链滴

# Flutter 怎么构建画面

作者：yc654084303
原文链接：https://ld246.com/article/1571734478761
来源网站：链滴

## Widget 是什么?

这里有一个 “总所周知” 的答就是：**Widget并不真正的渲染对象** 。是的，事实上在 Flutter 中渲染是历了从 Widget 到 Element 再到 RenderObject 的过程。

## Widget 是不可变的，那么 Widget 是如何在不可变中去构建画面的?

上面我们知道，Widget 是需要转化为 Element 去渲染的，而从下图注释可以看到，事实上 **Widget 只是 Element 的一个配置描述** ，告诉 Element 这个实例如何去渲染。

```
/// Describes the configuration for an [Element].
///描述[元素]的配置。
/// Widgets are the central class hierarchy in the Flutter framework. A widget
/// is an immutable description of part of a user interface. Widgets can be
/// inflated into elements, which manage the underlying render tree.
///widgets是flutr框架中的中心类层次结构。小部件
///是用户界面的一部分的不可变描述。小部件可以是
///已膨胀为元素，用于管理底层呈现树。
/// Widgets themselves have no mutable state (all their fields must be final).
/// If you wish to associate mutable state with a widget, consider using a
/// [StatefulWidget], which creates a [State] object (via
/// [StatefulWidget.createState]) whenever it is inflated into an element and
/// incorporated into the tree.
///小部件本身没有可变状态（它们的所有字段都必须是final）。
///如果希望将可变状态与小部件关联，请考虑使用
/// [statefulwidget]，它创建一个[状态]对象（通过
/// [statefulWidget.createState]）当它被膨胀成一个元素时
///合并到树中
/// A given widget can be included in the tree zero or more times. In particular
/// a given widget can be placed in the tree multiple times. Each time a widget
/// is placed in the tree, it is inflated into an [Element], which means a
/// widget that is incorporated into the tree multiple times will be inflated
/// multiple times.
///给定的小部件可以包含在树中零次或多次。特别地
///给定的小部件可以多次放置在树中。每次一个小部件
///放在树中，它被膨胀成一个[element]，这意味着
///多次合并到树中的小部件将被膨胀
///多次。
/// The [key] property controls how one widget replaces another widget in the
/// tree. If the [runtimeType] and [key] properties of the two widgets are
/// [operator==], respectively, then the new widget replaces the old widget by
/// updating the underlying element (i.e., by calling [Element.update] with the
/// new widget). Otherwise, the old element is removed from the tree, the new
/// widget is inflated into an element, and the new element is inserted into the
/// tree.
///[key]属性控制一个小部件如何替换
///树。如果两个小部件的[RuntimeType]和[Key]属性是
///[operator==]，然后新的小部件替换旧的小部件
///更新底层元素（即，使用
///new widget）。否则，将从树中删除旧元素
///widget被扩展成一个元素，新元素被插入到
///树。
```

```
/// See also:
///
///  * [StatefulWidget] and [State], for widgets that can build differently
///    several times over their lifetime.
///  * [InheritedWidget], for widgets that introduce ambient state that can
///    be read by descendant widgets.
///  * [StatelessWidget], for widgets that always build the same way given a
///    particular configuration and ambient state.
/// *[statefulwidget]和[state]，用于可以以不同方式构建的小部件
///他们一生中有好几次。
///*[InheritedWidget]，对于引入环境状态的Widget
///由子代小部件读取。
///*[无状态小部件]，对于总是以相同方式构建的小部件
///特定配置和环境状态。

@immutable
abstract class Widget extends DiagnosticableTree {
  /// Initializes [key] for subclasses.
  const Widget({ this.key });

  /// Controls how one widget replaces another widget in the tree.
  @override
  String toStringShort() {
    return key == null ? '$runtimeType' : '$runtimeType-$key';
  }

  @override
  void debugFillProperties(DiagnosticPropertiesBuilder properties) {
    super.debugFillProperties(properties);
    properties.defaultDiagnosticsTreeStyle = DiagnosticsTreeStyle.dense;
  }


  /// Whether the `newWidget` can be used to update an [Element] that currently
  /// has the `oldWidget` as its configuration.
  static bool canUpdate(Widget oldWidget, Widget newWidget) {
    return oldWidget.runtimeType == newWidget.runtimeType
        && oldWidget.key == newWidget.key;
  }
}
```

## Widget 和 Element 之间是怎样的对应关系呢?

从上源码注释也可知： **Widget 和 Element 之间是一对多的关系** 。实际上渲染树是由 Element 实的节点构成的树，而作为配置文件的 Widget 可能被复用到树的多个部分，对应产生多个 Element 象。

## 那么RenderObject 又是什么？它和上述两个的关系是什么?

从源码注释写着 An object in the render tree 可以看出到 RenderObject 才是实际的渲染对象，而过 Element 源码我们可以看出：**Element 持有 RenderObject 和 Widget。**

```
/// The configuration for this element.
```

```
  @override
  Widget get widget => _widget;
  Widget _widget;

  /// The object that manages the lifecycle of this element.
  @override
  BuildOwner get owner => _owner;
  BuildOwner _owner;

...

  /// The render object at (or below) this location in the tree.
  ///
  /// If this object is a [RenderObjectElement], the render object is the one at
  /// this location in the tree. Otherwise, this getter will walk down the tree
  /// until it finds a [RenderObjectElement].
  RenderObject get renderObject {
    RenderObject result;
    void visit(Element element) {
      assert(result == null); // this verifies that there's only one child
      if (element is RenderObjectElement)
        result = element.renderObject;
      else
        element.visitChildren(visit);
    }
    visit(this);
    return result;
  }
```

再结合下面源码，可以大致总结出三者的关系是：

**配置文件 Widget 生成了 Element，而后创建 RenderObject 关联到 Element 的内部 renderObject 对象上，最后Flutter 通过 RenderObject 数据来布局和绘制。** 理论上你也可以认为 RenderObject 是最终给 Flutter 的渲染数据，它保存了大小和位置等信息，Flutter 通过它去绘制出画面。

```
  /// Creates an instance of the [RenderObject] class that this
  /// [RenderObjectWidget] represents, using the configuration described by this
  /// [RenderObjectWidget].
  ///
  /// This method should not do anything with the children of the render object.
  /// That should instead be handled by the method that overrides
  /// [RenderObjectElement.mount] in the object rendered by this object's
  /// [createElement] method. See, for example,
  /// [SingleChildRenderObjectElement.mount].
  @protected
  RenderObject createRenderObject(BuildContext context);
```

说到 RenderObject ，就不得不说 **RenderBox** ： A render object in a 2D Cartesian coordinate system，从源码注释可以看出，它是在继承 RenderObject 基础的布局和绘制功能上，实现了“笛卡尔标系”：以 Top、Left 为基点，通过宽高两个轴实现布局和嵌套的。

RenderBox 避免了直接使用 RenderObject 的麻烦场景，其中 RenderBox 的布局和计算大小是在 performLayout() 和 performResize() 这两个方法中去处理，很多时候我们更多的是选择继承 RenderBox

去实现自定义。

## 综合上述情况，我们知道：

● Widget只是显示的数据配置，所以相对而言是轻量级的存在，而 Flutter 中对 Widget 的也做了一的优化，所以每次改变状态导致的 Widget 重构并不会有太大的问题。

● RenderObject 就不同了，RenderObject 涉及到布局、计算、绘制等流程，要是每次都全部重新建开销就比较大了。

所以针对是否每次都需要创建出新的 Element 和 RenderObject 对象，Widget 都做了对应的判断便于复用，比如：在 newWidget 与oldWidget 的 runtimeType 和 key 相等时会选择使用 newWidgt 去更新已经存在的 Element 对象，不然就选择重新创建新的 Element。

**由此可知：**

**Widget 重新创建，Element 树和 RenderObject 树并不会完全重新创建。**