本文欢迎转载，但烦请保留此行出处信息：<http://www.onevcat.com/2012/11/memory-in-unity3d/>

Unity3D在内存占用上一直被人诟病，特别是对于面向移动设备的游戏开发，动辄内存占用飙上一两百兆，导致内存资源耗尽，从而被系统强退造成极差的体验。类似这种情况并不少见，但是绝大部分都是可以避免的。虽然理论上Unity的内存管理系统应当为开发者分忧解难，让大家投身到更有意义的事情中去，但是对于Unity对内存的管理方式，官方文档中并没有太多的说明，基本需要依靠自己摸索。最近在接手的项目中存在严重的内存问题，在参照文档和Unity Answer众多猜测和证实之后，稍微总结了下Unity中的内存的分配和管理的基本方式，在此共享。

虽然Unity标榜自己的内存使用全都是“Managed Memory”，但是事实上你必须正确地使用内存，以保证回收机制正确运行。如果没有做应当做的事情，那么场景和代码很有可能造成很多非必要内存的占用，这也是很多Unity开发者抱怨内存占用太大的原因。接下来我会介绍Unity使用内存的种类，以及相应每个种类的优化和使用的技巧。遵循使用原则，可以让非必要资源尽快得到释放，从而降低内存占用。

**Unity中的内存种类**

实际上Unity游戏使用的内存一共有三种：程序代码、托管堆（Managed Heap）以及本机堆（Native Heap）。

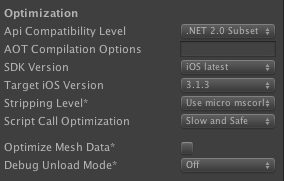
程序代码包括了所有的Unity引擎，使用的库，以及你所写的所有的游戏代码。在编译后，得到的运行文件将会被加载到设备中执行，并占用一定内存。这部分内存实际上是没有办法去“管理”的，它们将在内存中从一开始到最后一直存在。一个空的Unity默认场景，什么代码都不放，在iOS设备上占用内存应该在17MB左右，而加上一些自己的代码很容易就飙到20MB左右。想要减少这部分内存的使用，能做的就是减少使用的库，稍后再说。

托管堆是被Mono使用的一部分内存。[Mono](http://www.mono-project.com/Main_Page)项目一个开源的.net框架的一种实现，对于Unity开发，其实充当了基本类库的角色。托管堆用来存放类的实例（比如用new生成的列表，实例中的各种声明的变量等）。“托管”的意思是Mono“应该”自动地改变堆的大小来适应你所需要的内存，并且定时地使用垃圾回收（Garbage Collect）来释放已经不需要的内存。关键在于，有时候你会忘记清除对已经不需要再使用的内存的引用，从而导致Mono认为这块内存一直有用，而无法回收。

最后，本机堆是Unity引擎进行申请和操作的地方，比如贴图，音效，关卡数据等。Unity使用了自己的一套内存管理机制来使这块内存具有和托管堆类似的功能。基本理念是，如果在这个关卡里需要某个资源，那么在需要时就加载，之后在没有任何引用时进行卸载。听起来很美好也和托管堆一样，但是由于Unity有一套自动加载和卸载资源的机制，让两者变得差别很大。自动加载资源可以为开发者省不少事儿，但是同时也意味着开发者失去了手动管理所有加载资源的权力，这非常容易导致大量的内存占用（贴图什么的你懂的），也是Unity给人留下“吃内存”印象的罪魁祸首。

**优化程序代码的内存占用**

这部分的优化相对简单，因为能做的事情并不多：主要就是减少打包时的引用库，改一改build设置即可。对于一个新项目来说不会有太大问题，但是如果是已经存在的项目，可能改变会导致原来所需要的库的缺失（虽说一般来说这种可能性不大），因此有可能无法做到最优。

[](http://www.onevcat.com/wp-content/uploads/2012/11/unity-setting.png)image

当使用Unity开发时，默认的Mono包含库可以说大部分用不上，在Player Setting（Edit->Project Setting->Player或者Shift+Ctrl(Command)+B里的Player Setting按钮）面板里，将最下方的Optimization栏目中“Api Compatibility Level”选为.NET 2.0 Subset，表示你只会使用到部分的.NET 2.0 Subset，不需要Unity将全部.NET的Api包含进去。接下来的“Stripping Level”表示从build的库中剥离的力度，每一个剥离选项都将从打包好的库中去掉一部分内容。你需要保证你的代码没有用到这部分被剥离的功能，选为“Use micro mscorlib”的话将使用最小的库（一般来说也没啥问题，不行的话可以试试之前的两个）。库剥离可以极大地降低打包后的程序的尺寸以及程序代码的内存占用，唯一的缺点是这个功能只支持Pro版的Unity。

这部分优化的力度需要根据代码所用到的.NET的功能来进行调整，有可能不能使用Subset或者最大的剥离力度。如果超出了限度，很可能会在需要该功能时因为找不到相应的库而crash掉（iOS的话很可能在Xcode编译时就报错了）。比较好地解决方案是仍然用最强的剥离，并辅以较小的第三方的类库来完成所需功能。一个最常见问题是最大剥离时Sysytem.Xml是不被Subset和micro支持的，如果只是为了xml，完全可以导入一个轻量级的xml库来解决依赖（Unity官方推荐[这个](http://unity3d.com/support/documentation/Images/manual/Mono.Xml.zip)）。

关于每个设定对应支持的库的详细列表，可以在[这里](http://docs.unity3d.com/Documentation/ScriptReference/MonoCompatibility.html)找到。关于每个剥离级别到底做了什么，[Unity的文档](http://unity3d.com/support/documentation/Manual/iphone-playerSizeOptimization.html)也有说明。实际上，在游戏开发中绝大多数被剥离的功能使用不上的，因此不管如何，库剥离的优化方法都值得一试。

**托管堆优化**

Unity有一篇不错的关于[托管堆代码如何写比较好](http://unity3d.com/support/documentation/Manual/Understanding%20Automatic%20Memory%20Management.html)的说明，在此基础上我个人有一些补充。

首先需要明确，托管堆中存储的是你在你的代码中申请的内存（不论是用js，C#还是Boo写的）。一般来说，无非是new或者Instantiate两种生成object的方法（事实上Instantiate中也是调用了new）。在接收到alloc请求后，托管堆在其上为要新生成的对象实例以及其实例变量分配内存，如果可用空间不足，则向系统申请更多空间。

当你使用完一个实例对象之后，通常来说在脚本中就不会再有对该对象的引用了（这包括将变量设置为null或其他引用，超出了变量的作用域，或者对Unity对象发送Destory()）。在每隔一段时间，Mono的垃圾回收机制将检测内存，将没有再被引用的内存释放回收。总的来说，你要做的就是在尽可能早的时间将不需要的引用去除掉，这样回收机制才能正确地把不需要的内存清理出来。但是需要注意在内存清理时有可能造成游戏的短时间卡顿，这将会很影响游戏体验，因此如果有大量的内存回收工作要进行的话，需要尽量选择合适的时间。

如果在你的游戏里，有特别多的类似实例，并需要对它们经常发送Destroy()的话，游戏性能上会相当难看。比如[小熊推金币](http://www.onevcat.com/showcase/coinpush/)中的金币实例，按理说每枚金币落下台子后都需要对其Destory()，然后新的金币进入台子时又需要Instantiate，这对性能是极大的浪费。一种通常的做法是在不需要时，不摧毁这个GameObject，而只是隐藏它，并将其放入一个重用数组中。之后需要时，再从重用数组中找到可用的实例并显示。这将极大地改善游戏的性能，相应的代价是消耗部分内存，一般来说这是可以接受的。关于对象重用，可以参考[Unity关于内存方面的文档中Reusable Object Pools部分](http://docs.unity3d.com/Documentation/Manual/UnderstandingAutomaticMemoryManagement.html)，或者Prime31有一个是用Linq来建立重用池的视频教程（Youtube，需要翻墙，[上](http://www.youtube.com/watch?v=IX041ZvgQKE" \t "_blank)，[下](http://www.youtube.com/watch?v=d9078u8ft58)）。

如果不是必要，应该在游戏进行的过程中尽量减少对GameObject的Instantiate()和Destroy()调用，因为对计算资源会有很大消耗。在便携设备上短时间大量生成和摧毁物体的话，很容易造成瞬时卡顿。如果内存没有问题的话，尽量选择先将他们收集起来，然后在合适的时候（比如按暂停键或者是关卡切换），将它们批量地销毁并且回收内存。Mono的内存回收会在后台自动进行，系统会选择合适的时间进行垃圾回收。在合适的时候，也可以手动地调用System.GC.Collect()来建议系统进行一次垃圾回收。要注意的是这里的调用真的仅仅只是建议，可能系统会在一段时间后在进行回收，也可能完全不理会这条请求，不过在大部分时间里，这个调用还是靠谱的。

**本机堆的优化**

当你加载完成一个Unity的scene的时候，scene中的所有用到的asset（包括Hierarchy中所有GameObject上以及脚本中赋值了的的材质，贴图，动画，声音等素材），都会被自动加载（这正是Unity的智能之处）。也就是说，当关卡呈现在用户面前的时候，所有Unity编辑器能认识的本关卡的资源都已经被预先加入内存了，这样在本关卡中，用户将有良好的体验，不论是更换贴图，声音，还是播放动画时，都不会有额外的加载，这样的代价是内存占用将变多。Unity最初的设计目的还是面向台式机，几乎无限的内存和虚拟内存使得这样的占用似乎不是问题，但是这样的内存策略在之后移动平台的兴起和大量移动设备游戏的制作中出现了弊端，因为移动设备能使用的资源始终非常有限。因此在面向移动设备游戏的制作时，尽量减少在Hierarchy对资源的直接引用，而是使用Resource.Load的方法，在需要的时候从硬盘中读取资源，在使用后用Resource.UnloadAsset()和Resources.UnloadUnusedAssets()尽快将其卸载掉。总之，这里是一个处理时间和占用内存空间的trade off，如何达到最好的效果没有标准答案，需要自己权衡。

在关卡结束的时候，这个关卡中所使用的所有资源将会被卸载掉（除非被标记了DontDestroyOnLoad）的资源。注意不仅是DontDestroyOnLoad的资源本身，其相关的所有资源在关卡切换时都不会被卸载。DontDestroyOnLoad一般被用来在关卡之间保存一些玩家的状态，比如分数，级别等偏向文本的信息。如果DontDestroyOnLoad了一个包含很多资源（比如大量贴图或者声音等大内存占用的东西）的话，这部分资源在场景切换时无法卸载，将一直占用内存，这种情况应该尽量避免。

另外一种需要注意的情况是脚本中对资源的引用。大部分脚本将在场景转换时随之失效并被回收，但是，在场景之间被保持的脚本不在此列（通常情况是被附着在DontDestroyOnLoad的GameObject上了）。而这些脚本很可能含有对其他物体的Component或者资源的引用，这样相关的资源就都得不到释放，这绝对是不想要的情况。另外，static的单例（singleton）在场景切换时也不会被摧毁，同样地，如果这种单例含有大量的对资源的引用，也会成为大问题。因此，尽量减少代码的耦合和对其他脚本的依赖是十分有必要的。如果确实无法避免这种情况，那应当手动地对这些不再使用的引用对象调用Destroy()或者将其设置为null。这样在垃圾回收的时候，这些内存将被认为已经无用而被回收。

需要注意的是，Unity在一个场景开始时，根据场景构成和引用关系所自动读取的资源，只有在读取一个新的场景或者reset当前场景时，才会得到清理。因此这部分内存占用是不可避免的。在小内存环境中，这部分初始内存的占用十分重要，因为它决定了你的关卡是否能够被正常加载。因此在计算资源充足或是关卡开始之后还有机会进行加载时，尽量减少Hierarchy中的引用，变为手动用Resource.Load，将大大减少内存占用。在Resource.UnloadAsset()和Resources.UnloadUnusedAssets()时，只有那些真正没有任何引用指向的资源会被回收，因此请确保在资源不再使用时，将所有对该资源的引用设置为null或者Destroy。同样需要注意，这两个Unload方法仅仅对Resource.Load拿到的资源有效，而不能回收任何场景开始时自动加载的资源。与此类似的还有AssetBundle的Load和Unload方法，灵活使用这些手动自愿加载和卸载的方法，是优化Unity内存占用的不二法则～