【Unity基础教程】重点知识汇总

（十）

Unity C#设计模式之单例模式

单例模式（概念）

概念：单例模式（Singleton Pattern）是一种设计模式，保证一个类在整个应用程序的生命周期中**只有一个实例**，并提供一个**全局**访问点。它常用于管理游戏中的全局状态（如游戏管理器、音频管理器、数据管理器等），以便避免**重复实例化和复杂的依赖**关系。

在Unity中，单例模式非常常用，因为游戏开发中经常需要全局管理类，比如：

- GameManager: 管理游戏状态。
- AudioManager: 管理音效和背景音乐。
- UIManager: 管理 UI 的显示和隐藏。

***注意：**单例模式本质上是一种**通用**的设计模式，属于软件工程中的**创建型**设计模式之一。它并非特定于Unity或C#。它的定义和使用在许多编程语言和框架中都适用，包括Java、C++、Python等，而不是某个特定平台或语言的专属设计。

单例模式（核心特性及实现方式）

核心特性：

- **唯一性：**一个类只能有一个实例。
- **全局访问：**可以通过静态方法或属性访问到该实例。
- **延迟初始化：**实例在第一次访问时创建。

实现方式：

在Unity中，单例模式的实现可以分为两种主要情况：

- **普通类**单例：不继承MonoBehaviour，适用于不需要挂载到GameObject上的管理类。
- **MonoBehaviour**单例：继承自MonoBehaviour，适用于需要附加到GameObject上的类。

普通类单例（实际应用）

普通类单例主要用于与Unity的**生命周期**无关的类。

```
public class MySingleton
{
    private static MySingleton _instance;

    // 私有构造函数，防止外部实例化
    private MySingleton() { }

    // 公共静态属性，返回唯一实例
    public static MySingleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new MySingleton();
            }
            return _instance;
        }
    }

    // 示例方法
    public void DoSomething()
    {
        Debug.Log("MySingleton is working!");
    }
}
```

```
public class GameManager : MonoBehaviour
{
    void Start()
    {
        // 调用普通单例MySingleton中的DoSomething方法
        MySingleton.Instance.DoSomething();
    }
}
```

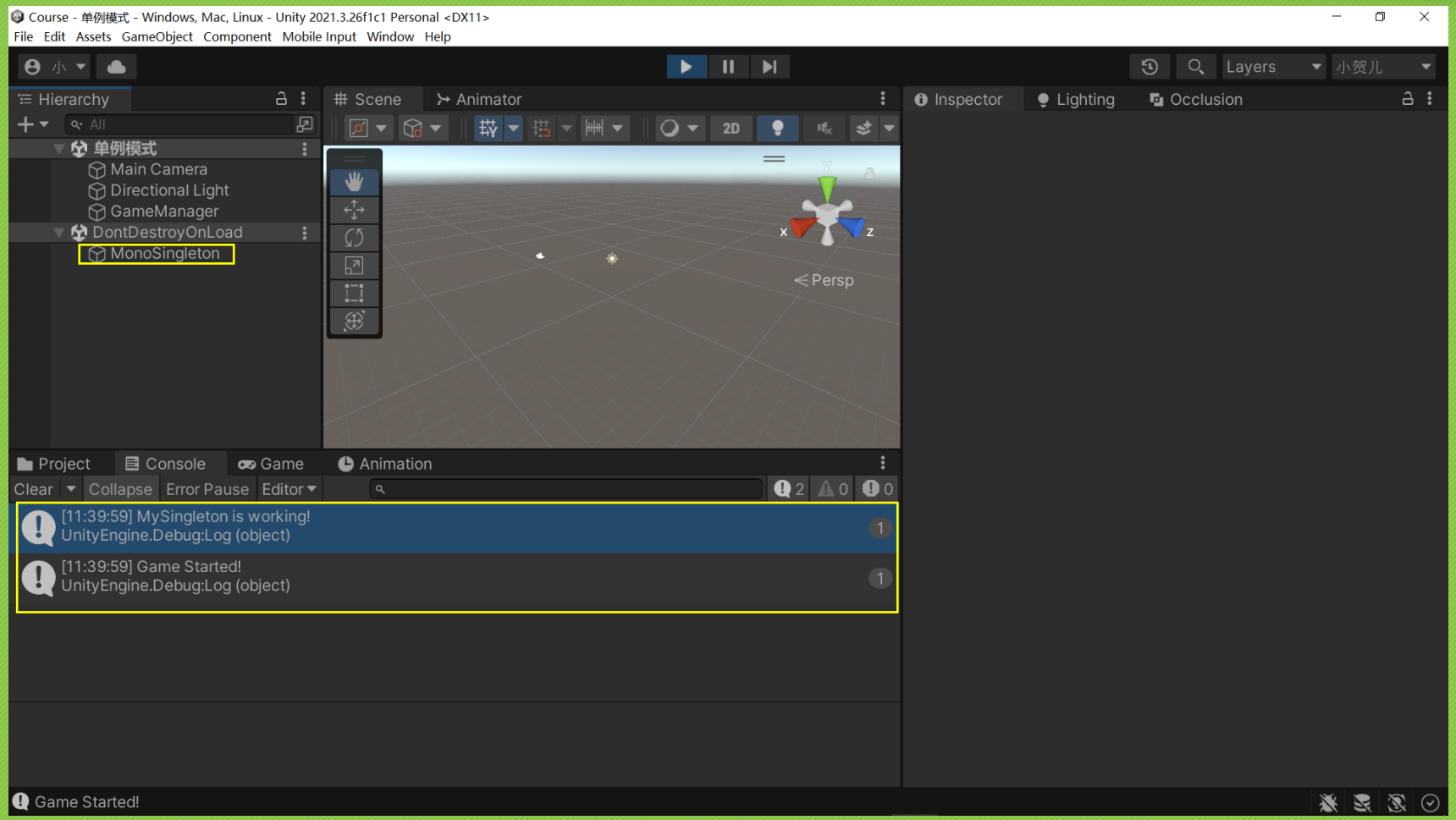
MonoBehaviour单例（实际应用）

在Unity中，大部分单例类是继承自MonoBehaviour的，因为它们需要利用Unity的**生命周期**和**功能**。

```
public class MonoSingleton : MonoBehaviour
{
    // 静态实例
    public static MonoSingleton Instance { get; private set; }
    // Unity 的 Awake 方法
    private void Awake()
    {
        // 如果实例已经存在，并且不是当前对象，则销毁当前对象
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }
        // 设置为单例实例
        Instance = this;
        // 如果需要跨场景保持实例，启用以下代码
        DontDestroyOnLoad(gameObject);
    }
    // 示例方法
    public void StartGame()
    {
        Debug.Log("Game Started!");
    }
}
```

```
public class GameManager : MonoBehaviour
{
    void Start()
    {
        // 调用普通单例MySingleton中的DoSomething方法
        MySingleton.Instance.DoSomething();
        // 调用MonoBehaviour单例中的StartGame方法
        MonoSingleton.Instance.StartGame();
    }
}
```

两种单例（运行结果）


Course - 单例模式 - Windows, Mac, Linux - Unity 2021.3.26f1c1 Personal <DX11>
File Edit Assets GameObject Component Mobile Input Window Help
Layers
Hierarchy
Scene
Animator
Inspector
Lighting
Occlusion
单例模式
Main Camera
Directional Light
GameManager
DontDestroyOnLoad
MonoSingleton
Persp
Project
Console
Game
Animation
Clear
Collapse
Error Pause
Editor
[11:39:59] MySingleton is working!
UnityEngine.Debug:Log (object)
[11:39:59] Game Started!
UnityEngine.Debug:Log (object)
Game Started!

使用单例模式（注意事项）

- **线程安全：**对于普通单例模式，如果涉及**多线程**，可以使用线程锁（**lock**）保证线程安全。但在Unity中，单线程（**主线程**）是主流，这个问题通常可以忽略。
- **场景切换问题：**MonoBehaviour单例应使用**DontDestroyOnLoad**，以防止对象在场景切换时被销毁。
- **生命周期管理：**确保单例的生命周期合适。如果单例对象在场景中没有被销毁，但不再需要，可能会**浪费内存**。（在Awake方法中，确保销毁重复的实例）
- **避免滥用：**单例模式虽然方便，但可能导致代码**耦合性**过高。如果一个类职责过多，考虑是否应拆分成多个类。

*普通类单例（线程锁）

在普通类单例中，如果涉及多线程环境，可以使用**lock**关键字保证线程安全。

```
public class ThreadSafeSingleton
{
    // 单例实例，用于保存唯一的对象
    private static ThreadSafeSingleton _instance;
    // 锁对象，用于确保多线程访问时的线程安全
    private static readonly object _lock = new object();
    // 私有构造函数，防止通过外部代码直接实例化该类
    private ThreadSafeSingleton() { }
    // 单例访问点，获取唯一实例
    public static ThreadSafeSingleton Instance
    {
        get
        {
            // 使用锁来确保线程安全
            lock (_lock)
            {
                // 如果实例尚未创建，则创建一个新的实例
                if (_instance == null)
                {
                    _instance = new ThreadSafeSingleton();
                }
                // 返回唯一实例
                return _instance;
            }
        }
    }
}
```

【Unity基础教程】重点知识汇总

（十）

Unity C#设计模式之单例模式