# 14.4 Flutter运行机制-从启动到显示

本节我们主要介绍一下Flutter从启动到显示的过程。

### 启动

Flutter的入口在“lib/main.dart”的main()函数中，它是Dart应用程序的起点。在Flutter应用中，main()函数最简单的实现如下：

void main() {  
 runApp(MyApp());  
}

可以看main()函数只调用了一个runApp()方法，我们看看runApp()方法中都做了什么：

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

参数app是一个widget，它是Flutter应用启动后要展示的第一个Widget。而WidgetsFlutterBinding正是绑定widget 框架和Flutter engine的桥梁，定义如下：

class WidgetsFlutterBinding extends BindingBase with GestureBinding, ServicesBinding, SchedulerBinding, PaintingBinding, SemanticsBinding, RendererBinding, WidgetsBinding {  
 static WidgetsBinding ensureInitialized() {  
 if (WidgetsBinding.instance == null)  
 WidgetsFlutterBinding();  
 return WidgetsBinding.instance;  
 }  
}

可以看到WidgetsFlutterBinding继承自BindingBase 并混入了很多Binding，在介绍这些Binding之前我们先介绍一下Window，下面是Window的官方解释：

The most basic interface to the host operating system’s user interface.

很明显，Window正是Flutter Framework连接宿主操作系统的接口。我们看一下Window类的部分定义：

class Window {  
   
 // 当前设备的DPI，即一个逻辑像素显示多少物理像素，数字越大，显示效果就越精细保真。  
 // DPI是设备屏幕的固件属性，如Nexus 6的屏幕DPI为3.5   
 double get devicePixelRatio => \_devicePixelRatio;  
   
 // Flutter UI绘制区域的大小  
 Size get physicalSize => \_physicalSize;  
  
 // 当前系统默认的语言Locale  
 Locale get locale;  
   
 // 当前系统字体缩放比例。   
 double get textScaleFactor => \_textScaleFactor;   
   
 // 当绘制区域大小改变回调  
 VoidCallback get onMetricsChanged => \_onMetricsChanged;   
 // Locale发生变化回调  
 VoidCallback get onLocaleChanged => \_onLocaleChanged;  
 // 系统字体缩放变化回调  
 VoidCallback get onTextScaleFactorChanged => \_onTextScaleFactorChanged;  
 // 绘制前回调，一般会受显示器的垂直同步信号VSync驱动，当屏幕刷新时就会被调用  
 FrameCallback get onBeginFrame => \_onBeginFrame;  
 // 绘制回调   
 VoidCallback get onDrawFrame => \_onDrawFrame;  
 // 点击或指针事件回调  
 PointerDataPacketCallback get onPointerDataPacket => \_onPointerDataPacket;  
 // 调度Frame，该方法执行后，onBeginFrame和onDrawFrame将紧接着会在合适时机被调用，  
 // 此方法会直接调用Flutter engine的Window\_scheduleFrame方法  
 void scheduleFrame() native 'Window\_scheduleFrame';  
 // 更新应用在GPU上的渲染,此方法会直接调用Flutter engine的Window\_render方法  
 void render(Scene scene) native 'Window\_render';  
  
 // 发送平台消息  
 void sendPlatformMessage(String name,  
 ByteData data,  
 PlatformMessageResponseCallback callback) ;  
 // 平台通道消息处理回调   
 PlatformMessageCallback get onPlatformMessage => \_onPlatformMessage;  
   
 ... //其它属性及回调  
   
}

可以看到Window类包含了当前设备和系统的一些信息以及Flutter Engine的一些回调。现在我们再回来看看WidgetsFlutterBinding混入的各种Binding。通过查看这些 Binding的源码，我们可以发现这些Binding中基本都是监听并处理Window对象的一些事件，然后将这些事件按照Framework的模型包装、抽象然后分发。可以看到WidgetsFlutterBinding正是粘连Flutter engine与上层Framework的“胶水”。

* GestureBinding：提供了window.onPointerDataPacket 回调，绑定Framework手势子系统，是Framework事件模型与底层事件的绑定入口。
* ServicesBinding：提供了window.onPlatformMessage 回调， 用于绑定平台消息通道（message channel），主要处理原生和Flutter通信。
* SchedulerBinding：提供了window.onBeginFrame和window.onDrawFrame回调，监听刷新事件，绑定Framework绘制调度子系统。
* PaintingBinding：绑定绘制库，主要用于处理图片缓存。
* SemanticsBinding：语义化层与Flutter engine的桥梁，主要是辅助功能的底层支持。
* RendererBinding: 提供了window.onMetricsChanged 、window.onTextScaleFactorChanged 等回调。它是渲染树与Flutter engine的桥梁。
* WidgetsBinding：提供了window.onLocaleChanged、onBuildScheduled 等回调。它是Flutter widget层与engine的桥梁。

WidgetsFlutterBinding.ensureInitialized()负责初始化一个WidgetsBinding的全局单例，紧接着会调用WidgetsBinding的attachRootWidget方法，该方法负责将根Widget添加到RenderView上，代码如下：

void attachRootWidget(Widget rootWidget) {  
 \_renderViewElement = RenderObjectToWidgetAdapter<RenderBox>(  
 container: renderView,   
 debugShortDescription: '[root]',  
 child: rootWidget  
 ).attachToRenderTree(buildOwner, renderViewElement);  
}

注意，代码中的有renderView和renderViewElement两个变量，renderView是一个RenderObject，它是渲染树的根，而renderViewElement是renderView对应的Element对象，可见该方法主要完成了根widget到根 RenderObject再到根Element的整个关联过程。我们看看attachToRenderTree的源码实现：

RenderObjectToWidgetElement<T> attachToRenderTree(BuildOwner owner, [RenderObjectToWidgetElement<T> element]) {  
 if (element == null) {  
 owner.lockState(() {  
 element = createElement();  
 assert(element != null);  
 element.assignOwner(owner);  
 });  
 owner.buildScope(element, () {  
 element.mount(null, null);  
 });  
 } else {  
 element.\_newWidget = this;  
 element.markNeedsBuild();  
 }  
 return element;  
}

该方法负责创建根element，即RenderObjectToWidgetElement，并且将element与widget 进行关联，即创建出 widget树对应的element树。如果element 已经创建过了，则将根element 中关联的widget 设为新的，由此可以看出element 只会创建一次，后面会进行复用。那么BuildOwner是什么呢？其实他就是widget framework的管理类，它跟踪哪些widget需要重新构建。

### 渲染

回到runApp的实现中，当调用完attachRootWidget后，最后一行会调用 WidgetsFlutterBinding 实例的 scheduleWarmUpFrame() 方法，该方法的实现在SchedulerBinding 中，它被调用后会立即进行一次绘制（而不是等待“vsync” 信号），在此次绘制结束前，该方法会锁定事件分发，也就是说在本次绘制结束完成之前Flutter将不会响应各种事件，这可以保证在绘制过程中不会再触发新的重绘。下面是scheduleWarmUpFrame() 方法的部分实现(省略了无关代码)：

void scheduleWarmUpFrame() {  
 ...  
 Timer.run(() {  
 handleBeginFrame(null);   
 });  
 Timer.run(() {  
 handleDrawFrame();   
 resetEpoch();  
 });  
 // 锁定事件  
 lockEvents(() async {  
 await endOfFrame;  
 Timeline.finishSync();  
 });  
 ...  
}

可以看到该方法中主要调用了handleBeginFrame() 和 handleDrawFrame() 两个方法，在看这两个方法之前我们首先了解一下Frame 和 FrameCallback 的概念：

* Frame: 一次绘制过程，我们称其为一帧。Flutter engine受显示器垂直同步信号“VSync”的驱使不断的触发绘制。我们之前说的Flutter可以实现60fps（Frame Per-Second），就是指一秒钟可以触发60次重绘，FPS值越大，界面就越流畅。
* FrameCallback：SchedulerBinding 类中有三个FrameCallback回调队列， 在一次绘制过程中，这三个回调队列会放在不同时机被执行：
  1. transientCallbacks：用于存放一些临时回调，一般存放动画回调。可以通过SchedulerBinding.instance.scheduleFrameCallback 添加回调。
  2. persistentCallbacks：用于存放一些持久的回调，不能在此类回调中再请求新的绘制帧，持久回调一经注册则不能移除。SchedulerBinding.instance.addPersitentFrameCallback()，这个回调中处理了布局与绘制工作。
  3. postFrameCallbacks：在Frame结束时只会被调用一次，调用后会被系统移除，可由 SchedulerBinding.instance.addPostFrameCallback() 注册，注意，不要在此类回调中再触发新的Frame，这可以会导致循环刷新。

现在请读者自行查看handleBeginFrame() 和 handleDrawFrame() 两个方法的源码，可以发现前者主要是执行了transientCallbacks队列，而后者执行了 persistentCallbacks 和 postFrameCallbacks 队列。

### 绘制

渲染和绘制逻辑在RendererBinding中实现，查看其源码，发现在其initInstances()方法中有如下代码：

void initInstances() {  
 ... //省略无关代码  
   
 //监听Window对象的事件   
 ui.window  
 ..onMetricsChanged = handleMetricsChanged  
 ..onTextScaleFactorChanged = handleTextScaleFactorChanged  
 ..onSemanticsEnabledChanged = \_handleSemanticsEnabledChanged  
 ..onSemanticsAction = \_handleSemanticsAction;  
   
 //添加PersistentFrameCallback   
 addPersistentFrameCallback(\_handlePersistentFrameCallback);  
}

我们看最后一行，通过addPersistentFrameCallback 向persistentCallbacks队列添加了一个回调 \_handlePersistentFrameCallback:

void \_handlePersistentFrameCallback(Duration timeStamp) {  
 drawFrame();  
}

该方法直接调用了RendererBinding的drawFrame()方法：

void drawFrame() {  
 assert(renderView != null);  
 pipelineOwner.flushLayout(); //布局  
 pipelineOwner.flushCompositingBits(); //重绘之前的预处理操作，检查RenderObject是否需要重绘  
 pipelineOwner.flushPaint(); // 重绘  
 renderView.compositeFrame(); // 将需要绘制的比特数据发给GPU  
 pipelineOwner.flushSemantics(); // this also sends the semantics to the OS.  
}

我们看看这些方法分别做了什么：

#### flushLayout()

void flushLayout() {  
 ...  
 while (\_nodesNeedingLayout.isNotEmpty) {  
 final List<RenderObject> dirtyNodes = \_nodesNeedingLayout;  
 \_nodesNeedingLayout = <RenderObject>[];  
 for (RenderObject node in   
 dirtyNodes..sort((RenderObject a, RenderObject b) => a.depth - b.depth)) {  
 if (node.\_needsLayout && node.owner == this)  
 node.\_layoutWithoutResize();  
 }  
 }  
 }   
}

源码很简单，该方法主要任务是更新了所有被标记为“dirty”的RenderObject的布局信息。主要的动作发生在node.\_layoutWithoutResize()方法中，该方法中会调用performLayout()进行重新布局。

#### flushCompositingBits()

void flushCompositingBits() {  
 \_nodesNeedingCompositingBitsUpdate.sort(  
 (RenderObject a, RenderObject b) => a.depth - b.depth  
 );  
 for (RenderObject node in \_nodesNeedingCompositingBitsUpdate) {  
 if (node.\_needsCompositingBitsUpdate && node.owner == this)  
 node.\_updateCompositingBits(); //更新RenderObject.needsCompositing属性值  
 }  
 \_nodesNeedingCompositingBitsUpdate.clear();  
}

检查RenderObject是否需要重绘，然后更新RenderObject.needsCompositing属性，如果该属性值被标记为true则需要重绘。

#### flushPaint()

void flushPaint() {  
 ...  
 try {  
 final List<RenderObject> dirtyNodes = \_nodesNeedingPaint;   
 \_nodesNeedingPaint = <RenderObject>[];  
 // 反向遍历需要重绘的RenderObject  
 for (RenderObject node in   
 dirtyNodes..sort((RenderObject a, RenderObject b) => b.depth - a.depth)) {  
 if (node.\_needsPaint && node.owner == this) {  
 if (node.\_layer.attached) {  
 // 真正的绘制逻辑   
 PaintingContext.repaintCompositedChild(node);  
 } else {  
 node.\_skippedPaintingOnLayer();  
 }  
 }  
 }  
 }   
}

该方法进行了最终的绘制，可以看出它不是重绘了所有 RenderObject，而是只重绘了需要重绘的 RenderObject。真正的绘制是通过PaintingContext.repaintCompositedChild()来绘制的，该方法最终会调用Flutter engine提供的Canvas API来完成绘制。

#### compositeFrame()

void compositeFrame() {  
 ...  
 try {  
 final ui.SceneBuilder builder = ui.SceneBuilder();  
 final ui.Scene scene = layer.buildScene(builder);  
 if (automaticSystemUiAdjustment)  
 \_updateSystemChrome();  
 ui.window.render(scene); //调用Flutter engine的渲染API  
 scene.dispose();   
 } finally {  
 Timeline.finishSync();  
 }  
}

这个方法中有一个Scene对象，Scene对象是一个数据结构，保存最终渲染后的像素信息。这个方法将Canvas画好的Scene传给window.render()方法，该方法会直接将scene信息发送给Flutter engine，最终由engine将图像画在设备屏幕上。

#### 最后

需要注意的是：由于RendererBinding只是一个mixin，而with它的是WidgetsBinding，所以我们需要看看WidgetsBinding中是否重写该方法，查看WidgetsBinding的drawFrame()方法源码：

@override  
void drawFrame() {  
 ...//省略无关代码  
 try {  
 if (renderViewElement != null)  
 buildOwner.buildScope(renderViewElement);   
 super.drawFrame(); //调用RendererBinding的drawFrame()方法  
 buildOwner.finalizeTree();  
 }   
}

我们发现在调用RendererBinding.drawFrame()方法前会调用 buildOwner.buildScope() （非首次绘制），该方法会将被标记为“dirty” 的 element 进行 rebuild() 。

### 总结

本节介绍了Flutter APP从启动到显示到屏幕上的主流程，读者可以结合前面章节对Widget、Element以及RenderObject的介绍来加强细节理解。