# 3.2 状态管理

响应式的编程框架中都会有一个永恒的主题——“状态(State)管理”，无论是在React/Vue（两者都是支持响应式编程的Web开发框架）还是Flutter中，他们讨论的问题和解决的思想都是一致的。所以，如果你对React/Vue的状态管理有了解，可以跳过本节。言归正传，我们想一个问题，StatefulWidget的状态应该被谁管理？Widget本身？父Widget？都会？还是另一个对象？答案是取决于实际情况！以下是管理状态的最常见的方法：

* Widget管理自己的状态。
* Widget管理子Widget状态。
* 混合管理（父Widget和子Widget都管理状态）。

如何决定使用哪种管理方法？下面是官方给出的一些原则可以帮助你做决定：

* 如果状态是用户数据，如复选框的选中状态、滑块的位置，则该状态最好由父Widget管理。
* 如果状态是有关界面外观效果的，例如颜色、动画，那么状态最好由Widget本身来管理。
* 如果某一个状态是不同Widget共享的则最好由它们共同的父Widget管理。

在Widget内部管理状态封装性会好一些，而在父Widget中管理会比较灵活。有些时候，如果不确定到底该怎么管理状态，那么推荐的首选是在父widget中管理（灵活会显得更重要一些）。

接下来，我们将通过创建三个简单示例TapboxA、TapboxB和TapboxC来说明管理状态的不同方式。 这些例子功能是相似的 ——创建一个盒子，当点击它时，盒子背景会在绿色与灰色之间切换。状态 \_active确定颜色：绿色为true ，灰色为false，如图3-4所示。a large grey box with the text, ‘Inactive’

下面的例子将使用GestureDetector来识别点击事件，关于该GestureDetector的详细内容我们将在后面“事件处理”一章中介绍。

### 3.2.1 Widget管理自身状态

\_TapboxAState 类:

* 管理TapboxA的状态。
* 定义\_active：确定盒子的当前颜色的布尔值。
* 定义\_handleTap()函数，该函数在点击该盒子时更新\_active，并调用setState()更新UI。
* 实现widget的所有交互式行为。

// TapboxA 管理自身状态.  
  
//------------------------- TapboxA ----------------------------------  
  
class TapboxA extends StatefulWidget {  
 TapboxA({Key key}) : super(key: key);  
  
 @override  
 \_TapboxAState createState() => new \_TapboxAState();  
}  
  
class \_TapboxAState extends State<TapboxA> {  
 bool \_active = false;  
  
 void \_handleTap() {  
 setState(() {  
 \_active = !\_active;  
 });  
 }  
  
 Widget build(BuildContext context) {  
 return new GestureDetector(  
 onTap: \_handleTap,  
 child: new Container(  
 child: new Center(  
 child: new Text(  
 \_active ? 'Active' : 'Inactive',  
 style: new TextStyle(fontSize: 32.0, color: Colors.white),  
 ),  
 ),  
 width: 200.0,  
 height: 200.0,  
 decoration: new BoxDecoration(  
 color: \_active ? Colors.lightGreen[700] : Colors.grey[600],  
 ),  
 ),  
 );  
 }  
}

### 3.2.2 父Widget管理子Widget的状态

对于父Widget来说，管理状态并告诉其子Widget何时更新通常是比较好的方式。 例如，IconButton是一个图标按钮，但它是一个无状态的Widget，因为我们认为父Widget需要知道该按钮是否被点击来采取相应的处理。

在以下示例中，TapboxB通过回调将其状态导出到其父组件，状态由父组件管理，因此它的父组件为StatefulWidget。但是由于TapboxB不管理任何状态，所以TapboxB为StatelessWidget。

ParentWidgetState 类:

* 为TapboxB 管理\_active状态。
* 实现\_handleTapboxChanged()，当盒子被点击时调用的方法。
* 当状态改变时，调用setState()更新UI。

TapboxB 类:

* 继承StatelessWidget类，因为所有状态都由其父组件处理。
* 当检测到点击时，它会通知父组件。

// ParentWidget 为 TapboxB 管理状态.  
  
//------------------------ ParentWidget --------------------------------  
  
class ParentWidget extends StatefulWidget {  
 @override  
 \_ParentWidgetState createState() => new \_ParentWidgetState();  
}  
  
class \_ParentWidgetState extends State<ParentWidget> {  
 bool \_active = false;  
  
 void \_handleTapboxChanged(bool newValue) {  
 setState(() {  
 \_active = newValue;  
 });  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Container(  
 child: new TapboxB(  
 active: \_active,  
 onChanged: \_handleTapboxChanged,  
 ),  
 );  
 }  
}  
  
//------------------------- TapboxB ----------------------------------  
  
class TapboxB extends StatelessWidget {  
 TapboxB({Key key, this.active: false, @required this.onChanged})  
 : super(key: key);  
  
 final bool active;  
 final ValueChanged<bool> onChanged;  
  
 void \_handleTap() {  
 onChanged(!active);  
 }  
  
 Widget build(BuildContext context) {  
 return new GestureDetector(  
 onTap: \_handleTap,  
 child: new Container(  
 child: new Center(  
 child: new Text(  
 active ? 'Active' : 'Inactive',  
 style: new TextStyle(fontSize: 32.0, color: Colors.white),  
 ),  
 ),  
 width: 200.0,  
 height: 200.0,  
 decoration: new BoxDecoration(  
 color: active ? Colors.lightGreen[700] : Colors.grey[600],  
 ),  
 ),  
 );  
 }  
}

### 3.2.3 混合状态管理

对于一些组件来说，混合管理的方式会非常有用。在这种情况下，组件自身管理一些内部状态，而父组件管理一些其他外部状态。

在下面TapboxC示例中，手指按下时，盒子的周围会出现一个深绿色的边框，抬起时，边框消失。点击完成后，盒子的颜色改变。 TapboxC将其\_active状态导出到其父组件中，但在内部管理其\_highlight状态。这个例子有两个状态对象\_ParentWidgetState和\_TapboxCState。

\_ParentWidgetStateC类:

* 管理\_active 状态。
* 实现 \_handleTapboxChanged() ，当盒子被点击时调用。
* 当点击盒子并且\_active状态改变时调用setState()更新UI。

\_TapboxCState 对象:

* 管理\_highlight 状态。
* GestureDetector监听所有tap事件。当用户点下时，它添加高亮（深绿色边框）；当用户释放时，会移除高亮。
* 当按下、抬起、或者取消点击时更新\_highlight状态，调用setState()更新UI。
* 当点击时，将状态的改变传递给父组件。

//---------------------------- ParentWidget ----------------------------  
  
class ParentWidgetC extends StatefulWidget {  
 @override  
 \_ParentWidgetCState createState() => new \_ParentWidgetCState();  
}  
  
class \_ParentWidgetCState extends State<ParentWidgetC> {  
 bool \_active = false;  
  
 void \_handleTapboxChanged(bool newValue) {  
 setState(() {  
 \_active = newValue;  
 });  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Container(  
 child: new TapboxC(  
 active: \_active,  
 onChanged: \_handleTapboxChanged,  
 ),  
 );  
 }  
}  
  
//----------------------------- TapboxC ------------------------------  
  
class TapboxC extends StatefulWidget {  
 TapboxC({Key key, this.active: false, @required this.onChanged})  
 : super(key: key);  
  
 final bool active;  
 final ValueChanged<bool> onChanged;  
   
 @override  
 \_TapboxCState createState() => new \_TapboxCState();  
}  
  
class \_TapboxCState extends State<TapboxC> {  
 bool \_highlight = false;  
  
 void \_handleTapDown(TapDownDetails details) {  
 setState(() {  
 \_highlight = true;  
 });  
 }  
  
 void \_handleTapUp(TapUpDetails details) {  
 setState(() {  
 \_highlight = false;  
 });  
 }  
  
 void \_handleTapCancel() {  
 setState(() {  
 \_highlight = false;  
 });  
 }  
  
 void \_handleTap() {  
 widget.onChanged(!widget.active);  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 // 在按下时添加绿色边框，当抬起时，取消高亮   
 return new GestureDetector(  
 onTapDown: \_handleTapDown, // 处理按下事件  
 onTapUp: \_handleTapUp, // 处理抬起事件  
 onTap: \_handleTap,  
 onTapCancel: \_handleTapCancel,  
 child: new Container(  
 child: new Center(  
 child: new Text(widget.active ? 'Active' : 'Inactive',  
 style: new TextStyle(fontSize: 32.0, color: Colors.white)),  
 ),  
 width: 200.0,  
 height: 200.0,  
 decoration: new BoxDecoration(  
 color: widget.active ? Colors.lightGreen[700] : Colors.grey[600],  
 border: \_highlight  
 ? new Border.all(  
 color: Colors.teal[700],  
 width: 10.0,  
 )  
 : null,  
 ),  
 ),  
 );  
 }  
}

另一种实现可能会将高亮状态导出到父组件，但同时保持\_active状态为内部状态，但如果你要将该TapBox给其它人使用，可能没有什么意义。 开发人员只会关心该框是否处于Active状态，而不在乎高亮显示是如何管理的，所以应该让TapBox内部处理这些细节。

### 3.2.4 全局状态管理

当应用中需要一些跨组件（包括跨路由）的状态需要同步时，上面介绍的方法便很难胜任了。比如，我们有一个设置页，里面可以设置应用的语言，我们为了让设置实时生效，我们期望在语言状态发生改变时，APP中依赖应用语言的组件能够重新build一下，但这些依赖应用语言的组件和设置页并不在一起，所以这种情况用上面的方法很难管理。这时，正确的做法是通过一个全局状态管理器来处理这种相距较远的组件之间的通信。目前主要有两种办法：

1. 实现一个全局的事件总线，将语言状态改变对应为一个事件，然后在APP中依赖应用语言的组件的initState 方法中订阅语言改变的事件。当用户在设置页切换语言后，我们发布语言改变事件，而订阅了此事件的组件就会收到通知，收到通知后调用setState(...)方法重新build一下自身即可。
2. 使用一些专门用于状态管理的包，如Provider、Redux，读者可以在pub上查看其详细信息。

本书将在“功能型组件”一章中介绍Provider包的实现原理及用法，同时也将会在“事件处理与通知”一章中实现一个全局事件总线，读者有需要可以直接翻看。