

简书



Flutter 完整开发实战详解

Flutter 完整开发实战详解(一、Dart 语言和 Flutter 基础)

前言

在如今的 Flutter 大潮下，本系列是让你看完会安心的文章。本系列将完整讲述：如何快速从 0 开发一个完整的 Flutter APP，配套高完成度 Flutter 开源项目 [GSYGithubAppFlutter](#)。同时也会提供一些 Flutter 的开发细节技巧，并针对开发过程中可能遇到的问题进行填坑。

系列文章分为三篇，第一部分是基础篇（针对 Dart 语言和 Flutter 基础），第二部分是 App 快速开发实战篇，第三部分是细节填坑篇。

笔者相继开发过 Flutter、React Native、Weex 等主流跨平台框架项目，其中 Flutter 的跨平台兼容性无疑最好。前期开发调试完全在 Android 端进行的情况下，第一次在 IOS 平台运行居然没有任何错误，并且还没出现 UI 兼容问题，相信对于经历过跨平台开发的猿们而言，这是多么的不可思议画面。并且 Flutter 的 HotLoad 相比较其他两个平台，也是丝滑的让人无法相信。吹爆了！

这些特点其实这得益于 Flutter Engine 和 Skia，如果有兴趣的可以看看笔者之前的[《移动端跨平台开发的深度解析》](#)。好了，感慨那么多，让我们进入正题吧。

一、基础篇

本篇主要涉及：环境搭建、Dart 语言、Flutter 的基础。

1、环境搭建

Flutter 的环境搭建十分省心，特别对应 Android 开发者而言，只是在 Android Studio

上安装插件，并下载 flutter Sdk 到本地，配置在环境变量即可。其实中文网的[搭建 Flutter 开发环境](#)已经很贴心详细，从平台指引开始安装基本都不会遇到问题。

这里主要是需要注意，因为某些不可抗力的原因，国内的用户需要配置 Flutter 的代理，并且国内用户在

搜索 Flutter 第三方包时，也是在 <https://pub.flutter-io.cn> 内查找，下方是需要配置到环境变量的地址。（ps Android Studio 下运行 IOS 也是蛮有意思的(๑_๑)）

```
///win 直接配置到环境编辑即可，mac 配置到 bash_profile

export PUB_HOSTED_URL=https://pub.flutter-io.cn //国内用户需要设置

export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn //国内用户需要设置
```

2、Dart 语言下的 Flutter

在跨平台开领域被 JS 一统天下的今天，Dart 语言的出现无疑是一股清流。作为后来者，Dart 语言有着不少 Java、kotlin 和 JS 的影子，所以对于 Android 原生开发者、前端开发者而言无疑是非常友好的。

官方也提供了包括 IOS 开发者，React Native 等开发者迁移到 Flutter 上的文档，所以请不要担心，Dart 语言不会是你掌握 Flutter 的门槛。甚至作为开发者，就算你不懂 Dart 也可以看着代码摸索。

Come on，下面主要通过对比，简单讲述下 Dart 的一些特性，主要涉及的是 Flutter 下使用。

基本类型

var 可以定义变量，如 `var tag = "666"`，这和 JS、Kotlin 等语言类似，同时 Dart 属于动态类型语言，支持闭包。

Dart 中 number 类型分为 `int` 和 `double`，其中 java 中的 long 对应的也是 Dart 中的 int 类型。Dart 中没有 float 类型。

Dart 下只有 bool 型可以用于 if 等判断，不同于 JS 这种使用方式是不合法的 `var g = "null"; if(g){}`。

DART 中，switch 支持 String 类型。

- 变量

- Dart 不需要给变量设置 `setter getter` 方法，这和 kotlin 等类似。Dart 中所有的基础类型、类等都继承 Object，默认值是 NULL，自带 getter 和 setter，而如果是 final 或者 const 的话，那么它只有一个 getter 方法。

Dart 中 `final` 和 `const` 表示常量，比如 `final name = 'GSY'; const value= 1000000;` 同时 `static const` 组合代表了静态常量。其中 `const` 的值在编译期确定，`final` 的值要到编译时才确定。（ps Flutter 在 Release 下是 AOT 模式。）

Dart 下的数值，在作为字符串使用时，是需要显式指定的。比如：`int i = 0; print("aaaa" + i);` 这样并不支持，需要 `print("aaaa" + i.toString());` 这样使用。这和 Java 与 JS 存在差异。所以在使用动态类型时，需要注意不要把 `number` 类型当做 `String` 使用。

DART 中数组等于列表，所以 `var list = [];` 和 `List list = new List();` 可以简单看做一样。

• 方法

Dart 下 `??`、`??=` 属于操作符，如：`AA ?? "999"` 表示如果 `AA` 为空，返回 `999`；`AA ??= "999"` 表示如果 `AA` 为空，给 `AA` 设置成 `999`。

Dart 方法可以设置 参数默认值 和 指定名称 。比如：`getDetail(String userName, reposName, {branch = "master"})` 方法，这里 `branch` 不设置的话，默认是 “master”。参数类型 可以指定或者不指定。调用效果：`getRepositoryDetailDao("aaa", "bbbb", branch: "dev");`

Dart 不像 Java ，没有关键词 `public`、`private` 等修饰符，`_` 下横向直接代表 `private` ，但是有 `@protected` 注解。

Dart 中多构造函数，可以通过如下代码实现的。默认构造方法只能有一个，而通过 `Model.empty()` 方法可以创建一个空参数的类，其实方法名称随你喜欢。而变量初始化值时，只需要通过 `this.name` 在构造方法中指定即可：

```
class ModelA {
  String name;
  String tag;

  //默认构造方法，赋值给 name 和 tag
  ModelA(this.name, this.tag);

  //返回一个空的 ModelA
  ModelA.empty();

  //返回一个设置了 name 的 ModelA
  ModelA.forName(this.name);
}
```

Flutter

Flutter 中支持 `async/await`。这一点和 ES7 很像，如下代码所示，只是定义的位置不同。同时异步操作也和 ES6 中的 `Promise` 很像，只是 Flutter 中返回的是 `Future` 对象，通过 `then` 可以执行下一步。如果返回的还是 `Future` 便可以 `then().then().then()` 的流式操作了。

```
///模拟等待两秒，返回 OK
request() async {
  await Future.delayed(Duration(seconds: 1));
  return "ok!";
}

///得到"ok!"后，将"ok!"修改为"ok from request"
doSomething() async {
  String data = await request();
  data = "ok from request";
  return data;
}

///打印结果
renderSome() {
  doSomething().then((value) {
    print(value);
    ///输出 ok from request
  });
}
```

Flutter 中 `setState` 很有 React Native 的既视感，Flutter 中也是通过 `state` 跨帧实现管理数据状态的，这个后面会详细讲到。

Flutter 中一切皆 `Widget` 呈现，通过 `build` 方法返回 `Widget`，这也是和 React Native 中，通过 `render` 函数返回需要渲染的 `component` 一样的模式。

3、Flutter Widget

在 Flutter 中，一切的显示都是 `Widget`。`Widget` 是一切的基础，作为响应式的渲染，属于 MVVM 的实现机制。我们可以通过修改数据，再用 `setState` 设置数据，Flutter 会自动通过绑定的数据更新 `Widget`。所以你需要做的就是实现 `Widget` 界面，并且和数据绑定起来。

`Widget` 分为有状态和无状态两种，在 Flutter 中每个页面都是一帧。无状态就是保持在那一帧。而有状态的 `Widget` 当数据更新时，其实是绘制了新的 `Widget`，只是 `State` 实现了跨帧的数据同步保存。

这里有个小 Tip，当代码框里输入 `stl` 的时候，可以自动弹出创建无状态控件的模板选项，而输入 `stf` 的时，就会弹出创建有状态 Widget 的模板选项。

代码格式化的时候，括号内外的逗号都会影响格式化时换行的位置。

如果觉得默认换行的线太短，可以在设置-Editor-Code Style-Dart-Wrapping and Braces-Hard wrap at 设置你接受的数值。

3.1、无状态 StatelessWidget

直接进入主题，下方代码是无状态 Widget 的简单实现。

继承 `StatelessWidget`，通过 `build` 方法返回一个布局好的控件。可能现在你还对 Flutter 的内置控件不熟悉，but **Don't worry, take is easy**，后面我们会详细介绍。这里你只需要知道，一个无状态的 Widget 就是这么简单。

Widget 和 Widget 之间通过 `child:` 进行嵌套。其中有的 Widget 只能有一个 child，比如下方的 `Container`；有的 Widget 可以多个 child，也就是 `children:`，比如 `Colum` 布局。下方代码便是 `Container` Widget 嵌套了 `Text` Widget。

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

class DEMOWidget extends StatelessWidget {
  final String text;

  //数据可以通过构造方法传递进来
  DEMOWidget(this.text);

  @override
  Widget build(BuildContext context) {
    //这里返回你需要的控件
    //这里末尾有没有的逗号，对于格式化代码而已是不一样的。

    return Container(
      //白色背景
      color: Colors.white,

      //Dart 语法中，?? 表示如果 text 为空，就返回尾号后的内容。
      child: Text(text ?? "这就是无状态 DMEO"),
    );
  }
}
```

```
}
```

3.2、有状态 StatefulWidget

继续直插主题，如下代码，是有状态的 widget 的简单实现。

你需要创建管理的是主要是 `State`，通过 `State` 的 `build` 方法去构建控件。在 `State` 中，你可以动态改变数据，这类似 MVVM 实现，在 `setState` 之后，改变的数据会触发 `Widget` 重新构建刷新。而下方代码中，是通过延两秒之后，让文本显示为 "这就变了数值"。

如下代码还可以看出，`State` 中主要的声明周期有：

- **initState**：初始化，理论上只有初始化一次，第二篇中会说特殊情况下。
- **didChangeDependencies**：在 `initState` 之后调用，此时可以获取其他 `State`。
- **dispose**：销毁，只会调用一次。

看到没，Flutter 其实就是这么简单！你的关注点只要在：创建你的 `StatelessWidget` 或者 `StatefulWidget` 而已。你需要的就是在 `build` 中堆积你的布局，然后把数据添加到 `Widget` 中，最后通过 `setState` 改变数据，从而实现画面变化。

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

class DemoStateWidget extends StatefulWidget {

  final String text;

  ///通过构造方法传值
  DemoStateWidget(this.text);

  ///主要是负责创建 state
  @override
  _DemoStateWidgetState createState() => _DemoStateWidgetState(text);
}

class _DemoStateWidgetState extends State<DemoStateWidget> {

  String text;

  _DemoStateWidgetState(this.text);
```

```
@override
void initState() {
  ///初始化，这个函数在生命周期中只调用一次
  super.initState();
  ///定时 2 秒
  new Future.delayed(const Duration(seconds: 1), () {
    setState(() {
      text = "这就变了数值";
    });
  });
}

@override
void dispose() {
  ///销毁
  super.dispose();
}

@override
void didChangeDependencies() {
  ///在 initState 之后调 Called when a dependency of this [State] object changes.
  super.didChangeDependencies();
}

@override
Widget build(BuildContext context) {
  return Container(
    child: Text(text ?? "这就是有状态 DME0"),
  );
}
}
```

4、Flutter 布局

Flutter 中拥有需要将近 30 种内置的 **布局 Widget**，其中常用有 Container、Padding、Center、Flex、Stack、Row、Column、ListView 等，下面简单讲解它们的特性和使用。

| 类型 | 作用特点 |
|-----------|---|
| Container | 只有一个子 Widget。默认充满，包含了 padding、margin、color、宽高、decoration 等配置。 |
| Padding | 只有一个子 Widget。只用于设置 Padding，常用于嵌套 child，给 child 设置 padding。 |
| Center | 只有一个子 Widget。只用于居中显示，常用于嵌套 child，给 child 设置居中。 |
| Stack | 可以有多个子 Widget。子 Widget 堆叠在一起。 |
| Column | 可以有多个子 Widget。垂直布局。 |
| Row | 可以有多个子 Widget。水平布局。 |
| Expanded | 只有一个子 Widget。在 Column 和 Row 中充满。 |
| ListView | 可以有多个子 Widget。自己意会吧。 |

- **Container**：最常用的默认布局！只能包含一个 `child`，支持配置 padding、margin、color、宽高、decoration（一般配置边框和阴影）等配置，在 Flutter 中，不是所有的控件都有 宽高、padding、margin、color 等属性，所以才会有 Padding、Center 等 Widget 的存在。

```
new Container(  
  ///四周 10 大小的 margin  
  margin: EdgeInsets.all(10.0),  
  height: 120.0,  
  width: 500.0,  
  ///透明黑色遮罩  
  decoration: new BoxDecoration(  
    ///弧度为 4.0  
    borderRadius: BorderRadius.all(Radius.circular(4.0)),  
    ///设置了 decoration 的 color，就不能设置 Container 的 color。  
    color: Colors.black,  
    ///边框  
    border: new Border.all(color: Color(GSYColors.subTextColor), width: 0.3)),  
  child: new Text("666666"));
```


- **Column、Row** 绝对是必备布局，横竖布局也是日常中最常见的场景。如下方所示，它们常用的有这些属性配置：主轴方向是 **start** 或 **center** 等；副轴方向方向是 **start** 或 **center** 等；**mainAxisSize** 是充满最大尺寸，或者只根据子 **Widget** 显示最小尺寸。

```
//主轴方向，Column 的竖向、Row 我的横向
mainAxisAlignment: MainAxisAlignment.start, //默认是最大充满、还是根据 child 显示最小大小
mainAxisSize: MainAxisSize.max, //副轴方向，Column 的横向、Row 我的竖向
crossAxisAlignment :CrossAxisAlignment.center,
```

- **Expanded** 在 **Column** 和 **Row** 中代表着平均充满，当有两个存在的时候默认均分充满。同时页可以设置 **flex** 属性决定比例。

```
new Column(
  ///主轴居中,即是竖直向居中
  mainAxisAlignment: MainAxisAlignment.center,
  ///大小按照最小显示
  mainAxisSize : MainAxisSize.min,
  ///横向也居中
  crossAxisAlignment : CrossAxisAlignment.center,
  children: <Widget>[
    ///flex 默认为 1
    new Expanded(child: new Text("1111"), flex: 2,),
    new Expanded(child: new Text("2222")),
  ],
);
```

接下来我们来写一个复杂一些的控件。首先我们创建一个私有方法 **_getBottomItem**，返回一个 **Expanded Widget**，因为后面我们需要将这个方法返回的 **Widget** 在 **Row** 下平均充满。

如代码中注释，布局内主要是现实一个居中的 **Icon** 图标和文本，中间间隔 **5.0** 的 **padding**：

```
///返回一个居中带图标和文本的 Item
_getBottomItem(IconData icon, String text) {
  ///充满 Row 横向的布局
  return new Expanded(
    flex: 1,
    ///居中显示
    child: new Center(
```

```
///横向布局

child: new Row(

  ///主轴居中,即是横向居中

  mainAxisAlignment: MainAxisAlignment.center,

  ///大小按照最大充满

  mainAxisAlignment : MainAxisAlignment.max,

  ///竖向也居中

  crossAxisAlignment : CrossAxisAlignment.center,

  children: <Widget>[

    ///一个图标,大小 16.0, 灰色

    new Icon(

      icon,

      size: 16.0,

      color: Colors.grey,

    ),

    ///间隔

    new Padding(padding: new EdgeInsets.only(left:5.0)),

    ///显示文本

    new Text(

      text,

      //设置字体样式: 颜色灰色, 字体大小 14.0

      style: new TextStyle(color: Colors.grey, fontSize: 14.0),

      //超过的省略为...显示

      overflow: TextOverflow.ellipsis,

      //最长一行

      maxLines: 1,

    ),

  ],

),

);

}
```

 1000

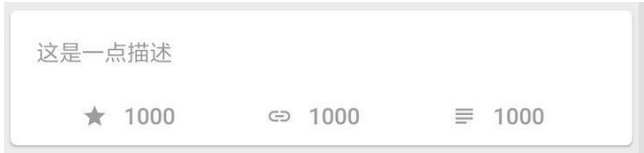
item 效果

接着我们把上方的方法，放到新的布局里。如下流程和代码：

- 首先是 `Container` 包含了 `Card`，用于快速简单的实现圆角和阴影。
- 然后接下来包含了 `FlatButton` 实现了点击，通过 `Padding` 实现了边距。
- 接着通过 `Column` 垂直包含了两个子 `Widget`，一个是 `Container`、一个是 `Row`。
- `Row` 内使用的就是 `_getBottomItem` 方法返回的 `Widget`，效果如下图。

```
@override
Widget build(BuildContext context) {
  return new Container(
    ///卡片包装
    child: new Card(
      ///增加点击效果
      child: new FlatButton(
        onPressed: (){print("点击了哦");},
        child: new Padding(
          padding: new EdgeInsets.only(left: 0.0, top: 10.0, right: 10.0, bottom: 10.0),
          child: new Column(
            mainAxisAlignment: MainAxisAlignment.min,
            children: <Widget>[
              ///文本描述
              new Container(
                child: new Text(
                  "这是一点描述",
                  style: TextStyle(
                    color: Color(GSYColors.subTextColor),
                    fontSize: 14.0,
                  ),
                  ///最长三行，超过 ... 显示
                  maxLines: 3,
                  overflow: TextOverflow.ellipsis,
                ),
                margin: new EdgeInsets.only(top: 6.0, bottom: 2.0),
                alignment: Alignment.topLeft),
              new Padding(padding: EdgeInsets.all(10.0)),
            ],
          ///三个平均分配的横向图标文字
        ),
      ),
    ),
  );
}
```

```
new Row(  
  crossAxisAlignment: CrossAxisAlignment.start,  
  children: <Widget>[  
    _getBottomItem(Icons.star, "1000"),  
    _getBottomItem(Icons.link, "1000"),  
    _getBottomItem(Icons.subject, "1000"),  
  ],  
)  
],  
)  
))),  
);  
}
```



完整 Item

Flutter 中，你的布局很多时候就是这么一层一层嵌套出来的，当然还有其他更高级的布局方式，这里就先不展开了。

5、Flutter 页面

Flutter 中除了布局的 Widget，还有交互显示的 Widget 和完整页面呈现的 Widget。其中常见的有 MaterialApp、Scaffold、AppBar、Text、Image、FlatButton 等。下面简单介绍这些 Widget，并完成一个页面。

| 类型 | 作用特点 |
|-------------|---|
| MaterialApp | 一般作为 APP 顶层的主页入口，可配置主题，多语言，路由等 |
| Scaffold | 一般用户页面的承载 Widget，包含 appbar、snackbar、drawer 等 material design 的设定。 |
| AppBar | 一般用于 Scaffold 的 appbar ，内有标题，二级页面返回按键等，当然不止这些，tabbar 等也会需要它 。 |

| 类型 | 作用特点 |
|------------|---|
| Text | 显示文本，几乎都会用到，主要是通过 style 设置 TextStyle 来设置字体样式等。 |
| RichText | 富文本，通过设置 <code>TextSpan</code> ，可以拼接出富文本场景。 |
| TextField | 文本输入框： <code>new TextField(controller: //文本控制器, obscureText: "hint 文本");</code> |
| Image | 图片加载： <code>new FadeInImage.assetNetwork(placeholder: "预览图", fit: BoxFit.fitWidth, image: "url");</code> |
| FlatButton | 按键点击： <code>new FlatButton(onPressed: () {},child: new Container());</code> |

那么再次直插主题实现一个简单完整的页面试试。如下方代码：

- 首先我们创建一个 StatefulWidget: `DemoPage`。
- 然后在 `_DemoPageState` 中，通过 `build` 创建了一个 `Scaffold`。
- `Scaffold` 内包含了一个 `AppBar` 和一个 `ListView`。
- `AppBar` 类似标题了区域，其中设置了 `title` 为 Text Widget。
- `body` 是 `ListView`, 返回了 20 个之前我们创建过的 `DemoItem` Widget。

```
import 'package:flutter/material.dart';
import 'package:gsy_github_app_flutter/test/DemoItem.dart';

class DemoPage extends StatefulWidget {
  @override
  _DemoPageState createState() => _DemoPageState();
}

class _DemoPageState extends State<DemoPage> {
  @override
  Widget build(BuildContext context) {
    ///一个页面的开始

    ///如果是新页面，会自带返回按钮

    return new Scaffold(
      ///背景样式
      backgroundColor: Colors.blue,
      ///标题栏，当然不仅仅是标题栏
      appBar: new AppBar(
        ///这个 title 是一个 Widget
        title: new Text("Title"),
      ),
    ),
```

```
///正式的页面开始

///一个 ListView, 20 个 Item

body: new ListView.builder(
  itemBuilder: (context, index) {
    return new DemoItem();
  },
  itemCount: 20,
),
);
}
```

最后我们创建一个 `StatelessWidget` 作为入口文件，实现一个 `MaterialApp` 将上方的 `DemoPage` 设置为 home 页面，通过 `main` 入口执行页面。

```
import 'package:flutter/material.dart';
import 'package:gsy_github_app_flutter/test/DemoPage.dart';

void main() {
  runApp(new DemoApp());
}

class DemoApp extends StatelessWidget {
  DemoApp({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(home: DemoPage());
  }
}
```



最终显示

好吧，第一部分终于完了，这里主要讲解都是一些简单基础的东西，适合安利入坑，后续还有两篇主要实战，敬请期待哟！(^_^)ゞ

Flutter 完整开发实战详解(二、 快速开发实战篇)

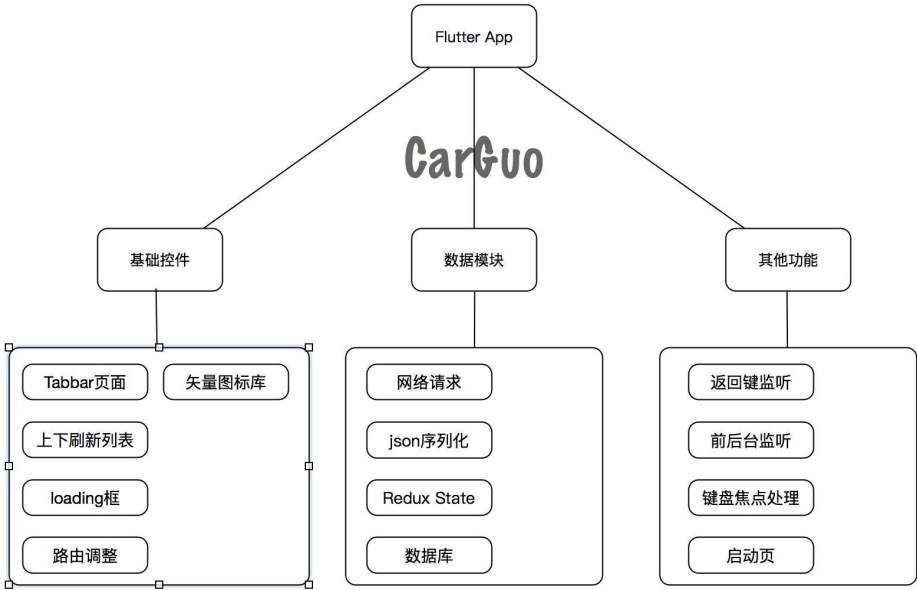
作为系列文章的第二篇，继《Flutter 完整开发实战详解(一、 Dart 语言和 Flutter 基础)》之后，本篇将为你着重展示：如何搭建一个通用的 Flutter App 常用功能脚手架，快速开发一个完整的 Flutter 应用。

友情提示：本文所有代码均在 [GSYGithubAppFlutter](#) ，文中示例代码均可在其中找到，看完本篇相信你应该可以轻松完成如下效果。相关基础还请看[篇章一](#)。

我们的目标是！(^_^) ʘ

前言

本篇内容结构如下图，主要分为：**基础控件**、**数据模块**、**其他功能** 三部分。每大块中的小模块，除了涉及的功能实现外，对于实现过程中笔者遇到的问题，会一并展开阐述。本系列的最终目的是：**让你感受Flutter 的喜悦！**那么就让我们愉悦的往下开始吧！(●_●)



我是简陋的下图

一、基础控件

所谓的基础，大概就是砍柴功了吧！

1、Tabbar 控件实现

Tabbar 页面是常有需求，而在 Flutter 中：**Scaffold + AppBar + Tabbar + TabbarView** 是 Tabbar 页面的最简单实现，但在加上 **AutomaticKeepAliveClientMixin** 用于页面 keepAlive 之后，诸如[#11895](#) 的问题便开始成为 Crash 的元凶。直到 flutter v0.5.7 sdk 版本修复后，问题依旧没有完全解决，所以无奈最终修改了实现方案。

目前笔者是通过 **Scaffold + Appbar + Tabbar + PageView** 来组合实现效果，从而解决上述问题。因为该问题较为常见，所以目前已经单独实现了测试 Demo，有兴趣的可以看看 [TabBarWithPageView](#)。

下面我们直接代码走起，首先作为一个 Tabbar Widget，它肯定是一个 `StatefulWidget`，所以我们先实现它的 `State`：

```
class _GSYTabBarState extends State<GSYTabBarWidget> with SingleTickerProviderStateMixin {  
  ///...省略非关键代码  
  
  @override  
  void initState() {  
    super.initState();  
  
    ///初始化时创建控制器  
  
    ///通过 with SingleTickerProviderStateMixin 实现动画效果。  
    _tabController = new TabController(vsync: this, length: _tabItems.length);  
  }  
  
  @override  
  void dispose() {  
    ///页面销毁时，销毁控制器  
    _tabController.dispose();  
    super.dispose();  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    ///底部 TabBar 模式  
    return new Scaffold(  
      ///设置侧边滑出 drawer，不需要可以不设置  
      drawer: _drawer,  
      ///设置悬浮按钮，不需要可以不设置  
      floatingActionButton: _floatingActionButton,  
      ///标题栏  
      appBar: new AppBar(  
        backgroundColor: _backgroundColor,  
        title: _title,  
      ),  
      ///页面主体，PageView，用于承载 Tab 对应的页面  
      body: new PageView(  
        ///必须有的控制器，与 tabBar 的控制器同步  
        controller: _pageController,  

```

```

    ///每一个 tab 对应的页面主体, 是一个 List<Widget>
    children: _tabViews,
    onPageChanged: (index) {
      ///页面触摸作用滑动回调, 用于同步 tab 选中状态
      _tabController.animateTo(index);
    },
  ),
  ///底部导航栏, 也就是 tab 栏
  bottomNavigationBar: new Material(
    color: _backgroundColor,
    ///tabBar 控件
    child: new TabBar(
      ///必须有的控制器, 与 pageView 的控制器同步
      controller: _tabController,
      ///每一个 tab item, 是一个 List<Widget>
      tabs: _tabItems,
      ///tab 底部选中条颜色
      indicatorColor: _indicatorColor,
    ),
  ));
}
}

```

如上代码所示, 这是一个底部 TabBar 的页面的效果。TabBar 和 PageView 之间通过 `_pageController` 和 `_tabController` 实现 Tab 和页面的同步, 通过 `SingleTickerProviderStateMixin` 实现 Tab 的动画切换效果 (ps 如果有需要多个嵌套动画效果, 你可能需要 `TickerProviderStateMixin`)。从代码中我们可以看到:

手动左右滑动 `PageView` 时, 通过 `onPageChanged` 回调调用 `_tabController.animateTo(index);` 同步 TabBar 状态。

`_tabItems` 中, 监听每个 `TabBarItem` 的点击, 通过 `_pageController` 实现 PageView 的状态同步。

而上面代码还缺少了 `TabBarItem` 的点击, 因为这块被放到了外部实现。当然你也可以直接在内部封装好控件, 直接传递配置数据显示, 这个可以根据个人需要封装。

外部调用代码如下：每个 Tabbar 点击时，通过 `pageController.jumpTo` 跳转页面，每个页面需要跳转坐标为：当前屏幕大小乘以索引 `index`。

```
class _TabBarBottomPageWidgetState extends State<TabBarBottomPageWidget> {

  final PageController pageController = new PageController();

  final List<String> tab = ["动态", "趋势", "我的"];

  ///渲染底部 Tab
  _renderTab() {
    List<Widget> list = new List();

    for (int i = 0; i < tab.length; i++) {
      list.add(new FlatButton(onPressed: () {
        ///每个 Tabbar 点击时，通过 jumpTo 跳转页面
        ///每个页面需要跳转坐标为：当前屏幕大小 * 索引 index。
        topPageControl.jumpTo(MediaQuery
          .of(context)
          .size
          .width * i);
      }, child: new Text(
        tab[i],
        maxLines: 1,
      )));
    }
    return list;
  }

  ///渲染 Tab 对应页面
  _renderPage() {
    return [
      new TabBarPageFirst(),
      new TabBarPageSecond(),
      new TabBarPageThree(),
    ];
  }
}
```

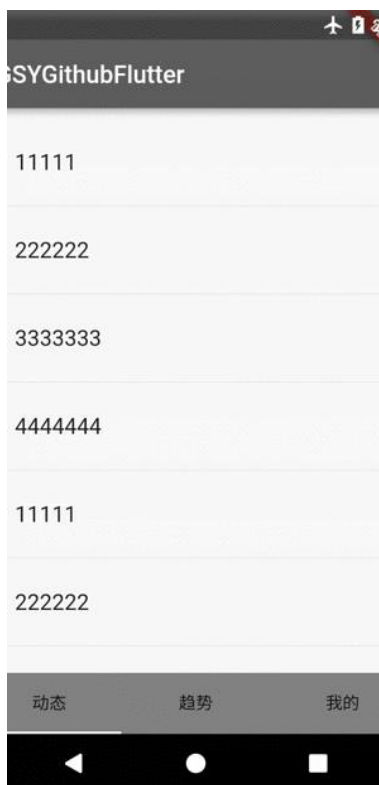
```

@override
Widget build(BuildContext context) {
  ///带 Scaffold 的 Tabbar 页面
  return new GSYTabBarWidget(
    type: GSYTabBarWidget.BOTTOM_TAB,
    ///渲染 tab
    tabItems: _renderTab(),
    ///渲染页面
    tabViews: _renderPage(),
    topPageControl: pageController,
    backgroundColor: Colors.black45,
    indicatorColor: Colors.white,
    title: new Text("GSYGithubFlutter"));
}
}

```

如果到此结束，你会发现页面点击切换时，`StatefulWidget` 的子页面每次都会重新调用 `initState`。这肯定不是我们想要的，所以这时你就需要 `AutomaticKeepAliveClientMixin`。

每个 Tab 对应的 `StatefulWidget` 的 `State`，需要通过 `with AutomaticKeepAliveClientMixin`，然后重写 `@override bool get wantKeepAlive => true;`，就可以实不重新构建的效果了，效果如下图。



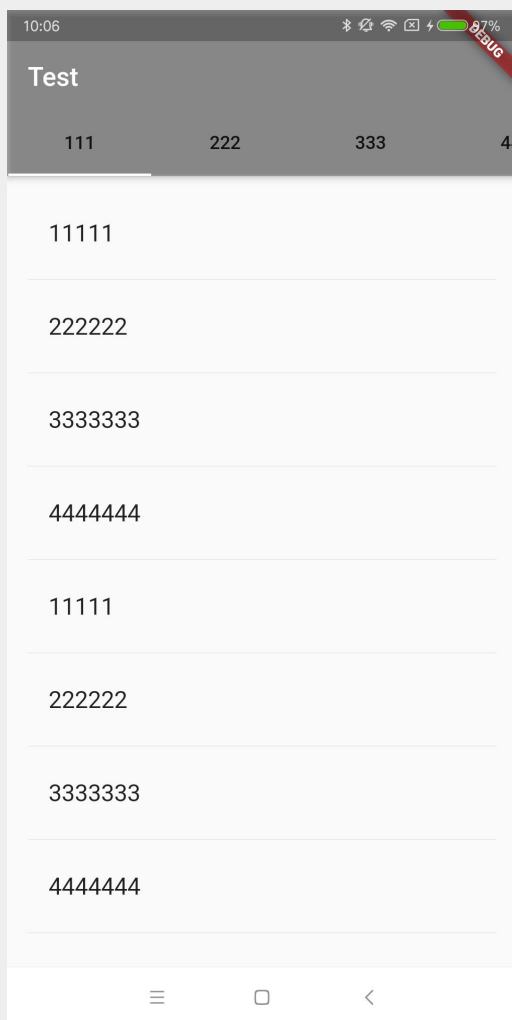
页面效果

既然底部 Tab 页面都实现了，干脆顶部 tab 页面也一起完成。如下代码，和底部 Tab 页的区别在于：

- 底部 tab 是放在了 Scaffold 的 bottomNavigationBar 中。
 - 顶部 tab 是放在 AppBar 的 bottom 中，也就是标题栏之下。
- 同时我们在顶部 TabBar 增加 isScrollable: true 属性，实现常见的顶部 Tab 的效果，如下方图片所示。

```
return new Scaffold(  
  ///设置侧边滑出 drawer，不需要可以不设置  
  drawer: _drawer,  
  ///设置悬浮按钮，不需要可以不设置  
  floatingActionButton: _floatingActionButton,  
  ///标题栏  
  appBar: new AppBar(  
    backgroundColor: _backgroundColor,  
    title: _title,  
    ///tabBar 控件  
    bottom: new TabBar(  
      ///顶部时，tabBar 为可以滑动的模式  
      isScrollable: true,  
      ///必须有的控制器，与 pageView 的控制器同步  
      controller: _tabController,  
      ///每一个 tab item，是一个 List<Widget>  
      tabs: _tabItems,  
      ///tab 底部选中条颜色  
      indicatorColor: _indicatorColor,  
    ),  
  ),  
  ///页面主体，PageView，用于承载 Tab 对应的页面  
  body: new PageView(  
    ///必须有的控制器，与 tabBar 的控制器同步  
    controller: _pageController,  
    ///每一个 tab 对应的页面主体，是一个 List<Widget>  
    children: _tabViews,  
    ///页面触摸作用滑动回调，用于同步 tab 选中状态  
    onPageChanged: (index) {
```

```
        _tabController.animateTo(index);  
      },  
    ),  
  );
```



顶部 TabBar 效果

在 TabBar 页面中，一般还会出现：父页面需要控制 PageView 中子页的需求。这时候就需要用到 GlobalKey 了。比如 `GlobalKey<PageOneState> stateOne = new GlobalKey<PageOneState>();`，通过 `globalKey.currentState` 对象，你就可以调用到 PageOneState 中的公开方法。这里需要注意 GlobalKey 需要全局唯一，一般可以在 build 方法中创建。

2、上下刷新列表

毫无争议，必备控件。Flutter 中 为我们提供了 RefreshIndicator 作为内置下拉刷新控件；同时我们通过给 ListView 添加 ScrollController 做滑动监听，在最后增加一个 Item，作为上滑加载更多的 Loading 显示。

如下代码所示，通过 `RefreshIndicator` 控件可以简单完成下拉刷新工作。这里需要注意一点是：可以利用 `GlobalKey<RefreshIndicatorState>` 对外提供 `RefreshIndicator` 的 `RefreshIndicatorState`，这样外部就可以通过 `GlobalKey` 调用 `globalKey.currentState.show()`，主动显示刷新状态并触发 `onRefresh`。

上拉加载更多在代码中是通过 `_getListCount()` 方法，在原本的数据基础上，增加实际需要渲染的 item 数量给 `ListView` 实现的，最后通过 `ScrollController` 监听到底部，触发 `onLoadMore`。

如下代码所示，通过 `_getListCount()` 方法，还可以配置空页面，头部等常用效果。其实就是在内部通过改变实际 item 数量与渲染 Item，以实现更多配置效果。

```
class _GSPullLoadWidgetState extends State<GSPullLoadWidget> {
  ///...

  final ScrollController _scrollController = new ScrollController();

  @override
  void initState() {
    ///增加滑动监听
    _scrollController.addListener(() {
      ///判断当前滑动位置是不是到达底部，触发加载更多回调
      if (_scrollController.position.pixels == _scrollController.position.maxScrollExtent) {
        if (this.onLoadMore != null && this.control.needLoadMore) {
          this.onLoadMore();
        }
      }
    });
    super.initState();
  }

  ///根据配置状态返回实际列表数量
  ///实际上这里可以根据你的需要做更多的处理
  ///比如多个头部，是否需要空页面，是否需要显示加载更多。
  _getListCount() {
    ///是否需要头部
    if (control.needHeader) {
      ///如果需要头部，用 Item 0 的 Widget 作为 ListView 的头部
      ///列表数量大于 0 时，因为头部和底部加载更多选项，需要对列表数据总数+2
      return (control.dataList.length > 0) ? control.dataList.length + 2 : control.dataList.length + 1;
    } else {
```

```

    ///如果不需要头部, 在没有数据时, 固定返回数量 1 用于空页面呈现

    if (control.dataList.length == 0) {

        return 1;

    }

    ///如果有数据, 因为部加载更多选项, 需要对列表数据总数+1

    return (control.dataList.length > 0) ? control.dataList.length + 1 : control.dataList.length;

}

}

///根据配置状态返回实际列表渲染 Item

_getItem(int index) {

    if (!control.needHeader && index == control.dataList.length && control.dataList.length != 0) {

        ///如果不需要头部, 并且数据不为 0, 当 index 等于数据长度时, 渲染加载更多 Item (因为 index 是从 0 开始)

        return _buildProgressIndicator();

    } else if (control.needHeader && index == _getListCount() - 1 && control.dataList.length != 0) {

        ///如果需要头部, 并且数据不为 0, 当 index 等于实际渲染长度 - 1 时, 渲染加载更多 Item (因为 index 是从 0 开始)

        return _buildProgressIndicator();

    } else if (!control.needHeader && control.dataList.length == 0) {

        ///如果不需要头部, 并且数据为 0, 渲染空页面

        return _buildEmpty();

    } else {

        ///回调外部正常渲染 Item, 如果这里有需要, 可以直接返回相对位置的 index

        return itemBuilder(context, index);

    }

}

}

@override

Widget build(BuildContext context) {

    return new RefreshIndicator(

        ///GlobalKey, 用户外部获取 RefreshIndicator 的 State, 做显示刷新

        key: refreshKey,

        ///下拉刷新触发, 返回的是一个 Future

        onRefresh: onRefresh,

        child: new ListView.builder(

            ///保持 ListView 任何情况都能滚动, 解决在 RefreshIndicator 的兼容问题。

            physics: const AlwaysScrollableScrollPhysics(),

```



```
    ///根据状态返回子孔健
    itemBuilder: (context, index) {
      return _getItem(index);
    },
    ///根据状态返回数量
    itemCount: _getListCount(),
    ///滑动监听
    controller: _scrollController,
  ),
);
}

///空页面
Widget _buildEmpty() {
  ///...
}

///上拉加载更多
Widget _buildProgressIndicator() {
  ///...
}
}
```

效果如图

3、Loading 框

在上一小节中，我们实现上滑加载更多的效果，其中就需要展示 Loading 状态的需求。默认系统提供了 `CircularProgressIndicator` 等，但是有追求的我们怎么可能局限于此，这里推荐一个第三方 Loading 库：`flutter_spinkit`，通过简单的配置就可以使用丰富的 Loading 样式。

继续上一小节中的 `_buildProgressIndicator` 方法实现，通过 `flutter_spinkit` 可以快速实现更不一样的 Loading 样式。

```
///上拉加载更多
Widget _buildProgressIndicator() {
  ///是否需要显示上拉加载更多的 loading
```

```
Widget bottomWidget = (control.needLoadMore)

? new Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[

  ///loading 框

  new SpinKitRotatingCircle(color: Color(0xFF24292E)),

  new Container(

    width: 5.0,

  ),

  ///加载中文本

  new Text(

    "加载中...",

    style: TextStyle(

      color: Color(0xFF121917),

      fontSize: 14.0,

      fontWeight: FontWeight.bold,

    ),

  )

])

/// 不需要加载

: new Container();

return new Padding(

  padding: const EdgeInsets.all(20.0),

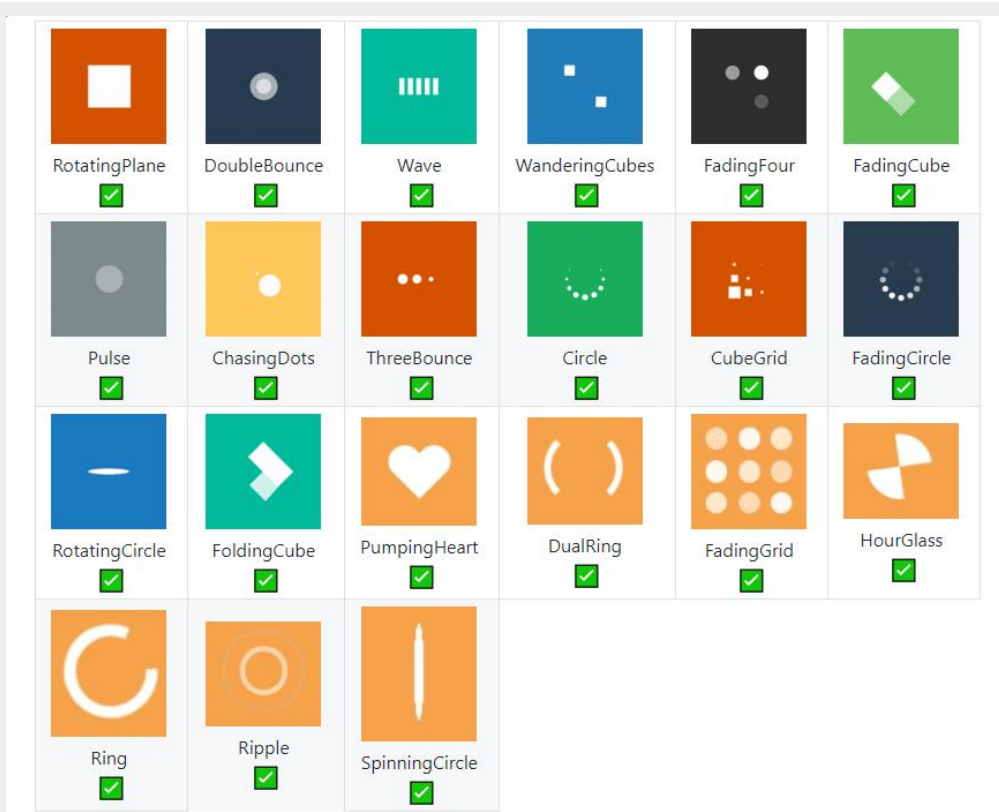
  child: new Center(

    child: bottomWidget,

  ),

);

}
```



loading 样式

4、矢量图标库

矢量图标对笔者是必不可少的。比起一般的 png 图片文件，矢量图标在开发过程中：可以轻松定义颜色，并且任意调整大小不模糊。矢量图标库是引入 ttf 字体库文件实现，在 Flutter 中通过 `Icon` 控件，加载对应的 `IconData` 显示即可。

Flutter 中默认内置的 `Icons` 类就提供了丰富的图标，直接通过 `Icons` 对象即可使用，同时个人推荐阿里巴巴的 `iconfont`。如下代码，通过在 `pubspec.yaml` 中添加字体库支持，然后在代码中创建 `IconData` 指向字体库名称引用即可。

```
fonts:
  - family: wxcIconFont
    fonts:
      - asset: static/font/iconfont.ttf

.....

///使用 Icons
new Tab(
  child: new Column(
```

```

        mainAxisAlignment: MainAxisAlignment.center,

        children: <Widget>[new Icon(Icons.list, size: 16.0), new Text("趋势")],

    ),

),

///使用 iconfont

new Tab(

    child: new Column(

        mainAxisAlignment: MainAxisAlignment.center,

        children: <Widget>[new Icon(IconData(0xe6d0, fontFamily: "wxcIconFont"), size: 16.0), new Text("我

的")],

    ),

)

```

5、路由跳转

Flutter 中的页面跳转是通过 `Navigator` 实现的，路由跳转又分为：**带参数跳转**和**不带参数跳转**。不带参数跳转比较简单，默认可以通过 `MaterialApp` 的路由表跳转；而带参数的跳转，参数通过跳转页面的构造方法传递。常用的跳转有如下几种使用：

```

///不带参数的路由表跳转

Navigator.pushNamed(context, routeName);

///跳转新页面并且替换，比如登录页跳转主页

Navigator.pushReplacementNamed(context, routeName);

///跳转到新的路由，并且关闭给定路由的之前的所有页面

Navigator.pushNamedAndRemoveUntil(context, '/calendar', ModalRoute.withName('/'));

///带参数的路由跳转，并且监听返回

Navigator.push(context, new MaterialPageRoute(builder: (context) => new NotifyPage())).then((res) {

    ///获取返回处理

});

```

同时我们可以看到，`Navigator` 的 `push` 返回的是一个 `Future`，这个 `Future` 的作用是在页面返回时被调用的。也就是你可以通过 `Navigator` 的 `pop` 时返回参数，之后在 `Future` 中可以的监听中处理页面的返回结果。

```

@optionalTypeArgs

static Future<T> push<T extends Object>(BuildContext context, Route<T> route) {

    return Navigator.of(context).push(route);
}

```

```
}
```

二、数据模块

数据为王，不过应该不是隔壁老王吧。

1、网络请求

当前 Flutter 网络请求封装中，国内最受欢迎的就是 [Dio](#) 了，Dio 封装了网络请求中的**数据转换、拦截器、请求返回**等。如下代码所示，通过对 Dio 的简单封装即可快速网络请求，真的很简单，更多的可以查 Dio 的官方文档，这里就不展开了。（真的不是懒）

```
///创建网络请求对象
Dio dio = new Dio();
Response response;
try {
  ///发起请求
  ///url 地址，请求数据，一般为 Map 或者 FormData
  ///options 额外配置，可以配置超时，头部，请求类型，数据响应类型，host 等
  response = await dio.request(url, data: params, options: option);
} on DioError catch (e) {
  ///http 错误是通过 DioError 的 catch 返回的一个对象
}
```

2、Json 序列化

在 Flutter 中，json 序列化是有些特殊的。不同与 JS，比如使用上述 Dio 网络请求返回，如果配置了返回数据格式为 **json**，实际上的到会是一个 **Map**。而 Map 的 **key-value** 使用，在开发过程中并不是很方便，所以你需要对 **Map** 再进行一次转化，转为实际的 **Model** 实体。

所以 `json_serializable` 插件诞生了，[中文网 Json](#) 对其已有一段教程，这里主要补充说明下具体的使用逻辑。

```
dependencies:
  # Your other regular dependencies here
  json_annotation: ^0.2.2
dev_dependencies:
```

```
# Your other dev_dependencies here

build_runner: ^0.7.6

json_serializable: ^0.3.2
```

如下发代码所示：

创建你的实体 **Model** 之后，继承 **Object** 、然后通过 `@JsonSerializable()` 标记类名。

通过 `with _$TemplateSerializerMixin`，将 `fromJson` 方法委托到 `Template.g.dart` 的实现中。其中 `*.g.dart`、`_$*SerializerMixin`、`_$*FromJson` 这三个的引入，和 **Model** 所在的 **dart** 的文件名与 **Model** 类名有关，具体可见代码注释和后面图片。

最后通过 `flutter packages pub run build_runner build` 编译自动生成转化对象。（个人习惯完成后手动编译）

```
import 'package:json_annotation/json_annotation.dart';

///关联文件、允许 Template 访问 Template.g.dart 中的私有方法
///Template.g.dart 是通过命令生成的文件。名称为 xx.g.dart，其中 xx 为当前 dart 文件名称
///Template.g.dart 中创建了抽象类_$TemplateSerializerMixin，实现了_$TemplateFromJson 方法 part 'Template.g.dart';
///标志 class 需要实现 json 序列化功能@JsonSerializable()
///'xx.g.dart'文件中，默认会根据当前类名如 AA 生成 _$AASerializerMixin
///所以当前类名为 Template，生成的抽象类为 _$TemplateSerializerMixin
class Template extends Object with _$TemplateSerializerMixin {

  String name;

  int id;

  ///通过 JsonKey 重新定义参数名
  @JsonKey(name: "push_id")
  int pushId;

  Template(this.name, this.id, this.pushId);

  ///'xx.g.dart'文件中，默认会根据当前类名如 AA 生成 _$AAeFromJson 方法
```

```

///所以当前类名为 Template，生成的抽象类为 _$TemplateFromJson

factory Template.fromJson(Map<String, dynamic> json) => _$TemplateFromJson(json);

}

```

```

var prefix = '\_${className}';

var buffer = new StringBuffer();

final classAnnotation = _valueForAnnotation(annotation);

if (classAnnotation.createFactory) {
  var toSkip = _writeFactory(
    buffer, classElement, fields, prefix, classAnnotation.nullable);

  // If there are fields that are final - that are not set via the generated
  // constructor, then don't output them when generating the `toJson` call.
  for (var field in toSkip) {
    fields.remove(field.name);
  }
}

// Now we check for duplicate JSON keys due to colliding annotations.
// We do this now, since we have a final field list after any pruning done
// by `createFactory`.

fields.values.fold(new Set<String>(), (Set<String> set, fe) {
  var jsonKey = _jsonKeyFor(fe).name ?? fe.name;
  if (!set.add(jsonKey)) {
    throw new InvalidGenerationSourceError(
      'More than one field has the JSON key `\$jsonKey`.',
      todo: 'Check the `JsonKey` annotations on fields.');
  }
  return set;
});

if (classAnnotation.createToJson) {
  var mixClassName = '\_${prefix}SerializerMixin';
  var helpClassName = '\_${prefix}JsonMapWrapper';
}

```

序列化源码部分

上述操作生成后的 `Template.g.dart` 下的代码如下，这样我们就可以通过 `Template.fromJson` 和 `toJson` 方法对实体与 `map` 进行转化，再结合 `json.decode` 和 `json.encode`，你就可以愉悦的在 `string`、`map`、实体间相互转化了。

注意：新版 json 序列化中做了部分修改，代码更简单了，详见 demo

```

part of 'Template.dart';

Template _$TemplateFromJson(Map<String, dynamic> json) => new Template(
  json['name'] as String, json['id'] as int, json['push_id'] as int);

abstract class _$TemplateSerializerMixin {
  String get name;

  int get id;

  int get pushId;

  Map<String, dynamic> toJson() =>

```

```
<String, dynamic>{'name': name, 'id': id, 'push_id': pushId};
}
```

3、Redux State

相信在前端领域、Redux 并不是一个陌生的概念。作为全局状态管理机，用于 Flutter 中再合适不过。如果你没听说过，**Don't worry**，简单来说就是：它可以跨控件管理、同步 State。所以 `flutter_redux` 等着你征服它。

大家都知道在 Flutter 中，是通过实现 `State` 与 `setState` 来渲染和改变 `StatefulWidget` 的。如果使用了 `flutter_redux` 会有怎样的效果？

比如把用户信息存储在 `redux` 的 `store` 中，好处在于：比如某个页面修改了当前用户信息，所有绑定的该 `State` 的控件将由 `Redux` 自动同步修改。`State` 可以跨页面共享。

更多 `Redux` 的详细就不再展开，接下来我们讲讲 `flutter_redux` 的使用。在 `redux` 中主要引入了 `action`、`reducer`、`store` 概念。

- `action` 用于定义一个数据变化的请求。
- `reducer` 用于根据 `action` 产生新状态
- `store` 用于存储和管理 `state`，监听 `action`，将 `action` 自动分配给 `reducer` 并根据 `reducer` 的执行结果更新 `state`。

所以如下代码，我们先创建一个 `State` 用于存储需要保存的对象，其中关键代码在于 `UserReducer`。

```
///全局 Redux store 的对象，保存 State 数据
class GSYSate {
  ///用户信息
  User userInfo;

  ///构造方法
  GSYSate({this.userInfo});
}

///通过 Reducer 创建 用于 store 的 Reducer
GSYSate appReducer(GSYSate state, action) {
  return GSYSate(
    ///通过 UserReducer 将 GSYSate 内的 userInfo 和 action 关联在一起
```



```

        userInfo: UserReducer(state.userInfo, action),
    );
}

```

下面是上方使用的 `UserReducer` 的实现。这里主要通过 `TypedReducer` 将 `reducer` 的处理逻辑与定义的 `Action` 绑定，最后通过 `combineReducers` 返回 `Reducer<State>` 对象应用于上方 `Store` 中。

```

/// redux 的 combineReducers, 通过 TypedReducer 将 UpdateUserAction 与 reducers 关联起来
final UserReducer = combineReducers<User>([
    TypedReducer<User, UpdateUserAction>(_updateLoaded),
]);

/// 如果有 UpdateUserAction 发起一个请求时
/// 就会调用到 _updateLoaded
/// _updateLoaded 这里接受一个新的 userInfo, 并返回
User _updateLoaded(User user, action) {
    user = action.userInfo;
    return user;
}

/// 定一个 UpdateUserAction, 用于发起 userInfo 的改变
/// 类名随你喜欢定义, 只要通过上面 TypedReducer 绑定就好
class UpdateUserAction {
    final User userInfo;

    UpdateUserAction(this.userInfo);
}

```

下面正式在 Flutter 中引入 `store`, 通过 `StoreProvider` 将创建的 `store` 引用到 Flutter 中。

```

void main() {
    runApp(new FlutterReduxApp());
}

class FlutterReduxApp extends StatelessWidget {

    /// 创建 Store, 引用 GSYState 中的 appReducer 创建的 Reducer
    /// initialState 初始化 State
    final store = new Store<GSYState>(appReducer, initialState: new GSYState(userInfo: User.empty()));
}

```

```
FlutterReduxApp({Key key}) : super(key: key);

@override
Widget build(BuildContext context) {
  /// 通过 StoreProvider 应用 store
  return new StoreProvider(
    store: store,
    child: new MaterialApp(
      home: DemoUseStorePage(),
    ),
  );
}
```

在下方 DemoUseStorePage 中, 通过 StoreConnector 将 State 绑定到 Widget; 通过 StoreProvider.of 可以获取 state 对象; 通过 dispatch 一个 Action 可以更新 State。

```
class DemoUseStorePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    ///通过 StoreConnector 关联 GSYSate 中的 User
    return new StoreConnector<GSYSate, User>(
      ///通过 converter 将 GSYSate 中的 userInfo 返回
      converter: (store) => store.state.userInfo,
      ///在 userInfo 中返回实际渲染的控件
      builder: (context, userInfo) {
        return new Text(
          userInfo.name,
          style: Theme.of(context).textTheme.display1,
        );
      },
    );
  }
}

.....///通过 StoreProvider.of(context) (带有 StoreProvider 下的 context)
```

```

/// 可以任意的位置访问到 state 中的数据
StoreProvider.of(context).state.userInfo;

.....///通过 dispatch UpdateUserAction, 可以更新 State
StoreProvider.of(context).dispatch(new UpdateUserAction(newUserInfo));

```

看到这是不是有点想静静了？先不管静静是谁，但是 **Redux** 的实用性是应该比静静更吸引人，作为一个有追求的程序猿，多动手撸撸还有什么拿不下的山头是不？更详细的实现请看：[GSYGithubAppFlutter](#)。

4、数据库

在 GSYGithubAppFlutter 中，数据库使用的是 **sqflite** 的封装，其实就是 **sqlite** 语法的使用而已，有兴趣的可以看看完整代码 [DemoDb.dart](#)。这里主要提供一种思路，按照 **sqflite** 文档提供的方法，重新做了一小些修改，通过定义 **Provider** 操作数据库：

在 **Provider** 中定义表名与数据库字段常量，用于创建表与字段操作；

提供数据库与数据实体之间的映射，比如数据库对象与 **User** 对象之间的转化；

在调用 **Provider** 时才先判断表是否创建，然后再返回数据库对象进行用户查询。

如果结合网络请求，通过闭包实现，在需要数据库时先返回数据库，然后通过 **next** 方法将网络请求的方法返回，最后外部可以通过调用 **next** 方法再执行网络请求。如下所示：

```

UserDao.getUserInfo(userName, needDb: true).then((res) {

  ///数据库结果

  if (res != null && res.result) {

    setState(() {

      userInfo = res.data;

    });

  }

  return res.next;

}).then((res) {

  ///网络结果

  if (res != null && res.result) {

    setState(() {

      userInfo = res.data;

    });

  }

});

```

```

    });
  }
});

```

三、其他功能

其他功能，只是因为想不到标题。

1、返回按键监听

Flutter 中，通过 `WillPopScope` 嵌套，可以用于监听处理 Android 返回键的逻辑。其实 `WillPopScope` 并不是监听返回按键，如名字一般，是当前页面将要被 `pop` 时触发的回调。

通过 `onWillPop` 回调返回的 `Future`，判断是否响应 `pop`。下方代码实现按下返回键时，弹出提示框，按下确定退出 App。

```

class HomePage extends StatelessWidget {
  /// 单击提示退出
  Future<bool> _dialogExitApp(BuildContext context) {
    return showDialog(
      context: context,
      builder: (context) => new AlertDialog(
        content: new Text("是否退出"),
        actions: <Widget>[
          new FlatButton(onPressed: () => Navigator.of(context).pop(false), child: new Text("取消")),
          new FlatButton(
            onPressed: () {
              Navigator.of(context).pop(true);
            },
            child: new Text("确定"))
        ],
      ));
  }

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {

```

```
return WillPopScope(  
  onWillPop: () {  
    ///如果返回 return new Future.value(false); popped 就不会被处理  
    ///如果返回 return new Future.value(true); popped 就会触发  
    ///这里可以通过 showDialog 弹出确定框，在返回时通过 Navigator.of(context).pop(true);决定是否退出  
    return _dialogExitApp(context);  
  },  
  child: new Container(),  
);  
}  
}
```

2、前后台监听

`WidgetsBindingObserver` 包含了各种控件的生命周期通知，其中的 `didChangeAppLifecycleState` 就可以用于做前后台状态监听。

```
/// WidgetsBindingObserver 包含了各种控件的生命周期通知  
class _HomePageState extends State<HomePage> with WidgetsBindingObserver {  
  
  ///重写 WidgetsBindingObserver 中的 didChangeAppLifecycleState  
  @override  
  void didChangeAppLifecycleState(AppLifecycleState state) {  
    ///通过 state 判断 App 前后台切换  
    if (state == AppLifecycleState.resumed) {  
  
    }  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return new Container();  
  }  
}
```

3、键盘焦点处理

一般触摸收起键盘也是常见需求，如下代码所示，`GestureDetector` + `FocusScope` 可以满足这一需求。

```
class _LoginPageState extends State<LoginPage> {  
  @override  
  Widget build(BuildContext context) {  
    ///定义触摸层  
    return new GestureDetector(  
      ///透明也响应处理  
      behavior: HitTestBehavior.translucent,  
      onTap: () {  
        ///触摸手气键盘  
        FocusScope.of(context).requestFocus(new FocusNode());  
      },  
      child: new Container(  
      ),  
    );  
  }  
}
```

4、启动页

IOS 启动页，在 `ios/Runner/Assets.xcassets/LaunchImage.imageset/` 下，有 `Contents.json` 文件和启动图片，将你的启动页放置在这个目录下，并且修改 `Contents.json` 即可，具体尺寸自行谷歌即可。

Android 启动页，在 `android/app/src/main/res/drawable/launch_background.xml` 中已经有写好的启动页，`<item><bitmap>` 部分被屏蔽，只需要打开这个屏蔽，并且将你启动图修改为 `launch_image` 并放置到各个 `mipmap` 文件夹即可，记得各个文件夹下提供相对于大小尺寸的文件。

自此，第二篇终于结束了！(///▽///)





Flutter 完整开发实战详解(三、 打包与填坑篇)

作为系列文章的第三篇，继[篇章一](#)和[篇章二](#)之后，本篇将为你着重展示：**Flutter 开发过程的打包流程、APP 包对比、细节技巧与问题处理**。本篇主要描述的 Flutter 的打包、在开发过程中遇到的各类问题与细节，算是对上两篇的补全。

友情提示：本文所有代码均在 [GSYGithubAppFlutter](#)，要不试试？(๑_๑)。

一、打包

首先我们先看结果，如下表所示，是 **Flutter** 与 **React Native**、**IOS** 与 **Android** 的纵向与横向对比。

| 项目 | IOS | Android |
|----------------------------|--|---|
| GSYGithubAppFlutter |  IPA Runner-iPhone 6s Plus.ipa IOS 应用 - 14.3 MB flutter-ipa |  GSYGithubAppFlutter-1.0.8.apk 文档 - 11.2 MB flutter-apk |
| GSYGithubAppRN |  IPA GSYGithubAPP-iPhone 6s Plus.ipa IOS 应用 - 9.2 MB rn-ipa |  GSYGithubApp-1.9.apk 文档 - 17.4 MB rn-apk |

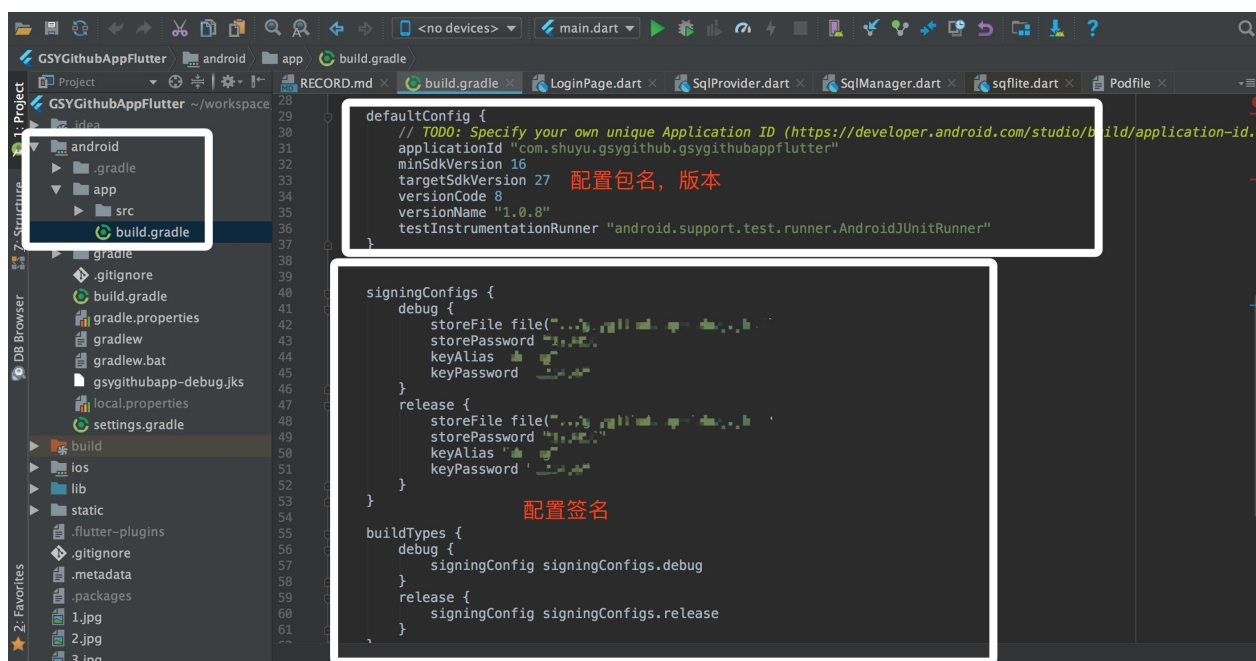
从上表我们可以看到：

Flutter 的 apk 会比 ipa 更小一些，这其中的一部分原因是 Flutter 使用的 **Skia** 在 Android 上是自带的。

横向对比 **React Native**，虽然项目不完全一样，但是大部分功能一致的情况下，Flutter 的 Apk 确实更小一些。这里又有一个细节，rn 的 ipa 包体积小很多，这其实是因为 **javascriptcore** 在 ios 上是内置的原因。

对上述内容有兴趣的可以看看《[移动端跨平台开发的深度解析](#)》。

1、Android 打包



I'm Android

在 Android 的打包上，笔者基本没有遇到什么问题，在 `android/app/build.gradle` 文件下，配置 `applicationId`、`versionCode`、`versionName` 和签名信息，最后通过 `flutter build app` 即可完成编译。编程成功的包在 `build/app/outputs/apk/release` 下。

2、IOS 打包与真机运行

在 IOS 的打包上，笔者倒是经历了一波曲折，这里主要讲笔者遇到的问题。

首先你需要一个 apple 开发者账号，然后创建证书、创建 AppId，创建配置文件、最后在 `info.plist` 文件下输入相关信息，更详细可看官方的《发布的 IOS 版 APP》的教程。

但由于笔者项目中使用了第三方的插件包如 `shared_preferences` 等，在执行 `Archive` 的过程却一直出现如下问题：

在 `Archive` 时提示找不到

```

#import <connectivity/ConnectivityPlugin.h> ///file not found
#import <device_info/DeviceInfoPlugin.h>
#import <flutter_statusbar/FlutterStatusbarPlugin.h>
#import <flutter_webview_plugin/FlutterWebviewPlugin.h>
  
```



```
#import <fluttertoast/FluttertoastPlugin.h>
#import <get_version/GetVersionPlugin.h>
#import <package_info/PackageInfoPlugin.h>
#import <share/SharePlugin.h>
#import <shared_preferences/SharedPreferencesPlugin.h>
#import <sqflite/SqflitePlugin.h>
#import <url_launcher/UrlLauncherPlugin.h>
```

通过 Android Studio 运行到 IOS 模拟器时没有任何问题，说明这不是第三方包问题。通过查找问题发现，在 IOS 执行 **Archive** 之前，需要执行 **flutter build release**，如下图在命令执行之后，Pod 的执行目录会发现改变，并且生成打包需要的文件。（ps 普通运行时自动又会修改回来）

| | |
|--|---|
| <pre>- connectivity (from `symlinks/plugins/connectivity/ios`) - device_info (from `symlinks/plugins/device_info/ios`) - Flutter (from `symlinks/flutter/ios`) - flutter_statusbar (from `symlinks/plugins/flutter_statusbar/ios`) - flutter_webview_plugin (from `symlinks/plugins/flutter_webview_p - fluttertoast (from `symlinks/plugins/fluttertoast/ios`) - get_version (from `symlinks/plugins/get_version/ios`) - package_info (from `symlinks/plugins/package_info/ios`) - share (from `symlinks/plugins/share/ios`) - shared_preferences (from `symlinks/plugins/shared_preferences/io - sqflite (from `symlinks/plugins/sqflite/ios`) - url_launcher (from `symlinks/plugins/url_launcher/ios`) SPEC REPOS: https://github.com/cocoapods/specs.git: - FMDB - Reachability EXTERNAL SOURCES: connectivity: :path: ".symlinks/plugins/connectivity/ios" device_info: :path: ".symlinks/plugins/device_info/ios" Flutter: :path: ".symlinks/flutter/ios" flutter_statusbar: :path: ".symlinks/plugins/flutter_statusbar/ios" flutter_webview_plugin:</pre> <p style="text-align: center;">Podfile.lock</p> <pre> XShellScriptBuildPhase section */ 87C2C48C36989195F6D5E /* [CP] Embed Pods Frameworks */ = { isa = PBXShellScriptBuildPhase; buildActionMask = 2147483647; files = (); inputPaths = ("\${SRCROOT}/Pods/Target Support Files/Pods-Runner/Pods-Runner-f "\${BUILT_PRODUCTS_DIR}/FMDB/FMDB.framework", "\${PODS_ROOT}/../symlinks/flutter/ios/Flutter.framework", "\${BUILT_PRODUCTS_DIR}/Reachability/Reachability.framework",); };</pre> <p style="text-align: center;">project.pbxproj</p> | <pre>- connectivity (from `symlinks/plugins/connectivity/ios`) - device_info (from `symlinks/plugins/device_info/ios`) - Flutter (from `symlinks/flutter/ios-release`) - flutter_statusbar (from `symlinks/plugins/flutter_statusbar/io - flutter_webview_plugin (from `symlinks/plugins/flutter_webview - fluttertoast (from `symlinks/plugins/fluttertoast/ios`) - get_version (from `symlinks/plugins/get_version/ios`) - package_info (from `symlinks/plugins/package_info/ios`) - share (from `symlinks/plugins/share/ios`) - shared_preferences (from `symlinks/plugins/shared_preferences/ - sqflite (from `symlinks/plugins/sqflite/ios`) - url_launcher (from `symlinks/plugins/url_launcher/ios`) SPEC REPOS: https://github.com/cocoapods/specs.git: - FMDB - Reachability EXTERNAL SOURCES: connectivity: :path: ".symlinks/plugins/connectivity/ios" device_info: :path: ".symlinks/plugins/device_info/ios" Flutter: :path: ".symlinks/flutter/ios-release" flutter_statusbar: :path: ".symlinks/plugins/flutter_statusbar/ios" flutter_webview_plugin:</pre> <pre> XShellScriptBuildPhase section */ 87C2C48C36989195F6D5E /* [CP] Embed Pods Frameworks */ = { isa = PBXShellScriptBuildPhase; buildActionMask = 2147483647; files = (); inputPaths = ("\${SRCROOT}/Pods/Target Support Files/Pods-Runner/Pods-Runner-fr "\${BUILT_PRODUCTS_DIR}/FMDB/FMDB.framework", "\${PODS_ROOT}/../symlinks/flutter/ios-release/Flutter.framework", "\${BUILT_PRODUCTS_DIR}/Reachability/Reachability.framework",); };</pre> |
|--|---|

文件变化

但是实际在执行 **flutter build release** 后，问题依然存在，最终翻山越岭(´ `□´) ㄟ——，终于找到两个答案：

[Issue#19241](#) 下描述了类似问题，但是他们因为路径问题导致，经过尝试并不能解决。

[Issue#18305](#) 真实的解决了这个问题，居然是因为 Pod 的工程没引入：

```
open ios/Runner.xcodeproj

I checked Runner/Pods is empty in Xcode sidebar.
```

```
drop Pods/Pods.xcodeproj into Runner/Pods.  
"Valid architectures" to only "arm64" (I removed armv7 armv7s)
```

最后终于成功打包，心累啊(///▽///)。同时如果希望直接在真机上调试 Flutter，可以参考：《Flutter 基础—开发环境与入门》下的 **IOS 真机** 部分。

二、细节

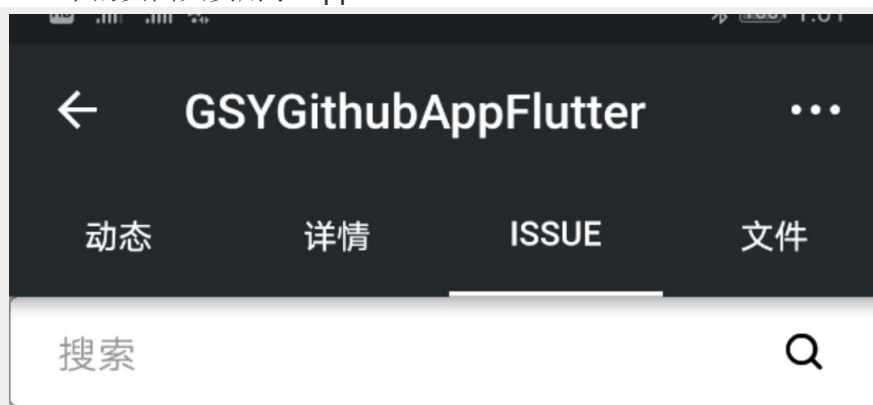
这里主要讲一些小细节

1、AppBar

在 Flutter 中 AppBar 算是常用 Widget，而 AppBar 可不仅仅作为标题栏和使用，AppBar 上的 **leading** 和 **bottom** 同样是有用的功能。

- AppBar 的 **bottom** 默认支持 **TabBar**，也就是常见的顶部 Tab 的效果，这其实是因为 **TabBar** 实现了 **PreferredSizeWidget** 的 **preferredSize**。

所以只要你的控件实现了 **preferredSize**，就可以放到 AppBar 的 **bottom** 中使用。比如下图搜索栏，这是 TabView 下的页面又实用了 AppBar。



leading：通常是左侧按键，不设置时一般是 **Drawer** 的图标或者返回按钮。

flexibleSpace：位于 **bottom** 和 **leading** 之间。

2、按键

Flutter 中的按键，如 **FlatButton** 默认是否有边距和最小大小的。所以如果你想要无 **padding**、**margin**、**border**、默认大小 等的按键效果，其中一种方式如下：

```
///new RawMaterialButton(  
    materialTapTargetSize: MaterialTapTargetSize.shrinkWrap,  
    padding: padding ?? const EdgeInsets.all(0.0),  
    constraints: const BoxConstraints(minWidth: 0.0, minHeight: 0.0),  
    child: child,  
    onPressed: onPressed);
```

如果在再上 **Flex** ，如下所示，一个可控的填充按钮就出来了。

```
new RawMaterialButton(  
    materialTapTargetSize: MaterialTapTargetSize.shrinkWrap,  
    padding: padding ?? const EdgeInsets.all(0.0),  
    constraints: const BoxConstraints(minWidth: 0.0, minHeight: 0.0),  
    ///flex  
    child: new Flex(  
        mainAxisAlignment: mainAxisAlignment,  
        direction: Axis.horizontal,  
        children: <Widget>[],  
    ),  
    onPressed: onPressed);
```

3、StatefulWidget 赋值

这里我们以给 **TextField** 主动赋值为例，其实 **Flutter** 中，给有状态的 **Widget** 传递状态或者数据，一般都是通过各种 **controller** 。如 **TextField** 的主动赋值，如下代码所示：

```
final TextEditingController controller = new TextEditingController();  
  
@override  
void didChangeDependencies() {  
    super.didChangeDependencies();  
    ///通过给 controller 的 value 新创建一个 TextEditingValue  
    controller.value = new TextEditingValue(text: "给输入框填入参数");  
}
```

```

@override
Widget build(BuildContext context) {
  return new TextField(
    ///controller
    controller: controller,
    onChanged: onChanged,
    obscureText: obscureText,
    decoration: new InputDecoration(
      hintText: hintText,
      icon: iconData == null ? null : new Icon(iconData),
    ),
  );
}

```

其实 `TextEditingValue` 是 `ValueNotifier`，其中 `value` 的 `setter` 方法被重载，一旦改变就会触发 `notifyListeners` 方法。而 `TextEditingController` 中，通过调用 `addListener` 就监听了数据的改变，从而让 UI 更新。

当然，赋值有更简单粗暴的做法是：传递一个对象 `class A` 对象，在控件内部使用对象 `A.b` 的变量绑定控件，外部通过 `setState({ A.b = b2 })` 更新。

4、GlobalKey

在 Flutter 中，要主动改变子控件的状态，还可以使用 `GlobalKey`。比如你需要主动调用 `RefreshIndicator` 显示刷新状态，如下代码所示。

```

GlobalKey<RefreshIndicatorState> refreshIndicatorKey;

showForRefresh() {
  ///显示刷新
  refreshIndicatorKey.currentState.show();
}

@override
Widget build(BuildContext context) {
  refreshIndicatorKey = new GlobalKey<RefreshIndicatorState>();
}

```

```

return new RefreshIndicator(
  key: refreshIndicatorKey,
  onRefresh: onRefresh,
  child: new ListView.builder(
    ///.....
  ),
);
}

```

5、Redux 与主题

使用 Redux 来做 Flutter 的全局 State 管理最合适不过，由于 Redux 内容较多，如果感兴趣的可以看看 [篇章二](#)，这里主要通过 Redux 来实现实时切换主题的效果。

如下代码，通过 `StoreProvider` 加载了 `store`，再通过 `StoreBuilder` 将 `store` 中的 `themeData` 绑定到 `MaterialApp` 的 `theme` 下，之后在其他 `Widget` 中通过 `Theme.of(context)` 调你需要的颜色，最终在任意位置调用 `store.dispatch` 就可实时修改主题，效果如后图所示。

```

class FlutterReduxApp extends StatelessWidget {
  final store = new Store<GSYState>(
    appReducer,
    initialState: new GSYState(
      themeData: new ThemeData(
        primarySwatch: GSYColors.primarySwatch,
      ),
    ),
  );

  FlutterReduxApp({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    /// 通过 StoreProvider 应用 store
    return new StoreProvider(
      store: store,
      ///通过 StoreBuilder 获取 themeData
      child: new StoreBuilder<GSYState>(builder: (context, store) {

```

```

return new MaterialApp(
  theme: store.state.themeData,
  routes: {
    HomePage.sName: (context) {
      return HomePage();
    },
  });
}),
);
}
}

```



主题

6、Hotload 与 Package

Flutter 在 Debug 和 Release 下分别是 JIT 和 AOT 模式，而在 DEBUG 下，是支持 Hotload 的，而且十分丝滑。但是需要注意的是：如果开发过程中安装了新的第三方包，而新的第三方包如果包含了原生代码，需要停止后重新运行哦。

`pubspec.yaml` 文件下就是我们的包依赖目录，其中 `^` 代表大于等于，一般情况下 `upgrade` 和 `get` 都能达到下载包的作用。但是：`upgrade` 会在包有更新的情况下，更新 `pubspec.lock` 文件下包的版本。

三、问题处理

- `Waiting for another flutter command to release the startup lock`：如果遇到这个问题：

- 1、打开 flutter 的安装目录 `/bin/cache/`
- 2、删除 `lockfile` 文件
- 3、重启 AndroidStudio

dialog 下的黄色线

`yellow-lines-under-text-widgets-in-flutter`: `showDialog` 中，默认是没使用 `Scaffold`，这回导致文本有黄色溢出线提示，可以使用 `Material` 包一层处理。

TabBar + TabView + KeepAlive 的问题

可以通过 TabBar + PageView 解决，具体可见 [篇章二](#)。

-

自此，第三篇终于结束了！(///▽///)

Flutter 完整开发实战详解(四、 Redux、主题、国际化)

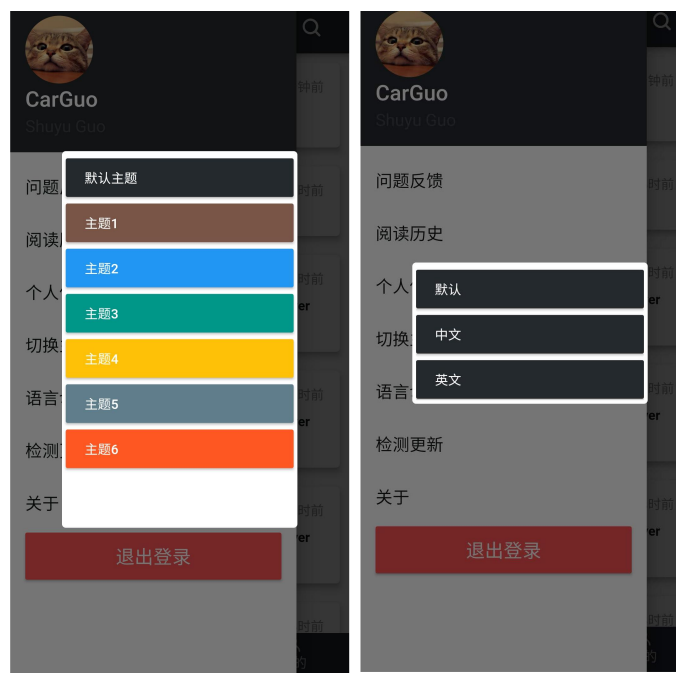
作为系列文章的第四篇，本篇主要介绍 Flutter 中 Redux 的使用，并结合 Redux 完成实时的主题切换与多语言切换功能。

前文：

- [一、Dart 语言和 Flutter 基础](#)
- [二、快速开发实战篇](#)
- [三、打包与填坑篇](#)

Flutter 作为响应式框架，通过 `state` 实现跨帧渲染的逻辑，难免让人与 React 和 React Native 联系起来，而其中 React 下“广为人知”的 **Redux** 状态管理，其实在 Flutter 中同样适用。

我们最终将实现如下图的效果，相应代码在 [GSYGithubAppFlutter](#) 中可找到，本篇 Flutter 中所使用的 Redux 库是 [flutter_redux](#)。



Let's do it

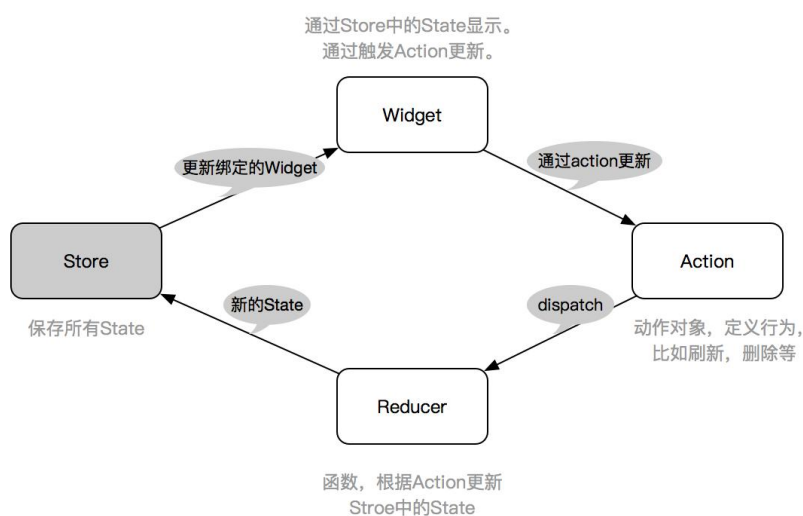
一、Redux

Redux 的概念是**状态管理**，那在已有 `state` 的基础上，为什么还需要 Redux ？

因为使用 Redux 的好处是：**共享状态**和**单一数据**。

试想一下，App 内有多个地方使用到登陆用户的数据，这时候如果某处对用户数据做了修改，各个页面的同步更新会是一件麻烦的事情。

但是引入 Redux 后，某个页面修改了当前用户信息，所有绑定了 Redux 的控件，将由 Redux 自动同步刷新。**See!** 这在一定程度节省了我们的工作量，并且单一数据源在某些场景下也方便管理。同理我们后面所说的 主题 和 多语言 切换也是如此。



大致流程图

如上图，Redux 的主要由三部分组成：**Store**、**Action**、**Reducer**。

- **Action** 用于定义一个数据变化的请求行为。
- **Reducer** 用于根据 **Action** 产生新状态，一般是一个方法。
- **Store** 用于存储和管理 **state**。

所以一般流程为：

- 1、Widget 绑定了 Store 中的 **state** 数据。
- 2、Widget 通过 **Action** 发布一个动作。
- 3、Reducer 根据 **Action** 更新 **state**。
- 4、更新 Store 中 **state** 绑定的 **Widget**。

根据这个流程，首先我们要创建一个 **Store**。

如下图，创建 **Store** 需要 **reducer**，而 **reducer** 实际上是一个带有 **state** 和 **action** 的方法，并返回新的 **State**。

```

Store(
  this.reducer, {
    State initialState,
    List<Middleware<State>> middleware = const [],
    bool syncStream: false,

    /// If set to true, the Store will not emit onChange events if the new State
    /// that is returned from your [reducer] in response to an Action is equal
    /// to the previous state.
    ///
    /// Under the hood, it will use the `==` method from your State class to
    /// determine whether or not the two States are equal.
    bool distinct: false,
  })
),
typedef State Reducer<State>(State state, dynamic action);

```

所以我们需要先创建一个 `State` 对象 `GSYState` 类，用于储存需要共享的数据。比如下方代码的：用户信息、主题、语言环境 等。

接着我们需要定义 `Reducer` 方法 `appReducer`：将 `GSYState` 内的每一个参数，和对应的 `action` 绑定起来，返回完整的 `GSYState`。这样我们就确定了 `State` 和 `Reducer` 用于创建 `Store`。

```

///全局 Redux store 的对象，保存 State 数据
class GSYState {
  ///用户信息
  User userInfo;

  ///主题
  ThemeData themeData;

  ///语言
  Locale locale;

  ///构造方法
  GSYState({this.userInfo, this.themeData, this.locale});
}

///创建 Reducer
///源码中 Reducer 是一个方法 typedef State Reducer<State>(State state, dynamic action);
///我们自定义了 appReducer 用于创建 store
GSYState appReducer(GSYState state, action) {
  return GSYState(
    ///通过自定义 UserReducer 将 GSYState 内的 userInfo 和 action 关联在一起
    userInfo: UserReducer(state.userInfo, action),

```

```

    ///通过自定义 ThemeDataReducer 将 GSYSState 内的 themeData 和 action 关联在一起
    themeData: ThemeDataReducer(state.themeData, action),

    ///通过自定义 LocaleReducer 将 GSYSState 内的 locale 和 action 关联在一起
    locale: LocaleReducer(state.locale, action),

  );
}

```

如上代码，**GSYSState** 的每一个参数，是通过独立的自定义 **Reducer** 返回的。比如 **themeData** 是通过 **ThemeDataReducer** 方法产生的，**ThemeDataReducer** 其实是将 **ThemeData** 和一系列 **Theme** 相关的 **Action** 绑定起来，用于和其他参数分开。这样就可以独立的维护和管理 **GSYSState** 中的每一个参数。

继续上面流程，如下代码所示，通过 **flutter_redux** 的 **combineReducers** 与 **TypedReducer**，将 **RefreshThemeDataAction** 类和 **_refresh** 方法绑定起来，最终会返回一个 **ThemeData** 实例。也就是说：用户每次发出一个 **RefreshThemeDataAction**，最终都会触发 **_refresh** 方法，然后更新 **GSYSState** 中的 **themeData**。

```

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

///通过 flutter_redux 的 combineReducers, 创建 Reducer<State>
final ThemeDataReducer = combineReducers<ThemeData>([
  ///将 Action, 处理 Action 动作的方法, State 绑定
  TypedReducer<ThemeData, RefreshThemeDataAction>(_refresh),
]);

///定义处理 Action 行为的方法, 返回新的 State
ThemeData _refresh(ThemeData themeData, action) {
  themeData = action.themeData;
  return themeData;
}

///定义一个 Action 类///将该 Action 在 Reducer 中与处理该 Action 的方法绑定
class RefreshThemeDataAction {

  final ThemeData themeData;

  RefreshThemeDataAction(this.themeData);
}

```

OK, 现在我们可以愉快地创建 **Store** 了。如下代码所示, 在创建 **Store** 的同时, 我们通过 **initialState** 对 **GSYState** 进行了初始化, 然后通过 **StoreProvider** 加载了 **Store** 并且包裹了 **MaterialApp**。至此我们完成了 **Redux** 中的初始化构建。

```
void main() {
  runApp(new FlutterReduxApp());
}

class FlutterReduxApp extends StatelessWidget {
  /// 创建 Store, 引用 GSYState 中的 appReducer 创建 Reducer
  /// initialState 初始化 State
  final store = new Store<GSYState>(
    appReducer,
    initialState: new GSYState(
      userInfo: User.empty(),
      themeData: new ThemeData(
        primarySwatch: GSYColors.primarySwatch,
      ),
      locale: Locale('zh', 'CH')),
  );

  FlutterReduxApp({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    /// 通过 StoreProvider 应用 store
    return new StoreProvider(
      store: store,
      child: new MaterialApp(),
    );
  }
}
```

And then, 接下来就是使用了。如下代码所示, 通过在 **build** 中使用 **StoreConnector**, 通过 **converter** 转化 **store.state** 的数据, 最后通过 **builder** 返回实际需要渲染的控件, 这样就完成了**数据和控件的绑定**。当然, 你也可以使用 **StoreBuilder**。

```
class DemoUseStorePage extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    ///通过 StoreConnector 关联 GSYSState 中的 User

    return new StoreConnector<GSYSState, User>(

      ///通过 converter 将 GSYSState 中的 userInfo 返回

      converter: (store) => store.state.userInfo,

      ///在 userInfo 中返回实际渲染的控件

      builder: (context, userInfo) {

        return new Text(

          userInfo.name,

        );

      },

    );

  }

}
```

最后，当你需要触发更新的时候，只需要如下代码即可。

```
StoreProvider.of(context).dispatch(new UpdateUserAction(newUserInfo));
```

So，或者简单的业务逻辑下，**Redux** 并没有什么优势，甚至显得繁琐。但是一旦框架搭起来，在复杂的业务逻辑下就会显示格外愉悦了。

二、主题

Flutter 中官方默认就支持主题设置，`MaterialApp` 提供了 `theme` 参数设置主题，之后可以通过 `Theme.of(context)` 获取到当前的 `ThemeData` 用于设置控件的颜色字体等。

`ThemeData` 的创建提供很多参数，这里主要说 `primarySwatch` 参数。`primarySwatch` 是一个 `MaterialColor` 对象，内部由 10 种不同深浅的颜色组成，用来做主题色调再合适不过。

如下图和代码所示，Flutter 默认提供了很多主题色，同时我们也可以通过 `MaterialColor` 实现自定义的主题色。



```
MaterialColor primarySwatch = const MaterialColor(
  primaryValue,
  const <int, Color>{
    50: const Color(primaryLightValue),
    100: const Color(primaryLightValue),
    200: const Color(primaryLightValue),
    300: const Color(primaryLightValue),
    400: const Color(primaryLightValue),
    500: const Color(primaryValue),
    600: const Color(primaryDarkValue),
    700: const Color(primaryDarkValue),
    800: const Color(primaryDarkValue),
    900: const Color(primaryDarkValue),
  },
);
```

那如何实现实时的主题切换呢？当然是通过 `Redux` 啦！

前面我们已经在 `GSYState` 中创建了 `themeData`，此时将它设置给 `MaterialApp` 的 `theme` 参数，之后我们通过 `dispatch` 改变 `themeData` 即可实现主题切换。

注意，因为你的 **MaterialApp** 也是一个 **StatefulWidget**，如下代码所示，还需要利用 **StoreBuilder** 包裹起来，之后我们就可以通过 **dispatch** 修改主题，通过 **Theme.of(context).primaryColor** 获取主题色啦。

```
@override
Widget build(BuildContext context) {
  /// 通过 StoreProvider 应用 store
  return new StoreProvider(
    store: store,
    child: new StoreBuilder<GSYState>(builder: (context, store) {
      return new MaterialApp(
        theme: store.state.themeData);
    }),
  );
}

....

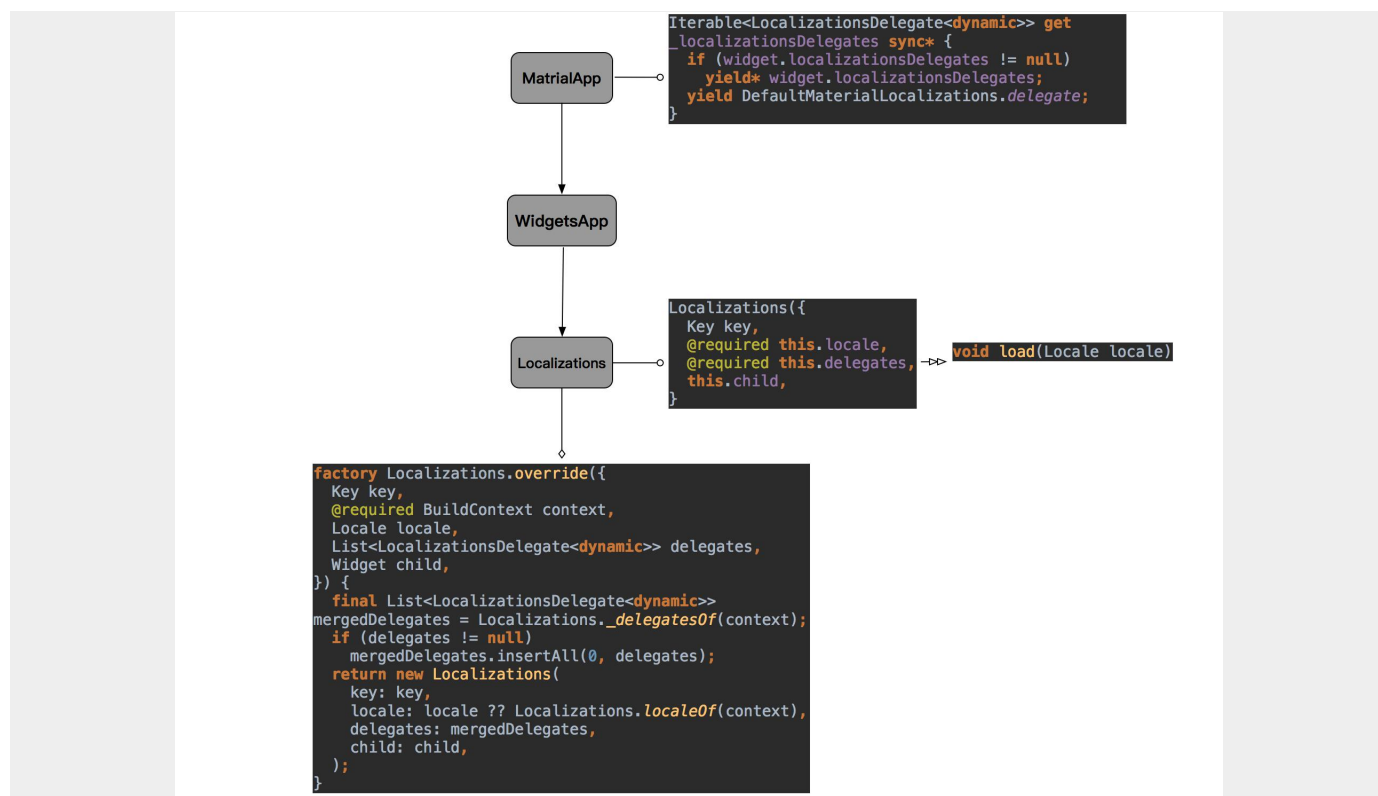
ThemeData themeData = new ThemeData(primarySwatch: colors[index]);
store.dispatch(new RefreshThemeDataAction(themeData));
```



愉悦的切换

三、国际化

Flutter 的国际化按照官网文件 [internationalization](#) 看起来稍微有些复杂，也没有提及实时切换，所以这里介绍下快速的实现。当然，少不了 **Redux** ！



大致流程

如上图所示大致流程，同样是通过默认 `MaterialApp` 设置，自定义的多语言需要实现的是：`LocalizationsDelegate` 和 `Localizations`。最终流程会通过 `Localizations` 使用 `Locale` 加载这个 `delegate`。所以我们要做的是：

- 实现 `LocalizationsDelegate`。
- 实现 `Localizations`。
- 通过 `Store` 的 `Locale` 切换语言。

如下代码所示，创建自定义 `delegate` 需要继承 `LocalizationsDelegate` 对象，其中主要实现 `load` 方法。我们可以是通过方法的 `locale` 参数，判断需要加载的语言，然后返回我们自定义好多语言实现类 `GSYLocalizations`，最后通过静态 `delegate` 对外提供 `LocalizationsDelegate`。

```

/**
 * 多语言代理
 * Created by guoshuyu
 * Date: 2018-08-15
 */
class GSYLocalizationsDelegate extends LocalizationsDelegate<GSYLocalizations> {

  GSYLocalizationsDelegate();
}

```



```

@override

bool isSupported(Locale locale) {

    ///支持中文和英语

    return ['en', 'zh'].contains(locale.languageCode);

}

///根据 locale, 创建一个对象用于提供当前 locale 下的文本显示

@override

Future<GSYLocalizations> load(Locale locale) {

    return new SynchronousFuture<GSYLocalizations>(new GSYLocalizations(locale));

}

@override

bool shouldReload(LocalizationsDelegate<GSYLocalizations> old) {

    return false;

}

///全局静态的代理

static GSYLocalizationsDelegate delegate = new GSYLocalizationsDelegate();

}

```

上面提到的 `GSYLocalizations` 其实是一个自定义对象，如下代码所示，它会根据创建时的 `Locale`，通过 `locale.languageCode` 判断返回对应的语言实体：`GSYStringBase` 的实现类。

因为 `GSYLocalizations` 对象最后会通过 `Localizations` 加载，所以 `Locale` 也是在那时，通过 `delegate` 赋予。同时在该 `context` 下，可以通过 `Localizations.of` 获取 `GSYLocalizations`，比如：`GSYLocalizations.of(context).currentLocalized.app_name`。

```

///自定义多语言实现 class GSYLocalizations {

    final Locale locale;

    GSYLocalizations(this.locale);

    ///根据不同 locale.languageCode 加载不同语言对应

    ///GSYStringEn 和 GSYStringZh 都继承了 GSYStringBase

    static Map<String, GSYStringBase> _localizedValues = {

        'en': new GSYStringEn(),

```

```

    'zh': new GSYSStringZh(),
  };

GSYSStringBase get currentLocalized {
  return _localizedValues[locale.languageCode];
}

///通过 Localizations 加载当前的 GSYLocalizations
///获取对应的 GSYSStringBase
static GSYLocalizations of(BuildContext context) {
  return Localizations.of(context, GSYLocalizations);
}
}

///语言实体基类
abstract class GSYSStringBase {
  String app_name;
}

///语言实体实现类
class GSYSStringEn extends GSYSStringBase {
  @override
  String app_name = "GSYGithubAppFlutter";
}

///使用
GSYLocalizations.of(context).currentLocalized.app_name

```

说完了 `delegate`，接下来就是 `Localizations` 了。在上面的流程图中可以看到，`Localizations` 提供一个 `override` 方法构建 `Localizations`，这个方法中可以设置 `locale`，而我們需要的正是实时的动态切换语言显示。

如下代码，我们创建一个 `GSYLocalizations` 的 `Widget`，通过 `StoreBuilder` 绑定 `Store`，然后通过 `Localizations.override` 包裹我们需要构建的页面，将 `Store` 中的 `locale` 和 `Localizations` 的 `locale` 绑定起来。

```

class GSYLocalizations extends StatefulWidget {
  final Widget child;

  GSYLocalizations({Key key, this.child}) : super(key: key);
}

```

```

@override
State<GSYLocalizations> createState() {
  return new _GSYLocalizations();
}
}class _GSYLocalizations extends State<GSYLocalizations> {

@override
Widget build(BuildContext context) {
  return new StoreBuilder<GSYState>(builder: (context, store) {
    ///通过 StoreBuilder 和 Localizations 实现实时多语言切换
    return new Localizations.override(
      context: context,
      locale: store.state.locale,
      child: widget.child,
    );
  });
}
}

```

如下代码，最后将 `GSYLocalizations` 使用到 `MaterialApp` 中。通过 `store.dispatch` 切换 `Locale` 即可。

```

@override
Widget build(BuildContext context) {
  /// 通过 StoreProvider 应用 store
  return new StoreProvider(
    store: store,
    child: new StoreBuilder<GSYState>(builder: (context, store) {
      return new MaterialApp(
        ///多语言实现代理
        localizationsDelegates: [
          GlobalMaterialLocalizations.delegate,
          GlobalWidgetsLocalizations.delegate,
          GSYLocalizationsDelegate.delegate,
        ],

```

```
    locale: store.state.locale,
    supportedLocales: [store.state.locale],
    routes: {
      HomePage.sName: (context) {
        ///通过 Localizations.override 包裹一层。---这里
        return new GSYLocalizations(
          child: new HomePage(),
        );
      },
    });
  });
});

///切换主题
static changeLocale(Store<GSYState> store, int index) {
  Locale locale = store.state.platformLocale;
  switch (index) {
    case 1:
      locale = Locale('zh', 'CH');
      break;
    case 2:
      locale = Locale('en', 'US');
      break;
  }
  store.dispatch(RefreshLocaleAction(locale));
}
```

最后的最后，在改变时记录状态，在启动时取出后 `dispatch`，至此主题和多语言设置完成。

自此，第四篇终于结束了！(///▽///)

Flutter 完整开发实战详解(五、 深入探索)

作为系列文章的第五篇，本篇主要探索下 Flutter 中的一些有趣原理，帮助我们更好的去理解和开发。

前文：

- [一、Dart 语言和 Flutter 基础](#)
- [二、快速开发实战篇](#)
- [三、打包与填坑篇](#)
- [四、Redux、主题、国际化](#)

一、WidgetsFlutterBinding

这是一个胶水类。

1、Mixins

混入其中(￣.￣)!

是的，Flutter 使用的是 Dart 支持 Mixin，而 Mixin 能够更好的解决多继承中容易出现的问题，如：方法优先顺序混乱、参数冲突、类结构变得复杂化等等。

Mixin 的定义解释起来会比较绕，我们直接代码从中出吧。如下代码所示，在 Dart 中 `with` 就是用于 mixins。可以看出，`class G extends B with A, A2`，在执行 G 的 `a`、`b`、`c` 方法后，输出了 `A2.a()`、`A.b()`、`B.c()`。所以结论上简单来说，就是相同方法被覆盖了，并且 `with` 后面的会覆盖前面的。

```
class A {
  a() {
    print("A.a()");
  }

  b() {
    print("A.b()");
  }
}

class A2 {
  a() {
    print("A2.a()");
  }
}
```

```

class B {
  a() {
    print("B.a()");
  }

  b() {
    print("B.b()");
  }

  c() {
    print("B.c()");
  }
}

class G extends B with A, A2 {

}

testMixins() {
  G t = new G();
  t.a();
  t.b();
  t.c();
}

/// *****输出*****
///I/flutter (13627): A2.a()
///I/flutter (13627): A.b()
///I/flutter (13627): B.c()

```

接下来我们继续修改下代码。如下所示，我们定义了一个 `Base` 的抽象类，而 `A`、`A2`、`B` 都继承它，同时再 `print` 之后执行 `super()` 操作。

从最后的输入我们可以看出，`A`、`A2`、`B` 中的所有方法都被执行了，且只执行了一次，同时执行的顺序也是和 `with` 的顺序有关。如果你把下方代码中 `class A.a()` 方法的 `super` 去掉，那么你将看不到 `B.a()` 和 `base a()` 的输出。

```
abstract class Base {  
  a() {  
    print("base a()");  
  }  
  
  b() {  
    print("base b()");  
  }  
  
  c() {  
    print("base c()");  
  }  
}  
  
class A extends Base {  
  a() {  
    print("A.a()");  
    super.a();  
  }  
  
  b() {  
    print("A.b()");  
    super.b();  
  }  
}  
  
class A2 extends Base {  
  a() {  
    print("A2.a()");  
    super.a();  
  }  
}  
  
class B extends Base {  
  a() {  
    print("B.a()");  
    super.a();  
  }  
  
  b() {
```

```
    print("B.b()");
    super.b();
  }

  c() {
    print("B.c()");
    super.c();
  }
}

class G extends B with A, A2 {

}

testMixins() {
  G t = new G();
  t.a();
  t.b();
  t.c();
}

///I/flutter (13627): A2.a()
///I/flutter (13627): A.a()
///I/flutter (13627): B.a()
///I/flutter (13627): base a()
///I/flutter (13627): A.b()
///I/flutter (13627): B.b()
///I/flutter (13627): base b()
///I/flutter (13627): B.c()
///I/flutter (13627): base c()
```

2、WidgetsFlutterBinding

说了那么多，那 Mixins 在 Flutter 中到底有什么用呢？这时候我们就要看 Flutter 中的“胶水类”：`WidgetsFlutterBinding`。

`WidgetsFlutterBinding` 在 Flutter 启动时 `runApp` 会被调用，作为 App 的入口，它肯定需要承担各类的初始化以及功能配置，这种情况下，Mixins 的作用就体现出来了。


```
// A concrete binding for applications based on the Widgets framework.  
// This is the glue that binds the framework to the Flutter engine.  
class WidgetsFlutterBinding extends BindingBase with GestureBinding, ServicesBinding, SchedulerBinding, PaintingBinding, SemanticsBinding, RendererBinding, WidgetsBinding {  
  
  /// Returns an instance of the [WidgetsBinding], creating and  
  /// initializing it if necessary. If one is created, it will be a  
  /// [WidgetsFlutterBinding]. If one was previously initialized, then  
  /// it will at least implement [WidgetsBinding].  
  ///  
  /// You only need to call this method if you need the binding to be  
  /// initialized before calling [runApp].  
  ///  
  /// In the "flutter_test" framework, [testWidgets] initializes the  
  /// binding instance to a [TestWidgetsFlutterBinding], not a  
  /// [WidgetsFlutterBinding].  
  static WidgetsBinding ensureInitialized() {  
    if (WidgetsBinding.instance == null) {  
      new WidgetsFlutterBinding();  
      return WidgetsBinding.instance;  
    }  
  }  
  
}  
  
class WidgetsBinding extends BindingBase with SchedulerBinding, GestureBinding, RendererBinding {  
  // class is intended to be used as a mixin, and should not be  
  
  // glue between the render tree and the Flutter engine.  
  class RendererBinding extends BindingBase with ServicesBinding, SchedulerBinding, SemanticsBinding, HitTestable  
  
  // TODO(jonahwilliams): move the remaining semantic related bindings here.  
  class SemanticsBinding extends BindingBase with ServicesBinding {  
  
  // mixes the [ServicesBinding] to be mixed in earlier.  
  class PaintingBinding extends BindingBase with ServicesBinding {  
  
  schedulingStrategy].  
  class SchedulerBinding extends BindingBase with ServicesBinding {  
  
  et}).  
  class ServicesBinding extends BindingBase {  
  
  nding for the gesture subsystem.  
  class GestureBinding extends BindingBase with HitTestable, HitTestDispatcher, HitTestTarget
```

从上图我们可以看出，**WidgetsFlutterBinding** 本身是并没有什么代码，主要是继承了 **BindingBase**，而后通过 **with** 黏上去的各类 **Binding**，这些 **Binding** 也都继承了 **BindingBase**。

看出来没，这里每个 **Binding** 都可以被单独使用，也可以被“黏”到 **WidgetsFlutterBinding** 中使用，这样做的效果，是不是比起一级一级继承的结构更加清晰了？

最后我们打印下执行顺序，如下图所以，不出所料、 $(\neg \nabla \neg)$ ！

```
I/flutter ( 1864): WidgetsFlutterBinding
I/flutter ( 1864): WidgetsBinding initInstances
I/flutter ( 1864): RendererBinding initInstances
I/flutter ( 1864): SemanticsBinding initInstances
I/flutter ( 1864): PaintingBinding initInstances
I/flutter ( 1864): SchedulerBinding initInstances
I/flutter ( 1864): ServicesBinding initInstances
I/flutter ( 1864): GestureBinding initInstances
I/flutter ( 1864): BindingBase initInstances
```

二、InheritedWidget

InheritedWidget 是一个抽象类，在 Flutter 中扮演者十分重要的角色，或者你并未直接使用过它，但是你肯定使用过和它相关的封装。

```

abstract class InheritedWidget extends ProxyWidget {
  /// Abstract const constructor. This constructor enables subclasses to provide
  /// const constructors so that they can be used in const expressions.
  const InheritedWidget({ Key key, Widget child })
    : super(key: key, child: child);

  @override
  InheritedElement createElement() => new InheritedElement(this);

  /// Whether the framework should notify widgets that inherit from this widget.
  ///
  /// When this widget is rebuilt, sometimes we need to rebuild the widgets that
  /// inherit from this widget but sometimes we do not. For example, if the data
  /// held by this widget is the same as the data held by `oldWidget`, then then
  /// we do not need to rebuild the widgets that inherited the data held by
  /// `oldWidget`.
  ///
  /// The framework distinguishes these cases by calling this function with the
  /// widget that previously occupied this location in the tree as an argument.
  /// The given widget is guaranteed to have the same [runtimeType] as this
  /// object.
  @protected
  bool updateShouldNotify(covariant InheritedWidget oldWidget);
}

```

如上图所示，**InheritedWidget** 主要实现两个方法：

创建了 **InheritedElement**，该 **Element** 属于特殊 **Element**，主要增加了将自身也添加到映射关系表 **_inheritedWidgets** 【注 1】，方便子孙 **element** 获取；同时通过 **notifyClients** 方法来更新依赖。

增加了 **updateShouldNotify** 方法，当方法返回 **true** 时，那么依赖该 **Widget** 的实例就会更新。

所以我们可以简单理解：**InheritedWidget** 通过 **InheritedElement** 实现了由下往上查找的支持（因为自身添加到 **_inheritedWidgets**），同时具备更新其子孙的功能。

注 1：每个 **Element** 都有一个 **_inheritedWidgets**，它是一个 **HashMap<Type, InheritedElement>**，它保存了上层节点中出现的 **InheritedWidget** 与其对应 **element** 的映射关系。

```

/// incorporated into the tree in the future.
abstract class Element extends DiagnosticableTree implements BuildContext {
  /// Creates an element that uses the given widget as its configuration.
  ///
  /// Typically called by an override of [Widget.createElement].
  Element(Widget widget)
    : assert(widget != null),
      _widget = widget;

  Element _parent;
}

```

接着我们看 **BuildContext**，如上图，**BuildContext** 其实只是接口，**Element** 实现了它。

InheritedElement 是 **Element** 的子类，所以每一个 **InheritedElement** 实例是一个 **BuildContext** 实例。同时我们日常使用中传递的 **BuildContext** 也都是一个 **Element**。

所以当我们遇到需要共享 **State** 时，如果逐层传递 **state** 去实现共享会显示过于麻烦，那么了解了上面的 **InheritedWidget** 之后呢？

是否将需要共享的 **State**，都放在一个 **InheritedWidget** 中，然后在使用的 **widget** 中直接取用就可以呢？答案是肯定的！所以如下方这类代码：通常如 焦点、主题色、多语言、用户信息 等都属于 App 内的全局共享数据，他们都会通过 **BuildContext (InheritedElement)** 获取。

```
///收起键盘
FocusScope.of(context).requestFocus(new FocusNode());

/// 主题色
Theme.of(context).primaryColor

/// 多语言
Localizations.of(context, GSYLocalizations)

/// 通过 Redux 获取用户信息
StoreProvider.of(context).userInfo

/// 通过 Redux 获取用户信息
StoreProvider.of(context).userInfo

/// 通过 Scope Model 获取用户信息
ScopedModel.of<UserInfo>(context).userInfo
```

综上所述，我们从先 **Theme** 入手。

如下方代码所示，通过给 **MaterialApp** 设置主题数据，通过 **Theme.of(context)** 就可以获取到主题数据并绑定使用。当 **MaterialApp** 的主题数据变化时，对应的 **Widget** 颜色也会发生变化，这是为什么呢(≡`°)!!?

```
///添加主题
new MaterialApp(
  theme: ThemeData.dark()
);

///使用主题色
new Container( color: Theme.of(context).primaryColor,
```

通过源码一层层查找，可以发现这样的嵌套：**MaterialApp -> AnimatedTheme -> Theme -> _InheritedTheme extends InheritedWidget**，所以通过 **MaterialApp** 作为入口，其实就是嵌套在 **InheritedWidget** 下。

```
//
static ThemeData of(BuildContext context, { bool shadowThemeOnly = false }) {
  final _InheritedTheme inheritedTheme =
    context.inheritFromWidgetOfExactType(_InheritedTheme);
  if (shadowThemeOnly) {
    if (inheritedTheme == null || inheritedTheme.theme.isMaterialAppTheme)
      return null;
    return inheritedTheme.theme.data;
  }

  final ThemeData colorTheme = (inheritedTheme != null) ? inheritedTheme.theme.data : _kFallbackTheme;
  final MaterialLocalizations localizations = MaterialLocalizations.of(context);
  final TextTheme geometryTheme = localizations?.localTextGeometry ?? MaterialTextGeometry.englishLike;
  return ThemeData.localize(colorTheme, geometryTheme);
}
```

如上图所示，通过 `Theme.of(context)` 获取到的主题数据，其实是通过 `context.inheritFromWidgetOfExactType(_InheritedTheme)` 去获取的，而 `Element` 中实现了 `BuildContext` 的 `inheritFromWidgetOfExactType` 方法，如下所示：

```
@override
InheritedWidget inheritFromWidgetOfExactType(Type targetType) {
  assert(_debugCheckStateIsActiveForAncestorLookup());
  final InheritedElement ancestor = _inheritedWidgets == null ? null : _inheritedWidgets[targetType];
  if (ancestor != null) {
    assert(ancestor is InheritedElement);
    _dependencies ??= new HashSet<InheritedElement>();
    _dependencies.add(ancestor);
    ancestor._dependents.add(this);
    return ancestor.widget;
  }
  _hadUnsatisfiedDependencies = true;
  return null;
}
```

那么，还记得上面说的 `_inheritedWidgets` 吗？既然 `InheritedElement` 已经存在于 `_inheritedWidgets` 中，拿出来用就对了。

前文: `InheritedWidget` 内的 `InheritedElement`，该 `Element` 属于特殊 `Element`，主要增加了将自身也添加到映射关系表 `_inheritedWidgets` 最后，如下图所示，在 `InheritedElement` 中，`notifyClients` 通过 `InheritedWidget` 的 `updateShouldNotify` 方法判断是否更新，比如在 `Theme` 的 `_InheritedTheme` 是：

```
bool updateShouldNotify(_InheritedTheme old) => theme.data != old.theme.data;
```



```

/// result of calling [State.setState] above the inherited widget.
@override
void notifyClients(InheritedWidget oldWidget) {
  if (!widget.updateShouldNotify(oldWidget))
    return;
  assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
  for (Element dependent in _dependents) {
    assert(() {
      // check that it really is our descendant
      Element ancestor = dependent._parent;
      while (ancestor != this && ancestor != null)
        ancestor = ancestor._parent;
      return ancestor == this;
    }());
    // check that it really depends on us
    assert(dependent._dependencies.contains(this));
    dependent.didChangeDependencies();
  }
}

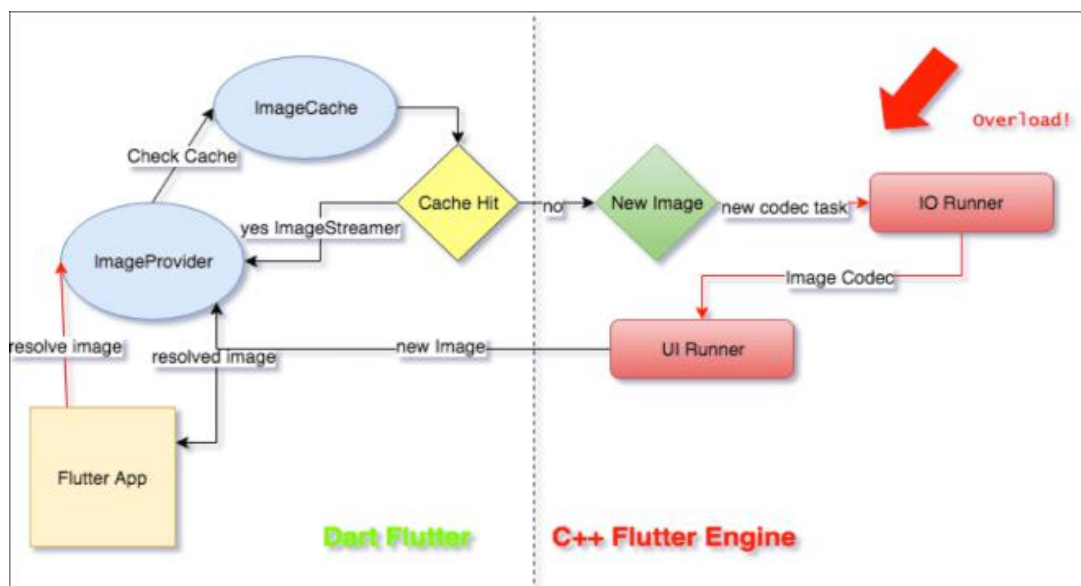
```

所以本质上 Theme、Redux 、 Scope Model、 Localizations 的核心都是 **InheritedWidget**。

三、内存

最近闲鱼技术发布了《Flutter 之禅 内存优化篇》，文中对于 Flutter 的内存做了深度的探索，其中有一个很有趣的发现是：

- Flutter 中 ImageCache 缓存的是 ImageStream 对象，也就是缓存的是一个异步加载的图片的对象。
- 在图片加载解码完成之前，无法知道到底将要消耗多少内存。
- 所以容易产生大量的 IO 操作，导致内存峰值过高。



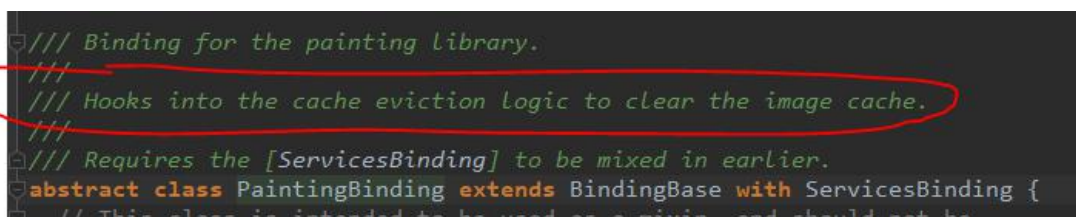
图片来自闲鱼技术

如上图所示，是图片缓存相关的流程，而目前的拮据处理是通过：

- 在页面不可见的时候没必要发出多余的图片
- 限制缓存图片的数量
- 在适当的时候 CG

更详细的内容可以阅读文章本体，这里为什么讲到这个呢？是因为 **限制缓存图片的数量** 这一项。

还记得 **WidgetsFlutterBinding** 这个胶水类吗？其中 Mixins 了 **PaintingBinding** 如下图所示，被“黏”上去的这个 **binding** 就是负责图片缓存



```
/// Binding for the painting library.
///
/// Hooks into the cache eviction logic to clear the image cache.
///
/// Requires the [ServicesBinding] to be mixed in earlier.
abstract class PaintingBinding extends BindingBase with ServicesBinding {
  // This class is intended to be used as a mixin, and should not be
```

在 **PaintingBinding** 内有一个 **ImageCache** 对象，该对象全局一个单例的，同时再图片加载时的 **ImageProvider** 所使用，所以设置图片缓存大小如下：

```
//缓存个数 100
PaintingBinding.instance.imageCache.maximumSize=100;//缓存大小 50m
PaintingBinding.instance.imageCache.maximumSizeBytes= 50 << 20;
```

四、线程

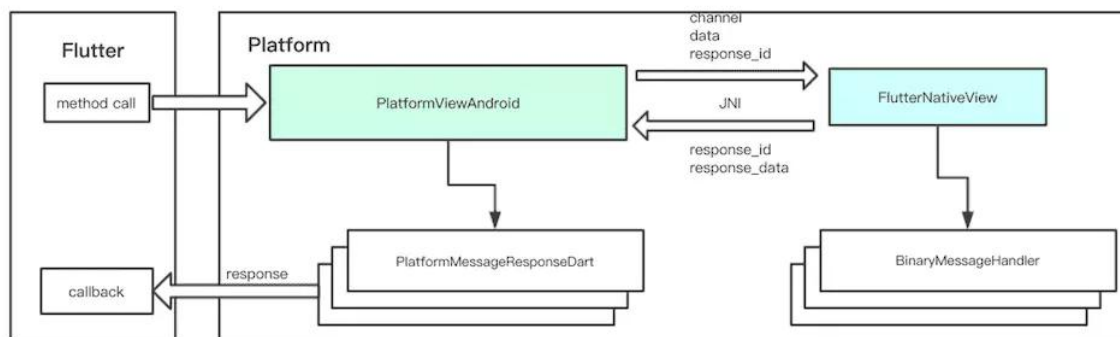
在闲鱼技术的 [深入理解 Flutter Platform Channel](#) 中有讲到：**Flutter** 中有四大线程，**Platform Task Runner**、**UI Task Runner**、**GPU Task Runner** 和 **IO Task Runner**。

其中 **Platform Task Runner** 也就是 **Android** 和 **iOS** 的主线程，而 **UI Task Runner** 就是 **Flutter** 的 **UI** 线程。

如下图，如果做过 **Flutter** 中 **Dart** 和原生端通信的应该知道，通过 **Platform Channel** 通信的两端就是 **Platform Task Runner** 和 **UI Task Runner**，这里主要总结起来是：

因为 **Platform Task Runner** 本来就是原生的主线程，所以尽量不要在 **Platform** 端执行耗时操作。

因为 Platform Channel 并非是线程安全的，所以消息处理结果回传到 Flutter 端时，需要确保回调函数是在 Platform Thread（也就是 Android 和 iOS 的主线程）中执行的。



图片来自闲鱼技术

五、热更新

逃不开的需求。

1、首先我们知道 Flutter 依然是一个 **iOS/Android** 工程。

2、Flutter 通过在 BuildPhase 中添加 shell（xcode_backend.sh）来生成和嵌入 **App.framework** 和 **Flutter.framework** 到 iOS。

3、Flutter 通过 Gradle 引用 **flutter.jar** 和把编译完成的二进制文件添加到 Android 中。

其中 Android 的编译后二进制文件存在于 `data/data/包名/app_flutter/flutter_assets/` 下。做过 Android 的应该知道，这个路径下是可以很简单更新的，所以你懂的 ω =。

IOS? 据我了解，貌似动态库 **framework** 等引用是不能用热更新的，除非你不需要审核！

自此，第五篇终于结束了！(///▽///)

资源推荐

- Github : <https://github.com/CarGuo>
 - 本文代码 : <https://github.com/CarGuo/GSYGithubAppFlutter>
- 完整开源项目推荐:

- [GSYGithubAppWeex](#)
- [GSYGithubApp React Native](#)

文章

《Flutter 完整开发实战详解(一、 Dart 语言和 Flutter 基础)》

《Flutter 完整开发实战详解(二、 快速开发实战篇)》

《Flutter 完整开发实战详解(三、 打包与填坑篇)》

《Flutter 完整开发实战详解(四、 Redux、主题、国际化)》

《Flutter 完整开发实战详解(五、 深入探索)》

《跨平台项目开源项目推荐》

《移动端跨平台开发的深度解析》