Flutter（五）有状态的StatefulWidget

AlanGe

一. StatefulWidget

在开发中，某些Widget情况下我们展示的数据并不是一层不变的：

比如Flutter默认程序中的计数器案例，点击了+号按钮后，显示的数字需要+1；

比如在开发中，我们会进行下拉刷新、上拉加载更多，这时数据也会发生变化；

而StatelessWidget通常用来展示哪些数据固定不变的，如果数据会发生改变，我们使用StatefulWidget；

1.1. 认识StatefulWidget

1.1.1. StatefulWidget介绍

如果你有阅读过默认我们创建Flutter的示例程序，那么你会发现它创建的是一个StatefulWidget。

为什么选择StatefulWidget呢?

- 因为在示例代码中，当我们点击按钮时，界面上显示的数据会发生改变；

- 这时，我们需要一个变量来记录当前的状态，再把这个变量显示到某个Text Widget上；

- 并且每次变量发生改变时，我们对应的Text上显示的内容也要发生改变；

但是有一个问题，我之前说过定义到Widget中的数据都是不可变的，必须定义为final，为什么呢?

- 这次因为Flutter在设计的时候就决定了一旦Widget中展示的数据发生变化，就重新构建整个Widget；

- 下一个章节我会讲解Flutter的渲染原理，Flutter通过一些机制来限定定义到Widget中的成员变量必须是final的；

Flutter如何做到我们在开发中定义到Widget中的数据一定是final的呢?

我们来看一下Widget的源码：

```
@immutable
abstract class Widget extends DiagnosticableTree {
    // ...省略代码
}
```

这里有一个很关键的东西@immutable

- 我们似乎在Dart中没有见过这种语法，这实际上是一个

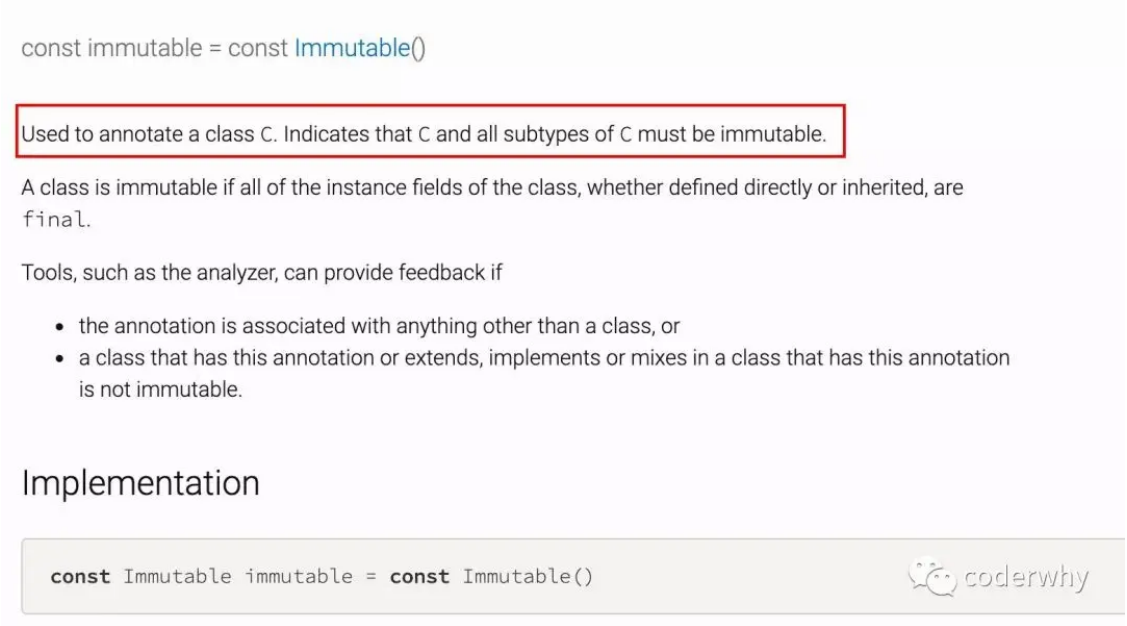
注解，这涉及到Dart的元编程，我们这里不展开讲；

- 这里我就说明一下这个@immutable是干什么的；

实际上官方有对@immutable进行说明：

- **来源：** https://api.flutter.dev/flutter/meta/immutable-constant.html

- **说明：** 被@immutable注解标明的类或者子类都必须是不可变的



图片

结论： 定义到Widget中的数据一定是不可变的，需要使用final来修饰

1.1.2. 如何存储Widget状态?

既然Widget是不可变，那么StatefulWidget如何来存储可变的状态呢?

- StatelessWidget无所谓，因为它里面的数据通常是直接定义完后就不修改的。

- 但StatefulWidget需要有状态（可以理解成变量）的改变，这如何做到呢?

Flutter将StatefulWidget设计成了两个类：

- 也就是你创建StatefulWidget时必须创建两个类：

- 一个类继承自StatefulWidget，作为Widget树的一部分；

- 一个类继承自State，用于记录StatefulWidget会变化的状态，并且根据状态的变化，构建出新的Widget；

创建一个StatefulWidget，我们通常会按照如下格式来做：

- 当Flutter在构建Widget Tree时，会获取`State的实例`，并且它调用build方法去获取StatefulWidget希望构建的Widget；

- 那么，我们就可以将需要保存的状态保存在MyState

中，因为它是可变的；

```
class MyStatefulWidget extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    // 将创建的State返回
    return MyState();
  }
}

class MyState extends State<MyStatefulWidget> {
  @override
  Widget build(BuildContext context) {
    return <构建自己的Widget>;
  }
}
```

思考：为什么Flutter要这样设计呢?

这是因为在Flutter中，只要数据改变了Widget就需要重新构建（rebuild）

1.2. StatefulWidget案例

1.2.1. 案例效果和分析

我们通过一个案例来练习一下StatefulWidget，还是之前的计数器案例，但是我们按照自己的方式进行一些改进。

案例效果以及布局如下：

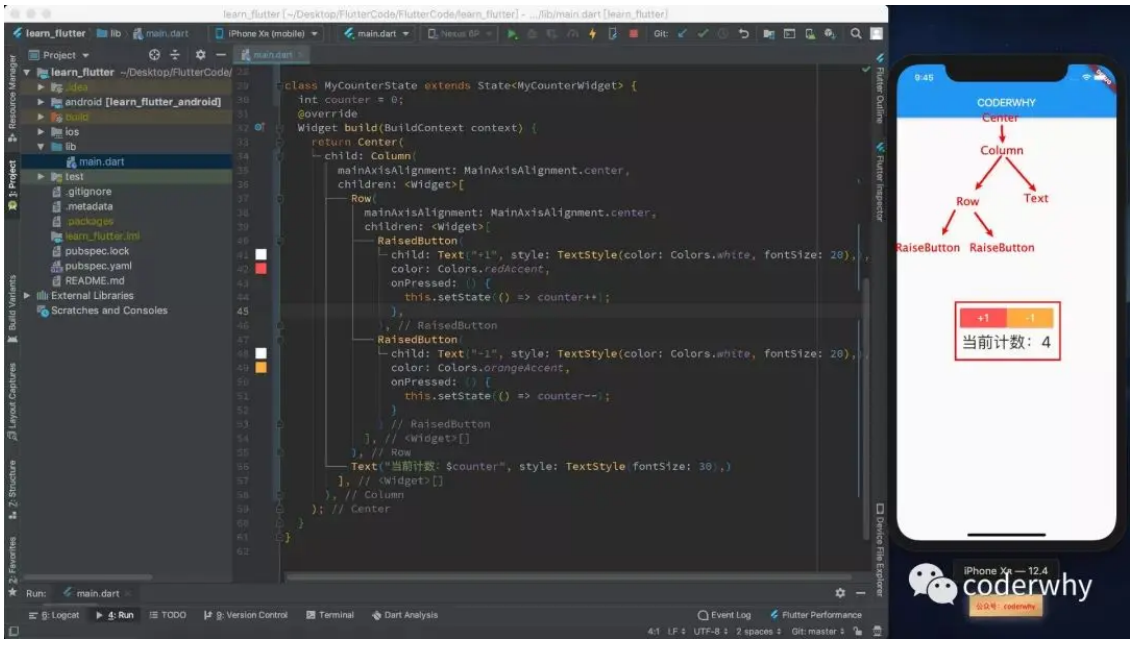
- 在这个案例中，有很多布局对于我们来说有些复杂，我

们后面会详细学习，建议大家根据我的代码一步步写出来来熟悉Flutter开发模式；

- Column小部件：之前我们已经用过，当有垂直方向布局时，我们就使用它；

- Row小部件：之前也用过，当时水平方向布局时，我们就使用它；

- RaiseButton小部件：可以创建一个按钮，并且其中有一个`onPress属性`是传入一个`回调函数`，当按钮点击时被回调；



1.2.2. 创建StatefulWidget

下面我们来看看代码实现：

- 因为当点击按钮时，数字会发生变化，所以我们需要使用一个 StatefulWidget，所以我们需要创建两个类；

- MyCounterWidget 继承自 StatefulWidget，里面需要实现 createState 方法；

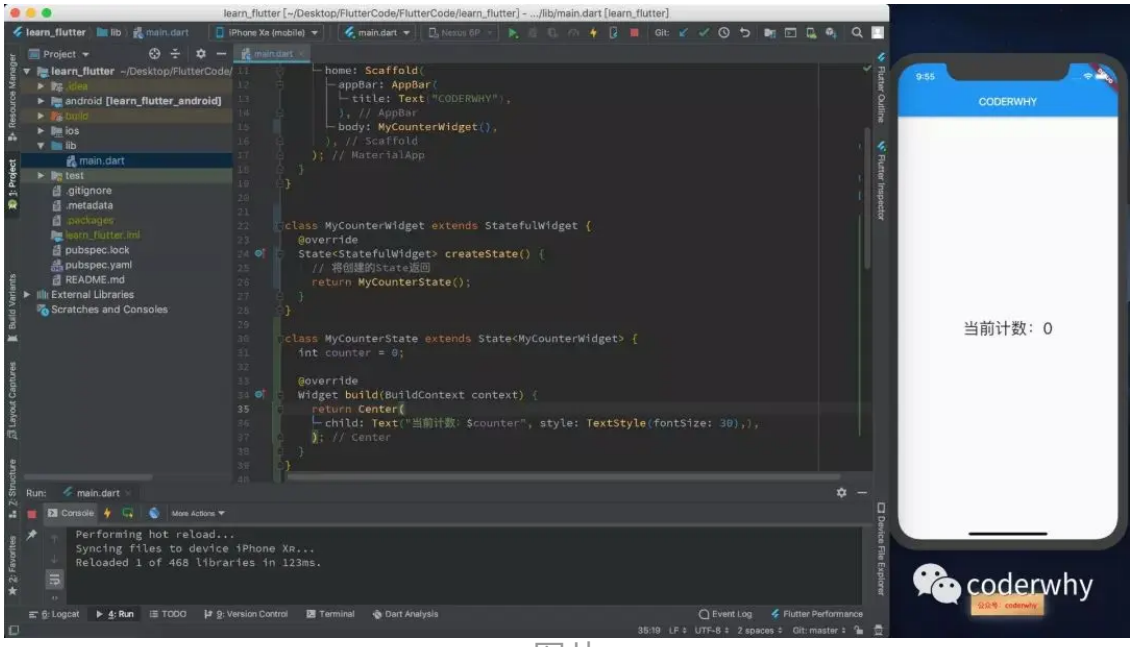
- MyCounterState 继承自 State，里面实现 build 方法，并且可以定义一些成员变量；

```
class MyCounterWidget extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    // 将创建的State返回
    return MyCounterState();
  }
}

class MyCounterState extends State<MyCounterWidget> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text("当前计数: $counter", style:
TextStyle(fontSize: 30),),
```

```
    );
  }
}
```



图片

1.2.3. 实现按钮的布局

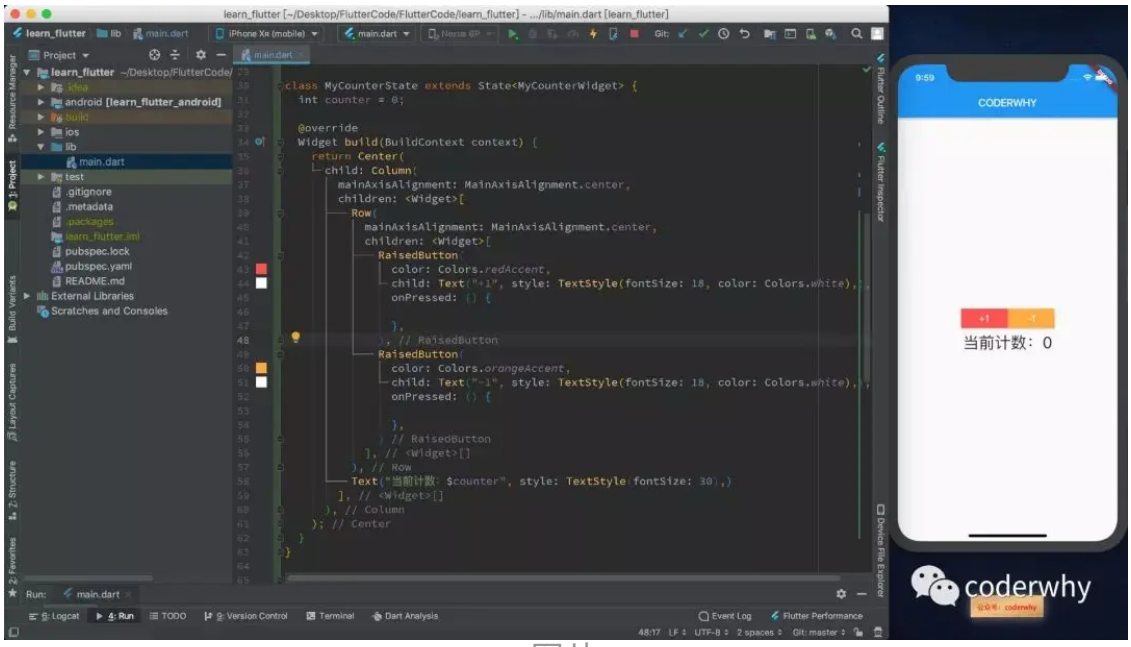
```
class MyCounterState extends State<MyCounterWidget> {
  int counter = 0;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Row(
```

```
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              RaisedButton(
                color: Colors.redAccent,
                child: Text("+1", style:
TextStyle(fontSize: 18, color: Colors.white),),
                onPressed: () {

                },
              ),
              RaisedButton(
                color: Colors.orangeAccent,
                child: Text("-1", style:
TextStyle(fontSize: 18, color: Colors.white),),
                onPressed: () {

                },
              )
            ],
          ),
          Text("当前计数：$counter", style:
TextStyle(fontSize: 30),)
        ],
      ),
    );
  }
}
```



图片

1.2.4. 按钮点击状态改变

我们现在要监听状态的改变，当状态改变时要修改`counter变量`：

- 但是，直接修改变量可以改变界面吗？不可以。

- 这是因为Flutter并不知道我们的数据发生了改变，需要来重新构建我们界面中的Widget；

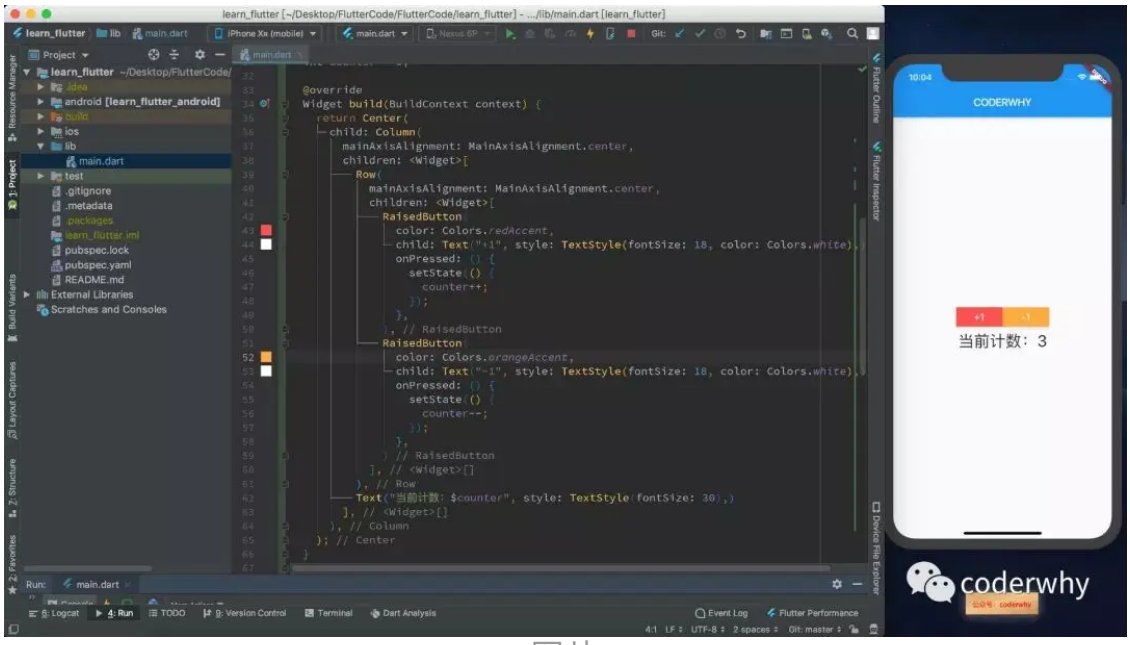
如何可以让Flutter知道我们的状态发生改变了，重新构建我们的Widget呢？

- 我们需要调用一个State中默认给我们提供的setState方法；

- 可以在其中的回调函数中修改我们的变量；

```
onPressed: () {
  setState(() {
    counter++;
  });
},
```

这样就可以实现想要的效果了：



图片

1.3. StatefulWidget生命周期

1.3.1. 生命周期的理解

什么是生命周期呢?

- 客户端开发：iOS开发中我们需要知道UIViewController从创建到销毁的整个过程，Android开

发中我们需要知道 Activity 从创建到销毁的整个过程。以便在不同的生命周期方法中完成不同的操作；

- 前端开发中：Vue、React 开发中组件也都有自己的生命周期，在不同的生命周期中我们可以做不同的操作；

Flutter 小部件的生命周期：

- StatelessWidget 可以由父 Widget 直接传入值，调用 build 方法来构建，整个过程非常简单；

- 而 StatefulWidget 需要通过 State 来管理其数据，并且还要监控状态的改变决定是否重新 build 整个 Widget；

- 所以，我们主要讨论 StatefulWidget 的生命周期，也就是它从创建到销毁的整个过程；

1.3.2. 生命周期的简单版

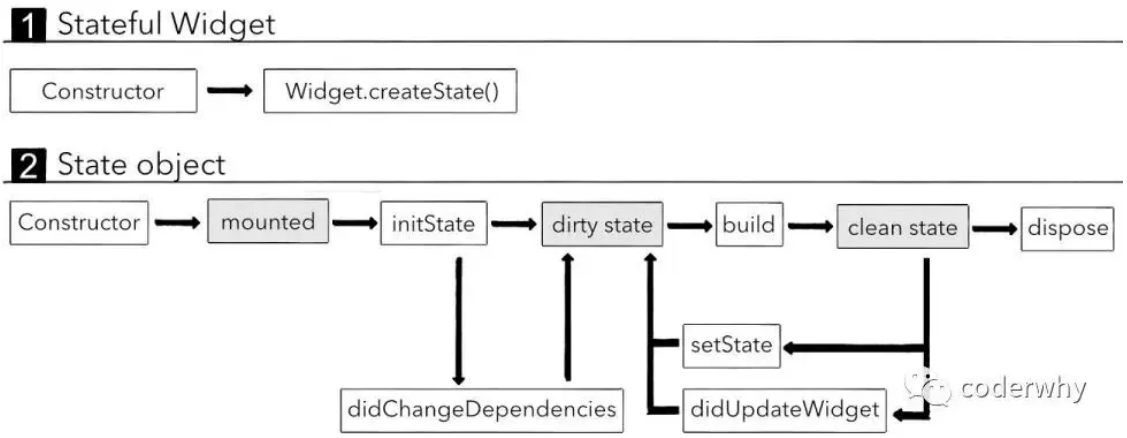
在这个版本中，我讲解那些常用的方法和回调，下一个版本中我解释一些比较复杂的方法和回调

那么 StatefulWidget 有哪些生命周期的回调呢？它们分别在什么情况下执行呢?

- 在下图中，灰色部分的内容是 Flutter 内部操作的，我们并不需要手动去设置它们；

- 白色部分表示我们可以去监听到或者可以手动调用的方法；

我们知道StatefulWidget本身由两个类组成的：`StatefulWidget`和`State`，我们分开进行分析



图片

首先，执行**StatefulWidget**中相关的方法：

- 1、执行StatefulWidget的构造函数（Constructor）来创建出StatefulWidget；

- 2、执行StatefulWidget的createState方法，来创建一个维护StatefulWidget的State对象；

其次，调用createState创建State对象时，执行State类的相关方法：

- 1、执行State类的构造方法（Constructor）来创建State对象；

- 2、执行 initState，我们通常会在这个方法中执行一些数据初始化的操作，或者也可能会发送网络请求；

```
@protected
@mustCallSuper
void initState() {
  assert(_debugLifecycleState == _StateLifecycle.created
}
```


- 注意：这个方法是重写父类的方法，必须调用 super，因为父类中会进行一些其他操作；

- 并且如果你阅读源码，你会发现这里有一个注解（annotation）：@mustCallSuper

- 3、执行 didChangeDependencies 方法，这个方法在两种情况下会调用

- 情况一：调用 initState 会调用；

- 情况二：从其他对象中依赖一些数据发生改变时，比如前面我们提到的 InheritedWidget（这个后面会讲到）；

- 4、Flutter 执行 build 方法，来看一下我们当前的 Widget 需要渲染哪些 Widget；

- 5、当前的 Widget 不再使用时，会调用 dispose 进行销毁；

- 6、手动调用 setState 方法，会根据最新的状态（数据）来重新调用 build 方法，构建对应的 Widgets；

- 7、执行 didUpdateWidget 方法是在当父 Widget 触发重建（rebuild）时，系统会调用 didUpdateWidget 方法；

我们来通过代码进行演示：

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

main(List<String> args) {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("HelloWorld"),
        ),
```

```
        body: HomeBody(),
      ),
    );
  }
}


class HomeBody extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    print("HomeBody build");
    return MyCounterWidget();
  }
}


class MyCounterWidget extends StatefulWidget {

  MyCounterWidget() {
    print("执行了MyCounterWidget的构造方法");
  }

  @override
  State<StatefulWidget> createState() {
    print("执行了MyCounterWidget的createState方法");
    // 将创建的State返回
    return MyCounterState();
```

```
  }
}

class MyCounterState extends State<MyCounterWidget> {
  int counter = 0;

  MyCounterState() {
    print("执行MyCounterState的构造方法");
  }

  @override
  void initState() {
    super.initState();
    print("执行MyCounterState的init方法");
  }

  @override
  void didChangeDependencies() {
    // TODO: implement didChangeDependencies
    super.didChangeDependencies();
    print("执行MyCounterState的didChangeDependencies方法");
  }

  @override
  Widget build(BuildContext context) {
    print("执行执行MyCounterState的build方法");
    return Center(
```

```
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              RaisedButton(
                color: Colors.redAccent,
                child: Text("+1", style:
TextStyle(fontSize: 18, color: Colors.white),),
                onPressed: () {
                  setState(() {
                    counter++;
                  });
                },
              ),
              RaisedButton(
                color: Colors.orangeAccent,
                child: Text("-1", style:
TextStyle(fontSize: 18, color: Colors.white),),
                onPressed: () {
                  setState(() {
                    counter--;
                  });
                },
              )
            ],
          ),
```

```
          Text("当前计数：$counter", style:
TextStyle(fontSize: 30),)
        ],
      ),
    );
  }

  @override
  void didUpdateWidget(MyCounterWidget oldWidget) {
    super.didUpdateWidget(oldWidget);
    print("执行MyCounterState的didUpdateWidget方法");
  }

  @override
  void dispose() {
    super.dispose();
    print("执行MyCounterState的dispose方法");
  }
}
```

打印结果如下：

```
flutter: HomeBody build
flutter: 执行了MyCounterWidget的构造方法
flutter: 执行了MyCounterWidget的createState方法
flutter: 执行MyCounterState的构造方法
flutter: 执行MyCounterState的init方法
```

```
flutter: 执行MyCounterState的didChangeDependencies方法
flutter: 执行执行MyCounterState的build方法

// 注意：Flutter会build所有的组件两次（查了GitHub、Stack Overflow，目前没查到原因）
flutter: HomeBody build
flutter: 执行了MyCounterWidget的构造方法
flutter: 执行MyCounterState的didUpdateWidget方法
flutter: 执行执行MyCounterState的build方法
```

当我们改变状态，手动执行 setState 方法后会打印如下结果：

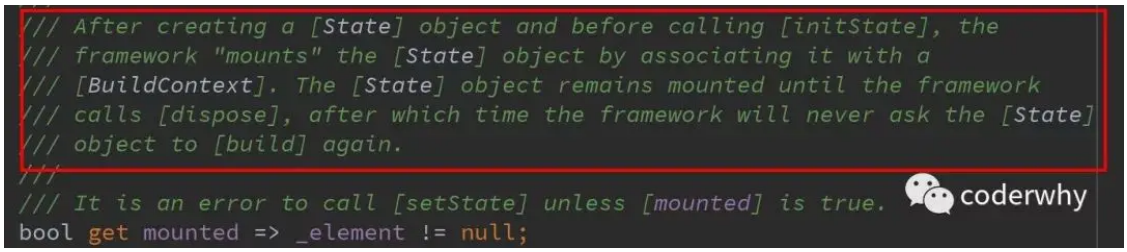
```
flutter: 执行执行MyCounterState的build方法
```

1.3.3. 生命周期的复杂版（选读）

我们来学习几个前面生命周期图中提到的属性，但是没有详细讲解的

1、mounted 是 State 内部设置的一个属性，事实上我们不了解它也可以，但是如果你想深入了解它，会对 State 的机制理解更加清晰；

- 很多资料没有提到这个属性，但是我这里把它列出来，是内部设置的，不需要我们手动进行修改；



2、dirty state的含义是脏的State

- 它实际是通过一个Element的属性来标记的；

- 将它标记为dirty会等待下一次的重绘检查，强制调用build方法来构建我们的Widget；

3、clean state的含义是干净的State

- 它表示当前build出来的Widget，下一次重绘检查时不需要重新build；

二. Flutter的编程范式

这个章节又讲解一些理论的东西，可能并不会直接讲授Flutter的知识，但是会对你以后写任何的代码，都具备一些简单的知道思想；

2.1. 编程范式的理解

编程范式对于初学编程的人来说是一个虚无缥缈的东西，但是却是我们日常开发中都在默认遵循的**一些模式和方法**；

比如我们最为熟悉的 面向对象编程就是一种编程范式，与之对应或者结合开发的包括：面向过程编程、函数式编程、面向协议编程；

另外还有两个对应的编程范式：命令式编程 和 声明式编程

- **命令式编程：** 命令式编程非常好理解，就是一步步给计算机命令，告诉它我们想做什么事情；

- **声明式编程：** 声明式编程通常是描述目标的性质，你应该是什么样的，依赖哪些状态，并且当依赖的状态发生改变时，我们通过某些方式通知目标作出相应；

上面的描述还是太笼统了，我们来看一些具体点的例子；

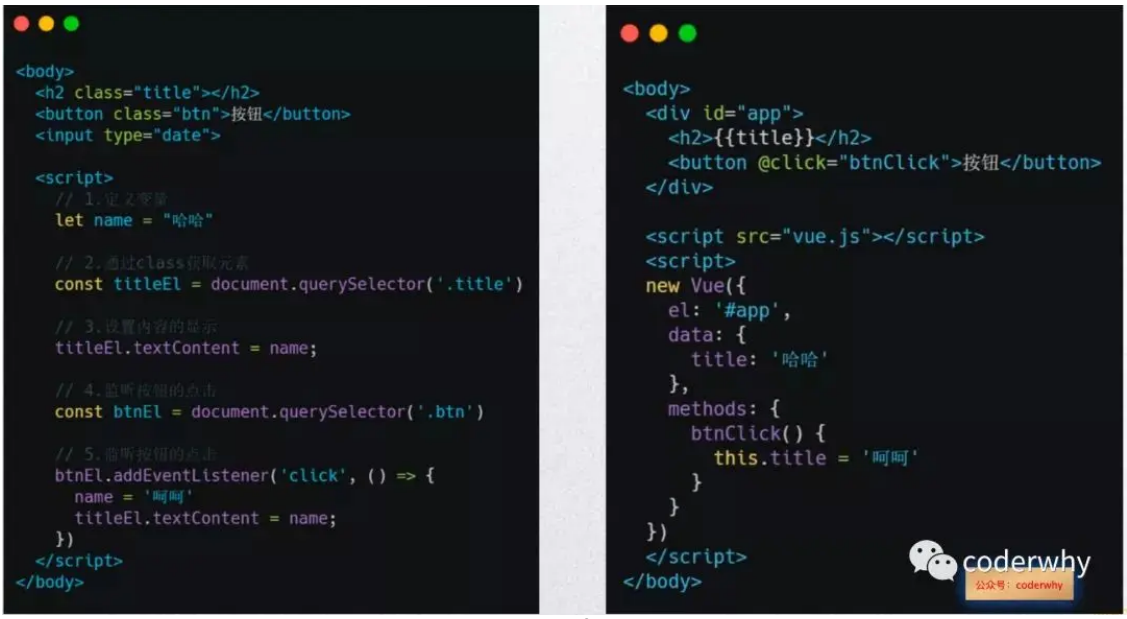
2.2. 前端的编程范式

下面的代码没有写过前端的可以简单看一下

下面的代码是在前端开发中我写的两个 demo，作用都是点击按钮后修改 h2 标签的内容：

- **左边代码：** 命令式编程，一步步告诉浏览器我要做什么事情；

- **右边代码：** 声明式编程，我只是告诉 h2 标签中我需要显示 title，当 title 发生改变的时候，通过一些机制自动来更新状态；



图片

2.3. Flutter的编程范式

从2009年开始（数据来自维基百科），声明式编程就开始流行起来，并且目前在Vue、React、包括iOS中的SwiftUI中以及Flutter目前都采用了声明式编程。

现在我们来开发一个需求：显示一个Hello World，之后又修改成了Hello Flutter

如果是传统的命令式编程，我们开发Flutter的模式很可能是这样的：（注意是想象中的伪代码）

- 整个过程，我们需要一步步告诉Flutter它需要做什么；

```
final text = new Text();
var title = "Hello World";
text.setContent(title);


// 修改数据
```

```
title = "Hello Flutter";
text.setContent(title);
```

如果是声明式编程，我们通常会维护一套数据集：

- 这个数据集可能来自己父类、来自自身 State 管理、来自 InheritedWidget、来自统一的状态管理的地方；

- 总之，我们知道有这么一个数据集，并且告诉 Flutter 这些数据集在哪里使用；

```
var title = "Hello World";

Text(title); // 告诉Text内部显示的是title

// 数据改变
title = "Hello Flutter";
setState(() => null); // 通知重新build Widget即可
```

上面的代码过于简单，可能不能体现出 Flutter 声明式编程的优势所在，但是在以后的开发中，我们都是按照这种模式在进行开始，我们一起来慢慢体会；

响应式编程

参考：小码哥 Flutter