Unity Cookbook中文版

从游戏开发到AI实时解决方案

（第2版）

[澳] 帕里斯・巴特菲尔德 – 艾迪生（Paris Buttfield-Addison）
[澳] 乔恩・曼宁（Jon Manning）
[澳] 蒂姆・纽金特（Tim Nugent）著
周子衿 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权清华大学出版社出版

清華大學出版社
北 京

内容简介

作为游戏开发实用指南，本书第2版经过全面更新，沿用深受读者欢迎的三合一模式，针对139个有价值的问题给出了详尽的解决方案和有价值的讨论，旨在帮助初学者和中级开发者深入学习Unity引擎的强大功能。全书共13章，主题涉及基础脚本编写到高级技术（如AI、动画和网络等）。通过这本实用性更强的教程，读者将学会如何应用代码片段快速而准确地解决实际问题，尤其是3D游戏开发和2D游戏开发、仿真和人工智能项目中的2D/3D图形、物理、AI、声音、叙事、输入、脚本和网络等问题。

本书内容全面，可操作性强，适合希望高效使用Unity的专业人员和其他游戏开发爱好者阅读和参考。

北京市版权局著作权合同登记　图字号：01-2024-4857

图书在版编目（CIP）数据

Unity Cookbook中文版（第2版）/
书名原文：

ISBN

I. ① … II. ① … ② … III. ①　　②　　IV. ①TP18

中国版本图书馆CIP数据核字(2025)第　　号

责任编辑：文开琪
封面设计：Karen Montgomery，张健
责任校对：方婷
责任印制：
出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn
　　地　　址：北京清华大学学研大厦A座　　**邮政编码**：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：
经　　销：全国新华书店
开　　本：178mm × 233mm　　**印　　张**：25　　**字　　数**：340千字
版　　次：2025年1月第1版　　**印　　次**：
定　　价：109元

产品编号：109277-01

O'Reilly Media, Inc.介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、互动学习、认证体验、图书、视频，等等，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书一直被视为学习开创未来之技术的权威资料。我们所做的一切是为了帮助各领域的专业人士学习最佳实践，发现并塑造科技行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

O'Reilly Radar博客有口皆碑。

——*Wired*

O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。

——*Business 2.0*

O'Reilly Conference是聚集关键思想领袖的绝对典范。

——*CRN*

O'Reilly的一本书就代表一个有用、有前途、需要学习的主题。

——*Irish Times*

Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。

——*Linux Journal*

译者序

能熬制出鹄鸟之羹的名厨兼贤相伊尹如是说：“治大国若烹小鲜。”在我们游戏开发人员看来，游戏制作也不例外。为谁做，做什么，怎么做，用什么做，这些问题想清楚之后，距离修身齐家治天下估计也不远了。如果把最后做成的游戏比作丰盛的晚餐，那么 Unity 就是让我们身临其中长袖善舞的那个设备齐全、充满种种可能性的魔法厨房。如同大厨需要巧妙地运用厨房里的各种器具来确保每道菜品都能达到完美的口感与外观，游戏开发者也需要充分利用 Unity 提供的各种功能，才能制作出足以触动玩家并使其回味无穷的精品游戏。

然而，Unity 的“设备”往往过于复杂，容易让人晕头转向甚至有“入宝山而空回”的怅惘和遗憾。显然，为了解决这个普遍的问题，本书的作者为我们开发者精心设计了这本“烹饪指南”，其中收录了大量的“配方”——也就是 Unity 开发过程中常见问题的解决方案。无论是初学者还是经验丰富的开发者，都可以在本书中找到自己需要的“配方”。

本书的作者在游戏开发领域拥有深厚的资历，他们精通 Unity 引擎的各种功能和工具，并且在实际游戏开发中积累了大量宝贵的经验。他们将这些年摸索得到的最佳实践和技术心得整理成书中的一个个“配方”，这些“配方”已经反复在实战中得到了验证，能够帮助读者有效解决游戏开发中的各种实际问题。

此外，相比前一版，第 2 版紧跟 Unity 的最新进展，新增许多与时俱进的内容，旨在帮助读者始终站在技术的最前沿。

如果你在某个月黑风高的深夜还在为一个棘手的技术问题而头疼不已，那么请相信我，翻开这本书，说不定你能从中发现无数个让你拍手叫好的灵感与技巧。

当我刚开始制作游戏的时候，一度联想到济慈，他说过一句话：“在我看来，几乎人人都可以像蜘蛛那样，从体内吐出丝来结成自己的空中堡垒。刚开始工作的时候，只是外挂到树叶和树枝的几个尖儿，然后来回兜兜转转，到最后，竟然可以在空中布满美丽迂回的线，形成一张足以让自己赖以生存的网。”作为游戏制作人员，我们在用 Unity 制作游戏的时候，或许也可以“外挂”，在兜兜转转的旅途中，学得尽兴，做得开心，玩得酣畅淋漓！

最后，祝大家能够和我一样，能够翻开这本书，深入宝山而满归。

目录

前言

欢迎阅读本书（第 2 版）！在过去的 20 年，实时 3D 和游戏开发工具不断在进步，吸引了越来越多的人参与游戏开发，无论是大型 3A 游戏还是个人利用业余时间开发的独立游戏。

我们很高兴有机会更新这本书，也为我们为这本书付出的心血感到自豪。希望在你使用 Unity 引擎进行游戏开发时，这本书能成为一个你的得力助手。自从 2007 年我们开始使用 Unity 以来，Unity 已经取得了长足的进步，使用它的时间越长，我们越容易挖掘出它的新用法和新的优势！这一切非常令人兴奋。相信我们，这是我们的亲身体验。

我们 Secret Lab 工作室开发的大部分电子游戏都是用 Unity 开发的。Secret Lab 的主要作品包括与澳大利亚广播公司、航空公司、博物馆和学校合作开发的多款少儿游戏，曾获得 BAFTA 和 IGF 独立游戏大奖的《林中之夜》（Night in the Woods，由 Finji 发行）以及广受欢迎的 Yarn Spinner 插件。我们开始接触 Unity 以来，它就一直是我们创作路上不可或缺的伙伴。

这本书汇集了一系列“配方”，旨在解决 Unity 开发过程中经常遇到的问题。虽然本书可能无法解答你对 Unity 的所有疑问，但涵盖了日常游戏开发中需要完成的大部分任务。第 2 版增加了大量新的内容，反映了 Unity 自本书第 1 版发布以来的诸多变化，并全面更新了所有相关内容。尽管 Unity 的更新速度并不快，但保持走在技术的前沿仍然十分重要。

我们把自己初次使用 Unity 时遇到的难题和困惑整理到本书的配方中。真心希望它们能够为你提供帮助。

现在正是使用 Unity 的黄金时期。实时开发和游戏开发的世界日新月异，可以选择的工具和技术也达到了前所未有的水平。勇敢地迈出步伐，创造伟大的作品吧！

目标读者和写作风格

本书假设读者对 C# 语言或其他类似语言（如 C++ 语言、Java 语言、Go 语言、Swift 语言、Python 语言等）有一定的了解，但对 Unity 一无所知。本书旨在成为读者的得力助手，帮助读者更快地掌握 Unity 并开发出自己的游戏。

如果欣赏我们的写作风格，并希望深入学习更多与 Unity 相关的知识，推荐阅读《Unity 手机游戏开发》（*Mobile Game Development with Unity*），这本书同样由 O'Reilly 优秀的团队出版（*https://oreil.ly/XO9Jf*）。

本书的截图均来自 Windows 系统，但书中讨论的内容普遍适用于 Windows、macOS 和 Linux 等不同操作系统下的 Unity 使用场景。

本书的结构

本书分为 13 章。

- 第 1 章“Unity 基础”，介绍使用 Unity 时需要了解的基本概念，包括游戏对象、组件、场景以及如何使用 Unity 编辑器。
- 第 2 章“编写脚本”，讲解 Unity 的脚本接口，我们通过这种方式来编写代码，实现项目功能。讲解了基础知识之后，本章接着介绍一系列实用示例，其中包括如何实现保存和加载系统、如何高效地处理对象以及如何以便于代码和 Unity 处理的方式存储数据。
- 第 3 章“输入”，讲解如何获取用户的输入，如键盘输入、鼠标输入和游戏手柄输入。此外，本章还将探讨如何配置响应用户点击等输入事件的游戏对象。

- 第 4 章“数学知识”，简要说明开发游戏时需要了解的一些数学概念，如向量、矩阵和四元数。本章还要介绍这些概念的一些实际应用，如检测一个游戏对象是否位于玩家前方。
- 第 5 章“2D 图形”，探讨 Unity 内置的 2D 图形和物理系统。你将从中学习如何显示精灵（sprite），如何对它们进行排序以及如何实现它们之间的碰撞效果。
- 第 6 章“3D 图形”，讲解 Unity 的材质和着色系统，包括材质和着色器的工作原理、如何在 Unity 中创建着色器以及如何在场景中实现最佳视觉效果。
- 第 7 章“3D 物理与角色控制”，说明如何利用 Unity 的 3D 物理系统来满足常见的游戏玩法需求，如拾取和投掷对象，以及创建玩家可以乘坐的移动平台。
- 第 8 章“动画与运动”，介绍 Unity 的动画系统，说明如何实现角色在不同动画状态之间的自然过渡以及如何将玩家的操作与角色动画结合起来。本章还要介绍 Unity 的摄像机系统以及讲解如何使摄像机跟随目标移动。
- 第 9 章“逻辑与游戏玩法”，专注于设计和创建游戏玩法。本章要涵盖许多常见的游戏玩法，如管理玩家的任务状态、监控赛车是否在违规抄近道以及管理游戏对象之间的伤害处理。
- 第 10 章“行为、模拟和 AI”，讲解如何赋予游戏角色智能，使其能够检测到视野中的玩家、自动导航、规避障碍物以及寻找藏身处。本章还要简要探讨如何使用 Unity 进行模拟以及使用机器学习技术实现真正的人工智能。
- 第 11 章“音频与音乐”，介绍 Unity 中的音频系统。本章要从播放音频的基础知识讲起，然后逐步深入到更高级的功能，如将音频路由到不同的音频组以及在游戏角色说话时自动调整背景音乐的音量。
- 第 12 章“Unity 的 UI 系统”，介绍用于构建 UI（用户界面）的工具。本章还要专门讲解如何通过扩展 Unity 编辑器来创建自定义工具。
- 第 13 章“文件、网络和截图”，作为本书的最后一章，要探讨如何进行网络通信、如何从网上获取数据以及如何通过代码保存截图。

排版约定

本书使用的排版约定如下：

斜体

表示新术语、URL、电子邮件地址、文件名和文件扩展名

`Constant width`（等宽字体）

表示程序代码和段落内的程序元素，如变量名、函数名、数据库、数据类型、环境变量、语句和关键字等

`Constant width bold`（粗体等宽字体）

表示用户应该逐字键入的命令或其他文本

这个元素表示提示或建议。

这个元素表示一般性的附注。

这个元素表示警告或注意事项。

示例代码使用规范

如果想下载本书中的代码，请访问本书的 GitHub 存储库（*https://oreil.ly/unity-dev-cookbook-code*）。我们还为本书专门创建了一个网页（*https://oreil.ly/9b_pc*），其中保存了书中的所有源代码以及其他资源。

这本书旨在帮助你学习。一般来说，除非计划大量复制或使用书中的代码，否则可以随意在自己的程序和文档中应用本书提供的示例代码，不需要事先征得我们的同

意。举例来说，编写一个包含书中多个代码片段的程序不需要特别许可。但是，若要出售或分发 O'Reilly 书籍中的示例代码，则务必事先获得我们的正式许可。引用本书和书中的示例代码来回答问题不需要特别许可，但是，如果在你的产品文档中大量引用本书的示例代码，就需要获得正式许可。

虽然不是强制性要求，但我们总是欢迎并感激任何对书籍归属权益的标注。书籍归属标注应包含书名、作者、出版社和 ISBN。例如：“《Unity Cookbook》，作者：Paris Buttfield-Addison，Jon Manning 和 Tim Nugent（O'Reilly 出版社）。2023 年版权所有，Secret Lab Pty. Ltd.、ISBN 978-1-098-11371-1”。

如果认为自己使用示例代码的方式可能超出合理使用范围或上述许可范围，请随时通过以下邮箱联系我们：*permissions@oreilly.com*。

O'Reilly 在线学习平台（O'Reilly Online Learning）

近 40 年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 *http://oreilly.com*。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

致谢

帕里斯想感谢他伟大的母亲，如果没有她，他绝对无法像现在这样充满激情地开展每一项工作。此外，他还想感谢他了不起的妻子玛尔斯（Mars）。她们是他最坚强的支柱。

乔恩想感谢他的母亲、父亲以及大家庭里所有的成员，感谢所有人的大力支持。

蒂姆想要感谢他的父母和家人，即使他的生活态度比较散漫，他们也始终表现出极大的包容和理解。

我们仨都要向我们的朋友瑞秋·鲁梅利奥蒂斯（Rachel Roumeliotis）致以诚挚的感谢，她在过去 10 年所提供的专业技术指导和宝贵建议对我们仨的写作事业有着不可估量的帮助。

感谢本书的编辑米歇尔·克罗宁（Michele Cronin）。我们非常享受与她的合作，并且已经迫不及待地想开始写新书了。此外，还要特别感谢本书上一版的编辑杰夫·布莱尔（Jeff Bleiel）。他情绪稳定，表达清晰，充满热情，与他合作非常愉快。希望将来能够再次与他合作！

同样，要感谢我们在本书写作过程接触到的 O’Reilly Media 的所有工作人员，他们无疑是各自领域的权威。O’Reilly Media 拥有众多才华横溢的员工，无论是正式员工、合同工还是其他相关人士，他们都表现出令人惊叹的专业水准。

特别感谢托尼·格雷（Tony Gray）和苹果大学联盟（Apple University Consortium，AUC），他们为我们及这里提及的其他人员提供了巨大的支持。如果没有他们，这本书将不会诞生。听说托尼现在也踏上了写作的道路——不好意思，提前剧透啦！

还要感谢尼尔·戈尔德斯坦（Neal Goldstein），我们之所以踏上写作的道路，完全要归功于（或者说，归咎于）他。

还要感谢 MacLab 的朋友们（他们知道自己是谁，并且他们仍在守望着海军上将 Dolphin 的化神之路）以及克里斯托弗·吕格（Christopher Lueg）教授、莱昂妮·艾利斯（Leonie Ellis）副教授以及塔斯马尼亚大学的当前以及之前的工作人员，感谢他们的包容和支持。

感谢戴夫·J.（Dave J.）及其团队提供的咖啡。如果哪天你来到澳大利亚塔斯马尼亚州的霍巴特市，请务必在 Yellow Bernard 咖啡店品尝一杯咖啡。那里的咖啡品质一流。

感谢所有认真而专业的技术审阅人员。他们的建议带来了很大的帮助。

最后，感谢购买这本书的每一位读者朋友——在此深表感激！如果有任何反馈或建议，欢迎通过 *lab@secretlab.com.au* 与我们联系，或在推特 X 上找到我们 *@thesecretlab*。

第 1 章

Unity 基础

如果想使用 Unity 开发游戏，首先需要理解 Unity 的工作原理以及使用方法。本章介绍 Unity 的界面和功能，让大家为使用编辑器构建项目做好准备。Unity 较为复杂，对于新手，尤其是缺乏专业软件开发经验的用户，可能会使其望而生畏。虽然有时颇具挑战性，但它以独特而引人入胜的方式将编程与艺术融合到了一起，可以为用户带来极大的成就感。

如果之前从未接触过 Unity，建议先阅读本章以及第 2 章中的所有示例，然后再来探索书中的其他内容。

本章要介绍许多与 Unity 相关的专业术语，这将有助于理解本书的后续内容。

1.1 Unity 编辑器

问题

如何在 Unity 编辑器中导航？各个组件的作用分别是什么？如何使用 Unity 编辑器？如何进行自定义设置？

解决方案

Unity 的用户界面由一系列内容窗口构成，每个窗口的功能都不同。本节介绍其中最重要的窗口，并探讨如何根据个人的工作习惯来优化编辑器的布局。

在启动 Unity 并创建新的项目后，首先将看到编辑器的主窗口，如图 1-1 所示。

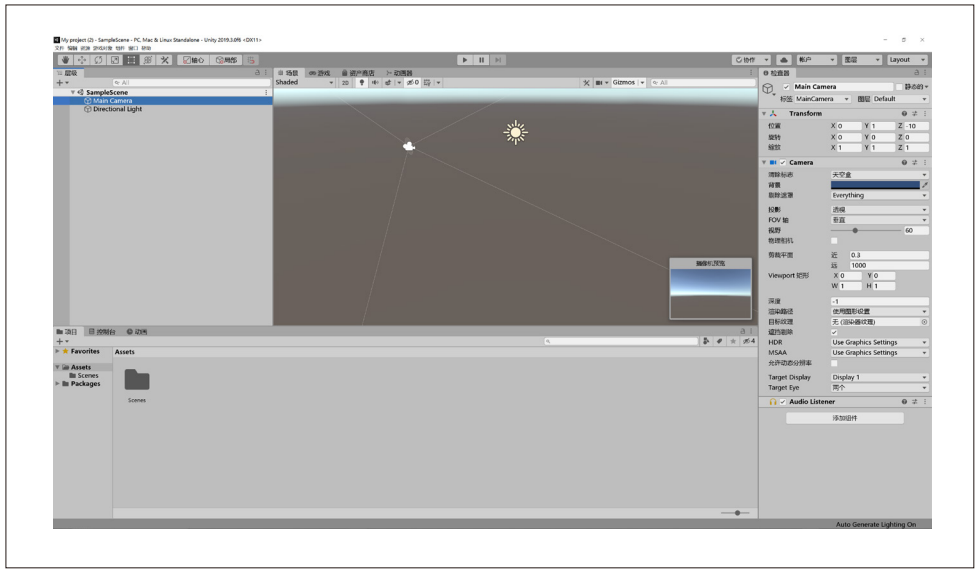

图 1-1：Unity 编辑器的主界面

Unity 首次启动时使用的布局与图 1-1 一致。如果需要，可以通过拖动任意窗口的边缘来调整窗口的大小。此外，也可以通过拖动窗口的标签来改变它的位置。如果将一个标签拖动到另一个窗口的边缘，它将紧挨着窗口的边缘。如果将一个标签拖动到窗口中间，它将被添加到窗口顶部的标签栏中。

有几个特性对于所有 Unity 用户来说都特别重要。让我们逐一来看看它们的作用。

Unity 默认处于“深色模式”，但为了在本书中提供更为清晰的屏幕截图，我们将其改成了“浅色模式”。因此，即使 Unity 界面颜色与书中有所不同，也请不要担心。

工具栏

工具栏包含对整个 Unity 环境产生影响的控件（图 1-2），它固定在编辑器窗口的顶部，不可移动。

图 1-2：工具栏

工具栏的控件从左至右依次如下：

工具面板

此面板控制着选中对象时出现的变换控件的行为。一次只能选择一种工具，这些工具如下：

手形工具，可以在场景视图中通过按住鼠标左键并拖动鼠标来移动场景视图。

移动工具，可以移动当前选中的对象。

旋转工具，可以围绕选中的对象的枢轴点或中心旋转该对象。

缩放工具，可以围绕选中的对象的枢轴点或中心缩放该对象。

矩形工具，选中的对象周围将会出现一个矩形，可以通过这个矩形来缩放和移动对象。这个工具主要用于处理 2D 对象，比如精灵和 UI 元素。

变换工具，此工具集移动、旋转和缩放功能于一体，可以通过它来移动、旋转和缩放选中的游戏对象。

自定义工具，游戏中由代码定义的所有自定义工具都将显示在此处。

轴心 / 中心切换按钮

这个按钮决定着变换控件是放置在对象的局部枢轴点还是对象的几何中心。对于某些 3D 模型来说，这可能会有所不同；例如，人物模型的枢轴点通常被设置在脚部。

局部 / 全局切换按钮

这个按钮决定着变换工具是在全局坐标系还是局部坐标系中进行操作。例如，在局部空间中，拖动移动工具的蓝色“前向”箭头会使游戏对象根据自身的朝

向向前移动，而在全局模式中，该操作则忽略对象的朝向，直接按照全局坐标系的方向进行移动。

播放按钮

单击这个按钮将启动播放模式，开始运行游戏。可以通过再次单击这个按钮来结束游戏模式并返回编辑模式。

可以在游戏模式中编辑场景，但在结束游戏时，对场景所做的任何修改都不会被保留。因此，在进行任何重大编辑前，务必记得检查当前是否处于游戏模式。

暂停按钮

单击这个按钮将暂停游戏。如果当前处于游戏模式，那么游戏将立即暂停。即使当前不处于游戏模式，也可以单击暂停按钮；如此一来，在单击播放按钮时，游戏将在第一帧之后立即暂停。

步进按钮

单击这个按钮将游戏推进一帧，同时保持游戏处于暂停状态。

具体来说，步进“一帧”意味着 Unity 将按固定的时间步（fixed timestep）[注 1] 推进游戏，然后重新渲染游戏。默认的时间步是 0.02 秒；可以在项目的“时间”设置中更改这个值（在“编辑”菜单中选择“项目设置 ”→“时间”）。

Collab 菜单

这个菜单提供 Unity 协作（Unity 的版本控制服务）的控制选项。

本书不会过多讨论 Unity 协作，关于这个功能的更多详情可以参见 Unity 的官方手册（*https://oreil.ly/9i3QP*）。

译注 1：又称“时间步长”，通常指游戏中的一个基本计算单位，决定着游戏世界中事件的发生频率。

服务按钮

单击这个按钮将打开“服务”视图，可以通过它来使用 Unity 的基于网络的服务，比如 Cloud Build、Unity Analytics 等。若想了解更多信息，请参阅 Unity 的支持与服务页面（*https://oreil.ly/75Qto*）。

账户按钮

可以通过这个按钮来设置 Unity 账户。

图层按钮

可以通过这个按钮来选择当前可见或可选择的图层。

Layout 按钮

可以通过这个按钮来保存和恢复窗口的自定义布局。如果设置了适合不同任务（比如编辑动画或布置关卡）的布局，可以通过这个按钮来轻松地在不同布局之间切换，这样就不需要每次都亲自调整窗口位置了。

场景视图

在场景视图中，可以查看、选择和修改场景中的游戏对象，如图 1-3 所示。在场景视图中，可以通过单击鼠标左键来选中任何对象；在选中对象后，就可以使用变换控件来移动、旋转或缩放它，具体取决于在工具栏中选择的是哪种工具，如图 1-4 所示。

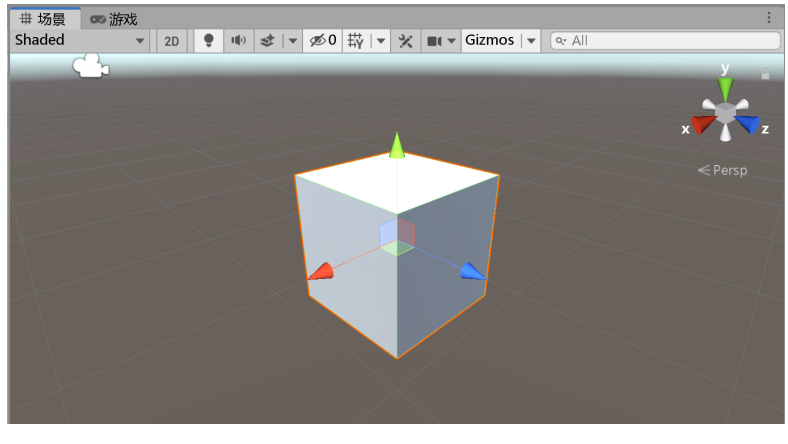

图 1-3：场景视图

要使用选定对象的变换控件，只需要点击并拖动游戏对象上的箭头（使用移动工具时）、圆圈（使用旋转工具时）或方框（使用缩放工具时）即可。如果同时按住 Shift 键，可以按照预定义的增量来进行移动、旋转和缩放。想要在场景视图中移动，可以在窗口左上角的面板中选择手形工具，然后按住鼠标左键并拖动鼠标。还可以同时按住 Alt 键（Mac 上为 Option 键）来旋转视图；按住 Alt 和鼠标右键（Mac 上为 Option 和鼠标右键）然后拖动鼠标，可以在 Unity 编辑器中平移视图。使用鼠标的滚轮或触摸板的双指手势可以放大和缩小视图。

按下 F 键可以快速地将场景视图聚焦于当前选定的对象。

图 1-4 显示了选定对象上的变换工具的例子。以摄像机为例，当它被选中且变换工具处于移动模式时，点击并拖动箭头将使摄像机沿箭头指示的方向移动。

游戏视图

游戏视图显示了通过摄像机看到的画面，并展示了玩家在独立程序中运行游戏时所看到的内容，如图 1-5 所示。游戏视图只有在播放模式下才能够交互。

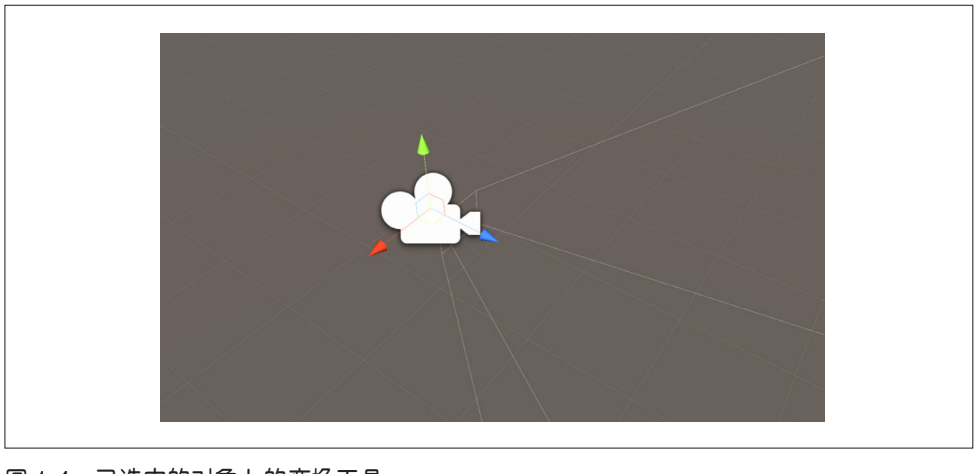

图 1-4：已选中的对象上的变换工具

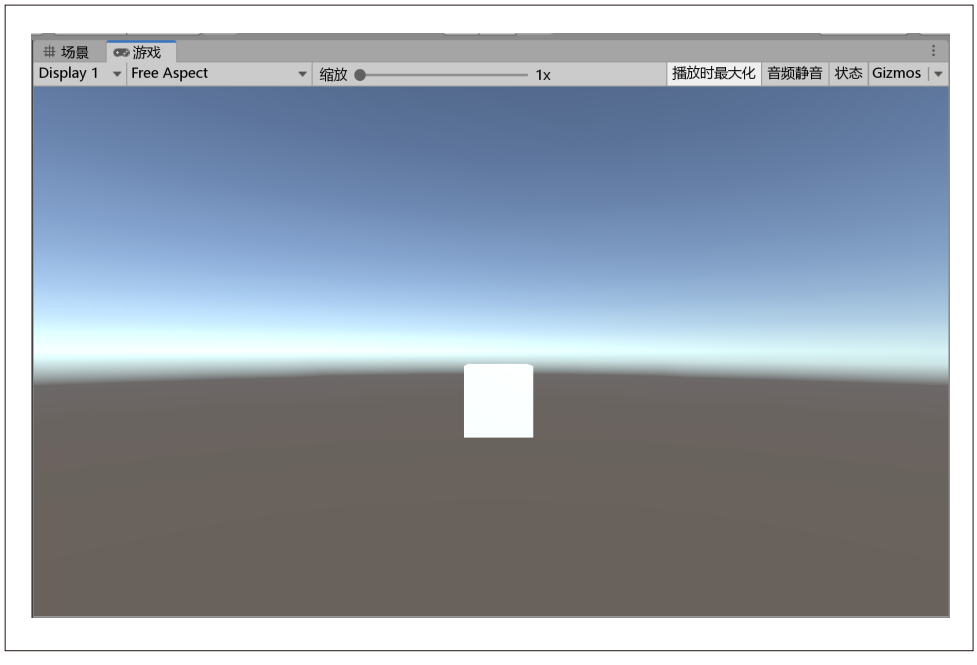

图 1-5：游戏视图

游戏视图顶部有几个控件，可以用来自定义游戏的展示方式。

Display（显示）菜单

用于选择要游戏视图中显示哪个显示器的内容。在 Unity 中可以修改摄像机的设置，以此将渲染结果输出到主屏幕或任何其他外部显示设备。

分辨率菜单

用于为游戏视图设置纵横比或特定的分辨率。

缩放滑块

用于放大游戏渲染视图。

“播放时最大化”开关

启用此开关后，当游戏进入播放模式时，游戏视图将填充整个编辑器窗口。

“音频静音”开关

用于禁用和启用游戏的所有音频输出。举例来说，在不希望游戏音频干扰到其他音乐时，这个功能非常有用。

“状态”开关

这个开关决定了是否打开一个显示状态统计信息的浮窗。

Gizmos 按钮

用于控制是否在游戏视图中显示代表特定场景对象（例如摄像机）的辅助图标，就像在场景视图中那样。

检查器视图

检查器视图显示了当前选中对象的信息，如图 1-6 所示。可以在这里调整游戏对象上附加的每个组件。

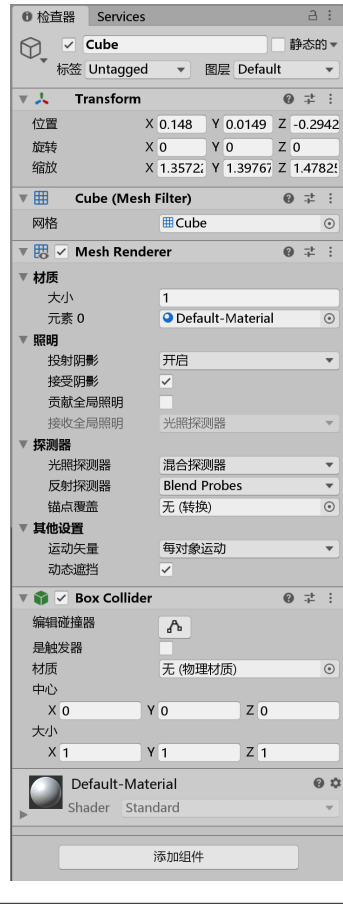

图 1-6：检查器视图

组件是 Unity 的重要组成部分，将留到 1.3 节深入讨论。

在检查器的顶部可以设置当前选中对象的名称。此外，还可以通过单击对象名称左侧的图标并选择要使用的符号来为对象设置图标。这尤其适用于不可见的游戏对象。

默认情况下，在选择了不同的游戏对象时，检查器的内容也会随之更改。如果想要固定显示某个特定对象的信息，可以单击检查器右上角的锁定图标。

层级视图

层级视图列出当前场景中的所有游戏对象，如图 1-7 所示。通过这个视图可以浏览和选择场景中的对象。如果在层级视图中选中一个对象，它将同时在场景视图中被选中，反之亦然。此外，还可以通过拖放操作，将一个对象设置为另一个对象的子对象。

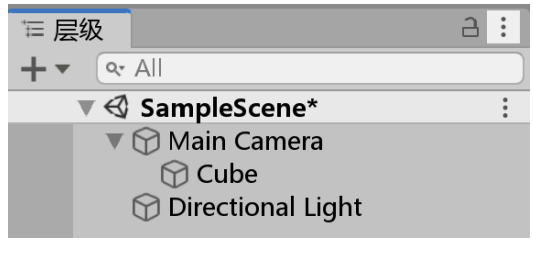

图 1-7：层级视图

项目视图

项目视图展示了项目的 Assets（资源）文件夹的内容，如图 1-8 所示。在这里可以查看、移动和重命名文件夹中的所有文件和资源。

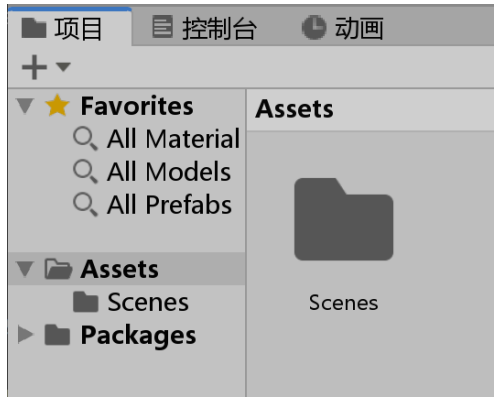

图 1-8：项目视图

想要重命名文件时，请在项目视图内操作，而不是在 Unity 外部操作。Unity 会为每个资源文件生成一个扩展名为 *.meta* 的文件来存储资源的元数据信息，例如纹理的导入设置。举例来说，*Hero.png* 资源文件旁边将有一个 *Hero.png.meta* 文件。如果在 Unity 内部重命名、移动或删除资源文件，Unity 也会相应地更新 *.meta* 文件；但如果在 Unity 外部重命名、移动或删除资源文件，Unity 将无法识别这些更改，导致必须重新生成 *.meta* 文件，这意味着文件的所有引用以及导入设置都会丢失。

讨论

想要关闭任意窗口时，可以右键单击窗口顶部的标签，然后在弹出的菜单中选择“关闭选项卡”。如果想要重新打开一个已关闭的窗口，通常可以在“窗口”菜单中找到并打开它。如果找不到窗口，可以打开“窗口”菜单，然后选择“布局”→“恢复出厂设置”。

1.2 游戏对象

问题

游戏的场景是用游戏对象填充的，那么应该如何创建和修改游戏对象呢？

解决方案

要想新建一个空的游戏对象，可以打开“游戏对象”菜单，然后选择“创建空对象”。这样就可以把一个新的空游戏对象添加到场景中。因为空对象没有任何渲染组件，所以在场景中它是不可见的。若想进一步了解组件，请参见 1.3 节。

同样，也可以通过快捷键 Control-Shift-N（Mac 上是 Command-Shift-N）来新建空的游戏对象。

选中游戏对象后，就可以在检查器中更改其名称。

可以将游戏对象设为其他游戏对象的子对象。当游戏对象移动、旋转或缩放时，它的子对象也会相应地受到影响。这意味着我们可以创建复杂的游戏对象层级结构，让它们作为一个整体，协同工作；例如，可以将汽车的各个车轮设置为它的子对象，这样在汽车移动时，车轮将会自动保持在正确的位置。若想将一个游戏对象设为另一个对象的子对象，只需在层级视图中将其拖放到另一个对象上即可（请参见前面的“层级视图”小节）。

还可以通过在层级视图中拖动游戏对象来调整它的排序。可以使用 Control-Equal（Mac 上为 Command-Equals）快捷键将选定对象移至其同级对象的末尾，还可以使用 Control-Minus 快捷键（Mac 上为 Command-Minus）将对象移到同级对象顶端。

讨论

此外，还可以通过将资源拖放到场景视图中来创建新的游戏对象。例如，如果将一个 3D 模型资源拖放到场景视图中，Unity 将创建一个游戏对象，其中包含渲染该模型需要的所有组件。

如果想要新建一个空对象作为某个对象的子对象，可以打开“游戏对象”菜单并选择“创建空子对象”来快速创建，对应的快捷键是 Alt-Shift-N（Mac 上为 Option-Shift-N）。

1.3 组件

问题

如何添加和修改控制着游戏对象外观与行为的组件？

解决方案

游戏对象本身只是一个空的容器。真正赋予它各种功能的是附加在其上的组件。

为了更好地理解组件的概念，我们将创建一个内置了一些实用组件的新游戏对象：我们将在场景中添加一个立方体。

按照以下步骤操作。

1. 打开“游戏对象”菜单，选择“3D 对象”→“立方体”。一个新的 Cube 游戏对象将被添加到场景中，如图 1-9 所示。

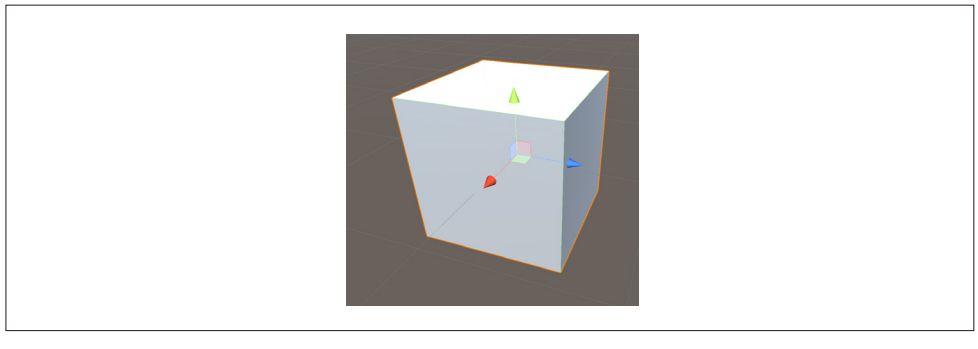

图 1-9：新添加到场景中的立方体

2. 在层级或场景视图中选中新的 Cube 游戏对象。检查器将自动刷新，以显示附加在 Cube 游戏对象上的组件列表，如图 1-10 所示。

每个游戏对象都至少包含一个组件：Transform（变换）组件。Transform 组件负责存储对象的位置、旋转和缩放，并跟踪其父对象。Transform 组件是不可移除的。

图 1-10 新立方体的检查器

Cube 游戏对象还包含其他几个组件。它们的功能都不同。

`MeshFilter`（网格过滤器）

这个组件用于从本地磁盘文件中加载网格，供 MeshRenderer 使用。新建的 Cube 游戏对象所使用的资源是由 Unity 提供的；但在开发自己的游戏时，大多数游戏对象都将使用我们自己添加到项目中的资源。

`MeshRenderer`（网格渲染器）

负责在屏幕上绘制网格，并且会使用材质资源来确定网格的外观。

BoxCollider（盒形碰撞体）

负责定义对象在游戏世界中的物理形状。

检查器中展示了组件的可配置属性，用户可以通过调整这些属性来改变它们的工作方式。举例来说，MeshFilter 组件只有一个属性：它要使用的网格。这是一个对象字段（object field）的例子，它是一个指向项目中的另一个游戏对象的引用。MeshFilter 的对象字段可以指向任何网格对象；而这些对象可以在项目资源中找到。对象字段能够引用的对象类型由组件本身决定；例如，MeshFilter 的字段中不接受除网格之外的对象。

对象字段不仅可以引用资源， 也可以指向场景中的其他对象。

有两种添加组件的方式，一是使用“组件”菜单，二是单击检查器底部的“添加组件”按钮。无论采用哪种方法，都能够选择并添加所需的组件类型。

讨论

如果想要移除某个组件，可以单击组件右上角的三个点的图标，并选择“移除组件”。

若想将组件复制到另一个游戏对象中，请单击三个点的图标，并选择“复制组件”。然后，选中另一个游戏对象，并单击任何现有组件的三个点图标（如果游戏对象不包含任何组件，请使用 Transform 组件）。单击“粘贴为新组件”后，之前复制的组件就会被粘贴过来。

脚本（详见第 2 章的讨论）也是组件，它们在游戏对象中的应用和操作与其他类型的组件并无二致。

1.4 预制件

问题

如何将游戏对象存储为文件，以便重用它的副本？

解决方案

默认情况下，在场景中创建的游戏对象仅存储在这个场景中。

如果想预定义一个对象，并创建它的多个副本，可以将其存储为预制件（prefab）。预制件是一种资源，它保存了游戏对象的所有属性；通过实例化预制件，可以在场景中创建它的副本。

要创建预制件，首先需要在场景中创建原始对象。举例来说，可以打开“游戏对象”菜单并选择“3D 对象”→“立方体”来创建一个新的立方体，然后，Cube 游戏对象将出现在场景和层级视图中。

接下来，将 Cube 游戏对象从层级视图拖放到项目视图中。这将创建一个新文件（图 1-11）——它就是预制件！同时，可以看到层级视图中的 Cube 游戏对象的名称变成了蓝色，这表明它是预制件的实例。

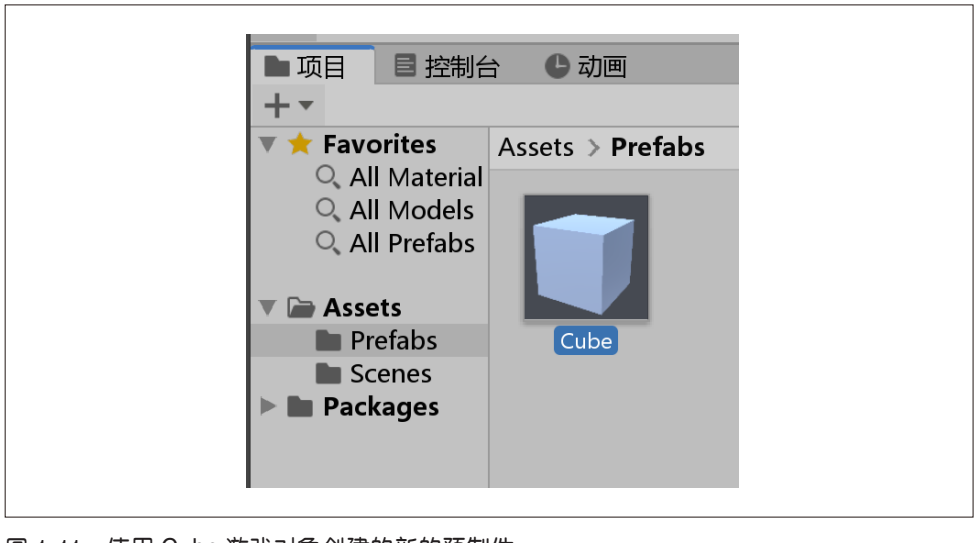

图 1-11：使用 Cube 游戏对象创建的新的预制件

现在可以放心大胆地从场景中删除原始的 Cube 游戏对象了。

可以通过将预制件拖放到场景中来创建其实例。实例是预制件中存储的游戏对象和组件的副本。

若想编辑预制件，只需要选中项目视图中的预制件资源，然后单击检查器中的“打开预制件”，就可以对其进行更改，如图 1-12 所示。

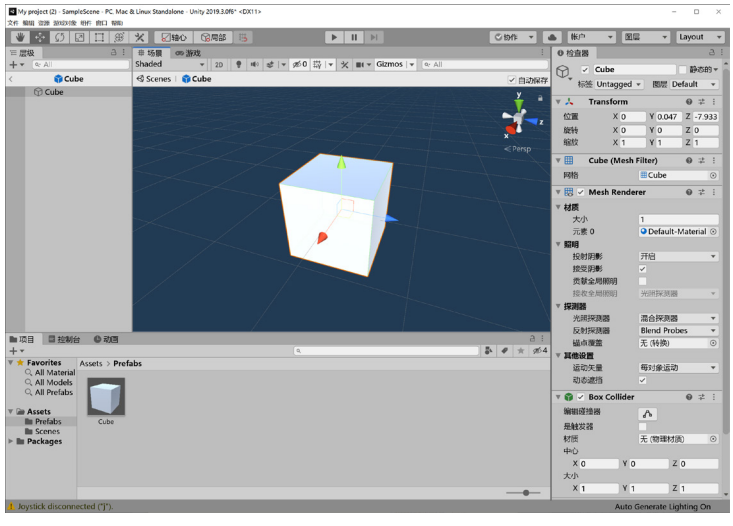

图 1-12：编辑 Cube 预制件

编辑完成后，单击“层级”视图中的返回箭头。对预制件所做的所有更改将自动同步到项目中所有的实例上。

讨论

预制件实例与原始预制件之间存在关联；对预制件进行更改时，对预制件所做的任何修改都会自动反映到所有相关实例上。反之则不成立，默认情况下，在实例上所做的更改不会影响原始预制件。举例来说，如果将 Cube 预制件的实例添加到场景中并修改其缩放属性，那么这些修改只会应用到这个实例上。此外，为了帮助用户识别，更改的属性将以粗体和蓝线突出显示。

但是，如果想要将更改应用到其他实例上，可以右键单击已更改的属性，然后单击“应用到预制件”。如果想要应用所有更改，可以在检查器顶部打开“覆盖”菜单，然后单击“应用所有”。

在“覆盖”菜单中，还可以查看对实例所做的所有更改。

1.5 场景

问题

场景是游戏对象的容器，应该如何创建和编辑场景呢？

解决方案

在创建新的项目时，Unity 会自动创建一个新的空场景。按下 Control-S（Mac 上为 Command-S），Unity 会将场景文件保存到磁盘；如果是第一次保存场景，Unity 会提示选择保存路径。

可以通过打开“文件”菜单并选择“新建场景”来创建更多场景。别忘了及时保存新建的场景，将它存储在硬盘中。

讨论

场景可以有多种不同的用途。例如，游戏的主菜单和各个关卡都可以作为单独的场景来存储。在游戏运行过程中，可以利用代码来加载新建的场景。这个问题将在 9.1 节中进一步讨论。

1.6 资源

问题

如何向项目中添加文件并配置 Unity 以将这些文件作为资源导入？

解决方案

要将文件添加到项目中，只需要将文件从资源管理器（Windows）或 Finder（macOS）拖放到项目视图中即可。Unity 会自动处理这些文件，并使它们成为项目中的可用资源。

文件被添加到 Unity 项目中后，就可以称为“资源”（asset）了。

在 Unity 导入文件后，就可以通过选择文件并查看检查器来配置 Unity 的导入方式了。不同的文件格式有不同的导入选项；例如，可以将图像设置为以精灵形式导入，这样 Unity 会生成一个附加的精灵数据资源，以便在精灵系统中使用（将在第 5 章讨论），也可以将图像作为纹理类型导入。

讨论

Unity 支持多种文件格式，具体如下所示。

3D 对象

支持 Autodesk FBX 和 Collada 格式；此外，如果安装了相应的软件，Unity 还可以导入 Maya、Cinema 4D、3ds Max、Cheetah3D、Modo、LightWave、Blender 和 SketchUp 等文件。

音频

支持 WAV、MP3、OGG 和 AIFF 格式；Unity 还支持多种音乐模块文件格式，包括 Impulse Tracker (.it)、Scream Tracker (.s3m)、Extended Module File Format (.xm) 和 Module File Format (.mod)。

2D 纹理

支持 Adobe Photoshop、BMP、PNG、JPG、BMP 和 TGA 等格式。

文本

支持 TXT 和 JSON 这两种格式。

Unity 能够捕捉到大多数在外部程序中对文件进行的更改，但这通常依赖于特定的应用程序。例如，如果想要在 Unity 中将 Blender 的 .blend 文件作为资源使用，则需要确保系统中安装了 Blender 应用程序。

1.7 构建 Unity 项目

问题

如何才能构建 Unity 项目，以便将游戏发布给玩家？

解决方案

若想构建游戏，可以打开“文件”菜单并选择“生成设置”（图 1-13），打开 Build Settings 窗口。

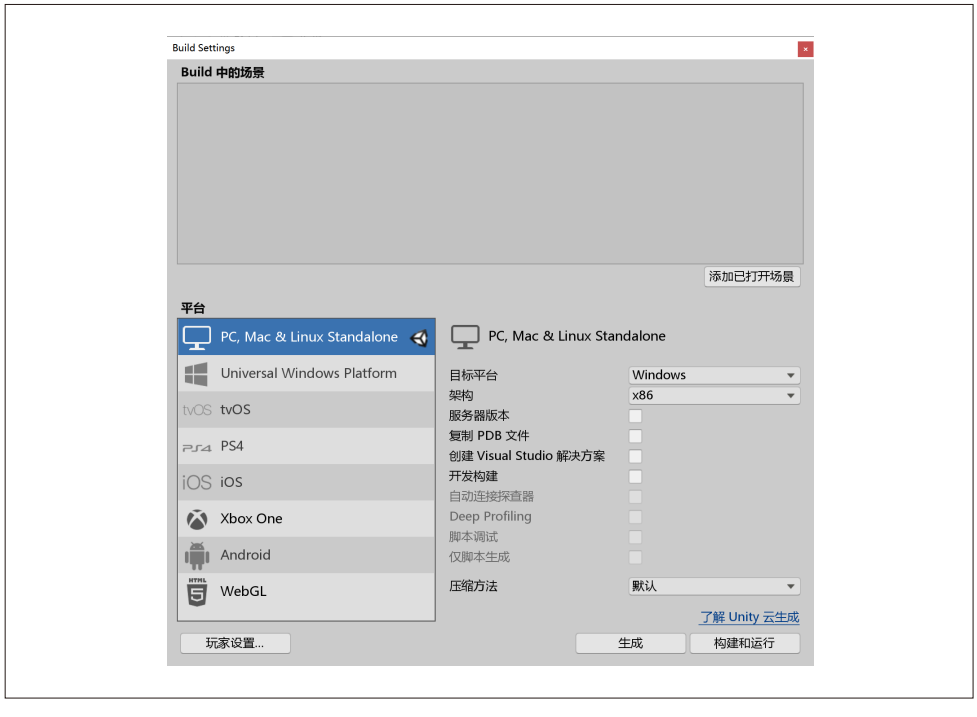

图 1-13：Build Settings 窗口

在构建游戏时，需要指定要包含的场景。如果尚未保存当前场景，请在“文件”菜单中选择“保存”，或按下 Control-S（Mac 上为 Command-S）。可以把想包含的场景拖放到“Build 中的场景”列表中，或者单击“添加已打开的场景”按钮来添加当前打开的所有场景。

接下来需要选择要构建到的平台。Unity 支持许多平台，包括个人电脑、手机、游戏机等等。一次只能选择一个平台；选中的平台将会以蓝色高亮显示。如果想为另一个平台构建项目，请选中它并单击“切换平台”按钮。

一切准备就绪后，就可以单击“生成”按钮了。Unity 会询问要保存构建文件的位置；设置完成后，Unity 就会开始构建。

讨论

如果想为某些平台构建项目，可能需要下载特定的平台支持模块。如果未安装必要的模块，Unity 将无法构建项目。要获取这些模块，可以选中平台并单击“打开下载页面”或其他类似的按钮，浏览器中应该会打开对应的页面，可以在其中下载模块。

某些平台（比如一些游戏主机）需要特殊的许可；若想了解更多信息，请参阅 Unity 的平台模块安装页面（*https://oreil.ly/KHvvw*）。

1.8 访问偏好设置

问题

如何访问 Unity 的全局设置和当前项目的设置？

解决方案

若想查看特定于项目的设置，请打开“编辑”菜单，选择“项目设置”。Project Settings 窗口将会出现（图 1-14）。

在 Project Settings 窗口中，可以调整游戏的各种设置，包括最大分辨率、构建名称、图形质量设置、输入等。

若想调整影响 Unity 应用程序本身的设置，则需要打开首选项窗口。

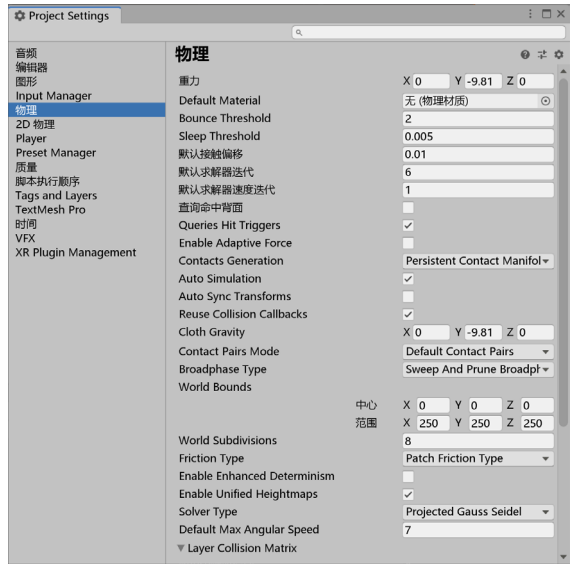

图 1-14：Project Settings 窗口

讨论

在首选项窗口（图 1-15）中，可以配置颜色、快捷键和各种适用于所有项目的设置。在 Windows 系统上，可以通过“编辑”菜单访问偏好设置；而在 MacOS 系统上，则需要通过 Unity 菜单进行访问。

图 1-15：首选项窗口

1.9 安装 Unity 包

问题

如何安装 Unity 包，以扩展 Unity 引擎的特性？

解决方案

可以使用 Unity 包管理器来安装新的包，这些包通常能够为引擎带来新的特性。

包和当前项目是绑定的。这意味着，在安装一个包后，它只会在当前项目中生效。如果想在其他项目中使用这个包，则需要重复一遍安装过程。

若想查看包管理器，请打开“窗口”菜单并选择“包管理器”。可以在其中搜索并通过互联网安装新的包。

讨论

Unity 的包管理器提供一种灵活的组件管理方式，允许各个组件独立于 Unity 引擎进行发布和更新。

Unity 包管理器通常简称为 UPM（Unity Package Manager）。

本书会用到一些包，并且目前 Unity 的大多数功能都是以包的形式提供的。若想了解更多信息，可以查看 Unity 的官方文档（*https://oreil.ly/-XVoh*）。

第 2 章

编写脚本

在 Unity 中创建行为需要通过编写代码——也就是编写脚本——来实现。Unity 中的脚本是用 C# 语言（发音同“C sharp”）编写的。C# 最初由微软开发。随着时间而发展，它已经拥有了一个完整的开源实现，Unity 正是在它的基础上实现的。

在深入了解本章的各个示例之前，强烈建议回顾一下第 1 章的内容！

本书并不会提供全面的编程或 C# 语言的学习指南——那可以单独写一本书了。本书的目的是展示如何使用 C# 语言编写 Unity 脚本。

虽然 C# 是 Unity 的主要编程语言，但我们也可以选择其他语言进行开发。Unity 可以与多种语言进行集成，并支持一系列图形化脚本工具，这些内容将在后续章节中进行简要介绍，但不会深入展开。

如果想要全面学习 C# 语言，强烈推荐阅读《C# 7.0 核心技术指南》（*C# 7.0 in a Nutshell*，网址为 *https://oreil.ly/j5KmC*）以及《深入浅出 C#》（*Head First C#*，网址为 *https://oreil.ly/wv54k*）。

2.1 向 Unity 场景中的对象添加脚本

问题

如何为游戏对象添加脚本，以便通过 C# 代码实现特定的行为逻辑？

解决方案

Unity 游戏对象是组件的容器。MonoBehaviour——得名于 Unity 使用的脚本运行时 Mono——是一个脚本组件。将 MonoBehaviour 附加到游戏对象后，它将参与到游戏的更新循环中，让我们能够在每一帧中执行代码。

即使在开发游戏以外的其他项目，比如模拟软件，Unity 中的基本对象仍然被称为“游戏对象”。所以本书也将沿用这一术语。

若想为游戏对象添加代码逻辑，则需要新建一个 MonoBehaviour 脚本，并将其作为组件附加到游戏对象上。这意味着首先需要在 Unity 场景中创建一个游戏对象。

1. 打开“游戏对象”菜单并通过“3D 对象”→“立方体”来创建一个新的 Cube 游戏对象。
2. 使用移动工具将 Cube 移动到摄像机前的某个位置。

接下来，我们将创建一个新的脚本。

1. 右键单击项目窗口中的 Assets 文件夹，选择“创建”→“C# 脚本”。
2. 这将创建一个新文件，并且 Unity 会提示我们为它命名。将文件命名为 Mover。

Unity 使用我们输入的名称来确定文件的内容。因此，编写 C# 类时，保持类名与文件名的一致至关重要，因为 Unity 会使用这些信息来保存场景的内容。如果决定重命名文件，请相应地更新类名。

3. 双击脚本文件。Unity 将使用默认编辑器打开它：对于 Windows 用户，这通常

是 Visual Studio；macOS 用户则会使用 Visual Studio for Mac；Linux 用户则可能是默认文本编辑器。

如果想要更改 Unity 用于编辑脚本的程序，可以打开“首选项”并选择“外部工具”→“外部编辑器”。对于 macOS 用户，我们强烈推荐使用 Visual Studio Code 和 BBEdit。

在脚本文件中，可以看到下面这样的默认代码：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Mover : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Start 方法在第一帧更新前调用，而 Update 方法每帧调用一次。

4. 像下面这样修改 Mover 类：

```
public class Mover : MonoBehaviour
{
    public Vector3 direction = Vector3.up;

    float speed = 0.1f;
```

```
    void Update()
    {
        var movement = direction * speed;
        this.transform.Translate(movement);
    }
}
```

虽然对于简单的演示来说，这段代码非常合适，但它并不是最有效的移动对象的方法。如果想知道原因，并探索更高效的方法，请参阅 2.3 节。

这段代码执行了以下操作。

- 创建了一个名为 direction 的公共 Vector3 变量，并将其值初始化为 Vector3.up。这是一个等于（0，1，0）的 Vector3。
- 创建了一个名为 speed 的私有 float 变量，并将其值初始化为 0.1f。

speed 值末尾的 f 在 C# 语言中很重要，因为不带 f 的非整数数值（如 0.1）会被默认为 double（双精度）类型而不是 float（单精度）类型。这意味着以下代码会导致编译错误：

```
float speed = 0.1;
```

正确的写法如下：

```
float speed = 0.1f;
```

- 定义了一个名为 Update 的新方法，并在这个方法中，创建了 movement 变量，该变量的值是 direction 和 speed 的乘积。然后，transform 属性上的 Translate 方法将会被调用，以移动对象。

创建好脚本之后，就可以将其作为组件附加到游戏对象上了。

1. 选中之前新建的 Cube 游戏对象。
2. 将 *Mover.cs* 脚本拖放到检查器中。

   在检查器中，可以查看和修改所有公共变量。由于 direction 的默认值是 Vector3.up，可以看到它的值显示为（0，1，0），如图 2-1 所示。

3. 按下播放按钮。游戏对象将开始向上移动。再次按下播放按钮，以停止游戏。
4. 修改 `direction` 变量中的一个数字，然后再次按下播放按钮。可以看到，游戏对象会以不同的方向或速度移动。

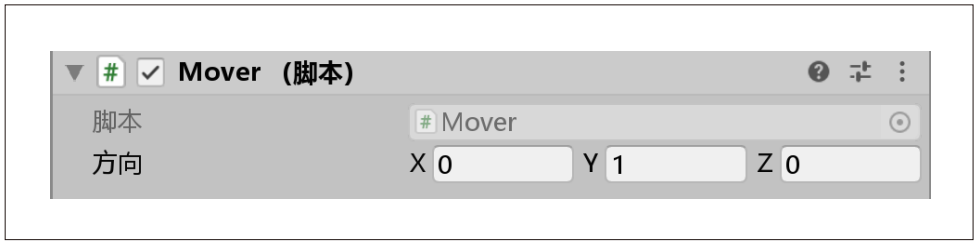

图 2-1：Mover 组件的检查器，显示方向变量

讨论

`MonoBehaviour` 是一个 C# 类。可以创建它的子类并实现特定的方法，Unity 会在适当的时机调用这些方法；例如，定义了 `Update` 方法后，Unity 会在每一帧调用它；定义了 `OnDestroy` 方法后，Unity 会在要从场景中移除游戏对象的时候调用它。2.2 节将深入探讨这些关键方法。

除了通过编辑器菜单创建脚本，还可以通过“添加组件”按钮创建脚本，该按钮位于检查器底部。单击“添加组件”后，在搜索框中输入脚本的名称。Unity 会尝试搜索同名组件；它还会提供一个选项来创建具有该名称的脚本（图 2-2）。通过这种方式创建的新脚本将存放在 *Assets* 文件夹中，但我们随时可以将它们移动到其他位置。

标记为 `public`（公开）的变量会显示在检查器中。这些变量同样可以被其他脚本访问。如果希望一个变量在检查器中可见，但又不希望它被其他脚本访问，可以移除 `public` 访问修饰符，并在其上方添加 `[SerializeField]` 属性，如下所示：

```
// 这个变量是私有变量，但由于 SerializeField 属性，它会显示在检查器中
[SerializeField]
float speed = 0.1f;
```

如果希望一个变量是公开变量，但又不想在检查器中显示它，可以考虑使用 `[HideInInspector]` 属性。

图 2-2：通过“添加组件”菜单创建新脚本

2.2 在脚本（或游戏对象）生命周期的特定时刻执行代码

问题

如何在脚本生命周期的特定时刻执行代码？

解决方案

Unity 会在不同的时间点调用 `MonoBehaviour` 的不同方法。

`Start`

`Start` 方法只会在每个对象的每个脚本首次激活时调用一次。如果脚本在游戏开始时已经附加到对象上，那么 `Start` 方法会在游戏开始的第一帧被调用。

`Update`

`Update` 方法在每个活动对象的每个脚本上每帧调用一次，这发生在摄像机渲染场景之前。由于每帧都会被调用，它是进行连续性任务（比如移动）的理想选择。

`Awake`

和 `Start` 方法类似，`Awake` 方法也在脚本首次激活时被调用。Unity 会在所有

Awake 方法执行完成后再执行 Start 方法；这意味着 Awake 方法是初始化脚本间引用（比如使用 GetComponent 和 FindObjectOfType，参见 2.4 节和 2.5 节）的最佳选择，而 Start 方法则是从这些引用访问数据的最佳选择。

LateUpdate

LateUpdate 方法在 Update 方法之后被调用。与 Awake 和 Start 类似，Unity 会在所有脚本的 Update 方法都执行完毕后再调用 LateUpdate。这提供了一个在其他操作之后执行特定行为的机会。例如，如果一个游戏对象包含用于移动的脚本，另一个游戏对象包含用于旋转以面向第一个对象的脚本，那么就应该将移动代码放在 Update 中，将旋转代码放在 LateUpdate 中，以确保旋转代码能够使游戏对象朝向正确的方向。

讨论

除了这些常见方法，还有其他一些方法适用于在特定情况下调用。例如，如果脚本实现了 OnDestroy 方法，那么在需要销毁对象时，OnDestroy 方法就会被调用。有些方法只有在游戏对象附加了特定类型的组件时才会被调用。例如，如果脚本实现了 OnBecameVisible 方法，并且游戏对象带有 Renderer 组件（比如 SpriteRenderer 或 MeshRenderer），那么当对象进入摄像机视野时，OnBecameVisible 方法就会被调用。

这些方法没有被任何父类重写。当一个类实现了某个方法时，Unity 会检查这个类，并记录下这个类实现了这个方法的事实，然后直接调用这个方法。

因此，这些方法是公开还是私有并不重要；Unity 无论如何都会调用它们。为了遵循编码规范，许多开发者倾向于将这些方法设为私有方法，以防止其他类调用它们，因为只有 Unity 引擎应该调用它们。

可以通过删除类中未使用的 Update 方法来优化帧率。更多信息请参阅 Unity 文档（*https://oreil.ly/vPggZ*）。

默认情况下，Unity 不保证不同脚本的方法运行顺序。但如果想要确保某些类型的脚本（如类型 A）在其他类型（如类型 B）之前执行 Update 方法，可以手动配置它们的执行顺序。要做到这一点，可以打开“编辑”菜单，选择“项目设置”，然后

选择“脚本执行顺序”。在这里，我们可以管理脚本的执行顺序，确保它们按照期望的顺序执行，如图 2-3 所示。

图 2-3：配置脚本执行顺序

2.3 创建与帧率无关的行为

问题

在游戏运行时，如何确保某些过程在固定时间段内进行，不受每秒帧数（即帧率）的影响？

解决方案

可以通过 Time 类的 deltaTime 属性来获取渲染上一帧所需要的时间，并将其整合到计算中：

```
void Update()
{
    var movement = direction * speed;

    // 乘以 deltaTime；现在 movement 代表每秒移动的单位数，而不是每帧移动的单位数
```

```
        movement *= Time.deltaTime;

        this.transform.Translate(movement);
    }
```

讨论

Unity 总是试图以尽可能快的速度运行所有内容，渲染尽可能多的帧数（直到达到 GPU 和显示器等硬件的能力极限）。由于场景复杂度的不同，每秒渲染的帧数可能会有所变化，相应地，渲染每一帧所需要的时间可能也不同。

因此，可以编写一个简单的 Update 方法，让对象每帧移动固定的距离：

```
    void Update()
    {
        var movement = direction * speed;

        this.transform.Translate(movement);
    }
```

这段简单代码的作用是，当渲染器每秒能够渲染更多帧时，对象移动的次数会更多；在每秒渲染的帧数较少时，对象移动的次数就会减少。因为对象在每次渲染一帧时都会移动，所以帧率的波动会导致对象的移动速度不稳定且每秒移动距离也不同。

为了解决这个问题，可以计算渲染上一帧所花费的时间。这可以通过 Time.deltaTime 来获得。

在数学中，希腊字母 delta（Δ）通常用来表示值的变化。因此，deltaTime 指的是自上一帧以来经过了多少时间。

将 movement 乘以 Time.deltaTime 后，movement 就不再表示固定距离，而是表示特定时间内移动的距离。如果把这个值传递给 transform 的 Translate 方法，对象就会以固定速度移动，不受帧率的影响。

deltaTime 并不是唯一可用的属性。如果想知道项目运行了多长时间，可以访问 time

变量。此外，还可以通过修改 Time.timeScale 属性来控制时间的流逝速度，这个属性也可以用来暂停游戏。

2.4 使用游戏对象上的组件

问题

如何使用附加到游戏对象的组件？

解决方案

利用 GetComponent 方法，可以获取脚本所附加到的游戏对象上的其他组件。

在调用 GetComponent 时，需要把所需组件的类型作为泛型参数传递。举例来说，如果想要获取游戏对象上的 Renderer 组件，则应该调用 GetComponent<Renderer> 方法。

如果游戏对象不包含相应类型的组件，GetComponent 将返回 null。如果试图使用 null 对象调用方法或访问属性和字段，将会引发空引用异常（NullPointerException），轻则导致脚本在当前帧停止执行，重则导致整个项目完全崩溃。所以，在尝试使用之前，务必检查返回值是否为 null。

获取另一个组件的引用后，就可以使用它了。举例来说，如果游戏对象带有 MeshRenderer 组件，就可以通过 GetComponent 访问该组件，并通过访问其属性来随时间改变其材质的颜色：

```
public class ColorFading : MonoBehaviour
{

    void Update()
    {
        var meshRenderer = GetComponent<MeshRenderer>();
        // 在使用之前检查返回值是否为 null
        if (meshRenderer == null) {
            return;
        }

        var sineTime = Mathf.Sin(Time.time) + 1 / 2f;
        var color = new Color(sineTime, 0.5f, 0.5f);
```

```
            meshRenderer.material.color = color;
        }
    }
```

讨论

除了 GetComponent 方法，Unity 还提供了许多相关方法，可以用来访问多个组件、访问父对象上的组件，或访问子对象上的组件。

`GetComponent<T>`

在当前对象上查找类型为 T 的组件。

`GetComponents<T>`

在当前对象上查找所有类型为 T 的组件，并将它们以数组形式返回。

`GetComponentInChildren<T>`

在当前对象或其子对象（或其子对象的子对象，执行深度优先搜索）中查找第一个类型为 T 的组件。

`GetComponentsInChildren<T>`

与 GetComponentInChildren<T> 类似，但会查找所有类型为 T 的组件，并将它们以数组形式返回。

`GetComponentInParent<T>`

在当前对象或其父对象（直到顶级对象）上查找第一个类型为 T 的组件。

`GetComponentsInParent<T>`

在当前对象或其父对象（直到顶级对象）上查找所有类型为 T 的组件，并将它们以数组形式返回。

如果每帧都会用到一个对象，可以在 Awake 方法或 Start 方法中调用 GetComponent（或其相关函数）并将结果存储在类的变量中，以稍微提高性能。

下面的代码和之前的代码大体相同，只有一个区别——GetComponent 仅在 Start 方法中调用一次：

```
public class ColorFading : MonoBehaviour
{
```

```
    // 在执行 Start 方法之后，meshRenderer 组件将存储到这里
    MeshRenderer meshRenderer;

    void Start()
    {
        // 获取组件并缓存它
        meshRenderer = GetComponent<MeshRenderer>();
    }
    void Update()
    {

        // 在使用之前检查返回值是否为 null
        if (meshRenderer == null) {
            return;
        }

        // meshRenderer 已被存储，因此可以直接使用它

        var sineTime = Mathf.Sin(Time.time) + 1 / 2f;
        var color = new Color(sineTime, 0.5f, 0.5f);
        meshRenderer.material.color = color;
    }
}
```

2.5 查找附加到游戏对象的对象

问题

在不确定组件具体附加到哪个游戏对象的情况下，我们如何使用这个组件？

解决方案

如果想要查找指定类型的单个组件，我们可以使用 FindObjectOfType 方法：

```
// 在场景中寻找一个 Mover 对象
var mover = FindObjectOfType<Mover>();
```

如果存在多个该类型的组件，FindObjectOfType 将选择其中的一个返回。它通常会按照 InstanceID 的升序选择对象。举例来说，InstanceID 为 -24084 的对象会优先于 InstanceID 为 -24082 的对象之前被返回。

如果想要查找所有特定类型的组件，可以使用 FindObjectsOfType，它会返回一个数组，其中包含该类型的所有组件。

```
// 在场景中寻找所有 Mover 对象
var allMovers = FindObjectsOfType<Mover>();
```

请注意，类似于 GetComponent（参见 2.4 节），如果找不到需要的内容，FindObjectOfType 将返回 null。在尝试使用返回的结果之前，请确保检查它们是否为 null。

这些方法只会返回当前处于活动状态的对象。它们无法检索到任何处于非活动状态的对象。

讨论

FindObjectOfType 通过遍历场景中所有对象的所有组件来执行搜索。不要在 Update 方法中调用这个方法，因为这会给 Unity 带来大量工作，而且没必要每帧都执行搜索。最好在 Start 或 Awake 中调用该方法，并将结果存储在变量中，以便后续使用。

2.6 单例模式

问题

如何创建始终只有一个实例的类，并且其他代码能够随时访问这个实例？

解决方案

单例（singleton）是一个有意如此设计为只有一个实例的类。在游戏开发中，一些需要全局访问的功能，如游戏管理器、输入管理器或其他通用工具，通常以单例的形式实现。

若想创建单例，需要先创建游戏对象，然后新建一个脚本并附加到游戏对象上。接着，在类上创建一个公共的静态变量，该变量的类型与类相同。然后，在 Awake 方法中，

将该变量的值设为 this，这意味着代码的其他部分都可以引用并访问这个特定的类实例。示例代码如下：

```
public class SimpleSingleton : MonoBehaviour
{
    // static（静态）变量在类的所有实例间共享
    public static SimpleSingleton instance;

    void Awake() {
        // 当这个对象被唤醒时，将实例变量设置为当前对象
        // 由于 instance 是公共静态变量，任何类都可以从任何位置访问它并调用其方法
        instance = this;
    }

    // 这是一个示例方法，只要场景中存在带有 SimpleSingleton 组件的游戏对象，
    // 代码的其他部分就可以调用这个方法
    public void DoSomething() {
            Debug.Log(" 你好！我是单例！ ");
    }
}
```

可以通过使用类上的 instance 变量来访问单例类的实例：

```
// 访问单例并调用其方法
SimpleSingleton.instance.DoSomething();
```

只要场景中至少有一个游戏对象附加了这个脚本，这个单例实例就可以在代码的任何地方被访问和使用。

讨论

如果存在多个单例类，重复编写 instance 变量和 Awake 方法可能是一项很繁琐的工作，并且很容易导致意外错误。

为了避免这种问题，可以通过使用泛型（generic）创建更高级的单例类。这种类设计为可被继承，并提供了 instance 方法的自动实现。此外，它还能够确保 instance 变量指向单一实例——如果脚本已经附加到场景中的游戏对象上，则会找到它；如果没有，则创建一个新的实例：

```
// 这是一个泛型类，意味着可以根据指定的 T 类型创建多个版本
// 这里还添加了一个类型约束，要求 T 必须是 MonoBehaviour 的子类
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    // instance 属性。这个属性只有 getter，因此其他部分的代码无法修改它
    public static T instance {
        get {
            // 如果还没有实例，则通过在场景中查找或创建一个新的来获取
            if (_instance == null) {
                _instance = FindOrCreateInstance();
            }
            return _instance;
        }
    }

    // 存储实际实例的变量。它是私有的，只能通过上面的 instance 属性访问。
    private static T _instance;

    // 尝试查找此单例的实例。如果找不到，则新建一个实例
    private static T FindOrCreateInstance() {

        // 尝试查找实例
        var instance = GameObject.FindObjectOfType<T>();

        if (instance != null) {
            // 如果找到实例，则返回它，它将被用作共享实例
            return instance;
        }

        // 由于脚本组件必须附加到游戏对象上，在新建实例前，需要先创建一个游戏对象

        // 确定单例的名称
        var name = typeof(T).Name + " Singleton";

        // 使用该名称创建作为容器的游戏对象
        var containerGameObject = new GameObject(name);

        // 创建并附加一个 'T' 类型的新实例；我们将返回这个新实例
        var singletonComponent = containerGameObject.AddComponent<T>();

        return singletonComponent;
    }
}
```

可以通过继承 Singleton 类来定义自己的新单例类。注意，需要在两个地方指定类名，首先是在声明类名时，然后是在泛型参数中（尖括号 < > 内）：

```
public class MoverManager : Singleton<MoverManager>
{
    public void ManageMovers() {
        Debug.Log(" 正在做一些事儿！ ");
    }
}
```

以这种方式定义的任何类都可以通过 `instance` 变量访问。这个属性永远不会为 `null`；如果没有场景中设置实例，代码会在 `instance` 变量第一次被使用时设置它。

```
MoverManager.instance.ManageMovers();
```

单例模式虽然便捷，但可能导致架构复杂性增加，难以维护。当一个类调用另一个类的方法时，两者之间会形成架构依赖：在需要更改或删除方法的时候，必须同时修改所有调用该方法的代码。如果一个类可以从多个地方被调用，就很容易创建大量依赖关系。随着代码复杂性的增加，维护工作会变得越来越困难。如果发现自己在多个不同的地方调用了单例类，可能需要考虑重新组织代码，以减少依赖关系。

2.7 使用协程来管理运行中的代码

问题

如何编写跨多个帧执行的代码？

解决方案

在常规应用程序中，代码的目标通常是尽可能快地运行并完成任务，但游戏不一样。在游戏中，许多动作和事件都需要随时间进行；例如，一个球需要一定的时间才能滚下山坡，而且游戏中的事件往往也依赖于时间。

Unity 中的协程（coroutine，一种特殊的生成器）提供了一种解决方案，允许开发者编写随时间执行的代码。协程方法可以在执行过程中主动暂停（yield），然后在适当的时候恢复执行。

协程方法使用 C# 语言的 yield return 语法，作为生成器来实现。

协程方法返回 IEnumerator，并使用 StartCoroutine 方法启动。在方法内部，可以通过 yield return 一个表示等待时间的对象来暂时挂起该方法的执行。

例如，假设需要编写一个方法，该方法首先执行一系列操作，然后等待一秒钟，再接着执行后续操作。为此，可以创建一个新方法，使其返回 IEnumerator，并在需要等待一秒钟的地方使用 yield return 来返回 WaitForSeconds 对象：

```
IEnumerator LogAfterDelay() {
    Debug.Log(" 等我一秒钟！ ");

    yield return new WaitForSeconds(1);

    Debug.Log(" 我回来了！ ");
}
```

yield return 只能在返回 IEnumerator 的方法中使用。

要启动协程方法，需要调用 StartCoroutine 方法，并传入协程方法的返回值作为参数。Unity 将自动启动协程，并确保在适当的时机恢复执行：

```
StartCoroutine(LogAfterDelay());
```

Start 方法也可以作为协程方法使用。只需将其返回类型设为 IEnumerator，就可以在其中使用 yield return，如下所示：

```
IEnumerator Start()
{
    Debug.Log(" 你好 ...");

    yield return new WaitForSeconds(1);

    Debug.Log("... 世界！ ");
}
```

与其他方法一样，协程方法也可以接收参数。例如，对于以下方法：

```
IEnumerator Countdown(int times) {
    for (int i = times; i > 0; i--) {
        Debug.LogFormat("{0}...", i);
        yield return new WaitForSeconds(1);
    }
    Debug.Log(" 倒计时结束！ ");
}
```

可以通过在启动时提供参数来调用这个协程：

```
// 协程参数在函数调用时提供
StartCoroutine(Countdown(5));
```

由于协程方法会将控制权交还给 Unity 引擎，所以它们可以与无限循环结合使用：

```
IEnumerator LogEveryFiveSeconds() {

    // 进入一个无限循环，在其中等待 5 秒，然后继续做其他事
    while (true) {
        yield return new WaitForSeconds(5);
        Debug.Log(" 嗨！ ");
    }

}
```

在使用 `while (true)` 这样的无限循环时要小心！如果循环中没有适当的 `yield` 语句，Unity 将无法中断循环，导致程序卡死。在这种情况下，将无法退出播放模式，也无法保存场景中的任何未保存更改。请务必谨慎！

如果使用 `yield return null`，Unity 将等待一帧后继续执行。例如，下面的协程被用作帧计数器，每渲染 100 帧就记录一次：

```
IEnumerator RunEveryHundredFrames()
{
    while (true)
    {
        // 返回 null 时，协程会暂停执行，直到下一帧才继续
        yield return null;
```

```
            if (Time.frameCount % 100 == 0)
            {
                Debug.LogFormat("第{0}帧！ ", Time.frameCount);
            }
        }
    }
```

最后，可以使用 yield break 退出协程：

```
    while (true) {
        yield return null;

        // 在第 354 帧停止
        if (Time.frameCount == 354) {
            yield break;
        }
    }
```

讨论

本例主要使用 WaitForSeconds。不过，还有其他一些可以用来等待不同时长的对象。

WaitForEndOfFrame 会等到所有摄像机完成渲染后，在帧显示到屏幕上之前执行。举例来说，可以利用它来在所有渲染工作完成后进行屏幕截图。

WaitForSecondsRealtime 会等待固定的时间，与 WaitForSeconds 相似，但它不受 Time.timeScale 的影响。

WaitUntil 和 WaitWhile 接受一个委托（即内联函数或方法引用），该委托在每一帧重复调用，以确定是否应该继续执行。它必须不带任何参数，并返回一个布尔值。

WaitWhile 会等到提供的函数返回 false：

```
    // 持续等待，直到此对象的 Y 坐标不小于 5
    yield return new WaitWhile(() => transform.position.y < 5);
```

WaitUntil 会等到提供的函数不再返回 false：

```
    // 持续等待，直到此对象的 Y 坐标小于 5
    yield return new WaitUntil(() => transform.position.y < 5);
```

2.8 使用对象池高效管理对象

问题

项目经常需要创建和销毁对象，如何使这个过程更加高效？

解决方案

创建一个对象池系统，以便在不需要使用时停用对象，需要时再重新启用，而不是销毁对象并彻底将其从内存中删除。

我们将创建一个脚本来管理 GameObject 队列，并公开一个名为 GetObject 的方法。在 GetObject 方法被调用时，它首先会查看队列中是否存在任何未在使用且可以回收的对象。如果有，则从队列中移除一个这样的对象并返回；如果没有，则创建一个新对象。

一旦不再需要某个对象，它会联系创建它的对象池，停用，并重新加入队列，以备将来再次使用。在整个过程中，对象都不会被销毁，而是被暂时存储起来以备后用。

讨论

为了实现这一功能，我们将编写一个新的脚本来执行这些操作。我们还将定义一个接口，允许对象在进入或离开队列时接收通知；虽然这不是必须的，但对于需要在每次（重新）进入场景时执行初始化设置的对象来说，这个接口非常有用。

1. 创建一个新的 C# 脚本并命名为 PooledObject.cs。
2. 在脚本中添加以下代码：

```
public interface IObjectPoolNotifier {
    // 在对象回到对象池时调用
    void OnEnqueuedToPool();

    // 在对象离开池或刚刚被创建时调用。如果 created 为 true，
    // 则表示对象是新创建的，而不是被回收的

    // 通过这种方式，可以用同一个方法来设置对象，无论是第一次还是后续使用。
    void OnCreatedOrDequeuedFromPool(bool created);
}
```

3. 如下修改 ObjectPool 类：

```
public class ObjectPool : MonoBehaviour
{

    // 用于实例化的预制体
    [SerializeField]
    private GameObject prefab;

    // 用于存储当前未被使用的对象的队列
    private Queue<GameObject> inactiveObjects = new Queue<GameObject>();

    // 从对象池中获取一个对象。如果队列中没有可用对象，
    // 则实例化一个新对象
    public GameObject GetObject() {

        // 检查是否存在任何待用对象可以重用
        if (inactiveObjects.Count > 0) {

            // 从队列中取出一个待用对象
            var dequeuedObject = inactiveObjects.Dequeue();

            // 队列中的对象均被设置为子对象，需要将它们移回根节点
            dequeuedObject.transform.parent = null;
            dequeuedObject.SetActive(true);

            // 如果对象带有实现了 IObjectPoolNotifier 的 MonoBehaviour 脚本，
            // 则通知它们对象已经离开了对象池

            var notifiers = dequeuedObject
                .GetComponents<IObjectPoolNotifier>();

            foreach (var notifier in notifiers) {
                // 通知脚本，对象已经离开对象池
                notifier.OnCreatedOrDequeuedFromPool(false);
            }

            // 将对象返回给调用者使用
            return dequeuedObject;

        } else {

            // 如果池中没有待用对象，则基于 prefab 新建一个对象

            var newObject = Instantiate(prefab);
```

```
            // 添加池标签（pool tag），使对象在完成后能够返回到对象池中
            var poolTag = newObject.AddComponent<PooledObject>();

            poolTag.owner = this;

            // 将池标签标记为不在检视面板中显示
            // 它没有可配置的选项，仅用于存储对创建它的对象池的引用
            poolTag.hideFlags = HideFlags.HideInInspector;

            // 如果对象带有实现了 IObjectPoolNotifier 的 MonoBehaviour 脚本，
            // 则通知它们对象已被创建。
            var notifiers = newObject
                .GetComponents<IObjectPoolNotifier>();

            foreach (var notifier in notifiers) {
                notifier.OnCreatedOrDequeuedFromPool(true);
            }

            // 返回新创建的对象
            return newObject;
        }
    }

    // 停用对象，使其回到对象池队列，以便将来重用
    public void ReturnObject(GameObject gameObject) {
        // 检索所有需要通知的 IObjectPoolNotifier 组件
        var notifiers = gameObject
            .GetComponents<IObjectPoolNotifier>();

        foreach (var notifier in notifiers) {
            // 通知组件对象将要回到对象池
            notifier.OnEnqueuedToPool();
        }

        // 停用对象，并将其设置为对象池的子对象，以保持层级面板整洁有序
        gameObject.SetActive(false);
        gameObject.transform.parent = this.transform;

        // 将对象放入非活动对象的队列中，等待下一次使用
        inactiveObjects.Enqueue(gameObject);
    }
}
```

这段代码目前还不能编译，因为尚未创建 PooledObject 类。我们很快就会创建这个类。

为了能让对象返回到创建它们的对象池中，它们需要持有对该池的引用。为此，我们将新建一个简单的脚本，该脚本不包含任何逻辑，仅用于存储对对象池的引用。

由于这个脚本与游戏玩法无关，所以没有必要在检查器中显示它。这就是 hideFlags 属性发挥作用的时候了；通过在 GetObject 方法中将其设为 HideInInspector，可以让 Unity 不显示这个脚本组件。这种方法虽然方便，但使用时务必小心，因为如果不能在检查器中看到组件，我们就很容易忘记它的存在。如果忘记自己在某个隐藏的脚本组件中编写的逻辑，很可能会在游戏因此而出错时没有任何头绪。

1. 新建一个脚本并命名为 *PooledObject.cs*，将以下代码添加到脚本中：

```
// 一个简单的脚本，仅用于存储对 ObjectPool 的引用
// 将会被 ReturnToPool 扩展方法所使用
public class PooledObject : MonoBehaviour
{
    public ObjectPool owner;
}
```

   为了简化游戏对象返回对象池的过程，我们将为 GameObject 类添加一个便捷的方法——ReturnToPool 扩展方法。创建 ReturnToPool 方法之后，就可以在任何游戏对象上调用 这个方法来让游戏对象返回创建它的对象池中。

2. 在 *PooledObject.cs* 中添加以下代码：

```
// 这个类为 GameObject 类提供了一个扩展方法：ReturnToPool
public static class PooledGameObjectExtensions {

    // 这是一个扩展方法（注意 this 参数）。
    // 这意味着它是添加到 GameObject 类所有实例上的新方法；
    // 可以这样调用它：gameObject.ReturnToPool()

    // 将对象返回创建它的对象池中
    public static void ReturnToPool(this GameObject gameObject) {
        // 查找 PooledObject 组件
```

```
            var pooledObject = gameObject.GetComponent<PooledObject>();

            // 找到了吗？
            if (pooledObject == null) {
                // 如果没找到，说明这个对象并非来自对象池
                Debug.LogErrorFormat(gameObject,
                " 无法将 {0} 返回对象池中，因为它并非从对象池创建。",
                gameObject);
                return;
            }

            // 通知原始对象池，这个对象应该返回其中
            pooledObject.owner.ReturnObject(gameObject);
        }
    }
```

至此，对象池系统的开发工作就全部完成了。

测试

为了测试对象池，我们需要创建两个脚本。一个脚本将尝试使用对象池，另一个脚本则会在等待指定时间后将自己（实际上是它附加到的游戏对象）返回对象池中。先从创建对象池开始：

1. 新建一个 C# 脚本，并将其命名为 *ObjectPoolDemo.cs*。在其中添加以下代码：

```
// 使用对象池的测试示例
public class ObjectPoolDemo : MonoBehaviour
{
    // 我们将从这个对象池中获取对象
    [SerializeField]
    private ObjectPool pool;

    IEnumerator Start() {
        // 每隔 0.1 秒到 0.5 秒从对象池中获取一个对象并放置它
        while (true) {

            // 从对象池中获取（或创建，怎样都可以）一个对象
            var o = pool.GetObject();

            // 在半径为 4 的球体内随机选择一个点
            var position = Random.insideUnitSphere * 4;
```

```
            // 将对象放置到选定位置
            o.transform.position = position;

            // 等待 0.1 到 0.5 秒，然后重复此过程
            var delay = Random.Range(0.1f, 0.5f);

            yield return new WaitForSeconds(delay);
        }
    }
}
```

该脚本使用对象池来创建对象并在短暂的延迟后将它们随机放置在场景中任一位置。

2. 现在，编写一个在短暂延迟后自动返回对象池的脚本。创建一个新的 C# 脚本并命名为 *ReturnAfterDelay.cs*，在其中添加以下代码：

```
// 演示对象池的使用过程的脚本示例
// 这个对象会在等待一秒钟后返回对象池
public class ReturnAfterDelay : MonoBehaviour, IObjectPoolNotifier
{
    // 当对象被创建或从对象池中取出时，可以在此处进行必要的初始化设置
    public void OnCreatedOrDequeuedFromPool(bool created)
    {
        Debug.Log(" 已从对象池中取出！ ");
        StartCoroutine(DoReturnAfterDelay());
    }

    // 当对象被返还到对象池时，此方法将被调用
    public void OnEnqueuedToPool()
    {
        Debug.Log(" 已返回对象池！ ");
    }

    IEnumerator DoReturnAfterDelay()
    {
        // 等待一秒钟然后返回到对象池
        yield return new WaitForSeconds(1.0f);
        // 将对象返还到它原本所属的池中
        gameObject.ReturnToPool();
    }
}
```

注意，这个脚本调用的不是 `Destroy`，而是 `ReturnToPool` 方法。如果调用的是 `Destroy`，对象占用的内存会被释放。

现在可以准备一个场景来应用这些脚本。首先，新建一个将要实例化的对象。

1. 打开“游戏对象”菜单，选择“3D 对象”→“球体”来创建一个球体。
2. 将球体命名为 Pooled Object。
3. 将 ReturnAfterDelay.cs 脚本拖放到检查器中。
4. 将球体拖放到项目面板中，将其设为预制件。
5. 从场景中删除球体。

接下来创建对象池本身，并配置它来创建和管理 Pooled Object 预制件的实例。

1. 打开“游戏对象”菜单，选择“创建空对象”来创建一个新的空游戏对象。
2. 将新对象命名为 Object Pool。
3. 选中该对象，将 *ObjectPool.cs* 脚本拖放到检查器中。
4. 将 Pooled Object 预制件拖放到“预制件”插槽中。

最后，创建一个使对象池生成对象的对象。

1. 打开“游戏对象”菜单，选择“创建空对象”来创建一个新的空游戏对象。
2. 将新对象命名为 Object Pool Demo。
3. 将 *ObjectPoolDemo.cs* 脚本拖放到检查器中。
4. 从层级窗口中，将 Object Pool 对象拖放到 Pool 插槽中。

播放游戏以进行测试。球体将会出现并消失。一开始，层级视图中将会不停出现新的对象，但随着时间的推移，对象的数量将会稳定下来，因为对象池中的对象已经足以满足生成新对象的需求，如图 2-4 所示。当对象返回池中时，它们会被停用，并暂时被设置为对象池的子对象，而不是被完全销毁并释放它们占用的内存。

图 2-4：对象池中的对象

讨论

实例化预制件意味着 Unity 需要分配内存。这本身不是问题，真正麻烦的是，不再需要这些内存时会发生什么。Unity 使用垃圾回收器（Garbage Collector，简称 GC）来管理内存，这意味着系统会定期清理不再被引用的内存。这个过程被称为“垃圾收集”，它对性能有一定的影响，需要尽可能地避免，特别是在开发像游戏这样对性能要求较高的软件的时候。另外，内存分配本身也会消耗 CPU 资源，这也需要考虑在性能成本之内。

为了减少垃圾收集的工作，应该尽量避免分配和释放内存，尽可能重用现有内存。对象池就是这样一种机制，它通过有效管理对象的生命周期，减少了对分配新对象的需求。当需要一个对象时，对象池首先会检查是否存在已经分配了内存但当前未使用的对象。如果存在，则激活这些对象，将其从非活动对象池中移除并返回。只有当不存在未使用对象，才会创建新的对象。

与通过 `Destroy` 方法彻底销毁对象不同，对象池使得对象在不再需要时能够返回到池中。为了简化操作，本例为 `GameObject` 类添加了一个扩展方法，使得将对象返还到对象池变得和销毁对象一样简单。

2.9 在资源中使用 ScriptableObject 存储数据

问题

如何在资源文件中存储数据，以便脚本能够访问这些数据？

解决方案

创建一个继承自 ScriptableObject 类的子类，并在其中定义所需要的属性。

例如，可以创建一个存储颜色的资源，然后通过编写脚本来在游戏启动时使用该资源设置渲染器的颜色。现在，先来创建一个存储颜色的资源：

1. 创建一个新的 C# 脚本，命名为 *ObjectColour.cs*，并添加以下代码：

```
// 在“资源”->“创建”菜单中创建一个条目，以便创建此类型的新资产
[CreateAssetMenu]

// 记得将父类从 MonoBehaviour 改为 ScriptableObject！
public class ObjectColour : ScriptableObject
{
    public Color color;
}
```

   [CreateAssetMenu] 属性的作用是在 Unity 编辑器的“资源”→“创建”菜单中添加一个新条目，使我们能够快速创建存储 ScriptableObject 子类实例的资源文件。

   由于对象作为资源存储在磁盘上，所以可以像其他资源（比如模型和纹理）一样拖放到不同场景中的对象的检查器面板里，实现跨场景的复用。

任何可以在 MonoBehaviour 子类中使用的变量类型都可以在 ScriptableObject 中使用。

2. 打开“资源”菜单，选择“创建”→ Object Color。这将创建一个新的 ObjectColour 资源。

3. 选中这个资源并查看检查器（图 2-5）。可以看到，它像 MonoBehaviour 一样公开了变量。

4. 在检查器中将颜色设为 Red（红色）。

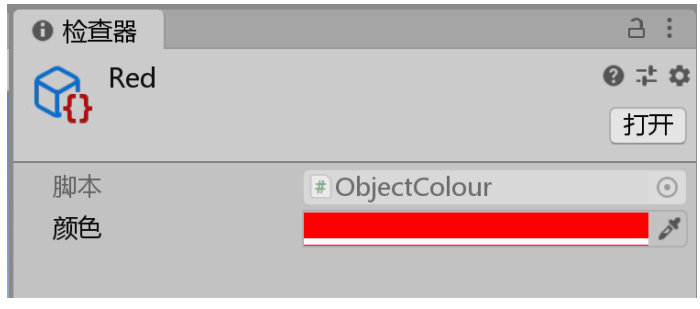

图 2-5：ObjectColour 资源的检查器

接下来，我们将创建一个简单的脚本，该脚本将使用 ObjectColour 资源中存储的颜色来修改渲染器的材质。

1. 新建一个 C# 脚本文件，命名为 *SetColorOnStart.cs*，并添加以下代码：

```
public class SetColorOnStart : MonoBehaviour
{
    // 我们将从这个 ScriptableObject 中获取数据
    [SerializeField] ObjectColour objectColour;

    private void Update()
    {
        // 如果没有提供 ObjectColour，则不使用它
        if (objectColour == null)
        {
            return;
        }

        GetComponent<Renderer>().material.color = objectColour.color;
    }
}
```

2. 打开 “游戏对象”菜单，选择“3D 对象”→“立方体”创建一个立方体。

3. 将 *SetColorOnStart.cs* 脚本拖放到立方体上。

4. 选中立方体，并将之前创建的 ObjectColour 资源拖放到立方体的 Object Colour 插槽中，如图 2-6 所示。

5. 运行游戏。立方体将会变成资源中存储的颜色。

如果创建多个使用相同资源的立方体，它们都使用相同的资源。它们都会显示相同的颜色。

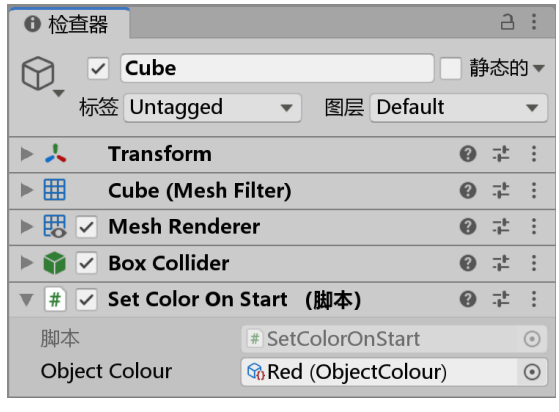

图 2-6：使用了 ScriptableObject 的对象的检查器

讨论

通过这种方法，可以将对象的数据与逻辑分离，并在游戏中的多个不同对象之间共享和重用数据。此外，这种方法还使得快速替换对象的整个数据集变得更容易了——简单替换它使用的资源即可。

第 3 章

输入

让玩家能够操作游戏是非常重要的，这是交互式游戏的基本原则之一。本章将探讨游戏开发者需要处理的常见输入需求。幸运的是，Unity 提供了多种输入方式的解决方案，包括键鼠输入和游戏手柄输入，甚至包括更复杂的系统，比如鼠标指针控制。本章将逐一介绍它们的基本用法。

在本章的学习中，你将了解如何使用传统的 Input 类和 Unity 新推出的输入系统。这两种工具在现代 Unity 项目中都发挥着重要的作用。

3.1 获取简单的键盘输入

问题

如何以最简单的方式检测用户的键盘输入？

解决方案

如果想要检测被按下的键，我们可以使用传统的 `Input` 类的 `GetKeyDown` 方法、`GetKeyUp` 方法和 `GetKey` 方法：

```
if (Input.GetKeyDown(KeyCode.A))
{
```

```
        Debug.Log("A 键被按下了！ ");
    }

    if (Input.GetKey(KeyCode.A))
    {
        Debug.Log("A 键被按住了！ ");
    }

    if (Input.GetKeyUp(KeyCode.A))
    {
        Debug.Log("A 键被松开了！ ");
    }

    if (Input.anyKeyDown)
    {
        Debug.Log(" 有键被按下了！ ");
    }
```

讨论

这些方法会在不同情况下做出响应。

- GetKeyDown 在按键在当前帧被首次按下时返回 true。
- GetKeyUp 在按键在当前帧被释放时返回 true。
- GetKey 在按键在当前帧被按住时返回 true。

此外，还可以使用 anyKeyDown 属性来判断当前是否有任何键被按下，在让玩家“按任意键继续”的场景中，这个属性非常有用。

Input 类是 Unity 旧有的输入管理器的一部分，从 Unity 最初的版本就已经存在了。Input 类简单易用，并且在新项目中默认启用，但它也存在一些局限：它不支持自定义按钮与游戏内操作的映射，设置游戏内操作的过程比较麻烦，并且开发者通常需要在它的基础上构建多个额外的系统。

Unity 新推出的输入系统是更理想的选择，它解决了传统 Input 类的限制。本章的余下部分都将使用这个系统。

3.2 使用 Unity 输入系统

问题

如何使用 Unity 输入系统获取用户输入？

解决方案

我们将通过包管理器（详见 1.9 节）安装 Unity 输入系统，按照以下步骤操作。

1. 选择“窗口”→“包管理器”以打开包管理器窗口。
2. 在窗口左上角的“包”设置中，将“在项目中”更改为“Unity 注册表”。
3. 在包列表中找到 Input System，或者在右上角的搜索字段中搜索 Input System。
4. 从列表中选择 Input System，然后单击“安装”按钮。

Unity 将会把输入系统下载并安装到项目中。Unity 可能会询问是否要更新项目的设置以使用新的输入系统；选择“是”，然后编辑器将会重启。

安装输入系统后，就可以在代码中使用它了。举例来说，通过以下代码，可以使用输入系统重新实现 3.1 节所展示的功能：

```
using UnityEngine.InputSystem;

if (Keyboard.current.aKey.wasPressedThisFrame)
{
    Debug.Log($"A 键被按下了！ ");
}

if (Keyboard.current.aKey.isPressed)
{
    Debug.Log($"A 键被按住了！ ");
}

if (Keyboard.current.aKey.wasReleasedThisFrame)
{
    Debug.Log($"A 键被松开了！ ");
}

if (Keyboard.current.anyKey.wasPressedThisFrame)
{
```

```
        Debug.Log("有键被按下了！ ");
    }
```

讨论

尽管可以直接访问特定按钮，比如键盘上的键，但更理想的方法是使用“Input Actions”（输入动作）来处理用户输入，这将在下一节中介绍。

在开发过程中，有时候可能会需要直接访问按钮。比如，在游戏开发的早期或测试阶段，配置一些仅供开发者使用的快捷键可以极大地提升效率。这些键可以执行一些特殊功能，比如免疫一切伤害。因为这些功能通常不会向玩家公开，所以没有必要将它们设置为 Input Actions，直接访问它们即可。

3.3 使用输入动作

问题

如何设置 Input Actions，以将实际输入与游戏内的动作关联起来？

解决方案

在 Unity 输入系统中，Input Actions 是玩家可以在游戏中执行的动作（例如移动、跳跃、转向），这些动作通过一个或多个用户可以交互的物理控制器（例如键盘上的 W 键，或游戏手柄上的左摇杆）来实现。

Input Actions 使我们能够站在高层游戏操作的角度进行思考，而不是必须考虑具体按下了什么按钮这样的底层细节。更重要的是，因为单个动作上可以绑定多个控件，开发支持多种控制方式的游戏变得更加轻松了。

在按照本节的步骤操作之前，请确保已经按照 3.2 节的步骤在项目中安装了 Unity 输入系统。

为了展示 Input Actions 的使用方法，我们首先将创建一个可以通过用户输入控制的简单场景。

1. 在新的空场景中，打开“游戏对象”菜单并选择“3D 对象”→“立方体”，以创建一个新的立方体。
2. 选中新创建的立方体，在检查器中单击“添加组件”。输入 **`CubePlayer`**，并选择 New script，然后单击“创建并添加”。
3. 打开新创建的 *CubePlayer.cs* 文件，将其中的所有内容替换为以下内容：

```
using UnityEngine.InputSystem;

public class CubePlayer : MonoBehaviour
{
    [SerializeField] float moveSpeed = 5;
    private Vector2 moveInput = Vector2.zero;

    public void Update()
    {
        var movement = new Vector3(moveInput.x, moveInput.y, 0) * moveSpeed *
Time.deltaTime;
        transform.Translate(movement);
    }

    public void ChangeColour()
    {
        var color = Color.HSVToRGB(Random.value, 0.8f, 1f);
        GetComponent<Renderer>().material.color = color;
    }

    public void OnMove(InputAction.CallbackContext context)
    {
        moveInput = context.ReadValue<Vector2>();
    }
}
```

4. 保存文件并返回 Unity。

如果现在进入播放模式，立方体将不会执行任何操作。我们很快就会解决这个问题！

我们已经准备好了一个可以响应玩家输入的立方体，接下来需要创建一个对象来处理这些输入并将它们传递给立方体。

1. 打开“游戏对象”菜单并选择“创建空对象”，以创建一个新的空游戏对象。将新对象命名为 Player Input。
2. 单击“添加组件”按钮，并选择 Input → Player Input。
3. 单击 Actions 字段右侧的选择器（圆形图标），选择 *DefaultInputActions*。

`DefaultInputActions` 是一组预定义的输入动作。其中一个预定义动作是 Move（移动），它绑定了与移动相关的常见控件，比如键盘上的 WASD 键和游戏手柄上的左摇杆。我们将使立方体根据这个动作来移动。

1. 在 CubePlayer.cs 中添加以下方法：

```
public void OnMove(InputAction.CallbackContext context)
{
    // 每当此动作发生变化时，获取当前值并存储它
    moveInput = context.ReadValue<Vector2>();
}
```

2. 保存文件并返回 Unity。
3. 接下来，我们需要输入系统，以便在 Move 动作状态发生变化时调用 OnMove 方法。首先，选中 Player Input 对象。
4. 在检查器中，将 Player Input 组件的 Behaviour 设置为 Invoke Unity Events（调用 Unity 事件）。
5. 组件底部将出现“事件”这一新设置。展开“事件”，再展开 Player。
6. 单击 Move 事件旁边的 + 按钮以添加新条目。
7. 将 Cube 游戏对象拖放到对象插槽中。
8. 将函数下拉菜单从 No Function 改为 CubePlayer → OnMove。
9. 单击播放按钮，尝试使用 W、A、S 和 D 键移动立方体。

通过这种设置，每当玩家进行移动操作时，Player Input 组件就会调用立方体的 `CubePlayer` 脚本中的 `OnMove` 方法。这个方法接收一个 `InputAction.CallbackContext` 方法，允许代码获取接收到的具体输入的相关信息。Move 动作被定义为包含二维信息（即左右和前后）的动作。在 `OnMove` 方法中，我们通过调用 `ReadValue<Vector2>` 来获取这些输入数据，并将其存储在 `moveInput` 变量中。接着，

我们在 Update 方法中使用这个变量来实际移动立方体。

之所以在 Update 而不是在 OnMove 中执行实际移动，是因为 OnMove 只会在输入变化时被调用。如果按下键盘上的 W 键，OnMove 将被调用一次，然后在松开 W 键时将再次被调用。因为我们希望按下 W 键可以“开始”移动，而不是“移动一次”，所以需要存储跨多个帧持续的信息。

如果有游戏手柄，请将其连接到电脑并移动左摇杆，可以看到，立方体将随你的操作而移动。市面上大多数游戏手柄都适用，尤其是 Xbox、PlayStation 或 Switch 等主流游戏主机的控制器，它们不仅兼容性好，也更受玩家青睐。

讨论

默认的 Input Actions 已经预设一些常见的动作，如移动、查看和射击。如果你做的游戏只需要这些基本动作，那就非常简单了！如果需要更多动作，则可以自行创建。

Input Actions 存储在项目中的资源中，因此如果要创建新的自定义输入动作，将需要创建一个用于存储它们的资源文件。

1. 选中 Player Input 游戏对象，在检查器中查看它的 Player Input 组件。
2. 通过单击 Actions 插槽并按退格键来清除它。
3. 单击 Create Actions（创建动作）按钮。
4. Unity 会询问想要保存新资源的位置，此时将其保存在 Assets 文件夹的某处即可。

Unity 将打开新创建的资源，现在可以查看动作列表了。

也可以使用“资源”→“创建”→ Input Actions 菜单创建新的输入动作资源，但这将创建一个空集合。如果只是想要在默认集合中添加新动作，最好通过 Player Input 组件来完成这一步骤，以避免重新设置所有内容。

默认的动作被分成两种不同的输入映射：Player（玩家）和 UI（用户界面）。

输入映射代表玩家可以采用的不同操作模式。举例来说，默认集合中的 Player 用于游戏的常规操作，UI 则用于与用户界面的交互操作。多个输入映射使游戏可以根据不同的操作需求，为相同的控制输入赋予不同的意义；例如，当输入映射设置为 UI 时，左摇杆的作用是“在用户界面中导航”，而不是“控制角色移动”。

现在让我们来看看 Player 输入映射。它包含三个动作：Move（移动）、Look（查看）和 Shoot（射击）。每个动作都配置多个绑定（binding），即与玩家可以交互的实体按钮之间的连接。例如，Move 绑定了游戏手柄上的左摇杆；键盘上的 W 键、A 键、S 键和 D 键；VR 控制器的主要 2D 轴；操纵杆（joystick）的摇杆。

现在，让我们创建一个新的动作——它将是一个按钮（也就是说，我们可以检测它是否被按下），用于改变立方体的颜色。

1. 在 Actions 一栏中，单击 + 按钮创建新的动作。将新动作命名为“Change Colour”（更改颜色）。

   注意，Action Type（动作类型）默认设置为按钮。保持这个设置不变。不过，还存在其他动作类型；例如，Move 动作的类型是“值”，它被配置为输出一个 2D 向量（这就是为什么 PlayerCube.cs 中的 `OnMove` 使用了 `ReadValue<Vector2>`）。将 Change Colour 的类型设置为“按钮”，可以在按下按钮和松开按钮时触发事件。

2. 新创建的动作没有任何已配置的绑定，我们需要自行设置。现在，选中 Change Colour 下的 <No Binding>。

3. 打开 Path（路径）下拉菜单，选择 Keyboard（键盘）→ By Location of Key (US Layout)（根据键位 [美式布局]）→ C。

   Path 下拉菜单允许我们配置绑定到动作上的具体物理控件。可以添加任意数量的绑定；举例来说，如果展开 Move 动作，我们可以看到大量绑定，每个绑定对应不同种类的输入设备。

   将路径设为“根据键位”使我们能够指定键盘上的一个位置。在这个例子中，我们将其设置为美式键盘布局中 C 键所在的位置。

键盘和按键含义

尽管键盘最初是为了打字而设计的，但在游戏中，它们的使用方式却大不相同。设计电脑游戏控制方案时，游戏设计师会先确定玩家手指的最佳放置位置，然后在此基础上进行反向设计。

毕竟，字母 W 本身并没有“前进”的含义，但 W、A、S 和 D 键的布局恰好适合美式 QWERTY 键盘布局，使得惯用手为右手的玩家可以轻松地将左手放在键盘上，便捷地控制角色的前后左右移动。

然而，如果将这些动作绑定到特定的字母键上，那么使用不同键盘布局的玩家可能会感到不便。此外，使用“键盘第二行第三个键（忽略功能键）”这种方式来描述按键位置也显得不够简单直观。

“根据键位”的绑定方法避免了这种问题，它通过指定美式布局上的一个键的位置而不是名称来工作。在将 Change Colour 动作绑定到 C 键的位置时，无论玩家使用的是何种键盘，动作都会绑定到相应位置的键上。

值得注意的是，“根据键位”尤其适合那些需要玩家频繁操作并培养肌肉记忆的按钮对于那些不常使用的动作，那么最好将其与键名绑定，绑定到特定的命名键，而不是键位，因为这样可以方便玩家根据键名快速找到相应的键。

4. 将 Use In Control Scheme（在控制方案中使用）设置为 Keyboard & Mouse（键鼠）。这样一来，在玩家选择这些控制方案时，相应的绑定就会生效。
5. 我们还需要将该动作绑定到游戏手柄上的按钮。在 Actions 列表中单击 Change Colour 右侧的 + 按钮，并选择 Add Binding（添加绑定）。这将新增一个空绑定。
6. 选中新绑定，并将 Path 设置为 Gamepad（游戏手柄）→ Button South（南侧按钮）。

几乎每个现代游戏手柄的右侧都有 4 个呈菱形排列的按钮。它们通常被称为 Face Buttons。南侧按钮代表菱形底部的按钮。不同的手柄对这些按钮有不同的称呼：南侧按钮在 PlayStation 控制器上是 X 按钮，在 Xbox 控制器上是 A 按钮，在 Nintendo Switch 控制器上则是 B 按钮。就像“根据键位”通过

指定键位而不是键名来工作一样，“南侧按钮”让我们可以指定手柄上特定位置的按钮。

7. 将 Use In Control Scheme 设置为 Gamepad（游戏手柄）。

如果没有游戏手柄可用，可以将其绑定到其他输入设备上（如鼠标左键）。

8. 单击 Save Asset（保存资源）并关闭窗口。

我们已经创建了新的动作并将其绑定到控件上，接下来还需要使立方体能够对该动作做出响应。

1. 在 *CubePlayer.cs* 中添加以下方法：

```
public void OnChangeColour(InputAction.CallbackContext context) {
    // 在执行此动作时，改变颜色
    if (context.performed) {
        ChangeColour();
    }
}
```

2. 保存文件，返回 Unity。
3. 选中 Player Input 对象，在检查其中查看它的 Player Input 组件。

在之前创建新动作资源的时候，Input Actions 字段已经与新资源关联了起来。如果想更改 Input Actions 关联的动作资源，可以通过拖放别的资源来实现。

4. 展开“事件”部分，找到 Player → Change Colour。
5. 单击 + 按钮来添加一个新的事件接收器。
6. 将 Cube 游戏对象拖放到“对象”字段中，并将函数改为 CubePlayer → OnChangeColour。
7. 按照之前的步骤，重新将 Move 动作与 CubePlayer 的 `OnMove` 方法关联；因为在更改组件使用的动作资源时，原有的绑定将会失效。

进入播放模式，按下键盘上的 C 键或游戏手柄上的南侧按钮（或任何绑定到 Change Colour 动作的控件）。立方体的颜色应该会改变。

3.4 锁定和隐藏鼠标光标

问题

如何在游戏中使用鼠标移动，但不在屏幕上显示鼠标光标？

解决方案

将 `Cursor.lockState` 属性设置为 `CursorLockMode.Locked` 或 `CursorLockMode.Confined` 可以防止鼠标光标离开游戏界面，将 `Cursor.visible` 设置为 `false` 可以使光标不可见：

```
// 将光标锁定在屏幕或窗口的中心
Cursor.lockState = CursorLockMode.Locked;

// 防止光标离开游戏窗口
Cursor.lockState = CursorLockMode.Confined;

// 不对鼠标光标施加任何限制
Cursor.lockState = CursorLockMode.None;

// 隐藏鼠标光标
Cursor.visible = false;
```

讨论

在 Unity 编辑器中，可以通过 Escape 键来重新启用鼠标光标，并解除对它的锁定。

3.5 响应鼠标悬停和点击事件

问题

如何检测用户何时将鼠标移动到对象上并在用户点击时触发事件？

解决方案

要检测鼠标光标在屏幕上的位置以及它何时悬停在场景中的对象上，可以使用 Unity 的事件系统。要开始使用事件系统，需要先通过“游戏对象”菜单创建一个带有 EventSystem 组件的对象。

1. 打开“游戏对象”菜单，选择 UI → “时间系统”。这将创建一个新的 EventSystem 对象。

在创建 Canvas 的时候，Unity 会自动创建一个 EventSystem，因为 Unity 的 UI 系统与本示例采用了相同的技术。

2. 接下来，我们将添加一个 Raycaster（射线投射器）组件，事件系统将使用它来确定光标下的对象。射线投射器从摄像机位置发出射线，通过光标位置延伸至场景中，然后检测射线命中了哪些对象。如果射线命中了某个带有碰撞体的游戏对象，事件系统就会向该对象发送事件。
3. 选中 Main Camera 游戏对象。
4. 打开“组件”菜单，选择“事件”→“物理光线投射器”。也可以通过单击检查器底部的“添加组件”按钮并从中选择 Event → Physics Raycaster 来添加它。

本例中通过利用 3D 物理系统来操作 3D 对象。对于 2D 游戏开发，虽然可以应用相似的技术，但需要为摄像机添加“2D 物理光线投射器”而不是“物理光线投射器”。物理射线投射器仅检测 3D 碰撞体，而 2D 物理射线投射器仅检测 2D 碰撞体。如果场景中同时使用了这两种类型的碰撞体，可以在相机上同时使用这两个组件。

5. 接下来，我们将编写一个脚本，当光标移动到脚本所附加的对象上时，脚本会检测到光标，并改变游戏对象的渲染器颜色。创建一个新的 C# 脚本，命名为 *ObjectMouseInteraction.cs*。
6. 在类的顶部添加以下 using 指令：

```
using UnityEngine.EventSystems;
```

创建事件系统后，如果项目已经配置为使用 Unity 输入系统（参见 3.2 节），Unity 可能会提示需要将组件更新为与输入系统兼容的版本。按照提示，单击“替换为 InputSystemUIModule”来完成替换。

为了访问用于处理指针（如鼠标）的类和接口，我们需要访问 `UnityEngine.EventSystems` 命名空间。完成这一步后，需要使类符合特定的接口，这些接口会将 `MonoBehaviour` 类标记为需要了解和响应输入事件的类。

7. 删除代码中的 ObjectMouseInteraction 类，使用以下代码来替换：

```
public class ObjectMouseInteraction : MonoBehaviour,
    IPointerEnterHandler,  // 管理鼠标光标进入对象时的事件
    IPointerExitHandler,   // 管理鼠标光标离开对象时的事件
    IPointerUpHandler,     // 管理鼠标按钮在对象上松开时的事件
    IPointerDownHandler,   // 管理鼠标按钮在对象上按下时的事件
    IPointerClickHandler   // 管理鼠标在对象上按下并松开时的点击事件
{

}
```

通过实现这些接口，Unity 将能够识别并响应附加对象上的输入事件，并触发此脚本中的相关方法。

这个示例中的接口数量可能超出了实际应用中的需要（例如，本例同时处理了指针的松开、按下和点击），但这主要是为了进行更全面的演示。不过，除此之外还有很多可注册的事件，若想了解更多信息，请查阅 Unity 官方文档。

8. 接下来，为了简化更改对象颜色的操作，我们将在 `ObjectMouseInteraction` 类中添加一个变量和相关方法：

```
Material material;
void Start() {
    material = GetComponent<Renderer>().material;
}
```

9. 最后，我们需要实现一些在鼠标悬停或点击时触发的方法。在 `ObjectMouseInteraction` 类中添加以下方法：

```
public void OnPointerClick(PointerEventData eventData)
{
```

```
        Debug.LogFormat("{0} 被点击了！ ", gameObject.name);
    }

    public void OnPointerDown(PointerEventData eventData)
    {
        Debug.LogFormat(" 指针在 {0} 上按下！ ", gameObject.name);
        material.color = Color.green;
    }

    public void OnPointerEnter(PointerEventData eventData)
    {
        Debug.LogFormat(" 指针进入 {0} 的范围！ ", gameObject.name);
        material.color = Color.yellow;
    }

    public void OnPointerExit(PointerEventData eventData)
    {
        Debug.LogFormat(" 指针离开 {0} 的范围！ ", gameObject.name);
        material.color = Color.white;
    }

    public void OnPointerUp(PointerEventData eventData)
    {
        Debug.LogFormat(" 指针在 {0} 上松开了！ ", gameObject.name);
        material.color = Color.yellow;
    }
```

完成代码的编写后，我们就可以开始使用它了。我们将创建一个立方体并在此立方体上附加该组件，当鼠标与立方体交互时，立方体的颜色将会改变。

1. 打开“游戏对象”菜单，选择“3D 对象”→“立方体”。
2. 将新的立方体放置在摄像机可以看到的位置。
3. 将 *ObjectMouseInteraction.cs* 脚本拖放到立方体上。

运行游戏。当鼠标移动到立方体上方或点击它时，立方体的颜色会发生变化。

讨论

注意，摄像机的 Physics Raycaster 组件只能“看到”带有碰撞体的对象。没有碰撞体的对象将不会接收这些事件。此外，还可以通过查看 Event System 对象的检查器

来调试事件系统。在游戏运行时，它会显示关于光标位置以及当前被射线命中的对象的信息，如图 3-1 所示。

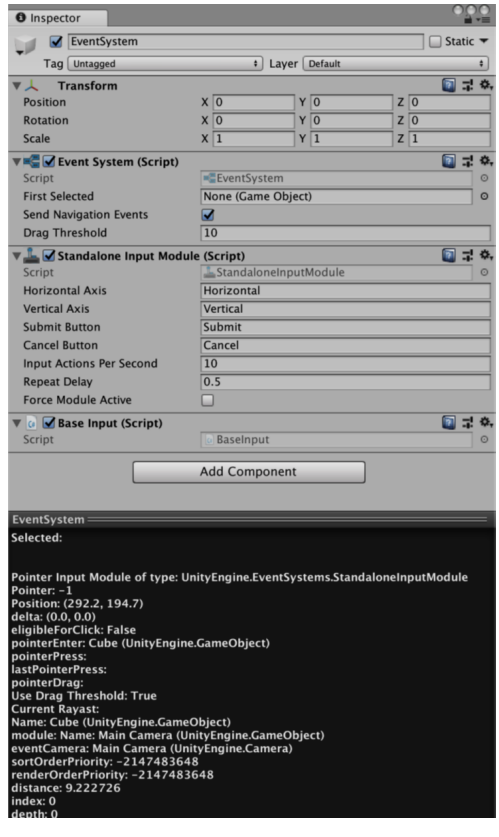

图 3-1：Event System 游戏对象的检查器中显示的调试信息