# **Unity优化技巧（中）**

作者：[韩宇飞](http://gad.qq.com/user/index?id=2044697)

链接：<http://gad.qq.com/article/detail/39805>

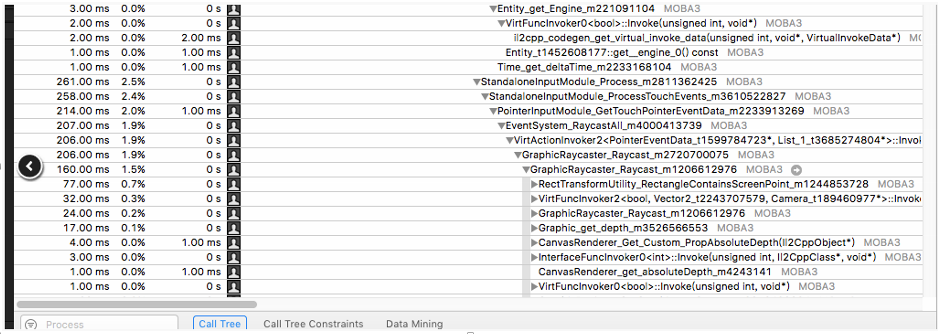
**本文首发于知乎专栏：**[MACK的游戏开发笔记](https://www.zhihu.com/people/han-yu-25-35)**，欢迎各位关注。**

下面介绍一下具体的优化方式，按照硬件划分分为CPU，GPU，内存&硬盘。

**CPU**

* **Top10**

使用Profile找到CPU占用最靠前的函数，从最高的开始依次分析优化。定位的方法有很多，Unity的Profile，UWA的性能测试工具，比较推荐的是使用XCode，可以抓取一段时间内函数的开销。更品均准确也可以看到更底层。



如过函数比较复杂可以使用BeginSample/EndSample拆分，我在项目里通过添加条件编译进行了封装类似：MDebug.BeginSample("Character.TakeDamage");

这样可以跟随意和高效的定位到关心的地方。下面列举部分比较通用的优化方案。

* **LOD，代码只在必要时才会运行**

最常用的优化方式。例如屏幕外的角色不计算动画更新，不计算技能效果冒字血条等，屏幕外的角色休眠，只有主角才会冒字等等。还可以做一些LOD，例如AI可以做行为树lod，动画LOD，粒子特效LOD，更新频率LOD（更新频率随着游戏帧率，离主角的距离，重要程度，视锥，类似CSM的提醒分段，以及周围的角色个数动态调整。例如轩辕城擂台，同屏高配200人，未优化在68帧左右优化的后到94帧左右）

* **限帧，负载均衡**

为了降低耗电发热量我们会根据玩家机型进行限帧。另一方面我们将逻辑帧和渲染帧分开执行，逻辑代码以更低的帧率执行，部分逻辑也可以使用线程，负载均衡分段计算等方法提升性能。

* **算法**

一些代码本身的运算。例如优化物理运算，空间换时间使用查表预计算等方式对计算提速，减少频繁索引FindGetComponentd以及各种运算，优化遍历利用稀疏矩阵九宫四叉树等等。

* **服务器计算或者客户端计算**

根据不同类型的游戏也会调整一些运算是在服务器还是客户端，如果服务器性能强大可以让服务器计算物理，寻路，AI，战斗逻辑等复杂运算，客户端只要变现即可，特别是MMO的项目。而如果希望服务器开发较少提升开发效率和降低服务器性能要求，也可以将绝大部分运算放在客户端，例如帧同步的游戏我们就采取的这种方式。当然有时候也是两者相结合并没有绝对标准。

* **Unity接口**
  + OnGUI，FixedUpdate，Update等空函数也会有gc开销，因为会产生从C++到C#层调用的开销。
  + MainCamera是一个遍历操作Camera比较多的时候不要频繁调用。
  + 尽量少使用GetComponent，AddComponent（还会产生GC），Find等操作。
  + 使用Unity5.6有个新函数SetPositionAndRotation。因为Transform的Position每次脏了会有一次消息，Rotation也会有，并且会开一个线程来做这个操作极大提升性能。所以最好每帧只设置一次，并且使用SetPositionAndRotation一次性设置可以提升一倍的性能开销。
  + 其他。
* **物理**

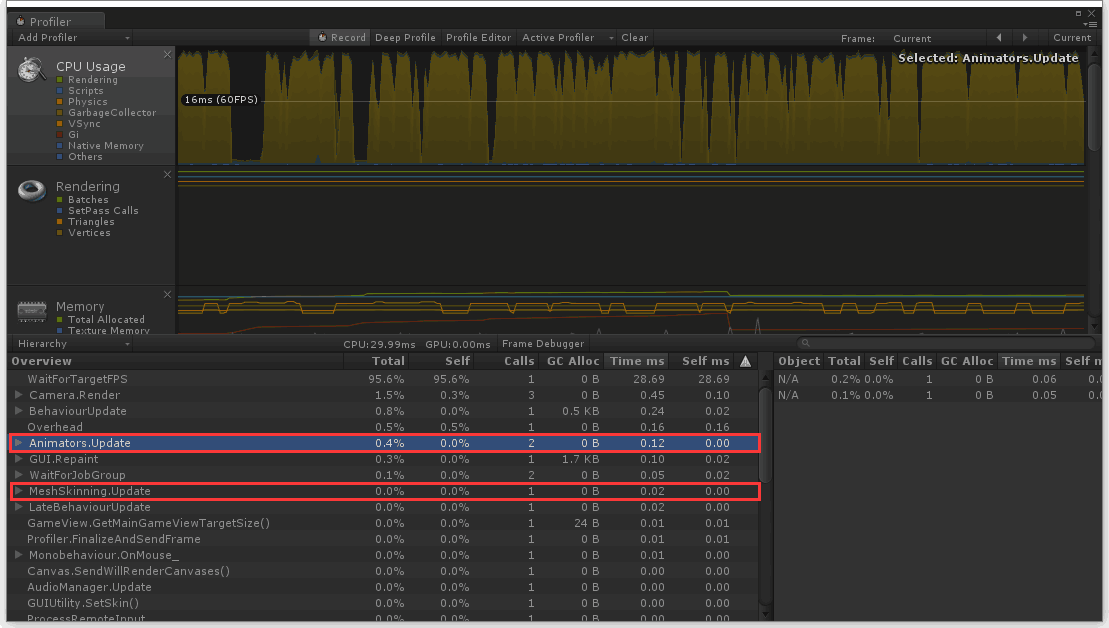
Unity使用PhysX作为物理引擎，本身优化还是很好的，会做空间划分。Unity官方建议碰撞对小于100，其实这个标准非常严苛，我们测试在300左右物理的开销还是蛮少。优化方面可以通过分层减少碰撞对，尽量使用BoxCollider而不是MeshCollider，UI界面不需要点击的控件不要打开Raycaster。因为我们只使用了最基本的射线检测，其他物理是自己实现的，主要的优化还是在物理算法上。如过在Profile中发现Physucs.Simulate开销比较大就是物理需要优化了。

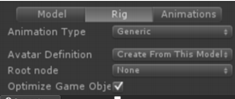
* **IL2CPP & C++**

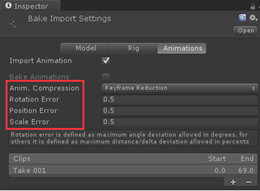
把Unity编译设置成IL2CPP，编译成C++版运行效率会有较大提升。还可以把一些运算逻辑放到C++的库里，这样可以优化更极致减少gc。

* **动画**

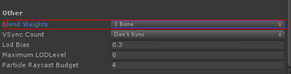
如过Prifile中发现Animator.Update或者MeshSkinning.Udpate开销比较大就说明动作可能需要优化。



* + 优化：打开Optimize  
    GameObject，可以把一些无效的节点骨骼去掉，注意如果有自定义的节点需要拖到不被优化列表里。
  + 压缩：打开Keyframe  
    Reduction，可以压缩很多不必要的关键帧，这个值越大压缩比率越高失真越严重。



* + BoneWeights：顶点受骨骼影响，对要求不高的环境可能一个骨骼就够了。可以每个模型设置，也可以实时全局改变。



* + BakeMesh：对于同屏需要显示大量模型可以使用SkinnedMeshRenderer.BakeMesh，把动画烘成模型，这样在渲染的时候可以合并（带动画不能合并）。可以大幅减少DC，省去蒙皮计算，不过缺点是内存增大，增加DynamicBatching的CPU开销，表现会差一些。
  + 不使用Animator：Animator的开销比Animaton比一个量级。
  + 不可见不更新设置CullCompletely，但是需要注意一些消息也会停掉如果对动画有依赖会出问题。
  + 其他：骨骼LOD，GPU Skinning（有些设备和情况会更慢），使用Bone代替CS等等。



* **UI**

UI也是个开销大头，一般会占到30%-50%。UGUI对应Profile中Canvas.BuildBatch &  
Canvas.SendWillRenderCanvases开销，类似NGUI的LastUpdate，UI的优化又很多文章这里也简单列举一下。

* + 动态静态分离：因为UI会合并。NGUI是按Panel进行重建的、UGUI是按Canvas进行重建的，防止动态UI触发合并导致静态UI也一起合并。
  + 预加载，常驻，即时释放：UI按类型划分，比较大的常用的UI在创建的时候会卡顿，可以进行预加载。主城到战斗场景，在保证峰值内存的情况下，将英雄界面工会界面等常驻内存，可加快Loading速度，实测优化后提升一倍以上loading速度。其他不常用界面拆分成小界面，使用即时加载，关闭时卸载节省内存。需要注意的是，UI节点过多也会导致加载缓慢，我们曾经Loading要10秒，其中序列化UI占了一半左右的时间（贴图预先加载测试），减少UI节点数，太大了拆开。
  + 图集：合理拆分UI图集，区分公共图集（常驻）和非公共图集。太大容易造成冗余加载，容易导致内存占用过大，导致内存显存交换开销。太小有容易导致显存碎片影响效率。规则很复杂。
  + 内存池：UI冒字等频繁创建的UI使用内存池减少创建的时间和内存碎片。
  + Active/Deactive：不推荐通过Active/Deactive来频繁切换UI界面，因为会触发UI合并操作，可以通过移到屏幕外的做法或者设置Layer。但需要注意移到屏幕外还是会被合并渲染，如果是长时间不显示的还是Deactive比较好需要视情况而定。
  + UISprite来代替UITexture：Texture不会合并。
  + 不移动不可见的UI不更新：例如血条名字等。
  + layout group, canvas group组件，任何子节点变了父节点都会用getcompent找到laygroup。这是Unity的UGUI的两大坑。
  + 检查不需要拾取的Raycast target是否关了。
  + 资源预加载：例如前面介绍的UI预加载，内存允许的情况下所有资源都应该预加载，结合内存池。我们游戏中所有变现逻辑，角色，怪物，道具，UI都会做预加载，并且有一套池膨胀和回收的策略。
  + Shader预加载。
  + 等等。
* **GC**

GC是一个非常高开销的系统调用也，是大部分卡顿的主要原因，不能完全控制。因此我们要尽量减少代码堆内存分配过量防止频繁触发GC，同时也可以在Loading或者对性能不敏感的时候主动GC。

* + 升级Unity：Unity5.6修改了粒子系统的源码减少了lamda表达式的gc。
  + 减少一些字符串拼接，使用StringBuilder代替string减少GC开销，不要使用富文本改变Text组件的颜色直接通过修改Text组件颜色来改。
  + 内存池：前面说过GameObject的内存池，另外还有类对象的内存池。所有频繁反复创建删除的都应该使用。两个用途，减少加载创建释放的时间，减少内存碎片降低GC的频率。
  + Unity接口：AddComponent，OnGUI，UI合并频率，delegate，等（一些Foreach，协程等Unity已经优化）。
  + 逻辑优化：避免频繁创建开辟空间。
  + 插件的GC优化：对行为树，FMODStudio等一些插件的源码进行了修改减少GC。
  + 等等。