深入理解Android屏幕刷新机制与View三大流程关系

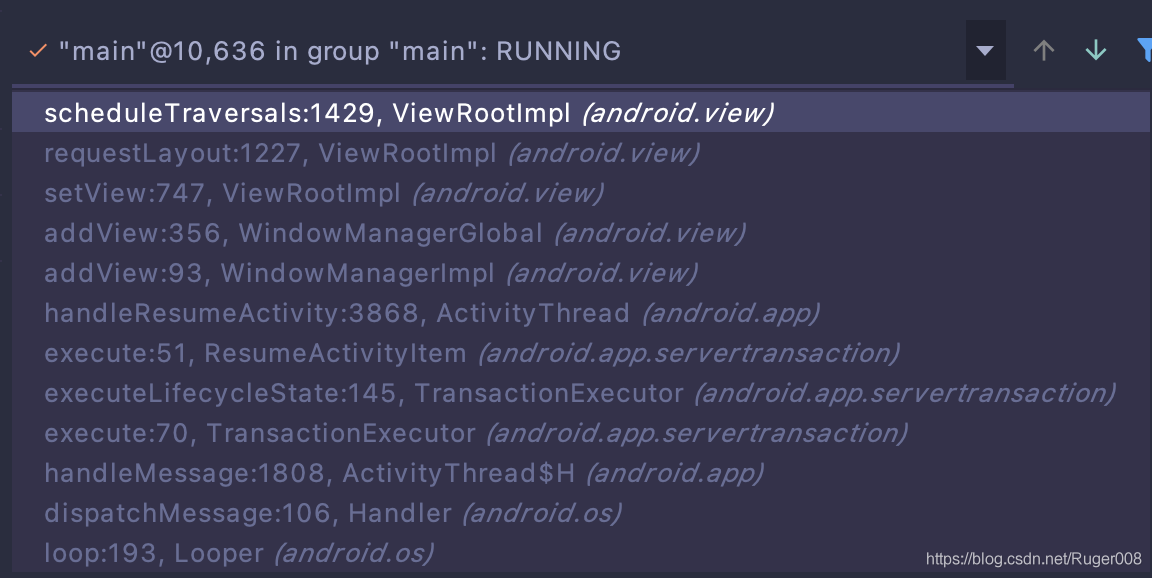
博客仅作为个人学习使用，不定期更新，方便以后复习时能够快速拾回这段遗忘的知识。如果有读者无意间搜索到这篇文章，阅读后觉得对您有所帮助，希望能点个赞，感谢支持。

requestLayout与invalidate

       众所周知，View三大流程是Measure、Layout、Draw，在SDK中我们在View中只能找到 requestLayout (包括Measure与Layout)与 invalidate (包括Draw)这两个方法。合并Measure与Layout是可以理解的，既然当View尺寸发生变化时说必然需要重新Measure和Layout的，这两个流程其实是密不可分的。可以看出仅需分析这两个方法就足以覆盖View三大流程，这两个方法的实现都在 ViewRootImpl 中， ViewRootImpl 作为 DecorView 的父View，在View渲染过程中充当窗口的角色与WMS直接进行通讯。明确了目标后，我们继续进行分析。

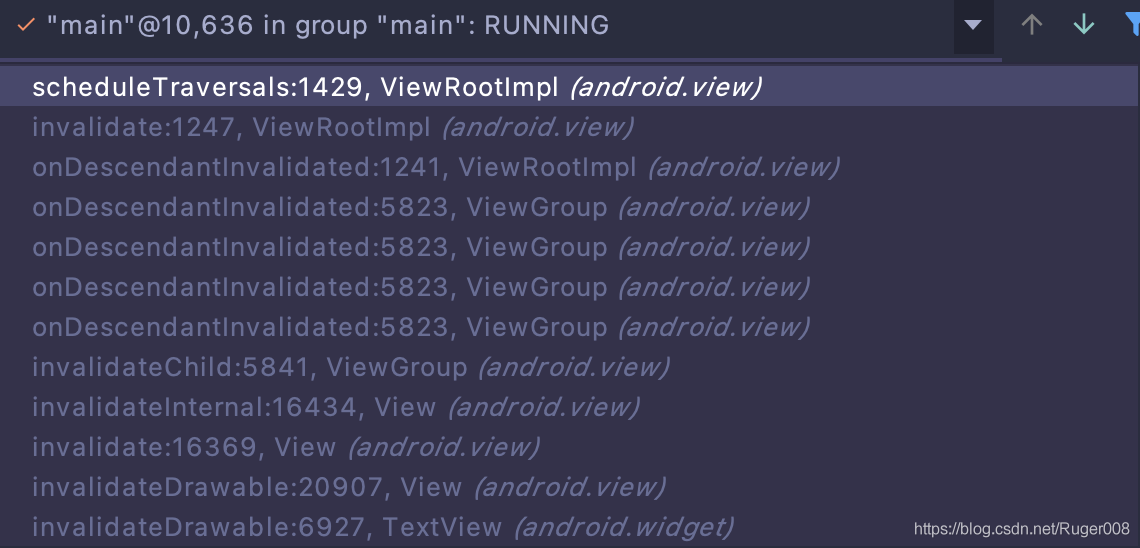
requestLayout

       首次 requestLayout 是由 ViewRootImpl 自发调用（以后只能由子View向上传播来调用）。可以看出这里间接调用了 scheduleTraversals 。



invalidate

       我们都知道 invalidate 作用是重绘，仍然会走到 scheduleTraversals 。所以我们就分析分析这个神秘的 scheduleTraversals 。插句题外话，有的小伙伴可能就要问了，难道 invalidate 和 requestLayout 都要重新执行一边三大流程嘛，那么区分还有意义吗？其实不是的，在 requestLayout 中会设置mLayoutRequested，进而实际在 doTraversals 过程中内部判断是否要执行 performMeasure 、 performLayout 或 performDraw 。

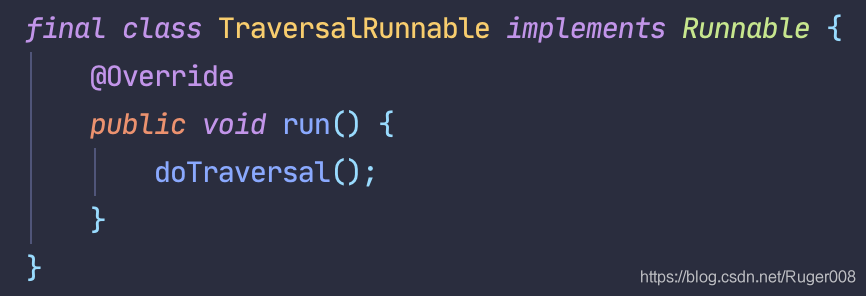


scheduleTraversals

       从名字上就可以看得出来，既然叫schedule, 那么Traversals过程肯定是异步调用的，让我们翻翻源码看一下。这里有两个过程比较重要，一个调用了 postSyncBarrier ，另一个调用了 postCallback ，后面会分析这两者分别做了什么。



        别忘了我们来这里的目的，是为了看看 doTraversals 是怎么被调用到的，其实 doTraversals 过程被包裹在了mTraversalRunnable之中。



       可以看出最后的 doTraversal 是交由**Choreographer**回调完成的。所以我们这接下来的任务就是分别分析 postSyncBarrier 与 postCallback 了。

postSyncBarrier与postCallback

postSyncBarrier

       这里就涉及到**Handler**同步屏障机制了，同步屏障机制的目的很明确就是为了让某些特殊信息可以更快的得到相应，此处的目的就是让绘制消息优先级超过主线程中其他消息，从而避免绘制不及时导致的用户界面卡顿。

       每个线程**Looper**中的**MessageQueue**在获取下一个**Message**的时候，如果碰到了同步屏障，那么不会取出这个同步屏障，而是会遍历后续的**Message**，找到第一个异步消息取出并返回。这里跳过了所有的同步消息，直接执行异步消息。所以同步屏障机制就像它的名字一样，可以屏蔽同步消息，让异步消息得以快速执行。

       值得注意的是同步屏障不会自动移除，使用完成之后需要手动进行移除，不然会造成同步小心无法被处理。所以对应到代码中是什么时候移除同步屏障的呢？

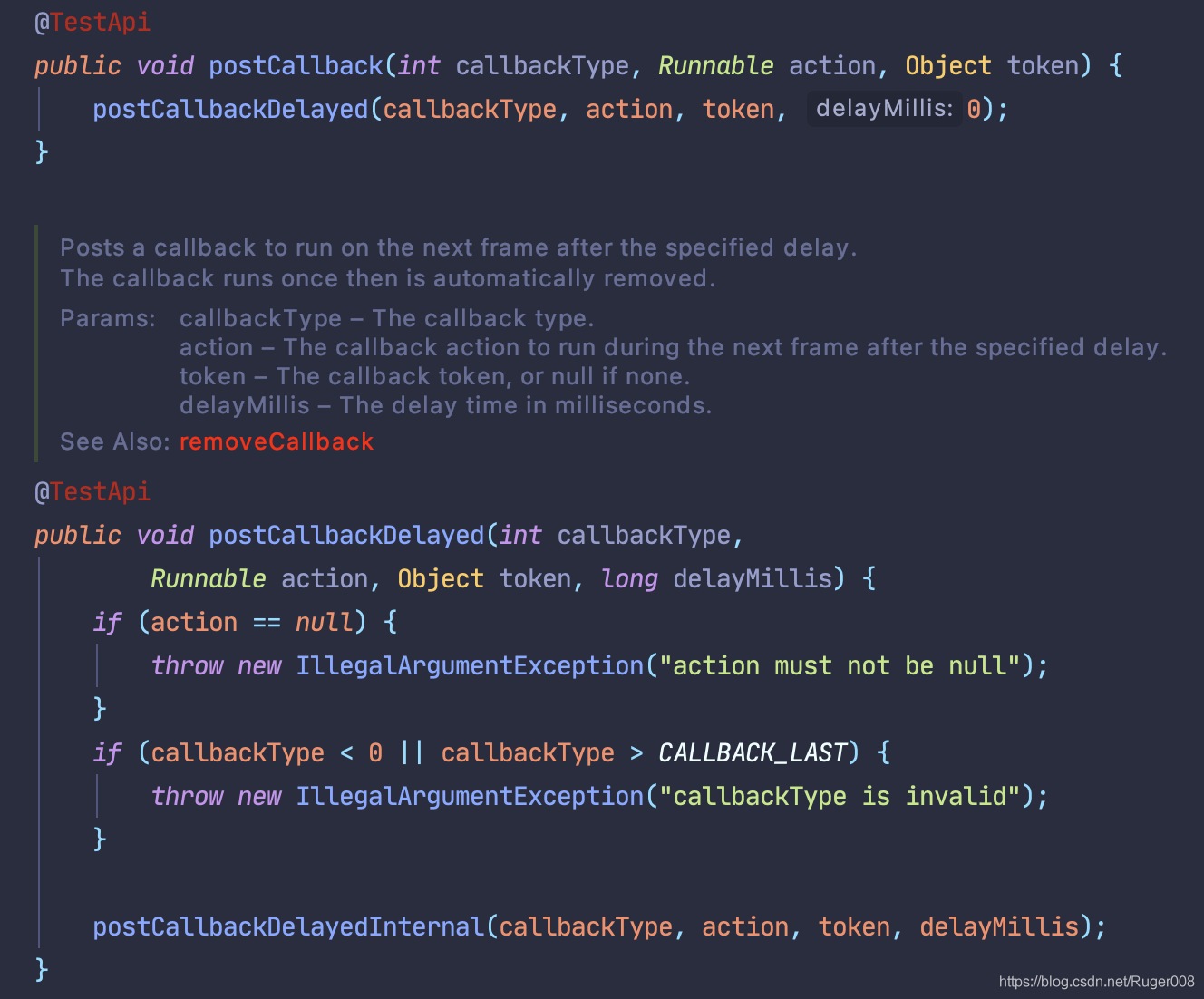


       可以看出当 doTraversals 被**Choreographer**回调时会移除之前添加的同步屏障。有关同步屏障机制就讲这么多，这里就不着重分析**MessageQueue**源码了，因为这已经偏离了这篇文章的主题。

感兴趣的话可以拓展阅读 https://juejin.cn/post/6940607471153053704#heading-1。

       那么问题来了，doTraversal何时才会被Choreographer编舞者回调呢？我们需要继续分析postCallback了。

postCallback

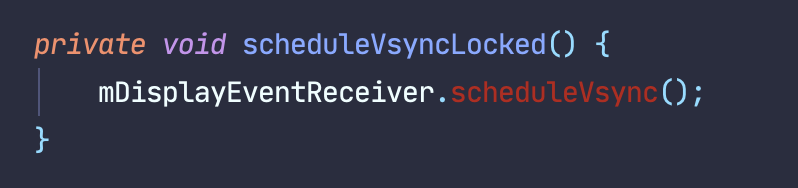


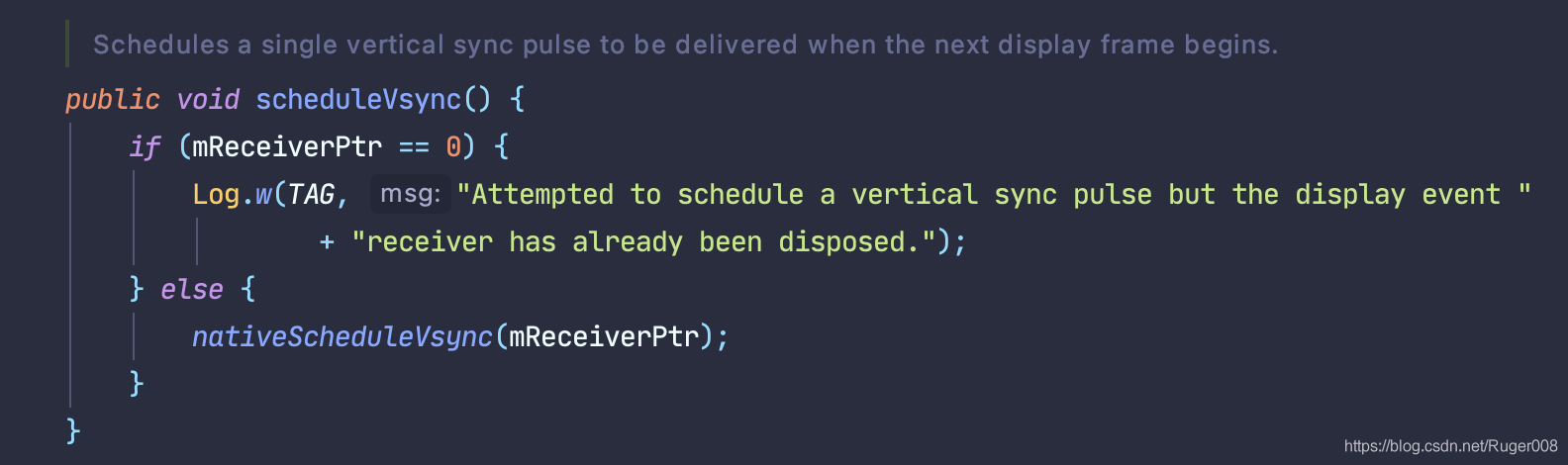


       postCallback 会走到 postCallbackDelayedInternal ，传入的runnable也会被存储到这个mCallbackQueues之中。由于传入的delayMillis为零，所以此时dueTime等于now会走第一个逻辑分支调用 sheduleFrameLocked 。



       一路最终会走到 scheduleVsyncLocked ，这里为什么 isRunningOnLooperThreadLocked 返回true，只因其内部判断当前线程**Looper**是不是构建**Choreographer**时传入的**Looper**，由于都是在主线程所以必然返回true。我们继续跟下去。





       跟到这里我们看到这个native前缀就不用继续跟下去了，因为其具体实现已经写到了native层，通过JNI进行的方法调用。那么这个Vsync到底是什么东西呢？

屏幕刷新与Vsync脉冲

       由于本人并没有深入native分析具体底层实现，所以只能谈谈自己对于Vsync脉冲的理解，如有不对希望大家能够指出。目前Android手机的主流显示屏幕有LCD也有OLED，虽然不同屏幕的次像素排列以及工作过程都不同，所有硬件实现细节都被抽象成HAL层接口，并由ART层**SurfaceFlinger**进行封装，但是这些底层实现原理都是不需要我们关注的。屏幕刷新率是近年来比较热门的话题，很多厂商通过插帧技术甚至能够实现120赫兹的高刷新率。那么屏幕刷新率是如何体现到SDK中的呢？这里就涉及到了Vsync脉冲。对于60赫兹刷新率的手机，每 16ms（1s / 60 = 0.016666667 ）屏幕硬件会发送一个Vsync脉冲通知界面进行刷新。所以可以看出的是，Vsync脉冲频率越高，若存在UI绘制任务则渲染频率也会越高，直接导致的问题就是更加耗电。

感兴趣的话可以拓展阅读 https://cloud.tencent.com/developer/article/1685247。

       那么对于Vsync信号的处理在哪呢？



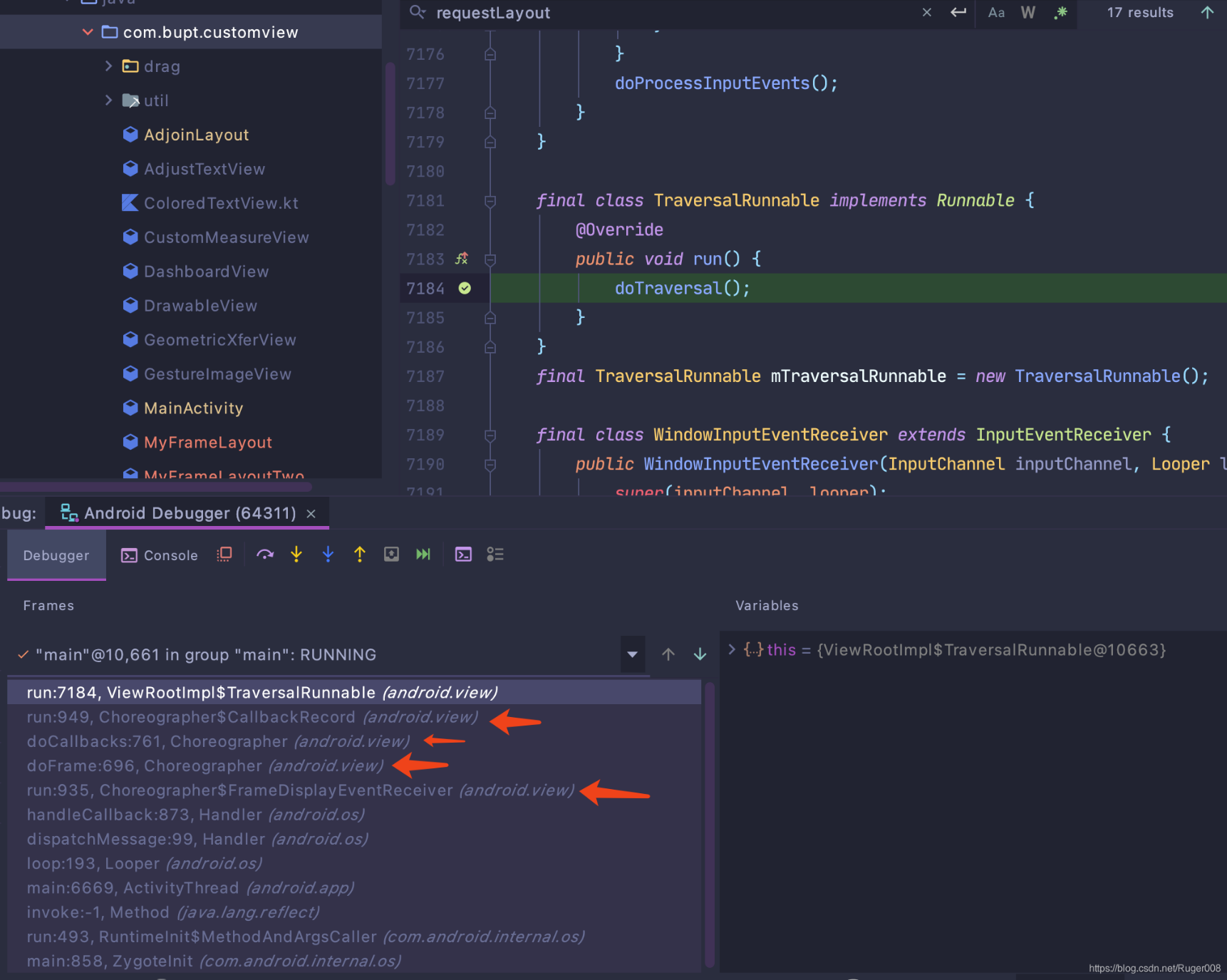
       注意的是前面调用scheduleVsync的实例类型就是这个**FrameDisplayEventReceiver**，也就是说这个类主要就是处理Vsync信号的。在 onVsync 回调中发送了一个异步消息到消息队列，由于存在同步屏障机制，此时的callback会优先得到处理，逻辑便会走进 doFrame 了。



       可以看出 doFrame 中间接调用了 doCallbacks 方法，还记得我们在 postCallBack 传入的callbackType嘛？那就是CALLBACK\_TRAVERSAL。



       首先他会从之前的mCallbackQueues取出所有CALLBACK\_TRAVERSAL类型的callback，然后分别执行，执行后将所有callback进行销毁，避免重复执行，此时doTraversals就会被回调了。不信？动态调试给你看。



这证明了他们之间的关系。

同步屏障的存在可能引发IdleHandler任务不执行

有两种场景可能会触发这个问题。

1. 自定义View在重写的onDraw中直接或间接调用invalidate()。
2. 重写onMeasure、layout中直接或间接调用requestLayout()。

第一种场景出现的可能性高一些，但是为什么会出现这种情况呢？

IdleHandler是什么

说句实话，我还没有遇到使用它的场景，不过弄懂它的原理，在解决类似场景问题时能够想到有这个东西就足够了。

       就像它的名字一样，当主线程不忙碌时才能去执行**IdleHandler**里的任务。（主线程**Looper**中的**MessageQueue**没有Message时 或 当前还没有到达下一条message开始执行的时间）

  
       补充一点，这里可以看到 queueIdle 有一个返回值keep，其目的很明确，即是否运行**IdleHandler**中的任务在下次空闲时能够再次执行。

IdleHandler任务为何会不执行呢？

       原因很简单，在自定义View重写的 onDraw 方法中直接或间接调用了 invalidate 方法，则会出现一个死循环。每次 invalidate 时都会向主线程post一个同步屏障，然后等待下次Vsync脉冲的到来再进行重绘任务。虽然这并不影响主线程其他任务的执行（由于同步屏障也作为一个同步消息，每次都加入到队列末尾，等待轮询到队首后才会发挥作用）。所以这也导致了主线程**Looper**的消息队列永远不会出现空闲状态，所以**IdleHandler**任务也不会得到执行。

如何解决这个问题？

       既然知道了原理，那么解决这个问题的最有效方法就是在 onDraw 中对于 invalidate 的调用增加一些逻辑限制条件，避免无限循环的 invalidate 调用，但具体要依赖使用场景来找到适合的解决方案。不管如何，大家懂了原理后，在开发中能够想起有这么一回事，那么这篇文章的目的就达到了。感谢阅读。