移动端性能优化

一. 资源内存、Mono 堆内存等常见游戏内存控制

1. 概念解释

首先，在讨论内存相关的各项参数和制定标准之前，我们需要先理清在各种性能工具的统计数据中常出现的各种内存参数的实际含义。

在安卓系统中，我们最常见到和关心的 PSS（Proportional Set Size）内存，其含义为一个进程在 RAM 中实际使用的空间地址大小，即实际使用的物理内存。就结果而言，当一个游戏进程中 PSS 内存峰值越高、占当前硬件的总物理内存的比例越高，则该游戏进程被系统杀死（闪退）的概率也就越高。

而在 PSS 内存中，除了 Unused（未使用）部分外，我们一般比较关心 Reserved Total 内存和 Lua、Native 代码、插件等系统缓存、第三方库的自身分配等内存，因此 Reserved Total 占比一般较高，故其大小和走势，也是性能分析工具的主要统计对象。

Reserved Total 和 Used Total 为 Unity 引擎在内存方面的总体分配量和总体使用量。一般来说，引擎在分配内存时并不是向操作系统“即拿即用”而是首先获取一定量的连续内存，然后供自己内部使用，待空余内存不够时，引擎才会向系统再次申请一定量的连续内存进行使用。

注意：对于绝大多数平台而言，Reserved Total 内存＝Reserved Unity 内存＋GFX 内存＋FMOD 内存＋Mono 内存。

1) Reserved Unity 内存

Reserved Unity 和 Used Unity 为 Unity 引擎自身各个模块内部的内存分配，包括各个 Manager 的内存占用、序列化信息的内存占用和部分资源的内存占用等等。

通过针对大量项目的深度分析，发现导致 Reserved Unity 内存分配较大的原因主要有以下几种：

序列化信息内存占用：Unity 引擎的序列化信息种类繁多，其中最为常见且内存占用较大的为 SerializedFile。该序列化信息的内存分配主要是项目通过特定 API（WWW.LoadFromCacheOrDownload、CreateFromFile 等）加载 AssetBundle 文件所致。

资源内存占用：主要包括 Mesh、AnimationClip、RenderTextures 等资源。对于未开启“Read/Write Enable”选项的 Mesh 资源，其内存占用是统计在 GFX 内存中供 GPU 使用的，但开启该选项后，网格数据会在 Reserved Unity 中保留一份，便于项目在运行时对 Mesh 数据进行实时的编辑和修改。同时，如果研发团队同样开启了纹理资源的“Read/Write Enable”选项（默认情况下为关闭），则纹理资源同样会在 Reserved Unity 中保留一份，进而造成其更大的内存占用。

2)GFX 内存

GFX 内存为底层显卡驱动所反馈的内存分配量，该内存分配由底层显卡驱动所控制。一般来说，该部分内存占用主要由渲染相关的资源量所决定，包括纹理资源、Mesh 资源、Shader 资源传向 GPU 的部分，以及解析这些资源的相关库所分配的内存等。

3)托管堆内存

托管堆内存表示项目运行时代码分配的托管堆内存分配量。对于使用 Mono 进行代码编译的项目，其托管堆内存主要由 Mono 分配和管理；对于使用 IL2CPP 进行代码编译的项目，其托管堆内存主要由 Unity 自身分配和管理。

4)内存参数标准

在我们了解了内存相关的各项参数的含义之后，知道了避免游戏闪退的重点在于控制

PSS 内存峰值。而 PSS 内存的大头又在于 Reserved Total 中的资源内存和 Mono 堆内存。对于使用 Lua 的项目来说，还应关注 Lua 内存。

只有当 PSS 内存峰值控制在硬件总内存的 0.5-0.6 倍以下的时候，闪退风险才较低。举例而言，对于 2GB 的设备而言，PSS 内存应控制在 1GB 以下为最佳，3GB 的设备则应控制在 1.5GB 以下。

而对于大多数项目而言，PSS 内存大约高于 Reserved Total 200MB-300MB 左右，故 2GB 设备的 Reserved Total 应控制在 700MB 以下、3GB 设备则控制在 1GB 以下。

特别的，Mono 堆内存需要予以关注，因为在很多项目中，Mono 堆内存除了存在本身驻留偏高或存在泄露风险的问题外，其大小还会影响 GC 耗时。将其控制在 80MB 以下为最佳。

下表提供了细化到每一种资源内存的推荐标准，制定较为严格。不过，仍需要开发者根据自身项目的实际情况予以调整。比如某个 2D 项目节省了几乎所有网格资源的使用，那么其他资源的标准就可以放宽很多。

内存类型		推荐值	
		2G	3G
资源内存	Texture	140MB	210 MB
	Mesh	60 MB	100 MB
	Shader	40 MB	60 MB
	Animation Clip	40 MB	60 MB
Mono 堆内存		80 MB	80 MB
Lua 内存		100 MB	100 MB

基于项目实情制定内存标准后，一般需进一步与美术、策划协商，给出合理的美术规范参数，并撰写成文档。定好资源规范后，定时检查项目里的所有美术资源是否符合规范，及时修改和更新。检查美术是否合规，可以利用 Unity 提供的回调函数去编写自动化工具，提高效率。

如果资源若不能批量处理成高中低配版本，就需要美术为各个画质等级制作不同的资源。

2. 常见的共通性问题

1) 疑似冗余现象

一般情况下，出现这种问题是由 AssetBundle 资源加载导致的，即在制作 AssetBundle 文件时，部分共享资源（比如 Texture、Mesh 等）被同时打入到多份不同的 AssetBundle 文件中但没有进行依赖打包，从而当加载这些 AssetBundlel 时，内存中出现了多份同样的资源，即资源冗余，建议对其进行严格的检测和完善。

值得一提的是，所谓“疑似冗余资源”，是指在检测过程中，我们尝试搜索项目运行时的冗余资源并将其反馈给用户。但是，我们并无法保证该项检测的 100％正确性。这是因为，我们判断的标准是根据资源的名称、内存占用等属性（因资源类型不同可能有格式、Read/Write、时长等属性，以报告资源列表中呈现的属性为准）而定，当两个资源的名称、内存占用等属性均一致时，我们认为这两个资源可能为同一资源，即其中一个为“冗余”资源。但项目中确实也存在资源不同但各项属性都相同的情况。因此，我们将通过以上规则提取出的资源归为“疑似冗余资源”。所以，是否确实为冗余资源，还需要结合项目实情和在线 AssetBundle 检测报告才能下结论。

2)未命名资源

在资源列表中，有时发现存在资源名称为 N/A 的资源。一般来说名为 N/A 的资源都是在代码中 new 出来但是没有予以命名的。建议通过.name 方法对这些资源进行命名，方便资

源统计和管理，尤其是其中冗余比较严重的或者个别内存占用非常大的 N/A 资源应予以关注和严格排查。

3)常驻资源内存占用大

在资源列表中，有时结合资源的生命周期曲线发现，一部分本身内存占用较大的资源在被加载进内存后，驻留在内存中，直到测试流程结束都没有被卸载，可能造成越到游戏后期资源内存占用越大、峰值越高。建议排查这些资源是否有常驻在内存中的必要。如果不再需要被使用，则应检查为什么场景切换时没有卸载；对于持续时间久的单场景中持续驻留的资源，则可以考虑手动卸载。

对于资源是否常驻的考量涉及内存压力和 CPU 耗时压力之间的取舍。简单来说，如果当前项目内存压力较大，而场景切换时的 CPU 耗时压力较小，则可以考虑改变缓存策略，在场景切换时及时卸载下一个场景用不到的资源，在需要时再重新加载。

4)资源分离打包与加载

游戏中会有很多地方使用同一份资源。比如，有些界面共用同一份字体、同一张图集，有些场景共用同一张贴图，有些怪物使用同一个 Animator 等等。在制作游戏安装包时将这些公用资源从其它资源中分离出来，单独打包。比如若资源 A 和 B 都引用了资源 C，则将 C 分离出来单独打一个 bundle。

在游戏运行时，如果要加载 A，则先加载 C；之后如果要加载 B，因为 C 的实例已经在内存，所以只要直接加载 B，让 B 指向 C 即可。如果打包时不将 C 从 A 和 B 分离出来，那么 A 的包里会有一份 C，B 的包里也会有一份 C，冗余的 C 会将安装包撑大；并且在运行时，如果 A 和 B 都加载进内存，内存里就会有两个 C 实例，增大了内存占用。

资源分离打包与加载是最有效的减小安装包体积与运行时内存占用的手段。一般打包粒度越细，这两个指标就越小；而且当两个 renderQueue 相邻的 DrawCall 使用了相同的贴图、材质和 shader 实例时，这两个 DrawCall 就可以合并。但打包也并不是越细就越好。如果运行时要同时加载大量小 bundle，那么加载速度将会非常慢——时间都浪费在协程之间的调度和多批次的小 I/O 上了；而且 DrawCall 合并不见得会提高性能，有时反而会降低性能。因此需要有策略地控制打包粒度。

5)界面的延迟加载和定时卸载策略

如果一些界面的重要性较低，并且不常被使用，可以等到界面需要打开显示的时候才从 bundle 加载资源，并且在界面关闭时将其卸载出内存，或者等一段时间再卸载。不过这个方法有两个代价：一是会影响体验，玩家要求打开界面时，界面的显示会有延迟；二是更容易出 bug，上层写逻辑时要考虑异步情况，当程序员要访问一个界面时，这个界面未必会在内存里。

3. 纹理资源-Texture

1) 纹理格式

在我们主动选择压缩格式之前，Unity 本身会对图片做一些处理，无论放入的是 PNG、JPG、PSD 或者 TGA，Unity 都会帮我们调整为 Texture 2D，这是一种简单的调度策略。

那么既然 Unity 本身都已经智能转换好了，为什么还要给予开发者选择压缩格式的选择呢，直接对 Texture 2D 封装好压缩方法不就可以了吗？

其实 Unity 相对于其他开发引擎有一个很明显的优势，就是多适配性，那么为了实现这种多适配性，对于单一平台的针对性就会相对减弱，而不同平台的性能表现又不尽相同，就需要开发者来根据平台特点来选择针对游戏平台的压缩格式。

同样，即使在同一平台，也会根据不同的情况有着不同的压缩需求，比如有一些关键的主页面图片，玩家感知强的地方，就对图片的质量要求高些，一些边角的辅助图片，可能要

求就低一些，如果都使用高质量压缩，性能方面就造成了浪费，但若都是用低质量压缩，质量又跟不上。所以，同样需要根据实际需求选择不同的压缩格式。

纹理格式设置不合理通常是造成纹理资源占据较大内存的主要原因之一。即便是对于很多已经建立过美术资源标准并统一修改过纹理格式的项目而言，仍然很容易统计到存在大量的 RGBA32、ARGB32、RGBA Half、RGB24 等格式的纹理资源。这些格式的纹理不但内存占用较大，还会导致游戏包体较大、加载这些资源的耗时较高、纹理带宽较高等等问题。

出现这类问题的原因主要有以下几种：存在一些“漏网之鱼”，比如美术命名不规范导致没有被回调函数修改，或者是代码中创建的资源没有设置其纹理格式；硬件或纹理资源本身不支持目标格式纹理，导致被解析为未压缩格式的纹理。

对于前一种情况，在资源列表中发现有问题的资源后，需要回到项目中自行排查修改；对于后一种情况，推荐的硬件支持的纹理格式主要有 ASTC 和 ETC2。

其中 ETC2 格式需要对应的纹理分辨率为 4 的倍数，在对应的纹理开启了 Mipmap 时更是严格要求其分辨率为 2 的次幂。否则，该纹理将被解析成未压缩格式。

2)分辨率

纹理资源的分辨率（即资源列表中的长度和宽度参数）同样也是造成内存占用过大的主要原因。一般来说，分辨率越高，其内存占用则越大。（占用的内存量是与其分辨率的平方成正比的）也就是说，分辨率变为原来的二倍，那么内存量就需要消耗四倍。因此，从资源优化的角度来讲，需要对其进行一定的限制，简单的来说，一个 Button 的纹理通常低于 128*128，如果使用 1024*1024 规格的大小就造成了性能的浪费。

其中最为需要关注的是占据较大分辨率（一般为≥1024）的纹理。对于移动平台来说，过于精细的表现通过玩家的肉眼很难分辨出差异，而过大的分辨率往往意味着不必要的浪费。

在不同档位的机型上使用不同分辨率大小的纹理资源是非常实用且易操作的分级策略。这一点即便对于图集纹理也同样适用，特别地，Unity 针对 SpriteAltast 提供了 Variant 功能，可以快捷的复制一份原图集并根据 Scale 参数降低该变体图集的分辨率，以供较低的分级使用。

3)Read/Write Enable

上文提到过，纹理资源的内存占用是计算在 GFX 内存中的，也就是传向 GPU 端的部分。而开启 Read/Write Enabled 选项的纹理资源还会保留一份内存在 CPU 端，从而造成该资源内存占用翻倍。

实际上，不需要在运行时进行修改的资源是不需要开启 Read/Write Enabled 选项的，开发者应排查并关闭不必要的设置从而降低内存开销。

4)Mipmap

Mip Map 类似于模型的 LOD，同样是一种基于渲染距离改变渲染贴图精度的技术。其优势是在物体距离渲染距离比较远时，可以节省性能。但是使用 Mip Map 时会增大内存占用量。

在我们使用 Mip map 时，假设大小为 256*256，并且会生成 8 张不同精度的图片。根据 2 的幂次方进行递减计算每张贴图大小并累加。当一张 256*256 纹理开启 Mipmap 时，它的内存占用会上升为原始数据的 1.33 倍。对于 3D 对象，比如场景中的地形、物件或人物，其纹理的 Mipmap 功能是建议开启的，可以在运行时降低带宽。但值得注意的是，在真人真机测试报告中的 Mipmap 页面中，统计了游戏过程中开启 Mipmap 纹理的各个 Mipmap 通道的屏占比变化趋势。如果场景中的 3D 物体大面积地使用 1/2 乃至 1/4、1/8 的 Mipmap 通道，说明该 3D 物体使用的纹理分辨率偏高，存在浪费现象。可以改用更低分辨率的纹理。

虽然 Mip map 本身是一种性能优化的技术，但是在 2D 精灵或者 UI 元素这些不会改变

渲染精度的纹理上，只会占用多余的内存，所以在 2D 精灵或者 UI 元素上使用纹理时记得不要勾选 Mip map。

5)各向异性与三线性过滤

开启纹理的各向异性滤波有利于地面等物体的显示效果，但会导致 GPU 宣染带宽上升。其中的原理是，纹理压缩采样时会去读缓存里面的信息，如果没读到就会往离 GPU 更远的地方去读 System Memory，因此所花的时间周期也就会增多。当开启各向异性导致采样点增多的时候，发生 Cache Miss 的概率就会变大，从而导致带宽上升的更多。在引擎中可以通过脚本关闭纹理资源的各向异性；或者对于需要开启各向异性的纹理，引擎中可以设置其采样次数为 1-16，也建议尽量设为较低的值。

将纹理设置为三线性过滤，纹理会在不同的 Mipmap 通道之间进行模糊，相比双线性过滤 GPU 渲染带宽将会上升。三线性插值采 8 个采样点（双线性采 4 个采样点），同样会使 Cache Miss 的概率变大，从而导致带宽上升，应尽量避免使用三线性过滤。

6)图集制作

图集的打包主要是优化 UI 图形渲染过程中 Draw call 的数量，其基本原理也是通过 UI 元素合批来减少 Draw Call，进入提升 CPU 的性能表现。

图集制作不够科学也是项目中常会发生的问题。资源列表中有时会出现数量峰值较高的图集纹理，但不一定是冗余。一种情况是，大量小图被打包到同一图集中，导致该图集纹理资源设置的最大分辨率（比如 2048＊2048）一张装不下这么多小图，该资源就会生成更多的纹理分页来打包这些小图。因此，只要游戏过程中依赖某一张纹理分页中的某一张小图，就会将该资源、也即该资源下所有的分页都全部加载进内存中，从而造成不必要的浪费。所以一般建议控制到 2-3 张分页以内较为合理。

即便不出现上述这个较为极端的现象，很多项目中也会出现“牵一发而动全身”的现象。即明明只用图集中的一张或几张小图，却将内存占用颇大的整个纹理都加载进了内存。

为此，在制作打包图集时，严格按照小图的使用场景、分类进行打包是非常重要的策略。选用合适的分辨率从而避免纹理没有被填充满而导致浪费，也是开发者重要注意的点。

7)使用 Text Mesh Pro 的情况

TextMeshPro 能为 UI 组件提供更好的表现和便利的功能，使得其受到不少开发者的青睐。但使用 TMP 而产生的 TMP 字体图集纹理（名称中带有 SDFAtlas，格式为 Alpha8 的纹理）也有一些坑值得注意

① 有时，结合字体资源列表注意到内存中还存在 TMP 图集纹理对应的.ttf 字体文件。说明该 TMP 字体图集为动态字体。可以考虑在项目开发结束、确保游戏要用到的字符都已添加到动态字体的 Altas 纹理中后，将动态 TMP 重新设置为静态 TMP，并且解除对.ttf 文件的依赖。这样一来，对应的字体资源将不会出现在内存中。不过，如果这种字体还被用作用户输入，则不建议采用此方法。

② Atlas 字体纹理的分辨率较大。此时建议在引擎中排查字符有没有填满图集纹理，纹理的制作生成是否合理。对于动态 TMP，如果没有填满，如只占据了纹理的 3/4 不到，则可以考虑开启 Multi Atlas Textures 选项，并设置纹理大小，举例而言就可以使 1 张 4096＊4096 的纹理变为 3 张 2048＊2048 的纹理，节省 32MB-3＊8MB＝8MB 的空间。

③ 资源列表中有 TMP 相关的资源（LiberationSans SDF Atlas.、EmojiOne），它们都是 TMP 的默认设置，可以在 Project Settings-TextMesh ProSettings 中解除对这些默认资源的依赖，就不会出现在内存中了。由于 Multi Atlas Textures 是动态 TMP 的选项，所以①、②无法同时使用，可以根据项目实情酌情选用。

4. 网格资源-Mesh

1) 顶点(Vertex)和面片数

顶点和三角形面片数过多的网格资源不仅会造成较高的内存占用，同时也不利于裁剪，容易增加渲染面数，在渲染时对 GPU 和 CPU 造成压力。针对这些网格，一方面可以简化网格，减少顶点数和面数，制作低模版本，供中低端机型分级使用；而另一方面针对单个顶点数过高的静态网格，比如一些复杂的地形和建筑，可以考虑拆分成若干个重复的小网格重新拼接。只要做好合批操作，就能以付出一点 Culling 计算耗时为代价，减少同屏渲染面片数。

2)顶点属性

如果没有统一美术资源标准且在导入时没有进行处理，则项目中的网格很有可能包含大量“多余”的顶点数据。这里的“多余”数据是指网格数据中包含了渲染时 Shader 中所不需要的数据。举例而言，如果网格数据中含有 Position、UV、Normal、Color、Tangents 等顶点数据，但其渲染所用的 Shader 中仅需要 Position、UV 和 Normal，则网格数据中的 Color 和 Tangent！则为“多余”数据，从而造成不必要的内存浪费。其中，一个小网格资源带有顶点属性，会使所在的 Combined Mesh 也带有顶点属性，需要予以注意。

针对这个问题，一个比较简单的方法是，尝试开启 " Optimize Mesh Data”选项。该选项位于 Player Setting 的 Other Settings 中。勾选后，引擎会在发布时遍历所有的网格数据，将其“多余”数据进行去除，从而降低其数据量大小。但是，需要注意的是，对于在 Runtime 情况下有修改 Material 需求的网格，建议研发团队对其进行额外的注意。如果 Runtime 时需要为某一个 GameObject 修改更为复杂、需要访问更多顶点属性的 Material，则建议先将这些 Material 挂载在相应的 Prefab 上再进行发布，以免引擎去除 Runtime 中会进行使用的网格数据。

3)Read/Write Enable

在资源列表中，常常统计到大量顶点属性不显示为-1（或“_”）的网格资源。此时，顶点属性不显示为-1，且会使得网格占用内存上升。一般而言，不需要在 CPU 端进行修改的网格是不需要开启 Read/Write 的。可以在编辑器中通过 APl 修改这些网格的 Read/Write 属性，或者对于 FBX 中的网格可以直接在 Inspector 窗口中修改。

5. 动画资源-Animation

一般来说，内存占用大于 200KB，且时长较短的动画资源就可以被认为是内存占用偏大的动画资源，有一定的优化空间。

针对动画资源的优化方法有：

1) 将 Animation Type 改成 Generic

相比另一种 Legacy 类型，Generic 实际上使用了 Unity 新版的 Mecanim 动画系统，整体性能要好很多，一般不建议使用老版的动画系统，而第三种 Humanoid 同样是新版动画系统提供给人性角色的特殊工作流，具有灵活复用性的优点，但对模型的骨骼数量有要求（即人形骨骼），可以根据项目需要选用。

2)将 Anim.Compression 改成 Optimal

Optimal 实际上就是让 Unity 在数个算法中自动选择最优的曲线表达方式，从而占用最小的存储空间。而 Keyframe Reduction 则是一个相对稳定保守的算法，对动画的表现效果产生影响的概率更小。

3)关闭 Resample Curves 选项

官方文档中称开启该选项会有一定的性能提升，但事实上根据《自动化规范 Unity 资源的实践》中的说法，上文提到的开启 Resample Curves 的性能提升体现在播放时而非加载时、

且效果微乎其微；反倒是还可能造成错误的动画表现。所以结合实验数据，大部分情况下，这个选项是建议关闭的。

4)考虑使用 API 剔除动画资源的 Scale 曲线和压缩动画的精度

其中，压缩动画精度的做法可以参考《Unity 动画文件优化探究》。

以上四种方法都可以有效降低动画资源的内存占用，但（2）、（4）两种理论上会造成动画精度的损失，但不一定会看得出来。建议研发团队自行调试，在确保动画表现不受影响的情况下尽量优化其内存占用。

6. 音频资源-Audio

对于时长较长的BGM和一些常规的时长较短但内存大的音频资源，有一定的优化空间。针对音频资源的优化方法有：

1) 开启 Force To Mono

开启音频资源的 Force To Mono 会使音频被自动混合为单声道，而并非丢失一个声道，从而在对表现效果影响较小的前提下大幅降低音频内存。

2)修改其加载方式

修改其加载方式（Load Type）为 Compressed In Memory 或 Streaming。Compressed In Memory 适用于大部分常规音频，而 Streaming 则适合时常较长且内存占用大的背景音乐。

3)修改压缩格式

对于 Compressed In Memory 的音频，修改其压缩格式（Compression Format）为压缩率更大的格式，如 Vorbis、MP3；

4)降低音频质量参数

对于 Vorbis、MP3 压缩格式的音频，还可以继续调低其 Quality 参数，进一步压缩其内存。

以上方式都可以有效减少音频资源内存（其中 Streaming 可以稳定降至 2OOKB 左右），但会造成一定的耗时代价或音质降低，可以酌情选用。

7. 材质资源-Material

材质资源本身内存占用较小，我们一般更加关注如何优化其数量，因为它的数量过多会影响之后会提到的 Resource.UnloadUnusedAssets API 的耗时。

材质资源数量过多，往往主要是因为 Instance 类型的冗余 Material 资源过多。一般来说，该种情况的出现是因为通过代码访问并修改了 meshrender.material 的参数，因此 Unity 引擎会实例一份新的 Material 来达到效果，进而造成内存上的冗余。对此，建议通过 MaterialPropertyBlock 的方式来进行优化，具体相关操作和例子见如下文章《使用 MaterialPropertyBlocka 来替换 Material 属性操作》。不过这种方法在 URP 下不适用，会打断 SRP Batcher。除此之外，则需要关注和优化非 Instancel 的材质资源的疑似冗余现象，不再赘述。

除了数量上的问题外，材质资源往往还涉及到一些纹理采样和 Shader 使用相关的问题，导致一些额外的内存和 GPU 性能浪费。

对于使用纯色纹理采样的材质，可以将纹理采样替换为一个颜色参数，从而节省一张纹理采样的开销；而对于空纹理采样的材质，Unity 会采样内置提供的纹理，但是计算得到的颜色是一个常数，仍然属于浪费；又对于包含无用纹理采样的材质，由于 Unity 的机制，材质球会自动保存其上的纹理采样，即使更换 Shader 也不会把原来依赖的纹理去除，所以可能会造成误依赖实际不需要的纹理带进包体的情况，从而造成内存的浪费。

8. Render Texture

1) 渲染分辨率

一些 RT 资源能反映项目当前的渲染分辨率。对于 GPU 和渲染模块压力较大的项目，在中低端机型上降低其渲染分辨率是非常直观有效的分级策略。一般低端机型上可以考虑不采用真机分辨率，降到 0.8-0.9 倍，甚至很多团队会选择 0.7 倍或 720P

如果一些其他的 RT 资源分辨率过高也应引起注意，尤其是 2048＊2048 以上的资源。应当排查是否有必要用到如此精细的 RT，在低端机上考虑采用更低分辨率的效果。

2)抗锯齿

开启多倍 AA 会使 RT 占用内存成倍上升，并对 GPU 造成压力。建议排查是否有必要开启 AA，尤其是在中低端机上，可以考虑关闭此效果。

3)后处理

一些常见的后处理相关的 RT（如 Bloom、Blur）是从 1/2 渲染分辨率开始采样，可以考虑改从 1/4 开始采样、并减少下采样次数，从而节省内存并降低后处理对渲染的压力。

4)URP 下的 RT

使用 URP 时，内存中会多出 CameraColorTexture 和 CameraDepthAttachment 两份 RT 资源作为渲染目标，而开启 URP 相机的 CopyDepth 和 CopyColor 设置时会额外产生 CameraDepthTexture 和 CameraOpaqueTexture 作为中间 RT。出现这两种 RT 时，需要排查确实是否用到 CopyDepth 和 CopyColor，否则应予以关闭以避免不必要的浪费。

9. Shader 资源

1) ShaderLab

Unity2019.4.20 是 Shader 内存统计方法的一个转折点。在此之前，Shader 的内存主要统计在 ShaderLab 中，而之后则主要统计在 Shader 资源自身身上。

对于 Unity2019.4.20 之前的版本的项目，查看 ShaderLab 的内存需要在 Unity Profiler 中 TakeSample。无论是 Shader 资源本体还是 ShaderLab 内存占用过高，都要着手于控制 Shader 的数量和变体数量。

2)变体数

变体数过多是造成一个 Shader 资源内存占用过大、占用包体过大的主要原因。在项目迭代中可能会出现已经被弃用或者没有被实际使用到的关键字，导致变体成倍上升；又或者 Shader 写的比较复杂，其中一些关键字组合永远不会被用到，从而导致很多变体是多余的。

针对上述情况，Unity 提供了回调函数，在项目打 AssetBundle 包或者 Build 时先剔除用不到的关键字或关键字组合相关的变体。剔除 Shader 变体的方法可以参考《Stripping scriptable shader variants》。

3)冗余

Shader 冗余尤其需要予以关注，Shader 的冗余不光导致内存上升，还可能造成重复解析，即运行时不必要的 Shader.Parse 和 Shader.CreateGPUProgram API 调用耗时。

4)Standard Shader

在资源列表中发现 Standard、ParticleSystem/Standard Unlit。这两种 Shader 变体数量多，其加载耗时会非常高，内存占用也偏大，不建议直接在项目中使用。出现的原因一般是导入的 FBX 模型中或者 Unity 自身生成的一些 3D 对象使用了自带的 Default Material，从而依赖了 Standard Shader，建议予以排查精简。

如果确实要使用 Standard Shader 或 ParticleSystem/Standard Unlit，应考虑自己重写一个

Shader 并只包含自己需要用到的变体。

10. 字体资源

若单个字体资源内存占用超过 10MB，可以认为该字体资源内存偏大。可以考虑使用 FontPruner 字体精简工具或其他字体精简工具，对字体进行瘦身，减小内存占用。

我们也需要关注项目中字体数量过多的情况，因为每个 Font 会对应一个 Font Texture 字体纹理，所以字体资源数量多了，Font Texture 的数量也多了，从而占用较多内存。

11. 内存管理和 GC 优化

1) 托管堆基础知识

a. Unity 游戏运行时内存占用分以下几部分

Mono 堆：C# 代码
Native 堆：资源，Unity 引擎逻辑，第三方逻辑。
库代码：Unity 库，第三方库。
Unity 的 GC（即 Mono Runtime GC）
C#的 GC（CLR GC）
Lua GC

b. Mono 堆

代码分配的内存，是通过 Mono 虚拟机，分配在 Mono 堆内存上的，其内存占用量一般较小，主要目的是程序员在处理程序逻辑时使用；比如：通过 System 命名空间中的接口分配的内存，将会通过 Mono Runtime 分配在 Mono 堆。

c. Native 堆

而 Unity 的资源，是通过 Unity 的 C++层，分配在 Native 堆内存上的那部分内存。
比如通过 UnityEngine 命名空间中的接口分配的内存，将会通过 Unity 分配在 Native 堆。

d. GC 和堆内存联系

对于目前绝大多数基于 Unity 引擎开发的项目而言，其托管堆内存是由 Mono 分配和管理的。“托管”的本意是 Mono 可以自动地改变堆的大小来适应你所需要的内存，并且适时地调用 GC 操作来释放已经不需要的内存，但是 GC 并不是实时管理的，是需要通过程序员手动或系统定时触发的。因为 GC 是一个耗时的操作，可能在有些系统中触发的不合时宜（明显卡顿）。所以，GC 也需要优化，需要控制在合事宜的情况触发。比如游戏中我们需要在切换 Loading 时触发 GC，而在游戏战斗中控制不能被触发。

因此优化 GC，就是优化堆内存，就是尽量减少堆内存，及时回收堆内存。

2)垃圾回收何时触发

a. 被动触发

GC 会不时地自动运行（频率因平台而异）。
堆分配时，堆上的可用内存不足时会触发 GC。

b. 主动触发

Mono GC 主动调用的接口是 Resources.UnloadUnusedAssets()。
C# GC 也提供了同样的接口 GC.Collect()。
LuaGC 主动调用 collectgarbage("collect"):做一次完整的垃圾收集循环。
有一点需要说明的是，Resources.UnloadUnusedAssets()内部本身就会调用 GC.Collect()。
Unity 还提供了另外一个更加暴力的方式——Resources.UnloadAssets()来卸载资源，但是这个接口无论资源是不是“垃圾”，都会直接删除，是一个很危险的接口，建议确定资源不使用的情况下，再调用该接口。

12. 合批优化

动态合批与静态合批其本质是将多次绘制请求，在允许的条件下进行合并处理，减少 CPU 对 GPU 绘制请求的次数，达到提高性能的目的。

1) 啥是合批

批量渲染其实是个老生常谈的话题，它的另一个名字叫做“合批”。
在日常开发中，通常说到优化、提高帧率时，总是会提到它。

2)为啥要合批

批量渲染是通过减少 CPU 向 GPU 发送渲染命令（DrawCall）的次数，以及减少 GPU 切换渲染状态的次数，尽量让 GPU 一次多做一些事情，来提升逻辑线和渲染线的整体效率。
但这是建立在 GPU 相对空闲，而 CPU 把更多的时间都耗费在渲染命令的提交上时，才有意义。

3)调用 Draw Call 性能消耗原因是啥

我们的应用中每一次渲染，进行的 API 调用都会经过 Application->Runtime->Driver(驱动)->显卡(GPU)，其中每一步都会有一定的耗时。
每调用一次渲染 API 并不是直接经过以上说的所有组件，去通知 GPU 执行我们的调用。
Runtime 会将所有的 API 调用先转换为设备无关的“命令”（之所以是设备无关的，主要是因为这样我们写的程序就可以运行在任何特性兼容的硬件上了。运行时库使不同的硬件架构相对我们变的透明。）
Draw Call 性能消耗原因是命令从 Runtime 到 Driver 的过程中，CPU 要发生从用户模式到内核模式的切换。
模式切换对于 CPU 来说是一件非常耗时的工作，所以如果所有的 API 调用 Runtime 都直接发送渲染命令给 Driver，那就会导致每次 API 调用都发生 CPU 模式切换，这个性能消耗是非常大的。
Runtime 中的 Command Buffer 可以将一些没有必要马上发送给 Driver 的命令缓冲起来，在适当的时机一起发送给 Driver，进而在显卡执行。以这样的方式来寻求最少的 CPU 模式切换，提升效率。

4)合批优化的是 CPU 还是 GPU

合批只是对 CPU 的优化，与 GPU 没有任何关系。

合批是节省了 CPU 的相关准备工作的工作量。

合批后，经过 VS,PS，深度测试，模板测试后，此时已没有了纹理，顶点，索引的概念，只剩下一个个孤立的像素，各像素间没有任何关系了。像素送到 GPU 后进行批量处理，呈现到屏幕硬件上。

因此合批与 GPU 没有任何关系，也几乎没有影响。不管是一批还是多批，最终在此帧送到 GPU 的像素数量是相等的，数据是相同的。合批与否，对 GPU 的影响仅是像素到达的慢了还是快了，几乎不影响 GPU 的性能。

目前比较新的图形 API 如：DirectX12、Vulkan、Metal，在 Driver 上的性能消耗已经降低了很多。但是通过合批来降低渲染的 Draw call 仍然是十分必要的。这也是我们对 Draw call 优化唯一能够做的事情。接下来我们介绍一下几种常见的合批处理。

5)合批技术分离线合批和实时合批

a. 离线合批

离线合批就是在游戏运行前，先用工具把相关资源做合批处理，以减轻引擎实时合批的负担。

适合离线合批的是静态模型和场景物件。如场景地表装饰面：石头/砖块等等。

离线合批方式有：

美术利用专业建模工具合批。如 3D Max/Maya 等。

利用引擎插件或工具。如 Unity 的插件 MeshBaker 和 DrawCallMinimizer，可以将静态物体进行合批。

自制离线合批工具。如果第三方插件无法满足项目需求，就要程序专门实现离线合批工具。

b. 实时合批—Runtime Batch

Unity 引擎内建了两种合批渲染技术：Static batching（静态合批）和 Dynamic batching（动态合批）。

静态合批是勾选 Static，Unity 在 Build 的时候，会自动去生成合并的网格，并将合并后的数据以文件形式存储。这样当场景被加载时，一次性提交整个合并模型的顶点数据，根据引擎的场景管理系统判断各个子模型的可见性。然后设置一次渲染状态，调用多次 Draw call 分别绘制每一个子模型。

静态合批必须共享相同的材质，运行时不能移动，旋转或缩放。

优点：

静态合批采用了以空间换时间的策略来提升渲染效率。

静态合批并不减少 Draw call 的数量（但是在编辑器时由于计算方法区别 Draw call 数量是会显示减少了的），但是由于我们预先把所有的子模型的顶点变换到了世界空间下，并且这些子模型共享材质，所以在多次 Draw call 调用之间并没有渲染状态的切换，渲染 API 会缓存绘制命令，起到了渲染优化的目的。另外，在运行时所有的顶点位置处理不再需要进行计算，节约了计算资源。

缺点：

打包之后体积增大，应用运行时所占用的内存体积也会增大。

需要额外的内存来存储合并的几何体。

注意如果多个 GameObject 在静态批处理之前共享相同的几何体，则会在编辑器或运行时为每个 GameObject 创建几何体的副本，这会增大内存的开销。例如，在密集的森林将树标记为静态可能会产生严重的内存影响。

静态合批在大多数平台上的限制是 64k 顶点和 64k 索引。

动态合批是专门为优化场景中共享同一材质的动态 GameObject 的渲染设计的。目标是以最小的代价合并小型网格模型，减少 Drawcall。

动态合批的原理也很简单，在进行场景绘制之前将所有的共享同一材质的模型的顶点信息变换到世界空间中，然后通过一次 Draw call 绘制多个模型，达到合批的目的。模型顶点变换的操作是由 CPU 完成的，所以这会带来一些 CPU 的性能消耗。

限制：

900 个顶点以下的模型。

如果我们使用了顶点坐标，法线，UV，那么就只能最多 300 个顶点。

如果我们使用了 UV0，UV1，和切线，又更少了，只能最多 150 个顶点。

如果两个模型缩放大小不同，不能被合批的，即模型之间的缩放必须一致。

合并网格的材质球的实例必须相同。即材质球属性不能被区分对待，材质球对象实例必须是同一个。

如果他们有 Lightmap 数据，必须相同的才有机会合批。

使用多个 pass 的 Shader 是绝对不会被合批。因为 Multi-pass Shader 通常会导致一个物体要连续绘制多次，并切换渲染状态。这会打破其跟其他物体进行 Dynamic batching 的机会。

延迟渲染是无法被合批。

6)GPU Instancing

GPU Instancing 没有动态合批那样对网格数量的限制，也没有静态网格那样需要这么大的内存，它很好的弥补了这两者的缺陷，但也有存在着一些限制，我们下面来逐一阐述。

与动态和静态合批不同的是，GPU Instancing 并不通过对网格的合并操作来减少 Drawcall，GPU Instancing 的处理过程是只提交一个模型网格让 GPU 绘制很多个地方，这些不同地方绘制的网格可以对缩放大小，旋转角度和坐标有不一样的操作，材质球虽然相同但材质球属性可以各自有各自的区别。

从图形调用接口上来说 GPU Instancing 调用的是 OpenGL 和 DirectX 里的多实例渲染接口。我们拿 OpenGL 来说：

PS：

第一个是无索引的顶点网格集多实例渲染，

void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, Glsizei primCount);

第二个是索引网格的多实例渲染，

void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const void* indices, GLsizei primCount);

第三个是索引基于偏移的网格多实例渲染。

void glDrawElementsInstancedBaseVertex(GLenum mode, GLsizei count, GLenum type, const void* indices, GLsizei instanceCount, GLuint baseVertex);

这三个接口都会向 GPU 传入渲染数据并开启渲染，与平时渲染多次要多次执行整个渲染管线不同的是，这三个接口会分别将模型渲染多次，并且是在同一个渲染管线中。

如果只是一个坐标上渲染多次模型是没有意义的，我们需要将一个模型渲染到不同的多个地方，并且需要有不同的缩放大小和旋转角度，以及不同的材质球参数，这才是我们真正

需要的。

GPU Instancing 正为我们提供这个功能，上面三个渲染接口告知 Shader 着色器开启一个叫 InstancingID 的变量，这个变量可以确定当前着色计算的是第几个实例。

有了这个 InstancingID 就能使得我们在多实例渲染中，辨识当前渲染的模型到底使用哪个属性参数。Shader 的顶点着色器和片元着色器可以通过这个变量来获取模型矩阵、颜色等不同变化的参数。

7)SRP Batcher

SRP Batcher 并不会减少 DrawCall，减少的是 DrawCall 之间的 GPU 设置的工作量。因为 Unity 在发出 DrawCall 之前必须进行很多设置。实际的 CPU 成本便来自这些设置，而不是来自 GPU DrawCall 本身（DrawCall 只是 Unity 需要推送到 GPU 命令缓冲区的少量字节）。

SRP Batcher 通过批处理一系列 Bind 和 Draw GPU 命令来减少 DrawCall 之间的 GPU 设置的工作量。也就是之前一堆绑定和绘制的 GPU 命令，我们可以使用批处理减少绘制调用之间的 GPU 设置。

二. 以引擎模块为划分的 CPU 耗时调优

1. 耗时瓶颈

当一个项目由于 CPU 端性能瓶颈而产生帧率偏低、卡顿明显的现象时，如何提炼出哪个模块的哪个问题是造成性能瓶颈的主要问题就成了关键。尽管我们已经对引擎中主要模块做了整理，各个模块间会出现的问题还是会千奇百怪不可一以概之，而且它们对 CPU 性能压力的贡献也不尽相同。那么我们就需要对什么样的耗时可以认为是潜在的性能瓶颈有准确的认知。

在移动端项目中，我们 CPU 端性能优化的目标是能够在中低端机型上大部分时间跑满 30 帧的流畅游戏过程。为了达成这一目标，简单做一下除法就得到我们的 CPU 耗时均值应控制在 33Ms 以下。当然，这并不意味着 CPU 均值已经在 33ms 以下的项目就已经把 CPU 耗时控制的很好了。游戏运行过程中性能压力点是不同的，可能一系列界面中压力很小、但反过来游戏中最重要的战斗场景中帧率很低、又或者是存在大量几百毫秒甚至几秒的卡顿，而最终平均下来仍然低于 33ms。

在一次测试中，当 33s 及以上耗时的帧数占总顿数的 10%以下时，可以认为项目 CPU 性能整体控制在正常范围内。而这个占比越高，说明当前项目的 CPU 性能瓶颈越严重。

以上的讨论内容主要是围绕着我们对 CPU 性能的宏观的优化目标，和内存一样，我们仍要结合具体模块的具体数据来排查和解决项目中实际存在的问题。

2. 渲染模块

1) 多线程渲染

一般情况下，在单线程渲染的流程中，在游戏每一帧运行过程中，主线程（CPU1）先执行 Update，在这里做大量的逻辑更新，例如游戏 Al、碰撞检测和动画更新等；然后执行 Render，在这里做渲染相关的指令调用。在渲染时，主线程需要调用图形 API 更新渲染状态，例如设置 Shader、纹理、矩阵和 Alphal 融合等，然后再执行 DrawCall，所有的这些图形 API 调用都是与驱动层交互的，而驱动层维护着所有的渲染状态，这些 API 的调用有可能会触发驱动层的渲染状态改变，从而发生卡顿。由于驱动层的状态对于上层调用是透明的，因此卡顿是否会发生以及卡顿发生的时间长短对于 API 的调用者（CPU1）来说都是未知的。而此时其它 CPU 有可能处于空闲等待的状态，从而造成浪费。因此可以将渲染部分抽离出来，放到其它的 CPU 中，形成单独的渲染线程，与逻辑线程同时进行，以减少主线程卡顿。

其大致的实现流程是，在主线程中调用的图形 API 被封装成命令，提交到渲染队列，

这样就可以节省在主线程中调用图形 API 的开销，从而提高帧率；渲染线程从渲染队列获取渲染指令并执行调用图形 API 与驱动层交互，这部分交互耗时从主线程转到渲染线程。

而 Unity 在 Project Settings 中支持且默认开启了 Multithreaded Rendering，一般建议保持开启。开启多线程渲染时，CPU 等待 GPU 完成工作的耗时会被统计 Gfx.WaitForPresent 函数中，而关闭多线程渲染时这一部分耗时则被主要统计到 Graphics.PresentAndSync 中。所以，项目中是否统计到 Gfx.WaitForPresent 函数耗时是判断是否开启了多线程渲染的一个依据。特别地，在项目开发和测试阶段可以考虑暂时性地关闭多线程渲染并打包测试，从而更直观地反映出渲染模块存在的性能瓶颈。

对于正常开启了多线程渲染的项目，Gfx.WaitForPresent 的耗时走向也有相当的参考意义。测试中局部的 GPU 压力越大，CPU 等待 GPU 完成工作的时间也就越长，Gfx.WaitForPresent 的耗时也就越高。所以，当 Gfx.WaitForPresent 存在数十甚至上百毫秒地持续耗时时，说明对应场景的 GPU 压力较大。

2)同屏渲染面片数

影响渲染效率的两个最基本的参数无疑就是 Triangle 和 DrawCall。

通常情况下，Triangle 面片数和 GPU 渲染耗时是成正比的，而对于大部分项目来说，不透明 Triangle 数量又往往远比半透明 Triangle 要多，尤其需要关注。建议在低端机型上将同屏渲染面片数控制在 25 万面以内，即便是高端机也不建议超过 60 万面。当使用工具发现局部同屏渲染面片数过高后，可以结合 Frame Debugger 对重点帧的渲染物体进行排查。

常见的优化方案是，在制作上需要严格控制网格资源的面片数，尤其是一些角色和地形的模型，应严格警惕数万面及以上的网格；另外，一个很好的方法是通过工具减少场景中的面片数——比如在低端机上使用低模、减少场景中相对不重要的小物件的展示——进而降低渲染的开销。

需要指出的是，所关注和统计的面片数量并不是当前帧场景模型的面片数，而是当前帧所渲染的面片数，其数值不仅与模型面片数有关，也和渲染次数相关，更加直观地反映出同屏渲染面片数造成的渲染压力。例如：场景中的网格模型面片数为 1 万，而其使用的 Shader 拥有 2 个渲染 Pass，或者有 2 个相机对其同时渲染；又或者使用了 SSAO、Reflection 等后处理效果中的一个，那么此处所显示的 Triangle 数值将为 2 万。所以，在低端机上应严格警惕这些一下就会使同屏渲染面片数加倍的操作，即便对于高端机也应做好权衡，三思而后用。

3)Batch 与 DrawCall

在 Unity 中，我们需要区分 DrawCall 和 Batch。在一个 Batch 中会存在有多个 DrawCall，出现这种情况时我们往往更关心 Batch 的数量，因为它才是把渲染数据提交给 GPU 的单位，也是我们需要优化和控制数量的真正对象。

降低 Batch 的方式通常有动态合批、静态合批、SRP Batcher 和 GPU Instancing 这四种，围绕 Batch 优化的讨论较为复杂，下面简单总结静态合批、动态合批(Dynamic Batching)、SRP Batcher 和 GPU Instancing 的合批条件和优缺点。

a. Draw Call

Unity 引擎早期，衡量 CPU 在渲染时的资源消耗大多都是通过 Draw Call 的数量。因为 CPU 在渲染流水线中的处理阶段是应用程序阶段(Application Stage)，主要是做一些数据的准备与提交工作，而 Draw Call 的数量代表了 CPU 向 GPU 提交数据的次数，Draw Call 本身只是一些数据流的字节，主要的性能消耗在于 CPU 的数据准备阶段。

b. Batcher

由于合批的出现，就不再是每一个渲染对象都会产生一个 Draw Call，所以这个时候就提出了一个新的衡量标准：Batcher。

c. Set Pass Call

前面也说过，CPU 在渲染阶段，性能消耗的峰值一般不在于 Draw Call，而往往存在于对其数据准备的阶段，因此单纯以数据的提交数量为衡量标准并不准确，同时在数据准备的过程中，假如前后两个材质发生了变化，会更大幅度的消耗性能，这也是整个 CPU 在渲染阶段最消耗性能的步骤，因此 Unity 通过 Set Pass Call 来作为性能消耗的标准。

d. 静态合批-Static Batching

条件	不同 Mesh，只要使用相同的材质球即可。
优点	节省顶点信息地绑定；节省几何信息地传递；相邻材质相同时，，节省材质地传递。
缺点	离线合并时，若合并的 Mesh 中存在重复资源，则容易使得合并后包体变大；运行时合并，则生成 Combine Mesh 的过程会造成 CPU 短时间峰值；同样的，若合并的 Mesh 中存在重复资源，则会使得合并后内存占用变大。

简单的来说，Static Batching 通过对一些小的网格进行合并备份到内存中，当执行渲染操作时，CPU 一次性将合并的内存的发送给 GPU 来减少 Draw Call 的数量，不过这样做有一定的限制：

① 对象必须是静态的，不可移动。

② 合并的对象使用相同的材质。

同时在使用 Static Batching 时需要额外的内存来存储组合的几何体，导致内存在一定程度上的浪费。简单来说，作为通过内存的上的置换可以获得时间上的高效运行，需要根据实际情况来谨慎添加渲染对象，避免获取 CPU 性能优势时产生不必要的内存问题。

而关于 Static Batching 的使用，首先需要在 Project Setting 中的 Player 选项中勾选 Static Batching。

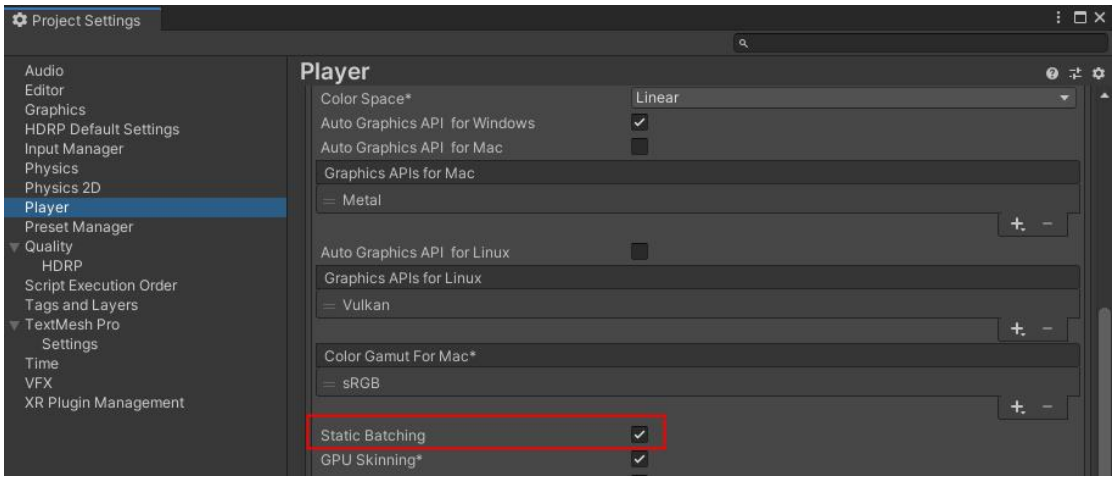

图 1. 使用静态批处理需确保开启了 Static Batching

接下来就可以在 Inspector 面板中对需要 Static Batching 的对象勾选上 Batching Static 即可。

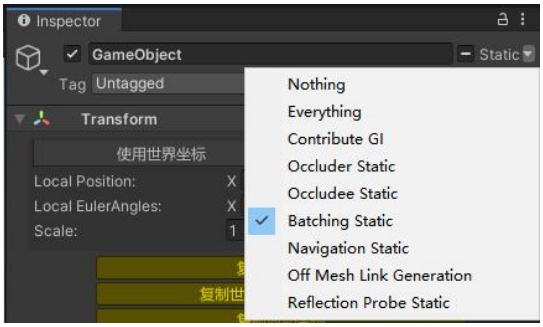

图 2.　　设置对象是否使用 Static Batching

a. Dynamic Batching

Dynamic Batching 同样是可以对有共同材质的对象进行相关的合并，但是其对象可以为动态的，而且这一过程是动态进行的，只需要在 Project Setting 中的 Player 中勾选上 Dynamic Batching 即可。注意，在 URP 模板中，这一选项移到了 URP 的配置文件中。

虽然 Dynamic Batching 的设置步骤很简单，但是其使用条件却很苛刻，需要满足一系列的限定条件，才能实现合批的效果。

批处理动态游戏对象在每个顶点都有一定开销，因此批处理仅会应用于总共包含不超过 900 个顶点属性且不超过 300 个顶点的网格。如果 Shader 使用顶点位置、法线和单个 UV，最多可以批处理 300 个顶点，而如果 Shader 使用顶点位置、法线、UV0、UV1 和切线，则只能批处理 180 个顶点。

如果游戏对象在变换中包含镜像，则不会对这些对象进行批处理（例如，具有 +1 缩放的游戏对象 A 和具有 －1 缩放的游戏对象 B 无法一起接受批处理）。即使游戏对象基本相同，使用不同的材质实例也会导致游戏对象不能一起接受批处理。例外情况是阴影投射物渲染。

带有光照贴图的游戏对象具有其他渲染器参数：光照贴图索引和光照贴图偏移/缩放。通常，动态光照贴图的游戏对象应指向要批处理的完全相同的光照贴图位置。多 pass Shader 会中断批处理。

几乎所有的 Unity Shader 都支持前向渲染中的多个光照，有效地为它们执行额外 pass。“其他逐像素光照”的绘制调用不进行批处理。 旧版延迟（光照 pre-pass）渲染路径会禁用动态批处理，因为它必须绘制两次游戏对象

看起来限制条件很多，但是简单的总结大概是模型要简单，同时使用的 Shader 一定要是单 Pass 的。同时因为单 Pass 的限定，对于延迟渲染来说，由于将光照分离到单独的 Pass 去处理而导致受光的对象完全没有办法进行动态合批的操作，所以会直接屏蔽掉 Dynamic Batching。

b. SRP Batcher

条件	不同 Mesh，只要使用相同的 Shader 且变体一样即可。
优点	节省 Uniform Buffer 的写入操作；按 Shader 分 Batch，预先生成 Uniform Buffer，Batch 内部无 CPU Write。

缺点	Constant Buffer（CBuffer）的显存固定开销；不支持 MaterialPropertyBlock。

原理

Unity 中，我们可以在一帧内的任何时间修改任何材质的 Property。但是，这种做法有一些缺点。例如，DrawCall 使用新材质时，要执行许多工作。因此，场景中的材质越多，Unity 必须用于设置 GPU 数据的 CPU 消耗也越多。解决此问题的传统方法是减少 DrawCall 的数量以优化 CPU 渲染成本，因为 Unity 在发出 DrawCall 之前必须进行很多设置。实际的 CPU 成本便来自该设置，而不是来自 GPU DrawCall 本身（DrawCall 只是 Unity 需要推送到 GPU 命令缓冲区的少量字节）

正如 Set Pass Call 的描述那样，游戏在渲染阶段对 CPU 的性能消耗主要在于材质切换阶段的一些工作，而 SRP Batcher 通过在 GPU 的数据缓冲区的持久化存储来换取 CPU 的新材质的准备时间，从而降低 CPU 的数据准备压力。

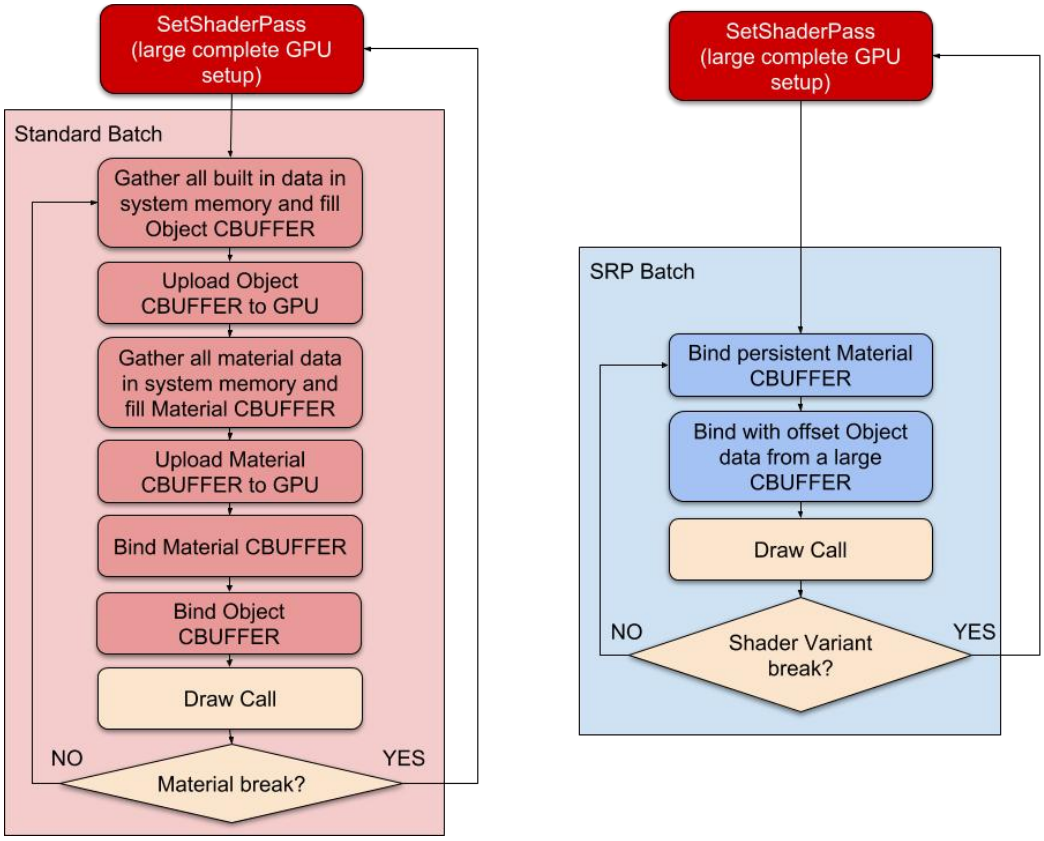

图 3. SRP 具体工作流

为了获得最佳渲染性能，每个 SRP 批处理应包含尽可能多的绑定 draw 命令。为了实现这一点，尽可能多的使用具有相同 Shader 的不同材质，但是必须尽可能少的使用 Shader 变体

当 Unity 在渲染循环中检测到新材质时，CPU 会收集所有属性并将它们绑定到常量缓冲区(CBUFFER)中的 GPU。GPU 缓冲区的数量取决于 Shader 如何声明其 CBUFFER。

SRP Batcher 是一个低级渲染循环，使材质数据持久保留在 GPU 内存中。如果材质内容不变，SRP Batcher 就不会改变渲染状态。实际上，SRP Batcher 会使用专用的代码路径来快速更新大型 GPU 缓冲区中的 Unity 引擎属性，如下所示：

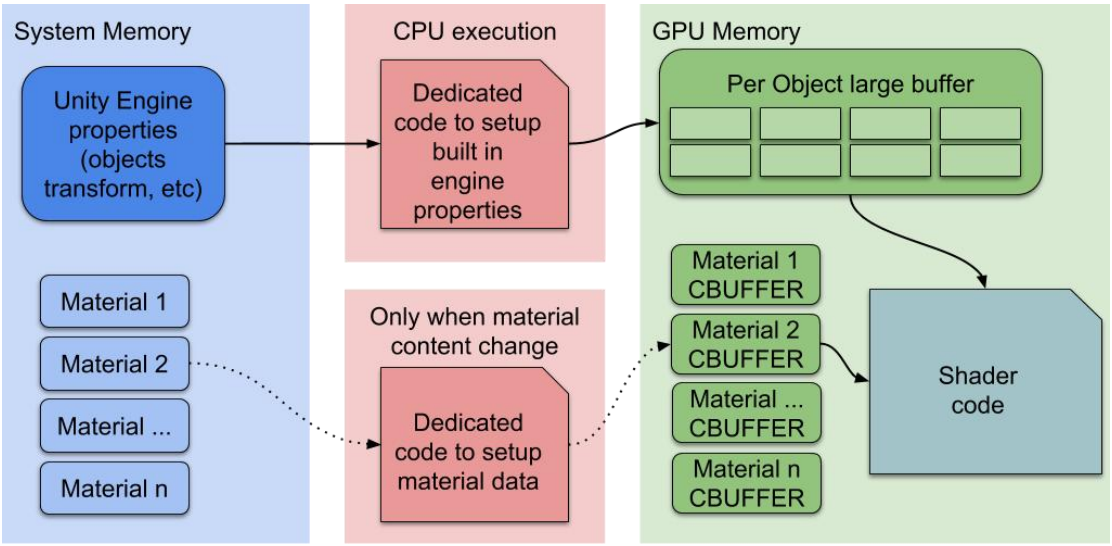

图 4.　SRP Batcher 使用专用代码路径更新大型 GPU 缓冲区中的 Unity 引擎属性

在此处，CPU 仅处理上图中标记为 Per Object large buffer 的 Unity 引擎属性。所有材质在 GPU 内存中都有持久的 CBUFFER，可供随时使用。这样会加快渲染速度，原因是：现在，所有的材质内容都持久保留在 GPU 内存中。专用代码为每个对象的所有属性管理一个大型的每个对象 GPU 常量缓冲区。

限制条件

渲染的对象必须是网格或蒙皮网格（该对象不能是粒子）。

Shader 必须与 SRP Batcher 兼容。HDRP 和 URP 中的所有 Lit Shader 和 Unlit Shader 均符合此要求（这些着色器的“粒子”版本除外）。

为了使着色器与 SRP Batcher 兼容：

必须在一个名为 UnityPerDraw 的 CBUFFER 中声明所有内置引擎属性。例如 unity_ObjectToWorld 或 unity_SHAr。

必须在一个名为 UnityPerMaterial 的 CBUFFER 中声明所有材质属性。

c. GPU Instancing

条件	相同的 Mesh，且使用相同的材质球。
优点	适用于渲染同种大量怪物的需求，合批的同时能够降低动画模块的耗时。
缺点	可能存在负优化，反而使 DrawCall 上升；Instancing 有时候被打乱，可以自己分组用 API 渲染。

使用 GPU Instancing 可使用少量 DrawCall，一次绘制（或渲染）同一网格的多个副本。它对于绘制诸如建筑物、树木和草地之类的在场景中重复出现的对象非常有用：

GPU Instancing 在每次 DrawCall 时仅渲染相同的网格，但每个实例可以具有不同的参数（例如，颜色或比例）以增加变化并减少外观上的重复。

GPU Instancing 可以降低每个场景使用的 DrawCall 数量。可以显著提高项目的渲染性能。

与其他合批手段类似，GPU Instancing 同样有一些使用限制条件。

Unity 自动选取要实例化的网格渲染器组件和 Graphics.DrawMesh 调用。请注意，不支持 SkinnedMeshRenderer。

Unity 仅在单个 GPU 实例化 DrawCall 中批量处理那些共享相同网格和相同材质的游戏对象。使用少量网格和材质可以提高实例化效率。要创建变体，请修改着色器脚本为每个实例添加数据。

可以看出与其他两种合批手段不同的是，除了材质相同之外，其主要是对于使用同一网格的物体有效，所以正如名字的 Instancing 那样，是通过 GPU 直接对于某一物体进行实例化来降低 CPU 对场景物体的数据命令准备所产生的性能消耗的技术手段

4)Shader.CreateGPUProgram

该 API 常常在渲染模块主函数的堆栈中出现，并造成渲染模块中的大多数函数峰值。它是 Shader 第一次渲染时产生的耗时，其耗时与渲染 Shader 的复杂程度相关。当它在游戏过程中被调用并且造成较高的耗时峰值时应引起注意。

对此，我们可以将 Shader 通过 ShaderVariantCollection 收集要用到的变体并进行 AssetBundle 打包。在将该 ShaderVariantCollection 资源加载进内存后，通过在游戏前期场景调用 ShaderVariantCollection.WarmUp 来触发 Shader.CreateGPUProgram，并将此 SVC 进行缓存，从而避免在游戏运行时触发此 API 的调用、避免局部的 CPU 高耗时。

然而即便是已经做过以上操作的项目也常会检测到运行时偶尔的该 API 耗时峰值，说明存在一些“漏网之鱼”。开发者可以结合 Profiler 的 Timeline 模式，选中触发调用 Shader.CreateGPUProgram 的帧来查看具体是哪些 Shader 触发了该 APl。

5)Culling

绝大多数情况下，Culling 本身耗时并不显眼，它的意义在于反映一些与渲染相关的问题。

a. 相机数量多

当渲染模块主函数的堆栈中 Culling 耗时的占比比较高（一般项目中在 10%-20%左右）。

b. 场景中小物件多

Culling 耗时与场景中的 GameObject 小物件数量的相关性比较大。这种情况建议研发团队优化场景制作方式，关注场景中是否存在过多小物件，导致 Culling 耗时增高。可以考虑采用动态加载、分块显示，或者 Culling Group、Culling Distance 等方法优化 Culling 的耗时。

c. Occlusion Culling

如果项目使用了多线程渲染且开启了 Occlusion Culling（遮挡剔除），通常会导致子线程的压力过大而使整体 Culling 过高，

由于 Occlusion Culling 需要根据场景中的物体计算遮挡关系，因此开启 Occlusion Culling 虽然降低了渲染消耗，其本身的性能开销却也是值得注意的，并不一定适用于所有场景。这种情况建议开发者选择性地关闭一部分 Occlusion Culling 去测试一下渲染数据的整体消耗进行对比，再决定是否需要开启这个功能。

d. 包围盒更新

Culling 的堆栈中有时出现的 FinalizeUpdateRendererBoundingVolumes 为包围盒更新耗时。一般常见于 Skinned Mesh 和粒子系统的包围盒更新上。如果该 API 出现很频繁，则要通过截图去排查此时是否有较大量的 Skinned Mesh 更新，或者较为复杂的粒子系统更新。

e. PostProcessingLayer.OnPreCull/WaterReflection.OnWillRenderObject

PostProcessLayer.OnPreCull 这一方法和项目中使用的 PostProcessing Stack 相关。可以在 PostProcessManager.cs 中添加静态变量 GlobalNeedUpdateSettings，在切场景的时候通过设置 PostProcessManager.GlobalNeedUpdateSettings 为 true 来 UpdateSettings.。这样就可以避免每帧都做 UpdateSettings 操作，从而减少一部分耗时。

PS：

```
internal void UpdateSettings (PostProcessLayer postProcessLayer，Camera camera)
{
    if （!GlobalNeedUpdateSettings）
        return;
        GlobalNeedUpdateSettings  = false;
```

WaterReflection.OnWillRenderObject 则是项目中使用到的水面反射效果的相关耗时，若该项耗时较高，可以关注一下实现方式上是否有可优化的空间，比如去除一些不必要的粒子、小物件等的反射渲染。

3. UI 模块

在 Unity 引擎中，主流的 UI 框架有 UGUI、NGUI 以及使用越来越多的 FairyGUI，我们主要从使用最多的 UGUI 来进行说明。

1) UGUI EventSystem.Update

EventSystem.Update 函数为 UGU 的事件系统耗时，其耗时偏高时主要关注以下两个因素：

a. 触发调用耗时高

作为 UGUI 事件系统的主函数，该函数主要是在触摸释放时触发，当本身有较高的 CPU 开销时，通常都是因为调用了其它较为耗时的函数引起。因此需要通过添加 Profiler.BeginSample/EndSample 打点来对所触发的逻辑进行进一步地检测，从而排查出具体是哪一个子函数或者代码段造成的高耗时。

b. 轮询耗时高

所有 UGUI 组件在创建时都默认开启了 Raycast Target 这一选项，实际上是为接受事件响应做好了准备。事实上，大部分比如 Image、Text 类型的 UI 组件是不会参与事件响应的，但仍然会在鼠标/手指划过或悬停时参与轮询，所以通过模拟射线检测判断 UI 组件是否被划过或悬停，造成不必要的耗时。尤其在项目中 UI 组件比较多时，关闭不参与事件响应的组件的 Raycast Target 设置，可以有效降低 EventSystem.Update()耗时。

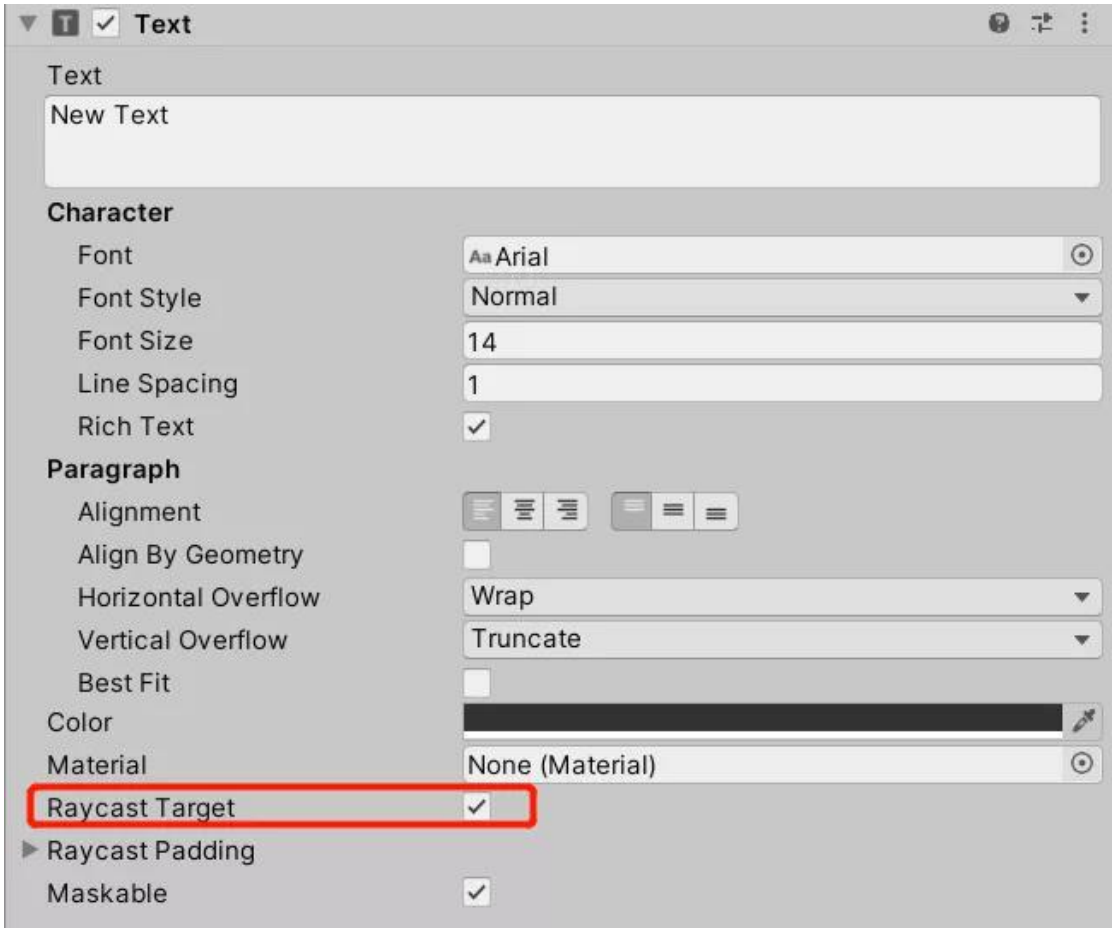

图 5.　　组件上的 Raycast Target

2)UGUI Canvas.SendWillRenderCanvases

Canvas.SendWillRenderCanvases 函数的耗时代表的是 UI 元素自身变化带来的更新耗时，这是需要和 Canvas.BuildBatch 的网格重建的耗时所区分的。

持续的高耗时往往是由于 UI 元素过于复杂且更新过于频繁造成。UI 元素的自身更新包括：替换图片、文本或颜色发生变化等等。UI 元素发生位移、旋转或者缩放并不会引起该函数有开销。该函数的耗时取决于 UI 元素发生更新的数量以及 UI 元素的复杂度，因此要优化此函数的开销通常可以从如下几点着手：

a.　降低频繁更新的 UI 元素的频率

比如小地图的怪物标记、角色或者怪物的血条等，可以控制逻辑在变动超过某个阈值时才更新 UI 的显示，再比如技能 CD 效果，伤害飘字等控制隔帧更新。

b.　尽量让复杂的 UI 不要发生变动

如某些字符串特别多且又使用了 Rich Text、Outline 或者 Shadow 效果的 Text，Image Type 为 Tiled 的 Image 等。这些 UI 元素因为顶点数量非常多，一旦更新便会有较高的耗时。如果某些效果需要使用 Outline 或者 Shadowmap，但是却又频繁的变动，如飘动的伤害数字，可以考虑将其做成固定的美术字，这样顶点数量就不会翻 N 倍。

c. 关注 Font.CacheFontForText

该函数往往会造成一些耗时峰值。该 APl 主要是生成动态字体 Font Texture 的开销，在运行时突发高耗时，很有可能是一次性写入很多新的字符，导致 Font Texture 纹理扩容。可以从减少字体种类、减少字体字号、提前显示常用字以扩充动态字体 FontTextures 等方式去优化这一项的耗时。

3)UGUI Canvas.BuildBatch

Canvas.BuildBatch 为 Ul 元素合并的 Mesh 需要改变时所产生的调用。通常之前所提到的 Canvas.SendWillRenderCanvases()的调用都会引起 Canvas.BuildBatch 的调用。另外，Canvas 中的 UI 元素发生移动也会引起 Canvas.BuildBatch 的调用。

Canvas.BuildBatch 是在主线程发起 UI 网格合并，具体的合并过程是在子线程中处理的，当子线程压力过大，或者合并的 UI 网格过于复杂的时候，会在主线程产生等待，等待的耗时会被统计到 EmitWorldScreenspaceCameraGeometry 中。

这两个函数产生高耗时，说明发生重建的 Canvas 非常复杂，此时需要将 Canvas 进行细分处理，通常是将静态的元素放在一个 Canvas 中，将发生更新的 UI 元素放入另一个 Canvas 中，这样静态的 Canvas 由于缓存不会发生网格更新，从而降低网格更新的复杂度，减少网格重建的耗时。

4)UGUI CanvasRenderer.SyncTransform

我们常注意到有些项目的部分帧中 CanvasRenderer.SyncTransform 调用频繁。当 Canvas.SyncTransform 触发次数非常频繁时，会导致它的父节点 UGUl.Rendering.UpdateBathes 产生非常高的耗时。

在 Unity2018 版本及以后的版本中，Canvas 下某个 UI 元素调用 SetActive（false 改成 true）会导致该 Canvas 下的其它 Ul 元素触发 SyncTransform，从而导致 UI 更新的整体开销上升，在 Unity2017 的版本中只会导致该 UI 元素本身触发 SyncTransform。

所以，针对 UI 元素（如 Image、Text）特别多的 Canvas，需要注意是否存在一些 UI 元素在频繁地 SetActive，对于这种情况建议使用 SetScale（0 或者 1）来代替 SetActive（false 或者 true）。或者，也可以将 Canvas 适当拆分，让需要进行 SetActive(true)操作的元素和其它元素不在一个 Canvas 下，就不会频繁调用 SyncTransform 了。

5) UGUI Ul DrawCall

通常战斗场景中其它模块耗时压力大，此时 UI 模块更要仔细控制性能开销。一般而言，战斗场景中的 UI DrawCall 控制到 40-50 左右为最佳。

在不减少 UI 元素的前提下，控制 DrawCall 的问题，其实也就是如何使得 UI 元素尽量合批的问题。一般的合批要求材质相同，而在 UI 中却常常会发生明明是使用同一材质、同一图集制作的 UI 元素却无法合批的现象。这其实和 UGUI DrawCall 的计算原理有关。

在 UGUI 的制作过程中，建议关注以下几点：

① 同一 Canvas 下的 UI 元素才能合批。不同 Canvas 即使 Order in Layer 相同也不合批，所以 UI 的合理规划和制作非常重要。

② 尽量整合并制作图集，从而使得不同 UI 元素的材质图集一致。图集中的按钮、图标等需要使用图片的比较小的 UI 元素，完全可以整合并制作图集。当它们密集地同时出现时，就有效降低了 DrawCall。

③ 在同一 Canvas 下、且材质和图集一致的前提下，避免层级穿插。简单概括就是，应使得符合合批条件的 UI 元素的“层级深度”相同；

④ 将相关 UI 的 Pos Z 尽量统一设置为 0，Z 值不为 0 的 UI 元素只能与 Hierarchy 中相

邻元素尝试合批，所以容易打断合批。

⑤ 对于 Alpha 为 0 的 Image，需要勾选其 CanvasRenders 组件上的 Cull Transparent Mesh 选项，否则依然会产生 DrawCall 且容易打断合批。

6)UI Prefab 制作不规范

a. prefab 里嵌套 prefab，会导致 prefab 打的 AB 很大

图 6. 预制体中嵌套预制体

上图摆放了 3 个 Image，它是独立的 prefab，会单独打 AB。在打 AB 的时候，这 3 个 Image 都会被打进主体的 AB 里面，导致主体的 AB 增大。

推荐通过脚本中引用的方式挂载 prefab，即在脚本中用一个 GameObject 的属性指向 prefab，然后在业务逻辑里单独动态加载，这种方式打 ab 的时候不会打进主体 AB 里。

这样改动后 UI 界面可能需要重新调整位置，可行的方法是：

① 有自适应组件的，无需额外操作。

② 位置不规则的，可以在主体 prefab 中创建对应的空节点来设置位置。

③ 位置规则的，在代码里自行计算位置。

b. 动态加载的图片，最好不要保留在 prefab 上

动态加载的图片，一般是由开发人员在逻辑代码里根据数据进行动态加载，因此不需要放在 prefab 上，留在 prefab 上会使得 AB 增大，包体增大。

这样做也存在不好的地方，图片资源引用查找是一个问题！

c. 纯色图片使用九宫格

避免用大的纯色图片，应该用九格宫图片拉伸来代替。

d. 图片能合成一张的就合成一张，能打进图集的就打进图集

我们使用 4 张未合并图集的图像进行拼合，那么就会产生 4 个 batch。

推荐做法：

① 将用到的图片打到图集中。

② 将用到的图片合成一张图片，如果没有特殊原因（如某些情况只显示其中一张图），完全可以合成一张图片。

e. 公用的 UI 应尽量做成独立的 Prefab

在多个地方用到的 UI 要抽离出来做成独立的 Prefab，避免资源重复打进 AB，导致 AB 变大，包体变大。

推荐做法：

① 本身对 AB 影响较小的可以不用抽离。

② 如果对 AB 影响较大的，有必要抽离，这样不仅能减少 AB 大小，同时能加快 AB 资源加载。

f. 默认关闭 RaycastTarget

RaycastTarget 对性能有一定的影响，应该有对应的开发控制是否要开启。

推荐做法：

① 扩展 Image、RawImage、Text 等组件，默认使 RaycastTarget 关闭，按需开启。

② 扩展 Graphic，提供形如 EmptyRaycast 的类以实现点击等交互功能。

g. 减少自动布局组件的使用

自动布局组件包括 GridLayoutGroup、VirtualLayoutGroup 和 HorizontalLayoutGroup，这些组件存在一定的性能消耗，每当他们中的子元素标记为 Dirty 的时候都会重新计算布局。

推荐做法：

① 挂载空节点作为位置节点。

② 代码里自行计算位置。

h. Mask 和 RectMask2D 使用注意事项

Mask 依赖 Image 组件，占用两个 Batch，多一倍 Overdraw，可以裁剪任意形状。

RectMask2D 不依赖 Image 组件，不占用 Batch，没有 Overdraw，只能裁剪规则形状。

因此，一般情况下，规则的裁剪尽量用 RectMask2D 代替 Mask。

i. 注意 RenderTexture 特效使用

在制作 RenderTexture 特效的时候，注意 RenderTexture 特效尺寸大小，能小就尽量小，比如 300*300 的尽量用 256*256 来实现，尽量避免全屏 RenderTexture 特效。

推荐做法：

需要有后处理效果的，用 RT 特效代替 UIParticle。

RT 特效的尺寸尽可能的小。

尽量避免全屏特效的使用。

j. 是否需要 Canvas 动静分离

UI 元素每次发生如位置、大小、材质等属性变化都会触发所属 Canvas 下的所有元素网

格重建，如果 UI 很复杂，会产生一定的消耗。

动的元素（即属性发生变化）会触发网格重建，静的元素（即属性没有变化）不会触发网格重建。

动静分离就是要减去静态部分网格重建消耗，将动态的元素放到一个 Canvas 下，将静态的元素放到另外一个 Canvas 下，只有动态的 Canvas 才会触发网格重建。（Canvas 之间是不能合批的！）unity 在 5.2 中就已经将网格重建的计算放到了多线程中去做了，因此对游戏线程影响不大，但是频繁的线程交互也是一定的消耗的。

修改发生位置变化的对象的数量，可以发现 UpdateBatches 的耗时会跟着发生变化，得出的结论是：

① UI 中对象属性的变化（位置、缩放等）会导致 UpdateBatches 消耗增高，并且会触发 Canvas.BuildBatch。

② UpdateBatches 耗时增高的原因是等待 Canvas.BuildBatch 完成和位置变化的消耗（即 CanvasRender.SyncTransform 和 CanvasRender.SyncWorldRect）。

③ Canvas.BuildBatch 本身消耗很低，mesh 合并的工作是在多线程中进行，UpdateBatches 需要等待多线程中 mesh 合并完成。

④ UI 中发生属性变化的对象越多，UpdateBatches 的消耗越高。

⑤ UpdateBatches 的消耗跟 UI 界面的 Batches 数量关系不大。

一般 UI 界面发生属性变化的对象并不多，对性能的影响非常低。

推荐做法：

① 一般的界面不推荐采用动静分离，各个 canvas 不能合批。

② 战斗飘字、血条、怪物名字等可以放到单独的 Canvas 中。

③ 可以采取一定的策略，如降频更新 UI、设定阈值更新 UI（超过多少值才会更新）。

4. 物理模块

1) Auto Simulation

在 Unity2017.4 版本之后，物理模拟的设置选项 Auto Simulation 被开放并且默认开启，即项目过程中总是默认进行着物理模拟。但在一些情况下，这部分的耗时是浪费的。

判断物理模拟耗时是否被浪费的一个标准就是 Contacts 数量，即游戏运行时碰撞组件数量。一般来说，碰撞对的数量越多，则物理系统的 CPU 耗时越大。但在很多移动端项目中，我们都检测到在整个游戏过程中 Contacts 数量始终为 O。

在这种情况下，开发者可以关闭物理的自动模拟来进行测试。如果关闭 Auto Simulation 并不会对游戏逻辑产生任何影响，在游戏过程中依然可以进行很好地对话、战斗等，则说明可以节省这方面的耗时。同时也需要说明的是，如果项目需要使用射线检测，那么在关闭 Auto Simulation 后需要开启 Auto Sync Transforms，来保证射线检测可以正常作用。

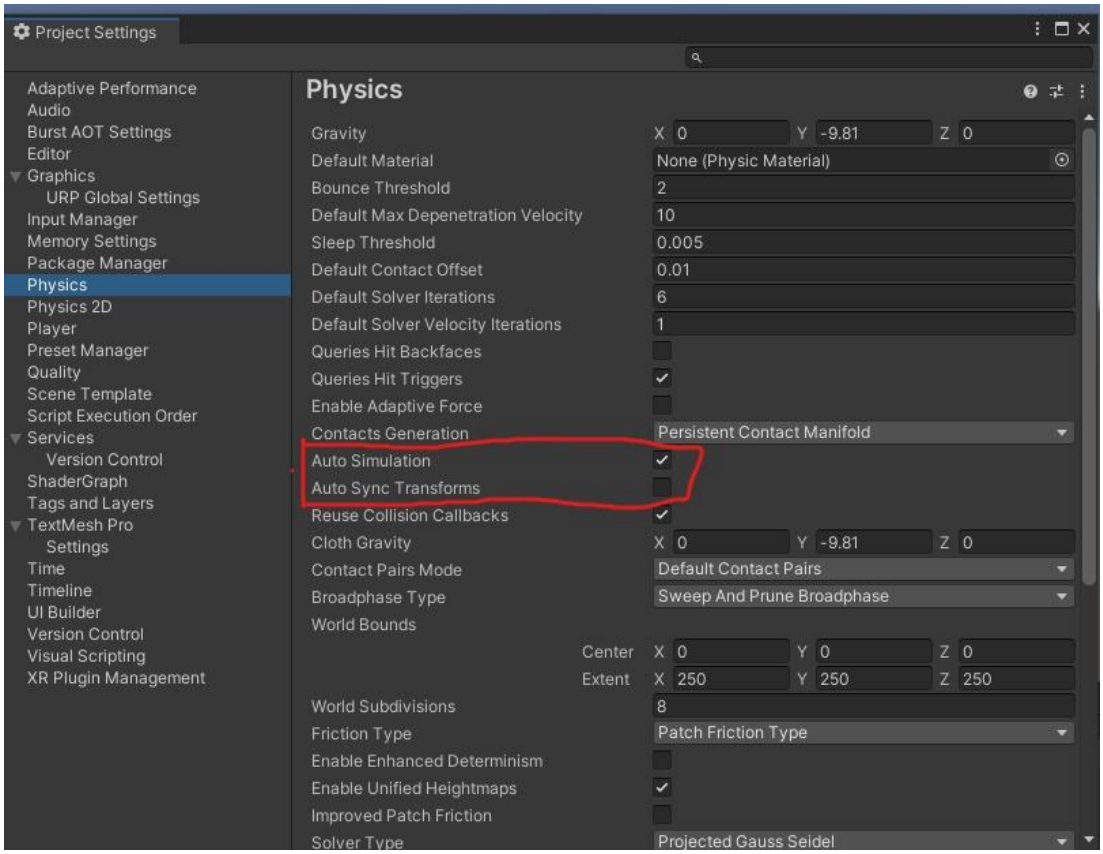

图 7.　　Unity 物理模块的自动模拟默认开启

2)物理更新次数

Unity 物理模拟过程的主要耗时函数是在 FixedUpdate 中的，也就是说，当每帧该函数调用次数越高、物理更新次数也就越频繁，每帧的耗时也就相应地高。

物理更新次数，或者说 FixedUpdate 的每帧调用次数，是和 Unity Project Settings 的 Time 设置中最小更新间隔（Fixed Timestep）以及最大允许时间（Maximum Allowed Timestep）相关的。这里我们需要先知道物理系统本身的特性，即当游戏上一帧卡顿时，Unity 会在当前帧非常靠前的阶段连续调用 N 次 FixedUpdate.PhysicsFixedUpdate，Maximum AllowedTimestep 的意义就在于限制物理更新的次数。它决定了单帧物理最大调用次数，该值越小，单帧物理最大调用次数越少。现在设置这两个值分别为 20ms 和 100ms，那么当某一帧耗时 30ms 时，物理更新只会执行 1 次；耗时 200ms 时也只会执行 5 次。

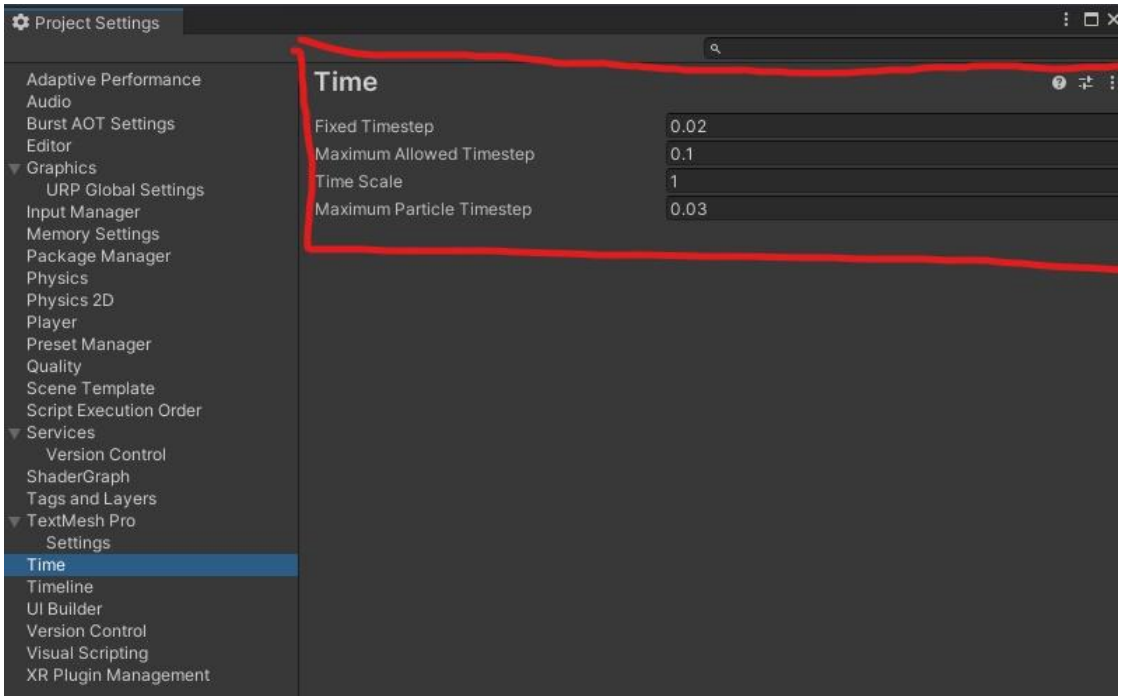

图 8.　　Unity Time 设置

所以一个行之有效的方法是调整这两个参数的设置，尤其是控制更新次数的上限（默认为 17 次，最好控制到 5 次以下），物理模块的耗时就不会过高；另一方面则是先优化其它模块的 CPU 耗时，当项目运行过程中耗时过高的帧很少，则 FixedUpdate 也不会总是达到每帧更新次数的上限。这对于其它 FixedUpdate 中的函数是同理的，也是基于这种原因，我们一般不建议在 FixedUpdater 中写过多游戏逻辑。

3)Contacts

就像上面提到的，如果我们确实用到物理模拟，则一般碰撞对的数量越多，物理系统的 CPU 耗时也就越大。所以，严格控制碰撞对数量对于降低物理模块耗时非常重要。

首先，很多项目中可能存在一些不必要的 Rigidbody 组件，在开发者不知情的地方造成了不必要的碰撞，从而产生了耗时浪费；另外，可以检查修改 Project Settings 的 Physics 设置中的 Layer Collision Matrix，取消不必要的层之间的碰撞检测，将 Contacts 数量尽可能降低。

5. 动画模块

1) Mecanim 动画系统

Mechanic 动画系统是 Unity 公司从 Unity4.O 之后开始引入的新版动画系统（使用 Animator 控制动画），相比于 Legacy（遗留）的 Animation 控制系统，在功能上，Mecanim 动画系统主要有以下几点优势：

① 针对人形角色提供了一套特殊的工作流，包括 AvatarE 的创建以及 Muscles 肌肉的调节；

② 动画重定向（Retarting）的能力，可以非常方便地把一个动画从一个角色模型应用到其他角色模型上；

③ 提供了可视化的 Animator 编辑器，可以快捷预览和创建动画片段；

④ 更加方便地创建状态机以及状态之间 Transition 的转换；

⑤ 便于操作的混合树功能。

在性能上，对于骨骼动画且曲线较多的动画，使用 Animator 的性能是要比 Animation

要好的，因为 Animator 是支持多线程计算的，而且 Animator 可以通过开启 Optimized GameObjects 进行优化。相反，对于比较简单的类似于移动旋转这样的动画，使用 Animation 控制则比 Animator 要高效一些。

2)BakeMesh-网格烘培

对于一两千面这样面数较少且动画时长较短的对象，如 MOBA、SLG 中的小兵等，可考虑用 SkinnedMeshRenderer.BakeMesh 的方案，用内存换 CPU 耗时。其原理是将一个蒙皮动画的某个时间点上的动作，Bake 成一个不带蒙皮的 Mesh，从而可以通过自定义的采样间隔，将一段动画转成一组 Mesh 序列顿。而后在播放动画时只需选择最近的采样点（即一个 Mesh）进行赋值即可，从而省去了骨骼更新与蒙皮计算的时间（几乎没有动画，只是赋值的动作）。整个操作比较适合于面片数小的人物，因为此举省去了蒙皮计算。其作用在于：用内存换取计算时间，在场景中大量出现同一个带动画的模型时，效果会非常明显。该方法的缺点是内存的占用极大地受到模型顶点数、动画总时长及采样间隔的限制。因此，该方法只适用于顶点数较少，且动画总时长较短的模型。同时，Bake 的时间较长，需要在加载场景时完成。

3)Active Animator 数量

Active（激活）状态的 Animator 个数会极大地影响动画模块的耗时，而且是一个可量化的重要标准，控制其数量到一个相对合理的值是我们优化动画模块的重要手段。需要开发者结合画面排查对应的数量是否合理。

a. Animator Culling Mode

控制 Active Animator 的一个方法是针对每个动画组件调整合理 Animator.CullingMode 设置。该项设置一共有三个选项：AlwaysAnimate、CullUpdateTransforms 和 CullComplete。

默认的 AlwaysAnimate 使得当前物体不管是不是在视域体内，或者在视域体被 LOD Culling 掉了，Animator 的所有东西都仍然更新；其中，UI 动画一定要选 AlwaysAnimate，不然会出现异常表现。

而设置为 CullUpdateTransforms 时，当物体不在视域体内，或者被 LOD Culling 掉后，逻辑继续更新，就表示状态机是更新的，动画资源中连线的条件等等也都是会更新和判断的；但是 Retarget、IK 和从 C++回传 Transform 这些显示层的更新就不做了。所以，在不影响表现的前提下把部分动画组件尝试设置成 CullUpdateTransforms 可以节省物体不可见时动画模块的显示层耗时。

最后，CullComplete 就是完全不更新了，适用于场景中相对不重要的动画效果，在低端机上需要保留显示但可以考虑让其静止的物体，分级地选用该设置。

b. DOTween 插件

很多时候，UI 动画也会贡献大量的 Active Animator。针对一些简单的 UI 动画，如改变颜色、缩放、移动等效果建议改用 DOTween 制作，性能比原生的 UI 动画要好得多。

4)开启 Apply Root Motion 的 Animator 数量

在 Animators.Update 的堆栈中，有时会看到 Animator.ApplyBuiltinRootMotion 占比过高，这一项通常和项目中开启了 ApplyRoot Motion 的模型动画相关。如果其动画不需要产生位移，则不必开启此选项。

5) Animator.Initialize

Animator.Initialize APl 会在含有 Animator 组件的 GameObject 被 Active 和 Instantiate 时触发，耗时较高。因此尤其是在战斗场景中不建议过于频繁地对含有 Animator 的 GameObject 进行 Deactive/Active GameObjectt 操作。对于频繁实例化的角色和 Ul，可尝试通过缓冲池的方式进行处理，在需要隐藏角色时，不直接 Deactive 角色的 GameObject，而是 Disable Animator 组件，并把 GameObject 移到屏幕外；在需要隐藏 UI 时，不直接 Deactive UI 对象，而是将其 SetScale＝O 并且移出屏幕的方式，也不会触发 Animator.Initialize。

6)Meshskinning.Update 和 Animators.WriteJob

网格资源(Mesh)对于动画模块耗时的影响是十分显著的。

一方面，Meshskinning.Update 耗时较高时。主要因素为蒙皮网格的骨骼数和面片数偏高，所以可以针对网格资源进行减面和 LOD 分级。

另一方面，默认设置下，我们经常发现很多项目中角色的骨骼节点的 Transform 一直都是在场景中存在的，这样在 Native 层计算完它们的 Transform 后，会回传给 C＃层，从而产生一定的耗时。

在场景中角色数量较多，骨骼节点的回传会产生一定的开销，体现在动画模块的主函数之一 PreLateUpdate.DirectorUpdateAnimationEnd 的 Animators.WriteJob 子函数上。

对此开发者可以考虑勾选 FBX 资源中 Rig 页签下的 Optimize Game Objects 设置项，将骨骼节点“隐藏”，从而减少这部分的耗时。

7)GPU Skinning/Compute Skinning

特别地，对于 Unity 引擎原生的 GPU Skinning 设置项（新版 Unity 中为 Compute Skinning），理论上会在一定程度上改变网格和动画的更新方法以优化对骨骼动画的处理，但从针对移动平台的多项测试结果来看，无论是在 iOS 还是安卓平台上，多个 Unity 版本提供的 GPU Skinning.对性能的提升效果都不明显，甚至存在负优化的现象。在 Unity 的迭代中已对其逐步优化，将相关操作放到渲染线程中进行，但其实用性还需要进一步考察。

6. 粒子系统

1) Playing 粒子系统数量

粒子系统数量和 Playing 状态的粒子系统数量，前者是指内存中所有的 ParticleSysteml 的总数量，包含正在播放的和处于缓存池中的；后者指的是正在播放的 ParticleSystem 组件的数量，这个包含了屏幕内和屏幕外的，我们建议在一顿中出现的数量峰值不超过 50（1GB 机型）。

针对这两个数值，我们一方面关注粒子系统数量峰值是否偏高，可选中某一峰值帧查看到底是哪些粒子系统缓存着、是否都合理、是否有过度缓存的现象；另一方面关注 Playing 数量峰值是否偏高，可选中某一峰值帧查看到底是哪些粒子系统在播放、是否都合理、是否能做些制作上的优化。

2) Prewarm

ParticleSystem.Prewarm 的耗时有时也需要关注。当有粒子系统开启了 Prewarm 选项，其在场景中实例化或者由 Deactive 转为 Activel 时，会立即执行一次完整的模拟。

但 Prewarml 的操作通常都有一定的耗时，经测试，大量开启 Prewarm 的粒子系统同时 SetActive 时会造成耗时峰值。建议在不必要的情况下，将其关闭。

7. 加载模块

1) Shader 加载

a. Shader.Parse

Shader.Parse 是指 Shader 加载进行解析的操作，如果此操作较为频繁，通常是由于 Shader 的重复加载导致的，这里的重复可以理解为两层意思。

第一层是由于 Shader 的冗余导致的，通常是因为打包 AssetBundlef 的时候，Shader 被被动打进了多个不同的 AssetBundler 中而没有进行依赖打包，这样当这些 AssetBundle 中的资源进行加载的时候，会被动加载这些 Shader，就进行了多次“重复的”Shader.Parse，所以同一种 Shader 就在内存中有多份了，这就是冗余了。

要去除这种冗余的方法也很简单，就是把这些会冗余的 Shader 依赖打包进一个公共的 AssetBundle 包。这样就会主动打包了，而不是被动进入某些使用了这个 Shader 的包体中。如果对这个 Shader 进行了主动打包，那么其它使用了这个 Shader 的 AssetBundle 中就只会对这个 Shader 打出来的公共 AssetBundle 进行引用，这样在内存中就只有一份 Shader，其它用到这个 Shader 的时候就直接引用它，而不需要多次进行 Shader.Parse 了。

第二层意思是同一个 Shader 多次地加载卸载，没有缓存住导致的。假设 AssetBundle 进行了主动打包，生成了公共的 AssetBundle，这样在内存中只有这一份 Shader，但是因为这个 Shader 加载完后（也就是 Shader.Parse）没有进行缓存，用完马上被卸载了。下次再用到这个 Shader 的时候，内存里没有这个 Shader 了，那就必须再重新加载进来，这样同样的一个 Shader 加载解析了多次，就造成了多次的 Shader.Parse。一般而言，经过变体优化以后的开发者自己写的 Shader 内存占用都不高，可以统一在游戏开始时加载并缓存。

特别地，对于 Unity 内置的 Shader，只要是变体数量不多的，可以放进 Project Settings 中的 Always Included 中去，从而避免这一类 Shader 的冗余和重复解析。

b. Shader.CreateGPUProgram

该 API 也会在加载模块主函数甚至 UI 模块、逻辑代码的堆栈中出现。相关的讨论上文已经涉及，优化方法相同，不再赘述。

2)Resources.UnloadUnusedAssets

该 APl 会在场景切换时被 Unity 自动调用，一般单次调用耗时较高，通常情况下不建议手动调用。

但在部分不进行场景切换或用 Additive 加载场景的项目中，不会调用该 APl，从而使得项目整体资源数量和内存有上升趋势。对于这种情况则可以考虑每 5-10min 手动调用一次。

Resources.UnloadUnusedAssets 的底层运作机理是，对于每个资源，遍历所有 Hierarchy Tree 中的 GameObjects 结点，以及堆内存中的对象，检测该资源是否被某个 GameObject 或对象（组件）所使用，如果全部都没有使用，则引擎才会认定其为 Unused 资源，进而进行卸载操作。简单来讲，Resources.UnloadUnusedAssetsl 的单次耗时大致随着（（GameObject 数量＋Mono 对象数量）＊Asset 数量）的乘积变大而变大。

因此，该过程极为耗时，并且场景中 GameObject/Asset 数量越高，堆内存中的对象数越高，其开销也就越大。对此，我们的建议如下：

a. Resources.UnloadAsset/AssetBundle.Unload（True）

研发团队可尝试在游戏运行时，通过 Resources.UnloadAsset/AssetBundle.Unload（True 来去除已经确定不再使用的某一资源，这两个 API 的效率很高，同时也可以降低 Resources.UnloadUnusedAssets 统一处理时的压力，进而减少切换场景时该 API 的耗时。

b. 严格控制场景中材质资源和粒子系统的使用数量

专门提到这两种资源，因为在大多数项目中，虽然它们的内存占用一般不是大头，但往往资源数量远高于其他类型的资源，很容易达到数千的数量级，从而对单次 Resources.UnloadUnusedAssets 耗时有较大贡献。

c. 降低驻留的堆内存

堆内存中的对象数量同样会显著影响 Resources.UnloadUnusedAssets 的耗时，这在上文也已经讨论过。

3)加载 AssetBundle

使用 AssetBundle 加载资源是目前移动端项目中比较普遍的做法。

而其中，应尽量用 LZ4 压缩格式打包 AssetBundle，并用 LoadFromFile 的方式加载。经测试，这种组合下即便是较大的 AssetBundle 包（包含 10 张 1024＊1024 的纹理），其加载耗时也仅零点几毫秒。而使用其他加载方式，如 LoadFromMemory，加载耗时则上升到了数十毫秒；而使用 WebRequest 加载则会造成 AssetBundle 包的驻留内存显著上升。

这是因为，LoadFromFile 是一种高效的 API，用于从本地存储（如硬盘或 SD 卡）加载未压缩或 LZ4 压缩格式的 AssetBundle。

在桌面独立平台、控制台和移动平台上，API 将只加载 AssetBundle 的头部，并将剩余的数据留在磁盘上。AssetBundle 的 Objects 会按需加载，比如：加载方法（例如：AssetBundle.Load）被调用或其 InstancelD 被间接引用的时候。在这种情况下不会消耗过多的内存但在 Editor 环境下，APl 还是会把整个 AssetBundle 加载到内存中，就像读取磁盘上的字节和使用 AssetBundle.LoadFromMemoryAsync 一样。如果在 Editor 中对项目进行了分析，此 API 可能会导致在 AssetBundle 加载期间出现内存尖峰。但这不应影响设备上的性能，在做优化之前，这些尖峰应该在设备上重新再测试一遍、

要注意，这个 API 只针对未压缩或 LZ4 压缩格式，因为如果使用 LZMA 压缩，它是针对整个生成后的数据包进行压缩的，所以在未解压之前是无法拿到 AssetBundle 的头信息的。

由于 LoadFromMemory 的加载效率相较其他的接口而言，耗时明显增大，因此我们不建议大规模使用，而且堆内存会变大。如果确实有对 AssetBundle 文件加密的需求，可以考虑仅对重要的配置文件、代码等进行加密，对纹理、网格等资源文件则无需进行加密。因为目前市面上已经存在一些工具可以从更底层的方式来获取和导出渲染相关的资源，如纹理、网格等，因此，对于这部分的资源加密并不是十分的必要性。

4)加载资源

有关加载资源所造成的耗时，若加载策略比较合理，则一般发生在游戏—开始和场景切换时，往往不会造成严重的性能瓶颈。但不排除一些情况需要予以关注，那么可以把资源加载耗时的排序作为依据进行排查。

对于单次加载耗时过高的资源，比如达到数百毫秒甚至几秒时，就应考察这类资源是否过于复杂，从制作上考虑予以精简。

对于反复频繁加载且耗时不低的资源，则应该在第一次加载后予以缓存，避免重复加载造成的开销值得一提的是，在 Unity 的异步加载中有时会出现每帧进行加载所能占用的最高耗时被限制，但主线程中却在空转的现象。尤其是在切场景的时候集中进行异步加载，有时会耗费几十甚至数十秒的时间，但其中大部分时间是被空转浪费的。这是因为控制异步加载每帧最高耗时的 API Application.backgroundLoadingPriority 默认值为 BelowNormal，每帧最多只加载 4ms。此时一般建议把该值调为 High，即最多 50ms 每帧。

5)实例化和销毁

实例化同样主要存在单个资源实例化耗时过高或某个资源反复频繁实例化的现象。根据耗时多少排列后，针对疑似有问题的资源，前者考虑简化，或者可以考虑分帧操作，比如对于一个较为复杂的 UI Prefab，可以考虑改为先实例化显眼的、重要的界面和按钮，而翻页后的内容、装饰图标等再进行实例化；后者则建立缓存池，使用显隐操作来代替频繁的实例化。

6)激活和隐藏

激活和隐藏的耗时本身不高，但如果单帧的操作次数过多就需要予以关注。可能出于游戏逻辑中的一些判断和条件不够合理，很多项目中往往会出现某一种资源的显隐操作次数过多，且其中 SetActive（True 远比 SetActive（False）次数多得多、或者反之的现象，亦即存在大量不必要的 SetActive 调用。由于 SetActive APl 会产生 C＃和 Native 的跨层调用，所以数量一多，其耗时仍然是很可观的。针对这种情况，除了应该检查逻辑上是否可以优化外，还可以考虑在逻辑中建立状态缓存，在调用该 API 之前先判断资源当前的激活状态。相当于使用逻辑的开销代替该 API 的开销，相对耗时更低一些。

7)逻辑代码

逻辑代码的 CPU 耗时优化更多是结合项目实际需求、考验程序员本人的过程，很难定量定性进行讨论。

但是，我们发现有越来越多的团队在使用 JobSystem 将主线程中的部分逻辑代码放入子线程中来进行处理，对于可以并行运算的逻辑，非常推荐将其放入到子线程中来处理，这样可以有效降低主线程 CPU 处理逻辑运算的压力。

8)同时加载多个模型

加载卡顿性能优化其实就是这三种方式：按需加载，异步加载，分帧加载。

a. 按需加载

需要显示的时候才显示，不需要的就隐藏，不渲染，减少 GPU 压力。

b. 异步加载

同步加载会导致主线程停顿，等待单帧完成太多事情：加载纹理、模型、动画，实例化，单帧加载耗时过高，超过一帧的时间，导致掉帧；我们可以让主线程继续往前运行，不等待返回结果，这样就不会造成主线程卡死。

c. 预加载

所有模型都是要显示的，我们已经预知需要用到所有模型，可以让模型全部加载完成，再开始动画。

d. 合并网格

每个模型都是由各个部件组成的，有头饰、脸部、披风、上衣、下装、手套、鞋子、还有阴影，如果不去合批的话，同屏多个模型，Draw Call 过高，导致 GPU 压力过大，GPU 来不及绘制，导致掉帧，我们可以把上述（除了阴影）都合并网格，检查调用 Draw Call 命令次数。

e. 多线程加载

在线程中创建游戏对象？所有与 Unity 相关的对象都不是线程安全的，只能从主线程访问。

奇淫技巧，把主线程传参到线程，让线程访问主线程 API。

8. GC 优化

1) 可变参数

params 是个语法糖，它跟传入一个数组是等价的。

Func(1,2,3); == Func(new int[]{1,2,3});

Func(); == Func(Array.Empty<int>)，这是 C#缓存的空数组，并不会产生新的临时对象。

所以只要可变参数不为空，就会产生临时数组，产生 GC，优化方法是使用一系列重载方法代替，覆盖高频调用的部分，允许部分产生 GC 来处理。

比如 C#的 string.Format 的实现 ，把常用的 1、2、3 个参数单独提出来，剩下的用可变参数实现。

PS：

```
public static String Format(String format, Object arg0)
public static String Format(String format, Object arg0, Object arg1)
public static String Format(String format, Object arg0, Object arg1, Object arg2)
public static String Format(String format, params Object[] args)
```

2)常用容器数据结构及增长方式

List、Stack、Queue 这几个容器都维护着一个一定容量的数组，当添加数据，数组长度不够的时候，容器容量会增长，新数组长度是旧数组的两倍大小，建立新数组之后，会将数据从旧数组复制到新数组中，最后旧数组会做完内存垃圾丢弃。

PS：

```
if(Count+1 < array.Length){
        var newArray = new T[array.Length * 2];
        Array.Copy(array, 0, newArray, array.Length);
        array = newArray;
}
```

因此，在初始化容器的时候，如果能知道数据大概的大小，最后指定容器的初始值，这

样才能最大限度防止容量的频繁改变。

3)匿名方法和协程

总结，尽量避免使用匿名方法，如果需要使用，可以缓存一下再用。

PS：

```
Action action;
void Func(){
    if(action == null){
        action=()=> a = 2;
    }
    Call(action);
}
```

4)Unity 所有返回为数组的 API 都有 GC Alloc

比如 var texts = GetComponentsInChildren<Text>(); 每次调用都会产生一个新的数组对象，可以配合 ListPool 进行优化。

PS：

```
var texts = ListPool<Text>.Get();
GetComponentsInChildren(texts);
...
ListPool<Text>.Release(texts);
```

再比如：var materails = renderer.sharedMaterials;==>GetSharedMaterials(xxx);

navMeshPath.corners==>GetCornersNonAlloc(xxx)

Physics.RayCastAll==>Physics.RayCastNonAlloc

5)UGUI Prefab Text

prefab 中有大量空的 Text，初始化的时候有一个很严重的 GC Alloc，因为初始化的时候，会先初始化 TextGenerator，如果 Text 为空，则会按 50 个字来初始化，这种可以不让它为空，或者填一个空格进去。

PS：

```
public TextGenerator(): this(50){}
public TextGenerator(int initialCapacity){...}
```

6)字符串比较

先用 string 变量存储 obj.name，这样只有一个内存空间保存；如果不存储 obj.name，每一次比较都会产生新的内存空间、不使用 obj.tag==”Tag” 进行比较，而是使用避免 GC 的 obj.CompareTag(“tag”) 。

三. 画面表现与 GPU 压力的权衡

1. 带宽

1) GPU 压力与发热/耗能

芯片底层的情况远比我们想象的复杂，从经验和跟芯片厂家的专业硬件工程师沟通后得出的结论是。移动设备上 GPU 带宽的压力还是比较影响能耗的，特别是在发热这一方面也有不小的影响。但这是定性的说法，目前我们和芯片厂商都没有特别定量的公式来具体说明其影响大小。因此，当一个项目的发热或者能耗较大，带宽是开发者特别需要关注的地方。一般而言长时间维持在 55℃以上就是需要警惕的温度了

2)GPU 压力与帧率

上文中已提到过，GPU 压力大会使得 CPU 等待 GPU 完成工作的耗时增加，与此同时，也会使得渲染模块 CPU 端的主要函数耗时上升，从而影响帧率、造成卡顿。

除此之外，由于移动端硬件客观上存在体积较小、散热难的问题，使得 GPU 发热会物理上显著影响到 CPU 芯片温度同时上升，严重的即产生降频现象。

3)优化带宽

带宽数据是衡量 GPU 压力的重要参考。以相对高端的机型而言，全分辨率情况下，如果需要跑满 30 帧并发热情况稳定，则需要将总带宽控制到 3000MB/S 以下。为此，常见的优化手段有：

a. 压缩格式

在内存的章节中已经或多或少地讨论过，使用合理的压缩格式，能够有效降低纹理带宽。

b. 纹理 Mipmap

对于 3D 场景而言，对其中的物体所用到的纹理开启 Mipmap 设置，能够用一小部分内存上升作为代价有效降低带宽。当物体离相机越远，引擎就会使用更低层级的 Mipmap 进行纹理采样。但 Mipmap 设置也与合理的纹理分辨率挂钩，一个常见的现象是在实际渲染过程中只用到或者绝大部分用了 Mipmap 的第 0 层进行采样，从而浪费了内存，所以要考虑降低这类纹理的分辨率。

c. 合理的纹理采样方式

除了合理使用 Mipmap 非 0 层采样外，还应关注项目中各向异性采样和三线性插值采样。概括来说，纹理压缩采样时会去读缓存里面的东西，如果没读到就会往离 GPU 更远的地方去读 System Memory，因此所花的周期越多。

采样点增多的时候，cache miss 的概率就会变大，造成带宽上升。各向异性采样次数在 Unity 中设置有 1-16，应尽量设置为 1；三线性采样采 8 个顶点，相对于双线性采样是翻倍的。

d. 修改渲染分辨率

直接修改渲染分辨率为 0.9 倍乃至更低，减少参与纹理采样的像素，更加有效地降低带宽。

此外，还应注意读顶点的带宽。相比纹理，它的占比一般会比较小。但不同于纹理的是，修改渲染分辨率可以有效降低读纹理的带宽，但读顶点的带宽不会受到影响。所以在上文中针对网格资源和同屏渲染面片数的控制卓有成效后，读顶点的带宽值应该占总带宽的 10%-20%较为合理。

4)Overdraw

Overdraw，即多次绘制同一像素造成的 GPU 开销。在场景中渲染顺序控制合理的理想状况下，不透明物体的 Overdraw 应控制在 1 层。所以，造成 Overdraw 的主要元凶就是半透

明物体，也即粒子系统和 Ul。

a. 粒子系统

建立一个专门的空的测试场景，在其中顺次播放我们项目中要用到的粒子系统，结合测试截图找到播放时 GPU 耗时较高的粒子系统。

在筛选出需要优化的粒子系统后，对于低端设备尽可能降低它们的复杂程度和屏幕覆盖面积，从而降低其渲染方面的开销，提升低端设备的运行流畅性。具体做法如下：

① 在中低端机型上对粒子系统的 Max Particles 最大粒子数量进行限制。
② 在中低端机型上只保留“重要的”的粒子系统，比如对于一个火焰燃烧的特效，只保留火焰本身，而关闭掉周边的烟尘效果；
③ 尽可能降低粒子特效在屏幕中的覆盖面积，覆盖面积越大，越容易产生重叠遮盖，从而造成更高的 Overdraw。

b. UI

① 当某个全屏 UI 打开时，可以将被背景遮挡住的其他 UI 进行关闭。
② 对于 Alpha 为 0 的 Ul，可以将其 Canvas Renderer 组件上的 CullTransparent Mesh 进行勾选，这样既能保证 UI 事件的响应，又不需要对其进行渲染。
③ 尽可能减少 Mask 组件的使用，使用 Mask 不仅提高绘制的开销，同时会造成 DrawCall 上升。在 Overdraw 较高的情况下，可以考虑使用 RectMask2D 代替。
④ 在 URP 下需要额外关心是否有没必要的 Copy Colors 或者 Copy Depth 存在。尤其是在 UI 和战斗场景中的相机使用同一个 RendererPipelineAsset 的情况下，容易出现不必要的渲染耗时和带宽浪费，这样会对 GPU 造成不必要的开销。通常建议 UI 相机和场景相机使用不同的 RendererData。

2. 渲染效果

除了粒子特效外，我们往往还喜欢用一些炫酷的渲染效果来丰富游戏的表现，比如体积雾、体积光、水体、次表面反射等等，然而场景中用到的此类效果越多，Shader 越复杂，给 GPU 带来的压力越是大到远远超出接受范围的程度。

优化和权衡是决定最后留下哪些渲染效果的主要手段。

一方面，从多个方案中对比选取效果和性能较优的，对开源方案根据自身项目需要进行精简优化；另一方面，根据机型分档和当前 GPU 压力，取重点而舍次要。

3. 后处理

Bloom 几乎是最受开发者喜爱、最为常见的后处理效果了。常见的一个问题是，Bloom 默认是从 1/2 渲染分辨率开始进行下采样的。对此，可以考虑在中低端机型上从 1/4 分辨率开始进行下采样，或减少下采样次数。各种后处理效果的性能开销和实际使用场景并不相同，在实际项目中遇到的问题也往往各不相同。

4. 渲染策略

1) 绘制顺序

当场景中存在先画离相机较远的不透明物体，再画离相机较近的物体，而且两者有所重合时，较远物体被较近物体所遮挡部分的像素就有可能被绘制两次，从而造成 Overdraw。

这种情况常发生在地形上。本来当不透明物体的 Render Queue 一致时，引擎会自动判

断并优先绘制离相机更近的物体。但对于地形而言往往有的部分比其他物体离相机更近，有的却更远，从而被优先绘制。

所以，需要通过对 Render Queue 等设置，使得离相机越近的物体（如任务、物体等）越先绘制，而较远的如地形等最后绘制。则在移动平台上，通过 Ealy-Z 机制，硬件会在片元着色器之前就进行深度测试，离得较远的物体被遮挡的像素深度检测不通过，从而节省不必要的片元计算。

2)无效绘制

存在一些视觉效果不明显可以关闭，或者可以用消耗更低的绘制方案的情况。

比如一种较为常见的情况是，某些绘制背景的 DrawCall，本身屏占比较大，开销不小，但在引擎中开关这个 DrawCall 没有明显的视觉变化，可能是在制作过程中被弃用或者被其他 DrawCall 完全掩盖了的效果，则可以考虑予以关闭。

还有一种情况是，一些背景是用模型绘制的并带有模糊、雾效等额外的渲染效果。但场景中视角固定、这些背景也几乎不发生变化，则可以考虑用静态图替代这些复杂的绘制作为背景，在低端机上把更多性能留给主要的游戏逻辑和表现效果。

3)渲染面积

渲染面积过大造成的性能问题已经在粒子特效中有所反映和讨论了，但事实上对于不透明物体也适用。对于一个 DrawCall 而言，当它的渲染面积较大、且渲染资源多而复杂时，两者便呈现出一种乘积的作用，它意味着有更多的像素参与纹理采样，参与 Shader 计算，给 GPU 带来更高的压力。

5. Shader 复杂度

除了纹理、网格、Render Texture，还有一种对 GPU 压力贡献极大的渲染资源，也就是 Shader。尤其关注 Fragment Shader 的屏占比、指令数和时钟周期数，渲染的像素越多、复杂度越高，说明该 Shader 资源越需要予以优化。