一、StatefulWidget简介

StatefulWidget是UI可以变化的Widget

代码范例实现如下：

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

void main() => runApp(TestApp ("Hello World"));
class TestApp extends StatefulWidget {
 String content;

 TestApp(this.content);

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

class _TestState extends State<TestApp> {
  bool isShowText = true;

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

  @override
  Widget build(BuildContext context) {
  // TODO: implement build
  return new MaterialApp(
  title: "Flutter App",
  home: Scaffold(
  appBar: AppBar(
  title: Text("StatefulWidget"),
  ),
  body: Center(
  child: GestureDetector(
  child: isShowText ? Text(widget.content) : null,
  onTap: increment,
  ),
  ),
```

```
40   ),
41   );
42   }
43  }
```

从代码逻辑来看，StatefulWidget分为两部分：

- StatefulWidget
- State

而显示组件以及方法逻辑都在State中。

1、StatefulWidget

StatefuWidget的主要功能是创建State

```
1  class TestApp extends StatefulWidget {
2   String content;
3
4   TestApp(this.content);
5
6   @override
7   State<StatefulWidget> createState() {
8   // TODO: implement createState
9   return _TestState();
10   }
11  }
```

定义一个StatefulWidget需要完成两部：

- 继承StatefulWidget

```
1  class TestApp extends StatefulWidget
```

- 实现createState()方法

```
1   @override
2   State<StatefulWidget> createState() {
3   // TODO: implement createState
4   return _TestState();
5   }
```

该方法的返回值是State<StatefulWidget>，因此该方法需要返回我们定义的State

2、State

State就是组件的状态

```
1  class _TestState extends State<TestApp> {
2   bool isShowText = true;
3
```

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

   @override
   Widget build(BuildContext context) {
   // TODO: implement build
   return new MaterialApp(
   title: "Flutter App",
   home: Scaffold(
   appBar: AppBar(
   title: Text("StatefulWidget"),
   ),
   body: Center(
   child: GestureDetector(
   child: isShowText ? Text(widget.content) : null,
   onTap: increment,
   ),
   ),
   ),
   );
   }
}
```

定义State需要完成三步：

- 继承State，State中的泛型就是我们之前定义的StatefulWidget组件

```
class _TestState extends State<TestApp>
```

- 实现build方法

```
@override
 Widget build(BuildContext context) {
 // TODO: implement build
 return new MaterialApp(
 title: "Flutter App",
 home: Scaffold(
 appBar: AppBar(
 title: Text("StatefulWidget"),
 ),
  body: Center(
```

```
child: GestureDetector(
child: isShowText ? Text(widget.content) : null,
onTap: increment,
),
),
),
);
}
```

在该方法中我们定义需要的Widget并返回

- 更新数据后，需要更新UI，调用setState()方法

二、State

1、State的功能

State的功能主要有两部分：

- build()——构建Widget
- setState()——刷新UI

(1) build()

build方法中创建用于显示的Widget

(2) setState()

setState用于更新组件状态，刷新UI，在调用setState后，会触发State的build方法，用于强制重建Widget，重建Widget的时候，会重新绑定数据，此时数据已经改变，从而达到刷新UI的目的

2、seState源码分析

setState方法会触发State的build（）方法，从而引起组件的重建，重建的时候会重新绑定数据，从而刷新UI

setState源码

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

其中VoidCallback是一个无参构造函数，上面我们通过

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

其中

```
() {
```

```
  widget.content += "d";
  }
```

其实就是传入了一个无参构造函数的实现

首先会执行final dynamic result = fn() as dynamic;，其中fn就是我们传入的无参构造函数

执行该函数后会继续执行_element.markNeedsBuild();

该方法的作用是触发Widget的重建，会调用State的build方法

3、State的成员变量

State有是三个成员变量：

- widget
- context
- mounted

（1）widget

widget是State的成员变量，它的类型是Widget，可以通过它来访问我们定义的WidgetfulWidget组件中的成员变量。

例如上面代码中的：

在State中通过widget.content访问StatefulWidget中的content成员变量

```
child: isShowText ? Text(widget.content) : null,
```

（2）context

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

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

（3）mounted

mounted是bool类型，表示当前State是否加载到树立，在State创建后，initState()创建之前，framework通过BuildContext相关联，来将State对象加载到树中，此时mounted为true，当State被dispose只有，mounted被为置为false。

由于State状态比较复杂，如果setState（）使用不注意，很容易抛出异常，因此为了保险起见，mounted一般这么使用

```
if（mounted）{
 setState（）{

 }
}
```

只有确定mounted之后才可以调用setState

四、StatefulWidget的理解

1、为什么StatefulWidget分为StatefulWidget和State两部分

主要原因有两点：

（1）保存当前App的状态

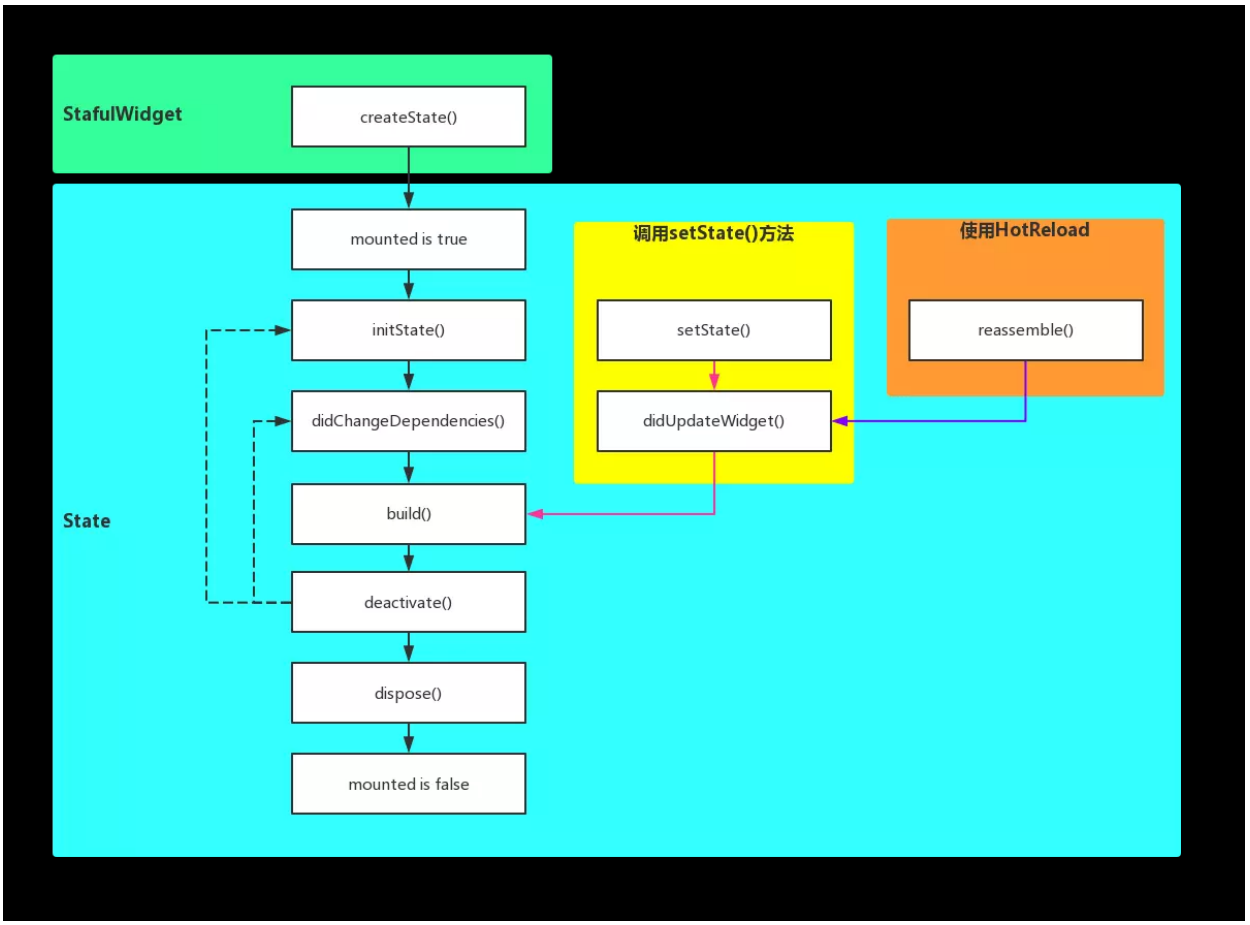
当UI需要更新时，假设Widget和State都重建，而State保存的是UI显示的数据，那么State重建后，这些数据也就丢失了，从而导致UI显示异常。因此分为两部分，State不重建，而StatefulWidget重建，保证重建后UI显示数据正常显示

（2）性能

Widget只是一个组件，相对重建成本比较低，而State保存的组件数据，重建成本高，因此通过只是重建Widget而不重建State，从而减少不必要的的重建，提高性能

2、生命周期

由于StatefulWidget分为两部分，因此生命周期也分为State和StatefulWidget的生命周期



（1）StatefulWidget的生命周期

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

（2）State的生命周期

- mounted is true

在initState执行过之前，首先mounted为置为true，之后才可以执行setState（）方法

- initState()

initState方法创建State对象后调用的第一个方法（在构造方法后执行的第一个方法），也就是说如果要使用BuildContext，那么需要在initState()之后再调用。

重写该方法时需要调用super.initState()

- didChangeDependicies

ininState方法执行后会立即执行didChangeDependicies。

当Widget依赖的数据被调用时，此方法就会被调用

需要注意，当我们定义的Widget连接到 InheritedWidget，则每次重建小窗口都会调用该方法

- build

build方法是在didChangeDependicies或者didUpdateWidget之后调用，主要的作用是构建Widget

每次State对象更改时，都会通过该方法重建组件，从而更新UI

- setState

当状态有变化，需要刷新UI时，就通过该方法来触发重建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key都一样，则会调用didUpdateWidget。

在didUpdateWidget中，会将新的Widget的配置赋值给State，相当于重新执行了一次initState一次。调用完该方法后再去调用build方法。

- deactive

当StatefulWidget从树中移除时，会触发deactive。但是如果在这帧结束以前，有其他地方地方用到了这个Widget，此时会重新将Widget插入到树中，这里就涉及到Widget的重用。

而使用不同的方法重用，则会用不同的生命周期，也就是图上虚线表示的。

虚线中会重新进入initState或者didChangeDependicies

- dispose

当StatefulWidget从树中移除时调用dispose方法。

在这个方法中可以执行一些销毁逻辑，类似于Android中的onDestory

- mounted is false

State对象不能remounted，所以一旦mounted为false，就不能使用setState（），否则会抛出异常

- State的HotReload的生命周期——reassemble

开发期间，如果执行HotReload，会触发reassemble，该方法只有在debug模式中Hot Reload才会触发。调用该方法后执行didUpdateWidget

五、总结

生命周期在学习阶段都是纸上谈兵，实践的时候结合总结