功能型Widget简介

功能型Widget指的是不会影响UI布局及外观的Widget，它们通常具有一定的功能，如事件监听、数据存储等，我们之前介绍过的FocusScope（焦点控制）、PageStorage（数据存储）、NotificationListener（事件监听）都属于功能型Widget。由于Widget是Flutter的一等公民，功能型Widget非常多，我们不会去一一介绍，本章中主要介绍几种常用的功能型Widget。

本章目录

导航返回拦截WillPopScope

为了避免用户误触返回按钮而导致APP退出，在很多APP中都拦截了用户点击返回键的按钮，当用户在某一个时间段内点击两次时，才会认为用户是要退出（而非误触）。Flutter中可以通过WillPopScope来实现返回按钮拦截，我们看看WillPopScope的默认构造函数：

```
const WillPopScope({
  ...
  @required WillPopCallback onWillPop,
  @required Widget child
})
```

onWillPop是一个回调函数，当用户点击返回按钮时调用（包括导航返回按钮及Android物理返回按钮），该回调需要返回一个Future对象，如果返回的Future最终值为`false`时，则当前路由不出栈(不会返回)，最终值为`true`时，当前路由出栈退出。我们需要提供这个回调来决定是否退出。

示例

为了防止用户误触返回键退出，我们拦截返回事件，当用户在1秒内点击两次返回按钮时，则退出，如果间隔超过1秒则不退出，并重新记时。代码如下：

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

class WillPopScopeTestRoute extends StatefulWidget {
  @override
  WillPopScopeTestRouteState createState() {
    return new WillPopScopeTestRouteState();
  }
}

class WillPopScopeTestRouteState extends
State<WillPopScopeTestRoute> {
  DateTime _lastPressedAt; //上次点击时间

  @override
  Widget build(BuildContext context) {
    return new WillPopScope(
        onWillPop: () async {
          if (_lastPressedAt == null ||
              DateTime.now().difference(_lastPressedAt) >
Duration(seconds: 1)) {
            //两次点击间隔超过1秒则重新计时
            _lastPressedAt = DateTime.now();
            return false;
          }
          return true;
        },
        child: Container(
          alignment: Alignment.center,
          child: Text("1秒内连续按两次返回键退出"),
        )
    );
  }
}
```

读者可以运行示例看看效果。

InheritedWidget

InheritedWidget是Flutter中非常重要的一个功能型Widget，它可以高效的将数据在Widget树中向下传递、共享，这在一些需要在Widget树中共享数据的场景中非常方便，如Flutter中，正是通过InheritedWidget来共享应用主题(Theme)和Locale(当前语言环境)信息的。

> InheritedWidget和React中的context功能类似，和逐级传递数据相比，它们能实现组件跨级传递数据。InheritedWidget的在Widget树中数据传递方向是从上到下的，这和Notification的传递方向正好相反。

didChangeDependencies

在介绍StatefulWidget时，我们提到State对象有一个回调`didChangeDependencies`，它会在“依赖”发生变化时被Flutter Framework调用。而这个“依赖”指的就是是否使用了父widget中InheritedWidget的数据，如果使用了，则代表有依赖，如果没有使用则代表没有依赖。这种机制可以使子组件在所依赖的主题、locale等发生变化时有机会来做一些事情。

我们看一下之前“计数器”示例应用程序的InheritedWidget版本，需要说明的是，本例主要是为了演示InheritedWidget的功能特性，并不是计数器的推荐实现方式。

首先，我们通过继承InheritedWidget，将当前计数器点击次数保存在ShareDataWidget的`data`属性中：

```dart
class ShareDataWidget extends InheritedWidget {
  ShareDataWidget({
    @required this.data,
    Widget child
  }) :super(child: child);

  int data; //需要在子树中共享的数据，保存点击次数

  //定义一个便捷方法，方便子树中的widget获取共享数据
  static ShareDataWidget of(BuildContext context) {
    return context.inheritFromWidgetOfExactType(ShareDataWidget);
  }

  //该回调决定当data发生变化时，是否通知子树中依赖data的Widget
  @override
  bool updateShouldNotify(ShareDataWidget old) {
    //如果返回false，则子树中依赖(build函数中有调用)本widget
    //的子widget的`state.didChangeDependencies`会被调用
    return old.data != data;
  }
}
```

然后我们实现一个子widget _TestWidget，在其build方法中引用ShareDataWidget中的数据；同时，在其`didChangeDependencies()`回调中打印日志：

```
class _TestWidget extends StatefulWidget {
  @override
  __TestWidgetState createState() => new __TestWidgetState();
}

class __TestWidgetState extends State<_TestWidget> {
  @override
  Widget build(BuildContext context) {
    //使用InheritedWidget中的共享数据
    return Text(ShareDataWidget
        .of(context)
        .data
        .toString());
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    //父或祖先widget中的InheritedWidget改变(updateShouldNotify返回true)
时会被调用。
    //如果build中没有依赖InheritedWidget，则此回调不会被调用。
    print("Dependencies change");
  }
}
```

最后，我们创建一个按钮，每点击一次，就将ShareDataWidget的值自增：

```
class InheritedWidgetTestRoute extends StatefulWidget {
  @override
  _InheritedWidgetTestRouteState createState() => new
_InheritedWidgetTestRouteState();
}

class _InheritedWidgetTestRouteState extends
State<InheritedWidgetTestRoute> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return  Center(
      child: ShareDataWidget( //使用ShareDataWidget
        data: count,
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Padding(
              padding: const EdgeInsets.only(bottom: 20.0),
              child: _TestWidget(),//子widget中依赖ShareDataWidget
            ),
            RaisedButton(
              child: Text("Increment"),
              //每点击一次，将count自增，然后重新build,ShareDataWidget的
data将被更新
              onPressed: () => setState(() => ++count),
            )
          ],
        ),
      ),
    );
  }
}
```

运行后界面如下：

每点击一次按钮，计数器就会自增，控制台就会打印一句日志：

```
I/flutter ( 8513): Dependencies change
```

可见依赖发生变化后，其`didChangeDependencies()`会被调用。但是读者要注意，如果_TestWidget的build方法中没有使用ShareDataWidget的数据，那么它的`didChangeDependencies()`将不会被调用，因为它并没有依赖ShareDataWidget。

> 思考题：Flutter framework是怎么知道子widget有没有依赖InheritedWidget的?

应该在didChangeDependencies()中做什么?

一般来说，子widget很少会重写此方法，因为在依赖改变后framework也都会调用`build()`方法。但是，如果你需要在依赖改变后执行一些昂贵的操作，比如网络请求，这时最好的方式就是在此方法中执行，这样可以每次`build()`都执行一次。

主题

Theme Widget可以为Material APP定义主题数据（ThemeData），Material组件库里很多Widget都使用了主题数据，如导航栏颜色、标题字体、Icon样式等。Theme内会使用InheritedWidget来为其子树Widget共享样式数据。

ThemeData

ThemeData是Material Design Widget库的主题数据，Material库的Widget需要遵守相应的设计规范，而这些规范可自定义部分都定义在ThemeData，所以我们可以通过ThemeData来自定义应用主题。我们可以通过`Theme.of`方法来获取当前的ThemeData。

> 注意，Material Design 设计规范中有些是不能自定义的，如导航栏高度，ThemeData只包含了可自定义部分。

我们看看ThemeData部分数据：

```
ThemeData({
  Brightness brightness, //深色还是浅色
  MaterialColor primarySwatch, //主题颜色样本，见下面介绍
  Color primaryColor, //主色，决定导航栏颜色
  Color accentColor, //次级色，决定大多数Widget的颜色，如进度条、开关等。
  Color cardColor, //卡片颜色
  Color dividerColor, //分割线颜色
  ButtonThemeData buttonTheme, //按钮主题
  Color cursorColor, //输入框光标颜色
  Color dialogBackgroundColor,//对话框背景颜色
  String fontFamily, //文字字体
  TextTheme textTheme,// 字体主题，包括标题、body等文字样式
  IconThemeData iconTheme, // Icon的默认样式
  TargetPlatform platform, //指定平台，应用特定平台控件风格
  ...
})
```

上面只是ThemeData的一小部分属性，完整列表读者可以查看SDK定义。上面属性中需要说明的是`primarySwatch`，它是主题颜色的一个"样本"，通过这个样本可以在一些条件下生成一些其它的属性，例如，如果没有指定`primaryColor`，并且当前主题不是深色主题，那么`primaryColor`就会默认为`primarySwatch`指定的颜色，还有一些相似的属性如`accentColor`、`indicatorColor`等也会受`primarySwatch`影响。

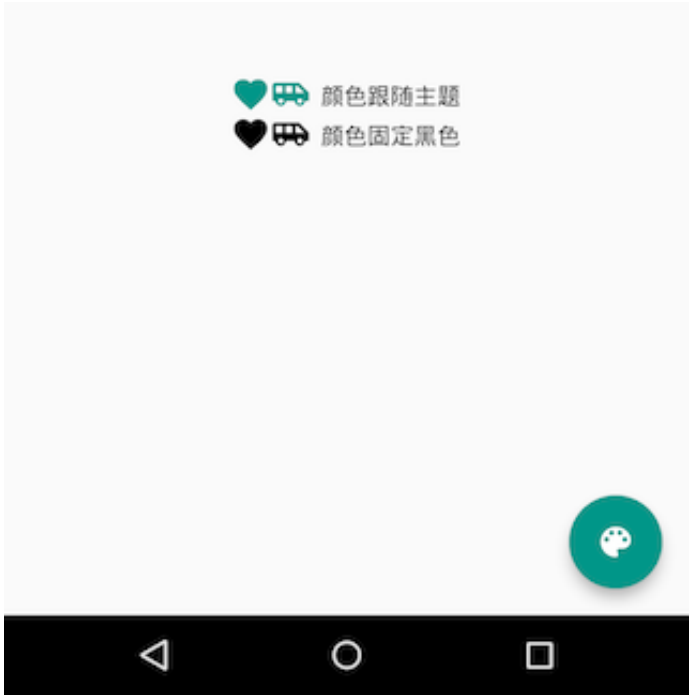
示例

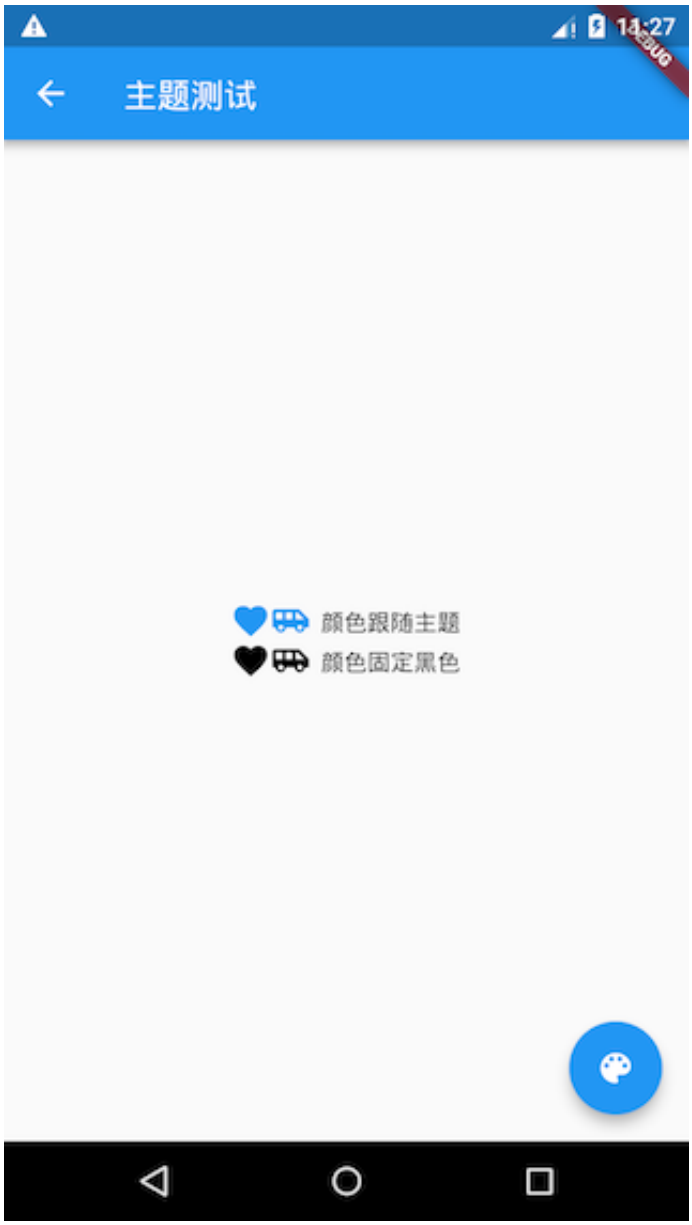
我们实现一个路由换肤功能：

```
class ThemeTestRoute extends StatefulWidget {
  @override
  _ThemeTestRouteState createState() => new _ThemeTestRouteState();
}

class _ThemeTestRouteState extends State<ThemeTestRoute> {
  Color _themeColor = Colors.teal; //当前路由主题色

  @override
  Widget build(BuildContext context) {
    ThemeData themeData = Theme.of(context);
    return Theme(
      data: ThemeData(
          primarySwatch: _themeColor, //用于导航
栏、FloatingActionButton的背景色等
          iconTheme: IconThemeData(color: _themeColor) //用于Icon颜色
      ),
      child: Scaffold(
        appBar: AppBar(title: Text("主题测试")),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            //第一行Icon使用主题中的iconTheme
            Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Icon(Icons.favorite),
                  Icon(Icons.airport_shuttle),
                  Text("  颜色跟随主题")
                ]
            ),
            //为第二行Icon自定义颜色（固定为黑色）
```

```
            Theme(
              data: themeData.copyWith(
                iconTheme: themeData.iconTheme.copyWith(
                    color: Colors.black
                ),
              ),
              child: Row(
                  mainAxisAlignment: MainAxisAlignment.center,
                  children: <Widget>[
                    Icon(Icons.favorite),
                    Icon(Icons.airport_shuttle),
                    Text("  颜色固定黑色")
                  ]
              ),
            ),
          ],
        ),
        floatingActionButton: FloatingActionButton(
            onPressed: () =>  //切换主题
                setState(() =>
                _themeColor =
                _themeColor == Colors.teal ? Colors.blue :
Colors.teal
                ),
            child: Icon(Icons.palette)
        ),
      ),
    );
  }
}
```

运行后点击右下角悬浮按钮则可以切换主题：


颜色跟随主题
颜色固定黑色



需要注意的有三点：

- 可以通过局部主题覆盖全局主题，正如代码中通过Theme为第二行图标指定固定颜色（黑色）一样，这是一种常用的技巧，Flutter中会经常使用这种方法来自定义子树主题。那么为什么局部主题可以覆盖全局主题？这主要是因为Widget中使用主题样式时是通过`Theme.of(BuildContext context)`来获取的，我们看看其简化后的代码：

  ```
  static ThemeData of(BuildContext context, { bool shadowThemeOnly
  = false }) {
     // 简化代码，并非源码
     return context.inheritFromWidgetOfExactType(_InheritedTheme)
  }
  ```

  `context.inheritFromWidgetOfExactType`会在widget树中从当前位置向上查找第一个类型为`_InheritedTheme`的Widget。所以当局部使用Theme后，其子树中`Theme.of()`找到的第一个`_InheritedTheme`便是该Theme的。

- 本示例是对单个路由换肤，如果相对整个应用换肤，可以去修改MaterialApp的theme属性。