Unity 优化基础

郑华

2018 年 8 月 9 日

目录

1 基础

DrawCall drawcall 是啥？其实**就是对底层图形程序**（比如：OpenGL ES）**接口的调用**，以在屏幕上画出东西。

一个 DrawCall
- 一次 DirectX -> DrawIndexedPrimitive()
- 一次 OpenGL -> DrawElements()
- 一个 Batch

所以，是**谁去调用这些接口**呢？--> CPU。

Fragment VF(Vertex,Fragment)，Vertex 是**顶点**，那Fragment 是啥呢？说它之前需要先说一下**像素**，像素是构成**数码影像**的基本单元呀。e 而Fragment 是有可能成为**像素**的东西。啥叫有可能？就是最终会不会被画出来不一定，是潜在的像素。

所以，这会**涉及到谁**呢？--> GPU。

Batching batching 是啥？**将批处理之前需要很多次调用**（*drawcall*）**的物体合并，之后只需要调用一次底层图形程序的接口就行**。听上去这简直就是优化的终极方案啊！但是，理想是美好的。

例子 如果在画面上有一张“木”椅子、一张“铁”桌子，既两物件分别使用不同材质球或者不同的 Shader，那么理论上就会有两个 Draw Call。

在 DirectX 或 OpenGL 里，对不同物件指定不同贴图或不同 Shader 的描述，就会需要呼叫两次 Draw Call。具体的伪代码如下：

```
    // 渲染桌子
    SetShader( “Diffuse"␣);
␣␣␣␣␣SetTexture(␣“铁" );
    DrawPrimitive( DeskVertexBuffer );

    // 渲染椅子
    SetShader( “VertexLight"␣);
␣␣␣␣␣SetTexture(␣“木" );
    DrawPrimitive( ChairVertexBuffer );
```

每次对 Sahder 的变动或者贴图的变动，基本上就是对 Rendering Pipeline 的设定做修改，所以需要不同的 Draw Call 来完成物件的绘制。这就是为什么 UNITY 官方文件里，要你尽量使用相同的材质球，就是为了以减少 Draw Call 的数量！

-->Notice: 而Batch 就是把材质球相同的（Shader）、贴图相同的（Texture）、**用一个 Draw-Call 优化的手段**。

内存的分配 除了Unity3D自己的内存损耗。我们可是还带着 Mono 呢啊，还有托管的那一套东西呢。更别说你一激动，又引入了自己的几个dll。这些都是内存开销上需要考虑到的。

总结 对于普通的优化需要注意的方面有以下三个方面：CPU 、GPU 、Memory 。

2 CPU 方面的优化

2.1 DrawCalls

DrawCall 是 **CPU 调用底层图形接口**。比如有上千个物体，每一个的渲染都需要去调用一次底层接口，而每一次的调用 CPU 都需要做很多工作，那么 CPU 必然不堪重负。但是对于 GPU 来说，图形处理的工作量是一样的。

所以**对 DrawCall 的优化**，主要就是**为了尽量解放 CPU 在调用图形接口上的开销**。所以针对 drawcall 我们**主要的思路**就是每个物体尽量减少渲染次数，多个物体最好一起渲染。所以，按照这个思路就有了以下几个方案：

1. 使用Draw Call Batching，也就是**描绘调用批处理**。Unity 在运行时可以将一些物体进行合并，从而用一个描绘调用来渲染他们。具体下面会介绍。
2. 通过把**纹理打包成图集**来尽量减少材质的使用。
3. **尽量少的使用反光**，阴影之类的，因为那会使物体多次渲染。

2.2 Draw Call Batching

首先我们**要先理解**为何 2 个没有使用相同材质的物体即使使用批处理，也无法实现 *Draw Call* 数量的下降和性能上的提升。

因为被“批处理”的 2 个物体的网格模型需要使用相同材质的目的，在于其纹理是相同的，这样才可以实现同时渲染的目的。因而保证材质相同，是为了保证被渲染的纹理相同。

因此，为了将 2 个纹理不同的材质合二为一，我们就需要进行上面列出的第二步，**将纹理打包成图集**。具体到合二为一这种情况，就是将 2 个纹理合成一个纹理。这样我们**就可以只用一个材质来代替之前的 2 个材质了**。

而Draw Call Batching 本身，也还会**细分为 2 种**-静态批处理、动态批处理。

Static Batching

静态？那就是不动的咯。还有呢？额，听上去状态也不会改变，没有“生命”，比如山山石石，楼房校舍啥的。那和什么比较类似呢？嗯，聪明的各位一定觉得和场景的属性很像吧！所以我们的场景似乎就可以采用这种方式来减少 draw call 了。

那么写个定义：**只要这些物体不移动，并且拥有相同的材质，静态批处理就允许引擎对任意大小的几何物体进行批处理操作来降低描绘调用。**

那要如何使用静态批来减少 Draw Call 呢？你只需要明确指出哪些物体是静止的，并且在游戏中永远不会移动、旋转和缩放。想完成这一步，你只需要在检测器（Inspector）中将 Static 复选框打勾即可，如下图所示：

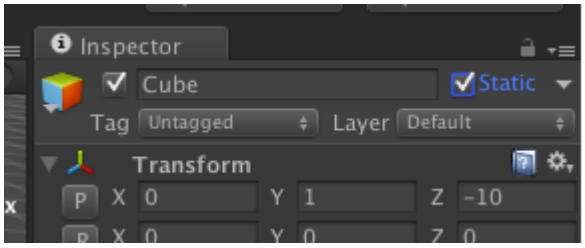

图 1: 设置

至于效果如何呢？

举个例子：新建 4 个物体，分别是Cube，Sphere, Capsule, Cylinder, 它们有**不同的网格模型**，但是也有相同的材质（Default-Diffuse）。

首先，我们不指定它们是 *static* 的。**Draw Call 的次数是 4 次**，如图：

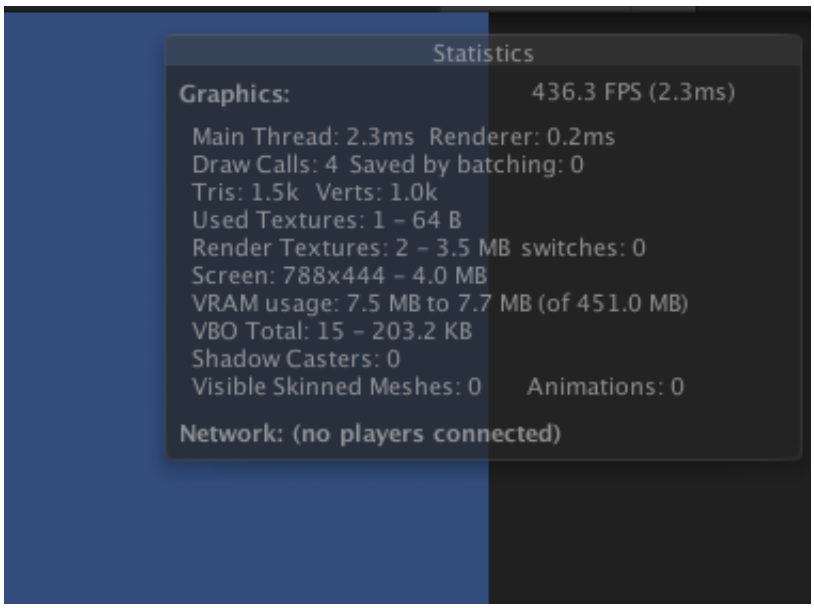

图 2: 未设置静态批处理

我们**现在将它们 4 个物体都设为 static**，在来运行一下：

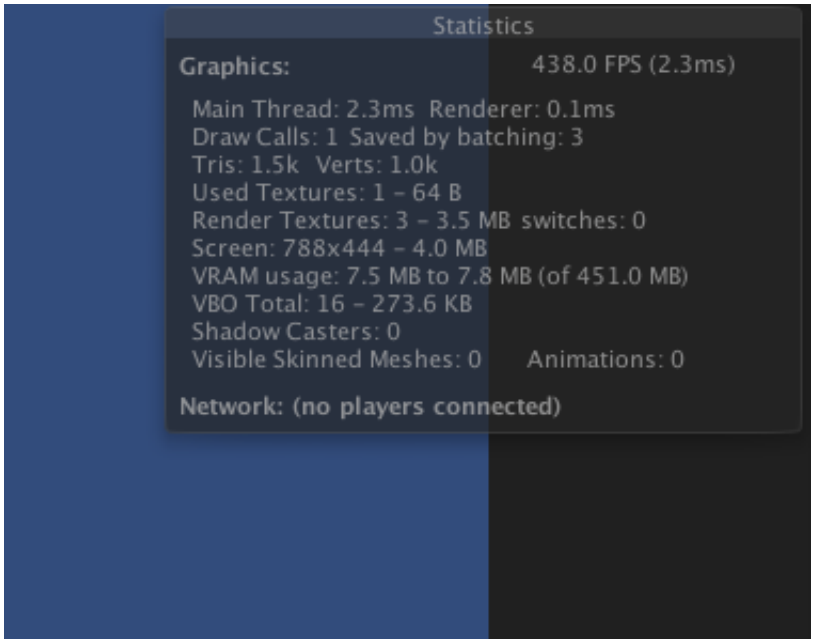

图 3: 设置静态批处理后

如图，**Draw Call 的次数变成了 1**，而`Saved by batching`的次数变成了3。

静态批处理的好处很多，**其中之一**就是与下面要说的动态批处理相比，约束要少很多。**所以一般推荐的是 draw call 的静态批处理来减少 draw call 的次数**。那么接下来，我们就继续聊聊 draw call 的动态批处理。

Dynamic Batching

首先要明确一点，Unity3D 的 **draw call 动态批处理机制**是引擎自动进行的，无需像静态批处理那样手动设置 *static*。我们举一个动态实例化prefab 的例子，如果动态物体**共享相同的材质**，则引擎会自动对 *draw call* 优化，也就是使用批处理。首先，我们将一个cube 做成prefab，然后再实例化 500 次，看看draw call 的数量。

```
for(int i = 0; i < 500; i++)
{
    GameObject cube;
    cube = GameObject.Instantiate(prefab) as GameObject;
}
```

draw call 的数量：

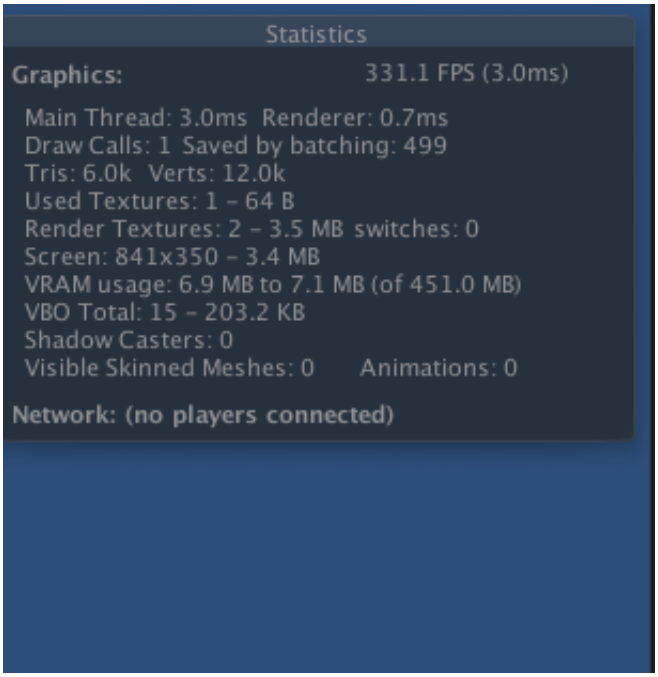

图 4: 动态批处理-1

可以看到draw call 的数量为 1，而 saved by batching 的数量是 499。而这个过程中，我们除了实例化创建物体之外什么都没做。不错，unity3d 引擎为我们自动处理了这种情况。

但是有很多童靴也遇到**这种情况**，就是我也是**从 prefab 实例化创建的物体，为何我的 draw call 依然很高呢**？这就是匹夫上文说的，draw call 的**动态批处理存在着很多约束**。

下面就演示一下，**针对 cube 这样一个简单的物体的创建，如果稍有不慎就会造成 draw call 飞涨的情况**。

我们同样是创建 500 个物体，**不同的**是其中的 *100* 个物体，每个物体的大小都不同，也就是 *Scale* 不同。

```
for(int i = 0; i < 500; i++)
{
    GameObject cube;
    cube = GameObject.Instantiate(prefab) as GameObject;
    if(i / 100 == 0)
    {
        cube.transform.localScale = new Vector3(2 + i, 2 + i, 2 + i);
    }
}
```

图 5: 动态批处理-2

我们看到draw call 的数量**上升到了 101 次**，而saved by batching 的数量也下降到了 399。各位看官可以看到，仅仅是一个简单的cube 的创建，如果scale 不同，竟然也不会去做批处理优化。

这仅仅是动态批处理机制的一种约束，那我们**总结一下动态批处理的约束**，各位也许也能从中找到为何动态批处理在自己的项目中不起作用的原因：

1. 批处理动态物体需要在每个顶点上进行一定的开销，所以**动态批处理仅支持小于 900 顶点的网格物体**。

2. **如果**你的**着色器**使用**顶点位置，法线和 UV 值**三种属性，**那么你只能批处理 300 顶点以下的物体；如果**你的**着色器**需要使用顶点位置，法线，*UV0*，*UV1* 和切向量，那你只能批处理 180 顶点以下的物体。

3. **不要使用缩放**。分别拥有缩放大小 (1,1,1) 和 (2,2,2) 的两个物体**将不会进行批处理**。

4. 统一缩放的物体不会与非统一缩放的物体进行批处理。

5. 使用缩放尺度 (1,1,1) 和 (1,2,1) 的两个物体将不会进行批处理，但是使用缩放尺度 (1,2,1) 和 (1,3,1) 的两个物体将可以进行批处理。

6. **使用不同材质的实例化物体（instance）将会导致批处理失败**。

7. 拥有 lightmap 的物体含有额外（隐藏）的材质属性，比如：lightmap 的偏移和缩放系数等。所以，**拥有 lightmap 的物体将不会进行批处理**（除非他们指向 *lightmap* 的同一部分）。

8. **多通道的 shader 会妨碍批处理操作**。比如，几乎 unity 中所有的着色器在前向渲染中都支持多个光源，并为它们有效地开辟多个通道。

9. **预设体的实例会自动地使用相同的网格模型和材质**。

所以，**尽量使用静态的批处理**。

2.3 物理组件

从性能优化的角度考虑，物理组件能少用还是少用为好。

2.4 处理内存-GC

虽然 GC 是用来处理内存的，但的确增加的是 CPU 的开销。因此它的确能达到释放内存的效果，但代价更加沉重，会加重 CPU 的负担，因此**对于 GC 的优化目标**就是尽量少的触发 *GC*。

首先我们要明确所谓的 **GC 是 Mono 运行时的机制**，而非 *Unity3D* 游戏引擎的机制，所以 **GC 也主要是针对 Mono 的对象来说的，而它管理的也是 Mono 的托管堆**。

搞清楚这一点，你也就明白了GC **不是用来处理引擎的** *assets*（纹理啦，音效啦等等）的内存释放的，**因为 U3D 引擎也有自己的内存堆而不是和 Mono 一起使用所谓的托管堆**。

其次我们要搞清楚**什么东西会被分配到托管堆上**？是**引用类型**。比如**类的实例，字符串，数组**等等。而作为 *int*, *float*, 包括结构体 *struct* 其实都是值类型，它们会被分配在栈上而非堆上。所以**我们关注的对象无外乎就是类实例，字符串，数组这些了**。

那么 **GC 什么时候会触发**呢？两种情况：

1. 首先**堆的内存不足时**，会自动调用 GC。

2. 其次也可以通过编程**手动的调用** GC。

所以为了达到优化 CPU 的目的，我们就不能频繁的触发 GC。而上文也说了 GC 处理的是托管堆，而不是 Unity3D 引擎的那些资源，**所以 GC 的优化说白了也就是代码的优化**。那么匹夫觉得有以下几点是需要注意的：

1. **字符串连接的处理**。因为将两个字符串连接的过程，其实是生成一个新的字符串的过程。而之前的旧的字符串自然而然就成为了垃圾。而作为引用类型的字符串，其空间是在堆上分配的，被弃置的旧的字符串的空间会被 *GC* 当做垃圾回收。

2. **尽量不要使用 foreach，而是使用 for**。*foreach* 其实会涉及到迭代器的使用，而据传说每一次循环所产生的迭代器会带来 *24 Bytes* 的垃圾。那么循环 *10* 次就是 *240Bytes*。

3. **不要直接访问 gameobject 的 tag 属性**。比如`if(go.tag == “human”)` 最好换成`if(go.CompareTag (“human”))`。因为访问物体的 *tag* 属性会在堆上额外的分配空间。如果在循环中这么处理，留下的垃圾就可想而知了。

4. **使用“池”，以实现空间的重复利用**。

2.5 代码-脚本

这里要提到的所谓代码质量是基于一个前提的：Unity3D 是用 C++ 写的，而我们的代码是用 C# 作为脚本来写的，那么问题就来了 **脚本和底层的交互开销是否需要考虑呢？**也就是说，**我们用 Unity3D 写游戏的“游戏脚本语言”，也就是 C# 是由 mono 运行时托管的**。而功能是底层引擎的 C++ 实现的，**“游戏脚本”中的功能实现都离不开对底层代码的调用。那么这部分的开销，我们应该如何优化呢？**

调用方式 ->

以物体的`Transform`组件为例，我们应该只访问一次，之后就将它的引用保留，而非每次使用都去访问。这里有人做过一个小实验，就是对比通过方法`GetComponent<Transform>()` 获取`Transform` 组件, 通过`MonoBehaviour`的`transform` 属性去取，以及保留引用之后再去访问所需要的时间：

- `GetComponent<Transform>()` = 619ms
- `Monobehaviour.Tramsform` = 60ms
- `CachedMB` = 8ms

- `Manual Cache` = 3ms

如上所述，**最好不要频繁使用**`GetComponent`，尤其是在循环中。

没用关闭可见性 ->

善于使用`OnBecameVisible()`, 来控制物体的`update()` 函数的执行**以减少开销**。

使用内建的数组 ->

比如用`Vector3.zero` 而不是`new Vector(0, 0, 0);`

对于方法的参数的优化 ->

善于使用 **ref 关键字-->***对象的传递都是引用。*

*值类型的参数，是通过将实参的值复制到形参，来实现按值传递到方法，也就是我们通常说的按值传递。*复制嘛，总会让人感觉很笨重。比如`Matrix4x4` 这样比较复杂的值类型，如果直接复制一份新的，反而不如将值类型的引用传递给方法作为参数。

3 GPU 方面的优化

GPU 与 CPU 不同，所以**侧重点也不一样**。**GPU 的瓶颈**主要存在**在如下的方面：**

- **填充率**，*可以简单的理解为图形处理单元每秒渲染的像素数量。*
- **像素的复杂度**，*比如动态阴影，光照，复杂的 shader* 等等
- **几何体的复杂度**（*顶点数量*）
- **GPU 的显存带宽**

那么针对以上 4 点，其实仔细分析我们就可以发现，影响的 GPU 性能的无非就是 2 大方面，**一方面是顶点数量过多，像素计算过于复杂**。*另一方面就是 GPU 的显存带宽*。那么针锋相对的两方面举措也就十分明显了。

- 减少顶点数量，简化计算复杂度。
- 压缩图片，以适应显存带宽。

3.1 减少绘制的数目

那么第一个方面的优化也就是减少顶点数量，简化复杂度，具体的举措就总结如下了：

- **保持材质的数目尽可能少**。这使得 Unity 更容易进行批处理。
- **使用纹理图集**（一张大贴图里包含了很多子贴图）来代替一系列单独的小贴图。它们可以更快地被加载，具有很少的状态转换，而且批处理更友好。
- **如果使用了纹理图集和共享材质,使用 Renderer.sharedMaterial 来代替 Renderer.material**。
- **使用光照纹理 (lightmap) 而非实时灯光**。
- 使用 LOD，好处就是对那些离得远，看不清的物体的细节可以忽略。
- **遮挡剔除**（Occlusion culling）
- 使用 mobile 版的 shader。因为简单。

3.2 优化显存带宽

第二个方向呢？压缩图片，减小显存带宽的压力。

- OpenGL ES 2.0 使用 ETC1 格式压缩等等，在打包设置那里都有。
- **使用 mipmap**。

3.3 MipMap

这里匹夫要着重介绍一下`MipMap` 到底是啥。因为有人说过`MipMap` 会占用内存呀，但为何又会优化显存带宽呢？那就不得不从`MipMap` 是什么开始聊起。一张图其实就能解决这个疑问。

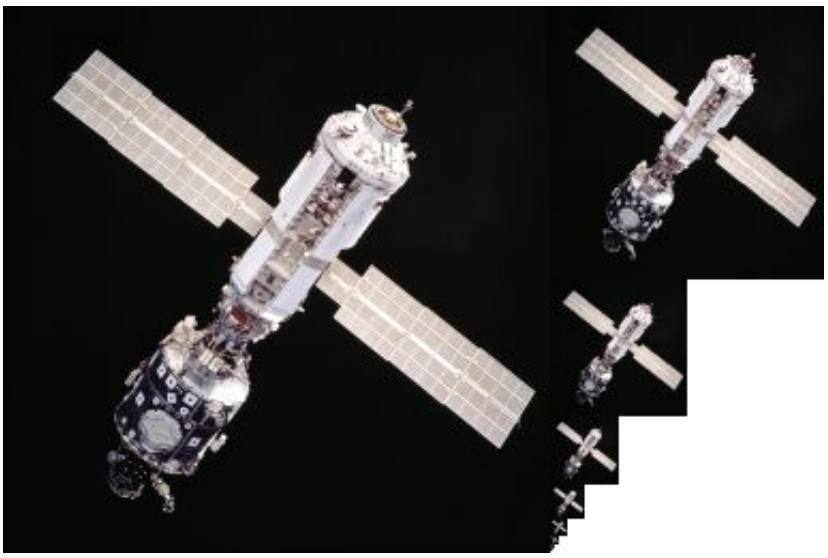

图 6: 上面是一个 mipmap 如何储存的例子，左边的主图伴有一系列逐层缩小的备份小图

是不是很一目了然呢？`Mipmap` 中每一个层级的小图都是主图的一个特定比例的缩小细节的复制品。因为存了主图和它的那些缩小的复制品，所以内存占用会比之前大。但是为何又优化了显存带宽呢？**因为可以根据实际情况，选择适合的小图来渲染。**所以，**虽然会消耗一些内存，但是为了图片渲染的质量（比压缩要好），这种方式也是推荐的。**

4 内存方面的优化

既然要聊 Unity3D 运行时候的内存优化，那我们自然首先要知道 Unity3D 游戏引擎是如何分配内存的。大概可以分成三大部分：

- Unity3D 内部的内存
- Mono 的托管内存
- 若干我们自己引入的 DLL 或者第三方 DLL 所需要的内存。

4.1 U3D 内部内存

Unity3D 的内部内存都会存放一些什么呢？各位想一想，除了用代码来驱动逻辑，一个游戏还需要什么呢？对，各种资源。所以简单总结一下 Unity3D 内部内存存放的东西吧：

- 资源：纹理、网格、音频等等
- `GameObject`和各种组件。

- 引擎内部逻辑需要的内存：渲染器，物理系统，粒子系统等等

4.2 Mono 托管内存

因为我们的游戏脚本是用C# 写的，**同时还要跨平台**，所以带着一个 **Mono 的托管环境**显然**必须的**。那么 Mono 的托管内存自然就不得不放到内存的优化范畴中进行考虑。那么我们所说的 Mono 托管内存中存放的东西和 Unity3D 内部内存中存放的东西究竟有何不同呢？其实 Mono 的内存分配就是很传统的运行时内存的分配了：

值类型：*int* 型啦，*float* 型啦，结构体 *struct* 啦，*bool* 啦之类的。它们都存放在堆栈上（注意额，不是堆所以不涉及 GC）。

引用类型：其实可以狭义的理解为各种类的实例。比如游戏脚本中对游戏引擎各种控件的封装。其实很好理解，C# 中肯定要有对应的类去对应游戏引擎中的控件。那么这部分就是C# 中的封装。由于是在堆上分配，所以会涉及到 GC。

而 Mono 托管堆中的那些封装的对象，除了在在 Mono 托管堆上分配封装类实例化之后 所需要的内存之外，还会牵扯到其背后对应的游戏引擎内部控件在 Unity3D 内部内存上的分配。

例子-WWW ->

一个在.cs **脚本中**声明的 **WWW 类型的对象 www**，*Mono* 会在 **Mono 托管堆**上**为 www** 分配它所需要的**内存**。同时，这个实例对象背后的所代表的引擎资源所需要的内存也需要被分配。

一个WWW 实例背后的资源：

- 压缩的文件
- 解压缩所需的缓存
- 解压缩之后的文件

如下图所示：

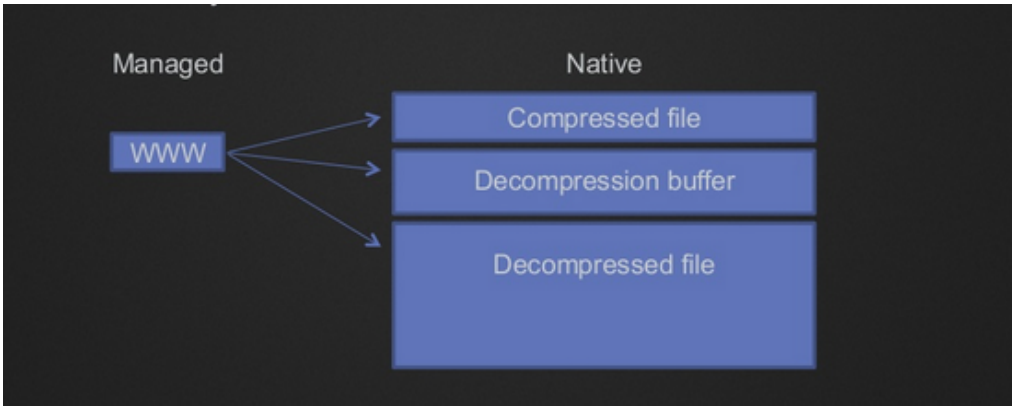

图 7: Mono 托管堆

例子-AssetBundle 内存处理 ->

以下载 Assetbundle 为例子，聊一下内存的分配。

```
IEnumerator DownloadAndCache ()
{
    // Wait for the Caching system to be ready
    while (!Caching.ready)
        yield return null;

    // Load the AssetBundle file from Cache if it exists with the same version or download
        and store it in the cache
    using(WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version))
    {
        yield return www; //WWW是第1部分
        if (www.error != null)
            throw new Exception("WWW␣download␣had␣an␣error:" + www.error);
        AssetBundle bundle = www.assetBundle;//AssetBundle是第2部分
        if (AssetName == "")
            Instantiate(bundle.mainAsset);//实例化是第3部分
        else
            Instantiate(bundle.Load(AssetName));
        // Unload the AssetBundles compressed contents to conserve memory
        bundle.Unload(false);

    } // memory is freed from the web stream (www.Dispose() gets called implicitly)
}
```

内存分配的三个部分匹夫已经在代码中标识了出来：

- Web Stream：包括了压缩的文件，解压所需的缓存，以及解压后的文件。

- `AssetBundle`：Web Stream 中的文件的映射，或者说引用。
- *实例化之后的对象*：就是引擎的各种资源文件了，会在内存中创建出来。

```
WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)
```

1. 将压缩的文件读入内存中
2. 创建解压所需的缓存
3. 将文件解压，解压后的文件进入内存
4. 关闭掉为解压创建的缓存

```
AssetBundle bundle = www.assetBundle;
```

1. AssetBundle 此时相当于一个桥梁，从 Web Stream 解压后的文件到最后实例化创建的对象之间的桥梁。
2. 所以 AssetBundle 实质上是 Web Stream 解压后的文件中各个对象的映射。而非真实的对象。
3. 实际的资源还存在 Web Stream 中，所以此时要保留 Web Stream。

```
Instantiate(bundle.mainAsset);
```

通过 AssetBundle 获取资源，实例化对象。最后各位可能看到了官网中的这个例子使用了：

```
using(WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)){ }
```

这种`using` 的用法。这种用法其实就是为了在使用完`Web Stream` 之后，将内存释放掉的。因为 WWW 也继承了`idispose` 的接口，所以可以使用`using` 的这种用法。其实相当于最后执行了：

```
www.Dispose();
```

OK,`Web Stream` 被删除掉了。那还有谁呢？对 Assetbundle。那么使用

```
bundle.Unload(false);
```

4.3 使用 Unity Profier 检测内存

5 参考

优化基础：https://blog.csdn.net/cordova/article/details/52241249

相关解释：https://blog.csdn.net/hany3000/article/details/44033243