Unity 性能优化 (2019.3)

译文原址：《 Fixing Performance Problems 2019.3》

译者邮箱： lingzelove2008@163.com

第一章：游戏脚本最佳实践

引言

当游戏在移动设备上运行的时候，游戏的每一帧可能需要 CPU 执行上百万个指令；为了保持平稳的帧率，CPU 必须在规定时间内执行完相应的指令，一旦 CPU 无法及时执行所有指令，我们游戏就会变的卡顿

很多问题会导致 CPU 需要执行密集指令，比如：为渲染做准备工作、进行超复杂的物理运算或者太多的动画回调；本文主要探讨的是：我们编写的脚本导致的 CPU 性能问题。我们将学习脚本是如何转换成 CPU 指令，哪些代码会让 CPU 执行过多的指令，以及如何修复这些代码

诊断代码中的问题

CPU 超负荷运行的直接现象就是游戏不够流畅(帧频不稳)。然而其他性能问题也可能导致类似的现象。如果我们游戏有类似症状，我们首先想到的是要利用 Unity 的 Profiler 来分析是不是因为 CPU 超负荷运行引起的；如果是那么进一步分析到底因为代码问题引起的；还是因为物理计算或者动画太多引起的（如何使用 Profiler 分析性能问题，请移步到：Diagnosing Performance Problems tutorial.）

Unity 构建和运行游戏

为什么我们的代码可能造成性能问题呢？我们首先需要知道当构建游戏的时候发生了什么事情，了解幕后发生的事情有助于我们提高游戏性能。构建游戏的时候，Unity 会把游戏运行需要的所有东西打包到对应设备能运行的程序内。众所周知，CPU 只能执行机器代码(原生代码：native code)，不能执行一些高级语言编写的代码（比如: C#）。第一步：Unity 会将我们编写的高级语言转换成其他语言；这个转换过程我们称之为 **编译**（compiling）, Unity 会把我们写的 C# 代码编译成 CIL（译者注：这一步是通过 Mono 实现的），CIL 是一种很容易编译成不同原生代码的通用中间语言。第二步： CIL 然后被编译为目标设备对应的原生代码； 它发生在我们构建游戏时(静态编译，AOT)或代码运行之前(即时编译，JIT)

源代码与编译代码之间的关系

代码在未编译之前称为”源代码”，源代码直接决定了编译后的原生代码的结构和内容，结构良好且高效的源代码将生成结构良好且高效的已编译代码

有些 CPU 指令的执行时间比其他指令长。比如: 计算平方根比计算两个数相乘耗时更长。实际上 CPU 执行这两个指令的时间都非常短暂，但是它们之间相对来说还是有快慢之分

一些在源代码中看起来非常简单的操作，编译之后的原生代码会非常复杂。比如将一个元素插入列表，执行此操作所需要的指令要比按索引从数组中访问元素多得多

理解了以上两点，即便你对一些原生代码底层如果工作不是很熟悉，但是可以通过源代码和编译以后的代码之间的联系编写高性能的源代码

Unity 引擎代码与脚本代码在运行时通讯

用 C# 编写的脚本代码与 Unity 引擎核心代码的运行方式略有不同。Unity 引擎的内核代码是用 C++编写的，并且已经被编译成原生代码。这个编译后的引擎代码包含在我们安装的 Unity 程序中

源代码被编译成 CIL 以后，称为托管代码。当托管代码被编译成原生代码时，它与托管运行时集成。托管运行时负责内存管理和安全检查，以确保代码中的错误只是会导致异常，而不会造成设备崩溃

当数据从托管代码传递回引擎代码时，CPU 可能需要将数据从托管运行时使用的格式转换为引擎代码所需的格式，这种转换称为编组（marshlling）。托管代码和引擎代码之间的任何单个调用的开销都不是很大，但是我们必须知道这里是有开销的

代码性能不佳的原因

现在我们已经了解了 Unity 构建和运行游戏时代码会发生什么，我们也知道，我们的代码之所以性能很低，那是因为它们导致了 CPU 执行了密集指令

第一种可能是我们的代码很浪费或者结构很差。常见的例子可能是：当一个函数只需要被调用一次时，它却被重复调无数次

第二种可能是我们的代码看起来结构良好，但是对其他代码进行不必要的大量调用。这方面的一个例子可能是导致托管代码和引擎代码之间不必要的重复通讯从而造成 CPU 开销

第三种可能是我们的代码的没有任何问题，但是在不需要调用它的时候它依然被调用(一个例子是：怪物自动寻敌代码，当主角离怪物足够远的时候，这段代码应该停止执行)

最后一种可能是我们的代码需求太高了。这个我们只能重新设计游戏以降低它对性能的要求。实现这种优化超出了本文的范围，因为它非常依赖于游戏本身，但是阅读这篇文章并考虑如何使我们的游戏尽可能地拥有高性能仍然会对我们有好处

提高代码性能

一旦确定我们游戏中的性能问题是由于代码造成的，我们就应该仔细想想怎么解决这些问题。优化臃肿的函数似乎是个好的开始，当然也有可能这个函数为了实现这个功能已经是最优写法了，再去改动需要花费的代价太大（有可能导致功能异常或者带来新的bug）。我们可以在一个被数百个游戏对象所引用的脚本中创造一个小的效率节约，那么这就会给我来很可观的性能提升。此外，可能为了提高 CPU 的性能，会导致更多的内存消耗或者将工作量分担给了 GPU

尽量将代码移出循环

循环是个老生常谈的低效率代码之一，嵌套循环更甚。如果它们是被包含在 Update 这类每一帧都会被调用的函数中，那么效率更低。在下面的示例中，无论条件是否满足，for 循环每一帧都被执行，这是非常糟糕的代码

```
void Update()
{
	for(int i = 0; i < myArray.Length; i++)
	{
		if(exampleBool)
		{
			ExampleFunction(myArray[i]);
		}
	}
}
```

一个小小的改变，代码只在满足条件的情况下才会循环，情况就会好很多

```
void Update()
{
	if(exampleBool)
	{
		for(int i = 0; i < myArray.Length; i++)
		{
			ExampleFunction(myArray[i]);
		}
```

```
        }
}
```

这是一个简单的例子，我们应该认真检查代码中的循环结构，尤其是被频繁调用的循环。Update() 是每一帧都会被调用的函数，将代码移出 Update() 在需要时才运行，这是提高性能的一种好办法，而且也是最容易实现的办法之一

只有情况发生改变时才调用代码

让我们来看一个非常简单的例子(只在状态发生变化时才调用它)。在下面的代码中，在 Update() 中调用 DisplayScore() 。事实上分数的值不可能每一帧都发生变化，这意味着，我们可以将它移出 Update()

```
private int score;

public void IncrementScore(int incrementBy)
{
        score += incrementBy;
}

void Update()
{
        DisplayScore(score);
}
```

通过一个简单的更改，我们现在确保只有在 Score 的值发生改变时才调用 DisplayScore()

```
private int score;

public void IncrementScore(int incrementBy)
{
        score += incrementBy;
        DisplayScore(score);
}
```

每隔 X 帧执行代码

如果代码需要频繁运行并且不能被事件触发，那其实也并不意味着每一帧都需要调用，可以每隔 X 帧调用一次

```
void Update()
{
        ExampleExpensiveFunction();
```

```
}
```

我们每 3 帧调用一次这段代码就行了，在下面的代码中，我们使用模运算来确保该函数每隔 3 帧才会调用一次

```
private int interval = 3;

void Update()
{
	if(Time.frameCount % interval == 0)
	{
		ExampleExpensiveFunction();
	}
}
```

这种技术的另一个好处是很容易在不同的帧之间分散开销很大的代码，从而避免峰值。在下面的示例中，两个性能消耗很大的代码就可以错帧执行，非常好的方法

```
private int interval = 3;

void Update()
{
	if(Time.frameCount % interval == 0)
	{
		ExampleExpensiveFunction();
	}
	else if(Time.frameCount % 1 == 1)
	{
		AnotherExampleExpensiveFunction();
	}
}
```

缓存变量

我们通常调用 GetComponent() 来访问组件。在下面的示例中，我们在 Update() 中调用 GetComponent() 来获取 Renderer 组件，并将它传给 ExampleFunction() 使用。这段代码可以工作，但是它的效率很低

```
void Update()
{
	Renderer myRenderer = GetComponent<Renderer>();
	ExampleFunction(myRenderer);
}
```

下面的代码只调用 GetComponent() 一次，因为结果被变量 myRenderer 缓存。变量可以在 Update() 中重用，而无需调用 GetComponent() 重新获取

```
private Renderer myRenderer;

void Start()
{
        myRenderer = GetComponent<Renderer>();
}

void Update()
{
        ExampleFunction(myRenderer);
}
```

对于频繁调用返回结果的函数的情况，我们应该检查代码，最好通过使用缓存来降低这些调用

使用正确的数据结构

为了正确地决定使用哪种数据结构，我们需要了解不同数据结构的优缺点，并仔细考虑我们希望代码做什么。我们可能有数千个元素需要每一帧中迭代一次，或者我们可能有少量的元素需要频繁地添加和删除。这些不同的问题最好由不同的数据结构来解决。 如果数据结构对你来说是一个新的知识领域，可以参考:数据结构

尽量减少垃圾回收的影响

垃圾回收是 Unity 管理内存的一部分。我们的代码使用内存的方式决定了垃圾收集的频率和 CPU 成本，因此理解垃圾回收是如何工作的非常重要。在下一章中，我们将深入讨论垃圾回收，并提供几种不同的策略来最小化垃圾收集的影响

使用对象池

实例化和销毁一个对象通常比停用和重新激活一个对象花费更多。如果对象包含启动代码，例如在 Awake() 或 Start() 函数中调用 GetComponent()，则更是如此。如果我们需要生成和处理多个相同对象，比如射击游戏中的子弹，那么使用对象池就非常有必要

对象池是一种技术，它不是创建和销毁对象的实例，而是临时停用对象，然后根据需要回收和重新激活。尽管对象池作为一种管理内存使用的技术广为人知，但作为一种减少过度使用 CPU 的技术，它也很有用。Unity 教程上有关于在 Unity 中实现对象池的应用的内容

避免昂贵的 Unity API 的调用

有时，我们的代码对其他函数或 api 的调用可能会出乎意料性能消耗。因为有些看起来像变量的东西可能包含额外的代码、触发事件或从托管代码调用引擎代码的访问器（accessor）

在这一节中，我们将看一些 Unity API 调用的例子，这些调用的代价比看起来的要高。这些例子展示了成本的不同潜在原因，建议的解决方案可以应用于其他类似的情况

很遗憾，我无法列出哪些 Unity API 你绝对禁止调用。因为每个 API 有时候你必须使用，而且合理使用的时候开销并不会很大。在任何情况下，我们都必须仔细分析我们的游戏，找出导致开销昂贵的原因，并仔细考虑如何以最适合我们游戏的方式解决问题。

SendMessage()

SendMessage() 和 BroadcastMessage() 是非常灵活的函数，它们几乎不需要了解项目的结构，而且实现起来非常快。因此，这些函数对于原型或初学者来说非常有用。然而，它们是非常耗费性能的 API。这是因为这些函数使用了反射

建议仅将 SendMessage() 和 BroadcastMessage()用于快速出 Demo 的时期，并尽可能使用其他函数代替。例如，如果我们知道要在哪个组件上调用某个函数，我们应该直接引用该组件并直接调用该函数；如果我们不知道在哪个组件上调用该函数，那么可以考虑使用事件或委托

Find()

Find() 和相关的函数功能强大，但是开销也很大。这些函数需要 Unity 遍历内存中的每个 GameObject 和组件。这意味着在小项目中不需要特别关注效率，但是随着项目复杂性增加，它们的使用成本会越来越高

Transform

设置 transform 位置或旋转将导致内部 *OnTransformChanged* 事件传播到该 transform 的所有子节点。这意味着设置一个 transform 位置和旋转值相对比较昂贵，特别是在有许多子元素的转换中

为了限制这些内部事件的数量，我们应该避免设置这些属性的值。例如，我们不应该单独设置 transform 的 x,y,z 的值。而是将 x，y，z 的值放入 vector3 中，然后将 vector3 的值赋给 transform，这只会导致一个 *OnTransformChanged* 事件

如果代码经常使用 Transform.position，应该用 Transform.localPosition 替换。这将导致更少的 CPU 指令，并可能最终提高性能。如果我们经常使用 Transform.position，我们应该在可能的地方缓存它

Update()

Update()，LateUpdate() 和其他事件函数看起来像简单的函数，但是它们有隐藏的开销。每次调用这些函数，都需要在引擎代码和托管代码之间进行通信。除此之外，Unity 在调用这些函数之前还进行了一些安全检查。安全检查确保 GameObject 处于有效状态，没有被销毁等等

因此，空 Update() 调用可能特别浪费。你可能猜想，因为函数是空的，并且我们的代码不包含对它的直接调用，所以空函数不会运行。事实并非如此：在幕后，即使 Update() 函数的主体为空，这些安全检查和本机调用仍然会发生。为了避免浪费 CPU 时间，我们应该确保项目中不包含空的 Update()调用(译者注：可以写个脚本，在打包的时候执行该脚本，检测到了空 Update()函数，自动删除它)

关于 Update()，这篇文章带来了深入的探讨：10000 Update() calls

Vector2 和 Vector3

我们知道一些操作会导致比其他操作更多的 CPU 指令。向量数学操作就是一个例子：它们比浮点或整数操作更复杂。尽管两次这样的计算所花费的时间实际差异很小，但是在足够大的范围内，这样的操作可能影响性能

在数学运算中使用 Unity 的 Vector2 和 Vector3 结构是很常见和方便的，特别是在处理 transform 时。如果我们在 Update() 中的嵌套循环对大量游戏对象执行这些操作，我们很可能会给 CPU 带来不必要的工作。在这些情况下，我们可以通过执行 int 或 float 计算来节省性能

在文章的前面，我们了解到执行平方根计算所需的 CPU 指令比简单乘法所需的 CPU 指令要多的多。Vector2.Magnitude 和 Vector3.Magnitude 就是一个例子，因为它们都涉及平方根计算。如果我们的游戏广泛而频繁地使用 magnitude 或 Distance，那么我们就有可能使用 Vector2.sqrMagnitude 和 Vector3.sqrMagnitude 代替来避免相对昂贵的平方根计算

Camera.main

Camera.main 是一个方便的 Unity API ，它返回对第一个被标记为“Main Camera”的已激活相机的引用。这是另一个类似于变量但实际上是访问器（accessor）的例子。在这种情况下，访问器在幕后调用类似 Find()的函数。因此 Camera.main 面临着与 Find() 相同的问题：它搜索内存中的所有游戏对象和组件，并且使用起来非常耗费性能。为了避免这种潜在的昂贵调用，我们应该缓存 Camera.main 的结果，或者避免使用它，并手动管理对相机的引用

其他的 Unity API 调用和进一步优化

我们已经考虑了一些 Unity API 调用的常见例子，它们可能会带来意想不到的代价，并且了解这种代价背后的不同原因。然而，这并不是提高 Unity API 调用效率的全部方法。这里有一篇关于 Unity 性能优化的文章，它包含了许多我们可能会发现有用的其他 Unity API 优化

只在需要运行时运行代码

鲁迅说过：” 最快的代码是没有运行的代码” 。通常，解决性能问题的最有效方法不是使用高级技术而是删除不需要的代码

裁剪

Unity 包含代码来检查对象是否在摄像机视锥体内。如果它们不在摄像机的视锥体内，则渲染这些对象相关的代码不会运行。这里的术语是**视锥体剔除**

我们可以对脚本中的代码采取类似的方法。如果我们有与对象的可视状态相关的代码，当玩家看不到该对象时，我们可能不需要执行此代码。在有许多对象的复杂场景中，这可以节省大量的性能

在下面的简化示例代码中，我们有一个巡逻敌人的示例。每次调用 Update()时，控制这个敌人的脚本都会调用两个示例函数：一个与移动敌人有关，一个与它的可视状态有关

```
void Update()
{
        UpdateTransformPosition();
        UpdateAnimations();
}
```

在下面的代码中，我们现在检查敌人的渲染器是否在任何摄像机的视锥体内。与敌人的可视状态相关的代码仅在敌人可见时才运行

```
private Renderer myRenderer;

void Start()
{
   myRenderer = GetComponent<Renderer>();
}

void Update()
{
   UpdateTransformPosition();

   if (myRenderer.isVisible)
   {
      UpateAnimations();
   }
}
```

在玩家看不到的情况下禁用代码有几种方法。如果我们知道我们的场景中的某些对象在游戏中的某个特定点是不可见的，我们可以手动禁用它们。当我们不确定并且需要计算可见性时，我们可以使用粗略的计算（例如，检查玩家背后的对象）、OnBecameInvisible() 和 OnBecameVisible() 等函数，或者更详细的 raycast。最好的实现在很大程度上取决于我们的游戏，而实验和分析是必不可少的。

LOD

LOD 是另一种常见的渲染优化技术。最接近玩家的物体使用更详细的网格和纹理以完全保真的方式渲染。远处的物体使用较少的细节网格和纹理。我们的代码也可以使用类似的方法。例如，我们可能有一个敌人的 AI 脚本决定了它的行为。这种行为的一部分可能涉及昂贵的操作。以确定它可以看到和听到什么，以及它应该如何对输入作出反应。我们可以使用一个详细级别的系统来启用和禁用这些昂贵的操作

总结

我们已经了解了在 Unity 游戏构建和运行时我们编写的代码会发生什么变化，为什么我们的代码会导致性能问题，以及如何最小化昂贵代码对游戏的影响。我们已经了解了代码中导致性能问题的一些常见原因，并考虑了一些不同的解决方案。使用这些知识和分析工具，我们现在应该能够诊断、理解和修复与游戏中的代码相关的性能问题

第二章：优化垃圾回收

游戏运行的时候，会使用内存存储游戏中的一些数据，当这些数据不再需要的时候，存储这些数据的内存区域就可以重新被使用。“垃圾”就是用来形容存储在内存中但是再也不会被使用的数据，而垃圾回收就是清理这些数据释放这些被占用的内存块的过程。(译者注：垃圾回收并不是真的把内存块里面存储的数据先清空，它只是标记这个内存块可以被重新放入新的数据，新的数据会覆盖旧的数据，旧数据就被冲掉了)

Unity 的垃圾回收是 Unity 内存管理的一部分，游戏性能不佳，可能是因为垃圾回收太频繁，或者每一次垃圾回收要耗费大量时间和 CPU，因此，垃圾收回其实是导致游戏性能问题的常见原因之一

这一章，我们学习垃圾回收是怎么工作的，怎么高效的使用内存，以及如何减少垃圾回收对游戏性能的影响

Unity 的内存管理

为了搞明白垃圾回收什么时候发生以及如何工作，我们首先要理解在 Unity 引擎中，内存是如何被分配的。Unity 在执行核心引擎代码和执行我们的脚本代码时，使用了不同的方法来管理内存分配

Unity 在运行核心引擎代码的时候，使用的内存管理方法称为：**手动内存管理。**这意味着核心引擎代码必须明确当前内存的使用状态，它不采用垃圾回收机制，所有的内存分配和释放都需要自己去管理，关于手动内存管理，本章不做过多介绍，有兴趣的可以自己去百度（C++就是一种手动内存管理的语言）

Unity 在运行我们写的脚本代码时，使用的内存管理方法称为：**自动内存管理。**这意味着我们不需要明确的告诉 Unity 该怎么去管理内存，Unity 已经帮我们搞定

关于 Unity 自动内存管理的简单概述：

1. Unity 可以访问两个大的内存块：栈和堆（也称之为：托管堆），栈用于存放短期的小块数据，而堆用于存储长期的大块数据

2. 创建一个变量的时候，Unity 会在栈或者堆上申请当前类型所需大小的内存块

3. 当这个变量处于词法作用域(也就是能被访问到)的时候，那么这个变量所对应的内存块表示正在被使用，也可以说这块内存处于被分配状态。我们将栈上的变量称为栈内存对象，将堆上的对象称为堆内存对象

4. 当变量离开词法作用域，那么保存这个变量的内存地址就无法被引用到。那么这一块内存可以释放到内存池中被重新使用了。栈上的内存释放非常迅速，当变量不再被使用的时候，马上释放。但是堆上的对象即便它对应的变量已经不再被任何地方引用，也不会马上被释放，这一块的内存仍然是处于被分配状态

5. 垃圾回收器会定期标识和释放不再被引用的堆内存，整理和清理整个堆内存池

第二章：游戏图形渲染最佳实践

本文我们将学习 Unity 在渲染每一帧幕后所发生的事情，以及有哪些问题会导致渲染卡顿，以及怎么解决这些问题

阅读本文之前，首先要明白对于提升渲染性能，没有一招鲜的办法。影响渲染性能的因素太多：包括游戏类型、设备硬件、操作系统等。重要的是我们通过观察，实践得到数据然后分析数据从而有针对性的解决问题。本文包含一些常见的渲染性能问题和解决办法以及一些扩展链接供你更深层次的了解其原理；当然，很有可能你们游戏中出现的性能问题，这篇文章并没有指出；不过，这篇文章你还是值得看看的，理解事物的本质对解决问题很有帮助

渲染简介

开始之前我们快速的过一遍，看看 Unity 渲染每一帧时发生了什么，理解整个流程以及每个过程发生的时机会对我们解决性能问题有帮助。渲染流程如下：

- CPU（central processing unit）计算哪些物体需要渲染以及为这些物体设置渲染状态
- CPU 发送图形渲染指令给 GPU（graphics processing unit）
- GPU 渲染物体

“渲染管线（rendering pipeline）”通常用来描述渲染的过程；这十分贴切，高效的渲染就像车间的流水线，无停顿，高效率。渲染每一帧的过程中，CPU 都会做如下工作：

- 检测场景中的每一个物体是否需要渲染；一个物体被渲染需要符合一定的条件。比如：物体必须处于摄像机的可视范围内（注意：哪怕只有一部分处于可视范围，仍然需要渲染整个物体），不在范围内的则剔除，如果想了解更多关于物体剔除方面的内容，请查阅：Understanding the View Frustum
- CPU 收集和计算将要被渲染的物体的相关信息并发送命令给 GPU（包括将网格和纹理等传送到显存，以及设置渲染状态，调用图形 API），这个过程称之为：Draw Calls
- CPU 给每一个 Draw Call 创建的数据包，称之为：Batch（批次），批次有时候会包含一些 Draw Call 以外的数据，但这些数据对性能没有什么影响，因此本文将不会讨论这些数据

  每一个批次至少包含一个 Draw Call，CPU 针对批次会做如下事情：

- CPU 会发出更改指令，让 GPU 更改渲染状态，这个指令称之为：SetPass Call，SetPass Call 告诉 GPU 使用哪些设置去渲染网格。SetPass Call 指令只有在渲染下一个网格的设置和渲染上一个网格的设置不一样时，才会发出
- CPU 通过 Draw Call 命令告诉 GPU，就按照上一次 SetPass Call 的渲染设置去渲染指定的网格
- 有些情况下，一个批次可能需要不止一个 pass，pass 是一段 shader 代码，并且新的 pass 需要更改渲染状态，对于批次中的每个 pass ，CPU 必须发送新的 SetPass Call 指令，然后再次发送 Draw Call 通知 GPU 按照新设置的渲染状态去渲染网格

  同时，GPU 也会做如下事情：

- GPU 按照 CPU 发送到 Command buffer 内的指令顺序处理
- 如果当前指令是 SetPass Call，那么 GPU 更新渲染状态
- 如果当前指令是 Draw Call，那么 GPU 根据上一次设置的渲染状态来渲染网格。渲染网格的过程有很多阶段，这里就不一一阐述，不过你可以了解一下：顶点着色器（Vertex Shader）是用来处理网格顶点的，而片元着色器（Fagment Shader）是用来处理每一个像素的
- 这个过程会重复执行，直到 Command buffer 内的指令都被执行完毕

  大概了解了 Unity 渲染的简要流程，让我们来考虑渲染的过程中可能会出现的问题

  渲染问题

  关于渲染，最重要的一点是：每一帧之内，CPU 和 GPU 都要按时完成自己的任务。他们中任何一个任务超时的话，那就会造成渲染问题。渲染问题一般有两个基本原因：1. CPU

约束，渲染过程中，CPU 为每一帧渲染准备数据花费的时间太长，导致渲染瓶颈 2. GPU 约束，渲染数量过于膨大，导致 GPU 渲染一帧需要花费的时间过长

有果必有因，在性能出问题的时候，找出导致性能问题的原因才是首要任务。针对不同的问题，我们才能给出不同的解决方案。修复性能问题，其实也是一项平衡性的工作，比如：牺牲内存用于提高 CPU 性能，牺牲游戏画质解决渲染瓶颈。我们会使用 Unity 自带的两个工具来定位问题：Profiler 和 Frame Debugger

CPU 瓶颈

基本上，渲染每一帧的过程中，CPU 就干三件事儿：1. 决定渲染哪些物体 2. 为渲染准备好数据以及设置渲染状态 3.发送图形渲染 API 。 这三类工作包含很多独立任务，这些任务可能是通过多线程完成的；当这些任务被分配到不同的线程执行时，我们称之为：多线程渲染

Unity 的渲染过程中，有三类线程参与：主线程（main thread）、渲染线程（render thread）、辅助线程（worker threads）。主线程用于我们游戏中主要的 CPU 任务（也包括一些渲染任务），渲染线程主要是用来给 GPU 发送指令的，而辅助线程则用来处理单独的任务，比如：物体剔除和网格蒙皮计算。哪些任务执行在哪个线程，取决于我们游戏运行设备的硬件以及游戏的设置。比如：设备的 CPU 核心数量越多，我们就会线程越多的辅助线程。正是由于这个原因，在目标设备上进行性能分析十分有必要，在不同的设备上，我们的游戏表现可能有天壤之别

由于多线程渲染非常复杂而且非常依赖硬件条件，所以在尝试提升性能的时候，我们要理解是那些任务导致的 CPU 瓶颈。如果游戏卡顿是因为剔除物体是否在摄像机是窗内的线程超时，那么我们减少 Draw Call 对提升游戏体验，并没有什么卵用

注意：不是所有平台都支持多线程渲染，WebGL 就不支持。在不支持多线程的平台，所有的任务都是在同一个线程中执行。如果在这种平台碰到 CPU 瓶颈问题，那么就试着去优化所有可能对 CPU 性能有帮助的点

Graphics Jobs

Project Settings ->Player->Other Setting->Rendering-> Graphics jobs 选项可以让 Unity 将那些本该由主线程处理的渲染任务分配到辅助线程中。在可以使用该功能的平台上，它将带来显著的性能提升（可以开启和关闭这个功能来做性能对比，该功能目前仍是预览版）

发送指令到 GPU

发送命令到 GPU 这个过程，一般是 CPU 瓶颈的最常见原因。大多数平台这个任务是由渲染线程完成的，个别平台是在辅助线程（比如：PS4）

而最耗时的操作是 SetPass Call 指令，如果 C PU 性能问题是因为发送指令到 GPU 引起的，那么减少 SetPass Calls 的数量通常是最有效的改善性能的办法。我们可以通过 Profiler 查看到 SetPass Call 和 批次的数量。多少 SetPass Call 会造成性能问题，这个和游戏运行设备的硬件有很大关系，在高端 PC 上可以发送的 SetPass Call 数量远大于移动设备

SetPass Call 数量以及对应批次数量取决很多因素，稍后会详细介绍。然而一般来说：

- 减少批次数量 或者 让更多物体共用相同的渲染状态，通常会减少 SetPass Call 数量
- 减少 SetPass Call 数量，通常能提升 CPU 性能

减少批次能提升 CPU 性能，即便减少批次并没有减少 SetPass Call 的数量。因为 CPU 能够更有效的处理单个批次。一般来说有以下方式，能减少批次和 SetPass Call 的数量：

- 减少需要渲染的物体数量，能减少批次和 SetPass Call 的数量
- 减少每个物体被渲染的次数，能减少 SetPass Call 的数量
- 合并需要渲染的物体，可以减少渲染批次

减少需要渲染的物体数量

这是减少批次和 SetPass Call 最简单的方法了，有以下方法可以降低渲染物体的数量：

- 直接减少场景中的可见物体数量。比如：场景中有很多人物需要渲染，我们可以选择性的只渲染一部分人
- 使用摄像机的 Far Clipping Planes 属性来降低摄像机的绘制范围。这个属性表示距离摄像机多远的物体不再被渲染
- 通过 Occlusion culling 技术来关闭被其他物体完全遮挡的物体，这样就不用渲染被遮挡的物体了。请注意：这个功能不适用于所有场景，但是在某些场景它确实带来很可观的性能提升。另外，我们可以通过手动关闭物体来实现自己的遮挡剔除。比如：如果我们场景中包含一些过场切换才会出现的物体，那么在过场播放之前或者结束以后，就应该手动隐藏他们以减少需要渲染的物体。手动剔除往往比 Unity 提供的动态剔除更高效

减少每个物体渲染的次数

实时光，阴影，反射等效果可以极大的提高游戏的真实感；但其实这些效果都非常消耗性能，因为这些效果会导致物体被渲染多次

我们对游戏的渲染路径的设置，对这些功能的性能消耗也有实质性的影响。渲染路径：表示绘制场景的时候，渲染计算的执行顺序；不同的渲染路径最主要的区别是他们怎么处理实时光，阴影和反射。通常来说，如果我们的游戏运行在比较高端的设备上，并且运用了实时光，阴影和反射，那么延迟渲染（Deferred Rendering）是比较好的选择。向前渲染（Forward Rendering）适用于不使用以上功能的低端设备。实时光，阴影和反射是个比较复杂的功能，最好能研究相关主题充分了解实现原理，可参考：灯光和渲染-2019.3

不管选择哪种渲染路径，实时光，阴影，反射都会极大的影响性能，所以优化他们十分有必要：

- 动态光照是个非常复杂的主题，它超出了本文的讨论范畴，你可以参考这篇文章去深入了解：Shadow troubleshooting
- 动态光照非常消耗性能，当我们场景中包含很多静态物体时，我们可以利用烘焙技术去预先计算场景的光照，生成光照贴图，详细可以点击：Lighting
- 如果我们想使用实时阴影，你可以通过这篇文章来提交性能：Shadow Cascades。文章介绍了阴影的设置，以及这些设置怎么影响性能
- 反射探头创建真实的反射，但是会影响合批。因此我们最好最小化的使用它，反射探头的优化请参考：Reflection Probe performance

合批物体

一个批次可以包含多个物体的数据，为了能合并批次，物体必须满足如下条件：

- 共享相同的材质
- 一样的渲染状态（比如：纹理，shader 等）

合并物体确实可以提高性能；同时我们也要分析，合并物体带来的性能提高会不会反而造成其他方面更大的性能消耗。关于合批可以参考我上一篇文章：静态批处理、动态批处理、GPU Instancing

纹理图集（Texture Atlasing），是把大量的小尺寸的纹理合并成一张大的纹理。它通常在 2D 游戏以及 UI 系统中使用，Unity 内置了图集工具：Sprite Packer

我们也可以手动合并共享材质和纹理的网格（可通过编辑器直接操作，也可以在运行时调用 API 操作），但是我们必须意识到，有可能我们手动合并以后，本来会被剔除而无需渲染的物体，因为共享相同的网格而必须渲染，而这些物体可能还有光照，阴影等更加消耗性能的操作，所以要权衡利弊

在脚本中，我们必须小心使用 Renderer.material，这个接口会拷贝材质，并返回拷贝后的引用。这样做也会破坏合批，因为它和其他物体不再有相同的材质引用了。如果我们一定要访问合批物体的材质，应该使用 Renderer.shareMaterial

蒙皮网格

SkinnedMeshRenderers 通常用在网格动画中，渲染蒙皮的任务通常在主线程或者单独的辅助线程（取决于游戏的设置以及目标设备硬件）。渲染蒙皮是个很消耗性能的动作，如果在 Profiler 中看到是渲染蒙皮对 CPU 造成性能瓶颈，这里有几个方法我们可以改进性能

- 考虑是否真的有必要使用蒙皮网格渲染组件，如果物体并不需要运动，尽可能使用普通的网格渲染组件（MeshRenderer）
- 如果我们只在某些时刻运动物体，我们应该用细节较少的网格，（SkinnedMeshRenderers 组件有一个函数 BakeMesh，可以用匹配的动作创建一个网格）
- 关于蒙皮网格的优化，可以参考：Skinned Mesh Renderer。（蒙皮网格消耗是在每个顶点上，因此，使用顶点较少的模型以及减少骨骼数量也可以提高性能）
- 在某些平台，蒙皮可以被 GPU 快速处理。如果目标设备 GPU 比较强，则可以对当前平台开启 GPU Skinning

GPU 瓶颈

如果游戏是 GPU 性能限制，那么首先就得找到 GPU 瓶颈原因。一般 GPU 性能最常见的问题是填充率限制，尤其是移动平台。当然显存带宽和顶点处理也可能影响

填充率

填充率是指 GPU 在屏幕上每秒可以渲染的像素数量；如果我们游戏是因为填充率导致的 GPU 性能问题，那么意味着我们游戏每帧尝试绘制的像素数量超过了 GPU 的处理能力，检查是否填充率引起的 GPU 性能问题其实很简单：

- 打开 Profiler，注意 GPU 时间

- 重新设置渲染分辨率 Screen.SetResolution(width,height,true)
- 重新打开 Profiler，如果 GPU 性能提升，那么大概率就是填充率的原因了

  如果是填充率问题，那么我们有如下几个方法解决这个问题

- 片元着色器是告诉 GPU 怎么去绘制每一个像素的 shader 代码，如果这段代码效率低，那么就容易发生性能问题，复杂的片元着色器是很常见的引起填充率问题的原因
- 如果我们游戏使用了 Unity 内置的 Shader，那么应该使用针对移动平台的 the mobile shaders
- 如果游戏使用的是 Unity 的 Standard Shader，那么要理解 Unity 编译这些 shader 是基于当前材质设置，只有那些当前使用的功能才会被编译。这意味着，移除 detail maps 可以减少片元着色器的复杂度
- 如果使用的是定制的 shader，那么应该尽量优化它。优化 shader 是个很大的话题，可以参考：Optimizations
- Overdraw 是指相同位置的像素被绘制了多次。一般发生在某个物体在其他物体之上。为了理解 Overdraw，我们必须理解 Unity 在场景中绘制物体的顺序。物体的 shader 决定了物体的绘制顺序，通常由 render queue 属性决定。最常见引起 Overdraw 的因素是透明材质，未优化的粒子以及重叠的 UI 元素。关于 Overdraw 优化，可以参考：Optimizing Unity UI

  显存带宽（Memory bandwidth）

  显存带宽是指 GPU 读写专用内存的速度，如果我们的游戏受限于显存带宽，通常意味着我们使用的纹理太大了，我们可以用如下方法检测是否是显存带宽问题：

- 打开 Profiler，并关注 GPU 各项数据
- Project Settings->Quality->Texture Quality，设置纹理质量，降低当前平台的纹理质量
- 重新打开 Profiler，重新查看 GPU 各项数据。如果性能改善，那么就是显存带宽问题

  如果是显存带宽问题，那么我们需要降低纹理的内存占用，针对不同游戏通常有不同的解决方案，这里我们提供几个优化纹理的方法

- 纹理压缩技术可以同时极大的降低纹理在内存中的占用。Texture Import Setting 讲述了纹理压缩格式和各种设置的详细信息

- Mipmaps，多级渐远纹理是 Unity 对远处物体使用低分辨率纹理。Unity 场景视图中的 The Mipmaps Draw Mode 允许我们查看哪些物体适用多级渐远纹理。其实，Mipmap 最主要的目的是为了提高质量，如果没有 mipmap 纹理很大，采样频率却很小的情况下，模型看去来质量会很差

  相邻的两个屏幕像素采样的纹素差的很远

  此时会大大降低缓存命中率

  顶点处理

  顶点处理是指 GPU 处理网格中的每一个顶点，顶点处理的消耗主要受两个因素的影响：顶点数量以及操作每个顶点的复杂度。有一些方法可以优化这个：

- 降低网格复杂度
- 使用法线贴图模拟更高几何复杂度的网格，关于法线贴图可以参考：Normal map
- 如果游戏未使用法线贴图，在网格导入的设置中，可以关闭顶点的切线，这可以降低每个顶点的数据量
- LOD，当物体远离摄像机的时候，降低物体网格的复杂度的技术，可以有效的降低 GPU 需要渲染的顶点数量，并且不会影响视觉表现，具体细节请参考：LODGroup
- 顶点着色器，是一段 shder 代码，降低它的复杂度，可以提升性能