游戏引擎中最能表现并发设计思想的应用就非渲染线程莫属了。把渲染逻辑从游戏线程中分离出来，单独放入一个工作线程里处理凸显了并发执行的优势。原本的渲染逻辑都是在游戏逻辑后串行处理的，早期的游戏引擎也是这么设计的，因为它的结构相对比较简单，容易实现。关键是在上古时期，cpu还只有一个核，即便用了渲染线程也属于脱裤子发屁完全没有必要。但是到了cpu的双核时代，这种情况发生了显著的变化，人们发现cpu单核的工作频率已经遇到了瓶颈，再也不可能提高了，否则cpu就会直接烧掉。以前那种单靠cpu升级就能免费获得的软件性能提升的时代开始一去不复返，游戏行业也面临着同样的问题和挑战。

epic事先做出了改变，它应该是当时做商业引擎的公司中第一个公开支持多线程渲染的厂商。那时的商业引擎像quake，source等还在单核模式下苦苦挣扎。有了多线程渲染的特性支持，cpu端处理渲染命令所占用的帧时间仿佛一下子消失不见了。这其实就是我之前谈到过的并发计算的好处，渲染的任务被时空折叠了，如果你对并发还有什么疑问，建议出门右转看一下我写的另一篇主题文章。**由于渲染线程与主线程的任务是交替执行的，也就是主线程负责游戏世界的模拟，随后根据模拟的结果生成渲染的指令，这些渲染指令并不是在主线程里执行的，而是被投递到了一个独立的工作线程里，这个工作线程维护着一个Ring Buffer，它会把Ring Buffer中由主线程提交的渲染命令按照顺序逐一的提取出来并依次执行。因此我们说这种执行方式是并发的，渲染线程里执行的渲染命令其实是上一帧游戏线程模拟的结果，而在渲染线程处理渲染命令的这段时间里，游戏线程又在模拟下一帧的内容，如此循环往复下去。**这有点酷似CPU的指令多级流水线，只不过现在仅有两级而已。**那为什么要使用Ring Buffer呢？我们知道两个线程之间要进行数据的交互，必须要确保数据的传递是线程安全的，不能出现这个线程还没有写完，另一个线程就开始读取的情况，否则数据的完整性就会缺失。**Ring Buffer能够方便且低成本的解决这类问题，它可以在不加入互斥锁的情况下让数据单向的流动，因为我们知道**在多线程渲染的框架里基本都是游戏线程去生成命令，而渲染线程只是一个消费者**，那么同步操作可以不必那么的复杂。但是要注意一点，Ring Buffer即便使用了Lockfree的设计原则来构建，依然会有额外的访问开销，相对于非线程安全的队列来说还是比较大的。**所以在Unreal中，我们会发现它并不是将所有原生的渲染命令依次的压入Ring Buffer中，而是把渲染场景的操作当成一个统一的渲染命令去执行。**例如渲染场景所需要的渲染物件的Culling，渲染命令的排序，渲染参数的填充，以及渲染命令的执行等步骤都会被放入同一个渲染命令里。除此之外，其他的渲染命令的设计就单纯多了，不会像场景渲染命令那样把很多的操作复合在一起完成，例如纹理数据的填充，顶点和索引数据的写入等。

我们知道CryEngine也是按照这种思路设计的，不过跟Unreal有点细微的差别，CryEngine的Frustum Culling和Occlusion Culling是在主线程里处理的。但是事情总有例外，***像早期的Unity就不按套路出牌。它依旧是把抽象的渲染命令依次的由主线程压入到渲染线程的Ring Buffer中，然后再由渲染线程顺序执行它们，这种做法其实开销是比较大的***，因为我说过**Ring Buffer的填充并非免费，而抽象的渲染命令操作粒度相对较小，这就造成每帧里引擎都会频繁的写入和读取Ring Buffer中的内容。**Unity也意识到了这个问题，在后来的版本做了相应的改进，引入了所谓的Graphics Jobs的概念。**引擎会把渲染命令的构造工作放入到若干个线程里去并行处理，并为每一个线程创建一个独立的命令队列，它们负责临时存放那些新生成的渲染命令，由于Buffer是互相独立的，所以也就没有了Thread Contention，最后再由主线程统一把这些独立的命令队列合并到渲染线程里去执行。**一旦在Unity设置中开启了Graphics Jobs的特性，渲染命令的处理性能就会得到明显的提升。

**对于Unity家的Graphics Jobs的功能，我还想做一些补充说明。它并不完全是为了减少访问Ring Buffer的次数，最主要的目的还是和Unreal一样，希望解决渲染指令生成缓慢的问题**。**为什么渲染指令的构建会比较费时呢？**这里会有多种的原因存在，首先最明显的当然是shader的绑定了，你可能会问shader的绑定不就是设置一个shader对象的指针到runtime的设备上下文里吗？这个操作能有啥消耗呀，但你要知道现代引擎的材质系统都很复杂，它可以支持各种各样不同的渲染效果，而且还允许外部做自由的扩展。所以不同的渲染对象可能绑定了不尽相同的材质实例，这些材质***实例之所以效果变化多端，也是因为材质实例里拥有不同的shader变体。为了选择合适shader变体，unity需要在渲染线程里为每一个物体做搜索，这些搜索还要考虑于不同pass的keywords组合，因此查找shader变体的操作肯定会消耗一定的cpu时间。***

像早期的CryEngine，它会把几乎所有的shader变体都放到同一个数据库里进行管理。所以每次为了找到期待的shader变体，都需要在运行时拼凑出一个超大的hash值，这个hash值指示了当前这个drawcall关联的材质想要开启的预编译宏设定。但即便是同一个材质不同的pass也可能选择不同的shader变体，例如depth pass和g-buffer pass就不会使用同一个shader变体。另外为了支持前向光照，虽然最多只有4盏光源会影响同一个物体，可还是会有很多的分支组合，包括选择何种光照模型，是否要投射阴影等。所以这个hash值就变得特别的复杂，显然它拖慢了shader绑定的速度，你要知道一个drawcall可不只是绑定一个shader。

**除了shader变体的检索，渲染状态的设置同样会影响性能。**该功能跟shader变体一样也要组合出一个hash值，根据这个hash值定位到不同的渲染状态对象，如果发现不存在就创建出一个新的，因此不同的drawcall能够共享相同的渲染状态对象，有效的规避了部分渲染状态的频繁切换。

unity还有一个比较坑爹的问题，就是纹理资源的绑定也会有开销。因为引擎内部为了纹理对象的访问安全，强制纹理的操作都要通过纹理句柄来做，函数间传递的也是这个句柄。本质上说这个句柄就是一个弱对象指针，当它需要访问具体的纹理指针时会根据句柄去搜索纹理对象库，这样就不用担心访问到野指针了，但搜索开销却不容小觑。

按理来说对于纯静态的模型，也就是那些材质属性不会随着时间发生变化，且世界空间位置也没有改变的物体，它们的constant buffer应该也是静态不变的，可以预先构建好，不需要每帧都进行更新。但是很多引擎包括unity，还是选择每帧对其进行更新，这会让constant buffer里的数据被反复填充。虽然看起来有点暴力，但这种方案也有好处，就是constant buffer的实例不会太多太分散，buffer绑定所造成的状态切换也因此减少。

总体来说unreal的设计稍微合理一些，**它将constant buffer按照每帧的使用频率分层级进行管理，每帧都会变化的shader全局参数放在一个专门的buffer里，**例如帧时间，摄像机位置，随机种子等。**那些随着pass变化的数据则是放到另一个buffer里，**例如视距阵，投影矩阵，因为gbuffer和shadowmap，或者reflectionmap，这些pass它们的摄像机位置都不一样。**而对于静态的材质参数数据只需要构建一次就可以了，它会放入一个静态的constant buffer里，不需要每帧进行更新，那些每帧会发生变化的数据则被存放在动态的buffer中。**由于引入了新的draw mesh pipeline，unreal会为primitive的属性建立一个buffer统一管理起来。primitive的相关属性包括世界空间的变换矩阵，包围盒信息等，对于静态的primitive这些数据就不会每帧做更新了，而且还能做动态的drawcall batching，真是一举两得。但是primitive的场景数据并非按照struct of array的模式做的内存布局，所以在读取的性能上会有一些损失，因为cache容易miss。

我想到一个历史故事，当年我们在移植d3d12的时候就在dynamic constant buffer上栽了一个跟头。因为Map的功能需要应用程序自己去实现，我们必须事先在upload heap中分配出一大块内存，然后每帧要用一个fence去追踪这个buffer的使用情况，判断这个buffer何时可以重用。否则如果每次填充buffer时都分配一个新的buffer，内存肯定吃不消。因为gpu的任务对于cpu都是异步的，所以我们可以把fence作为一个信号指标，当轮询到这个fence处于已通知状态时，就说明fence之前的所有渲染命令已经全部执行完毕了。那么与此同时它上面绑定的资源也就不会再被占用，应用程序又可以重新填充这个buffer，并绑定到新的drawcall上。d3d11的map函数的内部实现基本也是按照这个方式来设计的，所谓动态资源的renaming。

讲了半天我还没有提及那个问题，大多数情况下在d3d11里填充constant buffer时我们会用到map\_write，map\_write\_discard或者是map\_write\_no\_overwrite这些标记，因此它要求访问这个buffer时尽量是只写的，最好不要有读操作，否则会有性能上的惩罚。之前我们对此不是那么重视，所以在写代码时不会专门检查这类问题。由于很多的写操作是很隐蔽的，如果只是粗看代码，一时半会很难反应过来，但是仔细分析过反汇编的代码后才意识到确实是先从buffer读出了之前的结果，经过计算后再写回去的，却误以为只是一个纯粹的写操作。在用d3d11开发时我没有感觉这些读写过程会对性能有多大的伤害，但是在d3d12里却成为了一个显著的问题，居然变成了性能的热点，当时看到vtune的数据时差点没把下巴给吓掉。其中的罪魁祸首就是d3d12的upload heap中的内存块使用了page\_writecombine的保护模式，它可是反cache的，不会主动的维持cache一致性，所以数据的读取会相当慢，至少低一个数量级以上。

关于shader绑定的效率问题有其他的解决思路。unreal和cryengine不谋而合，由于需要支持d3d12的接口，它们都抽象了一个类似于pso的对象结构，所以即便是d3d11也有这种渲染状态聚合物的对应实现，只不过最后往设备上下文设置时才转换成实际分离的接口调用。正因为采取了这种设计方法，引擎就可以事先把所有关联的shader变体都cache到这个对象里缓存起来，而不需要每次应用时再对shader变体进行搜索。当然对象结构也包含了其他的渲染状态，例如光栅化状态，深度模板状态和混合状态等。只不过这种方式浪费了一些内存，属于用空间换取了时间，但何乐而不为呢？！

由于硬件的遮挡测试需要回读GPU的数据，所以在Culling阶段Unity访问这些数据并不是那么的方便。因为它和CryEngine一样，Culling的处理都是在主线程里完成的，只不过真正的计算会由主线程Dispatch到任务线程里完成，但本质上这些任务还是由主线程控制发起的，主线程需要确切的知道回读的GPU数据是否已经到达CPU端，如果发现没有就绪，那么Culling任务就不能启动。为了避免每帧中不同的时间点在渲染线程里多次轮询Readback Buffer是否已经Ready，明显这种做法基本上是不可取的。而且由于Unity的Culling逻辑和渲染线程是分离的，它不能直接访问渲染线程里的设备上下文，所以我们会在渲染线程每帧Present的后面强制做一次同步的Map，使得Readback Buffer的Copy命令(把数据从Default Heap传输到Readback Heap中)在这个时刻点必须执行完毕，否则一直做空等待。当Map成功返回结果，就把这些数据拷贝到一个主线程能够读取的Buffer中，等下一帧跟渲染线程与主线程同步时，主线程的逻辑就可以安全的从这个Buffer中读到GPU的数据了。这个方法看起来有点暴力，但还是行之有效的。

这里稍微做个细节上的补充，因为Readback堆的内存使用了Page\_WriteBack这个模式，所以它可以让Readback的数据区不管做读取还是写入的操作都不会影响性能。之前我解释过Upload堆的内存为什么不能去读取它的数据，Readback堆的内存特性跟它恰恰相反，它是Cache友好的。但值得注意的是，对于能够保证Cache一致性的UMA(例如某些集成显卡)，Readback堆和Upload堆的Page属性都选择了writeback，换句话说就是upload堆也可以自由的读取数据，而没有性能上的惩罚。

为了避免帧率的频繁抖动，一般GPU会缓冲最多三帧的渲染命令用来做平滑。也就是当调用Present结束时，GPU并没有立刻完成当前帧的渲染工作，除非遇到了V-Sync事件。这时如果像我之前所说的那样，在Present函数的后面直接对Readback Buffer做一次同步的Map，那么Driver就会马上将Copy操作之前的所有的渲染命令(可能这时命令还没有压入到硬件的Queue中，甚至也没有完成Translate)，也包括Copy本身全部Flush给GPU，然后原地等待GPU完成这些渲染指令。可以想象这种野蛮的同步操作打破了Driver的并发性，让GPU缓冲命令和负载均衡的美梦瞬间破灭。但是类似遮挡查询结果这种对时间周期比较敏感的数据，如果不及时回读并应用，那么就会带来很大的副作用，引发高概率的False Positive和False Negative的问题。所以延缓Readback Buffer的读取并不现实，只能寄希望于拷贝点和回读点尽量离得远一点，因为这样就有充足的时间留给数据的计算和拷贝，可以节省Map时做空等待的周期，但理论上再远也不会超过一帧的时间。

为了继续降低每个渲染帧的显示间隔，Unreal还会把渲染线程的渲染命令发送到一个RHI Thread中。这个线程专门把抽象的渲染命令翻译成图形API的具体函数调用，从宏观上看相当于做了一个三级流水线，分别处理逻辑模拟，渲染命令生成和渲染命令翻译这三个独立的任务。如同我在另一篇文章里分析过的原因，这种把帧处理流程分割成若干个子步骤的方法其实并没有减少延迟(我所谓的延迟是指从玩家输入到对应的画面输出的时间间隔)，而仅仅是提升了帧率，相当于我的输入要等到三帧以后才能看到结果，反馈的时间长度依然没有发生变化，但是每帧画面的显示间隔确实变为了原来的三分之一左右。

曾记得我在做d3d12移植时也碰到过类似的问题，当时很奇怪为什么用d3d12的api替换了d3d11的api后性能没有发生太大的变化。按理说d3d12的接口是很高效的，因为它的实现相对比较轻量，没有那么多繁琐的校验逻辑(由于d3d12失去了强大的异常保护，所以稍有不慎就很容易导致程序崩溃)，而且我们还做了大量定制的优化。但经过多次压力测试，有个别时候居然会落后于d3d11的性能。那时怎么也想不通，后经高人点拨，这才明白原来driver会有一个专门的线程去处理runtime发送过来的命令，包括将shader字节码转译成机器码也会有独立的线程负责。但是到了d3d12时代，driver的功能变得越发的单薄，很多事情都交由应用程序去处理，driver不再负责了。那些处理runtime命令的线程也被取消，d3d12的runtime会直接操作driver的核心函数。问题的原因找到后，解决起来就有方向了，为了模拟driver在d3d11中所做的行为，我们也弄了一个命令队列，它就像RHI Thread那样变成了异步并发的模式，从此帧率得到了明显的提升。

还有一个类似的事情也能说明并发的好处，那就是把多个GPU串联在一起做的交错帧渲染方法(Alternate Frame Rendering)，每一个GPU就好像是一级的流水线，例如两个GPU在一起工作，那么第一个GPU可能专门负责渲染奇数帧，而另一个GPU则负责渲染偶数帧，它们的处理是交替进行的，互不依赖。当然除了AFR的模式，其实还存在一种叫做Split Frame Rendering的模式，SFR就是把多个GPU并行起来处理同一帧的数据，例如GPU A处理屏幕的左上角，GPU B处理右上角等。这里显而易见SFR才能真正的在提高帧率的同时去降低延迟，但它的实现却比AFR复杂很多，任务的分割和调度极难处理，往往多个GPU的利用率会参差不齐，而且如果需要GPU之间传输一些中间数据，还会给带宽带来额外的开销。

很多人认为之所以要用并发来处理渲染逻辑是因为GPU的计算独立于CPU，但我不这么认为，虽然这两个硬件所组成的系统确实是异构的。主要的原因还是由于每一帧的逻辑都有时序依赖的，不能打乱顺序执行，必须先做完模拟后，才能进行渲染。那么对于这种长任务就只有通过并发改造才能提高帧率，当然中间还可以将一些局部无时序关系的逻辑并行起来，例如后文即将提到的并行渲染命令处理的功能。

除了并发的优化手段，我们还可以利用多任务并行去加速渲染流程。我之前也分析过，只有并行才能真正拯救延迟的问题，大多数游戏需要是低延迟，快响应，而不是那些骗人的高帧率。但是并行的渲染功能基本都需要Graphics API的原生支持。因为不管是unity还是早期版本的unreal，它们的并行渲染架构都没有做到真正意义上多线程同时构建硬件的渲染命令，而只是一种近似模拟，并行处理的是引擎抽象封装的渲染命令，但即便如此也比串行的过程快很多，因为引擎的渲染命令转换成runtime的函数还有很多额外的工作要考虑，例如之前提到的shader的搜索，资源的绑定和buffer的填充等任务。

由于底层使用的是high level的图形api，所以根本没有办法在不同的线程里同时访问runtime的设备context，除非给每一次的访问都加上一个互斥锁，才能保证它的线程安全性。但要是那样做的话会适得其反，由于锁的频繁碰撞导致处理速度变得更慢了。opengl就是这样设计的，它不允许不同的线程调用它的接口，调用接口的线程必须和创建context的线程保持一致。d3d11之所以可以支持多线程的接口访问，是因为它内部提供了线程安全的运行模式，估计也是通过加锁来防止临界资源的恶意竞争，所以一般情况下大部分引擎都只会选择单线程的模式。

为了更好的支持多线程渲染的应用开发，d3d11还提供了一个延迟context的机制，它允许应用程序并行的收集渲染指令，这些指令会被临时缓存在延迟context里，最后再把它们提交到立即context里执行，延迟context是没有办法自己直接去执行这些命令的。当然deferred context也不是什么都不做，它也能执行一些简单的校验工作，另外填充动态的constant buffer同样没有问题。d3d11的延迟context的工作原理还是与d3d12有较大的区别，因为d3d11的延迟context并不能在工作线程里完成硬件指令的翻译操作，而是要等把它们放到立即context里执行时才会真正开始构建硬件的渲染指令。所以我猜测延迟context记录的还是一些渲染命令的中间状态。我们知道多个shadowmap的渲染其实是可以并行的，因为它们之前没有任何的逻辑耦合，此外shadowmap的渲染和gbuffer也是没有依赖的，reflectionmap同样也与它们没有冲突，所以这些pass的渲染命令收集和执行完全能够并行起来。我记得最早利用d3d11的延迟context特性的游戏就是total war，这个游戏里有大量需要渲染的角色和物件，如果是串行的填充渲染命令帧率肯定会非常的低。

这里打个也许不是那么恰当的比喻，类似高级语言的编译器，它会为每一个源码文件生成一个对应的目标文件，而这些Object文件里面存放的只是些临时的中间结果，还需要通过链接器将它们装配在一起并转换成机器码才能运行。由于中间文件里有许多的外部依赖和引用，在单独编译这些文件时还没有办法全部确定，所以进行全局优化也要等到代码链接的阶段。我们在d3d11的延迟context里所做的api调用，其实不过是做了一些预处理和合法性检查的工作，这与编译器生成中间文件的过程很相似，难道它不也是在做同词法和语法分析差不多的工作吗。

对于像d3d11这种高级图形api来说最主要的问题还是状态的设置太零散，虽然人类理解起来很合理清晰，但硬件里的对应概念却是聚合在一起的原子操作。各种渲染状态，不同阶段的shader绑定，primitive类型，顶点布局以及渲染目标格式等，它们在硬件中是以pipeline state object的形式存在的，是一个不可分割的整体，因为上下游的数据传递是环环相扣的，所以必须通盘考虑，例如上游的输出属性要与下游的输入属性对应起来。还有就是游戏runtime的某些渲染状态设置可能要转换成shader的一部分内部代码，这个也得driver帮忙。所以应用程序设置到d3d11中的状态会被driver编译并打包生成一个个互相独立的pso，如果状态集合里的状态全部一样则会共享同一个pso，因此driver还要负责查找相同的状态集合对应的pso，以避免重复创建。正是由于d3d11接口的这种设计，导致了延迟context想把渲染状态直接翻译成硬件pso的目标不能达成，因为此刻可能有一些渲染状态是当前这个context不知道的，虽然这些状态已经在另外的context被设置了。我们知道opengl也是一种高级的图形api，新版本里面有一个pipeline object的概念，但那只是把program聚合在一起，对于硬件管线来说并不完整，所以依然要通过driver来做转换。由于d3d12的pipeline state object中，上下游不同阶段的信息是全面的，因此driver可以针对不同的硬件环境做最优化的处理，而且这些pso的机器码还能缓存起来，下次启动程序时可以直接从文件里读取，而不必每次都进行构建。这种做法节省了大量的运行时成本，否则一旦出现大量pso的集中构建就会引起游戏卡顿。

这里还要强调一点，硬件里处理渲染命令并不存在多个不同的命令执行队列，其实就只有一个。driver内部也是通过ring buffer进行命令的上传，然后再由gpu完成任务的后续调度。所以不管是d3d12还是d3d11都仅能靠同一个队列提交渲染指令，即便它们可以并行的去构建这些命令，只不过d3d12的runtime基本不会帮助应用程序去校验渲染状态的有效性，而是要求上层逻辑自己保证。另外就是刚才说到的pso，它们只能由应用程序自己维护和创建，这样多个command list就可以独自完成硬件指令的编译和构建了，不需要等全部放在一起后才能开始做。再补充一下，刚才提到渲染任务队列只有一个其实严格来说也不准确，硬件上会有三个独立的任务队列，一个是处理3d的队列，它包括所有类型的命令，涵盖了光栅化，compute以及数据拷贝。第二个队列是专门负责处理compute命令的，里面也可以有拷贝命令。而第三个队列则是专门执行数据拷贝的任务，它的命令类型最单纯。这些队列之间的任务如果有时序依赖，那么就需要通过barrier和fence进行同步，以保证数据访问的安全。上述这些功能也是d3d11所不具备的，很多对硬件行为的细节控制在d3d11看来基本都是完全透明的，很多信息的传递也是笼统和含糊的，它只能依靠driver心领神会来完成指定的操作，所以我们才说这类的api属于高级api，类似于汇编语言和高级语言的区别。

之前说到GPU有三个主要的工作队列(分别对应三个硬件引擎)，3d命令队列是一个万能的队列，里面可以执行任何类型的命令。d3d11由于不能控制设备context使用哪个命令队列，所以在性能优化上会有许多的限制。本质上d3d11的context都是基于3d引擎的，因此即便是创建再多的deferred context也于事无补，它们都不能并行执行。我在cryengine源码中看到系统为texture streaming专门创建了一个deferred context，想用它来负责纹理数据的上传和拷贝，而不希望这些操作影响到渲染绘制的命令执行。但估计现实会事与愿违，如同我分析的一样，这些上传和拷贝的命令要是和其他的渲染任务混合在一起放在同一个3d命令队列里执行，它肯定会妨碍渲染命令的工作，即便这些渲染命令并没有引用这些纹理。因为3d命令队列里的command list每帧都会进行同步，要求之前放入队列的所有命令必须同时完成。但大多数情况下，并不是全部纹理数据的上传和拷贝的命令都必须在当前帧结束。其实texture streaming的处理完全可以跨帧执行，对时效性要求没有渲染指令那么高，延迟几帧，甚至几十帧都没有太大的问题。不过也许driver会根据一些外部的提示来把这个专门处理纹理streaming的deferred context放入到copy引擎里执行，但那要看底层是否提供类似的支持了，这种事情只能听天由命，因为d3d11不是显式可控的。我猜测大概只有当DriverConcurrentCreates这个特性被驱动支持时，且在CreateTexture2D调用中就把纹理数据借由初始化参数传入函数，才能让copy engine生效。记得早年我在为cryengine移植d3d12接口时就给texture streaming功能设计了一个特殊的工作队列，这个队列里的操作会放入到copy引擎的命令队列里执行，和渲染的命令队列互不干扰，只是需要通过设置fence和barrier进行同步。

texture streaming中之所以会有显存对显存拷贝的操作，是因为当新的mipmap流入时，原来分配的纹理对象里的mipmap数量就会不足，所以需要另外创建一个拥有合适mipmap数量的纹理对象，这时旧纹理对象里的mipmap数据就可以直接拷贝到新的纹理对象中，而不必重新从主存上传到显存。毕竟显存内的数据拷贝更加快速，其他旧纹理不存在的mipmap数据则需要从主存里读取，并通过upload heap的buffer上传到显存中。另外由于主存里的纹理数据布局与GPU期待的不一样，所以还需要对其做swizzle变换(把row-major的布局改成内部的特殊布局)，这需要额外的处理时间。显然纹理mipmap流出时也会经历反向的过程，只是不用再上传数据了，但新建和拷贝的操作必不可少。

对于主机平台其实也并非一定要在上传的阶段执行swizzle变换，因为目前的主机硬件系统都采用了uma的架构，既主存和显存共享一套内存单元，所以在内存里的数据可以同时对gpu和cpu可见。而且主机gpu端的纹理布局也是确定的，不像桌面端那样需要兼容不同厂商的格式，关键很多厂商出于保密的原因，一般不对外透露它的纹理内存布局。可主机就不一样了，它是一个封闭的生态圈，所以厂商会向开发者透露所有必要的硬件实现细节，因此你可以通过sdk的接口事先将纹理数据转换成gpu端的布局格式，并保存在文件里。等运行时加载到内存里后就不用再进行转换了，节省了不少的时间和能耗。其实桌面端的d3d12的sdk也提供了一个标准的swizzle格式(Z-order curve)，只不过大部分厂商都没有明确声明这个标准布局就是它们硬件内部原生支持的格式。所以估计GPU还会对上传的纹理数据进行重排，否则访问效率就会变低，重排数据有利于提升cache命中率。

我心中一直有一个疑惑，就是手机端也用的是uma的架构，但似乎硬件厂商并没有公开gpu端的纹理布局格式，当然也没有sdk可以对其进行离线转换。很显然swizzle的操作是要消耗一定量的带宽的，那么手机端对于能耗这么敏感，按理说事先转换布局有诸多的好处，这是一个稳赚不赔的买卖，何乐而不为呢？！

之前系列文章中会有一些遗漏且不足之处，我会统一在补遗的文章里做出额外的说明，并把最新的一些思考也记录其中，希望对大家的实践能有所启发和帮助。

下面是关于multi-engine的一些新的观点和看法。先前的描述可能不是那么的严谨，这里算是与时俱进的做些补充。请结合早期的文章一起阅读，有了上下文，也许理解起来更容易一些。

微软似乎意识到d3d11对于multi-engine的支持太过于简单了，缺乏很多显式的控制能力。于是它在后期的版本中逐步对此进行了若干的加强，例如新增了CreateDeferredContext3接口，它能够创建一种新的Context，这个Context拥有把命令Flush到不同Engine的能力。甚至在ID3D11DeviceContext4这个接口中还加入了Signal的功能，它可以在完成命令处理后发信号给Fence。是的，D3D11.3也能创建Fence了，这样ImmediateContext和D3D12的CommandQueue就基本可以等价视之。

当初我说过在d3d11中不方便将CommandList单独提交到CopyEngine或者ComputeEngine中，引入新的Context和Device后(ID3D11DeviceContext3)，这些问题就迎刃而解了。例如TextureStreaming可以在异步工作线程里把Copy操作Flush到D3D11\_CONTEXT\_TYPE\_COPY的队列中，而并不一定非要在渲染线程里执行。同时Query也能创建和执行在不同的ContextType上，有了这些特性的帮助，我们就可以在主线程或者渲染线程里，通过轮询或者强制等待Event对象来断定先前的队列里的命令是否已经处理结束。因为这些Event会被插入到Copy命令之后，只有Event对象收到完成通知了，我们才能放心的把这些纹理绑定到ImmediateContext中。否则如果这些新建的纹理依然在Copy Engine中执行，同时它又被Shader访问到，那肯定会造成数据冲突，并引起系统的异常。

除了Copy Engine的异步化，我们还能利用Compute Engine做一些异步的通用计算任务。如果能把一些计算任务与当前在执行的渲染命令重叠起来，那么就可以充分利用GPU的处理单元，让它们的负载始终处于饱和的状态。例如3D Engine在渲染Shadowmap时，大多数ALU单元和纹理采样单元是闲置的，它对ROP和Raster单元的依赖比较强。于是我们可以将一些计算密集型的任务放置到Compute Engine中，与Shadowmap的Pass同时执行。显然此种安排对于提高GPU处理单元的利用率是大有裨益的，而且还缩短了每帧的执行时间，这就是计算并发的好处所在，只不过它消除不了延迟，上一帧的计算结果下一帧才能应用。

之前谈到d3d提供了一个标准的swizzle模式，这个模式是可能有硬件支持的。通过查询D3D11\_FEATURE\_DATA\_D3D11\_OPTIONS2里的StandardSwizzle属性，可以得知该设备是否支持标准的重排模式。而只要硬件支持，就能够使用CreateTexture2D1函数去创建一个满足该模式的纹理对象出来。由于我们在CPU端可以事先对主行序的纹理数据按照StandardSwizzle的要求进行重排，那么后面利用Staging纹理做拷贝时就不再需要对数据进行重排操作了，自然节省了Upload的执行时间。

这里我还想借机辨析一下D3D11\_QUERY\_EVENT与D3D12中Fence的关系，因为在D3D12中，如果希望单独知道某个命令是否已经处理完毕了，是很困难的。而D3D11的D3D11\_QUERY\_EVENT，它可以被插入到Context上的任意一个命令之后，通过GetData接口，你就能轻松判断这个命令及这个命令之前的所有命令是否已经完成了。但是在D3D12中，你只能以一个Command List为最小单位去检查命令是否已经被执行完毕，也就是说当一个或者若干个Command List被放入到Command Queue中执行时，可以在调用完ExecuteCommandLists之后，通过Signal插入一个Fence。这个Fence能够判断传入ExecuteCommandLists的所有Command List是否全部结束了。假设想用D3D12模拟D3D11的D3D11\_QUERY\_EVENT，那只能把D3D11\_QUERY\_EVENT之前的命令放入一个Command List，它后面的命令又放入到另一个Command List，第一个Command List先用ExecuteCommandLists去处理，接着通过Signal插入Fence，最后才用ExecuteCommandLists执行第二个Command List，这样一来就能达到D3D11的相似效果了。但明显上述方式是相当麻烦的，如果插入的事件比较多，那么Command List就会被切割得非常的零碎。不过也许D3D11的Runtime不会那样去设计，分割Command List的方法感觉有点愚笨，我猜测它是在每个Present之前且本帧所有命令调用结束之后才会放置一个Fence。因此即便Query发生在渲染命令的中间，也会通过等待刚才提及的Fence，确认Present之前全部的Command都被执行完成了，才让GetData有效，而不仅仅是Query之前的命令执行完毕后。这属于一种常见的batch化的处理方法。

综上所述，Fence的同步粒度是很大的，它会关联到某些硬件的中断上，不像ResourceBarrier，ResourceBarrier可以做逐个命令的同步，所以十分的轻量级。可是ResourceBarrier并不支持CPU端访问，它是一个纯GPU的对象实体。OS可以利用Fence完成Command List的调度，因为我们可以把Fence看作是一种依赖关系的分界线。当硬件不支持多Engine的Command Queue时，OS可以把不同Command Queue中的Command List按照Fence规定的依赖顺序平展合并成单一的Command List，放入同一个Engine里去执行。除了D3D11\_QUERY\_EVENT，其他的D3D11的Query也都需要利用Fence才能确认Readback的操作是否已经执行完毕，过程是类似的。

D3D11的Command List是何时被放入到Command Queue中执行的呢？大约是在SwapChain的Present方法调用的时候，而Immediate Context内部应该有一个Command List的Pool。当Present调用时，当前被绑定在Context的Command List就开始执行Close函数，即做抽象命令的翻译，并生成硬件能识别的指令。然后使用同样放置在Immediate Context内部的Command Queue，执行它的ExecuteCommandLists方法。当下一帧Immediate Context准备重新工作时，Context会查询在上次ExecuteCommandLists调用后插入的Fence。如果Fence处于完成状态，则Reset这个Command List，随后就可以往这个Command List填充新的命令了。一旦发现Fence仍然处于未完成状态，表示前一个Command List中的命令还没有被GPU全部取走，就从Pool中申请一个新的Command List，把它绑定到Context上。这样看来ExecuteCommandLists都是隐式被调用的，那有没有显式调用的地方呢？我想是有的，那就是Flush函数。

之前的文章里我说过，驱动是以异步并发的形式去构建硬件命令的。所以Command List会被派发到另一个工作线程里做构建，然后等到下一帧时再调用ExecuteCommandLists去执行上一帧构建好了的Command List。当前帧的Command List的填充和上一帧Command List的构建可以在时间线上重叠起来，因此Command List从填充到被执行至少要延迟两帧。

虽然d3d11的内部实现是一个黑盒子，但是我们能用d3d12的概念去做类比和分析，因为d3d12已经取代了原来驱动的部分功能，并且和硬件的底层结构很接近了，所以这个推测的过程应该八九不离十，大致是相仿的。

早前提到D3D11引入了一个新的Signal函数，它能与Fence结合在一起使用。除此之外我还讲过Fence与原来的D3D11\_QUERY\_EVENT有许多共通之处，按理说并不需要加Fence这个看似冗余的新概念进来，因为之前的功能已经完全够用了。那到底是为什么呢？个人认为是由于微软想让D3D11全面支持multi-adapter，用这个新引入的fence就可以实现跨设备的命令同步。其实先前版本的device类已经能够创建跨设备的资源，通过调用OpenSharedResource函数，在不同的adapter中引用相同的资源。但是应用程序还需要确定两个设备在进行数据交换时，拷贝命令何时结束。之前是没有什么手段能够达成这一目的的。因为数据从设备a拷贝到设备b，只有跨设备的fence才有能力监控两方的操作是否都已经完成。这就有点像socket在跨进程同步数据时所做的工作，不仅要了解接收方的情况，也要清楚发送方的状态，那样才能确保数据完整的到达远端。