# Flutter 中的 Key

# 前言

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

  .............
}
```

Flutter 中一切皆 Widget，而 Widget 的构造方法中有个可选参数 Key。一般情况下我们不需要用到这个 Key，不设置这个参数即可。
当需要在一个 StatefulWidget 集合中进行添加、删除、重排序等操作时，就需要传入 Key 了。

# 先来看一个例子

场景：页面上有两个颜色块和一个按钮，点击按钮切换两个颜色块的位置。
下面分别用例子来演示 Stateless 和 Stateful 两种方式实现颜色块。

## Stateless 方式

颜色块代码：

```
class StatelessColorfulTile extends StatelessWidget {
  final Color color = UniqueColorGenerator.getColor();

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

  @override
  Widget build(BuildContext context) => Container(width:
100, height: 100, color: color);
}
```

其中生成颜色的 UniqueColorGenerator 的代码如下：

```
class UniqueColorGenerator {
  static List<Color> colors = [
    Colors.red,
    Colors.green,
    Colors.blue,
    Colors.yellow,
    Colors.purple,
  ];
  static Random random = Random();

  static Color getColor() =>
colors[random.nextInt(colors.length)];
}
```

接下来我们实现一个页面，把两个颜色块放到一个 Row 中去，并添加一个 FloatingActionButton 用于实现切换颜色块位置的功能。代码如下：

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Key Demo',
      theme: ThemeData(primarySwatch: Colors.teal),
      home: PositionedTiles(),
    );
  }
}

class PositionedTiles extends StatefulWidget {
  @override
  State<StatefulWidget> createState() =>
PositionedTilesState();
}

class PositionedTilesState extends State<PositionedTiles> {
  List<Widget> _tiles;

  @override
  void initState() {
    super.initState();
    _tiles = [StatelessColorfulTile(),
```

```
StatelessColorfulTile()];
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Key Demo")),
      body: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: _tiles,
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.swap_horiz),
        onPressed: swapTiles,
      ),
    );
  }

  void swapTiles() {
    setState(() {
      _tiles.insert(1, _tiles.removeAt(0));
    });
  }
}
```

运行一下看看，发现点击右下角的 FAB 按钮能够正常的切换两个颜色块的位置。

接下来再看看 Stateful 方式的颜色块能不能正常切换。

## Stateful 方式

颜色块代码改为用 StatefulWidget

```
class StatefulColorfulTile extends StatefulWidget {
  StatefulColorfulTile({Key key}) : super(key: key);

  @override
  State<StatefulWidget> createState() =>
_StatefulColorfulTileState();
}

class _StatefulColorfulTileState extends
State<StatefulColorfulTile> {
  final Color color = UniqueColorGenerator.getColor();

  @override
  Widget build(BuildContext context) => Container(width:
100, height: 100, color: color);
}
```

把 PositionedTilesState 中的 _tiles 的内容替换为 StatefulColorfulTile

```
  @override
  void initState() {
    super.initState();
    _tiles = [StatefulColorfulTile(),
StatefulColorfulTile()];
  }
```

最后运行一下看看，是不是能够正常切换两个颜色块的位置了？答案是：不能切换了。Interesting!

## Stateful + Key 方式

要想让 StatefulWidget 也能正常切换的话，就需要用到 Key 这个参数，给每个颜色块传入一个独立的 Key。
先不管原理，我们试一下再说。
把 PositionedTilesState 中的 _tiles 的内容修改如下，添加 Key：

```
@override
void initState() {
  super.initState();
  _tiles = [
    StatefulColorfulTile(key: UniqueKey()),
    StatefulColorfulTile(key: UniqueKey()),
  ];
}
```

然后在运行一下看看，发现可以正常切换颜色块的位置了。
关于原理，我们要从 Widget / Element 的更新机制说起。

# Widget / Element 的更新机制

Widget 源码中有个 canUpdate 方法：

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

  final Key key;

  .........

  static bool canUpdate(Widget oldWidget, Widget newWidget)
{
    return oldWidget.runtimeType == newWidget.runtimeType
        && oldWidget.key == newWidget.key;
  }
}
```

我们知道 Widget 只是一个配置，是不可以修改的。Element 才是真正被使用的对象，它是可以修改的。

当有新的 Widget 时，会比较新旧 Widget 的类型和 Key，如果完全一样，则返回 True，表示只需要更新 Widget 即可，和 Widget 关联的 Element 不需要更新，Element 指向新的 Widget。反之如果类型或者 Key 不一样，则返回 false，Widget 和 Element 都需要更新。

当我们不传入 Key 的时候，只比较 runtimeType。由于例子中我们的两个颜色块是同一个类型的，所以 canUpdate 都返回 true。

下面通过源码看看 Stateless 和 Stateful 两种更新机制

## Stateless

看下 StatelessWidget 的源码：

```
abstract class StatelessWidget extends Widget {
  const StatelessWidget({ Key key }) : super(key: key);

  @override
  StatelessElement createElement() =>
StatelessElement(this);

  @protected
  Widget build(BuildContext context);
}
```

可以看到 StatelessWidget 关联了 StatelessElement。

看下 StatelessElement 的源码：

```
class StatelessElement extends ComponentElement {
  StatelessElement(StatelessWidget widget) : super(widget);

  @override
  StatelessWidget get widget => super.widget as
StatelessWidget;

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

```
  ..........
}
```

可以看到 StatelessElement 会调用 StatelessWidget 的 build 方法来获取 Widget。

所以，当新的 Widget 来了，直接调用新的 Widget 的 build 方法就能够更新画面了，不需要更新 StatelessElement。

这就是 canUpdate 返回 true，也能正常切换颜色块的原因。

## Stateful

看下 StatefulWidget 的源码：

```
abstract class StatefulWidget extends Widget {
  const StatefulWidget({ Key key }) : super(key: key);

  @override
  StatefulElement createElement() => StatefulElement(this);

  @protected
  State createState();
}
```

可以看到 StatefulWidget 关联了 StatefulElement。

看下 StatefulElement 的源码：

```
class StatefulElement extends ComponentElement {
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        super(widget) {
    ..........
  }

  @override
  Widget build() => _state.build(this);

  State<StatefulWidget> get state => _state;
  State<StatefulWidget> _state;

  ..........
}
```

没有设置key，canUpdate返回true的情况：

可以看到，StatefulElement 会调用 State 的 build 方法来获取 Widget。所以，当新的 Widget 来了，canUpdate 返回 true，虽然 StatefulWidget 更新了，但是 StatefulElement 中的 _state 还是老的 StatefulWidget 的 state，自然页面也不会有什么变化了。所以 StatefulElement 必须要更新才能正常切换颜色块。

可以看到 RenderObjectElement 中的 updateChildren 方法中有这么一段源码：

```
Map<Key, Element> oldKeyedChildren;
......
```

```
final Widget newWidget = newWidgets[newChildrenTop];
if (haveOldChildren) {
  final Key key = newWidget.key;
  if (key != null) {
    oldChild = oldKeyedChildren[key];
  ........
}

.........

final Element newChild = updateChild(oldChild, newWidget,
IndexedSlot<Element>(newChildrenTop, previousChild));
newChildren[newChildrenTop] = newChild;
previousChild = newChild;

.........
```

设置了key，canUpdate返回false的情况：

如果设置了 Key，那么 RenderObjectElement 就会用新的 Widget 的 Key 在老的 Element 列表中搜索，找出匹配这个 Key 的 Element 来更新，如果没有一样 Key 的 Element，则创建一个新的 Element。伴随着 Element 的更新，对应的 RenderObject 也会跟着更新，自然画面也就正常变化了。

找得到key，只更新位置

1.找不到key，则创建新的Element，位置和颜色都会更新；找得到key，则更新Element，只更新位置，颜色不变化

上文提到的在老的 Element 列表中搜索新的 Widget 的 Key 匹配的 Element，这个老的 Element 列表必须被一个父 Element 包含着。如果是不同的父 Element，是检索不到的。比如下面的例子，我们把 _tiles 里面的两个颜色块再包裹一层：

key和widget必须在同一级，被同一个widget包裹着，就检索得到

```
@override
void initState() {
  super.initState();
  _tiles = [
    Container(
      child: StatefulColorfulTile(key: UniqueKey()),
    ),
    Container(
      child: StatefulColorfulTile(key: UniqueKey()),
    ),
  ];
}
```

这时候运行一下看看，颜色块还是能切换的。但是这时候因为在老的 Element 列表里面检索不到，所以会重新创建一个新的 Element，你会发现颜色会随机变化，已经不是原来的颜色了。

为了解决问题，必须把 Key 设置到同一个父 Widget 的两个 Container 上去，如下：

```
@override
void initState() {
  super.initState();
  _tiles = [
    Container(
      key: UniqueKey(),
      child: StatefulColorfulTile(),
    ),
    Container(
```

```
        key: UniqueKey(),
        child: StatefulColorfulTile(),
      ),
    ];
  }
```

运行之后发现颜色不会随机变化了。

至此，你已经了解了 Key 的作用以及原理的。

# 那么到底有哪几种 Key 呢?

## 1. Key

```
@immutable
abstract class Key {
  const factory Key(String value) = ValueKey<String>;

  @protected
  const Key.empty();
}
```

Key 默认是使用 ValueKey

Key 有两个子类 LocalKey 和 GlobalKey

## 2. LocalKey

LocalKey 的用途是同一个父 Widget 下的所有子 Widget 进行比较。比如上文提到的例子。

Localkey 有三个子类

1. ValueKey：以一个值作为 Key
2. ObjectKey：以一个对象作为 Key。当多个值才能唯一标识的时候，将这多个值组合成一个对象。比如【学校 + 学号】才能唯一标识一个学生。
3. UniqueKey：生成唯一随机数（对象的 Hash 值）作为 Key。注意：如果直接在控件构建的时候生成，那么每次构建都会生成不同的 Key。

Valuekey 有个子类：PageStorageKey，专门用于存储页面滚动位置。

## 3. GlobalKey

通过 GlobalKey 能够跨 Widget 访问状态。

看一个例子，如下：

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

class MyApp extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Key Demo',
      theme: ThemeData(primarySwatch: Colors.teal),
      home: Home(),
    );
  }
}

class Home extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => _HomeState();
}

class _HomeState extends State<Home> {
  final GlobalKey<_SwitcherState> _globalKey =
GlobalKey<_SwitcherState>();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("GlobalKey Demo"),
      ),
      body: Switcher(key: _globalKey),
      floatingActionButton: FloatingActionButton(
        onPressed: () =>
_globalKey.currentState.changeState(),
      ),
    );
  }
}

class Switcher extends StatefulWidget {
```

```
  Switcher({Key key}) : super(key: key);

  @override
  State<StatefulWidget> createState() => _SwitcherState();
}

class _SwitcherState extends State<Switcher> {
  bool _isActive;

  @override
  void initState() {
    super.initState();
    _isActive = false;
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Switch.adaptive(
        value: _isActive,
        onChanged: (value) {
          _isActive = value;
          setState(() {});
        },
      ),
    );
  }

  changeState() {
    _isActive = !_isActive;
    setState(() {});
  }
}
```

body 处有个 Switch 控件。

floatingActionButton 有一个 FAB 按钮。
给 Switcher 设置了一个 GlobalKey，然后再 FAB 按钮里面就能用这个 GlobalKey 访问 Switcher 的 State 了，通过 Switcher 的 State 来控制 Switcher 的开关。