.8 SRP Batcher的激活。

SRP Batcher也可以通过以下C#代码在运行时启用或禁用。

▼ 清单 7.3 启用 SRP Batcher

```
1: GraphicsSettings.useScriptableRenderPipelineBatching = true.
```

必须满足以下两个条件才能使着色器与SRP Batcher兼容。

1. 在一个名为**UnityPerDraw**的单一CBUFFER中定义每个对象的内置属性
2. 在一个名为**UnityPerMaterial**的单一CBUFFER中定义每个材料的属性

至于**UnityPerDraw**，Universal Render Pipeline等着色器基本上默认支持它，但你需要自己为**UnityPerMaterial**设置CBUFFER。

第7章 调试实践--图形

CBUFFER_START(UnityPerMaterial)每种材料的属性如下
)和CBUFFER_END。

▼列表7.4 统一的材料

```
1 : 属性。  
2: {  
3:     _Colour1 ("Colour 1", Color) = (1,1,1,1)  
4:     _Colour2 ("Color 2", Color) = (1,1,1,1)  
5: }  
6:  
7: CBUFFER_START(UnityPerMaterial)  
8:  
9: float4 _Colour1;  
10: float4 _Colour2;  
11:  
12: cbuffer_end
```

上述措施允许你创建支持SRP Batcher的着色器，但你也可以从检查器中检查着色器是否与SRP Batcher兼容。如果着色器检查器中的**SRP Batcher**项目是兼容的，那么它就支持SRP Batcher，如果它不兼容，那么它就不支持SRP Batcher。
以下是调查的结果摘要。



图7.9 支持SRP Batcher的着色器。

7.4 SpriteAtlas。

2D游戏和用户界面经常使用许多精灵来构建屏幕。**SpriteAtlas**是一个防止在这种情况下发生大量的绘制调用的功能。

SpriteAtlas可用于通过将多个精灵合并到一个纹理中来减少绘制调用。

要创建SpriteAtlas，首先必须从软件包管理器将**2D Sprite**安装到项目中。

第7章 调试实践--图形

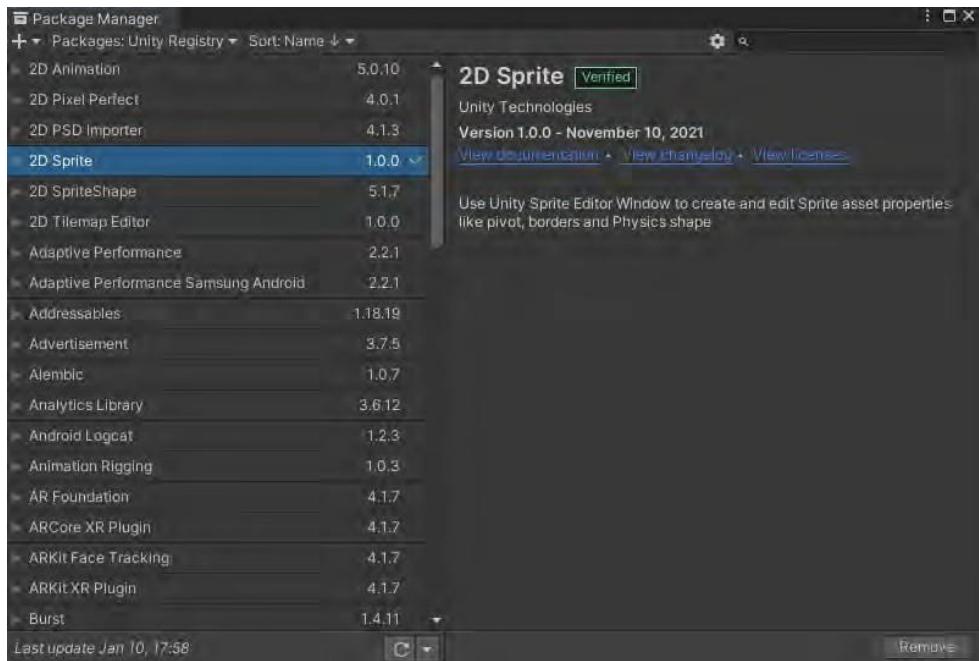
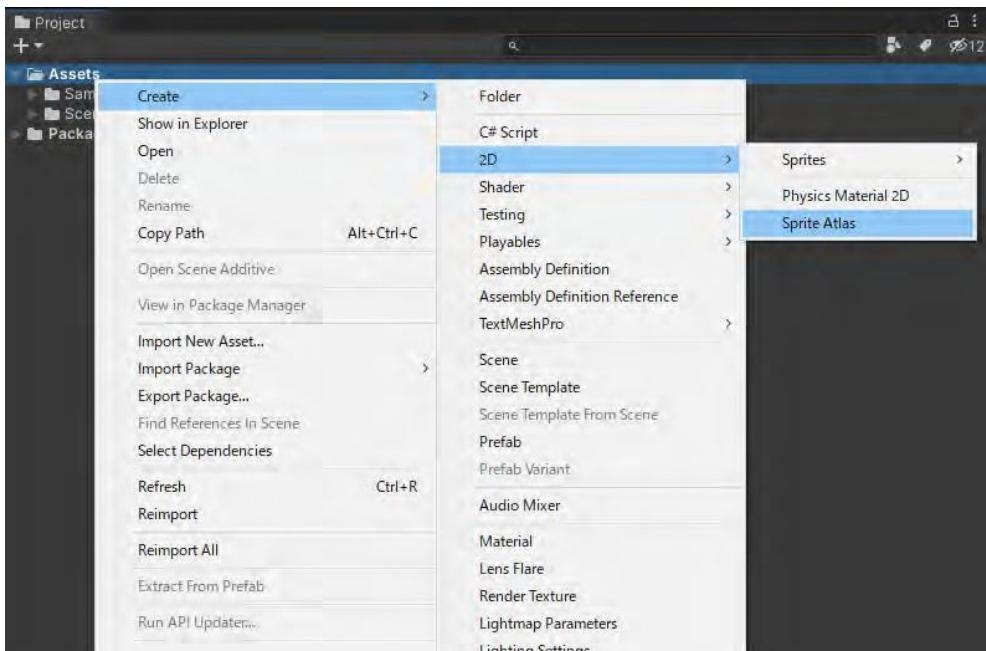


图7.10 2D Sprite。

安装后，在项目视图中右击，选择“创建->2D->Sprite Atlas”来创建SpriteAtlas资产。

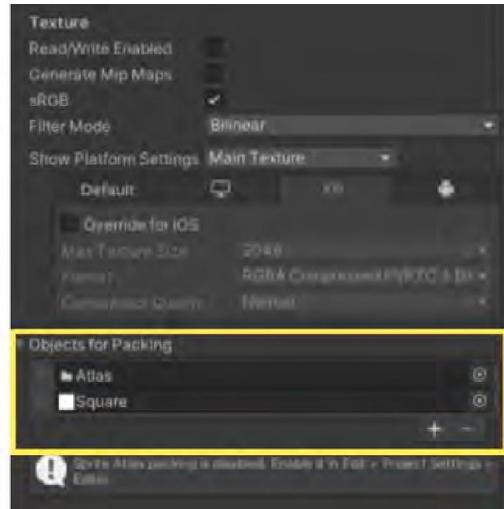
7.4 SpriteAtlas。



▲ 图7.11 创建SpriteAtlas

要指定一个精灵要被绘制成图谱，将该精灵或包含该精灵的文件夹设置为SpriteAtlas检查器中的“打包对象”项目。

第7章 调试实践--图形



▲ 图7.12 为包装设置对象

上述设置将导致在构建和Unity Editor回放过程中进行贴图处理，当目标精灵被绘制时，将引用与SpriteAtlas集成的纹理。

也可以通过以下代码直接从SpriteAtlas获取精灵。

▼ 清单7.5从SpriteAtlas加载雪碧

```
1 : [SerializeField]。
2: private SpriteAtlas atlas;
3:
4: public Sprite LoadSprite(string spriteName) 5:
{
6:     //从SpriteAtlas中获取Sprite，以Sprite名称作为参数 7: var
    sprite = atlas.GetSprite(spriteName);
8:     返回精灵。
9: }
```

在SpriteAtlas中加载一个Sprite要比只加载一个消耗更多的内存，因为要加载整个Atlas的纹理。因此，应该谨慎使用SpriteAtlas，例如适当地划分它。

本节是针对SpriteAtlas V1编写的；SpriteAtlas V2在操作上可能会有重大变化，比如不能指定要绘制地图的精灵的文件夹。

7.5 剔除

在Unity中，一个被称为“剔除”的过程被用来提前消除那些最终不会在屏幕上显示的部分的处理。

7.5.1 光锥剔除法

视锥剔除是一个用于从绘图中省略摄像机绘图范围以外的物体的过程，即视锥。这保证了相机范围之外的物体不会被计算出来进行绘制。

默认情况下，光锥剔除是在没有任何设置的情况下进行的。对于顶点着色器负载较高的物体，适当地分割网格也是很有用的，可以使其受到删减，减少绘制成本。

7.5.2 背后的杀戮

后方剔除是指从绘图中省略（应该是）摄像机看不到的多边形的背面的过程。大多数网格是封闭的（只有前面的多边形对摄像机可见），所以不需要绘制背面。

在Unity中，如果没有在着色器中指定，多边形的背面就会被剔除，但可以通过在SubShader中指定来切换剔除设置，如下所示。

▼ 清单7.6 设置剔除功能

```

1 : 子着色器
。
2: {    标签 { "RenderType"="Opaque" }
4:     LOD 100
5:
6:     后面的Cull / 前面的,
7:     关闭
8:     通过

```

```
9:      {      CGPROGRAM
10:     #pragma vertex vert
11:     #pragma fragment frag
12:     ENDCG
13:
14: }
15: }
```

有三种设置--背面、正面和关闭--每种设置都有以下效果。

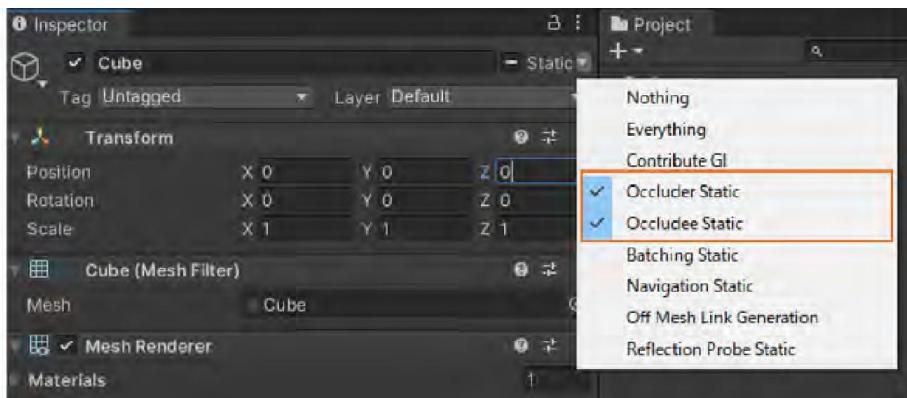
- 背面--不要画与视点相对的多边形
- 正面 - 不要在与视点相同的方向上绘制多边形。
- 关 - 禁用背面剔除功能，并绘制所有的表面

7.5.3 咬合剔除

遮挡剔除是一个将被物体遮挡住的、相机无法看到的物体从渲染目标中省略的过程。

这个函数在运行时根据预先烘焙的闭塞检测数据判断一个物体是否被闭塞，并将闭塞的物体从渲染目标中移除。

要使一个物体受到遮挡剔除的影响，可以从检查器的静态标志中启用 **Occluder Static** 或 **Occludee Static**；如果 **Occluder Static** 被禁用，**Occludee Static** 被启用，那么物体不再被认为是要被隐身的，而只是被认为是要被隐身的。在相反的情况下，该物体将不再被识别为隐身侧，而只被处理为隐身侧。



▲图7.13 遮盖物剔除的静态标志

要为闭塞剔除进行预烘烤，会显示闭塞剔除窗口。在这个窗口中，可以改变每个对象的静态标志，可以改变烘烤设置等，按烘烤按钮就可以进行烘烤。

第7章 调试实践--图形

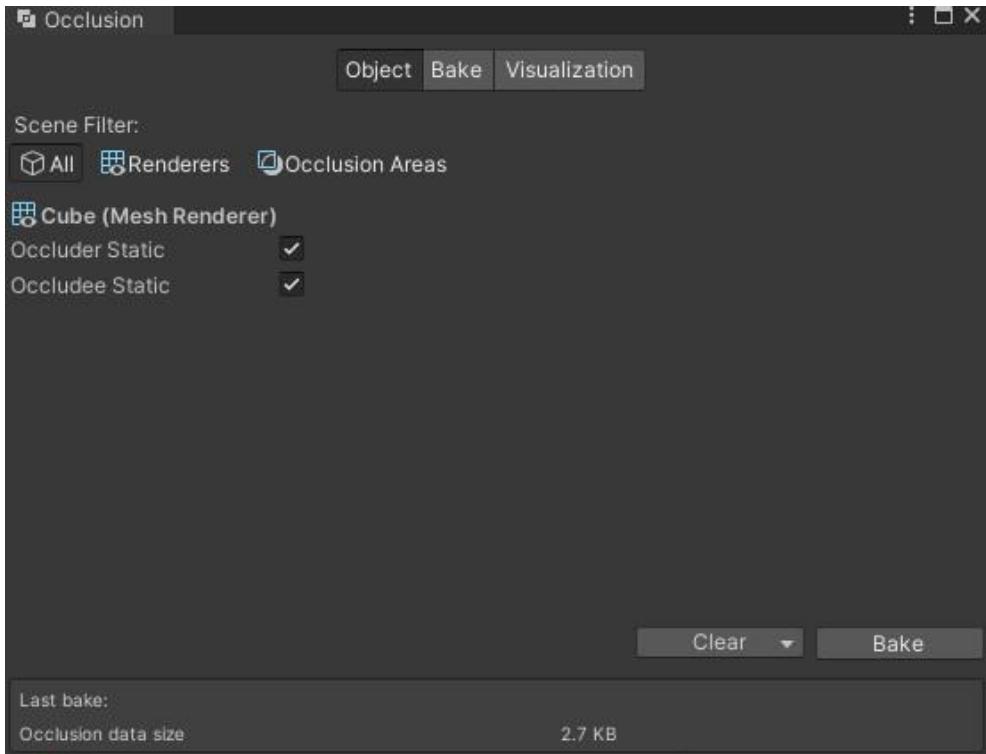


图7.14 遮挡剔除窗口

遮挡剔除减少了绘图成本，而不是剔除过程的CPU。

负载是施加在系统上的，所以有必要平衡每个负载并进行适当的设置。

只有物体绘制过程因闭塞剔除而减少，而实时阴影绘制等过程保持不变。

7.6 着色器

着色器对图形表示非常有效，但往往会导致性能问题。

7.6.1 降低浮点类型的精度

GPU（特别是在移动平台上）对较小的数据类型的计算比对较大的数据类型的计算要快。因此，在可能的情况下，将浮点数类型从**float**类型（32位）替换为**half**类型（16位）是非常有用的。

flat类型应该在需要精度的时候使用，例如在深度计算中，但是在颜色计算中，即使降低精度也很难在结果的外观上造成很大的差异。

7.6.2 用顶点着色器进行计算。

顶点着色器的处理是针对网格中的顶点数量进行的，而片段着色器的处理是针对最终被写入的像素数量进行的。一般来说，顶点着色器的执行频率往往低于片段着色器，所以复杂的计算应该尽可能由顶点着色器执行。

顶点着色器的计算结果通过着色器语义传递给片段着色器，但需要注意的是，这里传递的数值是插值的，看起来可能与片段着色器计算时不同。

▼清单7.7。 用顶点着色器进行预算算

```

1 : CGPROGRAM。
2: #pragma vertex vert 3:
# pragma fragment frag 4:
5: #include "UnityCG.cginc".
6:
7: 结构appdata
8: {
9:     float4 vertex : POSITION;
10:    float2 uv : TEXCOORD0;
11: }
12:
13: 结构 v2f
14: {
15:     float2 uv : TEXCOORD0;
16:     float3 factor : TEXCOORD1;
17:     float4 vertex : SV_POSITION;
18: }
19:
20: sampler2D _MainTex;
21: float4 _MainTex_ST;
22:
23: v2f vert (appdata v)
24: {
25:     v2f o;
26:     o.vertex = UnityObjectToClipPos(v.vertex);

```

第7章 调试实践--图形

```
27:     o.uv = TRANSFORM_TEX(v.uv, _MainTex);
28:
29:     // 进行复杂的预计算。
30:     o.因子=计算因子();
31:
32:     返回o。
33: }
34:
35: fixed4 frag (v2f i) : SV_Target
36: {
37:     fixed4 col = tex2D(_MainTex, i.uv);
38:
39:     //在片段着色器中使用顶点着色器计算的值
40:     col *= i.因素。
41:
42:     返回col。
43: }
44: ENDCG
```

7.6.3 预先填充纹理的信息。

如果着色器内复杂的计算结果不被外部数值所改变，那么将预先计算的结果存储为纹理元素可能会很有用。

一种方法是在Unity中实现一个生成专用纹理的工具，或者作为各种DCC工具的扩展。如果已经在使用的纹理的alpha通道没有被使用，那么最好将其写入或准备一个专用纹理。

例如，用于调色的**LUT**（颜色对应表）对纹理进行预着色校正，其中每个像素的坐标对应于每种颜色。通过在着色器中根据原始颜色对纹理进行采样，结果几乎与预先应用颜色校正的原始颜色完全相同。



图7.15 颜色校正前的LUT纹理（1024x32）。

7.6.4 ShaderVariantCollection

ShaderVariantCollection可用于在使用着色器之前对其进行编译，并防止出现峰值。

ShaderVariantCollection提供了一个游戏中使用的着色器变体的列表。

7.6 着色器

可以作为一个集合持有，通过在项目视图中选择“创建->着色器->着色器变体集合”来创建。

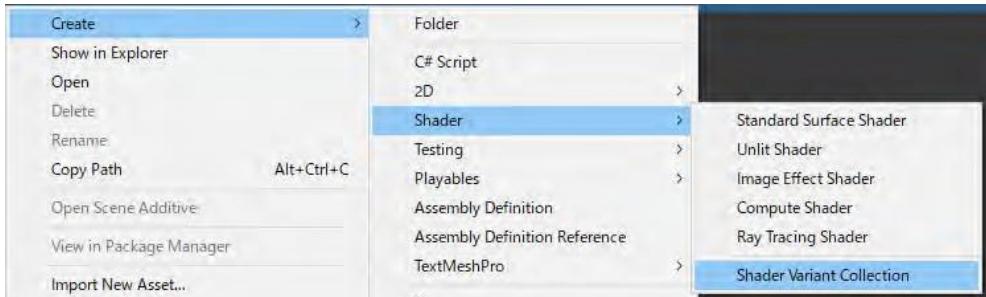


图7.16 创建一个ShaderVariantCollection。

在创建的ShaderVariantCollection的Inspector视图中，按Add Shader来添加目标着色器，然后选择要为该着色器添加的变量。

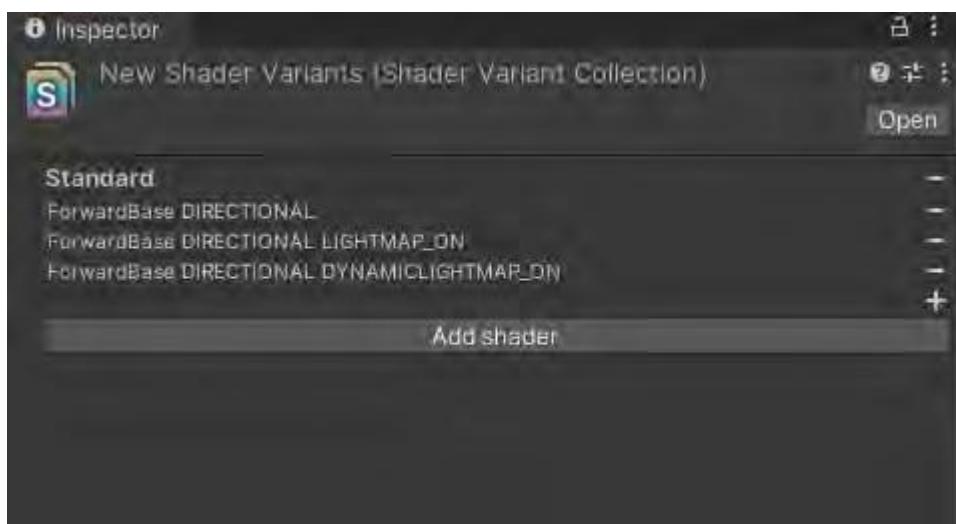


图7.17 ShaderVariantCollection的检查器。

ShaderVariantCollection在图形设置的**Shader**预加载部分。

添加到预装着色器中，在应用程序启动时进行编译

第7章 调试实践--图形

可以设置着色器的变体。

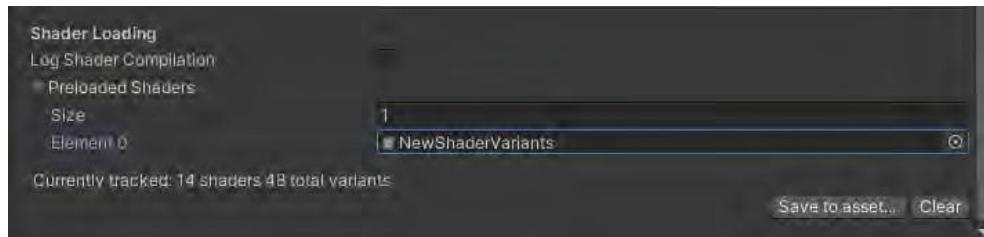


图7.18 预装的着色器。

也可以通过从脚本中调用 **ShaderVariantCollection.WarmUp()**，在相关的 **ShaderVariantCollection** 中明确地预编译着色器变体。

▼列表7.8 **ShaderVariantCollection.WarmUp**

```
1: public void PreloadShaderVariants(ShaderVariantCollection collection)
2: {
3:     // 明确地预编译着色器的变体
4:     如果(!collection.isWarmedUp)
5:     {
6:         collection.WarmUp();
7:     }
8: }
```

7.7 写作

照明是游戏中艺术表现的最重要元素之一，但它往往对性能有很大影响。

7.7.1 实时阴影

实时阴影的生成消耗了大量的绘图调用和填充率。因此，在使用实时阴影时，应仔细考虑设置。

减少平局次数

为了减少影子产生的抽水电话，可以考虑以下政策

- 减少掉落阴影的物体的数量。
- 分批将抽奖活动集中在一起。

有几种方法可以减少掉落阴影的物体的数量，但一个简单的方法是关闭MeshRenderer中的**Cast Shadows**设置。这将从阴影渲染中删除该对象。这个设置在Unity中通常是打开的，在使用阴影的项目中应该注意。

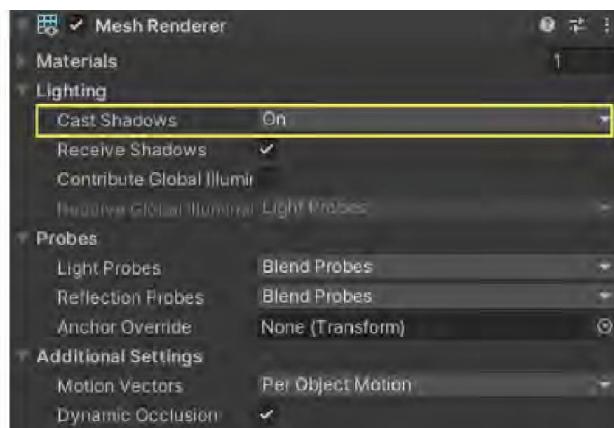


图7.19 投射阴影。

减少物体在阴影贴图中的最大绘制距离也是很有用的：质量设置中的阴影距离项目允许你改变阴影贴图的最大绘制距离，这个设置将阴影下降的物体数量减少到必要的最小值。这个设置可以让你把落下阴影的对象的数量减少到必要的最低限度。调整这个设置也会降低阴影的分辨率，因为阴影被绘制到阴影贴图的分辨率所能达到的最小范围。

第7章 调试实践--图形

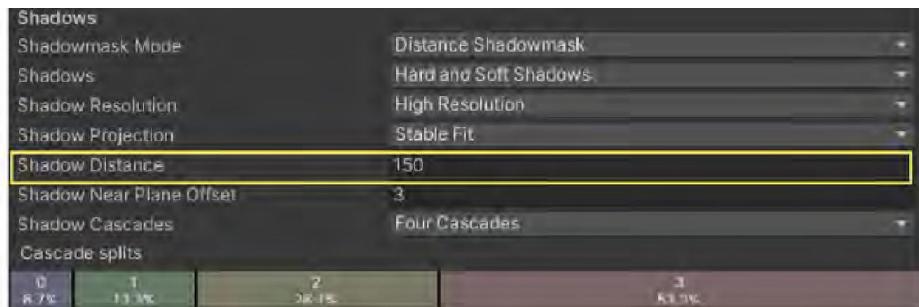


图7.20 阴影距离。

与普通绘图一样，阴影绘图可以采用批处理的方式来减少绘图调用。关于批处理技术的更多信息，见7.3 减少绘图调用。

节省的填充率

阴影引起的填充率取决于阴影图的绘制和受阴影影响的物体的绘制。

你可以通过调整质量设置中的阴影部分的一些设置来节省相应的填充率。



▲ 图 7.21 质量设置 -> 阴影

在阴影部分，你可以改变阴影格式，硬阴影产生清晰的阴影边界，但负载相对较低，而软阴影产生模糊的阴影边界，但负载较高。

阴影分辨率和阴影级联字段用于指定阴影图的阴影分辨率。

这是一个引起共鸣的项目，更大的设置会增加阴影贴图的分辨率，消耗更多的填充率。然而，这一设置也与阴影的质量有很大关系，所以应该仔细调整，在性能和质量之间取得平衡。

有些设置可以从灯光组件的检查器中改变，这样就可以为单个灯光改变设置。

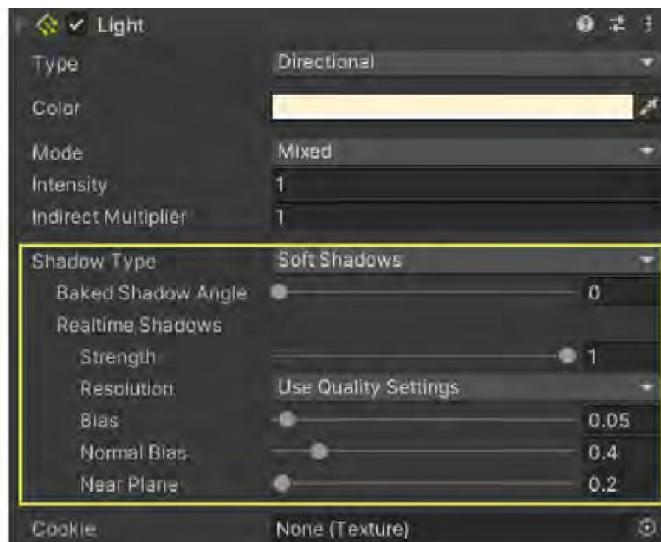


图7.22 光线组件的阴影设置。

伪影子

根据游戏类型和艺术风格的不同，使用板块多边形等技术来模拟物体的阴影也可能是有用的。这种方法有很强的使用限制，灵活性不高，但比起通常的实时阴影渲染方法，它要轻得多。

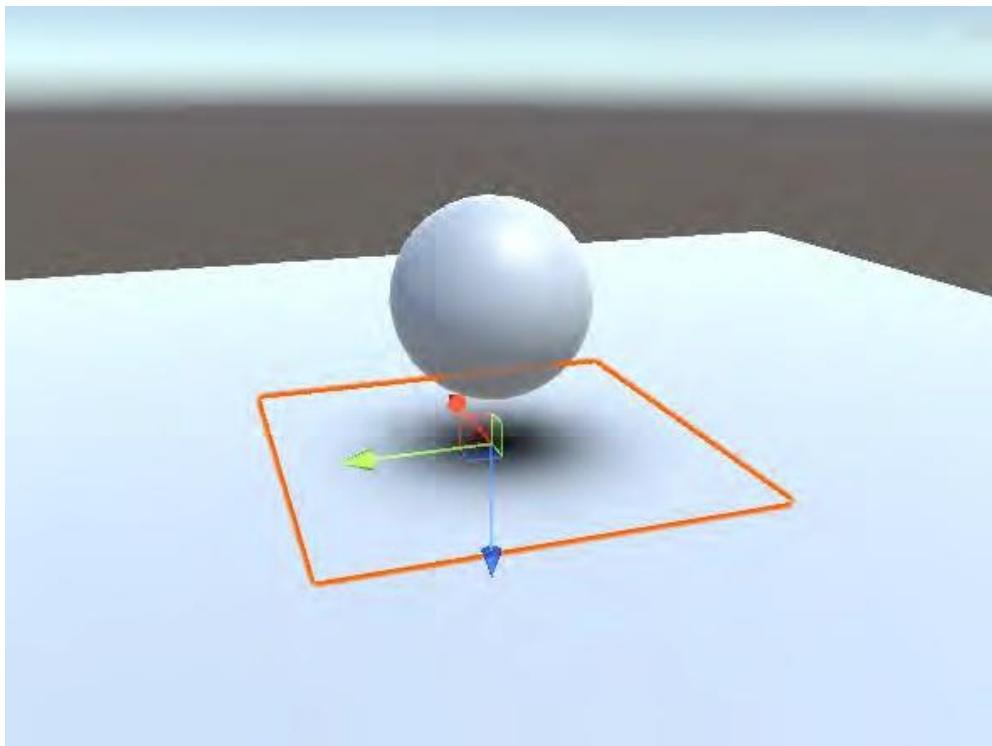


图7.23 带板块多边形的伪阴影

7.7.2 光影映射

通过提前将光照效果和阴影烘烤到纹理中，高质量的光照表达可以在比实时生成低得多的负载下实现。

要烘烤一个光照图，你首先需要设置放置在场景中的光照组件的模式。
将混合或烘烤项目改为混合或烘烤。

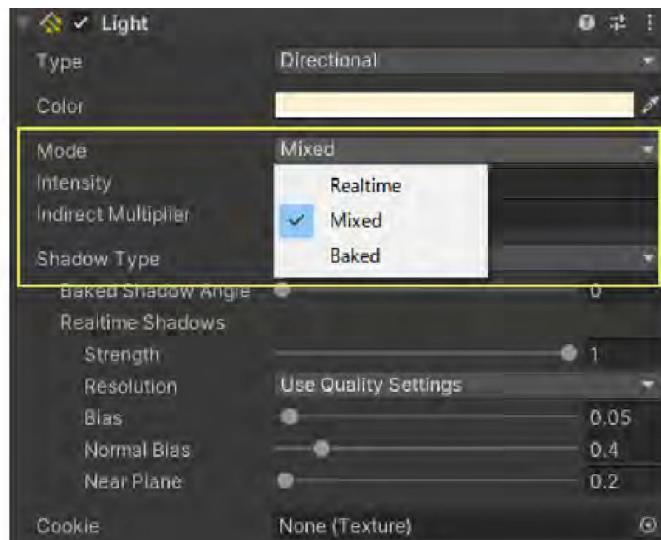
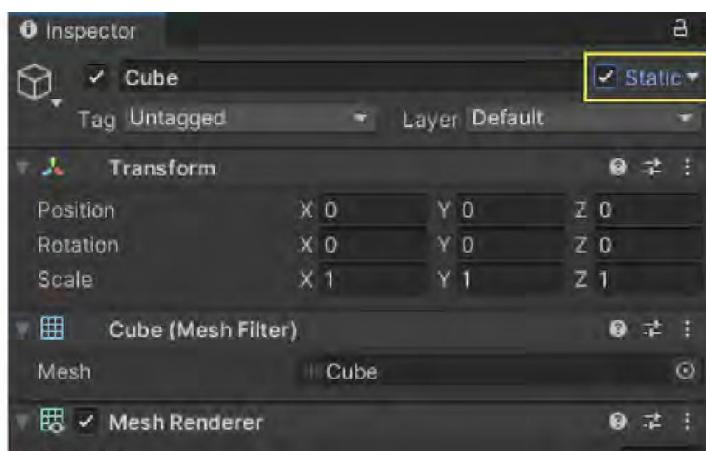


图7.24 照明的模式设置。

它还能使对象的静态标志得到烘烤。



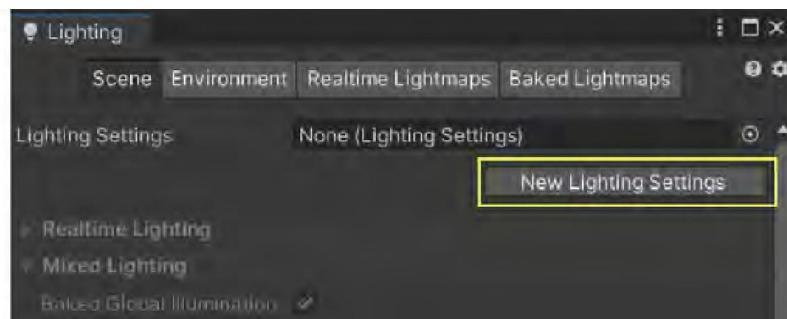
▲ 图7.25 启用静态

在这种状态下，从菜单中选择‘窗口->渲染->照明’，并选择照明查看。

默认情况下，照明设置资产没有被指定，因此新的

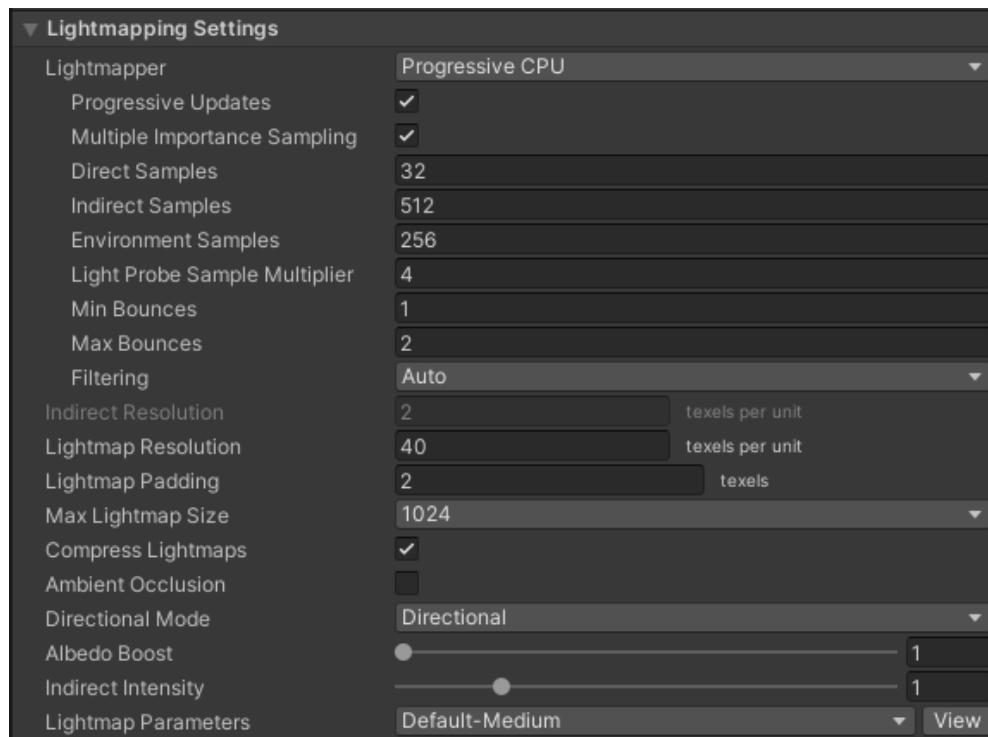
第7章 调试实践--图形

按下照明设置按钮，创建一个新的。



▲ 图7.26 新的照明设置

关于光影的设置主要是在光影设置标签中进行的。



▲ 图 7.27 光线映射设置

有许多设置选项，调整这些值将改变光影图的烘烤速度和质量。因此，你需要根据你所要求的速度和质量来设置适当的设置。

在这些设置中，**Lightmap Resolution**对性能影响最大。这个设置决定了Unity中每个单元分配多少光照图文本，由于最终的光照图大小取决于这个值，它对存储和内存容量以及纹理的访问速度都有很大影响。

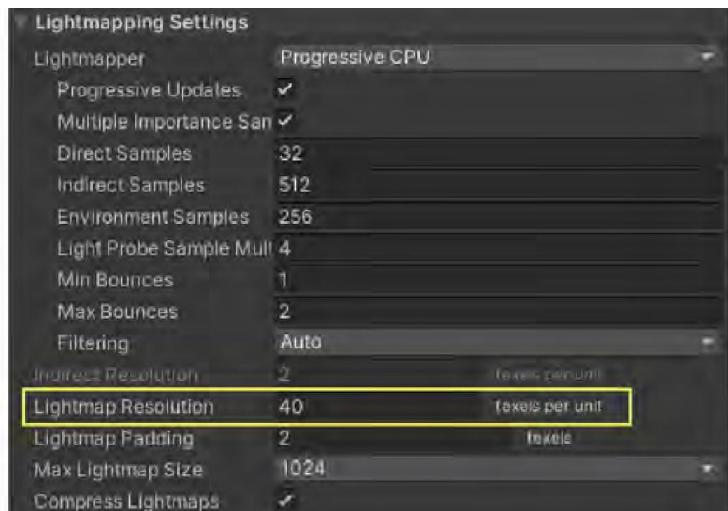


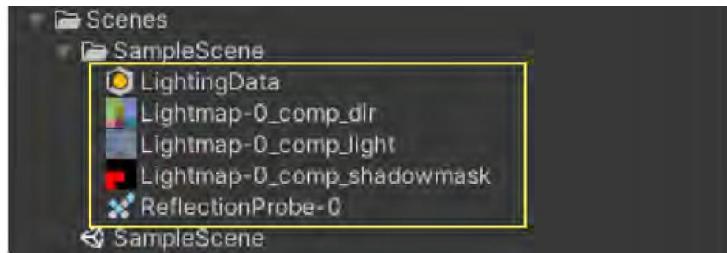
图7.28 光照图分辨率

最后，可以通过按检查器视图底部的生成照明按钮来烘烤光照图。烘烤完成后，你会看到烘烤后的光照图存储在一个与场景同名的文件夹中。

第7章 调试实践--图形



图7.29 生成照明



▲图7.30 烘烤光线图。

7.8 详细程度

用高多边形细节来渲染离摄像机较远的物体是低效的；**细节水平 (LOD)** 技术可以用
来根据物体与摄像机的距离来降低其细节水平。

在Unity中，LOD可以通过给一个物体添加**LOD组**组件来控制。

7.9 纹理流

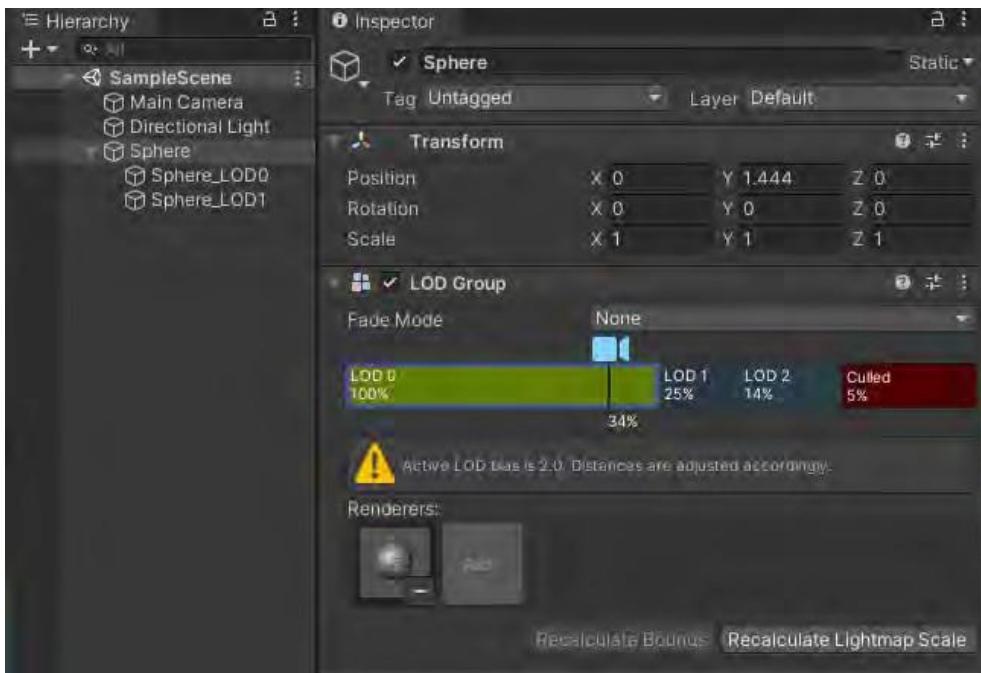


图7.31 LOD组。

通过在连接了LOD组的GameObject的子代上放置一个带有每个LOD级别的网格的渲染器，并在LOD组中设置每个LOD级别，LOD级别可以根据摄像机的情况进行切换。还可以为每个LOD组设置哪个LOD级别分配给摄像机的距离。

使用LOD通常可以减少绘图负荷，但必须注意避免内存和存储压力，因为每个LOD级别的所有网格都被加载。

7.9 纹理流

Unity的纹理流可以用来减少纹理所需的内存空间和加载时间。纹理流是一项功能，通过根据摄像机在场景中的位置加载mipmaps来节省GPU内存。

要启用这一功能，请在质量设置中激活纹理流。

第7章 调试实践--图形

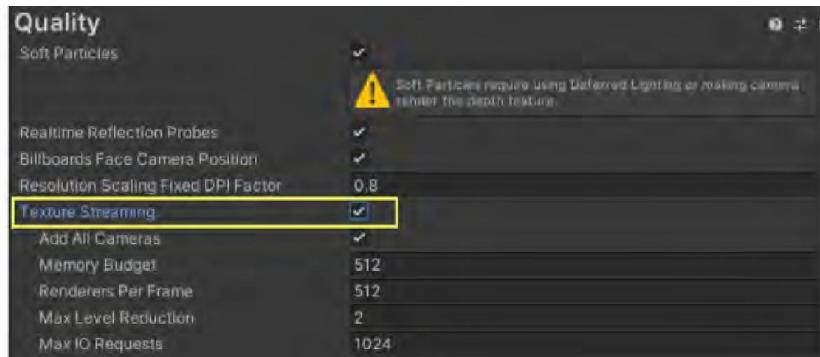


图7.32 纹理流。

此外，还需要改变纹理导入设置，以启用纹理的流式mipmaps。打开纹理的检查器，在高级设置中激活流式Mipmaps。

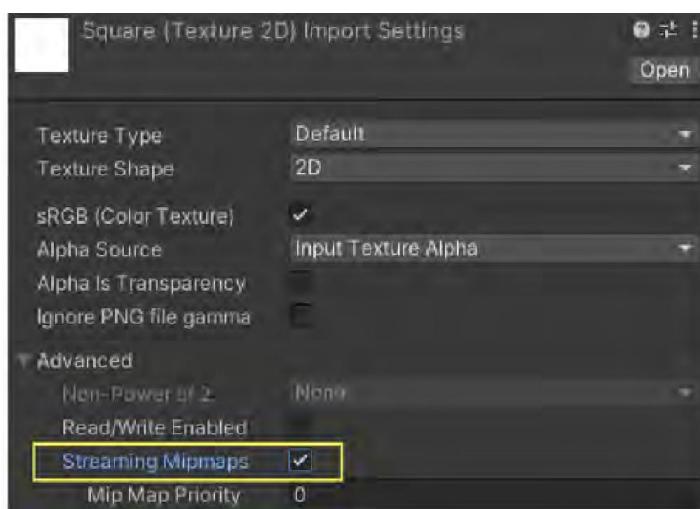


图7.33 流动的Mipmaps。

这些设置确保指定纹理的mipmap被流化。你也可以通过调整质量设置中的内存预算项目来限制加载纹理的总内存用量。纹理流系统将在不超过此处设置的内存量的情况下加

载mipmaps。



统一

性能调整

CHAPTER

0

第8章。

调试实践--用户界面

uGUI，Unity的标准UI系统，以及在屏幕上绘制文本的机制。

介绍了TextMeshPro的调谐实践。

8.1 分割画布

在uGUI中，当Canvas中的一个元素发生变化时，会运行一个进程（rebuild）来重建整个Canvas UI的网格。变化可以是任何东西，从外观的重大变化，如主动切换、移动或尺寸变化，到乍一看不明显的小变化。重建过程的成本很高，如果执行得太频繁或Canvas中存在大量的UI，会对性能产生负面影响。

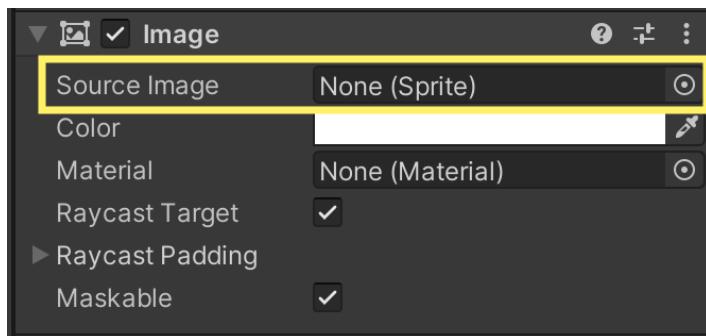
相比之下，通过将Canvas划分为UI的各个部分，就有可能创建一个可重建的UI。这可以减少动画重建的成本。例如，如果你有会移动的动画用户界面和不会移动任何东西的用户界面，你可以把它们放在不同的Canvas下，以尽量减少必须进行的动画重建的数量。

然而，需要仔细考虑如何划分画布，因为如果画布被分割，绘图批次将无法工作。

当Canvas被嵌套在Canvas之下时，Canvas的分割也是有效的。如果子画布中的元素发生变化，只对子画布进行重建，而不是对父画布进行重建。然而，仔细观察，当子Canvas中的UI被SetActive切换到活动状态时，情况似乎有所不同。这时，如果在父画布中放置大量的用户界面，似乎会出现负载变大的现象。我们不知道为什么会出现这种行为的细节，但似乎在切换嵌套Canvas中的UI的活动状态时应该小心。

8.2 统一白

在开发用户界面时，经常会有这样的情况：你想显示一个简单的矩形对象。UnityWhite是一个内置的Unity纹理，当Image或RawImage组件中使用的图像没有被指定时，就会使用它（图8.1）。UnityWhite可以在框架调试器中看到（图8.2）。这个机制可以用来绘制一个白色的矩形，然后与一个乘法颜色相结合，形成一个简单的矩形显示。



▲图8.1 使用UnityWhite



图8.2 使用中的UnityWhite。

然而，由于UnityWhite与项目中提供的SpriteAtlas是不同的纹理，这导致了绘制批次的断裂。这增加了绘图调用，降低了绘图效率。

因此，应该在SpriteAtlas中添加一个小的（例如 4×4 像素）白色正方形图像，并使用Sprite来绘制一个简单的矩形。对这一点。

因此，如果使用相同的SpriteAtlas，它将是相同的材料，所以批次可以工作。

8.3 布局组件

uGUI 提供了一个 Layout 组件，其功能是将对象整齐地排列。例如，VerticalLayoutGroup用于垂直对齐，GridLayoutGroup用于网格上的对齐（图8.3）。



图8.3 左边是VerticalLayoutGroup，右边是GridLayoutGroup的例子。

对于Layout组件，当目标对象被创建或某些属性被编辑时，Layout的重建会发生，Layout的重建是一个昂贵的过程，网格的重建也是如此。

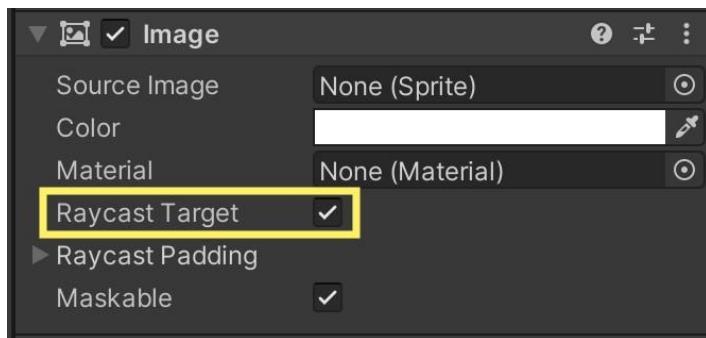
避免因Layout重建而导致性能下降的一个好方法是尽可能避免使用Layout组件。

如果你不需要动态放置，例如，根据文本内容改变位置，那么你就不需要使用布局组件。即使你真的需要动态定位，用你自己的脚本来控制它可能会更好，例如，如果它在屏幕上被大量使用。另外，如果要求相对于父本放置在一个特定的位置，即使父本的尺寸发生变化，也可以通过调整RectTransform锚来实现。如果你在创建预制件时使用了Layout组件，因为它方便放置，请务必删除它并保存它。

8.4 雷击目标

8.4 雷击目标

Graphic, Image和RawImage的基类，有一个叫做Raycast Target的属性（图8.4）。如果这个属性被启用，图形就会成为点击和触摸的目标。只要有可能，禁用该属性将提高性能，因为当屏幕被点击或触摸时，启用该属性的对象将成为目标。



▲图8.4 雷射目标属性

这个属性默认是启用的，但实际上许多图形不需要启用这个属性。另一方面，Unity有一个叫做Presets^{*1}的功能，允许你改变项目中的默认值。具体来说，你可以分别为Image和RawImage组件创建预设，并从项目设置中的预设管理器将它们注册为默认预设。你也可以使用这个功能来默认禁用Raycast Target属性。

8.5 面罩

要在uGUI中表现遮罩，可以使用Mask组件或RectMask2d组件。

^{*1} <https://docs.unity3d.com/ja/current/Manual/Presets.html>

第8章 调试实践--用户界面

掩模使用模板来实现掩模，所以每增加一个组件，绘图成本就会增加。相比之下，RectMask2d使用着色器参数来实现遮罩，所以减少了绘图成本的增加。然而，Mask可以被镂空成任何形状，而RectMask2d只能被镂空成一个矩形。

人们普遍认为，如果有RectMask2d，应该选择它，但最近Unity那么，在使用RectMask2d时也必须注意。

具体来说，当RectMask2d被启用时，随着遮罩目标数量的增加，每一帧都会出现用于剔除的CPU负载，与遮罩目标数量的增加成正比。这种现象，即即使UI没有以任何方式移动，每一帧都会出现CPU负载，似乎是uGUI内部实现中的一个修复的副作用，该修复是由这似乎是Unity 2019.3中包含的一个问题^{*2}的修复的副作用。

因此，尽可能避免使用RectMask2d是很有效的，即使使用它，在不需要它的时候也要把enabled设置为false，并把遮罩目标保持在必要的最小范围。

8.6 TextMeshPro.

在TextMeshPro中设置文本的常用方法是将文本分配给文本属性，但有一个替代方法叫SetText。SetText有许多重载，例如，它接受一个字符串和一个浮动值作为参数。使用这种方法，如清单8.1所示，第一个可以显示两个参数的值。然而，假设label是TMP_Text（或继承）类型的变量，number是float类型。

▼ 清单8.1 SetText的使用实例

```
1: label.SetText("{0}", number);
```

这种方法的优点是减少了生成字符串的成本。

▼ 清单8.2没有SetText的例子

```
1: label.text = number.ToString();
```

^{*2} <https://issuetracker.unity3d.com/issues/rectmask2d-diffrently-masks-image-in-the->

play-mode-when-animating-rect-transform-pivot-property

8.7 切换用户界面的显示

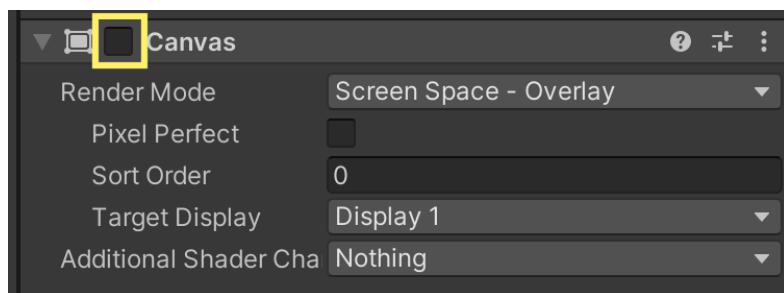
在使用文本属性的方法中，如清单8.2所示，执行了一个浮动的ToString()，每次执行这个过程都会产生一个字符串的代价。相比之下，使用SetText的方法旨在生成尽可能少的字符串，这在性能方面是有利的，特别是在要显示的文本经常变化的情况下。

这个TextMeshPro的功能在与ZString³结合时也非常强大，ZString是一个在字符串生成中减少内存分配的库。使用这些方法，可以实现灵活的文本显示，同时降低字符串生成的成本。

8.7 切换 用户界面的显示方式

uGUI组件的特点是用SetActive切换对象的成本很高。这是由于OnEnable为各种重建和初始化掩码设置了Dirty标志。因此，考虑SetActive方法的替代方法来切换用户界面的显示是很重要的。

第一种方法是将Canvas enabled设置为false（图8.5）。这将阻止画布下的所有对象被绘制。这种方法的缺点是，只有当你想把整个对象隐藏在Canvas下面时才能使用。



▲ 图8.5 禁用Canvas。

³ <https://github.com/Cysharp/ZString>

第3章 调试实践--用户界面

另一种方法是使用CanvasGroup，它有一个功能，可以让你一次性调整它下面所有对象的透明度。通过使用这个函数并将透明度设置为0，CanvasGroup下的所有对象都可以被隐藏起来（图8.6）。



图8.6 将CanvasGroup的透明度设为0

虽然这些方法有望避免由SetActive引起的负载，但可能需要注意，因为GameObject将保持在活动状态。例如，请注意，如果定义了一个更新方法，它将继续在隐藏状态下运行，这可能会导致负载的意外增加。

作为参考，我们测量了使用每种方法在显示和非显示之间切换时对带有图像组件的1280个游戏对象的处理时间（表8.1）。处理时间是在Unity编辑器中测量的，没有使用Deep Profile。该方法的处理时间是实际切换的执行时间^{*4}和该帧中UIEvents.WillRenderCanvases的执行时间之和。把它们加在一起的原因是，UI重建是在这种情况下进行的。

▼ 表8.1 显示切换的处理时间

技术	处理时间（显示）	处理时间（隐藏）
设置活动	323.79毫秒。	209.93毫秒。
启用帆布	61.25 ms	61.23 ms
CanvasGroup的阿尔法	3.64 ms	3.40 ms

表8.1中的结果显示，在这里测试的情况下，使用CanvasGroup的方法的处理时间是迄今为止最短的。

^{*4} 例如，如果使用SetActive，则调用SetActive方法的部分被包围在Profiler.BeginSample和Profiler.EndSample中并被测量。



统一

性能调整

CHAPTER

0

第9章。

调音实践--脚本（统一）。

随意使用Unity提供的功能会导致意想不到的陷阱。本章通过实例介绍了与Unity的内部实现有关的性能调优技术。

9.1 空的统一事件函数

如果定义了Unity提供的事件函数，如Awake、Start和Update，它们在运行时被缓存在Unity的内部列表中，并由列表的迭代执行。

将不需要的事件函数留在原地会使列表膨胀并增加迭代的成本，因为它们仅仅因为被定义而被缓存，即使它们在函数中不执行任何处理。

例如，在Unity上新生成的脚本，如下面的示例代码所示，将包含Start和Update从一开始就被定义了，但如果不需要的话，一定要删除这些函数。

▼ 清单9.1在 Unity上新生成的脚本

```
1: public class NewBehaviourScript : MonoBehaviour
2: {
3:     // 在第一帧更新之前调用Start void Start()
4:     {
5:     }
6:     }
7:     }
8:     }
9:     // 每一帧调用一次更新 void
10:    Update()
11:    {
12:    }
13:    }
14: }
```

9.2 访问标签和名称

继承自UnityEngine.Object的类被提供了标签和名称属性。

各自的实现都取自UnityCsReference。可以看出，这两个调用过程都是在本地代码中实现的。

Unity用C#实现脚本，但Unity本身是用C++实现的。由于C#内存空间和C++内存空间不能共享，所以要分配内存来将字符串信息从C++端传递到C#端。这是在每次调用时进行的，所以如果要多次访问它，应该进行缓存。

关于Unity如何工作以及C#和C++之间的内存的更多信息，请参阅Unity Runtime。

▼清单9.2。 改编自UnityCsReference GameObject.bindings.cs*1

```
1: public extern string tag
2: {
3:     [FreeFunction("GameObjectBindings::GetTag", HasExplicitThis = true)] 获
4:     取。
5:     [FreeFunction("GameObjectBindings::SetTag", HasExplicitThis = true)]
6:     设置。
7: }
```

▼清单9.3。 UnityCsReference 取自UnityEngineObject.bindings.cs*2

```
1: 公用字符串name
2: {
3:     get { return GetName(this); }
4: } set { SetName(this, value); }
5: }
6:
7: [FreeFunction("UnityEngineObjectBindings::GetName")]
8: extern static string GetName([NotNull("NullExceptionObject")] Object obj);
```

*¹ <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/GameObject.bindings.cs>

*² <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/UnityEngineObject.bindings.cs>

9.3 检索组件

GetComponent()是另一个需要注意的问题，它可以检索附属于同一个游戏对象的其他组件。

与上一节中的标签和名称属性一样，需要注意的是，“搜索”给定类型的组件，以及调用本地代码中实现的进程，都是有成本的。

在下面的示例代码中，每一帧搜索一个Rigidbody组件的成本是这将需要。如果你经常访问该网站，请使用网站的预缓存版本。

▼清单9.4 每一帧的GetComponent()代码

```
1: void Update()
2: {
3:     刚体 rb = GetComponent<Rigidbody>();
4:     rb.AddForce (Vector3.up * 10f);
5: }
```

9.4 获得改造的机会

变换组件是一个经常被访问的组件，用于位置、旋转、比例（扩张和收缩）和父子关系的改变。它通常会更新多个值，如下面的示例代码所示。

▼清单9.5访问transform的例子

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale) 2:
{
3:     transform.position = position;
4:     transform.rotation = rotation;
5: }     transform.localScale = scale;
6: }
```

一旦获得了变换，在Unity内部就会调用GetTransform()过程。这比上一节中的GetComponent()进行了优化，速度更快。然而，它比缓存的情况要慢，所以它也应该被缓存，并像下面的示例代码那样访问。位置和旋转也可以通过使用SetPositionAndRotation()来减少。

9.5 需要明确销毁的类

▼ 清单 9.6 缓存转换的例子

```
1: void SetTransform(Vector3 position, Quaternion rotation, Vector3 scale) 2:  
{  
3:     var transformCache = transform;  
4:     transformCache.SetPositionAndRotation(position, rotation);  
5:     transformCache.localScale = scale。  
6: }
```

9.5 需要明确销毁的类

由于Unity是用C#开发的，那些不再被GC引用的对象被释放了。然而，Unity中的一些类需要被明确地销毁。典型的例子包括Texture2D、Sprite、Material和PlayableGraph，如果它们是用新的或专门的Create函数创建的，必须明确地销毁。

▼ 清单9.7 生成和显式销毁

```
1: void Start()  
{  
2:     _texture = new Texture2D(8, 8);  
3:     _sprite = Sprite.Create(_texture, new Rect(0, 0, 8, 8), Vector2.0);  
4:     _material = new Material(shader);  
5:     _graph = PlayableGraph.Create();  
6:  
7: }  
8:  
9: void OnDestroy().  
10: {  
11:     Destroy(_texture);  
12:     Destroy(_sprite);  
13:     Destroy(_material);  
14:  
15:     如果  
16:     (_graph.IsValid())  
17:     {  
18:         _graph.Destroy()  
19:     }
```

9.6 字符串规范

避免使用字符串来指定在Animator中播放哪些状态和在Material中操作哪些属性。

第9章 调试实践--脚本(Unity)

▼ 清单9.8 字符串规范的例子

```
1: _animator.Play("Wait");
2: _material.SetFloat("_Prop", 100f);
```

在这些函数里面，`Animator.StringToHash()`和`Shader.PropertyToID()`被执行，将字符串转换为唯一的识别值。在多次访问该函数时，每次都进行转换是很浪费的，所以建议缓存识别值并使用它们。定义一个列出缓存的识别值的类，如下面的例子所示，对方便使用有好处。

▼ 清单9.9 识别值缓存的例子

```
1: public static class ShaderProperty
2: {
3:     public static readonly int Color = Shader.PropertyToID("_Color");
4:     public static readonly int Alpha = Shader.PropertyToID("_Alpha"); 5:
5:     public static readonly int ZWrite = Shader.PropertyToID("_ZWrite");
6: }
7: public static class AnimationState
8: {
9:     public static readonly int Idle = Animator.StringToHash("idle");
10:    public static readonly int Walk = Animator.StringToHash("walk");
11:    public static readonly int Run = Animator.StringToHash("run");
12: }
```

9.7 JsonUtility的陷阱。

Unity提供了一个名为`JsonUtility`的类，用于序列化/反序列化JSON。官方文档^{*3}还指出，它比C#标准更快，通常用于注重性能的实现。

基准测试表明，`JsonUtility`的速度明显要快得多（尽管功能比.NET的少）。

然而，有一件与性能有关的事情需要注意。这就是“空处理”。

下面的示例代码显示了序列化过程和它的结果。尽管明确地将类A的成员`b1`设置为空，但类B和类C是

^{*3} <https://docs.unity3d.com/ja/current/Manual/JSONSerialization.html>

9.8 渲染和MeshFilter的陷阱

你可以看到，它被序列化为由默认构造函数生成。如果要序列化的字段有一个空值，在转换为JSON时将会出现一个假的对象，所以最好是考虑到这个开销。

▼清单9.10 串行化行为

```
1: [可序列化] 公用类A { 公用B b1; }
2: [可序列化] 公众类B { 公众C c1; 公众C c2; } 3: [可序列化]
公众类C { 公众int n; }
4:
5: void Start()。
6: {
7:     Debug.Log(JsonUtility.ToJson(new A() { b1 = null, }));
8:     // {"b1":{"c1":{"n":0}, "c2":{"n":0}}}.
9: }
```

9.8 渲染和MeshFilter的陷阱

用Renderer.material获取的材质和用MeshFilter.mesh获取的网格都是重复的实例，在使用完后必须明确地销毁它们。官方文件^{*4}^{*5}也清楚地说明了每一个人的情况。

如果该材质被其他渲染器使用，这将克隆共享的材质，并从现在开始使用它。

当游戏对象被销毁时，你有责任销毁自动实例化的网格。

获得的材料和网格应该保存在成员变量中，并在适当的时候销毁。

▼清单9.11。 明确销毁复制的材料

```
1: void Start()。
2: {
3:     _material = GetComponent<Renderer>().material;
4: }
5:
6: void OnDestroy()。
7: {
```

^{*4} <https://docs.unity3d.com/ja/current/ScriptReference/Renderer-material.html>

^{*5} <https://docs.unity3d.com/ja/current/ScriptReference/MeshFilter-mesh.html>

```
8:     如果(_material !=  
null) { 9:     Destroy(_material)  
10:    }  
11: }
```

9.9 删除了日志输出代码。

Unity 提供了日志输出的函数，如 Debug.Log()，Debug.LogWarning() 和 Debug.LogError()。虽然这些功能很有用，但也有一些问题。

- 日志输出本身是一个相当重的过程。
- 也在发布版本中执行。
- GC.Alloc 是由创建或串联字符串引起的

如果Unity中的日志设置被关闭，堆栈跟踪将停止，但日志仍将被输出；在Unity中设置 UnityEngine.Debug.unityLogger.logEnabled为false，将不会输出任何日志，但该函数将是内部的。因为它只是分支，所以将完成函数调用的成本和不应该需要的字符串的创建和串联。另一个选择是使用#**if**指令，但要处理所有的日志输出处理是不现实的。

▼清单9. 12#**if**指令

```
1: #if UNITY_EDITOR.  
2:     Debug.LogError($"Error {e}").  
3: #endif
```

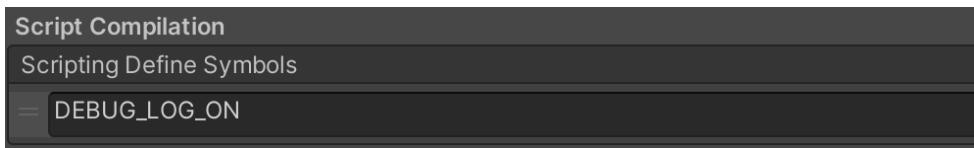
在这种情况下可以利用条件属性：对于具有条件属性的函数，如果指定的符号没有被定义，编译器将删除调用部分。如清单9.13中的示例所示，最好给自己类中的每个函数添加一个**Conditional**属性，作为通过自己的日志输出类在Unity端调用日志函数的规则，这样在必要时可以删除整个函数调用。

9.10 用Burst加快代码速度

▼清单9. 13条件属性示例

```
1: public static class Debug
2: {
3:     private const string MConditionalDefine = "DEBUG_LOG_ON";
4:
5:     [System.Diagnostics.Conditional(MConditionalDefine)]
6:     public static void Log(object message)
7:         => UnityEngine.Debug.Log(message);
8: }
```

需要注意的一点是，指定的符号必须能够被函数调用者引用。#在每个调用具有条件属性的函数的文件中定义符号是不现实的。Unity有一个叫做脚本化符号的功能，允许你为整个项目定义符号。这可以在项目设置->播放器->其他设置中进行配置。



▲图9.1 脚本定义符号。

9.10 用Burst加快代码速度

Burst⁶是用于高性能C#脚本的官方Unity编译器。

Burst使用C#语言的一个子集来编写代码；Burst将C#代码转换成IR（Intermediate Representation），这是一种名为LLVM的编译器基础设施⁷的中间语法，然后在优化IR后将其转换成机器语言。

然后，代码被尽可能地矢量化，并被一个主动使用指令SIMD的过程所取代。预计这将产生更快的方案产出。 SIMD是单指令/多数据的意思，即用一条指令同时处理多个数据。

⁶ <https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/docs/QuickStart.html>

⁷ <https://llvm.org/>

第9章 调试实践--脚本(Unity)

这指的是指令，使它们被应用于换句话说，通过积极使用SIMD指令，数据在一条指令中被一起处理，这比使用普通指令要快。

9.10.1 用Burst加快代码速度

Burst使用C#语言的一个子集，称为高性能C#（HPC#）^{*8}来编写代码。

HPC#的一个特点是，C#的引用类型，即类和数组，是不可用的。因此，作为一项规则，数据结构要用结构来描述。

集合，比如数组，可以使用NativeContainer^{*9}，比如NativeArray< T >。使用；关于HPC#的更多信息，请参考脚注中列出的文档。 Burst是与C#作业系统结合使用的。Burst与C#作业系统结合使用，因此其自身的处理过程在实现IJob的作业的Execute方法中描述。定义的工作有一个BurstCompi
通过指定-le属性，作业被Burst优化。

清单9.14显示了一个对给定数组的每个元素进行平方处理并将其存储在输出数组中的例子。

▼清单9.14。 简单验证的工作实施

```
1: [BurstCompile].  
2: private struct MyJob : IJob  
3: {  
4:     [ReadOnly].  
5:     public NativeArray<float> Input;  
6:  
7:     [WriteOnly].  
8:     public NativeArray<float> Output;  
9:  
10:    public void Execute()  
11:    {  
12:        for (int i = 0; i < Input.Length; i++)  
13:        {  
14:            Output[i] = Input[i] * Input[i];  
15:        }  
16:    }  
17: }
```

清单9.14第14行的每个元素都可以独立计算（计算中没有顺序依赖），并且可以使用SIMD指令一起计算，因为输出数组的内存对齐是连续的。

^{*8}<https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/>

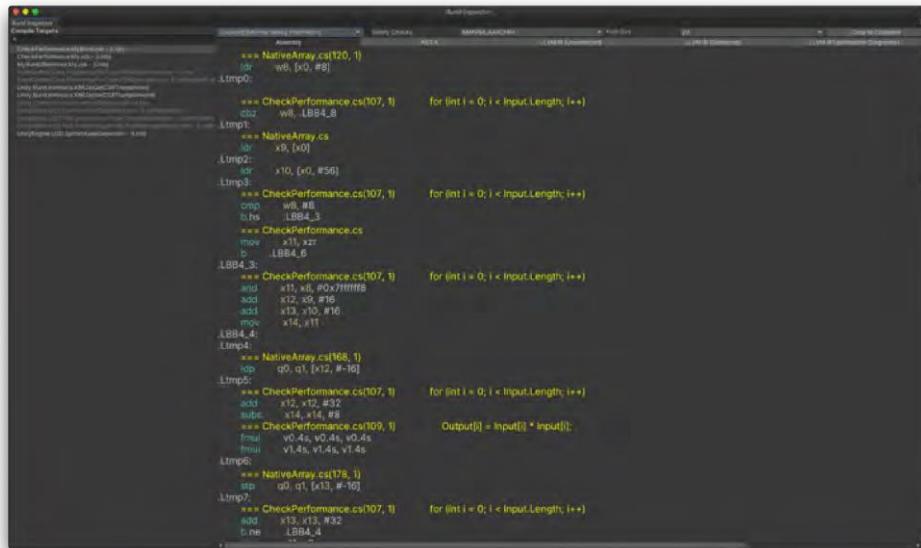
CSharpLanguageSupport_Types.html

*⁹ <https://docs.unity3d.com/Manual/JobSystemNativeContainer.html>

9.10 用Burst加快代码速度

要查看代码被转换为什么汇编，见图9.2 突发检查器

这可以用



The screenshot shows the Unity Editor's Burst Inspector window. It displays assembly code for a SIMD multiplication operation. The code is annotated with comments from the C# source code, such as `NativeArray.cs(120, 1)` and `CheckPerformance.cs(107, 1)`. The assembly instructions include `fmul` (multiplication), `add` (addition), and `mov` (move). Registers used include `x0` through `x15` and `y0` through `y15` for SIMD lanes. Labels like `LBB4_1`, `LBB4_2`, etc., are present. The code is organized into several blocks corresponding to different parts of the C# code.

图9.2. Burst Inspector可以用来查看代码是如何被转化为程序集的。

清单9.14中第14行的处理被列在ARMV8A_AARCH64的汇编中。

转换为9.15。

▼清单9.15。 清单9.14第14行的ARMV8A_AARCH64的汇编

1:	fmul	V0.4S, V0.4S, V0.4S
2:	fmul	v1.4s, v1.4s, v1.4s

汇编操作数以`.4s`为后缀的事实证实了使用了SIMD指令。

比较由纯C#实现的代码和由Burst优化的代码在真实机器上的性能。

安卓Pixel 4a和IL2CPP是在实际设备上建立的，有脚本后台进行比较。数组的大小为 $2^{20} = 1,048,576$ 。同样的过程重复了10次，并取得了平均处理时间。

第9章 调试实践--脚本(Unity)

表9.1显示了性能比较的测量结果。

表9.1 纯C#实现和Burst优化代码的处理时间比较

技术	处理时间 (隐藏)
纯粹的C#实现	5.73 ms
按突发事件实施	0.98 ms

我们观察到，与纯C#实现相比，速度提高了约5.8倍。



统一

性能调整

CHAPTER

1

第十章 调音实践--脚本（C#）。

第10章。

调音实践--脚本(C#)

本章主要通过实际案例介绍了C#代码的性能调优技术；这里不涉及基本的C#符号，而是开发对性能要求高的游戏时需要注意的设计和实现。

10.1 GC.分配案件以及如何处理这些案件

正如2.5.2垃圾收集中所介绍的，在这一节中，我们将首先了解GC.Alloc用于哪些具体过程。

10.1.1 参考类型新

第一个是GC.Alloc的一个非常直接的案例。

▼列表10.1 每一帧的GC.Alloc代码

```
1: 私有 void Update() 2: {  
3:     const int listCapacity = 100;  
4:     // GC.Alloc与List<int>中的  
5:     var list = new List<int>(listCapacity);  
6:     for (var index = 0; index < listCapacity; index++)  
7:     {  
8:         // 将索引打包成一个没有特殊含义的List。  
9:         list.Add(index);  
10:    }  
11:    // 从列表中随机获取一个值  
12:    var random = UnityEngine.Random.Range(0, listCapacity); var  
13:    randomValue = list[随机];  
14:    // ... 从一个随机值做一些事情...  
15: }
```

这段代码的主要问题是，每一帧执行的Update方法会使List<in t>变成新的。

为了解决这个问题，可以事先生成一个List<int>，用来生成一个

有可能避免GC.Alloc。

▼清单10.2。 消除GC.Alloc每一帧的代码

```
1: 私有的静态的 readonly int listCapacity = 100; 2:  
//提前生成列表。  
3: private readonly List<int> _list = new List<int> (listCapacity) ;  
4:  
5: private void Update()  
6: {  
7:     _list.Clear();  
8:     for (var index = 0; index < listCapacity; index++) 9:  
    {  
10:         // 将索引打包成一个List, 虽然它并不意味着什么特别的东西。  
11:         _list.Add(index);  
12:     }  
13:     // 从列表中随机获取一个值  
14:     var random = UnityEngine.Random.Range(0, listCapacity);  
15:     var randomValue = _list[随机];  
16:     // ... 从一个随机值做一些事情...  
17: }
```

你不太可能写出像这里的示例代码那样的无稽之谈，但类似的例子可以在更多的情况下找到，比你想象的要多。

如果你失去了GC.Alloc。

你可能已经注意到，上面清单10.2中的示例代码就是所需的全部内容。

```
1 : var randomValue = UnityEngine.Random.Range (0,  
listCapacity) ; 2 : // ... 从一个随机值做一些事情...
```

在性能调优中考虑消除GC.Alloc是很重要的，但总是考虑消除无意义的计算是加快进程的一个步骤。

以下是调查的结果摘要。

10.1.2 lambda-style

Lambda表达式也很有用，但它们在游戏中的使用是有限的，因为它们也可能导致GC.Alloc，这取决于它们的使用方式。在此，假设定义了以下代码。

第十章 调音实践--脚本（C#）。

▼清单10.4。 lambda表达样本的假设代码

```
1: // 成员变量
2: private int _memberCount = 0;
3:
4: // 静态变量
5: 私有静态int _staticCount = 0; 6:
7: // 成员方法
8: private void IncrementMemberCount() 9:
{
10:     _memberCount++;
11: }
12:
13: // 静态方法
14: 私有的静止无效的IncrementStaticCount()
15: {
16:     _staticCount++;
17: }
18:
19: // 成员方法，只调用收到的行动。
20: private void InvokeActionMethod(System.Action action
action) 21: {
22:     action.Invoke();
23: }
```

当一个变量在lambda表达式中被引用时，GC.Alloc会发生，如下所示。

▼清单10.5。 GC.Alloc通过引用λ表达式中的一个变量的情况

```
1: // 如果引用了成员变量就会发生委托分配 2:
InvokeActionMethod(() => { _memberCount++; });
3:
4: // 如果引用了一个局部变量，就会发生闭合分配 5: int count
= 0;
6: // 也发生了与上述相同的代表分配情况
7: InvokeActionMethod(() => { count++; });
```

然而，这些GC.Alloc可以通过引用静态变量来避免，如下所示。

▼清单10.6。 在lambda表达式中，静态变量没有被GC.Alloc引用的情况

```
1: // 如果引用的是静态变量，则不会发生GC Alloc。
2: InvokeActionMethod(() => { _staticCount++; });
```

lambda表达式中的方法引用也是GC.Alloc' d不同，取决于它们的描述方式。

▼清单10.7。 在lambda表达式中引用GC.Alloc方法的情况

```
1: // 如果引用了成员方法，就会发生委托分配 2:  
InvokeActionMethod(() => { IncrementMemberCount(); }); 3:  
4: // 如果直接指定成员方法，就会发生委托分配 5:  
InvokeActionMethod(IncrementMemberCount);  
6:  
7: // 直接指定一个静态方法将导致一个Delegate分配 8:  
InvokeActionMethod(IncrementStaticCount);
```

为了避免这些，你需要以语句形式引用静态方法，如下所示。

▼清单10.8。 GC.Alloc在lambda表达式中没有提到方法的情况

```
1: // 如果一个静态方法在lambda表达式中被引用，GC.Alloc就不会发生。  
2: InvokeActionMethod(() => { IncrementStaticCount(); })。
```

这样一来，Action只有第一次是新的，但从第二次开始它就被内部缓存了，以避免GC.Alloc。

然而，让所有的变量和方法都变成静态的并不是一个安全和可读的代码选项。在需要快速的代码中，对于每一帧或不确定时间发生的事件，不使用lambda表达式进行设计是比较安全的，而不是使用大量的静态和消除GC.Alloc。

10.1.3 使用泛型的案例来框定

在以下使用仿制药的情况下，什么会导致装箱？

▼清单10.9。 可能使用泛型的方框的例子

```
1: public readonly struct GenericStruct<T> : IEquatable<T>  
2: {  
3:     Private readonly T _value;  
4:  
5:     public GenericStruct(T value)  
6:     {  
7:         _value = value;  
8:     }  
9:  
10:    public bool Equals(T other)
```

第十章 调音实践--脚本（C#）。

```
11:     {  
12:         var result = _value.Equals(other);  
13:         return result;  
14:     }  
15: }
```

在这种情况下，程序员在GenericStruct中实现了IEquatable<T>接口，但忘记了对T的限制。因此，一个没有实现IEquatable<T>接口的类型可以被指定为T，有一种情况是下面的Equals被隐含地投到Object类型。

▼清单10.10 Object.cs.

```
1: public virtual bool Equals(object obj);
```

例如，为T指定一个没有实现IEquatable< T>接口的结构，将导致在Equals参数中被投向对象，这将导致装箱。为了提前防止这种情况的发生，请作如下更改。

▼清单10.11。防止拳击的限制实例

```
1: public readonly struct GenericOnlyStruct<T> : IEquatable<T>  
2:     其中，T : IEquatable<T>。  
3: {  
4:     Private readonly T _value;  
5:  
6:     public GenericOnlyStruct(T value)  
7:     {  
8:         _value = value;  
9:     }  
10:  
11:     public bool Equals(T other)  
12:     {  
13:         var result = _value.Equals(other);  
14:         返回结果。  
15:     }  
16: }
```

使用where子句（通用类型约束），T所接受的类型是IEquatable<T>>>将它们限制在实现它们的类型中，可以防止这些意外的框定。

永远不要忘记最初的目的

正如第2.5.2节垃圾收集所描述的那样，有很多情况下，选择结构的目的是为了避免游戏中运行时的GC.Alloc。然而，为了减少GC.Alloc，并不总是能够通过把所有东西都变成结构来加快游戏速度。

一个常见的错误是为了避免GC.Alloc而加入结构，正如预期的那样，这减少了与GC相关的成本，但由于值类型的复制成本，大数据量导致了低效的处理。

也有一些技术可以通过使用方法参数通过引用传递来减少复制成本，从而进一步避免这种情况。虽然这可能会导致速度加快，但在这种情况下，你应该考虑从一开始就选择一个类，并实现一个预先生成的实例，并在周围使用。Alloc不应该被取消，但最终目标是减少每帧的处理时间。

10.2 关于for/foreach

正如第2.6节算法和计算复杂度中所介绍的那样，根据数据的数量，循环变得很耗时。

另外，乍看之下，循环是同一个过程，但由于代码的编写方式不同，其效率也不同。

这里，SharpLab^{*1}被用来存储列表和数组的内容，使用foreach/for在1让我们来看看将代码从IL反编译到C#的结果，它只是一个一个地进行检索。首先，让我们看看一个foreach循环，在这个循环中，向List添加值等是通过使用缩略语。

▼ 清单10.12用foreach转动列表的例子

```
1: var list = new List<int>(128);
2: foreach (var val in list)
3: {
4: }
```

^{*1} <https://sharplab.io/>

第十章 调音实践--脚本（C#）。

▼ 清单 10.13 用 foreach 转化 List 的例子的反编译结果。

```
1: List<int>.Enumerator enumerator = new List<int>(128).GetEnumerator(); 2:  
try  
3: {  
4:     while (enumerator.MoveNext())  
5:     {  
6:         int current = enumerator.Current;  
7:     }  
8: }  
9: 最后  
。  
10: { ((IDisposable)enumerator).Dispose()  
11:   。  
12: }
```

当转入foreach时，可以看到实现是获取枚举器，用**MoveNext()**推进到下一步，用**Current**引用值。此外，看看list.cs²中**MoveNext()**的实现，似乎增加了各种属性访问的数量，如大小检查，处理量大于索引器的直接访问。

接下来，让我们来看看当你交出的时候会发生什么。

▼ 清单 10.14 使用 for 的转向列表示例

```
1: var list = new List<int>(128);  
2: for (var i = 0; i < list.Count; i++) 3:  
{  
4:     var val = list[i];  
5: }
```

▼ 清单 10.15 当 List 被转入时的反编译结果为

```
1: List<int> list = new List<int>(128); 2:  
int num = 0;  
3: while (num < list.Count)  
4: {  
5:     int num2 = list[num];  
6:     num++.  
7: }
```

在C#中，**for**语句是**while**语句的糖衣语法，可以看出它是通过索引器（**public T this[int index]**）通过引用获得的。另外，仔细观察这个**while**语句可以发现，条件表达式包含了**list.Count**。这意味着，对Count属性的访问

² <https://referencesource.microsoft.com/#mscorlib/system/collections/generic/list.cs>

10.2 关于for/foreach

随着Counts数量的增加，对Count属性的访问数量也成比例增加，根据数量的不同，负载变得不可忽略。如果Count在循环中不发生变化，可以通过在循环前缓存来减少属性访问的负载。

▼ 清单 10.16 用for翻转列表的例子：改进版本

```
1: var count = list.Count;
2: for (var i = 0; i < count; i++) 3:
{
    4:     var val = list[i];
    5: }
```

▼ 清单 10.17 转动清单的例子 10. 17带 有 for的 清单：改进版本的反编译结果

```
1: List<int> list = new List<int>(128);
2: int count = list.Count;
3 : int num = 0;
4: while (num < count)
5: {
6:     int num2 = list[num];
7:     num++;
8: }
```

缓存计数减少了财产访问的数量，加快了进程。这个循环中的两个比较都不是由GC.Alloc加载的，差异是由于实现上的不同。

在数组的情况下，foreach也被优化了，几乎改变了写在For不应提供。

▼ 清单 10.18。 用 foreach转动一个数组的例子

```
1: var array = new int[128]; 2:
foreach (var val in array) 3: {
4: }
```

▼ 清单 10.19。 围绕数组的 foreach的一个例子的反编译结果

```
1: int[] array = new int[128]; 2:
int num = 0;
3: while (num < array.Length)
4: {
5:     int num2 = array[num];
6:     num++;
```

7: }

为了验证，事先分配了一个随机数，作为数据的数量 10,000,000，和一个List<int>应计算数据的总和。验证环境是Pixel 3a，Unity 2021.3.1f1 该项目实施于。

▼ 表10.1 List<int>中每个描述方法的测量结果。

类型。	时间ms
列表：foreach	66.43
列表：为	62.49
列表：for (Count cache)	55.11
阵列：用于	30.53
数组：foreach	23.75

在List<int>的情况下，与精细对齐条件的比较表明，带有Count优化的for甚至比foreach更快。List中的foreach可以被改写为带有Count优化的for，以foreach可以重写为for，并进行Count优化，以减少foreach过程中MoveNext()和Current属性的开销，从而使其更快。此外，最快的List和Array之间的比较分别显示，Array比Lists快大约2.3%。

其结果是两倍多的速度：即使foreach和for的编写方式使IL产生相同的结果，foreach也会产生更快的结果，而且数组上的foreach也得到了很好的优化。

从上述结果来看，在有大量数据和必须提高处理速度的情况下，应该考虑用数组代替Lists< T>。然而，如果重写不充分，例如当一个字段中定义的List没有被本地缓存和引用时，可能无法加快进程。

10.3 对象池

正如自始至终提到的，在游戏开发中，预先生成和使用对象而不动态生成它们是很重要的。这就是所谓的对象池。例如，可以在游戏阶段避免GC.Alloc，方法是通过在加载阶段一起生成将在游戏阶段使用的对象，并在使用时才将其分配和引用到池中的对象。

除了减少分配，对象池还可以用来减少加载时间，方法是实现屏幕转换而不必每次都重新创建构成屏幕的对象，通过保留计算成本非常高的进程的结果来避免多次执行繁重的计算，等等。它被用在各种情况下。

尽管我们在最广泛的意义上使用了对象一词，但它不仅适用于最小的数据单位，也适用于Coroutines和Actions等。例如，考虑提前创建超过预期数量的Coroutines，并在必要时使用它们，例如在一个2分钟的游戏中，将执行多达20次，先创建IEnumerators，只在必要时使用它们。只在使用StartCoroutine时，可以减少生成的成本。

10.4 绳子

字符串对象是代表字符串的System.Char对象的顺序集合。字符串可以用一种方式使用，GC.Alloc很容易发生。例如，使用字符连接操作符 "+"连接两个字符串的结果是创建一个新的字符串对象；由于字符串的值在其创建后不能被改变（它是不可改变的），那些看似改变值的操作创建并返回一个新的字符串对象。一个新的字符串对象被创建并返回。

▼清单10.20。 字符串串联以创建一个字符串

```

1: 私有字符串CreatePath() 2
:
3:     var path = "root";
4:     path += "/";
5:     path += "Hoge";
6:     path += "/";
7:     path += "Fuga";
8:     返回路径。
9: }
```

在上面的例子中，每个字符串连接产生一个字符串，导致总分配量为164字节分配。

如果字符串经常变化，请使用一个具有可改变值的StringBuilder。以防止大量创建字符串对象。通过在StringBuilder对象中进行字符连接和删除等操作，最后检索值并将其ToString()到字符串对象中，可以将内存分配限制为只获取。另外，当使用StringBuilders时，一定要设置Capacity

第十章 调音实践--脚本（C#）。

如果不指定，初始值将是16。如果不指定，默认值为16。当缓冲区因字符数增加而扩展时，如Append，会进行内存分配和数值复制，所以一定要设置一个适当的Capacity，以避免无意中的扩展。

▼ 清单10.21 用StringBuilder创建一个字符串

```
1 : 私有的只读的StringBuilder _ stringBuilder = new StringBuilder(16); 2 :  
私有的字符串CreatePathFromStringBuilder()  
3: {  
4:     _ stringBuilder.Clear();  
5:     _ stringBuilder.Append("root");  
6:     _ stringBuilder.Append("/");  
7:     _ stringBuilder.Append("Hoge");  
8:     _ stringBuilder.Append("/");  
9:     _ stringBuilder.Append("Fuga");  
10:    返回 _ stringBuilder.ToString();  
11: }
```

在使用StringBuilder的例子中，如果StringBuilder是提前生成的（在上面的例子中，在生成时分配112Byte），那么ToString()检索生成的字符串只需要50Byte分配。

然而，当你想避免GC.Alloc时，也不建议使用StringBuilder，因为它只是在数值操作过程中不太可能引起分配，而且如上所述，在执行ToString()时将创建一个字符串对象。另外，由于\$""语法被转换为string.Format，而string.Format的内部实现使用StringBuilder，最终无法避免ToString()的代价。上一节描述的对象的使用在这里也应该适用，有可能被提前使用的字符串应该是预生成的字符串对象并使用。

然而，在有些情况下，必须在游戏中进行字符串操作和创建字符串对象。在这种情况下，你需要事先为字符串准备好一个缓冲区，并对其进行扩展，使其可以照常使用，可以通过实现你自己的不安全代码，或者使用面向Unity的库的扩展（例如，将NonAlloc应用于TextMeshPro的能力）。考虑引入。

³ <https://github.com/Cysharp/ZString>

10.5 LINQ和延迟评估

本节介绍了如何通过使用LINQ减少GC.Alloc以及延迟评估的要点。

10.5.1 通过使用LINQ减轻GC.Alloc的影响

在LINQ中，GC.Alloc出现在清单10.22这样的情况下。

▼ 清单 10.22 发生 GC.Alloc 的例子。

```
1: var oneToTen = Enumerable.Range(1, 11).ToArray();
2: var query = oneToTen.Where(i => i % 2 == 0).Select(i => i * i);
```

清单10.22中的GC.Alloc的原因是由于LINQ的内部实现。此外，一些LINQ方法会根据调用者的类型进行优化，所以GC.Alloc的大小会根据调用者的类型而变化。

▼ 清单 10.23 特定类型的执行速度验证

```
1: 私有 int[] 数组。
2: 私有的 List<int> list。
3: private IEnumerable<int> ienumerable;
4:
5: public void GlobalSetup()
6: {
7:     array = Enumerable.Range(0, 1000).ToArray();
8:     List = Enumerable.Range(0, 1000).ToList();
9:     ienumerable = Enumerable.Range(0, 1000)
10: }   。
11:
12: public void RunAsArray()
13: {
14:     var query = array.Where(i => i % 2 == 0);
15:     foreach (var i in query){}。
16: }
17:
18: public void RunAsList()
19: {
20:     var query = list.Where(i => i % 2 == 0);
21:     foreach (var i in query){}。
22: }
23:
24: public void RunAsIEnumerable()
25: {
26:     var query = ienumerable.Where(i => i % 2 == 0);
27:     foreach (var i in query){}。
28: }
```

第十章 调音实践--脚本（C#）。

测量清单10.23中定义的每个方法的基准，得到的结果如图10.1所示。结果显示，堆分配的大小以 $T[] \rightarrow \text{List} < T > \rightarrow \text{IEnumerable} < T >$ 的顺序增加。

因此，在使用LINQ时，通过了解运行时类型，可以减少GC.Alloc的大小。

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
RunAsArray	4.210 us	0.2735 us	0.0150 us	1.00	0.00	48 B
RunAsList	4.942 us	0.2517 us	0.0138 us	1.17	0.00	72 B
RunAsIEnumerable	7.326 us	3.4885 us	0.1912 us	1.74	0.04	96 B

图10.1 各类型执行速度的比较

LINQ中 GC.Alloc的原因

使用LINQ引起的GC.Alloc的部分原因是LINQ的内部实现：许多LINQ方法采取 $\text{IEnumerable} < T >$ 并返回 $\text{IEnumerable} < T >$ ，这种API设计使得使用方法链的直观性达到了这种API设计允许使用方法链进行直观的描述。LINQ内部实例化了一个实现 $\text{Enumerable} < T >$ 的类，然后调用 $\text{GetEnumerator}()$ 来实现循环处理。由于对 $\text{GetEnumerator}()$ 的调用，内部发生了Alloc。

10.5.2 LINQ延迟评估

LINQ方法，如Where和Select是懒人评估，它将评估推迟到实际需要的结果。另一方面，像ToArray这样的方法被定义为即时评估。

现在考虑下面清单10.24中代码的情况。

▼清单10.24。 方法与即时评价穿插进行

```
1: 私有的静止无效的LazyExpression()
2: {
3:     var array = Enumerable.Range(0, 5).ToArray();
4:     var sw = Stopwatch.StartNew();
5:     var query = array.Where(i => i % 2 == 0).Select(HeavyProcess).ToArray();
6:     Console.WriteLine($"Query: {sw.ElapsedMilliseconds}").
7:
8:     foreach (var i in query)
9:     {
10:         Console.WriteLine($"diff: {sw.ElapsedMilliseconds}").
11:     }
12: }
13:
14: 私人静态int HeavyProcess(int x) 15 :
{
16:     Thread.Sleep(1000);
17:     返回x。
18: }
```

清单10.25是执行清单10.24的结果。通过在最后添加一个ToArray，它被立即评估，执行Where和Select方法并评估值的结果在对查询进行赋值时被返回。因此，HeavyProcess也被调用，所以可以看出，处理时间是在生成查询的时间点上进行的。

▼清单10.25。 增加即时评估方法的结果

```
1: 查询: 3013
2: diff: 3032
3: diff: 3032
4: diff: 3032
```

因此，无意中调用LINQ的即时评估方法会导致这些点的瓶颈：需要一次性查看整个序列的方法，如ToArray、OrderBy和Count，都是即时评估，所以在调用它们时要注意成本，并且使用。

10.5.3 选择 "避免使用LINQ"。

解释了使用LINQ时GC.Alloc的原因，如何缓解以及延迟评估的关键点。本节解释了使用LINQ的标准。前提是，LINQ是一个有用的语言特性，但使用时，堆分配和执行速度比不使用时更差。事实上，微软的Unity Performance

第十章 调音实践--脚本（C#）。

建议^{*4}规定了“避免使用LINQ”；清单10.26比较了有无LINQ的同一逻辑实现的基准测试

。

▼ 清单10.26 使用和不使用 LINQ 的性能比较

```
1: 私有的int[] array。
2:
3: public void GlobalSetup().
4: {
5:     array = Enumerable.Range(0, 100_000_000).ToArray()。
6: }
7:
8: public void Pure().
9: {
10:    foreach (var i in array)
11:    {
12:        如果(i % 2 == 0)
13:        {
14:            var _ = i * i;
15:        }
16:    }
17: }
18:
19: public void UseLinq().
20: {
21:     var query = array.Where(i => i % 2 == 0).Select(i => i * i)。
22:     foreach (var i in query)
23:     {
24:     }
25: }
```

结果显示在图10.2中。执行时间的比较表明，与没有LINQ的情况相比可以看出，使用LINQ的过程花费了19倍的时间。

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated
Pure	26.06 ms	3.230 ms	0.177 ms	1.00	0.00	26 B
UseLinq	514.55 ms	354.586 ms	19.436 ms	19.75	0.79	920 B

图10.2 有和没有LINQ的性能比较结果

尽管上述结果清楚地表明，使用LINQ会带来性能上的损失。在某些情况下，如使用LINQ时，更容易沟通编码意图。这一点。

^{*4} <https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-optimization-best-practices>

统一的建议#避免昂贵的操作

10.6 避免了异步/等待的开销

基于对LINQ和LINQ行为的理解，是否使用LINQ的规则，或者使用LINQ的规则，可以在项目内讨论。

10.6 避免了异步/等待的开销

Async/await是C# 5.0中增加的一个语言特性，它允许将异步进程写得如同没有回调的直接同步进程一样。

10.6.1 在不必要的情况下避免使用async。

定义了async的方法将由编译器生成代码以实现异步处理。而有了async关键字，编译器的代码生成总是被执行。因此，即使是可能同步完成的方法，如清单10.27中的方法，实际上也是在进行编译器代码生成。

▼ 清单 10.27 可能同步完成的异步进程

```
1 : 使用System.Threading.Tasks.Methods。
2 : 使用System.Threading.Tasks.Methods。
3:
4 : 命名空间A {
5:     公用类B {
6:         public async Task HogeAsync(int i) {
7:             如果 (i == 0) {
8:                 Console.WriteLine("i是0")。
9:                 回归。
10:            }
11:            await Task.Delay(TimeSpan.FromSeconds(1))。
12:        }
13:
14:        public void Main() {
15:            int i = int.Parse(Console.ReadLine())。
16:            Task.Run(() => HogeAsync(i))。
17:        }
18:    }
19: }
```

在像清单10.27这样的情况下，为IAsyncStateMachine的实现生成一个状态机结构的代价，在同步完成时不需要，可以通过分割可能同步终止的HogeAsync并像清单10.28那样实现它而省略。

第十章 调音实践--脚本（C#）。

▼清单10.28。 同步和异步处理的分割实施

```
2 : 使用System.Threading.Tasks。  
3:  
4 : 命名空间A {  
5:     公用类B {  
6:         public async Task HogeAsync(int i) {  
7:             await Task.Delay(TimeSpan.FromSeconds(1));  
8:         }  
9:  
10:        public void Main() {  
11:            int i = int.Parse(Console.ReadLine());  
12:            如果 (i == 0) {  
13:                Console.WriteLine("i是0");  
14:            } else {  
15:                Task.Run(() => HogeAsync(i));  
16:            }  
17:        }  
18:    }  
19: }
```

Async/await如何工作

async/await语法是在编译时使用编译器代码生成来实现的：带有**async**关键字的方法添加了一个进程，该进程在编译时生成了一个实现**IAsync StateMachine**的结构，当**await**目标进程完成后异步/等待功能是通过管理推进状态的状态机实现的。此外，这个**IAsyncStateMachine**是一个定义在**System.Runtime.CompilerServices**命名空间的接口，只对编译器可用。

10.6.2 避免捕获同步上下文

从已经保存到另一个线程的异步处理中返回到调用线程的机制是同步上下文，而之前的上下文可以通过使用**await**来捕获。捕获这个同步上下文是在每个等待中完成的，这就产生了每个等待的开销。由于这个原因，它被广泛用于Unity开发中

对于Unity，实现不使用ExecutionContext和SynchronisationContext以避免捕获同步上下文的开销。在某些情况下可以看到性能的改善。

10.7 通过stackalloc进行优化

数组通常是在堆区分配的，所以将数组作为局部变量分配会导致GC.Alloc每次都发生，这可能会导致尖峰。另外，对堆区的读写比对堆区的读写效率低一些。

由于这个原因，C#提供了一种在堆栈上分配数组的语法，仅限于不安全代码。

不像清单10.29中那样使用new关键字，而是使用stackalloc关键字来在堆栈上分配一个数组。

▼ 清单 10.29 使用 stackalloc 在堆栈中分配一个数组

```
1: // stackalloc是不安全的。  
2: 不安全  
3: {  
4:     // 在堆栈中分配一个ints数组。  
5:     byte* buffer = stackalloc byte[BufferSize];  
6: }
```

从C# 7.2开始，Span<T>结构允许在没有不安全的情况下使用stackalloc，如清单10.30中所示。

▼ 清单 10.30。 使用Span<T>结构一起在堆栈上分配一个数组

```
1: Span<byte> buffer = stackalloc byte[BufferSize];
```

对于Unity，它从2021.2开始作为标准配置提供。对于早期版本，Span<T>并不存在，必须安装System.Memory.dll。

用stackalloc分配的数组只在堆栈中使用，不能在类或结构域中持有。它们必须始终作为局部变量使用。

分配有大量元素的数组需要一定的处理时间，即使它们被分配在堆栈上。如果在应该避免的地方需要进行堆分配，比如在更新循环内部，可能需要使用堆分配方法。

*5 <https://tech.cygames.co.jp/archives/3417/>

第十章 调音实践--脚本（C#）。

如果你想使用有大量质数的数组，最好在初始化时提前分配，或者准备一个数据结构，如对象池，并以使用时可以出租的方式来实现它。另外，需要注意的是，由stackalloc分配的堆栈区域在函数退出前不会被释放。

必须谨慎行事。例如，清单10.31中的代码表明，在循环中分配的数组是在退出Hoge方法时，所有的都被保持和释放，这可能会在循环时引起堆栈溢出。

▼ 清单10.31。 使用stackalloc在堆栈中分配一个数组

```
1: 不安全的无效Hoge()
2: {
3:     for (int i = 0; i < 10000; i++) 4:
4:     {
5:         // 循环次数的数组累积。
6:         byte* buffer = stackalloc byte[10000];
7:     }
8: }
```

10.8 通过密封优化IL2CPP后端下的方法调用

在Unity中使用IL2CPP作为后端进行构建时，方法调用是使用类似C++的vtable机制来实现类的虚拟方法调用^{*6}。

具体来说，对于一个类的每个方法调用定义，都会自动生成代码，如清单10.32所示。

▼ 清单 10.32 由IL2CPP生成的方法调用的C++代码

```
1: 结构 VirtActionInvoker0
2: {
3:     typedef void (*Action)(void*, const RuntimeMethod*); 4:
5:     static inline void Invoke (
6:         Il2CppMethodSlot slot, RuntimeObject* obj)
7:     {
8:         const VirtualInvokeData& invokeData =
9:             il2cpp_codegen_get_virtual_invoke_data(slot, obj);
10:        ((Action)invokeData.methodPtr)(obj, invokeData.method);
11:    }
12: }
```

^{*6} <https://blog.unity.com/technology/il2cpp-internals-method-calls>

10.8 通过密封优化IL2CPP后端下的方法调用

这不仅为病毒性方法生成了类似的C++代码，也为那些在编译时没有继承的、非虚拟的方法生成了类似的代码。这种自动生成的行为导致代码大小和方法调用的处理时间增加。

这个问题可以通过在类的定义中添加密封的修饰符来避免⁷。

如果你定义了一个像清单10.33那样的类，并调用了一个方法，由IL2CPP生成的C++代码将导致一个像清单10.34那样的方法调用。

▼ 清单 10.33 没有密封的类定义和方法调用

```
1: 公共抽象类动物 2: {  
3:     公共抽象的字符串Speak(); 4: }  
5:  
6: 公共类牛 : 动物 7: {  
8:     public override string Speak() {  
9:         返回 "Moo".  
10:    }  
11: }  
12:  
13: var cow = new Cow();  
14: //调用说话方法  
15: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼ 清单 10.34。与清单10.33中的方法调用相对应的C++代码

```
1: // var cow = new Cow(); 2:  
Cow_t1312235562 * L_14 =  
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(  
4:         Cow_t1312235562_il2cpp_TypeInfo_var);  
5: Cow ctor_m2285919473(L_14, /*隐藏的参数*/NULL); 6: V_4 =  
L_14。  
7: Cow_t1312235562 * L_16 = V_4。  
8:  
9: // cow.Speak()  
10: String_t* L_17 = VirtFuncInvoker0< String_t* >::Invoke(  
11:     4 /* System.String AssemblyCSharp.Cow::Speak() */, L_16);
```

清单10.34显示VirtualFuncInvoker0<String_t*>::Invoke被调用，尽管它不是一个虚拟方法的调用，证实了一个类似虚拟方法的方法调用正在进行中下面的例子显示了一个虚拟方法的调用。

⁷ <https://blog.unity.com/technology/il2cpp-optimizations-devirtualization>

第十章 调音实践--脚本（C#）。

另一方面，如清单10.35所示，在清单10.33中用**sealed**修饰符定义Cow类，生成的C++代码如清单10.36所示。

▼ 清单 10.35 使用 密封的类定义和方法调用

```
1: public sealed class Cow : Animal
2: {
3:     public override string Speak() {
4:         返回 "Moo"。
5:     }
6: }
7:
8: var cow = new Cow();
9: //调用说话方法
10: Debug.LogFormat("The cow says '{0}'", cow.Speak());
```

▼ 清单 10.36。与清单10.35中的方法调用相对应的C++代码

```
1: // var cow = new Cow(); 2:
Cow_t1312235562 * L_14 =
3:     (Cow_t1312235562 *)il2cpp_codegen_object_new(
4:         Cow_t1312235562_il2cpp_TypeInfo_var);
5: Cow ctor_m2285919473(L_14, /*隐藏的参数*/NULL); 6: V_4 =
L_14。
7: Cow_t1312235562 * L_16 = V_4。
8:
9: // cow.Speak()
10: String_t* L_17 = Cow_Speak_m1607867742(L_16, /*隐藏的参数*/NULL);
```

因此，可以看出，该方法调用了Cow_Speak_m1607867742，直接调用了该方法。

然而，Unity官方表示，在相对较新的Unity中，这种优化是部分自动的⁸。

这意味着，即使没有明确指定密封，这种优化也可以自动进行。

然而，正如论坛中提到的”[il2cpp]在Unity 2018.3中'sealed'不再像说的那样工作了吗？”⁸，截至2019年4月，这个实现并不完美。并非如此。

鉴于目前的情况，建议检查由IL2CPP生成的代码，并决定每个项目的密封修改器设置

◦

⁸ <https://forum.unity.com/threads/il2cpp-is-sealed-not-worked-as-said-anymore-in-unity-2018-3.659017/#post-4412785>

为了更可靠地直接调用方法，并期待未来IL2CPP的优化，将sealed修改器设置为可优化的标记可能是一个好主意。

10.9 通过内联进行优化

方法调用有一定的成本。因此，不仅在C#中，而且作为一种普遍的优化，相对较小的方法调用会通过编译器或其他方式进行内联优化。

具体来说，对于清单10.37这样的代码，内联允许清单10.38代码的生成如10.38所示。

▼清单10.37。 内联前的代码

```
1: int F(int a, int b, int c) 2:  
{  
3:     var d = Add(a, b);  
4:     var e = Add(b, c);  
5:     var f = Add(d, e);  
6:  
7:     返回f。  
8: }  
9:  
10: int Add(int a, int b) => a + b。
```

▼清单10.38。 列表10.37的内嵌代码

```
1: int F(int a, int b, int c) 2:  
{  
3:     var d = a + b;  
4:     var e = b + c;  
5:     var f = d + e;  
6:  
7:     返回f。  
8: }
```

内嵌是通过复制和扩展方法中的内容来完成的，如清单10.38，以及清单10.37中Func方法中对Add方法的调用。

在IL2CPP中，在代码生成过程中没有进行特别的内联优化。然而，从Unity 2020.2开始，可以为一个方法指定MethodImpl属性并指定其参数MethodOptions.AggressiveInlining到生成的C++代码。现在，内联指定器被赋予了相应的函数这意味着现在可以在C++代码水平上进行内联。

第十章 调音实践--脚本（C#）。

内联的好处是，它不仅降低了方法调用的成本，而且还节省了在方法调用时指定的参数的复制。

例如，算术方法可用于相对较大的尺寸，如Vector3和Matrix。

多个结构被作为参数。如果结构作为参数被如实传递，它们都会作为传递值被复制并传递给方法，所以如果参数的数量或传递结构的大小很大，方法调用和参数复制会产生相当大的处理成本。此外，方法调用也会成为处理负荷不可忽视的情况，因为它们经常被用于周期性过程，如物理操作和动画的实现。

在这种情况下，通过内联的方式进行优化可能会很有用。事实上，在**Unity**的新算术库**Unity.Mathematics**中，`MethodOptions.AggressiveInlining`被指定用于每个方法^{调用*9。}

另一方面，内联有一个缺点，即由于方法中的过程的扩展，代码大小会增加。

因此，建议考虑内联，特别是对于那些经常在一个框架内被调用的方法和热路径。还应该注意的是，指定一个属性并不总是导致内联。

被内联的方法仅限于那些内容较小的方法，所以想要被内联的方法需要保持其处理量小。

此外，在**Unity 2020.2**和更早的版本中，属性规范没有内联指定器，而C++并不保证通过指定内联指定器的

因此，如果你想确保内联，你也可以考虑对热路径的方法进行手动内联，尽管这将降低可读性。

*9 <https://github.com/Unity-Technologies/mathematics/blob/main/Runtime/MethodImplOptions.AggressiveInlining.md>

Technologies/Unity.Mathematics/blob/f476dc88954697f71e5615b5f57462495bc973a7/src/Unity.Mathematics/math.cs#L1894



统一

性能调整

CHAPTER

第11章。

调音练习 - 播放器设置

本章介绍了项目设置中影响性能的播放器项目。

11.1 脚本后端

Unity允许你在Android和Standalone（Windows、macOS和Linux）等平台上选择Mono或IL2CPP作为脚本后端。建议选择IL2CPP，因为它能提高性能，如第二章“基础知识”中所述，IL2CPP。

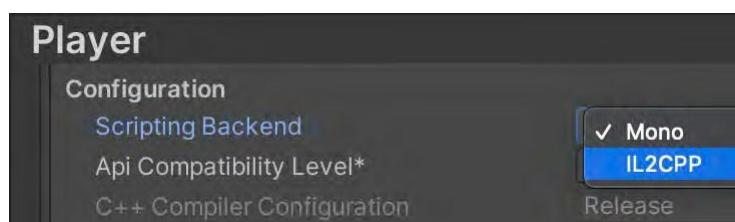
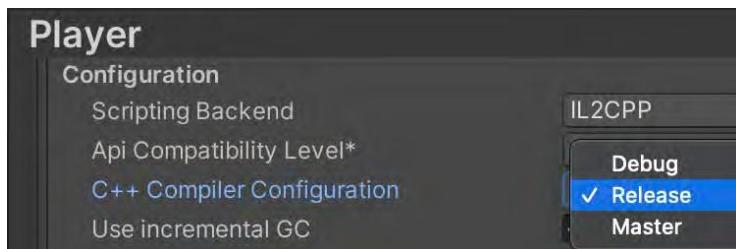


图11.1 脚本后端配置。

另外，如果将脚本后端改为IL2CPP，除了在某些平台上之外
可以选择**C++编译器配置**。

11.2 剥离发动机代码/管理剥离水平



▲图11.2 C++编译器配置设置。

在这里，你可以选择Debug、Release和Master，每一种都在构建时间和优化程度之间有自己的权衡，所以最好根据你的构建目标来使用它们。

11.1.1 调试

由于没有进行优化，它在运行时表现不佳，但与其他设置相比，构建时间是最短的。

11.1.2 发布

优化提高了运行时的性能，构建的二进制文件的大小更小，但构建的时间会增加。

11.1.3 掌握

该平台可用的所有优化功能都已启用。例如，Windows的构建使用更积极的优化，如使用链接时间代码生成（LTCG）。这样做的好处是，与Release设置相比，构建时间会进一步增加，但如果可以接受的话，Unity建议使用Master设置进行生产构建。

11.2 剥离发动机代码/管理剥离水平

Strip Engine Code和**Managed Stripping Level**有望通过分别从Unity功能和编译C#时产生的CIL字节码中移除未使用的代码来减少构建的二进制文件的大小。

然而，确定一个给定的代码是否被使用强烈依赖于静态分析

第11章 调音练习--演奏者设置

这可能导致意外删除代码中没有直接引用的类型，或在反射中动态调用的代码。

在这种情况下，在**link.xml**文件中或通过指定Preserve属性将其删除。
以是些可以避免的最重要的问题。¹

11.3 加速器频率 (iOS)

一个iOS特有的设置允许你改变加速度计的采样频率。默认设置是60赫兹，所以要适当地设置频率。特别是，如果你不使用加速度计，一定要禁用该设置。

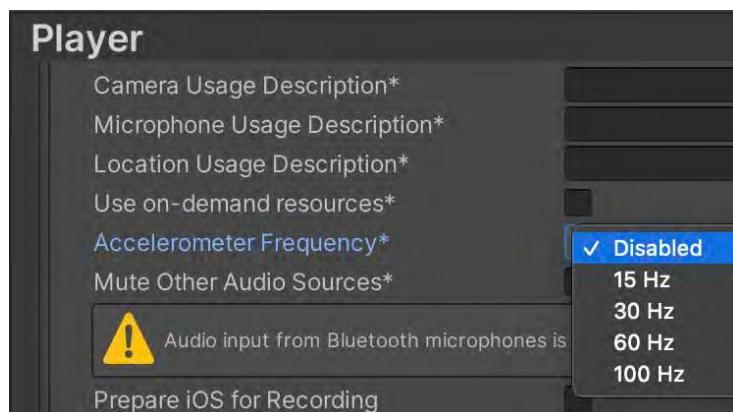


图11.3 采样频率设置。

¹ <https://docs.unity3d.com/2020.3/Documentation/Manual/ManagedCodeStripping.html>



统一

性能调整

CHAPTER

第12章。

调试实践 - 第三方

本章概述了在实施第三方库时从性能的角度应该注意的事项，这些库在Unity中开发游戏时经常使用。

12.1 DOTween

DOTween^{*1}是一个允许脚本实现平滑动画的库。例如，一个放大和缩小的动画可以很容易地写成以下代码。

▼清单12.1。 DOTween使用实例

```
1: public class Example : MonoBehaviour {  
2:     public void Play() {  
3:         DOTween.Sequence().  
4:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))  
5:             .Append(transform.DOScale(Vector3.one, 0.125f)).  
6:     }  
7: }
```

12.1.1 设置自动杀戮

DOTween.Sequence(), transform.DOScale(...) 等，创建Tween的过程基本上涉及到内存分配，所以考虑为经常回放的动画重复使用实例。

默认情况下，当动画完成时，Tween会被自动销毁，所以 SetAutoKill(false)抑制了这一点。第一个用例可以用以下代码代替

*1 <http://dotween.demigiant.com/index.php>

▼ 清单12.2重用 Tween实例。

```

1:     private Tween _tween;
2:
3:     私有的Awake() {
4:         _tween = DOTween.Sequence()
5:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
6:             .Append(transform.DOScale(Vector3.one, 0 . 125f))
7:             .SetAutoKill(false).
8:             .Pause().
9:     }
10:
11:    public void Play() {
12:        _tween.Restart().
13:    }

```

请注意，如果一个调用SetAutoKill(false)的Tween没有被明确地销毁，它将会泄漏。最好是在不再需要时调用Kill()，或者使用下面描述的**SetLink**。

▼ 清单12.3 明确抛弃一个 Tween

```

1:     private void OnDestroy() {
2:         _tween.Kill().
3:     }

```

12.1.2 设置链接

调用**SetAutoKill(false)**的Tweens和用**SetLoops(-1)**设置Tween无限期重复的Tweens将不会被自动销毁，所以你需要自己管理它们的寿命。你可能希望用**SetLink(gameObject)**将这样的Tween链接到其相关的GameObject上，这样，一旦GameObject被销毁，Tween就会被销毁。

▼ 清单 12.4在游戏对象的生命周期内赢得一个Tween的机会

```

1:     私有的Awake() {
2:         _tween = DOTween.Sequence()
3:             .Append(transform.DOScale(Vector3.one * 1.5f, 0.25f))
4:             .Append(transform.DOScale(Vector3.one, 0 . 125f))
5:             .SetAutoKill(false).
6:             .SetLink(gameObject)
7:             .Pause().
8:     }

```

第12章 调试实践 - 第三方

12.1.3 DOTween检查员。

你可以通过在Unity编辑器的播放过程中选择名为 [DOTween] 的游戏对象，从检查器中检查DOTween的状态和设置。

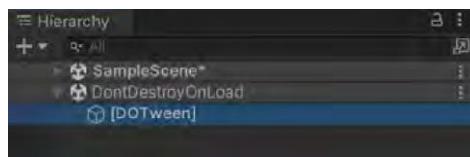


图12.1 [DOTween] GameObject。



图12.2 DOTween检查器。

当调查一个Tween是否仍然在运行，即使相关的GameObject已经被销毁，或者一个Tween是否处于暂停状态并在没有被销毁的情况下泄漏时，这可能是有用的。

12.2 统一制药公司（UniRx

UniRx^{*2}是一个实现统一优化的反应式扩展的库。一套丰富的操作符和Unity特有的辅助工具使得复杂条件的事件处理可以被简明地描述。

12.2.1 取消订阅。

UniRx允许你订阅一个流发布者的IObservable，以接收其消息的通知。

在这个订阅过程中，用于接收通知的对象实例、用于处理消息的回调等被创建。为了避免这些实例在订阅者的生命周期结束后仍留在内存中，当订阅者不再需要接收通知时，基本上是由他们负责退订的。

有几种取消订阅的方法，但出于性能考虑，Subscr
最好是保留ibe的IDisposable返回值，明确地进行处置。

```

1: public class Example : MonoBehaviour {
2:     private IDisposable _disposable; 3:
4:     私有的Awake() {
5:         _disposable = Observable.EveryUpdate()
6:             .订阅(_ => {
7:                 // 每一帧要进行的处理。
8:             });
9:     }
10:
11:    private void OnDestroy() {
12:        _disposable.Dispose()
13:    }
14: }
```

如果该类继承自MonoBehaviour，当它被Destroyed时，也可以通过调用AddTo(this)自动释放，尽管内部调用AddComponent来监控Destroy会有开销。这也是一个很好的选择，因为它写起来比较简单。

^{*2} <https://github.com/neuecc/UniRx>

```
1:     private void Awake() {
2:         Observable.EveryUpdate()
3:             .subscribe_ => {
4:                 // 每一帧要执行的过程。
5:             })
6:             .AddTo(this);
7: }
```

12.3 UniTask。

UniTask是一个强大的库，用于Unity中的高性能异步处理，具有零分配异步处理和基于值的UniTask类型。它还实现了与Unity的PlayerLoop一致的执行时间控制，从而完全取代了传统的coroutines。

12.3.1 UniTask v2.

UniTask v2是UniTask的主要升级版，于2020年6月发布，性能有了明显的提高，如整个异步方法的零分配，以及额外的功能，如异步LINQ支持和外部资产的等待支持。UniTask v2已经更新，包括异步LINQ和对外部资产的等待支持等功能。³

另一方面，UniTask.Delay(...).⁴)和其他由工厂返回的任务在调用的时候被调用，而从UniTask v1更新时应该小心，因为它包括一些破坏性的变化，如禁止⁴对正常的UniTask实例进行多重等待。然而，积极的优化进一步提高了性能，所以基本上UniTask v2才是正确的选择。

12.3.2 UniTask Tracker。

UniTask Tracker可以用来可视化等待的UniTask和它们创建的堆栈跟踪。

³ <https://tech.cygames.co.jp/archives/3417/>

⁴ UniTask.Preserve可以用来转换为一个可以等待多次的UniTask。

12.3 UniTask。



图12.3 UniTask Tracker。

例如，假设你有一个MonoBehaviour，当你与某些东西碰撞时，_hp会减少1。

```
1: public class Example : MonoBehaviour {
2:     private int _hp = 10;
3:
4:     public UniTask WaitForDeadAsync() {
5:         return UniTask.WaitUntil(() => _hp <= 0);
6:     }
7:
8:     private void OnCollisionEnter(Collision collision) {
9:         _hp -= 1;
10:    }
11: }
```

如果在这个MonoBehaviour的_hp完全减少之前Destroyed，WaitForDeadAsync的返回值中的UniTask就失去了完成的机会，因为_hp不会再减少，它将继续永远等待。

建议使用这个工具来检查是否有泄漏的UniTask，例如那些有不正确设置的退出条件的UniTask。

防止任务泄密

在示例代码中，任务泄漏是因为在满足其自身的退出条件之前，任务不可能发生泄漏。

其原因是没有考虑到Destroy的情况。

要做到这一点，只需检查它是否已被销毁。另外，从this.GetCancellationTokenOnDestroy()获得的CancellationToken也可以传递给WaitForDeadAsync，这样任务就会在Destroy时被取消了。

第12章 调试实践 - 第三方

```
1: // 检查自身是否被销毁的模式。
2: public UniTask WaitForDeadAsync() {
3:     return UniTask.WaitUntil(() => this == null || _hp <= 0);
4: }
5:
6: //传递CancellationToken的模式。
7: public UniTask WaitForDeadAsync(CancellationToken token) {
8:     return UniTask.WaitUntil(
9:         () => _hp <= 0,
10:         cancellationToken: token) 。
11: }
```

▼清单12.9。 WaitForDeadAsync(CancellationToken) 示例调用

```
1: 例子 例子=...
2: var token = example.GetCancellationTokenOnDestroy();
3: await example.WaitForDeadAsync (token) 。
```

在销毁时，前一个UniTask顺利完成，而后者则抛出一个OperationCancelledException。哪种行为更受欢迎取决于情况。
系统的实施会因实施类型的不同而不同，所以建议选择一个合适的实施方式。

到头来

本书到此结束。我们希望通过这本书，那些说自己对性能调优没有信心的人能够说，“我有点明白了，我想试试”。随着越来越多的人在他们的项目中实践它，他们将能够更快地处理问题，他们的项目的稳定性也会增加。

你也可能遇到用本书介绍的信息无法解决的复杂事件。但即使在这种情况下，你要做的事情也是一样的。那就是进行剖析，分析原因，并采取相应的措施。

从这一点出发，你可以通过实践充分运用自己的知识、经验和想象力。我们希望你喜欢用这种方式进行性能调整。谢谢你读到最后。

作者。

本节介绍了参与本出版物的作者。请注意，每位作者的简介和他们所负责的章节在撰写时是最新的。

饭田拓哉。

SGE核心技术部，兼任SGE核心技术部/Grange公司项目经理，第一章：“性能调优入门”，第三章：“分析到”。

他负责撰写了《儒》等书。我目前正在参与各子公司的优化工作。我在工作中做了各种各样的事情，但我每天都努力工作，以提高开发速度和质量。

Haruki Yano / Twitter: @harumak_11 / GitHub: Haruma-K

CyberAgent公司SGE核心技术部/客户端工程师 负责编写第二章：基础知识，包括“2.2 渲染”和“2.3数据表示方法”。我工作的主要重点是开发共同的基础设施以提高开发效率。他为Unity开发和发布各种OSS，包括商业和个人使用，同时也是Unity博客LIGHT11 的作者。

运作。

石黑祐介

SGE核心技术部，CyberAgent公司。

他撰写了第2章“基础知识”和第5章“调整实践--资产捆绑”的部分内容，并被分配到Ameba Games（现在的QualArts）的基础设施开发团队担任Unity工程师，参与了各种基础设施的开发，包括实时基础设施、聊天基础设施、资产捆绑管理基础设施“Octo”、认证和计费基础设施。从事各种基础设施的开发，如实时基础设施、聊天基础设施、资产捆绑管理平台“Octo”、认证和计费基础设施。目前，他已被调到SGE核心技术部，领导整体基础设施建设，并专注于优化整个游戏部门的开发效率和质量。

大木八幡

SGE核心技术部， CyberAgent公司。

写了第九章：调谐实践--脚本（Unity） 。在Grange公司和Gecrest公司从事游戏开发和运营。目前属于SGE的核心技术部门，正在开发基础设施。

附录作者。

Mitsutoshi Nakamura (NAKAMURO.) / Twitter: @megalo_23

隶属于Applibot公司/游戏创作者

负责编写2.5《C#基础》的前半部分，第10章，调音练习--脚本（C#）。通过写这本书，他计划减少在开发阶段结束时被叫去帮忙的机会，并有更多时间开发新游戏。他在游戏开发方面的活动包括优化、指导、音乐创作和配音。就个人而言，他正在Famulite实验室运行一个应用程序。

Shunsuke Ohba / Twitter: @ohbashunsuke

附属于Samsup公司/工程经理

他负责撰写第四章：调控实践--资产。前设计师-工程师，在使用Flash创建互动网站后，在职业生涯中期加入CyberAgent公司。开发完AmebaPig后，转到Unity工程。作为工程师领导，参与了许多游戏的推出，包括麻将、弹球和实时战斗。在个人方面，他在Twitter和他的博客Shibuya Hotogisu Tsushin 上提供信息。

石井格库

与Samsup公司合作/服务器和客户端工程师

他负责撰写第11章：调音实践--演奏者设置和第12章：调音实践--第三方。在被分配到Samsup公司后，他作为Unity工程师从事新游戏应用的开发工作。在参与了几个应用程序的发布后，他转而从事服务器端工程。目前在Samsup担任服务器端工程师，在SGE核心技术部担任Unity工程师，都是在服务器/客户端。

Shunsuke Saito / Twitter: @shun_shun_mummy

属于Colourful Palette Inc / 客户端工程师

她负责撰写第10章《调谐实践--脚本（C#）》的部分内容。在被分配到Colourful Palette Inc后，他在所负责的项目中围绕UI系统进行了客户端实时通信和开发的设计和实施。他还致力于调整现有的功能和开发自动生成模板代码的工具。

田村和则

属于Qualarts公司。

他负责撰写第8章：“调校实践--用户界面”。在QualArts公司担任Unity工程师，从事游戏开发和内部基础设施开发。对通过人工智能提高游戏开发的效率感兴趣，目前正努力在游戏部门内利用人工智能。

Tomoya Yamaguchi / Twitter: @togucchi

属于Colourful Palette Inc / 客户端工程师

她负责撰写第7章：“调整实践--图形”。在Colourful Palette Inc. 从事3D绘图和现场相关系统的开发工作。目前正在新的3D相关技术的验证工作。

向井雄一郎 / Twitter: @yucchiy_.

与Applibot公司的关系/客户端工程师

负责编写第6章“调谐实践-物理”和第10章“调谐实践-脚本（C#）”的部分内容。

统一的性能调谐圣经。

2022年7月8日 第一版，第一次印刷 已出版

作者 : CyberAgent公司SGE核心技术部

设计公司 CyberAgent SGE总部 PR设计办公室 出版办公室 CyberAgent公司



统

P1EIR. f'OR. MA. nce
TUN.