

StatefulWidget 及 State

StatefulWidget 是 UI 可以变化的 Widget。

StatefulWidget 的实现

下面是一段实现 StatefulWidget 的 Demo 代码，将下面代码复制到 main.dart 里并运行：

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

void main() => runApp(MyApp("Hello World"));

class MyApp extends StatefulWidget {
  // This widget is the root of your application.

  String content;

  MyApp(this.content);

  @override
  State<StatefulWidget> createState() {
    // TODO: implement createState
    return MyAppState();
  }
}

class MyAppState extends State<MyApp> {

  bool isShowText =true;
```

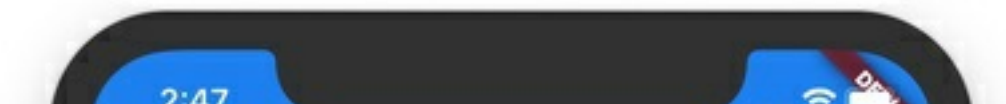
```

void increment(){
  setState(() {
    widget.content += "d";
  });
}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: Scaffold(
      appBar: AppBar(title: Text("Widget --
StatefulWidget及State")),
      body: Center(
        child: GestureDetector(
          child: isShowText?
Text(widget.content) :null,
          onTap: increment,
        ),
      ),
    ),
  );
}
}

```

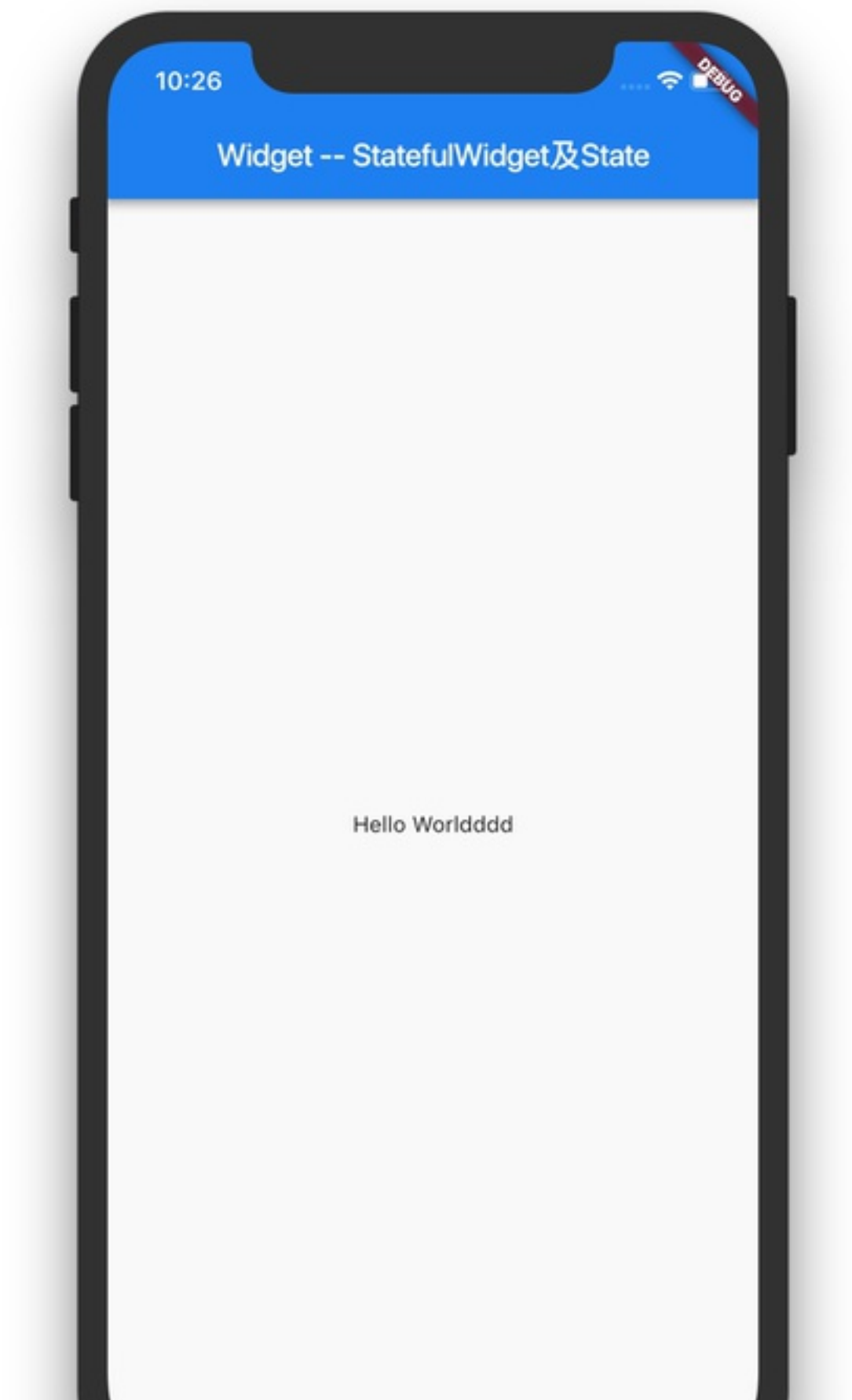
这里的 MyApp 就是一个 StatefulWidget，当点击 Hello World 的文本框，内容会变。刚开始运行的效果为：



Widget -- StatefulWidget及State

Hello World

点击点击 Hello World 的文本框几次后，界面就会变为：





看下 MyApp 的实现代码：

```
class MyApp extends StatefulWidget {  
  // This widget is the root of your application.  
  
  @override  
  State<StatefulWidget> createState() {  
    // TODO: implement createState  
    return MyAppState("Hello World");  
  }  
}  
  
class MyAppState extends State<MyApp> {  
  
  @override  
  Widget build(BuildContext context) {  
    return ...  
  }  
}
```

可以看到实现 StatefulWidget，需要两部分组成：

1. StatefulWidget
2. State

1. StatefulWidget

```
class MyApp extends StatefulWidget {  
  // This widget is the root of your application.  
  
  String content;  
  
  MyApp(this.content);  
  
  @override  
  State<StatefulWidget> createState() {  
    // TODO: implement createState  
    return MyAppState();  
  }  
}
```

StatefulWidget 实现步骤：

1. 首先继承 StatefulWidget
2. 实现 createState() 的方法，返回一个 State

StatefulWidget 的功能

StatefulWidget 的主要功能就是创建 State。

2. State

State 即是状态。

```
class MyAppState extends State<MyApp> {  
  
    void increment(){  
        setState(() {  
            widget.content += "d";  
        });  
    }  
  
    @override  
    Widget build(BuildContext context) {  
        return ...  
    }  
}
```

State 的实现步骤：

1. 首先继承 State，State 的泛型类型是上面定义的 Widget 的类型
2. 实现 build() 的方法，返回一个 Widget
3. 需要更改数据，刷新 UI 的话，调用 setState()

State 的定义

State 用到了泛型，它的定义是这样子的：

```
State<T extends StatefulWidget>
```

State 的功能

State 有两个功能：

1. build() —— 创建 Widget
2. setState() —— 刷新 UI

1. build() —— 创建Widget

State 的 build() 函数创建 Widget，用于显示 UI。

2. setState() —— 更新状态，刷新 UI

调用 setState() 方法，在 setState() 里更改数据的值，然后 setState() 会触发 State 的 build() 方法，引起强制重建 Widget，重建 Widget 的时候会重新绑定数据，而这时数据已经发生变化，从而达到刷新 UI 的目的。

setState() 在 State 里很重要，接下来在单独讲一下 setState() 的使用。

首先看一下，setState() 在源码里的定义如下：

```
@protected
void setState(VoidCallback fn) {
    ...
}
```

setState() 里要传入一个无参的函数，所以使用方法如下：

```
setState(() {
    widget.content += "d";
});
```

在无参函数内部，对要刷新的数据进行更改。

我们可以看一下 setState() 的源码，去掉没有必要的代码，就是：


```
@protected
void setState(VoidCallback fn) {
    final dynamic result = fn() as dynamic;
    _element.markNeedsBuild();
}
```

第一行代码，执行无参函数 `fn()`，并把结果类型转换为 `dynamic`，并赋值给 `result`。

第二行代码会触发 `Widget` 创建。

这里要注意，更改数据的代码必选在 `setState()` 之前写，或者在 `setState()` 内的无参函数里写，才能刷新数据，否则是没有用的。

这里还有一个问题，`Text` 是 `MyApp` 的子 `Widget`，但 `Text` 是 `StatelessWidget`，为什么 `Text` 的内容可以改变？

`setState()` 可以刷新UI的原理是，`setState()` 会触发 `StatefulWidget` 强制重建，重建的时候会重新创建 `Widget` 和绑定数据，从而实现了刷新 UI。所以只要 `MyApp` 是 `StatefulWidget`，那么它的子类在 `setState()` 的作用下都可以被强制刷新。

State 的成员变量

`State` 里面有三个重要的成员变量：

1. `widget`
2. `context`
3. `mounted`

1. widget

widget 是 State 的成员变量，它的类型是 Widget，前面的代码里你可能注意到了，有这种使用用法：

```
child: Text(widget.content)
```

widget 可以访问 StatefulWidget 中的成员变量。

2. context

context 也是 State 的成员变量，它的类型是 BuildContext，它的一种用法如下：

```
Widget build(BuildContext context)
```

BuildContext 是 Flutter 里的重要概念。

3. mounted

mounted 是 bool 类型，表示当前 State 是否加载到树里。State 对象创建之后，initState() 创建之前，framework 通过与 BuildContext 相关联，来将 State 对象加载到树中，此时 mounted 会变为 true，当 State dispose 之后，mounted 就变为 false。

mounted 属性很有用，因为 setState() 只有在 mounted 为 true 的时候才能用，当 mounted 为 false 时调用会抛异常。

因为 State 的状态比较复杂，如果 setState() 使用不注意，很容易抛异常，所以保险起见，mounted 一般这么用：

```
if(mounted){  
    setState((){  
        ...  
    })  
}
```

只有在确定 State mounted 之后，才调用 setState()。

为什么 StatefulWidget 被分成 StatefulWidget 和 State 两部分？

一方面是为了保存当前 APP 的状态，另一个重要的原因就是为了性能！

当 UI 需要更新时候，假设 Widget 和 State 都重建，可是 State 里保存了 UI 显示的数据，State 重建，创建新的实例，UI 之前的状态就会丢失，导致 UI 显示异常，所以要分成两部分，一部分会重建，一部分不会重建，重建的部分就是 StatefulWidget，不会重建的部分就是 State。

Widget 重建的成本很低，但 State 的重建成本很高，因此将 StatefulWidget 分成两部分：重建成本低的 Widget 和重建成本高的 State。这样就使得 State 不会被频繁重建，也就提高了性能。

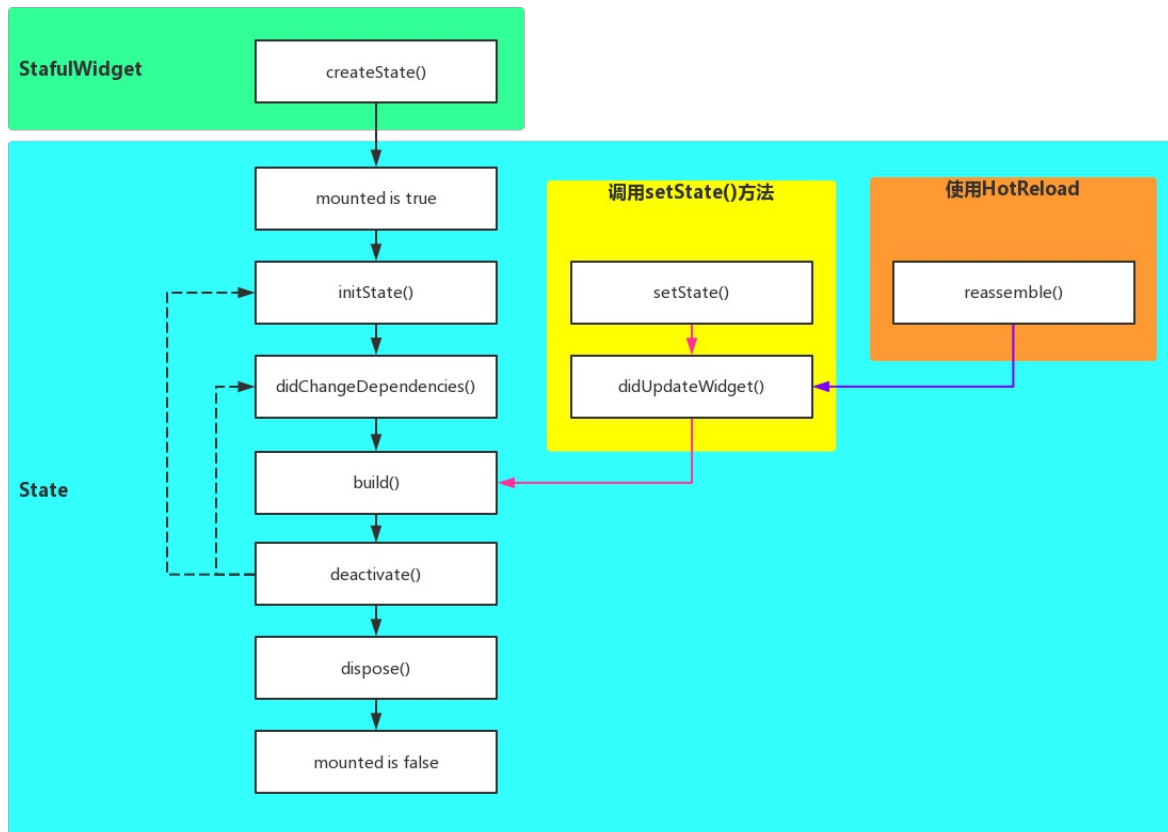
StatefulWidget 的重新定义

StatefulWidget 是有 State(状态) 的Widget，当 Widget 在运行时需要改变时，就要用 StatefulWidget。

StatefulWidget 的生命周期

因为 StatefulWidget 由 StatefulWidget 和 State 两部分组成，所以也有 StatefulWidget 的生命周期和 State 生命周期。

StatefulWidget 的生命周期如下：



StatefulWidget的生命周期

StatefulWidget 的生命周期很简单，只有一个，即 `createState` 函数：

- `createState` (`createState`函数)

State 的生命周期

- `mounted is true`

`mounted` 是 `boolean`，只有当 `mounted` 为 `true` 时，才能使用 `setState()`。

- `initState`

`initState()` 方法是在创建 `State` 对象后要调用的第一个方法（在构造函数之后）。

一旦 `initState()` 方法完成，`State` 对象就初始化完成了，`BuildContext` 也可以用了。所以如果你要用 `BuildContext`，那么需要在 `initState()` 之后的生命周期里用到。

可以在这里执行其他的初始化，例如执行依赖于 `BuildContext` 或 `Widget` 的初始化，或者 `animations`、`controllers` 等动画相关的初始化。

如果你要重写此方法，需要首先调用 `super.initState()` 方法。

- `didChangeDependencies`

`initState()` 方法运行完后，就立即运行 `didChangeDependencies()` 方法。

当 `Widget` 依赖的数据被调用时，此方法也会被调用。

此外，请注意，如果您的 `Widget` 链接到 `InheritedWidget`，则每次重建此窗口小部件时都会调用此方法。

如果重写此方法，则应首先调用 `super.didChangeDependencies()`。

- `build`

`build()` 方法在 `didChangeDependencies()`（或者 `didUpdateWidget()`）之后调用。这是构建 `Widget` 的地方。

每次 State 对象更改时（或者当 InheritedWidget 需要通知“已注册”的小部件时）都会调用此方法！

为了强制重建，需要调用 setState() 方法。

至此，一个 Widget 从创建到显示的声明周期就完成了，如果在对应的方法里加上 log，会看到如下的 log 输出：

```
Launching lib/main.dart on iPhone XR in debug mode...
Xcode build done.
3.9s
flutter: initState
flutter: didChangeDependencies
flutter: build
```

- setState()

当状态有变化，需要刷新UI的时候，就调用 setState()，会触发强制重建 Widget。

- didUpdateWidget()

当 Widget 重建后，新的 Widget 会和旧的 Widget 进行对比，如果新的 Widget 和旧的 Widget 的 runtimeType 和 Widget.key 都一样，那么就会调用 didUpdateWidget()。

在 didUpdateWidget() 里，会把新的 Widget 的配置赋值给 State，相当于重新 initState() 了一次。

调用完这个方法之后，再去调用 build() 方法。

至此 setState() 的生命周期也完成了，会看到如下的 log 输出：

```
didUpdateWidget  
build
```

- **deactive**

当 State 从树中移除时，就会触发 **deactive**。但是如果在这帧结束前，如果有其他地方使用到了这个Widget，就会重新把Widget 插入到树里，这就涉及到了 Widget 的重用，Widget 的重用和 Key 有关。

这里使用不同的方法重用，会有不同的生命周期，所以这里使用的是虚线表示的。

- **dispose**

当 StaefulWidget 从树中移除时调用 **dispose()** 方法。

可以在这里执行一些清理逻辑（例如侦听器），重写此方法时，需要首先调用 **super.dispose()**。

至此完成了 Widget 销毁的生命周期，log 输出如下：

```
deactive  
dispose
```

- **mounted is false**

State 对象不能 remounted，所以一旦 **mounted is false**，就不能在使用 **setState()**，会抛异常。

- **State HotReload 的生命周期 -- *reassemble***

在开发期间，执行 HotReload，就会触发 **reassemble()**，这提供了重新初始化在 **initState()** 方法中准备的任何数据的机会，包括全局变量。

前面讲了，全局变量不能用 HotReload，但是可以在 `reassemble()` 里改值，但是并没有卵用，因为这个只会在 Debug 阶段 Hot Reload 的时候触发。

更改 `reassemble()` 里的 `content` 的值，然后执行 Hot Reload，输出：

```
reassemble  
didUpdateWidget  
build
```

State 的生命周期在代码中对应的方法如下：

```
class XXXState extends State<XXX> {  
  
  @override  
  void initState() {  
    // TODO: implement initState  
    super.initState();  
    print("initState");  
    context.runtimeType;  
  }  
  
  @override  
  void didChangeDependencies() {  
    // TODO: implement didChangeDependencies  
    super.didChangeDependencies();  
    print("didChangeDependencies");  
  }  
  
  @override  
  void didUpdateWidget(MyApp oldWidget) {  
    // TODO: implement didUpdateWidget  
    super.didUpdateWidget(oldWidget);  
  }  
}
```



```
    print("didUpdateWidget");
}

@override
Widget build(BuildContext context) {
    print("build");
    return ...
}

@override
void dispose() {
    // TODO: implement dispose
    super.dispose();
    print("dispose");
}

@override
void reassemble() {
    // TODO: implement reassemble
    super.reassemble();
    print("reassemble");
}
}
```

总结

StatefulWidget 里 Widget 这部分的功能是：

- 创建 State

StatefulWidget 里 State 这部分的功能是：

- 创建 Widget
- 更新状态，刷新 UI