【Unity基础教程】重点知识汇总

（十三）

Unity C#设计模式之对象池模式

对象池模式（概念）

概念：对象池（Object Pool）是一种**优化内存和性能的设计模式**，通过**复用**一组已经存在的对象，**减少频繁的实例化和销毁操作**对内存和性能的影响。在游戏开发中，频繁创建和销毁对象（例如子弹、特效、敌人等）会导致性能问题，因为这些操作会消耗CPU时间，并触发垃圾回收（GC），从而导致性能下降或卡顿。对象池通过在游戏中复用对象，避免了这些问题。

注意：对象池并不是Unity独有的概念，它是**通用**的设计模式，广泛应用于各种编程语言和框架中。C#中可以通过编程实现对象池的逻辑，而Unity的场景中，结合GameObject的生命周期管理，对象池的应用尤为普遍。（通过禁用和启用对象**（SetActive(false)** 和**SetActive(true)）**，可以避免销毁和重新实例化）

对象池模式（核心思想及优势）

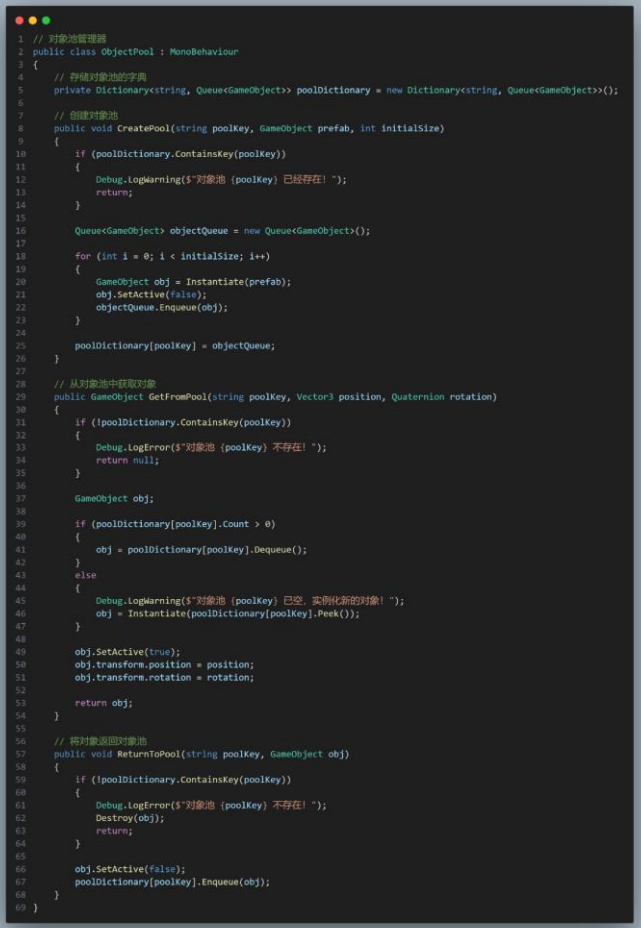
核心思想：

- 创建一个用于存储可复用对象的池（**列表或队列**）。
- 从池中获取对象时，检查是否有**闲置**的对象：
  - 如果有，直接使用。
  - 如果没有，创建一个新对象并加入池中。
- 回收对象时，将其返回池中，使其可以被复用。

优势：

- **性能优化：**通过重用对象，减少了内存分配和垃圾回收的开销。
- **减少延迟：**避免了在关键时刻创建新对象的延迟。
- **更好的资源管理：**可以控制对象的数量，避免资源浪费。

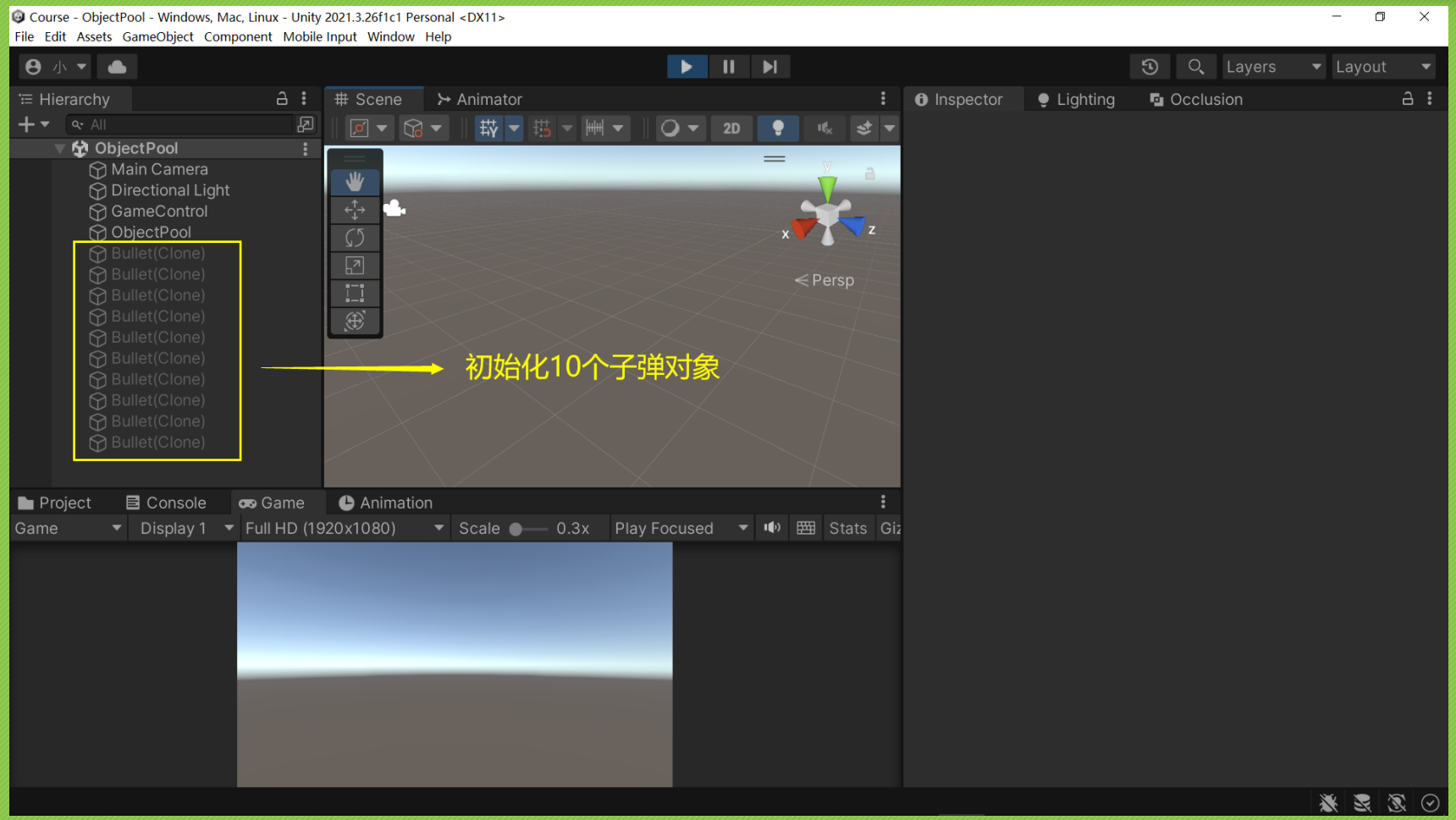
对象池模式（具体实现）



```csharp
// 对象池管理器
public class ObjectPool : MonoBehaviour
{
    // 存储对象池的字典
    private Dictionary<string, Queue<GameObject>> poolDictionary = new Dictionary<string, Queue<GameObject>>();

    // 创建对象池
    public void CreatePool(string poolKey, GameObject prefab, int initialSize)
    {
        if (poolDictionary.ContainsKey(poolKey))
        {
            Debug.LogWarning($"对象池 {poolKey} 已经存在！");
            return;
        }

        Queue<GameObject> objectQueue = new Queue<GameObject>();

        for (int i = 0; i < initialSize; i++)
        {
            GameObject obj = Instantiate(prefab);
            obj.SetActive(false);
            objectQueue.Enqueue(obj);
        }

        poolDictionary[poolKey] = objectQueue;
    }

    // 从对象池中获取对象
    public GameObject GetFromPool(string poolKey, Vector3 position, Quaternion rotation)
    {
        if (!poolDictionary.ContainsKey(poolKey))
        {
            Debug.LogError($"对象池 {poolKey} 不存在！");
            return null;
        }

        GameObject obj;

        if (poolDictionary[poolKey].Count > 0)
        {
            obj = poolDictionary[poolKey].Dequeue();
        }
        else
        {
            Debug.LogWarning($"对象池 {poolKey} 已空，实例化新的对象！");
            obj = Instantiate(poolDictionary[poolKey].Peek());
        }

        obj.SetActive(true);
        obj.transform.position = position;
        obj.transform.rotation = rotation;

        return obj;
    }

    // 将对象返回对象池
    public void ReturnToPool(string poolKey, GameObject obj)
    {
        if (!poolDictionary.ContainsKey(poolKey))
        {
            Debug.LogError($"对象池 {poolKey} 不存在！");
            Destroy(obj);
            return;
        }

        obj.SetActive(false);
        poolDictionary[poolKey].Enqueue(obj);
    }
}
```

```csharp
// 初始化对象池
public class GameControl : MonoBehaviour
{
    public ObjectPool objectPool;
    public GameObject bulletPrefab;

    void Start()
    {
        // 初始化一个名为 "Bullets" 的对象池，初始大小为10
        objectPool.CreatePool("Bullets", bulletPrefab, 10);
    }
}
```

对象池模式（运行结果）



操作指引：

1.在Unity中创建一个Bullet的Prefab。

2.在场景中分别添加一个空物体，并挂载上面的ObjectPool和GameControl脚本。

对象池模式（具体实现2）

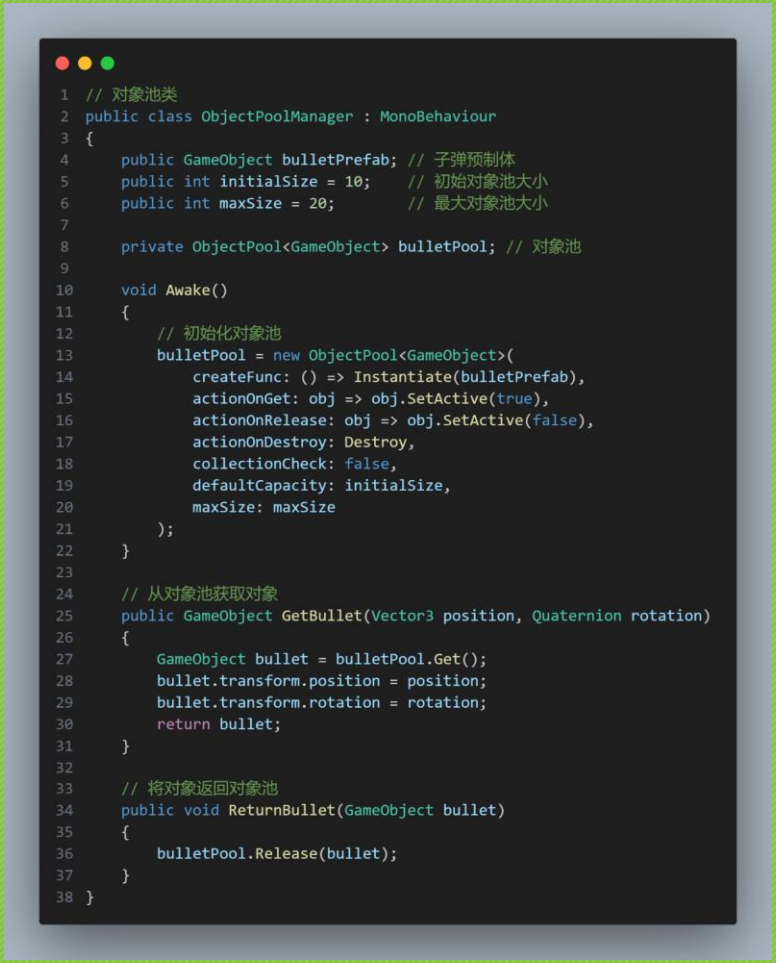
```
public class BulletController : MonoBehaviour
{
    public ObjectPool objectPool;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            // 从对象池获取子弹
            GameObject bullet = objectPool.GetFromPool("Bullets", transform.position, Quaternion.identity);

            // 给子弹添加简单的移动逻辑
            bullet.GetComponent<Rigidbody>().velocity = transform.forward * 10f;

            // 返回对象池
            StartCoroutine(ReturnBulletToPool(bullet, 2f));
        }
    }

    private IEnumerator ReturnBulletToPool(GameObject bullet, float delay)
    {
        // 延迟2秒后将对象返回对象池
        yield return new WaitForSeconds(delay);
        objectPool.ReturnToPool("Bullets", bullet);
    }
}
```

操作指引：

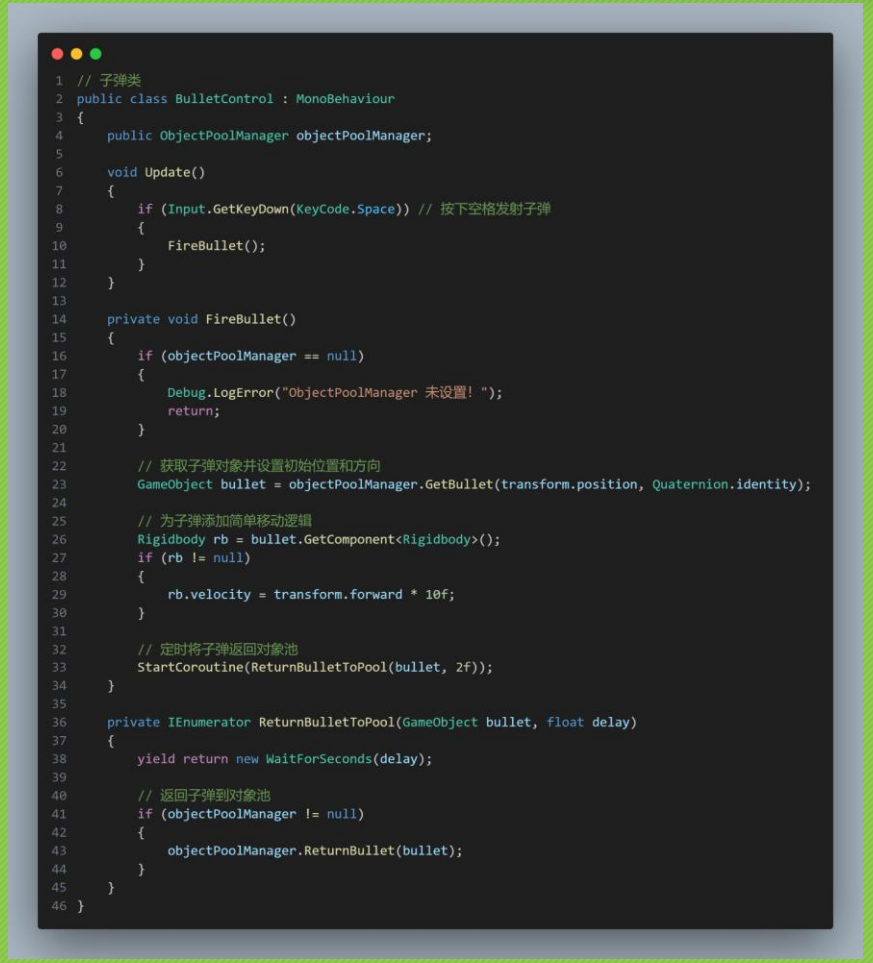
1.在Bullet预制体上添加一个Rigidbody组件。

2.挂载脚本后，播放运行，按下键盘空格键发射子弹。

*使用Unity官方对象池类（脚本重构）



```csharp
// 对象池类
public class ObjectPoolManager : MonoBehaviour
{
    public GameObject bulletPrefab; // 子弹预制体
    public int initialSize = 10;    // 初始对象池大小
    public int maxSize = 20;        // 最大对象池大小

    private ObjectPool<GameObject> bulletPool; // 对象池

    void Awake()
    {
        // 初始化对象池
        bulletPool = new ObjectPool<GameObject>(
            createFunc: () => Instantiate(bulletPrefab),
            actionOnGet: obj => obj.SetActive(true),
            actionOnRelease: obj => obj.SetActive(false),
            actionOnDestroy: Destroy,
            collectionCheck: false,
            defaultCapacity: initialSize,
            maxSize: maxSize
        );
    }

    // 从对象池获取对象
    public GameObject GetBullet(Vector3 position, Quaternion rotation)
    {
        GameObject bullet = bulletPool.Get();
        bullet.transform.position = position;
        bullet.transform.rotation = rotation;
        return bullet;
    }

    // 将对象返回对象池
    public void ReturnBullet(GameObject bullet)
    {
        bulletPool.Release(bullet);
    }
}
```



```csharp
// 子弹类
public class BulletControl : MonoBehaviour
{
    public ObjectPoolManager objectPoolManager;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space)) // 按下空格发射子弹
        {
            FireBullet();
        }
    }

    private void FireBullet()
    {
        if (objectPoolManager == null)
        {
            Debug.LogError("ObjectPoolManager 未设置! ");
            return;
        }

        // 获取子弹对象并设置初始位置和方向
        GameObject bullet = objectPoolManager.GetBullet(transform.position, Quaternion.identity);

        // 为子弹添加简单移动逻辑
        Rigidbody rb = bullet.GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.velocity = transform.forward * 10f;
        }

        // 定时将子弹返回对象池
        StartCoroutine(ReturnBulletToPool(bullet, 2f));
    }

    private IEnumerator ReturnBulletToPool(GameObject bullet, float delay)
    {
        yield return new WaitForSeconds(delay);

        // 返回子弹到对象池
        if (objectPoolManager != null)
        {
            objectPoolManager.ReturnBullet(bullet);
        }
    }
}
```

补充说明：

- Unity在**2021.2**版本中引入了ObjectPool<T>，用于实现高效的对象池。代码更**简洁**，且内置优化避免了手动管理队列的复杂性。
- 提供createFunc、actionOnGet、actionOnRelease等**回调**，便于扩展功能，例如重置对象状态。
- 使用**预分配**的策略，支持最大容量限制，有助于避免资源泄漏或超出限制的实例化。

*自定义对象池与Unity官方对象池（区别）

特性	Unity 内置对象池（`ObjectPool<T>`）	自定义对象池
易用性	提供统一的接口和事件回调，开发者只需提供初始化、回收等逻辑即可。	完全由开发者手动实现，需自行管理对象的创建、重置和销毁。
扩展性	支持高度自定义（通过回调方法）来适应不同的对象需求，如初始化或状态重置。	需要为每种对象池单独编写代码，扩展和维护成本较高。
性能	内置优化，避免不必要的队列操作和集合检查，支持最大容量限制和动态扩展。	性能取决于开发者的实现，可能会出现集合操作低效、队列满时实例化过多等问题。
内存管理	动态控制最大容量，避免内存泄漏或无限制增长的风险。	如果未正确实现，可能导致内存泄漏或对象过度实例化。
代码复杂性	简化的代码结构，通过回调完成复杂功能，逻辑清晰。	需要手动编写和维护复杂逻辑，代码量较多且易出错。
兼容性	与 Unity 的其他功能（如 DOTS、ECS 等）集成良好。	独立实现，通常与 Unity 的高级功能不兼容。

选择建议：

- **简单需求**：如果项目需要的对象池逻辑较为简单（如子弹或简单特效），通用（自定义）对象池的实现可能已经足够。
- **复杂需求**：如果需要高效、稳定、易扩展的解决方案（如资源密集型场景或大规模对象管理），推荐使用Unity官方对象池类。

使用对象池的优点及注意事项（总结）

优点：

- **性能优化：** 避免频繁的**Instantiate（克隆）**和**Destroy（销毁）**，减少内存分配和垃圾回收。
- **可控性：** 可以精确管理对象的生命周期。
- **灵活性：** 适用于各种需要频繁创建和销毁的场景。

注意事项：

- **初始池大小：** 初始池的大小需要根据实际情况调整，太小可能导致频繁扩展池，太大则会浪费内存。
- **对象重用逻辑：** 每次获取和归还对象时，记得重置对象的状态（如位置、旋转等）。
- **线程安全：** 如果在多线程环境中使用对象池，需要处理线程安全问题。

【Unity基础教程】重点知识汇总

（十三）

Unity C#设计模式之对象池模式