Flutter中的Key详解

发布于2022-01-14 14:10:22 阅读 430

在Flutter中，几乎每一个Widget都有一个key。虽然我们在日常的开发中极少会使用到这个key，但是实际上key的存在是很有必要的。那么key到底是什么？它有什么作用？在哪些场景下会必要要使用key呢？

接下来我们就详细聊聊key。

踩过的坑

对于一个List列表，比如说银行卡列表、新闻列表等，列表中的单个元素的UI组件我们一般是要对其进行封装复用的，这样的话，在循环引用的时候就会出现很多同级的该Widget实例。此时注意，**当复用的widget是Stateful类型的widget时，我们一定要为其指定key以对其做唯一标识，否则就会因为复用机制而出现意想不到的Bug**。同样的，**如果这个共用的StatefulWidget里面有使用到了另外一个StatefulWidget，那么在应用的时候也必须要为另外的这个statefulWidget指定key**，否则的话在状态管理方面一定会出现问题。

一般而言，上述场景中指定的Key使用ValueKey即可，参数就传某个唯一标识就行，比如id。

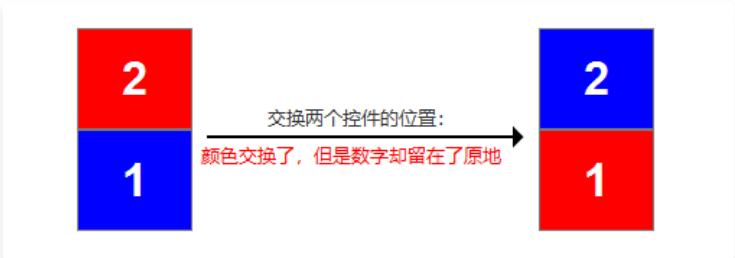
Key是什么

在Flutter中，**Key是不能重复使用的**，所以Key一般用来做唯一标识。组件在更新的时候，其状态的保存主要是通过判断**组件的类型**或者**key值**是否一致。因此，当各组件的类型不同的时候，类型已经足够用来区分不同的组件了，此时我们可以不必使用key。但是如果同时存在多个同一类型的控件的时候，此时类型已经无法作为区分的条件了，我们就需要使用到key。

举个简单的例子，如果将两个乒乓球A和B随机打乱，从中任意挑出一个，你知道你拿到的是A还是B吗？但是如果在两个乒乓球上分别标出字母A和B，那就一目了然了，这就是Key存在的意义。这时你可能会问，如果不使用Key来做唯一的标识，拿错了就拿错了呗，有什么后果吗？

没有Key的时候会发生什么

先来看个例子：一个Column布局中垂直放置两个同类型的stateful有状态组件，其中color直接作为statefulWidge的属性，而count存在于state中。当交换两个控件的位置时，情况如下图所示：

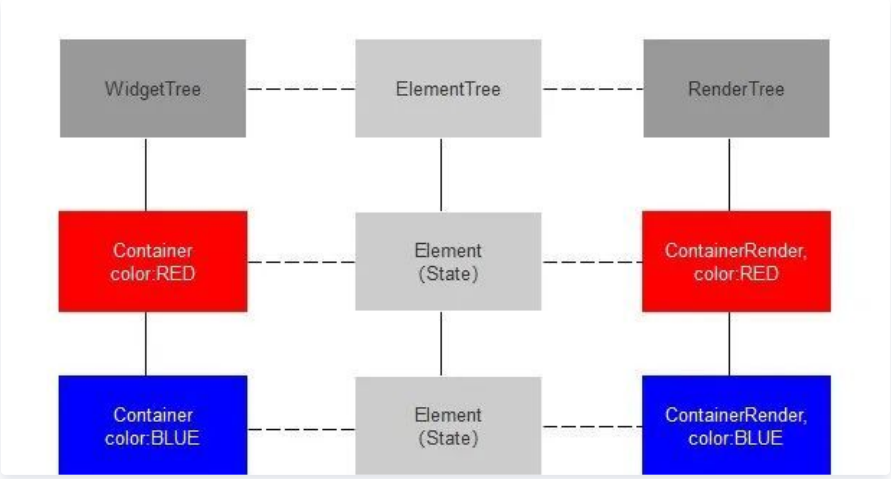


控件对应的代码如下：

```
class CustomButton extends StatefulWidget {
  final Color color;
  CustomButton(this.color);

  @override
  _CustomButtonState createState() => _CustomButtonState();
}

class _CustomButtonState extends State<StatefulContainer> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return InkWell(
      onTap: () {
        setState(() {
          count++;
        });
      },
      child: Container(
        width: 100,
```

```
        height: 100,
        color: widget.color,
        child: Text("$count", style: TextStyle(fontSize: 20, color: Colors.white)),
      ),
    );
  }
}
```

我之前在Widget，构建Flutter界面的基石中介绍过，Flutter的界面渲染离不开三棵树：Widget、Element、RenderObject。当Widget被加载的时候，它并不是马上就会被绘制出来，而是会先创建出它对应的Element，然后Element再根据Widget的配置信息在对应位置生成一个RenderObject，从而进行绘制的：



我在在Widget，构建Flutter界面的基石中详细介绍过Widget、Element、RenderObject，这里再简单描述下：

1，Widget，主要用来配置组件的不可变信息，如上面例子中的颜色信息。我们编写的众多Widget之间的相互嵌套，最终会组合成一个树，系统通过分析当前Widgets树的结构来决定这个页面应该被展示成什么样子，widget本身只是作为配置信息的载体存在，真正负责UI渲染的是下面的RenderObject。

2，Element，**在创建Widget的时候都会在对应的位置上创建一个Element（对于Stateful类型的widget来说，state就是存储在Element中的），Element又会根据widget的配置信息在对应位置上生成一个RenderObject的实例，在每一个Element中都持有着相对应的widget和renderObject的引用，是联系二者的桥梁**。

3，RenderObject，保存了widget的颜色、大小等布局相关信息，用来进行最终的UI绘制。

基于Element的复用机制的解释

在Flutter中，**Widget是不可变的**，它仅仅作为配置信息的载体而存在，并且任何配置或者状态的更改都会导致Widget的销毁和重建，但好在Widget本身是非常轻量级的，因此实际耗费的性能很小。与之相反，RenderObject就不一样了，实例化一个RenderObject的成本是非常高的，频繁地实例化和销毁RenderObject对性能的影响非常大，因此为了高性能地构建用户界面，Flutter使用Element的复用机制来尽可能地减少RenderObject的频繁创建和销毁。**当Widget改变的时候，Element会通过组件类型以及对应的Key来判断旧的Widget和新的Widget是否一致**：

1，**如果某一个位置的旧Widget和新Widget不一致，就会重新创建Element**，重建Element的同时也重建了RenderObject；

2，**如果某一个位置的旧Widget和新Widget一致**，只是配置发生了变化，比如组件的颜色变了，**此时Element就会被复用**，而只需要修改Widget对应的Element的RenderObject中的颜色设置即可，无需再进行十分耗性能的RenderObject的重建工作。

我们再来看上面的例子，当我们在不指定Key的情况下交换两组件的位置，由于组件类型并未发生变化，此时**Element树中第一位置存储了数字2的element发现widget树中第一位置新的Widget和它创建的RenderObject中旧的widget一致（因为类型一样），因此就建立了对应关系复用了State中的数字**；同样的道理，Element树中第二位置存储了数字1的Element发现，Widget树中第二位置上新的Widget与他创建的RenderObject中旧的Widget一致，也就建立了对应关系复用了state中的数字。最终的结果就是，虽然Widget被交换了位置，但是所有的Element还是按照原来的位置被重新复用了；同时因为Element的复用，当颜色发生变化的时候，RenderObject也不会被销毁重建，只是修改其中的颜色配置，然后再渲染到UI上。

添加Key之后

修改上例中组件CustomButton的构造方法：

```
...
CustomButton(this.color, {Key key}) : super(key: key);
...
```

使用的时候：

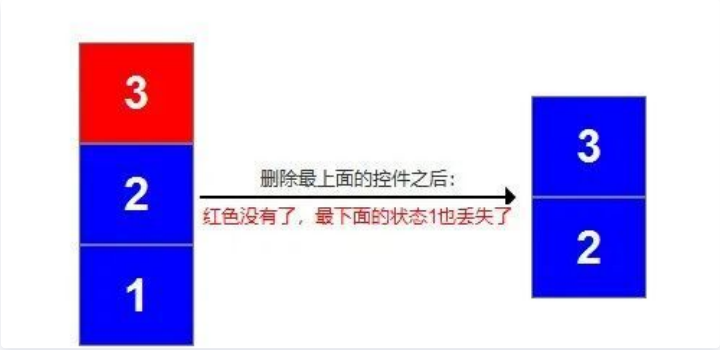
```
Column(
  children: [
    CustomButton(Colors.red, key:ValueKey('A'),),
    CustomButton(Colors.blue, key:ValueKey('B'),),
  ],
);
```

再次交换两组件的位置，我们发现颜色和数字都发生了变化，这是因为**使用了Key之后，此时Element树中第一位置存储了数字2的Element发现Widget树中第一位置上的最新的Widget和它创建的RenderObject中旧的widget不一致了（因为Key不一样了），此时Element不会立刻被销毁，而是会继续在同级目录下逐个查找，如果能找到和旧的Widget一致的新的Widget，那么该Element还是会被保存下来复用，并重新建立Element和新widget位置的对应关系**；相反，如果没有找到一致的，那么旧的Element就会被销毁而重新创建，而一旦Element被销毁，那么其中保存的state也会丢失，对应的旧的RenderObject也会被销毁。

如上所述，Element树中第一位置存储了数字2的Element会继续对比Widget树中第二位置的widget，此时发现一致（因为Key一致），则建立对应关系并复用Element；同理，Element树中第二位置存储了数字1的Element对比发现widget树中第一位置的widget跟旧的widget一致，也建立了对应的关系并复用Element，这样，**最终因为加了Key，Element也随Key准确对应到了新的widget上**（也可以理解成，新的Widget通过Key准确找到了旧widget对应的element）。

需要注意的是，**上面使用的Key是ValueKey，如果使用UniqueKey，你会发现每次交换位置之后，对应的数字均被清零了**，这是因为UniqueKey在每次页面刷新的时候都会重新生成另外的新值，也就是说key发生了变化，这样各个位置上的新老Widget对比均不一样，那么Element就会被销毁重新生成。

接下来我们再来看一个当没有Key时删除某一个控件的例子：



当删除最上面的红色组件之后，Element树中第一位置存储了数字3的Element发现Widget树中第一位置的新的widget和他创建的RenderObject中旧的widget一致，因此就建立了对应关系复用了state中的数字；同理，Element树中第二位置存储了数字2的Element，它发现Widget树中第二位置的新的widget和它创建的RenderObject中旧的Widget一致，于是也建立了对应关系并且复用了state中的数字；而Element树中第三位置上存储了数字1的Element，去找Widget树中对应位置上的widget的时候，发现不存在，就会认为widget树中第三位置的widget 被删除了（实际上被删除的是第1个widget，但是由于没有key做唯一标识，Element已经分不清谁是谁了），此时对应的Element树中位于第三位置上的存储了数字1的Element就会被删除。

组件新增的时候也是一样的道理，这里不赘述。

综上所述，**Key的存在是在某些特定场景下才会有意义的，大多数情况下我们并不需要用到Key，但是当我们需要对同级目录下多个相同类型的StatefulWidget组件进行添加、移除或者重新排序的时候，那就需要使用Key，否则就会出现意想不到的问题**。典型的一个场景就是：ListView组件中的Item组件公用。

那么，我们该如何去创建一个Key呢?

Key的种类及用法

flutter 中的key总的来说分为以下两种：

局部键（LocalKey）：ValueKey、ObjectKey、UniqueKey

全局键（GlobalKey）：GlobalObjectKey

1，**ValueKey**

ValueKey是通过某个具体的Value值来做区分的Key，如下：

```
key: ValueKey(1),
key: ValueKey("2"),
key: ValueKey(true),
key: ValueKey(0.1),
key: ValueKey(Person()), // 自定义类实例
```

可以看到，**ValueKey的值可以是任意类型，甚至可以是我们自定义的类的实例。**

例如，现在有一个展示所有学生信息的ListView列表，每一项itemWidget所对应的学生对象均包含某个唯一的属性，例如学号、身份证号等，那么这个时候就可以使用ValueKey，其值就是对应的学号或者身份证号。

当类型为自定义的对象的时候，判断两个Key是否相等的依据是两个对象==运算的结果。

例如，自定义的Student类如下，仅包含一个name参数：

```
class Student {
  final String name;
  Student(this.name);
}
```

现在我新建两个对象，其name属性赋值一样：

```
Student s1 = Student(name: "gpf");
Student s2 = Student(name: "gpf");
print(s1==s2);  // 返回false，因为两个对象在内存中的地址不一样
```

可见，属性值相等的两个对象仍然有可能是不一样的。

现在来看下面两个ValueKey：

```
// 每次页面刷新，都会new一个全新对象，因此下面的写法类似于UniqueKey()
CustomButton(Colors.blue, key: ValueKey(new Student('gpf'))),
CustomButton(Colors.red, key: ValueKey(new Student('gpf'))),
```

对于上面的代码写法，当交换位置时，Element会判断新老widget的key不一样（因为每一次刷新都new了新的Student对象，并且通过==判断这两个Student对象是不相等的），因此该Element就不会被重用，因此状态会丢失。为了避免状态丢失，我们可以将创建Student对象放在外面，然后在ValueKey中引用即可，这样Student对象就不会随着页面刷新被重新创建，刷新前后对象就一致了，此时交换组件位置就会发现状态和颜色都发生了交换。

但是如果我们**重写了自定义的Student类中==判断和hashCode函数**，规定了判等规则，如下：

```
class Student {
  final String name;
  Student(this.name);

  @override
  bool operator ==(Object o) =>
      identical(this, o) ||
      o is Student && runtimeType == o.runtimeType && name == o.name;

  @override
  int get hashCode => name.hashCode;
}
```

此时再比较两个属性相同的对象就会发现：

```
print(s1==s2);  // 返回了true，虽然==比较返回了true，但两个对象在内存中的地址还是不一样的
```

虽然此时s1和s2是相等的，但是两个对象在内存中的地址还是不一样的。

这样一来，同样的写法：

```
CustomButton(Colors.blue, key: ValueKey(new Student('gpf'))),
CustomButton(Colors.red, key: ValueKey(new Student('gpf'))),
```

我们会发现，出现了异常，因为两个组件的key一样了。

2，**ObjectKey**

ObjectKey的使用场景如下：

现有一个所有学生信息的ListView列表，每一项itemWidget对应的学生对象不存在某个唯一的属性（比如学号、身份证号），任一属性均有可能与另外一名学生重复，只有多个属性组合起来才能唯一的定位到某个学生，那么此时使用ObjectKey就最合适不过了。

ObjectKey判断两个Key是否相同的依据是：两个对象是否具有相同的内存地址：

```
CustomButton(Colors.blue, key: ObjectKey(new Student('gpf'))),
CustomButton(Colors.red, key: ObjectKey(new Student('gpf'))),
```

对于上面的代码，**不论自定义对象Student是否重写了==运算符判断，均会被视为不同的Key**，交换位置后刷新页面，所有的数字都会被清零。

除非将创建的Student对象放在build方法之外，然后在ObjectKey中引用，这样页面刷新前后引用的还是同一个内存地址中的对象，Key在刷新前后就一样了，此时数字也会随着颜色一起交换了，如下：

```
Student s1 = Student(name: "gpf");
Student s2 = Student(name: "gpf");
.....
CustomButton(Colors.red, key: ObjectKey(s1)),
CustomButton(Colors.blue, key: ObjectKey(s2)),
```

3，**UniqueKey**

顾名思义，**UniqueKeyshi一个唯一键，不需要参数，并且每一次刷新都会生成一个新的Key。**

典型的一个使用场景就是AnimatedSwitcher：

```
AnimatedSwitcher(
  duration: Duration(milliseconds: 1000),
  child: Container(
    key: UniqueKey(),
    height: 100,
    width: 100,
    color: Colors.red,
  ),
)
```

默认情况下，AnimatedSwitcher组件的动画效果只有在child发生变化的时候才会出现，这里的变化指的是child组件的类型或者Key发生变化。比如每修改一次Container组件的高度，刷新之后都想触发动画的话，就必须保证每次刷新前后child组件的key不一样（因为组件类型始终没变），此时UniqueKey刚好合适，否则就不会有动画效果。

4，**GlobalKey && GlobalObjectKey**

GlobalObjectKey和上面的局部键ObjectKey是否相等的判定规则是一样的，用法也类似，这里不针对GlobalObjectKey做过多解释，我们接下来看一下GlobalKey的使用。

GlobalKey是全局唯一的键，一般而言，GlobalKey有如下几种用途：

用途1：获取配置、状态以及组件位置尺寸等信息

```
class MainPage extends StatelessWidget {

  final GlobalKey<_CounterState> _globalKey = GlobalKey();

  @override
  Widget build(BuildContext context) {
    return Material(
      child: Column(
        children: [
          SizedBox(
            height: 80,
          ),
          Center(
            child: Counter(
              fontSize: 26,
              key: _globalKey,
            ),
          ),
          RaisedButton(
            child: Text("获取及更新控件状态"),
            onPressed: () {
              final _CounterState _counterState = _globalKey.currentState;
              print("当前状态：${_counterState._count}");
              // 更新组件状态
              _counterState.setState(() {
                _counterState._count++;
              });
            },
          ),
          RaisedButton(
```

```
            child: Text("获取控件蓝图信息"),
            onPressed: () {
              final Counter counterWidget = _globalKey.currentWidget;
              print("当前的字体大小为: ${counterWidget.fontSize}");
            },
          ),
          RaisedButton(
            child: Text("获取控件大小及位置"),
            onPressed: () {
              RenderBox renderBox = _globalKey.currentContext.findRenderObject();
              print("组件的宽度: ${renderBox.size.width}");
              print("组件的高度: ${renderBox.size.height}");
              double dx = renderBox.localToGlobal(Offset.zero).dx;
              double dy = renderBox.localToGlobal(Offset.zero).dy;
              print("组件的x坐标: $dx");
              print("组件的y坐标: $dy");
            },
          ),
        ],
      ),
    );
  }
}

class Counter extends StatefulWidget {
  double fontSize;
  Counter({this.fontSize, Key key}) : super(key: key);

  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  @override
  Widget build(BuildContext context) {
    return InkWell(
      child: Container(
        child: Text(
          '$_count',
          style: TextStyle(fontSize: widget.fontSize),
        ),
      ),
      onTap: () {
        setState(() {
          _count++;
        });
      },
    );
  }
}
```

(1) _globalKey.currentWidget：**获取当前组件的配置信息（存在widget树中）**

(2) _globalKey.currentState：**获取当前组件的状态信息（存在Element树中）**

(3) _globalKey.currentContext：**获取当前组件的大小以及位置信息。**

用途2：实现控件的局部刷新

将需要单独刷新的widget从复杂的布局中抽离出去，然后通过传GlobalKey引用，这样就可以通过GlobalKey实现跨组件的刷新了。

示例如下：

```
import 'package:flutter/material.dart';

class TestGpfPage extends StatefulWidget {
  @override
  _TestGpfPageState createState() => _TestGpfPageState();
}
```

```dart

class _TestGpfPageState extends State<TestGpfPage> {

  int _count=0;
  GlobalKey<_TextWidgetState> textKey = GlobalKey();

  @override
  void initState() {
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("flutter局部刷新"),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            TextWidget(textKey),///需要更新的Text
            SizedBox(height: 20,),
            Text(_count.toString(),style: TextStyle(fontSize: 20),), ///这个Text不会刷新，只会刷
            SizedBox(height: 20,),
            RaisedButton(
              child: Text("count++"),
              onPressed: (){
                _count++;
                /// 调用该key对应的StatefulWidget类型组件的refreshCount方法来刷新显示
                textKey.currentState.refreshCount(_count);
              },
            ),
          ],
        ),
      ),
    );
  }
}

class TextWidget extends StatefulWidget {

  TextWidget(Key key) : super(key: key);

  @override
  _TextWidgetState createState() => _TextWidgetState();
}

class _TextWidgetState extends State<TextWidget> {

  String _text="0";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(_text,style: TextStyle(fontSize: 20),),
    );
  }

  void refreshCount(int count) {
    setState(() {
      _text = count.toString();
    });
  }
}
```

文章分享自微信公众号：

iOS小生活

复制公众号名称

本文参与 腾讯云自媒体分享计划，欢迎热爱写作的你一起参与！

作者：拉维

原始发表时间：2022-01-04

如有侵权，请联系 cloudcommunity@tencent.com 删除。

网站 Flutter iOS Android 存储

举报

点赞 3

分享