学习Unity和C#游戏编程（第2版）

[美] 凯西·哈德曼 / 著　周子衿 / 译
（Casey Hardman）

清華大學出版社
北京

内 容 简 介

本书作为一本面向初学者的 Unity 游戏编程入门书籍，旨在帮助读者从零开始掌握游戏开发的核心编程技能。全书分为 5 个部分，共 44 章，内容涵盖 Unity 基础、编程核心知识以及不同类型的游戏项目开发。第 1~5 章介绍 Unity 引擎的基础概念，帮助读者搭建开发环境，为后续学习打下基础。第 6~12 章深入编程核心，通过动手实践，让读者理解代码编写的基本原理和实际应用。第 13~25 章通过障碍赛游戏项目，让读者学习角色移动、关卡设计、预制件和脚本编写等基础技能。第 26~35 章通过构建塔防游戏，让读者掌握寻路机制和基础编程概念的深化应用。第 36~44 章通过开发游乐场游戏项目，让读者探索 Unity 物理引擎，实现复杂交互和 3D 物理模拟。

通过本书的系统学习和实践，读者不仅能掌握编程技巧，还能积累大量实践经验，为开发复杂游戏奠定扎实的基础。书中游戏项目涉及俯视角、塔防和 3D 物理模拟等，可以帮助读者全面掌握 Unity 引擎的用法。本书可以帮助游戏开发人员快速了解和掌握 Unity 与 C# 的核心知识以及关键技巧。

北京市版权局著作权合同登记号 图字： 01-2024-4305
First published in English under the title
Game Programming with Unity and C#: A Complete Beginner's Guide by Casey Hardman, edition: Second

图书在版编目（CIP）数据

学习 Unity 和 C# 游戏编程 : 第 2 版 / (美) 凯西・哈德曼 (Casey Hardman) 著 ; 周子衿译 . -- 北京 : 清华大学出版社 , 2025. 5. -- ISBN 978-7-302-68833-4

Ⅰ. TP311.5

中国国家版本馆 CIP 数据核字第 2025LK4720 号

责任编辑：文开琪
封面设计：李　坤
责任校对：方心悦
责任印制：杨　艳
出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.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：河北鹏润印刷有限公司
经　　销：全国新华书店
开　　本：185mm×210mm　　**印　　张**：19　　**字　　数**：616千字
版　　次：2025年5月第1版　　**印　　次**：2025年5月第1次印刷
定　　价：126.00元

产品编号：107906-01

译者序

作为知名的游戏开发引擎，Unity（又称“团结引擎”）自 2005 年诞生以来，凭借其易用性、强大的功能以及广泛的平台支持，迅速获得了全球游戏开发者的青睐。

许多游戏的背后功臣都是 Unity。这不仅包括小巧精致的独立游戏——比如以其精妙几何美学和梦幻般关卡设计闻名的《纪念碑谷》，还包括大型工作室开发的多人在线游戏——比如长盛不衰、玩家多达数亿人的《炉石传说》和《王者荣耀》等。基于 Unity 开发的众多游戏作品中，这些只是冰山一角。Unity 引擎的广泛应用不仅体现了它的技术实力，还展示了它在游戏开发领域的影响力。

Unity 引擎的强大之处在于它集成了编辑器、渲染器、物理引擎、动画系统和音频处理等多个核心组件，使得开发者能够在单一环境中完成从创意构思到成品发布的整个过程。更重要的是，Unity 支持 C# 编程语言，这意味着开发者可以借助丰富的 API 和脚本来实现复杂的游戏逻辑和交互。另外，Unity 的跨平台特性更是让游戏能够无缝接入 PC、移动设备、游戏主机乃至 Web 端，从而极大地拓展了游戏的受众范围。

《学习 Unity 和 C# 游戏编程》（第 2 版）不仅是一本面向初学者的全面参考和技术指南，更是一本聚焦于游戏开发的启蒙书籍。作者凯西 • 哈德曼凭借其丰富的编程和游戏开发经验，将复杂的技术概念以浅显易懂的方式呈现在读者面前。书中不仅详细介绍 Unity 的基础知识，还通过一系列精心设计的项目实践逐步引导读者掌握 Unity 的核心功能和 C# 编程技巧。无论是游戏开发的新手，还是有一定基础的开发者，都能从本书中获得丰富的知识和灵感。

本书的特点在于其实用性和系统性。从 Unity 的基础操作到复杂的脚本编程，从游戏对象的创建到用户界面的设计，每个环节都有详尽的讲解和示例。此外，本书还特别关注游戏开发的实战应用，通过分析和构建具体的游戏项目，使读者能够将所学知识应用到实际开发中，体验从 0 到 1 创造游戏的全过程。

本书中文版的截图均来自 Unity 2023.2.20f1 版本。截至目前，该版本的汉化仍不完全，为了方便读者与 Unity 界面对照，本书保留了未汉化的英文选项，同时在首次提到这些选项时标出对应的中文，例如 Layout（布局）。值得一提的是，Unity 2020.3.0f1 或其他一些版本的汉化较为全面，读者可以根据自己的需要自由选择。

衷心希望本书能够成为大家游戏开发旅程中的踏脚石，引导大家成功走向彼岸。游戏开发是一场充满挑战和机遇的冒险，希望大家和我一样，能够保持好奇心，能够一直在路上。

前言

欢迎来到 Unity 游戏编程探索之旅。本书将带领你从零开始电子游戏开发，通过丰富的实践练习来保持学习的热情。本书的重点不在于完成大型游戏项目，也不在于追求华丽的视觉效果，而是带领大家掌握编程技巧和深入了解 Unity 引擎的用法。对这些基础知识有了透彻的理解之后，你就可以进一步扩展自己的知识面，开发更加复杂和精美的游戏。

书中介绍的游戏类型虽然可能不是你最感兴趣的，但建议你最好还是按照章节顺序阅读本书。如此一来，你将能够系统地学习许多关键的编码技巧和实用窍门，相比随意跳过某些章节或独自挑战高难度的游戏项目，这种方式让人进步得更快。

Unity 是一个跨平台的游戏开发引擎，可以在 Windows、Mac 或 Linux 操作系统上运行。本书主要采用 Windows 相关的术语，不过即便你使用其他操作系统，应该也能轻松跟上学习进度，不会有太大的区别。

至于系统配置要求，近五年内购买的大多数现代计算机都能够轻松运行本书用到的软件。本书的示例项目不涉及复杂的图形处理或算法，所以它们应该能在大多数系统上流畅地运行。然而，太旧的系统可能导致 Unity 引擎运行缓慢，以至于影响使用体验。以下链接列出了 Unity 编辑器当前的长期支持版本（2022.3.6）的系统要求：

docs.unity3d.com/Manual/system-requirements.html

本书各章简要介绍如下。

第 Ⅰ 部分“Unity 基础”（第 1 章至第 5 章）介绍 Unity 游戏引擎的基础概念，并帮助你准备好所有需要的工具，为后续实践做好准备。

第 Ⅱ 部分“编程基础”（第 6 章至第 12 章）深入探讨编程的核心知识。你将开始动手编写代码并学习编程的基本原理。这几章将确保你不仅了解要编写什么代码，还能够明白为什么要写这些代码以及这些代码有哪些实际作用。

在本书的其余部分中，将逐一攻克不同的项目，制作可玩的游戏，你后续可以根据个人喜好来为这些游戏添加新特性或进行优化。通过学习这些部分的内容，你将积累大量实践经验。我们将实现真实的游戏机制，并解决初学者在游戏编程领域中可能遇到的各种挑战和难题。

第 Ⅲ 部分“游戏项目 1：障碍赛”（第 13 章至第 25 章）是一款俯视角游戏，玩家可以使

用 WASD 或方向键移动游戏角色，以避开各种形式的障碍：地面上巡逻和空中游荡的敌人、飞过的子弹和地面的尖刺陷阱。在开发这个项目的过程中，你将学习基本的移动和旋转操作、设计关卡，学习运用一些基本的 Unity 概念——比如预制件[①]和脚本，并编写基本的 UI（用户界面）。

第 IV 部分“游戏项目 2：塔防游戏”（第 26 章至第 35 章）要构建一个塔防游戏的基础框架。玩家需要在游戏地图上布置防御工事，以阻止敌人移到地图的另一端。这一部分要介绍基本的寻路机制（即敌人如何自动绕过任意障碍物），并在此基础上进一步深化对基础编程概念的理解。

第 V 部分“游戏项目 3：游乐场”（第 36 章至第 44 章）是一个支持第一人称和第三人称视角的 3D 物理模拟环境，这部分要实现更为复杂的鼠标控制移动、跳跃、蹬墙跳以及重力系统。在这个项目中，我们将探索 Unity 物理引擎的各种可能性，包括使用射线检测来识别游戏对象，以及设置关节和刚体等。

① 译注：prefab，又称“预制体”，这种资源用于存储游戏对象及其组件的配置和状态，是一种可重用的游戏对象模板。

简明目录

第 I 部分 Unity 基础

第 1 章 安装与设置 /002
第 2 章 Unity 基础 /008
第 3 章 操作场景 /016
第 4 章 父对象及其子对象 /022
第 5 章 预制件 /031

第 II 部分 编程基础

第 6 章 编程入门 /040
第 7 章 代码块与方法 /046
第 8 章 条件 /058
第 9 章 处理对象 /067
第 10 章 使用脚本 /080
第 11 章 继承 /089
第 12 章 调试 /103

第III部分 游戏项目 1：障碍赛

第 13 章 障碍赛游戏：设计与概述 /110
第 14 章 玩家移动 /116
第 15 章 死亡与重生 /135
第 16 章 基本款危险物 /142
第 17 章 墙壁和终点 /155
第 18 章 巡逻者 /162
第 19 章 漫游者 /181
第 20 章 冲刺 /191
第 21 章 设计关卡 /197
第 22 章 菜单和用户界面 /206
第 23 章 游戏内暂停菜单 /217
第 24 章 尖刺陷阱 /222
第 25 章 障碍赛游戏：总结 /232

简明目录

第 IV 部分 游戏项目 2：塔防游戏

第 26 章 塔防游戏：设计与概述 /240
第 27 章 摄像机的移动控制 /244
第 28 章 敌人与投射物 /254
第 29 章 防御塔和瞄准机制 /266
第 30 章 建造模式 UI/282
第 31 章 构建与出售 /293
第 32 章 游戏模式的逻辑 /314
第 33 章 敌人的逻辑 /323
第 34 章 更多类型的防御塔 /337
第 35 章 塔防游戏：总结 /349

第 V 部分 游戏项目 3：游乐场

第 36 章 游乐场：设计与概述 /356
第 37 章 鼠标瞄准摄像机 /359
第 38 章 进阶 3D 移动 /376
第 39 章 蹬墙跳 /392
第 40 章 推和拉 /399
第 41 章 移动的平台 /410
第 42 章 关节和秋千 /417
第 43 章 力场和弹簧垫 /424
第 44 章 结语 /430

详细目录

第 I 部分 Unity 基础

第 1 章 安装与设置 …… 002
1.1 轻量级应用 Unity Hub …… 002
1.2 安装代码编辑器 …… 003
1.3 安装 Unity …… 004
1.4 创建项目 …… 006
1.5 小结 …… 007
第 2 章 Unity 基础 …… 008
2.1 窗口 …… 008
2.2 “项目”窗口 …… 009
2.3 “场景”窗口 …… 009
2.4 “层级”窗口 …… 010
2.5 “检查器”窗口 …… 010
2.6 组件 …… 011
2.7 添加游戏对象 …… 013
2.8 小结 …… 015
第 3 章 操作场景 …… 016
3.1 变换工具 …… 016
3.2 位置和轴 …… 018
3.3 创建地板 …… 019
3.4 缩放和单位测量 …… 019
3.5 小结 …… 021
第 4 章 父对象及其子对象 …… 022
4.1 子游戏对象 …… 022
4.2 世界坐标与局部坐标 …… 024
4.3 构建简单的建筑物 …… 025
4.4 枢轴点 …… 027
4.5 小结 …… 029
第 5 章 预制件 …… 031
5.1 制作和放置预制件 …… 031
5.2 编辑预制件 …… 032
5.3 覆盖值 …… 033
5.4 嵌套预制件 …… 036
5.5 预制件变体 …… 037
5.6 小结 …… 038

第 II 部分 编程基础

第 6 章 编程入门 …… 040
6.1 编程语言和语法 …… 040
6.2 代码的作用 …… 041
6.3 强类型与弱类型 …… 042
6.4 文件扩展名 …… 043
6.5 脚本 …… 044

6.6 小结 045
第 7 章 代码块与方法 046
7.1 语句和分号 046
7.2 代码块 046
7.3 注释 047
7.4 方法 048
7.5 调用方法 051
7.6 基本数据类型 052
7.7 通过方法返回值 053
7.8 操作符 056
7.9 小结 057
第 8 章 条件 058
8.1 if 语句块 058
8.2 重载 060
8.3 枚举 060
8.4 else 语句块 061
8.5 else if 语句块 062
8.6 条件操作符 063
8.6.1 等于操作符 063
5.6.2 大于和小于 064
8.6.3 逻辑或操作符 065
8.6.4 逻辑与操作符 065
8.7 小结 066
第 9 章 处理对象 067
9.1 类 067
9.2 变量 068
9.3 访问类成员 070
9.4 实例方法 072
9.5 声明构造函数 074
9.6 使用构造函数 076
9.7 静态成员 077
9.8 小结 079
第 10 章 使用脚本 080
10.1 using 声明语句和命名空间 081
10.2 脚本类 082
10.3 旋转变换 084
10.4 帧与秒 085
10.5 属性 087
10.6 小结 088
第 11 章 继承 089
11.1 继承机制的应用：RPG 游戏中的物品系统 089
11.2 声明类 090
11.3 构造函数链 092
11.4 子类型和类型转换 095
11.5 类型检查 098
11.6 虚方法 099
11.7 数字值类型 100
11.8 小结 102
第 12 章 调试 103
12.1 设置调试器 103
12.2 断点 104
12.3 善用 Unity 官方文档 107
12.4 小结 108

第 III 部分 游戏项目 1：障碍赛

第 13 章 障碍赛游戏：设计与概述……110
13.1 游戏玩法概述……110
13.2 技术概览……111
13.2.1 玩家控制……112
13.2.2 死亡与重生……112
13.2.3 关卡……112
13.2.4 关卡选择界面……113
13.2.5 障碍物……113
13.3 项目设置……114
13.4 小结……115
第 14 章 玩家移动……116
14.1 创建 Player 游戏对象……117
14.2 材质和颜色……118
应用材质……120
14.3 声明变量……121
14.4 属性……123
14.5 跟踪速度……125
14.6 应用移动……130
14.7 小结……134
第 15 章 死亡与重生……135
15.1 启用与禁用……136
15.2 Die 方法……138
15.3 Respawn 方法……139
15.4 小结……141
第 16 章 基本款危险物……142
16.1 碰撞检测……142
16.2 Projectile 脚本……148
16.3 Shooting 脚本……151
16.4 小结……154
第 17 章 墙壁和终点……155
17.1 墙壁……155
17.2 终点……157
17.3 场景的构建设置……159
17.4 小结……161
第 18 章 巡逻者……162
18.1 巡逻点……162
18.2 数组……163
18.3 设置巡逻点……164
18.4 检测巡逻点……167
for 循环……169
18.5 巡逻点排序……171
18.6 Patroller 的移动……175
18.7 小结……180
第 19 章 漫游者……181
19.1 漫游区域……181
19.2 创建 Wanderer 游戏对象……184
19.3 Wanderer 脚本……184
19.4 处理状态……187
19.5 根据状态做出响应……188
19.6 小结……190
第 20 章 冲刺……191
20.1 定义变量……191
20.2 Dashing 方法……193
20.3 最后一步……195
20.4 小结……196
第 21 章 设计关卡……197
21.1 混用组件……197
21.1.1 四向射击装置……197

21.1.2 旋转刀片……198
21.1.3 旋转刀片木马……199
21.2 预制件和变体……200
21.3 创建关卡……202
21.4 添加墙壁……204
21.5 预览关卡的摄像机……204
21.6 小结……205
第 22 章 菜单和用户界面……206
22.1 UI 解决方案……206
22.1.1 IMGUI……206
22.1.2 Unity UI（uGUI）……206
22.1.3 UI Toolkit……207
22.2 场景流……207
22.3 LevelSelectUI 脚本……209
22.4 小结……215
第 23 章 游戏内暂停菜单……217
23.1 时间暂停……217
23.2 小结……221
第 24 章 尖刺陷阱……222
24.1 设计陷阱……222
24.2 Spike 的升降……225
24.3 编写脚本……226
24.4 添加碰撞体……230
24.5 小结……231
第 25 章 障碍赛游戏：总结……232
25.1 构建项目……232
25.2 玩家设置……234
25.3 回顾……235
25.4 额外特性……236
25.5 小结……238

第 IV 部分 游戏项目 2：塔防游戏

第 26 章 塔防游戏：设计与概述……240
26.1 游戏玩法概述……240
26.2 技术概览……242
26.3 项目设置……243
26.4 小结……243
第 27 章 摄像机的移动控制……244
27.1 准备工作……244
27.2 箭头键移动……247
27.3 应用移动……249
27.4 拖动鼠标进行移动……250
27.5 缩放……252
27.6 小结……253
第 28 章 敌人与投射物……254
28.1 图层和物理效果……254
28.2 基本敌人……256
28.3 投射物……259
28.4 小结……265
第 29 章 防御塔和瞄准机制……266
29.1 Targeter……266
29.2 防御塔的继承……273
29.2.1 Tower 类……273
29.2.2 TargetingTower 类……274
29.2.3 FiringTower 类……274
29.3 基类……274
29.4 箭塔……276
29.5 小结……281

第 30 章 建造模式 UI ····· 282
30.1 UI 基础知识 ····· 283
30.2 RectTransform 组件 ····· 286
30.3 构建 UI ····· 287
30.4 小结 ····· 292
第 31 章 构建与出售 ····· 293
31.1 事件 ····· 293
31.2 准备工作 ····· 294
31.3 建造模式的逻辑 ····· 298
31.4 字典 ····· 304
31.5 鼠标单击事件方法 ····· 307
31.6 小结 ····· 313
第 32 章 游戏模式的逻辑 ····· 314
32.1 出生点和目标点 ····· 314
32.2 锁定 PLAY 按钮 ····· 315
32.3 为寻路做准备 ····· 316
32.4 寻找路径 ····· 319
32.5 小结 ····· 322
第 33 章 敌人的逻辑 ····· 323
33.1 游戏模式设置 ····· 323
33.2 生成敌人 ····· 328
33.3 敌人的移动 ····· 331
33.4 小结 ····· 336
第 34 章 更多类型的防御塔 ····· 337
34.1 抛物线弹道 ····· 337
34.2 炮塔 ····· 343
34.3 高温地板 ····· 346
34.4 路障 ····· 347
34.5 小结 ····· 348
第 35 章 塔防游戏：总结 ····· 349
35.1 继承 ····· 349
35.2 Unity 的 UI 系统 ····· 350
35.3 射线投射 ····· 351
35.4 寻路 ····· 351
35.5 附加功能 ····· 351
35.5.1 生命条 ····· 352
35.5.2 护甲和伤害类型 ····· 352
35.5.3 更复杂的路径 ····· 353
35.5.4 攻击范围指示器 ····· 353
35.5.5 升级防御塔 ····· 354
35.6 小结 ····· 354

第 V 部分 游戏项目 3：游乐场

第 36 章 游乐场：设计与概述 ····· 356
36.1 功能概述 ····· 356
36.1.1 摄像机 ····· 356
36.1.2 玩家移动 ····· 356
36.1.3 推动与拉动 ····· 357
36.1.4 移动平台 ····· 357
36.1.5 关节和摆动 ····· 357
36.1.6 力场和跳板 ····· 357
36.2 项目的准备工作 ····· 358
36.3 小结 ····· 358
第 37 章 鼠标瞄准摄像机 ····· 359
37.1 创建 Player 游戏对象 ····· 359
37.2 工作原理 ····· 360
37.3 脚本设置 ····· 361

37.4 快捷键 366
37.5 鼠标输入 367
37.6 第一人称模式 370
37.7 第三人称模式 372
37.8 测试 375
37.9 小结 375
第 38 章 进阶 3D 移动 376
38.1 工作原理 376
38.2 Player 脚本 379
38.3 移动速度 383
38.4 应用移动 387
38.5 速度衰减 389
38.6 重力和跳跃 390
38.7 小结 391
第 39 章 蹬墙跳 392
39.1 变量 392
39.2 检测墙壁 394
39.3 执行跳跃 396
39.4 小结 398
第 40 章 推和拉 399
40.1 脚本设置 399
40.2 FixedUpdate 方法 402
40.3 检测目标 403
40.4 拉动和推送 405
40.5 绘制光标 407
40.6 小结 409
第 41 章 移动的平台 410
41.1 平台的移动 410
41.2 平台的碰撞检测 413
41.3 小结 416
第 42 章 关节和秋千 417
42.1 创建摆动装置 417
42.2 连接关节 421
42.3 小结 423
第 43 章 力场和弹簧垫 424
43.1 编写脚本 424
43.2 为力场创建游戏对象 425
43.3 向 Player 游戏对象添加速度 426
43.4 施加力 427
43.5 小结 429
第 44 章 结语 430
44.1 物理游乐场：项目回顾 430
44.2 Unity 进阶 431
44.2.1 资源商店 431
44.2.2 协程 432
44.2.3 脚本执行顺序 434
44.3 C# 进阶 434
44.3.1 委托 434
44.3.2 文档注释 435
44.3.3 异常 437
44.4 C# 语言中的高级概念 438
44.4.1 操作符重载 438
44.4.2 类型转换 439
44.5 泛型类型 439
44.6 小结 439

第 I 部分
Unity 基础

第 1 章　安装与设置
第 2 章　Unity 基础
第 3 章　操作场景
第 4 章　父对象及其子对象
第 5 章　预制件

第 1 章 安装与设置

安装软件的过程往往非常简单直观——下载安装包，然后运行安装程序，在弹出的“安装向导”中，接受使用条款，选择程序的安装路径，选择一些额外选项（如果有的话），就可以开始安装了。很简单，对吧？所以这里就不赘述安装过程了，我只说明需要安装哪些应用。

1.1 轻量级应用 Unity Hub

由于经常发布带有新特性、错误修复和小改进的新版本，所以 Unity 为此提供了一个名为 Unity Hub 的轻量级应用。通过 Unity Hub，不仅可以安装最新版本的 Unity 引擎，还可以安装和管理旧版本，以及集中查看所有 Unity 项目。

即使已经升级到最新版 Unity，也有必要保留一个旧版本的 Unity。有时可能需要始终使用同一个版本开发游戏，以防止新版本的特性与旧项目不兼容——技术在不断进步，新特性有时可能与原有的内容发生冲突。有时，旧有特性在新版本中会被重新设计，不再可用。在这种情况下，最好坚持使用旧版本开发项目，以免花费太多时间去适应新的版本。

为此，首先安装 Unity Hub，然后可以通过 Hub 来安装 Unity 引擎。为了下载 Unity Hub，请使用网络浏览器访问以下网址：

unity.com/download

页面上应该有一个下载按钮，该按钮的形状可能根据使用的操作系统而有所不同。我使用的是 Windows 操作系统，因此显示的按钮是“下载 Windows 版”，如图 1-1 所示。

图 1-1 下载 Windows 版

单击此按钮，下载 Unity Hub 安装程序。下载完成后，运行安装程序并按照提示进行操作。

在 Unity Hub 安装完成后，请运行。Unity 可能要求你接受一个许可协议并创建一个账户。这个设置是一次性的，一旦设置完毕，Unity Hub 基本上就会保持登录状态，不会频繁地要求你重新登录。不过，如果需要创建账户，请记住密码和用户名！

除非你供职于一家利用 Unity 开发产品并从中获得收益的大公司，否则就该选择免费的“个人版”（Unity Personal）许可证，然后继续执行下一步操作。Unity 将此许可证描述为适用于“过去 12 个月内收入和筹集资金少于 10 万美元的个人和小型组织。”

获得许可证并成功创建账户后，在 Unity Hub 界面的左侧，应该可以看到一个“安装量”标签，如图 1-2 所示。

图 1-2 选中“安装量”标签

可以在这里查看电脑上安装的所有 Unity 引擎版本，还可以在这里安装新版本。不过，安装太多版本可能会迅速消耗硬盘空间，因此可能需要卸载一些旧版本以避免这种情况。

在进一步讨论 Unity 版本之前，不妨先探讨一下如何选择代码编辑器。虽然在 Unity Hub 中可以在安装 Unity 的时候顺便安装代码编辑器，但根据不同的需求，市面上的其他代码编辑器可能更合适你。

1.2 安装代码编辑器

编写代码不是在平时用来写作和写简历的文字处理软件中进行的，而是在代码编辑器中进行的。代码编辑器是专为编写代码而设计的文本编辑器，它们有高亮显示关键字和符号的功能，能够理解和格式化代码，而且，它们通常内置许多功能，使我们能够更加高效和便捷地编写与处理代码。

其中一个重要的功能便是调试，调试允许代码编辑器在游戏运行时连接到 Unity 引擎。我们可以在代码中设置断点，当程序执行到断点时， Unity 编辑器中运行的游戏就会自动暂停。如此一来，我们便可以在游戏暂停的情况下查看代码的状态、单步执行代码以及随时恢复游戏的运行。这尤其适合用来定位代码中的问题。

下面简单看看几款可选的代码编辑器。

- Microsoft Visual Studio 是一款功能丰富且免费的编辑器，它支持 Windows 和 Mac OS。它可以直接通过 Unity Hub 安装，并且支持调试。但遗憾的是，它不支持 Linux 系统。Visual Studio 提供了用于商业和专业用途的付费版本，但对大多数业余爱好者而言，免费的 Community 版本完全够用了。
- Microsoft Visual Studio Code（简称 VS Code），是 Visual Studio 的轻量级版本，可以在 Windows、Mac OS 和 Linux 上运行。然而，VS Code 的最新版本并不支持 Unity 的调试功能，旧版本也只提供了实验性的调试支持。与 Visual Studio 相比，VS Code 的安装文件小得多，对电脑存储空间有限的用户来说，这无疑是一个加分项。
- JetBrains Rider 是一个支持 Windows、Mac OS 和 Linux 的编辑器，提供了 Unity 的调试支持。作为一款跨平台集成开发环境，它的功能与 Visual Studio 类似，但它不是免

费的。30 天免费试用期结束后，用户需要按月或按年订阅后才能继续使用该产品的最新版本。值得注意的是，一次性支付年费可以获得当前版本的永久使用权，但如果想要使用在此之后发布的新版本，则需要再次付费。

就个人而言，我作为 Windows 用户，经常使用 Visual Studio Code 来完成多种编程任务，因为我喜欢它轻巧简洁的设计理念。然而，如果在安装 Unity 时顺便通过 Unity Hub 安装 Visual Studio（注意，这不是 Visual Studio Code），就可以直接在 Windows 或 Mac OS 上调试 Unity。因此，我也安装了 Visual Studio，专门用于 Unity 游戏开发和 C# 编程。

第 12 章将介绍如何使用 Microsoft Visual Studio 来调试代码，所以如果你想按照第 12 章的指导进行学习，那么安装 Visual Studio 将是一个不错的选择。

调试虽然很有用，但如果你现在想保持简单，或者在使用 Linux 并且不打算付费使用 Rider，那么完全可以先选择 Visual Studio Code 作为代码编辑器，等到读完这本书的其他部分并开始独立探索时，再视情况而学习有关调试的知识。

总而言之，我的建议如下。

- 对于 Windows 和 Mac 用户，推荐使用 Visual Studio，因为它不仅支持通过 Unity Hub 轻松安装，还拥有强大的调试功能。但如果电脑存储空间比较紧张，我更建议选择更节省空间的 Visual Studio Code。
- 对于 Linux 用户，如果想保持简单，以后再学习有关调试的知识，那么 Visual Studio Code 将是一个不错的选择。但如果需要一套完整的开发工具，并且不介意在免费试用期后开始付费，那么选择 JetBrains Rider 更合适。

若想下载 Visual Studio Code，请在浏览器中输入以下网址：

code.visualstudio.com/download

请在这个页面上根据使用的操作系统（Windows、Mac 或 Linux）单击对应的下载按钮。安装程序下载完成后，运行程序并按照提示进行操作。

若想下载 JetBrains Rider，请在浏览器中输入网址：jetbrains.com/rider/。

在首页上可以看到几个显眼的“下载”按钮，单击其中的任意一个并按照提示进行操作即可。

1.3 安装 Unity

好了，是时候安装 Unity 编辑器了。在如图 1-2 所示的“安装量”标签页，单击右上角的“安装编辑器”按钮。随后会弹出一个窗口，其中列出可供选择的版本。

Unity 的版本号以发布年份为前缀，后跟一个句点和详细的版本号，例如 2021.2 或 2022.1。

最顶端的版本是最新的 LTS（Long Term Support）版本，即“长期支持版本”。LTS 版本适用于已经投入生产环境或即将发布的项目，它们不会引入任何破坏性更改，用户也不需要重构任何内容，这些版本会定期更新以修复小错误或提高性能和稳定性。

LTS 版本下方列出了最新特性和更新“其他版本”，这些版本的版本号通常比较新。

如果是与团队一起开发一个大型项目，最好使用 LTS 版本，以确保 Unity 引擎不会在开发过程中发生重大更改。如果只是在探索和学习或是想尝试最新的功能，则可以使用其他版本。甚至可以尝试使用 Beta 版本或 Alpha 版本，但请注意，这些版本可能相对不那么稳定。

考虑到我们的主要目的是学习，所以更合适的选择是“其他版本”一栏中的最新版本（位于 LTS 下方）。单击该版本号右侧的“安装”按钮。

接下来，Unity Hub 会提示选择一些要与 Unity 引擎一同安装的“模块”。这些模块为 Unity 添加了一些额外功能，但安装它们会占用更多的电脑存储空间。暂时不安装也没有关系，如果之后发现需要这些模块，随时可以添加。

Unity Hub 默认勾选了 Microsoft Visual Studio Community 模块。我们可以根据自己想要使用的代码编辑器选择保留勾选或取消勾选。

其他值得注意的模块是“平台”模块，有了它们，我们就可以把 Unity 游戏项目构建到不同的操作系统、环境和硬件平台上。

“构建”（building）游戏项目指的是将原本只能在 Unity 引擎中运行的项目转为实际的应用程序，玩家可以通过这个应用来玩游戏。

Unity 支持将项目构建到以下不同的平台上：

- 电脑，支持的操作系统有 Windows、Mac OS 和 Linux；
- Android；
- iOS；
- WebGL（在网页浏览器中玩游戏）；
- Xbox One；
- PS5 和 PS4；
- 任天堂 Switch。

Unity 还支持将项目构建到不同的扩展现实（Extended Reality，XR）平台。

正如之前提到的那样，如果未来需要构建到这些平台（本书不会讨论它们），随时可以通过 Unity Hub 安装它们。

此外，还可以选择将 Unity 官方文档作为一个模块下载到本地。这份文档对学习和解决问题很有帮助。文档也可以在线查看（我个人也倾向于这么做），但如果需要在没有网络连接

的情况下使用 Unity，在本地安装文档可能是一个明智的选择，以便离线查阅文档。

选好模块后，单击右下角的按钮。如果勾选了任何模块，这个按钮可能会显示“继续”，并会提供与所选模块相关的更多选项。如果不勾选模块，按钮会直接显示“安装”。

安装完成后，就可以使用所选版本的引擎创建 Unity 项目了。Unity Hub 还有一个贴心的功能：在打开一个项目时，它会自动识别并运行与该项目关联的 Unity 编辑器版本（前提是电脑上安装了这个版本）。这意味着，即便不同的项目使用不同版本的 Unity 引擎，也不需要记住每个项目对应的引擎版本，只需启动 Unity Hub，然后单击想要打开的项目，如此而已。

1.4 创建项目

现在，为了在学习过程中有一个可以实践操作的环境，是时候使用 Unity Hub 创建我们的第一个项目了。单击 Unity Hub 左侧的“项目”标签，然后单击右上角蓝色的“新项目”按钮。

此时会弹出一个对话框，可以在其中选择一个项目模板作为基础。

模板只是项目的一个简单的起点。前两个名为“2D”和“3D”的模板是最基础和简单的模板。它们基本上是空白的，其中“2D”适用于 2D 游戏，而“3D”适用于 3D 游戏。

我们要从一个空白的 3D 项目入手，所以请选择 3D 模板。选中后，它周围会显示蓝色边框[①]，如图 1-3 所示。

图 1-3 创建 Unity 项目

① 译注：在本书的描述中，我们会采用在电脑上所看到的颜色（不同于在纸质版图书中看到的灰度显示）。全书所有图片的彩色版可通过封底的“一书一码”方式免费下载。

对话框的右侧提供有关所选模板的说明。说明的下方有“项目设置”一栏，可以在此设置项目名称并选择项目的存储位置。

我们将把项目命名为“ExampleProject”。请注意，两个单词之间没有空格，因为文件路径通常不支持空格。

可以任意选择项目的存储位置，无论选择哪个路径，Unity 都会在那里新建一个与项目同名的文件夹。这个文件夹被称为项目的“根目录”，用于存放项目所有的文件和资源。

设置路径后，单击右下角的“创建项目”按钮，然后等待 Unity 创建基础项目文件，这可能需要几分钟时间。在创建完毕后，Unity 编辑器就会自动打开。这个时候，全新的项目已经准备就绪，我们可以开始编辑了。

1.5 小结

本章要点回顾如下。

1. Unity Hub 可以用来下载新版本的 Unity 编辑器、卸载不再需要的旧版本、创建项目以及打开现有的项目。
2. 通过 Unity Hub 打开项目会自动启动 Unity 编辑器，我们将在其中使用 Unity 引擎来开发游戏。
3. Unity 游戏项目存储在本地计算机上，所有相关文件——包括我们自己创建的美术资源和代码——都被存储在以项目名称命名的“根目录”中。
4. 编写代码时，我们将使用专为编码设计的文本编辑器，后者提供了普通文本编辑器所不具备的许多实用功能，可以让我们方便地格式化代码和浏览代码。

第 2 章 Unity 基础

在安装 Unity 并创建一个新的项目后，是时候探索这个引擎的用户界面了。毕竟，在开发游戏的过程中，我们将频繁与之交互，所以越早熟悉 Unity 越好。

2.1 窗口

Unity 的用户界面（UI）由不同的窗口[①]组成，每个窗口都有自己的用途。每个窗口的左上角都有一个小标签，上面标明了窗口的类型。这些窗口都是独立的，我们可以任意调整它们的位置和大小，甚至可以直接关闭窗口。Unity 中还有一些其他类型的窗口，我们可以根据自己的需要随时将它们添加并显示在屏幕上。

可以看到，如果用鼠标左键单击并拖动这些窗口的标签，我们便可以轻松地将窗口移动到程序的其他位置。通过这种方式，我们可以分割一个窗口的空间，把另一个窗口拖进来。如果想要改变窗口的布局，只需要拖动窗口，观察 Unity 如何响应。

图 2-1 两个窗口标签并列

此外，还可以将窗口挪到其他窗口旁边，将它们的标签并排放置（图 2-1）。“项目”窗口目前处于激活状态并在窗口中显示，但如果单击选中“控制台”标签，显示的窗口会变为“控制台”窗口。

Unity 还支持把当前所有窗口的大小和位置保存为自定义布局，并为这个布局命名。这个功能通过 Unity 编辑器右上角的 Layout（布局）下拉列表实现，单击即可查看所有可用的内置布局，如图 2-2 所示。

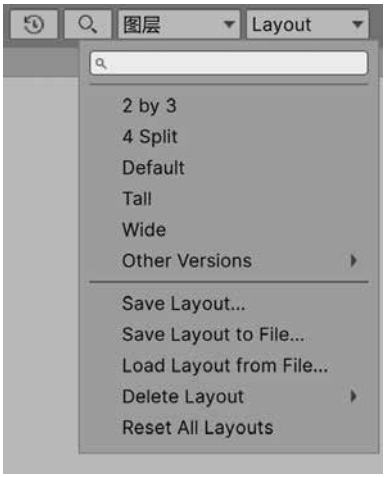

图 2-2 Unity 编辑器右上角的 Layout 下拉列表

① 译注：在 Unity 的 UI 元素中，窗口（window）是顶层容器，如“检查器”窗口；有自己的标题栏和边框，内部可包含多个面板（panel）。面板是用来组织 UI 元素的容器，可以包含其他 UI 组件（如按钮，文本或图像）。视图（view）通常指一个可视化的区域，用于显示内容，可以是面板的一部分。窗格（pane）是可滚动的区域，通常用于显示大量内容。

默认情况下，Unity 使用 Default（默认）布局。如果尝试切换到不同的布局，Unity 会自动重新排列各个窗口。如果按照自己的偏好对布局进行调整，则可以在下拉菜单中选择 Save Layout...（保存布局）选项，为当前布局命名并将它保存下来。如此一来，即使不小心关闭了一个窗口或意外打乱了布局，也可以通过加载保存的自定义布局来恢复原样。

布局在处理不同的游戏开发任务时尤其有用。不同的开发任务可能需要不同的窗口布局，而保存布局的功能能够让我们在不同工作流程之间轻松切换——简单点几下鼠标即可。

默认的布局包含所有重要的基本窗口，所以不更改布局也没关系。接下来，让我们来探索一下这些窗口的功能。

2.2 “项目”窗口

默认情况下，“项目”窗口位于编辑器底部的中间区域，与 “控制台”窗口相邻——这个窗口将留到以后介绍，因为它与代码密切相关。可以在“项目”窗口中查看所有资源。资源（asset，也称为“资产”）指的是可以在游戏中使用的各种元素，比如美术、音效、音乐、代码文件、游戏关卡等。

开始创建资源时，它们会显示在“项目”窗口中。“项目”窗口的工作原理与计算机的文件系统类似，允许我们将资源存储在文件夹（或称“目录”）的地方。例如，可以创建一个文件夹来存放各种音效，再创建一个文件夹来存放所有代码文件（在 Unity 中称为“脚本”），等等。我们可以在这个窗口中任意组织这些资源，并且查找、选择和在 Unity 引擎中使用资源都是通过这个窗口来进行的。

我们创建的项目默认包含两个文件夹：Packages 文件夹（这个暂时不需要关注）和 Assets 文件夹，这是存放所有资源的根目录。单击文件夹旁边的箭头可以隐藏或显示其中的内容（如果有内容的话）。如果展开 Assets 文件夹，你会发现里面已经有了一个 Scenes 文件夹，其中包含一个名为“SampleScene”的资源。

2.3 “场景”窗口

“场景”窗口占据了屏幕的大部分空间，通常位于左上方，与“游戏”窗口相邻。它展示了游戏所发生的环境。游戏中的“关卡”在 Unity 中通常被称为“场景”。场景以资源的形式保存，所以在保存场景后，我们可以在“项目”窗口中找到它们。当前场景是项目默认包含的 SampleScene 资源。

每个场景都包含一系列对象。可以通过“场景”窗口来查看场景并在其中移动，通过一个类似于漂浮摄像机的视角观察游戏世界。这是查看和编辑游戏环境的主视口（viewport）。

目前打开的示例场景空空荡荡的，只有一个光源和一个摄像机。光源是一个不可见的对象，负责为场景中的所有游戏对象提供光照；而摄像机则是玩家在游戏中用来观察场景的工具。场景中还展示了基本的天空和地平线效果。

随着开发进程的推进，我们可能会创建更多的场景。所有的场景文件都存储在 Scenes 文件夹中。可以双击加载其他场景，以便查看或编辑相应的场景内容。

在“场景”窗口中，按住右键并移动鼠标可以旋转视角，类似于第一人称游戏中的转视角操作。保持按住右键不放，可以使用 WASD 键来控制摄像机的移动：W 键让摄像机向前移动，S 键后退，A 键向左平移，D 键向右平移。也可以使用 Q 向下移动，E 向上移动。此外，按住鼠标中键并拖动鼠标可以平移摄像机，而不改变其朝向。

2.4 “层级”窗口

“层级”窗口默认位于左上角，其中显示当前场景中的所有对象。如前所述，一个场景本质上是多个对象的集合。在从一个场景切换到另一个场景，实际上是把前一场景中的所有对象隐藏起来，然后显示新场景中的所有对象。

“层级”窗口中列出了之前提到的场景中的两个对象：Directional Light（定向光）和 Main Camera（主摄像机）。场景默认包含一个光源和一个摄像机，所以可以在“层级”窗口中看到它们（这个摄像机不是“场景”窗口中用来查看场景的摄像机，而是代表玩家视角的摄像机）。

这些对象被称为 **GameObjects**（游戏对象）。简单来说，游戏对象就是场景中的某个对象。它可以是一个道具，比如一个宝箱、一株植物或一棵树；也可以是玩家角色、敌人、放在地方上的强化道具或无形的光源。游戏对象甚至可以什么都不做，只是无形地存在于场景中。在最简单的形态下，它们可能只是空间中的一个带有名称的点。

2.5 “检查器”窗口

“检查器”窗口默认位于 Unity 编辑器的右侧，是一个经常会用到的重要窗口。

正如上一节所说的那样，“层级”窗口列出了场景中的所有游戏对象。如果在“层级”

窗口中单击任何一个游戏对象，“检查器”窗口就会相应地展示该游戏对象的相关信息。“检查器”窗口的顶部会显示一个包含游戏对象名称的文本框，可以通过单击这个文本框来重命名游戏对象。此外，文本框下方还显示了两个下拉菜单，分别是“标签”和“图层”，我们稍后将会探索它们的用途。

“检查器”窗口的主要作用是显示所选游戏对象上添加的所有组件。

2.6 组件

在 Unity 中，组件（component）指的是附加到游戏对象上的游戏功能。组件不能独立存在，必须添加到游戏对象上。

Unity 引擎默认包含许多不同种类的组件，每种组件都有不同的用途。例如，光源组件用于照明，它们可以是像太阳那样照亮整个场景的光源，也可以是像手电筒那样的光束。

现在，让我们详细了解一下光源组件。由于“检查器”窗口主要用于展示所选游戏对象的组件，所以我们需要在“层级”窗口中单击 Directional Light 游戏对象选中它。选中后，“检查器”窗口将会刷新，显示如图 2-3 所示的内容。

在“检查器”窗口顶部的游戏对象名称等基本信息下方，有两栏，每一栏分别对应一个添加到游戏对象上的组件。图 2-3 中，这个游戏对象添加了两个组件：Transform（变换组件）和 Light（光源组件）。可以通过单击组件的标题栏（组件名称所在的位置）来隐藏（也称为“折叠”）或显示它们。组件标题下方列出了组件的各种属性，这些属性以可编辑的值字段的形式展现，可以通过修改这些值来调整组件，比如增强光线强度、改变颜色、调整

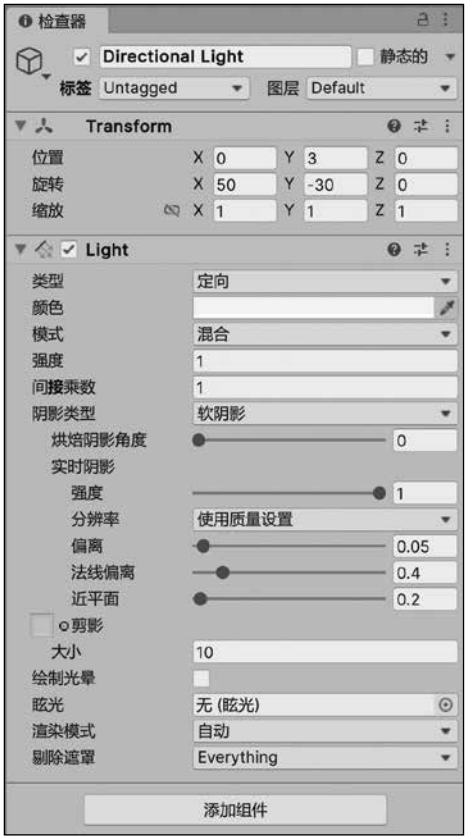

图 2-3 选中默认 Directional Light

阴影投射方式等。字段的名称显示在左侧，而字段的值则显示在右侧。Unity 中有许多种用于编辑不同类型数据的字段，这些字段值可能是数字，也可能是小的“滑块”，我们可以通过点击并拖动滑块来调整数值。

“检查器”窗口的主要功能是查看和编辑添加到游戏对象上的那些组件的属性。

另一个例子是 Camera（摄像机）组件。它是不可或缺的，其作用是在游戏运行时将场景渲染（“渲染”意味着“绘制到屏幕上”）到玩家的屏幕上。如果在“层级”窗口中选择 Main Camera 游戏对象，就可以在“检查器”窗口中看到 Camera 组件的详细信息。

在本书后面的章节中，我们将开始动手开发游戏，并尝试使用各种组件。Unity 提供了详尽的官方文档，单击“检查器”窗口中某一组件标题右侧的问号图标，即可在默认浏览器中打开该组件的文档页面。

游戏的代码文件也将以组件的形式添加到游戏对象上。在这种情况下，它们被称为“脚本”（Script）。添加到游戏对象上的脚本组件将负责执行其中的代码。因此，将代码整合到游戏中的方法是把它作为脚本组件附加到某个游戏对象上。

这意味着在编写和定义脚本时如果操作得当，我们将能够灵活地重用和组合不同的功能模块。

例如，第一个示例项目是一个障碍赛游戏，玩家必须避开各种障碍。在开发这样的游戏项目时，为了保持游戏的趣味性，可能需要设计多种障碍物，比如发射火球的障碍物、绕圈旋转的刀刃以及在两个固定点之间来回滚动的尖刺球。

这些不同的功能可以通过独立的脚本组件来实现：一个“射击”组件，用于定期在游戏对象前方发射火球；一个“旋转”组件，使对象不断旋转；还有一个“巡逻”组件，使对象能在两个或更多个点之间来回移动。然后还需要设计一个“危险品”组件，把它添加到火球、旋转刀刃和滚动的尖刺球，使其在接触到玩家时杀死玩家。

组件的一个炫酷之处在于，我们可以将不同的组件相互组合，创造出各种新型障碍物。

不同的功能模块封装在独立的组件中，所以任何游戏对象都可以具备射击、旋转、巡逻或在接触时杀死玩家的能力。而且，Unity 并不限制单个游戏对象上的组件数量，这意味着我们可以将射击和旋转组件添加到同一个游戏对象，从而创造出一个能够绕圈旋转的火球发射器。同理，还可以在这个对象上添加刀刃，甚至可以创造能够一边巡逻一边向前发射火球的尖刺球。

你应该已经掌握了重点：只要每个功能都封装在独立的脚本组件中，我们就可以轻松地将任意脚本组件的组合应用到一个游戏对象上，使其具备我们编写的各种功能。

这是 Unity 组件系统的主要优势之一。它提供了一个可以让我们自由组合不同功能的系统。

2.7 添加游戏对象

这一节将探索如何使用 Unity 创建和操作一些游戏对象。我们不会在 Unity 引擎中创建角色模型、宇宙飞船、枪支或其他花哨的东西。Unity 不是用于创建 3D 对象的建模软件，也不是用于创建 2D 对象的图像编辑器，它是一个游戏引擎，我们要在其他软件中制作模型和动画，然后将其直接导入 Unity 中。只需要将这些资源放入项目文件夹，Unity 就会自动处理它们，并让我们能够直接把资源拖到场景中。

为了专注于学习 Unity 引擎和编程，本书不会涉及复杂的美术工作。不过，Unity 中可以快速创建一些基本形状的游戏对象。Unity 编辑器的顶部（标题栏）下方分别是文件（File）、编辑（Edit）、资源（Assets）、游戏对象（GameObjects）、组件（Component）、服务（Services）、窗口（Window）和帮助（Help），单击即可展开包含更多选项的下拉菜单。[①]

可以使用“游戏对象”下拉菜单创建一些简单、常用的游戏对象，比如基本的 3D 形状、摄像机、光源等。

具体来说，可以通过选择“游戏对象”（GameObject）|“3D 对象”|“立方体”来创建一个如图 2-4 所示的立方体。或者，也可以在“层级”窗口中的任何位置单击右键（但不要单击现有游戏对象的名称），并在弹出的菜单中选择“3D 对象”|“立方体”。

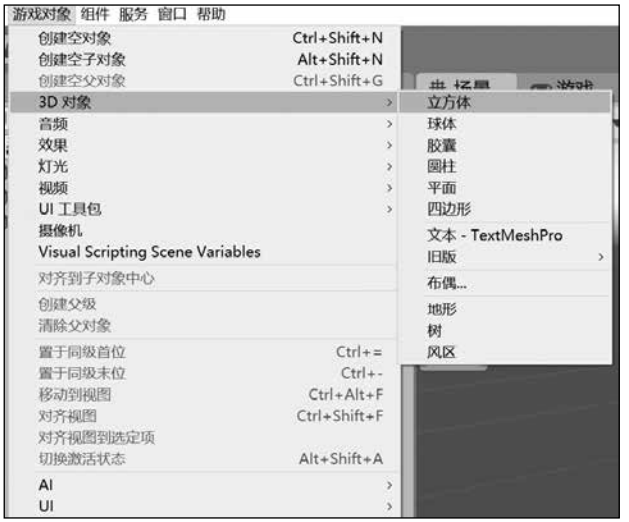

图 2-4 使用游戏对象菜单创建立方体

① 译注：由于 Unity（团结）引擎的汉化不完整，有些菜单中的命令还保留了英文，故在此提供英文名称供大家参考。

执行这些操作之后，就会看到“层级”窗口中出现了一个新的 Cube 游戏对象。此外，如果摄像机朝向它，我们就应该可以直接在场景中看到它。

如果我们没有在场景中看到这个立方体，则说明视图设置可能不正确。可以通过一系列简单的操作来调整视图，直接对准当前场景中的游戏对象：首先，在“层级”窗口中单击立方体对象选中它；其次，将鼠标悬停在“场景”窗口上使其成为焦点；最后，按下键盘上的 F 键。这是一个非常实用的快捷操作，它能够迅速将视图移动到选中的对象上，让我们可以清晰地看到它。如果在“场景”窗口中迷失了方向，距离游戏对象太远，那么简单使用这个快捷键即可快速定位。

为了扩展早些时候关于组件的讨论，让我们检查一下刚刚创建的这个新立方体中有哪些组件。确保立方体已经被选中，然后查看“检查器”窗口。

“检查器”窗口最上方的组件始终是 Transform（变换）组件。每个游戏对象都有一个 Transform 组件，后者决定了对象的位置、旋转和缩放，它对游戏对象来说是不可或缺的。其他组件都可以通过代码来动态地添加或移除，但 Transform 组件不可以。如果想移除 Transform 组件，就必须删除整个游戏对象。毕竟，场景中的每一个对象都需要有一个明确的位置，对吧？它总得存在于某处。

除了 Transform 组件，立方体还有其他的一些组件，比如 Mesh Filter（网格过滤器）和 Mesh Renderer（网格渲染器）。

在 Unity 中，“网格”一词几乎和“3D 模型”同义，我们可以把它看作是定义了构成 3D 模型的各个表面的资源。我们不会从零开始创建自己的网格，而是会使用 Unity 提供的默认形状，比如立方体、胶囊、球体和圆柱体。而“渲染”一词虽然听起来很花哨，但它的含义很简单，指的是在屏幕上绘制或展示图像的过程。

所以，网格渲染器是一个可以让 3D 模型绘制到屏幕上的组件，只不过实际绘制工作是由摄像机组件完成的。

网格过滤器用于存储要传递给网格渲染器的网格。它几乎总是与网格渲染器同时出现，因为过滤器负责指导渲染器应该渲染什么。

“检查器”窗口中的 Mesh Renderer 组件旁边有一个复选框。可以通过单击这个复选框来启用和禁用这类组件。如果想验证渲染器组件是否在绘制立方体到场景中，可以尝试单击复选框，若是取消勾选，立方体将立即从“场景”窗口中消失。再次勾选复选框后，它就会重新出现。

2.8 小结

本章要点回顾如下。

1. Unity 编辑器由多个窗口组成，每个窗口都有独特的用途。可以通过单击和拖拽窗口左上角的标签来重新排列它们或调整它们的大小。
2. 资源（Asset）是游戏中使用的文件，包括美术、音频和代码。这些资源都显示在“项目”窗口中，只需要简单地从“项目”窗口中拖放它们即可将其整合到游戏中。
3. 场景（Scene）资源代表一个游戏环境，例如一个单独的关卡。可以通过在“项目”窗口中双击场景资源来加载它们，加载之后，就可以在“场景”窗口中查看和编辑它们。
4. 游戏对象（GameObjects）是存在于场景中的对象。它们的功能由添加的组件来定义。Unity 提供的众多内置组件可以用来实现各种基本功能，比如渲染 3D 模型、投射光线、提供物理效果和碰撞检测等。
5. 可以通过“检查器”窗口来查看附加到游戏对象上的组件。在这个窗口中，可以通过编辑相关的字段来自定义组件的功能，例如调整光源的亮度。每个组件都是独立的实例，拥有自己的属性值，这些属性可以根据需要调整，以实现不同的效果。
6. 每个游戏对象都有 Transform 组件，这是一个不可或缺的组件，定义着对象的位置、旋转和缩放。其他类型的组件可以通过代码动态地添加和移除，但 Transform 组件不行，因为每个游戏对象只能有一个 Transform 组件，且该组件不能被删除。

第 3 章 操作场景

第 2 章介绍了 Unity 引擎中最重要的几个窗口，讲解了如何创建简单的对象并通过“检查器”窗口查看它们的组件。这一章将说明如何在“场景”窗口中对游戏对象进行移动、旋转和缩放等操作。

3.1 变换工具

第 2 章提到，Transform 组件（变换工具）是所有游戏对象都有的组件，它决定了对象在游戏世界中的位置、缩放和旋转状态。“场景”窗口中的变换工具让我们能够与选定游戏对象的变换进行交互。

变换工具按钮默认位于“场景”窗口的左上角，但也可以通过顶端的两条线在“场景”窗口内移动它们。此外，也可以将变换工具按钮拖放到“场景”窗口的标签栏中，与其他工具按钮一起固定在窗口顶部，如图 3-1 所示。

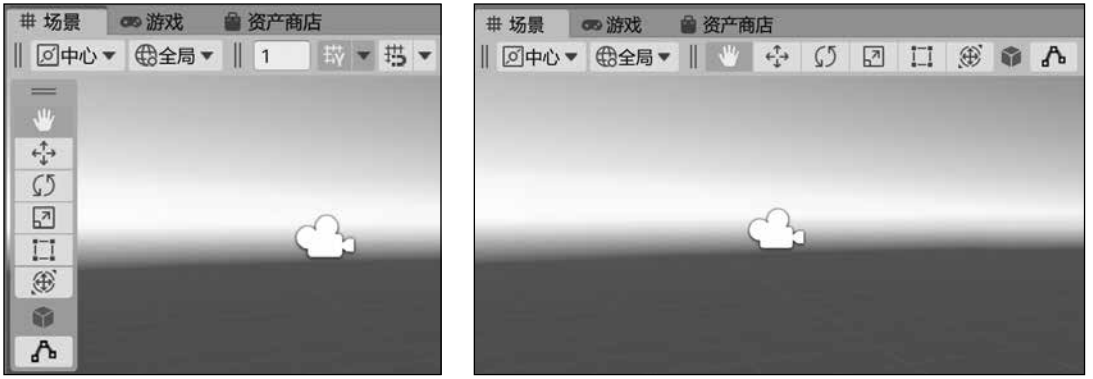

图 3-1 变换工具从“场景”窗口弹出（左图）并固定在窗口标签下（右图）

Unity 提供了 6 种变换工具来分别对应不同的操作功能，可以通过 6 个按钮快速切换。具体会显示哪些变换工具取决于当前选中的游戏对象，有些对象还有一个用于自定义编辑器工具的第 7 个按钮，但我们现在不必关注它，如果没有看到这个额外的按钮，也不用担心。

可以单击各个按钮切换到不同的工具。同一时间只能有一个工具处于激活状态，它们的

用途各不相同。

通过快捷键 Q、W、E、R、T 和 Y，可以从上至下（或从左至右）在这些工具之间快速切换。

第一个工具是手形工具，它对应快捷键 Q，可以通过它在“场景”窗口中单击左键并拖动鼠标来移动场景摄像机。它不是用来编辑场景的，而是用来查看场景的。此外，按住鼠标中键拖动场景时，手形工具也会自动激活。

其他工具则可以用来编辑选中的游戏对象。在“场景”窗口中，不同的变换工具将在选定的游戏对象旁边显示相应的辅助图标（Gizmo）。可以通过单击和拖动这些辅助图标来使用变换工具以及与游戏对象交互。选中一个游戏对象并在这些工具之间切换，可以看到围绕该对象显示的辅助工具图标会随着所选工具的改变而变化。

快捷键 W 激活的是移动工具，它会在选中的游戏对象上显示箭头状的辅助图标。通过拖动这些箭头，可以沿着特定的方向移动对象。如果想同时沿着两个坐标轴移动对象，只需要点击并拖动箭头之间的矩形区域即可。

快捷键 E 对应的是旋转工具，它会在选中的游戏对象上显示圆形辅助工具。可以通过单击并拖动这些圆圈来让对象沿不同的轴进行旋转。如果拖动辅助工具中心，也就是圆圈之间的区域，对象可以同时沿多个轴进行旋转。此外，拖动最外层的灰色圆圈可以让对象相对于当前摄像机视角进行旋转。

快捷键 R 则对应着缩放工具，它的辅助工具类似于箭头，但端头是立方体。可以通过单击并拖动这些立方体箭头来分别调整对象的宽度（红色）、长度（蓝色）或高度（绿色）。拖动辅助工具中心的立方体可以对整个对象进行等比例缩放——即均衡地增加或减少对象的宽度、高度和长度。

快捷键 T 对应的是矩形工具。虽然这一工具主要用于 2D 项目，但在 3D 场景中同样有着独特的作用。所选对象周围会显示一个矩形的辅助工具，矩形的四个角上有圆圈。可以通过拖动矩形的边或角来使对象按照矩形的比例进行扩大或缩小，这一操作会同时影响对象的位置和缩放。这尤其适用于只调整对象单侧大小而不改变另一侧的情况，因为缩放工具默认会对对象的所有侧面进行等比例缩放。

辅助工具中心还有一个圆圈，可以单击并以沿矩形对齐的两个方向拖动对象。矩形辅助工具的操作始终为这两个方向。如果将视角移动到游戏对象的另一侧，矩形辅助工具将会随之翻转，再次正对摄像机。

快捷键 Y 对应的是变换组件工具，它结合了快捷键 W、快捷键 E 和快捷键 R 所对应的工具，同时显示用于移动位置的箭头、用于旋转的圆圈以及用于缩放的中心立方体。

3.2 位置和轴

3D 空间中的位置是如何定义的呢？由三个坐标值来定义，即 X 轴、Y 轴和 Z 轴的数值。

- X 轴代表水平方向，控制对象的左右位置。
- Y 轴代表垂直方向，控制对象的上下位置。
- Z 轴代表深度方向，控制对象的前后位置。

这些坐标位置通常以（X，Y，Z）的形式表示。例如，（15，20，25）的意思是 X 位置的值为 15，Y 位置的值为 20，Z 位置的值为 25。

如果坐标位置是（0，0，0），就意味着对象处于“世界的原点”，也就是宇宙的中心，或者至少是场景的中心。

- 在 X 位置的值上加 5，对象就会向右移动 5 个单位。
- 从 X 位置的值上减 5，对象就会向左移动 5 个单位。

Y 位置的值和 Z 位置的值同理：增加值使对象沿着相应的轴向正方向移动，而减少值则使其向相反方向移动。

- 增加 Y 位置的值使对象向上移动，减少它则使对象向下移动。
- 增加 Z 位置的值让对象向前移动，减少它则让对象向后移动。

这三个值的共同定义了对象在游戏世界中的位置。每一个值都对应一个轴，所以也有人会称其为“X 轴”或“Y 轴”或“Z 轴”。

缩放和旋转也基于相似的原理：它们同样沿着这三个轴进行，每个轴代表不同的方向。

- X 缩放代表宽度，左右方向。
- Y 缩放代表高度，上下方向。
- Z 缩放代表长度，前后方向。

不难猜到，旋转的工作方式和移动很相似。对象的方向由三个轴上 0 到 360 之间的角度值定义。

在 Unity 中，移动、旋转和缩放工具（快捷键分别是 W、E 和 R）对应的辅助工具都有颜色编码，X 轴总是用红色表示，Y 轴总是用绿色表示，而 Z 轴总是用蓝色表示。

这种颜色编码几乎算得上是通用标准。如果使用其他软件制作 3D 模型，应该会看到相同的颜色编码，尽管有些软件可能与 Unity 中的定义相反，将 Y 轴用于控制前后方向，Z 轴用于控制上下方向。

3.3 创建地板

现在，让我们应用前面学到的知识来创建一些立方体，并对它们进行移动和缩放。不过，在开始之前，先来制作一个地板。按照与上一章创建立方体类似的方法来创建一个平面：“游戏对象”|“3D 对象”|“平面”。

可以将平面看作是立方体的一面——一个没有厚度的平面。这些平面是单面的，从背面看是完全不可见的。如果试着将摄像机移动到平面的下方并向上看，会发现什么也看不到，就像这个平面完全不存在一样。但是，把它用作地板还是很合适的，因为我们不太可能从背面观察它。

地板应该放在哪里是显而易见的，这可以通过“检查器”来设置。在“层级”窗口中选择新建的 Plane（平面）对象，然后在“检查器”窗口中查看它的 Transform 组件。

如前所述，变换组件包含三个部分：位置（游戏对象在游戏场景中的具体位置）、旋转（游戏对象的旋转角度）和缩放（游戏对象的大小）。

请记住，“检查器”窗口的主要目的是与组件交互，而不仅仅是查看它们的数据：它允许我们直接编辑变换组件的位置、旋转和缩放的具体数值。只需要单击相应的字段并输入期望的数值，就能够轻松地调整各个轴的参数。

如果知道要设置的确切数值，这种设置方式就非常有用，因为使用变换工具来把位置和旋转调整到精确的值很麻烦。我们想让这个地板位于世界原点（即场景的中心），所以如果还没有把它的三个位置都设为 0 的话，可以通过“检查器”窗口进行更改。至于旋转，应该已经是（0，0，0），所以保持不变即可。

3.4 缩放和单位测量

先来谈谈“缩放”这个概念。有人可能会问：“空间的单位是什么？”把一个游戏对象的位置从 0 改为 1 时，实际上意味着什么？它移动的空间是多少？

这个概念可能会让一些人感到困惑。你可能期待得到一个明确的答案。Unity 开发者应该已经定义了它，对吧？它可能是 1 英尺（12 英寸，30.48 厘米），也许是 1 米，或者是 1 码（91.44 厘米）。

但事实并非如此。不过也不要担心，这仍然很容易理解，因为我们必须自行决定一个单位代表什么。举例来说，我们可以规定一个单位代表 1 英尺。只要在每次测量时都遵循这一标准，那么 1 单位就代表 1 英尺。还可以根据这个标准来设计人物的身高，这大约在 5 到 6 个单位之间。如果创建一个 1 英寸（2.54 厘米）大小的对象，就把它设置成一个单位的十二分之一

（大约 0.083 个单位）。如果想创建一个 1 码长的对象，就把它设置为 3 个单位。

不过，还有一个与组件的缩放有关的事情需要注意，那就是缩放值并不代表对象在宽度、长度和高度上各占多少单位，它实际上是一个系数，用于与网格（3D 模型）的尺寸相乘。

网格本身有自己的尺寸，而 Transform 组件的缩放值会与这个尺寸相乘。

对于立方体而言，这一点很容易理解。立方体的网格的宽、高和长均为 1 个单位。如果将其缩放值设置为 5，那么它在每个轴上都将是 1 单位的 5 倍，所以立方体的尺寸将是 5 个单位。

但对平面来说就有些麻烦了。平面网格宽度和长度都是 10 个单位（并且由于平面非常薄，所以它的高度可以忽略）。这意味着，当平面网格的缩放值被设置为（1，1，1）时，它的长和宽实际上都是 10 个单位。

如果创建一个平面和一个立方体，将它们的缩放值都保留为默认值（1，1，1），并将它们放到同一位置，你会发现平面比立方体大得多，如图 3-2 所示。

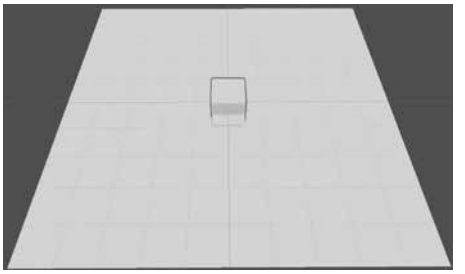

图 3-2 具有相同缩放值的平面和立方体放在完全相同的位置

这是因为它们的实际网格大小不一样。尽管缩放相同，但立方体网格的宽度和长度是 1 个单位，而平面则是 10 个单位。由于缩放值只是网格大小的一个系数，而不是游戏对象的实际大小，（1，1，1）的缩放值并不会改变网格的实际尺寸，它只是简单地将尺寸乘以 1，也就是保持原尺寸不变。

现在，如果将立方体的比例值改为（10，1，1），它的宽度将变成与平面一样的 10 个单位，如图 3-3 所示。

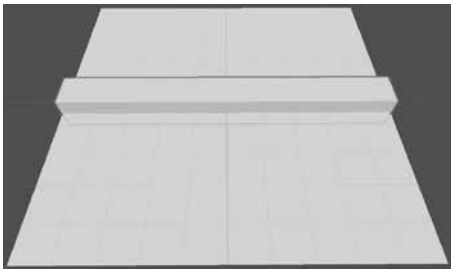

图 3-3 具有不同缩放值的平面与立方体放在同一个位置

总而言之，只需要记住一点：网格有自己的大小，缩放值只是网格大小的一个系数，而不是直接用于描述网格大小的值。

现在，让我们继续完成地板的设置工作。地板最好大一点，以便日后能方便地放置更多游戏对象。为此，需要将地板在 X 轴和 Z 轴上的缩放值设置为 10——注意，这实际上意味着地板的长度和宽度将增加到 100 个单位。当然，为了方便起见，这里最好不要使用缩放工具，而是通过“检查器”窗口来将 X 轴和 Z 轴的缩放值设置为 10。

3.5 小结

本章要点回顾如下。

1. 可以使用变换工具（快捷键 W、快捷 E 和快捷 R）来调整游戏对象的位置、旋转和缩放。
2. 游戏对象的位置通过 X、Y 和 Z 三个坐标值来表示。增加某个值会使其朝某一方向移动，而减少某个值会使其朝相反的方向移动。X 代表水平方向，向右为正，向左为负；Y 代表垂直方向，向上为正，向下为负；Z 代表深度方向，向前为正，向后为负。通过将这三个值组合在一起，可以精确地在三维空间中定位一个点。
3. 缩放是用于游戏对象的实际网格大小的系数，而不是游戏对象的宽、高和长。变换组件的 X、Y 和 Z 缩放值会与网格的原始尺寸相乘，以调整游戏对象的大小。
4. 默认情况下，单独的一个单位不对应于特定数量的英尺、英寸或米。我们必须自行决定一个单位代表什么，并在整个 Unity 项目中保持一致性，确保所有对象的大小比例是正确的。

第 4 章 父对象及其子对象

在设置好地板后，是时候探索 Unity 游戏引擎中一些重要的概念了。Unity 提供了一种被称为“父子关系”的机制，允许不同的游戏对象通过这种层级结构相互关联，其中“子对象”附加到“父对象”上，并随着父对象的移动、旋转和缩放而移动、旋转和缩放。这提供了两种方式来定义一个对象的位置：一是世界位置（world position），表示对象在场景中的位置；二是局部位置（local position），表示对象相对于其父对象的位置。此外，通过设置父子关系，还可以确定旋转的枢轴点（pivot point），以此来改变对象旋转时的中心点。

4.1 子游戏对象

“层级”窗口之所以被称为“层级”，有其特定的原因。虽然前面没有详细说明，但下文很快就会见分晓。

在 Unity 中，一个游戏对象可以包含任意数量的其他游戏对象，这种包含关系被称为“父子关系”（parenting）：一个游戏对象可以是多个其他游戏对象的父对象（parent），而这些被包含的游戏对象则被称为它的“子对象”（children）。

从技术上讲，Unity 通过变换组件的相互关联来实现父子关系，这是该概念的核心。将一个游戏对象设置为另一个游戏对象的子对象，相当于将它以物理方式附加到父对象上。因为变换组件负责处理游戏对象的位置、旋转和尺寸，所以这本质上意味着将两个变换组件绑定在一起。

当父对象移动时，子对象也会随之移动。当父对象进行旋转时，子对象会相应地围绕父对象的旋转中心转动，即使它们之间相隔很远。当父对象变小或变大时，子对象也会按比例跟着变化。

现在，让我们通过实际操作来看看这具体是如何运作的。

创建两个立方体，随意地将它们放置在任何位置，只要不完全重叠在一起即可——如果需要，它们可以紧挨着彼此或部分重叠。可以使用位置变换工具（快捷键 W）来调整它们的位置。

现在，将其中一个立方体放大。可以在“检查器”窗口中将它在各个轴上的缩放设置为 1.5，或是直接用缩放变换工具（快捷键 R）把它拉大一点。完成后，这个立方体应该与图 4-1 类似。

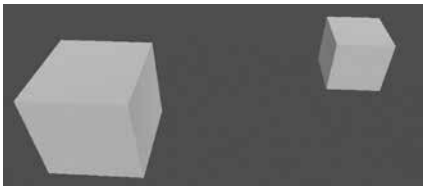

图 4-1 缩放后的立方体（左）和保持原始大小的立方体（右）

将较大的立方体设置为父对象。选中它后可在“层级”窗口中查看它是哪个 Cube（在选中后，它将被高亮显示）。然后，在“检查器”窗口中将其名称更改为 Parent，这样一来，就更容易区分了。

现在，单击“层级”窗口中的另一个 Cube 选中它，然后在“层级”窗口中将它拖放到 Parent 立方体上。

拖放完成后，就会看到“层级”窗口终于呈现出“层级”关系。被拖动的 Cube 现在成为 Parent 立方体的子对象，并且它在“层级”窗口中被包含在其父对象之内。这种关系是通过缩进来显示的：子对象会向右缩进一格。图 4-2 展示了父立方体、其内部的子立方体以及另一个不是子对象的游戏对象。注意，Cube 子对象向右缩进了一格，表示它是 Parent 的子对象。Cube 游戏对象是 Parent 游戏对象的子对象。第三个游戏对象不是子对象，这可以通过缩进来判断。

图 4-2 “层级”窗口中显示的立方体

此外，父对象左侧出现了一个小箭头图标，可以单击它在“层级”窗口中展开或折叠（收起）它的子游戏对象。折叠结构复杂的层级可以使窗口更加整洁。

由于 Cube 已经成为父对象的子对象，所以对父对象进行的任何变换操作都会影响到子立方体。子对象的变换不会影响到父对象——无论是移动、旋转还是缩放子对象，都不会对父对象造成任何影响。但对父对象进行的任何移动、旋转或缩放都将影响到子对象。

为了更好地理解这一点，我们不妨亲自实践一下。首先，在“层级”窗口中选择 Parent 立方体，然后在“场景”窗口中利用变换工具（快捷键 W、快捷 E 和快捷 R）对立方体进行移动、旋转和缩放。如果发现变换工具的辅助工具显示在两个立方体之间，那么可能是因为 Tool Handle Position（工具控制柄位置）被设置成了“中心”。按 Z 键将其设置为“轴心”，然后辅助工具应该会在 Parent 立方体上居中显示。

在移动和旋转父对象时，子对象将随之移动并围绕其旋转，就像有一根看不见的杆将它们连接在一起。在缩放父对象时，子对象也会相应地放大或缩小，并且它们之间的距离仍将保持相同的比例。

层级并不局限于简单的父子关系，还可以构建出复杂的层级，比如孙子、曾孙等——想设置多少层就可以设置多少层。可以添加另一个立方体并使其成为子立方体的子对象，这意味着它将成为 Parent 立方体的孙子。如此一来，移动 Parent 立方体时，其子对象和孙子对象也会随之移动。

如果想“解除”子对象的父子关系（使其不再拥有父对象），就可以在“层级”窗口中

把子对象拖到父对象的上方或下方。请注意，在“层级”窗口中拖动游戏对象时，鼠标经过的任何其他游戏对象都会以浅蓝色高亮显示，表示我们正在将被拖动的游戏对象（即，被“捡起来”的对象）指定为高亮显示的游戏对象的子对象。如果鼠标指针下方没有游戏对象，就会显示一条带有圆圈的蓝线。这就是在“层级”窗口中移动游戏对象的方式：可以解除它的父子关系，将它重新放置到另一个父对象的子代中，或是更改它在现有子代中的位置。

请留意蓝线左侧的圆圈：在将游戏对象放置在另一个对象的子对象中时，这个圆圈会向右边进一步缩进。在游戏对象的层级结构比较复杂且具有多个级别的子对象时，更容易观察到这一点。就当前解除对象间父子关系的需求而言，只需要简单地将子对象拖到其他对象下方的空白区域中并松开鼠标，使其没有父对象即可。

4.2 世界坐标与局部坐标

现在，让我们来了解一下世界坐标和局部坐标之间的区别。当一个游戏对象具有父对象时，它的位置可以用两种方式来表示。

- 世界位置：该对象在场景中的绝对位置。
- 局部位置：该对象相对于父对象的位置。

如果一个对象没有父对象，“检查器”窗口中显示的位置就是世界位置，而如果有父对象，这个位置就会是局部位置。

在世界坐标中，位置为（5，0，0）意味着“在世界中心的右侧 5 个单位”。

但是在局部坐标中，同样的位置意味着“在父对象的右侧 5 个单位”。这与父对象的旋转相对应。注意，这里说的是“在父对象的右侧”，而不是“右侧”。在旋转父对象时，子对象的局部位置不会改变。在 X 轴上加 1 并不会使对象在世界空间中向右移动 1 个单位，而是会使对象相对于父对象向右移动 1 个单位。

基于此，在某些情况下，我们就可以使用局部坐标而不是世界坐标来表示方向。举例来说，在一个玩家可以旋转视角（比如通过鼠标来控制角色的朝向）的游戏中，当玩家角色发射了子弹时，子弹应该向玩家角色局部的前方射出，而不是世界坐标系中的前方。

为了形象地理解这一点，我们不妨将世界方向想象成指南针上的方位。可以将世界坐标系中的“前进”方向类比为北方（Z 轴正方向），“后退”方向类比为南方（Z 轴负方向），“向右”为东方（X 轴正方向），“向左”为西方（X 轴负方向）。想想看，从枪口射出的子弹或者魔术师手中喷出的火焰不是应该一直“向北”发射，而是应该是沿着枪口或手掌指向的局部方向发射，对吧？

局部位置的另一个复杂之处在于，它会受到父对象的缩放值影响。这意味着如果父对象的缩放值不是（1，1，1），那么在位置上增加 1 个单位时，增加的单位实际上并不是 1 个。局部位置会与父对象的缩放值相乘。举例来说，如果父对象的 X 位置缩放为 2，那么在 X 轴上增加 1 个单位实际上是在世界空间中增加了 2 个单位。

4.3 构建简单的建筑物

接下来，让我们使用立方体来搭建一个类似建筑物的结构——一个方形的摩天大楼，外部没有任何装饰，只有平坦的表面。我们将学习如何在创建对象时定位它们，并在此过程中练习使用父子关系的概念。

图 4-3 Skyscraper 的外观

这个建筑物由三个立方体组成。底部的立方体将作为基座，它将更厚且更短。中间和顶部的立方体则依次比下一层更细长、更高。我们将使这些立方体居中，使它们的 X 位置和 Z 位置（即前后和左右）相同，同时通过上下移动它们，形成一个类似塔状的结构，就像堆叠起来的积木一样。完成之后，它的外观将类似于图 4-3 所展示的那样。

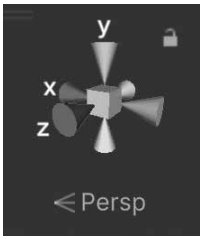

图 4-4 右上角的辅助工具

首先，在开始执行这项任务之前，请确保自己没有在“场景”窗口中迷失方向。可以使用右上角的辅助工具来观察摄像机是如何旋转的，它几乎就像是一个指南针（图 4-4）。箭头对应的轴旁边是其名称，且每个箭头都按照对应的轴进行颜色编码。

- X 轴是红色的，对应左右方向。增加 X 值向右移动，减少 X 值向左移动。
- Y 轴是绿色的，对应上下方向。增加 Y 值向上移动，减少 Y 值向下移动。
- Z 轴是蓝色的，对应前后方向。增加 Z 值向前移动，减少 Z 值向后移动。

这些彩色箭头都指向它们所对应的轴的正方向。灰色箭头则指向相反的方向，也就是轴的负方向。换句话说，彩色箭头指向实际的世界方向，即右、上和前。因此，对应 Y 轴的绿色箭头始终指向上方。如果它指向下方，就意味着摄像机是倒置的。辅助工具展示了真正的“上”在哪个方向。如果绿色 Y 轴箭头没有像图 4-4 中那样指向上方，请按住鼠标右键并移动鼠标来重新调整摄像机。

现在，创建一个立方体。确保选中它，并在“检查器”窗口顶部将其名称改为“Cube

Base”。我们会使它厚重但较短。在“检查器”窗口中将它的缩放值设置为（10，4，10）。这意味着 X 轴和 Z 轴为 10，但 Y 轴（高度）仅为 4。

在创建游戏对象时，有几种方法可以确保一个对象与另一个对象对齐。我们创建的每个对象都放置在摄像机前方的某个特定位置。如果在创建第一个立方体后没有移动过摄像机，那么在创建第二个立方体时，它们将位于相同的位置。

如果使用过前文介绍的快捷方式，即按下 F 键将摄像机聚焦于选定的对象上，那么也相当于间接设定了摄像机的位置，使得任何新建的对象都与聚焦对象处于同一位置。这样做的好处是，新建的对象会出现在聚焦对象的正上方。

因此，如果在创建 Cube Base 后移动过摄像机，请确保 Cube Base 被选中，然后将鼠标悬停在“场景”窗口上并按下 F 键。这将使 Cube Base 成为焦点对象，摄像机将会移动到能看到它的位置。此时创建的任何新对象都会和 Cube Base 位于同一位置。试着创建一个新的立方体，它应该出现在相同的位置上。或者也可以直接在“检查器”窗口中把 Cube Base 的位置值复制下来，粘贴到新建立方体的位置中。

但是，考虑到所有游戏对象最终都会连接到一起，我们无论如何都会用到父子关系。将上方的立方体连接到下方的立方体是很合理的，对吧？因此，在创建新的立方体时，可以将其命名为“Cube Middle”，并使它成为 Cube Base 的子对象。这样一来，就可以更轻松地判断它们是否正确地对齐了，因为“检查器”现在显示的位置是 Cube Middle 的局部位置。如果两个立方体处于同一位置，Cube Middle 的位置将会是（0，0，0）。

Cube Middle 需要在 X 轴和 Z 轴上居中对齐，所以如果它的 X 位置和 Z 位置值不是 0，请将它们设置为 0。接下来，可以直接使用位置（使用 W 键）和缩放（使用 R 键）变换工具将 Cube Middle 移动到 Cube Base 上方，并根据需要调整其大小。

可以对第三个立方体 Cube Top 执行同样的操作。可以直接对 Cube Middle 进行复制粘贴（快捷键 Ctrl+C 和 Ctrl+V）和重命名，使其与 Cube Middle 处于同一个位置；或者也可以创建一个新立方体，将其局部位置设置为（0，0，0），使其居中。注意，Cube Top 需要被设置为 Cube Middle 的子对象，而不是 Cube Base，因为 Cube Top 与中间的立方体相连，也就是说，在旋转或移动中间的立方体时，顶部的立方体应该跟随着它一起旋转或移动。

可以再次调整 Cube Top 的缩放和位置，就像之前对其他立方体所做的那样。保持其局部 X 位置和 Z 位置为 0 以使它居中，并把它放在 Cube Middle 上面。

就这样构建一个由 Cube Base 作为父对象、Cube Middle 作为中间层、Cube Top 作为顶层的层级结构。可以根据自己的喜好来调整这些立方体的缩放，如果希望它们的外观与图 4-3 中展示的摩天大楼版本相匹配，就可以如下设置它们的世界缩放值：

- Cube Base：（10，4，10）
- Cube Middle：（5，6，5）
- Cube Top：（2，5，2）

注意，这里说的是“世界”缩放。和位置一样，缩放也可以用“局部”或“世界”这两种方式来表示。局部缩放是相对于其父对象而言的。子对象的 X、Y 和 Z 缩放值会与父对象相应的 X、Y 和 Z 缩放值相乘。这可能会使事情复杂化，因为需要进行一些额外的数学运算才能计算出实际缩放的单位。

为了免去计算的麻烦，可以简单地解除立方体的父子关系，使其采用世界缩放，然后在设置好缩放之后再恢复它们的父子关系。如果仔细观察，会发现在为一个立方体设置父子关系或解除父子关系时，它的缩放值会发生变化，但实际大小却没有任何不同。和位置一样，缩放值只是从世界切换到局部（或相反），但它们代表的实际大小是相同的。

4.4 枢轴点

现在，立方体已经组装完毕，是时候探讨“枢轴点”（pivot point）这个与对象层级相关的重要概念了，它指的是对象进行旋转时所围绕的中心点。

选中 Cube Base 并切换到位置变换工具（快捷键 W）。正如之前提到的那样，变换工具的操作柄位置可以通过按快捷键 Z 在“中心”和“轴心”之间切换。此外，“场景”窗口的标签下方也有一个写着“中心”或“轴心”的按钮。

“中心”意味着辅助工具将位于对象及其所有子对象之间的中心点上。而“轴心”则意味着辅助工具将位于所选对象的枢轴点上。

这个设置不仅影响到辅助工具的位置，还会影响到旋转和缩放等变换工具的工作方式。如果选择一个父对象并使用“中心”模式旋转它，那么父对象及其子对象都会它们共同的中心点进行旋转。如果使用“轴心”，那么子对象都会围绕父对象旋转。

立方体的枢轴点位于它的几何中心。这意味着在旋转立方体时，它会围绕自身的中心点进行旋转。

枢轴点是对象在变换位置上的确切坐标点。如果立方体的位置是（5，5，5），就意味着它的中心位于（5，5，5），而不是位于它的侧面、底部或顶部。了解这一点对于精确地定位对象来说至关重要。

每个网格（三维模型）都有一个枢轴点。在 Unity 提供的基本形状网格中，如立方体、球体、平面和圆柱体，枢轴点通常位于它们的几何中心。但在一些情况下，枢轴点的位置可能

有所不同，比如使用从网上下载的网格、由合作美术人员设计的网格或是自己制作的网格时。如果网格枢轴点的位置异于常规，在使用网格的时候通常很快就会被看出来。

例如，假设美术人员提供了一个手枪网格。我们为这把枪创建了一个游戏对象，并且通过代码把枪放到了玩家角色的手的位置。但是，枪并没有出现在玩家角色的手中，而是跑到了旁边的某个地方。

这个问题是枢轴点设置不当导致的。记住，对象的枢轴点是网格上实际与对象位置相对应的点。如果想把枪精确地放置在玩家的手中，那么枪的枢轴点应该位于枪柄。如此一来，无论怎样调整游戏对象的位置，出现在那里的都会是枪柄，而不是枪管。

幸运的是，我们目前只需要处理基本形状的网格，所以一切都应该是相当直观和可预测的。虽然有些情况下可能还是需要手动调整对象的枢轴点，但这其实并不难做到。

让我们通过游戏来说明这一点。对于之前创建的摩天大楼，我们想让玩家能够购买这些建筑并把它们任意放置在游戏场景中。摩天大楼的枢轴点将位于其底部立方体（它是最大的立方体，并且是所有其他立方体的父对象）。

这个枢轴点需要稍做处理，因为如果使用地板的表面位置来放置建筑，底部立方体的中心点将位于地板上，而它的下半部分会隐没在地板下，不会被摄像机渲染出来。为了将这些建筑整齐地放在地板上，我们必须每次都将它们向上移动半个底部立方体的高度。如果有多种不同的建筑，并且每个建筑的底部立方体高度各不相同，这种做法将使编程工作变得非常烦琐。

为了解决这个问题，枢轴点需要在建筑物的 X 轴和 Z 轴上居中，但在 Y 轴上处于最底部，如此一来，在指定建筑物的位置时，它的底部将会与这个位置完全对齐，并且不会穿透地板。

要实现这一点，有一个简单的解决方案，那就是创建一个空的游戏对象，这是一个除了变换组件之外不带任何组件的游戏对象——仅仅是空间中的一个具有缩放和旋转属性的点。当然，在必要的情况下也可以为它添加其他组件，但在本例中不需要这么做。

可以通过在菜单中选择“游戏对象”|“创建空对象”或使用快捷键 Ctrl+Shift+N 来创建空游戏对象。为了清晰起见，请给它起一个合适的名称，比如“Skyscraper”。它是建筑层级结构的根游戏对象（root GameObject）——这意味着它是一个包含所有相关游戏对象的主父（master parent）对象。在移动它时，整个建筑都会移动。因此，将其命名为它所代表的建筑类型，比如 Skyscraper（摩天大楼）是一个直观且有意义的做法。使用恰当的名称是一个值得培养的好习惯。

接下来，简单将这个空对象放到枢轴点的目标位置并将其设置为 Cube Base 的父对象即可。我将展示一个有助于准确进行定位的技巧。

还记得吗？前面提到过，局部位置是相对于其父对象的缩放值来确定的。如果 Skyscraper 的父对象在 Y 轴上的缩放值是 10，那么 Skyscraper 在 Y 位置的每一点都将计为 10 个单位。这个是与父对象的缩放值相乘得到的结果。此外，也可以使用分数来设置缩放值，比如我们也可以使用分数，例如 0.5 代表 Y 缩放值的一半，0.25 为四分之一，依此类推。

我们可以巧妙地利用这一原理。首先，将空对象 Skyscraper 设置为 Cube Base 的子对象。如此一来，Skyscraper 就会使用相对于 Cube Base 的局部位置进行定位。

根据之前的讨论，现在 Skyscraper 在 Y 轴上的 1 个单位相当于 Cube Base 的高度。我们知道，如果我们把 Skyscraper 放在（0，0，0）的位置，它就会与 Cube Base 位于一处。立方体的枢轴点位于中心，所以 Skyscraper 现在与枢轴点完全重合。接下来要做的是将 Skyscraper“向下”移动半个 Cube Base 的高度，使其与 Cube Base 的底部齐平。因为现在使用的是局部位置，所以只需要简单地将 Skyscraper 的 Y 位置设置为 -0.5 即可。就像前面提到的那样，增加 Y 坐标的值会使对象向上移动，而减少 Y 坐标的值则会使对象向下移动。因此，为了向下移动，必须确保在 Y 坐标前加一个负号“-”。

设置好位置之后，Skyscraper 对象应该恰好位于 Cube Base 的底部，同时在其他轴上居中。接下来要做的是将 Cube Base 设置为 Skyscraper 对象的子对象：先解除两者之间的父子关系，使 Skyscraper 成为没有父对象的独立对象，然后将 Cube Base 拖到 Skyscraper 上。

如此一来，Skyscraper 就成了位于整个建筑底部的根游戏对象。它现在成为新的枢轴点，在旋转 Skyscraper 对象时，所有关联的对象都会围绕它进行旋转，也就是说，整个建筑的旋转枢轴现在位于底部，而不是位于底部立方体的中心。现在，如果把 Skyscraper 挪到地板上的其他地方，整个建筑都会跟着挪过去，这正是我们想要的效果。下一章将说明这样做的另一个好处。

当然，也可以在“场景”窗口中使用变换工具来直接调整枢轴点的位置，从而省去数值调整以及建立和解除父子关系的麻烦。虽然这样做可能无法精确移到目标位置，但在当前应用场景中，这点小偏差不会有太大影响——但有些情况下要求做到百分百的精确，所以了解正规的做法是很有必要的。

4.5 小结

本章要点回顾如下。

1. 不同游戏对象之间可以建立父子关系。当父对象的位置、旋转或缩放发生变化时，子对象也会相应地移动、旋转和缩放，就像它们连接在父对象上一样。

2. 世界位置指的是游戏对象在场景中的位置，这与其他对象无关。局部位置则指的是游戏对象相对于其父对象的位置。两者可以用来表示同一个位置，只是参照的坐标系不同。
3. 枢轴点是对象的世界位置所对应的点。举个例子，如果想设置枪的位置，使枪柄正好位于玩家的手中，那么枪的枢轴点需要设置在枪柄上。如果枢轴点设在其他地方，比如在枪管上，那么玩家手中握着的将是枪管，而不是枪柄。
4. 旋转一个对象时，它的所有子对象都会围绕该对象的枢轴点旋转。这会对对象的旋转方式造成显著的影响。
5. 如果要重新设置游戏对象的枢轴点，就可以创建一个新的空对象并把它放在枢轴点的目标位置，然后让游戏对象成为空对象的子对象。

第 10 章 使用脚本

完全掌握对象的基础知识后，现在是时候学习脚本了。容我友情提醒一句——我们离开始动手编写第一个游戏越来越近了。

脚本是代码文件所对应的组件，负责将代码逻辑嵌入到游戏中。在前面的章节中，我们已经使用一个脚本（MyScript）运行了代码，但还没有深入挖掘它们的潜力。

就其本质而言，脚本也是一种对象。它是一个包含了类定义的代码文件，这个类作为组件，允许我们像添加摄像机、光照、碰撞体、网格渲染器等其他游戏组件一样，将脚本添加到游戏对象上。

在创建自定义类实例时，需要使用构造函数和关键字 new，然而在创建脚本的实例时，则需要通过 Unity 的编辑器来创建，这不难做到，将脚本组件添加到游戏对象上即可。

此外，如果想通过代码来动态地创建脚本的实例，可以使用 Unity 内置的方法来完成。需要注意的是，作为组件，脚本必须添加到某个 GameObject 上才能发挥作用。

如前所述，脚本中可以声明像 Update 和 Start 这样的事件方法，以在游戏中的特定时刻执行代码。实际上，Unity 提供了许多其他的事件方法，有些可能永远不会用到，而有些则会在后面的示例项目中使用。

脚本旨在提供可以添加到游戏对象上的功能模块，它既可以封装复杂的逻辑——如控制玩家的行为，也可以用于简单的任务，比如让一个游戏对象持续旋转。

在实现这些功能时，脚本往往需要依赖于变量。就像在类中定义实例变量一样，我们可以为脚本定义实例变量，使每一个脚本实例都包含这些数据。由于脚本是组件，所以我们可以在 Unity 的检查其中查看这些变量，并单独调整各个实例的变量值。例如，一个控制对象不断旋转的脚本可能包含一个向量变量（包括 X 值、Y 值和 Z 值），用来定义它每秒旋转的幅度。通过在检查器中调整这些变量，我们可以使用相同的脚本来在不同的游戏对象上实现不同的行为，比如有的可能旋转得更快，有的可能围绕不同的轴旋转。

此外，我们还可以在游戏运行时通过检查器来实时修改这些脚本变量的值，而当游戏结束时，这些值会恢复到初始设置。这是一个非常便利的特性，让我们能够随意测试不同的游戏设置（比如调整玩家的跳跃高度或下落速度等物理属性），而不必担心丢失原有的设置。

现在，让我们动手实践，创建一个能够使游戏对象以一定速度持续旋转的简单脚本。这

个脚本不需要太多代码，但它会让我们对游戏开发中的脚本有一个初步的了解。

选中“项目”标签，打开“项目”窗口，然后单击下方的加号按钮并创建一个 C# 脚本，如图 10-1 所示。

将新建的脚本命名为“SimpleRotation”。打开脚本后，你会看到一些熟悉的代码模板。考虑到你对编程越来越熟练，所以这次我们不妨从头到尾过一遍。

图 10-1 单击加号按钮，新建脚本

10.1 using 声明语句和命名空间

在脚本文件的顶部，首先映入我们眼帘的是以下代码：

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

这些被称为“using 声明”的语句，用于告知编译器这个文件将用到其他哪些代码。using 声明以 using 关键字开头，后面跟的是对命名空间的引用。

命名空间（namespace）是一个简单的命名代码块，其中包含其他定义。从某种意义上说，它们就像是计算机上的文件夹。文件夹可以存储文件以及其他文件夹，而命名空间可以存储定义（比如类）以及其他命名空间。

命名空间的目的是将相关的代码块独立于不会使用它的其他部分。

通过命名空间，我们可以方便地引用其内部定义的其他命名空间或类，这个过程就像访问对象的成员一样简单，简单使用句点符号 . 即可。如果在一个命名空间内声明了某些内容，就必须通过命名空间来访问它。这意味着我们需要输入命名空间的名称、句点，然后是想要引用的定义（例如，一个类）。但是，这种做法可能显得有些烦琐，为了简化这一过程，需要引入 using 语句。

在代码中使用 using 语句相当于是在告诉编译器：“把这个命名空间里的所有东西都给我。”如此一来，我们就可以直接引用命名空间内的类，而不需要“进入”命名空间去获取它们。

简而言之，using 语句总是指向一个命名空间，并让我们能够直接引用该命名空间内的所有定义。这种做法不仅提高了编码效率，而且还使得代码更加简洁。

在前面的代码中，using 指向的命名空间是默认基础框架的一部分。

System.Collections 和 System.Collections.Generic 这两个命名空间包含可以用于存储其他对

象的“集合”的类。虽然这次动手实践不会用到它们，但它们的用途非常广泛，这也是为什么脚本会默认包含这些命名空间。

UnityEngine 命名空间包含使用 Unity 编写游戏时会用到的大部分核心定义，因此自然也是默认包含在脚本中的。

UnityEngine 中包括许多定义主要组件的类，其中有一些我们耳熟能详的组件，比如 Transform（变换）、Camera（摄像机）、Light（灯光）、Mesh Renderer（网格渲染器）和 Mesh Filter（网格过滤器）。当然，还有用于表示游戏对象的 GameObject 类。

如果脚本的开头没有 using UnityEngine; 这一行，我们将不得不通过输入 UnityEngine.GameObject 或 UnityEngine.Transform 等来引用这些类。但得益于 using 语句的存在，我们可以直接使用 GameObject 或 Transform 来引用它们。

尽管这次动手实践将不会使用其他两个命名空间中的任何内容，但保留这些 using 声明不会造成任何影响。

10.2 脚本类

继续阅读脚本，可以看到类的定义：

```
public class SimpleRotation : MonoBehaviour
```

这个类会自动采用和脚本文件相同的名称。这一点很关键。如果文件名和类名不匹配，就不能将脚本作为组件附加到游戏对象上！因此，请记住，在重命名脚本的时候，需要同时在“项目”窗口中重命名脚本文件以及脚本中声明的脚本类名称。

这一行的末尾涉及类继承的概念，这将在第 11 章中详细探讨。目前只需要了解一点——MonoBehaviour 部分非常关键，它使得这个类能够作为组件被添加，而不仅仅是一个普通类。尽管 MonoBehaviour 这个名字听起来可能有些古怪，但每次看到它的时候，我们把它理解成“脚本”即可。

类定义内部的代码块非常眼熟，它们和之前在 MyScript 中看到的代码完全相同。代码块中包含一些注释，这些注释以 // 开头，它们会被编译器忽略，只是一些供人阅读的附注。代码块中还定义了 void Start() 方法和 void Update() 方法，每个方法后面都有一个空的代码块。我们已经了解了这些方法的作用：Update 方法每帧被调用一次，而 Start 方法只会在游戏开始时调用一次。

现在，是时候开始为脚本声明变量了。这次会用到一种全新的数据对象——Vector3。Vector3 作为一个对象，能够存储三个浮点数：X、Y 和 Z。实际上， Transform 组件的位置、

旋转和缩放都是通过 Vector3 的实例来表示的，因为它们都包含 X 值、Y 值和 Z 值。这只是我们第一次在代码中与这种数据类型打交道。

我们将使用 Vector3 来表示游戏对象每秒在每个轴上旋转的幅度。如你所知，三个轴（X、Y 和 Z）分别以不同的方式转动游戏对象。

在脚本类（SimpleRotation）的代码块内声明变量是编写脚本时的常规做法。虽然理论上讲，变量可以放置在脚本的任何地方，但最好把它们放在顶部。这样做的原因是，当开发者（包括我们在内）需要查看变量声明时，通常会在类的顶部寻找，而不是像无头苍蝇一样在各个方法之间寻找。

作为一名有经验的开发者，你应该已经知道如何声明变量了。既然如此，请试着声明一个公共的、类型为 Vector3、名为 “rotationPerSecond”的变量：

```
public Vector3 rotationPerSecond;
```

声明了变量后，不妨顺手删去 Start 方法，因为本例不会用到它。不过，我们很快就会用到 Update 方法，因此请不要将其删除。

将变量声明为公共变量非常重要。受保护变量和私有变量在“检查器”窗口中是不可见的，所以如果不将变量设为公共的，就无法单独为每个脚本调整变量值。

现在，保存代码并打开 Unity 编辑器，在编辑器中找到一个想要旋转的游戏对象，比如在前面的章节中创建的 Skyscraper（如果还保留着它们的话），或者也可以在场景中新建一个平面或一个立方体，只能在场景中看到的游戏对象。

接着，将 SimpleRotation 脚本添加到游戏对象上。这可以通过多种方式来完成：一是将“项目”窗口中的脚本文件拖到“层级”窗口、“场景”窗口中的游戏对象上或游戏对象的“检查器”窗口中；二是在选中游戏对象后转到“检查器”窗口，单击所有组件下方的“添加组件”按钮，并在弹出的菜单中导航到 SimpleRotation 脚本，此外，也可以简单地通过在搜索栏中输入脚本名称来查找它。

成功添加脚本后，就可以在“检查器”窗口中看到刚才创建的变量了，后者将以一个可编辑字段的形式出现，如图 10-2 所示。

图 10-2 “检查器”窗口中显示的 SimpleRotation 脚本组件的实例

“检查器”窗口中，为这个变量添加适当的空格和大写字母，它现在变成了 Rotation Per

Second，它旁边显示着三个数字字段，每个字段对应一个轴。

如果没有看到这些字段，请确保将变量声明为公共的，并确保它位于脚本类代码块（class SimpleRotation 后的代码块）内。当然，还要确认一下在添加变量声明后是否保存了代码编辑器中的脚本文件！

10.3 旋转变换

现在，变量已经创建完毕，但我们还需要设法用它来旋转对象。好消息是，这可以通过在 Update 方法中添加一行代码来实现。利用之前声明的 rotationPerSecond 变量，在 Update 方法中添加以下代码：

```
transform.Rotate(rotationPerSecond * Time.deltaTime);
```

游戏对象的 Transform 组件负责处理其位置、旋转和缩放等属性。由此可以推断，旋转游戏对象需要通过 Transform 组件来实现。

SimpleRotation 的类是一个脚本，我们自然就获得了对所有脚本共有的某些成员的访问权限。其中之一便是 transform——请注意，这里的首字母是小写的。Transform 是 Unity 中的一个类，而 transform 则是脚本中的一个成员，它指向脚本附加的游戏对象的 Transform 组件。

类声明中指向 MonoBehaviour 的那一部分赋予我们访问这个成员的权限——它让 SimpleRotation 成为一个脚本，而不仅仅是 C# 语言中一个普通的类。这种机制被称为“继承”，将在下一章中进一步探讨。现在，只需要知道脚本类能够自动访问所有脚本共有的实用成员即可。另一个类似的成员是 gameObject，它指向脚本附加到的游戏对象。

综上所述，我们需要通过 transform 成员来引用 Transform 组件。幸运的是，Unity 贴心地在 Transform 类中提供了一个用于执行旋转操作的 Rotate 实例方法，该方法非常便捷实用。这个方法有几个重载，而我们将使用只接受一个 Vector3 类型参数的版本。它根据给定的 Vector3 的值来旋转 Transform 组件。鉴于旋转涉及 X、Y 和 Z 三个值，它理所当然地期望接受一个 Vector3 类型的参数，而不是单个浮点数。

现在，脚本中的代码应该是下面这样的：

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

public class SimpleRotation : MonoBehaviour
{
```

```
    public Vector3 rotationPerSecond;

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(rotationPerSecond * Time.deltaTime);
    }
}
```

保存更改后返回 Unity 编辑器。注意，由于没有在脚本中为 rotationPerSecond 指定默认值，所以在脚本的所有新实例中，它的值将默认为（0，0，0），这意味着对象根本不会旋转。选中之前添加过脚本的游戏对象，并在“检查器”窗口中将它的 rotationPerSecond 变量设置为（0，0，0）以外的值，无论是正数还是负数值都可以。作为参考，设置为 360 意味着每秒旋转一整圈。

现在如果运行游戏，应该会看到游戏对象旋转起来了。为自己的成就感到自豪吧！这只是一个开始，后面将探索更多有趣的功能，比如实现玩家的移动控制。

就像之前提到的那样，在游戏运行的过程中也可以实时调整 rotationPerSecond 的值。试着更改 rotationPerSecond 的值，游戏对象应该立即做出反应。在停止游戏后，这个值将恢复到游戏开始之前的状态。

10.4 帧与秒

可以看到，在将 rotationPerSecond 作为参数传递给 Rotate 方法时，我们进行了一些数学运算。这里使用了乘法操作符 *，后面跟着 Time.deltaTime，这是一个浮点数。将 Vector3 乘以一个浮点数意味着将向量的每个轴与该浮点数相乘，所以这实际上是在计算（x * Time.deltaTime, y * Time.deltaTime, z * Time.deltaTime）。那么，Time.deltaTime 这个浮点数代表着什么含义呢？

我们知道，Update 方法每帧被调用一次，而游戏的帧率可能并不稳定。当前的场景中没有什么会显著影响计算机性能的东西，所以帧率应该会维持在较高的水平，使得 Update 方法每秒钟被调用数百次。但在玩游戏的过程中，帧率可能会上下波动，并不是恒定不变的。

之所以将变量命名为 rotationPerSecond 而不是 rotationPerFrame，是因为它表示的是每秒的旋转幅度，而不是每帧的旋转幅度。如果直接将 rotationPerSecond 的值作为参数传递，那么游戏对象旋转的速度将远远超出预期。游戏对象将会每帧旋转一次，这意味着它每秒会旋转数几百次（这样做会产生很有趣的效果，可以试试看）。好消息是，将旋转频率从“每帧”改为“每秒”实际上相当地简单。

Time 是 UnityEngine 内置类，提供了一系列实用的静态变量和方法，而 deltaTime 就是其中之一。它是一个持续更新的静态浮点数，记录着自上一帧以来经过的时间（以秒为单位）。这个时间通常非常短。例如，如果游戏以每秒 100 帧运行，那么 deltaTime 就是 0.01 秒。我们将这个值与 rotationPerSecond 相乘，从而将旋转速度从“每帧”调整成“每秒”。

为了更好地理解，可以把这个例子简化一下，假设游戏的帧率是每秒两帧，那么每帧的时间就是 0.5 秒，对吧？所以每次调用 Update 方法时，Time.deltaTime 的值将是 0.5。将任何数值乘以 0.5，得到的结果都会是原数值的一半。这意味着每次调用时得到的结果都是 rotationPerSecond 的一半。如果旋转速度被设置为每秒 50 度，那么每帧的旋转幅度就是 25，两帧的旋转幅度就会是 50，完全符合我们的预期。

这种概念同样可以在移动游戏对象的时候用于确保它们以每秒为单位进行移动，而不是每帧。实现这一点非常简单，只需将移动量乘以 Time.deltaTime 即可。

为了更深刻地理解为什么选择按秒移动而不是按帧移动，让我们进一步探讨这种方法底层的逻辑。

这涉及“帧率独立”（framerate independence）的概念。在帧率独立的游戏中，即使帧率较低导致游戏运行时出现卡顿而影响玩家的体验，也不会让游戏看起来像是在进行慢动作一样。这是因为游戏使用了静态浮点数 deltaTime，根据实际流逝的时间来计算对象的移动。

另一方面，在“帧率依赖”（framerate dependent）的游戏中，所有的数值都是基于每帧来设定的，而不是每秒。开发者会设定一个“目标帧率”，例如每秒 30 帧或 60 帧，并在此基础上限制游戏的帧率，不允许它超过这个目标值。然后，开发者将会以“每帧”为单位来设定游戏的行为和事件，并期望游戏始终稳定在目标帧率上。如果实际帧率低于这个目标值，游戏就会呈现出慢动作的效果，因为所有操作都是基于每帧来执行的，而帧与帧之间的实际时间间隔并不被考虑在内。

这种做法尤其常见于针对游戏主机开发的游戏，因为开发者清楚地了解目标主机的硬件配置以及该硬件的处理能力，可以围绕着这些配置来开发游戏。

另一方面，PC 游戏需要适应各种不同的硬件配置，如果游戏过分依赖于特定的帧率，不仅会让硬件性能不足、达不到目标帧率的玩家体验很差，也会让那些硬件配置较高的玩家感到不满，因为他们的硬件本可以支持更为流畅的游戏体验，却受帧率限制在每秒 30 或 60 帧。

Unity 平台具有跨 PC 和游戏主机开发游戏的能力，以“每秒”为单位的话，我们能够为不同的平台采用统一的度量系统。

10.5 属性

属性（attribute）可以看作是附加到代码定义上的类实例。这听起来可能有些抽象，但它是将元数据引入代码中的有效手段。程序员可以声明属性来指定关于定义的某些特定信息，然后将属性的实例附加到声明的定义上（比如类、变量、方法等）。其他代码随后可以读取这些属性并执行相应的操作。

UnityEngine 命名空间提供了很多实用的属性。例如，HideInInspector 属性就是一个典型的例子。通过将它附加到变量，我们可以在 Unity 编辑器的“检查器”窗口中隐藏该变量。利用这个属性，我们可以在隐藏变量的同时保持它为公共变量，以便其他脚本能够通过引用访问它。在某些情况下，我们可能希望脚本中有 public 类型的变量，但又不想让它们能够在“检查器”窗口中被修改（或是不想让它们占用“检查器”窗口的空间）。

若想将属性应用到定义上，需要在定义前添加一对方括号 []，并在其中键入属性名称。

现在，让我们尝试通过添加属性来隐藏 rotationPerSecond 成员：

```
[HideInInspector] public Vector3 rotationPerSecond;
```

定义本身保持不变，唯一的区别是变量声明前多了一对方括号 [] 和一个属性名。保存代码并返回 Unity 编辑器，选中带有 SimpleRotation 脚本的游戏对象，可以看到 rotationPerSecond 已经从“检查器”窗口中消失了。

单个定义可以有多个属性，每个属性都需要有一对方括号 []。为了提高代码的可读性，可以将属性和定义分为两行：

```
[HideInInspector]
public Vector3 rotationPerSecond;
```

如果有多个属性，甚至可以为每个属性单独分配一行。

不过，在实际应用中，我们并不想对 rotationPerSecond 成员应用 HideInInspector 属性，因为我们需要在“检查器”窗口中编辑它。所以现在可以把这个属性删掉了。

现在再来看看另一个实用的属性 Header。通过将 Header 属性附加到变量上，可以在“检查器”窗口中的变量名上方添加一个粗体的标题。这个属性对于在“检查器”窗口中视觉上区分不同的变量组非常有用，能够让“检查器”窗口界面更加清晰、有序。举个例子，在一个定义玩家角色行为的脚本中，可能需要为控制移动、跳跃和攻击的变量组设置不同的标题。

Header 属性通过接受一个字符串参数来设置标题文本。声明带参数的属性的方式类似于调用方法或构造函数。在键入属性名称后，需要添加一对圆括号 () 并在其中添加字符串：

```
[Header("My Variables")]
public Vector3 rotationPerSecond;
```

保存并查看“检查器”窗口，我们会看到 rotationPerSecond 变量上方出现一个粗体显示的标题“My Variables”，如图 10-3 所示。

图 10-3 检查器中显示的 SimpleRotation 脚本实例 ①

如果试图将 Header 属性应用到在“检查器”窗口中不可见的变量上，比如带有 [HideInInspector] 属性的变量或私有变量，那么标题将不会出现在“检查器”窗口中。

这两个属性虽然简单，却在维护“检查器”窗口的整洁美观上发挥着重要的作用。

10.6 小结

本章讲解了如何将代码文件作为脚本组件附加到游戏对象上，学习了如何在“检查器”窗口中公开展示变量，以自定义每个实例的设置。要点回顾如下。

1. 在脚本类中声明的公共变量可以在“检查器”窗口中进行查看和编辑。如果想要测试不同的设置，可以在游戏运行过程中调整这些值，但这些调整在游戏停止后将不会被保存。
2. 脚本文件中声明的类必须与脚本文件同名。如果名称不一致，就无法将脚本作为组件附加到游戏对象上。
3. 脚本默认可以访问 transform 成员变量，它指向脚本附加到的游戏对象的 Transform 组件。
4. 属性（attribute）通过方括号 [] 来声明。Unity 提供了一些内置特性，通过将它们应用到变量上，可以实现多种实用的功能，比如在“检查器”窗口中隐藏变量或为变量添加粗体标题，使“检查器”窗口的界面更加整洁有序。

① 译注：当前版本的中文版 Unity 疑似有 bug，导致一些本来不该加粗的文本也会加粗显示。例如，这里的 Rotation Per Second 就不应该是粗体。

第 11 章 继承

继承（inheritance）是面向对象编程的核心概念之一。它是一种机制，允许新的数据类型（比如类）从现有的数据类型中继承字段（比如变量和方法）。

假设有两个类，它们拥有的字段几乎完全相同，只不过其中一个类多了一个字段或方法。分别编写这两个类的代码显然是一项重复且乏味的工作，因为它们的大部分功能是相同的。而且如果分别编写的话，在需要修改它们共有的某个属性时，就必须分别对两个类进行修改，并确保它们保持一致。如果这样的类不止两个的话，问题还会变得更加棘手。

继承提供了一种解决方案。我们可以创建一个基类，其中包含两个类共有的功能。然后，其他类可以从这个基类继承，自动获得其变量和方法。如此一来，就可以集中管理所有功能，确保它们在所有子类中保持一致。

11.1 继承机制的应用：RPG 游戏中的物品系统

在角色扮演游戏（RPG）中，继承的概念起着至关重要的作用，尤其是在构建物品系统的时候。游戏中的每个物品通常都包含下面几个字段：

- 一个整数值，表示物品在商店中的售价；
- 一个布尔值，表示物品是否“可出售”；
- 一个字符串，用于表示物品的名称。

此外，还可以根据需要添加其他字段，比如物品描述和物品重量。

除了这些通用属性，游戏中还存在一些具有特定属性的物品。假设我们想创建一些装备于特定槽位的盔甲和一些装备于其他槽位的武器，比如只能放入“鞋子”槽位的“雷霆战靴”。

这可以通过继承来实现。首先需要创建一个基类，其中包含所有物品共有的成员。由于这个基类是所有物品的组成部分，所以简单地将其命名为“Item”。在此基础上，我们可以创建多个从 Item 类继承的子类，但它们将拥有更具体的用途和特性。

假设每件装备——无论是盔甲还是武器——都有一个随着使用而逐渐消耗的耐久度。在装备盔甲并承受伤害时，盔甲的耐久度就会降低；在装备武器并攻击敌人时，武器的耐久度就会降低。

为了实现这个游戏机制，我们可以创建一个名为“Equipment”的子类。由于 Equipment 子类继承自 Item 类，它拥有 Item 类中的所有字段，比如物品名称和物品描述等。除此之外，

我们还为其增加了两个整数字段：一个用于表示当前耐久度，另一个用于表示最大耐久度。每当装备承受了一定次数的攻击后，其当前耐久度就会下降一点。当前耐久度降为 0 时，装备就会损坏。玩家可以通过修理装备来使当前耐久度恢复到最大耐久度，以再次使用装备。

接着，我们需要创建一个名为“Weapon”的类，它继承自 Equipment，并额外添加了一些字段，比如最小和最大伤害值、攻击速度以及武器类型（一个枚举类型，包括斧、剑、锤子、刀、锋利的石片等选项）。

还需要创建一个名为“Armor”的类，它同样继承自 Equipment 子类，其中包含一个用于表示防御力的字段。此外，我们还需要一个枚举来表示盔甲的类型（如靴子、腰带、手套、胸甲或头盔），这将用来决定盔甲对应的装备槽。

如果还想为消耗品（如药水或食物）创建一个类，可以再创建一个直接从 Item 类继承的 Consumable 类。最终，这些类的层级结构如下所示，缩进表示它们之间的继承关系：

```
Item
    Equipment
        Armor
        Weapon
    Consumable
```

我们需要明确一些术语的定义。子类（subclass）比超类（superclass）更加具体。在这个例子中，Item 是超类，而像 Consumable 或 Equipment 这样更具体的版本是它的子类。Equipment 还有两个更具体的版本，分别是 Weapon 和 Armor。这构建了一个类的层次结构，其中越细分的子类就越具体。

11.2 声明类

我们可以使用之前声明的 Item 类作为基类。它目前还嵌套在我们的“老朋友”MyScript 类中，这意味着其他脚本无法访问 Item 类。如果一个嵌套类仅限于其所属的类内部使用，这么做就是合理的，但 Item 这样比较通用的类应该在独立的脚本文件中声明，以便在整个项目范围内访问。

由于我们目前只将 Item 类用作示例，为了简单起见，可以让它继续嵌套在 MyScript 类中。不过，我们需要删掉不会再用的一些构造函数和方法：

```
public class MyScript : MonoBehaviour
{
    class Item
    {
```

```
        public string name = "Unnamed Item";
        public int worth = 1;
        public bool canBeSold = true;
    }
}
```

以上代码定义了 Item 基类，这是物品的最基本类型，其中声明了所有物品共有的变量，并为它们设置了默认值。这些默认值其实不是必须要有的，因为我们稍后将为这些变量添加构造函数。不过，保留这些默认值也无伤大雅。

现在，让我们来定义 Equipment 类——它将被用作 Weapon 和 Armor 的共同基类。把 Equipment 类的定义放置在与 Item 类相同的代码块中，紧挨着 Item 类的定义。这意味着 Equipment 和 Item 是同一级的，都嵌套在同一个代码块中：

```
public class MyScript : MonoBehaviour
{
    class Item
    {
        public string name = "Unnamed Item";
        public int worth = 1;
        public bool canBeSold = true;
    }
    class Equipment : Item
    {
        public int currentDurability = 100;
        public int maxDurability = 100;
    }
}
```

继承的语法与常规类声明非常相似，唯一的区别在于 class Equipment 之后紧跟着一个冒号 :，这表明 Equipment 类将继承自另一个类。随后，我们指定要继承的基类名称，即 Item。这一小段代码便是能使 Equipment 继承 Item 所需要的全部代码。

接下来要声明 Armor 类。此外，这里还要运用到之前学习的知识，声明一个 ArmorType 枚举，用于区分不同类型的盔甲（如手套、头盔等）。将这两个定义放在 MyScript 代码块中，紧接在 Equipment 类之后：

```
enum ArmorType
{
    Helmet,
    Chest,
    Gloves,
```

```
    Belt,
    Boots
}
class Armor : Equipment
{
    public ArmorType type = ArmorType.Helmet;
    public int defense = 1;
}
```

枚举的声明非常简单直接，这在前面已经讨论过。

Armor 类继承自 Equipment，而 Equipment 又继承自 Item。这创建了一个继承链。通过这种方式，Armor 类最终继承了在 Item 类中声明的所有变量，以及在 Equipment 类中声明的变量。除此之外，Armor 类还包含一些额外的成员：一个用于存储 ArmorType 枚举实例的变量，其默认值设为 Helmet，代表头盔；还有一个用于表示盔甲防御力的 int 变量，其默认值为 1。

接下来声明 Weapon 类。将以下代码添加到 MyScript 代码块中，紧接在 Armor 类之后：

```
enum WeaponType
{
    Sword,
    Axe,
    Hammer
}
class Weapon : Equipment
{
    public WeaponType type = WeaponType.Sword;
    public int minDamage = 1;
    public int maxDamage = 2;
    public float attackTime = .6f;
}
```

Weapon 类的声明过程与 Armor 类相似。这段代码首先定义一个 WeaponType 枚举，它包含一系列基本武器类型。然后，这段代码声明 Weapon 类，它继承自 Equipment 并添加了一些额外的成员，包括武器类型、每次攻击的最小和最大伤害值以及执行单次攻击所需要的时间。

11.3 构造函数链

为了便利地创建实例并确保为所有成员赋正确的值，我们需要为这些类添加构造函数。可以想象，为每个类都单独声明一个构造函数是一项相当烦琐的工作，尤其是考虑到继承体

系中子类通常需要设置在基类中定义的成员变量的值时。以 Item 为例，它的每一个子类都需要为 Item 类中定义的 worth 和 name 等成员声明参数并赋值。

幸运的是，有一个方法可以简化这个过程——构造函数链（constructor chaining）[①]。通过利用这个方法，子类的构造函数可以调用基类的构造函数，而基类的构造函数又可以调用它的基类的构造函数，依此类推。这个过程实质上是在构造函数的声明中直接调用基类的构造函数，并且即时传递所需的参数。

可以这样理解构造函数链的概念：尽管每个类的构造函数都必须声明它的所有基类的参数，但这并不意味着我们必须手动设置各个参数，因为这些任务可以交给基类中的构造函数来完成。也就是说，每个构造函数仍然要声明包括基类成员在内的参数，但那些并非该子类独有的参数会被“传递”给构造函数链中的上一层，由基类的构造函数负责赋值。例如，Equipment 类会把在 Item 中定义的成员的参数“传递”上去，由 Item 的构造函数来处理这些参数。

下面来看具体如何操作。首先要编写 Item 类的构造函数。在 Item 类的代码块中添加以下代码：

```
public Item(string name, int worth, bool canBeSold)
{
    this.name = name;
    this.worth = worth;
    this.canBeSold = canBeSold;
}
```

这是之前学习过的标准构造函数定义。由于参数名称与 Item 中声明的成员名称完全相同，在为这些成员变量赋值时，我们采用了 this 关键字来区分参数和变量。除此之外，我们所做的就是将参数值赋给 Item 类的变量。

接下来定义 Equipment 类的构造函数。这个构造函数声明位于 Equipment 类中，紧跟在变量声明之后：

```
public Equipment(string name, int worth, bool canBeSold, int maxDurability)
:base(name, worth, canBeSold)
{
    // 应用最大耐久度：
    this.maxDurability = maxDurability;

    // 使当前耐久度等于最大耐久度：
    currentDurability = maxDurability;
}
```

① 译注：指在一个类的构造函数中调用另一个构造函数，通常基于这样的考虑：代码复用以及使构造函数保持清晰。在 C# 语言中，这是通过 this 关键字和 base 关键字来实现的。

现在，情况开始复杂起来了。可以看到，参数后面紧跟着一个 :base。这就是构造函数链。base 关键字指的是正在继承的基类，在本例中，它指的是 Item 类。base 后面是一对括号，这相当于是在调用上一层的构造函数，也就是基类 Item 的构造函数。

在调用上一层的构造函数时，需要传递当前构造函数声明的初始参数，这些参数与上层构造函数中声明的参数相同。这些参数并非 Equipment 类专有，而是从 Item 继承来的，因此可以将它们传递给 Item 的构造函数来处理。毕竟，我们之前已经定义了 Item 构造函数来处理这些值，而作为程序员，我们总是力求避免重复劳动。

此外，Equipment 类的构造函数中还声明了一个新的参数 maxDurability。这个参数是 Equipment 独有的，所以不需要将它传递给 Item 类的构造函数。Item 并不负责处理耐久度，我们需要在 Equipment 类的构造函数中使用这个参数来设置耐久度。

可以看到，这里只为 maxDurability 定义了参数，而没有为 currentDurability 定义参数。这段代码首先将 maxDurability 参数的值赋给相应的类成员（this.maxDurability = maxDurability），然后直接将 currentDurability 设置为 maxDurability 的值，如此一来，武器将默认具有最大耐久度。请记住，由于 currentDurability 是当前类的一个实例成员，我们可以直接通过其名称来访问它。而且，由于构造函数中没有名为 currentDurability 的参数，所以在引用这个变量时不必添加 this 关键字。

接下来，我们将在此基础上进一步扩展，为 Armor 类定义一个链式构造函数。和之前一样，将构造函数添加到 Armor 类中的变量之后：

```
public Armor(string name, int worth, bool canBeSold, int maxDurability,
ArmorType type, int defense)
:base(name, worth, canBeSold, maxDurability)
{
    this.type = type;
    this.defense = defense;
}
```

这个构造函数的逻辑与之前的构造函数相同。先按照与基类相同的顺序声明初始参数（这部分可以直接复制过来），再把它们传递给基类的构造函数。这次还涉及 Equipment 类，所以 maxDurability 参数也被添加到对基类构造函数的调用中。接着，这段代码为 Armor 类独有的两个成员（type 和 defense）添加了额外的参数，并在函数体中应用它们。

为了更直观地展示构造函数的参数是如何沿着继承链向上传递的，下面列出每个构造函数，其中每个构造函数中的新增参数以粗体表示：

```
public Item(string name, int worth, bool canBeSold)
```

- public Equipment(string name, int worth, canBeSold, **int maxDurability**)
- - public Armor(string name, int worth, canBeSold, int maxDurability, **ArmorType type, int defense**)
- - public Weapon(string name, int worth, canBeSold, int maxDurability, **WeaponType type, int minDamage, int maxDamage, float attackTime**)

那些“沿着链向上传递”的参数以普通文本的形式显示，而由当前构造函数直接处理的参数则以粗体显示。代码的缩进显示了类与类之间的继承关系。

现在，在 Armor 类和 Weapon 类中，currentDurability 参数的赋值将会自动实现，因为这些参数被传递给 Equipment 构造函数，让它代为处理。如果不使用构造函数链，我们将不得不到处复制粘贴各种代码，这不仅会让代码显得杂乱无章，而且在需要进行修改时还很容易出错。

接下来声明 Weapon 构造函数。到这里，你应该已经能够独立完成这项任务了：

```
public Weapon(string name, int worth, bool canBeSold, int maxDurability, WeaponType type,
int minDamage, int maxDamage, float attackTime)
:base(name, worth, canBeSold, maxDurability)
{
    this.type = type;
    this.minDamage = minDamage;
    this.maxDamage = maxDamage;
    this.attackTime = attackTime;
}
```

这个构造函数首先为 Item 类的成员声明参数，再为 Equipment 类的成员（maxDurability）声明参数，然后再声明它自己独有的成员。然后，它把除了独有参数以外的所有参数都传递过去，就像之前在 Armor 类中所做的那样。

好了，构造函数链至此就告一段落，我们已经为 Item、Equipment、Armor 和 Weapon 这几个类完成了所有构造函数和数据的设置。

11.4 子类型和类型转换

在处理涉及继承关系的类时，我们经常需要使用基类来存储对象，然后等到实际使用时再确定对象具体属于哪个子类，并据此做出相应的处理。

例如，我们可以将玩家装备的盔甲作为 Armor 类型的引用，并将武器作为 Weapon 类型的引用进行存储，因为玩家在盔甲槽和武器槽中只能装备这两种类型的物品。但是在管理玩家的物品栏时，情况就不同了。因为玩家可以拾取各种类型的物品，所以最好使用 Item 这一基类来存储玩家背包中的各个物品。

继承机制在这里提供了很大的帮助。如果创建一个类型为 Item 的变量，它将可以存储 Item 的任何子类型的引用。因此，在玩家的物品栏中，所有物品都可以作为 Item 类型存储，而这些物品实际上可以是 Item 的任何子类型，例如 Weapon 或 Armor，而不会导致编译器报错。不过要注意，在将这些物品视为 Item 时，我们只能访问 Item 类中定义的成员，比如 name 和 worth 等。试图访问子类特有的成员将会引发错误。

这时候就轮到类型转换（typecast）“出马”了。如果想访问 Weapon 或 Armor 等子类中定义的成员，就必须先将 Item 类型的引用转换为相应的子类型。

我们通过类型转换来告诉编译器期望的类型是什么，如此一来，编译器就可以将泛型类型（如 Item）的引用看作是对更具体的类型（如 Weapon）的引用。

请看以下代码。这段代码创建一个 Weapon，并为它的参数提供了一些泛型值，把它定义成了一个“Rusty Axe”（生锈的斧头）。最关键的是，它没有被存储为 Weapon 类型的变量，而是被存储在一个类型为 Item 的局部变量中：

```
void Start()
{
    Item item = new Weapon("Rusty Axe", 4, true, 40, WeaponType.Axe, 4, 9, .6f);
}
```

■**补充说明：**虽然之前简要介绍过 f 的概念，但请容我再次重申一下它的作用。之所以在数值参数（例如 .6f）的末尾添加字母 f，是为了明确告知编译器该数值应被理解为 float 类型。本章的结尾将会解释为什么这一步是不可或缺的。

这个变量声明应用了我们刚刚学到的知识。之所以能将 Weapon 存储在 Item 变量中，是因为 Weapon 是 Item 的子类。虽然 Weapon 更加具体，但仍然可以归类为 Item，因为它具 Item 中的所有成员，即使它有一些额外的成员也没关系。但反过来就不行了，我们不能将 Item 或 Equipment 的实例存储在 Weapon 或 Armor 类型的变量中。

原因在于，在引用 Weapon 或 Armor 的实例时，我们期望它们具有这些子类的所有成员。如果一个变量声明为 Weapon 或 Armor 类型，但实际上存储的却是 Item，那么在尝试调用 Weapon 或 Armor 独有的成员时，就不可避免地会引发错误。这就是为什么编译器从一开始就不允许我们这样做。使用强类型语言的一个好处就是，它能够强制我们遵守这些规则，以保持代码的清晰和整洁，以免引入错误。

我们之前已经将 Weapon 实例作为 Item 的类型进行了存储，让我们尝试将其转回 Weapon 引用，看看会发生什么。添加一行代码，声明一个类型为 Weapon 的局部变量，并将 Item 对象的值赋给这个变量：

```
void Start()
{
    Item item = new Weapon("Rusty Axe", 4, true, 40, WeaponType.Axe, 4, 9, .6f);
    Weapon weapon = item;
}
```

我们知道 item 变量中存储了一个 Weapon，所以这段代码理论上应该可以工作——但如果保存并回到 Unity，我们将会看到下面这样的错误提示：

Cannot implicitly convert type ‘MyScript.Item’ to ‘MyScript.Weapon’. An explicit conversion exists (are you missing a cast?)

意思是“无法隐式地将类型‘MyScript.Item’转换为‘MyScript.Weapon’。存在显式转换方式（你是否忘记进行类型转换了？）”这样的错误提示指出了我们想要进行的操作：隐式类型转换。

类型转换可以隐式（implicit）或显式（explicit）地进行。两者的区别在于，程序员是否明确下达了转换的指示。以上代码并没有告诉编译器“要把这个类型转换成那个类型”，因此被视为隐式转换，这种转换会在没有明确指示的情况下自动进行。

错误提示充当了一种防御机制，防止我们无意间执行本不该执行的类型转换。虽然我们知道这个 Item 中存储了一个 Weapon，但编译器并不知道，哪怕这就是在前一行代码中声明的。

因此，我们必须进行显式类型转换。这是一种在程序运行时（即在游戏过程中）即时进行的转换，需要通过特殊的语法来实现。这就是称之为“显式”转换的原因——我们知道代码会执行这种转换。类型转换是在我们的明确要求下进行的，而不是自动发生的。

实现这种类型转换有两种方法，它们的功能本质上是相同的，但在类型不匹配的情况下，它们的行为略有不同。

第一种方法是在 item 引用之前添加一对圆括号，在其中写上名称，表示想转换为什么目标类型：

```
Weapon weapon = (Weapon)item;
```

在采用这种方法的情况下，如果在运行时发现类型不匹配（例如 item 实际上并没有存储 Weapon 实例或者 Weapon 的子类实例），程序就会抛出错误。如果类型匹配，这个方法则可以成功地将 item 类型转换为 Weapon 类型。

第二种方法是使用 as 操作符：

```
Weapon weapon = item as Weapon;
```

这种方法在类型不匹配时不是抛出异常，而是返回 null，这相当于一个不指向任何内容的

引用。如果匹配，它将成功返回 Weapon 类型。

当然，as 操作符不抛出异常并不代表程序能够正常运作。如果后续代码使用了这个 weapon 变量（比如尝试从中访问数据或调用方法），而它的值为 null，那么还是会产生错误——只不过是另一种类型的错误。

11.5 类型检查

在目前处理过的所有测试用例中，我们都对数据的类型了如指掌，所以不必担心出错。但在实际应用中，往往需要先确认引用的实际类型是否符合预期，然后才能与之交互。

假设有一个指向 Item 的引用。为了避免引入其他问题，暂且假设这个 Item 引用是由负责管理玩家物品栏的代码提供的。为了确定它应该具备哪些功能，我们需要先识别出它的具体子类型。

这可以通过几种方法来做到。假设我们有一个名为“item”的变量或参数，它的类型是 Item。那么，应该如何判断它是否为 Weapon 或 Armor 呢？

一种方法是使用操作符 is。它的左侧接受一个值，右侧直接引用一个类型。如果左侧的值正好是右侧指定的类型，或者是该类型的一个子类（即更具体的类型），那么它就会返回 true，否则会返回 false：

```
if (item is Weapon)
    Debug.Log("Item is a weapon.");

else if (item is Armor)
    Debug.Log("Item is an armor piece.");

else if (item is Equipment)
    Debug.Log("Item is some kind of equipment, but not Armor or Weapon.");
```

这段代码会根据不同情况记录不同的日志信息：如果 item 变量被识别为 Weapon 类型或 Armor 类型，系统将记录一条相应的消息；如果 item 变量既不属于 Weapon 也不属于 Armor，但被归类为 Equipment，我们将记录一条通用的消息；如果 item 变量不属于 Equipment 类型，那么系统将不会记录任何日志。

另一种方法是使用上一节提到的操作符 as 将 item 变量赋值给某个子类的新变量，然后通过 if 语句来测试结果是否为 null。如果为 null，就意味着 item 变量不属于那个子类；如果不为 null，则意味着 item 变量属于它或它的子类：

```
Weapon weapon = item as Weapon;
if (weapon == null)
```

```
{
    // 转换失败，'item' 不是 Weapon 的实例
}
else
{
    // 转换成功，可以继续使用 weapon 这个引用
}
```

有时，我们可能需要检查一个对象的实例是否与某个类型完全匹配。实现这一目标的代码相对没有那么直观。我们需要在对象上调用 GetType 实例方法来以获取它的具体类型，然后使用操作符 == 将这个类型与目标类型进行比较，看它们是否完全相等。不过，在进行这种比较时，不能直接使用类型的名称，而是必须使用 typeof(…) 将类型名称括起来，如以下代码所示：

```
if (item.GetType() == typeof(Equipment))
    Debug.Log("Item type is exactly Equipment.");
```

在声明 item 变量时，我们为它赋了一个 Weapon 实例，所以在检查 item 变量的类型是否为 Equipment 时，得到的结果将是 false，日志消息不会被记录。原因在于，Weapon 虽然是 Equipment 的子类，但它并不等同于 Equipment。

通过运用以上三种方法，我们几乎可以处理所有需要进行类型检查的场景。

11.6 虚方法

最后一个有关继承的关键概念是虚方法（virtual method）的概念。第二个示例游戏项目要用到虚方法，到时候我再详细介绍它们的语法和用途。但考虑到这里正在讨论继承的话题，我们不妨先来了解一下虚方法的定义和用途。

将方法声明为虚方法意味着它们可以被子类重写（override），如此一来，子类就可以添加自己的功能，甚至完全覆盖以便下一级类型可以增加自己的功能或甚至完全重载（overwrite）基类的功能。

让我们通过一个具体的例子来说明。假设我们创建了一些类来代表不同的物品类型，比如 Consumable:Item 类和 Food:Consumable 类。

Consumable 类用于表示像药水这样的可消耗物品，使用这些物品可以即时产生某种效果。它内部声明了一个名为“Use”的虚方法，该方法接受一个 target 参数，指向游戏中一个特定的实体，比如玩家或 NPC。当玩家或 NPC 使用药水时，他们会调用 Use 方法并提供自己作为 target 传入。虚方法决定着对 target 对象产生什么效果。

我们可以在此基础上创建一些子类，比如 HealthPotion:Consumable 和 ManaPotion:Consumable，并在这些类中重写虚方法 Use。虚方法的各个实现可以利用传入的 target 参数执行不同的操作。例如，生命药水（Health Potion）用于恢复 target 的血量，而法力药水（Mana Potion）用于恢复其法力值。每种药水都以不同的方式定义 Use 方法。

此外，还可以创建一个 Food:Consumable 子类。重写其中的 Use 方法，通过食物的 tastiness（美味度）来降低 target 的饥饿度，这是 Food 类独有的成员变量。

然后，只需要引用 Consumable，我们就可以在任何 target 对象上调用 Use 方法，而不用关心这个 Consumable 具体属于什么子类型。系统会自动选择并执行合适的方法重写——如果 consumable 是食物，则减少目标的饥饿度；如果是药水，则根据药水的类型来恢复目标的生命值或法力值。

这个功能非常强大，它允许子类针对特定事件做出独特的响应，或者以自己的方式实现一些共有的特性。

11.7 数字值类型

前面几个小节提到，本章的末尾要解释为什么需要在一些数值的末尾添加 f（如 0.6f），而现在正是做出解释的好时机。在 C# 语言中，f 是一个“后缀”（suffix）。

后缀可以添加到数字值的末尾，用于指定数值应该以哪种数据类型存储。前文已经介绍过 int 和 float 这两种类型，但除了这两者，还有其他很多类型，它们在所能存储的数值范围上各有限制。此外，还有一些类型能够存储更广泛的数值范围，但会占用更多的内存空间；而另一些类型占用的内存相对较少，但能够存储的数值范围也比较小。

举例来说，sbyte 类型的范围就比 int 类型小得多。int 类型能够存储的数值范围极广，最高可达 20 多亿，最低可达负 20 多亿，而 sbyte 类型却只能存储 -128 到 127 之间的值。sbyte 类型在计算机上占用的空间更少，但这也意味着在许多应用场景中，sbyte 类型无法存储足够大的数值。

许多数据类型还有无符号（unsigned）版本，它们不能存储负数（它们的最小值为 0），但相应地，它们能存储的正数值是标准版本的两倍。例如，sbyte 中的 s 代表“有符号”（signed），而无符号版本则称为 byte，同样，int 的无符号版本被称为 uint，其数值可以超过 40 亿，但不能小于 0。

有些数字类型可以通过在数字后面添加一个后缀来表示，比如加 f 表示 float。有些类型没有这样的后缀，需要通过显式转换来实现。例如，(byte)120 可以将数字 120 从 int 类型转换为 byte 类型。

在默认情况下，没有小数部分的数值会被识别为 int 类型，除非该数值超出 int 类型的存储

范围。如果数值超出 int 类型的范围，编译器会自动选择一个更大的数据类型来存储该数值。

带有小数部分的值将存储为 double（双精度浮点数或称双浮数），它是 float 类型的两倍大，通常在小数部分更为精确。如果参数期望得到一个 float 值，但我们直接传入了 0.6 这样的数字，那么这实际上是传入了一个 double 值。为了修正这个错误，我们需要在数字末尾加上 f 后缀，使它成为一个 float 值。

在大多数情况下，int 类型和 float 类型已经完全够用了，并且 Unity 引擎中的几乎所有内置方法都期望数值类型是这两种类型之一。Unity 的内置方法使用 float 类型来表示带小数点的值，所以你以后会经常看到数字后面的 f。

这个例子再次说明了类型转换的用途，并展示了隐式转换和显式转换的应用。那么，在需要 float 类型的时候，为什么 double 类型不能隐式（自动）转换成 float 呢？为什么我们必须亲手在数字后面加上这个 f 呢？

答案很简单：因为 double 类型存储的信息比 float 类型多。从 double 类型转换为 float 类型会丢失一些小数位的信息，而这些信息可能对程序很重要。因此，隐式转换是不被允许的——我们必须明确提出转换的要求。同样地，从 float 类型转换为 int 类型时，小数部分会被舍弃，所以这种转换也需要显式地进行。

然而，如果将一个 byte 类型的值——如前所述，这是一个从 0 到 255 的数字——赋值给 int 类型的变量，那么转换将能够隐式地进行。它将会自动发生，不需要我们干涉。这是因为 int 类型可以无损地存储 byte 的所有信息，因而这种转换不会有丢失数据的风险。

就像前面提到的那样，在 Unity 中，我们几乎只会处理 int 和 float 这两种数据类型。但如果需要使用其他数据类型，可以参考表 11-1，其中列出了整型数据类型及其后缀（请注意，有些类型没有后缀）。

表 11-1 整数数据类型及其后缀

数据类型	值范围	后缀
sbyte	–128 到 127	--
byte	0 到 255	--
short	–32 768 到 32 767	--
ushort	0 到 65 535	--
int	–2 147 483 648 到 2 147 483 647	--
uint	0 到 4 294 967 295	U
long	–9 223 372 036 854 775 808 到 9 223 372 036 854 775 807	L
ulong	0 到 18 446 744 073 709 551 615	UL

此外，还有三种不同的数据类型可以用来表示带有小数的数值，也就是浮点数：

- float（浮点数，也称为单精度浮点数或单浮数）通过 f 或 F 后缀来表示；
- double（双精度浮点数或双浮数）在未指定任何后缀时作为默认选择，如果没有后缀的话，可以使用 d 或 D 后缀来表示；
- decimal 通过 m 或 M 后缀来表示。

double 类型之所以被称为 double，是因为它占用的内存是 float 类型的两倍，而 decimal 占用的内存是 double 类型的两倍，也就是 float 类型的四倍。

浮点数值并不总是完全准确的，特别是在存储非常大的数值时。有时，在读取之前设置的某个值时，我们可能会发现它的小数部分有微小的偏差。比 float 类型更大的数据类型能够存储更大的数值，并在处理过程中保持更高的精度。不过，在大多数 Unity 开发场景中，float 类型已经足以满足需求了。只有在处理很大的数值时，才可能需要考虑使用更高精度的数据类型。

11.8 小结

现在，你已经对继承的概念有了初步的了解。这是一个很广泛的主题，我们尚未探索所有相关知识点，但对于开发简单的游戏，这些知识已经足够了。随着编程技能的不断提升，你将自然而然地掌握更多的专业知识。

接下来，我们将运用这些概念并深化对它们的理解，学习如何通过共享类之间的通用功能来避免代码重复。

本章要点回顾如下。

1. 子类是继承自基类的类型。举例来说，Armor 类型是继承自基类 Equipment 的子类。
2. 子类继承了基类的所有成员，比如变量和方法。同时，子类还可以声明自身特有的成员。这意味着子类型比上级类型更具体。
3. 构造函数链使得子类能够将其构造函数中与基类共享的参数“传递上去”，交给基类的构造函数处理。
4. 类型为基类的变量或参数可以存储该基类的任意子类的对象。例如， Item 类可以存储 Weapon 或 Armor 的实例，且不会引发任何错误。

第 12 章 调试

第 1 章简单介绍过调试的概念。现在，我们已经对编程有了一定的了解，是时候进一步探索代码编辑器的调试功能及其用法了。如果使用的代码编辑器不支持在 Unity 中进行调试，你将无法按照本章的步骤进行操作，不过，你仍然可以了解这些功能的工作原理和应用价值。本章将会使用 Microsoft Visual Studio Community 2019 来进行演示，但调试的基本功能在不同的编辑器中相差不大，所以即便使用其他编辑器（比如 JetBrains Rider），调试的界面和操作方式也应该是类似的。

调试是代码编辑器的一个强大功能，它允许开发者将代码中的任何一行标记为断点。当程序执行到这一行时，游戏或应用程序将会暂停，然后开发者就可以切换回代码编辑器并查看变量的值。这些值在暂停时是冻结的，并且可以在游戏暂停的情况下逐行执行代码。

在代码未能按照预期执行而需要找出原因时，就轮到调试功能大显身手了。它尤其适用于处理无法直接查看的数据的情况。一个常见的替代方法是通过调用 Debug.Log 来查看数据是否与预期不符。虽然这种方法有时候能够解决问题，但它通常既费时又费力。

利用调试功能，我们可以在问题区域放置一个断点，在程序执行到断点时，游戏就会暂停，我们可以方便地查看所有变量的当前值。举个例子，如果在脚本的 Update 调用过程中设置断点，则可以查看该脚本自身的实例变量（在脚本类中声明的变量）以及在 Update 方法内部声明的局部变量。然后，我们还可以逐行执行代码，甚至可以进入其他方法的调用中，逐行查看它们的执行情况。

12.1 设置调试器

如果还没有启动 Visual Studio，可以这样启动它：单击 Unity 编辑器窗口顶部的“资源”菜单按钮，然后单击下拉菜单底部的“打开 C# 项目”。如果之前通过 Unity Hub 与 Unity 一起安装了 Visual Studio，单击该选项应该会自动启动 Visual Studio。[①]

在 Visual Studio 中，顶部有一个按钮 Attach to Unity（附加到 Unity），如图 12-1 所示。

① 译注：如果在前面的章节中把默认编辑器设置成 VS Code，请在“编辑”|“首选项”|“外部工具”中的“外部脚本编辑器”里选择 Visual Studio。

图 12-1 圈出 Visual Studio 窗口顶部的“附加到 Unity”按钮

在 Unity 中进行调试之前，必须单击这个按钮，它会指示 Visual Studio 开始监听来自 Unity 编辑器的信号。

12.2 断点

在开始调试之前，先来探索一下通过设置断点来使程序在执行到某一行代码时暂停。如果不设置断点，即使将调试器附加到 Unity，也无法进行调试。

一种方法是单击代码行左侧的空白处（行号左边）设置断点。此外。也可以使用功能键 F9 来在文本光标（text cursor）当前所在的代码行上设置断点。

设置断点后，代码行左侧会出现一个红点。如果需要移除断点，可以再次单击这个点或再次按 F9。

让我们通过一个简单的例子来演示调试功能。新建一个名为“DebuggingTest”的脚本并将它的实例添加到场景中的一个游戏对象上。在这个例子中，游戏对象本身并不重要，所以随便选一个就可以。

这里不会用到 Update 方法，所以可以把它删掉。我们将使用 Start 事件来声明一个局部变量并将它的值更改三次，如下所示：

```
public class DebuggingTest : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        int a = 5;
        a += 5;
        a *= 2;
        a = 0;
    }
}
```

这段代码本身没有太大的意义，我们只是随意更改变量的值，以便通过调试器逐行查看它的变化。

在 int a = 5; 这一行添加一个断点，如图 12-2 所示。行号左边会出现一个点，同时这行代码会高亮显示。

图 12-2 在代码行左侧添加的断点

现在，请确保已经把 DebuggingTest 脚本附加到场景中的游戏对象上，单击 Visual Studio 中的“附加到 Unity”按钮，然后转到 Unity 编辑器并单击播放按钮运行游戏。

DebuggingTest 脚本实例中的 Start 方法将立即被调用，断点将使 Unity 编辑器中的游戏暂停运行。返回 Visual Studio，断点所在的代码行将高亮显示，表明代码已经执行到这里并暂停了下来。程序将不会执行后面的代码，除非我们给出指令。Unity 编辑器将会冻结，等待我们通过 Visual Studio 发出继续执行的指令。在那之前，游戏将保持暂停运行状态。

Visual Studio 底部的“局部变量”窗口显示了当前上下文中所有可用的变量——即断点所在的代码块中的变量。可以看到，目前变量 a 的值为 0，如图 12-3 所示。

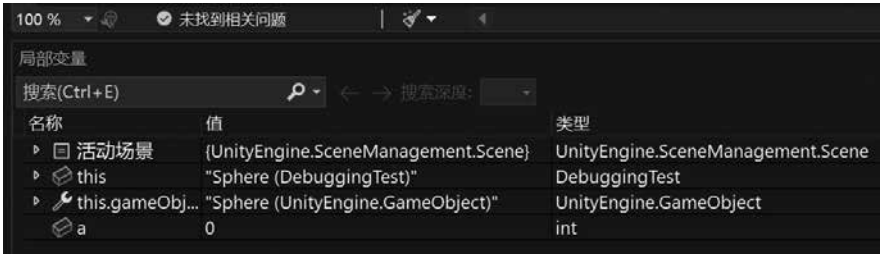

图 12-3 断点被激活，“局部变量”窗口显示当前上下文中的所有变量及其对应的值

窗口中还显示了关键字 this，它用于引用正在执行当前方法的类。由于关键字 this 关联的脚本包含许多成员，它旁边有一个小箭头，表示可以“展开”查看其内容。单击箭头就可以查看脚本的其他成员，不过这个例子比较简单，所以显示的成员并不多。在处理复杂的数据类型的时候，可以通过这种方式来查看其中的成员。

让我们把注意力放回变量 a 上。可以看到，它的值是 0。这是因为断点会在相关代码行执行之前暂停运行程序，而不是之后。因此，int a = 5; 还没有将值 5 赋给变量 a。

在程序被断点暂停后，原本的“附加到 Unity”按钮会变为“继续”按钮。按下这个按钮会使游戏继续运行并继续调试，也就是说，当程序运行到下一个断点时，游戏将再次暂停。但在这个例子中，游戏将不会再次暂停，因为 Start 方法中只设置了一个断点。

“继续”按钮右侧还有一些其他的选项，如图 12-4 所示。

图 12-4 Visual Studio 中的调试选项

下面将从左到右说明这些按钮的作用。

- 停止调试（快捷键 Shift+F5）：单击此按钮将结束当前的调试过程。游戏将在 Unity 中继续运行，并且不会在断点处暂停。
- 重新启动（快捷键 Ctrl+Shift+F5）：单击此按钮将重启调试器，但不会重启 Unity 编辑器中的游戏。在 Unity 开发环境中，这个功能可能不太常用，更常见的做法是单击“停止调试”按钮，在 Unity 编辑器中停止游戏，然后再次附加到 Unity 并运行游戏。
- 显示下一条语句（快捷键 Alt+Numpad *）：单击此按钮可以快速定位到即将执行的下一条语句。如果在调试过程中需要查看多个脚本文件，并希望快速回到当前断点所在的代码行时，这个按钮将很有帮助。
- 逐语句（功能键 F11）：在包含方法调用的代码行上暂停时，可以通过这个按钮让程序执行流程进入（步入）方法内部并在该方法的第一行代码处暂停（如果不包含方法调用，它的效果将与“逐过程”相同）。
- 逐过程（功能键 F10）：单击此按钮会运行当前代码行，然后再次暂停执行。这个按钮的英文是 Step Over（跳过），这可能会让人误以为它会直接跳过执行当前代码行，但实际上，这指的是它不会像“逐语句”那样步入当前代码行中的方法调用，而是会执行当前代码行，并移到下一行。
- 跳出（快捷键 Shift+F11）：单击此按钮可以执行当前方法的剩余部分然后再次暂停。如果不小心步入一个方法，就可以通过这个按钮跳出来。

这些调试控制工具都很实用。例如，在想要观察一个方法是如何逐步计算并最终返回结果的时候，可以通过“逐语句”按钮来逐行执行代码。执行完毕后，调试器将回到最初调用这个方法的代码行。如果一行代码中调用了多个方法，可以按照它们的执行顺序来逐一步入每个方法。假设我们在一行代码边设置了断点，这行代码不仅调用一个方法，还调用了许多其他方法来返回该方法的参数值，如下所示：

```
SomeMethod(A() + B(), C(), D());
```

在这种情况下，我们可以首先步入 A，再步入 B，然后是 C，接着是 D，最后是 SomeMethod 的调用。

现在，是时候将理论付诸实践了：我们已经设置了一个断点，不妨通过实际的操作来看看它的作用。在附加到 Unity 并运行游戏之后，程序将在声明变量的第一行代码处暂停。

单击“逐过程”按钮，可以看到当前代码行被执行，而下一行代码将会高亮显示。在“局部变量”窗口中，应该可以看到变量 a 的值更新为 5。

现在，程序在执行 a += 5; 前暂停了。再次单击“逐过程”按钮运行这行代码，可以看到

变量 a 的值更新为 10。接着运行 a *= 2，可以看到变量 a 的值从 10 变成 20。

当 Start 方法中的所有可执行代码行执行完毕后，如果再次单击“逐过程”按钮，程序将会跳转到下一行需要执行的代码。这甚至可能不是我们自己创建的代码，而是 Unity 引擎的内部代码。在这种情况下，最好利用断点来标记要检查的代码行，并在检查完毕后单击“继续”按钮，否则可能在各种脚本文件中迷失方向。

最后需要注意的是，这些控制按钮其实都不会跳过执行任何代码。代码总是会被执行的，这些按钮仅仅决定了在再次暂停之前要执行多少行代码。

12.3 善用 Unity 官方文档

调试是自行查找问题原因的一种手段。解决问题是程序员的主要职责之一：我们必须识别出问题所并找出解决问题的方法。那么，作为程序员，我们应该如何寻找有用的资源呢？

本书可以事无巨细地指导你执行各种操作并讲解方方面面的知识，但总有一天，你需要独立探索，自行解决遇到的问题。这决定着一个程序员的成败。实际上，真正的学习主要发生在你开始自主编写代码并逐步构建解决方案的过程中。毕竟，你的最终目标是独立开发游戏，不是吗？

在开发过程中遇到挑战时，获取相关信息是至关重要的。任何像样的搜索引擎通常都会将 Unity 的官方文档作为第一个结果返回给你。只需要在搜索栏中输入“unity”和想要了解的类名或组件类型（比如 Light 或 Rigidbody）即可。

Unity 将其文档分为两种形式：手册和脚本 API（Application Programming Interface，应用程序编程接口）。

- 手册（Manual）更像是一个教学页面，详细介绍了组件的工作原理和使用方式，通常配有图片和示例。它的目的是向用户介绍新概念。
- 脚本 API（Scripting API）则是专为程序员编写的技术文档，它介绍了 Unity 中的类和它们的成员，包括变量、属性与方法等。这个文档包含专门介绍每个字段的页面，其中描述了字段的用途，并提供了更多重要信息，比如变量存储的数据类型，方法的返回类型、方法接受的参数的类型和名称，方法的各种重载以及它们之间的差异等。它旨在帮助程序员了解 Unity 的内置类型和方法，并且通常会提供代码示例，展示如何使用这些成员或类型或与之进行交互。

后一种形式的文档在各种编程环境中很常见。其他游戏引擎很可能也会提供具有类似结构和布局的 API 文档，说明类型及其成员详细信息。举个例子，微软——C# 编程语言背后的公司——就提供了有关 C# 内置类型的文档。

举个例子，如果想了解如何在用户的计算机上保存和读取文件，可以了解一下与 System.IO 命名空间相关的内容。如果在浏览器中搜索关键字“System.IO 命名空间”，微软的官方文档应该会出现在搜索结果的前列。官方文档中列出了保存和读取文件所需要的各种类和方法，还配有详细的说明和代码示例。如果不知道相关的命名空间是什么，也可以使用更宽泛的搜索词，比如“如何在 C# 中读取文本文件”，然后根据搜索结果顺藤摸瓜，找到完成这项任务的方式。

高质量的 API 文档能够显著提升使用特定开发环境时的编程体验。有时候，我们不想阅读关于如何使用某个组件的完整教程，只想快速了解一下将要使用的数据类型和它们包含的字段。有了优质的文档，我们只需要简单地搜索一下就可以获得相关信息并迅速开始工作。

独立查找信息的能力可以在独立解决问题和实现所需功能上产生巨大的效用。

12.4 小结

本章介绍如何运行调试器和使用它的控制功能。虽然它目前还没有发挥太大作用，但如果代码出现问题，调试器将有望成为我们的救命稻草。调试器可以逐行执行代码并读取变量在每一步中的值，与简单调用 Debug.Log 相比，调试器能够更加有效地识别问题。另外，本章还简要说明了 Unity 文档的结构，讲解了如何独立查找信息。

第III部分
游戏项目 1：障碍赛

第 13 章 障碍赛游戏：设计与概述
第 14 章 玩家移动
第 15 章 死亡与重生
第 16 章 基本款危险物
第 17 章 墙壁和终点
第 18 章 巡逻者
第 19 章 漫游者
第 20 章 冲刺
第 21 章 设计关卡
第 22 章 菜单和用户界面
第 23 章 游戏内暂停菜单
第 24 章 尖刺陷阱
第 25 章 障碍赛游戏：总结

第 13 章 障碍赛游戏：设计与概述

现在，通过前面的描述，我们已经基本了解了 Unity 引擎的基础知识和编程的基本原理，是时候动手开发第一个示例游戏项目了。对于前面所学的知识，即使还没有完全的把握，也没关系，随着你逐步将这些概念应用到实践中，对它们的理解会愈发深刻。

在开始之前，请注意，我们开发的示例项目不会包含华丽的图形和音频。本书更侧重于这些元素背后的代码，而不是创建或导入美术资源和音频资源。我们这样做的目的是让你专注于游戏编程这一领域，直到你完全掌握。在读完这本书后，你可以根据自己的目标向不同的方向发展。许多独立游戏开发者对所有与游戏开发有关的领域（比如画画和作曲）均有涉猎，但你也可以利用互联网上丰富的资源（有些甚至是免费的），让自己有更多的时间来提升编程的技能。或者，也可以和一些志同道合的人合作开发游戏，把美术和音频方面的任务交给他们。

开发游戏是一项艰巨的任务，这涉及许多不同的部分和环节。在将这些部分组合到一起的过程中，情况可能发生变化——某些看似绝妙的点子在实际应用中可能不尽人意。这种情况在所难免，但这不足以成为不做准备就投入开发的借口。

过度计划和缺乏计划，两者之间存在一个微妙的平衡点，在这方面，我们可能永远无法做到完美。有时候，计划中的一些事项永远不会落地，原因可能很多，比如技术上无法实现、时间不够等。另一方面，一些开发者几乎不做计划，而是倾向于“佛系”推进项目。随着时间的推移，相信你会逐步找到适合自己的平衡点。

在本章中，我们将在动手开发之前做一些规划，这有助于我们更快达到预期目标，减少不必要的重复工作。

13.1 游戏玩法概述

先来看一看游戏的玩法。游戏将采用一个从上方俯瞰玩家角色的俯视角摄像机。玩家使用 WASD 键或箭头键来实现移动，在按住移动键的同时按空格键可以进行冲刺。每次使用冲刺技能后都有一个短暂的冷却时间，但除此之外，使用冲刺技能不会有任何消耗。

游戏关卡中的墙壁将用立方体来构建，并且这些立方体的颜色有别于地板的颜色，以便进

行区分。玩家一旦接触这些墙壁就会触发碰撞机制，从而被有效地限制在预设的游戏场地内。

在游戏中，玩家的目标是避开障碍物，成功地抵达终点。每个关卡都设定了起点和终点，玩家从起点出发，最终需要到达终点（一个简单的圆形领奖台）。如果玩家因为碰到障碍物而死亡，他们将会从起点重新开始，在顺利抵达终点后，游戏才会结束（当然，玩家也有可能因为挫败感太强而退出游戏）。

障碍物包括以下几种。

- *巡逻危险物*（Patrolling hazard）：它们会沿着预设的路线行动，完成一轮后返回起始点，并继续下一轮循环。场景中的每个巡逻障碍物都有不同的路线。作为危险物，它是致命的，会在接触玩家时杀死玩家。
- *投射物*（Projectile）：投射物由障碍物发射，它们沿着直线飞行，直至撞到墙壁。投射物同样会在接触玩家时杀死玩家。
- *游荡危险物*（Wandering hazard）：每个游荡危险物的行动范围都被限制在单独的矩形区域内，它们会时不时地随机选择一个新的目标点。在确定了新的目标点之后，这些障碍物会缓慢旋转以面向目标点，然后开始向它移动。开始移动之前的旋转是为了给玩家留出反应时间，以便及时闪避。
- *尖刺陷阱*（Spike trap）：它们会定期激活，升起一圈致命的尖刺然后再降下来。玩家需要在尖刺降下时通过陷阱。

主菜单将提供选择关卡的功能，玩家可以逐一浏览各个关卡的地图，然后在准备就绪后，通过单击按钮来开始游玩当前选择的关卡。

在游戏过程中，如果玩家希望退出当前关卡，可以通过按下 Esc 键调出游戏内的菜单系统，并返回主菜单。

我们首先实现最重要的东西。游戏开发者通常会首先集中精力开发游戏的核心机制，因为直接决定了游戏是否好玩。如果游戏的核心玩法缺乏趣味性，菜单和关卡选择界面做得再好也无济于事，毕竟玩家是来玩游戏的。

这个游戏的核心机制包括玩家移动、障碍物设置以及玩家死亡和重生。我们将在同一个场景中实现这些核心机制，在确保它们能够如期工作后，再去实现关卡选择界面等细节。

13.2 技术概览

在着手开发游戏之前，还需要概述自己想要实现哪些内容，并思考如何在游戏引擎中实现它们。随着经验的积累，这个过程会变得越来越得心应手，但即使是新手，也应该尝试在着手

开发之前弄清楚每个功能的基本实现思路。在这个过程中，很可能会发现一些之前未曾考虑到的问题或原计划的缺陷。此外，在脑海中对项目有一个整体认识，有助于指导游戏开发过程。

13.2.1 玩家控制

玩家角色由两个立方体组成：一个较大的立方体作为主体，顶部再放置一个不同颜色的小立方体，指向玩家角色的局部前进方向（否则很难看出玩家角色面向哪边）。如图 13-1 所示，底部的箭头标示玩家的局部前进轴。

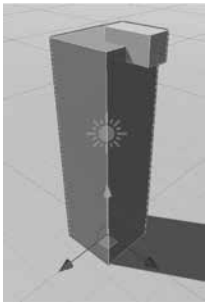

图 13-1 玩家面向摄像机

Player 脚本将附加到根游戏对象——一个空的对象上。两个代表玩家的立方体将作为子对象添加到这个根对象上。在根据玩家的最新移动方向调整玩家角色的朝向时，只需要旋转这些立方体，根游戏对象则保持不变。

如此一来，就可以将摄像机设置为玩家角色的子对象，使其跟随玩家移动。由于根游戏对象本身不会旋转，摄像机也不会旋转——如果摄像机也跟着旋转的话，场面会相当尴尬。我们希望摄像机和玩家模型的相对位置保持不变，即使模型旋转时也是如此。

Player 脚本包含所有与玩家行为相关的功能，主要涉及玩家的移动和闪避。

考虑到不同玩家的操作偏好，玩家的移动操作可以通过 WASD 键或箭头键实现。我们将为移动设置动量，使玩家需要一定的时间来起步和停止。不过，这个效果不会太夸张，否则玩家可能会觉得路太滑。我们需要保持操作的精确性，以避免玩家角色在应该停下的时候滑出去太远，但也得有一点动量，不然玩家角色的移动会显得非常生硬。

玩家可以通过按下空格键来进行闪避，这将使角色迅速冲向前方。

这个动作很快会结束，但在此期间，常规移动不会被计算，以确保只有闪避速度会生效。此外，在按下空格键时，玩家必须正在朝着某个方向移动，否则闪避动作不会被触发。

13.2.2 死亡与重生

在玩家死亡后，他们会在一小段时间后在关卡的起点重生。为了避免玩家在死亡状态下继续控制角色移动，需要在重生之前禁用 Player 脚本。学习如何禁用脚本是一个重要的知识点，因为它在游戏开发中是一个非常有用的功能。

13.2.3 关卡

关卡的设计非常简单，其中，一个具有独特颜色的平面将被用作地面，被放置在玩家下方，

平面上有一些被用作墙壁的方块，可以根据需要调整它们的大小。当玩家移动时，他们将与墙壁碰撞，因此不能离开游戏场地。这是一个俯视角游戏，将使用正交摄像机，这种摄像机不使用透视效果，有点像以 2D 的方式观察世界。因此，在玩家的视角里，墙壁和地板其实都是同样的平面，只能通过颜色来区分两者。玩家看不到墙壁的侧面（因为没有透视），游戏中也不会有阴影或光照效果。这听上去可能不太好理解，但不用担心，在进行到相关阶段时，这些概念很快会变得清晰起来。

这种关卡设计方法虽然稍显粗糙，但它的首要优势在于功能性强且能够迅速完成。如果目标是开发一款商业游戏，自然需要考虑使用更专业的美术资源和更精细的关卡设计，但这并不是我们当前的目标。

为了让玩家能够通关，需要使用一个 Goal 脚本来检测玩家何时走到（或者“滑到”）终点，并让成功通关的玩家返回关卡选择界面。

13.2.4 关卡选择界面

每个关卡都将拥有一个独立的场景。游戏启动时，默认加载的是主场景，这个场景中集成了游戏菜单代码。玩家可以通过菜单来逐一选择各个关卡，并且可以加载所选关卡的场景以进行预览。在加载新的场景时，游戏会自动清理并移除先前的场景。

每个关卡场景都将配备一个预览摄像机，它将被放置在关卡上方，让玩家能看到关卡的完整布局。

在玩家开始游玩某个关卡后，游戏禁用预览摄像机并切换到玩家摄像机。

13.2.5 障碍物

在前面的章节中，我们在刚开始学习组件的时候简单讨论过障碍物的设计概念。现在，是时候考虑如何为这些障碍物创建脚本了。

- Hazard 脚本：使游戏对象在接触玩家时杀死玩家。
- Wanderer 脚本：使游戏对象无规律地游荡。
- Patroller 脚本：用于为游戏对象设置循环移动的路径。
- Shooting 脚本：使游戏对象定期发射投射物。
- SpikeTrap：使致命的游戏对象迅速向外弹出，然后回到原位，恢复成无害的状态，在设定的等待时间过后再次激活。

通过在游戏对象上混合使用这些脚本，可以实现我们设计的所有障碍物。

13.3 项目设置

现在，准备一个新的 Unity 项目来开发这款游戏。打开 Unity Hub 并单击右上角的“新建项目”按钮来创建一个新项目。前面进行过类似的操作，所以这里弹出的窗口应该并不陌生。选择 3D 模板，把新建的项目命名为“ObstacleCourse”，并将其保存到合适的文件夹中，如图 13-2 所示。

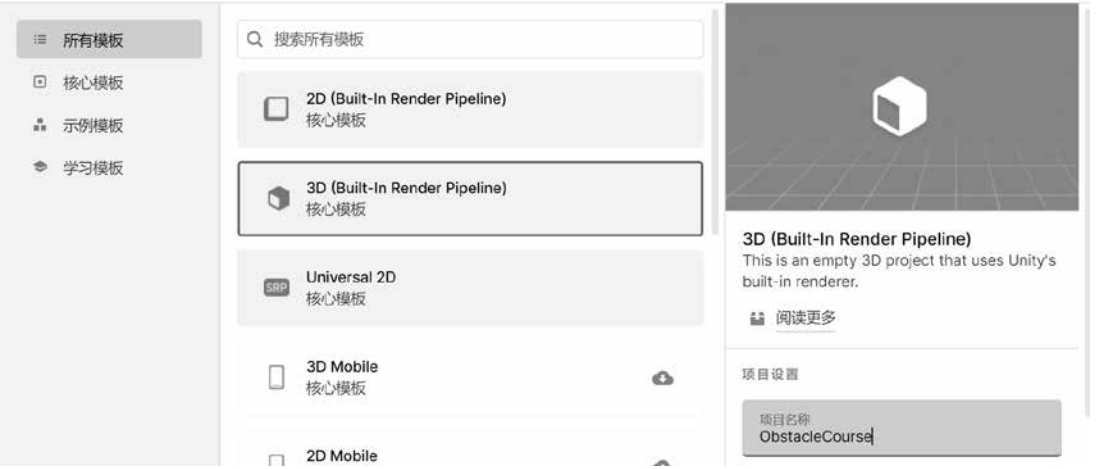

图 13-2 使用 Unity Hub 中默认的 3D 模板新建 ObstacleCourse 项目

在“项目”窗口中导航到 Assets 文件夹，可以看到其中已经有一个 Scenes 文件夹。为了给 Scenes 文件夹“做个伴儿”，我们再在 Assets 文件夹中新建三个文件夹，并分别将它们命名为 Materials、Prefabs 和 Scripts。

现在，将 Assets/Scenes 文件夹中默认包含的 SampleScene 资源重命名为“main”。这可以通过在“项目”窗口中右键单击资源然后选择“重命名”来完成。重命名完成后，Unity 可能会弹出一个窗口来询问是否想要重新加载场景，选择“继续”即可。

场景将默认包含一个 Directional Light 和 Main Camera，可以把它们保留下来。另外，我们还需要为游戏创建一个简单的地面。

- 依次选择顶部菜单左侧的游戏对象、“3D 对象”和“平面”。
- 如果尚未选中 Plane，请在“层级”窗口中选中它，然后查看“检查器”窗口。
- 将 Plane 重命名为“Floor”。
- 如果 Transform 组件的位置不是（0，0，0），请改为（0，0，0），以确保它位于世界原点。
- 为了确保地面足够宽敞，能容纳游戏关卡，将它的缩放值设置为（1000，1，1000）。

在设置完成后，“项目”窗口应该和图 13-3 一致。

图 13-3 新增文件夹并重命名场景后的“项目”窗口

13.4 小结

本章确定了这一示例项目的玩法：一个简单的俯视角障碍赛游戏，玩家将使用 WASD 键或方向键来移动，尝试避开危险的障碍物，顺利抵达关卡终点。对游戏玩法有了清晰的概念后，我们是时候开始循序渐进地实现游戏的各个组成部分了。

第 14 章 玩家移动

在新的项目中，我们首先着手实现玩家的移动操作。为了锻炼编程技能并学习一些新的方法，我们将采用一些比较花哨的设计。

就像第 13 章提到的那样，在理想情况下，玩家将能够流畅地移动——既不显得生硬，也不至于太滑。玩家从静止加速到最大速度以及从最大速度减速到停止，都应该有一个平缓的过渡期。这样的过渡不会太长，因为这种类型的游戏需要精确地操控角色，所以千万不能让玩家感觉自己刹不住脚。

接下来要讲到的内容涉及前面讨论过的概念，所以你应该不会觉得太陌生。

游戏将逐帧处理玩家的移动速度。速度由 Vector3 数据类型表示，它存储了 X、Y 和 Z 三个轴的值，用来描述玩家在移动过程中的位置变化。游戏使用“单位 / 秒”作为度量单位。例如，如果速度是（15，0，12），那么玩家将以每秒 15 个单位的速度向右移动，以每秒 12 个单位的速度向前移动。玩家不会在 Y 轴上移动，因为在这个游戏中，玩家不能上下移动，只能进行水平方向的移动，包括前进、后退、向左和向右。在游戏中，玩家可能会将 Z 轴上的移动称为“向上”和“向下”移动。这是因为在使用上下箭头操控角色时，实际上就是在 Z 轴上移动，但实际上，在世界空间中，Z 轴代表的是前进和后退。

由于游戏摄像机从上方向下俯瞰玩家，如果玩家真的在世界空间中“向上”，就意味着他们正在朝着摄像机移动。因此，在这个游戏中，玩家的速度不会涉及 Y 轴。玩家角色将始终位于平坦的地面上，Y 轴上的速度永远是 0，因为这只会使他们逼近或远离摄像机。

为了让玩家角色流畅地移动，我们将根据玩家的按键输入来提高或降低速度，并根据这个速度不断地更新玩家角色的位置（每秒一次）。WASD 和箭头键都可以用来控制角色，玩家可以根据自己的偏好进行选择。在按住相应的移动键时，速度将逐渐变化，从而实现理想的移动效果。

在移动时，玩家的角色模型需要朝向他们移动的方向。为了更直观地表示这一点，我们将创建一个由几个立方体组成的简单模型，其中一个立方体会指向玩家的局部前进轴，清晰地表明模型面朝哪个方向。虽然这可能不够美观，但它有效地完成了任务，让我们得以专注于编写代码的工作。

14.1 创建 Player 游戏对象

现在，是时候开始设置 Player 游戏对象了。在当前阶段，所有开发工作都将在 main 场景中完成，它将暂时充当一个试验场，用于进行各种开发和测试。

图 14-1 展示了设置完 Player 游戏对象后的“层级”窗口，稍后将逐步讲解创建各个游戏对象的过程。“层级”窗口中的 Floor 是在上一章中创建的，它将被用作游戏的地面。Floor 是一个简单的平面，其位置和缩放分别是（0，0，0）和（1000，1，1000）。

图 14-1 设置 Player 游戏对象

在移动玩家角色时，实际上移动的是根游戏对象 Player 的 Transform 组件。当然，这意味着 Player 的子对象也会随之移动。然而，游戏对象 Player 的 Transform 组件不会旋转，因为它关联着摄像机。请记住，子对象会表现得像是与父对象有物理上的连接一样，所以如果旋转 Player 游戏对象，摄像机将随之一起旋转，这无疑会使玩家感到大为震惊和疑惑。

为了避免这种情况，我们创建了空对象 Model 作为 Player 的子对象，它是“模型容器”（model holder），专门用来存放所有与模型相关的游戏对象。脚本引用空对象 Model 的 Transform 组件来进行旋转，使玩家模型面朝移动的方向。通过这种方式，（与摄像机相关联的）根游戏对象将不会旋转，只有玩家模型会旋转。

Model 有两个立方体作为子对象，分别为 Base 和 Top。以下是创建 Player 对象及其所有子对象的步骤。

1. 创建一个空游戏对象（在 Windows 系统上使用快捷键 Ctrl+Shift+N，在 Mac 系统上使用 Cmd+Shift+N）并将其命名为“Player”。
2. 为 Player 创建一个空的子对象（在选中 Player 的前提下，在 Windows 上使用快捷键 Alt+Shift+N，在 Mac 上使用快捷键 Opt+Shift+N）并将其命名为“Model”。确保它的局部位置是（0，0，0），与 Player 重叠。
3. 右键单击“层级”窗口中的 Model 并通过“3D 对象”➤“立方体”为 Model 创建一个子对象。将这个立方体命名为“Base”，并将其局部位置设置为（0，2.5，0），缩放设置为（1.4，5，1.4）。这里的缩放是局部的，但由于 Base 的所有父对象的缩放均为 1，所以它的局部缩放与世界缩放相等。Base 的局部 Y 位置的值被设置为 Y 缩放值的一半，以确保立方体的底部与枢轴点对齐，而不是以中心点对齐（枢轴点是 Player 游戏对象的位置）。

4. 创建第二个立方体作为 Base 的子对象，并将其命名为“Top”。把它的位置值设置为（0，0.5，0.5），缩放值设置为（0.33，0.1，0.7）。
5. Main Camera 游戏对象应该已经存在于场景中了。如果没有的话，就通过“游戏对象”➤“摄像机”来创建一个 Camera，然后将其重命名为“Main Camera”。将摄像机拖到 Player 游戏对象上，使它成为 Player 的子对象（注意，不要把它设置为 Model 的子对象）。现在，将它的局部位置值设置为（0，24，-5），旋转值设置为（70，0，0），这样的设置让摄像机位于玩家角色上方稍微靠后的位置，并向下倾斜了一定的角度。图 14-2 展示了 Main Camera 的 Transform 组件在“检查器”窗口中的设置。

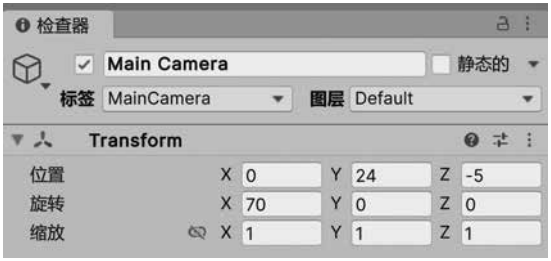

图 14-2 Main Camera 的 Transform 组件在“检查器”窗口中的设置

6. 最后，将 Player 游戏对象从“层级”窗口拖入“项目”窗口的 Prefab 文件夹，为 Player 创建一个预制件。

此时的“层级”窗口应该和图 14-1 相一致。

Top 立方体将指向玩家的前进轴（即玩家面向的方向），这样就能直观地识别出当前的方向。

14.2 材质和颜色

现在，是时候处理一下立方体的颜色了。两个立方体将有不同的颜色，以便轻松地区分 Top 和 Base。为了实现这一点，需要 Unity 中的一种内置资源——材质（material）。在 Unity 中，材质用于存储关于如何渲染网格（或 2D 图像）的信息。最值得注意的是，材料允许我们将纹理应用到网格上。纹理本质上是普通的 2D 图像，可以被拉伸并包裹到一个网格（即 3D 对象）上，从而为原本的单色平面对象赋予岩石、树皮、瓷砖地板、石膏墙等外观。不同游戏的美术风格各异，一些游戏会使用真实物体的照片，另一些游戏则可能采用风格化的图画。

材质提供了多种设置选项，但它们主要与将美术资源整合到游戏项目中有关。就像前面提到的那样，本书不会深入探讨这方面的内容。在这个项目中，我们只会使用材质最基础的功能——用它来改变游戏对象的颜色。

在“项目”窗口中，右键单击 Materials 文件夹并选择 Create，然后选择 Material。创建两个材质，并分别将它们命名为“Player Base”和“Player Top”。选中任意一个新创建的材质，会看到“检查器”窗口中显示了一对可编辑的字段，看上去十分复杂。幸运的是，我们只需要调整其中的一个小控件。“Main Maps”粗体文本下面的“反射率”一词右侧有一个颜色栏（参见图 14-3 中圈出的部分），旁边有一个滴管图标。

图 14-3 Player Base 材质的“检查器”窗口

这是材质的颜色字段，它默认是白色的。“反射率”右侧的矩形显示了材质当前使用的颜色。单击这个白色矩形将弹出一个“颜色”窗口，可以在其中更改颜色，如图 14-4 所示。除此之外，也可以单击滴管图标，然后单击屏幕上的任意位置来选取那里的颜色。

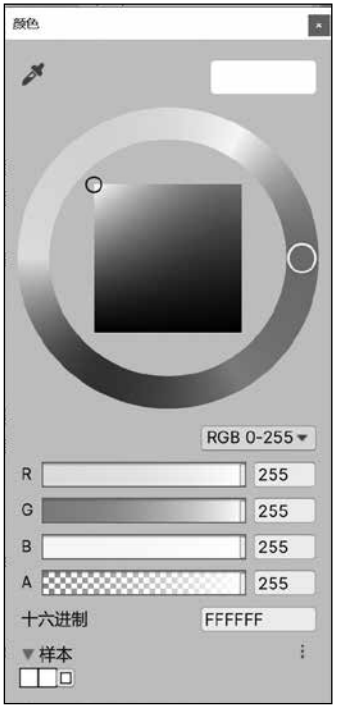

图 14-4 “颜色”窗口

现在来简单了解一下计算机是如何看待颜色的。这是游戏开发者经常遇到的概念，即使主要负责的是编程。别担心，这个概念并不复杂。

颜色弹出窗口的上半部分是一个交互式色轮，中间有一个矩形区域。单击色轮中的颜色可以改变色相，而单击矩形区域可以调节颜色的饱和度和亮度。

窗口的下半部分提供了几个可以设置数值的输入框。前三个字段对应着颜色的各个分量，它们的左侧都有一个字母标识，右侧有一个彩色方框，紧随其后是一个显示数字的小方框。数字代表了颜色分量的具体数值，字母则是对应的颜色属性的缩写。彩色方框是一个可视化工具，它形象地展示了当该分量的数值增加或减少时，颜色将如何变化。可以通过单击彩色方框或拖动方框内的滑块来改变颜色分量，或者如果对颜色数值比较熟悉，也可以直接编辑数字。

那么，颜色组件（color component）是什么呢？这取决

于使用的颜色模型。计算机主要使用两种模型来描述颜色：RGB（红、绿、蓝）和 HSV（色相、饱和度、明度）。每种模型以不同的方式表示颜色，但它们之间可以轻松地相互转换。可以通过色轮下方的下拉菜单来更改颜色模型，它的默认设置是 RGB（0-255）。

HSV 模型往往更容易理解。色相（hue）指的是颜色——红色、绿色、蓝色、紫色、青色、黄色等。它是一个范围为 0 到 360 的值，与色轮上的角度相对应。饱和度（saturation）是一个范围为 0 到 100 的值，表示颜色的鲜艳度。饱和度越低，颜色就会显得越灰。明度（value）表示颜色的亮度。它的值越低，颜色就越接近于黑色。明度的范围也是 0 到 100。

RGB 模型则表示颜色中红色、绿色和蓝色的相对含量，其数值可以是从 0 到 255 的整数，或者是从 0 到 1 之间的分数。通过按照不同比例混合这三种原色，可以产生各种不同的颜色。例如，将 RGB 设定为（0，0，0）则可以显示黑色，设定为最大值（255，255，255）可以显示白色，黄色则是红色和绿色混合的结果（255，255，0）。如果一个颜色的三个分量都比较均匀，那么它的饱和度会相对较低；如果一个颜色的一个或两个分量比较高，就会显得比较鲜艳。这种颜色经常出现在玩具或豪华跑车上。如果所有分量的数值都比较低，颜色将会显得比较暗沉。

现在，前三个字段的含义已经很清楚了。那么，旁边写着字母 A 的第四个字段又代表什么呢？

这个字段代表 Alpha 值，它存在于所有颜色模型中。Alpha 值决定了颜色的透明度，范围是 0 到 100。现在更改它不会产生任何效果，因为这个材质不支持设置透明度。但在常规情况下，Alpha 越低，颜色就越透明。

至于最底部的字段，它表示颜色的十六进制（hexadecimal）颜色代码，有时被简称为 hex 颜色代码，这是一个用于描述颜色的字符串，由字母和数字组成。如果更改这个字段，所有颜色分量都会被更改。十六进制颜色代码对于用户来说可能不太直观，毕竟它看上去就像是一串随机的字符组合，但它提供了一种设置颜色的简便方式，只需要输入我提供的十六进制颜色代码，就可以获得我正在使用的颜色（如果需要的话）。因此，接下来我会使用十六进制颜色代码来描述材质的颜色。

我准备把 Base 设置为深蓝色，Top 设置为浅蓝色。你完全可以发挥自己的创意，选择自己喜欢的颜色，但如果想参考我的设置，那么 Base 的十六进制颜色代码是 7CBAD0，Top 的是 B6DAF5。只需要将这些代码输入到对应材质的颜色选择器中的十六进制字段，得到的颜色就可以与我的相同。

应用材质

设置好材质之后，就可以将它们应用于相应的立方体游戏对象了。从技术上讲，这是通过“检查器”窗口中的 Mesh Renderer 组件来完成的，但更为简便的方法是直接把材质资源从

“项目”窗口拖动到“层次”窗口或“场景”窗口中的目标游戏对象上，使材质自动应用于 Mesh Renderer 组件。注意，请把 Player Base 材质应用到较大的立方体上，将 Player Top 应用到顶部的小立方体上。

如果感兴趣的话，还可以创建一个新的材质并将其应用于 Floor 对象，把它设置成不同的颜色。我设置了一个十六进制颜色代码为 C3D7C4 的颜色，使 Floor 对象变成了类似薄荷绿的灰绿色。

至此，Player 对象的层级结构已经设置完毕，并且模型也变成了彩色。接下来，我们该动手编写代码了。

14.3 声明变量

在开始为玩家的移动编写代码前，需要先创建一个脚本，后者将作为组件添加到 Player 对象上。在“项目”窗口中的 Scripts 文件夹上单击右键，选择 Create ➤ C# Script 并将新建的脚本命名为“Player”。将脚本实例作为组件附加到名为 Player 的根游戏对象上，具体做法为将 Player 脚本从“项目”窗口拖到“层级”窗口中的 Player 对象上。

现在，在“项目”窗口中双击打开新建的 Player 脚本。如果代码编辑器没有启动，则可能需要在顶部菜单栏的“编辑”|“首选项”|“外部工具”中的“外部脚本编辑器”中选择想要使用的编辑器。

在脚本中，首先声明用于控制玩家移动的变量。这些变量的声明位于脚本类的开头处：

```
public class Player : MonoBehaviour
{
    // 引用
    [Header("References")]
    public Transform trans;
    public Transform modelTrans;
    public CharacterController characterController;

    // 移动
    [Header("Movement")]
    [Tooltip(" 每秒移动的单位数，最高速度。")]
    public float movespeed = 24;

    [Tooltip(" 达到最高速度所需的时间，以秒为单位。")]
    public float timeToMaxSpeed = .26f;
```

```
    private float VelocityGainPerSecond { get { return movespeed / timeToMaxSpeed; } }

    [Tooltip("从最高速度到静止所需要的时间，以秒为单位。")]
    public float timeToLoseMaxSpeed = .2f;

    private float VelocityLossPerSecond { get { return movespeed / timeToLoseMaxSpeed; } }

    [Tooltip("尝试向与当前行进方向相反的方向移动时的动量乘数(例如,玩家在向左移动时试图向右移动)。")]
    public float reverseMomentumMultiplier = 2.2f;

    private Vector3 movementVelocity = Vector3.zero;
}
```

这段代码涉及的大部分知识点都在之前的章节中讲过了，但 VelocityGainPerSecond 和 VelocityLossPerSecond 的声明比较陌生。它们是“属性”（property），[①] 我们还没有深入讨论过这个概念。现在不理解它们的含义也没关系，接下来我们很快就会解释清楚。

这段代码首先声明了对计划使用的其他组件的引用。声明组件类型（如 Transform）的变量后，就可以直接在“检查器”窗口中设置该变量的值，这样就不需要通过编写代码来获取特定的组件了。只需要在“检查器”窗口中设置一次引用，就可以在代码中的任何地方使用它们。为了让“检查器”窗口的布局更加清晰整洁，第一个引用 trans 设置了 Header 属性（attribute），为这些引用添加了一个粗体标题。

trans 变量指向 Player 对象的 Transform 组件，这样做主要是出于性能考虑——比起每次都使用 Unity 的内置成员 transform，脚本直接持有一个对 Transform 的引用会比较高效。modelTrans 引用了空对象 Model 的 Transform 组件。最后的 characterController 则引用了一个尚未创建的组件 CharacterController。这个组件是控制角色移动的关键，本章稍后会介绍并添加这个组件。

脚本的下一部分包含一个注释 // 移动和一个用于区分与移动相关的变量的新标题。这段代码还使用了 Tooltip 属性（attribute）来为每个变量提供描述。将鼠标悬停在 Unity“检查器”窗口中的变量字段上的时候，这些描述会显示出来。它们不仅像代码注释一样有助于理解代码，

① 译注：在 C# 语言中，虽然 property 和 attribute 都可以被译作“属性”，但两者的定义并不相同。property 是一个类的成员，用于控制对类的一个特定字段的访问。它包含 get 和 set 方法，用于获取或更新该字段的值。例如，在 Person 类中，Name 可以作为一个 property 存在，让我们可以方便地获取或设置一个人的姓名。attribute 则用于为程序的部分元素（如类、方法或属性）添加元数据。这些元数据可以在编译阶段或程序运行时发挥作用，指导程序的行为或提供有关信息。例如，“[Serializable]”就是一个 attribute，在类上添加这个 attribute 时，它会告诉编译器这个类的实例可以被序列化。

还能帮助在 Unity 引擎中显示变量的具体用途。在与其他人合作开发游戏时，这些描述可以帮助他们理解如何使用你编写的脚本。

虽然工具提示简单说明了各个变量的用途，但为了加深理解，让我们再集中回顾一下各个变量的定义。

- movespeed 是玩家每秒能达到的最大速度。
- timeToMaxSpeed 是玩家从静止状态加速至最大速度所需要的时间，单位是秒。
- timeToLoseMaxSpeed 是玩家松开移动键后，从最大速度减速到完全停止所需要的时间。
- reverseMomentumMultiplier 用于帮助玩家更容易地在一个方向上停止移动，并立即开始向相反方向移动。当玩家开始向反方向移动时（例如，玩家原本在向左移动，但突然调头向右移动），他们获得的反向速度将乘以这个数值。因此，如果希望玩家在反向移动时能够以双倍速度加速，就需要将 reverseMomentumMultiplier 其设置为 2。如果希望速度提升 50%，就将其设置为 1.5。虽然我们不打算这么做，但如果这个值低于 1，反向移动就会更加困难。设置这个变量是为了让玩家的移动操作更加灵敏。

这些变量都可以从 Unity 编辑器的检查器中访问。如果需要，随时可以通过更改这些变量来调整玩家的移动特性。随着游戏障碍物和机制的逐步完善，这些数值可能会根据实际体验进行优化。

代码最后一行的 movementVelocity 是一个私有的 Vector3 类型变量。Vector3 是 Unity 引擎提供的数据类型，用以表示 X、Y、Z 这三个轴的向量。正如前文所述，向量可以表示从旋转角度（每个轴在 0 到 360 之间）到位置和缩放的各种事物。在这里，它被用来表示玩家的当前速度，单位是每秒移动的单位数。它的值在 X 轴和 Z 轴上的范围是正负 movespeed，而 Y 值永远不会改变。

14.4 属性

VelocityGainPerSecond 和 VelocityLossPerSecond 是我们创建的第一批属性（property）。通过声明这些属性，我们把一些常用的数学运算“折叠”到更简洁且直观的名称中。如此一来，每次需要进行这些运算的时候，就可以直接输入名称而不是数学表达式了。记住，不要重复自己！

属性可以看作是变量和方法的结合体。属性的声明方式类似于变量，但后面跟的是代码块而不是分号，并且它们没有默认值。在引用和设置属性时，我们可以像处理变量一样对待它们：既可以读取属性的值，也可以给它们设置新的值。

但在声明属性时，需要定义在引用它们的值时返回的内容和 / 或在设置它们的值时发生的

操作。在这方面，属性表现得很像方法。在属性的代码块中，可以声明 getter 和 setter。顾名思义，它们分别使用 get 关键字和 set 关键字来声明，并且后面都跟着一个代码块。

getter 是在属性被读取时执行的代码块。和方法一样，它必须使用 return 关键字来返回一个值。

setter 则是在属性被设置为新值时执行的代码块。它不返回任何值。在 setter 代码块中，可以通过 value 关键字来访问即将被赋给属性的新值。

属性可以选择性地声明 getter、setter 或者两者都声明。声明了哪个就能执行哪种操作。也就是说，如果属性没有声明 getter，就不能获取它的值；同理，如果属性没有声明 setter，就不能设置它的值。前面的代码并没有声明 setter，因为我们不打算设置这些属性的值，它们只是一个执行常用数学运算的快捷方式。因此，前面的代码只声明了 get 块，并在其中编写了一行代码来返回数学运算的结果。

由于格式的原因，这段代码看起来可能有些乱。就像之前提到的那样，C# 语言并不真正关注换行和缩进，这不是语法的一部分。当计算机读取代码时，它并不会使用换行和缩进来判断语句和代码块的结束，而是会通过花括号和分号来判断。无论把一条语句拆成多少行，只要恰当地使用分号来结束那个语句（并且没有在任何名称中间插入空格、制表符或换行符），计算机就能够正确地理解代码的意图。使用换行和缩进只是为了保持代码的一致性和可读性，而不是因为语法的强制要求。

在一些情况下，可以根据需要对这些规则进行调整，甚至打破它们。在属性声明中，我们就是这样做的。由于它们只是 get 代码块中的一行简单代码，我们没有使用常规的格式，而是把声明写在了单行代码中。

常规的格式可能是下面这样的：

```
private float VelocityGainPerSecond
{
    get
    {
        return movespeed / timeToMaxSpeed;
    }
}
```

如果你喜欢的话，采用这种常规方式也是完全可以的。有些人认为只有这种方式才是“正确的方式”，但我认为单行声明更为简洁，所以更倾向于采取那种方式。不过，如果我的 getter 包含的代码多于一行，或者有 setter，我肯定也会采用常规方式编写声明。这个例子很好地说明了一点：有时候，适当地打破一些传统的代码格式化“规则”是无伤大雅的，在开发个人项目的时候尤其如此。

14.5 跟踪速度

前几节充分地介绍了脚本中的可用变量，而在这一节中，我们将开始编写每帧更新的逻辑，赋予玩家角色移动的能力。这部分逻辑与 movementVelocity 变量紧密相关，因为它定义了每秒的移动距离。

在开始编写 Update 方法中的代码之前，最好先将 Update 方法中的不同逻辑块拆分成更小的方法——这是一个值得培养的好习惯。

例如，为了将移动逻辑与 Update 方法中可能存在的其他逻辑分离开来，可以通过 private void Movement() 来简单地声明一个方法，在其中编写所有与移动有关的逻辑，然后在 Update 方法中调用这个方法。这种做法对于处理特别复杂的脚本非常有帮助，因为代码编辑器提供了折叠代码块的功能，让我们能够把暂时不需要处理的代码隐藏起来。

在着手编写具体的移动逻辑之前，不妨先来看看 Movement 方法的大致结构。当然，所有这些代码都嵌套在脚本类的代码块中，位于前几节中声明的变量下方：

```
private void Movement()
{
    // 在此处编写实现移动的代码
}
private void Update()
{
    Movement();
}
```

Movement 方法被声明为一个简单的私有方法，它不返回任何内容，并且被紧随其后的 Update 方法所调用。如果未来需要在 Player 脚本中添加新的功能，也可以采用同样的方式来将新的逻辑单独放在一个代码块中。

现在，是时候开始为 Movement 方法编写代码了。请注意，这些代码每帧都会执行，因此它们会以小而频繁的增量持续运行。所以，任何需要以每秒为频率应用的值都必须乘以 Time.deltaTime，就像第 10 章中做过的那样。

首先，我们将使用一系列 if 和 else 语句块来根据玩家的按键输入以及 movementVelocity 变量的当前状态来更改 movementVelocity（以便在向相反方向移动时应用 reverseMomentumMultiplier）。之后，我们将检查在这一帧是否存在任何 movementVelocity，如果存在，就使用它来控制玩家的移动。

接下来，我们将从 Z 轴入手，逐步实现角色的前进和后退功能。第一个要实现的是 W 键或上箭头键的按键检测，以处理前进速度。为了确保速度不会超过任意方向上的 movespeed，

需要使用简单的数学方法 Mathf.Min 和 Mathf.Max。它们各自接收两个浮点数参数，并返回这两个数值中的较小值（Mathf.Min）或较大值（Mathf.Max）。很好理解，对吧？

这种方法经常被用来对一个可变的值设置上限。与传统的先增加数值再检查是否超出上限的方法相比，使用 Mathf.Min 或 Mathf.Max 可以用一行代码简洁地实现相同的效果。我们不是通过增加或减少的方式来调整某个数值，而是直接将其设置为 Min 或 Max 调用的结果。以下是一个使用这种方法增加数值的示例：

```
value = Mathf.Min(maximumValue, value + addedAmount);
```

在这个示例中，value 变量的值被设置成 Min 方法的结果。Min 函数接受两个参数，并返回两者中较小的值。第一个参数是为 value 设定的最大值（maximumValue），第二个是 value 当前的值加上想要增加的值。如果相加后的 value 的值低于最大值，那么它就会作为结果返回。但如果超过最大值，maximumValue 就会作为结果返回。

同样的逻辑也可以应用于减法操作：

```
value = Mathf.Max(minimumValue, value - addedAmount);
```

这段代码使用了 Mathf.Max，并且执行减法操作而不是加法。在这种情况下，如果 value 减去某个值得到的结果高于最小值（minimumValue），它就会作为结果返回。反之，则 minimumValue 会作为结果返回：

- 在增加某个值时，使用 Min 函数来设置上限；
- 在减少某个值时，使用 Max 函数来设置下限。

将这些概念综合应用，可以通过以下代码实现玩家的向前移动：

```
// 如果按住 W 或上箭头键：
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
{
    if (movementVelocity.z >= 0) // 如果原本在向前移动
        // 使用 VelocityGainPerSecond 增加 Z 轴速度，但最高不能超过 movespeed：
        movementVelocity.z = Mathf.Min(movespeed, movementVelocity.z + VelocityGainPerSecond
        * Time.deltaTime);

    else // 如果原本在向后移动
        // 使用 reverseMomentumMultiplier 增加 Z 轴速度，但不能超过 0：
        movementVelocity.z = Mathf.Min(0, movementVelocity.z + VelocityGainPerSecond *
        reverseMomentumMultiplier * Time.deltaTime);
}
```

这段代码乍一看可能有些复杂，但不用担心，下文会一步步地解析它的作用。请记住，当玩家向后移动时，velocity.z 的值将是负数，而向前移动时则是正数。由于摄像机位于玩家上方俯视着玩家角色，所以在屏幕上，“前进”的方向对应于向上，“后退”则对应于向下。

如果 velocity.z 的值等于 movespeed，则意味着玩家正在以最大的速度向前移动。相反，如果 velocity.z 的值等于 -movespeed（即负的 movespeed），那么玩家就以最大的速度向后移动。

如果把这段代码翻译成大白话，可以像下面这样表达：

- 如果玩家按住了 W 键或上箭头键；
- ……并且如果当前具有前进（正）动量，就增加前进动量，但要确保它不会超过 movespeed；
- ……否则，如果玩家当前具有后退（负）动量，就增加并使用 reverseMomentumMultiplier。由于当前的速度是负数，它不能超过 0。一旦速度达到 0，就停止使用 reverseMomentumMultiplier。

掌握这些概念后，本章后续的内容就很容易理解了。每当需要改变速度的时候，都将利用 Min 和 Max 来确保在赋值过程中将数值限制在一个合适的范围内。

在增加速度时，前面的代码使用一个之前声明的属性：VelocityGainPerSecond。在不涉及反方向移动时，它是像下面这样应用的：

```
movementVelocity.z + VelocityGainPerSecond * Time.deltaTime
```

这个过程很好理解，因为这个属性的名称相当直观地描述了它的作用：以当前的 Z 速度为基础，然后加上每秒的速度增量。由于频率是“每秒”，所以它必须乘以 Time.deltaTime，就像之前学过的那样。前面还提到过另一个知识点：乘法操作符 * 的优先级高于操作符 + 或 -。这意味着乘法操作符左右两侧的数字会首先相乘，然后再基于乘法的结果进行加法或减法运算。由此可以推断出，与 Time.deltaTime 相乘的是 VelocityGainPerSecond，而不是 movementVelocity.z 与 VelocityGainPerSecond 相加后的结果。

如果想让代码更加清晰明了，我们可以使用括号来分隔它们。有些人认为没有必要这么做，因为它执行的操作是相同的，但如果觉得这样有助于阅读和理解，就尽管这么做：

```
movementVelocity.z + (VelocityGainPerSecond * Time.deltaTime)
```

继续往下看，在向相反的方向移动时，代码是下面这样的：

```
movementVelocity.z + VelocityGainPerSecond * reverseMomentumMultiplier * Time.deltaTime
```

这与前面的代码相差无几，只不过其中添加了一个操作符“*”以应用 reverseMomentumMultiplier。我们知道，在乘法算式中，乘数的前后顺序并不重要。无论如何

调整后面三个引用的顺序，结果都将保持不变。

学习了这些概念后，你已经掌握了理解这个速度处理系统所需要的几乎所有知识。本章后面不会再涉及什么新知识了，最多也只是之前学习的知识的变体。

接下来要处理的是实现向后移动的代码。这段代码将紧跟在刚刚编写的 if 语句块之后（即检查 W 键或上箭头键是否被按下的那部分代码）：

```
// 如果按住 S 或下箭头键：
else if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
{
    // 如果原本在向前移动
    if (movementVelocity.z > 0)
        movementVelocity.z = Mathf.Max(0, movementVelocity.z - VelocityGainPerSecond *
        reverseMomentumMultiplier * Time.deltaTime);
    // 如果原本在向后移动或没有移动
    else
        movementVelocity.z = Mathf.Max(-movespeed, movementVelocity.z –
        VelocityGainPerSecond * Time.deltaTime);
}
```

这段代码与之前处理向前移动的很相似，只不过有一些小修改。首先，它使用关键字 else if 而不是 if 来检查玩家是否按下了 S 或下箭头键。这是为了确保在玩家同时按住上下移动键时，其中一个键（在这里是向上移动键）会优先生效，而不会两个键同时生效。

此外，由于这里是对速度进行减法操作，所以使用的是 Max 函数而不是 Min 函数。

再次把这段代码翻译成大白话：

- 如果玩家按住 S 键或下箭头键；
- ……并且玩家当前具有前进（正）动量，则在应用 reverseMomentumMultiplier 的同时减少前进动量，但要确保它不低于 0；
- ……否则，如果玩家当前具有后退（负）动量或根本没有动量，则减少动量，但要确保它不低于负的 movespeed。

现在只剩下一个关键条件需要处理：当玩家既不按前进键也不按后退键时，速度应该如何递减。如果不处理这个条件的话，玩家角色在松开移动键后仍然会持续移动。这显然不是我们想要的。

为了解决这个问题，需要在检查向后移动键是否被按下的 else if 语句块之后添加一个 else 语句块，只要前进或后退这两个键均未被按下，这个 else 语句块就会被执行。当前动量如果为正，它就会逐渐被减少到 0；当前动量如果为负，则会逐渐增加到 0；动量如果正好为 0，则不执行

任何操作：

```
else // 如果前进或后退键均未被按下
{
    // 必须逐渐将 Z 速度减至 0
    if (movementVelocity.z > 0)    // 如果原本在向前移动，
        // 将 Z 速度减去 VelocityLossPerSecond，但不能低于 0：
        movementVelocity.z = Mathf.Max(0, movementVelocity.z - VelocityLossPerSecond * Time.
        deltaTime);

    else // 否则，如果原本在向后移动，
        // 将 Z 速度加上 VelocityLossPerSecond，但不能高于 0：
        movementVelocity.z = Mathf.Min(0, movementVelocity.z + VelocityLossPerSecond * Time.
        deltaTime);
}
```

之后在实现左右移动的时候，直接应用类似的逻辑即可。比如，可以直接把处理前后移动的代码（即到目前为止在 Movement 方法中的全部内容）复制一遍，然后把键位改成 A/D 和左 / 右箭头。同时，还需要把所有对 Z 轴的引用都改成对 X 轴的引用，相应地修改即可。

在进行这些改动时一定要细心！如果忘记把某个地方的“.z”改成“.x”，游戏可能会出现一些意想不到的行为。向左 / 向右移动的实现代码如下：

```
// 如果玩家按住 D 键或右箭头键：
if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))
{
    // 如果原本向右移动
    if (movementVelocity.x >= 0)
        // 使用 VelocityGainPerSecond 增加 X 轴速度，但最高不能超过 movespeed：
        movementVelocity.x = Mathf.Min(movespeed, movementVelocity.x + VelocityGainPerSecond
        * Time.deltaTime);

    else // 如果原本向左移动，
        // 使用 reverseMomentumMultiplier 增加 X 轴速度，但最高不能超过 0：
        movementVelocity.x = Mathf.Min(0, movementVelocity.x + VelocityGainPerSecond *
        reverseMomentumMultiplier * Time.deltaTime);
}

// 如果玩家按住 A 键或左箭头键：
else if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
{
    // 如果原本在向右移动
```

```
        if (movementVelocity.x > 0)
            movementVelocity.x = Mathf.Max(0, movementVelocity.x - VelocityGainPerSecond *
            reverseMomentumMultiplier * Time.deltaTime);

        else // 如果原本在向左移动或没有移动
            movementVelocity.x = Mathf.Max(-movespeed, movementVelocity.x –
            VelocityGainPerSecond * Time.deltaTime);
    }
    else // 如果向左或向右键均未被按下
    {
        // 必须逐渐将 X 速度减至 0
        // 如果原本在向右移动
        if (movementVelocity.x > 0)
            // 将 X 速度减去 VelocityLossPerSecond，但不能低于 0：
            movementVelocity.x = Mathf.Max(0, movementVelocity.x - VelocityLossPerSecond * Time.
            deltaTime);
        // 否则，如果原本在向左移动
        else
            // 将 X 速度加上 VelocityLossPerSecond，但不能高于 0：
            movementVelocity.x = Mathf.Min(0, movementVelocity.x + VelocityLossPerSecond * Time.
            deltaTime);
    }
```

14.6 应用移动

现在只剩最后一步了——应用移动速度，让玩家能够真正地在游戏世界中移动。这将通过 Unity 内置的 Character Controller 组件来实现，后者主要用于为角色提供移动和碰撞检测。如果简单地通过设置玩家角色的 Transform 位置来进行移动，那么玩家角色将不会执行任何碰撞检测——它们会像幽灵一样穿过所有障碍物。而如果使用 Character Controller 组件来移动玩家角色，它们会与遇到的障碍物发生碰撞并沿其表面滑动——前提是这些障碍物带有碰撞体组件。在通过“游戏对象”菜单创建基本形状（比如立方体、球体等）时，Unity 会自动为它们配备与形状相匹配的碰撞体。

首先，让我们为 Player 游戏对象（带有 Player 脚本组件的那个对象）添加一个 Character Controller 组件。

在“层级”窗口中选择 Player 对象，然后单击“检查器”窗口底部的“添加组件”按钮，这将展开一个包含 Unity 所有组件的下拉菜单。在这个菜单中，可以通过依次单击 Physics 和

Character Controller 来添加 Character Controller 组件，也可以在下拉菜单中的搜索框里输入“character controller”，然后按下回车键直接添加这个组件，或 Character Controller 组件出现在搜索结果中时直接点击它。

Character Controller 组件采用的是胶囊碰撞体，这在“场景”窗口中看起来像是胶囊行形状绿色的线框，表示了玩家角色在当前 Character Controller 组件设置下的大小。

在“检查器”窗口中找到 Character Controller 组件的中心、半径和高度设置，并把中心改为（0，2.5，0），半径改为 1，高度改为是 5。这样的设置可以让碰撞体覆盖玩家模型的主要部分，并在模型两侧留出一点空间。其余的设置可以保持不变。修改完毕后，Character Controller 组件在“检查器”窗口中的设置应该与图 14-5 一致。

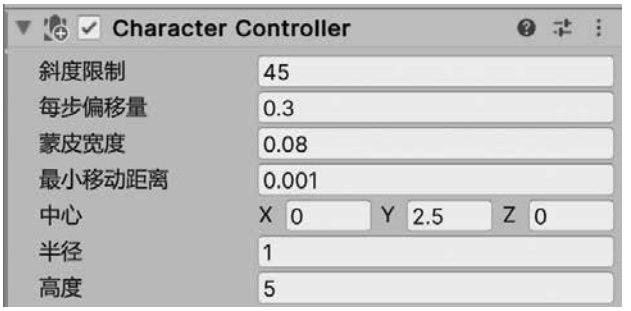

图 14-5 “检查器”窗口中显示 Player 游戏对象的 Character Controller 组件配置

设置好 Character Controller 组件之后，就可以设置之前在 Player 脚本中声明的对变量的引用。在 Unity 编辑器中，可以通过多种不同的方式来为组件设置引用。

- 在“检查器”窗口中找到位于 Player 脚本中“角色控制器”字段右侧的小圆圈图标，单击它之后，会弹出一个窗口，其中列出场景中所有带有 CharacterController 的游戏对象。双击其中的一个。
- 在“检查器”窗口中找到 CharacterController 组件标题栏中的粗体文本“Character Controller”，左键单击这个文本并将它拖放到 Player 脚本中的“角色控制器”字段上，以建立引用。
- 从“层级”窗口中将 Player 游戏对象拖放到“检查器”窗口中的变量字段上。Unity 识别出该字段需要一个 CharacterController 引用，并在拖过去的游戏对象上找到对应的组件。

采用任何一种方式都可以达成目的，但一些方式有时会更加便捷。

设置好字段后，它将显示拥有 CharacterController 组件的游戏对象的名称，右边跟着

（Character Controller）表示实际存储的组件类型。

同样，我们还需要设置 Trans 和 ModelTrans 的引用，这两个字段应该位于刚刚设置的 CharacterController 字段的上方。

操作方式与之前相同。可以在“检查器”窗口中将 Transform 组件的标题拖动到 Trans 字段上，然后从“层次”窗口中将 Model 游戏对象拖到“检查器”窗口中的相应字段上。

现在可以回到代码中，开始应用移动功能了。在 Movement 方法的底部，将以下代码添加到所有处理输入和移动速度的代码之后：

```
// 如果玩家在任一方向上移动（左 / 右或上 / 下）：
if (movementVelocity.x != 0 || movementVelocity.z != 0)
{
    // 应用移动速度：
    characterController.Move(movementVelocity * Time.deltaTime);
    // 使 Model Holder 面朝最后移动的方向：
    modelTrans.rotation = Quaternion.Slerp(modelTrans.rotation, Quaternion.
    LookRotation(movementVelocity), .18F);
}
```

这段代码的逻辑相当直观。首先通过一个 if 语句来检查 X 轴和 Z 轴是否有移动：如果 X 不为 0 或（|| 操作符）如果 Z 不为 0。如果是，就通过访问 characterController 引用并调用其 Move 方法来应用移动。这个方法接受一个 Vector3 类型的参数，表示这一帧需要移动的距离。这就是创建 movementVelocity 的意义。但需要注意的是，由于移动速度是按“每秒”计算的，所以需要将它乘以 Time.deltaTime。

之后，我们需要旋转 Model Holder，以使其面向移动方向。这涉及到一些之前没讲过的旋转操作，所以看起来可能有点陌生。

Unity 采用“四元数”（quaternion）来表示旋转。尽管四元数背后的数学原理相对复杂（听起来甚至有些吓人），但好消息是，我们并不需要掌握这些原理就可以在 Unity 中使用它。大多数情况下，只需要调用 Unity 的内置方法就可以实现旋转。唯一需要了解的是，四元数本质上代表一种旋转，它可以表示一个对象指向的方向或者对象倾斜的角度。在 Unity 的“检查器”窗口中，为了便于用户操作，旋转被表示为 Vector3 类型，并且每个分量的值范围都是 0 到 360 度。然而，在 Unity 的内部，旋转是通过四元数来表示和管理的。

前面的代码用到了两个方法：Quaternion.Slerp 和 Quaternion.LookRotation。

Slerp 是游戏开发中一个常用术语，尤其是在处理向量和旋转的时候。它是 spherical linear interpolation（球面线性插值）的缩写。简单来说，Slerp 是一个接受三个参数的方法：

- 开始时的旋转状态（一个四元数）；
- 目标旋转状态（也是一个四元数）；
- 一个浮点数，表示希望以多快的速度到达目标旋转状态。

这个浮点数是一个介于 0 和 1 之间的乘数，它基本上意味着“这一帧需要完成目标旋转的百分之多少？”所以如果它被设置为 0，对象将不会旋转；如果设置为 1，对象将立即旋转到目标状态；而如果设置为 0.18，就意味着每一帧将旋转目标旋转的 18%。

可以亲自体验一下，观察它是如何实现平滑的旋转效果的。在每一帧的更新中，我们都通过调用 Slerp 方法来设置 Model Holder 的旋转。Slerp 方法的每次调用都以当前的旋转状态作为起点，并以目标旋转作为终点。例如，第一帧将旋转至目标旋转的 18%，然后下一帧又会旋转剩余部分的 18%，依此类推。随着每一帧的更新，当前旋转状态与目标旋转状态之间的差距会逐渐变小。这产生了一种自然的弹簧效果（spring effect）。一开始时的旋转更快更明显，但随着越来越接近目标旋转状态，旋转速度会逐渐减缓至停止。这样做不仅使旋转看起来比每帧旋转固定角度更加流畅，还使得完成 180 度旋转的时间和 90 度旋转相同。

为了获得目标旋转（Slerp 调用中的第二个参数），我们调用 Quaternion.LookRotation 方法，它接受一个 Vector3 参数，并返回一个使对象朝向 Vector3 的方向的旋转（Quaternion）。

总的来说，这段代码将 movementVelocity 作为参数传递给 Quaternion.LookRotation 调用，以获取一个四元数，使对象朝向当前速度所对应的方向。然后，这段代码利用 Slerp 方法，使对象每一帧都向目标旋转状态旋转 18%。

如果不需要平滑的旋转效果，可以如下修改代码：

```
modelTrans.rotation = Quaternion.LookRotation(movementVelocity);
```

这种方法会立即改变玩家模型的朝向，使其与移动方向一致。虽然效果没有那么差，但会显得有点突兀，特别是从静止状态突然转向相反方向的时候。

至此，玩家角色的移动功能已经实现完成。它看上去很不错。现在，我们终于可以运行游戏并观察它的实际效果了。保存对 Player 脚本的更改，并确保已经在 Player 脚本的引用部分正确设置了相关变量。此外，最好在玩家角色周围放置一些立方体或球体，否则在空荡荡的游戏场景中可能很难判断玩家角色是否在移动。

玩家角色应该可以通过 WASD 键和箭头键移动。摄像机应该固定在玩家上方，随着玩家的移动而移动（但相对位置保持不变）。玩家模型将平滑地旋转，指向当前的移动方向。

14.7 小结

在这一章中，我们赋予了玩家移动的能力。本章介绍了如何通过创建和应用基本材质来为游戏对象添加颜色，并讲解了如何使用 Mathf.Max 方法和 Mathf.Min 方法在调整数值时限制它们的范围，此外，本章还说明了如何处理玩家角色每帧的速度变化，使玩家角色能够根据 WASD 键或方向键的输入逐渐加速或减速。在前面的章节中，我们在创建 Player 对象时富有先见之明地把玩家模型和 Player 的其他部分分离开。这种做法的好处在本章中得到了体现——无论玩家模型如何旋转，摄像机始终都保持在相同的位置。

本章要点回顾如下。

1. 属性（property）和变量很相似，但它明确地定义了获取或设置变量时应该执行的代码。
2. 在数值增加的操作中，使用 Min 函数可以限制数值不超过预设的最大值。
3. 在数值减少的操作中，使用 Max 函数可以确保数值不低于预设的最小值。
4. Slerp 可以在两个表示旋转的四元数之间平滑差值。
5. 四元数是一种用于表示旋转的数据类型。Transform.rotation 成员是一个四元数，表示 Transform 组件当前的朝向。
6. Quaternion.LookRotation 方法接受一个 Vector3 作为参数并返回一个四元数，后者代表如何旋转才能使 Transform 组件朝向该 Vector3 所指向的方向所需要的旋转。