

Unity®



性能调谐圣经

撰写人：.

CyberAgent智能手机游戏和娱乐部门

SGE核心技术团队

网络代理。

Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント | ゲーム・エンターテイメント事業部SGEコア技術本部 著



性能。 调音圣经》。

撰写人：.

CyberAgent智能手机游戏和。
娱乐部SGE核心技术团队

网络代理。

Unityパフォーマンスチューニングバイブル

株式会社サイバーエージェント ゲーム・エンターテイメント事業部SGEコア技術本部 著

介绍。

本文件的目的是在Unity应用程序的性能调整方面遇到问题时作为参考。

我们觉得性能调优是一个可以利用过去的技术诀窍的领域，因此它是一个很容易成为一个老年性领域的领域。在这个领域没有经验的人可能会有这样的印象：这有些困难。其中一个原因可能是导致性能下降的原因差异很大。

然而，性能调整工作流程是可以成型的。遵循这一流程，就能更容易地确定原因，只寻找适合该事件的解决方案。在寻找解决方案时，知识和经验会有所帮助。因此，本手册主要是为了帮助你学习“工作流程”和“经验知识”。

它计划作为一份内部文件来制作，但我们希望许多人能够花时间看一看，刷一刷。我们希望它对那些看过的人有一些帮助。

关于此书

本手册假设内容是针对智能手机的应用。请注意，一些解释可能不适用于其他平台。除非另有说明，本文件中使用的Unity版本是Unity 2020.3.24f1。

本出版物的结构

本手册分为三个主要部分。前两章专门论述了调整到第三章涵盖了第三章的基础知识，第三章关于各种测量工具的使用，第四章起关于各种调谐实践。由于各章是独立的，你可以根据自己的水平只阅读必要的章节。以下各节对各章进行了概述。

第1章，“性能调优入门”，描述了性能调优。

本节介绍了调查的工作流程。它首先描述了开始前的准备工作，然后解释了如何隔离原因并进行调查。目的是，通过阅读本章，你将准备好开始性能调优。

第2章“基础知识”描述了你应该知道的关于硬件、绘图流程、Unity机制等性能调优的基本知识。当你觉得自己的知识不足时，不妨阅读第二章起的内容。

在第三章“剖析工具”中，你可以学习如何使用用于因果调查的各种工具。建议在第一次使用测量工具时将其作为参考。

第四章起，“调整实践”，包含了一系列的实践，从资产到脚本。这里描述的许多内容都可以立即在现场使用，所以请通读。

GitHub

该出版物有一个储存库^{*1}。将根据需要进行补充和更正。另外，PR和问题以指出更正和建议补充的内容。如果你愿意，请使用它。

免责声明。

本出版物中的信息仅供参考。因此，使用本出版物进行的任何开发、生产或操作都必须由您自己承担风险和判断。本公司对基于这些信息的任何开发、生产或运营的结果不承担责任。

^{*1} <https://github.com/CyberAgentGameEntertainment/UnityPerformanceTuningBible/>

目录

介绍。	ii.
关于本出版物.....	ii
本出版物的结构.....	ii
GitHub.....	iii.
免责声明	iii.

第1章 性能调优的入门	1
--------------------	----------

1.1 初步准备工作.....	2
1.1.1 确定指标.....	3
1.1.2 了解最大的内存使用量。.....	5
1.1.3 决定哪些设备可以保证工作。	7
1.1.4 界定质量设定规范。	8
1.2 防患于未然	8
1.3 在性能调整方面的工作.....	9
1.3.1 准备工作.....	9
1.3.2 性能下降的类型.....	10
1.4 隔离内存超限的原因.....	11
1.4.1 内存泄漏。	11
1.4.2 内存使用量简直太高了。	12
1.5 让我们来调查一下内存泄漏的问题。	12
1.6 让我们减少记忆。	13
1.6.1 资产	13
1.6.2 燃气管道（单声道）	15
1.6.3 其他	15
1.6.4 插件	16

1.6.5 检查规格	17
------------------	----

1.7	隔离处理失败的原因	17
1.8	调查瞬时负荷	18
1.8.1	通过 GC加注	18
1.8.2	重型加工导致的尖峰	19
1.9	调查稳态负荷	19
1.9.	1CPU绑定	20
1.9.	.2GPU绑定	20
1.10	摘要	22
第二章	基础知识	23
2.1	硬件	24
2.1.1	1SoC	24
2.1.2	iPhone、Android和SoC	25
2.1.3	3CPU	26
2.1.4	4GPU	30
2.1.5	内存	33
2.1.6	存储	37
2.2	渲染	40
2.2.1	渲染管线	41
2.2.2	半透明绘图和过度绘图	43
2.2.3	绘制调用设置-通过调用和批处理	44
2.3	数据表示	45
2.3.1	Bits and bytes	46
2.3.2	图像	47
2.3.3	压缩图像	48
2.3.4	网格	49
2.3.5	关键帧动画	51
2.4	统一是如何工作的	53
2.4.1	二进制文件和运行时间	53
2.4.2	资产实体	57
2.4.3	线路	58

2.4.4	游戏循环	60
2.4.5	GameObject	62
2.4.6	AssetBundle	64
2.5	C#基础知识。	67

2.5.1	堆栈和堆	68
2.5.2	Gavage Collection	68
2.5.3	结构 (struct)	70
2.6	算法和计算复杂性	73
2.6.1	关于计算复杂性	74
2.6.2	基本集合和数据结构	77
2.6.3	设备以降低计算复杂性	80
第三章：剖析工具		81
3.1	统一的程序	82
3.1.1	测量方法	84
3.1.2	CPU使用率	88
3.1.3	记忆	93
3.2	档案分析器	99
3.2.1	介绍	99
3.2.2	如何操作	100
3.2.3	分析结果（单模式）	101
3.2.4	分析结果（比较模式）	106
3.3	框架调试器	107
3.3.1	分析屏幕	108
3.3.2	详细屏幕	108
3.4	储存器	111
3.4.1	介绍的方法	112
3.4.2	如何操作	113
3.5	堆积浏览器	122

3.5.1	介绍。	122
3.5.2	操作说明	123
3.6	Xcode	127
3.6.1	简介方法	127
3.6.2	调试导航器	128
3.6.3	GPU帧捕获	132
3.6.4	记忆图表	141
3.7	器械	143
3.7.1	时代周刊	144
<hr/>		
3.7.2	拨款。	146
3.8	Android Studio	149
3.8.1	Profile方法	150
3.8.	2CPU测量	151
3.8.	3内存测量	153
3.9	RenderDoc	154
3.9.1	测量方法	155
3.9.2	如何查看采集数据	158
第四章	调试实践--资产	166
4.1	纹理	167
4.1.1	进口设置	167
4.1.2	读/写	168
4.1.3	生成Mip地图	169
4.1.4	阿尼索水平	169
4.1.5	压缩设置	170
4.2	网状物	171
4.2.1	启用了读/写功能	171
4.2.2	顶点压缩	172
4.2.3	网眼压缩	173

4.2.4	优化网格数据.....	174
4.3	材料。	175
4.4	动画	176
4.4.1	调整皮肤重量的数量.....	176
4.4.2	关键的削减。	177
4.4.3	减少了更新的频率.....	179
4.5	粒子系统	180
4.5.1	减少颗粒的数量.....	180
4.5.2	注意，噪音很重。	183
4.6	音频	184
4.6.1	负载类型	184
4.6.2	压缩格式	186
4.6.3	指定采样率.....	187
4.6.4	对于声音效果，设置为单声道。	187
4.7	资源 / 流媒体资产	188

目录

4.7.1 减缓启动时间 资源文件夹	189
4.8 ScriptableObject	189
第五章 调试实践--资产捆绑	191
。	
5.1 资产包的颗粒度	192
5.2 AssetBundle加载API。	192
5.3 资产包的卸载策略。	193
5.4 优化同时加载的AssetBundles的数量。	193
第六章 调音实践--物理学	195
。	
6.1 物理学开/关	196
6.2 优化固定时间步长和固定更新的频率	196
6.2. 1 允许的最大时间步长	197
6.3 碰撞几何的选择	198
6.4 碰撞矩阵和层优化	198
6.5 射影优化	199
6.5.1 Raycast类型	199
6.5.2 优化射线传输参数	200
6.5. 3RaycastAll和RaycastNonAlloc	200
6.6 对撞机和刚体	201
6.6. 1刚体和睡眠状态	202
6.7 优化碰撞检测	205
6.8 优化其他项目设置	206
6.8. 1Physics.autoSyncTransforms	206
6.8. 2Physics.reuseCollisionCallbacks	206
第七章 调音实践--图形	207
。	
7.1 调整分辨率	208
7.1. 1DPI设置	208
7.1.2 通过脚本设置分辨率	209
7.2 半透明性和过度拉伸.	210

7.3	减少全局调用	211
7.3.1	动态配料	211
7.3.2	静态配料	213
7.3.	3GPU实例化	214
7.3	.4SRP Batcher	217

7.4	SpriteAtlas	219
7.5	删减.....	223
7.5.1	视盘摘除术	223
7.5.2	后方剔除	223
7.5.3	遮挡剔除	224
7.6	着色器.....	226
7.6.1	降低浮点类型的精度	227
7.6.2	使用顶点着色器进行计算	227
7.6.3	预先填充纹理的信息	228
7.6.	4ShaderVariantCollection	228
7.7	写作.....	230
7.7.1	实时影子	230
7.7.2	光的映射	234
7.8	详细程度.....	238
7.9	纹理流.....	239
第8章	调试实践--用户界面	241
。	
8.1	分割画布	242
8.2	UnityWhite	243
8.3	布局组件	244
8.4	雷击目标.....	245
8.5	面具	245
8.6	TextMeshPro	246
8.7	切换用户界面显示	247
第9章	调音实践--脚本（统一）。	249
。	
9.1	空的Unity事件功能。	250
9.2	访问标签和名称	251
9.3	检索组件	252
9.4	访问转化.....	252
9.5	需要明确销毁的类.....	253
9.6	字符串规格	253
9.7	JsonUtility的陷阱。	254
9.8	渲染和MeshFilter的陷阱	255

9.9	删除日志输出代码	256
-----	--------------------	-----

目录

9.10 用Burst加快代码的速度。	257
9.10.1 用Burst加速代码	258
第10章 调音实践--脚本(C#)	261
。	
10.1 GC.分配案件和如何处理这些案件。	262
10.1.1 参考类型新	262
10.1.2 Lambda表达式	263
10.1.3 使用泛型来装箱的案例.	265
10.2 关于for/foreach	267
10.3 对象池	270
10.4 弦。	271
10.5 LINQ和延迟评估。	273
10.5.1 通过使用LINQ来缓解GC.Alloc的问题。	273
10.5.2 LINQ延迟评估	274
10.5.3 选项 ”避免使用LINQ”	275
10.6 避免了async/await的开销。	277
10.6.1 在不需要时避免使用async	277
10.6.2 避免同步的上下文捕获	278
10.7 通过stackalloc进行优化。	279
10.8 通过密封优化IL2CPP后端下的方法调用	280
10.9 通过内联进行优化。	283
第11章 调音练习 - 播放器设置	285
。	
11.1 脚本后端.	286
11.1.1 调试	287
11.1.2 发布	287
11.1.3 主站	287
11.2 剥离发动机代码/管理剥离水平	287
11.3 加速器频率 (iOS)	288
第12章 调试实践 - 第三方	289
。	
12.1 DOTween	290

12.1.1 SetAutoKill	290
12.1.2 SetLink	291
12.1.3 DOTween检查器	292
12.2 统一制药	293

12.2.1 取消订阅	293
12.3 统一任务	294
12.3.1 UniTask v2	294
12.3.2 统一任务跟踪器	294
 到头来	 297
 作者。	 298



性能调谐圣经

CHAPTER

01

第1章

パフォーマンス
チューニングを始めよう

CyberAgent Smartphone Games & Entertainment

第一章。

开始进行性能调优

本章介绍了性能调优所需的提前准备工作以及相关的工作流程。

首先，我们将介绍在开始性能调优之前，你需要决定和考虑什么。如果你的项目仍处于早期阶段，你肯定应该通读它。即使项目已经进展到一定程度，再次检查所描述的信息是否被考虑到也是一个好主意。下一节解释了如何解决正在经历性能下降的应用程序。通过学习如何隔离原因以及如何解决它，你将能够实施一系列的性能调整流程。

1.1 提前准备

在进行性能调优之前，要决定你想要实现的指标。这很容易用语言表达出来，但实际上是一项极具挑战性的任务。这是因为世界上充满了各种规格的设备，而拥有低规格设备的用户也不能被忽视。在这种情况下，有必要考虑各种事情，如游戏规格、目标用户群和是否将在海外开发游戏。这项工作不能由工程师单独完成。有必要与其他行业的人协商确定质量线，也有必要进行技术核查。

当没有足够的功能实施或资产来测量负载时，这些指标很难从初始阶段确定。因此，一种选择是在项目进展到一定程度后再决定。然而，重要的是要确保在项目进入大规模生产阶段之前做出决定。这是因为一旦开始大规模生产，改变的成本可能是巨大的。决定本节介绍的指标需要时间，但不要急于求成，要坚定地进行。

害怕大规模生产阶段后的规格变化。

现在已经过了大规模生产阶段，但在低规格设备上的渲染却出现了瓶颈。

假设你有一个项目。内存使用量已经接近极限，所以不能使用根据距离切换到低负载模型的方法。因此，我们决定减少模型中顶点的数量。

首先，你需要下一个新订单来减少数据。将需要一个新的采购订单。接下来，导演需要重新检查质量。最后，它还需要进行调试。这是一个简化的描述，但在实践中会有更详细的操作和调度安排。

上面可能有几十或几百个资产需要处理，即使是在大规模生产之后。这非常耗费时间和劳动力，对项目来说可能是致命的。

为了防止这种情况发生，创建了最苛刻的场景，其指标是事先核实它们是否得到履行是非常重要的。

1.1.1 确定指标

确定指标将设定目标，以实现目标。反之，如果没有指标，你将永远无法到达终点。

表1.1列出了应确定的指标。

▼ 表1.1 指标。

(数据)项	元素
帧率	你在任何时候都以什么帧率为目？
记忆	估计哪个屏幕有最大的内存，并确定其限制。
过渡时间	多长时间是适当的过渡期等待？
热度	在连续游戏的X个小时内，可以容忍多少热量？
电池	连续播放X小时，多少电池消耗是可以接受的？

在表1.1的指标中，帧率和内存特别重要，所以一定要决定这些指标。在这一点上，让我们把低规格的设备排除在外。首先要决定音量区设备的指标，这一点很重要。

容量区的定义由项目决定。可以通过对其他书目进行基准测试或进行市场调查来决定。另外，考虑到移动设备的长期更换，你可以用大约四年前的中档产品作为基准。设定一个目标的旗帜，即使理由有点模糊不清。从那里，你可以进行调整。

让我们考虑一个实际的例子。假设你有一个项目，现在有以下目标。

- 我们希望改善竞争性应用中的一切不足之处。
- 特别是在游戏中，你希望它能顺利地运行。
- 除上述情况外，应该与竞争对手差不多。

当团队将这个模糊的目标口头化时，出现了以下指标。

- 帧率
 - 就电池消耗而言，游戏内60帧，游戏外30帧
该框架应是。
- 记忆
 - 为了加快过渡时间，设计也应在游戏中保留一些游戏外的资源。最大使用量应是1GB。
- 过渡时间
 - 在游戏内和游戏外的过渡时间应力求与竞争者处于同一水平。在3秒的时间内。
 -
- 热度
 - 与竞争对手的水平相同。在经过验证的设备上连续1小时不加热。(未收费)。
- 电池
 - 与竞争对手的水平相同。在测试设备上连续使用1小时后，电池消耗约为20%。

如果你已经决定用这种方式瞄准一个指标，就用一个参考装置来触摸它。如果根本没有达到目标，就是一个很好的指标。

按游戏类型进行优化

在这种情况下，帧率被设置为60帧，因为主题是平滑地移动。对于节奏感强的动作游戏和判断力强的游戏，如第一人称射击游戏（FPS），高帧率也是可取的。然而，高帧率也有弊端。帧率越高，消耗的电池电量就越多。使用的内存越多，就越有可能在暂停时被操作系统杀死。考虑这些优势和劣势，决定每个游戏类型的适当目标。

1.1.2 了解最大的内存使用量。

本节重点讨论最大内存的使用。要知道最大的内存使用量，首先要确定支持的设备可以保留多少内存。基本上，最好是在保证工作的最低规格设备上进行验证。然而，由于内存分配机制可能根据操作系统的版本而改变，如果可能的话，建议准备多个不同主要版本的设备。另外，由于测量逻辑因测量工具的不同而不同，因此最好将所使用的工具限定为一种。

这里介绍一下作者在iOS上的验证，供参考。在验证项目中，Texture2D是在运行时生成的，并测量其崩溃的时间。代码如下。

▼ 列表 1.1 验证码。

```
1: private List<Texture2D> _textureList = new List<Texture2D>();  
2: ...  
3: public void CreateTexture(int size) {  
4:     Texture2D texture = new Texture2D(size, size, TextureFormat.RGBA32, false); 5:  
    _textureList.Add(texture);  
6: }
```

验证结果如图1.1所示。

端末	搭載メモリ (GB)	OS	メモリ使用量 (GB)
iPhone6	1	12.4.1	0.65
iPhone6S	2	10.0.1	1.37
		11.3	2.61
		12.1.2	1.37
		13.6	1.42
iPhone7	2	10.3.1	1.31
		11	2.64
		12.4	1.37
		13.3.1	1.42
iPhone7 Plus	3	12.0.1	2.00
iPhoneX	3	12.1	1.76
iPhoneXR	3	13.5.1	1.81

▲图1.1 碰撞阈值

验证环境使用Unity 2019.4.3和Xcode 11.6，以及Xcode的调试导航器的Memory部分的值。根据这一验证结果，可以说在拥有2GB板载内存的设备上，如iPhone 6S和7，最好将内存保持在1.3GB以内。还可以看到，当支持具有1GB板载内存的设备时，如iPhone6，内存使用限制要严格得多。iOS11的另一个特点是，内存的使用很突出，也许是由于不同的内存管理机制。在验证时，请注意操作系统之间的这种差异是罕见的。

由于图1.1中的验证环境有些陈旧，所以在撰写本文时，使用最新的环境重新测量了一些测量结果。我们使用了两个版本的Unity 2020.3.25和2021.2.0以及Xcode13.3.1，并在OS版本为14.6和15.4.1的iPhoneXR上构建。结果，测量值没有特别大的差异，所以我们认为数据仍然是可靠的。

内存测量工具

推荐使用Xcode和AndroidStudio等符合本地标准的工具作为测量工具。例如，在Unity Profiler测量中，由插件分配的本地内存区域被排除。在IL2CPP构建的情况下，IL2CPP元数据（约100MB）也不包括在测量中。另一方面，在本地工具Xcode的情况下，应用程序分配的所有内存都被测量。因此，最好使用一个符合本地标准的工具，以更准确地测量数值。

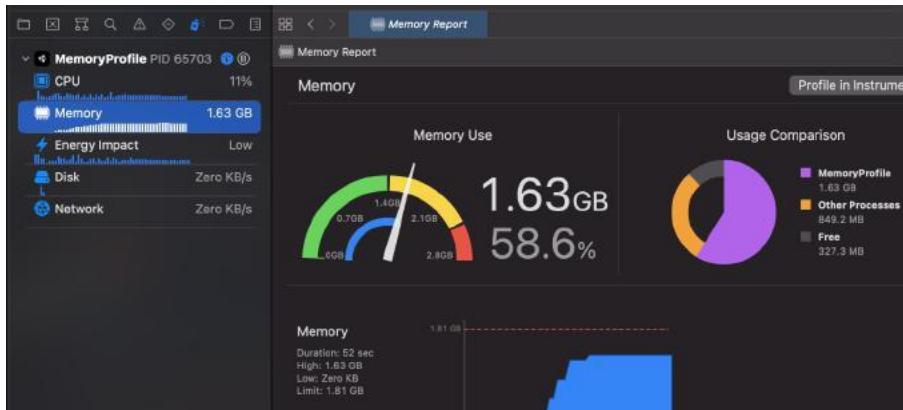


图1.2 Xcode调试导航器

1.1.3 决定哪些设备可以保证工作。

作为决定在性能调优方面走多远的指标，决定保证工作的最小设备数量也很重要。在没有经验的情况下，很难立即决定选择这种有保障的操作终端，但不要临时决定，而是从确定低规格终端的候选人开始。

作者推荐的方法是参考“SoC规格”的测量数据。具体来说，寻找网络上的基准测量应用程序所测量的数据。首先，掌握参考设备的规格，然后选择一个测量值比它低一些的设备。

选择几个图案作为终点。

一旦设备被识别，就可以实际安装和测试该应用程序。如果它运行缓慢，也不要气馁。现在你终于站在了起跑线上，你可以讨论要删除什么。下一节介绍了削减功能时应考虑的重要规格。

有几个基准测量应用程序，但作者使用Antutu作为标准。这里是测量数据的汇总网站存在的原因，志愿者也积极地报告测量数据。

1.1.4 界定质量设定规范。

由于市场上充斥着不同规格的设备，要用一种规格覆盖大量的设备可能很困难。因此，近年来，在游戏中设置几个质量设置，以保证在广泛的设备上稳定运行，已经成为普遍做法。

例如，以下项目可以分为高、中、低质量设置。

- 屏幕分辨率
- 显示的对象数量
- 阴影。
- 后期效果功能
- 帧率
- 能够跳过CPU密集型的脚本，等等。

然而，这将降低外观和感觉的质量，所以要与导演协商，共同探讨什么线路是项目可以接受的。

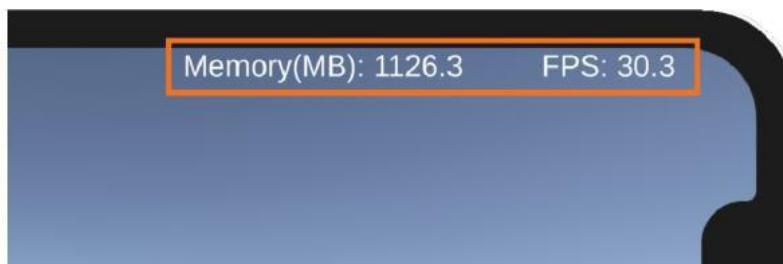
1.2 防止

性能下降，就像缺陷一样，随着时间的推移可能有许多原因，这使得调查更加困难。最好是在应用程序中实施一种机制，使其能够尽早被注意到。实现这一点的一个简单而有效的方法是在屏幕上显示当前的应用状态。至少有以下内容

1.3 参与性能调整

建议在任何时候都在屏幕上显示。

- 目前的帧率。
- 当前的内存使用情况



▲图1.3 性能可视化

虽然帧率可以通过经验来检测，以显示性能正在下降，但内存只能通过崩溃来检测。如图1.3所示，只要在屏幕上有一个恒定的显示，就会增加在早期阶段检测到内存泄漏的概率。

这种展示方法可以通过更多的创意而变得更加有效。例如，如果你想达到的目标帧率是30帧，你可能想把25-30帧显示为绿色，20-25帧显示为黄色，低于这个数值的显示为红色。这样，你可以直观地看到你的申请是否符合标准。

1.3 从事性能调整

无论如何努力在性能退化发生之前防止它，都很难完全防止它。这是不可避免的。性能下降是发展的一个不可分割的部分。总有一天，你将不得不面对性能调整。下面的章节解释了应该如何解决性能调优的问题。

1.3.1 准备工作

在你开始进行性能调优之前，这里有一些重要的第一步需要记住。比方说，你有一个帧率很慢的应用程序。很明显，显示了几个丰富的模型。你周围的人说，这些模型一定

是原因。情况真的是这样吗？证据在哪里？

你将需要仔细评估它。在准备进行性能调优时，有两件事你必须注意。

- 1 第二是测量和确定原因。不要猜测。
- 2 第二，总是比较修改后的结果。比较前后的资料是一个好主意。关键是要检查是否有全面的性能下降，而不仅仅是在固定区域。关于性能调整的可怕之处在于，在极少数情况下，被修改的部分可能会更快，但系统的其他部分的负载会增加，导致整体性能下降。这是一个真正的弊端。



▲图1.4 性能调整的准备工作

确保你能找出问题的原因，并确保你的速度越来越快。这是性能调整的一个重要心态。

1.3.2 性能下降的类型

每种情况都可能指的是不同类型的性能下降。在本出版物中，它们被广泛定义为以下三种类型(图1.5)

1.4 隔离内存超限的原因



图1.5 性能下降的原因

首先，崩溃可以分为两种主要类型：“内存超出”或“程序执行错误”。后者不是性能调优的范畴，在本文中没有详细介绍。

接下来，“CPU和GPU处理时间”可能是造成屏幕减速和加载时间长的主要原因。在下面的章节中，我们将重点讨论“内存”和“处理时间”，并深入研究性能下降问题。

1.4 隔离内存超限的原因

我们已经讨论了崩溃的可能原因是内存过多。现在我们将进一步分解记忆力过剩的原因。

1.4.1 内存泄漏。

内存超限的一个可能原因是内存泄漏。要检查这一点，请检查内存使用量是否随着场景的转换而逐渐增加。这里的场景转换不仅仅是屏幕转换，还包括大屏幕的变化。例如，从标题屏幕到游戏外，从游戏外到游戏内，等等。测量时，步骤如下。

1. 注意某些场景下的内存使用情况。
2. 过渡到另一个场景。
3. 重复步骤' 1'至' 2'三至五次。

如果测量的结果是内存使用量的净增长，那么肯定有什么东西在泄露。这可以说是一个无形的故障。首先，消除泄漏。

在进行' 2'的转换之前，在中间有几个屏幕转换也是一个好主意。因为也有可能只有在一屏幕上加载的资源被异常泄露。

这是因为它 是可能的。

如果你 确信有泄漏, 请调查泄漏的原因。关于具体调查方法的解释, 见1.5调查内存泄漏。

重复的原因。

根据作者的经验, 在释放资源后 (在UnloadUnusedAssets之后), 由于时间问题, 一些资源没有被释放。这些未释放的资源在过渡到下一个场景时被释放。相反, 在反复转换过程中, 内存使用量逐渐增加, 最终会导致崩溃。为了将前者的问题与后者区分开来, 本文件推荐了一种在内存测量过程中多次重复转换的方法。

顺便说一句, 如果出现前者这样的问题, 那么在资源释放时, 某些对象可能还持有一个引用, 并随后被释放。这不是致命的, 但调查原因和解决问题是一个好主意。

1.4.2 内存使用量简直太高了。

如果在没有泄漏的情况下, 内存使用率很高, 就有必要探索可以减少内存的领域。在1.6节 ”让我们减少内存” 中, 解释了具体的方法。

1.5 让我们来调查一下内存泄漏的问题。

首先, 重现内存泄漏, 然后使用下面描述的工具来找到原因。这里简要介绍一下这些工具的特点。关于如何使用这些工具的细节在第3章 ”剖析工具” 中有所涉及, 所以在调查时请参考这些工具。

储存器(记忆)

剖析器工具默认包含在Unity编辑器中。这使其很容易进行测量。基本上, 你会在设置了 ”详细” 和 ”收集对象引用”的情况下对内存进行快照并进行调查。与其他工具不同, 该

工具的测量数据不能与快照进行比较。使用。

1.6 让我们减少记忆。

关于如何做到这一点的更多信息，见3.1.3内存。

储存器

这必须从软件包管理器中安装。树状图以图形方式显示内存内容；它由Unity官方支持，目前仍在频繁更新；从0.5版本开始，跟踪引用关系的方法得到了极大的改进，所以建议使用最新版本。有关使用的详情，请参见3.4内存管理器。

堆栈资源管理器。

这个需要从软件包管理器中安装。这是一个由个人开发的工具，但它非常容易使用，而且很轻便。当v0.5版的内存编辑器不可用时，它可以作为v0.5版内存编辑器的替代品。关于如何使用它的细节，请参见3.5 堆浏览器。

1.6 让我们减少记忆。

减少记忆的关键是将其从大面积中移除。这是因为削减1,000个1KB的纹理只会导致1MB的减少。然而，如果一个10MB的纹理被压缩到2MB，它将被减少8MB。考虑成本效益，并注意你应该先从减少最大的项目开始。

在本节中，用于减少内存的工具将被讨论为Profiler(Memory)。如果你以前没有使用过，请参考“3.1.3 内存”以了解更多信息。

随后的章节将介绍在进行削减时需要注意的事项。

1.6.1 资产

如果在简单视图中有很多与资产有关的内容，可能会有不需要的资产或内存泄漏。这里的资产相关区域是指图1.6中的矩形所包围的区域。



▲图1.6 与资产有关的项目

在这种情况下，有三件事需要调查

无用资产调查

不必要的资产是指对当前场景完全不需要的资源。例如，只在标题画面中使用的背景音乐，即使在游戏外也会常驻内存。首先，只使用当前场景需要的东西。

重复的资产调查

这往往发生在提供资产捆绑支持的时候。由于资产捆绑依赖关系分离不力，同一资产被包含在多个资产捆绑中。然而，过多的分离依赖关系会导致下载文件数量和文件部署成本的增加。在测量这个区域时，可能有必要培养一种平衡感。关于资产捆绑的更多信息，见2.4.6 AssetBundle。

查阅相关规定。

审查每个项目，以确保遵守规定。如果没有任何规定，请检查一下，可能没有正确估计内存。

例如，对于纹理，你可以检查以下细节。

- 尺寸是否合适？
- 压缩设置是否合适？

1.6 减少内存!

- MipMap的配置是否正确？
- 适当的读/写设置，等等。

关于每项资产需要注意的事项，见第4章 “调校实践-- 资产”。

1.6.2 燃气管道（单声道）

如果在Simple View中有大量的GC (Mono)，很可能是一次发生了大量的GC.Alloc。另外，由于GC.Alloc每一帧都会发生，内存可能会被分割成碎片。这些可能导致管理堆区域过度扩张。在这种情况下，稳定地减少GC.Alloc。

参见 “2.1.5 内存”以了解更多关于管理堆的信息。同样地，GC.Alloc. 关于“堆栈和堆”的细节将在第2.5.1节 “堆栈和堆”中论述。

特定版本的符号差异

从2020.2开始，它被标记为 "GC"，而到2020.1及以下，它被描述为 "Mono"。
两者都是指管理堆的占用量。

1.6.3 其他。

检查 “详细视图”是否有任何可疑的项目。例如，打开其他部分并进行调查是一个好主意。

Name	Memory
▶ Assets (33256)	490.0 MB
▼ Other (792)	338.6 MB
System.ExecutableAndDLLs	208.0 MB
▶ SerializedFile (592)	64.4 MB
▶ Profiling (8)	32.5 MB
▶ PersistentManager.Remapper (1)	16.0 MB
▶ Rendering (9)	8.2 MB
▶ Managers (28)	7.9 MB
Objects	1.3 MB
▶ MemoryPools (16)	105.4 KB
▶ Physics2D Module (1)	104.1 KB
▶ File System (2)	79.7 KB
▶ Job System (1)	2.9 KB
▶ ParticleSystem Module (2)	2.3 KB
▶ Animation Module (4)	1.7 KB
▶ MemoryProfiling (1)	96 B

▲图1.7 其他项目。

根据作者的经验，SerializedFile和PersistentManager.Remapper是相当臃肿的案例。如果可以在几个项目之间进行数字比较，那么将它们进行一次比较是一个好主意。对比各自的数字可能会发现异常值。更多信息请见2.详细视图。

1.6.4 插件

到目前为止，Unity测量工具已经被用来隔离原因。然而，Unity只能测量Unity管理的内存。检查第三方产品是否在分配额外的内存。

具体来说，使用本地测量工具（Xcode中的仪器）。欲了解更多信息。
见第3.7节“工具”。

1.6.5 检查规格

这是最后的手段。如果到目前为止，没有可以削减的领域，就必须考虑规范。以下是一些例子。

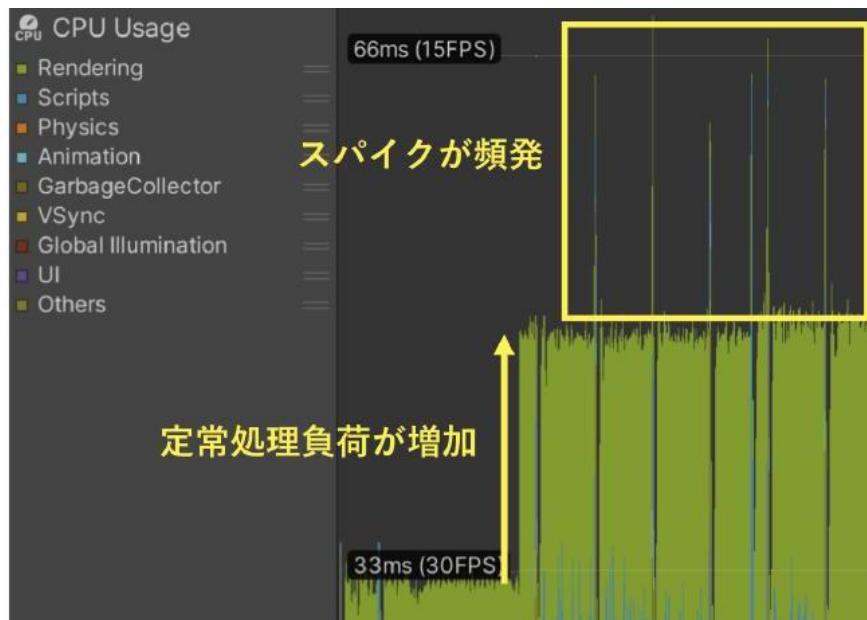
- 改变纹理的压缩率
 - 将纹理的某一部分的压缩率提高一个系数。
- 改变装货和卸货的时间。
 - 释放常驻内存中的对象，每次加载它们。
- 改变道路规格
 - 在游戏中少了一个需要加载的角色变体。

所有这些都有很大的影响范围，可以从根本上关系到游戏的乐趣。因此，规格考虑是最后的手段。为了避免这种情况，确保在早期估计和测量内存。

1.7 隔离处理失败的原因

本节介绍了测量和优化处理时间的过程。处理屏幕处理速度减慢的方法因其是“瞬间”还是“稳定”的处理速度减慢而不同。

瞬时加工量的下降是以针尖般的加工负荷来衡量的。由于其外观，它们也被称为尖峰。



▲图1.8 尖峰和稳态处理负荷

在图1.8中，测量数据显示了稳态负荷的突然增加，以及周期性的尖峰。这两个事件都需要进行性能调整。首先，将介绍相对简单的瞬时负荷研究。随后，将解释稳态负荷研究。

1.8 调查瞬时负荷。

Profiler (CPU) 被用来调查尖峰的原因。

关于如何使用该工具的详细信息，见 “3.1.2 CPU用量”。首先，分离出原因是否是由GC造成的。深度检测不需要隔离原因本身，但需要它来解决问题。

1.8.1 通过GC加注

如果发生了垃圾收集 (GC)，GC.Alloc必须减少。你可以使用 “深层定义” (Deep Profile) 来查看哪些进程正在分配多少资源。首先应该减少的领域是那些具有成本效益的领域。诸如以下各节

最好是在中心位置纠正眼睛。

- 每一帧分配的区域。
- 出现大量分配物的地区。

拨款越少越好，但这并不意味着拨款应该始终为零。例如，在生成过程中发生的分配（Instantiate）不能被阻止。关于GC的更多信息，见2.5.2 垃圾回收。

1.8.2 因繁重的处理而产生的尖峰

如果GC不是原因，那么某种繁重的处理正在瞬间进行。在这里，也可以使用“深度定义”（Deep Profile）来调查哪些处理是繁重的，以及处理的程度，并审查那些需要最长时间处理的部分。

常见的临时繁重加工可能包括以下内容。

- 实例化进程
- 主动切换大量对象或具有深层结构的对象
- 屏幕捕捉过程，等等。

由于这是一个相当依赖项目代码的部分，因此没有一个放之四海而皆准的解决方案。如果实际测量发现了原因，与项目成员分享测量结果，并讨论应该如何改进。

1.9 调查稳态负荷。

在提高稳定的处理负荷时，减少一帧内的处理是很重要的，这大致可分为CPU处理和GPU处理。首先，隔离这两种处理方式中的哪一种是瓶颈，或者它们是否具有相同的处理负荷，是很有用的。

| 在CPU上存在瓶颈的情况被称为CPU绑定，而在GPU上存在瓶颈的情况被称为GPU绑定。 |

一个简单的隔离方法是，如果以下任何一项适用于你，你很可能有一个GPU绑定。

- 当屏幕分辨率降低时，处理负荷会显著提高。
- 当Profiler测量时，**Gfx.WaitForPresent**是存在的。

反之，如果这些都不存在，就有可能出现CPU跳票。以下各节解释了如何调查CPU弹出和GPU弹出的情况。

1.9.1 CPU绑定

CPU Bound使用CPU（Profiler），这在上一节也有涉及，使用Deep Profile进行调查，以检查某一特定算法是否有很大的处理负荷。如果没有大的处理负荷，这意味着系统同样很重，所以要稳步改进。如果在进行稳步改进后仍未达到目标减排值，不妨回到“1.1.4 确定质量设置规格”，重新考虑。

1.9.2 GPU的约束。

对于绑定GPU的情况，帧调试器是一个很好的调查工具。关于如何使用它的细节，见3.3框架调试器。

该决议是否适当？

在GPU的边界中，分辨率对GPU的处理负荷有很大影响。因此，如果分辨率设置不当，首要任务是设置一个合适的分辨率。

首先，检查你是否有正确的分辨率和假定的质量设置。检查的一个好方法是在Frame Debugger中查看正在处理的渲染目标的分辨率。如果没有刻意执行以下内容，请努力优化它们。

- 只有UI元素是以设备的完整分辨率绘制的。
- 用于后期效果的临时纹理的高分辨率，如：。

是否有任何不需要的物品？

检查框架调试器是否有不必要的绘图。例如，可能有一个摄像机处于活动状态，但不需要，而且在幕后可能正在进行无关的绘画。如果有很多情况下，由于其他障碍物的存在，之前的绘图被浪费了，你可能也要考虑闭塞剔除的问题。关于闭塞剔除的更多信息，见7.5.3闭塞剔除。

请注意，闭塞剔除需要对数据进行预先准备，这将增加内存的使用，因为数据被部署在内存中。将事先准备好的信息建立在记忆中，以便通过这种方式提高性能，这是一种常见的技术。内存和性能往往是成反比的，所以在采用某种东西时，注意内存是一个好主意。

分批进行是否合适？

将对象画在一起被称为批处理。Batching对于GPU的弹跳是有效的，因为它可以提高绘制效率。例如，静态批处理可以用来将多个不动的物体的网格组合在一起。

由于有许多不同的配料方法，下面列出了一些典型的方法。如果你对其中任何一个感兴趣，请看‘7.3 减少抽奖’。

- 静态批处理
- 动态配料
- GPU实例化
- SRP Batcher, 等等。

单独看一下负载。

如果处理负荷仍然过高，你必须逐一查看。这可能是由于对象中的顶点太多，或者是由于Shader的处理。为了隔离这个问题，切换每个对象的活动状态，看看处理负荷如何变化。具体来说，试着删除背景，看看会发生什么，试着删除人物，看看会发生什么，

以此类推，把范围缩小到每个类别。哪个类别的处理负荷最高？

如果是这样，建议你进一步看看以下因素。

- 是否有太多的物体要画？
 - 考虑是否可以将它们拉到一起。
- 1 对象中的顶点数量是否太多？
 - 减少，考虑LOD。
- 用一个简单的Shader来代替Shader是否能改善处理负荷？
 - 审查着色器处理。

否则

每一个GPU过程都可以说是沉重的，堆积在一起的。在这种情况下，你需要稳定地添加一个我们一次只能改进一件事。

另外，在这里，和CPU绑定一样，如果没有完全达到目标减少，最好回到“1.1.4确定质量设置规格”，重新考虑一下。

1.10 摘要

这一章涵盖了在性能调优之前和期间需要注意的事项。

在进行性能调整之前需要注意的事项包括

- 确定“指标”、“保证操作终端”和“质量设置规范”。
 - 在大规模生产之前验证和确定指标。
- 应该建立一个系统，使其更容易注意到性能的下降。

在性能调整过程中需要注意的事项包括

- 隔离业绩不佳的原因并采取适当的行动。
- 必须执行“测量”、“改进”和“重新测量（检查结果）的顺序。

正如到目前为止所解释的那样，性能调整是关于测量和隔离原因。即使发生了本文档中没有描述的实例，如果遵循基本原则，也不太可能成为大问题。如果你以前从未做过性能调优，我们希望你能将本章的信息用于实践。



性能调谐圣经

CHAPTER

02

第2章

基礎知識

第二章。

基础知识

在进行性能调整时，必须对整个应用程序进行调查和修改。因此，有效的性能调整需要广泛的知识，从硬件到3D渲染和Unity力学。因此，本章总结了进行性能调优所需的基本知识。

2.1 硬件

计算机硬件由五个主要设备组成：输入设备、输出设备、存储设备、计算设备和控制设备。这些被称为计算机的五个主要设备。本节总结了这些硬件单元的基础知识，这对性能调整很重要。

2.1.1 索克（日本品牌氟哌啶醇）。

计算机由各种设备组成。典型的设备包括一个用于控制和运算的CPU，一个用于图形计算的GPU和一个用于处理音频和视频等数字数据的DSP。例如，在大多数台式电脑中，这些都是独立的，作为独立的集成电路，它们被组合起来形成计算机。相比之下，在智能手机中，这些设备是在单个芯片上实现的，以减少尺寸和功耗。这被称为片上系统，或SoC。

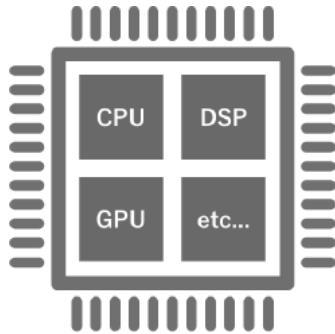


图2.1 SoC

2.1.2 iPhone-Android和SoC

智能手机根据其型号的不同，其板载的SoC也有所不同。

例如，iPhone使用的SoC称为A系列，是由苹果公司设计的。该系列使用字母“A”和数字的组合来命名，如A15，数字随每个版本的增加而增加。

相比之下，许多安卓设备使用的是一种名为Snapdragon的SoC。这是由一家名为高通的公司制造的SoC，被命名为类似骁龙8代或骁龙888。

此外，虽然iPhone是由苹果公司制造的，但安卓系统是由各种制造商制造的。这就是为什么安卓系统容易出现依赖模型的故障。

▼ 表2.1 主要的安卓系统芯片

系列名称	制造商	车载设备的趋势
骁龙	高通公司	用于广泛的终端
赫利奥。	联发科公司	在一些低成本的终端中使用
麒麟牌啤酒	喜瑞康公司。	华为生产的手机
兴发xf187在线娱乐 兴发xf187在线娱乐 兴发xf187在线娱乐	三星公司	三星公司生产的终端机

在进行性能调优时，重要的是要了解设备的SoC中使用的是什么，以及它有哪些规格。

迄今为止，Snapdragon的命名一直以“Snapdragon”字符串和一个三位数的数字为主。

这些数字是有意义的：800是旗舰机型，存在于所谓的高端手机中。从这里开始，性能和价格随着数字的减少而减少，400s是所谓的低端设备。

即使数字在400左右，也很难说，因为发布日期越新，性能就越好，但基本上，数字越高，性能就可以被视为越高。此外，2021年宣布，这种命名方式很快就会用完，所以未来将改成类似骁龙8代的命名方式。至于这样的命名规则，它是确定设备性能的一个指标，因此，它是对设备的一种保护。

这在性能调整时是很有用的，可以记住。

2.1.3 CPU

CPU（中央处理器）是计算机的大脑，它不仅执行程序，而且还与计算机的各种硬件组件进行协调。在实际调整性能时，了解CPU中进行的是什么处理，有什么样的特点，是很有用的，所以本节从性能角度进行解释。

CPU基础知识

决定程序执行速度的不仅仅是简单的算术能力，还包括复杂程序步骤的执行速度。例如，有些程序涉及四个算术运算，但也涉及分支：在程序执行之前，CPU不知道下一条指令将被调用。因此，CPU的硬件被设计为能够快速连续地处理各种指令。

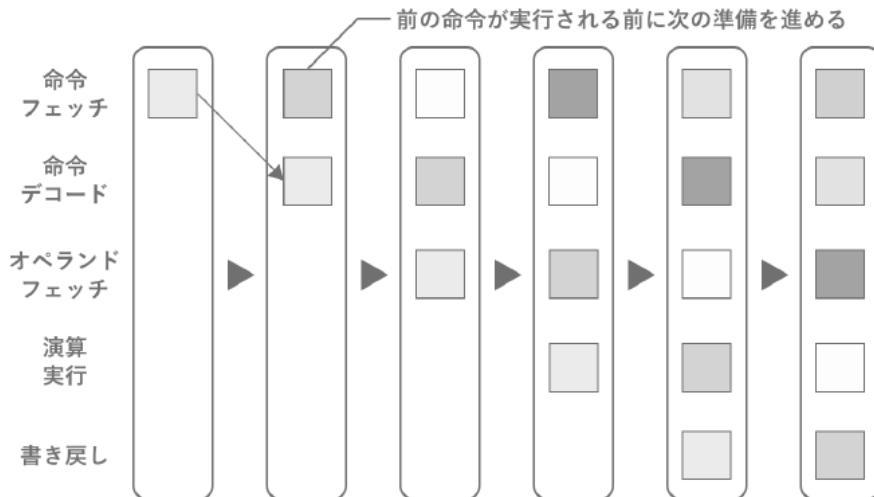
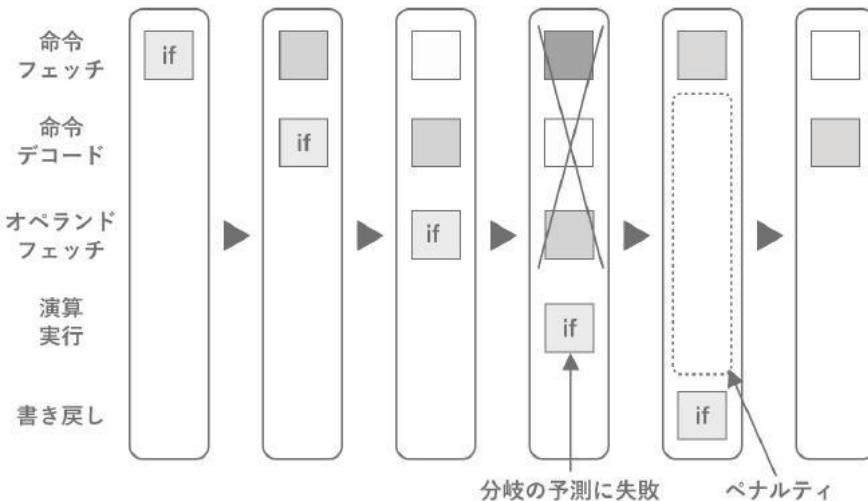


图2.2 CPU流水线结构

在CPU内部处理的指令流被称为流水线，在处理的同时预测流水线上的下一条指令。如果下一条指令没有被预测到，就会发生一个被称为流水线停滞的暂停，流水线被重置。大多数停滞是由分支引起的。分支机构本身也在一定程度上预计到了结果，但它仍然会犯错误。在不学习内部结构的情况下也可以进行性能调优，但只要知道这些东西就可以帮助你在编写代码时注意避免循环中的分支等问题。



▲ 图2.3 CPU流水线停顿

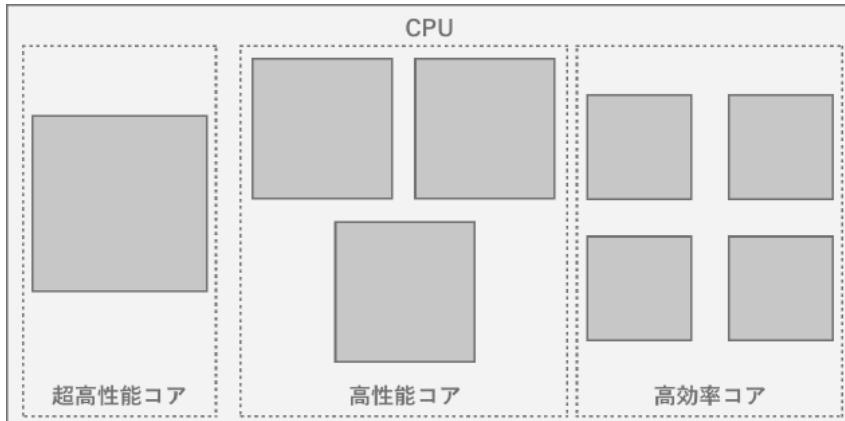
CPU计算能力

CPU的计算能力由时钟频率（单位：Hz）和内核数量决定。时钟频率表明CPU每秒钟可以运行多少次。因此，时钟频率越高，程序执行速度就越快。

另一方面，内核的数量有助于提高CPU的并行计算能力。核心是CPU运行的基本单位，如果有一个以上的核心，则称为多核心。最初只有单核，但在单核的情况下，为了运行多个程序，要交替运行的程序被切换。这被称为上下文切换，其成本非常高。如果你习惯于使用智能手机，你可能认为总是有一个应用程序（进程）在运行，但实际上有许多不同的进程在并行运行，包括操作系统。为了在这种情况下提供最佳的处理能力，具有多个核心的多核处理器已经成为主流。就智能手机而言，从2022年起，2-8个内核将成为主流。

具有不对称内核（big.LITTLE）的CPU在最近的多核中成为主流（尤其是智能手机）。非对称内核是指有一个高性能内核和一个省电内核在一起的CPU。非对称内核的优点是，通常只使用省电内核以节省电池消耗，而在需要性能的时候，如在游戏中，可以切换内核。然而，省电的内核并不像最具并行性能的内核那样高效。

应该注意的是，不能仅用核心数来确定，因为大值会减少。

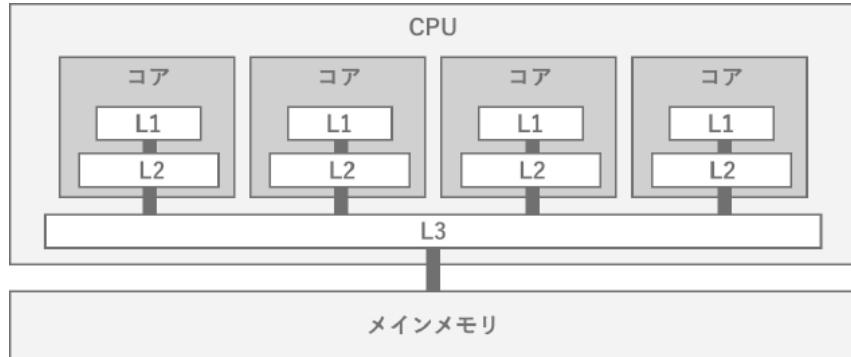


▲ 图2.4 骁龙8代1的异构核心配置

一个程序是否能用掉多个核心，也取决于程序的并行处理描述。例如，有些情况下，游戏引擎通过在单独的线程中运行物理引擎来精简物理引擎，或者通过Unity的JobSystem来利用并行处理，等等。然而，游戏主循环的运行本身不能被并行化，所以即使有多个内核，内核本身的高性能也是有利的。然而，游戏的主循环本身不能被并行化，因此，即使是多核的情况下，内核本身的高性能也是有优势的。

CPU缓存内存

CPU和主存储器在物理上相距甚远，只需要一小部分时间（延迟）来访问。因此，在执行程序时试图访问存储在主存储器中的数据时，这个距离是一个主要的性能瓶颈。为了解决这个延迟问题，在CPU内部安装了一个缓存存储器。缓存存储器主要是存储在主存储器中的部分数据，这样就可以快速访问程序所需的数据。缓存存储器有三种类型：L1、L2和L3缓存，数字越小表示速度越高但容量越小。L3高速缓存为2-4MB级别。因此，CPU缓存不能存储所有数据，只能处理最近的数据。



▲ 图2.5 CPU的L1、L2和L3缓存与主内存的关系。

提高程序性能的关键是如何有效地将数据放在高速缓存中，但由于高速缓存不能由程序自由控制，所以数据定位很重要。在游戏引擎中，在管理内存时很难意识到数据的位置性，但一些机制，如Unity的JobSystem，可以实现内存分配，增加数据的位置性。

2.1.4 GPU

CPU专门负责执行程序，而**GPU**（图形处理单元）是专门负责图像处理和图形渲染的硬件。

GPU基础知识

GPU被设计为专门从事图形处理，因此其结构与CPU非常不同，后者被设计为并行处理大量的简单计算。例如，如果要将一幅图像转换为黑白，基于CPU的计算必须从内存中读取某些坐标的RGB值，将其转换为灰度，然后逐个像素地返回到内存中。这样的过程不涉及任何分支，每个像素的计算也不依赖于其他像素的结果，所以很容易在每个像素上并行进行计算。

因此，GPU可以进行并行处理，如对大量数据应用相同的操作，速度很快，因此可以高速进行图形处理。特别是，图形系统需要大量的浮点运算，而GPU特别擅长浮点运算。每秒浮点运算次数，被称为**FLOPS**，是衡量一秒钟内可进行的运算次数的标准。

一般采用的是绩效指标。由于仅靠计算能力是很难理解的，所以还使用了一个叫做填充率的指标，它表示每秒可以绘制多少个像素。

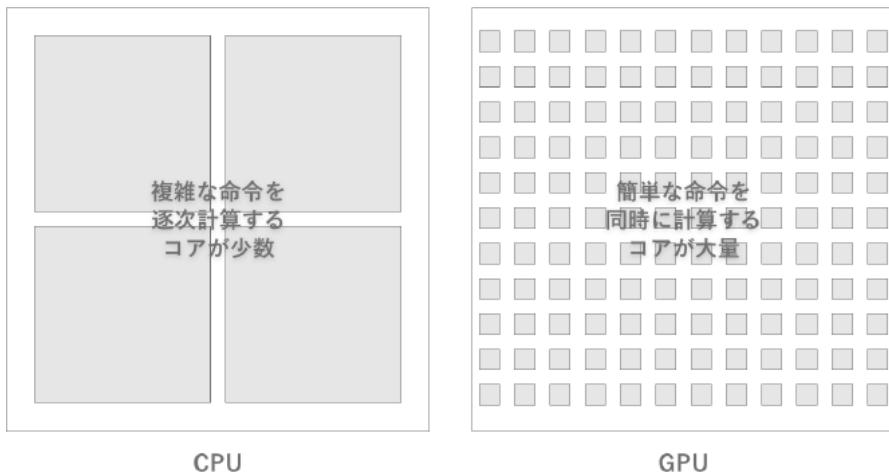


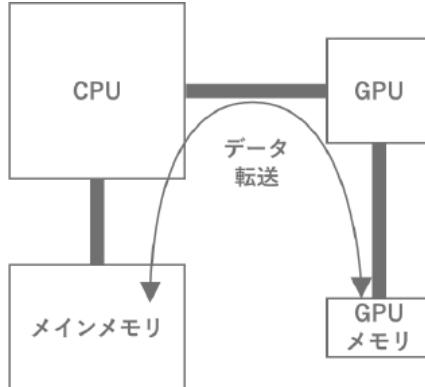
图2.6 CPU和GPU之间的差异

GPU计算能力

GPU的硬件特点是有大量的内核（几十到几千个），包含整数和浮点运算单元。为了部署大量的核心，运行复杂程序所需的单元已经被删除，因为不再需要CPU。另外，与CPU一样，其运行的时钟频率越高，每秒可执行的操作就越多。

GPU内存

当然，GPU也需要一个内存区域进行临时存储，以便处理数据。通常情况下，这个区域与主内存不同，它是专门用于GPU的。因此，为了进行任何形式的处理，数据必须从主内存转移到GPU的内存。处理后，数据以相反的顺序返回到主存储器。请注意，如果传输量很大，例如在传输多个高分辨率纹理时，传输需要时间并成为处理瓶颈。



▲ 图2.7 GPU内存传输

然而，在移动设备中，常见的架构是在CPU和GPU之间共享主内存，而不是为GPU配备专用内存。虽然这样做的好处是可以动态地改变GPU的内存容量，但它的缺点是CPU和GPU共享传输带宽。在这种情况下，数据仍然必须在CPU和GPU的存储区之间传输。

GPGPU

由于GPU可以对大量数据进行高速的并行运算，而这是CPU所不擅长的，因此近年来GPU被用于图形处理以外的用途，并被称为**GPGPU**（通用型GPU）。特别是，已经有许多案例表明，GPU被应用于人工智能等机器学习和区块链等计算处理，这导致了对GPU需求的快速增长，并产生了推动其价格上涨的效果。GPGPU也可以在Unity中通过利用一个叫做Compute Shader的功能来使用。

2.1.5 记忆

基本上所有的数据都保存在主存储器中，因为CPU当时只拥有计算所需的数据。由于不可能使用超过物理容量的内存，如果使用太多，内存无法分配，进程就会被操作系统强制终止。这通常被称为被**OOM**（Out Of Memory）杀死。2022目前，大多数智能手机的内存容量为4-8GB，但仍然必须注意不要使用过多的内存。

另外，如上所述，由于内存与CPU分离，性能本身会因是否进行了内存感知的实现而有所不同。本节解释了程序和内存之间的关系，以便可以进行性能感知的执行。

内存 - 硬件

虽然从物理距离上看，如果主存储器在SoC中是有利的，但存储器不包括在SoC中。这是有原因的，例如，如果内存被包含在SoC中，那么内存容量就不能在不同的设备之间改变。然而，如果主存储器很慢，就会明显影响程序的执行速度，所以要用相对较快的总线来连接SoC和存储器。智能手机中常用的内存和总线标准是LPDDR，它有几代产品，理论上的传输速率为几Gbps。当然，理论上的性能不可能总是实现，但在游戏开发中，这很少是一个瓶颈，所以没有必要意识到它。

内存和操作系统

在一个操作系统内，有许多进程同时运行，主要是系统进程和用户进程。系统进程是在运行操作系统中发挥重要作用的进程，以服务的形式驻留，无论用户的意图如何，它们中的大多数都会继续运行。另一方面，用户进程是按照用户的意愿启动的进程，对操作系统的运行来说并不是必不可少的。

智能手机上的应用有两种显示状态：前台（最前台）和后台（隐藏），一般来说，当某一应用处于前台时，其他应用就处于后台。当应用程序在后台时，进程以暂停状态存在，以方便返回过程，而内存则保持原状。但是，如果整个系统使用的内存用完了，就会根据操作系统确定的优先顺序杀死该进程。这时最可能被杀死的进程是

是指在后台使用大量内存的用户应用程序（~游戏）。这意味着使用大量内存的游戏在进入后台时更有可能被杀死，导致返回游戏时用户体验更差，不得不重新开始。

如果在它试图分配内存时没有其他进程可以被杀死，它将自己被杀死。在某些情况下，如iOS，它被控制，以便没有一个进程可以使用超过一定比例的物理内存。因此，首先可以分配的内存量是有限制的：在2022年，3GB内存的iOS设备的限制将是1.3-1.4GB左右，所以这可能是创建游戏时的上限。

内存互换

在现实中，有许多不同的硬件设备的物理内存容量非常小，操作系统试图以各种方式分配虚拟内存容量，以便在这类设备上运行尽可能多的进程。这就是内存互换。

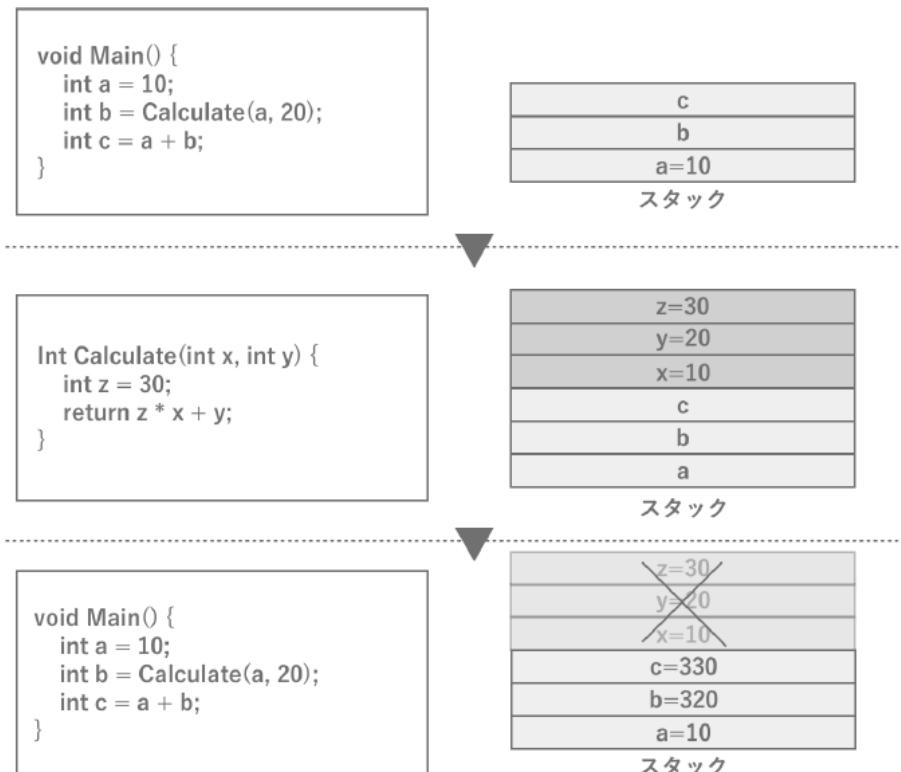
内存交换中使用的一种技术是内存压缩。它通过压缩和存储在内存中来节省物理空间，主要是在一段时间内未被访问的内存。然而，由于压缩和解压的成本，这不是针对那些积极使用的区域，而是针对那些已经进入后台的应用程序，例如。

另一种方法是将未使用的内存保存到存储空间：在存储空间充足的硬件上，如PC，没有必要终止进程以保留内存，而是将未使用的内存保存到存储空间，以释放物理内存。这样做的好处是比内存压缩能保证更大的内存量，但在智能手机中没有使用，因为存储比内存慢，所以有很强的性能限制，而且在智能手机中也不实用，因为智能手机的存储规模本来就小。

堆栈和堆

你可能至少听过一次堆栈和堆积这两个术语。堆栈实际上是一个专门的固定区域，与程序的运行有很深的关系。当一个函数被调用时，堆栈被分配给参数和局部变量，当函数返回到原函数时，堆栈被释放，返回值被累积。换句话说，当在一个函数中调用下一个函数时，当前函数的信息保持不变，下一个函数被加载到内存中。通过这种方式，函数调用机制得到了实现。堆栈内存在架构中。

这取决于，但由于容量本身非常小，只有1MB，所以只能存储有限的数据。



※厳密にはスタックは引数やローカル変数以外にも使われるがここでは省略

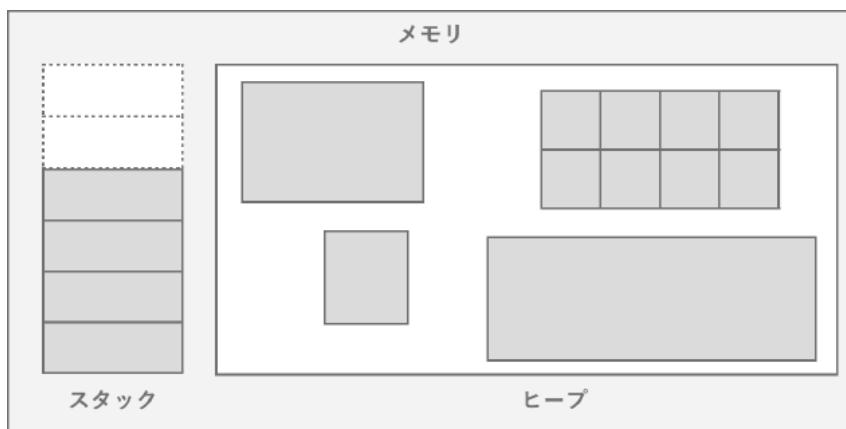
图2.8 堆栈操作示意图

另一方面，堆是一个可以在程序中自由使用的内存区域。每当程序需要时，它可以发出内存分配指令（C语言中的malloc），分配并使用大量的数据。当然，当你使用完后，你需要释放它（免费）。在C#中，内存分配和去分配是在运行时自动进行的，所以实现者不需要明确地做这个。

由于操作系统不知道何时需要以及需要多少内存容量，因此它从内存的空闲区域保证了它的安全，并在需要时传递给它。如果在尝试分配内存时，不能连续分配内存的大小，说明内存不足。这个关键词“连续”很重要。一般来说，如果内存分配和释放是重复

的，内存碎片

发生碎片化。当内存被分割时，有可能在整个总数中没有足够的自由空间，但在一个序列中没有足够的自由空间。在这种情况下，操作系统首先执行**堆的扩展**。换句话说，它分配了新的内存给进程，从而确保了连续的空间。然而，由于整个系统的内存是有限的，如果新分配的内存用完了，操作系统将杀死该进程。



▲ 图2.9 堆栈和堆

在比较堆栈和堆的时候，内存分配性能有一个明显的区别。这是因为一个函数所需的堆栈内存量是在编译时确定的，所以内存区域已经分配好了，而堆在执行前并不知道它需要多少内存，所以它每次都要搜索和分配空闲空间。这就是为什么堆的速度很慢，而栈的速度很快。

堆栈溢出错误。

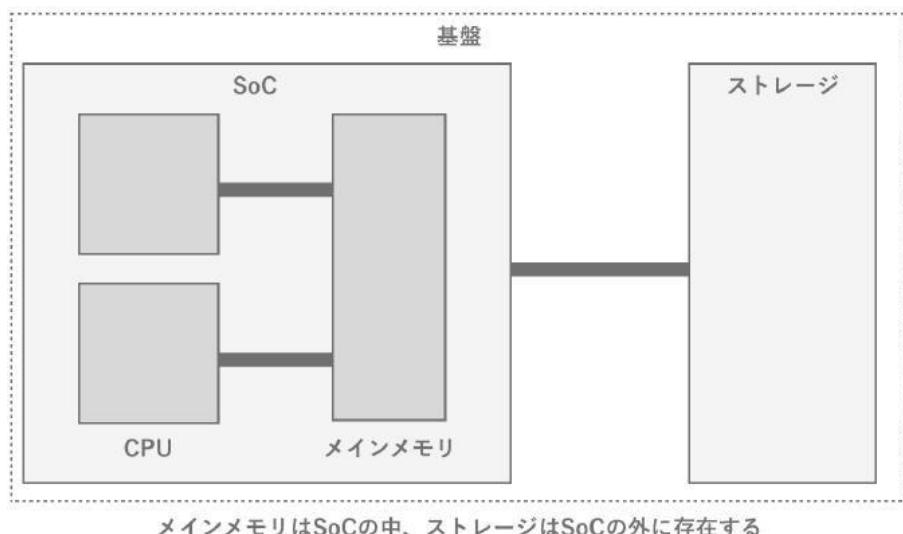
堆栈溢出错误是由于递归调用函数等导致堆栈内存被用完而发生的错误。

iOS/Android的默认堆栈大小是1MB，所以当递归调用的搜索大小增加时，它们更容易发生。一般来说，可以通过改用不导致递归调用的算法，或改用不使递归调用深入的算法来抵制这种错误。

2.1.6 储存

如果你实际经历了调整过程，你可能会注意到，在读取文件的情况下，往往需要很长的时间。读取文件是指从存储文件的仓库中读取数据，并将其写入内存，以便程序可以处理。了解那里实际发生的情况在调谐时很有用。

首先，一个典型的硬件架构有专门的存储用于数据持久性。存储的特点是容量大，并且能够在没有电源的情况下持久保存数据（非易失性）。利用这一特点，大量的资产以及应用程序本身的程序都存储在存储器中，并在应用程序启动时从存储器中加载和执行，例如。



▲ 图2.10 SoC和存储之间的关系

RAM和ROM

特别是在日本，人们通常把RAM写成智能手机内存，把ROM写成存储，但ROM实际上是指只读存储器。顾名思义，它应该是只读不写的，但在日本，这个词的使用似乎有强烈的习惯性。

然而，从多个角度来看，与程序执行周期相比，对该存储的读写过程是非常缓慢的。

- 由于与内存相比，与CPU的物理距离较远，因此延迟较高，读/写速度较慢。
- 有很多浪费，因为数据是逐块读取的，包括指令数据和它的周围环境。
- 序列读/写速度快，而随机读/写速度慢。

特别重要的是，这种随机读/写的速度很慢。首先，什么情况是顺序的，什么情况是随机的：当一个文件从头开始按顺序读/写时，是顺序的，但当读/写一个文件的多个部分或同时读/写几个小文件时，是随机的。随机的。需要注意的是，即使在同一目录下读/写多个文件，它们也不一定是按物理顺序定位的，所以如果它们在物理上相距甚远，就会出现随机现象。

从存储器中读取的过程

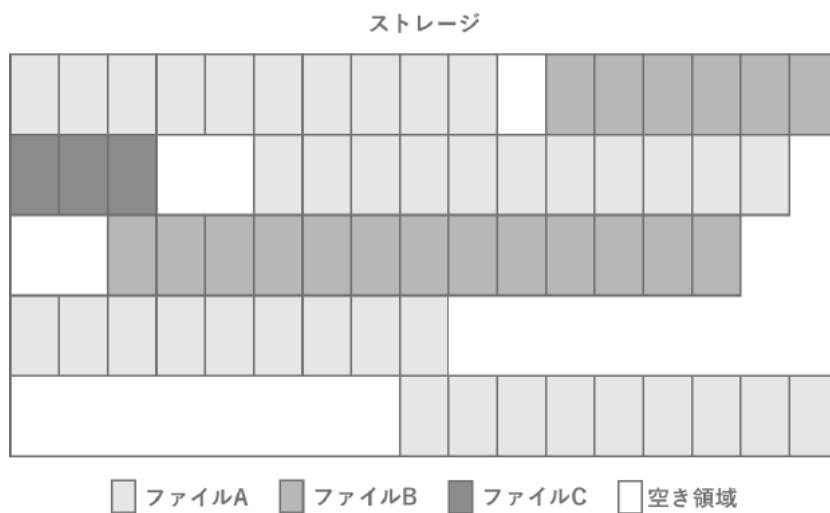
当读出存储的文件时，细节被省略了，但大致是该过程如下。

1. 程序命令存储控制器从存储中读取文件的哪个区域。
2. 存储控制器接收指令并计算出要用数据在物理上读取的区域。
3. 负载数据。
4. 将数据写进内存

5. 程序通过内存访问数据

也可能有更多的层，如控制器，这取决于硬件和架构。没有必要准确记住，但必须意识到，与从内存中读取相比，有更多的硬件处理步骤。

典型的存储也是通过将单个文件分块写入，例如4KB，来实现性能和空间效率。这些块在物理上并不总是连续的，即使它们是一个单一的文件。文件的物理分散被称为碎片，而消除碎片的操作被称为**碎片整理**。虽然碎片化对智能手机来说不是一个问题，但在考虑个人电脑时，它是需要注意的问题。



▲ 图2.11 存储碎片化。

个人电脑和智能手机中的存储类型

在PC领域，HDD和SSD是主流--你可能以前没有见过HDD，但它们是以磁盘形式记录的媒体，就像CD一样，磁头在磁盘上移动以读取磁性。因此，它们在结构上很庞大，而且由于涉及物理运动，它们也是一种具有高延迟的设备。近年来，固态硬盘开始流行，它与HDD不同，不产生物理运动，因此具有高速性能，但另一方面，它的读/写循环次数（寿命）是有限制的，因此，如果频繁的读/写循环发生，就会无法使用。~~智能手机与SSD的不同之处在于，它们使用一种被称为NAND的闪存。~~

最后，在智能手机中，存储的实际读/写速度有多快？如果你想读取一个10MB的文件，即使在理想条件下，你也需要100毫秒来读取整个文件。此外，如果要读取几个小文件，将出现随机访问，这将使读取时间更长。因此，意识到读取一个文件实际上需要惊人的时间总是好的。对于个别设备的具体性能，有一些网站^{*1}收集了基准测试结果，你可以参考一下。

综上所述，在读写文件时，应考虑以下几个方面的问题

- 存储器的读/写速度出奇的慢，预计不会像内存那样快。
- 尽可能减少同时读/写的文件数量（例如，错开时间，合并成一个文件）。

2.2 渲染

在游戏中，渲染的处理负荷常常对性能产生负面影响。因此，渲染方面的知识对于性能调优是至关重要的。因此，本节总结了渲染的基本原理。

^{*1} https://maxim-saplin.github.io/cpdt_results/

2.2.1 渲染管线

在计算机图形学中，对诸如三维模型的顶点坐标和灯光的坐标和颜色等数据进行一系列处理，以输出最终输出到屏幕上每个像素的颜色。这种处理机制被称为渲染管道。

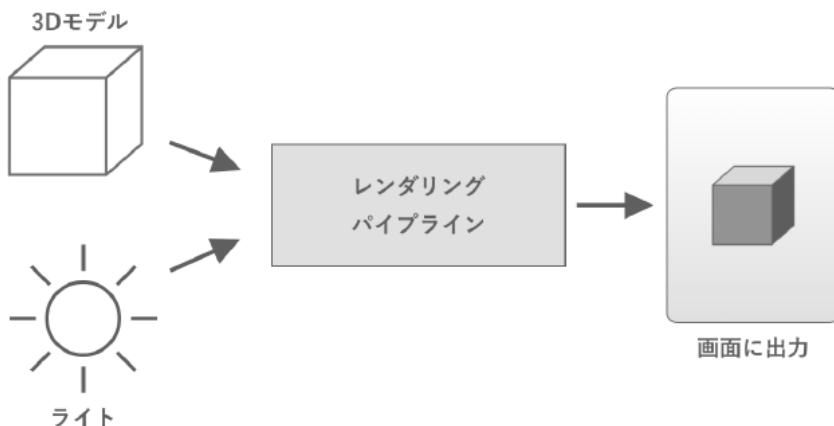
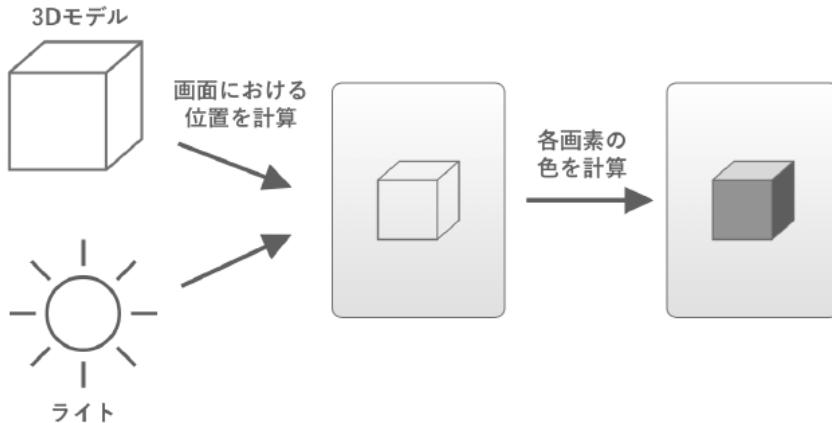


图2.12 渲染管道。

渲染管道开始时，将必要的数据从CPU发送到GPU。这包括要渲染的3D模型的顶点和灯光的坐标，以及关于物体的材料和相机的信息。

GPU对这些信息进行编译，并计算出物体在屏幕上的位置，当它被摄像机观看时。这个过程被称为坐标转换。

一旦确定了物体将在屏幕上显示的位置，就需要确定物体的颜色。然后，GPU会计算“当光线照亮该材料的模型时，屏幕上相应的像素会是什么颜色”。



▲图2.13 计算位置和颜色

在上述过程中，“物体将出现在屏幕的什么位置”由一个叫做顶点着色器的程序计算，而“屏幕上每个像素的相应部分将是什么颜色”则由一个叫做片段着色器的程序计算。而且这些着色器可以自由编写。因此，在顶点和片段着色器中写入重度处理会增加处理负荷。

此外，顶点着色器是按3D模型中的顶点数量来处理的，所以顶点越多，处理负荷就越大。要渲染的像素数量越多，碎片着色器的处理能力就越强。

实际的渲染管线。

实际渲染管道中的顶点着色器和片段着色器。

除了DAR，还有许多其他的过程，但本文旨在提供对性能调优所需概念的理解，因此将限于简化的描述。

2.2.2 半透明绘图和过度绘图

渲染时，有关物体的透明度是一个重要问题。例如，考虑两个物体，当从相机看时，它们部分重叠。

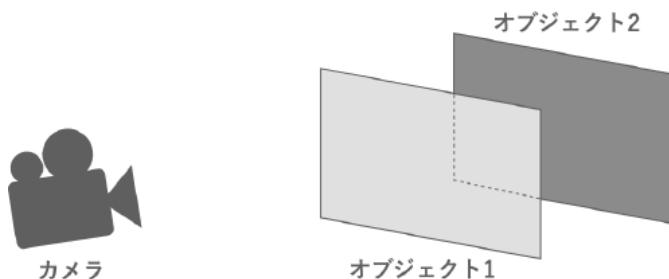


图2.14 两个重叠的物体。

首先考虑这两个对象都是不透明的情况。在这种情况下，从相机看到的前景物体首先被画出来。这样，在绘制后面的物体时，因为与前面的物体重叠而不可见的部分就不需要处理了。这意味着在这个区域可以跳过碎片着色器的操作，从而达到优化处理负荷的目的。

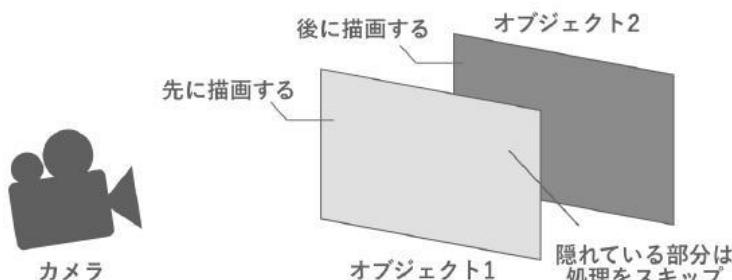


图2.15 不透明的绘图

另一方面，如果两个物体都是半透明的，那么，如果远处的物体没有显示出来，即使它与前景的物体重叠，也是不自然的。这

绘制过程从摄像机看到的远处的物体开始依次进行，重叠区域的颜色与已经绘制的颜色相混合。

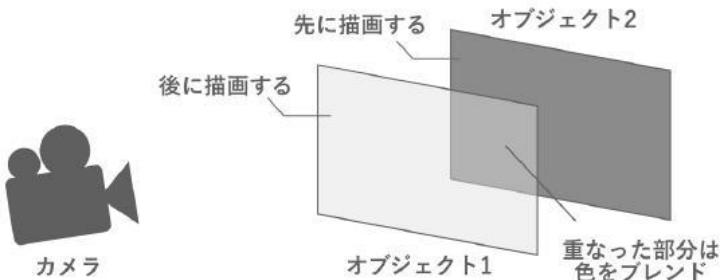


图2.16 半透明的图纸

因此，与不透明绘图不同，半透明绘图也必须对重叠的对象进行处理。如果有两个半透明的物体被画在上面，整个屏幕会被处理两次。以这种方式将半透明的物体画在彼此的上面，被称为过度绘制。过多的过度绘制会给GPU带来沉重的处理负荷，导致性能下降，所以在绘制半透明物体时，有必要设置适当的规定。

假设为正向渲染。

有几种实现渲染管道的方法。其中。
本节中的描述假设是向前渲染。有些方面并不部分适用于其他渲染技术，如延迟渲染。

2.2.3 绘图调用--设置传递调用和批处理

在渲染过程中，处理负荷不仅在GPU上，也在CPU上。

如上所述，在渲染一个物体时，CPU向GPU发出指令来绘制它。这被称为**draw call**，对有多少个要渲染的对象就执行多少次。此时，纹理等信息也在之前的绘制调用中被发送到了GPU。

如果它们与GPU绘制的对象不同，就会对它们进行处理，将它们设置为GPU的对象。这被称为“**套路调用**”，是一个相对沉重的过程。这个过程是在CPU的渲染线程中进行的，这给CPU带来了处理负荷，如果数量太多，会影响性能。

Unity实现了一种叫做**drawcall batching**的机制来减少**drawcalls**。这是一种机制，具有相同纹理和其他信息的物体的网格，即相同的材料，在CPU端处理中被提前组合起来，并通过单个绘制调用来绘制。有两种批处理方式：动态批处理和静态批处理，前者是在运行时进行批处理，后者是提前创建合并的网格。

可编写脚本的渲染管道还实现了一个叫做**SRP Batcher**的机制。这允许将set-pass调用合并为一个调用，即使网格和材质不同，只要着色器的变体是相同的。这种机制并没有减少绘图调用的数量，但它确实减少了设置通行的调用，因为正是设置通行的调用具有最高的处理负荷。

关于这些批处理的更详细的信息，见“7.3 减少绘图调用”。

GPU实例化。

一个可以产生类似于批处理效果的功能是**GPU实例化**。这是GPU的一个特性，它允许具有相同网格的对象在一个单一的绘制调用/设置路径调用中被绘制。

2.3 数据表示方法

游戏使用各种数据，包括图像、3D模型、音频和动画。了解这些是如何表现为数字数据的，对于计算内存和存储容量，以及正确配置压缩等设置都很重要。本节总结了基本的数据表示方法。

2.3.1 位和字节

计算机所能代表的最小单位是一个比特；一个比特是一个二进制数字。可以表示由0或1代表的范围，即0和1的两个组合。这意味着只能表达简单的信息，例如，开关的开/关。



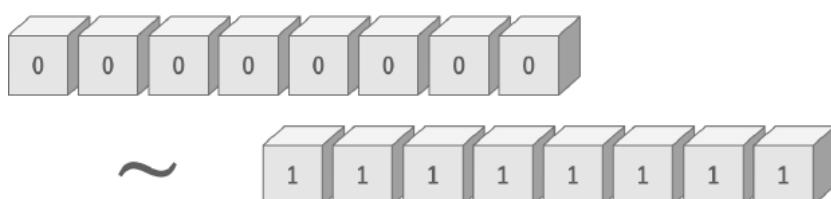
▲图2.17 一个比特的信息量

在这里，使用两个比特，我们可以看到，我们可以表示两个二进制数字的范围，即四个可能的组合：四个，所以我们可以表示哪个键被按下的信息--例如，上、下、左或右。



▲ 图2.18 2比特的信息

同样，当涉及到8位时，可由8位二进制数字表示的范围，即 $2^8 = 256$ 个街道。这些信息是。在这一点上，似乎可以表达各种信息。而这8位是1巴字节是以字节为单位表示的。换句话说，一个字节是一个单位，可以代表256个不同数量的信息。



▲ 图2.19 8位信息内容

另一个更大的数字的计量单位是千字节（KB），它代表1000个字节，而有一兆字节（MB）代表1000千字节。

千字节和小米的字节

以上，1KB被写成1,000字节，但在某些情况下，1KB可能被称为1,024字节。当明确叫出时，1,000字节被称为1千字节（KB），1,024字节被称为1基比字节（KiB）。这同样适用于兆字节。

2.3.2 图片。

图像数据被表示为一组像素。例如，一个 8×8 像素的图像对于一幅图像来说，它总共由 $8 \times 8 = 64$ 个像素组成。

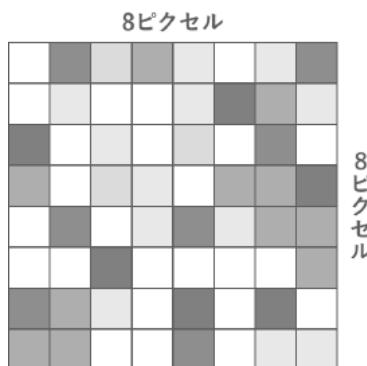
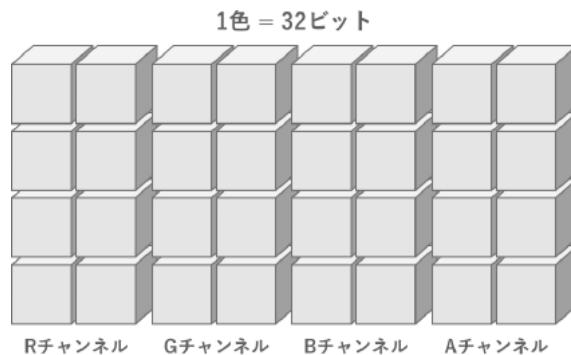


图2.20 图像数据。

然后每个像素都有自己的颜色数据。那么，数字数据中的颜色是如何表示的呢？

颜色首先是由四个元素组合而成的：红色（Red）、绿色（Green）、蓝色（Blue）和透明度（Alpha）。这些被称为通道，并由每个通道的首字母RGBA来表示。

常用的颜色表示方法True Colour以256步表示每个RGBA值。正如上一节所解释的，256步意味着8比特。换句话说，真彩色可以用4个通道×8比特=32比特的信息来表示。



▲图2.21 一种颜色的信息量

因此，例如，对于一个 8×8 像素的真彩图像，信息量为 $8\text{像素} \times 8\text{像素} \times 4\text{通道} \times 8\text{比特} = 2,048\text{比特} = 256\text{字节}$ ；对于一个 $1,024 \times 1,024$ 像素的真彩图像，信息量为 $1,024\text{像素} \times 1,024\text{像素} \times 4\text{通道} \times 8\text{比特} = 33,554,432\text{比特} = 4,194,304\text{字节} = 4,096\text{千字节} = 4\text{兆字节}$ 。

2.3.3 图像的压缩

在实践中，图像大多是作为压缩数据使用的。

压缩是通过设计存储数据的方法来减少数据量的过程。例如，假设有五个相同颜色的像素彼此相邻。在这种情况下，每个像素

如果你有一个颜色的信息和一排有五个的信息，而不是单一颜色的五个颜色信息，那么信息量就会减少。



图2.22 压缩。

在实践中，有许多更复杂的压缩方法。

作为一个具体的例子，我们介绍ASTC，一个典型的移动压缩格式。应用ASTC6x6格式，一个1024x1024的纹理被从4兆字节压缩到大约0.46兆字节。换句话说，其结果是压缩不到八分之一的容量，这就承认了进行压缩的重要性。

作为参考，下面介绍主要用于移动领域的ASTC格式的压缩率。

▼ 表2.2 压缩格式和压缩率

压缩格式	压缩比
ASTC RGB(A) 4x4	0.25
ASTC RGB(A) 6x6	0.1113
ASTC RGB(A) 8x8	0.0625
ASTC RGB(A) 10x10	0.04
ASTC RGB(A) 12x12	0.0278

在Unity中，可以使用纹理导入设置为每个平台指定各种压缩方法。因此，通常的做法是导入未压缩的图像，并使用导入设置来压缩它们，以生成最终要使用的纹理。

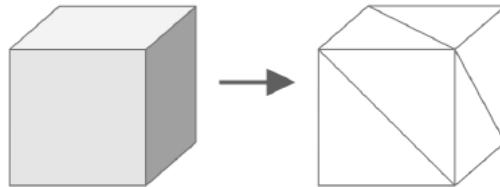
GPU和压缩格式

当然，按照某些规则压缩的图像必须按照这些规则进行解压缩。这个解压过程是在运行时进行的。为了最大限度地减少这种处理负荷，使用GPU支持的压缩格式非常重要。ASTC是一种典型的压缩格式，由移动设备上的GPU支持。

2.3.4 网眼

在3DCG中，三维形状是通过在三维空间中将许多三角形连接起来来表示的。这个三角形的集合被称为**网格**。

立体は三角形で構成される



▲图2.23 由三角形组合而成的三维物体

这个三角形可以作为数据表示为三维空间中三个点的坐标信息。这些点中的每一个都被称为顶点，其坐标被称为顶点坐标。另外，有一个网眼被称为一个顶点的所有顶点信息都存储在一个数组中。

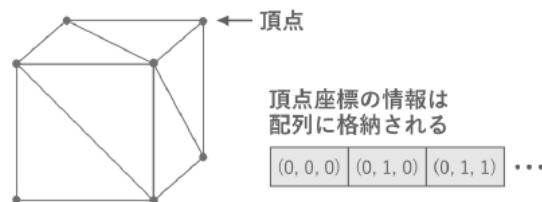
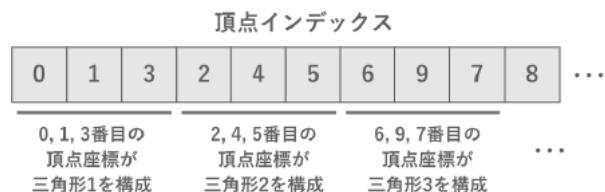


图2.24 顶点信息。

由于顶点信息被存储在一个数组中，因此需要单独的信息来表明哪些顶点信息被组合成一个三角形。这被称为顶点索引，被表示为一个int类型的数组，代表顶点信息数组的索引。



▲ 图2.25 顶点指数

对物体进行纹理和照明需要额外的信息。例如，贴图纹理需要UV坐标。照明也需要诸如顶点颜色、法线和切线等信息。

下表总结了主要顶点信息和每个顶点的信息量。

▼ 表2.3 顶点信息

名字。	每个顶点的信息量
顶点坐标	三维平面=12字节
紫外线坐标	二维平面=8个字节
顶色	4维平面=16字节
法向量	三维平面=12字节
切线	三维平面=12字节

事先确定顶点的数量和顶点信息的类型是很重要的，因为网格数据会随着顶点数量和单个顶点处理的信息量增加而增长。

2.3.5 关键帧动画

游戏在许多方面使用动画，如用户界面的动画和3D模型的运动。实现动画的一个典型方法是关键帧动画。

一个关键帧动画由一个数据数组组成，代表某一时间（关键帧）的数值。关键帧之间的数值是由插值决定的，因此可以把它们当作平滑、连续的数据。

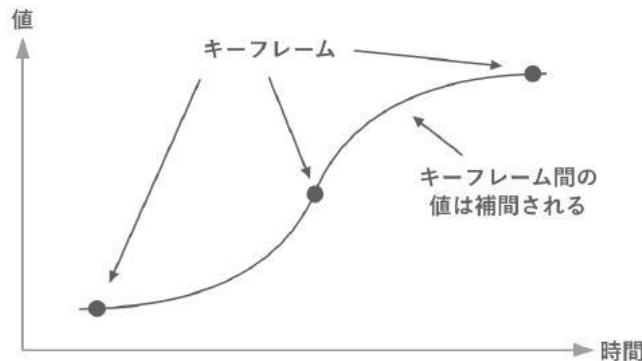


图2.26 关键框架。

除了时间和数值之外，关键帧还有其他信息，如切线及其权重。通过使用这些进行插值计算，可以用较少的数据量实现更复杂的动画。

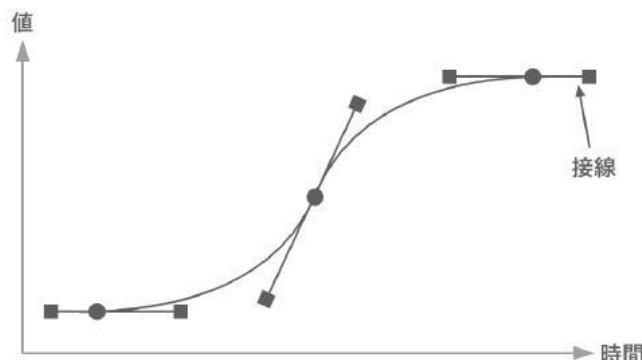


图2.27 切线和权重

在关键帧动画中，关键帧越多，动画就越复杂。然而，数据量也随着关键帧的数量而增加。由于这个原因，关键帧的数量必须被适当地设置。

在Unity中，关键帧可以在模型导入设置中减少，如下图所示。

2.4 统一的工作方式

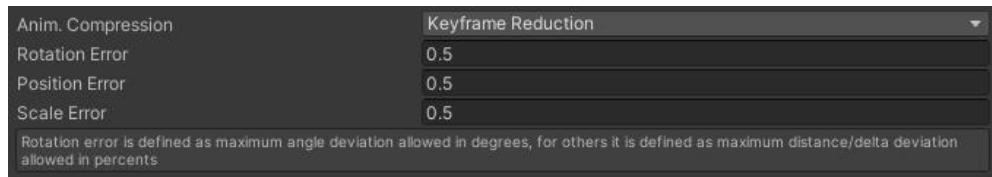


图2.28 导入设置。

关于如何设置的细节，见4.4 动画。

2.4 统一的工作方式

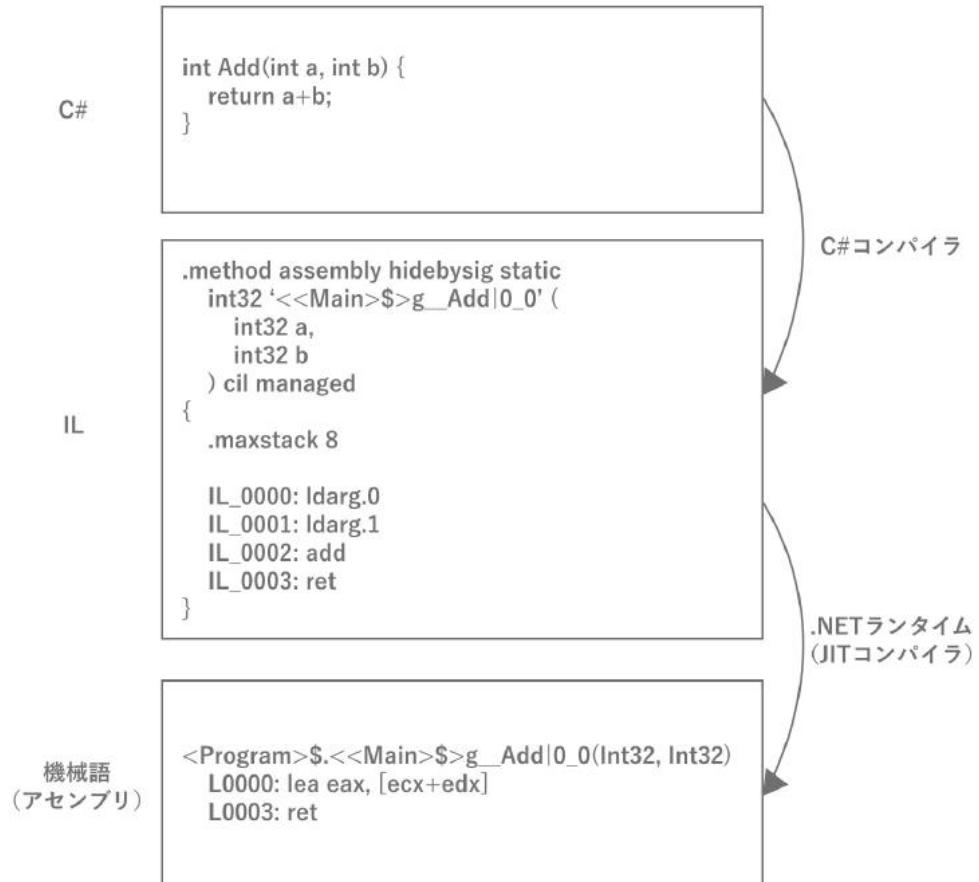
不言而喻，了解Unity引擎的实际工作原理对于调整你的游戏非常重要。本节解释你应该知道的Unity操作原则。

2.4.1 二进制文件和运行时间

首先，本节解释了Unity实际上是如何运行运行时的。

C#和运行时

当在Unity中创建一个游戏时，开发者用C#语言对其进行编程，这是一种编译语言，因为在Unity中开发游戏时，通常会进行编译（构建）。然而，C#与传统的C语言和其他语言不同的是，在编译时，它被编译为.NET的中间语言（IL）。在.NET框架运行时，将其转换为机器语言。



▲图2.29 C# 编译过程

之所以使用IL，是因为一旦转换为机器语言，二进制文件只能在单一平台上运行；而使用IL，只需为该平台提供运行时间，就可以在任何平台上运行，从而不需要单独的这消除了为每个平台准备二进制文件的需要。因此，Unity的基本原则是，通过编译源代码得到的IL在各自环境的运行系统上执行，从而实现多平台兼容。

让我们检查一下IL代码。

很少见到的IL代码，对于内存分配和执行速度等性能考虑非常重要。例如，一个数组和一个列表看似有相同的foreach循环，但产生不同的IL代码，而数组是性能更好的代码。你也可能会发现非故意的隐藏堆分配。为了了解C#和IL代码之间的对应关系，建议定期检查你的C#代码的IL转换结果。IL代码本身是一种叫做汇编的低级语言，很难理解。在这种情况下，可以使用名为SharpLab^{*2}的网络服务，通过C#->IL->C#检查代码，反之亦然，使其更容易理解。本手册后半部分第10章“调谐实践--脚本（C#）”中介绍了一个实际转换的例子。

IL2CPP

如上所述，Unity基本上将C#编译成IL代码并在运行时运行，但从2015年左右开始在一些环境中出现了问题。如上所述，C#需要在每个环境中运行一个运行时，以便执行IL代码，但事实上，在此之前，Unity已经使用**Mono**，一个.NET框架的OSS实现，多年来一直如此。多年来，Unity公司一直使用.NET框架的OSS实现，而且Unity公司自己也修改和使用它。换句话说，为了让Unity成为64位兼容，有必要让分叉的Mono成为64位兼容。当然，这需要大量的工作，所以Unity通过开发一种名为**IL2CPP**的技术来克服这一挑战。

IL2CPP顾名思义就是IL到CPP，是一种将IL代码转换为C++代码的技术，一旦输出为C++代码，就可以在相应的开发工具链中编译为机器语言，因为C++是一种高度通用的语言，在任何开发环境中都有原生支持。C++代码可以在相应的开发工具链中被编译成机器语言。因此，支持64位是工具链的工作，Unity方面不需要处理这个问题。另外，与C#不同的是，代码在构建时被编译成机器语言，所以不需要在运行时将其转换为机器语言。

^{*2} <https://sharplab.io/>

其好处是，它消除了对性能的需求并提高了性能。

虽然C++代码一般有构建速度慢的缺点，但IL2CPP技术已经成为Unity的基石，一举解决了64位支持和性能问题。

统一的运行时间

顺便说一下，在Unity中，开发者用C#语言对游戏进行编程，但Unity自己的运行时间，即所谓的引擎，实际上并不以C#语言运行。源码本身是用C++编写的，而被称为播放器的部分是预先构建好的，可以在每个环境中运行。

- 用于快速和节省内存的性能
- 支持尽可能多的平台
- 为了保护发动机的知识产权（黑框）。

由于开发者编写的C#代码只能在C#中运行，Unity需要两个区域：原生运行的引擎部分和用户代码部分，在C#运行时运行。引擎和用户代码在执行过程中通过适当的数据交换进行工作。例如，当从C#调用**GameObject.transform**时，游戏的所有执行状态，如场景的状态，都是在引擎内部管理的，所以第一步是进行本地调用，访问本地区域的内存数据，并向C#返回值。需要注意的是，C#和本地的内存是不共享的，所以每次在C#中需要数据的时候，内存都会在C#这边分配。API调用也很昂贵，会发生本地调用，所以需要一种不需要频繁调用的缓存值的优化技术。

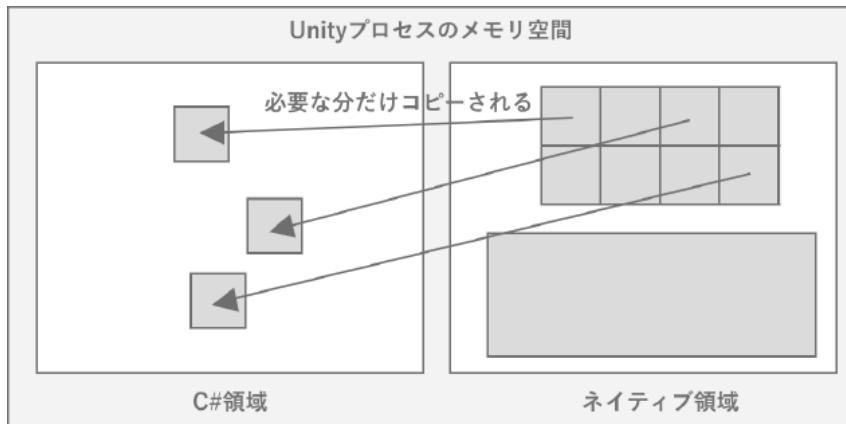


图2.30 Unity中内存状态的图像。

因此，在开发Unity时，你需要在一定程度上注意到隐形的引擎部分。由于这个原因，看一下连接Unity引擎和C#本地区域的接口的源代码是个好主意。幸运的是，Unity在GitHub上发布了^{*3}的C#部分，这非常有用，因为它显示了这主要是一个本地调用。我们建议在必要时利用这一点。

2.4.2 资产实体

正如上一节所解释的，Unity引擎是原生运行的，因此在C#端基本上没有数据。这同样适用于对资产的处理：资产在本地区域被加载，只有引用被返回到C#或数据被复制回来。因此，在加载资产时，主要有两种方法：指定一个路径让Unity引擎加载资产，或者直接传递原始数据，如字节数组。如果指定了路径，C#端就不会消耗内存，因为它是在本地区域加载的，但如果从C#端加载和处理字节数组等数据并传递，C#端和本地端都会加倍消耗内存。

资产实体在本机上的事实也增加了调查资产多负载和泄漏的难度。这是因为开发人员主要关注的是剖析和调试C#端，仅仅看C#端的执行状态是很难理解的，而C#端仅仅看它的执行状态则更难理解。

³ <https://github.com/Unity-Technologies/UnityCsReference>

问题是，本地领域的剖析依赖于Unity提供的API，这限制了可用工具的数量。我们将在本文中介绍使用各种工具的分析技术，如果你知道C#和本地之间的空间，就会更容易理解。

2.4.3 课题。

线程是程序执行的一个单位，通常由一个进程中的几个线程组成。

由于一个CPU核心一次只能处理一个线程，为了处理多个线程，程序是通过在线程之间高速切换来执行的。这被称为上下文切换。上下文切换会产生开销，所以如果频繁发生，处理效率会降低。

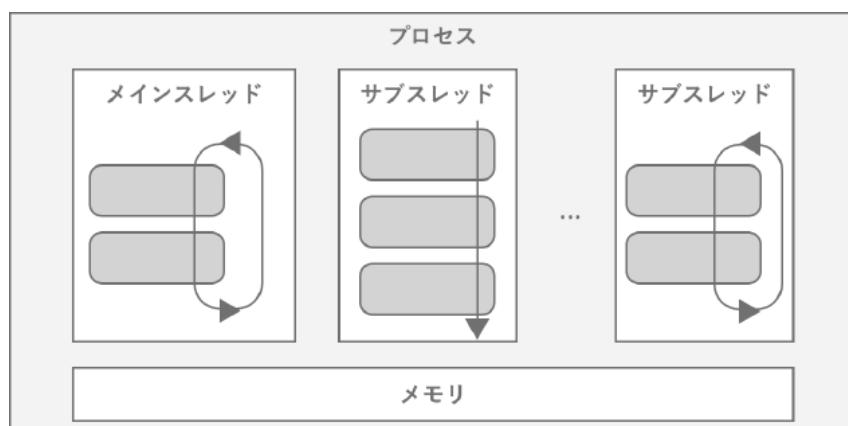


图2.31 线程的示意图

当程序运行时，会创建一个底层的主线程，程序根据需要从该线程中创建并管理其他线程。Unity的游戏循环被设计为在一个线程上运行，因此用户编写的脚本基本上会在主线程上运行。Unity游戏循环被设计为单线程运行。相反，试图从主线程之外调用Unity APIs将导致大多数API的错误。

当从主线程创建另一个线程来执行一个进程时，不知道该线程何时被执行，何时完成。因此，在线程之间进行同步处理

2.4 统一的工作方式

一种叫做信号的机制是允许一个线程等待另一个线程处理消息的手段。如果你想等待另一个线程的进程，你可以通过让该线程用信号通知你来释放等待。这种信号等待也在Unity内部使用，在剖析时可以观察到，但需要注意的是，正如名字WaitFor~所暗示的，它只是在等待另一个进程。

统一内部的主题

然而，如果每个进程都在主线程中执行，那么整个程序的处理将需要很长的时间。如果有几个繁重的进程，而这些进程又不是相互依存的，那么，如果可以通过在一定程度上同步处理这些进程，就有可能缩短程序的执行时间。为了实现这些加速，在游戏引擎中使用了一些并行进程。其中之一是渲染线。顾名思义，这是一个专门用于渲染的线程，负责将主线程计算出的帧绘制信息作为图形命令发送给GPU。

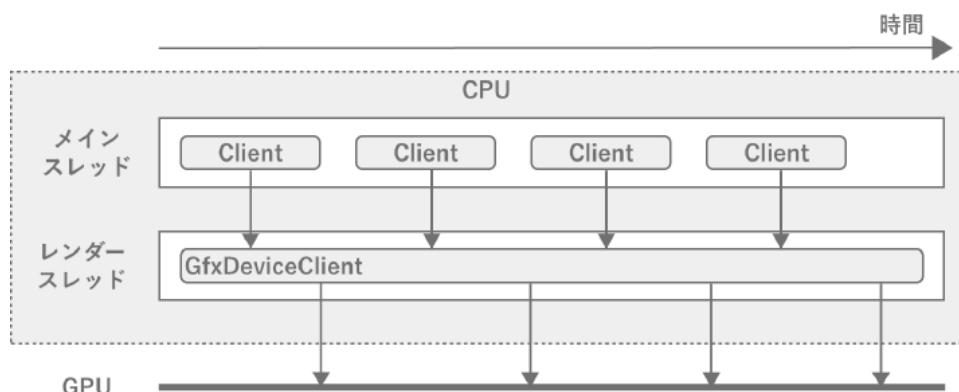


图2.32 主线程和渲染线程

主线程和渲染线程像流水线一样运行，因此，当渲染线程在处理时，下一帧的计算就开始了。然而，如果在渲染线程中处理一帧的时间变得太长，即使下一帧的计算已经完成，主线程也无法开始绘制下一帧，主线程将不得不等待。在游戏开发中，需要注意的是，如果主线程或渲染线程变得过于沉重，FPS就会下降。

可以并行处理的用户进程的线程

此外，还有一些游戏特有的计算任务可以并行执行，如物理引擎和摇动。在Unity中，工人线程的存在是为了在主线程之外执行这种计算。工作线程执行的是通过JobSystem生成的计算任务，所以如果使用JobSystem可以减少主线程的处理负荷，就应该积极使用。当然，你也可以不使用JobSystem来生成自己的线程。

虽然线程对性能调整很有用，但我们建议不要在黑暗中使用它们，因为过度使用有可能反过来降低性能并增加处理的复杂性。

2.4.4 游戏循环

常见的游戏引擎，包括Unity，有一个常规的引擎进程，称为游戏循环（player loop）。一般来说，描述一个循环的简明方式如下。

1. 控制器的输入处理，如键盘、鼠标和触摸显示器
2. 计算出一帧时间内应该进展的游戏状态。
3. 渲染新的游戏状态。
4. 根据目标FPS，等到下一帧。

这个循环重复进行，将游戏作为视频输出到GPU。如果处理一帧需要更长的时间，FPS当然会下降。

Unity中的游戏循环

Unity中的游戏循环在官方的Unity参考资料中有所说明，你们都或多或少地看到过。

⁴ <https://docs.unity3d.com/ja/current/Manual/ExecutionOrder.html>

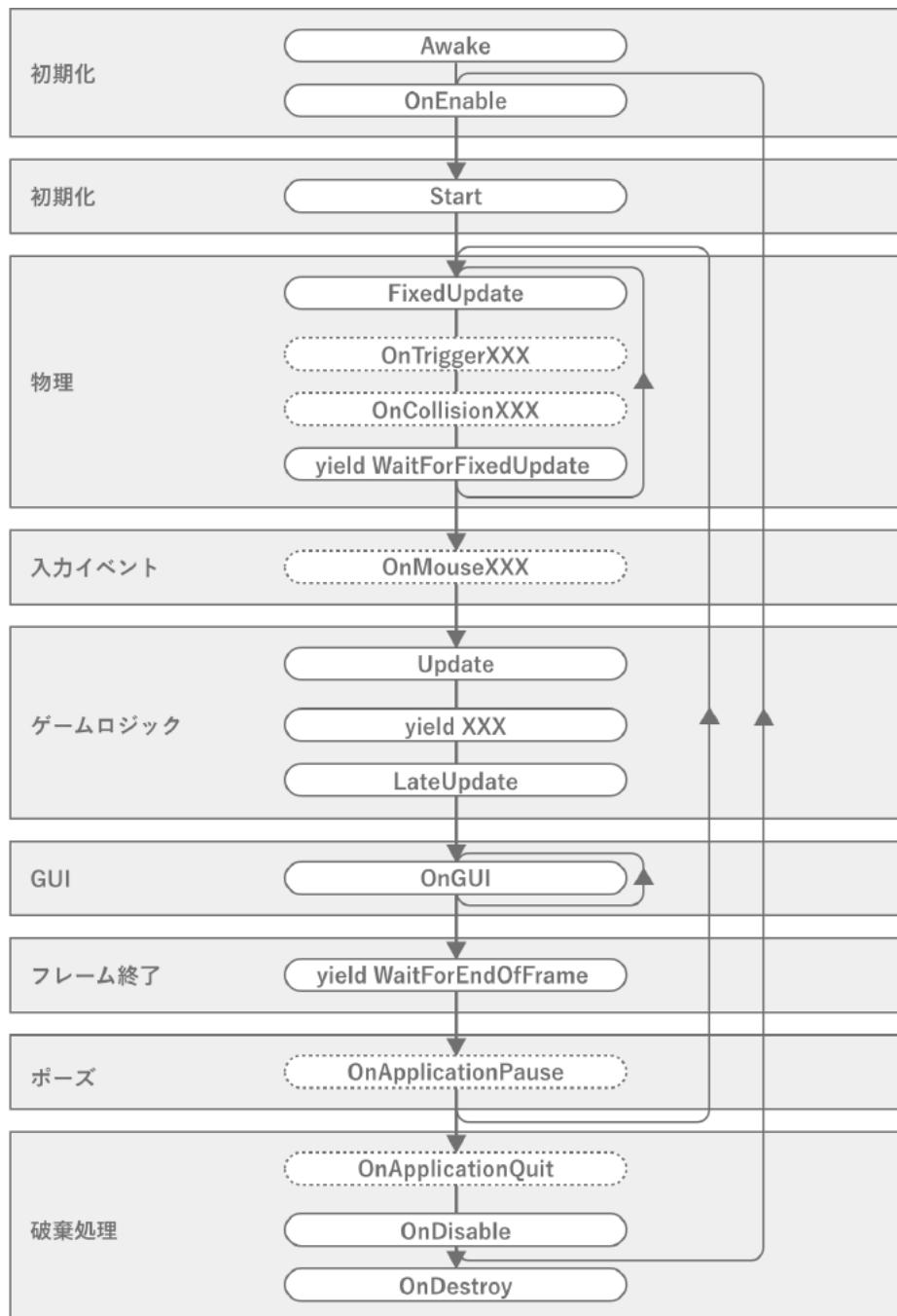


图2.33 Unity事件的执行顺序。

这张图严格显示了MonoBehaviour事件的执行顺序，这与作为游戏引擎的游戏循环⁵不同，但对于开发者应该知道的游戏循环来说，这已经足够了。特别重要的事件有：Awake、OnEnable、Start、 FixedUpdate、Update、LateUpdate、OnDisable、OnDestroy和各种coroutine的处理时机。事件执行顺序或时间上的错误会导致意外的内存泄漏或额外的计算。因此，应该了解调用重要事件的时间性质和同一事件中的执行顺序。

对于物理操作来说，有一些特殊的问题，比如物体滑过而不被检测到碰撞，如果它们与正常游戏循环执行的时间间隔相同。由于这个原因，循环通常在与游戏循环不同的时间间隔内运行，因此物理程序的运行频率更高。然而，如果循环的运行频率很高，它们可能会与主游戏循环的更新过程发生冲突，所以有必要在一定程度上同步这一过程。因此，要注意，如果物理计算的工作量超过了必要的范围，可能会影响到框架的绘制过程，如果框架的绘制过程比较重，物理计算可能会延迟和滑过，这可能会相互影响。

2.4.5 游戏对象

如上所述，Unity引擎本身是原生运行的，所以C#中的Unity API在大多数情况下也是调用内部原生API的一个接口。对于GameObjects和定义附属于它们的组件的MonoBehaviours也是如此，它们总是有来自C#端的本地引用。然而，如果本地端管理数据，并且在C#端也有对它们的引用，那么在销毁它们时就会有不便之处。这是因为对于在本地端已经销毁的数据，C#的引用不能被自行删除。

事实上，清单2.1检查了被破坏的GameObject是否为空，但在日志中输出为真。这对于标准的C#行为来说是不自然的，因为_gameObject没有被分配为null，所以仍然应该有一个对GameObject类型实例的引用。

⁵ <https://tsubakit1.hateblo.jp/entry/2018/04/17/233000>

2.4 统一的工作方式

▼ 列表 2.1 销毁后的参考测试

```
public class DestroyTest : UnityEngine.MonoBehaviour
{
    private UnityEngine.GameObject _gameObject;

    private void Start()
    {
        _gameObject = new UnityEngine.GameObject("test");
        StartCoroutine(DelayedDestroy());
    }

    System.Collections.IEnumerator DelayedDestroy()
    {
        // 缓存 WaitForSeconds以重复使用
        var waitOneSecond = new UnityEngine.WaitForSeconds(1f);
        yield return waitOneSecond;

        Destroy(_gameObject);
        // 产量返回waitOneSecond。

        // _gameObject不是空的，但结果是真的
        UnityEngine.Debug.Log(_gameObject == null);
    }
}
```

这是因为Unity的C#方面的机制控制了对被丢弃数据的访问。事实上，如果你参考UnityEngine.Object的源代码⁶，在Unity的C#实现部分，你会发现以下内容。

▼ 清单2.2 UnityEngine.Object中==操作符的实现

```
// 摘录。
public static bool operator==(Object x, Object y) {
    return CompareBaseObjects(x, y);
}

static bool CompareBaseObjects(UnityEngine.Object lhs,
    UnityEngine.Object rhs)
{
    bool lhsNull = ((对象)lhs) == null; bool
    rhsNull = ((对象)rhs) == null;

    如果 (rhsNull && lhsNull) 返回true。
    if (rhsNull) return !IsNativeObjectAlive(lhs);
    if (lhsNull) return !
}
```

⁶ <https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Runtime/Export/Scripting/UnityEngineObject.bindings.cs>

```
        return lhs.m_InstanceID == rhs.m_InstanceID;
    }

    static bool IsNativeObjectAlive(UnityEngine.Object o)
    {
        如果(o.GetCachedPtr() != IntPtr.Zero)
            返回true。

        if (o is MonoBehaviour || o is ScriptableObject)
            return false;

        返回 DoesObjectWithInstanceIDExist(o.GetInstanceID())。
    }
}
```

综上所述，对被破坏的实例进行空值比较将是真实的，因为当进行空值比较时，本机方面会检查数据的存在。这导致非空的**GameObjects**实例的行为就像它们是部分空的一样。这个属性乍一看很有用，但它也有一个非常棘手的方面。这是因为`_gameObjec t`实际上不是空的，这会导致内存泄漏。单个`_gameObject`的内存泄漏是显而易见的，但如果，你有一个对该组件的巨大数据的引用，例如一个主控，这将导致巨大的内存泄漏，因为该引用仍然是C#，不受垃圾收集的影响。为了避免这种情况，需要采取一些措施，如将null赋值给`_gameObject`。

2.4.6 资产捆绑

智能手机的游戏受限于应用程序的大小，并不是所有的资产都可以包含在应用程序中。因此，为了根据需要下载资产，Unity有一个叫做**AssetBundle**的机制，它打包了多个资产并动态加载。乍一看，这似乎很容易处理，但在大型项目中，这需要对内存和**AssetBundle**有充分的了解，并进行仔细的设计，因为如果设计不当，内存可能会浪费在意想不到的地方。因此，本节从调整的角度解释了你需要知道的关于**AssetBundle**的内容。

AssetBundle压缩设置

资产包在构建时默认以**LZMA**压缩方式进行压缩。这可以通过改变**BuildAssetBundleOptions**为**UncompressedAssetBundle**来改变为无压缩，或者通过改变为**ChunkBased Compression**来改变为**LZ4**压缩。这些设置之间的差异往往显示在下面的表2.4中。

2.4 统一的工作方式

▼ 表2.4 AssetBundle压缩设置之间的差异

(数据)项	非压缩的	LZMA	LZ4。
文件大小	特大号	超小	小
加载时间	(太)快	深夜	相当快。

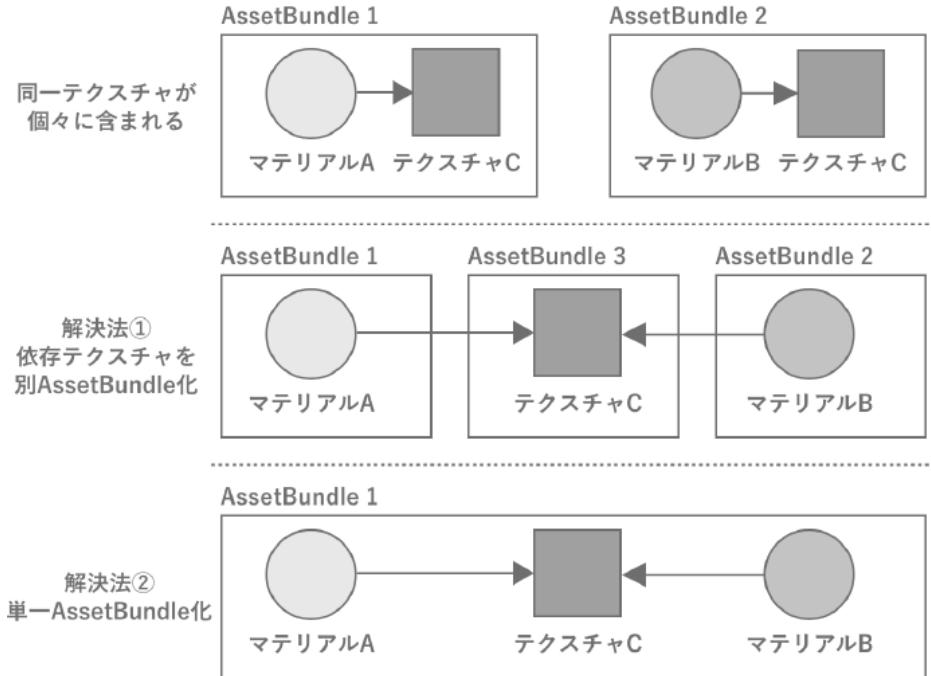
换句话说，未压缩的文件对最快的加载时间有好处，但为了避免浪费智能手机的存储空间，它基本上不能使用，因为文件大小是致命的大。另一方面，LZMA具有最小的文件大小，但由于算法问题，具有解压速度慢和部分解压的缺点。LZ4是一个平衡速度和文件大小的压缩设置，正如其名称ChunkBasedCompression所暗示的那样，允许部分读取，而不必像LZMA那样解压整个文件。

AssetBundle也有Caching.compressio nEnabled，它在终端缓存时改变压缩设置。换句话说，通过使用LZMA进行传输，并在终端转换为LZ4，可以最大限度地减少下载量，并在实际使用时享受LZ4的好处。然而，终端上的重新压缩意味着终端上更高的CPU处理成本以及内存和存储空间的暂时浪费。

资产捆绑的依赖性和重复性

如果一项资产依赖于一个以上的资产，在对其进行AssetBundle-ing时必须注意。例如，如果材料A和材料B依赖于纹理C，而你只对材料A和B进行资产捆绑，而不对纹理进行资产捆绑，那么生成的两个资产捆绑中的每一个都将包含纹理C，这将导致重复和这是在浪费空间。当然，这在空间使用方面是浪费的，但在内存方面也是浪费的，因为当两种材料加载到内存时，纹理是单独实例化的。

为了避免同一资产出现在多个AssetBundle中，纹理C必须变成一个独立的AssetBundle并依赖于材料的AssetBundle，或者材料A、B和纹理C必须合并成一个AssetBundle。材料A、B和纹理C必须被做成一个单一的AssetBundle。

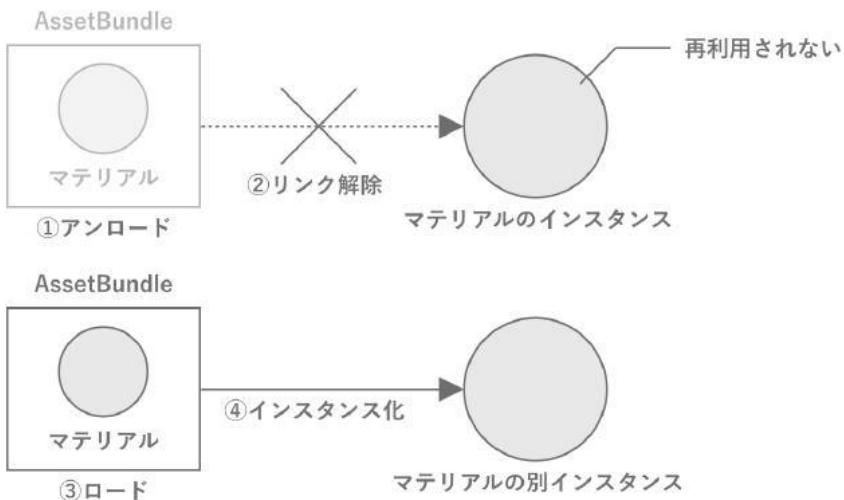


▲ 图2.34 带有AssetBundle依赖关系的例子。

从AssetBundle加载的资产的身份

从AssetBundle加载资产的一个重要属性是，只要AssetBundle被加载，无论你加载多少次，同一资产的同一实例都会被返回。这意味着Unity内部管理加载的资产，AssetBundle和资产在Unity内部被捆绑在一起。这个属性可以用来避免在游戏端创建一个资产缓存机制，而把它留给Unity。

然而，如果用 AssetBundle.Unload(false) 卸载资产，图2.35
请注意，即使你再次从同一个AssetBundle加载相同的资产，它也会变成一个不同的实例，就像下面的例子一样。这是因为AssetBundle在卸货时与资产解除了联系，资产的管理处于一种无序状态。



▲ 图2.35 由于对AssetBundle和assets管理不当而导致的内存泄漏的例子。

销毁从AssetBundle加载的资产

用AssetBundle.Unload(true)卸载一个AssetBundle时，会完全销毁加载的资产，所以不存在内存问题，但当使用AssetBundle.Unload(true)时，就会出现内存问题。Unload(false)，资产将不会被销毁，除非在适当的时候调用资产卸载指令。UnloadUnusedAssets必须被适当地调用，以便在切换场景时销毁资产。UnloadUnusedAssets，还需要注意的是，正如其名称所暗示的，如果引用仍然存在，则不会被释放。注意，如果使用Addressable，AssetBundle.Unload(true)会被内部调用。

2.5 C#基础知识。

本节介绍了C#的语言规范和程序执行行为，这对性能调优至关重要。

2.5.1 堆栈和堆

在“堆栈和堆”中，我们介绍了堆栈和堆作为程序执行期间的内存管理方法的存在。堆栈由操作系统管理，而堆则由程序员管理。换句话说，知道堆内存是如何管理的，就可以实现内存感知的实现。管理堆内存的机制在很大程度上取决于程序来源的源代码的语言规范，因此本节介绍C#中的堆内存管理。

原有的堆内存使用时必须分配，内存使用完毕后必须释放。如果内存没有被释放，就会发生内存泄漏，应用程序使用的内存区域会扩大，最终导致崩溃。然而，C#并没有一个明确的内存释放过程。这是因为在执行C#程序的.NET运行环境中，堆内存由运行时自动管理，已用完的内存会在适当的时候释放。由于这个原因，堆内存也被称为管理堆。

栈上分配的内存与函数的寿命相匹配，所以只需要在函数结束时释放，但堆上分配的内存很可能会在函数的寿命之外继续存在。这意味着在不同的时间需要和使用堆内存，所以需要一种机制来自动和有效地使用堆内存。这种机制被称为垃圾收集，其细节将在下一节介绍。

事实上，Unity中的**GC.Alloc**是它自己的术语，由垃圾收集和Alloc代表正在分配给堆内存的内存（Allocation）。因此，减少GC.Alloc可以减少动态分配的堆内存的数量。

2.5.2 垃圾收集

在C#内存管理中，搜索和释放未使用的内存被称为垃圾收集，简称“GC”。垃圾收集器是循环执行的。然而，执行的确切时间取决于算法。它使堆上的所有对象都被一次性扫描，所有已经被解除引用的对象都被删除。这意味着，被解除引用的对象被删除，内存空间被释放出来。

2.5 C#基础知识。

有不同的垃圾收集算法，但Unity默认使用Boehm GC算法；Boehm GC算法的特点是“非生成性”和“非压缩性”。非特定世代”意味着每一次垃圾收集运行都要一次性搜索整个堆。这降低了性能，因为搜索区域随着堆的扩大而扩大。“无压缩”意味着在记忆中没有对象的移动来关闭对象之间的间隙。这意味着碎片化，即在内存中产生小的空隙，往往会发生，并且管理的堆往往会扩大。

每一个都是一个计算昂贵的同步进程，会停止所有其他进程，所以在游戏中运行它们会导致所谓的“停止世界”进程下降。

从Unity 2018.3开始，现在可以指定GCMode，并且可以暂时禁用。

```
1: GarbageCollector.GCMode = GarbageCollector.Mode.Disabled.
```

然而，如果在GC Alloc被禁用期间执行，堆区就会被扩展和消耗，最终无法新分配，导致应用程序崩溃。因为内存使用量很容易增加，所以有必要实现，使GC Alloc在禁用期间完全不执行，而且实现成本很高，所以实际能使用的情况很有限。例如，只禁用射击游戏的射击部分）。

另外，从Unity 2019开始，可以选择增量GC；有了增量GC，现在的垃圾收集过程是跨帧进行的，所以大的峰值可以比以前得到缓解。然而，对于那些需要在减少每帧处理时间的同时最大限度地提高功率的游戏来说，有必要在实现中避免GC Alloc。具体例子见。
拨款，以及如何处理”。

我们应该何时开始工作？

由于游戏有大量的代码，在所有功能的实施完成后，性能。
在进行调整时，往往无法避免GC Alloc的设计/。

可遇到的实施。从早期设计阶段就不断意识到这种情况的发生，编码时往往回减少因返工而产生的成本，提高总的开发效率。

理想的实施流程是首先创建一个强调速度的原型，以验证手感和核心可玩性，然后在进入下一阶段的全面生产时审查和重组设计。在这个重组阶段，努力消除 GC.Alloc 是健康的。在某些情况下，为了加快开发进程，可能需要降低代码的可读性，所以从原型阶段开始工作也会降低开发速度。

2.5.3 结构(struct)

在C#中，有两种类型的复合类型定义：类和结构。基本前提是，类是一种引用类型，结构是一种值类型；我们将引用MSDN的《在类和结构之间的选择》^{*7}，回顾每一种类型的特点、选择它们的标准以及对它们的使用说明。

内存分配位置的差异

引用类型和值类型的第一个区别是，它们分配内存的方式不同。虽然这有点不精确，但可以安全地假设参考类型是在内存的堆区分配的，并且要进行垃圾回收。值类型被分配到内存中的堆栈区，不受垃圾收集的影响。值类型的分配和取消分配通常比引用类型的费用低。

然而，在引用类型的字段中声明的值类型和静态变量被分配到堆区。注意，定义为结构的变量因此不一定分配到堆栈区。

对数组的处理

值类型的数组是内联分配的，数组元素与值类型的实体（实例）排成一列。另一方面，在引用类型的数组中，数组元素是按照引用类型的实体的引用（地址）来排列的。因此，值类型的数组的分配和取消分配比引用类型更困难。

⁷ <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-class-and-struct>之间

成本也低得多。在大多数情况下，值类型的数组还有一个优势，即引用的位置性（空间位置性）得到了极大的改善，这增加了CPU缓存内存的点击概率，有利于加快处理速度。

复制价值

在引用型赋值（assignment）中，引用（地址）被复制。另一方面，在值型赋值（assignment）中，整个值被复制。在32位环境中，地址的大小为4字节，在64位环境中为8字节。因此，大参考类型的赋值比大于地址大小的值类型的赋值成本低。

另外，在使用方法进行数据交换（参数和返回值）方面，引用类型通过值传递引用（地址），而值类型通过值传递实例本身。

```
1: private void HogeMethod(MyStruct myStruct, MyClass myClass){...}.
```

例如，在这个方法中，MyStruct的整个值被复制了。这意味着，随着“我的结构”的大小增加，复制成本也会增加。另一方面，对于MyClass来说，只有myClass的引用被复制成一个值，所以即使MyClass的大小增加，复制成本也保持不变，因为它只针对地址大小。由于复制成本的增加与处理负荷直接相关，因此必须根据要处理的数据的大小来适当选择。

不变性

对一个引用类型的实例所做的改变也会影响到引用同一实例的其他位置。另一方面，当一个价值类型的实例被传递时，它的副本就会被创建。如果一个价值类型的实例被改变，当然不会影响该实例的副本。拷贝不是由程序员明确创建的，而是在传递参数或返回值时隐含地创建的。作为一个程序员，你可能至少经历过一次这样的错误，你以为你在改变一个值，但实际上你只是在对副本设置值，这不是你想做的。建议值类型应该是不可改变的，因为可改变的值类型会让许多程序员感到困惑。

参照调用

一个常见的误用是“引用类型总是通过引用传递”，但前面提到的如前所述，引用（地址）复制是基础，引用传递是在使用ref/in/out参数修饰符时完成的。

```
1: private void HogeMethod(ref MyClass myClass){...}.
```

瞬间，因为引用（地址）是在引用类型的值传递中复制的。
替换源实例并不影响复制源实例，但如果是通过引用传递，就有可能替换源实例
。

```
1: private void HogeMethod(ref MyClass myClass) 2:  
{  
3:     // 重写参数中传递的原始实例  
4:     myClass = new MyClass();  
5: }
```

拳击

框选是将一个值类型转换为一个对象类型或将一个值类型转换为一个接口类型的过程。
箱子是在堆上分配的对象，需要进行垃圾回收。因此，过量的入盒和出盒将导致GC.Alloc。相比之下，当引用类型被铸造时，不会发生这样的框定。

▼ 清单2.7 当从一个值类型投射到一个对象类型时的拳击情况

```
1: int num = 0;  
2: object obj = num; // boxed  
3: num = (int) obj; // 拆箱
```

这种直截了当、毫无意义的拳法从来没有被使用过，但当它被用在方法中的时候呢？

▼清单2.8。 隐性铸造与盒式铸造的例子

```
1: private void HogeMethod(object data){ ...  
} 2:  
3: // 缩写。  
4:  
5 : int num = 0。  
6: HogeMethod(num); //按参数进行装箱
```

这种无意识打拳的情况确实存在。

与简单的任务相比，装箱和拆箱是一个繁重的过程。框选一个值类型需要分配和构建一个新的实例。开箱所需的铸件也是一个重要的负荷，虽然没有装箱那么多。

关于选择类和结构的标准

- 应考虑结构的条件：
 - 如果类型实例很小，并且经常有一个很短的有效期
 - 如果经常嵌入其他物体中
- 撤销结构的条件：除非该类型具有以下所有特征
 - 与原始类型 (int、double等) 一样，当逻辑上表示一个单一值时
 - 实例的大小小于16字节
 - Immutable (不可改变的)。
 - 不需要频繁装箱。

还有一些类型不符合上面的选择标准，但被定义为结构，如Vector4和Quaternion，它们在Unity中经常被使用，虽然不少于16字节。检查如何有效地处理这些问题，包括在复制成本增加的情况下如何避免这些问题，并考虑在某些情况下创建你自己的优化版本，具有同等的功能。

2.6 算法和计算复杂性

游戏编程中使用了各种算法。根据算法的创建方式，计算结果可能是相同的，但由于沿途的不同计算过程，性能可能会有所不同。

可以有很大差异。例如，你会想要一个措施来评估C#中作为标准提供的算法的效率如何，以及你实现的算法的效率如何，分别。作为这些的衡量标准，采用的是计算复杂性的衡量标准。

2.6.1 计算量

计算复杂性是对算法计算效率的衡量，可细分为衡量时间效率的时间复杂性和衡量内存效率的域复杂性。计算复杂性顺序用 O 符号（Landau的符号）表示。由于计算机科学和数学定义在这里并不重要，有兴趣的人应该参考其他书籍。此外，在本文中，被描述为计算量的东西被当作时间计算量来对待。

主要的常用量表示为 $O(1)$ 、 $O(n)$ 、 $O(n^2)$ 和 $O(n \log n)$ 。

是。括号中的 n 表示数据的数量。很容易直观地看到一个过程对数据数量的依赖程度以及过程数量的增加频率。在计算复杂性方面比较性能， $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$

数据数和计算步骤数分别见表2.5和图2.36。表2.5显示了数据数和计算步骤数的比较，图2.36显示了对数比较。

$O(1)$ 被排除在外，因为它的性能与数据的数量无关，而且在没有比较的情况下，它的性能明显优越。例如， $O(\log n)$ 是非常好的，10000个样本有13个计算步骤，1000万个样本有23个计算步骤。

▼ 表2.5 主要计算量中的数据和计算步骤的数量

N	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
10	3	10	33	100	1,000
100	7	100	664	10,000	1,000,000
1,000	10	1,000	9,966	1,000,000	1,000,000,000
10,000	13	10,000	132,877	100,000,000	1,000,000,000,000

2.6 算法和计算复杂性

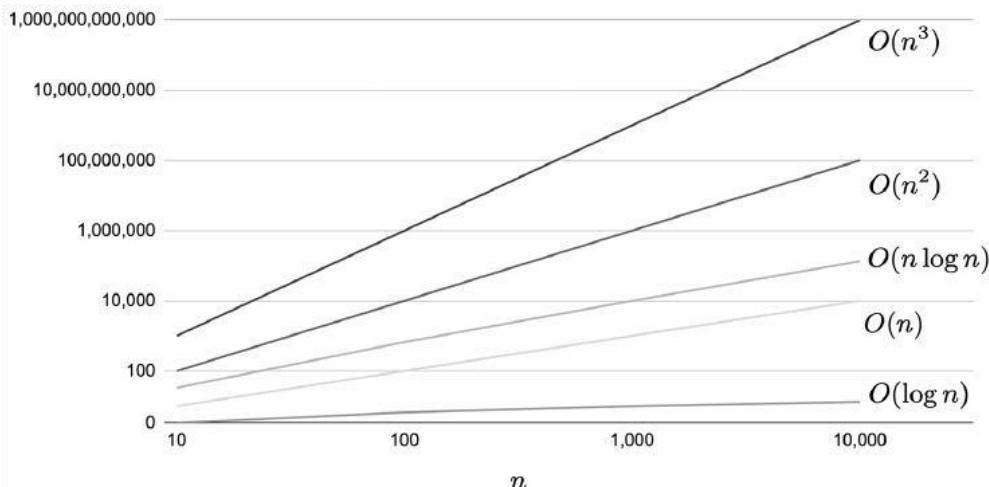


图2.36 各计算量的对数表示法的性能差异比较。

为了证明每种方法的计算复杂性，下面给出了一些代码样本。首先。

$O(1)$ 表示计算复杂性是恒定的，与数据的数量无关。

▼清单2.9 $O(1)$ 的代码示例

```
1: 私有 int GetValue(int[] array) 2
  : {
    //让array成为一个包含某个整数值的数组 4:      var
    value = array[0];
    5:      返回值。
  6: }
```

除了这个方法的存在理由外，这个过程显然与数组中的数据数量无关，并且需要一个固定的周期数（在这里是一个）。

现在让我们看一下 $O(n)$ 的示例代码。

▼清单2.10。 $O(n)$ 的代码示例

```
1: private bool HasOne(int[] array, int n)
2: {
3:   // 假设数组的长度=n, 包含一些整数值
4:   for (var i = 0; i < n; ++i)
5:   {
6:     var value = array[i]; if
7:       (value == 1)
8:     {
```

```
9:           返回true。
10:      }
11:  }
12: }
```

如果整数数组中存在1，这个函数简单地返回真。如果偶然在数组的开头有一个1，那么这个过程可能会在最快的时间内完成，但是如果数组中的任何地方都没有1，或者在数组的末尾第一次有一个1，那么这个循环将一直进行到最后，所以这个过程将被执行 N 次。这种最坏的情况被表示为 $O(n)$ ，它给出了一个随着数据数量增加的计算复杂性的概念。

现在让我们看一下 $O(n^2)$ 的例子。

▼清单2.11。 $O(n^2)$ 的代码示例

```
1 : 私有的bool HasSameValue(int[] array1, int[] array2, int n) 2
:
3: // 假设数组1和数组2的长度=n，并且包含一些整数值
4: for (var i = 0; i < n; ++i)
5: {
6:     var value1 = array1[i];
7:     for (var j = 0; j < n; ++j)
8:     {
9:         var value2 = array2[j];
10:        if (value1 == value2)
11:        {
12:            返回true。
13:        }
14:    }
15: }
16:
17: 返回错误。
18: }
```

这是一个双循环方法，只有当两个数组中的任何一个包含相同的值时才返回真。最坏的情况是，它们都是不匹配的，在这种情况下，程序将运行 n^2 次。

顺便说一句，计算复杂性的概念只用最大阶数来表示。创建一个方法，对上述例子中的三个方法各执行一次，将导致 $O(n^2)$ 的最大秩序。

(它不可能是 $O(n^2 + n + 1)$)

还应注意的是，计算量只是在数据数量足够大时的一个指导，与实际测量时间没有必然联系，因为在某些情况下，计算量可能显得很庞大，如 $O(n^5)$ ，但当数据数量较少时，就不是问题。

测量处理时间是否足够，考虑到每次的数据数量，同时参考计算量。

我们建议

2.6.2 基本集合和数据结构

C#提供了具有各种数据结构的集合类。本节介绍了在哪些情况下应该采用这些类别，以经常使用的类别为例，并考虑到主要方法的计算复杂性。

这里介绍的集合类中的方法的计算复杂度都可以在MSDN上找到，所以在选择最合适的数据结构时，检查一下比较安全。

列表 $<T>$ 。

最常用的是List $< T >$ 。该数据结构是一个数组。当数据的顺序很重要或者数据经常被索引检索或更新时，它是有效的。相反，如果有大量的元素插入或删除，最好避免使用List $< T >$ ，因为它的计算成本很高，因为需要在操作的索引后复制索引。

此外，当Add试图超过Capacity时，阵列分配的内存会被扩展。当内存被扩展时，会分配两倍的当前容量，所以要用 $O(1)$ 来使用Add，要设置适当的初始值，以便在不引起扩展的情况下使用它。

▼ 表2.6 List<T>。

方法	计算复杂性
添加	$O(1)$ 然而，如果超过了容量， $O(n)$
插入	$O(n)$
IndexOf/Contains 。	$O(n)$
RemoveAt	$O(n)$
分类	$O(n \log n)$

链接列表<T>。

LinkedList< T>的数据结构是一个链表。链接列表是一种基本的数据结构，其中每个节点都有对下一个节点的引用；C#中的LinkedList< T>是一个双向的链接列表，所以它分别有对上一个和下一个节点的引用。LinkedList< T>的元素是它具有强大的添加和删除元素的功能，但不擅长访问数组中的特定元素。当你想创建一个临时持有需要经常添加或删除的数据的进程时，它是合适的。

▼ 表2.7 LinkedList<T>

方法	计算复杂性
添加第一个/最后一个。	$O(1)$
AddAfter/AddBefore。	$O(1)$
删除/删除第一/删除最后。	$O(1)$
包含。	$O(n)$

队列<T>。

Queue< T>是一个实现FIFO（先入先出）方法的集合类。Queue< T>使用循环数组，Enqueue将元素添加到末端，Dequeue将元素从开始删除，而Dequeue将元素从开始删除。Peek是一个检索第一个元素的操作，不删除。TrimExcess是一种减少容量的方法，但从性能调整的角度来看，它不适合于诸如遍历等操作。从性能调整的角度来看，如果能够在使用时首先不增加或减少容量，就可以进一步发挥Queue< T>的优势。

2.6 算法和计算复杂性

▼ 表2.8 队列<T>

方法	计算复杂性
查询	$O(1)$ 然而, 如果超过了容量, $O(n)$
去排队。	$O(1)$
窥视。	$O(1)$
包含。	$O(n)$
修剪多余的 部分	$O(n)$

堆栈[T]。

Stack<T>是一个实现后进先出 (LIFO) 方法的集合类 : Stack<T>被实现为一个数组 : push将一个元素添加到顶部, pop在检索顶部元素时将其删除。Peek是一个取出第一个元素的操作, 但不删除它。堆栈和队列一样, 如果只使用Push和Pop, 可以达到很高的性能。注意不要搜索元素, 注意增加或减少容量。

▼ 表2.9 堆栈<T>。

方法	计算复杂性
推动	$O(1)$ 然而, 如果超过了容量, $O(n)$
流行。	$O(1)$
窥视。	$O(1)$
包含。	$O(n)$
修剪多余的 部分	$O(n)$

字典< TKey, TValue >

到目前为止, 所介绍的集合都是按语义排列的, 而Dictionary< TKey, TValue >是一个专门用于索引的集合类。该数据结构以哈希表 (一种关联数组) 的形式实现。Dictionary< TKey, TValue >的缺点是消耗大量的内存, 但是它的快速引用速度为 $O(1)$, 这一点可以弥补。在不需要枚举和遍历的情况下, 它非常有用, 重点在于引用值。另外, 一定要预先设定好容量。

▼ 表 2.10 Dictionary< TKey, TValue >（字典）。

方法	计算复杂性
添加	$O(1)$ 然而，如果超过了容量， $O(n)$
尝试获取价值	$O(1)$
移除	$O(1)$
包含键	$O(1)$
含有价值	$O(n)$

2.6.3 降低计算量的设备

除了迄今为止介绍的藏品外，还有其他各种藏品。当然，单独用List< T >（数组）也可以实现类似的处理，但选择一个更合适的集合类将优化计算量。简单地在实施方法时意识到计算的复杂性，将有助于避免沉重的处理。作为优化代码的一种方式，为什么不检查一下你所创建的方法的计算复杂性，看看你是否可以使它们的计算强度降低？

设计的手段：记忆化

假设你有一个方法（ComplexMethod），其计算复杂度非常高，必须进行计算。然而，有时不可能降低计算的复杂性。在这种情况下，就会使用一种叫做记忆化的技术。

这里的ComplexMethod是指，给定一个参数后，唯一地返回相应的结果。让我们假设一下。首先，在第一次传递参数时，它要经过一个复杂的过程。第二次和随后的几次，它首先检查是否被缓存，如果被缓存，只返回结果并退出。这样一来，无论第一次的计算量有多大，都有可能在第二次和以后的时间里将计算量减少到 $O(1)$ 。如果事先知道可以传递的参数，就可以在游戏前计算和缓存这些参数，这实际上使处理这些参数的计算时间为 $O(1)$ 。



性能调谐圣经

CHAPTER

03

第3章

プロファイリング
ツール

CyberAgent Smartphone Games & Entertainment

第三章。

剖析工具

剖析工具被用来收集和分析数据，识别瓶颈和确定性能指标。仅仅是Unity引擎就提供了几个这样的工具。还有其他符合本地标准的工具，如Xcode和Android Studio，以及针对GPU的工具，如RenderDoc。因此，了解每个工具的特点并选择适合你的工具是很重要的。本章介绍了这些工具中的每一个，目的是让你使用这些工具并运行起来。

3.0.1 测量时应注意的事项

Unity允许应用程序在编辑器上运行，因此可以在“真实机器”和“编辑器”中进行测量。在进行测量时，有必要牢记每个环境的特点。

在与编辑一起测量时，最大的优势是能够快速尝试和出错。然而，编辑器本身的处理负荷和编辑器使用的内存区域也被测量，所以测量结果中存在很多噪音。另外，由于规格与实际机器的规格完全不同，可能难以确定瓶颈，结果也可能不同。

因此，基本上建议在实际设备上进行剖析测量。然而，只有在“在两种环境下都发生”的情况下，只在编辑器上完成工作才更有效率，因为那里的工作成本更低。大多数时候，这种现象在两种环境中都能再现，但在极少数情况下，它可能只在其中一种环境中再现。因此，首先在实际设备上检查这一现象。接下来，建议在编辑器中也检查一下繁殖情况，然后在编辑器中纠正问题。当然，在这个过程结束时，检查实际设备上的校正情况总是一个好主意。

3.1 统一的程序

Unity Profiler是一个内置于Unity编辑器的剖析工具。这个工具可以逐帧地收集信息。可

以测量的项目范围很广，包括

3.1 统一的程序

每个项目被称为剖析器模块，在Unity 2020的14版中

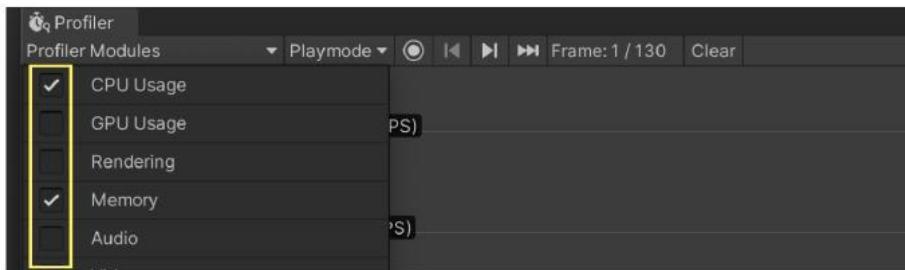
还有一个关于资产和文件I/O的章节。这个模块仍在更新中，在Unity 2021.2中还增加了一个关于资产的新模块和一个关于文件输入输出的模块。因此，由于模块的多样性，Unity Profiler是一个粗略了解性能的好工具。模块列表见图3.1。

モジュール名	説明
CPU Usage	スクリプトやアニメーションなど、CPUが使用した時間の内訳
GPU Usage	オブジェクトのレンダリングなど、GPUが使用した時間の内訳
Rendering	SetPass や Batchingなど描画に関わる情報
Memory	アプリケーション全体でのメモリ割り当てに関する情報
Audio	Audioに関するメモリ割り当て、CPU使用率などの情報
Video	Videoに関するメモリ割り当て、バッファリングフレーム数などの情報
Physics	物理エンジンに関するオブジェクト
Physics2D	2Dの物理エンジンに関するオブジェクトの情報 (RigidBody2D)
Network Messages (非推奨)	マルチプレイヤー高レベルAPI (非推奨)に関する送受信の情報
Network Operations (非推奨)	マルチプレイヤー低レベルAPI (非推奨)に関する送受信の情報
UI	UIに関する処理時間の情報
Global Illumination	UI表示時のパッチ数や頂点数の情報
Virtual Texturing	ライトプローブなど、GIライティングに使用した時間の情報
Asset Loading (2021.2以降)	Texture や Mesh など、ロードタイミングやサイズなどの情報
File Access (2021.2以降)	I/Oに費やした時間など、ファイルアクセスに関わる情報

▲图3.1 剖析器模块列表

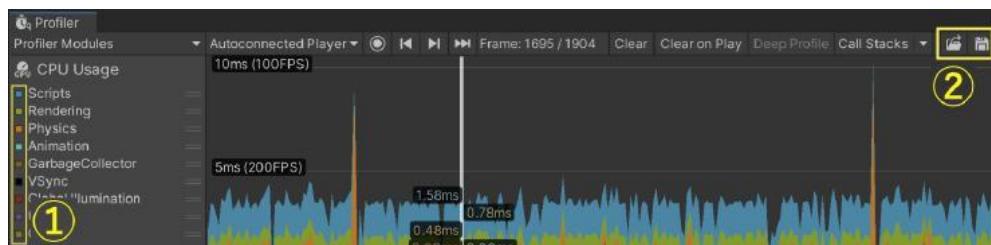
这些模块可以在分析器上显示或不显示。然而，不显示的模块甚至不被测量。反之，如果所有的人都显示出来，那么编辑器就会过载。

第3章 剖析工具



▲图3.2 Profiler模块的显示/隐藏功能。

它还介绍了所有剖析器工具共有的有用功能。



▲图3.3 Profiler功能的解释。

在图3.3中，'①'列出了每个模块所测量的项目。通过点击这个项目，你可以在右手边的时间轴上切换显示和不显示它。只显示必要的项目将使视图更容易阅读。你也可以通过拖动它来重新排列这个项目，右侧的图表就会按这个顺序显示。(2)"是一个用于保存和加载测量数据的功能。如果有必要，你可以保存测量结果。只有显示在剖析器上的数据可以被保存。

在本出版物中，对图3.1中经常使用的CPU用量和内存模块进行了解释。

3.1.1 测量方法

本节介绍了在真实设备上使用Unity Profiler的测量方法。测量过程有两个部分：构建前和应用程序启动后。编辑器中的测量方法只是在执行过程中按下测量按钮，所以细节就省略了。

建设前的任务

预构建任务是启用开发构建。一旦激活，就可以与分析器建立连接。

还有一个选项叫“深度测量”，提供更详细的测量。如果这个选项被启用，所有函数调用的处理时间都会被记录下来，从而更容易识别瓶颈函数。缺点是，测量本身需要非常大的开销，这使得它的速度很慢，并消耗大量的内存。请注意，这意味着即使这个过程看起来花了很长的时间，也可能没有正常情况下的那么长。基本上，它只在正常的配置文件不能提供足够的信息时使用。

如果Deep Profile使用大量的内存，例如在一个大型项目中，由于内存不足，可能无法进行测量。在这种情况下，你别无选择，只能加入自己的测量处理，参考“补充：关于采样器”一节中的“3.1.2 CPU使用”。

这些可以从脚本中明确设置，也可以从GUI中设置。首先介绍从脚本中设置它们的方法。

▼清单3.1。如何从脚本中设置开发构建

```
1: BuildPlayerOptions buildPlayerOptions = new BuildPlayerOptions(); 2:  
/*省略场景和构建目标设置 */  
3:  
4: buildPlayerOptions.options |= BuildOptions.Development;  
5: // 只在你想启用深度剖析模式时添加。  
6: buildPlayerOptions.options |= BuildOptions.EnableDeepProfilingSupport; 7:  
8: BuildReport report = BuildPipeline.BuildPlayer(buildPlayerOptions);
```

清单3.1中重要的一点是指定BuildOptions.Development。

接下来，从GUI进行配置，进入Build Settings，勾选Development Build，如图3.4所示。



图3.4 构建设置。

申请启动后要做的工作

启动应用程序后，有两种方法可以连接到Unity Profiler：“远程连接”和“有线（USB）连接”。远程连接比有线连接有更多的环境限制，配置文件可能不会像预期那样工作。例如，你可能必须连接到同一个Wifi网络，你可能必须禁用仅适用于安卓的移动通信，或者你可能需要腾出其他端口。由于这个原因，本节将重点讨论有线连接，在那里程序简单，剖析可靠。如果你想进行远程连接，你可以在官方文件的帮助下进行尝试。

开始 对于iOS，连接到分析器的方法如下。

1. 从Build设置中把目标平台改为iOS。
2. 将设备连接到电脑上，并启动开发构建应用程序。
3. 在Unity Profiler中选择要连接的终端（图3.5）。
4. 开始记录



图3.5 选择要连接的终端

进行测量的Unity编辑器不一定是你所建立的项目。建议创建一个新的项目进行测量，因为它很轻。

接下来，对于安卓系统，比iOS系统的步骤略多。

1. 从Build设置中把目标平台改为Android。
2. 将设备连接到电脑上，并启动开发构建应用程序。
3. 输入adb forward命令。(该命令的细节见下文)。
4. 从Unity Profiler中选择要连接的终端。
5. 开始记录

adb forward命令需要应用程序的软件包名称。例如，如果软件包名称是”jp.co.sample.app”，输入以下内容。

▼ 清单 3. 2adb forward 命令

```
1: adb forward tcp:34999 localabstract:Unity-jp.co.sample.app
```

如果adb不被识别，请设置adb路径。网络上有很多信息解释如何设置，所以我将跳过这一部分。

作为一个快速的故障排除，如果你不能连接，请检查以下内容

- 两种设备通用
 - 在已执行的应用程序的右下角是否有一个开发构建的符号？
- 适用于安卓系统
 - 终端上的USB调试功能是否启用？
 - 在adb forward命令中输入的软件包名称是否正确？
 - 当输入adb设备命令时，设备是否被正确识别。

作为补充，如果应用程序直接在Build And Run中执行，上述的adb forward命令将在内部执行。因此，测量时不需要命令输入。

自动连接程序

在构建配置中，有一个选项叫做“自动连接程序”（Autoconnect Profiler）。

这个选项用于在应用程序启动时自动连接到编辑器的分析器。因此，它不是剖析的强制性设置。这同样适用于远程剖析：只有 WebGL 不能在没有这个选项的情况下进行剖析，但对于移动端来说，这不是一个非常有用的选项。

再进一步说，如果这个选项被启用，编辑器的IP地址在构建时被写入二进制文件，并在启动时尝试连接到该地址。例如，如果你在一个专用的构建机器上构建，这就没有必要，除非你在该机器上进行剖析。相反，它只会增加应用程序启动时自动连接超时的等待时间（约8秒）。

从脚本中，选项 `BuildOptions.ConnectWithProfiler`

要注意的是，名字是在名称中，很容易被误认为是强制性的。

3.1.2 CPU使用率

CPU 使用情况显示如图3.6所示。

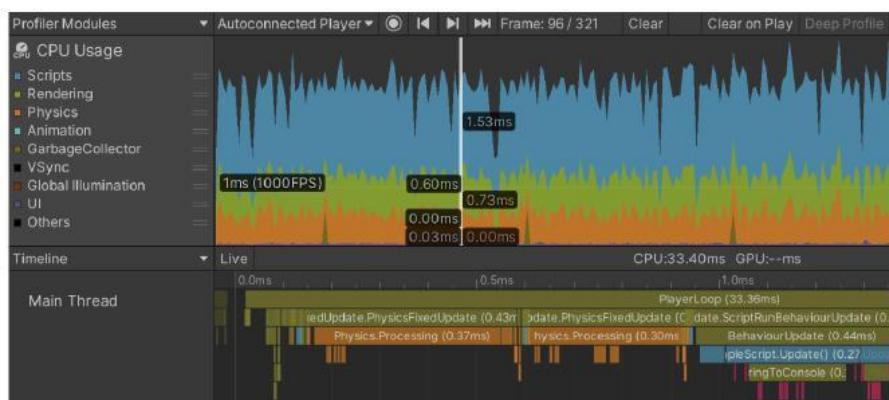


图3.6 CPU用量模块（时间轴显示）

有两种主要的方式来检查这个模块

- 层次结构（原始层次结构）。
- 时间轴。

首先，对“层次结构”视图所显示的内容和如何使用它进行了解释。

1. 层次结构视图

层次结构视图看起来像图3.7。

Hierarchy		Live	Main Thread	CPU:33.46ms GPU:--ms			
		Total	Self	Calls	GC Alloc	Time ms	Self ms
Overview							
▼ PlayerLoop		99.8%	0.1%	1	2.4 KB	33.42	0.06
► WaitForTargetFPS		88.5%	88.5%	1	0 B	29.63	29.62
► PostLateUpdate.FinishFrameRendering		7.7%	0.2%	1	0 B	2.59	0.07
▼ Update.ScriptRunBehaviourUpdate		1.5%	0.0%	1	2.4 KB	0.51	0.00
▼ BehaviourUpdate		1.5%	0.0%	1	2.4 KB	0.51	0.01
► SampleScript.Update()		1.2%	0.2%	2	2.4 KB	0.41	0.08
EventSystem.Update()		0.1%	0.1%	1	0 B	0.04	0.04
DebugUpdator.Update()		0.0%	0.0%	1	0 B	0.02	0.02

图3.7 层次结构视图。

该视图的一个特点是，测量结果以列表形式排列，并可按标题中的项目进行排序。在进行调查时，可以通过从列表中打开感兴趣的项目来确定瓶颈。然而，显示的信息是在“选定的线程”中所花费的时间的指示。例如，如果你使用作业系统或多线程渲染，另一个线程的处理时间就不包括在内。如果你想检查这一点，你可以通过选择一个线程来实现，如图3.8所示。

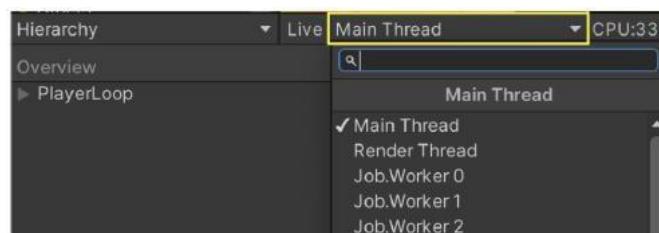


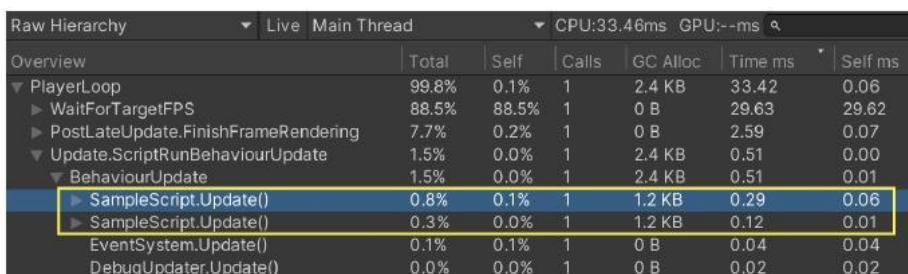
图3.8 线程选择

下一节将介绍标题项。

▼ 表3.1 层次结构标头信息

标题名称	描述。
概述。	样品名称。
共计	处理该功能所需的总时间。(以%显示)
自己	这个函数本身的处理时间。不包括子功能的处理时间。(以%显示)
呼叫。	在一帧内调用的次数。
GC分配。	本函数分配的脚本的堆内存。
时间ms	总的来说是毫秒。
自身毫。	自己在毫秒中。

Calls将几个函数调用合并为一个项目，作为一个视图更容易看到。然而，不清楚是所有的人都有相同的处理时间，还是只有一个人的处理时间更长。原始层次结构视图与层次结构视图不同的是，Calls总是固定在1。在图3.9中，Raw Hierarchy视图显示了对同一个函数的多次调用。



▲图3.9 原始层次结构视图。

总结到目前为止所讲的，层次结构视图用于以下目的

- 识别和优化减慢处理速度的瓶颈（Time ms, Self ms）。
- 理解并优化GC分配（GC Allocation）。

在执行这些任务时，建议在检查前对每个需要的项目进行降序排序。

在打开物品时，往往会有个很深的层次结构。在这种情况下，在Mac上按住Option键（或在Windows上按住Alt键），同时打开一个项目，就可以打开整个层次结构。反之，在按住键的同时关闭一个项目，就会关闭它下面的层次结构。

2. 时间轴视图

另一种检查时间线视图的方法如下。

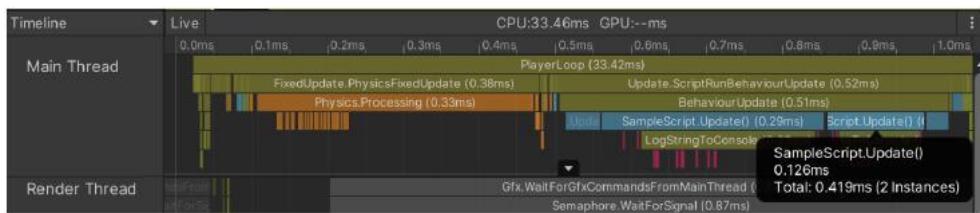


图3.10 时间轴视图。

在时间线视图中，层次结构视图中的项目被可视化为方框，因此，在观察整个画面时，你可以直观地看到负载的位置，一目了然。而且，由于它是鼠标可及的，即使是很深的层次结构也可以拖动，以获得一个完整的概览。此外，在时间轴上，不需要在线程之间切换，它们都显示出来了。这使得我们很容易看到每个线程中正在做什么以及什么时候做。由于这些特点，它主要用于以下目的

- 我想了解一下处理负荷的总体情况。
- 想了解并调整每个线程的处理负荷

时间线不适合用于排序操作，以确定繁重的处理顺序，或用于检查分配的总量。因此，Hierarchy视图更适合于调整分配。

补充信息：关于采样器

有两种方法可以在每个功能的基础上衡量处理时间。
深度模型模式。另一种方法是直接嵌入到脚本中。

如果直接嵌入，请说明如下。

▼ 使用清单3. 3Begin/EndSample的方法

```
1: using UnityEngine.Profiling;
2: /* ... 缩略语... */
3: private void TestMethod()
4: {
5:     for (int i = 0; i < 10000; i++)
6:     {
7:         Debug.Log("Test")
8:     }
9: }
10:
11: private void OnClickedButton()
12: {
13:     Profiler.BeginSample("Test Method")
14:     .
15:     TestMethod();
16:     Profiler.EndSample();
}
```

嵌入的样本在层次结构和时间线视图中都有显示。

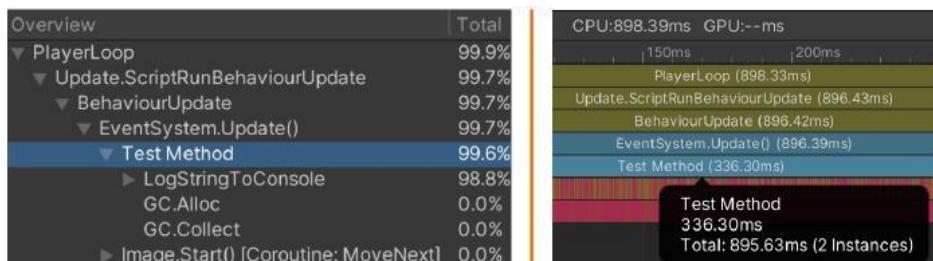


图3.11 采样器显示。

还有一个值得一提的特点。如果剖析代码不是Development Build，开销为零，因为调用者被禁用。在未来加工负荷可能增加的地区进行预建可能是一个好主意。

BeginSample方法是一个静态函数，可以很容易地使用，但也有一个具有类似功能的CustomSampler。这是从Unity 2017开始添加的，比BeginSample的测量开销更少，这意味着可以测量更准确的时间。

3.1 统一的程序

▼ 清单3.4 使用 CustomSampler的方法

```
1: using UnityEngine.Profiling;
2: /* ... 缩略语... */
3: private CustomSampler _samplerTest = CustomSampler.Create("Test");
4:
5: private void TestMethod()
6: {
7:     for (int i = 0; i < 10000; i++)
8:     {
9:         Debug.Log("Test");
10:    }
11: }
12:
13: private void OnClickedButton()
14: {
15:     _samplerTest.Begin();
16:     TestMethod();
17:     _samplerTest.End();
18: }
```

不同的是，你需要提前创建实例；自定义采样器还允许你在测量后在脚本中获得测量时间。如果你需要更多的精度，或者你想根据处理时间发出警告，CustomSampler是一个不错的选择。

3.1.3 记忆

内存模块的显示如图3.12所示。

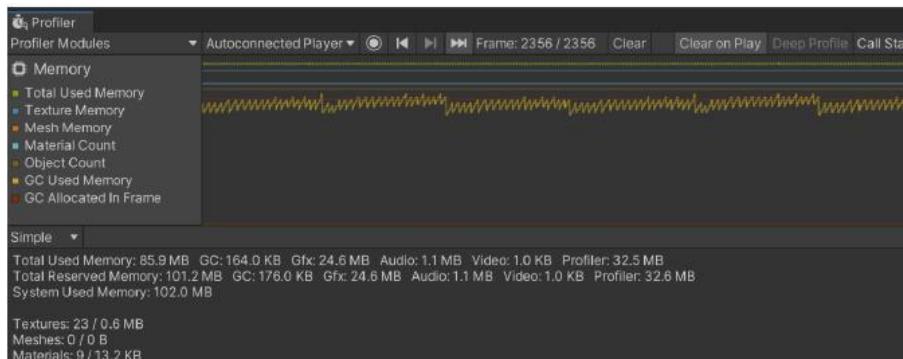


图3.12 内存模块。

有两种方法来检查这个模块

- 简单的观点
- 详细查看

首先，本节解释了显示的内容和如何使用简单视图。

1. 简单的观点

简单视图看起来像图3.13。

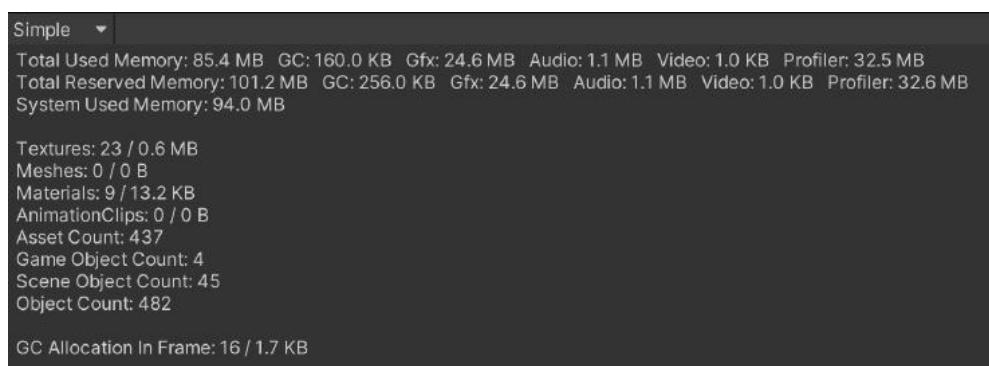


图3.13 简单视图。

本节介绍视图中列出的项目。

已用内存总量

Unity分配（使用中）的内存总量。

保留的总内存

目前Unity保留的内存总量；在操作系统端提前保留一定量的连续内存区域作为池，在需要时再分配。当池子的面积不够时，会再次向操作系统申请并扩大。

系统使用的内存

应用程序使用的总内存量。这个项目也衡量在总保留中没有衡量的项目（如插件）。然而，这仍然不能跟踪所有的内存分配。如果你想得到一个准确的情况，你需要使用一个本地兼容的剖析工具，如Xcode。

图3.13中总使用内存右边列出的项目的含义如下。

3.1 统一的程序

▼ 表3.2 简单视图术语表

术语	描述。
ĀĀĀ	在堆区使用的内存量，由于GC Alloc等因素而增加。
Gfx	纹理、着色器、网格等分配的内存数量。
音频	用于音频播放的内存量。
视频。	视频 用于播放的内存数量。
检察官 。	用于剖析的内存量。

作为术语名称的补充说明，从Unity 2019.2开始，“Mono”已被改名为“GC”，“FMOD”已被改名为“Audio”。

图3.13还显示了使用的其他资产的数量和分配的内存量，如下图所示。

- 纹理
- 网络。
- 材料
- 动画剪辑
- 音频剪辑

还有关于对象数量和GC分配的信息，如下所示。

资产计数

装载的资产总数。

游戏对象数量

场景中存在的游戏对象的数量。

场景对象计数

场景中存在的组件、游戏对象等的总数量。

对象数量

由应用程序生成和加载的所有对象的总数。如果这个值在增加，很可能是一些对象在泄漏。

框架中的GC分配

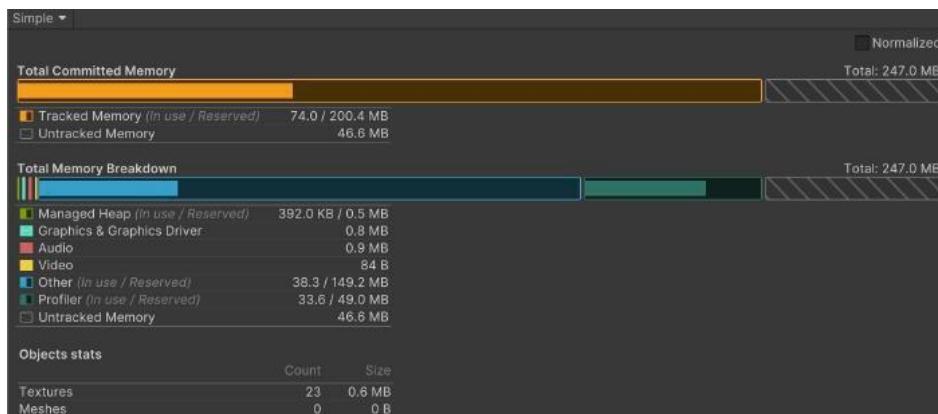
在一个框架内发生分配的次数和总量。

最后，从这些信息中总结出简单视图的用例。

- 了解和监测堆积面积和保留的扩展时间
- 检查各种资产和物体的泄漏情况。

- GC分配监测

Unity 2021及以后版本的“简单”视图的用户界面有了很大的改进，使其更容易看到显示的项目。内容本身没有大的变化，所以你仍然可以使用这里介绍的知识。然而，请注意，有些名字已经被改变。例如，GC已被重新命名为管理堆。



▲ 图3.14 从2021年起的简单视图。

2. 详细查看

Detailed视图看起来像图3.15。

Name	Memory	Ref count	Referenced By:
▶ Other (81)	44.9 MB		
▼ Assets (420)			
▶ AudioManager (1)	1.5 MB		
▼ Shader (7)	1.1 MB		
Hidden/Internal-GUI/RoundedRectWithColorPerBorder	233.3 KB		
Hidden/Internal-Procedural	45.2 KB	2	
Hidden/Internal-GUI/RoundedRect	44.8 KB	1	
Hidden/Internal-GUI/RoundedRect	40.5 KB	2	
▶ Sprites/Default	32.2 KB	3	
Hidden/Internal-GUITextureClip	31.9 KB	3	
Hidden/Internal-GUITexture	24.0 KB	3	
Hidden/BlitCopy	14.6 KB	2	
▶ MonoScript (396)	135.9 KB		

图3.15 详细视图

3.1 统一的程序

按 "取样" 按钮，可以把这个显示的结果作为当时的记忆快照；与 "简单" 视图不同，这不是实时更新，所以如果你想刷新显示，你需要再次取样。

在图3.15的右侧，有一个名为 "引用者" 的项目。这显示了引用当前所选对象的对象。如果有资产泄漏，关于对象的参考信息可能有助于解决问题。这个显示只有在 "收集对象参考" 被启用时才会显示。启用该功能会增加取样时的处理时间，但一般建议不启用。

你可能会看到ManagedStaticReferences()这个符号来表示Referenced By。这意味着它被某个静态对象所引用。如果你熟悉这个项目，这些信息可能足以让你有所了解。如果没有，建议使用3.5堆栈浏览器。

详细视图中的标题项目在此不作解释，因为它们的含义与看上去的含义相同。操作方法与 "3.1.2 CPU使用情况" 中的 "1. 分层视图" 相同，对每个标题有排序功能，对项目有分层视图。本节对 "名称" 项目中显示的顶部节点进行解释。

▼ 表3.3 详细的顶级节点

名称。	描述。
资产	场景中不包括已加载的资产。
未获救。	由代码在运行时生成的资产。 一个由代码生成的对象，例如，new Materiala()。
场景记忆	包括在加载场景中的资产。
其他。	对上述对象以外的、被Unity在不同系统中使用的对象的赋值。

在顶级节点中，"其他" 项下所列的项目可能对你来说并不熟悉。其中有些是值得了解的，接下来将讨论。

系统.可执行程序和文件

表示用于二进制文件、 DLLs等的分配数量。它可能不适用于某些平台或终端，在这种情况下，它被当作0B处理。项目的内存负荷并不像所述值那么大，因为它可能与其他使用共同框架的应用程序共享。与其急于减少这一项，还不如改善资产。最有效的方法是减少DLLs和不必要的Scripts。最简单的方法是改变剥离水平。然而，在运行时有遗漏类型和方法的风险，所以要仔细调试。

串行化文件

表示元信息，如AssetBundle中的对象表或作为类型信息的类型树。这可以通过AssetBundle.Unload(true or false)释放。最有效的方法是使用Unload(false)在资产加载后只释放这些元信息，但要注意，如果释放时机和资源引用管理做得不仔细，资源会被重复加载，容易发生内存泄漏。

PersistentManager.Remapper.

它管理内存中的对象和磁盘上的对象之间的关系；Remapper使用一个内存池，当它用完时，内存池会以翻倍的方式扩展，不会减少。应注意不要过度膨胀。具体来说，如果加载了大量的AssetBundles，映射区域将不够用，会被扩大。因此，卸载不必要的AssetBundles是一个好主意，以减少同时加载的文件数量。如果一个资产包包含大量不需要的资产，将其拆分也是一个好主意。

最后，我们总结了到目前为止所介绍的使用详细视图的案例。

- 详细了解和调整特定时序下的内存
 - 没有不需要的或意外的资产
- 调查内存泄漏。

3.2 档案分析器

Profile Analyzer是对Profiler的CPU Usage获得的数据进行更详细分析的工具：Unity Profiler只允许你查看每一帧的数据，Profile Analyzer允许你根据指定帧的间隔获得平均数、中位数、最小和最大值。值、中位数、最小值和最大值，基于指定的框架区间。这可以让你适当地处理每一帧不同的数据，这样你就可以在优化时更清楚地显示改进的效果。它也是一个非常有用的工具，用于比较和可视化优化的结果，因为它有一个测量数据之间的比较功能，而CPU Usage无法做到。

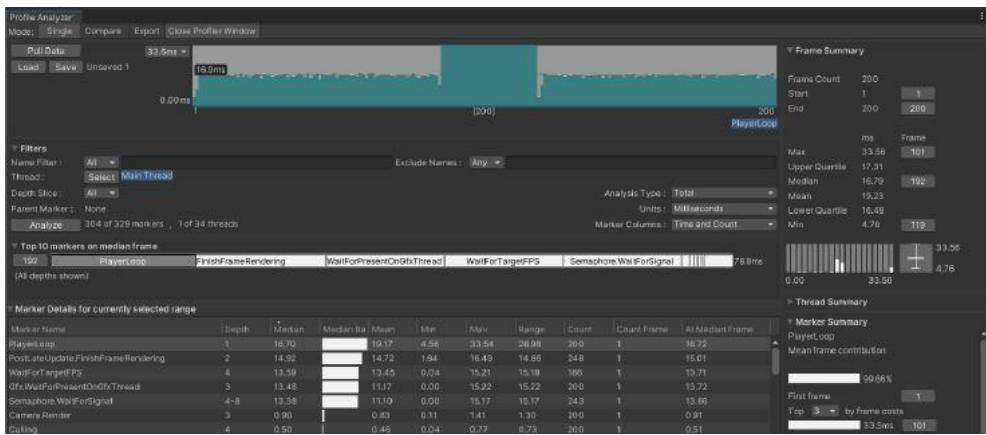


图3.16 Profile分析仪。

3.2.1 介绍的方法

该工具可以从Unity官方支持的Package Manager中安装，将Packages改为Unity Registry，在搜索框中输入'Profile'。安装后，可以通过“窗口->分析->模板分析器”来启动该工具。



图3.17 从PackageManager安装

3.2.2 如何操作

启动后，Profile Analyzer立即看起来像图3.18。

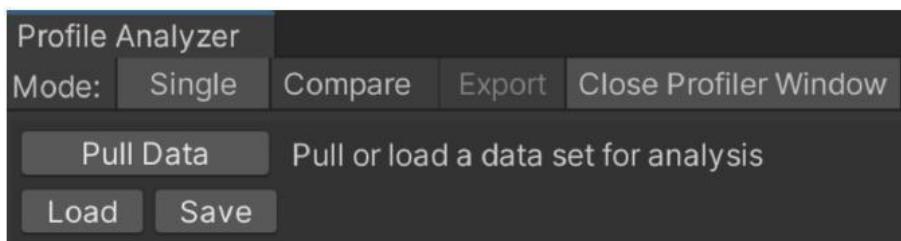


图3.18 开机后立即投入使用

有两种功能模式：“单一”和“比较”；单一模式用于分析单一测量，而比较模式用于比较两个测量数据。

“拉数据”允许你分析用Unity Profiler测量的数据并显示结果。提前用Unity Profiler测量。

“保存”和“加载”允许你保存和加载Profile Analyzer所分析的数据。当然，如果你只保留Unity Profiler中的数据，那就没有问题。在这种情况下，你必须在Unity Profiler中加载数据，并在Profile Analyzer中每次提取数据。如果该程序很麻烦，最好将数据保存为专用数据。

3.2.3 分析结果（单模式）

分析结果屏幕的结构如下。标记这个词出现在这里，指的是进程的名称（方法名称）

。

- 分析部分设置屏幕。
- 显示项目过滤器输入屏幕。
- 标记的中值 前10名
- 标志物分析结果。
- 框架摘要
- 主题摘要
- 所选标记的摘要

让我们来看看每一个显示屏幕。

1. 分析部分设置屏幕。

每一帧的处理时间被显示出来，所有的帧都被初步选中。如图3.19所示，帧间隔可以通过拖动来改变，必要时可以调整。



图3.19 框架部分的指定

2. 过滤器输入屏幕

过滤输入屏幕允许对分析结果进行过滤。

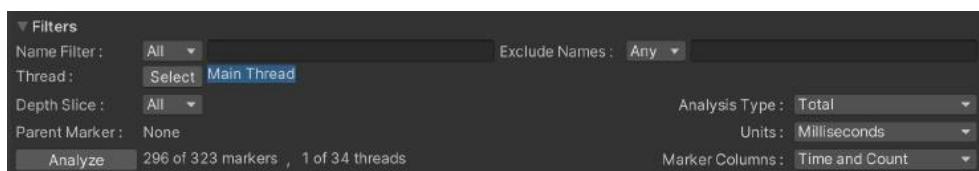


图3.20 过滤器输入屏幕。

每项内容如下。

▼ 表3.4 过滤器条目。

项目名称	描述。
名称过滤器	通过你要搜索的过程的名称进行过滤。
排除过滤器	通过你想从搜索中排除的过程的名称进行过滤。
课题。	选择的线程会显示在分析结果中。 如果你需要，可以在其他线程上添加信息。
深度切片	在CPU使用率中介绍的层次结构显示的数量。 例如，如果深度是3，就会显示第三个层次。
分析类型	CPU使用情况 允许在总数和自身之间切换，如在标题项目中发现的那样。
单位。	时间显示可以改成毫秒或微秒。
标记栏。	可以改变分析结果的标题显示。

当深度切片设置为全部时，就会显示名为PlayerLoop的顶层节点，或者显示同一进程的不同层，这样就很难看清。在这种情况下，建议将深度固定为2~3，并设置为显示渲染、动画和物理等子系统。

3. 标记的中值 前10名

这个屏幕按照中值对每个标记的处理时间进行排序，只显示前10个标记。

下表显示。你可以一目了然地看到前十个标记中的每一个占用了多少处理时间。



▲ 图3.21 前10个中位数的标记物

4. 标志物分析结果。

每个标志物的分析结果都会显示出来；你不妨根据标志物名称下所列的治疗名称以及中位数和平均值来分析要改进的治疗。如果你把鼠标指针移到一个标题项目上，就会显示这个项目的描述，这样你就可以在不确定它的内容时参考它。

Marker Name	Depth	Media	Mean	Min	Max	Range	Count	Count Freq	At Median F
PlayerLoop	1	16.71	19.24	4.56	33.54	28.98	199	1	16.71
PostLateUpdate.FinishFrameRendering	2	14.93	14.80	1.73	16.49	14.76	247	1	14.83
WaitForTargetFPS	4	13.59	13.53	0.04	15.21	15.18	185	1	13.35
Gfx.WaitForPresentOnGfxThread	3	13.49	11.22	0.00	15.22	15.22	199	1	13.35
Semaphore.WaitForSignal	4-8	13.38	11.16	0.00	15.17	15.17	242	1	13.28
Camera.Render	3	0.90	0.82	0.11	1.41	1.30	199	1	1.16
Culling	4	0.50	0.46	0.04	0.77	0.73	199	1	0.69
SceneCulling	5	0.35	0.33	0.02	0.62	0.60	199	1	0.52
PostLateUpdate.ProfilerEndFrame	2-3	0.30	0.30	0.13	1.10	0.97	247	1	0.33
Profiler.FlushCounters	3	0.28	0.26	0.03	1.10	1.07	199	1	0.33
PrepareSceneNodes	6	0.27	0.25	0.01	0.53	0.52	199	1	0.43
Profiler.FlushMemoryCounters	4	0.22	0.20	0.02	1.08	1.06	199	1	0.23

▲ 图3.22 每个处理的分析结果

平均数和中位数

平均值是将所有数值相加后除以数据数得到的数值。与此相反。

中位数是排序后的数据中间的数值。如果有偶数个数据，则从中位数前后的数据中取平均值。

平均数有受到数值相差极大的数据影响的倾向。如果经常出现尖峰或采样数量不足，参考中位数可能更好。

图 3.23 显示了一个中位数和平均值之间存在巨大差异的例子。



▲ 图3.23 中位数和均值

在了解这两个数值的特点后，分析数据。

5. 框架摘要

该屏幕显示测量数据的帧统计。

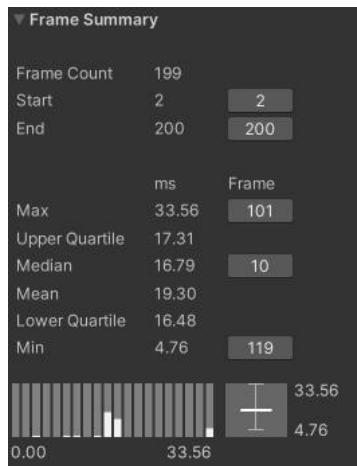


图3.24 框架摘要屏幕。

关于被分析的帧的间隔和数值的变化程度的信息是用boxplots和柱状图显示的。箱形图需要对四分位数的理解。四分位数是定义值，数据排序如表3.5所示。

▼ 表3.5 四分位数

名称。	描述。
最小值 (Min)	闵行区。
下四分位数	价值在最低价值的25%处
中位数 (中位数)	价值在最低值的50%处
上四分位数	从最低值开始的75%位置的价值
最大值 (Max)	最大值。

25%和 75%之间的区间被框住了，这就叫盒须图。

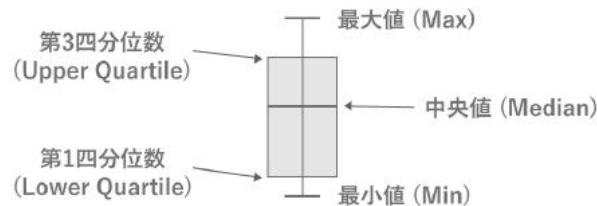


图3.25 箱形和胡须图

直方图在横轴上显示处理时间，在纵轴上显示数据数量，这对于查看数据分布也很有用。在帧摘要中，将光标悬停在帧上就可以查看间隔和帧数。

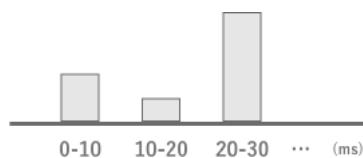


图3.26 直方图。

最好是了解如何看待这些图表，然后分析其特点。

6. 主题摘要

该屏幕显示所选线程的统计数据。可以查看每个线程的箱形和胡须图。

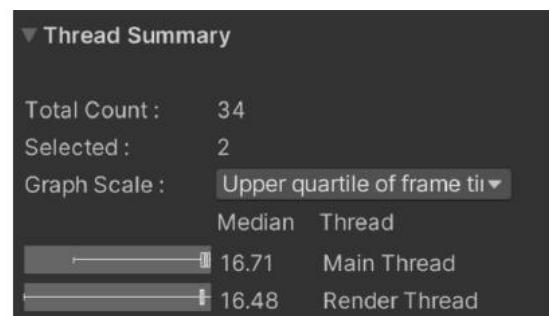


图3.27 框架摘要屏幕。

7. 所选标记的摘要

4. 在‘4. 标记分析结果’屏幕中选择的标记的摘要。当前所选标记的处理时间以盒须图或直方图的形式显示。

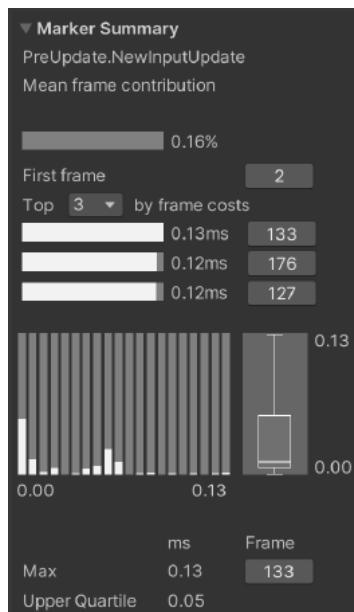


图3.28 选择中的标志物摘要

3.2.4 分析结果（比较模式）。

在这种模式下，两组数据可以进行比较。可以为每个上层和下层数据设置要分析的间隔。

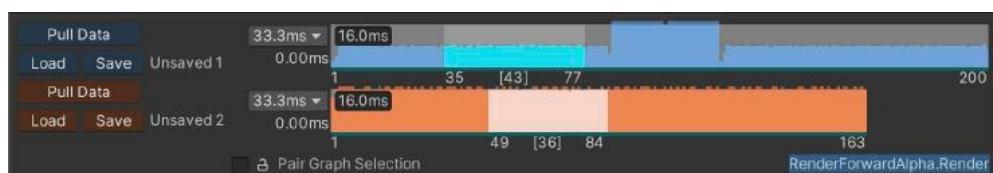


图3.29 设置比较数据

3.3 框架调试器

屏幕的使用与单人模式基本相同，但“左”和“右”字会出现在不同的屏幕上，如图3.30所示。

Marker Name	Left Median	<	Right Median	Diff	Abs Diff	Col
Gfx.WaitForGfxCommandsFromMainThread	0.66		31.85	31.19	31.19	76
WaitForTargetFPS	13.90		32.44	18.55	18.55	43
Semaphore.WaitForSignal	14.64		31.85	17.21	17.21	121
PlayerLoop	16.70		33.29	16.59	16.59	43
PostLateUpdate.FinishFrameRendering	15.18		0.40	-14.78	14.78	43
Gfx.PresentFrame	14.29		0.10	-14.20	14.20	43

▲ 图3.30 标记的比较

这表明哪个数据是哪个，并与图3.29中显示的颜色一致；左边是来自顶部的数据，右边是来自底部的数据。这种模式将使我们更容易分析出好的和坏的调谐结果。

3.3 框架调试器

帧调试器是一个工具，它允许你分析当前显示的屏幕是如何被渲染的。该工具默认安装在编辑器中，可以通过选择“窗口->分析->框架调试器”打开。

它可以在编辑器或实际设备上使用。当在真正的机器上使用时，如Unity Profiler需要一个用“开发构建”构建的二进制文件。启动应用程序，选择设备连接点，并按“启用”来显示绘图说明。

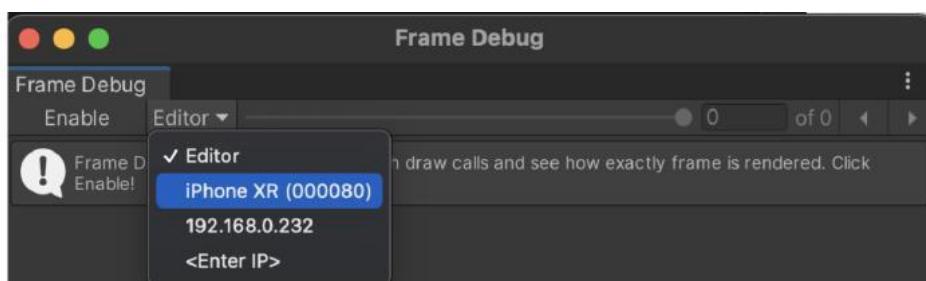


图3.31 FrameDebugger连接屏幕。

第3章 剖析工具

3.3.1 分析屏幕

按“启用”，你将看到以下屏幕。

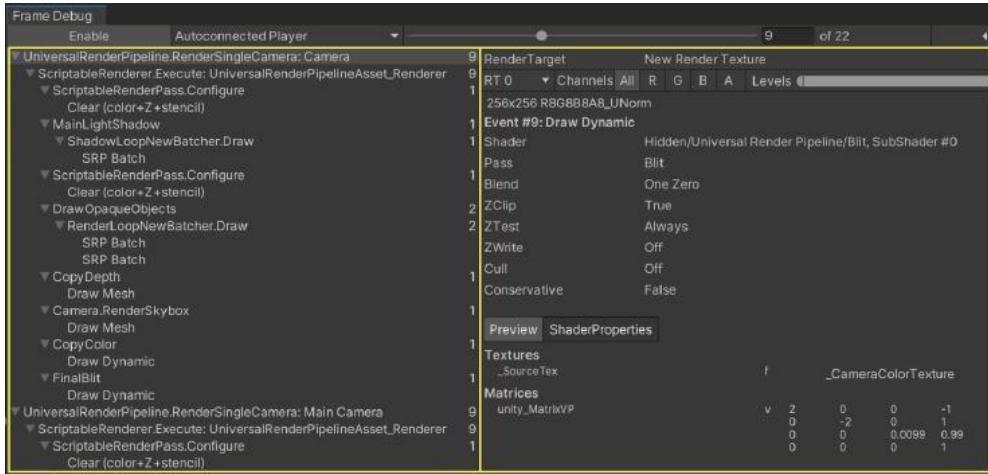


图3.32 FrameDebugger捕获。

左边的框架显示一个项目为一个绘图指令，指令从上往下依次发出。右边的框架显示关于绘图指令的详细信息。你可以看到哪些着色器被处理，以及哪些属性。在看这个屏幕的时候，要注意进行以下的分析。

- 是否有任何不必要的命令？
- 图纸批改没有适当的效果或无法总结
- 绘图目标的分辨率是否太高？
- 你是否使用了一个非预期的着色器？

3.3.2 详细屏幕

上一节介绍的图3.32中的右边框详细解释了。

控制面板

首先，让我们来谈谈上部的控制面板。

3.3 框架调试器



▲ 图3.33 上部操作面板

当存在多个渲染目标时，RT0可以被改变。通道允许你改变是显示所有的RGBA还是只显示其中一个通道。等级是一个滑块，允许你调整绘图结果的亮度。这是非常有用的，例如，当渲染是黑暗的，如在环境或间接照明下，调整亮度，使其更容易看到。

绘图纲要

在这个区域，你可以找到关于图纸目的地的分辨率和格式的信息。很明显，如果有一个分辨率更高的渲染器，你会立即注意到。你还可以看到其他信息，如正在使用的着色器的名称、Cull等通行证设置以及正在使用的关键词。底部的句子'为什么这样~'描述了为什么这幅画不能被批改。在图3.34的情况下，它指出批处理是不可能的，因为第一个绘图调用被选中。由于原因描述得如此详细，如果你想设计一个批处理过程，你可以依靠这些信息来调整批处理。

```
824x1210 B10G11R11_UFloatPack32
Event #13: SRP Batch
Draw Calls           3
Shader               Universal Render Pipeline/Lit, SubShader #0
Pass                 ForwardLit (UniversalForward)
Keywords             _MAIN_LIGHT_SHADOWS
Blend                One Zero
ZClip                True
ZTest                LessEqual
ZWrite               On
Cull                 Back
Conservative        False

Why this draw call can't be batched with the previous one
SRP: First call from ScriptableRenderLoopJob
```

▲图3.34 中间部分的绘图概述。

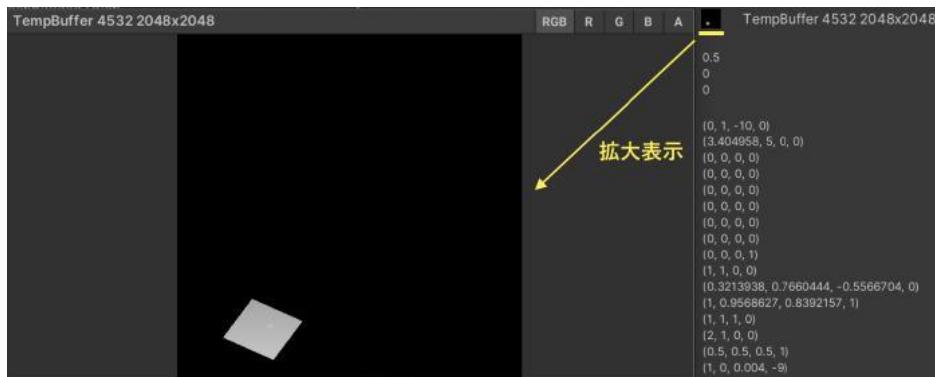
关于着色器属性的更多信息。

这个区域包含正在绘制的着色器的属性信息。这对调试是很有用的。

Preview			ShaderProperties		
Textures					
unity_SpecCube0	f				
_BaseMap	f				UnityWhite
_MainLightShadowmapTexture	f				TempBuffer 1 2048x2048
FLOATS					
_Smoothness	f	0.5			
_Metallic	f	0			
_Surface	f	0			
Vectors					
_WorldSpaceCameraPos	vf	(0, 1, -10, 0)			
unity_OrthoParams	v	(2.310268, 5, 0, 0)			

▲ 图3.35 关于下层着色器属性的详细信息。

有时你可能想详细检查属性信息中显示的Texture2D处于什么状态。要做到这一点，在Mac上按住Command键（在Windows上按住Control键），然后点击图像进行放大。



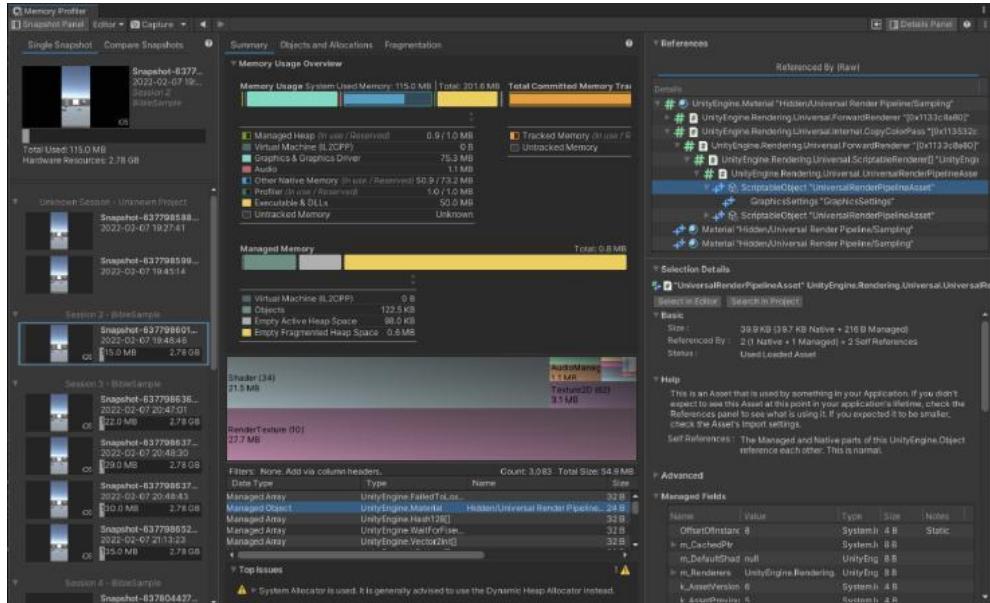
▲ 图3.36 放大Texture2D预览

3.4 储存器

Memory Profiler是Unity作为预览包提供的官方工具，与Unity Profiler的Memory模块相比，它有以下主要优势

- 捕获的数据与屏幕截图一起存储在本地。
- 可视化和易于理解的每个类别的内存占用情况
- 数据可以进行比较。

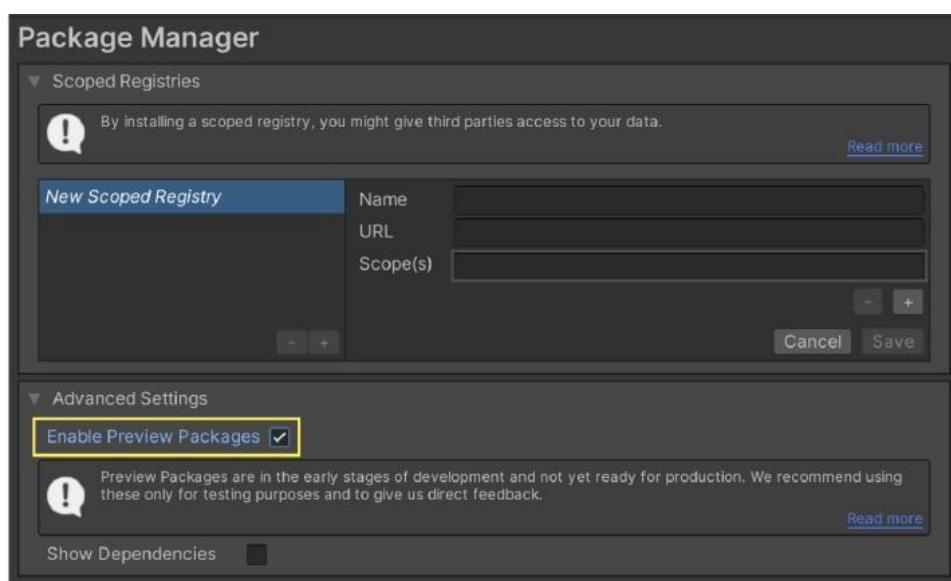
从0.4版本开始，Memory Profiler的用户界面发生了重大变化。本书使用0.5版，这是写作时的最新版本。对于0.4及以上版本，需要Unity 2020.3.12f1或更高版本来使用所有功能。此外，v0.4和v0.5乍一看是一样的，但v0.5在功能上有很大更新。特别是，现在的对象引用非常容易遵循，所以基本上推荐使用v0.5或更高版本。



▲ 图3.37 内存管理器。

3.4.1 介绍的方法

在Unity 2020中，预览版本的软件包必须在“项目设置->软件包管理器”中启用“启用预览软件包”。



▲图3.38 激活预览包

然后从Unity注册表的软件包中安装Memory Profiler。安装后，该工具可以通过“窗口->分析->内存探测器”启动。

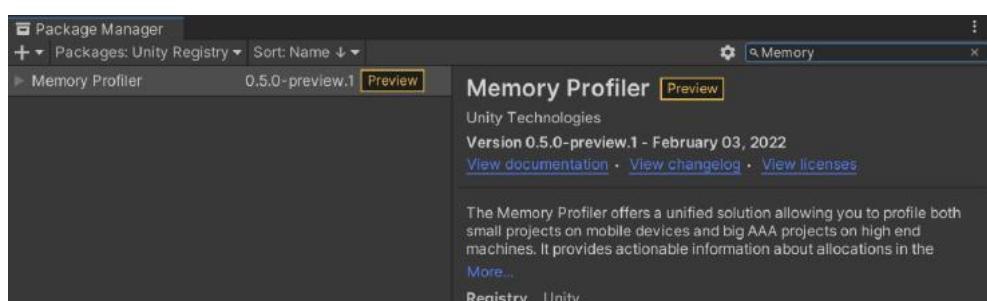
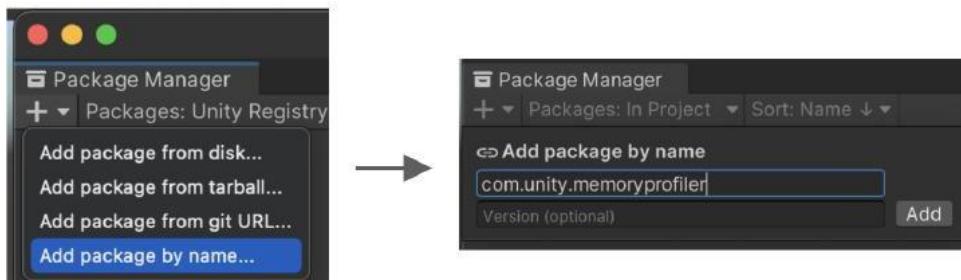


图3.39 从PackageManager安装

3.4 储存器

另外，从Unity 2021开始，添加软件包的方式已经改变。要添加一个软件包，你需要按“按名称添加软件包”，并输入“com.unity.memoryprofiler”。



▲ 图3.40 2021年后如何添加

3.4.2 如何操作

记忆程序由四个主要元素组成。

- 工具栏
- 快照小组
- 测量结果
- 详见面板

将对这些领域中的每一项进行解释。

1. 工具栏

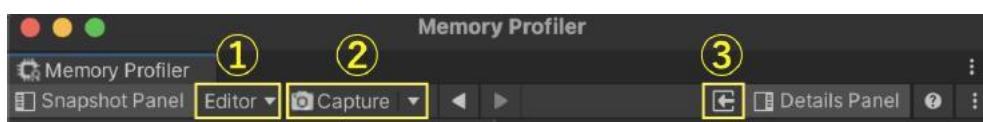


图3.41 工具栏区域。

图3.41显示了头的捕捉。按钮（1）允许你选择测量目标。按键（2）在按下时测量记忆。可选的是，测量目标可以设置为只针对本地对象，或者禁用屏幕截图。使用基本的默认设置不会有错误。按钮③按下后可以读取测量数据。通过按“快照面板”或“细节面板”，你可以显示或隐藏左侧或右侧的信息面板。你也可以点击“？”来打开官方文档。

关于测量，有一个注意事项：测量所需的内存是新分配的，此后不再释放。关于测量要注意的一点是，测量所需的内存是新分配的，不会再被释放。然而，它不会无限制地增加，经过几次测量后最终会稳定下来。在测量时分配的内存量将取决于项目的复杂性。请注意，如果你不知道这个假设，当你看到内存使用量膨胀时，你可能会错误地认为有泄漏。

2. 快照小组

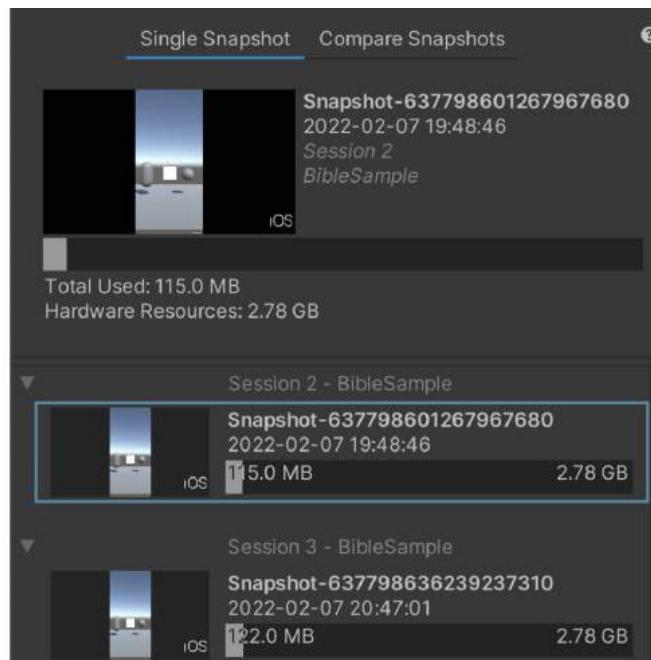


图3.42 快照面板（单件）

3.4 存储器程序

快照面板显示测量数据，并允许你选择查看哪些数据。这些数据是按会话分组的，从应用开始到会话结束。你还可以通过右键点击删除或重命名测量的数据。

在顶部是'单一快照'和'比较快照'，比较快照。

按一下会改变显示方式，变成一个用于比较测量数据的用户界面。

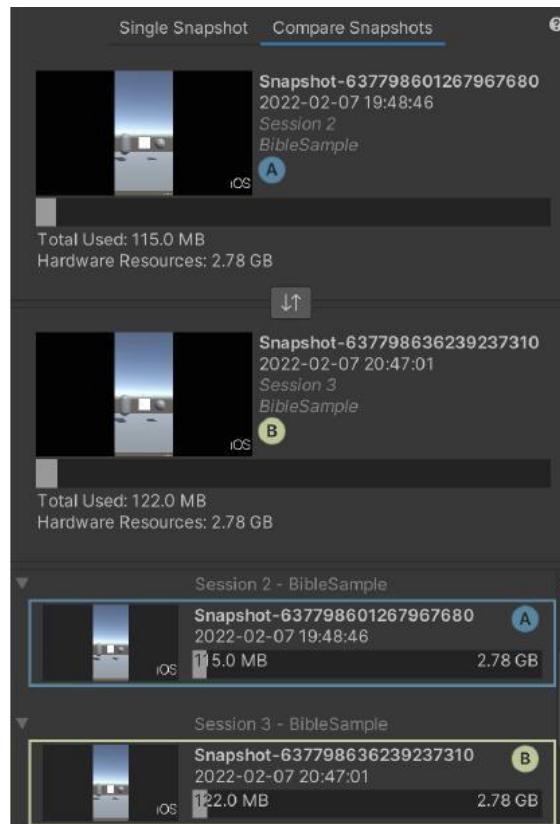


图3.43 快照面板 (Comapre)。

'A' 是单一快照中选择的数据，'B'是组合快照中选择的数据。你可以通过按交换按钮在 "A" 和 "B" 之间进行切换，所以你可以在它们之间进行切换，而不必一直回到 "单一快照" 屏幕。

3. 测量结果

测量结果显示在三个标签中：摘要、对象和分配以及碎片化。摘要屏幕的上半部分被称为“内存使用概况”，显示了当前内存的概况。点击一个项目会在细节面板上显示解释，所以检查你不明白的项目是一个好主意。

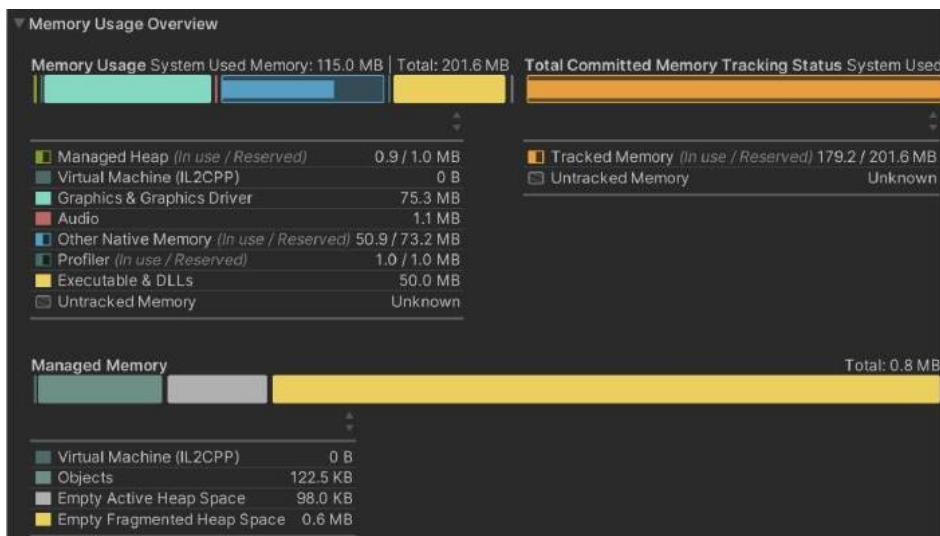


图3.44 内存使用概况

屏幕的下一部分被称为“树状图”，以图形方式显示每一类对象的内存使用情况。选择每个类别，你就可以看到该类别中的对象。在图3.45中，Texture2D类别被选中。

3.4 存储器程序



▲ 图3.45 树木图。

然后，屏幕的下半部分被称为树状图表。在这里，对象的列表以表格形式排列。通过按下树状图表的标题，可以对显示的项目进行分组、排序和过滤。



▲ 图3.46 头部操作。

特别是，对类型进行分组使其更容易分析，应积极主动地使用。

Data Type	[Type]	Name	Size	Referenced By
Native Obj...	▶ AudioListener (2)		432 B	2
Native Obj...	AudioManager (1)	AudioManager	1.1 MB	0
Native Obj...	BoxCollider (1)	Cube	256 B	1
Native Obj...	BuildSettings (1)	BuildSettings	0.6 KB	0
	▶ Camera (2)		8.5 KB	4

▲ 图3.47 按类型分组。

第3章 剖析工具

另外，如果在树状图中选择了一个类别，就会自动设置一个过滤器，只显示该类别的对象。

Filters:		Count: 34 Total Size: 21.5 MB	
Data Type	Type	Name	Size
Native Obj...	Shader	Hidden/Universal Render Pipeline...	19.1 MB
Native Obj...	Shader	Hidden/Universal Render Pipeline...	417.1 KB
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.8 KB
Native Obj...	Shader	Hidden/Universal Render Pipeline...	217.2 KB
Native Obj...	Shader	Hidden/Universal Render Pipeline...	211.3 KB
Native Obj...	Shader	Hidden/Universal Render Pipeline...	195.4 KB
Native Obj...	Shader	Universal Render Pipeline/Lit	181.4 KB

图3.48 自动过滤器设置。

最后，描述了使用比较快照时的用户界面变化：内存使用概述显示了每个对象的差异。

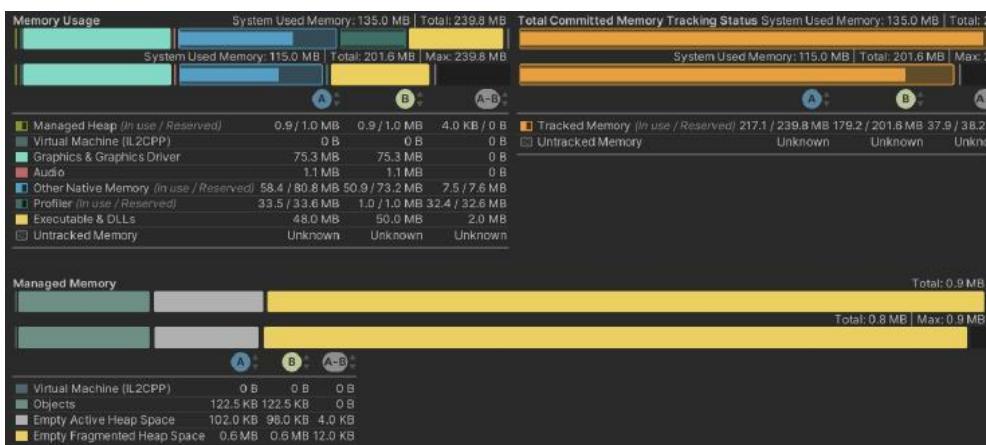


图3.49 比较快照的内存使用概况。

树状图表还为标题添加了一个Diff条目，它有以下类型

3.4 储存器

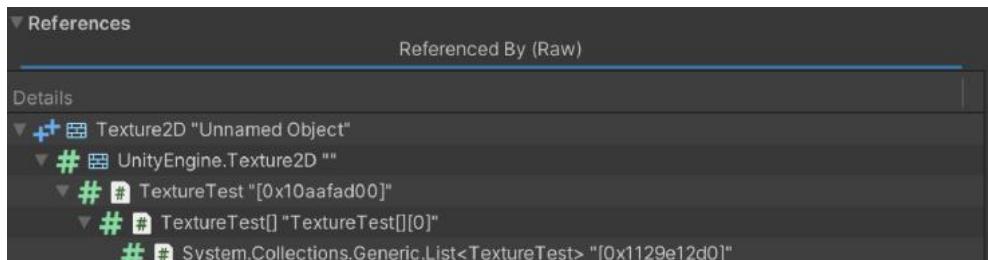
▼ 表3.6 树状图表（比较）

扩散。	描述。
一样。	A, B 相同的物体
不在A区（已删除）。	A中的物体, 但B中没有
不在B区（新）。	物体不在A处而在B处

你可以通过查看这些信息来检查内存是否在增加或减少。

4. 详见面板

当你想跟踪所选对象的参照关系时, 就会用到这个面板。通过检查这个引用者, 你将能够确定持续引用抓取的原因。



▲图3.50 参考文献。

底部的“选择细节”部分包含了关于该对象的详细信息。除其他事项外, 帮助部分还包含关于如何发布的建议。如果你不确定该怎么做, 不妨阅读一下。

第3章 剖析工具

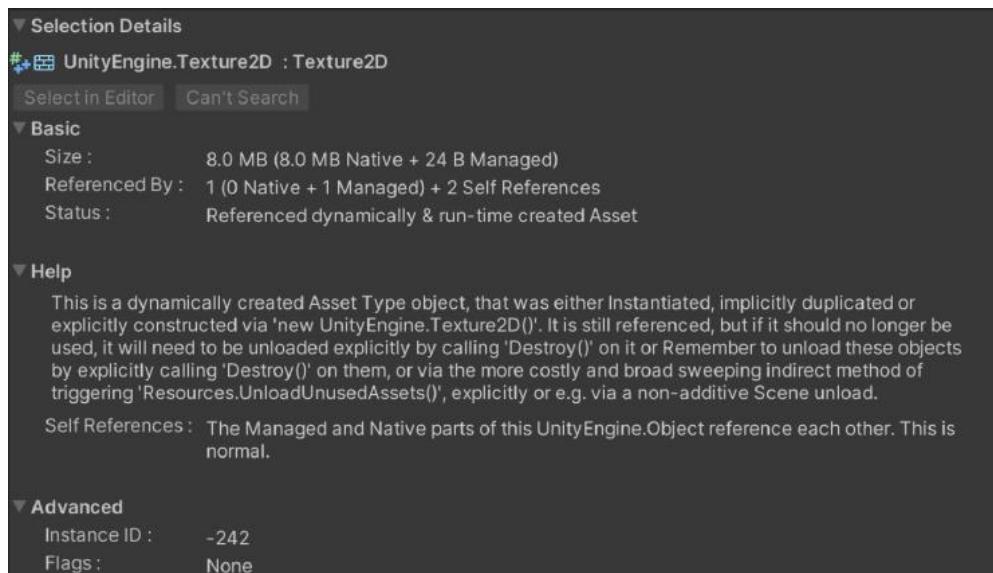
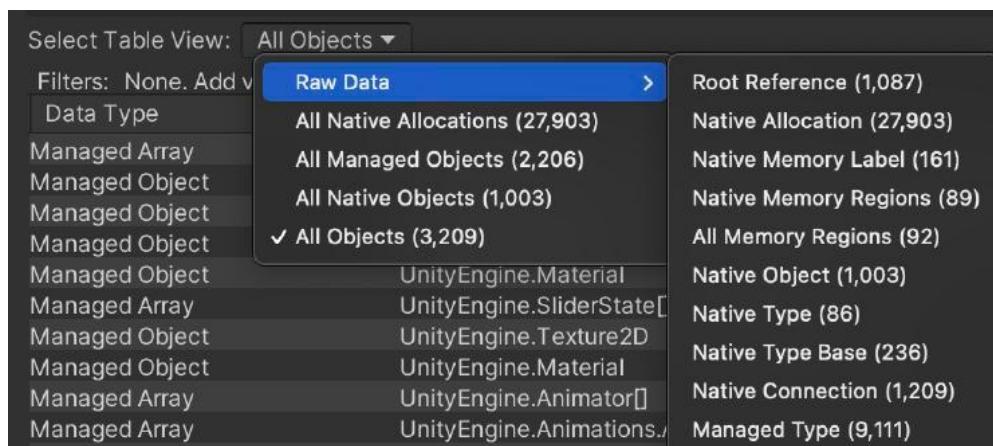


图3.51 选择细节。

补充信息：除摘要外的测量结果

与摘要不同，对象和分配允许更详细的内容，如分配，以表格形式查看。



▲图3.52 指定一个表视图

3.4 储存器

“Fragmentation”将虚拟内存的情况可视化，并可用于调查碎片化。它可能很难使用，因为有很多非直观的信息，如内存地址。

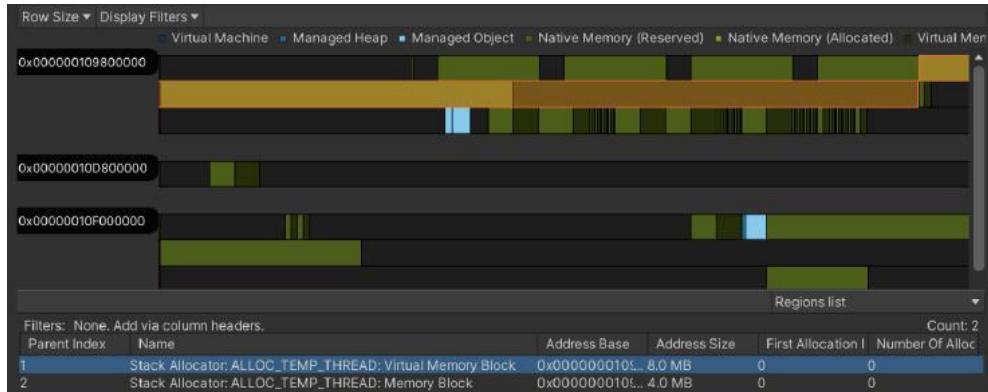


图3.53 分裂。

Memory Profiler v0.6增加了一个名为“Memory Breakdowns”的新功能，它需要Unity 2022.1或更高版本，但允许你在列表视图中查看TreeMaps和Unity Subsystems等对象信息。现在还可以检查可能的重复对象。

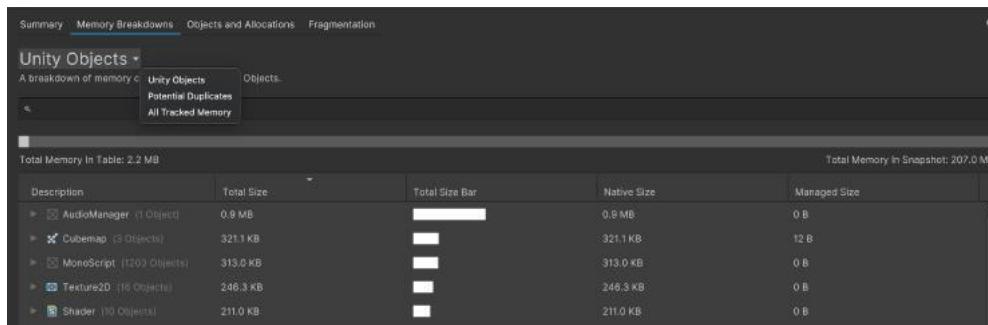
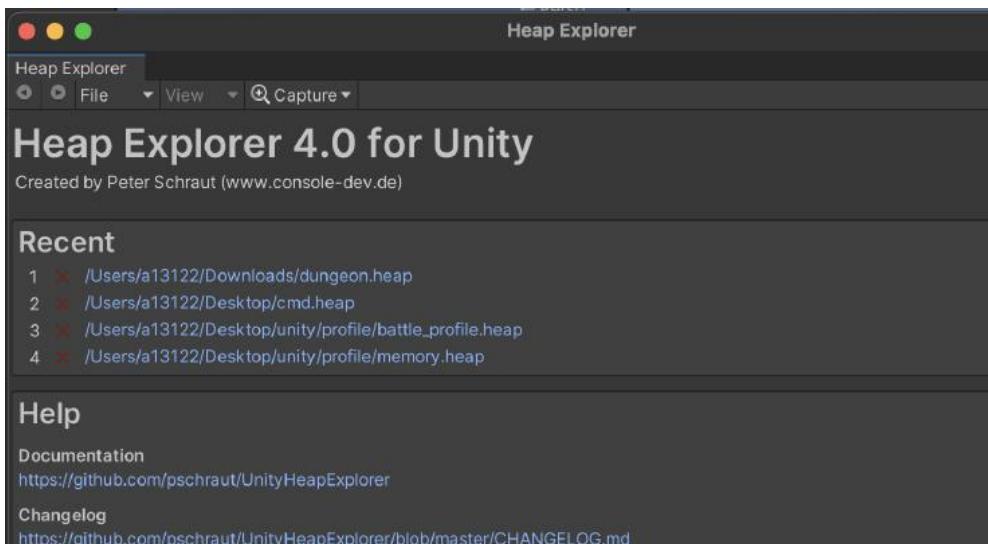


图3.54 内存分解。

3.5 堆栈资源管理器。

Heap Explorer是一个开源工具，来自私人开发者Peter^{77*1}。与Memory Profiler一样，这也是一个经常被用来调查内存的工具。在0.4之前的版本中，由于参考文献没有以列表的形式显示，所以Memory Profiler的追踪工作非常费力，虽然这在0.5之后得到了改进，但有些人可能使用的Unity版本不支持。在这种情况下，它作为一种替代工具仍有很大的价值，所以我们想在本文中介绍它。



▲ 图3.55 堆浏览器。

3.5.1 介绍的方法

该工具复制GitHub存储库^{*2}中列出的软件包URL，然后从软件包管理器中的“Add Package from Git url”添加该软件包。安装后该工具可以在“窗口->分析->内存探测器”下启动。

^{*1} <https://github.com/pschraut>

^{*2} <https://github.com/pschraut/UnityHeapExplorer>

3.5 堆栈资源管理器。

3.5.2 如何操作

Heap Explorer的工具栏看起来像这样

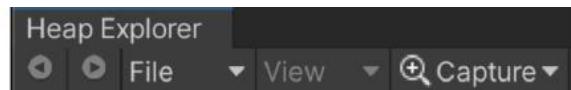


图3.56 堆浏览器工具栏。

左边和右边的箭头

允许你在操作中往回走，往前走。这在追踪参考资料时特别有用。

文件

测量文件可以被保存和加载。它们以.heap为扩展名被保存。

浏览

你可以在不同的显示屏幕之间切换。有许多不同的类型，如果你有兴趣，请看文件。

捕获

测量。然而，测量目标不能在**Heap Explorer**中改变。目标必须在Unity Profiler或Unity提供的其他工具中改变。保存 "将结果保存到文件中并显示出来，而 "分析 "则不保存而显示出来。需要注意的是，与 "内存管理器 "一样，测量过程中分配的内存不会被释放。

第3章 剖析工具

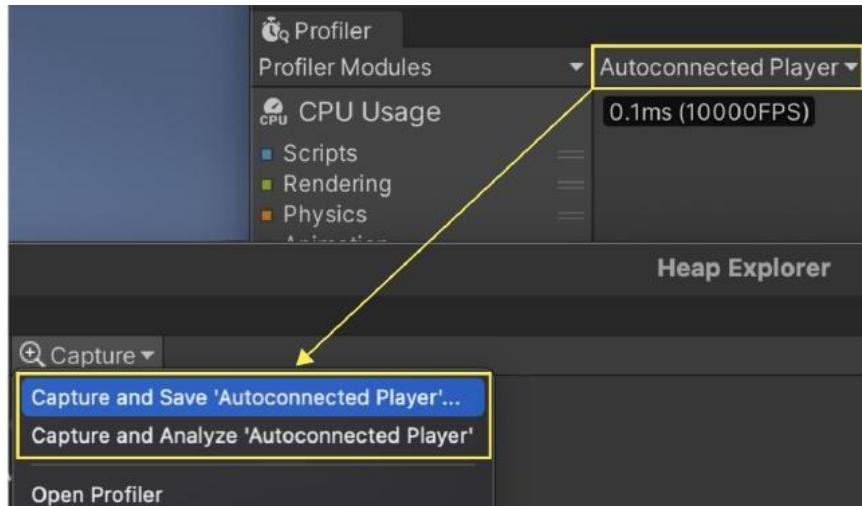


图3.57 切换测量目标

测量结果屏幕看起来像这样这个屏幕被称为“概览”。

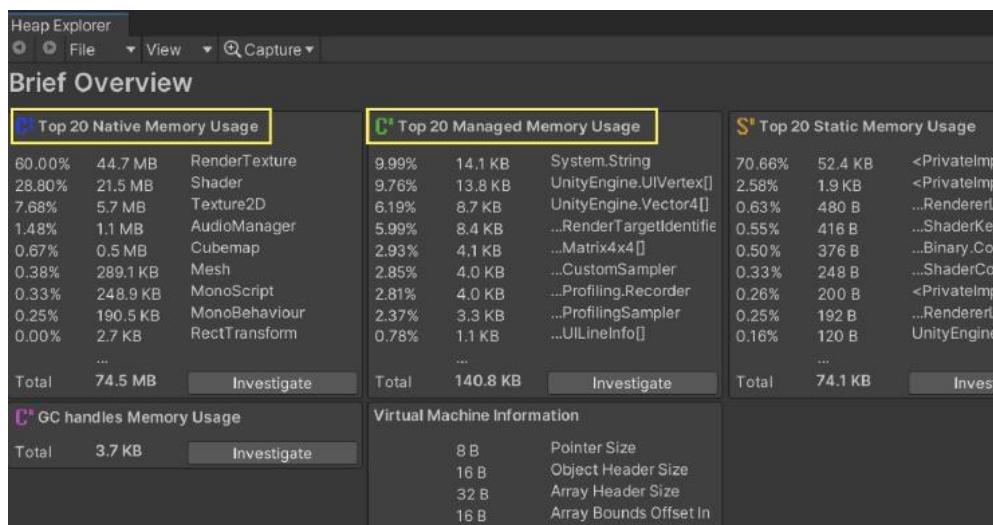


图3.58 堆资源管理器测量结果（概览）

在概览中，特别关注的类别是本地内存使用量和管理内存使用量，当按下调查按钮时，它们分别用绿线标记。

3.5 堆栈资源管理器。

了解更多信息。

在下面的章节中，我们将重点讨论显示类别细节的重要方面。

1. 宗旨

如果Native Memory是Investigate，C++对象将出现在这个区域。

对于管理型内存，C#对象会显示在这个区域。

C++ Objects							
Type	Name	Size	Count	DDoL	Persistent	Address	InstanceID
► RenderTexture		44.7 MB	5				
► Shader		21.5 MB	35				
▼ Texture2D		5.7 MB	63				
Texture2D	rock	2.7 MB		False	True	0x151D107E0	4114
(Texture2D)	AreaTex	350.5 KB		False	True	0x14FF75F50	4066
(Texture2D)	Large01	256.5 KB		False	True	0x15199E440	4092
(Texture2D)	Medium01	256.5 KB		False	True	0x14FF73010	4010

图3.59 对象显示区

头部有几个不熟悉的项目，应该添加。

▲ADDOL

Don't Destroy On Load的缩写。它显示该对象是否被指定为一个在场景转换后不被销毁的对象。

持久性

它是否是一个持久性的对象。这是Unity在启动时自动生成的对象。

通过选择图3.59中的对象，更新以下章节介绍的显示区域。

2. 参考资料：

目标对象被引用的对象被显示出来。

第3章 剖析工具

Referenced by 2 object(s)		
Type	C++ Name	Address
PostProcessD	PostProcessData	0x11413E660
...Texture2D	Medium06	0x1112B97C0

▲图3.60 参考文献。

3. 参考资料：

目标对象所指的对象被显示出来。

References to 1 object(s)		
Type	C++ Name	Address
GCHandle	UnityEngine.Texture2D	0x1112B97C0

▲图3.61 参考文献。

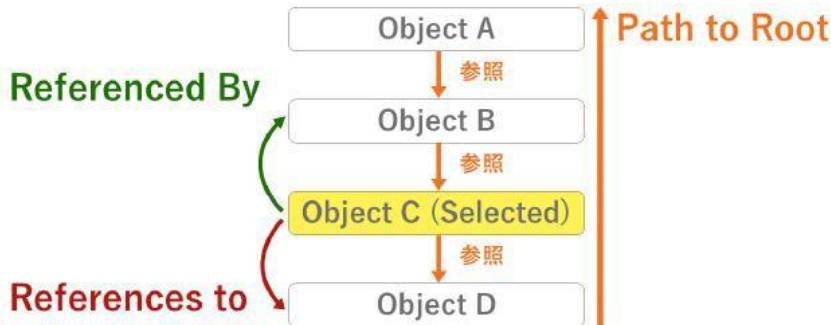
4. 通往根部的路径

引用目标对象的根对象被显示出来。这在调查内存泄漏时很有用，因为你可以看到是什么在持有引用。

2 Path(s) to Root		
Type	C++ Name	Depth Address
UnityEngine.Rendering.RenderPipelineManager	static s_CurrentPipelineAsset	6
	UniversalRenderPipelineAsset	0x106F1B
	UniversalRenderPipelineAsset	0x1519CF
	ForwardRendererData	0x1519D2
	PostProcessData	0x1519D2
	Texture2D	0x15199E
GraphicsSettings	GraphicsSettings	5 0x14FF50
	UniversalRenderPipelineAsset	0x1519CF
	UniversalRenderPipelineAsset_Renderer	0x1519D2
	PostProcessData	0x1519D2
	Texture2D	0x15199E

▲图3.62 通往根部的路径。

下面的图片总结了到目前为止的项目。



▲ 图3.63的图像参考。

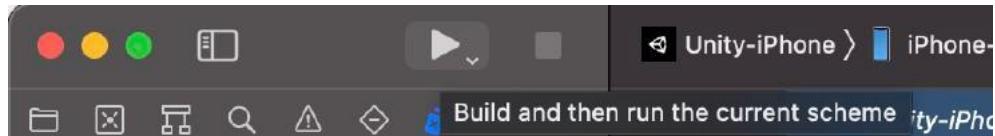
正如到目前为止所介绍的，Heap Explorer提供了一套完整的内存泄漏和内存调查所需的功能。它也非常轻便，所以请考虑使用这个工具。如果你喜欢它，最好再加一颗星以示感谢。

3.6 Xcode

Xcode是苹果公司提供的集成开发环境工具；当Unity中的目标平台被设置为iOS时，构建产品就是Xcode项目。建议使用Xcode进行严格的验证，因为它提供的数字比Unity更准确。在这一节中，我们将涉及三种剖析工具：Debug Navigator、GPU Frame Capture和Memory Graph。

3.6.1 简介方法

有两种方法可以从Xcode进行剖析：第一种是直接从Xcode在终端上构建和运行应用程序。如图3.64所示，只需按下运行按钮即可开始分析。本文件中省略了建筑证书等设置。



▲图3.64 Xcode运行按钮

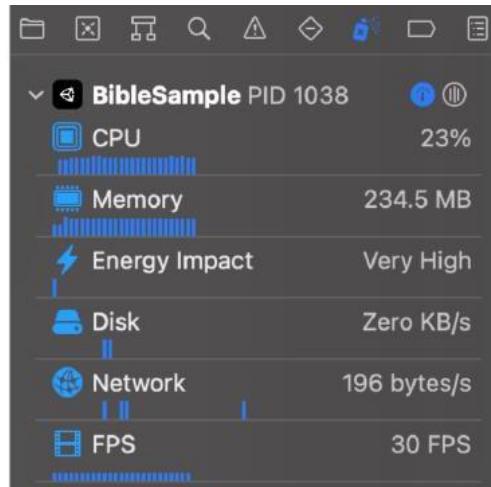
第二种方法是将一个正在运行的应用程序附加到Xcode调试器上。这可以通过在运行应用程序后从Xcode菜单 "Debug -> Attach to Process" 中选择运行中的进程来进行分析。请注意，构建证书必须是针对开发者的（苹果开发）；你不能用Ad Hoc或企业证书附加。



▲图3.65 Xcode中的调试器附件。

3.6.2 调试导航器

调试导航器允许你仅仅通过从Xcode运行应用程序来检查调试仪表，如CPU和内存。运行应用程序后，按图3.66中的喷雾符号，显示六个项目。或者，从Xcode菜单中，按查看它也可以通过'->导航仪->调试'打开。以下各节将对这些项目进行解释。



▲ 图3.66 选择调试导航器

1. CPU测量仪

你可以看到有多少CPU被使用。还可以查看每个线程的使用率。

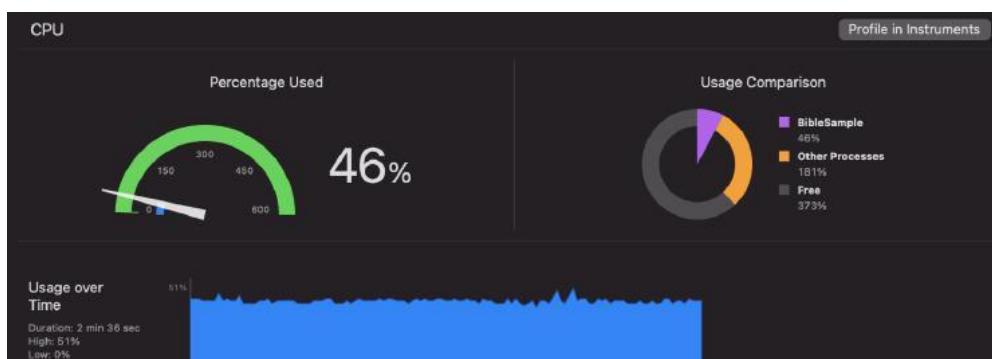


图3.67 CPU测量仪。

2. 记忆仪。

可以查看内存消耗的概况。详细的分析，包括细分，是不可能的。

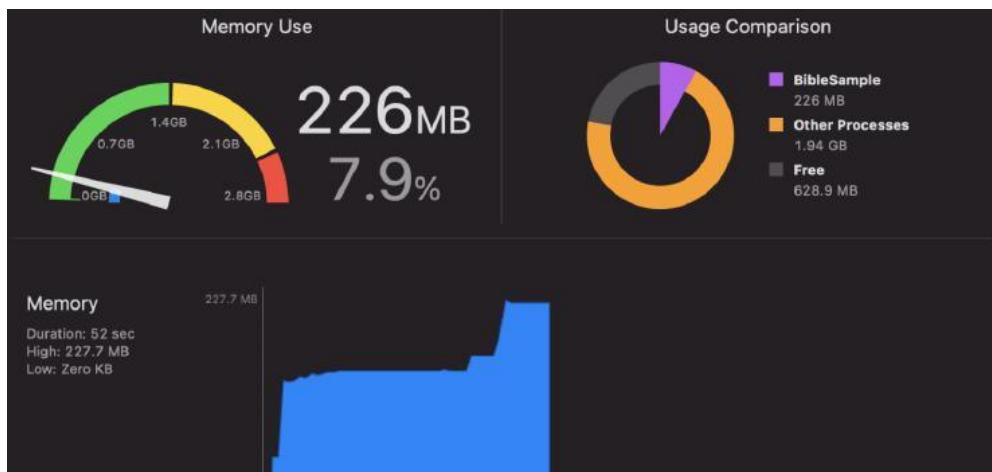


图3.68 内存表。

3. 能量计

你可以看到你的耗电概况，并按CPU、GPU、网络等细分使用。

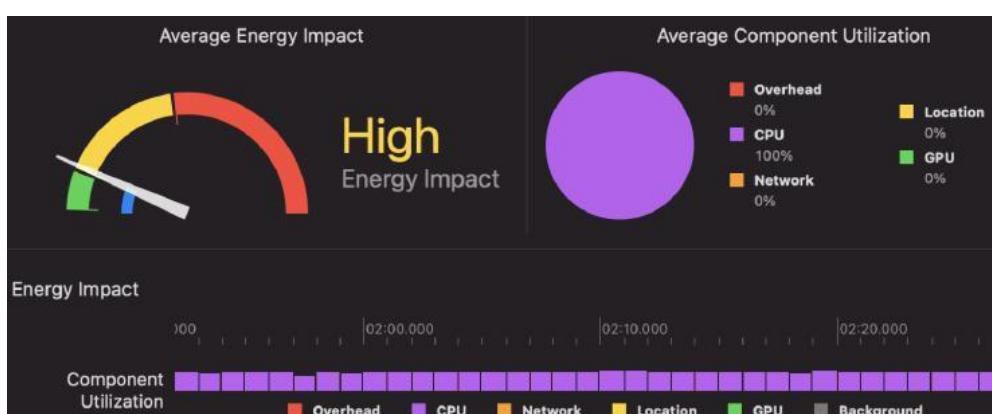


图3.69 能量表。

4. 磁盘测量仪。

文件I/O概述。这对于检查文件是否在意外时间被读取或写入很有用。

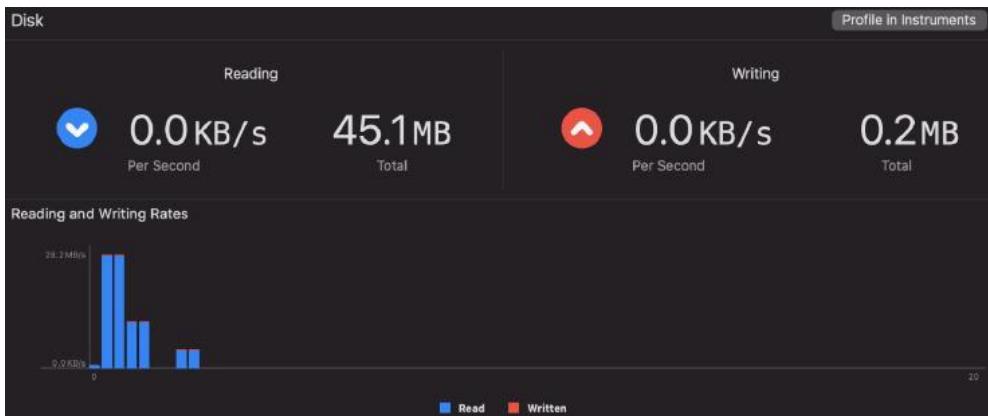


图3.70 磁盘仪。

5. 网络测量仪

你可以看到网络通信的概况，这和磁盘一样，对检查意外的通信很有用。

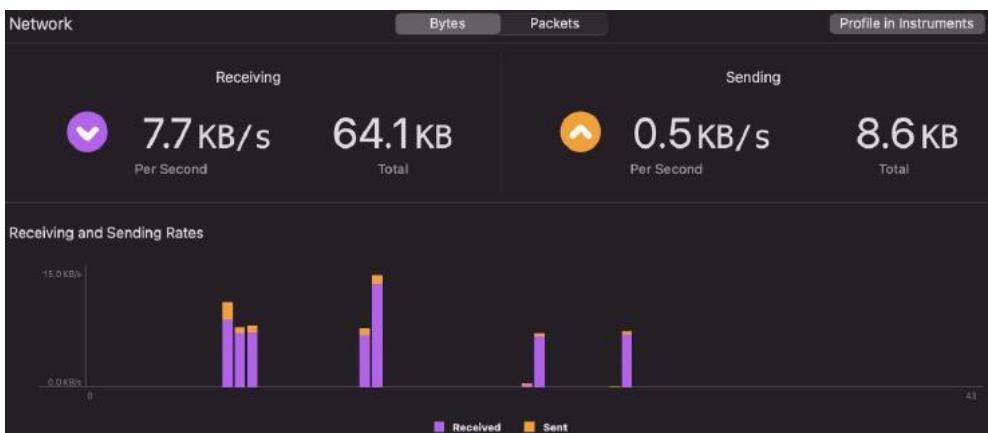


图3.71 网络仪表。

6. FPS测量仪。

这个仪表在默认情况下是不显示的。当启用GPU帧捕获（在3.6.3节中描述了GPU帧捕获）时，它就会显示出来，让你不仅可以检查FPS，还可以检查着色器阶段的使用情况以及每个CPU和GPU的处理时间。

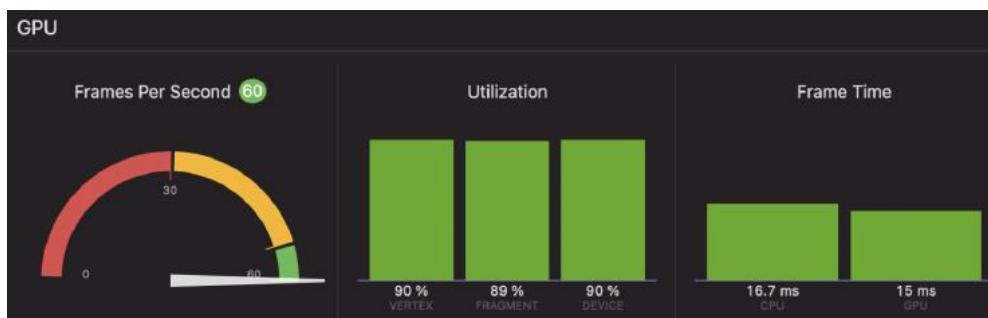


图3.72 FPS表。

3.6.3 GPU帧捕获

GPU Frame Capture是一个允许在Xcode上进行帧调试的工具。像Unity的帧调试器一样，它允许你看到渲染完成之前的过程，由于每个着色器阶段的信息比Unity中的更多，它可能对分析和改进瓶颈很有用。下一节将介绍如何使用它。

1. 准备

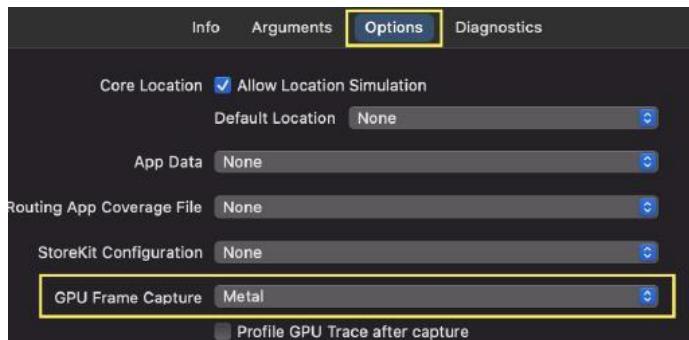
要在Xcode中启用GPU Frame Capture，你需要编辑方案。首先，通过选择'产品->方案->编辑方案'打开方案编辑屏幕。

3.6 Xcode



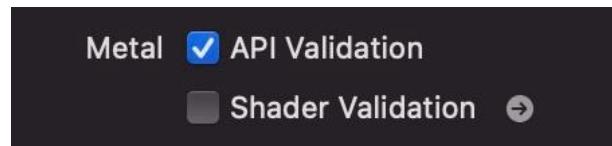
图3.73 计划编辑屏幕。

接下来，从“选项”标签，将GPU帧捕获改为“金属”。



▲ 图3.74 启用GPU帧捕获功能

最后，从“诊断”标签，启用Metal的“API验证”。



▲ 图3.75 启用API验证

2. 完全自动化的公共图灵测试，将计算机和人类区分开来

在执行过程中，通过按下调试栏中的相机图标来进行捕捉。根据场景的复杂性，第一次捕捉可能需要一些时间，所以要有耐心。请注意，在Xcode13或更高版本中，该图标已被改为金属图标。

Xcode12以前



Xcode13以降



图3.76 GPU帧捕获按钮

当捕获完成后，将显示以下摘要屏幕。

3.6 Xcode

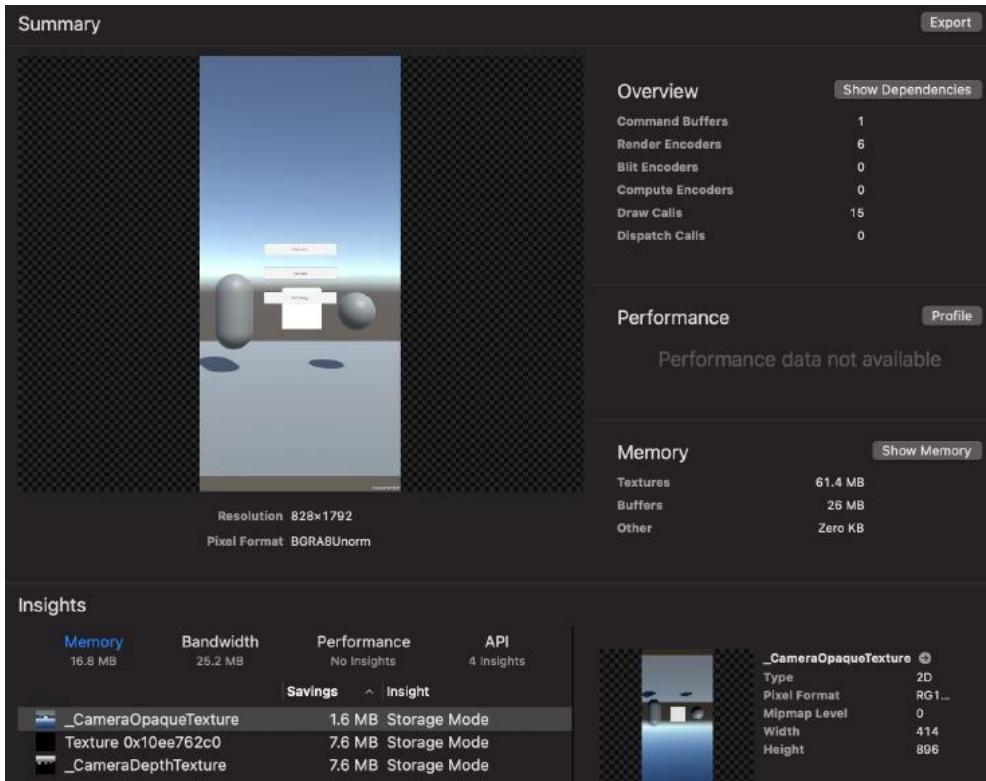
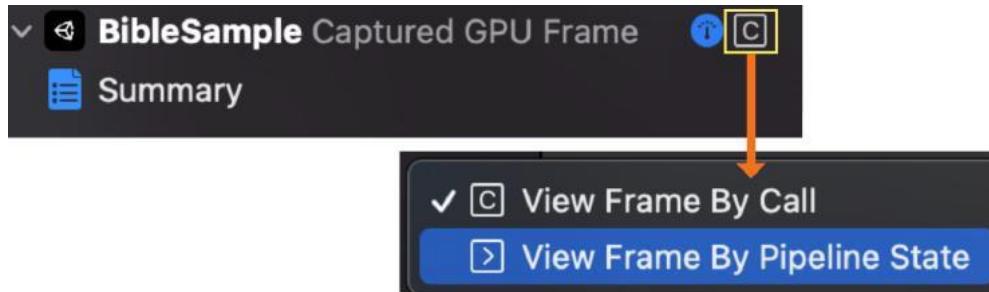


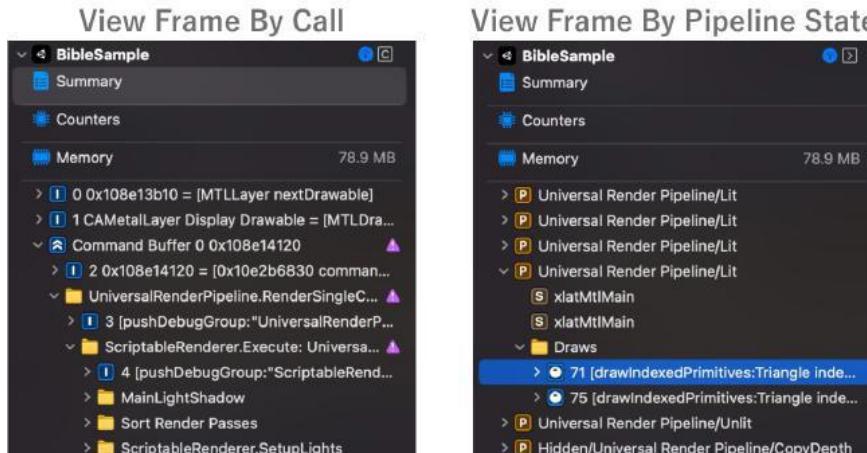
图3.77 摘要屏幕。

从这个摘要屏幕，你可以移动到一个屏幕，在那里你可以检查绘图的依赖性、内存和其他细节。导航区也显示与绘图有关的命令。有两种显示方法：“按调用查看帧”和“按管道状态查看帧”。



▲图3.78 改变显示方式

在“按调用”显示中，所有的绘图命令都按调用的顺序列出。这包括缓冲区的设置和其他绘制的准备工作，因此有非常多的命令被排在一起。另一方面，通过管道状态，只列出了每个着色器所绘制的几何图形的绘制命令。建议根据你所调查的内容，在不同的显示器之间切换。



▲图3.79 显示的差异

可以按下导航区的绘图命令，查看该命令使用的属性信息。属性信息包括纹理、缓冲器、采样器、着色器功能和几何。每个属性都可以被双击以获得更多信息。例如，你可以看到着色器代码本身，以及采样器是Repeat还是Clamp。

3.6 Xcode

Vertex					
Plane	Index	1 KB	Offset: 0x0		
ScratchBuffer0_1	Buffer 0	4 MB	Offset: 0x240	VGlobals	Read
ScratchBuffer0_1	Buffer 1	4 MB	Offset: 0x403c0	UnityPerDraw	Read
Compute_0	Buffer 2	80 bytes	Offset: 0x0	UnityPerMaterial	Read
ScratchBuffer0_1	Buffer 3	4 MB	Offset: 0x4c0	MainLightShadows	Read
Plane	Buffer 4	6 KB	Offset: 0x0	vertexBuffer.0	Read
Geometry	Post Vertex Tran...				
Vertex Attributes	Vertex Attributes				
xlatMtlMain (Universal Re...	Vertex Function	Library 0x11aaa6a20 [Jus...]			
Fragment					
Texture 0x114305d50	Texture 0	(Cube) 128 × 128	RGBA8Unorm	unity_SpecCube0	Read
UnityWhite	Texture 1	4 × 4	RGBA8Unorm	_BaseMap	Read
TempBuffer 1 2048x2048	Texture 2	2048 × 2048	Depth32Float	_MainLightShado...	Read

图3.80 绘图命令的细节。

几何属性不仅以表格形式显示顶点信息，而且还允许用户移动摄像机来检查几何体。

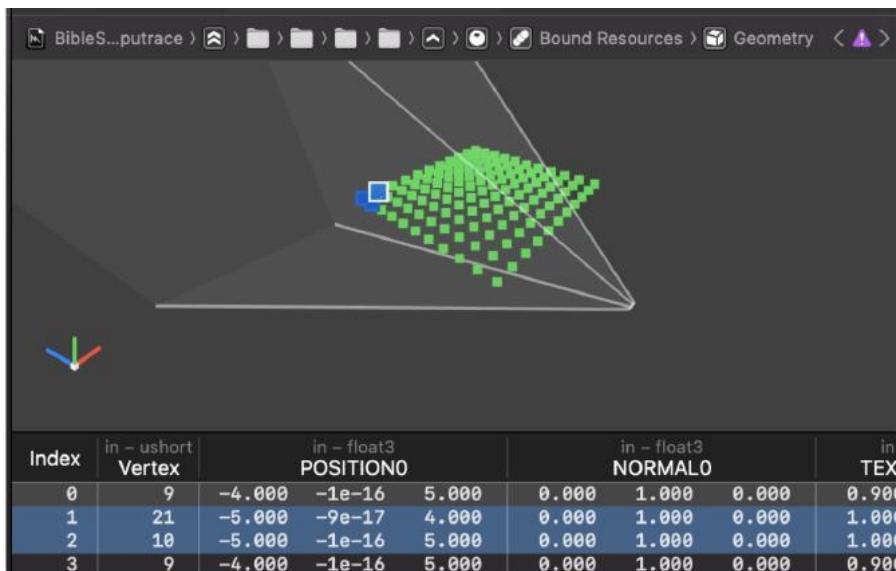


图3.81 几何图形查看器。

下一节将介绍“摘要”屏幕中“性能”栏的“参数”。点击这个按钮，开始进行更详细地分析。当分析完成后，绘制的时间将显示在导航仪区域。

> Hidden/Universal Render Pipeline/Blit	541.92 µs
> Universal Render Pipeline/Lit	437.30 µs
> Skybox/Procedural	398.49 µs
> Hidden/Universal Render Pipeline/CopyDepth	296.52 µs

▲ 图3.82 剖面图后显示。

分析的结果也可以在“计数器”屏幕上更详细地查看。在这个屏幕上，你可以以图形方式看到每个绘图的处理时间，如顶点、光栅化和片段。



图3.83 计数器屏幕。

下文介绍了“摘要”屏幕中“显示内存”一栏的情况。点击这个按钮将带你到一个屏幕，你可以检查GPU正在使用的资源。显示的信息主要是纹理和缓冲区。检查是否有任何不必要的项目是个好主意。

3.6 Xcode

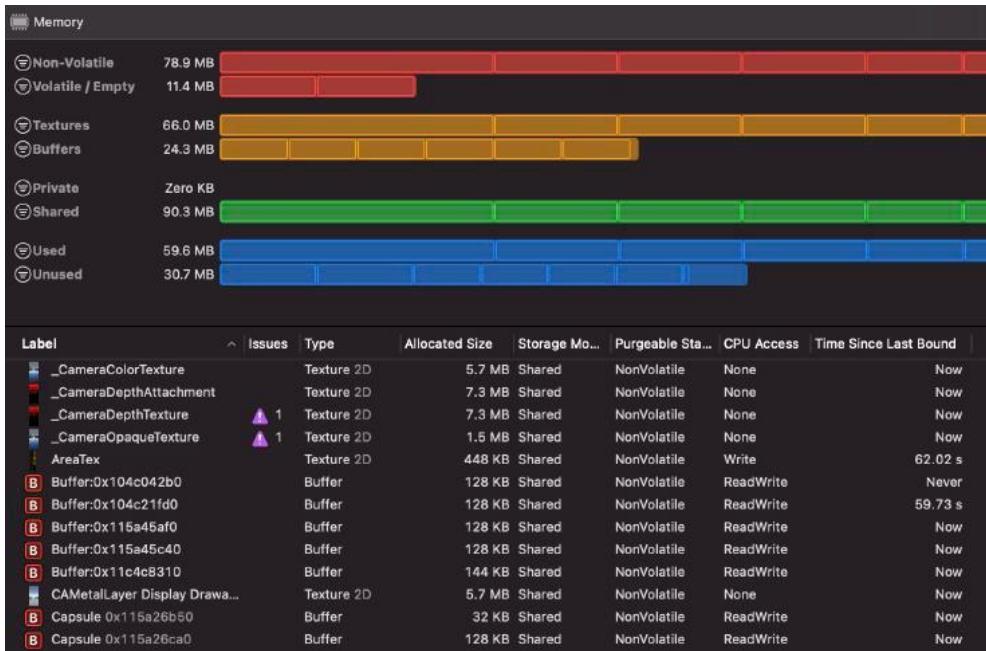
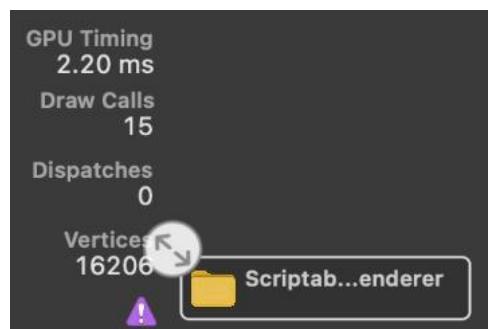


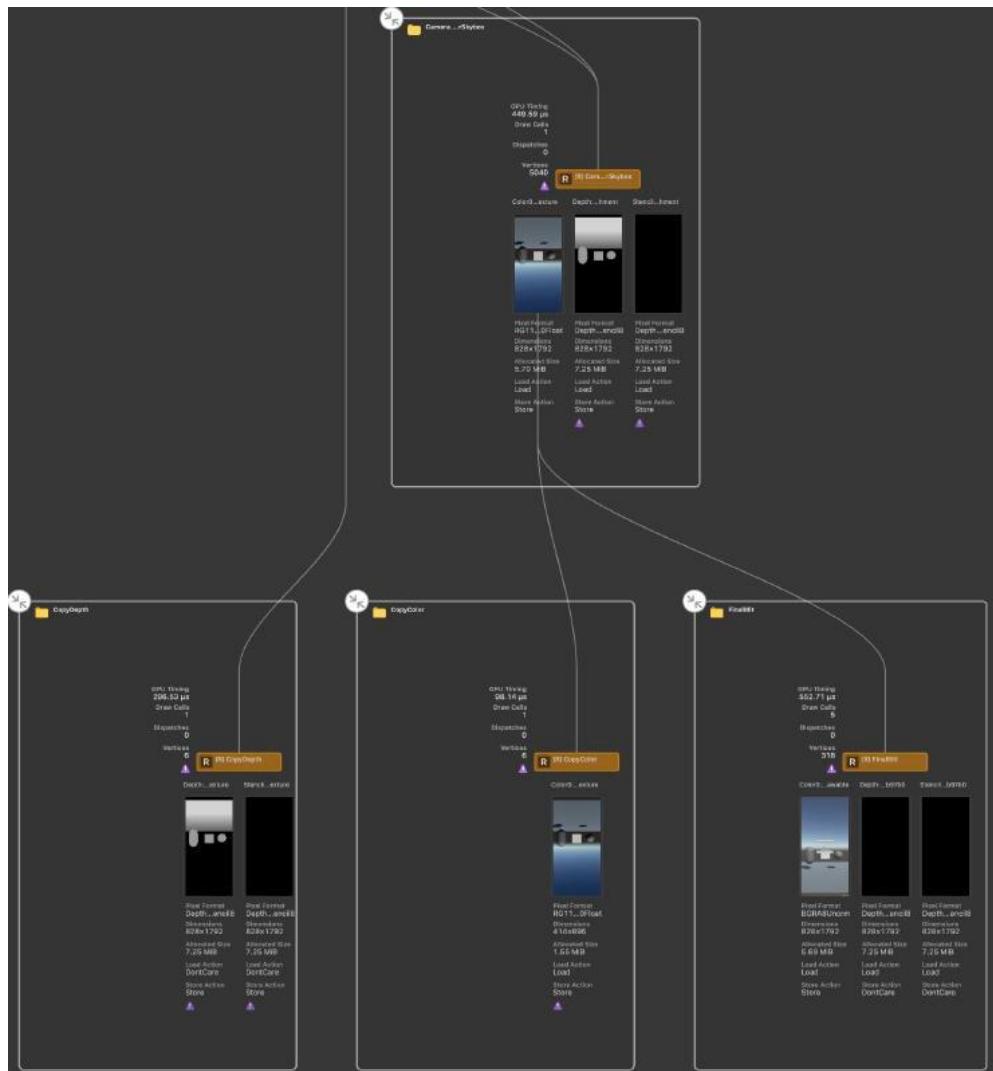
图3.84 GPU资源确认界面。

最后，对“摘要”屏幕的“概览”部分的“显示依赖关系”进行了解释。点击这个按钮将显示每个渲染路径的依赖关系，通过在查看依赖关系时点击“箭头向外”的按钮，你可以打开该级别以下的进一步依赖关系。



▲ 图3.85 打开依赖关系的按钮

当你想看哪些图纸取决于什么时，请使用此屏幕。



▲ 图3.86 打开层次结构

3.6.4 记忆图表

这个工具可以让你分析捕获时机的内存情况。左边的导航区显示实例，通过选择它们，参考关系以图形方式显示。右边的检查器区域显示有关实例的详细信息。

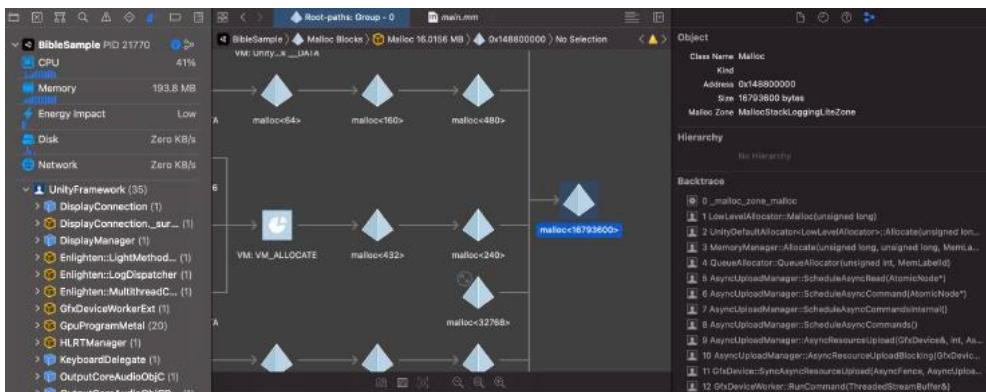


图3.87 MemoryGraph捕获屏幕。

这个工具可以用来调查那些不能在Unity上测量的对象的内存使用情况，例如插件。

下面解释如何使用它。

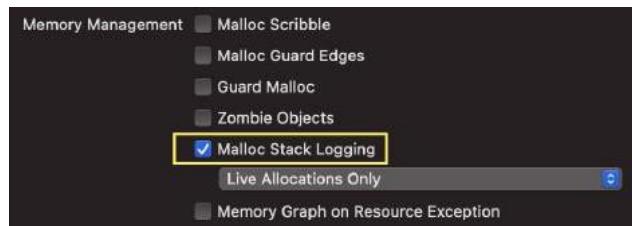
1. 提前准备

需要对计划进行编辑以获得Memory Graph中的有用信息。

进入“产品->方案->编辑方案”，打开方案编辑界面。然后点击

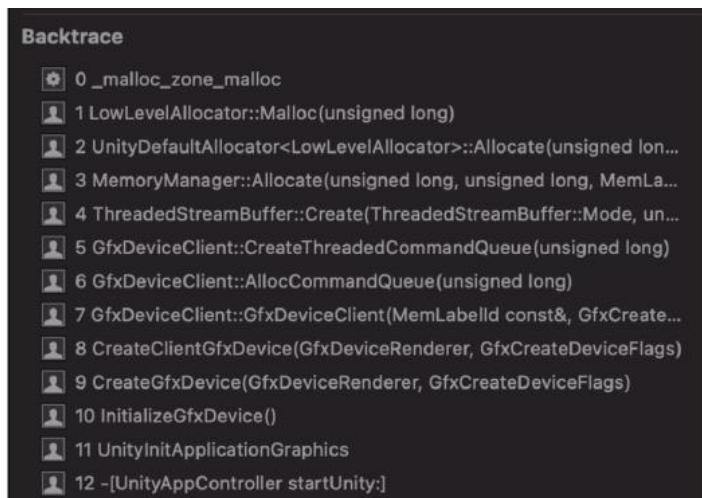
从“诊断”标签，启用“Malloc堆栈记录”。

第3章 剖析工具



▲ 图3.88 启用Malloc栈记录

通过启用这个功能，Backtrace将显示在Inspector中，你将能够看到它是如何被分配的。



▲ 图3.89 背面追踪显示。

2. 完全自动化的公共图灵测试，将计算机和人类区分开来

捕获是在应用程序运行时按下调试栏的一个类似分支的图标来进行的。



图3.90 内存图表捕捉按钮

内存图也可以通过“文件->导出内存图”保存为文件。这个文件可以用vmmap命令、heap命令和malloc_history命令进行更深入的调查。如果你有兴趣，请调查。作为一个例子，vmmap命令的摘要显示如下，使你能够掌握整个情况，这在MemoryGraph中是难以掌握的。

▼ 清单3. 5vmmap摘要命令

```
1: vmmap --summary hoge.memgraph
```

REGION TYPE	VIRTUAL SIZE	RESIDENT SIZE	DIRTY SIZE	SWAPPED SIZE	VOLATILE SIZE	NONVOL SIZE	EMPTY SIZE	REGION COUNT
<hr/>								
Activity Tracing	256K	32K	32K	0K	0K	32K	0K	1
CoreAnimation	32K	32K	32K	0K	0K	16K	0K	2
Foundation	16K	16K	16K	0K	0K	0K	0K	1
IOAccelerator	99.2M	78.4M	78.4M	0K	0K	78.4M	13.2M	98
IOKit	384K	304K	304K	0K	0K	0K	0K	19
IOSurface	17.1M	17.1M	17.1M	0K	0K	17.1M	0K	3
Image IO	16K	0K	0K	16K	0K	0K	0K	1
Kernel Alloc Once	32K	16K	16K	0K	0K	0K	0K	1
MALLOC guard page	192K	0K	0K	0K	0K	0K	0K	12
MALLOC metadata	336K	288K	288K	0K	0K	0K	0K	15
MALLOC_LARGE	69.8M	23.7M	23.7M	27.0M	0K	0K	0K	1215
MALLOC_LARGE metadata	256K	144K	144K	80K	0K	0K	0K	1
MALLOC_NANO	512.0M	304K	304K	32K	0K	0K	0K	1
MALLOC_SMALL	64.0M	5360K	3888K	9.9M	0K	0K	0K	8
MALLOC_SMALL (empty)	8192K	0K	0K	48K	0K	0K	0K	1
MALLOC_TINY	19.0M	8320K	8320K	2416K	0K	0K	0K	19
MALLOC_TINY (empty)	4096K	128K	128K	0K	0K	0K	0K	4
Performance tool data	4784K	4592K	4592K	192K	0K	0K	0K	43
								
TOTAL	1.6G	355.9M	143.5M	41.6M	0K	95.6M	13.2M	4362

图3.91 MemoryGraph摘要显示。

3.7 器械

Xcode有一个叫做Instruments的工具，专门用于详细的测量和分析；可以通过选择“产品->分析”来建立仪器。一旦完成，就会打开一个屏幕，选择一个测量模板，如下图所示。

第3章 剖析工具

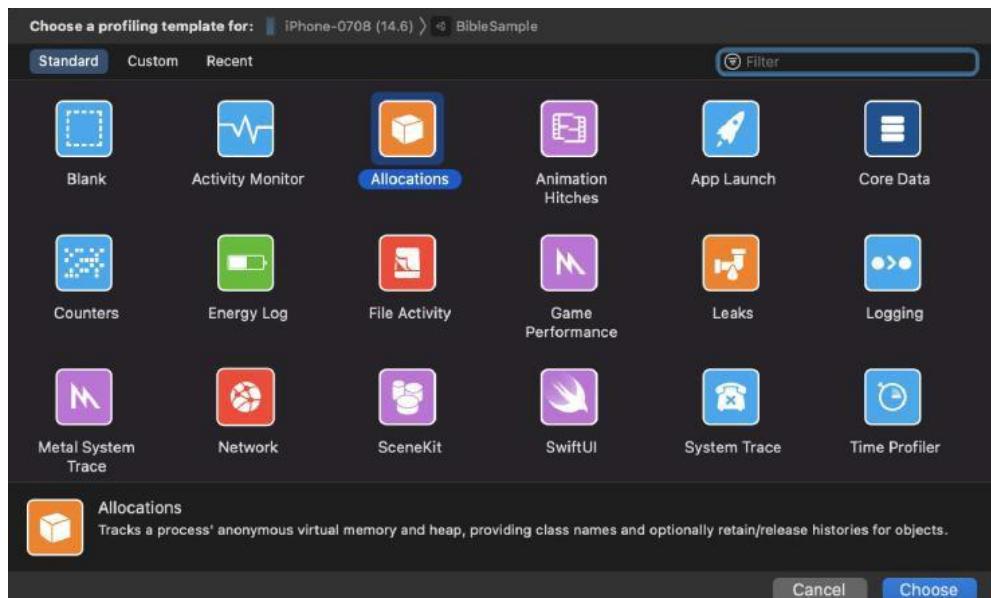


图3.92 仪器模板选择屏幕

从大量的模板中可以看出，仪器可以分析广泛的内容。本节重点介绍其中最常用的“时间规划器”和“分配”。

3.7.1 时代周刊

Time Profiler是一个测量代码执行时间的工具，它在Unity Profiler中。与CPU模块一样，它被用来提高处理时间。要开始测量，必须按下Time Profiler工具条上红色圈出的记录按钮。

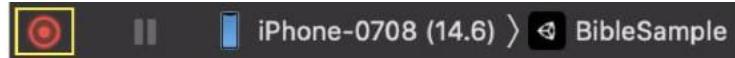


图3.93 记录开始按钮。

当进行测量时，显示屏将看起来像图 3.94。

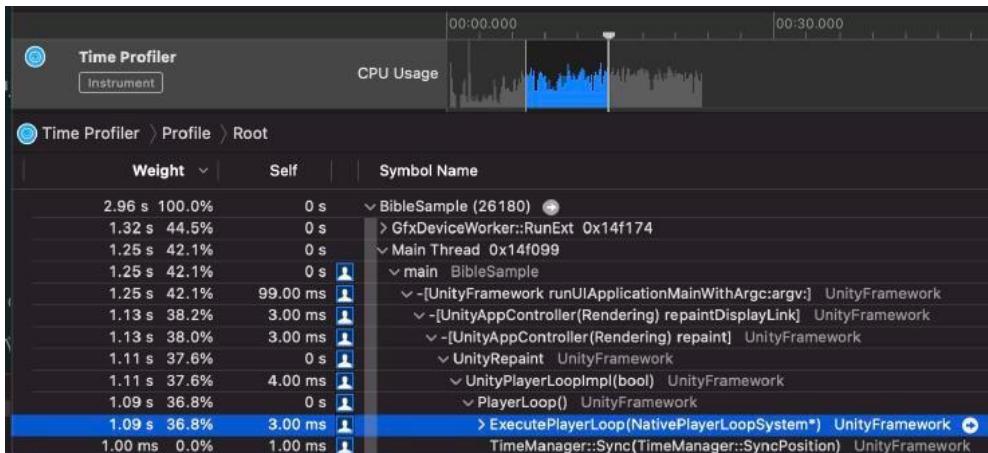
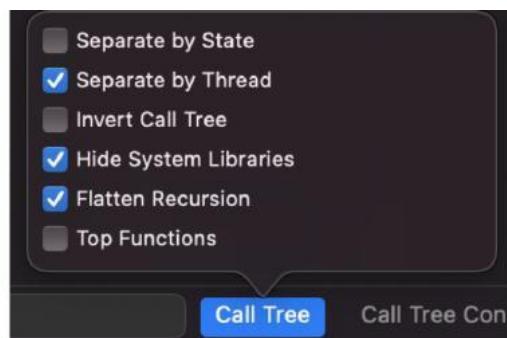


图3.94 测量结果

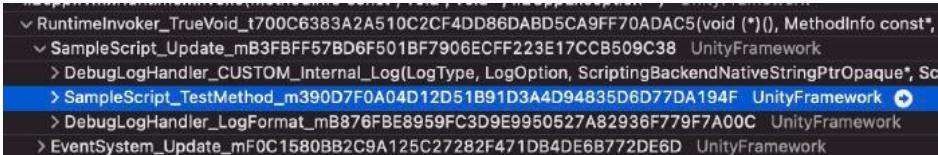
与Unity Profiler不同的是，分析是分节进行的，而不是一帧一帧地进行。底部的树状视图显示了区间内的处理时间。在优化游戏逻辑的处理时间时，建议在树状视图中分析PlayerLoop下面的处理。为了使树状视图的显示更容易看到，建议在Xcode的底部设置Call Trees的设置，如图3.95所示。特别是，检查隐藏系统库。这使得通过隐藏无法访问的系统代码，更容易进行调查。



▲ 图3.95 呼叫树设置。

通过这种方式，可以对处理时间进行分析和优化。

时间管理器中的符号名称与统一管理器中的不同。没有大的区别，但符号是“`class_name_function_name_random_string`”。



▲ 图3.96 时间管理器中的符号名称

3.7.2 拨款。

Allocations是一个测量内存使用的工具。它用于内存泄漏和使用的改进。

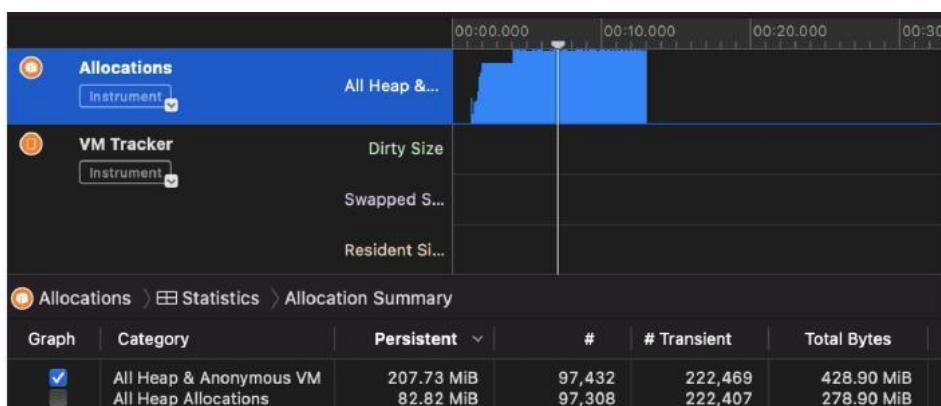
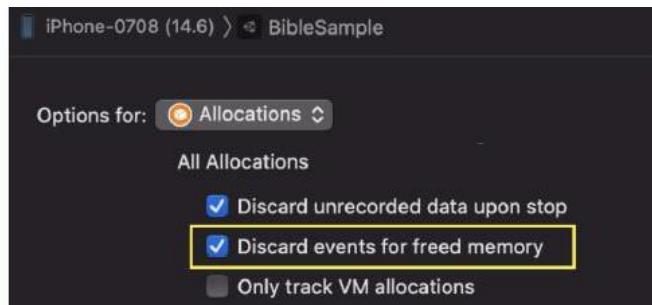


图3.97 分配的测量屏幕。

在你开始测量之前，打开“文件->录音选项”，并勾选“丢弃自由内存的事件”。



▲ 图3.98 选项设置屏幕。

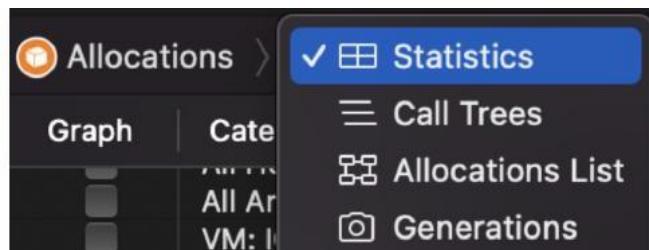
如果这个选项被启用，当内存被释放时，该记录就会被销毁。



▲ 图3.99 根据选项设置而产生的差异

从图3.99中可以看出，有选项和无选项时，外观都有很大变化。有了选项，只有在分配内存时才会记录行数。另外，当分配的区域被释放时，记录的线路会被销毁。这意味着在这个选项下，如果有一行留在内存中，它就没有被从内存中释放出来。例如，如果设计是通过场景转换来释放内存，那么如果在转换之前有许多行留在场景部分，就可能怀疑有内存泄漏。在这种情况下，请关注树状视图中的细节。

屏幕底部的“树状视图”显示指定范围的细节，与“时间编辑器”类似。
有四种不同的方式来显示这个树状视图。



▲ 图3.100 选择一种显示方法

最推荐的显示方法是呼叫树。这使你能够跟踪哪个代码引起的分配。呼叫树显示选项在屏幕的底部，你可以设置隐藏系统库等选项，其方式与图3.95中介绍的时间管理器相同。图3.101捕捉到了调用树的显示，显示出12.05MB的分配是由SampleScript的OnClicked产生的。

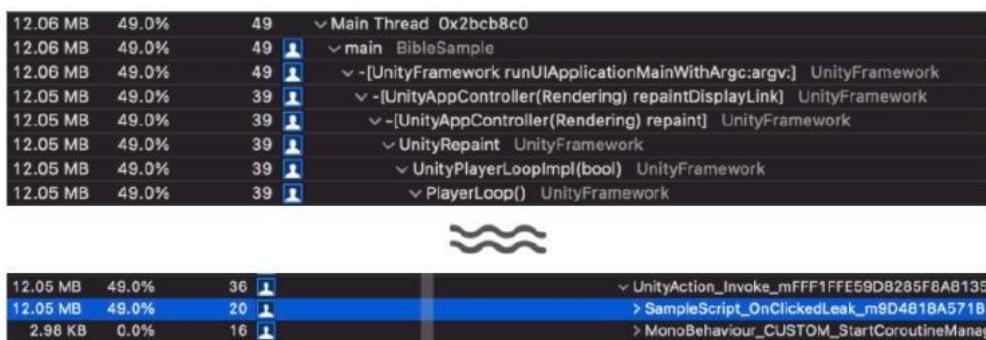


图3.101 呼叫树显示。

最后，我们介绍一个叫做“生成”的功能：在Xcode的底部有一个叫做“标记生成”的按钮。



▲ 图3.102 标记生成按钮

按这个键将存储当时的记忆。然后你可以再次按下标记

按“生成”键记录新分配的内存量与之前的数据相比。

Snapshot	Timestamp	Growth	# Persistent
> Generation A	00:15.602.827	168.47 MiB	96,330
> Generation B	00:20.858.176	75.71 MiB	6,219
> Generation C	00:24.390.265	256 Bytes	1

▲图3.103 世代。

图3.103中的每一个世代在详细查看时都以呼叫树的形式显示，因此有可能跟踪导致内存增加的原因。

3.8 安卓工作室

Android Studio是一个集成的Android开发环境工具。这个工具可以用来衡量应用程序的状态。四个可配置的项目是CPU、内存、网络和能源。本节首先介绍了剖析方法，然后介绍了CPU和内存的测量项目。

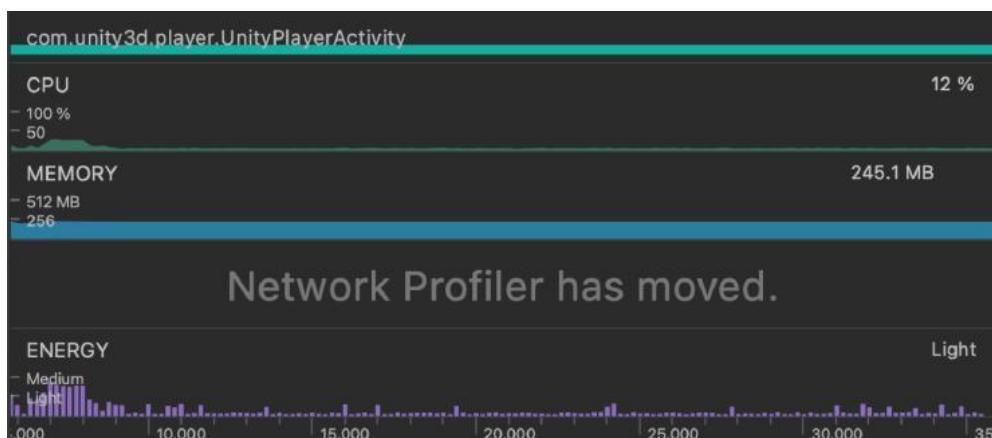


图3.104 简介屏幕。

3.8.1 简介方法

有两种方法可以进行剖析：第一，通过Android Studio构建和剖析。在这种方法中，你首先从Unity导出Android Studio项目，在构建设置中勾选“导出项目”，然后构建。

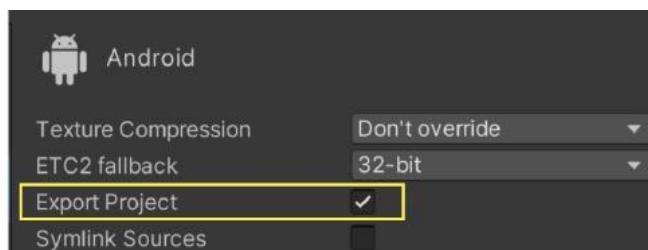


图3.105 导出一个项目

接下来，在Android Studio中打开导出的项目。然后，在连接安卓设备的情况下，按右上角类似仪表的图标，开始构建。构建完成后，应用程序将启动，配置文件将开始。

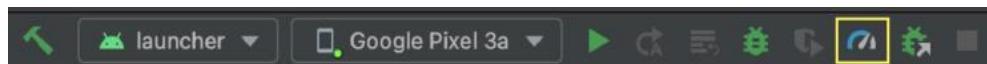
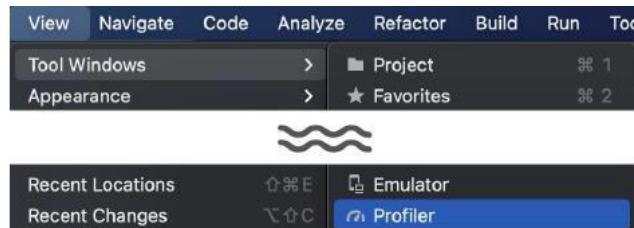


图3.106 序列启动图标。

第二种方法是将运行中的进程附加到调试器上，并对其进行测量。首先，从Android Studio菜单“视图->工具窗口->Profiler”中打开Android Profiler。



▲ 图3.107 打开安卓系统编辑器

接下来，从打开的Profiler的**SESSIONS**中选择要测量的环节。要连接一个会话，要测量的应用程序必须正在运行。另外，二进制文件必须从开发建设中调整。一旦会话被连接，简介就开始了。



▲ 图3.108 选择要剖析的SESSION

第二种附加到调试器的方法值得记住，因为它不需要导出项目，而且是现成的。

严格来说，你需要在AndroidManifest.xml中设置debuggable和profileable，而不是在Unity的Development Build中。在Unity中，如果你进行Development Build，你会自动设置debuggable被设置为true。

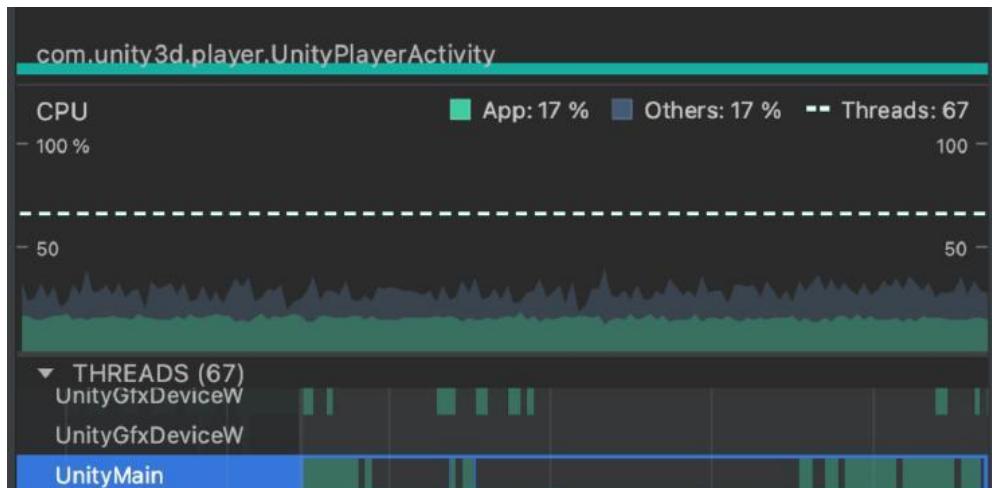
3.8.2 CPU测量

CPU测量屏幕看起来像图3.109。仅仅这个屏幕并不能告诉你什么东西消耗了多少处理

时间。要获得更详细的视图，请点击[s](#)

第3章 剖析工具

必须选择红色。



▲ 图3.109 CPU测量顶部屏幕，线程选择

在选择了一个线程后，通过按下记录按钮来测量该线程的callstack。如图3.110所示，有几种测量类型可供选择，但“Callstack Sample Recording”应该是可以的。

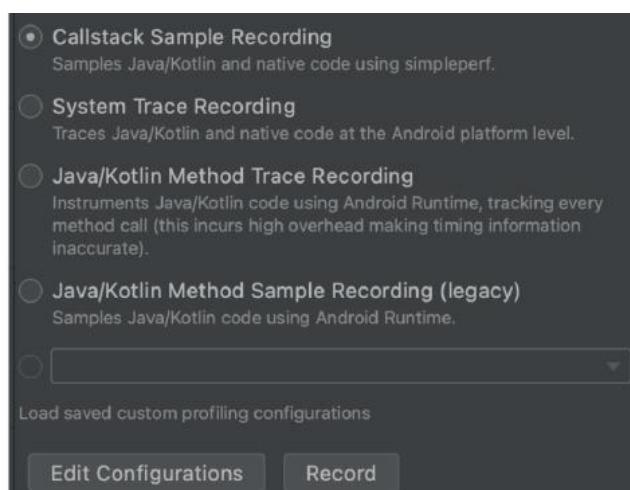


图3.110 记录开始按钮。

3.8 安卓工作室

按下“停止”按钮就可以完成测量并显示结果。结果屏幕显示它看起来就像Unity Profiler中的一个CPU模块。

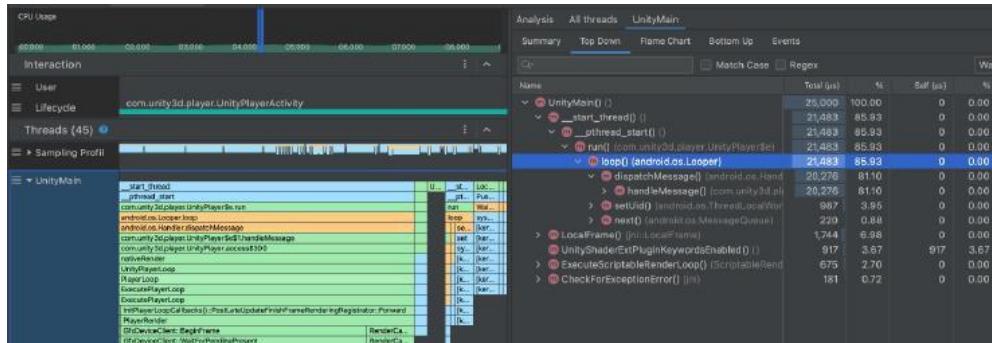


图3.111 呼叫堆栈测量结果屏幕。

3.8.3 记忆测量

内存测量屏幕如图3.112所示。在这个屏幕上不能看到内存分解。

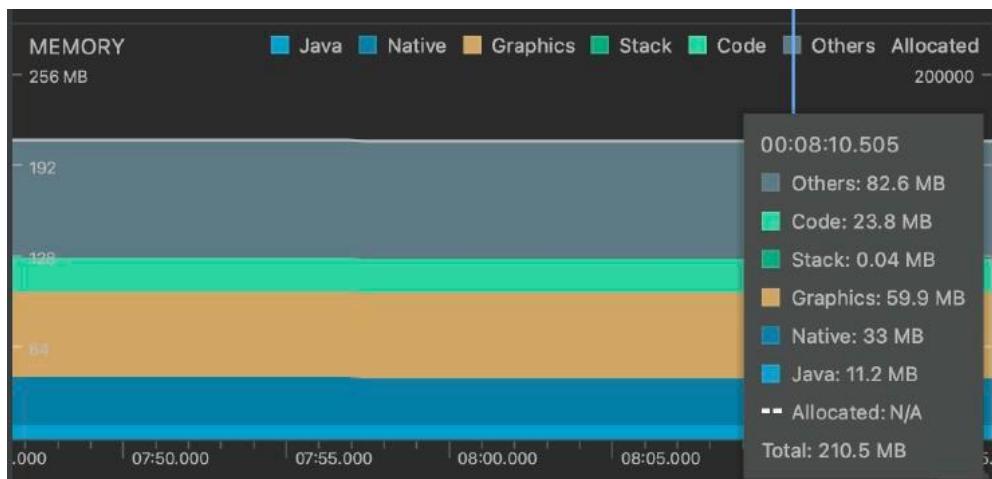


图3.112 内存测量屏幕。

如果你想看到内存的细分，你需要进行额外的测量。有三种测量方法

第3章 剖析工具

“捕获堆转储”功能允许在按下按钮的时候捕获内存信息。捕获堆转储”允许在按下按钮的时候获取内存信息。其他按钮是用来分析测量部分的分配。

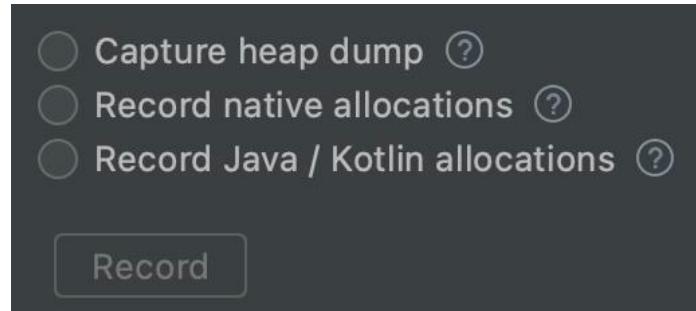


图3.113 内存测量选项。

作为一个例子，图3.114捕获了一个堆倾倒的测量结果。详细的分析会比较困难，因为颗粒度有点粗大。

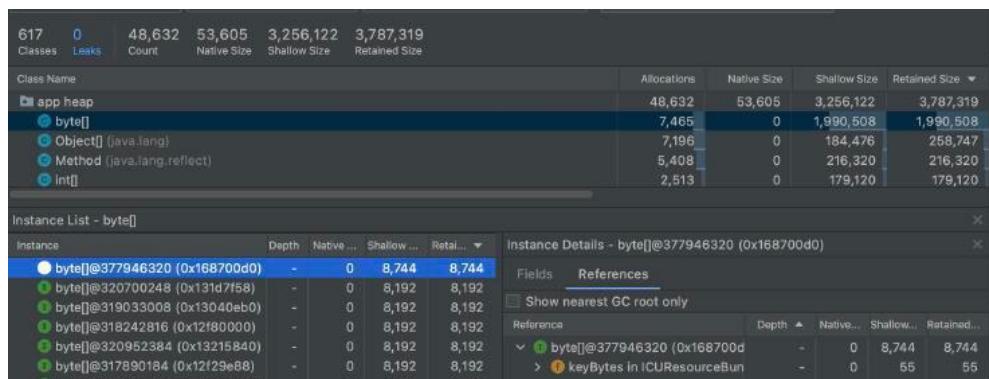


图3.114 堆栈倾倒结果。

3.9 RenderDoc

RenderDoc是一个开源的、高质量的图形调试器工具，可以免费使用。该工具目前支持Windows和Linux，但不支持Mac。支持的图形API是

Vulkan、OpenGL（ES）、D3D11和D3D12。因此，它可以在Android上使用，但不能在iOS上使用。

在这一节中，我们将实际剖析一个Android应用程序。然而，请注意，对安卓系统进行分析有一些限制。首先，安卓操作系统版本必须是6.0或更高。而且要测量的应用程序必须启用可调试功能。如果在构建时选择了“开发构建”，这就没有问题了。请注意，该简介所使用的RenderDoc的版本是v1.18。

3.9.1 测量方法

首先，准备RenderDoc。从官方网站^{*3}下载安装程序并安装该工具。安装后，打开RenderDoc工具。

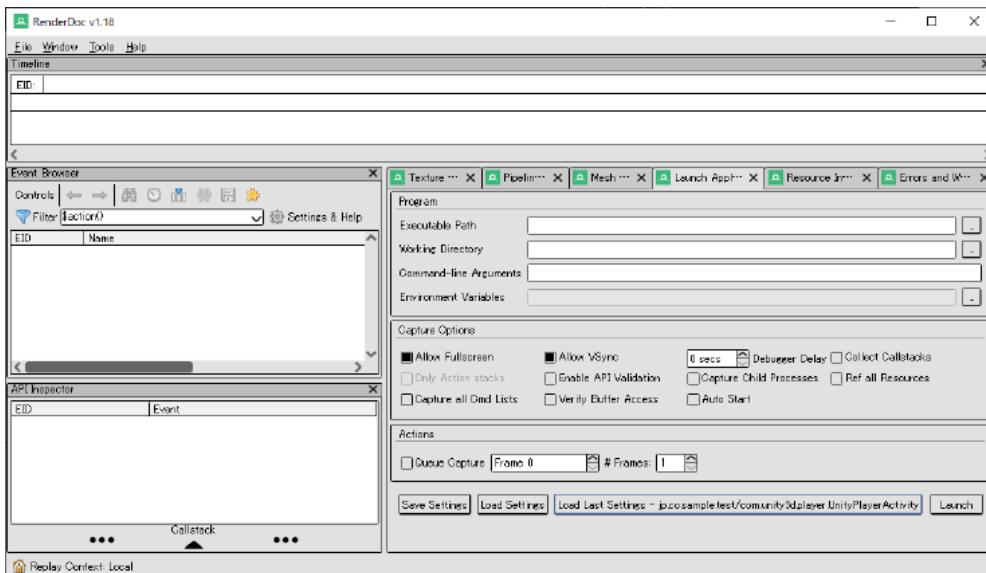
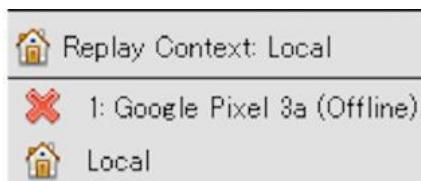


图3.115 启动RenderDoc后的屏幕。

接下来，将你的安卓设备连接到RenderDoc。按左下角的房子符号，显示连接到PC的设备列表。从列表中，选择被测量的

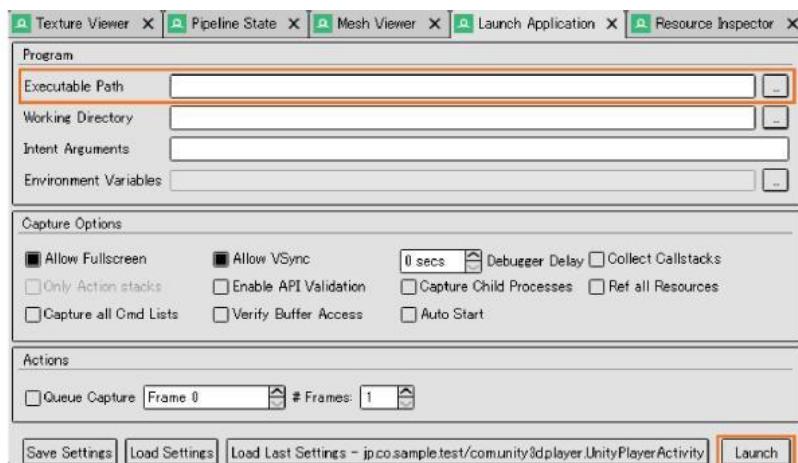
^{*3} <https://renderdoc.org/>

请选择你要使用的设备。



▲ 图3.116 连接到设备

接下来，选择你想从连接的设备上启动的应用程序。从右侧的标签中选择启动应用程序，并从可执行路径中选择你要运行的应用程序。



▲ 图3.117 启动应用程序标签 选择正在运行的应用程序

当文件浏览器窗口打开时，寻找这次要测量的Pacakge名称并选择活动。

选择以下内容。

3.9 RenderDoc

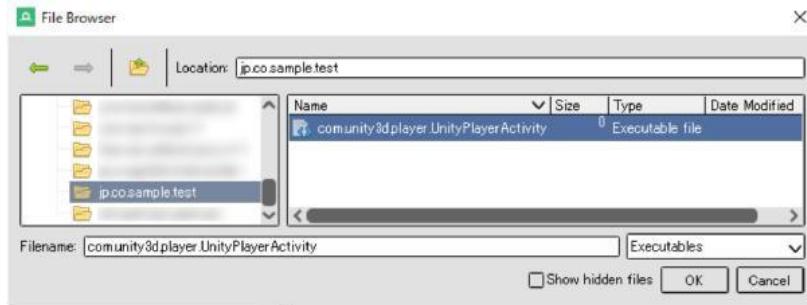
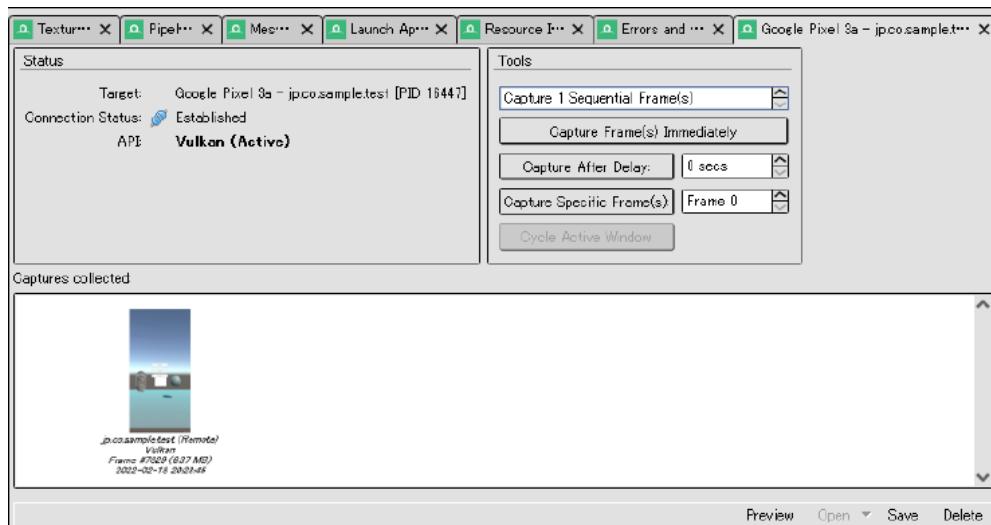


图3.118 选择要测量的应用程序

最后，按下“启动应用程序”中的“启动”按钮将在设备上启动该应用程序。此外，在 RenderDoc 上增加了一个新的标签，用于测量。

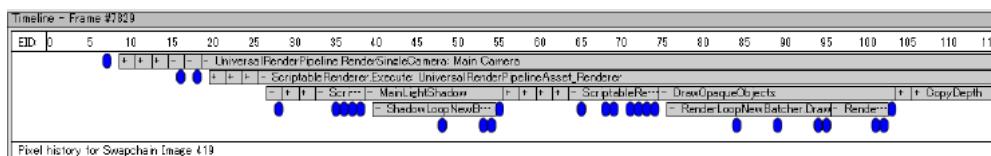


▲ 图3.119 用于测量的标签

如果你按下立即捕获帧（s）作为试验，帧数据将被捕获并在捕获收集中排成一列。双击该数据以打开捕获的数据。

3.9.2 如何查看捕获数据

RenderDoc有广泛的功能，但在本节中，我们将集中讨论功能中最重要的部分。首先，在屏幕的顶部显示了捕获的帧的时间线。你可以直观地看到每条绘图命令是按照什么顺序执行的。



▲图3.120时间轴。

接下来是事件浏览器。这里按从上到下的顺序列出了每条命令。

The screenshot shows the RenderDoc Event Browser window. The interface includes a toolbar with filters for 'Action' (set to 'Event'), 'Duration' (set to '0.00'), and other settings. The main area is a table titled 'Event Browser' with columns for 'EID', 'Name', and 'Duration (μs)'. The table lists several events, many of which are collapsed under category headers. The events shown include 'ScriptableRenderPass.Configure', 'DrawOpaqueObjects', 'RenderLoopNewBatcher.Draw', 'vkCmdDrawIndexed', and 'RenderLoop.Draw'. The duration column provides a rough estimate of the execution time for each command.

EID	Name	Duration (μs)
64-73	ScriptableRenderPass.Configure	0.00
75-102	DrawOpaqueObjects	4.59125
76-94	RenderLoopNewBatcher.Draw	3.75
84	vkCmdDrawIndexed(600, 1)	2.13542
89	vkCmdDrawIndexed(2304, 1)	0.83333
94	vkCmdDrawIndexed(36, 1)	0.78125
96-101	RenderLoop.Draw	0.78125
101	vkCmdDrawIndexed(2496, 1)	0.78125
104	ScriptableRenderPass.Configure	0.00

图3.121 事件浏览器。

在事件浏览器中按顶部的“时钟符号”，在“持续时间”中显示每个命令的处理时间。处理时间根据测量时间的不同而不同，所以它只能被视为一个粗略的指导。另外，对DrawOpaqueObjects命令的细分显示，有三个是批量处理的，只有一个是非批量外绘制的。

下文介绍了每个窗口的情况，这些窗口在右侧的标签中排列。在这些标签中，你会发现

在事件浏览器中有一个窗口，你可以看到关于所选命令的详细信息。

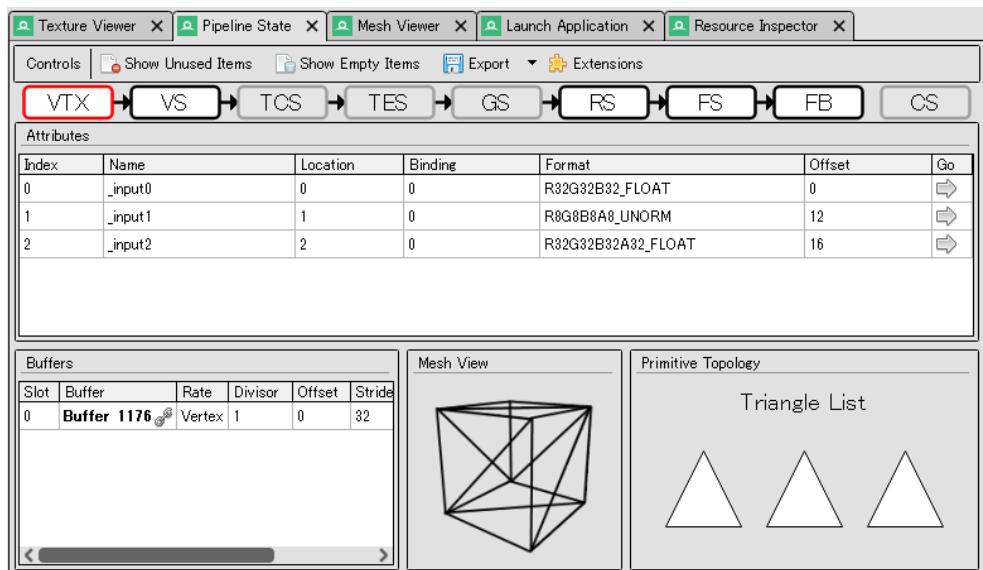
3.9 RenderDoc

特别重要的是网格浏览器、纹理浏览器和管道状态。



图3.122 每个窗口。

首先，让我们来谈谈管道状态，在这里你可以看到在物体被绘制到屏幕之前，每个着色器阶段使用了哪些参数。你还可以查看所使用的着色器和它们的内容。



▲图3.123 管线状态。

在管道状态下显示的阶段性名称是缩写，因此代表官方名称。
总结于3.7。

▼ 表3.7 PipelineState的正式名称

舞台名称	正式名称
VTX	顶点输入
versus (vs, v.)	顶点着色器
TCS	镶嵌控制着色器
TES	镶嵌评估着色器
ÂÂÂ	几何图形着色器
ÂÂÂ	光栅器
雇员	片段着色器
基金会	帧缓冲器
CS	计算着色器

在图3.123中，选择了VTX阶段，在这里你可以看到拓扑结构和顶点输入数据。其他细节，如输出目标纹理状态和混合状态，可以在图3.124的FB阶段看到。

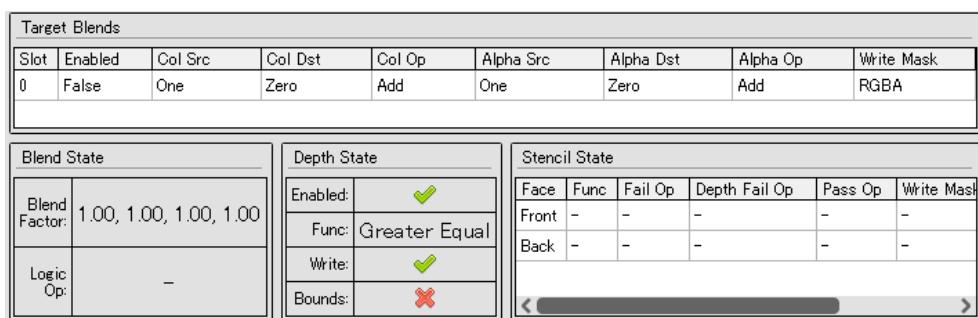


图3.124 FB（帧缓冲器）的状态

图3.125中的FS阶段也允许用户看到碎片着色器中使用的纹理和参数。

3.9 RenderDoc.

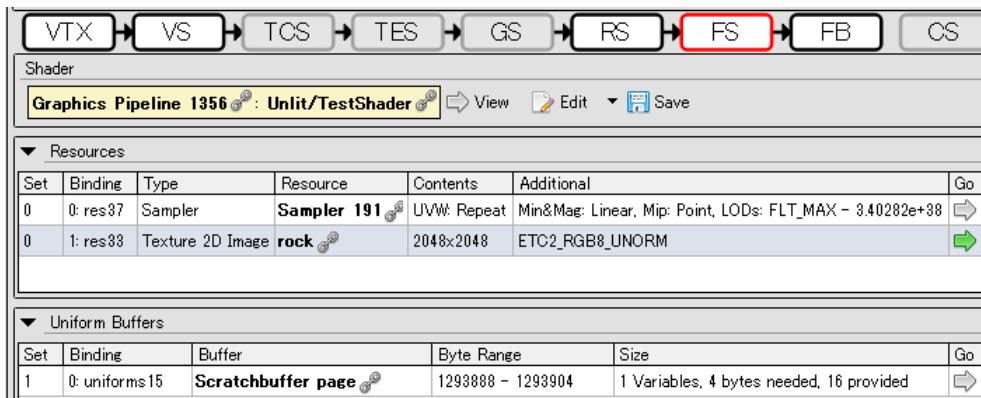


图3.125 FS的状态（碎片着色器）。

在FS阶段中间的资源显示了所使用的纹理和采样器。在FS阶段底部的统一缓冲区部分，显示了CBuffer。这个CBuffer包含数字属性，如平面和颜色。每个项目的右边都有一个“前进”的箭头图标，可以按下它来查看更详细的数据。

所用的着色器显示在FS阶段的上部；按“查看”可以看到着色器代码。对于Disassembly类型，推荐使用GLSL，以使显示更容易理解。

The screenshot shows the RenderDoc Disassembly view. The "Disassembly type" dropdown is set to "GLSL (SPIRV-Cross)". The code listed is:

```
1 #version 450
2
3 layout(set = 1, binding = 0, std140) uniform _13_15
4 {
5     float _m0;
6 } _15;
7
8 layout(set = 0, binding = 1) uniform medium texture2D _33;
9 layout(set = 0, binding = 0) uniform medium sampler _37;
```

▲ 图3.126 检查着色器代码。

接下来是网格浏览器。这允许你直观地查看网格信息，对优化和调试非常有用。

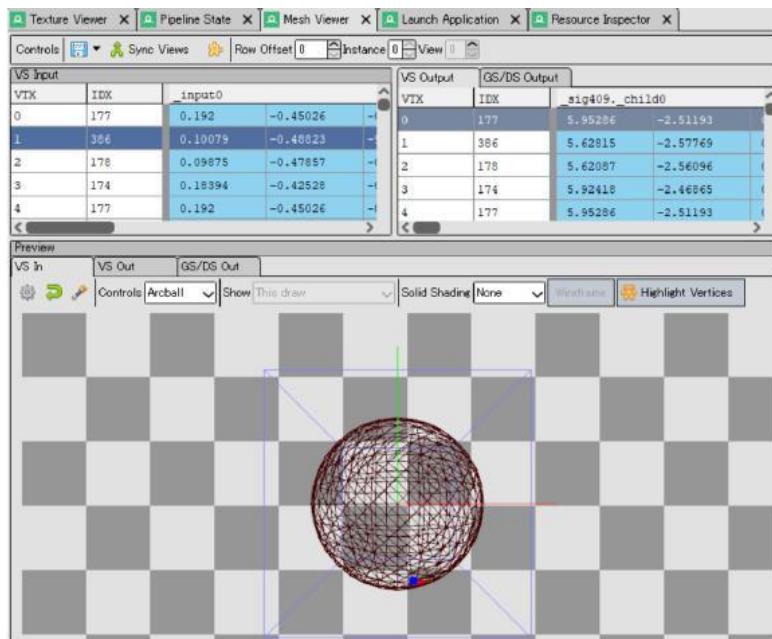


图3.127 网格浏览器。

网格浏览器的上部以表格形式显示网格顶点信息。下部有一个预览屏幕，可以通过移动相机来检查形状。两者都有单独的输入和输出标签，所以你可以看到转换前后的数值和外观的变化。

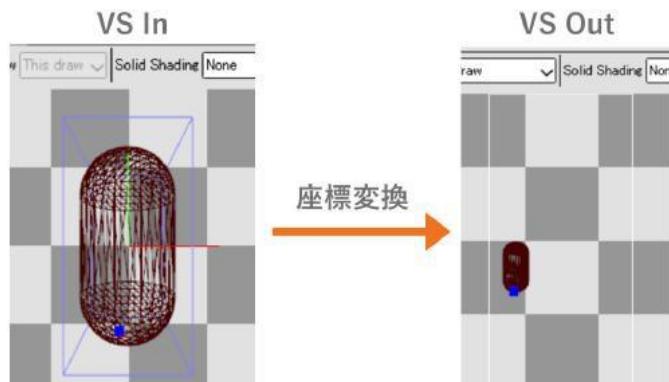


图3.128 网格浏览器中In和Out的预览显示。

最后，还有纹理查看器。这个屏幕显示在事件浏览器中选择的命令的“用于输入的纹理”和“输出结果”。

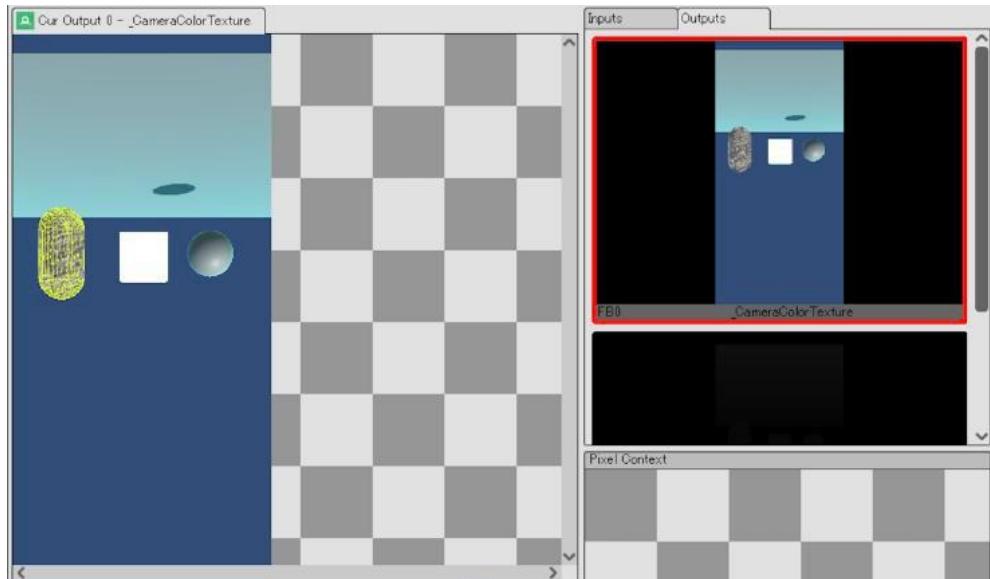


图3.129 纹理查看器的纹理验证屏幕。

在屏幕的右侧区域，你可以看到输入和输出的纹理。按下所显示的纹理将在屏幕的左侧区域反映出来。屏幕的左侧不仅可以显示，还可以过滤颜色通道和应用工具栏设置。

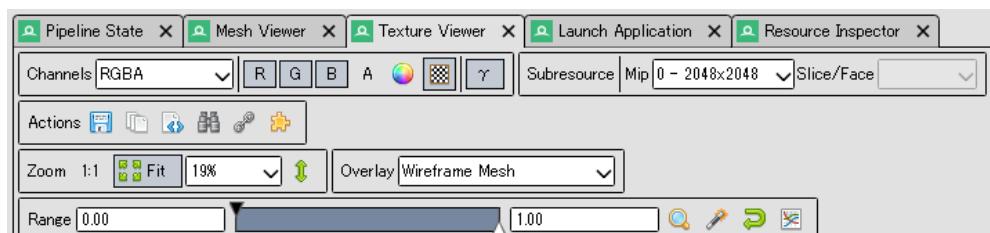


图3.130 纹理查看器工具条。

在图3.129中，'Wireframe Mesh' 被选为Overlay，所以用这个命令绘制的对象有一个黄色的线框显示，视觉上更容易理解。

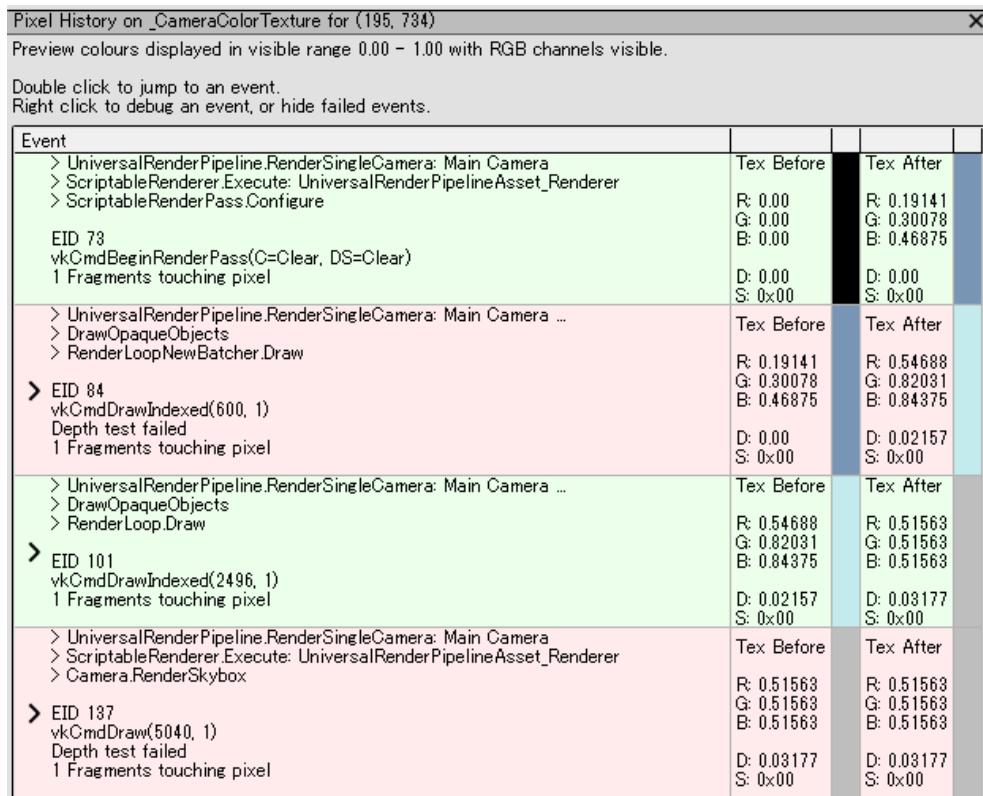
纹理查看器也有一个叫做“像素背景”的功能。该功能允许你查看所选像素的绘制历史。历史记录意味着你可以看到一个像素被填充的频率。这是一个调查和优化超限的有用功能。然而，它只是在每个像素的基础上，所以它不适合在全局基础上调查过度绘制。要调查，在图3.129左侧的屏幕上右击你要调查的区域，位置会反映在像素上下文中。



图3.131 对像素背景的反思

接下来，按下像素上下文中的历史按钮，可以看到该像素的绘制历史。

3.9 RenderDoc.



▲ 图3.132 像素绘制历史

在图3.132中，有四段历史。绿线表示所有的管道测试，如深度测试，已经通过，管道已经绘制完毕。如果一些测试失败，没有被画出来，则为红色。在捕获的图像中，屏幕清除过程和胶囊绘制都是成功的，而平面和天盒则未能通过深度测试。



性能调谐圣经

CHAPTER

04

第4章

Tuning Practice
— Asset —

CyberAgent Smartphone Games & Entertainment

第四章。

调试实践--资产

游戏制作涉及处理大量不同类型的资产，如纹理、网格、动画和声音。因此，本章总结了有关这些资产的实用知识，包括调整性能时需要注意的设置。

4.1 纹理

作为纹理来源的图像数据是游戏制作的一个重要部分。另一方面，内存消耗相对较高，所以设置必须适当。

4.1.1 进口设置

图4.1显示了Unity中的纹理导入设置。

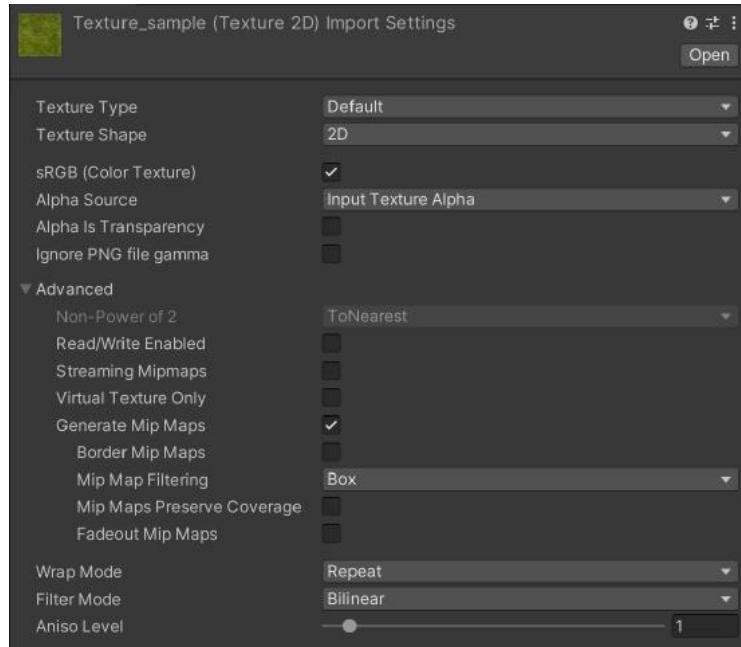


图4.1 纹理设置

以下是一些需要注意的一些最重要的情况。

4.1.2 读/写

该选项在默认情况下是禁用的。如果禁用，纹理只被部署到GPU内存。如果启用，它们不仅被复制到GPU内存，而且还将被复制到主内存，因此消耗的数量是两倍。因此，如果你不使用Texture.**GetPixel**或Texture.**SetPixel**等API，只在Shader中访问它们，请确保禁用它们。

如清单4.1所示，通过设置**makeNoLonger-Readable**为true，也可以避免运行时生成的纹理被复制到主内存中。

▼ 清单 4.1配置 makeNoLongerReadable

```
1 : texture2D.Apply(updateMipmaps, makeNoLongerReadable: true)
```

纹理从GPU内存传输到主内存是很耗时的，所以如果可以读取的话，通过在两者中部署纹理可以提高性能。

4.1.3 生成Mip地图

启用Mip Map设置会使纹理内存的使用量增加约1.3倍。这个设置一般用于三维物体，以减少锯齿和远处物体的纹理传输；对于二维精灵和用户界面图像来说，它基本上是不必要的，所以它应该被禁用。

4.1.4 阿尼索水平

Aniso Level是一个允许以浅角度绘制物体的功能，而不会使纹理的外观变得模糊。这一功能主要用于延伸较远的物体，如地面或地板；Aniso Level值越高，它提供的好处越多，但处理成本也越高。

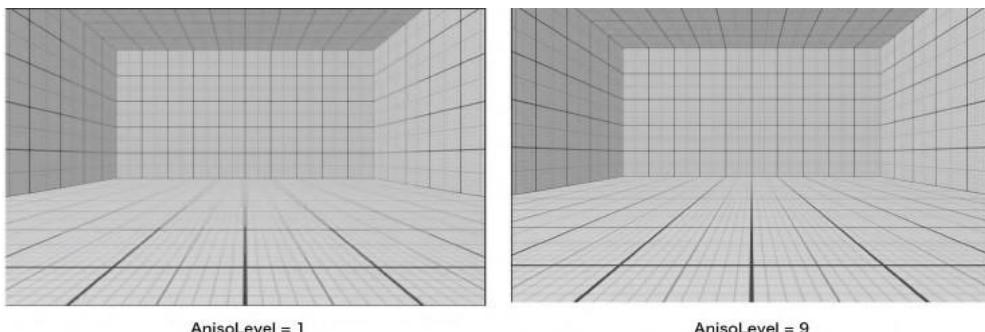
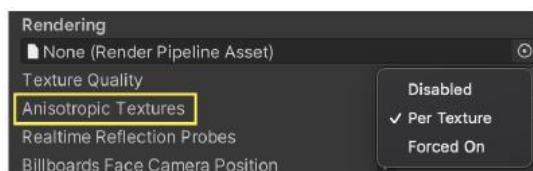


图4.2 Aniso水平适应图像

设定的数值从0到16，但规格有点特殊。

- 0: 无论项目设置如何，总是禁用
- 1: 基本上禁用。但是，如果项目设置为强制开启，则数值被钳制在9~16之间。
- 否则：设置为该值

当纹理被导入时，该值默认被设置为1。因此，除非你使用高规格的设备，否则不建议强制开启；强制开启可以在“项目设置->质量”中的各向异性纹理下设置。



▲图4.3 强制开启设置。

检查无效的对象是否启用了Aniso级别设置，或者对于有效的对象是否设置得过高。

Aniso Level的效果不是线性的，而是分阶段切换行为。笔者验证过，它的变化分为四个阶段：0-1、2-3、4-7和8以后。

4.1.5 压缩设置

纹理应该被压缩，除非有特定的理由不这样做。如果项目中出现了未压缩的纹理，可能是人为错误或超出了规定。尽快检查这个问题。关于压缩设置的更多信息，见2.3.3压缩图像一节。

建议使用TextureImporter来自动进行这些压缩设置，以避免人为错误。

▼清单4. 2TextureImporter自动化实例

```

1: 使用UnityEditor。
2:
3: public class ImporterExample : AssetPostprocessor
4: {
5:     private void OnPreprocessTexture()
6:     {
7:         var importer = assetImporter as TextureImporter;
8:         importer.isReadable = false; // 读/写设置等可能 9:
10:        var settings = new TextureImporterPlatformSettings();
11:        // Android = "Android", PC = "Standalone"
12:        settings.name = "iPhone";
13:        settings.overridden = true;
14:        settings.textureCompression = TextureImporterCompression.Compressed;
15:        settings.format = TextureImporterFormat.ASTC_6x6; // 指定压缩格式
16:        importer.SetPlatformTextureSettings(settings);
17:    }
18: }
```

另外，不是所有的纹理都需要采用相同的压缩格式。例如，在用户界面图像中，具有整体梯度的图像很可能会因为压缩而出现明显的质量损失。在这种情况下，建议只对部分目标图像设置较低的压缩率。相反，3D模型等纹理不太可能出现质量损失，因此建议找到一个合适的设置，如高压缩比。

4.2 网络。

下面是在Unity中处理导入的网格（模型）时需要考虑的一些要点。导入的模型数据的性能可以根据设置来改善。有四点需要注意

- 启用了读/写功能
- 顶点压缩
- 网眼压缩
- 优化网格数据

4.2.1 启用了读/写功能

第一个Mesh音符是读/写启用。模型检查器中的这个选项默认是禁用的。

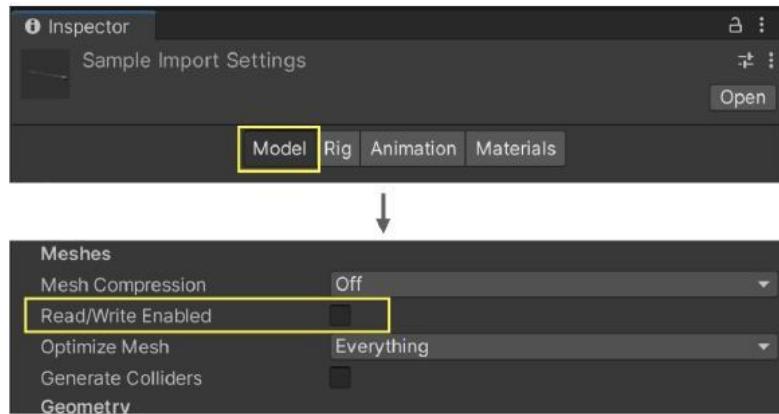


图4.4 读/写设置。

如果你不需要在运行期间访问网格，请将其关闭。具体来说，如果模型被放置在Unity上，只用来播放AnimationClip，可以禁用读写功能而不会有任何问题。

如果启用，CPU可访问的信息被保存在内存中，从而消耗两倍的内存。请通过简单地禁用它来检查内存的节省情况。

4.2.2 顶点压缩

顶点压缩是一个选项，可以将网格顶点信息的精度从平面变为半平面。这减少了运行时的内存使用和文件大小。它可以在“其他”中的“项目设置->播放器”中设置，默认设置如下。

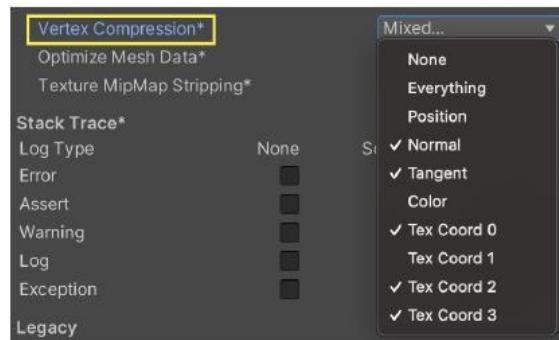


图4.5 顶点压缩的默认设置。

然而, 请注意, 如果满足以下条件, 该顶点压缩功能将被禁用

- 启用读/写功能。
- 启用网格压缩
- 启用动态批处理并可适应的网格 (少于300个顶点和少于900个顶点属性)。

4.2.3 网眼压缩

网格压缩允许你改变网格的压缩率。压缩率越高, 文件大小就越小, 所需的存储量就越少。压缩的数据在运行时被解压缩。这对运行时的内存使用没有影响。

在 Mesh Compression 中的压缩设置有四个选项。

- 关 : 无压缩。
- 低 : 低压缩量
- 中度 : 中度压缩。
- 高 : 高压缩

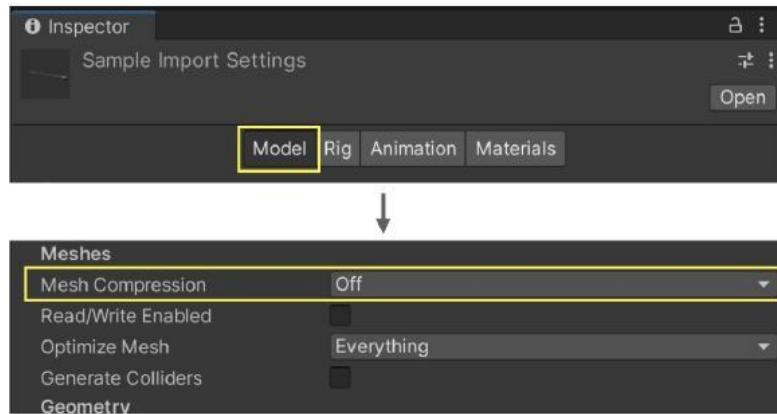


图4.6 网格压缩。

正如第4.2.2节顶点压缩中提到的，启用该选项可以禁用顶点压缩。特别是在有严格内存使用限制的项目中，请在设置该选项前注意这一缺点。

4.2.4 优化网格数据

优化网格数据会自动删除网格中不必要的顶点数据。不必要的顶点数据是由使用中的着色器自动确定的。这具有在运行时减少内存和存储的效果。这可以在“项目设置”->“播放器”的“其他”中设置。



图4.7 优化网格数据设置。

然而，虽然这个选项很方便，因为顶点数据会被自动删除，但要注意，它可能会引起意想不到的问题。例如，在运行时在材料和着色器之间切换时，所访问的属性可能会被删除，从而导致不正确的绘图结果。在其他情况下，当只捆绑Mesh作为资产捆绑时，如果Material设置不正确，可能会被判定为不必要的顶点数据。这在只有对网格的引用时

很常见，比如在粒子系统中。

4.3 材料

材料是一个重要的特征，决定了物体的绘制方式。虽然这是一个熟悉的功能，但如果使用不当，很容易造成内存泄漏。本节告诉你如何安全使用材料。

只通过访问参数来重复。

材料的主要注意事项是，仅仅通过访问它们的参数就可以复制它们。而很难注意到它正在被复制。

看一下下面的代码。

▼ 清单 4.3 被复制的材料的例子。

```

1、材料材质。
2:
3: void Awake()
4: {
5:     material = renderer.material;
6:     Material.color = Colour.green;
7: }
```

这是一个简单的过程，将材料的颜色属性设置为color.green。渲染器的材质被复制了。然后，复制的对象必须明确地被销毁。

▼ 清单 4.4 删除重复材料的例子

```

1、材料材质。
2:
3: void Awake()
4: {
5:     material = renderer.material;
6:     Material.color = Colour.green;
7: }
8:
9: void OnDestroy().
10: {
11:     如果 (材料 != null)
12:     {
13:         销毁(材料)
14:     }
15: }
```

以这种方式销毁重复的材料可以避免内存泄漏。

彻底清理产生的材料。

动态生成的材料也很容易出现内存泄漏。确保在你使用完后销毁所产生的材料。

请看下面的示例代码。

▼清单4.5。 动态生成的材料删除的例子

```
1、 材料材质。  
2:  
3: void Awake()  
4: {  
5:     material = new Material(); // Material is generated dynamically.  
6: }  
7:  
8: void OnDestroy()  
9: {  
10:    如果 (材料 != null)  
11:    {  
12:        Destroy(material); // 销毁使用过的材料 13:    }  
14: }
```

当你用完后，销毁生成的材料（OnDestroy）。根据项目的规则和规范，在适当的时间销毁材料。

4.4 动画

动画是一种广泛使用的资产，无论是2D还是3D。本节介绍了与Animation Clip和Animator有关的做法。

4.4.1 调整皮肤重量的数量

运动内部通过计算每个顶点受每个骨骼影响的程度来更新位置。在位置计算中考虑到的骨骼数量被称为皮肤重量或影响因素的数量。因此，可以通过调整皮重物的数量来减少负荷。然而，减少皮肤重量的数量可能会导致一个奇怪的外观，所以在调整时要核实这一点。

皮肤权重的数量可以从'项目设置->质量'其他方面设置。



▲ 图4.8 调整皮肤重量

这个设置也可以从脚本中动态调整。因此，它可以被微调，例如，对于低规格的设备，皮肤权重设置为2，对于高规格的设备设置为4。

▼清单4. 6皮肤重量的配置变化

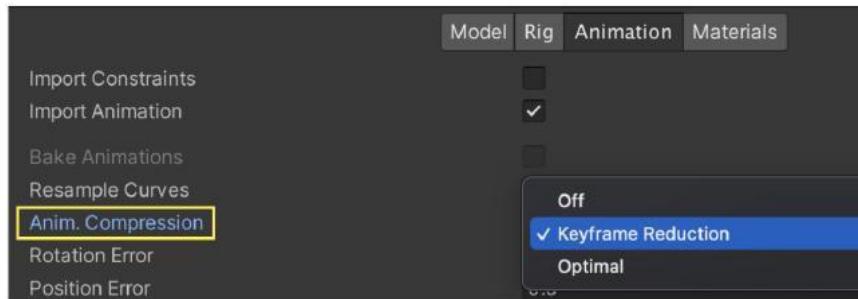
```

1: //如何完整地切换QualitySettings。
2: // 参数编号按照配置屏幕上的Levels顺序从上往下依次为0、1....。 ... 按配置屏幕上的级别顺序排列
。
3: QualitySettings.SetQualityLevel(0)。
4:
5: // 如何只改变SkinWeights
6: QualitySettings.skinWeights = SkinWeights.TwoBones。

```

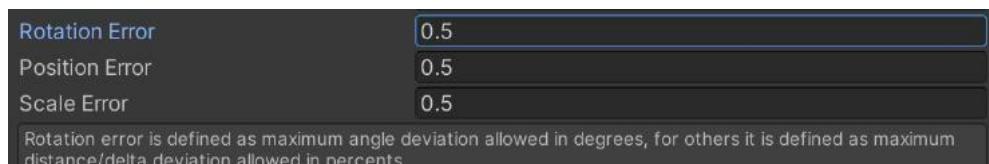
4.4.2 关键的减少

动画文件的存储和运行时的内存消耗很大，这取决于按键的数量。减少按键数量的一个方法是使用一种叫做Anim的功能。这个选项可以通过选择模型导入设置中的动画选项卡来找到；如果启用了Anim.Compression，在导入资产时将自动删除不必要的键。



▲ 图4.9 Anim. 压缩设置屏幕。

如果数值的变化很小，关键帧的减少会减少关键。具体来说，如果与前一条曲线相比，钥匙在误差（Error）范围内，就会被删除。这个误差范围是可以调整的。



▲ 图4.10 错误设置。

让事情变得复杂一点的是，误差设置中的每一项都有不同的数值单位--旋转是度，位置和比例是百分比。捕获的图像的公差为旋转0.5度，位置和比例为0.5%。详细的算法可以在Unity文档中找到*1。

最优是更令人困惑的，但它比较了两种还原方法，密集曲线格式和关键帧还原，并采用了数据较小的那一种。要记住的关键点是，Dense的尺寸比Keyframe Reduction小。然而，它往往比较嘈杂，可能会导致动画质量下降。在了解了这些特点之后，你应该目测一下实际的动画，看看它是否可以接受。

*1 <https://docs.unity3d.com/Manual/class-AnimationClip.html#tolerance>

4.4.3 减少了更新的频率

Animator默认更新每一帧，即使它不在屏幕上。有一个叫做“剔除模式”的选项，允许你改变这种更新方法。

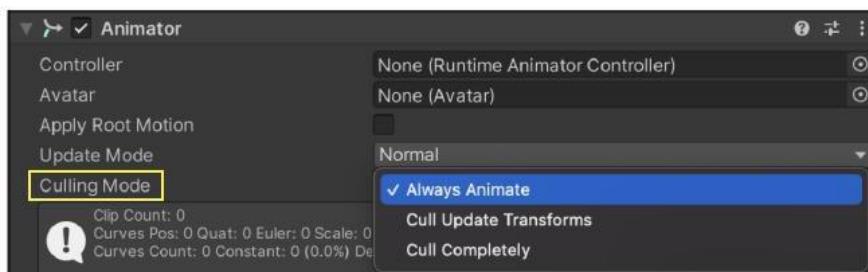


图4.11 剔除模式。

每个选项的含义如下。

表4.1 剔除模式的描述

类型。	意义
始终保持动画。	即使在屏幕外，也始终更新。(默认设置)
Cull 更新转换	离屏时不写 IK 或 Transform。 状态机更新确实如此。
彻底清除	如果他们不在屏幕上，他们不会更新状态机。 动画完全停止。

关于这些选项，有一些要点需要注意。首先，在设置Cull Completely时，使用Root motion时必须注意。例如，如果你有一个动画从屏幕外的帧进来，该动画将立即停止，因为它是在屏幕外。其结果是，动画不会永远定格。

接下来是Cull Update Transform。这似乎是一个非常可用的选项，因为它只是跳过了转变的更新。然而，如果你有变形依赖的操作，如摇晃，就要小心。例如，如果角色定格了，更新将从那一刻的姿势跳过。当角色再次定格时，它将被更新为一个新的姿势，这可能会导致晃动的物体大幅移动。建议在改变设置之前，了解每个选项的利弊。

这些设置也不允许对动态动画的更新频率进行微调。例如，你可以优化动画的更新频率，对于离摄像机较远的物体，将其减半。在这种情况下，你需要使用 AnimationClipPlayable 或者停用 Animator 并自己调用 Animator.Update。两者都需要编写自己的脚本，但后者比前者更容易实现。

4.5 粒子系统

游戏效果是游戏表现的一个重要部分，在 Unity 中经常使用粒子系统。本章从性能调整的角度介绍了粒子系统的使用，以及避免故障的注意点。

两个重要的观点是

- 减少颗粒的数量。
- 注意，噪音很重。

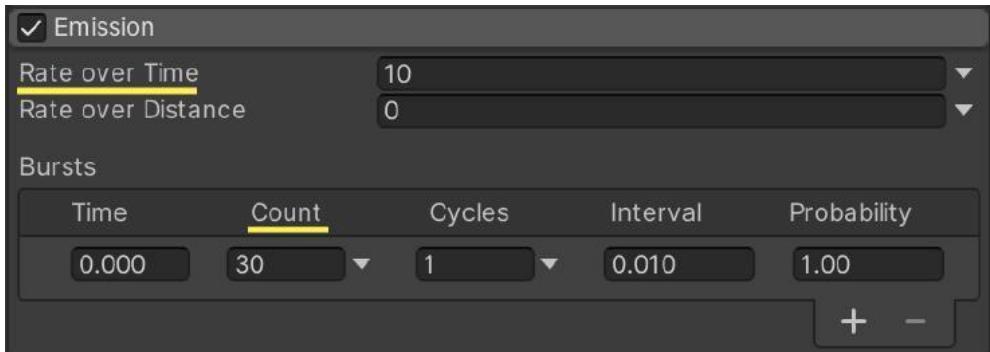
4.5.1 减少颗粒的数量。

粒子的数量与负荷有关：由于粒子系统是由 CPU 驱动的（CPU 粒子），粒子越多，CPU 的负荷越高。作为一项基本政策，将颗粒的数量设置为必要的最小值。根据需要调整颗粒的数量。

有两种方法来限制粒子的数量。

- 排放的模块数量的限制。
- 主模块中的“最大颗粒”限制了释放的最大零件数量。

4.5 粒子系统



▲ 图4.12 排放模块对排放数量的限制

- 速率随时间变化：每秒钟释放的件数
- 突发事件 > 计数：在突发事件发生时要释放的件数

调整这些设置以达到所需的最小颗粒数。

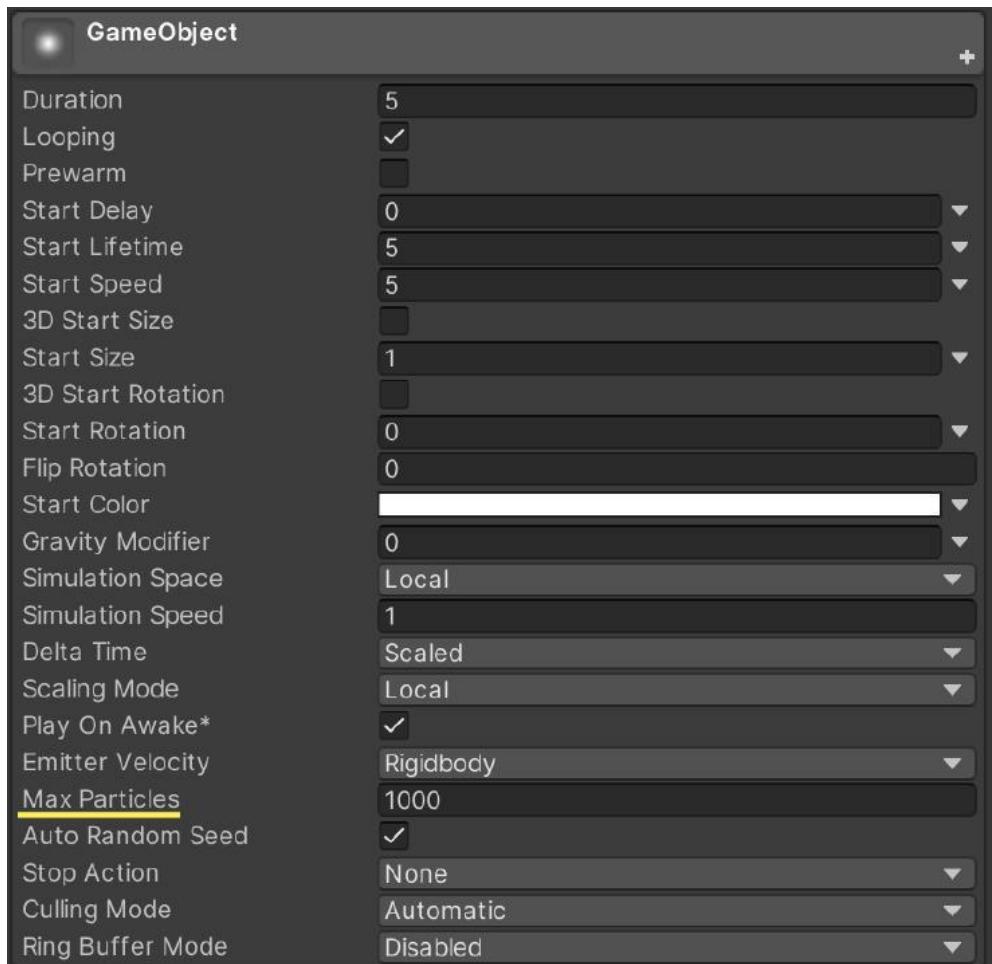


图4.13 最大颗粒限制了释放的零件数量。

另一种方法是主模块中的Max Particles。在上面的例子中，超过1000个颗粒将不再被排放。

还要注意副发射器。

在限制粒子的数量方面，子发射器模块也需要注意。

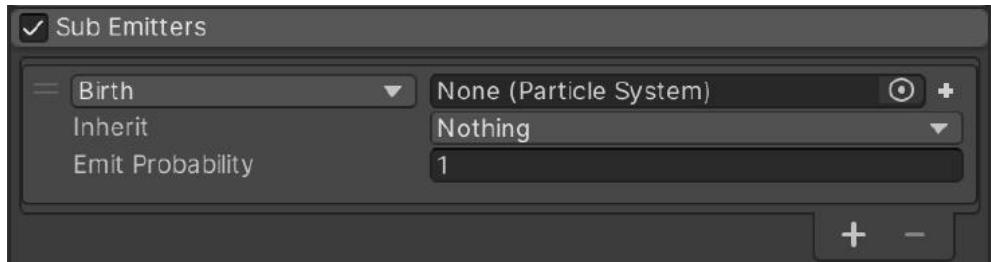


图4.14 子发射器模块。

子发射器模块在特定时间生成任意的粒子系统（例如，在创建时，在生命结束时，等等），使用子发射器时要小心，因为有些设置可以一次性达到峰值数量。

4.5.2 注意，噪音很重。

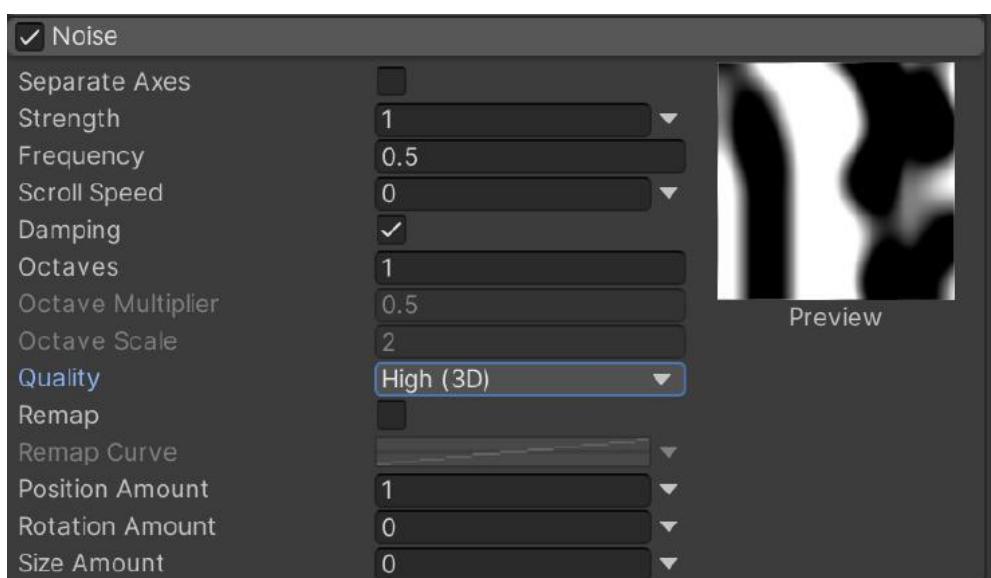


图4.15 噪声模块。

噪音模块的质量很容易过载。噪音可以用来创造有机颗粒，是提高效果质量的一个简

单方法。

这种功能最常见的用途是在使用计算机的内存。因为它是一个经常使用的功能，你要小心它的性能。

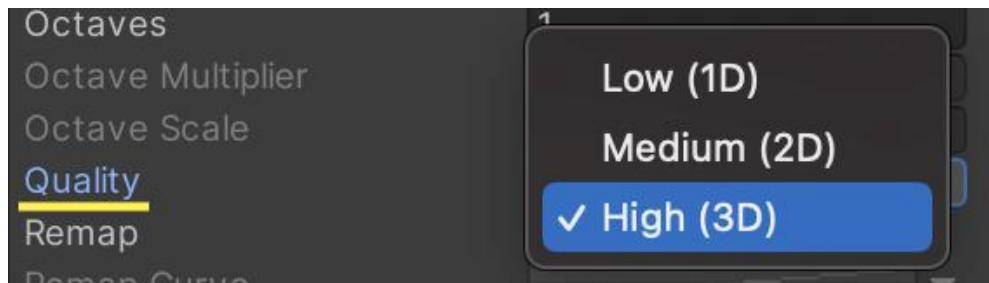


图4.16 噪声模块的质量

- 低 (1D)
- Midium (2D).
- 高 (3D)

质量的负荷随着尺寸的增加而增加。如果你不需要噪声，请关闭噪声模块。如果你需要使用噪音，先将质量设置为低，然后根据你的要求提高质量。

4.6 音频

导入声音文件的默认状态在性能上是一个改进点。有三个配置项目。

- 负载类型
- 压缩格式
- 强制转为单声道

将这些设置为适当的背景音乐、声音效果和游戏中常用的声音。

4.6.1 负载类型

有三种方法来加载声音文件 (AudioClip)。



图4.17 AudioClip LoadType.

- 加载时解压
- 压缩在内存中
- 流媒体

加载时解压

加载时解压是将未压缩的内容加载到内存中；它对CPU的要求较低，因此播放时的等待时间较短。另一方面，它使用更多的内存。

建议用于长度较短、需要立即回放的声音效果；与背景音乐或长的语音文件一起使用时应注意，因为它们会占用大量的内存。

压缩在内存中

在内存中压缩，以压缩状态将AudioClip加载到内存中。这意味着它在播放时被解压缩。这意味着CPU负载很高，播放延迟很可能发生。

适用于文件大小较大，不希望直接部署到内存中的声音，或对有轻微播放延迟没有问题的声音。它经常与语音一起使用。

流媒体

流媒体，顾名思义，是一种加载和播放方法。内存使用率很低，但CPU负载很高。建议与长的BGM一起使用。

▼ 表4.2 装载方法和主要用途摘要

类型。	使用
-----	----

加载时解压	音效
压缩在内存中	声音
流媒体	BGM

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。对于嘈杂和高音量的声音，如脚步声、碰撞声和武器声，情况尤其如此。

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	音效

分脉冲编 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。

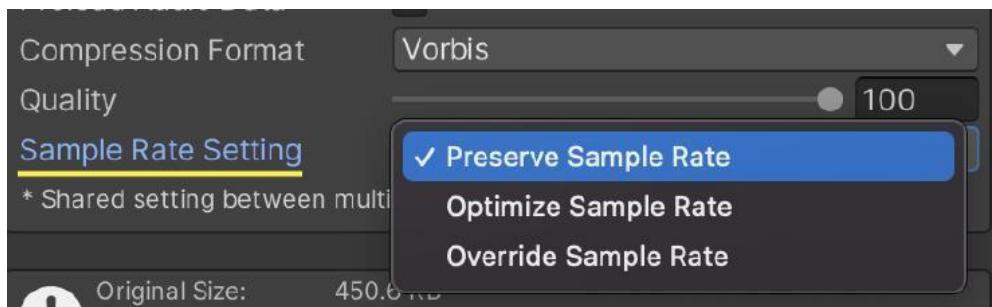


图4.19 采样率设置

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。高于原始音源并不能提高质量。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。



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

在项目中，有一些特殊的文件夹。从性能的角度来看，以下两点尤其需要注意

- 资源文件夹
- StreamingAssets文件夹。

Unity通常只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，即使是实际上不需要的文件，如果它们被储存起来，也会被包括在构建中，而且构建的sa

4.8 脚本对象

这导致了是的扩张。

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 减缓启动时间的资源文件夹

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 ScriptableObject。

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入和读取的性能。

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**]。

第4章 调试实践--资产

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: }
```



性能调谐圣经

CHAPTER

05

第5章

Tuning Practice
— AssetBundle —

CyberAgent Smartphone Games & Entertainment

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。本章描述了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。这是因为即使你只增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，严格的内存管理是很严重的，所以如果内存负载足够，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有一个上限，所以不可能同时有超过这个数量的文件打开。“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Persist有entManager.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。



性能调谐圣经

CHAPTER

06

第6章

Tuning Practice
— Physics —

CyberAgent Smartphone Games & Entertainment

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，也就是20毫秒。

6.2 固定时间步长和固定更新频率的优化

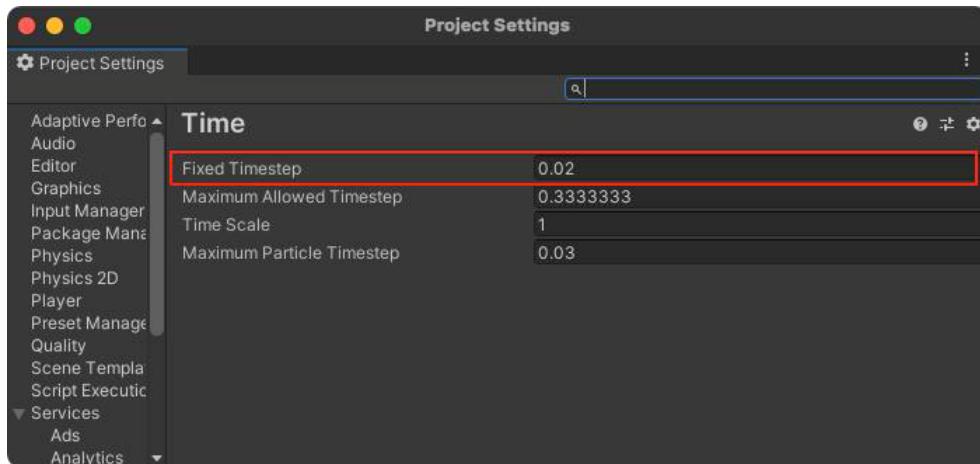


图6.1 项目设置中的固定时间段项目

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，但只要不引起游戏中的任何行为问题，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, 物理操作在一帧内可以使用的最大时间量。

第6章调谐练习--物理学

可以设置一个大值。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。

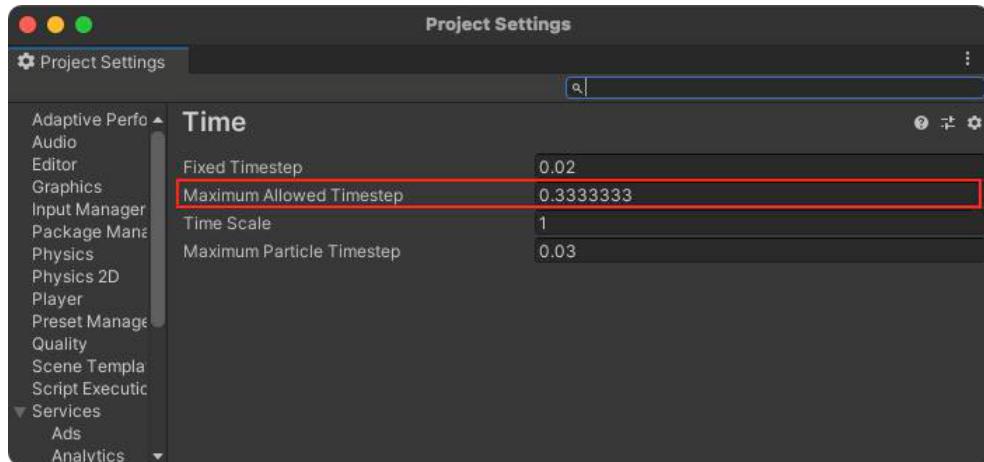


图6.2 项目设置中的最大允许时间步长项目

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。

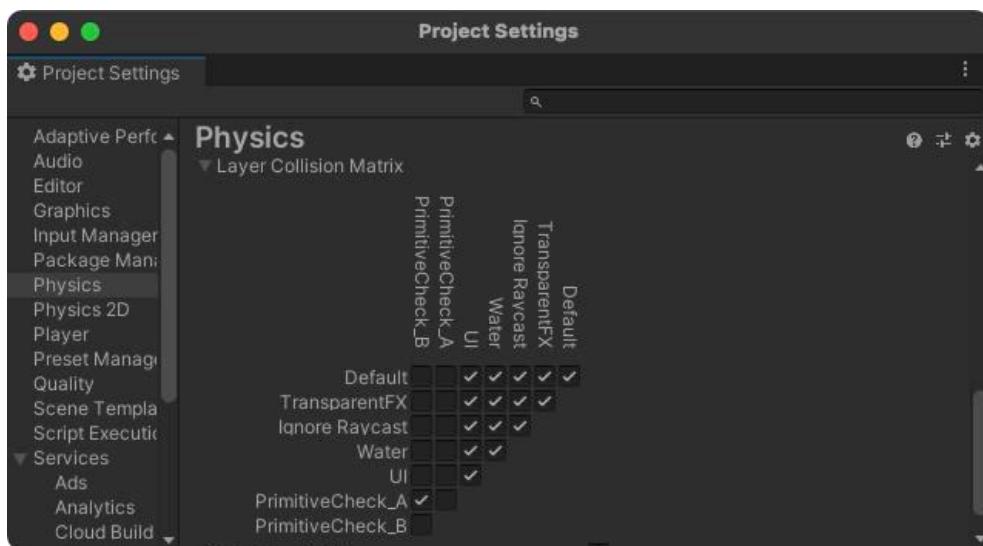


图6.3 项目设置中的图层碰撞矩阵项目

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还确保不需要作为命中的层中的位不被设置，而不需要作为命中的层中的位不被设置。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，有一个叫做Physics.RaycastNonAlloc的方法，当把一个分配好的数组作为参数传给他时，他会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: // 使用池中保留的内容
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:     指示：
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，即这个对撞机只用于总是停留在同一地点而从不移动的几何体。

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学引擎与其他物体碰撞。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

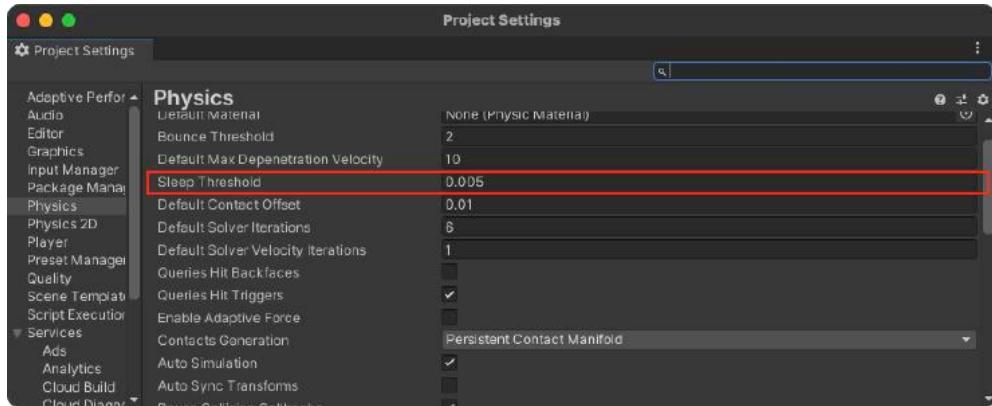


图6.4 项目设置中的睡眠阈值项目

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

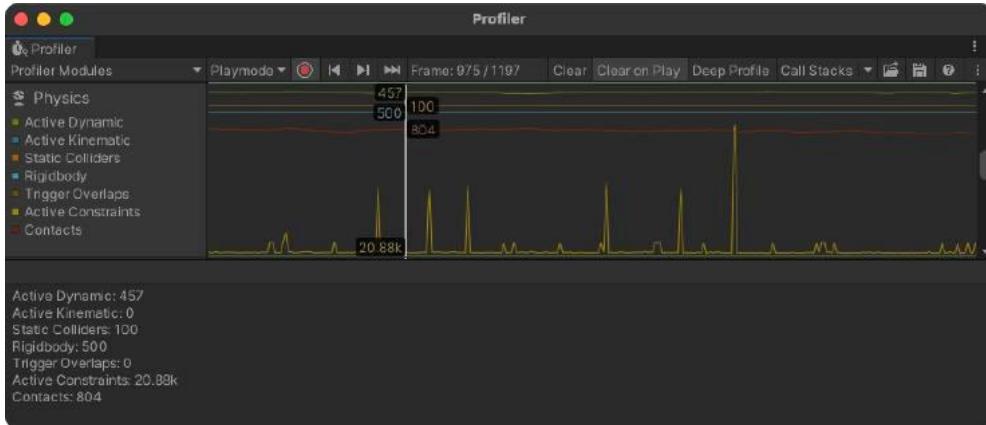


图6.5. 物理学条目在Profiler中。你可以看到活动的刚体的数量，以及物理引擎上每个元素的数量。

物理学调试器也可以用来查看场景中哪些物体是活动的。

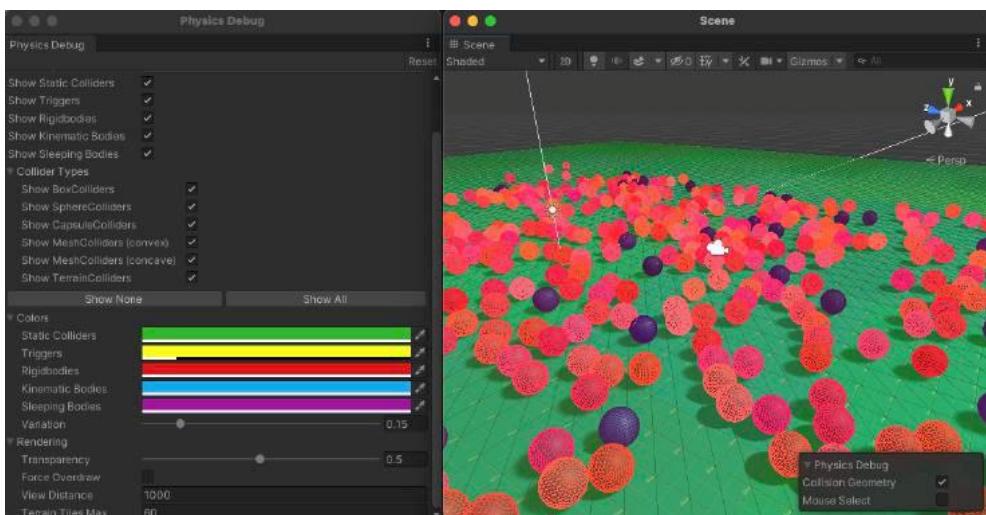


图6.6. 物理调试器，显示场景中的物体在物理引擎方面的状态，用颜色编码。

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。

如果有什么不方便的地方，可以考虑连续碰撞测定：连续只对动态和静态碰撞器组合有效，而连续动态对动态碰撞器也有效。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，因为它可能导致所谓的**幽灵碰撞（Ghost Collision）**现象，即多个碰撞器距离很近，相互之间的碰撞是错误的。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相互碰撞。

6.8 其他项目设置的优化

除了到目前为止介绍的设置，本节还介绍了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。



性能调谐圣经

CHAPTER

07

第7章

Tuning Practice
— Graphics —

CyberAgent Smartphone Games & Entertainment

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）。

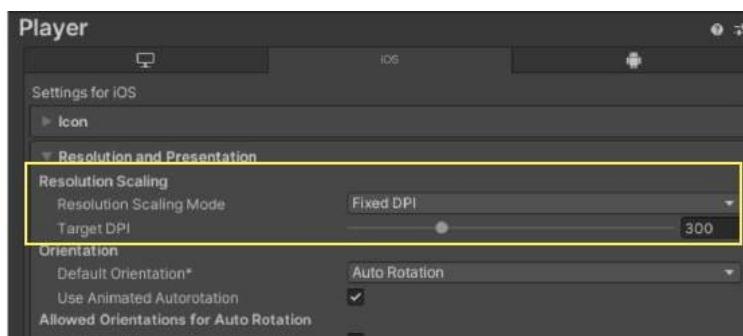


图7.1 分辨率缩放模式。

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。

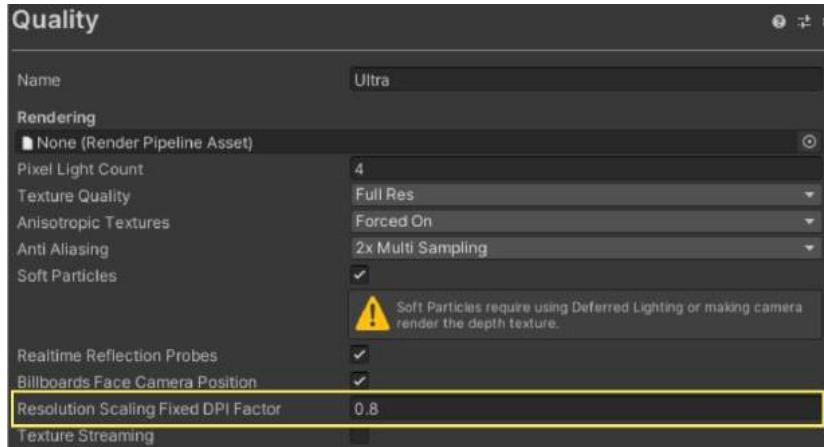


图7.2 分辨率比例 DPI比例系数

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。

当前分辨率可以通过Screen.width或Screen.height获得，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

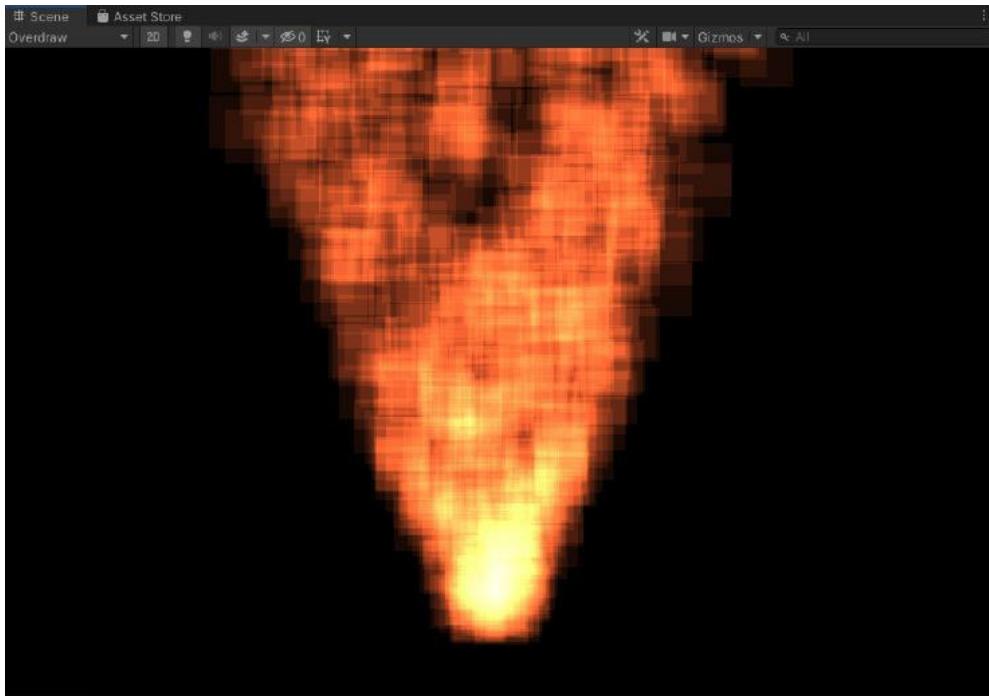


图7.3 超拔模式。

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

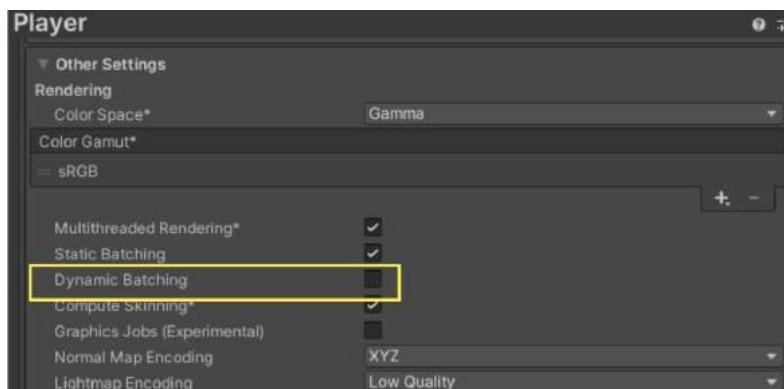
第7章 调试实践--图形

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**

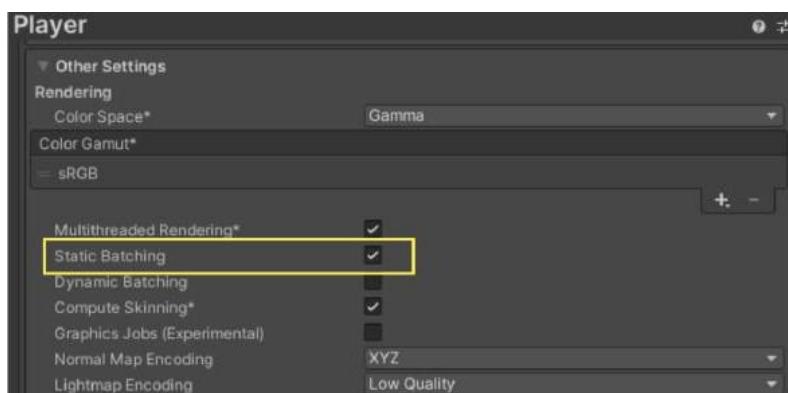
Batcher, 可以用来实现类似于动态配料的效果。

7.3 减少平局调用

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

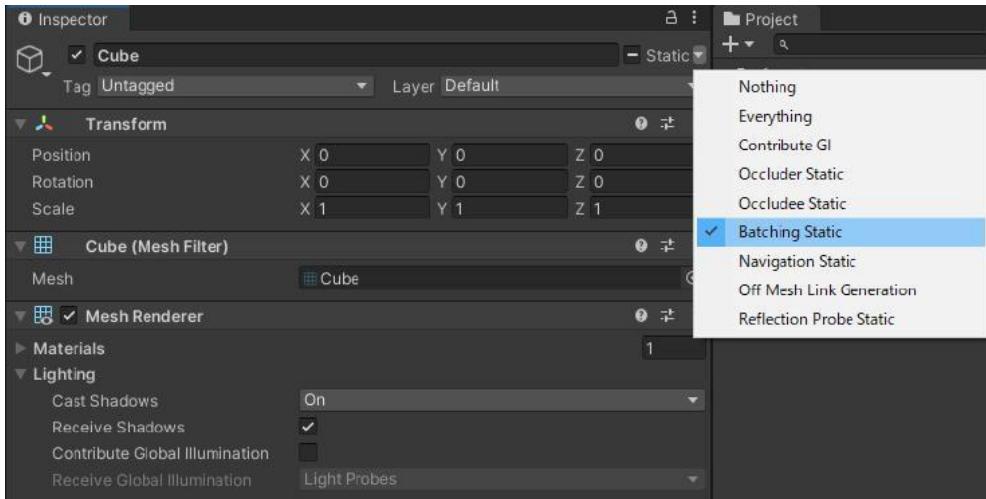


图7.6 分批静止。

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

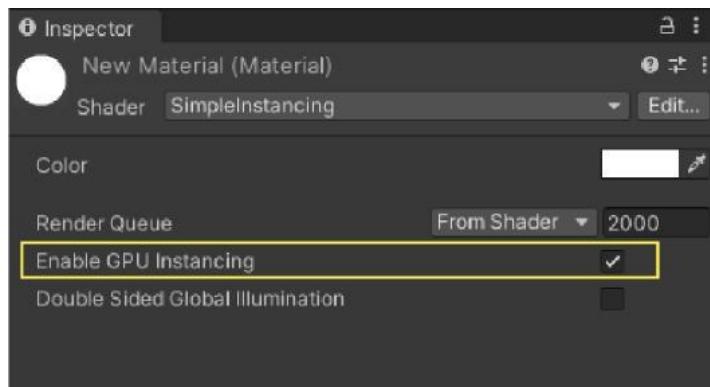


图7.7 启用实例。

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_START(Props)` 来设置，如上面的着色器代码。
`_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 减少平局次数

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。

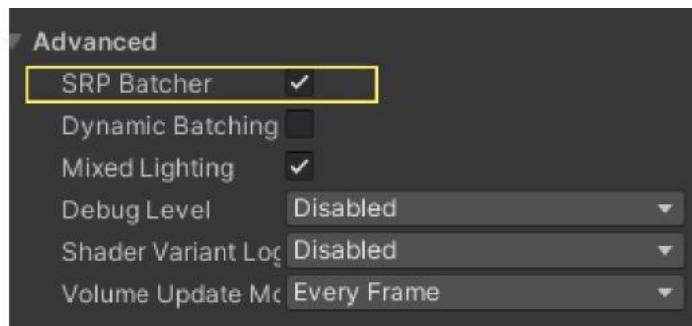


图7.8 SRP Batcher的激活。

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。

7.4 SpriteAtlas。

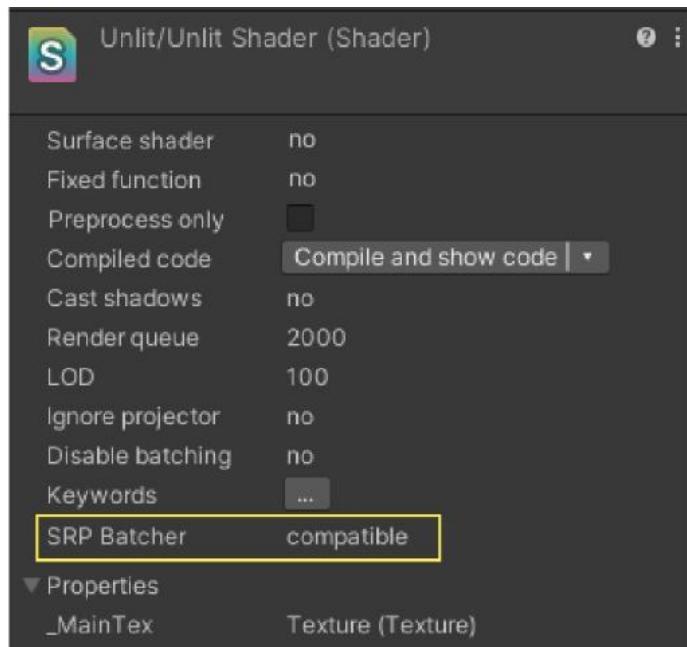


图7.9 支持SRP Batcher的着色器。

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

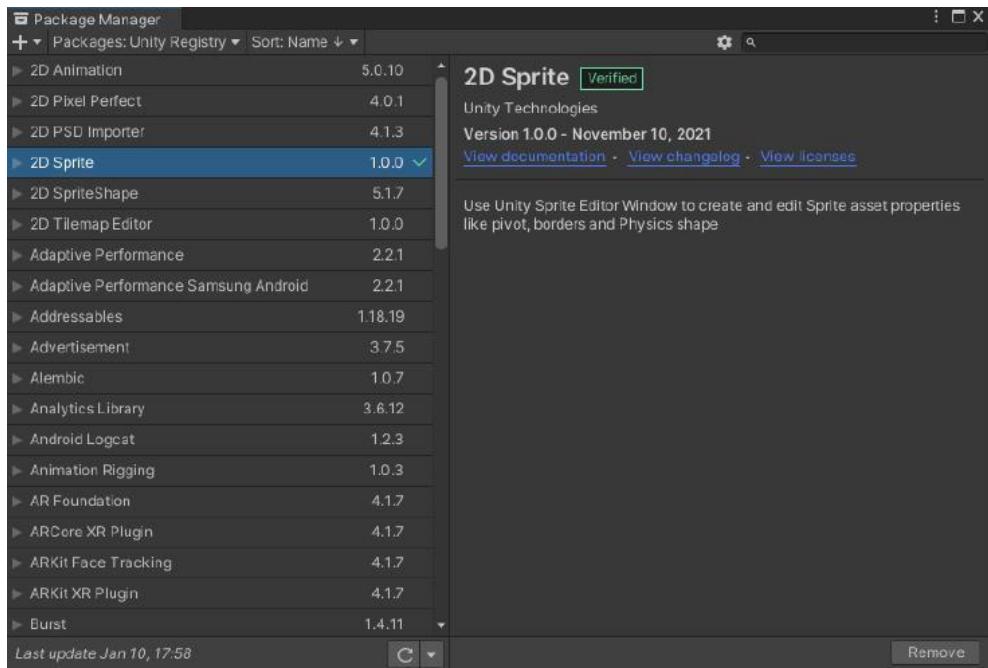
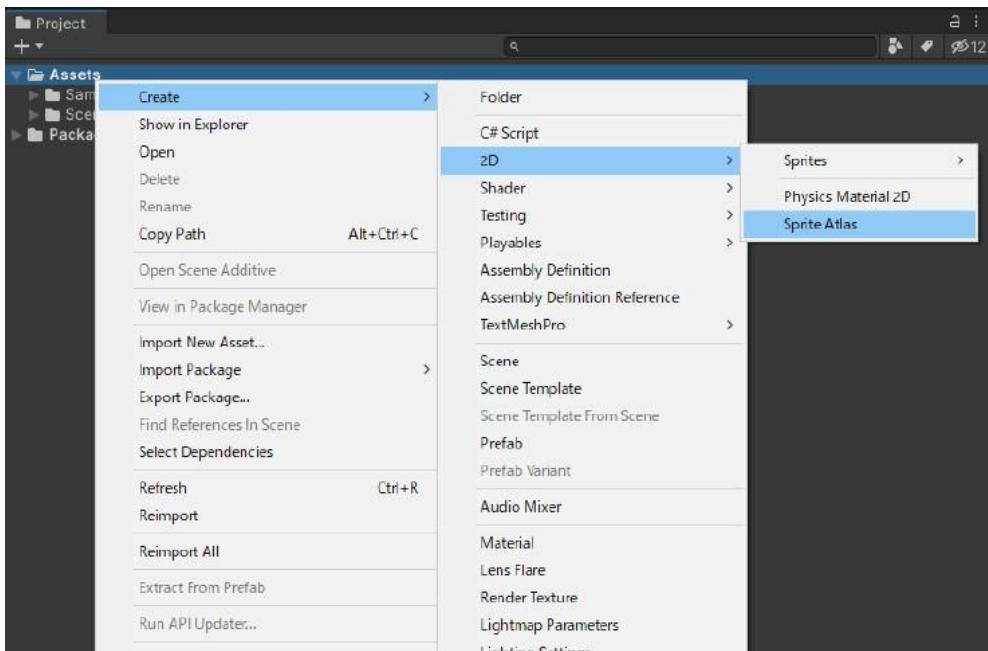


图7.10 2D Sprite。

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

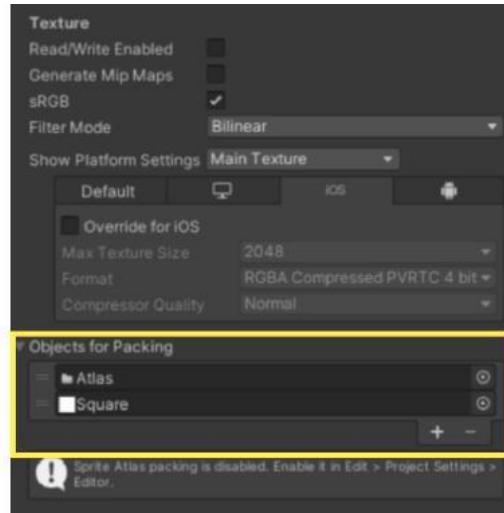
7.4 SpriteAtlas。



▲ 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，在绘制目标精灵时，将引用与SpriteAtlas集成的纹理。

也可以用下面的代码直接从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
7:     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 光杯剔除法

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，多边形的背面会被剔除，但可以通过在SubShader中指定来切换剔除设置，具体方法如下：a. 在SubShader中，输入以下内容。

▼ 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:      {
10:         CGPROGRAM
11:         #pragma vertex vert
12:         #pragma fragment frag
13:         ENDCG
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

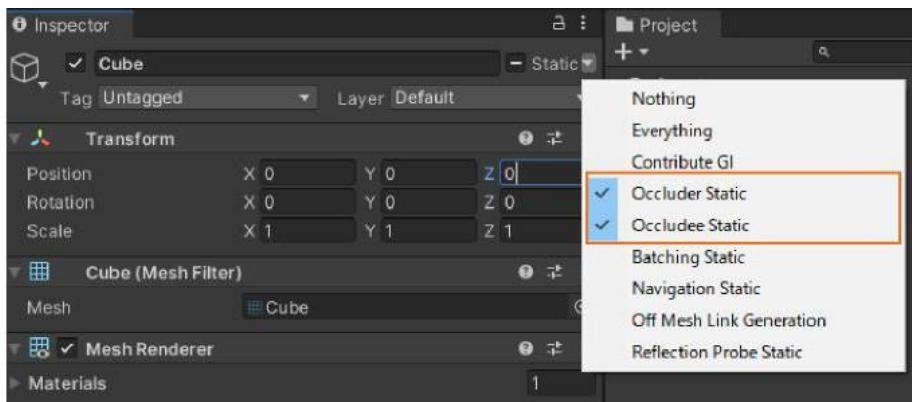
7.5.3 咬合剔除

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的一面，而只被处理为隐身的一面。

7.5 剔除



▲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

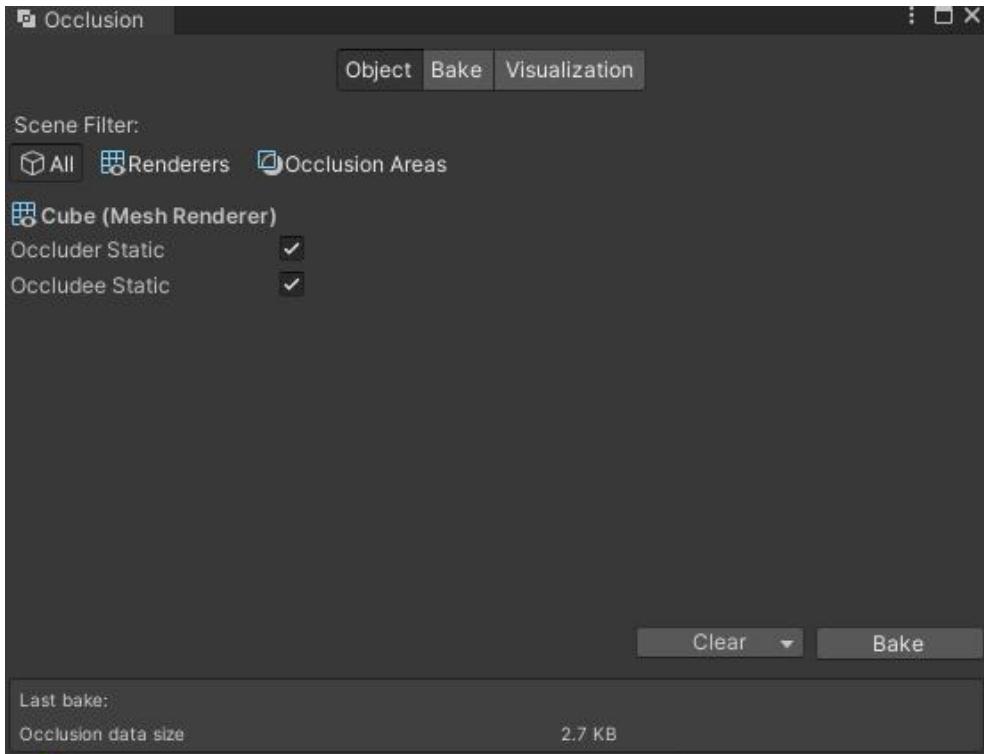


图7.14 遮挡剔除窗口

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要比对较大的数据类型的计算快。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，比如在深度计算中，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。

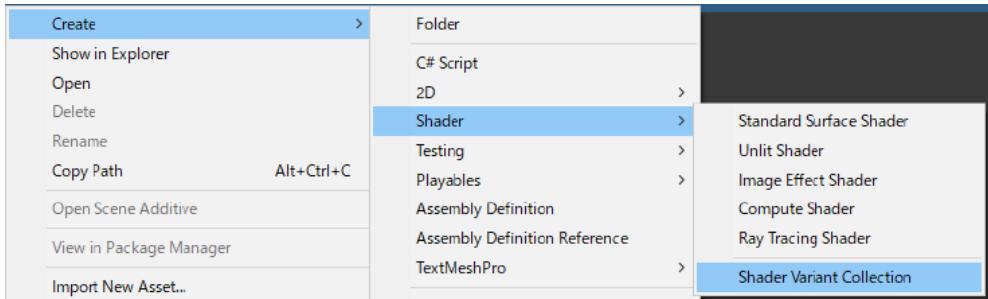


图7.16 创建一个ShaderVariantCollection。

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。

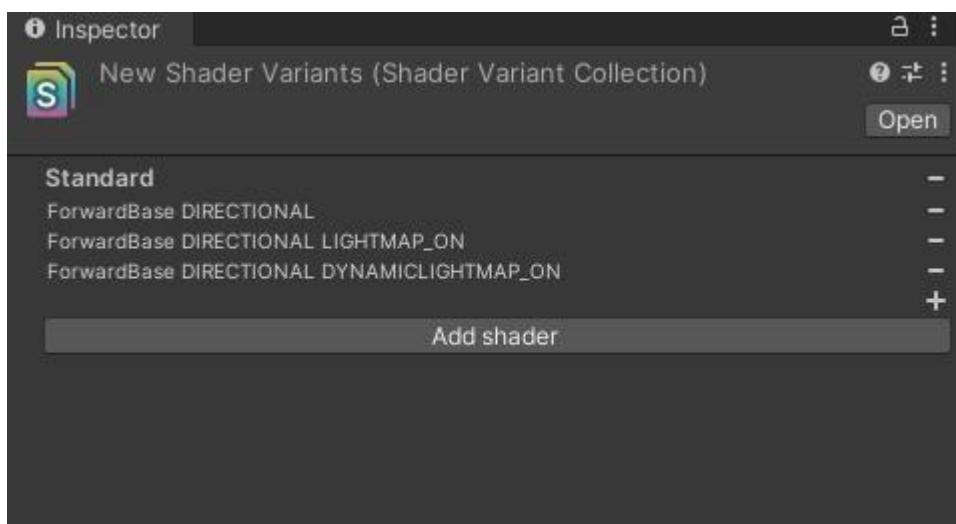


图7.17 ShaderVariantCollection的检查器。

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。

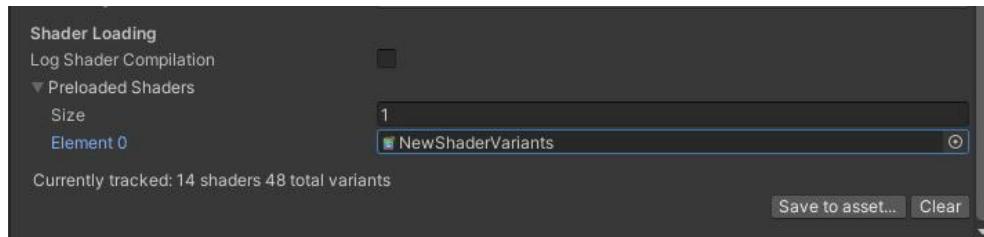


图7.18 预装的着色器。

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。

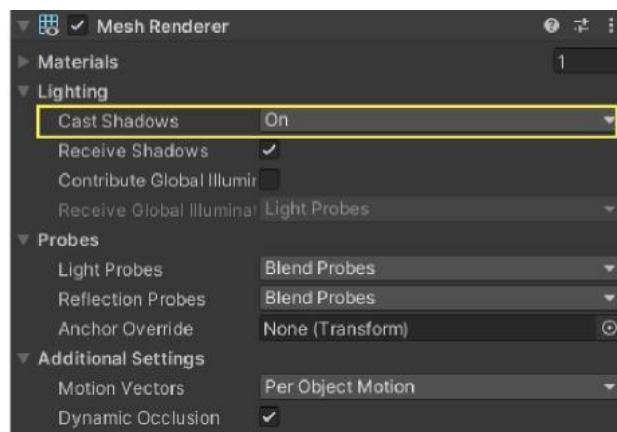


图7.19 投射阴影。

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



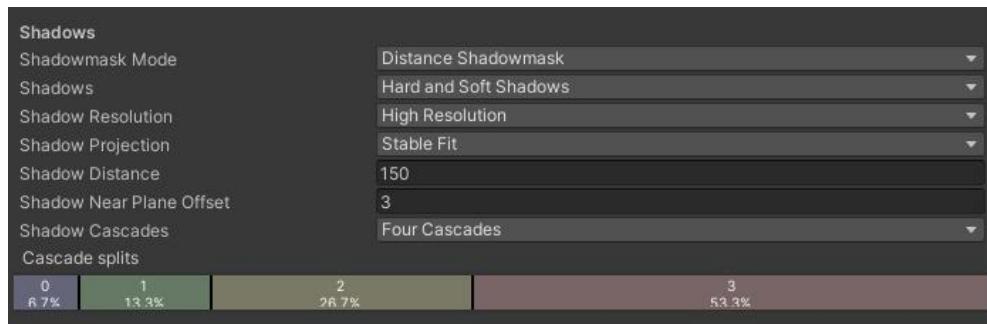
图7.20 阴影距离。

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。

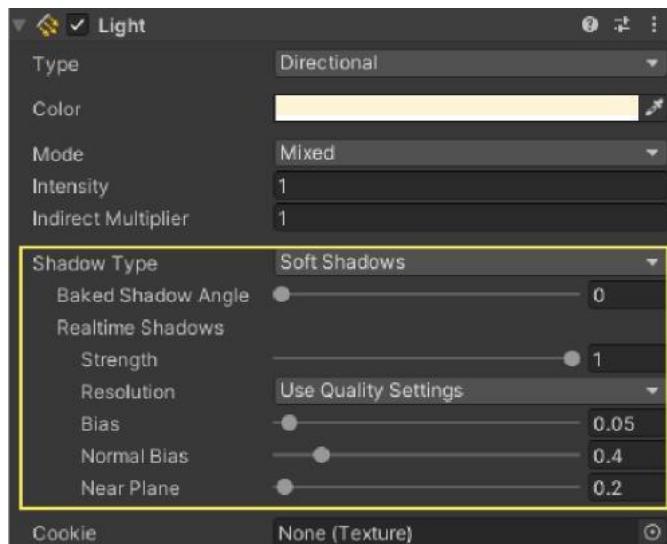


图7.22 光线组件的阴影设置。

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。

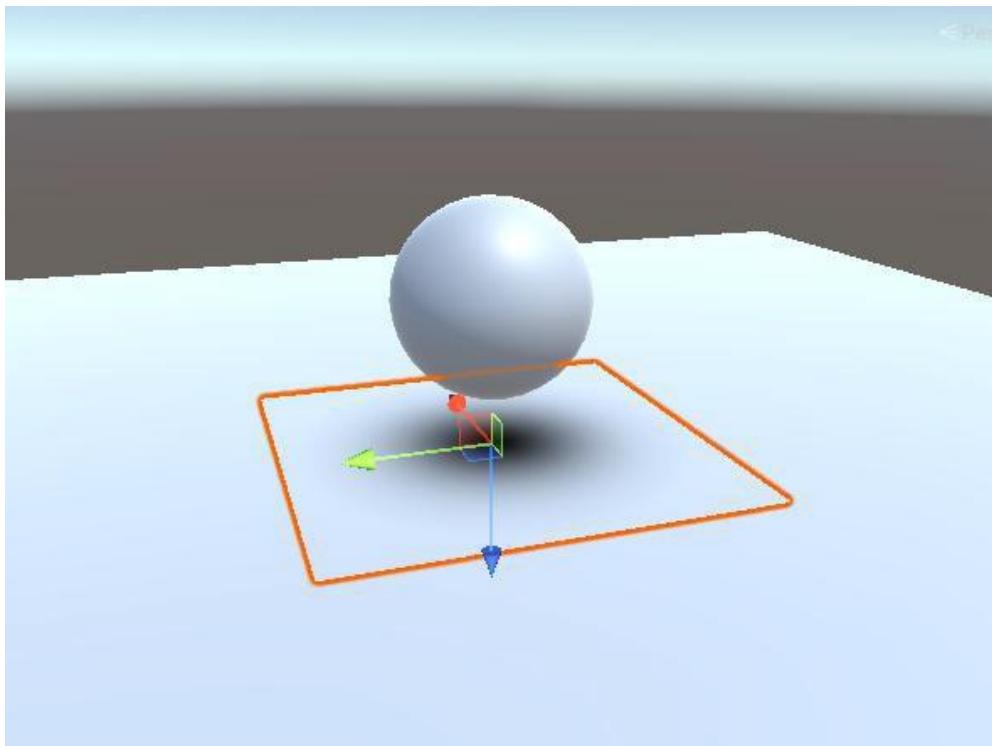


图7.23 带板块多边形的伪阴影

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。

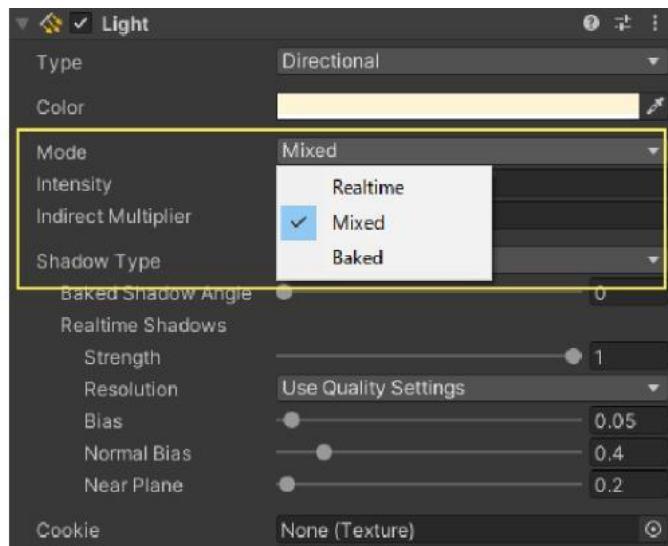
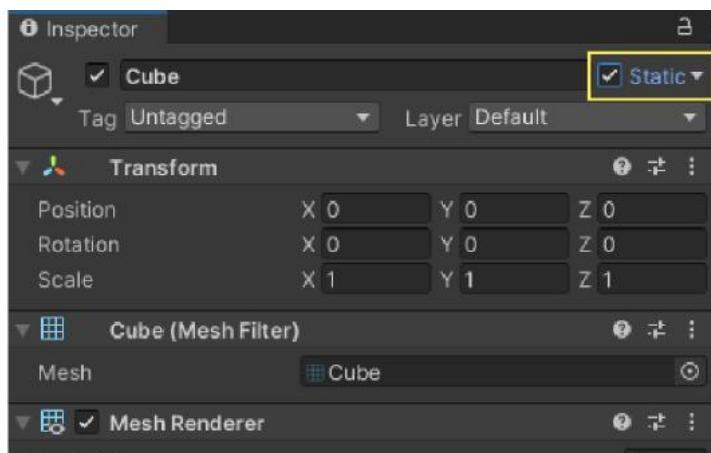


图7.24 照明的模式设置。

它还能使对象的静态标志得到烘烤。



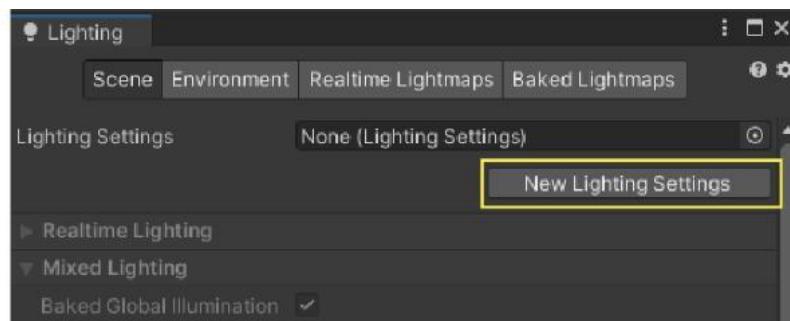
▲ 图7.25 启用静态

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选项卡中进行。

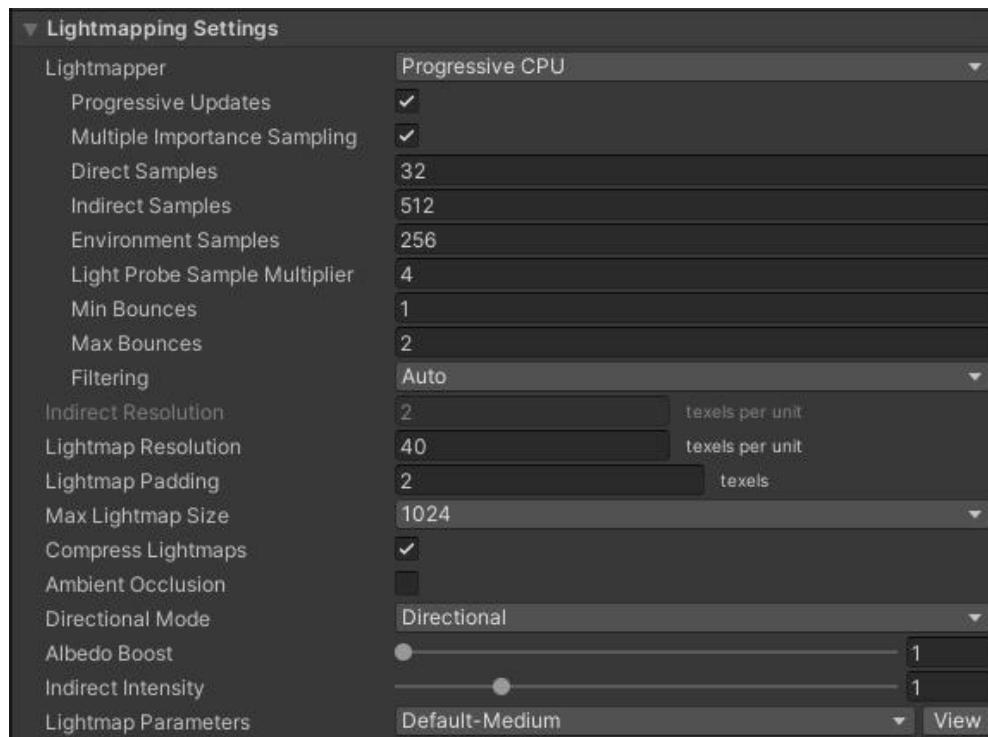


图7.27 光线映射设置

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。

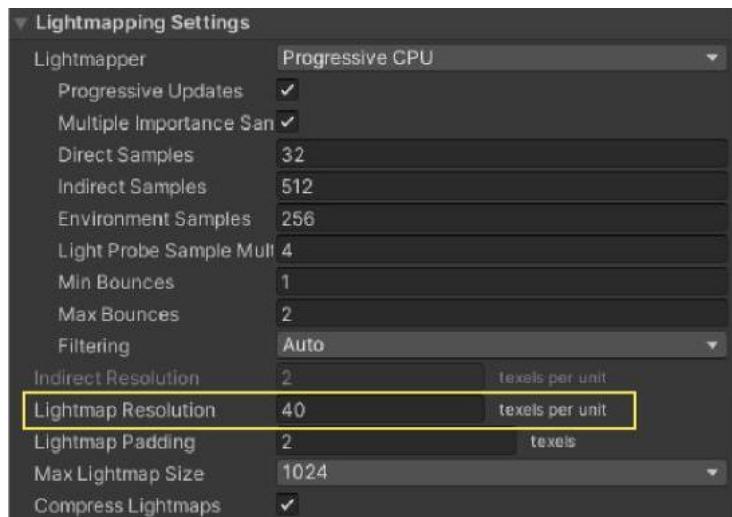


图7.28 光照图分辨率

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影图。烘烤完成后，你会看到烘烤后的光影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

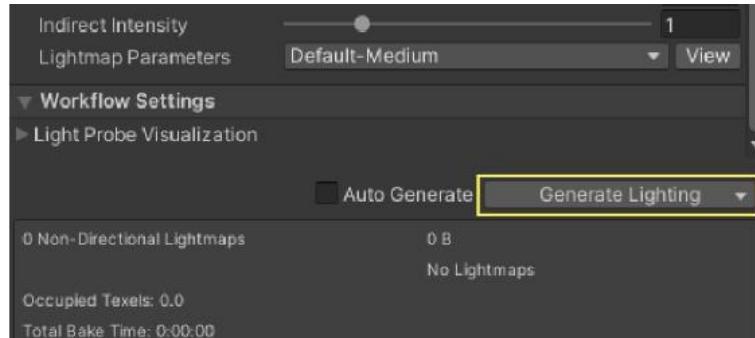
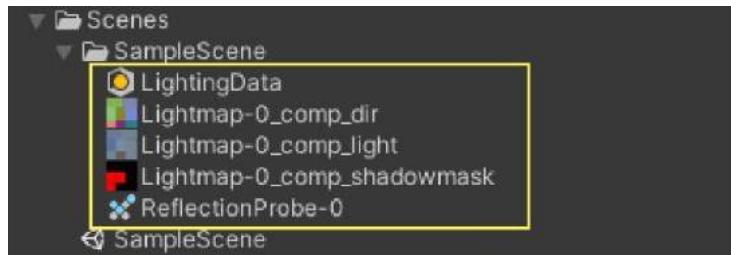


图7.29 生成照明



▲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可以通过向一个物体添加**LOD组**组件来控制。

7.9 纹理流

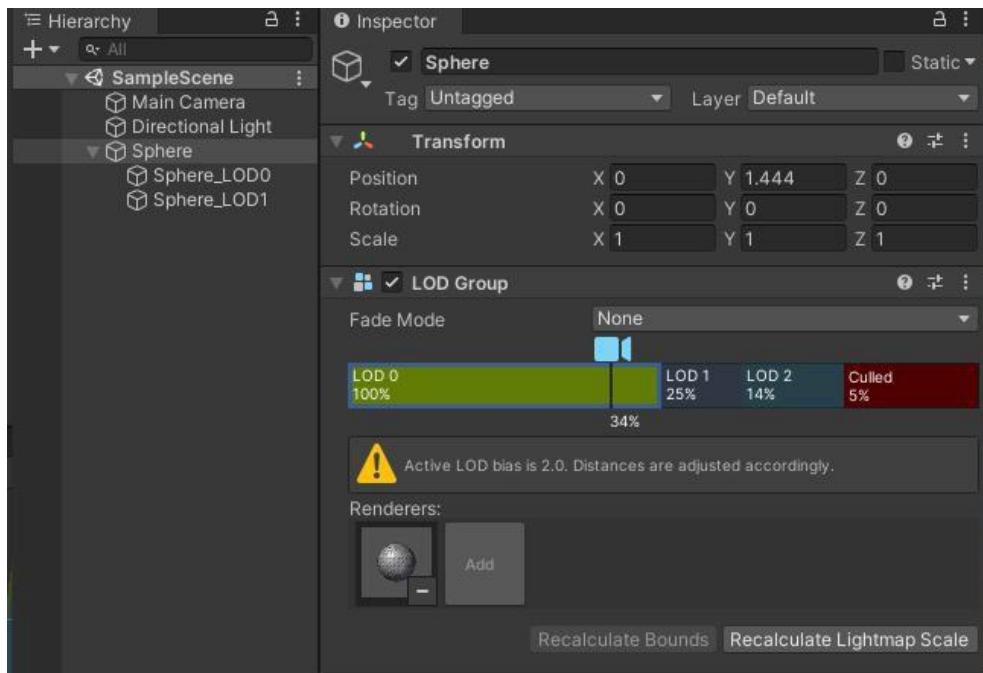


图7.31 LOD组。

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

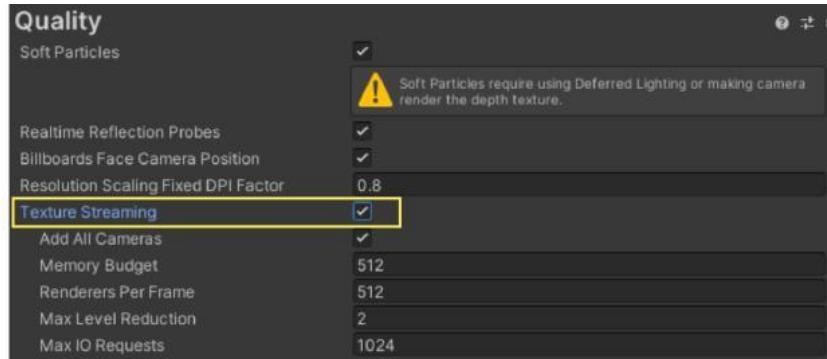


图7.32 纹理流。

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

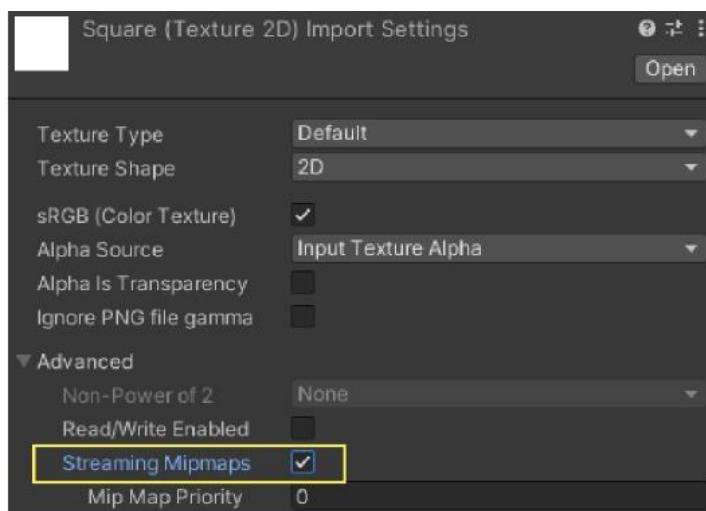


图7.33 流动的Mipmaps。

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。



性能调谐圣经

CHAPTER

08

第8章

Tuning Practice

— UI —

CyberAgent Smartphone Games & Entertainment

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。

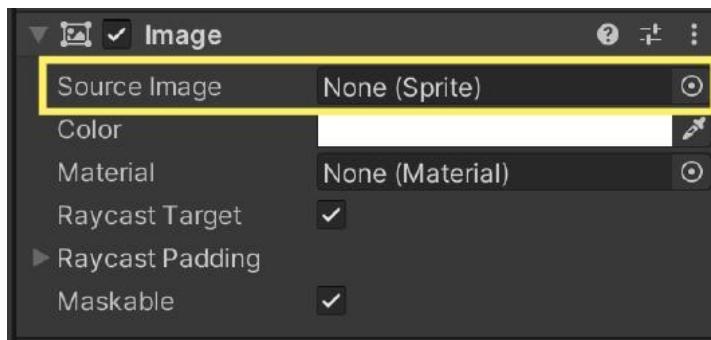
相比之下，通过将Canvas划分为UI的各个部分，就有可能创建一个可重建的UI。这可以减少动画重建的成本。例如，如果你有会移动的动画用户界面和不会移动任何东西的用户界面，你可以把它们放在不同的Canvas下，以尽量减少必须进行的动画重建的数量。

然而，需要仔细考虑如何分割画布，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和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）。

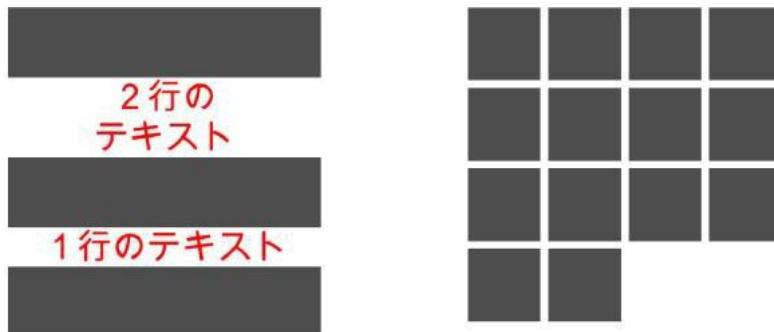


图8.3 左边是VerticalLayoutGroup，右边是GridLayoutGroup的例子。

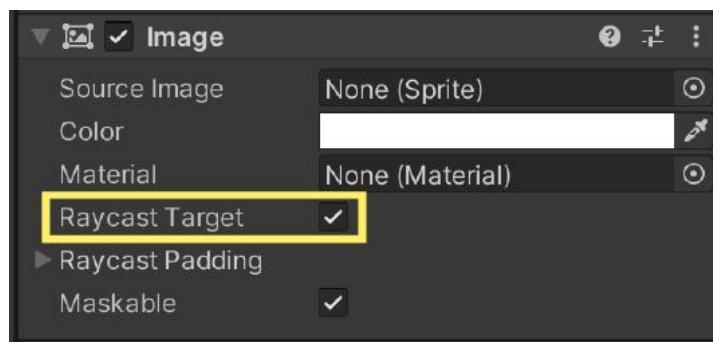
对于Layout组件，当目标对象被创建或某些属性被编辑时，Layout的重建会发生，Layout的重建是一个昂贵的过程，网格的重建也是如此。

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

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，每一帧都会出现剔除的CPU负载，与遮罩目标数量的增加成正比，这种现象，即使UI没有任何移动，每一帧都会出现CPU负载，似乎是uGUI内部实现的一个修复的副作用，从Unity 2019.3的评论中可以看出。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

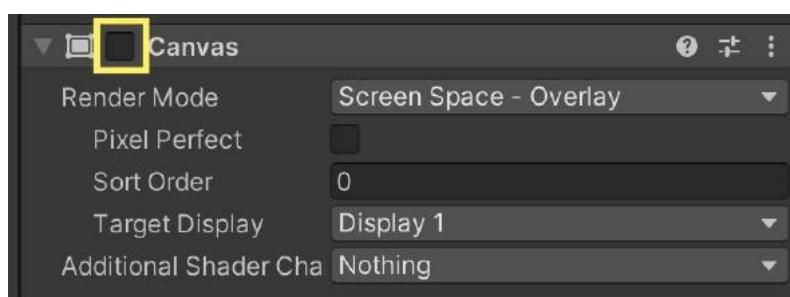
在使用文本属性的方法中，如清单8.2所示，执行了一个浮动的ToString()，每次执行这个过程都会产生一个字符串的代价。相比之下，使用SetText的方法旨在生成尽可能少的字符串，这在性能方面是有利的，特别是在要显示的文本经常变化的情况下。

这个TextMeshPro的功能在与ZString^{*3}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8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 设置CanvasGroup的透明度为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。



性能调谐圣经

CHAPTER

09

第9章

Tuning Practice
— Script (Unity) —

CyberAgent Smartphone Games & Entertainment

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，所以也应该像下面的示例代码那样进行缓存和访问。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，经常被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，销毁自动实例化的网格是你的责任。

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()。虽然这些是有用的功能，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

在Unity中关闭日志设置将停止堆栈跟踪，但日志仍然会被输出；在Unity中设置 UnityEngine.Debug.unityLogger.logEnabled为false将不会输出任何日志，但函数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
```

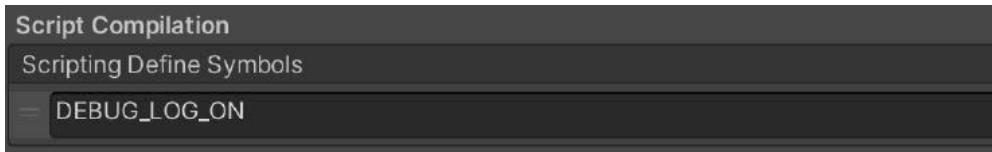
在这种情况下可以利用条件属性：对于具有条件属性的函数，如果指定的符号没有被定义，编译器将删除调用部分。如清单9.13中的示例所示，最好给自己类中的每个函数添加一个Conditional属性，作为通过自己的日志输出类在Unity端调用日志函数的规则，这样在必要时可以删除整个函数调用。

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

Burst^{*6}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 >。

它被使用；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

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.s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。



性能调谐圣经

CHAPTER

10

第10章

Tuning Practice
— Script (C#) —

CyberAgent Smartphone Games & Entertainment

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通过引用入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，什么原因可能导致装箱？

▼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，而 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，在有大量数据且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，而是被原封不动地引用时，可能就无法加快进程了。

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的池子，只有在使用时才对池子里的对象进行赋值

和引用。

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。上一节中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预生成字符串对象并使用。

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 $T[] \rightarrow \text{List}\langle T \rangle \rightarrow \text{IEnumerable}\langle T \rangle$ 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 $\text{IEnumerable}\langle T \rangle$ 并返回 $\text{IEnumerable}\langle T \rangle$ ，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 $\text{Enumerable}\langle T \rangle$ 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处理的时间要长19倍。

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的结果是性能更差。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-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;
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:             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

```
1 : 使用System.Agency.Ltd;
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关键字的方法在编译时添加一个进程，该进程生成一个实现IAsyncStateMachine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，可能需要使用不使用堆分配的方法。

*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，并以在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。

需要注意的是。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:     {
5:         // 循环次数的数组累积。
6:         byte* buffer = stackalloc byte[10000];
7:     }
8: }
```

10.8 在IL2CPP后端下通过密封的方法调用进行优化

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: {  
7:     public override string Speak() {  
8:         返回 "Moo".  
9:     }  
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。不是的。

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的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，MethodImpl属性可以为一个方法和参数指定MethodOptions.AggressiveInlining到生成的C++代码。内联指定器现在被赋予了相应的函数的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，它们都会作为值传递被复制并传递给方法，所以如果参数的数量或传递的结构大小很大，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，通过内联进行优化是非常有用的。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/Documentation/MethodOptions.md>

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



Unity®

性能调谐圣经

CHAPTER

11

第11章

Tuning Practice
— Player Settings —

CyberAgent Smartphone Games & Entertainment

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。

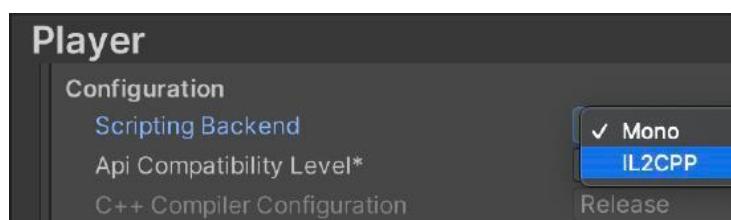
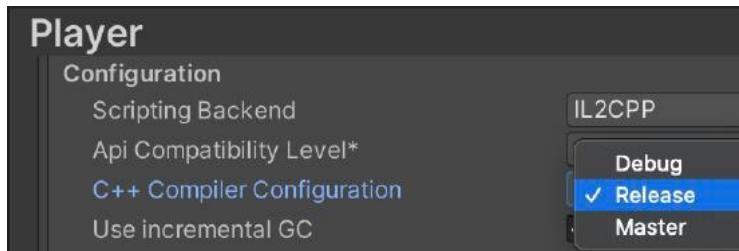


图11.1 脚本后端配置。

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。

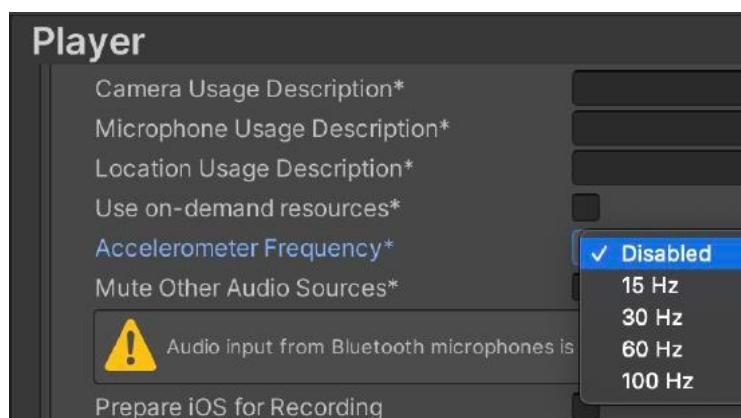


图11.3 采样频率设置。

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



Unity®

性能调谐圣经

CHAPTER

12

第12章

Tuning Practice
– Third Party –

CyberAgent Smartphone Games & Entertainment

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:     }

```

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。

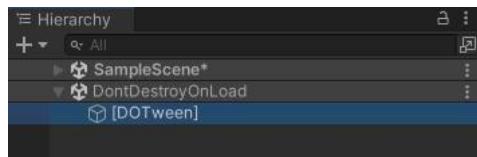


图12.1 [DOTween] GameObject。

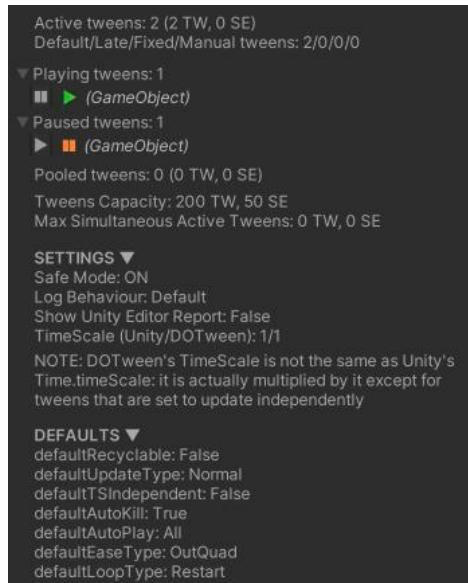


图12.2 DOTween检查器。

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、处理消息的回调等对象的实例被创建。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:                 // 每一帧要执行的程序。
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。它还可以根据Unity的PlayerLoop来控制执行时间，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(...).)和其他由工厂返回的任务在调用时被调用，而在从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。

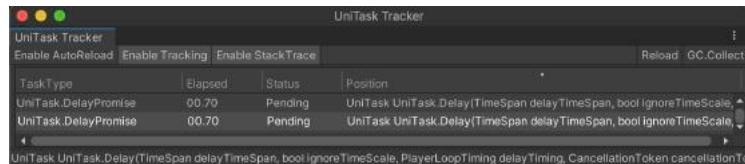


图12.3 UniTask Tracker。

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。

防止任务泄漏

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Destroy时间，前一个UniTask顺利完成，但后一个却没有Operation抛出 CanceledException。哪种行为更可取，取决于情况，应选择适当的实施方式。

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公司工程经理，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。

Mitsutoshi Nakamura (NAKAMURO.) / Twitter: @megal_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 Hottojisus 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。

附录作者。

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。目前正在对新的3D相关技术进行验证工作。

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



性能调谐圣经

CyberAgent 智能手机游戏和娱乐

CyberAgent.