Table of Contents

前言 1.1
零、GSY历程 1.2
一、Dart语言和Flutter基础 1.3
二、 快速开发实战篇 1.4
三、 打包与填坑篇 1.5
四、 Redux、主题、国际化 1.6
五、 深入探索 1.7
六、 深入Widget原理 1.8
七、 深入布局原理 1.9
八、 实用技巧与填坑 1.10
九、 深入绘制原理 1.11
十、 深入图片加载流程 1.12
十一、全面深入理解Stream 1.13
十二、全面深入理解状态管理设计 1.14
十三、全面深入触摸和滑动原理 1.15
十四、混合开发打包 Android 篇 1.16
十五、全面理解State与Provider 1.17
十六、详解自定义布局实战 1.18
十七、实用技巧与填坑二 1.19
十八、 神奇的ScrollPhysics与Simulation 1.20
番外 1.21
番外 1.22
- Flutter 跨平台框架应用实战-2019极光开发者大会 1.22.1
- Flutter 面试知识点集锦 1.22.2
- 全网最全 Flutter 与 ReactNative深入对比分析 1.22.3
- Flutter 开发实战与前景展望 - RTC Dev Meetup 1.22.4

Flutter完整开发实战详解系列，GSY Flutter 系列专栏整合，不定期更新

在如今的 Fultter 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 和 独立多案例学习型项目 ，同时会提供一些Flutter的开发细节技巧，之后深入源码和实战为你全面解析 Flutter 。

如果克隆太慢，可尝试码云地址下载

- **在线阅读地址**
- **PDF 下载地址**
- **Github 地址 CarGuo**
- **掘金博客 恋猫de小郭**
- **开源 Flutter 多案例学习型项目**

目录

- 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
- 五、 深入探索
- 六、 深入Widget原理
- 七、 深入布局原理
- 八、 实用技巧与填坑
- 九、 深入绘制原理

- 十、 深入图片加载流程
- 十一、全面深入理解Stream
- 十二、全面深入理解状态管理设计
- 十三、全面深入触摸和滑动原理
- 十四、混合开发打包 Android 篇
- 十五、全面理解State与Provider
- 十六、详解自定义布局实战
- 十七、实用技巧与填坑二
- 十八、 神奇的ScrollPhysics与Simulation
- 番外
  - Flutter 跨平台框架应用实战-2019极光开发者大会
  - 全网最全 Flutter 与 ReactNative深入对比分析
  - Flutter 面试知识点集锦
  - Flutter 开发实战与前景展望 - RTC Dev Meetup

如果对您有所帮助，欢迎投喂：

让 GSY 成为你 Flutter 学习路上的“保姆”吧。

> - Flutter 完整开发实战详解系列文章
> - 开源 Flutter 多案例学习型项目
> - 开源 Flutter 完整实战项目
> - 开源 Flutter 电子书项目

自 2018 年 06 月以来， `Flutter` 开始在 **GSY** 系列中初绽锋芒，在经历一年的发展之后，目前 **GSY** **`Flutter`** 系列已包含有 **《Flutter完整开发实战详解》 系列文章 、 多案例学习型项目 GSYFlutterDemo 、完整实战项目 GSYGithubAppFlutter 、 Flutter 电子书项目 GSYFlutterBook** 等，目前改系列项目的 **star** 情况如下所示：

项目	Star
GSYGithubAppFlutter	stars 9.3k
GSYFlutterBook	stars 2.1k
GSYFlutterDemo	

一、Flutter完整开发实战详解

《Flutter完整开发实战详解》 系列文章，更新至今已有 **主系列文章 15 篇，番外系列文章 3 篇** ，内容主要覆盖 **开发实战、源码分析、填坑技巧、面试集锦** 等等，并且该系列目前仍处于更新阶段。

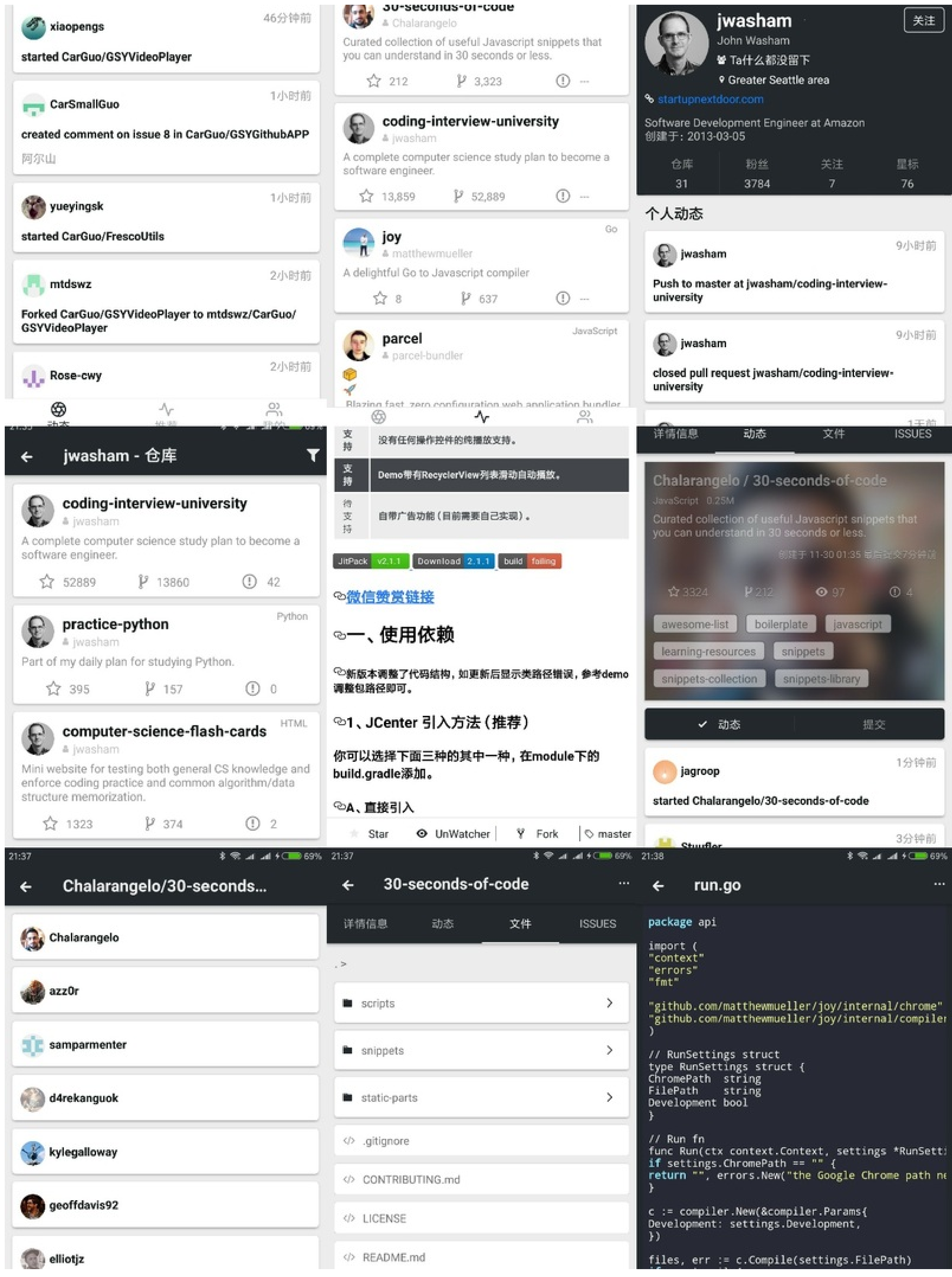
通过本系列文章，你将快速了解到 `Flutter` 中的各种特性和实战技巧，掌握 `Flutter Framework` 的工作原理，从入门到出家应有尽有。

同时为了更方便学习， **《Flutter完整开发实战详解》** 系列文章会同步整合到 **GSYFlutterBook** 项目中，项目将通过**在线 Gitbook 和离线 PDF 方式， 进一步满足你的学习要求。**

二、GSYGithubAppFlutter

GSYGithubAppFlutter 项目属于 Flutter 完整实战项目，项目从 **状态管理、控件展示、数据请求保存、平台交互、动画效果等，完整展示了如何实现一个 Flutter 的应用项目，同时针对一些特殊场景进行填坑，并混入了多种开发和设计模式**， 项目最终的目的，是希望可以成为你实战过程中的引路者。

> GSYGithubApp 系列项目起源于 `React Native` ，目前共有四个版本。
>
> | 时间 | 项目 |
> | --- | --- |
> | 2017-11-07 | GSYGithubApp React Native 版开源 |
> | 2018-04-22 | GSYGithubApp Weex 版开源 |
> | 2018-06-26 | GSYGithubApp Flutter 版开源 |
> | 2018-11-08 | GSYGithubApp Kotlin 版开源 |

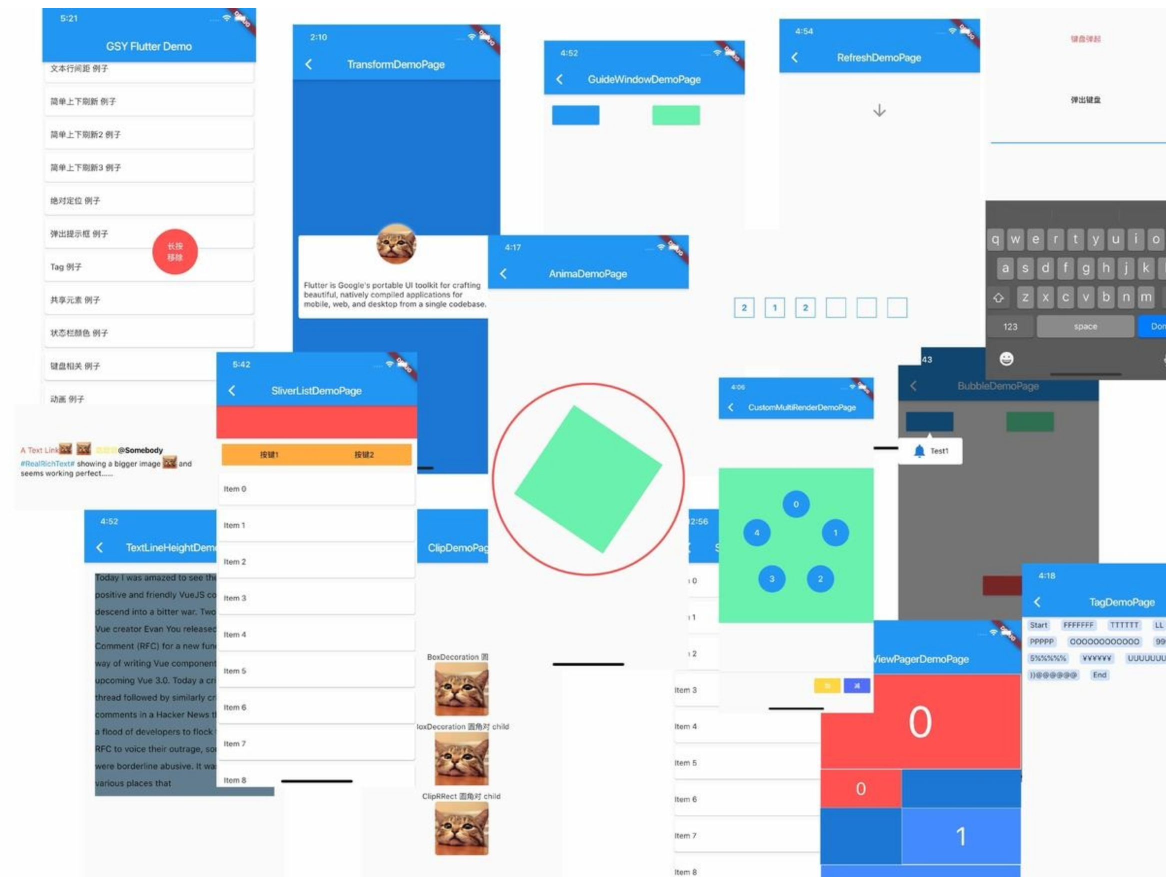


三、GSYFlutterDemo

GSYFlutterDemo **是本月刚创建的学习型项目**， 因为 GSYGithubAppFlutter 属于完整型项目，不适合频繁调整和 Demo 示例，所以在接受到用户反馈后， **更轻便的 GSYFlutterDemo 诞生了。**

GSYFlutterDemo 作为简单示例和解决方案 Demo ，它可以给你学习和工作中提供一些便捷的帮助，比如 **如何自定义布局，如何滚动控件到指定 `child position` ，如何调整 `Text` 控件的 `Line Space` ，如何监听键盘的弹出和收起** 等等，所以例子方案都独立实现，方便阅读 CV 。

> 其中一些需求因为 `Flutter` 特性限制，需要特殊处理才能实现。



最后

GSY `Flutter` 系列断断续续一路走来，有着太多的机缘巧合在推动前进，个人是希望 **GSY** 能成为你 `Flutter` 学习路上的“保姆”，最终能产生交流互动，共同成长。

未来 **《Flutter完整开发实战详解》** 系列文件将继续更新，同时逐步完善 **GSYFlutterDemo** 中的各种案例，并同步优化 **GSYGithubAppFlutter** 中的各种问题，**你的认可就是我坚持的动力！**

> 学习并非一朝一夕，我相信在分享过程中的“碰撞”，能让我们更快的进步，因为码农并不孤单！

其他推荐

- Flutter 状态管理示例
- Flutter 混合开发示例
- GSYGithubAPP React Native
- GSYGithubApp Kotlin

- GSYVideoPlayer Android 播放器

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：如何入门 Flutter 开发，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，提供 Flutter 的开发技巧和问题处理，之后深入源码和实战为你全面解析 Flutter。

> 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，是多么的不可思议画面，并且 Fluuter 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！

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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 Stuido 上安装插件，并到 GitHub Clone Flutter 项目到本地之后执行 flutter doctor 命令就可以完成配置，其实中文网的搭建 Futter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，国内的用户有时候需要配置 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* 也是蛮有意思的感觉）

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

官方也提供了包括 iOS 、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。

2.1、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 也算半个动态类型语言，同时支持闭包。
- `Dart` 属于是**强类型语言**，但可以用 `var` 来声明变量， `Dart` 会**自推导出数据类型**，所以 `var` 实际上是编译期的“语法糖”。 **`dynamic` 表示动态类型**， 被编译后，实际是一个 `object` 类型，在编译期间不进行任何的类型检查，而是在运行期进行类型检查。
- 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。

2.2、变量

- Dart 不需要给变量设置 `setter getter` 方法， 这和 kotlin 等语言类似。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 的值要到运行时才确定。
- 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。

2.3、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(Sting 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);
}
```

2.4、Flutter

Flutter 中支持 `async` / `await` ，**如下代码所示，** `async` / `await` 其实只是语法糖，最终会编译为 Flutter 中返回 `Future` 对象，之后通过 `then` 可以执行下一步。如果返回的还是 `Future` 便可以 `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。
- Stream 对应的 async* / yield 也可以用于异步，这个后面会说到。

3、Flutter Widget

在 Flutter 中一切的显示都是 Widget ， Widget 是一切的基础，利用响应式模式进行渲染。

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，其实是创建了新的 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，如下下代码所示是无状态 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 it easy** ，后面我们就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` Column 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，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DMEO"),
    );
  }
}
```

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： 最常用的默认控件，但是实际上它是由多个内置控件组成的模版，只能包含一个 `child` ，支持 *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maring
       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 中代表着平均充满的作用，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
          mainAxisSize : MainAxisSize.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

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, bot
tom: 10.0),
                child: new Column(
                  mainAxisSize: MainAxisSize.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"),
                      ],
                    ),
```

```
                    ],
                  ),
                ))),
        );
    }
```

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 Wdiget，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 。
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 `TextSpan` ，可以拼接出富文本场景。
TextField	文本输入框 ： `new TextField(controller: //文本控制器, obscureText: "hint文本");`
Image	图片加载: `new FadeInImage.assetNetwork( placeholder: "预览图", fit: BoxFit.fitWidth, image: "url");`
FlatButton	按键点击: `new FlatButton(onPressed: () {},child: new Container());`

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，会自带返回按键
    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());
```

```
  }
}
```

14:54
100%
DEBUG
Title
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000
这是一点描述
1000
1000
1000

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，后续更多实战等你开启

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- 本文相关 ： GSYGithubAppFlutter
- GSYGithubAppWeex
- GSYGithubApp React Native

作为系列文章的第二篇，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。**

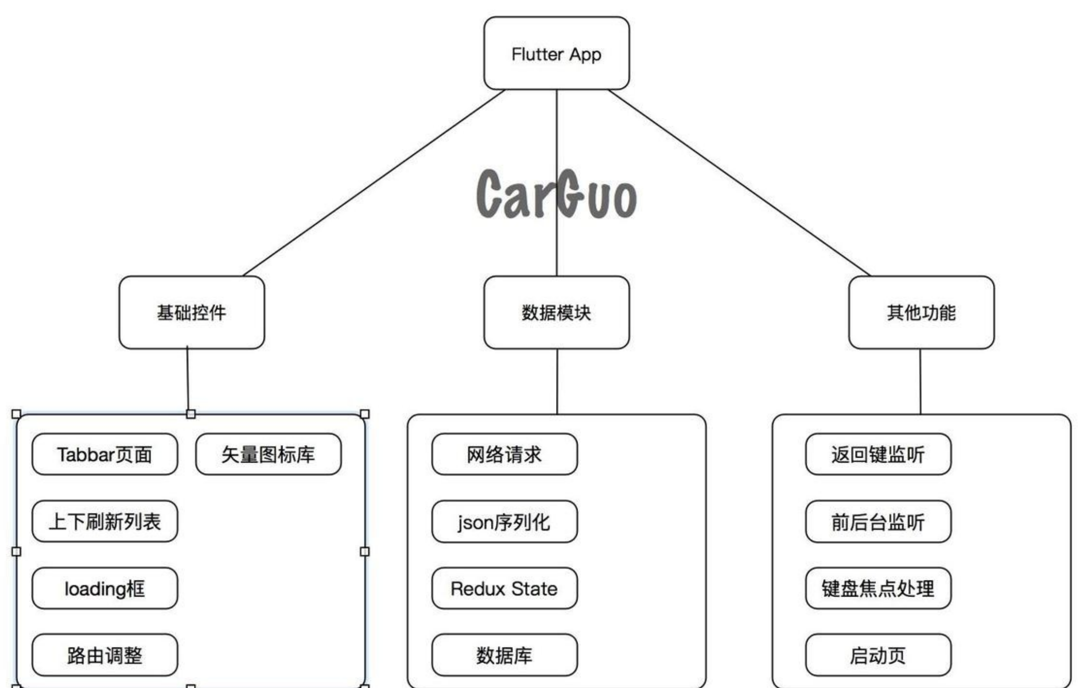
> 友情提示：本文所有代码均在 **GSYGithubAppFlutter** ，文中示例代码均可在其中找到，看完本篇相信你应该可以轻松完成如下效果。相关基础还请看篇章一。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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 的愉悦！** 那么就让我们愉悦的往下开始吧！



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，早期诸如 #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。（1.9.1 stable 中已经修复）

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。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,
        ///设置悬浮按键，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。

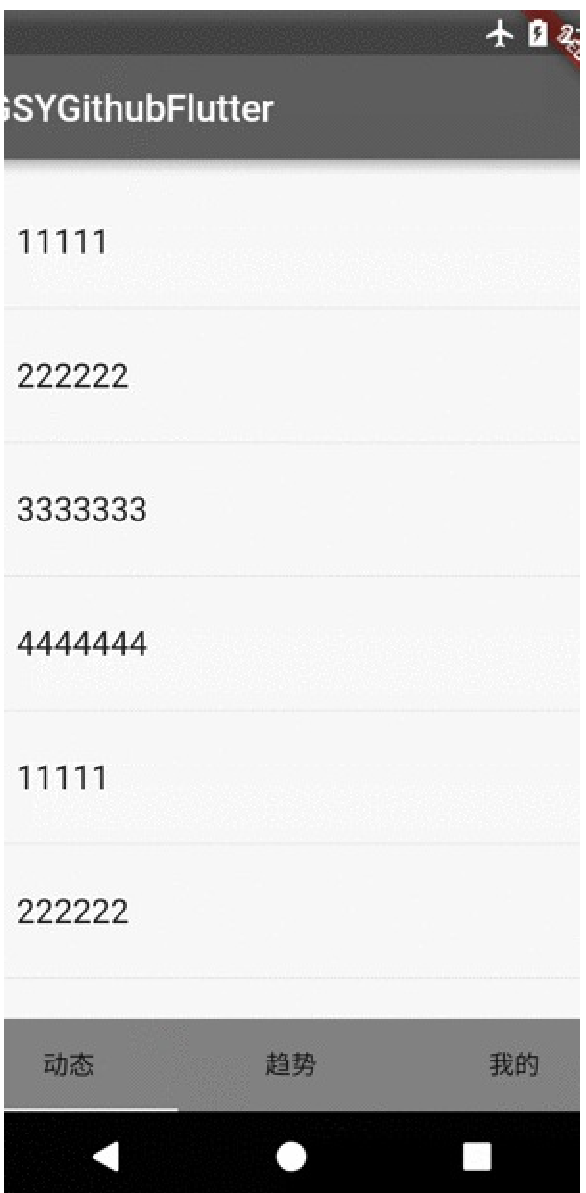
外部调用代码如下：每个 Tabbar 点击时，通过 `pageController.jumpTo` 跳转页面，每个页面需要跳转坐标为：**当前屏幕大小乘以索引 index** 。

```
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。



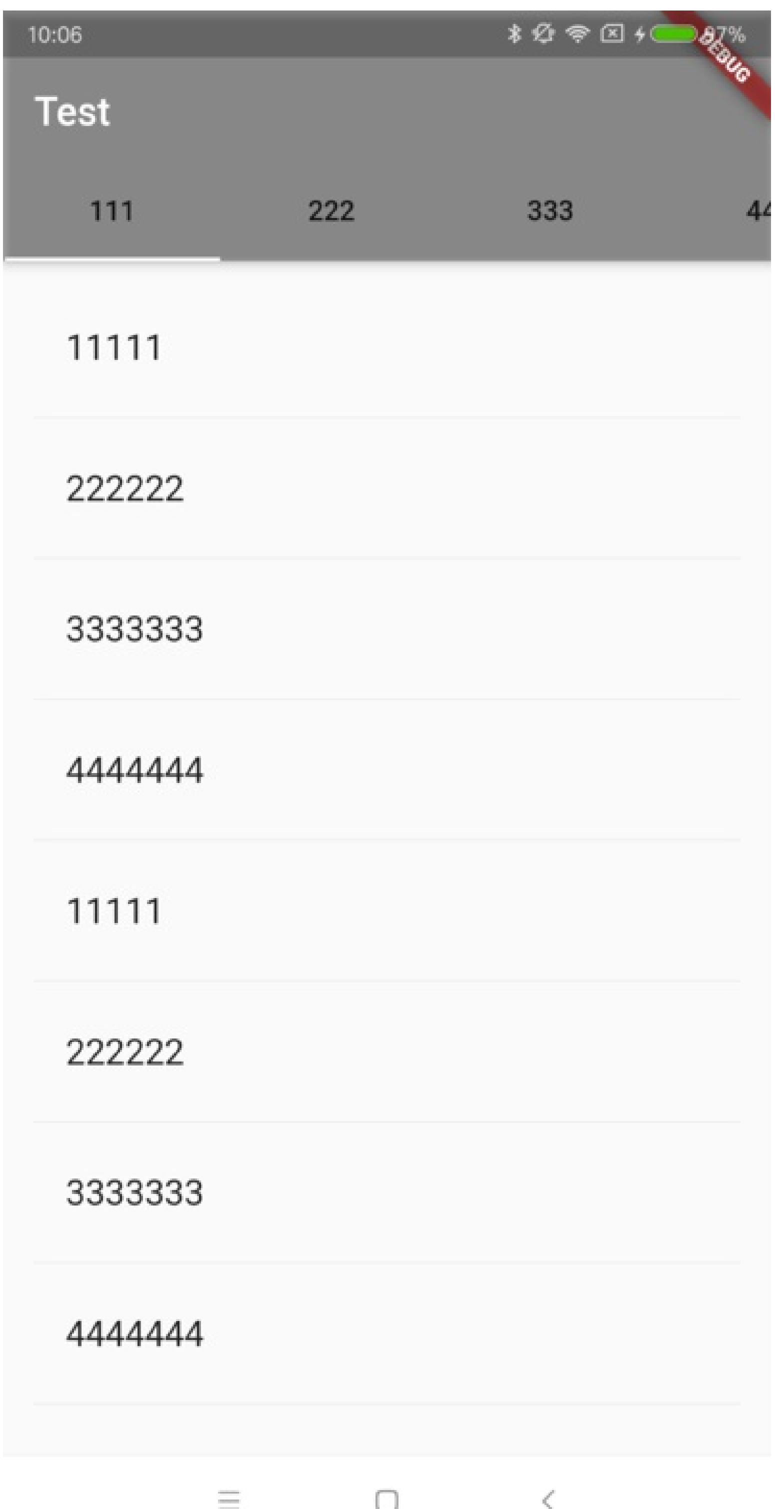
既然底部Tab页面都实现了，干脆顶部tab页面也一起完成。如下代码，和底部Tab页的区别在于：

- 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,
        ///设置悬浮按键，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);
        },
      ),
    );
```



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` 实例需要全局唯一。

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`** 。

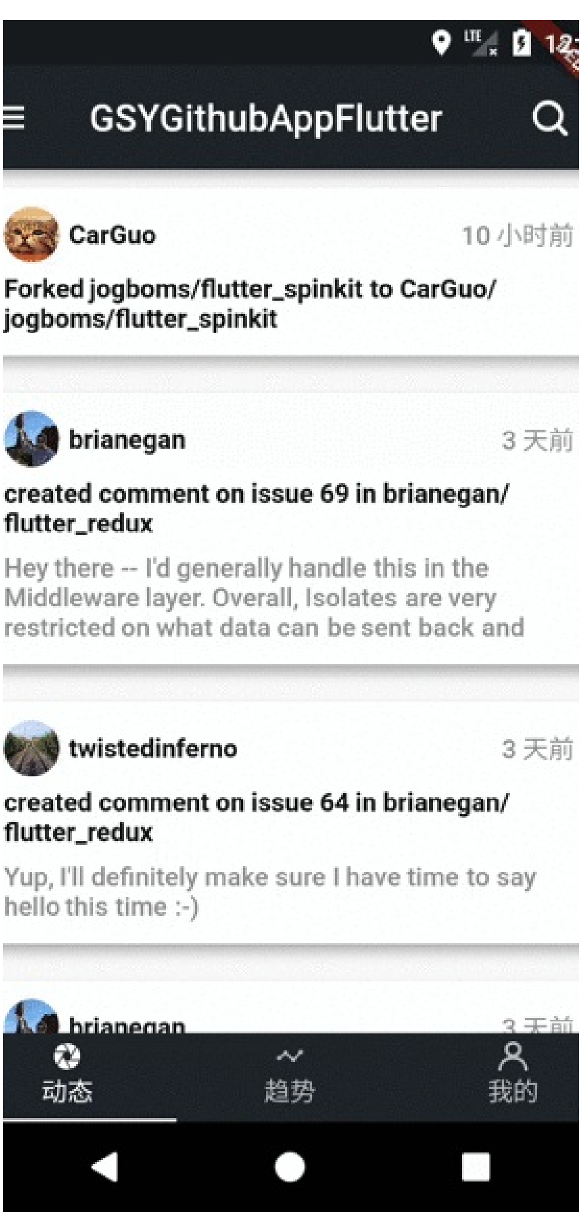
上拉加载更多在代码中是通过 `_getListCount()` 方法，在原本的数据基础上，增加实际需要渲染的 item 数量给 ListView 实现的，最后**通过 `ScrollController` 监听到底部，触发 `onLoadMore`** 。

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 _GSYPullLoadWidgetState extends State<GSYPullLoadWidget> {
  ///···
  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.maxScroll
Extent) {
        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.dataLi
st.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ind
ex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() {
       ///···
    }
  }
```

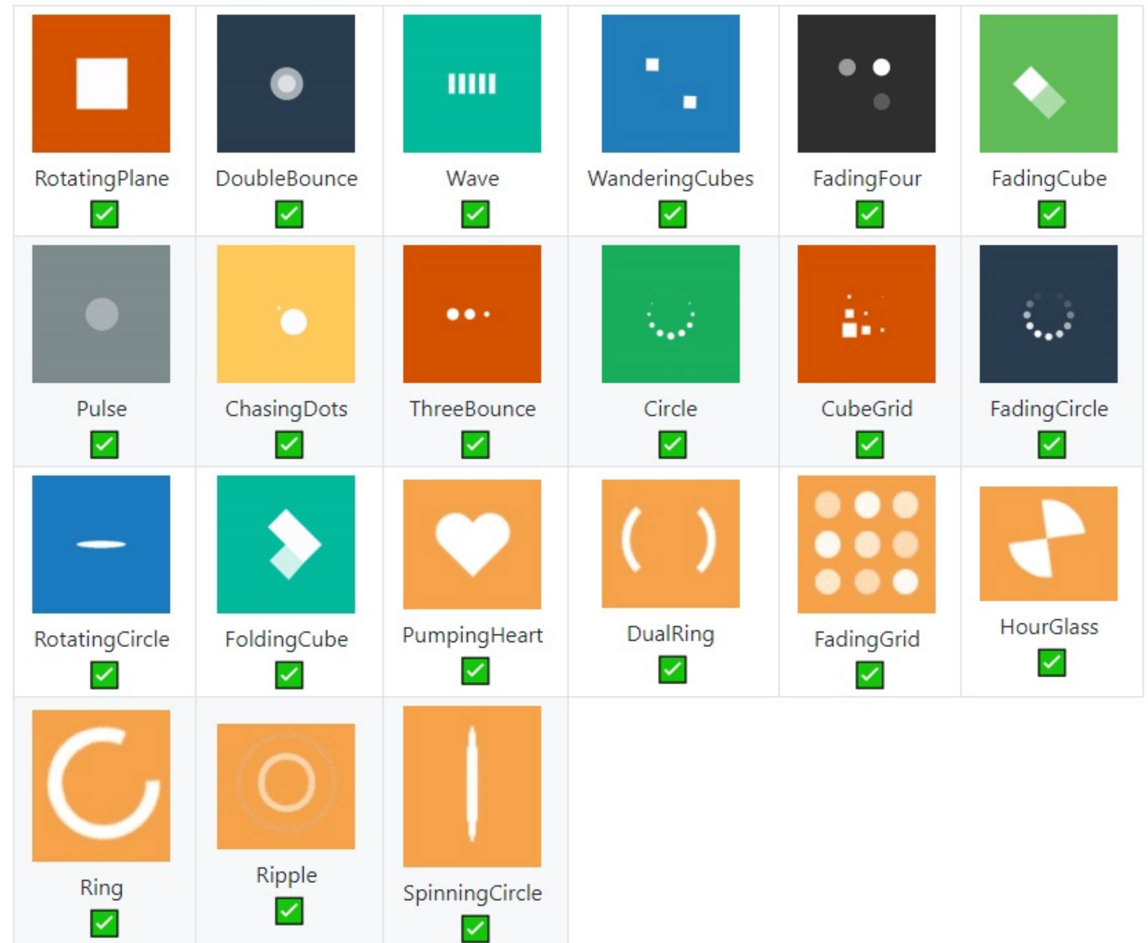


3、Loading框

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,
     ),
   );
 }
```



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，同时个人推荐阿里爸爸的 **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：

> 新版本开始可以给 `pushNamed` 设置 `arguments` 参数，然后在新页面通过 `ModalRoute.of(context).settings.arguments` 获取。

```
///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。

```
///创建网络请求对象，主要最好吧 dio 实例全局单里
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);
}
```

```dart
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';
```

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**。

```dart
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};
}
```

注意：新版*json*序列化中做了部分修改，代码更简单了，详见 *demo* 。

3、Redux

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，后续会有详细介绍，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 GSYState {
  ///用户信息
  User userInfo;
  ///构造方法
  GSYState({this.userInfo});

}

///通过 Reducer 创建 用于store 的 Reducer
GSYState appReducer(GSYState state, action) {
  return GSYState(
    ///通过 UserReducer 将 GSYState 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 GSYState 中的 User
    return new StoreConnector<GSYState, User>(
      ///通过 converter 将 GSYState 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), c
hild:  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(tru
e);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！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubAppWeex
- GSYGithubApp React Native

作为系列文章的第三篇，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。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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** 。

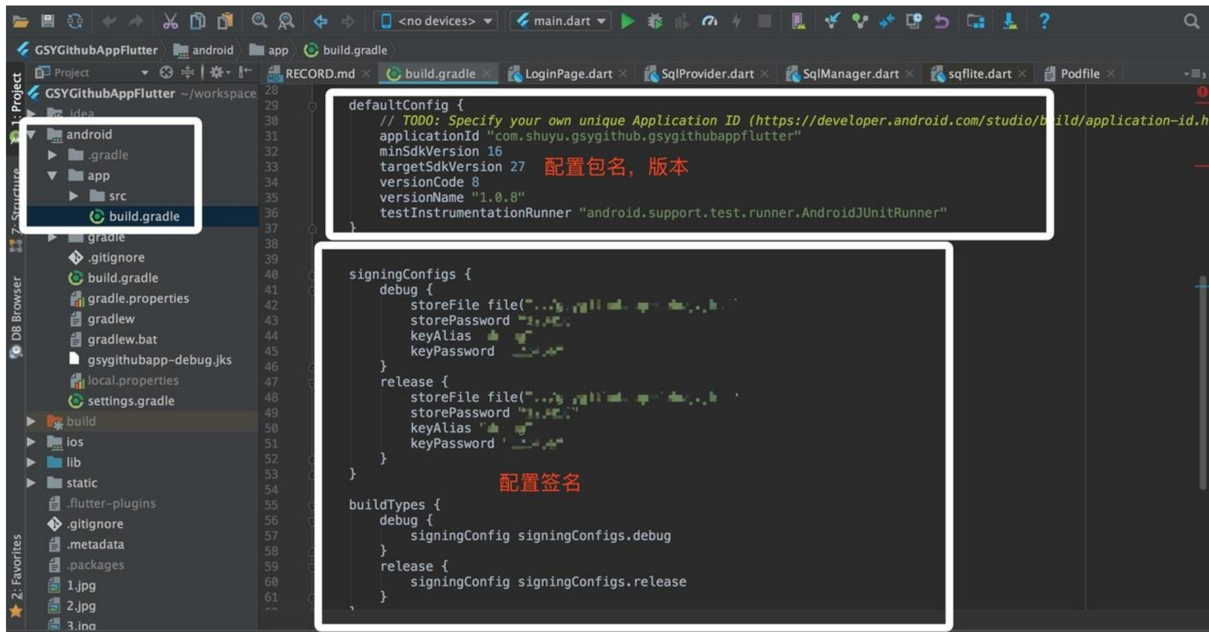
项目	IOS	Android
GSYGithubAppFlutter	IPA Runner-iPhone 6s Plus.ipa iOS 应用 – 14.3 MB	GSYGithubAppFlutter-1.0.8.apk 文稿 – 11.2 MB
GSYGithubAppRN	IPA GSYGithubAPP-iPhone 6s Plus.ipa iOS 应用 – 4.2 MB	GSYGithubApp-1.9.apk 文稿 – 17.4 MB

从上表我们可以看到：

- Fluuter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



在 Android 的打包上，笔者基本没有遇到什么问题，在 `android/app/build.grade` 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。

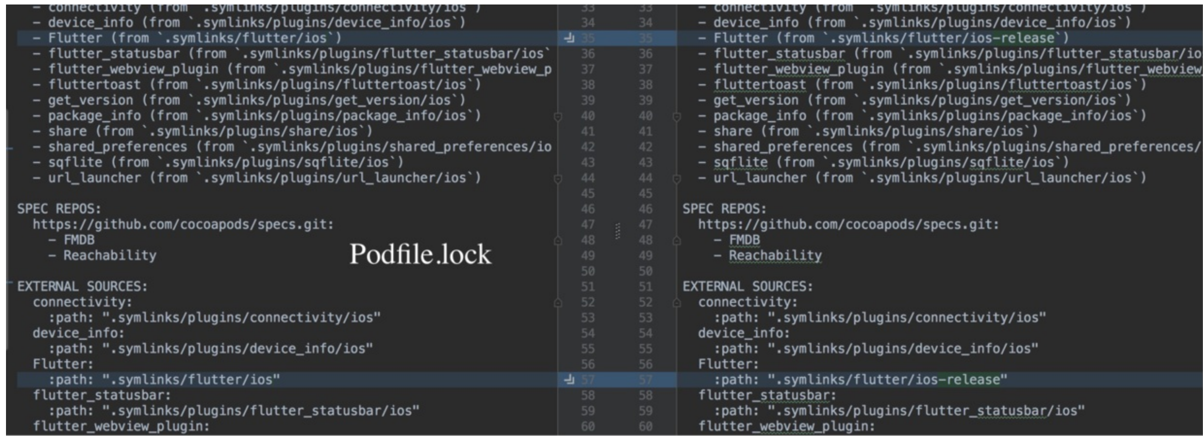
首先你需要一个 apple 开发者账号，然后创建证书、创建AppId，创建配置文件、最后在 `info.plist` 文件下输入相关信息，更详细可看官方的《发布的IOS版APP》的教程。

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）



但是实际在执行 `flutter build release` 后，问题依然存在，最终翻山越岭(╯`□′)╯︵┻━┻，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。

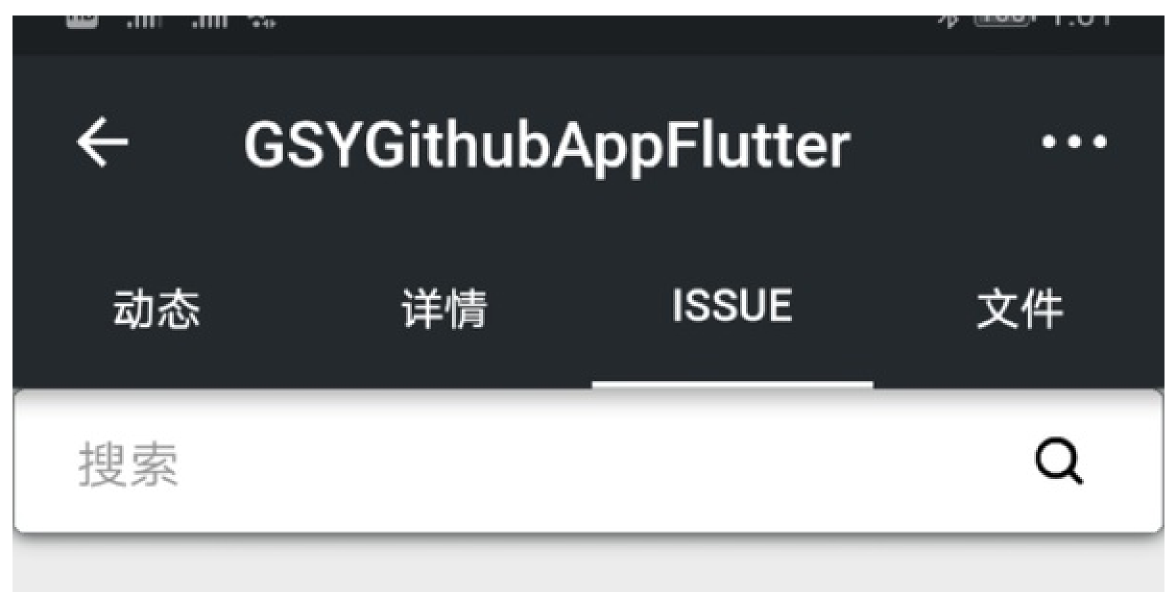
二、细节

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，一个可控的填充按键就出来了。

```
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(
       ///·····
     ),
   );
 }
```

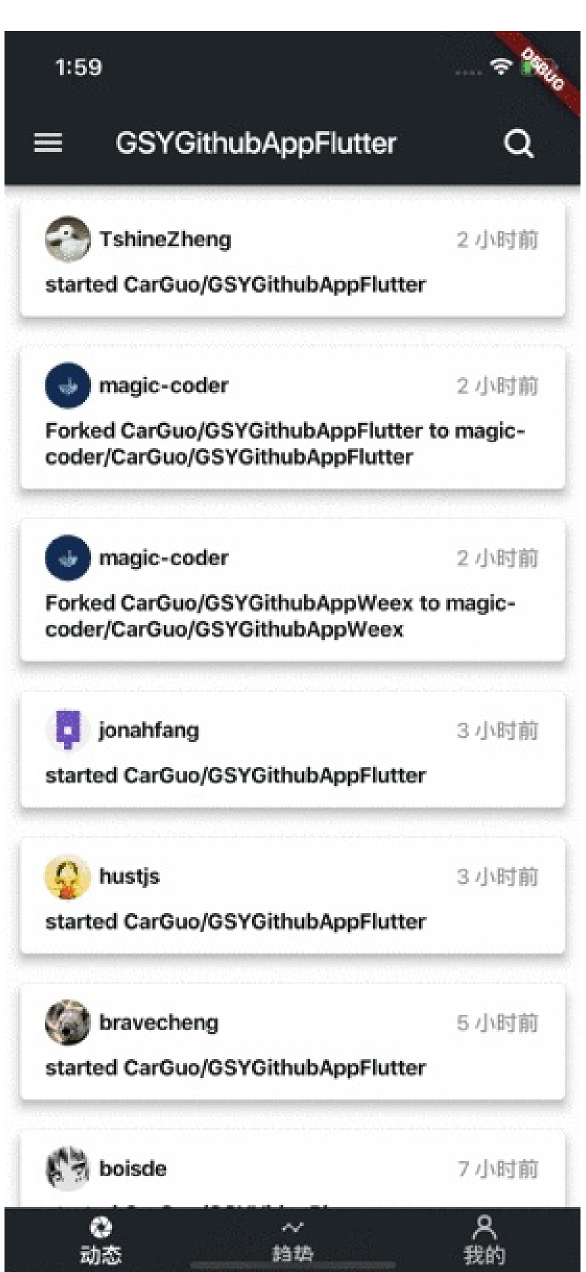
5、Redux 与主题

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();
                },
              });
        }),
      );
    }
  }
```



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！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubAppWeex
- GSYGithubApp React Native

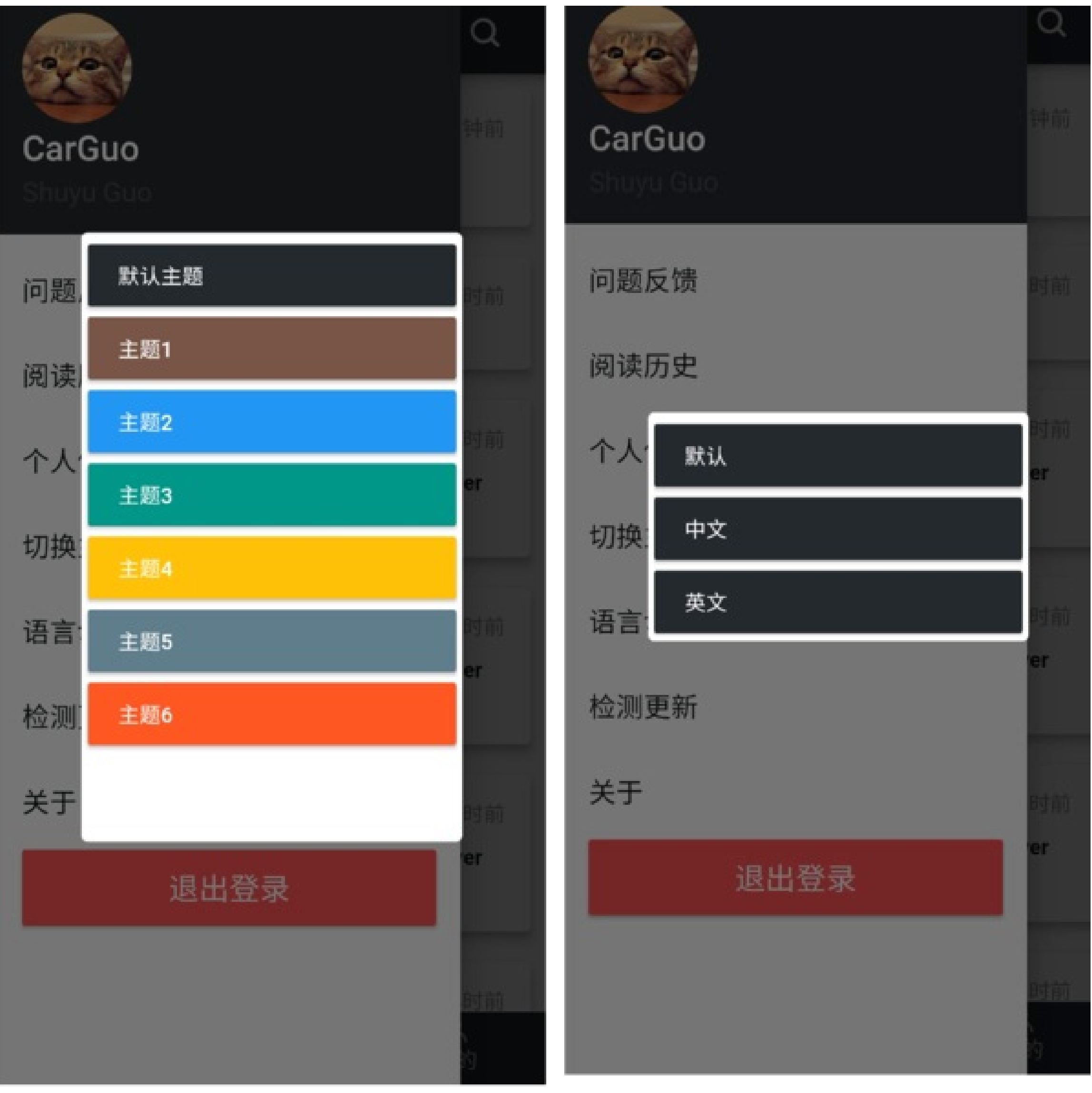
作为系列文章的第四篇，本篇主要介绍 Flutter 中 Redux 的使用，并结合Redux 完成实时的**主题切换**与**多语言切换**功能。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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 。

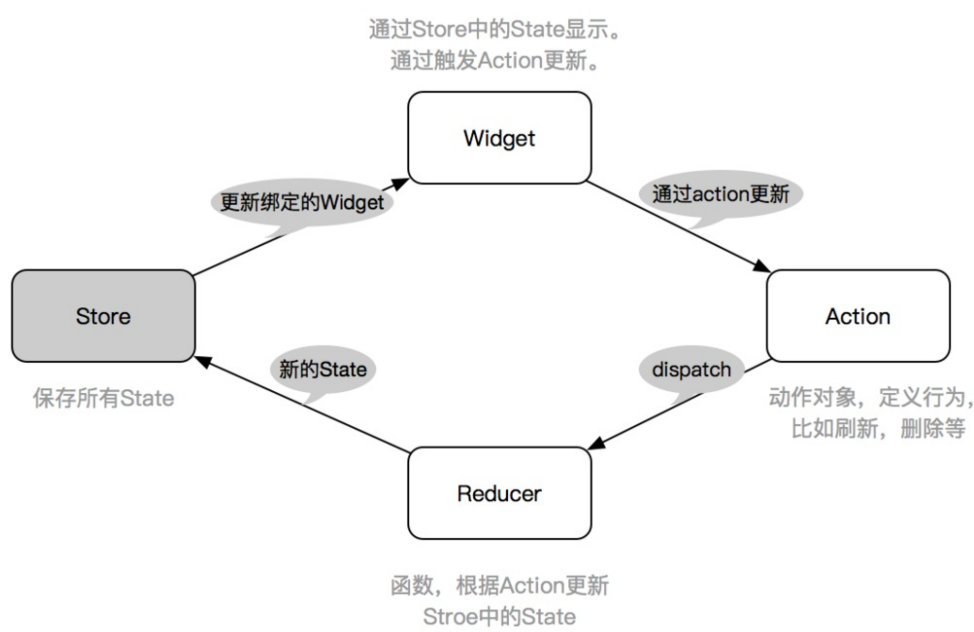


一、Redux

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。



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 GSYState 内的 themeData 和 action 关联在一起
    themeData: ThemeDataReducer(state.themeData, action),

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

如上代码，**GSYState** 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 GSYState 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 GSYState 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，现在我们可以愉悦的创建 **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 GSYState 中的 User
    return new StoreConnector<GSYState, User>(
      ///通过 converter 将 GSYState 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。

```
return [
  GSYColors.primarySwatch,
  Colors.brown,
  Colors.blue,
  Colors.teal,
  Colors.amber,
  Colors.blueGrey,
  Colors.deepOrange,
];
```

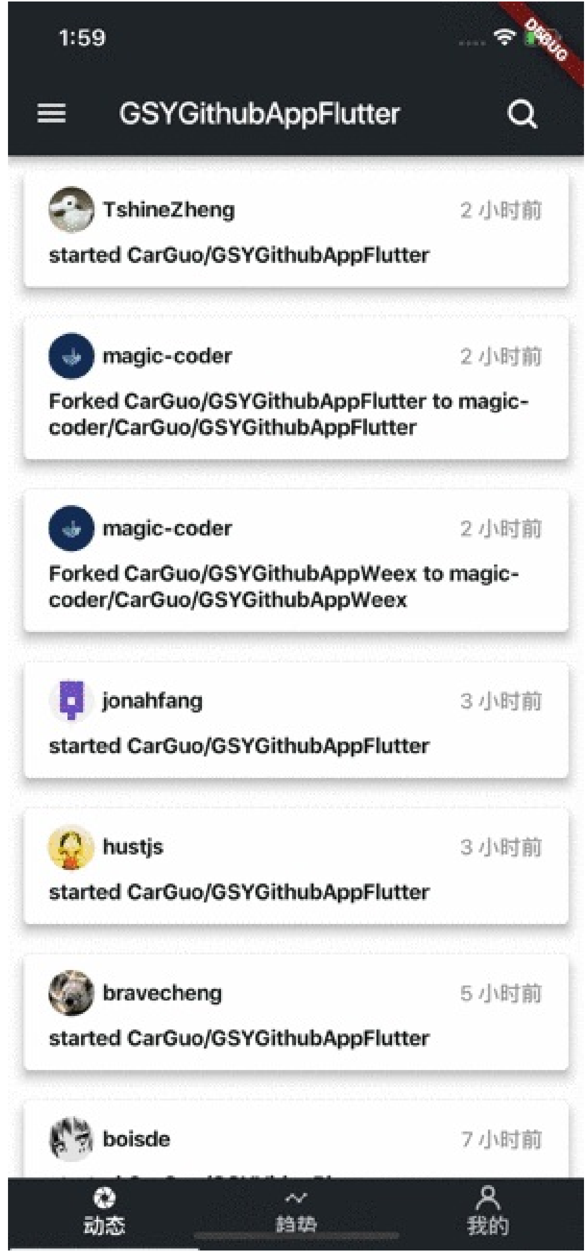
```
MaterialColor primarySwatch = const MaterialColor(
    primaryValue,
    const <int, Color>{
      50: const Color(primaryLightValue),
      100: const Color(primaryLightValue),
      200: const Color(primaryLightValue),
      300: const Color(primaryLightValue),
      400: const Color(primaryLightValue),
      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。

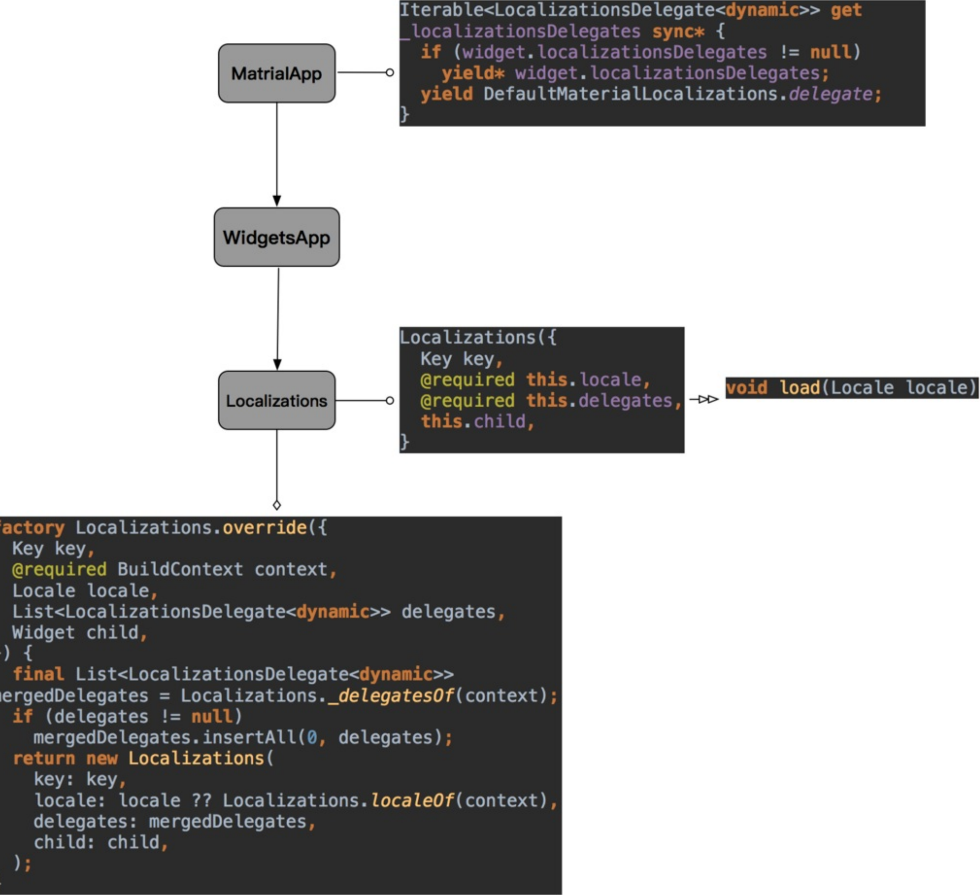
注意，因为你的 **MaterialApp** 也是一个 `StatefulWidget` ，如下代码所示，还需要利用 `StoreBuilder` 包裹起来，之后我们就可以通过 `dispatch` 修改主题，通过 `Theme.of(context).primaryColor` 获取主题色啦。

```
 @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));
```



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！



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 GSYStringZh(),
  };

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

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

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

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

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

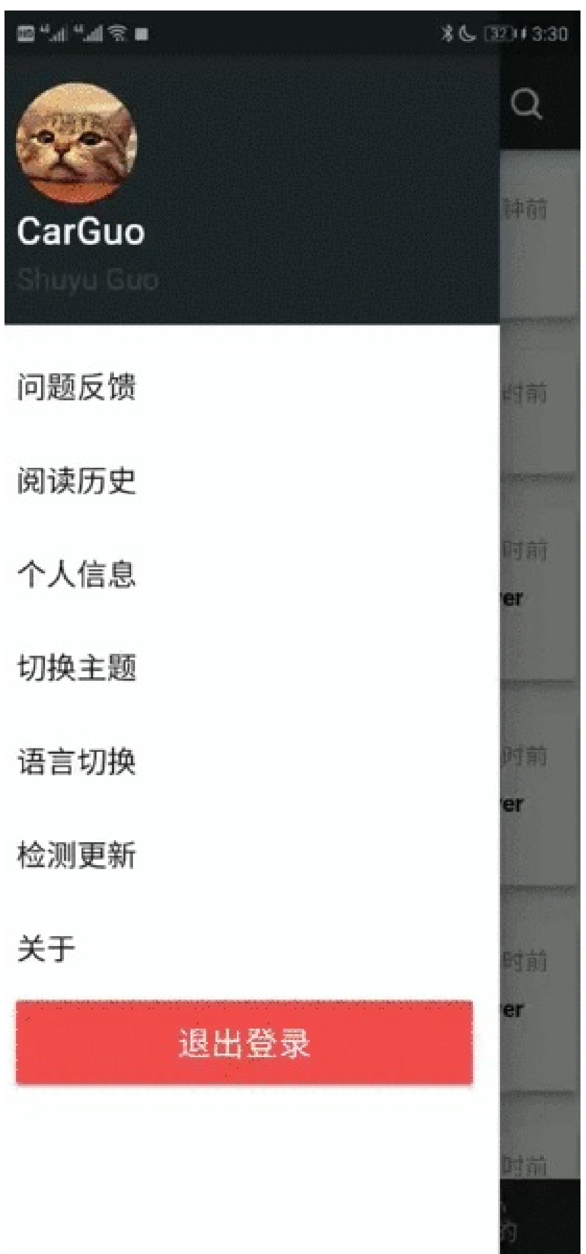
说完了 delegate ， 接下来就是 `Localizations` 了。在上面的流程图中可以看到， Localizations 提供一个 `override` 方法构建 `Localizations` ，这个方法中可以设置 locale，而我们需要的正是**实时的动态切换语言显示**。

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！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubAppWeex
- GSYGithubApp React Native

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。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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。

```dart
/// 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;
  }
}
```

```dart
class WidgetsBinding extends BindingBase with SchedulerBinding, GestureBinding, RendererBinding {
s class is intended to be used as a mixin, and should not be
```

```dart
glue between the render tree and the Flutter engine.
class RendererBinding extends BindingBase with ServicesBinding, SchedulerBinding, SemanticsBinding, HitTestable
```

```dart
// TODO(jonahwilliams): move the remaining semantic related bindings here.
class SemanticsBinding extends BindingBase with ServicesBinding {
```

```dart
ires the [ServicesBinding] to be mixed in earlier.
class PaintingBinding extends BindingBase with ServicesBinding {
```

```dart
hedulingStrategy].
class SchedulerBinding extends BindingBase with ServicesBinding {
```

```dart
ct]).
class ServicesBinding extends BindingBase {
```

```dart
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` 。

看出来了没，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，不出所料ヽ(￣▽￣)ﾉ。

```
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(ｷ｀ﾟДﾟ´)!!?

```
  ///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;
```

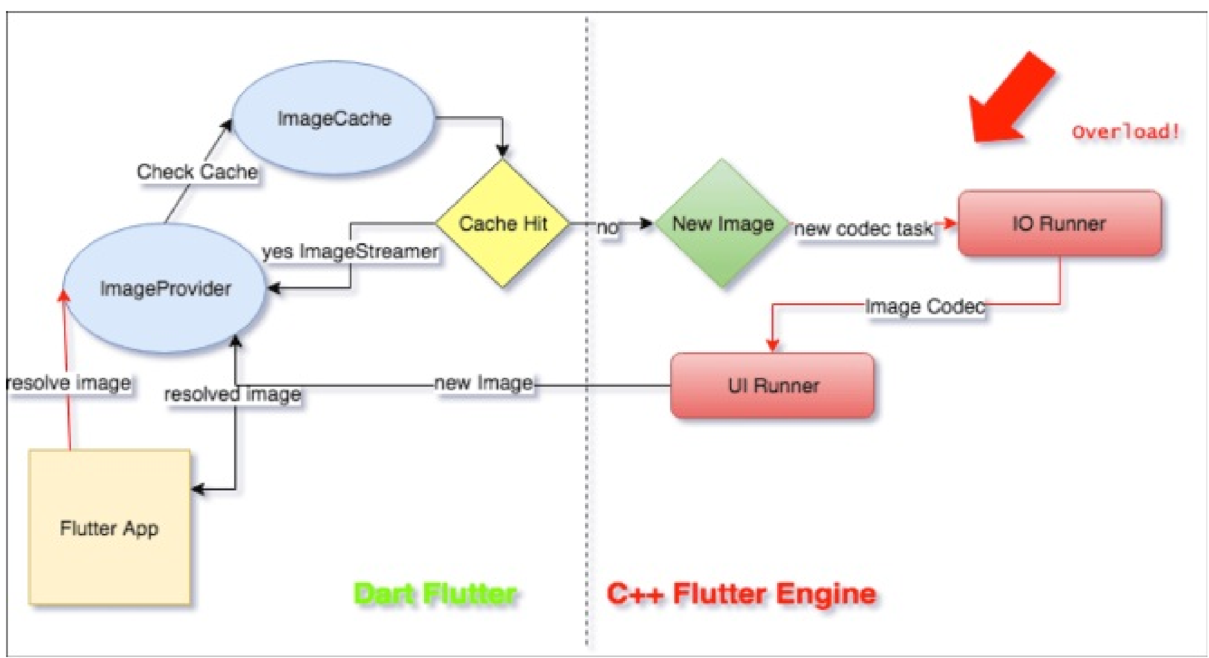
```
/// result of calling [State.setState] above the inherited widget.
@override
void notifyClients(InheritedWidget oldWidget) {
  if (!widget.updateShouldNotify(oldWidget))
    return;
  assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
  for (Element dependent in _dependents) {
    assert(() {
      // 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。



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 {
```

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;
```

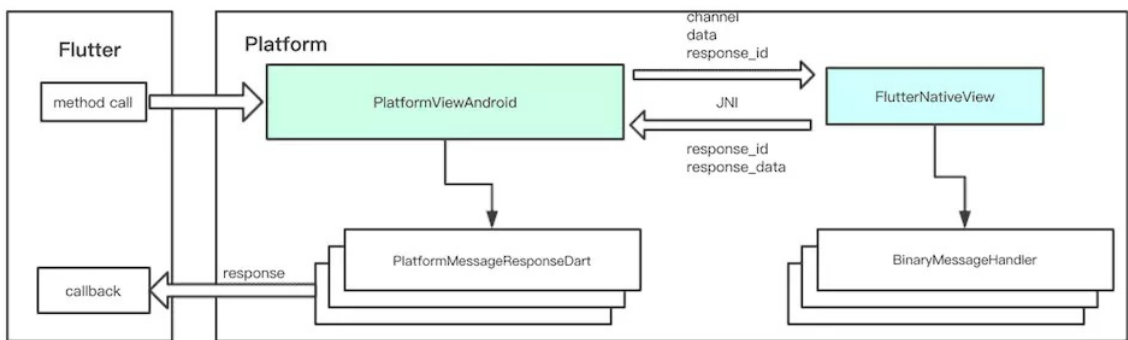
四、线程

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。



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￣=。

> **⚠注意，1.7.8 之后的版本，Android 下的 Flutter 已经编译为纯 so 文件。**

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！

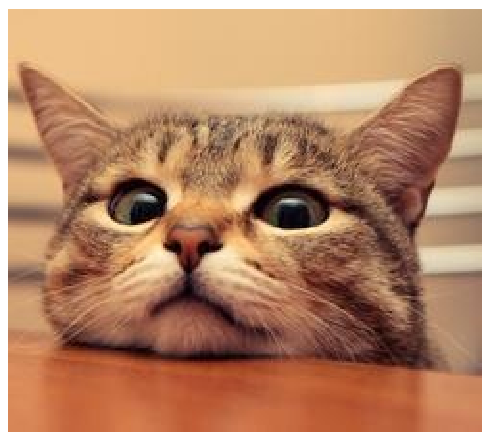
> 自此，第五篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubAppWeex
- GSYGithubApp React Native



作为系列文章的第六篇，本篇主要在前文的探索下，针对描述一下 Widget 中的一些有意思的原理。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

首先我们需要明白，Widget 是什么？这里有一个 “总所周知” 的答案是：**Widget并不真正的渲染对象** 。是的，事实上在 Flutter 中渲染是经历了从 `Widget` 到 `Element` 再到 `RenderObject` 的过程。

我们都知道 Widget 是不可变的，那么 Widget 是如何在不可变中去构建画面的？上面我们知道，`Widget` 是需要转化为 `Element` 去渲染的，而从下图注释可以看到，事实上 **Widget 只是 Element 的一个配置描述** ，告诉 Element 这个实例如何去渲染。

```
@immutable
abstract class Widget extends DiagnosticableTree {

  /// Inflates this configuration to a concrete instance.
  ///
  /// A given widget can be included in the tree zero or more times. In particular
  /// a given widget can be placed in the tree multiple times. Each time a widget
  /// is placed in the tree, it is inflated into an [Element], which means a
  /// widget that is incorporated into the tree multiple times will be inflated
  /// multiple times.
  @protected
  Elemcent createElement();
```

那么 Widget 和 Element 之间是怎样的对应关系呢？从上图注释也可知： **Widget 和 Element 之间是一对多的关系** 。实际上渲染树是由 Element 实例的节点构成的树，而作为配置文件的 Widget 可能被复用到树的多个部分，对应产生多个 Element 对象。

那么 `RenderObject` 又是什么？它和上述两个的关系是什么？从源码注释写着 `An object in the render tree` 可以看出到 `RenderObject` 才是实际的渲染对象，而通过 Element 源码我们可以看出：**Element 持有 RenderObject 和 Widget。**

```
/// The configuration for this element.
@override
Widget get widget => _widget;
Widget _widget;


/// The render object at (or below) this location in the tree.
///
/// If this object is a [RenderObjectElement], the render object is the one at
/// this location in the tree. Otherwise, this getter will walk down the tree
/// until it finds a [RenderObjectElement].
RenderObject get renderObject {
  RenderObject result;
  void visit(Element element) {
    assert(result == null); // this verifies that there's only one child
    if (element is RenderObjectElement)
      result = element.renderObject;
    else
      element.visitChildren(visit);
  }
  visit(this);
  return result;
}
```

再结合下图，可以大致总结出三者的关系是：**配置文件 Widget 生成了 Element，而后创建 RenderObject 关联到 Element 的内部 `renderObject` 对象上，最后Flutter 通过 RenderObject 数据来布局和绘制。** 理论上你也可以认为 RenderObject 是最终给 Flutter 的渲染数据，它保存了大小和位置等信息，Flutter 通过它去绘制出画面。

```
/// Creates an instance of the [RenderObject] class that this
/// [RenderObjectWidget] represents, using the configuration described by this
/// [RenderObjectWidget].
///
/// This method should not do anything with the children of the render object.
/// That should instead be handled by the method that overrides
/// [RenderObjectElement.mount] in the object rendered by this object's
/// [createElement] method. See, for example,
/// [SingleChildRenderObjectElement.mount].
@protected
RenderObject createRenderObject(BuildContext context);
```

说到 `RenderObject` ，就不得不说 **`RenderBox`** ： `A render object in a 2D Cartesian coordinate system` ，从源码注释可以看出，它是在继承 `RenderObject` 基础的布局和绘制功能上，实现了“笛卡尔坐标系”：以 Top、Left 为基点，通过宽高两个轴实现布局和嵌套的。

RenderBox 避免了直接使用 `RenderObject` 的麻烦场景，其中 `RenderBox` 的布局和计算大小是在 `performLayout()` 和 `performResize()` 这两个方法中去处理，很多时候我们更多的是选择继承 `RenderBox` 去实现自定义。

综合上述情况，我们知道：

- Widget只是显示的数据配置，所以相对而言是轻量级的存在，而 Flutter 中对 Widget 的也做了一定的优化，所以每次改变状态导致的 Widget 重构并不会有太大的问题。
- RenderObject 就不同了，RenderObject 涉及到布局、计算、绘制等流程，要是每次都全部重新创建开销就比较大了。

所以针对是否每次都需要创建出新的 Element 和 RenderObject 对象，Widget 都做了对应的判断以便于复用，比如：在 `newWidget` 与 `oldWidget` 的 *runtimeType* 和 *key* 相等时会选择使用 `newWidget` 去更新已经存在的 Element 对象，不然就选择重新创建新的 Element。

由此可知：**Widget 重新创建，Element 树和 RenderObject 树并不会完全重新创建。**

看到这，说个题外话：那一般我们可以怎么获取布局的大小和位置呢?

首先这里需要用到我们前文中提过的 `GlobalKey` ，通过 key 去获取到控件对象的 `BuildContext` ，而我们也知道 `BuildContext` 的实现其实是 `Element` ，而 `Element` 持有 `RenderObject` 。So，我们知道的 `RenderObject` ，实际上获取到的就是 `RenderBox` ，那么通过 `RenderBox` 我们就只大小和位置了。

```
showSizes() {
  RenderBox renderBoxRed = fileListKey.currentContext.findRenderObject();
  print(renderBoxRed.size);
}

showPositions() {
  RenderBox renderBoxRed = fileListKey.currentContext.findRenderObject();
  print(renderBoxRed.localToGlobal(Offset.zero));
}
```

--

> 自此，第六篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubAppWeex
- GSYGithubApp React Native

作为系列文章的第七篇，本篇主要在前文的基础上，再深入了解 Widget 和布局中的一些常识性问题。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

在第六篇中我们知道道了 `Widget` 、 `Element` 、 `RenderObject` 三者之间的关系，其中我们最为熟知的 `Widget` ，作为“配置文件”的存在，在 Flutter 中它的功能都是比较单一的，属于 “颗粒度比较细的存在” ，写代码时就像拼乐高“积木”，那这“积木”究竟怎么拼的？下面就 **深入** 去挖挖有意思的东西吧。(￣▽￣)

一、单子元素布局

在 Flutter 单个子元素的布局 Widget 中， **`Container`** 无疑是被用的最广泛的，因为它在“功能”上并不会如 `Padding` 等 Widget 那样功能单一，这是为什么呢?

究其原因，从下图源码可以看出， **`Container`** 其实也只是把其他“单一”的 Widget 做了二次封装，然后通过配置来达到“多功能的效果”而已。

```
@override
Widget build(BuildContext context) {
  Widget current = child;

  if (child == null && (constraints == null || !constraints.isTight)) {
    current = LimitedBox(
      maxWidth: 0.0,
      maxHeight: 0.0,
      child: ConstrainedBox(constraints: const BoxConstraints.expand()),
    ); // LimitedBox
  }

  if (alignment != null)
    current = Align(alignment: alignment, child: current);

  final EdgeInsetsGeometry effectivePadding = _paddingIncludingDecoration;
  if (effectivePadding != null)
    current = Padding(padding: effectivePadding, child: current);

  if (decoration != null)
    current = DecoratedBox(decoration: decoration, child: current);

  if (foregroundDecoration != null) {
    current = DecoratedBox(
      decoration: foregroundDecoration,
      position: DecorationPosition.foreground,
      child: current,
    );
  }

  if (constraints != null)
    current = ConstrainedBox(constraints: constraints, child: current);

  if (margin != null)
    current = Padding(padding: margin, child: current);

  if (transform != null)
    current = Transform(transform: transform, child: current);

  return current;
}
```

接着我们先看 `ConstrainedBox` 源码，从下图源码可以看出，它是继承了 `SingleChildRenderObjectWidget` ，关键是 override 了 `createRenderObject` 方法，返回了 **`RenderConstrainedBox`** 。

> 这里体现了第六篇中的 Widget 与 RenderObject 的关系

是的， **`RenderConstrainedBox`** 就是继承自 `RenderBox` ，从而实现 `RenderObject` 的布局，这里我们得到了它们的关系如下：

Widget	RenderObject
ConstrainedBox	RenderConstrainedBox

```
/// * The [catalog of layout widgets](https://flutter.io/widgets/layout/).
class ConstrainedBox extends SingleChildRenderObjectWidget {
  /// Creates a widget that imposes additional constraints on its child.
  ///
  /// The [constraints] argument must not be null.
  ConstrainedBox({
    Key key,
    @required this.constraints,
    Widget child,
  }) : assert(constraints != null),
       assert(constraints.debugAssertIsValid()),
       super(key: key, child: child);

  /// The additional constraints to impose on the child.
  final BoxConstraints constraints;

  @override
  RenderConstrainedBox createRenderObject(BuildContext context) {
    return RenderConstrainedBox(additionalConstraints: constraints);
  }

  @override
  void updateRenderObject(BuildContext context, RenderConstrainedBox renderObject) {
    renderObject.additionalConstraints = constraints;
  }

  @override
  void debugFillProperties(DiagnosticPropertiesBuilder properties) {
    super.debugFillProperties(properties);
    properties.add(DiagnosticsProperty<BoxConstraints>('constraints', constraints, showName: false));
  }
}
```

然后我们继续对其他每个 Widget 进行观察，可以看到它们也都是继承 `SingleChildRenderObjectWidget` ，而“简单来说”它们不同的地方就是 `RenderObject` 的实现了：

Widget	RenderBox （RenderObject）
Align	RenderPositionedBox
Padding	RenderPadding
Transform	RenderTransform
Offstage	RenderOffstage

所以我们可以总结：**真正的布局和大小计算等行为，都是在 `RenderBox` 上去实现的。** 不同的 Widget 通过各自的 `RenderBox` 实现了“差异化”的布局效果。**所以找每个 Widget 的实现，找它的 `RenderBox` 实现就可以了。** （当然，另外还有 `RenderSliver` ，这里暂时不讨论）

这里我们通过 **`Offstage`** 这个Widget 小结下， **`Offstage`** 这个 Widget 是通过 `offstage` 标志控制 **child** 是否显示的效果，同样的它也有一个 `RenderOffstage` ，如下图，通过 `RenderOffstage` 的源码我们可以“真实”看到 `offstage` 标志位的作用：

```
@override
double computeMinIntrinsicWidth(double height) {
  if (offstage)
    return 0.0;
  return super.computeMinIntrinsicWidth(height);
}

@override
double computeMaxIntrinsicWidth(double height) {
  if (offstage)
    return 0.0;
  return super.computeMaxIntrinsicWidth(height);
}

@override
double computeMinIntrinsicHeight(double width) {
  if (offstage)
    return 0.0;
  return super.computeMinIntrinsicHeight(width);
}

@override
double computeMaxIntrinsicHeight(double width) {
  if (offstage)
    return 0.0;
  return super.computeMaxIntrinsicHeight(width);
}
```

所以大部分时候，我们的 Widget 都是通过实现 `RenderBox` 实现布局的，**那我们可不可抛起 Widget 直接用 `RenderBox` 呢？答案明显是可以的，如果你闲的　疼的话！**

Flutter 官方为了治疗我们“　疼”，提供了一个叫 **`CustomSingleChildLayout`** 的类，它抽象了一个叫 `SingleChildLayoutDelegate` 的对象，让你可以更方便的操作 `RenderBox` 来达到自定义的效果。

```
class CustomSingleChildLayout extends SingleChildRenderObjectWidget {
  /// Creates a custom single child layout.
  ///
  /// The [delegate] argument must not be null.
  const CustomSingleChildLayout({
    Key key,
    @required this.delegate,
    Widget child
  }) : assert(delegate != null),
       super(key: key, child: child);

  /// The delegate that controls the layout of the child.
  final SingleChildLayoutDelegate delegate;

  @override
  RenderCustomSingleChildLayoutBox createRenderObject(BuildContext context) {
    return RenderCustomSingleChildLayoutBox(delegate: delegate);
  }

  @override
  void updateRenderObject(BuildContext context, RenderCustomSingleChildLayoutBox renderObject) {
    renderObject.delegate = delegate;
  }
}
```

如下图三张源码所示， `SingleChildLayoutDelegate` 的对象提供以下接口，并且接口 **前三个** 是按照顺序被调用的，通过实现这个接口，你就可以轻松的控制RenderBox 的 布局位置、大小 等。

```dart
abstract class SingleChildLayoutDelegate {
  /// Creates a layout delegate.
  ///
  /// The layout will update whenever [relayout] notifies its listeners.
  const SingleChildLayoutDelegate({ Listenable relayout }) : _relayout = relayout;

  final Listenable _relayout;

  ///一下顺序执行的哦

  ///给定大小
  Size getSize(BoxConstraints constraints) => constraints.biggest;

  ///约束限制子控件的大小
  BoxConstraints getConstraintsForChild(BoxConstraints constraints) => constraints;

  /// 确定位置
  Offset getPositionForChild(Size size, Size childSize) => Offset.zero;

  ///是否需要重新布局
  bool shouldRelayout(covariant SingleChildLayoutDelegate oldDelegate);
}
```

```dart
@override
void performLayout() {
  size = _getSize(constraints);
  if (child != null) {
    final BoxConstraints childConstraints = delegate.getConstraintsForChild(constraints);
    assert(childConstraints.debugAssertIsValid(isAppliedConstraint: true));
    child.layout(childConstraints, parentUsesSize: !childConstraints.isTight);
    final BoxParentData childParentData = child.parentData;
    childParentData.offset = delegate.getPositionForChild(size, childConstraints.isTight ? childConstraints.smallest : child.size);
  }
}
```

```dart
/// A delegate that controls this object's layout.
SingleChildLayoutDelegate get delegate => _delegate;
SingleChildLayoutDelegate _delegate;
set delegate(SingleChildLayoutDelegate newDelegate) {
  assert(newDelegate != null);
  if (_delegate == newDelegate)
    return;
  final SingleChildLayoutDelegate oldDelegate = _delegate;
  if (newDelegate.runtimeType != oldDelegate.runtimeType || newDelegate.shouldRelayout(oldDelegate))
    markNeedsLayout();
  _delegate = newDelegate;
  if (attached) {
    oldDelegate?._relayout?.removeListener(markNeedsLayout);
    newDelegate?._relayout?.addListener(markNeedsLayout);
  }
}
```

二、多子元素布局

事实上“多子元素布局”和单子元素类似，通过“举一反三”我们就可以知道它们的关系了，比如：

- `Row` 、 `Colum` 都继承了 `Flex` ，而 Flex 继承了 `MultiChildRenderObjectWidget` 并通过 `RenderFlex` 创建了 `RenderBox` ；
- `Stack` 同样继承 `MultiChildRenderObjectWidget` 并通过 `RenderStack` 创建了 `RenderBox` ；

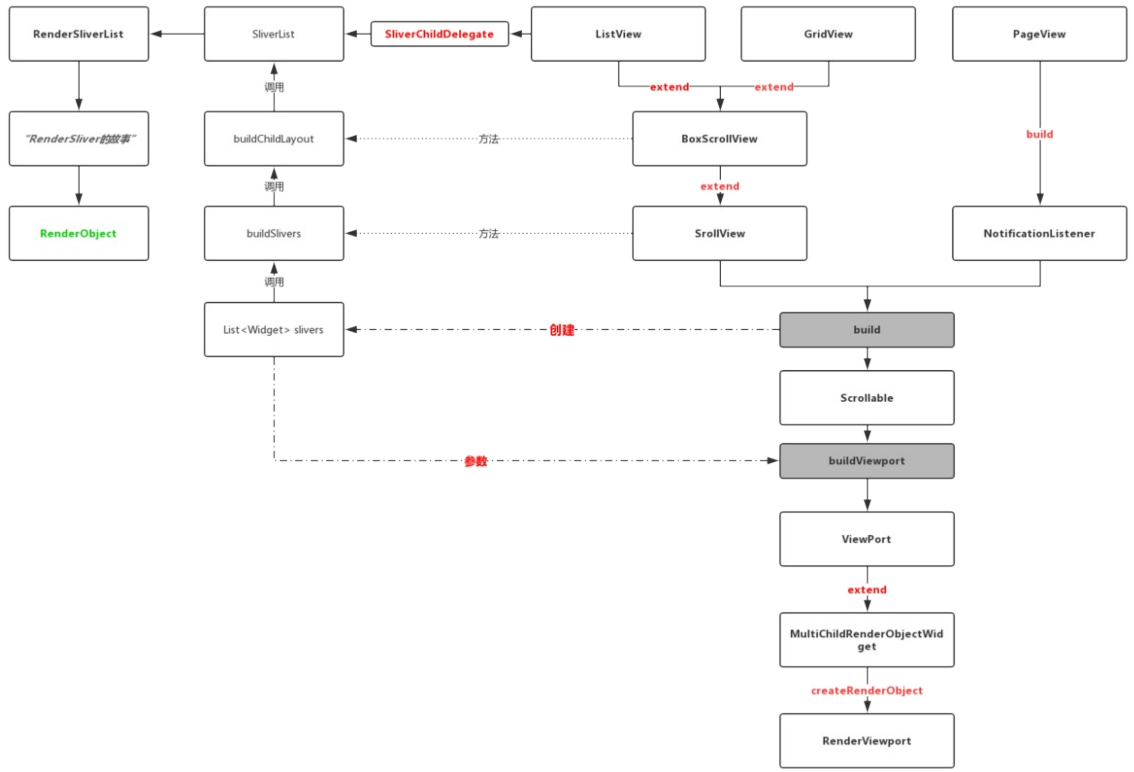
Widget	RenderBox （RenderObject）
Row/Colum/Flex	RenderFlex
Stack	RenderStack
Flow	RenderFlow

Wrap	RenderWrap

同样“多子元素布局”也提供了 `CustomMultiChildLayout` 和 `MultiChildLayoutDelegate` 满足你的“　疼”需求。

三、多子元素滑动布局

滑动布局作为 “多子元素布局” 的另一个分支，如 `ListView` 、 `GridView` 、 `Pageview` ，它们在实现上要复杂的多，从下图一个的流程上我们大致可以知道它们的关系：



由上图我们可以知道，流程最终回产生两个 *RenderObject* ：

- `RenderSliver` ： *Base class for the render objects that implement scroll effects in viewports.*
- `RenderViewport` ： *A render object that is bigger on the inside.*

```
/// [RenderViewport] cannot contain [RenderBox] children directly. Instead, use
/// a [RenderSliverList], [RenderSliverFixedExtentList], [RenderSliverGrid], or
/// a [RenderSliverToBoxAdapter], for example.
```

并且从 `RenderViewport` 的说明我们知道， `RenderViewport` 内部是不能直接放置 `RenderBox` ，需要通过 `RenderSliver` 大家族来完成布局。而从源码可知： **`RenderViewport` 对应的 Widget `Viewport` 就是一个 `MultiChildRenderObjectWidget` 。** （你看，又回到 `MultiChildRenderObjectWidget` 了吧。）

再稍微说下上图的流程：

- `ListView` 、 `Pageview` 、 `GridView` 等都是通过 `Scrollable` 、 `ViewPort` 、 `Sliver` 大家族实现的效果。这里简单不规范描述就是：一个“可滑动”的控件，嵌套了一个“视觉窗口”，然后内部通过“碎片”展示 *children* 。
- 不同的是 `PageView` 没有继承 `SrollView` ，而是直接通过 `NotificationListener` 和 `ScrollNotification` 嵌套实现。

> 注意 `TabBarView` 内部就是： `NotificationListener` + `PageView`

是不是觉得少了什么？哈哈哈，有的有的，官方同样提供了解决“　疼”的自定义滑动 **`CustomScrollView`** ，它继承了 `ScrollView` ，可通过 slivers 参数实现布局，这些 `slivers` 最终回通过 `Scrollable` 的 `buildViewport` 添加到 `ViewPort` 中，如下代码所示：

```
CustomScrollView(
  slivers: <Widget>[
    const SliverAppBar(
      pinned: true,
      expandedHeight: 250.0,
      flexibleSpace: FlexibleSpaceBar(
        title: Text('Demo'),
      ),
    ),
    SliverGrid(
      gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(
        maxCrossAxisExtent: 200.0,
        mainAxisSpacing: 10.0,
        crossAxisSpacing: 10.0,
        childAspectRatio: 4.0,
      ),
      delegate: SliverChildBuilderDelegate(
        (BuildContext context, int index) {
          return Container(
            alignment: Alignment.center,
            color: Colors.teal[100 * (index % 9)],
            child: Text('grid item $index'),
          );
        },
        childCount: 20,
      ),
    ),
    SliverFixedExtentList(
      itemExtent: 50.0,
      delegate: SliverChildBuilderDelegate(
        (BuildContext context, int index) {
          return Container(
            alignment: Alignment.center,
            color: Colors.lightBlue[100 * (index % 9)],
            child: Text('list item $index'),
```

```
          );
        },
      ),
    ),
  ],
)
```

不知道你看完本篇后，有没有对 Flutter 的布局有更深入的了解呢？让我们愉悦的堆积木吧！

> 自此，第七篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

作为系列文章的第八篇，本篇是主要讲述 Flutter 开发过程中的实用技巧，让你少走弯路少掉坑，全篇属于很干的干货总结，以实用为主，算是在深入原理过程中穿插的实用篇章。

文章汇总地址：

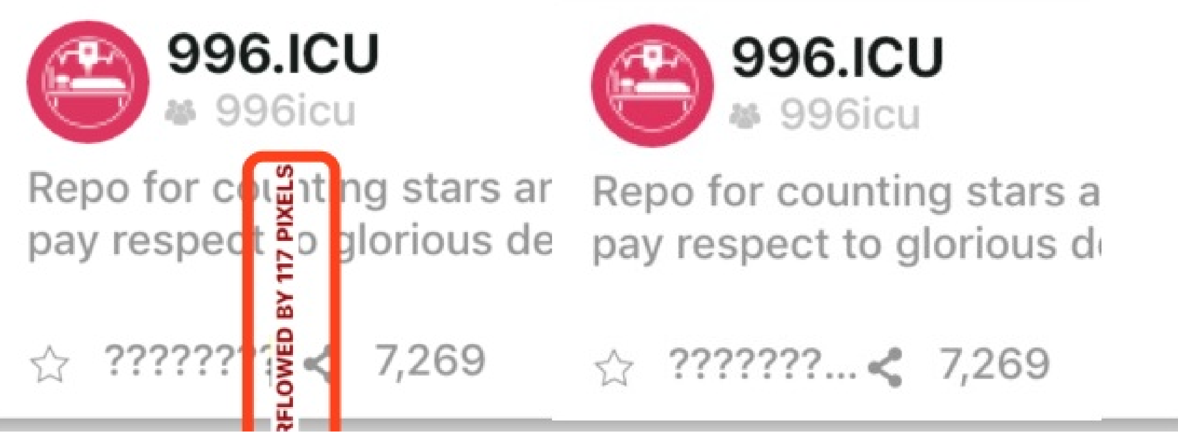
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

1、Text 的 TextOverflow.ellipsis 不生效

有时候我们为 `Text` 设置 *ellipsis* ，却发现并没有生效，而是出现如下图左边提示 `overflowed` 的警告。

其实大部分时候，这是 `Text` 内部的 `RenderParagraph` 在判断 `final bool didOverflowWidth = size.width < textSize.width;` 时， *size.width* 和 *textSize.width* 是相等导致的。

所以你需要给 `Text` 设置一个 `Container` 之类的去约束它的大小，或者是 `Row` 中通过 `Expanded` + `Container` 去约束你的 `Text` ，如果不知道于应该多大，可以通过 `LayoutBuilder` 设置。



2、获取控件的大小和位置

看过第六篇的同学应该知道， 我们可以用 `GlobalKey` ，通过 *key* 去获取到控件对象的 `BuildContext` ，而前面我们也说过 `BuildContext` 的实现其实是 `Element` ，而 `Element` 持有 `RenderObject` 。So，我们知道的 `RenderObject` ，实际上获取到的就是 `RenderBox` ，那么通过 RenderBox 我们就只大小和位置了：

```
showSizes() {
  RenderBox renderBoxRed = fileListKey.currentContext.findRenderObject();
  print(renderBoxRed.size);
}

showPositions() {
  RenderBox renderBoxRed = fileListKey.currentContext.findRenderObject();
  print(renderBoxRed.localToGlobal(Offset.zero));
```

```
    }
```

3、获取状态栏高度和安全布局

如果你看过 `MaterialApp` 的源码，你应该会看到它的内部是一个 `WidgetsApp` ，而 `WidgetsApp` 内有一个 `MediaQuery` ，熟悉它的朋友知道我们可以通过 `MediaQuery.of(context).size` 去获取屏幕大小。

其实 `MediaQuery` 是一个 `InheritedWidget` ，它有一个叫 `MediaQueryData` 的参数，这个参数是通过如下图设置的，再通过源码我们知道，一般情况下 `MediaQueryData` 的 `padding` 的 `top` 就是状态栏的高度。

所以我们可以通过 `MediaQueryData.fromWindow(WidgetsBinding.instance.window).padding.top` 获取到状态栏高度，当然有时候可能需要考虑 `viewInsets` 参数。

```
  return MediaQuery(
    data: MediaQueryData.fromWindow(WidgetsBinding.instance.window),
    child: Localizations(
      locale: appLocale,
      delegates: _localizationsDelegates.toList(),
      child: title,
    ), // Localizations
  ); // MediaQuery
```

至于 `AppBar` 的高度，默认是 `Size.fromHeight(kToolbarHeight + (bottom?.preferredSize?.height ?? 0.0))`， *kToolbarHeight* 是一个固定数据，当然你可以通过实现 `PreferredSizeWidget` 去自定义 `AppBar` 。

同时你可能会发现，有时候在布局时发现布局位置不正常，居然是从状态栏开始计算，这时候你需要用 `SafeArea` 嵌套下，至于为什么，看源码你就会发现 `MediaQueryData` 的存在。

4、设置状态栏颜色和图标颜色

简单的可以通过 `AppBar` 的 *brightness* 或者 `ThemeData` 去设置状态栏颜色。

但是如果你不想用 `AppBar` ，那么你可以嵌套 `AnnotatedRegion<SystemUiOverlayStyle>` 去设置状态栏样式，通过 `SystemUiOverlayStyle` 就可以快速设置状态栏和底部导航栏的样式。

同时你还可以通过 `SystemChrome.setSystemUIOverlayStyle` 去设置，前提是你没有使用 `AppBar` 。**需要注意的是，所有状态栏设置是全局的，** 如果你在 A 页面设置后，B 页面没有手动设置或者使用 AppBar ，那么这个设置将直接呈现在 B 页面。

5、系统字体缩放

现在的手机一般都提供字体缩放，这给应用开发的适配上带来一定工作量，所以大多数时候我们会选择禁止应用跟随系统字体缩放。

在 Flutter 中字体缩放也是和 `MediaQueryData` 的 `textScaleFactor` 有关。所以我们可以在需要的页面，通过最外层嵌套如下代码设置，将字体设置为默认不允许缩放。

```
      MediaQuery(
```

```
        data: MediaQueryData.fromWindow(WidgetsBinding.instance.window).copyWith(text
ScaleFactor: 1),
        child: new Container(),
      );
```

6、Margin 和 Padding

在使用 `Container` 的时候我们经常会使用到 *margin* 和 *padding* 参数，其实在上一篇我们已经说过， `Container` 其实只是对各种布局的封装，内部的 *margin* 和 *padding* 其实是通过 `Padding` 实现的，而 `Padding` 不支持负数，所以如果你需要用到负数的情况下，推荐使用 `Transform` 。

```
    Transform(
        transform: Matrix4.translationValues(10, -10, 0),
        child: new Container(),
      );
```

7、控件圆角裁剪

日常开发中我们大致上会使用两种圆角方案：

- 一种是通过 `Decoration` 的实现类 `BoxDecoration` 去实现。
- 一种是通过 `ClipRRect` 去实现。

其中 `BoxDecoration` 一般应用在 `DecoratedBox` 、 `Container` 等控件，这种实现一般都是直接 *Canvas* 绘制时，针对当前控件的进行背景圆角化，并不会影响其 *child* 。这意味着如果你的 *child* 是图片或者也有背景色，那么很可能圆角效果就消失了。

而 `ClipRRect` 的效果就是会影响 *child* 的，具体看看其如下的 RenderObject 源码可知。

```
/// Clip further painting using a rounded rectangle.
///
/// * `needsCompositing` is whether the child needs compositing. Typically
///   matches the value of [RenderObject.needsCompositing] for the caller.
/// * `offset` is the offset from the origin of the canvas' coordinate system
///   to the origin of the caller's coordinate system.
/// * `bounds` is the region of the canvas (in the caller's coordinate system)
///   into which `painter` will paint in.
/// * `clipRRect` is the rounded-rectangle (in the caller's coordinate system)
///   to use to clip the painting done by `painter`.
/// * `painter` is a callback that will paint with the `clipRRect` applied. This
///   function calls the `painter` synchronously.
/// * `clipBehavior` controls how the path is clipped.
void pushClipRRect(bool needsCompositing, Offset offset, Rect bounds, RRect clipRRect, PaintingContextCallback painter, { Clip clipBehavior = Clip.antiAlias }
  assert(clipBehavior != null);
  final Rect offsetBounds = bounds.shift(offset);
  final RRect offsetClipRRect = clipRRect.shift(offset);
  if (needsCompositing) {
    pushLayer(ClipRRectLayer(clipRRect: offsetClipRRect, clipBehavior: clipBehavior), painter, offset, childPaintBounds: offsetBounds);
  } else {
    clipRRectAndPaint(offsetClipRRect, clipBehavior, offsetBounds, () => painter(this, offset));
  }
}
```

8、PageView

如果你在使用 `TarBarView` ，并且使用了 `KeepAlive` 的话，那么我推荐你直接使用 `PageView` 。因为目前到 1.2 的版本，在 `KeepAlive` 的 状态下，跨两个页面以上的 Tab 直接切换， `TarBarView` 会导致页面的 `dispose` 再重新 `initState` 。尽管 `TarBarView` 内也是封装了 `PageView` + `TabBar` 。

你可以直接使用 `PageView` + `TabBar` 去实现，然后 tab 切换时使用 `_pageController.jumpTo(MediaQuery.of(context).size.width * index);` 可以避免一些问题。当然，这时候损失的就是动画效果了。事实上 `TarBarView` 也只是针对 `PageView` + `TabBar` 做了一层封装。

除了这个，其实还有第二种做法，使用如下方 **`PageStorageKey`** 保持页面数状态，但是因为它是 *save and restore values*，所以的页面的 `dispose` 再重新 `initState` 方法，每次都会被调用。

```
    return new Scaffold(
      key: new PageStorageKey<your value type>(your value)
    )
```

9、懒加载

Flutter 中通过 `FutureBuilder` 或者 `StreamBuilder` 可以和简单的实现懒加载，通过 `future` 或者 `stream` “异步” 获取数据，之后通过 `AsyncSnapshot` 的 data 再去加载数据，至于流和异步的概念，以后再展开吧。

10、Android 返回键回到桌面

Flutter 官方已经为你提供了 android_intent 插件了，这种情况下，实现回到桌面可以如下简单实现：

```
  Future<bool> _dialogExitApp(BuildContext context) async {
    if (Platform.isAndroid) {
      AndroidIntent intent = AndroidIntent(
        action: 'android.intent.action.MAIN',
        category: "android.intent.category.HOME",
      );
      await intent.launch();
    }

    return Future.value(false);
  }
 .....
 return WillPopScope(
      onWillPop: () {
        return _dialogExitApp(context);
      },
      child:xxx);
```

> 自此，第八篇终于结束了！(///▽///)

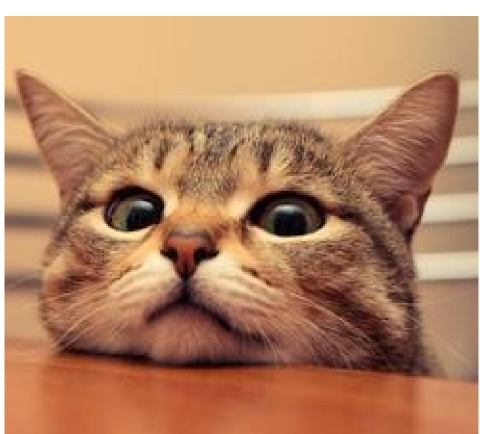
资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**

- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex



作为系列文章的第九篇，本篇主要深入了解 Widget 中绘制相关的原理，探索 Flutter 里的 RenderObject 最后是如何走完屏幕上的最后一步，结尾再通过实际例子理解如何设计一个 Flutter 的自定义绘制。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

在第六、第七篇中我们知道了 `Widget` 、 `Element` 、 `RenderObject` 的关系，同时也知道了 `Widget` 的布局逻辑，最终所有 `Widget` 都转化为 `RenderObject` 对象， 它们堆叠出我们想要的画面。

所以在 Flutter 中，最终页面的 `Layout` 、 `Paint` 等都会发生在 Widget 所对应的 `RenderObject` 子类中，而 `RenderObject` 也是 Flutter 跨平台的最大的特点之一：**所有的控件都与平台无关**，这里简单的人话就是： **Flutter 只要求系统提供的 “Canvas”，然后开发者通过 Widget 生成 `RenderObject` “直接” 通过引擎绘制到屏幕上。**

> ps 从这里开始篇幅略长，可能需要消费您的一点耐心。

一、绘制过程

我们知道 `Widget` 最终都转化为 `RenderObject` ， 所以了解绘制我们直接先看 `RenderObject` 的 `paint` 方法。

如下图所示，所有的 `RenderObject` 子类都必须实现 `paint` 方法，并且该方法并不是给用户直接调用，需要更新绘制时，你可以通过 `markNeddsPaint` 方法去触发界面绘制。

```
/// Paint this render object into the given context at the given offset.
///
/// Subclasses should override this method to provide a visual appearance
/// for themselves. The render object's local coordinate system is
/// axis-aligned with the coordinate system of the context's canvas and the
/// render object's local origin (i.e, x=0 and y=0) is placed at the given
/// offset in the context's canvas.
///
/// Do not call this function directly. If you wish to paint yourself, call
/// [markNeedsPaint] instead to schedule a call to this function. If you wish
/// to paint one of your children, call [PaintingContext.paintChild] on the
/// given `context`.
///
/// When painting one of your children (via a paint child function on the
/// given context), the current canvas held by the context might change
/// because draw operations before and after painting children might need to
/// be recorded on separate compositing layers.
void paint(PaintingContext context, Offset offset) { }
```

那么，按照“国际流程”，在经历大小和布局等位置计算之后，最终 `paint` 方法会被调用，该方法带有两个参数： `PaintingContext` 和 `Offset` ，它们就是完成绘制的关键所在，那么相信此时大家肯定有个疑问就是：

- `PaintingContext` 是什么?
- `Offset` 是什么?

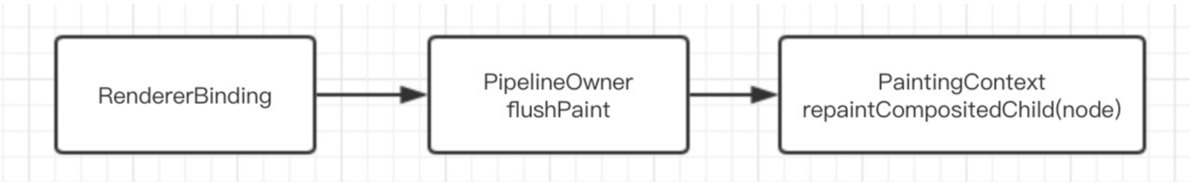
通过飞速查阅源码，我们可以首先了解到有：

- `PaintingContext` 的关键是 **A place to paint**， 同时它在父类 `ClipContext` 是包含有 `Canvas` ，并且 `PaintingContext` 的构造方法是 `@protected` ，只在 `PaintingContext.repaintCompositedChild` 和 `pushLayer` 时自动创建。
- `Offset` 在 `paint` 中主要是提供当前控件在屏幕的相对偏移值，提供绘制时确定绘制的坐标。

```
/// A place to paint.
///
/// Rather than holding a canvas directly, [RenderObject]s paint using a painting
/// context. The painting context has a [Canvas], which receives the
/// individual draw operations, and also has functions for painting child
/// render objects.
///
/// When painting a child render object, the canvas held by the painting context
/// can change because the draw operations issued before and after painting the
/// child might be recorded in separate compositing layers. For this reason, do
/// not hold a reference to the canvas across operations that might paint
/// child render objects.
///
/// New [PaintingContext] objects are created automatically when using
/// [PaintingContext.repaintCompositedChild] and [pushLayer].
class PaintingContext extends ClipContext {

  /// Creates a painting context.
  ///
  /// Typically only called by [PaintingContext.repaintCompositedChild]
  /// and [pushLayer].
  @protected
  PaintingContext(this._containerLayer, this.estimatedBounds)
    : assert(_containerLayer != null),
      assert(estimatedBounds != null);

  final ContainerLayer _containerLayer;
```

OK，继续往下走，那么既然 `PaintingContext` 叫 Context， 那它肯定是存在上下文关系，那它是在哪里开始创建的呢?

通过调试源码可知，项目在 `runApp` 时通过 `WidgetsFlutterBinding` 启动，而在以前的篇幅中我们知道， `WidgetsFlutterBinding` 是一个“胶水类”，它会触发 *mixin* 的 `RendererBinding` ，如下图创建出根 node 的 `PaintingContext` 。



好了，那么 `Offset` 呢? 如下图，对于 `Offset` 的传递，是通过父控件和子控件的 offset 相加之后，一级一级的将需要绘制的坐标结合去传递的。

目前简单来说，**通过 `PaintingContext` 和 `Offset` ，在布局之后我们就可以在屏幕上准确的地方绘制会需要的画面。**

```
/// Paints each child by walking the child list forwards.
///
/// See also:
///
///  * [defaultHitTestChildren], which implements hit-testing of the children
///    in a manner appropriate for this painting strategy.
void defaultPaint(PaintingContext context, Offset offset) {
  ChildType child = firstChild;
  while (child != null) {
    final ParentDataType childParentData = child.parentData;
    context.paintChild(child, childParentData.offset + offset);
    child = childParentData.nextSibling;
  }
}
```

1、测试绘制

这里我们先做一个有趣的测试。

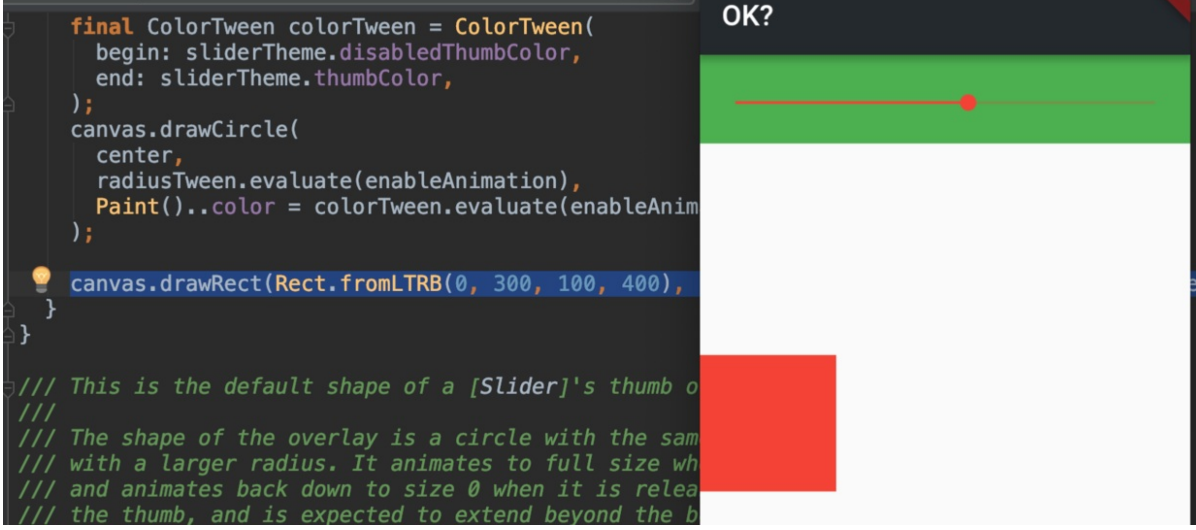
我们现在屏幕上通过 `Container` 限制一个高为 60 的绿色容器，如下图，暂时忽略容器内的 `Slider` 控件， 我们图中绘制了一个 *100 x 100* 的红色方块，这时候我们会看到下图右边的效果是：纳尼？为什么只有这么小?

事实上，因为正常 Flutter 在绘制 `Container` 的时候， `AppBar` 已经帮我们计算了状态栏和标题栏高度偏差，但我们这里在用 `Canvas` 时直接粗暴的 `drawRect` ，绘制出来的红色小方框，**左部和顶部起点均为0，其实是从状态栏开始计算绘制的。**

```
    radiusTween.evaluate(enableAnimation),
    Paint()..color = colorTween.evaluate(enableAn
  );

  canvas.drawRect(Rect.fromLTRB(0, 0, 100, 100),
}
```

OK?

那如果我们调整位置呢？把起点 top 调整到 300，出现了如下图的效果：纳尼？红色小方块居然画出去了，明明 `Container` 只有绿色的大小。



其实这里的问题还是在于 `PaintingContext` ，它有一个参数是 `estimatedBounds` ，而 `estimatedBounds` 正常是在创建时通过 `child.paintBounds` 赋值的，但是对于 `estimatedBounds` 还有如下的描述：**原来画出去也是可以。**

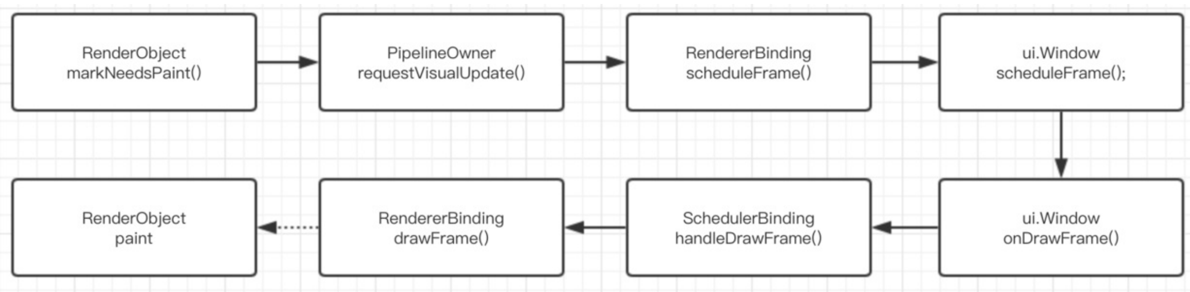
```
The canvas will allow painting outside these bounds.
The [estimatedBounds] rectangle is in the [canvas] coordinate system.
```

所以到这里你可以通俗的总结， **对于 Flutter 而言，整个屏幕都是一块画布，我们通过各种 `Offset` 和 `Rect` 确定了位置，然后通过 `PaintingContext` 的 `Canvas` 绘制上去，目标是整个屏幕区域，整个屏幕就是一帧，每次改变都是重新绘制。**

2、RepaintBoundary

当然，**每次重新绘制并不是完全重新绘制**，这里面其实是存在一些规制的。

还记得前面的 `markNeedsPaint` 方法吗？我们先从 `markNeedsPaint()` 开始， 总结出其大致流程如下图，可以看到 `markNeedsPaint` 在 `requestVisualUpdate` 时确实触发了引擎去更新绘制界面。



接着我们看源码，如源码所示，当调用 `markNeedsPaint()` 时， `RenderObject` 就会往上的父节点去查找，根据 `isRepaintBoundary` 是否为 true，会决定是否从这里开始去触发重绘。换个说法就是，**确定要更新哪些区域。**

所以其实流程应该是：**通过 `isRepaintBoundary` 往上确定了更新区域，通过 `requestVisualUpdate` 方法触发更新往下绘制。**

```
void markNeedsPaint() {
  assert(owner == null || !owner.debugDoingPaint);
  if (_needsPaint)
    return;
  _needsPaint = true;
  if (isRepaintBoundary) {
    assert(() {
      if (debugPrintMarkNeedsPaintStacks)
        debugPrintStack(label: 'markNeedsPaint() called for $this');
      return true;
    }());
    // If we always have our own layer, then we can just repaint
    // ourselves without involving any other nodes.
    assert(_layer != null);
    if (owner != null) {
      owner._nodesNeedingPaint.add(this);
      owner.requestVisualUpdate();
    }
  } else if (parent is RenderObject) {
    // We don't have our own layer; one of our ancestors will take
    // care of updating the layer we're in and when they do that
    // we'll get our paint() method called.
    assert(_layer == null);
    final RenderObject parent = this.parent;
    parent.markNeedsPaint();
    assert(parent == this.parent);
  } else {
    assert(() {
      if (debugPrintMarkNeedsPaintStacks)
        debugPrintStack(label: 'markNeedsPaint() called for $this (root of render tree)');
      return true;
    }());
    // If we're the root of the render tree (probably a RenderView),
    // then we have to paint ourselves, since nobody else can paint
    // us. We don't add ourselves to _nodesNeedingPaint in this
    // case, because the root is always told to paint regardless.
    if (owner != null)
      owner.requestVisualUpdate();
  }
}
```

并且从源码中可以看出， `isRepaintBoundary` 只有 `get` ，所以它只能被子类 `override` ，由子类表明是否是为重绘的边缘，比如 `RenderProxyBox` 、 `RenderView` 、 `RenderFlow` 等 `RenderObject` 的 `isRepaintBoundary` 都是 true。

所以如果一个区域绘制很频繁，且可以不影响父控件的情况下，其实可以将 override `isRepaintBoundary` 为 true。

3、Layer

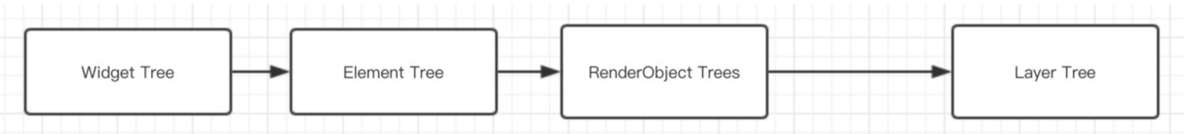
上文我们知道了，当 `isRepaintBoundary` 为 true 时，那么该区域就是一个可更新绘制区域，而当这个区域形成时， 其实就会新创建一个 **`Layer`** 。

不同的 `Layer` 下的 `RenderObject` 是可以独立的工作，比如 `OffsetLayer` 就在 `RenderObject` 中用到，它就是用来做定位绘制的。

```
/// A layer that is displayed at an offset from its parent layer.
///
/// Offset layers are key to efficient repainting because they are created by
/// repaint boundaries in the [RenderObject] tree (see
/// [RenderObject.isRepaintBoundary]). When a render object that is a repaint
/// boundary is asked to paint at given offset in a [PaintingContext], the
/// render object first checks whether it needs to repaint itself. If not, it
/// reuses its existing [OffsetLayer] (and its entire subtree) by mutating its
/// [offset] property, cutting off the paint walk.
class OffsetLayer extends ContainerLayer {
  /// Creates an offset layer.
```

同时这也引生出了一个结论：**不是每个 `RenderObject` 都具有 `Layer` 的，因为这受 `isRepaintBoundary` 的影响。**

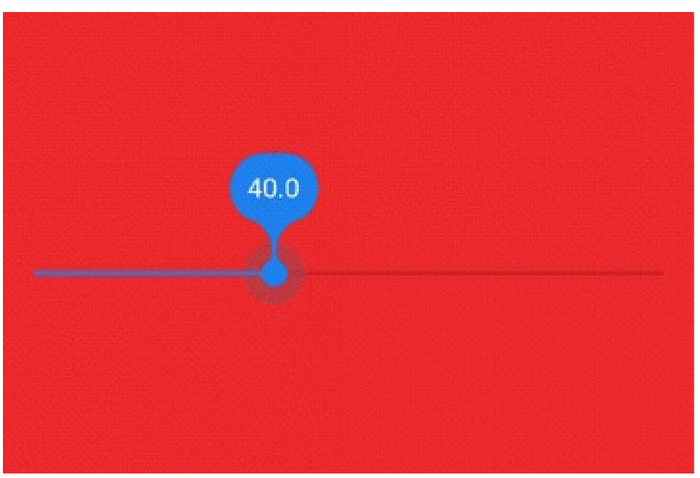
其次在 `RenderObject` 中还有一个属性叫 `needsCompositing` ，它会影响生成多少层的 **`Layer`** ，而这些 **`Layer`** 又会组成一棵 **Layer Tree** 。好吧，到这里又多了一个树，实际上这颗树才是所谓真正去给引擎绘制的树。



到这里我们大概就了解了 `RenderObject` 的整个绘制流程，并且这个**绘制时机我们是去“触发”的，而不是主动调用，并且更新是判断区域的。** 嗯～有点 React 的味道！

二、Slider 控件的绘制实现

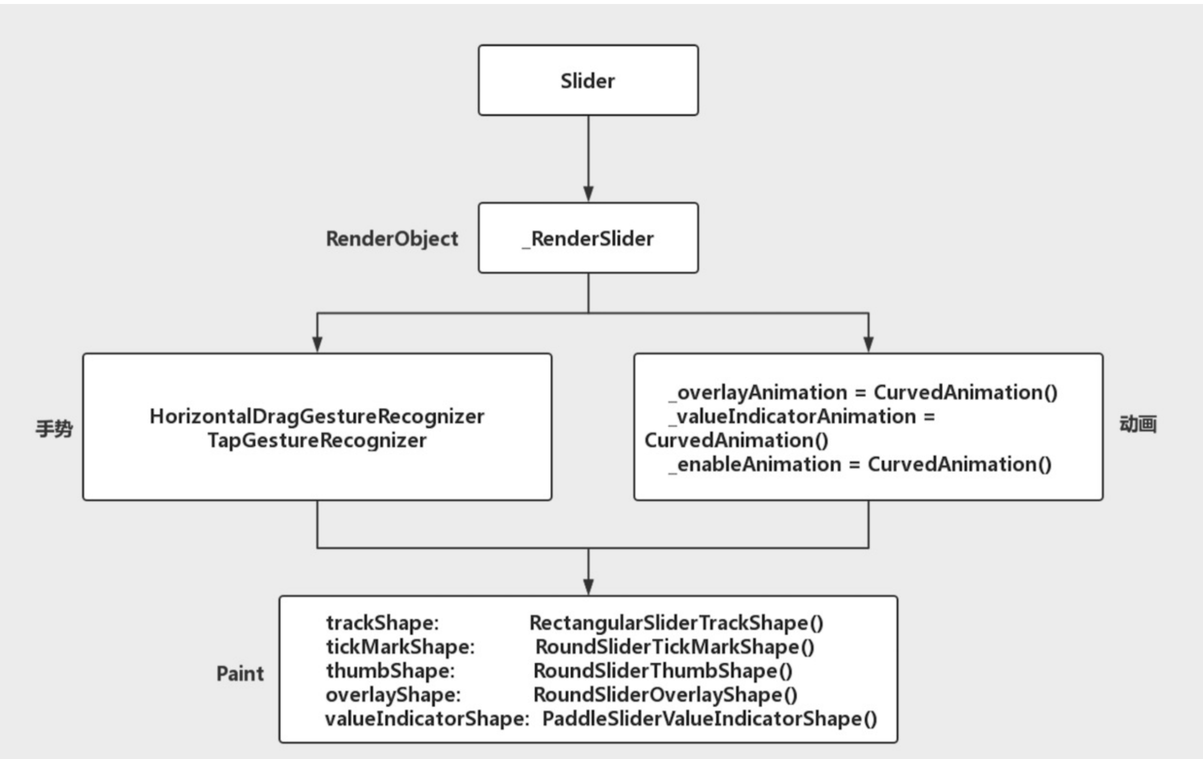
前面我们讲了那么多绘制的流程，现在让我们从 `Slider` 这个控件的源码，去看看一个绘制控件的设计实现吧。



整个 `Slider` 的实现可以说是很 `Flutter` 了，大体结构如下图。

在 `_RenderSlider` 中，除了 **手势** 和 **动画** 之外，其余的每个绘制的部分，都是独立的 *Component* 去完成绘制，而这些 *Component* 都是通过 `SliderTheme` 的 `SliderThemeData` 提供的。

巧合的是， `SliderTheme` 本身就是一个 `InheritedWidget` 。看过以前篇章的同学应该会知道， `InheritedWidget` 一般就是用于做状态共享的，所以如果你需要自定义 `Slider` ，完成可以通过 `SliderTheme` 嵌套，然后通过 `SliderThemeData` 选择性的自定义你需要的模块。



并且如下图，在 `_RenderSlider` 中注册时手势和动画，会在监听中去触发 `markNeedsPaint` 方法，这就是为什么你的触摸能够响应画面的原因了。

```
///手势和触摸
final GestureArenaTeam team = GestureArenaTeam();
_drag = HorizontalDragGestureRecognizer()
  ..team = team
  ..onStart = _handleDragStart
  ..onUpdate = _handleDragUpdate
  ..onEnd = _handleDragEnd
  ..onCancel = _endInteraction;
_tap = TapGestureRecognizer()
  ..team = team
  ..onTapDown = _handleTapDown
  ..onTapUp = _handleTapUp
  ..onTapCancel = _endInteraction;
///通过Animation实现数值上的动画效果
_overlayAnimation = CurvedAnimation(
  parent: _state.overlayController,
  curve: Curves.fastOutSlowIn,
);
_valueIndicatorAnimation = CurvedAnimation(
  parent: _state.valueIndicatorController,
  curve: Curves.fastOutSlowIn,
);
_enableAnimation = CurvedAnimation(
  parent: _state.enableController,
  curve: Curves.easeInOut,
);
}
```

```
@override
void attach(PipelineOwner owner) {
  super.attach(owner);
  _overlayAnimation.addListener(markNeedsPaint);
  _valueIndicatorAnimation.addListener(markNeedsPaint)
  _enableAnimation.addListener(markNeedsPaint);
  _state.positionController.addListener(markNeedsPaint)
}

@override
void detach() {
  _overlayAnimation.removeListener(markNeedsPaint);
  _valueIndicatorAnimation.removeListener(markNeedsPai
  _enableAnimation.removeListener(markNeedsPaint);
  _state.positionController.removeListener(markNeedsPa
  super.detach();
}

void _startInteraction(Offset globalPosition) {
  if (isInteractive) {
    _active = true;
    // We supply the *current* value as the start location, so that if we have
    // a tap, it consists of a call to onChangeStart with the previous value
    // a call to onChangeEnd with the new value.
    if (onChangeStart != null) {
      onChangeStart(_discretize(value));
    }
    _currentDragValue = _getValueFromGlobalPosition(globalPosition);
    onChanged(_discretize(_currentDragValue));
    _state.overlayController.forward();
    if (showValueIndicator) {
      _state.valueIndicatorController.forward();
      _state.interactionTimer?.cancel();
      _state.interactionTimer = Timer(_minimumInteractionTime * timeDilation,
        _state.interactionTimer = null;
        if (!_active &&
```

同时可以看到 `_SliderRender` 内的参数都重写了 `get` 、 `set` 方法，在 `set` 时也会有 `markNeedsPaint()` ，或者调用 `_updateLabelPainter` 去间接调用 `markNeedsLayout` 。

```
int get divisions => _divisions;
int _divisions;
set divisions(int value) {
  if (value == _divisions) {
    return;
  }
  _divisions = value;
  markNeedsPaint();
}

String get label => _label;
String _label;
set label(String value) {
  if (value == _label) {
    return;
  }
  _label = value;
  _updateLabelPainter();
}

SliderThemeData get sliderTheme => _sliderTheme;
SliderThemeData _sliderTheme;
set sliderTheme(SliderThemeData value) {
  if (value == _sliderTheme) {
    return;
  }
  _sliderTheme = value;
  markNeedsPaint();
}

ThemeData get theme => _theme;
ThemeData _theme;
set theme(ThemeData value) {
  if (value == _theme) {
    return;
  }
  _theme = value;
  markNeedsPaint();
}
```

至于 `Slider` 内的各种 Shape 的绘制这里就不展开了，都是 `Canvas` 标准的 `pathTo` 、 `drawRect` 、 `translate` 、 `drawPath` 等熟悉的操作了。

> 自此，第九篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

作为系列文章的第十篇，本篇主要深入了解 Flutter 中图片加载的流程，剥析图片流程中有意思的片段，结尾再实现 Flutter 实现本地图片缓存的支持。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

在 Flutter 中，图片的加载主要是通过 `Image` 控件实现的，而 `Image` 控件本身是一个 **StatefulWidget** ，通过前文我们可以快速想到， `Image` 肯定对应有它的 **RenderObject** 负责 *layout* 和 *paint* ，那么这个过程中，图片是如何变成画面显示出来的?

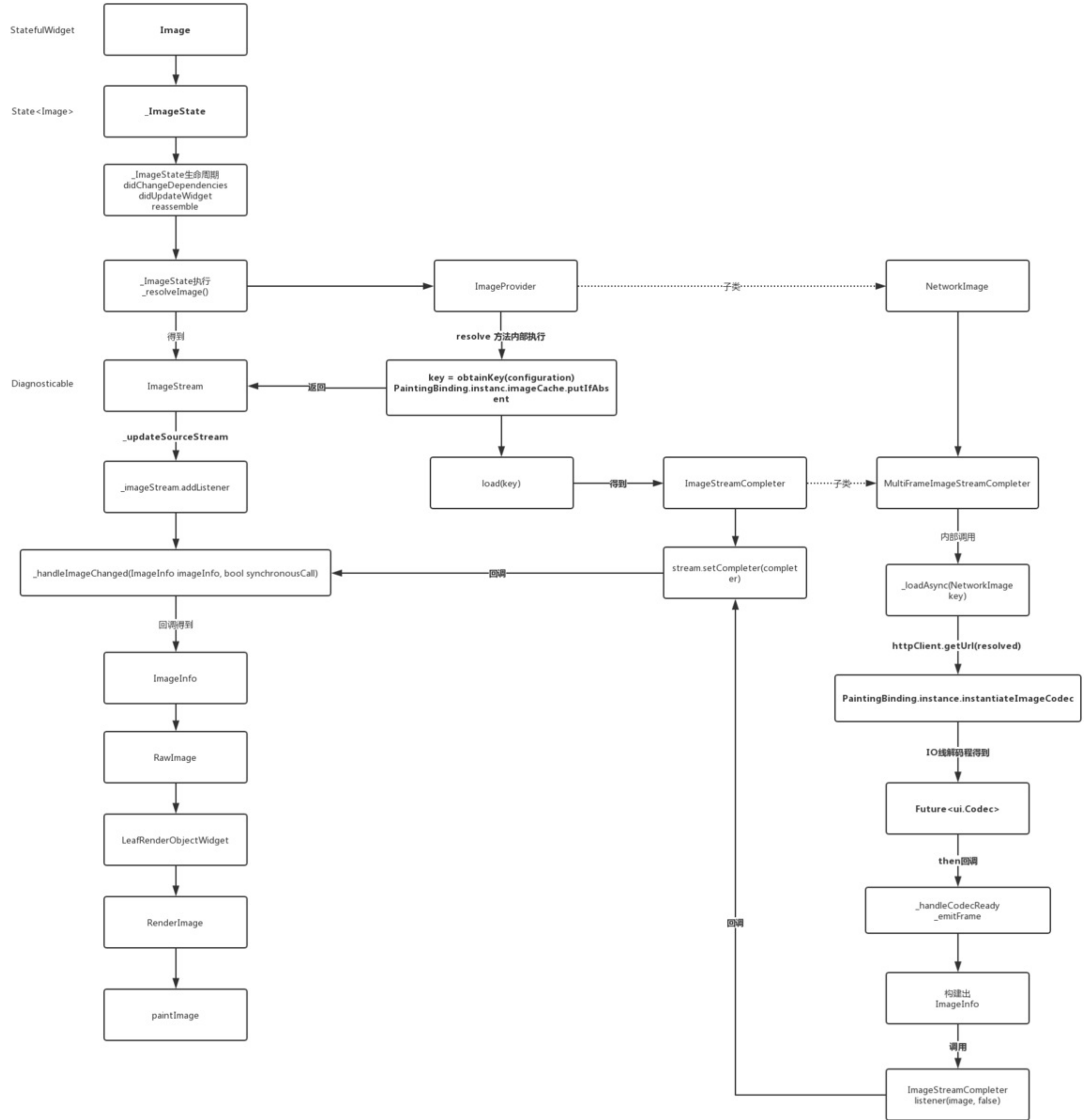
一、图片流程

Flutter 的图片加载流程其实“并不复杂”，具体可点击下方大图查看，以网络图片加载为例子，先简单总结，其中主要流程是：

- 1、首先 `Image` 通过 `ImageProvider` 得到 `ImageStream` 对象
- 2、然后 `_ImageState` 利用 `ImageStream` 添加监听，等待图片数据
- 3、接着 `ImageProvider` 通过 `load` 方法去加载并返回 `ImageStreamCompleter` 对象
- 4、然后 `ImageStream` 会关联 `ImageStreamCompleter`
- 5、之后 `ImageStreamCompleter` 会通过 http 下载图片，再经过 `PaintingBinding` 编码转化后，得到 `ui.Codec` 可绘制对象，并封装成 `ImageInfo` 返回
- 6、接着 `ImageInfo` 回调到 `ImageStream` 的监听，设置给 `_ImageState` build 的 `RawImage` 对象。
- 7、最后 `RawImage` 的 `RenderImage` 通过 paint 绘制 `ImageInfo` 中的 `ui.Codec`

> 注意，这的 `ui.Codec` 和后面的 `ui.Image` 等，只是因为 Flutter 中在导入对象时，为了和其他类型区分而加入的重命名： `import 'dart:ui' as ui show Codec;`

是不是感觉有点晕了？relax！后面我们将逐步理解这个流程。



在 Flutter 的图片的加载流程中，主要有三个角色：

- **`Image`** ：用于显示图片的 Widget，最后通过内部的 **`RenderImage` 绘制**。
- **`ImageProvider`** ：提供加载图片的方式如 `NetworkImage` 、 `FileImage` 、 `MemoryImage` 、 `AssetImage` 等，从而获取 **`ImageStream`** ，用于**监听结果**。
- **`ImageStream`** ：图片的加载对象，通过 `ImageStreamCompleter` 最后会返回一个 `ImageInfo` ，而 `ImageInfo` 内包含有 `RenderImage` 最后的**绘制对象 `ui.Image`** 。

从上面的大图流程可知，网络图片是通过 `NetworkImage` 这个 *Provider* 去提供加载的，各类 *Provider* 的实现其实大同小异，其中主要需要实现的方法主要如下图所示：

```
/// Converts an ImageProvider's settings plus an ImageConfiguration to a key
/// that describes the precise image to load.
///
/// The type of the key is determined by the subclass. It is a value that
/// unambiguously identifies the image (_including its scale_) that the [load]
/// method will fetch. Different [ImageProvider]s given the same constructor
/// arguments and [ImageConfiguration] objects should return keys that are
/// '==' to each other (possibly by using a class for the key that itself
/// implements [==]).
@protected
Future<T> obtainKey(ImageConfiguration configuration);

/// Converts a key into an [ImageStreamCompleter], and begins fetching the
/// image.
@protected
ImageStreamCompleter load(T key);
```

1、obtainKey

该方法主要用于标示当前 `Provider` 的存在，比如在 `NetworkImage` 中，这个方法返回的是 `SynchronousFuture<NetworkImage>(this)` ，也就是 `NetworkImage` 自己本身，并且得到的这个 key 在 `ImageProvider` 中，是用于**作为内存缓存的 key 值**。

在 `NetworkImage` 中主要是通过 `runtimeType` 、 `url` 、 `scale` 这三个参数判断两个 `NetworkImage` 是否相等，所以除了 `url` ，图片的 `scale` 同样会影响缓存的对象哦。

2、load(T key)

`load` 方法顾名思义就是加载了，而该方法中所使用的 key ，毫无疑问就是上面 `obtainKey` 方法所提供的。

`load` 方法返回的是 `ImageStreamCompleter` 抽象对象，它主要是用于**管理和通知 `ImageStream` 中得到的 `dart:ui.Image`** ，比如在 `NetworkImage` 中的是子类 `MultiFrameImageStreamCompleter` ，它可以处理多帧的动画，如果图片只有一针，那么将执行一次都结束。

3、resolve

`ImageProvider` 的关键在于 `resolve` 方法，从流程图我们可知，该方法在 `Image` 的生命周期回调方法 `didChangeDependencies` 、 `didUpdateWidget` 、 `reassemble` 里会被调用，从下方源码可以看出，上面我们所实现的 `obtainKey` 和 `load` 都会在这里被调用

```
/// method.
ImageStream resolve(ImageConfiguration configuration) {
  assert(configuration != null);
  final ImageStream stream = ImageStream();

  final Zone dangerZone = Zone.current.fork(
    specification: ZoneSpecification(
      handleUncaughtError: (Zone zone, ZoneDelegate delegate, Zone parent, Object error, StackTrace stackTr
        handleError(error, stackTrace);
      }
    )
  );
  dangerZone.runGuarded(() {
    Future<T> key;
    try {
      key = obtainKey(configuration);
    } catch (error, stackTrace) {
      handleError(error, stackTrace);
      return;
    }
    key.then<void>((T key) {
      obtainedKey = key;
      final ImageStreamCompleter completer = PaintingBinding.instance
          .imageCache.putIfAbsent(key, () => load(key), onError: handleError);
      if (completer != null) {
        stream.setCompleter(completer);
      }
    }).catchError(handleError);
  });
  return stream;
}
```

> 这个有个有意思的对象，就是 `Zone` ！
>
> 因为在 Flutter 中，同步异常可以通过try-catch捕获，而异步异常如 `Future` ，是无法被当前的 try-catch 直接捕获的。
>
> 所以在 Dart中 `Zone` 的概念，你可以给执行对象指定一个 `Zone`，类似提供一个沙箱环境，而在这个沙箱内，你就可以全部可以捕获、拦截或修改一些代码行为，比如所有未被处理的异常。

`resolve` 方法内主要是用到了 **`PaintingBinding.instance.imageCache.putIfAbsent(key, () => load(key)`** ， `PaintingBinding` 是一个胶水类，主要是通过 Mixins 粘在 `WidgetsFlutterBinding` 上使用，而以前的篇章我们说过， `WidgetsFlutterBinding` 就是我们的启动方法 `runApp` 的执行者。

所以图片缓存是在PaintingBinding.instance.imageCache内单例维护的。

如下图所示， `putIfAbsent` 方法内部，主要是通过 `key` 判断**内存中是否已有缓存、或者正在缓存的对象**，如果是就返回该 `ImageStreamCompleter` ，不然就调用 `loader` 去加载并返回。

值得注意的是，此时的的 cache 是有两个状态的，因为返回的 `ImageStreamCompleter` 并不代表着图片就加载完成，所以如果是首次加载，会先有 **`_PendingImage`** 用于标示该key的图片处于**加载中的状态**，并且添加一个 `listener`， 用于图片加载完成后，替换为缓存 `_CacheImage` 。

```
ImageStreamCompleter putIfAbsent(Object key, ImageStreamCompleter loader(), { ImageErrorListener onError }) {
  ImageStreamCompleter result = _pendingImages[key]?.completer;
  if (result != null)
    return result;
  final _CachedImage image = _cache.remove(key);
  if (image != null) {
    _cache[key] = image;
    return image.completer;
  }
  try {
    result = loader();
  } catch (error, stackTrace) {
    if (onError != null) {
      onError(error, stackTrace);
      return null;
    } else {
      rethrow;
    }
  }
  void listener(ImageInfo info, bool syncCall) {
    // Images that fail to load don't contribute to cache size.
    final int imageSize = info?.image == null ? 0 : info.image.height * info.image.width * 4;
    final _CachedImage image = _CachedImage(result, imageSize);
    // If the image is bigger than the maximum cache size, and the cache size
    // some change.
    if (maximumSizeBytes > 0 && imageSize > maximumSizeBytes) {
      _maximumSizeBytes = imageSize + 1000;
    }
    _currentSizeBytes += imageSize;
    final _PendingImage pendingImage = _pendingImages.remove(key);
    if (pendingImage != null) {
      pendingImage.removeListener();
    }

    _cache[key] = image;
    _checkCacheSize();
  }
  if (maximumSize > 0 && maximumSizeBytes > 0) {
    _pendingImages[key] = _PendingImage(result, listener);
    result.addListener(listener);
  }
  return result;
```

发现没有，这里和我们理解上的 Cache 概念稍微有点不同，以前我们缓存的一般是 key - bitmap 对象，也就是实际绘制数据，**而在 Flutter 中，缓存的仅是 `ImageStreamCompleter` 对象，而不是实际绘制对象 `dart:ui.Image`** 。

3、ImageStreamCompleter

`ImageStreamCompleter` 是一个抽象对象，它主要是用于**管理和通知 `ImageStream` ，处理图片数据后得到的包含有 `dart:ui.Image` 的对象 ImageInfo** 。

接下来我们看 `NetworkImage` 中的 `ImageStreamCompleter` 实现类 `MultiFrameImageStreamCompleter` 。如下图代码所示， `MultiFrameImageStreamCompleter` 主要通过 `codec` 参数获得渲染数据，而这个数据来源通过 `_loadAsync` 方法得到，该方法主要通过 **http 下载图片后，对图片数据通过 `PaintingBinding` 进行 `ImageCodec` 编码处理，将图片转化为引擎可绘制数据。**

```
@override
ImageStreamCompleter load(NetworkImage key) {
  return MultiFrameImageStreamCompleter(
    codec: _loadAsync(key),
    scale: key.scale,
    informationCollector: (StringBuffer information) {
      information.writeln('Image provider: $this');
      information.write('Image key: $key');
    },
  );
}

static final HttpClient _httpClient = HttpClient();

Future<ui.Codec> _loadAsync(NetworkImage key) async {
  assert(key == this);

  final Uri resolved = Uri.base.resolve(key.url);
  final HttpClientRequest request = await _httpClient.getUrl(resolved);
  headers?.forEach((String name, String value) {
    request.headers.add(name, value);
  });
  final HttpClientResponse response = await request.close();
  if (response.statusCode != HttpStatus.ok)
    throw Exception('HTTP request failed, statusCode: ${response?.statusCode}, $resolved');

  final Uint8List bytes = await consolidateHttpClientResponseBytes(response);
  if (bytes.lengthInBytes == 0)
    throw Exception('NetworkImage is an empty file: $resolved');

  return PaintingBinding.instance.instantiateImageCodec(bytes);
}
```

而在 `MultiFrameImageStreamCompleter` 内部， `ui.Codec` 会被 `ui.Image` ，通过 `ImageInfo` 封装起来，并逐步往回回调到 `_ImageState` 中，然后通过 `setState` 将数据传递到 `RenderImage` 内部去绘制。

```
Future<void> _decodeNextFrameAndSchedule() async {
  try {
    _nextFrame = await _codec.getNextFrame();
  } catch (exception, stack) {
    reportError(
      context: 'resolving an image frame',
      exception: exception,
      stack: stack,
      informationCollector: _informationCollector,
      silent: true,
    );
    return;
  }
  if (_codec.frameCount == 1) {
    // This is not an animated image, just return it and don't schedule more
    // frames.
    _emitFrame(ImageInfo(image: _nextFrame.image, scale: _scale));
    return;
  }
  _scheduleAppFrame();
}
```

怎么样，现在再回过头去看开头的流程图，有没有一切明了的感觉？

二、本地图片缓存

通过上方流程的了解，我们知道 Flutter 实现了图片的内存缓存，但是并没有实现图片的本地缓存，所以我们入手的点，应该从 `ImageProvider` 开始。

通过上面对 `NetworkImage` 的分析，我们知道图片是在 `_loadAsync` 方法通过 http 下载的，所以最简单的就是，我们从 `NetworkImage` cv 一份代码，修改 `_loadAsync` 支持 http 下载前读取本地缓存，下载后通过将数据保存在本地。

结合 `flutter_cache_manager` 插件，如下方代码所示，就可以快速简单实现图片的本地缓存：

```
  Future<ui.Codec> _loadAsync(NetworkImage key) async {
     assert(key == this);

     /// add this start
     /// flutter_cache_manager DefaultCacheManager
     final fileInfo = await DefaultCacheManager().getFileFromCache(key.url);
     if(fileInfo != null && fileInfo.file != null) {
       final Uint8List cacheBytes = await fileInfo.file.readAsBytes();
       if (cacheBytes != null) {
         return PaintingBinding.instance.instantiateImageCodec(cacheBytes);
       }
     }
     /// add this end

     final Uri resolved = Uri.base.resolve(key.url);
     final HttpClientRequest request = await _httpClient.getUrl(resolved);
     headers?.forEach((String name, String value) {
       request.headers.add(name, value);
     });
     final HttpClientResponse response = await request.close();
     if (response.statusCode != HttpStatus.ok)
       throw Exception('HTTP request failed, statusCode: ${response?.statusCode}, $r
esolved');

     final Uint8List bytes = await consolidateHttpClientResponseBytes(response);
     if (bytes.lengthInBytes == 0)
       throw Exception('NetworkImage is an empty file: $resolved');

     /// add this start
     await DefaultCacheManager().putFile(key.url, bytes);
     /// add this edn

     return PaintingBinding.instance.instantiateImageCodec(bytes);
   }
```

三、其他补充

1、缓存数量

在闲鱼关于 Flutter 线上应用的内存分析文章中，有过对图片加载对内存问题的详细分析，其中就有一个是 `ImageCache` 的问题。

上面的流程我们知道， `ImageCache` 缓存的是一个异步对象，缓存异步加载对象的一个问题是，在图片加载解码完成之前，你无法知道到底将要消耗多少内存，并且大量的图片加载，会导致的解码任务需要产生大量的IO。

而在 Flutter 中， `ImageCache` 默认的缓存大小是

```
const int _kDefaultSize = 1000;
const int _kDefaultSizeBytes = 100 << 20; // 100
```

所以简单粗暴的做法是： `PaintingBinding.instance.imageCache.maximumSize = 100;` 同时在页面不可见时暂停图片的加载等。

2、.9图

在 Image中，可以通过 `centerSlice` 配置参数设置.9图效果哦。

> 自此，第十篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

作为系列文章的第十一篇，本篇将非常全面带你了解 Flutter 中最关键的设计之一，深入原理帮助你理解 Stream 全家桶，这也许是目前 Flutter 中最全面的 Stream 分析了。

文章汇总地址：

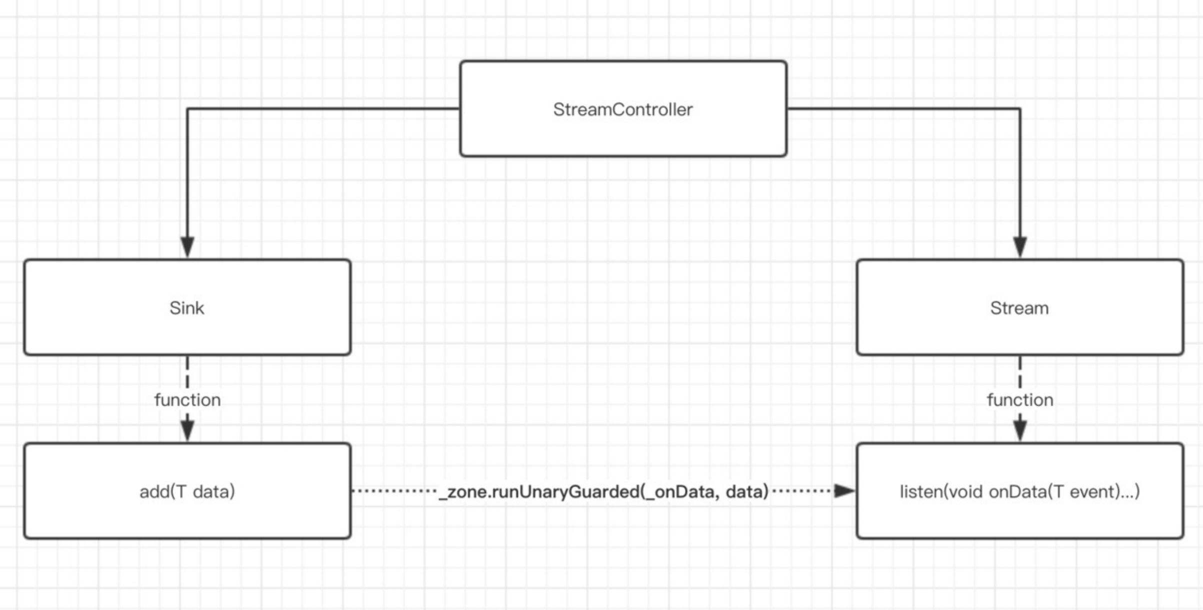
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

一、Stream 由浅入深

`Stream` 在 Flutter 是属于非常关键的概念，在 Flutter 中，状态管理除了 `InheritedWidget` 之外，无论 `rxdart` ，`Bloc` 模式，`flutter_redux` ，`fish_redux` 都离不开 `Stream` 的封装，而事实上 `Stream` 并不是 Flutter 中特有的，而是 Dart 中自带的逻辑。

通俗来说，`Stream` 就是事件流或者管道，事件流相信大家并不陌生，简单的说就是：**基于事件流驱动设计代码，然后监听订阅事件，并针对事件变换处理响应。**

而在 Flutter 中，整个 `Stream` 设计外部暴露的对象主要如下图，主要包含了 `StreamController` 、`Sink` 、`Stream` 、`StreamSubscription` 四个对象。



1、Stream 的简单使用

如下代码所示，`Stream` 的使用并不复杂，一般我们只需要：

- 创建 `StreamController` ，
- 然后获取 `StreamSink` 用做事件入口，
- 获取 `Stream` 对象用于监听，
- 并且通过监听得到 `StreamSubscription` 管理事件订阅，最后在不需要时关闭即可，看起来是不是很简单？

```
class DataBloc {
  ///定义一个Controller
  StreamController<List<String>> _dataController = StreamController<List<String>>()
;
  ///获取 StreamSink 做 add 入口
  StreamSink<List<String>> get _dataSink => _dataController.sink;
  ///获取 Stream 用于监听
  Stream<List<String>> get _dataStream => _dataController.stream;
  ///事件订阅对象
  StreamSubscription _dataSubscription;

  init() {
    ///监听事件
    _dataSubscription = _dataStream.listen((value){
      ///do change
    });
    ///改变事件
    _dataSink.add(["first", "second", "three", "more"]);

  }

  close() {
    ///关闭
    _dataSubscription.cancel();
    _dataController.close();
  }
}
```

在设置好监听后，之后每次有事件变化时， `listen` 内的方法就会被调用，同时你还可以通过操作符对 `Stream` 进行变换处理。

如下代码所示，是不是一股 `rx` 风扑面而来?

```
_dataStream.where(test).map(convert).transform(streamTransformer).listen(onData);
```

而在 Flutter 中， 最后结合 `StreamBuilder` ，就可以完成 **基于事件流的异步状态控件** 了！

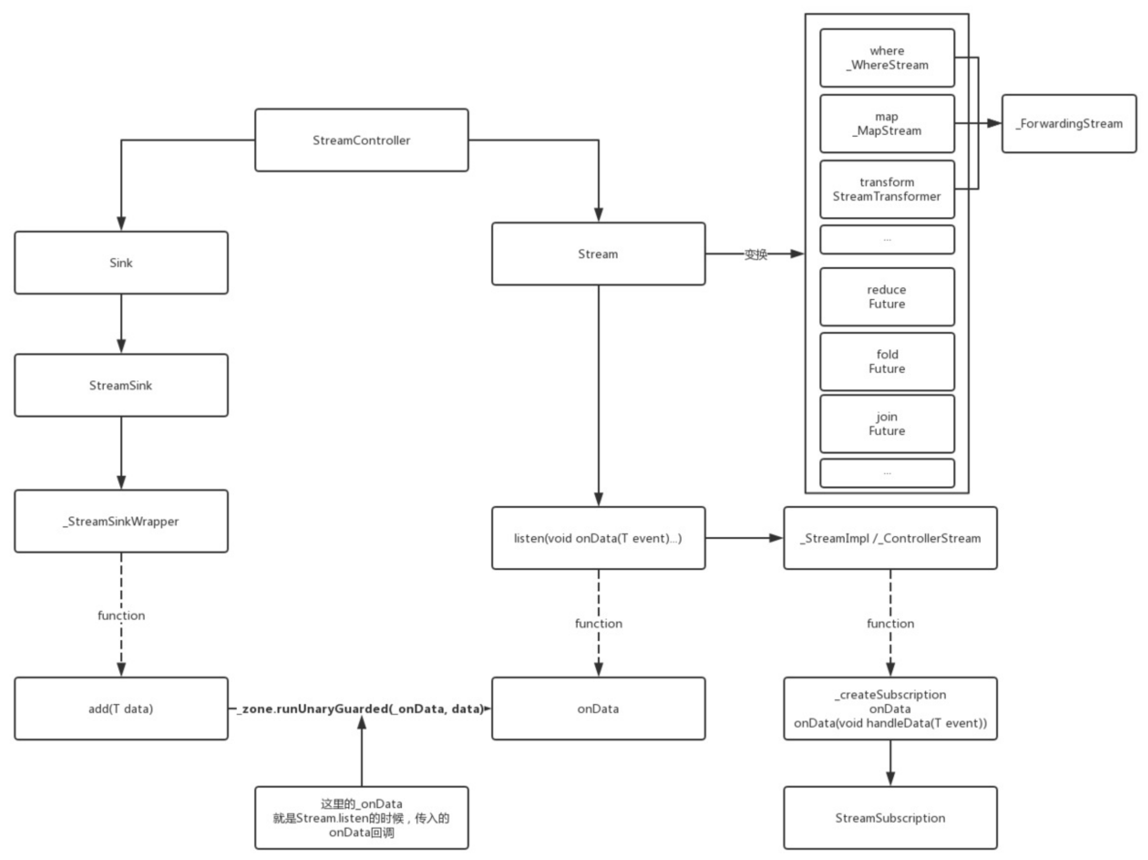
```
StreamBuilder<List<String>>(
    stream: dataStream,
    initialData: ["none"],
    ///这里的 snapshot 是数据快照的意思
    builder: (BuildContext context, AsyncSnapshot<List<String>> snapshot) {
      ///获取到数据，为所欲为的更新 UI
      var data = snapshot.data;
      return Container();
    });
```

那么问题来了，**它们内部究竟是如果实现的呢？原理是什么？各自的作用是什么？都有哪些特性呢？后面我们将开始深入解析这个逻辑** 。

2、Stream 四天王

从上面我们知道，在 Flutter 中使用 `Stream` 主要有四个对象，那么这四个对象是如何“勾搭”在一起的？他们各自又担任什么责职呢？

首先如下图，我们可以从进阶版的流程图上看出 整个 `Stream` 的内部工作流程。



Flutter中 `Stream` 、 `StreamController` 、 `StreamSink` 和 `StreamSubscription` 都是 `abstract` 对象，他们对外抽象出接口，而内部实现对象大部分都是 `_` 开头的如 `_SyncStreamController` 、 `ControllerStream` 等私有类，在这基础上整个流程概括起来就是：

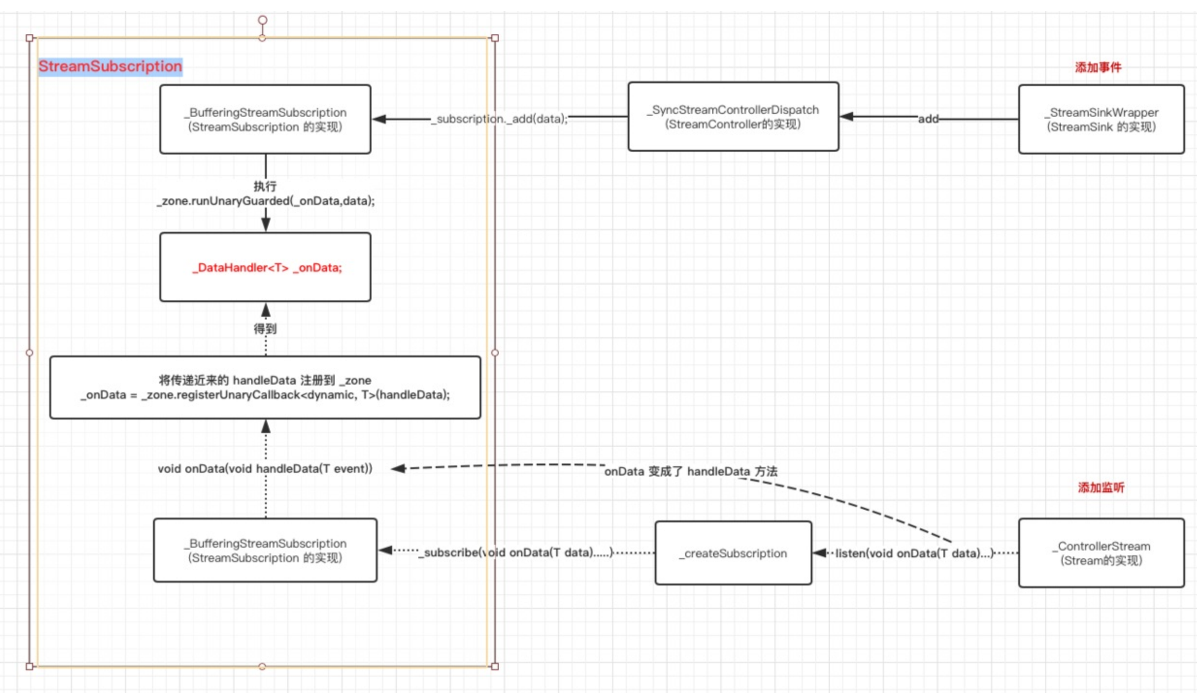
有一个事件源叫 `Stream` ，为了方便控制 `Stream` ，官方提供了使用 `StreamController` 作为管理；同时它对外提供了 `StreamSink` 对象作为事件输入口，可通过 `sink` 属性访问; 又提供 `stream` 属性提供 `Stream` 对象的监听和变换，最后得到的 `StreamSubscription` 可以管理事件的订阅。

所以我们可以总结出：

- StreamController ：如类名描述，用于整个 `Stream` 过程的控制，提供各类接口用于创建各种事件流。
- StreamSink：一般作为事件的入口，提供如 `add` ， `addStream` 等。

- Stream：事件源本身，一般可用于监听事件或者对事件进行转换，如 `listen` 、 `where` 。
- StreamSubscription：事件订阅后的对象，表面上用于管理订阅过等各类操作，如 `cacenl` 、 `pause` ，同时在内部也是事件的中转关键。

回到 `Stream` 的工作流程上，在上图中我们知道， 通过 `StreamSink.add` 添加一个事件时， 事件最后会回调到 `listen` 中的 `onData` 方法，这个过程是通过 `zone.runUnaryGuarded` 执行的，这里 `zone.runUnaryGuarded` 是什么作用后面再说，我们需要知道这个 `onData` 是怎么来的？



如上图，通过源码我们知道：

- 1、 `Stream` 在 `listen` 的时候传入了 `onData` 回调，这个回调会传入到 `StreamSubscription` 中，之后通过 `zone.registerUnaryCallback` 注册得到 `_onData` 对象 (不是前面的 *`onData`* 回调哦)。
- 2、 `StreamSink` 在添加事件是，会执行到 `StreamSubscription` 中的 `_sendData` 方法，然后通过 `_zone.runUnaryGuarded(_onData, data);` 执行 1 中得到的 `_onData` 对象，触发 `listen` 时传入的回调方法。

可以看出整个流程都是和 `StreamSubscription` 相关的，现在我们已经知道从 **事件入口到事件出口** 的整个流程时怎么运作的，那么这个过程是**怎么异步执行的呢？其中频繁出现的 `zone` 是什么？

3、线程

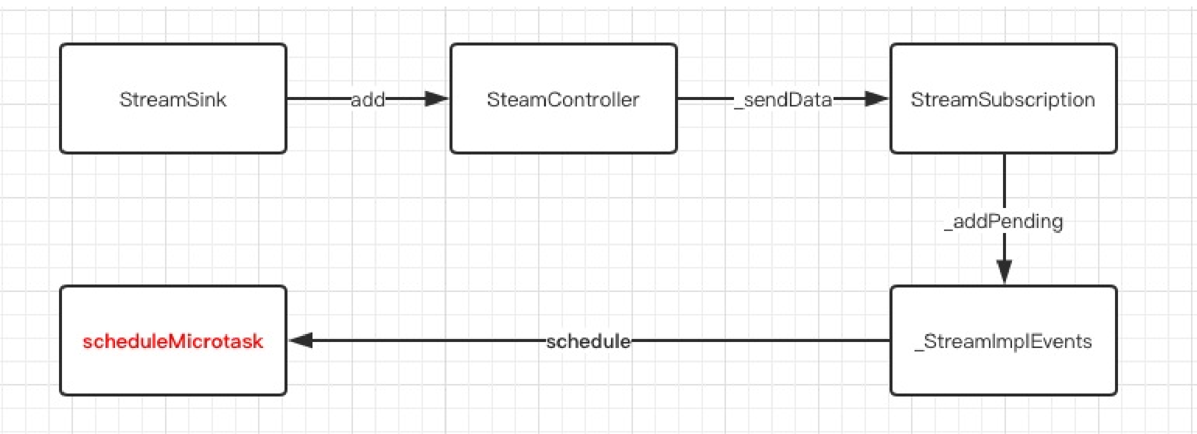
首先我们需要知道，Stream 是怎么实现异步的？

这就需要说到 Dart 中的异步实现逻辑了，因为 Dart 是 **单线程应用** ，和大多数单线程应用一样，Dart 是以 **消息循环机制** 来运行的，而这里面主要包含两个任务队列，一个是 **microtask** 内部队列，一个是 **event** 外部队列，而 *microtask* 的优先级又高于 *event* 。

默认的在 Dart 中，如 点击、滑动、*IO*、绘制事件 等事件都属于 **event** 外部队列，**microtask** 内部队列主要是由 Dart 内部产生，而 `Stream` 中的执行异步的模式就是 `scheduleMicrotask` 了。

> 因为 *microtask* 的优先级又高于 *event* ，所以如果 *microtask* 太多就可能会对触摸、绘制等外部事件造成阻塞卡顿哦。

如下图，就是 Stream 内部在执行异步操作过程执行流程：



4、Zone

那么 `Zone` 又是什么？它是哪里来的？

在上一篇章中说过，因为 Dart 中 `Future` 之类的异步操作是无法被当前代码 `try/cacth` 的，而在 Dart 中你可以给执行对象指定一个 `Zone` ，类似提供一个**沙箱环境** ，而在这个沙箱内，你就可以全部可以捕获、拦截或修改一些代码行为，比如所有未被处理的异常。

那么项目中默认的 `Zone` 是怎么来的？在 Flutter 中，**Dart 中的 `Zone` 启动是在 `_runMainZoned` 方法** ，如下代码所示 `_runMainZoned` 的 `@pragma("vm:entry-point")` 注解表示该方式是给 Engine 调用的，到这里我们知道了 `Zone` 是怎么来的了。

```
///Dart 中

@pragma('vm:entry-point')
// ignore: unused_element
void _runMainZoned(Function startMainIsolateFunction, Function userMainFunction) {
  startMainIsolateFunction((){
    runZoned<Future<void>>(····);
  }, null);
}

///C++ 中
if (tonic::LogIfError(tonic::DartInvokeField(
          Dart_LookupLibrary(tonic::ToDart("dart:ui")), "_runMainZoned",
          {start_main_isolate_function, user_entrypoint_function}))) {
    FML_LOG(ERROR) << "Could not invoke the main entrypoint.";
    return false;
}
```

那么 `zone.runUnaryGuarded` 的作用是什么？相较于 `scheduleMicrotask` 的异步操作，官方的解释是：**在此区域中使用参数执行给定操作并捕获同步错误。** 类似的还有 `runUnary` 、 `runBinaryGuarded` 等，所以我们知道前面提到的 `zone.runUnaryGuarded` 就是 **Flutter 在运行的这个 zone 里执行已经注册的 `_onData` ，并捕获异常。**

5、异步和同步

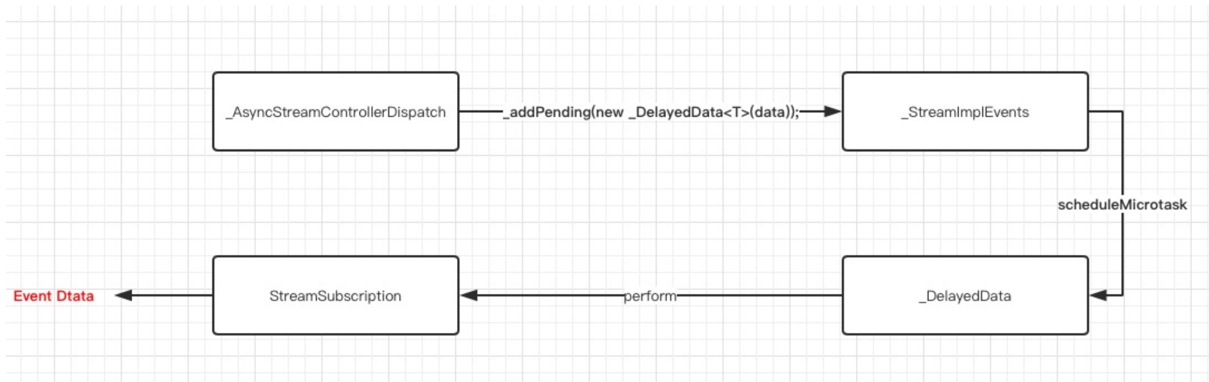
前面我们说了 `Stream` 的内部执行流程，那么同步和异步操作时又有什么区别？具体实现时怎么样的呢？

我们以默认 `Stream` 流程为例子， `StreamController` 的工厂创建可以通过 `sync` 指定同步还是异步，默认是异步模式的。 而无论异步还是同步，他们都是继承了 `_StreamController` 对象，区别还是在于 `mixins` 的是哪个 **`_EventDispatch`** 实现：

- `_AsyncStreamControllerDispatch`
- `_SyncStreamControllerDispatch`

上面这两个 `_EventDispatch` 最大的不同就是在调用 `sendData` 提交事件时，是直接调用 `StreamSubscription` 的 `_add` 方法，还是调用 `_addPending(new _DelayedData<T>(data));` 方法的区别。

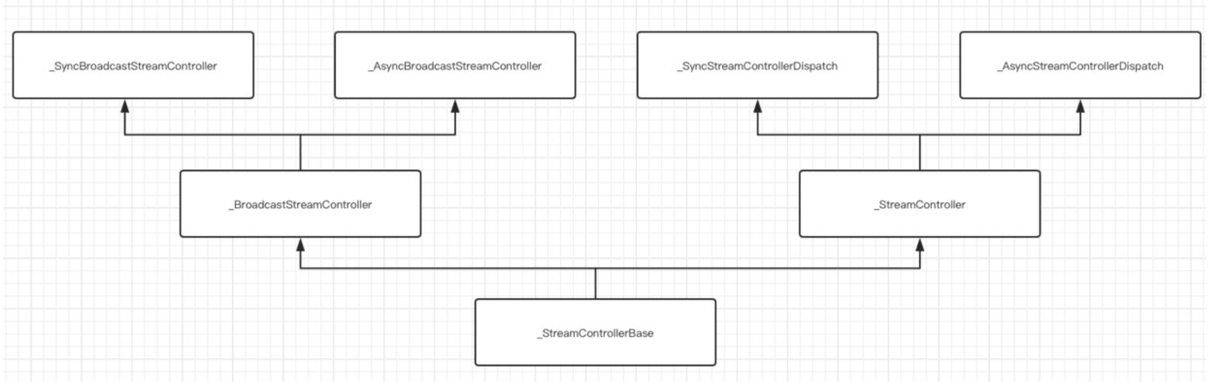
如下图， **异步执行的逻辑就是上面说过的 `scheduleMicrotask` ，在 `_StreamImplEvents` 中 `scheduleMicrotask` 执行后，会调用 `_DelayedData` 的 `perform` ，最后通过 `_sendData` 触发 `StreamSubscription` 去回调数据 。**



6、广播和非广播。

在 `Stream` 中又非为广播和非广播模式，如果是广播模式中， `StreamControlle` 的实现是由如下所示实现的，他们的基础关系如下图所示：

- `_SyncBroadcastStreamController`
- `_AsyncBroadcastStreamController`



广播和非广播的区别在于调用 `_createSubscription` 时，内部对接口类 `_StreamControllerLifecycle` 的实现，同时它们的差异在于：

- 在 `_StreamController` 里判断了如果 `Stream` 是 `_isInitialState` 的，也就是订阅过的，就直接报错 *"Stream has already been listened to."*，只有未订阅的才创建 `StreamSubscription` 。
- 在 `_BroadcastStreamController` 中，`_isInitialState` 的判断被去掉了，取而代之的是 `isClosed` 判断，并且在广播中，`_sendData` 是一个 `forEach` 执行：

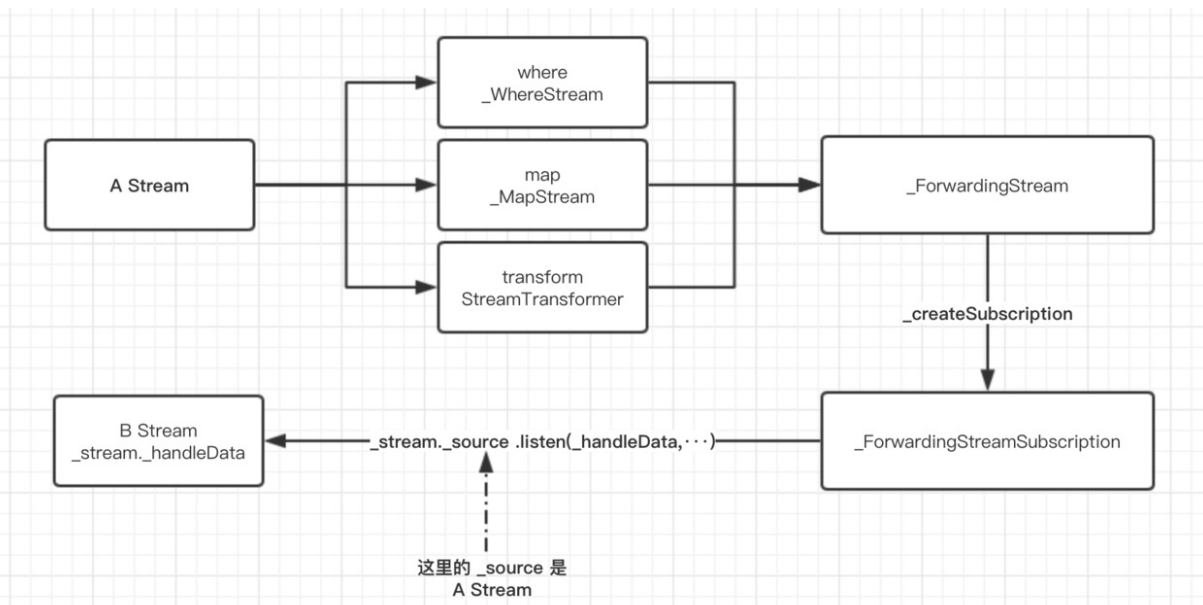
```
_forEachListener((_BufferingStreamSubscription<T> subscription) {
    subscription._add(data);
  });
```

7、Stream 变换

`Stream` 是支持变换处理的，针对 `Stream` 我们可以经过多次变化来得到我们需要的结果。那么这些变化是怎么实现的呢?

如下图所示，一般操作符变换的 `Stream` 实现类，都是继承了 `_ForwardingStream` ，在它的内部的 `_ForwardingStreamSubscription` 里，会通过上一个 `Pre A Stream` 的 `listen` 添加 `_handleData` 回调，之后在回调里再次调用新的 `Current B Stream` 的 `_handleData` 。

所以事件变化的本质就是，**变换都是对 `Stream` 的 `listen` 嵌套调用组成的。**

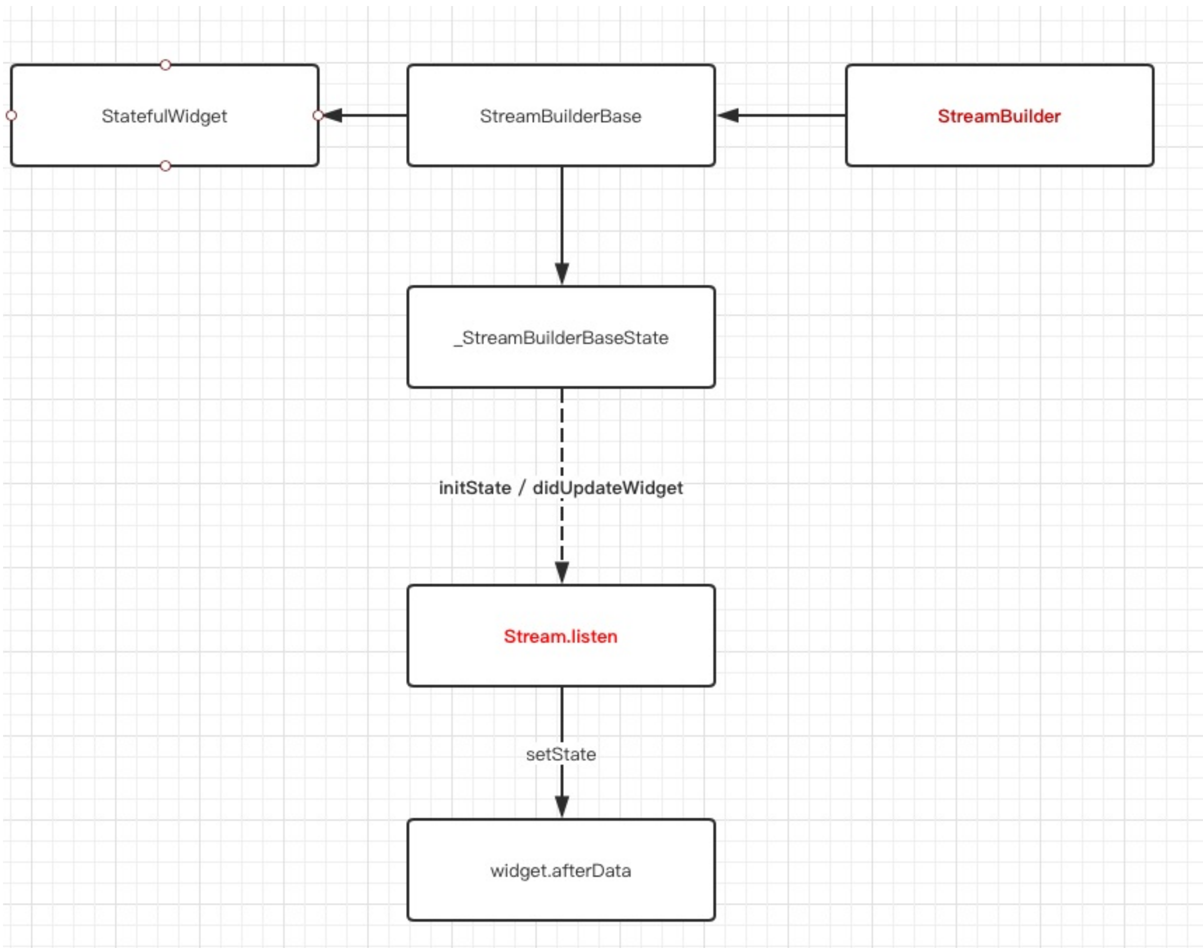


同时 `Stream` 还有转换为 `Future` , 如 `firstWhere` 、 `elementAt` 、 `reduce` 等操作符方法, 基本都是创建一个内部 `_Future` 实例，然后再 `listen` 的回调用调用 `Future` 方法返回。

二、StreamBuilder

如下代码所示, 在 Flutter 中通过 `StreamBuilder` 构建 Widget , 只需提供一个 `Stream` 实例即可, 其中 `AsyncSnapshot` 对象为数据快照，通过 `data` 缓存了当前数据和状态，那 `StreamBuilder` 是如何与 `Stream` 关联起来的呢?

```
StreamBuilder<List<String>>(
    stream: dataStream,
    initialData: ["none"],
    ///这里的 snapshot 是数据快照的意思
    builder: (BuildContext context, AsyncSnapshot<List<String>> snapshot) {
      ///获取到数据，为所欲为的更新 UI
      var data = snapshot.data;
      return Container();
    });
```



如上图所示， `StreamBuilder` 的调用逻辑主要在 `_StreamBuilderBaseState` 中， `_StreamBuilderBaseState` 在 `initState` 、 `didUpdateWidget` 中会调用 `_subscribe` 方法，从而调用 `Stream` 的 `listen` ，然后通过 `setState` 更新UI，就是这么简单有木有？

> 我们常用的 `setState` 中其实是调用了 `markNeedsBuild` ， `markNeedsBuild` 内部标记 `element` 为 `diry` ，然后在下一帧 `WidgetsBinding.drawFrame` 才会被绘制，这可以看出 `setState` 并不是立即生效的哦。

三、rxdart

其实无论从订阅或者变换都可以看出， Dart 中的 `Stream` 已经自带了类似 `rx` 的效果，但是为了让 `rx` 的用户们更方便的使用，ReactiveX 就封装了 `rxdart` 来满足用户的熟悉感，如下图所示为它们的对应关系：

Dart	RxDart
StreamController	Subject
Stream	Observable

在 `rxdart` 中， `Observable` 是一个 `Stream` ，而 `Subject` 继承了 `Observable` 也是一个 `Stream` ，并且 `Subject` 实现了 `StreamController` 的接口，所以它也具有 Controller 的作用。

如下代码所示是 `rxdart` 的简单使用，可以看出它屏蔽了外界需要对 `StreamSubscription` 和 `StreamSink` 等的认知，更符合 `rx` 历史用户的理解。

```
final subject = PublishSubject<String>();

subject.stream.listen(observerA);
subject.add("AAAA1");
subject.add("AAAA2"));

subject.stream.listen(observeB);
subject.add("BBBB1");
subject.close();
```

这里我们简单分析下，以上方代码为例，

- `PublishSubject` 内部实际创建是创建了一个广播 `StreamController<T>.broadcast` 。
- 当我们调用 `add` 或者 `addStream` 时，最终会调用到的还是我们创建的 `StreamController.add` 。
- 当我们调用 `onListen` 时，也是将回调设置到 `StreamController` 中。
- `rxdart` 在做变换时，我们获取到的 `Observable` 就是 this，也就是 `PublishSubject` 自身这个 `Stream` ，而 `Observable` 一系列的变换，也是基于创建时传入的 `stream` 对象，比如：

```
@override
Observable<S> asyncMap<S>(FutureOr<S> convert(T value)) =>
    Observable<S>(_stream.asyncMap(convert));
```

所以我们可以看出来， `rxdart` 只是对 `Stream` 进行了概念变换，变成了我们熟悉的对象和操作符，而这也是为什么 `rxdart` 可以在 `StreamBuilder` 中直接使用的原因。

所以，到这里你对 Flutter 中 Stream 有全面的理解了没?

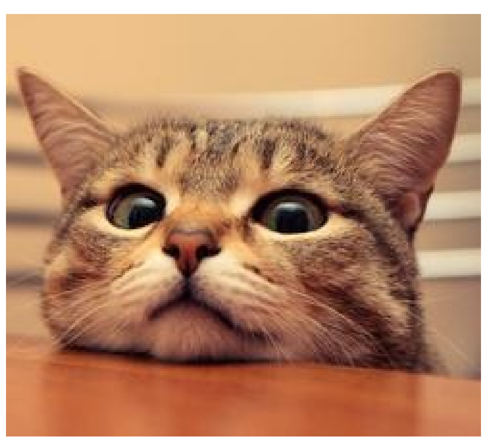
> 自此，第十一篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型习项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex



作为系列文章的第十二篇，本篇将通过 scope_model 、 BloC 设计模式、flutter_redux 、 fish_redux 来全面深入分析， Flutter 中大家最为关心的状态管理机制，理解各大框架中如何设计实现状态管理，从而选出你最为合适的 state “大管家”。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

在所有 **响应式编程** 中，状态管理一直老生常谈的话题，而在 Flutter 中，目前主流的有 `scope_model` 、 `BloC 设计模式` 、 `flutter_redux` 、 `fish_redux` 等四种设计，它们的 复杂度 和 上手难度 是逐步递增的，但同时 **可拓展性** 、**解耦度** 和 **复用能力** 也逐步提升。

基于前篇，我们对 `Stream` 已经有了全面深入的理解，后面可以发现这四大框架或多或少都有 `Stream` 的应用，不过还是那句老话，**合适才是最重要，不要为了设计而设计** 。

> 本文Demo源码
>
> GSYGithubFlutter 完整开源项目

一、scoped_model

`scoped_model` 是 Flutter 最为简单的状态管理框架，它充分利用了 Flutter 中的一些特性，只有一个 dart 文件的它，极简的实现了一般场景下的状态管理。

如下方代码所示，利用 `scoped_model` 实现状态管理只需要三步 ：

- 定义 `Model` 的实现，如 `CountModel` ，并且在状态改变时执行 `notifyListeners()` 方法。
- 使用 `ScopedModel` Widget 加载 `Model` 。
- 使用 `ScopedModelDescendant` 或者 `ScopedModel.of<CountModel>(context)` 加载 `Model` 内状态数据。

是不是很简单？那仅仅一个 dart 文件，如何实现这样的效果的呢？后面我们马上开始剥析它。

```
class ScopedPage extends StatelessWidget {
  final CountModel _model = new CountModel();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: new Text("scoped"),
        ),
        body: Container(
          child: new ScopedModel<CountModel>(
            model: _model,
            child: CountWidget(),
          ),
```

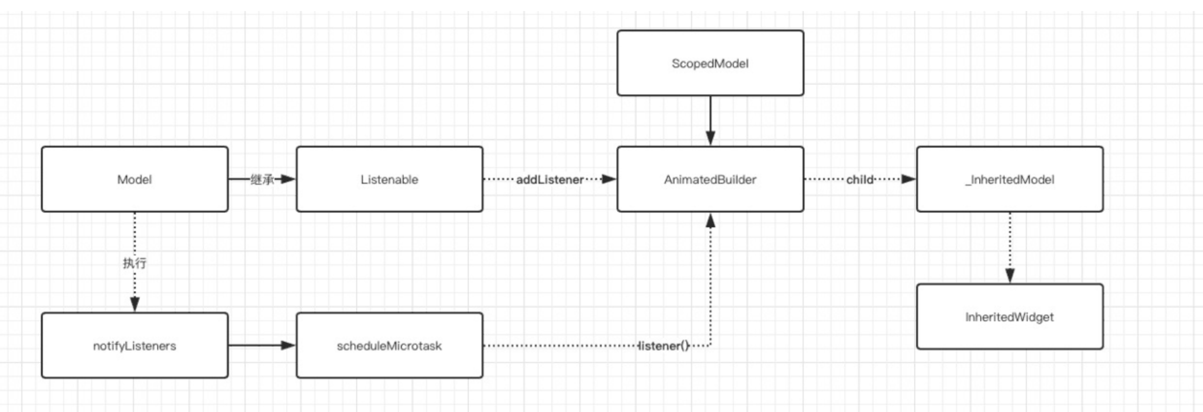
```
        ));
    }
  }

  class CountWidget extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return new ScopedModelDescendant<CountModel>(
          builder: (context, child, model) {
          return new Column(
            children: <Widget>[
              new Expanded(child: new Center(child: new Text(model.count.toString()))
  ),
              new Center(
                child: new FlatButton(
                    onPressed: () {
                      model.add();
                    },
                    color: Colors.blue,
                    child: new Text("+")),
              ),
            ],
          );
        });
    }
  }

  class CountModel extends Model {
    static CountModel of(BuildContext context) =>
        ScopedModel.of<CountModel>(context);

    int _count = 0;

    int get count => _count;

    void add() {
      _count++;
      notifyListeners();
    }
  }
```

如下图所示，在 `scoped_model` 的整个实现流程中， `ScopedModel` 这个 Widget 很巧妙的借助了 `AnimatedBuildler` 。

因为 `AnimatedBuildler` 继承了 `AnimatedWidget` ，在 `AnimatedWidget` 的生命周期中会对 `Listenable` 接口添加监听，而 `Model` 恰好就实现了 `Listenable` 接口，整个流程总结起来就是：

- `Model` 实现了 `Listenable` 接口，内部维护一个 `Set<VoidCallback> _listeners` 。
- 当 `Model` 设置给 `AnimatedBuildler` 时， `Listenable` 的 `addListener` 会被调用，然后

添加一个 `_handleChange` 监听到 `_listeners` 这个 Set 中。
- 当 `Model` 调用 `notifyListeners` 时，会通过异步方法 `scheduleMicrotask` 去从头到尾执行一遍 `_listeners` 中的 `_handleChange` 。
- `_handleChange` 监听被调用，执行了 `setState({})` 。



整个流程是不是很巧妙，机制的利用了 `AnimatedWidget` 和 `Listenable` 在 Flutter 中的特性组合，至于 `ScopedModelDescendant` 就只是为了跨 Widget 共享 `Model` 而做的一层封装，主要还是通过 `ScopedModel.of<CountModel>(context)` 获取到对应 Model 对象，这这个实现上， `scoped_model` 依旧利用了 Flutter 的特性控件 **`InheritedWidget`** 实现。

InheritedWidget

在 `scoped_model` 中我们可以通过 `ScopedModel.of<CountModel>(context)` 获取我们的 Model ，其中最主要是因为其内部的 build 的时候，包裹了一个 `_InheritedModel` 控件，而它继承了 `InheritedWidget` 。

为什么我们可以通过 `context` 去获取到共享的 `Model` 对象呢?

首先我们知道 `context` 只是接口，而在 Flutter 中 `context` 的实现是 `Element` ，在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement>` `_inheritedWidgets` 的对象。

`_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 Map 会被**一级一级往下传递与合并** 。

所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 时，就可以往上查找到父控件的 Widget，从在 `scoped_model` 获取到 `_InheritedModel` 中的 `Model` 。

二、BloC

`BloC` 全称 *Business Logic Component* ，它属于一种设计模式，在 Flutter 中它主要是通过 `Stream` 与 `SteamBuilder` 来实现设计的，所以 `BloC` 实现起来也相对简单，关于 `Stream` 与 `SteamBuilder` 的实现原理可以查看前篇，这里主要展示如何完成一个简单的 `BloC` 。

如下代码所示，整个流程总结为:

- 定义一个 `PageBloc` 对象，利用 `StreamController` 创建 `Sink` 与 `Stream` 。
- `PageBloc` 对外暴露 `Stream` 用来与 `StreamBuilder` 结合；暴露 add 方法提供外部调用，内部通过 `Sink` 更新 `Stream` 。
- 利用 `StreamBuilder` 加载监听 `Stream` 数据流，通过 snapShot 中的 data 更新控件。

当然，如果和 `rxdart` 结合可以简化 `StreamController` 的一些操作，同时如果你需要利用 `BloC` 模式实现状态共享，那么自己也可以封装多一层 `InheritedWidgets` 的嵌套，如果对于这一块有疑惑的话，推荐可以去看看上一篇的 Stream 解析。

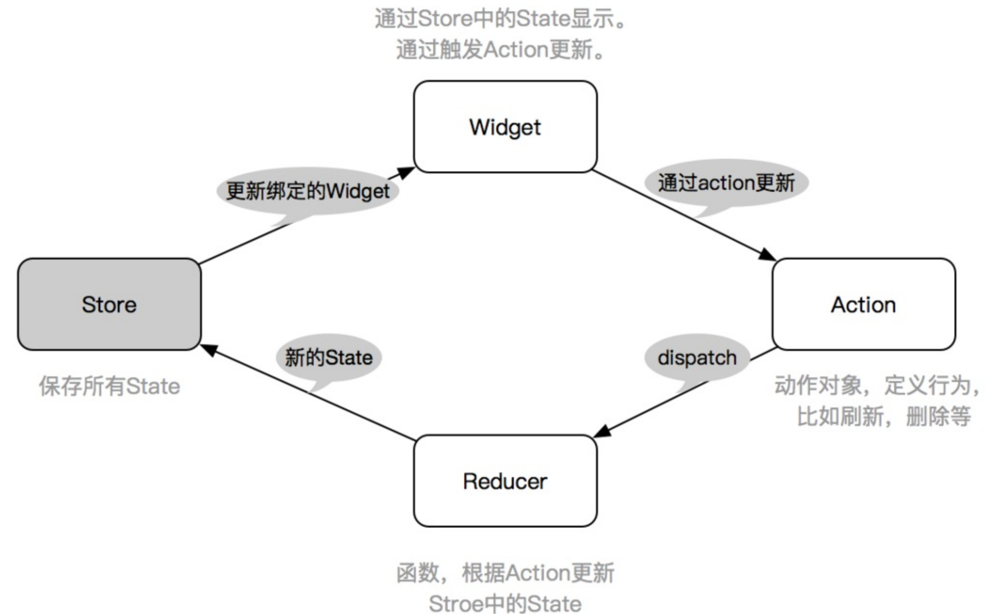
```
class _BlocPageState extends State<BlocPage> {
  final PageBloc _pageBloc = new PageBloc();
  @override
  void dispose() {
    _pageBloc.dispose();
    super.dispose();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        child: new StreamBuilder(
            initialData: 0,
            stream: _pageBloc.stream,
            builder: (context, snapShot) {
              return new Column(
                children: <Widget>[
                  new Expanded(
                      child: new Center(
                          child: new Text(snapShot.data.toString()))),
                  new Center(
                    child: new FlatButton(
                        onPressed: () {
                          _pageBloc.add();
                        },
                        color: Colors.blue,
                        child: new Text("+")),
                  ),
                  new SizedBox(
                    height: 100,
                  )
                ],
              );
            }),
      ),
    );
  }
}
class PageBloc {
  int _count = 0;
```

```
  ///StreamController
  StreamController<int> _countController = StreamController<int>();
  ///对外提供入口
  StreamSink<int> get _countSink => _countController.sink;
  ///提供 stream StreamBuilder 订阅
  Stream<int> get stream => _countController.stream;
  void dispose() {
    _countController.close();
  }
  void add() {
    _count++;
    _countSink.add(_count);
  }
}
```

三、flutter_redux

相信如果是前端开发者，对于 `redux` 模式并不会陌生，而 `flutter_redux` 可以看做是利用了 `Stream` 特性的 `scope_model` 升级版，通过 `redux` 设计模式来完成解耦和拓展。

当然，**更多的功能和更好的拓展性，也造成了代码的复杂度和上手难度**，因为 `flutter_redux` 的代码使用篇幅问题，这里就不展示所有代码了，需要看使用代码的可直接从 demo 获取，现在我们直接看 `flutter_redux` 是如何实现状态管理的吧。



如上图，我们知道 `redux` 中一般有 `Store` 、 `Action` 、 `Reducer` 三个主要对象，之外还有 `Middleware` 中间件用于拦截，所以如下代码所示：

- 创建 `Store` 用于管理状态 。
- 给 `Store` 增加 `appReducer` 合集方法，增加需要拦截的 `middleware` ，并初始化状态。
- 将 `Store` 设置给 `StoreProvider` 这个 `InheritedWidget` 。
- 通过 `StoreConnector` / `StoreBuilder` 加载显示 `Store` 中的数据。

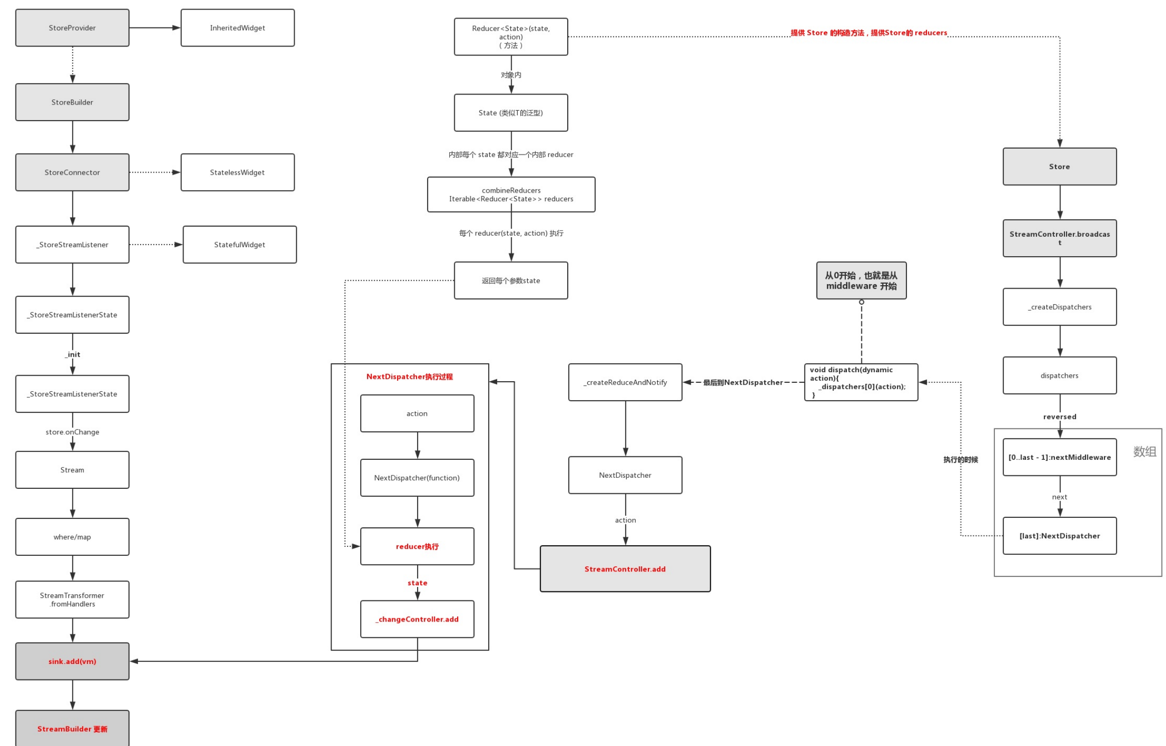
之后我们可以 `dispatch` 一个 **Action** ，在经过 `middleware` 之后，触发对应的 **Reducer** 返回数据，而事实上这里核心的内容实现，**还是 `Stream` 和 `StreamBuilder` 的结合使用** ，接下来就让我们看看这个流程是如何联动起来的吧。

```
class _ReduxPageState extends State<ReduxPage> {

  ///初始化store
  final store = new Store<CountState>(
    /// reducer 合集方法
    appReducer,
    ///中间键
    middleware: middleware,
    ///初始化状态
    initialState: new CountState(count: 0),
  );

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: new Text("redux"),
        ),
        body: Container(
          /// StoreProvider InheritedWidget
          /// 加载 store 共享
          child: new StoreProvider(
            store: store,
            child: CountWidget(),
          ),
        ));
  }
}
```

如下图所示，是 `flutter_redux` 从入口到更新的完整流程图，整理这个流程其中最关键有几个点是：

- `StoreProvider` 是 `InheritedWidgets` ，所以它可以通过 `context` 实现状态共享。
- `StreamBuilder` / `StoreConnector` 的内部实现主要是 `StreamBuilder` 。
- `Store` 内部是通过 `StreamController.broadcast` 创建的 `Stream` ，然后在 `StoreConnector` 中通过 `Stream` 的 `map` 、 `transform` 实现小状态的变换，最后更新到 `StreamBuilder` 。

那么现在看下图流程有点晕？下面我们直接分析图中流程。



可以看出整个流程的核心还是 `Stream` ，基于这几个关键点，我们把上图的流程整理为：

- 1、 `Store` 创建时传入 `reducer` 对象和 `middleware` 数组，同时通过 `StreamController.broadcast` 创建了 `_changeController` 对象。
- 2、 `Store` 利用 `middleware` 和 `_changeController` 组成了一个 `NextDispatcher` **方法数组** ，并把 `_changeController` 所在的 `NextDispatcher` 方法放置在数组**最后位置。**
- 3、 `StoreConnector` 内通过 `Store` 的 `_changeController` 获取 `Stream` ，并进行了一系列变换后，**最终 `Stream` 设置给了 `StreamBuilder`** 。
- 4、当我们调用 `Stroe` 的 `dispatch` 方法时，我们会先进过 `NextDispatcher` 数组中的一系列 `middleware` 拦截器，最终调用到队末的 `_changeController` 所在的 `NextDispatcher` 。
- 5、最后一个 `NextDispatcher` 执行时会先执行 `reducer` 方法获取新的 `state` ，然后通过 `_changeController.add` 将状态加载到 `Stream` 流程中，触发 `StoreConnector` 的 `StreamBuilder` 更新数据。

> 如果对于 `Stream` 流程不熟悉的还请看上篇。

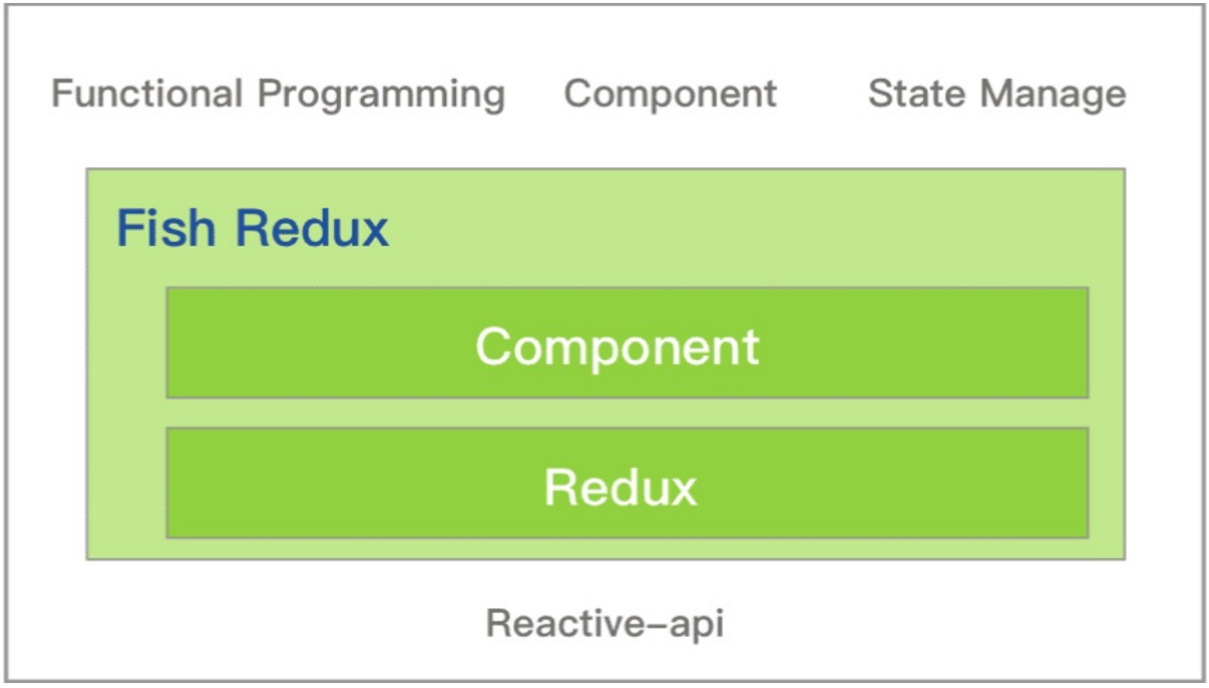
现在再对照流程图会不会清晰很多了?

在 `flutter_redux` 中，开发者的每个操作都只是一个 `Action` ，而这个行为所触发的逻辑完全由 `middleware` 和 `reducer` 决定，这样的设计在一定程度上将业务与UI隔离，同时也统一了状态的管理。

> 比如你一个点击行为只是发出一个 `RefrshAction` ，但是通过 `middleware` 拦截之后，在其中异步处理完几个数据接口，然后重新 `dispatch` 出 `Action1` 、 `Action2` 、 `Action3` 去更新其他页面， 类似的 `redux_epics` 库就是这样实现异步的 `middleware` 逻辑。

四、fish_redux

如果说 `flutter_redux` 属于相对复杂的状态管理设置的话，那么闲鱼开源的 `fish_redux` 可谓 “不走寻常路” 了，虽然是基于 `redux` 原有的设计理念，同时也有使用到 `Stream` ，但是相比较起来整个设计完全是 **超脱三界，如果是前面的都是简单的拼积木，那是 `fish_redux` 就是积木界的乐高。**



因为篇幅原因，这里也只展示部分代码，其中 `reducer` 还是我们熟悉的存在，而闲鱼在这 `redux` 的基础上提出了 `Comoponent` 的概念，这个概念下 `fish_redux` 是从 `Context` 、 `Widget` 等地方就开始全面“入侵”你的代码，从而带来“超级赛亚人”版的 `redux` 。

如下代码所示，默认情况我们需要：

- 继承 `Page` 实现我们的页面。
- 定义好我们的 `State` 状态。
- 定义 `effect` 、 `middleware` 、 `reducer` 用于实现**副作用、中间件、结果返回处理。**
- 定义 `view` 用于绘制页面。
- 定义 `dependencies` 用户装配控件，这里最骚气的莫过于**重载了 + 操作符，然后利用 `Connector` 从 `State` 挑选出数据，然后通过 `Component` 绘制。**

现在看起来使用流程是不是变得复杂了？

但是这带来的好处就是 **复用的颗粒度更细了，装配和功能更加的清晰。** 那这个过程是如何实现的呢？后面我们将分析这个复杂的流程。

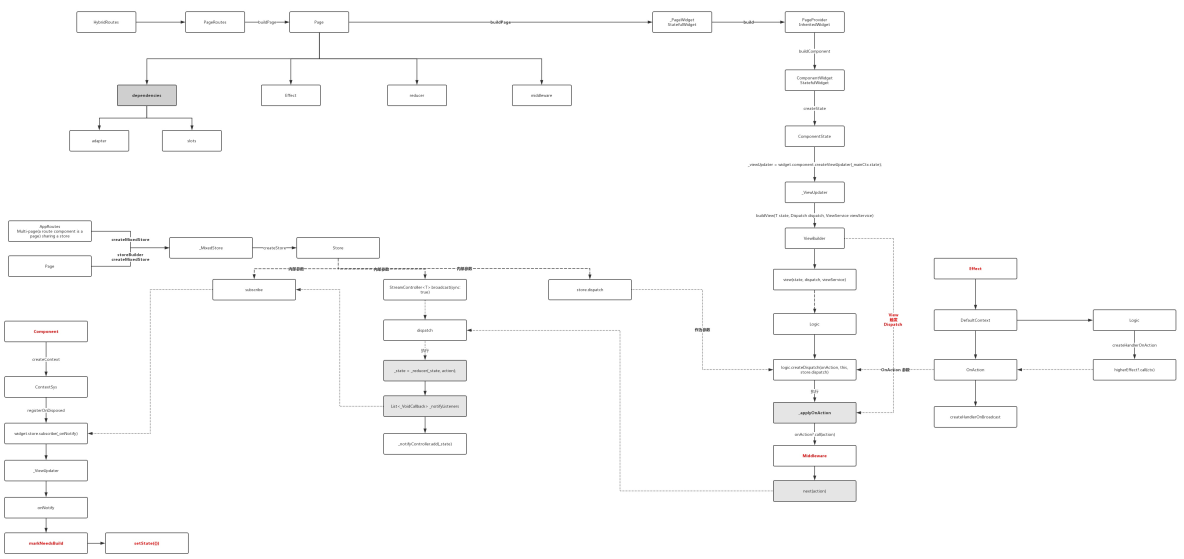
```
class FishPage extends Page<CountState, Map<String, dynamic>> {
  FishPage()
      : super(
          initState: initState,
          effect: buildEffect(),
          reducer: buildReducer(),
```

```
            ///配置 View 显示
            view: buildView,
            ///配置 Dependencies 显示
            dependencies: Dependencies<CountState>(
                slots: <String, Dependent<CountState>>{
                  ///通过 Connector() 从 大 state 转化处小 state
                  ///然后将数据渲染到 Component
                  'count-double': DoubleCountConnector() + DoubleCountComponent()
                }
            ),
            middleware: <Middleware<CountState>>[
              ///中间键打印log
              logMiddleware(tag: 'FishPage'),
            ]
    );
}

///渲染主页
Widget buildView(CountState state, Dispatch dispatch, ViewService viewService) {
  return Scaffold(
      appBar: AppBar(
        title: new Text("fish"),
      ),
      body: new Column(
        children: <Widget>[
          ///viewService 渲染 dependencies
          viewService.buildComponent('count-double'),
          new Expanded(child: new Center(child: new Text(state.count.toString()))),
          new Center(
            child: new FlatButton(
                onPressed: () {
                  ///+
                  dispatch(CountActionCreator.onAddAction());
                },
                color: Colors.blue,
                child: new Text("+")),
          ),
          new SizedBox(
            height: 100,
          )
        ],
      ));
}
```

如下大图所示，整个联动的流程比 `flutter_redux` 复杂了更多（如果看不清可以点击大图），而这个过程我们总结起来就是：

- 1、 `Page` 的构建需要 `State` 、 `Effect` 、 `Reducer` 、 `view` 、 `dependencies` 、 `middleware` 等参数。

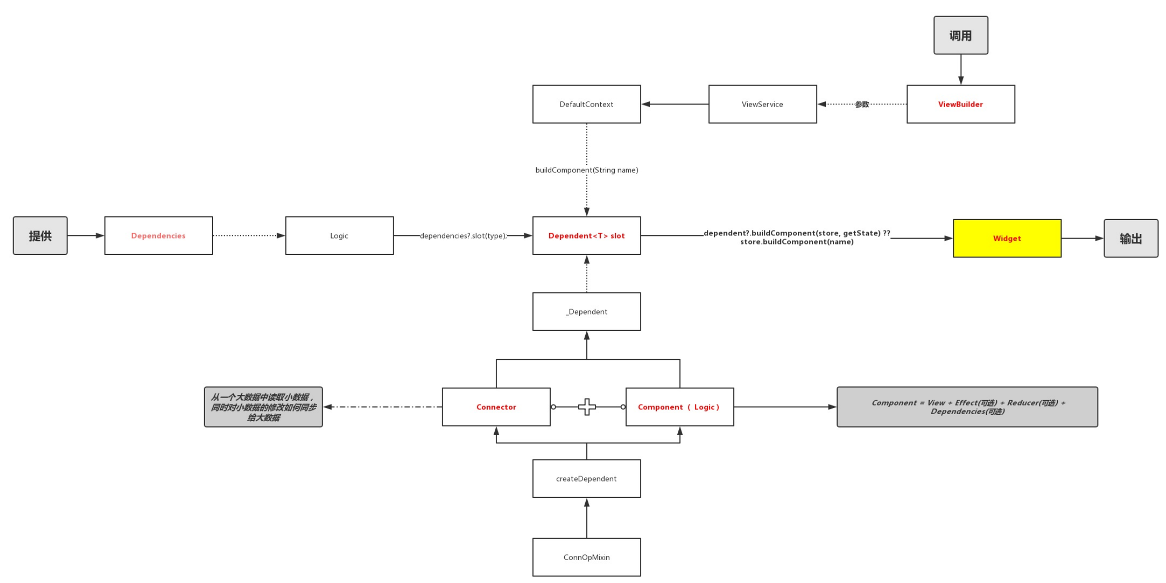
- 2、 `Page` 的内部 `PageProvider` 是一个 `InheritedWidget` 用户状态共享。
- 3、 `Page` 内部会通过 `createMixedStore` 创建 `Store` 对象。
- 4、 `Store` 对象对外提供的 `subscribe` 方法，在订阅时**会将订阅的方法添加到内部 `List<_VoidCallback> _listeners`** 。
- 5、 `Store` 对象内部的 `StreamController.broadcast` 创建出了 `_notifyController` 对象用于广播更新。
- 6、 `Store` 对象内部的 `subscribe` 方法，会在 `ComponentState` 中添加订阅方法 `onNotify` ，**如果调用在 `onNotify` 中最终会执行 `setState` 更新UI。**
- 7、 `Store` 对象对外提供的 `dispatch` 方法，执行时内部会执行 4 中的 `List<_VoidCallback> _listeners` ，触发 `onNotify` 。
- 8、 `Page` 内部会通过 `Logic` 创建 `Dispatch` ，执行时经历 `Effect` -> `Middleware` -> `Stroe.dispatch` -> `Reducer` -> `State` -> **`_notifyController`** -> `_notifyController.add(state)` 等流程。
- **9、以上流程最终就是 `Dispatch` 触发 `Store` 内部 `_notifyController` ， 最终会触发 `ComponentState` 中的 `onNotify` 中的 `setState` 更新UI**



是不是有很多对象很陌生？

确实 `fish_redux` 的整体流程更加复杂，内部的 `ContxtSys` 、 `Componet` 、 `ViewSerivce` 、 `Logic` 等等概念设计，这里因为篇幅有限就不详细拆分展示了，但从整个流程可以看出 `fish_redux` **从控件到页面更新，全都进行了新的独立设计，而这里面最有意思的，莫不过 `dependencies`** 。

如下图所示，得益于 `fish_redux` 内部 `ConnOpMixin` 中对操作符的重载，我们可以通过 `DoubleCountConnector() + DoubleCountComponent()` 来实现 `Dependent` 的组装。



`Dependent` 的组装中 `Connector` 会从总 State 中读取需要的小 State 用于 `Component` 的绘制，这样很好的达到了 **模块解耦与复用** 的效果。

而使用中我们组装的 `dependencies` 最后都会通过 `ViewService` 提供调用调用能力，比如调用 `buildAdapter` 用于列表能力，调用 `buildComponent` 提供独立控件能力等。

可以看出 `flutter_redux` 的内部实现复杂度是比较高的，在提供组装、复用、解耦的同时，也对项目进行了一定程度的入侵，这里的篇幅可能不能很全面的分析 `flutter_redux` 中的整个流程，但是也能让你理解整个流程的关键点，细细品味设计之美。

> 自此，第十二篇终于结束了！(///▽///)

资源推荐

- 本文Demo ： https://github.com/CarGuo/state_manager_demo
- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

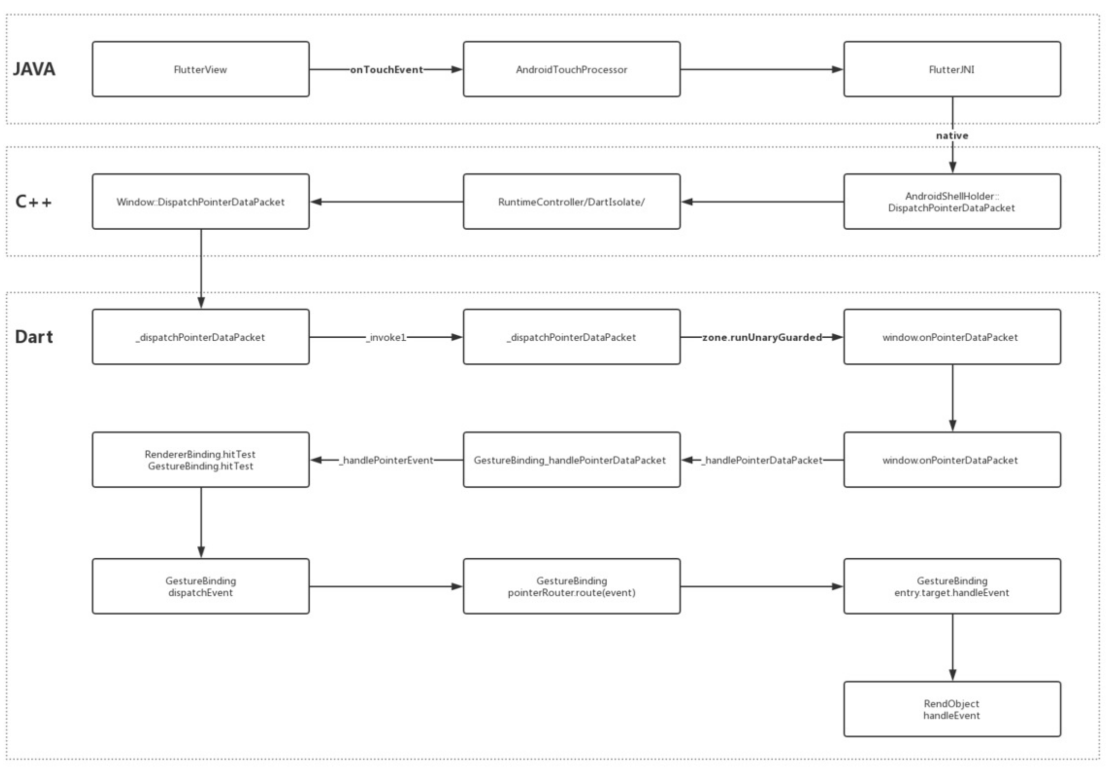
- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

本篇将带你深入了解 Flutter 中的手势事件传递、事件分发、事件冲突竞争，滑动流畅等等的原理，帮你构建一个完整的 Flutter 闭环手势知识体系，这也许是目前最全面的手势事件和滑动源码的深入文章了。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

Flutter 中默认情况下，以 Android 为例，所有的事件都是起原生源于 `io.flutter.view.FlutterView` 这个 `SurfaceView` 的子类，整个触摸手势事件实质上经历了 **JAVA => C++ => Dart** 的一个流程，整个流程如下图所示，无论是 Android 还是 IOS ，原生层都只是将所有事件打包下发，比如在 Android 中，手势信息被打包成 `ByteBuffer` 进行传递，最后在 Dart 层的 `_dispatchPointerDataPacket` 方法中，通过 `_unpackPointerDataPacket` 方法解析成可用的 `PointerDataPacket` 对象使用。



那么具体在 Flutter 中是如何分发使用手势事件的呢？

1、事件流程

在前面的流程图中我们知道，在 Dart 层中手势事件都是从 `_dispatchPointerDataPacket` 开始的，之后会通过 `Zone` 判断环境回调，会执行 `GestureBinding` 这个胶水类中的 `_handlePointerEvent` 方法。*(如果对 `Zone` 或者 `GestureBinding` 有疑问可以翻阅前面的篇章)*

如下代码所示， `GestureBinding` 的 `_handlePointerEvent` 方法中主要是 `hitTest` 和 `dispatchEvent` ： **通过 `hitTest` 碰撞，得到一个包含控件的待处理成员列表 `HitTestResult` ，然后通过 `dispatchEvent` 分发事件并产生竞争，得到胜利者相应。**

```
void _handlePointerEvent(PointerEvent event) {
  assert(!locked);
  HitTestResult hitTestResult;
  if (event is PointerDownEvent || event is PointerSignalEvent) {
    hitTestResult = HitTestResult();
    ///开始碰撞测试了，会添加各个控件，得到一个需要处理的控件成员列表
    hitTest(hitTestResult, event.position);
    if (event is PointerDownEvent) {
      _hitTests[event.pointer] = hitTestResult;
    }
  } else if (event is PointerUpEvent || event is PointerCancelEvent) {
    ///复用机制，抬起和取消，不用hitTest，移除
    hitTestResult = _hitTests.remove(event.pointer);
  } else if (event.down) {
    ///复用机制，手指处于滑动中，不用hitTest
    hitTestResult = _hitTests[event.pointer];
  }
  if (hitTestResult != null ||
      event is PointerHoverEvent ||
      event is PointerAddedEvent ||
      event is PointerRemovedEvent) {
    ///开始分发事件
    dispatchEvent(event, hitTestResult);
  }
}
```

了解了结果后，接下来深入分析这两个关键方法：

1.1 、hitTest

`hitTest` 方法主要为了得到一个 `HitTestResult` ，这个 `HitTestResult` 内有一个 `List<HitTestEntry>` 是用于分发和竞争事件的，而每个 `HitTestEntry.target` 都会存储每个控件的 `RenderObject` 。

因为 `RenderObject` 默认都实现了 `HitTestTarget` 接口，所以可以理解为： **`HitTestTarget` 大部分时候都是 `RenderObject` ，而 `HitTestResult` 就是一个带着碰撞测试后的控件列表。**

事实上 `hitTest` 是 `HitTestable` 抽象类的方法，而 Flutter 中所有实现 `HitTestable` 的类有 **`GestureBinding` 和 `RendererBinding`** ，它们都是 `mixins` 在 `WidgetsFlutterBinding` 这个入口类上，并且因为它们的 `mixins` 顺序的关系，所以 **`RendererBinding` 的 `hitTest` 会先被调用，之后才调用 `GestureBinding` 的 `hitTest`** 。

那么这两个 hitTest 又分别干了什么事呢?

1.2、RendererBinding.hitTest

在 `RendererBinding.hitTest` 中会执行 `renderView.hitTest(result, position: position);` ，如下代码所示， `renderView.hitTest` 方法内会执行 `child.hitTest` ，它将尝试将符合条件的 child 控件添加到 `HitTestResult` 里，最后把自己添加进去。

```
///RendererBinding

bool hitTest(HitTestResult result, { Offset position }) {
    if (child != null)
      child.hitTest(result, position: position);
    result.add(HitTestEntry(this));
    return true;
  }
```

而查看 `child.hitTest` 方法源码，如下所示， `RenderObjcet` 中的 `hitTest` ，会通过 `_size.contains` 判断自己是否属于响应区域，确认响应后执行 `hitTestChildren` 和 `hitTestSelf` ，尝试添加下级的 child 和自己添加进去，这样的**递归就让我们自下而上的得到了一个 `HitTestResult` 的相应控件列表了，最底下的 Child 在最上面。**

```
  ///RenderObjcet

  bool hitTest(HitTestResult result, { @required Offset position }) {
    if (_size.contains(position)) {
      if (hitTestChildren(result, position: position) || hitTestSelf(position)) {
        result.add(BoxHitTestEntry(this, position));
        return true;
      }
    }
    return false;
  }
```

1.3、GestureBinding.hitTest

最后 `GestureBinding.hitTest` 方法不过最后把 `GestureBinding` 自己也添加到 `HitTestResult` 里，最后因为后面我们的流程还会需要回到 `GestureBinding` 中去处理。

1.4、dispatchEvent

`dispatchEvent` 中主要是对事件进行分发，并且通过上述添加进去的 `target.handleEvent` 处理事件，如下代码所示，在存在碰撞结果的时候，是会通过循环对每个控件内部的 `handleEvent` 进行执行。

```
  @override // from HitTestDispatcher
  void dispatchEvent(PointerEvent event, HitTestResult hitTestResult) {
      ///如果没有碰撞结果，那么通过 `pointerRouter.route` 将事件分发到全局处理。
    if (hitTestResult == null) {
```

```
    try {
      pointerRouter.route(event);
    } catch (exception, stack) {
    return;
  }
  ///上面我们知道 HitTestEntry 中的 target 是一系自下而上的控件
  ///还有 renderView 和 GestureBinding
  ///循环执行每一个的 handleEvent 方法
  for (HitTestEntry entry in hitTestResult.path) {
    try {
      entry.target.handleEvent(event, entry);
    } catch (exception, stack) {
    }
  }
}
```

事实上并不是所有的控件的 `RenderObject` 子类都会处理 `handleEvent` ，大部分时候，只有带有 `RenderPointerListener` (RenderObject) / `Listener` (Widget) 的才会处理 `handleEvent` 事件，并且从上述源码可以看出，**handleEvent 的执行是不会被拦截打断的。**

那么问题来了，如果同一个区域内有多个控件都实现了 `handleEvent` 时，那最后事件应该交给谁消耗呢?

更具体为一个场景问题就是：**比如一个列表页面内，存在上下滑动和 Item 点击时，Flutter 要怎么分配手势事件？** 这就涉及到事件的竞争了。

> **核心要来了，高能预警！！！**

2、事件竞争

Flutter 在设计事件竞争的时候，定义了一个很有趣的概念：**通过一个竞技场，各个控件参与竞争，直接胜利的或者活到最后的第一位，你就获胜得到了胜利。** 那么为了分析接下来的“战争”，我们需要先看几个概念：

- **`GestureRecognizer`** ：手势识别器基类，基本上 `RenderPointerListener` 中需要处理的手势事件，都会分发到它对应的 `GestureRecognizer` ，并经过它处理和竞技后再分发出去，常见有 ： `OneSequenceGestureRecognizer` 、 `MultiTapGestureRecognizer` 、 `VerticalDragGestureRecognizer` 、 `TapGestureRecognizer` 等等。
- **`GestureArenaManagerr`** ：手势竞技管理，它管理了整个“战争”的过程，原则上竞技胜出的条件是 ：**第一个竞技获胜的成员或最后一个不被拒绝的成员。**
- **`GestureArenaEntry`** ：提供手势事件竞技信息的实体，内封装参与事件竞技的成员。
- **`GestureArenaMember`**：参与竞技的成员抽象对象，内部有 `acceptGesture` 和 `rejectGesture` 方法，它代表手势竞技的成员，默认 `GestureRecognizer` 都实现了它，**所有竞技的成员可以理解为就是 `GestureRecognizer` 之间的竞争。**

- **`_GestureArena`** ：`GestureArenaManager` 内的竞技场，内部持参与竞技的 `members` 列表，官方对这个竞技场的解释是： **如果一个手势试图在竞技场开放时(isOpen=true)获胜，它将成为一个带有“渴望获胜”的属性的对象。当竞技场关闭(isOpen=false)时，竞技场将寻找一个“渴望获胜”的对象成为新的参与者，如果这时候刚好只有一个，那这一个参与者将成为这次竞技场胜利的青睐存在。**

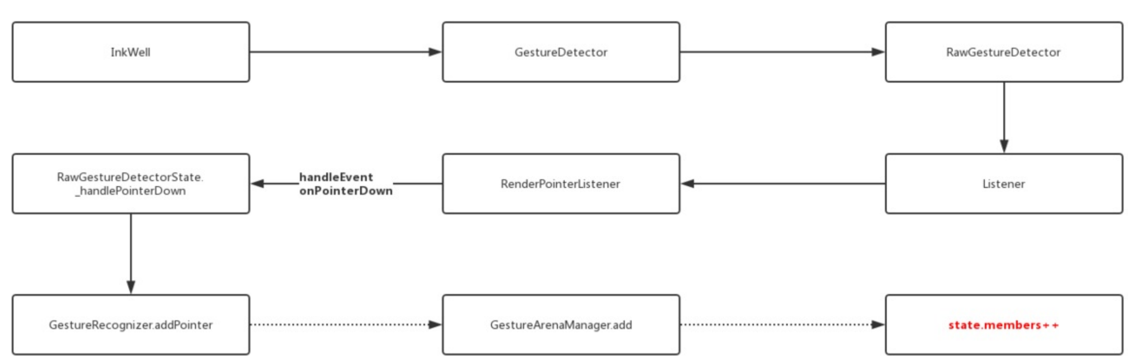
好了，知道这些概念之后我们开始分析流程，我们知道 `GestureBinding` 在 `dispatchEvent` 时会先判断是否有 `HitTestResult` 是否有结果，一般情况下是存在的，所以直接执行循环 `entry.target.handleEvent` 。

2.1、PointerDownEvent

循环执行过程中，我们知道 `entry.target.handleEvent` 会触发 `RenderPointerListener` 的 `handleEvent` ，而事件流程中第一个事件一般都会是 `PointerDownEvent` 。

> `PointerDownEvent` 的流程在事件竞技流程中相当关键，因为它会触发 `GestureRecognizer.addPointer` 。

`GestureRecognizer` 只有通过 `addPointer` 方法将 `PointerDownEvent` 事件和自己绑定，并添加到 `GestureBinding` 的 `PointerRouter` 事件路由和 `GestureArenaManager` 事件竞技中，后续的事件这个控件的 `GestureRecognizer` 才能响应和参与竞争。



> 事实上 **Down** 事件在 Flutter 中一般都是用来做添加判断的，如果存在竞争时，大部分时候是不会直接出结果的，而 **Move** 事件在不同 `GestureRecognizer` 中会表现不同，而 **UP** 事件之后，一般会强制得到一个结果。

所以我们知道了**事件在 `GestureBinding` 开始分发的时候，在 `PointerDownEvent` 时需要响应事件的 `GestureRecognizer` 们，会调用 `addPointer` 将自己添加到竞争中。之后流程中如果没有特殊情况，一般会执行到参与竞争成员列表的 last，也就是 `GestureBinding` 自己这个 handleEvent 。**

如下代码所示，走到 `GestureBinding` 的 `handleEvent` ，在 Down 事件的流程中，一般 `pointerRouter.route` 不会怎么处理逻辑，然后就是 `gestureArena.close` 关闭竞技场了，尝试得到胜利者。

```
@override // from HitTestTarget
void handleEvent(PointerEvent event, HitTestEntry entry) {
```

```
      /// 导航事件去触发 `GestureRecognizer` 的 handleEvent
      /// 一般 PointerDownEvent 在 route 执行中不怎么处理。
    pointerRouter.route(event);

    ///gestureArena 就是 GestureArenaManager
    if (event is PointerDownEvent) {

        ///关闭这个 Down 事件的竞技，尝试得到胜利
      /// 如果没有的话就留到 MOVE 或者 UP。
      gestureArena.close(event.pointer);

    } else if (event is PointerUpEvent) {
        ///已经到 UP 了，强行得到结果。
      gestureArena.sweep(event.pointer);

    } else if (event is PointerSignalEvent) {
      pointerSignalResolver.resolve(event);
    }
  }
```

让我们看 `GestureArenaManager` 的 `close` 方法，下面代码我们可以看到，如果前面 Down 事件中没有通过 `addPointer` 添加成员到 `_arenas` 中，那会连参加的机会都没有，而进入 `_tryToResolveArena` 之后，**如果 `state.members.length == 1` ，说明只有一个成员了，那就不竞争了，直接它就是胜利者，直接响应后续所有事件。** 那么如果是多个的话，就需要后续的竞争了。

```
  void close(int pointer) {
      /// 拿到我们上面 addPointer 时添加的成员封装
    final _GestureArena state = _arenas[pointer];
    if (state == null)
      return; // This arena either never existed or has been resolved.
    state.isOpen = false;
    ///开始打起来吧
    _tryToResolveArena(pointer, state);
  }

  void _tryToResolveArena(int pointer, _GestureArena state) {
    if (state.members.length == 1) {
      scheduleMicrotask(() => _resolveByDefault(pointer, state));
    } else if (state.members.isEmpty) {
      _arenas.remove(pointer);
    } else if (state.eagerWinner != null) {
      _resolveInFavorOf(pointer, state, state.eagerWinner);
    }
  }
```

2.2 开始竞争

那竞争呢？接下来我们以 `TapGestureRecognizer` 为例子，如果控件区域内存在两个 `TapGestureRecognizer` ，那么在 `PointerDownEvent` 流程是不会产生胜利者的，这时候如果没有 MOVE 打断的话，**到了 UP 事件时，就会执行 `gestureArena.sweep(event.pointer);` 强行选取一个。**

而选择的方式也是很简单，**就是 `state.members.first` ，从我们之前 `hitTest` 的结果上理解的话，就是控件树的最里面 Child 了。** 这样胜利的 member 会通过 `members.first.acceptGesture(pointer)` 回调到 `TapGestureRecognizer.acceptGesture` 中，**设置 `_wonArenaForPrimaryPointer` 为 ture 标志为胜利区域，然后执行 `_checkDown` 和 `_checkUp` 发出事件响应触发给这个控件。**

而这里有个有意思的就是，Down 流程的 `acceptGesture` 中的 `_checkUp` 因为没有 `_finalPosition` 此时是不会被执行的，**`_finalPosition` 会在 `handlePrimaryPointer` 方法中，获得 `_finalPosition` 并判断 `_wonArenaForPrimaryPointer` 标志为，再次执行 `_checkUp` 才会成功。**

> `handlePrimaryPointer` 是在 UP 流程中 `pointerRouter.route` 触发 `TapGestureRecognizer` 的 `handleEvent` 触发的。

那么问题来了， `_checkDown` 和 `_checkUp` 时在 UP 事件一次性被执行，那么如果我长按住的话， `_checkDown` 不是没办法正确回调了?

当然不会，在 `TapGestureRecognizer` 中有一个 `didExceedDeadline` 的机制，在前面 Down 流程中，**在 `addPointer` 时 `TapGestureRecognizer` 会创建一个定时器**，这个定时器的时间时 `kPressTimeout = 100毫秒` ，**如果我们长按住的话，就会等待到触发 `didExceedDeadline` 去执行 `_checkDown` 发出 `onTabDown` 事件了。**

> `_checkDown` 执行发送过程中，会有一个标志为 `_sentTapDown` 判断是否已经发送过，如果发送过了也不会在重发，之后回到原本流程去竞争，手指抬起后得到胜利者相应，同时在 `_checkUp` 之后 `_sentTapDown` 标识为会被重置。

这也可以分析点击下的几种场景:

普通按下:

- 1、区域内只有一个 `TapGestureRecognizer` ：Down 事件时直接在竞技场 `close` 时就得到竞出胜利者，调用 `acceptGesture` 执行 `_checkUp` ，到 Up 事件的时候通过 `handlePrimaryPointer` 执行 `_checkUp` ，结束。
- 2、区域内有多个 `TapGestureRecognizer` ：Down 事件时在竞技场 `close` 不会竞出胜利者，在 Up 事件的时候，会在 `route` 过程通过 `handlePrimaryPointer` 设置好 `_finalPosition` ，之后经过竞技场 `sweep` 选取排在第一个位置的为胜利者，调用 `acceptGesture` ，执行 `_checkDown` 和 `_checkUp` 。

长按之后抬起:

1、区域内只有一个 `TapGestureRecognizer` ：除了 Down 事件是在 `didExceedDeadline` 时发出 `_checkDown` 外其他和上面基本没区别。

- 2、区域内有多个 `TapGestureRecognizer` ： Down 事件时在竞技场 `close` 时不会竞出胜利者，但是会触发定时器 `didExceedDeadline` ，先发出 `_checkDown` ，之后再经过 `sweep` 选取第一个座位胜利者，调用 `acceptGesture` ，触发 `_checkUp`

那么问题又来了，你有没有疑问，如果有区域两个 `TapGestureRecognizer` ，长按的时候因为都触发了 `didExceedDeadline` 执行 `_checkDown` 吗？

答案是：会的！**因为定时器都触发了 `didExceedDeadline` ，所以 `_checkDown` 都会被执行，从而都发出了 `onTapDown` 事件。但是后续竞争后，只会执行一个 `_checkUp` ，所有只会有一个控件响应 `onTap` 。**

竞技失败：

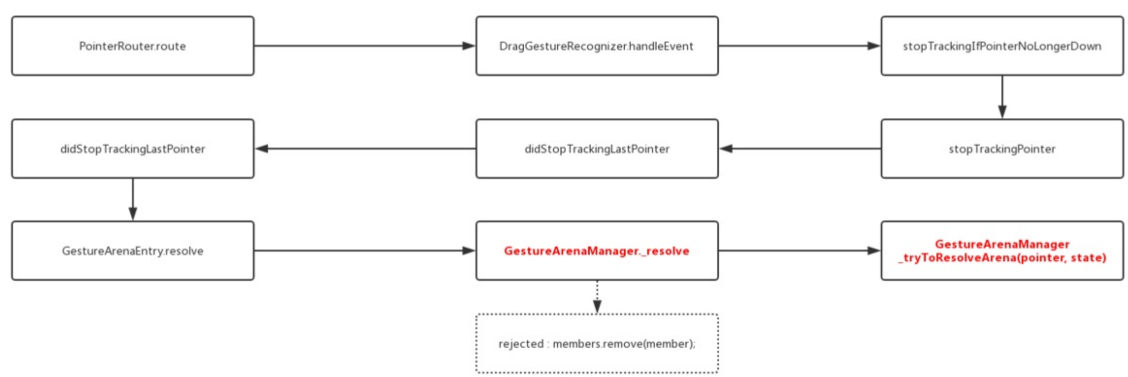
在竞技场竞争失败的成员会被移出竞技场，移除后就没办法参加后面事件的竞技了，比如 `TapGestureRecognizer` 在接受到 `PointerMoveEvent` 事件时就会直接 `rejected` , 并触发 `rejectGesture` ，之后定时器会被关闭，并且触发 `onTapCancel` ，然后重置标志位.

总结下：

Down 事件时通过 `addPointer` 加入了 `GestureRecognizer` 竞技场的区域，在没移除的情况下，事件可以参加后续事件的竞技，在某个事件阶段移除的话，之后的事件序列也会无法接受。事件的竞争如果没有胜利者，在 UP 流程中会强制指定第一个为胜利者。

2.3 滑动事件

滑动事件也是需要在 Down 流程中 `addPointer` ，然后 MOVE 流程中，通过在 `PointerRouter.route` 之后执行 `DragGestureRecognizer.handleEvent` 。



在 `PointerMoveEvent` 事件的 `DragGestureRecognizer.handleEvent` 里，会通过在 `_hasSufficientPendingDragDeltaToAccept` 判断是否符合条件，如：

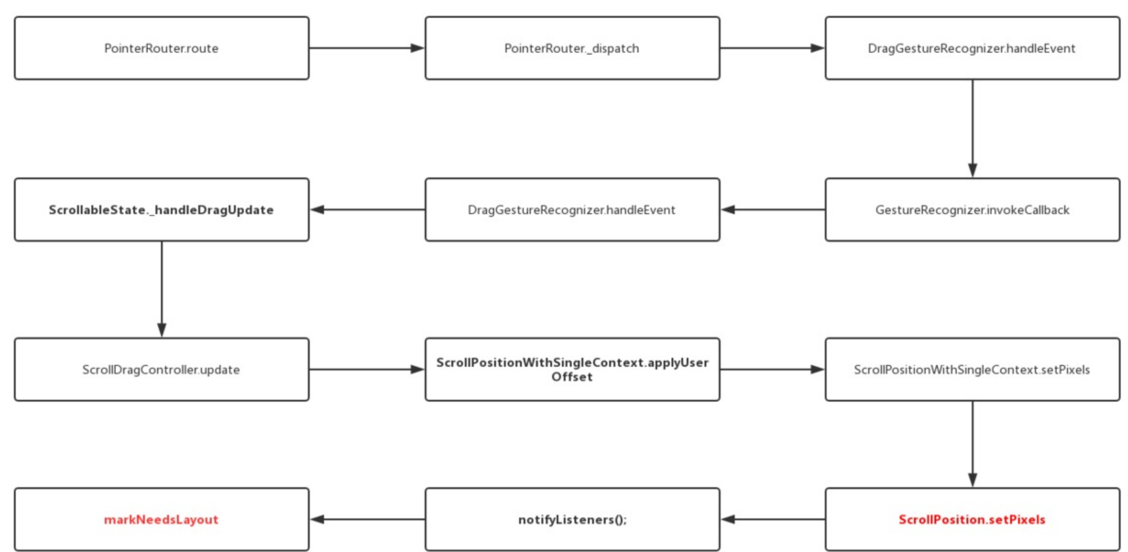
```
bool get _hasSufficientPendingDragDeltaToAccept => _pendingDragOffset.dy.abs() > kT
ouchSlop;
```

如果符合条件就直接执行 `resolve(GestureDisposition.accepted);` ，将流程回到竞技场里，然后执行 `acceptGesture` ，然后触发 `onStart` 和 `onUpdate` 。

回到我们前面的上下滑动可点击列表，是不是很明确了：**如果是点击的话，没有产生 MOVE 事件，所以 `DragGestureRecognizer` 没有被接受，而Item 作为 Child 第一位，所以响应点击。如果有 MOVE 事件，`DragGestureRecognizer` 会被 `acceptGesture`，而点击 `GestureRecognizer` 会被移除事件竞争，也就没有后续 UP 事件了。**

那这个 `onUpdate` 是怎么让节目动起来的?

我们以 `ListView` 为例子，通过源码可以知道， `onUpdate` 最后会调用到 `Scrollable` 的 `_handleDragUpdate` ，这时候会执行 `Drag.update` 。



通过源码我们知道 `ListView` 的 `Drag` 实现其实是 `ScrollDragController`，它在 `Scrollable` 中是和 `ScrollPositionWithSingleContext` 关联的在一起的。那么 `ScrollPositionWithSingleContext` 又是什么?

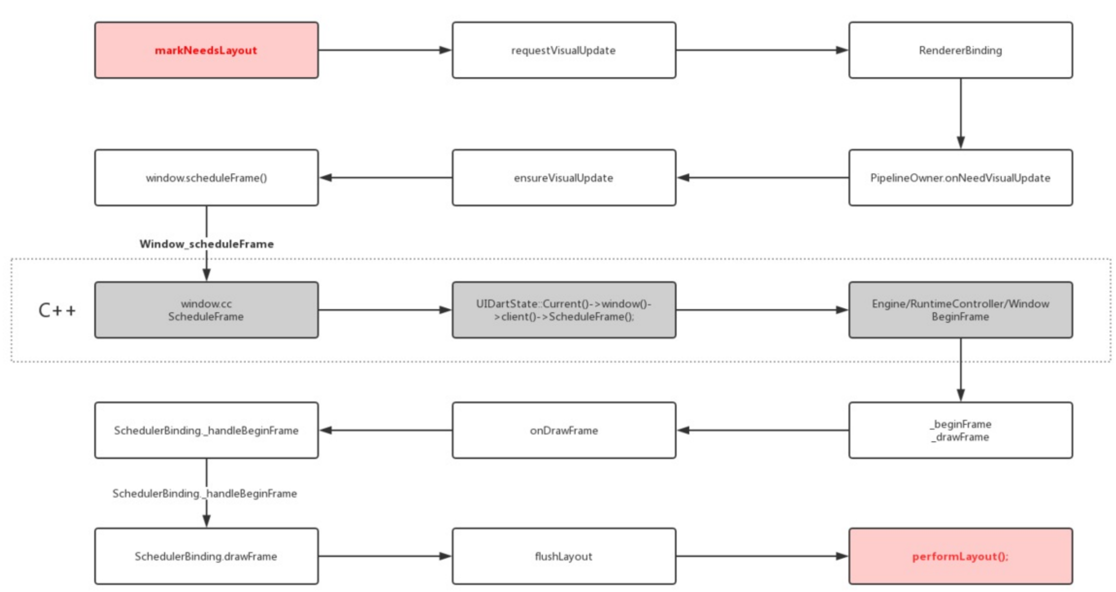
`ScrollPositionWithSingleContext` 其实就是这个滑动的关键，它其实就是 `ScrollPosition` 的子类，而 `ScrollPosition` 又是 `ViewportOffset` 的子类，而 `ViewportOffset` 又是一个 `ChangeNotifier` ，出现如下关系：

> 继承关系：**ScrollPositionWithSingleContext : ScrollPosition : ViewportOffset : ChangeNotifier**

所以 **ViewportOffset** 就是滑动的关键点。上面我们知道响应区域 `DragGestureRecognizer` 胜利之后执行 `Drag.update` ，最终会调用到 `ScrollPositionWithSingleContext` 的 `applyUserOffset` ，导致内部确定位置的 `pixels` 发生改变，并执行父类 `ChangeNotifier` 的方法 `notifyListeners` 通知更新。

而在 `ListView` 内部 `RenderViewportBase` 中，这个 `ViewportOffset` 是通过 `_offset.addListener(markNeedsLayout);` 绑定的，so ，**触摸滑动导致 `Drag.update` ，最终会执行到 `RenderViewportBase` 中的 `markNeedsLayout` 触发页面更新。**

至于 `markNeedsLayout` 如何更新界面和滚动列表，这里暂不详细描述了，给个图感受下：



> 自此，第十三篇终于结束了！(///▽///)

资源推荐

- 本文Demo ： https://github.com/CarGuo/state_manager_demo
- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

本篇将带你深入了解 Flutter 中打包和插件安装等原理，帮你快速完成 Flutter 集成到现有 Android 项目，实现混合开发支持。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

一、前言

随着各种跨平台框架的不断涌现，**很多时候我们会选择混合开发模式作为脚手架**，因为企业一般不会把业务都压在一个框架上，同时除非是全新项目，不然出于对原有业务重构的 **成本和风险** 考虑，都会选择混合开发去尝试入坑。

但是混合开发会对 **打包、构建和启动等流程熟悉度要求较高**，同时遇到的问题也更多，以前我在 `React Native` 也写过类似的文章：《从Android到React Native开发（四、打包流程解析和发布为Maven库）》，而这方面是有很多经验可以通用的，**所以适当的混开模式有利于避免一些问题，同时只有了解 `Flutter` 整体项目的构建思路，才有可能更舒适的躺坑。**

> 额外唠叨一句，跨平台的意义更多在于解决多端逻辑的统一，至少避免了逻辑重复实现，所以企业刚开始，一般会选择一些轻量级业务进行尝试。
>
> ## 官方未来将有 `Flutter build aar` 的方法可提供使用。

二、打包

一般跨平台混合开发会有两种选择：

- 1、将 `Flutter` 整体框架依赖和打包脚本都集成到主项目中。
- 2、以 `aar` 的完整库集成形式添加到主项目。

两种实现方法各有利弊：

- 第一种方式可以**更方便运行时修改问题，但是对主项目“污染”会比较高，同时改动会大一些。**
- 第二种方式 **需要单独调试后，更新 `aar` 文件再集成到项目中调试，但是这类集成方式更干净，同时 `Flutter` 相关代码可独立运行测试，且改动较小。**

一般而言，对于普通项目我是建议以 **第二种方式集成到项目中的**，通过新建一个 `Flutter` 工程，然后对工程进行组件化脚本处理，让它 **既能以 apk形式单独运行调试，又能打包为aar形式对外提供支持。**

相信对于原生平台熟悉的应该知道，我们可以通过简单修改项目 `gradle` 脚本，让它快速支持这个能力，如下图片所示，图片中为省略的部分脚本代码，完整版可见 flutter_app_lib 。

```
def isLib = true

if(isLib) {
    apply plugin: 'com.android.library'
} else  {
    apply plugin: 'com.android.application'
}

apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"

android {
    defaultConfig {
        if(!isLib) {
            applicationId "com.shuyu.flutter_app_lib"
        }
        if(isLib) {
            ndk {
                //设置支持的SO库架构
                abiFilters 'armeabi', 'armeabi-v7a', 'x86'
            }
        }
    }
}
```

我们通过了 `isLib` 标记为去简单实现了项目的打包判断，当项目作为 `lib` 发布时，**设置 `isLib` 为 true，之后执行 `./gradlew assembleRelease` 即可**，剩下的工作依旧是 `Flutter` 自身的打包流程，而对于打包后的 `aar` 文件直接在原生项目里引入即可完成依赖。

而一般接入时，**如果需要 `token` 、用户数据等信息，推荐提供定义好原生接口，如 `init(String token, String userInfo)` 等，然后通过 `MethodChannel` 将信息同步到 `Flutter` 中。**

对于原生主工程，只需要接入 `aar` 文件，完成初始化并打开页面，而无需关心其内部实现，和引入普通依赖并无区别。

> 你可能需要修改的还有 `AndroidManifset` 中的启动 `MainActivity` 移除，然后添加一个自定义 `Activity` 去继承 `FlutterActivity` 完成自定义。

三、插件

如果普通情况下，到上面就可以完成 `Flutter` 的集成工作了，但是往往事与愿违，**一些 `Flutter` 插件在提供功能时，往往是通过原生层代码实现的，如 `flutter_webview` 、 `android_intent` 、 `device_info` 等等，那这些代码是怎么被引用的呢?**

> 这里稍微提一下，用过 `React Native` 的应该知道，带有原生代码的 `React Native` 插件，在 `npm` 安装以后，需要通过 `react-native link` 命令完成安装处理。**这个命令会触发脚本修改原生代码，从而修改 `gradle` 脚本增加对插件项目的引用，同时修改 `java` 代码实现插件的模版引入，这使得项目在一定程度被插件“污染”。**

在 `React Native` 中带有原生代码的插件，会被以本地 `Module` 工程的方式引入，那 `Flutter` 呢?

其实原理上 `Flutter` **带有原生代码的插件，在插件安装后，也是会以本地 `Module Project` 的形式引入**， 但是它整个过程更加巧妙，让开发中对这个过程几乎无感。

如下图所示，不知道你注意过没有，在插件安装之后，所有带原生代码的插件，都会以路径和插件名的 `key=value` 形式 **存在 `.flutter-plugins` 文件中。**

```
android_intent=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/android_intent-0.3.0+2/
connectivity=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/connectivity-0.4.3+1/
device_info=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/device_info-0.4.0+1/
flutter_lottie=/Users/guoshuyu/.pub-cache/git/flutter_lottie-a59dffd8282746de1c0e463ae12bbd737205aec1/
flutter_statusbar=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/flutter_statusbar-0.0.1/
flutter_webview_plugin=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/flutter_webview_plugin-0.3.4/
fluttertoast=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/fluttertoast-3.0.4/
package_info=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/package_info-0.4.0+3/
path_provider=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/path_provider-0.5.0+1/
permission_handler=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/permission_handler-3.0.0/
share=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/share-0.6.1/
shared_preferences=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/shared_preferences-0.5.2/
sqflite=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/sqflite-1.1.5/
url_launcher=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/url_launcher-5.0.2/
webview_flutter=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/webview_flutter-0.3.6/
```

而在 `android` 工程的 `settings.gradle` 里，如下图所示，**会通过读取该文件将 `.flutter-plugins` 文件中的项目一个个 `include` 到主工程里。**

```
def plugins = new Properties()
def pluginsFile = new File(flutterProjectRoot.toFile(), '.flutter-plugins')
if (pluginsFile.exists()) {
    pluginsFile.withReader('UTF-8') { reader -> plugins.load(reader) }
}

plugins.each { name, path ->
    def pluginDirectory = flutterProjectRoot.resolve(path).resolve('android').toFile()
    include ":$name"
    project(":$name").projectDir = pluginDirectory
}
```

之后就是主工程里的 `apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"` 脚本的引入了，这个脚本一般在于 `flutterSDK/packages/flutter_tools/gradle/` 目录下，如下代码所示，其中最关键的部分同样是 **读取 `.flutter-plugins` 文件中的项目，然后一个一个再 `implementation` 到主工程里完成依赖。**

```
File pluginsFile = new File(project.projectDir.parentFile.parentFile, '.flutter-plugins')
Properties plugins = readPropertiesIfExist(pluginsFile)

plugins.each { name, _ ->
    def pluginProject = project.rootProject.findProject(":$name")
    if (pluginProject != null) {
        project.dependencies {
            if (project.getConfigurations().findByName("implementation")) {
                implementation pluginProject
            } else {
                compile pluginProject
            }
        }
```

自此所有原生代码的 `Flutter` 插件，都被作为本地 `Module Project` 的形式引入主工程了， 最后脚本会自动生成一个 `GeneratedPluginRegistrant.java` 文件，实现原生代码的引用注册， 而这个过程对你完全是无感的。

说了那么多就是为了说明，既然插件是被当作本地 `Module Project` 的形式引入，那么这时候按照原来直接打包 aar 是会有问题的：

```
`Android` 默认 `gradle` 脚本打包时，对于 `project` 和远程依赖只会打包引用而不会打包源码和资源
。
```

所以这时候就需要 `fat-aar` 的加持了，关于 `fat-aar` 的详细概念可见：《从Android到React Native开发（四、打包流程解析和发布为Maven库）》，这里可以简单理解为，**这是一个支持将引用代码和资源到合并到一个 aar 的插件。**

如下代码所示，我们在原本的组件化脚本上，**通过增加 `apply plugin: 'com.kezong.fat-aar'` 引入插件，然后参考 `Flutter` 脚本对 `.flutter-plugins` 文件中的项目进行 `embed` 依赖引用即可**，这时候再打包出的 `aar` 文件即为完整 `Flutter` 项目代码。

由 Xnip 截图

```
def isLib = true

if(isLib) {
    apply plugin: 'com.android.library'
} else  {
    apply plugin: 'com.android.application'
}

apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"
if(isLib) {
    apply plugin: 'com.kezong.fat-aar'
}

android {
    defaultConfig {
        if(!isLib) {
            applicationId "com.shuyu.flutter_app_lib"
        }
    }
}

dependencies {
    ///为库的方式才添加本地仓库依赖，这个本地仓库目前是从 include 那里读取的。
    if(isLib) {
        def flutterProjectRoot = rootProject.projectDir.parentFile.toPath()
        def plugins = new Properties()
        def pluginsFile = new File(flutterProjectRoot.toFile(), '.flutter-plugins')
        if (pluginsFile.exists()) {
            pluginsFile.withReader('UTF-8') { reader -> plugins.load(reader) }
        }
        plugins.each { name, _ ->
            println name
            embed project(path: ":$name", configuration: 'default')
        }
    }

}
```

> 完整版可见 flutter_app_lib 。

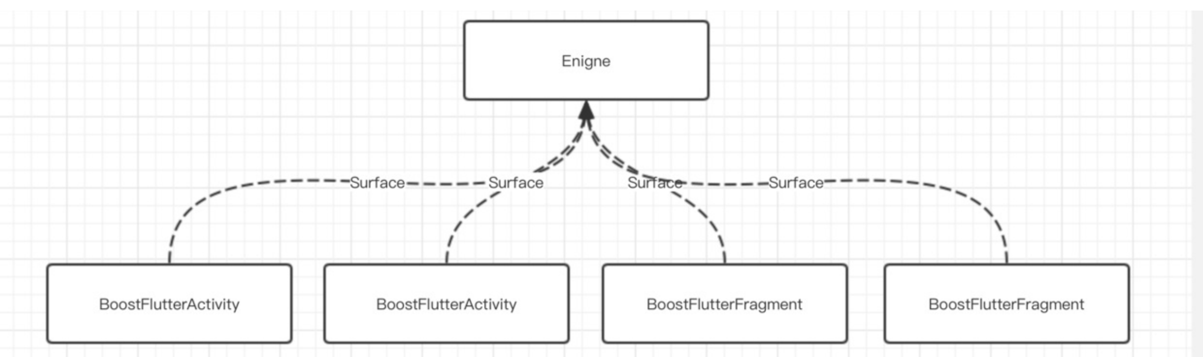
四、堆栈

最后需要说的问题就是堆栈了。

如果说混合开发中最难处理的是什么，那一定是各平台之间的堆栈管理，**一般情况下我们都会避免混合堆栈的相互调用**，但是面对不得不如此为之的情况下，闲鱼给出了他们的答案：**`fluttet_boost`**。

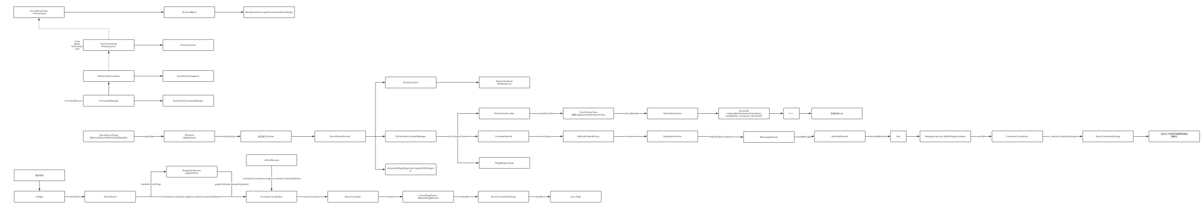
我们知道 `Flutter` 整个项目都是绘制在一个 `Surface` 画布上，而 `fluttet_boost` 将堆栈统一到了原生层，通过一个单例的 `flutter engine` 进行绘制。

每个 `FlutterFragment` 和 `FlutterActivity` 都是一个 `Surface` 承载容器，切换页面时就是切换 `Surface` 渲染显示，而对于不渲染的页面通过 `Surface` 截图缓存画面显示。



这样整个 `Flutter` 的路由就被映射到原生堆栈中，统一由原生页面堆栈管理，`Flutter` 内每 `push` 一个页面就是打开一个 `Activity` 。

> `flutter_boost` 截止到我测试的时间 2019-05-16, 只支持 1.2之前的版本。 `flutter_boost` 的整体流程相对复杂，同时对于 `Dialog` 的支持并不好，且业务跳转深度太深时会出现黑屏问题。



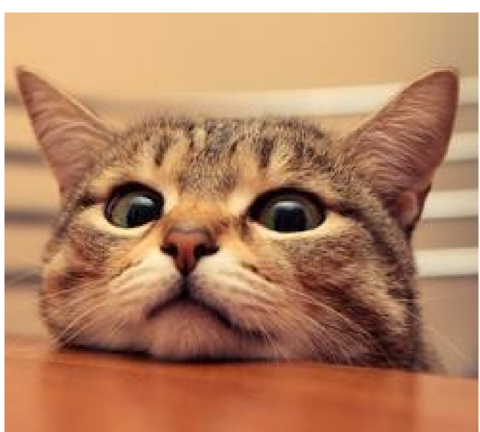
> 自此，第十四篇终于结束了！(///▽///)

资源推荐

- 本文Demo ： https://github.com/CarGuo/flutter_app_lib
- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex



本篇将带你深入理解 Flutter 中 State 的工作机制，并通过对状态管理框架 **Provider** 解析加深理解，看完这一篇你将更轻松的理解你的 “State 大后宫” 。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

```
⚠第十二篇中更多讲解状态的是管理框架，本篇更多讲解 Flutter 本身的状态设计。
```

一、State

1、State 是什么？

我们知道 Flutter 宇宙中万物皆 `Widget` ，而 **`Widget` 是 `@immutable` 即不可变的，所以每个 `Widget` 状态都代表了一帧。**

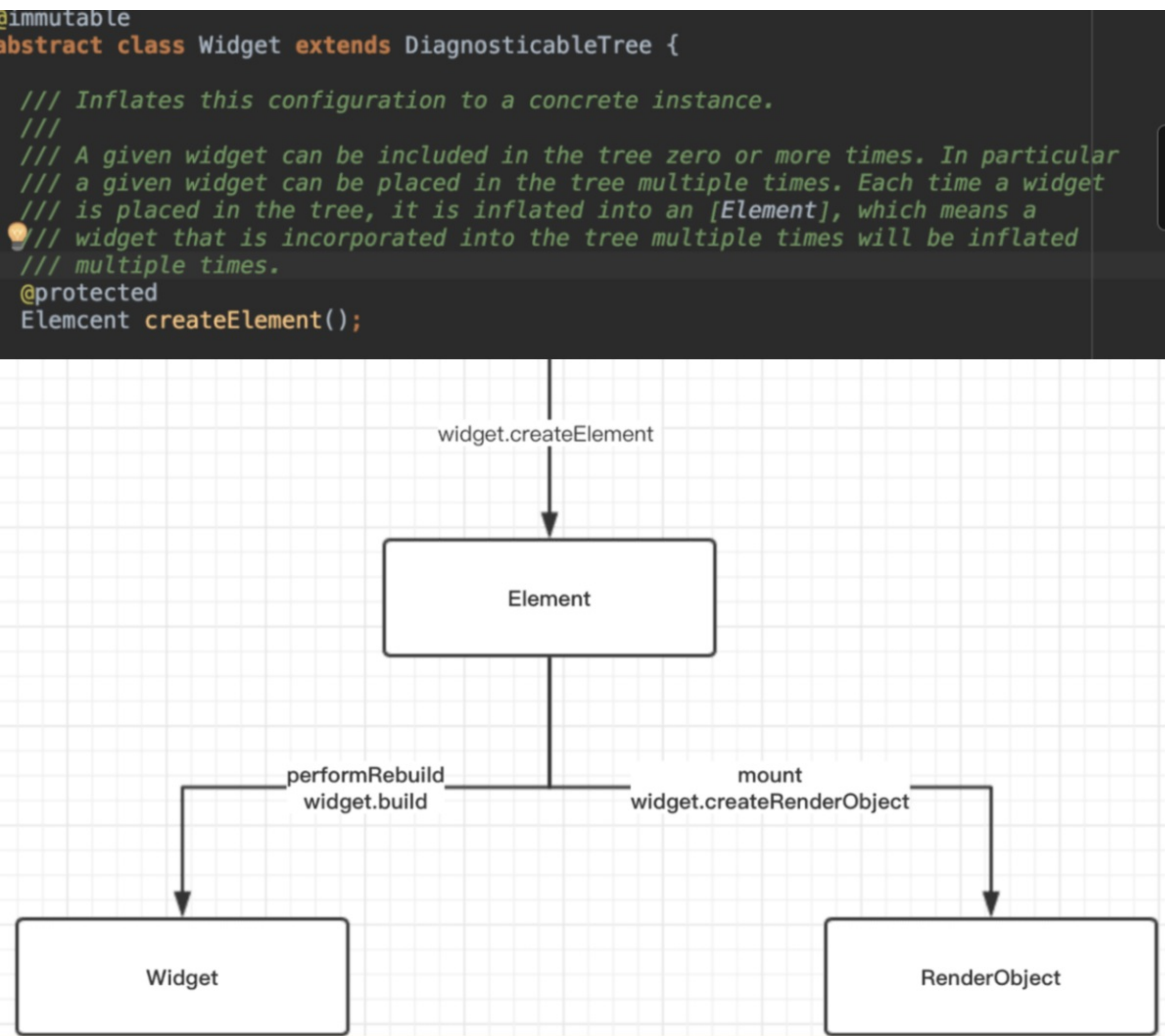
在这个基础上， **`StatefulWidget` 的 `State` 帮我们实现了在 `Widget` 的跨帧绘制** ，也就是在每次 `Widget` 重绘的时候，通过 `State` 重新赋予 `Widget` 需要的绘制信息。

2、State 怎么实现跨帧共享？

这就涉及 Flutter 中 `Widget` 的实现原理，在之前的篇章我们介绍过，这里我们说两个涉及的概念：

- Flutter 中的 `Widget` 在一般情况下，是需要通过 `Element` 转化为 `RenderObject` 去实现绘制的。
- `Element` 是 `BuildContext` 的实现类，同时 `Element` 持有 `RenderObject` 和 `Widget` ，**我们代码中的 `Widget build(BuildContext context) {}` 方法，就是被 `Element` 调用的。**

了解这个两个概念后，我们先看下图，在 Flutter 中构建一个 `Widget` ，首先会创建出这个 `Widget` 的 `Element` ，**而事实上 `State` 实现跨帧共享，就是将 `State` 保存在 `Element` 中，这样 `Element` 每次调用 `Widget build()` 时，是通过 `state.build(this);` 得到的新 `Widget` ，所以写在 `State` 的数据就得以复用了。**



那 `State` 是在哪里被创建的?

如下图所示，**`StatefulWidget` 的 `createState` 是在 `StatefulElement` 的构建方法里创建的，**这就保证了只要 `Element` 不被重新创建，`State` 就一直被复用。

同时我们看 `update` 方法，当新的 `StatefulWidget` 被创建用于更新 UI 时，新的 `widget` 就会被重新赋予到 `_state` 中，而这的设定也导致一个常被新人忽略的问题。

```
/// An [Element] that uses a [StatefulWidget] as its configuration.
class StatefulElement extends ComponentElement {
  /// Creates an element that uses the given widget as its configuration.
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        super(widget) {
    _state._element = this;
    _state._widget = widget;

  }

  /// 1、createState 只在 StatefulElement 创建时才会被创建的。
  /// 2、StatefulElement 的 createElement 一般只在 inflateWidget 调用。
  /// 3、updateChild 执行 inflateWidget 时， 如果 child 存在可以更新的话，不会执行 inflateWidget。
  @override
  void update(StatefulWidget newWidget) {
    super.update(newWidget);
    assert(widget == newWidget);
    final StatefulWidget oldWidget = _state._widget;
    _dirty = true;
    _state._widget = widget;
    rebuild();
  }
```

我们先看问题代码，如下图所示：

- 1、在 `_DemoAppState` 中，我们创建了 `DemoPage` ，并且把 `data` 变量赋给了它。
- 2、 `DemoPage` 在创建 `createState` 时，又将 `data` 通过直接传入 `_DemoPageState` 。
- 3、在 `_DemoPageState` 中直接将传入的 `data` 通过 `Text` 显示出来。

运行后我们一看也没什么问题吧？ **但是当我们点击 4 中的 `setState` 时，却发现 3 中 `Text` 没有发现改变，** 这是为什么呢？

```dart
class DemoApp extends StatefulWidget {
  @override
  _DemoAppState createState() => _DemoAppState();
}

class _DemoAppState extends State<DemoApp> {
  String data = "init";

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      home: Scaffold(
      body: Scaffold(
        body: DemoPage("Test", data, 30),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            data = "setState";
          });
        },
      ),
    ));
  }
}

class DemoPage extends StatefulWidget {
  final String title;
  final String data;
  final int count;

  DemoPage(this.title, this.data, this.count);

  @override
  _DemoPageState createState() => _DemoPageState(this.data);
}

class _DemoPageState extends State<DemoPage> {

  final String data;

  _DemoPageState(this.data);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new ListView.builder(
        itemBuilder: (context, index) {
          /// widget.data
          return new Text(data);
        },
        itemCount: widget.count,
      ),
    );
  }
}
```

1

4

2

3

问题就在于前面 `StatefulElement` 的构建方法和 `update` 方法：

`State` 只在 `StatefulElement` 的构建方法中创建，当我们调用 `setState` 触发 `update` 时，只是执行了 `_state.widget = newWidget` ，而我们通过 `_DemoPageState(this.data)` 传入的 *data* ，在传入后执行 `setState` 时并没有改变。

如果我们采用上图代码中 3 注释的 **`widget.data`** **方法，因为** **`_state.widget = newWidget`** **时，** **`State`** **中的** **`Widget`** **已经被更新了，** **`Text`** **自然就被更新了。**

3、setState 干了什么？

我们常说的 `setState` ，其实是调用了 `markNeedsBuild` ， **`markNeedsBuild` 内部会标记 `element` 为 `diry` ，然后在下一帧 `WidgetsBinding.drawFrame` 才会被绘制，这可以也看出 `setState` 并不是立即生效的。**

```
@protected
void setState(VoidCallback fn) {
  assert(fn != null);
  final dynamic result = fn() as dynamic;
  assert(() {
    if (result is Future) {
      throw FlutterError();
    }
    return true;
  }());
  _element.markNeedsBuild();
}
```

4、状态共享

前面我们聊了 Flutter 中 `State` 的作用和工作原理，接下来我们看一个老生常谈的对象： **`InheritedWidget`** 。

状态共享是常见的需求，比如用户信息和登陆状态等等，而 Flutter 中 `InheritedWidget` 就是为此而设计的，在第十二篇我们大致讲过它：

> 在 `Element` 的内部有一个 `Map<Type, InheritedElement> _inheritedWidgets;` 参数， **`_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时，它才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并。**
>
> 所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 时，就可以通过这个 `Map` 往上查找，从而找到这个上级的 `InheritedWidget` 。

噢，是的， **`InheritedWidget` 共享的是 `Widget` ，只是这个 `Widget` 是一个 `ProxyWidget` ，它自己本身并不绘制什么，但共享这个 `Widget` 内保存有的值，却达到了共享状态的目的。**

如下代码所示，Flutter 内 `Theme` 的共享，共享的其实是 `_InheritedTheme` 这个 `Widget` ，而我们通过 `Theme.of(context)` 拿到的，其实就是保存在这个 `Widget` 内的 `ThemeData` 。

```
  static ThemeData of(BuildContext context, { bool shadowThemeOnly = false }) {
    final _InheritedTheme inheritedTheme = context.inheritFromWidgetOfExactType(_In
heritedTheme);
```

```
    if (shadowThemeOnly) {
      /// inheritedTheme 这个 Widget 内的 theme
      /// theme 内有我们需要的 ThemeData
      return inheritedTheme.theme.data;
    }
    ...
  }
```

这里有个需要注意的点，就是 `inheritFromWidgetOfExactType` 方法刚了什么?

我们直接找到 `Element` 中的 `inheritFromWidgetOfExactType` 方法实现，如下关键代码所示：

- 首先从 `_inheritedWidgets` 中查找是否有该类型的 `InheritedElement` 。
- 查找到后添加到 `_dependencies` 中，并且通过 `updateDependencies` **将当前 `Element` 添加到 `InheritedElement` 的 `_dependents` 这个Map 里。**
- 返回 `InheritedElement` 中的 `Widget` 。

```
  @override
  InheritedWidget inheritFromWidgetOfExactType(Type targetType, { Object aspect })
{
    /// 在共享 map _inheritedWidgets 中查找
    final InheritedElement ancestor = _inheritedWidgets == null ? null : _inherited
Widgets[targetType];
    if (ancestor != null) {
      /// 返回找到的 InheritedWidget ，同时添加当前 element 处理
      return inheritFromElement(ancestor, aspect: aspect);
    }
    _hadUnsatisfiedDependencies = true;
    return null;
  }

  @override
  InheritedWidget inheritFromElement(InheritedElement ancestor, { Object aspect })
{
    _dependencies ??= HashSet<InheritedElement>();
    _dependencies.add(ancestor);
   /// 就是将当前 element（this） 添加到  _dependents 里
   /// 也就是 InheritedElement 的 _dependents
   /// _dependents[dependent] = value;
    ancestor.updateDependencies(this, aspect);
    return ancestor.widget;
  }

  @override
  void notifyClients(InheritedWidget oldWidget) {
    for (Element dependent in _dependents.keys) {
      notifyDependent(oldWidget, dependent);
    }
  }
```

这里面的关键就是 `ancestor.updateDependencies(this, aspect);` 这个方法：

我们都知道，获取 `InheritedWidget` 一般需要 `BuildContext` ，如 `Theme.of(context)` ，而 `BuildContext` 的实现就是 `Element` ，**所以当我们调用 `context.inheritFromWidgetOfExactType` 时，就会将这个 `context` 所代表的 `Element` 添加到 `InheritedElement` 的 `_dependents` 中。**

这代表着什么？

比如当我们在 `StatefulWidget` 中调用 `Theme.of(context).primaryColor` 时，**传入的 `context` 就代表着这个 `Widget` 的 `Element` ，在 `InheritedElement` 里被“登记”到 `_dependents` 了。**

而当 `InheritedWidget` 被更新时，如下代码所示， `_dependents` 中的 `Element` 会被逐个执行 `notifyDependent` ，最后触发 `markNeedsBuild` ，这也是为什么当 `InheritedWidget` 被更新时，通过如 `Theme.of(context).primaryColor` 引用的地方，也会触发更新的原因。

```
@override
void notifyClients(InheritedWidget oldWidget) {
  assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
  for (Element dependent in _dependents.keys) {
    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));
    notifyDependent(oldWidget, dependent);
  }
}
```

> 下面开始实际分析 **Provider** 。

二、Provider

为什么会有 ***Provider*** ？

因为 Flutter 与 React 技术栈的相似性，所以在 Flutter 中涌现了诸如 `flutter_redux` 、 `flutter_dva` 、 `flutter_mobx` 、 `fish_flutter` 等前端式的状态管理，它们大多比较复杂，而且需要对框架概念有一定理解。

而作为 Flutter 官方推荐的状态管理 `scoped_model` ，又因为其设计较为简单，有些时候不适用于复杂的场景。

所以在经历了一端坎坷之后，**今年 Google I/O 大会之后， Provider 成了 Flutter 官方新推荐的状态管理方式之一。**

它的特点就是： **不复杂，好理解，代码量不大的情况下，可以方便组合和控制刷新颗粒度** ， 而原 Google 官方仓库的状态管理 flutter-provide 已宣告GG ， provider 成了它的替代品。

```
⚠注意，`provider` 比 `flutter-provide` 多了个 `r`。
```

> 题外话：以前面试时，偶尔会被面试官问到“你的开源项目代码量也不多啊”这样的问题，每次我都会笑而不语，**虽然代码量能代表一些成果，但是我是十分反对用代码量来衡量贡献价值，这和你用加班时长来衡量员工价值有什么区别？**

0、演示代码

如下代码所示， 实现的是一个点击计数器，其中：

- `_ProviderPageState` 中使用 `MultiProvider` 提供了多个 `providers` 的支持。
- 在 `CountWidget` 中通过 `Consumer` 获取的 `counter` ，同时更新 `_ProviderPageState` 中的 `AppBar` 和 `CountWidget` 中的 `Text` 显示。

```
class _ProviderPageState extends State<ProviderPage> {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(builder: (_) => ProviderModel()),
      ],
      child: Scaffold(
        appBar: AppBar(
          title: LayoutBuilder(
            builder: (BuildContext context, BoxConstraints constraints) {
              var counter =  Provider.of<ProviderModel>(context);
              return new Text("Provider ${counter.count.toString()}");
            },
          )
        ),
        body: CountWidget(),
      ),
    );
  }
}

class CountWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<ProviderModel>(builder: (context, counter, _) {
      return new Column(
        children: <Widget>[
          new Expanded(child: new Center(child: new Text(counter.count.toString()))
),
          new Center(
            child: new FlatButton(
                onPressed: () {
                  counter.add();
                },
                color: Colors.blue,
                child: new Text("+")),
```

```
          )
        ],
      );
    });
  }
}

class ProviderModel extends ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void add() {
    _count++;
    notifyListeners();
  }
}
```

所以上述代码中，我们通过 `ChangeNotifierProvider` 组合了 `ChangeNotifier` (ProviderModel) 实现共享；利用了 `Provider.of` 和 `Consumer` 获取共享的 `counter` 状态；通过调用 `ChangeNotifier` 的 `notifyListeners();` 触发更新。

这里几个知识点是：

- 1、 **Provider** 的内部 `DelegateWidget` 是一个 `StatefulWidget` ，所以可以更新且具有生命周期。
- 2、状态共享是使用了 `InheritedProvider` 这个 `InheritedWidget` 实现的。
- 3、巧妙利用 `MultiProvider` 和 `Consumer` 封装，实现了组合与刷新颗粒度控制。

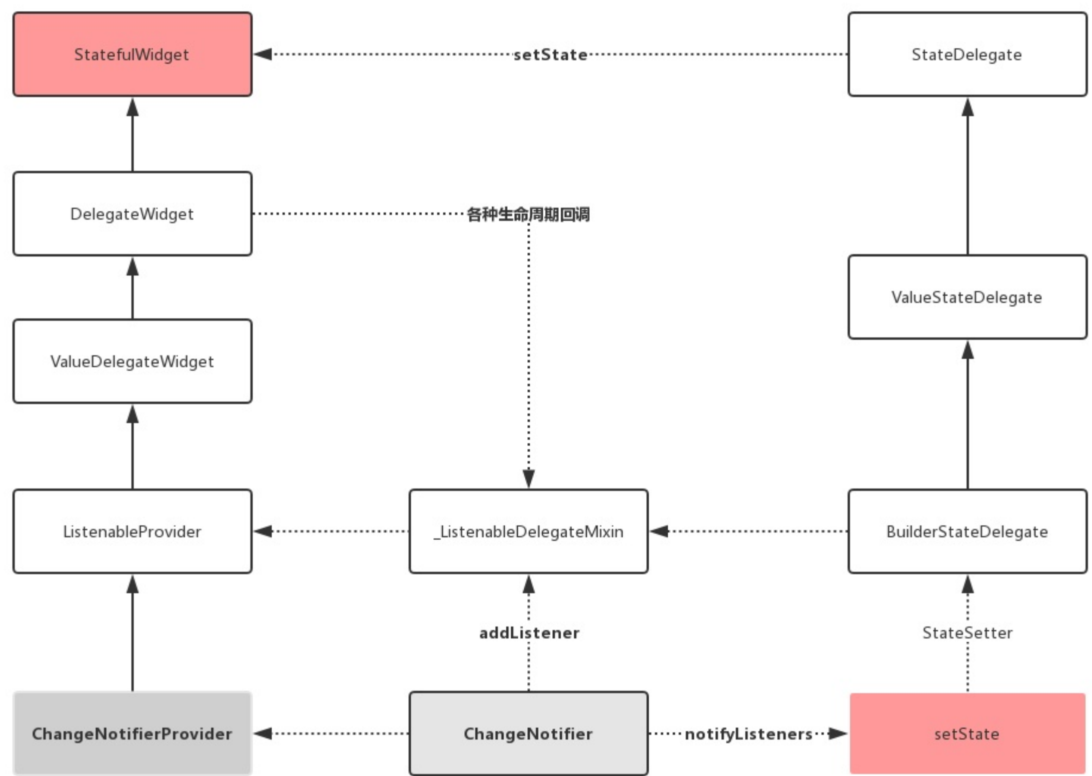
接着我们逐个分析

1、Delegate

既然是状态管理，那么肯定有 `StatefulWidget` 和 `setState` 调用。

在 **Provider** 中，一系列关于 `StatefulWidget` 的生命周期管理和更新，都是通过各种代理完成的，如下图所示，上面代码中我们用到的 `ChangeNotifierProvider` 大致经历了这样的流程：

- 设置到 `ChangeNotifierProvider` 的 `ChangeNotifer` 会被执行 `addListener` 添加监听 `listener` 。
- `listener` 内会调用 `StateDelegate` 的 `StateSetter` 方法，从而调用到 `StatefulWidget` 的 `setState` 。
- 当我们执行 `ChangeNotifer` 的 `notifyListeners` 时，就会最终触发 `setState` 更新。



而我们使用过的 `MultiProvider` 则是允许我们组合多种 `Provider` ，如下代码所示，传入的 `providers` 会倒序排列，最后组合成一个嵌套的 Widget tree ，方便我们添加多种 `Provider` ：

```
@override
Widget build(BuildContext context) {
  var tree = child;
  for (final provider in providers.reversed) {
    tree = provider.cloneWithChild(tree);
  }
  return tree;
}

/// Clones the current provider with a new [child].
/// Note for implementers: all other values, including [Key] must be
/// preserved.
@override
MultiProvider cloneWithChild(Widget child) {
  return MultiProvider(
    key: key,
    providers: providers,
    child: child,
  );
}
```

通过 `Delegate` 中回调出来的各种生命周期，如 `Disposer` ，也有利于我们外部二次处理，减少外部 `StatefulWidget` 的嵌套使用。

2、InheritedProvider

状态共享肯定需要 `InheritedWidget` ， `InheritedProvider` 就是 `InheritedWidget` 的子类，所有的 `Provider` 实现都在 `build` 方法中使用 `InheritedProvider` 进行嵌套，实现 `value` 的共享。

3、Consumer

`Consumer` 是 `Provider` 中比较有意思的东西，它本身是一个 `StatelessWidget` ，只是在 `build` 中通过 `Provider.of<T>(context)` 帮你获取到 `InheritedWidget` 共享的 `value` 。

```
  final Widget Function(BuildContext context, T value, Widget child) builder;

 @override
  Widget build(BuildContext context) {
    return builder(
      context,
      Provider.of<T>(context),
      child,
    );
  }
```

那我们直接使用 `Provider.of<T>(context)` ，不使用 `Consumer` 可以吗?

当然可以，但是你还记得前面，我们在介绍 `InheritedWidget` 时所说的：

> 传入的 `context` 代表着这个 `Widget` 的 `Element` 在 `InheritedElement` 里被“登记”到 `_dependents` 了。

`Consumer` 做为一个单独 `StatelessWidget` ，**它的好处就是 `Provider.of<T>(context)` 传入的 `context` 就是 `Consumer` 它自己。** 这样的话，我们在需要使用 `Provider.value` 的地方用 `Consumer` 做嵌套， `InheritedWidget` 更新的时候，就不会更新到整个页面，而是仅更新到 `Consumer` 这个 `StatelessWidget` 。

所以 `Consumer` 贴心的封装了 `context` 在 `InheritedWidget` 中的“登记逻辑”，从而控制了状态改变时，需要更新的精细度。

同时库内还提供了 `Consumer2` ～ `Consumer6` 的组合，感受下：

```

  @override
  Widget build(BuildContext context) {
    return builder(
      context,
      Provider.of<A>(context),
```

```
        Provider.of<B>(context),
        Provider.of<C>(context),
        Provider.of<D>(context),
        Provider.of<E>(context),
        Provider.of<F>(context),
        child,
      );
```

这样的设定，相信用过 BLoC 模式的同学会感觉很贴心，以前正常用做 BLoC 时，每个 `StreamBuilder` 的 `snapShot` 只支持一种类型，多个时要不就是多个状态合并到一个实体，要不就需要多个*StreamBuilder*嵌套。

当然，如果你想直接利用 `LayoutBuilder` 搭配 `Provider.of<T>(context)` 也是可以的：

```
LayoutBuilder(
            builder: (BuildContext context, BoxConstraints constraints) {
              var counter =  Provider.of<ProviderModel>(context);
              return new Text("Provider ${counter.count.toString()}");
            }
```

其他的还有 `ValueListenableProvider` 、 `FutureProvider` 、 `StreamProvider` 等多种 `Provider` ，可见整个 **Provider** 的设计上更贴近 Flutter 的原生特性，同时设计也更好理解，并且兼顾了性能等问题。

Provider 的使用指南上，更详细的 Vadaski 的 《Flutter | 状态管理指南篇——Provider》 已经写过，我就不重复写轮子了，感兴趣的可以过去看看。

> 自此，第十五篇终于结束了！(///▽///)

资源推荐

- 本文Demo ： https://github.com/CarGuo/state_manager_demo
- Github ： https://github.com/CarGuo/
- **开源 Flutter 完整项目： https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目： https://github.com/CarGuo/GSYFlutterBook**

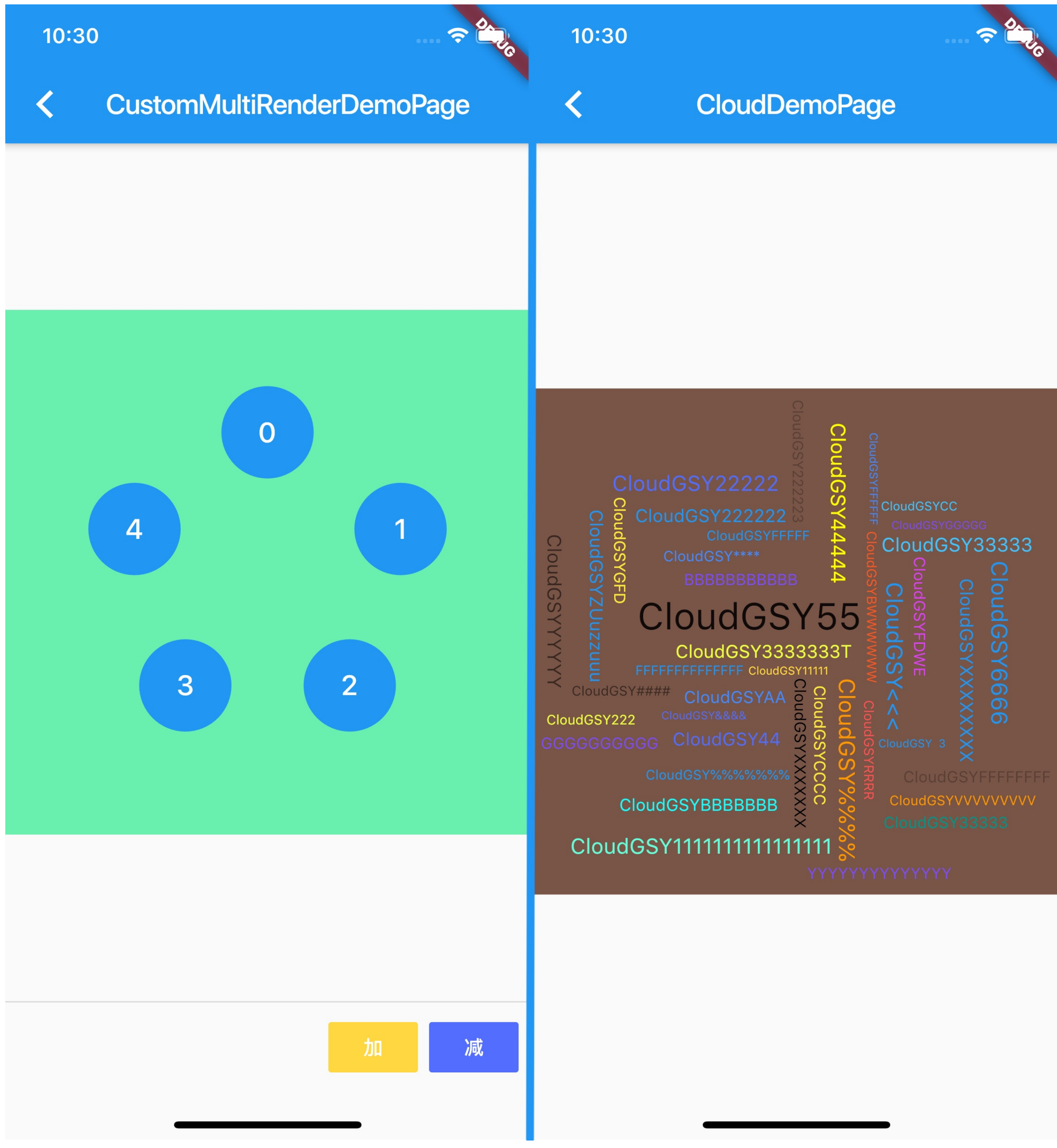
完整开源项目推荐：

- GSY Flutter 实战系列电子书
- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

本篇将解析 Flutter 中自定义布局的原理，并带你深入实战自定义布局的流程，利用两种自定义布局的实现方式，完成如下图所示的界面效果，看完这一篇你将可以更轻松的对 Flutter 为所欲为。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏



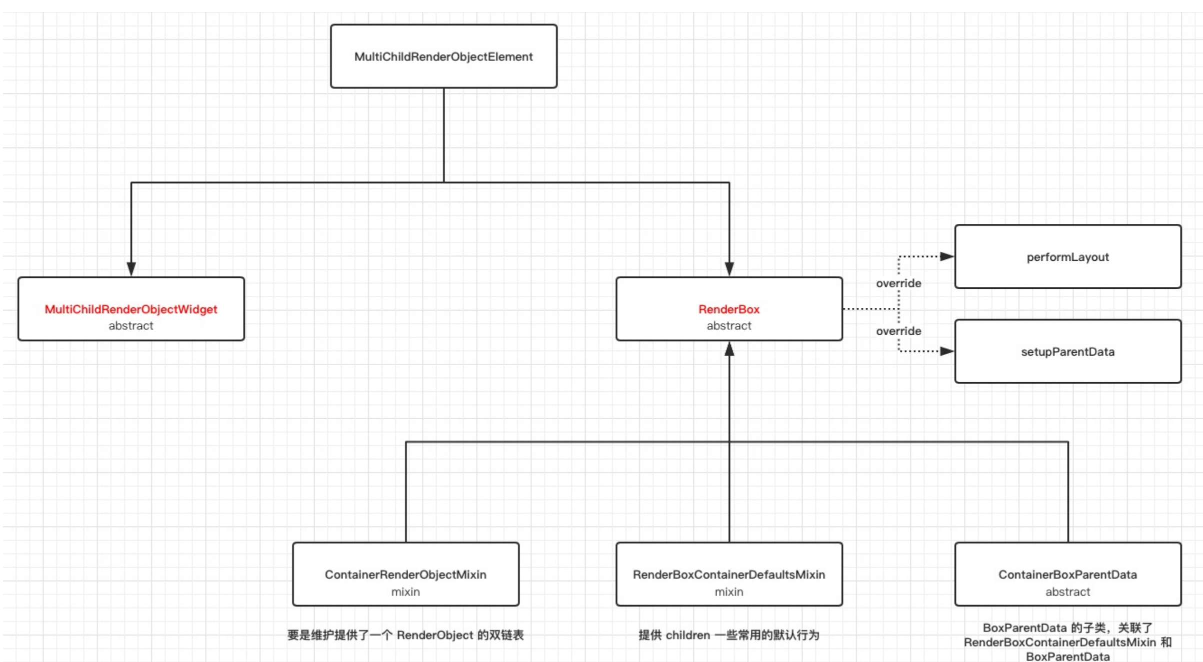
一、前言

在之前的篇章我们讲过 `Widget` 、 `Element` 和 `RenderObject` 之间的关系，所谓的 **自定义布局，事实上就是自定义 `RenderObject` 内 `child` 的大小和位置** ，而在这点上和其他框架不同的是，在 Flutter 中布局的核心并不是嵌套堆叠，**Flutter 布局的核心是在于 `Canvas`** ，我们所使用的 `Widget` ，仅仅是为了简化 `RenderObject` 的操作。

> 在《九、 深入绘制原理》的测试绘制 中我们知道， 对于 Flutter 而言，整个屏幕都是一块画布，我们通过各种 `Offset` 和 `Rect` 确定了位置，然后通过 `Canvas` 绘制 UI，而整个屏幕区域都是绘制目标，如果在 `child` 中我们 "不按照套路出牌" ，我们甚至可以不管 `parent` 的大小和位置随意绘制。

二、MultiChildRenderObjectWidget

了解基本概念后，我们知道 **自定义 `Widget` 布局的核心在于自定义 `RenderObject`** ，而在官方默认提供的布局控件里，大部分的布局控件都是通过继承 `MultiChildRenderObjectWidget` 实现，那么一般情况下自定义布局时，我们需要做什么呢?



如上图所示，一般情况下实现自定义布局，**我们会通过继承 `MultiChildRenderObjectWidget` 和 `RenderBox` 这两个 `abstract` 类实现，而 `MultiChildRenderObjectElement` 则负责关联起它们，** 除了此之外，还有有几个关键的类 ： `ContainerRenderObjectMixin` 、 `RenderBoxContainerDefaultsMixin` 和 `ContainerBoxParentData` 。

`RenderBox` 我们知道是 `RenderObject` 的子类封装，也是我们自定义 `RenderObject` 时经常需要继承的，那么其他的类分别是什么含义呢?

1、ContainerRenderObjectMixin

故名思义，这是一个 `mixin` 类， `ContainerRenderObjectMixin` 的作用，**主要是维护提供了一个双链表的 children `RenderObject`** 。

通过在 `RenderBox` 里混入 `ContainerRenderObjectMixin` ， 我们就可以得到一个双链表的 children ， 方便在我们布局时，**可以正向或者反向去获取和管理 `RenderObject` 们** 。

2、RenderBoxContainerDefaultsMixin

`RenderBoxContainerDefaultsMixin` 主要是对 `ContainerRenderObjectMixin` 的拓展，是对 `ContainerRenderObjectMixin` 内的 children 提供常用的默认行为和管理，接口如下所示：

```
/// 计算返回第一个 child 的基线 ，常用于 child 的位置顺序有关
double defaultComputeDistanceToFirstActualBaseline(TextBaseline baseline)

/// 计算返回所有 child 中最小的基线，常用于 child 的位置顺序无关
double defaultComputeDistanceToHighestActualBaseline(TextBaseline baseline)

/// 触摸碰撞测试
bool defaultHitTestChildren(BoxHitTestResult result, { Offset position })

/// 默认绘制
void defaultPaint(PaintingContext context, Offset offset)

/// 以数组方式返回 child 链表
List<ChildType> getChildrenAsList()
```

3、ContainerBoxParentData

`ContainerBoxParentData` 是 `BoxParentData` 的子类，主要是关联了 `ContainerDefaultsMixin` 和 `BoxParentData` ， **`BoxParentData` 是 `RenderBox` 绘制时所需的位置类。**

通过 `ContainerBoxParentData` ，我们可以将 `RenderBox` 需要的 `BoxParentData` 和上面的 `ContainerParentDataMixin` 组合起来，**事实上我们得到的 children 双链表就是以 `ParentData` 的形式呈现出来的。**

```
abstract class ContainerBoxParentData<ChildType extends RenderObject> extends BoxPa
rentData with ContainerParentDataMixin<ChildType> { }
```

4、MultiChildRenderObjectWidget

`MultiChildRenderObjectWidget` 的实现很简单 ，它仅仅只是继承了 `RenderObjectWidget` ，然后提供了 `children` 数组，并创建了 `MultiChildRenderObjectElement` 。

> 上面的 `RenderObjectWidget` 顾名思义，它是提供 `RenderObject` 的 `Widget` ，那有不存在 `RenderObject` 的 `Widget` 吗？
>
> 有的，比如我们常见的 `StatefulWidget` 、 `StatelessWidget` 、 `Container` 等，它们的 `Element` 都是 `ComponentElement` ， `ComponentElement` 仅仅起到容器的作用，而它的 `get renderObject` 需要来自它的 `child` 。

5、MultiChildRenderObjectElement

前面的篇章我们说过 `Element` 是 `BuildContext` 的实现， 内部一般持有 `Widget` 、 `RenderObject` 并作为二者沟通的桥梁，那么 `MultiChildRenderObjectElement` 就是我们自定义布局时的桥梁了， 如下代码所示， `MultiChildRenderObjectElement` 主要实现了如下接口，其主要功能是对内部 `children` 的 `RenderObject` ，实现了插入、移除、访问、更新等逻辑：

```
    /// 下面三个方法都是利用 ContainerRenderObjectMixin 的 insert/move/remove 去操作
    /// ContainerRenderObjectMixin<RenderObject, ContainerParentDataMixin<RenderObj
ect>
    void insertChildRenderObject(RenderObject child, Element slot)
    void moveChildRenderObject(RenderObject child, dynamic slot)
    void removeChildRenderObject(RenderObject child)

    /// visitChildren 是通过 Element 中的 ElementVisitor 去迭代的
    /// 一般在 RenderObject get renderObject 会调用
    void visitChildren(ElementVisitor visitor)

    /// 添加忽略child _forgottenChildren.add(child);
    void forgetChild(Element child)

    /// 通过 inflateWidget ， 把 children 中 List<Widget> 对应的 List<Element>
    void mount(Element parent, dynamic newSlot)

    /// 通过 updateChildren 方法去更新得到  List<Element>
    void update(MultiChildRenderObjectWidget newWidget)
```

所以 `MultiChildRenderObjectElement` 利用 `ContainerRenderObjectMixin` 最终将我们自定义的 `RenderBox` 和 `Widget` 关联起来。

6、自定义流程

上述主要描述了 `MultiChildRenderObjectWidget` 、 `MultiChildRenderObjectElement` 和其他三个辅助类 `ContainerRenderObjectMixin` 、 `RenderBoxContainerDefaultsMixin` 和 `ContainerBoxParentData` 之间的关系。

了解几个关键类之后，我们看一般情况下，实现自定义布局的简化流程是：

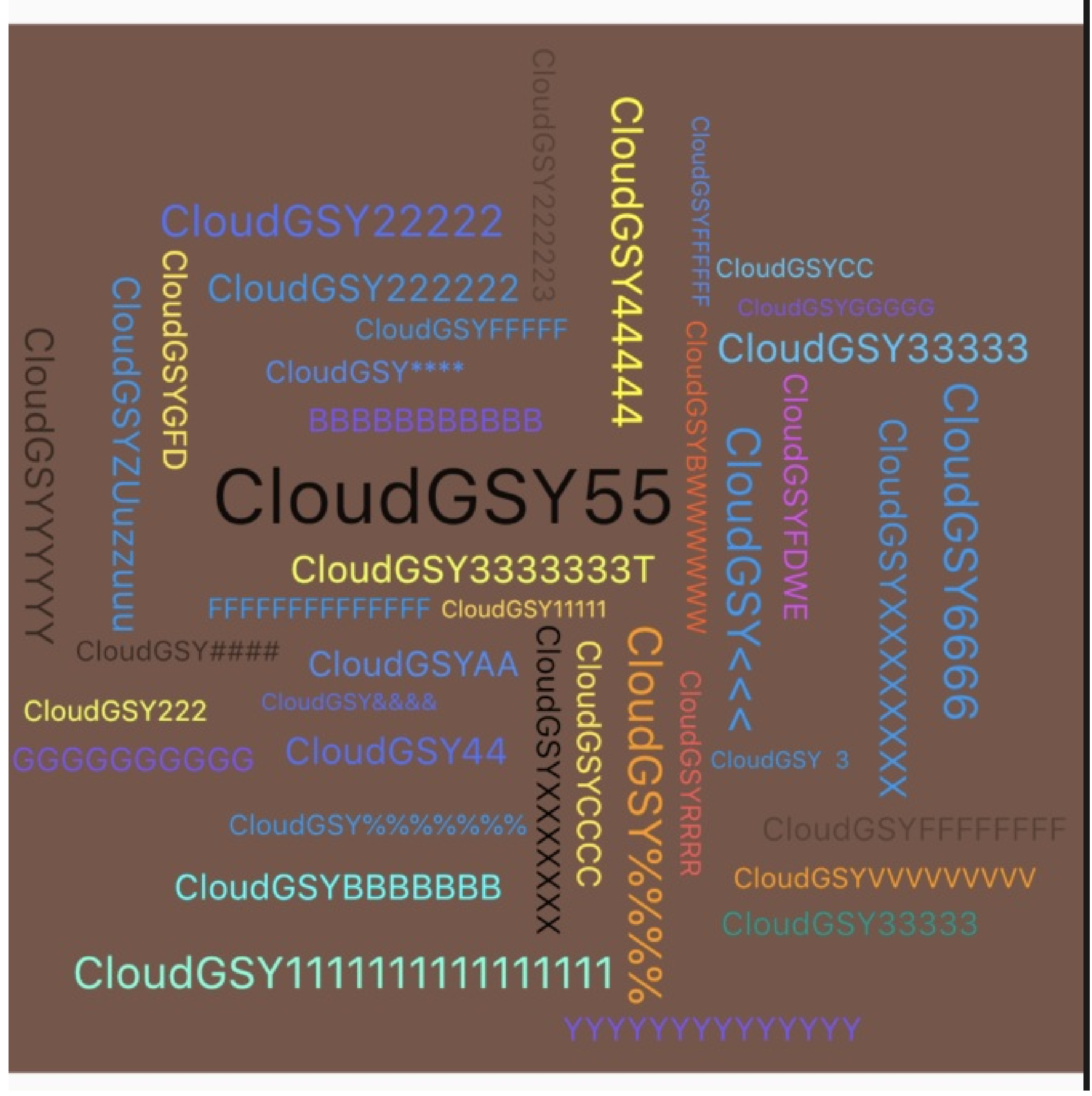
- 1、自定义 `ParentData` 继承 `ContainerBoxParentData` 。
- 2、继承 `RenderBox` ，同时混入 `ContainerRenderObjectMixin` 和 `RenderBoxContainerDefaultsMixin` 实现自定义 `RenderObject` 。
- 3、继承 `MultiChildRenderObjectWidget` ，实现 `createRenderObject` 和 `updateRenderObject` 方法，关联我们自定义的 `RenderBox` 。
- 4、override `RenderBox` 的 `performLayout` 和 `setupParentData` 方法，实现自定义布局。

当然我们可以利用官方的 `CustomMultiChildLayout` 实现自定义布局，这个后面也会讲到，现在让我们先从基础开始， 而上述流程中混入的 `ContainerRenderObjectMixin` 和 `RenderBoxContainerDefaultsMixin` ，在 `RenderFlex` 、 `RenderWrap` 、 `RenderStack` 等官方

实现的布局里，也都会混入它们。

三、自定义布局

自定义布局就是在 `performLayout` 中实现的 `child.layout` 大小和 `child.ParentData.offset` 位置的赋值。



首先我们要实现类似如图效果，我们需要自定义 `RenderCloudParentData` 继承 `ContainerBoxParentData` ，用于记录宽高和内容区域：

```
class RenderCloudParentData extends ContainerBoxParentData<RenderBox> {
  double width;
  double height;

  Rect get content => Rect.fromLTWH(
```

```
        offset.dx,
        offset.dy,
        width,
        height,
      );
}
```

然后自定义 `RenderCloudWidget` 继承 `RenderBox` ，并混入 `ContainerRenderObjectMixin` 和 `RenderBoxContainerDefaultsMixin` 实现 `RenderBox` 自定义的简化。

```
class RenderCloudWidget extends RenderBox
    with
        ContainerRenderObjectMixin<RenderBox, RenderCloudParentData>,
        RenderBoxContainerDefaultsMixin<RenderBox, RenderCloudParentData> {
  RenderCloudWidget({
    List<RenderBox> children,
    Overflow overflow = Overflow.visible,
    double ratio,
  })  : _ratio = ratio,
        _overflow = overflow {
   ///添加所有 child
    addAll(children);
  }
```

如下代码所示，接下来主要看 `RenderCloudWidget` 中 `override performLayout` 中的实现，这里我们只放关键代码：

- 1、我们首先拿到 `ContainerRenderObjectMixin` 链表中的 `firstChild` ，然后从头到位读取整个链表。
- 2、对于每个 child 首先通过 `child.layout` 设置他们的大小，然后记录下大小之后。
- 3、以容器控件的中心为起点，从内到外设置布局，这是设置的时候，需要通过记录的 `Rect` 判断是否会重复，每次布局都需要计算位置，直到当前 child 不在重复区域内。
- 4、得到最终布局内大小，然后设置整体居中。

```
///设置为我们的数据
@override
void setupParentData(RenderBox child) {
  if (child.parentData is! RenderCloudParentData)
    child.parentData = RenderCloudParentData();
}

@override
  void performLayout() {
    ///默认不需要裁剪
    _needClip = false;

    ///没有 childCount 不玩
    if (childCount == 0) {
```

```
    size = constraints.smallest;
    return;
  }

  ///初始化区域
  var recordRect = Rect.zero;
  var previousChildRect = Rect.zero;

  RenderBox child = firstChild;

  while (child != null) {
    var curIndex = -1;

    ///提出数据
    final RenderCloudParentData childParentData = child.parentData;

    child.layout(constraints, parentUsesSize: true);

    var childSize = child.size;

    ///记录大小
    childParentData.width = childSize.width;
    childParentData.height = childSize.height;

    do {
      ///设置 xy 轴的比例
      var rX = ratio >= 1 ? ratio : 1.0;
      var rY = ratio <= 1 ? ratio : 1.0;

      ///调整位置
      var step = 0.02 * _mathPi;
      var rotation = 0.0;
      var angle = curIndex * step;
      var angleRadius = 5 + 5 * angle;
      var x = rX * angleRadius * math.cos(angle + rotation);
      var y = rY * angleRadius * math.sin(angle + rotation);
      var position = Offset(x, y);

      ///计算得到绝对偏移
      var childOffset = position - Alignment.center.alongSize(childSize);

      ++curIndex;

      ///设置为遏制
      childParentData.offset = childOffset;

      ///判处是否交叠
    } while (overlaps(childParentData));

    ///记录区域
    previousChildRect = childParentData.content;
```

```
      recordRect = recordRect.expandToInclude(previousChildRect);

      ///下一个
      child = childParentData.nextSibling;
    }

    ///调整布局大小
    size = constraints
        .tighten(
          height: recordRect.height,
          width: recordRect.width,
        )
        .smallest;

    ///居中
    var contentCenter = size.center(Offset.zero);
    var recordRectCenter = recordRect.center;
    var transCenter = contentCenter - recordRectCenter;
    child = firstChild;
    while (child != null) {
      final RenderCloudParentData childParentData = child.parentData;
      childParentData.offset += transCenter;
      child = childParentData.nextSibling;
    }

    ///超过了嘛?
    _needClip =
        size.width < recordRect.width || size.height < recordRect.height;
  }
```

其实看完代码可以发现，关键就在于你怎么设置 `child.parentData` 的 `offset` ，来控制其位置。

最后通过 `CloudWidget` 加载我们的 `RenderCloudWidget` 即可， 当然完整代码还需要结合 `FittedBox` 与 `RotatedBox` 简化完成，具体可见： GSYFlutterDemo

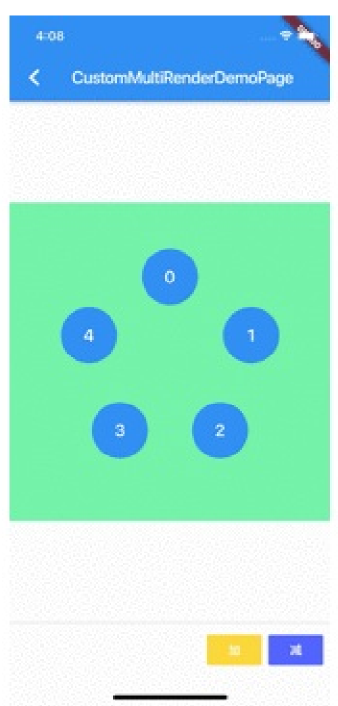
```
class CloudWidget extends MultiChildRenderObjectWidget {
  final Overflow overflow;
  final double ratio;

  CloudWidget({
    Key key,
    this.ratio = 1,
    this.overflow = Overflow.clip,
    List<Widget> children = const <Widget>[],
  }) : super(key: key, children: children);

  @override
  RenderObject createRenderObject(BuildContext context) {
    return RenderCloudWidget(
      ratio: ratio,
```

```
      overflow: overflow,
    );
  }

  @override
  void updateRenderObject(
      BuildContext context, RenderCloudWidget renderObject) {
    renderObject
      ..ratio = ratio
      ..overflow = overflow;
  }
}
```

最后我们总结，实现自定义布局的流程就是，实现自定义 `RenderBox` 中 `performLayout` child 的 `offset` 。

四、CustomMultiChildLayout

`CustomMultiChildLayout` 是 Flutter 为我们封装的简化自定义布局实现，**它的内部同样是通过 `MultiChildRenderObjectWidget` 实现，但是它为我们封装了 `RenderCustomMultiChildLayoutBox` 和 `MultiChildLayoutParentData` ，并通过 `MultiChildLayoutDelegate` 暴露出需要自定义的地方。**



使用 `CustomMultiChildLayout` 你只需要继承 `MultiChildLayoutDelegate` ，并实现如下方法即可：

```
void performLayout(Size size);
```

```
  bool shouldRelayout(covariant MultiChildLayoutDelegate oldDelegate);
```

通过继承 `MultiChildLayoutDelegate` ，并且实现 `performLayout` 方法，我们可以快速自定义我们需要的控件，当然便捷的封装也代表了灵活性的丧失，可以看到 `performLayout` 方法中只有布局自身的 `Size` 参数，所以完成上图需求时，**我们还需要 child 的大小和位置**，也就是 `childSize` 和 `childId` 。

`childSize` 相信大家都能故名思义，那 `childId` 是什么呢?

这就要从 `MultiChildLayoutDelegate` 的实现说起，**在 `MultiChildLayoutDelegate` 内部会有一个 `Map<Object, RenderBox> _idToChild;` 对象，这个 `Map` 对象保存着 `Object id` 和 `RenderBox` 的映射关系，而在 `MultiChildLayoutDelegate` 中获取 `RenderBox` 都需要通过 `id` 获取。**

`_idToChild` 这个 `Map` 是在 `RenderBox performLayout` 时，在 `delegate._callPerformLayout` 方法内创建的，创建后所用的 `id` 为 `MultiChildLayoutParentData` 中的 id，**而 `MultiChildLayoutParentData` 的 id ，可以通过 `LayoutId` 嵌套时自定义指定赋值。**

而完成上述布局，我们需要知道每个 child 的 index ，所以我们可以把 index 作为 id 设置给每个 child 的 `LayoutId` 。

所以我们可以通过 `LayoutId` 指定 id 为数字 index ， 同时告知 delegate ，这样我们就知道 child 顺序和位置啦。

> 这个 id 是 `Object` 类型 ，所以你懂得，你可以赋予很多属性进去。

如下代码所示，这样在自定义的 `CircleLayoutDelegate` 中，就知道每个控件的 `index` 位置，也就是知道了，圆形布局中每个 item 需要的位置。

我们只需要通过 `index` ，计算出 child 所在的角度，然后利用 `layoutChild` 和 `positionChild` 对每个item进行布局即可，完整代码:GSYFlutterDemo

```
///自定义实现圆形布局
class CircleLayoutDelegate extends MultiChildLayoutDelegate {
  final List<String> customLayoutId;

  final Offset center;

  Size childSize;

  CircleLayoutDelegate(this.customLayoutId,
      {this.center = Offset.zero, this.childSize});

  @override
  void performLayout(Size size) {
    for (var item in customLayoutId) {
      if (hasChild(item)) {
        double r = 100;
```

```
          int index = int.parse(item);

          double step = 360 / customLayoutId.length;

          double hd = (2 * math.pi / 360) * step * index;

          var x = center.dx + math.sin(hd) * r;

          var y = center.dy - math.cos(hd) * r;

          childSize ??= Size(size.width / customLayoutId.length,
              size.height / customLayoutId.length);

          ///设置 child 大小
          layoutChild(item, BoxConstraints.loose(childSize));

          final double centerX = childSize.width / 2.0;

          final double centerY = childSize.height / 2.0;

          var result = new Offset(x - centerX, y - centerY);

          ///设置 child 位置
          positionChild(item, result);
        }
      }
    }

    @override
    bool shouldRelayout(MultiChildLayoutDelegate oldDelegate) => false;
  }
```

总的来说，第二种实现方式相对简单，但是也丧失了一定的灵活性，可自定义控制程度更低，但是也更加规范与间接，同时我们自己实现 `RenderBox` 时，也可以用类似的 delegate 的方式做二次封装，这样的自定义布局会更行规范可控。

> 自此，第十六篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp

作为系列文章的第十七篇，本篇再一次带来 Flutter 开发过程中的实用技巧，让你继续弯道超车，全篇均为个人的日常干货总结，以实用填坑为主，让你少走弯路狂飙车。

文章汇总地址：

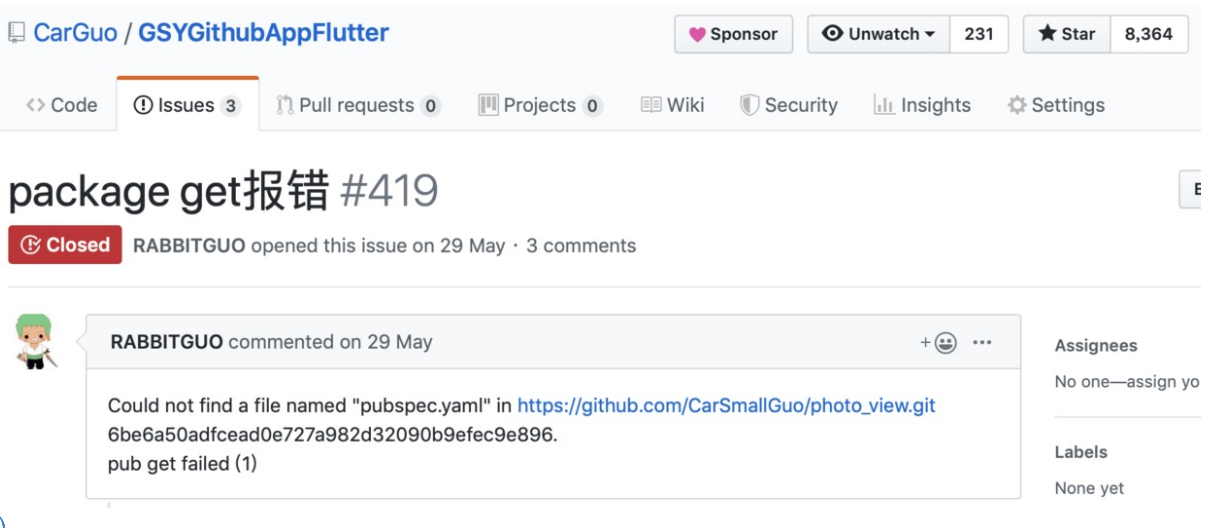
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

1、Package get git 失败

Flutter 项目在引用第三库时，一般都是直接引用 `pub` 上的第三方插件，但是有时候我们为了安全和私密，会选择使用 git 引用，如：

```
photo_view:
  git:
    url: https://github.com/CarSmallGuo/photo_view.git
    ref: master
```

这时候在执行 `flutter packages get` 过程中，如果出现失败后，再次执行 `flutter packages get` 可能会遇到如下图所示的问题：



)

而 `flutter packages get` 提示 `git` 失败的原因，主要是：

在下载包的过程中出现问题，下次再拉包的时候，**在 `.pub_cache` 内的 `git` 目录下会检测到已经存在目录，但是可能是空目录等等，导致 `flutter packages get` 的时候异常。**

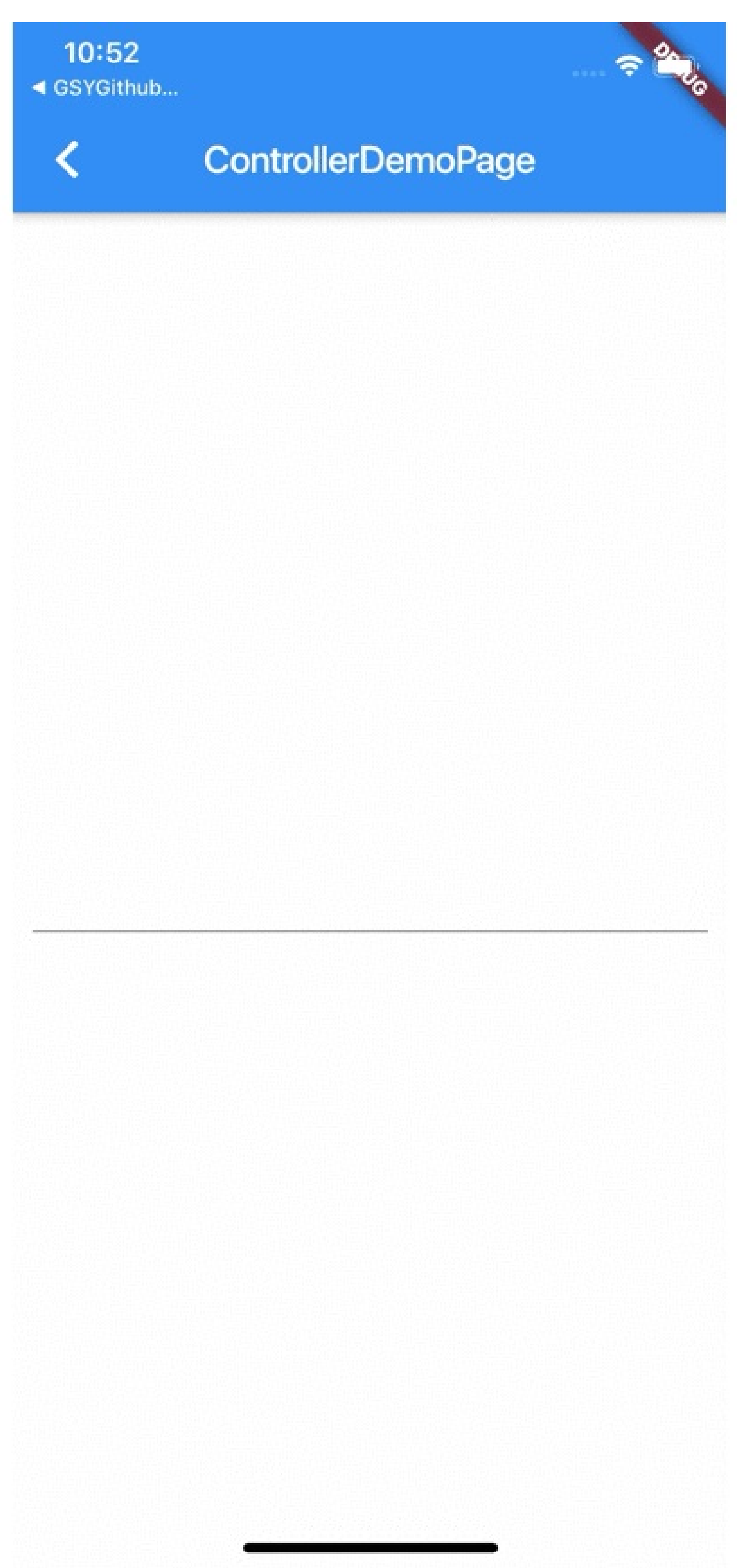
所以你需要清除掉 `.pub_cache` 内的 `git` 的异常目录，然后最好清除掉项目下的 `pubspec.lock` ，之后重新执行 `flutter packages get` 。

> `win` 一般是在 `C:\Users\xxxxx\AppData\Roaming\Pub\Cache` 路径下有 `git` 目录。
>
> `mac` 目录在 `~/.pub-cache` 。

2、TextEditingController

```
TextEditingController controller;
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: new Text("ControllerDemoPage"),
    ),
    body: GestureDetector(
      behavior: HitTestBehavior.translucent,
      onTap: () {
        FocusScope.of(context).requestFocus(new FocusNode());
      },
      child: new Container(
        margin: EdgeInsets.all(10),
        child: new Center(
          child: new TextField(
            controller: controller ?? TextEditingController(),
          ),
        ),
      ),
    ),
  );
}
```

如上代码所示，红线部分表示，如果 `controller` 为空，就赋值一个 `TextEditingController` ，这样的写法会导致如下图所示问题：



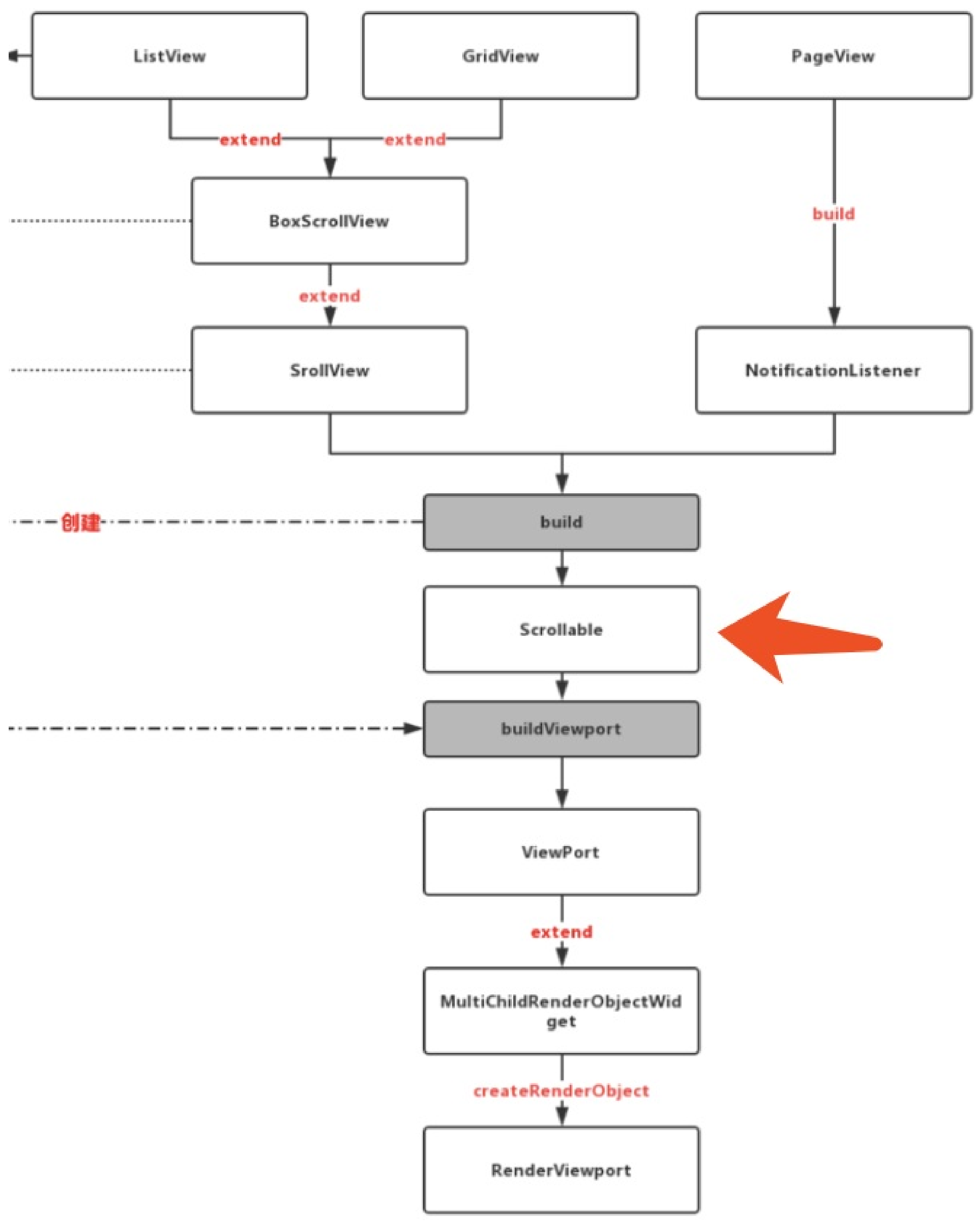
弹出键盘时输入成功后，收起键盘时输入的内容消失了！ 这是因为键盘的弹出和收起都会触发页面 `build` ，而在 `controller` 为 `null` 时，每次赋值的 `TextEditingController` 会导致 `TextField` 的 `TextEditingValue` 重置。

```
@override
void initState() {
  super.initState();
  if (widget.controller == null)
    _controller = TextEditingController();
}

@override
void didUpdateWidget(TextField oldWidget) {
  super.didUpdateWidget(oldWidget);
  if (widget.controller == null && oldWidget.controller != null)
    _controller = TextEditingController.fromValue(oldWidget.controller.value);
  else if (widget.controller != null && oldWidget.controller == null)
    _controller = null;
  final bool isEnabled = widget.enabled ?? widget.decoration?.enabled ?? true;
  final bool wasEnabled = oldWidget.enabled ?? oldWidget.decoration?.enabled ?? true;
  if (wasEnabled && !isEnabled) {
    _effectiveFocusNode.unfocus();
  }
  if (_effectiveFocusNode.hasFocus && widget.readOnly != oldWidget.readOnly) {
    if(_effectiveController.selection.isCollapsed) {
      _showSelectionHandles = !widget.readOnly;
    }
  }
}
```

如上图所示，因为当 `TextField` 的 `controller` 不为空时，update 时是不会执行 `value` 的拷贝，所以为了避免这类问题，如下图所示， **需要先在全局构建 `TextEditingController` 再赋值，如果 `controller` 为空直接给 null 即可，避免 `build` 时每次重构 `TextEditingController`** 。

```dart
class ControllerDemoPage extends StatefulWidget {
  @override
  _ControllerDemoPageState createState() => _ControllerDemoPageState();
}

class _ControllerDemoPageState extends State<ControllerDemoPage> {

  /// State 中可以跨 Widget 保存 controller
  final TextEditingController controller = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: new Text("ControllerDemoPage"),
      ),
      body: GestureDetector(
        behavior: HitTestBehavior.translucent,
        onTap: () {
          FocusScope.of(context).requestFocus(new FocusNode());
        },
        child: new Container(
          margin: EdgeInsets.all(10),
          child: new Center(
            child: new TextField(
              controller: controller,
            ),
          ),
        ),
      ),
    );
  }
}
```

3、Scrollable



如上图所示，在之前第七篇的时候分析过，**滑动列表内一般都会有 `Scrollable` 的存在，而 `Scrollable` 恰好是一个 `InheritedWidget`** ，这就给我们在 `children` 中调用 `Scrollable` 相关方法提供了便利。

如下代码所依，通过 `Scrollable.of(context)` 我们可以更解耦的在 `ListView/GridView` 的 `children` 对其进行控制。

```
ScrollableState state = Scrollable.of(context)
```

```
///获取 _scrollable 内 viewport 的 renderObject
RenderObject renderObject = state.context.findRenderObject();
///监听位置更新
state.position.addListener((){});
///通知位置更新
state.position.notifyListeners();
///滚动到指定位置
state.position.jumpTo(1000);
....
```

4、图片高斯模糊

10:40
DEBUG
BlurDemoPage
哇！！

在 Flutter 中，提供了 `BackdropFilter` 和 `ImageFilter` 实现了高斯模糊的支持，如下代码所示，可以快速实现上图的高斯模糊效果。

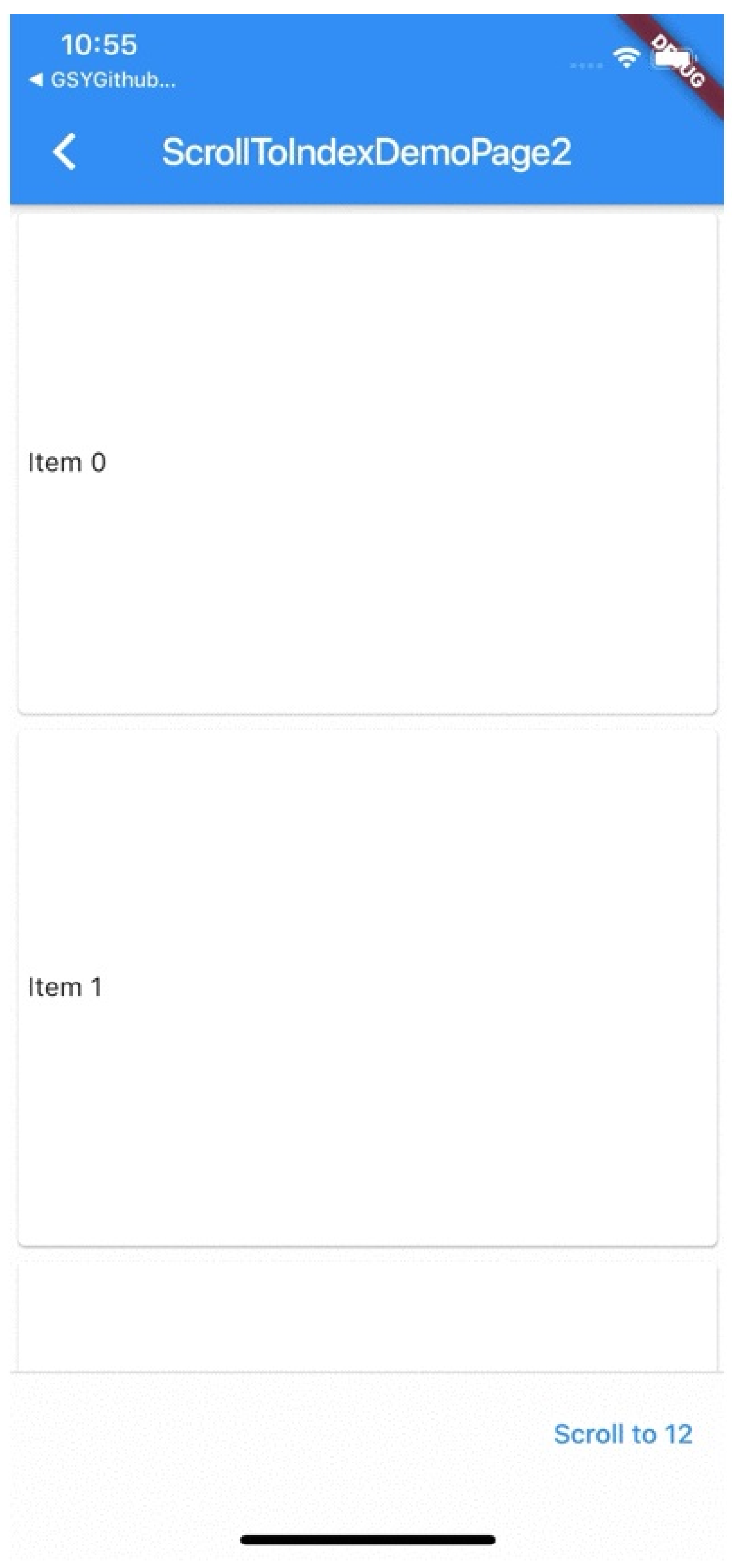
```
class BlurDemoPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: new Container(
        child: Stack(
          children: <Widget>[
            Positioned(
              top: 0,
              bottom: 0,
              left: 0,
              right: 0,
              child: new Image.asset(
                "static/gsy_cat.png",
                fit: BoxFit.cover,
                width: MediaQuery.of(context).size.width,
                height: MediaQuery.of(context).size.height,
              )),
            new Center(
              child: new Container(
                width: 200,
                height: 200,
                child: ClipRRect(
                  borderRadius: BorderRadius.circular(15.0),
                  child: BackdropFilter(
                    filter: ImageFilter.blur(sigmaX: 8.0, sigmaY: 8.0),
                    child: new Row(
                      mainAxisSize: MainAxisSize.max,
                      crossAxisAlignment: CrossAxisAlignment.center,
                      mainAxisAlignment: MainAxisAlignment.center,
                      children: <Widget>[
                        new Icon(Icons.ac_unit),
                        new Text("哇! ! ")
                      ],
                    )))))
          ],
        )));
  }
}
```

5、滚动到指定位置

因为目前 Flutter 并没有直接提供滚动到指定 `Item` 的方法，在每个 `Item` 大小不一的情况下，折中利用如图下所示代码，可以快速实现滚动到指定 `Item` 的效果：

```
_scrollToIndex() {
  var data = dataList[12]; ←保存有GlobalKey的数据列表

  ///获取 renderBox
  RenderBox renderBox =
  data.globalKey.currentContext.findRenderObject();

  ///获取位置偏移，基于 ancestor: SingleChildScrollView 的 RenderObject()
  double dy = renderBox
      .localToGlobal(Offset.zero,
      ancestor: scrollKey.currentContext.findRenderObject())
      .dy;

  ///计算真实位移
  var offset = dy + controller.offset;

  controller.animateTo(offset,
      duration: Duration(milliseconds: 500), curve: Curves.linear);
}
```

上图为部分代码，完整代码可见 scroll_to_index_demo_page2.dart，这里主要是给每个 `item` 都赋予了一个 `GlobalKey`，利用 `findRenderObject` 找到所需 `item` 的 `RenderBox`，然后使用 `localToGlobal` 获取 `item` 在 `ViewPort` 这个 `ancestor` 中的偏移量进行滚动：



当然还有另外一种实现方式，具体可见 scroll_to_index_demo_page.dart

6、findRenderObject

在 Flutter 中是存在 **容器 Widget** 和 **渲染Widget** 的区别的，一般情况下:

- `Text` 、 `Sliver` 、 `ListTile` 等都是属于渲染 Widget，其内部主要是 `RenderObjectElement` 。
- `StatelessWidget` / `StatefulWidget` 等属于容器 Widget，其内部使用的是 `ComponentElement` ， **`ComponentElement` 本身是不存在 `RenderObject` 的。**

结合前面篇章我们说过 `BuildContext` 的实现就是 `Element` ，所以 `context.findRenderObject()` 这个操作其实就是 `Element` 的 `findRenderObject()` 。

```
/// The render object at (or below) this location in the tree.
///
/// If this object is a [RenderObjectElement], the render object is the one at
/// this location in the tree. Otherwise, this getter will walk down the tree
/// until it finds a [RenderObjectElement].
RenderObject get renderObject {
  RenderObject result;
  void visit(Element element) {
    assert(result == null); // this verifies that there's only one child
    if (element is RenderObjectElement)
      result = element.renderObject;
    else
      element.visitChildren(visit);
  }
  visit(this);
  return result;
}
```

那么如上图所示， `findRenderObject` 的实现最终就是获取 `renderObject` ，在 `Element` 中 `renderObject` 的获取逻辑就很清晰了，**在遇到 `ComponentElement` 时，执行的是 `element.visitChildren(visit);`** ，递归直到找到 `RenderObjectElement` 。

所以如下代码所示， `print("${globalKey.currentContext.findRenderObject()}");` 最终输出了 `SizedBox` 的 `RenderObject` 。

```
    return Scaffold(
      appBar: AppBar(
        title: new Text("ControllerDemoPage"),
      ),
      body: new Container(
        margin: EdgeInsets.all(10),
        child: new Center(
          child: new GlobalText(key: globalKey),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          print("${globalKey.currentContext.findRenderObject()}");
          /// 输出 RenderConstrainedBox#5a24b relayoutBoundary=up3
          /// RenderConstrainedBox 为 SizedBox 的 RenderObject
        },
        child: new Text("C"),
      ),
    );
  }
}

class GlobalText extends StatefulWidget {
  GlobalText({Key key}) :super(key: key);
  @override
  _GlobalTextState createState() => _GlobalTextState();
}

class _GlobalTextState extends State<GlobalText> {
  @override
  Widget build(BuildContext context) {
    return SizedBox(
      height: 100,
      width:100,
      child: new Text("FFFFF"),
    );
  }
```

7、行间距

10:39

◀ Sands Res...

DEBUG

TextLineHeightDemoPage

Today I was amazed to see the usually positive and friendly VueJS community descend into a bitter war. Two weeks ago Vue creator Evan You released a Request for Comment (RFC) for a new function-based way of writing Vue components in the upcoming Vue 3.0. Today a critical Reddit thread followed by similarly critical comments in a Hacker News thread caused a flood of developers to flock to the original RFC to voice their outrage, some of which were borderline abusive. It was claimed in various places that

在 Flutter 中，是没有直接设置 `Text` 行间距的方法的， `Text` 显示的效果是如下图所示的逻辑组成：

```
/// The vertical components of strut are as follows:
///
///  * `leading * fontSize / 2` or half the font leading if `leading` is undefined (half leading)
///  * `ascent * height`
///  * `descent * height`
///  * `leading * fontSize / 2` or half the font leading if `leading` is undefined (half leading)
///
/// The sum of these four values is the total height of the line.
///
/// The `ascent + descent` is equivalent to the [fontSize]. Ascent is the font's
/// spacing above the baseline without leading and descent is the spacing below the
/// baseline without leading. Leading is split evenly between the top and bottom.
/// The values for `ascent` and `descent` are provided by the font named by
/// [fontFamily]. If no [fontFamily] or [fontFamilyFallback] is provided, then the
/// platform's default family will be used.
```

那么我们应该如何处理行间距呢？如下图所示，**通过设置 `StrutStyle` 的 `leading` ，然后利用 `Transform` 做计算翻方向位置偏移，因为 `leading` 是上下均衡的，所以计算后就可以得到我们所需要的行间距大小。**（虽然无法保证一定 100%像素准确，你是否还知道其他方法？）

```
class TextLineHeightDemoPage extends StatelessWidget {

  final double leading = 0.9;

  final double fontSize = 16;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: new Text("TextLineHeightDemoPage"),
      ),
      body: Container(
        color: Colors.blueGrey,
        margin: EdgeInsets.all(20),

        ///利用 Transform 偏移将对应权重部分位置
        child: Transform.translate(
          offset: Offset(0, -fontSize * leading / 2),
          child: new Text(
            textContent,
            strutStyle:
                StrutStyle(forceStrutHeight: true, height: 1, leading: leading),
            style: TextStyle(
                fontSize: fontSize,
                color: Colors.black,
                //backgroundColor: Colors.greenAccent),
                ),
          ),
        ),
      ),
    );
  }
}
```

> 这里额外提一点，可以通过父节点使用 `DefaultTextStyle` 来实现局部样式的共享哦。

8、Builder

```dart
class Builder extends StatelessWidget {
  /// Creates a widget that delegates its build to a callback.
  ///
  /// The [builder] argument must not be null.
  const Builder({
    Key key,
    @required this.builder,
  }) : assert(builder != null),
       super(key: key);

  /// Called to obtain the child widget.
  ///
  /// This function is called whenever this widget is included in its parent's
  /// build and the old widget (if any) that it synchronizes with has a distinct
  /// object identity. Typically the parent's build method will construct
  /// a new tree of widgets and so a new Builder child will not be [identical]
  /// to the corresponding old one.
  final WidgetBuilder builder;

  @override
  Widget build(BuildContext context) => builder(context);
}
```

在 Flutter 中存在 `Builder` 这样一个 Widget，看源码发现它其实就是 `StatelessWidget` 的简单封装，那为什么还需要它的存在呢？

如下图所示，相信一些 Flutter 开发者在使用 `Scaffold.of(context).showSnackBar(snackbar)` 时，可能 遇到过如下错误，这是因为传入的 `context` 属于错误节点导致的，**因为此处传入的 `context` 并不能找到页面所在的 `Scaffold` 节点。**

```dart
class _ExpandedScaffoldPageState extends State<ExpandedScaffoldPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ExpandableNotifier(
        child: Container(),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          showSnackBar();
        },
        child: new Text("点我"),
      ),
    );
  }

  ///会提示错误
  showSnackBar() {
    Scaffold.of(context).showSnackBar(new SnackBar(content: Text("FFFFFFFFF")));
  }

  ExpandableController getController() {
    ///只会返回null
    return ExpandableController.of(context);
  }
}
```

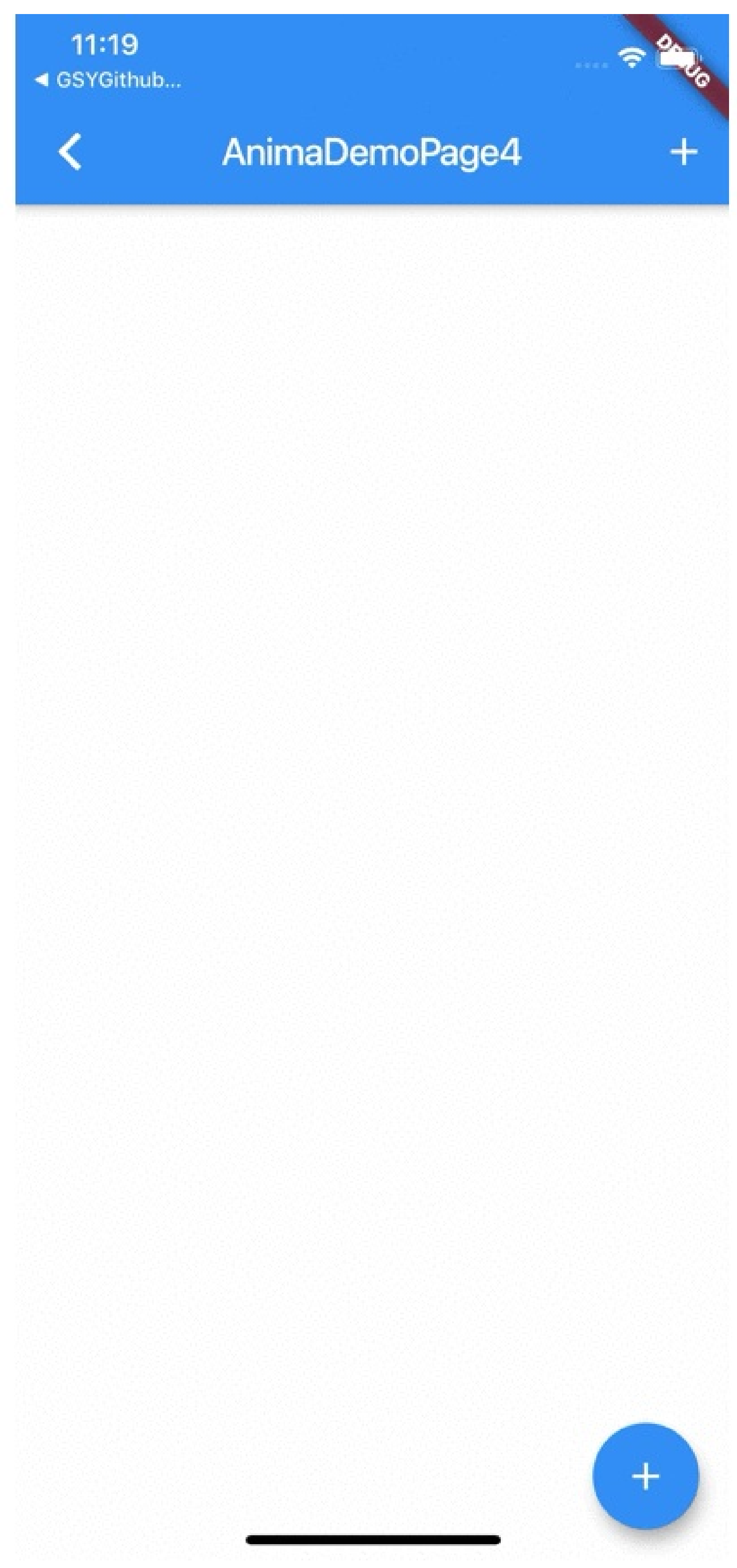
```
main.dart
Console  More Actions
Performing hot restart...
Syncing files to device iPhone X...
Restarted application in 1,374ms.
flutter: ══╡ EXCEPTION CAUGHT BY GESTURE ╞═══════════════════════════
flutter: The following assertion was thrown while handling a gesture:
flutter: Scaffold.of() called with a context that does not contain a Scaffold.
flutter: No Scaffold ancestor could be found starting from the context that was passed to Scaffold.of(). This
```

所以这时候 `Builder` 的作用就体现了，如下所示，通过 `builder` 方法返回赋予的 `context` ，在向上查找 `Scaffold` 的时候，就可以顺利找到父节点的 `Scaffold` 了，这也一定程度上体现了 `ComponentElement` 的作用之一。

```
class _ExpandedScaffoldPageState extends State<ExpandedScaffoldPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ExpandableNotifier(
        child: Container(),
      ),
      floatingActionButton: Builder(
        builder: (ctx) {
          return FloatingActionButton(
            onPressed: () {
              showSnackBar(ctx);
            },
            child: new Text("点我"),
          );
        },
      ),
    );
  }

  showSnackBar(context) {
    Scaffold.of(context).showSnackBar(new SnackBar(content: Text("FFFFFFFFF")));
  }

  ExpandableController getController(context) {
    return ExpandableController.of(context);
  }
}
```

9、快速实现动画切换效果

如下图所示，



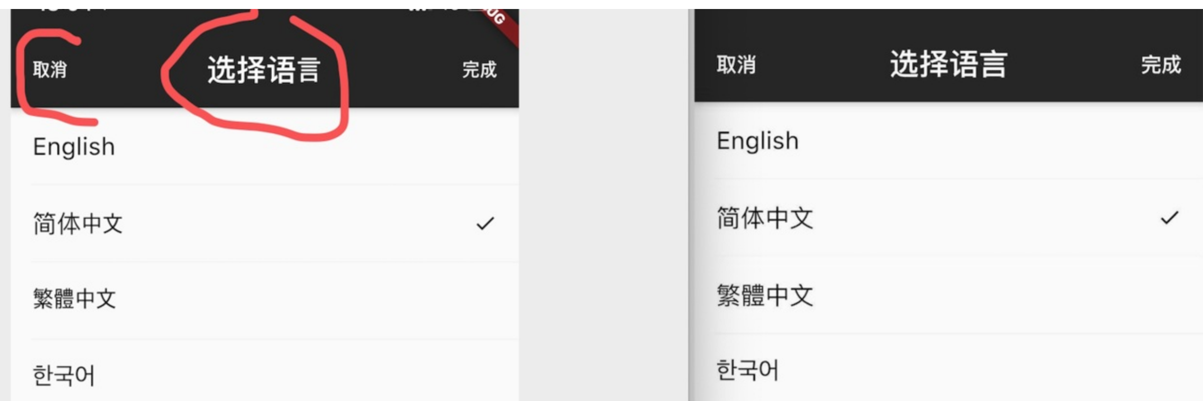
要实现如上图所示动画效果，在 Flutter 中提供了 `AnimatedSwitcher` 封装简易实现。

如下图所示，**通过嵌套 `AnimatedSwitcher` ，指定 `transitionBuilder` 动画效果，然后在数据改变时，同时改变需要执行动画的 `key` 值，即可达到动画切换的效果。**

```
child: AnimatedSwitcher(
  transitionBuilder: (child, anim) {
    return ScaleTransition(child: child, scale: anim);
  },
  duration: Duration(milliseconds: 300),
  child: IconButton(
    key: ValueKey(iconData),
    icon: Icon(iconData),
    onPressed: () {
      setState(() {
        if (iconData == Icons.clear)
          iconData = Icons.add;
        else
          iconData = Icons.clear;
      });
    }),
),
```

10、多语言显示异常

在官方的 https://github.com/flutter/flutter/issues/36527 issue 中可以发现，**Flutter 在韩语/日语 与中文同时显示，会导致 iOS 下出现文字渲染异常的问题**， 如下图所示，左边为异常情况。



改问题解决方案暂时有两种：

- **增加字体 ttf， 全局指定改字体显示。**
- **修改主题下所有 `TextTheme` 的 `fontFamilyFallback` ：**

```
getThemeData() {
  var themeData = ThemeData(
        primarySwatch: primarySwatch
   );

    var result = themeData.copyWith(
      textTheme: confirmTextTheme(themeData.textTheme),
      accentTextTheme: confirmTextTheme(themeData.accentTextTheme),
      primaryTextTheme: confirmTextTheme(themeData.primaryTextTheme),
    );
    return result;
}
```

```
/// 处理 ios 上，同页面出现韩文和简体中文，导致的显示字体异常
confirmTextTheme(TextTheme textTheme) {
  getCopyTextStyle(TextStyle textStyle) {
    return textStyle.copyWith(fontFamilyFallback: ["PingFang SC", "Heiti SC"]);
  }

  return textTheme.copyWith(
    display4: getCopyTextStyle(textTheme.display4),
    display3: getCopyTextStyle(textTheme.display3),
    display2: getCopyTextStyle(textTheme.display2),
    display1: getCopyTextStyle(textTheme.display1),
    headline: getCopyTextStyle(textTheme.headline),
    title: getCopyTextStyle(textTheme.title),
    subhead: getCopyTextStyle(textTheme.subhead),
    body2: getCopyTextStyle(textTheme.body2),
    body1: getCopyTextStyle(textTheme.body1),
    caption: getCopyTextStyle(textTheme.caption),
    button: getCopyTextStyle(textTheme.button),
    subtitle: getCopyTextStyle(textTheme.subtitle),
    overline: getCopyTextStyle(textTheme.overline),
  );
}
```

> ps：通过 `WidgetsBinding.instance.window.locale;` 可以获取到手机平台本身的当前语言情况，不需要 `context` ，也不是你设置后的 `Locale` 。

11、长按输入框导致异常的情况

如果项目存在多语言和主题切换的场景，可能会遇到长按输入框导致异常的场景，目前可推荐两种解放方法：

- 1、可以给你的自定义 `ThemeData` 强制指定固定一个平台，但是该方式会导致平台复制粘贴弹出框没有了平台特性：

```
 ///防止输入框长按崩溃问题
platform: TargetPlatform.android
```

- 2、增加一个自定义的 `LocalizationsDelegate` ，实现多语言环境下的自定义支持：

```
class FallbackCupertinoLocalisationsDelegate
    extends LocalizationsDelegate<CupertinoLocalizations> {
  const FallbackCupertinoLocalisationsDelegate();

  @override
  bool isSupported(Locale locale) => true;
```

```
  @override
  Future<CupertinoLocalizations> load(Locale locale) => loadCupertinoLocalizations(
locale);

  @override
  bool shouldReload(FallbackCupertinoLocalisationsDelegate old) => false;
}

class CustomZhCupertinoLocalizations extends DefaultCupertinoLocalizations {
  const CustomZhCupertinoLocalizations();

  @override
  String datePickerMinuteSemanticsLabel(int minute) {
    if (minute == 1) return '1 分钟';
    return minute.toString() + ' 分钟';
  }

  @override
  String get anteMeridiemAbbreviation => '上午';

  @override
  String get postMeridiemAbbreviation => '下午';

  @override
  String get alertDialogLabel => '警告';

  @override
  String timerPickerHourLabel(int hour) => '小时';

  @override
  String timerPickerMinuteLabel(int minute) => '分';

  @override
  String timerPickerSecond(int second) => '秒';

  @override
  String get cutButtonLabel => '裁剪';

  @override
  String get copyButtonLabel => '复制';

  @override
  String get pasteButtonLabel => '粘贴';

  @override
  String get selectAllButtonLabel => '全选';
}

class CustomTCCupertinoLocalizations extends DefaultCupertinoLocalizations {
  const CustomTCCupertinoLocalizations();
```

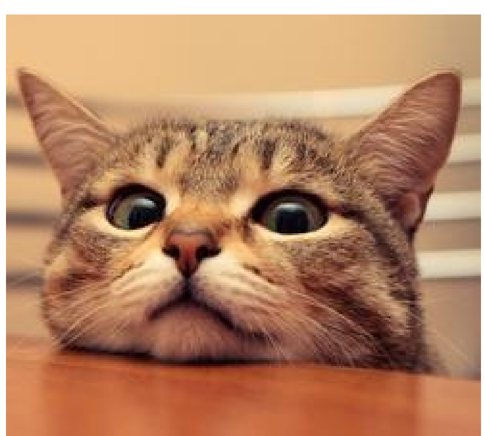
```
  @override
  String datePickerMinuteSemanticsLabel(int minute) {
    if (minute == 1) return '1 分鐘';
    return minute.toString() + ' 分鐘';
  }

  @override
  String get anteMeridiemAbbreviation => '上午';

  @override
  String get postMeridiemAbbreviation => '下午';

  @override
  String get alertDialogLabel => '警告';

  @override
  String timerPickerHourLabel(int hour) => '小时';

  @override
  String timerPickerMinuteLabel(int minute) => '分';

  @override
  String timerPickerSecond(int second) => '秒';

  @override
  String get cutButtonLabel => '裁剪';

  @override
  String get copyButtonLabel => '復制';

  @override
  String get pasteButtonLabel => '粘貼';

  @override
  String get selectAllButtonLabel => '全選';
}

Future<CupertinoLocalizations> loadCupertinoLocalizations(Locale locale) {
  CupertinoLocalizations localizations;
  if (locale.languageCode == "zh") {
    switch (locale.countryCode) {
      case 'HK':
      case 'TW':
        localizations = CustomTCCupertinoLocalizations();
        break;
      default:
        localizations = CustomZhCupertinoLocalizations();
    }
  } else {
    localizations = DefaultCupertinoLocalizations();
  }
```

```
  return SynchronousFuture<CupertinoLocalizations>(localizations);
}
```

> 自此，第十七篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp



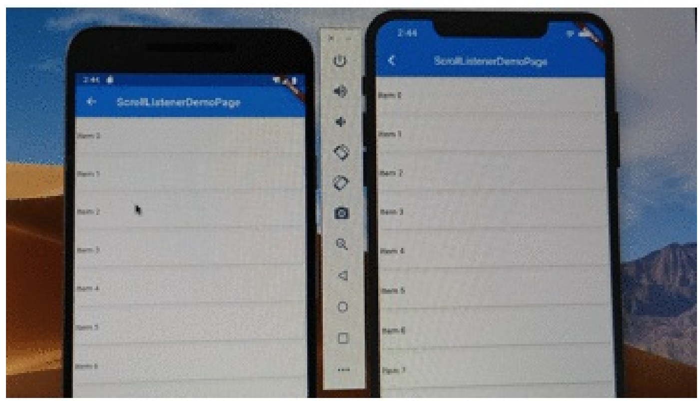
作为系列文章的第十八篇，本篇将通过 ScrollPhysics 和 Simulation ，带你深入走进 Flutter 的滑动新世界，为你打开 Flutter 滑动操作的另一扇窗。

文章汇总地址：

> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

一、前言

如下图所示，Flutter 默认的可滑动 `Widget` ，在 Android 和 iOS 上具备不同的 **滑动与边缘拖拽效果** ，这是因为在不同平台上，默认使用了不同的 **`ScrollPhysics` 与 `Simulation`** ，后面我们将逐步介绍这两大主角的实现原理，**最终让你对 Flutter 世界的滑动拖拽进阶到 “为所欲为” 的境界。**



> 下方开始高能干货，请自带茶水食用。

二、 ScrollPhysics

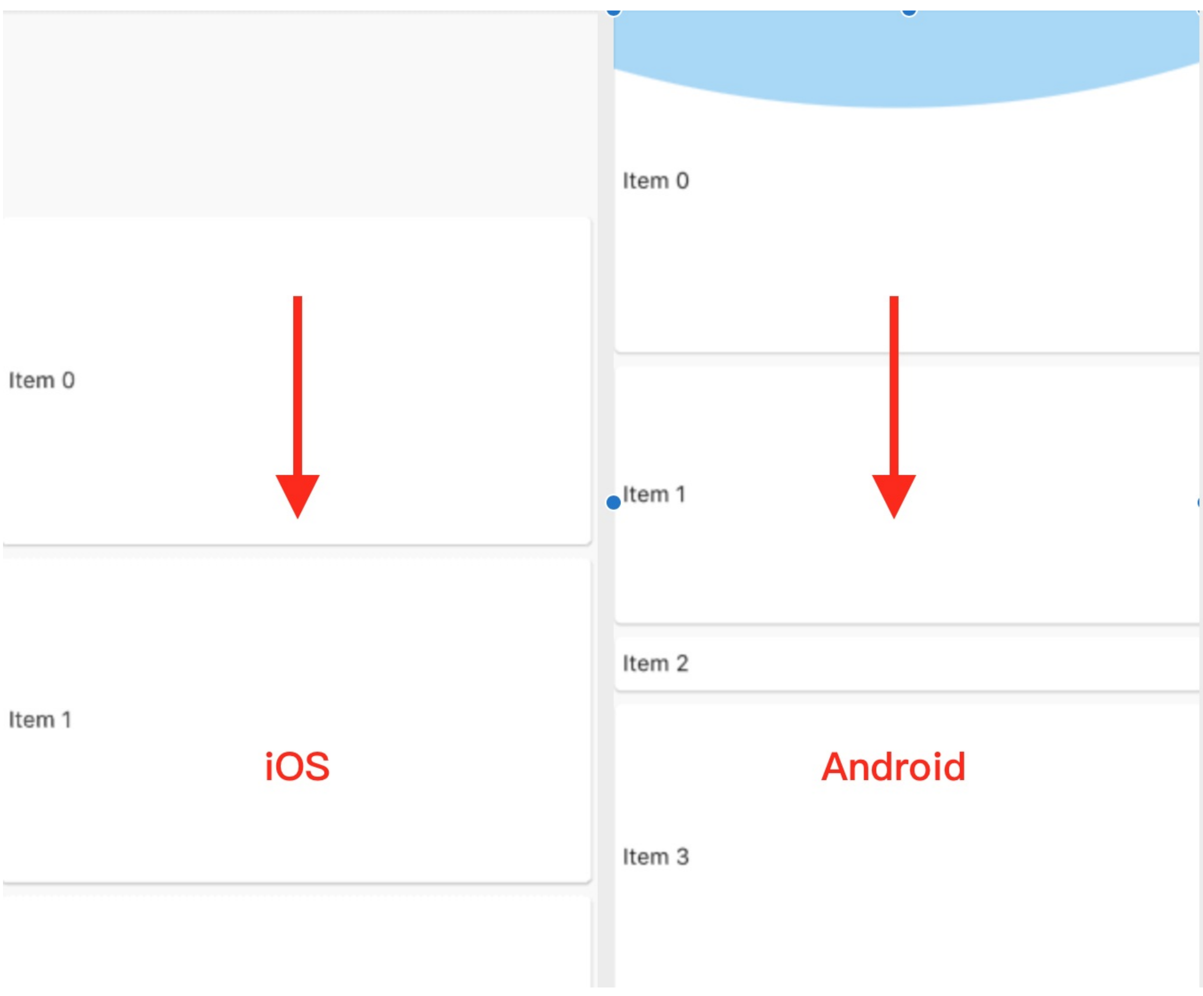
首先介绍 `ScrollPhysics` ，在 Flutter 官方的介绍中， `ScrollPhysics` 的作用是 **确定可滚动控件的物理特性，** 常见的有以下四大金刚：

- **`BouncingScrollPhysics`** ：允许滚动超出边界，但之后内容会**反弹**回来。
- **`ClampingScrollPhysics`** ： 防止滚动超出边界，**夹住** 。
- **`AlwaysScrollableScrollPhysics`** ：始终**响应**用户的滚动。
- **`NeverScrollableScrollPhysics`** ：**不响应**用户的滚动。

在开发过程中，一般会通过如下代码进行设置：

```
CustomScrollView(physics: const BouncingScrollPhysics())
ListView.builder(physics: const AlwaysScrollableScrollPhysics())
GridView.count(physics: NeverScrollableScrollPhysics())
```

但在一般我们都不会主动去设置 `physics` **属性，** 那么默认情况下，为什么在 Flutter 中的 `ListView` 、 `CustomScrollView` 等 `Scrollable` 控件，在 Android 和 iOS 平台的滚动和边界拖拽效果，会有如下图所示的平台区别呢？



这里的关键就在于 `ScrollConfiguration` 和 `ScrollBehavior` 。

2.1、ScrollConfiguration 和 ScrollBehavior

我们知道，所有的滑动控件都是通过 `Scrollable` 进行滑动的。

如下代码所示，在 `Scrollable` 内的 **`_updatePosition`** 方法里，当 `widget.physics == null` 时， **`_physics` 默认是从 `ScrollConfiguration.of(context)` 的 `getScrollPhysics(context)` 方法获取**，而 `ScrollConfiguration.of(context)` 返回的是一个 `ScrollBehavior` 对象。

```
// Only call this from places that will definitely trigger a rebuild.
void _updatePosition() {
  _configuration = ScrollConfiguration.of(context);
  _physics = _configuration.getScrollPhysics(context);
  if (widget.physics != null)
    _physics = widget.physics.applyTo(_physics);
  final ScrollController controller = widget.controller;
  final ScrollPosition oldPosition = position;
```

```
    if (oldPosition != null) {
      controller?.detach(oldPosition);
      scheduleMicrotask(oldPosition.dispose);
    }
    _position = controller?.createScrollPosition(_physics, this, oldPosition)
      ?? ScrollPositionWithSingleContext(physics: _physics, context: this, oldPosit
ion: oldPosition);
    assert(position != null);
    controller?.attach(position);
  }
```

所以默认情况下， `ScrollPhysics` 是和 `ScrollConfiguration` 和 `ScrollBehavior` 有关系。

那么 **`ScrollBehavior`** 是这么工作的?

查看 **`ScrollBehavior`** 的源码可知，它的 `getScrollPhysics` 方法中，**默认实现了平台返回了不同的 `ScrollPhysics`** ，所以默认情况下，在不同平台上的滚动和边缘推拽，会出现不一样的效果：

```
  ScrollPhysics getScrollPhysics(BuildContext context) {
    switch (getPlatform(context)) {
      case TargetPlatform.iOS:
        return const BouncingScrollPhysics();
      case TargetPlatform.android:
      case TargetPlatform.fuchsia:
        return const ClampingScrollPhysics();
    }
    return null;
  }
```

前面说过， **`ScrollPhysics` 是确定可滚动控件的物理特性**，那么如前图所示，**Android 平台上拖拽溢出的蓝色半圆的怎么来的? `ScrollConfiguration` 的 `ScrollBehavior` 是在什么时候被设置的?**

查看 `ScrollConfiguration` 的源码我们得知， **`ScrollConfiguration` 和 `Theme` 、 `Localizations` 等一样是 `InheritedWidget`，那么它应该是从上层往下共享的。**

所以查看 `MaterialApp` 的源码，得到如下代码，可以看到 **`ScrollConfiguration` 是在 `MaterialApp` 内默认嵌套的，并且通过 `_MaterialScrollBehavior` 设置了 `ScrollBehavior`，其 override 的 `buildViewportChrome` 方法，就是实现了Android 上溢出拖拽的半圆效果，** 其中 `GlowingOverscrollIndicator` 就是半圆效果的绘制控件。

```
@override
Widget build(BuildContext context) {
   ....
    return ScrollConfiguration(
      behavior: _MaterialScrollBehavior(),
      child: result,
    );
```

```
}
class _MaterialScrollBehavior extends ScrollBehavior {
  @override
  TargetPlatform getPlatform(BuildContext context) {
    return Theme.of(context).platform;
  }
  @override
  Widget buildViewportChrome(BuildContext context, Widget child, AxisDirection axis
Direction) {
    switch (getPlatform(context)) {
      case TargetPlatform.iOS:
        return child;
      case TargetPlatform.android:
      case TargetPlatform.fuchsia:
        return GlowingOverscrollIndicator(
          child: child,
          axisDirection: axisDirection,
          color: Theme.of(context).accentColor,
        );
    }
    return null;
  }
}
```

到这里我们就知道了，在默认情况下可滑动控件的 `ScrollPhysics` 是如何配置的：

- **1、 `ScrollConfiguration` 是一个 `InheritedWidget` 。**
- **2、 `MaterialApp` 内部利用 `ScrollConfiguration` 并共享了一个 `ScrollBehavior` 的子类 `_MaterialScrollBehavior` 。**
- **3、 `ScrollBehavior` 默认根据平台返回了特定的 `BouncingScrollPhysics` 和 `ClampingScrollPhysics` 效果。**
- **4、 `_MaterialScrollBehavior` 中针对 Android 平台实现了 `buildViewportChrome` 的蓝色半球拖拽溢出效果。**

> ps： 我们可以通过实现自己的 `ScrollBehavior` ， 实现自定义的拖拽溢出效果。

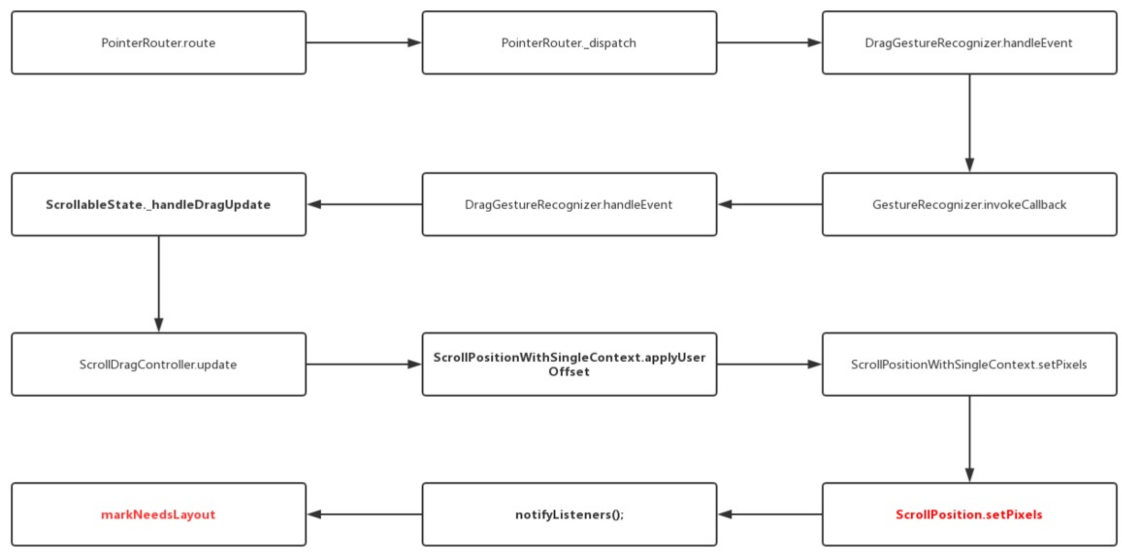
三、ScrollPhysics 工作原理

那么 `ScrollPhysics` 是怎么实现滚动和边缘拖拽的呢？ `ScrollPhysics` 默认是没有什么代码逻辑的，它的主要定义方法如下所示：

```
/// [position] 当前的位置, [offset] 用户拖拽距离
/// 将用户拖拽距离 offset 转为需要移动的 pixels
double applyPhysicsToUserOffset(ScrollMetrics position, double offset)

/// 返回 overscroll ，如果返回 0 ，overscroll 就一直是0
/// 返回边界条件
```

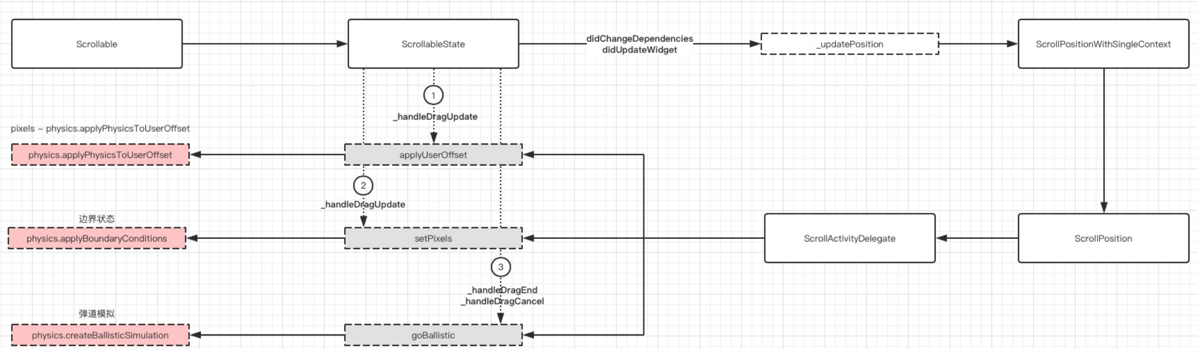
```
double applyBoundaryConditions(ScrollMetrics position, double value)

///创建一个滚动的模拟器
Simulation createBallisticSimulation(ScrollMetrics position, double velocity)

///最小滚动数据
 double get minFlingVelocity

///传输动量，返回重复滚动时的速度
double carriedMomentum(double existingVelocity)

///最小的开始拖拽距离
double get dragStartDistanceMotionThreshold

///滚动模拟的公差
///指定距离、持续时间和速度差应视为平等的差异的结构。
Tolerance get tolerance
```

上方代码标注了 `ScrollPhysics` 各个方法的大致作用，而在前面 《十三、全面深入触摸和滑动原理》 中，我们深入解析过触摸和滑动的原理，大致流程从触摸开始往下传递， 最终触发 `layout` 实现滑动的现象：



而 `ScrollPhysics` 的工作原理就穿插在其中，其流程如下图所示, 主要的逻辑在于红色标注的的三个方法：

- **`applyPhysicsToUserOffset`** ：通过 physics 将用户拖拽距离 `offset` 转化为 `setPixels` (滚动) 的增量。
- **`applyBoundaryConditions`** ：通过 physics 计算当前滚动的边界条件。
- **`createBallisticSimulation`** ： 创建自动滑动的模拟器。



这三个方法的触发时机在于 `_handleDragUpdate` 、 `_handleDragCancel` 和 `_handleDragEnd` ，也就是拖动过程和拖动结束的时机：

- **`applyPhysicsToUserOffset` 和 `applyBoundaryConditions` 是在 `_handleDragUpdate` 时被触发的。**
- **`createBallisticSimulation` 是在 `_handleDragCancel` 和 `_handleDragEnd` 时被触发的。**

所以默认的 **`BouncingScrollPhysics`** 和 **`ClampingScrollPhysics`** 最大的差异也在这个三个方法。

3.1、applyPhysicsToUserOffset

`ClampingScrollPhysics` 默认是没有重载 `applyPhysicsToUserOffset` 方法的，**当 `parent == null` 时，用户的滑动 `offset` 是什么就返回什么：**

```
double applyPhysicsToUserOffset(ScrollMetrics position, double offset) {
  if (parent == null)
    return offset;
  return parent.applyPhysicsToUserOffset(position, offset);
}
```

`BouncingScrollPhysics` 中对 `applyPhysicsToUserOffset` 方法进行了 `override` ，其中 **用户没有达到边界前，依旧返回默认的 `offset` ，当用户到达边界时，通过算法来达到模拟溢出阻尼效果。**

```

 ///摩擦因子
 double frictionFactor(double overscrollFraction) => 0.52 * math.pow(1 - overscroll
Fraction, 2);

 @override
  double applyPhysicsToUserOffset(ScrollMetrics position, double offset) {
    assert(offset != 0.0);
    assert(position.minScrollExtent <= position.maxScrollExtent);

    if (!position.outOfRange)
      return offset;

    final double overscrollPastStart = math.max(position.minScrollExtent - position
```

```
.pixels, 0.0);
    final double overscrollPastEnd = math.max(position.pixels - position.maxScrollE
xtent, 0.0);
    final double overscrollPast = math.max(overscrollPastStart, overscrollPastEnd);
    final bool easing = (overscrollPastStart > 0.0 && offset < 0.0)
        || (overscrollPastEnd > 0.0 && offset > 0.0);

    final double friction = easing
        // Apply less resistance when easing the overscroll vs tensioning.
        ? frictionFactor((overscrollPast - offset.abs()) / position.viewportDimensi
on)
        : frictionFactor(overscrollPast / position.viewportDimension);
    final double direction = offset.sign;

    return direction * _applyFriction(overscrollPast, offset.abs(), friction);
  }
```

3.2、applyBoundaryConditions

`ClampingScrollPhysics` 的 `applyBoundaryConditions` 方法中，在计算边界条件值的时候，**滑动值会和边界值相减得到相反的数据，使得滑动边界相对静止，从而达到“夹住”的作用**，也就是**动态边界**，所以默认请下 Android 上滚动到了边界就会停止响应。

```
  @override
  double applyBoundaryConditions(ScrollMetrics position, double value) {
    if (value < position.pixels && position.pixels <= position.minScrollExtent) //
underscroll
      return value - position.pixels;
    if (position.maxScrollExtent <= position.pixels && position.pixels < value) //
overscroll
      return value - position.pixels;
    if (value < position.minScrollExtent && position.minScrollExtent < position.pix
els) // hit top edge
      return value - position.minScrollExtent;
    if (position.pixels < position.maxScrollExtent && position.maxScrollExtent < va
lue) // hit bottom edge
      return value - position.maxScrollExtent;
    return 0.0;
  }
```

> ps：前面说过蓝色的半圆是默认的 `ScrollBehavior` 内 `buildViewportChrome` 方法实现的。

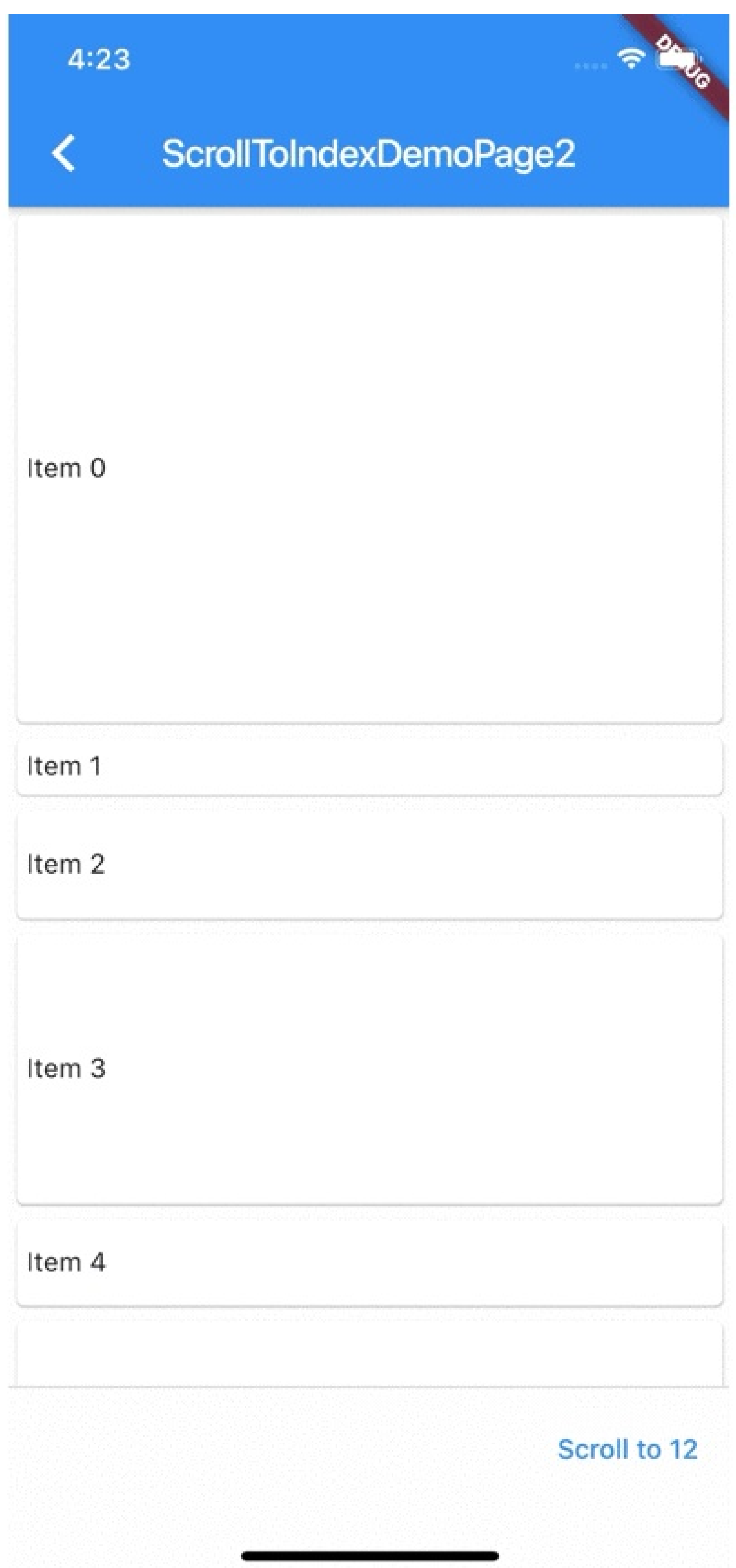
`BouncingScrollPhysics` 中 `applyBoundaryConditions` 直接返回 0，**也就是达到 0 是就边界，过了 0 的就是边界外的拖拽效果了。**

```
  @override
  double applyBoundaryConditions(ScrollMetrics position, double value) => 0.0;
```

3.3、createBallisticSimulation

因为 `createBallisticSimulation` 是在 `_handleDragCancel` 和 `_handleDragEnd` 时触发的，其实就是停止触摸的时候，**当 `createBallisticSimulation` 返回 `null` 时， `Scrllable` 将进入 `IdleScrollActivity` ，也就是停止滚动的状态。**

如下图所示，完全没有 `Simulation` 的列表滚动，是不会连续滚动的。



`ClampingScrollPhysics` 的 `createBallisticSimulation` 方法中，**使用了 `ClampingScrollSimulation` (固定) 和 `ScrollSpringSimulation` (弹性) 两种 `Simulation`** ，如下代码所示，理论上只有 `position.outOfRange` 才会触发弹性的回弹效果，但 `ScrollPhysics` 采用

了类似 **双亲代理模型** ，其 `parent` 可能会触发 `position.outOfRange` ，所以推测这里才会有 `ScrollSpringSimulation` 补充的判断。

如下代码可以看出，**只有在 `velocity` 速度大于默认加速度，并且是可滑动范围内，才返回 `ClampingScrollPhysics` 模拟滑动，否则返回 null 进入前面所说的 Idle 停止滑动，这也是为什么普通慢速拖动，不会触发自动滚动的原因。**

```
@override
  Simulation createBallisticSimulation(
      ScrollMetrics position, double velocity) {
    final Tolerance tolerance = this.tolerance;
    if (position.outOfRange) {
      double end;
      if (position.pixels > position.maxScrollExtent)
        end = position.maxScrollExtent;
      if (position.pixels < position.minScrollExtent)
        end = position.minScrollExtent;
      assert(end != null);
      return ScrollSpringSimulation(
        spring,
        position.pixels,
        end,
        math.min(0.0, velocity),
        tolerance: tolerance,
      );
    }
    if (velocity.abs() < tolerance.velocity) return null;
    if (velocity > 0.0 && position.pixels >= position.maxScrollExtent)
      return null;
    if (velocity < 0.0 && position.pixels <= position.minScrollExtent)
      return null;
    return ClampingScrollSimulation(
      position: position.pixels,
      velocity: velocity,
      tolerance: tolerance,
    );
  }
```

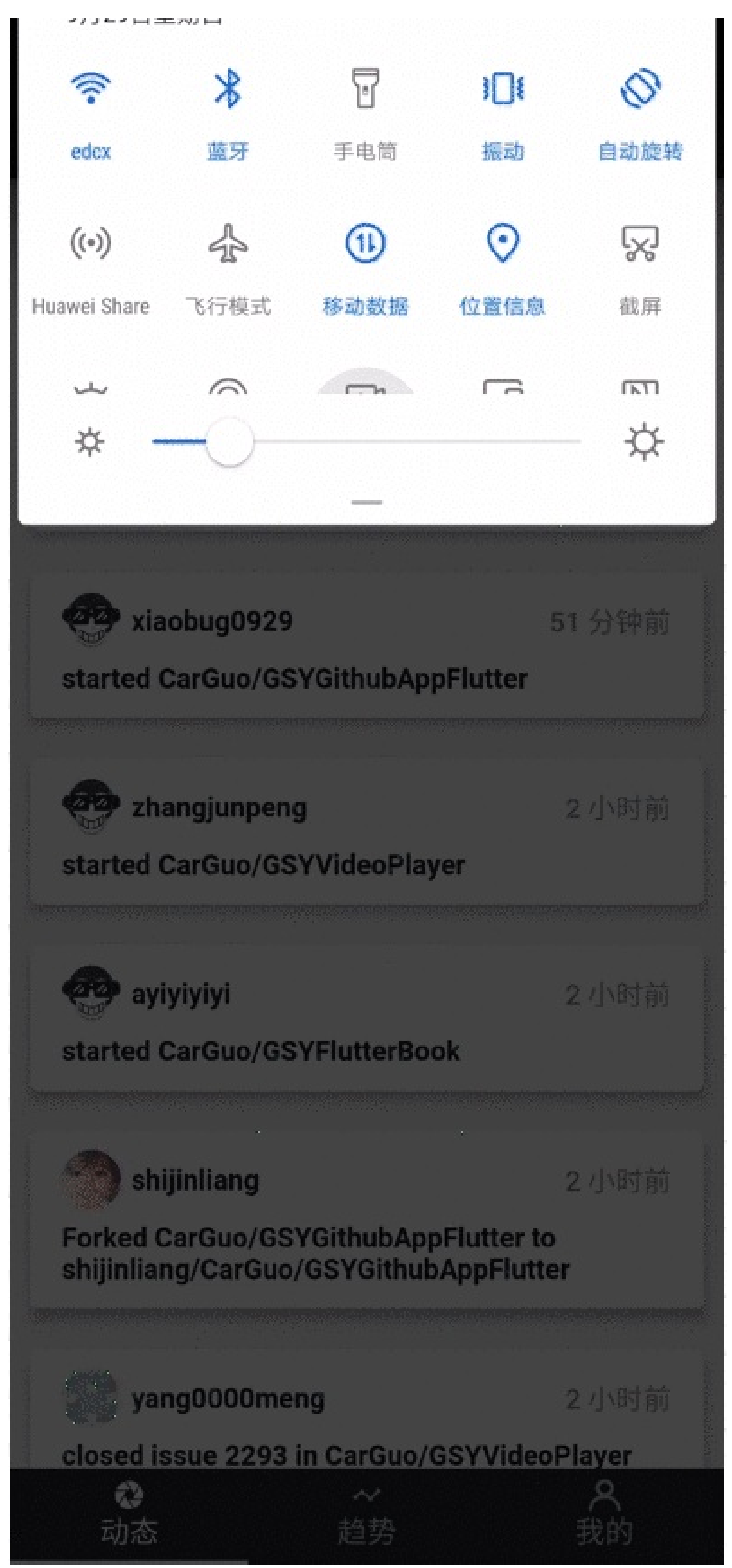
`BouncingScrollPhysics` 的 `createBallisticSimulation` 则简单一些，**只有在结束触摸时，初始速度大于默认加速度或者超出区域，才会返回 `BouncingScrollSimulation` 进行模拟滑动计算，否则经进入前面所说的 Idle 停止滑动。**

```
  @override
  Simulation createBallisticSimulation(ScrollMetrics position, double velocity) {
    final Tolerance tolerance = this.tolerance;
    if (velocity.abs() >= tolerance.velocity || position.outOfRange) {
      return BouncingScrollSimulation(
        spring: spring,
        position: position.pixels,
```

```
        velocity: velocity * 0.91, // TODO(abarth): We should move this constant cl
oser to the drag end.
        leadingExtent: position.minScrollExtent,
        trailingExtent: position.maxScrollExtent,
        tolerance: tolerance,
      );
    }
    return null;
  }
```

可以看出，在停止触摸时，**列表是否会继续模拟滑动是和 `velocity` 和 `tolerance.velocity` 有关，也就是速度大于指定的加速度时才会继续滑动**，并且在可滑动区域内 `ClampingScrollSimulation` 和 `BouncingScrollSimulation` 呈现的效果也不一样。

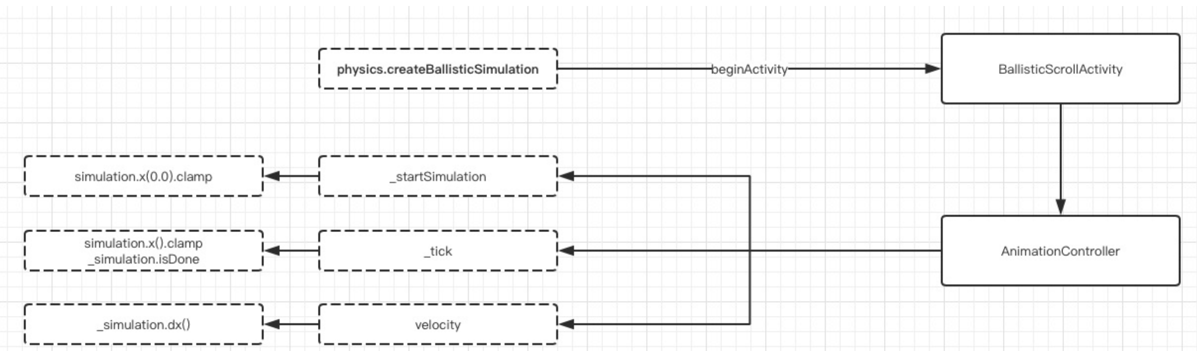
如下图所示，**第一页面的 `ScrollSpringSimulation` 在停止滚动前是有一定的减速效果的；而第二个页面 `ClampingScrollSimulation` 是直接快速滑动到边界。**



> 事实上，通过选择或者调整 `Simulation` ，就可以对列表滑动的速度、阻尼、回弹效果等实现灵活的自定义。

四、Simulation

前面最后说到了，利用 `Simulation` 实现对列表的滑动、阻尼、回弹效果的实现处理，那么 `Simulation` 是如何工作的呢?



如上图所示，**在 `Simulation` 的创建是在 `ScrollPositionWithSingleContext` 的 `goBallistic` 方法中被调用的**，然后通过 `BallisticScrollActivity` 去触发执行。

```
  @override
  void goBallistic(double velocity) {
    assert(pixels != null);
    final Simulation simulation = physics.createBallisticSimulation(this, velocity)
;
    if (simulation != null) {
      beginActivity(BallisticScrollActivity(this, simulation, context.vsync));
    } else {
      goIdle();
    }
  }
```

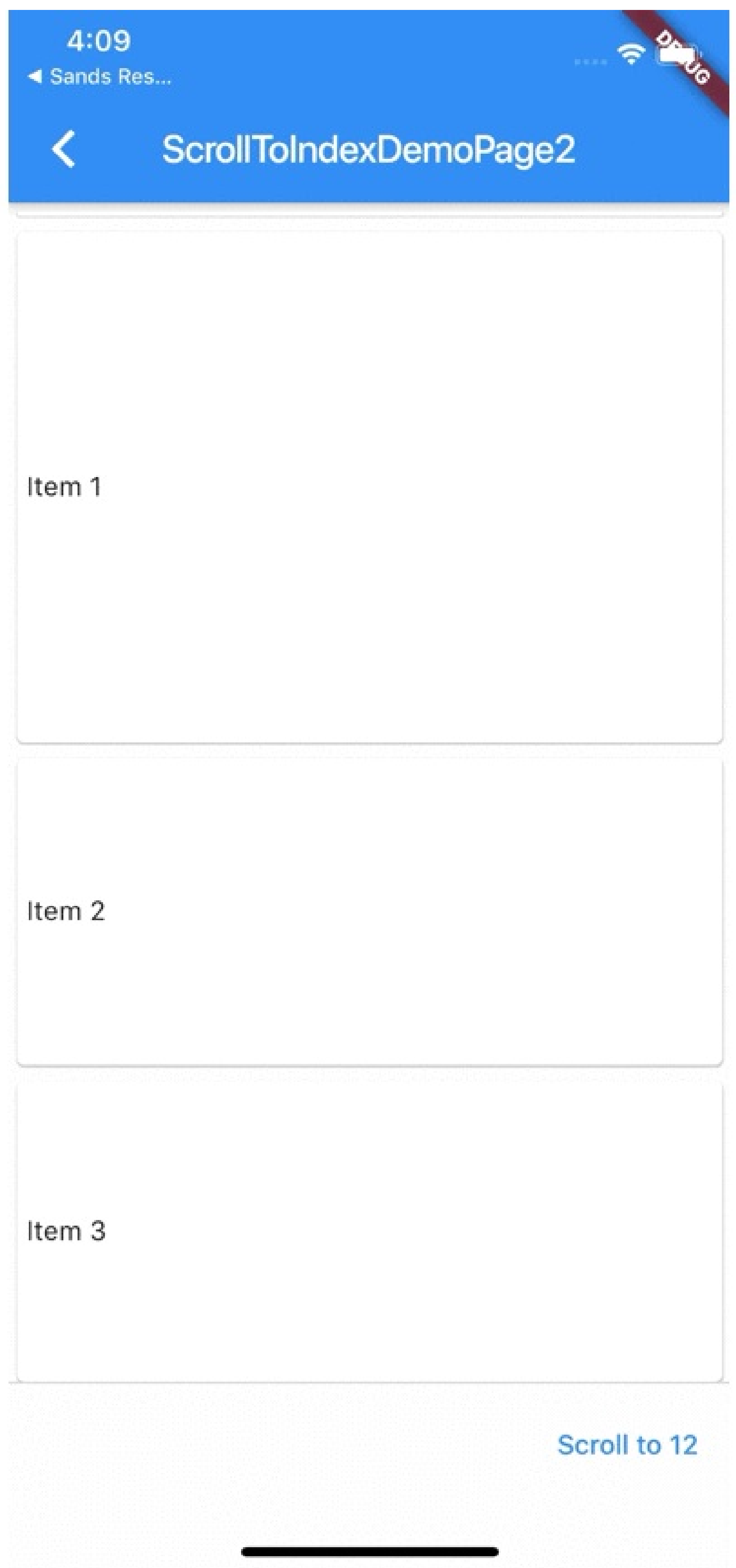
在 `BallisticScrollActivity` 状态中，**`Simulation` 被用于驱动 `AnimationController` 的 `value` ，然后在动画的回调中获取 `Simulation` 计算后得到的 `value` 进行 `setPixels(value)` 实现滚动。**

> 这里又涉及到了动画的绘制机制，动画的机制等新篇再详细说明，简单来说就是 **当系统 `drawFrame` 的 `vsync` 信号到来时，会执行到 `AnimationController` 内部的 `_tick` 方法，从而触发 `_value = _simulation.x(elapsedInSeconds).clamp(lowerBound, upperBound);` 改变和 `notifyListeners();` 通知更新。**

对于 `Simulation` 的内部计算逻辑这里就不展开了，大致上可知 **`ClampingScrollSimulation` 的摩擦因子是固定的，而 `BouncingScrollSimulation` 内部的摩擦因子和计算，是和传递的位置有关系。**

这里需要着重提及的就是，为什么 `BouncingScrollPhysics` 会自动回弹呢?

其实也是 `BouncingScrollSimulation` 的功劳，因为 `BouncingScrollSimulation` 构建时，会传递有 `leadingExtent:position.minScrollExtent` 和 `trailingExtent: position.maxScrollExtent` 两个参数，**在 underscroll 和 overscroll 的情况下，会利用 `ScrollSpringSimulation` 实现弹性的回滚到 `leadingExtent` 和 `trailingExtent` 的动画，从而达到如下图的效果：**



最后

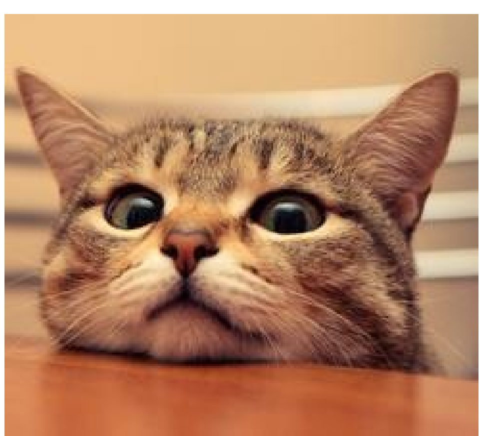
到这里 Flutter 的 `ScrollPhysics` 和 `Simulation` 就基本分析完了，严格意义上， `Simulation` 应该是属于动画的部分，但是这里因为 `ScrollPhysics` 也放到了一起。

总结起来就是 `ScrollPhysics` 中控制了用户触摸转化和边界条件，并且在用户停止触摸时，利用 `Simulation` 实现了自动滚动与溢出回弹的动画效果。

> 自此，第十八篇终于结束了！(///▽///)

资源推荐

- Github ： https://github.com/CarGuo
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp



番外系列

作为对 Flutter 系列额外的补充，不定时给你带来有趣的文章。

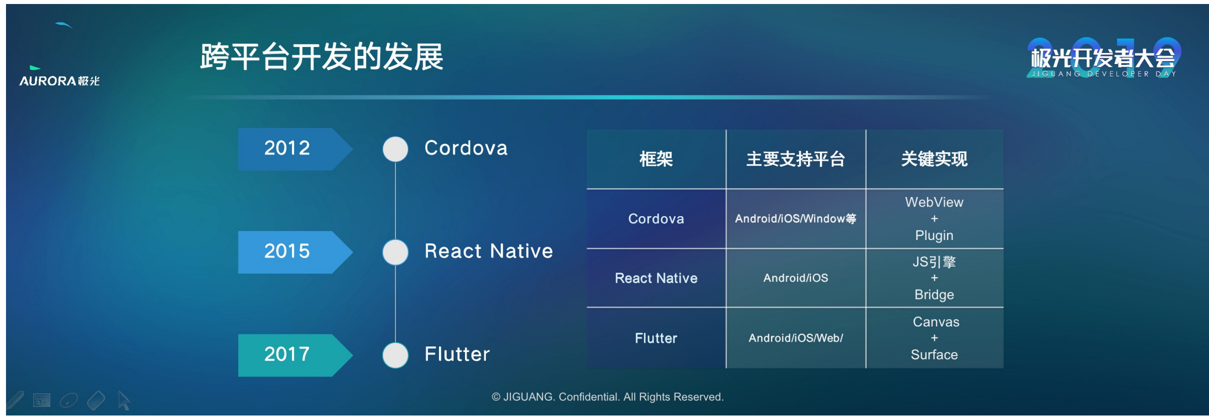
> 大家好，我是郭树煜，掘金 《*Flutter* 完整开发实战详解》 系列的作者，Github GSY 系列开源项目的维护人员，系列包括 GSYVideoPlayer 、 `GSYGitGithubApp` (Flutter \ ReactNative \ Kotlin \ Weex 四大版本)、GSYFlutterBook 电子书等，系列总 star 数在 25k 左右，目前 Github 中国区粉丝数暂居 67 名，主要负责移动端项目开发，大前端方向，主要涉及领域有 Android、Flutter、React Native、Weex 、小程序等等。

这次分享的主题主要涉及：**移动端跨平台开发的发展**、**Flutter Widget 的实现原理** 、 **Flutter 的实战技巧** 、**Flutter Web的现状** 四个方面，而整体主题将围绕 Widget 为中心展开。

一、移动端跨平台开发的发展

按照惯例，我们先介绍历史进程，随着用户终端种类的百花齐放，如今跨平台开发已然成为移动领域的热门话题之一，移动端跨平台开发技术的发展，也代表着开发者对于**性能**、**复用**、**高效**上不断的追求。

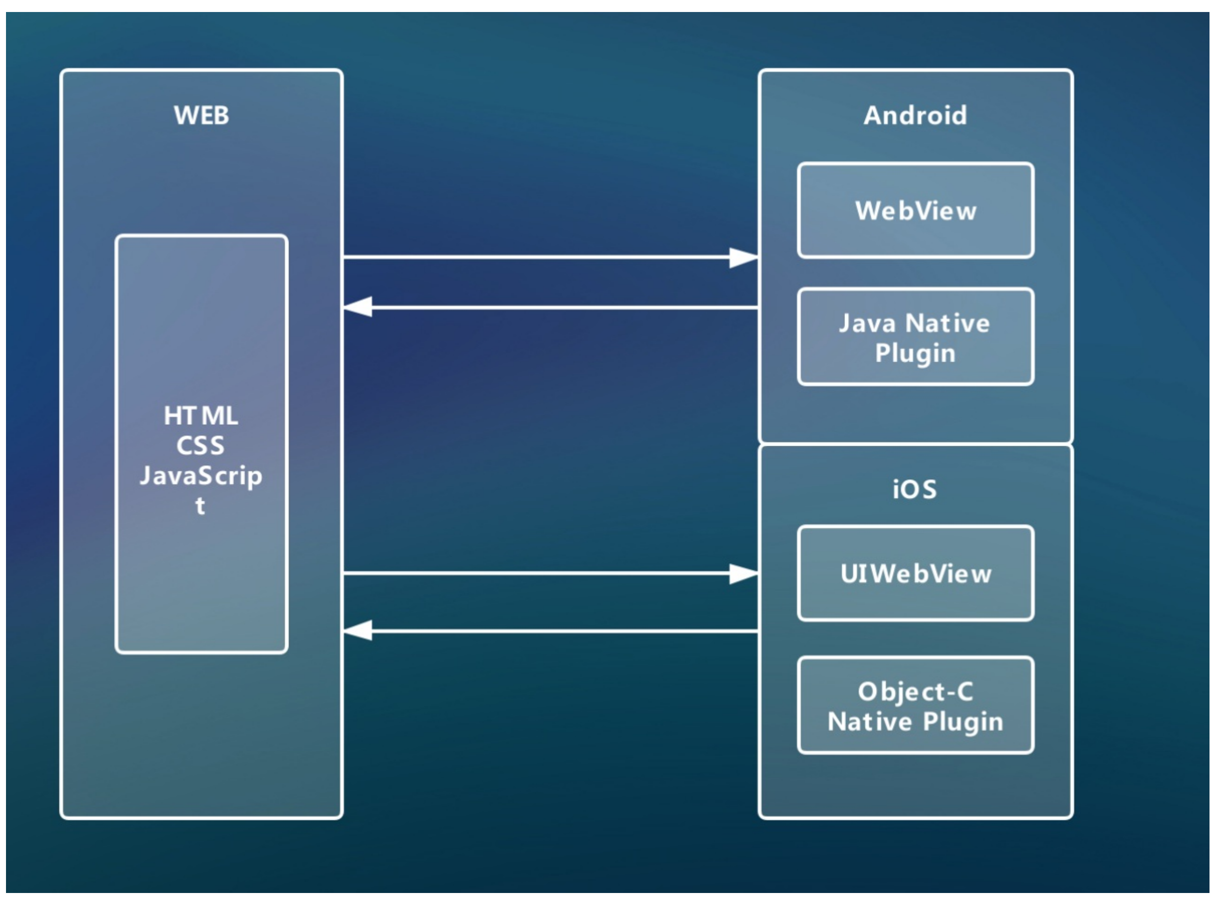
移动端的跨平台开发主要有三个阶段，这些阶段的代表框架主要有： **`Cordova`** 、 **`React Native`** 、 **`Flutter`** 等，如下图所示，是移动端的跨平台发展历程：



Cordova

`Cordova` 作为早期跨平台领域应用最广泛的框架，为前端人员所熟知，其主要原理就是：

将 web 代码打包到本地，利用平台的 WebView 进行加载，通过内部约定好的 JS 通讯协议，加载和调用具备平台原生能力的插架。



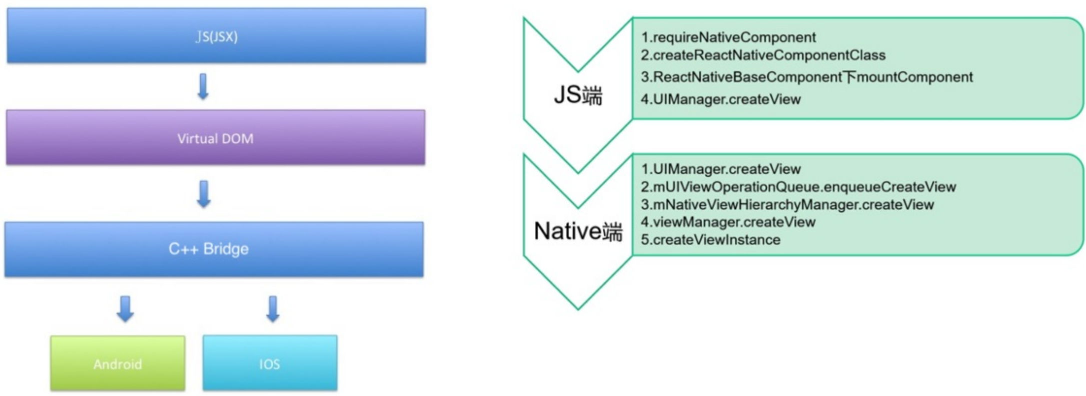
`Cordova` 让前端开发人员可以快速的构建移动应用，获取平台入口，对早期 web 上欠缺的如**摄像机、本地缓存、文件读写**等能力进行快速支持。

> 早期的移动开发市场除了 Android 和 iOS 之外，还有 WindowPhone、黑莓等，`Cordova` 简单又实用的理念，使得它成为早期热门的跨平台框架，至今仍在更新的 `ionic` 框架，也是在其基础上进行了封装发展。

React Native

`Cordova` 虽然实用方便，但是由于 `WebView` 的性能瓶颈，开发者开始追求**更高性能，且具备平台特色**的跨平台能力，这时候由 Facebook 开源的 `React Native` 框架开始引领新潮流。

`React Native` 让 JS 代码运行在框架内置的 JS 引擎（JavaScriptCore）上，利用 JS 引擎实现了跨平台能力，同时又将 JS 控件，对应解析为平台原生控件进行渲染，从而实现性能的优化与提升。



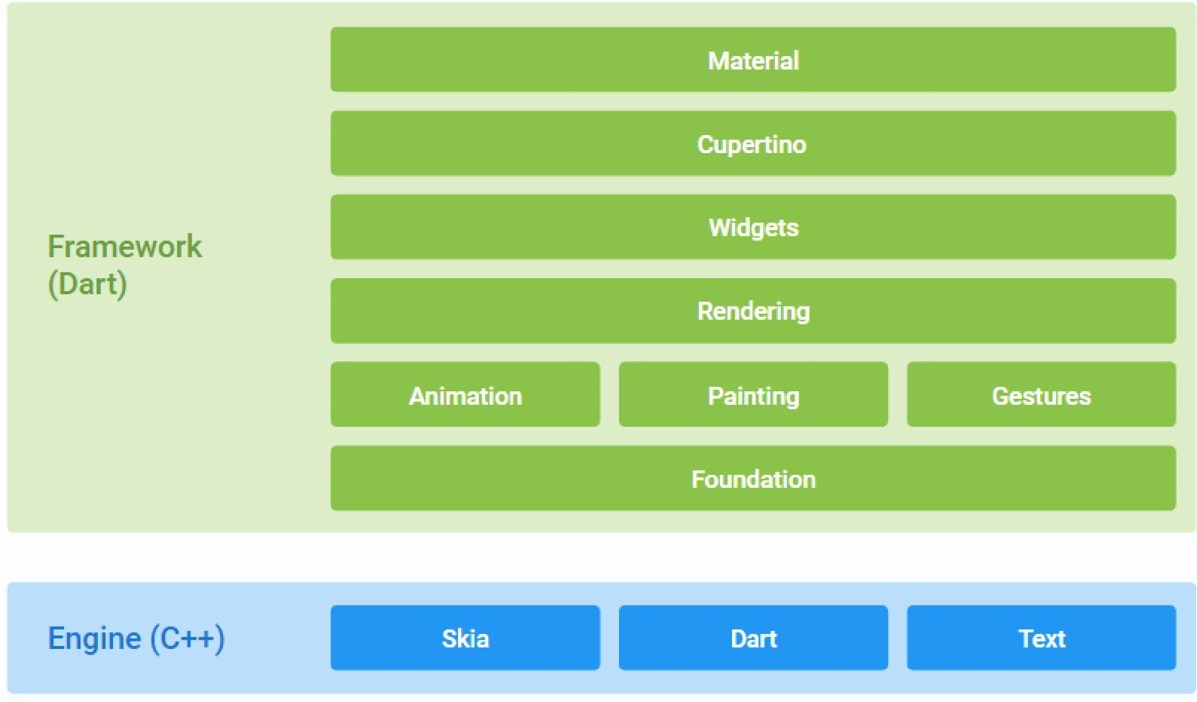
由于 `React` 框架的盛行， `React Native` 也开始成为 `React` 开发人员，将自身能力拓展到应用开发的最佳选择之一。同时 `React Native` 也是应用开发人员，接触前端的不错尝试。

> 后来阿里开源的 `Weex` 框架设计相似，利用了 V8 引擎实现跨平台，不过使用了 `Vue` 的设计理念，而 `Weex` 因为种种原因，最终还是没能大面积推广开来。

Flutter

事实上 `JS Bridge` 同样存在性能等限制，Facebook 也在着力优化这一问题，比如 `HermesJS` 、底层大规模重构等， 而 JS -> 平台控件映射，也导致了框架和平台耦合过多，在版本兼容和系统升级等问题上让框架维护越发困难。

这时候谷歌开源了 `Flutter` ，**它另辟蹊径，只要求平台提供一个 `Surface` 和一个 `Canvas` ，剩下的 `Flutter` 说："你可以躺下了，我们来自己动"。**



`Flutter` 的跨平台思路快速让他成为“新贵”，**连跨平台界的老大哥 “JS” 语言都“视而不见”，大胆的选择 `Dart` 也让 `Flutter` 在前期的推广中饱受争议。**

> 短短两年，不算 PR ， `Flutter` 的 issue 已经有近 1.8 万的 closed 和 8000+ open , 这代表了它的热度，也代表着它需要面对的问题和挑战。 不支持 Release 模式下的热更新，也让用户更多徘徊于 React Native 不愿尝试。
>
> **不过有一点可以确定的，那就是 `Flutter` 的版本号上是彻底战胜了 `React Naitve` 。**

总结起来，我们可以看到，移动端跨平台的发展，**从单纯的套壳打包，到提供高性能的跨平台控件封装，再到现在的控件与平台脱离的发展。** 整个发展历程，就是对 **性能、复用、高效** 的不断追求。

题外话，什么要学习跨平台?

1、开发成本

我直接学 `Java` / `Kotlin` 、 `Object-C` / `Swift` 、 `JavaScript` / `CSS` 去写各平台的代码可以吗?

当然可以，这样的性能肯定最有保证，但是跨平台的主要优势在于代码逻辑的复用，减少各平台同一逻辑，因人而异的开发成本。

2、学习机会

一般情况下，各平台开发者容易局限在自己的领域开发，**而作为应用开发者，跨平台是接触另一平台或领域的过渡机会。**

> 下面开始今天的主题 Flutter ，Flutter 整体涉及的内容很多，由于篇幅问题，本篇我们的主题整体都围绕一个 `Widget` 展开。**Flutter 作为跨平台 UI 框架， `Widget` 是其灵魂设定之一。**

二、Flutter Widget 的实现原理

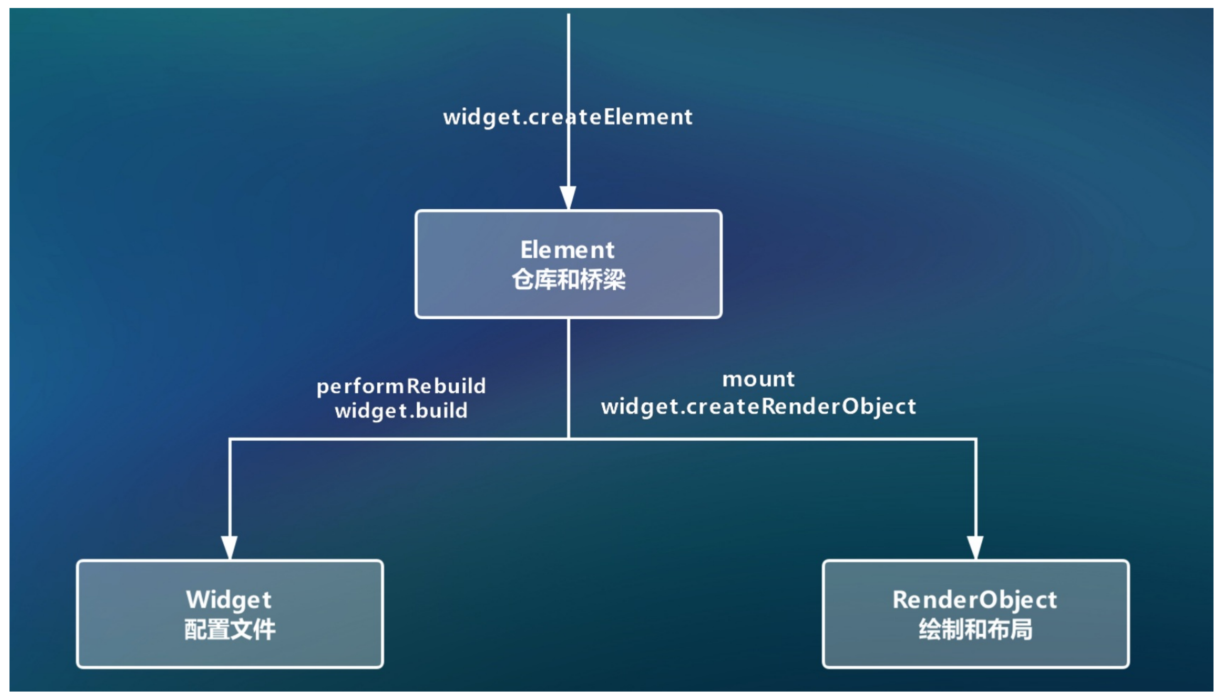
Flutter 是 UI 框架，Flutter 内一切皆 `Widget` ，每个 `Widget` 状态都代表了一帧， `Widget` 是不可变的。 那么 `Widget` 是怎么工作的呢？

如下图可以看到，是一个简单的 Flutter `Widget` 页面代码，页面包含了一个标题和容易，那在页面 `build` 时，它是怎么表绘制出来的呢？同时它是如何保证性能？ 而 `Widget` 又是怎么样的一个概念？后面我们将逐步揭晓。

```
class DemoPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: new Text("标题"),
      ),
      body: new Container(
        child: new Center(
          child: new Text("内容"),
        ),
      ),
    );
  }
}
```

首先看上图代码，其实如图的代码并不是真正的 `View` 级别代码，它们更像是配置文件。

而要知道 `Widget` 是如何工作的，这就涉及到 Flutter 的三大金刚： **`Widget`** 、 **`Element`** 、 **`RenderObject`** 。事实上，这三大金刚才能组成了 Flutter Framework 的基础渲染闭环。



如上图所示，当一个 `Widget` 被“加载“的时候，它并不是马上被绘制出来，而是会对应先创建出它的 `Element` ，然后通过 `Element` 将 `Widget` 的配置信息转化为 `RenderObject` 实现绘制。

所以，在 Flutter 中大部分时候我们写的是 `Widget` ，但是 `Widget` 的角色反而更像是“配置文件”，真正触发工作的其实是 `RenderObject` 。

小结一下这里的关系就是：

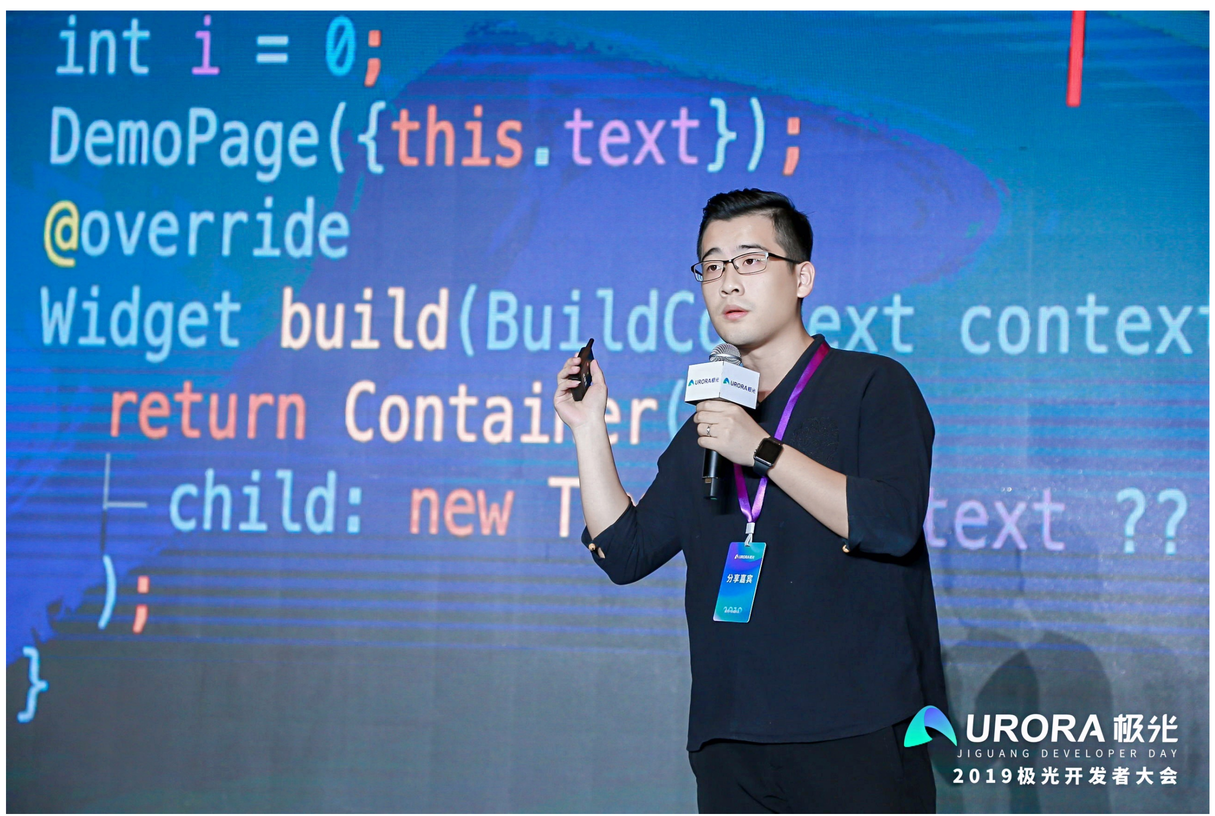
- `Widget` 是配置文件。
- `Element` 是桥梁和仓库。
- `RenderObject` 是解析后的绘制和布局。

对应详细的解释就是：

- **所以我们写的 `Widget` ，它需要转化为相应的 `RenderObject` 去工作；**
- `Element` 持有 `Widget` 和 `RenderObject` ，作为两者的桥梁，并保存着一些状态参数，**我们在 Flutter 框架中常见到的 `BuildContext` ，其实就是 `Element` 的抽象**；
- 最后框架会将 `Widget` 的配置信息，转化到 `RenderObject` 内，告诉 `Canvas` 应该在哪个 `Rect` 内，绘制多大 `Size` 的数据。

所以 `Widget` 和我们以前的布局概念不一样，因为 `Widget` 是不可变的（ `immutable` ），且只有一帧，且不是真正工作的对象，每次画面变化，都会导致一些 `Widget` 重新 `build` 。

那到这里，我们可能就会关心性能的问题，**Flutter 是如何保证性能呢？**



1.1、Widget 的轻量级

其实就是回归到了 `Widget` 的定位，作为“配置文件”，**`Widget` 的变化，是否也会导致 `Element` 和 `RenderObject` 也会重新创建?**

答案是不一定会， `Widget` 只是一个 “配置文件” 的作用，是非常轻量级的，**它的存在，只是起到对 `RenderObject` 的数据进行配置的作用。**

但是 `RenderObject` 就不一样了，**它涉及到了 `layout`** 、 **`paint`** 等真实 的绘制操作，可以认为是一个真正的 “View” ，如果频繁创建就会导性能出现问题。

所以在 Flutter 中，**会有一系列的判断，来处理 `Widget` 到 `RenderObject` 转化的性能问题 ，这部分操作通常是在 `Element` 中进行的** ，例如 `updateChild` 时，会有如下图所示的判断：

```
@protected
Element updateChild(Element child, Widget newWidget, dynamic newSlot) {
  if (child != null) {
    if (child.widget == newWidget) {
      if (child.slot != newSlot)
        updateSlotForChild(child, newSlot);
      return child;
    }
    if (Widget.canUpdate(child.widget, newWidget)) {
      if (child.slot != newSlot)
        updateSlotForChild(child, newSlot);
      child.update(newWidget);
      return child;
    }
    deactivateChild(child);
  }
  return inflateWidget(newWidget, newSlot);
}
```

- 当 `element.child.widget == widget.build()` 时，就不会触发 `update` 操作；
- 在 `update` 时， `canUpdate(element.child.widget, newWidget)` 返回 true， `Element` 才会被更新；（这里代码中的 *`slot`* 一般为 *`Element`* 对象，有时候会传空）
- 其他还有利用 `isRelayoutBoundary` 、 `isRepaintBoundary` 等参数，来实现局部的更新判断，比如：**当执行 markNeedsPaint() 触发绘制时，会通过 `isRepaintBoundary` 是否为 `true` ，往上确定了更新区域，通过 `requestVisualUpdate` 方法触发更新往下绘制。**

> 通过 `isRepaintBoundary` 参数，对应的 `RenderObject` 可以组成一个 `Layer` 。

所以这就可以解答一些初学者的疑问，嵌套那么多 *`Widget`* ，性能会不会有问题?

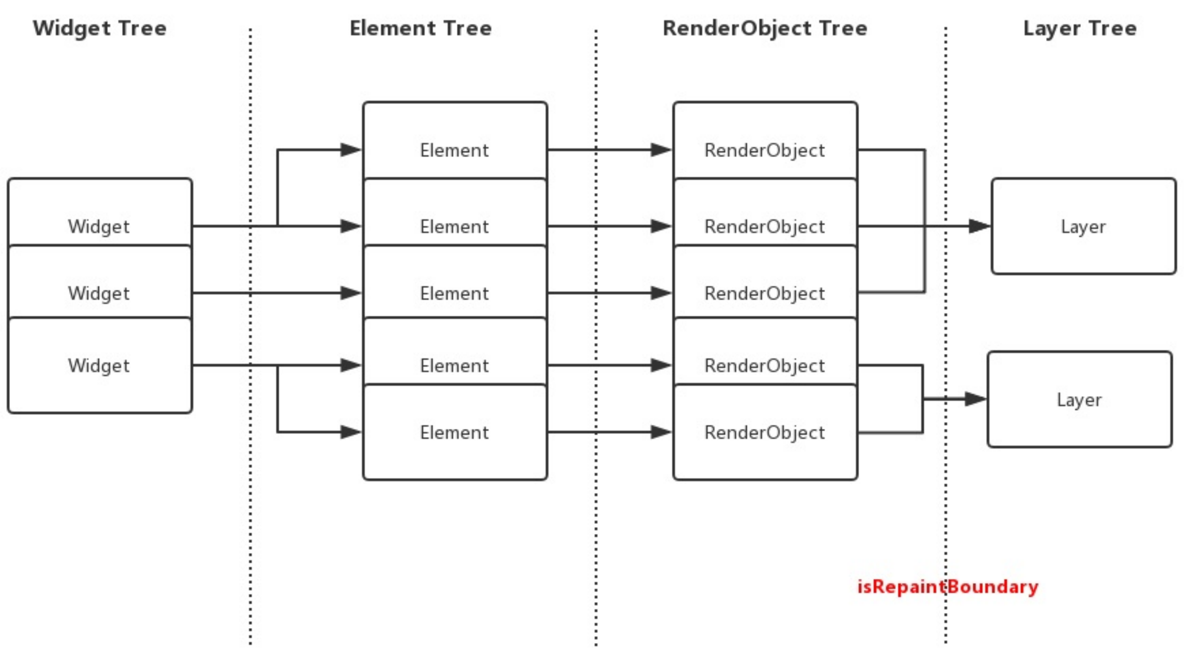
这也体现出 Flutter 在布局上和其他框架不同的地方，你写的 `Widget` 只是配置文件，堆叠嵌套了一堆控件，对最终的 `RenderObject` 而言，可能只是多几个 `Offset` 和 `Size` 计算而已。

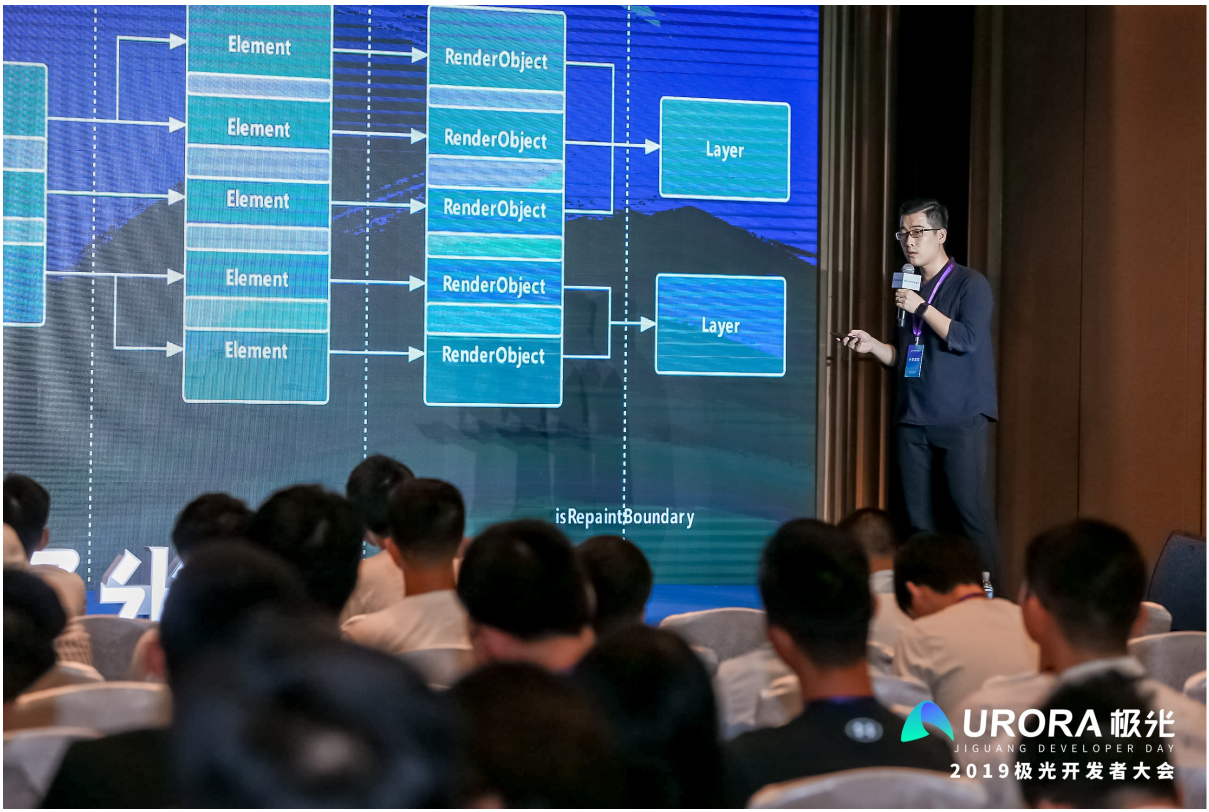
结合上面的理解，可以知道 `Widget` 大部分时候，其实只是轻量级的配置，对于性能问题，**你更需要关心的是 `Clip` 、 `Overlay` 、透明合成等行为，因为它们会需要产生 `saveLayer` 的操作，因为 `saveLayer` 会清空GPU绘制的缓存。**

最后总结个面试点：

- 同一个 `Widget` 可以同时描述多个渲染树中的节点，**作为配置文件是可以复用的。 `Widget` 和 `RenderObject` 一般情况是一对多的关系。** （前提是在 `Widget` 存在 `RenderObject` 的情况。）
- `Element` 是 `Widget` 的某个固定实例，与 `RenderObject` 一一对应。（前提是在 `Element` 存在 `RenderObject` 的情况。）
- `RenderObject` 内 `isRepaintBoundary` 标示使得它们组成了一个个 `Layer` 区域。

当 **`isRepaintBoundary` 为 `true` 时，该区域就是一个可更新绘制区域，而当这个区域形成时，就会新创建一个 `Layer`** 。但不是每个 `RenderObject` 都会有 `Layer` ，因为这受 `isRepaintBoundary` 的影响。

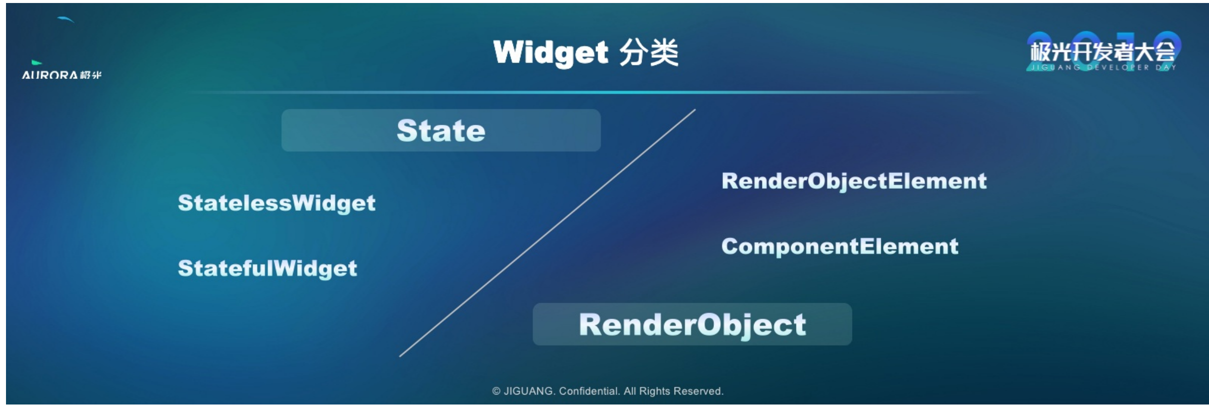




> 注意，Flutter 中常见的 `BuildContext` ，其实就是 `Element` 的抽象，通过 `BuildContext` ，我们一般情况就可以对应获得 `Element` ，也就是拿到了“仓库的钥匙”，通过 `context` 就可以去获取 `Element` 内持有的东西，比如前面所说的 `RenderObject` ，还有后面我们会谈到 `State` 等。

1.2 Widget 的分类

这里我们将 `Widget` 分为如下图所示分类：是否存在 `State` 、是否存在 `RenderObject` 。



> 其实还可以按照 `RenderBox` 和 `RenderSliver` 分类，但是篇幅原因以后再介绍。

1.2.1 是否存在 State

Flutter 中我们常用的 `Widget` 有： `StatelessWidget` 和 `StatefulWidget` 。

如下图， `StatelessWidget` 的代码很简单，因为 `Widget` 是不可变的，传入的 `text` 决定了它显示的内容，并且 `text` 也算是 `final` 的。

```
class DemoPage extends StatelessWidget {
  final String text;
  int i = 0;
  DemoPage({this.text});
  @override
  Widget build(BuildContext context) {
    return Container(
      child: new Text(this.text ?? "null"),
    );
  }
}
```

> 注意图中 `DemoPage` 有个黄色警告，这是因为我们定义了 `int i = 0` 不是 final 导致的，在 `StatelessWidget` 中， **非 final 的变量起始容易产生误解，因为 `Widget` 本事就是不可变的。**

前面我们说过 `Widget` 都是不可变的，在这个基础上， **`StatefulWidget` 的 `State` ，帮我们实现了 `Widget` 的跨帧绘制** ，也就是在每次 `Widget` 重构时，可以通过 `State` 重新赋予 `Widget` 需要的配置信息，而这里的 **`State` 对象，就是存在每个 `Element` 里的。**

> 同时，前面我们说过，Flutter 内的 `BuildContext` 其实就是 `Element` 的抽象，这说明我们可以通过 `context` 去获取 `Element` 内的东西，比如 `State` 、 `RenderObject` 、 `Widget` 。

```
Widget ancestorWidgetOfExactType
State ancestorStateOfType
State rootAncestorStateOfType
RenderObject ancestorRenderObjectOfType
```

如下图所示，保存在 `State` 中的 `text` ，当我们点击按键时， `setState` 时它被标志为 `"变化了"` ， **它可以主动发生改变，保存变量，不再只是“只读”状态了。**

```
class DemoPage extends StatefulWidget {
  @override
  _DemoPageState createState() => _DemoPageState();
}
class _DemoPageState extends State<DemoPage> {
  String text = "初始化";
  @override
  Widget build(BuildContext context) {
    return Container(
      child: FlatButton(
        onPressed: (){
          setState(() {
            text = "变化了";
          });
        },
        child: new Text(this.text ?? "null"),
      ),
    );
  }
}
```

1.2.2、容器 Widget/渲染 Widget

在 Flutter 中还有 **容器 Widget** 和 **渲染Widget** 的区别，一般情况下:

- `Text` 、 `Slider` 、 `ListTile` 等都是属于渲染 `Widget` ，其内部主要是 `RenderObjectElement` ，对应有 `RenderObject` 参数。
- `StatelessWidget` / `StatefulWidget` 等属于容器 `Widget` ，其内部使用的是 `ComponentElement` ， **`ComponentElement` 本身是不存在 `RenderObject` 的。**

所以作为容器 `Widget`， 获取它们的 `RenderObject` 时，获取到的是 `build` 后的树结构里，最上面**渲染 Widget**的 `RenderObject` 。

```
/// The render object at (or below) this location in the tree.
///
/// If this object is a [RenderObjectElement], the render object is the one at
/// this location in the tree. Otherwise, this getter will walk down the tree
/// until it finds a [RenderObjectElement].
RenderObject get renderObject {
  RenderObject result;
  void visit(Element element) {
    assert(result == null); // this verifies that there's only one child
    if (element is RenderObjectElement)
      result = element.renderObject;
    else
      element.visitChildren(visit);
  }
  visit(this);
  return result;
}
```

> 如上图所示 `findRenderObject` 的实现，最终就是获取 `renderObject`，**在遇到 `ComponentElement` 时，执行的是 `element.visitChildren(visit);`**，递归直到找到 `RenderObjectElement` ，再返回它的 `renderObject` 。

获取 `RenderObject` 在 Flutter 里很重要的，因为获取控件的位置和大小等，都需要通过 `RenderObject` 获取。

1.3、RenderObject

Flutter 中各类 `RenderObject` 的实现，大多都是颗粒度很细，功能很单一的存在：

Widget	RenderObject
Align	RenderPositionedBox
Padding	RenderPadding
ConstrainedBox	RenderConstrainedBox
Offstage	RenderOffstage

然而接触过 Flutter 的同学应该知道 **`Container`** 这个 `Widget` ， **`Container`** 的功能却不显单一，这是为什么呢?

如下图，因为 **`Container` 其实是容器 Widget**，它只是把其他“单一”的 Widget 做了二次封装，然后通过配置参数来达到 “多功能的效果” 而已。

```
@override
Widget build(BuildContext context) {
  Widget current = child;

  if (child == null && (constraints == null || !constraints.isTight)) {
    current = LimitedBox(
      maxWidth: 0.0,
      maxHeight: 0.0,
      child: ConstrainedBox(constraints: const BoxConstraints.expand()),
    ); // LimitedBox
  }

  if (alignment != null)
    current = Align(alignment: alignment, child: current);

  final EdgeInsetsGeometry effectivePadding = _paddingIncludingDecoration;
  if (effectivePadding != null)
    current = Padding(padding: effectivePadding, child: current);

  if (decoration != null)
    current = DecoratedBox(decoration: decoration, child: current);

  if (foregroundDecoration != null) {
    current = DecoratedBox(
      decoration: foregroundDecoration,
      position: DecorationPosition.foreground,
      child: current,
    );
  }

  if (constraints != null)
    current = ConstrainedBox(constraints: constraints, child: current);

  if (margin != null)
    current = Padding(padding: margin, child: current);

  if (transform != null)
    current = Transform(transform: transform, child: current);

  return current;
}
```

所以 Flutter 开发中，我们经常会根据功能定义出各类如 `Continer` 、 `Scaffold` 等脚手架模版，实现灵活与复用的界面开发。

回归到 `RenderObject` ，事实上 `RenderObject` 还属于比较“低级”的阶段，因为绘制到屏幕上我们还需要坐标体系和布局协议等，所以 **大部分 `Widget` 的 `RenderObject` 会是子类 `RenderBox` (`RenderSliver` 例外)** ，因为 `RenderObject` 本身只实现了基础的 `layout` 和 `paint` ，而绘制到屏幕上，我们需要的坐标和大小等，这些内容是在 `RenderBox` 中开始实现。

> `RenderSliver` 主要是在滚动控件中继承使用。

比如控件被绘制在 `x=10,y=20` 的位置，然后大小由 `parent` 对它进行约束显示， **`RenderBox` 继承了 `RenderObject` ，在其基础上实现了 `笛卡尔坐标系` 和布局协议。**

这里我们通过 `Offstage` 这个 `Widget` ，看下其 `RenderBox` 子类的实现逻辑， `Offstage` 是用于控制 `child` 是否显示的作用，如下图，可以看到 `RenderOffstage` 对于 `offstage` 标志位的内部逻辑：

```
@override
double computeMinIntrinsicWidth(double height) {
  if (offstage)
    return 0.0;
  return super.computeMinIntrinsicWidth(height);
}

@override
double computeMaxIntrinsicWidth(double height) {
  if (offstage)
    return 0.0;
  return super.computeMaxIntrinsicWidth(height);
}

@override
double computeMinIntrinsicHeight(double width) {
  if (offstage)
    return 0.0;
  return super.computeMinIntrinsicHeight(width);
}

@override
double computeMaxIntrinsicHeight(double width) {
  if (offstage)
    return 0.0;
  return super.computeMaxIntrinsicHeight(width);
}
```

那么 *Flutter* 中的布局协议是什么呢?

简单来说就是 `child` 和 `parent` 之间的大小应该怎么显示，由谁决定显示区域。 相信从 *Android* 到接触 *Flutter* 的同学有这样的疑惑， *Flutter* 中的 `match_parent` 和 `wrap_content` 逻辑需要怎么设置?

就我们从一个简单的代码分析，如下图所示， `Row` 布局我们没有设置任何大小，它是怎么确定自身大小的呢?

```
class DemoRow extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        new Text("FFFFF"),
        new Text("EEEEE"),
        new Text("QQQQQ"),
      ],
    );
  }
}
```

我们翻阅源码，可以发现其实 Flutter 中常用的 `Row` 、 `Column` 等其实都是 `Flex` 的子类，只是对 `Flex` 做了简单默认配置。

```
class Row extends Flex {
  Row({
    Key key,
    MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
    MainAxisSize mainAxisSize = MainAxisSize.max,
    CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,
    TextDirection textDirection,
    VerticalDirection verticalDirection = VerticalDirection.down,
    TextBaseline textBaseline,
    List<Widget> children = const <Widget>[],
  }) : super(
    children: children,
    key: key,
    direction: Axis.horizontal,
    mainAxisAlignment: mainAxisAlignment,
    mainAxisSize: mainAxisSize,
    crossAxisAlignment: crossAxisAlignment,
    textDirection: textDirection,
    verticalDirection: verticalDirection,
    textBaseline: textBaseline,
  );
}
```

那按照我们前面的理解，**看一个 `Widget` 的实现逻辑，就应该看它的 `RenderObject`** ，而在 `Flex` 布对应的 `RenderFlex` 中，我们可以看到如下一段代码：

```
@override
void performLayout() {
  assert(constraints != null);
  final double maxMainSize = _direction == Axis.horizontal ? constraints.maxWidth : constraints.maxHeight;
  final bool canFlex = maxMainSize < double.infinity;

  double crossSize = 0.0;
  double allocatedSize = 0.0; // Sum of the sizes of the non-flexible children.
  RenderBox child = firstChild;
  RenderBox lastFlexChild;
  //...
  final double idealSize = canFlex && mainAxisSize == MainAxisSize.max ? maxMainSize : allocatedSize;
  double actualSize;
  double actualSizeDelta;
  switch (_direction) {
    case Axis.horizontal:
      size = constraints.constrain(Size(idealSize, crossSize));
      actualSize = size.width;
      crossSize = size.height;
      break;
    case Axis.vertical:
      size = constraints.constrain(Size(crossSize, idealSize));
      actualSize = size.height;
      crossSize = size.width;
      break;
  }
```

可以看到在布局的时候，`RenderFlex` 首先要求 `constraints != null`，**`Flex` 布局的上层中必须存在约束，不然肯定会报错。**

之后，在布局时，`Row` 布局的 `direction` 是横向的，所以 `maxMainSize` 为上层布局的最大宽度，然后根据我们配置的 `mainAxisSize` 的参数：

- 当 `mainAxisSize` 为 `max` 时，我们 `Row` 的横向布局就是 `maxMainSize`；
- 当 `mainAxisSize` 为 `min` 时，我们 `Row` 的横向布局就是 `allocatedSize`；

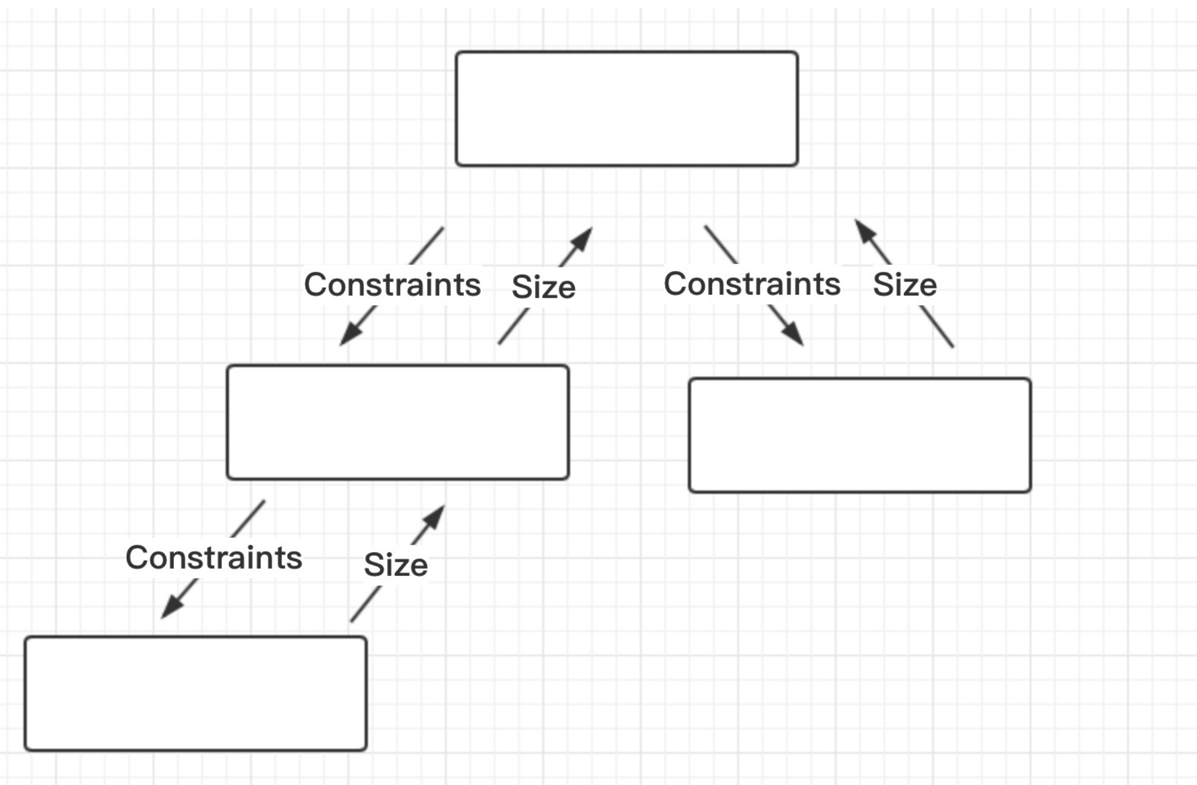
前面 `maxMainSize` 我们知道了是父布局的最大宽度，而 `allocatedSize` 其实就是 child 的宽度之和。所以结果很明显了：

对于 `Row` 来说，`mainAxisSize` 为 `max` 时就是 `match_parent`；`mainAxisSize` 为 `min` 时就是 `wrap_content`。

而高度 `crossSize`，**其实是由 `math.max(crossSize, _getCrossSize(child));` 决定，也就是 `child` 中最高的一个作为其高度。**

最后小结一个知识点：

布局一般都是由上层往下传递 `Constraints`，然后由下往上返回 `Size`。

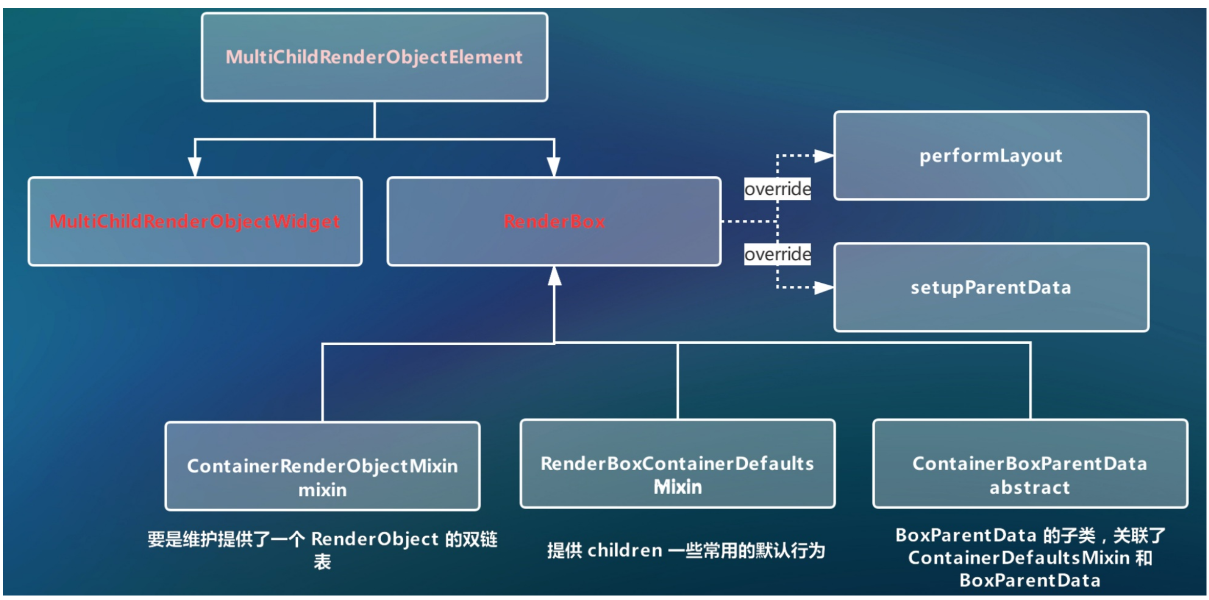


那如何直接自定义 *`RenderObject`* 布局?

抛开 Flutter 为我们封装的好的，三大金刚 `Widget` 、 `Element` 、 `RednerObject` 一个不少，当然， Flutter 内置了很多封装帮我们节省代码。

一般情况下自定义 `RenderObject` 布局：

- **我们会继承 `MultiChildRenderObjectWidget` 和 `RenderBox` 这两个 `abstract` 类，实现自己的 `Widget` 和 `RenderObject` 对象；**
- **然后利用 `MultiChildRenderObjectElement` 关联起它们；**
- 除此之外，还有几个关键的类： **`ContainerRenderObjectMixin`** 、 **`RenderBoxContainerDefaultsMixin`** 和 **`ContainerBoxParentData`** 等可以帮你减少代码量。



总结起来， 对于 Flutter 而言，整个屏幕都是一块画布，我们通过各种 `Offset` 和 `Rect` 确定了位置，然后通过 `Canvas` 绘制上去，目标是整个屏幕区域，整个屏幕就是一帧，每次改变都是重新绘制。

> 这里没有介绍 `RenderSliver` 相关，它的输入和输出和 `Renderbox` 又不大一样，有机会我们后面再详细介绍。

三、Flutter 的实战技巧

3.1、InheritedWidget

`InheritedWidget` 是 Flutter 的灵魂设定之一。

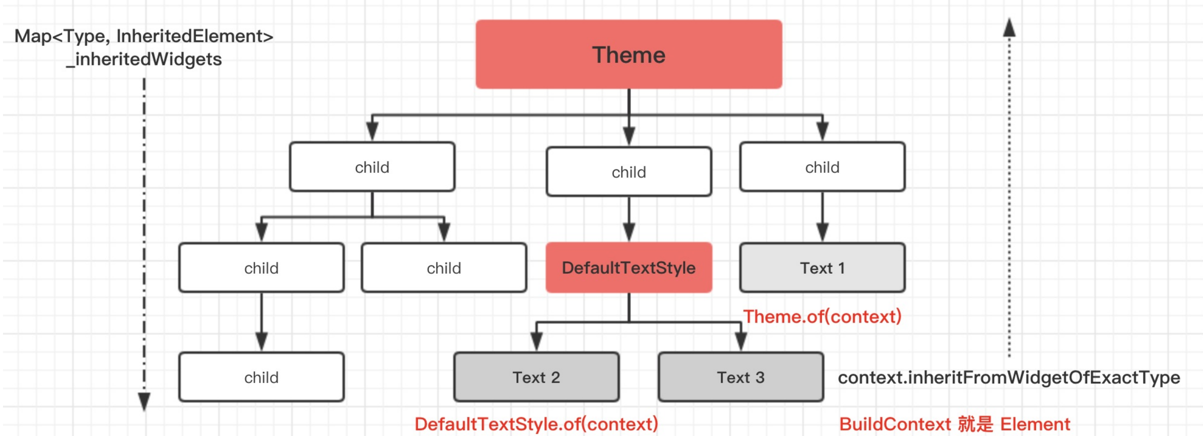
`InheritedWidget` 共享的是 `Widget` ，只是这个 `Widget` 是一个 `ProxyWidget` ，它自己本身并不绘制什么，但共享这个 `Widget` 内保存有的数据，从而到了共享状态的目的。

如下图所示，是 Flutter 中常见的 `Theme` ，其内部就是使用了 `_InheritedTheme` 这个 `InheritedWidget` 来实现主题的全局共享的。那么 `InheritedWidget` 是如何实现全局共享的呢？

```
class _InheritedTheme extends InheritedWidget {
  const _InheritedTheme({
    Key key,
    @required this.theme,
    @required Widget child,
  }) : assert(theme != null),
       super(key: key, child: child);

  final Theme theme;

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

其实在 `Element` 的内部有一个 `Map<Type, InheritedElement> _inheritedWidgets;` 参数，`_inheritedWidgets` **一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidget` 时，它才会被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并。**

所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 时，就可以通过这个 `Map` 往上查找，从而找到这个上级的 `InheritedWidget` 。（毕竟 `context` is `Element` ）



如我们的 `Theme` / `ThemeData` 、 `Text` / `DefaultTextStyle` 、 `Slider` / `SliderTheme` 等，如下代码所示，我们可以定义全局的 `ThemeData` 或者局部的 `DefaultTextStyle` ，从而实现全局的自定义和局部的自定义共享等。

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'GSY Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'GSY Flutter Demo'),
      routes: routers,
    );
  }
}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: new Text("TextLineHeightDemoPage"),
    ),
    body: Container(
      child: DefaultTextStyle(
        style: TextStyle(fontSize: 14, color: Colors.white),
        child: new Column(
          children: <Widget>[
            new Text("文本1"),
            new Text("文本2"),
            new Text("文本3"),
          ],
        ),
      ),
    ),
  );
}
```

> 其实，Flutter 中大部分的状态管理控件，其状态共享方法，也是基于 `InheritedWidget` 去实现的。

3.2、支持原生控件

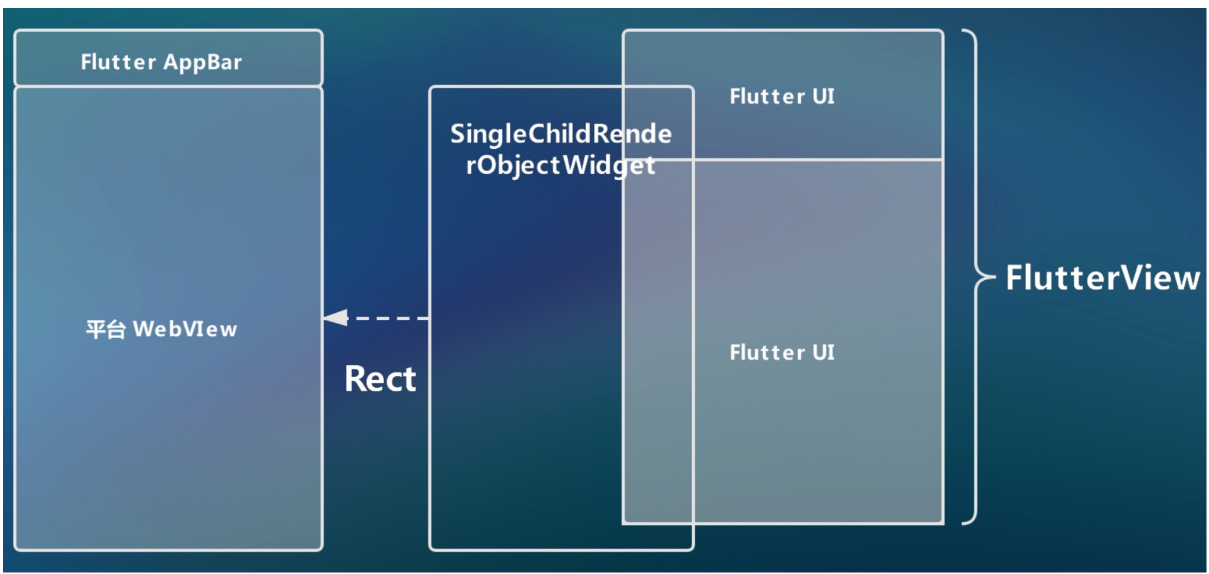
前面我们说过， Flutter 既然不依赖于原生控件，那么如何集成一些平台已有的控件呢？比如 `WebView` 和 `Map` ？

我们这里以 `WebView` 为例子：

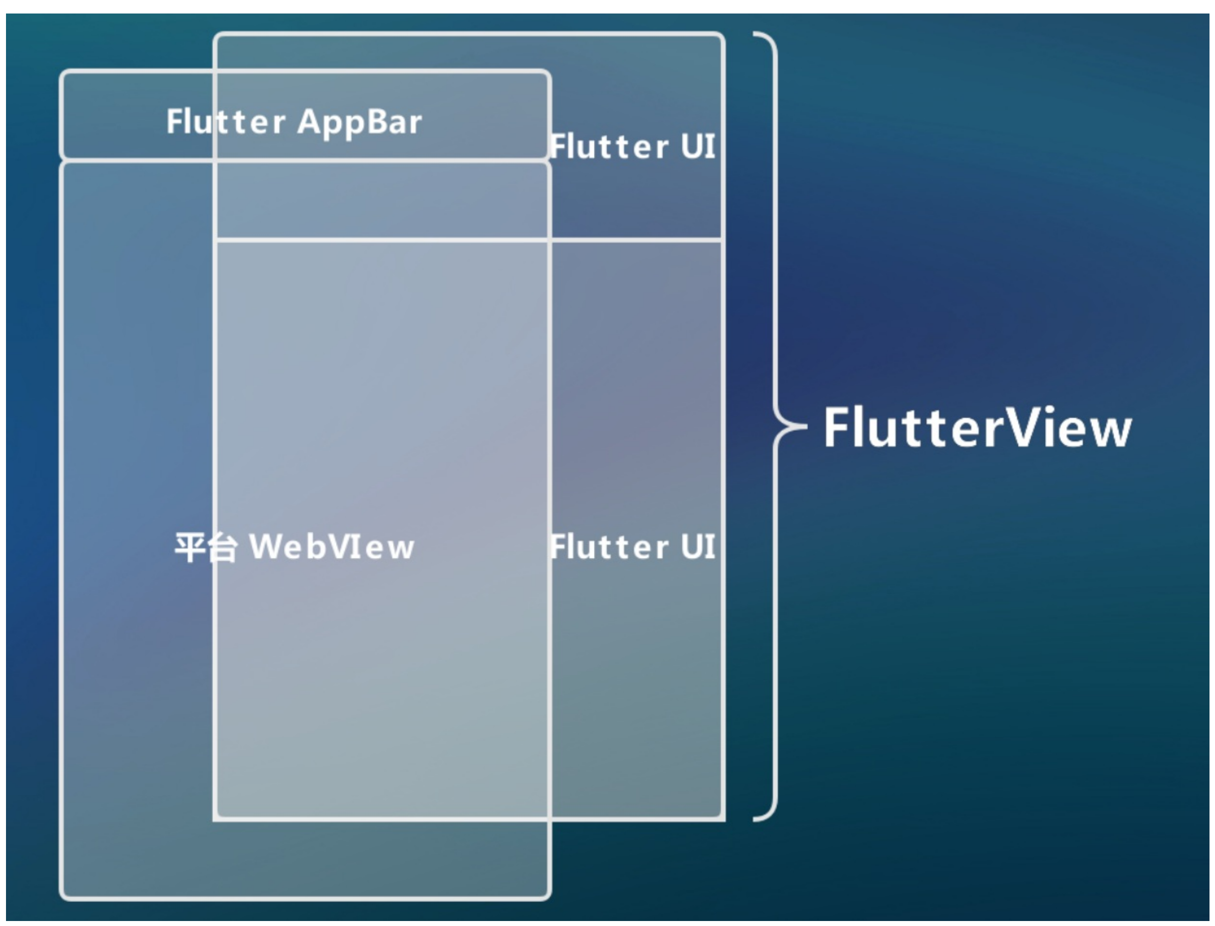
在官方 `WebView` 控件支持出来之前，第三方是直接在 FlutterView 上覆盖了一个新的原生控件，利用 Dart 中的占位控件**传递位置和大小**。

如下图，在 Flutter 端 `push` 出一个 **设定好位置和大小** 的 `SingleChildRenderObjectWidget` ，从而得到需要显示的大小和位置，将这些信息通过 `MethodChannel` 传递到原生层，在原生层 `addContentView` 一个指定大小和位置的 `WebView` 。

这时候 `WebView` 和 `SingleChildRenderObjectWidget` 处于一样的大小和位置，而空白部分则用 FLutter 的 `Appbar` 显示。



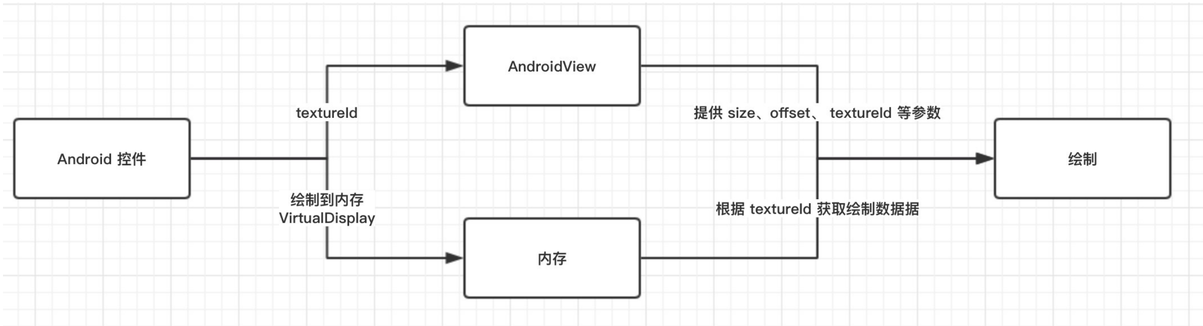
这样看起来就像是在 Flutter 中添加了 `WebView` ，但实际这脱离了 Flutter 的渲染树，其中一个问题就是，当你跳转 Flutter 其他页面的时候，会被 `WebView` 挡住；并且打开页面的动画， `Appbar` 和 `WebView` 难以保持一致。



后面 官方 `WebView` 控件支持出来后，这时候官方是利用 `PlatformView` 的设计，完成了不脱离 Flutter 渲染堆栈，也能集成平台原生控件的功能。

以 Android 为例，Android 上是利用了副屏显示的底层逻辑，使用 `VirtualDisplay` 类，创建一个虚拟显示器，需要调用 `DisplayManager` 的 `createVirtualDisplay()` 方法，将虚拟显示器的内容渲染在一个内存的 `Surface` 上，生成一个唯一的 `textureId` 。

如下图，之后渲染时将 `textureId` 传递给 `Dart` 层，渲染引擎会根据 `textureId` ，获取到内存里已渲染数据，绘制到 `AndroidView` 上进行显示。



3.3、错误处理

Flutter 中比较有趣的情况是，在 Dart 中的一些错误，并不会导致应用闪退，而是通过如下的红色堆栈 UI ，错误区域不同，可能是全屏红，也可能局部红，这种状态就和传统 APP 的“崩溃”状态不大一样了。

```
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
and a decoration
The color argument is just a shorthand for "decoration: new
BoxDecoration(color: color)".

'package:flutter/src/widgets/container.dart': Failed assertion: line 317
pos 15: 'color == null || decoration == null': Cannot provide both a color
```

在开发过程中这样的显示没太大问题，但事实发布线上版本就不合适了，所以我们一般会选择自定义错误显示。

如下图所示，一般我们可以通过如下处理，自定义我们的错误页面，并且收集错误信息。

```
void main() {
  runZoned(() {
    ErrorWidget.builder = (FlutterErrorDetails details) {
      Zone.current.handleUncaughtError(details.exception, details.stack);
      return Container(
          color: Colors.transparent
      );
    };
    runApp(FlutterReduxApp());
  }, onError: (Object obj, StackTrace stack) {
    print(obj);
    print(stack);
  });
}
```

重写 `ErrorWidget` 的 `builder` 方法，然后将信息收集到 `Zone` 中，返回自己的自定义错误显示，最后在 `Zone` 内利用 `onError` 统一处理错误。

> ps 图中的 `Zone` 等概念这里就不展开了，有兴趣的可以去以前的文章详细查看。

四、Flutter Web

最后简单说下 Flutter Web ， Flutter 在支持 Web 平台上的优势在于 Flutter UI 与平台的耦合度很低，而 Dart 起初就是为了 Web 而生，一拍即合下 Flutter 支持 Web 并不是什么意外。

但是 Web 平台就绕不过 JS ， 在 Web 平台，实际上 `Image` 控件最后会通过 dart2js 转化为 `<img>` 标签并通过 `src` 赋值显示。

```
/// This function is used by [load].
@protected
Future<ui.Codec> _loadAsync(AssetBundleImageKey key) async {
  final ByteData data = await key.bundle.load(key.name);
  if (data == null) throw 'Unable to read data';

  ///       内部实际是
  ///       final html.ImageElement imgElement = html.ImageElement();
  ///       imgElement.src = src;
  return await ui.instantiateImageCodec(data.buffer.asUint8List());
}
```
Web

```
Future<ui.Codec> _loadAsync(NetworkCacheImage key) async {

  final HttpClientRequest request = await _httpClient.getUrl(resolved);
  headers?.forEach((String name, String value) {
    request.headers.add(name, value);
  });
  return PaintingBinding.instance.instantiateImageCodec(bytes);
}

@override
```
Flutter

同时，多了一个平台就多了需要兼容的，目前 Flutter 的 issue 仍然不少，而 Web 支持虽然已经合并到主项目中，但是在兼容、性能等问题上还需要继续优化，比如 Flutter Web 中 `canvas.drawColor(Colors.black, BlendMode.clear);` 是会出现运行错误的，因为不支持 `BlendMode.clear` 。

资源推荐

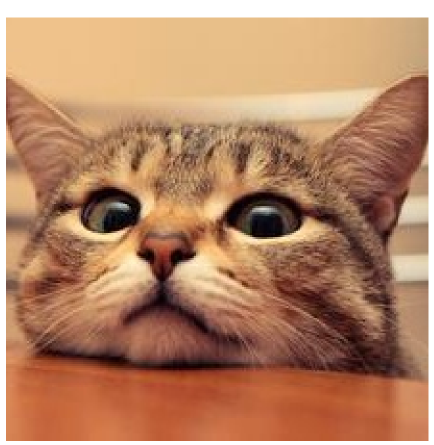
- Github ： https://github.com/CarGuo
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp

其他文章

《Flutter完整开发实战详解系列》

《移动端跨平台开发的深度解析》

《全网最全Flutter与React Native深入对比分析》



谷歌大会之后，有不少人咨询了我 **Flutter** 相关的问题，其中有不少是和面试相关的，如今一些招聘上也开始罗列 **Flutter** 相关要求，最后想了想还是写一期总结吧，也算是 **Flutter** 的阶段复习。

> ⚠系统完整的学习是必须需要的，这里只能帮你总结一些知识点，更多的还请查阅 `Dart/Flutter` 官网。

本篇主要是知识点总结，如有疑问可点击各文章链接了解详情，或者查阅我 掘金专栏。

Dart 部分

其实学习过 `JavaScript` 或者 `Java/Kotlin` 的人，在学习 `Dart` 上几乎是没什么难度的，**Dart 综合了动态语言和静态语言的特性,** 这里主要提供一些不一样，或者有意思的概念。

- 1、 `Dart` 属于是**强类型语言** ，但可以用 `var` 来声明变量， `Dart` 会**自推导出数据类型**， `var` 实际上是编译期的“语法糖”。 **`dynamic` 表示动态类型**， 被编译后，实际是一个 `object` 类型，在编译期间不进行任何的类型检查，而是在运行期进行类型检查。
- 2、 `Dart` 中 `if` 等语句只支持 `bool` 类型， `switch` 支持 String 类型。
- 3、 `Dart` **中数组和 `List` 是一样的。**
- 4、 `Dart` 中， **`Runes` 代表符号文字** , 是 UTF-32 编码的字符串, 用于如 `Runes input = new Runes('\u{1f596} \u{1f44d}');`
- 5、 **`Dart` 支持闭包。**
- 6、 `Dart` 中 number 类型分为 **int 和 double ， 没有 float 类型。**
- 7、 `Dart` 中 **级联操作符** 可以方便配置逻辑，如下代码：

```
event
  ..id = 1
  ..type = ""
  ..actor = "";
```

- 8、赋值操作符

比较有意思的赋值操作符有：

```
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
AA ~/999 ///AA 对于 999 整除
```

- 9、可选方法参数

`Dart` 方法可以设置 **参数默认值** 和 **指定名称** 。

比如： `getDetail(Sting 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");` 。

- 10、作用域

`Dart` 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 。

- 11、构造方法

`Dart` 中的多构造方法，可以通过命名方法实现。

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);
}
```

- 12、getter setter 重写

`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， `Object` 都支持 getter、setter 重写：

```
  @override
  Size get preferredSize {
    return Size.fromHeight(kTabHeight + indicatorWeight);
  }
```

- 13、Assert(断言)

`assert` 只在检查模式有效，在开发过程中， `assert(unicorn == null);` 只有条件为真才正常，否则直接抛出异常，一般用在开发过程中，某些地方不应该出现什么状态的判断。

- 14、重写运算符，如下所示重载 `operator` 后对类进行 +/- 操作。

```
class Vector {
```

```
  final int x, y;

  Vector(this.x, this.y);

  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

  ...
}

void main() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);

  assert(v + w == Vector(4, 5));
  assert(v - w == Vector(0, 1));
}
```

支持重载的操作符：

<table>
<tr><td><</td><td>+</td><td>|</td><td>[]</td></tr>
<tr><td>></td><td>/</td><td>^</td><td>[]=</td></tr>
<tr><td><=</td><td>~/</td><td>&</td><td>~</td></tr>
<tr><td>>=</td><td>*</td><td><<</td><td>==</td></tr>
<tr><td>–</td><td>%</td><td>>></td><td></td></tr>
</table>

- 类、接口、继承

`Dart` 中没有接口，类都可以作为接口，把某个类当做接口实现时，只需要使用 `implements` ，然后复写父类方法即可。

`Dart` 中支持 `mixins` ，按照出现顺序应该为 `extends` 、 `mixins` 、 `implements` 。

- ## Zone

`Dart` 中可通过 `Zone` 表示指定代码执行的环境，类似一个沙盒概念，在 `Flutter` 中 **C++** 运行 `Dart` 也是在 `_runMainZoned` 内执行 `runZoned` 方法启动，而我们也可以通过 `Zone` ，在运行环境内捕获全局异常等信息:

```
  runZoned(() {
    runApp(FlutterReduxApp());
  }, onError: (Object obj, StackTrace stack) {
    print(obj);
    print(stack);
  });
```

同时你可以给 `runZoned` 注册方法，在需要时执行回调，如下代码所示，这样的在一个 `Zone` 内任何地方，只要能获取 `onData` 这个 `ZoneUnaryCallback` ，就都可以调用到 `handleData`

```
///最终需要处理的地方
handleData(result) {
  print("VVVVVVVVVVVVVVVVVVVVVVVVVVV");
  print(result);
}

///返回得到一个 ZoneUnaryCallback
var onData = Zone.current.registerUnaryCallback<dynamic, int>(handleData);

///执行 ZoneUnaryCallback 返回数据
Zone.current.runUnary(onData, 2);
```

异步逻辑可以通过 `scheduleMicrotask` 可以插入异步执行方法：

```
Zone.current.scheduleMicrotask((){
  //todo something
});
```

更多可参看： 《Flutter完整开发实战详解(十一、全面深入理解Stream)》

- ## Future

`Future` 简单了说就是对 `Zone` 的封装使用。

比如 `Future.microtask` 中主要是执行了 `Zone` 的 `scheduleMicrotask` ，而 `result._complete` 最后调用的是 `_zone.runUnary` 等等。

```
  factory Future.microtask(FutureOr<T> computation()) {
    _Future<T> result = new _Future<T>();
    scheduleMicrotask(() {
      try {
        result._complete(computation());
      } catch (e, s) {
        _completeWithErrorCallback(result, e, s);
      }
    });
    return result;
  }
```

`Dart` 中可通过 `async` / `await` 或者 `Future` 定义异步操作，而事实上 `async` / `await` 也只是语法糖，最终还是通过编译器转为 `Future` 。

> 有兴趣看这里：

> generators
>
> code_generator.dart
>
> Flutter完整开发实战详解(十一、全面深入理解Stream)

- ## Stream

`Stream` 也是有对 `Zone` 的另外一种封装使用。

Dart 中另外一种异步操作， `async*` / `yield` 或者 `Stream` 可定义 `Stream` 异步， `async*` / `yield` 也只是语法糖，最终还是通过编译器转为 `Stream` 。 Stream 还支持同步操作。

1）、 `Stream` 中主要有 `Stream` 、 `StreamController` 、 `StreamSink` 和 `StreamSubscription` 四个关键对象，大致可以总结为：

- **`StreamController`** ：如类名描述，用于整个 `Stream` 过程的控制，提供各类接口用于创建各种事件流。
- **`StreamSink`** ：一般作为事件的入口，提供如 `add` ， `addStream` 等。
- **`Stream`** ：事件源本身，一般可用于监听事件或者对事件进行转换，如 `listen` 、 `where` 。
- **`StreamSubscription`** ：事件订阅后的对象，表面上用于管理订阅过等各类操作，如 `cacenl` 、 `pause` ，同时在内部也是事件的中转关键。

2)、**一般通过 `StreamController` 创建 `Stream` ；通过 `StreamSink` 添加事件；通过 `Stream` 监听事件；通过 `StreamSubscription` 管理订阅。**

3）、 `Stream` 中支持各种变化，比如 `map` 、 `expand` 、 `where` 、 `take` 等操作，同时支持转换为 `Future` 。

更多可参看： 《Flutter完整开发实战详解(十一、全面深入理解Stream)》

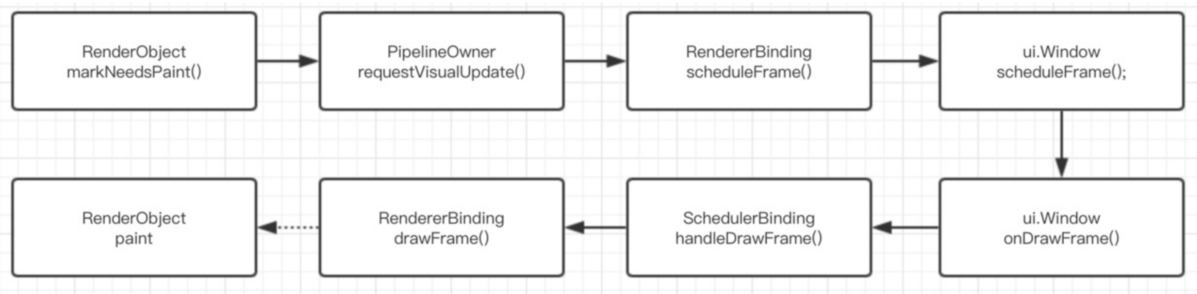
Flutter 部分

Flutter 和 React Native 不同主要在于 **Flutter UI是直接通过 skia 渲染的 ，而 React Native 是将 js 中的控件转化为原生控件，通过原生去渲染的** ，相关更多可查看：《移动端跨平台开发的深度解析》。

- Flutter 中存在 `Widget` 、 `Element` 、 `RenderObject` 、 `Layer` 四棵树，其中 **`Widget` 与 `Element` 是一对多的关系** ，
- `Element` 中持有 `Widget` 和 `RenderObject` ， 而 **`Element` 与 `RenderObject` 是一一对应的关系（除去 `Element` 不存在 `RenderObject` 的情况，如 `ComponentElement` 是不具备 `RenderObject`)** ，
- 当 `RenderObject` 的 `isRepaintBoundary` 为 `true` 时，那么个区域形成一个 `Layer` ，所以**不是每个 `RenderObject` 都具有 `Layer` 的，因为这受 `isRepaintBoundary` 的影响。**

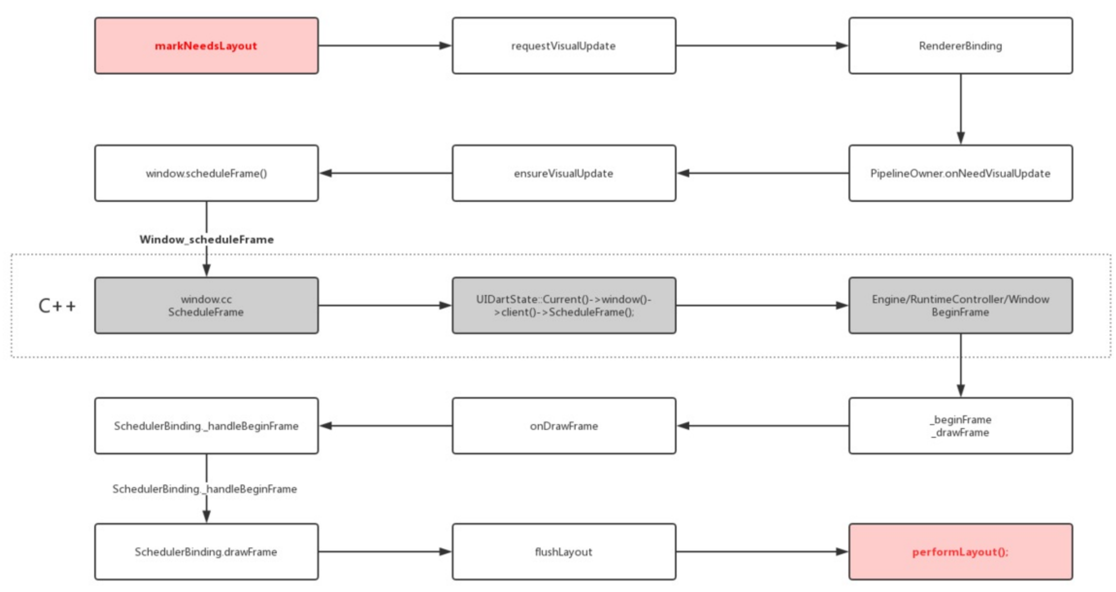
> 更多相关可查阅 《Flutter完整开发实战详解(九、 深入绘制原理)》

- Flutter 中 `Widget` 不可变，每次保持在一帧，如果**发生改变是通过 `State` 实现跨帧状态保存，而真实完成布局和绘制数组的是 `RenderObject` ，** `Element` 充当两者的桥梁， **`State` 就是保存在 `Element` 中。**
- **Flutter 中的 `BuildContext` 只是接口，而 `Element` 实现了它。**
- Flutter 中 **`setState` 其实是调用了 `markNeedsBuild`** ，该方法内部**标记此 `Element` 为 `Dirty`** ，然后在下一帧 `WidgetsBinding.drawFrame` 才会被绘制，这可以看出 **`setState` 并不是立即生效的。**
- Flutter 中 `RenderObject` 在 `attch` / `layout` 之后会通过 `markNeedsPaint();` 使得页面重绘，流程大概如下：



通过isRepaintBoundary 往上确定了更新区域，通过 requestVisualUpdate 方法触发更新往下绘制。

- 正常情况 `RenderObject` 的布局相关方法调用顺序是： **`layout` -> `performResize` -> `performLayout` -> `markNeedsPaint`** ，但是用户一般不会直接调用 `layout` ，而是通过 **`markNeedsLayout`** ，具体流程如下：



- Flutter 中一般 **json** 数据从 `String` 转为 `Object` 的过程中都需要先经过 `Map` 类型。

- Flutter 中 `InheritedWidget` 一般用于**状态共享**，如 `Theme` 、 `Localizations` 、 `MediaQuery` 等，都是通过它实现共享状态，这样我们可以通过 `context` 去获取共享的状态，比如 `ThemeData theme = Theme.of(context);`

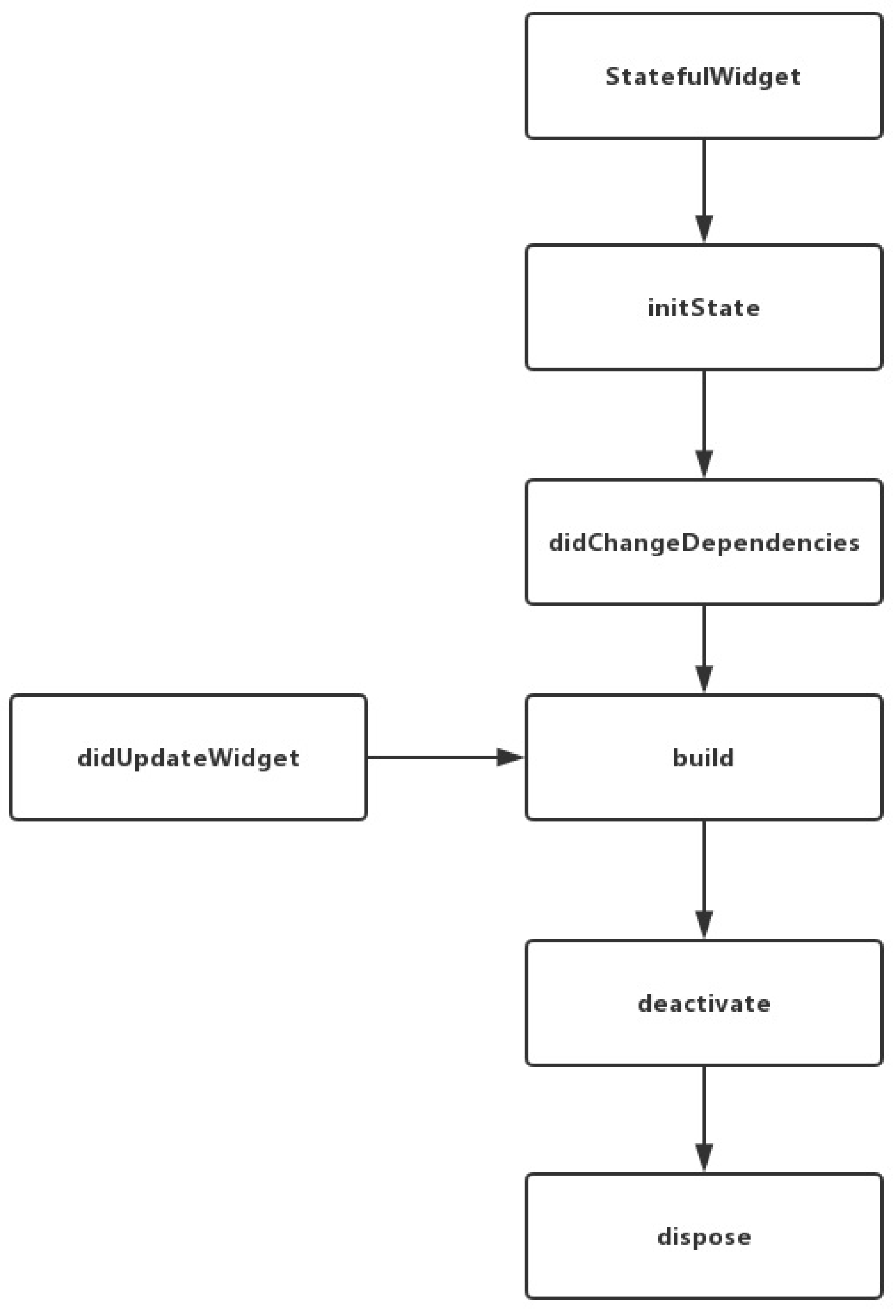
> 在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement> _inheritedWidgets` 的对象。
>
> `_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并 。
>
> 所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 时，就可以往上查找到父控件的 `Widget` 。

- Flutter 中默认主要通过 **`runtimeType` 和 `key`** 判断更新：

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

Flutter 中的生命周期

- **`initState()`** 表示当前 `State` 将和一个 `BuildContext` 产生关联，但是此时 `BuildContext` 没有完全装载完成，如果你需要在该方法中获取 `BuildContext` ，可以 `new Future.delayed(const Duration(seconds: 0, (){//context});` 一下。
- **`didChangeDependencies()`** 在 `initState()` 之后调用，当 `State` 对象的依赖关系发生变化时，该方法被调用，初始化时也会调用。
- **`deactivate()`** 当 `State` 被暂时从视图树中移除时，会调用这个方法，同时页面切换时，也会调用。
- **`dispose()`** Widget 销毁了，在调用这个方法之前，总会先调用 deactivate()。
- **`didUpdateWidge`** 当 `widget` 状态发生变化时，会调用。



- 通过 `StreamBuilder` 和 `FutureBuilder` 我们可以快速使用 `Stream` 和 `Future` 快速构建我们的异步控件：《Flutter完整开发实战详解(十一、全面深入理解Stream)》

- Flutter 中 `runApp` 启动入口其实是一个 `WidgetsFlutterBinding` ，它主要是通过 `BindingBase` 的子类 `GestureBinding` 、 `ServicesBinding` 、 `SchedulerBinding` 、 `PaintingBinding` 、 `SemanticsBinding` 、 `RendererBinding` 、 `WidgetsBinding` 等，通过 `mixins` 的组合而成的。
- Flutter 中的 Dart 的线程是以**事件循环和消息队列**的形式存在，包含两个任务队列，**一个是 microtask 内部队列，一个是 event 外部队列，而 microtask 的优先级又高于 event 。**

> 因为 microtask 的优先级又高于 event， 同时会阻塞event 队列，所以如果 microtask 太多就可能会对触摸、绘制等外部事件造成阻塞卡顿哦。

- Flutter 中存在**四大线程，分别为 `UI Runner` 、 `GPU Runner` 、 `IO Runner` ， `Platform Runner` （原生主线程）**，同时在 Flutter 中可以通过 `isolate` 或者 `compute` 执行真正的跨线程异步操作。

PlatformView

Flutter 中通过 `PlatformView` 可以嵌套原生 `View` 到 `Flutter` UI 中，这里面其实是使用了 `Presentation` + `VirtualDisplay` + `Surface` 等实现的，大致原理就是：

使用了类似副屏显示的技术， `VirtualDisplay` 类代表一个虚拟显示器，调用 `DisplayManager` 的 `createVirtualDisplay()` 方法，将虚拟显示器的内容渲染在一个 `Surface` 控件上，然后将 `Surface` 的 id 通知给 Dart，让 engine 绘制时，在内存中找到对应的 `Surface` 画面内存数据，然后绘制出来。em... **实时控件截图渲染显示技术。**

- **Flutter 的 Debug 下是 JIT 模式，release下是AOT模式。**
- Flutter 中可以通过 `mixins AutomaticKeepAliveClientMixin` ，然后重写 `wantKeepAlive` 保持住页面，记得在被保持住的页面 `build` 中调用 `super.build` 。（因为 mixins 特性）。
- **Flutter 手势事件主要是通过竞技判断的：**

主要有 `hitTest` 把所有需要处理的控件对应的 `RenderObject` ， 从 `child` 到 `parent` 全部组合成列表，从最里面一直添加到最外层。

然后从队列头的 child 开始 for 循环执行 `handleEvent` 方法，执行 `handleEvent` 的过程不会被拦截打断。

一般情况下 Down 事件不会决出胜利者，大部分时候是在 MOVE 或者 UP 的时候才会决出胜利者。

竞技场关闭时只有一个的就直接胜出响应，没有胜利者就拿排在队列第一个强制胜利响应。

同时还有 `didExceedDeadline` 处理按住时的 Down 事件额外处理，同时手势处理一般在 `GestureRecognizer` 的子类进行。

> 更多详细请查看：《Flutter完整开发实战详解(十三、全面深入触摸和滑动原理)》

- Flutter 中 `ListView` 滑动其实都是通过改变 `ViewPort` 中的 `child` 布局来实现显示的。

- 常用状态管理的：目前有 `scope_model` 、 `flutter_redux` 、 `fish_redux` 、 `bloc + Stream` 等几种模式，具体可见：《Flutter完整开发实战详解(十二、全面深入理解状态管理设计)》

Platform Channel

Flutter 中可以通过 `Platform Channel` 让 Dart 代码和原生代码通信的：

> - `BasicMessageChannel` ：用于传递字符串和半结构化的信息。
> - `MethodChannel` ：用于传递方法调用（method invocation）。
> - `EventChanne` l: 用于数据流（event streams）的通信。

同时 `Platform Channel` 并非是线程安全的，更多详细可查阅闲鱼技术的 《深入理解Flutter Platform Channel》

其中基础数据类型映射如下：

Dart	Android	iOS	Type
null	null	nil (NSNull when nested)	0
bool	java.lang.Boolean	NSNumber numberWithBool:	1=true 2=false
int	java.lang.Integer	NSNumber numberWithInt:	3
int, if 32 bits not enough	java.lang.Long	NSNumber numberWithLong:	4
int, if 64 bits not enough	java.math.BigInteger	FlutterStandardBigInteger	5
double	java.lang.Double	NSNumber numberWithDouble:	6
String	java.lang.String	NSString	7
Uint8List	byte[]	FlutterStandardTypedData typedDataWithBytes:	8
Int32List	int[]	FlutterStandardTypedData typedDataWithInt32:	9
Int64List	long[]	FlutterStandardTypedData typedDataWithInt64:	10
Float64List	double[]	FlutterStandardTypedData typedDataWithFloat64:	11
List	java.util.ArrayList	NSArray	12
Map	java.util.HashMap	NSDictionary	13

Android 启动页

Android 中 `Flutter` 默认启动时会在 `FlutterActivityDelegate.java` 中读取 AndroidManifset.xml 内 `meta-data` 标签，其中 `io.flutter.app.android.SplashScreenUntilFirstFrame` 标志位如果为 ture，就会启动 Splash 画面效果（类似IOS的启动页面）。

启动时原生代码会读取 `android.R.attr.windowBackground` 得到指定的 `Drawable`，用于显示启动闪屏效果，之后并且通过 `flutterView.addFirstFrameListener`，在 `onFirstFrame` 中移除闪屏。

> 好了，暂时都这里了，有问题修改会或则补充的，后面再加上。

资源推荐

- Github： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

完整开源项目推荐：

- GSYGithubApp Flutter
- GSYGithubApp React Native
- GSYGithubAppWeex

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)》

《Flutter完整开发实战详解(六、 深入Widget原理)》

《Flutter完整开发实战详解(七、 深入布局原理)》

《Flutter完整开发实战详解(八、 实用技巧与填坑)》

《Flutter完整开发实战详解(九、 深入绘制原理)》

《Flutter完整开发实战详解(十、 深入图片加载流程)》

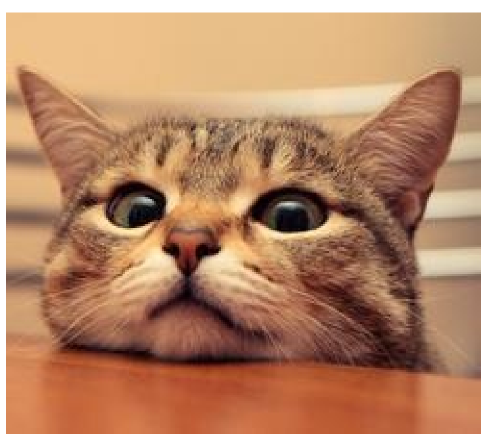
《Flutter完整开发实战详解(十一、全面深入理解Stream)》

《Flutter完整开发实战详解(十二、全面深入理解状态管理设计)》

《Flutter完整开发实战详解(十三、全面深入触摸和滑动原理)》

《跨平台项目开源项目推荐》

《移动端跨平台开发的深度解析》



作为 GSY 开源系列的作者，在去年也整理过 《移动端跨平台开发的深度解析》 的对比文章，时隔一年之后，本篇将重新由 **环境搭建、实现原理、编程开发、插件开发、编译运行、性能稳定、发展未来** 等七个方面，对当前的 **React Native** 和 **Flutter** 进行全面的分析对比，希望能给你更有价值的参考。

> 是的，这次没有了 Weex，**超长内容预警，建议收藏后阅。**

前言

临冬之际，移动端跨平台在经历数年沉浮之后，如今还能在舞台聚光灯下雀跃的， 也只剩下 **React Native** 和 **Flutter** 了，作为沉淀了数年的 “豪门” 与 19 年当红的 “新贵” ，它们之间的 “针锋相对” 也成了开发者们关心的事情。

> 过去曾有人问我：“他即写 *Java* 又会 *Object-C* ，在 *Android* 和 *IOS* 平台上可以同时开发，为什么还要学跨平台呢？ ”
>
> 而我的回答是：**跨平台的市场优势不在于性能或学习成本，甚至平台适配会更耗费时间，但是它最终能让代码逻辑（特别是业务逻辑），无缝的复用在各个平台上，降低了重复代码的维护成本，保证了各平台间的统一性，** 如果这时候还能保证一定的性能，那就更完美了。

类型	React Native	Flutter
语言	JavaScript	dart
环境	JSCore	Flutter Engine
发布时间	2015	2017
star	78k+	67k+
对比版本	0.59.9	1.6.3
空项目打包大小	Android 20M(可调整至 7.3M) / IOS 1.6M	Android 5.2M / IOS 10.1M
GSY项目大小	Android 28.6M / IOS 9.1M	Android 11.6M / IOS 21.5M
代码产物	JS Bundle 文件	二进制文件
维护者	Facebook	Google
风格	响应式，Learn once, write anywhere	响应式，一次编写多平台运行
支持	Android、IOS、(PC)	Android、IOS、(Web/PC)
使用代表	京东、携程、腾讯课堂	闲鱼、美团B端

一、环境搭建

无论是 **React Native** 还是 **Flutter** ，都需要 *Android* 和 *IOS* 的开发环境，也就是 *JDK* 、*Android SDK*、*Xcode* 等环境配置，而不同点在于 ：

- **React Native** 需要 `npm` 、 `node` 、 `react-native-cli` 等配置 。
- **Flutter** 需要 `flutter sdk` 和 *Android Studio* / *VSCode* 上的 **Dart** 与 **Flutter** 插件。

从配置环境上看， **Flutter** 的环境搭配相对简单，而 **React Native** 的环境配置相对复杂，而且由于 `node_module` 的“黑洞”属性和依赖复杂度等原因，目前在个人接触的例子中，**首次配置运行成功率 Flutter 是高于 React Native 的，且 Flutter 失败的原因则大多归咎于网络。**

> 同时跨平台开发首选 Mac ，没有为什么。

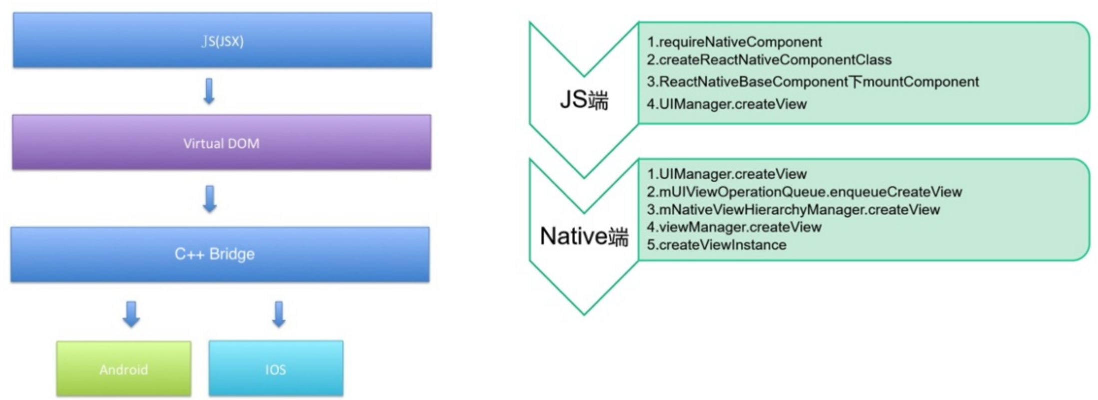
二、实现原理

在 *Android* 和 *IOS* 上，默认情况下 **Flutter** 和 **React Native** **都需要一个原生平台的 `Activity` / `ViewController` 支持，且在原生层面属于一个“单页面应用”，** 而它们之间最大的不同点其实在于 UI 构建：

- **React Native**：

React Native 是一套 UI 框架，默认情况下 **React Native** 会在 `Activity` 下加载 JS 文件，然后运行在 `JavaScriptCore` 中解析 *Bundle* 文件布局，最终堆叠出一系列的原生控件进行渲染。

简单来说就是 **通过写 JS 代码配置页面布局，然后 React Native 最终会解析渲染成原生控件**，如 `<View>` 标签对应 `ViewGroup/UIView` ， `<ScrollView>` 标签对应 `ScrollView/UIScrollView` ， `<Image>` 标签对应 `ImageView/UIImageView` 等。



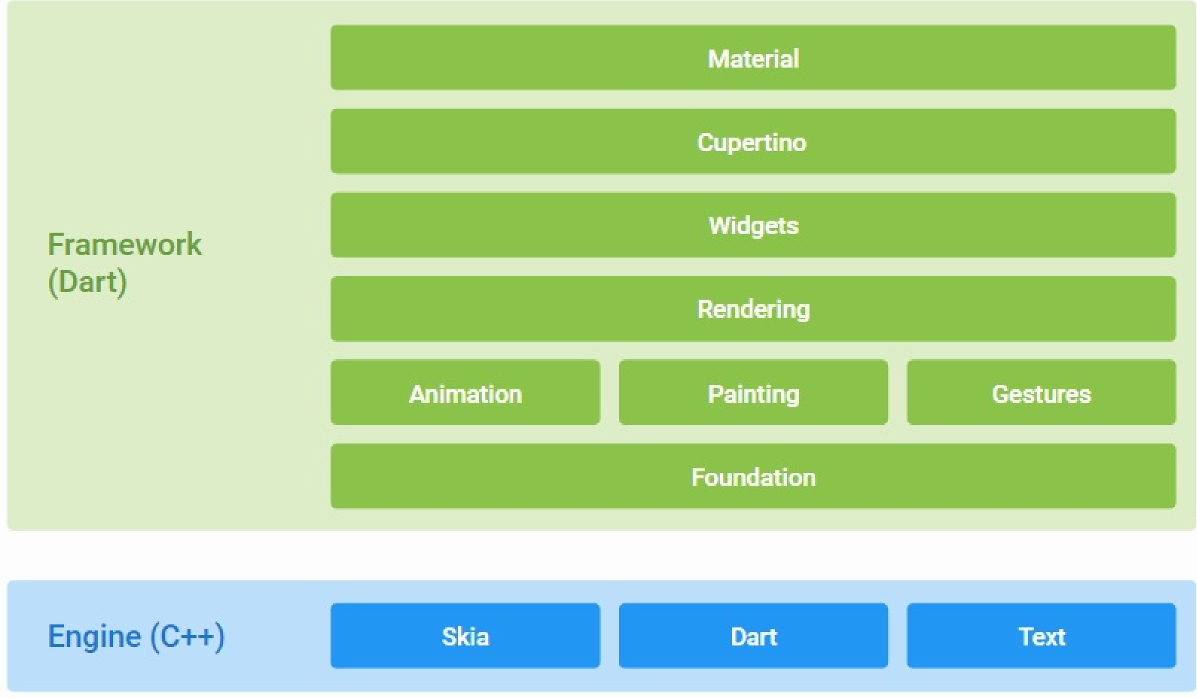
所以相较于如 `Ionic` 等框架而言， **React Native** 让页面的性能能得到进一步的提升。

- **Flutter**：

如果说 ***React Native*** 是为开发者做了平台兼容，那 ***Flutter*** 则更像是为开发者屏蔽平台的概念。

> **Flutter** 中只需平台提供一个 `Surface` 和一个 `Canvas` ，剩下的 **Flutter** 说：“你可以躺下了，我们来自己动”。

Flutter 中绝大部分的 `Widget` 都与平台无关， 开发者基于 `Framework` 开发 App ，而 `Framework` 运行在 `Engine` 之上，由 `Engine` 进行适配和跨平台支持。这个跨平台的支持过程，其实就是将 **Flutter UI 中的 `Widget` “数据化” ，然后通过 `Engine` 上的 `Skia` 直接绘制到屏幕上 。**



所以从以上可以看出：**React Native 的 *Learn once, write anywhere* 的思路，就是只要你会 *React* ，那么你可以用写 *React* 的方式，再去开发一个性能不错的App；而 Flutter 则是让你忘掉平台，专注于 Flutter UI 就好了。**

- **DOM：**

额外补充一点，React 的虚拟 *DOM* 的概念相信大家都知道，这是 React 的性能保证之一，而 Flutter 其实也存在类似的虚拟 *DOM* 概念。

> 看过我 **Flutter** 系列文章可能知道，**Flutter** 中我们写的 `Widget` ， 其实并非真正的渲染控件，这一点和 **React Native** 中的标签类似， `Widget` 更像配置文件， 由它组成的 `Widget` 树并非真正的渲染树。

`Widget` 在渲染时会经过 `Element` 变化， 最后转化为 `RenderObject` 再进行绘制， 而最终组成的 `RenderObject` 树才是 "真正的渲染 *Dom*" ， 每次 `Widget` 树触发的改变，并不一定会导致 `RenderObject` 树的完全更新。

所以在实现原理上 React Native 和 Flutter 是完全不同的思路，虽然都有类似"虚拟 *DOM* 的概念" ，但是React Native 带有较强的平台关联性，而 Flutter UI 的平台关联性十分薄弱。

三、 编程开发

React Native 使用的 *JavaScript* 相信大家都不陌生，已经 24 岁的它在多年的发展过程中，各端各平台中都出没着它的身影，在 Facebook 的 React 开始风靡之后，15 年移动浪潮下推出的 **React Native** ，让前端的 JS 开发者拥有了技能的拓展。

Flutter 的首选语言 *Dart* 语言诞生于 2011 年，而 2018 年才发布了 2.0，原本是为了用来对抗 *JavaScript* 而发布的开发语言，却在 *Web* 端一直不温不火，直到 17年 才因为 **Flutter** 而受关注起来，之后又因为 **Flutter For Web** 继续尝试后回归 *Web* 领域。

编程开发所涉及的点较多，后面主要从 `开发语言` 、`界面开发` 、`状态管理` 、`原生控件` 四个方面进行对比介绍。

> 至于最多吐槽之一就是为什么 **Flutter** 团队不选择 *JS* ，有说因为 *Dart* 团队就在 **Flutter** 团队隔壁，也有说谷歌不想和 **Oracle** 相关的东西沾上边。同时 **React Native** 更新快 4 年了，版本号依旧没有突破 1.0 。

3.1、 语言

因为起初都是为了 *Web* 而生，所以 *Dart* 和 *JS* 在一定程度上有很大的通识性。

如下代码所示， 它们都支持通过 `var` 定义变量，支持 `async/await` 语法糖，支持 `Promise` (`Future`) 等链式异步处理，甚至 `*` / `yield` 的语法糖都类似(虽然这个对比不大准确)，但可以看出它们确实存在“近亲关系” 。

```
/// JS

    var a = 1

    async function doSomeThing() {
        var result = await xxxx()
        doAsync().then((res) => {
            console.log("ffff")
        })
    }
    function* _loadUserInfo () {
        console.log("**********************");
        yield put(UpdateUserAction(res.data));
    }


/// Dart

  var a = 1;

  void doSomeThing() async {
    var result = await xxxx();
    doAsync().then((res) {
      print('ffff');
    });
  }
  _loadUserInfo() async* {
    print("**********************");
    yield UpdateUserAction(res.data);
  }
```

但是它们之间的差异性也很多，而最大的区别就是： **JS 是动态语言，而 Dart 是伪动态语言的强类型语言。**

如下代码中，在 `Dart` 中可以直接声明 `name` 为 `String` 类型，同时 `otherName` 虽然是通过 `var` 语法糖声明，但在赋值时其实会通过自推导出类型，而 `dynamic` 声明的才是真的动态变量，在运行时才检测类型。

```
// Dart

String name = 'dart';
var otherName = 'Dart';
dynamic dynamicName = 'dynamic Dart';
```

如下图代码最能体现这个差异，在下图例子中：

- `var i` 在全局中未声明类型时，会被指定为 `dymanic`，从而导致在 `init()` 方法中编译时不会判断类型，这和 JS 内的现象会一致。
- 如果将 `var i = "";` 定义在 `init()` 方法内，这时候 `i` 已经是强类型 `String` 了，所以编译器会在 `i++` 报错，**但是这个写法在 JS 动态语言里，默认编译时是不会报错的。**

如下图如果在初始化指定类型的，那么编译时就会告诉你错误了。

动态语言和非动态语言都有各种的优缺点，比如 JS 开发便捷度明显会高于 Dart，而 Dart 在类型安全和重构代码等方面又会比 JS 更稳健。

3.2、界面开发

React Native 在界面开发上延续了 *React* 的开发风格，**支持 scss/sass 、样式代码分离、在 0.59 版本开始支持 *React Hook* 函数式编程** 等等，而不同 *React* 之处就是更换标签名，并且样式和属性支持因为平台兼容做了删减。

如下图所示，是一个普通 **React Native** 组件常见实现方式，**继承 `Component` 类，通过 `props` 传递参数，然后在 `render` 方法中返回需要的布局，布局中每个控件通过 `style` 设置样式** 等等，这对于前端开发者基本上没有太大的学习成本。

```
import React, {Component} from 'react'
import {
    View, Text, TouchableOpacity
} from 'react-native';
import PropTypes from 'prop-types';
import styles from '../../style'
import * as Constant from '../../style/constant'
import UserImage from './UserImage'

class UserItem extends Component {
    constructor(props) {
        super(props)
    }

    render() {
        let {location, actionUser, actionUserPic} = this.props;
        return (
            <TouchableOpacity
                style={[{
                    padding: Constant.normalMarginEdge,
                    borderRadius: 4,
                }, styles.shadowCard]}
                onPress={() => {}}>
                <View style={styles.flexDirectionRowNotFlex}>
                    <UserImage uri={actionUserPic}/>
                    <View style={[styles.flex, styles.centerH, styles.flexDirectionRowNotFlex]}>
                        <Text style={[styles.flex, styles.smallText, {fontWeight: "bold"}]}>
                            {actionUser}
                        </Text>
                        <Text style={[styles.subSmallText, {marginTop: -20}]}>
                            {location}
                        </Text>
                    </View>
                </View>
            </TouchableOpacity>
        )
    }
}

const propTypes = {
    location: PropTypes.string,
    actionUser: PropTypes.string,
    actionUserPic: PropTypes.string,
    des: PropTypes.string,
};

UserItem.propTypes = propTypes;

export default UserItem
```

如下所示，如果再配合 *React Hooks* 的加持，函数式的开发无疑让整个代码结构更为简洁。

```
/**
 *
 * React Hooks 实现 reducer Demo
 *
 **/

const initialState = {count: 0};

function reducer(state, action) {
    switch (action.type) {
        case 'reset':
            return initialState;
        case 'increment':
            return {count: state.count + 1};
        case 'decrement':
            return {count: state.count - 1};
        default:
            // A reducer must always return a valid state.
            // Alternatively you can throw an error if an invalid action is dispatched.
            return state;
    }
}

export function DemoCounter({initialCount}) {
    const [state, dispatch] = useReducer(reducer, initialState: {count: initialCount});
    return (
        <View>
            <Text>Count: {state.count}</Text>
            <TouchableOpacity onPress={() => dispatch({type: 'reset'})}>
                <Text>Reset</Text>
            </TouchableOpacity>
            <TouchableOpacity onPress={() => dispatch({type: 'increment'})}>
                <Text>+</Text>
            </TouchableOpacity>
            <TouchableOpacity onPress={() => dispatch({type: 'decrement'})}>
                <Text>-</Text>
            </TouchableOpacity>
        </View>
    )
}
```

Flutter 最大的特点在于：**Flutter 是一套平台无关的 UI 框架，在 Flutter 宇宙中万物皆 `Widget` 。**

如下图所示，Flutter 开发中一般是通过继承 **无状态** `StatelessWidget` 控件或者 **有状态** `StatefulWidget` **控件** 来实现页面，然后在对应的 `Widget build(BuildContext context)` **方法内实现布局，利用不同** `Widget` **的** `child` **/** `children` **去做嵌套，通过控件的构造方法传递参数，最后对布局里的每个控件设置样式等。**

```
class UserItem extends StatelessWidget {
  final UserItemViewModel userItemViewModel;
  final VoidCallback onPressed;
  final bool needImage;

  UserItem(this.userItemViewModel, {this.onPressed, this.needImage = true});

  @override
  Widget build(BuildContext context) {
    return new Container(
      child: new GSYCardItem(
        child: new FlatButton(
          onPressed: onPressed,
          child: new Padding(
            padding: new EdgeInsets.only(
                left: 0.0, top: 5.0, right: 0.0, bottom: 10.0),
            child: new Row(
              children: <Widget>[
                new IconButton(
                    padding: EdgeInsets.only(
                        top: 0.0, left: 0.0, bottom: 0.0, right: 10.0),
                    icon: new ClipOval(
                      child: new FadeInImage.assetNetwork(
                        placeholder: GSYICons.DEFAULT_USER_ICON,
                        //预览图
                        fit: BoxFit.fitWidth,
                        image: userItemViewModel.userPic,
                        width: 30.0,
                        height: 30.0,
                      ),
                    ),
                    onPressed: null),
                new Expanded(
                    child: new Text(userItemViewModel.userName,
                        style: GSYConstant.smallTextBold)),
              ],
            ),
          ),
        ),
      ),
    );
  }
}
class UserItemViewModel {
  String userPic;
  String userName;
  UserItemViewModel.fromMap(User user) {
    userName = user.login;
    userPic = user.avatar_url;
  }
}
```

而对于 **Flutter** 控件开发，目前最多的吐槽就是 **控件嵌套和样式代码不分离** ，样式代码分离这个问题我就暂不评价，这个真要实际开发才能更有体会，而关于嵌套这里可以做一些 “洗白” ：

Flutter 中把一切皆为 `Widget` 贯彻得很彻底，**所以 `Widget` 的颗粒度控制得很细** ，如 `Padding` 、 `Center` 都会是一个单独的 `Widget` ，甚至**状态共享都是通过 `InheritedWidget` 共享 `Widget` 去实现的**，而这也是被吐槽的代码嵌套样式难看的原因。

事实上正是因为颗粒度细，所以你才可以通过不同的 `Widget` ，自由组合出多种业务模版， 比如 Flutter 中常用的 `Container` ，它就是官方帮你组合好的模板之一， **`Container` 内部其实是由 `Align` 、 `ConstrainedBox` 、 `DecoratedBox` 、 `Padding` 、 `Transform` 等控件组合而成** ，所以嵌套深度等问题完全是可以人为控制，甚至可以在帧率和绘制上做到更细致的控制。

当然，**官方也在不断地改进优化编写和可视化的体验**，如下图所示，从目前官方放出的消息上看，未来这个问题也会被进一步改善。

我们以一段 Flutter 代码为例，来讲解一下新的语法特性带来的变化:

```
Widget build(BuildContext context) {
  var items = [Text('目录')];
  // 把章节按分卷放到目录里
  for (var volume in volumes) {
    items.addAll(volume.chapters);
  }

  if (page != pages.last) items.add(Text('下一页'));

  items.add(Text('索引'));

  return Column(children: items);
}
```

上面这段代码用来显示一本电子书的目录。这个 UI 采用 Column 这个 widget 实现纵向布局。但是它有一些逻辑，需要用到循环和判断语句。这就导致了它需要把 Column 里面的内容先拼装到 items 这个变量里，然后再放到 Column 的 children 属性上。这里的问题是，最后的这个 Column 在概念上和视觉上都应该是先于它里面的内容出现的。但是由于语法的局限，这个空间关系被反了过来。代码看起来更像是命令式的而不是声明式的。

现在看看使用 Dart 2.3 新语法之后这段代码的写法:

```
Widget build(BuildContext context) =>
  Column(children: [
    Text('目录'),
    for (var volume in volumes)
      ...volume.chapters,
    if (page != pages.last) Text('下一页'),
    Text('索引'),
  ]);
```

首先，Column 可以按照我们直观的想法放在结构的最上层，然后“下一页”的逻辑也可以直接写到 List 的定义里面。每一个章节的名字可以用 for 元素和 spread operator 直接在 List 的定义里获得。改写之后的代码更精简，且更接近这个 UI 的直观描述。

Dart 1.x

```
// Dart 1.x
@override
Widget build(BuildContext context) {
  var items = [new Text('目录')];
  // 把章节按分卷放到目录里
  for (var volume in volumes) {
    items.addAll(volume.chapters);
  }

  if (page != pages.last) items.add(

  items.add(new Text('索引'));

  return new Scaffold(
    appBar: new AppBar(
      title: new Text(widget.title),
    ), // AppBar
    body: new Center(
      child: new Column(
        mainAxisAlignment: MainAxisA
        children: items,
      ), // Column
    ), // Center
  ); // Scaffold
}
```

Dart 2.0: Optional 'new'

```
//Dart 2.0
@override
Widget build(BuildContext context)
  var items = [Text('目录')];
  // 把章节按分卷放到目录里
  for (var volume in volumes) {
    items.addAll(volume.chapters);
  }

  if (page != pages.last) items.add

  items.add(Text('索引'));

  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ), // AppBar
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxis
        children: items,
      ), // Column
    ), // Center
  ); // Scaffold
}
```

Dart 2.3: UI as Code

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ), // AppBar
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlign
        children: [
          Text('目录'),
          for (var volume in volumes)
            ...volume.chapters,
          if (page != pages.last)
            Text('下一页'),
          Text('索引'),
        ],
      ), // Column
    ), // Center
  ); // Scaffold
}
```

Editor UI Guides

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAli
        children: [
          Text('目录'),
          for (var volume in volumes)
            ...volume.chapters,
          if (page != pages.last)
            Text('下一页'),
          Text('索引'),
        ],
      ),
    ),
  );
}
```

最后总结一下，抛开上面的开发风格，**React Native 在 UI 开发上最大的特点就是平台相关，而 Flutter 则是平台无关，比如下拉刷新，在 React Native 中， `<RefreshControl>` 会自带平台的不同下拉刷新效果，而在 Flutter 中，如果需要平台不同下拉刷新效果，那么你需要分别使用 `RefreshIndicator` 和 `CupertinoSliverRefreshControl` 做显示，不然多端都会呈现出一致的效果。**

3.3、状态管理

前面说过， **Flutter** 在很多方面都借鉴了 **React Native** ，所以在状态管理方面也极具“即视感”，比如**都是调用 `setState` 的方式去更新，同时操作都不是立即生效的**，当然它们也有着差异的地方，如下代码所示：

- 正常情况下 **React Native** 需要在 `Component` 内初始化一个 `this.state` 变量，然后通过 `this.state.name` 访问 。
- **Flutter** 继承 `StatefulWidget` ，然后在其的 `State` 对象内通过变量直接访问和 `setState` 触发更新。

```
/// JS

    this.state = {
       name: ""
    };

    ...

    this.setState({
        name: "loading"
    });

    ...

    <Text>this.state.name</Text>


/// Dart
```

```
var name = "";

setState(() {
   name =  "loading";
});

...

Text(name)
```

当然它们两者的内部实现也有着很大差异，比如 **React Native 受 React diff 等影响，而 Flutter 受 `isRepaintBoundary` 、 `markNeedsBuild` 等影响。**

而在第三方状态管理上，两者之间有着极高的相似度，如早期在 Flutter 平台就涌现了很多前端的状态管理框架如：flutter_redux 、fish_redux 、 dva_flutter 、 flutter_mobx 等等，它们的设计思路都极具 *React* 特色。

同时 **Flutter** 官方也提供了 scoped_model 、provider 等具备 **Flutter** 特色的状态管理。

所以在状态管理上 React Native 和 Flutter 是十分相近的，甚至是在跟着 React 走。

3.4、原生控件

在跨平台开发中，就不得不说到接入原有平台的支持，比如 在 *Android* 平台上接入 *x5* 浏览器 、接入视频播放框架、接入 *Lottie* 动画框架等等。

这一需求 **React Native** 先天就支持，甚至在社区就已经提供了类似 lottie-react-native 的项目。 **因为 React Native 整个渲染过程都在原生层中完成，所以接入原有平台控件并不会是难事** ，同时因为发展多年，虽然各类第三方库质量参差不齐，但是数量上的优势还是很明显的。

而 **Flutter** 在就明显趋于弱势，甚至官方在开始的时候，连 `WebView` 都不支持，这其实涉及到 **Flutter** 的实现原理问题。

因为 **Flutter 的整体渲染脱离了原生层面，直接和 GPU 交互，导致了原生的控件无法直接插入其中** ，而在视频播放实现上， **Flutter** 提供了外界纹理的设计去实现，但是这个过程需要的数据转换，很明显的限制了它的通用性， **所以在后续版本中** Flutter **提供了 `PlatformView` 的模式来实现集成。**

> 以 *Android* 为例子，在原生层 **Flutter** 通过 `Presentation` 副屏显示的原理，利用 `VirtualDisplay` 的方式，让 *Android* 控件在内存中绘制到 `Surface` 层。 `VirtualDisplay` 绘制在 `Surface` 的 **textureId** ，之后会通知到 *Dart* 层，在 *Dart* 层利用 `AndroidView` 定义好的 `Widget` 并带上 **textureId** ，那么 **Engine** 在渲染时，就会在内存中将 **textureId** 对应的数据渲染到 `AndroidView` 上。

`PlatformView` 的设计必定导致了性能上的缺陷，最大的体现就是内存占用的上涨，同时也引导了诸如键盘无法弹出#19718和黑屏等问题，甚至于在 *Android* 上的性能还可能不如外界纹理。

所以目前为止， Flutter 原生控件的接入上是仍不如 React Native 稳定。

四、 插件开发

React Native 和 **Flutter** 都是支持插件开发，不同在于 **React Native 开发的是 npm 插件，而 Flutter 开发的是 pub 插件。**

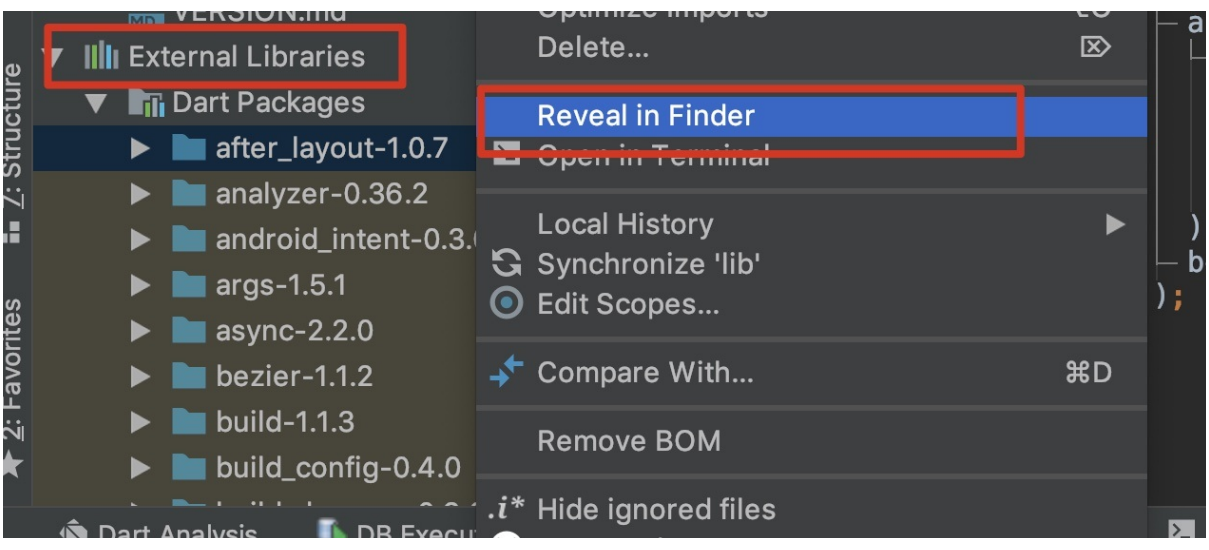
React Native 使用 *npm* 插件的好处就是：可以使用丰富的 *npm* 插件生态，同时减少前端开发者的学习成本。

但是使用 *npm* 的问题就是太容易躺坑，**因为 *npm* 包依赖的复杂度和深度所惑，以至于你都可能不知道 *npm* 究竟装了什么东西**，抛开安全问题，这里最直观的感受就是："为什么别人跑得起来，而我的跑不起来？"同时每个项目都独立一个 **node_module**，对于硬盘空间较小的 Mac 用户略显心酸。

Flutter 的 *pub* 插件默认统一管理在 *pub* 上，类似于 *npm* 同样支持 *git* 链接安装，而 `flutter packages get` 文件一般保存在电脑的统一位置，多个项目都引用着同一份插件。

> - win 一般是在 C:\Users\xxxxx\AppData\Roaming\Pub\Cache 路径下
> - mac 目录在 ~/.pub-cache

如果找不到插件目录，也可以通过查看 `.flutter-plugins` 文件，或如下图方式打开插件目录，至于为什么需要打开这个目录，感兴趣的可以看看这个问题 13# 。



最后说一下 **Flutter** 和 **React Native** 插件，在带有原生代码时不同的处理方法：

- **React Native** 在安装完带有原生代码的插件后，需要执行 `react-native link` 脚本去引入支持，具体如 *Android* 会在 `setting.gradle` 、 `build.gradle` 、 `MainApplication.java` 等地方进行侵入性修改而达到引用。
- **Flutter** 则是通过 `.flutter-plugins` 文件，保存了带有原生代码的插件 *key-value* 路径，之后 **Flutter** 的脚本会通过读取的方式，动态将原生代码引入，最后通过生成 `GeneratedPluginRegistrant.java` 这个忽略文件完成导入，这个过程开发者基本是无感的。

所以在插件这一块的体验， Flutter 是略微优于 React Native 的。

五、 编译和产物

React Native 编译后的文件主要是 `bundle` 文件，在 *Android* 中是 `index.android.bunlde` 文件，而在 *IOS* 下是 `main.jsbundle` 。

Flutter 编译后的产物在 *Android* 主要是：

- `isolate_snapshot_instr` 应用程序指令段
- `isolate_snapshot_data` 应用程序数据段
- `vm_snapshot_data` 虚拟机数据段
- `vm_snapshot_instr` 虚拟机指令段等产物

> ⚠注意，1.7.8 之后的版本，Android 下的 Flutter 已经编译为纯 so 文件。

在 IOS 主要是 **App.framework** ，其内部也包含了 `kDartVmSnapshotData` 、 `kDartVmSnapshotInstructions` 、 `kDartIsolateSnapshotData` 、 `kDartIsolateSnapshotInstructions` 四个部分。

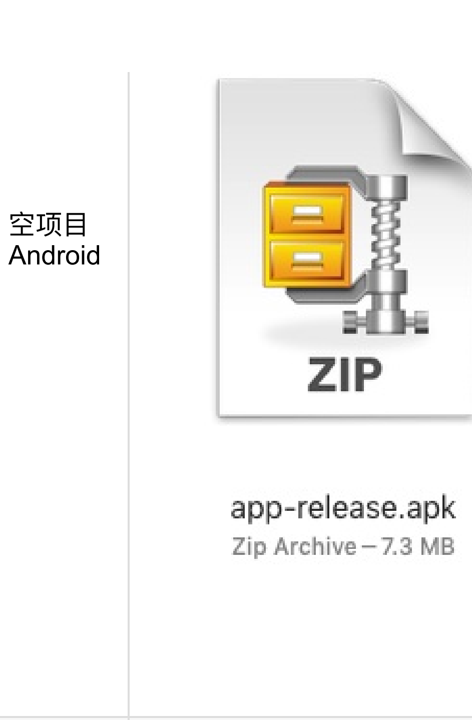
接着看完整结果，如下图所示，是空项目下 和 GSY 实际项目下， **React Native** 和 **Flutter** 的 Release 包大小对比。

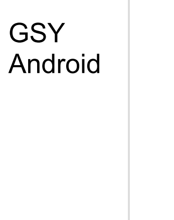
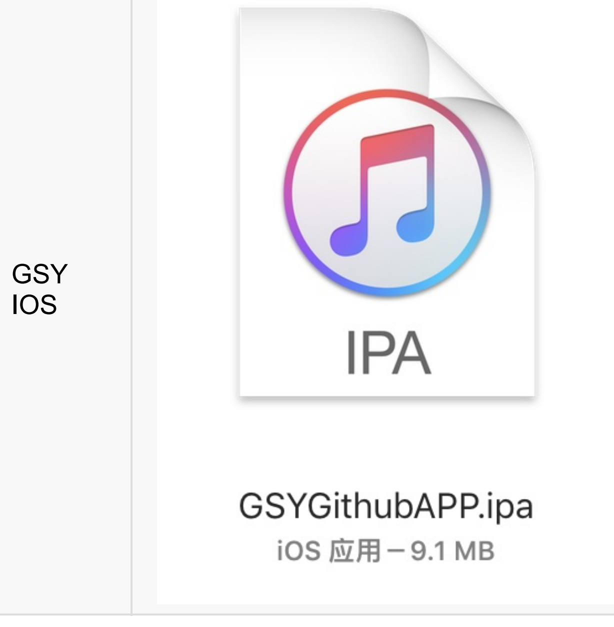
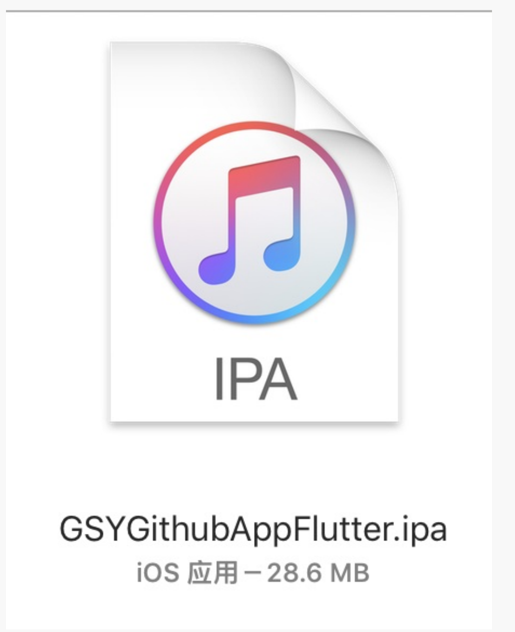
可以看出在 **React Native 同等条件下， Android 比 IOS 大很多** ，这是因为 *IOS* 自带了 **JSCore** ，而 *Android* 需要各类动态 **so** 内置支持，而且这里 *Android* 的动态库 **so** 是经过了 `ndk` 过滤后的大小，不然还会更大。

Flutter 和 **React Native** 则是相反，因为 *Android* 自带了 **skia** ， 所以比没有自带 **skia** 的 *IOS* 会小得多。

以上的特点在 GSY 项目中的 Release 包也呈同样状态。

类型	React Native	Flutter

空项目 Android	 	 app-release.apk Zip Archive – 5.2 MB
空项目 IOS	IPA test999.ipa iOS 应用 – 1.6 MB 标签 添加标签... 创建时间 今天 14:55 修改时间 今天 14:55	IPA Runner.ipa iOS 应用 – 10.1 MB

GSY Android 	 GSYGIthubApp.apk Zip Archive – 21.5 MB	 GSYGithubAppFlutter.apk Zip Archive – 11.6 MB
GSY IOS		

值得注意的是，Google Play 最近发布了 《8月不支持 64 位，App 将无法上架 Google Play! 》 的通知，同时也表示将停止 *Android Studio* 32 位的维护，而 `arm64-v8a` 格式的支持，**React Native** 需要在 0.59 以后的版本才支持。

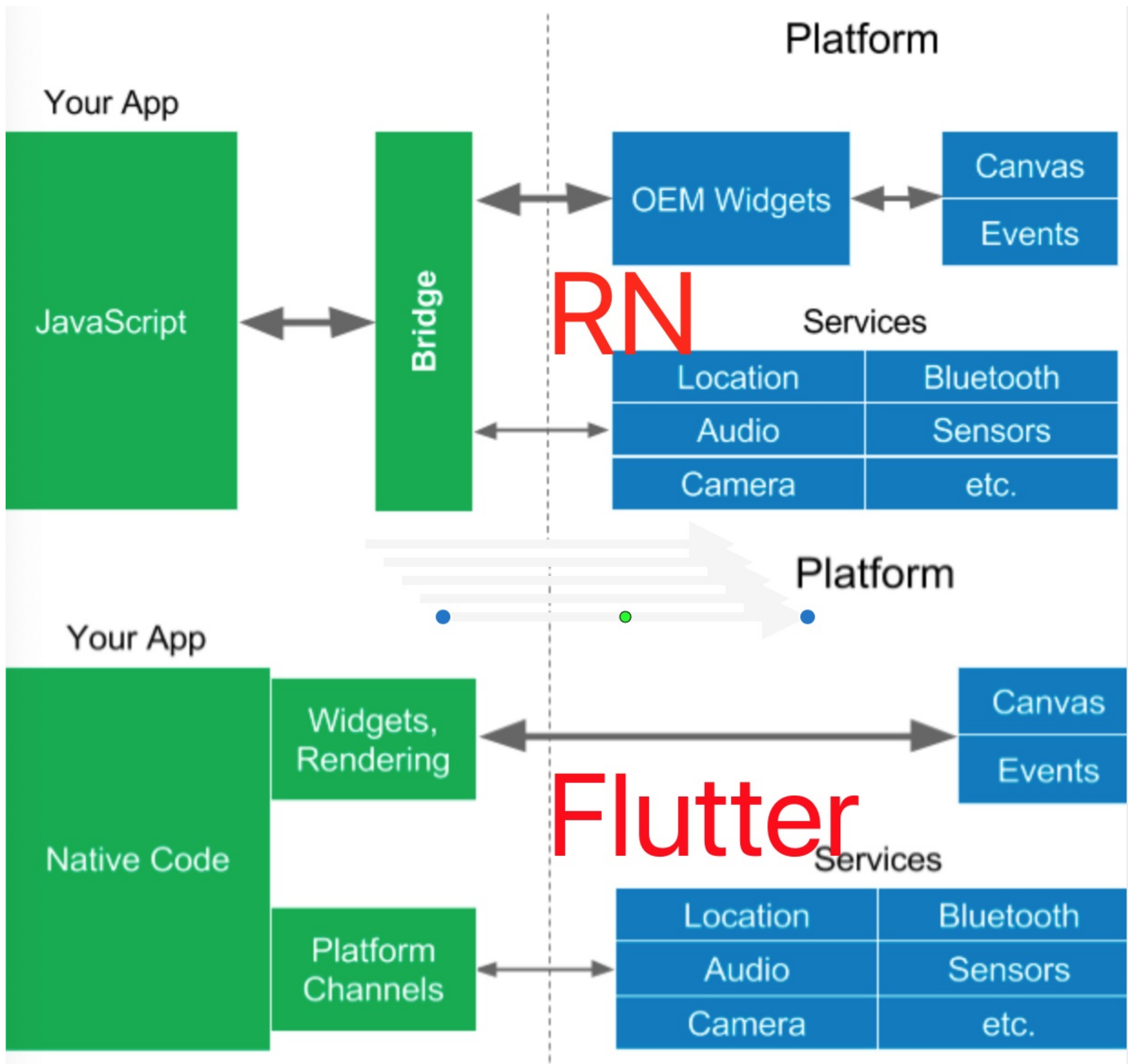
至于 **Flutter**，在打包时通过指定 `flutter build apk --release --target-platform android-arm64` 即可。

六、性能

说到性能，这是一个大家都比较关心的概念，但是有一点需要注意，**抛开场景说性能显然是不合适的，因为性能和代码质量与复杂度是有一定联系的。**

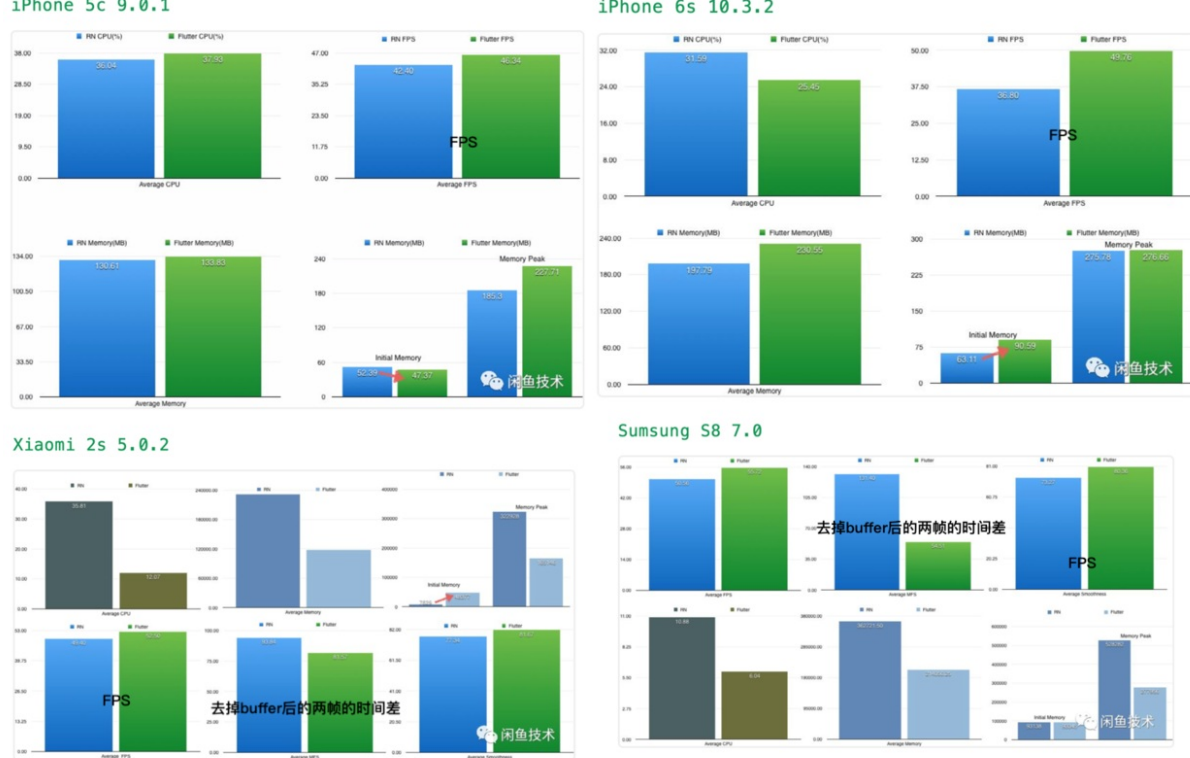
先说理论性能，**在理论上 Flutter 的设计性能是强于 React Native**，这是框架设计的理念导致的，Flutter 在少了 **OEM Widget**，直接与 CPU / GPU 交互的特性，决定了它先天性能的优势。

> 这里注意不要用模拟器测试性能，特别是IOS模拟器做性能测试，因为 Flutter 在 IOS模拟器中纯 CPU ， 而实际设备会是 GPU 硬件加速，同时只在 Release 下对比性能。



> 代码的实现方式不同，也可能会导致性能的损失，比如 **Flutter** 中 **skia** 在绘制时， `saveLayer` 是比较消耗性能的，比如 透明合成、*clipRRect* 等等，都会可能需要 `saveLayer` 的调用， 而 `saveLayer` 会清空GPU绘制的缓存，导致性能上的损耗，从而导致开发过程中如果掉帧严重。

最后如下图所示，是去年闲鱼用 GSY 项目做测试对比的数据，原文在《流言终结者- Flutter和RN谁才是更好的跨端开发方案？》 ，可以看出在去年的时候， **Flutter的整体帧率和绘制就有了明显的优势。**



> 额外补充一点，*JS* 和 *Dart* 都是单线程应用，利用了协程的概念实现异步效果，而在 **Flutter** 中 *Dart* 支持的 `isolate` ，却是属于完完全全的异步线程处理，可以通过 Port 快捷地进行异步交互，这大大拓展了 **Flutter** 在 *Dart* 层面的性能优势。

七、发展未来

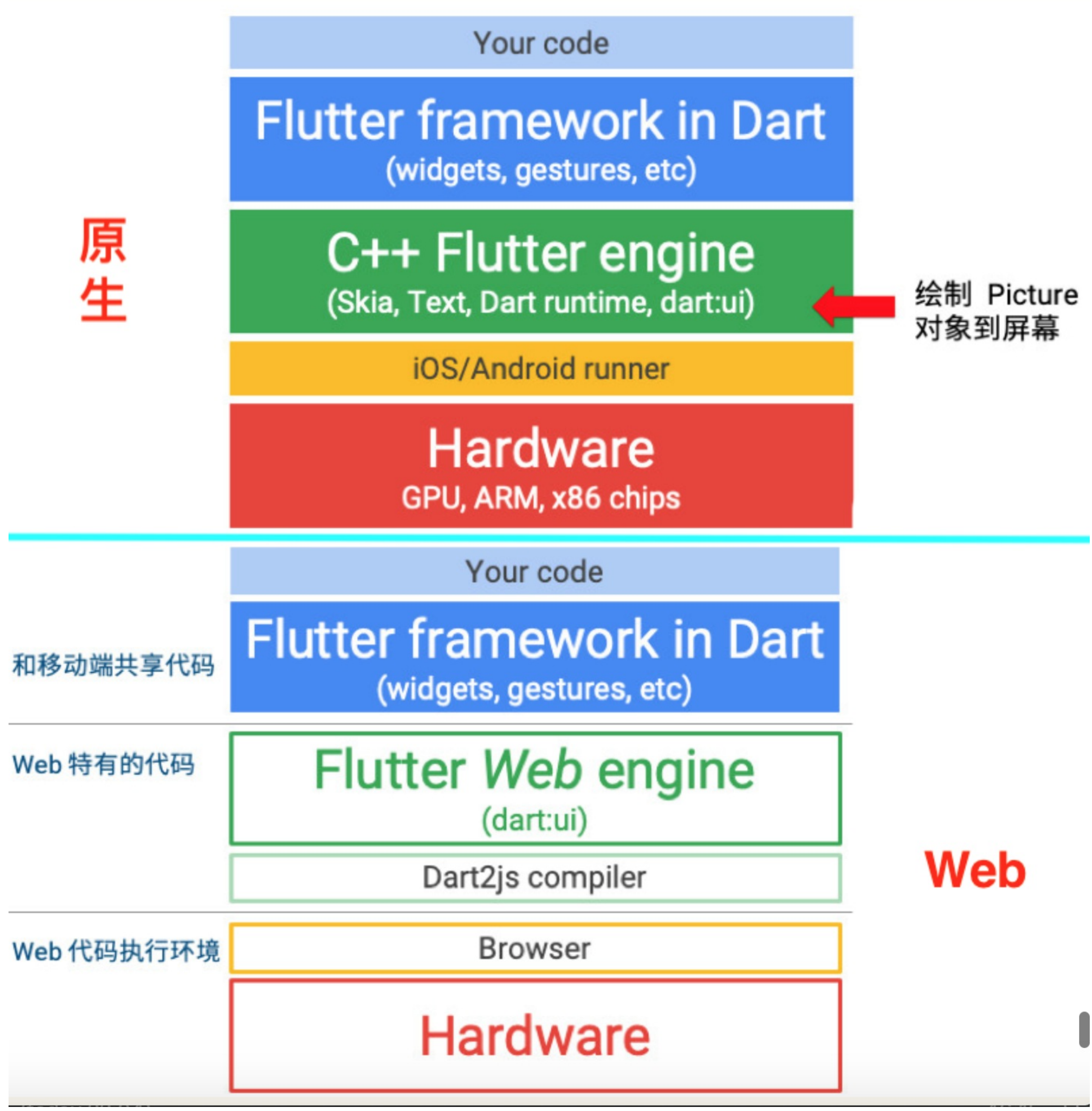
之前一篇 《为什么 Airbnb 放弃了 React Native?》 文章，让众多不明所以的吃瓜群众以为 **React Native** 已经被放弃，之后官方发布的 《Facebook 正在重构 React Native，将重写大量底层》 公示，又再一次稳定了军心。

同时 **React Native** 在 0.59 版本开始支持 *React Hook* 等特性，并将原本平台的特性控件从 **React Native** 内部剥离到社区，这样控件的单独升级维护可以更加便捷，同时让 **React Native** 与 **React** 之间的界限越发模糊。

Flutter UI 平台的无关能力，让 **Flutter** 在跨平台的拓展上更为迅速，尽管 **React Native** 也有 *Web* 和 *PC* 等第三方实现拓展支持，但是由于平台关联性太强，这些年发展较为缓慢， 而 **Flutter** 则是短短时间又宣布 *Web* 支持，甚至拓展到 *PC* 和嵌入式设备当中。

这里面对于 **Flutter For Web** 相信是大家最为关心的话题， 如下图所示，在 **Flutter** 的设计逻辑下，开发 **Flutter Web** 的过程中，你甚至感知不出来你在开发的是 Web 应用。

Flutter Web 保留了 大量原本已有的移动端逻辑，只是在 Engine 层利用 Dart2Js 的能力实现了差异化， 不过现阶段而言，Flutter Web 仍处在技术预览阶段，不建议在生产环境中使用 。



由此可以推测，不管是 Flutter 或者 React Native，都会努力将自己拓展到更多的平台，同时在自己的领域内进一步简化开发。

- 其他参考资料：

《Facebook 正在重构 React Native，将重写大量底层》

《React Native 的未来与React Hooks》

《庖丁解牛！深入剖析 React Native 下一代架构重构》

《Flutter 最新进展与未来展望》

> 自此，本文终于结束了，长呼一口气。

资源推荐

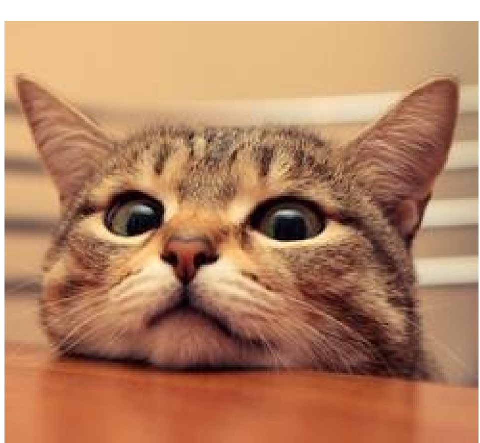
- Github： https://github.com/CarGuo/
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**

- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp

文章

《Flutter完整开发实战详解系列》

《移动端跨平台开发的深度解析》



大家好，我是郭树煜，Github GSY 系列开源项目的作者，系列包括有 GSYVideoPlayer 、 GSYGitGithubApp(Flutter\ReactNative\Kotlin\Weex)四大版本，目前总 star 在 17 k+ 左右，主要活跃在掘金社区，id 是恋猫的小郭，主要专栏有《Flutter完整开发实战详解》系列等，平时工作负责移动端项目的开发，工作经历从 Android 到 React Native 、Weex 再到如今的 Flutter ，期间也参与过 React 、 Vue 、小程序等相关的开发，算是一个大前端的选手吧。

> 这次主要是给大家分享 Flutter 相关的内容，主要涉及做一些实战和科普性质的内容。

一、移动开发的现状

恰逢最近谷歌 IO 大会结束，大会后也在线上线下和大家有过交流，总结了下大家最关系的问题有：

1、谷歌在 Kotlin-First 的口号下又推广 Dart + Flutter 冲突吗？

这个问题算是被问得最多的一个，先说观点：我个人认为其实这并不冲突，因为有个 **误区就是认为跨平台开发就可以抛弃原生开发！**

如果从事过跨平台开发的同学应该知道，平台提供的功能向来是有限的，而面对产品经理的各种 “点歪技能树” 的需求，很多时候你是需要基于框架外提供支持，常见的就是 **混合开发或者原生插件支持** 。

所以这里我表达的是，目前 **`Kotlin` 和 `Dart` 更多是相辅相成** ，而一旦业务复杂度到一定程度，跨平台框架还可能存在降低工作效率的问题，比如针对新需求，需要重复开发 `Android/IOS` 的原生插件做支持，这也是 Aribnb 曾经选择放弃 `React Native` 的原因之一。

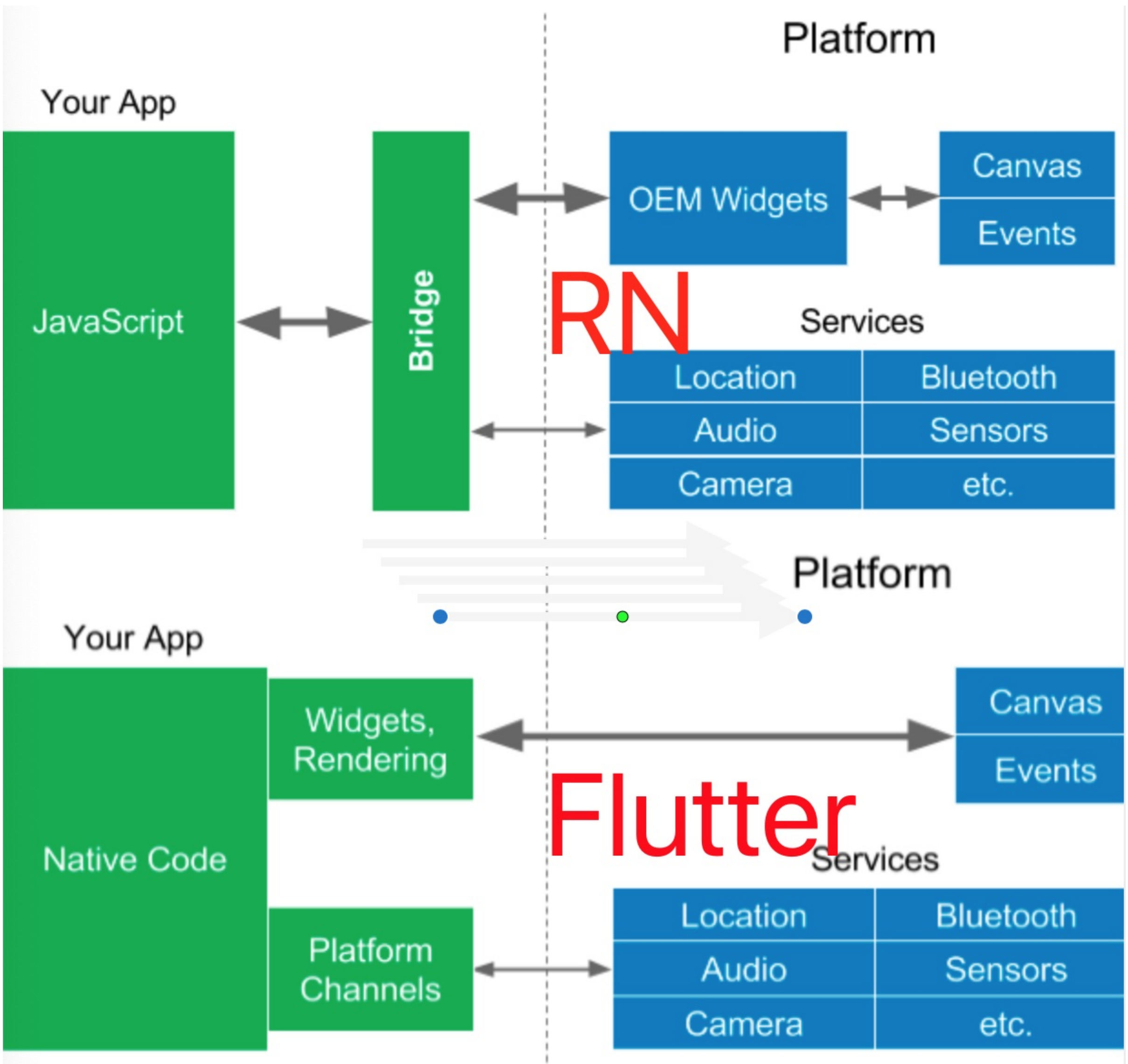
与我而言，**跨平台的意义在于解决的是端逻辑的统一** ，至少避免了逻辑重复实现，或者 `IOS` 和 `Android` 之间争论 谁对谁错 的问题，甚至可以统一到 web 端等等。

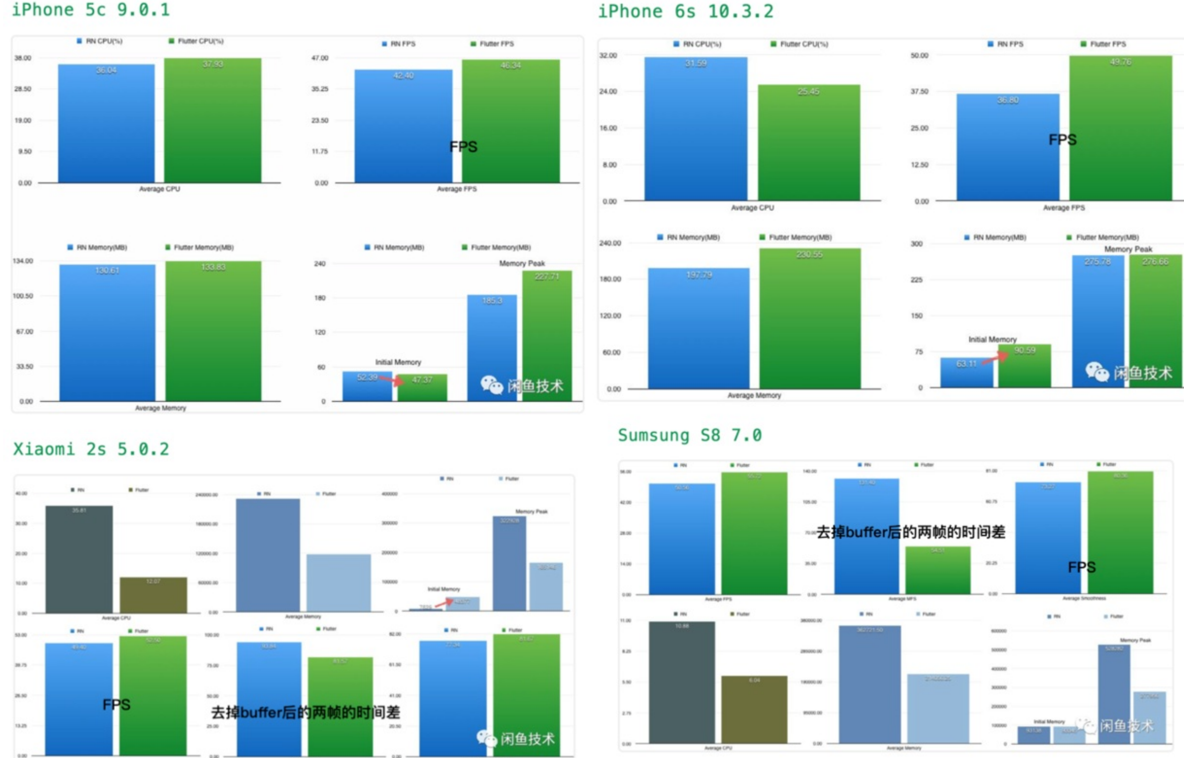
2、React Native 和 Flutter 之间的对比

`Flutter` 作为后来者，难免会被用来和 `React Native` 进行对比，在这个万物皆是 `JS` 的时代， `Dart` 和 `Flutter` 的出现显得尤为扎眼。

在设计上它们有着许多相似之处，响应式设计/*async*支持/*setState*更新 等等，同时也有着各种的差异，而大家最为关心的，无非 **性能、支持、上手难易、稳定性程度** 这四方面：

- **性能上 Flutter 的确实会比 React Native 好**，如下图所示，这是由框架底层决定的，当然目前 `React Native` 也在进行下一代的优化， 而对此最直观的数据就是：**GSY系列 在18年用于闲鱼测试下的对比数据了** 。


Platform
Your App
JavaScript
Bridge
OEM Widgets
Canvas
Events
RN
Services
Location
Bluetooth
Audio
Sensors
Camera
etc.
Platform
Your App
Native Code
Widgets, Rendering
Platform Channels
Canvas
Events
Flutter
Services
Location
Bluetooth
Audio
Sensors
Camera
etc.



> 同时注意不要用模拟器测试性能，特别是IOS模拟器做性能测试，因为 Flutter 在 IOS模拟器中纯 CPU ， 而实际设备会是 GPU 硬件加速，同时只在 Release 下对比性能。

- 支持上 Flutter 和 React Native ， 都存在第三方包质量参差不齐的问题，而目前在这一块 **Flutter 是弱于 React Native 的**，毕竟 `React Native` 发展已久，虽然版本号一直不到 1.0，但是在 `JS` 的加持下生态丰富，同时也是因为平台特性的原因，诸如 WebView 、地图等控件的支持上现在依旧不够好，这个后面也会说道。
- 上手难易度上， **`Flutter` 配置环境和运行的“成功率”比 React Native 高不少**， 这里面有 `node_module` 黑洞这个坑，也有 `React Native` 本身依赖平台控件导致的，至少我曾经试过接手一个 `React Native` 跑了一天都没跑起来的经历，同时 `Flutter` 在运行和SDK版本升级的阵痛也会少很多。
- 稳定性： **`Flutter` 中大部分异常是不会引起应用崩溃**， 更多会在 Debug 上体现为红色错误堆栈，Release 上 UI 异常等等。

> **如果你是前端，我会推荐你先学 `React Native` ，如果你是原生开发，我推荐你学 `Flutter`**
> 。
>
> 在 React Native 0.59.x 版本开始，React 已经将许多内置控件和库移出主项目，希望模糊 React 和 React Native 的界线，统一开发，这里的理念和 Flutter 很像。
>
> Flutter 暂时不支持热更新！！！！！！！

二、Flutter 实战

1、Dart 中有意思的一些东西

1.1、var 的语法糖和 dynamic

`var` 的语法糖是在赋值时才**自推导出类型的**，而 `dynamic` 是动态声明，在运行时检测，它们的使用有时候容易出现错误。

如下图所示以说，

- `var` 初始化时被指定为 `dynamic` 类型的。
- 然后赋值的时候初始化为 `String` 类型，这时候进行 ++ 操作就会出现运行时报错，
- 如下图2如果在初始化指定类型的，那么编译时就会告诉你错误了。

```
var i;    dymanic

init() {
  i = "";    String
  i++;
  print(i);
}
```

```
More Actions
══╡ EXCEPTION CAUGHT BY WIDGETS LIBRARY ╞══════
The following assertion was thrown building NotificationListe
type 'int' is not a subtype of type 'String' of 'other'
```

```
91   init() {
92     var i = "";
93     i++;
94     print(i);
The argument type 'int' can't be assigned to the parameter type 'String'.
```

1.2、各类操作符

如下图所示，`Dart` 支持很多有意思的操作符，如下图：

- 执行的时候首先是判断 `AA` 如果为空，就返回 `999` ；
- 之后如果 `AA` 为空，就为 `AA` 赋值 `999` ；
- 之后对 `AA` 进行整除 `999` ，输出结果 `10` 。

```
var AA;
init() {
  AA ?? 999;
  AA ??= 999;
  var result = AA ~/ 99;
  print(result);
  ///输出10
}
```

1.3、支持操作符重载

如下图所示， `Dart` 中是支持操作符重载的，这样可以比较直观我们的代码逻辑，并且简化代码时的调用。

```
class Vector {
  final int x, y;

  Vector(this.x, this.y);

  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

}

init() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);
  ///输出4
  print((v + w).x);
  ///输出1
  print((v - w).y);
}
```

1.4、方法当做参数传递

如下图所示，在 `Dart` 中方法时可以作为参数传递的，这样的形式可以让我们更灵活的组织代码的逻辑。

```
main() {
  doWhat(String name) {
    print(name);
    return "this $name";
  }
  doNext(int data) {
    print(data);
  }
  doSomeThing(doWhat, doNext);
}

doSomeThing(String doWhat(String name), void doNext(int data)) {
  var result = doWhat("guo");
  print(result);
  doNext(10);
}
```

1.5、async await / async* yield

在 `Dart` 中 `async await / async* yield` 等语法糖，代表 `Dart` 中的 `Future` 和 `Stream` 操作，它们对应 `Dart` 中的异步逻辑支持。

> sync* / yield 对应 `Stream` 的同步操作。

1.6、Mixins

在 `Dart` 中支持混入的模式，如下图所示，混入时的基础顺序是从右到左依次执行的，而且和 `super` 有关，同时 `Dart` 还支持 `mixin` 关键字的定义。

```
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()
}
```

> Flutter 的启动类用的就是 mixins 方式

1.7、isolate

`Dart` 中单线程模式中增加了 `isolate` 提供跨线程的真异步操作，而因为 `Dart` 中线程不会共享内存，所以也不存在死锁，从而也导致了 `isolate` 之间数据只能通过 `port` 的端口方式发送接口，类似于 `Scoket` 的方式，同时提供了 `compute` 的封装接口方便调用。

1.8 call

Dart 为了让类可以像函数一样调用，默认都可以实现 `call()` 方法，同样 `typedef` 定义的方法也是具备 `call()` 条件。

比如我定义了一个 `CallObject`

```
class CallObject {

  List<Widget> footerButton = [];

  call(int i, double e) => "$i xxxx $e";
}
```

就可以通过以下执行

```
CallObject callObject = CallObject();
print(callObject(11, 11.0));
print(callObject?.call(11, 11.0));
```

然后我定义了

```
typedef void ValueFunction(int i);

  ValueFunction vt = (int i){
    print("zzz $i");
  };
```

就可以通过直接执行和判空执行处理

```
 vt(666);
 vt?.call(777);
```

2、Flutter 中常见的

2.1、ChangeNotifier

如下图所示，`ChangeNotifier` 模式在 `Flutter` 中是十分常见的，比如 `TextField` 控件中，通过 `TextEditingController` 可以快速设置值的显示，这是为什么呢?

```
class TestInput extends StatefulWidget {
  @override
  _TestInputState createState() => _TestInputState();
}

class _TestInputState extends State<TestInput> {

  final TextEditingController controller = new TextEditingController(

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        alignment: Alignment.center,
        child: new TextField(
          textAlign: TextAlign.center,
          controller: controller,
        ),
      ),
      floatingActionButton: FloatingActionButton(onPressed: (){
        controller.text = "这是一段按键输入！！\n"
            "输入了哦";
      }),
    );
  }
}
```

如下图所示，这是因为 `TextEditingController` 它是 `ChangeNotifier` 的子类，而 `TextField` 的内部对其进行了 `addListener` ，同时我们改变值的时候调用了 `notifyListener` ，触发内部 `setState` 。

```
@override
void initState() {
  super.initState();
  widget.controller.addListener(_didChangeTextEditingValue);
  widget.focusNode.addListener(_handleFocusChanged);
  _scrollController.addListener(() { _selectionOverlay?.updateForScroll(); });
  _cursorBlinkOpacityController = AnimationController(vsync: this, duration: _fadeDuration);
  _cursorBlinkOpacityController.addListener(_onCursorColorTick);
  _floatingCursorResetController = AnimationController(vsync: this);
  _floatingCursorResetController.addListener(_onFloatingCursorResetTick);
}

void _didChangeTextEditingValue() {
  _updateRemoteEditingValueIfNeeded();
  _startOrStopCursorTimerIfNeeded();
  _updateOrDisposeSelectionOverlayIfNeeded();
  _textChangedSinceLastCaretUpdate = true;
  setState(() { /* We use widget.controller.value in build(). */ });
}
```

2.2、InheritedWidget

在 `Flutter` 中所有的状态共享都是通过它实现的，如自带的 `Theme` ， `Localizations` ，或者状态管理的 `scoope_model` 、 `flutter_redux` 等等，都是基于它实现的。

如下图是 `SliderTheme` 的自定义实现逻辑，默认 `Theme` 中是包含了 `SliderTheme` ，但是我们可以通过覆盖一个新的 `SliderTheme` 嵌套去实现自定义，然后通过 `SliderTheme theme = SliderTheme(context);` 获取，其中而 `context` 的实现就是 `Element` 。

```dart
@override
Widget build(BuildContext context) {
  return SliderTheme(
    data: SliderThemeData.fromPrimaryColors(
      primaryColor: Colors.amber,
      primaryColorDark: Colors.yellow,
      primaryColorLight: Colors.amberAccent,
      valueIndicatorTextStyle: TextStyle(
        fontSize: 14,
        color: Colors.red,
      ),
    ),
    child: Slider(
      value: _value,
      max: 100.0,
      min: 0.0,
      divisions: 100,
      label: '$_value value',
      onChanged: (data) {
        var color = SliderTheme.of(context).thumbColor;
        setState(() {
          print(data);
          _value = data;
        });
      }),
```

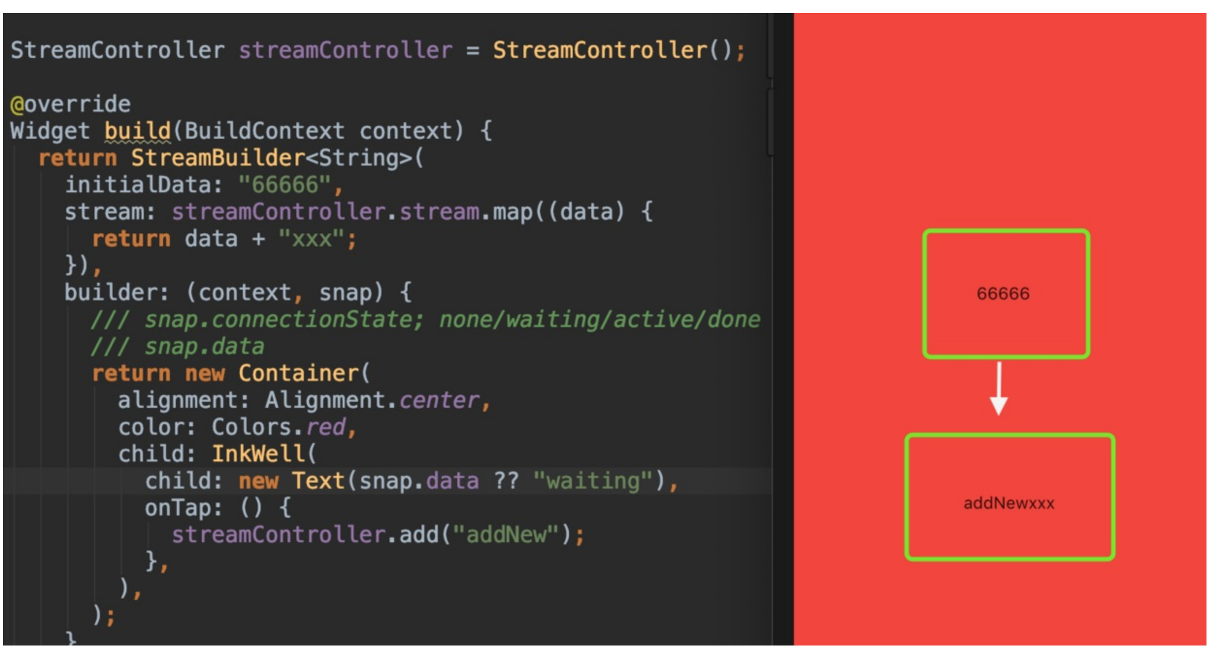
41.0 value

在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement> _inheritedWidgets` 的对象。

`_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并 。所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 时，就可以往上查找到父控件的 `Widget` 。

2.3、StreamBuilder

`StreamBuilder` 一般用于通过 `Stream` 异步构建页面的，如下图所示，通过点击之后，绿色方框的文字会变成 `addNewxxx` ，因为 `Stream` 进行了 `map` 变化，同时一般实现 `bloc` 模式的时候，经常会用到它们。



> 类似的还有 FutureBuilder

2.4、State 中的参数使用

一般 `Widget` 都是一帧的，而 `State` 实现了 `Widget` 的跨帧绘制，一般定义的时候，我们可以如下图一样实现，而如下图尖头所示，这时候我们点击 `setState` 改变的时候，是不会出现效果的，为什么呢?

由 Xnip 截图

```dart
class DemoApp extends StatefulWidget {
  @override
  _DemoAppState createState() => _DemoAppState();
}

class _DemoAppState extends State<DemoApp> {
  String data = "init";

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
        home: Scaffold(
      body: Scaffold(
        body: DemoPage("Test", data, 30),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            data = "setState";
          });
        },
      ),
    ));
  }
}

class DemoPage extends StatefulWidget {
  final String title;
  final String data;
  final int count;

  DemoPage(this.title, this.data, this.count);

  @override
  _DemoPageState createState() => _DemoPageState(this.data);
}

class _DemoPageState extends State<DemoPage> {

  final String data;

  _DemoPageState(this.data);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new ListView.builder(
        itemBuilder: (context, index) {
          /// widget.data
          return new Text(data);
        },
        itemCount: widget.count,
      ),
    );
  }
}
```

其实 State 对象的创建和更新时机导致的：

- 1、createState 只在 StatefulElement 创建时才会被创建的。
- 2、StatefulElement 的 createElement 一般只在 inflateWidget 调用。
- 3、updateChild 执行 inflateWidget 时， 如果 child 存在可以更新的话，不会执行 inflateWidget。

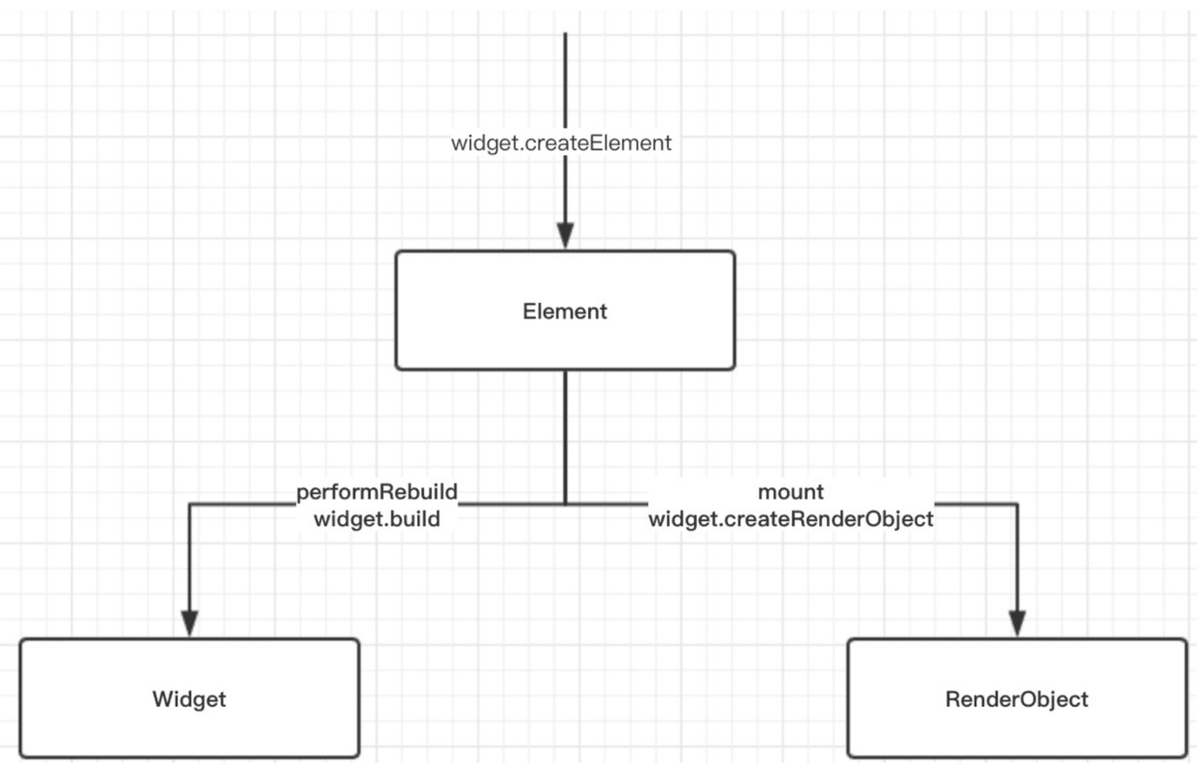
```
/// An [Element] that uses a [StatefulWidget] as its configuration.
class StatefulElement extends ComponentElement {
  /// Creates an element that uses the given widget as its configuration.
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        super(widget) {
    _state._element = this;
    _state._widget = widget;

  }

  /// 1、createState 只在 StatefulElement 创建时才会被创建的。
  /// 2、StatefulElement 的 createElement 一般只在 inflateWidget 调用。
  /// 3、updateChild 执行 inflateWidget 时， 如果 child 存在可以更新的话，不会执行 inflateWidget。
  @override
  void update(StatefulWidget newWidget) {
    super.update(newWidget);
    assert(widget == newWidget);
    final StatefulWidget oldWidget = _state._widget;
    _dirty = true;
    _state._widget = widget;
    rebuild();
  }
```

3、四棵树

Flutter 中主要有 `Widget` 、 `Element` 、 `RenderObject` 、 `Layer` 四棵树，它们的作用是：

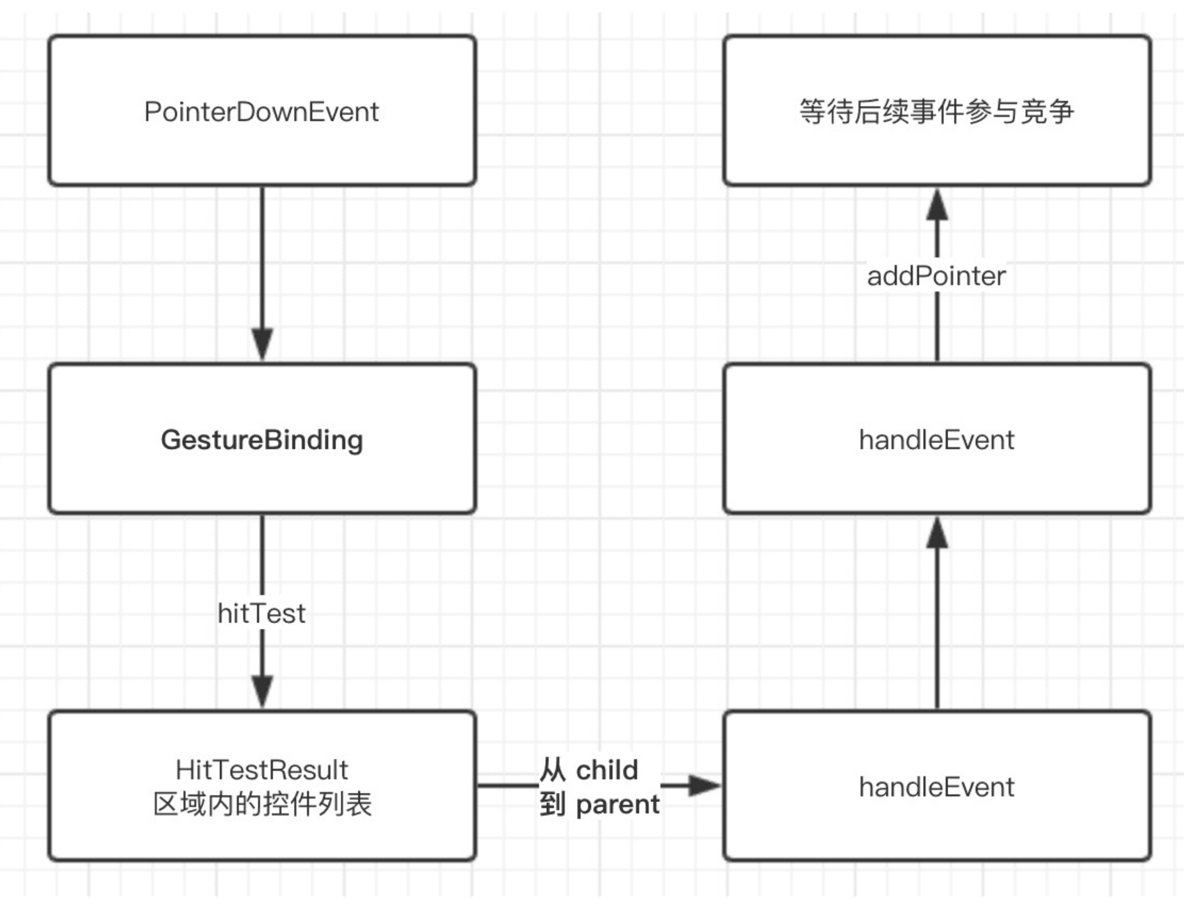
- **`Widget`** ：就是我们平常写的控件， `Flutter` 宇宙中万物皆 `Widget` ，它们都是不可变一帧，同时也是被人吐槽很多的嵌套模式，当然换个角度，事实上你把他当作 `Widget` 配置文件来写或者就好理解了。
- **`Element`** ：它是 `BuildContext` 的实现类， `Widget` 实现跨帧保存的 `state` 就是存放在这里，同时它也充当了 `Widget` 和 `RenderObject` 之间的桥梁。
- **`RenderObject`** ：它才是真正干活（layout、paint）等，同时它才是真实的 “dom” 。
- **`Layer`** ：一整块的重绘区域（isRepaintBoundary），决定重绘的影响区域。

> `skia` 在绘制的时候， `saveLayer` 是比较消耗性能的，比如透明合成、 `clipRRect` 等等都会可能需要 `saveLayer` 的调用， 而 `saveLayer` 会清空GPU绘制的缓存，导致性能上的损耗，所以开发过程中如果掉帧严重，可以针对这一块进行优化。



4、手势

`Flutter` 在手势中引入了竞技的概念，`Down` 事件在 `Flutter` 中尤为重要。



- `PointerDownEvent` 是一切的起源，在 `Down` 事件中一般不会决出胜利者。
- 在 `MOVE` 和 `UP` 的时候才竞争得到响应。
- 以点击为例子：`Down` 时添加进去参与竞争，`UP` 的时候才决定谁胜利，胜利条件是：

I、`UP` 的时候如果只有一个，那么就是它了。

II、`UP` 的时候如果有多个，那么强制队列里第一个直接胜利。

- 这里包含了有趣的点就是，**都在 `UP` 的时候才响应，那么 Down 事件怎么先传递出去了?**

`FLutter` 在这里做了一个 `didExceedDeadline` 机制，事实上在上面的 `addPointer` 的时候，会启动了一个定时器，默认 100 ms，如果超过指定时间没 `UP` ，那就先执行这个 `didExceedDeadline` 响应 `Down` 事件。

- 那问题又来了，如果这时候队列里两个呢?

它们的 `onTapDown` 都会被触发，但是 `onTap` 只有一个获得。

- 如果有两个滑动 `ScrollView` 嵌套呢?

举个简单的例子，两个 `SingleChildScrollView` 的嵌套时，在布局会经历：

> `performLayout` -> `applyContentDimensions` -> `applyNewDimensions` -> `context.setCanDrag(physics.shouldAcceptUserOffset(this));`

只有 `shouldAcceptUserOffset` 为 `ture` 时，才会添加 `VerticalDragGestureRecognizer` 去处理手势。

而判断条件主要是 `return math.max(0.0, child.size.height - size.height);` ，也就是**如果 child Scroll 的 height 小于父控件 Scroll 的时候，就会出现 child 不添加 VerticalDragGestureRecognizer 的情况，这时候根本就没有竞争了。**

5、动画

`Flutter` 中的动画是怎么执行的呢?

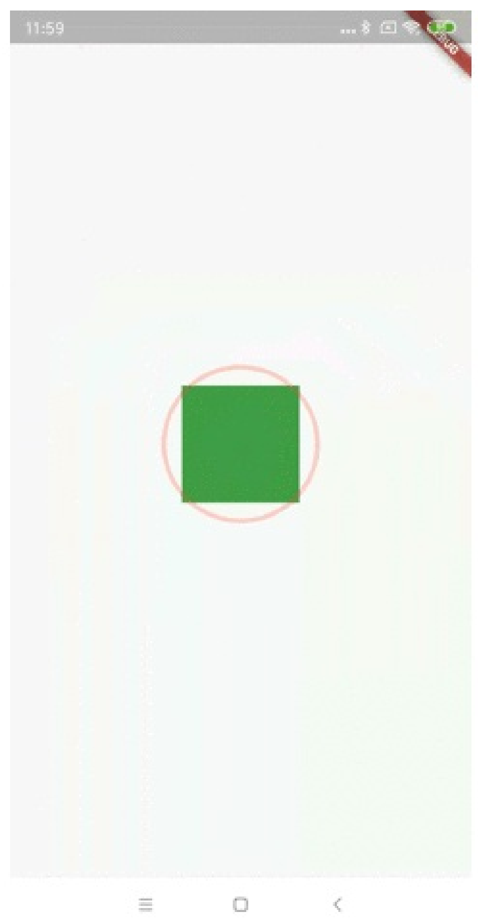
我们先看一段代码，然后这段代码执行的效果如下图2所示。

那既然 `Widget` 都是一帧，那么动画肯定有 `setState` 的地方了。

首先这里有个地方可以看下，这时候 200 这个数值执行后是会报错的，因为白框内可见 `Tween` 中的 `T` 在这时候会出现既有 `int` 又有 `double` ，无法判断的问题，所以真实应该是 200.0 。

```
class _DynamicPageState extends State<DynamicPage> with SingleTickerProviderStateMixin {
  AnimationController controller;
  Animation animation;
  @override
  void initState() {
    controller = new AnimationController(
        duration: const Duration(milliseconds: 3000), vsync: this);
    ///这里写了的 200 会不会有问题 dynamic
    animation = new Tween(begin: 0.0, end: 200).animate(controller)
      ..addListener(() {
        setState(() {
          //do change
        });
      });
    controller.repeat();
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Container(
              height: 100,
              width: 100,
              color: Colors.green,
              child: new InkWell(
                onTap: () {
                  print("InkWell");
                },
                child: new CustomPaint(
                    key: new GlobalKey(),
                    foregroundPainter: new _AnimationPainter(repaint: animation)),
              ),
            ),
          ],
        ),
      )// This trailing comma makes auto-formatting nicer for build methods.
    );
  }
}
```

```
class Tween<T extends dynamic> extends Animatable<T> {

  Tween({ this.begin, this.end });

  T begin;

  T end;
```



同时你发现没有，代码中 `parent` 的 `Container` 在 只有100的情况下，它的 `child` 可以正常的画 200，这是因为我们的 `paint` 没有跟着 `RenerObjcet` 的大小走， 所以一般情况下，整个屏幕都是我们的画版，**Canvas 绘制与父控件大小可以没关系。**

同时动画是通过 `vsync` 同步信号去触发的，就是我们 mixin 的 `SingleTickerProviderStateMixin` ，它内部的 `Ticker` 会通过 `SchedulerBinding` 的 `scheduleFrameCallback` 同步信号触发重绘 。

> 动画后的控件的点击区域，和你的动画数据改变的是 paint 还是 layout 有关 。

6、状态管理

`scope_model` 、 `flutter_redux` 、 `fish_redux` 、甚至还有有 `dva_flutter` 等等，可以看出状态管理在 `flutter` 中和前端十分相近。

这里简单说说 `scope_model` ，它只有一个文件，但是很巧妙，它利用的就是 `AnimationBuilder` 的特性。

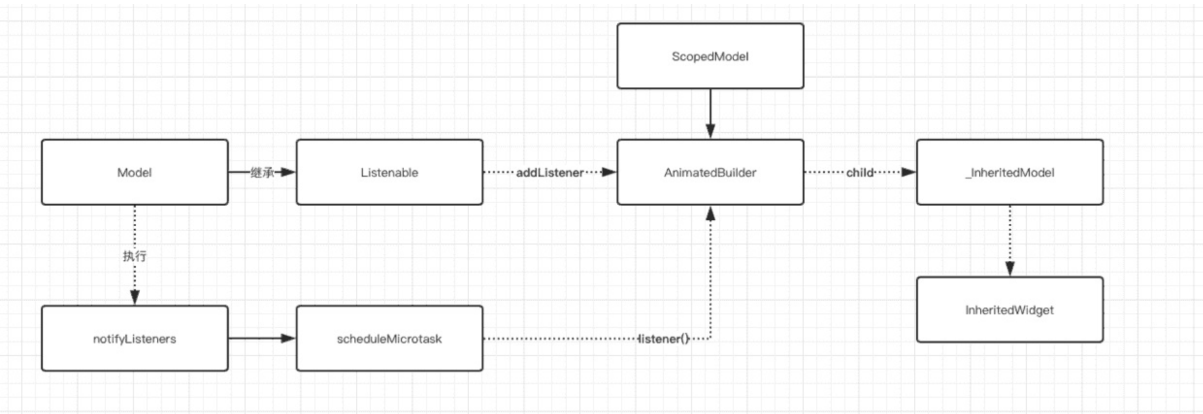
如下图是使用代码，在前面我们知道，状态管理使用的是 `InheritedWidget` 实现共享的，而当我们对 `Model` 进行数据改变时，通过调用 `notifyListeners` 通知页面更新了。

由 Xnip 截图

```
class ScopedPage extends StatelessWidget {
  final CountModel _model = new CountModel();
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: new Text("scoped"),
        ),
        body: Container(
          child: new ScopedModel<CountModel>(
            model: _model,
            child: CountWidget(),
          ),
        ));
  }
}

class CountWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new ScopedModelDescendant<CountModel>(
        builder: (context, child, model) {
          return new Column(
            children: <Widget>[
              new Expanded(child: new Center(child:
              new Text(model.count.toString()))),
              new Center(
                child: new FlatButton(
                    onPressed: () {
                      model.add();
                    },
                    color: Colors.blue,
                    child: new Text("+")),
              ),
            ],
          );
        });
  }
}

class CountModel extends Model {
  static CountModel of(BuildContext context) =>
      ScopedModel.of<CountModel>(context);

  int _count = 0;

  int get count => _count;

  void add() {
    _count++;
    notifyListeners();
  }
}
```

这里的原理是什么呢?

- 其实 `scope_model` 内部利用了 `AnimationBuilder` ，而 `Model` 实现了 `Listenable` 接口。
- 当 `Model` 设置给了 `AnimationBuilder` 时， `AnimationBuilder` 会执行 `addListener` 添加监听，而监听方法里会执行 `setState` 。
- 所以我们改变 `set` 方法时调用 `notifyListeners` 就触发了 `setState` 去更新了，这样体现出了前面说的 `FLutter` 常见的开发模式。



三、混合开发

以 `Android` 的角度来说，从方便调试和解耦集成上，我们一般会以 `aar` 的形式集成混合开发，这里就会涉及到 `gradle` 打包的一个概念。

1、如下代码所示，在项目中进行 `gradle` 脚本修改，组件化开发模式，用 `apk` 开发，用 `aar` 提供集成，正常修改 `gradle` 代码即可快速打包。

```
def isLib = true

if(isLib) {
    apply plugin: 'com.android.library'
} else  {
    apply plugin: 'com.android.application'
}

apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"

android {
    defaultConfig {
        if(!isLib) {
            applicationId "com.shuyu.flutter_app_lib"
        }
        if(isLib) {
            ndk {
                //设置支持的SO库架构
                abiFilters 'armeabi', 'armeabi-v7a', 'x86'
            }
        }
    }
}
```

那如果 `Flutter` 的项目插件带有本地代码呢?

> 如果开发过 `React Native` 的应该知道，在原生插件安装时会需要执行 `react-native link` ，而这时候会修改项目的gradle 和java代码。

2、 和 `React Native` 很有侵入性相比， `Flutter` 就很巧妙了。

如下图所示，安装过的插件会出现在 `.flutter_plugins` 文件中，然后通过读取文件，动态在 `setting.gradle` 和 `flutter.gradle` 中引入和依赖：

```
static
.flutter-plugins
.gitignore
.metadata
.packages
1.jpg
2.jpg
3.jpg
analysis_options.yaml
download.png
folder.png
framework2.png
GSYGithubFllutter-1.3.0.ipa
ios.gif
```

```
1  android_intent=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/android_intent-0.3.0+2/
2  connectivity=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/connectivity-0.4.3+1/
3  device_info=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/device_info-0.4.0+1/
4  flutter_lottie=/Users/guoshuyu/.pub-cache/git/flutter_lottie-a59dffd8282746de1c0e463ae12bbd737205aec1/
5  flutter_statusbar=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/flutter_statusbar-0.0.1/
6  flutter_webview_plugin=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/flutter_webview_plugin-0.3.4/
7  fluttertoast=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/fluttertoast-3.0.4/
8  package_info=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/package_info-0.4.0+3/
9  path_provider=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/path_provider-0.5.0+1/
10 permission_handler=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/permission_handler-3.0.0/
11 share=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/share-0.6.1/
12 shared_preferences=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/shared_preferences-0.5.2/
13 sqflite=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/sqflite-1.1.5/
14 url_launcher=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/url_launcher-5.0.2/
15 webview_flutter=/Users/guoshuyu/.pub-cache/hosted/pub.flutter-io.cn/webview_flutter-0.3.6/
16
```

```
def plugins = new Properties()
def pluginsFile = new File(flutterProjectRoot.toFile(), '.flutter-plugins')
if (pluginsFile.exists()) {
    pluginsFile.withReader('UTF-8') { reader -> plugins.load(reader) }
}

plugins.each { name, path ->
    def pluginDirectory = flutterProjectRoot.resolve(path).resolve('android').toFile()
    include ":$name"
    project(":$name").projectDir = pluginDirectory
}
```

```
File pluginsFile = new File(project.projectDir.parentFile.parentFile, '.flutter-plugins')
Properties plugins = readPropertiesIfExist(pluginsFile)

plugins.each { name, _ ->
    def pluginProject = project.rootProject.findProject(":$name")
    if (pluginProject != null) {
        project.dependencies {
            if (project.getConfigurations().findByName("implementation")) {
                implementation pluginProject
            } else {
                compile pluginProject
            }
        }
```

所以这时候我们可以参考打包，修改我们的gradle脚本，利用 fat-aar 插件将本地 projcet 也打包的 aar 里。

由 Xnip 截图

```
def isLib = true

if(isLib) {
    apply plugin: 'com.android.library'
} else  {
    apply plugin: 'com.android.application'
}

apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"
if(isLib) {
    apply plugin: 'com.kezong.fat-aar'
}

android {
    defaultConfig {
        if(!isLib) {
            applicationId "com.shuyu.flutter_app_lib"
        }
    }
}

dependencies {
    ///为库的方式才添加本地仓库依赖，这个本地仓库目前是从 include 那里读取的。
    if(isLib) {
        def flutterProjectRoot = rootProject.projectDir.parentFile.toPath()
        def plugins = new Properties()
        def pluginsFile = new File(flutterProjectRoot.toFile(), '.flutter-plugins')
        if (pluginsFile.exists()) {
            pluginsFile.withReader('UTF-8') { reader -> plugins.load(reader) }
        }
        plugins.each { name, _ ->
            println name
            embed project(path: ":$name", configuration: 'default')
        }
    }

}
```

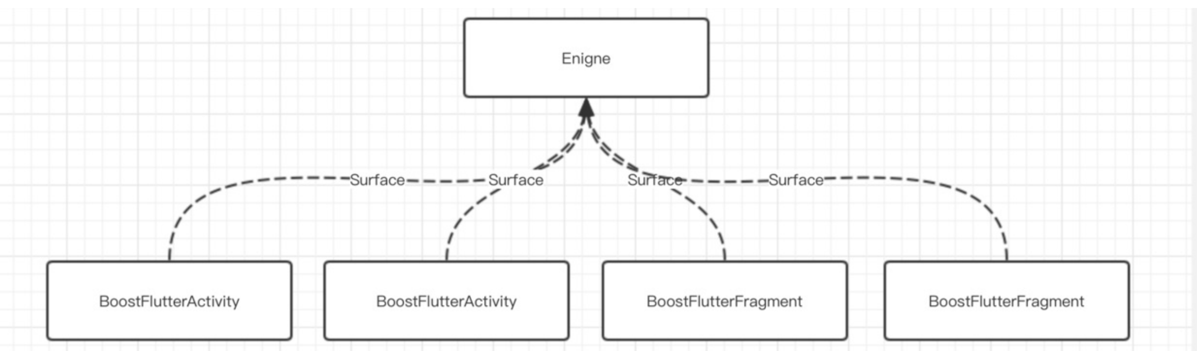
> **官方未来将有 `Flutter build aar` 的方法可提供使用。**

3、混合开发的最大痛点是什么？

肯定是堆栈管理!!! 所以项目开发了 `flutter_boost` 来解决这个问题。

- 堆栈统一到了原生层。
- 通过一个唯一 `engine` ，切换 `Surface` 渲染显示。
- 每个 `Activity` 就是一个 `Surface` ，不渲染的页面通过截图缓存画面。

> `flutter_boost` 截止到我测试的时间 2019-05-16, 只支持 1.2之前的版本



四、PlatformView

混合开发除了集成到原生工程，也有将原生控件集成到 Flutter 渲染树里里的需求。

首先我们看看没有 `PlatformView` 之前是如何实现 `WebView` 的，这样会有什么问题?

如下图所示，事实上 dart 中仅仅是用了一个 `SingleChildRenderObjectWidget` 用于占位，将大小传递给原生代码，然后在原生代码里显示出来而已。

```
class _WebviewScaffoldState extends State<WebviewScaffold> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: widget.appBar,
      resizeToAvoidBottomInset: widget.resizeToAvoidBottomInset,
      persistentFooterButtons: widget.persistentFooterButtons,
      bottomNavigationBar: widget.bottomNavigationBar,
      body: _WebviewPlaceholder(
        child: widget.initialChild ?? const Center(child: const CircularProgressIndicator()),
      ),
    );
  }
}

class _WebviewPlaceholder extends SingleChildRenderObjectWidget {
  @override
  RenderObject createRenderObject(BuildContext context) {
    return _WebviewPlaceholderRender(
      onRectChanged: onRectChanged,
    );
  }
}

class _WebviewPlaceholderRender extends RenderProxyBox {
  @override
  void paint(PaintingContext context, Offset offset) {
    super.paint(context, offset);
    final rect = offset & size;
    if (_rect != rect) {
      _rect = rect;
      notifyRect();
    }
  }
```

这样的时候必定会代码画面堆栈问题，因为这个显示脱离了 Flutter 的渲染树，通过出现动画肯定会不一致。

4.1 AndroidView

AndroidView -> TextureLayer，利用Android 上的副屏显示与虚拟内存显示原理。

- 共享内存，实时截图渲染技术。
- 存在问题，耗费内存，页面复杂时慢。

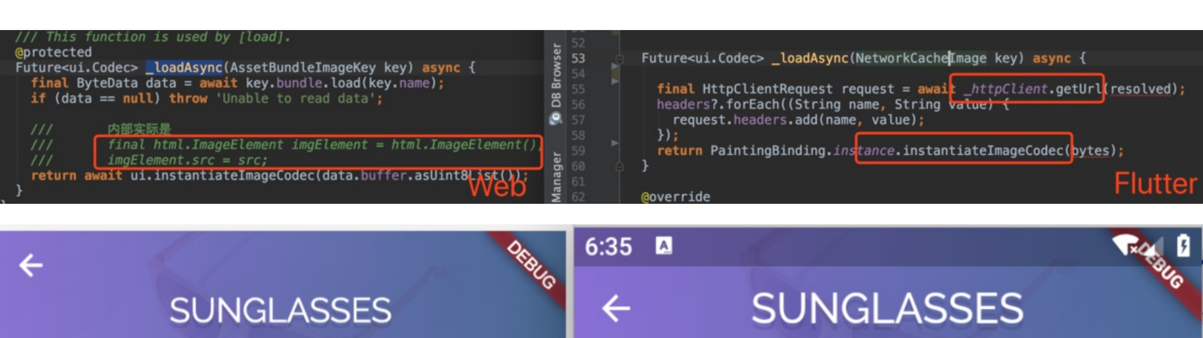
> 这部分因为之前以前聊过，就不赘述了

三、Flutter Web

RN因为是原生控件，所以在react 和react native 整合这件事上存在难度。

flutter 作为一个UI 框架，与平台无关，在web上利用的是dart2js的能力。 比如Image

- 因为 Flutter 是一套 UI 框架，整体 UI 几乎和平台无关，这和 React Native 有很大的区别。（我在开发过程中几乎无知觉）
- 在 flutter_web 中 UI 层面与渲染逻辑和 Flutter 几乎没有什么区别，底层的一些区别如：flutter_web 中的 Canvas 是 EngineCanvas 抽象，内部会借助 dart2js 的能力去生成标签。
- React Native 平台关联性太强，而 Flutter 在多平台上优势明显。我们期待官方帮我们解决大部分的适配问题。



SUNGLASSES

Flutter enables interactive animation
3K views – 5 days

Flutter enables interactive animation
3K views – 5 days

Flutter enables interactive animation
3K views – 5 days

Flutter enables interactive animation
3K views – 5 days

Flutter enables interactive animation
3K views – 5 days

6:35

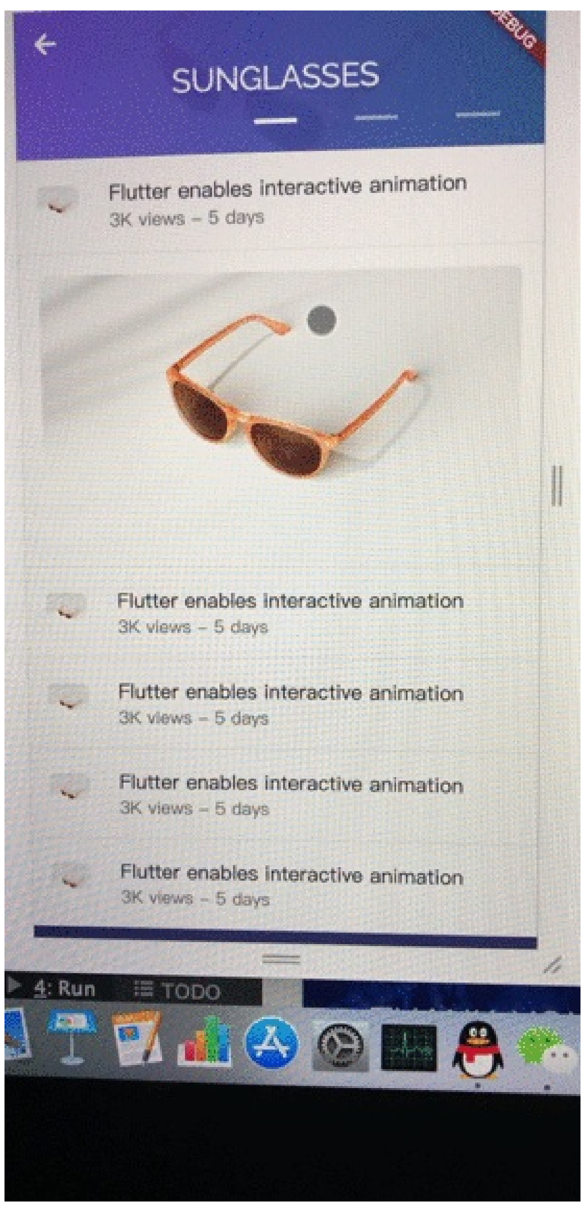
SUNGLASSES

Flutter enables interactive animation
3K views - 5 days

Flutter enables interactive animation
3K views - 5 days

Flutter enables interactive animation
3K views - 5 days

Flutter enables interactive animation



Flutter 的平台无关能力能带来什么?

- 1、某些功能页面，可以一套代码实现，利用插件安装引入，在web、移动app、甚至 pc 上，都可以编译出对应平台的高性能代码，而不会像 Weex 等一样存在各种兼容问题。
- 2、在应用上可以快速实现“降级策略”，比如某种情况下应用产生奔溃了，可以替换为同等 UI 的 h5 显示，而这些代码只需要维护一份。

RTC 开发者社区 | RTC Dev Meetup
圆桌讨论
邢尧
顺丰科技 全栈工程师
张乾泽
声网Agora高级架构师
郭树煜
移动开发工程师
小德
移动开发高级工程师

RTC 开发者社区
I came, I saw, I fixed

资源推荐

- RTC社区： https://rtcdeveloper.com
- Github： https://github.com/CarGuo/
- **开源 Flutter 完整项目： https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目: https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目： https://github.com/CarGuo/GSYFlutterBook**