Unity 游戏优化

(第 3 版)

[意]大卫·阿韦尔萨(Davide Aversa)
[英]克里斯·迪金森(Chris Dickinson) 著
蔡俊鸿 译

清華大學出版社
北 京

北京市版权局著作权合同登记号 图字：01-2021-2898

图书在版编目(CIP)数据

Unity 游戏优化：第 3 版 / (意) 大卫·阿韦尔萨(Davide Aversa)，(英)克里斯·迪金森(Chris Dickinson)著；蔡俊鸿译. — 北京：清华大学出版社，2022.8

书名原文：Unity Game Optimization: Enhance and extend the performance of all aspects of your Unity games, Third Edition

ISBN 978-7-302-61328-2

I. ①U… II. ①大… ②克… ③蔡… III. ①游戏程序－程序设计 IV. ①TP311.5

中国版本图书馆 CIP 数据核字(2022)第 122360 号

责任编辑：王 军
封面设计：孔祥峰
版式设计：思创景点
责任校对：成凤进
责任印制：朱雨萌

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-83470000 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经 销：全国新华书店
开 本：170mm×240mm 印 张：20.25 字 数：397 千字
版 次：2022 年 9 月第 1 版 印 次：2022 年 9 月第 1 次印刷
定 价：98.00 元

产品编号：092653-01

作者简介

Davide Aversa 博士拥有意大利罗马大学(University of Rome La Sapienza)的人工智能博士学位以及人工智能和机器人硕士学位。他对用于开发交互式虚拟代理和程序内容生成的人工智能有着浓厚的兴趣。他曾担任电子游戏相关会议的程序委员会成员，如 IEEE 计算智能和游戏会议，也经常参加 game-jam 比赛。他还经常撰写有关游戏设计和游戏开发的博客。

我要感谢家人在这一年里给我提供了稳定的生活；感谢 Twitter 上的 Unity 开发者帮助我澄清了 Unity 内部最模糊的元素；还要感谢 Keagan 和 Packt Publishing 的其他编辑帮助我完成这项工作，并对我延迟交稿表示理解。

Chris Dickinson 在英格兰一个安静的小镇长大，对数学、科学，尤其是电子游戏满怀热情。他喜欢玩游戏并剖析游戏的玩法，并试图确定它们是如何工作的。在看了爸爸破解一个 PC 游戏的十六进制代码来规避早期的版权保护后，他完全震惊了，他对科学的热情在当时达到了顶峰。Chris 获得电子物理学的硕士学位后，他飞到美国加州，在硅谷中心的科学研究领域工作。不久后，他不得不承认，研究工作并不适合他。在四处投简历之后，他找到了一份工作，最终让他走上了软件工程的正确道路(据说，这对于物理学专业毕业生来说并不罕见)。

Chris 是 IPBX 电话系统的自动化工具开发人员，他的性格更适合从事该工作。现在，他正在研究复杂的设备链，帮助开发人员修复和改进这些设备，并开发自己的工具。Chris 学习了很多关于如何使用大型、复杂、实时、基于事件、用户输入驱动的状态机方面的知识。在这方面，Chris 基本上是自学成才的，他对电子游戏的热情再次高涨，促使他真正弄清楚了电子游戏的创建方式。当他有足够的信心时，他回到学校攻读游戏和模拟编程的学士学位。当他获得学位时，已经可以用 C++编写自己的游戏引擎(尽管还很初级)，并在日常工作中经常使用这些技能。然而，由于想创建游戏(应该只是创建游戏，而不是编写游戏引擎)，Chris 选择了他最喜欢的公开发行的游戏引擎——一个称为 Unity3D 的优秀小工具，并开始制作一些游戏。

经过一段时间的独立开发游戏，Chris 遗憾地决定，这条特定的职业道路并不适合他，但他在短短几年内积累的知识，以大多数人的标准来看，令人印象深刻，

他喜欢利用这些知识帮助其他开发人员创建作品。从那以后，Chris 编写了一本关于游戏物理的教程(*Learning Game Physics with Bullet Physics and OpenGL*，Packt Publishing)和两本关于 Unity 性能优化的书籍。他娶了他一生的挚爱 Jamie，并在加州圣马特奥市的 Jaunt 公司(这是一家专注于提供 VR 和 AR 体验(如 360 视频)的虚拟现实/增强现实初创公司)工作，研究最酷的现代技术，担任测试领域的软件开发工程师(SDET)。

工作之余，Chris 一直苦恼于对棋盘游戏的沉迷(特别是《太空堡垒：卡拉狄加与血腥狂怒》)，他痴迷于暴雪的《守望先锋》和《星际争霸 2》，专注于Unity 最新版本，经常在纸上勾画关于游戏的构思。不久的将来，当时机成熟的时候(当他不再懈怠时)，相信他的计划就会实现。

审校者简介

Vincent Chu 是一名专业的 Unity 首席开发者(一名认证专家)，他领导着全球多个游戏项目，并在全球算法竞赛中名列前茅。他具有 Unity 游戏开发、软件架构、3D 建模和动画、渲染和着色器、网络和云解决方案方面的专业知识。

前　言

用户体验在所有游戏中都是重要的组成部分，它不仅包括游戏的剧情和玩法，也包括运行时画面的流畅性、与多人服务器连接的可靠性、用户输入的响应性，甚至由于移动设备和云下载的流行，它还包括最终程序文件的大小。由于 Unity 等工具提供了大量有用的开发功能，还允许个人开发者访问，因此游戏开发的门槛已大大降低。然而，由于游戏行业的竞争激烈，玩家对游戏最终品质的期望日益提高，因此就要求游戏的各方面应能经得起玩家和评论家的考验。

性能优化的目标与用户体验密不可分。缺乏优化的游戏会导致低帧率、卡顿、崩溃、输入延迟、过长的加载时间、不一致和令人不舒服的运行时行为、物理引擎的故障，甚至过高的电池消耗(移动设备通常被忽略的指标)。只要遭遇上述问题之一，就是游戏开发者的噩梦，因为即使其他方面都做得很好，评论也会只炮轰做得不好的一个方面。

性能优化的目标之一是最大化地利用可用资源，包括 CPU 资源，如消耗的 CPU 循环数、使用的主内存空间大小(称为 RAM)，也包括 GPU 资源(GPU 有自己的内存空间，称为 VRAM)，如填充率、内存带宽等。然而，性能优化最重要的目标是确保没有资源会不合时宜地导致性能瓶颈，优先级最高的任务得到优先执行。哪怕很小的、间歇性的停顿或性能方面的延迟都会破坏玩家的体验，打破沉浸感，限制开发人员尝试创建体验的潜力。另一个需要考虑的事项是，节省的资源越多，在游戏中创建的活动便越多，从而产生更有趣、更生动的玩法。

同样重要的是，要决定何时后退一步，停止增强性能。在一个拥有无限时间和资源的世界里，总会有一种方法能让游戏变得更出色、更快、更高效。在开发过程中，必须确定产品达到了可接受的质量水平。如果不这样做，就会重复实现那些很少或没有实际好处的变更，而每个变更都可能引入更多的 bug。

判断一个性能问题是否值得修复的最佳方法是回答一个问题：“用户会注意到它吗？”如果这个问题的答案是“不”，那么性能优化就是白费力气。软件开发中有句老话：

过早的优化是万恶之源。

过早优化是指在没有任何必要证据的情况下，为提高性能而重新编写和重构代码。这可能意味着在没有显示存在性能问题的情况下进行更改，或者进行更改

的原因是，我们只相信性能问题可能源于某个特定的领域，但没有证据证明的确存在该问题。

当然，Donald Knuth 提出的这一常见说法的含义是，编写代码时应该避免更直接、更明显的性能问题。然而，在项目末尾进行真正的性能优化将花费很多时间，而我们应该做好计划，以正确地改善项目，同时避免在未进行验证的情况下实施开销更大和更耗时的变更。这些错误会使整个软件开发团队付出沉重的代价，为没有成效的工作浪费时间是令人沮丧的。

本书介绍在 Unity 程序中检测和修复性能问题所需的工具、知识和技能，不管这些问题源于何处。这些瓶颈可能出现在 CPU、GPU 和 RAM 等硬件组件中，也可能出现在物理、渲染和 Unity 引擎等软件子系统中。

在每天充斥着高质量新游戏的市场中，优化游戏的性能将使游戏具有更大的成功率，从而增加在市场上脱颖而出的机会。

本书内容

本书适合想要学习优化技术，用新的 Unity 版本创建高性能游戏的游戏开发者。

第 1 章探索 Unity Profiler，研究剖析程序、检测性能瓶颈以及分析问题根源的一系列方法。

第 2 章学习 Unity 项目中 C#脚本代码的最佳实践，最小化 MonoBehaviour 回调的开销，改进对象间的通信等。

第 3 章探索 Unity 的动态批处理和静态批处理系统，讨论如何使用它们减轻渲染管线的负担。

第 4 章介绍艺术资源的底层技术，学习如何通过导入、压缩和编码避免常见的陷阱。

第 5 章研究 Unity 内部用于 3D 和 2D 游戏的物理引擎的细微差别，以及如何正确地组织物理对象，以提升性能。

第 6 章深入探讨渲染管线，如何改进在 GPU 或 CPU 上遭受渲染瓶颈的应用程序，如何优化光照、阴影、粒子特效等图形效果，如何优化着色器代码，以及一些用于移动设备的特定技术。

第 7 章关注 VR 和 AR 等娱乐媒介，还介绍了一些针对这些平台构建的程序所独有的性能优化技术。

第 8 章讨论如何检验 Unity 引擎、Mono 框架的内部工作情况，以及这些组件内部如何管理内存，以使程序远离过高的堆分配和运行时的垃圾回收。

第 9 章研究了多线程密集型游戏的 Unity 优化——DOTS，介绍了新的 C#作

业系统、新的 Unity ECS 和 Burst 编译器。

第 10 章讲解了如何将 Skinned MeshRenderer 转为 MeshRenderer，同时启用 GPU Instancing 优化大量动画对象。本章由译者根据本书内容所编写，对 Unity 的较新技术做了补充。

第 11 章介绍 Unity 专家用于提升项目工作流和场景管理的大量有用技术。

阅读本书的条件

本书主要关注 Unity 2019 和 Unity 2020 的特性和增强功能。书中讨论的很多技术可应用到 Unity 2018 或更旧版本的项目中，但这些版本列出的特性可能会有所不同，这些差异会在适当的地方突出显示。

值得注意的是，书中的代码应该用于 Unity 2020，但在撰写本文时，只能在 alpha 版本上进行测试。额外的不兼容性可能会出现在 Unity 2020 的非 alpha 阶段。

下载示例代码文件

本书提供相关代码、参考网站，以及本书中使用的屏幕截图、图表的彩色图像，可以扫描本书封底的二维码下载。

如果代码有更新，在现有的 GitHub 存储库上也会有更新。

约定

代码块的格式设置如下：

```
void DoSomethingCompletelyStupid() {
  Profiler.BeginSample("My Profiler Sample");
  List<int> listOfInts = new List<int>();
  for(int i = 0; i < 1000000; ++i) {
    listOfInts.Add(i);
  }
  Profiler.EndSample();
}
```

注意：警告或重要提示出现在此处。

提示：提示和技巧出现在此处。

读者反馈

欢迎读者提供反馈。

如果你对本书的任何内容有问题，请在电子邮件的主题中提到本书的标题，并发送电子邮件到 bookservice@263.net。

目　录

第Ⅰ部分　基本的脚本优化

在第Ⅰ部分，读者将学习如何使用内置的 Profiler 识别性能瓶颈，以及如何解决一些最常见的问题。本部分涵盖：

- 第 1 章　研究性能问题
- 第 2 章　脚本策略

第 1 章

研究性能问题

大多数软件产品的性能评估是一个非常科学的过程。首先，确定所能支持的最大/最小性能指标，如允许的内存使用量、可接受的 CPU 消耗量、并发用户数量等。接下来，使用为目标平台构建的应用程序版本在场景中对应用程序执行负载测试，并在收集检测到的数据时对其进行测试。收集了这些数据之后，就可以分析和搜索性能瓶颈。如果发现问题，将完成根源分析，对配置或应用程序代码进行更改以修复问题并重复此过程。

虽然游戏开发是一个非常艺术化的过程，但它仍然具有技术性。游戏应该有目标受众，来确定运行游戏的硬件限制是什么，需要达到什么性能目标(特别是主机游戏和手机游戏)。可以在应用程序上执行运行时测试，从多个子系统(CPU、GPU、内存、物理引擎、管道渲染等)中收集性能数据，并将它们与开发人员认为可以接受的数据进行比较。这些数据可以用来识别应用程序中的瓶颈，执行额外的检测，并确定问题的根源。最后，根据问题的类型应用修复程序来改进应用程序的性能，使其更符合预期。

修复性能之前，首先需要证明存在性能问题。在未确定存在性能问题之前，重写和重构代码是不明智的，因为预先优化很少能解决问题。一旦找到了存在性能问题的证据，下一个任务就是准确地找出瓶颈所在。确保理解为什么会出现性能问题是很重要的，否则可能会浪费更多的时间来应用补丁，而这些补丁只不过是有根据的猜测。这样做往往意味着只解决了问题的一个方面，而不是问题的根源，因此问题可能会在未来以其他方式，或以尚未发现的方式表现出来。

本章将探讨以下问题：

- 如何使用 Unity Profiler 收集分析数据。
- 如何分析 Profiler 数据以找到性能瓶颈。

- 隔离性能问题并确定问题根源的技术。

对以上问题有了全面了解后，就可以为后续章节的学习做好准备，后续章节将介绍对检测到的问题可用的解决方案。

1.1　使用 Unity Profiler 收集分析数据

Unity Profiler 内置于 Unity 编辑器中，它通过在运行时为大量的 Unity3D 子系统生成使用情况和统计报告，提供了一种缩小性能瓶颈搜索范围的便捷方法。可以供不同的子系统收集的数据如下：

- CPU 消耗量(每个主要子系统)。
- 基本和详细的渲染与 GPU 信息。
- 运行时内存分配和总消耗量。
- 音频源/数据的使用情况。
- 物理引擎(2D 和 3D)的使用情况。
- 网络消息传递和活动情况。
- 视频回放的使用情况。
- 基本和详细的用户界面性能。
- 全局光照统计数据。

一般情况下，有两种使用 Profiler 工具的方法：指令注入(instrumentation)和基准分析(benchmarking)，这两个术语通常可以互换。

指令注入通常意味着通过观察目标函数调用的行为，给哪里分配了多少内存，来密切观察应用程序的内部工作情况，会得到当前执行情况的精确图像，并可能找到问题的根源。然而，这通常不是一种发现性能问题的高效方法，因为任何应用程序的性能分析都会带来性能损耗。

当 Unity 应用程序在开发模式(由 Build Settings 菜单中的 Development Build 标志决定)下编译时，会启用附加的编译标志，导致应用程序在运行时生成特殊事件，这些事件会被 Profiler 记录并存储。当然，由于应用程序承担了所有额外工作负载，这将在运行时导致额外的 CPU 和内存开销。更糟的是，如果应用程序是通过 Unity 编辑器进行分析的，则会消耗更多的 CPU 和内存，从而确保编辑器更新其界面，渲染额外的窗口(如场景窗口)，并处理后台任务。这种分析成本并非总是可以忽略不计。在超大型的项目中，当启用 Profiler 时，有时会导致各种不一致和意想不到的行为：Unity 可能会耗尽内存，一些脚本可能会拒绝运行，物理引擎可能会停止更新(用于一帧的时间可能太长，以至于物理引擎达到每帧允许的最大更新)，等等。这是为了在运行时深入分析代码的行为而付出的必要代价，应该始终意识到它的存在。因此，在开始分析应用程序中的每一行代码之前，最好先对应

用程序进行一些基准分析。

基准分析涉及对应用程序执行表面级别的检测，应该在游戏运行于目标硬件期间收集一些基本数据，执行测试场景。测试用例可以是只持续几秒钟的简单游戏、不同场景的切换、一个关卡的一部分等。这个活动的目的是对用户可能体验到的东西有一个大致的感觉，并在性能明显变差时持续关注一段时间。这些问题可能相当严重，需要进一步分析。

进行基准分析的过程中，开发人员感兴趣的重要指标通常是渲染帧率(frame per second，FPS)、总体内存消耗和 CPU 活动的行为方式(寻找活动中较大的峰值)，有时还有 CPU/GPU 温度。这些指标的收集相对简单，可以作为性能分析的最佳首选方法，一个重要的原因是从长远来看，它会节省大量的时间，因为它确保开发人员只在用户会注意到的问题上花费时间。

只有在基准分析测试表明需要进一步分析之后，才应该更深入地研究注入的指令。如果想要真实的数据样本，应该尽可能多地模拟实际平台的行为，从而进行基准分析，这也是非常重要的。因此，不应该将通过 Editor 模式生成的基准数据作为真正游戏时的数据，因为 Editor 模式会带来一些额外的开销，可能会误导开发人员或者隐藏真实应用程序中潜在的竞争条件。相反，应用程序以独立格式在目标硬件上运行时，应将分析工具挂接到应用程序中。

许多 Unity 开发人员惊讶地发现，Editor 计算操作结果的速度有时比独立应用程序快得多，这在处理序列化数据(如音频文件、预制块和可脚本化的对象)时特别常见。这是因为编辑器将缓存以前导入的数据，能够比实际应用程序更快地访问这些数据。

下面介绍如何访问 Unity Profiler 并将它连接到目标设备，以便开始进行精确的基准分析测试。

提示：如果用户熟悉如何将 Unity Profiler 连接到应用程序，则可以跳到 1.1.2 节“Profiler 窗口”进行学习。

1.1.1 启动 Profiler

下面以一个简短的教程为例，讲解如何在不同的环境下将游戏连接到 Unity Profiler。

- 应用程序的本地实例，包括通过编辑器运行或独立运行的实例。
- 在浏览器上运行的 WebGL 应用程序的本地实例。
- 运行在 iOS 设备(如 iPhone 或 iPad)上的应用程序的远程实例。
- 运行在 Android 设备(如 Android 平板或手机)上的应用程序的远程实例。

- 分析编辑器自身。

下面简单介绍针对这些环境设置 Profiler 的要求。

1. 通过编辑器运行或独立运行的实例

在本实例中，访问 Profiler 的唯一方法是通过 Unity 编辑器启动它，并将它连接到应用程序的运行实例上。无论是在编辑器中以播放模式运行游戏，还是在本地或远程设备上运行独立的应用程序，又或者希望对编辑器本身进行配置，都是通过这种方式访问 Profiler。

要打开 Profiler，首先在编辑器中选择 Window | Analysis | Profiler 命令，或按 Ctrl +7 键(macOS 操作系统中按 cmd + 7 键)，如图 1-1 所示。

图 1-1　选择 Window | Analysis | Profiler 命令

如果编辑器已经在 Play 模式下运行，那么应该可以看到 Profiler 窗口正在收集分析数据。

要分析独立运行的项目，应确保在构建应用程序时启用了 Development Build 和 Autoconnect Profiler 标志。

通过 Profiler 窗口的 OSXPlayer(phoebe)选项，可以选择是分析基于编辑器的实例(通过编辑器的 Play 模式运行)，还是分析独立的实例(在编辑器外独立构建并运行)，如图 1-2 所示。

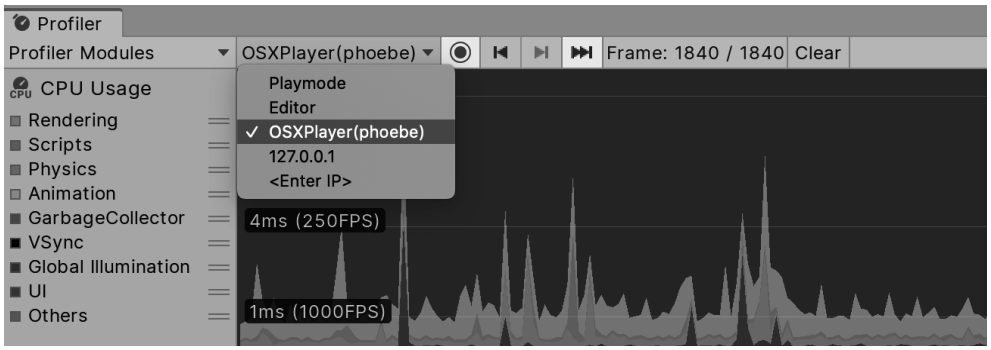

图 1-2 在 Profiler 中连接到游戏

注意：若在分析独立运行的项目时切换到Unity 编辑器，会挂起所有数据收集任务，因为应用程序不会在后台更新。

2. 连接到 WebGL 实例

Profiler 也可以连接到 Unity WebGL Player 的实例。为此，可以在构建 WebGL 应用程序并从编辑器中运行它时，确保启用 Development Build 和 Autoconnect Profiler 标志。应用程序会通过操作系统默认的浏览器运行，这将允许通过目标浏览器在更真实的场景中分析基于网页的应用程序，并测试在多种浏览器上出现的不一致行为(尽管需要不断地更改默认浏览器)。

遗憾的是，Profiler 连接只能在应用程序首次从编辑器中启动时建立。当前的 Profiler 不能连接到已在浏览器中运行的独立 WebGL 实例上，这限制了基准分析 WebGL 应用程序的准确性，因为有一些基于编辑器的开销，但它是此时唯一可用的方案。

3. 远程连接到 iOS 设备

Profiler 也可以连接到远程运行在 iOS 设备(如 iPad 或 iPhone)上的应用程序实例，通过共享 Wi-Fi 连接来实现。

注意：仅当 Unity(也是指 Profiler)运行在 Apple Mac 设备上时，才能远程连接到 iOS 设备。

可按照以下步骤将 Profiler 连接到 iOS 设备上。

(1) 确保构建应用程序时启用了 Development Build 和 Autoconnect Profiler 标志。

(2) 将 iOS 和 Mac 设备连接到本地 Wi-Fi 网络，或者连接到专用的 Wi-Fi 网络。

(3) 通过 USB 或光缆将 iOS 设备连接到 Mac 设备上。

(4) 使用 Build & Run 选项构建应用程序。

(5) 打开 Unity 编辑器的 Profiler 窗口，并从 Connected Player 中选择设备。

现在，应该看到 Profiler 窗口中正在收集 iOS 设备的分析数据。

提示：Profiler 使用 54998～55511 端口来广播分析数据。如果系统存在防火墙，请确保这些端口可用于向外发送数据。

为了解决构建 iOS 应用程序并将 Profiler 连接到它们时所出现的问题，可以查阅参考网站 1.1。

4. 远程连接到 Android 设备

有两种不同的方法可以将 Android 设备连接到 Unity Profiler：通过 Wi-Fi 连接或使用 Android Debug Bridge(ADB)工具。这两种方法在 Apple Mac 或 Windows PC 上都可用。

第一种方法是通过 Wi-Fi 连接 Android 设备，执行下面的步骤：

(1) 确保构建应用程序时启用了 Development Build 和 Autoconnect Profiler 标志。

(2) 将 Android 和桌面设备(Apple Mac 或 Windows PC)连接到本地 Wi-Fi 网络。

(3) 通过 USB 线将 Android 连接到桌面设备。

(4) 使用 Build & Run 选项构建应用程序。

(5) 在 Unity 编辑器中打开 Profiler，在 Connected Player 下选择设备。

接下来，应该会构建应用程序，并通过 USB 连接推送到 Android 设备，而 Profiler 应该会通过 Wi-Fi 进行连接。之后应该看到 Profiler 窗口正在收集 Android 设备的分析数据。

第二种方法是使用 ADB。这是 Android SDK 中的调试工具套件。为了使用 ADB 进行分析，可以执行下面的步骤：

(1) 确保根据 Unity 的 Android SDK/NDK 安装向导安装了 Android SDK，可以查询参考网站 1.2。

(2) 通过 USB 线将 Android 设备连接到桌面设备。

(3) 确保构建应用程序时启用了 Development Build 和 Autoconnect Profiler 标志。

(4) 使用 Build & Run 选项构建应用程序。

(5) 在 Unity 编辑器中打开 Profiler，并在 Connected Player 下选择设备。

现在，应该看到 Profiler 窗口正在收集 Android 设备的分析数据。

为了解决构建 Android 应用程序并将 Profiler 连接到它们时所出现的问题，可以查阅参考网站 1.3。

5. 编辑器分析

可以分析编辑器自身，通常在分析自定义的编辑器脚本时这样做。为此，可以启用 Profiler 窗口的 Editor 选项，并设置 Connected Player 为 Editor，如图 1-3 所示。

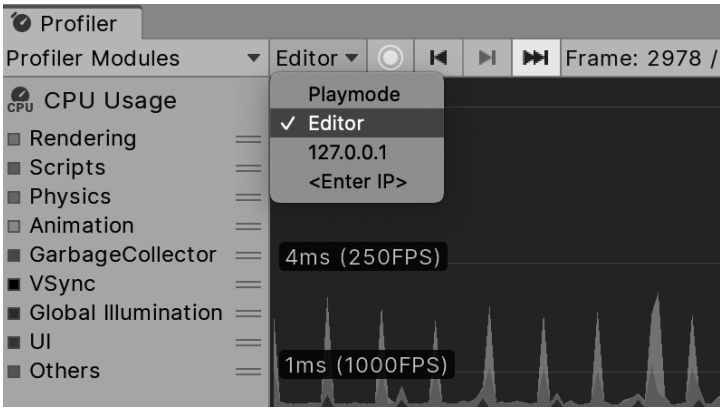

图 1-3 分析自定义编辑器脚本的性能

请注意，分析编辑器时，如果图中没有任何变化，那么可能是没有选择 Profiler 窗口的 Editor 选项，或者可能意外地连接到另一个游戏构建程序。

1.1.2 Profiler 窗口

Profiler 窗口可分成 4 部分，如图 1-4 所示。

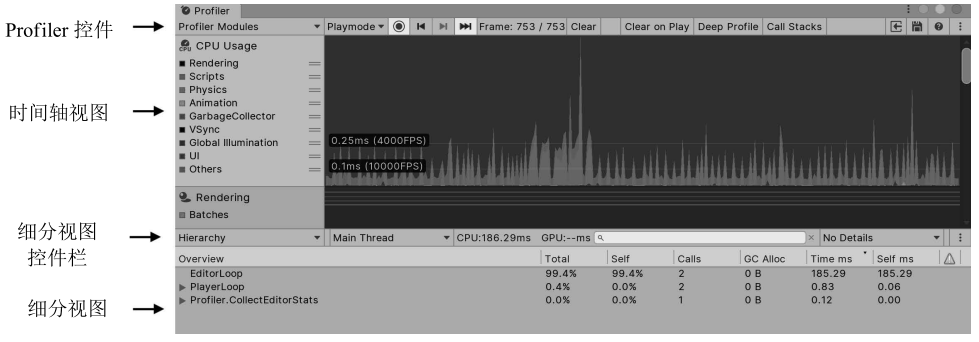

图 1-4 Profiler 窗口的组成部分

- Profiler 控件。
- 时间轴视图。

- 细分视图控件栏。
- 细分视图。

提示：时间轴视图有很多颜色，但不是每个人都能正常看到颜色。幸好，Unity 考虑到了人群中的色盲患者。在右上方的汉堡包菜单中，可以启用 Color Blind Mode，如图 1-5 所示。

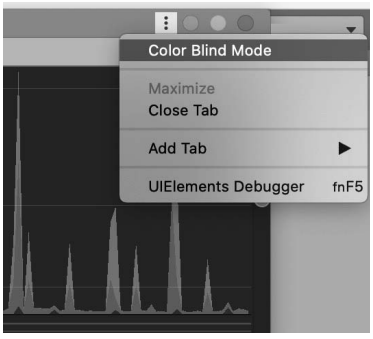

图 1-5　汉堡包菜单

1. Profiler 控件

图 1-4 所示窗口顶部的选项栏包括多个下拉菜单和开关按钮，它们可用于决定要分析什么数据，以及在每个子系统中收集数据的深度。这些内容将在稍后介绍。

1) Profiler Modules

默认情况下，Profiler 将为几个不同的子系统收集数据，这些子系统覆盖了Unity引擎在时间轴视图中的大部分子系统。这些子系统被组织成包含相关数据的各个区域。Profiler Modules 选项可以用来添加额外的区域，或者在它们被删除时恢复它们。请参考“时间轴视图”的内容，以获得可以分析的子系统的完整列表。

2) Playmode

Playmode 下拉菜单允许选择想要分析的 Unity 目标实例。该实例可以是当前编辑器应用程序、应用程序的本地独立实例，或者在远程设备上运行的应用程序实例。

3) 录制按钮

单击 Playmode 右边的录制按钮(双圆圈按钮)可以启用录制选项，从而让 Profiler 记录所分析的数据。启用此选项时，将连续记录数据。注意，只有在应用程序运行的情况下，才记录运行时数据。对于在编辑器中运行的应用程序，这意味着必须启用 Playmode，且不能暂停。另外，对于独立的应用程序，它必须是活动的窗口。如果启用了 Profile Editor，那么所显示的是为编辑器收集的数据。

4) Deep Profile

普通的分析只记录常见的 Unity 回调方法(如 Awake()、Start()、Update()和 FixedUpdate())所返回的时间和内存分配信息。启用 Deep Profile 选项可以用更深层次的指令重新编译脚本，允许它统计每个调用的方法。这将导致运行时的指令注入成本比正常情况下要大得多，并需要使用大量的内存，因为运行时收集的是整个调用堆栈的数据。因此，在大型项目中，Deep Profile 甚至是不可能的，因为 Unity 可能在测试开始之前就用完了内存，或者应用程序运行得太慢，以至于测试变得毫无意义。

提示：切换 Deep Profile 需要重新编译整个项目，才能再次开始分析，因此最好避免在测试中来回切换该选项。

由于 Deep Profile 选项无差别地统计整个调用栈，因此在大多数分析测试中启用这个选项是不明智的。当默认的分析选项无法提供足够的详情以指出问题根源时，最好保留这个选项；测试一个小场景的性能时，可以使用这个选项来隔离某种行为。

如果较大的项目和场景需要 Deep Profile，但 Deep Profile 选项在运行时会阻碍性能，那么可以使用其他方法来进行更详细的分析，详见 1.2.7 节“代码片段的针对性分析”。

5) Call Stacks

激活 Call Stacks 选项，Unity Profiler 在不启用 Deep Profile 的情况下也能更详细地收集游戏中相应的内存分配的信息，如图 1-6 所示。

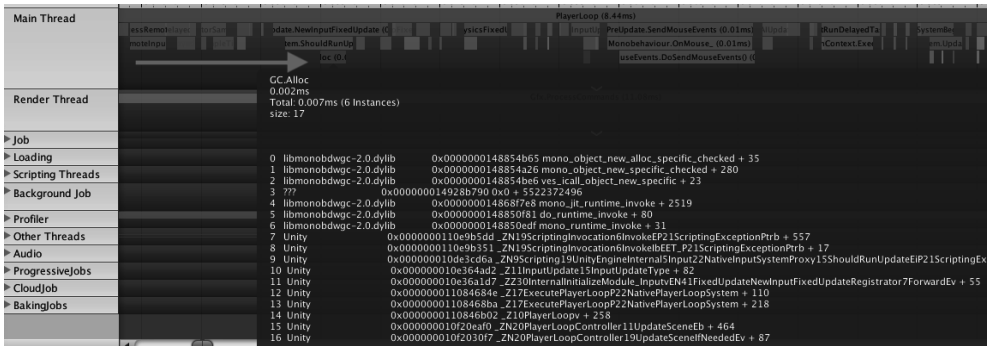

图 1-6 收集游戏内存分配的信息

如果启用该选项，就可以单击代表内存分配的红色框，Profiler 将显示内存分配的起源和原因，如图 1-7 所示。

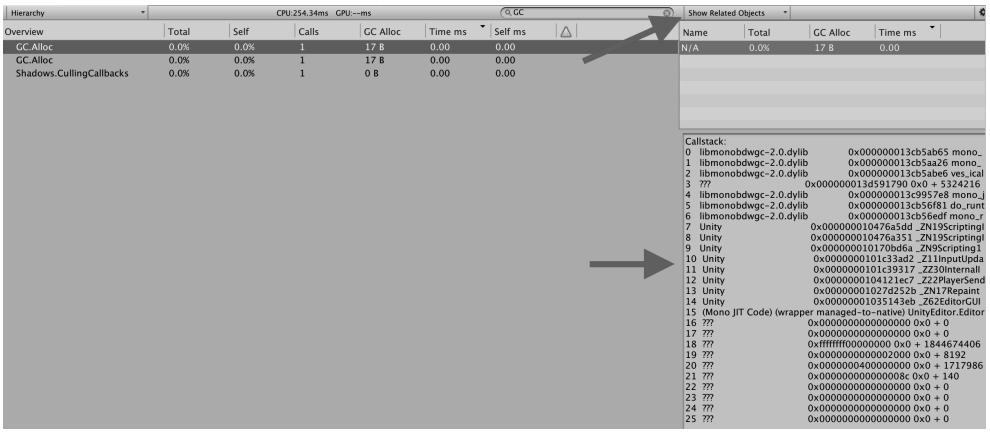

图 1-7　内存分配的起源和原因

相反，在 Hierarchy 视图中，仍然需要选择分配调用。然后在右上角的下拉菜单中切换到 Show Related Objects，选择其中一个 N/A 对象。在此之后，可以在下面的列表中看到调用栈信息。

第 8 章“掌握内存管理”将更多地讨论内存分配问题。

注意：在 Unity 2019.1 中，Allocation Callstack 只有在编辑器中进行分析时才能工作。

6) Clear

选择 Clear 选项，清除时间轴视图中所有的分析数据。

7) 加载按钮

单击右上角的加载按钮打开对话框窗口，加载任何之前(使用 Save 选项)保存的分析数据。

8) 保存按钮

单击右上角的保存按钮，可以将当前显示在时间轴视图上的所有 Profiler 数据保存到文件中。这种方式一次只能保存 300 帧数据，要保存更多数据，必须手动创建新文件。大多数情况下，这通常是足够的，因为当性能峰值出现时，有 5～10 秒的时间暂停应用程序并保存数据，以供后续分析 (例如，将它附着到 bug 报告上)，之后把它推到时间轴视图的左边。任何已保存的 Profiler 数据都可使用 Load 选项加载到 Profiler 中，以供将来检查。

9) 帧选择

帧选择区域由几个子元素组成。Frame Counter 显示已经分析了多少帧以及在当前时间轴视图中选中了哪一帧。有两个选项用于将当前选中的帧向前或向后移动一帧，而另一个选项(Current)将选中的帧重置为最新的帧并保持最新。这样，

细分视图在运行分析期间总是显示当前帧的分析数据，并显示 Current 字样。

2. 时间轴视图

时间轴视图显示运行期间的数据，如图 1-8 所示。

图 1-8 的左半部分所示为一系列用于启用/禁用不同行为/数据类型的复选框(彩色方块)，右半部分是分析数据的图形化展示。

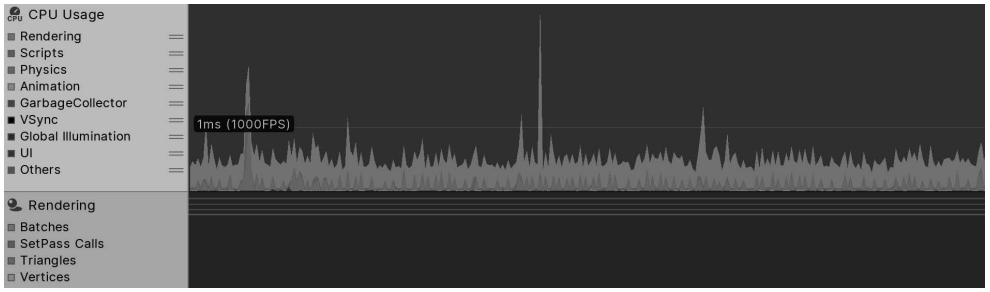

图 1-8 时间轴视图

彩色的复选框可以切换，以改变时间轴视图的图形部分内对应数据类型的可见性。

在时间轴视图中选中一个区域时，在当前选中帧的细分视图中(时间轴视图下方)会显示子系统的详细信息。根据在时间轴视图中选中区域的不同，细分视图将显示不同类型的信息。

单击区域右上角的×按钮可以将区域从时间轴视图中移除。通过控件栏中的 Add Profiler 可以将区域重新添加到时间轴视图中。

任何时候都可以单击时间轴视图的图形部分，查看给定帧的更多信息。此时会显示一个大的垂直白条(通常在线图的两边会附带一些额外的信息)，说明选中了哪一帧。

根据当前选中区域(该区域是高亮蓝色的)的不同，细分视图中会显示不同的信息，细分视图控件栏中也会有不同的选项。要改变选中的区域，只需要选中时间轴视图左边的复选框或单击图形区域，但是在图形区域内单击也可能会改变当前选中的帧，因此，如果期望看到同一帧的细分视图信息，单击图形区域时须小心。

3. 细分视图控件栏

根据时间轴上选中区域的不同，细分视图控件栏内将显示不同的下拉列表框和切换按钮选项。不同区域提供不同的控件，这些选项声明了什么信息是可见的，以及这些信息如何呈现在细分视图中。

4. 细分视图

根据当前选择的区域以及在细分视图控件栏中选择的选项的不同，细分视图

中显示的信息会有很大的不同。例如，一些区域在细分视图控件栏的下拉列表框中提供不同的模式，以显示信息的简单视图或详细视图，甚至同一信息的图形布局，这样分析起来就更容易。

接下来介绍细分视图的每个区域，以及细分视图中不同类型的可用信息和选项。

1) CPU Usage 区域

这个区域显示 CPU 的使用情况和统计数据。该区域可能是最复杂、最有用的区域，因为它包括 Unity 的大量子系统，诸如 MonoBehaviour 组件、摄像机、一些渲染和物理处理、用户界面(如果通过编辑器运行，还包括编辑器的界面)、音频处理、Profiler 等。

在细分视图中，显示 CPU 使用情况数据的模式有 3 种： Hierarchy 模式、Raw Hierarchy 模式和 Timeline 模式。

(1) Hierarchy 模式显示大部分调用栈的调用，为了方便起见，还会合并类似的数据元素和 Unity 的全局函数调用。例如渲染分隔符，如 BeginGUI()和 EndGUI()调用在这个模式中合并到了一起。Hierarchy 模式有助于决定执行哪个函数调用会花费最多的 CPU 时间。

(2) Raw Hierarchy 模式和 Hierarchy 模式类似，但前者会将全局 Unity 函数调用隔离到单独的条目中，而不是合并到一个大条目中，这将使细分视图更加难以阅读。如果尝试统计某个全局方法调用了多少次，或者确认这些调用中的某次调用比预计消耗了更多的 CPU 和内存，Raw Hierarchy 模式会很有用。例如，Raw Hierarchy 模式中，每个 BeginGUI()和 EndGUI()调用都会放在不同的条目中，与 Hierarchy 模式相比，这样能更清楚每个函数被调用了多少次。

Timeline 模式(不要和主 Timeline 视图混淆)可能是 CPU Usage 区域最有用的模式。Timeline 模式根据处理期间调用栈的展开和收缩方式，组织当前帧的 CPU 使用情况信息。

(3) Timeline 模式将细分视图在垂直方向上安排到不同的部分，代表运行时的不同线程，例如主线程、渲染线程和各种后台工作线程，称为 Unity Job System，用于加载诸如场景和其他资源等活动。水平轴表示时间，所以宽方块消耗的 CPU 时间比窄方块多，方块的宽窄也表示相对时间，更容易比较两个调用所消耗的时间。垂直轴代表调用栈，因此更深的链表示在那个栈上有更多调用。在 Timeline 模式下，细分视图顶部的方块是由 Unity 引擎在运行时调用的函数(从技术上说是回调)，例如 Start()、Awake()或 Update()，而下面的方块是函数里面调用的函数，可以是其他组件上的函数或常规 C#对象。

Timeline 模式提供了一种非常清晰、条理分明的方式，以明确调用栈中的哪个方法消耗的时间最多，以及如何与同一帧中调用的其他方法所消耗的处理时间进行比较。这可以花费最少的精力来评估导致性能问题的最大原因。

例如，查看图 1-9 所示性能数据视图的性能问题。快速浏览一下就可以发现，有 3 个方法导致出现问题，它们消耗的处理时间类似，因为它们的宽度类似。

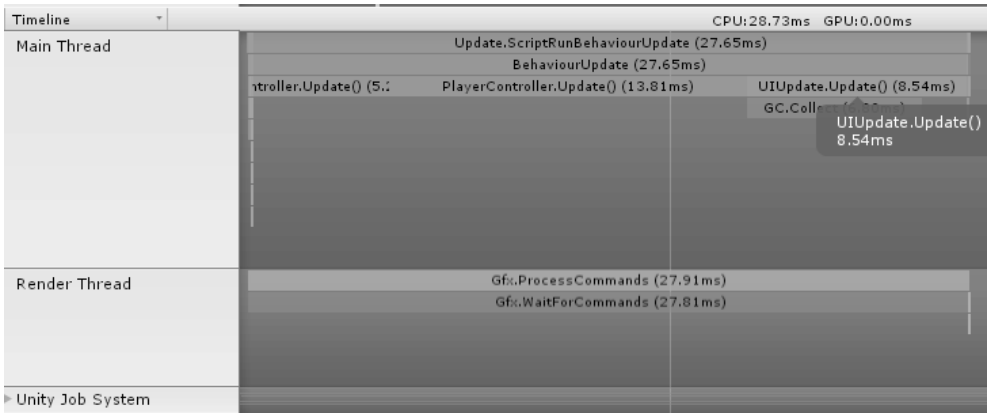

图 1-9 Timeline 模式的性能数据视图

图 1-9 中，调用了 3 个不同的 MonoBehaviour 组件，超出了 16.667 毫秒的预算。好消息是，有 3 种可能的方法可以改进性能，这意味着有很多机会找到可以改进的代码。坏消息是，提高一个方法的性能，只会提升那一帧整体处理效率的 1/3。因此，可能需要检查和优化这 3 个方法，以使处理时间在预算之内。

提示：使用 Timeline 模式时，最好折叠 Unity Job System 列表，因为它妨碍了 Main Thread 块中各项的显示，而 Main Thread 块可能是读者最感兴趣的内容。

通常，CPU Usage 区域有助于检测问题的解决，这些问题能通过第 2 章中研究的方案来解决。

2) GPU Usage 区域

GPU Usage 区域和 CPU Usage 区域类似，但前者展示的是发生在 GPU 上的方法调用和处理时间。这个区域中的 Unity 方法调用与摄像机、绘制、不透明的和透明的几何图形、光照和阴影等有关。

GPU Usage 区域提供了类似 CPU Usage 区域的层级信息，并估计调用各种渲染函数(如Camera.Render())所花费的时间(假定渲染是在当前时间轴视图中选择的帧期间发生的)。

阅读第 6 章“动态图形”时，会发现 GPU Usage 区域是很有用的工具。

3) 渲染区域

渲染区域提供了一些常用的渲染统计数据，关注 GPU 为渲染而准备的相关活动，这一系列活动发生在 CPU 上(与渲染行为不同，渲染是在 GPU 内部处理的，在 GPU Usage 区域显示详情)。细分视图提供了有用的信息，诸如 SetPass 调用的

数量(也称为 Draw Call)、渲染到场景的批次总数、通过动态批处理和静态批处理节省的批次数量及其生成方式，以及纹理的内存消耗。

渲染区域通常提供一个按钮，用于打开 Frame Debugger，详见第 3 章“批处理的优势”。阅读第 3 章和第 6 章后，会发现这个区域的其他信息很有用。

4) 内存区域

内存区域允许在细分视图中以 Simple 模式和 Detailed 模式检视应用程序的内存使用情况。

Simple 模式只提供子系统内存消耗的高层次概览，包括 Unity 底层引擎、Mono 框架(由垃圾回收器管理的整个堆的大小)、图形资源、音频资源、缓冲区，甚至用于保存 Profiler 收集的数据的内存。

Detailed 模式显示每个 GameObjects 和 MonoBehaviours 为其 Native 和 Managed 表示所消耗的内存。该模式还有一列，解释为什么对象可能消耗内存以及它可能何时被销毁。

注意：垃圾回收器是由 C# (Unity 选择的脚本语言)提供的一个通用特性，它自动释放为存储数据而分配的内存，但如果处理不好，有可能使应用程序出现短时间卡顿。这个话题，以及与本地和托管内存空间相关的更多话题，参见第 8 章“掌握内存管理”。

注意，单击 Take Sample < TargetName >按钮(TargetName 为当前进行性能分析的目标，这里当前为调试 Editor 模式下的性能，因此图中显示为 Take Sample Editor。如果当前正在调试 Playmode，那么该按钮会显示为 Take Sample Playmode)，信息仅在 Detailed 模式下通过手动采样来显示，如图 1-10 所示。这是 Detailed 模式下收集信息的唯一方法，因为每次更新时自动执行这种分析的代价非常高昂。

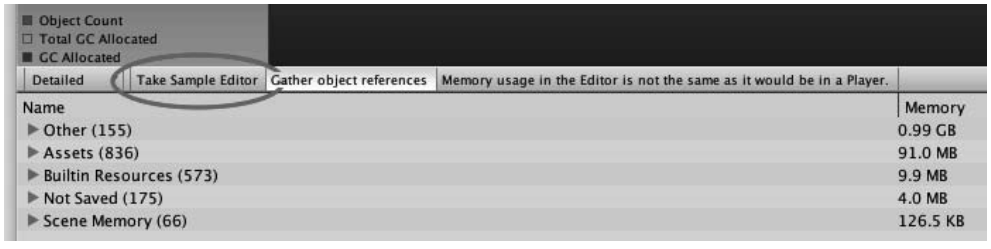

图 1-10　内存模块细分视图下的详细模式

细分视图通常提供标签为 Gather object references 的按钮，它可以收集一些对象更深层的内存信息。

第 8 章在深入学习内存管理的复杂性，以及 Native 和 Managed 的内存与垃圾

回收器时，内存区域会是有用的工具。

5) 音频区域

音频区域是音频统计数据的预览，也可以用于估量音频系统的 CPU 消耗，以及所有播放中或暂停的音源和音频剪辑的总内存消耗。

细分视图提供了很多有用信息，可以洞悉音频系统的运行方式，以及各种音频通道和组的用法。

音频区域在第 4 章讨论艺术资源时会很有用。

提示：进行性能优化时，音频通常被忽略，但如果没有正确管理，音频可能会成为性能瓶颈的重要来源，因为音频需要大量潜在的磁盘访问和 CPU 处理。不要忽略它！

6) Physics 3D 和 Physics 2D 区域

有两个不同的物理区域：Physics 3D(NVIDIA 的 PhysX)和 Physics 2D(Box2D)。这两个区域提供不同的物理统计数据，例如 Rigidbody、Collider 和 Contact 计数。

每个物理区域的细分视图为子系统的内部工作情况提供了一些基本信息，但通过浏览 Physics Debugger 可以更深入地洞悉问题，详见第 5 章“加速物理引擎”。

7) 网络消息区域和网络操作区域

这两个区域提供了 Unity 网络系统的信息，该系统是在 Unity 5 发布时引入的。所显示的信息取决于应用程序是否使用 Unity 提供的高级 API(HLAPI)或传输层 API(TLAPI)。HLAPI 是一个更易用的系统，用于管理 Player 和 GameObject 自动网络同步，而 TLAPI 只是套接字层级上操作的一个薄层，它允许 Unity 开发者构建自己的网络系统。

网络拥塞优化是一个可以写一整本书的主题，而正确的解决方案通常非常依赖应用程序的特定需求。这不是 Unity 特有的问题，因此本书不介绍网络拥塞优化的主题。

8) 视频区域

如果应用程序使用 Unity 的 VideoPlayer API，就会发现这个区域对于分析视频回放行为很有效果。

媒体回放优化也是一个很复杂的非 Unity 特有的问题，本书不予讨论。

9) UI 和 UI 详情区域

这些区域用于洞察使用 Unity 内建 UI 系统的应用程序。如果使用自定义生成的或第三方的 UI 系统(例如 NGUI)，这些区域可能就没什么用处了。

不够优化的 UI 通常会影响 CPU 和/或 GPU，因此第 2 章会提出 UI 的一些代码优化策略，第 6 章则介绍一些图形相关方法。

10) 全局光照区域

全局光照(Global Illumination，GI)区域为 Unity 的全局光照系统提供了大量优秀的细节。如果应用程序使用 GI，就应该参考这个区域，以验证应用程序是否正常执行。

这个区域在第 6 章研究光照和阴影时会很有用。

1.2　性能分析的最佳方法

通常，良好的代码实践和项目资源管理可以使性能问题的根源查找变得相对简单，而唯一的真正问题是要弄清楚如何改进代码。例如，如果方法只处理一个巨大的 for 循环，则可以安全地假定性能问题的出现在于循环迭代了太多次，或者循环由于以非顺序的方式读取内存导致缓存丢失，或者每次循环做了太多工作，或者循环为准备下一次迭代做了太多工作。

当然，无论是单独工作还是在团队中工作，许多代码并不总是以最干净的方式编写，需要经常分析较差的代码。有时被迫为了速度而采用一个笨拙的解决方案，因为开发人员经常没有时间去重构所有代码，以遵循最佳编码方式。实际上，以性能优化的名义所做的许多代码更改往往显得非常奇怪或晦涩难懂，常使代码库更难阅读。软件开发的共同目标是代码简洁、功能丰富、运行速度快。实现其中一个目标相对容易，但实现两个目标将花费更多的时间和精力，而实现 3 个目标几乎是不可能的。

在最基本的层面上，性能优化只是解决问题的另一种形式，而在解决问题时忽略明显的代码问题可能是一个代价高昂的错误。性能优化的目标是使用基准分析来观察应用程序，寻找问题行为的实例，然后使用指令注入工具在代码中寻找关于问题根源的线索。遗憾的是，开发人员常常很容易被无效的数据分散注意力，或者因为缺乏耐心或忽略了某个小细节而匆忙得出结论。许多人在软件调试过程中都遇到过这样的情况：如果简单地质疑并验证前面的假设，就可以更快地找到问题的根源。查找性能问题也一样。

解决问题之前应先列一份任务清单以帮助开发人员专注于这个问题，并确保不会浪费时间来尝试实现任何可能的优化，而这些优化对主要的性能瓶颈没有影响。当然，每个项目都是不同的，有自己独特的困难需要克服，但是以下检查列表适用于任何 Unity 项目：

- 验证目标脚本是否存在于场景中；
- 验证脚本在场景中出现的次数是否正确；
- 验证事件的正确顺序；
- 最小化正在进行的代码更改；

- 最小化内部影响；
- 最小化外部影响。

1.2.1 验证脚本是否存在

有时，人们期待看到有些东西，但却没有看到(比如，希望在鱼缸中看到金鱼在游动，但鱼缸里根本没有金鱼)。这些东西通常很容易觉察到，因为人类的大脑非常擅长模式识别，从而发现我们意料不到的差异。有时，人们会假设某些事情已经发生了，但实际上并未发生(比如，我们希望在单击图标后应用程序就会运行，但其实它根本没运行)。这些事情通常难以注意到，因为人们经常碰到的是第一类问题，会假设没有看到的东西会按预期工作(比如，应用程序正常运行时没有任何提示，仅在出错时才有警告或弹窗)。在 Unity 的环境中，有一个问题就是以这种方式表现出来的，那就是验证期望操作的脚本是否确实存在于场景中。

为了快速验证脚本是否存在，可以在 Hierarchy 窗口的文本框中输入以下内容：

```
t:<monobehaviour name>
```

例如，在 Hierarchy 文本框中输入 t:mytestmonobehaviour(注意，不区分大小写)，将显示一个包含 GameObject 的短列表，该列表当前至少有一个 MyTestMonoBehaviour 脚本作为组件。

提示：这个短列表还包括其组件从给定脚本名称派生的 GameObjects。

还应该再次检查它们所连接的 GameObjects 是否仍然处于激活状态，因为可能在之前的测试中意外地停用了该对象而禁用了它们。

1.2.2 验证脚本次数

如果查看 Profiler 数据时，注意到某个 MonoBehaviour 方法执行的次数比预期的多，或者执行的时间比预期的长，就可能需要再次检查它在场景中出现的次数是否与预期的一样多。有人在场景文件中创建对象的次数比预期多，或者意外地在代码中实例化对象的次数比预期多是完全有可能的。如果是这样，问题可能源于调用了冲突或重复的方法，产生了性能瓶颈。可以使用 1.2 节“性能分析的最佳方法”中的短列表方法来验证计数。

如果场景中期望出现特定数量的组件，但是短列表显示的组件数比期望的更多(或更少)，最好编写一些初始化代码来防止这种情况再次发生，也可以编写一些自定义编辑器辅助函数，给可能犯这个错误的关卡设计师发出警告。

防止这样的偶然错误对提高工作效率至关重要，因为经验表明，如果未明确声明不允许做某件事，那么无论出于什么原因，某个地方的某个人就会在某个时候做这件事。这可能会导致花费一个下午去查找问题，而最终发现是人为错误导致的。

1.2.3　验证事件的顺序

Unity应用程序主要执行从本机代码到托管代码的一系列回调。这一概念详见第 8 章，这里仅简要讨论，Unity的主线程并不像简单的控制台应用程序那样运行。在 Unity 应用程序中，代码执行时有明显的起点(通常是main()函数)，然后直接控制游戏引擎，初始化主要子系统，接着游戏运行在一个很大的 while 循环(通常称为游戏循环)中，检查用户输入，更新游戏，渲染当前的场景，重复下去。此循环只在玩家选择退出游戏时退出。

相反，Unity 负责处理游戏循环，开发人员期望在特定时刻调用诸如 Awake()、Start()、Update()和 FixedUpdate()等回调。最大的区别在于，不能对调用相同类型事件的顺序进行细粒度控制。当加载一个新场景时(无论是游戏的第一个场景，还是之后的场景)，都会调用每个 MonoBehaviour 组件的 Awake()回调，但是无法确定调用的顺序。

如果一组对象在 Awake()回调中配置一些数据，另一组对象在自己的 Awake()回调中对这些已配置数据执行一些处理，则场景对象的一些重组或重建，代码库和编译过程中的一个随机变化(还不清楚究竟是什么原因)就可能导致这些 Awake()调用的顺序发生改变，然后依赖对象可能尝试对未按预期方式初始化的数据进行处理。MonoBehaviour 组件提供的所有其他回调也是如此，如 Start()和 Update()。

在任何足够复杂的项目中，都无法确定在一组 MonoBehaviour 组件中调用同类型回调的顺序，所以要非常小心，不要假设对象回调是以特定的顺序发生的。实际上，编写代码时，永远不要假定这些回调需要以某种顺序来调用，因为它可能在任何时候中断。

处理后期初始化的一个更好的方法是调用 MonoBehaviour 组件的 Start()回调，它总是在每个对象的 Awake()之后，第一个 Update()之前被调用。后期更新也可以在 LateUpdate()回调中完成。

如果在确定事件的实际顺序时遇到困难，最好使用带 IDE (MonoDevelop、Visual Studio 等)的逐步调试器来处理，或者使用 Debug.Log()输出简单的日志语句。

提示：Unity 的日志器非常昂贵。日志不太可能改变回调的顺序，但如果使用得太频繁，可能会导致一些不必要的性能峰值，更好的做法是只对代码库中最相关的部分进行有针对性的日志记录。

协程通常用于编写一些事件序列的脚本，它们何时触发取决于所使用的 yield 类型。最难调试、最不可预测的类型可能是 WaitForSeconds yield 类型。Unity 引擎是不确定的，这意味着即使在相同的硬件上，一个会话和下一个会话中的行为也会稍微不同。例如，在一个会话中，可能会在应用程序运行的第一秒内调用 60 个更新，在下一秒中调用 59 个更新，在之后的一秒中调用 62 个更新。在另一个会话中，可能在第一秒内调用 61 个更新，在第二秒中调用 60 个，在第三秒中调用 59 个。

在协程启动和结束之间调用的 Update()回调数量是可变的，因此，如果协程依赖于某个对象的 Update()特定调用次数，就会出问题。一旦协程启动，最好保持它的简单性和独立性，不受其他行为的影响。若违反这条规则，将来的一些更改肯定会以意想不到的方式与协程交互，从而导致调试过程漫长而痛苦，因为在这期间需要纠正难以重现的会破坏游戏的错误。

1.2.4 最小化正在进行的代码更改

为了查找性能问题而对应用程序进行代码更改最好谨慎进行，因为随着时间的推移，更改很容易被忘记。向代码中添加调试日志语句可能很诱人，但是请记住，引入这些调用，重新编译代码并在分析完成后删除这些调用需要花费大量时间。此外，如果忘记删除它们，它们可能会在最终的构建版本中消耗不必要的运行时开销，因为 Unity 的调试控制台窗口日志记录会占用较多的 CPU 和内存资源。

解决这个问题的一个好方法是在做了更改的地方添加一个自己命名的标记或注释，以便以后很容易找到并删除它。还可以使用源代码控制工具，使代码库易于区分任何修改过的文件的内容，并将它们恢复到原始状态则能很好地确保不必要的修改不会进入最终版本。当然，如果同时应用了一个修复程序，但在提交更改之前没有对所有修改过的文件进行复查，则该解决方案不能确保以上情况发生。

在调试期间使用断点是首选方法，因为此时可以跟踪完整的调用栈、变量数据和条件代码路径(例如 if-else 块)，而没有任何更改代码的风险，也不会在重新编译上浪费时间。当然，这个方法并不总是可行。例如，试图确定在 1000 帧的某一帧中是什么导致奇怪的事情发生了，最好确定要查找的阈值，并添加包含断点的 if 语句，当该值超过阈值时将触发该语句。

1.2.5 最小化内部影响

Unity 编辑器有其独有的特点，这有时会使调试某些类型的问题变得混乱。

首先，如果处理一帧需要很长时间，比如游戏出现明显的卡顿，那么 Profiler 可能无法获取结果并记录在 Profiler 窗口中。如果希望在应用程序或场景的初始化期间捕获数据会很麻烦，后面的自定义 CPU 分析的相关内容将提供一些解决此

问题的备选方案。

如果试图通过按键启动测试，且已经打开了 Profiler，则在按键之前，不要忘了单击回到编辑器的 Game 窗口。如果 Profiler 是最近单击的窗口，那么编辑器将击键事件发送到该窗口，而不是运行的应用程序，因此没有 GameObject 会捕获该击键事件。这也会影响 GameView 的渲染任务，甚至 WaitForEndOfFrame yield 类型的协程。如果 Game 窗口在编辑器中不可见但处于活动状态，则不会向该视图渲染任何内容，因此不会触发依赖 Game 窗口渲染的事件。

垂直同步(或称为VSync)用于将应用程序的帧率匹配显示器的帧率，例如显示器的帧率可能是 60Hz(每秒 60 个循环，每个循环约16ms)，而如果游戏的渲染循环比这个速度快(例如 10ms)，游戏就会等待(6ms)，直到输出渲染的帧为止。该特性减少了屏幕撕裂，即在前一幅图像完成之前，就将新图像推送到显示器，于是新图像的一部分会在短时间内与旧图像重叠。

执行启用了 VSync 的 Profiler 可能会在 WaitForTargetFPS 标题下的 CPU Usage 区域中产生许多嘈杂的峰值，因为应用程序故意降低速度，以匹配显示器的帧率。这些峰值通常在编辑器模式下显得非常大，因为编辑器通常渲染到一个非常小的窗口上，这并不需要很多 CPU 或 GPU 工作来渲染。

这将产生不必要的混乱，导致更难发现真正的问题。在性能测试期间监视 CPU 峰值时，应该确保在 CPU Usage 区域下禁用 VSync 复选框。选择 Edit | Project Settings | Quality 命名，然后打开当前选择的平台的子页面，就可以完全禁用 VSync 功能。

还应该确保性能下降不是编辑器控制台窗口中出现了大量异常和错误消息而导致的直接结果。Unity 的 Debug.Log()和类似的方法，如 Debug.LogError()和 Debug.LogWarning()，在 CPU 使用率和堆内存方面消耗非常大，这会导致发生垃圾回收，甚至丢失 CPU 循环(详见第 8 章)。

对于以编辑器模式查看项目的人员来说，这种开销通常是不明显的，因为在编辑器模式中，大多数错误来自编译器或配置错误的对象。然而，在任何类型的运行时过程中使用它们都可能会有问题，特别是在分析期间，希望在没有外部中断的情况下观察游戏如何运行。例如，如果丢失了一个通过编辑器分配的对象引用，但它在 Update()回调中使用，那么一个 MonoBehaviour 就会在每次更新时抛出新的异常。这给数据的分析增加了很多不必要的干扰。

注意，可以使用图 1-11 所示的按钮隐藏不同的日志级别类型。额外的日志记录即使没有显示出来，也需要 CPU 和内存来执行，但允许过滤掉不想要的垃圾。尽管如此，启用所有这些选项通常是一个很好的实践，可以验证有没有遗漏任何重要的内容。

图 1-11 控制不同的日志级别的按钮

1.2.6 最小化外部影响

再次确认没有后台进程消耗 CPU 周期或占用大量内存很简单，但绝对必要。可用内存不足通常会干扰测试，因为会导致更多的缓存丢失，对虚拟内存页交换文件的硬盘访问，应用程序的响应速度通常较慢。如果应用程序突然表现得比预期糟糕得多，请再次检查系统的任务管理器(或等效的其他程序)是否有任何 CPU、内存、硬盘活动，这可能会导致某些问题。

1.2.7 代码片段的针对性分析

如果性能问题没有通过前面提到的检查表解决，则可能面临一个需要进一步分析的实际问题。Profiler 窗口可以有效地展示性能的大致概况，帮助找到需要调查的特定帧，并快速确定哪个 MonoBehaviour 和/或方法导致了问题。然后，需要确定问题是否会重现，在什么情况下出现性能瓶颈，以及问题代码块中问题的确切来源。

为了完成这项任务，可以采用一些有用的技术，对代码的目标部分执行一些分析。对于 Unity 项目，它们基本上可分为两类：

- 在脚本代码中控制 Profiler；
- 自定义定时和日志记录方法。

注意：本小节的重点是如何通过 C#代码调查脚本瓶颈。如何检测其他引擎子系统中瓶颈的来源将在相关章节中讨论。

1. Profiler 脚本控制

可以通过 Profiler 类在脚本代码中控制 Profiler。这个类有几个有用的方法，可以在 Unity 文档中使用它们，其中最重要的方法是在运行时激活和禁用分析功能的分隔符方法。要在脚本代码中控制 Profiler，可以通过 UnityEngine.Profiling.Profiler 类的 BeginSample()和 EndSample()方法来实现。

提示：分隔符方法 BeginSample()和 EndSample()仅在开发构建过程中编译，因此，它们不会在未选中开发模式的版本构建过程中编译或执行。这通常称为非操作代码。

BeginSample()方法有一个重载版本，允许样本的自定义名称出现在 CPU Usage 区域的 Hierarchy 模式中。例如，下面的代码将分析此方法的调用，并使数据出现在自定义标题下的细分视图中，如下所示：

```
void DoSomethingCompletelyStupid() {
  Profiler.BeginSample("My Profiler Sample");
  List<int> listOfInts = new List<int>();
    listOfInts.Add(i);
  }
  Profiler.EndSample();
}
```

调用这个设计不良的方法(生成一个列表，其中包含一百万个整数，但之后完全不操作该列表)应该会导致一个巨大的 CPU 使用峰值，占用几兆字节内存，显示 My Profiler Sample 标题下的细分视图，如图 1-12 所示。

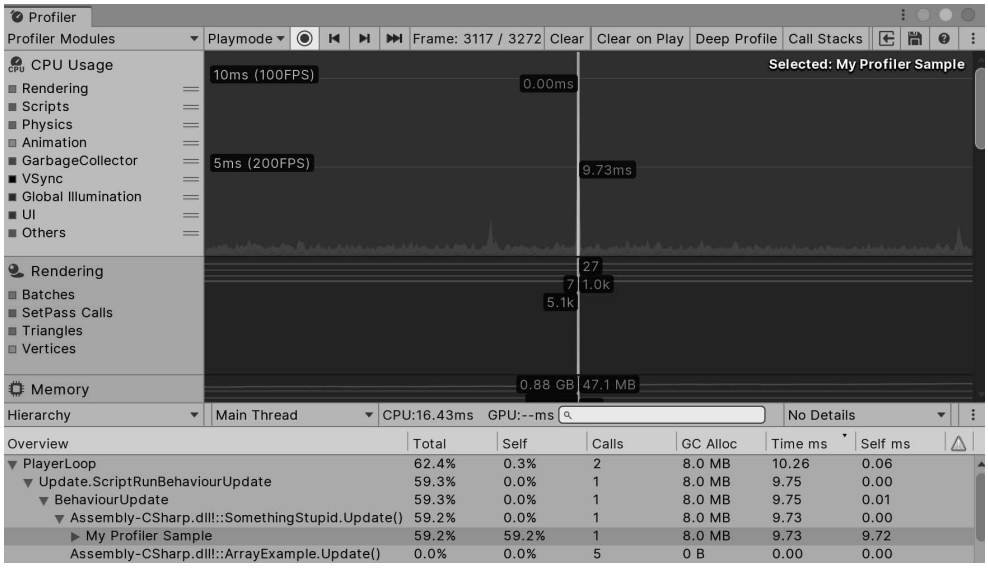

图 1-12　My Profiler Sample 标题下的细分视图

2. 自定义 CPU 分析

Profiler 只是可用的工具之一。有时开发人员可能希望对代码执行自定义的分析和日志记录，也许不确定 Unity Profiler 是否给出了正确的答案，也许认为它的开销太大了，或者只是想完全控制应用程序的每个方面。不管动机是什么，了解一些对代码进行独立分析的技术是一项有用的技能，毕竟不太可能在整个游戏开发生涯中都使用 Unity。

分析工具通常非常复杂，因此开发人员不太可能在合理的时间范围内自己生成一个类似的解决方案。在测试 CPU 使用情况时，真正需要的是一个准确的计时

系统，一种快速、低成本的信息记录方法，以及一些用于测试它们的代码。.NET 库(从技术上说，是 Mono 框架)在 System.Diagnostics 名称空间中提供了一个 Stopwatch 类，可以随时停止和启动 Stopwatch 对象，很容易度量自 Stopwatch 对象启动以来经过了多长时间。

遗憾的是，Stopwatch 类并不完全准确，它只能精确到毫秒，最多精确到 1/10 毫秒。使用 CPU 时钟计算高精度的实时时间是一项非常困难的任务，因此，下面尝试找到一种使 Stopwatch 类满足需求的方法。

精度很重要，提高精度的一个有效方法是多次运行相同的测试。假设测试代码块既容易重复又不是特别长，测试就应该能在一个合理的时间内运行成千上万次，甚至上百万次，然后用总消耗时间除以测试运行的次数，就得到单次测试的较准确的时间。

在进行提高精度的操作之前，应该先问问自己是否需要这样做。大多数游戏希望以 30FPS 或 60FPS 的速度运行，这意味着它们分别只有 33 毫秒或 16 毫秒的时间来计算整个帧的所有内容。因此，如果只需要将特定代码块的运行时间降低到 10 毫秒以下，那么为了获得微秒精度而重复数千次测试就太不值得了。

下面是一个自定义计时器的类定义，它使用 Stopwatch 计算给定测试次数的时间：

```
using System;
using System.Diagnostics;

public class CustomTimer : IDisposable {
   private string _timerName;
   private int _numTests;
   private Stopwatch _watch;

   // give the timer a name, and a count of the
   // number of tests we're running
   public CustomTimer( string timerName, int numTests) {
       _timerName = timerName;
       _numTests = numTests;
       if (_numTests <= 0) {
           _numTests = 1;
       }
       _watch = Stopwatch.StartNew();
   }

       // automatically called when the 'using()' block ends
       public void Dispose() {
       _watch.Stop();
       float ms = _watch.ElapsedMilliseconds;
       UnityEngine.Debug.Log( string.Format("{ 0} finished: {1: 0.00}
```

```
            " + "milliseconds total, {2: 0.000000} milliseconds per-test "
            + "for {3} tests", _timerName, ms, ms / _numTests, _numTests));
        }
    }
```

提示：在成员变量名之前添加下画线，可以区分类的成员变量(也称为字段)与方法的参数和局部变量，这是一种常用且有用的方法。

CustomTimer 类的用法示例如下：

```
const int numTests = 1000;
using(new CustomTimer("My Test", numTests)) {
    for(int i = 0; i < numTests; ++i) {
        TestFunction();
    }
}   // the timer's Dispose() method is automatically called here
```

在使用这种方法时，要注意以下三点。

第一，只是对多个方法调用时间进行平均。如果处理时间在不同的调用之间存在很大差异，那么就不能很好地在最终的平均值中表示出来。

第二，如果内存访问很常见，那么重复请求相同的内存块将导致人为提高缓存命中率(CPU 能很快找到内存中的数据，因为它保存在最近访问的同一区域)。与普通的调用相比，这将降低平均时间。

第三，由于类似的人为原因，实时(JIT)编译的效果被有效地隐藏起来，因为它只影响方法的第一次调用。JIT 编译是.NET 的功能之一，详见第 8 章。

using 块通常用于确保非托管资源在超出作用域时被正确销毁。当 using 块结束时，它将自动调用对象的 Dispose()方法来处理任何清理操作。为了实现这一点，对象必须实现 IDisposable 接口，这迫使它定义 Dispose()方法。

但是，可以根据相同的语言特性创建不同的代码块，该代码块创建一个短期对象，该对象在代码块结束时自动完成一些有用的操作，在前面的代码块中就是这样使用的。

提示：using 块不应与 using 语句相混淆，using 语句用于脚本文件的开头，以获取其他名称空间。注意，C#中管理名称空间的关键字与另一个关键字有命名冲突。

using 块和 CustomTimer 类提供了一种干净的方法来打包测试代码，这种方法清楚地指明了使用它的时间和场合。

另一个需要担心的问题是应用程序的预热时间。当场景启动时，如果需要从磁盘上加载大量数据，初始化复杂的子系统，如物理和渲染系统，在执行其他操

作之前需要解析大量的 Awake()和 Start()回调，则 Unity 的启动成本很大。这种早期的开销可能只持续一秒钟，但如果代码也在早期初始化期间执行，则会对测试结果产生重大影响。如果想要准确地测试，任何运行时测试都应该在应用程序达到稳定状态之后才开始，这一点至关重要。

理想情况下，可以在目标代码块的初始化完成后，在自己的场景中执行它。这并不总是可行的，因此作为备份计划，可以将目标代码块打包到 Input.GetKeyDown()中，以便在调用它时进行控制。例如，下面的代码只在按空格键时执行测试方法：

```
if(Input.GetKeyDown(KeyCode.Space)) {
    const int numTests = 1000;
    using(new CustomTimer("Controlled Test", numTests)) {
        for(int i = 0; i < numTests; i++) {
            TestFunction();
        }
    }
}
```

如前所述，Unity 的 Console 窗口日志记录机制的开销是非常昂贵的。因此，不应在分析测试中(或在进行游戏期间)使用这些日志方法。如果需要详细的分析数据，输出大量的各种消息(如在一个循环中执行计时测试，找出哪个迭代花的时间比其他迭代更多)，则明智之举是缓存日志数据，并在最后将它们全部输出，这与 CustomTimer 类相同。这将减少运行时开销，但会消耗一些内存。反之，如果在测试期间输出每个 Debug.Log()消息，则会导致每次耗费数毫秒的时间，从而影响测试结果。

CustomTimer 类还使用了 string.Format()方法。具体原因将在第 8 章“掌握内存管理”中介绍，简短的解释是因为直接使用+操作符(如代码 Debug.Log("Test: " + output);)生成自定义 string 对象会导致较高的内存分配量，会令系统进行垃圾回收。另外，这样做将与实现准确计时和分析的目标相冲突。

1.3 关于分析的思考

性能优化的方式之一是剥离那些消耗宝贵资源的不必要任务。可以最小化任何浪费，来最大化生产率。有效地利用手中的工具是极为重要的。了解一些最佳方法和技术来优化工作流是很有帮助的。

关于正确使用任何一种数据收集工具的建议，可以归纳为以下 3 种不同的策略：

- 理解 Profiler 工具；

- 减少干扰；
- 关注问题。

1.3.1　理解 Profiler 工具

Profiler 是一种设计良好、直观的工具，因此，只需要花一两个小时通过一个测试项目研究其选项并阅读文档，就可以了解它的大部分特性。对 Profiler 工具的优点、缺陷、特性和限制了解得越多，就越能理解它所提供的信息，所以花时间了解如何在场景设置下使用它是值得的。开发人员不希望在离软件发布还有两周时间的情况下，还有 100 个性能缺陷需要修复，而且还不知道如何有效地进行性能分析。

例如，应该始终注意时间轴视图中显示的图形的相对性质。时间轴视图不提供其垂直轴上的值，而是根据最后 300 帧的内容自动调整该轴，它可以使小的峰值因为有相对变化而导致看起来似乎问题较严重。因此，时间轴上的峰值或静止状态的值看起来很大，具有威胁性，但并不一定意味着存在性能问题。

时间轴视图中的几个区域提供了有用的基准分析条，它们以水平线的形式显示，并带有与它们相关的时间和 FPS 值，这些应该用来确定问题的严重程度。不要被 Profiler 工具骗了，以为大的峰值总是不好的。与往常一样，只有在用户注意到它时才重要。

例如，如果一个很高的 CPU 使用峰值没有超过 60FPS 或 30FPS 基准条(取决于应用程序的目标帧率)，那么明智的方法是忽略它，搜索别处的 CPU 性能问题。因为无论怎么改进有问题的代码块，最终用户都可能永远不会注意到它，因此该问题并不是一个影响用户体验的关键问题。

1.3.2　减少干扰

干扰的经典定义(至少在计算机科学领域)是没有意义的数据，而盲目捕获的没有特定目标的一批分析数据总是充满了开发人员不感兴趣的内容。较多的数据需要较多的时间来处理和过滤，这会让人分心。避免这种情况的最佳方法之一是删除对当前情况来说不重要的数据，来减少需要处理的数据量。

减少 Profiler 图形界面中的混乱现象，将便于确定哪些子系统导致资源使用的峰值。可以在每个时间轴视图区域中使用彩色复选框来缩小搜索范围。

提示：这些设置是在编辑器中自动保存的，因此请确保在下一个分析会话中重新启用它们，因为这可能会导致下一个分析会话中丢失一些重要的内容。

此外，GameObjects可以停用，以防止它们生成分析数据，这也有助于避免分

析数据的混乱。这自然会对停用的每个对象带来轻微的性能提升，然而，如果逐渐停用对象，在停用特定对象时，性能突然变得更容易接受，那么显然该对象与问题的根源有关。

1.3.3 关注问题

前面讨论了减少干扰，这个类别似乎是多余的，开发人员应该专注于解决眼前的问题，对吧?不完全是。专注是指不让自己因无关紧要的任务和徒劳无益的追求而分心。

使用 Unity Profiler 进行分析的性能代价很小，在选择Deep Profile 选项时，这种代价甚至更小。通过额外的日志记录可能会将较小的性能成本引入应用程序中。如果搜索持续几个小时，很容易忘记何时何地引入了分析代码。

可以通过度量来有效地改变结果。在数据采样期间实现的任何更改有时可能导致跟踪应用程序中不存在的 bug，而如果尝试在不使用其他分析工具的情况下复制场景，则可以节省大量时间。如果在不使用分析工具的情况下，瓶颈是可重现的、很明显的，就可以开始检测了。然而，如果在现有的调查中不断出现新的瓶颈，它们可能是测试代码引入的瓶颈，而不是新暴露出来的问题。

当完成了分析，完成了修复程序并准备进行下一个检测时，应该对应用程序进行最后一次分析，以验证更改是否达到了预期效果。

1.4 本章小结

本章介绍了如何检测和分析应用程序中的性能问题，讨论了Profiler 的许多特性和秘密，探索了各种策略。了解了各种不同的技巧和策略，只要体会它们背后的逻辑，并在情况允许的时候利用它们，就可以极大地提高效率。

本章介绍了发现需要改进的性能问题所需的技巧和策略。后续章节将探讨如何修复问题以及尽可能提高性能的方法。所以，学习了本章之后，应学习 C#开发的实践案例，以及如何在 Unity 脚本中避免常见的性能缺陷。