

- [介绍](#)
- [Hello World](#)
- [基础 Widget](#)
- [使用 Material 组件](#)
- [处理手势](#)
- [根据用户输入改变widget](#)
- [整合所有](#)
- [响应widget生命周期事件](#)
- [Key](#)
- [全局 Key](#)

## 介绍

Flutter Widget采用现代响应式框架构建，这是从 [React](#) 中获得的灵感，中心思想是用 widget构建你的UI。Widget描述了他们的视图在给定其当前配置和状态时应该看起来像什么。当widget的状态发生变化时，widget会重新构建UI，Flutter会对比前后变化的不同，以确定底层渲染树从一个状态转换到下一个状态所需的最小更改（译者语：类似于React/Vue中虚拟DOM的diff算法）。

**注意:** 如果您想通过代码来深入了解Flutter，请查看 [构建Flutter布局](#) 和 [为Flutter App添加交互功能](#)。

## Hello World

一个最简单的Flutter应用程序，只需一个widget即可！如下面示例：将一个widget传给 [runApp](#) 函数即可：

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

void main() {
  runApp(
    new Center(
      child: new Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

该[runApp](#)函数接受给定的[Widget](#)并使其成为widget树的根。在此示例中，widget树由两个[widget:Center](#)(及其子widget)和[Text](#)组成。框架强制根widget覆盖整个屏幕，这意味着文本“Hello, world”会居中显示在屏幕上。文本显示的方向需要在[Text](#)实例中指定，当使用[MaterialApp](#)时，文本的方向将自动设定，稍后将进行演示。

在编写应用程序时，通常会创建新的widget，这些widget是无状态的[StatelessWidget](#)或者是有状态的[StatefulWidget](#)，具体的选择取决于您的widget是否需要管理一些状态。widget的主要工作是实现一个[build](#)函数，用以构建自身。一个widget通常由一些较低级别widget组成。Flutter框架将依次构建这些widget，直到构建到最底层的子widget时，这些最低层的widget通常为[RenderObject](#)，它会计算并描述widget的几何形状。

## 基础 Widget

主要文章: [widget概述-布局模型](#)

Flutter有一套丰富、强大的基础widget，其中以下是很常用的：

- [Text](#)：该 widget 可让创建一个带格式的文本。
- [Row](#)、[Column](#)：这些具有弹性空间的布局类Widget可让您在水平（Row）和垂直（Column）方向上创建灵活的布局。其设计是基于web开发中的Flexbox布局模型。
- [Stack](#)：取代线性布局 (译者语：和Android中的LinearLayout相似)，[Stack](#)允许子 widget 堆叠，你可以使用 [Positioned](#) 来定位他们相对于Stack的上下左右四条边的位置。Stacks是基于Web开发中的绝度定位（absolute positioning）布局模型设计的。
- [Container](#)：[Container](#) 可让您创建矩形视觉元素。container 可以装饰为一个[BoxDecoration](#)，如 background、一个边框、或者一个阴影。[Container](#) 也可以具有边距（margins）、填充(padding)和应用其大小的约束(constraints)。另外，[Container](#)可以使用矩阵在三维空间中对其进行变换。

以下是一些简单的Widget，它们可以组合出其它的Widget：

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

class MyAppBar extends StatelessWidget {
  MyAppBar({this.title});

  // Widget子类中的字段往往都会定义为“final”

  final Widget title;
```

```

@override
Widget build(BuildContext context) {
  return new Container(
    height: 56.0, // 单位是逻辑上的像素（并非真实的像素，类似于浏览器中的像素）
    padding: const EdgeInsets.symmetric(horizontal: 8.0),
    decoration: new BoxDecoration(color: Colors.blue[500]),
    // Row 是水平方向的线性布局（linear layout）
    child: new Row(
      //列表项的类型是 <Widget>
      children: <Widget>[
        new IconButton(
          icon: new Icon(Icons.menu),
          tooltip: 'Navigation menu',
          onPressed: null, // null 会禁用 button
        ),
        // Expanded expands its child to fill the available space.
        new Expanded(
          child: title,
        ),
        new IconButton(
          icon: new Icon(Icons.search),
          tooltip: 'Search',
          onPressed: null,
        ),
      ],
    ),
  );
}

```

```

class MyScaffold extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Material 是UI呈现的“一张纸”
    return new Material(
      // Column is 垂直方向的线性布局.
      child: new Column(
        children: <Widget>[
          new MyAppBar(
            title: new Text(
              'Example title',
              style: Theme.of(context).primaryTextTheme.title,
            ),
          ),
        ],
      ),
    ),
  ),
}

```

```

        new Expanded(
          child: new Center(
            child: new Text('Hello, world!'),
          ),
        ),
      ],
    ),
  );
}

void main() {
  runApp(new MaterialApp(
    title: 'My app', // used by the OS task switcher
    home: new MyScaffold(),
  ));
}

```

请确保在pubspec.yaml文件中，将flutter的值设置为：uses-material-design: true。这允许我们可以使用一组预定义[Material icons](#)。

```

name: my_app
flutter:
  uses-material-design: true

```

为了继承主题数据，widget需要位于[MaterialApp](#)内才能正常显示，因此我们使用[MaterialApp](#)来运行该应用。

在MyAppBar中创建一个Container，高度为56像素（像素单位独立于设备，为逻辑像素），其左侧和右侧均有8像素的填充。在容器内部，MyAppBar使用Row布局来排列其子项。中间的title widget被标记为Expanded，这意味着它会填充尚未被其他子项占用的剩余可用空间。Expanded可以拥有多个children，然后使用flex参数来确定他们占用剩余空间的比例。

MyScaffold 通过一个Column widget，在垂直方向排列其子项。在Column的顶部，放置了一个MyAppBar实例，将一个Text widget作为其标题传递给应用程序栏。将widget作为参数传递给其他widget是一种强大的技术，可以让您创建各种复杂的widget。最后，MyScaffold使用了一个Expanded来填充剩余的空间，正中间包含一条message。

## 使用 Material 组件

主要文章: [Widgets 总览 - Material 组件](#)

Flutter提供了许多widgets，可帮助您构建遵循Material Design的应用程序。Material应用程序以[MaterialApp](#) widget开始，该widget在应用程序的根部创建了一些有用的widget，其中包括一个[Navigator](#)，它管理由字符串标识的Widget栈（即页面路由

栈)。 [Navigator](#)可以让您的应用程序在页面之间的平滑的过渡。 是否使用[MaterialApp](#)完全是可选的，但是使用它是一个很好的做法。

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

void main() {
  runApp(new MaterialApp(
    title: 'Flutter Tutorial',
    home: new TutorialHome(),
  ));
}

class TutorialHome extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //Scaffold是Material中主要的布局组件.
    return new Scaffold(
      appBar: new AppBar(
        leading: new IconButton(
          icon: new Icon(Icons.menu),
          tooltip: 'Navigation menu',
          onPressed: null,
        ),
        title: new Text('Example title'),
        actions: <Widget>[
          new IconButton(
            icon: new Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
          ),
        ],
      ),
      //body占屏幕的大部分
      body: new Center(
        child: new Text('Hello, world!'),
      ),
      floatingActionButton: new FloatingActionButton(
        tooltip: 'Add', // used by assistive technologies
        child: new Icon(Icons.add),
        onPressed: null,
      ),
    );
  }
}
```

现在我们已经从MyAppBar和MyScaffold切换到了AppBar和 Scaffold widget，我们的应用程序现在看起来已经有一些“Material”了！例如，应用栏有一个阴影，标题文本会自动继承正确的样式。我们还添加了一个浮动操作按钮，以便进行相应的操作处理。

请注意，我们再次将widget作为参数传递给其他widget。该 Scaffold widget 需要许多不同的widget的作为命名参数，其中的每一个被放置在Scaffold布局中相应的位置。同样，AppBar 中，我们给参数leading、actions、title分别传一个widget。这种模式在整个框架中会经常出现，这也可能是您在设计自己的widget时会考虑到一点。

## 处理手势

主要文章: [Flutter中的手势](#)

大多数应用程序包括某种形式与系统的交互。构建交互式应用程序的第一步是检测输入手势。让我们通过创建一个简单的按钮来了解它的工作原理：

```
class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: new Container(
        height: 36.0,
        padding: const EdgeInsets.all(8.0),
        margin: const EdgeInsets.symmetric(horizontal: 8.0),
        decoration: new BoxDecoration(
          borderRadius: new BorderRadius.circular(5.0),
          color: Colors.lightGreen[500],
        ),
        child: new Center(
          child: new Text('Engage'),
        ),
      ),
    );
  }
}
```

该GestureDetector widget并不具有显示效果，而是检测由用户做出的手势。当用户点击Container时，GestureDetector会调用它的onTap回调，在回调中，将消息打印到控制台。您可以使用GestureDetector来检测各种输入手势，包括点击、拖动和缩放。

许多widget都会使用一个GestureDetector为其他widget提供可选的回调。例如，IconButton、RaisedButton、和FloatingActionButton，它们都有一个onPressed回调，它会在

用户点击该widget时被触发。

## 根据用户输入改变widget

主要文章: [StatefulWidget](#), [State.setState](#)

到目前为止，我们只使用了无状态的widget。无状态widget从它们的父widget接收参数，它们被存储在[final](#)型的成员变量中。当一个widget被要求构建时，它使用这些存储的值作为参数来构建widget。

为了构建更复杂的体验 - 例如，以更有趣的方式对用户输入做出反应 - 应用程序通常会携带一些状态。Flutter使用StatefulWidgets来满足这种需求。StatefulWidgets是特殊的widget，它知道如何生成State对象，然后用它来保持状态。思考下面这个简单的例子，其中使用了前面提到[RaisedButton](#)：

```
class Counter extends StatefulWidget {  
  // This class is the configuration for the state. It holds the  
  // values (in this nothing) provided by the parent and used by the build  
  // method of the State. Fields in a Widget subclass are always marked "final".
```

```
  @override  
  _CounterState createState() => new _CounterState();  
}
```

```
class _CounterState extends State<Counter> {  
  int _counter = 0;  
  
  void _increment() {  
    setState(() {  
      // This call to setState tells the Flutter framework that  
      // something has changed in this State, which causes it to rerun  
      // the build method below so that the display can reflect the  
      // updated values. If we changed _counter without calling  
      // setState(), then the build method would not be called again,  
      // and so nothing would appear to happen.  
      _counter++;  
    });  
  }  
}
```

```
  @override  
  Widget build(BuildContext context) {  
    // This method is rerun every time setState is called, for instance  
    // as done by the _increment method above.  
    // The Flutter framework has been optimized to make rerunning  
    // build methods fast, so that you can just rebuild anything that  
    // needs updating rather than having to individually change
```

```

// instances of widgets.
return new Row(
  children: <Widget>[
    new RaisedButton(
      onPressed: _increment,
      child: new Text('Increment'),
    ),
    new Text('Count: $_counter'),
  ],
);
}
}

```

您可能想知道为什么StatefulWidget和State是单独的对象。在Flutter中，这两种类型的对象具有不同的生命周期：Widget是临时对象，用于构建当前状态下的应用程序，而State对象在多次调用[build\(\)](#)之间保持不变，允许它们记住信息(状态)。

上面的例子接受用户点击，并在点击时使`_counter`自增，然后直接在其`build`方法中使用`_counter`值。在更复杂的应用程序中，widget结构层次的不同部分可能有不同的职责；例如，一个widget可能呈现一个复杂的用户界面，其目标是收集特定信息（如日期或位置），而另一个widget可能会使用该信息来更改整体的显示。

在Flutter中，事件流是“向上”传递的，而状态流是“向下”传递的（译者语：这类似于React/Vue中父子组件通信的方式：子widget到父widget是通过事件通信，而父到子是通过状态），重定向这一流程的共同父元素是State。让我们看看这个稍微复杂的例子是如何工作的：

```

class CounterDisplay extends StatelessWidget {
  CounterDisplay({this.count});

  final int count;

  @override
  Widget build(BuildContext context) {
    return new Text('Count: $count');
  }
}

class CounterIncrementor extends StatelessWidget {
  CounterIncrementor({this.onPressed});

  final VoidCallback onPressed;

  @override

```



```

Widget build(BuildContext context) {
  return new RaisedButton(
    onPressed: onPressed,
    child: new Text('Increment'),
  );
}

class Counter extends StatefulWidget {
  @override
  _CounterState createState() => new _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;

  void _increment() {
    setState(() {
      ++_counter;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Row(children: <Widget>[
      new CounterIncrementor(onPressed: _increment),
      new CounterDisplay(count: _counter),
    ]);
  }
}

```

注意我们是如何创建了两个新的无状态widget的！我们清晰地分离了 显示 计数器（CounterDisplay）和 更改 计数器（CounterIncrementor）的逻辑。尽管最终效果与前一个示例相同，但责任分离允许将复杂性逻辑封装在各个widget中，同时保持父项的简单性。

## 整合所有

让我们考虑一个更完整的例子，将上面介绍的概念汇集在一起。我们假设一个购物应用程序，该应用程序显示出售的各种产品，并维护一个购物车。我们先来定义

ShoppingListItem：

```

class Product {
  const Product({this.name});
  final String name;
}

```

```
}
```

```
typedef void CartChangedCallback(Product product, bool inCart);
```

```
class ShoppingListItem extends StatelessWidget {  
  ShoppingListItem({Product product, this.inCart, this.onCartChanged})  
    : product = product,  
    super(key: new ObjectKey(product));
```

```
  final Product product;
```

```
  final bool inCart;
```

```
  final CartChangedCallback onCartChanged;
```

```
  Color _getColor(BuildContext context) {
```

```
    // The theme depends on the BuildContext because different parts of the tree
```

```
    // can have different themes. The BuildContext indicates where the build is
```

```
    // taking place and therefore which theme to use.
```

```
    return inCart ? Colors.black54 : Theme.of(context).primaryColor;
```

```
  }
```

```
  TextStyle _getTextStyle(BuildContext context) {
```

```
    if (!inCart) return null;
```

```
    return new TextStyle(  
      color: Colors.black54,
```

```
      decoration: TextDecoration.lineThrough,
```

```
    );
```

```
  }
```

```
@override
```

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

```
  return new ListTile(  
    onTap: () {
```

```
      onCartChanged(product, !inCart);
```

```
    },
```

```
    leading: new CircleAvatar(  
      backgroundColor: _getColor(context),
```

```
      child: new Text(product.name[0]),
```

```
    ),
```

```
    title: new Text(product.name, style: _getTextStyle(context)),
```

```
  );
```

```
}
```

```
}
```

该`ShoppingListItem` widget是无状态的。它将其在构造函数中接收到的值存储在`final`成员变量中，然后在`build`函数中使用它们。例如，`inCart`布尔值表示在两种视觉展示效果之间切换：一个使用当前主题的主色，另一个使用灰色。

当用户点击列表项时，widget不会直接修改其`inCart`的值。相反，widget会调用其父widget给它的`onCartChanged`回调函数。此模式可让您在widget层次结构中存储更高的状态，从而使状态持续更长的时间。在极端情况下，存储传给[runApp](#)应用程序的widget的状态将在整个生命周期中持续存在。

当父项收到`onCartChanged`回调时，父项将更新其内部状态，这将触发父项使用新`inCart`值重建`ShoppingListItem`新实例。虽然父项`ShoppingListItem`在重建时创建了一个新实例，但该操作开销很小，因为Flutter框架会将新构建的widget与先前构建的widget进行比较，并仅将差异部分应用于底层[RenderObject](#)。

我们来看看父widget存储可变状态的示例：

```
class ShoppingList extends StatefulWidget {
  ShoppingList({Key key, this.products}) : super(key: key);

  final List<Product> products;

  // The framework calls createState the first time a widget appears at a given
  // location in the tree. If the parent rebuilds and uses the same type of
  // widget (with the same key), the framework will re-use the State object
  // instead of creating a new State object.

  @override
  _ShoppingListState createState() => new _ShoppingListState();
}

class _ShoppingListState extends State<ShoppingList> {
  Set<Product> _shoppingCart = new Set<Product>();

  void _handleCartChanged(Product product, bool inCart) {
    setState(() {
      // When user changes what is in the cart, we need to change _shoppingCart
      // inside a setState call to trigger a rebuild. The framework then calls
      // build, below, which updates the visual appearance of the app.

      if (inCart)
        _shoppingCart.add(product);
      else
        _shoppingCart.remove(product);
    });
  }
}
```

```

    }

    @override
    Widget build(BuildContext context) {
      return new Scaffold(
        appBar: new AppBar(
          title: new Text('Shopping List'),
        ),
        body: new ListView(
          padding: new EdgeInsets.symmetric(vertical: 8.0),
          children: widget.products.map((Product product) {
            return new ShoppingListItem(
              product: product,
              inCart: _shoppingCart.contains(product),
              onCartChanged: _handleCartChanged,
            );
          }).toList(),
        ),
      );
    }
  }

void main() {
  runApp(new MaterialApp(
    title: 'Shopping App',
    home: new ShoppingList(
      products: <Product>[
        new Product(name: 'Eggs'),
        new Product(name: 'Flour'),
        new Product(name: 'Chocolate chips'),
      ],
    ),
  ));
}

```

ShoppingList类继承自[StatefulWidget](#)，这意味着这个widget可以存储状态。当ShoppingList首次插入到树中时，框架会调用其 `createState` 函数以创建一个新的 `_ShoppingListState` 实例来与该树中的相应位置关联（请注意，我们通常命名State子类时带一个下划线，这表示其是私有的）。当这个widget的父级重建时，父级将创建一个新的ShoppingList实例，但是Flutter框架将重用已经在树中的 `_ShoppingListState` 实例，而不是再次调用 [createState](#) 创建一个新的。

要访问当前ShoppingList的属性，`_ShoppingListState`可以使用它的`widget`属性。如果父级重建并创建一个新的ShoppingList，那么 `_ShoppingListState`也将用新的`widget`值重建

(译者语：这里原文档有错误，应该是 `ShoppingListState` 不会重新构建，但其 `widget` 的属性会更新为新构建的 `widget`)。如果希望在 `widget` 属性更改时收到通知，则可以覆盖 `didUpdateWidget` 函数，以便将旧的 `oldWidget` 与当前 `widget` 进行比较。

处理 `onCartChanged` 回调时，`ShoppingListState` 通过添加或删除产品来改变其内部 `shoppingCart` 状态。为了通知框架它改变了它的内部状态，需要调用 `setState`。调用 `setState` 将该 `widget` 标记为“dirty”(脏的)，并且计划在下次应用程序需要更新屏幕时重新构建它。如果在修改 `widget` 的内部状态后忘记调用 `setState`，框架将不知道您的 `widget` 是“dirty”(脏的)，并且可能不会调用 `widget` 的 `build` 方法，这意味着用户界面可能不会更新以展示新的状态。

通过以这种方式管理状态，您不需要编写用于创建和更新子 `widget` 的单独代码。相反，您只需实现可以处理这两种情况的 `build` 函数。

## 响应 `widget` 生命周期事件

主要文章: [State](#)

在 `StatefulWidget` 调用 `createState` 之后，框架将新的状态对象插入树中，然后调用状态对象的 `initState`。子类化 `State` 可以重写 `initState`，以完成仅需要执行一次的工作。例如，您可以重写 `initState` 以配置动画或订阅 `platform services`。`initState` 的实现中需要调用 `super.initState`。

当一个状态对象不再需要时，框架调用状态对象的 `dispose`。您可以覆盖该 `dispose` 方法来执行清理工作。例如，您可以覆盖 `dispose` 取消定时器或取消订阅 `platform services`。`dispose` 典型的实现是直接调用 `super.dispose`。

## Key

主要文章: [Key](#)

您可以使用 `key` 来控制框架将在 `widget` 重建时与哪些其他 `widget` 匹配。默认情况下，框架根据它们的 `runtimeType` 和它们的显示顺序来匹配。使用 `key` 时，框架要求两个 `widget` 具有相同的 `key` 和 `runtimeType`。

`Key` 在构建相同类型 `widget` 的多个实例时很有用。例如，`ShoppingList` 构建足够的 `ShoppingListItem` 实例以填充其可见区域：

- 如果没有 `key`，当前构建中的第一个条目将始终与前一个构建中的第一个条目同步，即使在语义上，列表中的第一个条目如果滚动出屏幕，那么它将不会再在窗口中可见。
- 通过给列表中的每个条目分配为“语义” `key`，无限列表可以更高效，因为框架将同步条目与匹配的语义 `key` 并因此具有相似（或相同）的可视外观。此

外，语义上同步条目意味着在有状态子widget中，保留的状态将附加到相同的语义条目上，而不是附加到相同数字位置上的条目。

## 全局 Key

主要文章: [GlobalKey](#)

您可以使用全局key来唯一标识子widget。全局key在整个widget层次结构中必须是全局唯一的，这与局部key不同，后者只需要在同级中唯一。由于它们是全局唯一的，因此可以使用全局key来检索与widget关联的状态。