- **产生DrawCall的条件**

> - 当Canvas下的所有节点区域的最小AABB包围盒与Canvas的绘制区域有交汇时，才会进行DrawCall计算。[**即便会进行DrawCall计算，也会有Canvas自身的2个DrawCall消耗**]
> - 当Canvas进行DrawCall计算时，即便把子元素移出至屏幕外，移出的子元素仍然参与DrawCall计算
> - 移出屏幕外的子元素DrawCall计算规则与仍然在屏幕内的元素计算规则一致。因此如果移出屏幕外的元素有区域重叠，会使DrawCall增高。

- **UI重绘时机【SendWillRenderCanvases】**

> - 当UI ModeifyMesh之后，就会触发Canvas的SendWillRendererCanvases，进行UI重绘。[RectTransform修改了SizeDetal、Anchor、pivot都会触发ModifyMesh，其他的SQT变化并不会触发ModifyMesh]
> - 切换UI的Active之后，会触发ModifyMesh，会触发SendWillRendererCanvases消耗
> - 切换UI的CanvasGroup或CanvasRenderer的alpha，不会触发ModifyMesh，也就是说设置alpha为0也能够达到SetActive=false的效果，并且alpha设置为0之后，也不会进行DrawCall计算
> - 切换UI的Parent之后，也会触发ModifyMesh，会触发SendWillRendererCanvases消耗

- **UI Despawn方式**

> 1. SetDeactive 最直接的方式。优点：无bug。 缺点： 会造成SendWillRendererCanvaes
> 2. 将UI节点移出屏幕外。 优点： 不会产生SendWillRenderCanvases 。 缺点： 移出出屏幕外同样会进行DrawCall计算。同时需要进行UI区域判断，不能造成UI重叠增加DrawCall。需要手动关闭Update函数。需要注意OnEnable与OnDisable的函数中逻辑是否能正常工作。如果Paent中有Layout，需要在移出屏幕的时候给UI添加IgnoreLayout
> 3. 给UI添加CanvasGroup组件，Despawn的时候设置alpha为0。 优点： 不会造成SendWillRendererCanvaes，不会进行DrawCall计算。缺点： 如果Parent包含了Layout，仍然需要添加IgnorLayout。需要手动关闭Update函数。需要注意OnEnable与OnDisable的函数中逻辑是否能正常工作。需要注意CanvasGroup参数造成的其他逻辑因素。
> 4. 切换Parent。优点：无 。 缺点： 会造成SendWillRendererCanvaes。切换parent会造成其他消耗，比如重新设置SQT等变量。

- **修改Image.color与修改CanvasRender.color的对比**

> - 测试数据： 66个Image
> - 修改Iamge.Color ， Cpu消耗约 15ms,Gpu消耗 0.7ms
> - 修改CanvasRender.color，Cpu消耗约 1.3ms，Gpu消耗 0.7ms

> - 如果能够修改CanvasRender.color得到与修改Iamge.Color一样效果的，优先修改Canvasrender

- **切换UI至UIContainer与defalutContainer【null】的对比**

> - 无意义

- **切换UI的Active与切换CanvasGroup的对比**

> - 测试数据： 66个Image
> - 设置Active,Cpu消耗20ms
> - 设置CanvasGroup，Cpu消耗 1.3ms。

- **重复设置Image的Color**

> - 重复设置Image的Color(相同Color)，并不会引起Rebuild

- **重复设置Text的text**

> - 重复设置Text的text(相同text)，并不会引起Rebuild

- **重复设置Iamge的sprite**

> - 重复设置Image的sprite(相同Sprite)，并不会引起Rebuild

- **重复设置Iamge的fillAmount**

> - 重复设置Image的fillAmount(相同数值，相差小于0.000001f)，并不会引起Rebuild

作用

- 在编辑状态分析当前UI的DrawCall【以下简称DC】消耗，以达到指导优化DC的目的
- 提供Text组件DC优化功能，通告抬高Text组件的层级，以达到整体合并Text组件的作用
- 以此类推，其他的batch也是可以通过类似的抬高层级的操作进行UI之间优化Batch的作用

UGUI DrawCall 分析过程

> UGUI的DrawCall分析过程应该分三种情况来讨论，不能一概而论。

1. 无Canvas情况,无Mask情况

> 首先这是最简单的情况，过程简述如下

- 1. 首先判断当前UI所占的Rect区域【相对于整个的Root节点而言】是否底下有重叠的情况，如果没有，那么当前就是第0层，将当前层级记录下来
- 2. 深度优先迭代Transform下的所有孩子，从最顶层的层级开始判断，查看当前UI Rect所占的区域是否与该层级的任意UI节点重叠，如果是，那么判断当前的UI节点是否能够被该层级的UI节点进行Batch，如果能，那么当前UI节点属于该层级，否则属于后面一个层级。如果没有重叠，那么依次往前面的层级检测，最低层级为0
- 3. 得到所有UI节点以及对应的层级后，将每个层级中的UI节点进行batch合并分析，能够进行合并的放在一个batch结构中。
- 4. 对所有的batch结构进行排序，首先能够确认的一个排序规则是，Text的batch先被绘制，然后才是进行Image的绘制。如果有Mask的情况，Mask的Text与Mask的Image会比无Mask的后被绘制。同时需要注意的是，即便是Text，但是由于font不同或者Text的材质不同，也是不能进行Batch的，那么此时应该如果判断batch排序呢，这里笔者猜测是根据Hierarchy中的先后顺序得出的，Image的也不例外
- 5. 最后将所有的层级从下到上取出所有的Batch集合放在一个统一的集合中，然后迭代该Batch集合，如果相邻的两个Batch之间能够进行再合并，那么就合并为一个Batch。
- 6. 最后，得出的Batch集合就是UI的DC数目，也是最后UI的渲染顺序。

2. 无Canvas但是有Mask情况

> 出现Mask的情况较第一种非常复杂，不同的地方如下：

- 1. 如果当前节点上面存在Mask组件，那么该节点下的所有子孩子都会添加上Mask标签。由于Mask下面还能再出现Mask，所以，这里的Mask标签应该用数量来记录，相同层级的相同数量的Mask也是能够进行合并的，注意直属Mask节点的合并规则与孩子节点的合并规则不同
- 2. 每一个UI节点上面的Mask数量不同，那么肯定不能进行合并。
- 3. 同一个层级进行Batch的排序时，Mask标签的数量越大，那么排序越靠前
- 4. 每添加了Mask组件之后，会多出一个DC,如果用Unity5.x进行FrameDebug，能够看到DC并没有绘制内容，只是在进行UI的AlphaClip，当然，这个DC也是能够被合并的，如果出现的时机一致的话，简单的讲就是MaskA所占的ImageA能够与MaskB所占的ImageB进行合并，那么这个Alpha裁剪的DC就能被合并。
- 5. 什么情况下MaskA所占的ImageA能够与MaskB所占的ImageB进行合并呢，当ImageA与ImageB能够进行合并，并且ImageA与ImageB没有发生重叠的情况下
- 6. 直属Mask的节点，不能与节点下的子节点进行合并，所有子节点的层级都相对+1
- 7. 如果MaskB与MaskA发生了重叠，那么MaskB的层级比MaskA子节点下层级最高的多1，因此会出现当MaskA的节点全部渲染完毕，并且出现了MaskA的AlpahClip 的Dc之后，才有可能进行MaskB的渲染
- 8. 如果当前Mask的子节点都出现在Mask外面，那么即便看不见了，还是会被计算DrawCall，同时会先渲染出现在Mask外面的节点。如果Mask的所有子节点都出现在Mask外面，那么Mask

自身的渲染将会被放在最后面，但是其实只是无所谓的了，因为这个并不会对DC的数量有所改变

- 9. 如果Mask外的当前节点与Mask直属节点的Rect重叠，那么当前节点的层级为Mask最大子节点层级+1，而如果只是与Mask的子节点区域重叠，那么只是Mask子节点的层级+1
- 10. Mask的子节点，如果原本是可以Batch，但是其中一个在Mask Rect 外部，一个Mask Rect内部，那么是不能够一起Batch的。
- 11. 在OutOfMask的情况下，如果Mask的子节点所处的Rect重叠区域只有Mask本身，那么该子节点的的层级与Mask节点层级一致【诡异】，如果重叠区域是Mask下的其他子节点，那么层级为重叠子节点层数+1

3. 有Canvas情况

> Canvas的情况就很好说明了

- 1. UGUI中的合并策略是按照Canvas为单位的，就是说上面的说的过程，全都是发生在单个Canvas下的操作
- 2. 如果在Hierarchy中出现了Canvas，那么会直接打断当前Hierarchy的合并策略。也就是说如果在一个Canvas的孩子节点中间出现了一个子Canvas,那么，可以直接当前有三个Canvas的层级进行处理。

UGUI DC计算 详细代码

例子

1. 无Canvas情况,无Mask情况

- 一个ImageA的层级是0，一个ImageB的层级是1，一个Text的层级是0，那么最后的DC是2，因为渲染顺序是Text-> ImageA - ImageB,而因为相邻的batch能够继续进行合并，所有ImageA与ImageB进行了最后的合并。
- 笔者在Unity5.x上做过一个尝试，一个TextA的层级是0，一个ImageA的层级是0，一个TextB的层级是1。按照我们的分析过程，此时的DC应该3，过程为TextA -> ImageA -> TextB,然而很诡异的现象是，DC为2，渲染过程为ImageA->TextA & TextB。这里是否能够做一个假设，Unity的Batch过程中能够对batch的排序做一个简单的调整呢？那么再看下面的例子。
- 一个TextA的层级为0，一个IamgeA的层级为0，一个ImageB的层级为0，一个TextB的层级为1。按照上面的实验过程，是否DC为2呢，过程是否是ImageA & ImageB -> TextA & TextB。然而并不是，DC为3，过程为TextA -> ImageA & ImageB -> TextB。所以例子2的出现只能说是一个意外，当然我们能够同一个一些其他的手段，使得TextA与TextB能够进行Batch，那么最后的DC也是2。具体的做法在下面会介绍到。

2. 无Canvas但是有Mask情况

- 一个ImageA-Mask 层级为0，一个ImageB层级为2，一个ImageC层级为2，一个TextA层级为2，如果ImageA不存在Mask，那么DC应该是2【ImageA与ImageB、ImageC能够进行合并】，而有了Mask之后，DC为4，过程为ImageA-Mask -> TextA -> ImageB & ImageC -> AlphaClip
- 一个ImageA1-Mask 层级为0，一个ImageA2层级为2，一个ImageA3层级为2，一个TextA层级为2，两外一个ImageB1-Mask层级也为0,【因为两个Mask并没有重叠】，一个ImageB2层级为2，一个ImageB3层级为2，一个TextB层级为2，那么DC还是为4，过程为ImageA1-Mask & ImageB1-Mask -> TextA & TextB -> ImageA2 & ImageA3 & ImageB2 & ImageB3 -> AlphaClip
- 如果MaskB与MaskA发生了重叠了，那么层级就变成了，ImageB1-Mask层级为3，【应为MaskA子节点最高层级为2】，ImageB2层级为4，ImageB3层级为4，TextB层级为4，最后的DC为8，过程为 ImageA1-Mask -> ImageA2 & ImageA3 -> TextA -> AlpahClip -> ImageB1-Mask -> ImageB2 & ImageB3 -> TextB -> AlphaClip

3. 有Canvas情况

- 在Hierarchy中的排序有ImageA ,ImageB ,ImageC，原来DC应该为1,3个Image都能够进行合并。给ImageB添加了Canvas之后，DC为3，因为ImageB自成为一个DC的同时，把ImageA与ImageC的相邻关系也给打断了

1. 所有的事件触发区域都使用**EmptyGraphic**，其他的**Graphic**的**Raycast Target**都必须设置为**false**

- **EmpthGraphic** 是一个只有逻辑区域，但是没有显示区域的Graphic，不用产生DrawCall
- **Ratcast Target** 默认的Graphic【**Image**、**Text**】都会勾选上Raycast Target标记，表示当前的RectTransform区域点击有效，然而太多的Raycast Target会让EventSystem产生更多的消耗
- 所有的事件触发区域都使用**EmptyGraphic**，有利于进行UI的Despawn与Spawn统一操作。

2. 将所有的Image.color ,Text.Color都修改成**CanvasRenderer.color**

- 修改Iamge.color与Text.color都会造成Canvas Rebuild，但是修改CanvasRenderer.color则不会
- 大部分的颜色需求都能够通过调节CanvasRenderer的Color来实现效果‘
- 其实原理还不是很清楚，为什么修改了CanvasRenderer的color之后，UI还能够继续batch，但总归效果是喜人的。

3. **限制**UI GameObject的Set Active与Set Deactive的操作。

- 每次UI更新Active标记，都会造成的Canvas Rebuild消耗
- 使用CanvasRenderer.setAlpha(0)或者CanvasGroup.alpha=0的方式，同样能够使得UI SetDeactive的效果，并且不会产生DC与Canvas Rebuild，但是需要注意的是，此时的点击区域与Update更新还是需要手动关闭

- 将UI移除出屏幕外也是一种作法，但是该作法存在着较大的缺陷，**移除出屏幕外的UI同样计算DC**，如果移除屏幕外的UI重叠在一起，那么会造成DC的不可预期增长。并且也需要手动关闭Update更新
- 避免频繁的设置UI的Parent，因为每次设置UI的Parent都会造成Canvas Rebuild
- 建议通过中间代理【UIObejctRoot】进行UIGameObject的Deactive操作。

4. 尽量避免使用Mask

- 使用Mask会造成多余DC，几乎每多一个Mask，都会产生一个DrawCall【**尽管Mask也能够Batch**】
- Mask会打破统一的DC Batch机制，使得更难以控制DC数量
- 请优先使用RectMask2D，与Mask的原理不一样，不会产生多余DC，也不会影响DC的计算规则。当然有其特定限制，只能用于Rect区域。

5. 不使用Text的Best Fit
6. 尽量避免Unity提供的Outline与Shadow

- Outline与Shadow会产生多**4倍**的定点数，这是不能够忍受的
- 使用自己提供的**SingleOutline**,只多一倍的定点数，并且更符合美术的预期

设置UGUI元素位置的时候，就算设置的位置和原位置一样，也会导致canvas重建，所以设置位置的时候最好有一定的判断

UGUI 和 NGUI 都有一个类来存储顶点uv颜色等信息，每一次重新绘制或者更改都会重新填充其中的数据

outline和tild image 长文本 最好不要是动态的 ，绘制消耗太大 。outline会复制5个原网格，定点数和边数增加5倍

补充UGUI合批知识点

其实原理很简单：对于每一个UI元素，对应其材质和shader找到一个batch，没有或者有但是被其他batch的UI挡住了就要新生成一个batch，
否则就合到已存在的batch。