# Unity3d中对象池(ObjectPool)的概念和用法

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。**

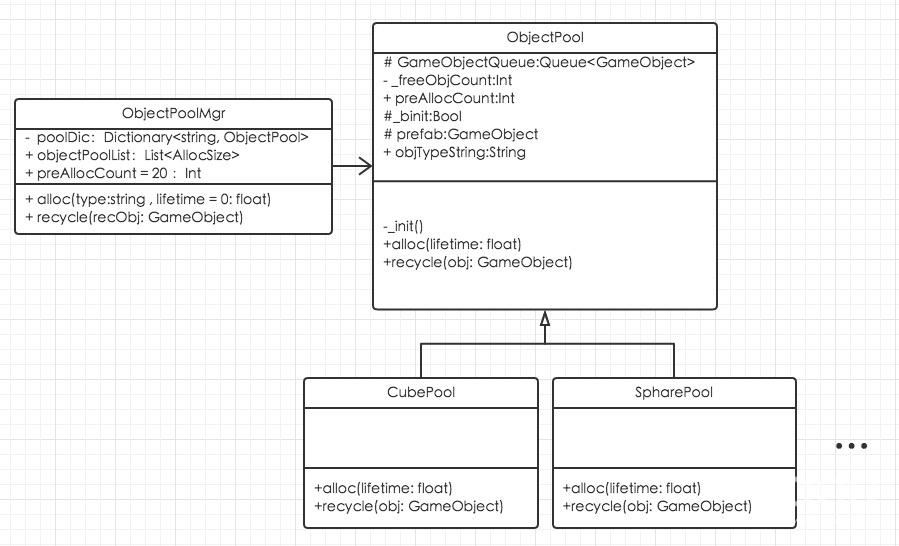
**2、接口简单，易于申请和回收。**

**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

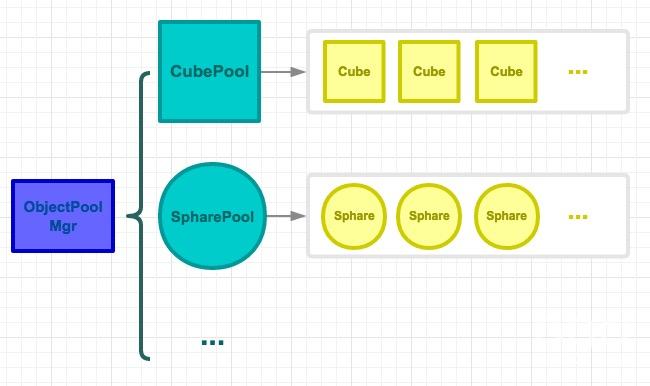
ObjectPoolMgr是对象池管理类。它对外提供接口，对内管理对象池实例。外部进行申请和释放操作都是通过调用ObjectPoolMgr中的接口进行的。

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



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，



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。

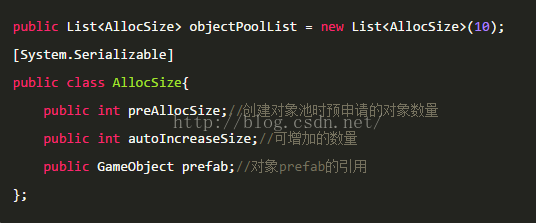
lifeTime < 0 创建Pool实例并实例化Pool中的对象，但不返回对象，返回值null。

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。

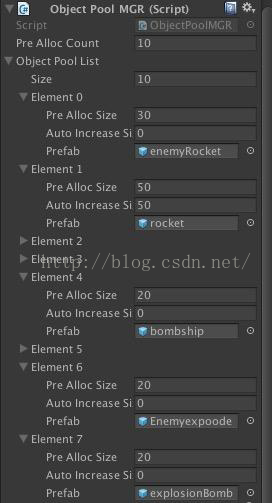
lifeTime < 0的目的预创建对象池，在游戏场景Loading时可以用这个方法先把对象池创建起来，避免游戏中创建对象池造成掉帧。



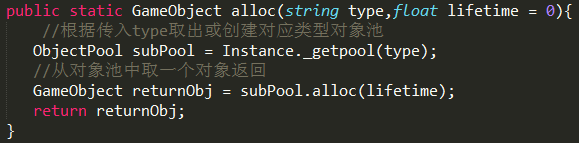
使用objectPoolList记录面板上的用户设置



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



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。

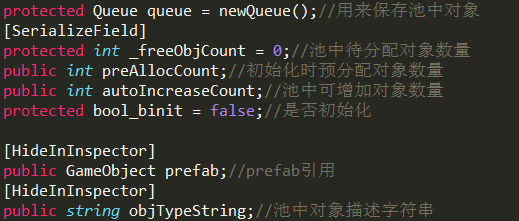
代码中\_getpool(string type)是ObjectPoolMgr中的私有方法。前面说过，ObjectPoolMgr有一个成员poolDic用来记录已创建的对象池实例，\_getpool方法先去poolDic中查找，找到直接返回。如果找不到说明还未创建，使用反射创建对象池，记录入poolDic，代码如下

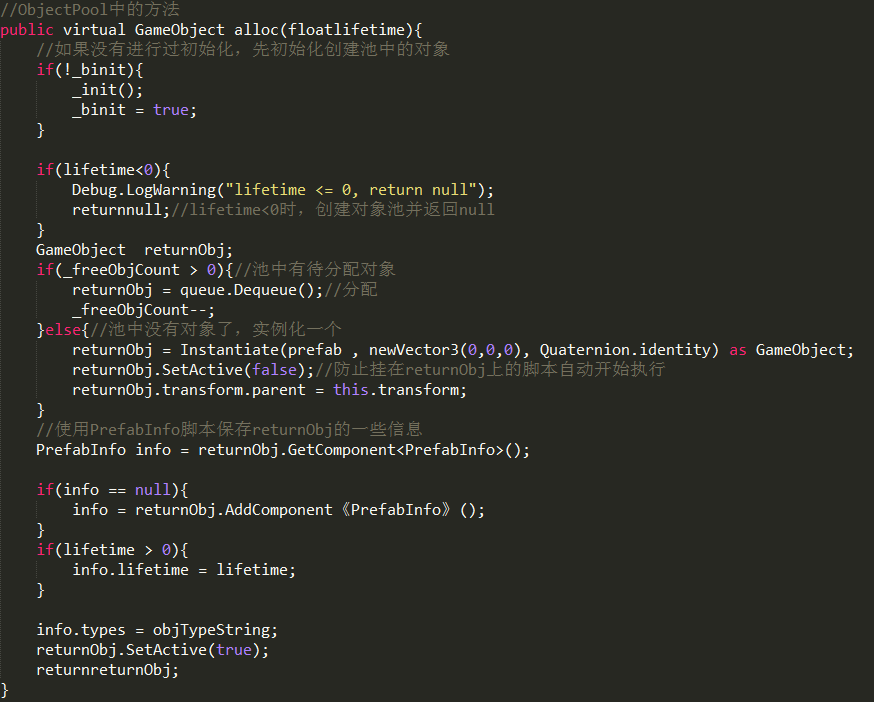


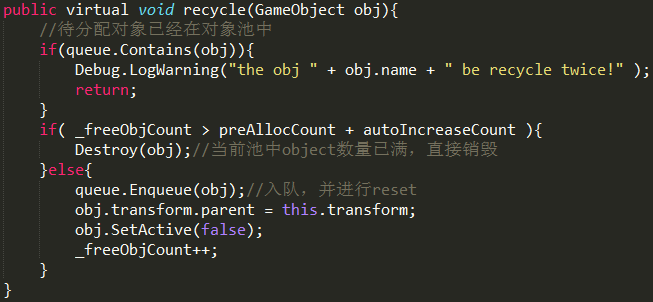
使用\_getpool获得了对象池实例后，会调用对象池的alloc方法分配一个对象。

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



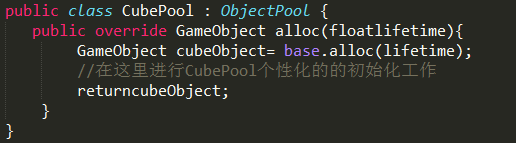




这里要注意的是，基类alloc和recycle方法要使用虚函数，子类override实现多态。

3>对象池子类CubePool

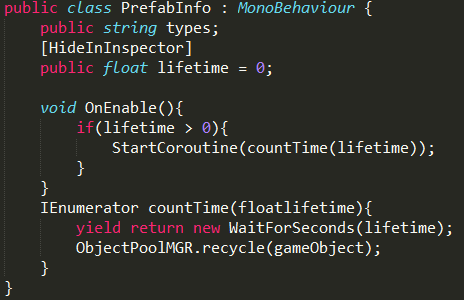
子类override父类的alloc和recycle，进行个性化的申请和回收工作。



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。



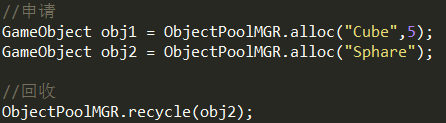
新增Pool方法

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



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：



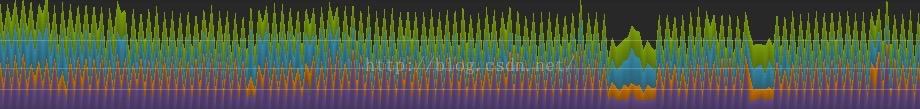
场景中有1843个GameObject，cpu使用呈震荡上涨趋势。因为不断有销毁和实例化，GameObject数量抖动上涨，Total GCAlloc抖动剧烈，有较频繁的内存回收。帧数降到了30左右。

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。

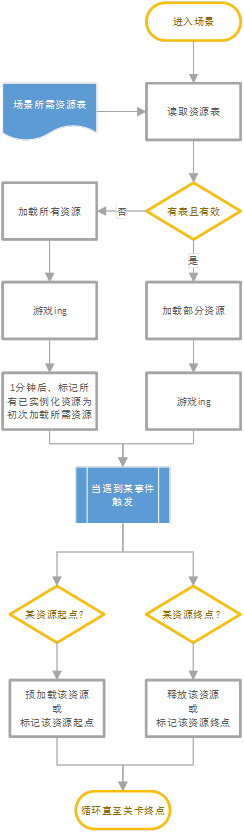
# Unity资源池的动态加载释放和内存优化处理

需求环境

在上一级的【解决方案】文章中，我们设计出了动态加载资源的业务流程，而这一节，我们就通过一些简单的代码，来实现出业务流程中的效果。

吸取之前文章的经验，如果按照正式项目的规格开发，本篇文章就会非常冗余，所以我们优化一下，仅仅针对技术点进行讲解与释放，具体与工程相关的，我们就不再文章中讲解，但你可以在Github的工程中找到它们。、

现在，我们先回顾一下之前所设计出的业务流程。



那么，在这个业务流程中，我可以定义出在游戏运行时，资源有三种状态：

**1、未加载**

**2、已经加载**

**3、已可以释放**

三种状态了某个资源此时的最佳使用环境，也就是说，接下来需要使用的资源，我就放到池中，而接下来很长一段时间内不需要使用的资源，我就彻底释放掉。以确保程序的内存总是在可控范围之内。

设计

为了达到这样的目的，我们就需要划分三个模块去做。

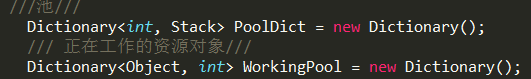
**1、最基础的资源加载，与池。**

**2、资源加载的自动记录过程。**

**3、资源加载的动态释放与加载过程。**

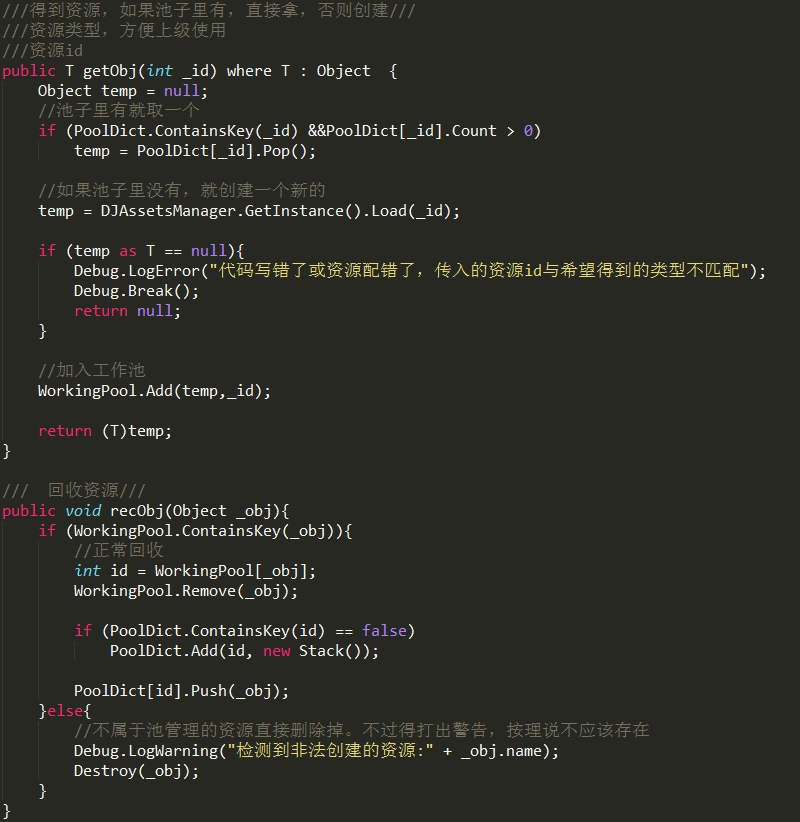
池

首先，池，因为我们是模拟，所以这个就比较容易实现，在现实工程中，则可能需要考虑不同资源类型的具体逻辑。



首先是2个定义，一个是回收池，一个是工作区，工作区用来反向查资源的ID，同时，也检测是否有资源是通过其他方法加载的，理论上，游戏内不应该存在其他的途径来加载资源。

接下来，就是2份逻辑代码，一个是创建资源，它用到了之前我们实现的资源管理器，另一个是回收资源。

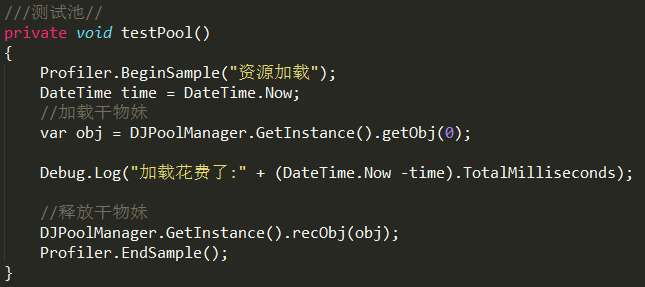


要注意的是，池仅仅负责资源的状态转换，并没有处理资源的开关，与销往逻辑，具体工程中可以根据资源类型分类编写，也可以给资源挂在统一的逻辑脚本去处理自己的销毁回调。

还有另一种方法，则是在使用池子进行资源销毁之前，自己动手对资源进行回收相关的处理，这样更依赖于人，不推荐团队使用，但此时我们做范例，就不额外引入更多的业务逻辑

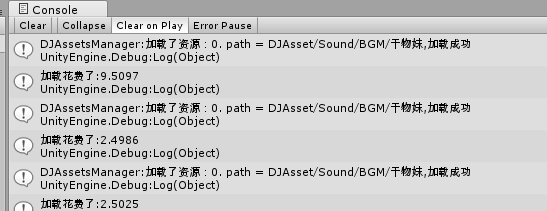
池测试

现在池已经弄好了，我们就需要简单的做一个池子的小测试。打开项目工程10-2PooL场景，我们能找到Test对象，它身上有脚本PoolTest.Cs 。当游戏运行时，我们就可以通过它去检查池子是否生效。



使用之后这个函数测试之后，我们可以发现，第一次加载花费了9毫秒，而第二次，则只用了2毫秒。

具体的花费，我们也需要通过性能分析器去查看，使用 Profiler.BeginSample("资源加载")进行标记，这里就不在额外扩展。



PS: 在文章代码中并没有对预制体进行管理，这其实是不好的，最好手动的控制他们的加载与释放。

资源生命周期的自动记录

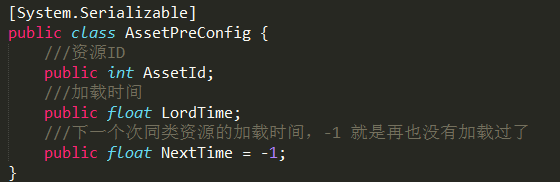
要记录资源的生命周期，首先我们得确定自己的游戏形势，如果是大世界类型的游戏，我们需要根据区域范围来确定资源表，那么如果是副本类型的，我们就需要以副本为单位记录一份资源表。

并且，有的资源我们希望是动态加载的，而有的资源，比如主角的特效，模型，音频等等，我们更希望它们是常驻的。所以，我们还需要区分一份资源是否需要动态加载。

表

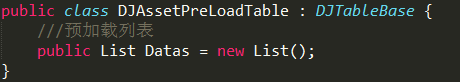
知道了需求后，我们就可以对自动记录表进行设计。为了讲解清晰，我尽量的保持任何一个元素都只是为了测试，不与业务逻辑挂钩。

在工程中，你可以到之前我们创建过的DJAssetsDefine 命名空间，里面我们新添加了这一次需要使用到的记录表。



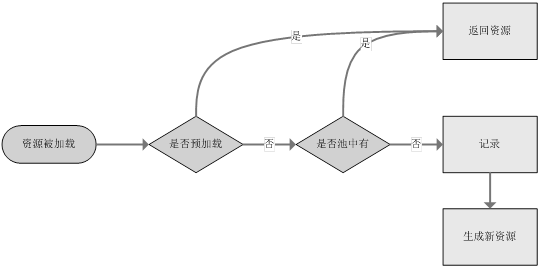
字段很简单，也有注释说明，大家看注释就好。

之后我们要让它成为一张表，所以需要再创建一个文件。在工程里可以找到名为：DJAssetPreLoadTable.cs 的代码文件。只有一个List，我打算直接使用List的索引来表示资源加载的前后关系，所以就不需要其他信息了。

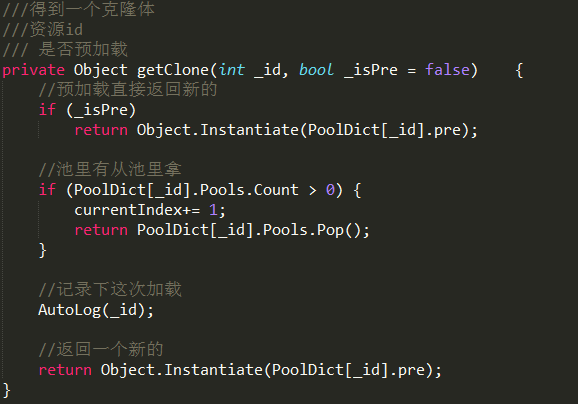


自动记录

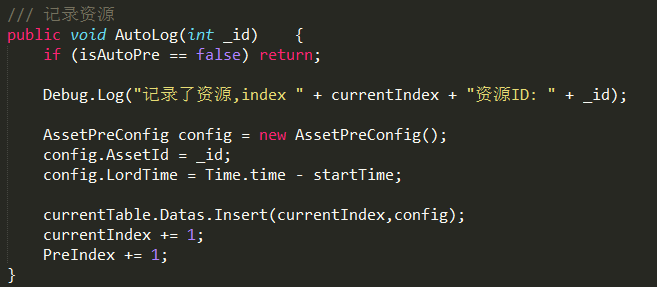
有了表以后，我们就可以在游戏运行时，把被加载的资源记录到表中。这里面包含了一个逻辑过程。



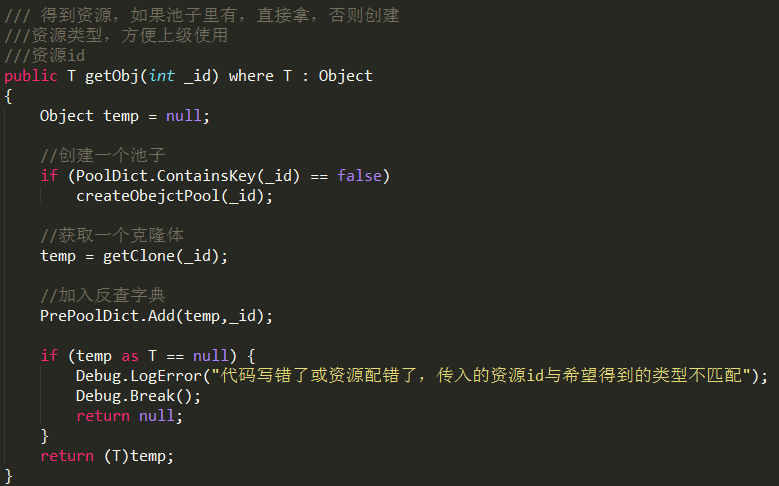
代码如下：



AutoLog就是我们记录代码，在PoolManager中，我定义一个新的字典，用来在运行时候读取与存储与自动记录有关的信息。下面是具体的AutoLog代码。



有了上面两个函数后，我对之前我们的资源getObj函数进行了一些修改，使得可以在加载资源时，把资源表信息的内容，记录下来。



好，有了这些代码以后，我们就可以开始测试了记录工作了。

当然，记录流程呢还有其他代码，比如开始与结束等等，都是一些业务逻辑上的代码，如果我把他们贴上来，就会让你迷糊，所以我贴出关键点，当读者感兴趣时，自己可以查阅github上的工程代码。

资源回收判定

大部分的资源被创建出来后，都有生命周期结束的时刻，当它的生命周期结束时，我们就需要决定是删除它还是仅仅回收到池中。

在我们的解决方案中，我定义了一个规则，并且为了测试，改变了参数。

**1、当一份资源创建时，根据下一次同类资源调用时间决定是否删除**

**2、 为了测试，调用间隔为10秒**

**3、因为要知道同类资源下次调用时间，但又不希望运行时循环表，在自动记录结束时，循环一次表进行判定。**

**4、如果一份资源被预加载了但是很久没被使用过，则从记录表中删除该条信息。（代码中未实现）。**

代码如下：



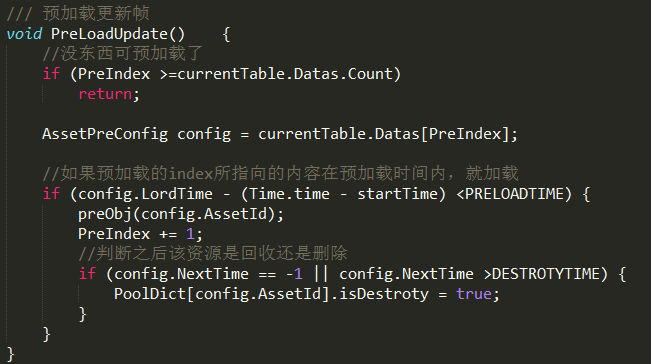
主要逻辑都有注释，所以读者应该可以看清楚关于资源回收的逻辑判定过程。至于额外的代码，就不贴出来，以免脑袋混乱。

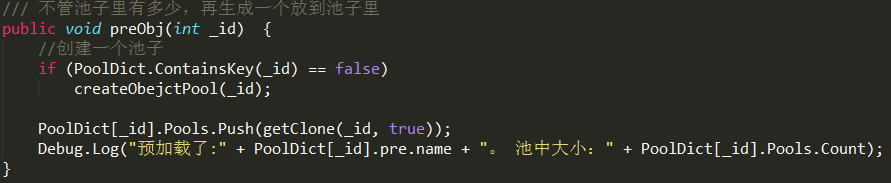
资源自动预加载

当我们有了表，也自动记录了，还有了资源回收机制以后，就可以开心的自动预加载记录好的资源了。

在工程中，我直接把这个过程写在了Update函数中，每一帧都检测当前是否有资源需要加载，同时为了性能考虑，同一帧绝对不加载1份以上的资源。

这里还有优化的空间，我们完全根据性能来决定什么是否集中预加载，什么时候不预加载，比如(战斗过程)。





上面的代码一个Update中运行的，当判断接下来2秒有一份资源请求时，就对其进行预加载。而下面的代码，就是生成一份资源，再直接丢入到池中。这样，当2秒后这份资源需要使时，它就可以直接从池子里获取。

测试

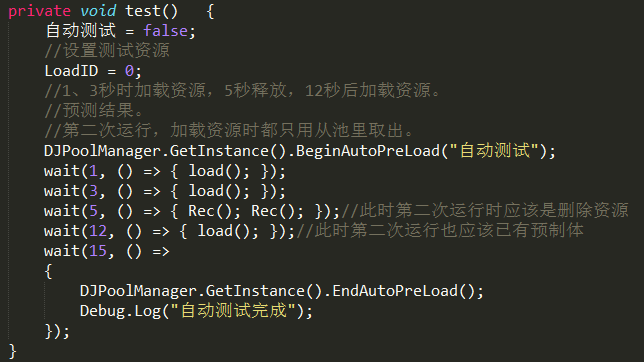
把功能点写完后，我们还需要对自己的代码进行测试，判断是否达到了预期的目标。因为这次测试比较复杂，所以我写了一个简单的测试代码来帮我们完成这个过程。

在场景10-2PooL中，可以找到脚本PoolTest.cs ，里面包含了这次的测试过程，具体规则如下：

**1、第一次测试，没有任何记录存在，每一次资源加载都经过克隆的过程。**

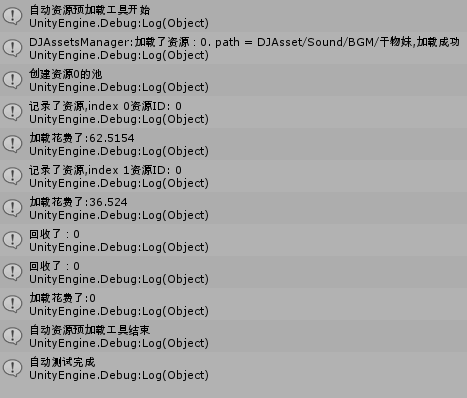
**2、第二次测试，前部分资源拥有记录，所以在回收的时候进行删除。**

**3、第三次测试，因为第二次检测到了后面10秒内还有同类资源，所以前面资源不释放。**

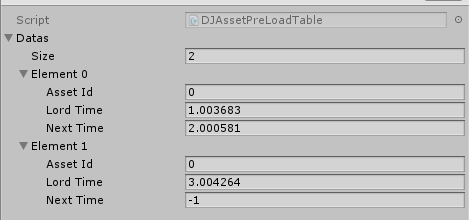


原本我希望第三次测试的时候，应该是再次预加载，前2份资源应该被删掉，但估算时间的时候算错了1秒。导致三次结果都不同，不过觉得这种用例用来展现“自动优化”的过程更好，所以就保留了下来。

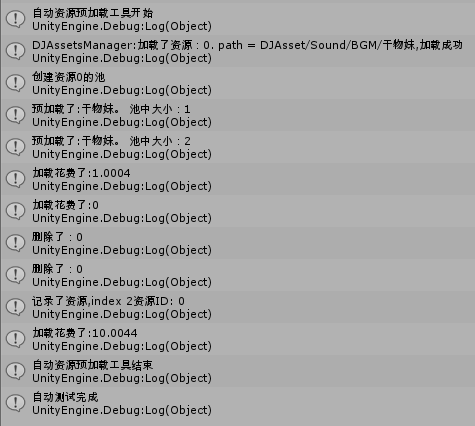
下面，就是三次测试的结果。



第一次 此时记录表内的内容

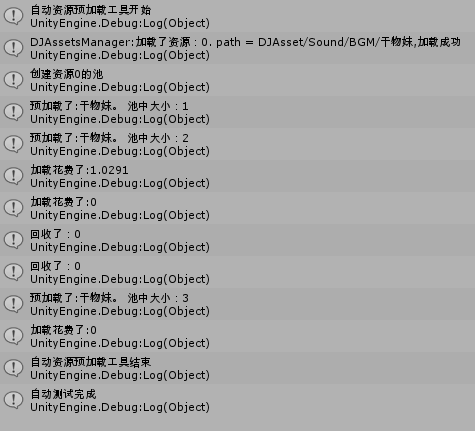


第二次



可以看到，前两次的资源都有预加载，所以时间上间断了。而第三次资源，却比第一次还要多，因为中间发生了资源删除事件。

第三次



这一次，没有任何资源是在使用时才被加载的，前2份资源也不会“轻易”的放弃了自己生命，而是等待这第3份的调用。彻底完成了优化的过程。

结束语

如果和业务逻辑相结合，我们所演示的功能是不够的，但却构建了整个自动化的资源加载与释放的核心框架，使得我们在项目后续的开发过程中，尽可能的不会在IO方面遇到困难。

同时，如果我们能继续对这部分的工作进行优化，还能制作出更平缓的游戏资源IO流程，提供更好的游戏性能。

---------------------

原文：https://blog.csdn.net/swj524152416/article/details/53391926