

基础Widget

本节介绍一下Flutter中常用的一些基础widget，由于大多数widget的属性都比较多，我们在介绍widget时会着重介绍常用的属性，而不会像API文档一样所有属性都介绍，关于属性详细的信息请参考Flutter SDK文档。

本章目录

- [Widget简介](#)
- [文本、字体样式](#)
- [按钮](#)
- [图片和Icon](#)
- [单选框和复选框](#)
- [输入框和表单](#)

Widget简介

概念

在前面的介绍中，我们知道，在Flutter中，几乎所有的对象都是一个Widget，与原生开发中的“控件”不同的是，Flutter中的widget的概念更广泛，它不仅可以表示UI元素，也可以表示一些功能性的组件如：用于手势检测的 `GestureDetector` widget、用于应用主题数据传递的 `Theme` 等等。而原生开发中的控件通常只是指UI元素。在后面的内容中，我们在描述UI元素时，我们可能会用到“控件”、“组件”这样的概念，读者心里需要知道他们就是widget，只是在不同场景的不同表述而已。由于Flutter主要就是用于构建用户界面的，所以，在大多数时候，读者可以认为widget就是一个控件，不必纠结于概念。

Widget与Element

在Flutter中，Widget的功能是“描述一个UI元素的配置数据”，它就是说，Widget其实并不是表示最终绘制在设备屏幕上的显示元素，而只是显示元素的一个配置数据。实际上，Flutter中真正代表屏幕上显示元素的类是 `Element`，也就是说Widget只是描述 `Element` 的一个配置，有关 `Element` 的详细介绍我们将在本书后面的高级部分深入介绍，读者现在只需要知道，Widget只是UI元素的一个配置数据，并且一个Widget可以对应多个 `Element`，这是因为同一个Widget对象可以被添加到UI树的不同部分，而真正渲染时，UI树的每一个节点都会对应一个 `Element` 对象。总结一下：

- Widget实际上就是Element的配置数据，Widget树实际上是一个配置树，而真正的UI渲染树是由Element构成；不过，由于Element是通过Widget生成，所以它们之间有对应关系，所以在表述上，我们可以宽泛的认为Widget树就是指UI控件树或UI渲染树。
- 一个Widget对象可以对应多个Element对象。

读者应该将这两点刻在心中。

主要接口

我们先来看一下Widget类的声明：

```

@immutable
abstract class Widget extends DiagnosticableTree {
  const Widget({ this.key });
  final Key key;

  @protected
  Element createElement();

  @override
  String toStringShort() {
    return key == null ? '$runtimeType' : '$runtimeType-$key';
  }

  @override
  void debugFillProperties(DiagnosticPropertiesBuilder properties)
  {
    super.debugFillProperties(properties);
    properties.defaultDiagnosticsTreeStyle =
    DiagnosticsTreeStyle.dense;
  }

  static bool canUpdate(Widget oldWidget, Widget newWidget) {
    return oldWidget.runtimeType == newWidget.runtimeType
      && oldWidget.key == newWidget.key;
  }
}

```

- `Widget` 类继承自 `DiagnosticableTree`，`DiagnosticableTree` 即“诊断树”，主要作用是提供调试信息。
- `Key`：这个 `key` 属性类似于 React/Vue 中的 `key`，主要的作用是决定是否在下次 `build` 时复用旧的 widget，决定的条件在 `canUpdate()` 方法中。
- `createElement()`：正如前文所述“一个 `Widget` 可以对应多个 `Element`”；Flutter Framework 在构建 UI 树时，会先调用此方法生成对应节点的 `Element` 对象。此方法是 Flutter Framework 隐式调用的，在我们开发过程中基本不会调用到。
- `debugFillProperties(...)` 复写父类的方法，主要是设置诊断树的一些特性。
- `canUpdate(...)` 是一个静态方法，它主要用于在 `Widget` 树重新 `build` 时复用旧的 widget，其实具体来说，应该是：是否用新的 `Widget` 对象去更新旧 UI 树上所对应的 `Element` 对象的配置；通过其源码我们可以看到，只要 `newWidget` 与 `oldWidget` 的 `runtimeType` 和 `key` 同时相等时就会用 `newWidget` 去更新 `Element` 对象的配置，否则就会创建新的 `Element`。

有关Key和Widget复用的细节将会在本书后面高级部分深入讨论，读者现在只需知道，为Widget显式添加key的话可能（但不一定）会使UI在重新构建时变的高效，读者目前可以先忽略此参数。本书后面的示例中，我们只在构建列表项UI时会显示指定Key。

另外 `Widget` 类本身是一个抽象类，其中最核心的就是定义了 `createElement()` 接口，在Flutter开发中，我们一般都不用直接继承 `Widget` 类来实现Widget，相反，我们通常会通过继承 `StatelessWidget` 和 `StatefulWidget` 来间接继承 `Widget` 类来实现，而 `StatelessWidget` 和 `StatefulWidget` 都是直接继承自 `Widget` 类，而这两个类也正是Flutter中非常重要的两个抽象类，它们引入了两种Widget模型，接下来我们将重点介绍一下这两个类。

Stateless Widget

在之前的章节中，我们已经简单介绍过`StatelessWidget`，`StatelessWidget`相对比较简单，它继承自 `Widget`，重写了 `createElement()` 方法：

```
@override
StatelessElement createElement() => new StatelessElement(this);
```

`StatelessElement` 间接继承自 `Element` 类，与`StatelessWidget`相对应（作为其配置数据）。

`StatelessWidget`用于不需要维护状态的场景，它通常在 `build` 方法中通过嵌套其它Widget来构建UI，在构建过程中会递归的构建其嵌套的Widget。我们看一个简单的例子：

```

class Echo extends StatelessWidget {
  const Echo({
    Key key,
    @required this.text,
    this.backgroundColor: Colors.grey,
  }): super(key: key);

  final String text;
  final Color backgroundColor;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        color: backgroundColor,
        child: Text(text),
      ),
    );
  }
}

```

上面的代码，实现了一个回显字符串的 `Echo` widget。

按照惯例，widget的构造函数应使用命名参数，命名参数中的必要参数要添加 `@required` 标注，这样有利于静态代码分析器进行检查，另外，在继承widget时，第一个参数通常应该是 `Key`，如果接受子widget的child参数，那么通常应该将它放在参数列表的最后。同样是按照惯例，widget的属性应被声明为 `final`，防止被意外改变。

然后我们可以通过如下方式使用它：

```

Widget build(BuildContext context) {
  return Echo(text: "hello world");
}

```



Stateful Widget

和StatelessWidget一样，StatefulWidget也是继承自Widget类，并重写了 `createElement()` 方法，不同的是返回的 `Element` 对象并不相同；另外 StatefulWidget类中添加了一个新的接口 `createState()`，下面我们看看 StatefulWidget的类定义：

```
abstract class StatefulWidget extends Widget {  
  const StatefulWidget({ Key key }) : super(key: key);  
  
  @override  
  StatefulElement createElement() => new StatefulElement(this);  
  
  @protected  
  State createState();  
}
```

- `StatefulElement` 间接继承自 `Element` 类，与 `StatefulWidget` 相对应（作为其配置数据）。`StatefulElement` 中可能会多次调用 `createState()` 来创建状态 (State) 对象。
- `createState()` 用于创建和 `Stateful widget` 相关的状态，它在 `Stateful widget` 的生命周期中可能会被多次调用。例如，当一个 `Stateful widget` 同时插入到 `widget树` 的多个位置时，Flutter framework 就会调用该方法为每一个位置生成一个独立的 `State` 实例，其实，本质上就是一个 `StatefulElement` 对应一个 `State` 实例。

在本书中经常会出现“树”的概念，在不同的场景可能指不同的意思，在说“`widget树`”时它可以指 `widget` 结构树，但由于 `widget` 与 `Element` 有对应关系（一可能对多），在有些场景（Flutter 的 SDK 文档中）也代指“UI 树”的意思。而在 `stateful widget` 中，`State` 对象也和 `StatefulElement` 具有对应关系（一对一），所以在 Flutter 的 SDK 文档中，可以经常看到“从树中移除 `State` 对象”或“插入 `State` 对象到树中”这样的描述。其实，无论哪种描述，其意思都是在描述“一棵构成用户界面的节点元素的树”，读者不必纠结于这些概念，还是那句话“得其神，忘其形”，因此，本书中出现的各种“树”，如果没有特别说明，读者都可抽象的认为它是“一棵构成用户界面的节点元素的树”。

State

一个 `StatefulWidget` 类会对应一个 `State` 类，`State` 表示与其对应的 `StatefulWidget` 要维护的状态，`State` 中的保存的状态信息可以：

1. 在 `widget build` 时可以被同步读取。
2. 在 `widget` 生命周期中可以被改变，当 `State` 被改变时，可以手动调用其 `setState()` 方法通知 Flutter framework 状态发生改变，Flutter framework 在收到消息后，会重新调用其 `build` 方法重新构建 `widget树`，从而达到更新 UI 的目的。

`State` 中有两个常用属性：

1. `widget`，它表示与该State实例关联的widget实例，由Flutter framework动态设置。注意，这种关联并非永久的，因为在应用声明周期中，UI树上的某一个节点的widget实例在重新构建时可能会变化，但State实例只会在第一次插入到树中时被创建，当在重新构建时，如果widget被修改了，Flutter framework会动态设置State.widget为新的widget实例。
2. `context`，它是 `BuildContext` 类的一个实例，表示构建widget的上下文，它是操作widget在树中位置的一个句柄，它包含了一些查找、遍历当前Widget树的一些方法。每一个widget都有一个自己的context对象。

对于 `BuildContext` 读者现在可以先作了解，随着本书后面内容的展开，也会用到Context的一些方法，读者可以通过具体的场景对其有个直观的认识。关于 `BuildContext` 更多的内容，我们也将后面高级部分再深入介绍。

State生命周期

理解State的生命周期对flutter开发非常重要，为了加深读者印象，本节我们通过一个实例来演示一下State的生命周期。在接下来的示例中，我们实现一个计数器widget，点击它可以使计数器加1，由于要保存计数器的数值状态，所以我们应继承 `StatefulWidget`，代码如下：

```
class CounterWidget extends StatefulWidget {
  const CounterWidget({
    Key key,
    this.initValue: 0
  });

  final int initValue;

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

`CounterWidget` 接收一个 `initValue` 整形参数，它表示计数器的初始值。下面我们看一下State的代码：

```
class _CounterWidgetState extends State<CounterWidget> {
  int _counter;

  @override
```



```

void initState() {
    super.initState();
    // 初始化状态
    _counter=widget.initValue;
    print("initState");
}

@override
Widget build(BuildContext context) {
    print("build");
    return Center(
        child: FlatButton(
            child: Text('$ _counter'),
            // 点击后计数器自增
            onPressed:()=>setState(()=> ++_counter) ,
        ),
    );
}

@override
void didUpdateWidget(CounterWidget oldWidget) {
    super.didUpdateWidget(oldWidget);
    print("didUpdateWidget");
}

@override
void deactivate() {
    super.deactivate();
    print("deactive");
}

@override
void dispose() {
    super.dispose();
    print("dispose");
}

@override
void reassemble() {
    super.reassemble();
    print("reassemble");
}

@override
void didChangeDependencies() {
    super.didChangeDependencies();
    print("didChangeDependencies");
}

```

```
}
```

接下来，我们创建一个新路由，在新路由中，我们只显示一个 `CounterWidget`：

```
Widget build(BuildContext context) {  
  return CounterWidget();  
}
```

我们运行应用并打开该路由页面，在新路由页打开后，屏幕中央就会出现一个数字 0，然后控制台日志输出：

```
I/flutter ( 5436): initState  
I/flutter ( 5436): didChangeDependencies  
I/flutter ( 5436): build
```

可以看到，在 `StatefulWidget` 插入到 `Widget` 树时首先 `initState` 方法会被调用。

然后我们点击 ⚡ 按钮热重载，控制台输出日志如下：

```
I/flutter ( 5436): reassemble  
I/flutter ( 5436): didUpdateWidget  
I/flutter ( 5436): build
```

可以看到此时 `initState` 和 `didChangeDependencies` 都没有被调用，而此时 `didUpdateWidget` 被调用。

接下来，我们在 `widget` 树中移除 `CounterWidget`，将路由 `build` 方法改为：

```
Widget build(BuildContext context) {  
  // 移除计数器  
  //return CounterWidget();  
  // 随便返回一个Text()  
  return Text("xxx");  
}
```

然后热重载，日志如下：

```
I/flutter ( 5436): reassemble  
I/flutter ( 5436): deactivate  
I/flutter ( 5436): dispose
```

我们可以看到，在 `CounterWidget` 从widget树中移除时，`deactivate` 和 `dispose` 会依次被调用。

下面我们来看看各个回调函数：

- `initState`：当Widget第一次插入到Widget树时会被调用，对于每一个State对象，Flutter framework只会调用一次该回调，所以，通常在该回调中做一些一次性的操作，如状态初始化、订阅子树的事件通知等。不能在该回调中调用 `BuildContext.inheritFromWidgetOfExactType`（该方法用于在Widget树上获取离当前widget最近的一个父级 `InheritFromWidget`，关于 `InheritedWidget` 我们将在后面章节介绍），原因是在初始化完成后，Widget树中的 `InheritFromWidget` 也可能会发生变化，所以正确的做法应该在 `build()` 方法或 `didChangeDependencies()` 中调用它。
- `didChangeDependencies()`：当State对象的依赖发生变化时会被调用；例如：在之前 `build()` 中包含了一个 `InheritedWidget`，然后在之后的 `build()` 中 `InheritedWidget` 发生了变化，那么此时 `InheritedWidget` 的子widget的 `didChangeDependencies()` 回调都会被调用。典型的场景是当系统语言Locale或应用主题改变时，Flutter framework会通知widget调用此回调。
- `build()`：此回调读者现在应该已经相当熟悉了，它主要是用于构建Widget子树的，会在如下场景被调用：
 - i. 在调用 `initState()` 之后。
 - ii. 在调用 `didUpdateWidget()` 之后。
 - iii. 在调用 `setState()` 之后。
 - iv. 在调用 `didChangeDependencies()` 之后。
 - v. 在State对象从树中一个位置移除后（会调用`deactivate`）又重新插入到树的其它位置之后。
- `reassemble()`：此回调是专门为了开发调试而提供的，在热重载(hot reload)时会被调用，此回调在Release模式下永远不会被调用。
- `didUpdateWidget()`：在widget重新构建时，Flutter framework会调用 `Widget.canUpdate` 来检测Widget树中同一位置的新旧节点，然后决定是否

需要更新，如果 `Widget.canUpdate` 返回 `true` 则会调用此回调。正如之前所述，`Widget.canUpdate` 会在新旧widget的key和runtimeType同时相等时会返回true，也就是说在在新旧widget的key和runtimeType同时相等时 `didUpdateWidget()` 就会被调用。

- `deactivate()`：当State对象从树中被移除时，会调用此回调。在一些场景下，Flutter framework会将State对象重新插到树中，如包含此State对象的子树在树的一个位置移动到另一个位置时（可以通过GlobalKey来实现）。如果移除后没有重新插入到树中则紧接着会调用 `dispose()` 方法。
- `dispose()`：当State对象从树中被永久移除时调用；通常在此回调中释放资源。

todo: 这里缺一张生命周期图

注意：在继承StatefulWidget重写其方法时，对于包含@mustCallSuper标注的父类方法，都要在子类方法中先调用父类方法。

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，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，stateful widget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state。
- 父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？以下原则可以帮助你决定：

- 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`。



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。

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],
        ),
      ),
    );
  }
}

```

父widget管理子widget的state

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是一个图片按钮，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项。由于TapboxB不管理任何状态，因此它的父类为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widget处理。
- 当检测到点击时, 它会通知父widget。

```
// 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.onChange})
    : super(key: key);

  final bool active;
  final ValueChanged<bool> onChange;

  void _handleTap() {
    onChange(!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],
        ),
      ),
    );
  }
}

```

混合管理

对于一些widget来说，混和管理的方式非常有用。在这种情况下，widget自身管理一些内部状态，而父widget管理一些其他外部状态。

在下面TapboxC示例中，点击时，盒子的周围会出现一个深绿色的边框。点击时，边框消失，盒子的颜色改变。TapboxC将其 `_active` 状态导出到其父widget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` 对象:

- 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` state。
- `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widget.

```
//----- 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;

  _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);
  }

  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widget，同时保持 `_active` 状态为内部，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。

全局状态管理

当应用中包括一些跨widget（甚至跨路由）的状态需要同步时，上面介绍的方法很难胜任了。比如，我们有一个设置页，里面可以设置应用语言，但是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，我们的APP Widget能够重新build一下，但我们的APP Widget和设置页并不在一起。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widget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 Widget所在的父widget `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，我们触发语言改变事件，然后APP Widget那边就会收到通知，然后重新 `build` 一下即可。
2. 使用redux这样的全局状态包，读者可以在pub上查看其详细信息。

本书后面事件处理一章中会实现一个全局事件总线。

Flutter widget库介绍

Flutter提供了一套丰富、强大的基础widget，在基础widget库之上Flutter又提供了一套Material风格（Android默认的视觉风格）和一套Cupertino风格（iOS视觉风格）的widget库。要使用基础widget库，需要先导入：

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

下面我们介绍换一下常用的widget。

基础widget

- `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FrameLayout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开发中的绝对定位（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。

Material widget

Flutter提供了一套丰富的Material widget，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，比如一个Theme，它配置了应用的主题。是否使用 `MaterialApp` 完全是可选的，但是使用它是一个很好的做法。在之前的示例中，我们已经使用过多个Material widget了，如：`Scaffold`、`AppBar`、`FlatButton` 等。要使用Material widget，需要先引入它：

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

Cupertino widget

Flutter也提供了一套丰富的Cupertino风格的widget，尽管目前还没有Material widget那么丰富，但也在不断的完善中。值得一提的是在Material widget库中，有一些widget可以根据实际运行平台来切换表现风格，比如 `MaterialPageRoute`，在路由切换时，如果是Android系统，它将会使用Android系统默认的页面切换动画(从底向上)，如果是iOS系统时，它会使用iOS系统默认的页面切换动画（从右向左）。由于在前面的示例中还没有Cupertino widget的示例，我们实现一个简单的Cupertino页面：

```
//导入cupertino widget库
import 'package:flutter/cupertino.dart';

class CupertinoTestRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return CupertinoPageScaffold(
      navigationBar: CupertinoNavigationBar(
        middle: Text("Cupertino Demo"),
      ),
      child: Center(
        child: CupertinoButton(
          color: CupertinoColors.activeBlue,
          child: Text("Press"),
          onPressed: () {}
        ),
      ),
    );
  }
}
```

下面是在iPhoneX上页面效果截图：



Press

总结

Flutter提供了丰富的widget，在实际的开发中你可以随意使用它们，不要怕引入过多widget库会让你的应用安装包变大，这不是web开发，dart在编译时只会编译你使用了的代码。由于Material和Cupertino都是在基础widget库之上的，所以如果你的应用中引入了这两者之一，则不需要再引入 `flutter/widgets.dart` 了，因为它们内部已经引入过了。

文本及样式

Text

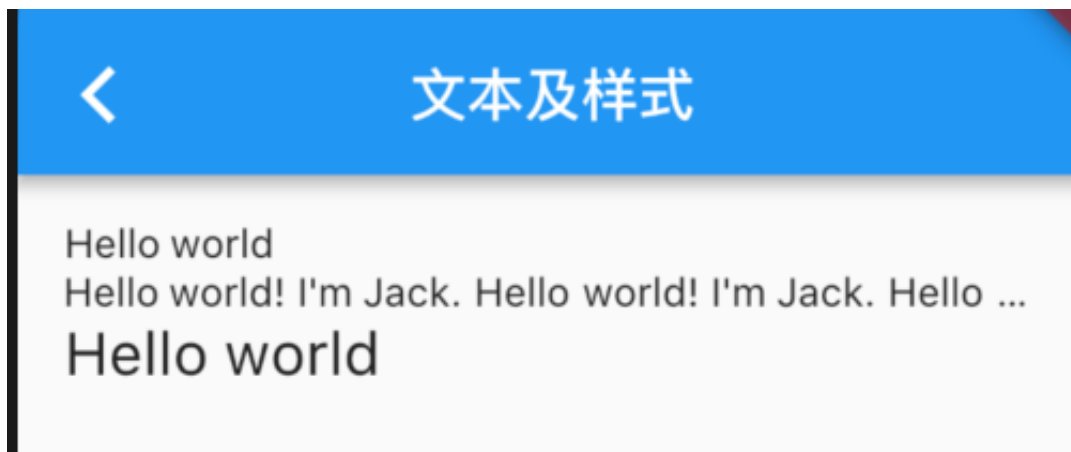
Text用于显示简单样式文本，它包含一些控制文本显示样式的一些属性，一个简单的例子如下：

```
Text("Hello world",
    textAlign: TextAlign.center,
);

Text("Hello world! I'm Jack. "*4,
    maxLines: 1,
    overflow: TextOverflow.ellipsis,
);

Text("Hello world",
    textScaleFactor: 1.5,
);
```

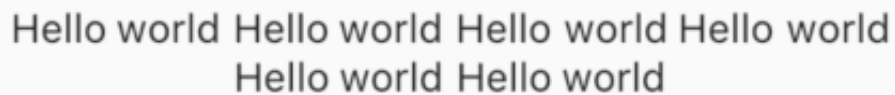
运行效果如下：



- `textAlign`：文本的对齐方式；可以选择左对齐、右对齐还是居中。注意，对齐的参考系是Text widget本身。本例中虽然是指定了居中对齐，但因为Text文本内容宽度不足一行，Text的宽度和文本内容长度相等，那么这时指定对齐方式是没有意义的，只有Text宽度大于文本内容长度时指定此属性才有意义。下面我们指定一个较长的字符串：

```
Text("Hello world "*6, // 字符串重复六次  
    textAlign: TextAlign.center,  
);
```

运行效果如下：



字符串内容超过一行，Text宽度等于屏幕宽度，第二行文本便会居中显示。

- `maxLines`、`overflow`：指定文本显示的最大行数，默认情况下，文本是自动折行的，如果指定此参数，则文本最多不会超过指定的行。如果有多余的文本，可以通过 `overflow` 来指定截断方式，默认是直接截断，本例中指定的截断方式 `TextOverflow.ellipsis`，它会将多余文本截断后以省略符“...”表示；`TextOverflow`的其它截断方式请参考SDK文档。
- `textScaleFactor`：代表文本相对于当前字体大小的缩放因子，相对于去设置文本的样式 `style` 属性的 `fontSize`，它是调整字体大小的一个快捷方式。该属性的默认值可以通过 `MediaQueryData.textScaleFactor` 获得，如果没有 `MediaQuery`，那么会默认值将为1.0。

TextStyle

`TextStyle`用于指定文本显示的样式如颜色、字体、粗细、背景等。我们看一个示例：


```
Text("Hello world",
  style: TextStyle(
    color: Colors.blue,
    fontSize: 18.0,
    height: 1.2,
    fontFamily: "Courier",
    background: new Paint()..color=Colors.yellow,
    decoration:TextDecoration.underline,
    decorationStyle: TextDecorationStyle.dashed
  ),
);
```

效果如下：



此示例只展示了TextStyle的部分属性，它还有一些其它属性，属性名基本都是自解释的，在此不再赘述，读者可以查阅SDK文档。值得注意的是：

- `height`：该属性用于指定行高，但它并不是一个绝对值，而是一个因子，具体的行高等于 `fontSize * height`。
- `fontFamily`：由于不同平台默认支持的字体集不同，所以在手动指定字体时一定要先在不同平台测试一下。
- `fontSize`：该属性和Text的 `textScaleFactor` 都用于控制字体大小。但是有两给主要区别：
 - `fontSize` 可以精确指定字体大小，而 `textScaleFactor` 只能通过缩放比例来控制。
 - `textScaleFactor` 主要是用于系统字体大小设置改变时对Flutter应用字体进行全局调整，而 `fontSize` 通常用于单个文本。

TextSpan

在上面的例子中，Text的所有文本内容只能按同一种样式，如果我们需要对一个Text内容的不同部分按照不同的样式显示，这时就可以使用 `TextSpan`，它代表文本的一个“片段”。我们看看TextSpan的定义：

```
const TextSpan({
  TextStyle style,
  String text,
  List<TextSpan> children,
  GestureRecognizer recognizer,
});
```

其中 `style` 和 `text` 属性代表该文本片段的样式和内容。 `children` 是一个 `TextSpan` 的数组，也就是说 `TextSpan` 可以包括其他 `TextSpan` 。而 `recognizer` 用于对该文本片段上用于手势进行识别处理。下面我们看一个效果，然后用 `TextSpan` 实现它。

Home: <https://flutterchina.club>

源码：

```
Text.rich(TextSpan(
  children: [
    TextSpan(
      text: "Home: "
    ),
    TextSpan(
      text: "https://flutterchina.club",
      style: TextStyle(
        color: Colors.blue
      ),
      recognizer: _tapRecognizer
    ),
  ],
))
```

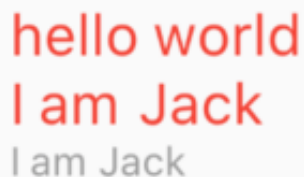
- 上面代码中，我们通过 `TextSpan` 实现了一个基础文本片段和一个链接片段，然后通过 `Text.rich` 方法将 `TextSpan` 添加到 `Text` 中，之所以可以这样做，是因为 `Text` 其实就是 `RichText` 的一个包装，而 `RichText` 是可以显示多种样式(富文本)的 widget。
- `_tapRecognizer` ，它是点击链接后的一个处理器（代码已省略），关于手势识别的更多内容我们将在后面单独介绍。

DefaultTextStyle

在widget树中，文本的样式默认是可以被继承的，因此，如果在widget树的某一个节点处设置一个默认的文本样式，那么该节点的子树中所有文本都会默认使用这个样式，而DefaultTextStyle正是用于设置默认文本样式的。下面我们看一个例子：

```
DefaultTextStyle(  
  //1. 设置文本默认样式  
  style: TextStyle(  
    color: Colors.red,  
    fontSize: 20.0,  
  ),  
  textAlign: TextAlign.start,  
  child: Column(  
    crossAxisAlignment: CrossAxisAlignment.start,  
    children: <Widget>[  
      Text("hello world"),  
      Text("I am Jack"),  
      Text("I am Jack",  
        style: TextStyle(  
          inherit: false, //2. 不继承默认样式  
          color: Colors.grey  
        ),  
      ),  
    ],  
  ),  
);
```

上面代码中，我们首先设置了一个默认的文本样式，即字体为20像素(逻辑像素)、颜色为红色。然后通过 `DefaultTextStyle` 设置给了子树Column节点处，这样一来Column的所有子孙Text默认都会继承该样式，除非Text显示指定不继承样式，如代码中注释2。示例运行效果如下：



hello world
I am Jack
I am Jack

使用字体

可以在Flutter应用程序中使用不同的字体。例如，我们可能会使用设计人员创建的自定义字体，或者其它第三方的字体，如[Google Fonts](#)中的字体。本节将介绍如何为Flutter应用配置字体，并在渲染文本时使用它们。

在Flutter中使用字体分两步完成。首先在 `pubspec.yaml` 中声明它们，以确保它们会打包到应用程序中。然后通过 `TextStyle` 属性使用字体。

在asset中声明

要将字体打文件打包到应用中，和使用其它资源一样，要先在 `pubspec.yaml` 中声明它。然后将字体文件复制到在 `pubspec.yaml` 中指定的位置。如：

```
flutter:
  fonts:
    - family: Raleway
      fonts:
        - asset: assets/fonts/Raleway-Regular.ttf
        - asset: assets/fonts/Raleway-Medium.ttf
          weight: 500
        - asset: assets/fonts/Raleway-SemiBold.ttf
          weight: 600
    - family: AbrilFatface
      fonts:
        - asset: assets/fonts/abrilfatface/AbrilFatface-Regular.ttf
```

使用字体

```
// 声明文本样式
const textStyle = const TextStyle(
  fontFamily: 'Raleway',
);

// 使用文本样式
var buttonText = const Text(
  "Use the font for this text",
  style: textStyle,
);
```

Package中的字体

要使用Package中定义的字体，必须提供 `package` 参数。例如，假设上面的字体声明位于 `my_package` 包中。然后创建TextStyle的过程如下：

```
const textStyle = const TextStyle(  
  fontFamily: 'Raleway',  
  package: 'my_package', // 指定包名  
);
```

如果在package包内部使用它自己定义的字体，也应该在创建文本样式时指定 `package` 参数，如上例所示。

一个包也可以只提供字体文件而不需要在pubspec.yaml中声明。这些文件应该包的 `lib/` 文件夹中。字体文件不会自动绑定到应用程序中，应用程序可以在声明字体时有选择地使用这些字体。假设一个名为my_package的包中有一个字体文件：

```
lib/fonts/Raleway-Medium.ttf
```

然后，应用程序可以声明一个字体，如下面的示例所示：

```
flutter:  
  fonts:  
    - family: Raleway  
      fonts:  
        - asset: assets/fonts/Raleway-Regular.ttf  
        - asset: packages/my_package/fonts/Raleway-Medium.ttf  
          weight: 500
```

`lib/` 是隐含的，所以它不应该包含在asset路径中。

在这种情况下，由于应用程序本地定义了字体，所以在创建TextStyle时可以不指定 `package` 参数：

```
const textStyle = const TextStyle(  
  fontFamily: 'Raleway',  
);
```

按钮

Material widget库中提供了多种按钮Widget如RaisedButton、FlatButton、OutlineButton等，它们都是直接或间接对RawMaterialButton的包装定制，所以他们大多数属性都和RawMaterialButton一样。在介绍各个按钮时我们先介绍其默认外观，而按钮的外观大都可以通过属性来自定义，我们在后面统一介绍这些属性。另外，所有Material库中的按钮都有如下相同点：

1. 按下时都会有“水波动画”。
2. 有一个 `onPressed` 属性来设置点击回调，当按钮按下时会执行该回调，如果不提供该回调则按钮会处于禁用状态，禁用状态不响应用户点击。

RaisedButton

RaisedButton 即"漂浮"按钮，它默认带有阴影和灰色背景。按下后，阴影会变大，如：



使用RaisedButton非常简单，如：

```
RaisedButton(  
  child: Text("normal"),  
  onPressed: () => {},  
);
```

FlatButton

FlatButton即扁平按钮，默认背景透明并不带阴影。按下后，会有背景色：



使用FlatButton也很简单，代码如下：

```
FlatButton(  
  child: Text("normal"),  
  onPressed: () => {},  
)
```

OutlineButton

OutlineButton默认有一个边框，不带阴影且背景透明。按下后，边框颜色会变亮、同时出现背景和阴影(较弱)：



使用OutlineButton也很简单，代码如下：

```
OutlineButton(  
  child: Text("normal"),  
  onPressed: () => {},  
)
```

IconButton

IconButton是一个可点击的Icon，不包括文字，默认没有背景，点击后会出现背景：



使用代码如下：

```
IconButton(  
  icon: Icon(Icons.thumb_up),  
  onPressed: () => {},  
)
```

自定义按钮外观

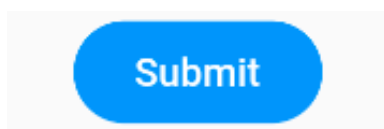
按钮外观可以通过其属性来定义，不同按钮属性大同小异，我们以FlatButton为例，介绍一下常见的按钮属性，详细的信息可以查看API文档。

```
const FlatButton({  
  ...  
  @required this.onPressed, // 按钮点击回调  
  this.textColor, // 按钮文字颜色  
  this.disabledTextColor, // 按钮禁用时的文字颜色  
  this.color, // 按钮背景颜色  
  this.disabledColor, // 按钮禁用时的背景颜色  
  this.highlightColor, // 按钮按下时的背景颜色  
  this.splashColor, // 点击时，水波动画中水波的颜色  
  this.colorBrightness, // 按钮主题，默认是浅色主题  
  this.padding, // 按钮的填充  
  this.shape, // 外形  
  @required this.child, // 按钮的内容  
})
```

其中大多数属性名都是自解释的，我们不赘述。下面我们通过一个示例来看看如何自定义按钮。

示例

定义一个背景蓝色，两边圆角的按钮。效果如下：

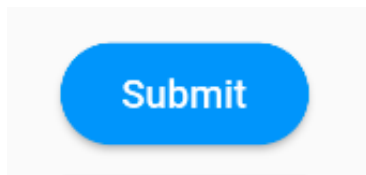


代码如下：


```
FlatButton(
  color: Colors.blue,
  highlightColor: Colors.blue[700],
  colorBrightness: Brightness.dark,
  splashColor: Colors.grey,
  child: Text("Submit"),
  shape: RoundedRectangleBorder(borderRadius:
    BorderRadius.circular(20.0)),
  onPressed: () => {},
)
```

很简单吧，在上面的代码中，我们主要通过 `shape` 来指定其外形为一个圆角矩形。因为按钮背景是蓝色(深色)，我们需要指定按钮主题 `colorBrightness` 为 `Brightness.dark`，这是为了保证按钮文字颜色为浅色。

细心的读者可能会发现这个按钮没有阴影(点击之后也没有)，这样会显得没有质感。其实这也很容易，将上面的 `FlatButton` 换成 `RaisedButton` 就行，其它代码不用改，换了之后我们看看效果：



是不是有质感了！之所以会这样，是因为 `RaisedButton` 默认有配置阴影：

```
const RaisedButton({
  ...
  this.elevation = 2.0, // 正常状态下的阴影
  this.highlightElevation = 8.0, // 按下时的阴影
  this.disabledElevation = 0.0, // 禁用时的阴影
  ...
})
```

值得注意的是，在Material widget库中，我们会在很多widget中见到`elevation`相关的属性，它们都是用来控制阴影的，这是因为阴影在Material设计风格中是一种很重要的表现形式，以后在介绍其它widget时，便不再赘述。

如果我们想实现一个背景渐变的圆角按钮，按钮有没有相应的属性呢？答案是否定的，但是，我们可以通过其它方式来实现，本文将在后面介绍Container时来介绍如何实现。

图片及ICON

图片

Flutter中，我们可以通过Image来加载并显示图片，Image的数据源可以是asset、文件、内存以及网络。

ImageProvider

`ImageProvider` 是一个抽象类，主要定义了图片数据获取的接口 `load()`，从不同的数据源获取图片需要实现不同的 `ImageProvider`，如 `AssetImage` 是实现了从Asset中加载图片的ImageProvider，而 `NetworkImage` 实现了从网络加载图片的ImageProvider。

Image

`Image` widget有一个必选的 `image` 参数，它对应一个ImageProvider。下面我们分别演示一下如何从asset和网络加载图片。

从asset中加载图片

1. 在工程根目录下创建一个 `images` 目录，并将图片 `avatar.png` 拷贝到该目录。
2. 在 `pubspec.yaml` 中的 `flutter` 部分添加如下内容：

```
assets:  
  - images/avatar.png
```

3. 加载该图片

```
Image(  
  image: AssetImage("images/avatar.png"),  
  width: 100.0  
);
```

`Image` 也提供了一个快捷的构造函数 `Image.asset` 用于从 `asset` 中加载、显示图片：

```
Image.asset("images/avatar.png",  
  width: 100.0,  
)
```

从网络加载图片

```
Image(  
  image: NetworkImage(  
    "https://avatars2.githubusercontent.com/u/20411648?  
s=460&v=4"),  
  width: 100.0,  
)
```

`Image` 也提供了一个快捷的构造函数 `Image.network` 用于从网络加载、显示图片：

```
Image.network(  
  "https://avatars2.githubusercontent.com/u/20411648?s=460&v=4",  
  width: 100.0,  
)
```

运行上面两个示例，图片加载成功后显示如下：



参数

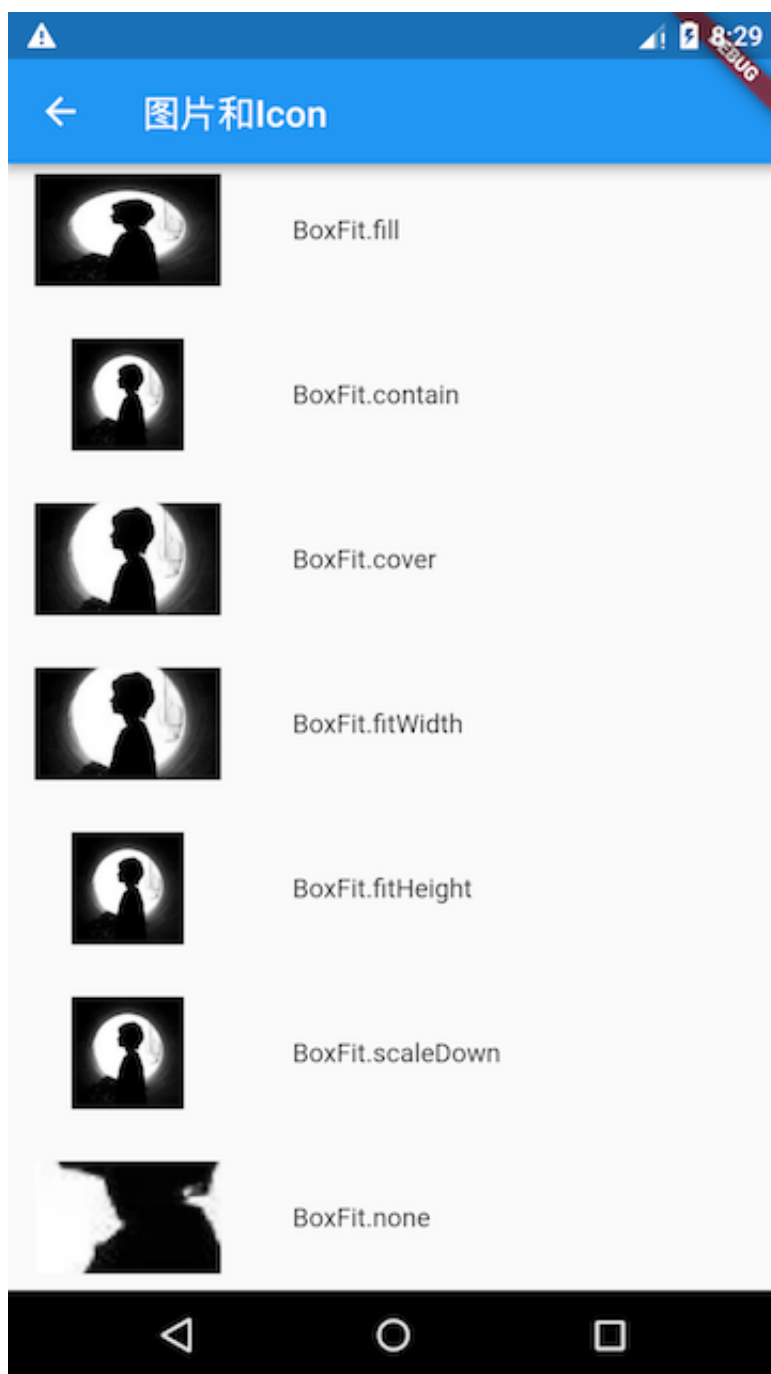
`Image` 在显示图片时定义了一系列参数，通过这些参数我们可以控制图片的显示外观、大小、混合效果等。我们看一下 `Image` 的主要参数：

```
const Image({
  ...
  this.width, // 图片的宽
  this.height, // 图片高度
  this.color, // 图片的混合色值
  this.colorBlendMode, // 混合模式
  this.fit, // 缩放模式
  this.alignment = Alignment.center, // 对齐方式
  this.repeat = ImageRepeat.noRepeat, // 重复方式
  ...
})
```

- `width`、`height`：用于设置图片的宽、高，当不指定宽高时，图片会根据当前父容器的限制，尽可能的显示其原始大小，如果只设置 `width`、`height` 的其中一个，那么另一个属性默认会按比例缩放，但可以通过下面介绍的 `fit` 属性来指定适应规则。
- `fit`：该属性用于在图片的显示空间和图片本身大小不同时指定图片的适应模式。适应模式是在 `BoxFit` 中定义，它是一个枚举类型，有如下值：
 - `fill`：会拉伸填充满显示空间，图片本身长宽比会发生变化，图片会变形。
 - `cover`：会按图片的长宽比放大后居中填满显示空间，图片不会变形，超出显示空间部分会被剪裁。
 - `contain`：这是图片的默认适应规则，图片会在保证图片本身长宽比不变的情况下缩放以适应当前显示空间，图片不会变形。

- `fitWidth`：图片的宽度会缩放到显示空间的宽度，高度会按比例缩放，然后居中显示，图片不会变形，超出显示空间部分会被剪裁。
- `fitHeight`：图片的高度会缩放到显示空间的高度，宽度会按比例缩放，然后居中显示，图片不会变形，超出显示空间部分会被剪裁。
- `none`：图片没有适应策略，会在显示空间内显示图片，如果图片比显示空间大，则显示空间只会显示图片中间部分。

一图胜万言：



- `color` 和 `colorBlendMode`：在图片绘制时可以对每一个像素进行颜色混合处理，`color` 指定混合色，而 `colorBlendMode` 指定混合模式，下面是一个简单的示例：

```
Image(  
  image: AssetImage("images/avatar.png"),  
  width: 100.0,  
  color: Colors.blue,  
  colorBlendMode: BlendMode.difference,  
);
```

运行效果如下（彩色）：



- `repeat`：当图片本身大小小于显示空间时，指定图片的重复规则。简单示例如下：

```
Image(  
  image: AssetImage("images/avatar.png"),  
  width: 100.0,  
  height: 200.0,  
  repeat: ImageRepeat.repeatY ,  
)
```

运行后效果如下：



ICON

Flutter中，可以像web开发一样使用iconfont，iconfont即“字体图标”，它是将图标做成字体文件，然后通过制定不同的字符而显示不同的图片。

在字体文件中，每一个字符都对应一个位码，而每一个位码对应一个显示字形，不同的字体就是指字形不同，即字符对应的字形是不同的。而在iconfont中，只是将位码对应的字形做成了图标，所以不同的字符最终就会渲染成不同的图标。

在Flutter开发中，iconfont和图片相比有如下优势：

1. 体积小：可以减小安装包大小。
2. 矢量的：iconfont都是矢量图标，放大不会影响其清晰度。
3. 可以应用文本样式：可以像文本一样改变字体图标的颜色、大小对齐等。
4. 可以通过TextSpan和文本混用。

使用Material Design字体图标

Flutter默认包含了一套Material Design的字体图标，在 `pubspec.yaml` 文件中的配置如下

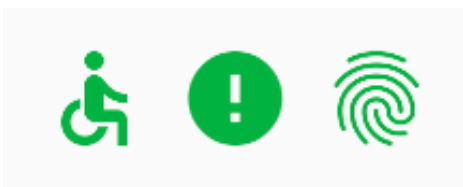
```
flutter:  
  uses-material-design: true
```

Material Design所有图标可以在其官网查看：<https://material.io/tools/icons/>

我们看一个简单的例子：

```
String icons = "";  
// accessible: &#xE914; or 0xE914 or E914  
icons += "\uE914";  
// error: &#xE000; or 0xE000 or E000  
icons += " \uE000";  
// fingerprint: &#xE90D; or 0xE90D or E90D  
icons += " \uE90D";  
  
Text(icons,  
  style: TextStyle(  
    fontFamily: "MaterialIcons",  
    fontSize: 24.0,  
    color: Colors.green  
  ),  
);
```

运行效果如下：



通过这个示例可以看到，使用图标就像使用文本一样，但是这种方式需要我们提供每个图标的码点，这并对开发者不友好，所以，Flutter封装了一个 `IconData` 和 `Icon` 来专门显示字体图标，上面的例子也可以用如下方式实现：

```
Row(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: <Widget>[  
    Icon(Icons.accessible,color: Colors.green,),  
    Icon(Icons.error,color: Colors.green,),  
    Icon(Icons.fingerprint,color: Colors.green,),  
  ],  
)
```

`Icons` 类中包含了所有Material Design图标的 `IconData` 静态变量定义。

使用自定义字体图标

我们也可以使用自定义字体图标。iconfont.cn上有很多字体图标素材，我们可以选择自己需要的图标打包下载后，会生成一些不同格式的字体文件，在Flutter中，我们使用ttf格式即可。

假设我们项目中需要使用一个书籍图标和微信图标，我们打包下载后导入：

1. 导入字体图标文件；这一步和导入字体文件相同，假设我们的字体图标文件保存在项目根目录下，路径为"fonts/iconfont.ttf"：

```
fonts:  
  - family: myIcon #指定一个字体名  
    fonts:  
      - asset: fonts/iconfont.ttf
```

2. 为了使用方便，我们定义一个 `MyIcons` 类，功能和 `Icons` 类一样：将字体文件中的所有图标都定义成静态变量：

```

class MyIcons{
  // fork 图标
  static const IconData book = const IconData(
    0xe614,
    fontFamily: 'myIcon',
    matchTextDirection: true
  );
  // 微信图标
  static const IconData wechat = const IconData(
    0xec7d,
    fontFamily: 'myIcon',
    matchTextDirection: true
  );
}

```

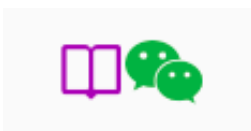
3. 使用

```

Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Icon(MyIcons.book,color: Colors.purple,),
    Icon(MyIcons.wechat,color: Colors.green,),
  ],
)

```

运行后效果如下：



单选开关和复选框

Material widgets库中提供了Material风格的单选开关Switch和复选框Checkbox，它们都是继承自StatelessWidget，所以它们本身不会保存当前选择状态，所以一般都是在父widget中管理选中状态。当用户点击Switch或Checkbox时，它们会触发onChanged回调，我们可以在此回调中处理选中状态改变逻辑。我们看一个简单的示例：

```

class SwitchAndCheckBoxTestRoute extends StatefulWidget {
  @override
  _SwitchAndCheckBoxTestRouteState createState() => new
  _SwitchAndCheckBoxTestRouteState();
}

class _SwitchAndCheckBoxTestRouteState extends
State<SwitchAndCheckBoxTestRoute> {
  bool _switchSelected=true; // 维护单选开关状态
  bool _checkboxSelected=true; // 维护复选框状态
  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Switch(
          value: _switchSelected, // 当前状态
          onChanged: (value){
            // 重新构建页面
            setState(() {
              _switchSelected=value;
            });
          },
        ),
        Checkbox(
          value: _checkboxSelected,
          activeColor: Colors.red, // 选中时的颜色
          onChanged: (value){
            setState(() {
              _checkboxSelected=value;
            });
          },
        ),
      ],
    );
  }
}

```

上面代码中，由于要维护Switch和Checkbox状态，所以SwitchAndCheckBoxTestRoute继承自StatefulWidget。在其build方法中分别构建了一个Switch和Checkbox，初始状态都为选中状态，当用户点击时，会将状态置反，然后回调用 `setState()` 通知framework重新构建UI。

属性及外观

Switch和Checkbox属性比较简单，读者可以查看API文档，它们都有一个 `activeColor` 属性，用于设置激活态的颜色。至于大小，到目前为止，Checkbox的大小是固定的，无法自定义，而Switch只能定义宽度，高度也是固定的。值得一提的是Checkbox有一个属性 `tristate`，表示是否为三态，其默认值为 `false`，这时Checkbox有两种状态即“选中”和“不选中”，对应的value值为 `true` 和 `false`；如果其值为 `true` 时，value的值会增加一个状态 `null`，读者可以自行了解。

总结

通过Switch和Checkbox我们可以看到，虽然它们本身是与状态（是否选中）关联的，但它们却不是自己来维护状态，而是需要父widget来管理状态，然后当用户点击时，再通过事件通知给父widget，这样是合理的，因为Switch和Checkbox是否选中本就与用户数据关联，而这些用户数据也不可能是它们的私有状态。我们在自定义widget时也应该思考一下哪种状态的管理方式最为合理。

输入框及表单

Material widget库中提供了丰富的输入框及表单Widget。下面我们分别介绍一下。

TextField

TextField用于文本输入，它提供了很多属性，我们先简单介绍一下主要属性的作用，然后通过几个示例来演示一下关键属性的用法。

```

const TextField({
  ...
  TextEditingController controller,
  FocusNode focusNode,
  InputDecoration decoration = const InputDecoration(),
  TextInputType keyboardType,
  TextInputAction textInputAction,
  TextStyle style,
  TextAlign textAlign = TextAlign.start,
  bool autofocus = false,
  bool obscureText = false,
  int maxLines = 1,
  int maxLength,
  bool maxLengthEnforced = true,
  ValueChanged<String> onChanged,
  VoidCallback onEditingComplete,
  ValueChanged<String> onSubmitted,
  List<TextInputFormatter> inputFormatters,
  bool enabled,
  this.cursorWidth = 2.0,
  this.cursorRadius,
  this.cursorColor,
  ...
})

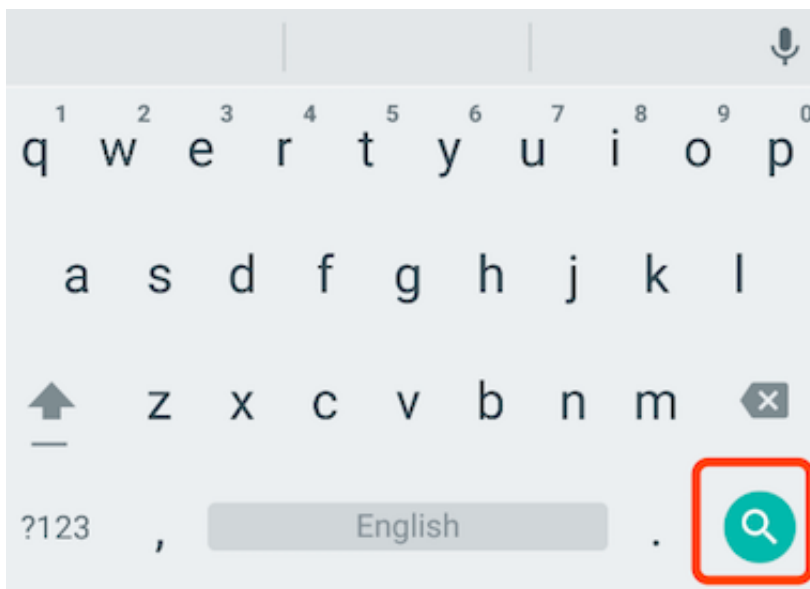
```

- controller：编辑框的控制器，通过它可以设置/获取编辑框的内容、选择编辑内容、监听编辑文本改变事件。大多数情况下我们都需要显式提供一个controller来与文本框交互。如果没有提供controller，则TextField内部会自动创建一个。
- focusNode：用于控制TextField是否占有当前键盘的输入焦点。它是我们和键盘交互的一个handle。
- InputDecoration：用于控制TextField的外观显示，如提示文本、背景颜色、边框等。
- keyboardType：用于设置该输入框默认的键盘输入类型，取值如下：

TextInputType枚举值	含义
text	文本输入键盘
multiline	多行文本，需和maxLines配合使用(设为null或大于1)

number	数字；会弹出数字键盘
phone	优化后的电话号码输入键盘；会弹出数字键盘并显示"* #"
datetime	优化后的日期输入键盘；Android上会显示“: -”
emailAddress	优化后的电子邮件地址；会显示“@ .”
url	优化后的url输入键盘； 会显示“/ .”

- `textInputAction`: 键盘动作按钮图标(即回车键位图标)，它是一个枚举值，有多个可选值，全部的取值列表读者可以查看API文档，下面是当值为 `TextInputAction.search` 时，原生Android系统下键盘样式：



- style: 正在编辑的文本样式。
- textAlign: 输入框内编辑文本在水平方向的对齐方式。
- autofocus: 是否自动获取焦点。
- obscureText: 是否隐藏正在编辑的文本，如用于输入密码的场景等，文本内容会用“•”替换。
- maxLines: 输入框的最大行数，默认为1；如果为 `null`，则无行数限制。
- maxLength和maxLengthEnforced: maxLength代表输入框文本的最大长度，设置后输入框右下角会显示输入的文本计数。maxLengthEnforced决定当输入文本长度超过maxLength时是否阻止输入，为true时会阻止输入，为false时不会阻止输入但输入框会变红。
- onChange: 输入框内容改变时的回调函数；注：内容改变事件也可以通过controller来监听。
- onEditingComplete和onSubmitted: 这两个回调都是在输入框输入完成时触发，比如按了键盘的完成键（对号图标）或搜索键（🔍图标）。不同的是两个回调签名不同，onSubmitted回调是 `ValueChanged<String>` 类型，它接收当前输入内容做为参数，而onEditingComplete不接收参数。
- inputFormatters: 用于指定输入格式；当用户输入内容改变时，会根据指定的格式来校验。
- enable: 如果为 `false`，则输入框会被禁用，禁用状态不接收输入和事件，同时显示禁用态样式（在其decoration中定义）。
- cursorWidth、cursorRadius和cursorColor: 这三个属性是用于自定义输入框光标宽度、圆角和颜色的。

示例：登录输入框

布局


```

Column(
  children: <Widget>[
    TextField(
      autofocus: true,
      decoration: InputDecoration(
        labelText: "用户名",
        hintText: "用户名或邮箱",
        prefixIcon: Icon(Icons.person)
      ),
    ),
    TextField(
      decoration: InputDecoration(
        labelText: "密码",
        hintText: "您的登录密码",
        prefixIcon: Icon(Icons.lock)
      ),
      obscureText: true,
    ),
  ],
);

```



获取输入内容

获取输入内容有两种方式：

1. 定义两个变量，用于保存用户名和密码，然后在onChange触发时，各自保存一下输入内容。
2. 通过controller直接获取。

第一种方式比较简单，不在举例，我们来重点看一下第二种方式，我们以用户名输入框举例：

定义一个controller：

```
//定义一个controller
TextEditingController _unameController=new TextEditingController();
```

然后设置输入框controller:

```
TextField(
  autofocus: true,
  controller: _unameController, //设置controller
  ...
)
```

通过controller获取输入框内容

```
print(_unameController.text)
```

监听文本变化

监听文本变化也有两种方式:

1. 设置onChange回调, 如:

```
TextField(
  autofocus: true,
  onChanged: (v) {
    print("onChange: $v");
  }
)
```

2. 通过controller监听, 如:

```
@override
void initState() {
  // 监听输入改变
  _unameController.addListener((){
    print(_unameController.text);
  });
}
```

两种方式相比，onChanged是专门用于监听文本变化，而controller的功能却多一些，除了能监听文本变化外，它还可以设置默认值、选择文本，下面我们看一个例子：

创建一个controller:

```
TextEditingController _selectionController = new  
TextEditingController();
```

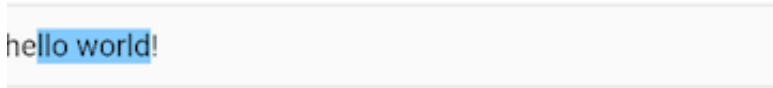
设置默认值，并从第三个字符开始选中后面的字符

```
_selectionController.text="hello world!";  
_selectionController.selection=TextSelection(  
  baseOffset: 2,  
  extentOffset: _selectionController.text.length  
);
```

设置controller:

```
TextField(  
  controller: _selectionController,  
)
```

运行效果如下：

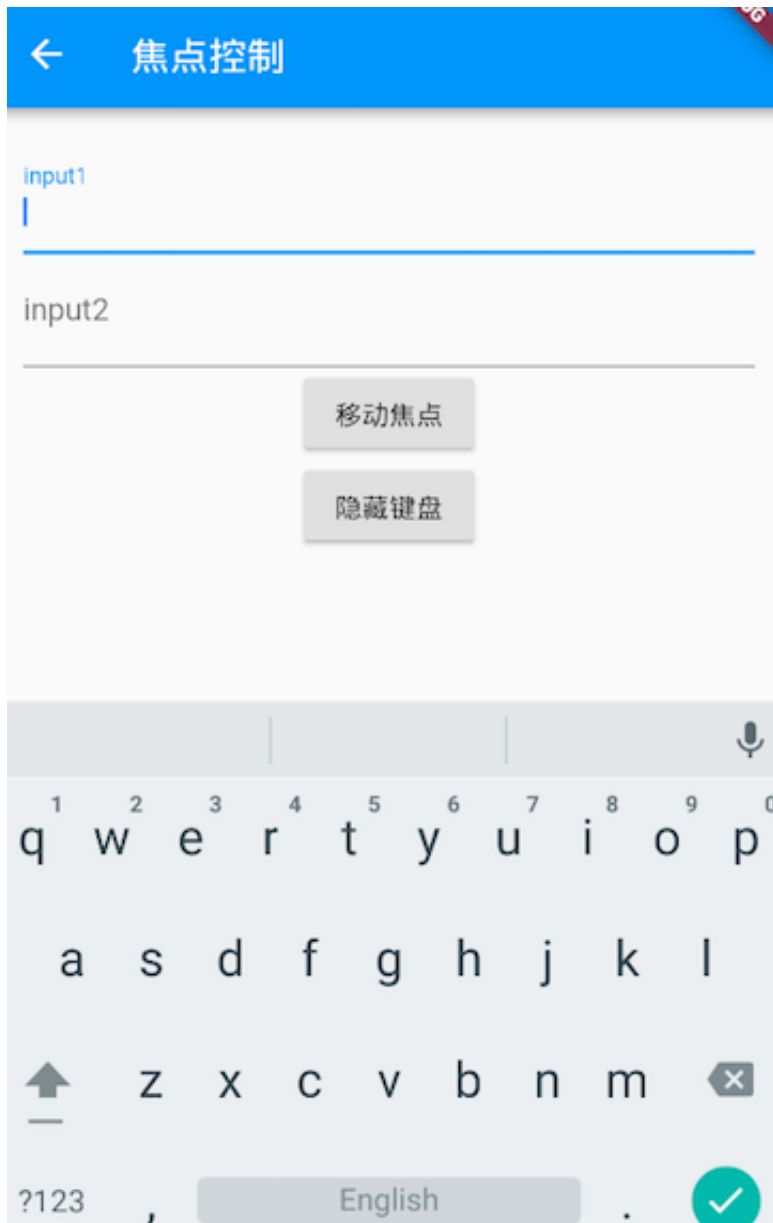


控制焦点

焦点可以通过FocusNode和FocusScopeNode来控制，默认情况下，焦点由FocusScope来管理，它代表焦点控制范围，可以在这个范围内可以通过FocusScopeNode在输入框之间移动焦点、设置默认焦点等。我们可以通过 FocusScope.of(context) 来获取widget树中默认的FocusScopeNode。下面看一个示例，在此示例中创建两个TextField，第一个自动获取焦点，然后创建两个按钮：

- 点击第一个按钮可以将焦点从第一个TextField挪到第二个TextField。
- 点击第二个按钮可以关闭键盘。

界面如下：



代码如下：

```
class FocusTestRoute extends StatefulWidget {  
  @override  
  _FocusTestRouteState createState() => new _FocusTestRouteState();  
}
```

```

class _FocusTestRouteState extends State<FocusTestRoute> {
  FocusNode focusNode1 = new FocusNode();
  FocusNode focusNode2 = new FocusNode();
  FocusScopeNode focusScopeNode;

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          TextField(
            autofocus: true,
            focusNode: focusNode1, // 关联 focusNode1
            decoration: InputDecoration(
              labelText: "input1"
            ),
          ),
          TextField(
            focusNode: focusNode2, // 关联 focusNode2
            decoration: InputDecoration(
              labelText: "input2"
            ),
          ),
          Builder(builder: (ctx) {
            return Column(
              children: <Widget>[
                RaisedButton(
                  child: Text("移动焦点"),
                  onPressed: () {
                    // 将焦点从第一个TextField移到第二个TextField
                    // 这是一种写法
                    FocusScope.of(context).requestFocus(focusNode2);
                    // 这是第二种写法
                    if(null == focusScopeNode){
                      focusScopeNode = FocusScope.of(context);
                    }
                    focusScopeNode.requestFocus(focusNode2);
                  },
                ),
                RaisedButton(
                  child: Text("隐藏键盘"),
                  onPressed: () {
                    // 当所有编辑框都失去焦点时键盘就会收起
                    focusNode1.unfocus();
                    focusNode2.unfocus();
                  },
                ),
              ],
            );
          })
        ],
      ),
    );
  }
}

```

```

        ),
        1,
    );
    },
    ),
    1,
    ),
    );
}
}

```

FocusNode和FocusScopeNode还有一些其它的方法，详情可以查看API文档。

监听焦点状态改变事件

FocusNode继承自ChangeNotifier，通过FocusNode可以监听焦点的改变事件，如：

```

...
// 创建 focusNode
FocusNode focusNode = new FocusNode();
...
// focusNode绑定输入框
TextField(focusNode: focusNode);
...
// 监听焦点变化
focusNode.addListener((){
    print(focusNode.hasFocus);
});

```

获得焦点时 `focusNode.hasFocus` 值为 `true`，失去焦点时为 `false`。

自定义样式

虽然我们可以通过`decoration`属性来定义输入框样式，但是有一些样式如下划线默认颜色及宽度都是不能直接自定义的，下面的代码没有效果：

```
TextField(  
  ...  
  decoration: InputDecoration(  
    border: UnderlineInputBorder(  
      // 下面代码没有效果  
      borderSide: BorderSide(  
        color: Colors.red,  
        width: 5.0  
      )),  
    prefixIcon: Icon(Icons.person)  
  ),  
),
```

之所以如此，是由于TextField在绘制下划线时使用的颜色是主题色里面的 `hintColor`，但提示文本颜色也是用的 `hintColor`，如果我们直接修改 `hintColor`，那么下划线和提示文本的颜色都会变。值得高兴的是decoration中可以设置 `hintStyle`，它可以覆盖 `hintColor`，并且主题中可以通过 `inputDecorationTheme` 来设置输入框默认的decoration。所以我们可以通过主题来自定义，代码如下：

```

Theme(
  data: Theme.of(context).copyWith(
    hintColor: Colors.grey[200], // 定义下划线颜色
    inputDecorationTheme: InputDecoration(
      labelStyle: TextStyle(color: Colors.grey), // 定义label字体样
      hintStyle: TextStyle(color: Colors.grey, fontSize:
14.0) // 定义提示文本样式
    ),
  ),
  child: Column(
    children: <Widget>[
      TextField(
        controller: _pwdController,
        decoration: InputDecoration(
          labelText: "密码",
          hintText: "您的登录密码",
          prefixIcon: Icon(Icons.lock)
        ),
        obscureText: true,
      ),
      TextField(
        decoration: InputDecoration(
          prefixIcon: Icon(Icons.search),
          labelText: "密码",
          hintText: "输入关键字搜索",
          hintStyle: TextStyle(color: Colors.grey, fontSize:
13.0)
        ),
      ),
    ],
  ),
)

```

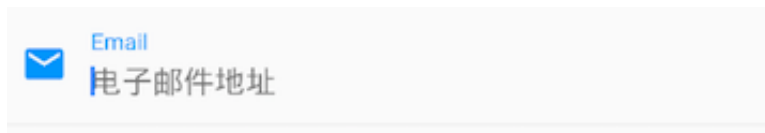
运行效果如下：



我们成功的自定义了下划线颜色和提问文字样式，细心的读者可能已经发现，通过这种方式自定义后，输入框在获取焦点时，labelText不会高亮显示了，正如上图中的"用户名"本应该显示蓝色，但现在却显示为灰色，并且我们还是无法定义下划线宽度。另一种灵活的方式是直接隐藏掉TextField本身的下划线，然后通过Container去嵌套定义样式，如：

```
Container(  
  child: TextField(  
    keyboardType: TextInputType.emailAddress,  
    decoration: InputDecoration(  
      labelText: "Email",  
      hintText: "电子邮件地址",  
      prefixIcon: Icon(Icons.email),  
      border: InputBorder.none // 隐藏下划线  
    )  
  ),  
  decoration: BoxDecoration(  
    // 下滑线浅灰色，宽度1像素  
    border: Border(bottom: BorderSide(color: Colors.grey[200],  
width: 1.0))  
  ),  
)
```

运行效果：



通过这种widget组合的方式，也可以定义背景圆角等。一般来说，优先通过decoration来自定义样式，如果decoration实现不了，再用widget组合的方式。

思考题：在这个示例中，下划线颜色是固定的，所以获得焦点后颜色仍然为灰色，如何实现点击后下滑线也变色呢？

表单Form

实际业务中，在正式向服务器提交数据前，都会对各个输入框数据进行合法性校验，但是对每一个TextField都分别进行校验将会是一件很麻烦的事。还有，如果用户想清除一组TextField的内容，除了一个一个清除有没有什么更好的办法呢？为此，Flutter提供了一个Form widget，它可以对输入框进行分组，然后进行一些统一操作，如输入内容校验、输入框重置以及输入内容保存。

Form

Form继承自StatefulWidget对象，它对应的状态类为FormState。我们先看看Form类的定义：

```
Form({
  @required Widget child,
  bool autovalidate = false,
  WillPopCallback onWillPop,
  VoidCallback onChanged,
})
```

- autovalidate：是否自动校验输入内容；当为 true 时，每一个子FormField内容发生变化时都会自动校验合法性，并直接显示错误信息。否则，需要通过调用 FormState.validate() 来手动校验。
- onWillPop：决定Form所在的路由是否可以直接返回（如点击返回按钮），该回调返回一个 Future 对象，如果Future的最终结果是false，则当前路由不会返回；如果为 true，则会返回到上一个路由。此属性通常用于拦截返回按钮。
- onChanged：Form的任意一个子FormField内容发生变化时会触发此回调。

FormField

Form的子元素必须是FormField类型，FormField是一个抽象类，定义几个属性，FormState内部通过它们来完成操作，FormField部分定义如下：

```
const FormField({
  ...
  FormFieldSetter<T> onSave, // 保存回调
  FormFieldValidator<T> validator, // 验证回调
  T initialValue, // 初始值
  bool autovalidate = false, // 是否自动校验。
})
```

为了方便使用，Flutter提供了一个TextFormField widget，它继承自FormField类，也是TextField的一个包装类，所以除了FormField定义的属性之外，它还包括TextField的属性。

FormState

FormState为Form的State类，可以通过 `Form.of()` 或 `GlobalKey` 获得。我们可以通过它来对Form的子孙FormField进行统一操作。我们看看其常用的三个方法：

- `FormState.validate()`：调用此方法后，会调用Form子孙FormField的 `validate` 回调，如果有一个校验失败，则返回false，所有校验失败项都会返回用户返回的错误提示。
- `FormState.save()`：调用此方法后，会调用Form子孙FormField的 `save` 回调，用于保存表单内容
- `FormState.reset()`：调用此方法后，会将子孙FormField的内容清空。

示例

我们修改一下上面用户登录的示例，在提交之前校验：

1. 用户名不能为空，如果为空则提示“用户名不能为空”。
2. 密码不能小于6位，如果小于6为则提示“密码不能少于6位”。

完整代码：

```
class FormTestRoute extends StatefulWidget {
  @override
  _FormTestRouteState createState() => new _FormTestRouteState();
}

class _FormTestRouteState extends State<FormTestRoute> {
  TextEditingController _unameController = new
  TextEditingController();
  TextEditingController _pwdController = new
  TextEditingController();
  GlobalKey _formKey= new GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return PageScaffold(
      title: "Form Test",
      body: Padding(
```

```

padding: const EdgeInsets.symmetric(vertical: 16.0,
horizontal: 24.0),
child: Form(
  key: _formKey, // 设置globalKey, 用于后面获取FormState
  autovalidate: true, // 开启自动校验
  child: Column(
    children: <Widget>[
      TextFormField(
        autofocus: true,
        controller: _unameController,
        decoration: InputDecoration(
          labelText: "用户名",
          hintText: "用户名或邮箱",
          icon: Icon(Icons.person)
        ),
        // 校验用户名
        validator: (v) {
          return v
            .trim()
            .length > 0 ? null : "用户名不能为空";
        }
      ),
      TextFormField(
        controller: _pwdController,
        decoration: InputDecoration(
          labelText: "密码",
          hintText: "您的登录密码",
          icon: Icon(Icons.lock)
        ),
        obscureText: true,
        // 校验密码
        validator: (v) {
          return v
            .trim()
            .length > 5 ? null : "密码不能少于6位";
        }
      ),
    ],
    // 登录按钮
  ),
  child: Padding(
    padding: const EdgeInsets.only(top: 28.0),
    child: Row(
      children: <Widget>[
        Expanded(
          child: RaisedButton(
            padding: EdgeInsets.all(15.0),
            child: Text("登录"),

```

```
color: Theme
    .of(context)
    .primaryColor,
textColor: Colors.white,
onPressed: () {
    //在这里不能通过此方式获取FormState, context不
    //print(Form.of(context));

    // 通过_formKey.currentState 获取FormState

    // 调用validate()方法校验用户名密码是否合法, 校
    // 验通过后提交数据。
    if(_formKey.currentState as
FormState).validate()){
        //验证通过提交数据
    }
},
),
),
),
),
),
),
),
),
);
}
}
```

运行后：

注意，登录按钮的onPressed方法中不能通过 `Form.of(context)` 来获取，原因是，此处的context为FormTestRoute的context，而 `Form.of(context)` 是根据所指定context向根去查找，而FormState是在FormTestRoute的子树中，所以不行。正确的做法是通过Builder来构建登录按钮，Builder会将widget节点的context作为回调参数：

```
Expanded(  
  // 通过Builder来获取RaisedButton所在widget树的真正context(Element)  
  child: Builder(builder: (context){  
    return RaisedButton(  
      ...  
      onPressed: () {  
        // 由于本widget也是Form的子代widget，所以可以通过下面方式获取  
        // FormState  
        if(Form.of(context).validate()){  
          // 验证通过提交数据  
        }  
      },  
    );  
  })  
)
```

其实context正是操作Widget所对应的Element的一个接口，由于Widget树对应的Element都是不同的，所以context也都是不同的，有关context的更多内容会在后面高级部分详细讨论。Flutter中有很多“of(context)”这种方法，在使用时读者一定要注意context是否正确。

进度条

和大多数UI库一样，Flutter也提供了两种进度指示器。一种是水平方向线性的LinearProgressIndicator，另一种是圆形的CircularProgressIndicator。两种指示器都支持循环模式和精确精度：

- 循环模式：循环模式会有一个循环的动画，用于无法获得精确进度的场景，循环模式为默认模式。
- 精确模式：用于可以获得精确进度的场景，比如大多数文件下载场景。精确模式需要提供 value 属性值，它的取值范围是[0,1]，代表当前进度的百分比。

水平进度条

LinearProgressIndicator是水平方向线性的进度条，下面分别给出循环模式和精确模式的示例：

```
Column(  
  children: <Widget>[  
    // 循环模式  
    LinearProgressIndicator(),  
    Padding(  
      padding: const EdgeInsets.symmetric(vertical: 8.0),  
      child: LinearProgressIndicator(  
        backgroundColor: Colors.grey[200],  
        value: .2, // 精确模式，进度20%  
      ),  
    ),  
  ],  
)
```

运行效果如下：



圆形进度条

CircularProgressIndicator是圆形进度条，下面分别给出循环模式和精确模式的示例：

```
Column(  
  children: <Widget>[  
    // 循环模式  
    CircularProgressIndicator(),  
    Padding(  
      padding: const EdgeInsets.only(top: 20.0),  
      child: CircularProgressIndicator(  
        value: .5, // 精确模式, 进度50%  
      ),  
    ],  
)
```



CircularProgressIndicator支持一个strokeWidth属性，表示圆形进度条画笔的宽度，默认为4像素（逻辑像素），我们看看strokeWidth值为2.0时的效果：



代码：


```
CircularProgressIndicator(  
    strokeWidth: 2.0,  
)
```

属性

由于LinearProgressIndicator和CircularProgressIndicator都继承自ProgressIndicator，所以它们都支持ProgressIndicator定义的一些属性，下面我们来看看常用属性：

```
ProgressIndicator({  
    ...  
    double value,  
    Color backgroundColor,  
    Animation<Color> valueColor,  
})
```

- value：进度条当前的精确进度，如果为 `null`，则精度条为循环模式。
- backgroundColor：进度条背景色，目前只有在LinearProgressIndicator设置有效，CircularProgressIndicator暂不支持背景色。
- valueColor：LinearProgressIndicator和CircularProgressIndicator的进度指示色都是一个动画值，也就是说指示色是可以执行动画的，只不过默认是使用了固定色(主题色)而已。我们可以指定一个 `Animation<Color>` 来为指示色执行动画：

```
CircularProgressIndicator(  
    valueColor: ColorTween(begin: Colors.blue, end: Colors.green)  
        .animate(_controller),  
)
```

这个示例运行后，圆形精度条的颜色会在蓝色和绿色之间渐变。关于动画的详细内容，请参考后面“动画”一章内容。

自定义进度条样式

定义进度条大小

进度条本身没有控制大小的属性，LinearProgressIndicator的默认高度是6像素，宽度自适应。CircularProgressIndicator默认的宽度和高度皆为36像素。我们可以通过指定父容器的宽高来自定义其大小，如：

```
...
  SizedBox(
    height: 2.0, // 高度指定为2像素
    child: LinearProgressIndicator()
  ),

  SizedBox(
    height: 80.0, // 高度和宽度均指定为80像素
    width: 80.0,
    child: CircularProgressIndicator(),
  )
...
```



自定义颜色

假如我们想指定进度条颜色为绿色，有两种方法可以自定义进度条颜色，：

- 通过主题来自定义：

```
Theme(  
  data: ThemeData(  
    primarySwatch: Colors.green,  
  ),  
  child: LinearProgressIndicator(  
    value: .2,  
  ),  
)
```

运行效果如下：



通过主题定义widget样式的方法在flutter中非常常见，我们将在后面专门“主题”章节详细介绍。

- 通过valueColor属性：

我们可以通过给valueColor指定一个 `AlwaysStoppedAnimation<Color>` 对象来使用固定的颜色（而非动画颜色）。

```
CircularProgressIndicator(  
  valueColor: new AlwaysStoppedAnimation<Color>(Colors.green),  
  value: .9,  
)
```

运行效果如下：

