【Unity】SRP底层渲染流程及原理

王江荣
学习是一个发现自己有多无知的过程

已关注

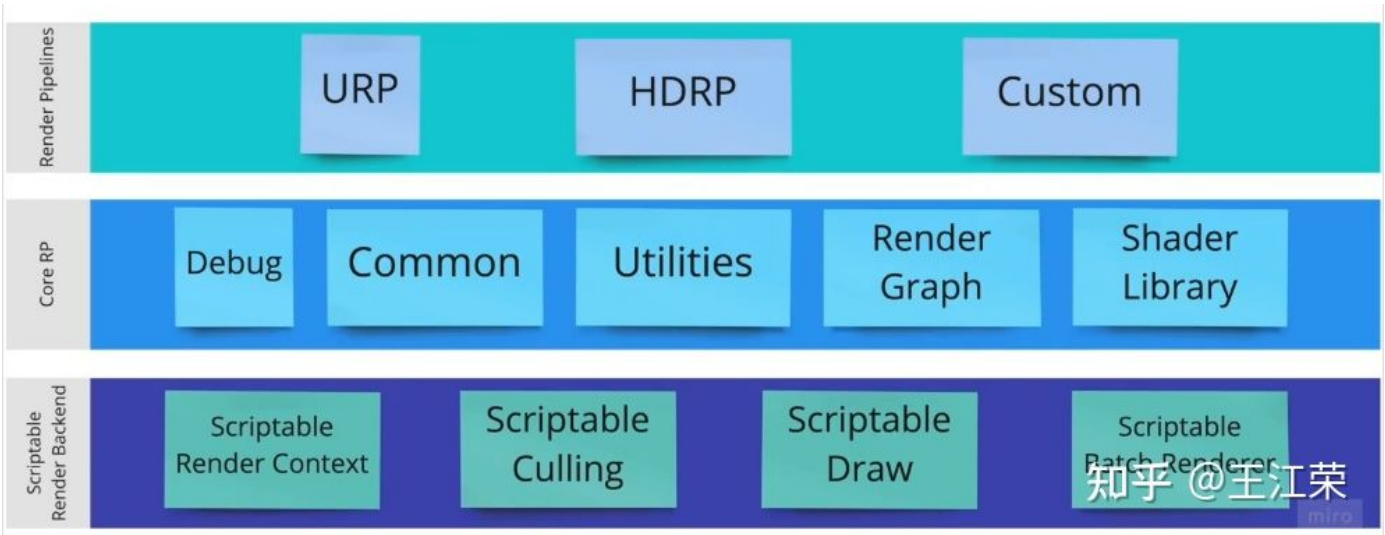
前言

近期的Unity官方分享会为我们介绍了SRP底层渲染流程及原理，本文进行一个简单的记录，也顺便加深自己的理解。

SRP简介

SRP的简单架构

整个 SRP 的简单架构如下图：



最上层为**Render Pipelines**，有我们常见的 URP 跟 HDRP，还有自己去扩展的一些自定义的渲染管线。

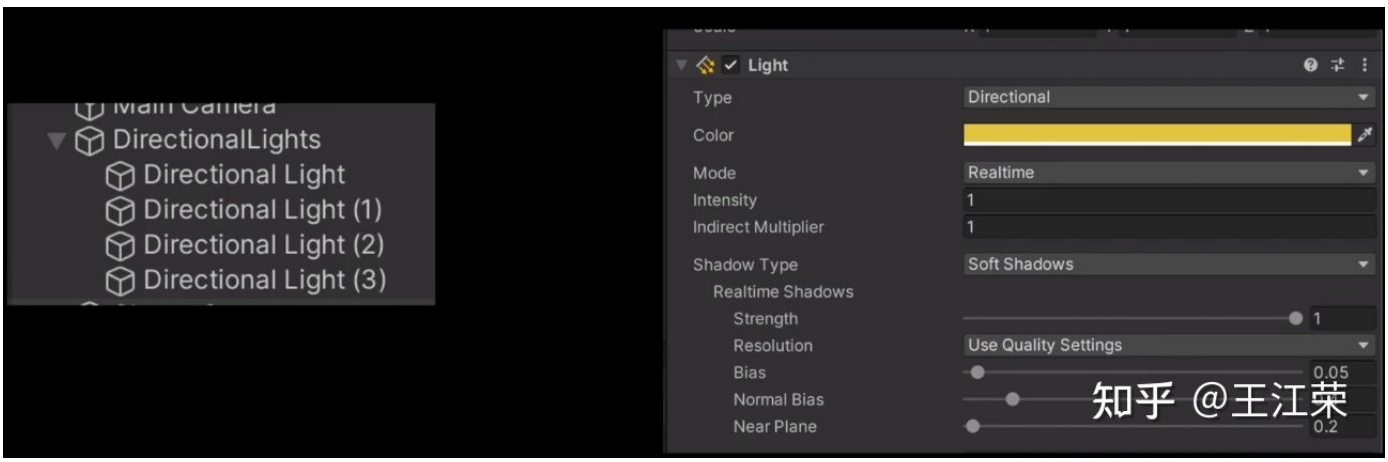
中间层为**Core Render Pipeline**，RP层依赖于该层，为我们提供了一些Common库，Shader Library等。

最下层为**Scriptable Render Backend**，属于c++层面，包括有Context，Culling和各种Draw函数，以及SRP特有的SRP Batcher。

其中Scriptable Render Backend层对大部分开发者来说是一个黑盒，unity本次分享的目的就是为了帮我们把这个黑盒打开，个人也在这记录一下，防止时间一久这个黑盒又对我关闭了。通过理解 Unity 在底层做了什么事情，我们更好的优化自己的项目（使用了SRP，URP或者HDRP的项目）。

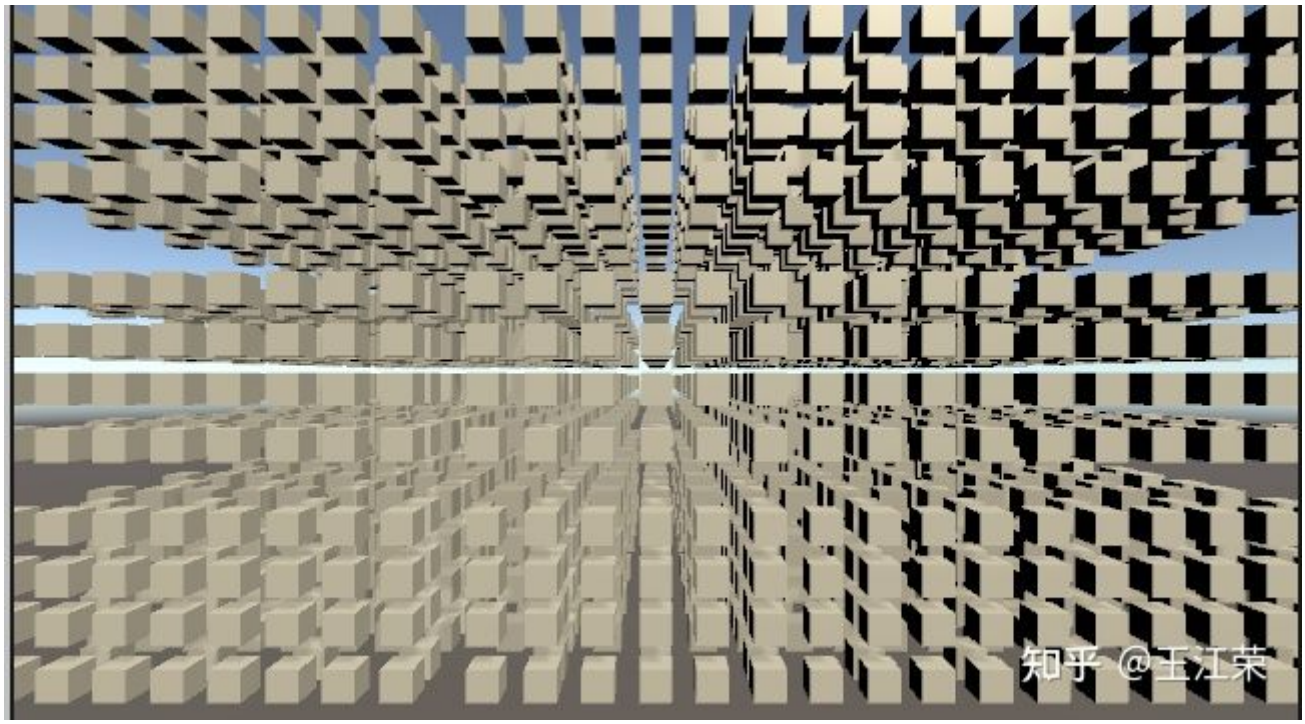
Render Loop流程

我们先模拟视频里说到的Demo，其实很简单，使用URP或者HDRP的环境即可，因为他们的都是在SRP的基础上扩展的。新建一个Scene，里面随便放四个平行光源，都设置为实时光并且开启软阴影，如下图：



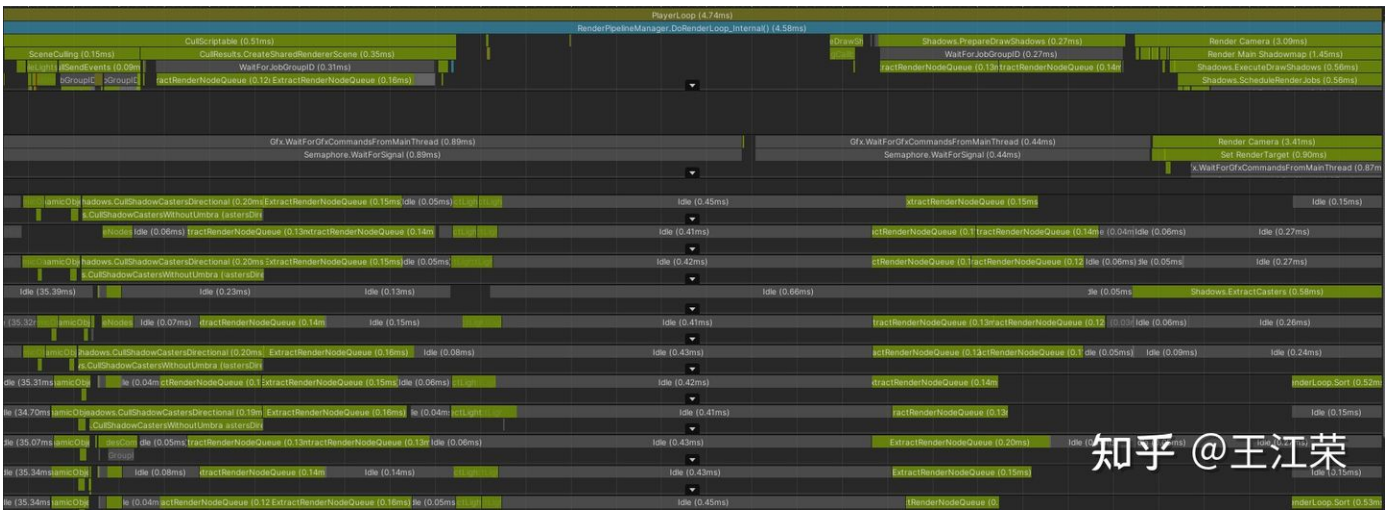
然后在场景中摆上茫茫多的Cube，或者其他复杂的场景，然后它们的Mesh Renderer都要可以产生阴影，如下图：

这样我们的Demo就搭建成了（如下图），low是low了点，但是不影响学习嘛。



Demo场景

我们运行一下，看看Profiler 的 Timeline 视图：



Profiler

基本上可以看到整个Render Loop的过程，Render Loop的官方解释为：

> A render loop is the term for all of the rendering operations that take place in a single frame.

Profiler 中各个节点代表着在一个Render Loop中各自负责的模块，其前后顺序自然就是每个模块的先后执行顺序。其中大部分模块都是通过**Job System**来实现的，也就是说我们的SRP本身是一个多线程的渲染。

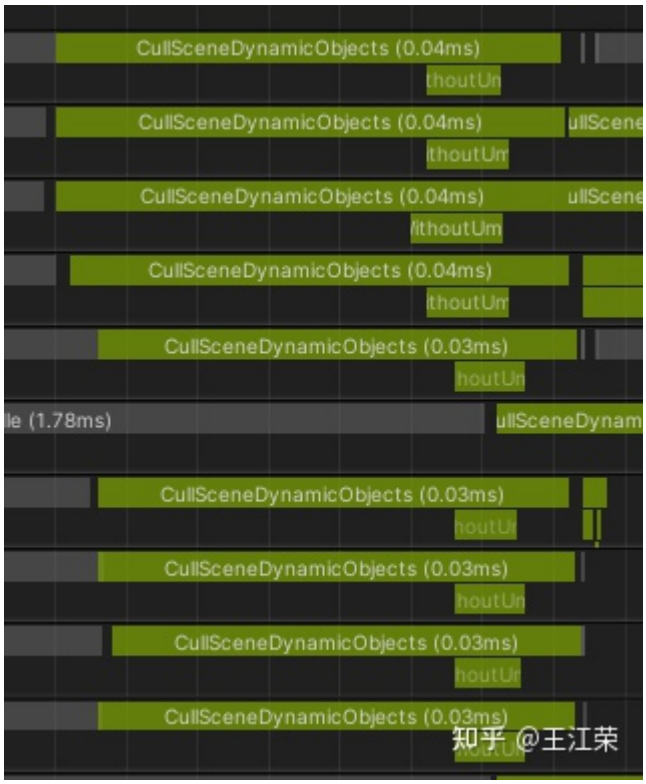
本次分享主要提到了有CullSceneDynamicObjects、Shadow.CullShadowCasterDirectional、ExtractRenderNodeQueue、RenderLoop.Sort、RenderLoopNewBatcher.Draw、SRPBatcher.Flush这些节点。

Scriptable Culling

我们先来看下Profiler中最前面的有关剔除的部分，其实就是我们在调用 **ScriptableRenderContext.Cull** 方法时，Unity 底层会做的一些事情。

Cull Dynamic Objects

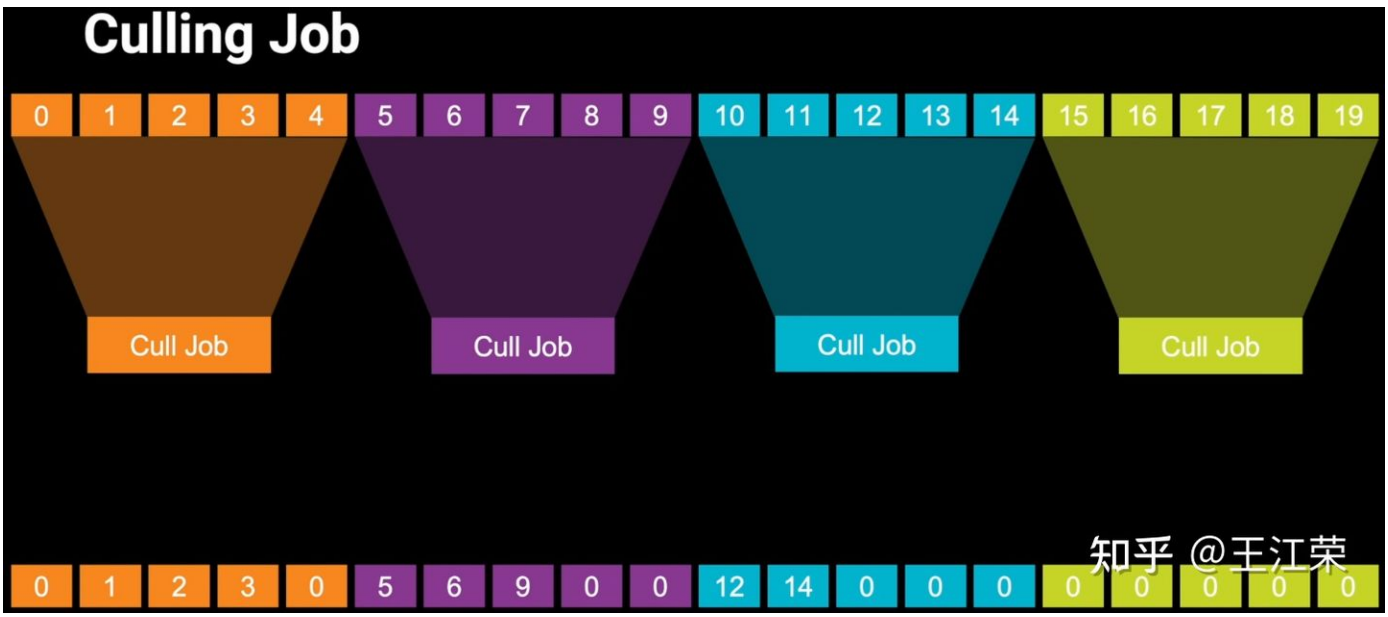
首先最前面的是CullSceneDynamicObjects，这部分其实是在**裁减场景里面的动态物体。**



CullSceneDynamicObjects

这几个 job 会产生一个名为 **IndexList** 的数据结构。对于场景中所有的物体（Renderer），Unity 内部维护了 Renderer 的列表，而我们的 IndexList 就会存储当前**可见的**所有 Renderer 的数组下标。

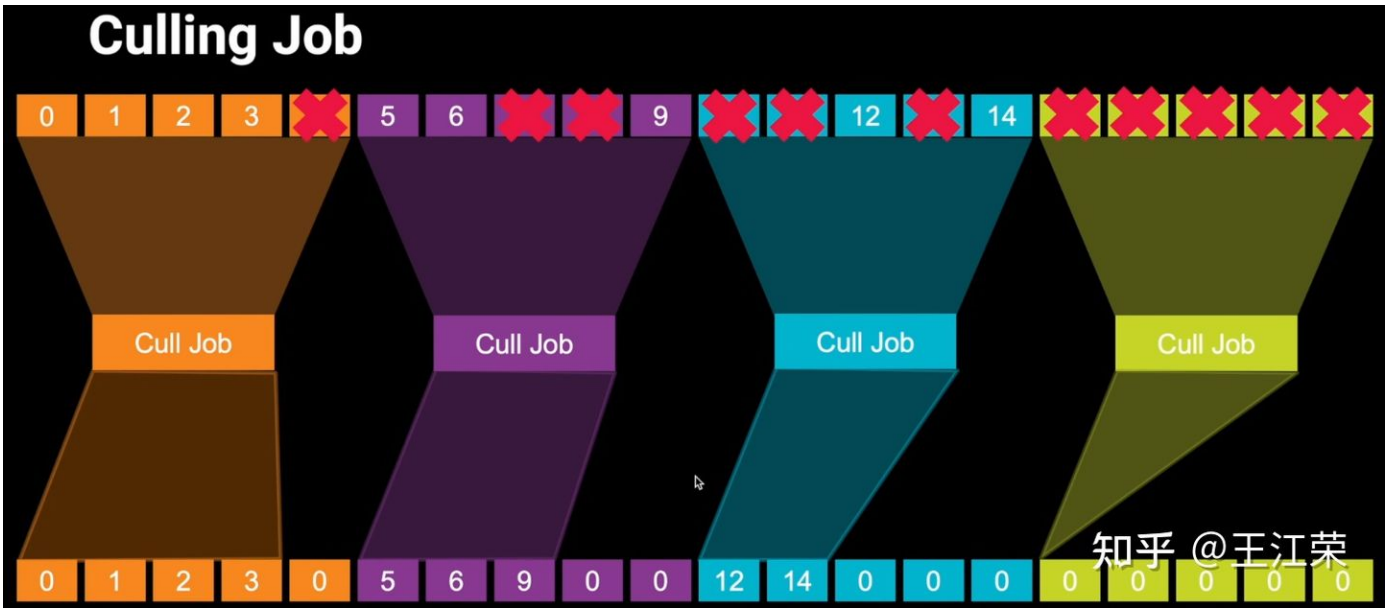
我们通过一个例子来看一下这些 job 是怎么工作的，如下图：



图中上面的部分从 0 到 19，这些其实就是 Unity 内部维护的Renderer的list，也就是场景中所有的Renderer。然后我们产生了4个 Job，分别处理 List 里面不同

的部分。而底下的部分就是我们 IndexList，这里能看到一个特点，就是**IndexList 跟 Renderer 的 list 是等长的**。

我们每一个 Job 处理裁减的时候，比如橘色这个 Job ，假设下标为4的Renderder是不可见的，那么 Cull Job 就会把 0、1、2、3 写到 IndexList 里面去。然后假设下标为7和8的Renderder是不可见的，那么紫色的 Job 会把 5、6、9 写到 IndexList 里面去。后面的Job操作类似，如下图：

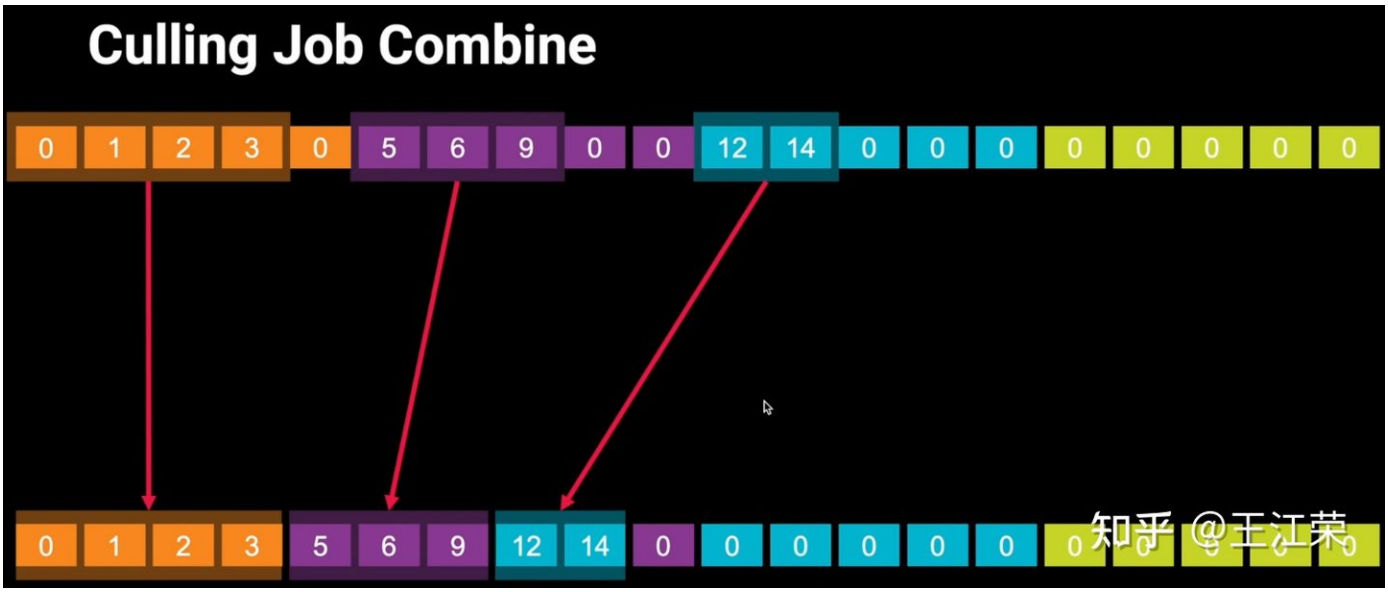


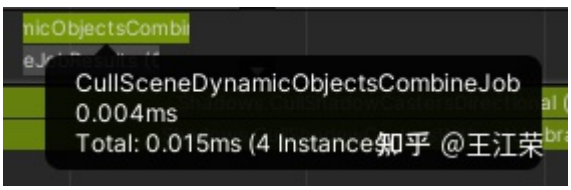
可以发现上面的操作中，并没有直接在 3 的后面插入我们的 5、6、9，而是写入在原本属于自己的那部分区间里，后续的Job也是如此。为什么要这么做呢?

这是因为Unity中每一个 Job 都是在不同的线程执行的，而上面每个Job处理Renderer的list时，互相之间没有任何的交集，即不会多个Job处理到相同下标的Renderer，因此就**不需要引入锁的概念**，它们访问的所有的数据都是线程安全的。

当我们写入 IndexList 的时候也是同样的。当我们写入的时候，橘色 Job 只会往 01234 这五个里面写数据，紫色往 56789 这里面写数据，我们在写入的时候也是不用考虑锁的，每个 Job 读写的数据都是独立的，所以我们的 Job 里面是不需要给任何数据加锁的，这个就**保障了 Culling 过程的速度**。

当然这会带来一个小问题，就是我们产生的 IndexList 本身是不连续的，里面会有很多不可用的 0。因此Unity在 Culling Job 完成之后，会有一个 **Combine** 的操作，把不连续的 IndexList，就是把后面这些可用的数据拷到前面来，这样就会产生一个连续的 IndexList。示意图如下：



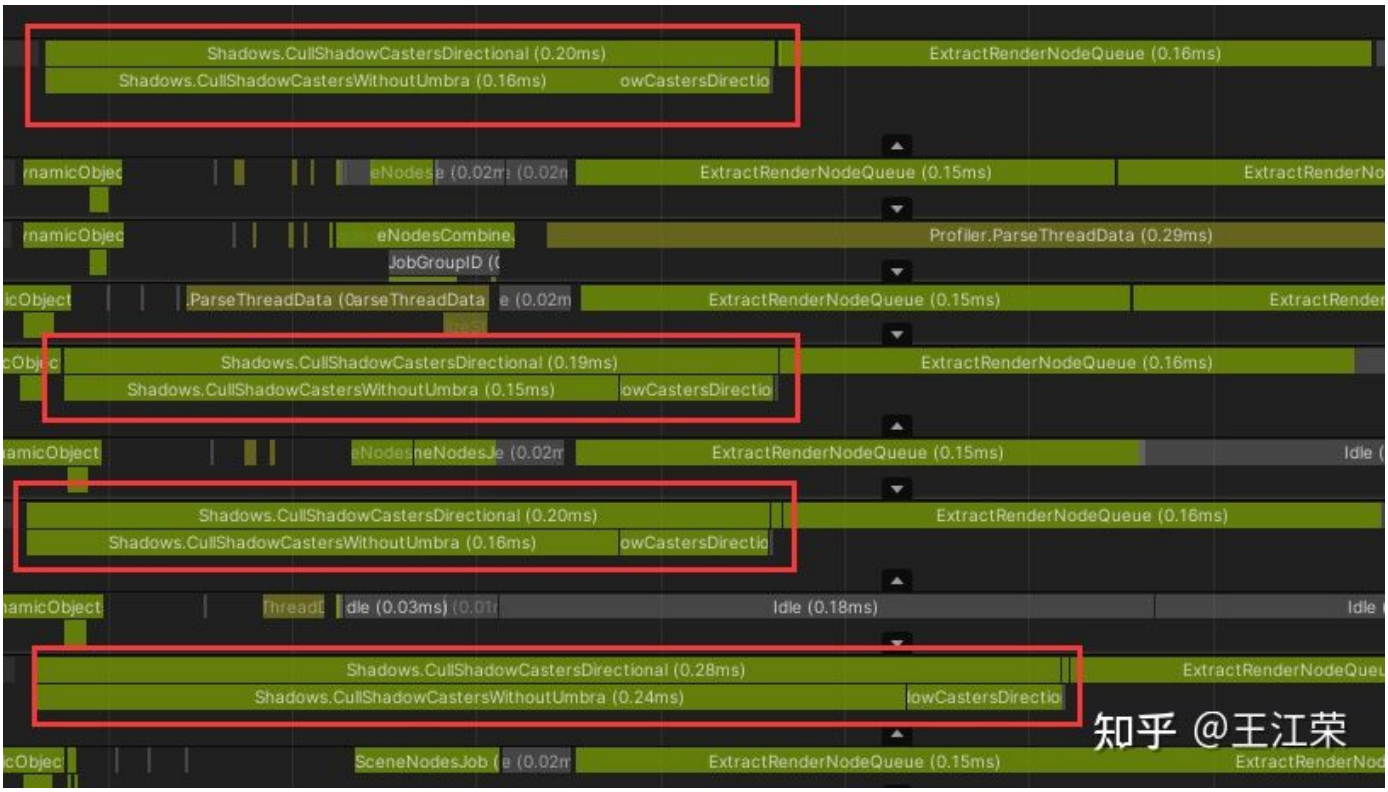


Combine Job

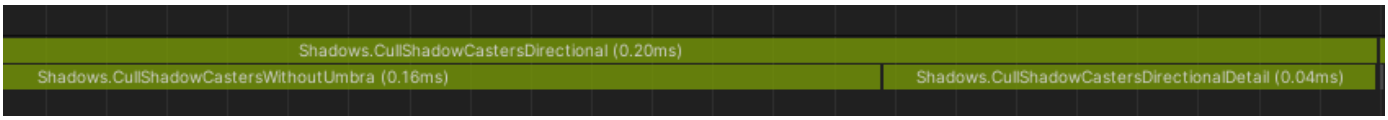
这时候我们就可以得到了场景里面目前可见的 Renderer 数组的下标，完成了场景里面动态物体裁减的过程。

Shadow Culling

可见物体有了，接下来就要进行有关阴影的剔除计算，其中就包括 Shadow.CullShadowCasterDirectional。因为我们的Demo中四盏灯，因此我们可以看下四块相关的Job，如下图：

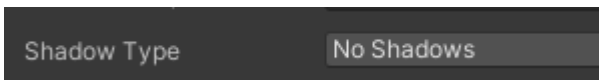


每块的高清无码图如下：

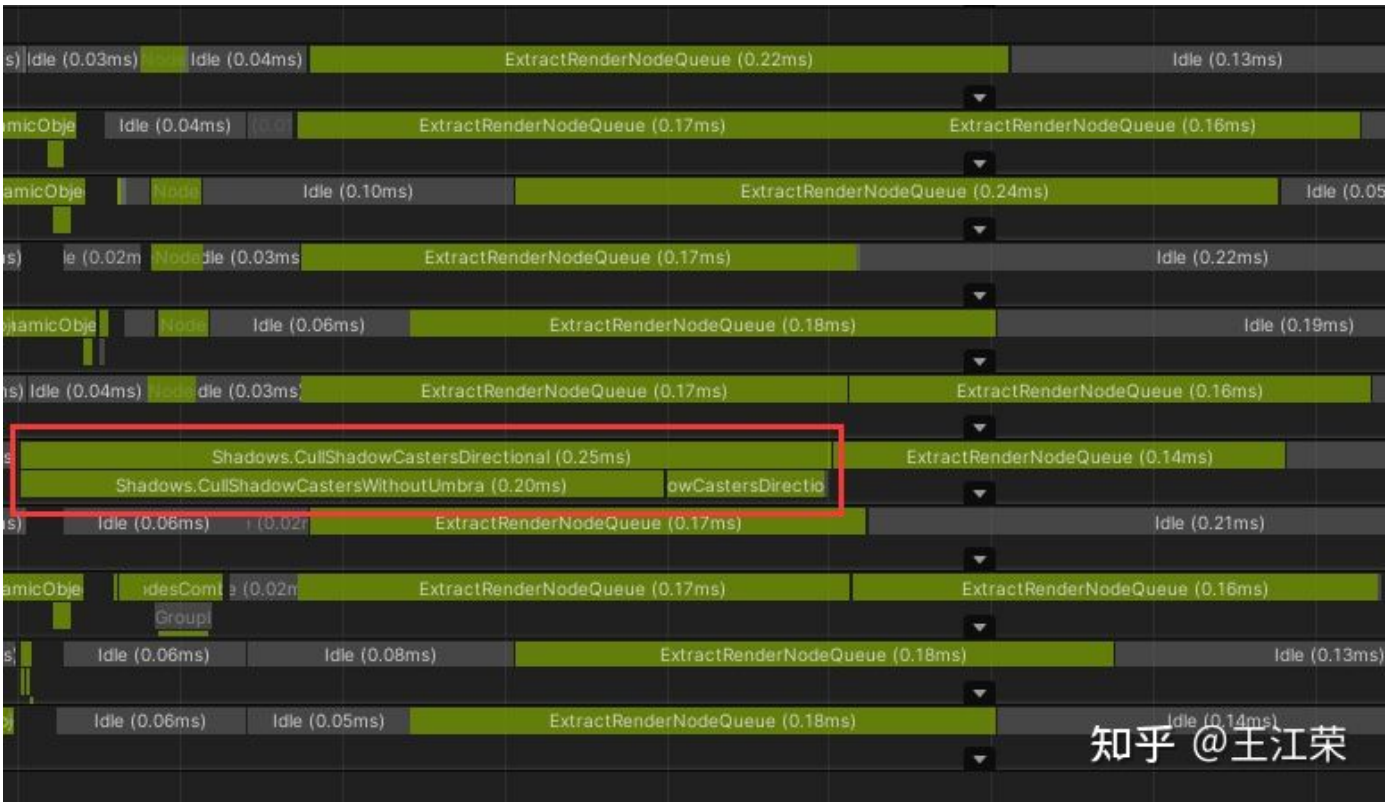


Shadow 的 Culling

如果此时我们把场景中四盏灯光中其中三盏灯光的 Shadow 的选项改成 No Shadows：



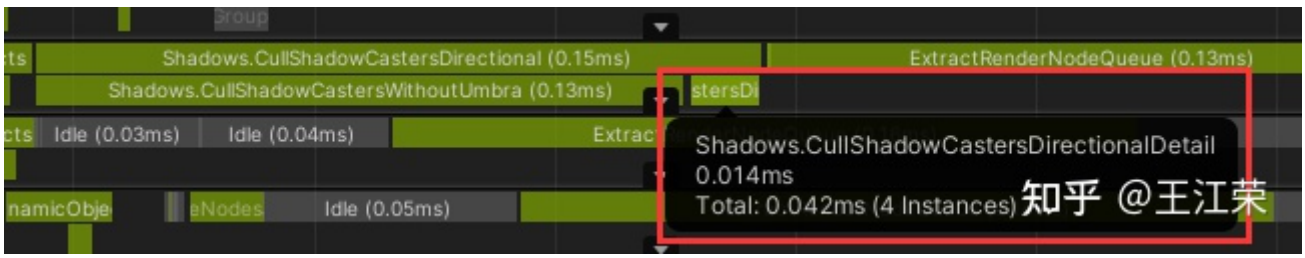
再来看 Profiler ， 可以发现相关Job变得只有一个了， 如下图：



因为 Unity 底层会为我们每一个产生阴影的灯光创建一个 Shadow 的 Job，去裁减我们整个的 Shadow。

我们再来看看其中叫做**Shadow.CullShadowCastersDirectionalDetnail**的这部分，它跟什么有关系呢？我们继续修改我们场景里面的设置，我们把场景里面所有的Renender中产生阴影的选项给关掉。

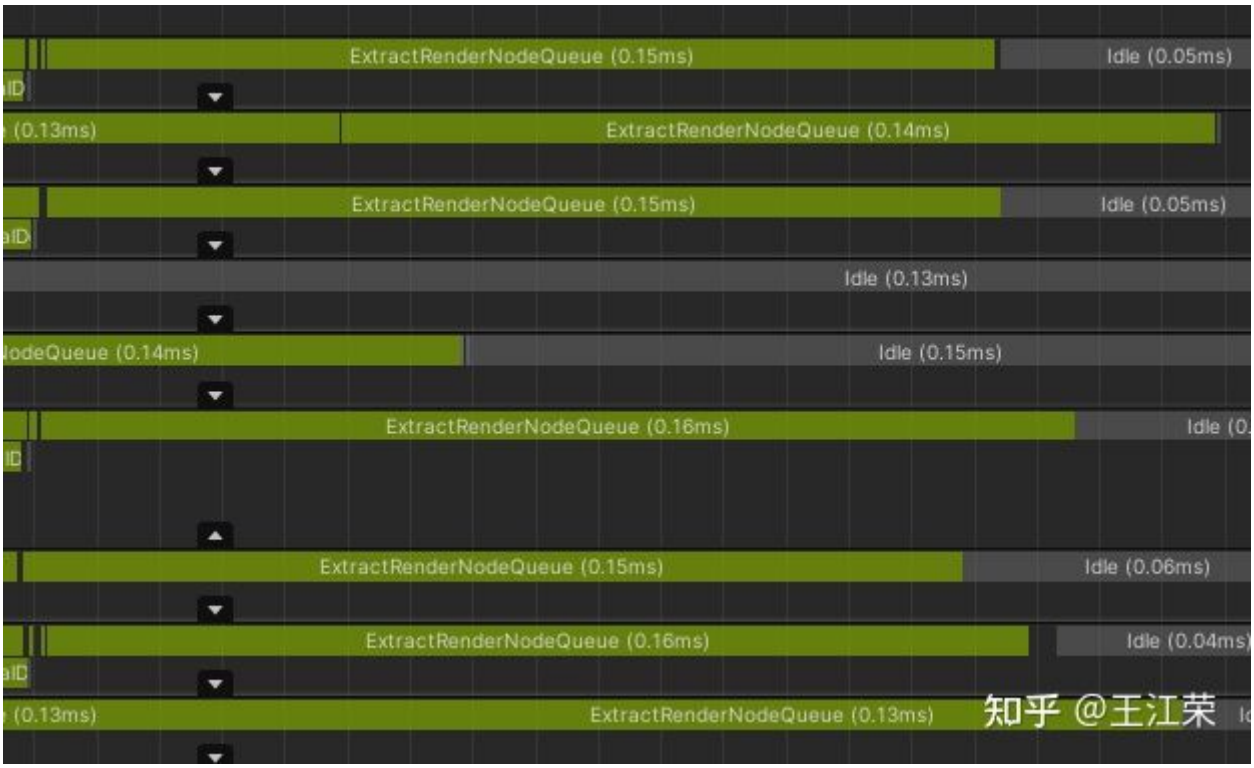
这个时候再看下Profiler， 可以发现这部分运行的开销也小了（所占的比例比上面的图小了很多）。



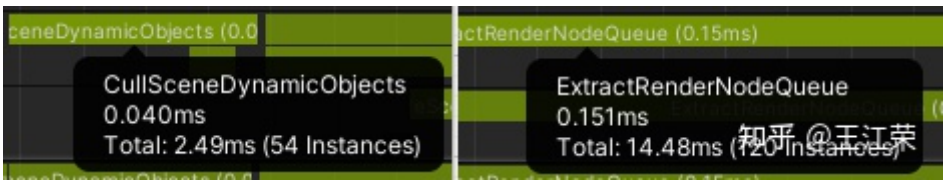
因此如果想减少整个阴影部分的裁减开销，首先要去检查一下场景里的灯光是否有必要产生阴影，其次要去检查场景里的物体应不应该产生阴影，如果发现不合理的应该把这些选项都关掉，这样阴影裁减这部分的开销就能够降下来。

ExtractRenderNodeQueue

我们再来看看后面的ExtractRenderNodeQueue部分，如下图：

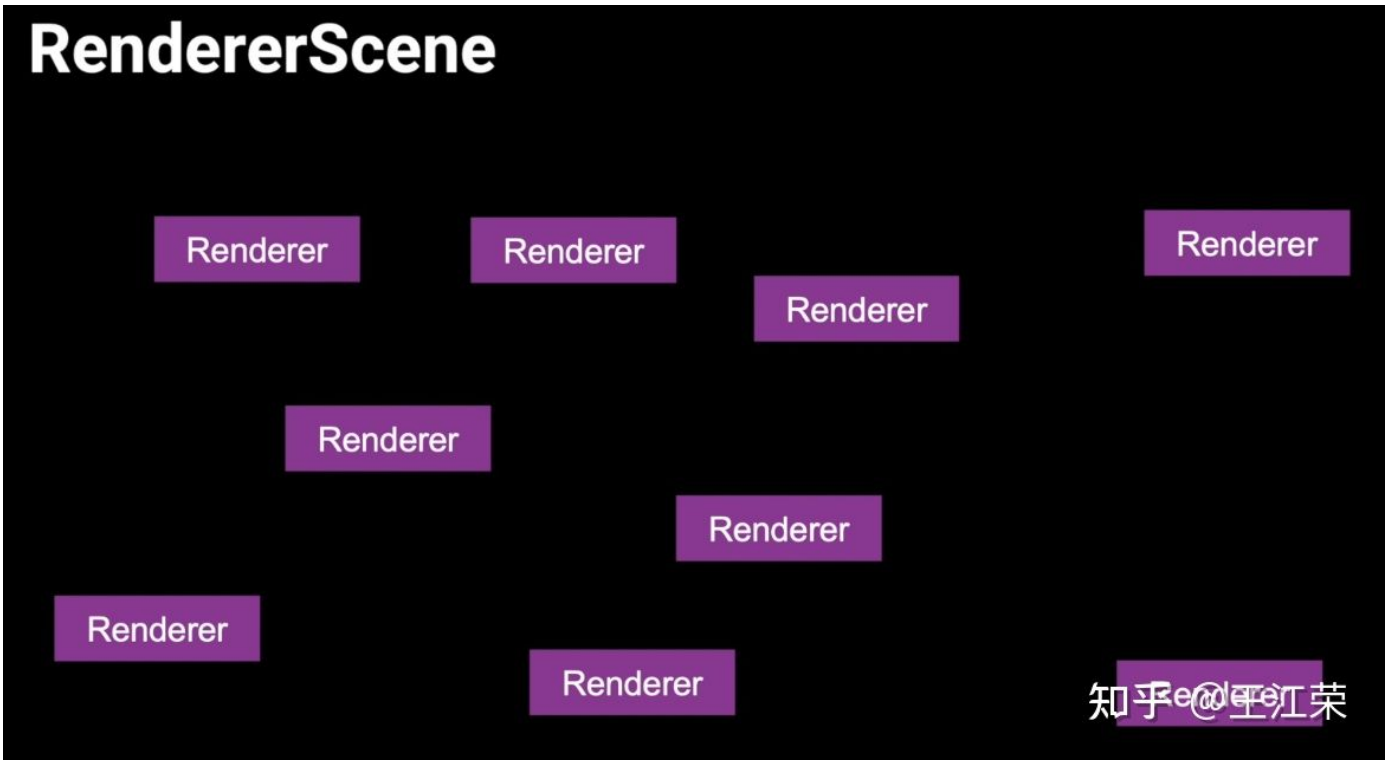


这一部分从耗时来看，比整个场景的动态物体裁剪的开销更大。



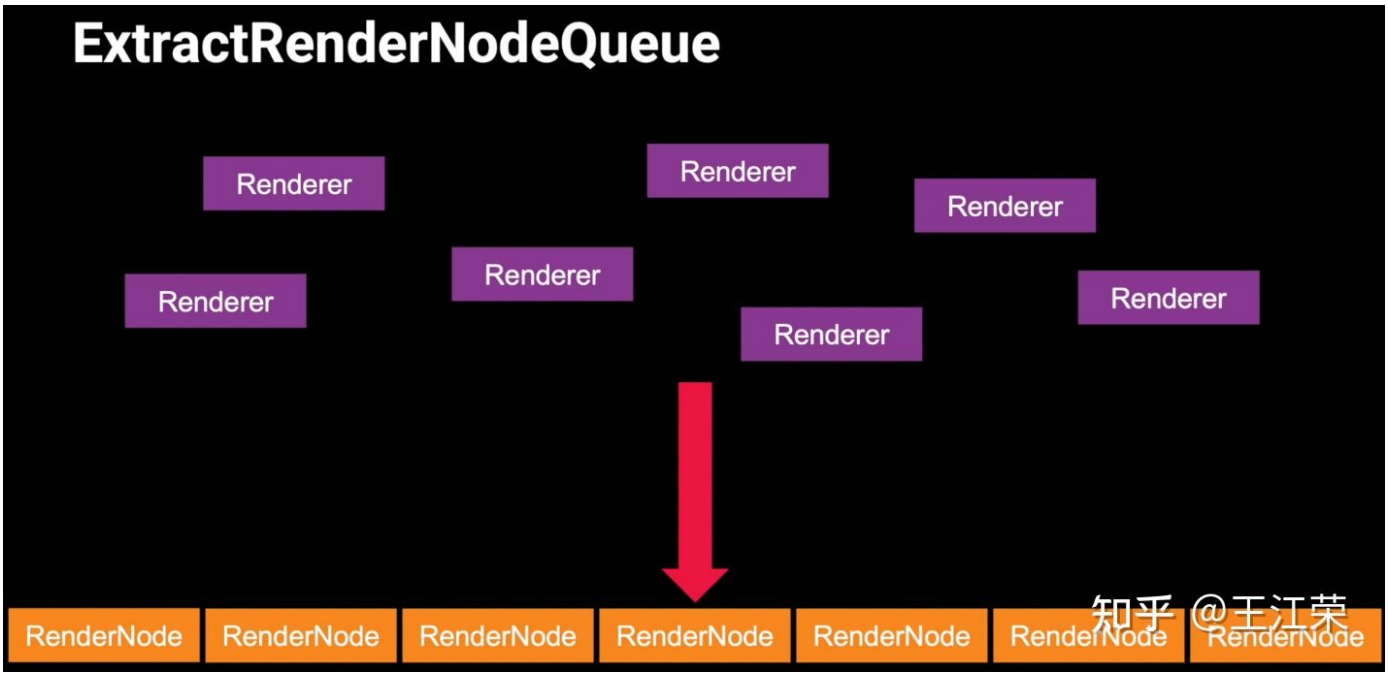
耗时对比

在理解它的作用之前，我们先来看下Renderer对象在内存里是怎么排布的，前面说到Unity里维护了一个Renderer的List，但是所有的Renderer在我们的内存里其实是一个乱序的排布，示意图如下：



那么当我们尝试去**顺序读取**Renderer对象的时候，造成的开销要比在内存里做连续的时候大得多。那么我们就要想办法把它们做成内存上连续的，来保证我们渲染的速度，Unity因此引入了一个新的数据结构：**RenderNode**，它其实就是我们Renderer对象的一个扁平化的版本。**RenderNode是一个非常大的全部都是值类型的Struct**，Unity会把Renderer里所有引用类型的数据展开然后拷贝到RenderNode里。RenderNode本身在内存上是连续的数据，由RenderNode组成的队列我们称之为**RenderNodeQueue**。RenderNodeQueue本身是线程安全的，因此它能够被直接拿来做多线程渲染。

所以我们ExtractRenderNodeQueue的整个过程就是遍历所有目前可见的Renderer对象，然后把它里面的数据拷贝到RenderNode上，最后把RenderNode组成一个RenderNodeQueue。示意图如下：



如果我们想要降低这部分的开销，应该减少场景里面可见的Renderer数量。

Scriptable Draw

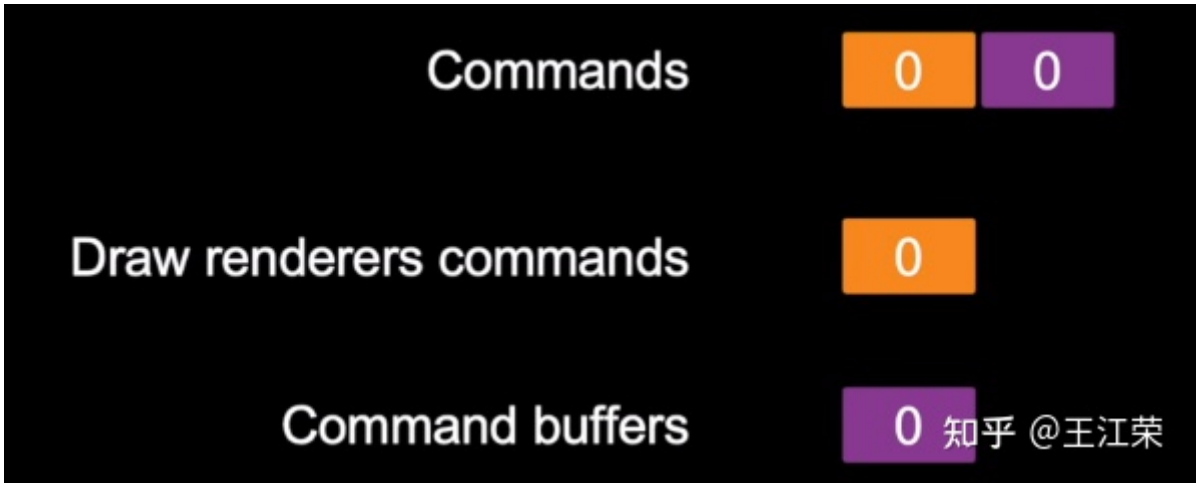
Culling完成之后就是我们的Draw模块了，它不仅包含有**CommandBuffer**中常用的**Blit**，**DrawMesh**这些方法还包括**ScriptableRenderContext**里的**DrawRenderers**，**DrawShadows**这些。那么在这些API被调用的时候会发生什么样的事情呢？我们用ExecuteCommandBuffer和DrawRenderers来举例（其他都是一模一样的）。

我们先来看下引擎内部的这些Commands存储，如下图：

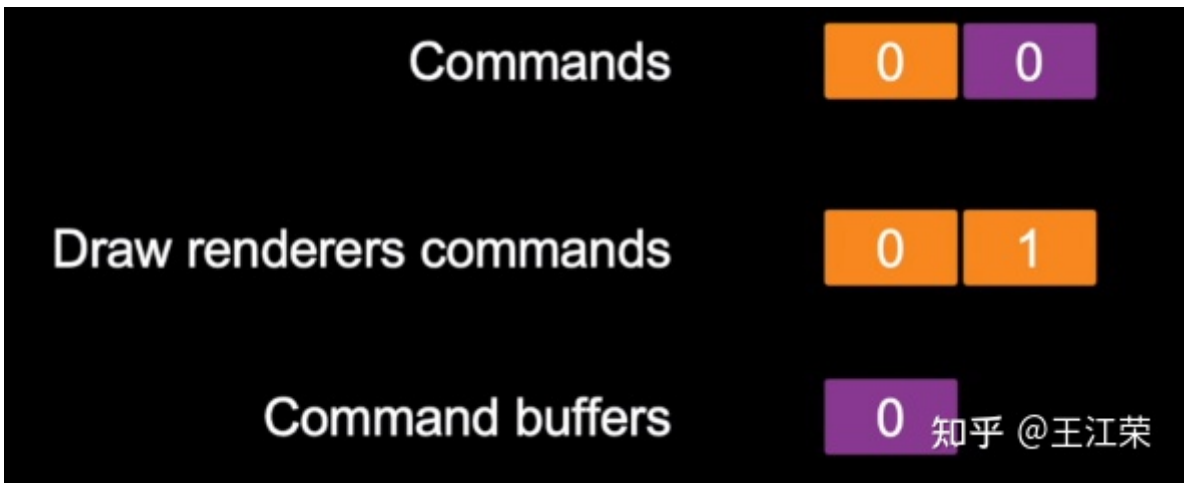
```
dynamic_array<ShadowDrawingSettings>  m_DrawShadowCommands;
dynamic_array<DrawRenderersCommand> m_DrawRenderersCommands;
dynamic_array<RenderingCommandBuffer*> m_CommandBuffers;

dynamic_array<Command> m_Commands;
```

图中有4个list，分别是m_DrawShadowCommands，m_DrawRenderersCommands，m_CommandBuffers和m_Commands，我们来简单看看他们之间的关系。

假设m_Commands，m_DrawRenderersCommands和m_CommandBuffers三个list的初始样式如下：



当我们调用DrawRenderers的时候，会产生一个DrawRenderersCommand，然后把这个Command加到m_DrawRenderersCommands中，如下图：



但是同时Unity也会产生一个Command对象，放到m_Commands中，如下图：

这个Command对象会记录Command的类型，例如示意图中橘色代表着DrawRenderersCommand，紫色代表着CommandBuffer，当然也会包括DrawShadowCommands，示意图中省略了。而Command里记录的下标（刚刚新增的橘色1）则是对应的DrawRenderersCommand在自己的list里的下标。

同理，那么假如我们在新增两个CommandBuffer，那么除了在m_CommandBuffers里新增两个对象外，还会在m_Commands中新增两个对象，存储对应Command的类型以及下标，如下图：

不管是调用DrawShadows，DrawRenderers还是ExecuteCommandBuffer的时候，其实就是在生成这个队列。也就是说我们在调用这些方法的时候，并不会立刻去进行相应的绘制操作，Unity只是把它们存到相应的队列里去，这时没有做任何的渲染动作的。

而什么时候去做渲染呢？就是我们下面要提到的Submit。

Scriptable Render Loop

也就是ScriptableRenderContext.Submit 方法底下做了什么样的事情。

整个Render Loop的伪代码如下：

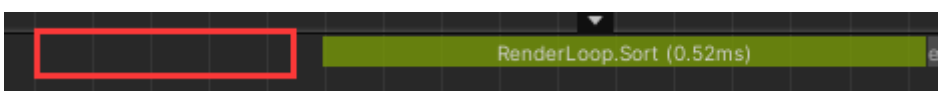
```
for (UInt32 i = 0; i != m_Commands.size(); i++)
{
    cmdType = m_Commands[i].type;
    switch (cmdType)
    {
        case DrawRenderers:
        {
            drawRenderers = drawRenderersData[m_Commands[i].perTypeIndex];
            ExecuteDrawRenderersCommand(drawRenderers);
            break;
        }
        ...
    }
}
```


Scriptable Render Loop

非常简单，就是一个For循环，遍历m_Commands队列。然后根据每一项的类型和下标，从m_DrawShadowCommands，m_DrawRenderersCommands，m_CommandBuffers中取得相应的对象，最后执行**ExecuteDrawRenderersCommand**来实现绘制。

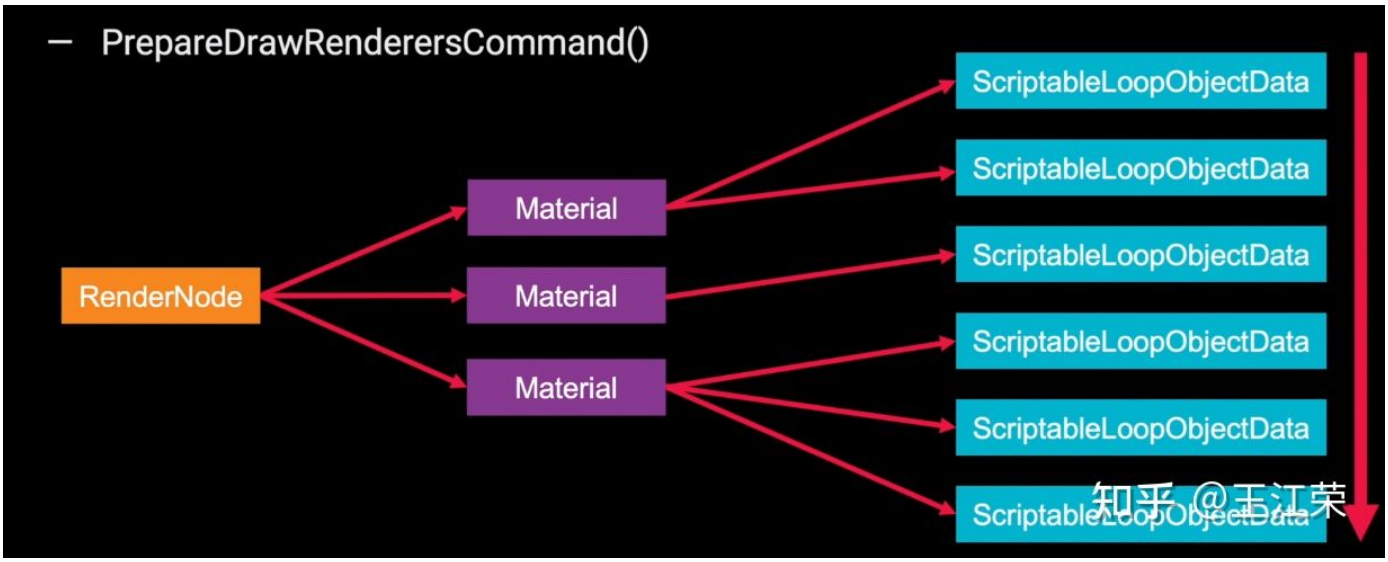
PrepareDrawRenderersCommand

PrepareDrawRenderersCommand操作Unity并没有直接在Profiler里展示出来，它其实对应的是下面这个部分（Sort之前）：



在前面，我们通过Culling得到了RenderNode和RenderNodeQueue，那么这些数据拿去做渲染是否已经足够了呢？我们知道一个Renderer会包含一个或多个的Material，然后一个Material又会包含一个或多个的Pass，然后我们还需要通过Sort来决定哪些东西先画哪些东西后画。

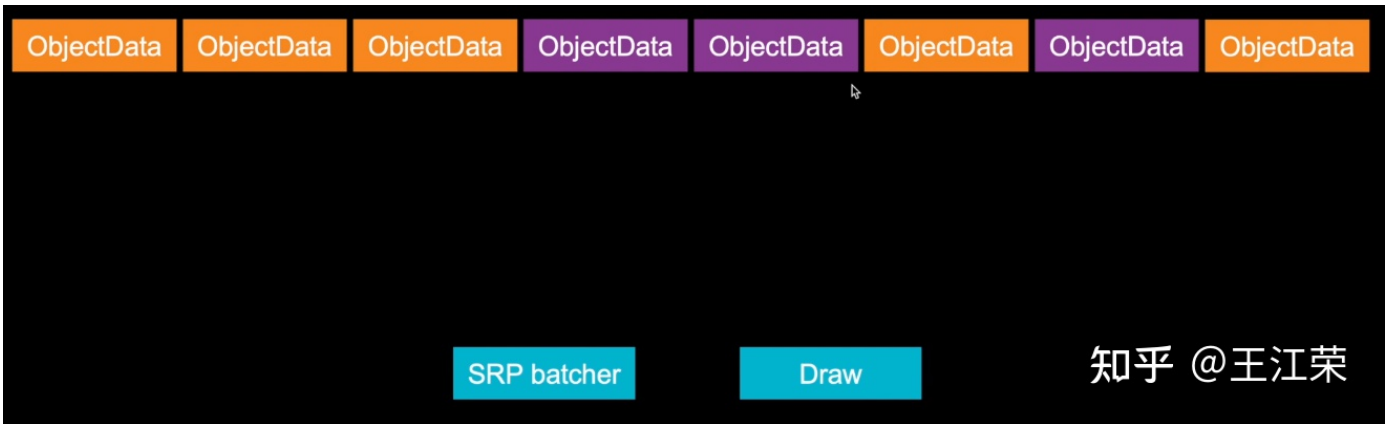
而PrepareDrawRenderersCommand的操作就是遍历我们所有的RenderNode，然后找到里面所有的Material，然后再遍历每个Material找到里面可以用的Pass，根据每个Pass去生成一个**ScriptableLoopObjectData**（简称ObjectData）。示意图如下：



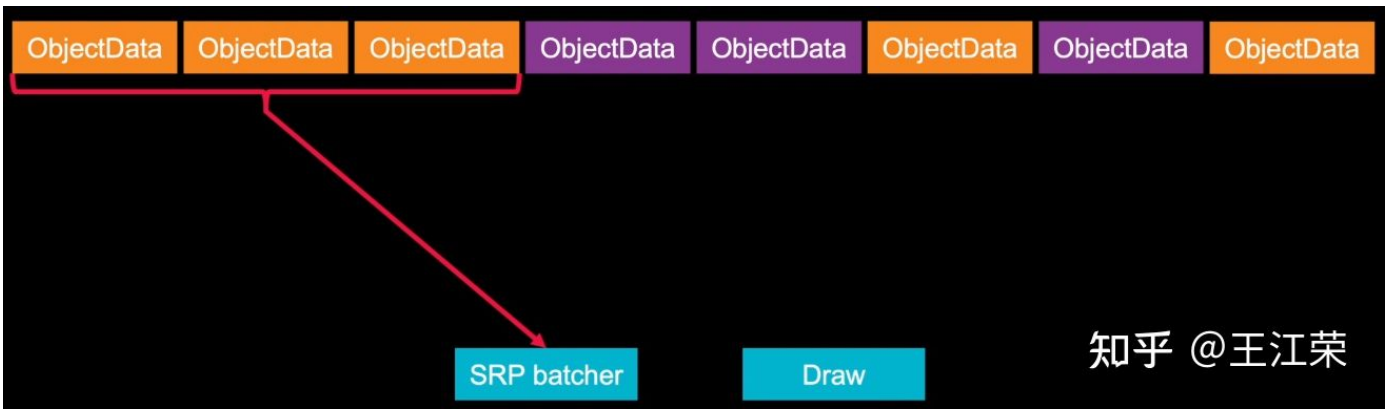
所有的ObjectData生成后，我们在对这些ObjectData进行一个排序，这样就可以得到一个确定的渲染顺序。然后我们做渲染的话就是拿到这些ObjectData，然后逐一进行渲染就可以了。

ScriptableRenderLoopDrawDispatch

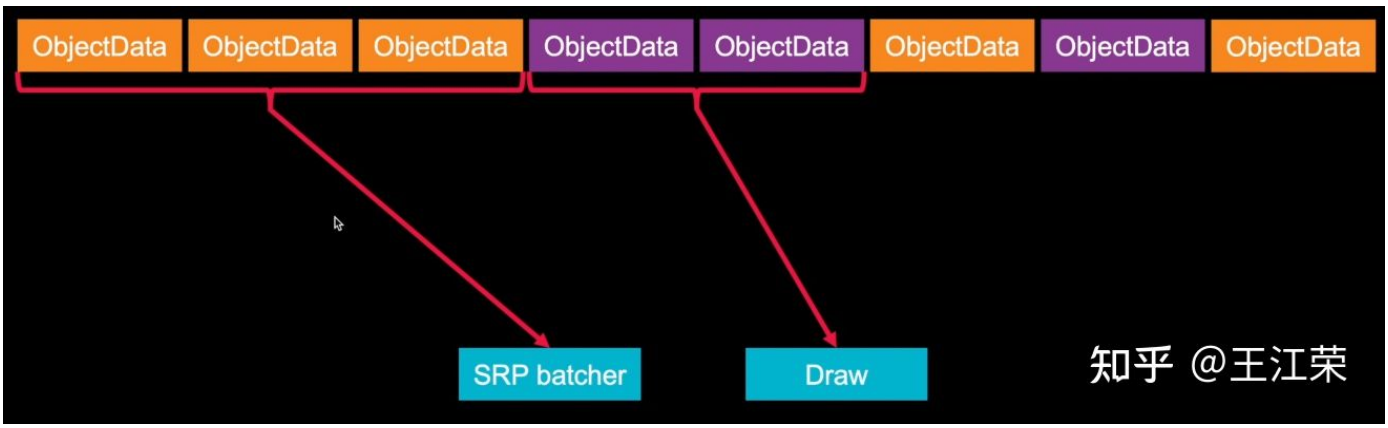
然后我们来看下DrawRenderers里具体的实现是怎么样的，如何进行Dispatch，示意图如下：



ObjectData里会有个标识来记录是否兼容SRP batcher，例如图中橘色代表兼容，紫色代表不兼容。当我们排序完成后，ScriptableRenderLoopDrawDispatch会根据SRP batcher是否兼容，找到所有的连续的ObjectData，如下图：



前三个是兼容的，会全部丢到SRP batcher渲染器里做渲染。然后后面两个是不兼容的，就会被丢到传统的Draw渲染器里去。如下图：

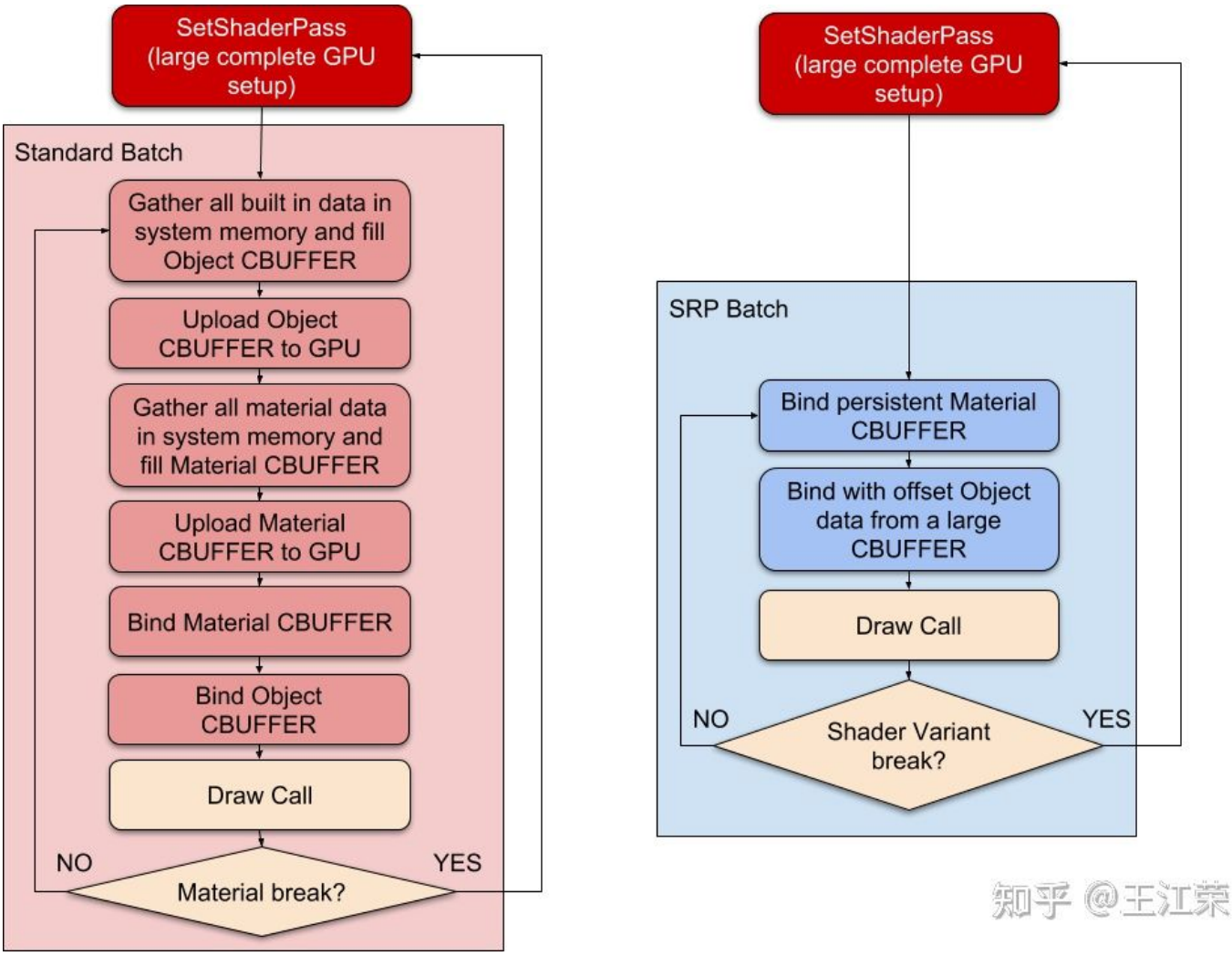


这里就会发现一个问题，当我们判断ObjectData是否要进入SRP batcher的时候，我们只判断它们是否兼容SRP batcher。也就是说如图中第一个ObjectData和第二个ObjectData它们可能本身是同一个Shader，也有可能不是同一个

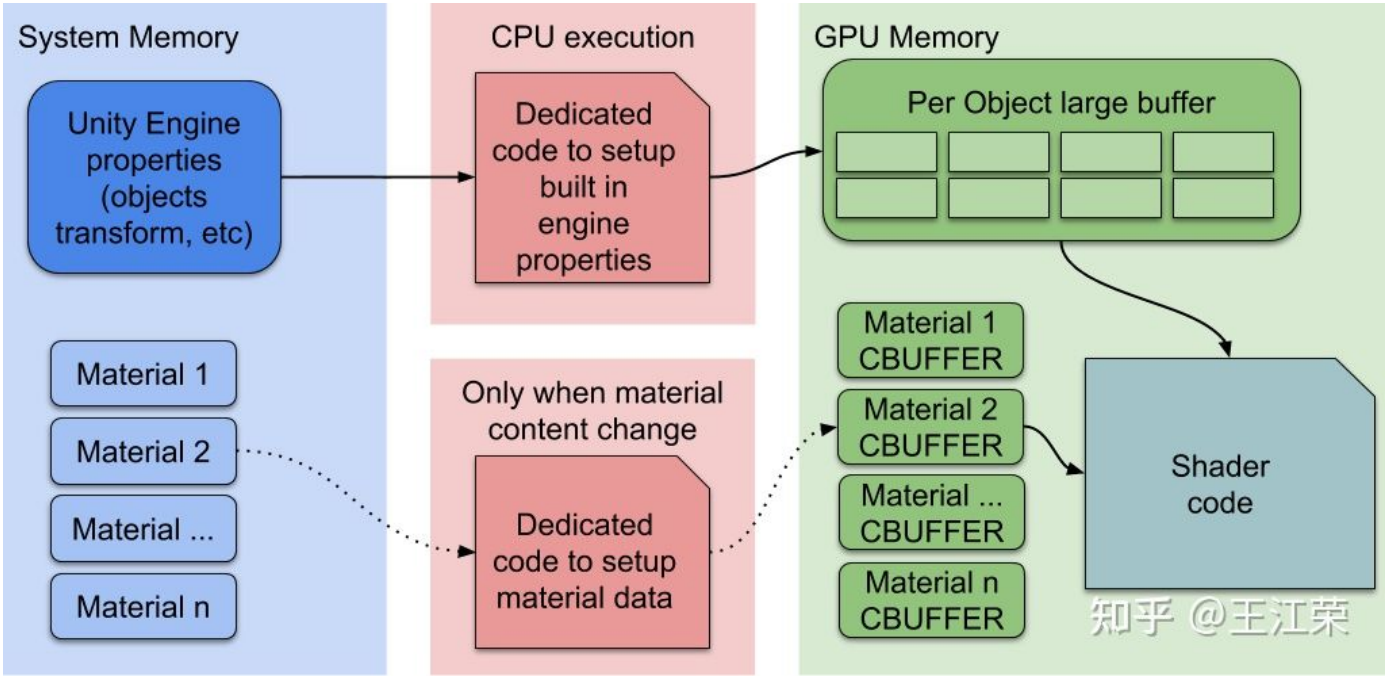
Shader，或者可能是同一个Shader不同的Pass。也就是说它们有可能能被batch在一起，也有可能不能，这个问题后面SRP batcher部分再介绍。

SRP batcher

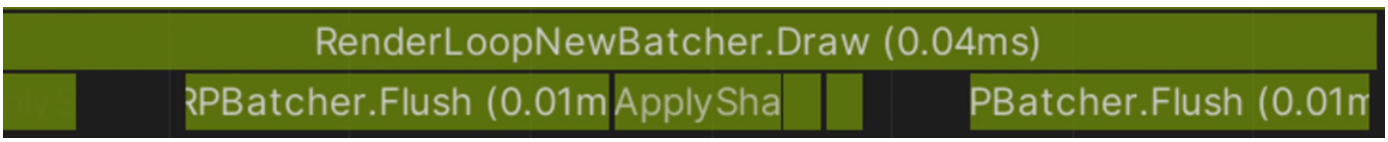
下图是官方文档提供的图：



很明显右边SRP batcher的复杂度更小，因此也更高效。SRP batcher最核心的部分为：Bind with offset Object data from a large CBUFFER，它会为我们准备一个large CBUFFER，**把batch里面每个draw call里小的CBUFFER组织成一个大的CBUFFER，然后统一的去上传到GPU**。工作流程如下图：



其中图中右上角Per Object large buffer部分，里面8个小方块就代表着8个小的buffer，意味着一次的batch里面有8次的draw call。每个draw call都需要一个小的buffer，会把我们引擎内部一些内置的数据（比如unity_ObjectToWorld）填充进去。然后我们为每个Object准备这些小buffer，然后组成一个大buffer，最后把这个大buffer一次性的传到GPU上。这些基本上就是我们SRP batcher做的工作，对应到Profiler里就是RenderLoopNewBatcher.Draw部分，如下图：



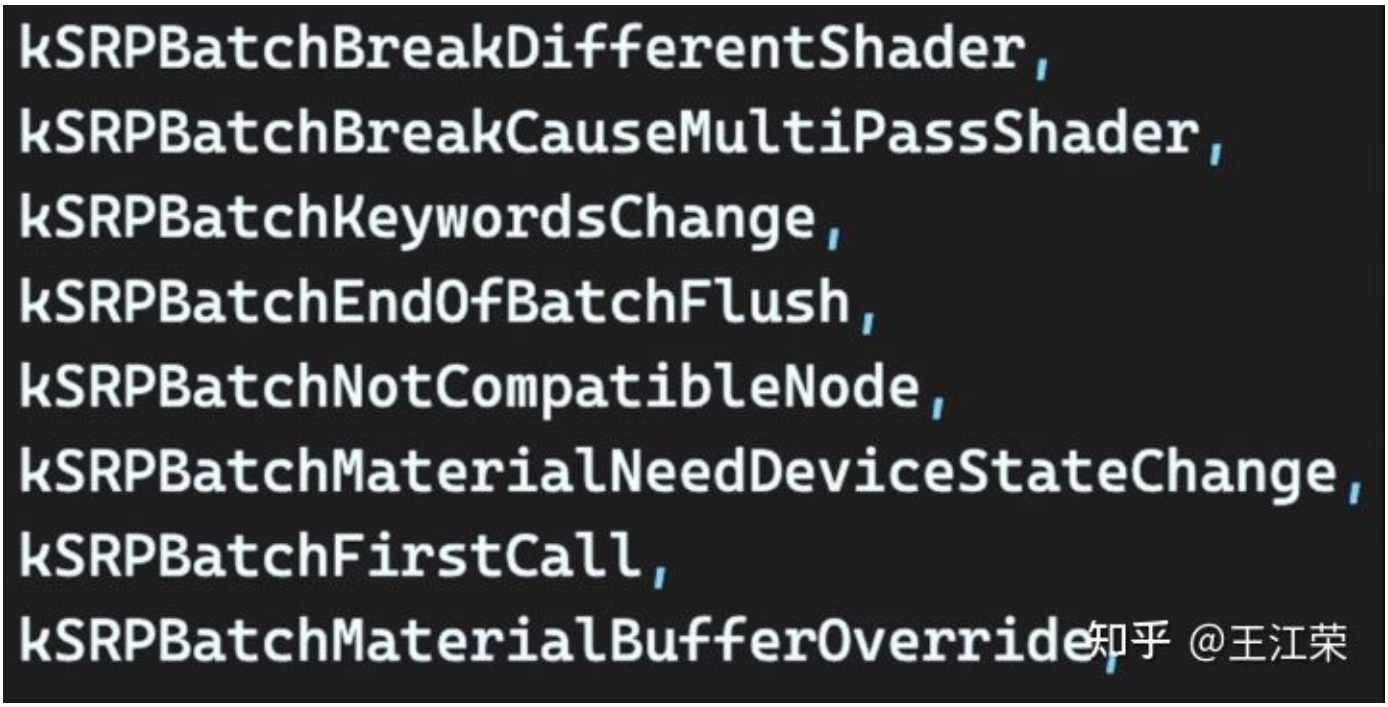
RenderLoopNewBatcher.Draw

然后我们来看看RenderLoopNewBatcher.Draw的工作原理，在前面ScriptableRenderLoopDrawDispatch后，我们知道哪些ObjectData会被传到SRP batcher中，如下图：

图中的代表着传到SRP batcher中的ObjectData，我们会遍历这些ObjectData，当解析第一个ObjectData时，会先创建一个batch。然后再看第二个ObjectData，看它能不能和前面的数据batch在一起，示意图颜色相同代表可以，因此前两个会被batch在一起。然后我们再看第三个ObjectData，发现是紫色的了，说明不能和前两个进行batch。这时就会产生一次**Flush**，会把前面两个ObjectData变成一个批次拿去做渲染，同时创建一个新的batch。然后再看后面的数据能否和当前的数据做batch，如此循环到最后。这里就解释了前面Dispatch时提到的问题。

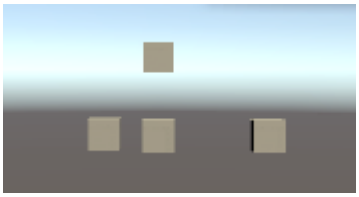
也就是说SRP batcher它只是放宽了我们合批的条件，我们传统的那些优化方案，例如减少材质，减少Shader的数量，RenderQueue的调整，通过Sort让能够合批的Shader尽量排在一起，他们依然是适用于SRP batcher的。

下图为打断合批的原因：

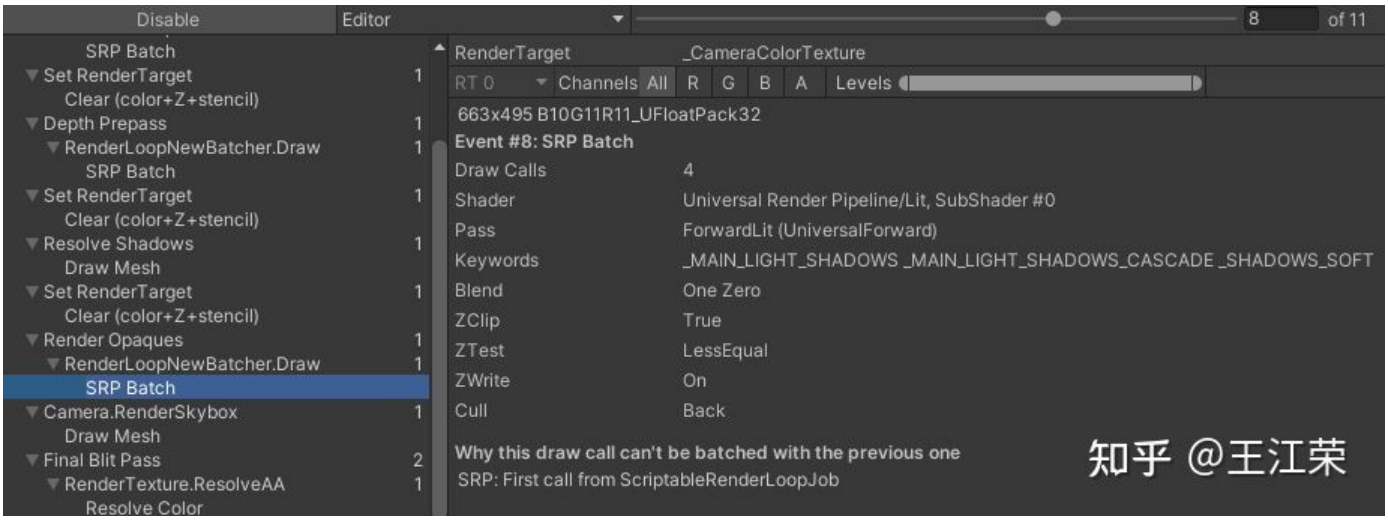


这些枚举，我们可以在Unity的Frame Debugger中看到，并且能看见更详细的解释，根据错误可以进一步优化我们合批的情况。

举个例子，假如我们场景中有如下四个材质一样的Cube：

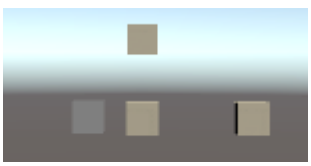


然后我们查看Frame Debugger，可以发现它们四个被batch在一起了。

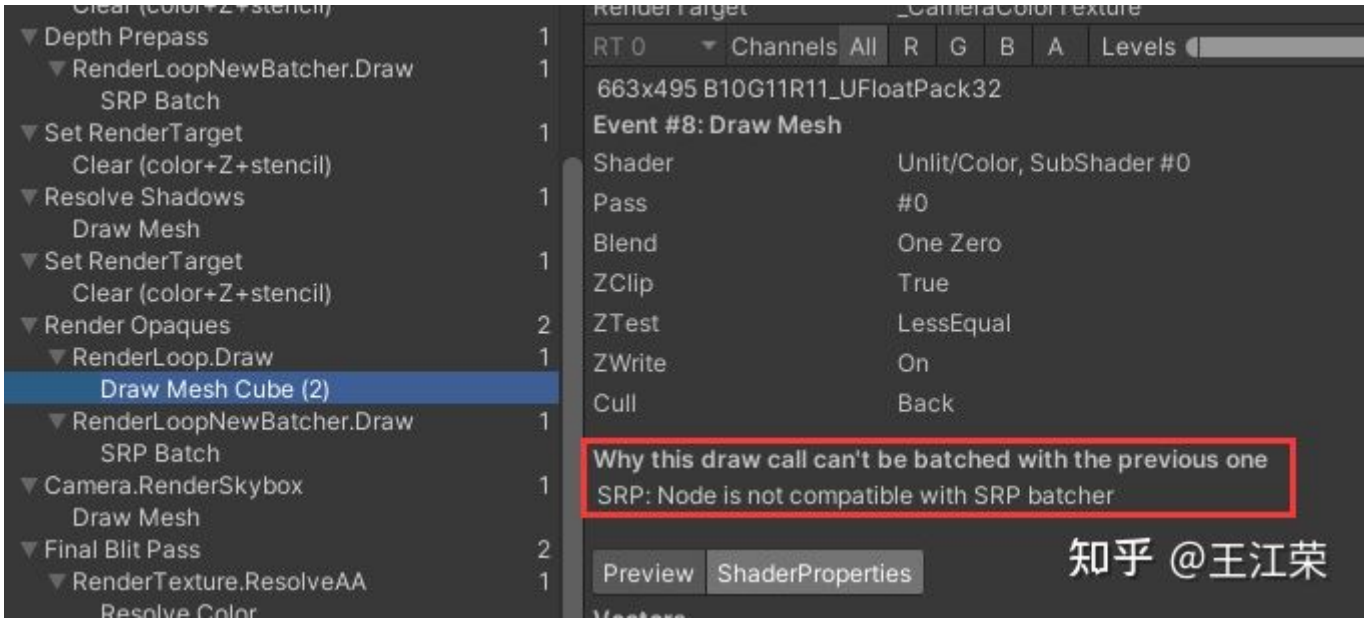


其中First call from ScriptableRenderLoopJob的介绍，就说明这个batch是第一个生成的。

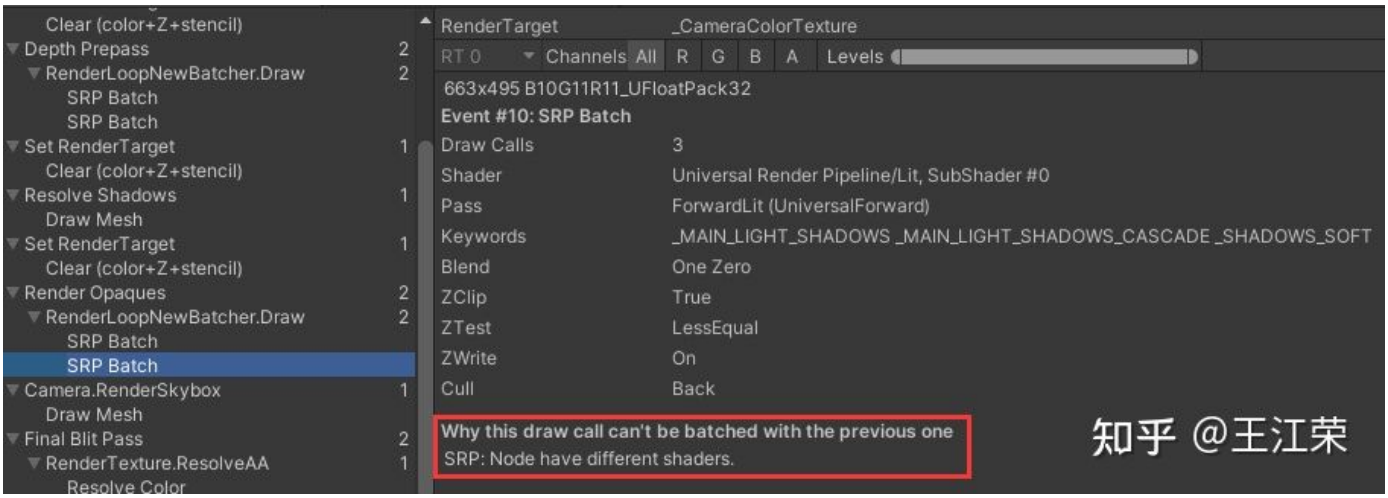
接着我们新建一个Material，使用 Unlit/Color 的Shader，赋予其中一个Cube，场景变为下面这样（最左边Cube用的自定义的Material）：



查看Frame Debugger，会发现这个Cube没有被Batch，因为我们使用的Shader不兼容SRP Batcher。



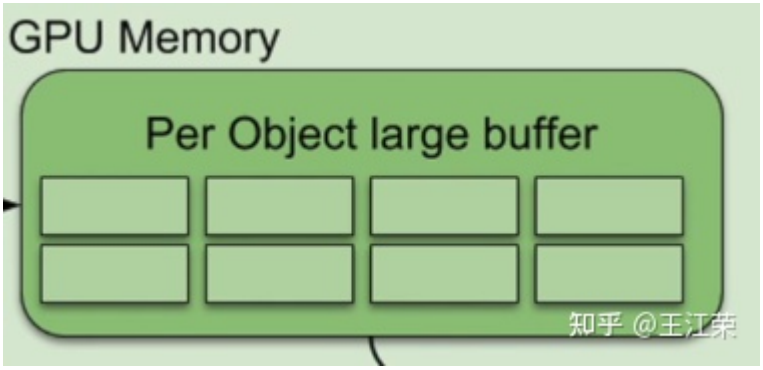
那么我们在换个兼容的Shader，Material里选择 Universal Render Pipeline/Unlit 的Shader，再看下Frame Debugger。



可以发现此时变成了两个Batch，因为不同的Shader。

SRPBatcher.Flush

前面提到的Flush操作，其实就是填充前面提到的Per Object large buffer里的小buffer。

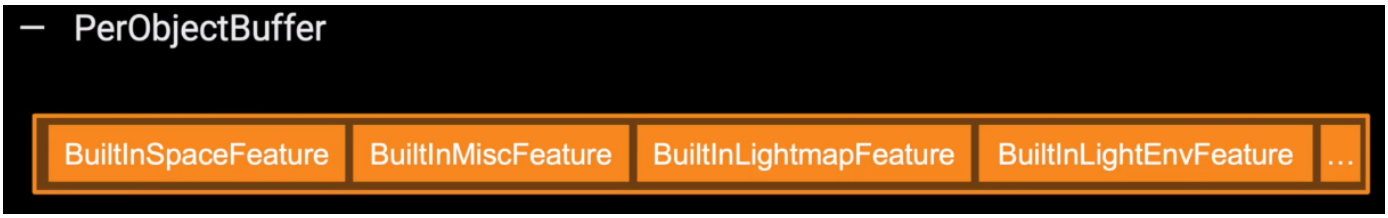


每个小buffer里的数据和其内存排布如下图：

“Space” block feature	type	Could be half
unity_ObjectToWorld unity_WorldToObject unity_LODFade unity_WorldTransformParams	float4x4 float4x4 float4 float4	No No No Yes
LightMap block feature	Type	Could be half
unity_LightmapST unity_DynamicLightmapST	float4 float4	No No
Render Layer block feature	Type	Could be half
unity_RenderingLayer	float4	No
Spherical Harmonic block feature	Type	Could be half
unity_SHAr unity_SHAg unity_SHAb unity_SHBr unity_SHBg unity_SHBb unity_SHC	float4 float4 float4 float4 float4 float4 float4	Yes Yes Yes Yes Yes Yes Yes
Probe Volume block feature	Type	Could be half
unity_ProbeVolumeParams unity_ProbeVolumeWorldToObject unity_ProbeVolumeSizeInv unity_ProbeVolumeMin	float4 float4x4 float4 float4	No No No No
Probe Occlusion block feature	Type	Could be half
unity_ProbesOcclusion	float4	No
Motion Vector block feature	Type	Could be half
unity_MatrixPreviousM unity_MatrixPreviousMI unity_MotionVectorsParams	float4x4 float4x4 float4	No No No
Light Indices block feature (used in LWRP)	Type	Could be half
unity_LightData unity_LightIndices[2]	float4 float4 (2 elements)	Yes Yes
Reflection Probe 0 block feature	Type	Could be half
unity_SpecCube0_HDR	float4	Yes
Reflection Probe 1 block feature	Type	Could be half
unity_SpecCube1_HDR	float4	Yes

PerObjectBuffer

接下来我们来看下PerObjectBuffer是如何去填写这些数据的。当一个Shader确定的时候，这个Shader使用了哪些**Feature**就已经确定了，如下图：



比如说我们的Shader里使用了BuiltInLightmapFeature，那么就会把LightmapFeature给它填充进去（数据参考上面的表格），如果没有使用就不会填这部分的数据。也就意味着我们的**Shader使用的Feature越少，我们一次能够合批的数量就会越多**。

填充完PerObjectBuffer后，就会把他们组成一个大的CBUFFER（**PerObjectLargeBuffer**），然后统一的传到GPU做渲染，如下图：

Q&A部分

1. **SRP batcher是工作在CPU层面的，它做的事情就是减少SetPass Call。**
Unity在很久以前就把Draw Call和SetPass Call做了区分：Draw Call本身就是调用一个图形的API，它本身的开销并不耗。而开销高是高在我们做切换渲染状态的时候要提前为显卡准备非常多的数据，也就是SetPass Call的工作，准备这些数据往往来说是开销比较高的。评判标准：不管是默认管线还是SRP，SetPass Call最好都不要超过150，Draw Call的话可以高一些。

2. **Vaulkan**：SRP batcher在CPU层面的开销，比较可以关注的一个点是android上的vaulkan，它已经越来越成熟，有不少项目在立项阶段把vaulkan作为首选的API。其实使用了vaulkan的话，会有一个明显的发现就是，**vaulkan在CPU上的开销要远远小于OpenGL**。所以推荐！！！

3. **SRP batcher和GPU Instance用的技术是差不多**，如果大家是想绘制单一的物体（像草这样的），推荐大家使用GPU Instance。但是如果想做正常的场景渲染，比如说场景里的material多于5个，SRP batcher的速度要比我们手动做GPU Instance要划算的多的。

参考：

https://blog.unity.com/technology/srp-batcher-speed-up-your-rendering

blog.unity.com