游戏对象生命周期

第 5 章

CHAPTER 5

Unity 场景中创建的对象都称为游戏对象(GameObject),游戏对象有一个从创建到消亡的生命周期,本章介绍对象整个生命周期的相关类、属性和方法,总结了游戏对象实例化的几种方法及资源动态加载的实现方法。最后介绍了外部模型的导入。

本章学习要点:

- GameObject 类。
- 对象的创建和编辑。
- 预制件和对象的实例化。
- 资源动态加载。
- 对象的销毁。
- 外部模型导入。

5.1 游戏对象

Unity 中的游戏对象是构成虚拟世界的基本单元,它们充当组件的容器,而组件是实现具体功能的实体,每个游戏对象都有一个 Transform 组件。

5.1.1 游戏对象概述

游戏对象是 Unity 编辑器中最重要的概念。Unity 框架中 Project 由场景组成,场景由多个游戏对象组成。在游戏场景中被渲染能看到的对象都是游戏对象,如角色、道具、建筑、可收集资源、特效等。不会被渲染不出现在游戏场景中,但对游戏场景做出贡献的对象也是游戏对象,如摄像机、灯光等。

空游戏对象是一种最简单的游戏对象,只有一个 Transform 组件,其本身不能做任何事情,需要通过为它添加组件和属性,才能使其成为角色、环境或特殊效果等。要赋予游戏对象属性,使其成为一个立方体、一盏灯、一棵树或一个摄像机,需要为其添加组件。根据想要创建的对象类型,可以在游戏对象中添加不同的组件组合。Unity 有许多不同的内置组件类型,也可以使用 Unity 脚本 API 制作自定义组件。

5.1.2 游戏对象 Inspector 面板

游戏对象由多个组件组合而成,游戏对象就是各种组件的容器。这些组件决定了游戏对象的外观和功能,选择不同的组件就可以组合出不同的游戏对象。组成游戏对象的组件全部显示在该

游戏对象的 Inspector 面板中，以方便用户进行查看修改测试。另外，在 Inspector 各组件上方还有其他一些属性，如是否激活复选框、Name 名称、Tag 标签和 Layer 层级设置等，如图 5-1 所示。

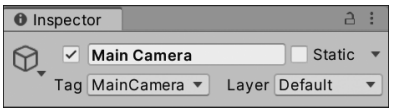

图 5-1　游戏对象的 Inspector 面板上方属性

5.1.3　GameObject 类和 gameObject 实例

1. GameObject 类

GameObject 类是所有游戏对象的父类。在脚本中，GameObject 类提供了一组方法，允许在代码中使用它们，包括查找、在游戏对象之间建立连接和发送消息等，添加或删除附加到游戏对象上的组件，以及设置与场景中它们的状态相关的值。一般来说，在 Inspector 面板中显示的属性，都能在脚本中获取和修改。GameObject 类有一些静态方法可以直接调用（表 5-1）。

表 5-1　GameObject 类常用静态方法

方　法	功　能
GameObject.CreatePrimitive()	创建一个系统自带的基本 3D 对象
GameObject.Find()	找到并返回一个名字为 name 的游戏对象
GameObject.FindWithTag()	返回一个用 tag 作标识的活动游戏对象
GameObject.FindGameObjectsWithTag()	返回一个用 tag 作标识的活动游戏对象数组
GameObject.FindObjectOfType<T>()	按指定类型（泛型 T，如 Camera）返回一个对象
GameObject.FindObjectsOfType<T>()	按指定类型（泛型 T）返回对象数组
GameObject.Instantiate()	实例化一个对象，继承 Object 的方法

2. gameObject 实例

脚本中的 gameObject 为挂载当前脚本的游戏对象实例，使用代码可以修改许多与游戏对象在场景中的状态相关的属性。当在场景中选择了一个游戏对象时，这些属性对应于 Inspector 面板顶部如图 5-1 中的属性。gameObject 常用成员变量和成员方法可以获取、修改和调用（表 5-2）。

表 5-2　gameObject 实例的成员属性和方法

属性和方法	功　能
gameObject.name	获取游戏对象的名字
gameObject.tag	获取游戏对象的标签
gameObject.activeSelf	获取游戏对象的激活状态
gameObject.activeInHierarchy	获取游戏对象的激活状态（受父对象的影响）
gameObject.SetActive()	设置游戏对象的激活状态
gameObject.GetComponent<ComponentType>()	获取该游戏对象上的指定类型组件
gameObject.AddComponent<ComponentType>()	为该游戏对象添加指定类型组件

5.2　创建游戏对象

可以在 Unity 编辑器中创建游戏对象，也可以通过代码动态创建游戏对象，并可以通过代码编辑游戏对象的属性。

5.2.1 创建基本 3D 对象

可以通过 GameObject 类的静态方法 CreatePrimitive()创建一个带有原始网格渲染器和适当碰撞器的基本 3D 游戏对象，创建出来的基本 3D 游戏对象默认位置在坐标原点(0,0,0)，默认名称与创建的基本 3D 游戏对象类型一致。CreatePrimitive()声明语句如下。

```
public static GameObject CreatePrimitive( PrimitiveType type);
```

CreatePrimitive()方法的参数为 PrimitiveType，PrimitiveType 是一个枚举类型，包含的基本 3D 对象类型如下。

```
PrimitiveType.Cube          //立方体
PrimitiveType.Sphere        //球体
PrimitiveType.Capsule       //胶囊体
PrimitiveType.Cylinder      //圆柱体
PrimitiveType.Plane         //平面
PrimitiveType.Quad          //正方形
```

如果项目没有在运行时引用以下组件 MeshFilter、MeshRenderer、BoxCollider 或 SphereCollider，CreatePrimitive()方法可能在运行时失败。避免这种情况的方法是声明这些组件类型的私有属性。系统将识别它们的用法，将它们包含在构建系统中，从而不会删除这些组件。一般情况下，CreatePrimitive()方法都可以直接使用，无须做特殊处理。

5.2.2 修改 3D 对象属性

通过 CreatePrimitive()方法生成 3D 对象后，可以设置该 3D 对象的各种属性，如名称(name)、标签(tag)、位置(position)等。要设置游戏对象的名称和位置，可以通过 name 属性和 position 属性，代码如下。

```
GameObject cube1 = GameObject.CreatePrimitive (PrimitiveType.Cube);
cube1.name = "立方体";
cube1.transform.position = new Vector3 (cube1.transform.position.x + 5, cube1.transform.
position.y,cube1.transform.position.z);
```


创建基本 3D 对象

【例 5-1】 创建基本 3D 游戏对象

(1) 通过 GameObject 类的 CreatePrimitive()方法创建几个基本 3D 对象，调整它们的位置，美化场景显示效果。

(2) 新建脚本，在 Start()方法中编写代码。

```
void Start()
{
    GameObject ob01 = GameObject.CreatePrimitive(PrimitiveType.Plane);
    ob01.transform.position = new Vector3(0, -1, 0);
    GameObject ob02 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    ob02.transform.position = Vector3.forward * 3;
    GameObject ob03 = GameObject.CreatePrimitive(PrimitiveType.Capsule);
    ob03.transform.position = Vector3.back * 3;
    GameObject ob04 = GameObject.CreatePrimitive(PrimitiveType.Cylinder);
    ob04.transform.position = Vector3.left * 3;
    GameObject ob05 = GameObject.CreatePrimitive(PrimitiveType.Cube);
    ob05.transform.position = Vector3.right * 3;
}
```

(3) 将脚本赋给主摄像机，运行效果如图 5-2 所示。

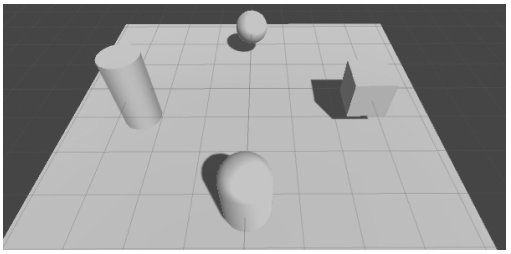

图 5-2　创建 3D 基本对象运行效果(Scene 视图)

【例 5-2】　创建 5×5 立方体墙体

创建 5×5 立方体墙体

(1) 创建一个 5 行 5 列的墙体,组成墙体的砖块是一个个立方体。

(2) 新建一个脚本,编写如下代码。变量 k 是立方体 name 属性的名称序号,变量 startPos 是每一行砖块的起始位置。通过 5×5 双重循环实现墙体的搭建,内层 j 循环实现每一行 5 块砖块的搭建,外层 i 循环实现 5 行(每一行已经搭建好)砖块的一层层的叠加搭建。因要留出砖块间的缝隙,每个砖块的大小等比缩放为原始大小的 95%。

```
int k = 0;                          //变量 k 是立方体 name 名称序号,初始值为 0
int startPos = -2;                  //变量 startPos 是每一行砖块的起始位置,共 5 块,因此初始值为-2
void Start()
{
    for (int i = 0; i < 5; i++)
    {
        startPos = -2;              //每一行都要把 startPos 初始值重置一下
        for (int j = 0; j < 5; j++)
        {
            GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                                                                    //生成一个 Cube
            cube.transform.localScale = new Vector3(0.95f, 0.95f, 0.95f);
                                                                    //将 Cube 等比缩小为 95%
            cube.transform.position = new Vector3(startPos++, i, 0);   //设置 Cube 的位置
            cube.name = "cube" + k++;                               //设置 Cube 的 name 属性
        }
    }
}
```

(3) 将脚本赋给主摄像机,运行效果如图 5-3 所示。

该种方法,组成墙体的每一个 Cube 立方体,都是一个独立的对象,系统要分别为它们分配资源,运行效率较低。

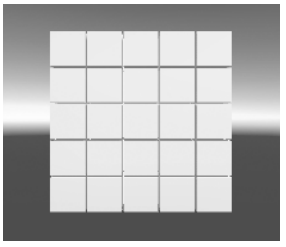

图 5-3　搭建的 5×5 墙体运行效果

5.3　预制件

Unity 预制件(Prefab)是可重用的游戏对象资源。预制件是 Unity 中一个非常重要的功能,用户可以创建、配置和存储游戏对象及其所有组件、属性值和子游戏对象等,作为一个可重用的资源。预制件可以看作游戏对象的模板,通过预制件可以轻松地对整个项目进行便捷地修改,提高开发效率。

5.3.1　预制件概述

Unity 中,Prefab 是存储在 Assets 中的一种资源,是优化系统资源和方便编辑复用大量类似对象的一种措施。也就是说,Unity 的预制件系统是创建、配置和存储游戏对象,实现一组相似对象的所有组件、属性值和子游戏对象编辑的可重用资源。

当想重用一个以特定方式配置的游戏对象时，如一个非玩家角色(NPC)、道具，应用于一个场景的多个地方或者应用于项目的多个场景中的物品、资源、子场景对象等，应该把该游戏对象(或者它们的组合)转换成一个预制件。这比简单地复制和粘贴要好，因为预制系统可以自动保持所有副本同步，方便后期的编辑修改，提高开发效率及质量。

Prefab 预制资源是一种资源类型，是存储在项目面板 Assets 文件夹中的一种可重复使用的游戏对象模板，可以在场景中创建基于该模板的实例。预制件可以多次放入多个场景中，当添加一个预制件到场景中，就创建了它的一个实例副本。所有的预制件实例链接到原始预制件，可以认为是原始预制件的克隆。不管项目中存在多少实例，当对 Assets 中的预制件进行任何更改，这些更改将自动应用于所有实例，而无须重复对预制件的每个实例副本进行相同的编辑。场景中创建的预制件资源实例对象在 Hierarchy 面板中以蓝色显示。为创建易于在多个级别上编辑的复杂对象层次结构，预制件可以嵌套。

5.3.2 创建预制件

1. 创建预制件的方法

创建预制件的方法有两种，一般创建的预制件放在 prefab 文件夹下。

第一种方法，从菜单选择 Assets|Create|Prefab，就创建了一个空预制件。可以编辑该空预制件，或者在 Hierarchy 面板中，选择想使之成为预制件的游戏对象，拖动该对象到项目面板 Assets 中的该空预制件上。

第二种方法，在 Hierarchy 面板中，将编辑好的原始对象直接拖到项目面板 Assets 中对应的预制件文件夹中(通常是 prefab 文件夹)，会自动创建一个预制件对象，原始对象成为预制件的一个实例。

第一种方法是先创建一个空预制件，再编辑或把场景中对象拖到空预制件上进行替换；第二种方法是以编辑好的对象为模板创建预制件。一般使用第二种方法，只需一步就创建好预制件了。

2. 预制件的编辑

创建的预制件如果比较复杂，如有多层游戏对象的嵌套，可以通过双击预制件，进入预制件编辑状态进行编辑，编辑完成后，单击 Hierarchy 面板上方左侧的小箭头，即可退出编辑状态。

5.3.3 原始预制件和预制件变体

然而并不是所有的预制件实例都必须是相同的。如果希望预制件的某些实例与其他实例不同，则可以创建 Original Prefab 覆盖单个预制件实例上的设置，还可以创建预制件的变体 Prefab Variants，如图 5-4 所示。

原始预制件(Original Prefab)和预制件变体(Prefab Variant)有什么不同呢？

假设场景中有一个通过预制件 A 创建的对象 A，通过对象 A 创建一个新的预制件时，有两种选择。

(1) 单击 Original Prefab 按钮创建一个原始预制件 B，则预制件 B 和预制件 A 断绝联系，它们之间的编辑互不影响。

(2) 单击 Prefab Variant 按钮创建一个预制件变体 A1，则预制件变体 A1 和预制件 A 类似，只不过预制件 A1 是预制件 A 的子类。当预制件 A 发生变化时，预制件变体 A1 会随着发生变化。但是当修改预制件 A1 时，预制件 A 不会发生变化。这和继承父类的子类一样，修改父类的公共属性时，子类也会随着变化，但是在子类做修改时，父类并不受影响。

通常应用中使用比较多的是创建原始预制件 B，和原来的预制件 A 完全脱离，成为一个全新的预制件，可以随意修改。当希望新建的预制件会跟随原来的预制件 A 变化而发生变化，则选择第二种预制件变体 A1，这样当创建多个预制件变体 A2，A3，…时，这些预制件变体就可以保持与预制件 A 的同步更新，但又可以保持自己独有的一些属性。这也可以看作多态的应用。

在项目开发后期，如果希望通过预制件 A 创建的对象，也拥有预制件 A1 的属性，从前面描述知道，对预制件 A1 的修改是不会作用于预制件 A 的。这时，可以使用覆盖功能，将预制件 A1 的属性覆盖应用于预制件 A，方法是选中预制件 A1，双击进入编辑窗口，在右侧的 Inspector 面板找到 Overrides 按钮，单击进行覆盖即可，如图 5-5 所示。

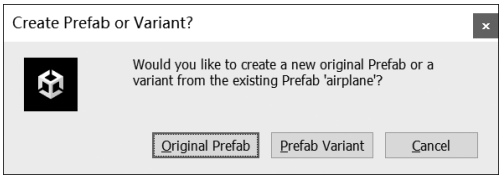

图 5-4　从场景中为游戏对象创建预制件对话框

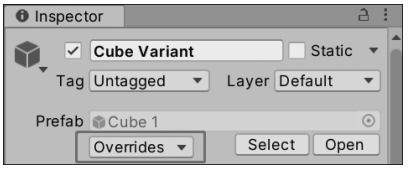

图 5-5　预制件变体的属性覆盖

5.4　实例化游戏对象

在 Unity 中，对象实例化主要指的是根据场景中的对象或预制件(Prefab)创建游戏对象的过程。实例化是面向对象编程中的一个概念，指的是用类来创建一个具体的对象实例。在 Unity 引擎中，对象实例化通常指使用场景中对象或预制件来生成场景中的游戏对象。

5.4.1　场景中对象的实例化

要将场景中的游戏对象进行实例化，首先要找到场景中的对象。

1. 对象查找

GameObject 的静态方法 Find()可以实现查找指定对象。Find()方法的声明语句如下，参数 name 为要查找对象的 name 属性，返回值为 GameObject 类型。

```
public static GameObject Find(string name)
```

2. 对象实例化

GameObject 继承自 Object 的静态方法 Instantiate()，作用是实例化参数指定的对象，即将参数传入的对象克隆一个出来。克隆出来的对象的位置、角度和大小与参数传入对象的位置、角度、大小一样。如果传入的不是场景中的对象(比如是 Assets 资源中的预制对象)，则克隆出来的对象位置在坐标原点。

```
public static Object Instantiate (Object original);
public static Object Instantiate (Object original, Transform parent);
public static Object Instantiate (Object original, Transform parent, bool
instantiateInWorldSpace);
public static Object Instantiate (Object original, Vector3 position, Quaternion rotation);
public static Object Instantiate (Object original, Vector3 position, Quaternion rotation,
Transform parent);
public static T Instantiate(T original);
public static T Instantiate (T original, Transform parent);
…
```

Instantiate()方法的参数如表 5-3 所示。

表 5-3 Instantiate()方法的参数

参　数	作　用
original	要实例化的现有对象
position	实例化对象的位置
rotation	实例化对象的方向
parent	实例化对象要指定给的父对象
instantiateInWorldSpace	如果实例化对象指定了父对象。该参数取值为 true,则实例化对象使用世界坐标系;取值为 false,则实例化对象使用父对象坐标系
T original	要克隆的类型为 T 的对象

传入的 original 可以是 GameObject 或者 Transform,也可以是包含特定组件的游戏对象,可以是场景中已有的对象,也可以是预制件。如果 original 是预制件,克隆出来的对象默认位置在坐标原点。

参数 original 是 GameObject 和 Transform 这两种类型,可以传入任意的游戏对象,因为所有的游戏对象都包含 Transform 组件,如以下代码。

original 是 GameObject 类型。

```
public GameObject obj;
void Start()
{
    GameObject.Instantiate(obj);
}
```

original 是 Transform 类型。

```
public Transform trans;
void Start()
{
    GameObject.Instantiate(trans);
}
```

参数 original 如果是具体的组件(如 Camera),则只能传入包含该组件的游戏对象。比如以下代码,需要传入 Camera 组件的游戏对象(如主摄像机或其他添加了 Camera 组件的对象)给变量 cam。

```
public Camera cam;
void Start()
{
    GameObject.Instantiate(cam);
}
```

3. 实例化对象类型转换

Instantiate()方法有返回值,可以赋给对象以便后续使用。当方法返回值和要赋值对象的类型不一致时,则要进行强制类型转换。这种情况使用泛型会更加方便,现在也基本使用泛型替代了原来的强制类型转换。在 Unity 中有很多使用泛型的情况。

```
public Camera cam;
void Start()
{
    Camera clone1 = (Camera)GameObject.Instantiate(cam);        //强制类型转换
    Camera clone2 = GameObject.Instantiate(cam) as Camera;      //使用 as 操作符实现类型转换
    Camera clone3 = GameObject.Instantiate<Camera>(cam);        //泛型
    Camera clone3 = GameObject.Instantiate(cam);                //泛型,直接使用
}
```

【例 5-3】　创建 5×5 立方体墙体改进 1


5×5 墙体改进 1～4

例 5-2 中构成墙体的砖块，是基本的 3D 游戏对象，如果想使用更复杂的砖块，可以通过以下方法实现。

（1）在场景中创建一个 Cube，新建一个绿色材质，将材质赋给 Cube。

（2）通过 GameObject 类的静态 Find()方法找到场景中的 Cube 对象，然后在双循环中实例化，修改例 5-2 的代码。

```
int k = 0;
int startPos = -2;
void Start()
{
    //在场景中找到游戏对象 Cube,并赋值给变量 cube
    GameObject cube = GameObject.Find("Cube");
    for (int i = 0; i < 5; i++)
    {
        startPos = -2;
        for (int j = 0; j < 5; j++)
        {
            //实例化 cube 的一个实例 obj
            GameObject obj = GameObject.Instantiate(cube);
            obj.transform.position = new Vector3(startPos++, i, 0);
            obj.transform.localScale = new Vector3(0.95f, 0.95f, 0.95f);
            obj.name = "Cube" + k++;
        }
    }
}
```

注意以上代码中，GameObject cube = GameObject.Find ("Cube")和 GameObject obj = GameObject.Instantiate(cube)可以合并为一行代码 GameObject obj = GameObject.Instantiate (GameObject.Find ("Cube"))。

（3）运行效果如图 5-6 所示。

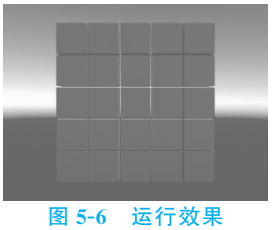
图 5-6　运行效果

5.4.2　预制件的实例化

生成 5×5 墙体改进 1 中，必须在场景中先创建一个对象 Cube，这个对象实际上是无用的，也不希望运行后在 Game 视图中显示出来。

1. 定义公有变量

在实际应用中，更多的是定义一个公有变量，暴露在 Inspector 面板中，然后将场景中 Cube 对象生成预制件，将预制件赋值给该公有变量。

2. Unity 中公有变量特性

Public 公有变量或全局变量，一般语言中都有，但是 Unity 脚本中的公有变量比较特殊。Unity 编辑器作为一个可视化交互式的编辑环境，有一个特性，就是脚本中所有的公有变量，会被显示在 Inspector 面板中。可以方便地为该公有变量赋值，或在运行时查看该变量值，甚至在运行时修改该变量值，交互式实时查看变量值修改对场景效果的影响。

【例 5-4】　创建 5×5 立方体墙体改进 2

（1）接例 5-3，在 Assets 文件夹下创建 prefab 文件夹，把场景中创建好的绿色 Cube 对象，拖到 prefab 文件夹里，从而创建一个预制件 Cube。然后把场景中的 Cube 对象删除。

（2）将例 5-3 的代码做如下改进，将语句 GameObject cube = GameObject.Find ("Cube");删除，声明一个 public 类型的变量 cube。

```
int k = 0;
int startPos;
public GameObject cube;                    //定义公有变量 cube,运行前该变量必须赋值
void Start()
{
    for (int i = 0; i < 5; i++)
    {
        startPos = -2;
        for (int j = 0; j < 5; j++)
        {
            //实例化公有变量 cube 的一个实例 obj
            GameObject obj = GameObject.Instantiate(cube);
            obj.transform.position = new Vector3(startPos++, i, 0);
            obj.transform.localScale = new Vector3(0.95f, 0.95f, 0.95f);
            obj.name = "Cube" + k++;
        }
    }
}
```

(3) 返回 Unity,将第一步做好的 prefab 文件夹中的 Cube 预制件,拖到 Inspector 面板中脚本下面变量 cube 后面的文本框中,如图 5-7 所示。实现为变量 cube 赋值。

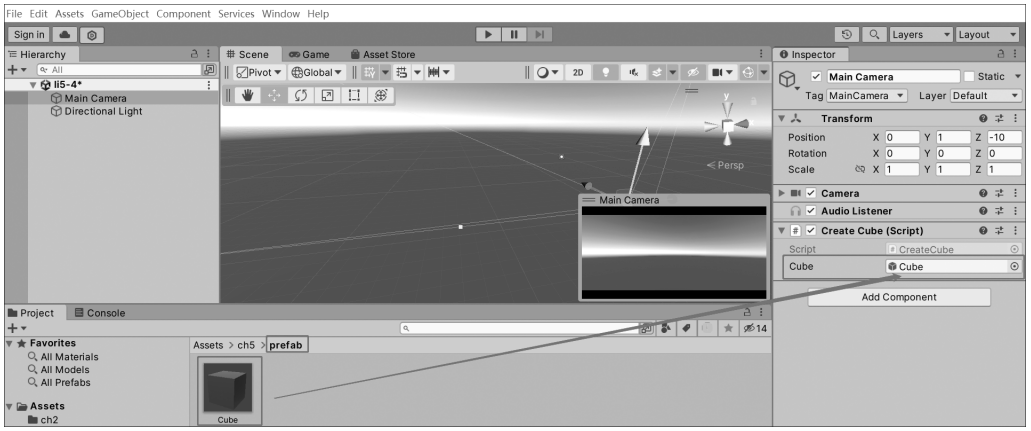

图 5-7 为变量 cube 赋值

(4) 运行程序,观察实现了与例 5-3 同样的效果。

5.4.3 私有变量的序列化

1. 序列化概念

序列化(Serialization)是将对象(变量)的状态信息转换为可以存储或传输的形式的过程。在序列化期间,对象将其当前状态写入临时或持久性存储区。以后,可以通过从存储区中读取或反序列化对象的状态,重新创建该对象。

序列化的主要作用是:①方便对象的存储,通常将对象以某种方式存储为文件,以后可以通过反序列化从文件中读取还原该对象;②方便对象的网络传输,将对象序列化为数据流,从发送端发送出去,经网络传输后,在接收端反序列化后还原该对象。

2. 私有变量的序列化

Unity 中显示在 Inspector 中的属性都同时具有序列化功能,序列化后的变量,读取时是有值的,不需要再次去赋值,因为它已经被保存下来了。public 公有变量默认是可以被 Serialize 的,所以 public 变量默认会在 Inspector 面板中显示,可以在编辑时为该变量赋值。Unity 会自动对 public 变量做序列化,但不对 private 变量做序列化。只有被序列化的变量才可以显示在

Inspector 面板上。因此一般情况下，Inspector 面板显示出的变量都为 public 变量。

但在大多数情况下，希望使用私有变量，以防止其他脚本修改该变量的值，增加安全性。但同时又想让该变量在 Inspector 面板中显示，方便交互式编辑。此时可以通过代码将该私有变量序列化。

3. SerializeField 实现私有变量的序列化

Unity 中可以通过添加 SerializeField 关键字实现私有变量的序列化。例如：

```
[SerializeField] private GameObject cube;
```

通过该语句，就可以在 Inspector 面板中显示私有变量 cube，并为该变量赋值了，同时私有变量 cube 也不会被其他脚本直接修改。

反过来，如果不想在 Inspector 窗口中显示某些 public 变量，可以使用 HideInInspector 关键字。例如：

```
[HideInInspector] public GameObject cube;
```

【例 5-5】　创建 5×5 立方体墙体改进 3

（1）将例 5-4 代码做如下改进，其他不用再做任何处理。

```
int k = 0;
int startPos;
[SerializeField] private GameObject cube;     //定义私有变量 cube,并序列化该变量
void Start()
{
    for (int i = 0; i < 5; i++)
    {
        startPos = -2;
        for (int j = 0; j < 5; j++)
        {
            GameObject obj = GameObject.Instantiate(cube);
            obj.transform.position = new Vector3(startPos++, i, 0);
            obj.transform.localScale = new Vector3(0.95f, 0.95f, 0.95f);
            obj.name = "Cube" + k++;
        }
    }
}
```

（2）运行，实现与例 5-4 同样的效果。

5.5　资源动态加载

5.5.1　资源动态加载概述

在 Unity 开发中，所有的游戏对象都直接创建在场景中，是不现实的，也是低效的，所以通常是在运行时动态创建游戏对象。也就是先创建好要用到游戏对象的预制件和其他资源文件，然后通过脚本实现资源运行时动态加载到场景中，从而创建出需要的场景环境、角色、特效等。

5.5.2　资源动态加载方法

资源动态加载有以下几种方法。

（1）将变量暴露在 Inspector 面板中，通过拖曳预制件进行赋值，如例 5-4 和例 5-5 使用的方法。这种方法在打包时，能够自动计算使用的资源，没有使用的资源不会出现在打包文件中。但拖曳赋值的方式确实比较烦琐。