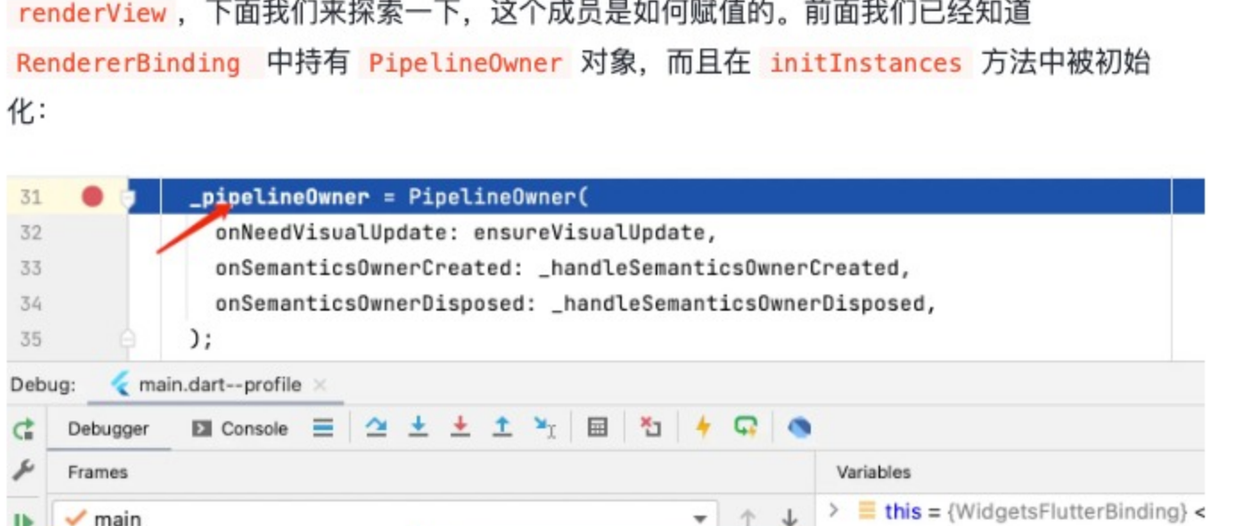
一、认识 PipelineOwner 的成员属性

前面的 BuildOwner 是负责对 Element 元素的维护管理，这里 PipelineOwner 就是对 RenderObject 渲染对象的维护管理。从下面可以看出，其中有这三个 RenderObject 列表。

- PipelineOwner
  - rootNode → AbstractNode?
  - _rootNode → AbstractNode?
  - rootNode(AbstractNode? value)
  - _nodesNeedingLayout → List<RenderObject>
  - _nodesNeedingPaint → List<RenderObject>
  - _nodesNeedingCompositingBitsUpdate → List<RenderObject>

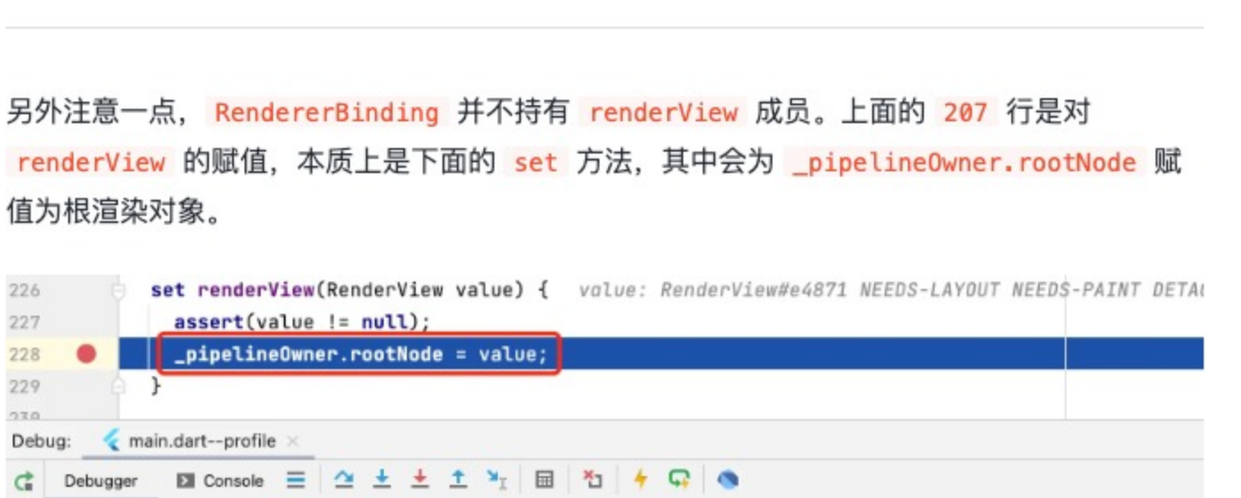
1. AbstractNode类型的 rootNode

看到 rootNode ，而且是 AbstractNode 类型的对象，自然联想到它就是根渲染对象 renderView ，下面我们来探索一下，这个成员是如何赋值的。前面我们已经知道 RendererBinding 中持有 PipelineOwner 对象，而且在 initInstances 方法中被初始化：



紧接着，在 initInstances 中会触发 initRenderView 方法。207 行会对 renderView 赋值，这也是根渲染对象实例化的场合。

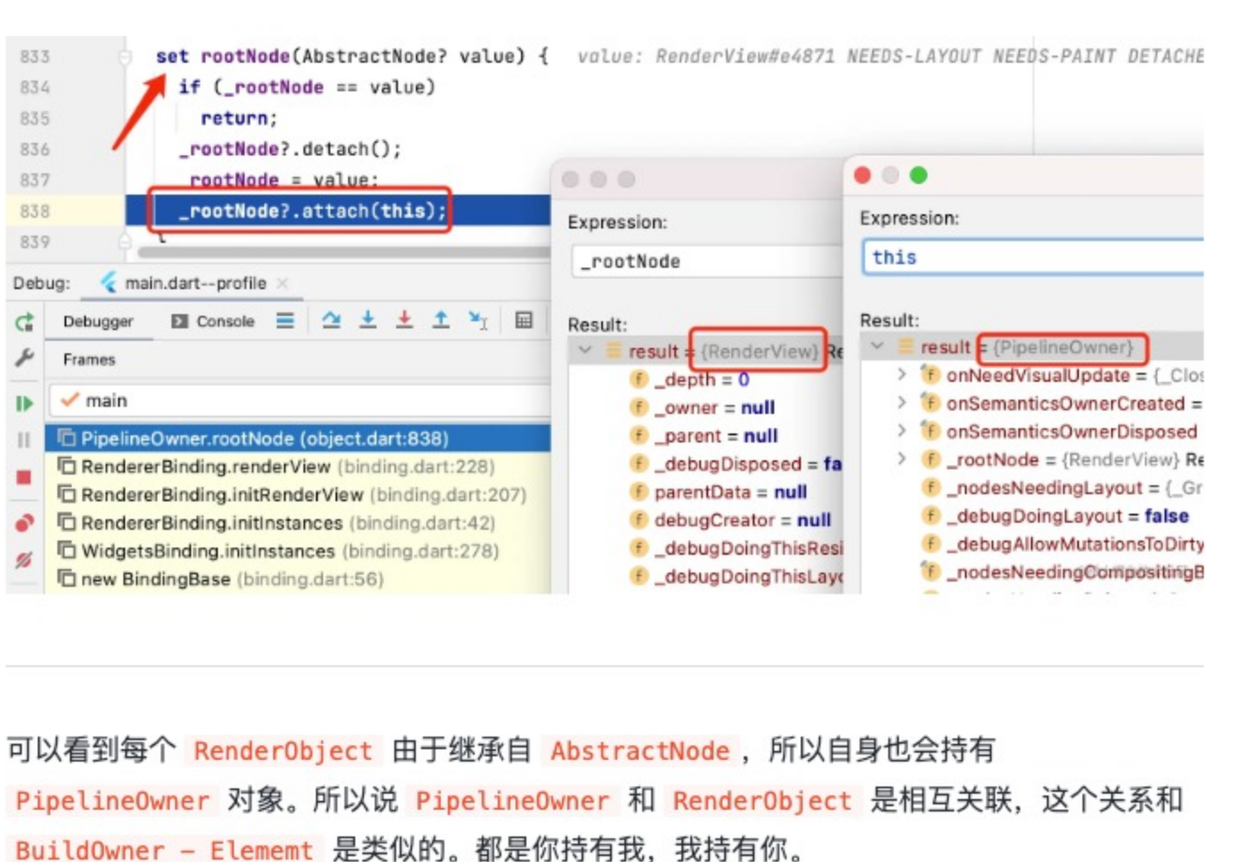
另外注意一点，RendererBinding 并不持有 renderView 成员，上面的 207 行是对 renderView 的赋值，本质上是下面的 set 方法，其中会为 _pipelineOwner.rootNode 赋值为根渲染对象。



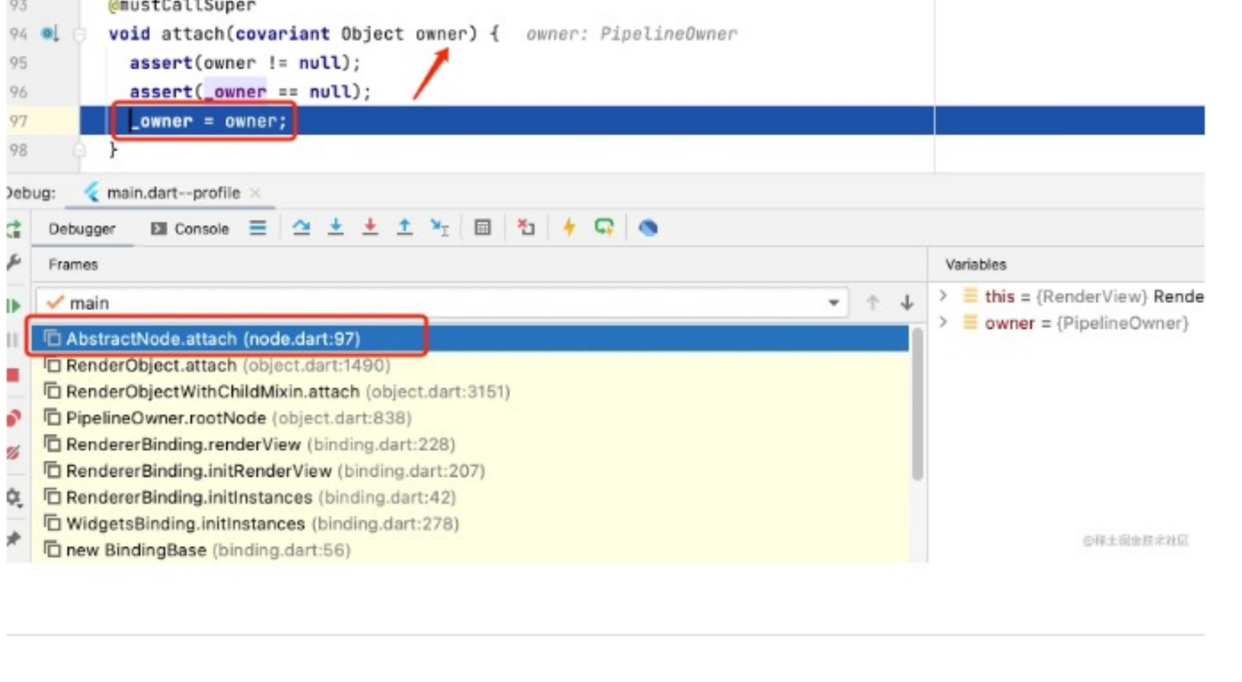
另外 RendererBinding 获取 renderView 的方式本质上也是通过 _pipelineOwner.rootNode 拿到的。从这里就可以看出，_pipelineOwner 中持有的 rootNode 在运行时确实是根渲染对象 RenderView 。

```dart
RenderView get renderView => _pipelineOwner.rootNode! as RenderView;
```

最后，在 PipelineOwner 设置 rootNode 属性时还有一点小细节。如下在 838 行，rootNode 会触发 attach 方法，入参是 this 。通过计算器可以看一下当前的两个对象，如下所示：



可以看到每个 RenderObject 由于继承自 AbstractNode ，所以自身也会持有 PipelineOwner 对象。所以说 PipelineOwner 和 RenderObject 是相互关联，这个关系和 BuildOwner - Element 是类似的，都是你持有我，我持有你。

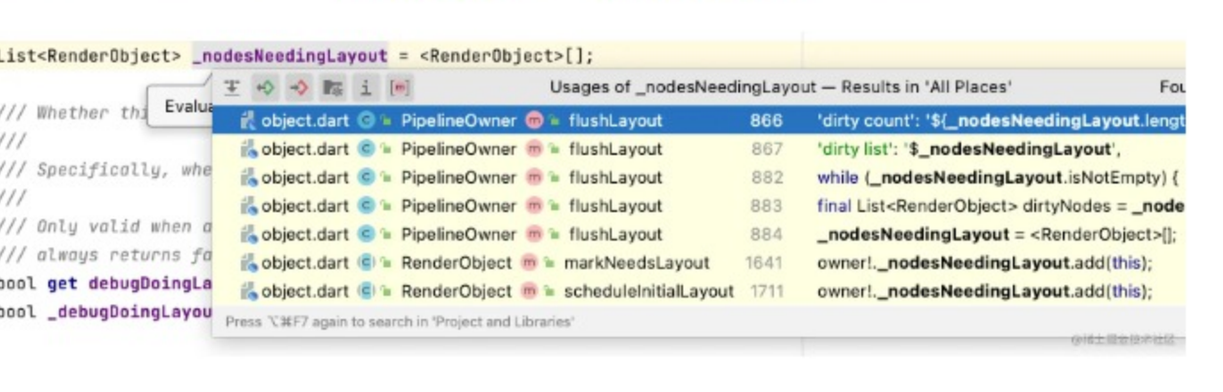


2. 三个渲染对象列表:

可以类比于 BuildOwner 中对脏元素的收集，PipelineOwner 中会通过 _nodesNeedingLayout 列表收集需要执行布局的渲染对象。

```dart
---->[PipelineOwner]----
List<RenderObject> _nodesNeedingLayout = <RenderObject>[];
```

可以看出这个列表主要是在 RenderObject 的 markNeedsLayout 方法中加入元素。

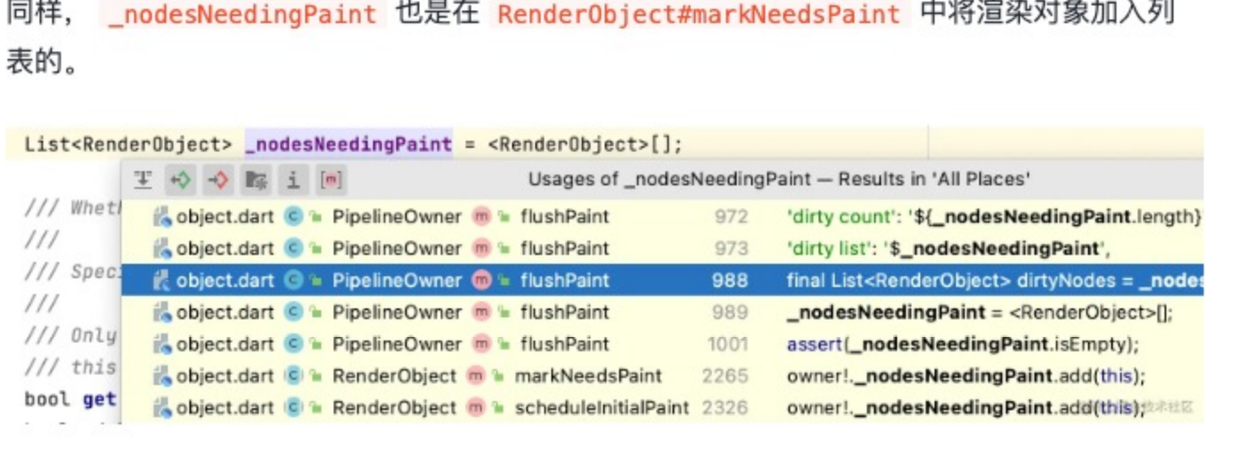


这时可以再类比一下 Element 的 markNeedsBuild 方法，会把当前元素加入到 BuildOwner 维护的脏元素列表中，但并非立刻更新。这是类似的如此，RenderObject 的 markNeedsLayout 由于标记某个渲染对象需要重新布局，也并非立刻触发渲染对象的更新逻辑。

另外，对于需要重新 绘制 和 合成 的渲染对象，也会通过相关列表进行维护。

```dart
---->[PipelineOwner]----
List<RenderObject> _nodesNeedingPaint = <RenderObject>[];
final List<RenderObject> _nodesNeedingCompositingBitsUpdate = <RenderObject>[];
```

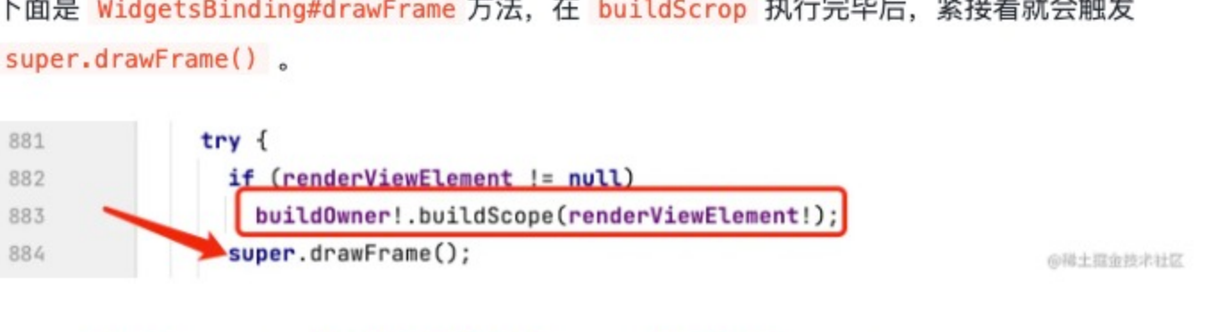
同样，_nodesNeedingPaint 也是在 RenderObject#markNeedsPaint 中将渲染对象加入列表的。



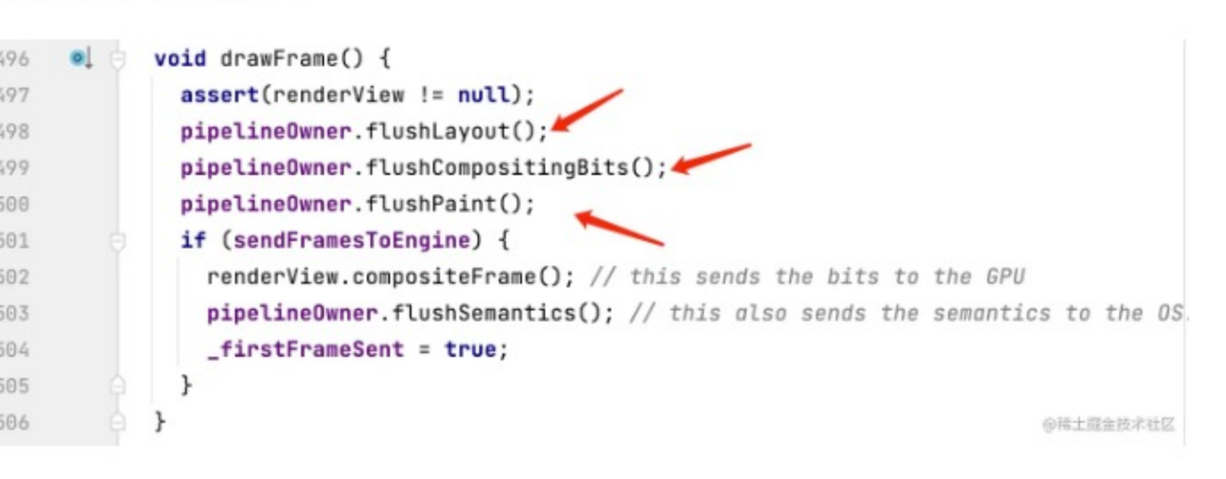
_nodesNeedingCompositingBitsUpdate 也是在 RenderObject#markNeedsCompositingBitsUpdate 方法中将渲染对象加入列表的。也就是说，RenderObject 的这三个 markNeedsXXX 的方法，会为 PipelineOwner 中对应的列表添加渲染对象。

3. 何时真正触发布局、绘制、合成

之前说过 BuildOwner#buildScrop 会在新帧到来时，在 drawFrame 中触发，从而对脏元素同一重建。其实对于 PipelineOwner 也是一样，他俩可以说是 drawFrame 的中的 "邻居" 。下面是 WidgetsBinding#drawFrame 方法，在 buildScrop 执行完毕后，紧接着就会触发 super.drawFrame() 。



而这个 super 指的就是 RendererBinding ,在其 drawFrame 方法中，会触发如下三个方法，真正进行 布局 、 合成 和 绘制 。从这里可以看出，先对脏元素进行从新构建，然后再对相关的渲染对象进行处理。



那接下来，就看一下，pipelineOwner.flushXXX 中具体逻辑是什么。

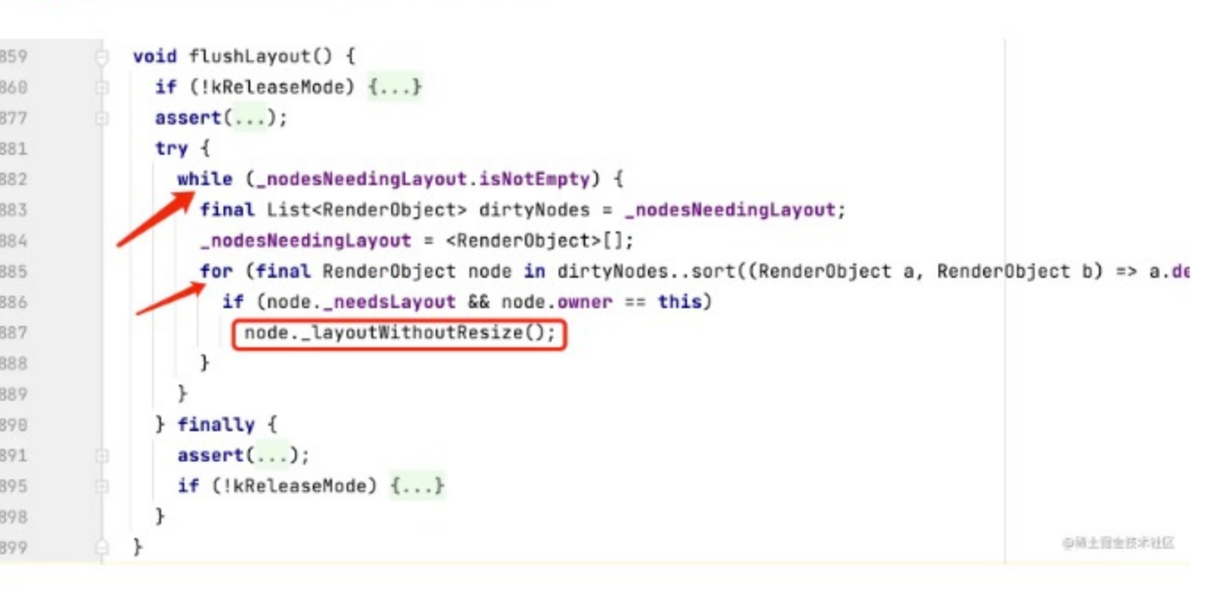
二、认识 PipelineOwner 的成员属性

主要成员方法如下:

- PipelineOwner
  - requestVisualUpdate() → void
  - flushLayout() → void
  - flushPaint() → void
  - flushCompositingBits() → void
  - _enableMutationsToDirtySubtrees(VoidCallback callback) → void

1.flushLayout：更新布局

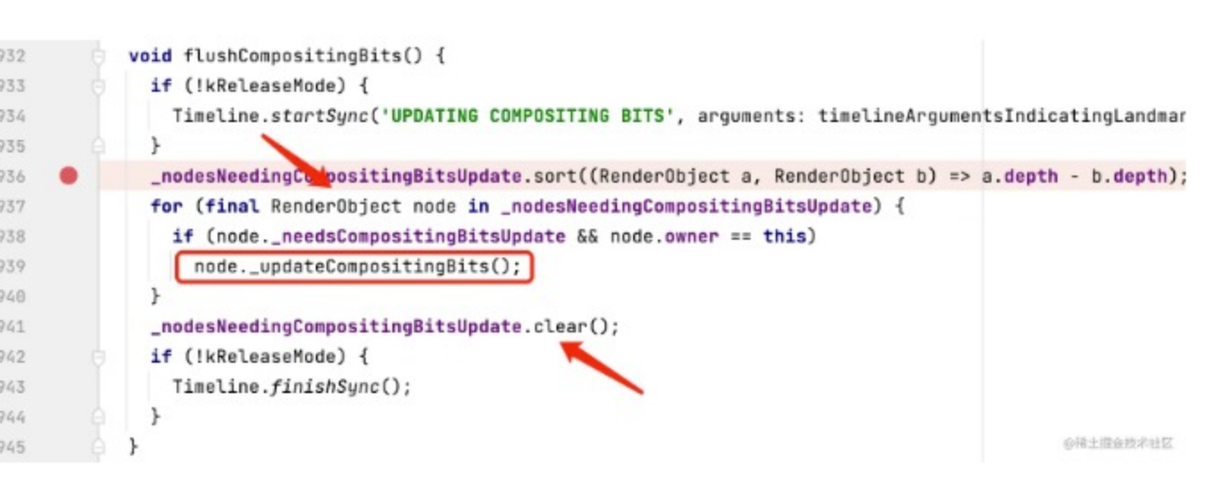
如下是 PipelineOwner#flushLayout 方法的实现，其核心是遍历 _nodesNeedingLayout ，也就是 脏布局渲染对象列表 ，如果其中节点的 _needsLayout 属性为 true ，会触发渲染对象的 _layoutWithoutResize 方法进行布局。



可能这里有人会有疑问，问什么这里不是直接对 _nodesNeedingLayout 列表进行遍历，而是用了一个 while 循环，和一个 for 循环，有这个必要吗？我们仔细分析一下，其中 while 循环的进入条件是 _nodesNeedingLayout 列表非空，而 884 行显示地将列表置空。这两行代码看似好像没什么意义，但在渲染元素触发 _layoutWithoutResize 方法的过程中，可能导致 _nodesNeedingLayout 列表加入值。这就像在脏元素重新构建过程中可能再次变脏一样，所以这里的代码是很漂亮的。

2. flushCompositingBits：更新合成位

这个方法是更新渲染绘制中的 _needsCompositing 的 bool 型标识，这个在下一篇介绍渲染对象时再进行详述。另外，当此方法执行完后 _nodesNeedingCompositingBitsUpdate 列表会被清空。

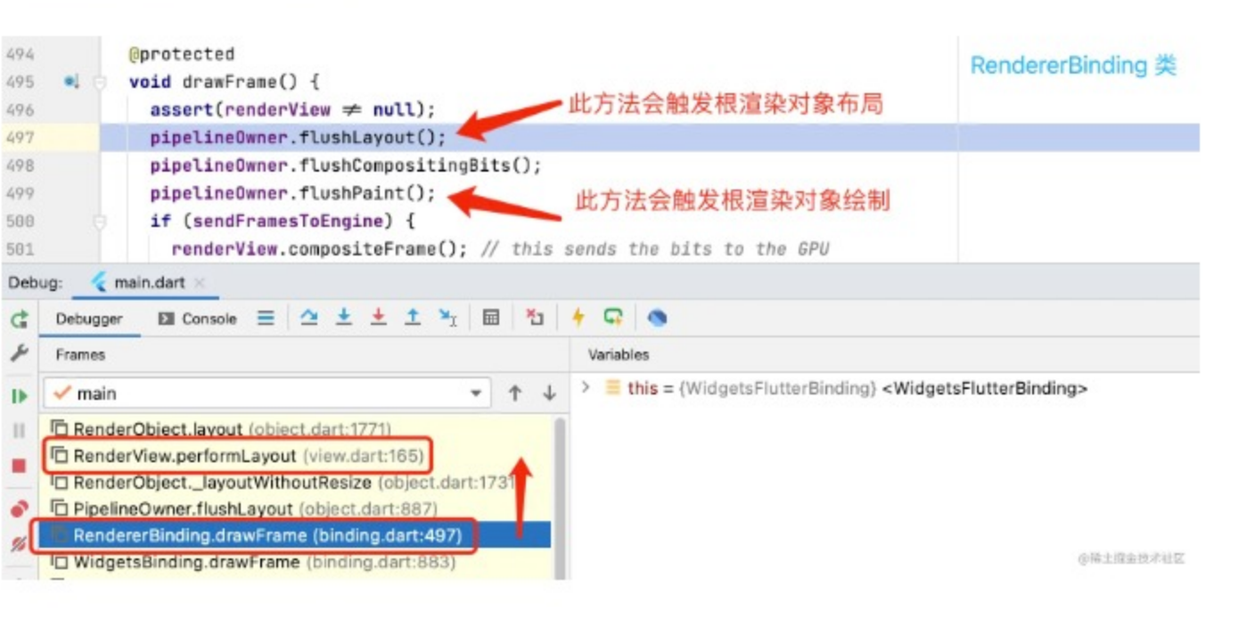


3.flushPaint： 更新绘制

如下是 PipelineOwner#flushPaint 方法的实现，其核心是遍历 _nodesNeedingPaint ，也就是 脏绘制渲染对象列表 ，如下所示，也是会有 _needsPaint 标识来决定是否需要进行绘制。

4. 根渲染对象的布局、绘制

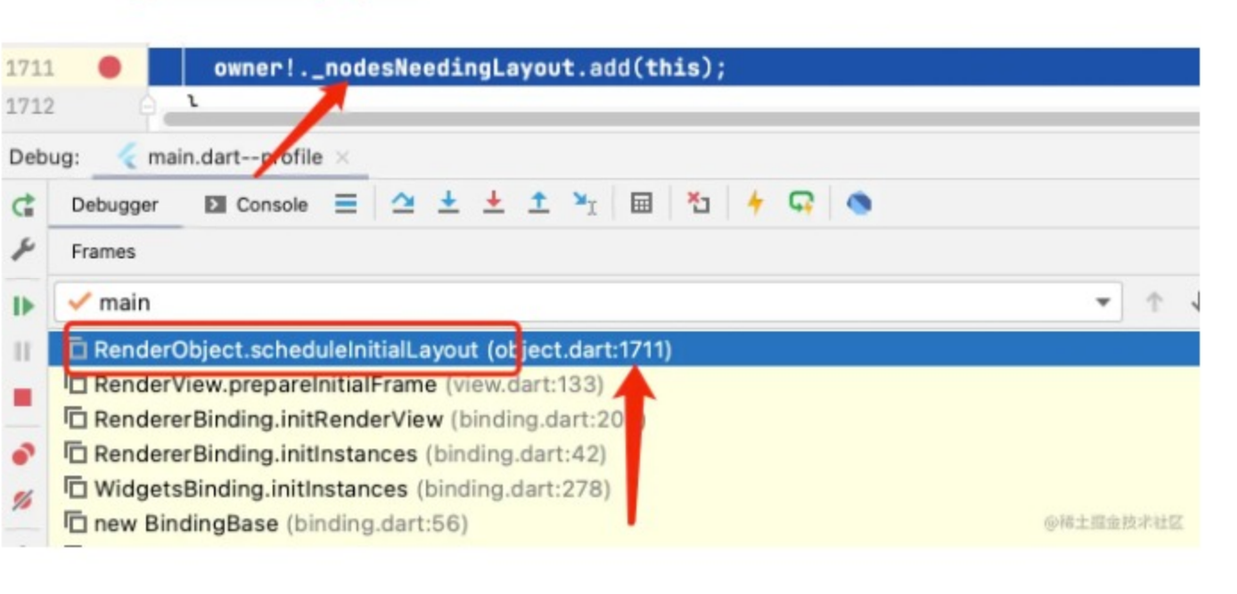
在 《Flutter 布局探索 - 薪火相传》 中，我们介绍过，在程序开始时，PipelineOwner#flushLayout 方法会触发根渲染对象 RenderView 进行布局的。从而对渲染树的所有渲染对象进行布局。



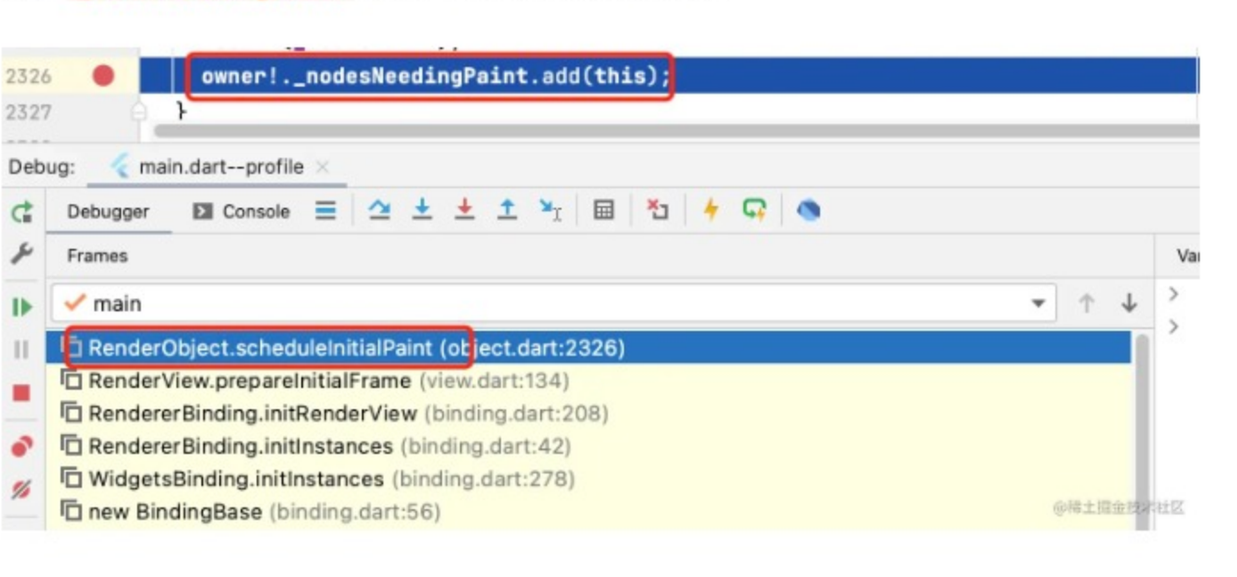
这是 RenderView 比较特殊的待遇，在 initRenderView 方法中，会 RenderView#prepareInitialFrame 方法，为初始帧做准备。

```dart
---->[RenderView#prepareInitialFrame]----
void prepareInitialFrame() {
  scheduleInitialLayout();
  scheduleInitialPaint(_updateMatricesAndCreateNewRootLayer());
}
```

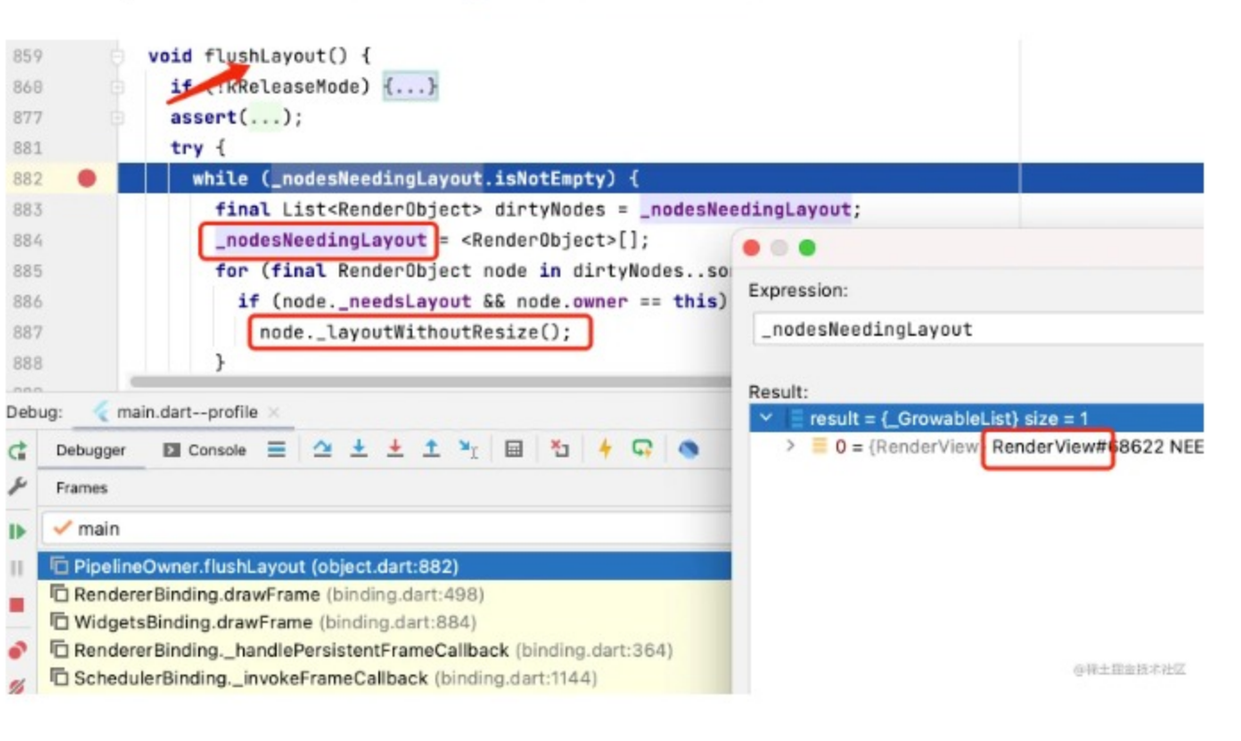
其中会为 _nodesNeedingLayout 列表添加渲染对象。



以及 _nodesNeedingPaint 列表，也会加入根渲染对象。



也就是说，在程序一开始时 flushLayout 和 flushPaint 会对根渲染对象进行布局和绘制。



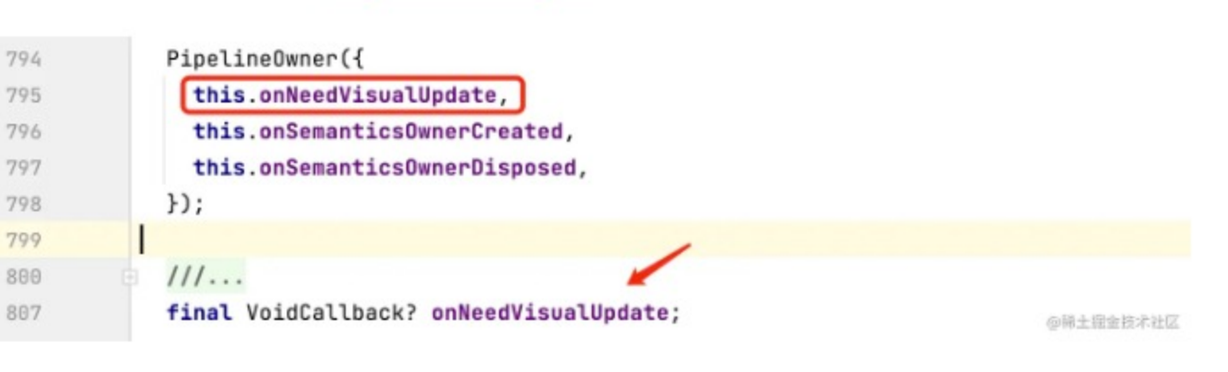
对于渲染树而言，根节点触发布局和渲染，子树节点都会触发。这就是为什么一开始就会触发全部渲染节点布局和绘制的原因。

5. requestVisualUpdate：申请更新

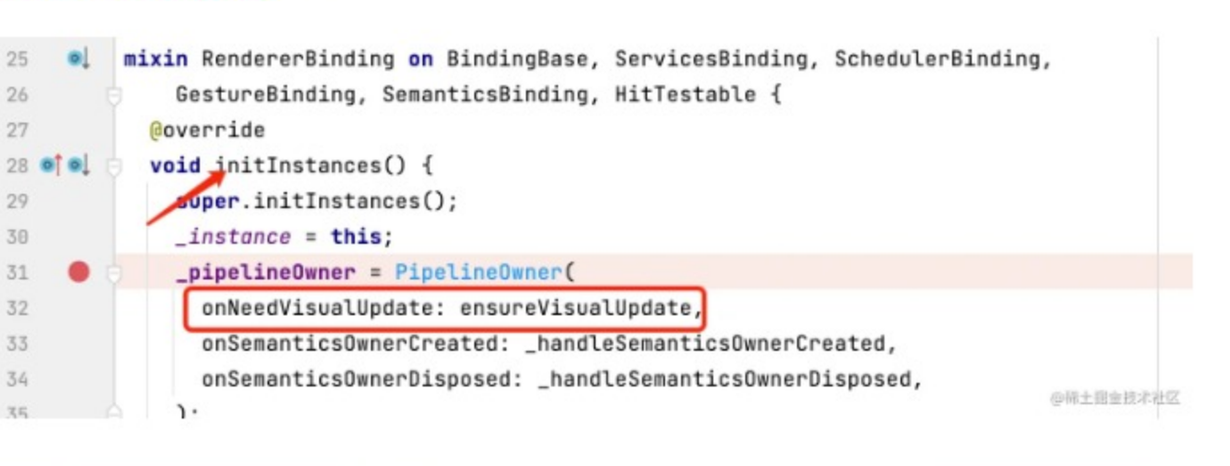
其中还有一个方法 requestVisualUpdate ，它会触发成员 onNeedVisualUpdate 函数。

```dart
void requestVisualUpdate() {
  onNeedVisualUpdate?.call();
}
```

而 onNeedVisualUpdate 成员会在构造中被初始化，所以只要知道 PipelineOwner 初始化中对应参数是什么，就知道 requestVisualUpdate 本质上触发的是什么逻辑。



我们又知道 PipelineOwner 在 RendererBinding#initInstances 中被实例化。如下参数是 ensureVisualUpdate 方法。



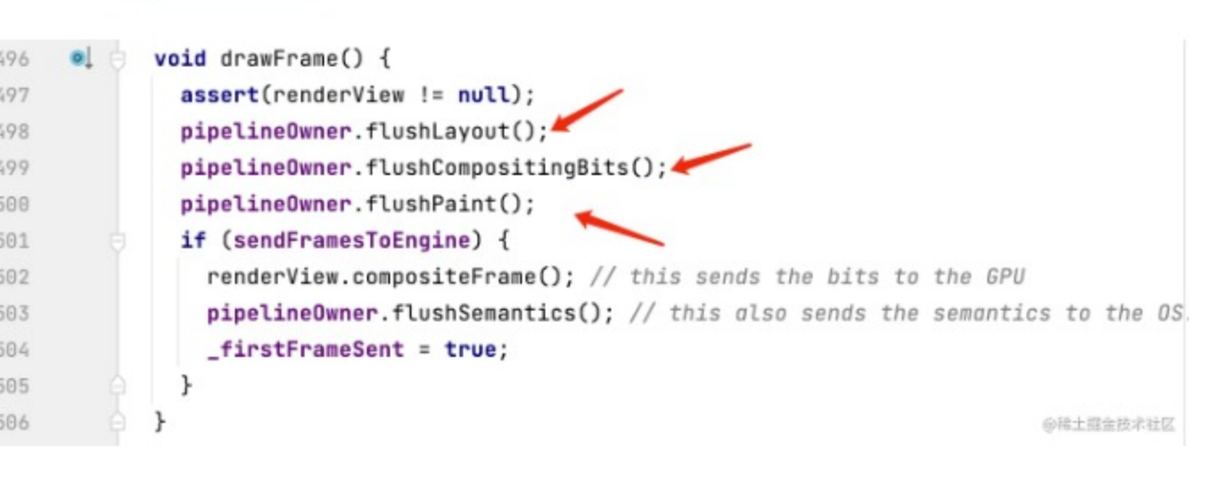
看到 ensureVisualUpdate 是不是有点熟悉，没错，它就是那个可以通过 scheduleFrame 方法申请新帧调度的方法，State#setState 方法也会来到这里。

```dart
---->[SchedulerBinding#ensureVisualUpdate]----
void ensureVisualUpdate() {
  switch (schedulerPhase) {
    case SchedulerPhase.idle:
    case SchedulerPhase.postFrameCallbacks:
      scheduleFrame();
      return;
    case SchedulerPhase.transientCallbacks:
    case SchedulerPhase.midFrameMicrotasks:
    case SchedulerPhase.persistentCallbacks:
      return;
  }
}
```

三、PipelineOwner 小结

总的来看 PipelineOwner 就是维护了需要执行 布局 、 合成位 、 绘制 的渲染对象，通过 RenderObject#markNeedsXXX 在列表中添加渲染对象，通过 PipelineOwner#flushXXX 遍历相关列表，执行相关逻辑。我们看到了 元素树的维护 和渲染对象 布局 、 绘制 在逻辑上的先后顺序：

也看到了 PipelineOwner 在绘制帧时的表现：



另外我们知道了，RenderObject 在关联到渲染树中，会持有父级的 PipelineOwner 。也就是说，每一个渲染对象持有的 PipelineOwner 来源于根渲染节点 RenderView 。而 RenderView 在实例化之后，就会和 PipelineOwner 进行关联。

除了 布局 、 合成位 、 绘制 之外，其实 PipelineOwner 也维护了第四类渲染对象，Semantics 相关的渲染对象。前三类就已经够复杂了，关于 Semantics 这里就不引申了，有兴趣的可以自己研究一下，流程应该和前三者类似，就不再花笔墨了。

- PipelineOwner
  - _nodesNeedingSemantics → Set<RenderObject>
  - flushSemantics() → void
  - semanticsOwner → SemanticsOwner?
  - _semanticsOwner → SemanticsOwner?
  - debugOutstandingSemanticsHandles → int
  - _outstandingSemanticsHandles → int
  - ensureSemantics({VoidCallback? listener}) → SemanticsHandle
  - _didDisposeSemanticsHandle() → void

接下剩下最后一点非常重要的内容，那就是 RenderObject#markNeedsXXX 的触发逻辑，这关系到 PipelineOwner 相关列表的长度，还有 RenderObject 中对于布局、合成位、绘制都有相关的 bool 标识进行限制，它们在 RenderObject 中是如何维护的，这对渲染机制都非常重要。下一章，将进军 RenderObject ，探索继续 ~