Flutter 调研文档

本文包含以下内容：

1.Flutter 简介

Flutter 是什么？

Flutter 是一个由谷歌开发的开源移动应用软件开发工具包，用于为 Android 和 iOS 开发应用，同时也将是 Google Fuchsia 下开发应用的主要工具。Flutter 框架包含了两套匹配特定设计语言的组件。称作 Material Design 的组件实现的是同名的谷歌设计语言，称作 Cupertino 的组件模仿了苹果 iOS 的。---维基百科

简单来说它与 RN、Ionic、Weex 等众多混合开发平台框架一样，都是一套代码实现多平台发布的跨平台框架。Flutter 是以 dart 为基础做出的一套 SDK，支持在 Android 和 iOS 上构建 APP。

Flutter 优缺点：

优点：
1.跨平台。一份代码，在 Android 和 iOS 平台上都可以运行，并且 Widget 外观符合各平台特色，在 Android 手机上符合 Material Design，在 iOS 手机上符合 Cupertino 风格；
2.支持快速开发。Hot Reload 的特点使得开发者修改代码后无须重新运行 APP，只需保存代码后即可在手机或模拟器上立即看到效果，根据官方的描述，等待时间不到 1 秒；
3.运行渲染速度快。生成的 APP 并没有使用原生系统中提供的控件,也没有使用 WebView，而是借助可移植的 GPU 加速的渲染引擎以及通过 AOT 编译生成的高性能本地 ARM 代码实现渲染，摆脱了原生控件的限制，流畅性不输原生界面；
4.支持在已有的原生 APP 项目中使用，只需在界面中嵌入 FlutterView 即可；
5.开源。核心代码可以方便的获取到，便于开发者了解运行机制和 DEBUG；
6.组件的兼容性。Flutter 提供的 widget 都是基于 skia 来实现和精心定制的，与具体平台没关，所以能保持很高的跨 os 跨 os version 的兼容性。

缺点：
1.APP 包比较大。由于依赖 Flutter 引擎，其 C/C++代码用 NDK 编译成各个 CPU 架构的 so 库，导致生成的 APK 文件比较大。当然基本所有的跨平台开发工具都有此缺点。官方描述生成的 APK 文件最小约为 6.7M；
2.Dart 语言的可读性与可维护性稍差，布局比较复杂；

3.由于发布时间比较短，第三方类库还比较少；
4.目前仅支持 2D 显示，但根据官方计划将来会有对 3D 的支持；
5.由于 Dart 代码会 AOT 编译到 native 代码中，可能会给 iOS 应用的热更新、插件化等技术带来困难；
6.由于并非使用原生的控件，其特有的绘制体系、事件体系以及 Layout Inspector 等工具都不再适用。

其他方面：

在研发效率上，旧页面迁移到 Flutter 的过程，效率是下降的。 以前沉淀的 UI 组件，需要在 Flutter 上重新实现。对于全新的页面需求，或者是已经迁移完成的页面上的新需求，可以明显看到 Flutter 跨端带来的效率提升。除了两端变一端的好处外，还有协同减少、一致性提升、Hot Reload 开发等好处。

此外，Flutter 线上仍然有一些 Crash 存在，虽然比例不高，但还是会带来一定的排查成本。定位这类问题需要对 Flutter 的原理有一定的理解。

2.Flutter 性能分析

在《Flutter 和原生应用性能对比》这篇文章中，从安装包大小、运行性能（CPU 资源占用、内存占用）、应用启动速度、FPS 等方面对 Flutter 和 Android 原生应用进行了对比。
链接：https://www.colabug.com/4279754.html

在《iOS 原生 VS Flutter 评测》中同样从以上几个方面对 Flutter 和 iOS 原生应用进行了对比。
链接：http://www.swifty.cc/2187.htm

在《Flutter 和 RN 对比》中对 Flutter 和 RN 进行了对比。
链接：https://www.jianshu.com/p/51c4f7f6e446

结论：

原生应用在内存、CPU 资源占用方面要低于 Flutter，并且安装包的体积也要小于 Flutter。所以，不考虑其他因素，单纯从性能角度来说，原生应用肯定是要优于 Flutter 的；

由于 Flutter 避免了 RN 的那种通过桥接器与 Javascript 通讯导致效率低下的问题,所以在性能方面比 RN 更高一筹。

3.Flutter ListView 简介

官方对于 ListView 的介绍：

What is the alternative to a ListView in Flutter?
The equivalent to a ListView in Flutter is ··· a ListView!
In an Android ListView, you create an adapter and pass it into the ListView, which renders each row with what your adapter returns. However, you have to make sure you recycle your rows, otherwise, you get all sorts of crazy visual glitches and memory issues.
Due to Flutter's immutable widget pattern, you pass a List of Widgets to your ListView, and Flutter takes care of making sure that scrolling is fast and smooth.

Flutter 用啥替代 ListView？
对的，用 ListView···
Android 里我们创建 adapter 控制如何渲染每个 child，同时我们还要负责复用 view，避免浪费大量的内存和附带的卡顿问题。

由于 Flutter 的 widget 是 immutable 的，你只负责生成 ListView ， Flutter 来确视图的顺畅平滑。

Ways for Initialization

There are some options for initialization and each one have their advantage.

Default Constructor

Take an explicit list of Widget. Useful when you have a small list of children, because it re-create each displayed item again, without recycle the view.

ListView.Builder

The view is created only for visible items, and each item is built on demand. Useful for large list of items.

4.从渲染机制和 View 更新角度探究 ListView item 重用

由于从官方对于 ListView 的介绍中，我们并没有发现对于 item 重用相关的介绍，所以我们从渲染机制和 View 更新角度来探究 Flutter ListView 中是否对 item 进行了重用。

Flutter 视图体系

Flutter 的思想十分类似 React，通过 Widget Tree 来表示整个 app 的视图。那么它的视图树是怎样的呢？
Flutter 视图树包含了三种树，分别是 **Widget**、**Element**、**RenderObject**。
下图介绍了三种树的基础 class 的对应关系和功能介绍：

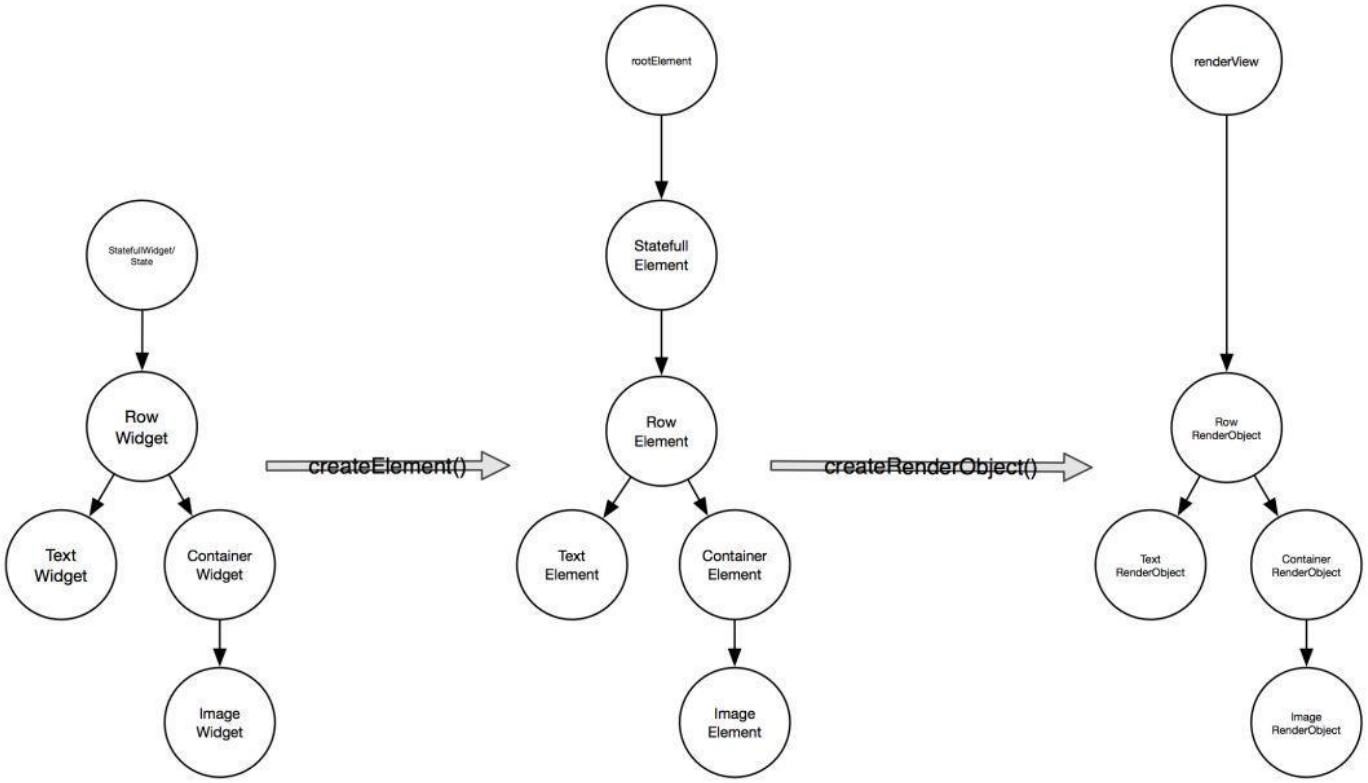
<table>
<tr><th>类型</th><th>Widget</th><th>Element</th><th>RenderObject</th><th>说明</th></tr>
<tr><td rowspan="2">组合型</td><td>StatelessWidget</td><td>StatelessElement</td><td>NA</td><td rowspan="2">Composer角色，将其他widget进行拼装组合成一个新的wigdet</td></tr>
<tr><td>StatefulWidget/State</td><td>StatefulElement</td><td>NA</td></tr>
<tr><td>代理型</td><td>ProxyWidget</td><td>ProxyElement</td><td>NA</td><td>数据传递</td></tr>
<tr><td rowspan="4">展示型</td><td>RenderObjectWidget</td><td>RenderObjectElement</td><td rowspan="4">RenderObject</td><td rowspan="4">具有实际展示内容的视图</td></tr>
<tr><td>SingleChildRenderObjectWidget</td><td>SingleChildRenderObjectElement</td></tr>
<tr><td>LeafRenderObjectWidget</td><td>LeafRenderObjectElement</td></tr>
<tr><td>MultiChildRenderObjectWidget</td><td>MultiChildRenderObjectElement</td></tr>
</table>

Widget：存放渲染内容、视图布局信息,widget 的属性最好都是 immutable(如何更新数据呢？查看后续内容)。
Element：存放上下文，通过 Element 遍历视图树，Element 同时持有 Widget 和 RenderObject。
RenderObject：根据 Widget 的布局属性进行 layout，paint Widget 传入的内容。

创建树的过程：

从调用 runApp(rootWidget)开始，将 rootWidget 传给 rootElement，作为 rootElement 的子节点，生成 Element 树，由 Element 树生成 Render 树。

大致过程如图所示：



注：

widget 重新创建，element 树和 renderObject 树是否也重新创建？

widget 只是一个配置数据结构，创建是非常轻量的，加上 Flutter 团队对 widget 的创建/销毁做了优化，不用担心整个 widget 树重新创建所带来的性能问题，但是 renderobject 就不一样了，renderobject 涉及到 layout、paint 等复杂操作，是一个真正渲染的 view，整个 view 树重新创建开销就比较大，所以答案是否定的。

树的更新过程：

（1）找到 widget 对应的 element 节点，设置 element 为 dirty，触发 drawframe，drawframe 会调用 element 的 performRebuild()进行树重建；
（2）widget.build() == null，deactive element.child,删除子树，流程结束；
（3）element.child.widget == NULL，mount 的新子树，流程结束；
（4）element.child.widget == widget.build() 无需重建，否则进入流程（5）；
（5）Widget.canUpdate(element.child.widget, newWidget) == true，更新 child 的 slot，element.child.update(newWidget)(如果 child 还有子节点，则递归上面的流程进行子树更新),流程结束，否则转（6）；
（6）Widget.canUpdate(element.child.widget, newWidget) ！= true（widget 的 classtype 或者 key 不相等），deactivew element.child，mount 新子树。

注：

1. element.child.widget == widget.build()，不会触发子树的 update，当触发 update 的时候，如果没有生效，要注意 widget 是否使用旧 widget，没有 new widget，导致 update 流程走到该 widget 就停止了。
2. 如何触发树更新

全局更新：调用 runApp(rootWidget)，一般 flutter 启动时调用后不再会调用。
局部子树更新，将该子树做 StatefullWidget 的一个子 widget，并创建对应的 State 类实例，通过调用 state.setState() 触发该子树的刷新。

3. StatefullWidget vs StatelessWidget

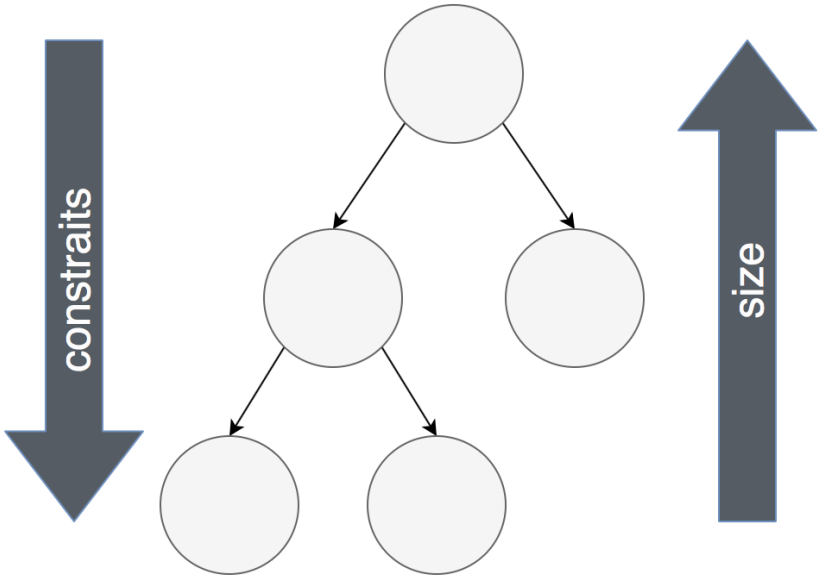
StatelessWidget：无中间状态变化的 widget，需要更新展示内容就得通过重新 new，Flutter 推荐尽量使用

StatelessWidget。
StatefullWidget：存在中间状态变化，那么问题来了，widget 不是都 immutable 的，状态变化存储在哪里？Flutter 引入 state 的类用于存放中间态，通过调用 state.setState()进行此节点及以下的整个子树更新。
4. 当 ListView 中的 item 滚动出可显示区域的时候，item 会被从树中 remove 掉，此 item 子树中所有的 state 都会被 dispose，state 记录的数据都会销毁，item 滚动回可显示区域时，会重新创建全新的 state。

Flutter 界面渲染过程

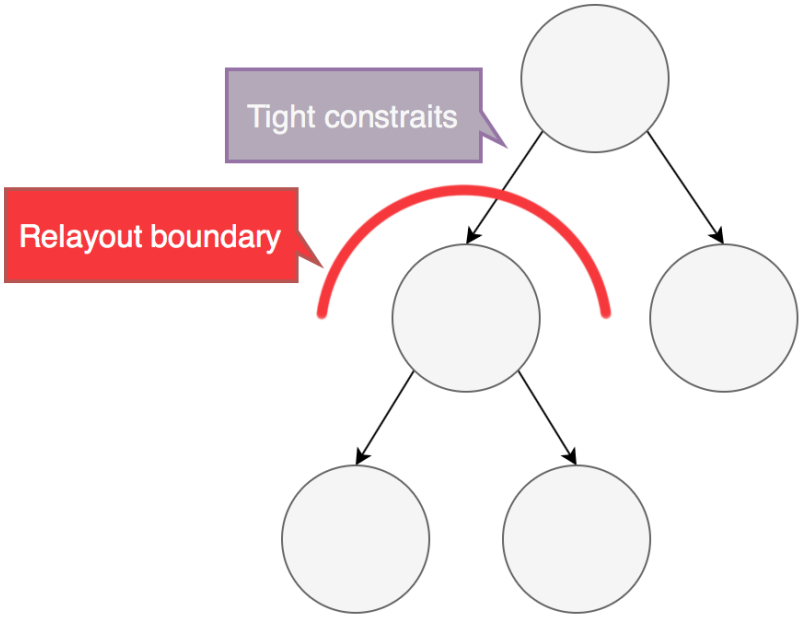
在 Flutter 界面渲染过程分为三个阶段：**布局、绘制、合成**，布局和绘制在 Flutter 框架中完成，合成则交由引擎负责。

控件树中的每个控件通过实现 RenderObjectWidget#createRenderObject(BuildContext context) → RenderObject 方法来创建对应的不同类型的 RenderObject 对象，组成渲染对象树。因为 Flutter 极大地简化了布局的逻辑，所以整个布局过程中只需要深度遍历一次：

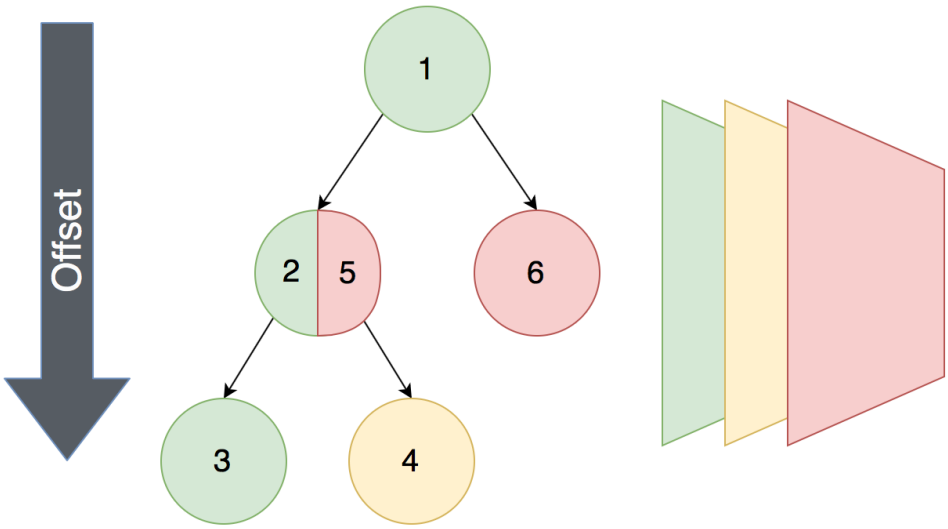


渲染对象树中的每个对象都会在布局过程中接受父对象的 Constraints 参数，决定自己的大小，然后父对象就可

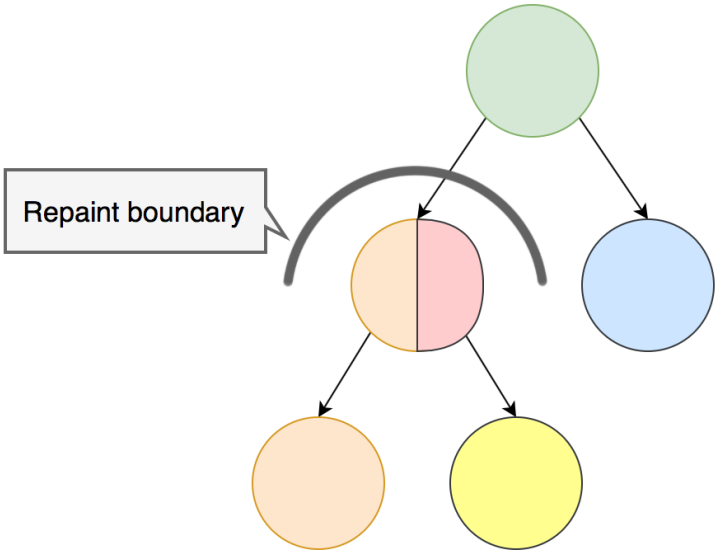
以按照自己的逻辑决定各个子对象的位置，完成布局过程。子对象不存储自己在容器中的位置，所以在它的位置发生改变时并不需要重新布局或者绘制。子对象的位置信息存储在它自己的 parentData 字段中，但是该字段由它的父对象负责维护，自身并不关心该字段的内容。同时也因为这种简单的布局逻辑，Flutter 可以在某些节点设置布局边界（Relayout boundary），即当边界内的任何对象发生重新布局时，不会影响边界外的对象，反之亦然：



布局完成后，渲染对象树中的每个节点都有了明确的尺寸和位置，Flutter 会把所有对象绘制到不同的图层上：



因为绘制节点时也是深度遍历，可以看到第二个节点在绘制它的背景和前景不得不绘制在不同的图层上，因为第四个节点切换了图层（因为“4”节点是一个需要独占一个图层的内容，比如视频），而第六个节点也一起绘制到了红色图层。这样会导致第二个节点的前景（也就是“5”）部分需要重绘时，和它在逻辑上毫不相干但是处于同一图层的第六个节点也必须重绘。为了避免这种情况，Flutter 提供了另外一个“重绘边界”的概念：

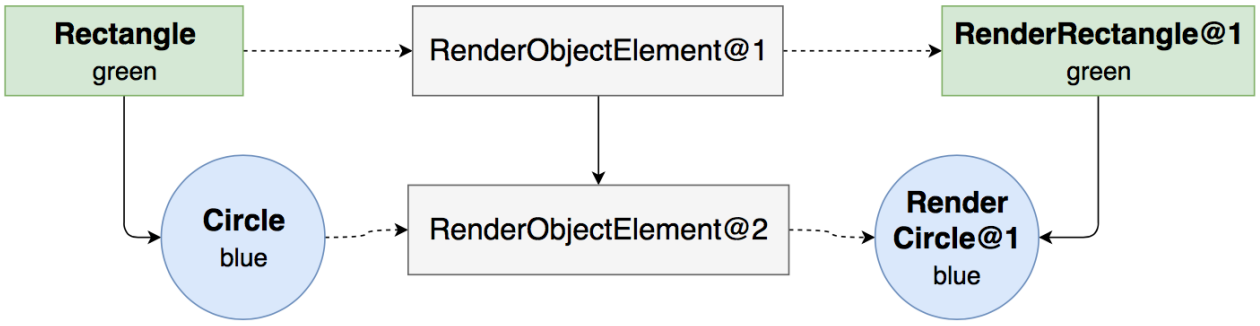


在进入和走出重绘边界时，Flutter 会强制切换新的图层，这样就可以避免边界内外的互相影响。典型的应用场景就是 ScrollView，当滚动内容重绘时，一般情况下其他内容是不需要重绘的。

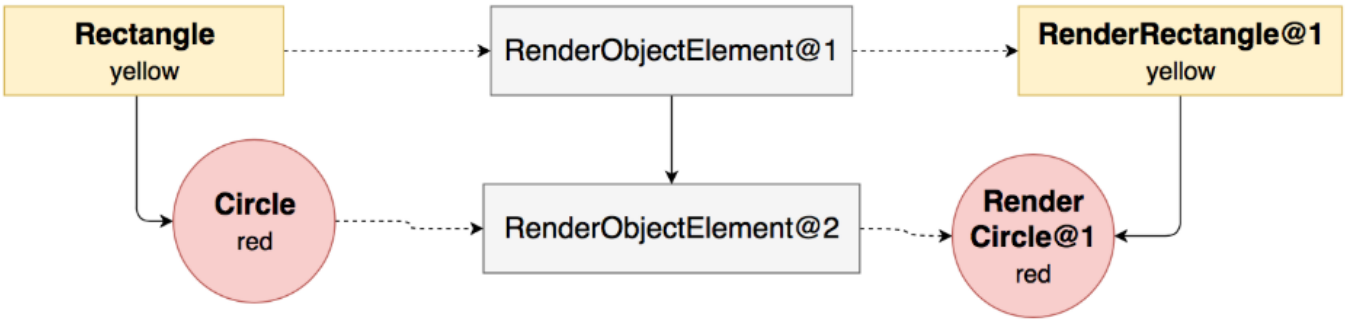
基于 Flutter 控件系统开发的程序都需要使用 WidgetsFlutterBinding，它是 Flutter 的控件框架和 Flutter 引擎的胶水层。Widget 就是所有控件的基类，它本身所有的属性都是只读的。RenderObjectWidget 所有的实现类则负责提供配置信息并创建具体的 RenderObjectElement。Element 是 Flutter 用来分离控件树和真正的渲染对象的中间层，控件用来描述对应的 element 属性，控件重建后可能会复用同一个 element。RenderObjectElement 持有真正负责布局、绘制和碰撞测试（hit test）的 RenderObject 对象。

StatelessWidget 和 StatefulWidget 并不会直接影响 RenderObject 的创建，它们只负责创建对应的 RenderObjectWidget，StatelessElement 和 StatefulElement 也是类似的功能。

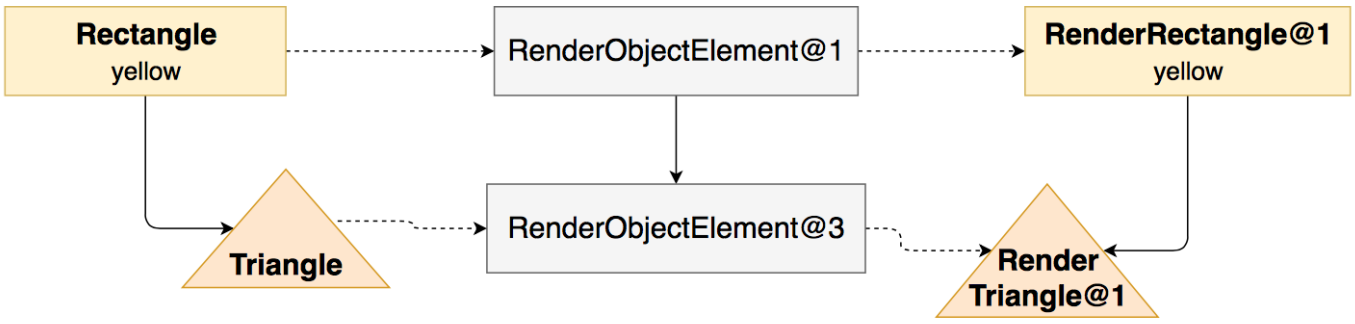
它们之间的关系如下图：



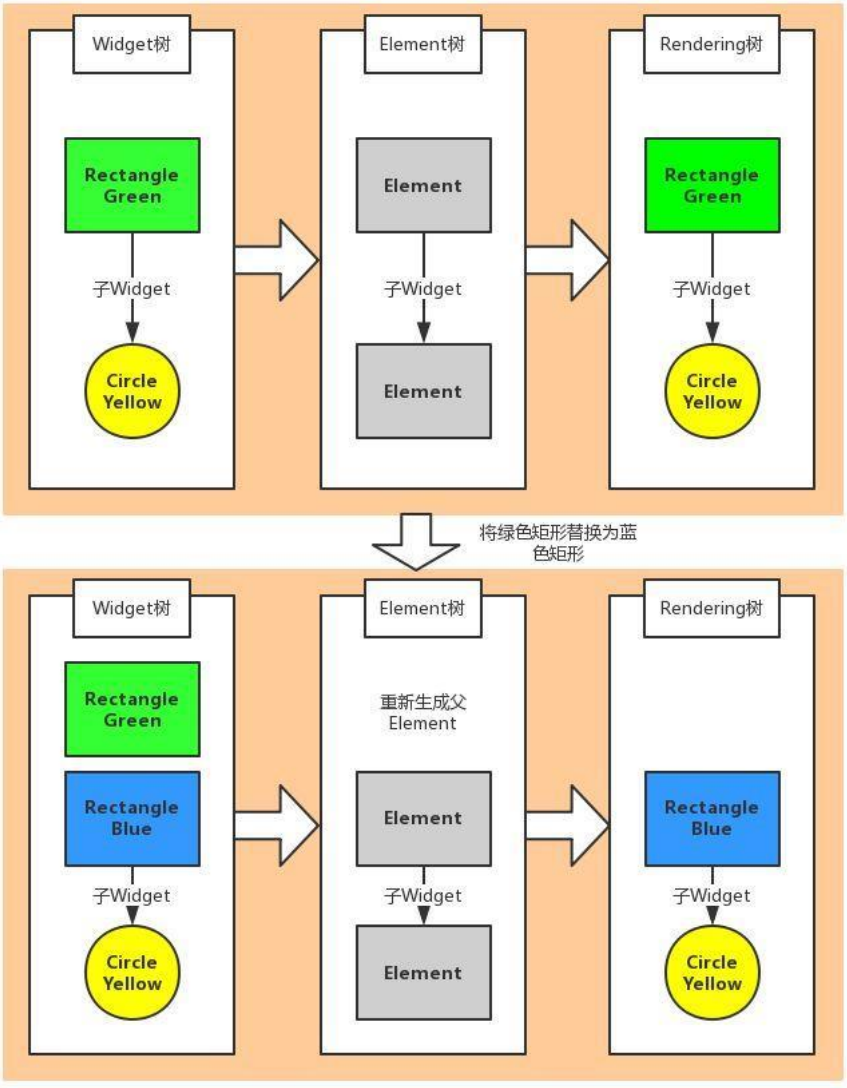
如果控件的属性发生了变化（因为控件的属性是只读的，所以变化也就意味着重新创建了新的控件树），但是其树上每个节点的类型没有变化时，element 树和 render 树可以完全重用原来的对象（因为 element 和 render object 的属性都是可变的）：



但是，如果控件树种某个节点的类型发生了变化，则 element 树和 render 树中的对应节点也需要重新创建：



更直观的过程如下图所示：



可以看到，矩形的子 Widget 的类型并没有改变，所以在 Element 树上也没有改变，到了 Rendering 树也没有重新渲染，这种设计理念对于刷新 UI 操作可以大大提高效率。

注：

一些内部优化：

1. Dart 语言为了更好的适应 FlutterUI 框架，在内存分配和垃圾回收做了很多优化。因为 Dart 在连续分配多个对象的时候，所需消耗的资源非常少。Dart 虚拟机可以快速分配内存给短期生存的对象。
2. Repaint Boundary

类似 Relayout boundary,Paint 阶段也有 Repaint Boundary，目的和 layout 一样，就是对应子树的 paint 不会导致外

部的 repaint，但是 Relayout boundary 需要开发人员自己设置，使用 RepaintBoundary widget 进行设置，ListView 在渲染的 item 默认都是使用了 RepaintBoundary，显而易见 ListView 的 children 之间都是相互独立的。Flutter 建议复杂的 image 渲染使用 RepaintBoundary，image 的渲染需要 io 操作，然后解码，最后渲染，使用 RepaintBoundary 可以进行 gpu 的缓存，但是不一定就会缓存，engine 会判断这个 image 是否足够复杂，毕竟 gpu 缓存还是非常珍贵的，同时 RepaintBoundary 还会对一些反复渲染的 layer 进行缓存处理（反复渲染 3 次及以上，这个是 Flutter 的视频中提到的）。

总结：

1. 从 View 的更新机制角度来说，由于 Widget 都是 immutable 的，如果想要修改 Widget 的某一属性就必须要重新创建一遍 Widget，所以我们没有办法对 widget 进行重用；
2. 从渲染机制的角度来说，Flutter 中有布局边界（Relayout boundary）和重绘边界（Repaint Boundary）的概念，使得像 ScrollView 和 ListView 这样的控件，当滚动内容重绘时，一般情况下其他内容是不需要重绘的。也正如官方对于 ListView 的介绍（The view is created only for visible items, and each item is built on demand. ）；
3. Flutter 中存在重用的概念，只不过重用的不是 widget，而是 element 树和 render 树中对象的重用（这二者重新创建的开销相对来说是比较大的）；
4. Dart 语言为了更好的适应 FlutterUI 框架，在内存分配和垃圾回收做了很多优化。因为 Dart 在连续分配多个对象的时候，所需消耗的资源非常少。Dart 虚拟机可以快速分配内存给短期生存的对象。而且 widget 只是一个配置数据结构，创建是非常轻量的；
5. 看了很多的开源项目，里面对于 ListView 的使用都只是用官方推荐的方法(ListView.Builder)，并没有特别地去实现复用，也正如官方介绍（You pass a List of Widgets to your ListView, and Flutter takes care of making sure that scrolling is fast and smooth.）那样；
6. 基于以上的研究仍然不太“死心”，于是做了一个实验，暂时不考虑数据更新的问题，我们让生成的 ListView 始终“复用”第一次创建的 widget，最终结果是滑动性能并没有可感知的提升。

5.结论

Flutter 虽然自身有很多优势，但相对于其他的跨平台方案来说仍然处于一个比较早期的阶段，暂时还未发布正式的 Release 版本。在性能（尤其是滑动）方面的提升，作为开发者来说能够做的还很有限。但是经过几个 Demo 的开发发现，Flutter 还是比较容易上手的（在文末提供了 Android 原生项目集成 Flutter 以及相互调用的 Demo）。

6.其他

（1）参考链接：

https://juejin.im/post/5ad305926fb9a028d6649cb7
https://www.jianshu.com/p/fe604938df0a
https://www.jianshu.com/p/e6cd8584fdbb
https://www.jianshu.com/p/265ede3359f9

（2）混合开发实践：

https://blog.csdn.net/u010960265/article/details/81533732
https://blog.csdn.net/Kinsomy/article/details/82878711
https://www.jianshu.com/p/d9b1290e9e28

Android 原生项目集成 Flutter 以及相互调用的 Demo：