Table of Contents

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 。

- 如果克隆太慢或者图片看不到，可尝试码云地址下载
- GSY新书：《Flutter开发实战详解》上架啦：京东 / 当当 / 电子版京东读书和Kindle

> 这个项目和新书《Flutter开发实战详解》是两个不同的内容哦，不要搞混了～

- 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**

公众号	掘金	知乎	CSDN	简书
GSYTech	点我	点我	点我	点我

目录

    - Flutter 2.0 下混合开发浅析
    - Flutter 搭建 iOS 命令行服务打包发布全保姆式流程
    - 给 Android 和 iOS 开发人员不一样的 Flutter 基础讲解
- 不一样角度带你了解 Flutter 中的滑动列表实现

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	
GSYFlutterBook	
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。

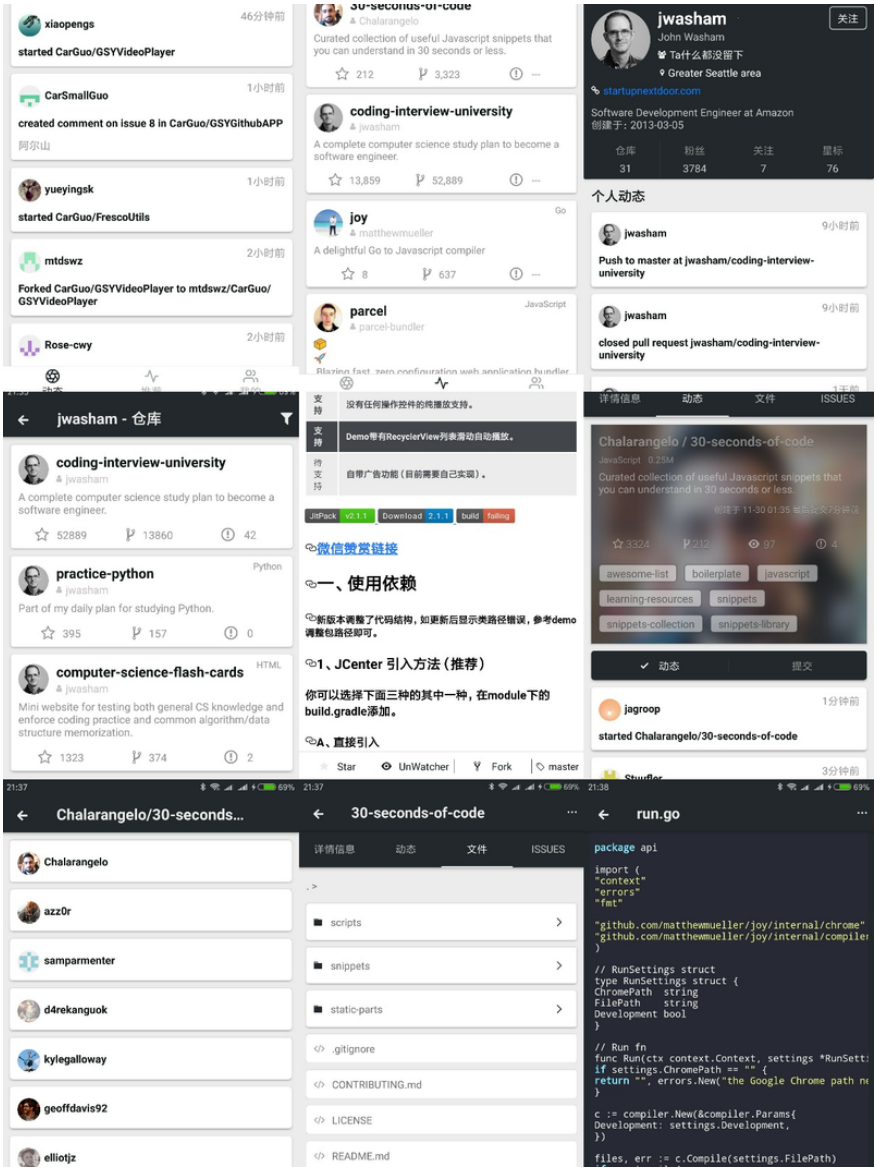
同时为了更方便学习，**《Flutter完整开发实战详解》** 系列文章会同步整合到 **GSYFlutterBook** 项目中，项目将通过**在线 Gitbook 和离线 PDF 方式， 进一步满足你的学习要求。**

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**

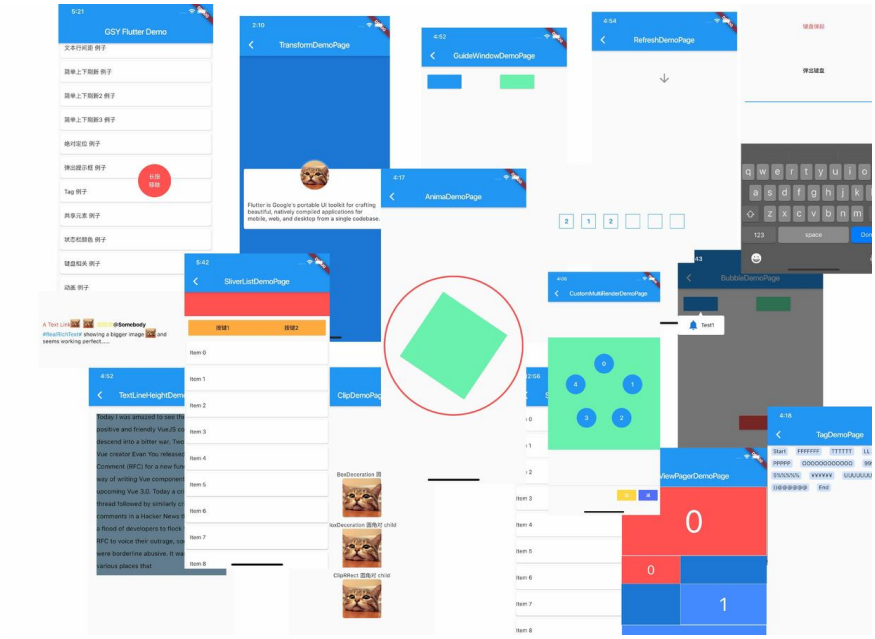


三、GSYFlutterDemo

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
export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.
```

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() => _DemoStateWidgetSt
}

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1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 [ S
    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.circula
          ///设置了decoration的color，就不能设置Container的
          color: Colors.black,
          ///边框
          border: new Border.all(color: Color(GSYColors
      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。

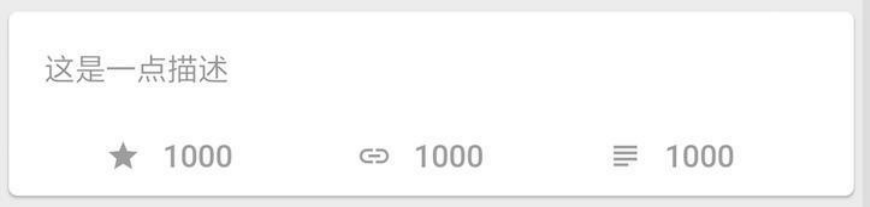
如代码中注释，布局内主要是现实一个居中的Icon图标和文本，中间间隔5.0的 padding：

```
///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
          ///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, font
            //超过的省略为...显示
            overflow: TextOverflow.ellipsis,
            //最长一行
            maxLines: 1,
          ),

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

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
              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.subText(
                          fontSize: 14.0,
                        ),
                        ///最长三行，超过 ... 显示
                        maxLines: 3,
                        overflow: TextOverflow.ellipsis,
                      ),
                      margin: new EdgeInsets.only(top: 6.
                      alignment: Alignment.topLeft),
                  new Padding(padding: EdgeInsets.all(10.

                  ///三个平均分配的横向图标文字
                  new Row(
                    crossAxisAlignment: CrossAxisAlignmen
                    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());
  }
}
```

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。**

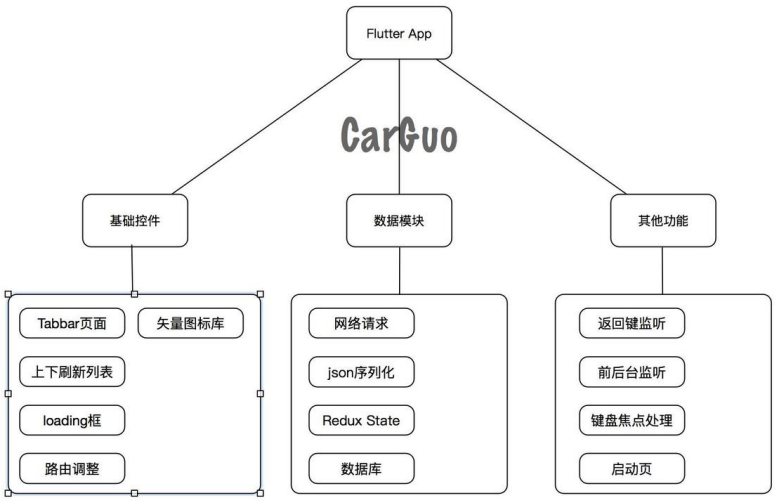
> 友情提示：本文所有代码均在 **GSYGithubAppFlutter** ，文中示例代码均可在其中找到，看完本篇相信你应该可以轻松完成如下效果。相关基础还请看篇章一。

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
    ///···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, lengt
  }

  @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。

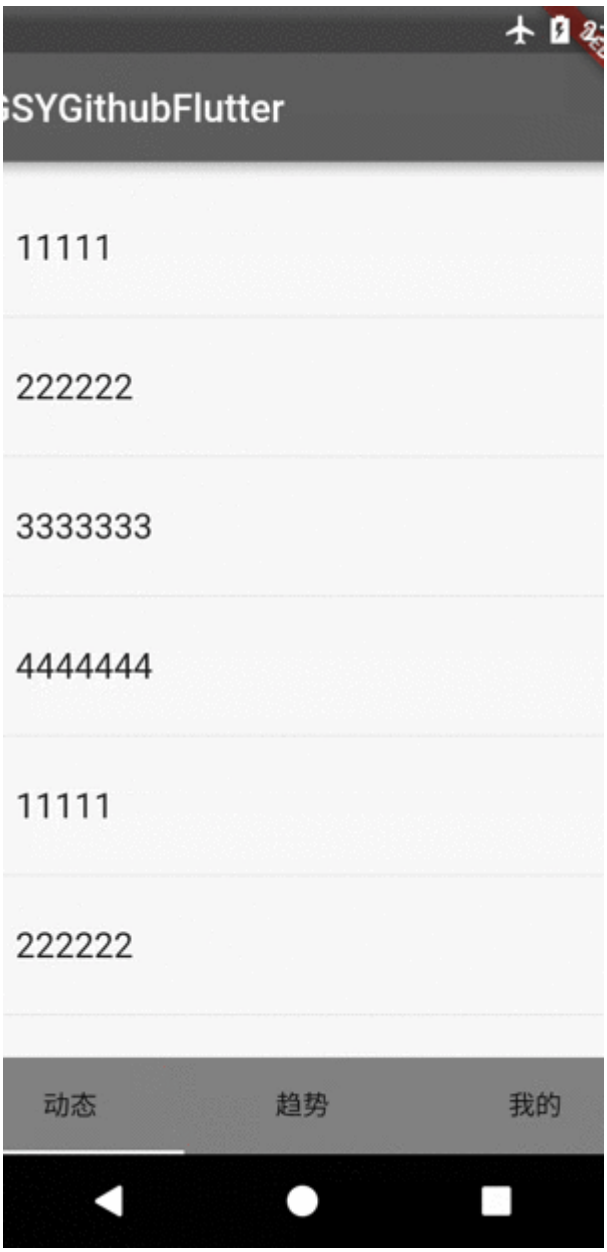
外部调用代码如下：每个 Tabbar 点击时，通过 `pageController.jumpTo` 跳转页面，每个页面需要跳转坐标为：**当前屏幕大小乘以索引 index** 。

```
class _TabBarBottomPageWidgetState extends State<TabBarBott

  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。

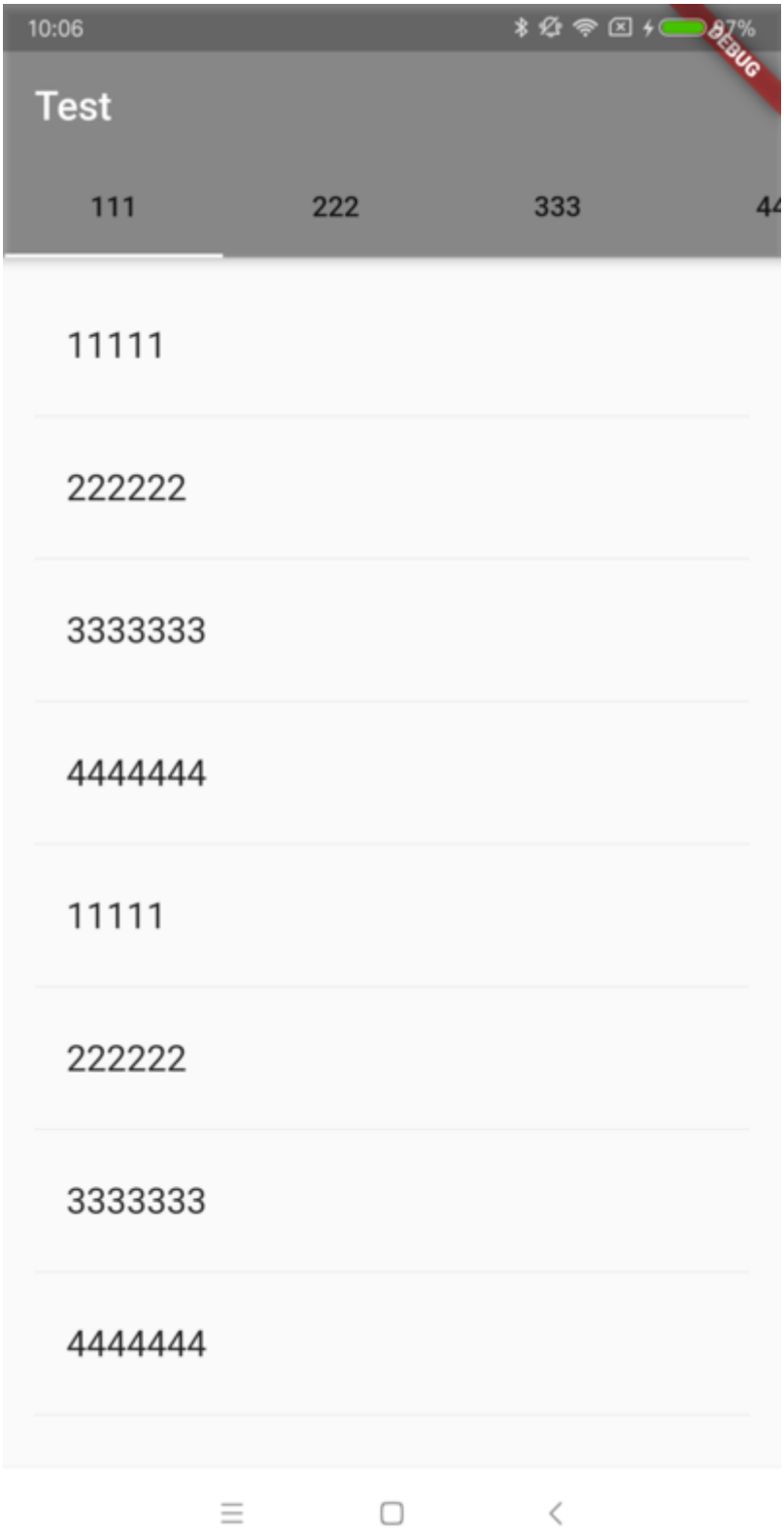


既然底部Tab页面都实现了，干脆顶部tab页面也一起完成。如下代码，和底部Tab页的区别在于：

- 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。

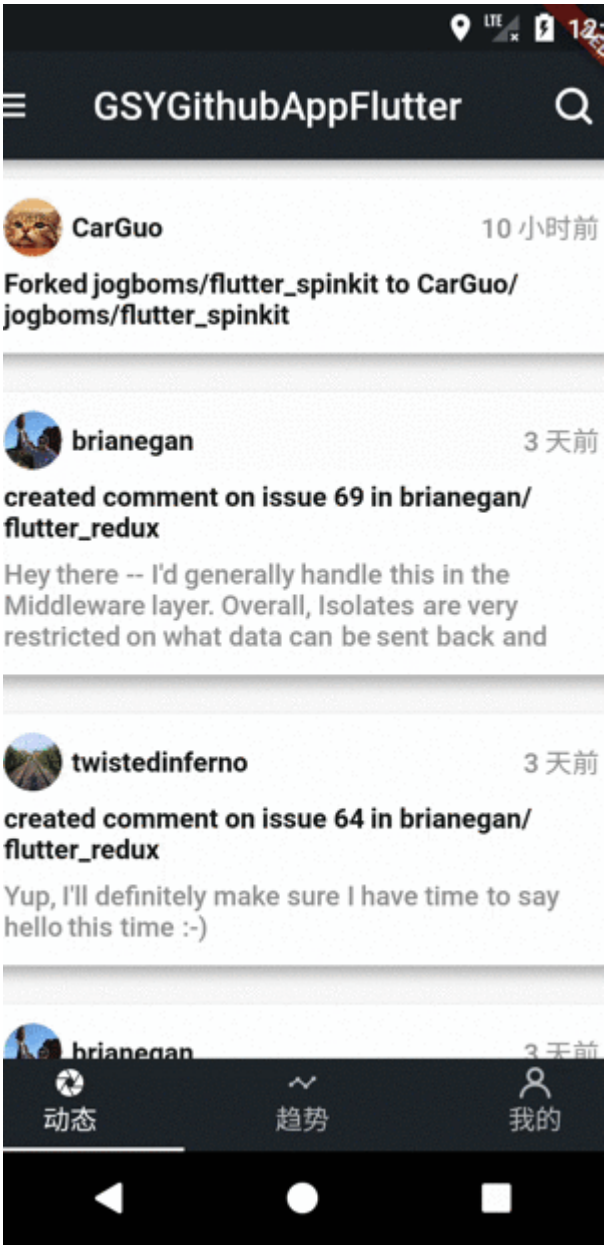
如下代码所示，通过 `RefreshIndicator` 控件可以简单完成下拉刷新工作，这里需要注意一点是：**可以利用 `GlobalKey<RefreshIndicatorState>` 对外提供 `RefreshIndicator` 的 `RefreshIndicatorState` ，这样外部就 可以通过 GlobalKey 调用 `globalKey.currentState.show();` ，主动显示刷新状态并触发 `onRefresh` 。**

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<GSYPullLoadWidg
  ///···
  final ScrollController _scrollController = new ScrollCont

  @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 == _scrollContr
        if (this.onLoadMore != null && this.control.needLoa
          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
      return (control.dataList.length > 0) ? control.dataLi
    } 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.dataLi
    }
  }

  ///根据配置状态返回实际列表渲染Item
  _getItem(int index) {
    if (!control.needHeader && index == control.dataList.le
      ///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
      return _buildProgressIndicator();
    } else if (control.needHeader && index == _getListCount
      ///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
      return _buildProgressIndicator();
    } else if (!control.needHeader && control.dataList.leng
      ///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
        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
        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
          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() {
       ///···
    }
}
```

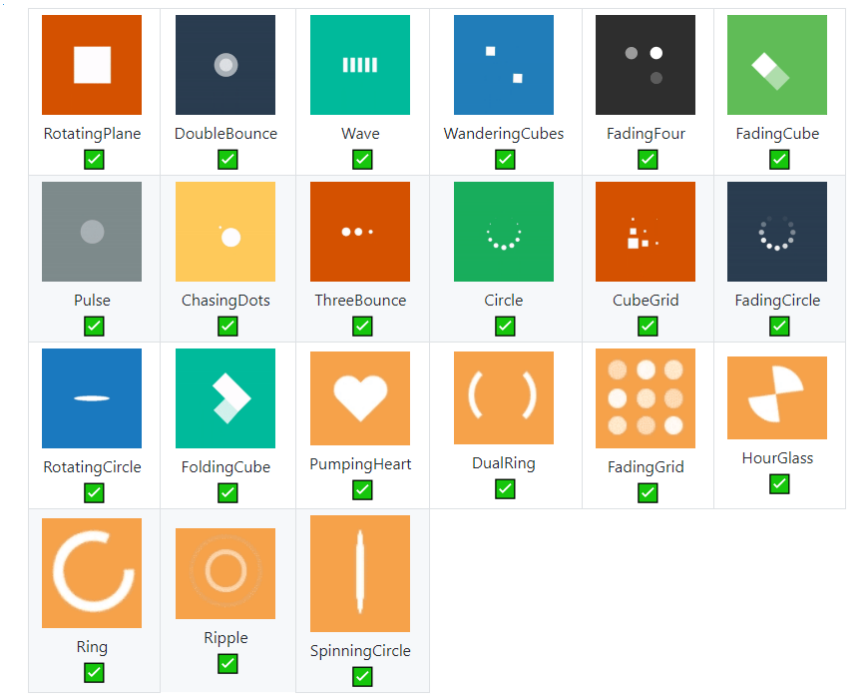


3、Loading框

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.cent
           ///loading框
           new SpinKitRotatingCircle(color: Color(0xFF242
           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:
            ),
          ),
         ///使用iconfont
          new Tab(
            child: new Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[ new Icon(IconData(0xe6d0,
            ),
          )
```

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', Mod

///带参数的路由跳转，并且监听返回
Navigator.push(context, new MaterialPageRoute(builder: (con
      ///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 contex
  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
      response = await dio.request(url, data: params, optio
    } 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。

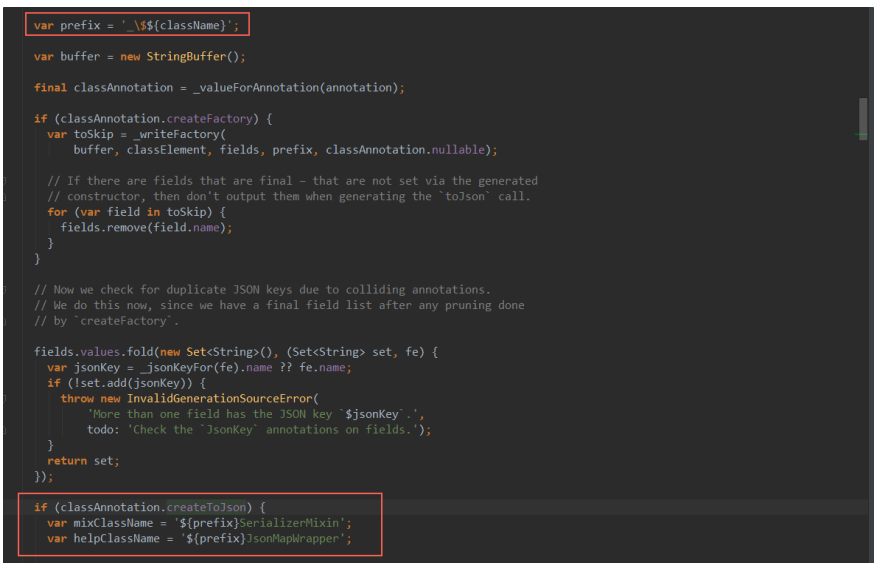
所以 `json_serializable` 插件诞生了， 中文网Json 对其已有一段教程，这里主要补充说明下具体的使用逻辑。

```
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
///Template.g.dart中创建了抽象类_$TemplateSerializerMixin，实
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 _$AASerialize
///所以当前类名为Template，生成的抽象类为 _$TemplateSerializerM
class Template extends Object with _$TemplateSerializerMix

  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 _$AAeFromJs
  ///所以当前类名为Template，生成的抽象类为 _$TemplateFromJson
  factory Template.fromJson(Map<String, dynamic> json) => _
}
```



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。

```
part of 'Template.dart';

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

abstract class _$TemplateSerializerMixin {
  String get name;
  int get id;
  int get pushId;
  Map<String, dynamic> toJson() =>
      <String, dynamic>{'name': name, 'id': id, 'push_id':
}
```

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
    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 UpdateUse
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, initialSta

  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 c
/// 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(new
```

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.
                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
        ///如果返回 return new Future.value(true); popped 就
        ///这里可以通过 showDialog 弹出确定框，在返回时通过 Navig
        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 WidgetsBi

  ///重写 WidgetsBindingObserver 中的 didChangeAppLifecycleS
  @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

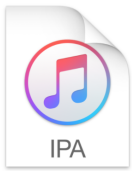
作为系列文章的第三篇，本篇将为你着重展示：**Flutter开发过程的打包流程、APP包对比、细节技巧与问题处理**，本篇主要描述的 Flutter 的打包、在开发过程中遇到的各类问题与细节，算是对上两篇的补全。

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

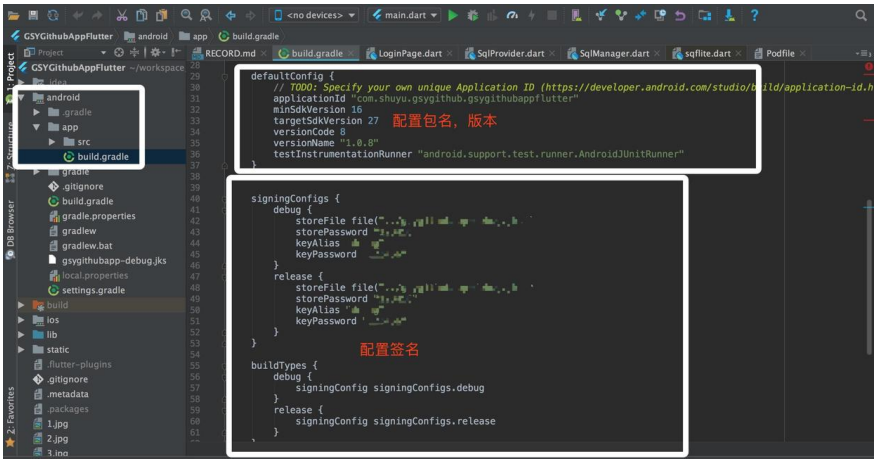
首先我们先看结果，如下表所示，是 **Flutter 与 React Native 、iOS 与 Android 的纵向与横向对比** 。

项目	IOS	Android
GSYGithubAppFlutter	 Runner-iPhone 6s Plus.ipa iOS 应用－14.3 MB	 GSYGithubAppFlutter-1.0.8.apk 文稿－11.2 MB
GSYGithubAppRN	 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。

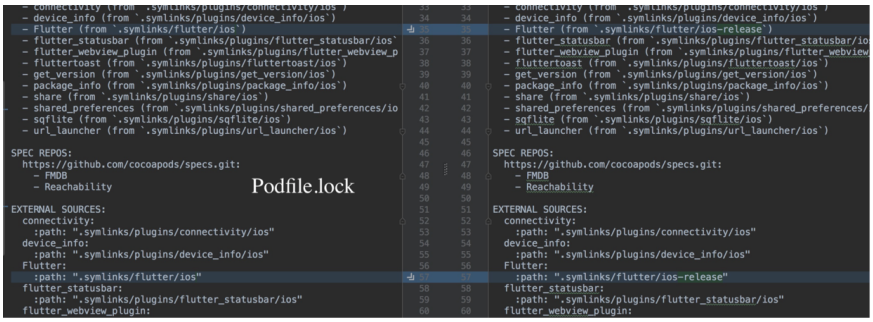
首先你需要一个 apple 开发者账号，然后创建证书、创建AppId，创建配置文件、最后在 `info.plist` 文件下输入相关信息，更详细可看官方的《发布的IOS版APP》的教程。

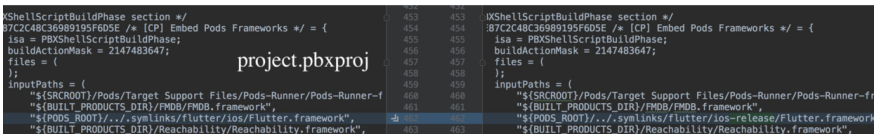
但由于笔者项目中使用了第三方的插件包如 `shared_preferences` 等，在执行 `Archive` 的过程却一直出现如下问题：

```
在 `Archive` 时提示找不到
#import <connectivity/ConnectivityPlugin.h>  ///file not fo
#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 armv
```

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。

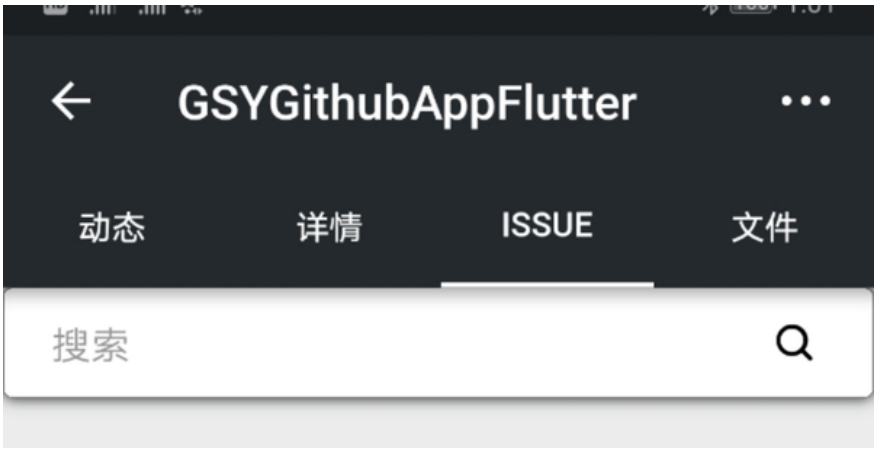
二、细节

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.shrinl
        padding: padding ?? const EdgeInsets.all(0.0),
        constraints: const BoxConstraints(minWidth: 0.0, mi
        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.shrinl
        padding: padding ?? const EdgeInsets.all(0.0),
        constraints: const BoxConstraints(minWidth: 0.0, mi
        ///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 TextEditingCo

@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
}


@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。

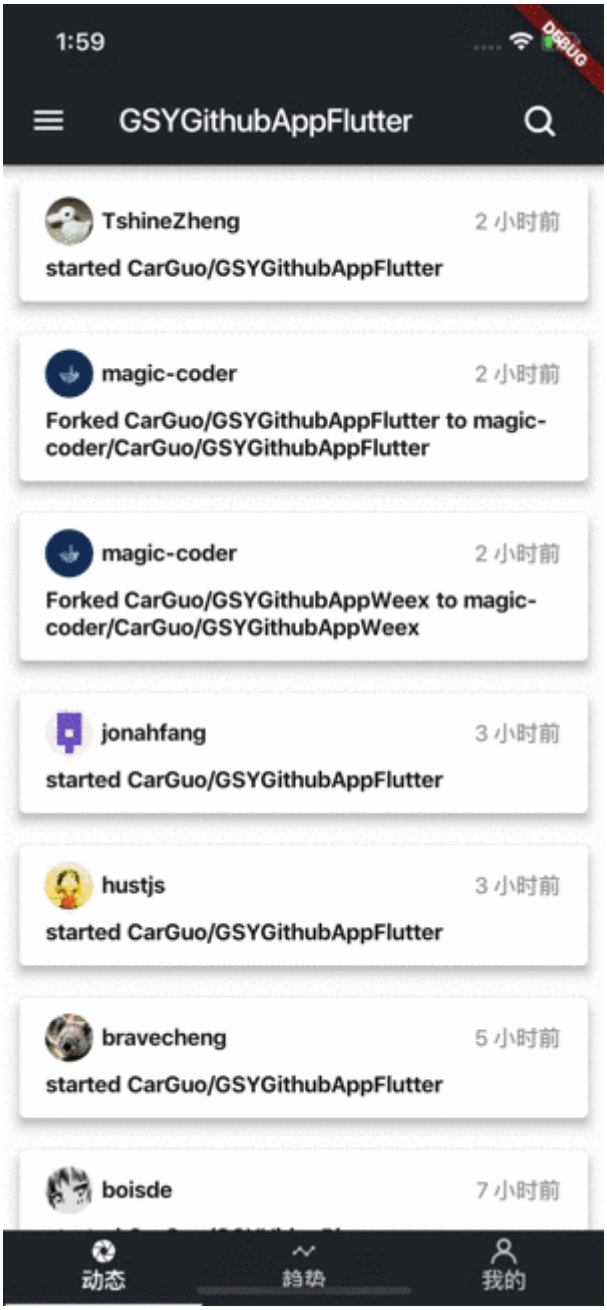
```
GlobalKey<RefreshIndicatorState> refreshIndicatorKey;

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

 @override
 Widget build(BuildContext context) {
   refreshIndicatorKey =  new GlobalKey<RefreshIndicatorSt
   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,
        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

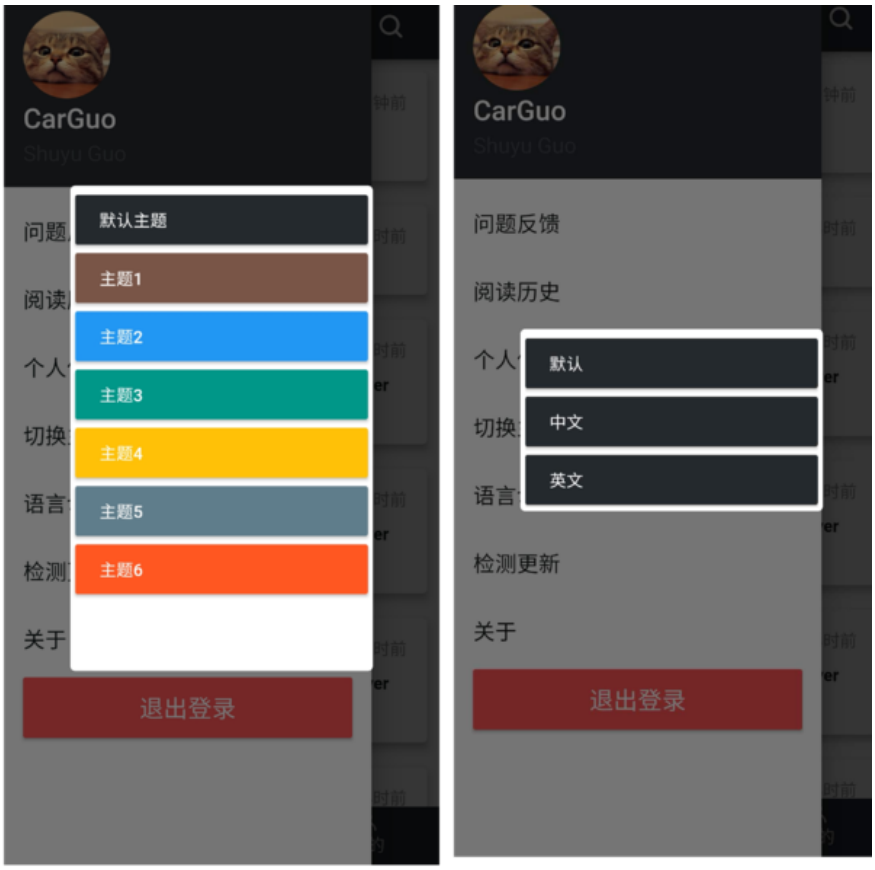
作为系列文章的第四篇，本篇主要介绍 Flutter 中 Redux 的使用，并结合 Redux 完成实时的**主题切换**与**多语言切换**功能。

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 。

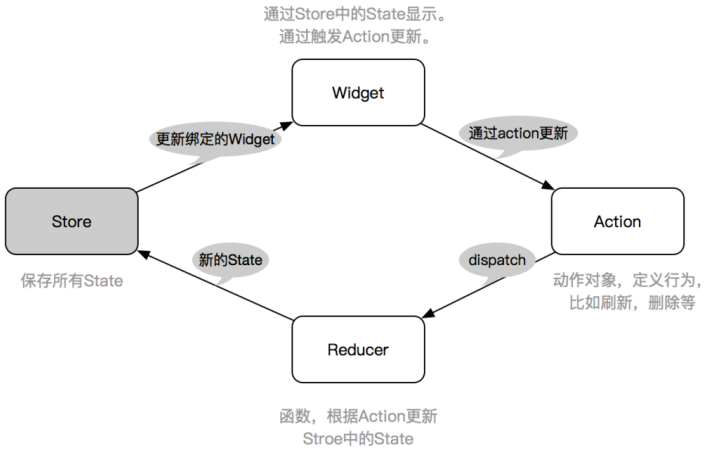


一、Redux

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>(Sta
///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 act
    userInfo: UserReducer(state.userInfo, action),

    ///通过自定义 ThemeDataReducer 将 GSYState 内的 themeData
    themeData: ThemeDataReducer(state.themeData, action),

    ///通过自定义 LocaleReducer 将 GSYState 内的 locale 和 act
    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(ne
```

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,
];
```

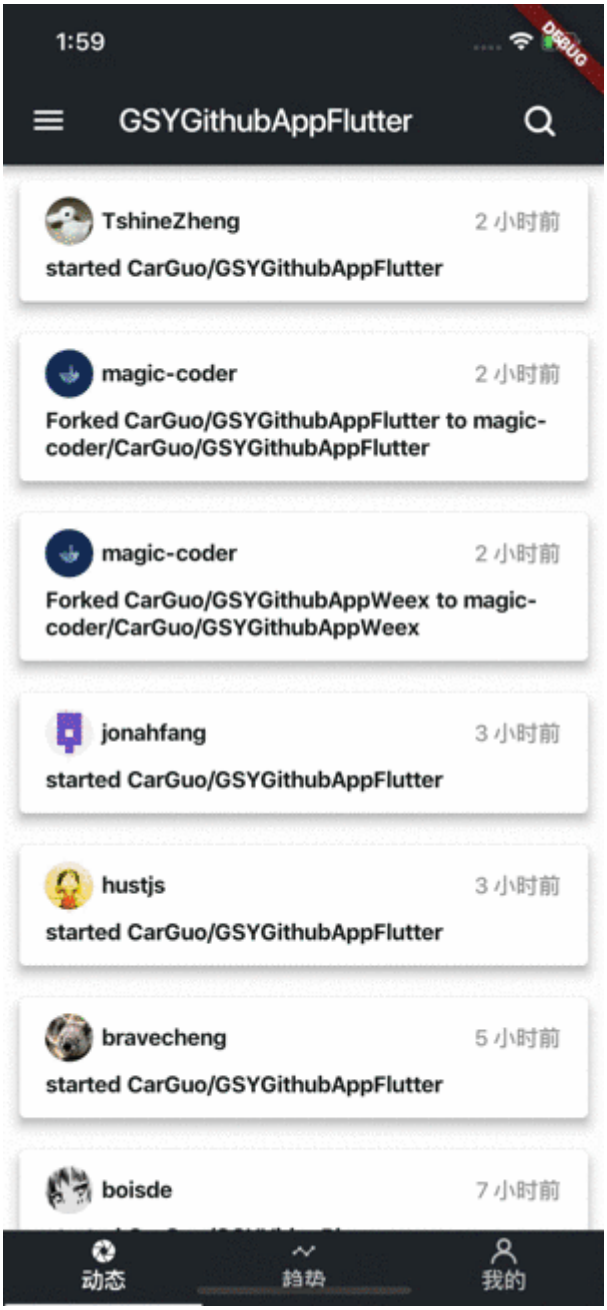
```
MaterialColor primarySwatch = const MaterialColor(
    primaryValue,
    const <int, Color>{
      50: const Color(primaryLightValue),
      100: const Color(primaryLightValue),
      200: const Color(primaryLightValue),
      300: const Color(primaryLightValue),
      400: const Color(primaryLightValue),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

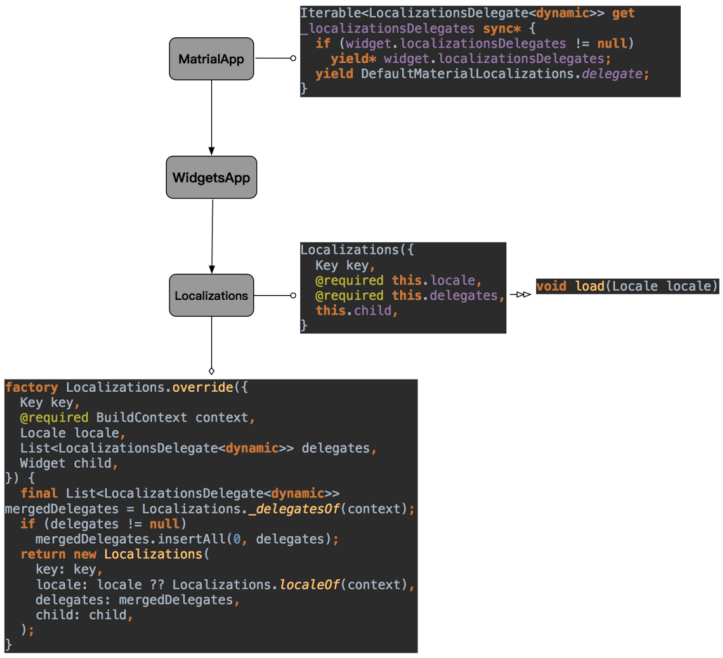
注意，因为你的 **MaterialApp** 也是一个 `StatefulWidget` ，如下代码所示，还需要利用 `StoreBuilder` 包裹起来，之后我们就可以通过 `dispatch` 修改主题，通过 `Theme.of(context).primaryColor` 获取主题色啦。

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

....

ThemeData  themeData = new ThemeData(primarySwatch: colors
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 LocalizationsDelegat

  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 GSYL
  }

  @override
  bool shouldReload(LocalizationsDelegate<GSYLocalizations>
    return false;
  }

  ///全局静态的代理
  static GSYLocalizationsDelegate delegate = new GSYLocaliz
}
```

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, st
      ///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,
       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
               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。

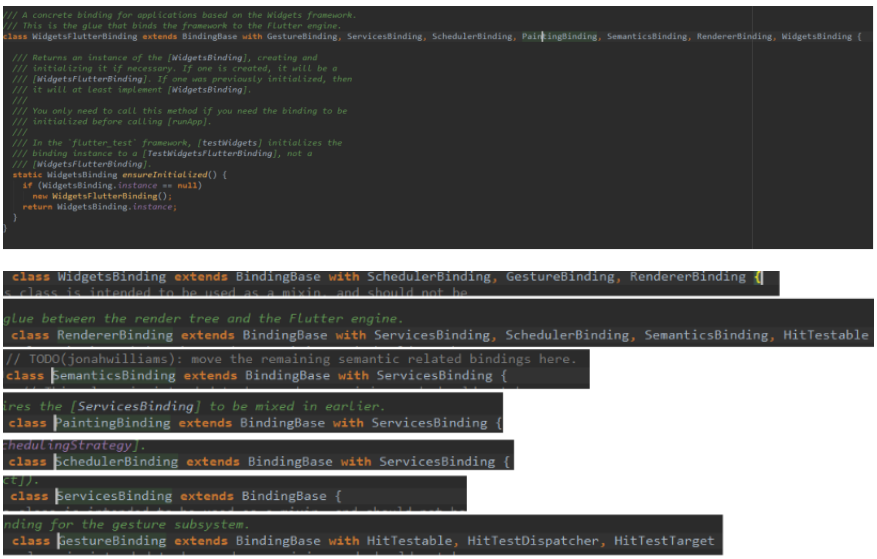
```
abstract class Base {
  a() {
    print("base a()");
  }

  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。



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，不出所料丶(￣▽￣)/。

```
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。

```dart
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);
}
```

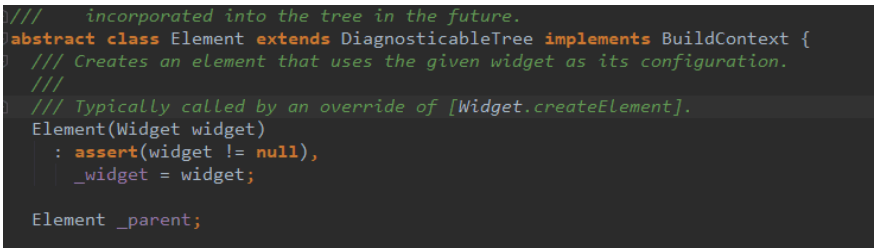
如上图所示，**InheritedWidget** 主要实现两个方法：

- 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。



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
```

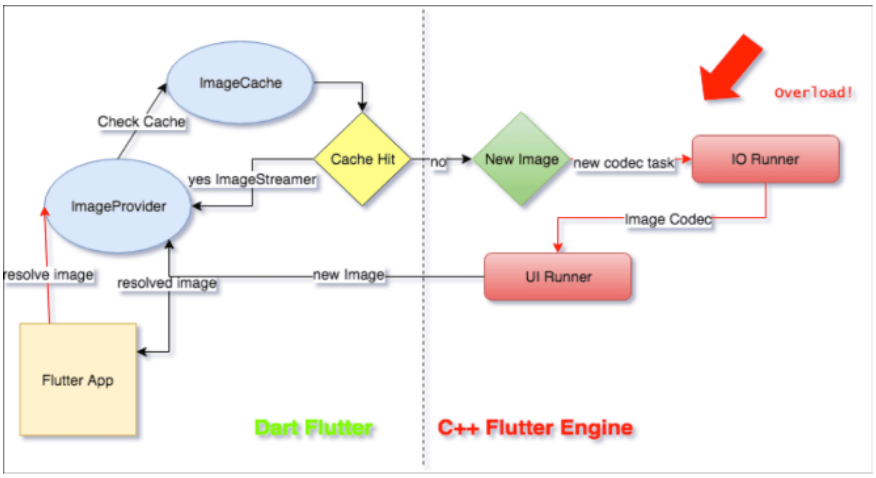
```
/// result of calling [State.setState] above the inherited widget.
@override
void notifyClients(InheritedWidget oldWidget) {
  if (!widget.updateShouldNotify(oldWidget))
    return;
  assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
  for (Element dependent in _dependents) {
    assert(()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

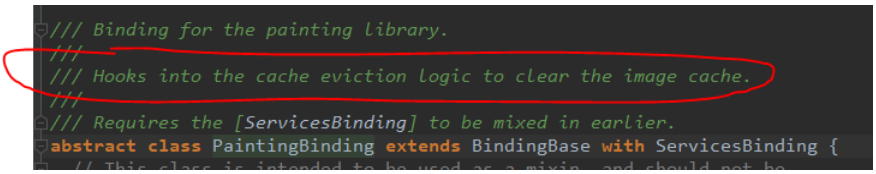


如上图所示，是图片缓存相关的流程，而目前的拮据处理是通过：

- 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



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 <<
```

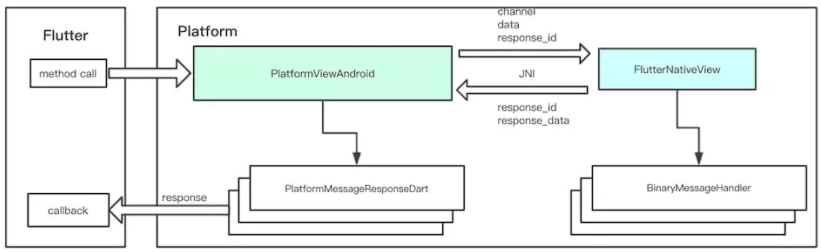
四、线程

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”* 的答案就是：**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。**

```dart
/// 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。

```dart
/// 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.fi
  print(renderBoxRed.size);
}

showPositions() {
  RenderBox renderBoxRed = fileListKey.currentContext.fi
  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。

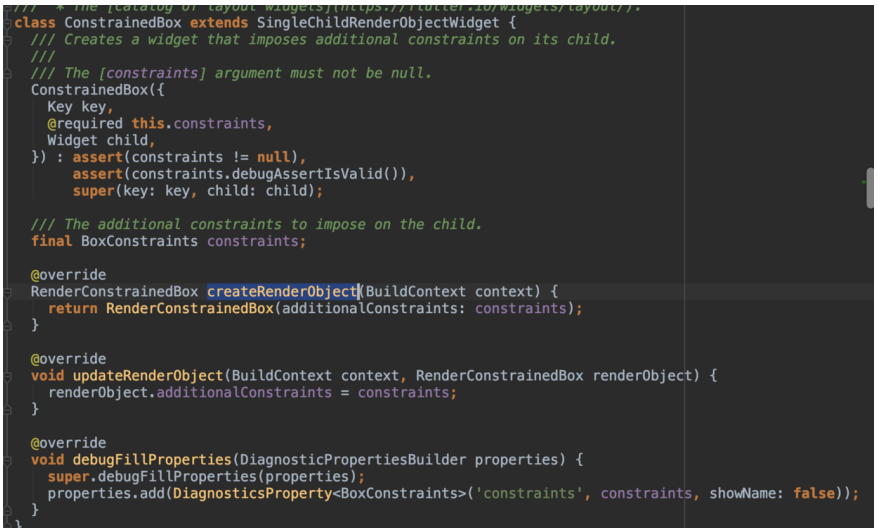
```dart
@override
Widget build(BuildContext context) {
  Widget current = child;

  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



```
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：

```dart
@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。

```dart
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：

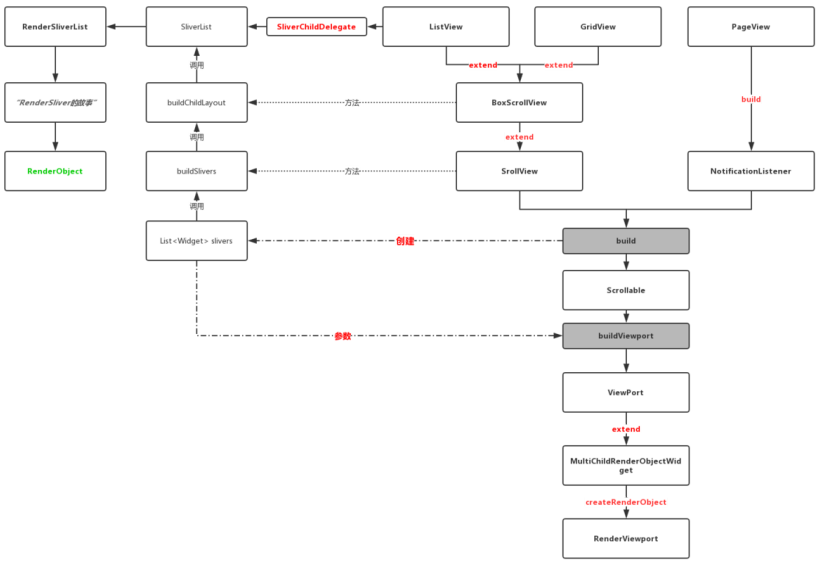
- `Row`、`Colum` 都继承了 `Flex`，而 Flex 继承了 `MultiChildRenderObjectWidget` 并通过 `RenderFlex` 创建了 `RenderBox`；
- `Stack` 同样继承 `MultiChildRenderObjectWidget` 并通过 `RenderStack` 创建了 `RenderBox`；

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 d
/// a [RenderSliverList], [RenderSliverFixedExtentList], [
/// 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: SliverGridDelegateWithMaxCrossAxisExter
        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：

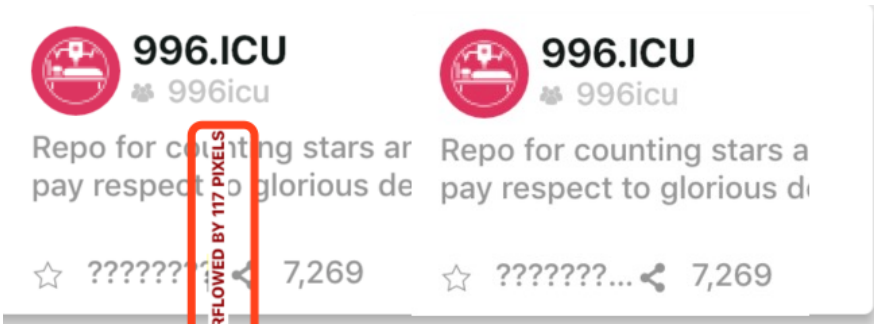
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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.fi
  print(renderBoxRed.size);
}

showPositions() {
  RenderBox renderBoxRed = fileListKey.currentContext.fi
  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。

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.instan
      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

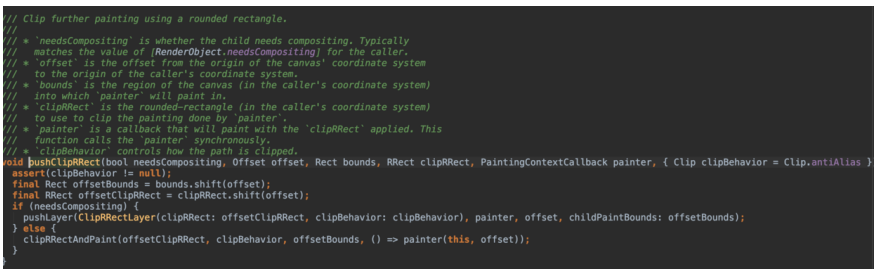
日常开发中我们大致上会使用两种圆角方案：

- 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。

```dart
/// 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?

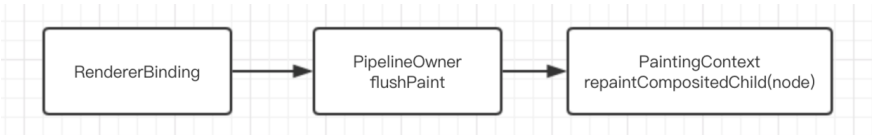
通过飞速查阅源码，我们可以首先了解到有：

- `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。**

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

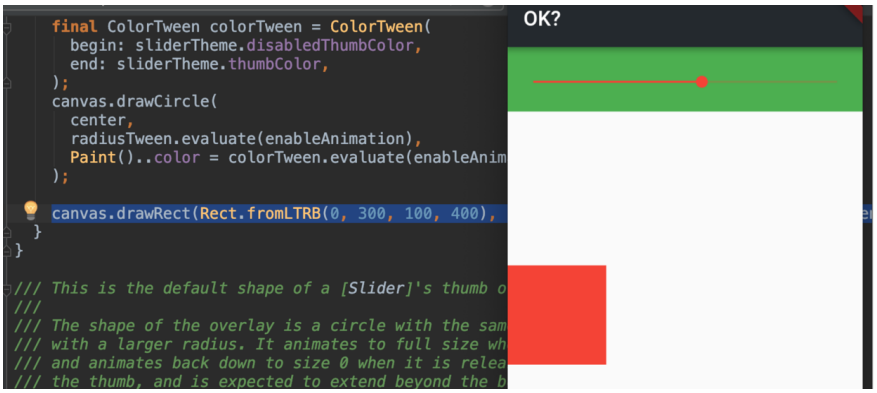
1、测试绘制

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。**

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。**

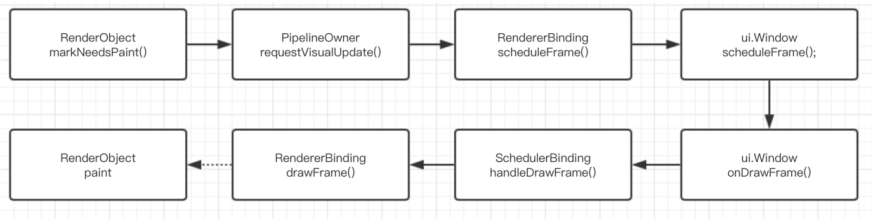
```
The canvas will allow painting outside these bounds.
The [estimatedBounds] rectangle is in the [canvas] coordina
```

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。**

```dart
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

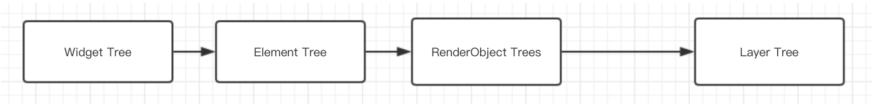
上文我们知道了，当 `isRepaintBoundary` 为 true 时，那么该区域就是一个可更新绘制区域，而当这个区域形成时， 其实就会新创建一个 **`Layer`** 。

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。

```dart
/// 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。**

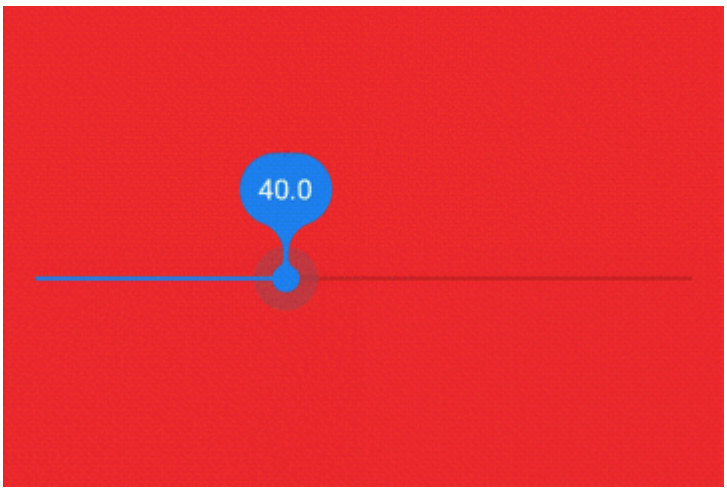
其次在 `RenderObject` 中还有一个属性叫 `needsCompositing` ，它会影响生成多少层的 **`Layer`** ，而这些 **`Layer`** 又会组成一棵 **Layer Tree** 。好吧，到这里又多了一个树，实际上这颗树才是所谓真正去给引擎绘制的树。



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

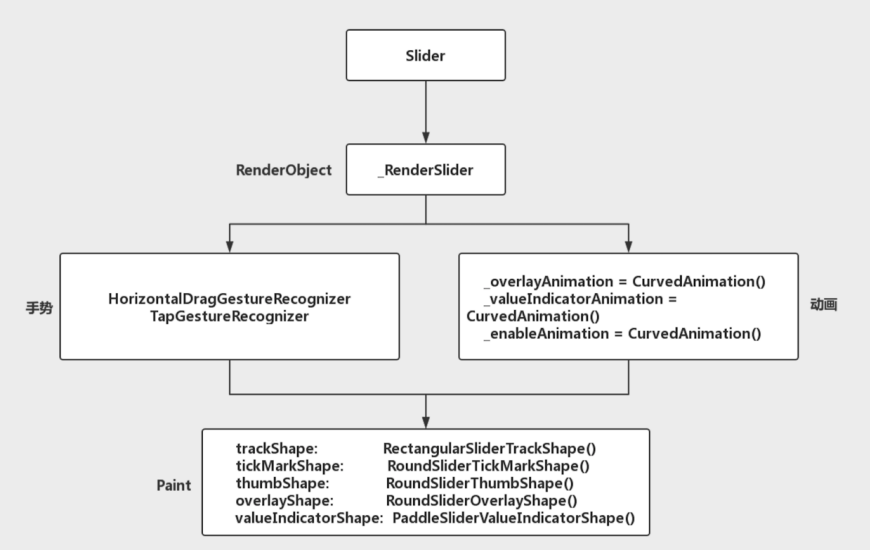
前面我们讲了那么多绘制的流程，现在让我们从 `Slider` 这个控件的源码，去看看一个绘制控件的设计实现吧。



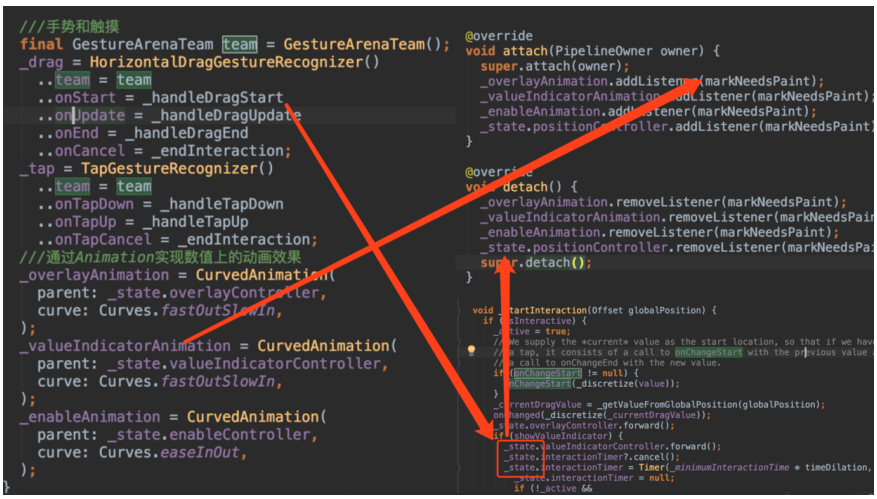
整个 `Slider` 的实现可以说是很 `Flutter` 了，大体结构如下图。

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。



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` 。

```dart
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?

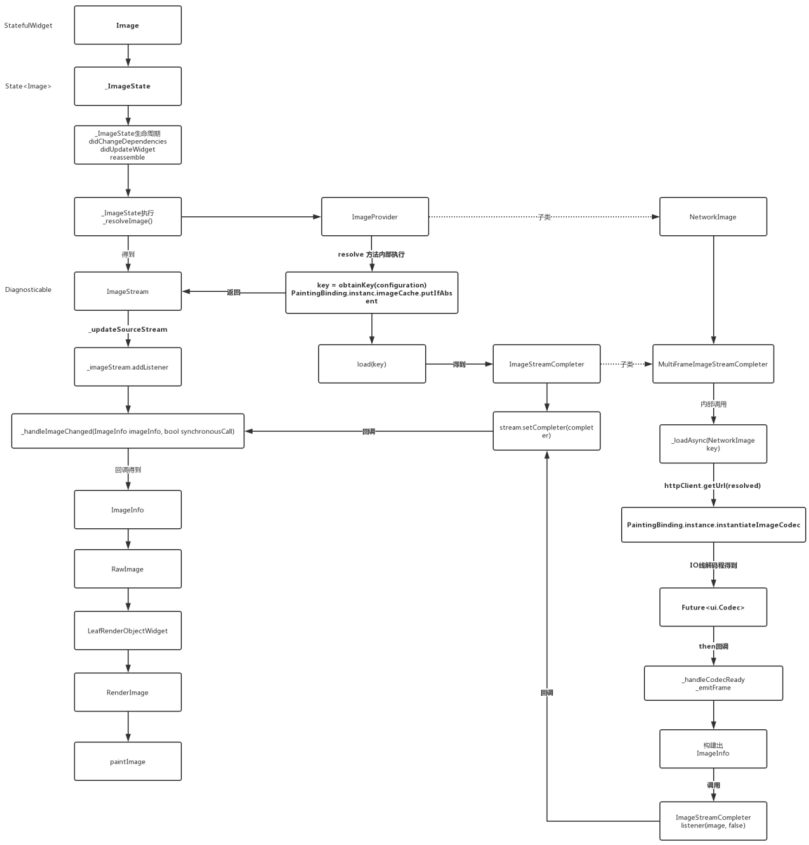
一、图片流程

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。

```dart
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().getFileFr
   if(fileInfo != null && fileInfo.file != null) {
     final Uint8List cacheBytes = await fileInfo.file.rea
     if (cacheBytes != null) {
       return PaintingBinding.instance.instantiateImageCo
     }
   }
   /// add this end

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

   final Uint8List bytes = await consolidateHttpClientRes
   if (bytes.lengthInBytes == 0)
     throw Exception('NetworkImage is an empty file: $res

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

   return PaintingBinding.instance.instantiateImageCodec(
 }
```

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：

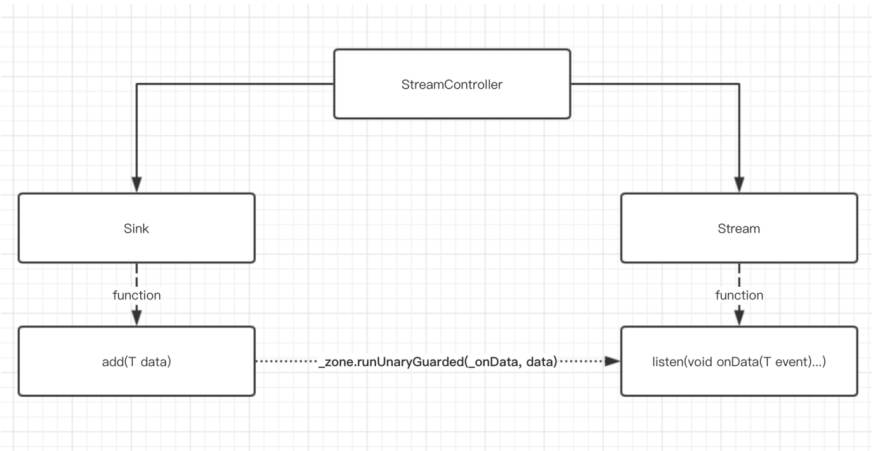
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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 = StreamC
  ///获取 StreamSink 做 add 入口
  StreamSink<List<String>> get _dataSink => _dataControlle
  ///获取 Stream 用于监听
  Stream<List<String>> get _dataStream => _dataController.s
  ///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(streamTrans
```

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!

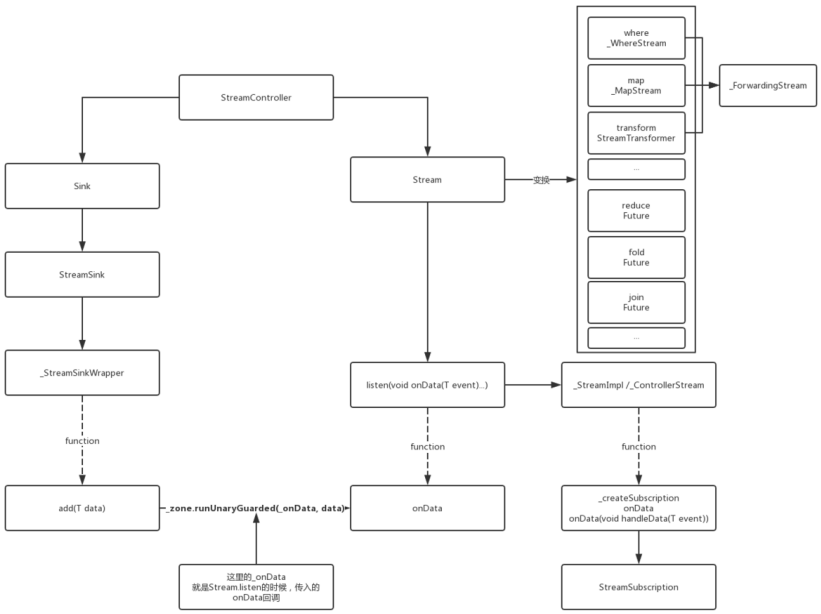
```
StreamBuilder<List<String>>(
    stream: dataStream,
    initialData: [ "none"] ,
    ///这里的 snapshot 是数据快照的意思
    builder: (BuildContext context, AsyncSnapshot<List<Str
      ///获取到数据，为所欲为的更新 UI
      var data = snapshot.data;
      return Container();
    });
```

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：

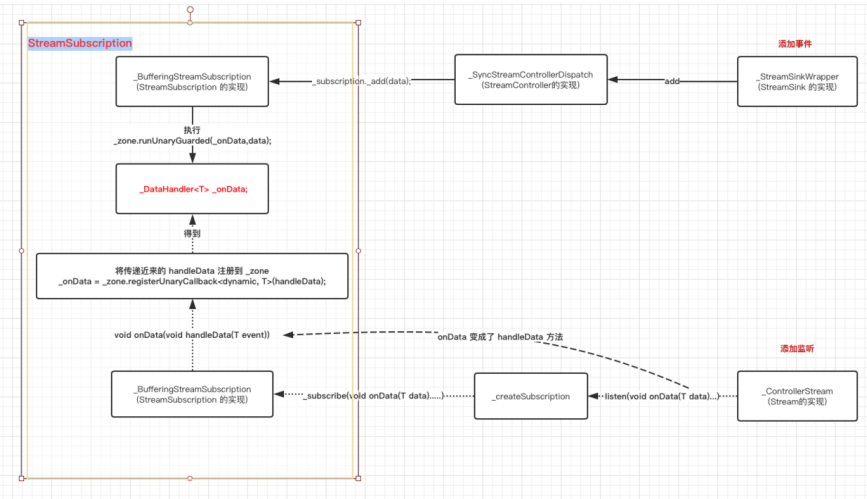
有一个事件源叫 `Stream` ，为了方便控制 `Stream` ，官方提供了使用 `StreamController` 作为管理；同时它对外提供了 `StreamSink` 对象作为事件输入口，可通过 `sink` 属性访问; 又提供 `stream` 属性提供 `Stream` 对象的监听和变换，最后得到的 `StreamSubscription` 可以管理事件的订阅。

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

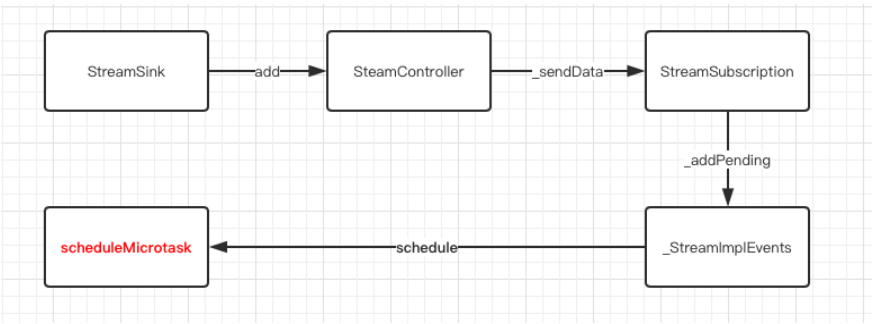
首先我们需要知道，Stream 是怎么实现异步的?

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, Funct
  startMainIsolateFunction((){
    runZoned<Future<void>>(····);
  }, null);
}

///C++ 中
if (tonic::LogIfError(tonic::DartInvokeField(
          Dart_LookupLibrary(tonic::ToDart("dart:ui")), "_
          {start_main_isolate_function, user_entrypoint_fu
    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

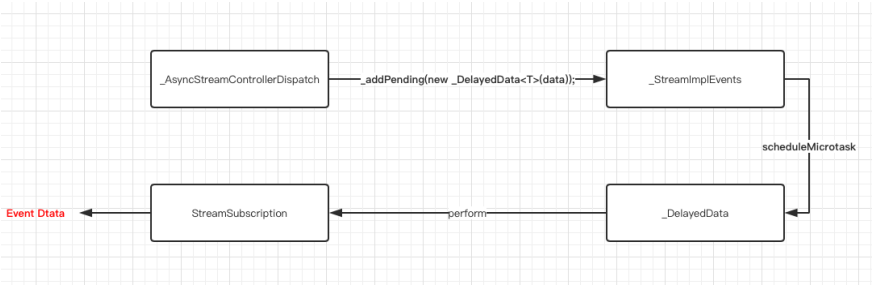
前面我们说了 `Stream` 的内部执行流程，那么同步和异步操作时又有什么区别？具体实现时怎么样的呢？

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。

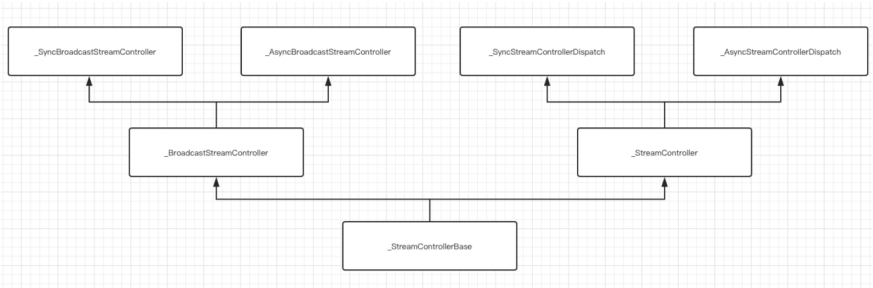
如下图， **异步执行的逻辑就是上面说过的 `scheduleMicrotask` ， 在 `_StreamImplEvents` 中 `scheduleMicrotask` 执行后，会调用 `_DelayedData` 的 `perform` ，最后通过 `_sendData` 触发 `StreamSubscription` 去回调数据 。**



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：

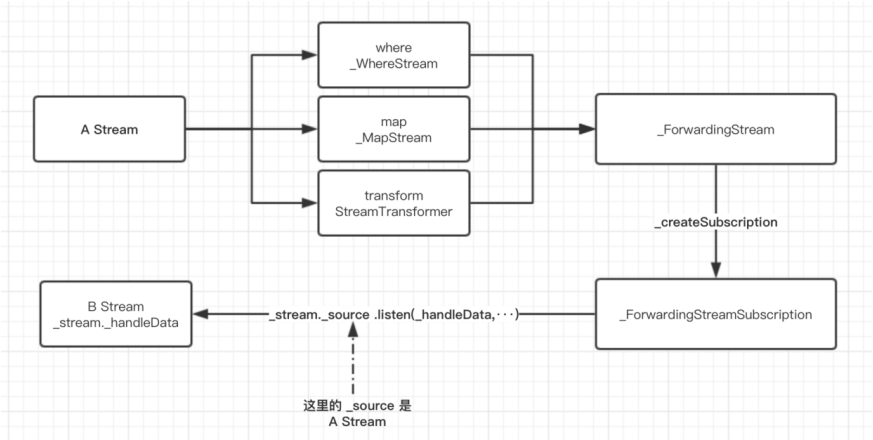
```
_forEachListener((_BufferingStreamSubscription<T> subsc
    subscription._add(data);
  });
```

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。**

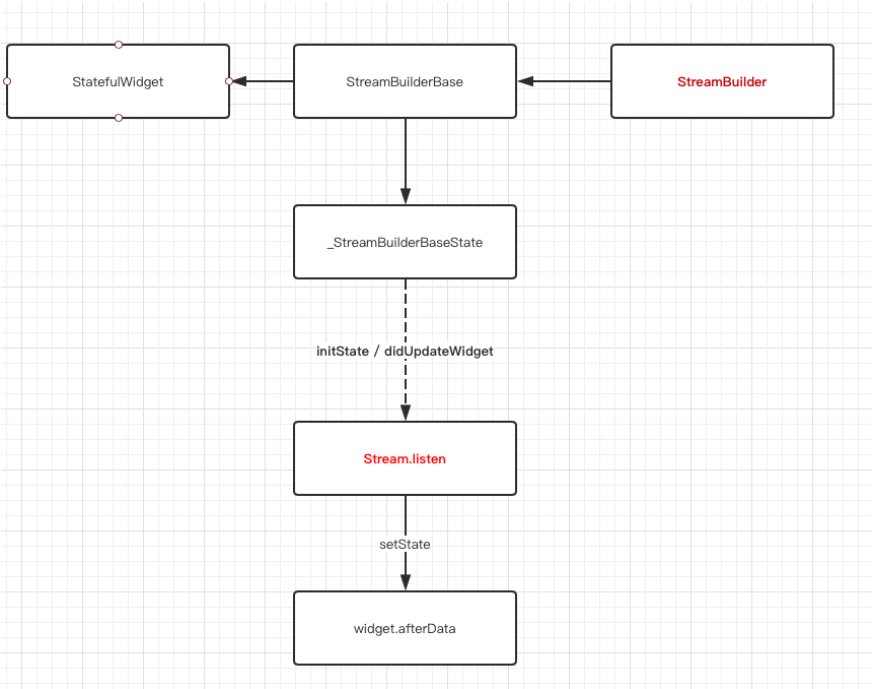


同时 `Stream` 还有转换为 `Future`，如 `firstWhere` 、`elementAt` 、 `reduce` 等操作符方法，基本都是创建一个内部 `_Future` 实例，然后再 `listen` 的回调用调用 `Future` 方法返回。

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<Str
      ///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。

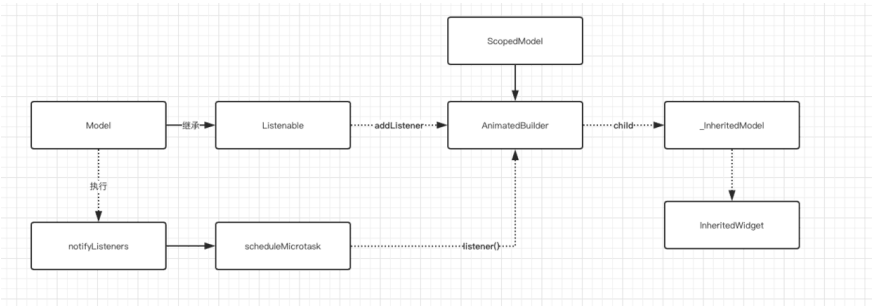
```
class ScopedPage extends StatelessWidget {
  final CountModel _model = new CountModel();

  @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(
            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。

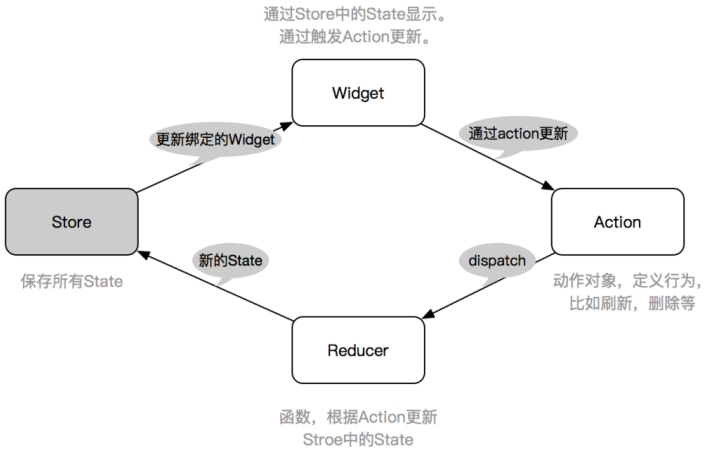
```
class _BlocPageState extends State<BlocPage> {
  final PageBloc _pageBloc = new PageBloc();
  @override
  void dispose() {
    _pageBloc.dispose();
    super.dispose();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        child: new StreamBuilder(
            initialData: 0,
            stream: _pageBloc.stream,
            builder: (context, snapShot) {
              return new Column(
                children: <Widget>[
                  new Expanded(
                      child: new Center(
                          child: new Text(snapShot.data.to$
                  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 = StreamControlle
  ///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

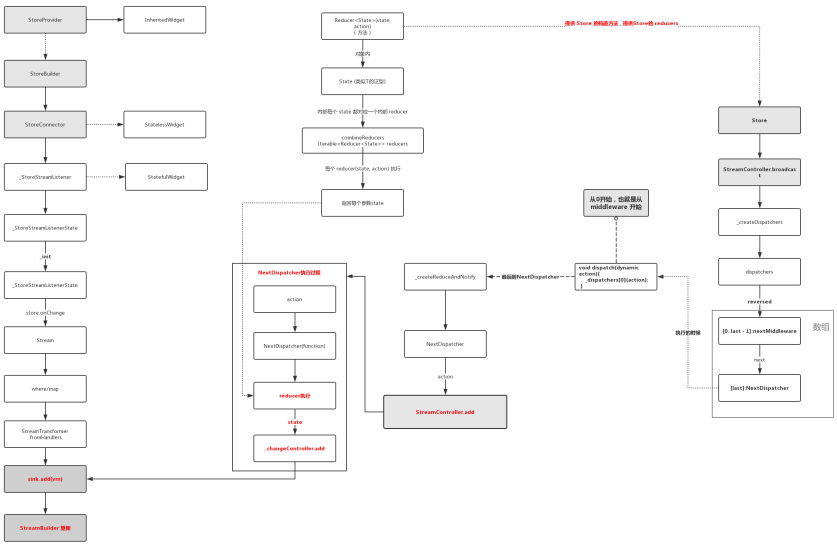
流程是如何联动起来的吧。

```
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。

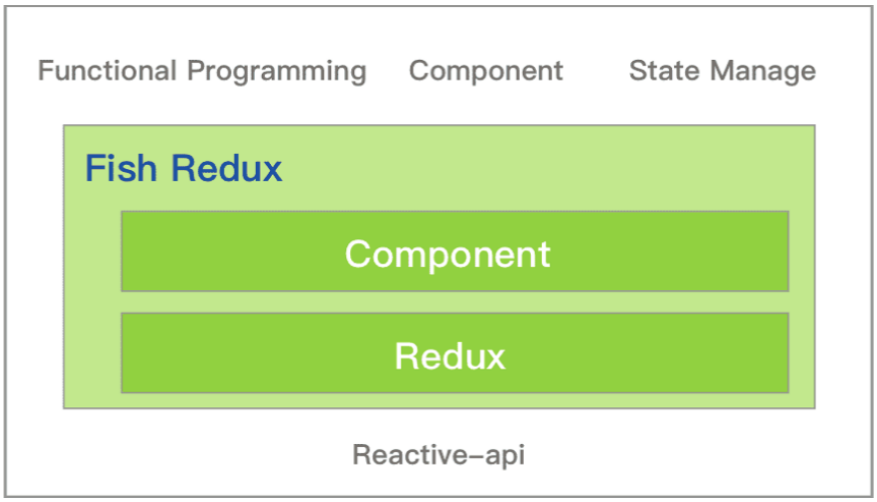
现在再对照流程图会不会清晰很多了？

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？

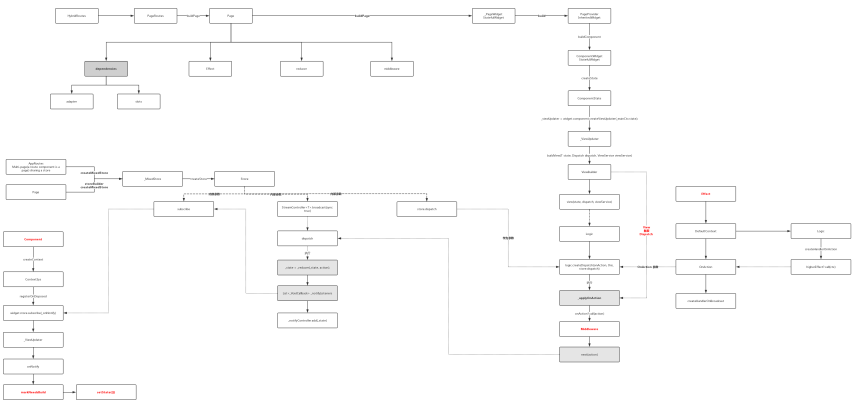
但是这带来的好处就是 **复用的颗粒度更细了，装配和功能更加的清晰。**
那这个过程是如何实现的呢？后面我们将分析这个复杂的流程。

```
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 stat
                ///然后将数据渲染到 Component
                'count-double': DoubleCountConnector() + Do
              }
          ),
          middleware: <Middleware<CountState>>[
            ///中间键打印log
            logMiddleware(tag: 'FishPage'),
          ]
  );
}

///渲染主页
Widget buildView(CountState state, Dispatch dispatch, ViewS
  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(st
          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：

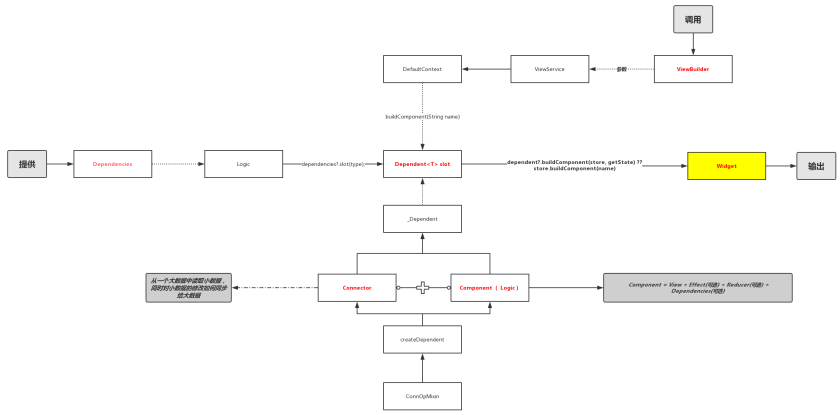
- 1、`Page` 的构建需要 `State` 、`Effect` 、`Reducer` 、`view` 、`dependencies` 、`middleware` 等参数。
- 2、`Page` 的内部 `PageProvider` 是一个 `InheritedWidget` 用户状态共享。
- 3、`Page` 内部会通过 `createMixedStore` 创建 `Store` 对象。
- 4、`Store` 对象对外提供的 `subscribe` 方法，在订阅时**会将订阅的方法添加到内部 `List<_VoidCallback> _listeners` 。**
- 5、`Store` 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 PointerSigna
      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
      hitTest(hitTestResult, event.position);
      if (event is PointerDownEvent) {
        _hitTests[event.pointer] = hitTestResult;
      }
    } else if (event is PointerUpEvent || event is Pointer
      ///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 pos
    if (_size.contains(position)) {
      if (hitTestChildren(result, position: position) ||  hi
        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 hit
     ///如果没有碰撞结果，那么通过 `pointerRouter.route` 将事
  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。**

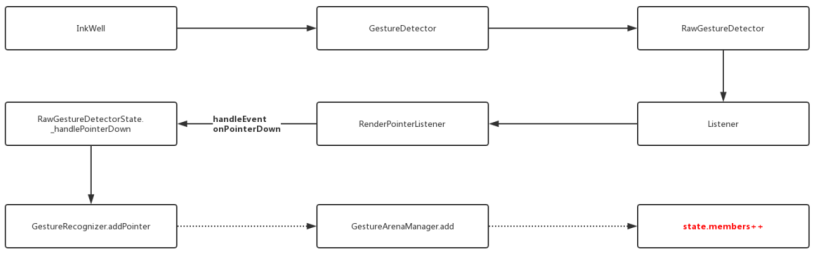
好了，知道这些概念之后我们开始分析流程，我们知道 `GestureBinding` 在 `dispatchEvent` 时会先判断是否有 `HitTestResult` 是否有结果，一般情况下是存在的，所以直接执行循环 `entry.target.handleEvent` 。

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 bee
  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, st
  } 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：

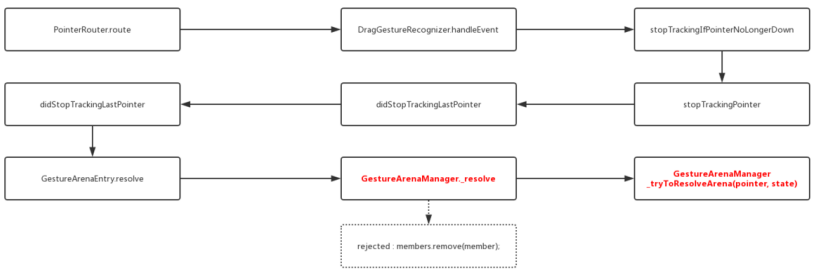
在竞技场竞争失败的成员会被移出竞技场，移除后就没办法参加后面事件的竞技了，比如 `TapGestureRecognizer` 在接受到 `PointerMoveEvent` 事件时就会直接 `rejected` ，并触发 `rejectGesture` ，之后定时器会被关闭，并且触发 `onTapCancel` ，然后重置标志位.

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：

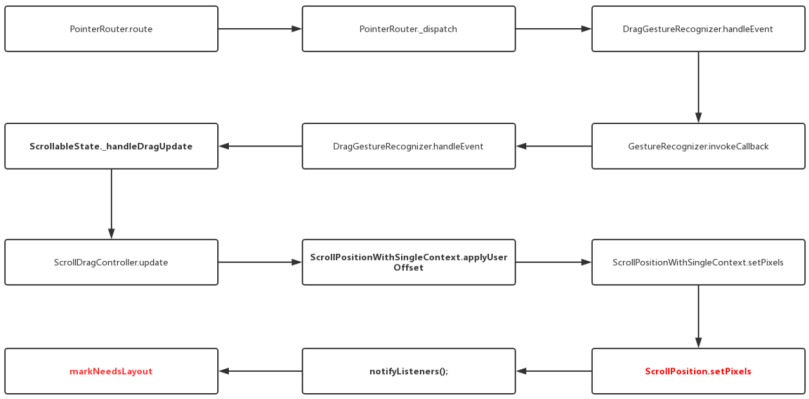
```
bool get _hasSufficientPendingDragDeltaToAccept => _pending
```

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?

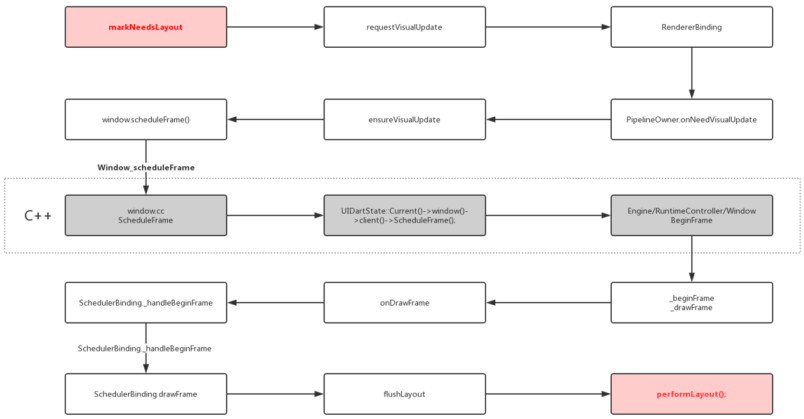
`ScrollPositionWithSingleContext` 其实就是这个滑动的关键，它其实就是 `ScrollPosition` 的子类，而 `ScrollPosition` 又是 `ViewportOffset` 的子类，而 `ViewportOffset` 又是一个 `ChangeNotifier`，出现如下关系：

> 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：

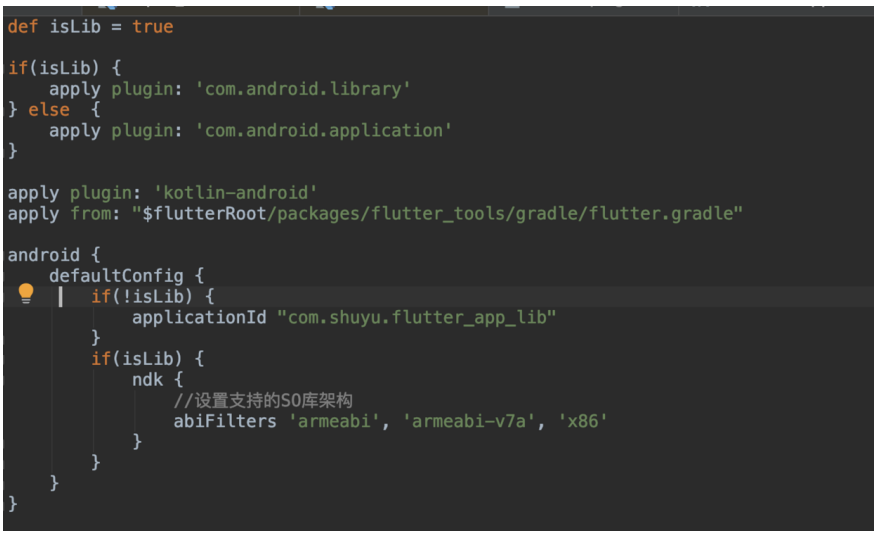
- 1、将 `Flutter` 整体框架依赖和打包脚本都集成到主项目中。
- 2、以 `aar` 的完整库集成形式添加到主项目。

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。**

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
```

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 。

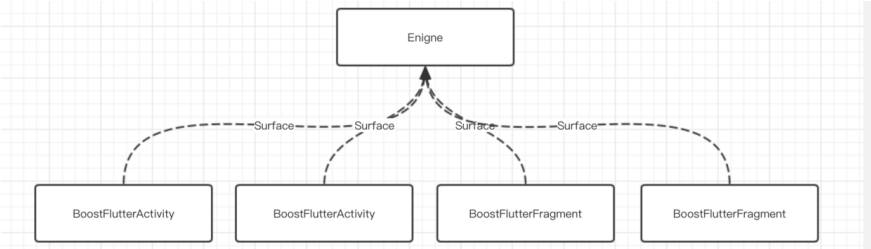
四、堆栈

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

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。**

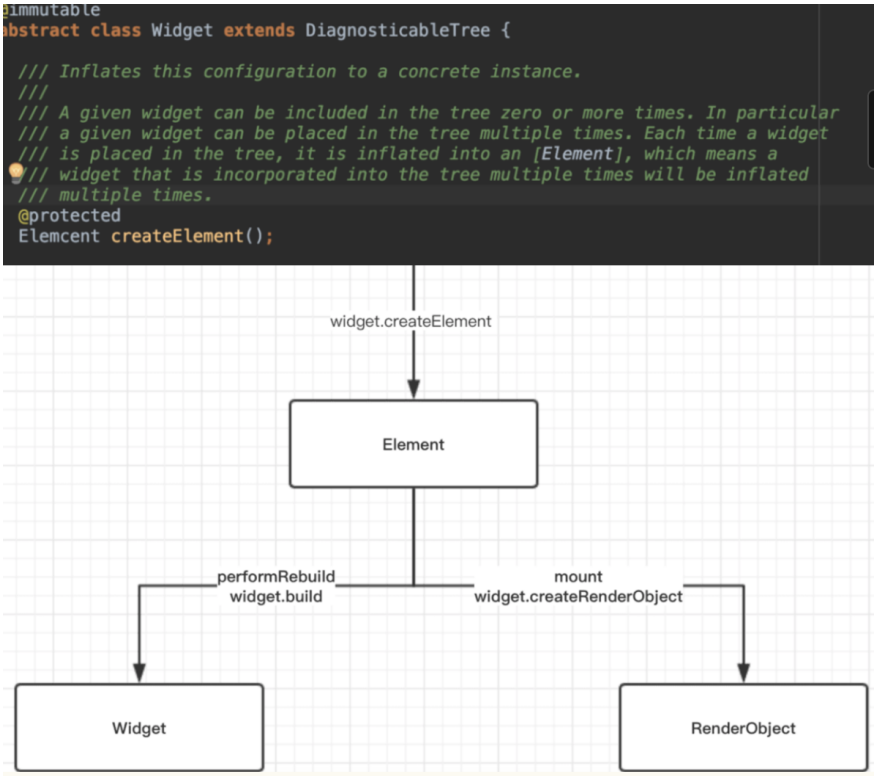
在这个基础上， **`StatefulWidget` 的 `State` 帮我们实现了在 `Widget` 的跨帧绘制** ，也就是在每次 `Widget` 重绘的时候，通过 `State` 重新赋予 `Widget` 需要的绘制信息。

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。

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?

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 shadowTh
  final _InheritedTheme inheritedTheme = context.inheritF
  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 targetT
  /// 在共享 map _inheritedWidgets 中查找
  final InheritedElement ancestor = _inheritedWidgets ==
  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 ances
  _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: (_) => ProviderMode
      ],
      child: Scaffold(
        appBar: AppBar(
          title: LayoutBuilder(
            builder: (BuildContext context, BoxConstraints
              var counter =  Provider.of<ProviderModel>(cor
              return new Text("Provider ${counter.count.to$
            },
          )
        ),
        body: CountWidget(),
      ),
    );
  }
}

class CountWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<ProviderModel>(builder: (context, count
      return new Column(
        children: <Widget>[
          new Expanded(child: new Center(child: new Text(co
          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。

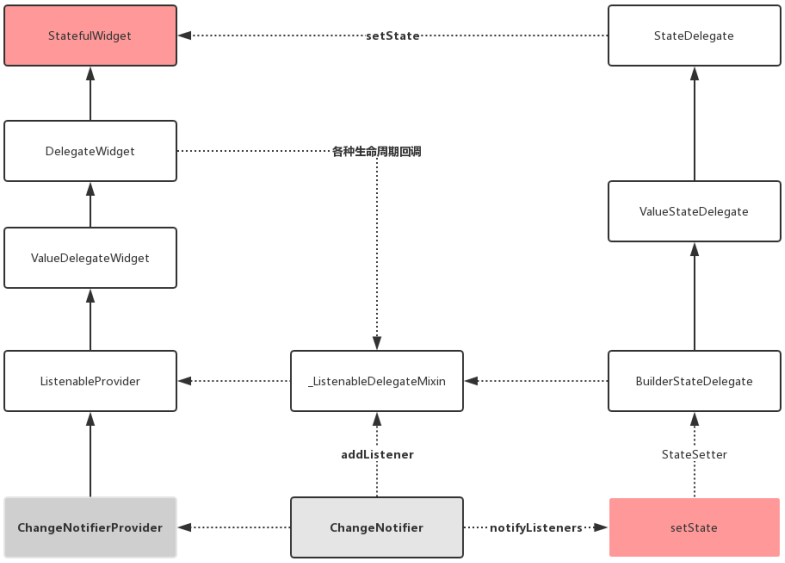
接着我们逐个分析

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 [k
/// 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, Widg

  @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

              var counter =  Provider.of<ProviderModel>(cor
              return new Text("Provider ${counter.count.to
            }
```

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

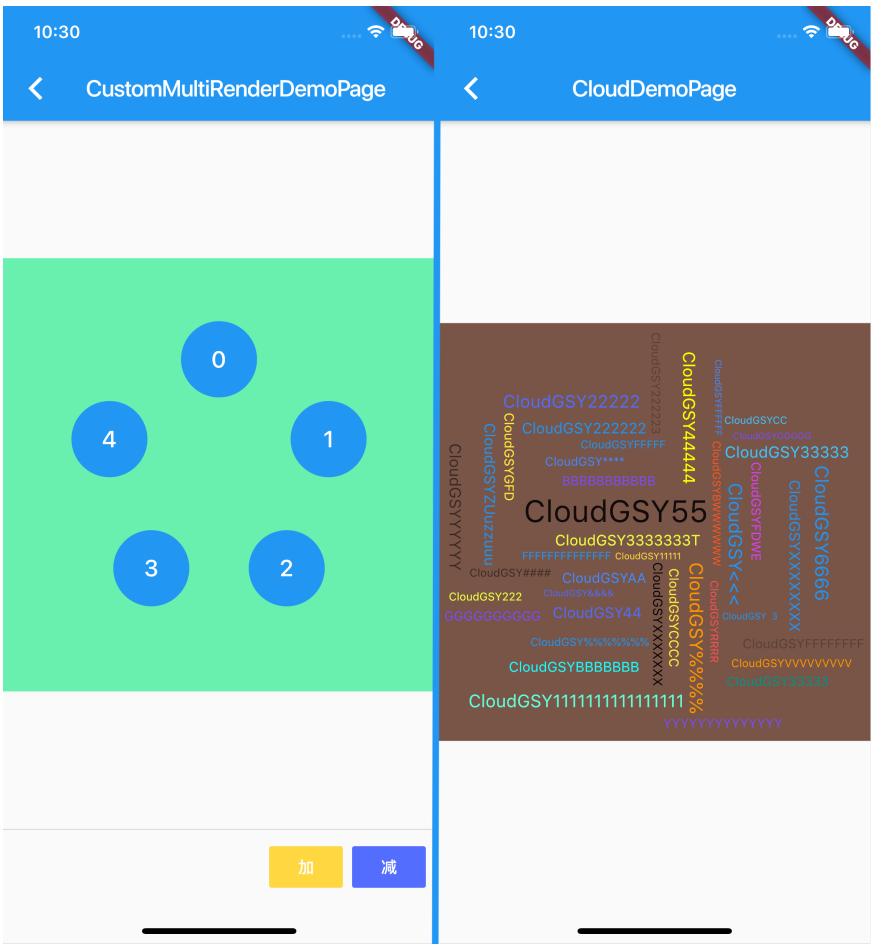
完整开源项目推荐：

- 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



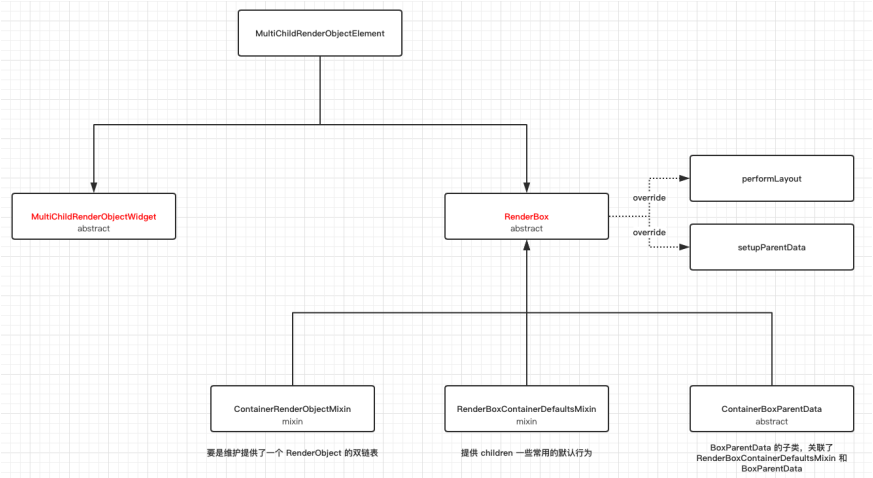
一、前言

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(Text

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

    /// 触摸碰撞测试
    bool defaultHitTestChildren(BoxHitTestResult result, {

    /// 默认绘制
    void defaultPaint(PaintingContext context, Offset offse

    /// 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 Ren
```

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 ins
    /// ContainerRenderObjectMixin<RenderObject, ContainerP
    void insertChildRenderObject(RenderObject child, Elemer
    void moveChildRenderObject(RenderObject child, dynamic
    void removeChildRenderObject(RenderObject child)

    /// visitChildren 是通过 Element 中的 ElementVisitor 去迭
    /// 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
    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。

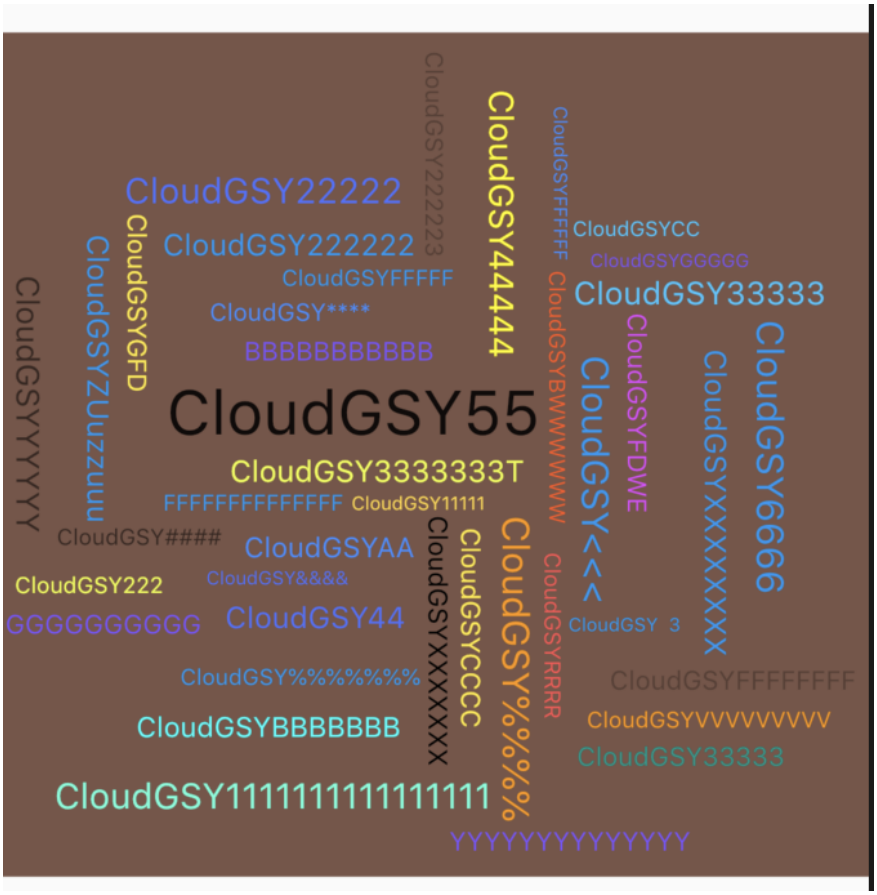
了解几个关键类之后，我们看一般情况下，实现自定义布局的简化流程是：

- 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<
  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, RenderCloudPa
        RenderBoxContainerDefaultsMixin<RenderBox, RenderC
  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.p

      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 + rotatio
        var y = rY * angleRadius * math.sin(angle + rotatio
        var position = Offset(x, y);

        ///计算得到绝对偏移
        var childOffset = position - Alignment.center.along

        ++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(previousChild

      ///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.p
      childParentData.offset += transCenter;
      child = childParentData.nextSibling;
    }

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

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。

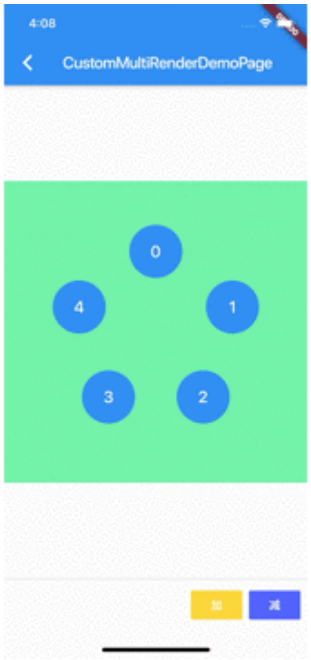
最后通过 `CloudWidget` 加载我们的 `RenderCloudWidget` 即可， 当然完整代码还需要结合 `FittedBox` 与 `RotatedBox` 简化完成，具体可见： GSYFlutterDemo

```
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 o
```

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.leng
            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)
}
```

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：

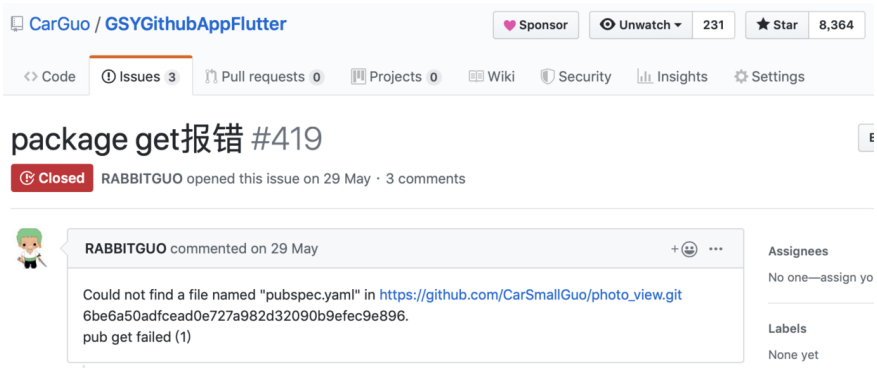
> Flutter 完整实战实战系列文章专栏
>
> Flutter 番外的世界系列文章专栏

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。**

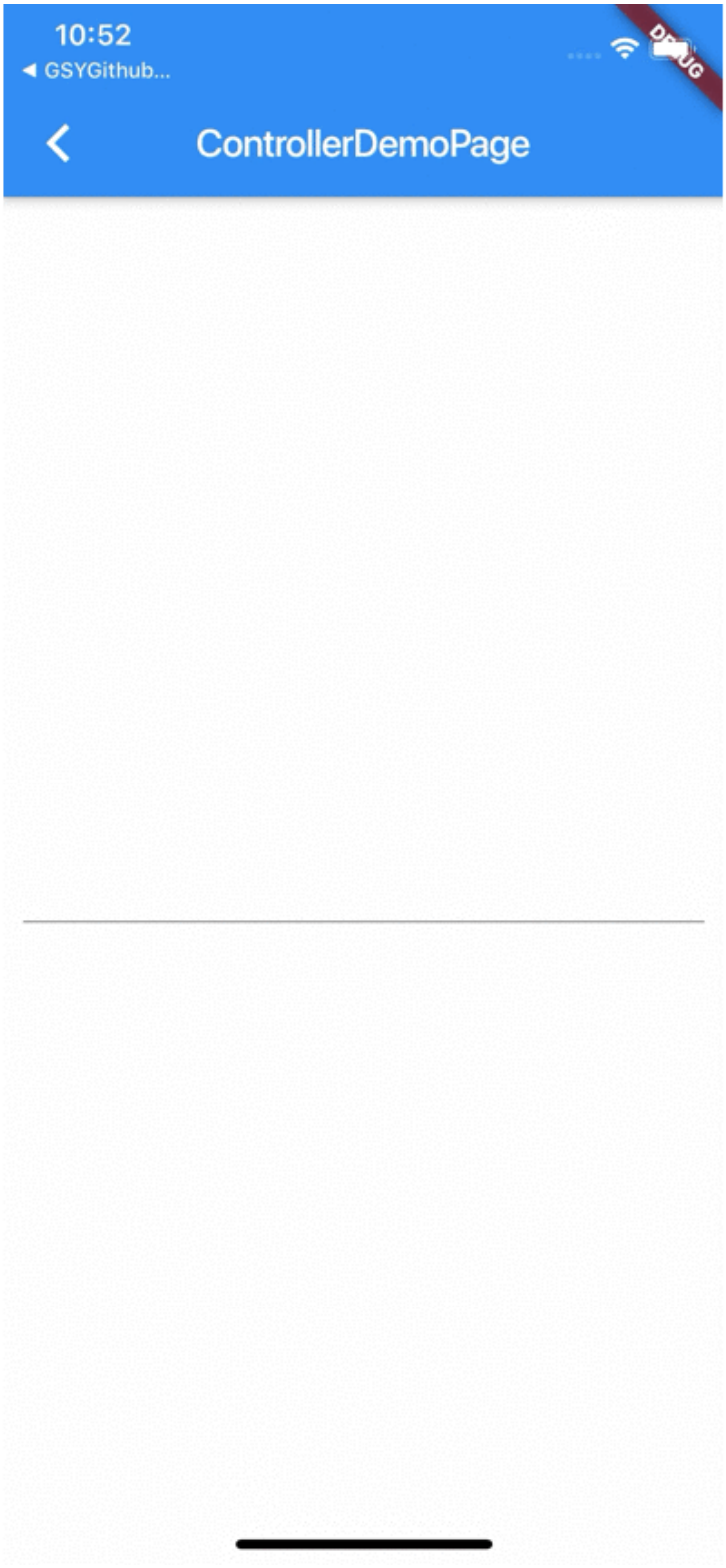
所以你需要清除掉 `.pub_cache` 内的 `git` 的异常目录，然后最好清除掉项目下的 `pubspec.lock` ，之后重新执行 `flutter packages get` 。

> `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：



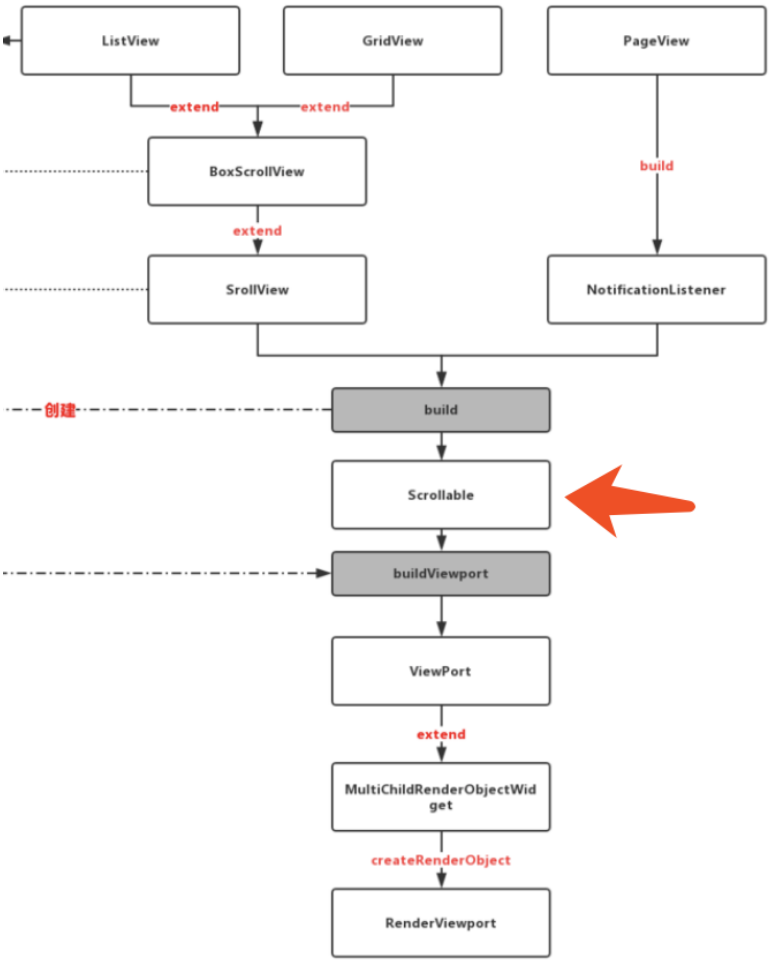
弹出键盘时输入成功后，收起键盘时输入的内容消失了！ 这是因为键盘的弹出和收起都会触发页面 `build` ，而在 `controller` 为 `null` 时，每次赋值的 `TextEditingController` 会导致 `TextField` 的 `TextEditingValue` 重置。

```dart
@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
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。

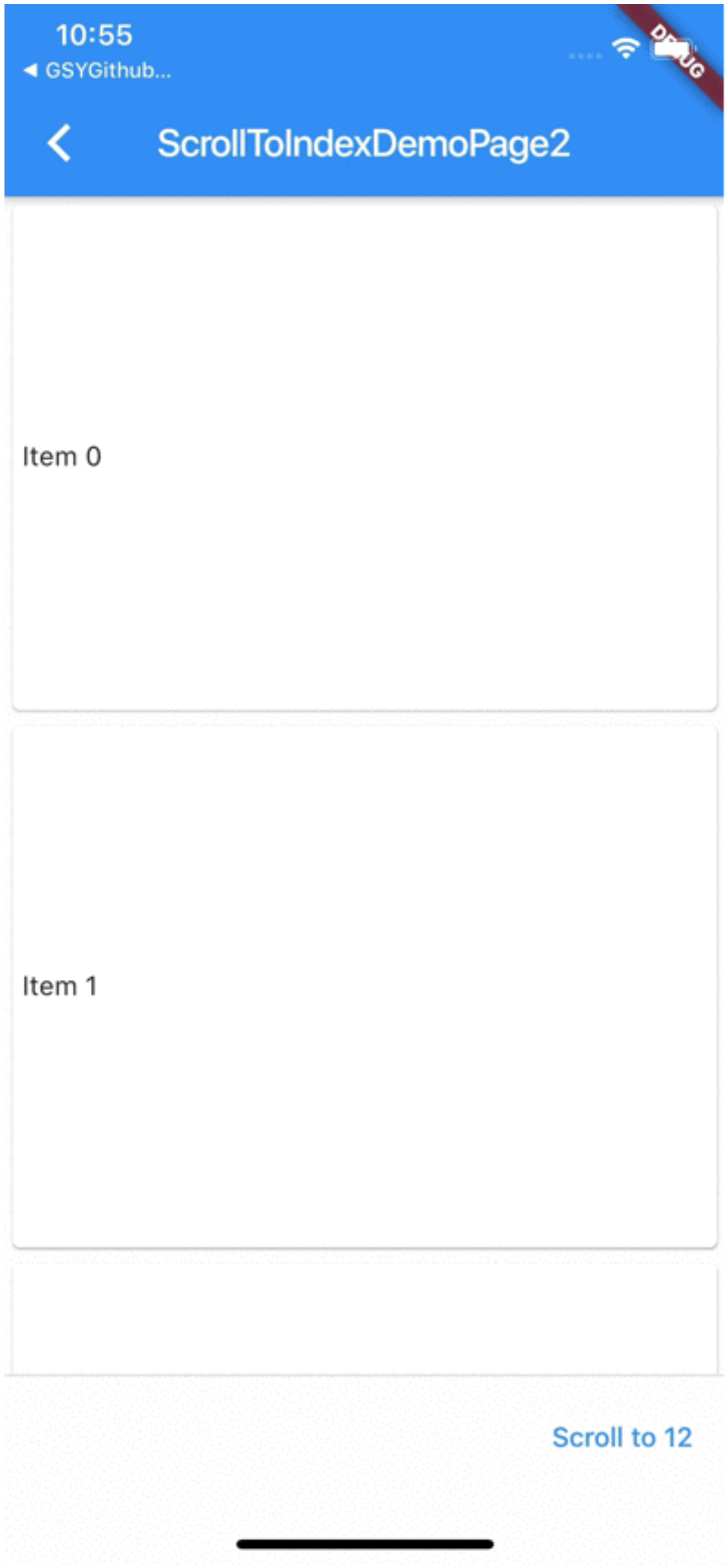
```
class BlurDemoPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: new Container(
        child: Stack(
          children: <Widget>[
            Positioned(
              top: 0,
              bottom: 0,
              left: 0,
              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, s
                    child: new Row(
                      mainAxisSize: MainAxisSize.max,
                      crossAxisAlignment: CrossAxisAlignmen
                      mainAxisAlignment: MainAxisAlignment.
                      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];

  ///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:

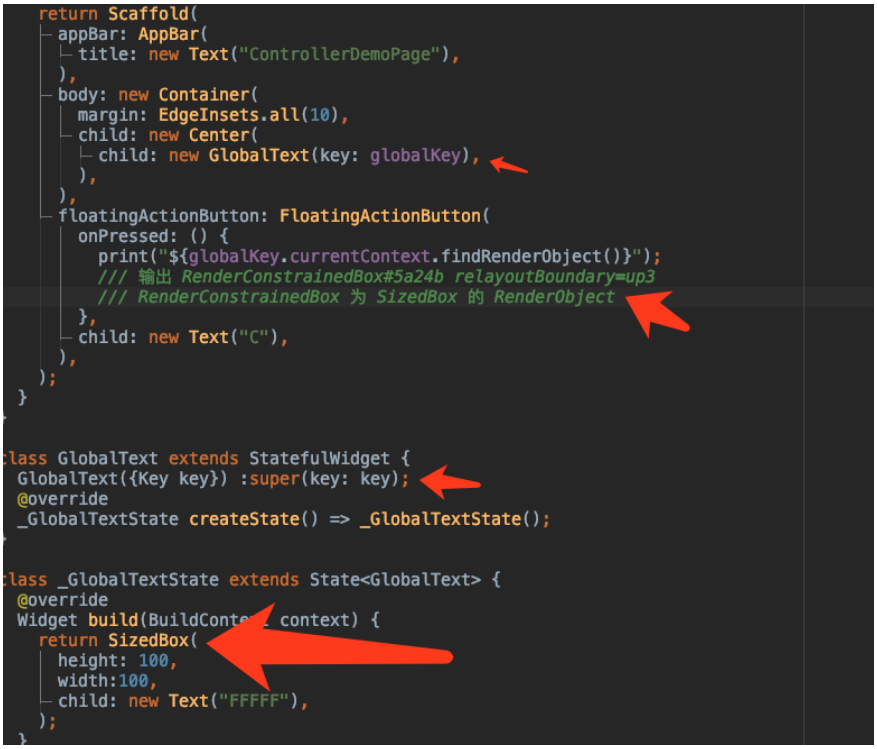
- `Text` 、 `Sliver` 、 `ListTile` 等都是属于渲染 Widget，其内部主要是 `RenderObjectElement` 。
- `StatelessWidget` / `StatefulWidget` 等属于容器 Widget，其内部使用的是 `ComponentElement` ， **`ComponentElement` 本身是不存在 `RenderObject` 的。**

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()` 。

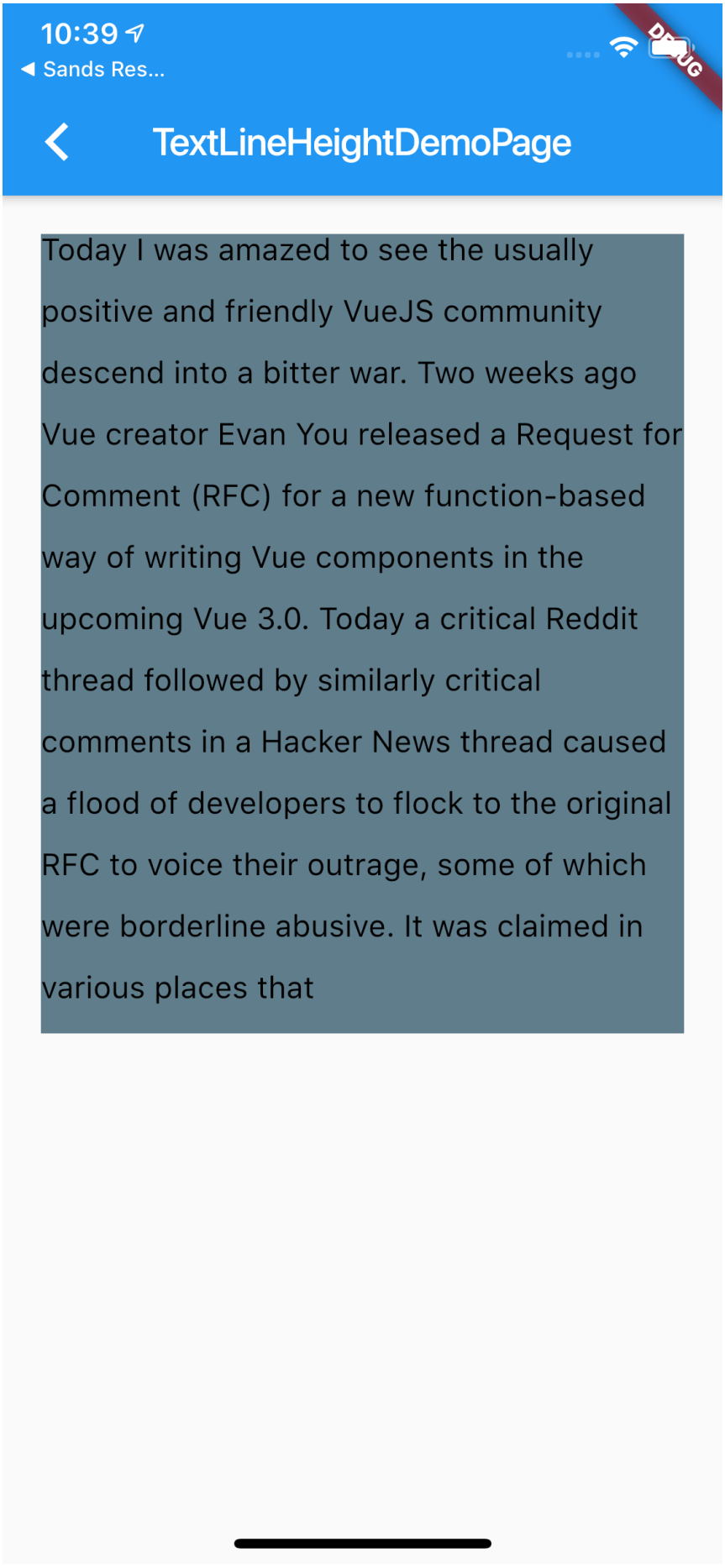
```
/// The render object at (or below) this location in the tree.
///
/// If this object is a [RenderObjectElement], the render object is the one at
/// this location in the tree. Otherwise, this getter will walk down the tree
/// until it finds a [RenderObjectElement].
RenderObject get renderObject {
  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` 。

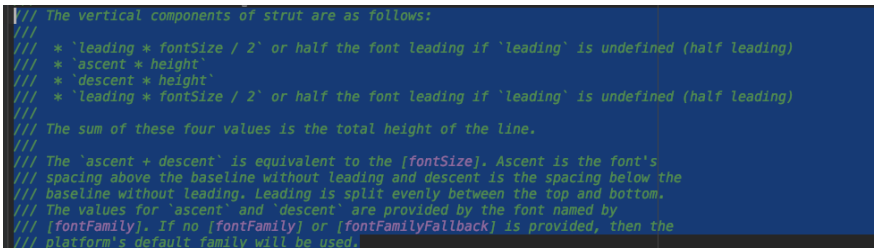
所以如下代码所示， `print("${globalKey.currentContext.findRenderObject()}");` 最终输出了 `SizedBox` 的 `RenderObject` 。


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(BuildConte context) {
return SizedBox(
height: 100,
width:100,
child: new Text("FFFFF"),
);

7、行间距

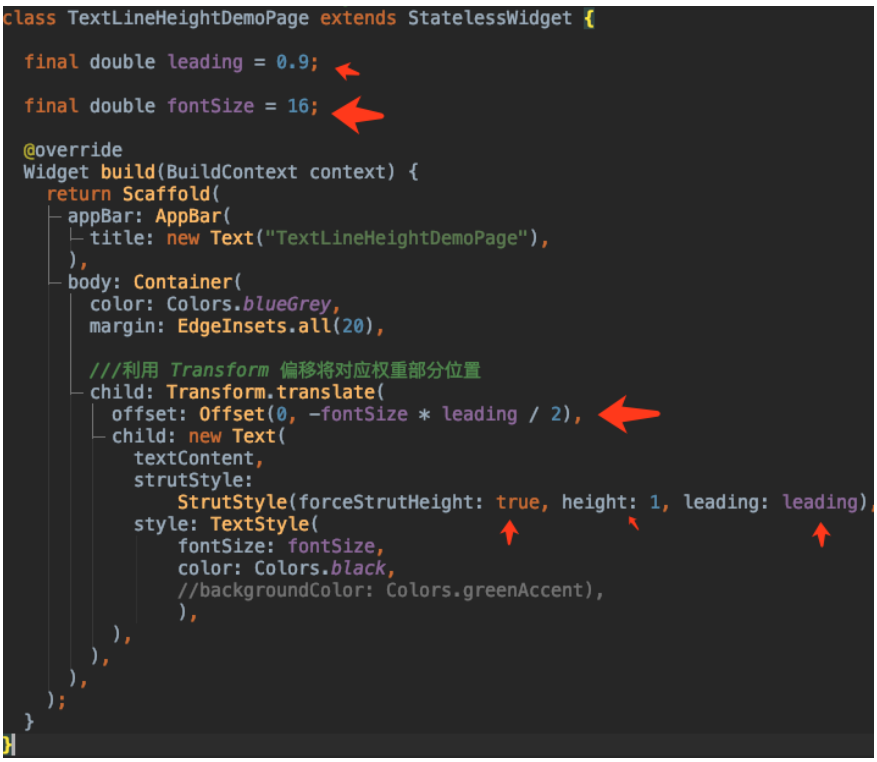

10:39
Sands Res...
TextLineHeightDemoPage
Today I was amazed to see the usually positive and friendly VueJS community descend into a bitter war. Two weeks ago Vue creator Evan You released a Request for Comment (RFC) for a new function-based way of writing Vue components in the upcoming Vue 3.0. Today 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),
                ),
          ),
        ),
      ),
    );
  }
}
```

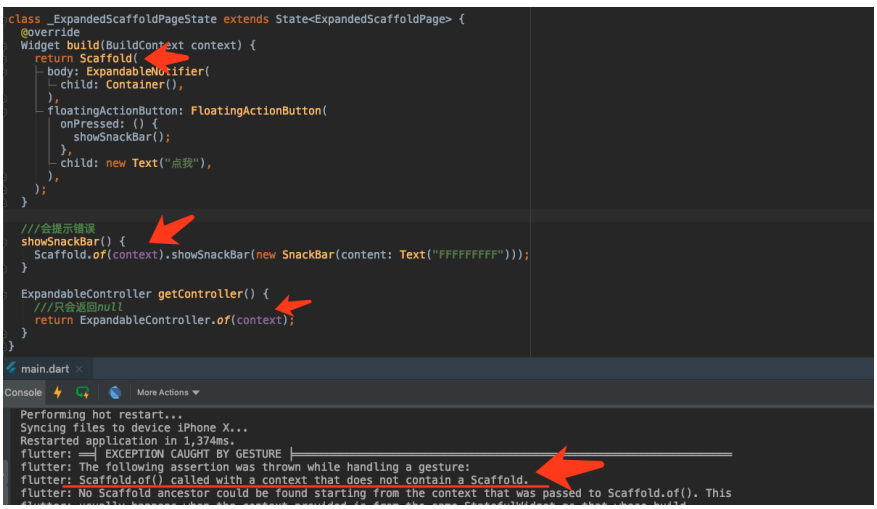
> 这里额外提一点，可以通过父节点使用 `DefaultTextStyle` 来实现局部样式的共享哦。

8、Builder

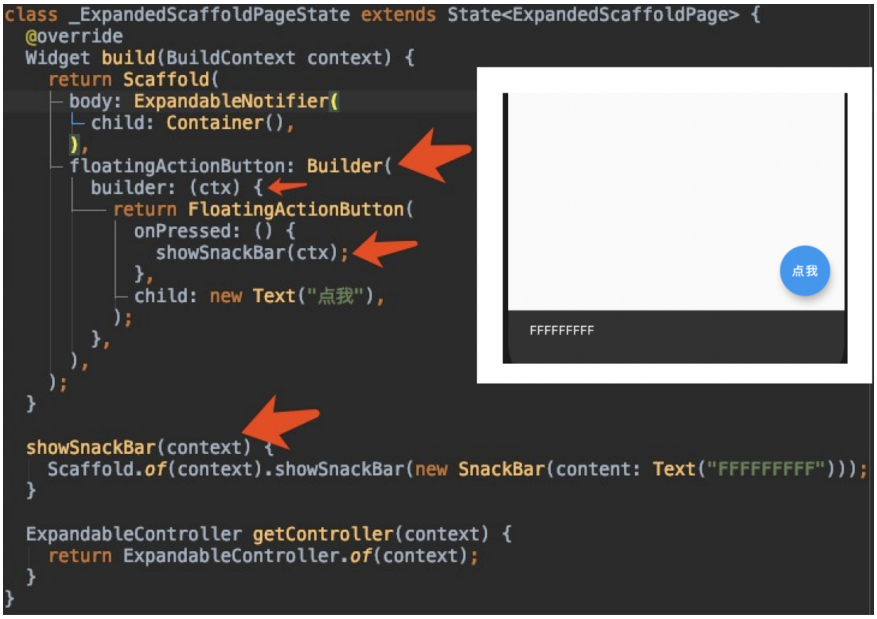
```dart
class Builder extends StatelessWidget {
  /// Creates a widget that delegates its build to a callback.
  ///
  /// The [builder] argument must not be null.
  const Builder({
    Key key,
    @required this.builder,
  }) : assert(builder != null),
       super(key: key);

  /// 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?

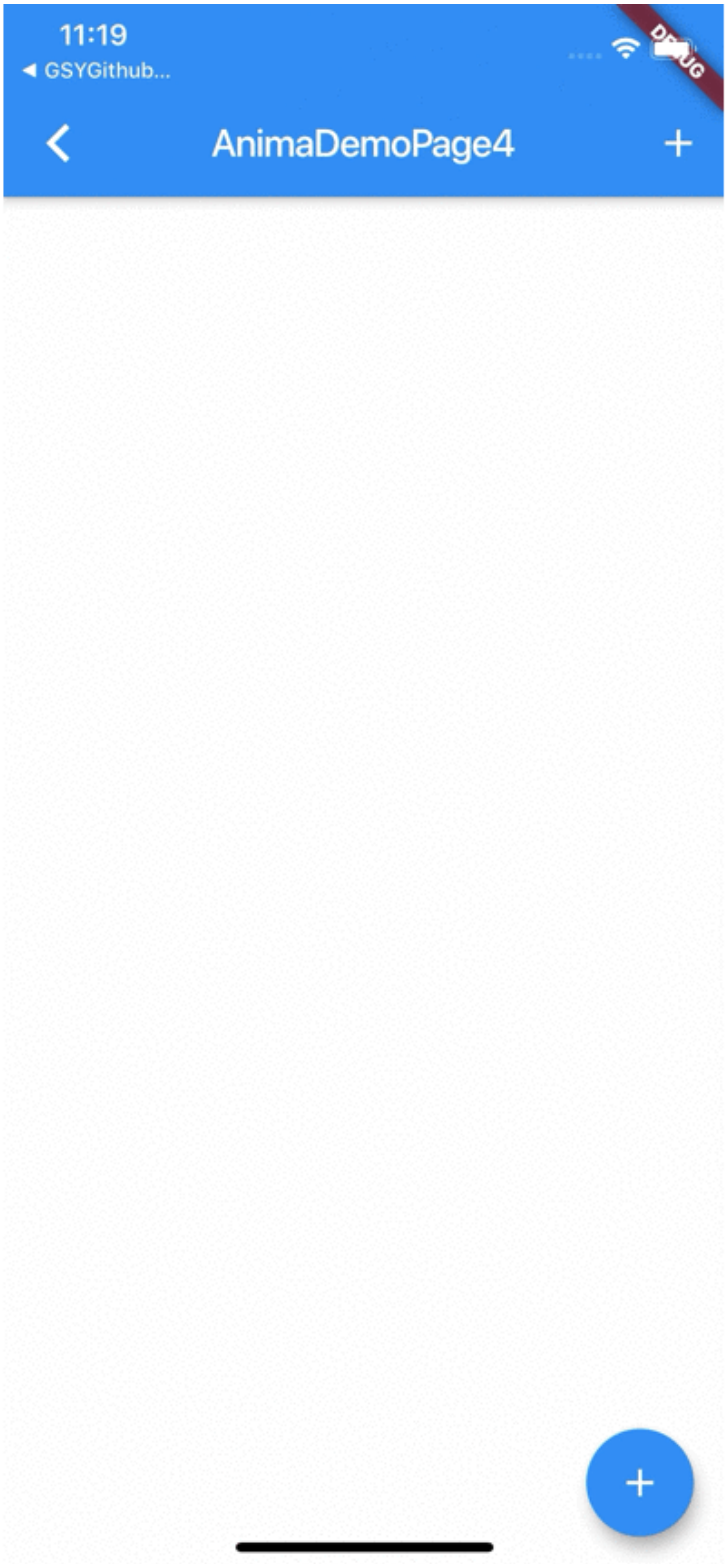
如下图所示，相信一些 Flutter 开发者在使用 `Scaffold.of(context).showSnackBar(snackbar)` 时，可能 遇到过如下错误，这是因为传入的 `context` 属于错误节点导致的，**因为此处传入的 `context` 并不能找到页面所在的 `Scaffold` 节点。**



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。



9、快速实现动画切换效果



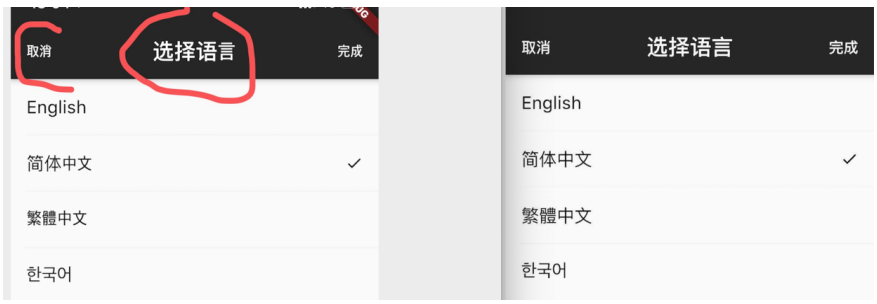
要实现如上图所示动画效果，在 Flutter 中提供了 `AnimatedSwitcher` 封装简易实现。

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.accentTex
      primaryTextTheme: confirmTextTheme(themeData.primary
    );
    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: ["PingFar
  }

  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) => loa

  @override
  bool shouldReload(FallbackCupertinoLocalisationsDelegate
}

class CustomZhCupertinoLocalizations extends DefaultCuperti
  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 DefaultCuperti
  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(L
  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>(localiza
}
```

> 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

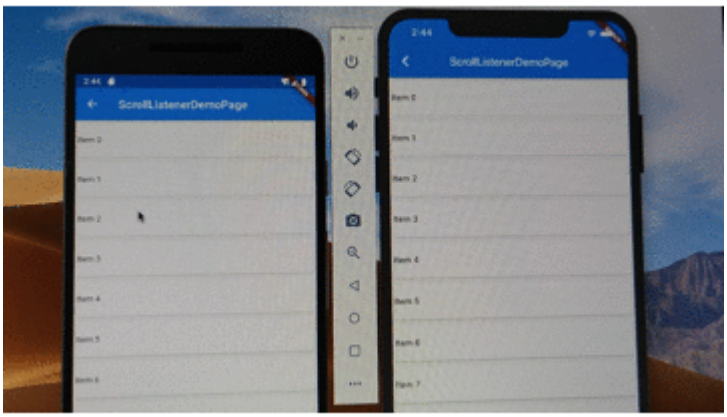
作为系列文章的第十八篇，本篇将通过 ScrollPhysics 和 Simulation ，带你深入走进 Flutter 的滑动新世界，为你打开 Flutter 滑动操作的另一扇窗。

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

如下图所示是Flutter 默认的可滑动 `Widget` 效果，在 Android 和 iOS 上出现了不同的 **滑动速度与边缘拖拽效果** ，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

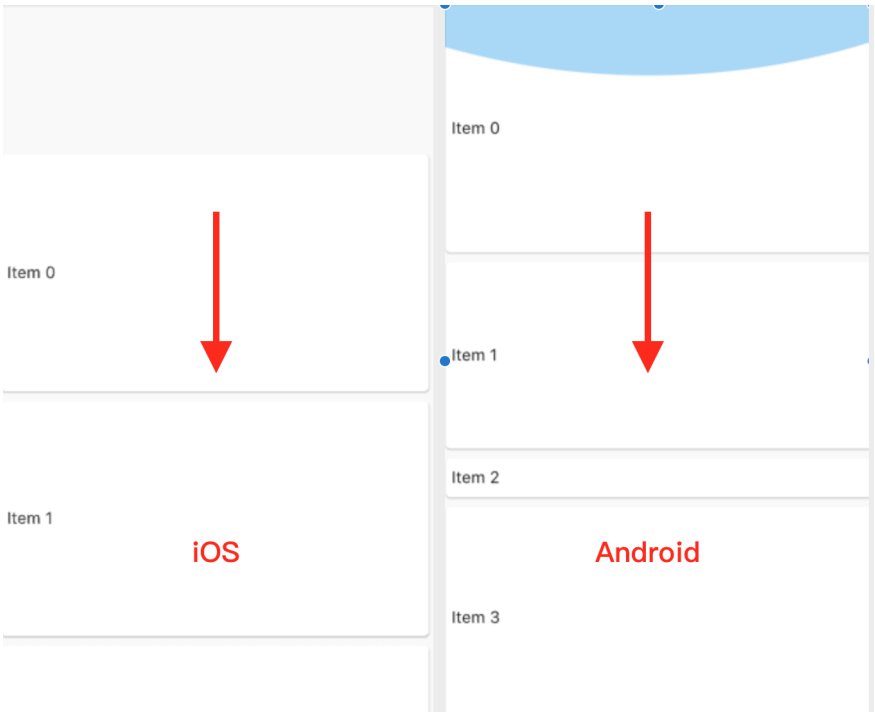
首先介绍 `ScrollPhysics` ，在 Flutter 官方的介绍中， `ScrollPhysics` 的作用是 **确定可滚动控件的物理特性，** 常见的有以下四大金刚：

- **`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 AlwaysScrollableScrollPhys
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` 控件中，在 Android 和 iOS 平台的滚动和边界拖拽效果，会出现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` 对触摸进行响应从而进行滑动的。

如下代码所示，在 `Scrollable` 的 **`_updatePosition`** 方法内，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 trigge
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,
    ?? ScrollPositionWithSingleContext(physics: _physics,
  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**，那么如上图所示，**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 c
    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。**

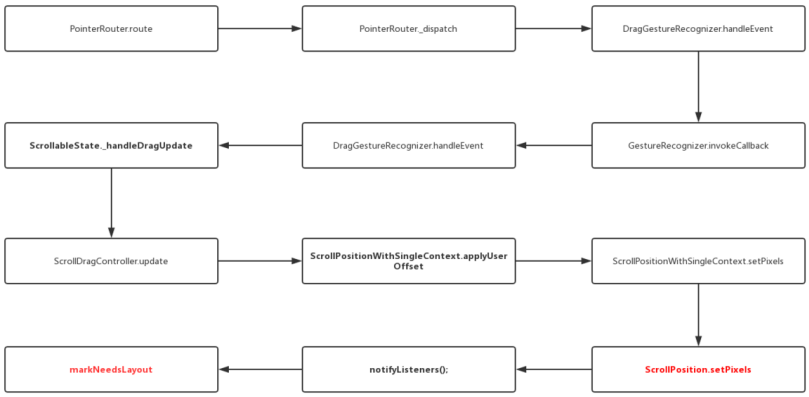
> ps： 我们可以通过实现自己的 `ScrollBehavior` ， 实现自定义的拖拽溢出效果。

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：

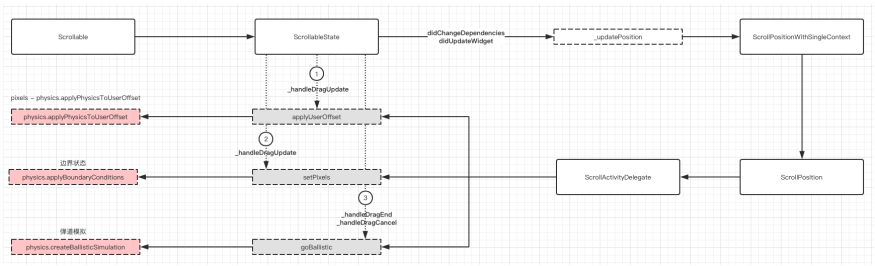
```
/// [position] 当前的位置，[offset] 用户拖拽距离
/// 将用户拖拽距离 offset 转为需要移动的 pixels
double applyPhysicsToUserOffset(ScrollMetrics position, dou

/// 返回 overscroll ，如果返回 0 ，overscroll 就一直是0
/// 返回边界条件
double applyBoundaryConditions(ScrollMetrics position, doub

///创建一个滚动的模拟器
Simulation createBallisticSimulation(ScrollMetrics positior

///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, d
    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 *

@override
 double applyPhysicsToUserOffset(ScrollMetrics position, d
   assert(offset != 0.0);
   assert(position.minScrollExtent <= position.maxScrollEx

   if (!position.outOfRange)
     return offset;

   final double overscrollPastStart = math.max(position.mi
   final double overscrollPastEnd = math.max(position.pixe
   final double overscrollPast = math.max(overscrollPastSt
   final bool easing = (overscrollPastStart > 0.0 && offse
       || (overscrollPastEnd > 0.0 && offset > 0.0);

   final double friction = easing
       // Apply less resistance when easing the overscroll
       ? frictionFactor((overscrollPast - offset.abs()) /
       : frictionFactor(overscrollPast / position.viewport
   final double direction = offset.sign;

   return direction * _applyFriction(overscrollPast, offse
 }
```

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, do
  if (value < position.pixels && position.pixels <= posit
    return value - position.pixels;
  if (position.maxScrollExtent <= position.pixels && posi
    return value - position.pixels;
  if (value < position.minScrollExtent && position.minScr
    return value - position.minScrollExtent;
  if (position.pixels < position.maxScrollExtent && posit
    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。

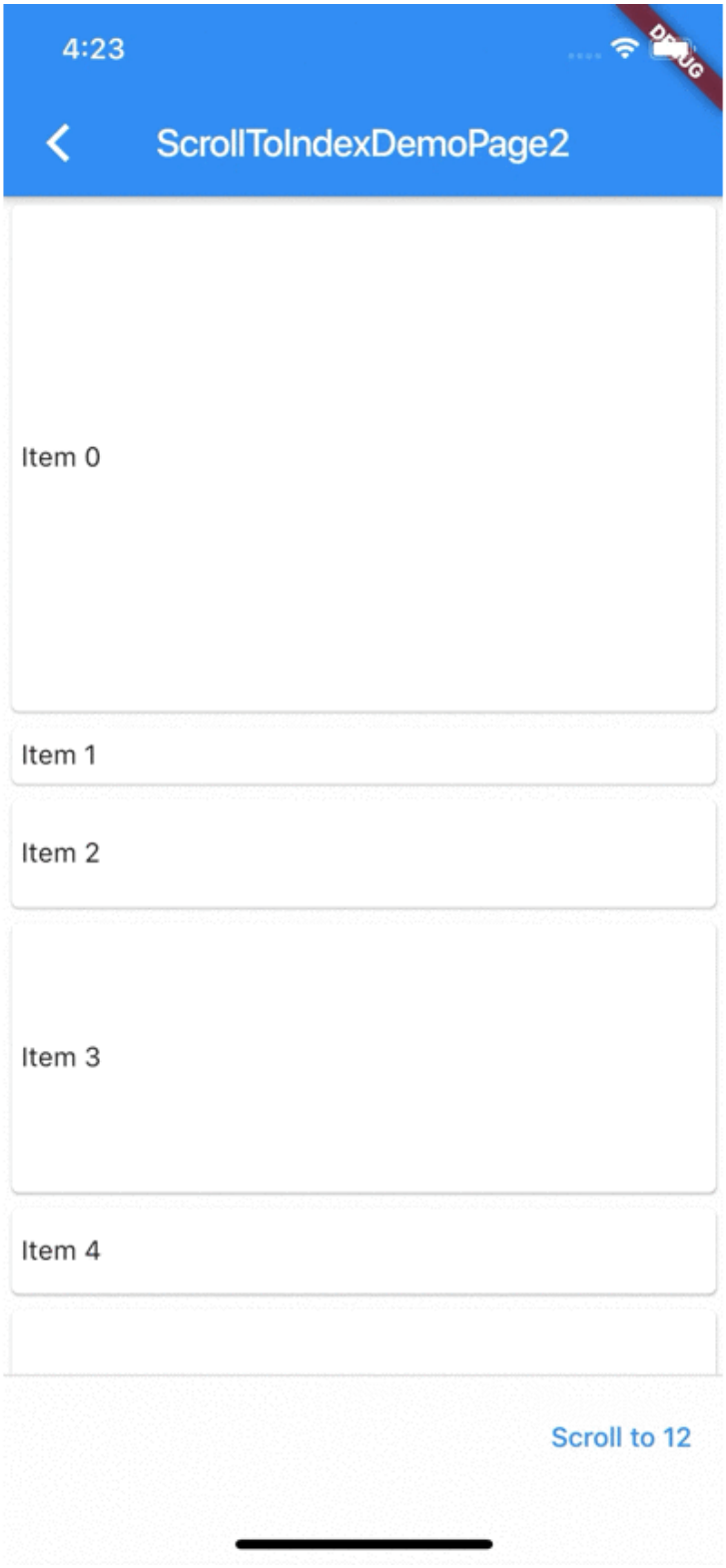
`BouncingScrollPhysics` 中 `applyBoundaryConditions` 直接返回 0，**也就是达到 0 是就边界，过了 0 的就是边界外的拖拽效果了。**

```
@override
double applyBoundaryConditions(ScrollMetrics position, do
```

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。**

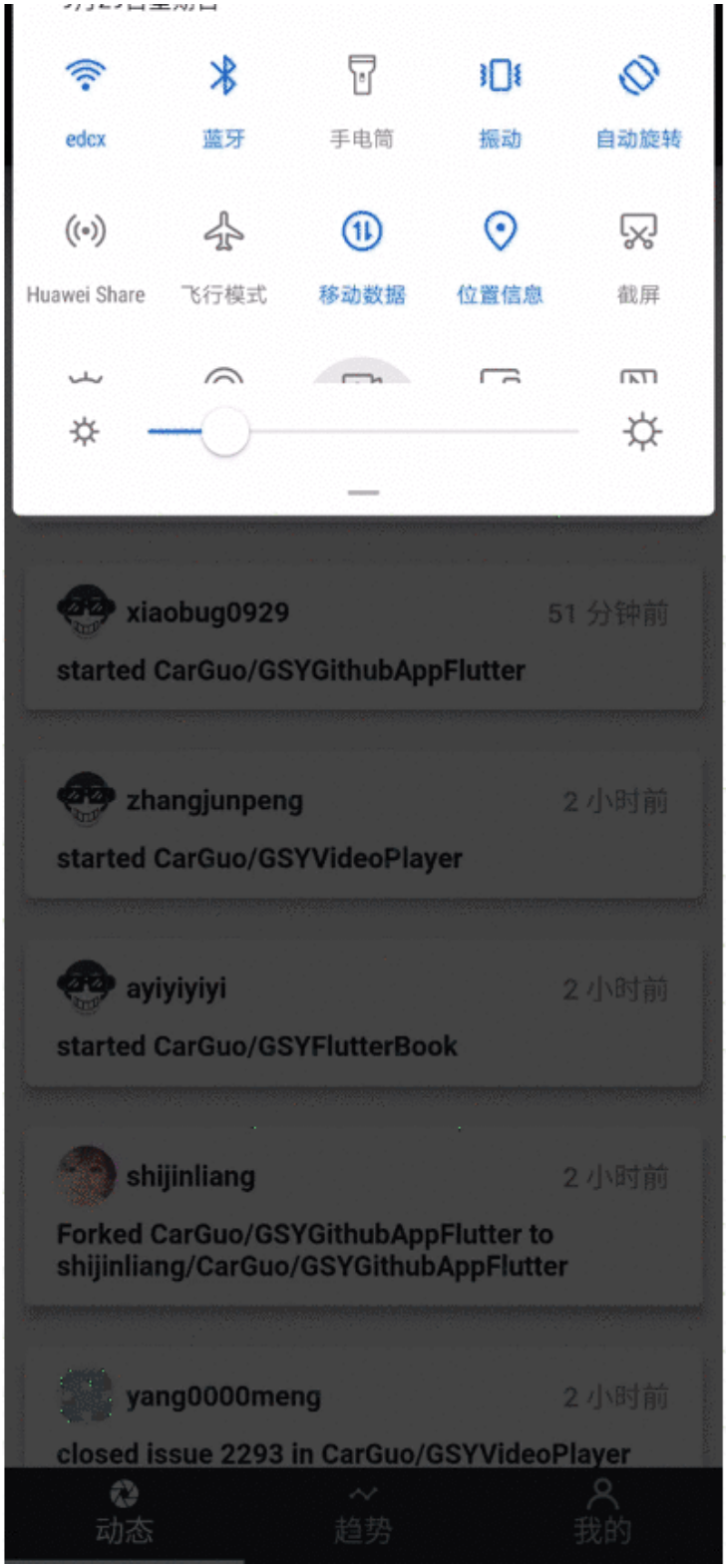
```
@override
  Simulation createBallisticSimulation(
      ScrollMetrics position, double velocity) {
    final Tolerance tolerance = this.tolerance;
    if (position.outOfRange) {
      double end;
      if (position.pixels > position.maxScrollExtent)
        end = position.maxScrollExtent;
      if (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.maxSc
      return null;
    if (velocity < 0.0 && position.pixels <= position.minSc
      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 positi
  final Tolerance tolerance = this.tolerance;
  if (velocity.abs() >= tolerance.velocity || position.ou
    return BouncingScrollSimulation(
      spring: spring,
      position: position.pixels,
      velocity: velocity * 0.91, // TODO(abarth): We shou
      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。

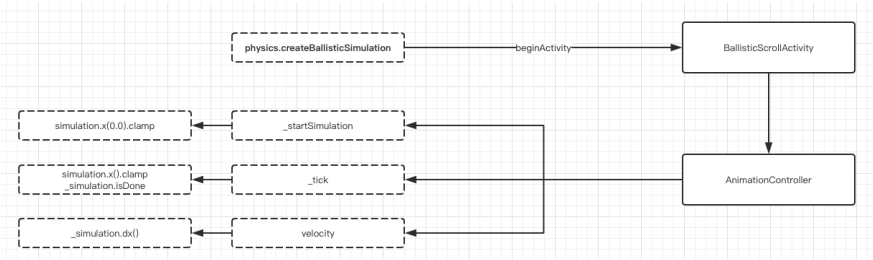
如下图所示，**第一页面的 `ScrollSpringSimulation` 在停止滚动前是有一定的减速效果的；而第二个页面 `ClampingScrollSimulation` 是直接快速滑动到边界。**



> 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.createBallisticS
  if (simulation != null) {
    beginActivity(BallisticScrollActivity(this, simulatio
  } else {
    goIdle();
  }
}
```

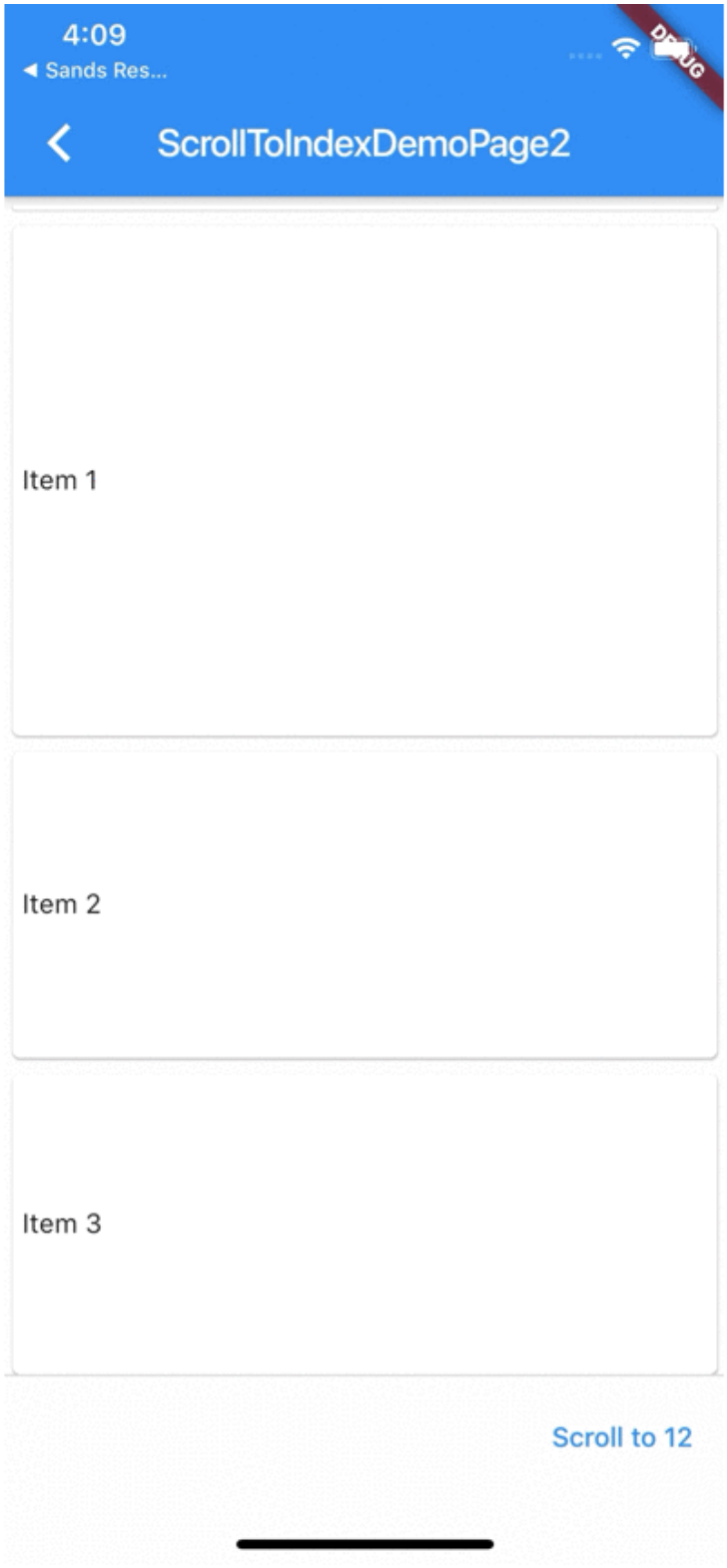
在 `BallisticScrollActivity` 状态中，**`Simulation` 被用于驱动 `AnimationController` 的 `value` ，然后在动画的回调中获取 `Simulation` 计算后得到的 `value` 进行 `setPixels(value)` 实现滚动。**

> 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

作为系列文章的第十九篇，本篇将科普 Android 和 iOS 平台的打包和提交审核流程。

因为很多 Flutter 开发人员可能只有单端的开发经验，对于另外一端的打包和提审流程不熟悉，或者是前端人员没有提交审核的经验，所以本篇将科普这一流程，让大家少走弯路。

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、Android 打包和审核流程

1、打包

事实上 Androd 的打包和审核流程都相对简单，打包 apk 只需要通过如下命令行就可以完成：

```
flutter build apk --target-platform android-arm64

flutter build apk --target-platform android-arm64 -t lib/ma
```

- 其中 `--target-platform` 是针对打包后的 so 文件， 对需要支持的框架进行选择，因为现在无论是 Goole Play 或者国内平台，都多都有要求应用需要支持 `arm64-v8a` 的 ABI 架构，所以一般打包也会选择指定 `target-platform` 来减小 apk 的体积。
- `-t` 表示指定其他 `main.dart` 打包，也可以不指定。
- 另外需要注意，Android 上需要在 `android/app/src/build.gradle` 下配置 `signingConfigs` 来指定打包密钥等信息，具体生成密钥这里就不详说，之后把 `signingConfigs` 配置到 `buildTypes` 就完成配置。

```
android {
    ....
    signingConfigs {
        config {
            keyAlias "xxxx"
            keyPassword "xxxx"
            storeFile file("../keystores/xxxxx.jks")
            storePassword "xxxx"
        }
    }
```

最后需要注意，如果你的 Apk 存在其他类型架构的 so 目录，比如 `armeabi-v7a` 等，那就需要在 `android/app/src/build.gradle` 的 `android { buildTypes {` 下加上 `ndk abiFilters` 进行过滤配置，因为 Android 下需要保证每个 ABI 目录内的 so 文件是完整齐全的，不然可能出现崩溃。

```
  buildTypes {
        release {
            signingConfig signingConfigs.config
            ndk {
                //设置支持的SO库架构
                abiFilters 'arm64-v8a'
            }
        }
        debug {
            signingConfig signingConfigs.config
            ndk {
                //设置支持的SO库架构
                abiFilters 'arm64-v8a', 'x86', 'x86_64'
            }
        }
    }
```

最后打包完的 Apk 默认会在如下图所示路径

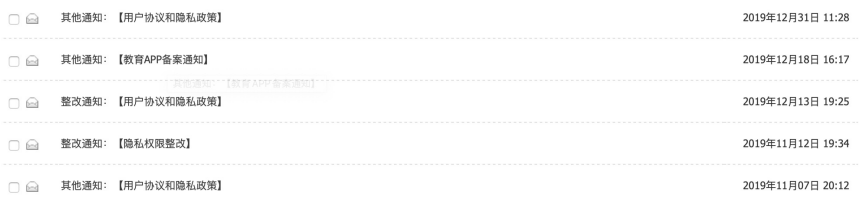
2、提交审核

其实在 Android 上提交审核是比较简单的，因为 Android 只需要提供 Apk 下载链接就可以直接安装，所以很多厂家都在有自己服务器上直接放上 Apk 文件，但是为了更好的体验和分发，大多数情况下也会选上传到各大应用平台，比如华为上没有上架的话，会出现如下图所示问题。

> 甚至有些 Apk 因为没有上架，会因为 `app_name` 等原因被当作病毒提醒。

事实上国内的应用市场审核并不麻烦，只是因为平台多且各家条件可能不一样变得比较繁琐，目前主流要求的有：

- `targetSdkVersion` 28 (9.0)；
- ABI 需要支持 `arm64-v8a` ；
- 应用需要针对 AndroidQ（10.0）进行适配，比如文件读取权限变更；
- 教育类应用需要备案；
- 需要提供用户隐私协议和权限说明；



之后就是一些平台的独立审核问题，比如 **360 平台审核要求你的 Apk 需要经过它们的应用加固**（加固后的作用就见仁见智），并且不少平台如**应用宝要求提供应用的版权说明等文件**，这些都是比较磨人的东西。

> 当然有些平台你可以不上，但是比如不上应用宝，你就很难获得*微信扫一扫后跳转打开应用和下载的能力*。

另外比如华为平台会有：根据工信部关于开展 APP 侵害用户权益专项整治工作的通知要求，应用内还需要提供帐号注销服务或销户功能能力。

可以看出 Android 的审核和条件其实并不繁琐，只是有些平台需要的东西比较磨人，具体需要上架可以根据需求自行斟酌了。

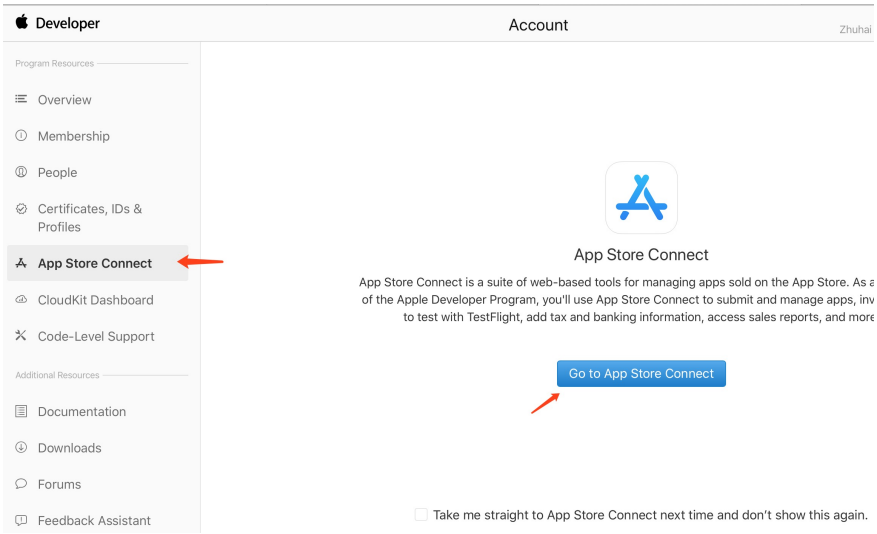
二、iOS 打包和审核流程

1、打包

iOS 的打包和审核流程相对复杂点，打包 iOS 首先你需要有*开发者账号*、*给应用申请和设置有* `Bundle Identifier` 、*配置文件*、*证书等信息*，相信已经到打包阶段了，这系列文件你不会欠缺吧?

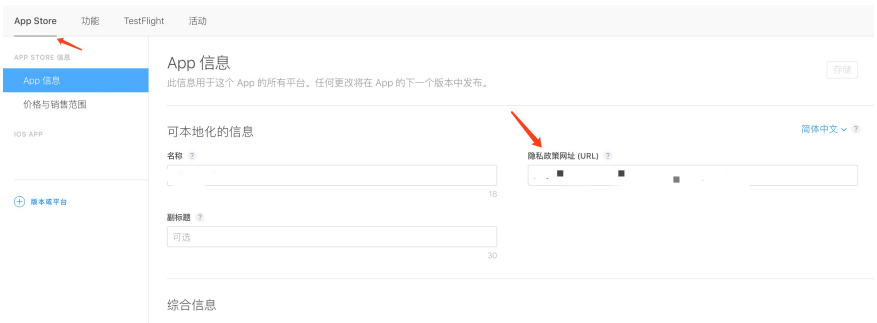
1.1 创建 App Store Connect

通过登录 https://developer.apple.com 网站，在 `Account` 的 `Certificates,IDs & Profiles` 可以找到你应用的信息，同时在 `App Store Connect` 栏目可以前往 https://appstoreconnect.apple.com 。

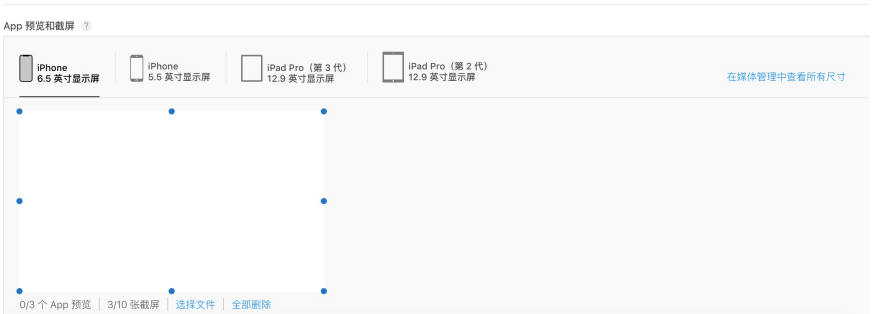


接着在 `我的 App` 按照提示创建应用，填写信息根据业务要求填写即可，这里主要说几个需要关注的点。

- 1、如下图所示在 App Store 的 App 信息里有一个隐私政策网站输入栏，这个是必填的，一般就是放一个 Html，具体可以参考类似的：https://guoshuyu.cn/home/index/privacy.html



- 2、需要上传应用的截图，一般需要准备 3-5 张预览图，但是这里需要 6.5 寸和 5.5 寸两种，如果还需要支持 iPad 版本那就还需要上传 12.9 的 iPad 图。这里推荐下，**如果没有设计师出稿件，推荐使用模拟器进行截图（注意不要截入 DEBUG 的 Label），6.5 寸可以用 iPhone 11promax 模拟器，5.5 寸的用 8plus 模拟器，打开具体页面后，按下 command + s 可以保存到桌面。**



> 这里需要注意，截图的画面不要太简单，最好能替体现应用的具体内容，不然很容易被拒绝，这里同时提供需要尺寸对应的设备型号。

截屏规范

设备尺寸或平台	截屏尺寸	要求	截屏源
6.5 英寸（iPhone 11 Pro Max、iPhone 11、iPhone Xs Max、iPhone Xʀ）	纵向分辨率为 1242 x 2688 像素 横向分辨率为 2688 x 1242 像素	若 App 在 iPhone 上运行，则此项为必需项	上传 6.5 英寸截屏
5.8 英寸（iPhone 11 Pro、iPhone X、iPhone Xs）	纵向分辨率为 1125 x 2436 像素 横向分辨率为 2436 x 1125 像素	若 App 在 iPhone 上运行，且未提供 6.5 英寸截屏，则此项为必需项	默认：缩小版 6.5 英寸截屏 可选：上传 5.8 英寸截屏
5.5 英寸（iPhone 6s Plus、iPhone 7 Plus、iPhone 8 Plus）	纵向分辨率为 1242 x 2208 像素 横向分辨率为 2208 x 1242 像素	若 App 在 iPhone 上运行，则此项为必需项	上传 5.5 英寸截屏
4.7 英寸（iPhone 6、iPhone 6s、iPhone 7、iPhone 8）	纵向分辨率为 750 x 1334 像素 横向分辨率为 1334 x 750 像素	若 App 在 iPhone 上运行，且未提供 5.5 英寸截屏，则此项为必需项	默认：缩小版 5.5 英寸截屏 可选：上传 4.7 英寸截屏
4 英寸（iPhone SE）	纵向分辨率（不含状态栏）为 640 x 1096 像素 纵向分辨率（不含状态栏）为 640 x 1136 像素 横向分辨率（不含状态栏）为 1136 x 600 像素 横向分辨率（不含状态栏）为 1136 x 640 像素	若 App 在 iPhone 上运行，且未提供 5.5 英寸或 4.7 英寸截屏，则此项为必需项	默认：缩小版 5.5 英寸或 4.7 英寸截屏 可选：上传 4 英寸截屏
3.5 英寸（iPhone 4s）	纵向分辨率（不含状态栏）为 640 x 920 像素 纵向分辨率（含状态栏）为 640 x 960 像素 横向分辨率（不含状态栏）为 960 x 600 像素 横向分辨率（含状态栏）为 960 x 640 像素	若 App 在 iPhone 上运行，且未提供 5.5 英寸 iPhone 截屏，则此项为必需项	默认：缩小版 5.5 英寸、4.7 英寸或 4 英寸截屏 可选：上传 3.5 英寸截屏
12.9 英寸（第 3 代 iPad Pro）	纵向分辨率为 2048 x 2732 像素 横向分辨率为 2732 x 2048 像素	若 App 在 iPad 上运行，则此项为必需项	上传适用于 12.9 英寸 iPad Pro（第 3 代）的截屏
12.9 英寸（第 2 代 iPad Pro）	纵向分辨率为 2048 x 2732 像素 横向分辨率为 2732 x 2048 像素	若 App 在 iPad 上运行，则此项为必需项	上传适用于 12.9 英寸 iPad Pro（第 2 代）的截屏
11 英寸（iPad Pro）	纵向分辨率为 1668 x 2388 像素 横向分辨率为 2388 x 1668 像素	若 App 在 iPad 上运行，且未提供适用于 12.9 英寸 iPad Pro（第 2 代）的截屏，则此项为必需项	默认：适用于 12.9 英寸 iPad Pro（第 3 代）的缩小版截屏 可选：上传 11 英寸截屏
10.5 英寸（第 7 代 iPad、iPad Pro、iPad Air）	纵向分辨率为 1668 x 2224 像素 横向分辨率为 2224 x 1668 像素	若 App 在 iPad 上运行，且未提供适用于 12.9 英寸 iPad Pro（第 2 代）的截屏，则此项为必需项	默认：适用于 12.9 英寸 iPad Pro（第 2 代）的缩小版截屏 可选：上传 10.5 英寸截屏
9.7 英寸（iPad、iPad mini）	高分辨率： 纵向分辨率（不含状态栏）为 1536 x 2008 像素 纵向分辨率（含状态栏）为 1536 x 2048 像素 横向分辨率（不含状态栏）为 2048 x 1496 像素 横向分辨率（含状态栏）为 2048 x 1536 像素 标准分辨率： 纵向分辨率（不含状态栏）为 768 x 1004 像素 纵向分辨率（含状态栏）为 768 x 1024 像素 横向分辨率（不含状态栏）为 1024 x 748 像素 横向分辨率（含状态栏）为 1024 x 768 像素	若 App 在 iPad 上运行，且未提供适用于 12.9 英寸 iPad Pro（第 2 代）或 10.5 英寸的截屏，则此项为必需项	默认：适用于 12.9 英寸 iPad Pro（第 2 代）的缩小版截屏或缩小版 10.5 英寸截屏 可选：上传 9.7 英寸截屏
Mac	具有 16:10 宽高比的下列分辨率之一。 1280 x 800 像素 1440 x 900 像素 2560 x 1600 像素 2880 x 1800 像素	对于 Mac App 为必需项	以任一所列尺寸上传用于 Mac 的截屏
Apple TV	1920 x 1080 像素 3840 x 2160 像素	对于 Apple TV App 为必需项	以任一所列尺寸上传用于 Apple TV 的截屏
Apple Watch	以下之一： 312 x 390 像素（Series 3） 368 x 448 像素（Series 5、Series 4）	对于 Apple Watch App，此项为必需项	以任一所列尺寸上传用于 Apple Watch 的截屏

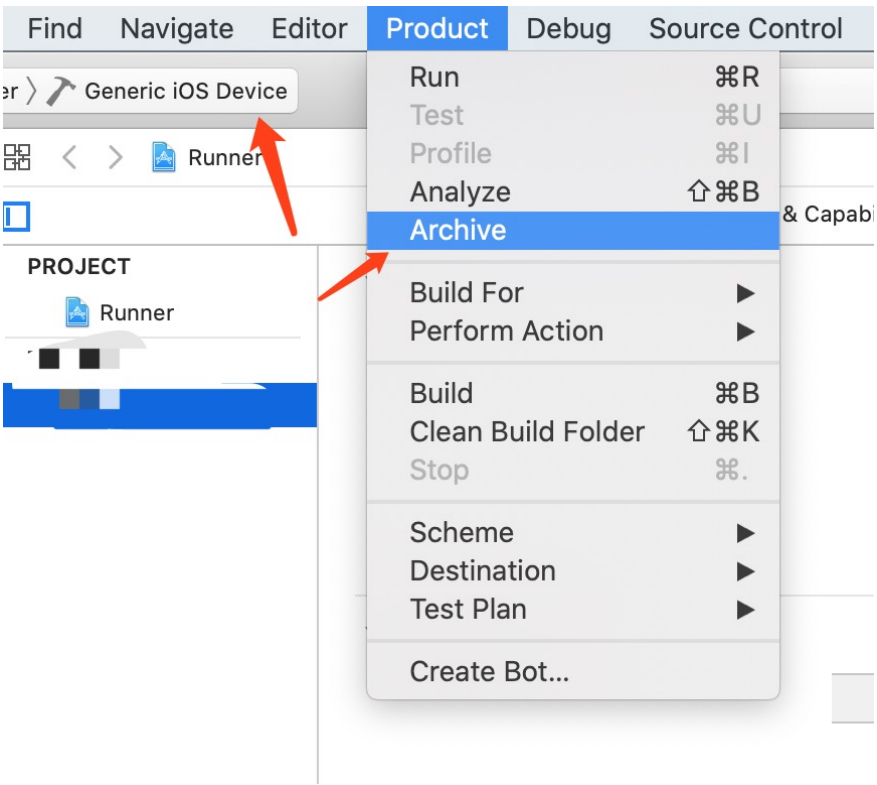
- 3、在版本的信息里还有技术支持网站的必填，这个具体可以参考：https://guoshuyu.cn/home/index/tech.html ，如果此处不符合条件也会出现审核不通过的问题。

- 4、另外如果 App 需要登录，还需要提供用户的测试账号和密码等。

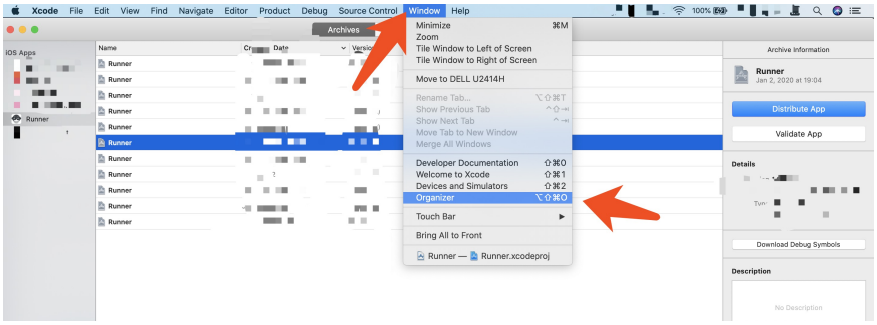
1.2 打包上传

打包 flutter iOS 首先需要执行 `flutter build ios` 命令，命令会生成 release 模式的下的 `framework` 文件，之后就可以进入 Xcode 流程。

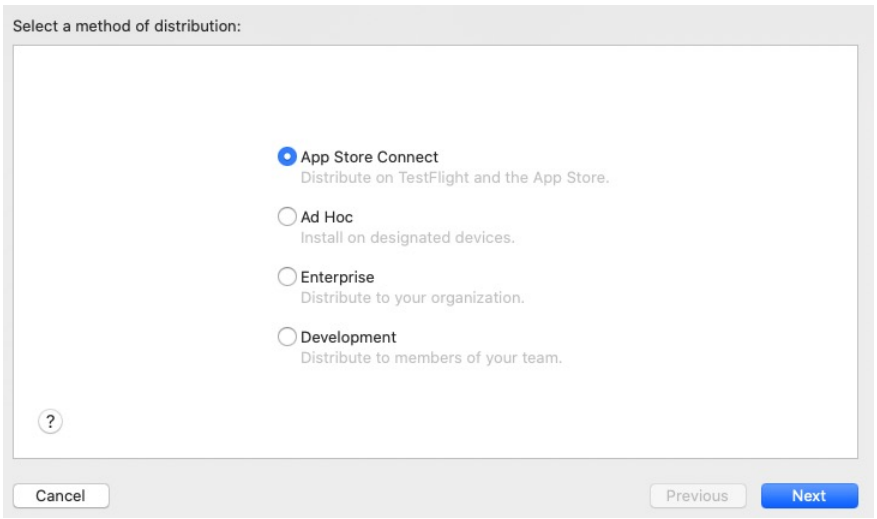
如下图所示，首先确保🔨位置不要选中模拟器，之后在 Product > Archive 就会开始导出打包。



打包成功后可以看到如下界面，找到你最新打包的那一项，选择 `Distribute App` 就可以进入下一步；另外打包过的项目在 Window > Organizer 也可以重新找到。

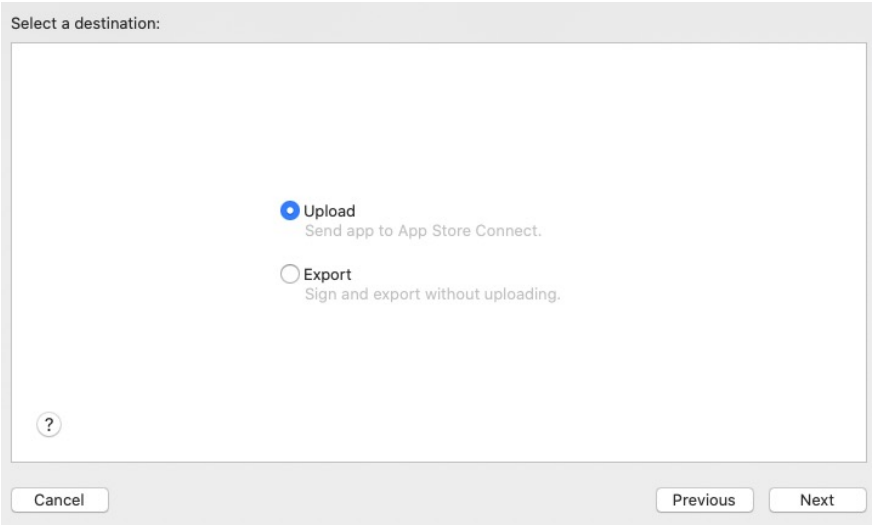


之后如下所示，就选择上传 `App Store Connect` 进行提交准备。

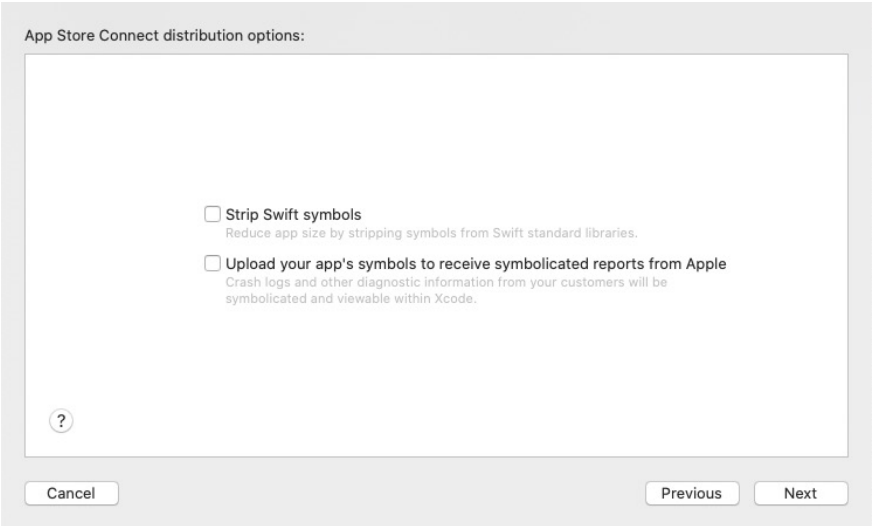


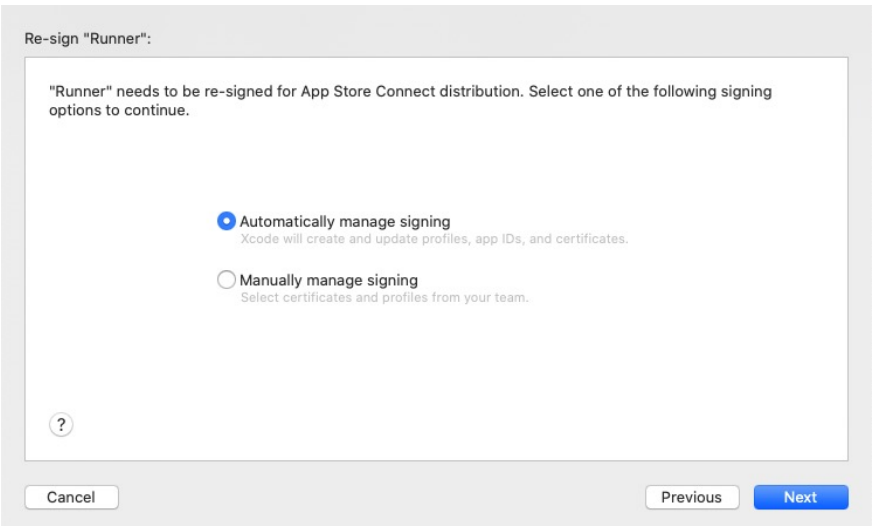
> 如果是选择导出测试 ipa 可以选择 `Development`，前提是对应机器的 `UDID` 等信息已经在打包配置文件内。

之后可以选择 Upload 或者 Export，Export 就是导出后再在本地上传，可以使用 `TransPorter` 工具再单独上传；Upload 就是前面之后直接上传。



接着出现的这个页面建议是不要勾选（不要问，问就是百度），然后直接next，然后选择自动签名，等签名成功后最后点击上传就可以了。





2、审核

上传成功后就，过一段时间可以在 `活动` 和 `TestFlight` 看到你提交的构建版本，然后你可能会收到如下所示的一封邮件：

App Store Connect

Dear Developer,

We identified one or more issues with a recent delivery for your app, "GSYGithubApp" 1.6.8 (31). Please correct the following issues, then upload again.

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSContactsUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSCalendarsUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSAppleMusicUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSMotionUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSSpeechRecognitionUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

Though you are not required to fix the following issues, we wanted to make you aware of them:

ITMS-90078: Missing Push Notification Entitlement - Your app appears to register with the Apple Push Notification service, but the app signature's entitlements do not include the "aps-environment" entitlement. If your app uses the Apple Push Notification service, make sure your App ID is enabled for Push Notification in the Provisioning Portal, and resubmit after signing your app with a Distribution provisioning profile that includes the "aps-environment" entitlement. Xcode does not automatically copy the aps-environment entitlement from provisioning profiles at build time. This behavior is intentional. To use this entitlement, either enable Push Notifications in the project editor's Capabilities pane, or manually add the entitlement to your entitlements file. For more information, see https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/HandlingRemoteNotifications.html#//apple_ref/doc/uid/TP40008194-CH6-SW1.

ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSLocationAlwaysUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

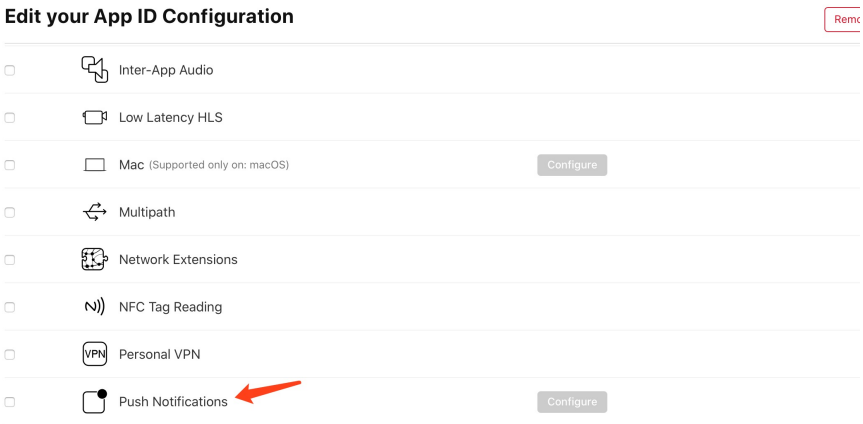
ITMS-90683: Missing Purpose String in Info.plist - Your app's code references one or more APIs that access sensitive user data. The app's Info.plist file should contain a NSLocationWhenInUseUsageDescription key with a user-facing purpose string explaining clearly and completely why your app needs the data. Starting Spring 2019, all apps submitted to the App Store that access user data are required to include a purpose string. If you're using external libraries or SDKs, they may reference APIs that require a purpose string. While your app might not use these APIs, a purpose string is still required. You can contact the developer of the library or SDK and request they release a version of their code that doesn't contain the APIs. Learn more (https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy).

Best regards,

其中比如 `ITMS-90683` 说的是没有在 `plist` 内配置 `NSContactsUsageDescription` 的 key-value，也就是向用户解释你为什么需要用到读取用户联系人的权限。

诸如此类的还有后几个都是，如果你应用内用到了对应的权限。就需要在 `plist` 配置上对应的 `key-value` 。

另外就是 `Push Notification Entitlement` 的警告，是说你的应用没有配置推送相关的证书和设置，如果你的应用没有用到对应的功能，比如在 `Developer` 后台看如下图所示的推送是否勾选了，如果勾选了就需要在应用内配置对应的推送服务，iOS 上 APNS 还需要设置对应的推送证书，一般推送证书还会分开发和生产两种，如果没有使用推送可以忽略警告。



还有就是 App 的启动页和 logo 尺寸记得配全，配置不全也会收到对应的警告，这个可能会影响审核。

之后在版本信息里选择需要提交的构建版本，之后提交审核即可，一般审核会从等到*审核* > *正在审核* > *审核结果*，这个过程一般在 24 或者 48 小时之内，但是如果赶上了像圣诞节这样的节日，苹果会因为放假放慢审核，另外被拒绝的太多次的话，也会影响审核速度。

如下图所示，最后提一些审核建议，比如：

- 前面说过的应用截图要尽量体现应用的主要内容；
- 不允许在应用内滥用应用更新提示，比如不允许应用自己跳转下载更新，只能是简单提示后跳转 app store ，如果把握不好尺度干脆在 iOS 上就不加；
- 不要在应用内带有 fir.im ，蒲公英等资源、链接、文本和SDK，不然很容易被扫描然后拒绝。

以上这些都是属于常犯的问题，更多的还请看：
https://developer.apple.com/cn/app-store/review/guidelines/

发件人 Apple

2. 3 Performance: Accurate Metadata
2. 5 Performance: Software Requirements
4. Design: Preamble
4. 1 Design: Copycats

Guideline 2.3.3 - Performance - Accurate Metadata

We noticed that your screenshots do not sufficiently reflect your app in use.

Specifically, your screenshots only display a splash screen and only display a login screen.

Next Steps

To resolve this issue, please revise your screenshots to ensure that they accurately reflect the app in use on the supported devices. For example, a gaming app should feature screenshots that capture actual gameplay from within the app. Marketing or promotional materials that do not reflect the UI of the app are not appropriate for screenshots.

The iPhone screenshots should reflect use on iPhone devices.

The iPad screenshots should reflect use on iPad devices.

App Store screenshots should accurately communicate your app's value and functionality. Use text and overlay images to highlight your app's user experience, not obscure it. Make sure app UI and product images match the corresponding device type in App Store Connect. This helps users understand your app and makes for a positive App Store experience.

Resources

For resources on creating great screenshots for the App Store, you may want to review the App Store Product Page information available on the Apple Developer website.

Please ensure you have made any screenshot modifications using Media Manager. You should confirm your app looks and behaves identically in all languages and on all supported devices. Learn more about uploading app previews and screenshots in App Store Connect Help.

Guideline 2.5.2 - Performance - Software Requirements

During review, your app installed or launched executable code, which is not permitted on the App Store. Specifically, your app uses the itms-services URL scheme to install an app.

Please note that while educational apps designed to teach, develop, or allow students to test executable code may, in limited circumstances, download code, such code may not be used for other purposes and such apps must make the source code completely viewable and editable by the user.

The next submission of this app may require a longer review time, and this app will not be eligible for an expedited review until this issue is resolved.

Next Steps

- Review the Software Requirements section of the App Store Review Guidelines.
- Ensure your app is compliant with all sections of the App Store Review Guidelines and the Terms & Conditions of the Apple Developer Program.
- Once your app is fully compliant, resubmit your app for review.

Submitting apps designed to mislead or harm customers or evade the review process may result in the termination of your Apple Developer Program account. Review the Terms & Conditions of the Apple Developer Program to learn more about our policies regarding termination.

Guideline 4.0 - Design

Your app includes an update button or alerts the user to update the app, but the update button or alert does not link directly to the app's page on the App Store.

Next Steps

To resolve this issue, please ensure that tapping the update button takes the user directly to the app's page on the App Store to update the app.

Guideline 4.1 - Design - Copycats

Your metadata appears to contain third party content. Specifically, your app leverages the popularity of Github.

The next submission of this app may require a longer review time, and this app will not be eligible for an expedited review until this issue is resolved.

Next Steps

- Review the Copycats section of the App Store Review Guidelines.
- Ensure your app is compliant with all sections of the App Store Review Guidelines and the Terms & Conditions of the Apple Developer Program.
- Provide rights to any protected or trademarked content in App Store Connect.
- Once your app is fully compliant, resubmit your app for review.

Submitting apps designed to mislead or harm customers or evade the review process may result in the termination of your Apple Developer Program account. Review the Terms & Conditions of the Apple Developer Program to learn more about our policies regarding termination.

You may attach documentary evidence in the App Review Information section in App Store Connect. In accordance with section 3.2(f of the Apple Developer Program License Agreement, you acknowledge that submitting falsified or fraudulent documentation can result in the termination of your Apple Developer Program account and the removal of your apps from the App Store. Once Legal has reviewed your documentation and confirms its validity, we will proceed with the review of your app.

> iOS 还有可以不用上架，只需要用户在手机上信任证书的可以使用 ipa 的开发者账号，但是这类开发者账号现在很难申请得到，并且这类账号的应用需要一年后重新打包一次更新。

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

作为系列文章的第二十篇，本篇将结合官方的技术文档科普 Android 上 `PlatformView` 的实现逻辑，并且解释为什么在 Android 上 `PlatformView` 的键盘总是有问题。

> 为什么 iOS 上相对稳定，文中也做了对应介绍。

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、为什么有 PlatformView

因为 Flutter 的实现在概念上类似于 Android 上的 `WebView`，Flutter 是通过将 `Widget Tree` 转化为纹理后通过 Skia 实现控件绘制，这造就了优秀的跨平台效果的同时，也带来了不可逆的兼容问题。

1.1、无法集成原生平台控件

这就像 WebView 一样，Flutter UI 不会转换为 Android 控件，而是由 Flutter Engine 使用 Skia 直接在 `SurfaceView` 上渲染出来。

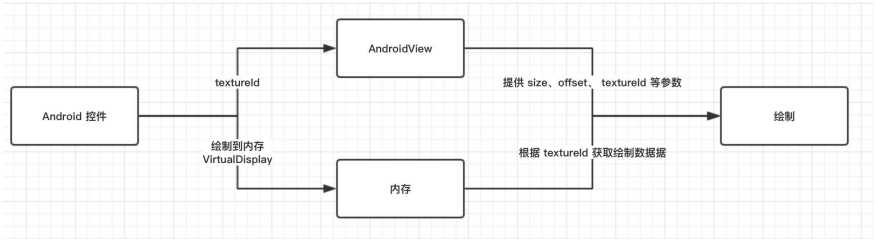
这意味着默认情况下 Flutter UI 永远不会包含 Android Native 的控件，也就是说无法在 Flutter 中集成如 `WebView` 或 `MapView` 这些常用的控件。

所以为解决这个问题，Flutter 创建了一个叫 `AndroidView` 的控件逻辑， 开发者使用该 Widget 可以将 Android Native 组件嵌入到 Flutter UI 中。

1.2、AndroidView 的实现

`AndroidView` 这个 Widget 需要和 Flutter 相结合才能完整显示：**在 Flutter 中通过将 `AndroidView` 需要渲染的内容绘制到 `VirtualDisplays` 中，然后在 `VirtualDisplay` 对应的内存中，绘制的画面就可以通过其 `Surface` 获取得到。**

> `VirtualDisplay` 类似于一个虚拟显示区域，需要结合 `DisplayManager` 一起调用，一般在副屏显示或者录屏场景下会用到。 `VirtualDisplay` 会将虚拟显示区域的内容渲染在一个 `Surface` 上。



如上图所示，**简单来说就是原生控件的内容被绘制到内存里，然后 Flutter Engine 通过相对应的 `textureId` 就可以获取到控件的渲染数据并显示出来。**

通过从 `VirtualDisplay` 输出中获取纹理，并将其和 Flutter 原有的 UI 渲染树混合，使得 Flutter 可以在自己的 Flutter Widget tree 中以图形方式插入 Android 原生控件。

1.3、 有其他可以实现的方式吗?

在 iOS 平台上就不使用类似 `VirtualDisplay` 的方法，而是**通过将 Flutter UI 分为两个透明纹理来完成组合：一个在 iOS 平台视图之下，一个在其上面。**

所以这样的好处就是：需要在“iOS平台”视图下方呈现的Flutter UI，最终会被绘制到其下方的纹理上；而需要在“平台”上方呈现的Flutter UI，最终会被绘制在其上方的纹理。**它们只需要在最后组合起来就可以了。**

通常这种方法更好，因为这意味着 Android Native View 可以直接添加到 Flutter 的 UI 层次结构中。

但是，Android 平台并不支持这种模式，因为在 iOS 上框架渲染后系统会有回调通知，例如：*当 iOS 视图向下移动 `2px` 时，我们也可以将其列表中的所有其他 Flutter 控件也向下渲染 `2px`* 。

但是在 Android 上就没有任何有关的系统 API，因此无法实现同步输出的渲染。**如果强行以这种方式在 Android 上使用，最终将产生很多如 `AndroidView` 与 Flutter UI 不同步的问题。**

> 有关此替代方法的详细讨论，详见 https://flutter.dev/go/nshc

2、相关问题和解决方法

尽管前面可以使用 `VirtualDisplay` 将 Android 控件嵌入到 Flutter UI 中，但这种 `VirtualDisplay` 的介入还有其他麻烦的问题需要处理。

2.1、触摸事件

默认情况下， `PlatformViews` 是没办法接收触摸事件。

因为 `AndroidView` 其实是被渲染在 `VirtualDisplay` 中，而每当用户点击看到的 `"AndroidView"` 时，其实他们就真正”点击的是正在渲染的 `Flutter` 纹理 。**用户产生的触摸事件是直接发送到 Flutter View 中，而不是他们实际点击的 `AndroidView` 。**

2.1.1、解决方法

- `AndroidView` 使用 Flutter Framework 中的点击测试逻辑来检测用户的触摸是否在需要特殊处理的区域内。

> 类似可见：《Flutter完整开发实战详解(十三、全面深入触摸和滑动原理)》

- 当触摸成功时会向 Android embedding 发送一条消息，其中包含 touch 事件的详细信息。
- 在 Android embedding 中，该事件的坐标最后会匹配到 `AndroidView` 在 `VirtualDisplay` 中的坐标，然后会创建一个 `MotionEvent` 用于 描述触摸的新控件，并将其转发到内部 `VirtualDisplay` 中真实的 `AndroidView` 中进行响应。

2.1.2、局限性

- 该实现逻辑会将新的 `MotionEvent` 直接分发给 `AndroidView` ，如果这个 View 又派生了其他视图，那么就可能会出现触摸信息被发送到错误的位置。
- `MotionEvent` 的转化过程中可能会因为机制的不同，存在某些信息没办法完整转化的丢失。

2.2、文字输入

通常， `AndroidView` 是无法获取到文本输入，因为 `VirtualDisplay` 所在的位置会始终被认为是 `unfocused` 的状态。

Android 目前不提供任何 API 来动态设置或更改的焦点 `Window` ， `Flutter` 中 `focused` 的 `Window` 通常是实际持有“真实的” Flutter 纹理和 UI ，并且对于用户直接可见。

而 **`InputConnections` （如何在 Android 中 输入文本）在 `unfocused` 的 View 中通常是会被丢弃。**

2.2.1、解决方法

- **Flutter 重写了 `checkInputConnectionProxy` 方法，这样 Android 会认为 Flutter View 是作为 `AndroidView` 和输入法编辑器（IME）的代理**，这样 Android 就可以从 Flutter View 中获取到 `InputConnections` 然后作用于 `AndroidView` 上面。

- **在 Android Q 开始 `InputMethodManager` （IMM）改为每个 `Window` 自己实例化而不是全局单例。**因此之前幼稚的“设置代理”的模式在 Q 开始不起作用。为了进一步解决这个问题，**Flutter 创建了一个 `Context` 的子类， 该子类返回的内容与 Flutter View 中的 `IMM` 相同，这样就不会需要在查询 `IMM` 时需要返回的真实的 `Window` 。**这意味着当 Android 需要 `IMM` 时， `VirtualDisplay` 仍然会使用 Flutter View 的 `IMM` 作为代理。
- 当要求 `AndroidView` 提供 `InputConnection` 时，它会检查 `AndroidView` 是否确实是输入的目标。如果是，那 `AndroidView` 中的 `InputConnection` 将被获取并返回给 Android 。
- Android 认为 Flutter View 是 `focused` 且可用的，因此 `AndroidView` 的 `InputConnection` 可以成功被获取并使用。

2.2.2、 Platforview 中的 WebView 键盘输入

在 Android N 之前的版本上 `WebView` 输入比较复杂，因为它们具有自己内部的逻辑来创建和设置输入连接，而这些输入连接并没有完全遵循 Android 的协议。在 `flutter_webview` 插件中，还需要添加其他解决方法以便在可以在 `WebView` 启用文本输入。

- 设置一个代理 View ， 该 View 与 `WebView` 在相同的线程上侦听输入连接。如果没有此功能， `WebView` 将在内部消耗所有 `InputConnection` 的呼叫，而不会通知 Flutter View 代理。
- 在代理线程中，返回 Flutter View 以创建输入。。
- `WebView` 失去焦点时，将输入连接重置回 Flutter 线程。这样可以防止文本输入“卡”在 WebView 内。

2.2.3、局限性

- 通常这个逻辑取决于 Android 的内部行为，并且可能会十分脆弱，比如： *1.12 版本下针对华为等设备出现的键盘输入异常等问题*。
- 某些文本功能仍然不可用，例如：*“复制”和“共享”对话框当前不可用。*

3、总结

`PlatformView` 的实现模式增加了 Flutter 的生命力和活力，但是相对的也引出了很多问题，比如 #webview-keyboard、#webview、#platform-views 相关的 issue 专题高居不下，并且如 webview_flutter 插件的文档所述：

> 该插件依赖 Flutter 的新机制来嵌入 Android 和 iOS 视图。由于该机制当前处于开发人员预览中，因此该插件也应被视为开发人员预览。
>
> `webview_flutter` 的键盘支持也尚未准备好用于生产，因为 Webview 中的键盘支持目前还处于实验性的阶段。

所以到这里相信你应该知道，为什么 Flutter 中的 `PlatforView` 在 Android 上如此之难兼容，并且键盘输入问题会那么多坑了。

> 自此，第二十篇终于结束了！(///▽///)

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
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp

作为系列文章的第二十一篇，本篇将通过不一样的角度来介绍 Flutter Framework 的整体渲染原理，深入剖析 Flutter 中构成 Layer 后的绘制流程，让开发者对 Flutter 的渲染原理和实现逻辑有更清晰的认知。

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、Layer 相关的回顾

先回顾下，我们知道在 Flutter 中的控件会经历 `Widget` -> `Element` -> `RenderObject` -> `Layer` 这样的变化过程，而其中 `Layer` 的组成由 `RenderObject` 中的 `isRepaintBoundary` 标志位决定。

> 当调用 `setState` 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往下去触发重绘，换个说法就是：**确定要更新哪些区域**。

比如 `Navigator` 跳转不同路由页面，每个页面内部就有一个 `RepaintBoundary` 控件，这个控件对应的 `RenderRepaintBoundary` 内的 `isRepaintBoundary` 标记位就是为 `true` ，从而路由页面之间形成了独立的 `Layer` 。

所以相关的 `RenderObject` 在一起组成了 `Layer`，而由 `Layer` 构成的 `Layer Tree` 最后会被提交到 Flutter Engine 绘制出画面。

那 `Layer` 是怎么工作的？它的本质又是什么？ Flutter Framework 中 `Layer` 是如何被提交到 Engine 中?

二、Flutter Framework 中的绘制

带着前面 `Layer` 的问题，我们先做个假设：如果抛开 Flutter Framework 中封装好的控件，我们应该如何绘制出一个画面？或者说如何创建一个 `Layer` ？

举个例子，如下代码所示，运行后可以看到一个居中显示的 100 x 100 的蓝色方块，并且代码里没有用到任何 `Widget` 、 `RenderObject` 甚至 `Layer` ，而是使用了 **`PictureRecorder` 、`Canvas` 、`SceneBuilder` 这些相对陌生的对象完成了画面绘制，并且在最后执行的是 `window.render`** 。

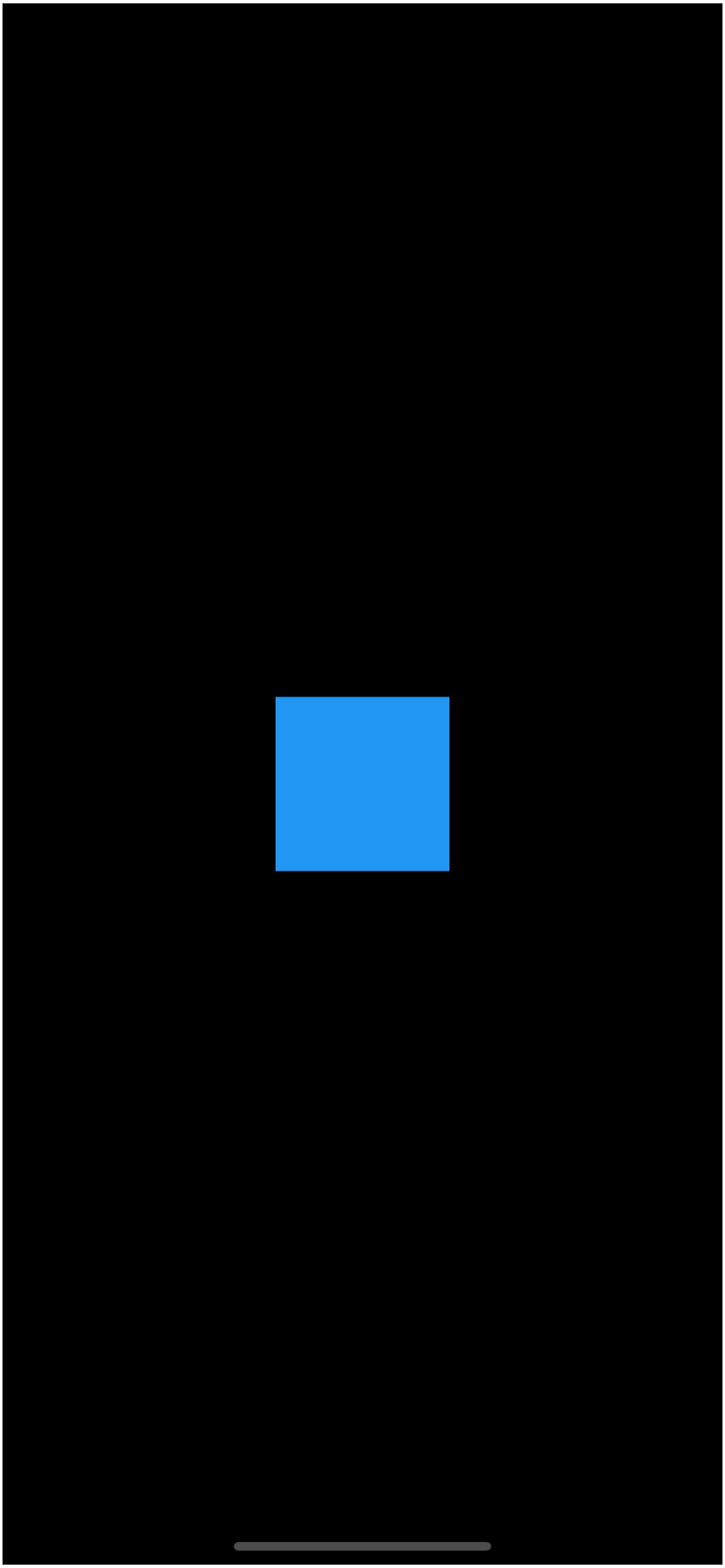
```
import 'dart:ui' as ui;

void main() {
  ui.window.onBeginFrame = beginFrame;

  ui.window.scheduleFrame();
}

void beginFrame(Duration timeStamp) {
  final double devicePixelRatio = ui.window.devicePixelRat

  ///创建一个画板
  final ui.PictureRecorder recorder = ui.PictureRecorder();

  ///基于画板创建一个 Canvas
  final ui.Canvas canvas = ui.Canvas(recorder);
  canvas.scale(devicePixelRatio, devicePixelRatio);

  var centerX = ui.window.physicalSize.width / 2.0;
  var centerY = ui.window.physicalSize.height / 2.0;

  ///画一个 100 的剧中蓝色
  canvas.drawRect(
      Rect.fromCenter(
          center: Offset.zero,
          width: 100,
          height: 100),
      new Paint()..color = Colors.blue);

  ///结束绘制
  final ui.Picture picture = recorder.endRecording();

  final ui.SceneBuilder sceneBuilder = ui.SceneBuilder()
    ..pushOffset(centerX, centerY)
    ..addPicture(ui.Offset.zero, picture)
    ..pop();

  ui.window.render(sceneBuilder.build());
}
```

因为在 Flutter 中 `Canvas` 的创建是必须有 `PictureRecorder` ，而 `PictureRecorder` 顾名思义就是创建一个图片用于记录绘制，所以在上述代码中：

- 先是创建了 `PictureRecorder` ；
- 然后使用 `PictureRecorder` 创建了 `Canvas` ；
- 之后使用 `Canvas` 绘制蓝色小方块；

- 结束绘制后通过 `SceneBuilder` 的 `pushOffset` 和 `addPicture` 加载了绘制的内容；
- 通过 `window.render` 绘制出画面。

> 需要注意⚠️： `render` 方法被限制必须在 `onBeginFrame` 或 `onDrawFrame` 中调用，所以上方代码才会有 `window.onBeginFrame = beginFrame;` 。在官方的 examples/layers/raw/ 下有不少类似的例子。



可以看到 Flutter Framework 在底层绘制的最后一步是 `window.render`，而如下代码所示： `render` 方法需要的参数是 `Scene` 对象，并且 `render` 方法是一个 `native` 方法，**说明 Flutter Framework 最终提交给 Engine 的是一个 `Scene`** 。

```
void render(Scene scene) native 'Window_render';
```

那 `Scene` 又是什么？前面所说的 `Layer` 又在哪里呢？它们之间又有什么样的关系？

三、Scene 和 Layer 之间的苟且

在 Flutter 中 `Scene` 其实是一个 `Native` 对象，它对应的其实是 `Engine` 中的 `scene.cc` 结构，而 Engine 中的 `scene.cc` 内包含了一个 `layer_tree_` 用于绘制，所以**首先可以知道 `Scene` 在 `Engine` 是和 `layer_tree_` 有关系**。

然后就是在 **Flutter Framework 中 `Scene` 只能通过 `SceneBuilder` 构建**，而 `SceneBuilder` 中存在很多方法比如：`pushOffset` 、 `pushClipRect` 、 `pushOpacity` 等，这些方法的执行后，可以通过 Engine 会创建出一个对应的 `EngineLayer` 。

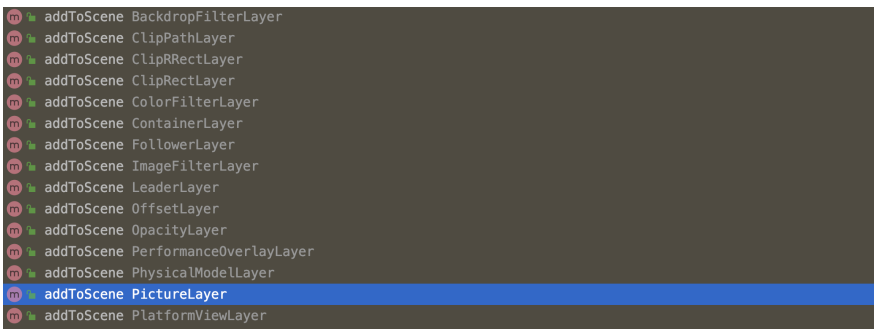
```
OffsetEngineLayer pushOffset(double dx, double dy, { Offs
  assert(_debugCheckCanBeUsedAsOldLayer(oldLayer, 'pushO
  final OffsetEngineLayer layer = OffsetEngineLayer._(_pu
  assert(_debugPushLayer(layer));
  return layer;
}
EngineLayer _pushOffset(double dx, double dy) native 'Sce
```

所以 `SceneBuilder` 在 `build` 出 `Scene` 之前，可以通过 `push` 等相关方法产生 `EngineLayer` ， 比如前面的蓝色小方块例子， `SceneBuilder` 就是通过 `pushOffset` 创建出对应的图层偏移。

接着看 Flutter Framework 中的 `Layer` ，如下代码所示，在 `Layer` 默认就存在 `EngineLayer` 参数，所以可以得知 `Layer` 肯定和 `SceneBuilder` 有一定关系。

```
@protected
ui.EngineLayer get engineLayer => _engineLayer;

@protected
set engineLayer(ui.EngineLayer value) {
  _engineLayer = value;
  if (!alwaysNeedsAddToScene) {

    if (parent != null && !parent.alwaysNeedsAddToScene)
      parent.markNeedsAddToScene();
    }
  }
}
ui.EngineLayer _engineLayer;

/// Override this method to upload this layer to the engi
///
/// Return the engine layer for retained rendering. When
/// corresponding engine layer, null is returned.

@protected
void addToScene(ui.SceneBuilder builder, [ Offset layerOf
```

其次在 `Layer` 中有一个关键方法： **`addToScene`** ，先通过注释可以得知这个方法是由子类实现，并且执行后可以得到一个 `EngineLayer` ，并且这个方法需要一个 `SceneBuilder` ，而查询该方法的实现恰好就有 `OffsetLayer` 和 `PictureLayer` 等。

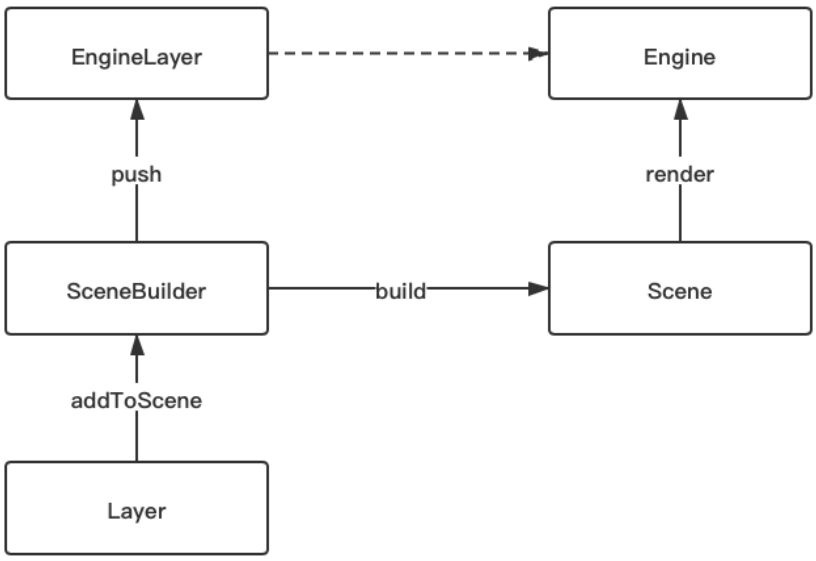


所以如下代码所示，在 `OffsetLayer` 和 `PictureLayer` 的 `addToScene` 方法实现中可以看到：

- `PictureLayer` 调用了 `SceneBuilder` 的 `addPicture` ；
- `OffsetLayer` 调用了 `SceneBuilder` 的 `pushOffset` ；

```
class PictureLayer extends Layer {
  ...
  @override
  void addToScene(ui.SceneBuilder builder, [ Offset layerOf
    builder.addPicture(layerOffset, picture, isComplexHint
  }
  ...
}

class OffsetLayer extends ContainerLayer {
  ...
  OffsetLayer({ Offset offset = Offset.zero }) : _offset =

  @override
  void addToScene(ui.SceneBuilder builder, [ Offset layerOf
    engineLayer = builder.pushOffset(
      layerOffset.dx + offset.dx,
      layerOffset.dy + offset.dy,
      oldLayer: _engineLayer as ui.OffsetEngineLayer,
    );
    addChildrenToScene(builder);
    builder.pop();
  }
  ...
}
```

所以到这里 **SceneBuilder** 和 **Layer** 通过 **EngineLayer** 和 **addToScene** 方法成功关联起来，而 **window.render** 提交的 **Scene** 又是通过 **SceneBuilder** 构建得到，所以如下图所示，**Layer** 和 **Scene** 就这样“苟且”到了一起。

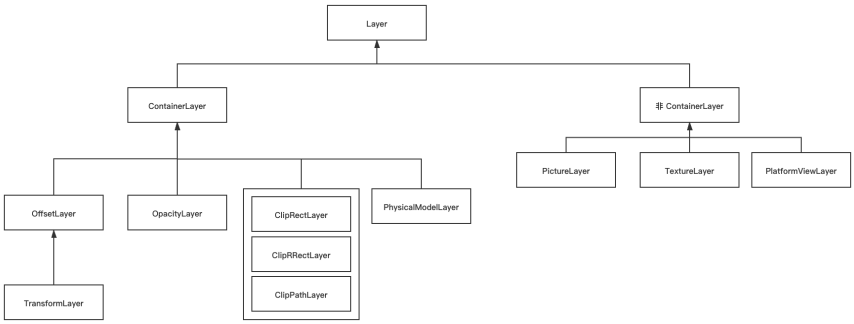


对面前面的蓝色小方块代码，如下代码所示，这里修改为使用 `Layer` 的方式实现，可以看到这样的实现更接近 Flutter Framework 的实现：**通过 `rootLayer` 一级一级 `append` 构建出 `Layer` 树，而 `rootLayer` 调用 `addToScene` 方法后，因为会执行 `addChildrenToScene` 方法，从而往下执行 child `Layer` 的 `addToScene`** 。

```
import 'dart:ui' as ui;

void main() {
  ui.window.onBeginFrame = beginFrame;

  ui.window.scheduleFrame();
}

void beginFrame(Duration timeStamp) {
  final double devicePixelRatio = ui.window.devicePixelRat

  ///创建一个画板
  final ui.PictureRecorder recorder = ui.PictureRecorder();

  ///基于画板创建一个 Canvas
  final ui.Canvas canvas = ui.Canvas(recorder);
  canvas.scale(devicePixelRatio, devicePixelRatio);

  var centerX = ui.window.physicalSize.width / 2.0;
  var centerY = ui.window.physicalSize.height / 2.0;

  ///画一个 100 的剧中蓝色
  canvas.drawRect(Rect.fromCenter(center: Offset.zero, widt
      new Paint()..color = Colors.blue);

  final ui.SceneBuilder sceneBuilder = ui.SceneBuilder();

  OffsetLayer rootLayer = new OffsetLayer();


  OffsetLayer offsetLayer = new OffsetLayer(offset: Offset(
  rootLayer.append(offsetLayer);

  PictureLayer pictureLayer = new PictureLayer(Rect.zero);
  pictureLayer.picture = recorder.endRecording();
  offsetLayer.append(pictureLayer);


  rootLayer.addToScene(sceneBuilder);


  ui.window.render(sceneBuilder.build());
}
```

四、Layer 的品种

这里额外介绍下 Flutter 中常见的 `Layer`，如下图所示，一般 Flutter 中 `Layer` 可以分为 `ContainerLayer` 和非 `ContainerLayer` 。



`ContainerLayer` 是可以具备子节点，也就是带有 `append` 方法，大致可以分为：

- 位移类（ `OffsetLayer` / `TransformLayer` ）；
- 透明类（ `OpacityLayer` ）
- 裁剪类（ `ClipRectLayer` / `ClipRRectLayer` / `ClipPathLayer`);
- 阴影类(`PhysicalModelLayer`)

为什么这些 `Layer` 需要是 `ContainerLayer` ？**因为这些 `Layer` 都是一些像素合成的操作，其本身是不具备“描绘”控件的能力，就如前面的蓝色小方块例子一样，如果要呈现画面一般需要和 `PictureLayer` 结合。**

> 比如 `ClipRRect` 控件的 `RenderClipRRect` 内部，在 `pushClipRRect` 时可以会创建 `ClipRRectLayer` ，而新创建的 `ClipRRectLayer` 会通过 `appendLayer` 方法触发 `append` 操作添加为父 `Layer` 的子节点。

而非 `ContainerLayer` 一般不具备子节点，比如:

- `PictureLayer` 是用于绘制画面，Flutter 上的控件基本是绘制在这上面；
- `TextureLayer` 是用于外界纹理，比如视频播放或者摄像头数据；
- `PlatformViewLayer` 是用于 iOS 上 `PlatformView` 相关嵌入纹理的使用；

举个例子，控件绘制时的 `Canvas` 来源于 `PaintingContext` ， 而如下代码所示 `PaintingContext` 通过 `_repaintCompositedChild` 执行绘制后得到的 `Picture` 最后就是提交给所在的 `PictureLayer.picture` 。

```
void stopRecordingIfNeeded() {
    if (!_isRecording)
      return;
    _currentLayer.picture = _recorder.endRecording();
    _currentLayer = null;
    _recorder = null;
    _canvas = null;
  }
```

五、Layer 的内外兼修

了解完 `Layer` 是如何提交绘制后，接下来介绍的就是 `Layer` 的刷新和复用。

我们知道当 `RenderObject` 的 `isRepaintBoundary` 为 `ture` 时，Flutter Framework 就会自动创建一个 `OffsetLayer` 来“承载”这片区域，而 `Layer` 内部的画面更新一般不会影响到其他 `Layer` 。

那 `Layer` 是如何更新？这就涉及了 `Layer` 内部的 `markNeedsAddToScene` 和 `updateSubtreeNeedsAddToScene` 这两个方法。

如下代码所示， `markNeedsAddToScene` 方法其实就是把 `Layer` 内的 `_needsAddToScene` 标记为 `true` ；而 `updateSubtreeNeedsAddToScene` 方法就是遍历所有 child `Layer` ，通过递归调用 `updateSubtreeNeedsAddToScene()` 判断是否有 `child` 需要 `_needsAddToScene` ，如果是那就把自己也标记为 `true` 。

```
  @protected
  @visibleForTesting
  void markNeedsAddToScene() {
    // Already marked. Short-circuit.
    if (_needsAddToScene) {
      return;
    }

    _needsAddToScene = true;
  }

  @override
  void updateSubtreeNeedsAddToScene() {
    super.updateSubtreeNeedsAddToScene();
    Layer child = firstChild;
    while (child != null) {
      child.updateSubtreeNeedsAddToScene();
      _needsAddToScene = _needsAddToScene || child._needsAd
      child = child.nextSibling;
    }
  }
```

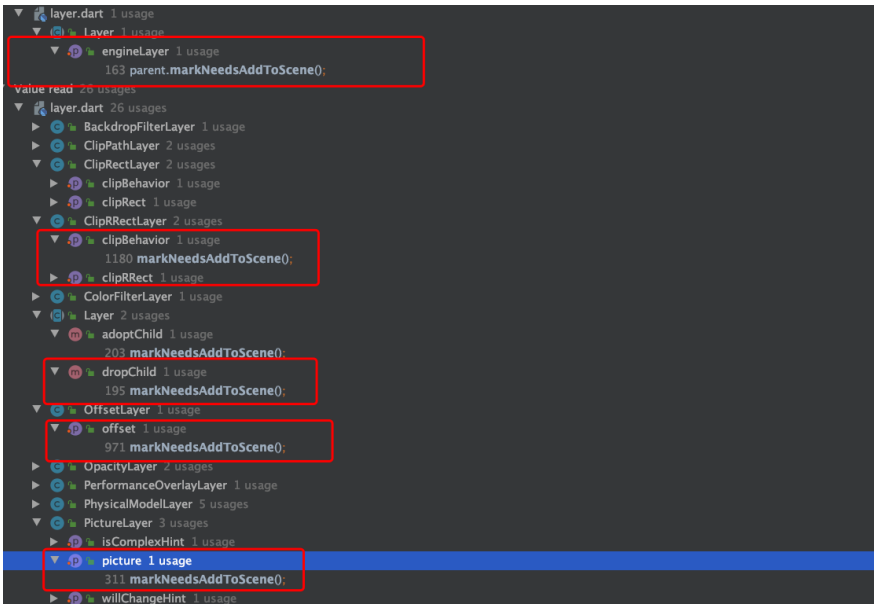
是不是和 `setState` 调用 `markNeedsBuild` 把自己标志为 `_dirty` 很像？**当 `_needsAddToScene` 等于 `true` 时，对应 `Layer` 的 `addToScene` 才会被调用；而当 `Layer` 的 `_needsAddToScene` 为 `false` 且 `_engineLayer` 不为空时就触发 `Layer` 的复用。**

```
void _addToSceneWithRetainedRendering(ui.SceneBuilder build

    if (!_needsAddToScene && _engineLayer != null) {
      builder.addRetained(_engineLayer);
      return;
    }
    addToScene(builder);

    _needsAddToScene = false;
  }
```

是的，当一个 `Layer` 的 `_needsAddToScene` 为 `false` 时 表明了自己不需要更新，那这个 `Layer` 的 `EngineLayer` 又存在，那 就可以被复用。举个例子：当一个新的页面打开时，底部的页面并没有发生变化时，它只是参与画面的合成，所以对于底部页面来说它“ `Layer` ”是可以直接被复用参与绘制。

那 `markNeedsAddToScene` 在什么时候会被调用？

如下图所示，当 `Layer` 子的参数，比如： `PictureLayer` 的 `picture` 、 `OffsetLayer` 的 `offset` 发生变化时， `Layer` 就会主动调用 `markNeedsAddToScene` 标记自己为“脏”区域。另外当 `Layer` 的 `engineLayer` 发生变化时，就会尝试触发父节点的 `Layer` 调用 `markNeedsAddToScene` ，这样父节点也会对应产生变化。




```
@protected
  set engineLayer(ui.EngineLayer value) {
    _engineLayer = value;

    if (!alwaysNeedsAddToScene) {
      if (parent != null && !parent.alwaysNeedsAddToScene)
        parent.markNeedsAddToScene();
      }
    }
  }
```

而 `updateSubtreeNeedsAddToScene` 是在 `buildScene` 的时候触发，在 `addToScene` 之前调用 `updateSubtreeNeedsAddToScene` 再次判断 child 节点，从而确定是否需要发生改变。

```
ui.Scene buildScene(ui.SceneBuilder builder) {
    List<PictureLayer> temporaryLayers;
    assert(() {
      if (debugCheckElevationsEnabled) {
        temporaryLayers = _debugCheckElevations();
      }
      return true;
    }());
    updateSubtreeNeedsAddToScene();
    addToScene(builder);

    _needsAddToScene = false;
    final ui.Scene scene = builder.build();

    return scene;
  }
```

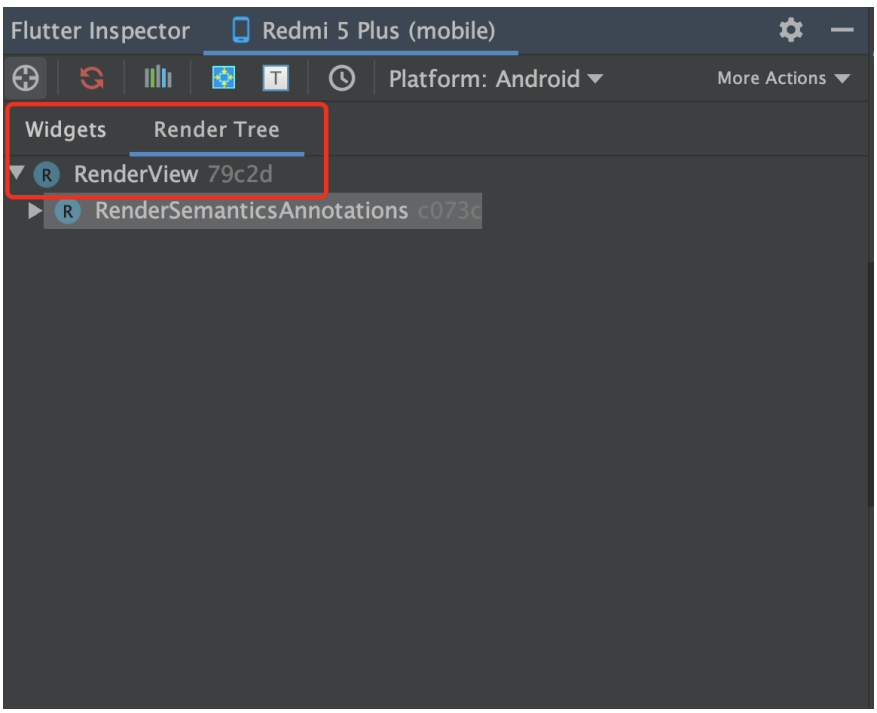
六、Flutter Framework 的 Layer 构成

最后回归到 Flutter Framework，在 Flutter Framework 中 `_window.render` 是在 `RenderView` 的 `compositeFrame` 方法中被调用；而 `RenderView` 是在 `RendererBinding` 的 `initRenderView` 被初始化；`initRenderView` 是在 `initInstances` 时被调用，也就是 `runApp` 的时候。

简单来说就是：**`runApp` 的时候创建了 `RenderView`，并且 `RenderView` 内部的 `compositeFrame` 就是通过 `_window.render` 来提交 `Layer` 的绘制。**

```
void compositeFrame() {
  Timeline.startSync('Compositing', arguments: timelineWh
  try {
    final ui.SceneBuilder builder = ui.SceneBuilder();
    final ui.Scene scene = layer.buildScene(builder);
    if (automaticSystemUiAdjustment)
      _updateSystemChrome();
    _window.render(scene);
    scene.dispose();
    assert(() {
      if (debugRepaintRainbowEnabled || debugRepaintTextR
        debugCurrentRepaintColor = debugCurrentRepaintCol
      return true;
    }());
  } finally {
    Timeline.finishSync();
  }
}
```

所以 `runApp` 的时候 Flutter 创建了 `RenderView`，并且在 `Window` 的 `drawFrame` 方法中调用了 `renderView.compositeFrame();` 提交了绘制，而 **`RenderView` 作为根节点，它携带的 `rootLayer` 为 `OffsetLayer` 的子类 `TransformLayer`，属于是 Flutter 中 `Layer` 的根节点。**



这里举个例子，如下图所示是一个简单的不规范代码，运行后出现的结果是一个黑色空白页面，这里我们通过 `debugDumpLayerTree` 方法打印出 `Layer` 的机构。

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    new Future.delayed(Duration(seconds: 1), () {
      debugDumpLayerTree();
    });
    return MaterialApp(
      title: 'GSY Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Container(),
      //routes: routers,
    );
  }
}
```

打印出的结果如下 LOG 所示，正如前面所说 `TransformLayer` 作为 `rooterLayer` 它的 `owner` 是 `RenderView`，然后它有两个 child 节点： child1 `OffsetLayer` 和 child2 `PictureLayer` 。

> 默认情况下因为 `Layer` 的形成机制（ `isRepaintBoundary` 为 `ture` 自动创建一个 `OffsetLayer` ）和 `Canvas` 绘制需要，至少会有一个 `OffsetLayer` 和 `PictureLayer` 。

```
I/flutter (32494): TransformLayer#f8fa5
I/flutter (32494):  │ owner: RenderView#2d51e
I/flutter (32494):  │ creator: [root]
I/flutter (32494):  │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ transform:
I/flutter (32494):  │   [0] 2.8,0.0,0.0,0.0
I/flutter (32494):  │   [1] 0.0,2.8,0.0,0.0
I/flutter (32494):  │   [2] 0.0,0.0,1.0,0.0
I/flutter (32494):  │   [3] 0.0,0.0,0.0,1.0
I/flutter (32494):  │
I/flutter (32494):  ├─child 1: OffsetLayer#4503b
I/flutter (32494):  │ │ creator: RepaintBoundary ← _FocusMa
I/flutter (32494):  │ │   ← PageStorage ← Offstage ← _Moda
I/flutter (32494):  │ │   _ModalScope<dynamic>-[LabeledGlob
I/flutter (32494):  │ │   ← _OverlayEntry-[LabeledGlobalKey
I/flutter (32494):  │ │   Stack ← _Theatre ←
I/flutter (32494):  │ │   Overlay-[LabeledGlobalKey<Overlay
I/flutter (32494):  │ │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ │
I/flutter (32494):  │ └─child 1: OffsetLayer#e8309
I/flutter (32494):  │     creator: RepaintBoundary-[GlobalK
I/flutter (32494):  │       FadeTransition ← FractionalTran
I/flutter (32494):  │       _FadeUpwardsPageTransition ← An
I/flutter (32494):  │       ← _FocusMarker ← Semantics ← Fo
I/flutter (32494):  │     offset: Offset(0.0, 0.0)
I/flutter (32494):  │
I/flutter (32494):  └─child 2: PictureLayer#be4f1
I/flutter (32494):      paint bounds: Rect.fromLTRB(0.0, 0.
```

根据上述 LOG 所示，首先看：

- `OffsetLayer` 的 `creator` 是 `RepaintBoundary`，而其来源是 `Overlay`，我们知道 Flutter 中可以通过 `Overlay` 做全局悬浮控件，而 `Overlay` 就是在 `MaterialApp` 的 `Navigator` 中创建，并且它是一个独立的 `Layer`；
- 而 `OffsetLayer` 的 child 是 `PageStorage`，`PageStorage` 是通过 `Route` 产生的，也即是默认的路由第一个页面。

所以现在知道为什么 `Overlay` 可以在 `MaterialApp` 的所有路由页面下全局悬浮显示了吧。

如下代码所示，再原本代码的基础上增加 `Scaffold` 后继续执行 `debugDumpLayerTree`。

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    new Future.delayed(Duration(seconds: 1), () {
      debugDumpLayerTree();
    });
    return MaterialApp(
      title: 'GSY Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        body: Container(),
      ),
      //routes: routers,
    );
  }
}
```

可以看到这里多了一个 `PhysicalModelLayer` 和 `PictureLayer` ， `PhysicalModelLayer` 是用于设置阴影等效果的，比如关闭 `debugDisablePhysicalShapeLayers` 后 `AppBar` 的阴影会消失，而之后的 `PictureLayer` 也是用于绘制。

```
I/flutter (32494): TransformLayer#ac14b
I/flutter (32494):  │ owner: RenderView#f5ecc
I/flutter (32494):  │ creator: [root]
I/flutter (32494):  │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ transform:
I/flutter (32494):  │   [0] 2.8,0.0,0.0,0.0
I/flutter (32494):  │   [1] 0.0,2.8,0.0,0.0
I/flutter (32494):  │   [2] 0.0,0.0,1.0,0.0
I/flutter (32494):  │   [3] 0.0,0.0,0.0,1.0
I/flutter (32494):  │
I/flutter (32494):  ├─child 1: OffsetLayer#c0128
I/flutter (32494):  │ │ creator: RepaintBoundary ← _FocusMa
I/flutter (32494):  │ │   ← PageStorage ← Offstage ← _Modal
I/flutter (32494):  │ │   _ModalScope<dynamic>-[LabeledGlob
I/flutter (32494):  │ │   ← _OverlayEntry-[LabeledGlobalKey
I/flutter (32494):  │ │   Stack ← _Theatre ←
I/flutter (32494):  │ │   Overlay-[LabeledGlobalKey<Overlay
I/flutter (32494):  │ │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ │
I/flutter (32494):  │ └─child 1: OffsetLayer#fb2a6
I/flutter (32494):  │   │ creator: RepaintBoundary-[GlobalK
I/flutter (32494):  │   │   FadeTransition ← FractionalTran
I/flutter (32494):  │   │   _FadeUpwardsPageTransition ← An
I/flutter (32494):  │   │   ← _FocusMarker ← Semantics ← Fo
I/flutter (32494):  │   │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │   │
I/flutter (32494):  │   └─child 1: PhysicalModelLayer#f1460
I/flutter (32494):  │     │ creator: PhysicalModel ← Animat
I/flutter (32494):  │     │   PrimaryScrollController ← _Sc
I/flutter (32494):  │     │   ← Builder ← RepaintBoundary-
I/flutter (32494):  │     │   FadeTransition ← FractionalTr
I/flutter (32494):  │     │ elevation: 0.0
I/flutter (32494):  │     │ color: Color(0xfffafafa)
I/flutter (32494):  │     │
I/flutter (32494):  │     └─child 1: PictureLayer#f800f
I/flutter (32494):  │         paint bounds: Rect.fromLTRB(0
I/flutter (32494):  │
I/flutter (32494):  └─child 2: PictureLayer#af14d
I/flutter (32494):      paint bounds: Rect.fromLTRB(0.0, 0.
I/flutter (32494):
```

最后通过再使用 `Navigator` 跳到另外一个页面，再新页面打印 `Layer` 树，可以看到又可以多了个 `PictureLayer` 、 `AnnotatedRegionLayer` 和 `TransformLayer` ： 其中多了的 `AnnotatedRegionLayer` 是用于处理新页面顶部状态栏的显示效果。

```
I/flutter (32494): TransformLayer#12e21
I/flutter (32494):  │ owner: RenderView#aa5c7
I/flutter (32494):  │ creator: [root]
I/flutter (32494):  │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ transform:
I/flutter (32494):  │   [0] 2.8,0.0,0.0,0.0
I/flutter (32494):  │   [1] 0.0,2.8,0.0,0.0
I/flutter (32494):  │   [2] 0.0,0.0,1.0,0.0
I/flutter (32494):  │   [3] 0.0,0.0,0.0,1.0
I/flutter (32494):  │
I/flutter (32494):  ├─child 1: OffsetLayer#fc176
I/flutter (32494):  │ │ creator: RepaintBoundary ← _FocusMa
I/flutter (32494):  │ │   ← PageStorage ← Offstage ← _Modal
I/flutter (32494):  │ │   _ModalScope<dynamic>-[LabeledGlob
I/flutter (32494):  │ │   ← _OverlayEntry-[LabeledGlobalKey
I/flutter (32494):  │ │   Stack ← _Theatre ←
I/flutter (32494):  │ │   Overlay-[LabeledGlobalKey<Overlay
I/flutter (32494):  │ │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │ │
I/flutter (32494):  │ └─child 1: OffsetLayer#b6e14
I/flutter (32494):  │   │ creator: RepaintBoundary-[GlobalK
I/flutter (32494):  │   │   FadeTransition ← FractionalTran
I/flutter (32494):  │   │   _FadeUpwardsPageTransition ← An
I/flutter (32494):  │   │   ← _FocusMarker ← Semantics ← Fo
I/flutter (32494):  │   │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │   │
I/flutter (32494):  │   └─child 1: PhysicalModelLayer#4fdc0
I/flutter (32494):  │     │ creator: PhysicalModel ← Animat
I/flutter (32494):  │     │   PrimaryScrollController ← _Sc
I/flutter (32494):  │     │   ClipDemoPage ← Semantics ← Bu
I/flutter (32494):  │     │   RepaintBoundary-[GlobalKey#0c
I/flutter (32494):  │     │   FadeTransition ← …
I/flutter (32494):  │     │ elevation: 0.0
I/flutter (32494):  │     │ color: Color(0xfffafafa)
I/flutter (32494):  │     │
I/flutter (32494):  │     ├─child 1: PictureLayer#6ee26
I/flutter (32494):  │     │   paint bounds: Rect.fromLTRB(0
I/flutter (32494):  │     │
I/flutter (32494):  │     ├─child 2: AnnotatedRegionLayer<S
I/flutter (32494):  │     │ │ value: {systemNavigationBarCo
I/flutter (32494):  │     │ │   systemNavigationBarDivider(
I/flutter (32494):  │     │ │   statusBarBrightness: Bright
I/flutter (32494):  │     │ │   Brightness.light, systemNav
I/flutter (32494):  │     │ │   Brightness.light}
I/flutter (32494):  │     │ │ size: Size(392.7, 83.6)
I/flutter (32494):  │     │ │ offset: Offset(0.0, 0.0)
I/flutter (32494):  │     │ │
I/flutter (32494):  │     │ └─child 1: PhysicalModelLayer#e
```

```
I/flutter (32494):  │     │   │  creator: PhysicalModel ← Ar
I/flutter (32494):  │     │   │    AnnotatedRegion<SystemUi(
I/flutter (32494):  │     │   │    FlexibleSpaceBarSettings
I/flutter (32494):  │     │   │    LayoutId-[ <_ScaffoldSlot.
I/flutter (32494):  │     │   │    AnimatedBuilder ← …
I/flutter (32494):  │     │   │  elevation: 4.0
I/flutter (32494):  │     │   │  color: MaterialColor(primar
I/flutter (32494):  │     │   │
I/flutter (32494):  │     │   └─child 1: PictureLayer#418ce
I/flutter (32494):  │     │       paint bounds: Rect.fromLT
I/flutter (32494):  │     │
I/flutter (32494):  │     └─child 3: TransformLayer#7f867
I/flutter (32494):  │       │  offset: Offset(0.0, 0.0)
I/flutter (32494):  │       │  transform:
I/flutter (32494):  │       │    [0] 1.0,0.0,0.0,-0.0
I/flutter (32494):  │       │    [1] -0.0,1.0,0.0,0.0
I/flutter (32494):  │       │    [2] 0.0,0.0,1.0,0.0
I/flutter (32494):  │       │    [3] 0.0,0.0,0.0,1.0
I/flutter (32494):  │       │
I/flutter (32494):  │       └─child 1: PhysicalModelLayer#9
I/flutter (32494):  │         │  creator: PhysicalShape ← _M
I/flutter (32494):  │         │    ConstrainedBox ← _FocusMa
I/flutter (32494):  │         │    Semantics ← RawMaterialBu
I/flutter (32494):  │         │    ← TickerMode ← Offstage ←
I/flutter (32494):  │         │  elevation: 6.0
I/flutter (32494):  │         │  color: Color(0xff2196f3)
I/flutter (32494):  │         │
I/flutter (32494):  │         └─child 1: PictureLayer#2a074
I/flutter (32494):  │             paint bounds: Rect.fromLT
I/flutter (32494):  │
I/flutter (32494):  └─child 2: PictureLayer#3d42d
I/flutter (32494):      paint bounds: Rect.fromLTRB(0.0, 0.
I/flutter (32494):
```

所以可以看到，Flutter 中的 `Widget` 在最终形成各式各样的 `Layer` ，每个 `Layer` 都有自己单独的区域和功能，比如 `AnnotatedRegionLayer` 在新的页面处理状态栏颜色的变化，而这些 `Layer` 最终通过 `SceneBuilder` 转化为 `EngineLayer` ，最后提交为 `Scene` 经由 Engine 绘制。

最后总结一下：**Flutter Framework 的 `Layer` 在绘制之前，需要经历 `SceneBuinlder` 的处理得到 `EngineLayer` ，其实 Flutter Framework 中的 `Layer` 可以理解为 `SceneBuinlder` 的对象封装，而 `EngineLayer` 才是真正的 Engine 图层，在之后得到的 `Scene` 会被提交 Engine 绘制。**

> 自此，第二十一篇终于结束了！(///▽///)

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

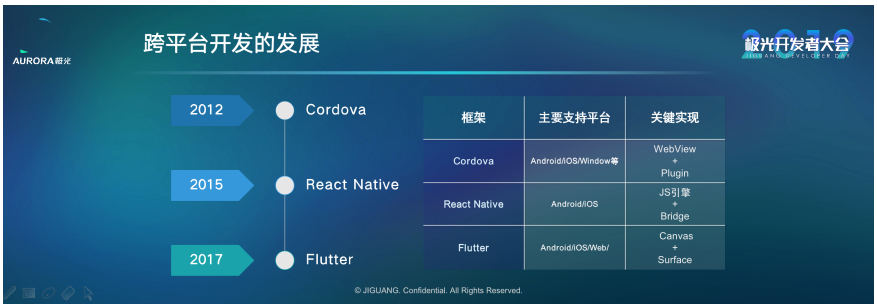
> 大家好，我是郭树煜，掘金 *《Flutter 完整开发实战详解》* 系列的作者，Github GSY 系列开源项目的维护人员，系列包括 GSYVideoPlayer 、 `GSYGitGithubApp` (Flutter \ ReactNative \ Kotlin \ Weex 四大版本)、GSYFlutterBook 电子书等，系列总 star 数在 25k 左右，目前 Github 中国区粉丝数暂居 67 名，主要负责移动端项目开发，大前端方向，主要涉及领域有 Android、Flutter、React Native、Weex 、小程序等等。

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。

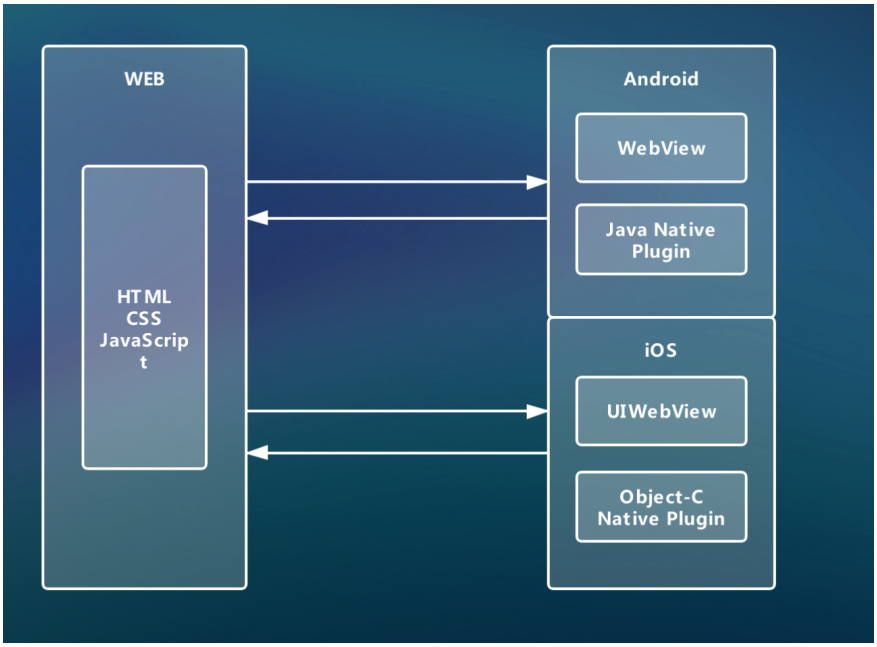
移动端的跨平台开发主要有三个阶段，这些阶段的代表框架主要有：`Cordova` 、 `React Native` 、 `Flutter` 等，如下图所示，是移动端的跨平台发展历程：



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。



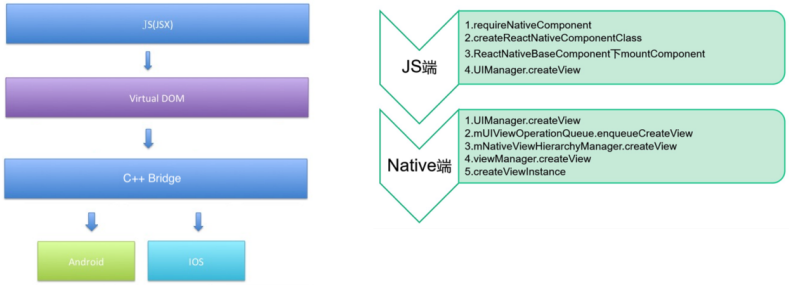
`Cordova` 让前端开发人员可以快速的构建移动应用，获取平台入口，对早期 web 上欠缺的如**摄像机、本地缓存、文件读写**等能力进行快速支持。

> 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。



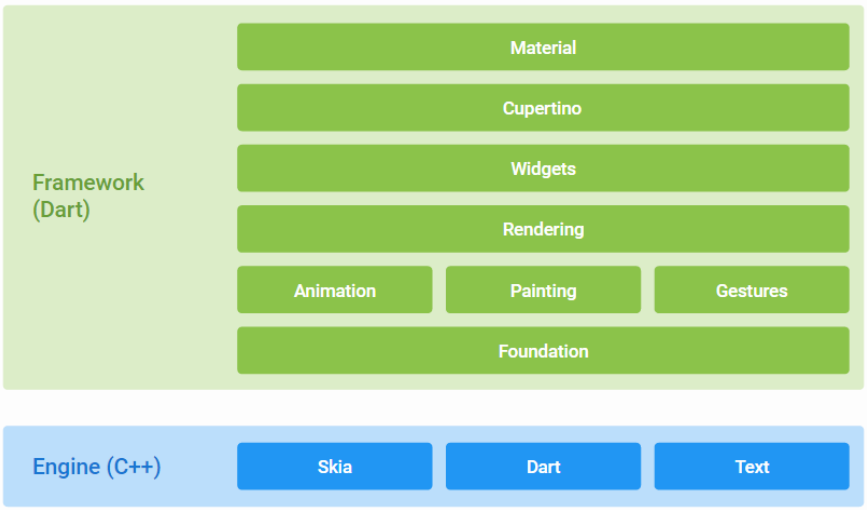
由于 `React` 框架的盛行， `React Native` 也开始成为 `React` 开发人员，将自身能力拓展到应用开发的最佳选择之一。同时 `React Native` 也是应用开发人员，接触前端的不错尝试。

> 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

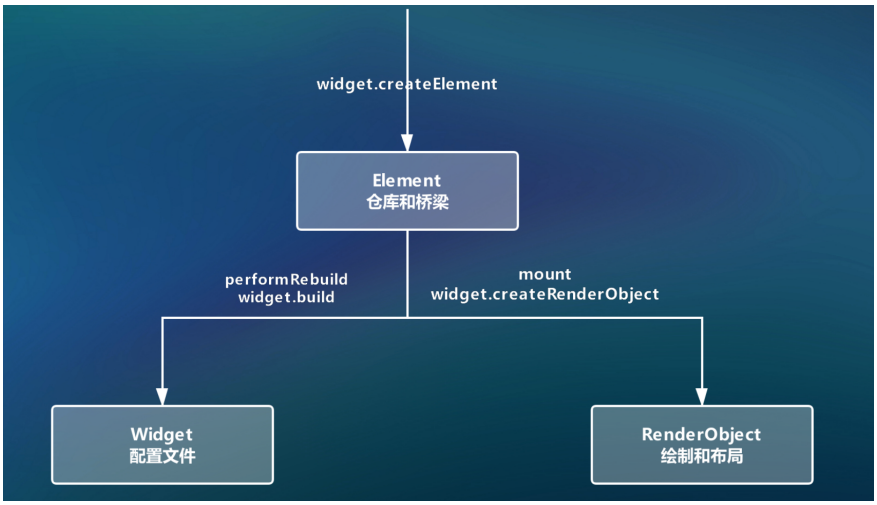
Flutter 是 UI 框架，Flutter 内一切皆 `Widget` ，每个 `Widget` 状态都代表了一帧， `Widget` 是不可变的。 那么 `Widget` 是怎么工作的呢?

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。

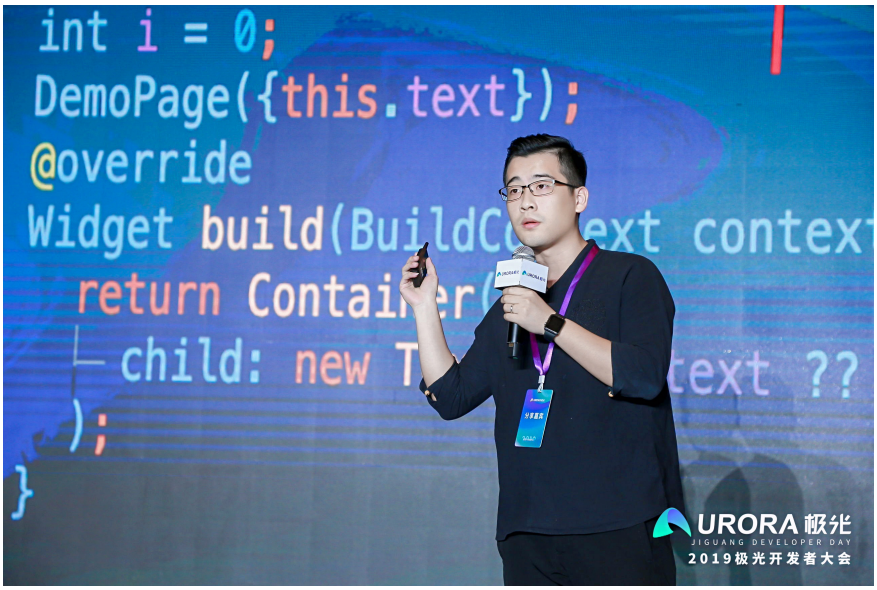
- `RenderObject` 是解析后的绘制和布局。

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。

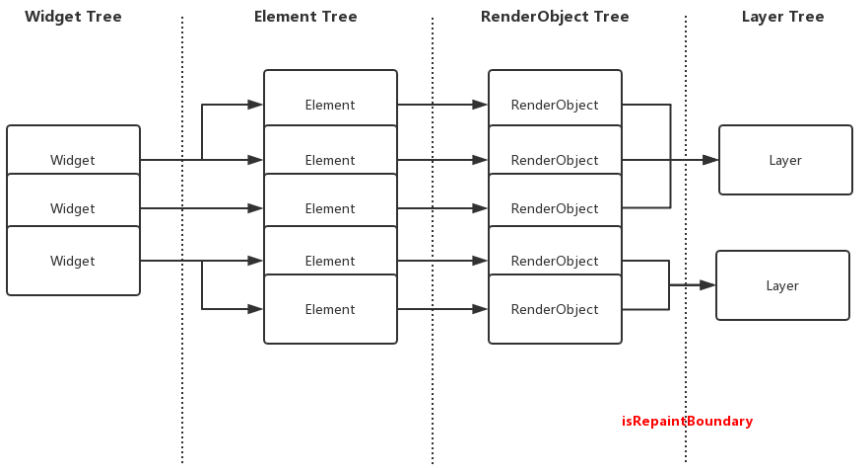
结合上面的理解，可以知道 `Widget` 大部分时候，其实只是轻量级的配置，对于性能问题，**你更需要关心的是 `Clip` 、 `Overlay` 、透明合成等行为，因为它们会需要产生 `saveLayer` 的操作，因为 `saveLayer` 会清空GPU绘制的缓存。**

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。

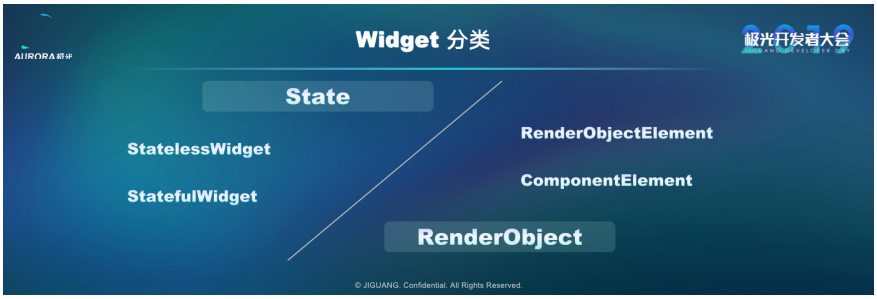
当 `isRepaintBoundary` 为 `true` 时，该区域就是一个可更新绘制区域，而当这个区域形成时，就会新创建一个 `Layer` 。 但不是每个 `RenderObject` 都会有 `Layer` ， 因为这受 `isRepaintBoundary` 的影响。



> 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"),
    );
  }
}
```

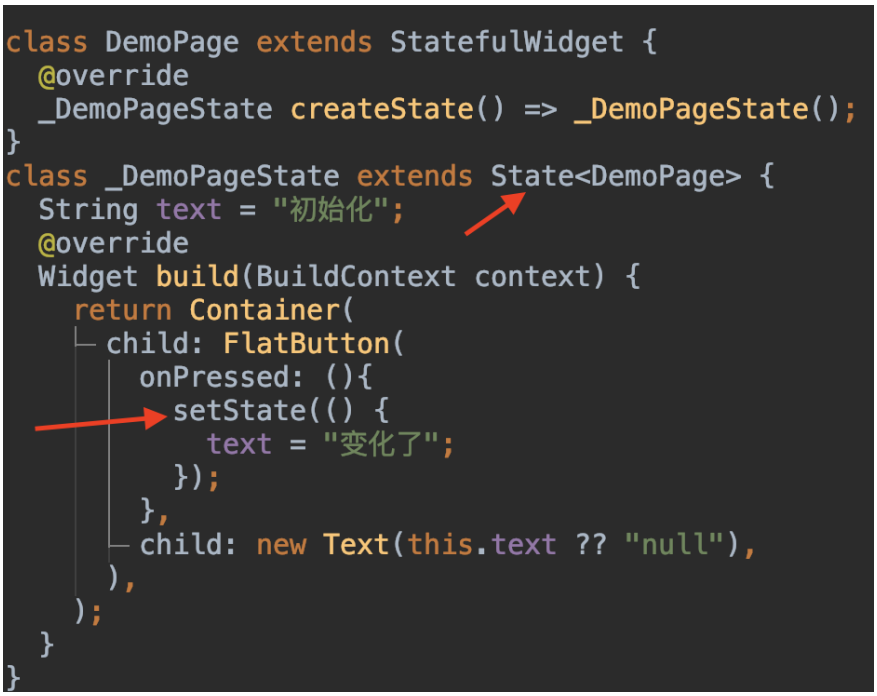
> 注意图中 `DemoPage` 有个黄色警告，这是因为我们定义了 `int i = 0` 不是 final 导致的，在 `StatelessWidget` 中， **非 final 的变量起始容易产生误解，因为 `Widget` 本事就是不可变的。**

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。**



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` 。

```dart
/// 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。

```dart
@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;
}
```

File is not writable

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。

```dart
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：

```dart
@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`；

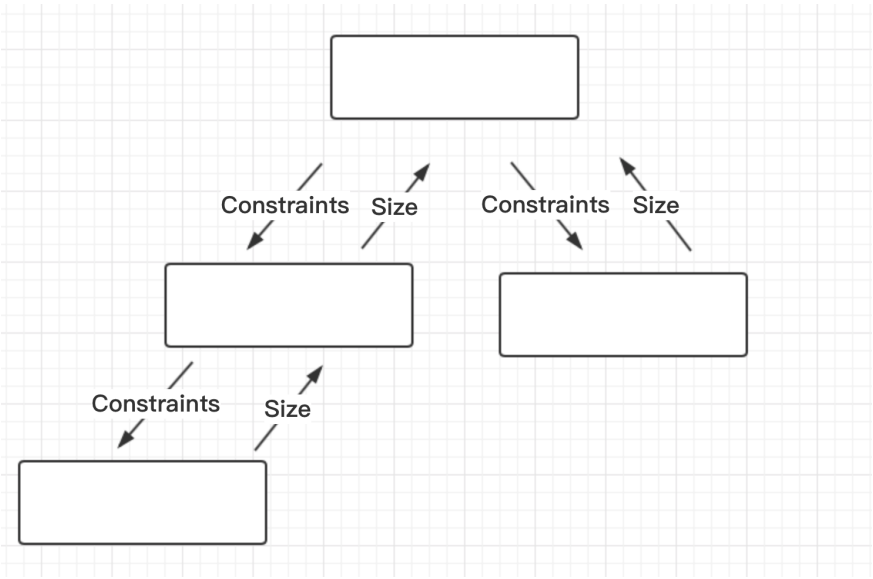
前面 `maxMainSize` 我们知道了是父布局的最大宽度，而 `allocatedSize` 其实就是 child 的宽度之和。所以结果很明显了：

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` 。

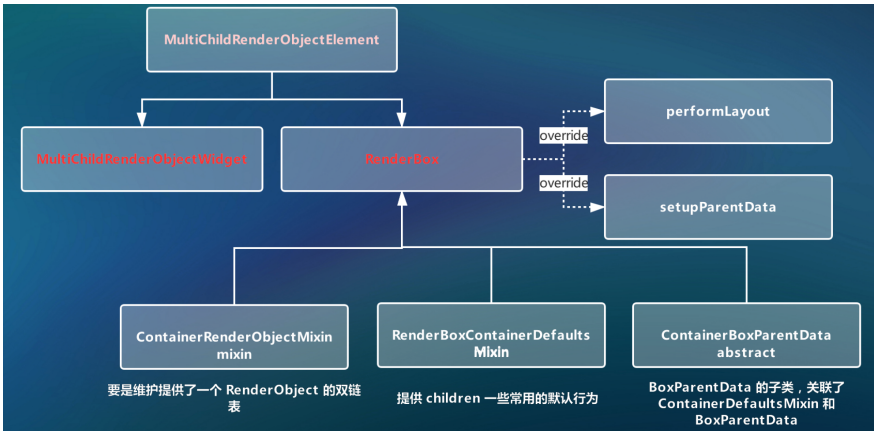


那如何直接自定义 `RenderObject` 布局?

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。

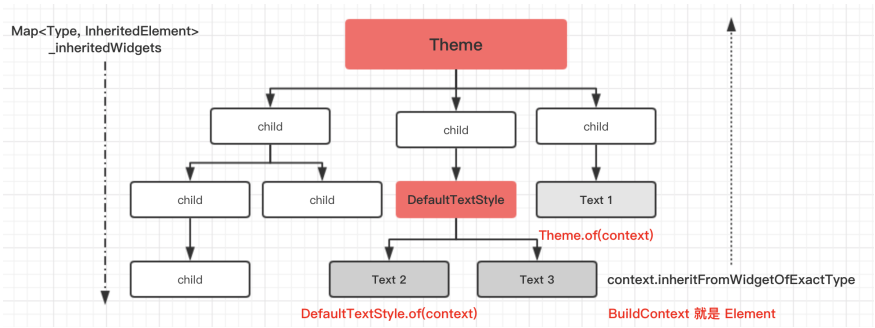
`InheritedWidget` 共享的是 `Widget` ，只是这个 `Widget` 是一个 `ProxyWidget` ，它自己本身并不绘制什么，但共享这个 `Widget` 内保存有的数据，从而到了共享状态的目的。

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?

```dart
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

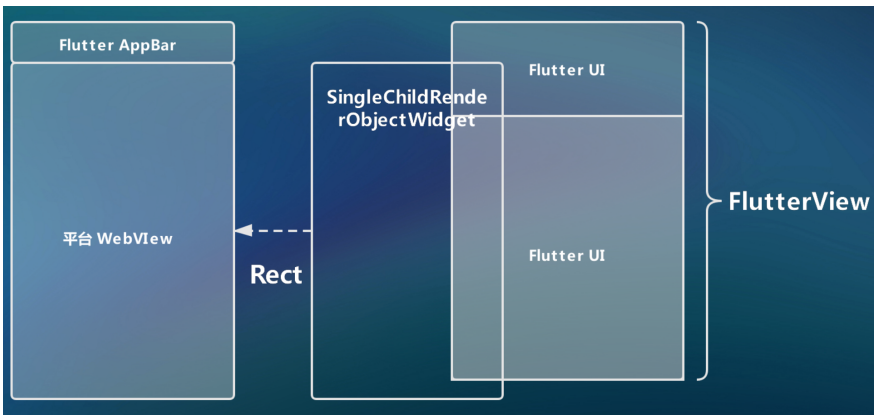
前面我们说过， Flutter 既然不依赖于原生控件，那么如何集成一些平台已有的控件呢？比如 `WebView` 和 `Map` ？

我们这里以 `WebView` 为例子：

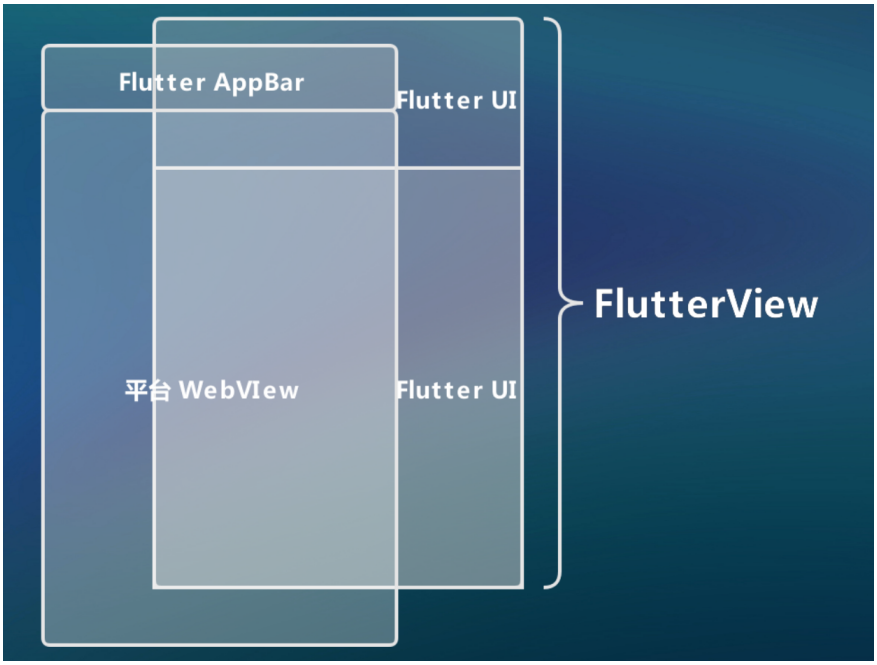
在官方 `WebView` 控件支持出来之前，第三方是直接在 FlutterView 上覆盖了一个新的原生控件，利用 Dart 中的占位控件**传递位置和大小**。

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。



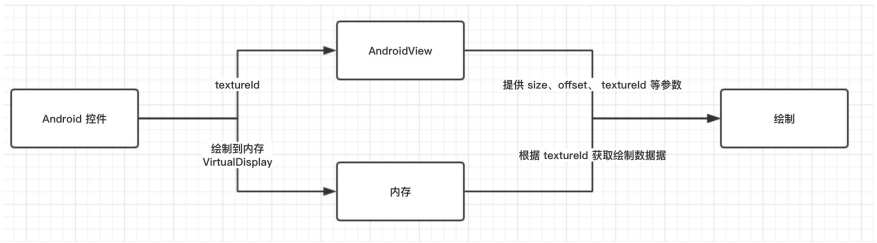
这样看起来就像是在 Flutter 中添加了 `WebView` ，但实际这脱离了 Flutter 的渲染树，其中一个问题就是，当你跳转 Flutter 其他页面的时候，会被 `WebView` 挡住；并且打开页面的动画， `Appbar` 和 `WebView` 难以保持一致。



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。

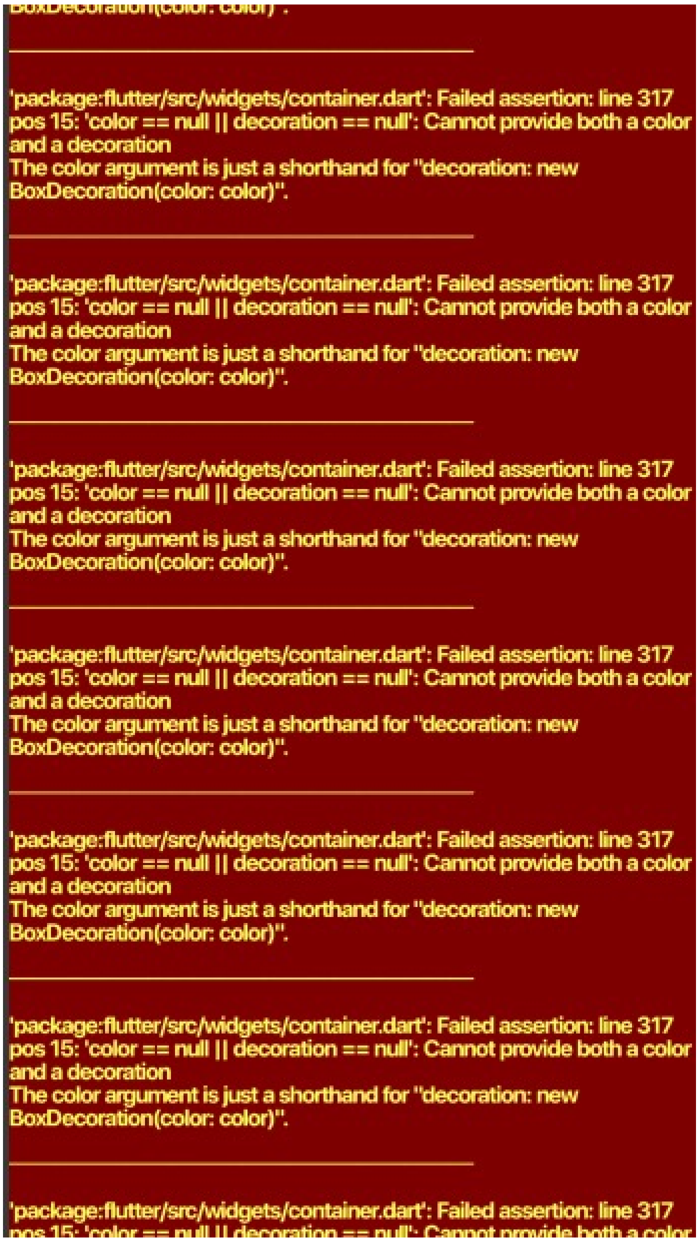
以 Android 为例，Android 上是利用了副屏显示的底层逻辑，使用 `VirtualDisplay` 类，创建一个虚拟显示器，需要调用 `DisplayManager` 的 `createVirtualDisplay()` 方法，将虚拟显示器的内容渲染在一个内存的 `Surface` 上，生成一个唯一的 `textureId` 。

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。



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);
  });
}
```

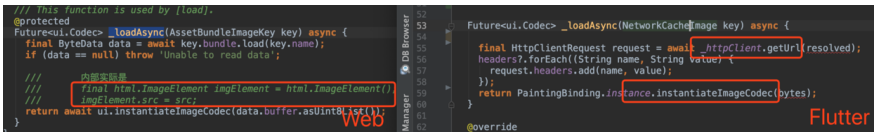
重写 `ErrorWidget` 的 `builder` 方法，然后将信息收集到 `Zone` 中，返回自己的自定义错误显示，最后在 `Zone` 内利用 `onError` 统一处理错误。

> 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。



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

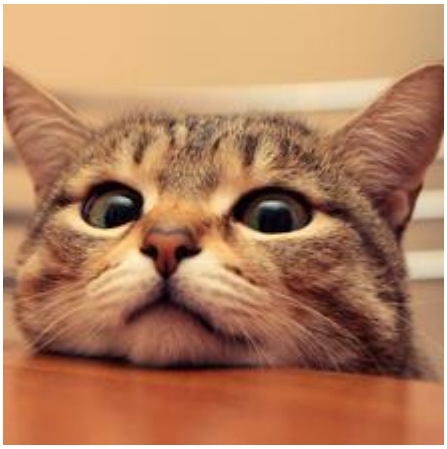
- Github： https://github.com/CarGuo
- **开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter**
- **开源 Flutter 多案例学习型项目：https://github.com/CarGuo/GSYFlutterDemo**
- **开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook**
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp

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

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：

<	+	|	[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
–	%	>>	

- 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("VVVVVVVVVVVVVVVVVVVVVVVVVV");
  print(result);
}

///返回得到一个 ZoneUnaryCallback
var onData = Zone.current.registerUnaryCallback<dynamic, in

///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)》

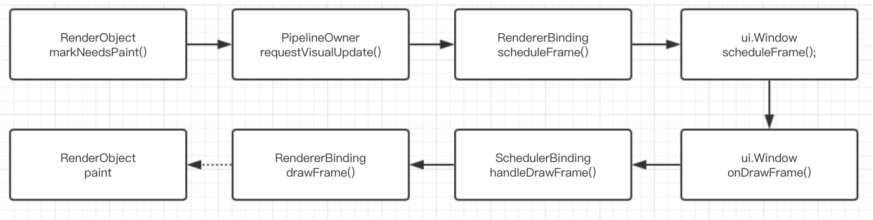
Flutter 部分

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。**

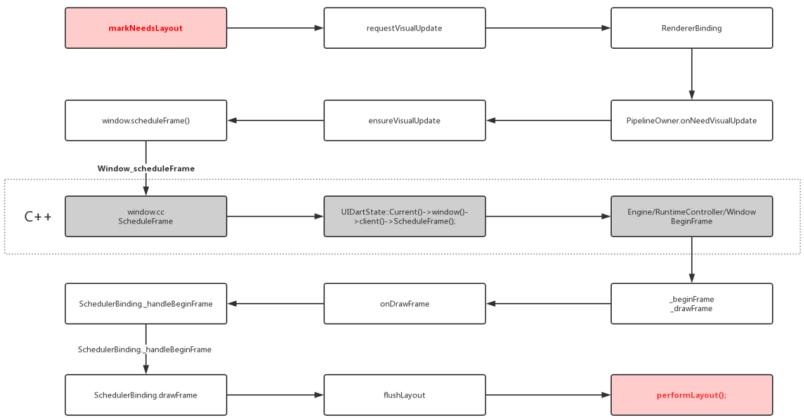
> 更多相关可查阅 《Flutter完整开发实战详解(九、 深入绘制原理)》

- 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);`

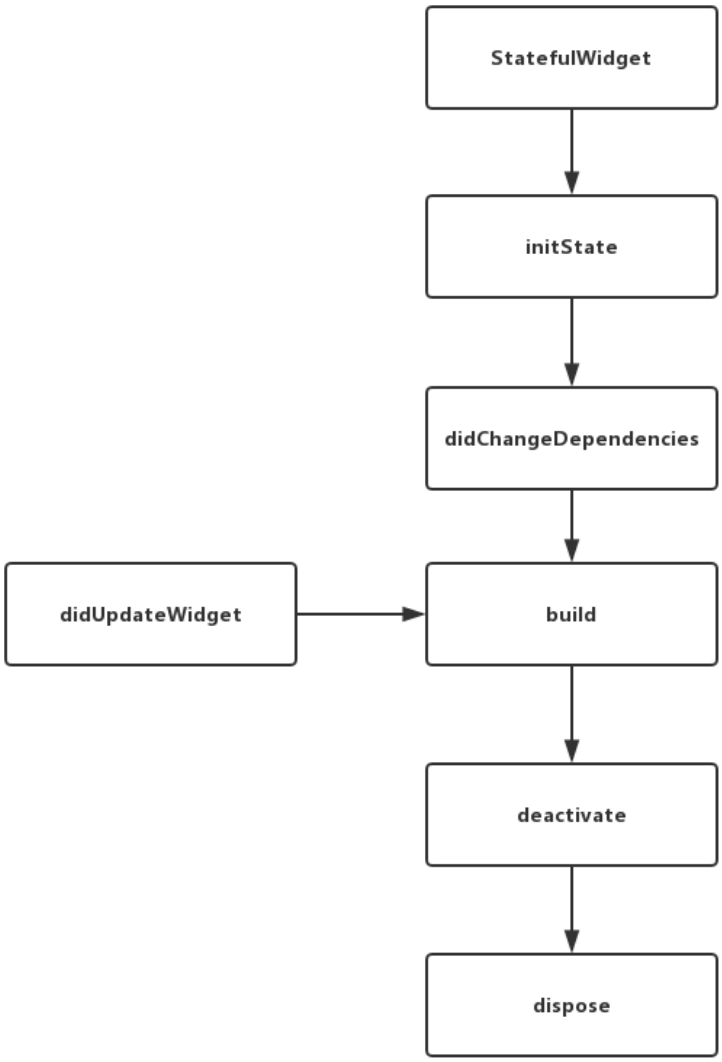
> 在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement> _inheritedWidgets` 的对象。
>
> `_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并 。
>
> 所以当我们通过 `context` 调用 `inheritFromWidgetOfExactType` 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。

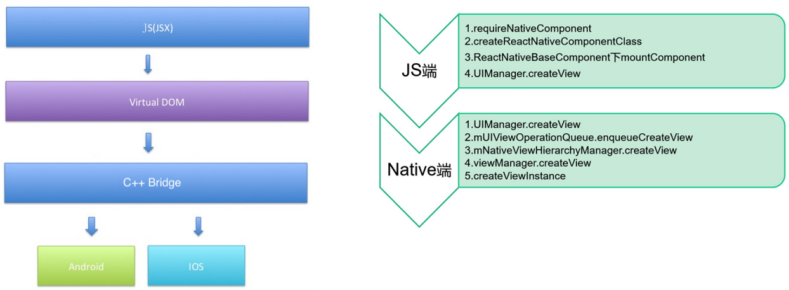
二、实现原理

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。



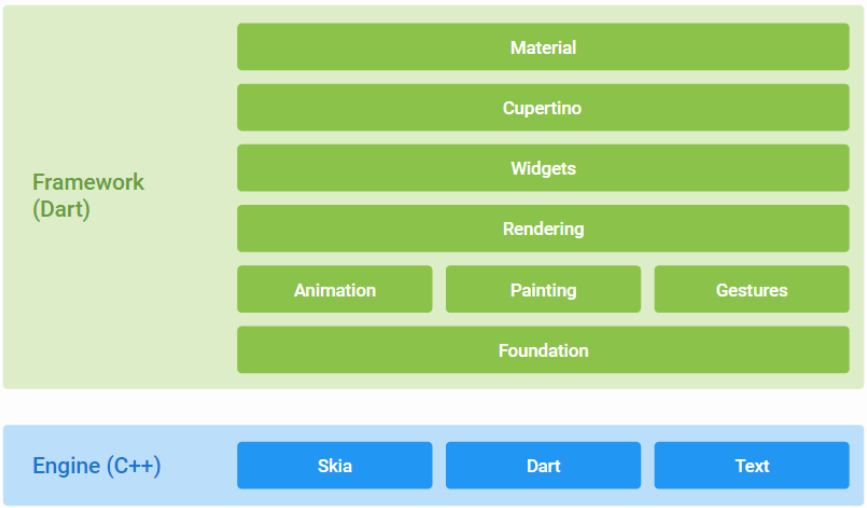
所以相较于如 `Ionic` 等框架而言， **React Native** 让页面的性能能得到进一步的提升。

- **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);
  }
```

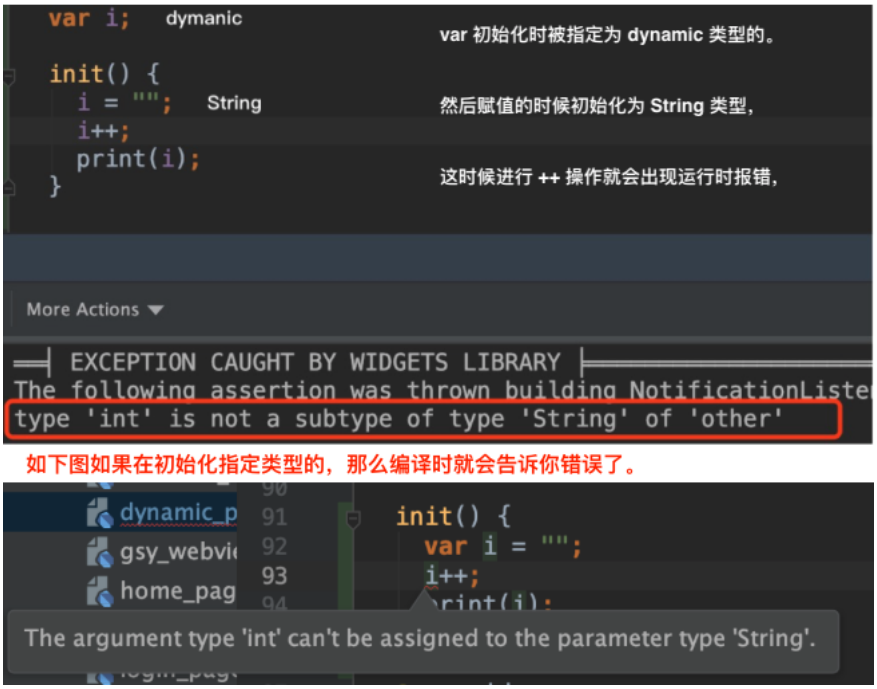
但是它们之间的差异性也很多，而最大的区别就是： **JS 是动态语言，而 Dart 是伪动态语言的强类型语言。**

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。**



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：

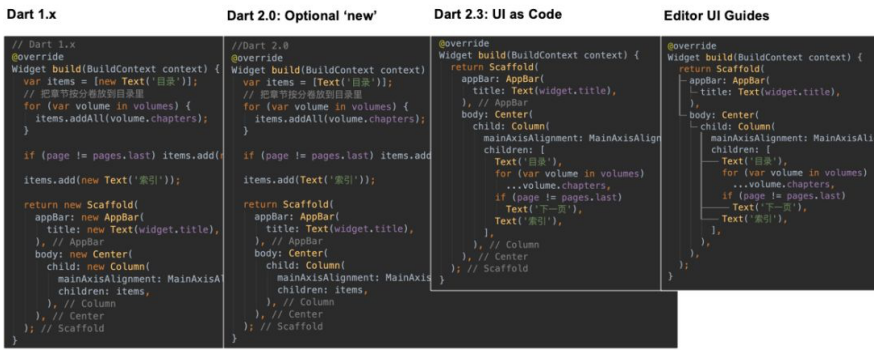
```
Widget build(BuildContext context) {
  var items = [Text('目录')];
  // 把章节按分卷放到目录里
  for (var volume in volumes) {
    items.addAll(volume.chapters);
  }

  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。



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。**

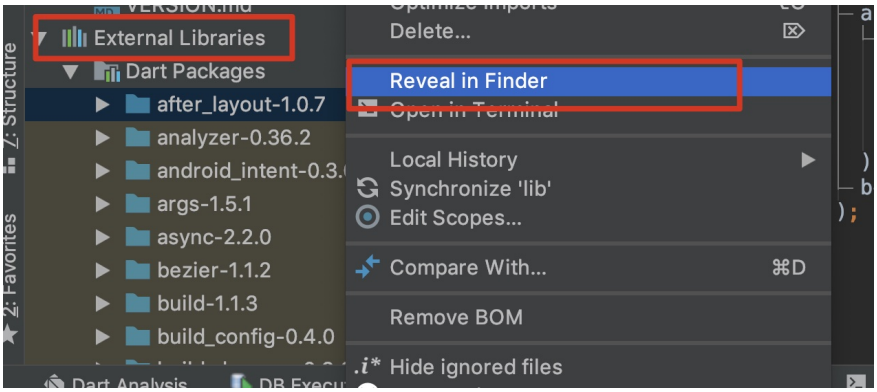
React Native 使用 *npm* 插件的好处就是：可以使用丰富的 *npm* 插件生态，同时减少前端开发者的学习成本。

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。

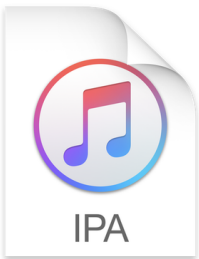
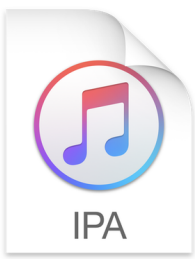
在 IOS 主要是 **App.framework** ，其内部也包含了 `kDartVmSnapshotData` 、 `kDartVmSnapshotInstructions` 、 `kDartIsolateSnapshotData` 、 `kDartIsolateSnapshotInstructions` 四个部分。

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 – 7.3 MB	 app-release.apk Zip Archive – 5.2 MB
空项目 IOS	 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	 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	 GSYGithubAPP.ipa iOS 应用 – 9.1 MB	 GSYGithubAppFlutter.ipa iOS 应用 – 28.6 MB

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。

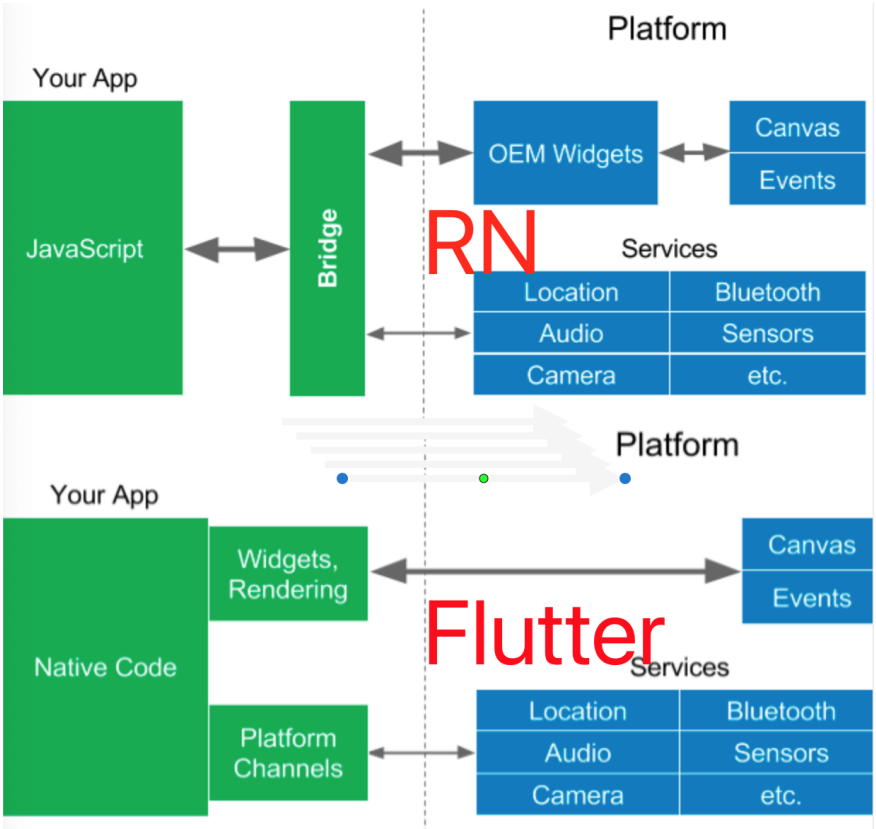
至于 **Flutter**，在打包时通过指定 `flutter build apk --release --target-platform android-arm64` 即可。

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

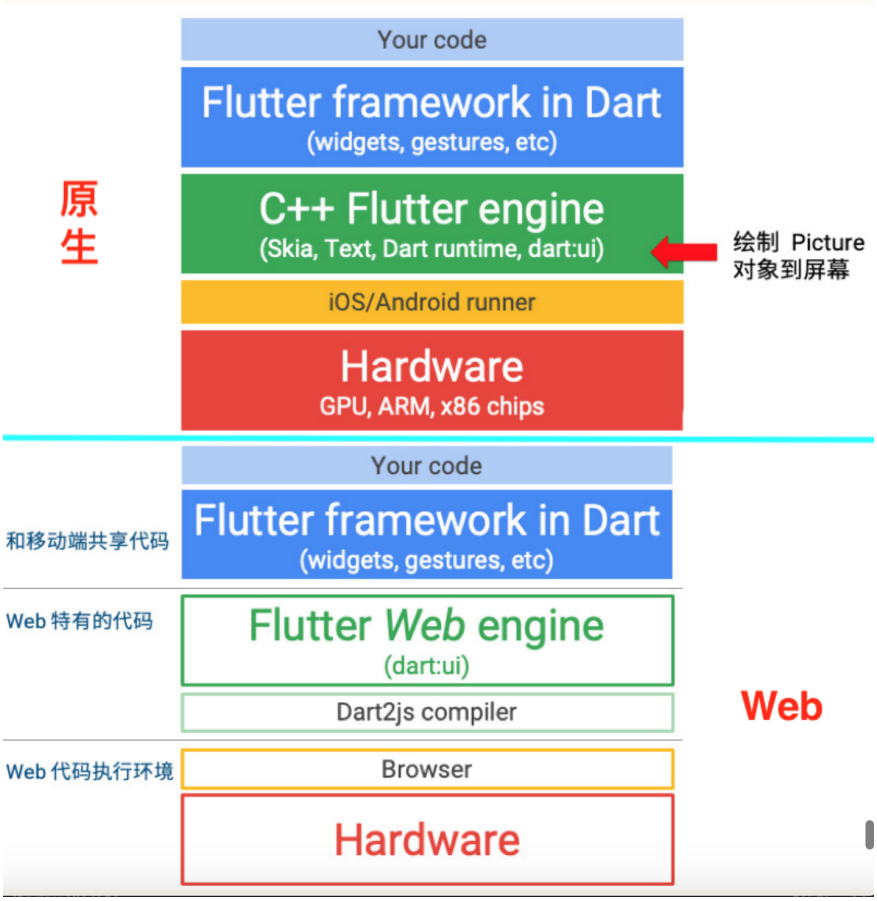
之前一篇 《为什么 Airbnb 放弃了 React Native?》 文章，让众多不明所以的吃瓜群众以为 **React Native** 已经被放弃，之后官方发布的《Facebook 正在重构 React Native，将重写大量底层》 公示，又再一次稳定了军心。

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。

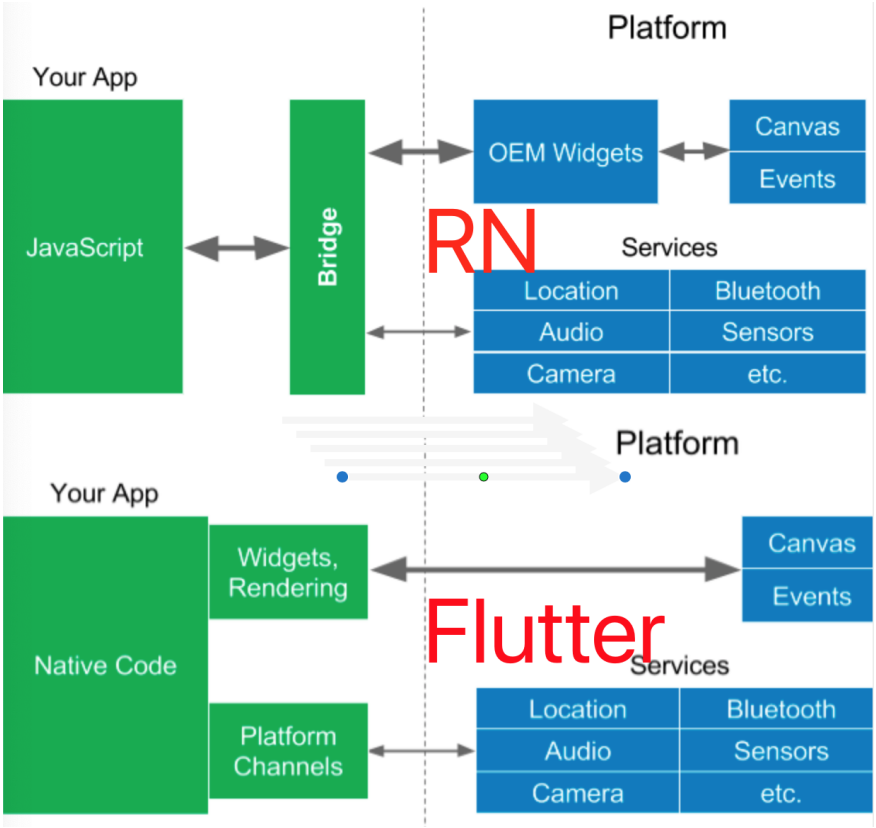
与我而言，**跨平台的意义在于解决的是端逻辑的统一**，至少避免了逻辑重复实现，或者 `IOS` 和 `Android` 之间争论 *谁对谁错* 的问题，甚至可以统一到 web 端等等。

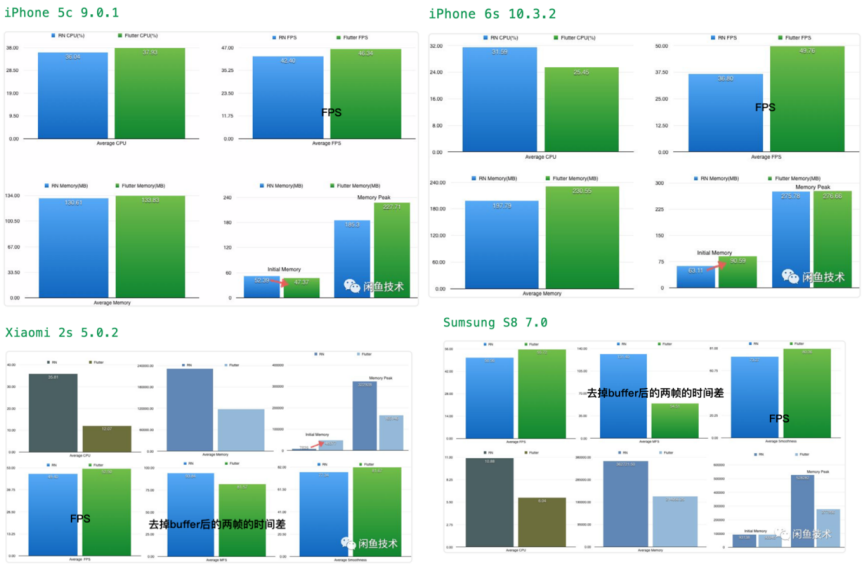
2、React Native 和 Flutter 之间的对比

`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** 。





> 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

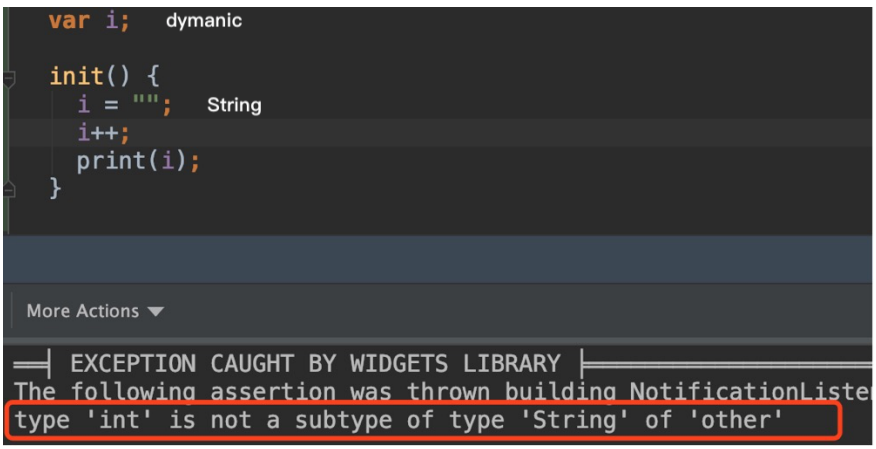
1、Dart 中有意思的一些东西

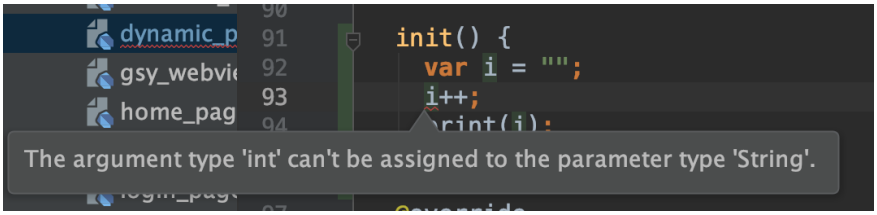
1.1、var 的语法糖和 dynamic

`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。





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");
  };
```

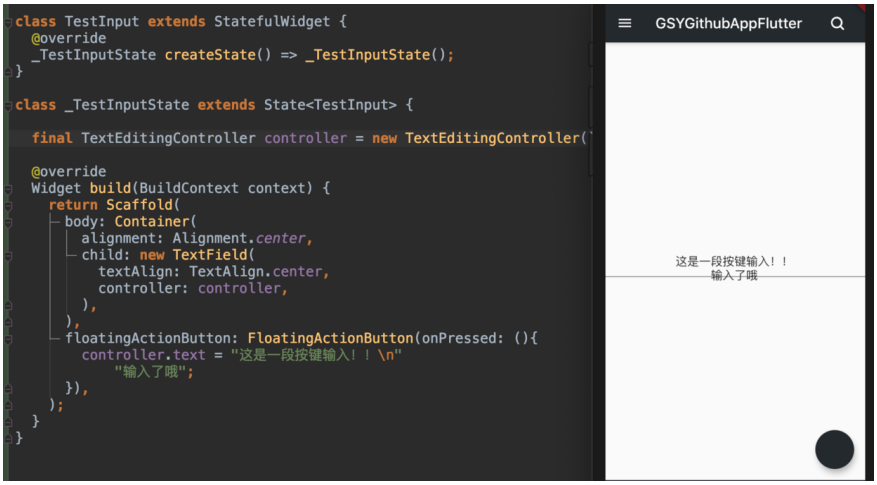
就可以通过直接执行和判空执行处理

```
 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?

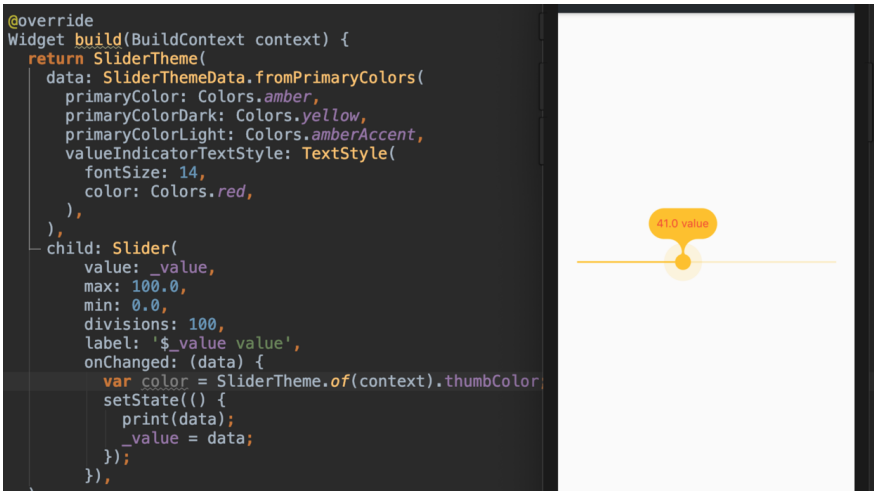


如下图所示，这是因为 `TextEditingController` 它是 `ChangeNotifier` 的子类，而 `TextField` 的内部对其进行了 `addListener`，同时我们改变值的时候调用了 `notifyListener`，触发内部 `setState`。

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`。

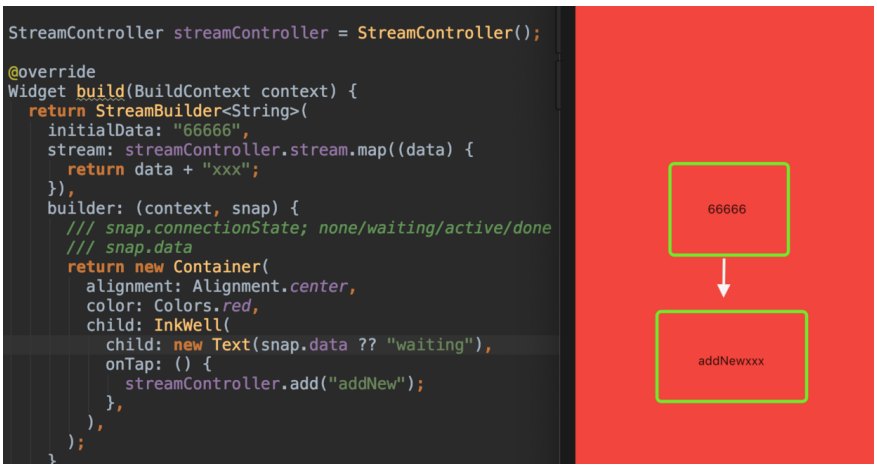


在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement> _inheritedWidgets` 的对象。

`_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?

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。

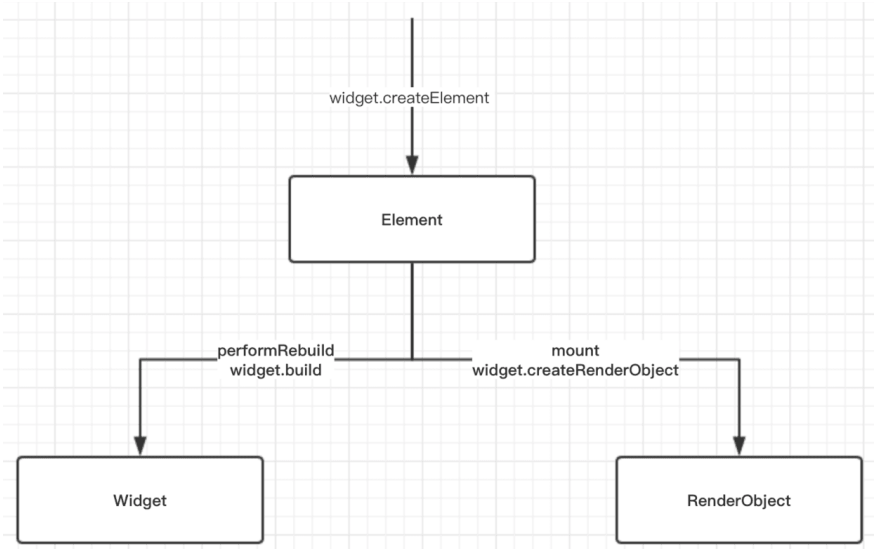
```dart
/// An [Element] that uses a [StatefulWidget] as its configuration.
class StatefulElement extends ComponentElement {
  /// Creates an element that uses the given widget as its configuration.
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        super(widget)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

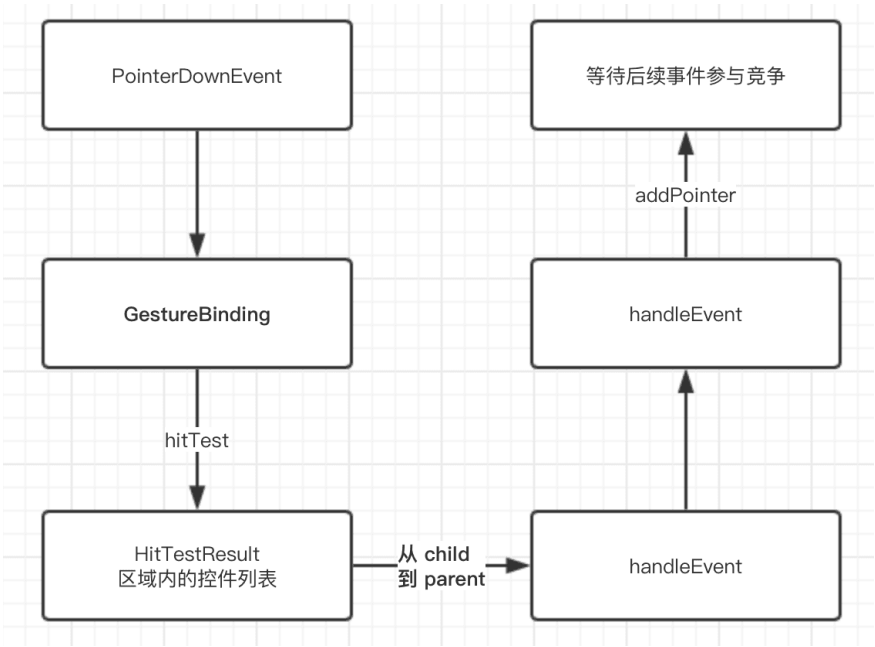
- **`Widget`** ： 就是我们平常写的控件， `Flutter` 宇宙中万物皆 `Widget` ，它们都是不可变一帧，同时也是被人吐槽很多的嵌套模式，当然换个角度，事实上你把他当作 `Widget` 配置文件来写或者就好理解了。
- **`Element`** ： 它是 `BuildContext` 的实现类， `Widget` 实现跨帧保存的 `state` 就是存放在这里，同时它也充当了 `Widget` 和 `RenderObject` 之间的桥梁。
- **`RenderObject`** ： 它才是真正干活（layout、paint）等，同时它才是真实的 “dom” 。
- **`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？

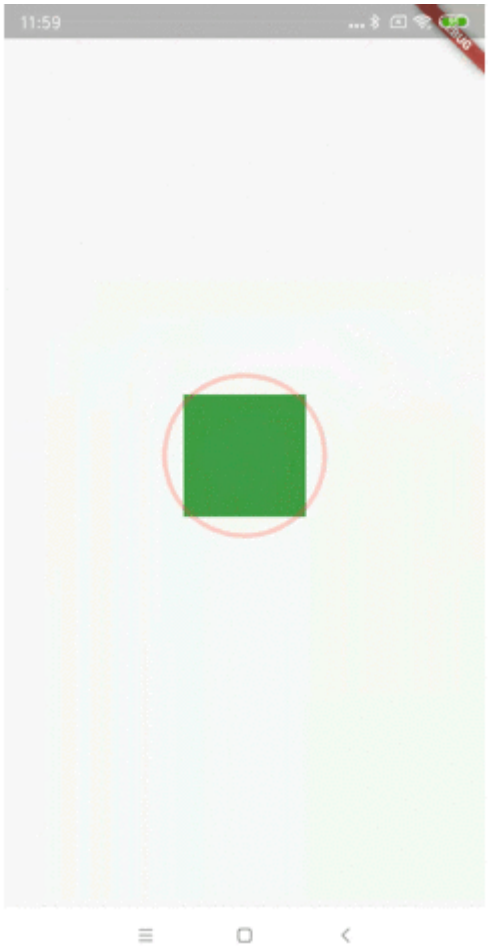
我们先看一段代码，然后这段代码执行的效果如下图2所示。

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 。

```dart
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.
    );
  }
}
```

```dart
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。

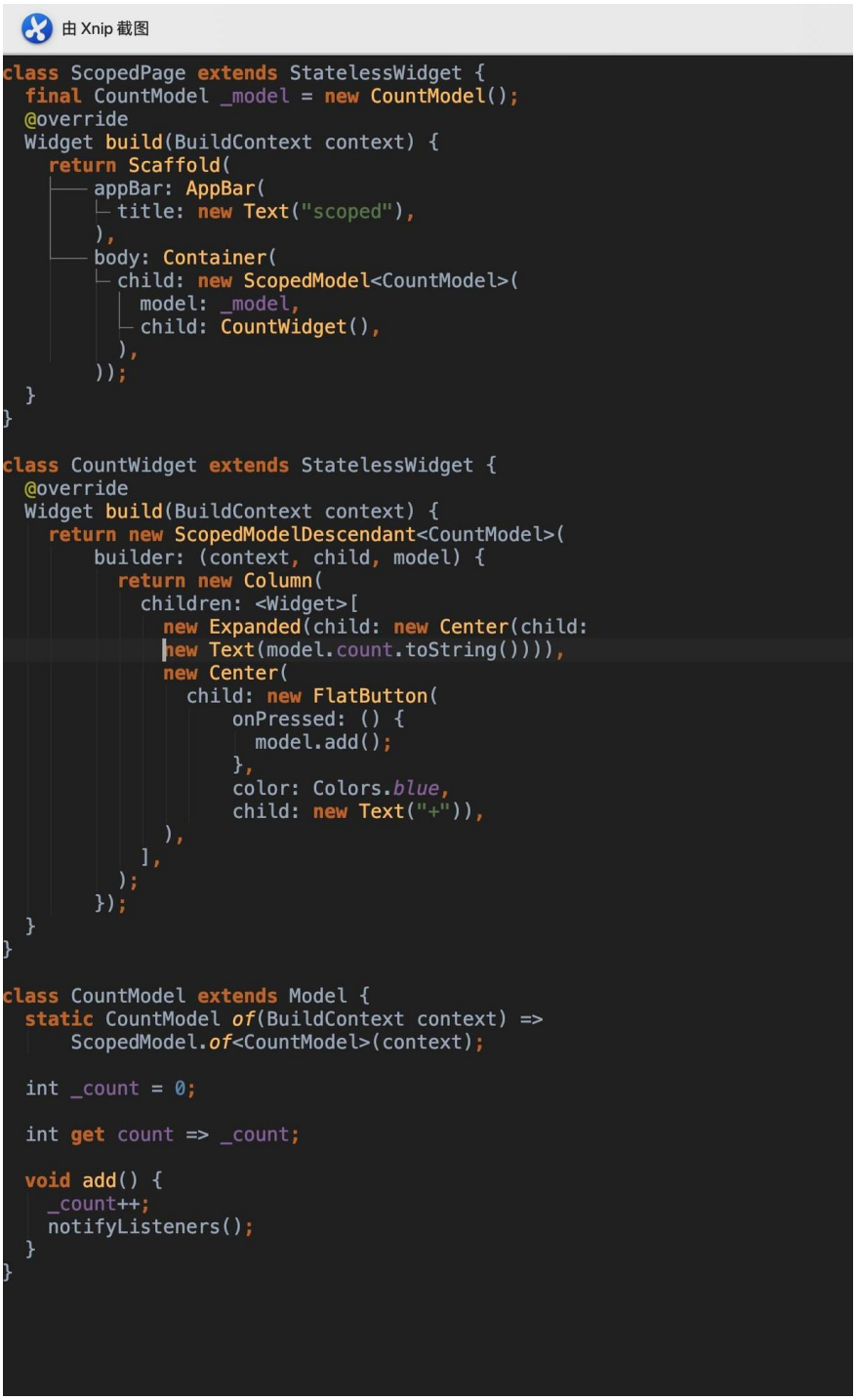
> 动画后的控件的点击区域，和你的动画数据改变的是 paint 还是 layout 有关 。

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。

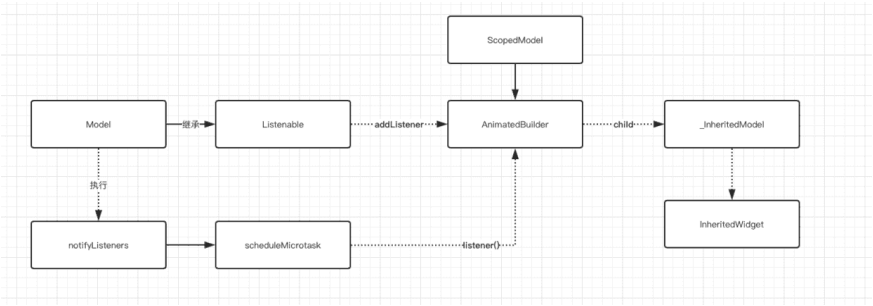
如下图是使用代码，在前面我们知道，状态管理使用的是 `InheritedWidget` 实现共享的，而当我们对 `Model` 进行数据改变时，通过调用 `notifyListeners` 通知页面更新了。



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：


```
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')
        }
    }

}
```

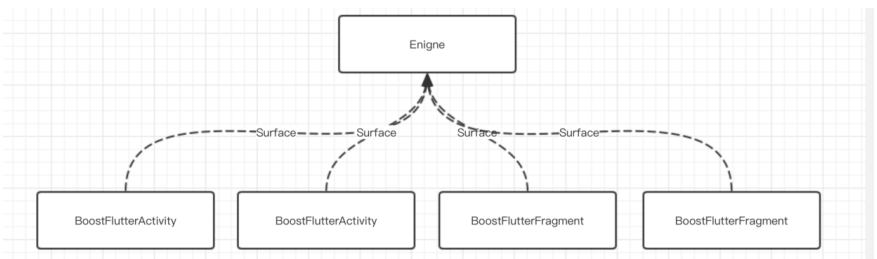
> **官方未来将有 `Flutter build aar` 的方法可提供使用。**

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。

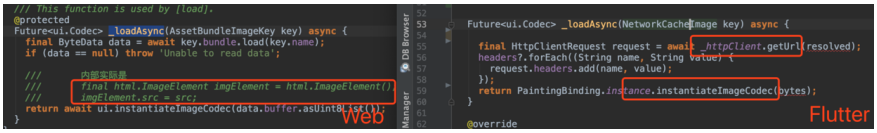
> 这部分因为之前以前聊过，就不赘述了

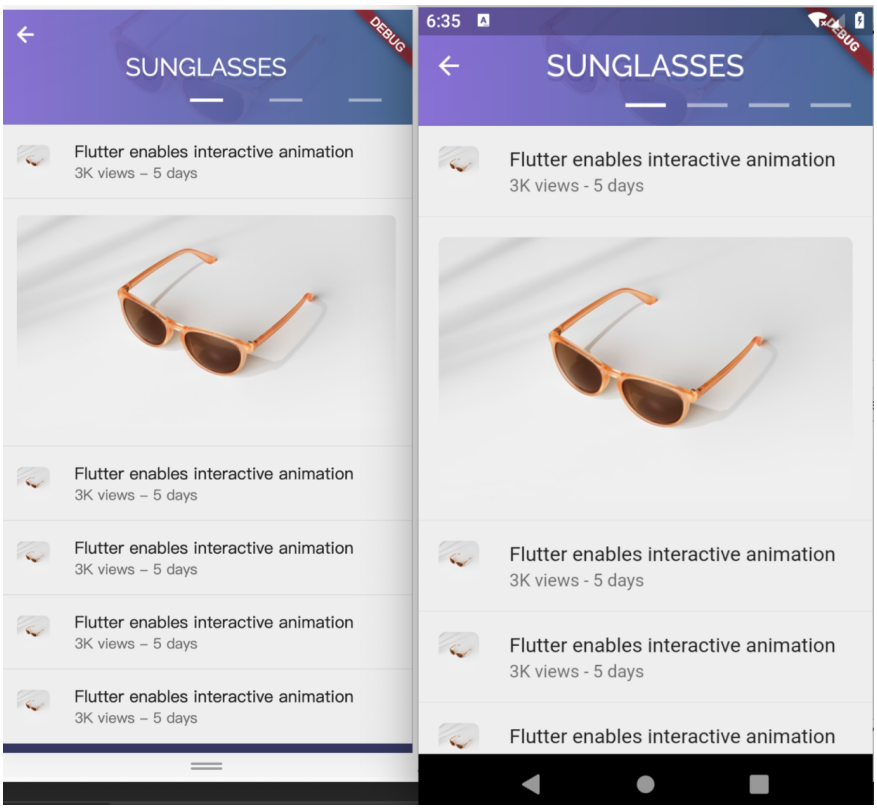
三、Flutter Web

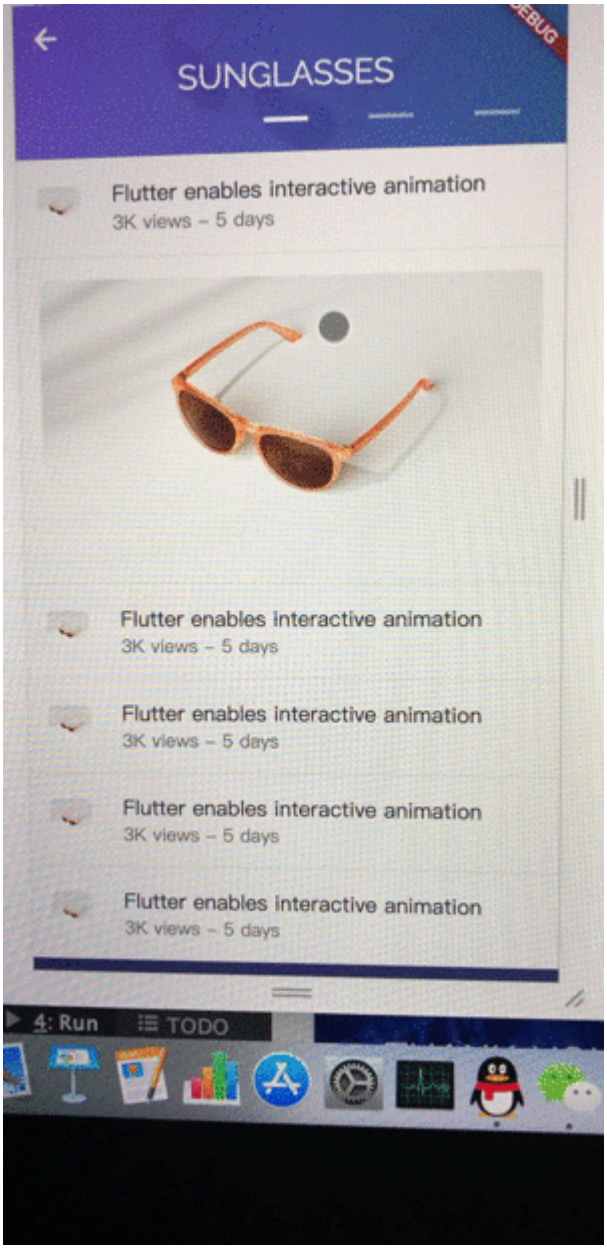
RN因为是原生控件，所以在react 和react native 整合这件事上存在难度。

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。







Flutter 的平台无关能力能带来什么?

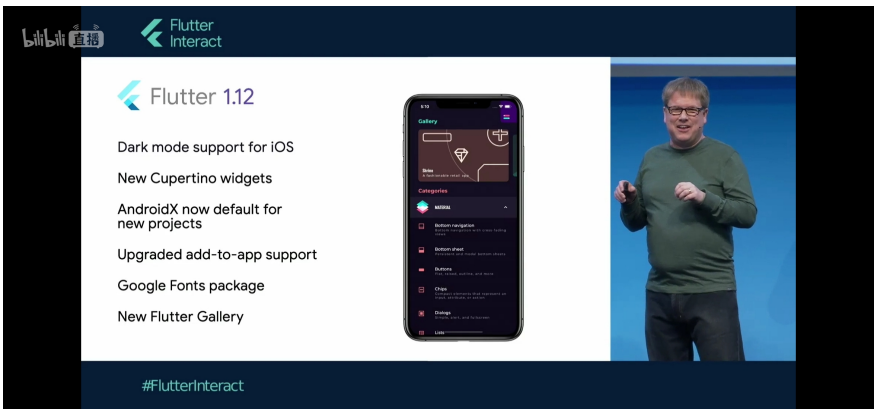
- 1、某些功能页面，可以一套代码实现，利用插件安装引入，在web、移动app、甚至 pc 上，都可以编译出对应平台的高性能代码，而不会像 Weex 等一样存在各种兼容问题。
- 2、在应用上可以快速实现“降级策略”，比如某种情况下应用产生奔溃了，可以替换为同等 UI 的 h5 显示，而这些代码只需要维护一份。

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

昨天谷歌为在 `Flutter Interact` 上为我们带来了 `Flutter 1.12` ，这是 1.9.x 的版本在经历 6 次 `hotfix` 之后，才带来的 stable 大版本更新。**该版本解决了 4,571 个报错，合并了 1,905 份 pr，同时本次发布也是 Flutter 一年内的第五个稳定版本。**

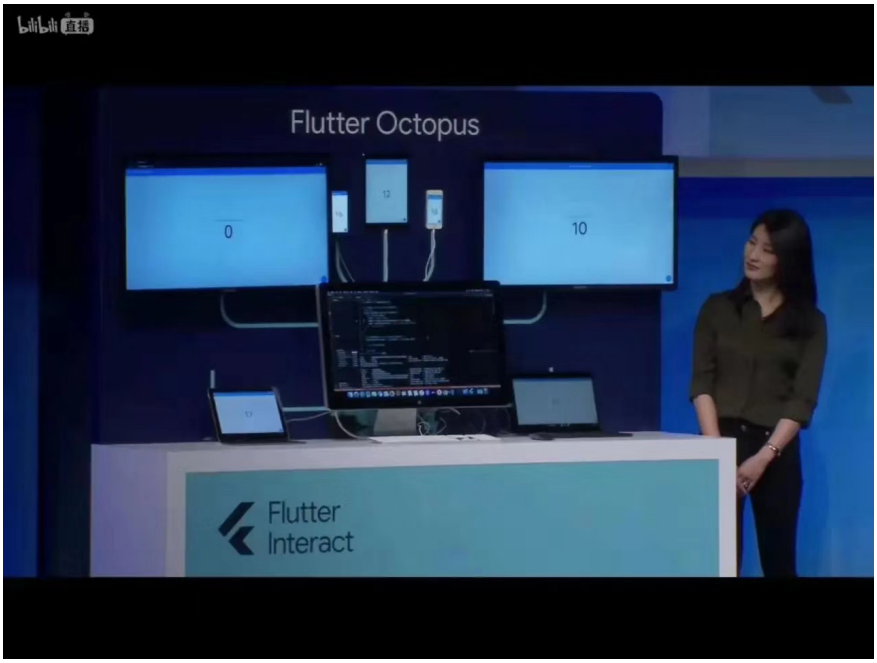
结合本次 `Flutter Interact` ，可以总结出几个关键词是: **`Platform`** 、 **`DartPad`** 、 **`Spuernova`** 、 **`AdobeXD`** 、 **`Hot UI`** 和 **`Layout Explorer`** 。



一、更多的平台

本次 `Flutter Interact` 提出了让开发者更聚焦于精美的应用开发，从**以设备为中心转变为以应用为中心的开发理念**，Flutter 将帮助开发者忽略 Android、iOS、Web、PC 等不同平台差异，如下图所示是现场一套代码同时调试 7 台设备的演示。

本次 Flutter 也开始兑现当初的承诺，目前 **Web 的支持已经发布到 Beta 分支，而 MacOS 的支持已经发布到 Master** 分支。虽然进度不算快，但是作为“白嫖党”表示还是很开心能看到有所推进。

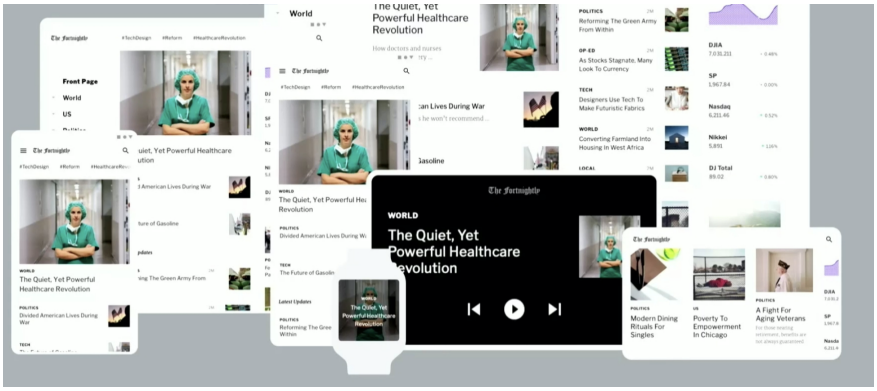


使用 `Flutter Web` 和 `Flutter MacOS` 需要通过如下命令行打开配置，并且执行 `flutter create xxxx` 就可以创建带有 Web 和 MacOS 的项目（如果已有项目也可以执行 `flutter create` 补全），并且需要注意**调试 MacOS 平台应用需要本地 Flutter SDK 要处于 `master` 分支，如果仅测试 Web 可以使用 `beta` 分支。**

```
flutter config --enable-macos-desktop
flutter config --enable-web

///其他平台的支持
flutter config --enable-linux-desktop
flutter config --enable-windows-desktop
```

最后可以通过 `run` 或者 `build` 命令运行和打包程序，同时需要注意这里提到的 `linux` 和 `window` 平台目前还未合并到主项目中，如果想测试可在 Desktop-shells 查看对应配置项目：flutter-desktop-embedding。

```
///调试运行
flutter run -d chrome
flutter run -d macOS

///打包
flutter build web
flutter build macos
```

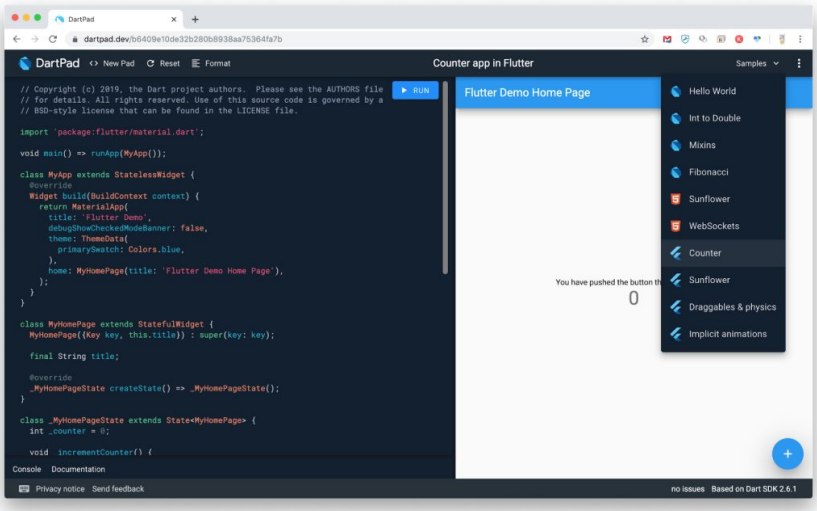


二、更多开发工具

1、DartPad

`DartPad` 是用于在线体验 Dart 功能的平台，而本次更新后 `DartPad` 也支持 Flutter 的在线编写预览，这代表着开发者可以在没有 `idea` 的情况下也能实时测试自己的 Flutter 代码，算是补全了 Flutter 的在线用例测试。

> DartPad 的官方地址：dartpad.dev 和国内镜像地址 dartpad.cn

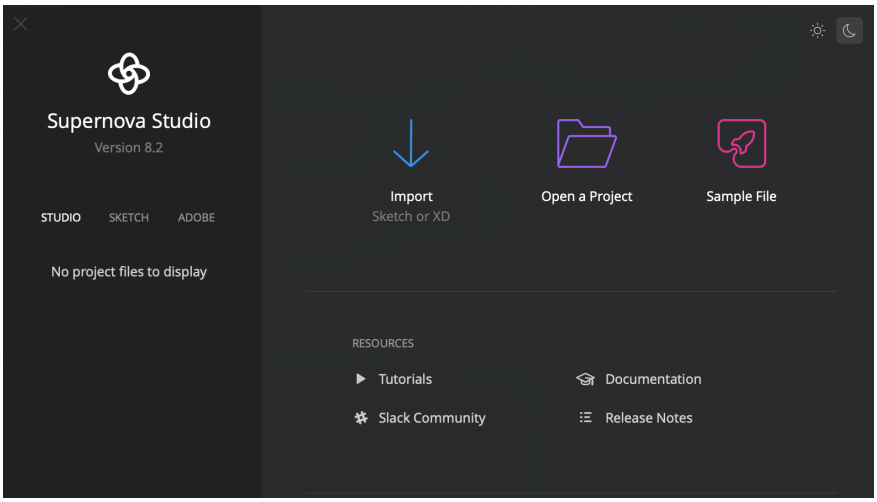


2、Spuernova

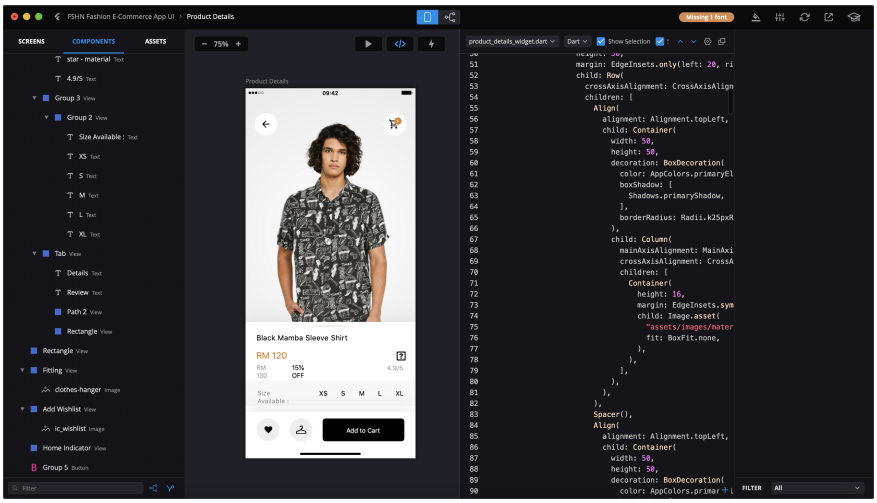
`Spuernova` 可以说是本次 `Flutter Interact` 的亮点之一，通过导入设计师的 `Sketch` 文件就可以生成 Flutter 代码，这无疑提升了 Flutter 的生产力和可想象空间，**虽然这种生成代码的方法并不罕见，完整实用程度有待考验，但是这也让开发者可以更聚焦于业务逻辑和操作逻辑。**

> 放心，这个坑不是谷歌 Flutter 团队开的，它属于另外一家商业公司。

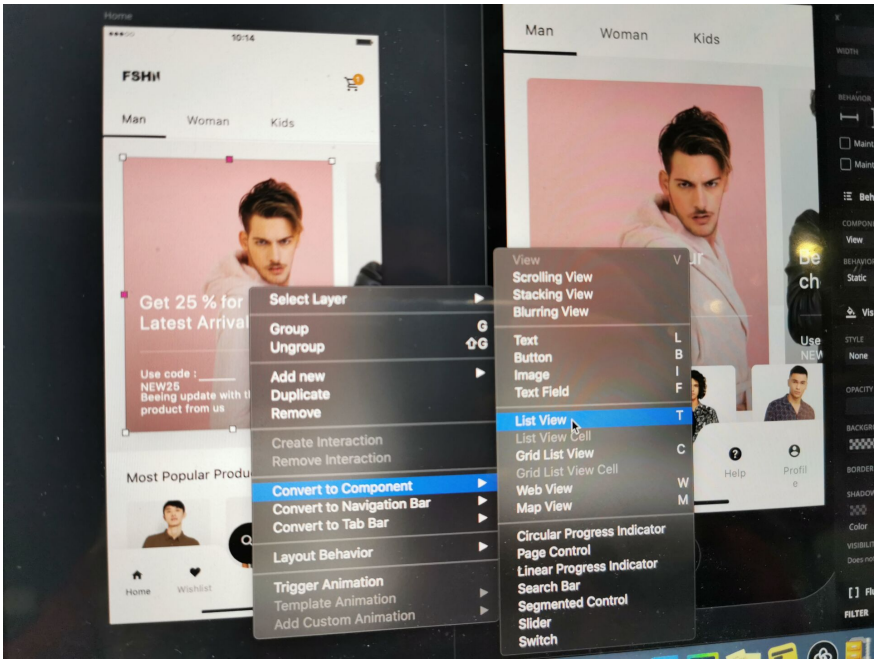
使用 `Spuernova` 可以从 https://supernova.io 下载 `Supernova Studio` ，之后需要注册用户信息（可能需要科学S网），最后就可以看到如下图所示的界面。



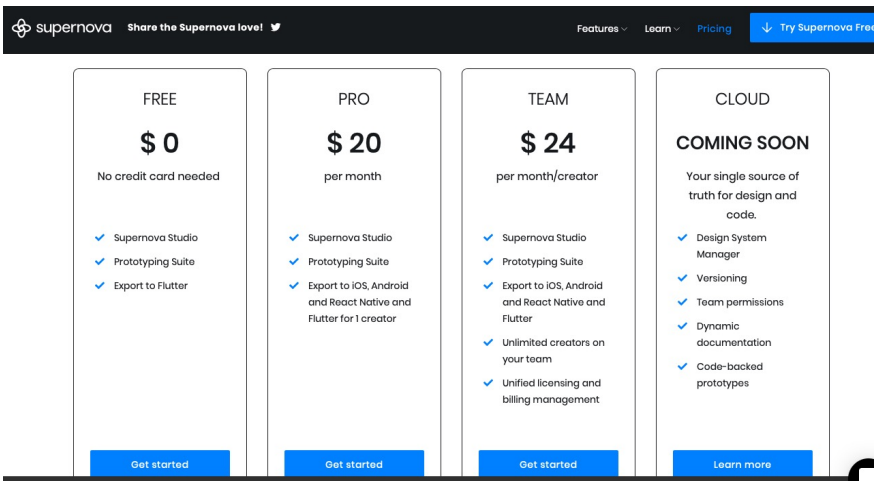
在导入 `Sketch` 文件后可以看到设计师完成的界面效果，同时选中 `"</>"` 按键，可以在右侧看到对应的 Flutter 代码，左侧可以看到对应的层级设计，但是这时候的代码看起来还比较简单和笨重，并且不具备交互能力。



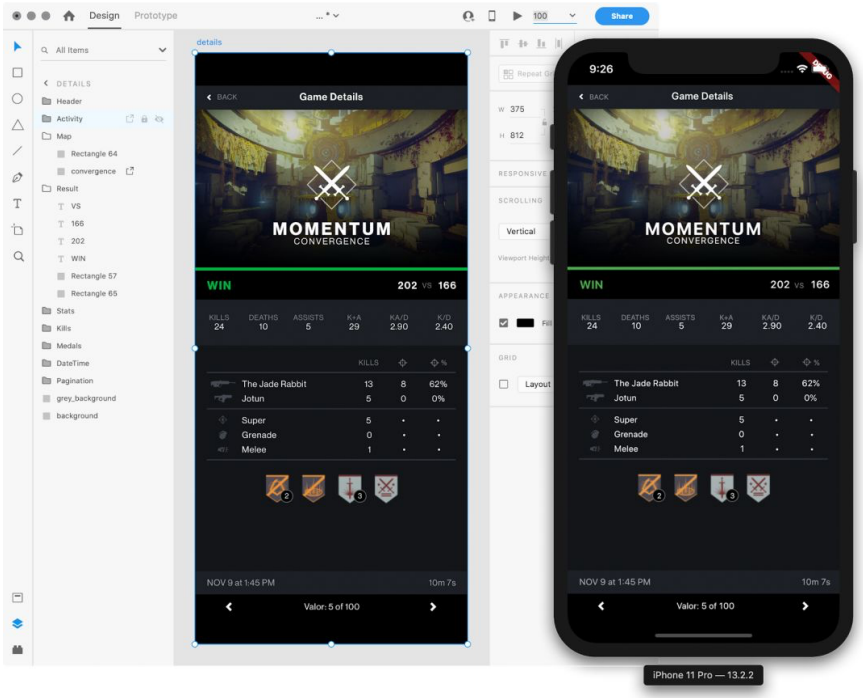
如果进一步配置，用户需要在对应的控件上，使用右键的弹出框配置控件的功能，比如 `List` 、 `Button` 、 `TextField` 等组件去 Convert 原有的控件，让控件更新具备交互能力，同时还可以为控件配置布局属性和动画效果等。



当然， `Spuernova` 并不是什么完全的公益项目，目前只有对于 Flutter 的简单支持上是免费的，其他项目支持还是处于收费状态。

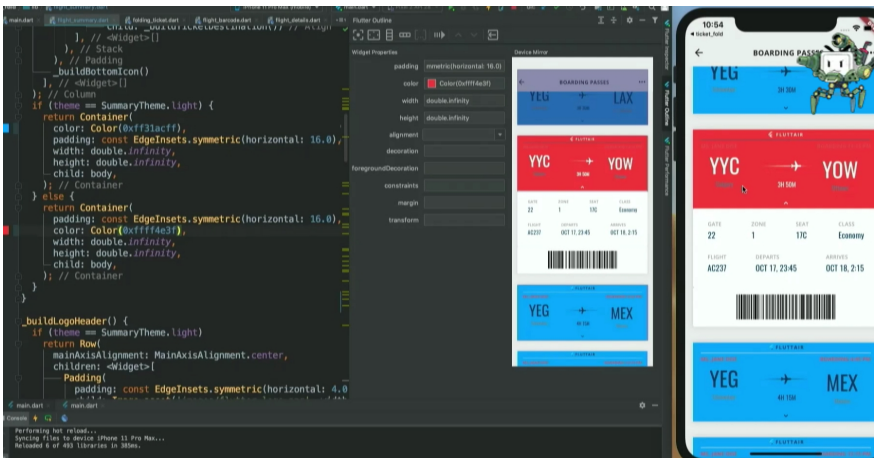


另外类似的还有 `AdobeXD`， Adobe 的 Creative Cloud 添加了 Flutter 支持，只需一个插件，用户就可以将 `AdobeXD` 导出到 Flutter，目前处于注册参加优先体验计划 的进度。

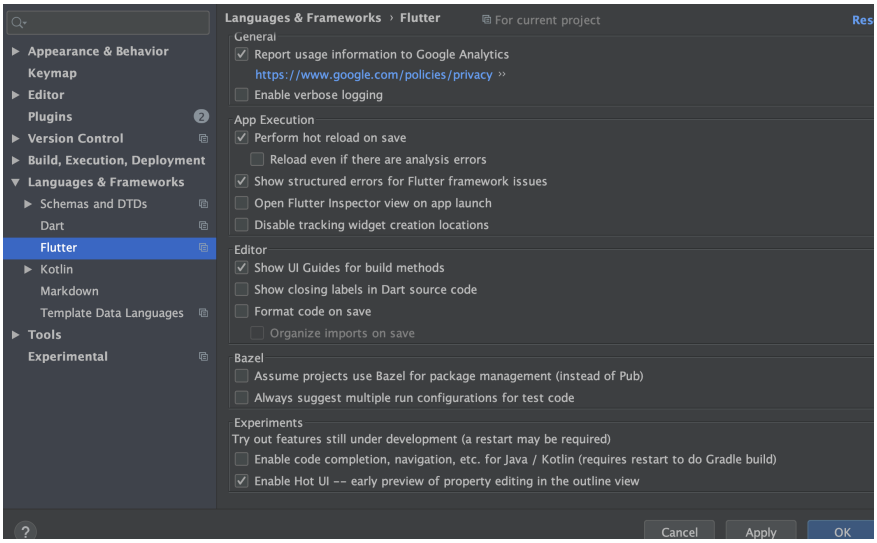


3、Hot UI

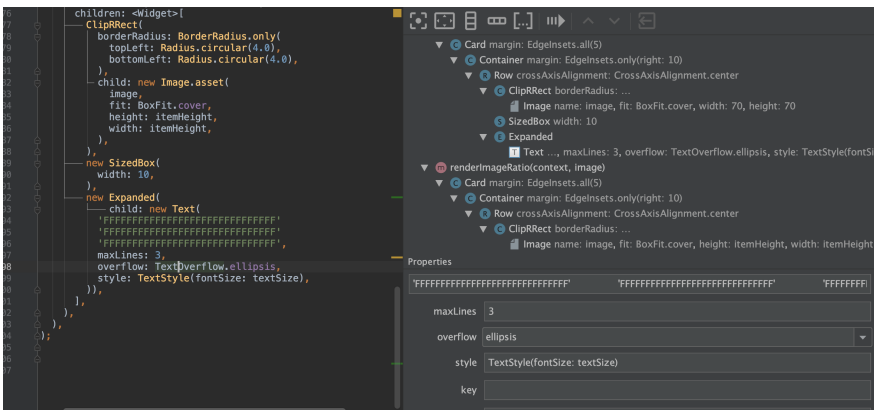
Hot UI 就是大家盼星盼月的预览功能，如下图所示，在 Android Studio 的 Flutter 插件中在开发 widget 开发的过程中，直接在 IDE 的镜像里进行预览并与之进行交互。



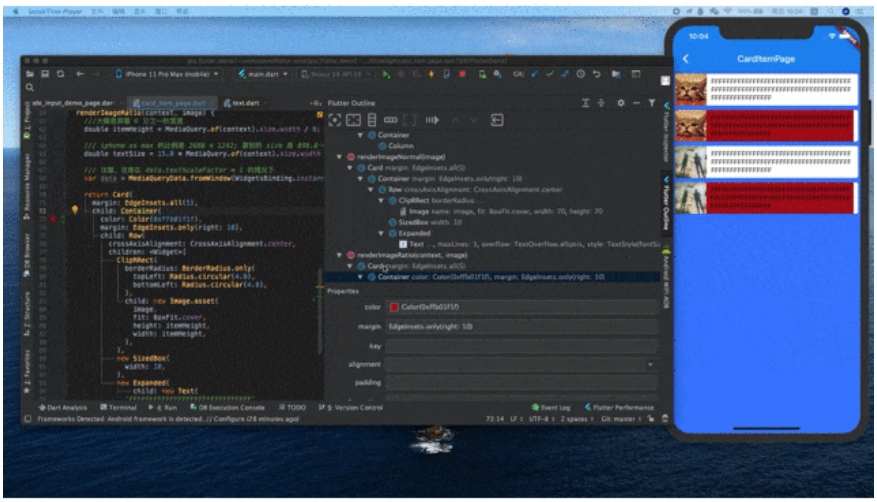
在官方的 HotUI-Getting-Started-instructions 中可以看到相关的描述：**This feature is currently experimental. To enable, go to Preferences > Languages & Frameworks > Flutter Then check "Enable Hot UI" under "Experiments".** 目前该功能还处于实验阶段，在 Android Studio 的设置中，如图所示底部勾选启动这个功能。



但是如下图所示，开启后会发现和官方宣传的不一样？因为目前预览的 `Screen mirror` 处于 `coming soon` 的状态。



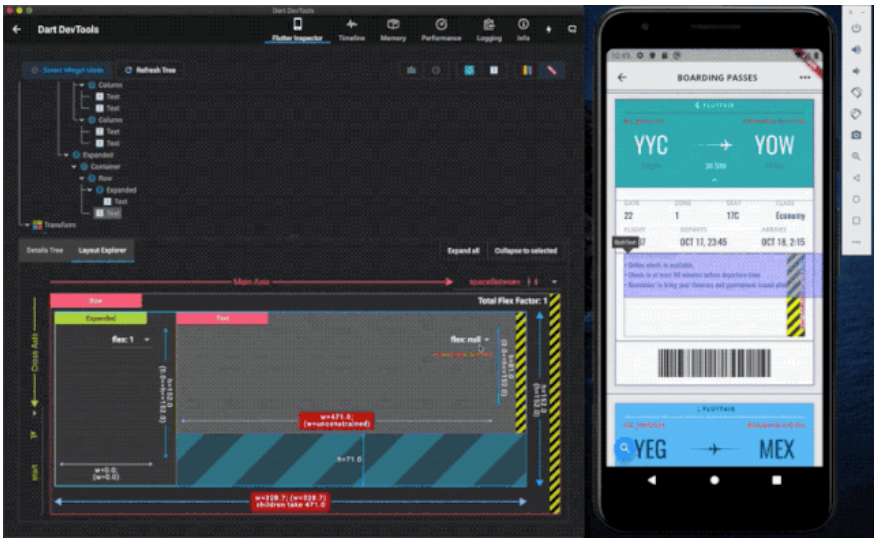
现阶段的 Hot UI 如下 GIF 所示，暂时只支持用户动态调试和配置控件的属性等逻辑，让我们期待官方填坑吧。



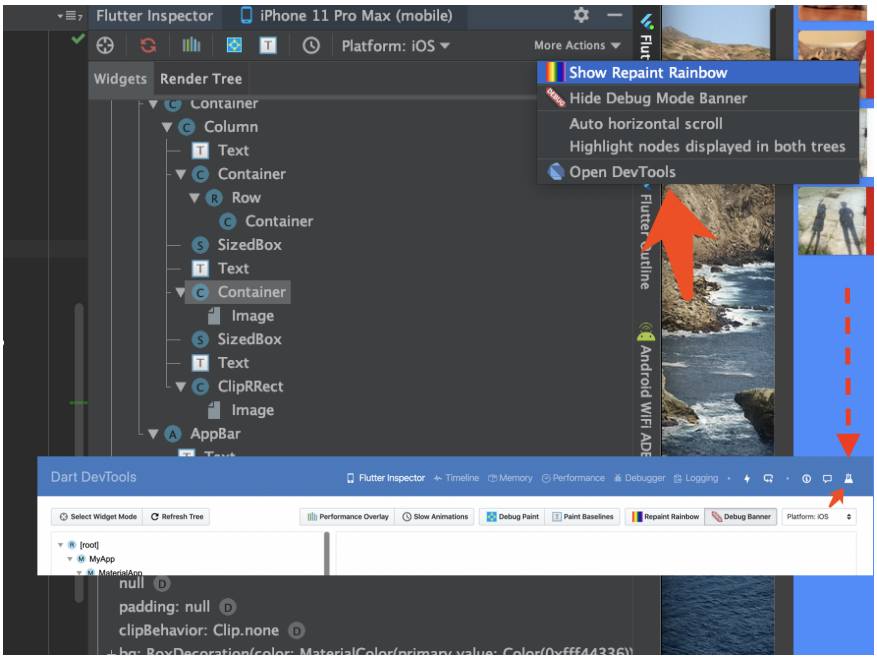
4、Layout Explorer

`Layout Explorer` 是另外实验性的布局调试模式， `Layout Explorer` 主要是用于帮助开发者更直观地适配屏幕和调试如 `overflowed` 等场景的问题。

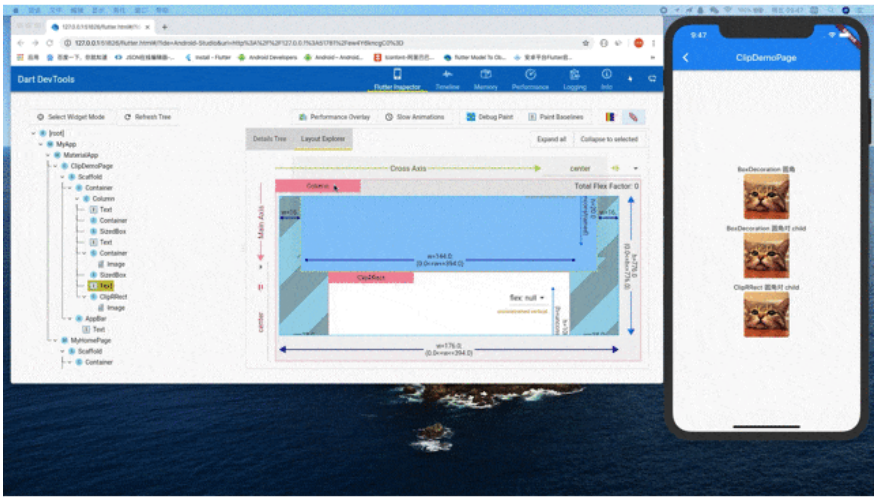
在最新的 `Dart DevTools` 工具添加了一个名为 `Layout Explorer` 的功能，它能够以可视化的方式呈现应用的布局信息，从而让检查器可以更好地发挥功，同时 `Layout Explorer` 不仅能以可视化的方式展现正在运行的应用中的 widget 布局，而且还允许以交互的方式更改布局选项。



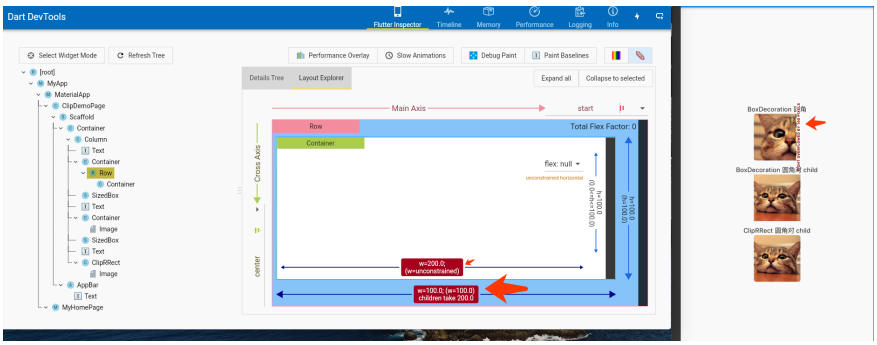
启动 `Layout Explorer` 同样需要 Flutter SDK 处于 `master` 分支，然后在程序运行之后，点击 `DevTools` 在 chrome 打开，之后点击最右侧的按键进入 Flutter 调试模式。



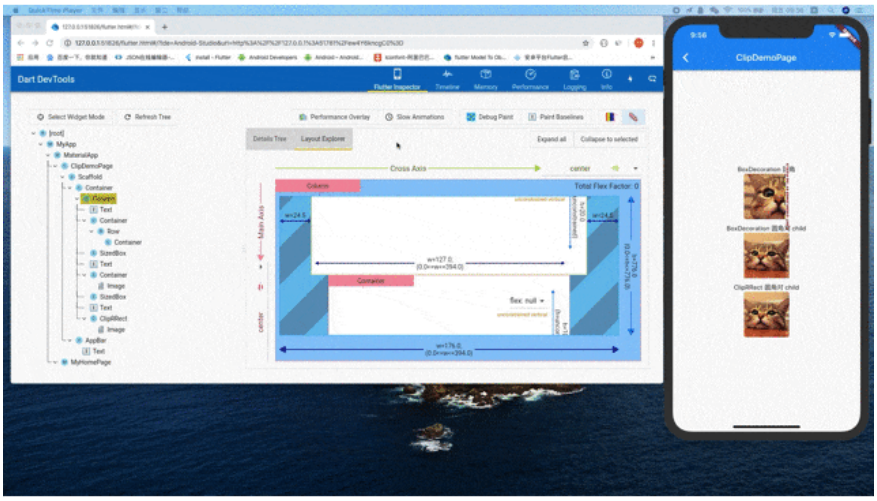
如下 GIF 所示，当选中的控件是具备 `Flex` 的支持时，可以看到有 `Layout Explorer` 的面板，在面板中可以动态调整控件的显示逻辑和控件的布局情况。



比如当控件出现了 `overflowed` ，我们可以很直观的看到问题的根源并且进行调整。



另外可以在 `Layout Explorer` 中动态调整控件的 flex 等相关信息，实时预览修改情况。



三、Flutter SDK 改进

Flutter SDK 相关的更新本次解决了 4,571 个报错，合并了 1,905 份 pr，同时包含了许多的新功能支持。

- 首先 Flutter 1.12 建议开发者将 Android 项目迁移到 AndroidX，SDK 的瘦身，增加了 google_fonts 字体的支持等。
- Android 插件的改进 Android plugins APIs，相比起以前更为简单明了，分割了 `FlutterPlugin` and `MethodCallHandler` ,同时提供 `ActivityAware` 、 `ServiceAware` 作为独立支持。
- iOS 13 深色模式,支持使用 darkTheme 设置，同时还增加了如 `CupertinoContextMenu` 、 `CupertinoSlidingSegmentedControl` 、 `CupertinoAlertDialog` 、 `CupertinoDatePicker` 等 iOS 风格的控件支持。

```
new MaterialApp(
   title: '',
   navigatorKey: navigatorKey,
   theme: model.themeData,
   darkTheme: model.darkthemeData,
   locale: model.locale,
```

- Add-to-App 混合集成模式的进一步的更新。
- 新增加了不兼容的 `breaking change` ，比如: PageView 启用 RenderSliverFillViewport 、 WidgetsBinding 中的 attachRootWidget 被替换为 scheduleAttachRootWidget 、Allow gaps in the initial

route、TextField's minimum height from 40 to 48 等需要开发者注意重新适配的修改，更多可查阅 release-notes-1.12.13。

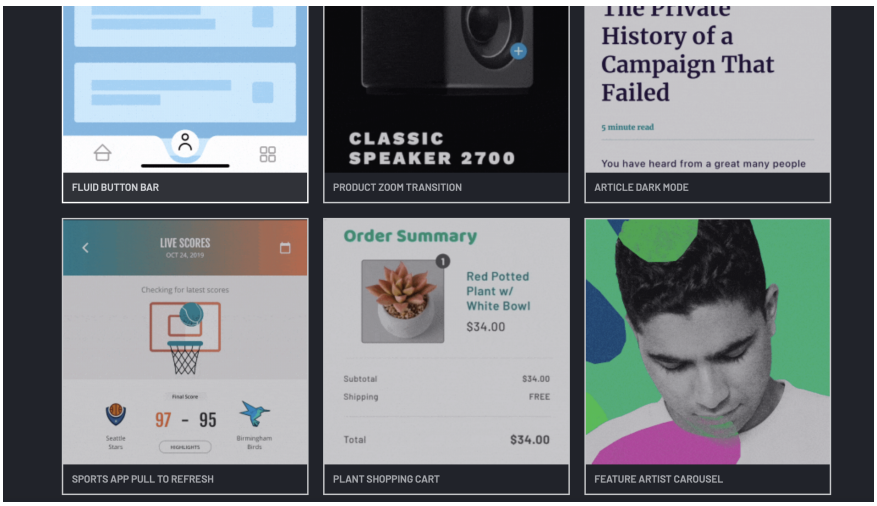
- 增加了 MediaQuery.systemGestureInsets 支持 Android Q 的手势导航；增加了 SliverIgnorePointer 、SliverOpacity、SliverAnimatedList 等控件支持；PageRouteBuilder 支持 fullscreenDialog。
- Dart 2.7 的发布，支持扩展方法。

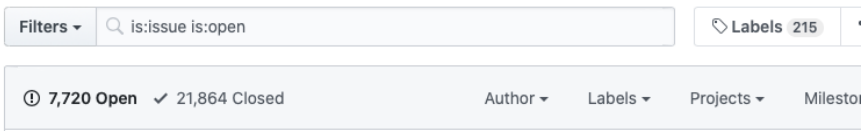
```
extension ExtendsFun on String {
  int parseInt() {
    return int.parse(this);
  }  double parseDouble() {
    return double.parse(this);
  }
}


main() {
  int i = '42'.parseInt();
  print(i);
}
```

更多完整的 release-notes 可见 release-notes-1.12.13

四、其他

本次 `Flutter Interact` 还推荐了 flutter-d-art 和 gskinner 等精美的开源项目，同时 **Flutter 本次也表示了将在未来优化代码的开发模式，而 Flutter 在不断开新坑的同时，也需要面对目前层出的问题。**





Flutter 过去的一年无疑是火热的，所以暴露的问题也指数级出现，比如最近开发中就遇到了**在断网时加载图后之后，再打开网络无法继续显示图片的问题。**

不过既然是开源项目，“白嫖”之余也得多靠自己，上述问题经过查找后，在自定义的 `ImageProvider` 里图片加载失败时，可以通过清除了 `ImageCache` 中的 `PendingImage` 来解决问题，同时因为 `Image` 的封装与 `DecorationImage` 的差异化，还需要对 `Image` 的 `didUpdateWidget` 做二次处理才解决了问题。

说这个问题其实就是想表达开源的意义，用一个框架不能够只是坐享其成的心态，开源的目的更是交流，不管什么框架都不可能尽善尽美，我们可以用更开放的心态去尝试和“批判”，而我们的岗位不就是解决这些问题的么？

Flutter 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

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

`Flutter Interact` 除了带来各种新的开发工具之外，最大的亮点莫过于 **1.12 稳定版本**的发布。

不同于之前的版本，**1.12.x 版本**对 `Flutter Framework` 做了较多的不兼容性升级，例如在 Dart 层： `ImageProvider` 的 `load` 增加了 `DecoderCallback` 参数、TextField's minimum height 从 40 调整到了 48 、PageView 开始使用 SliverLayoutBuilder 而弃用 RenderSliverFillViewport 等相关的不兼容升级。

但是上述的问题都不致命，因为只需要调整相关的 Dart 代码便可以直接解决问题，而此次涉及最大的调整，应该是 **Android 插件的改进 Android plugins APIs 的相关变化，该调整需要用户重新调整 Flutter 项目中 Android 模块和插件的代码进行适配。**

一、Android Plugins

1、介绍

在 Flutter 1.12 开始 Flutter 团队调整了 Android 插件的实现代码，**在 1.12 之后 Android 开始使用新的插件 API ，基于的旧的 `PluginRegistry.Registrar` 不会立即被弃用，但官方建议迁移到基于的新API `FlutterPlugin` ，另外新版本官方建议插件直接使用 `Androidx` 支持**，官方提供的插件也已经全面升级到 `Androidx` 。

与旧的 API 相比，新 API 的优势在于：为插件所依赖的生命周期提供了一套更解耦的使用方法，例如以前 `PluginRegistry.Registrar.activity()` 在使用时，如果 Flutter 还没有添加到 `Activity` 上时可能返回 `null` ，同时插件不知道自己何时被引擎加载使用，而新的 API 上这些问题都得到了优化。

1、升级

在新 API 上 Android 插件需要使用 `FlutterPlugin` 和 `MethodCallHandler` 进行实现，同时还提供了 **`ActivityAware`** 用于 `Activity` 的生命周期管理和获取，提供 **`ServiceAware`** 用于 `Service` 的生命周期管理和获取，具体迁移步骤为：

1、更新主插件类（ `*Plugin.java` ）用于实现 `FlutterPlugin` ， 也就是正常情况下 Android 插件需要继承 `FlutterPlugin` 和 `MethodCallHandler` 这两个接口，如果需要用到 `Activity` 有需要继承 `ActivityAware` 接口。

以前的 Flutter 插件都是直接继承 `MethodCallHandler` 然后提供 `registerWith` 静态方法；而升级后如下代码所示，这里还保留了 `registerWith` 静态方法，是因为还需要针对旧版本做兼容支持，同时

新版 API 中 `MethodCallHandler` 将在 `onAttachedToEngine` 方法中被初始化和构建，在 `onDetachedFromEngine` 方法中释放；同时 `Activity` 相关的四个实现方法也提供了相应的操作逻辑。

```
public class FlutterPluginTestNewPlugin implements FlutterP
  private static MethodChannel channel;

   /// 保留旧版本的兼容
  public static void registerWith(Registrar registerWith) {
    Log.e("registerWith", "registerWith");
    channel = new MethodChannel(registerWith.messenger(), "
    channel.setMethodCallHandler(new FlutterPluginTestNewPl
  }

  @Override
  public void onMethodCall(@NonNull MethodCall call, @NonNu
    if (call.method.equals("getPlatformVersion")) {
      Log.e("onMethodCall", call.method);
      result.success("Android " + android.os.Build.VERSION.
      Map<String, String> map = new HashMap<>();
      map.put("message", "message");
      channel.invokeMethod("onMessageTest", map);
    } else {
      result.notImplemented();
    }
  }

//// FlutterPlugin 的两个 方法
  @Override
  public void onAttachedToEngine(@NonNull FlutterPluginBind
    Log.e("onAttachedToEngine", "onAttachedToEngine");
    channel = new MethodChannel(flutterPluginBinding.getFlu
    channel.setMethodCallHandler(new FlutterPluginTestNewPl
  }

  @Override
  public void onDetachedFromEngine(@NonNull FlutterPluginBi
    Log.e("onDetachedFromEngine", "onDetachedFromEngine");
  }


  ///activity 生命周期
  @Override
  public void onAttachedToActivity(ActivityPluginBinding ac
    Log.e("onAttachedToActivity", "onAttachedToActivity");

  }

  @Override
  public void onDetachedFromActivityForConfigChanges() {
    Log.e("onDetachedFromActivityForConfigChanges", "onDeta
```

```
  }

  @Override
  public void onReattachedToActivityForConfigChanges(Activi
    Log.e("onReattachedToActivityForConfigChanges", "onReat
  }

  @Override
  public void onDetachedFromActivity() {
    Log.e("onDetachedFromActivity", "onDetachedFromActivity
  }
}
```

简单来说就是需要多继承 `FlutterPlugin` 接口，然后在 `onAttachedToEngine` 方法中构建 `MethodCallHandler` 并且 `setMethodCallHandler` ，之后同步在保留的 `registerWith` 方法中实现 `onAttachedToEngine` 中类似的初始化。

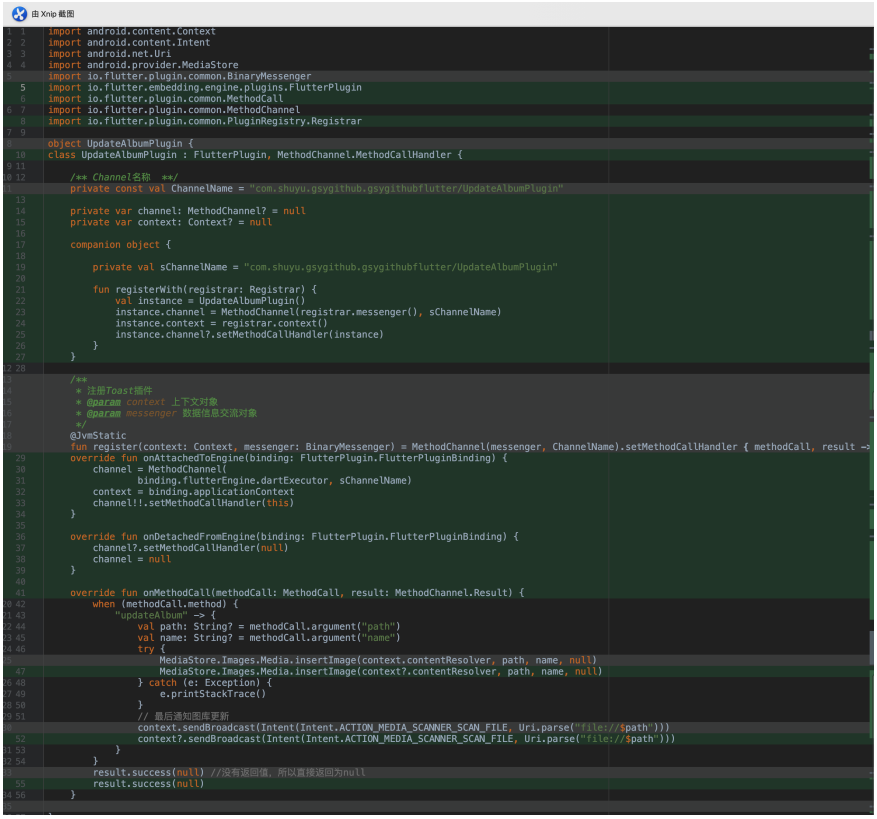
运行后的插件在正常情况下调用的输入如下所示：

```
2019-12-19 18:01:31.481 24809-24809/? E/onAttachedToEngine:
2019-12-19 18:01:31.481 24809-24809/? E/onAttachedToActivit
2019-12-19 18:01:31.830 24809-24809/? E/onMethodCall: getP


2019-12-19 18:05:48.051 24809-24809/com.shuyu.flutter_plugi
2019-12-19 18:05:48.052 24809-24809/com.shuyu.flutter_plugi
```

另外，**如果你插件是想要更好兼容模式对于旧版 Flutter Plugin 运行， `registerWith` 静态方法其实需要调整为如下代码所示：**

```
  public static void registerWith(Registrar registrar) {
    channel = new MethodChannel(registrar.messenger(), "flu
    channel.startListening(registrar.messenger());
  }
```

当然，如果是 Kotlin 插件，可能会是如下图所示类似的更改。



```
import android.content.Context
import android.content.Intent
import android.net.Uri
import android.provider.MediaStore
import io.flutter.plugin.common.BinaryMessenger
import io.flutter.embedding.engine.plugins.FlutterPlugin
import io.flutter.plugin.common.MethodCall
import io.flutter.plugin.common.MethodChannel
import io.flutter.plugin.common.PluginRegistry.Registrar

object UpdateAlbumPlugin {
class UpdateAlbumPlugin : FlutterPlugin, MethodChannel.MethodCallHandler {

    /** Channel名称  **/
    private const val ChannelName = "com.shuyu.gsygithub.gsygithubflutter/UpdateAlbumPlugin"

    private var channel: MethodChannel? = null
    private var context: Context? = null

    companion object {

        private val sChannelName = "com.shuyu.gsygithub.gsygithubflutter/UpdateAlbumPlugin"

        fun registerWith(registrar: Registrar) {
            val instance = UpdateAlbumPlugin()
            instance.channel = MethodChannel(registrar.messenger(), sChannelName)
            instance.context = registrar.context()
            instance.channel?.setMethodCallHandler(instance)
        }
    }

    /**
     * 注册Toast插件
     * @param context 上下文对象
     * @param messenger 数据信息交流对象
     */
    @JvmStatic
    fun register(context: Context, messenger: BinaryMessenger) = MethodChannel(messenger, ChannelName).setMethodCallHandler { methodCall, result ->
    override fun onAttachedToEngine(binding: FlutterPlugin.FlutterPluginBinding) {
        channel = MethodChannel(
                binding.flutterEngine.dartExecutor, sChannelName)
        context = binding.applicationContext
        channel!!.setMethodCallHandler(this)
    }

    override fun onDetachedFromEngine(binding: FlutterPlugin.FlutterPluginBinding) {
        channel?.setMethodCallHandler(null)
        channel = null
    }

    override fun onMethodCall(methodCall: MethodCall, result: MethodChannel.Result) {
        when (methodCall.method) {
            "updateAlbum" -> {
                val path: String? = methodCall.argument("path")
                val name: String? = methodCall.argument("name")
                try {
                    MediaStore.Images.Media.insertImage(context.contentResolver, path, name, null)
                    MediaStore.Images.Media.insertImage(context?.contentResolver, path, name, null)
                } catch (e: Exception) {
                    e.printStackTrace()
                }
                // 最后通知图库更新
                context.sendBroadcast(Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE, Uri.parse("file://$path")))
                context?.sendBroadcast(Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE, Uri.parse("file://$path")))
            }
        }
        result.success(null) //没有返回值，所以直接返回为null
        result.success(null)
    }
```

2、如果条件允许可以修改主项目的 `MainActivity` 对象，**将继承的 `FlutterActivity` 从 `io.flutter.app.FlutterActivity` 替换为 `io.flutter.embedding.android.FlutterActivity` ，之后 插件就可以自动注册；** 如果条件不允许不继承 `FlutterActivity` 的需要自己手动调用 `GeneratedPluginRegistrant.registerWith` 方法，当然到此处可能会提示 `registerWith` 方法调用不正确，不要急忽略它往下走。

```
/// 这个方法如果在下面的 3 中 AndroidManifest.xml 不打开 flutte
@Override
  public void configureFlutterEngine(@NonNull FlutterEngine
    GeneratedPluginRegistrant.registerWith(flutterEngine);
  }
```

> 如果按照 3 中一样打开了 v2，那么生成的 GeneratedPluginRegistrant 就是使用 FlutterEngine，不配置 v2 使用的就是 PluginRegistry 。

3、之后还需要调整 `AndroidManifest.xml` 文件，如下图所示，需要将原本的 `io.flutter.app.android.SplashScreenUntilFirstFrame` 这个 `meta-data` 移除，然后增加为 `io.flutter.embedding.android.SplashScreenDrawable` 和 `io.flutter.embedding.android.NormalTheme` 这两个 `meta-data` ，主要是用于应用打开时的占位图样式和进入应用后的主题样式。

这里还要注意，如上图所示需要在 `application` 节点内配置 `flutterEmbedding` 才能生效新的插件加载逻辑。

```
    <meta-data
        android:name="flutterEmbedding"
        android:value="2" />
```

4、之后就可以执行 `flutter packages get` 去生成了新的 `GeneratedPluginRegistrant` 文件，如下代码所示，新的 `FlutterPlugin` 将被 `flutterEngine.getPlugins().add` 直接加载，而旧的插件实现方法会通过 `ShimPluginRegistry` 被兼容加载到 v2 的实现当中。

```
@Keep
public final class GeneratedPluginRegistrant {
  public static void registerWith(@NonNull FlutterEngine f
    ShimPluginRegistry shimPluginRegistry = new ShimPlugin
    flutterEngine.getPlugins().add(new io.flutter.plugins.
    flutterEngine.getPlugins().add(new io.flutter.plugins.
    flutterEngine.getPlugins().add(new io.flutter.plugins.
      io.github.ponnamkarthik.toast.fluttertoast.Fluttertoa
    flutterEngine.getPlugins().add(new io.flutter.plugins.
    flutterEngine.getPlugins().add(new io.flutter.plugins.
      com.baseflow.permissionhandler.PermissionHandlerPlug
    flutterEngine.getPlugins().add(new io.flutter.plugins.
    flutterEngine.getPlugins().add(new io.flutter.plugins.
      com.tekartik.sqflite.SqflitePlugin.registerWith(shimP
    flutterEngine.getPlugins().add(new io.flutter.plugins.
    flutterEngine.getPlugins().add(new io.flutter.plugins.
  }
}
```

5、最后是可选升级，在 `android/gradle/wrapper` 下的 `gradle-wrapper.properties` 文件，可以将 `distributionUrl` 修改为 `gradle-5.6.2-all.zip` 的版本，同时需要将 `android/` 目录下的 `build.gradle` 文件的 gradle 也修改为 `com.android.tools.build:gradle:3.5.0`；另外 `kotlin` 插件版本也可以升级到 `ext.kotlin_version = '1.3.50'` 。

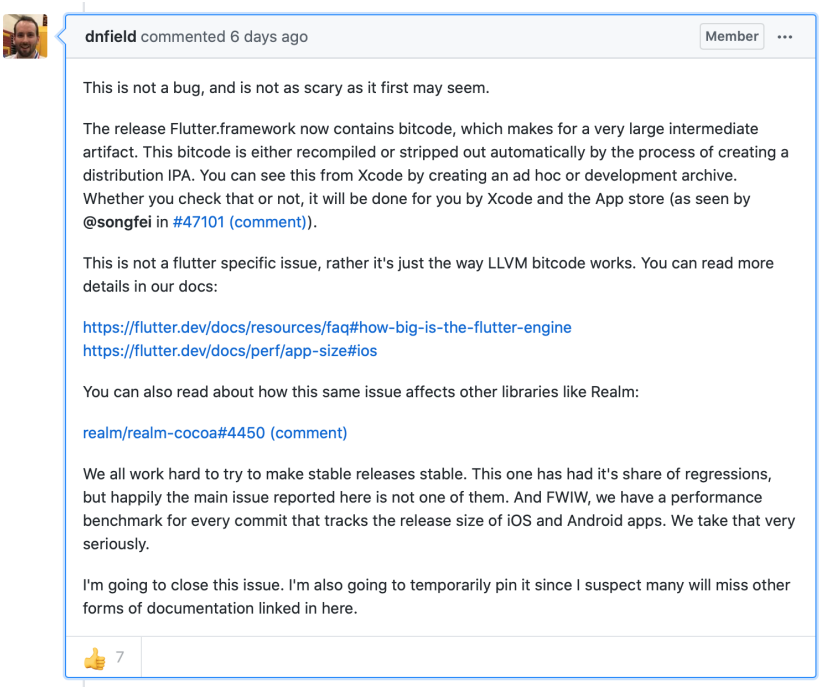
二、其他升级

1、如果之前的项目还没有启用 `Androidx` ，那么可以在 `android/` 目录下的 `gradle.properties` 添加如下代码打开 `Androidx` 。

```
android.enableR8=true
android.useAndroidX=true
android.enableJetifier=true
```

2、需要在忽略文件增加 `.flutter-plugins-dependencies` 。

3、更新之后如果对 iOS 包变大有疑问，可以查阅 #47101 ，这里已经很好的描述了这段因果关系；另外如果发现 iOS13 真机无法输入 log 的问题，可以查看 #41133 。



4、如下图所示，1.12.x 的升级中 iOS 的 `Podfile` 文件也进行了调整，如果还使用旧文件可能会到相应的警告，相关配置也在下方贴出。


```
# platform :ios, '9.0'

# CocoaPods analytics sends network stats synchronously affecting flutter build latency.
ENV['COCOAPODS_DISABLE_STATS'] = 'true'

project 'Runner', {
  'Debug' => :debug,
  'Profile' => :release,
  'Release' => :release,
}

def parse_KV_file(file, separator='=')
  file_abs_path = File.expand_path(file)
  if !File.exists? file_abs_path
    return [];
  end
  pods_ary = []
  generated_key_values = {}
  skip_line_start_symbols = ["#", "/"]
  File.foreach(file_abs_path) { |line|
      next if skip_line_start_symbols.any? { |symbol| line =~ /^\s*#{symbol}/ }
      plugin = line.split(pattern=separator)
      if plugin.length == 2
        podname = plugin[0].strip()
        path = plugin[1].strip()
        podpath = File.expand_path("#{path}", file_abs_path)
        pods_ary.push({:name => podname, :path => podpath});
      else
        puts "Invalid plugin specification: #{line}"
      end
  }
  return pods_ary
  File.foreach(file_abs_path) do |line|
    next if skip_line_start_symbols.any? { |symbol| line =~ /^\s*#{symbol}/ }
    plugin = line.split(pattern=separator)
    if plugin.length == 2
      podname = plugin[0].strip()
      path = plugin[1].strip()
      podpath = File.expand_path("#{path}", file_abs_path)
      generated_key_values[podname] = podpath
    else
      puts "Invalid plugin specification: #{line}"
    end
  end
  generated_key_values
end

target 'Runner' do
  use_frameworks!
  use_modular_headers!

  # Flutter Pod

  copied_flutter_dir = File.join(__dir__, 'Flutter')
  copied_framework_path = File.join(copied_flutter_dir, 'Flutter.framework')
  copied_podspec_path = File.join(copied_flutter_dir, 'Flutter.podspec')
  unless File.exist?(copied_framework_path) && File.exist?(copied_podspec_path)
    # Copy Flutter.framework and Flutter.podspec to Flutter/ to have something to link against if the xcode backend script has not run yet.
    # That script will copy the correct debug/profile/release version of the framework based on the currently selected Xcode configuration.
    # CocoaPods will not embed the framework on pod install (before any build phases can generate) if the dylib does not exist.

    generated_xcode_build_settings_path = File.join(copied_flutter_dir, 'Generated.xcconfig')
    unless File.exist?(generated_xcode_build_settings_path)
      raise "Generated.xcconfig must exist. If you're running pod install manually, make sure flutter pub get is executed first"
    end
    generated_xcode_build_settings = parse_KV_file(generated_xcode_build_settings_path)
    cached_framework_dir = generated_xcode_build_settings['FLUTTER_FRAMEWORK_DIR'];

    unless File.exist?(copied_framework_path)
      FileUtils.cp_r(File.join(cached_framework_dir, 'Flutter.framework'), copied_flutter_dir)
    end
    unless File.exist?(copied_podspec_path)
      FileUtils.cp(File.join(cached_framework_dir, 'Flutter.podspec'), copied_flutter_dir)
    end
  end

  # Keep pod path relative so it can be checked into Podfile.lock.
  pod 'Flutter', :path => 'Flutter'

  # Plugin Pods

  # Prepare symlinks folder. We use symlinks to avoid having Podfile.lock
  # referring to absolute paths on developers' machines.
  system('rm -rf .symlinks')
  system('mkdir -p .symlinks/plugins')

  # Flutter Pods
  generated_xcode_build_settings = parse_KV_file('./Flutter/Generated.xcconfig')
  if generated_xcode_build_settings.empty?
    puts "Generated.xcconfig must exist. If you're running pod install manually, make sure flutter packages get is executed first."
  end
  generated_xcode_build_settings.map { |p|
    if p[:name] == 'FLUTTER_FRAMEWORK_DIR'
      symlink = File.join('.symlinks', 'flutter')
      File.symlink(File.dirname(p[:path]), symlink)
      pod 'Flutter', :path => File.join(symlink, File.basename(p[:path]))
    end
  }

  # Plugin Pods
  plugin_pods = parse_KV_file('../.flutter-plugins')
  plugin_pods.map { |p|
    symlink = File.join('.symlinks', 'plugins', p[:name])
    File.symlink(p[:path], symlink)
    pod p[:name], :path => File.join(symlink, 'ios')
  }
  plugin_pods.each do |name, path|
    symlink = File.join('.symlinks', 'plugins', name)
    File.symlink(path, symlink)
    pod name, :path => File.join(symlink, 'ios')
  end
end

# Prevent Cocoapods from embedding a second Flutter framework and causing an error with the new Xcode build system.
install! 'cocoapods', :disable_input_output_paths => true

post_install do |installer|
  installer.pods_project.targets.each do |target|
    target.build_configurations.each do |config|
      config.build_settings['ENABLE_BITCODE'] = 'NO'
    end
  end
```

```
# Uncomment this line to define a global platform for your
# platform :ios, '9.0'

# CocoaPods analytics sends network stats synchronously af
ENV['COCOAPODS_DISABLE_STATS'] = 'true'

project 'Runner', {
  'Debug' => :debug,
  'Profile' => :release,
  'Release' => :release,
}

def parse_KV_file(file, separator='=')
  file_abs_path = File.expand_path(file)
  if !File.exists? file_abs_path
    return [];
  end
  generated_key_values = {}
  skip_line_start_symbols = ["#", "/"]
  File.foreach(file_abs_path) do |line|
    next if skip_line_start_symbols.any? { |symbol| line =
    plugin = line.split(pattern=separator)
    if plugin.length == 2
      podname = plugin[0].strip()
      path = plugin[1].strip()
      podpath = File.expand_path("#{path}", file_abs_path)
      generated_key_values[podname] = podpath
    else
      puts "Invalid plugin specification: #{line}"
    end
  end
  generated_key_values
end

target 'Runner' do
  use_frameworks!
  use_modular_headers!

  # Flutter Pod

  copied_flutter_dir = File.join(__dir__, 'Flutter')
  copied_framework_path = File.join(copied_flutter_dir, 'F
  copied_podspec_path = File.join(copied_flutter_dir, 'Flut
  unless File.exist?(copied_framework_path) && File.exist?
    # Copy Flutter.framework and Flutter.podspec to Flutte
    # That script will copy the correct debug/profile/relea
    # CocoaPods will not embed the framework on pod instal
```

```
    generated_xcode_build_settings_path = File.join(copied_
    unless File.exist?(generated_xcode_build_settings_path)
      raise "Generated.xcconfig must exist. If you're runni
    end
    generated_xcode_build_settings = parse_KV_file(generate
    cached_framework_dir = generated_xcode_build_settings[

    unless File.exist?(copied_framework_path)
      FileUtils.cp_r(File.join(cached_framework_dir, 'Flutt
    end
    unless File.exist?(copied_podspec_path)
      FileUtils.cp(File.join(cached_framework_dir, 'Flutter
    end
  end

  # Keep pod path relative so it can be checked into Podfi
  pod 'Flutter', :path => 'Flutter'

  # Plugin Pods

  # Prepare symlinks folder. We use symlinks to avoid havi
  # referring to absolute paths on developers' machines.
  system('rm -rf .symlinks')
  system('mkdir -p .symlinks/plugins')
  plugin_pods = parse_KV_file('../.flutter-plugins')
  plugin_pods.each do |name, path|
    symlink = File.join('.symlinks', 'plugins', name)
    File.symlink(path, symlink)
    pod name, :path => File.join(symlink, 'ios')
  end
end

# Prevent Cocoapods from embedding a second Flutter framewo
install! 'cocoapods', :disable_input_output_paths => true

post_install do |installer|
  installer.pods_project.targets.each do |target|
    target.build_configurations.each do |config|
      config.build_settings['ENABLE_BITCODE'] = 'NO'
    end
  end
end
```

好了，暂时就到这了。

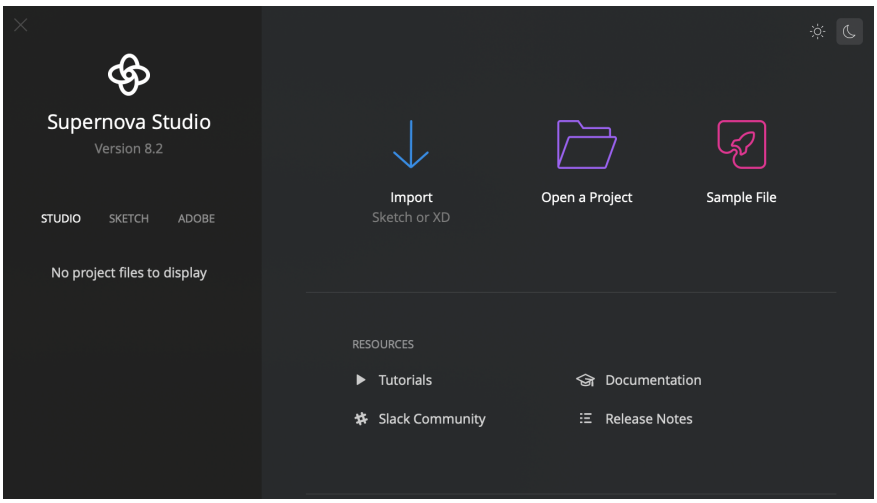
Flutter 文章汇总地址：

> 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

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

关于 Spuernova 我曾在 《Flutter Interact 的 Flutter 1.12 大进化和回顾》中介绍过：在 2019 年末的 Flutter Interact 大会上，Spuernova 发布了对 Flutter 的支持，**通过导入设计师的 Sketch 文件从而生成 Flutter 代码**，这无疑提升了 Flutter 的生产力和可想象空间。

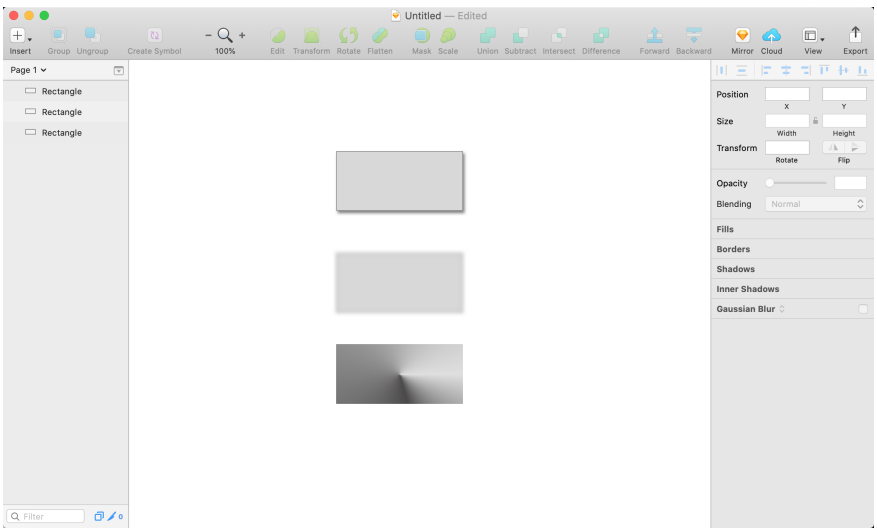


自动生成代码的方式并不罕见，可能不少有过类似经验的开发者会表示“不屑一顾”，也可能会有节奏党再一次拉起“开发药丸”的大旗，当然这次要分享的不会是这些，这次想要分享的是： **Spuernova 可以成为开发者和设计师之间另类的沟通桥梁。**

> **一般情况下设计师和程序员之间是存在某种程度的“生殖隔离”**，设计师产出的效果在开发手上很容易“难产”，那么如何给设计师解释“为什么做不了”和“需要怎么做”就是一件很费劲的事情，甚至关乎到“信任问题”。

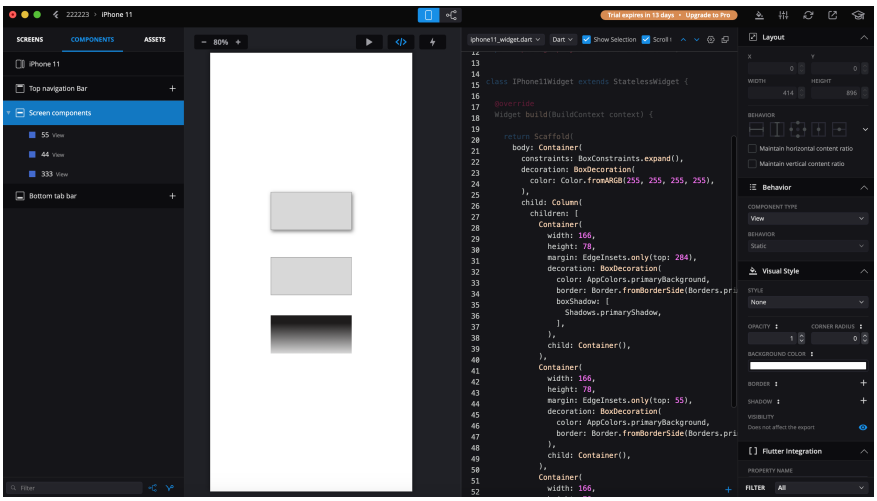
Spuernova 对 Flutter 的支持，可以让设计师很直观地知道 Flutter 能做到什么程度，从而让设计师能够更好地规范 UI 效果，提供沟通的友好度。

举个例子，如下图所示，在设计过程中 *阴影*、*模糊* 和 *渐变* 是常见的效果，而这些效果在 Sketch 上也可以很容易地被实现。



但是这些效果在 Flutter 中能够被完美还原吗?

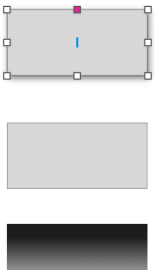
如下图所示，这时候设计师只需要将 Sketch 文件导入到 Spuernova 中，就可以直观地看到设计稿在 Flutter 中的默认渲染效果。



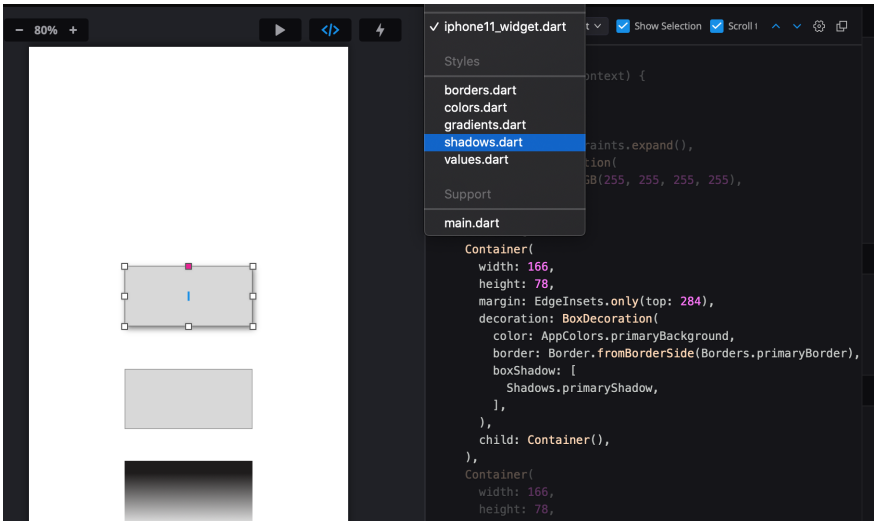
从上图可以看到，**Sketch 中的阴影效果被完美还原，但是模糊和渐变效果却发生了一些变化，说明了这个效果在 Flutter 上“并不支持”。**

> 这时候并不是说 Flutter 就完全没办法还原出设计稿的效果，只是说默认情况下官方并没有支持，所以实现这种效果需要一定成本。

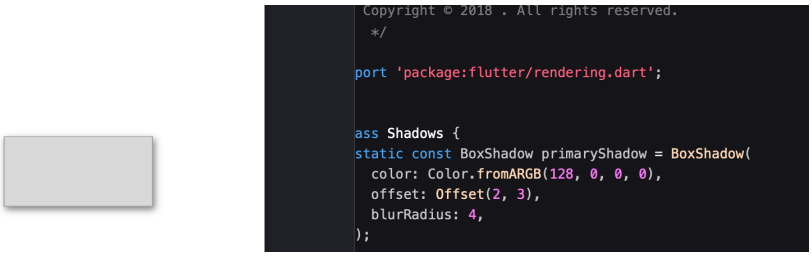
首先如下图所示，在选择阴影框的时候，可以看到在设计稿中的阴影在 Flutter 可以使用 `boxShadow` 实现，而 `boxShadow` 对应的实现代码被放在 `shadows.dart` 文件中。



```
),
child: Column(
  children: [
    Container(
      width: 166,
      height: 78,
      margin: EdgeInsets.only(top: 284),
      decoration: BoxDecoration(
        color: AppColors.primaryBackground,
        border: Border.fromBorderSide(Borders.primaryBorder),
        boxShadow: [
          Shadows.primaryShadow,
        ],
      ),
      child: Container(),
    ),
    Container(
      width: 166,
```



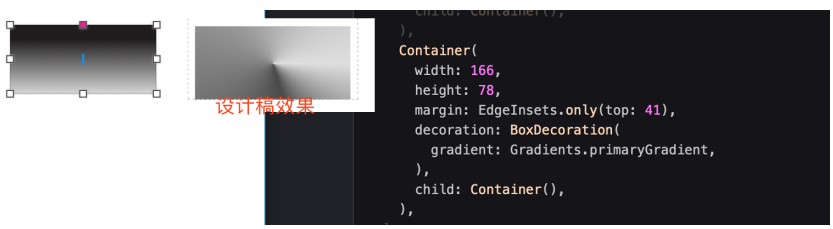
接着查看 `shadows.dart` 文件，可以看到对应的 `primaryShadow` 实现代码，**这时候开发就可以直接 cv 样式代码，不需要对着设计稿一遍一遍的调整参数**，并且在 Supernova 的右侧还有对应给设计师调整参数的工具栏，从而提供了设计和开发之间另类的“沟通语言”。

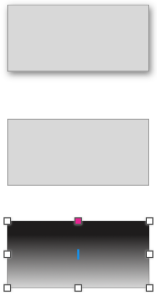


接着看模糊阴影实现，该效果在 Flutter 代码上直接消失了，其实高斯模糊的效果在 Flutter 上是可以实现，这里不过是单纯因为“纯色”效果而导致无法被正常“识别”。

接着看渐变效果，渐变效果在 Flutter 上是用 `Gradient` 实现的，只是设计稿中的渐变效果在 Flutter 上被识别为 `LinerGradient` ，呈现效果出现了偏差。

> 这里应该被识别为 `RadialGradient` 更为贴切，只是想要完成实现设计稿的效果还是有些难度。

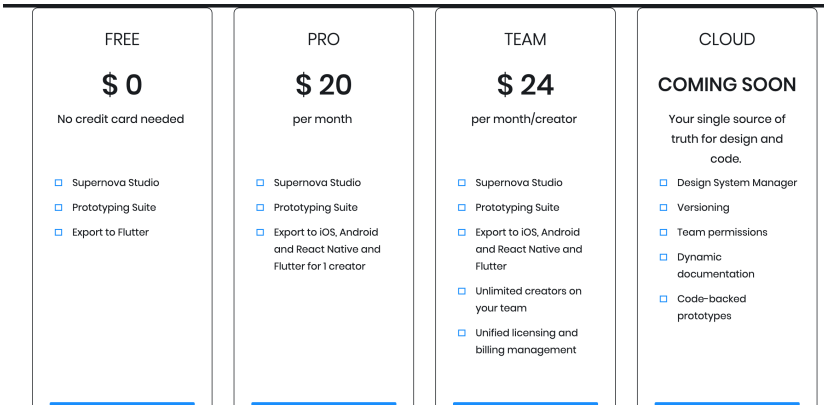




```dart
class Gradients {
  static const Gradient primaryGradient = LinearGradient(
    begin: Alignment(0.5, 0),
    end: Alignment(0.5, 1),
    stops: [
      0.20723,
      1,
    ],
    colors: [
      Color.fromARGB(255, 31, 29, 29),
      Color.fromARGB(255, 216, 216, 216),
    ],
  );
```

从上述例子可以看到 Spuernova 并不完美，甚至在列表、点击、动画等常见效果上还需要做额外的配置，**但是对于我而言 Spuernova 是和设计师沟通的平台，它用更直观的方法告诉了设计师“能做什么”，并且快速让我知道“需要做什么”。**

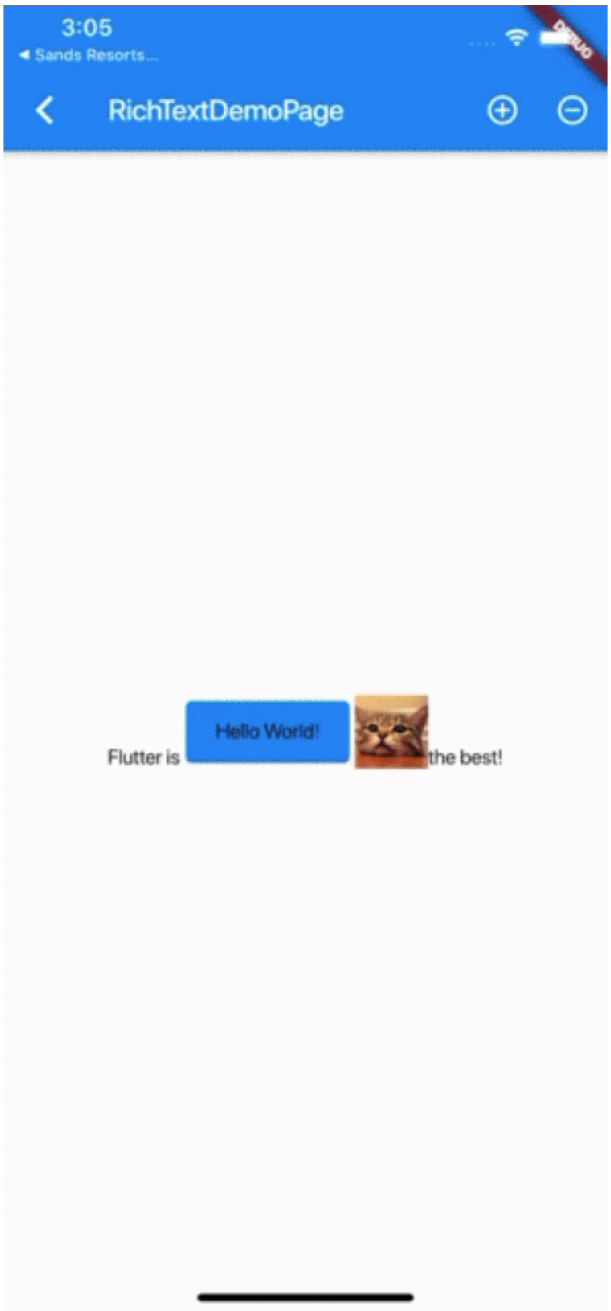
另外还有一个惊喜就是：**Spuernova 还支持 Sketch “转译” 为 Android 、iOS 和 react-native 代码，但是另一个惊喜就是除了 Flutter 之外其他需要收费。**



总的来说 Spuernova 确确实实提升了 Flutter 工程师的生产力，能在一定程度上成为设计师和程序员之间的“桥梁”，虽然它并不完美，但是值得一试。


GSYTech
移动开发的后花园，变强不秃。
长按二维码关注公众号
@恋猫de小郭
忧愁时写一篇文章
快乐时编一段人生
无论工作如何艰辛
我们相信分享总是美好的开始
Flutter 完整开发实战详解
GSYTech
Flutter完整开发实战详解(十八、神奇的...
Flutter完整开发实战详解(十七、实用技巧与填坑二)
Flutter 完整开发实战详解（十六、详解自定义布局...
Flutter完整开发实战详解(十五、全面理解State与...
程序猿的故事
GSYTech
六年码农生涯的 2019 总结：君子坐而论道，少年...
2019大前端浪潮下的年度终结
一个程序员的五年总结，给你不一样的角度
程序员郭小喵创作故事
程序猿郭小喵曾经的实习故事
Android 巴士站
GSYTech
移动开发者的音视频基础知识
做Android开发的，想转行，如何选择呢?
Android程序员的Java后台学习建议
Flutter 杂谈与开发大会
GSYTech
Flutter 会不会被苹果限制其发展?
Flutter Interact 的 Flutter 1.12 大进化和回顾
Flutter 框架在国内移动端市场的发展趋势，在游戏...
Flutter 升级 1.12 适配教程
Flutter 跨平台框架应用实

在移动开发中图文混排是十分常见的业务需求，如下图效果所示，本篇将介绍在 Flutter 中的图文混排效果与实现原理。



事实上，针对如上所示的图文混排需求，Flutter 官方提供了十分便捷的实现方式: **`WidgetSpan`** 。

如下代码所示，**通过 `Text.rich` 接入 `TextSpan` 和 `WidgetSpan` 就可以快速实现图文混排的需求，并且可以看出 `WidgetSpan` 不止支持图片控件**，它可以接入任何你需要的 `Widget` ，比如 `Card` 、 `InkWell` 等等。

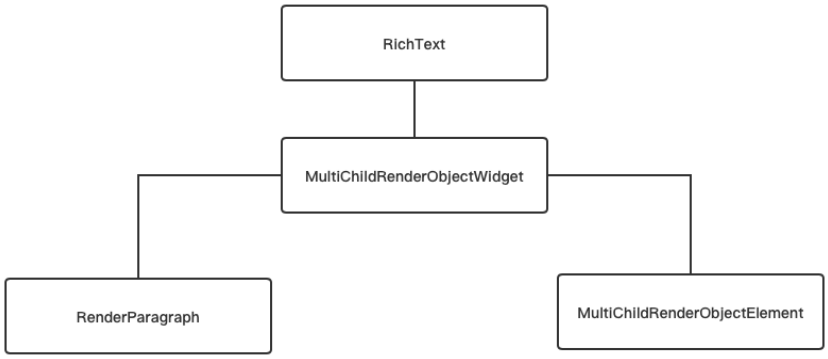
```
Text.rich(TextSpan(
      children: <InlineSpan>[
        TextSpan(text: 'Flutter is'),
        WidgetSpan(
            child: SizedBox(
          width: 120,
          height: 50,
          child: Card(
              color: Colors.blue,
              child: Center(child: Text('Hello World! '))),
        )),
        WidgetSpan(
            child: SizedBox(
          width: size > 0 ? size : 0,
          height: size > 0 ? size : 0,
          child: new Image.asset(
            "static/gsy_cat.png",
            fit: BoxFit.cover,
          ),
        )),
        TextSpan(text: 'the best! '),
      ],
    )
```

也就是说 **`WidgetSpan` 支持在文本中插入任意控件**，这大大提升了 Flutter 中富文本的自定义效果，比如上述演示效果中随意改变图片的大小。

那为什么 `WidgetSpan` 可以如何方便地实现文本和 Widget 混合效果呢？这就要从 `Text` 的实现说起。

实现原理

我们常用的 `Text` 控件其实只是 `RichText` 的封装，而 `RichText` 的实现如下图所示，主要可以分为三部分：**`MultiChildRenderObjectWidget`** 、**`MultiChildRenderObjectElement`** 和 **`RenderParagraph`** 。



正如我们知道的， Flutter 控件一般是由 `Widget` 、 `Element` 和 `RenderObeject` 三部分组成，而在 `RichText` 中也是如此，其中：

- `RenderParagraph` 主要是负责文本绘制、布局相关；
- `RichText` 继承 `MultiChildRenderObjectWidget` 主要是需要通过 `MultiChildRenderObjectElement` 来处理 `WidgetSpan` 中 children 控件的插入和管理。

那 `WidgetSpan` 究竟是如何混入在文本绘制中呢?

在前面的使用中，我们首先是传入了一个 `TextSpan` 给 `RichText` ，并在 `TextSpan` 的 `children` 中拼接我们需要的内容，那就从 `RichText` 开始挖掘其中的原理。

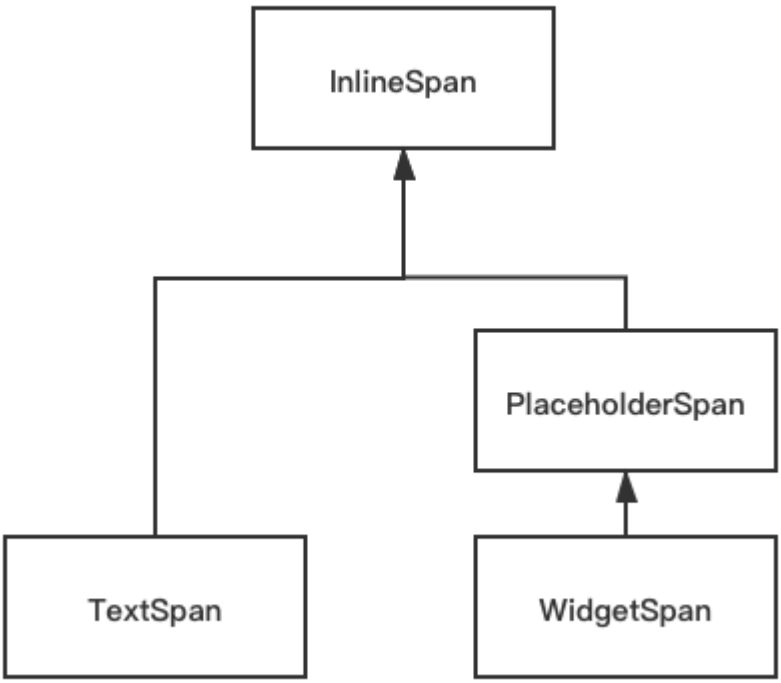
```dart
RichText({
  Key key,
  @required this.text,
  this.textAlign = TextAlign.start,
  this.textDirection,
  this.softWrap = true,
  this.overflow = TextOverflow.clip,
  this.textScaleFactor = 1.0,
  this.maxLines,
  this.locale,
  this.strutStyle,
  this.textWidthBasis = TextWidthBasis.parent,
}) : assert(text != null),
     assert(textAlign != null),
     assert(softWrap != null),
     assert(overflow != null),
     assert(textScaleFactor != null),
     assert(maxLines == null || maxLines > 0),
     assert(textWidthBasis != null),
     super(key: key, children: _extractChildren(text));

// Traverses the InlineSpan tree and depth-first collects the list of
// child widgets that are created in WidgetSpans.
static List<Widget> _extractChildren(InlineSpan span) {
  final List<Widget> result = <Widget>[];
  span.visitChildren((InlineSpan span) {
    if (span is WidgetSpan) {
      result.add(span.child);
    }
    return true;
  });
  return result;
}
```

输入给了 RichText 的父类 MultiChildRenderObjectWidget

如上代码所示，这里我们首先看 `RichText` 的入口，可以看到 `RichText` 开始是有一个 `_extractChildren` 方法，这个方法主要是将传入 `TextSpan` 的 `children` 里，所有的 `WidgetSpan` 通过 `visitChildren` 方法给递归筛选出来，然后传入给父类 `MultiChildRenderObjectWidget` 。

> 为什么需要这么做？在《十六、详解自定义布局实战》中介绍过，`MultiChildRenderObjectWidget` 的 children 最终会通过 `MultiChildRenderObjectElement` 作为桥梁，然后被插入到需要管理和绘制的 child 链表结构中，这样在 `RenderObject` 中方便管理和访问。

另外我们知道 `RichText` 传入的 `text` 其实是一个 `InlineSpan`，而 `TextSpan` 就是 `InlineSpan` 的子类，`WidgetSpan` 也是 `InlineSpan` 的子类实现，它们的关系如下图所示：



对于 `InlineSpan` 系列我们主要关注两个方法：**`visitChildren`** 和 **`build`** 方法，它的子类 `TextSpan` 和 `WidgetSpan` 都对这两个方法有自己对应的实现。

```
void build(ui.ParagraphBuilder builder, { double textSca

bool visitChildren(InlineSpanVisitor visitor);
```

```dart
@override
RenderParagraph createRenderObject(BuildContext context) {
  assert(textDirection != null || debugCheckHasDirectionality(context));
  return RenderParagraph(text,
    textAlign: textAlign,
    textDirection: textDirection ?? Directionality.of(context),
    softWrap: softWrap,
    overflow: overflow,
    textScaleFactor: textScaleFactor,
    maxLines: maxLines,
    strutStyle: strutStyle,
    textWidthBasis: textWidthBasis,
    locale: locale ?? Localizations.localeOf(context, nullOk: true),
  );
}
```

接着看 `RenderParagraph` ，如上代码所示， `RichText` 中的 `text` （ `InlineSpan` ） 会继续被传入到 `RenderParagraph` 中， `RenderParagraph` 继承了 `RenderBox` 并混入的 `ContainerRenderObjectMixin` 和 `RenderBoxContainerDefaultsMixin` 等。

> 混入的对象这部分在内容在 《十六、详解自定义布局实战》 也介绍过，这里只需要知道通过混入它们， `RenderParagraph` 就可以获得前面通过 `WidgetSpan` 传入到 `MultiChildRenderObjectElement` 的 children 链表，并且布局计算大小等。

```dart
/// A render object that displays a paragraph of text.
class RenderParagraph extends RenderBox
    with ContainerRenderObjectMixin<RenderBox, TextParentData>,
            RenderBoxContainerDefaultsMixin<RenderBox, TextParentData> {
```

之后 `RenderParagraph` 中的 `text` 之后会被放置到 `TextPainter` 中使用，并且通过 `_extractPlaceholderSpans` 方法将所有的 `PlaceholderSpans` 筛选出来。

`TextPainter` 主要用于实现文本的绘制，这里我们暂时不多分析，**而 `_extractPlaceholderSpans` 挑选出来的所有 `PlaceholderSpans` ，其实就是 `WidgetSpan` 。**

> `WidgetSpan` 是通过继承 `PlaceholderSpans` 从而实现了 `InlineSpan` ，而目前暂时 `PlaceholderSpans` 实现的类只有 `WidgetSpan` 。

```
RenderParagraph(InlineSpan text, {
  TextAlign textAlign = TextAlign.start,
  @required TextDirection textDirection,
  bool softWrap = true,
  TextOverflow overflow = TextOverflow.clip,
  double textScaleFactor = 1.0,
  int maxLines,
  TextWidthBasis textWidthBasis = TextWidthBasis.parent,
  Locale locale,
  StrutStyle strutStyle,
  List<RenderBox> children,
}) : assert(text != null),
     assert(text.debugAssertIsValid()),
     assert(textAlign != null),
     assert(textDirection != null),
     assert(softWrap != null),
     assert(overflow != null),
     assert(textScaleFactor != null),
     assert(maxLines == null || maxLines > 0),
     assert(textWidthBasis != null),
     _softWrap = softWrap,
     _overflow = overflow,
     _textPainter = TextPainter(
       text: text,
       textAlign: textAlign,
       textDirection: textDirection,
       textScaleFactor: textScaleFactor,
       maxLines: maxLines,
       ellipsis: overflow == TextOverflow.ellipsis ? _kEllipsis : null,
       locale: locale,
       strutStyle: strutStyle,
       textWidthBasis: textWidthBasis,
     ) {
  addAll(children);
  _extractPlaceholderSpans(text);
}
```

挑选出来的 `List<PlaceholderSpan>` 们会在 `RenderParagraph` 计算宽高等方法中被用到，比如 `computeMaxIntrinsicWidth` 方法等，**其中主要有 `_canComputeIntrinsics` 、`_computeChildrenWidthWithMaxIntrinsics` 、`_layoutText` 三个关键**方法，这三个方法结合处理了 `RenderParagraph` 中 Span 的尺寸和布局等。

```
@override
double computeMaxIntrinsicWidth(double height) {
  if (!_canComputeIntrinsics()) {
    return 0.0;
  }
  _computeChildrenWidthWithMaxIntrinsics(height);
  _layoutText(); // layout with infinite width.
  return _textPainter.maxIntrinsicWidth;
}
```

- **`_canComputeIntrinsics`** ： `_canComputeIntrinsics` 主要判断了 `PlaceholderSpan` 只支持的 `baseline` 配置。

```
// Intrinsics cannot be calculated without a full layout for
// alignments that require the baseline (baseline, aboveBaseline,
// belowBaseline).
bool _canComputeIntrinsics() {
  for (PlaceholderSpan span in _placeholderSpans) {
    switch (span.alignment) {
      case ui.PlaceholderAlignment.baseline:
      case ui.PlaceholderAlignment.aboveBaseline:
      case ui.PlaceholderAlignment.belowBaseline: {
        assert(RenderObject.debugCheckingIntrinsics,
          'Intrinsics are not available for PlaceholderAlignment.baseline, '
          'PlaceholderAlignment.aboveBaseline, or PlaceholderAlignment.belowBaseline,');
        return false;
      }
      case ui.PlaceholderAlignment.top:
      case ui.PlaceholderAlignment.middle:
      case ui.PlaceholderAlignment.bottom: {
        continue;
      }
    }
  }
  return true;
}
```

- **`_computeChildrenWidthWithMaxIntrinsics`**：`_computeChildrenWidthWithMaxIntrinsics` 中会**通过 `PlaceholderSpan` 去对应得到 `PlaceholderDimensions`**，得到的 `PlaceholderDimensions` 会用于后续如 `WidgetSpan` 的大小绘制信息。

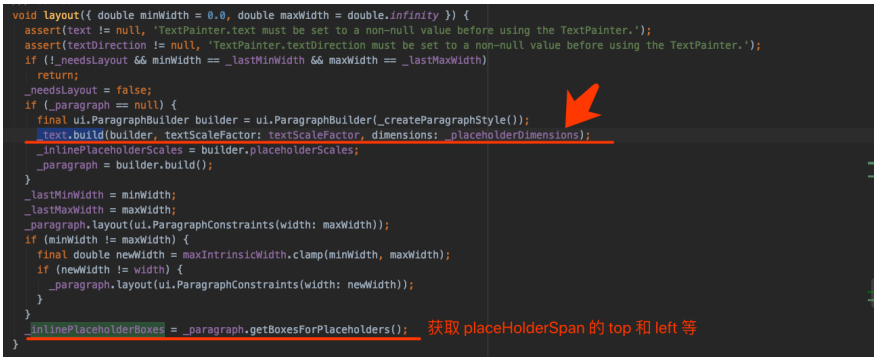
> 这个 `PlaceholderDimensions` 会通过 `setPlaceholderDimensions` 方法设置到 `TextPainter` 里面，这样 `TextPainter` 在 `layout` 的时候，就会将 `PlaceholderDimensions` 赋予 `WidgetSpan` 大小信息。

```
void _computeChildrenWidthWithMaxIntrinsics(double height) {
  RenderBox child = firstChild;
  final List<PlaceholderDimensions> placeholderDimensions = List<PlaceholderDimensions>(childCount);
  int childIndex = 0;
  while (child != null) {
    // Height and baseline is irrelevant as all text will be laid
    // out in a single line.
    placeholderDimensions[childIndex] = PlaceholderDimensions(
      size: Size(child.getMaxIntrinsicWidth(height), height),
      alignment: _placeholderSpans[childIndex].alignment,
      baseline: _placeholderSpans[childIndex].baseline,
    );
    child = childAfter(child);
    childIndex += 1;
  }
  _textPainter.setPlaceholderDimensions(placeholderDimensions);
}
```

- **`_layoutText`**：`_layoutText` 方法会调用 `_textPainter.layout`，从而执行 `_text.build` 方法，这个方法就会触发 `children` 中的 `WidgetSpan` 去执行 `build`。

```
void _layoutText({ double minWidth = 0.0, double maxWidth = double.infinity }) {
  final bool widthMatters = softWrap || overflow == TextOverflow.ellipsis;
  _textPainter.layout(
    minWidth: minWidth,
    maxWidth: widthMatters ?
      maxWidth :
      double.infinity,
  );
}
```

所以如下代码所示，`_textPainter.layout` 会执行 Span 的 `build` 方法，将 `PlaceholderDimensions` 设置到 `WidgetSpan` 里面，然后还有**通过 `_paragraph.getBoxesForPlaceholders()` 方法获取到控件绘制需要的 `left`、`right` 等信息**，这些信息来源是基于上面 `text.build` 的执行。



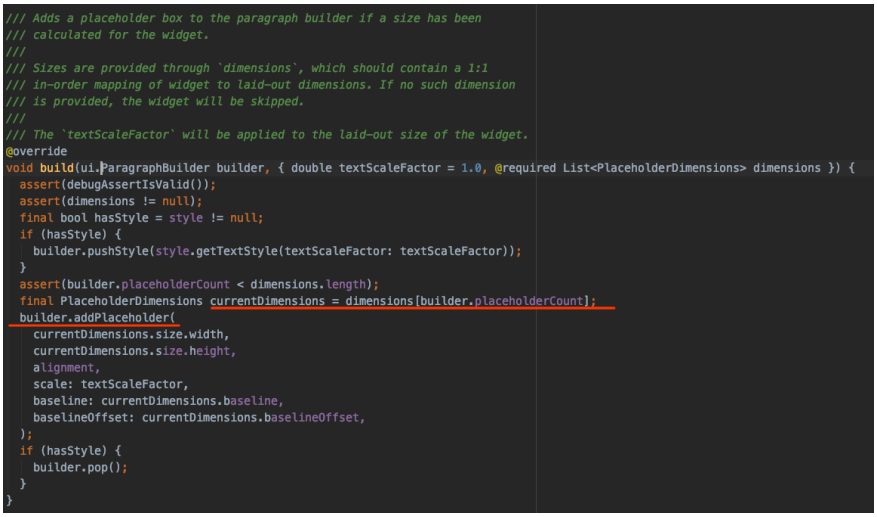
> _paragraph.getBoxesForPlaceholders() 获取到的 `TextBox` 信息，是基于后面我们介绍在 Span 里提交的 `addPlaceholder` 方法获取。

这些信息会在 `setParentData` 方法中被设置到 `TextParentData` 里，关于 `ParentData` 及其子类的作用，在《十六、详解自定义布局实战》同样有所介绍，这里就不赘述了，简单理解就是 `WidgetSpan` 绘制的时候所需要的 `offset` 位置信息会由它们提供。

```dart
// Iterate through the laid-out children and set the parentData offsets based
// off of the placeholders inserted for each child.
void _setParentData() {
  RenderBox child = firstChild;
  int childIndex = 0;
  while (child != null) {
    final TextParentData textParentData = child.parentData;
    textParentData.offset = Offset(
      _textPainter.inlinePlaceholderBoxes[childIndex].left,
      _textPainter.inlinePlaceholderBoxes[childIndex].top
    );
    textParentData.scale = _textPainter.inlinePlaceholderScales[childIndex];
    child = childAfter(child);
    childIndex += 1;
  }
}
```

之后如下代码所示，`WidgetSpan` 的 `build` 方法被执行，这里会有一个 `placeholderCount`，`placeholderCount` 默认是从 0 开始，而在执行 `addPlaceholder` 方法时会通过 `_placeholderCount++` 自增，这样下一个 `WidgetSpan` 就会拿到下一个 `PlaceholderDimensions` 用于设置大小。

> `addPlaceholder` 之后会执行到 Flutter Engine 中的流程了。



```
/// Adds a placeholder box to the paragraph builder if a size has been
/// calculated for the widget.
///
/// Sizes are provided through `dimensions`, which should contain a 1:1
/// in-order mapping of widget to laid-out dimensions. If no such dimension
/// is provided, the widget will be skipped.
///
/// The `textScaleFactor` will be applied to the laid-out size of the widget.
@override
void build(ui.ParagraphBuilder builder, { double textScaleFactor = 1.0, @required List<PlaceholderDimensions> dimensions }) {
  assert(debugAssertIsValid());
  assert(dimensions != null);
  final bool hasStyle = style != null;
  if (hasStyle) {
    builder.pushStyle(style.getTextStyle(textScaleFactor: textScaleFactor));
  }
  assert(builder.placeholderCount < dimensions.length);
  final PlaceholderDimensions currentDimensions = dimensions[builder.placeholderCount];
  builder.addPlaceholder(
    currentDimensions.size.width,
    currentDimensions.size.height,
    alignment,
    scale: textScaleFactor,
    baseline: currentDimensions.baseline,
    baselineOffset: currentDimensions.baselineOffset,
  );
  if (hasStyle) {
    builder.pop();
  }
}
```

最终 `RenderParagrash` 的 `paint` 方法会执行 `_textPainter.paint` 并把确定了大小和位置的 child 提交绘制。

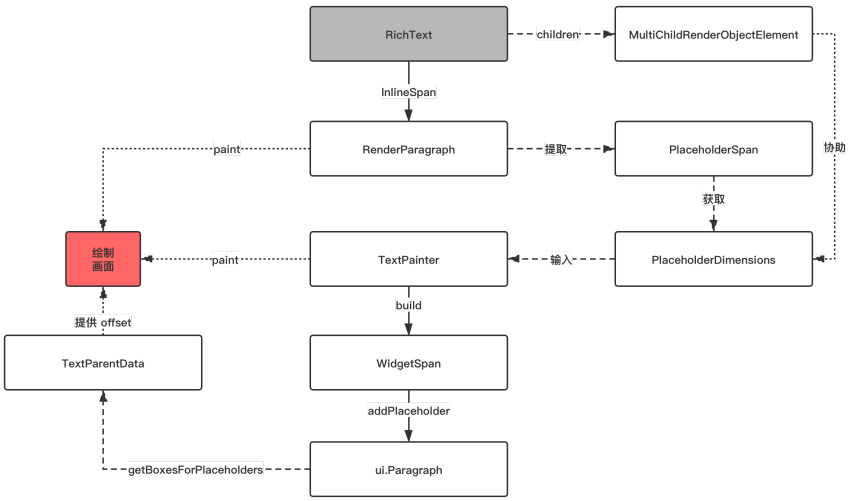
```
@override
void paint(PaintingContext context, Offset offset) {

  _layoutTextWithConstraints(constraints);

  _textPainter.paint(context.canvas, offset);

  RenderBox child = firstChild;
  int childIndex = 0;
  while (child != null) {
    assert(childIndex < _textPainter.inlinePlaceholderBoxes.length);
    final TextParentData textParentData = child.parentData;

    final double scale = textParentData.scale;
    context.pushTransform(
      needsCompositing,
      offset + textParentData.offset,
      Matrix4.diagonal3Values(scale, scale, scale),
          (PaintingContext context, Offset offset) {
        context.paintChild(
          child,
          offset,
        );
      },
    );
    child = childAfter(child);
    childIndex += 1;
  }
}
```

是不是有点晕，结合下图所示，总结起来其实就是:

- `RichText` 中传入 `TextSpan`， 在 `TextSpan` 的 children 中使用 `WidgetSpan`， `WidgetSpan` 里的 `Widget` 们会转成 `MultiChildRenderObjectElement` 的 `children`， 处理后得到一个 child 链表结构；
- 之后 `TextSpan` 进入 `RenderParagrash`，会抽取出对应 `PlaceholderSpan`（`WidgetSpan`），然后通过转化为

`PlaceholderDimensions` 保存大小等信息；

- 之后进去 `TextPainter` 会触发 `InlineSpan` 的 `build` 方法，从而将前面得到的 `PlaceholderDimensions` 传递到 `WidgetSpan` 中；
- `WidgetSpan` 中的控件信息通过 `addPlaceholder` 会被传递到 `Paragraph` ；
- 之后 `TextPainter` 中通过 `addPlaceholder` 的信息获取，调用 `_paragraph.getBoxesForPlaceholders()` 获取去控件绘制需要的 `offset` ；
- 有了大小和位置，最终文本中插入的控件，会在 `RenderParagrash` 的 `paint` 方法被绘制。



`RichText` 中插入控件的管理巧妙的依托到 `MultiChildRenderObjectWidget` 中，从而复用了原本控件的管理逻辑，之后依托引擎计算位置从而绘制完成。

至此，简简单单的 `WidgetSpan` 的实现原理解析完成～

资源推荐

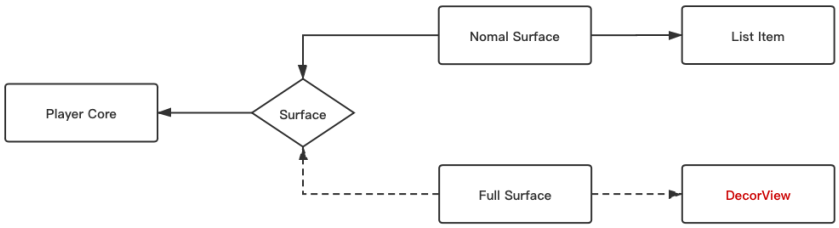
- Github： https://github.com/CarGuo
- 开源 Flutter 完整项目：https://github.com/CarGuo/GSYGithubAppFlutter
- 开源 Flutter 多案例学习型项目:https://github.com/CarGuo/GSYFlutterDemo
- 开源 Fluttre 实战电子书项目：https://github.com/CarGuo/GSYFlutterBook
- 开源 React Native 项目：https://github.com/CarGuo/GSYGithubApp


GSYTech
移动开发的后花园，变强不秃。
长按二维码关注公众号
@恋猫de小郭
忧愁时写一篇文章
快乐时编一段人生
无论工作如何艰辛
我们相信分享总是美好的开始
Flutter 完整开发实战详解
GSYTech
Flutter完整开发实战详解(十八、神奇的...
Flutter完整开发实战详解(十七、实用技巧与填坑二)
Flutter 完整开发实战详解（十六、详解自定义布局...
Flutter完整开发实战详解(十五、全面理解State与...
Flutter
程序猿的故事
GSYTech
六年码农生涯的 2019 总结：君子坐而论道，少年...
2019大前端浪潮下的年度终结
一个程序员的五年总结，给你不一样的角度
程序员郭小喵创作故事
程序猿郭小喵曾经的实习故事
Android 巴士站
GSYTech
移动开发者的音视频基础知识
做Android开发的，想转行，如何选择呢?
Android程序员的Java后台学习建议
Flutter 杂谈与开发大会
GSYTech
Flutter 会不会被苹果限制其发展?
Flutter Interact 的 Flutter 1.12 大进化和回顾
Flutter 框架在国内移动端市场的发展趋势，在游戏...
Flutter 升级 1.12 适配教程
Flutter 跨平台框架应用实战-2019极光开发者大会

一、前言

相信做过移动端视频开发的同学应该了解，想要实现视频从普通播放到全屏播放的逻辑并不是很简单，比如在 GSYVideoPlayer 中的动态全屏切换效果，就使用了创建全新的 `Surface` 来替换实现:

- 创建全新的 `Surface` ，并将对于的 `View` 添加到应用顶层的 `DecorView` 中；
- 在全屏时将新创建的 `Surface` 并设置到 Player Core ;
- 同步两个 `View` 的播放状态参数和旋转系统界面;
- 退出全屏时移除 `DecorView` 中的 `Surface` ，切换 List Item 中的 `Surface` 给 Player Core ，同步状态。

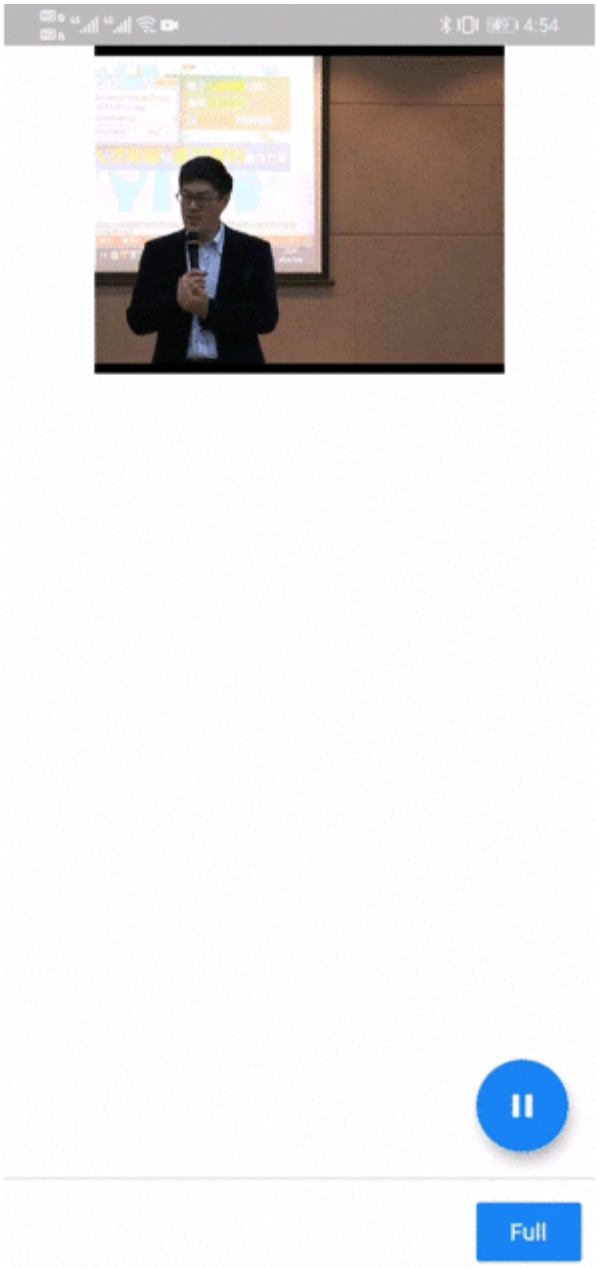


当然，不同的播放内核可能还需要做一些额外操作，**但是这一切在 Flutter 中就变得极为简单。**

```
事实上 Flutter 中实现全屏切换效果很简单，后面会一并介绍为什么在 Flut
```

二、实现效果

如下图所示是 Flutter 中实现后的全屏效果，而实现这个效果的关键就是**跳堆栈就可以了！是的，Flutter 中简单地跳页面就能够实现无缝的全屏切换。**



如下代码所示，首先在正常播放页面下加入官方 `video_player` 插件的 `VideoPlayer` 控件，并且初始化 `VideoPlayerController` 用于加载需要播放的视频并初始化，另外此处还用了 `Hero` 控件用于实现页面跳转过渡的动画效果。

```
  @override
  void initState() {
    super.initState();
    _controller = VideoPlayerController.network(
        'https://res.exexm.com/cw_145225549855002')
      ..initialize().then((_) {
        // Ensure the first frame is shown after the video
        setState(() {});
      });
  }

 Container(
   height: 200,
   margin: EdgeInsets.only(
       top: MediaQueryData.fromWindow(
               WidgetsBinding.instance.window)
           .padding
           .top),
   color: Colors.black,
   child: _controller.value.initialized
       ? Hero(
           tag: "player",
           child: AspectRatio(
             aspectRatio: _controller.value.aspectRatio,
             child: VideoPlayer(_controller),
           ),
         )
       : Container(
           alignment: Alignment.center,
           child: CircularProgressIndicator(),
         ),
 ))
```

如下代码所示，之后在全屏的页面中同样使用 `Hero` 控件和 `VideoPlayer` 控件实现过渡动画和视频渲染。

这里的 `VideoPlayerController` 可以通过构造方法传递进来，也可以通过 `InheritedWidget` 实现共享传递，只要是和前面普通播放界面的 `controller` 是同一个即可。

```
Container(
     color: Colors.black,
     child: Stack(
       children: <Widget>[
         Center(
           child: Hero(
             tag: "player",
             child: AspectRatio(
               aspectRatio: widget.controller.value.aspectR
               child: VideoPlayer(widget.controller),
             ),
           ),
         ),
         Padding(
           padding: EdgeInsets.only(top: 25, right: 20),
           child: IconButton(
             icon: const BackButtonIcon(),
             color: Colors.white,
             onPressed: () {
               Navigator.pop(context);
             },
           ),
         )
       ],
     ),
    )
```

另外在 Flutter 中，只需要通过 `SystemChrome.setPreferredOrientations` 方法就可以快速实现应用的横竖屏切换。

最后如下代码所示，只需要通过 `Navigator` 调用页面跳转就可以实现全屏和非全屏的无缝切换了。

```
  Navigator.of(context)
                      .push(MaterialPageRoute(builder: (con
                    return VideoFullPage(_controller);
                  }));
```

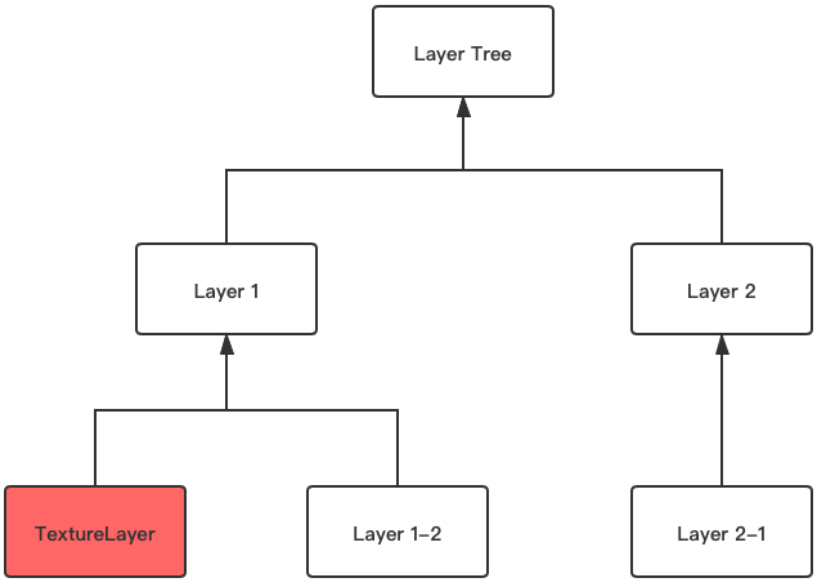
是不是很简单，只需要 `VideoPlayer` 、 `Hero` 和 `Navigator` 就可以快速实现全屏切换播放的效果，**那为什么在 Flutter 上可以这样简单的实现呢?**

三、实现逻辑

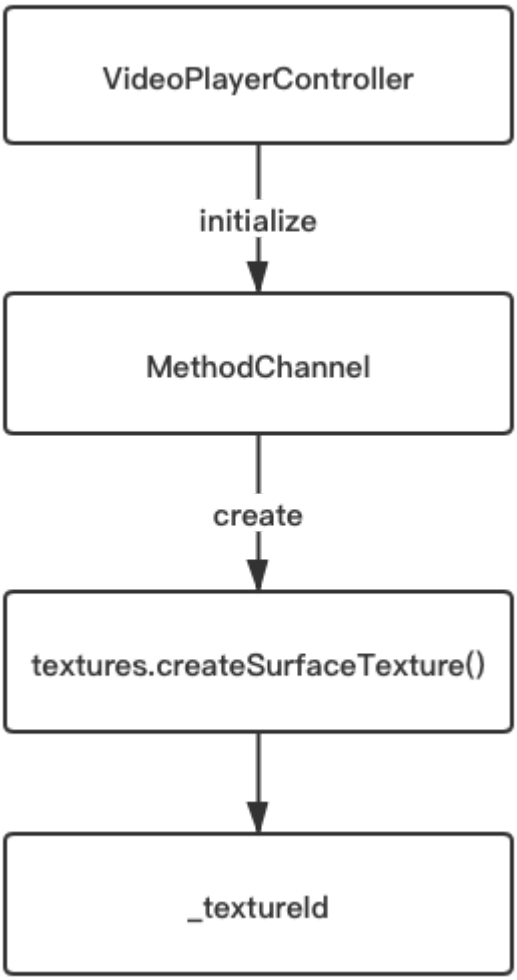
之所以可以如此简单地实现动态化全屏效果，其实主要涉及到 `video_player` 插件在 Flutter 上的实现：**外接纹理 `Texture`** 。

因为 Flutter 中的控件基本上是平台无关的，而其控件主要是由 Flutter Engine 直接绘制，简单地说就是：**原生平台仅仅提供了一个 `Activity` / `ViewController` 容器，之后由容器内提供一个 `Surface` 给 Flutter Engine 用户绘制。**

所以 Flutter 中控件的渲染堆栈是独立的，没办法和原生平台直接混合使用，这时候为了能够在 Flutter 中插入原生平台的部分功能，**Flutter 除了提供了 `PlatformView` 这样的实现逻辑之外，还提供了 `Texture` 作为外接纹理的支持。**



如上图所示，在《Flutter 完整实战详解》 中介绍过，**Flutter 的界面渲染是需要经历 `Widget` -> `RenderObject` -> `Layer` 的过程**，而在 `Layer` 的渲染过程中，当出现一个 `TextureLayer` 节点时，说明这个节点使用了 Flutter 中的 `Texture` 控件，那么这个控件的内容就会由原生平台提供，而**管理 `Texture` 主要是通过 `textureId` 进行识别的。**

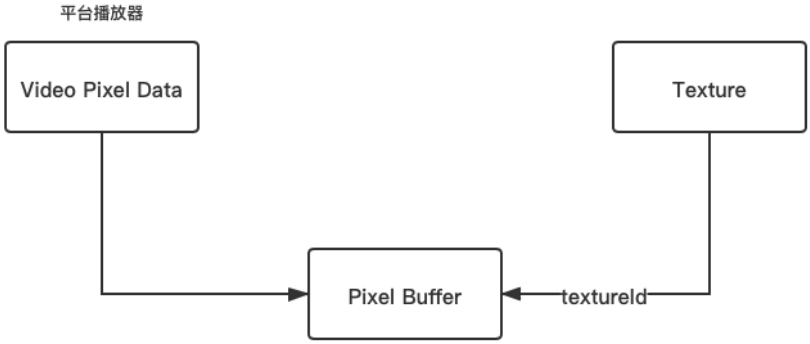


举个例子，在 Android 原生层中 `video_player` 使用的是 `exoplayer` 播放内核，那么如上图所示， **`VideoPlayerController` 会在初始化的时候通过 `MethodChannel` 和原生端通信，之后准备好播放内核和 `Surface` ，最后将对应的 `textureId` 返回到 Dart 中。**

所以在前面的代码中，需要在全屏和非全屏页面使用同一个 `VideoPlayerController` ，这样它们就具备了同一个 `textureId` 。

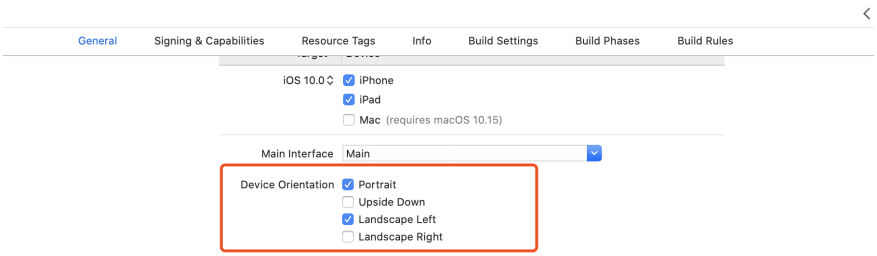
具备同一个 `textureId` 后，那么只要原生层不停止播放，`textureId` 对应的原生数据就一直处于更新状态，而这时候虽然跳转路由页面，但不同的 `VideoPlayer` 内部的 `Texture` 控件用的是同一个 `VideoPlayerController` ，也就是同一个 `textureId` ，所以它们会呈现出通用的画面。

如下图所示，这个过程简单总结就是： **Flutter 和原生平台通过 `PixelBuffer` 为介质进行交互，原生层将数据写入 `PixelBuffer` ，Flutter 通过注册好的 `textureId` 获取到 `PixelBuffer` 之后由 Flutter Engine 绘制。**



最后需要注意的是，在 iOS 上在实现页面旋转时，`SystemChrome.setPreferredOrientations` 方法可能会出现无效，这个问题在 issue #23913 和 #13238 中有提及，这里可能需要自己多实现一个原生接口进行兼容，当然在 auto_orientation 或者 orientation 等第三方库也进行了这方面的兼容。

另外 iOS 的页面旋转还确定是否打开了旋转配置的开关。

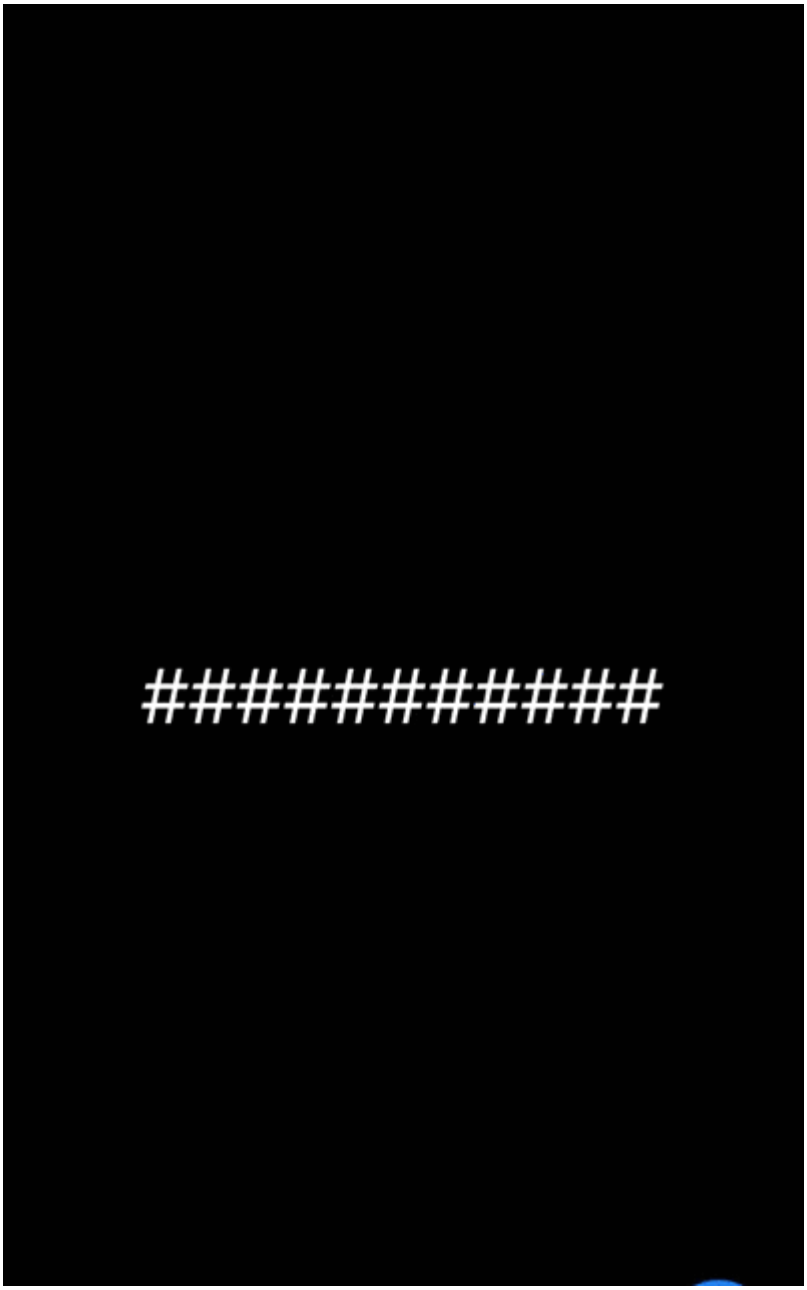


资源推荐

- 本文 Demo： flutter_video_full_controller
- 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


GSYTech
移动开发的后花园，变强不秃。
长按二维码关注公众号
@恋猫de小郭
忧愁时写一篇文章
快乐时编一段人生
无论工作如何艰辛
我们相信分享总是美好的开始
Flutter 完整开发实战详解
GSYTech
Flutter完整开发实战详解(十八、神奇的...
Flutter完整开发实战详解(十七、实用技巧与填坑二)
Flutter 完整开发实战详解（十六、详解自定义布局...
Flutter完整开发实战详解(十五、全面理解State与...
Flutter
程序猿的故事
GSYTech
六年码农生涯的 2019 总结：君子坐而论道，少年...
2019大前端浪潮下的年度终结
一个程序员的五年总结，给你不一样的角度
程序员郭小喵创作故事
程序猿郭小喵曾经的实习故
Android 巴士站
GSYTech
移动开发者的音视频基础知识
做Android开发的，想转行，如何选择呢?
Android程序员的Java后台学习建议
Flutter 杂谈与开发大会
GSYTech
Flutter 会不会被苹果限制其发展?
Flutter Interact 的 Flutter 1.12 大进化和回顾
Flutter 框架在国内移动端市场的发展趋势，在游戏...
Flutter 升级 1.12 适配教程
Flutter 跨平台框架应用实

事情是这样的，由于近期 Flutter 发布了 `1.17` 的稳定版，按照“惯例”开始着手把生产项目升级到 `1.12.13+hotfix.9` 版本，在升级适配完成之后，一个突如其来的 Bug 让我陷入了沉思。



如上图所示，可以看到在键盘 B 页面打开后，退回上一个页面 A 时键盘已经收起，但是原先键盘所在的区域在 A 页面变成了空白，而 A 页面内容也被 `resize` 成了键盘弹出后的大小。

1、Scaffold

针对这个问题，首先想到的 `Scaffold` 的 `resizeToAvoidBottomInset` 属性。

在 Flutter 中 `Scaffold` 默认情况下 `resizeToAvoidBottomInset` 为 `true`，当 `resizeToAvoidBottomInset` 为 `true` 时，`Scaffold` 内部会将 `mediaQuery.viewInsets.bottom` 参与到 `BoxConstraints` 的大小计算，也就是**键盘弹起时调整了内部的 `bottom` 位置来迎合键盘。**

但是问题发送在 A 界面，这时候键盘已经收起，`mediaQuery.viewInsets.bottom` 应该更新为 0，那为何界面没有产生应有的更新呢?

2、MediaQuery

那么猜测问题可能出现在 `MediaQuery` 上。

从源码我们得知 `MediaQuery` 是一个 `InheritedWidget`，它会往下共享对应的 `MediaQueryData`，在 `MediaQueryData` 中保存了各种设备的信息，比如 `size`、`devicePixelRatio`、`textScaleFactor`、`viewPadding` 以及 `viewInsets` 等。

那 `viewInsets` 是什么的呢？官方的解释是：

> “可以被系统显示的区域，通常是和设备的键盘等相关，当键盘弹出时 `viewInsets.bottom` 对应的就是键盘的顶部。”

那上面的 bug 看起来可能就是 `Scaffold` 的 `viewInsets.bottom` 在键盘收起来时没有正常重置。

3、Window

那这里首先我们要知道 `MediaQuery` 的 `viewInsets` 是怎么被设置的?

通过分析源码可以知道 **`MediaQuery` 的 `MediaQueryData` 来源于 `WidgetsBinding.instance.window`**，默认是在 `MaterialApp` 的 `_MediaQueryFromWindow` 中被设置：

```
  @override
  void didChangeMetrics() {
    setState(() {
      // The properties of window have changed. We use ther
      // function, so we need setState(), but we don't cach
    });
  }

  @override
  Widget build(BuildContext context) {
    return MediaQuery(
      data: MediaQueryData.fromWindow(WidgetsBinding.instar
      child: widget.child,
    );
  }
```

如上代码可以看到 `MediaQuery` 的 `MediaQueryData` 是来源于 `Window`，并且这里还注册了 `WidgetsBindingObserver` 的 `didChangeMetrics` 回调，也就是当 `window` 改变时，调用 `setState` 来更新 `MediaQuery` 中的 `MediaQueryData` 。

而在 `MediaQueryData.fromWindow` 中， `viewInsets` 是通过将 `window.viewInsets` 和 `window.devicePixelRatio` 相除后得到的像素密度值。

```
viewInsets = EdgeInsets.
fromWindowPadding(window.viewInsets, window.devicePixelRati
```

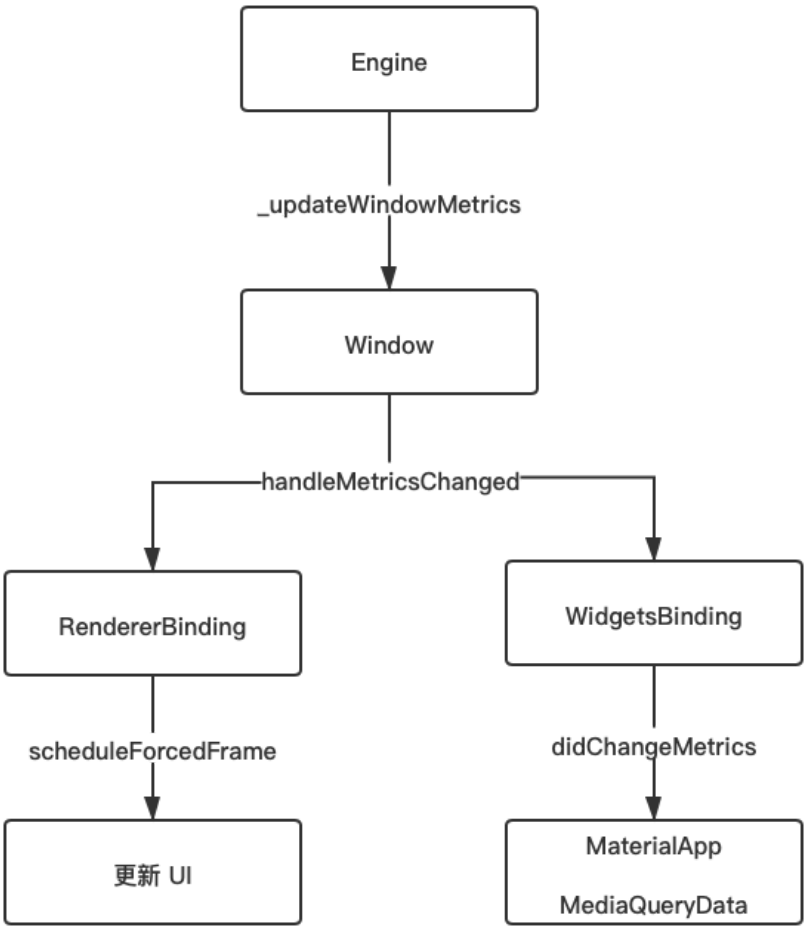
那 `Window` 的值又是哪里来的?

其实 **`Window` 的值来源于 Flutter Engine，在键盘弹出时 Flutter Engine 会通过 `_updateWindowMetrics` 方法更新 `Window` 数据，并执行 `window.onMetricsChanged` 和 `window._onMetricsChangedZone` 方法。**

其中 `onMetricsChanged` 回调最终会触发 `handleMetricsChanged` 方法，从而执行 `scheduleForcedFrame()` 更新界面和 `observer.didChangeMetrics();` 通知 `MaterialApp` 中的 `MediaQueryData` 更新。

```
@pragma('vm:entry-point')
// ignore: unused_element
void _updateWindowMetrics(
  double devicePixelRatio,
  double width,
  double height,
  double depth,
  double viewPaddingTop,
  double viewPaddingRight,
  double viewPaddingBottom,
  double viewPaddingLeft,
  double viewInsetTop,
  double viewInsetRight,
  double viewInsetBottom,
  double viewInsetLeft,
  double systemGestureInsetTop,
  double systemGestureInsetRight,
  double systemGestureInsetBottom,
  double systemGestureInsetLeft,
) {
  window
    .._devicePixelRatio = devicePixelRatio
    .._physicalSize = Size(width, height)
    .._physicalDepth = depth
    .._viewPadding = WindowPadding._(
        top: viewPaddingTop,
        right: viewPaddingRight,
        bottom: viewPaddingBottom,
        left: viewPaddingLeft)
    .._viewInsets = WindowPadding._(
        top: viewInsetTop,
        right: viewInsetRight,
        bottom: viewInsetBottom,
        left: viewInsetLeft)
    .._padding = WindowPadding._(
        top: math.max(0.0, viewPaddingTop - viewInsetTop),
        right: math.max(0.0, viewPaddingRight - viewInsetR
        bottom: math.max(0.0, viewPaddingBottom - viewInse
        left: math.max(0.0, viewPaddingLeft - viewInsetLef
    .._systemGestureInsets = WindowPadding._(
        top: math.max(0.0, systemGestureInsetTop),
        right: math.max(0.0, systemGestureInsetRight),
        bottom: math.max(0.0, systemGestureInsetBottom),
        left: math.max(0.0, systemGestureInsetLeft));
  _invoke(window.onMetricsChanged, window._onMetricsChange
}
```

所以可以看到，当键盘弹出和收起时， `Engine` 会更新 `Window` 的数据， `Window` 触发界面绘制更新，同时更新 `MaterialApp` 中的 `MediaQueryData` 。



4、Route

那按照这个情况，不可能出现上述键盘导致空白区域的问题，那问题**可能就是出现在 `Scaffold` 使用的 `MediaQueryData` 没有更新**。

这时候我突然想起，之前为了锁定页面的字体大小不跟随系统缩放，我在路由层使用了 `MediaQueryData.fromWindow` 复制一份 `MediaQuery` ，问题很可能出在这里：

```
Navigator.of(context).push(new CupertinoPageRoute(builder:
   return MediaQuery(
      data:MediaQueryData.fromWindow(WidgetsBinding.instanc
                          .copyWith(textScaleFactor: 1),
                  child: Page2(), );
   }));
```

不过这也不对，出现问题的是有键盘的 B 页面返回到没有键盘的 A 页面，这时候 A 页面已经打开，那之前打开 A 页面的 `WidgetsBinding.instance.window` 应该是对的，而 **A 页面所在的 `CupertinoPageRoute` 的 `builder` 方法，不可能在键盘 B 页面打开时再次被执行才对?**

但是在经过调试后震惊的发现，程序在进入 B 页面弹出键盘后，居然会触发了 A 页面 `CupertinoPageRoute` 的 `builder` 方法重新执行。

能够在跨页面触发更新，第一个想到的就是全局的状体管理框架，因为应用需要全局切换*主题*、*多语言和用户信息共享*等，在应用的顶层一般会通过状体管理框架往下共享和管理这些信息。

由于原本项目比较复杂，所以重新做了一个简单的测试 Demo ，并且引入比较简单的 `ScopedModel` 框架管理，然后在打开有键盘的 B 页面后执行延时一会执行 `notifyListeners();` ，发现果然出现了同样的问题。

```
    return ScopedModel(
      model: t,
      child: ScopedModelDescendant<TestModel>(
        builder: (context, child, model) {
          return MaterialApp(
            title: 'Flutter Demo',
            theme: ThemeData(
              primarySwatch: Colors.blue,
            ),
            home: MyHomePage(title: 'Flutter Demo Home Page
          );
        },
      ),
    );
```

5、Navigator

这里不禁就有疑问，为什么 `MaterialApp` 的更新会导致 `PageRoute` 重新 `builder` 呢?

这就涉及 `Navigator` 的相关逻辑，我们常用的 `Navigator` 其实是一个 `StatefulWidget` ，当 `MaterialApp` 被更新时，可以看到在 `NavigatorState` 的 `didUpdateWidget` 回调中会调用 `_history` 里所有路由的 `changedExternalState()` 方法。

```
@override
 void didUpdateWidget(Navigator oldWidget) {
   super.didUpdateWidget(oldWidget);
   if (oldWidget.observers != widget.observers) {
     for (NavigatorObserver observer in oldWidget.observer
       observer._navigator = null;
     for (NavigatorObserver observer in widget.observers)
       assert(observer.navigator == null);
       observer._navigator = this;
     }
   }
   for (Route<dynamic> route in _history)
     route.changedExternalState();
 }
```

而 `changedExternalState` 执行后会调用 `_forceRebuildPage` 将路由里的 `_page` 清空，这样自然下次 `Route` 在 `build` 时触发的 `PageRoute` 重新 `builder` 方法。

```
@override
 void changedExternalState() {
   super.changedExternalState();
   if (_scopeKey.currentState != null)
     _scopeKey.currentState._forceRebuildPage();
 }

.....

 void _forceRebuildPage() {
   setState(() {
     _page = null;
   });
 }
```

所以回归到最初的问题：**这个 bug 首先是因为不规范使用了 `MediaQueryData.fromWindow(WidgetsBinding.instance.window)`，之后又恰好在有键盘的页面打开后触发了 `MaterialApp` 的更新，导致了 `PageRoute` 重新 `builder`，使得没有键盘的 `Scaffold` 使用了弹出键盘的 `viewInsets.bottom`。**

所以这里只需要将 `MediaQueryData.fromWindow` 换成 `MediaQuery.of(context)` 就可以解决问题，而当在没有 `context` 或者需要直接使用 `MediaQueryData.fromWindow` 时，那一定要搭配上 `WidgetsBindingObserver.didChangeMetrics` 配合更新。

```
    Navigator.of(context).push(new CupertinoPageRoute(builc
      return MediaQuery(
        data:MediaQuery.of(context)
            .copyWith(textScaleFactor: 1),
        child: Page2(), );
    }));
```

最后说一句，虽然这个 bug 并不复杂，但是恰好能带出挺多经常忽略的知识点，所以长篇介绍这么多，也希望这样的 bug 解决思路，可以帮助到大家在日常开发过程中解决更多问题。

我们都知道在 Flutter 中可以通过 `fontFamily` 来引入第三方字体，例如**通常会将 svg 图标转换为 `iconfont.ttf` 来实现矢量图标的入**，而一般情况下我们是不会设置 `fontFamily` 来使用第三方字体， 那默认情况下 Flutter 使用的是什么字体呢?

会出现这个疑问，是因为有一天设计给我发了下面那张图，问我 *“为什么应用在苹果平台上的英文使用的是 `PingFang SC` 字体而不是 `.SF UI Display`”*？ 正如下图所示，它们的 *G* 字母在显示效果上会有所差异，比如 平方的 *G* 有明显的转折线。

这时候我不禁产生的好奇，在 Flutter 中引擎默认究竟是如何选择字体?

通过官方解释，在 `typography.dart` 源码中可以看到，

- Flutter 默认在 Android 上使用的是 `Roboto` 字体；
- 在 iOS 上使用的是 `.SF UI Display` 或者 `.SF UI Text` 字体。

> The default font on Android is Roboto and on iOS it is .SF UI Display or .SF UI Text (SF meaning San Francisco). If you want to use a different font, then you will need to add it to your app.

```
static const TextTheme blackCupertino = TextTheme(
  display4   : TextStyle(debugLabel: 'blackCupertino display4',   fontFamily: '.SF UI Display', inherit: true, color: Colors.black54, decoration: TextDecoration.none),
  display3   : TextStyle(debugLabel: 'blackCupertino display3',   fontFamily: '.SF UI Display', inherit: true, color: Colors.black54, decoration: TextDecoration.none),
  display2   : TextStyle(debugLabel: 'blackCupertino display2',   fontFamily: '.SF UI Display', inherit: true, color: Colors.black54, decoration: TextDecoration.none),
  display1   : TextStyle(debugLabel: 'blackCupertino display1',   fontFamily: '.SF UI Display', inherit: true, color: Colors.black54, decoration: TextDecoration.none),
  headline   : TextStyle(debugLabel: 'blackCupertino headline',   fontFamily: '.SF UI Display', inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  title      : TextStyle(debugLabel: 'blackCupertino title',      fontFamily: '.SF UI Display', inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  subhead    : TextStyle(debugLabel: 'blackCupertino subhead',    fontFamily: '.SF UI Text',    inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  body2      : TextStyle(debugLabel: 'blackCupertino body2',      fontFamily: '.SF UI Text',    inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  body1      : TextStyle(debugLabel: 'blackCupertino body1',      fontFamily: '.SF UI Text',    inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  caption    : TextStyle(debugLabel: 'blackCupertino caption',    fontFamily: '.SF UI Text',    inherit: true, color: Colors.black54, decoration: TextDecoration.none),
  button     : TextStyle(debugLabel: 'blackCupertino button',     fontFamily: '.SF UI Text',    inherit: true, color: Colors.black87, decoration: TextDecoration.none),
  subtitle   : TextStyle(debugLabel: 'blackCupertino subtitle',   fontFamily: '.SF UI Text',    inherit: true, color: Colors.black,   decoration: TextDecoration.none),
  overline   : TextStyle(debugLabel: 'blackCupertino overline',   fontFamily: '.SF UI Text',    inherit: true, color: Colors.black,   decoration: TextDecoration.none),
);

/// A material design text theme with light glyphs based on San Francisco.
///
/// This [TextTheme] provides color but not geometry (font size, weight, etc).
static const TextTheme whiteCupertino = TextTheme(
  display4   : TextStyle(debugLabel: 'whiteCupertino display4',   fontFamily: '.SF UI Display', inherit: true, color: Colors.white70, decoration: TextDecoration.none),
  display3   : TextStyle(debugLabel: 'whiteCupertino display3',   fontFamily: '.SF UI Display', inherit: true, color: Colors.white70, decoration: TextDecoration.none),
  display2   : TextStyle(debugLabel: 'whiteCupertino display2',   fontFamily: '.SF UI Display', inherit: true, color: Colors.white70, decoration: TextDecoration.none),
  display1   : TextStyle(debugLabel: 'whiteCupertino display1',   fontFamily: '.SF UI Display', inherit: true, color: Colors.white70, decoration: TextDecoration.none),
  headline   : TextStyle(debugLabel: 'whiteCupertino headline',   fontFamily: '.SF UI Display', inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  title      : TextStyle(debugLabel: 'whiteCupertino title',      fontFamily: '.SF UI Display', inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  subhead    : TextStyle(debugLabel: 'whiteCupertino subhead',    fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  body2      : TextStyle(debugLabel: 'whiteCupertino body2',      fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  body1      : TextStyle(debugLabel: 'whiteCupertino body1',      fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  caption    : TextStyle(debugLabel: 'whiteCupertino caption',    fontFamily: '.SF UI Text',    inherit: true, color: Colors.white70, decoration: TextDecoration.none),
  button     : TextStyle(debugLabel: 'whiteCupertino button',     fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  subtitle   : TextStyle(debugLabel: 'whiteCupertino subtitle',   fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
  overline   : TextStyle(debugLabel: 'whiteCupertino overline',   fontFamily: '.SF UI Text',    inherit: true, color: Colors.white,   decoration: TextDecoration.none),
);
```

那理论上在 iOS 使用的就是 `.SF UI Display` 字体才对，因为如下源码所示，在 `Typography` 中当 `platform` 是 `iOS` 时，使用的就是 `Cupertino` 相关的 `TextTheme`，而 `Typography` 中的 `white` 和 `black` 属性最终会应用到 `ThemeData` 的 `defaultTextTheme`、`defaultPrimaryTextTheme` 和 `defaultAccentTextTheme` 中，所以应该是使用 `.SF` 相关字体才会，为什么会显示的是 `PingFang SC` 的效果?

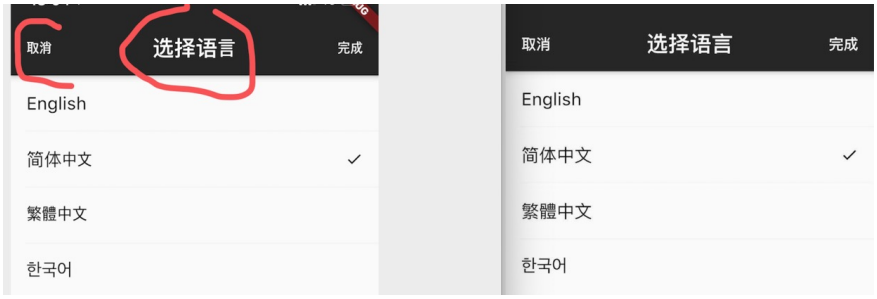
```
factory Typography({
   TargetPlatform platform = TargetPlatform.android,
   TextTheme black,
   TextTheme white,
   TextTheme englishLike,
   TextTheme dense,
   TextTheme tall,
 }) {
   assert(platform != null || (black != null && white != r
   switch (platform) {
     case TargetPlatform.iOS:
       black ??= blackCupertino;
       white ??= whiteCupertino;
       break;
     case TargetPlatform.android:
     case TargetPlatform.fuchsia:
       black ??= blackMountainView;
       white ??= whiteMountainView;
   }
   englishLike ??= englishLike2014;
   dense ??= dense2014;
   tall ??= tall2014;
   return Typography._(black, white, englishLike, dense, t
 }
```

为了搞清不同系统上字体的区别，在查阅了资料后可知：

- 默认在 iOS 上：
  - 中文字体：`PingFang SC`
  - 英文字体：`.SF UI Text` 、`.SF UI Display`
- 默认在 Android 上：
  - 中文字体：`Source Han Sans` / `Noto`
    - 英文字体：`Roboto`

也就是就 iOS 上除了 `.SF` 相关的字体外，还有 `PingFang` 字体的存在，这时候我突然想起在之前的 《Flutter完整开发实战详解(十七、 实用技巧与填坑二)》 中，因为国际化多语言在 `.SF` 会出现显示异常，所以使用了 `fontFamilyFallback` 强行指定了 `PingFang SC` 。

```
  getCopyTextStyle(TextStyle textStyle) {
    return textStyle.copyWith(fontFamilyFallback: ["PingFar
  }
```



终于破案了，因为当 `fontFamily` 没有设置时，就会使用 `fontFamilyFallback` 中的第一个值将作为首选字体，而在 `fontFamilyFallback` 中是顺序匹配的，当 `fontFamily` 和 `fontFamilyFallback` 两者都不提供，则将使用默认平台字体。

而在 1.12.13 版本下测试发现 `.SF` 导致的问题已经修复了，所以只需要将 `fontFamilyFallback` 相关的代码去除即可。

那在 iOS 上使用 `.SF` 字体有什么好处？ 按照网络上的说法是：

> `SF Text` 的字距及字母的半封闭空间，比如 `"a"!` 上半部分会更大，因其可读性更好，适用于更小的字体; `SF Display` 则适用于偏大的字体。具体分水岭就是 `20pt`，即字体小于 `20pt` 时用 `Text`，大于等于 `20pt` 时用 `Display`。
>
> 更棒的是由于 `SF` 属于动态字体，`Text` 和 `Display` 两种字体族是系统动态匹配的，也就是说你不用费心去自己手动调节，系统自动根据字体的大小匹配这两种显示模式。

那能不能在 Android 上也使用 `.SF` 字体呢？按照官方的说法：

- 在使用 Material package 时，在 Android 上使用的是 Roboto font，而 iOS 使用的是 `San Francisco font(SF)`；
- 在使用 Cupertino package 时，默认主题始终使用 `San Francisco font(SF)`；

但是因为 `San Francisco font license` 限制了该字体只能在 iOS、macOS 或 tvOS 上运行使用，所以如果使用了 Cupertino 主题的话，在 Android 上运行时使用 fallback font。

所以你觉得能不能在 Android 上使用?

最后再补充下，在官方的 architecture 中有提到，在 Flutter 中的文本呈现逻辑是有分层的，其中：

- 衍生自 Minikin 的 libtxt 库用于字体选择，分隔行等；
- HartBuzz 用于字形选择和成型；
- Skia作为 渲染 / GPU后端；
- 在 Android / Fuchsia 上使用 FreeType 渲染，在 iOS 上使用 CoreGraphics 来渲染字体 。

> 那读完本篇，你奇奇怪怪的知识点有没有增加?

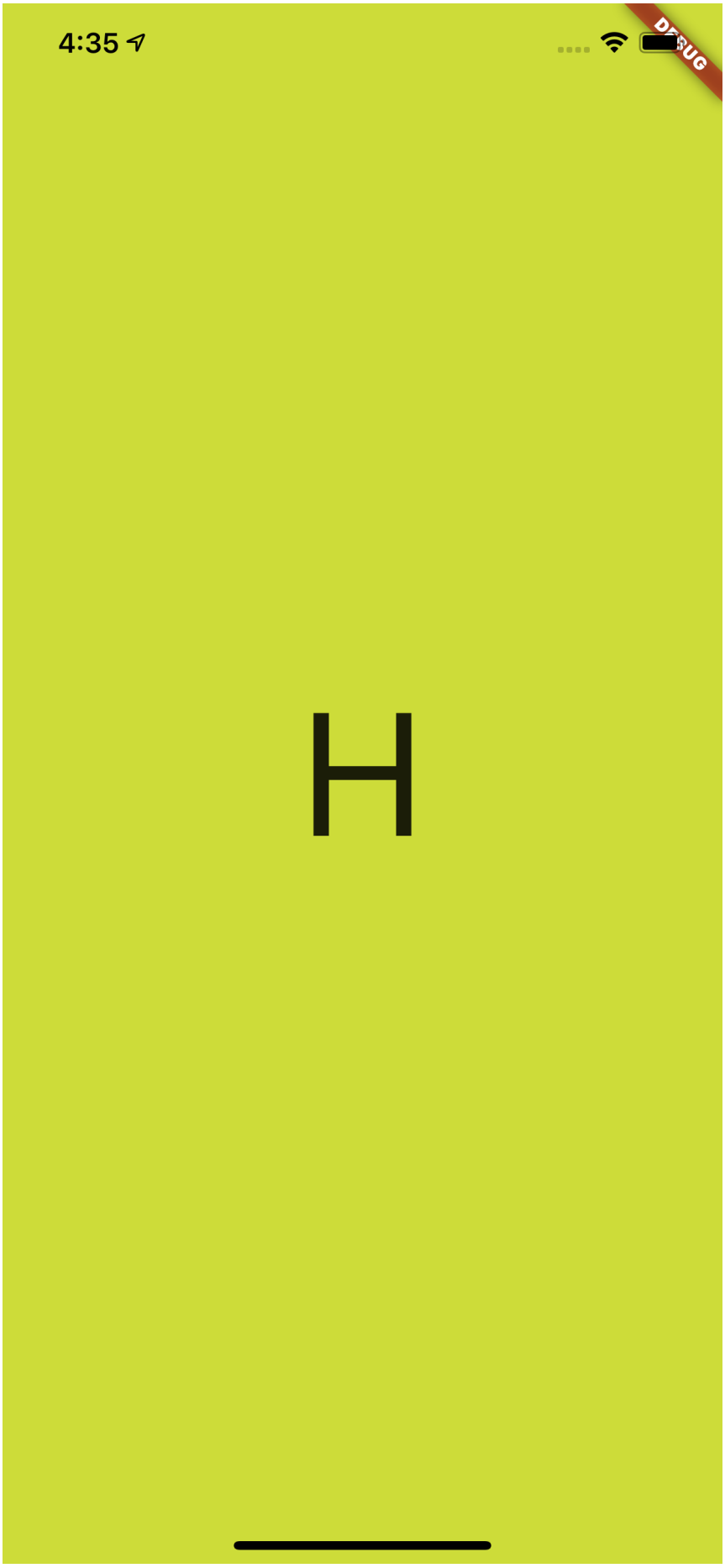

GSYTech
移动开发的后花园，变强不秃。
长按二维码关注公众号
@恋猫de小郭
忧愁时写一篇文章
快乐时编一段人生
无论工作如何艰辛
我们相信分享总是美好的开始
Flutter 完整开发实战详解
GSYTech
Flutter完整开发实战详解(十八、神奇的...
Flutter完整开发实战详解(十七、实用技巧与填坑二)
Flutter 完整开发实战详解（十六、详解自定义布局...
Flutter完整开发实战详解(十五、全面理解State与...
Flutter
程序猿的故事
GSYTech
六年码农生涯的 2019 总结：君子坐而论道，少年...
2019大前端浪潮下的年度终结
一个程序员的五年总结，给你不一样的角度
程序员郭小喵创作故事
程序猿郭小喵曾经的实习故
Android 巴士站
GSYTech
移动开发者的音视频基础知识
做Android开发的，想转行，如何选择呢?
Android程序员的Java后台学习建议
Flutter 杂谈与开发大会
GSYTech
Flutter 会不会被苹果限制其发展?
Flutter Interact 的 Flutter 1.12 大进化和回顾
Flutter 框架在国内移动端市场的发展趋势，在游戏...
Flutter 升级 1.12 适配教程
Flutter 跨平台框架应用实

本篇将带你深入理解 Flutter 开发过程中关于字体和文本渲染的“冷”知识，帮助你理解和增加关于 Flutter 中字体绘制的“无用”知识点。

> 毕竟此类相关的内容太少了

首先从一个简单的文本显示开始，如下代码所示，运行后可以看到界面内出现了一个 **H** 字母，它的 `fontSize` 是 **100**，`Text` 被放在一个高度为 **200** 的 `Container` 中，然后如果这时候有人问你：**`Text` 显示 H 字母需要占据多大的高度，你知道吗?**

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.black,
    body: Container(
      color: Colors.lime,
      alignment: Alignment.center,
      child: Container(
        alignment: Alignment.center,
        child: Container(
          height: 200,
          alignment: Alignment.center,
          child: new Row(
            children: <Widget>[
              Container(
                child: new Text(
                  "H",
                  style: TextStyle(
                    fontSize: 100,
                  ),
                ),
              ),
              Container(
                height: 100,
                width: 100,
                color: Colors.red,
              )
            ],
          ),
        )

      ),
    ),
  );
}
```

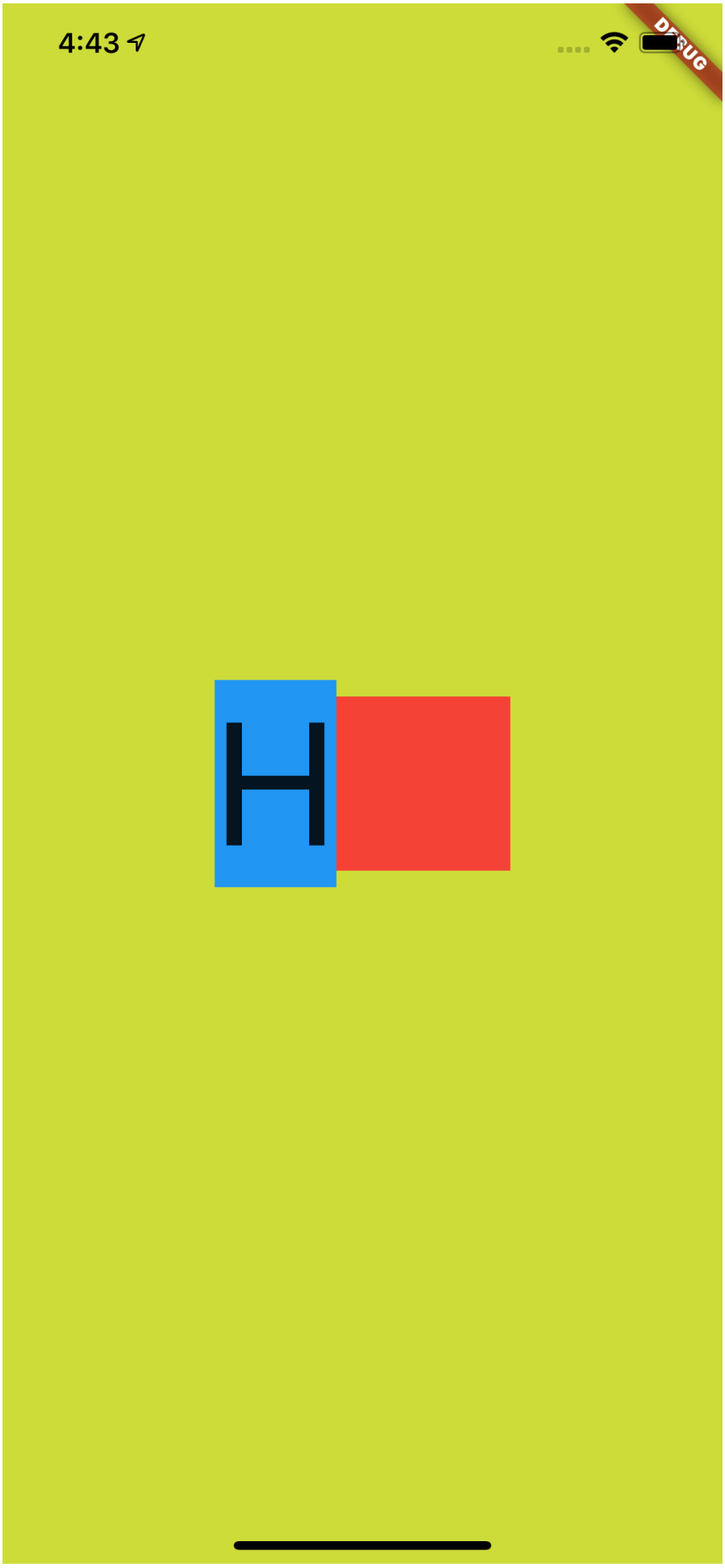

4:35
DEBUG
H

一、TextStyle

如下代码所示，为了解答这个问题，首先我们给 `Text` 所在的 `Container` 增加了一个蓝色背景，并增加一个 `100 * 100` 大小的红色小方块做对比。

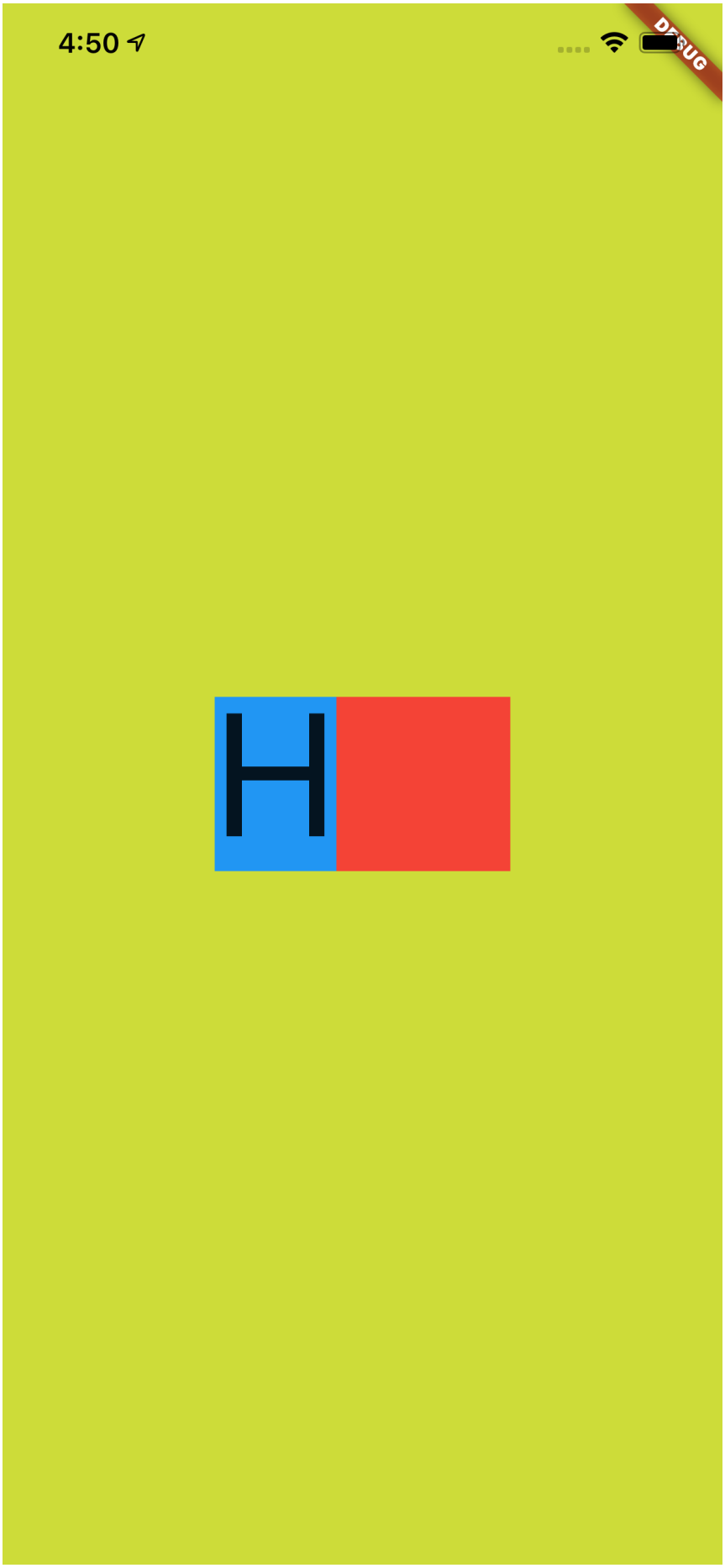
```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      body: Container(
        color: Colors.lime,
        alignment: Alignment.center,
        child: Container(
          alignment: Alignment.center,
          child: Container(
            height: 200,
            alignment: Alignment.center,
            child: new Row(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Container(
                  color: Colors.blue,
                  child: new Text(
                    "H",
                    style: TextStyle(
                      fontSize: 100,
                    ),
                  ),

                ),
                Container(
                  height: 100,
                  width: 100,
                  color: Colors.red,
                )
              ],
            ),
          )

        ),
      ),
    );
  }
```

结果如下图所示，可以看到 **H** 字母的上下有着一定的 `padding` 区域，蓝色 `Container` 的大小明显超过了 **100**，但是黑色的 **H** 字母本身并没有超过红色小方块，那蓝色区域的高度是不是 `Text` 的高度，它的大小又是如何组成的呢?


4:43
DEBUG

事实上，前面的蓝色区域是字体的行高，也就是 line height，关于这个行高，首先需要解释的就是 `TextStyle` 中的 `height` 参数。

默认情况下 `height` 参数是 `null`，当我们把它设置为 **`1`** 之后，如下图所示，可以看到蓝色区域的高度和红色小方块对齐，变成了 **100** 的高度，也就是行高变成了 **100**，而 **H** 字母完整的显示在蓝色区域内。


4:50
DEBUG

那 `height` 是什么呢？根据文档可知，首先 `TextStyle` 中的 `height` 参数值在设置后，其效果值是 `fontSize` 的倍数：

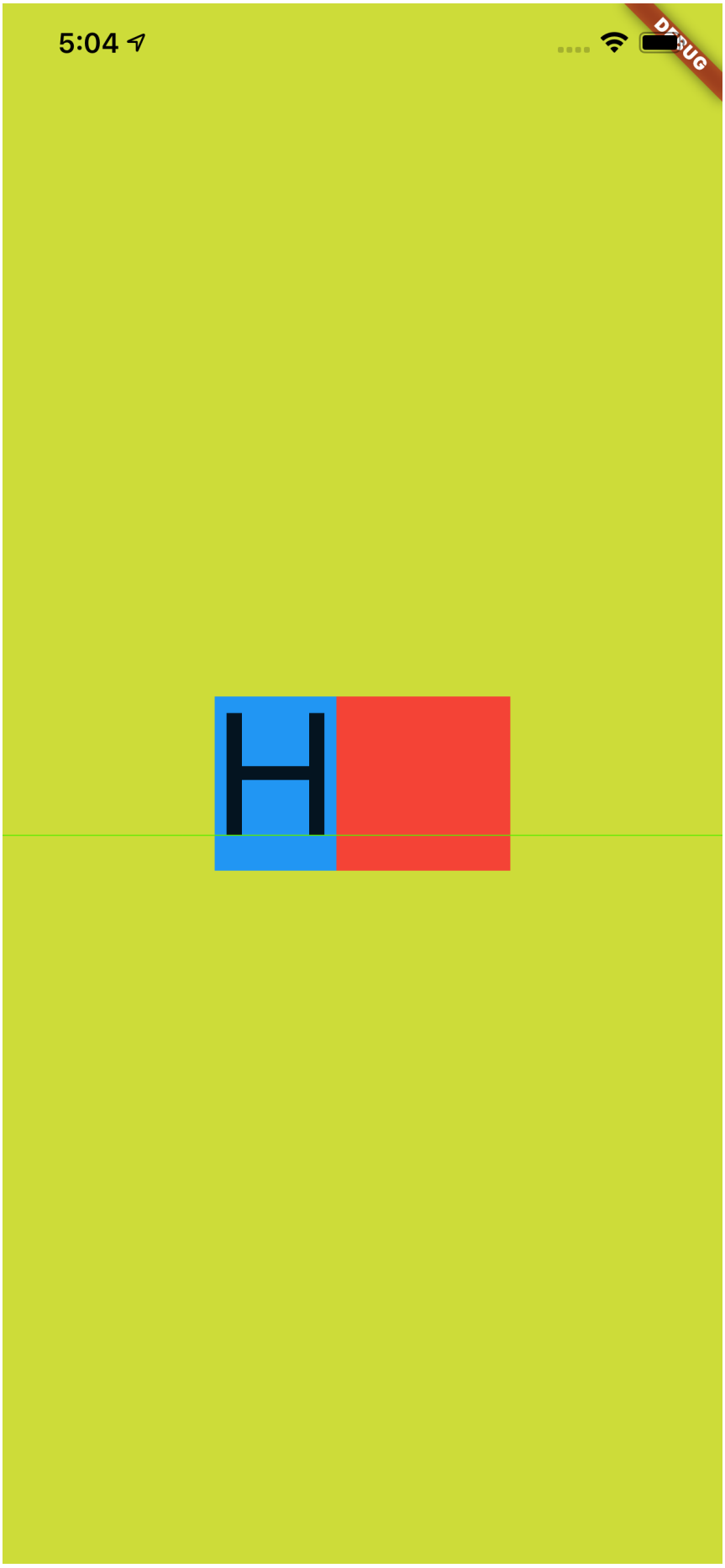
- 当 `height` 为空时，行高默认是使用字体的**量度**（这个**量度**后面会有解释）；
- 当 `height` 不是空时，行高为 `height` * `fontSize` 的大小；

如下图所示，蓝色区域和红色区域的对比就是 `height` 为 `null` 和 `1` 的对比高度。

另外上图的 `BaseLine` 也解释了：为什么 `fontSize` 为 100 的 **H** 字母，不是充满高度为 100 的蓝色区域。

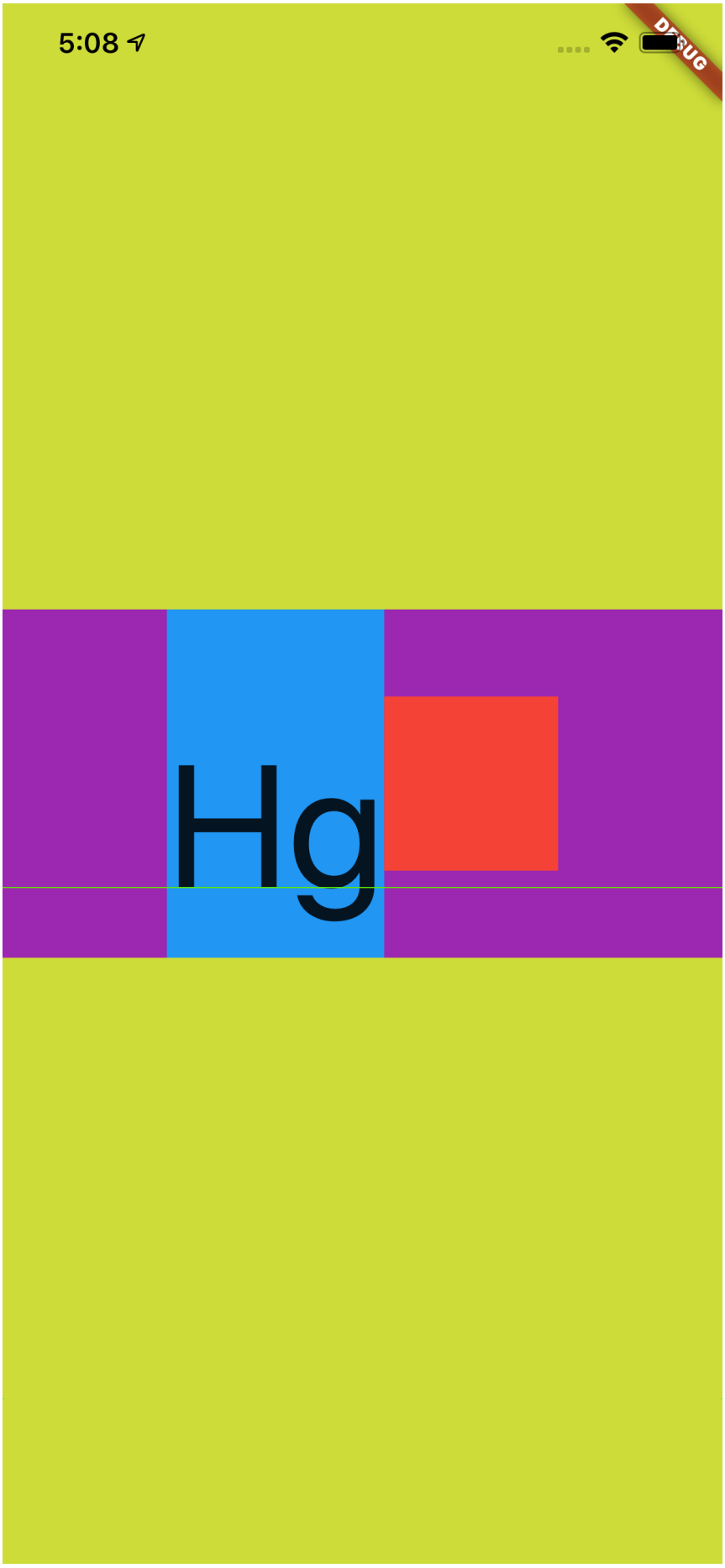
根据上图的示意效果，在 `height` 为 1 的红色区域内，**H** 字母也应该是显示在基线之上，而基线的底部区域是为了如 g 和 j 等字母预留，所以如下图所示，在 `Text` 内加入 g 字母并打开 Flutter 调试的文本基线显示，由 Flutter 渲染的绿色基线也可以看到符合我们预期的效果。

> 忘记截图由 g 的了，脑补吧。


5:04
DEBUG

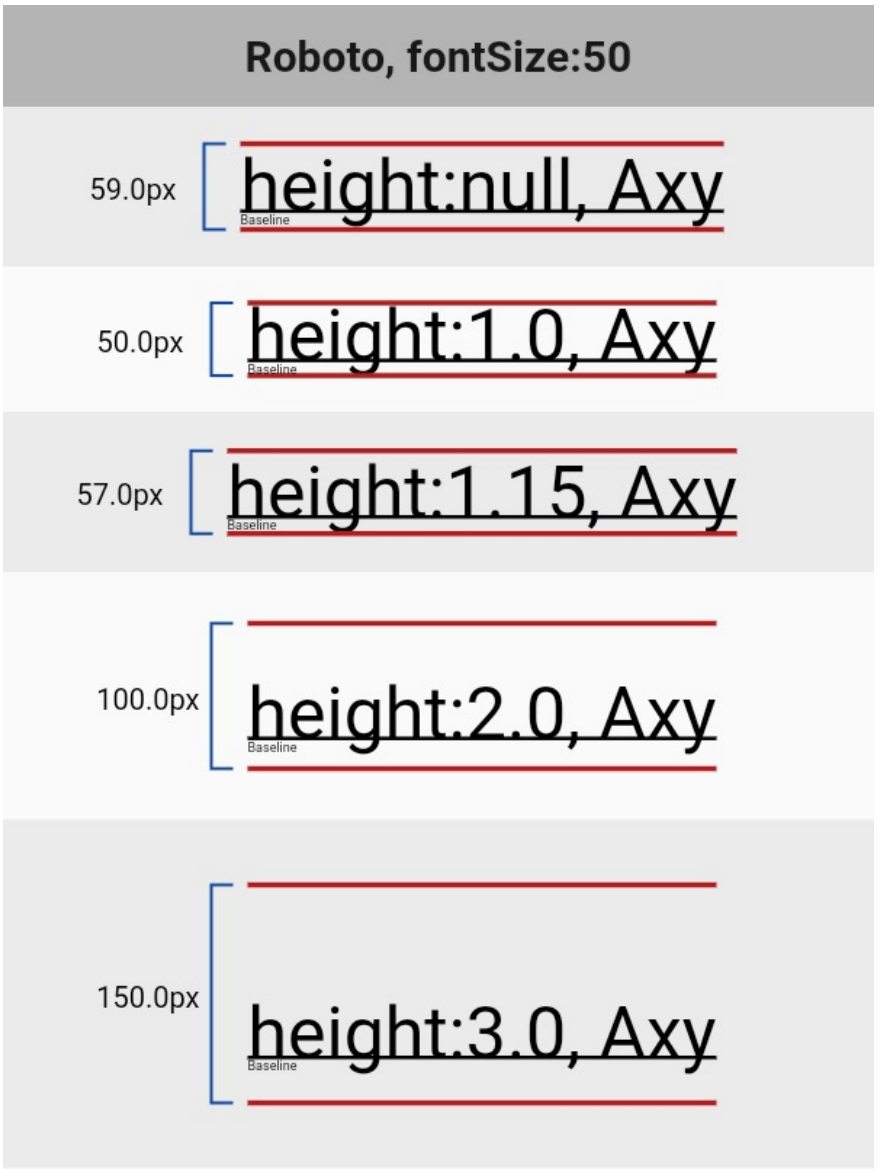
接着如下代码所示，当我们把 `height` 设置为 **`2`** ，并且把上层的高度为 **200** 的 `Container` 添加一个紫色背景，结果如下图所示，可以看到蓝色块刚好充满紫色方块，因为 `fontSize` 为 **100** 的文本在 **x2** 之后恰好高度就是 **200**。

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      body: Container(
        color: Colors.lime,
        alignment: Alignment.center,
        child: Container(
          alignment: Alignment.center,
          child: Container(
            height: 200,
            color: Colors.purple,
            alignment: Alignment.center,
            child: new Row(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Container(
                  color: Colors.blue,
                  child: new Text(
                    "Hg",
                    style: TextStyle(
                      fontSize: 100,
                      height: 2,
                    ),
                  ),

                ),
                Container(
                  height: 100,
                  width: 100,
                  color: Colors.red,
                )
              ],
            ),
          )

        ),
      ),
    );
  }
```


5:08
DEBUG
Hg

> 不过这里的 `Hg` 是往下偏移的，为什么这样偏移在后面会介绍，还会有新的对比。

最后如下图所示，是官方提供的在不同 `TextStyle` 的 `height` 参数下， `Text` 所占高度的对比情况。



二、StrutStyle

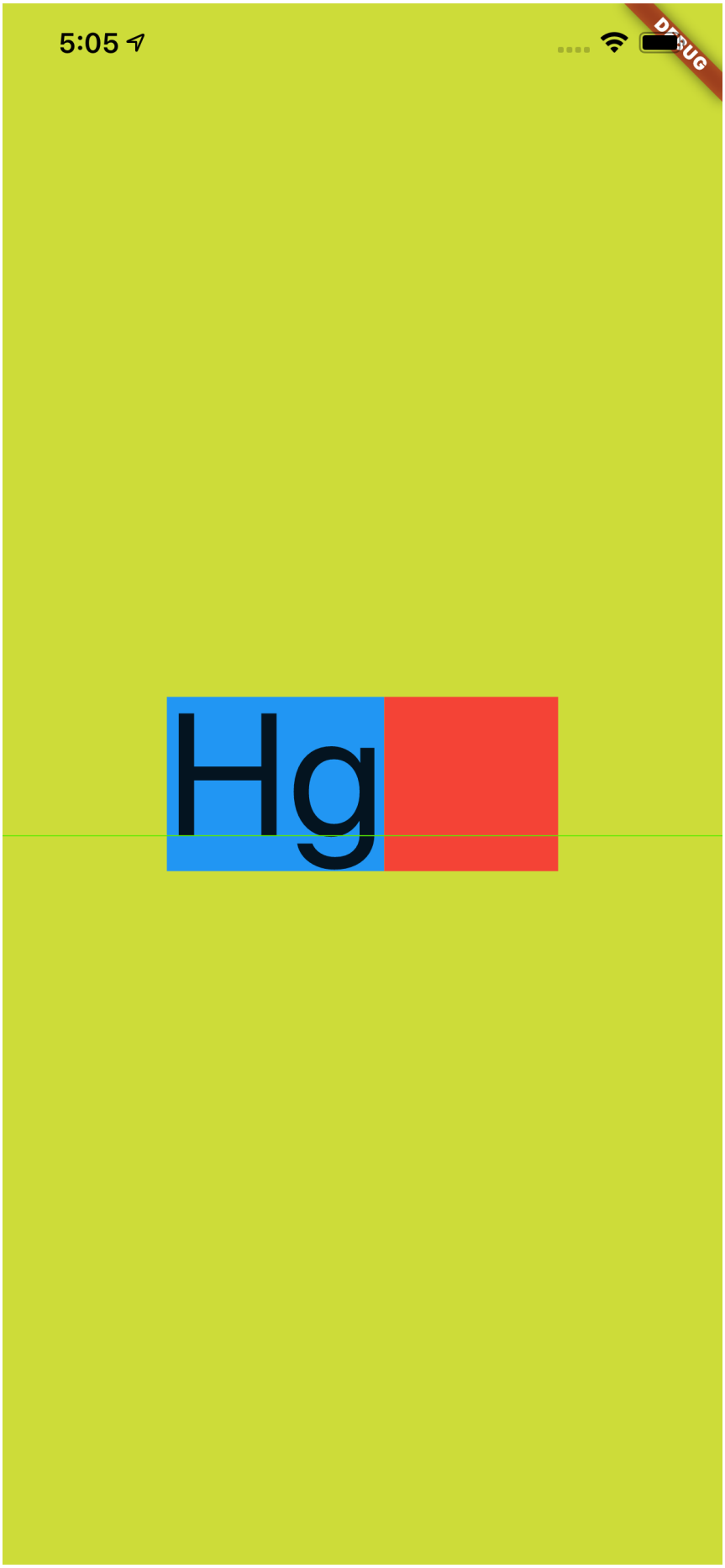
那再回顾下前面所说的默认字体的**量度**，这个默认字体的**量度**又是如何组成的呢？这就不得不说到 `StrutStyle` 。

如下代码所示，在之前的代码中添加 `StrutStyle` ：

- 设置了 `forceStrutHeight` 为 true ，这是因为只有 `forceStrutHeight` 才能强制重置 `Text` 的 `height` 属性；
- 设置了 `StrutStyle` 的 `height` 设置为 **`1`** ，这样 `TextStyle` 中的 `height` 等于 **`2`** 就没有了效果。

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.black,
    body: Container(
      color: Colors.lime,
      alignment: Alignment.center,
      child: Container(
        alignment: Alignment.center,
        child: Container(
          height: 200,
          color: Colors.purple,
          alignment: Alignment.center,
          child: new Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Container(
                color: Colors.blue,
                child: new Text(
                  "Hg",
                  style: TextStyle(
                    fontSize: 100,
                    height: 2,
                  ),
                  strutStyle: StrutStyle(
                    forceStrutHeight: true,
                    fontSize: 100,
                    height: 1
                  ),

                ),

              ),
              Container(
                height: 100,
                width: 100,
                color: Colors.red,
              )
            ],
          ),
        )

      ),
    ),
  );
}
```

效果如下图所示，虽然 `TextStyle` 的 `height` 是 **`2`** ,但是显示出现是以 `StrutStyle` 中 `height` 为 **`1`** 的效果为准。


5:05
DEBUG
Hg

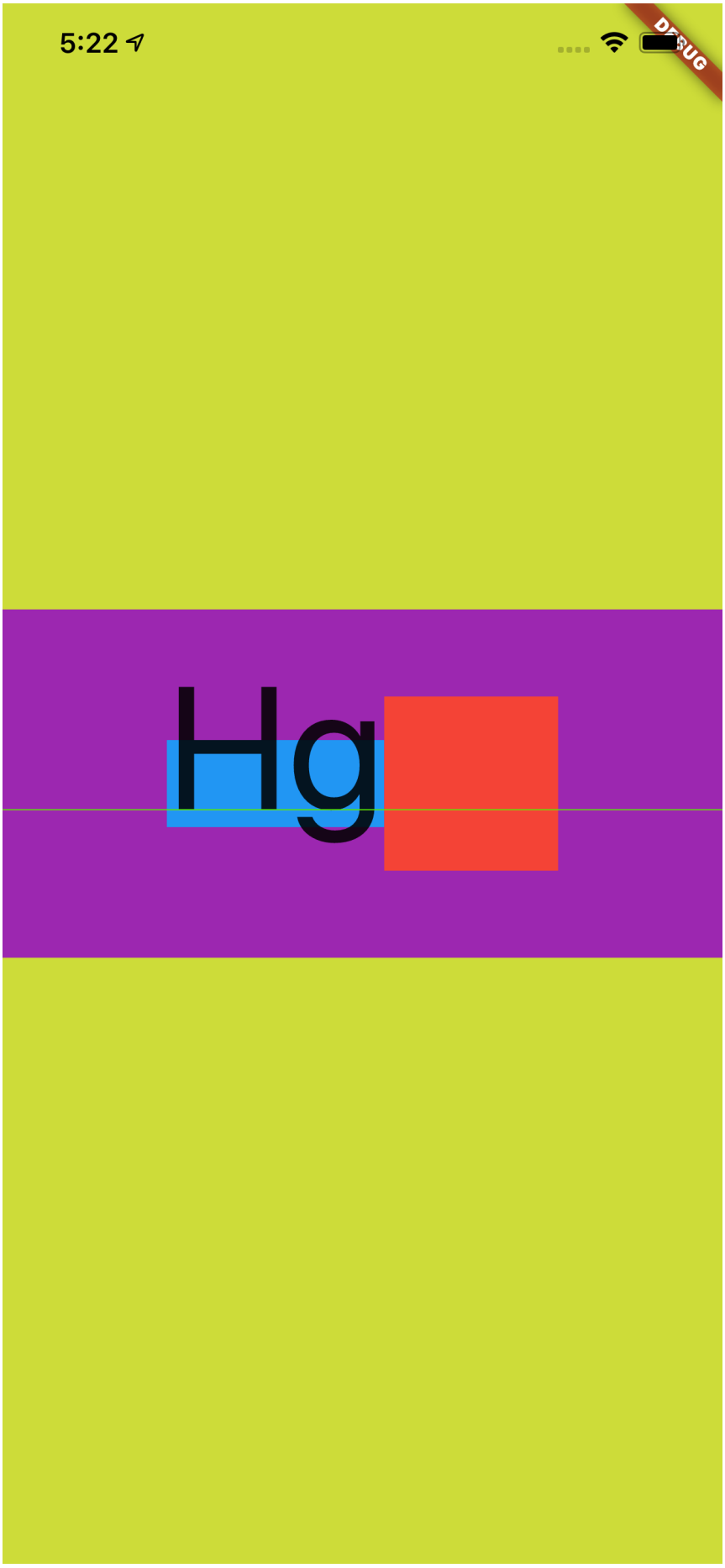
然后查看文档对于 `StrutStyle` 中 `height` 的描述，可以看到：`height` 的效果依然是 `fontSize` 的倍数，但是不同的是这里的对 `fontSize` 进行了补充说明： `ascent + descent = fontSize`，其中：

- `ascent` 代表的是基线上方部分；
- `descent` 代表的是基线的半部分

- 其组合效果如下图所示：

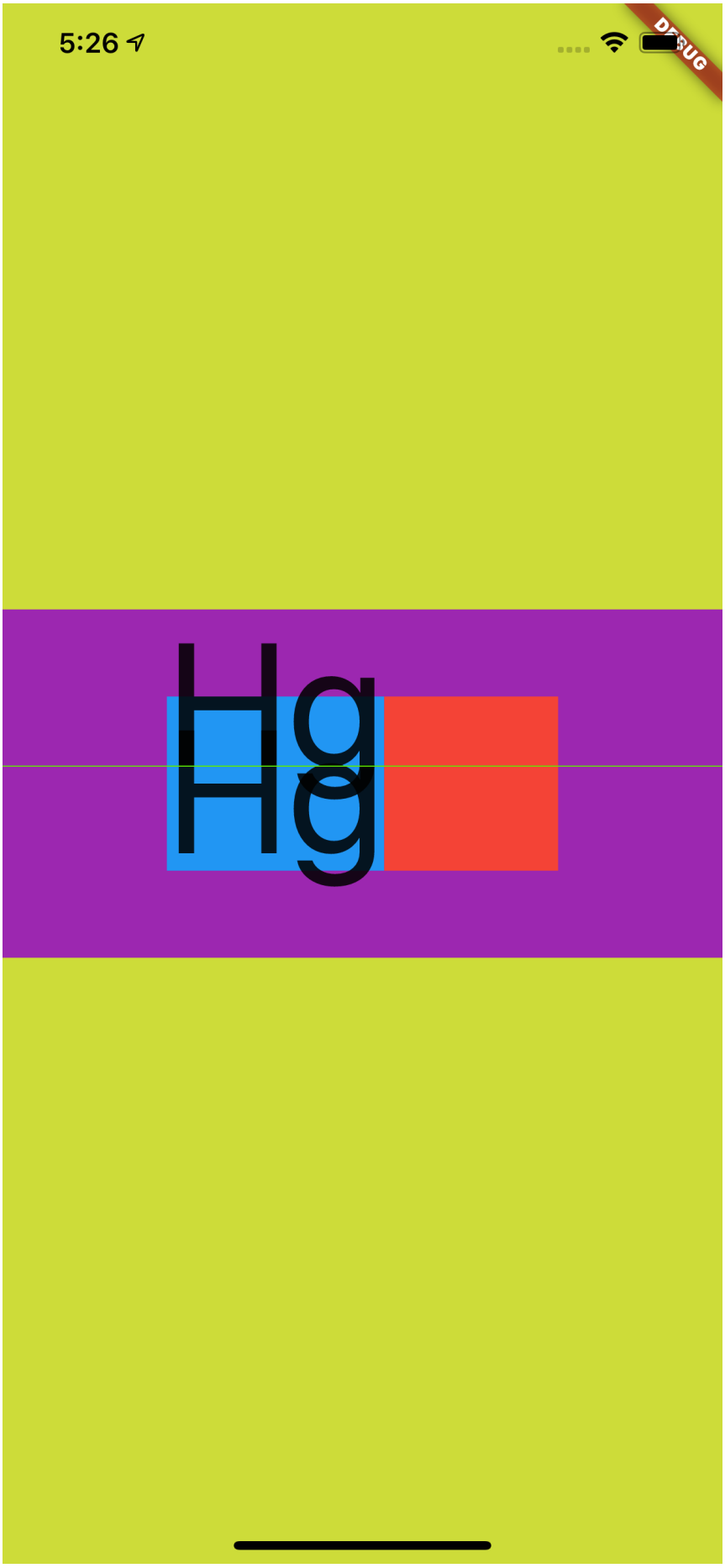
> Flutter 中 `ascent` 和 `descent` 是不能用代码单独设置。

除此之外，**`StrutStyle` 的 `fontSize` 和 `TextStyle` 的 `fontSize` 作用并不一样**：当我们把 `StrutStyle` 的 `fontSize` 设置为 **50**，而 `TextStyle` 的 `fontSize` 依然是 **100** 时，如下图所示，可以看到黑色的字体大小没有发生变化，而蓝色部分的大小变为了 **50** 的大小。


5:22
DEBUG
Hg

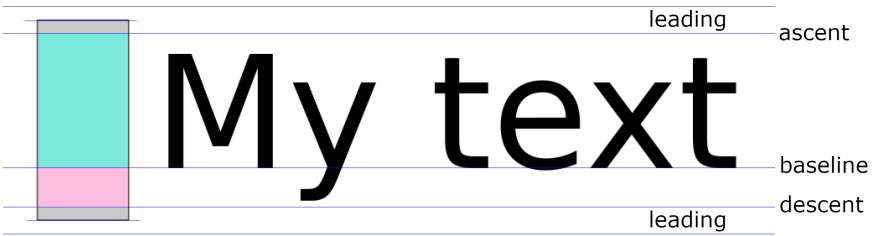
有人就要说那 `StrutStyle` 这样的 `fontSize` 有什么用?

这时候，如果在上面条件不变的情况下，把 `Text` 中的文本变成 `"Hg\nHg"` 这样的两行文本，可以看到换行后的文本重叠在了一起，**所以 `StrutStyle` 的 `fontSize` 也是会影响行高**。


5:26
DEBUG
Hg

另外，在 `StrutStyle` 中还有另外一个参数也会影响行高，那就是 `leading` 。

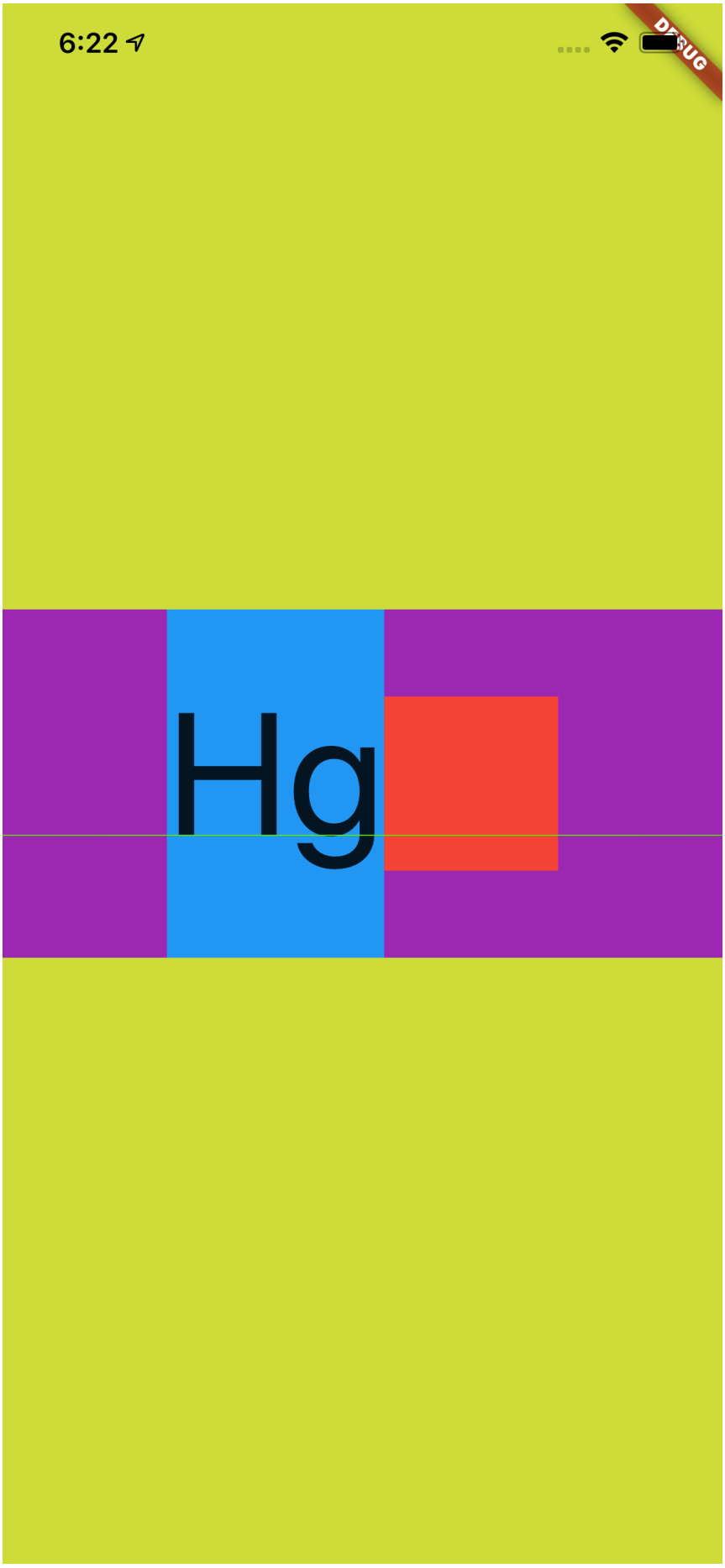
如下图所示，加上了 `leading` 后才是 Flutter 中对字体行高完全的控制组合， `leading` 默认为 `null` ，同时它的效果也是 `fontSize` 的倍数，并且分布是上下均分。



所以如下代码所示，当 `StrutStyle` 的 `fontSize` 为 **100** ， `height` 为 1， `leading` 为 1 时，可以看到 `leading` 的大小让蓝色区域变为了 **200**，从而 和紫色区域高度又重叠了，不同的对比之前的 `Hg` 在这次充满显示是居中。

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.black,
    body: Container(
      color: Colors.lime,
      alignment: Alignment.center,
      child: Container(
        alignment: Alignment.center,
        child: Container(
          height: 200,
          color: Colors.purple,
          alignment: Alignment.center,
          child: new Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Container(
                color: Colors.blue,
                child: new Text(
                  "Hg",
                  style: TextStyle(
                    fontSize: 100,
                    height: 2,
                  ),
                  strutStyle: StrutStyle(
                    forceStrutHeight: true,
                    fontSize: 100,
                    height: 1,
                    leading: 1
                  ),

                ),

              ),
              Container(
                height: 100,
                width: 100,
                color: Colors.red,
              )
            ],
          ),
        )

      ),
    ),
  );
}
```

> 因为 `leading` 是上下均分的，而 `height` 是根据 `ascent` 和 `descent` 的部分放大，明显 `ascent` 比 `descent` 大得多，所以前面的 `TextStyle` 的 `height` 为 2 时，充满后整体往下偏移。


6:22
DEBUG
Hg

三、backgroundColor

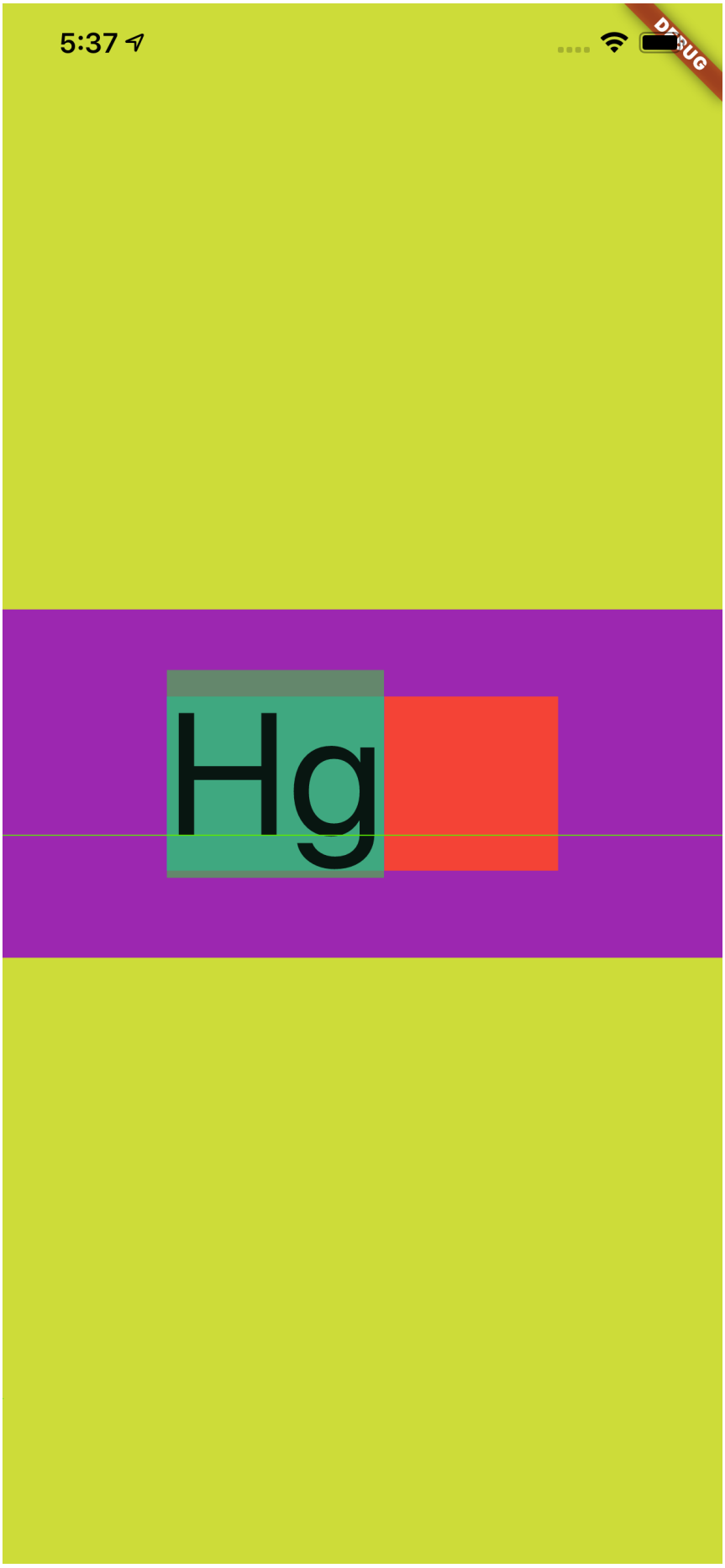
那么到这里应该对于 Flutter 中关于文本大小、度量和行高等有了基本的认知，接着再介绍一个属性： `TextStyle` 的 `backgroundColor` 。

> 介绍这个属性是为了和前面的内容产生一个对比，并且解除一些误解。

如下代码所示，可以看到 `StrutStyle` 的 `fontSize` 为 **100** ， `height` 为 **1** ，按照前面的介绍，蓝色的区域大小应该是和红色小方块一样大。

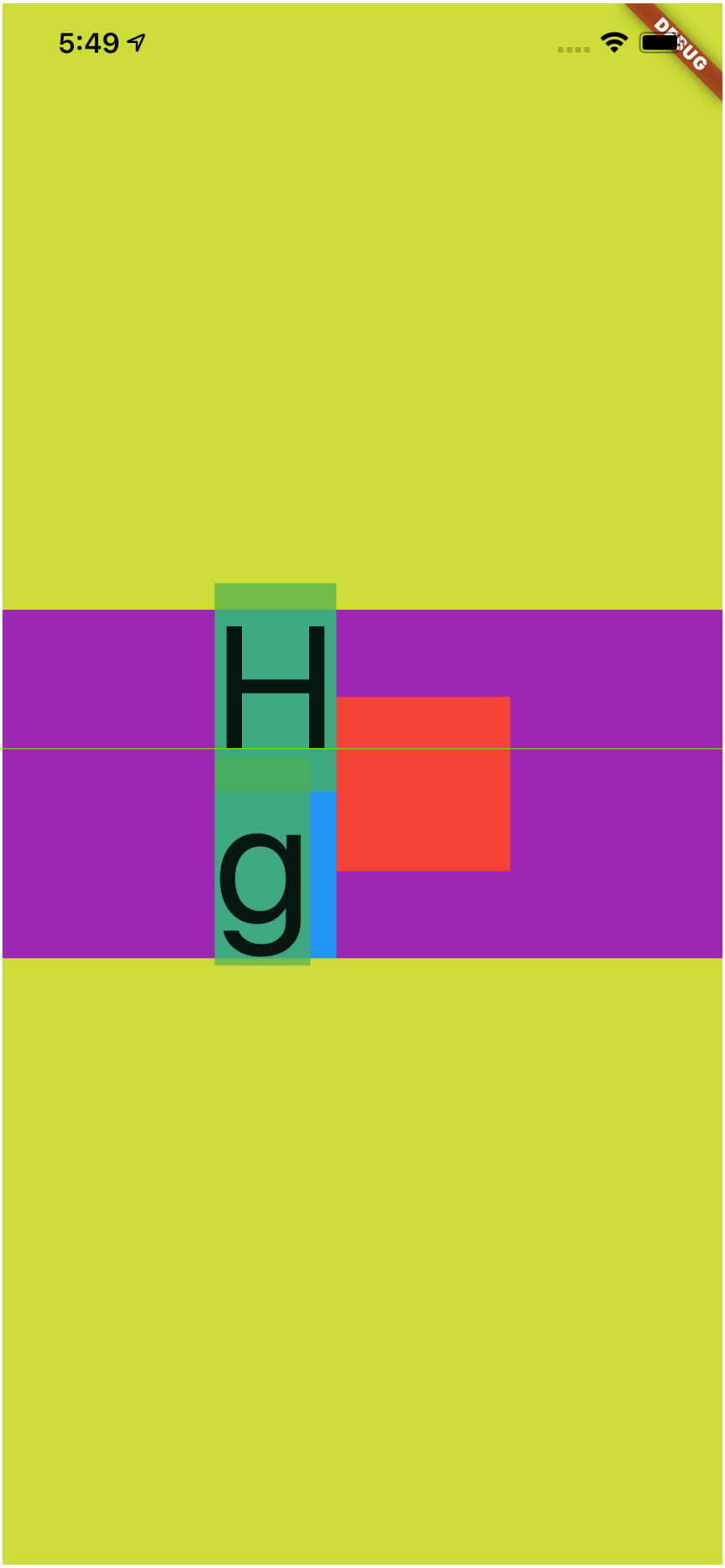
然后我们设置了 `TextStyle` 的 `backgroundColor` 为具有透明度的绿色，结果如下图所示，可以看到 `backgroundColor` 的区域超过了 `StrutStyle` ，显示为**默认情况下字体的度量**。

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.black,
    body: Container(
      color: Colors.lime,
      alignment: Alignment.center,
      child: Container(
        alignment: Alignment.center,
        child: Container(
          height: 200,
          color: Colors.purple,
          alignment: Alignment.center,
          child: new Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Container(
                color: Colors.blue,
                child: new Text(
                  "Hg",
                  style: TextStyle(
                    fontSize: 100,
                    backgroundColor: Colors.green.withAl
                  ),
                  strutStyle: StrutStyle(
                    forceStrutHeight: true,
                    fontSize: 100,
                    height: 1,
                  ),

                ),

              ),
              Container(
                height: 100,
                width: 100,
                color: Colors.red,
              )
            ],
          ),
        )

      ),
    ),
  );
}
```


5:37
DEBUG
Hg

这是不是很有意思，事实上也可以反应出，字体的度量其实一直都是默认的 `ascent + descent = fontSize`，我们可以改变 `TextStyle` 的 `height` 或者 `StrutStyle` 来改变行高效果，但是本质上的 `fontSize` 其实并没有变。

如果把输入内容换成 `"H\ng"`，如下图所示可以看到更有意思的效果。


5:49
DEBUG
H
g

四、TextBaseline

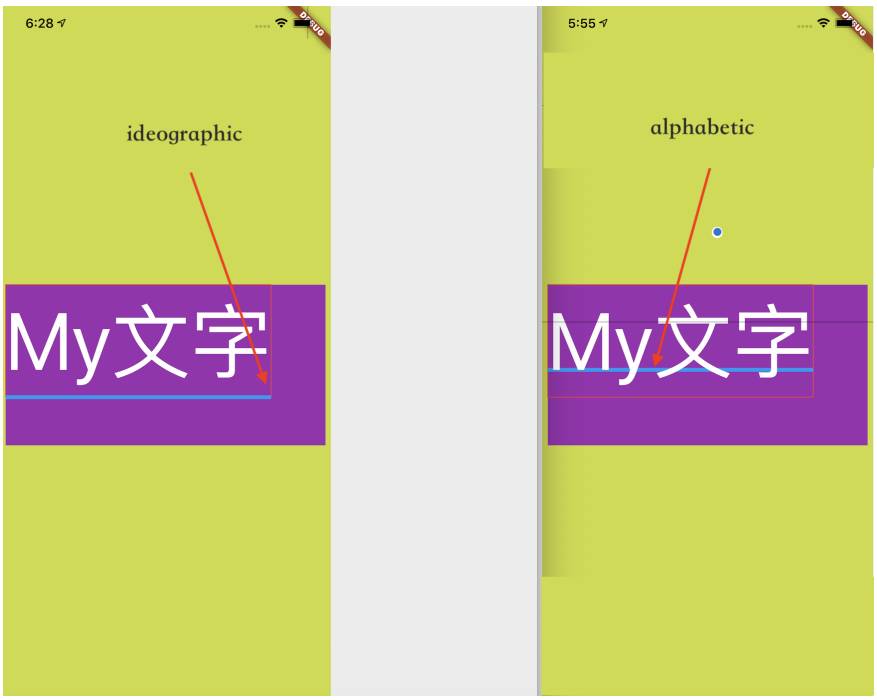
最后再介绍一个属性： `TextStyle` 的 `TextBaseline` ,因为这个属性一直让人产生“误解”。

关于 `TextBaseline` 有两个属性，分别是 `alphabetic` 和 `ideographic` ，为了更方便解释他们的效果，如下代码所示，我们通过 `CustomPaint` 把不同的基线位置绘制出来。

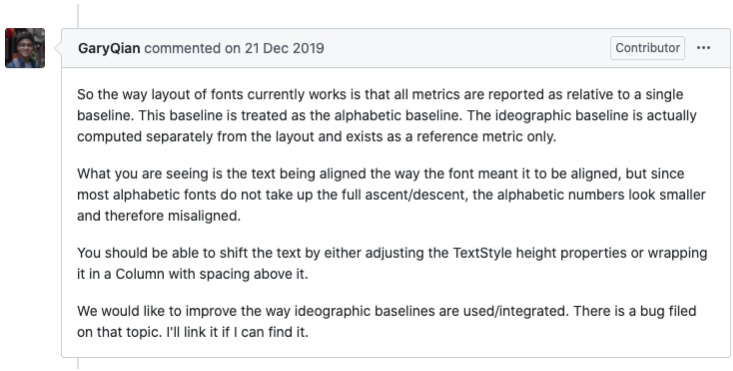
```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      body: Container(
        color: Colors.lime,
        alignment: Alignment.center,
        child: Container(
          alignment: Alignment.center,
          child: Container(
            height: 200,
            width: 400,
            color: Colors.purple,
            child: CustomPaint(
              painter: Text2Painter(),
            ),
          )

        ),
      ),
    );
  }

class Text2Painter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    var baseLine = TextBaseline.alphabetic;
    //var baseLine = TextBaseline.ideographic;

    final textStyle =
        TextStyle(color: Colors.white, fontSize: 100, textB
    final textSpan = TextSpan(
      text: 'My文字',
      style: textStyle,
    );
    final textPainter = TextPainter(
      text: textSpan,
      textDirection: TextDirection.ltr,
    );
    textPainter.layout(
      minWidth: 0,
      maxWidth: size.width,
    );

    final left = 0.0;
    final top = 0.0;
    final right = textPainter.width;
```

```
    final bottom = textPainter.height;
    final rect = Rect.fromLTRB(left, top, right, bottom);
    final paint = Paint()
      ..color = Colors.red
      ..style = PaintingStyle.stroke
      ..strokeWidth = 1;
    canvas.drawRect(rect, paint);

    // draw the baseline
    final distanceToBaseline =
        textPainter.computeDistanceToActualBaseline(baseLi

    canvas.drawLine(
      Offset(0, distanceToBaseline),
      Offset(textPainter.width, distanceToBaseline),
      paint..color = Colors.blue..strokeWidth = 5,
    );

    // draw the text
    final offset = Offset(0, 0);
    textPainter.paint(canvas, offset);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => true;
}
```

如下图所示，蓝色的线就是 baseLine，从效果可以直观看到不同 baseLine 下对齐的位置应该在哪里。



但是事实上 baseLine 的作用并不会直接影响 `TextStyle` 中文本的对齐方式，Flutter 中默认显示的文本只会通过 `TextBaseline.alphabetic` 对齐的，如下图所示官方人员也对这个问题有过描述 #47512。



GaryQian commented on 21 Dec 2019 Contributor

So the way layout of fonts currently works is that all metrics are reported as relative to a single baseline. This baseline is treated as the alphabetic baseline. The ideographic baseline is actually computed separately from the layout and exists as a reference metric only.

What you are seeing is the text being aligned the way the font meant it to be aligned, but since most alphabetic fonts do not take up the full ascent/descent, the alphabetic numbers look smaller and therefore misaligned.

You should be able to shift the text by either adjusting the TextStyle height properties or wrapping it in a Column with spacing above it.

We would like to improve the way ideographic baselines are used/integrated. There is a bug filed on that topic. I'll link it if I can find it.

> 这也是为什么要用 `CustomPaint` 展示的原因，因为用默认 `Text` 展示不出来。

举个典型的例子，如下代码所示，虽然在 `Row` 和 `Text` 上都是用了 `ideographic` ，但是其实并没有达到我们想要的效果。

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
     backgroundColor: Colors.black,
     body: Container(
       color: Colors.lime,
       alignment: Alignment.center,
       child: Container(
           alignment: Alignment.center,
           child: Row(
               crossAxisAlignment: CrossAxisAlignment.base
               textBaseline: TextBaseline.ideographic,
               mainAxisSize: MainAxisSize.max,
               children: [
                 Text(
                   '我是中文',
                   style: TextStyle(
                     fontSize: 55,
                     textBaseline: TextBaseline.ideographi
                   ),
                 ),
                 Spacer(),
                 Text('123y56',
                     style: TextStyle(
                       fontSize: 55,
                       textBaseline: TextBaseline.ideograp
                     )),
               ])),
     ),
   );
 }
```

> 关键就算 `Row` 设置了 `center` ，这段文本看起来还是不是特别“对齐”。

自从，关于 Flutter 中的字体相关的“冷”知识介绍完了，不知道你“无用”的知识有没有增多呢?


GSYTech
移动开发的后花园，变强不秃。
长按二维码关注公众号
@恋猫de小郭
忧愁时写一篇文章
快乐时编一段人生
无论工作如何艰辛
我们相信分享总是美好的开始
Flutter 完整开发实战详解
GSYTech
Flutter完整开发实战详解（十八、神奇的...
Flutter完整开发实战详解（十七、实用技巧与填坑二）
Flutter 完整开发实战详解（十六、详解自定义布局...
Flutter完整开发实战详解（十五、全面理解State与...
程序猿的故事
GSYTech
六年码农生涯的 2019 总结：君子坐而论道，少年...
2019大前端浪潮下的年度终结
一个程序员的五年总结，给你不一样的角度
程序员郭小喵创作故事
Android 巴士站
GSYTech
移动开发者的音视频基础知识
做Android开发的，想转行，如何选择呢?
Android程序员的Java后台学习建议
Flutter 杂谈与开发大会
GSYTech
Flutter 会不会被苹果限制其发展?
Flutter Interact 的 Flutter 1.12 大进化和回顾
Flutter 框架在国内移动端市场的发展趋势，在游戏...
Flutter 升级 1.12 适配教程

Flutter 1.17 对比上一个稳定版本，更多是带来了性能上的提升，其中一个关键的优化点就是 `Navigator` 的内部逻辑，本篇将带你解密 `Navigator` 从 1.12 到 1.17 的变化，并介绍 Flutter 1.17 上究竟优化了哪些性能。

一、Navigator 优化了什么?

在 1.17 版本最让人感兴趣的变动莫过于：**“打开新的不透明页面之后，路由里的旧页面不会再触发 `build` ”**。

虽然之前介绍过 `build` 方法本身很轻，但是在“不需要”的时候“不执行”明显更符合我们的预期，而这个优化的 PR 主要体现在 `stack.dart` 和 `overlay.dart` 两个文件上。

- `stack.dart` 文件的修改，只是为了将 `RenderStack` 的相关逻辑变为共享的静态方法 `getIntrinsicDimension` 和 `layoutPositionedChild` ，其实就是共享 `Stack` 的部分布局能力给 `Overlay` 。
- `overlay.dart` 文件的修改则是这次的灵魂所在。

二、Navigator 的 Overlay

事实上我们常用的 `Navigator` 是一个 `StatefulWidget` ， 而常用的 `pop` 、 `push` 等方法对应的逻辑都是在 `NavigatorState` 中，而 **`NavigatorState` 主要是通过 `Overlay` 来承载路由页面，所以导航页面间的管理逻辑主要在于 `Overlay`** 。

2.1、Overlay 是什么?

`Overlay` 大家可能用过，在 Flutter 中可以通过 `Overlay` 来向 `MaterialApp` 添加全局悬浮控件，这是因为 `Overlay` 是一个类似 `Stack` 层级控件，但是它可以通过 `OverlayEntry` 来独立地管理内部控件的展示。

比如可以通过 `overlayState.insert` 插入一个 `OverlayEntry` 来实现插入一个图层，而 `OverlayEntry` 的 `builder` 方法会在展示时被调用，从而出现需要的布局效果。

```
    var overlayState = Overlay.of(context);
    var _overlayEntry = new OverlayEntry(builder: (context)
      return new Material(
        color: Colors.transparent,
        child: Container(
          child: Text(
            "${widget.platform} ${widget.deviceInfo} ${widg
            style: TextStyle(color: Colors.white, fontSize:
          ),
        ),
      );
    });
    overlayState.insert(_overlayEntry);
```

2.2、Overlay 如何实现导航?

在 `Navigator` 中其实也是使用了 `Overlay` 实现页面管理，**每个打开的 `Route` 默认情况下是向 `Overlay` 插入了两个 `OverlayEntry`** 。

> 为什么是两个后面会介绍。

而在 `Overlay` 中， `List<OverlayEntry> _entries` 的展示逻辑又是通过 `_Theatre` 来完成的，在 `_Theatre` 中有 `onstage` 和 `offstage` 两个参数，其中：

- `onstage` 是一个 `Stack` ，用于展示 `onstageChildren.reversed.toList(growable: false)` ，也就是可以被看到的部分；
- `offstage` 是展示 `offstageChildren` 列表，也就是不可以被看到的部分；

```
    return _Theatre(
      onstage: Stack(
        fit: StackFit.expand,
        children: onstageChildren.reversed.toList(growable:
      ),
      offstage: offstageChildren,
    );
```

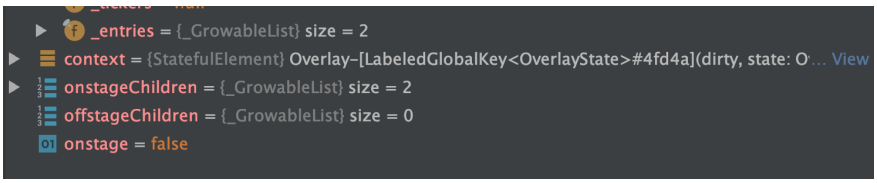
简单些说，比如此时有 [A、B、C] 三个页面，那么：

- C 应该是在 `onstage` ；
- A、B 应该是处于 `offstage` 。

当然，A、B、C 都是以 `OverlayEntry` 的方式被插入到 `Overlay` 中，而 A 、B、C 页面被插入的时候默认都是两个 `OverlayEntry` ，也就是 [A、B、C] 应该有 6 个 `OverlayEntry` 。

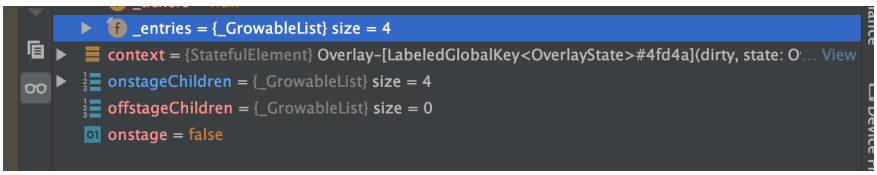
举个例子，程序在默认启动之后，首先看到的就是 A 页面，这时候可以看到 `Overlay` 中

- `_entries` 长度是 2，即 `Overlay` 中的列表总长度为2；
- `onstageChildren` 长度是 2，即当前可见的 `OverlayEntry` 是 2；
- `offstageChildren` 长度是 0，即没有不可见的 `OverlayEntry` ；

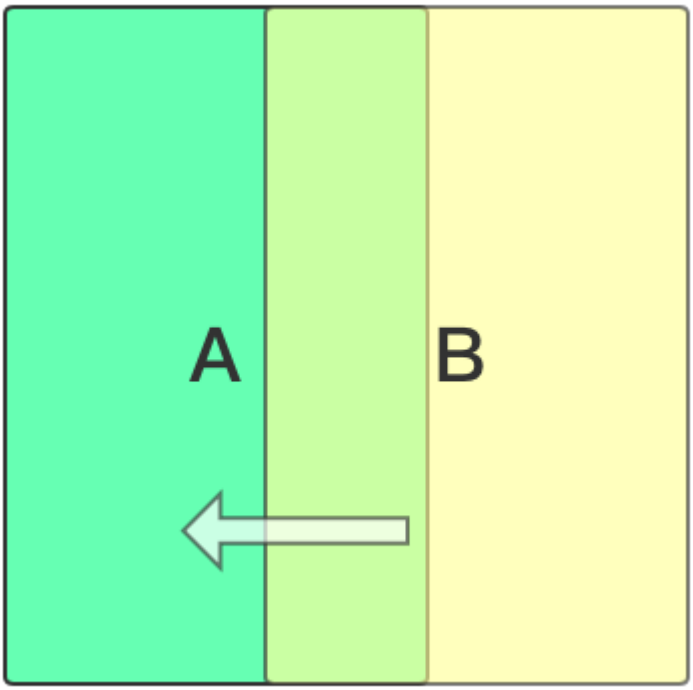


这时候我们打开 B 页面，可以看到 `Overlay` 中：

- `_entries` 长度是 4，也就是 `Overlay` 中多插入了两个 `OverlayEntry` ；
- `onstageChildren` 长度是 4，就是当前可见的 `OverlayEntry` 是 4 个；
- `offstageChildren` 长度是 0，就是当前还没有不可见的 `OverlayEntry` 。

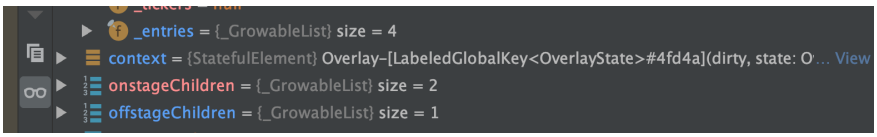


其实这时候 `Overlay` 处于页面打开中的状态，也就是 A 页面还可以被看到，B 页面正在动画打开的过程。



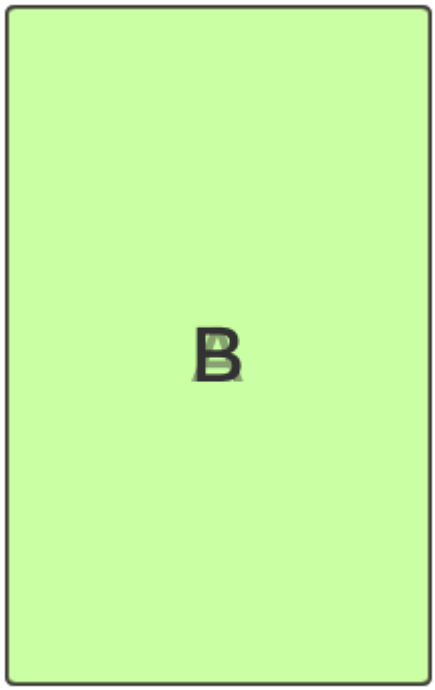
接着可以看到 `Overlay` 中的 `build` 又再次被执行：

- `_entries` 长度还是 4；
- `onstageChildren` 长度变为 2，即当前可见的 `OverlayEntry` 变成了 2 个；
- `offstageChildren` 长度是 1，即当前有了一个不可见 `OverlayEntry` 。



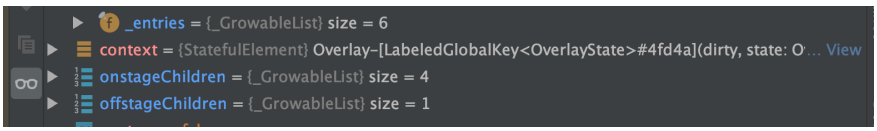
这时候 B 页面其实已经打开完毕，所以 `onstageChildren` 恢复为 2 的长度，也就是 B 页面对应的那两个 `OverlayEntry` ；而 A 页面不可见，所以 A 页面被放置到了 `offstageChildren` 。

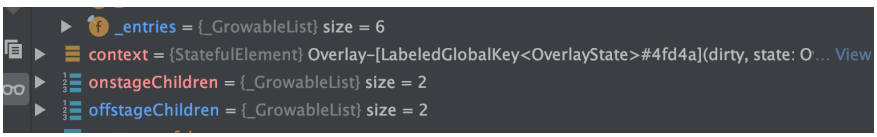
> 为什么只把 A 的一个 `OverlayEntry` 放到 `offstageChildren` ？这个后面会讲到。



接着如下图所示，再打开 C 页面时，可以看到同样经历了这个过程：

- `_entries` 长度变为 6；
- `onstageChildren` 长度先是 4 ，之后又变成 2 ，因为打开时有B 和 C 两个页面参与，而打开完成后只剩下一个 C 页面；
- `offstageChildren` 长度是 1，之后又变为 2，因为最开始只有 A 不可见，而最后 A 和 B 都不可见；





所以可以看到，每次打开一个页面：

- 先会向 `_entries` 插入两个 `OverlayEntry` ；
- 之后会先经历 `onstageChildren` 长度是 4 的页面打开过程状态；
- 最后变为 `onstageChildren` 长度是 2 的页面打开完成状态，而底部的页面由于不可见所以被加入到 `offstageChildren` 中；

2.3、Overlay 和 Route

为什么每次向 `_entries` 插入的是两个 `OverlayEntry` ？

这就和 `Route` 有关，比如默认 `Navigator` 打开新的页面需要使用 `MaterialPageRoute` ，而生成 `OverlayEntry` 就是在它的基类之一的 `ModalRoute` 完成。

在 `ModalRoute` 的 `createOverlayEntries` 方法中，通过 `_buildModalBarrier` 和 `_buildModalScope` 创建了两个 `OverlayEntry` ，其中：

- `_buildModalBarrier` 创建的一般是蒙层；
- `_buildModalScope` 创建的 `OverlayEntry` 是页面的载体；

所以默认打开一个页面，是会存在两个 `OverlayEntry` ，一个是蒙层一个是页面。

```
@override
Iterable<OverlayEntry> createOverlayEntries() sync* {
  yield _modalBarrier = OverlayEntry(builder: _buildModal
  yield OverlayEntry(builder: _buildModalScope, maintainS
}
```

那么一个页面有两个 `OverlayEntry` ，但是为什么插入到 `offstageChildren` 中的数量每次都是加1 而不是加2？

如果单从逻辑上讲，按照前面 [A、B、C] 三个页面的例子， `_entries` 里有 6 个 `OverlayEntry` ， 但是 B、C 页面都不可见了，把 B、C 页面的蒙层也捎带上不就纯属浪费了？

如从代码层面解释，在 `_entries` 在倒序 `for` 循环的时候：

- 在遇到 `entry.opaque` 为 `ture` 时，后续的 `OverlayEntry` 就进不去 `onstageChildren` 中；
- `offstageChildren` 中只有 `entry.maintainState` 为 `true` 才会被添加到队列；

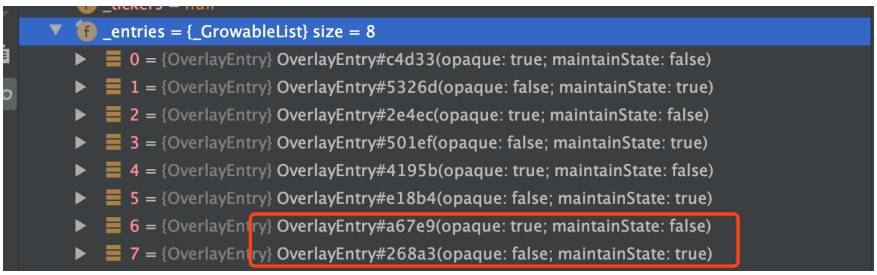
```
@override
Widget build(BuildContext context) {
  final List<Widget> onstageChildren = <Widget>[];
  final List<Widget> offstageChildren = <Widget>[];
  bool onstage = true;
  for (int i = _entries.length - 1; i >= 0; i -= 1) {
    final OverlayEntry entry = _entries[i];
    if (onstage) {
      onstageChildren.add(_OverlayEntry(entry));
      if (entry.opaque)
        onstage = false;
    } else if (entry.maintainState) {
      offstageChildren.add(TickerMode(enabled: false, ch
    }
  }
  return _Theatre(
    onstage: Stack(
      fit: StackFit.expand,
      children: onstageChildren.reversed.toList(growable
    ),
    offstage: offstageChildren,
  );
}
```

而在 `OverlayEntry` 中：

- `opaque` 表示了 `OverlayEntry` 是不是“阻塞”了整个 `Overlay`，也就是不透明的完全覆盖。
- `maintainState` 表示这个 `OverlayEntry` 必须被添加到 `_Theatre` 中。

所以可以看到，当页面完全打开之后，在最前面的两个 `OverlayEntry`：

- 蒙层 `OverlayEntry` 的 `opaque` 会被设置为 true，这样后面的 `OverlayEntry` 就不会进入到 `onstageChildren`，也就是不显示；
- 页面 `OverlayEntry` 的 `maintainState` 会是 `true`，这样不可见的时候也会进入到 `offstageChildren` 里；



那么 `opaque` 是在哪里被设置的?

关于 `opaque` 的设置过程如下所示，在 `MaterialPageRoute` 的另一个基类 `TransitionRoute` 中，可以看到一开始蒙层的 `opaque` 会被设置为 `false` ，之后在 `completed` 会被设置为 `opaque` ，而 `opaque` 参数在 `PageRoute` 里就是 `@override bool get opaque => true;`

> 在 `PopupRoute` 中 `opaque` 就是 `false` ，因为 `PopupRoute` 一般是有透明的背景，需要和上一个页面一起混合展示。

```
void _handleStatusChanged(AnimationStatus status) {
   switch (status) {
     case AnimationStatus.completed:
       if (overlayEntries.isNotEmpty)
         overlayEntries.first.opaque = opaque;
       break;
     case AnimationStatus.forward:
     case AnimationStatus.reverse:
       if (overlayEntries.isNotEmpty)
         overlayEntries.first.opaque = false;
       break;
     case AnimationStatus.dismissed:
       if (!isActive) {
         navigator.finalizeRoute(this);
         assert(overlayEntries.isEmpty);
       }
       break;
   }
   changedInternalState();
 }
```

到这里我们就理清了页面打开时 `Overlay` 的工作逻辑，默认情况下：

- 每个页面打开时会插入两个 `OverlayEntry` 到 `Overlay` ；
- 打开过程中 `onstageChildren` 是 4 个，因为此时两个页面在混合显示；
- 打开完成后 `onstageChildren` 是 2，因为蒙层的 `opaque` 被设置为 `ture` ，后面的页面不再是可见；
- 具备 `maintainState` 为 `true` 的 `OverlayEntry` 在不可见后会进入到 `offstageChildren` ；

> 额外介绍下，路由被插入的位置会和 `route.install` 时传入的 `OverlayEntry` 有关，比如： `push` 传入的是 `_history` (页面路由堆栈)的 last 。

三、新版 1.17 中 Overlay

那为什么在1.17 之前，打开新的页面时旧的页面会被执行 `build` ？ 这里面其实主要有两个点：

- `OverlayEntry` 都有一个 `GlobalKey<_OverlayEntryState>` 用户表示页面的唯一；
- `OverlayEntry` 在 `_Theatre` 中会有从 `onstage` 到 `offstage` 的过程；

3.1、为什么会 rebuild

因为 `OverlayEntry` 在 `Overlay` 内部是会被转化为 `_OverlayEntry` 进行工作，而 `OverlayEntry` 里面的 `GlobalKey` 自然也就用在了 `_OverlayEntry` 上，而当 `Widget` 使用了 `GlobalKey` ，那么其对应的 `Element` 就会是 "Global" 的。

在 `Element` 执行 `inflateWidget` 方法时，会判断如果 `Key` 值是 `GlobalKey` ，就会调用 `_retakeInactiveElement` 方法返回“已存在”的 `Element` 对象，从而让 `Element` 被“复用”到其它位置，而这个过程 `Element` 会从原本的 `parent` 那里被移除，然后添加到新的 `parent` 上。

这个过程就会触发 `Element` 的 `update` ，而 `_OverlayEntry` 本身是一个 `StatefulWidget` ，所以对应的 `StatefulElement` 的 `update` 就会触发 `rebuild` 。

3.2、为什么 1.17 不会 rebuild

那在 1.17 上，为了不出现每次打开页面后还 `rebuild` 旧页面的情况，这里取消了 `_Theatre` 的 `onstage` 和 `offstage` ，替换为 `skipCount` 和 `children` 参数。

并且 `_Theatre` 从 `RenderObjectWidget` 变为了 `MultiChildRenderObjectWidget` ，然后在 `_RenderTheatre` 中复用了 `RenderStack` 共享的布局能力。

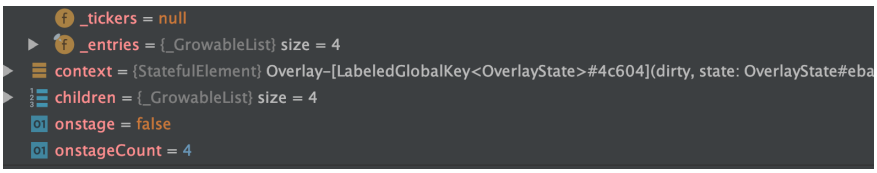
```
@override
Widget build(BuildContext context) {
  // This list is filled backwards and then reversed belo
  // it is added to the tree.
  final List<Widget> children = <Widget>[];
  bool onstage = true;
  int onstageCount = 0;
  for (int i = _entries.length - 1; i >= 0; i -= 1) {
    final OverlayEntry entry = _entries[i];
    if (onstage) {
      onstageCount += 1;
      children.add(_OverlayEntryWidget(
        key: entry._key,
        entry: entry,
      ));
      if (entry.opaque)
        onstage = false;
    } else if (entry.maintainState) {
      children.add(_OverlayEntryWidget(
        key: entry._key,
        entry: entry,
        tickerEnabled: false,
      ));
    }
  }
  return _Theatre(
    skipCount: children.length - onstageCount,
    children: children.reversed.toList(growable: false),
  );
}
```

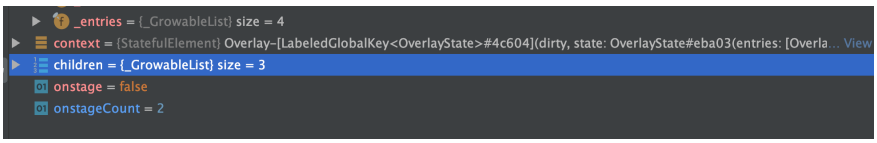
这时候等于 `Overlay` 中所有的 `_entries` 都处理到一个 `MultiChildRenderObjectWidget` 中，也就是同在一个 `Element` 中，而不是之前控件需要在 `onstage` 的 `Stack` 和 `offstage` 列表下来回切换。

在新的 `_Theatre` 将两个数组合并成一个 `children` 数组，然后将 `onstageCount` 之外的部分设置为 `skipCount` ，在布局时获取 `_firstOnstageChild` 进行布局，而当 `children` 发生改变时，触发的是 `MultiChildRenderObjectElement` 的 `insertChildRenderObject` ，而不会去“干扰”到之前的页面，所以不会产生上一个页面的 `rebuild` 。

```
RenderBox get _firstOnstageChild {
  if (skipCount == super.childCount) {
    return null;
  }
  RenderBox child = super.firstChild;
  for (int toSkip = skipCount; toSkip > 0; toSkip--) {
    final StackParentData childParentData = child.parentD
    child = childParentData.nextSibling;
    assert(child != null);
  }
  return child;
}

RenderBox get _lastOnstageChild => skipCount == super.chi
```

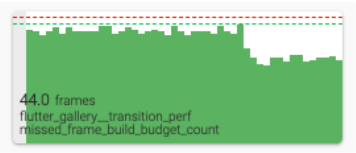
最后如下图所示，在打开页面后，`children` 会经历从 4 到 3 的变化，而 `onstageCount` 也会从 4 变为 2，也印证了页面打开过程和关闭之后的逻辑其实并没发生本质的变化。



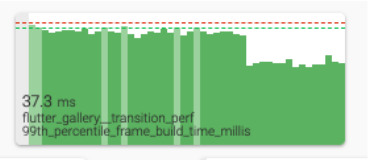


从结果上看，这个改动确实对性能产生了不错的提升。当然，这个改进主要是在不透明的页面之间生效，如果是透明的页面效果比如 `PopModal` 之类的，那还是需要 `rebuild` 一下。

This change positively affected some of our Gallery transition benchmarks:



- dropped from 63 frames to 50 (minus 21%)



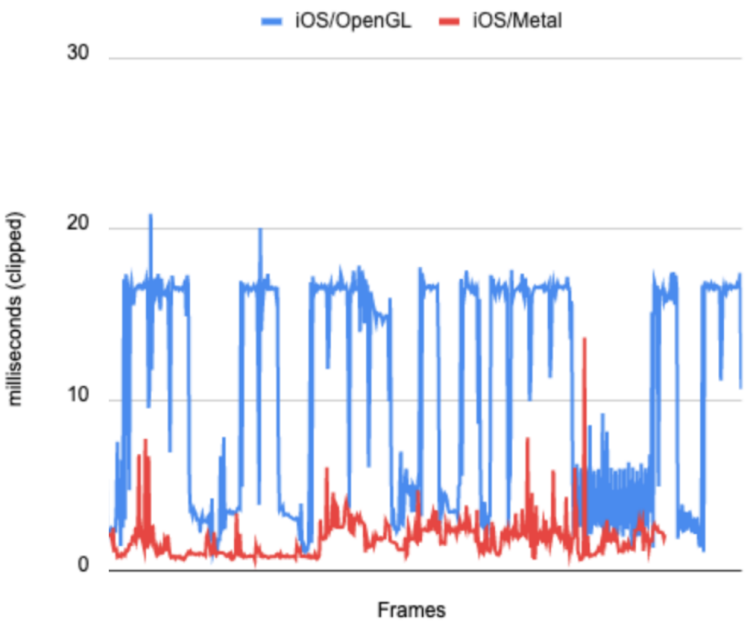
- dropped from 50.45ms to 36.64ms (minus 27%)

- dropped from 7.38ms to 6.13ms (minus 17%)

四、其他优化

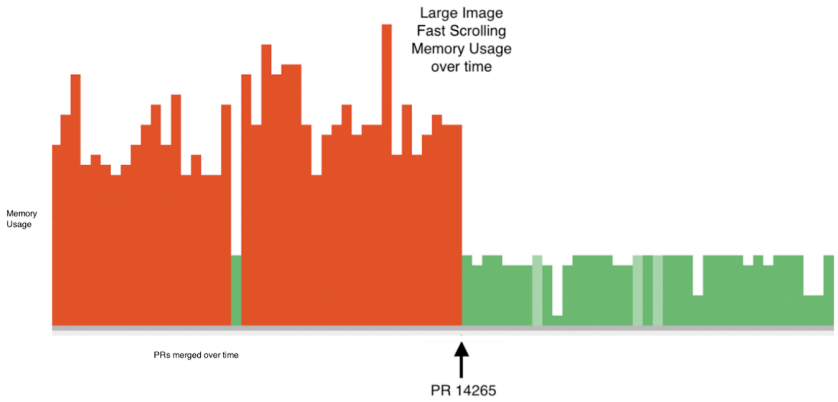
`Metal` 是 iOS 上类似于 `OpenGL ES` 的底层图形编程接口，可以在 iOS 设备上通过 api 直接操作 GPU 。

而 1.17 开始，Flutter 在 iOS 上对于支持 `Metal` 的设备将使用 `Metal` 进行渲染，所以官方提供的数据上看，这样可以提高 50% 的性能。更多可见：https://github.com/flutter/flutter/wiki/Metal-on-iOS-FAQ



Android 上也由于 Dart VM 的优化，体积可以下降大约 18.5% 的大小。

1.17对于加载大量图片的处理进行了优化，在快速滑动的过程中可以得到更好的性能提升（通过延时清理 IO Thread 的 Context），这样理论上可以在原本基础上节省出 70% 的内存。



好了，这一期想聊的聊完了，最后容我“厚颜无耻”地推广下鄙人最近刚刚上架的新书 **《Flutter 开发实战详解》，感兴趣的小伙伴可以通过以下地址了解：**

- **京东：https://item.jd.com/12883054.html**
- **当当：http://product.dangdang.com/28558519.html**

相信 Flutter 的开发者应该遇到过，对于大量数据的列表进行图片加载时，在 iOS 上很容易出现 `OOM` 的问题，这是因为 Flutter 特殊的图片加载流程造成。

> 在 Android 上 Flutter `Image` 主要占用的内存不是 `JVM` 的内存，而是 `Graphics` 相关的内存，这样的内存调用可以最大程度利用 Native 内存。

一、默认流程

Flutter 默认在进行图片加载时，会先通过对应的 `ImageProvider` 去加载图片数据，然后通过 `PaintingBinding` 对数据进行编码，之后返回包含编码后图片数据和信息的 `ImageInfo` 去实现绘制。

> 详细图片加载流程可见：《十、 深入图片加载流程)》

本身这个逻辑并没有什么问题，**问题就在于 Flutter 中对于图片在内存中的 Cache 对象是一个 `ImageStream` 对象**。

Flutter 中 `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**。

所以一开始最粗暴的情况是：通过 `PaintingBinding.instance` 去设置 `maximumSize` 和 `maximumSizeBytes`，但是这种简单粗爆的处理方法并不能解决长列表图片加载的溢出问题，因为在长列表中，快速滑动的情况下可能会在一瞬间“并发”出大量图片加载需求。

所以在 1.17 版本上，官方针对这种情况提供了场景化的处理方式：`ScrollAwareImageProvider`。

二、ScrollAwareImageProvider

1.17 中可以看到，在 `Image` 控件中原本 `_resolveImage` 方法所使用的 `imageProvider` 被 `ScrollAwareImageProvider` 所代理，并且多了一个叫 `DisposableBuildContext<State<Image>>` 的 context 参数。那 `ScrollAwareImageProvider` 的作用是什么呢?

```
void _resolveImage() {
  final ScrollAwareImageProvider provider = ScrollAwareIm
    context: _scrollAwareContext,
    imageProvider: widget.image,
  );
  final ImageStream newStream =
    provider.resolve(createLocalImageConfiguration(
      context,
      size: widget.width != null && widget.height != nul
    ));
  assert(newStream != null);
  _updateSourceStream(newStream);
}
```

其实 `ScrollAwareImageProvider` 对象最主要的使用就是在 `resolveStreamForKey` 方法中，通过 `Scrollable.recommendDeferredLoadingForContext` 方法去判断当前是不是需要推迟当前帧画面的加载，换言之就是：**是否处于快速滑动的过程**。

那 `Scrollable.recommendDeferredLoadingForContext` 作为一个 `static` 方法，如何判断当前是不是处于列表的快速滑动呢?

这就需要通过当前 `context` 的 `getElementForInheritedWidgetOfExactType` 方法去获取 `Scrollable` 内的 `_ScrollableScope` 。

> `_ScrollableScope` 是 `Scrollable` 内的一个 `InheritedWidget` ，而 Flutter 中的可滑动视图内必然会有 `Scrollable` ，所以只要 `Image` 是在列表内，就可以通过 `context.getElementForInheritedWidgetOfExactType<_ScrollableScope>()` 去获取到 `_ScrollableScope` 。

获取到 `_ScrollableScope` 就可以获取到它内部的 `ScrollPosition` ， 进而它的 `ScrollPhysics` 对应的 `recommendDeferredLoading` 方法，判断列表是否处于快速滑动状态。所以判断是否快速滑动的逻辑其实是在 `ScrollPhysics` 。

```
bool recommendDeferredLoading(double velocity, ScrollMetr
  assert(velocity != null);
  assert(metrics != null);
  assert(context != null);
  if (parent == null) {
    final double maxPhysicalPixels = WidgetsBinding.insta
    return velocity.abs() > maxPhysicalPixels;
  }
  return parent.recommendDeferredLoading(velocity, metric
}
```

> 关于 `ScrollPhysics` 的解释可以看 《十八、 神奇的ScrollPhysics与Simulation》

然后回到 `resolveStreamForKey` 方法，可以看到当 `Scrollable.recommendDeferredLoadingForContext` 返回 `true` 时就等待，等待就是会通过 `SchedulerBinding` 在下一帧绘制时再次调用 `resolveStreamForKey`， 递归再走一遍 `resolveStreamForKey` 的逻辑，如果判断此时不再是快速滑动，就走正常的图片加载逻辑。

```
@override
  void resolveStreamForKey(
    ImageConfiguration configuration,
    ImageStream stream,
    T key,
    ImageErrorListener handleError,
  ) {
    if (stream.completer != null ||  PaintingBinding.instanc
      imageProvider.resolveStreamForKey(configuration, stre
      return;
    }
    if (context.context == null) {
      return;
    }
    if (Scrollable.recommendDeferredLoadingForContext(conte
        SchedulerBinding.instance.scheduleFrameCallback((_)
          scheduleMicrotask(() => resolveStreamForKey(confi
        });
        return;
    }
    imageProvider.resolveStreamForKey(configuration, strear
  }
```

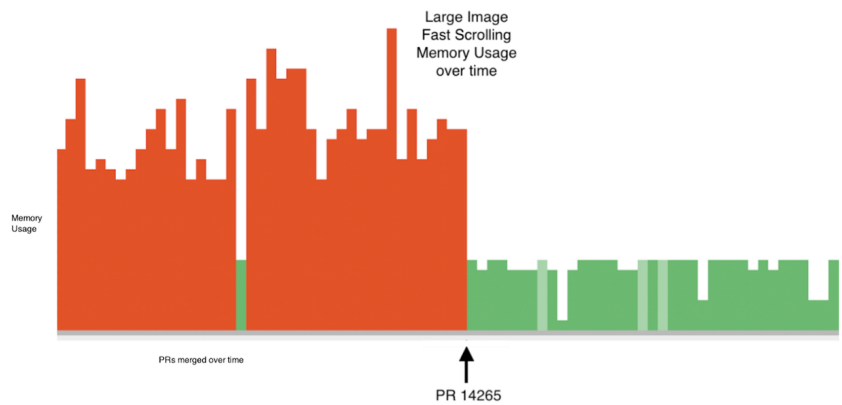
如上代码所示，可以看到在 `ScrollAwareImageProvider` 的 `resolveStreamForKey` 方法中，当 `stream.completer != null` 且存在缓存时，直接就去加载原本已有的流程，如果快速滑动过程中图片还没加载的，就先不加载。

> Flutter 中为了防止 `context` 在图片异步加载流程中持有导致内存泄漏，又针对 `Image` 封装了一个 `DisposableBuildContext` 。
>
> `DisposableBuildContext` 是通过持有 `State` 来持有 `context` 的，并且在 `dispose` 时将 `_state = null` 设置为 `null` 来清除对 `State` 的持有。所以可以看到 上述代码中， `context.context == null` 时直接就 `return` 了。

另外前面介绍的 `resolveStreamForKey` 也是新增加的方法，在原本的 `ImageProvider` 进行图片加载时，会通过 `ImageStream resolve` 方法去得到并返回一个 `ImageStream` 。

而 `resolveStreamForKey` 将原本 `imageCache` 和 `ImageStreamCompleter` 的流程抽象出来，并且在 `ScrollAwareImageProvider` 中重写了 `resolveStreamForKey` 方法的执行逻辑，这样快速滑动时，图片的下载和解码可以被中断，从而减少了不必要的内存占用。

虽然这种方法不能100%解决图片加载时 OOM 的问题，但是很大程度优化了列表中的图片内存占用，官方提供的数据上看理论上可以在原本基础上节省出 70% 的内存。



> 相关推荐：Merged Defer image decoding when scrolling fast #49389

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

在以前的《Android PlatformView 和键盘问题》一文中介绍过混合开发上 Android `PlatformView` 的实现和问题，原本 Android 平台上为了集成如 `WebView` 、 `MapView` 等能力，使用了 `VirtualDisplays` 的实现方式。

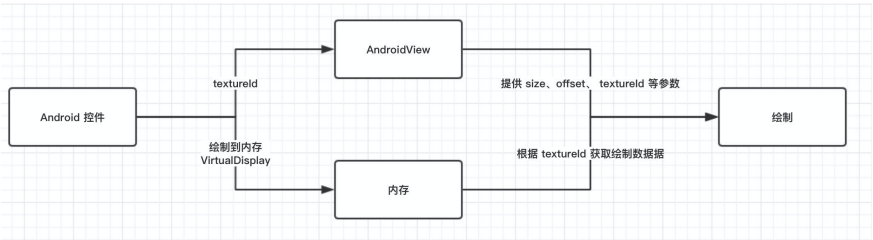
如今 1.20 官方开始尝试推出和 iOS `PlatformView` 类似的新 `Hybrid Composition` 模式，本篇将通过三小节对比介绍 `Hybrid Composition` 的使用和原理，一起来吃“螃蟹”吧～

> **反复提醒，是 1.20 不是 1.2 ～～～**

一、旧版本的 VirtualDisplay

1.20 之前在 Flutter 中通过将 `AndroidView` 需要渲染的内容绘制到 `VirtualDisplays` 中，然后在 `VirtualDisplay` 对应的内存中，绘制的画面就可以通过其 `Surface` 获取得到。

> `VirtualDisplay` 类似于一个虚拟显示区域，需要结合 `DisplayManager` 一起调用，一般在副屏显示或者录屏场景下会用到。 `VirtualDisplay` 会将虚拟显示区域的内容渲染在一个 `Surface` 上。



如上图所示，**简单来说就是原生控件的内容被绘制到内存里，然后 Flutter Engine 通过相对应的 `textureId` 就可以获取到控件的渲染数据并显示出来。**

这种实现方式最大的问题就在与触摸事件、文字输入和键盘焦点等方面存在很多诸多需要处理的问题；在 iOS 并不使用类似 `VirtualDisplay` 的方法，而是**通过将 Flutter UI 分为两个透明纹理来完成组合：一个在 iOS 平台视图之下，一个在其上面。**

所以这样的好处就是：需要在“iOS平台”视图下方呈现的Flutter UI，最终会被绘制到其下方的纹理上；而需要在“平台”上方呈现的Flutter UI，最终会被绘制在其上方的纹理。**它们只需要在最后组合起来就可以了。**

通常这种方法更好，因为这意味着 Native View 可以直接参与到 Flutter 的 UI 层次结构中。

二、 接入 Hybrid Composition

官方和社区不懈的努力下， 1.20 版本开始在 Android 上新增了 `Hybrid Composition` 的 `PlatformView` 实现，该实现将解决以前存在于 Android 上的大部分和 `PlatformView` 相关的问题，比如**华为手机上键盘弹出后 Web 界面离奇消失等玄学异常**。

使用 `Hybrid Composition` 需要使用到 PlatformViewLink、AndroidViewSurface 和 PlatformViewsService 这三个对象，首先我们要创建一个 dart 控件：

- 通过 `PlatformViewLink` 的 `viewType` 注册了一个和原生层对应的注册名称，这和之前的 `PlatformView` 注册一样；
- 然后在 `surfaceFactory` 返回一个 `AndroidViewSurface` 用于处理绘制和接受触摸事件；
- 最后在 `onCreatePlatformView` 方法使用 `PlatformViewsService` 初始化 `AndroidViewSurface` 和初始化所需要的参数，同时通过 Engine 去触发原生层的显示。

```
Widget build(BuildContext context) {
  // This is used in the platform side to register the viev
  final String viewType = 'hybrid-view-type';
  // Pass parameters to the platform side.
  final Map<String, dynamic> creationParams = <String, dyna

  return PlatformViewLink(
    viewType: viewType,
    surfaceFactory:
        (BuildContext context, PlatformViewController contr
      return AndroidViewSurface(
        controller: controller,
        gestureRecognizers: const <Factory<OneSequenceGestu

        hitTestBehavior: PlatformViewHitTestBehavior.opaque
      );
    },
    onCreatePlatformView: (PlatformViewCreationParams param
      return PlatformViewsService.initSurfaceAndroidView(
        id: params.id,
        viewType: viewType,
        layoutDirection: TextDirection.ltr,
        creationParams: creationParams,
        creationParamsCodec: StandardMessageCodec(),
      )
        ..addOnPlatformViewCreatedListener(params.onPlatfor
        ..create();
    },
  );
}
```

接下来来到 Android 原生层，在原生通过继承 `PlatformView` 然后通过 `getView` 方法返回需要渲染的控件。

```
package dev.flutter.example;

import android.content.Context;
import android.graphics.Color;
import android.view.View;
import android.widget.TextView;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import io.flutter.plugin.platform.PlatformView;

class NativeView implements PlatformView {
   @NonNull private final TextView textView;

    NativeView(@NonNull Context context, int id, @Nullable
        textView = new TextView(context);
        textView.setTextSize(72);
        textView.setBackgroundColor(Color.rgb(255, 255, 25
        textView.setText("Rendered on a native Android viev
    }

    @NonNull
    @Override
    public View getView() {
        return textView;
    }

    @Override
    public void dispose() {}
}
```

之后再继承 `PlatformViewFactory` 通过 `create` 方法来加载和初始化 `PlatformView` 。

```
package dev.flutter.example;

import android.content.Context;
import android.view.View;
import androidx.annotation.Nullable;
import androidx.annotation.NonNull;
import io.flutter.plugin.common.BinaryMessenger;
import io.flutter.plugin.common.StandardMessageCodec;
import io.flutter.plugin.platform.PlatformView;
import io.flutter.plugin.platform.PlatformViewFactory;
import java.util.Map;

class NativeViewFactory extends PlatformViewFactory {
  @NonNull private final BinaryMessenger messenger;
  @NonNull private final View containerView;

  NativeViewFactory(@NonNull BinaryMessenger messenger, @No
    super(StandardMessageCodec.INSTANCE);
    this.messenger = messenger;
    this.containerView = containerView;
  }

  @NonNull
  @Override
  public PlatformView create(@NonNull Context context, int
    final Map<String, Object> creationParams = (Map<String,
    return new NativeView(context, id, creationParams);
  }
}
```

最后在 `MainActivity` 通过 `flutterEngine` 的 `getPlatformViewsController` 去注册 `NativeViewFactory` 。

```
package dev.flutter.example;

import androidx.annotation.NonNull;
import io.flutter.embedding.android.FlutterActivity;
import io.flutter.embedding.engine.FlutterEngine;

public class MainActivity extends FlutterActivity {
    @Override
    public void configureFlutterEngine(@NonNull FlutterEngi
        flutterEngine
            .getPlatformViewsController()
            .getRegistry()
            .registerViewFactory("hybrid-view-type", new Na
    }
}
```

当然，如果需要在 Android 上启用 `Hybrid Composition` ，还需要在 `AndroidManifest.xml` 添加如下所示代码来启用配置：

```
<manifest xmlns:android="http://schemas.android.com/apk/res
    package="dev.flutter.example">
    <application
        android:name="io.flutter.app.FlutterApplication"
        android:label="hybrid"
        android:icon="@mipmap/ic_launcher">
        <!-- ... -->
        <!-- Hybrid composition -->

        <meta-data
            android:name="io.flutter.embedded_views_preview
            android:value="true" />
    </application>
</manifest>
```

另外，官方表示 `Hybrid composition` 在 Android 10 以上的性能表现不错，在 10 以下的版本中，Flutter 界面在屏幕上呈现的速度会变慢，这个开销是因为 Flutter 帧需要与 Android 视图系统同步造成的。

为了缓解此问题，应该避免在 Dart 执行动画时显示原生控件，例如可以使用placeholder 来原生控件的屏幕截图，并在这些动画发生时直接使用这个 placeholder。

三、 Hybrid Composition 的特点和实现原理

要介绍 `Hybrid Composition` 的实现，就不得不介绍本次新增的一个对象：`FlutterImageView` 。

> `FlutterImageView` 并不是一般意义上的 `ImageView` 。

事实上 `Hybrid Composition` 上混合原生控件所需的图层合成就是通过 `FlutterImageView` 来实现。 `FlutterImageView` 本身是一个普通的原生 `View` , 它通过实现了 `RenderSurface` 接口从而实现如 `FlutterSurfaceView` 的部分能力。

在 `FlutterImageView` 内部主要有 `ImageReader` 、 `Image` 和 `Bitmap` 三种类，其中：

- `ImageReader` 可以简单理解为就是能够存储 `Image` 数据的对象，并且可以提供 `Surface` 用于绘制接受原生层的 `Image` 数据。
- `Image` 就是包含了 `ByteBuffers` 的像素数据，它和 `ImageReader` 一般用在原生的如 `Camera` 相关的领域。
- `Bitmap` 是将 `Image` 转化为可以绘制的位图，然后在 `FlutterImageView` 内通过 `Canvas` 绘制出来。

可以看到 **`FlutterImageView` 可以提供 `Surface` ，可以读取到 `Surface` 的 `Image` 数据，然后通过 `Bitmap` 绘制出来。**

而在 `FlutterImageView` 中提供有 `background` 和 `overlay` 两种 `SurfaceKind` ，其中：

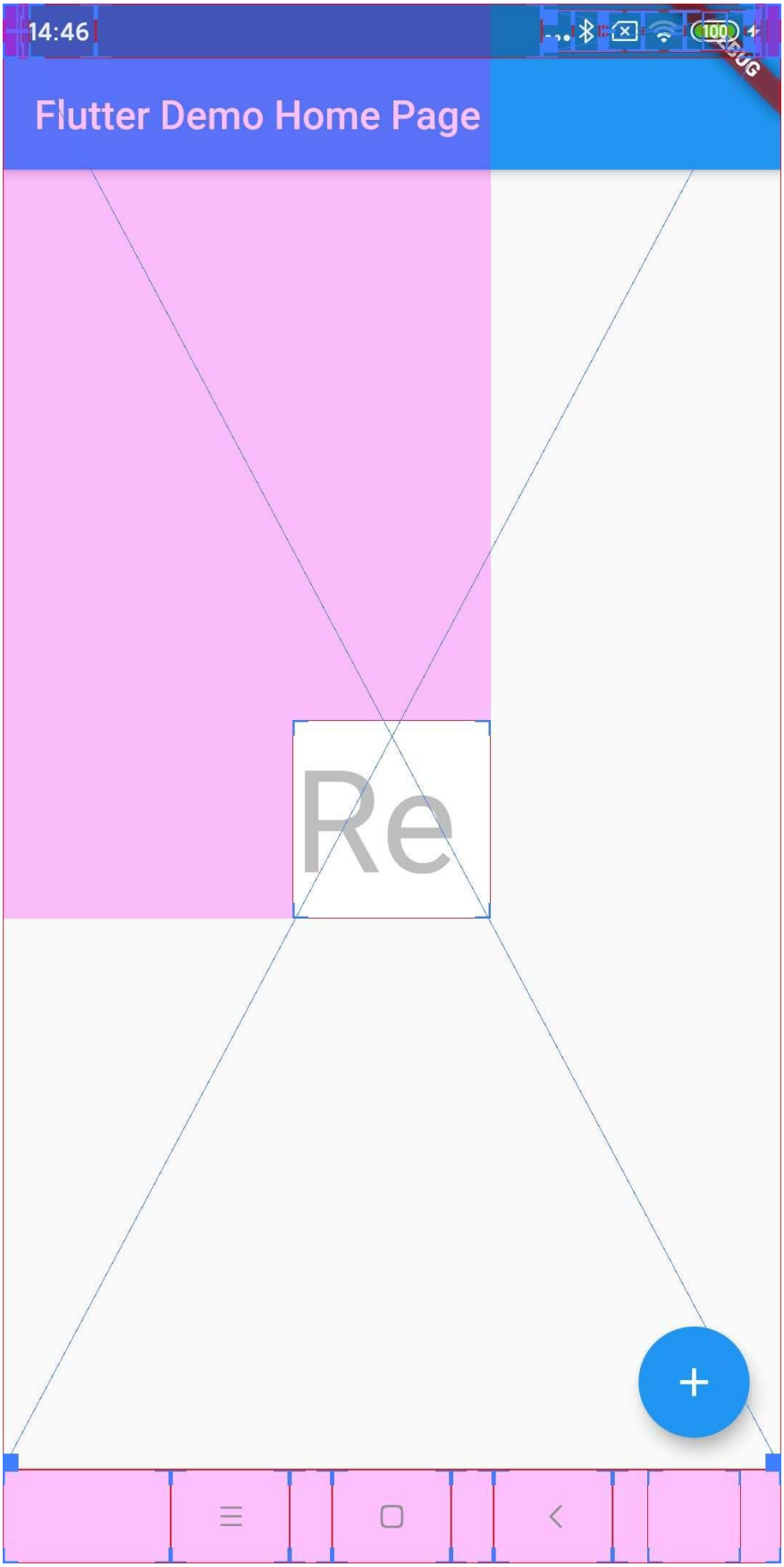
- `background` 适用于默认下 `FlutterView` 的渲染模式，也就是 Flutter 主应用的渲染默认，所以 `FlutterView` 其实现在有 `surface` 、 `texture` 和 `image` 三种 `RenderMode` 。
- `overlay` 就是用于上面所说的 `Hybrid Composition` 下用于和 `PlatformView` 合成的模式。

另外还有一点可以看到，在 `PlatformViewsController` 里有 `createAndroidViewForPlatformView` 和 `createVirtualDisplayForPlatformView` 两个方法，这也是 Flutter 官方在提供 `Hybrid Composition` 的同时也兼容 `VirtualDisplay` 默认的一种做法。

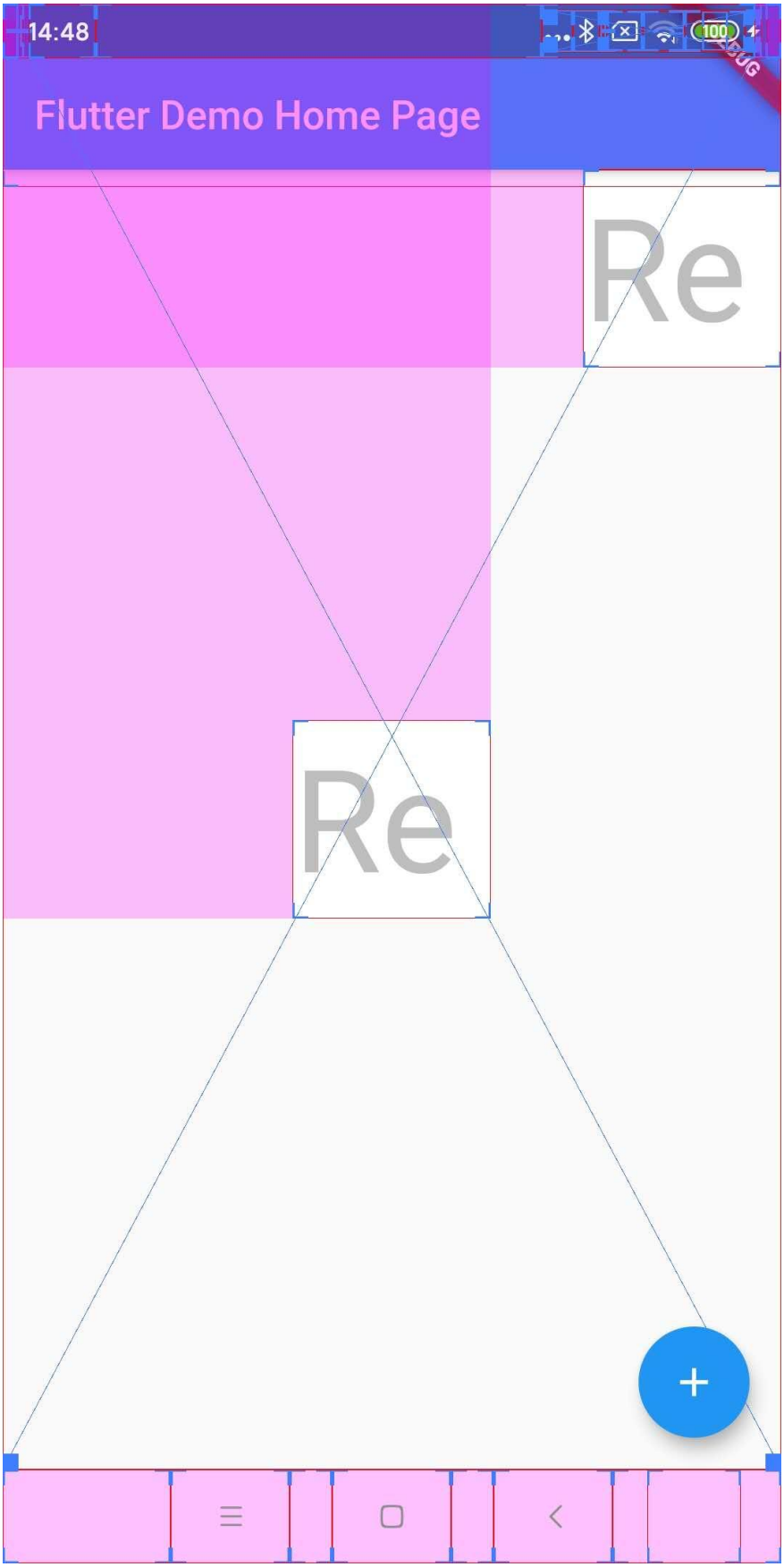
> `Hybrid Composition` Dart 层通过 `PlatformViewsService` 触发原生的 `PlatformViewsChannel` 的 `create` 方法，之后发起一个 `PlatformViewCreationRequest` 就会有 `usesHybridComposition` 的判断，如果为 ture 后面就是走的 `createAndroidViewForPlatformView` 。

那么 `Hybrid Composition` 模式下 `FlutterImageView` 是如何工作的呢?

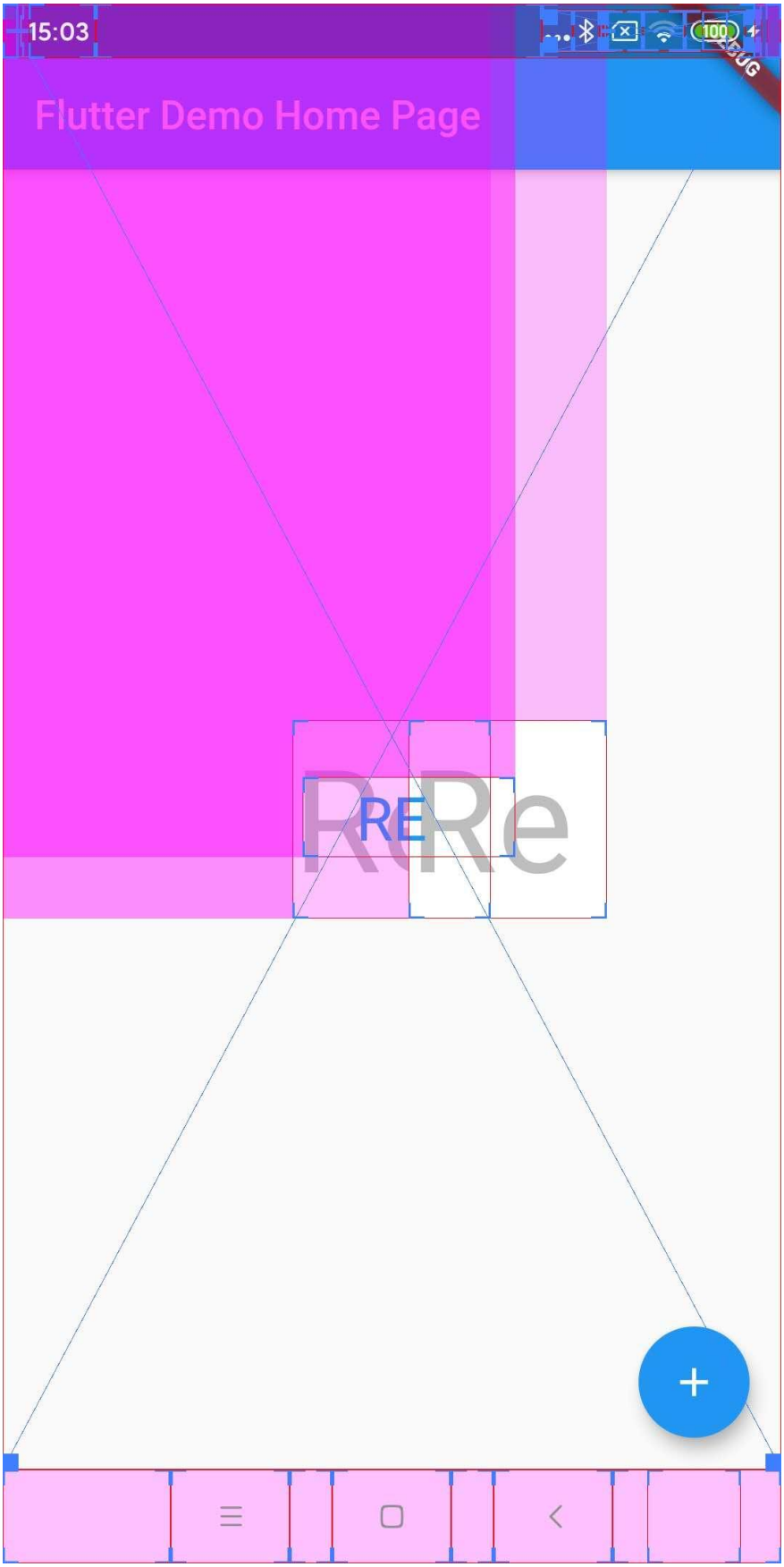
首先我们把上面第二小节的例子跑起来，同时打开 Android 手机的布局边界，可以看到屏幕中间出现了一个包含 `Re` 的白色小方块。通过布局边界可以看到， `Re` 白色小方块其实是一个原生控件。


14:46
100
Flutter Demo Home Page
Re

接着用同样的代码在不同位置增加一个 `Re` 白色小方块，可以看到屏幕的右上角又多了一个有布局边界的 `Re` 白色小方块，所以可以看到 `Hybrid Composition` 模式下的 `PlatformView` 是通过某种原生控件显示出来的。


14:48
100
DEBUG
Flutter Demo Home Page
Re
Re

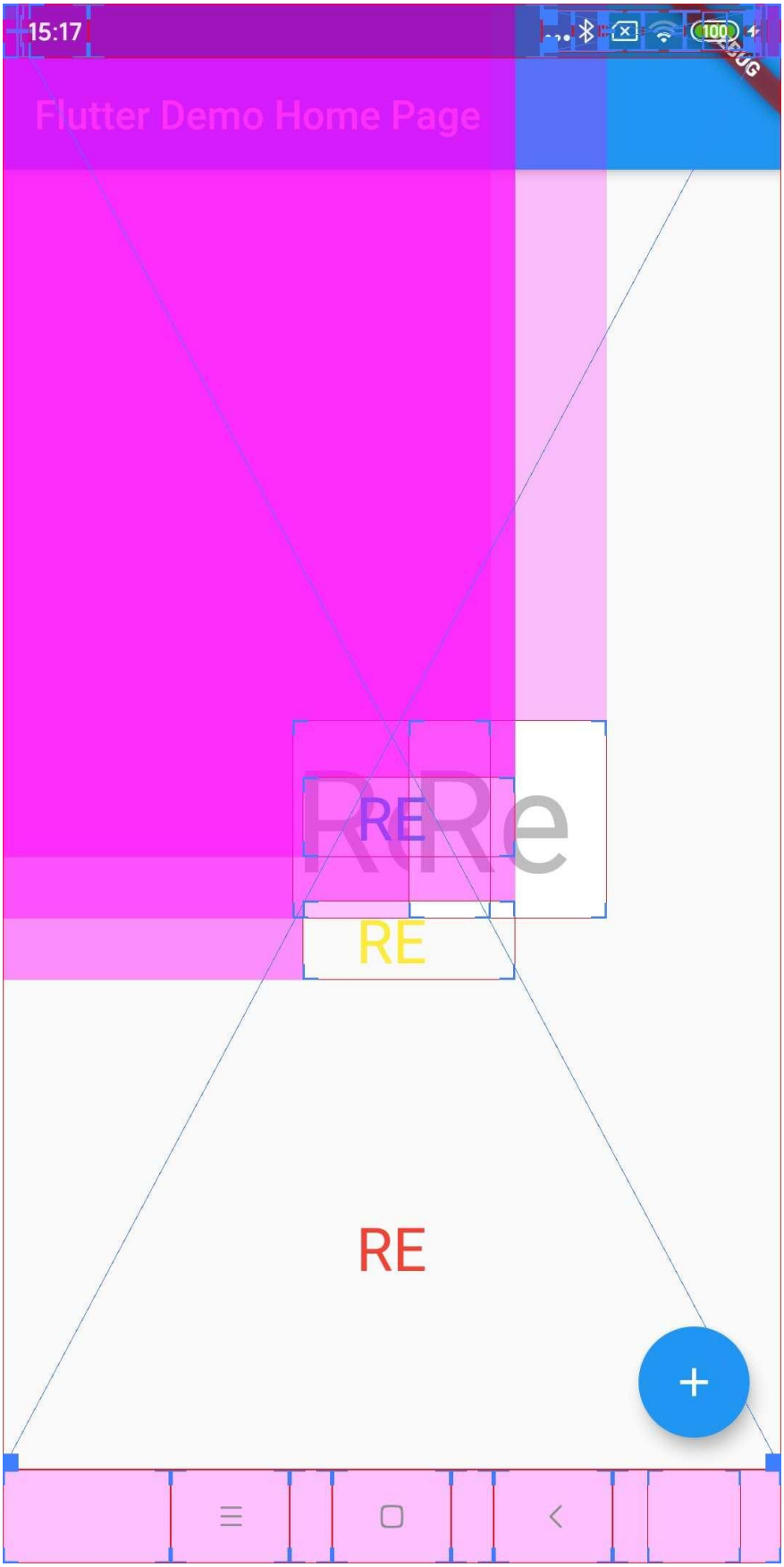
但是我们就会想了，在 `Flutter` 上放原生控件有什么稀奇的？这就算是图层合成了？那么接着把两个 `Re` 白色小方块放到一起，然后在它们上面不用 `PlatformView` 而是直接用默认的 `Text` 绘制一个蓝色的 `Re` 文本。



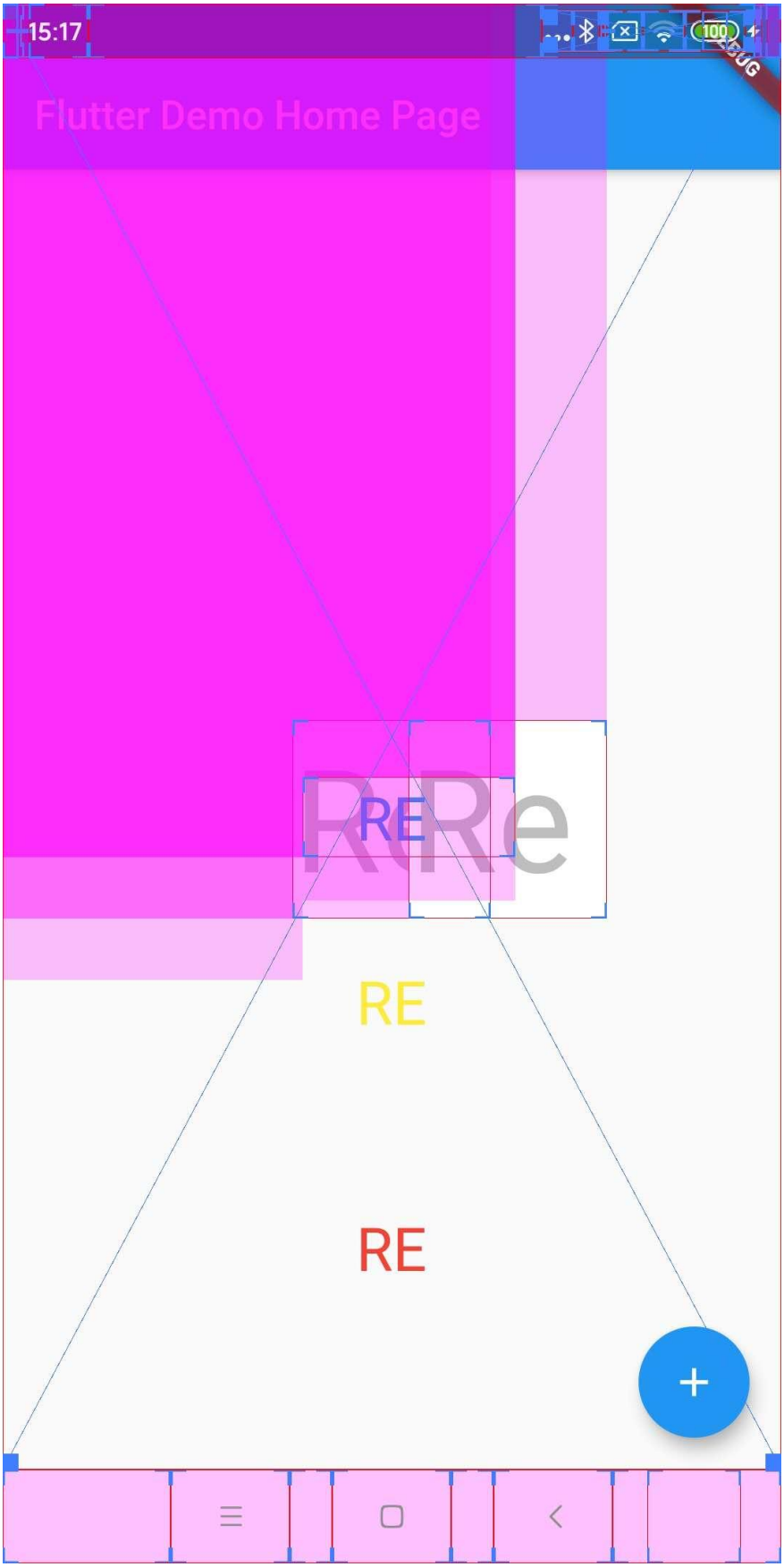
看到没有？在不用 `PlatformView` 的情况下， `Text` 绘制的蓝色的 `Re` 文本居然可以显示在白色不透明的原生 `Re` 白色小方块上！！！

> 也许有的小伙伴会说，这有什么稀奇的？但是知道 `Flutter` 首先原理的应该知道， `Flutter` 在原生层默认情况下就是一个 `SurfaceView` ，然后 Engine 把所有画面控件渲染到这个 `Surface` 上。
>
> 但是现在你看到了什么？我们在 Dart 层的 `Text` 蓝色的 `Re` 文本居然可以现在到 `Re` 白色小方块上，这说明 `Hybrid Composition` 不仅仅是把原生控件放到 Flutter 上那么简单。

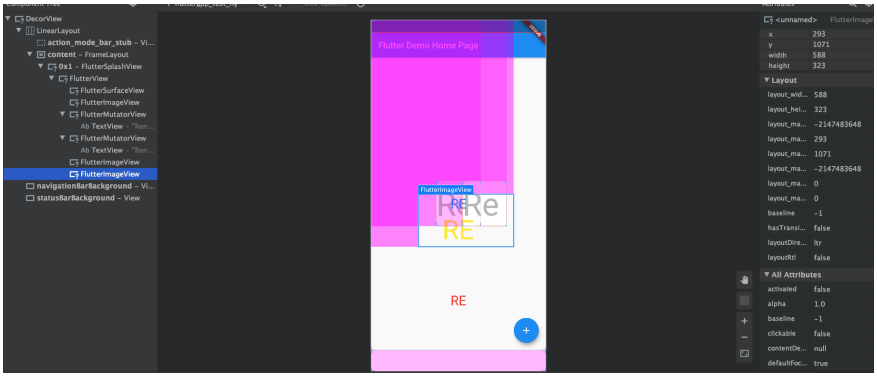
然后我们又发现了另外一个奇怪的问题，**用 Flutter 默认 `Text` 绘制的蓝色的 `Re` 文本居然也有原生的布局边界显示**？所以我们又用默认 `Text` 增加了黄色的 `Re` 文本和红色的 `Re` 文本，可以看到**只有和 `PlatformView` 有交集的 `Text` 出现了布局边界。**


15:17
100
DEBUG
Flutter Demo Home Page
RE
RE
RE

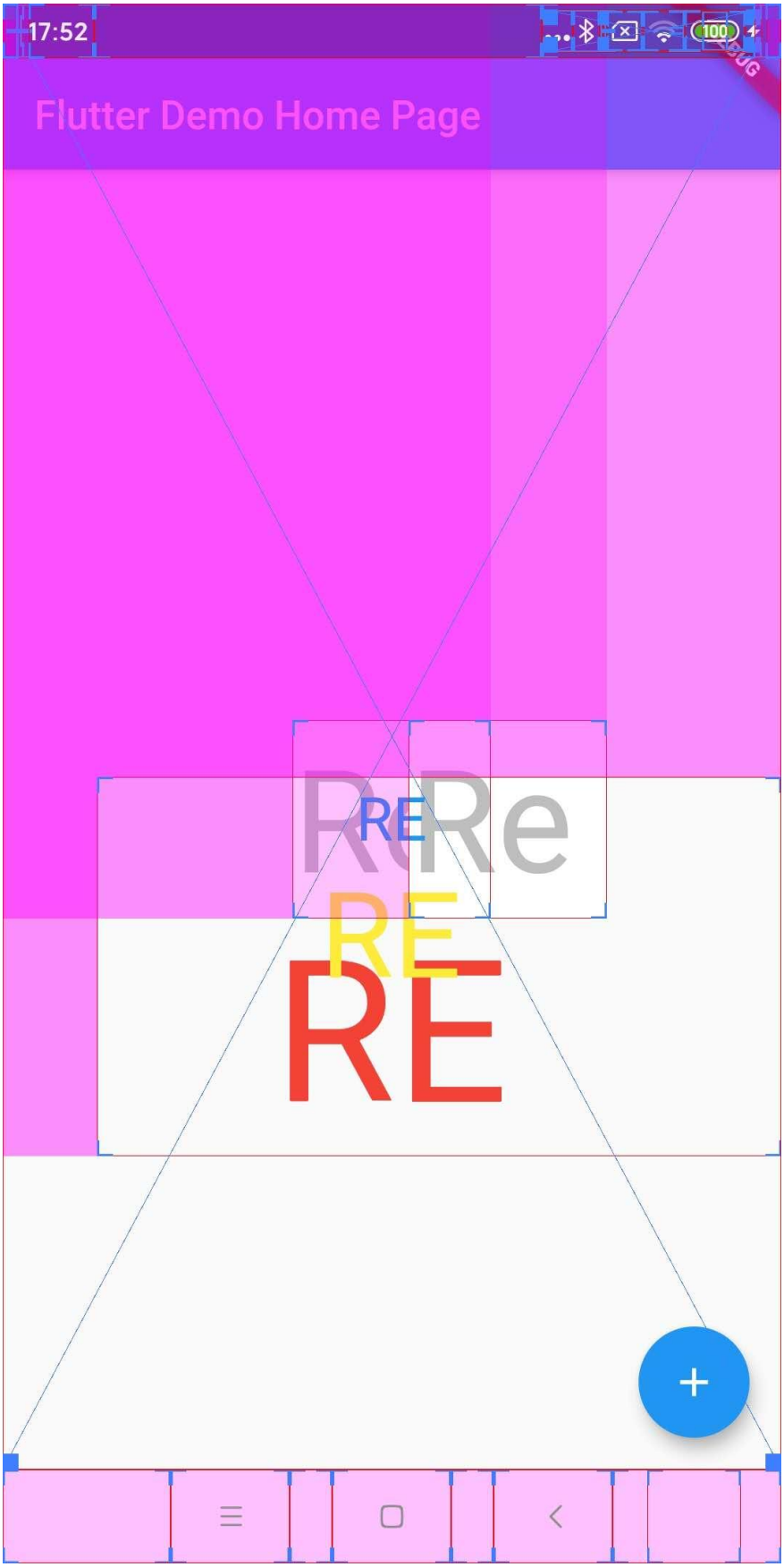
接着将黄色的 `Re` 文本往下调整后，可以看到黄色 `Re` 文本的布局边界也消失了，所以可以判定 `Hybrid Composition` 下 Dart 控件之所以可以显示在原生控件之上，是因为在和 `PlatformView` 有交集时通过某种原生控件重新绘制。



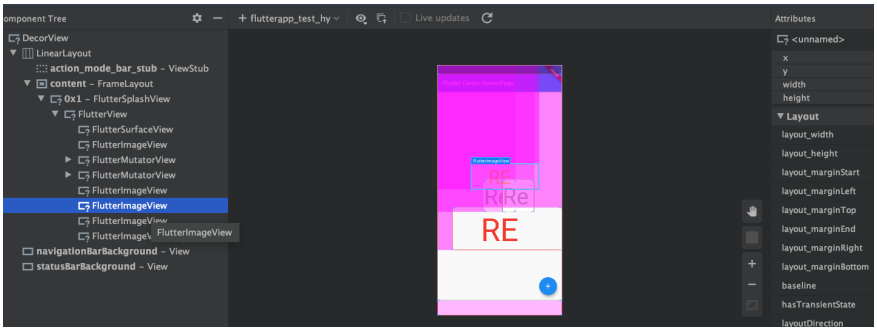
所以我们通过 `Layout Inspector` 可以看到，**重叠的 `Text` 控件是通过 `FlutterImageView` 层来实现渲染。**



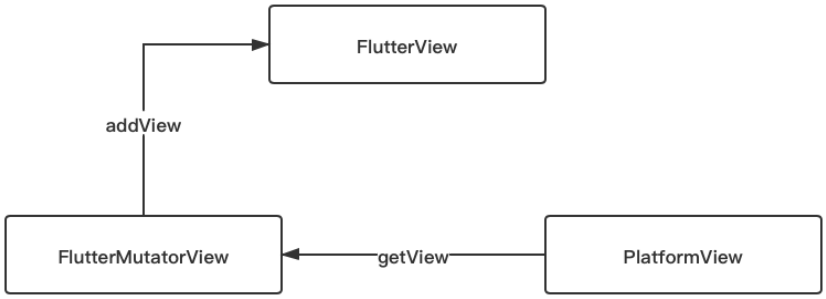
另外还有一个有趣的现象，那就是当 **Flutter 有不只一个默认的控件本被显示在一个 `PlatformView` 区域上时，那么这几个控件会共用一个 `FlutterImageView` 。**


17:52
100
Flutter Demo Home Page
RE
RE
RE

而如果他们不在一个区域内，那么就会各自使用自己的 `FlutterImageView` 。另外可以注意到，**用 `Hybrid Composition` 默认接入的 `PlatformView` 是一个 `FlutterMutatorView`** 。



其实 `FlutterMutatorView` 是用于调整原生控件接入到 `FlutterView` 的位置和 `Matrix` 的，一般情况下 `Hybrid Composition` 下的 `PlatformView` 接入关系是：



所以 `PlatformView` 是通过 `FlutterMutatorView` 把原生控件 `addView` 到 `FlutterView` 上，然后再通过 `FlutterImageView` 的能力去实现图层的混合。

那么 Flutter 是怎么判断控件需要使用 `FlutterImageView` ？

事实上可以看到，在 Engine 去 `SubmitFrame` 时，会通过 `current_frame_view_count` 去对每个 view 画面进行规划处理，然后会通过判定区域内是否需要 `CreateSurfaceIfNeeded` 函数，最终触发原生的 `createOverlaySurface` 方法去创建 `FlutterImageView` 。

```
    for (const SkRect& overlay_rect : overlay_layers.at(vie
      std::unique_ptr<SurfaceFrame> frame =
          CreateSurfaceIfNeeded(context,               //
                                view_id,               //
                                pictures.at(view_id),  //
                                overlay_rect           //
          );
      if (should_submit_current_frame) {
        frame->Submit();
      }
    }
```

至于在 Dart 层面 `PlatformViewSurface` 就是通过 `PlatformViewRenderBox` 去添加 `PlatformViewLayer` ，然后再通过在 `ui.SceneBuilder` 的 `addPlatformView` 调用 Engine 添加 `Layer` 信息。（这部分内容可见 《Flutter 画面渲染的全面解析》）

其实还有很多的实现细节没介绍，比如：

- `onDisplayPlatformView` 方法，也就是在展示 `PlatformView` 时，会调用 `flutterView.convertToImageView` 方法将 `renderSurface` 切换为 `flutterImageView` ；
- 在 `initializePlatformViewIfNeeded` 方法里初始化过的 `PlatformViews` 不会再次初始化创建；
- `FlutterImagaeView` 在 `createImageReader` 和 `updateCurrentBitmap` 时， Android 10 上可以通过 GPU 实现硬件加速，这也是为什么 `Hybrid Composition` 在 Android 10 上性能较好的原因。

因为篇（tou）幅(lan)剩下就不一一展开了，目前 `Hybrid Composition` 已经在 1.20 stable 版本上可用了，也解决了我在键盘上的一些问题，当然 Hybrid Composition 能否经受住考验那只能让时间决定了，毕竟一步一个坑不是么～

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

大家好我是 **《Flutter开发实战详解》** 的作者郭树煜，很高兴今天有机会在这里和大家分享关于 Flutter 和大前端的话题，今天我主要就从 *Flutter* 、*大前端和写作* 这三个方面给大家分享一些我的理解和想法。

1、 我为什么选择 Flutter?

初识Flutter

我接触 Flutter 的契机是因为要做一场公司的内部技术分享，因为公司要做技术选型，所以那时候分享的主题是 *《移动端跨平台开发的现状和分析》* ，而恰好那时候 Flutter 初出茅庐，就被我加入到 ReactNative 、Weex 的对比分析中。

初识 Flutter 的我对 Flutter 的感觉就是：**“什么东西？”**

是的，我对Flutter 的第一感觉并不看好：

- 首先 **Flutter 的嵌套写法就恶心到我了；**
- 另外 Flutter 选择了 Dart 而不是 JS，要知道 Dart 语言本来就是经历过“滑铁卢”的；
- 最后当时 Flutter 的第三方支持实在少的可怜

但是当时素材不够，为了凑字数，所以我还是把 Flutter 加入到了我的调研素材里。

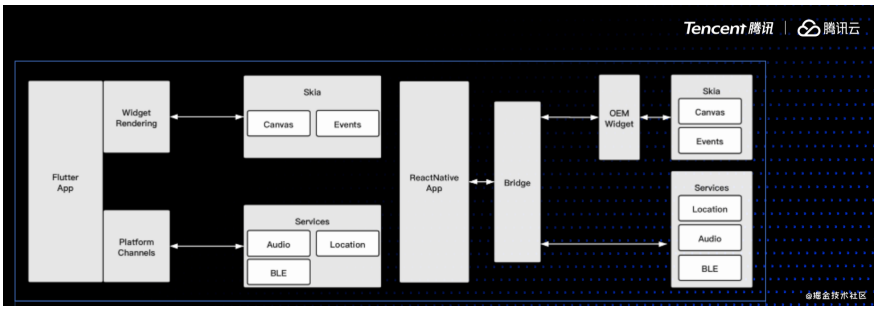
当然，真香定律，随着了解的深入，我发现我之前 **“草率了！”**

因为经历过 ReactNative 和 Weex，所以跨平台对我最大的困扰其实是**性能**和**兼容性适配**，这类问题里最具代表性的就是 Airbnb 当时刷屏的《为什么 Airbnb 放弃了 React Native?》 这篇文章，文中大致的意思是：**ReactNative 在后期不断需要面对特定的兼容性和性能问题，最终让 Airbnb 放弃了跨平台开发。**

而在不断深入了解 Flutter 之后，**首先最让我惊喜的就是它的渲染。**

我还记得当时在 Android 上开发完基本项目效果后，第一次在 iOS 上运行完居然没有出现问题，并且渲染结果还完全一致，甚至我在 Android 上使用原本应该 Android 上特有的界面效果，也自然地出现在 iOS上，这就让 Flutter 对我产生了一种极大的吸引。

Flutter 为什么能够实现优秀的跨平台渲染效果呢？这就涉及到 Flutter 独特的跨平台渲染引擎。



如图所示，这是因为 Flutter 独立的 UI 渲染让它具备了极低的平台耦合度，同时不依赖平台的渲染引擎让性能得到了提升，从而代码的复用率也得到了保证。

我为什么选择 Flutter？**就是为了提高代码逻辑的复用率，从而降低同一逻辑在不同平台因人而异的扯皮成本。**

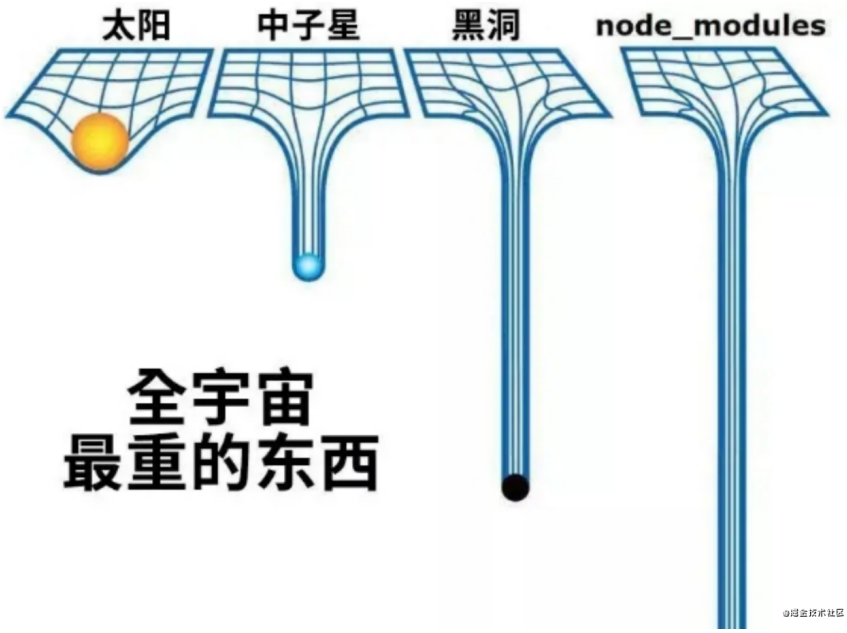
> 相信大家都遇到过，iOS 或者 Android 开发说，“这就是系统提供的效果，你要统一让对方改”的经历。。。

Flutter 的优势

那 Flutter 有什么优势呢?

- 1、 Flutter 的 UI 与平台无关，控件基本是通过独立的 Engine 绘制，不会有平台界面需要独立适配的问题，同一控件在不同平台所见即所得。
- 2、Skia 引擎绘制时直接与 GPU 交互，没有“中间商赚差价”，甚至不同平台版本支持 Vulkan 和 Metal 的渲染模式。
- 3、因为比较好的分层理念和独立的渲染引擎，Flutter 在版本升级上的成本比较低，也可以看到 Flutter 过去一两年的版本推进是迅速的。（这里的升级成本是对比 RN，因为界面可以不依赖平台api）

- 4、第三方依赖的深度很低，官方框架的依赖也很少，更容易追溯，这其实和 Flutter 初始创建时的理念有关系，这个后面会讲到，至于为什么说这个依赖深度，大家看这张图就知道了。



> Flutter 的依赖下载也是全局共享的，不像 `node_module` 每个地方一份。说起来以前就遇到过，同一个项目，因为硬盘空间不够而删除了一些项目的 `node_module` ，后来重新安装完居然就报错了，然后我们只能在黑洞里去找问题的节点。

Flutter 的劣势

当然，吹了一波 Flutter 后，也不是说 Flutter 就一定比其他框架好，一些场景下其实 ReactNative 明显是比 Flutter 更优秀的。

- 1、**混合开发上 Flutter 是没有优势，甚至说很拖垮**。真的是“成也萧何败也萧何”，Flutter 因为只需要平台的 `Surface` ，说到底就是提供一个 `View` 就可以了，所以它的页面堆栈和渲染树是脱离原生的，这就造成不管是和原生混合 Flutter ，还是 Flutter 混合原生都比较大的成本。

> 比如在原生框架中混入 Flutter，那页面堆栈的混合就是一个大问题，虽然有不少第三方框架做这个，不过总的体验和稳定性上还是存在偏差；其次就是在 Flutter 中混入原生 UI，最常见的就是 WebView，也是因为 Flutter 设计的特殊性，它的性能和键盘问题一直都是让人头疼的存在。

- 2、 也就是热更新，它没有像 ReactNative 一样成熟的 code-push 机制，打包后的二进制产物本身就不符合平台的动态化策略，虽然也有不少第三方框架利用类似映射等的方式，但是维护成本还是挺高的。
- 3、通过前面的介绍应该理解到，Flutter 其实是一个独立的类似游戏的引擎，所以它对内存的消耗是比较大的，特别如果你是用于混合开发的话，那相当于在你的应用里接入一个小型的游戏引擎。
- 4、体积，Flutter 打包后的动态库必定需要占用一定的体积，在 Android 上因为系统本身就支持 Skia 所以比较好，但是在 iOS 需要把 Skia 引擎也打包进 App 里，所以体积会变大不少。

Flutter 的嵌套问题

好了，说了那么多 Flutter 的好与坏，最后来说说大家比较关系的 Flutter 的嵌套问题。为什么可以这样设计嵌套?

其实有一定原因是： **`Widget` 并不是真正的控件**，如图所示的代码，其中 `testUseAll` 这个 `Text` 在同一个页面下在三处地方被使用，并且代码可以正常运行渲染，如果是一个真正的 `View`，是不能在一个页面下这样被多个地方加载使用的。

```
final textUseAll = new Text(
  "3333333",
  style: new TextStyle(fontSize: 18, color: Colors.red),
);
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Row(
        children: <Widget>[
          textUseAll,
          Text(' GSY '),
          textUseAll,
        ],
      ),
      floatingActionButton:  FloatingActionButton(
        onPressed: () {},
        child: Container(
          child: textUseAll,
        ),
      ),
    );
  }
}
```

在 Flutter 设定里， `Widget` 是配置文件告诉 Flutter 你想要怎么渲染，`Widget` 在 Flutter 里会经过 `Element` 、 `RenderObject` 、乃至 `Layer` 最终去进行渲染，所以作为配置文件的 `Widget` 可以是 `@immutable` ，可以每次状态更新都被重构。

所以 Flutter 不怕 `Widget` 嵌套带来的性能问题，因为它不是正式干活的，嵌套不会带来严重的性能损失， 但是，我们同样怕嵌套带来的可视化问题啊。

对于语法嵌套问题，不可否认 Flutter 的语法糖不过丰富，但是官方其实还是提供了一些解决方法，比如把配置文件抽象化，其中就是 Flutter 里最常见的 `Container` 。

Flutter 中 `Container` 本身只是一个容器 `Widget` ，它通过搭配了 `Padding` 、 `Align` 、 `ClipPath` 、 `ColoredBox` 、 `DecoratedBox` 、 `Transform` 等基础 `Widget` 来实现了灵活的功能搭配。

这其实就是 Flutter 一开始提供的思路之一，所以可以看出来 Flutter 和以往大家理解的 UI 框架和跨平台框架不一样，总结一下：

- 首先他的跨平台方式的创新，能实现更好的性能和代码效果，它的设计也让它不会因为嵌套而带来影响体验的性能损失。
- Flutter 在 Dart Framework 的分层逻辑，导致了开发者写的 `Widget` 只是配置文件， 内部的 `Element` 才算是实例，而 ·RenderObject· 才是绘制对象等等，这和以往的 UI 框架不大一样。

所以如果想要理解 Flutter ，你就要先理解 Flutter 中灵魂的设计，理解 `Widget` 、 `Element` 、 `RenderObject` 、 `Layer` 等的定位和设计，这也是我之前写的书里像表达的，在第三和第四章里核心的内容。

2、 2021年大前端有怎样的技术趋势

Flutter 属不属于前端

为什么说完 Flutter 才来聊大前端呢？其实我接触过不少前端开发和我说：“ Flutter 不应该属于前端”。

但是有趣的是 Flutter 就是来源于前端 Chrome 团队。

Flutter 的创始人和整个团队几乎都是来自 Web，在 Flutter 负责人 Eric 的相关访谈中， Eric 表示 Flutter 来自 Chrome 内部的一个实验，他们把一些乱七八糟的 Web 规范去掉后，在一些基准测试的性能居然能快 20 倍，因此 Google 内部开始立项，Flutter 出现了。

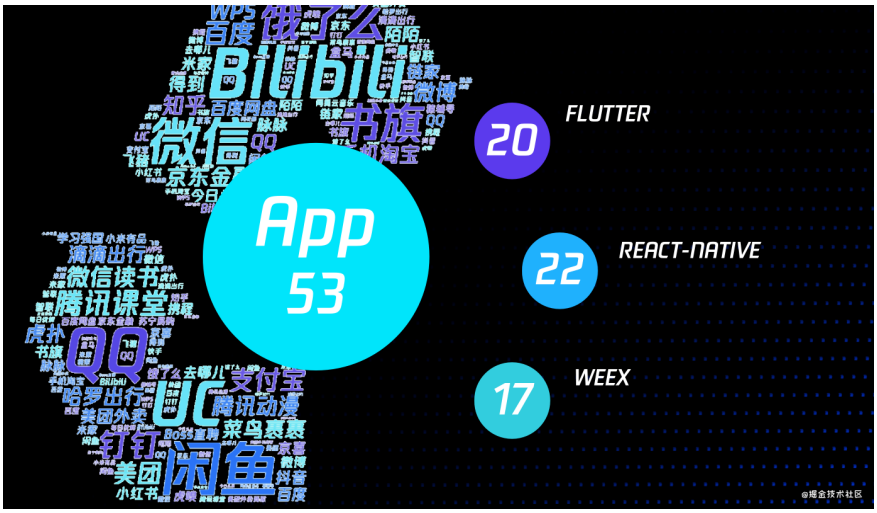
> 所以语法糖也少了。

另外 Dart 也是起源于 Web ，**可以说 Flutter 其实就是从前端诞生，并应用于客户端的技术。**

跨平台在国内运用多吗?

那大前端下跨平台技术在国内的运用多吗？答案是肯定的，这里可以看一个简单的数据，我之前自己单独做过一个53款 App 的数据统计，其中对于跨平台的运用：

- Flutter 有 20 款；
- React Native 有 22 款；
- Weex 有 17 款；



从数据就看得出来，其中不少 App 使用了两种以上的跨平台技术，甚至有三种都使用的，对于这部分数据感兴趣的可以在我公众号或者掘金上找我的的这篇分析。

另外这里我给大家推荐一个开源项目： `LibChecker` ，它可以查看你手机上已安装的包详细信息，不看不知道，自己看看才知道现在跨平台技术在现实中的运用情况，其中如图所示是我手机上应用使用 Flutter 、React

Native 、Weex 的应用情况，所以可以看到，大前端和跨平台已经深入到各种开发需求中。

当然，跨平台需要的是平台！所以平台的原生开发是基础支撑，所以平台本身的开发是不会“凉”的，这是一个重要的前提。

跨平台不是“降维打击”，跨平台和大前端只是让开发的能力更加通用，所以通过各种跨平台能力，客户端和前端融合成了大前端，注意是“融合”。

大前端需要什么？

知识的广度，这里的广度不是指你要懂很多技术，而是你要会技术的抽象与通用能力的拓展。

当然有人就说 **“嚼多不烂，杂而不精的情况怎么办？”**

这个确实是个问题，但是思考这个问题之前，我在和一些网友的交流中发现，有时候大家都只是停留在思考这个问题上， **主要是用这个问题来说服自己不需要学新的** 。

多而不精是对的，但是反之并不是，不是你不学多就自然而然的精了，所以这个属于个人衡量的问题。

如果按照简单的分数划分，精通不属于 0-60 分的范畴，而是 80-99分，这部分需要的毅力、悟性，最重要是要有工作平台的支持。

为什么这么说？在交流过程中一些人说想要深入xxx去精通某想技术，但是最终还是“三过门而不入”。

因为很多时候精通某项技术，是需要业务场景去验证和推进的，如果不是大体量的业务场景，没有经历过各种极端的考验，**很多时候所谓的精通只是表层精通**。

而大部分在“考试”中，一般人其实都可以做到范围是 75 分左右，**这个75分这就是我理解的大前端**。

在 0-60 分这部分是大多数人能掌握的程度，然后 60-75 分需要费些心思就可以达到的，**而这个区域内的能力是可以快速地横向应用**。

所以所谓的精通不是熟练掌握了 React，Vue 等框架调用和源码的背诵，也不是精通 Flutter ，Android 等框架的 API 调用技巧，而是你理解了这些东西的核心思想和理念。

- 比如把 Android 上关于 Canvas 的技能就利用到 Flutter 和 Web 上；
- 比如响应式开发和状态管理的理解可以让我在 RN 和 Flutter 也能很好地利用，甚至未来的 Jetpack Compose 也可以快速的上手；

技术的抽象能力让你的技术可以迁移适配，所以在我的眼中，大前端的未来 **“不是我会什么所以做什么，而需要什么我就能做什么。”**

3、 程序员为什么要写作?

这个话题其实有点“跳脱”了，但是这其实也是一个比较有用的过程。

写作的动力是什么?

其实大家都应该发现一个有趣的现象，那就是古人写诗的时候，优秀的诗句里有不少是怀才不足的主题。

“我觉得我被低估了，我希望找到伯乐。”

其实这在程序员的圈子里也类似，**如果有一天你发现同事突然开始更新博客和做开源项目，那么不用怀疑，你同事可能在打算跳槽了。**

其实说这个的意思是，写作是需要动力的，或者是写作的初始动力是什么，你是为了什么开始写作的。

一般就有就比如前面说的：

- 为了跳槽的；
- 还有为了给职业生涯增加背书，展示自己；
- 甚至也有是为了赚外快的；

而在经历了初期的写作冲动后，后续如何保持写作才是最难的，而这里面有个关键的因素。

“心态”

支持继续写作的一个关键点就是“心态”。

很多时候我们写东西会发现：*“哦，原来网上已经有人写过了”*，之后就放弃不写了，这是很正常的心态，但是这样的放弃就会让你越来越难产出内容，因为你不能保证你一定快过别人。

其实当你想写内容时，发现别人已经产出过了，那你可以参考下别人的内容是否和你的想法有什么不同，然后有什么遗漏或者可以升华的地方，甚至是你可以从另外的角度或者更系统的方式去描述你的观点。

> 站在别人的肩膀上可以看得更远，当然不是让你 cv 抄袭，因为抄袭的意义不大，又不是小学生罚默写。

这就是写作人需要的心态之一，**不要害怕撞文**。

而写作还有另外心态就是“服务”， 写作其实就是一种服务行业，一开始很多时候我们需要面临的问题就是没人看，有时会有鄙视或者否定，甚至有的知识单纯的情绪发泄，这些我都遇到过，因为你不可能面面俱到地服务好所有人。

我们提供的服务是内容，期望得到的是关注和交流，所以要比较自己陷入不必要的情绪陷阱。

“*不要成为别人情绪的垃圾桶，那就在开始时把盖子盖上或者远离*”，自此之后我佛系了，面对不善意的评论我会选择屏蔽或者直接给予肯定，这样可能帮我节省出更多的时候来做有用的事情。

最后好的写作进阶

最后说下什么是好的写作进阶：**那就是把高级的内容变成接地气的内容**

如果一篇文章你看完后觉得：*“哇，牛逼，但是为什么我就只看不懂？”*说明这篇文章并不是一篇好的科普内容，这部分经常出现在早期的“RxJava”、“协程” 等领域的文章。

那好的写作一般可以分为三个阶段：

- 1、写文档是第一步，因为你告诉别人怎么去理解你写的东西，所以如果你想开始写作，最简单就是从写文档开始，把你的东西介绍明白了，就是一个好的开始。
- 2、写源码分析是第二步，那就是学习和分享如何去理解别人的东西，这个过程可以让自己在学习的过程中有所总结，并且介绍别人的内容就是一种抽象能力的进步。
- 3、写问题解决和应用思想是第三步，告诉别人如何去理解别人的思想，这就需要作者对要介绍的内容有自己的理解，才能够把内容变成更好理解的接地气的文字内容。

其实介绍了这么多写作内容，就是想告诉大家：**“大前端是一种思想，就是让你已有的能力可以运用更广泛，而写作是帮助你把能力抽象化的一个过程。”**

这也是我在写书的经历，从如何使用 Flutter，深入理解 Flutter 的 xxx 以后，回过头来我有了新的理解。

觉得可以从这个角度去告诉后来的人应该如何理解 Flutter，这样虽然 Flutter 一直在更新，但是这部分内容是它设计的核心理念，所以它并不会过时，也能成为比较好的内容承载。

就如今天的主题就是：**大前端是一种思想，可以让已有的能力得到横向的应用，写作就是提炼你技术思想的过程，从而帮你把你的固定于某一个框架和某一个平台的能力变成抽象化能力。**

回顾了这段时间解答关于 Flutter 的各种问题后，我突然发现很多刚刚接触 Flutter 的萌新，对于 Flutter 都有着不同程度的误解，而每次重复的解释又十分浪费时间，最终我还是决定写篇文章来做个总结。

> 内容有点长，但是相信能帮你更好地去认识 Flutter 。

Flutter 的起源

Flutter 的诞生其实比较有意思，**Flutter 诞生于 Chrome 团队的一场内部实验**， 谷歌的前端团队在把前端一些“乱七八糟“的规范去掉后，发现在基准测试里性能居然提高了 20 倍，机缘巧合下 Flutter 就这么被立项。

所以 Flutter 是基于前端诞生，同时基于它的诞生缘由，可以看到 **Flutter 本身就不会有特别多的语法糖**，作为框架它比较“保守”，选择的 Dart 语言也是保守型的语言。**而它的编程模式，语法都带有浓厚的前端色彩，可是它却最先运用在移动客户端的开发。**

所以当 Flutter 面世的时候，就需要面对一个很尴尬的状态：

- **对于客户端原生开发而言，声明式的开发方式一上手就不习惯**，习惯了代码与布局分离（java\kotlin + xml ）和命令式的对象编程，声明式开发需要额外的学习成本；**同时也觉得 Flutter 的嵌套很“恶心”。**
- 对于前端开发而言，**Flutter 的环境配置很烦人**，除了 VSCode 和 Flutter SDK 之外，还需要原生的如 Java 、Gradle 、Android SDK 、XCode 等“出圈”的环境变量（时不时遇上网络问题），而且 **Flutter 所需要的原生平台知识点对前端来说很不友好**；**同时也觉得 Flutter 的嵌套很“恶心”。**

发现没有？我没有说 Dart 语言是学习成本，因为无论对于擅长 JS 的前端而言，还是对于掌握 Java\Kotlin\Swift 的客户端而言，**Dart 无论怎么看都是“弟弟”**。

另外不管是前端还是客户端，都会对 Flutter 的嵌套很“恶心”做出抨击，但是嵌套问题严重吗？这个我们后面会聊到。

综上所述， Flutter 对于前端入坑或者客户端入坑的萌新来说，都会有一定程度的门槛和心理抵触。**那对于前端或者客户端来说，有没有必须要学习 Flutter 呢?**

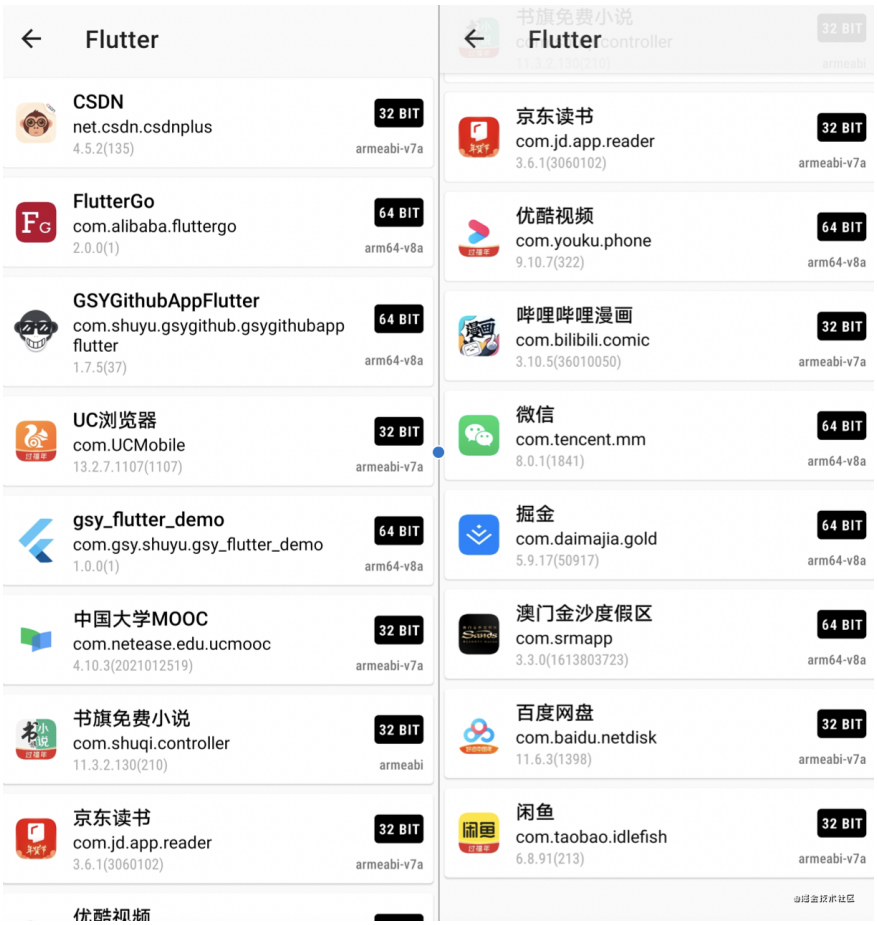
学习 Flutter 的理由

在我接触在大多 Flutter 萌新里，**有很大一部分其实是“被迫”使用 Flutter**，因为领导或者老板要求用 Flutter ，所以不得不“欲拒还迎”地开始学习 Flutter，这就是最“有力的”理由之一：“老板（领导）要”，除非你选择“跳槽”飞出三界。

1、个人竞争力层面

其实开发这个圈子很有意思，我们经常在长时间使用一项技术后，很容易就觉得这项技术很火，因为周边的人都在用，而其他的框架要凉，因为没人用的错觉，特别是在“媒体”的煽动下，“孕妇效应”很容易就带来认知上的误解。

去年中旬我在 《国内大厂在移动端跨平台的框架接入分析》 就针对 53 个样本做过简单的数据分析，可以看到其中 *flutter（19）*、*weex（17）*、*react-native（22）*，同时下图是在个人手机用 `libChecker` 统计出来使用 Flutter 的生产应用。



介绍这个只要是想表达：**Flutter 现在已经不是曾经的小众框架，这两年里它已经逐步成为主流的跨平台开发框架之一。**

所以 Flutter 确确实实可以成为你找工作的一个帮助，当然我并不推荐你从零开始学习 Flutter，**因为 Flutter 本身只是一个跨平台 UI 框架。**

> **理解上面这句话很重要，因为他可以避免你被“贩卖焦虑”，** Flutter 尽管支持移动端、Web 端和 PC 端，但是作为 UI 框架，它主要帮助我们解决的是 UI 和部分业务逻辑的“跨平台”， **而和平台相关的诸如蓝牙、平台交互、数据存储、打包构建等等都离不开原生的支持。**

现阶段的跨平台框架，不管的 Flutter 还是 react-native 和 weex ，它们的定位都是 UI 框架，**它们解决的是 UI 业务跨平台的成本**，它们的发展都离不开原生平台开发的支持。

如果原生平台都挂了，那还跨个蛋？比如现在谁还会说要跨 WinPhone ？**所以 Flutter 和原生平台应该是相互成长的局势，而不是那些《xxx制霸，###要凉的》的“节奏党”**，都是寄生和共生的关系，没有对应平台的开发经验，是很难把 Flutter 用得“愉悦”。

不过现在 Flutter 确确实实可以帮助到你的职业发展，因为通过 Flutter 放大你的业务开发能力，让你参与到更多的平台开发中，不过是大前端还是KPI。当然这些 react-native、 uni-app 也可以带给你，甚至对于前端开发来说可能更低，那为什么还要选择 Flutter 呢?

> 事实上还有一个有意思的点，对于 Android 原生开发来说，学会 Flutter 等于学会了 70% 以上的 Jetpack Compose 。

2、Flutter 的一致性

事实上从我个人一直比较推荐客户端学 Flutter ，因为对于前端来说 react-native、 uni-app 确实是性价更高的，当然好像各位的领导和老板们不是这么觉得。

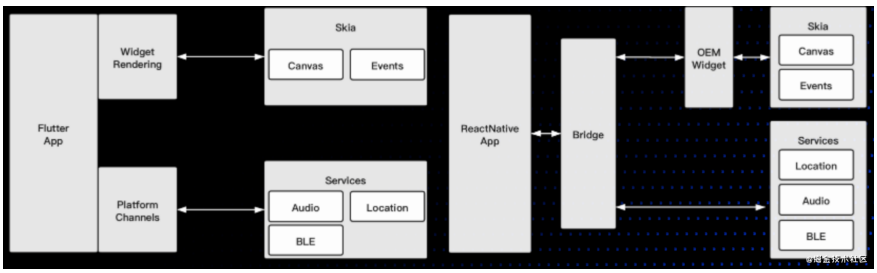
那么使用 Flutter 有什么额外的好处呢？**那就是 Flutter 的性能和一致性**。

因为 Flutter 作为 UI 框架，它是真的跨平台！ 为什么要强掉 *“真·跨平台”* ，因为和 react-native 、 weex 不同，Flutter 的控件不是通过原生控件去实现的渲染，而是由 Flutter Engine 提供的平台无关的渲染能力，也就是 **Flutter 的控件和平台没关系**。

> 简单来说，原生平台提供一个 `Surface` 作为画板，之后剩下的只需要由 Flutter 来渲染出对应的控件，而这个过程最终是打包成 AOT 的二进制完成。

所以 **Flutter 的 UI 控件可以做到所见即所得**，这个对我个人来说是很重要的进步。为什么这么说呢？这时候就需要拿 react-native 来做对比。

因为 react-native 是通过将 JS 里的控件转化为原生控件进行渲染，所以 rn 里的控件是需要依赖原生平台的控件，**所以不同系统之间原生控件的差异，同个系统的不同版本在控件上的属性和效果差异**，组合起来在后期开发过程中就是很大的维护成本。



在我 react-native 开发生涯中，就经常出现：

- 在 iOS 上调试好的样式，在 Android 上出现了异常；
- 在 Android 上生效的样式，在 iOS 上没有支持；
- 在 iOS 平台的控件效果，在 Android 上出现了不一样的展示，比如下拉刷新，`Appbar` 等；

当然，这些问题最终都可以通过 `if` `else` 和自定义平台控件来解决，但是随着项目的发展，这样的结果无疑违背了我使用跨平台的初衷。

而 Flutter 的控件特性决定了它没有这些问题，**我甚至经常只在 iOS 模拟器上开发测试所有界面逻辑，而不用担心 Android 上的兼容**，当然屏幕大小的适配是不可避免的。

> 从这个角度上不严谨地说， Flutter 更像是一个类 unity 的轻度游戏引擎，不过它提供的是 2D 的控件。

当然，Flutter 这样实现也有坏处，**那就是当你需要使用平台的控件作为混合开发时，Flutter 的成本和体验无疑被放大**，这一点上 react-native 反而有着先天的优势。

3、Flutter 的性能

其实前面也介绍过 Flutter 的性能一般情况下是比 react-native 好，关于这个也有 《Flutter vs React Native vs Native：深度性能比较》 的文章做深入的对比，这里主要介绍几个误区：

- 1、Flutter 在 debug 和 release 下的性能差距是巨大的，因为它们之间是 JIT 和 AOT 的区别。
- 2、不要在模拟器上测试性能，这个根本没有意义，因为在手机上 Flutter 会更多依赖 GPU 的能力。
- 3、混合开发 Flutter 是有性能有影响的，比如在原有 Android 项目里，把某个模块业务逻辑改用 Flutter 实现，这对性能和内存会有很大的考验，至于为什么？就是前面说过 Flutter 独立的控件渲染和堆栈管理带来的负面效果。
- 4、同一个框架在不同人手下会写出不一样的结果，一般情况下**对于普通开发者来说，流行的框架一般不会带来很大的性能瓶颈，反而是开发能力比较多导致项目的瓶颈。**

怎么学 Flutter？

当你快速搭建好环境，简单了解 Flutter 的 API 之后，学习 Flutter 在我看来主要有两个核心点：**响应式开发和 Widget 的背后是什么?**

1、响应式开发

响应式开发相信对于前端来说再熟悉不过，**这部分内容对于前端开发来说其实可以略过**，响应式编程也叫做声明式编程，这是现在前端开发的主流，当然对于客户端开发的一种趋势，比如 Jetpack Compose 、SwiftUI 。

> Jetpack Compose 和 Flutter 的相似程度绝对让你惊讶。

什么是响应式开发呢？简单来说其实就是**你不需要手动更新界面，只需要把界面通过代码“声明”好，然后把数据和界面的关系接好，数据更新了界面自然就更新了**。

从代码层面看，对于原生开发而言，**响应式开发中没有 xml 的布局，布局完全由代码完成，所见即所得，同时你也不会需要操作界面“对象”去进行赋值和更新，你所需要做的就是配置数据和界面的关系**。

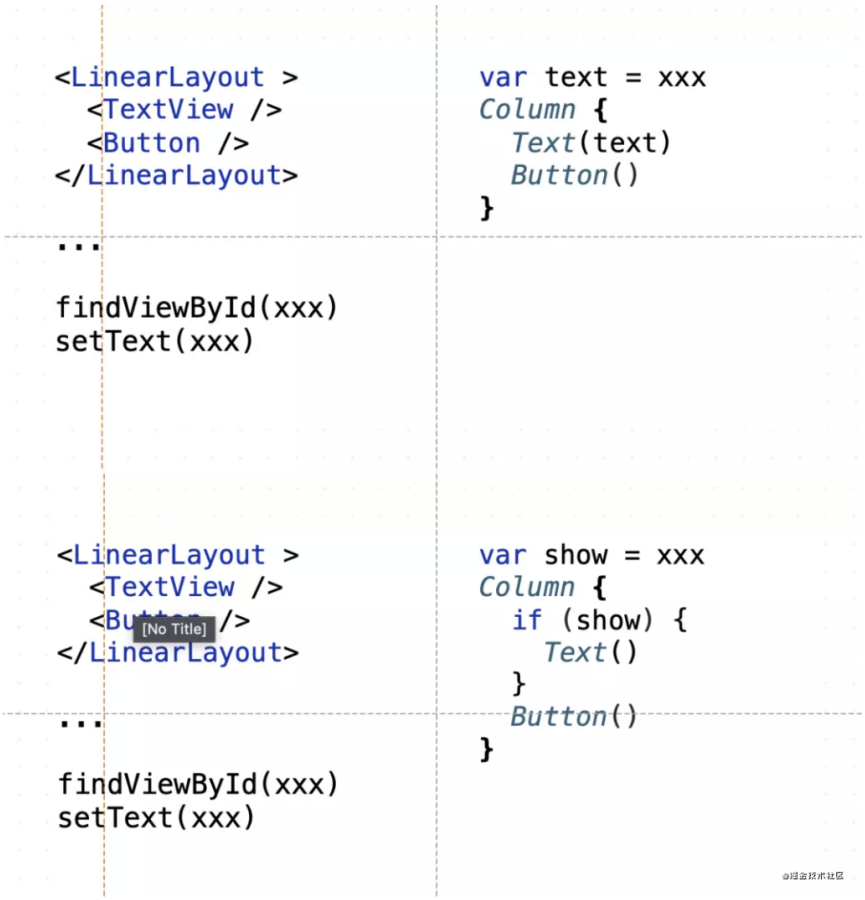
举个例子：

- 以前在 Android 上你需要写一个 xml ，然后布局一个 `TextView` ，通过 `findViewById` 得到这个对象，再调用 `setText` 去赋值；
- 现在 Flutter 里，你只需要声明一个 `Text` 的 `Widget` ，并把 `data.title` 这样的数据配置给 `Text` ，当数据改变了， `Text` 的显示内容也随之改变；

```
class TextWidget extends StatelessWidget {

  final Data data;

  TextWidget(this.data)

  @override
  Widget build(BuildContext context) {
    return Text(data.title);
  }
}
```

对于 Android 开发而言，大家可能觉得这不就是 `MVVM` 下的 `DataBinding` 也一样吗？其实还不大一样，更形象的例子，这里借用扔物线大佬在谷歌大会关于 Jetpack Compose 的分享，为什么 `Data Binding` 模式不是响应式开发：

> 因为 `Data Binding` （不管是这个库还是这种编程模式）并不能做到「声明式 UI」，或者说 声明式 UI 是一种比数据绑定更强的数据绑定，比如在 Compose 里你除了简单地绑定字符串的值，还可以用布尔类型的数据来控制界面元素是否存在，例如再创建另外一个布尔类型的变量，用它来控制你的某个文字的显示：



> 注意，当 `show` 先是 `true` 然后又变成 `false` 的时候，不是设置了一个 `setVisibility(GONE)` 这样的做法，而是直接上面的 `Text()` 在界面代码中消失了，每次数据改变所导致的界面更新看起来就跟界面关闭又重启、并用新的数据重新初始化了一遍一样，这才叫声明式 UI，这是数据绑定做不到的。
>
> 当然 Compose 并不是真的把界面重启了，它只会刷新那些需要刷新的部分，这样的话就能保证，它自动的更新界面跟我们手动更新一样高效。

在 Flutter 中也类似，当你通过这样的 `ture` 和 `false` 去布局时，是直接影响了 `Widget` 树的结构乃至更底层的渲染逻辑，所以作为 Android 开发在学习 Flutter 的时候，就需要习惯这种开发模式，“放弃” 在获取数据后，想要保存或者持有一个界面控件进行操作的想法。另外**在 Flutter 中，持有一个 Widget 控件去修改大部分时候是没意义的，也是接下来我们要聊的内容。**

2、Widget 的背后

Flutter 内一切皆 `Widget` ， `Widget` 是不可变的（immutable），每个 `Widget` 状态都代表了一帧。

理解这段话是非常重要的，这句话也是很多一开始接触 Flutter 的开发者比较迷惑的地方，因为 Flutter 中所有界面的展示效果，在代码层面都是通过 `Widget` 作为入口开始。

`Widget` 是不可变的，说明页面发生变化时 `Widget` 一定是被重新构建， `Widget` 的固定状态代表了一帧静止的画面，当画面发生改变时，对应的 Widget 一定会变化。

举个我经常说的例子，如下代码所示定义了一个 `TestWidget` ， `TestWidget` 接受传入的 `title` 和 `count` 参数显示到 `Text` 上，同时如果 `count` 大于 99，则只显示 99。

```
/// Warnning
/// This class is marked as '@immutable'
/// but one or more of its instance fields are not final
class TestWidget extends StatelessWidget {

  final String title;

  int count;

  TestWidget({this.title, this.count});

  @override
  Widget build(BuildContext context) {
    this.count = (count > 99) ? 99 : count;
    return Container(
      child: new Text("$title $count"),
    );
  }
}
```

这段代码看起来没有什么问题，也可以正常运行，但是在编译器上会有 *“This class is marked as '@immutable', but one or more of its instance fields are not final”* 的提示警告，这是因为 `TestWidget` 内的 `count` 成员变量没有加上 `final` 声明，从而在代码层面容易产生歧义。

> 因为前面说过 `Widget` 是 `immutable` ，所以它的每次变化都会导致自身被重新构建，也就是 `TestWidget` 内的 `count` 成员变量其实是不会被保存且二次使用。

如上所示代码中 `count` 成员没有 `final` 声明，所以理论是可以对 `count` 进行二次修改赋值，造成 `count` 成员好像被保存在 `TestWidget` 中被二次使用的错觉，容易产生歧义，比如某种情况下的 `widget.count` ，所以需要加这个 `final` 就可以看出来 `Widget` 的不可变逻辑。

如果把 `StatelessWidget` 换成 `StatefulWidget` ，然后把 `build` 方法放到 `State` 里， `State` 里的 `count` 就可以就可以实现跨帧保存。

```
class TestWidgetWithState extends StatefulWidget {
  final String title;

  TestWidgetWithState({this.title});

  @override
  _TestWidgetState createState() => _TestWidgetState();
}

class _TestWidgetState extends State<TestWidgetWithState> {
  int count;

  @override
  Widget build(BuildContext context) {
    this.count = (count > 99) ? 99 : count;
    return InkWell(
      onTap: () {
        setState(() {
          count++;
        });
      },
      child: Container(
        child: new Text("${widget.title} $count"),
      ),
    );
  }
}
```

所以这里最重要的是，首先要理解 **`Widget` 的不可变性质，然后知道了通过 `State` 就可以实现数据的跨 `Widget` 保存和恢复，那为什么 `State` 就可以呢?**

这就涉及到 Flutter 中另外一个很重要的知识点， `Widget` 的背后又是什么？事实上在 Flutter 中 Widget 并不是真正控件，**在 Flutter 的世界里，我们最常使用的 `Widget` 其实更像是配置文件，而在其后面的 `Element` 、 `RenderObject` 、 `Layer` 等才是实际“干活”的对象。**

> `Element` 、 `RenderObject` 、 `Layer` 才是需要学习理解的对象。

简单举个例子，如下代码所示，其中 `testUseAll` 这个 `Text` 在同一个页面下在三处地方被使用，并且代码可以正常运行渲染，如果是一个真正的 `View` ，是不能在一个页面下这样被多个地方加载使用的。

在 Flutter 设定里，`Widget` 是配置文件告诉 Flutter 你想要怎么渲染，`Widget` 在 Flutter 里会经过 `Element` 、`RenderObject` 、乃至 `Layer` 最终去进行渲染，所以作为配置文件的 `Widget` 可以是 `@immutable` ，可以每次状态更新都被重构。

所以回到最初说过的问题：**Flutter 的嵌套很恶心？是的 Flutter 设定上确实导致它会有嵌套的客观事实，但是当你把 `Widget` 理解成配置文件，你就可以更好地组织代码，比如 Flutter 里的 `Container` 就是一个抽象的配置模版。**

> 参考 `Container` 你就学会了 Flutter 组织代码逻辑的第一步。

同时因为 `Widget` 并不是真正干活的，所以嵌套事实上并不是嵌套 `View` ，一般情况下 `Widget` 的嵌套是不会带来什么性能问题，因为它不是正式干活的，嵌套不会带来严重的性能损失。

举个例子，当你写了一堆的 `Widget` 被加载时，第一次会对应产生出 `Element` ，之后 `Element` 持有了 `Widget` 和 `RenderObject` 。

简单的来说，一般情况下画面的改变，就是之后 `Widget` 的变化被更新到 `RenderObject` ，而在 Flutter 中能够跨帧保存的 `State` ，其实也是被 `Element` 所持有，从而可以用来跨 `Widget` 保存数据。

> 所以 `Widget` 的嵌套一般不会带来性能问题，每个 `Widget` 状态都代表了一帧，可以理解为这个“配置信息”代表了当前的一个画面，在 `Widget` 的背后，嵌套的 `Padding` 、 `Align` 这些控件，最后只是 `canvas` 时的一个“偏移计算”而已。

所以理解 `Widget` 控件很重要， `Widget` 不是真正的 `View` ，它只是配置信息，只有理解了这点，你才会发现 Flutter 更广阔的大陆，比如：

- Flutter 的控件是从 `Elemnt` 才开始是真正的工作对象；
- 要看一个 `Widget` 的界面效果是怎么实现，应该去看它对应的 `RenderObejcet` 是怎么绘制的；
- 要知道不同堆栈或者模块的页面为什么不会互相干扰，就去看它的 `Layer` 是什么逻辑；
- 是不是所有的 `Widget` 都有 `RenderObejcet` ？ `Widget` 、 `Elemnt` 、 `RenderObejcet` 、 `Layer` 的对应关系是什么？

这些内容才是学 Flutter 需要如理解和融汇贯通的，当你了解了关于 `Widget` 背后的这一套复杂的逻辑支撑后，你就会发现 Flutter 是那么的简单，在实现复杂控件上是那么地简单， `Canvas` 组合起来的能力是真的香。

当然具体展开这部分内容不是三言两语可以解释完，在我出版的**《Flutter开发实战详解》** 中第三章和第四章就着重讲解的内容，也是这出版本书主要的灵魂之处，这部分内容不会因为 Flutter 的版本迭代而过时的内容。

> 这算做了个小广告？？

Flutter 是个有坑的框架

最后讲讲 Flutter 的坑，事实上没有什么框架是没有坑的，如果框架完美得没有问题，那我们竞争力反而会越来越弱，可替换性会更高。

> 这也是为什么一开始 Andorid 和 iOS 开发很火热，而现在客户端开发招聘回归理性的原因，因为这个领域已经越来越成熟，自然就“卷”了。

事实上我一直觉得使用框架的我们并没有什么特殊价值，而解决使用框架所带来的问题才是我们特有的价值。

而 Flutter 的问题也不少，比如:

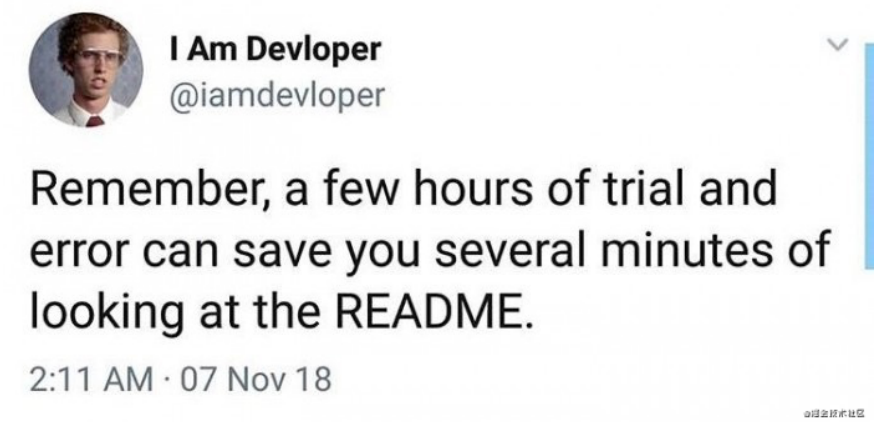
- `WebView` 的问题：Flutter 特有的 UI 机制，导致了 Flutter 需要通过特殊的方式来接入比如 `WebView` 、 `MapView` 这样的控件，而这部分也导致了接入后不断性能、键盘、输入框等的技术问题，具体可以参考： 《Hybrid Composition 深度解析》 和 《 Android PlatformView 和键盘问题》 。

- 图片处理和加载：**在图片处理和加载上 Flutter 的能力无疑是比较弱的**，同时对于单个大图片的加载和大量图片列表的显示处理上，**Flutter 很容易出现内存和部分 GPU 溢出的问题**。而这部分问题处理起来特别麻烦，如果需要借用原生平台来解决，则需要通过外界纹理的方式来完成，而这个实现的维护成本并不低。
- 混合开发是避免不了的话题：因为 Flutter 的控件和页面堆栈都脱离原生平台，所以混合开发的结果就会导致维护成本的提高，现在较多使用的 `flutter_boost` 和 `flutter_thrio` 都无法较好的真正解决混合开发中的痛点，所以对于 Flutter 来说这也是一大考验。

然而事实上在我收到关于 Flutter 的问题里，反而大部分和 Flutter 是没有关系的，比如：

- “ `flutter doctor` 运行之后卡住不动”
- “ `flutter run` 运行之后出现报错”
- “ `flutter pub get` 运行之后为什么提示 dart 版本不对”
- “运行后出现 Gradle 报错，显示 timeout 之类问题”
- “iOS 没办法运行到真机上”
- “xxx这样的控件有没有现成的”……

说实话，如果是这些问题，我觉得这并不是 Flutter 的问题，大部分时候是看 log 、看文档和网络的问题，甚至仅仅是搜索引擎检索技术的问题。。。。



虽然 Flutter 有着这样那样的问题，但是综合考虑下来，它对我来现阶段确实是最合适的 UI 框架。

最后

很久没写这么长的内容了，一般写这么长的内容能看完的人也不多，只是希望这篇文章能让你更全面地去理解 Flutter ，或者能帮你找到 Flutter 学习的方向，最后借用某位大佬说过的一句话：

> “能大规模商用的技术，都不需要太高的智商，否则这种技术就不可能规模化。某些程序员们，请停止你们的蜜汁自信。”

最近刚好有网友咨询一个问题，那就顺便借着这个问题给大家深入介绍下 Flutter 中键盘弹起时，`Scaffold` 的内部发生了什么变化，让大家更好理解 Flutter 中的输入键盘和 `Scaffold` 的关系。

如下图所示，当时的问题是：*当界面内有 `TextField` 输入框时，点击键盘弹起后，界面内底部的按键和 FloatButton 会被挤到键盘上面，有什么办法可以让底部按键和 FloatButton 不被顶上来吗？*

其实解决这个问题很简单，那就是只要**把 `Scaffold` 的 `resizeToAvoidBottomInset` 配置为 `false`** ，结果如下图所示，键盘弹起后底部按键和 FloatButton 不会再被顶上来，问题解决。**那为什么键盘弹起会和 `resizeToAvoidBottomInset` 有关系?**

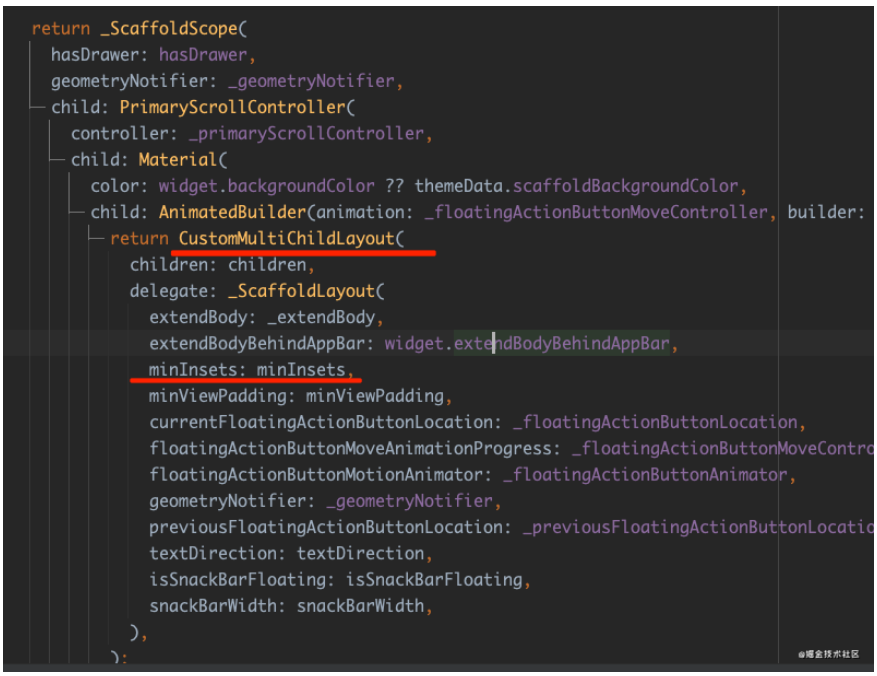
Scaffold 的 resize

`Scaffold` 是 Flutter 中最常用的页面脚手架，前面知道了通过 `resizeToAvoidBottomInset` ，我们可以配置在键盘弹起时页面的底部按键和 FloatButton 不会再被顶上来，其实这个行为是因为 `Scaffold` 的 `body` 大小被 `resize` 了。

那这个过程是怎么发生的呢？首先如下图所示，我们在 `Scaffold` 的源码里可以看到，当 `resizeToAvoidBottomInset` 为 true 时，会使用 `mediaQuery.viewInsets.bottom` 作为 `minInsets` 的参数，也就是可以确定：**键盘弹起时的界面 `resize` 和 `mediaQuery.viewInsets.bottom` 有关系。**

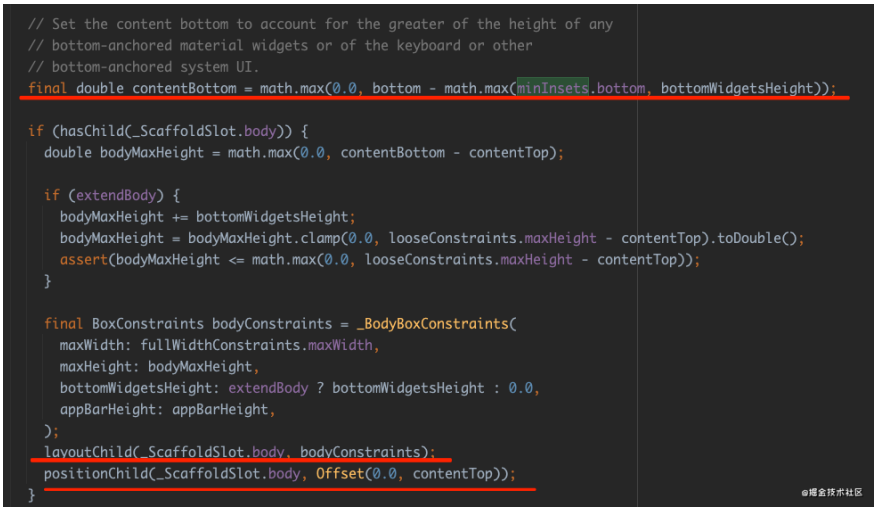
而如下图所示， `Scaffold` 内部的布局主要是靠 `CustomMultiChildLayout` ， `CustomMultiChildLayout` 的布局逻辑主要在 `MultiChildLayoutDelegate` 对象里。

前面获取到的 `minInsets` 会被用到 `_ScaffoldLayout` 这个 `MultiChildLayoutDelegate` 里面，也就是说 **`Scaffold` 的内部是通过 `CustomMultiChildLayout` 实现的布局，具体实现逻辑在 `_ScaffoldLayout` 这个 `Delegate` 里。**



> 关于 `CustomMultiChildLayout` 的详细使用介绍在之前的文章《详解自定义布局实战》里可以找到。

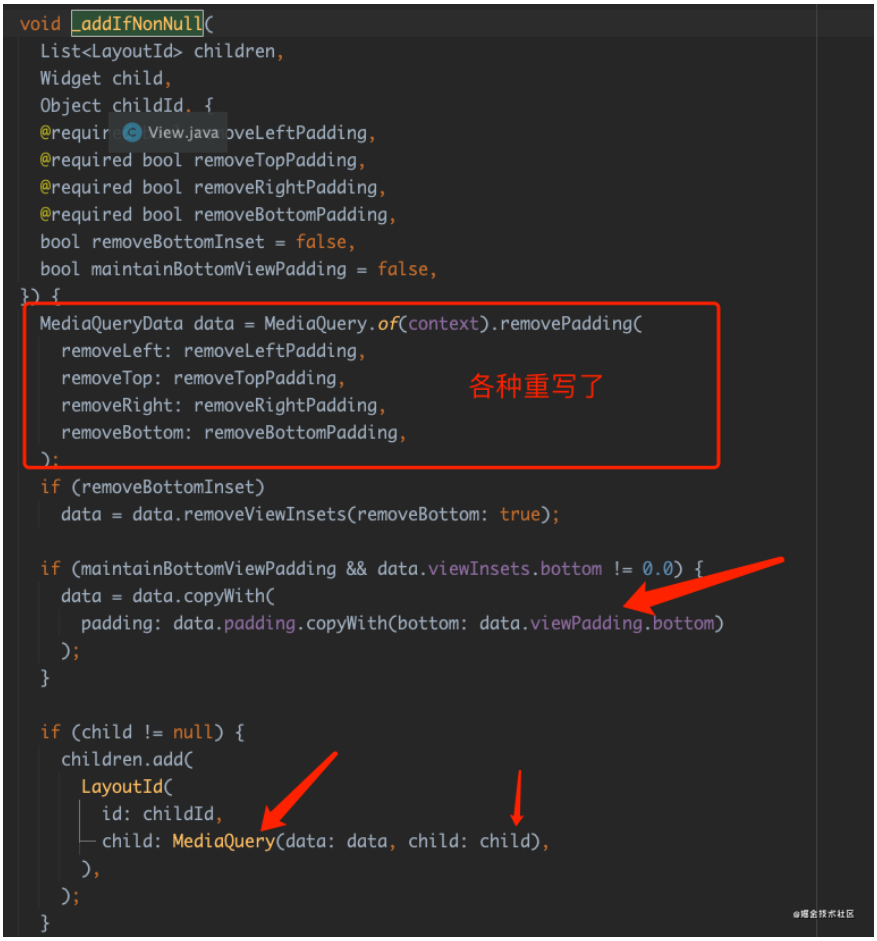
接着看 `_ScaffoldLayout` ， 在 `_ScaffoldLayout` 进行布局时，会通过传入的 `minInsets` 来决定 `body` 显示的 `contentBottom` ， 所以可以看到**事实上传入的 `minInsets` 改变的是 `Scaffold` 布局的 bottom 位置。**



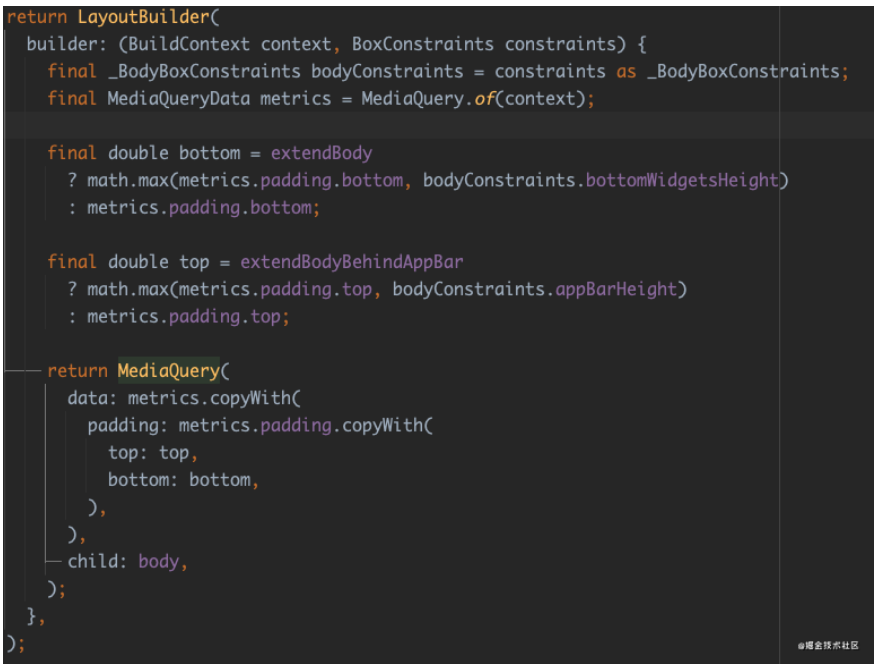
> 上图代码中使用的 `_ScaffoldSlot.body` 这个枚举其实是作为 `LayoutId` 的值，`MultiChildLayoutDelegate` 在布局时可以通过 `LayoutId` 获取到对应 child 进行布局操作，详细可见：《详解自定义布局实战》

那么 `Scaffold` 的 `body` 是什么呢？如上图代码所示，其实 `Scaffold` 的 `body` 是一个叫 `_BodyBuilder` 的对象，而这个 `_BodyBuilder` 内部其实是一个 `LayoutBuilder`。（注意，在 `widget.appbar` 不为 `null` 时，会 `removeTopPadding`）

所以如下图代码所示 `body` 在添加时，**它父级的 `MediaQueryData` 会被重载，特别是 `removeTopPadding` 会被清空，`viewInsets.bottom` 也是会被重置。**



最后如下代码所示，`_BodyBuilder` 的 `LayoutBuilder` 里会获取到一个 `top` 和 `bottom` 的参数，这两个参数都通过前面在 `_ScaffoldLayout` 布局时传入的 `constraints` 去判断得到，最终 `copyWith` 得到新的 `MediaQuery` 。



这里就涉及到一个有意思的点，在 `_BodyBuilder` 里的通过 `copyWith` 得到新的 `MediaQuery` 会影响什么呢？如下代码所示，这里用一个简单的例子来解释下。

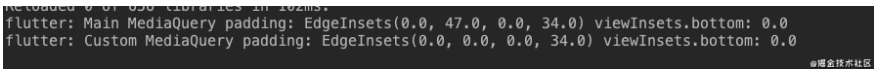
```
class MainWidget extends StatelessWidget {
  final TextEditingController controller =
      new TextEditingController(text: "init Text");
  @override
  Widget build(BuildContext context) {
    print("Main MediaQuery padding: ${MediaQuery.of(contex
    return Scaffold(
      appBar: AppBar(
        title: new Text("MainWidget"),
      ),
      extendBody: true,
      body: Column(
        children: [
          new Expanded(child: InkWell(onTap: (){
            FocusScope.of(context).requestFocus(FocusNode(
          })),
          ///增加 CustomWidget
          CustomWidget(),
          new Container(
            margin: EdgeInsets.all(10),
            child: new Center(
              child: new TextField(
                controller: controller,
              ),
            ),
          ),
          new Spacer(),
        ],
      ),
    );
  }
}
class CustomWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    print("Custom MediaQuery padding: ${MediaQuery.of(conte
    return Container();
  }
}
```

如上代码所示：

- 代码中定义了 `MainWidget` 和 `CustomWidget` 两个控件；

- `MainWidget` 里使用了 `Scaffold` ，并且 `CustomWidget` 在 `MainWidget` 里被使用；
- 分别在这两个 Widget 的 `build` 方法里打印出对应的 `MediaQuery.of(context).padding` 和 `MediaQuery.of(context).viewInsets.bottom` 的值；

如下图所示，在键盘弹起和不弹起时可以看到 `padding` 值是不同的，而 `viewInsets.bottom` 都为 0。



为什么 `padding` 值的 `top` 会不一致，**自然是因为 `CustomWidget` 和 `MainWidget` 获取到的 `MediaQuery.of(context)` 对象不是同一个数据。**

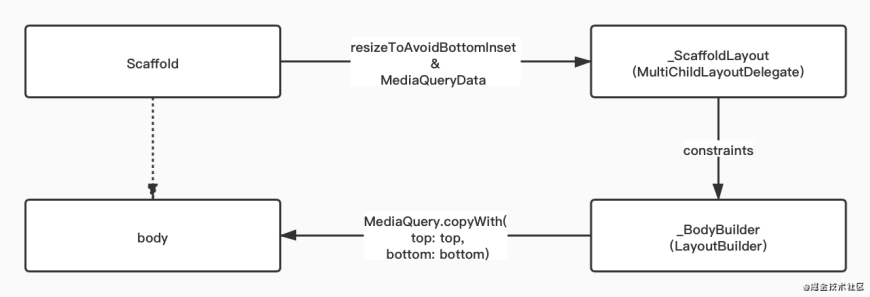
- `MainWidget` 使用的 `MediaQuery.of(context)` 得到的 `MediaQueryData` 是上级往下传递的，里面**包含了 `top: 47` 的状态栏高度和 `bottom: 34` 的底部安全区域高度**。
- `CustomWidget` 里面 `MediaQuery.of(context)` 得到的 `MediaQueryData` ，自然就是前面分析过的 `_BodyBuilder` 里的通过 `copyWith` 得到新的 `MediaQuery` ，所以 `CustomWidget` 得到的 `MediaQueryData` 其实**在 `Scaffold` 内部已经被重置了，所以它的 `top: 0` ，获取不到状态栏高度**。

> 事实上这就是大家为什么有时候 **`MediaQuery.of( context)` 可以获取到状态栏高度，有时候又获取不到的原因**，因为你的 `context` 获取到的是 `Scaffold` 之外的 `MediaQueryData` ，还是 `Scaffold` 内被重载过的 `MediaQueryData` ，自然会得到不一样的结果。

如下图所示，键盘弹起因为被 resize 了，所以界面的 `bottom` 安全区域变成了 0 ，而

- 在 `MainWidget` 中可以获取到 `viewInsets.bottom` 也就是键盘的高度；
- 在 `CustomWidget` 获取不到 `viewInsets.bottom` ，因为在 `Scaffold` 内被重载清除了。

总结一下：**`Scaffold` 的 `resizeToAvoidBottomInset` 会通过 `MediaQueryData` 影响 body 的布局，同时在 `Scaffold` 内 `MediaQuery` 会被重载，所以使用的 `context` 位置不同，获取到的 `MediaQueryData` 也不同，如果需要获取键盘高度和状态栏高度的话，最好使用 `Scaffold` 外的 `context` 。**



> 这里讲了 `MediaQuery` 和 `MediaQueryData` 的内容，为什么 `MediaQuery` 通过嵌套就可以重载？为什么通过 `context` 可以往上获取到离 `context` 最近的 `MediaQueryData`？因为 `MediaQuery` 是一个 `InheritedWidget`：《全面理解State》。

键盘如何影响 Scaffold

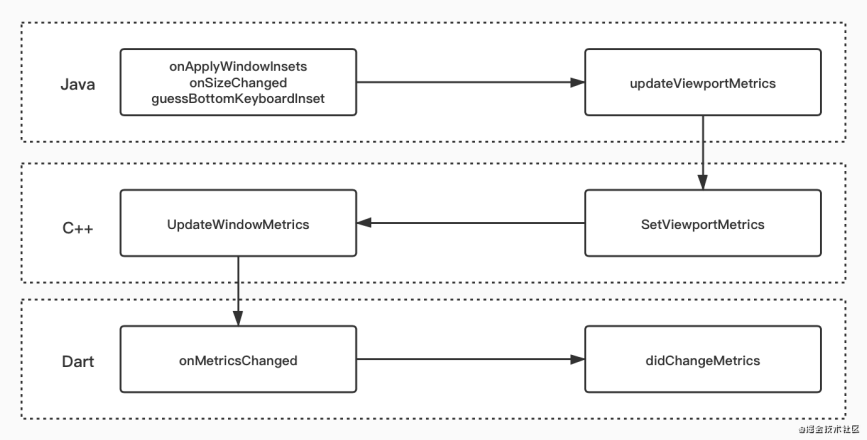
前面我们聊了 `Scaffold` 的 `resizeToAvoidBottomInset` 会通过 `MediaQueryData` 影响 body 的布局，那是怎么影响的呢？

事实上这得从 `MaterialApp` 说起，在 `MaterialApp` 内部的深处嵌套着一个叫 `_MediaQueryFromWindow` 的 Widget，它在内部通过 `WidgetsBinding.instance.addObserver` 对 App 的各种系统事件做了监听，并且对应都执行了 `setState`。

所以如下源码所示，当键盘弹出时，`build` 方法会被执行，而 `MediaQueryData` 就会通过 `MediaQueryData.fromWindow` 获取到新的 `MediaQueryData` 数据。

```
@override
 void initState() {
   super.initState();
   WidgetsBinding.instance.addObserver(this);
 }

 // ACCESSIBILITY

 @override
 void didChangeAccessibilityFeatures() {
   setState(() { });
 }

 // METRICS

 @override
 void didChangeMetrics() {
   setState(() {});
 }

 @override
 void didChangeTextScaleFactor() {
   setState(() { });
 }

 // RENDERING
 @override
 void didChangePlatformBrightness() {
   setState(() {});
 }

 @override
 Widget build(BuildContext context) {
   MediaQueryData data = MediaQueryData.fromWindow(Widgets
   if (! kReleaseMode) {
     data = data.copyWith(platformBrightness: debugBrightn
   }
   return MediaQuery(
     data: data,
     child: widget.child,
   );
 }

 @override
 void dispose() {
   WidgetsBinding.instance.removeObserver(this);
   super.dispose();
 }
```

举个例子，如下图所示，从 Android 的 Java 层弹出键盘开始，会把改变后的视图信息传递给 C++ 层，最后回调到 Dart 层，从而触发 `MaterialApp` 内的 `didChangeMetrics` 方法执行 `setState(() {});` ，进而让 `_MediaQueryFromWindow` 内的 `build` 更新了 `MediaQueryData` ，最终改变了 `Scaffod` 的 `body` 大小。



那么到这里，你知道如何在 Flutter 里正确地去获取键盘的高度了吧?

最后

从一个简单的 `resizeToAvoidBottomInset` 去拓展到 `Scaffod` 的内部布局和 `MediaQueryData` 与键盘的关系，其实这也是学习框架过程中很好的知识延伸，通过特定的问题去深入理解框架的实现原理，最后再把知识点和问题关联起来，这样问题在此之后便不再是问题，因为入脑了～

多余的前言

Flutter 2.0 发布时，其中最受大家关注之一的内容就是 `Add-to-App` 相关的更新，因为除了**热更新**之外，Flutter 最受大家诟病的就是**混合开发体验**不好。

> 为什么不好呢？因为 **Flutter 的控件渲染直接脱离了原生平台，也就是无论页面堆栈和渲染树都独立于平台运行**，这固然给 Flutter 带来了较好的跨平台体验，但是也造成了在和原生平台混合时存在高成本的问题。

且不说在已有的原生项目中集成 Flutter ，就是现阶段在 Flutter 中集成原生控件的 PlatformView 和 Hybrid Composition 体验也是有待提升，当然“有支持”和“能用”就已经是很不错的进展。

所以 Flutter 2.0 在千呼万唤中发布了 `FlutterEngineGroup` 用于支持官方的 `Add Flutter to existing app` 方案。

在此方案出现之前，类似的第三方支持有 `flutter_boost` 、`mix_stack` 、 `flutter_thrio` 等等 ，它们是否好用这里不讨论，但是这些方案都要面对的问题是：

> 非官方的支持必然存在每个版本需要适配的问题，而按照 Flutter 目前的 `issue closed` 和 `pr merge` 的速度，很可能每个季度的版本都存在较大的变动，**所以如果开发者不维护或者维护不及时，那么侵入性极强的这类框架很容易就成为项目的瓶颈。**

而官方提供的 `FlutterEngineGroup` 方案有没有缺陷？肯定有的，它目前看起来更像是被催生出来的状态，各方面的问题还是有的，比如某些地方还存在不能 `destroy` 的问题。（当然这个问题以及在 `master` 分支 merge 了）

```
/**
 * This tears down the messaging connections on the platform channel and the DataModel.
 */
fun detach() {
    // TODO: Uncomment after https://github.com/flutter/engine/pull/24644 is on stable.
    // engine.destroy();
    DataModel.instance.removeObserver( observer: this)
    channel.setMethodCallHandler(null)
}
```


但是官方提供的方案，就意味着这个设计得到了 Flutter 官方的保证，**在未来的版本中会有兼容的优势。**

`FlutterEngineGroup` 方案使用了多 Engine 混合模式，官方宣称除了一个 Engine 对象之外，后续每个 Engine 对象在 Android 和 iOS 上仅占用 180kB 。

> 以前的方案每多一个Engine ， 可能就会多出 19MB Android 和 13MB iOS 的占用。

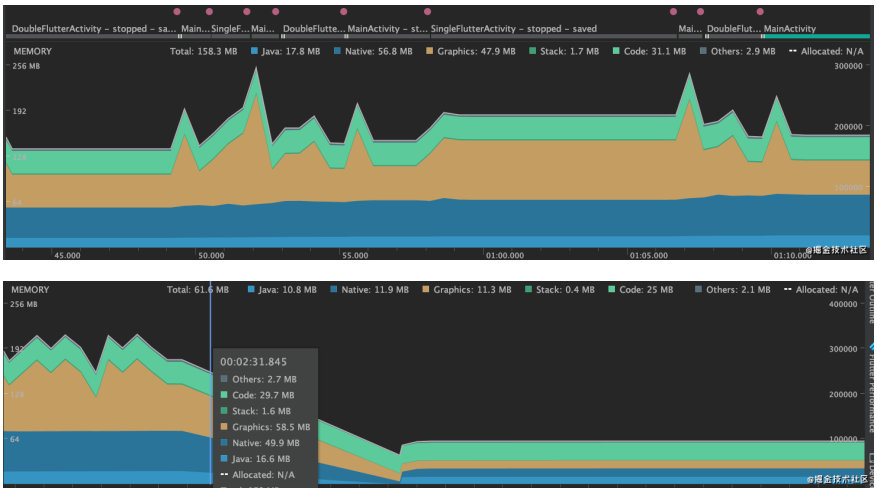
从 Flutter 官方提供的例子上看，`FlutterEngineGroup` 的 API 十分简单，**多个 Engine 实例的内部都是独立维护自己的内部导航堆栈**，所以可以做到每个 Engine 对应一个独立的模块。

所以使用 `FlutterEngineGroup` 之后，`FlutterEngine` 都将由 `FlutterEngineGroup` 去生成，生成的 `FlutterEngine` 可以独立应用于 `FlutterActivity` / `FlutterViewController`，甚至是 `FlutterFragment`：

> 所以就像例子上所示，你可以在一个 `Activity` 上显示两个独立的 FlutterView 。

这其实得益于通过 `FlutterEngineGroup` 生成的 `FlutterEngine` 可以**共享 GPU 上下文， font metrics 和 isolate group snapshot**，从而实现了更快的初始速度和更低的内存占用。

> **下图是使用官方实例打开16个页面之后的内存使用情况，并且每个页面成功返回且没有出现黑屏。**



简单的使用介绍

使用 `FlutterEngineGroup` 首先需要创建一个 `FlutterEngineGroup` 单例对象，之后每当需要创建 Engine 时，就通过它的 `createAndRunEngine(activity, dartEntrypoint)` 来创建对应的 `FlutterEngine` 。

```
        val app = activity.applicationContext as App
        // This has to be lazy to avoid creation before the
        val dartEntrypoint =
            DartExecutor.DartEntrypoint(
                FlutterInjector.instance().flutterLoader().
            )
        engine = app.engines.createAndRunEngine(activity, c
        this.delegate = delegate
        channel = MethodChannel(engine.dartExecutor.binaryM
```

以官方 Demo 的这段代码为例子：

1、首先通过 `findAppBundlePath` 和 `entrypoint` 创建出 `DartEntrypoint` 对象，这里的 **`findAppBundlePath` 主要就是默认的 `flutter_assets` 目录**；而 **`entrypoint` 其实就是 dart 代码里启动方法的名称**；也就是绑定了在 dart 中 `runApp` 的方法。

```
///kotlin
app.engines.createAndRunEngine(pathToBundle, "topMain")


///dart
@pragma('vm:entry-point')
void topMain() => runApp(MyApp());
```

2、通过上面创建的 `dartEntrypoint` 和 `context` ，使用 `FlutterEngineGroup` 就可以创建出对应的 `FlutterEngine` ，其实在内部就是通过 `FlutterJNI.nativeSpawn` 和原有的引擎交互，得到新的 Long 地址 id。

> 在 C++ 层类似于原有的 `RunBundleAndSnapshotFromLibrary` 方法，但是它不能更改包路径或者 asset，所以只能加载同一份 AOT 文件，这里得到的指针地址就是一个新的 `AndroidShellHolder` 。

3、最后利用生成的 `FlutterEngine` 的 `binaryMessenger` 来得到一个 `MethodChannel` 用于原生和 dart 之间的通信。

通过上述流程得到的 Engine ，自然就可以直接用于渲染运行新的 Flutter UI，比如直接继承 `FlutterActivity` ，然后 override `provideFlutterEngine` 方法返回得到的 Engine 。

```
class SingleFlutterActivity : FlutterActivity()

    .......

    override fun provideFlutterEngine(context: Context): F
        return engine
    }


}
```

是不是很简单？这么简单的接入后：

- 在 dart 层面可以通过 `MethodChannel` 打开原始页面；
- 在原生层可以通过新建 `FlutterEngine` 打开新的 Flutter 页面；
- 甚至你还可以在原生层打开一个 `FlutterView` 的 Dialog；

当然，到这里你可能已经注意到了，因为每个 Flutter 页面都是一个独立的 Engine ，由于 dart isolate 的设计理念，**每个独立 Engine 的 Flutter 页面内存是无法共享的**。

也就是说，当你需要共享数据时，只能在原生层持有数据，然后注入或者传递到每个 Flutter 页面中，就像官方所说的，**每个 Flutter 页面更像是一个独立 Flutter 模块**。

> 当然这也造成了一些不必要的麻烦，比如：**同一张图片，在原生层、不同 Flutter Engine 会出现多次加载的问题**，这种问题可能就需要你针对 Flutter 的图片加载使用外界纹理，来实现在原生层统一的内存管理等。

另外目前我发现问题还有： Android 11 上的 ARM TBI 问题 ，不过通过这次尝试，相信 `FlutterEngineGroup` 的进展将会越来越明朗，更早的被应用到生产环境中。

在以前的《Android 和 iOS 打包提交审核指南》里介绍了 Flutter 下打包 Android 和 iOS 的指南，不过这部分内容主要介绍的是如何在本地打包发布流程。

但事实上一般的产品发布流程，都会有专门的机器用于打包服务，**在统一干净的环境下进行打包更有利于发布的管理，避免各种本地环境差异问题**。

当然大多数时候可以直接使用第三方的 CI 服务，但是专门支持 Flutter 的第三方服务并不多，并且**自己动手还免费**，所以本篇主要介绍自己搭建独立打包服务的过程。

> 由于 Android 的命令打包服务比较简单，这里主要介绍配置搭建 iOS 下的 Flutter 打包和发布 CI，**其实主要也是 iOS 的 CI**。

一、参数支持

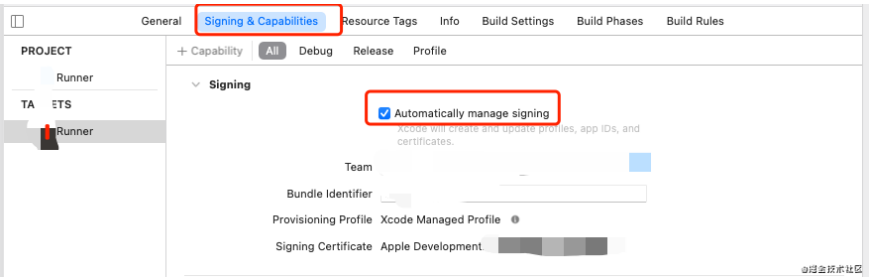
首先在 iOS 上很多的配置信息都是写在 `info.plist` 文件，所以一开始需要解决打包时支持动态修改 `info.plist` 的参数，这样有利于我们在输出不同环境的包配置，如：QA、Release、Dev 等等。

```
/usr/libexec/PlistBuddy -c "Set  CFBundleVersion ${CFBundle

/usr/libexec/PlistBuddy -c "Print  CFBundleVersion " ./Runn
```

在 Mac 上其实本身就自带了满足需求的命令行工具：`PlistBuddy`，如上命令所示

- 通过 `Set` 命令可以直接动态配置 `plist` 下的版本号、code 和第三方 App Id 等相关配置；
- 通过 `Print` 命令直接输出对应的 `plist` 信息；

完成 `plist` 配置的支持，接下来就需要在机器上配置**开发者信息**，最简单的做法就是打开 Xcode 然后直接登陆上开发者账号，通过账号直接让 Xcode 的 `Automatically manage signing` 帮助我们完成整个开发信息的配置过程。



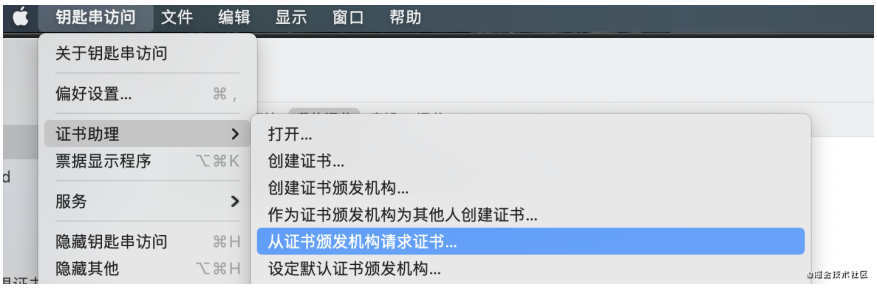
但是我个人不推荐这种方式，**打包机器本身可能会涉及多个项目组使用，都把自己的开发账号登陆在一个公用机器上存在风险，而且多个账号同时登陆容易混乱，最后直接登陆也不利于证书和描述和管理。**

所以要实现一个较为安全和通用的服务，这里比较推荐：**通过在机器上配置证书和 mobile provision 等文件的方式来完成打包认证。**

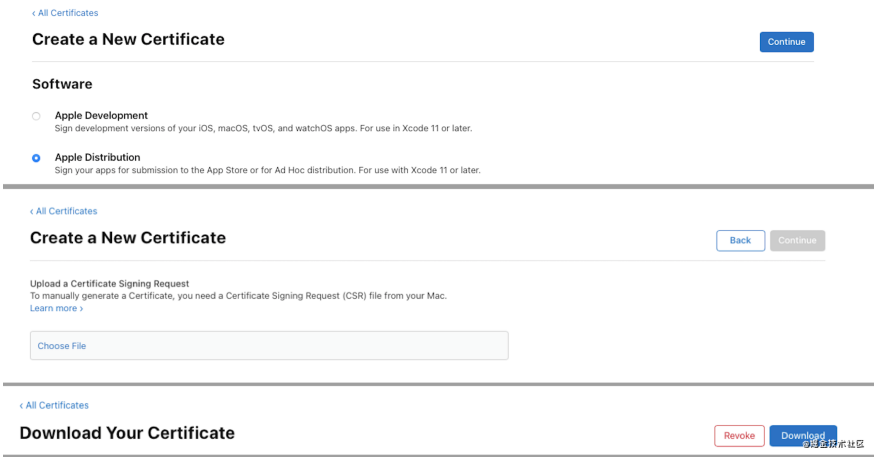
二、手动配置证书

手动配置证书和 mobile provision 会比较麻烦，但是它可以让服务更加通用，也让你更熟悉 iOS 打包的流程。

1、首先通过本地钥匙串创建 `CertificateSigningRequest.certSigningRequest` 文件，如图所示自动生成就可以了。



2、在苹果官方的 developer 上点击创建证书，上传步骤 1 中的 `CertificateSigningRequest.certSigningRequest` 文件，然后下载 `.cer` 证书文件。



3、这里需要注意**不能直接把这个 `.cer` 证书文件安装到打包服务上，而是把这个 `.cer` 先安装到上面第 1 步中生成的 `CertificateSigningRequest.certSigningRequest` 的机器上，然后通过导出证书生成带有密码的 `p12` 证书文件**，这个文件才是可以安装到打包机器上的证书文件。

4、安装证书，把 `p12` 文件放置到打包服务上，然后点击证书，输入 3 中创建时输入的密码，安装到钥匙串的 **“登陆”**，这时候就可以看到钥匙串证书里带有 **TeamId 的 Apple Distribution 证书**。

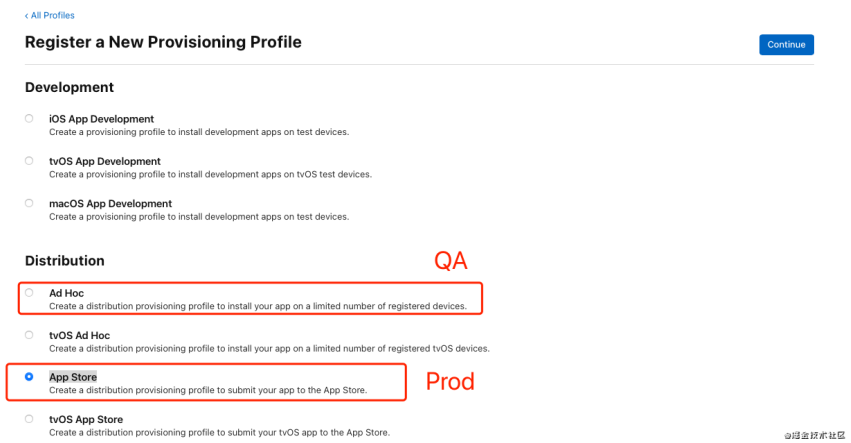
5、需要额外注意**安装后可能会看到说“证书不受信任”的提示，这可能是因为机器上缺少 AppleWWDRCA** (Apple Worldwide Developer Relations Certification Authority)证书，可以通过下面的地址进行安装解决：

- https://developer.apple.com/cn/support/code-signing/
- https://developer.apple.com/support/expiration/

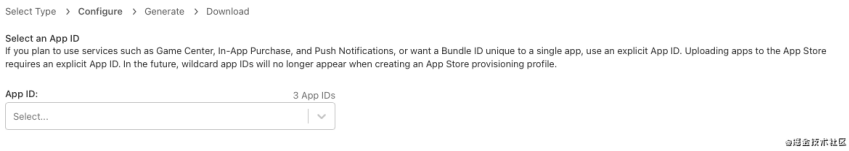
三、配置描述文件

配置完证书后就是配置描述文件，在苹果开发者网站的 Profiles 创建对应的 mobile provision 。

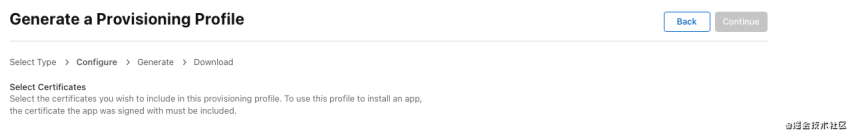
1、选择 `Distribution` - `App Store` 创建对应的打包模式，如果是 QA 的话一般选择 Ad Hoc，也就是需要文件绑定设备 UDID，而不需要上架 Store 的模式。



2、选择需要支持的 App Id ， 也就是 bundle Id 。



3、选择前面生成的 `Distribution` 证书 ， 这里主要一定要选择同意同一个。



4、最后输入 Provisioning Profile Name ， 这个 Name 在后面会有作用，另外如果是 `Ad Hoc` 的话，在这一步可以选择已经添加的 Devices 的 UDID 。

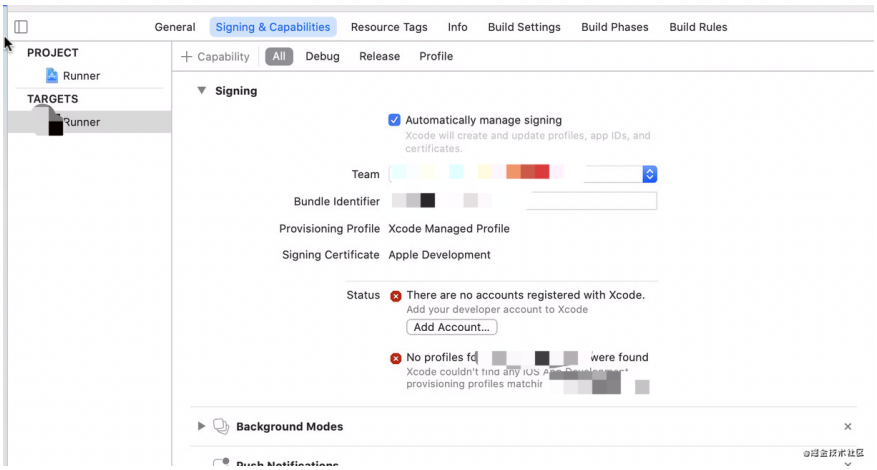
5、完成配置后下载这个 `mobile provision` 文件，将它放到打包机器上的 `/Users/你的账号/Library/MobileDevice/Provisioning Profiles` 目录下，后面会需要用到它。

> 如果是 store 版本的就选择 `Distribution - App Store` ， 如果是 QA 版本的就选择 `Distribution - Ad Hoc` ， 因为 `App Store` 打出来的包只能通过 Store 或者官方 TestFight 下载，而 `Ad Hoc` 打包的可以通过内部自定义分发下载（通过添加测试设备的 UDID）。

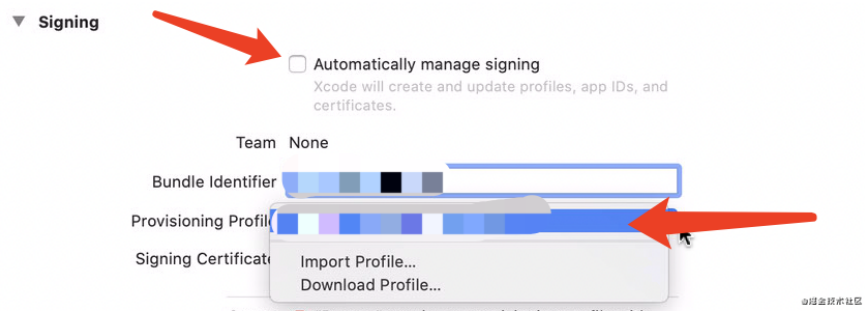
四、配置项目

完成了证书和描述文件的配置后，接下来就是针对项目的配置。

首先将需要打包的项目 clone 到打包机器上（只是为了做测试配置），然后打开项目 `ios/Runner.xcworkspace` 目录，这时候可以看到项目因为没有开发者账号，是如下图所示的状态：



然后我们取消选购 `Automatically manage signing` ， 然后选中我们前面放置的描述文件，就可以看到 Xcode 会自动匹配到钥匙串里的证书，然后显示正常的证书和描述文件配置了。



这里有一个需要注意的点，那就是项目在我们本地开发默认使用的就是 `Automatically manage signing` 的方式，因为这样比较方便，所以我们其实是需要**在打包时让它变成手动签名，并且指定 mobile provision 文件的模式**。

所以前面在打包机器上操作 Xcode 取消 `Automatically manage signing` 指定描述文件后，其实已经修改了项目的 `ios/Runner.xcodeproj/project.pbxproj` ，所以这时候你只需要通过 `git diff` 命令就可以导出一个 `patch` 文件，这样在项目被 clone 下来后，通过 `git apply` 直接调整项目的描述文件。

```
git diff >./release.patch
```

如果有多种编译模式，比如一个项目打包多个 bundleId 和描述文件（QA 、Release）， 那就可以生成多个 `.patch` 文件。

> ⚠️ 注意：**第三方打包机器上每次打包都是 `clone` 一个新项目，打包后删除该项目**，这样可以保证每次打包的独立和干净，而通过改生成不同的 `.patch` 文件，我们可以指向不同的 `mobile provision` ，从而加载不同的证书，甚至是同一个项目打包出不同的 `bundle id` 。

五、开始打包

1、开发打包之前，需要先执行 **`security unlock-keychain -p xxxxx`** ，解锁下 keychain ， 这里的 xxxxx 就是你 Mac 上的密码。

2、通过 `flutter build ios --release` 打包出 release 模式的 `App.framework` 和 `Flutter.framework` 。

3、通过 `xcodebuild` 命令，如下开始编译 iOS 代码了，其中 $PWD 是所在工作目录：

```
xcodebuild -workspace Runner.xcworkspace -scheme Runner -sd
```

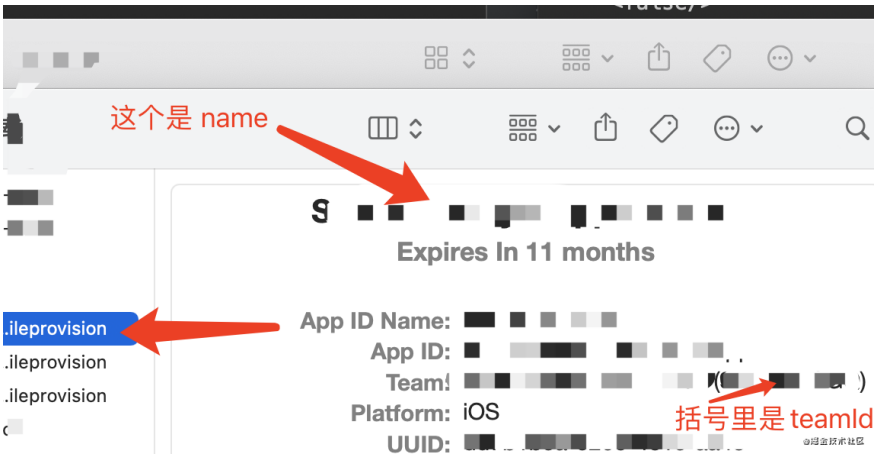
> ⚠️这里有一个需要注意，那就是**打包过程中如果出现 .sh 脚本的相关报错**，比如 `xcode_backend.sh" embed_and_thin` 或者 `PhaseScriptExecution Thin\ Binary /Users/xxxxx/Library/Developer/Xcode/DerivedData/` 的错误，推荐先在打包机上用 Xcode 执行一次完整的 `Archive` 流程，在首次执行过程应该会出现关于某些 sh 的授权执行弹框，输入密码点始终完成，然后再重新执行上述脚本。

4、执行完 `Archive` 之后，就可以进入 `export` 阶段，`exportArchive` 之前需要先准备一个 `ExportOptions.plist` 文件用户指定到处的配置，模板类似：

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http:
<plist version="1.0">
<dict>
    <key>destination</key>
    <string>export</string>
    <key>method</key>
    <string>app-store</string>
    <key>provisioningProfiles</key>
    <dict>
        <key>你的 bundleId </key>
        <string>前面 provision 定义的 name</string>
    </dict>
    <key>signingCertificate</key>
    <string>Apple Distribution</string>
    <key>signingStyle</key>
    <string>manual</string>

    <key>stripSwiftSymbols</key>
    <true/>
    <key>teamID</key>
    <string>你的开发证书的 Team Id</string>
    <key>uploadBitcode</key>
    <false/>
    <key>uploadSymbols</key>
    <false/>
</dict>
</plist>
```

其中

- `method` 的数值如果是 store 就写 `app-store` ，如果是 QA 就写 `ad-hoc` ；
- `provisioningProfiles` 的 `<dict>` 需要 `bundleId` 和前面 `provision` 定义的 `name` ；
- `teamID` 需要的是你的开发证书的 `Team Id` ；
- 如果是 store 可以增加 `uploadBitcode` 和 `uploadSymbols` 的配置，如果是 QA 则可以不指定，然后 QA 可以也指定 `thinning` 模式；



接着通过指定命令 `exportArchive` ，指定 ExportOptions.plist ，如果是有不同 id 或者不同模式，一般需要配置 QA 和 Prod 两种 `ExportOptions.plist` ，最终输出到 `package_path` 这时候你就得到了一个 ipa 文件。

```
xcodebuild -exportArchive -exportOptionsPlist ExportOptions
```

最后如果是 store 模式的，接下来你只需要通过 Mac 的 `Transporter` 将 ipa 上传到 App Store Connect，或者使用命令行工具将自己的应用或内容上传至 App Store Connect 。

```
$ xcrun altool --validate-app -f file -t platform -u userna
$ xcrun altool --upload-app -f file -t platform -u username
```

> 一般 altool 位于 /Applications/Xcode.app/Contents/Developer/usr/bin/altool ，更多可见 https://help.apple.com/asc/appsaltool/

如果你是 QA 模式，那么你需要先准备一个 `html` 文件，如下所示例子，通过 `a` 标签配置 `itms-service` 指定一个 `DistributionSummary.plist` 文件。

```
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
<a href=itms-services://?action=download-manifest&url=https
</body>
</html>
```

然后在 `DistributionSummary.plist` 文件中指定 `software-package` 的 ipa 下载地址，这样就可以完成 QA 的内部自助分发了。（*只能安装 QA provision 里已经配置了 UDID 那些机器*）

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http:
<plist version="1.0">
<dict>
    <key>items</key>
    <array>
        <dict>
            <key>assets</key>
            <array>
                <dict>
                    <key>kind</key>
                    <string>software-package</string>
                    <key>url</key>
                    <string>https://xxxx.xxxxxx.cn/xxxx/Ru
                </dict>
                <dict>
                    <key>kind</key>
                    <string>full-size-image</string>
                    <key>needs-shine</key>
                    <true/>
                    <key>url</key>
                    <string>http://xxxx.xxxxxx.cn/assets/ap
                </dict>
                <dict>
                    <key>kind</key>
                    <string>display-image</string>
                    <key>needs-shine</key>
                    <true/>
                    <key>url</key>
                    <string>http://xxxx.xxxxxx.cn/assets/ap
                </dict>
            </array>
            <key>metadata</key>
            <dict>
                <key>bundle-identifier</key>
                <string>com.xxxx.demo</string>
                <key>bundle-version</key>
                <string>1.0.0</string>
                <key>kind</key>
                <string>software</string>
                <key>title</key>
                <string>XXXX App download</string>
            </dict>
        </dict>
    </array>
</dict>
</plist>
```

六、多 Flutter 版本环境

如果需求有存在多个项目需要在一个机器打包，但是不同项目的 Flutter 等版本都不同，那么**对于 Mac 可以开启多个不同的登陆用户，这样就可以得到不同的打包环境**，当然这里主要注意的是 CocoaPod 的版本问题，因为比如：

- Flutter 1.22 版本默认是使用 1.8.0 之类的 Pod 版本，如果在 Flutter 1.22 上使用 1.10.0 的 Pod 版本会导致 logo 错误等问题；
- Flutter 2.0 需要的是 1.10.0 的 Pod 版本；

而**在 Mac 上默认 CocoaPod 是安装在 `usr/local/bin` 目录**，这个目录其实是多账号共享，所以为了解决这个问题，需要在每个账户环境下安装 `rvm` ，用于管理独立的 CocoaPod 版本。

简单地说：

- 1、先通过 curl 安装 rvm；

```
curl -L get.rvm.io |  bash -s stable && source ~/.rvm/script
```

- 2、通过 `rvm install 2.5.5` 安装对应的 ruby 版本，具体可以通过 `rvm list known` 选中你想要需要的版本

> 这里需要注意 `rvm install` 可能会失败，一般和 brew 需要 update 还有网络情况有关系；

- 3、可以安装多个 ruby 版本，然后通过 `rvm use <Version> --default` 或者 `rvm use <Version>` 来使用具体版本

> 不加 `defalut` 的话，下次启动命令行会变成原来的 `defalut` 版本；

- 4、在当前 ruby 版本下安装想要的 cocoapods 版本，这样当使用 `rvm use` 切换版本时，cocoapods 版本也会跟着切换。

```
sudo gem install cocoapods -v <Version> -n /usr/local/bin
```

事实上在不同用户下安装了 rvm 之后，彼此之间的 Pod 版本就已经分割开了。

七、最后

说了那么多，其实 Xcode 自动打包确实舒服很多，但是通过整个配置过程，也可以帮助你了解到以前不知道的打包和认证过程。

这里最后额外补充一句，通过如下命令，在打包 Android 或者 iOS 时，可以通过 `--dart-define` 来指定不同的 dart 参数.

```
flutter build ios --release --dart-define=CHANNEL=GSY --dar
```

在 dart 代码里可以通过 `String.fromEnvironment` 获取到对应的自定义配置参数。

```
const CHANNEL = String.fromEnvironment('CHANNEL');
const LANGUAGE = String.fromEnvironment('LANGUAGE');
```

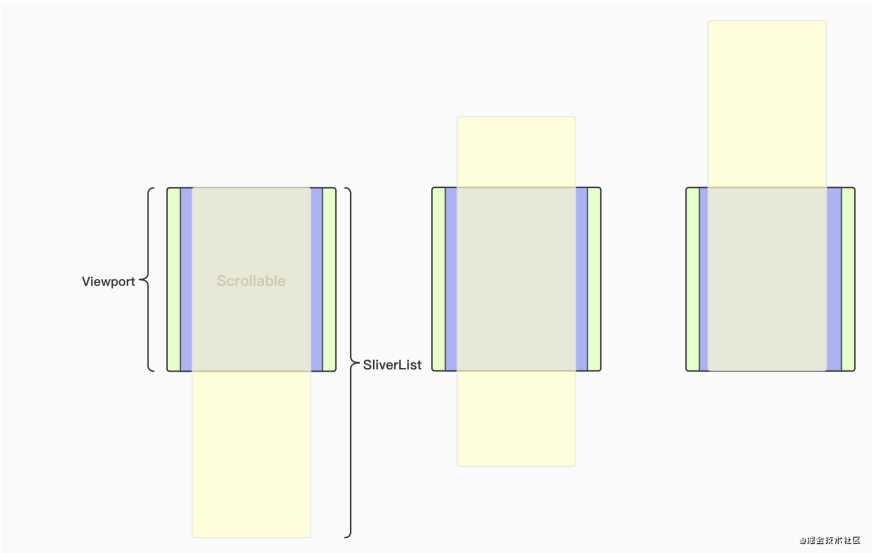
本篇主要帮助剖析理解 Flutter 里的列表和滑动的组成，用比较通俗易懂的方式，从常见的 `ListView` 到 `NestedScrollView` 的内部实现，帮助你更好理解和运用 Flutter 里的滑动列表。

> 本篇不是教你如何使用 API，而是一些日常开发中不常接触，但是很重要的内容。

Flutter 滑动列表

在 Flutter 里我们常见的滑动列表场景，简单地说其实是由三部分组成：

- `Viewport` ： 它是一个 *MultiChildRenderObjectWidget* 的控件，**它提供的是一个“视窗”的作用，也就是列表所在的可视区域大小；**
- `Scrollable` ：**它主要通过对手势的处理来实现滑动效果**，比如 *VerticalDragGestureRecognizer* 和 *HorizontalDragGestureRecognizer；*
- `Sliver` ： 准确来说应该是 *RenderSliver*， **它主要是用于在 Viewport 里面布局和渲染内容；**



以 `ListView` 为例，如上图所示是 `ListView` 滑动过程的变化，其中：

- 绿色的 `Viewport` 就是我们看到的列表窗口大小；
- 紫色部分就是处理手势的 `Scrollable`，让黄色部分 `SliverList` 在 `Viewport` 里产生滑动；
- 黄色的部分就是 `SliverList` ， 当我们滑动时其实就是它在 `Viewport` 里的位置发生了变化；

了解完这个基础理念后，就可以知道一般情况下 `Viewport` 和 `Scrollable` 的实现都是很通用的，所以一般在 **Flutter 里要实现不同的滑动列表，就是通过自定义和组合不同的 `Sliver` 来完成布局。**

> **准确说是完成 `RenderSliver` 的 `performLayout` 过程，通过 `SliverConstraints` 来得到对应的 `SliverGeometry` 。**

所以在 Flutter 里：

- `ListView` 使用的是 `SliverFixedExtentList` 或者 `SliverList` ；
- `GridView` 使用的是 `SliverGrid` ；
- `PageView` 使用的是 `SliverFillViewport` ；

> 当然这里有一个特殊的是 `SingleChildScrollView` ， 因为它是单个 `child` 的可滑动控件，它并没有使用 `RenderSliver` ，而是直接自定义了一个 `RenderObject` （RenderBox） ，并且**在 `performLayout` 时直接调整 `child` 的 `offset` 来达到滑动效果**。

RenderSliver

我们都知道 Flutter 中的整体渲染流程是 *Widget -> Element -> RenderObejct -> Layer* 这样的过程，而 **Flutter 里的布局和绘制逻辑都在 `RenderObejct` 。**

而事实上 `RenderObejct` 也可以分为两大基础子类：

- `RenderBox` ： 我们**常用的布局控件都是基于 RenderBox** 来实现布局；
- `RenderSliver` ：**主要用在 Viewport 里实现布局**， *Viewport* 里的直属 *children* 也需要是 *RenderSliver*；

那到这里你可能会有一个疑问：既然前面 `SingleChildScrollView` 里没有使用 `RenderSliver` ，直接使用 `RenderBox` 也可以实现滑动，**为什么还要用 Viewport + RenderSliver 的方式来实现列表滑动?**

RenderBox

在 `SingleChildScrollView` 内部使用的是 `RenderBox` ，那么在布局过程中自然而然会把整个 `child` 都进行布局和计算，绘制时主要也是通过 `offset` 和 `clip` 等来完成移动效果，这样的实现当 **`child` 比较复杂或者过长时，性能就会变差**。

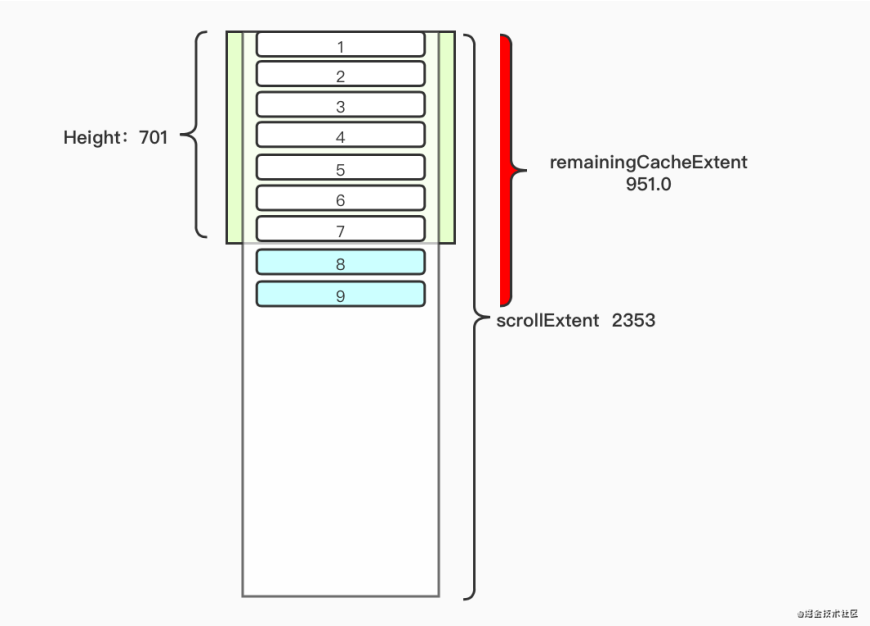
RenderSliver

`RenderSliver` 的实现相对 `RenderBox` 就复杂更多，前面介绍过 **`RenderSliver` 就是通过 `SliverConstraints` 来得到一个 `SliverGeometry`** ，其中：

- `SliverConstraints` 中有 *remainingPaintExtent* 可以用来表示剩余的可绘制具体的大小；
- `SliverGeometry` 里也有 `scrollExtent` （可滑动的距离）、 `paintExtent` （可绘制大小）、 `layoutExtent` （布局大小范围）、 `visible` (是否需要绘制)等参数；

所以通过这部分参数，**在 `Viewport` 里可以实现动态管理，节省资源，根据 `SliverGeometry` 判断需要绘制多大区域的内容，还剩多少内容可以绘制，需要加载的布局是哪些等等。**

简单地说就是可以实现“懒加载”，按需绘制，从而得到更流畅的滑动体验。



以 `ListView` 为例，如上图所示是一个高为 701 的 `ListView` ，实际布局渲染之后，对于 `SliverList` 输出的 `SliverGeometry` 而言：

- 设定里每个 item 的高度为 114；
- `scrollExtent` 是 2353，也就是整体可滑动距离等于 2353；
- `paintExtent` 是 701， 因为 `ListView` 的 `Viewport` 是 701，所以从 `SliverConstraints` 得到的 `remainingPaintExtent` 是 701，**所以默认只需要绘制和布局高度为 701 的部分；** （因为默认 paintExtent = layoutExtent ）
- 对 item 多出的蓝色 8-9 部分，这是因为在 `SliverConstraints` 内会有一个叫 `remainingCacheExtent` 的参数，它表示了需要提前缓存的布局区域， 也就是“预布局”的区域，这个区域默认大小是 **defaultCacheExtent= 250.0；**

> `ListView` 高度为 701， `defaultCacheExtent` 为默认的 250，也就是得到**第一次需要布局到底部的距离其实为 951**，按照每个 item 高度是 114 ，那么其实是有 8.3 个 item 高度，取整数也就是 9 个 item ，最终得到整体需要处理的区域大小为 114 *9 = 1026 ，在 * `SliverList` *内部就是* `endScrollOffset` *参数。*

所以根据以上情况， **`ListView` 会输出一个 `paintExtent` 为 701 ， `cacheExtent` 为 1026 的 `SliverGeometry`** 。

从这个例子可以看出， **`RenderSliver` 在实现可滑动列表的开销和逻辑上，会比直接使用 `RenderBox` 好和灵活很多**，同时也是为什么 `Viewport` 里需要使用 `RenderSliver` 而不是 `RenderBox` 的原因。

> ⚠️注意，这里比较容易有一个误区，那就是 `ListView` 是由 `Viewport` + `Scrollable` 和一个 `RenderSliver` 组成，所以在 **`ListView` 里只会有一个 `RenderSliver` 而不是多个**，想使用多个 `RenderSliver` 需要使用 `CustomScrollView` 。

最后顺便聊下 `CustomScrollView` ，事实上就是一个**开放了可自定义配置 `RenderSliver` 数组的滑动控件**，例如：

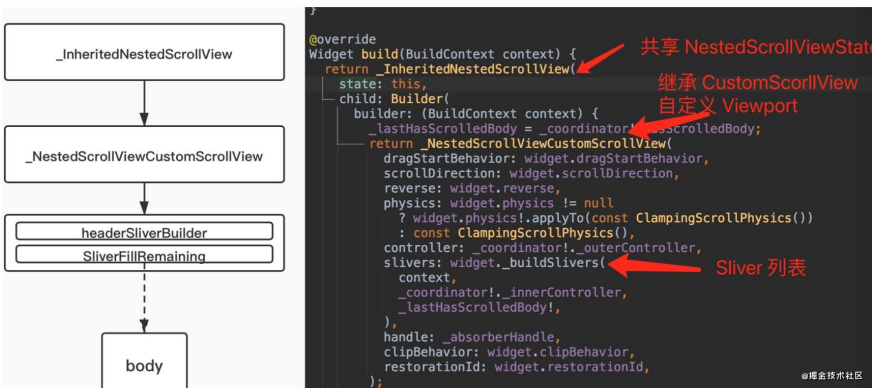
- 通过利用 `SliverList` + `SliverGrid` 就可以搭配出多样化的滑动列表；
- 通过 `CupertinoSliverRefreshControl` + `SliverList` 实现类似 iOS 原生的下拉刷新列表；

其他可用的内置 `Sliver` 还有： `SliverPadding` 、 `SliverFillRemaining` 、 `SliverFillViewport` 、 `SliverPersistentHeader` 、 `SliverAppbar` 等等。

NestedScrollView

为什么会把 `NestedScrollView` 单独拿出来说呢？这是因为 `NestedScrollView` 和前面介绍的滑动列表实现不大一样。

内部组成



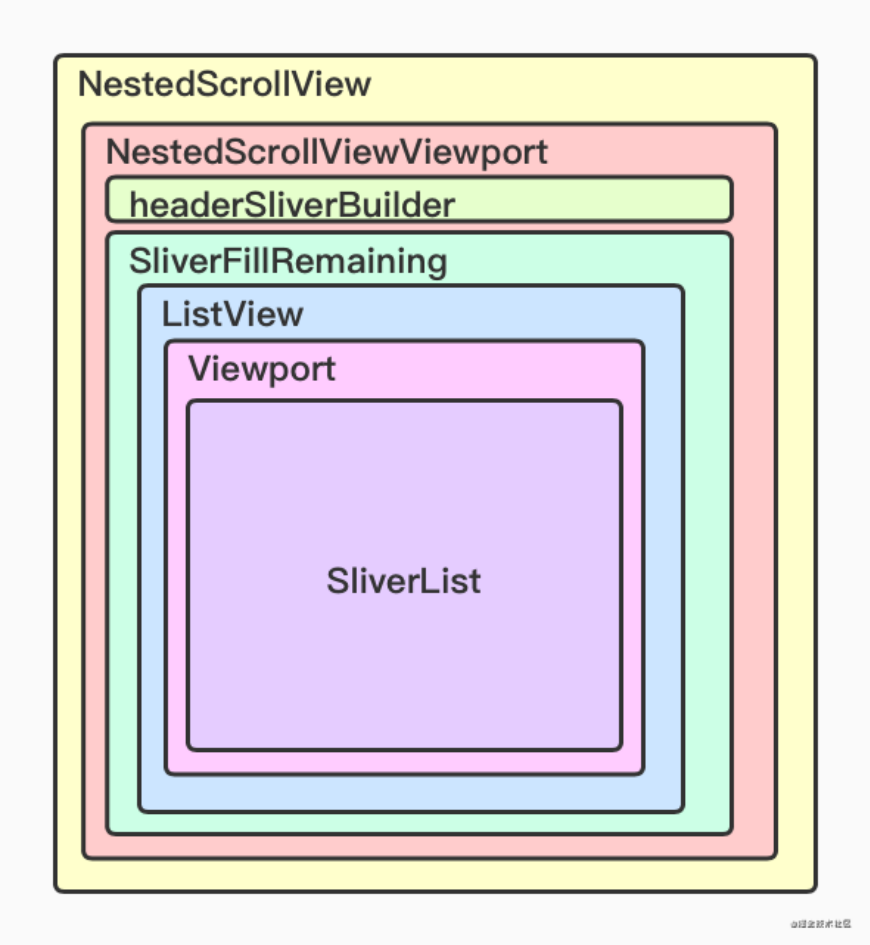
如上图所示，`NestedScrollView` 内部主要是通过继承 `CustomScrollView`，然后自定义一个 `NestedScrollViewViewport` 来实现联动的效果。

那这有什么特别的呢？如下代码所示，这是使用 `NestedScrollView` 常用的模式，那有看出什么特别的地方了吗？

```
child: NestedScrollView(
  physics: const AlwaysScrollableScrollPhysics(),
  headerSliverBuilder: _sliverBuilder,
  body: ListView.builder(
    itemBuilder: (_, index) {
      return Card(
        child: new Container(
          height: 60,
          padding: EdgeInsets.only(left: 10),
          alignment: Alignment.centerLeft,
          child: new Text("Item $index"),
        ),
      );
    },

    ///根据状态返回数量
    itemCount: listCount,
  ),
```


代码里 `NestedScrollView` 的 `body` 嵌套的是 `ListView`，前面我们介绍了 `ListView` 本身就是 `Viewport` + `Scrollable` + `SliverList` 组合，而 `NestedScrollView` 本身也有 `NestedScrollViewViewport`。

所以 `NestedScrollView` 的实现本质上其实就是 `Viewport` 嵌套 `Viewport`，会有两个 `Scrollable` 的存在，并且嵌套的 `ListView` 是被放在了 `NestedScrollView` 的 `Sliver` 里面，大致如下图所示。



这里面有几个关键的对象，其中：

- `SliverFillRemaining` ：用于充满 `Viewport` 的剩余空间，在 `NestedScrollView` 里面就是充满 `header` 之外的剩余空间；
- `NestedScrollViewViewport` ： 在原 `Viewport` 的基础上增加了一个 `SliverOverlapAbsorberHandle` 参数， `SliverOverlapAbsorberHandle` 本身是一个 `ChangeNotifier` ， 主要是用来当 `markNeedsLayout` 时对外发出通知，比如对 header 部分；

所以 `NestedScrollView` 本质上两个 `Viewport` 之间的嵌套，那他们之间是滑动关系是如何处理的？**这就要说到 `NestedScrollView` 里的 `_NestedScrollCoordinator` 对象。**

_NestedScrollCoordinator

`_NestedScrollCoordinator` 的实现比较复杂，简单地说 `_NestedScrollCoordinator` 内部创建了两个 `_NestedScrollController` ：

- `_outerController` ： 属于 `_NestedScrollViewCustomScrollView` 的 *controller* ， 也就是它

自己 *controller*；

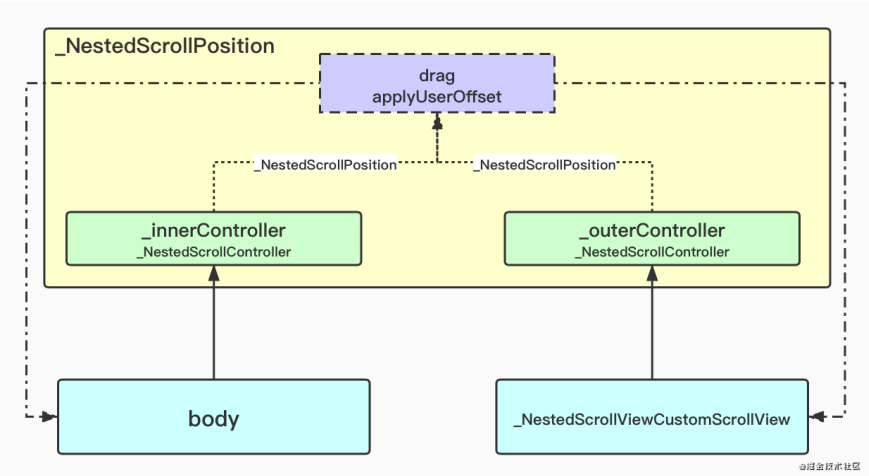
- `_innerController` ：属于 `body` 的 *controller*；

```
List<Widget> _buildSlivers(BuildContext context, ScrollController innerController, bool bodyIsScrolled) {
  return <Widget>[
    ...headerSliverBuilder(context, bodyIsScrolled),
    SliverFillRemaining(
      child: PrimaryScrollController(
        controller: innerController,
        child: body,
      ),
    ),
  ];
}
```

> 在 `ListView` 的父类 `ScrollView` 内部，默认情况下使用的就是 `PrimaryScrollController.of(context)` 这个 *controller*，因为 `PrimaryScrollController` 是一个 `InheritedWidget` 。

而整个联动滑动的流程，主要就是 `_NestedScrollCoordinator` 里和它创建的两个 `_NestedScrollController` 有关系：

- `_NestedScrollController` 的主要作用就是使用 `_NestedScrollPosition` 来替换 `ScrollPosition` ；
- `_NestedScrollCoordinator` 将 _outer 和 _inner 两个 `_NestedScrollController` 组合起来(_outer 和 _inner 分别被应用到 `NestedScrollView` 和 `body`)；
- `_NestedScrollPosition` 内部将 `Drag` 等手势操作传递回 `_NestedScrollCoordinator` 里。
- 最后在 `_NestedScrollCoordinator` 的 `drag` 和 `applyUserOffset` 等方法里进行内外滚动的分配；



SliverPersistentHeader

了解完 `NestedScrollView` 的布局和联动实现之外，最后简单介绍一下 `SliverPersistentHeader` ， 因为经常在 `NestedScrollView` 里使用的 `SliverAppBar` ，本质上 **`SliverAppBar` 的实现靠的就是 `SliverPersistentHeader`** 。

`SliverPersistentHeader` 主要是具备 `floating` 和 `pinned` 两个属性，它们的区别主要在于使用了不同的 `RenderSliver` 实现，而**最终不同的地方其实就是输出 `SliverGeometry` 的不同。**

```dart
@override
Widget build(BuildContext context) {
  if (floating && pinned)
    return _SliverFloatingPinnedPersistentHeader(delegate: delegate);
  if (pinned)
    return _SliverPinnedPersistentHeader(delegate: delegate);
  if (floating)
    return _SliverFloatingPersistentHeader(delegate: delegate);
  return _SliverScrollingPersistentHeader(delegate: delegate);
}
```

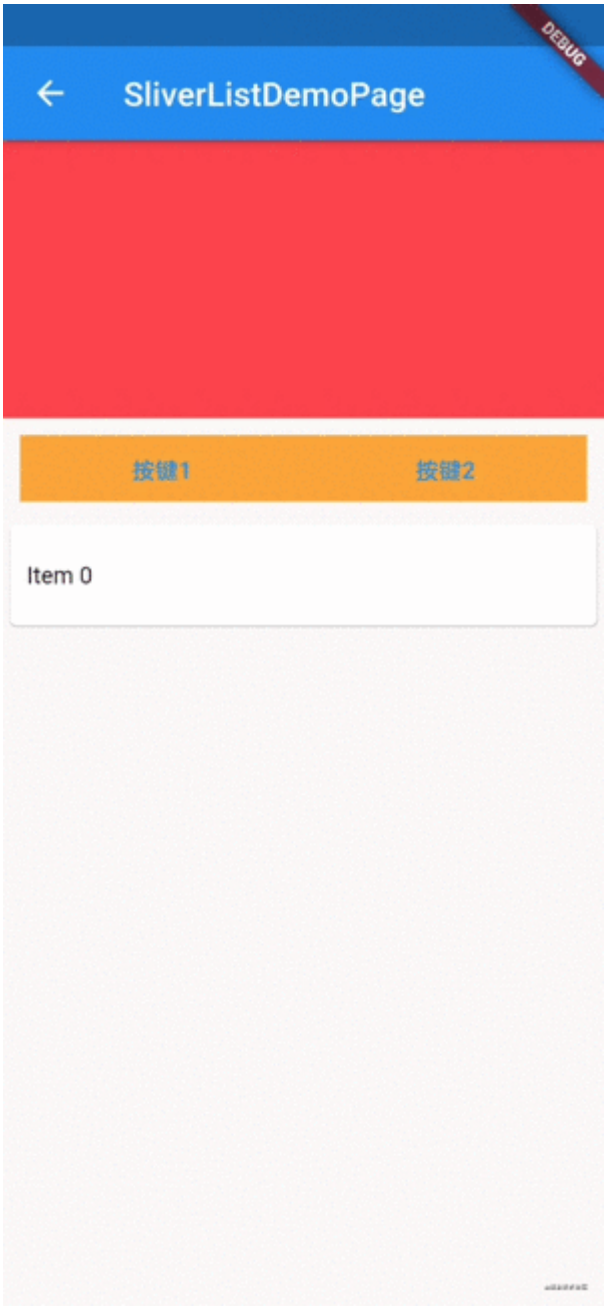
以第一个 `_SliverFloatingPinnedPersistentHeader` 和最后一个 `_SliverScrollingPersistentHeader` 之间的对比为例子，如下代码所示，在需要 `floating` 和 `pinned` 的 `Sliver` 上，可以看到 `paintExtent` 和 `layoutExtent` 都有一个最小值。

```dart
@override
double updateGeometry() {
  final double minExtent = this.minExtent;
  final double minAllowedExtent = constraints.remainingPaintExtent > minExtent ?
    minExtent :
    constraints.remainingPaintExtent;
  final double maxExtent = this.maxExtent;
  final double paintExtent = maxExtent - _effectiveScrollOffset!;
  final double clampedPaintExtent = paintExtent.clamp(
    minAllowedExtent,
    constraints.remainingPaintExtent,
  );
  final double layoutExtent = maxExtent - constraints.scrollOffset;
  final double stretchOffset = stretchConfiguration != null ?
    constraints.overlap.abs() :
    0.0;
  geometry = SliverGeometry(
    scrollExtent: maxExtent,
    paintOrigin: math.min(constraints.overlap, 0.0),
    paintExtent: clampedPaintExtent,
    layoutExtent: layoutExtent.clamp(0.0, clampedPaintExtent),
    maxPaintExtent: maxExtent + stretchOffset,
    maxScrollObstructionExtent: minExtent,
    hasVisualOverflow: true, // Conservatively say we do have overflow to avoid c
  );
  return 0.0;
```

```dart
@protected
double updateGeometry() {
  double stretchOffset = 0.0;
  if (stretchConfiguration != null && _childPosition == 0.0) {
    stretchOffset += constraints.overlap.abs();
  }
  final double maxExtent = this.maxExtent;
  final double paintExtent = maxExtent - constraints.scrollOffset;
  geometry = SliverGeometry(
    scrollExtent: maxExtent,
    paintOrigin: math.min(constraints.overlap, 0.0),
    paintExtent: paintExtent.clamp(0.0, constraints.remainingPaintExtent),
    maxPaintExtent: maxExtent + stretchOffset,
    hasVisualOverflow: true, // Conservatively say we do have overflow to avoi
  );
  return stretchOffset > 0 ? 0.0 : math.min(0.0, paintExtent - childExtent);
}
```

所以 `Sliver` 被固定住的原理，其实就是 `Viewport` 得到了它的 `paintExtent` 和 `layoutExtent` 并不为 0，所以会继续为这个 `Sliver` 绘制对应区域的内容。

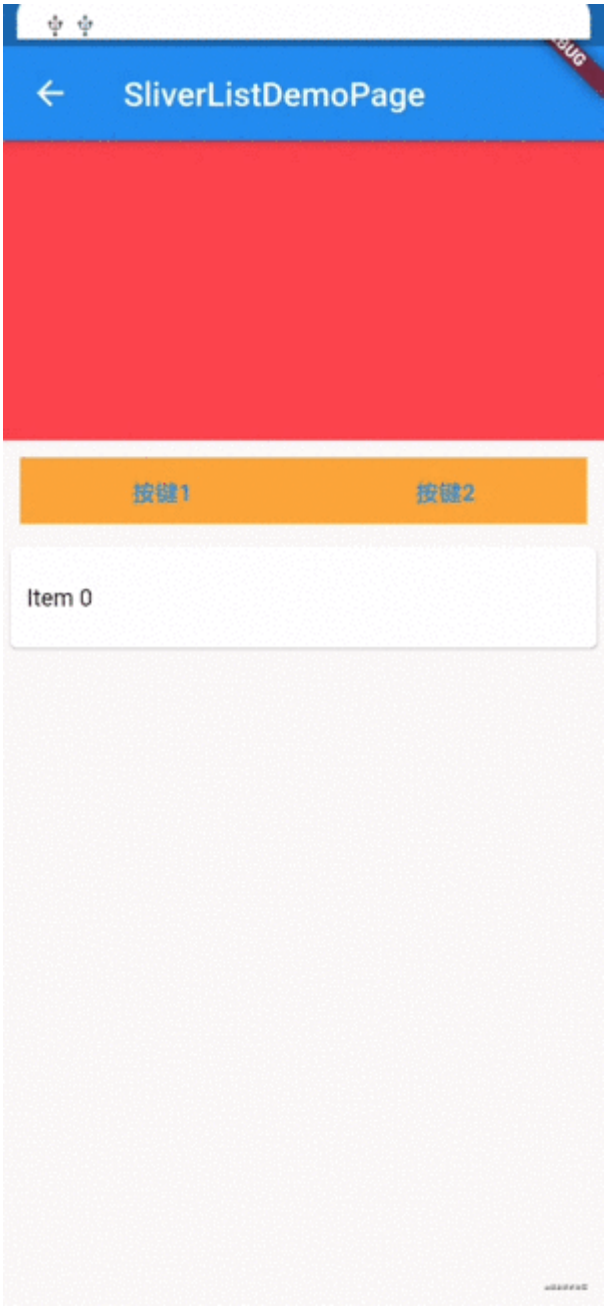
最后需要注意的是，**当你使用 `SliverPersistentHeader` 去固定住头部的时候，作为 `body` 的列表是不知道顶部有个固定区域。** 所以如果这时候不额外做一些处理，那么对于 `body` 而言，它的 `paintOrigin` 还是从最顶部开始而不是固定区域的下方。



> 如上动图所示，可以看到 item0 并没有在橙色区域停止滑动，而是继续往上滑动，这就是因为作为 `body` 的列表不知道顶部有固定区域。

这时候就可以通过使用 `SliverOverlapAbsorber` + `SliverOverlapInjector` 的组合来解决这个问题：

- 在 `SliverPersistentHeader` 的外层嵌套一个 `SliverOverlapAbsorber` 用于吸收 `SliverPersistentHeader` 的高度；
- 使用 `SliverOverlapInjector` 将这个高度配置到 `body` 列表中，让列表知道顶部存在一个固定高度的区域；



这部分例子可见：
https://github.com/CarGuo/gsy_flutter_demo/blob/master/lib/widget/sliver_list_demo_page.dart

好了，本篇关于 Flutter 滑动列表的实现原理就介绍完了，如果你还有什么想说的，欢迎留言讨论。