**Unity3d中对象池(ObjectPool)的实现**

**概述**

**什么是对象池？**

　　池(Pool)，与集合在某种意义上有些相似。 水池，是一定数量的水的集合；内存池，是一定数量的已经分配好的内存的集合；线程池，是一定数量的已经创建好的线程的集合。那么，对象池，顾名思义就是一定数量的已经创建好的对象(Object)的集合[1]。

　　在C/C++的程序中，如果一种对象，你要经常用malloc/free(或new/delete)来创建、销毁，这样子一方面开销会比较大，另一方面会产生很多内存碎片，程序跑的时间一长，性能就会下降。这个时候，就产生了对象池。可以事先创建好一批对象，放在一个集合中，以后每当程序需要新的对象时候，都从对象池里获取，程序用完该对象后，再把该对象归还给对象池。这样，就会少了很多的malloc/free(new/delete)的调用，在一定程度上提高了系统的性能，尤其在动态内存分配比较频繁的程序中效果较为明显。

**Unity3d中的对象池**

　　在使用unity3d做游戏时，经常有同一个Prefab用到多次，需要反复实例化(Instantiate)。但实例化是很消耗资源的，所以在游戏加载时把一批Prefab实例化好放在对象池中，游戏中用的时候拿出来，不用的时候放回去，避免反复申请和销毁。

　　存入对象池的元素应具有如下特征：1>场景中大量使用　2>存在一定的生命周期，会较为频繁的申请和释放。

**关于多线程的考虑**

　　因为Unity的API只能被主线程调用，我理解Unity提供的用户空间是单线程的（脚本中写While（true）挂在GameObject上，点运行整个Unity会卡死）。所以我们不需要将池实现支持多线程。在支持多线程的应用中，单例的初始化通常要加一个锁，在这里也没有必要。

**希望实现具有以下特征的对象池**

1、新增对象种类时操作简单，能够灵活控制每个对象池中生成的对象数量。

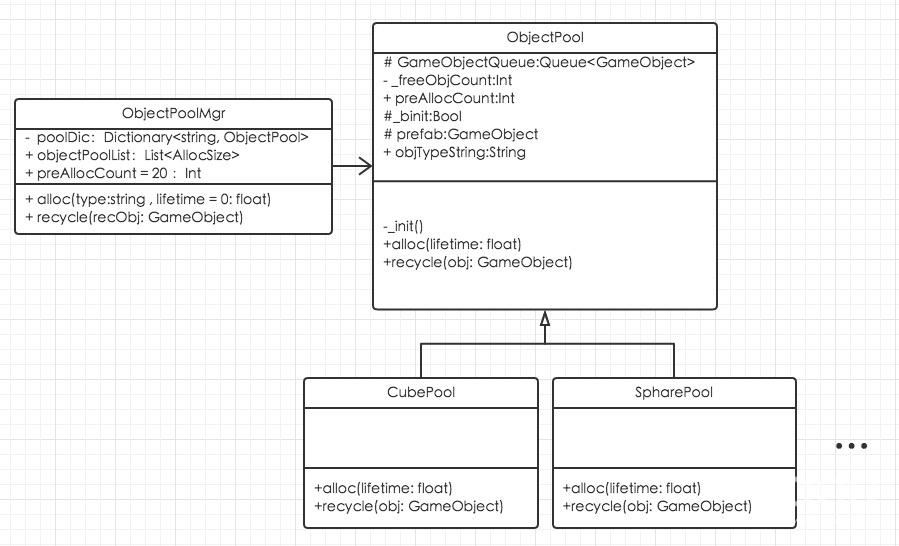
2、接口简单，易于申请和回收。

3、模块结构清晰，耦合度低。

4、申请和回收时，可以根据具体的对象类型做个性化操作。

**实现**

**1、模块设计**



　　\*UML规则参考自 UML 基础: 类图

　　模块中主要使用以下类

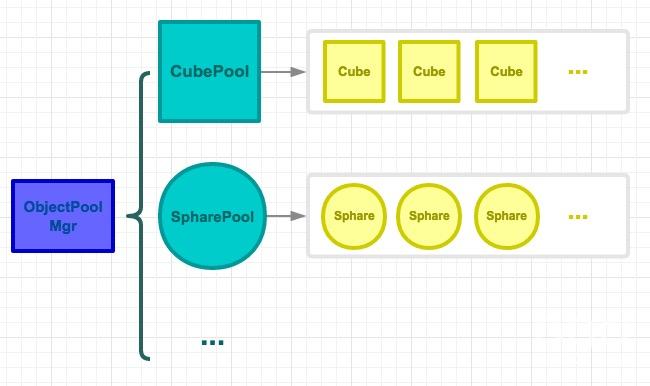
　　ObjectPoolMgr是对象池管理类。它对外提供接口，对内管理对象池实例。外部进行申请和释放操作都是通过调用ObjectPoolMgr中的接口进行的。

　　ObjectPool是对象池基类，它是抽象类，包含了通用的成员和方法。作为一个基类，它提可被子类继承的方法是virtual的，便于实现多态。

　　CubePool, SpharePool是具体的对象池实现类，继承自ObjectPool，可以override基类的alloc和recycle等方法实现个性化的操作。

　　PrefabInfo挂在对象池中的每一个对象上，用来记录对象类型等信息。

**2、数据管理方式**



　　数据管理方式如图，单例ObjectPoolMgr中使用字典管理着多种对象池实例，每种对象池实例中使用队列管理一定数量的同类型对象。

**3、主要接口和调用流程**

　　对象池管理类ObjectPoolMgr类中提供对外接口，主要是alloc和recycle，声明如下

|  |  |
| --- | --- |
| 1  2 | //回收接口。参数是待回收GameObject  public static void recycle(GameObject recycleObj)； |

　　游戏在使用申请到的GameObject时可能会在其中添加子物体，回收前会判断一下recycleObj中是否嵌套有其它属于对象池的Prefab，如果存在就分别进行回收。

　　另外如果对象池中待分配对象数量超过了用户设置的个数，直接销毁recycleObj而不再放回对象池。

**4、对象池的创建时机**

　　过早创建对象池，池中的对象会占用大量内存。若等到游戏使用对象时再创建对象池，可能因为大量实例化造成掉帧。所以，我认为在Loading界面创建下一个场景需使用的对象池是较为合理的。比如天天飞车中的NPC车，金币，赛道，在进入单局比赛后才用到。可以在进入比赛的Loading界面预先创建金币，NPC车，赛道的对象池，比赛中直接申请使用。

**5、具体实现**

1>ObjectPoolMgr

ObjectPoolMgr是对象池的管理类，提供接口，

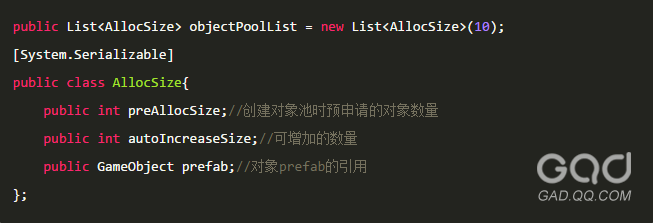
|  |  |
| --- | --- |
| 1  2 | public static GameObject alloc(string type, float lifetime = 0);  public static void recycle(GameObject recycleObj); |

参数lifeTime是存活时间，以秒为单位，定义如下  
lifeTime > 0 lifeTime秒后自动回收对象。  
lifeTime = 0 不自动回收对象，需游戏主动调用recycle回收。  
lifeTime < 0 创建Pool实例并实例化Pool中的对象，但不返回对象，返回值null。  
当lifeTime>0时，分配出去的GameObject上挂的PrefabInfo脚本会执行倒计时协程，计时器为0时调用recycle方法回收自己。它的适用对象如射击游戏中的子弹，申请时设定了lifeTime后不必关心回收的问题，当然游戏可以计时器在到时前主动发起回收。  
lifeTime < 0的目的预创建对象池，在游戏场景Loading时可以用这个方法先把对象池创建起来，避免游戏中创建对象池造成掉帧。

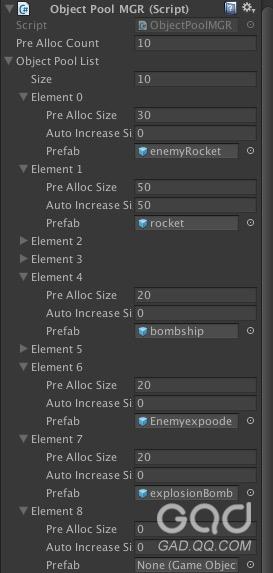
ObjectPoolMgr用成员poolDic维护已分配的对象池实例

|  |  |
| --- | --- |
| 1 | private Dictionary<string, ObjectPool> poolDic = new Dictionary<string, ObjectPool>(); |

使用objectPoolList记录面板上的用户设置



　　ObjectPoolMgr初始化时会在Unity的层次（Hierarchy）面板中创建GameObject并添加自身脚本。开发者可以在Inspector面板中直接创建新的对象池,如下图。Pre Alloc Size是对象池创建时预申请的对象数量。Auto Increase Size是池中的对象被申请完后进行一定数量的自增。prefab对象池关联的预制类型



比如，当游戏需要申请Cube对象时，会调用GameObject cube = ObjectPoolMGR.alloc("Cube");方法。ObjectPoolMGR.alloc代码如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static GameObject alloc(string type, float lifetime = 0){      //根据传入type取出或创建对应类型对象池     ObjectPool subPool = Instance.\_getpool(type);     //从对象池中取一个对象返回     GameObject returnObj = subPool.alloc(lifetime);     return returnObj;  } |

　　ObjectPoolMgr会根据传入的类型type，调用\_getpool(type)找到对应的Pool，再从其中取一个对象返回。  
　　代码中\_getpool(string type)是ObjectPoolMgr中的私有方法。前面说过，ObjectPoolMgr有一个成员poolDic用来记录已创建的对象池实例，\_getpool方法先去poolDic中查找，找到直接返回。如果找不到说明还未创建，使用反射创建对象池，记录入poolDic，代码如下



　使用\_getpool获得了对象池实例后，会调用对象池的alloc方法分配一个对象。

2>对象池基类ObjectPool

其中记录了对象池的通用方法和成员，如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | protected Queue queue = new Queue();//用来保存池中对象  [SerializeField]  protected int \_freeObjCount = 0;//池中待分配对象数量  public int preAllocCount;//初始化时预分配对象数量  public int autoIncreaseCount;//池中可增加对象数量  protected bool \_binit = false;//是否初始化  [HideInInspector]  public GameObject prefab;//prefab引用  [HideInInspector]  public string objTypeString;//池中对象描述字符串 |

ObjectPool中的方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | public virtual GameObject alloc(float lifetime){      //如果没有进行过初始化，先初始化创建池中的对象      if(!\_binit){          \_init();          \_binit = true;      }      if(lifetime<0){          Debug.LogWarning("lifetime <= 0, return null");          return null;//lifetime<0时，创建对象池并返回null      }      GameObject returnObj;      if(\_freeObjCount > 0){//池中有待分配对象          returnObj = queue.Dequeue();//分配          \_freeObjCount--;      }else{//池中没有对象了，实例化一个          returnObj = Instantiate(prefab , new Vector3(0,0,0), Quaternion.identity) as GameObject;          returnObj.SetActive(false);//防止挂在returnObj上的脚本自动开始执行          returnObj.transform.parent = this.transform;      }      //使用PrefabInfo脚本保存returnObj的一些信息      PrefabInfo info = returnObj.GetComponent《PrefabInfo 》();      if(info == null){          info =  returnObj.AddComponent《PrefabInfo 》();      }      if(lifetime > 0){          info.lifetime = lifetime;      }      info.types = objTypeString;      returnObj.SetActive(true);      return returnObj;  }    public virtual void recycle(GameObject obj){      //待分配对象已经在对象池中      if(queue.Contains(obj)){          Debug.LogWarning("the obj " + obj.name + " be recycle twice!" );          return;      }      if( \_freeObjCount > preAllocCount + autoIncreaseCount ){          Destroy(obj);//当前池中object数量已满，直接销毁      }else{          queue.Enqueue(obj);//入队，并进行reset          obj.transform.parent = this.transform;          obj.SetActive(false);          \_freeObjCount++;      }  } |

　　这里要注意的是，基类alloc和recycle方法要使用虚函数，子类override实现多态。

3>对象池子类CubePool  
　　子类override父类的alloc和recycle，进行个性化的申请和回收工作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class CubePool : ObjectPool {      public override GameObject alloc(float lifetime){          GameObject cubeObject= base.alloc(lifetime);          //在这里进行CubePool个性化的的初始化工作          return cubeObject;      }  } |

当然也可以直接复用基类的alloc方法，甚至不写CubePool类。当ObjectPoolMgr申请一个Cube但找不到CubePool类时，会使用通用方法进行分配和回收。

4>PrefabInfo  
　　PrefabInfo是挂在prefab实例上，用来记录prefab类型和lifetime等数据。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class PrefabInfo : MonoBehaviour {      public string types;      [HideInInspector]      public float lifetime = 0;      void OnEnable(){          if(lifetime > 0){              StartCoroutine(countTime(lifetime));          }      }      IEnumerator countTime(float lifetime){          yield return new WaitForSeconds(lifetime);          ObjectPoolMGR.recycle(gameObject);      }  } |

**新增Pool方法**

　　为一个新Perfab创建对象池需要以下两步，  
1、在unity面板中把prefab挂上，并设置prefab的实例化数量和可增加数量  
2、(可选)实现一个对应的Pool脚本。如果不实现，这个对象池会使用通用的申请和回收方法。

**调用申请和释放方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //申请  GameObject obj1 = ObjectPoolMGR.alloc("Cube",5);  GameObject obj2 = ObjectPoolMGR.alloc("Sphare");    //回收  ObjectPoolMGR.recycle(obj2); |

**总结**

**存在的问题**

　　对于从对象池中申请的GameObject，目前在游戏使用中不能改变其层次结构，不能添加新的Component，也不能在其中新增非对象池分配的GameObject。因为这些改变在回收时无法被发现，再次复用时可能出现意想不到的结果。

　　介于这种情况，我的使用方法是如果使用过程中改变了对象结构，用完后就Destroy，不再recycle。如果改变是可预期的，也可以重写子类recycle进行处理。

**待扩展**

　　当游戏收到内存告警时，应该可以释放对象池，增加可用内存。释放的策略可以有多种：

1、释放ObjectPoolMgr中所有的对象池(能释放大量内存,但Destroy会造成CPU消耗,下次申请时还要重新创建对象池)

2、压缩对象池中的对象数量(用户在使用对象池时设置了池中GameObject的基本数量和可增加数量，可以把增加的释放掉)

3、释放一些不常用的对象池和其中的对象(不常用的定义可以有很多，比如被申请的次数最少，最久未被使用等)

4、释放指定一种或几种对象池

　　等等

　　同样的，游戏申请GameObject而对象池中可申请数量为空，就需要扩展对象池。扩展对象池的策略可以有：

1>实例化两个GameObject，一个返回给游戏，一个放入池中以备下次申请

2>按照用户设置的AutoIncreaseCount，每次池为空时实例化相应数量的对象

…

　　空间申请和释放策略可以有多种，可以组合使用，但没有万全之策，可以根据游戏的特点去实现。

实现时踩的一个小坑

　　对象池基类ObjectPool的Start()和Update()方法最好不要使用，因为创建的子类会自动生成这两个方法，一不小心就覆盖了。

**对比测试**

　　我把对象池用于自己制作的小游戏SpaceBattle，做了一个简单测试。场景中有很多敌人，陨石，子弹（如图），我把这三种prefab放入对象池。



**不使用对象池：**



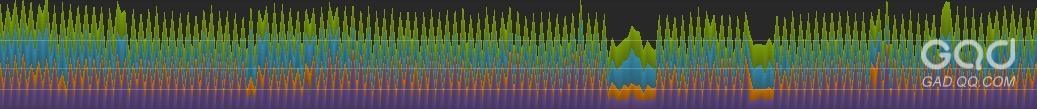
场景中有1843个GameObject，cpu使用呈震荡上涨趋势。因为不断有销毁和实例化，GameObject数量抖动上涨，Total GCAlloc抖动剧烈，有较频繁的内存回收。帧数降到了30左右。

**使用对象池：**



　　场景中有2291个GameObject，较上一个场景稍复杂一些。cpu抖动较上面平缓，基本保持在60帧到30帧之间，播放特效时帧数降低（特效未使用对象池）。最后能保持在60帧左右。随着Total GameObject数量缓步上涨，Total GCAlloc曲线平滑，说明内存操作不频繁，可以达到节省系统资源的效果。

　　可以看出，对象池对降低系统资源消耗是有作用的。在不使用对象池的测试中也遇到了一些极端情况：游戏频繁实例化和销毁对象时cpu剧烈抖动，这种情况应该尽力避免。



      另外也要合理设置对象池中的预分配对象数量。过多会占用大量内存，过少效果不好。

注：代码和文章都是年初写的，最近才整理发出来。时间久了可能会有记忆模糊或的地方，如果有疏漏或是错误还望大家不吝指出，谢谢