



Unity®。

性能调谐圣经

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パイプ

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



Unity®

0

性能调谐圣经

撰稿人

CyberAgent智能手机游戏和Entertainment部门

SGE公司CTO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・エンターテインメント事業部 SGEコア技術本部 著

简介

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。我们希望它能对那些看过的人有一些帮助。

对于本出版物：

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 组成

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，最好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 部分。

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任何责任。

^{*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
1.9.1	1CPU绑定	20
1.9.2	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	记忆	33
2.1.6	储存	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	位和字节	46
2.3.2	图片	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	课题	58

2.4.4	游戏循环.....	60
2.4.5	游戏对象.....	62
2.4.6	资产捆绑.....	64
2.5	C#基础知识.....	67
目录		
2.5.1	堆栈和堆.....	68
2.5.2	Garbage Collection.....	68
2.5.3	结构 (struct).....	70
2.6	算法和计算复杂性.....	73
2.6.1	关于计算量.....	74
2.6.2	基本集合和数据结构.....	77
2.6.3	降低计算复杂性的设备.....	80
第三章：剖析工具		81
3.0.1	关于测量的说明.....	82
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

3.7.2	拨款。	146
3.8	安卓工作室	149
3.8.1	简介方法	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	如何查看捕获数据	158
第四章。	调试实践--资产	167
4.1	纹理	168
4.1.1	进口设置	168
4.1.2	读/写	169
4.1.3	生成Mip地图	170
4.1.4	阿尼索水平	170
4.1.5	压缩设置	171
4.2	网状物	172
4.2.1	启用了读/写功能	172
4.2.2	顶点压缩	173
4.2.3	网眼压缩	174
4.2.4	优化网格数据	175
4.3	材料。	176
4.4	动画	177
4.4.1	调整皮肤重量的数量	177
4.4.2	关键减少	178
4.4.3	减少更新的频率	180
4.5	粒子系统	181
4.5.1	减少颗粒的数量。	181
4.5.2	注意, 噪音很重。	184
4.6	音频	185
4.6.1	负载类型	185

4.6.2	压缩格式	187
4.6.3	指定采样率。	188
4.6.4	对于声音效果，设置为单声道。	188
4.7	资源 / 流媒体资产	189

目录

4.7.1	减慢启动时间 资源文件夹	190
4.8	ScriptableObject	190
第五章	调试实践--资产捆绑	193
5.1	资产包的颗粒度	194
5.2	AssetBundle加载API。	194
5.3	资产包的卸载策略。	195
5.4	优化同时加载的AssetBundles的数量。	195
第六章	调音实践--物理学	197
6.1	物理学开/关	198
6.2	优化固定时间步长和固定更新的频率	198
6.2.1	允许的 最大时间步长	199
6.3	碰撞几何的选择	200
6.4	碰撞矩阵和层优化	200
6.5	射影优化	201
6.5.1	射线广播的类型	201
6.5.2	优化射线传输参数	202
6.5.3	RaycastAll和RaycastNonAlloc	202
6.6	对撞机和刚体.	203
6.6.1	刚体和睡眠状态	204
6.7	优化碰撞检测	207
6.8	优化其他项目设置	208
6.8.1	Physics.autoSyncTransforms	208
6.8.2	Physics.reuseCollisionCallbacks	208
第七章	调音实践--图形	209
7.1	调整分辨率	210
7.1.1	DPI设置	210
7.1.2	脚本化的分辨率设置	211
7.2	半透明性和过度拉伸.	212

7.3	减少平局调用.	213
7.3.1	动态配料	213
7.3.2	静态配料	215
7.3.	3GPU实例化	216
7.3	.4SRP Batchter	219

7.4	SpriteAtlas	221
7.5	删减.....	225
7.5.1	视盘摘除术	225
7.5.2	后方剔除.....	225
7.5.3	遮盖物的剔除	226
7.6	着色器.....	228
7.6.1	降低浮点类型的精度	229
7.6.2	使用顶点着色器进行计算.....	229
7.6.3	预先填充纹理的信息	230
7.6.	4ShaderVariantCollection	230
7.7	写作.....	232
7.7.1	实时的阴影.....	232
7.7.2	光线映射.....	236
7.8	详细程度.....	240
7.9	纹理流.....	241
第8章	调试实践--用户界面	243
8.1	分割画布	244
8.2	UnityWhite	245
8.3	布局组件	246
8.4	雷击目标.....	247
8.5	面具	247
8.6	TextMeshPro	248
8.7	切换用户界面显示	249
第9章	调音实践--脚本（统一）。	251
9.1	空的Unity事件功能。	252
9.2	访问标签和名称	253
9.3	检索组件	254
9.4	访问转化.....	254
9.5	需要明确销毁的类.....	255
9.6	字符串规格	255
9.7	JsonUtility的陷阱。	256
9.8	渲染和MeshFilter的陷阱	257

9.9	删除日志输出代码	258
-----	----------------	-----

目录

9.10	用Burst加快代码的速度。	259
9.10.1	用Burst加快代码速度	260
第10章	调音实践--脚本(C#)	263
10.1	GC.分配案件和如何处理这些案件。	264
10.1.1	参考类型新	264
10.1.2	Lambda表达式	265
10.1.3	使用泛型来装箱的案例.	267
10.2	关于for/foreach	269
10.3	对象池	272
10.4	弦。	273
10.5	LINQ和延迟评估。	275
10.5.1	通过使用LINQ来缓解GC.Alloc的问题。	275
10.5.2	LINQ延迟评估	276
10.5.3	选项 "避免使用LINQ".	277
10.6	避免了async/await的开销。	279
10.6.1	在不需要时避免使用async	279
10.6.2	避免同步的上下文捕获	280
10.7	通过stackalloc进行优化。	281
10.8	通过密封优化IL2CPP后端下的方法调用	282
10.9	通过内联进行优化。	285
第11章	调音练习 - 播放器设置	287
11.1	脚本后端.	288
11.1.1	调试	289
11.1.2	发布	289
11.1.3	主站	289
11.2	剥离发动机代码/管理剥离水平	289
11.3	加速器频率 (iOS)	290
第12章	调试实践 - 第三方	291
12.1	DOTween	292

12.1.1	SetAutoKill	292
12.1.2	SetLink	293
12.1.3	DOTween检查器	294
12.2	统一制药	295

12.2.1 取消订阅	295
12.3 统一任务	296
12.3.1 UniTask v2	296
12.3.2 统一任务跟踪器	296
到头来	299
关于作者	301



统一

性能调整

CHAPTER

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 提前半 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，而不可能忽视拥有低规格设备的用户。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。

这些指标从初始阶段就很难确定，因为当时没有足够的功能实现或资产来衡量负载。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荷模型的方法。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标？
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。考虑这些优点和缺点，并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，志愿者的人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。

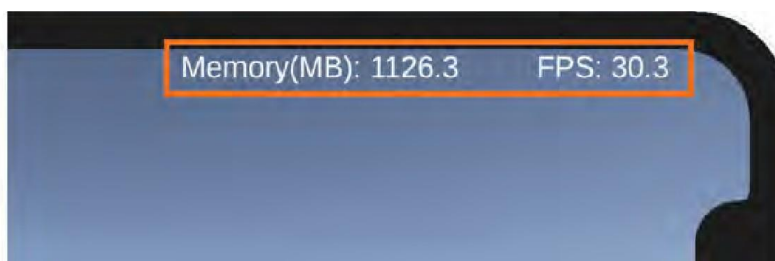
然而，这将降低外观和感觉的质量，所以要与导演协商，共同探讨什么线路是项目可以接受的。

1.2 防患于未然。 gu

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

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 在性能调整方面的工作。

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)



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，在本文中也没有详细介绍。

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 导致内存切断过多的原因 KE

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某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，就调查泄漏的原因。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。具体方法在1.6节“让我们减少内存”中解释。

1.5 调查内存泄漏。

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。

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项目，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。

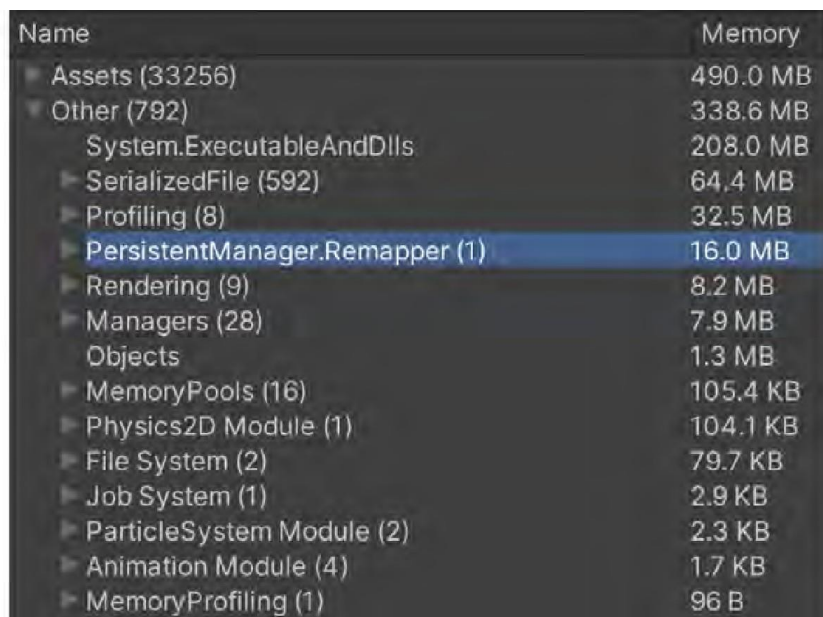
参见 “2.1.5 内存”以了解更多关于管理堆的信息。同样地, GC.Alloc。关于 “堆栈和堆”的细节将在第2.5.1节 “堆栈和堆”中论述。

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。

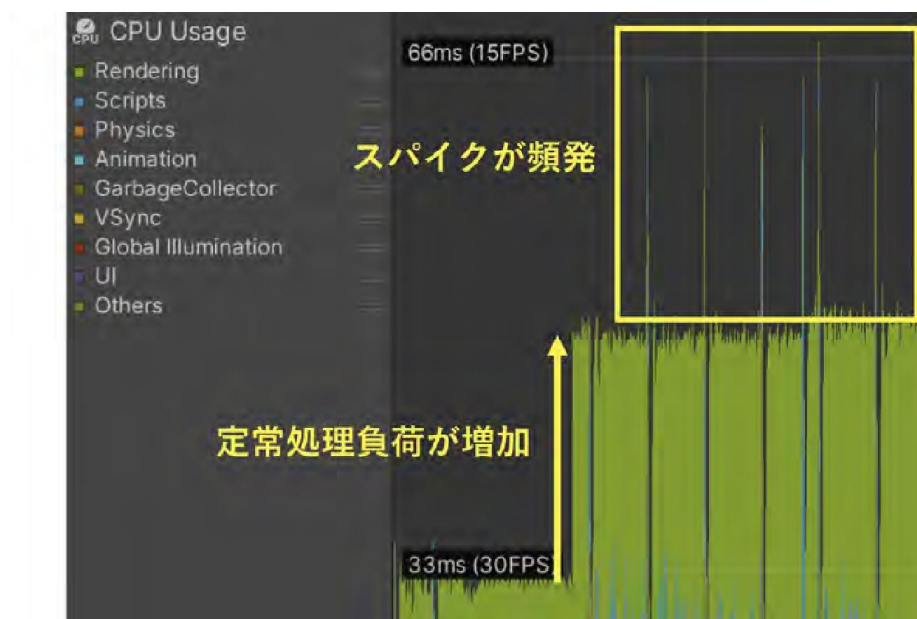
- 改变纹理的压缩率
 - 只对纹理的一个部分增加一个压缩率。
- 改变装货和卸货的时间。
 - 释放常驻内存中的对象，每次加载它们。
- 改变道路规格
 - 在游戏中少了一个需要加载的角色变体。

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“瞬时”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于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，你很可能被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，Frame Debugger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。

逐一查看负载。

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。即使发生了本文件中没有描述的实例，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，我们希望你能够将本章的信息用于实践。



统一

性能调整

CHAPTER

0

第二章。

基础知识

在进行性能调整时，需要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。

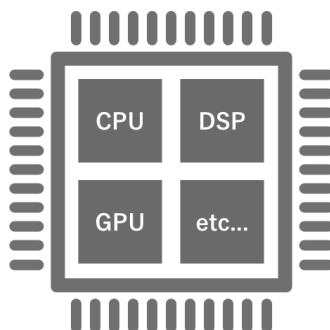


图2.1 SoC

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很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	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，所以本节从性能的角度来解释。

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。

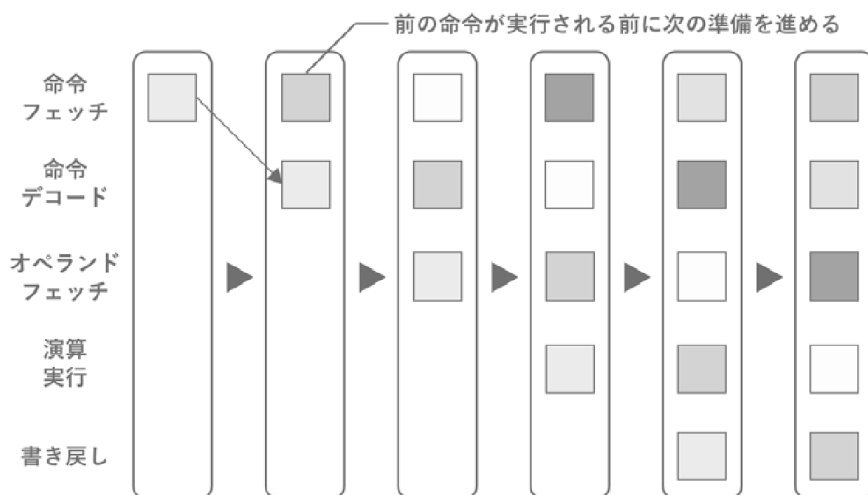
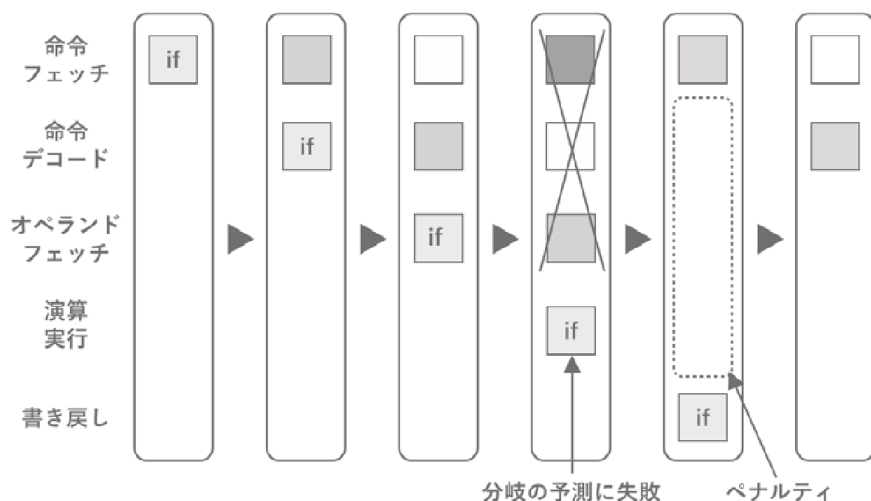


图2.2 CPU流水线结构

在CPU内部处理的指令流被称为流水线，下一条指令在流水线中被预测，因为它被处理。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

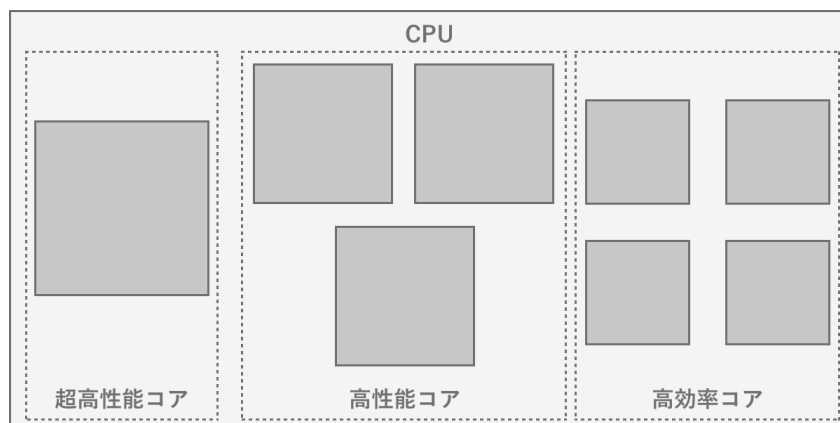
CPU计算能力

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（尤其是智能手机）中已成为主流。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。

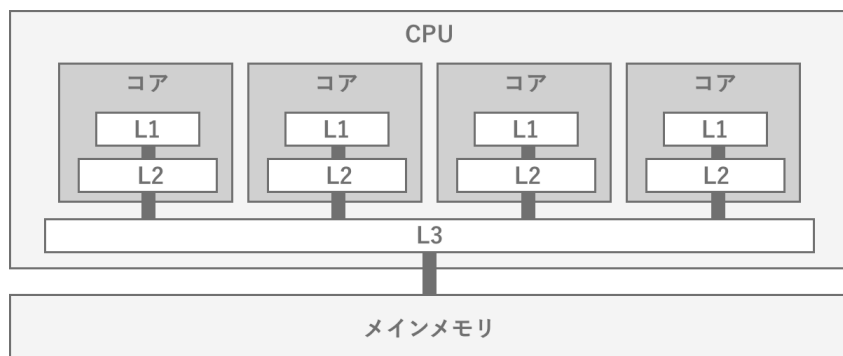


▲ 图2.4 骁龙8代1的异构核心配置

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专门用于执行程序，而GPU（图形处理单元）是专门用于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。

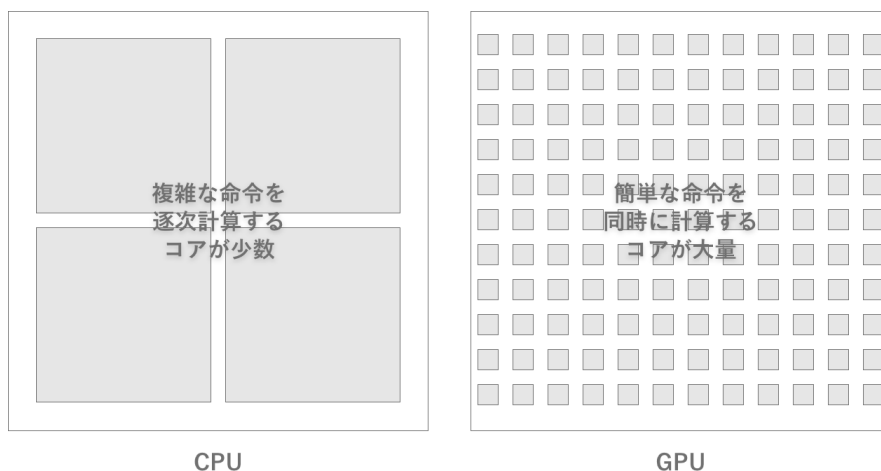


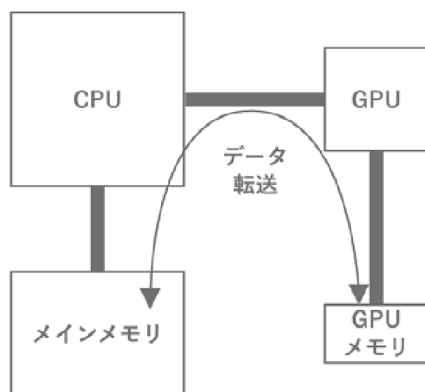
图2.6 CPU和GPU之间的差异

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可以高速对大量数据进行并行运算，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，但存储器并不包括在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。然而，如果整个系统使用的内存耗尽，则根据操作系统确定的优先级顺序杀死该进程。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，它自己就会成为被杀死的目标。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，而且对智能手机来说也不实用，因为智能手机的存储容量本来就小。

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。

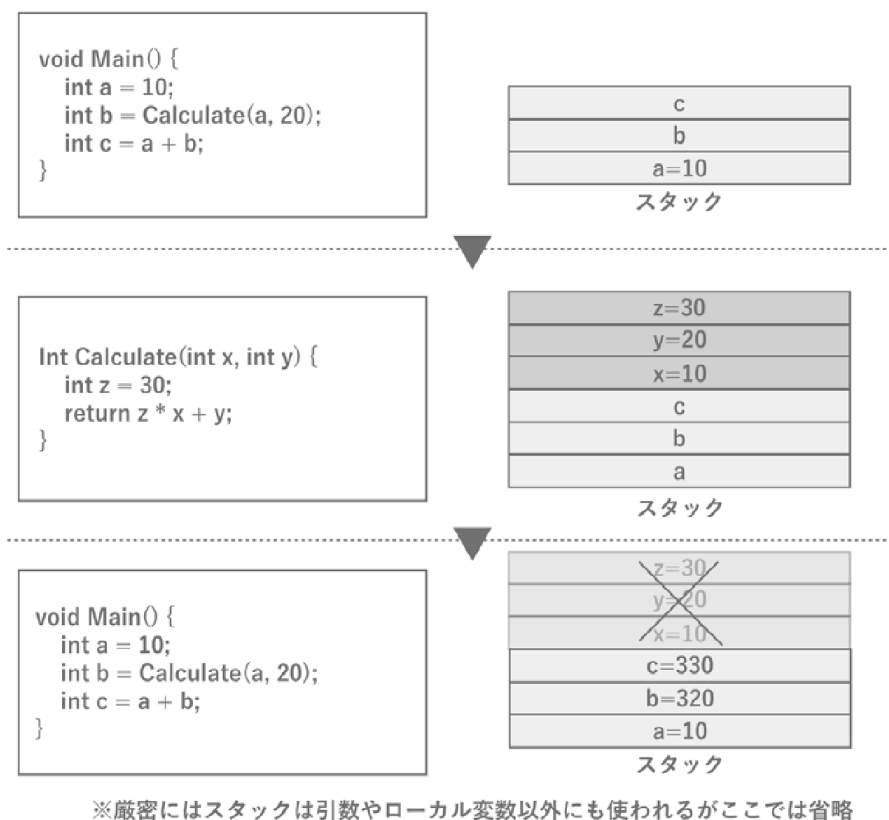


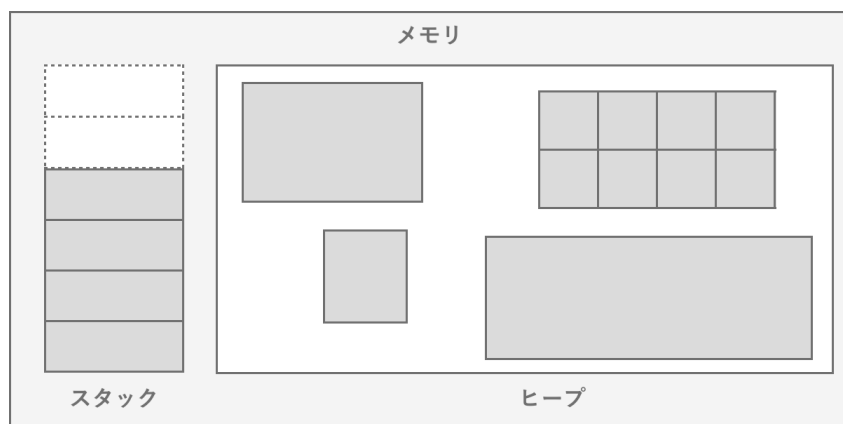
图2.8 堆栈操作示意图

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，如果重复进行内存分配和释

放，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。

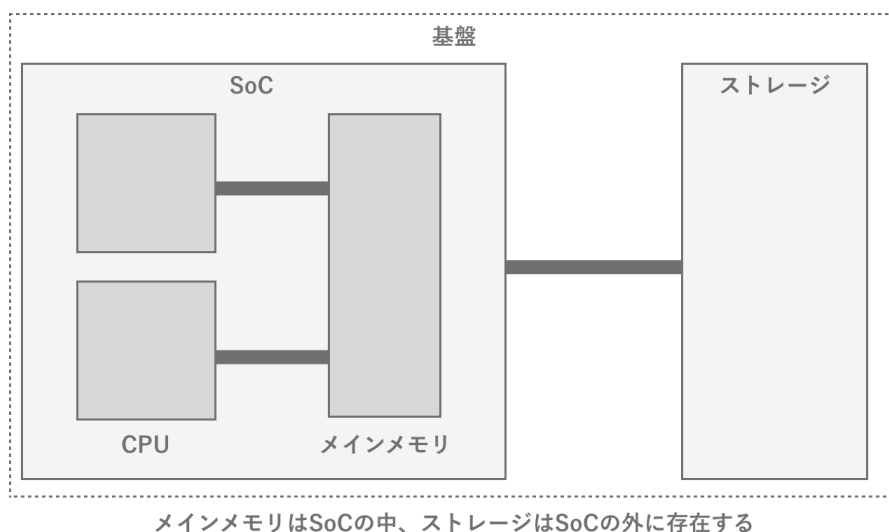
堆栈溢出错误。

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

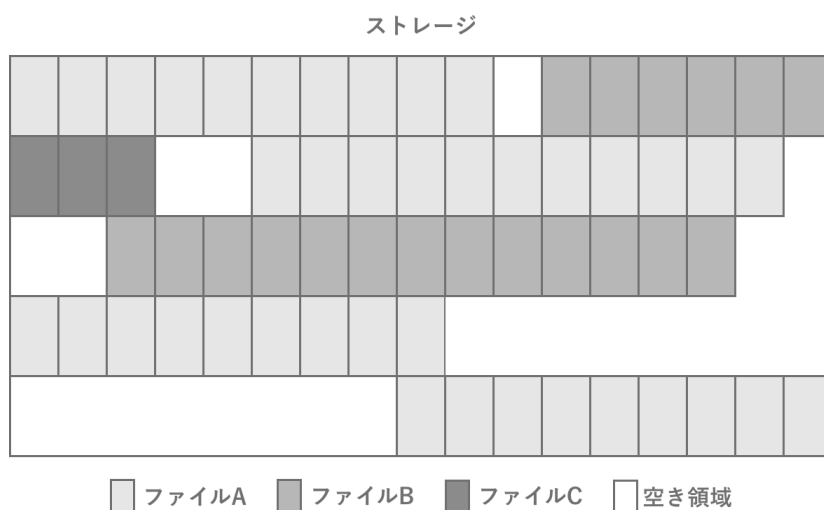
当读出存储的文件时，细节被省略了，但大致是该过程如下。

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. 加载数据。
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。至于个别设备的具体性能，有一些网站^{*1}收集了一些基准测试结果，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**。

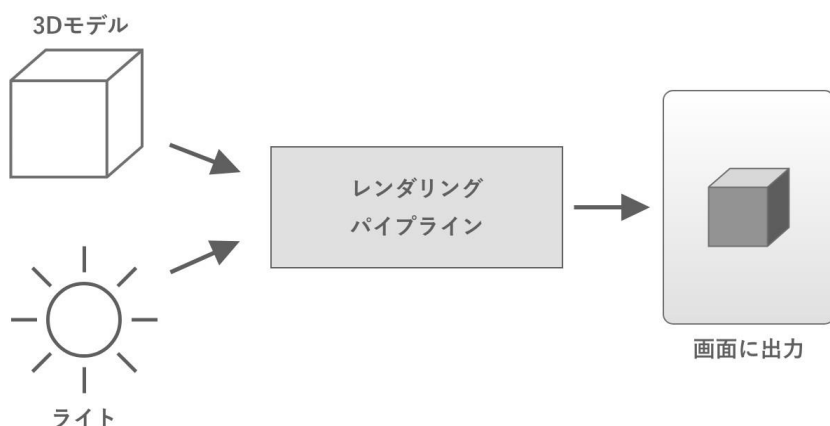
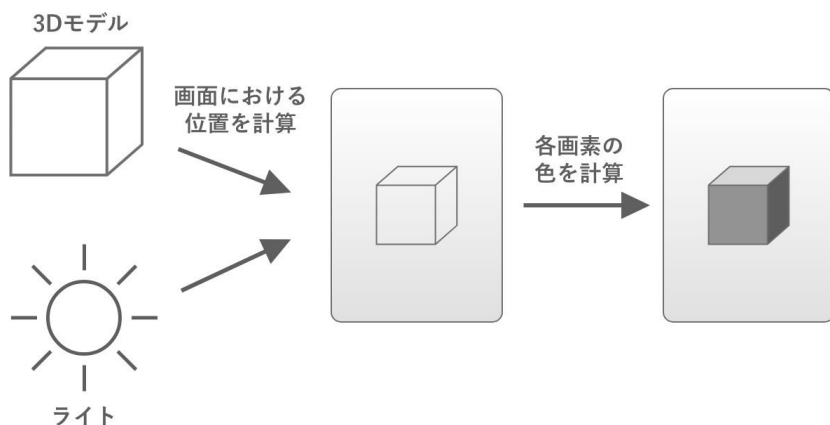


图2.12 渲染管道。

渲染管道开始时，将必要的信息从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之外，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，因此将仅限于简化描述。

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。

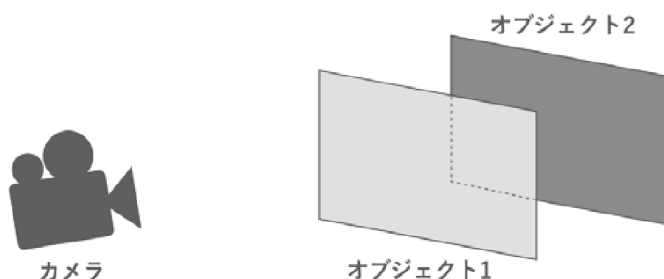


图2.14 两个重叠的物体。

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。

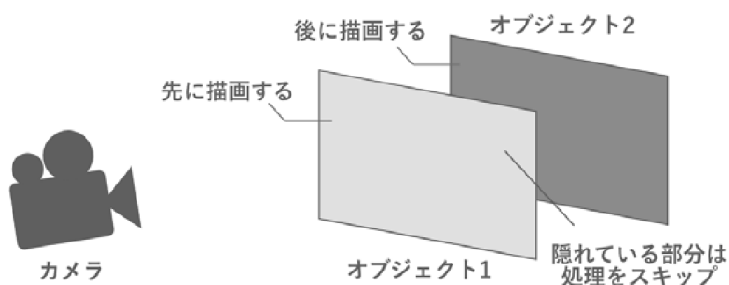


图2.15 不透明的绘图

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露出来，甚至与前景的物体重叠，那就不自然了。这个

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。

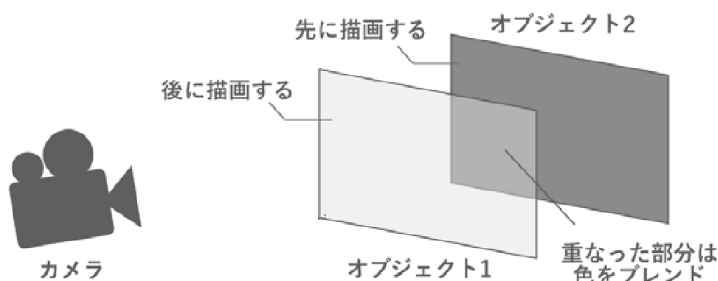


图2.16 半透明的图纸

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，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，只要着色器的变体相同即可。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过的调用，因为这些调用的处理负荷最高。

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。

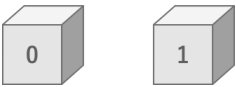
一个可以产生类似于批处理效果的功能是**GPU实例化**。这是GPU的一个特性，它允许具有相同网格的对象在一次绘图调用/设置路径调用中被绘制。

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。



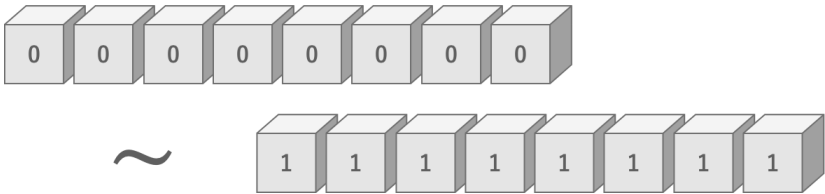
▲图2.17 一个比特的信息量

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。

千字节和小写的字节

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。

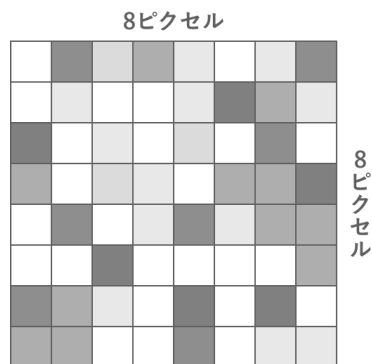
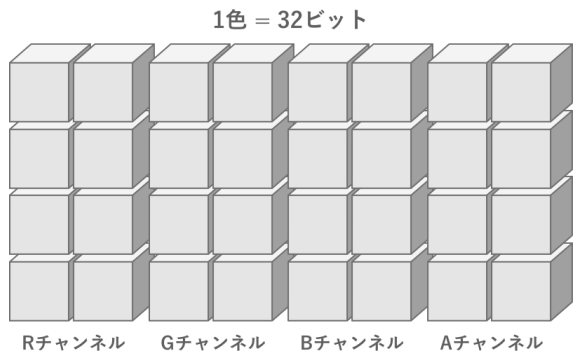


图2.20 图像数据。

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（Red）、绿（Green）、蓝（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 \times 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

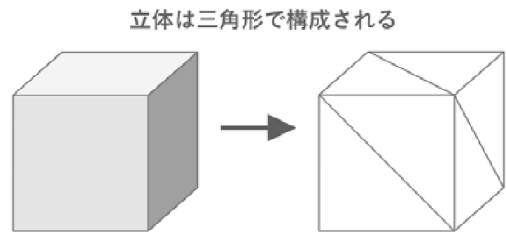
在Unity中，可以使用纹理导入设置为每个平台指定各种压缩方法。因此，通常的做法是导入未压缩的图像，并使用这些导入设置进行压缩，以生成最终要使用的纹理。

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**。



▲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。

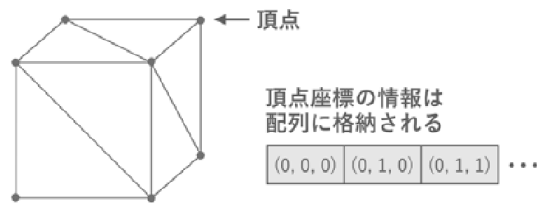


图2.24 顶点信息。

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。

頂点インデックス										
0	1	3	2	4	5	6	9	7	8	...
0, 1, 3番目の 頂点座標が 三角形1を構成			2, 4, 5番目の 頂点座標が 三角形2を構成			6, 9, 7番目の 頂点座標が 三角形3を構成			...	

▲ 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

名称。	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	四维平面=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通过插值获得的，因此可以把它们当作平滑、连续的数据。

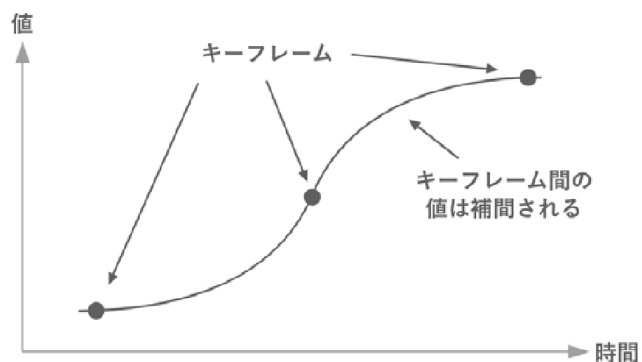


图2.26 关键框架。

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。

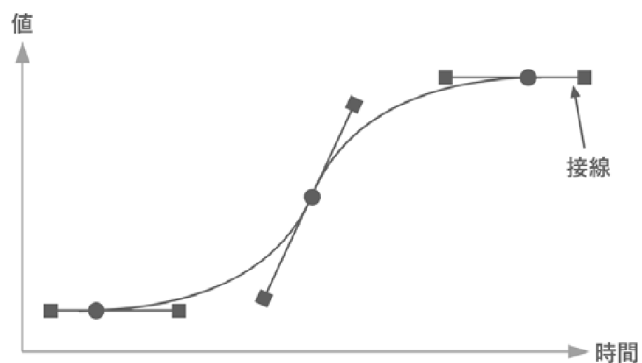


图2.27 切线和权重

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。

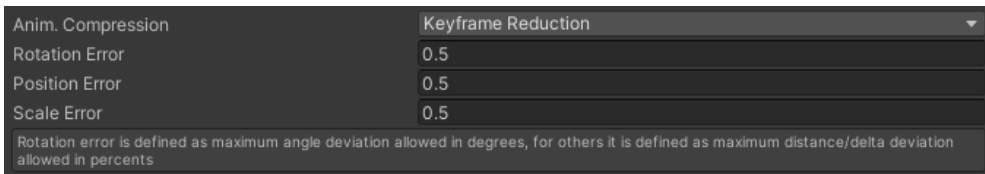


图2.28 导入设置。

关于如何设置的细节，见4.4 动画。

2.4 统一的工作方式

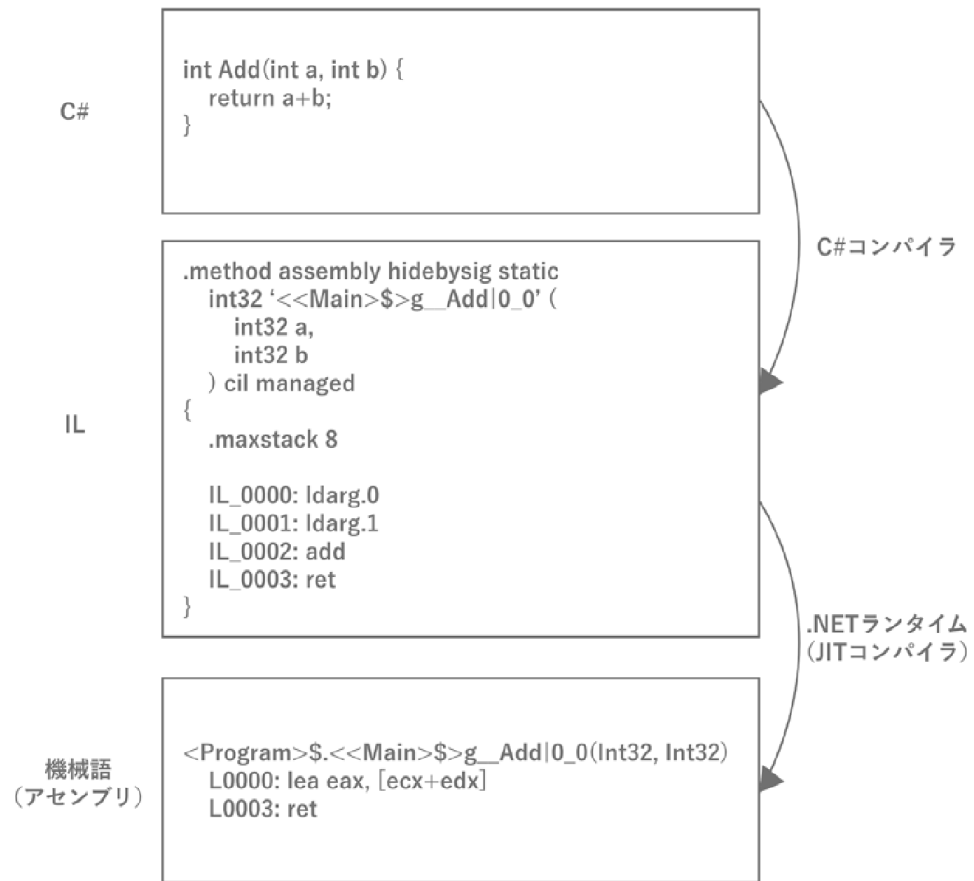
不言而喻，了解Unity引擎的实际工作原理对于调整你的游戏非常重要。本节解释你应该知道的Unity操作原则。

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#语言对其行为进行编程，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在相应环境的运行时上执行，从而实现多平台的兼容性。

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SharpLab²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。

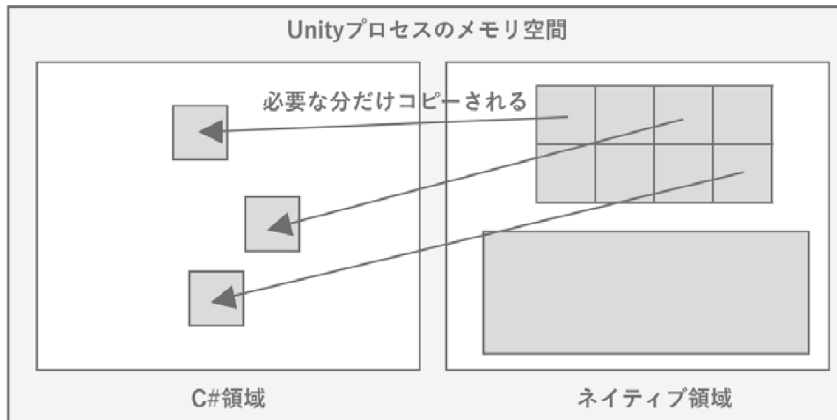


图2.30 Unity中内存状态的图像。

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集中在剖析和调试C#端，如果只看C#端的执行状态而不看E-mail，就很难理解。

^{*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。

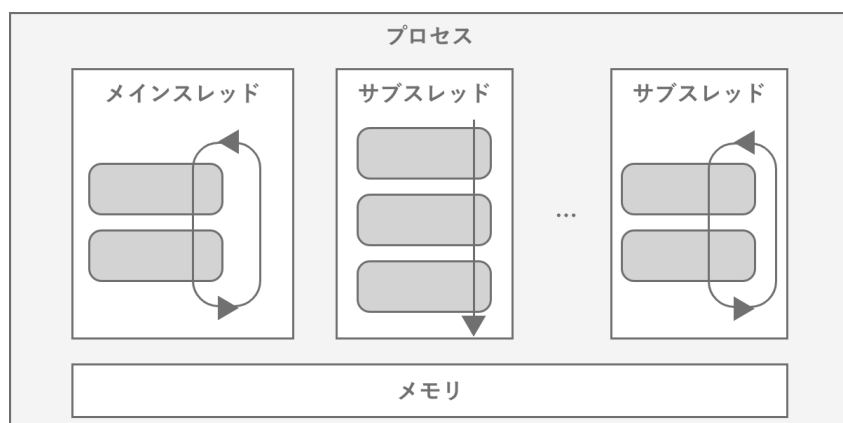


图2.31 线程的示意图

当程序运行时，会创建一个底层的主线程，程序会根据需要创建和管理其他线程。Unity的游戏循环被设计成在一个线程上运行，所以用户编写的脚本基本上是在主线程上运行。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

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，而这些进程又互不相干，如果这些进程能在一定程度上同步，使它们能并行处理，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。

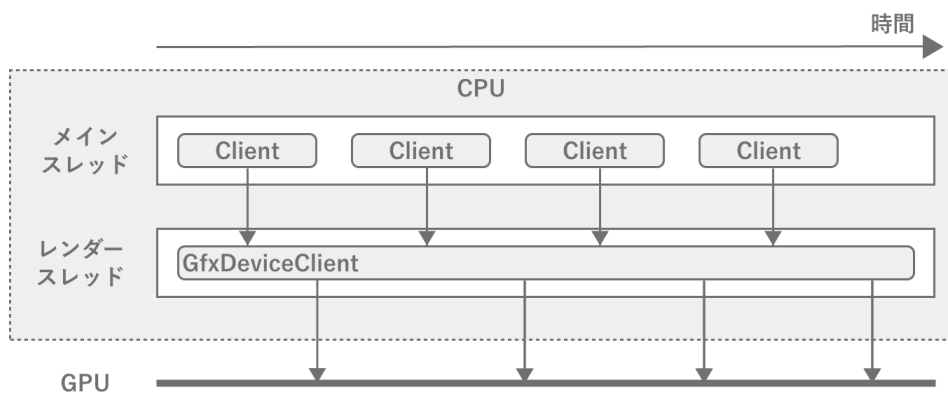


图2.32 主线程和渲染线程

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，还有一些可以并行执行的计算任务，如物理引擎和摇动，这些都是游戏特有的。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>

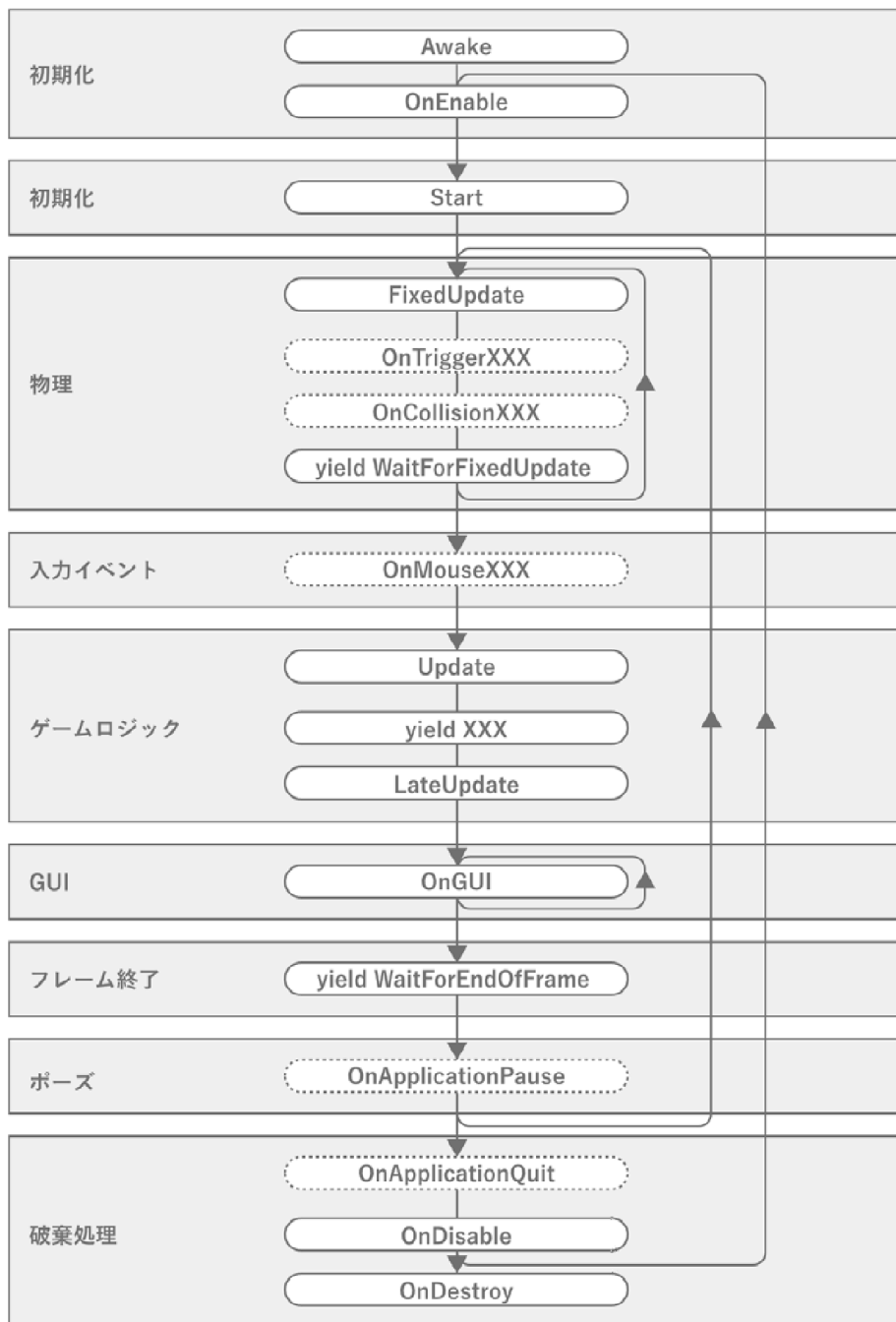


图2.33 Unity事件的执行顺序。

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coroutines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，如果这些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，因此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，会滑过，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>

▼ 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 `_gameObject` 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如下文表2.4所示。

▼ 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。

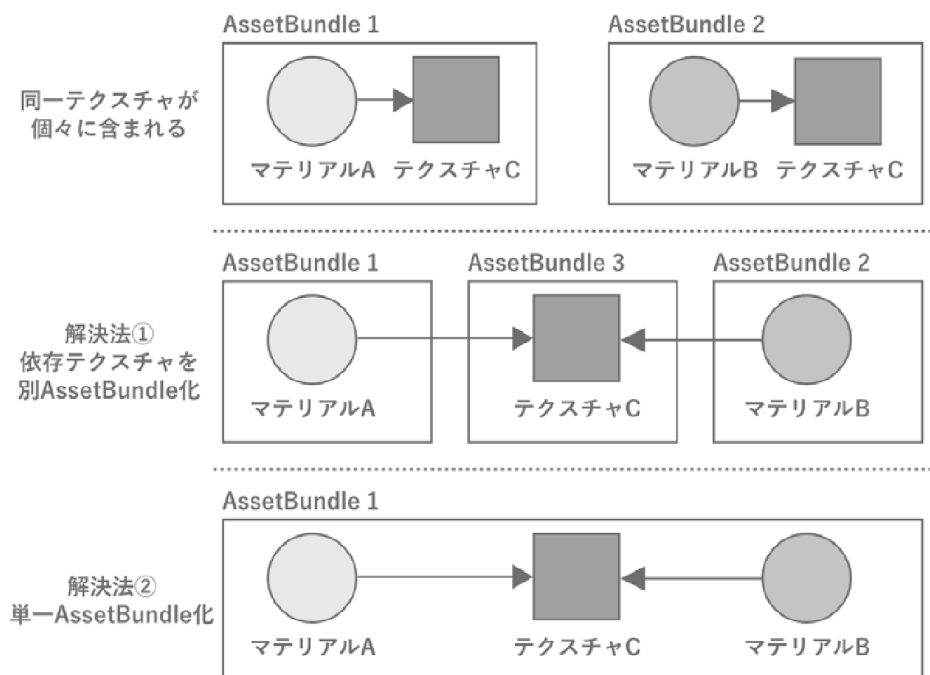
换句话说，未压缩的文件对最快的加载时间有好处，但为了避免浪费智能手机的存储空间，它基本上不能使用，因为文件大小是致命的大。另一方面，LZMA具有最小的文件大小，但由于算法问题，具有解压速度慢和部分解压的缺点。LZ4是一个平衡速度和文件大小的压缩设置，正如其名称ChunkBasedCompression所暗示的那样，允许部分阅读，而不必像LZMA那样解压整个文件。

AssetBundle也有Caching.compressionEnabled，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，终端侧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进行了资产捆绑，而没有对纹理进行资产捆绑，那么生成的两个资产捆绑将分别包含纹理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。

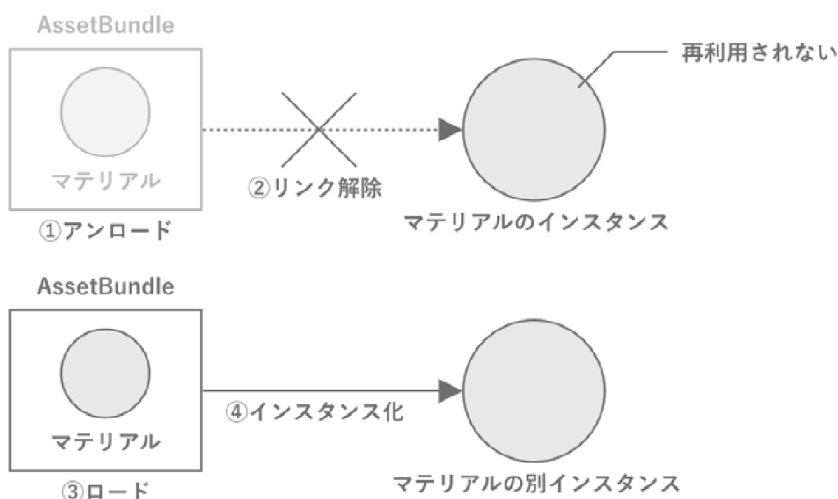


▲ 图2.34 带有AssetBundle依赖关系的例子。

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，而堆则由程序方管理。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。

原有的堆内存必须在需要的时候被分配，内存使用完毕后必须被释放。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，编码时往往会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对设计进行一次审查和重组。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。

内存分配目的地的差异

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，过量的入箱和出箱将导致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、毫无意义的拳法从未被使用过，但如果它被用在一个方法中呢？

▼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， $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$

表2.5和图2.36显示了数据的数量和计算步骤的数量。表2.5显示了数据数和计算步骤数的比较，图2.36显示了对数比较图。

$O(1)$ 被排除在外，因为它的性能与数据的数量无关，因此在没有比较的情况下，它的性能明显更优。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 and 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

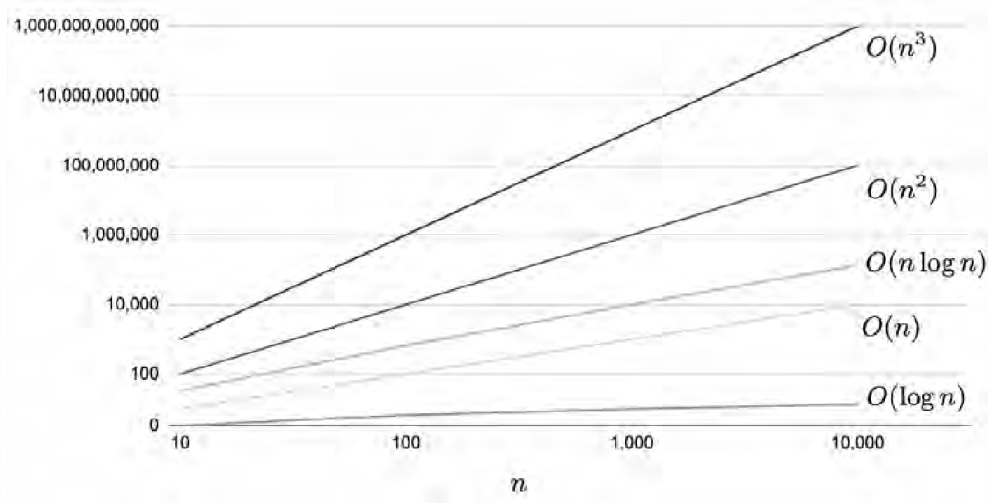


图2.36 各计算量的对数表示法的性能差异比较。

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
: {
3:     //让array成为一个包含某个整数值数组 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，并且需要固定的周期数（在本例中为1）。

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, 并包含一些整数值4:     for (var i
= 0; i < n; ++i)
5:     {
6:         var value = array[i];
7:         如果 (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:            如果(value1 == value2)
11:            {
12:                返回true。
13:            }
14:        }
15:    }
16:
17:    返回错误。
18: }
```

这个是一个双循环方法，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集合类时，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，这样可以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，它并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。

▼ 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，然后退出。这样一来，无论第一次的计算量有多大，都有可能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 $O(1)$ 。



统一

性能调整

CHAPTER

0

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. 1Unity Profiler

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表示時のバ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。



▲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