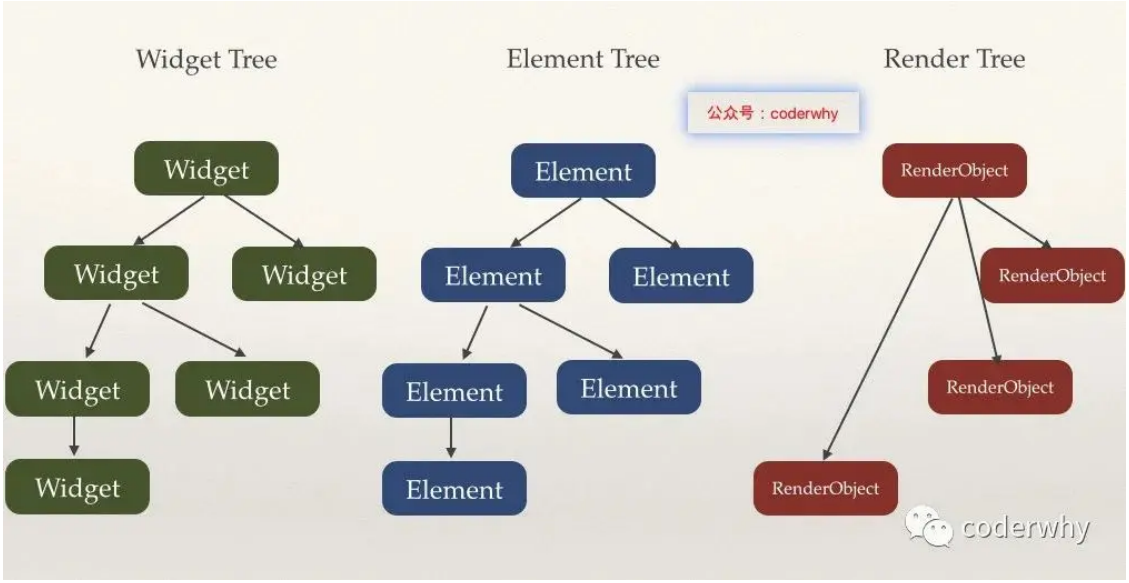
Flutter（十三）Widget-Element-RenderObject

AlanGe

一. Flutter的渲染流程

1.1. Widget-Element-RenderObject关系



图片

1.2. Widget是什么?

图片

官方对 Widget 的说明：

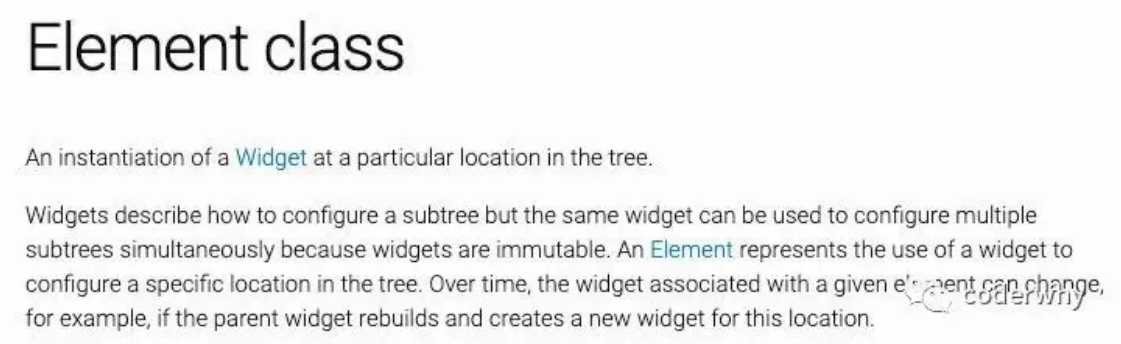
- Flutter 的 Widgets 的灵感来自 React，中心思想是构造你的 UI 使用这些 Widgets。
- Widget 使用配置和状态，描述这个 View（界面）应该长什么样子。
- 当一个 Widget 发生改变时，Widget 就会重新 build 它的描述，框架会和之前的描述进行对比，来决定使用最小的改变（minimal changes）在渲染树中，从一个状态到另一个状态。

自己的理解：

Widget描述控件的宽、高、颜色，控件之间的继承关系等

- Widget 就是一个个描述文件，这些描述文件在我们进行状态改变时会不断的 build。重新进行描述
- 但是对于渲染对象来说，只会使用最小的开销来更新渲染界面。

1.3. Element 是什么？



图片

官方对 Element 的描述：

- Element 是一个 Widget 的实例，在树中详细的位置。
- Widget 描述和配置子树的样子，而 Element 实际去配置在 Element 树中特定的位置。 element 元素，要素

1.4. RenderObject

官方对 RenderObject 的描述：

- 渲染树上的一个对象
- RenderObject 层是渲染库的核心。

二. 对象的创建过程

我们这里以 Padding 为例，Padding 用来设置内边距

2.1. Widget

Padding 是一个 Widget，并且继承自

SingleChildRenderObjectWidget

继承关系如下：

```
Padding -> SingleChildRenderObjectWidget ->
RenderObjectWidget -> Widget
```

我们之前在创建 Widget 时，经常使用 StatelessWidget 和 StatefulWidget，这种 Widget 只是将其他的 Widget 在 build 方法中组装起来，并不是一个真正可以渲染的 Widget（在之前的课程中其实有提到）。

在 Padding 的类中，我们找不到任何和渲染相关的代码，这是因为 Padding 仅仅作为一个配置信息，这个配置信息会随着我们设置的属性不同，频繁的销毁和创建。

问题：频繁的销毁和创建会不会影响 Flutter 的性能呢？

- 并不会，答案在我的另一篇文章中；
- https://mp.weixin.qq.com/s/J4XoXJHJSmn8VaMoz3BZJQ

那么真正的渲染相关的代码在哪里执行呢？

- RenderObject

2.2. RenderObject

我们来看Padding里面的代码，有一个非常重要的方法：

- 这个方法其实是来自RenderObjectWidget的类，在这个类中它是一个抽象方法；
- 抽象方法是必须被子类实现的，但是它的子类SingleChildRenderObjectWidget也是一个抽象类，所以可以不实现父类的抽象方法
- 但是Padding不是一个抽象类，必须在这里实现对应的抽象方法，而它的实现就是下面的实现

```
@override
RenderPadding createRenderObject(BuildContext context) {
  return RenderPadding(
    padding: padding,
    textDirection: Directionality.of(context),
  );
}
```

上面的代码创建了什么呢？RenderPadding

RenderPadding的继承关系是什么呢？

```
RenderPadding -> RenderShiftedBox -> RenderBox ->
RenderObject
```

我们来具体查看一下 RenderPadding 的源代码：

- 如果传入的_padding 和原来保存的 value 一样，那么直接 return；
- 如果不一致，调用_markNeedResolution，而_markNeedResolution 内部调用了 markNeedsLayout；
- 而 markNeedsLayout 的目的就是标记在下一帧绘制时，需要重新布局 performLayout；
- 如果我们找的是 Opacity，那么 RenderOpacity 是调用 markNeedsPaint，RenderOpacity 中是有一个 paint 方法的； Opacity 不透明的，模糊的

```
set padding(EdgeInsetsGeometry value) {
    assert(value != null);
    assert(value.isNonNegative);
    if (_padding == value)
      return;
    _padding = value;
    _markNeedResolution();
  }
```

2.3. Element

我们来思考一个问题：

- 之前我们写的大量的 Widget 在树结构中存在引用关系，但是 Widget 会被不断的销毁和重建，那么意味着

这棵树非常不稳定；

- 那么由谁来维系整个 Flutter 应用程序的树形结构的稳定呢？
- 答案就是 Element。
- 官方的描述：Element 是一个 Widget 的实例，在树中详细的位置。

Element 什么时候创建？

在每一次创建 Widget 的时候，会创建一个对应的 Element，然后将该元素插入树中。

- Element 保存着对 Widget 的引用；

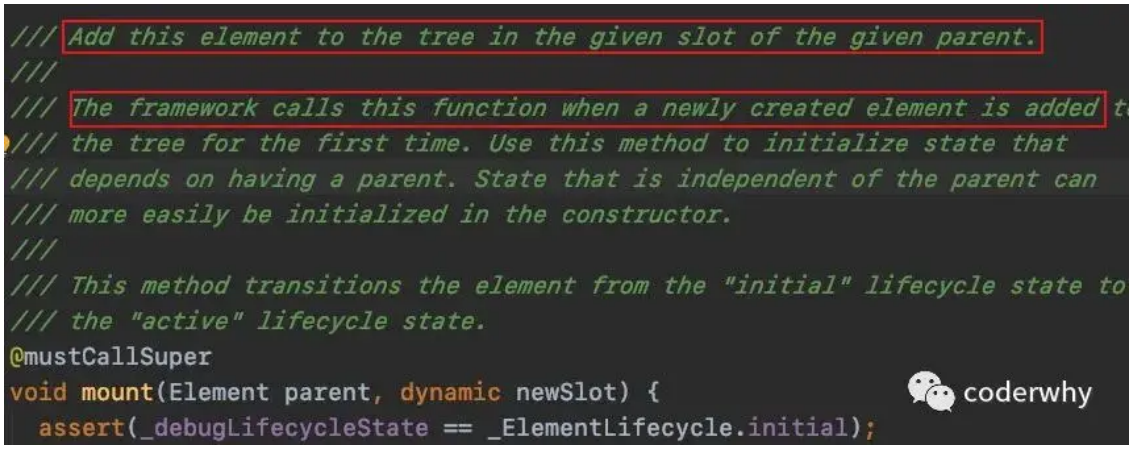
在 SingleChildRenderObjectWidget 中，我们可以找到如下代码：

- 在 Widget 中，Element 被创建，并且在创建时，将 this（Widget）传入了；
- Element 就保存了对 Widget 的应用；

```
@override
SingleChildRenderObjectElement createElement() =>
SingleChildRenderObjectElement(this);
```

在创建完一个 Element 之后，Framework 会调用 mount 方法

来将Element插入到树中具体的位置：



图片

在调用mount方法时，会同时使用Widget来创建RenderObject，并且保持对RenderObject的引用：

- _renderObject = widget.createRenderObject(this);

```
@override
  void mount(Element parent, dynamic newSlot) {
    super.mount(parent, newSlot);
    _renderObject = widget.createRenderObject(this);
    assert(() {
      _debugUpdateRenderObjectOwner();
      return true;
    }());
    assert(_slot == newSlot);
    attachRenderObject(newSlot);
    _dirty = false;
  }
```

但是，如果你去看类似于Text这种组合类的Widget，它也

会执行 mount 方法，但是 mount 方法中并没有调用 createRenderObject 这样的方法。

- 我们发现 ComponentElement 最主要的目的是挂载之后，调用 _firstBuild 方法

```
@override
  void mount(Element parent, dynamic newSlot) {
    super.mount(parent, newSlot);
    assert(_child == null);
    assert(_active);
    _firstBuild();
    assert(_child != null);
  }

  void _firstBuild() {
    rebuild();
  }
```

如果是一个 StatefulWidget，则创建出来的是一个 StatefulElement

我们来看一下 StatefulElement 的构造器：

- 调用 widget 的 createState()
- 所以 StatefulElement 对创建出来的 State 是有一个引用的
- 而 _state 又对 widget 有一个引用　　Element对widget有个引用

```
 StatefulElement(StatefulWidget widget)
     : _state = widget.createState(),
 ....省略代码
 _state._widget = widget;
```

而调用 build 的时候，本质上调用的是_state 中的 build 方法：

```
Widget build() => state.build(this);
```

2.4. build 的 context 是什么

在 StatelessElement 中，我们发现是将 this 传入，所以本质上 BuildContext 就是当前的 Element

```
Widget build() => widget.build(this);
```

我们来看一下继承关系图：

- Element 是实现了 BuildContext 类（隐式接口）

```
abstract class Element extends DiagnosticableTree
implements BuildContext
```

在 StatefulElement 中，build 方法也是类似，调用 state 的 build 方式时，传入的是 this this就是Element

```
Widget build() => state.build(this);
```

2.5. 创建过程小结

Widget只是描述了配置信息：

- 其中包含createElement方法用于创建Element
- 也包含createRenderObject，但是不是自己在调用

Element是真正保存树结构的对象：

- 创建出来后会由framework调用mount方法；
- 在mount方法中会调用widget的createRenderObject对象；
- 并且Element对widget和RenderObject都有引用；

RenderObject是真正渲染的对象：

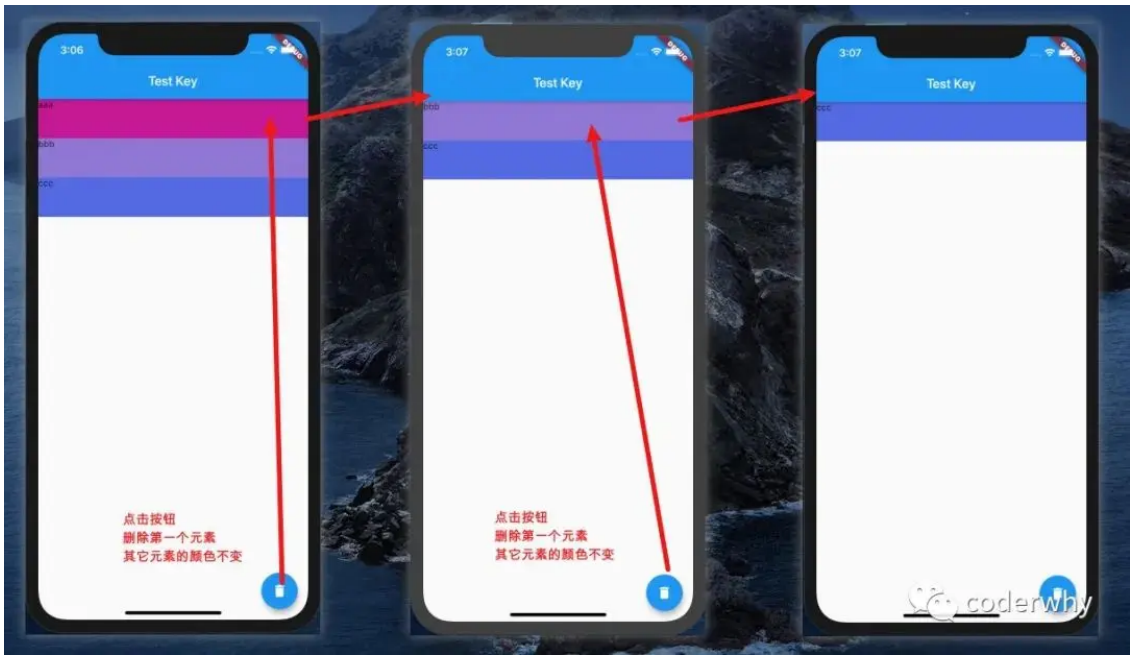
- 其中有`markNeedsLayout` `performLayout` `markNeedsPaint` `paint`等方法

三. Widget的key

在我们创建Widget的时候，总是会看到一个key的参数，它又是做什么的呢?

3.1. key的案例需求

我们一起来做一个key的案例需求



图片

home界面的基本代码：

```
class _HYHomePageState extends State<HYHomePage> {
  List<String> names = ["aaa", "bbb", "ccc"];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Test Key"),
      ),
      body: ListView(
        children: names.map((name) {
```

```
          return ListItemLess(name);
        }).toList(),
      ),

      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.delete),
        onPressed: () {
          setState(() {
            names.removeAt(0);
          });
        }
      ),
    );
  }
}
```

注意：待会儿我们会修改返回的 ListItem 为 ListItemLess 或者 ListItemFul

3.2. StatelessWidget 的实现

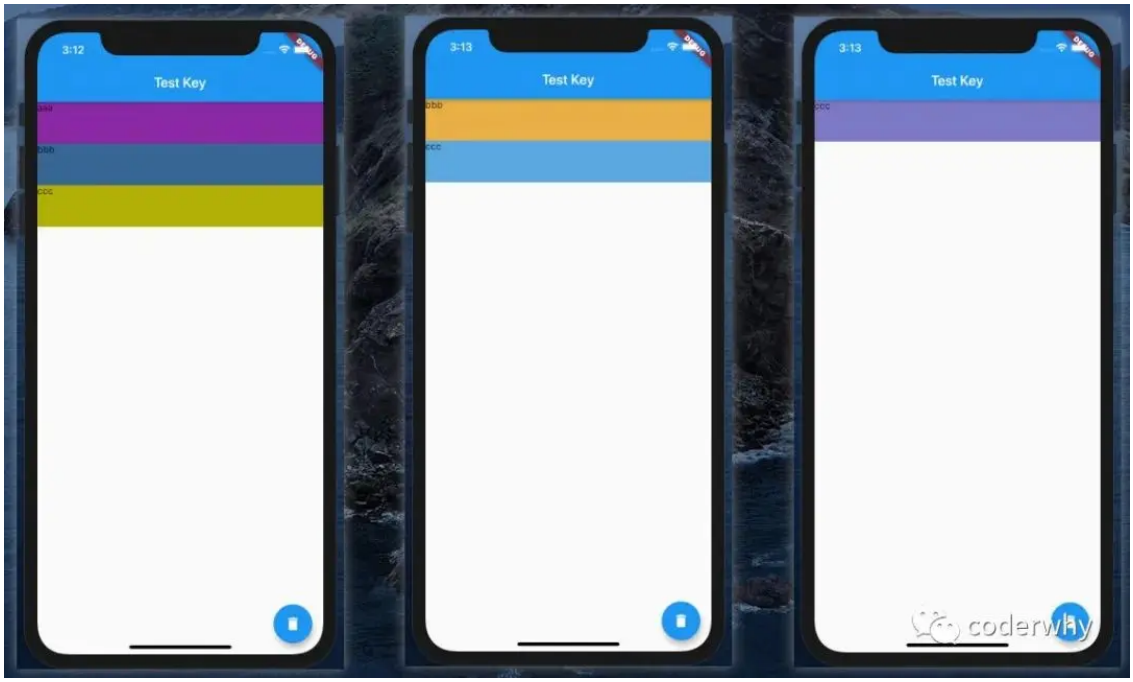
我们先对 ListItem 使用一个 StatelessWidget 进行实现：

```
class ListItemLess extends StatelessWidget {
  final String name;
  final Color randomColor = Color.fromARGB(255,
Random().nextInt(256), Random().nextInt(256),
Random().nextInt(256));

  ListItemLess(this.name);

  @override
```

```
  Widget build(BuildContext context) {
    return Container(
      height: 60,
      child: Text(name),
      color: randomColor,
    );
  }
}
```

它的实现效果是每删除一个，所有的颜色都会发现一次变化

- 原因非常简单，删除之后调用 setState，会重新 build，重新 build 出来的新的 StatelessWidget 会重新生成一个新的随机颜色



图片

3.3. StatefulWidget的实现（没有key）

我们对 ListItem 使用 StatefulWidget 来实现

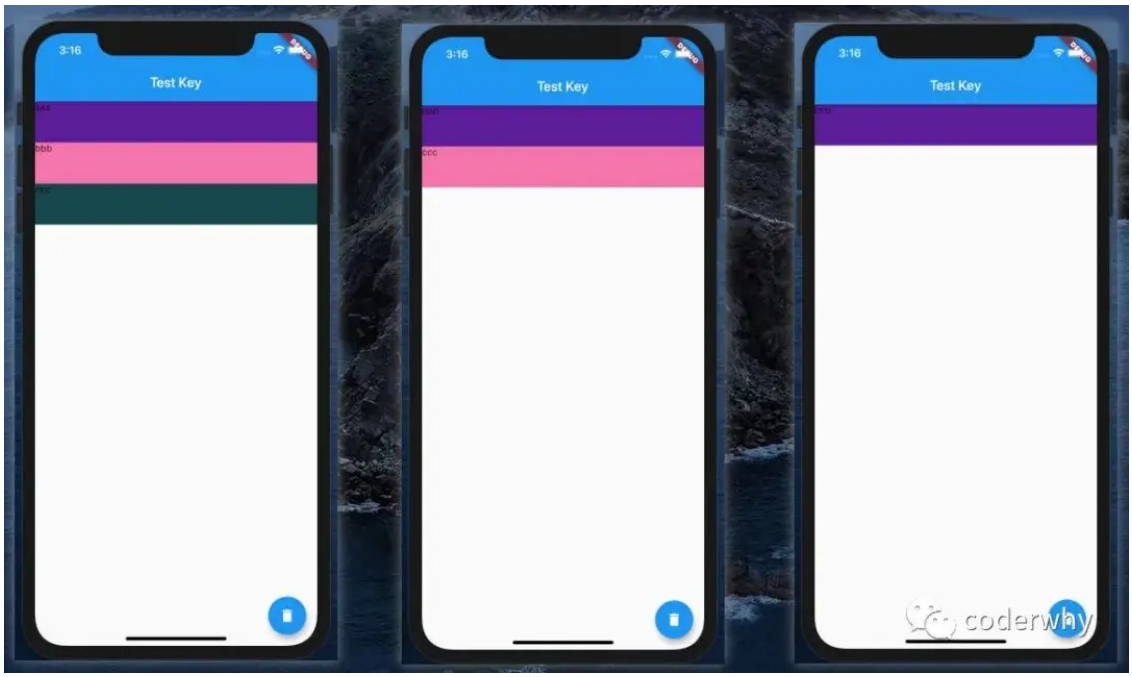
```
class ListItemFul extends StatefulWidget {
  final String name;
  ListItemFul(this.name): super();
  @override
  _ListItemFulState createState() => _ListItemFulState();
}

class _ListItemFulState extends State<ListItemFul> {
  final Color randomColor = Color.fromARGB(255,
Random().nextInt(256), Random().nextInt(256),
Random().nextInt(256));

  @override
  Widget build(BuildContext context) {
    return Container(
      height: 60,
      child: Text(widget.name),
      color: randomColor,
    );
  }
}
```

我们发现一个很奇怪的现象，颜色不变化，但是数据向上移动了

- 这是因为在删除第一条数据的时候，Widget 对应的 Element 并没有改变；
- 而 Element 中对应的 State 引用也没有发生改变；

没有调用_ListItemFulState中的setState方法，就不会重新调用State的build方法，那么widget对应的element就没有改变。

- 在更新 Widget 的时候，Widget 使用了没有改变的 Element 中的 State；



图片

3.4. StatefulWidget 的实现（随机 key）

我们使用一个随机的 key

ListItemFul 的修改如下：

```
class ListItemFul extends StatefulWidget {
  final String name;
  ListItemFul(this.name, {Key key}): super(key: key);
  @override
  _ListItemFulState createState() => _ListItemFulState();
}
```

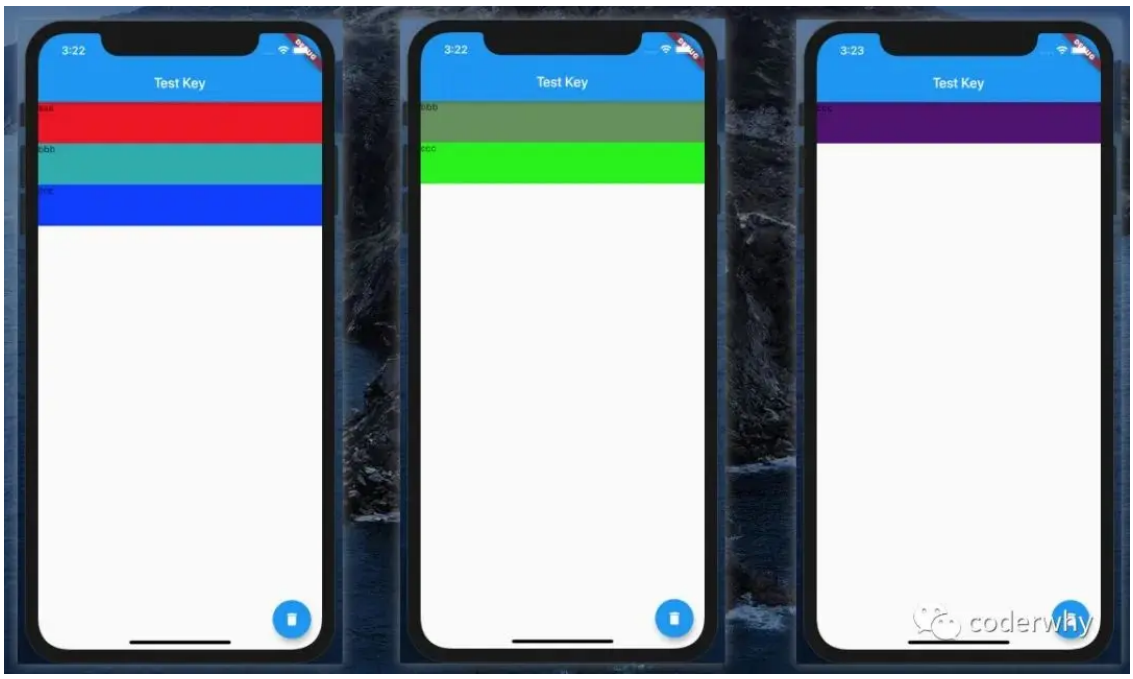
home界面代码修改如下：

```
body: ListView(
  children: names.map((name) {
    return ListItemFul(name, key:
ValueKey(Random().nextInt(10000)),);
  }).toList(),
),
```

这一次我们发现，每次删除都会出现随机颜色的现象：

- 这是因为修改了 key 之后，Element 会强制刷新，那么对应的 State 也会重新创建

```
// Widget类中的代码
static bool canUpdate(Widget oldWidget, Widget newWidget) {
  return oldWidget.runtimeType == newWidget.runtimeType
    && oldWidget.key == newWidget.key;
}
```



图片

3.5. StatefulWidget的实现（name为key）

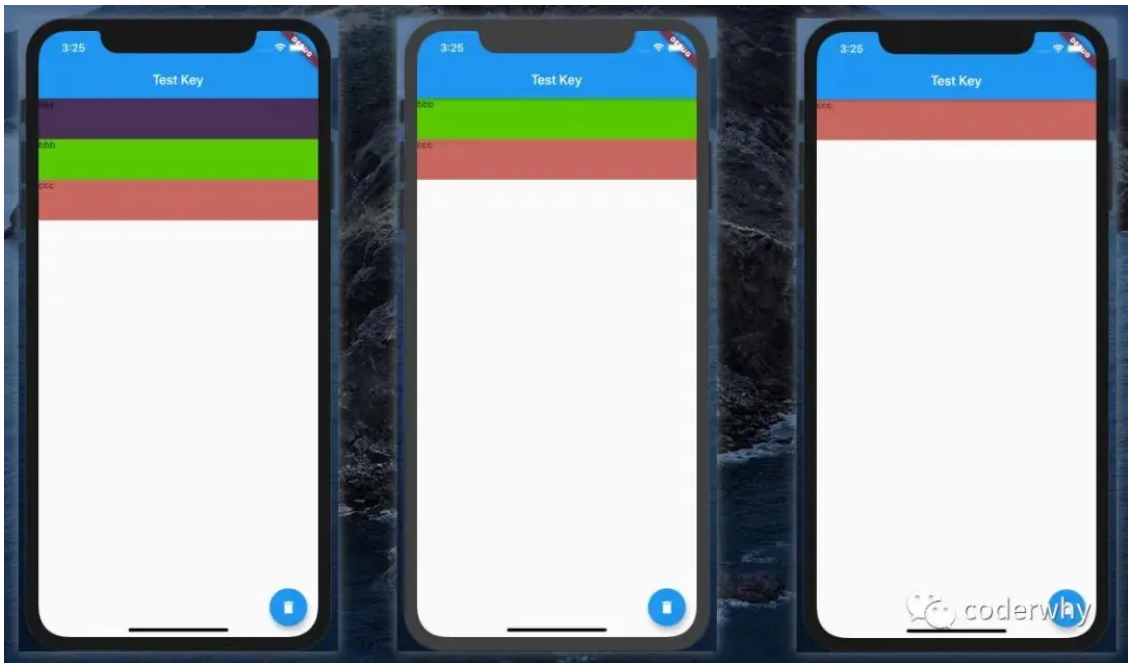
这次，我们将name作为key来看一下结果：

```
body: ListView(
  children: names.map((name) {
    return ListItemFul(name, key: ValueKey(name));
  }).toList(),
),
```

我们理想中的效果：

- 因为这是在更新widget的过程中根据key进行了diff算法
- 在前后进行对比时，发现bbb对应的Element和ccc对应的Element会继续使用，那么就会删除之前aaa对应

的Element，而不是直接删除最后一个Element



图片

3.6. Key的分类

Key本身是一个抽象，不过它也有一个工厂构造器，创建出来一个ValueKey

直接子类主要有：LocalKey和GlobalKey

- LocalKey，它应用于具有相同父Element的Widget进行比较，也是diff算法的核心所在；
- GlobalKey，通常我们会使用GlobalKey某个Widget对应的Widget或State或Element

3.6.1. LocalKey

LocalKey有三个子类

ValueKey：

- ValueKey是当我们以特定的值作为key时使用，比如一个字符串、数字等等

ObjectKey：

- 如果两个学生，他们的名字一样，使用name作为他们的key就不合适了
- 我们可以创建出一个学生对象，使用对象来作为key

UniqueKey

- 如果我们要确保key的唯一性，可以使用UniqueKey；
- 比如我们之前使用随机数来保证key的不同，这里我们就可以换成UniqueKey；

3.6.2. GlobalKey

GlobalKey可以帮助我们访问某个Widget的信息，包括Widget或State或Element等对象

我们来看下面的例子：我希望可以在HYHomePage中直接访

问HYHomeContent中的内容

```
class HYHomePage extends StatelessWidget {
  final GlobalKey<_HYHomeContentState> homeKey =
GlobalKey();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("列表测试"),
      ),
      body: HYHomeContent(key: homeKey),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.data_usage),
        onPressed: () {
          print("${homeKey.currentState.value}");
          print("${homeKey.currentState.widget.name}");
          print("${homeKey.currentContext}");
        },
      ),
    );
  }
}

class HYHomeContent extends StatefulWidget {
  final String name = "123";

  HYHomeContent({Key key}): super(key: key);

  @override
  _HYHomeContentState createState() =>
_HYHomeContentState();
}
```

```dart
class _HYHomeContentState extends State<HYHomeContent> {
  final String value = "abc";

  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```