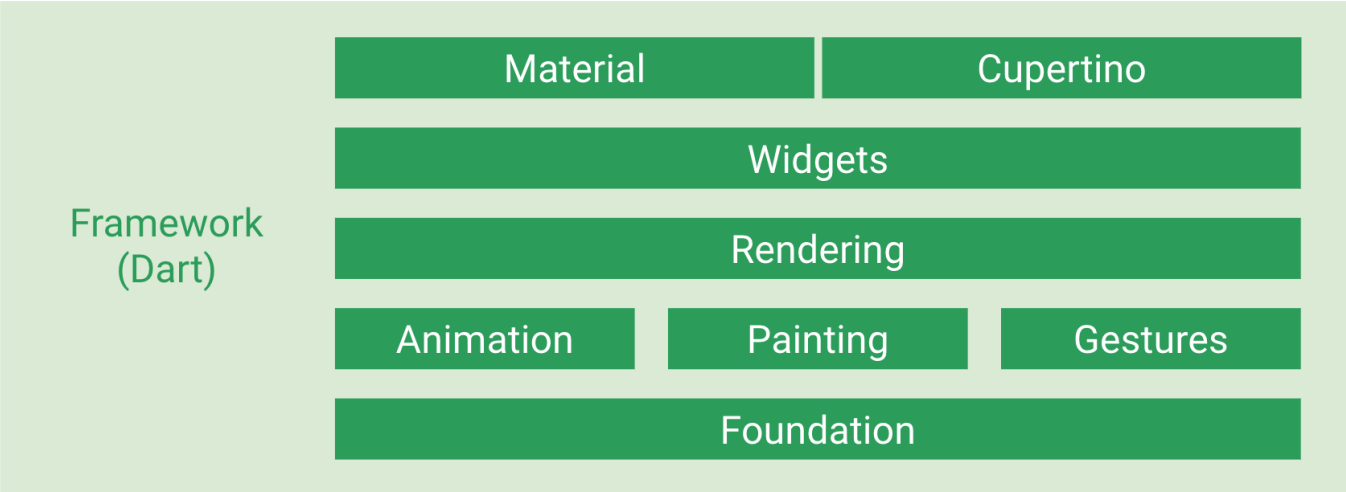
Flutter 原理和工程实践

简介

Flutter 的架构和原理

Flutter framework 层的架构图如下：



Foundation: foundation 提供了 framework 经常使用的一些基础类，包括但不限于：

- BindBase: 提供了提供单例服务的对象基类，提供了 Widgets、Render、Gestures等能力
- Key: 提供了 Flutter 常用的 Key 的基类
- AbstractNode：表示了控件树的节点

在 foundation 之上，Flutter 提供了 动画、绘图、手势、渲染和部件，其中部件就包括我们比较熟悉的 Material 和 Cupertino 风格

我们从 dart 的入口处关注 Flutter 的渲染原理

```
void runApp(Widget app) {
  WidgetsFlutterBinding.ensureInitialized()
    ..attachRootWidget(app)
    ..scheduleWarmUpFrame();
}
```

我们直接使用了 Widgets 层的能力

widgets

负责根据我们 dart 代码提供的 Widget 树，来构造实际的虚拟节点树

在 FLutter 的渲染机制中，有 3 个比较关键的概念：

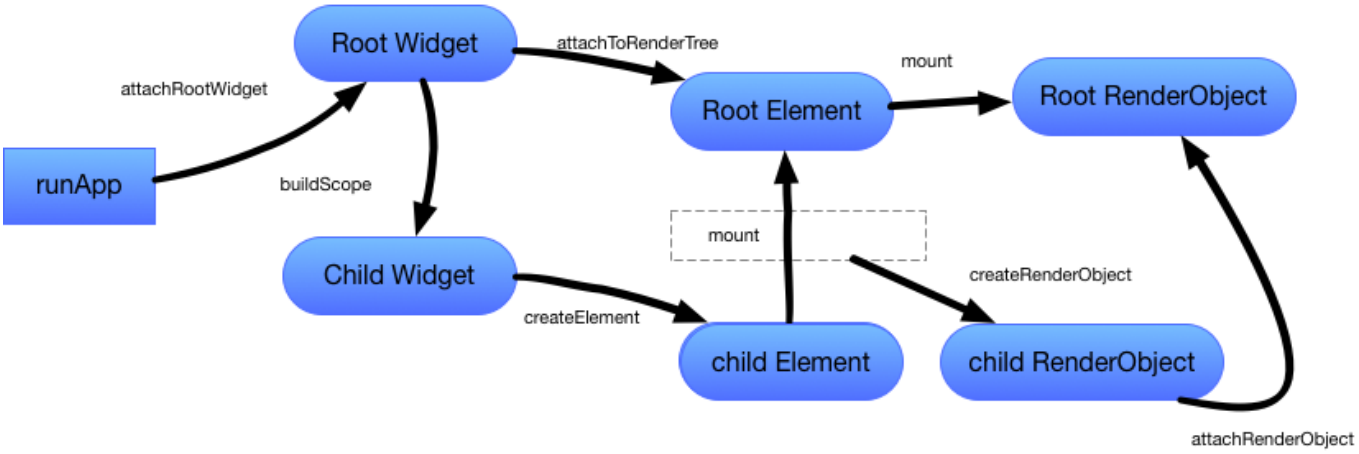
- Widget： 我们在 dart 中直接编写的 Widget，表示控件
- Element：实际构建的虚拟节点，所有的节点构造出实际的控件树，概念是类似前端经常提到的 vitrual dom
- RenderObject： 实际负责控件的视图工作。包括布局、渲染和图层合成

根据 `attachRootWidget` 的流程，我们可以了解到布局树的构造流程

1. `attachRootWidget` 创建根节点
2. `attachToRenderTree` 创建 root Element
3. Element 使用 `mount` 方法把自己挂载到父 Element。这里因为自己是根节点，所以可以忽略挂载过程
4. `mount` 会通过 `createRenderObject` 创建 root Element 的 RenderObject

到这里，整颗 tree 的 root 节点就构造出来了，在 `mount` 中，会通过 `BuildOwner#buildScope` 执行子节点的创建和挂载, 这里需要注意的是 child 的 RenderObject 也会被 attach 到 parent 的 RenderObejct 上去

整个过程我们可以通过下图表示



感兴趣可以参考 `Element`、`RenderObjectElement`、`RenderObject` 的源码

渲染

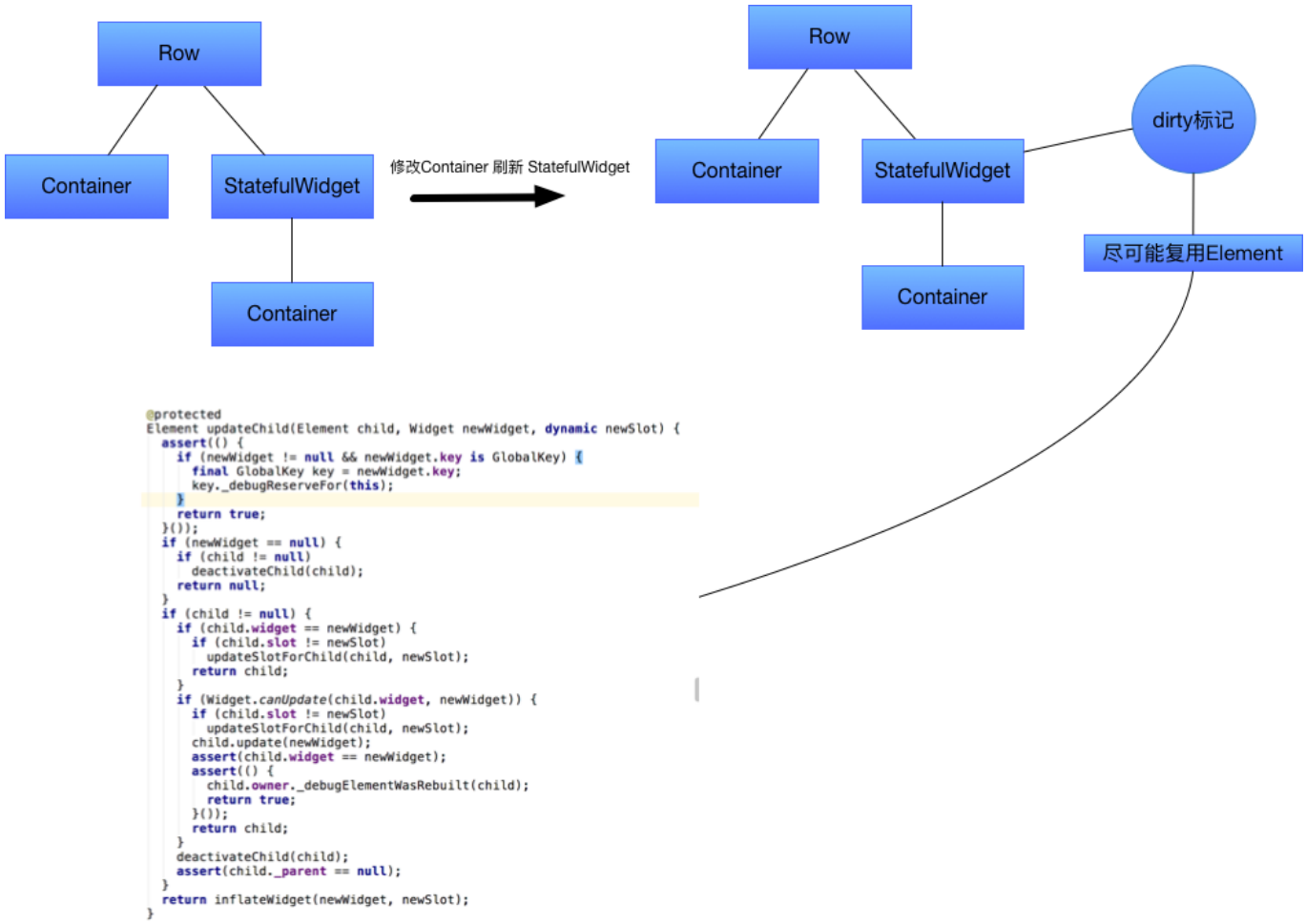
负责实际整个控件树 RenderObject 的布局和绘制

runApp 后会执行 `scheduleWarmUpFrame` 方法，这里就会开始调度渲染任务，进行每一帧的渲染

从 `handleBeginFrame` 和 `handleDrawFrame` 会走到 binding 的 `drawFrame` 函数，依次会调用 `WidgetsBinding` 和 `RendererBinding` 的 `drawFrame`。

这里会通过 Element 的 `BuildOwner`，去重新塑造我们的控件树。

大致原理如图



在构造或者刷新一颗控件树的时候，我们会把有改动部分的 Widget 标记为 dirty，并针对这部分执行 rebuild，但是 Flutter 会有判断来保证尽量复用 Element，从而避免了反复创建 Element 对象带来的性能问题。

在对 dirty elements 进行处理的时候，会对它进行一次排序，排序规则参考了 element 的深度:

```
static int _sort(Element a, Element b) {
    if (a.depth < b.depth)
      return -1;
    if (b.depth < a.depth)
      return 1;
    if (b.dirty && !a.dirty)
      return -1;
    if (a.dirty && !b.dirty)
      return 1;
    return 0;
  }
```

根据 depth 排序的目的，则是为了保证子控件一定排在父控件的左侧, 这样在 build 的时候，可以避免对子 widget 进行重复的 build。

在实际渲染过程中，Flutter 会利用 **Relayout Boundary**机制

```
void markNeedsLayout() {
    // ...
```

```
      if (_relayoutBoundary != this) {
        markParentNeedsLayout();
      } else {
        _needsLayout = true;
        if (owner != null) {
          owner._nodesNeedingLayout.add(this);
          owner.requestVisualUpdate();
        }
      }
      //...
    }
```

在设置了 relayout boundary 的控件中，只有子控件会被标记为 needsLayout，可以保证，刷新子控件的状态后，控件树的处理范围都在子树，不会去重新创建父控件，完全隔离开。

在每一个 RendererBinding 中，存在一个 `PipelineOwner` 对象，类似 WidgetsBinding 中的 `BuildOwner`. `BuilderOwner` 负责控件的build 流程，`PipelineOwner` 负责 render tree 的渲染。

```
@protected
  void drawFrame() {
    assert(renderView != null);
    pipelineOwner.flushLayout();
    pipelineOwner.flushCompositingBits();
    pipelineOwner.flushPaint();
    renderView.compositeFrame(); // this sends the bits to the GPU
    pipelineOwner.flushSemantics(); // this also sends the semantics to
the OS.
  }
```

RenderBinding 的 `drawFrame` 实际阐明了 render obejct 的渲染流程。即 布局(layout)、绘制(paint)、合成(compositeFrame)

调度(scheduler和线程模型)

在布局和渲染中，我们会观察到 Flutter 拥有一个 `SchedulerBinding`,在 frame 变化的时候，提供 callback 进行处理。不仅提供了帧变化的调度，在 `SchedulerBinding` 中，也提供了 task 的调度函数。这里我们就需要了解一下 dart 的异步任务和线程模型。

dart 的单线程模型，所以在 dart 中，没有所谓的主线程和子线程说法。dart 的异步操作采取了 event-looper 模型。

dart 没有线程的概念，但是有一个概念，叫做 isolate, 每个 isolate 是互相隔离的，不会进行内存的共享。在 main isolate 的 main 函数结束之后，会开始一个个处理 event queue 中的 event。也就是，dart 是先执行完同步代码后，再进行异步代码的执行。所以如果存在非常耗时的任务，我们可以创建自己的 isolate 去执行。

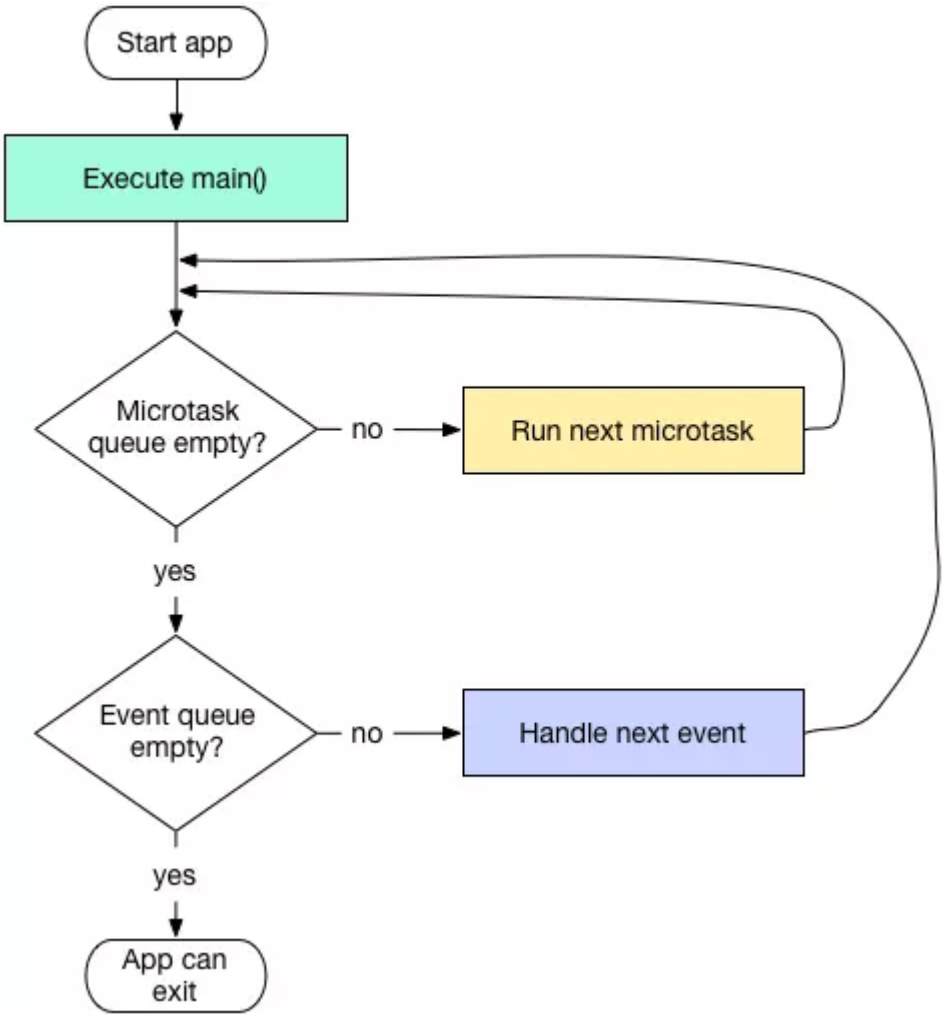
每一个 isolate 中，存在 2 个 event queue

- Event Queue
- Microtask Queue

event-looper 执行任务的顺序是

1. 优先执行 Microtask Queue 中的task
2. Microtask Queue 为空后，才会执行 Event Queue 中的事件

flutter 的异步模型如下图



Gesture

每一个 GUI 都离不开手势/指针的相关事件处理。

在 GestureBiding 中，在 `_handlePointerEvent` 函数中，`PointerDownEvent` 事件每处理一次，就会创建一个 `HintTest` 对象。在 `HintTest` 中，会存有每次经过的控件节点的 path。

最终我们也会看到一个 `dispatchEvent` 函数，进行事件的分发以及 `handleEvent`，对事件进行处理。

在根节点的 renderview 中，事件会开始从 `hitTest` 处理，因为我们添加了事件的传递路径，所以，时间在经过每个节点的时候，都会被”处理“。

```
@override // from HitTestDispatcher
  void dispatchEvent(PointerEvent event, HitTestResult hitTestResult) {
    if (hitTestResult == null) {
      assert(event is PointerHoverEvent || event is PointerAddedEvent ||
event is PointerRemovedEvent);
      try {
```

```
          pointerRouter.route(event);
        } catch (exception, stack) {
        }
        return;
      }
      for (HitTestEntry entry in hitTestResult.path) {
        try {
          entry.target.handleEvent(event, entry);
        } catch (exception, stack) {
        }
      }
    }
```

这里我们就可以看出来 Flutter 的时间顺序，从根节点开始分发，一直到子节点。同理，时间处理完后，会沿着子节点传到父节点，最终回到 `GestureBinding`。 这个顺序其实和 Android 的 View 事件分发 和 浏览器的事件冒泡 是一样的。

通过 `GestureDector` 这个 Widget, 我们可以触发和处理各种这样的事件和手势。具体的可以参考 Flutter 文档。

Material、Cupertino

Flutter 在 Widgets 之上，实现了兼容 Andorid/iOS 风格的设计。让APP 在 ui/ue 上有类原生的体验。

Flutter 的工程实践