

Flutter编程指南



作者：刘小壮

Github: <https://github.com/DeveloperErenLiu>

博客地址: <https://www.jianshu.com/u/2de707c93dc4>

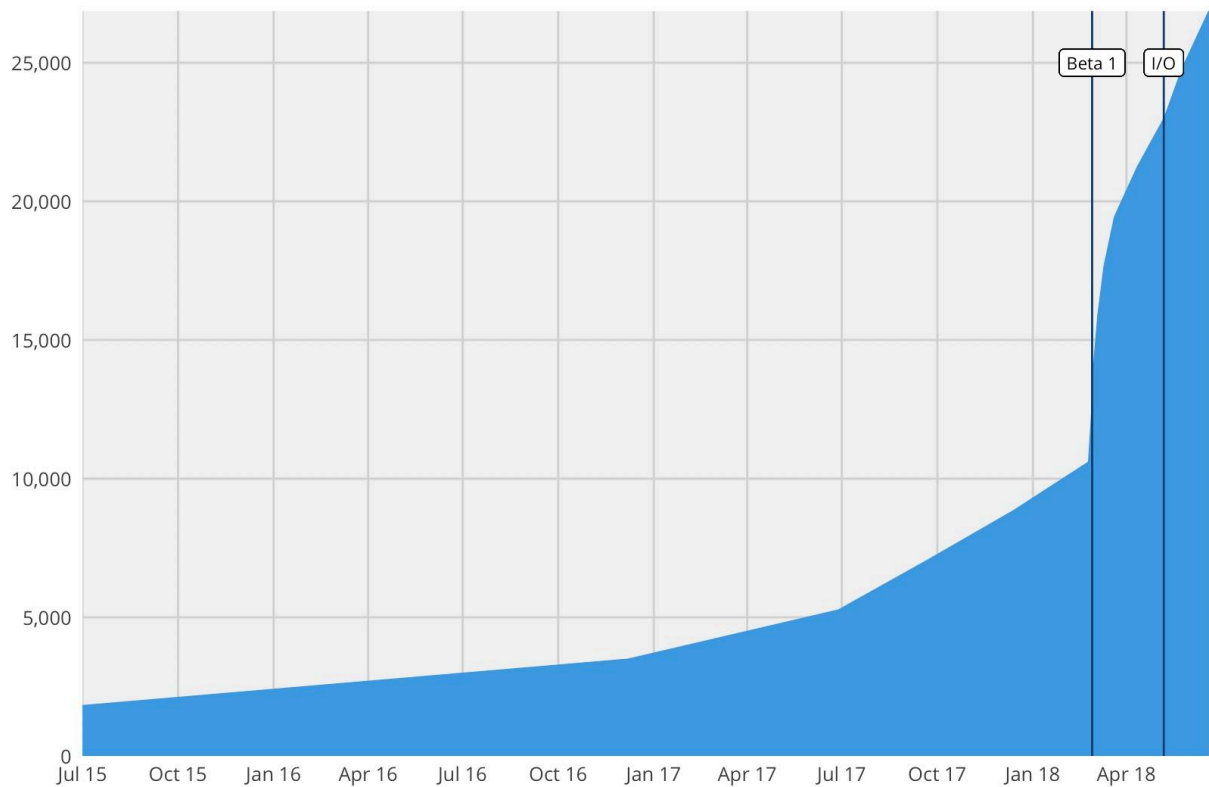
介绍

Flutter 是 Google 开发的新一代跨平台方案，Flutter 可以实现写一份代码同时运行在 iOS 和 Android 设备上，并且提供很好的性能体验。Flutter 使用 Dart 作为开发语言，这是一门简洁、强类型的编程语言。Flutter 对于 iOS 和 Android 设备，提供了两套视觉库，可以针对不同的平台有不同的展示效果。

Flutter 原本是为了解决 Web 开发中的一些问题，而开发的一套精简版 Web 框架，拥有独立的渲染引擎和开发语言，但后来逐渐演变为移动端开发框架。正是由于 Dart 当初的定位是为了替代 JS 成为 Web 框架，所以 Dart 的语法更接近于 JS 语法。例如定义对象构建方法，以及实例化对象的方式等。

在 Google 刚推出 Flutter 时，其发展很缓慢，终于在 18 年发布第一个 Beta 版之后迎来了爆发性增长，发布第一个 Release 版时增长速度更快。可以从 Github 上 Star 数据看出来这个增长的过程。在 19 年最新的 Flutter 1.2 版本中，已经开放 Web 支持的 Beta 版。

GitHub stars by date

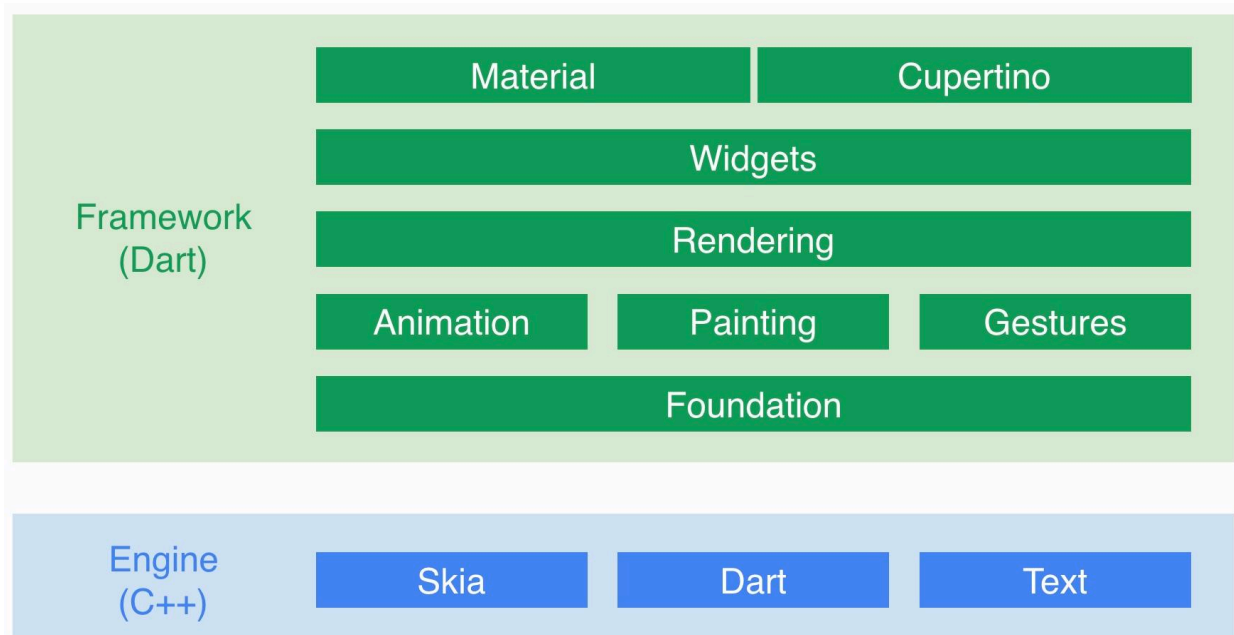


Source: <http://timqian.com/star-history/>

目前已经有不少大型项目接入 `Flutter`，阿里的咸鱼、头条的抖音、腾讯的NOW直播，都将 `Flutter` 当做应用程序的开发语言。除此之外，还有一些其他中小型公司也在做。

整体架构

`Flutter` 可以理解为开发SDK或者工具包，其通过 `Dart` 作为开发语言，并且提供 `Material` 和 `Cupertino` 两套视觉控件，视图或其他和视图相关的类，都以 `Widget` 的形式表现。`Flutter` 有自己的渲染引擎，并不依赖原生平台的渲染。`Flutter` 还包含一个用 `C++` 实现的 `Engine`，渲染也是包含在其中的。



Engine

Flutter 是一套全新的跨平台方案，Flutter 并不像 React Native 那样，依赖原生应用的渲染，而是自己有自己的渲染引擎，并使用 Dart 当做 Flutter 的开发语言。Flutter 整体框架分为两层，底层是通过 C++ 实现的引擎部分，Skia 是 Flutter 的渲染引擎，负责跨平台的图形渲染。Dart 作为 Flutter 的开发语言，在 C++ 引擎上层是 Dart 的 Framework。

Flutter 不仅仅提供了一套视觉库，在 Flutter 整体框架中包含各个层级阶段的库。例如实现一个游戏功能，上面一些游戏控件可以用上层视觉库，底层游戏可以直接基于 Flutter 的底层库进行开发，而不需要调用原生应用的底层库。Flutter 的底层库是基于 Open GL 实现的，所以 Open GL 可以做的 Flutter 都可以。

视觉库

在上层 Framework 中包含两套视觉库，符合 Android 风格的 Material，和符合 iOS 风格的 Cupertino。也可以在此基础上，封装自己风格的系统组件。Cupertino 是一套 iOS 风格的视觉库，包含 iOS 的导航栏、button、alertView 等。

Flutter 对不同硬件平台有不同的兼容，例如同样的 Material 代码运行在 iOS 和 Android 不同平台上，有一些平台特有的显示和交互，Flutter 依然对其进行了区分适配。例如滑动 ScrollView 时，iOS 平台是有回弹效果的，而 Android 平台则是阻尼效果。例如 iOS 的导航栏标题是居中的，Android 导航栏标题是向左的，等等。这些 Flutter 都做了区分兼容。

除了 Flutter 为我们做的一些适配外，有一些控件是需要我们自己做适配的，例如 AlertView，在 Android 和 iOS 两个平台下的表现就是不同的。这些 iOS 特性的控件都定义在 Cupertino 中，所以建议在进行 App 开发时，对一些控件进行上层封装。

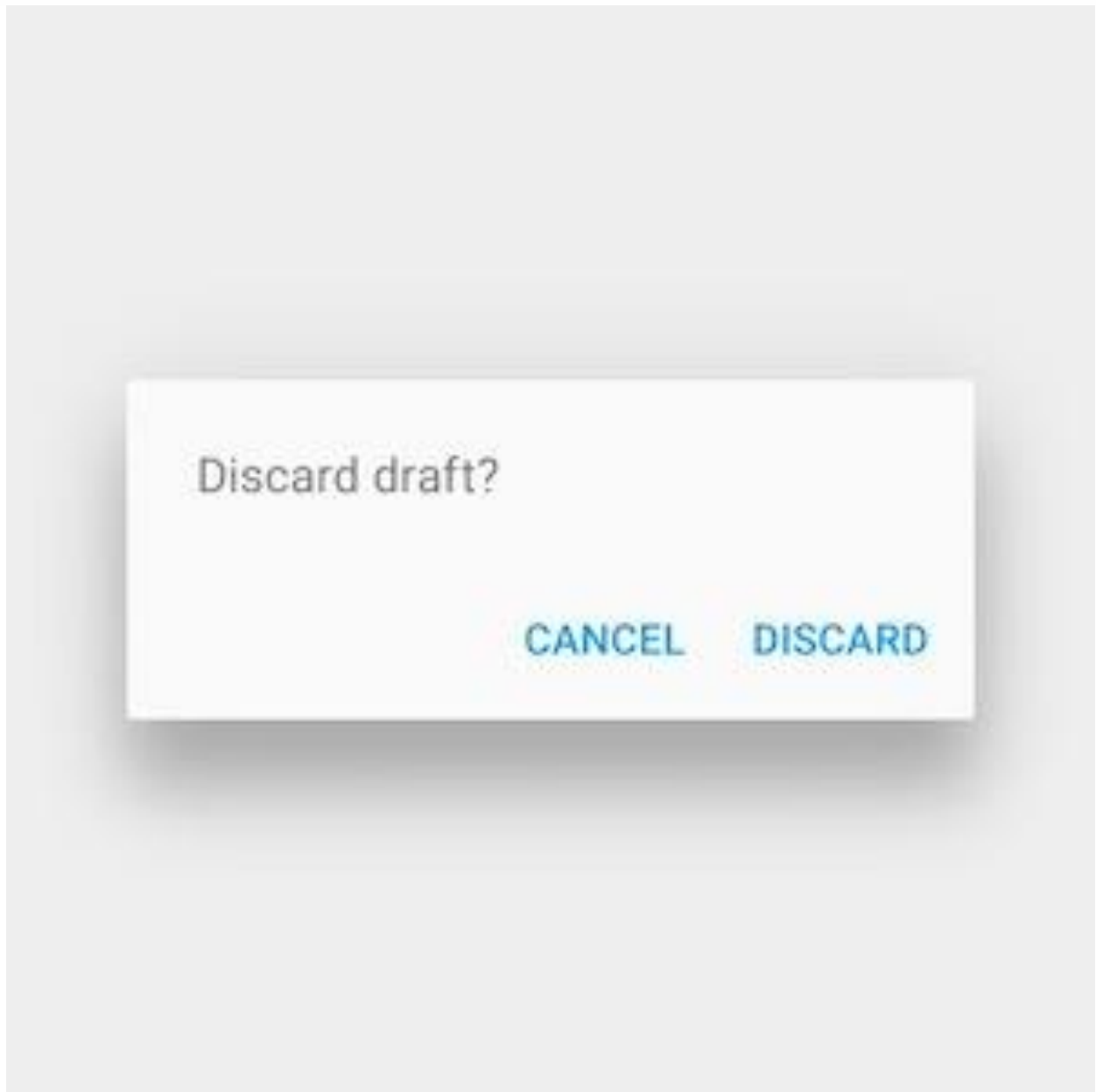
例如 AlertView 则对其进行一个二次封装，控件内部进行设备判断并选择不同的视觉库，这样可以保证各个平台的效果。

**Allow "Maps" to access your
location while you use the
app?**

Your current location will be displayed
on the map and used for directions,
nearby search results, and estimated
travel times.

Don't Allow

Allow



虽然 `Flutter` 对于iOS和Android两个平台，开发有 `cupertino` 和 `material` 两个视觉库，但实际开发过程中的选择，应该使用 `material` 当做视觉库。因为 `Flutter` 对iOS的支持并不是很好，主要对Android平台支持比较好，`material` 中的UI控件要比 `cupertino` 多好几倍。

Dart

`Dart`是 `Google` 在2011年推出的一款应用于 `Web` 开发的编程语言，`Dart` 刚推出的时候，定位是替代 `JS` 做前端开发，后来逐步扩展到移动端和服务端。

Platform	Use case	Get started
Mobile 	Create an app from a single codebase that runs on both iOS and Android.	Flutter
Web 	Create an app that runs in any modern browser.	Dart for the web
Server 	Create a command-line tool or server.	Server-side Dart

`Dart` 是 `Flutter` 的开发语言，`Flutter` 必须遵循 `Dart` 的语言特性。在此基础上，也会有自己的东西，例如 `Flutter` 的上层 `Framework`，自己的渲染引擎等。可以说，`Dart` 只是 `Flutter` 的一部分。

`Dart` 是强类型的，对定义的变量不需要声明其类型，`Flutter` 会对其进行类型推导。如果不想使用类型推导，也可以自己声明指定的类型。

Hot Reload

`Flutter` 支持亚秒级热重载，`Android Studio` 和 `VSCode` 都支持 `Hot Reload` 的特性。但需要区分的是，热重载和热更新是不同的两个概念，热重载是在运行调试状态下，将新代码直接更新到执行中的二进制。而热更新是在上线后，通过 `Runtime` 或其他方式，改变现有执行逻辑。

AOT、JIT

`Flutter` 支持 `AOT` (`Ahead of time`) 和 `JIT` (`Just in time`) 两种编译模式，`JIT` 模式支持在运行过程中进行 `Hot Reload`。刷新过程是一个增量的过程，由系统对本次和上次的代码做一次 `snapshot`，将新的代码注入到 `DartVM` 中进行刷新。但有时会不能进行 `Hot Reload`，此时进行一次全量的 `Hot Reload` 即可。

而 `AOT` 模式则是在运行前预先编译好，这样在每次运行过程中就不需要进行分析、编译，此模式的运行速度是最快的。`Flutter` 同时采用了两种方案，在开发阶段采用 `JIT` 模式进行开发，在 `release` 阶段采用 `AOT` 模式，将代码打包为二进制进行发布。

在开发原生应用时，每次修改代码后都需要重新编译，并且运行到硬件设备上。由于 `Flutter` 支持 `Hot Reload`，可以进行热重载，对项目的开发效率有很大的提升。

由于 `Flutter` 实现机制支持 `JIT` 的原因，理论上来说是支持热更新以及服务器下发代码的。可以从服务器。但是由于这样会使性能变差，而且还有审核的问题，所以 `Flutter` 并没有采用这种方案。

实现原理

`Flutter` 的热重载是基于 `State` 的，也就是我们在代码中经常出现的 `setState` 方法，通过这个来修改后，会执行相应的 `build` 方法，这就是热重载的基本过程。

`Flutter` 的 `hot reload` 的实现源码在下面路径中，在此路径中包含 `run_cold.dart` 和 `run_hot.dart` 两个文件，前者负责冷启动，后者负责热重载。

```
~/flutter/packages/flutter_tools/lib/src/run_hot.dart
```

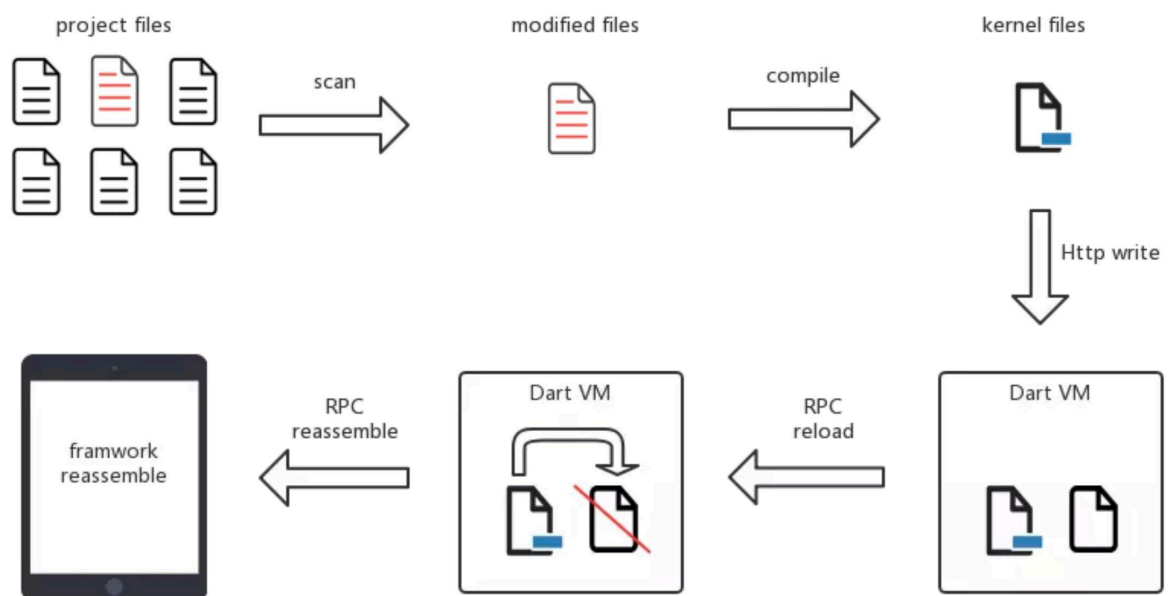
热重载的代码实现在 `run_hot.dart` 文件中，有 `HotRunner` 来负责具体代码执行。当 `Flutter` 进行热重载时，会调用 `restart` 函数，函数内部会传入一个 `fullRestart` 的 `bool` 类型变量。热重载分为全量和非全量，`fullRestart` 参数就是表示是否全量。以非全量热重载为例，函数的 `fullRestart` 会传入 `false`，根据传入 `false` 参数，下面是部分核心代码。


```

Future<OperationResult> restart({ bool fullRestart = false, bool pauseAfterRestart
= false, String reason }) async {
  if (fullRestart) {
    // .....
  } else {
    final bool reloadOnTopOfSnapshot = _runningFromSnapshot;
    final String progressPrefix = reloadOnTopOfSnapshot ? 'Initializing' :
'Performing';
    final Status status = logger.startProgress(
      '$progressPrefix hot reload...',
      progressId: 'hot.reload'
    );
    OperationResult result;
    try {
      result = await _reloadSources(pause: pauseAfterRestart, reason:
reason);
    } finally {
      status.cancel();
    }
  }
}

```

调用 `restart` 函数后，内部会调用 `_reloadSources` 函数，去执行内部逻辑。下面是大概逻辑执行流程。



在 `_reloadSources` 函数内部，会调用 `_updateDevFS` 函数，函数内部会扫描修改的文件，并将文件修改前后进行对比，随后会将被改动的代码生成一个 `kernel files` 文件。

随后会通过 `HTTP Server` 将生成的 `kernel files` 文件发送给 `Dart VM` 虚拟机，虚拟机拿到 `kernel` 文件后会调用 `_reloadSources` 函数进行资源重载，将 `kernel` 文件注入正在运行的 `Dart VM` 中。当资源重载完成后，会调用RPC接口触发 `Widgets` 的重绘。

跨平台方案对比

现在市面上RN、Weex的技术方案基本一样，所以这里就以RN来代表类似的跨平台方案。`Flutter`是基于 `GPU` 进行渲染的，而RN则将渲染交给原生平台，而自己只是负责通过 `JSCore` 将视图组织起来，并处理业务逻辑。所以在渲染效果和性能这块，`Flutter` 的性能比RN要强很多。

跨平台方案一般都需要对各个平台进行平台适配，也就是创建各自平台的适配层，RN的平台适配层要比 `Flutter` 要大很多。因为从技术实现来说，RN是通过 `JSCore` 引擎进行原生代码调用的，和原生代码交互很多，所以需要更多的适配。而 `Flutter` 则只需要对各自平台独有的特性进行适配即可，例如调用系统相册、粘贴板等。

`Flutter` 技术实现是基于更底层实现的，对平台依赖度不是很高，相对来说，RN对平台的依赖度是很高的。所以RN未来的技术升级，包括扩展之类的，都会受到很大的限制。而 `Flutter` 未来的潜力将会很大，可以做很多技术改进。

Widget

在 `Flutter` 中将显示以及和显示相关的部分，都统一定义为 `widget`，下面列举一些 `widget` 包含的类型：

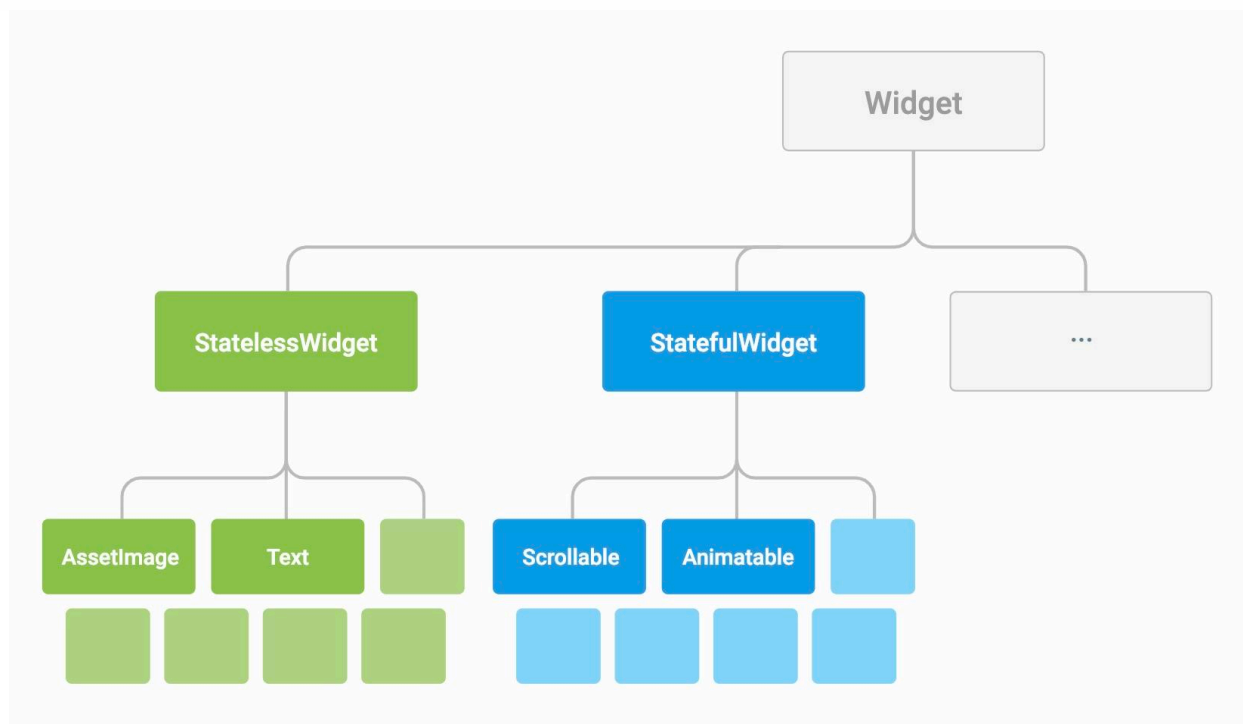
1. 用于显示的视图，例如 `ListView`、`Text`、`Container` 等。
2. 用来操作视图，例如 `Transform` 等动画相关。
3. 视图布局相关，例如 `Center`、`Expanded`、`Column` 等。

在 `Flutter` 中，所有的视图都是由 `Widget` 组成，`Label`、`AppBar`、`ViewController` 等。

在 `Flutter` 的设计中，组合的优先级要大于继承，整体视图类结构继承层级很浅但单层很多类。如果想定制或封装一些控件，也应该以组合为主，而不是继承。

在iOS开发中，我也经常采用这种设计方案，组合大于继承。因为如果继承层级过多的话，一个是不便于阅读代码，还有就是不好维护代码。例如底层需要改一个通用的样式，但这个类的继承层级比较复杂，这样改动的话影响范围就比较大，会将一些不需要改的也改掉，这时候就会发现继承很鸡肋。但在iOS中有 `Category` 的概念，这也是一种组合的方式，可以通过将一些公共的东西放在 `Category` 中，使继承的方便性和组合的灵活性达到一个平衡。

`Flutter` 中并没有单独的布局文件，例如iOS的XIB这种，代码都在 `Widget` 中定义。和 `UIView` 的区别在于，`Widget` 只是负责描述视图，并不参与视图的渲染。`UIView` 也是负责描述视图，而 `UIView` 的 `layer` 则负责渲染操作，这是二者的区别。



了解Widget

在应用程序启动时，`main` 方法接收一个 `Widget` 当做主页面，所以任何一个 `Widget` 都可以当做根视图。一般都是传一个 `MaterialApp`，也可以传一个 `Container` 当做根视图，这都是被允许的。

在 `Flutter` 应用中，和界面显示及用户交互的对象都是由 `Widget` 构成的，例如视图、动画、手势等。`Widget` 分为 `StatelessWidget` 和 `StatefulWidget` 两种，分别是无状态和有状态的 `Widget`。

`StatefulWidget` 本质上也是无状态的，其通过 `State` 来处理 `Widget` 的状态，以达到有状态，`State` 出现在整个 `StatefulWidget` 的生命周期中。

当构建一个 `Widget` 时，可以通过其 `build` 获得构建流程，在构建流程中可以加入自己的定制操作，例如对其设置title或视图等。

```
return Scaffold(  
  appBar: AppBar(  
    title: Text('ListView Demo'),  
  ),  
  body: ListView.builder(  
    itemCount: dataList.length,  
    itemBuilder: (BuildContext context, int index) {  
      return Text(dataList[index]);  
    },  
  ),  
);
```

有些 `Widget` 在构建时，也提供一些参数来帮助构建，例如构建一个 `ListView` 时，会将 `index` 返回给 `build` 方法，来区别构建的Cell，以及构建的上下文 `context`。

```
itemBuilder: (BuildContext context, int index) {  
    return Text(dataList[index]);  
}
```

StatelessWidget

`StatelessWidget` 是一种静态 `Widget`，即创建后自身就不能再进行改变。在创建一个 `StatelessWidget` 后，需要重写 `build` 函数。每个静态 `Widget` 都会有一个 `build` 函数，在创建视图对象时会调用此方法。同样的，此函数也接收一个 `Widget` 类型的返回值。

```
class RectangleWidget extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Center (  
            // UI Code  
        );  
    }  
}
```

StatefulWidget

`Widget` 本质上是不可被改变的，但 `StatefulWidget` 将状态拆分到 `State` 中去管理，当数据发生改变时由 `State` 去处理视图的改变。

下面是创建一个动态 `Widget`，当创建一个动态 `Widget` 需要配合一个 `State`，并且需要重写 `createState` 方法。重写此函数后，指定一个 `Widget` 对应的 `State` 并初始化。

下面例子中，在 `StatefulWidget` 的父类中包含一个 `Key` 类型的 `key` 变量，这是无论静态 `Widget` 还是动态 `Widget` 都具备的参数。在动态 `Widget` 中定义了自己的成员变量 `title`，并在自定义的初始化方法中传入，通过下面 `DynamicWidget` 类的构造方法，并不需要在内部手动进行 `title` 的赋值，`title` 即为传入的值，是由系统完成的。

```
class DynamicWidget extends StatefulWidget {  
    DynamicWidget({Key key, this.title}) : super (key : key);  
    final String title;  
  
    @override  
    DynamicWidgetState createState() => new DynamicWidgetState();  
}
```

由于上面动态 `Widget` 定义了初始化方法，在调用动态 `Widget` 时可以直接用自定义初始化方法即可。

```
DynamicWidget(key: 'key', title: 'title');
```

State

`StatefulWidget` 的改变是由 `State` 来完成的，`State` 中需要重写 `build` 方法，在 `build` 中进行视图组织。`StatefulWidget` 是一种响应式视图改变的方式，数据源和视图产生绑定关系，由数据源驱动视图的改变。

改变 `StatefulWidget` 的数据源时，需要调用 `setState` 方法，并将数据源改变的操作写在里面。使用动态 `Widget` 后，是不需要我们手动去刷新视图的。系统在 `setState` 方法调用后，会重新调用对应 `Widget` 的 `build` 方法，重新绘制某个 `Widget`。

下面的代码示例中添加了一个float按钮，并给按钮设置了一个回调函数 `_onPressAction`，这样在每次触发按钮事件时都会调用此函数。`counter` 是一个整型变量并和 `Text` 相关联，当 `counter` 的值在 `setState` 方法中改变时，`Text Widget` 也会跟着变化。

```
class DynamicWidgetState extends State<DynamicWidget> {
  int counter = 0;
  void _onPressAction() {
    setState(() {
      counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      body: Center(
        child: Text('Button tapped $_counter.')
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _onPressAction,
        tooltip: 'Increment',
        child: Icon(Icons.add)
      )
    );
  }
}
```

主要Widget

在iOS中有 `UINavigationController` 的概念，其并不负责显示，而是负责控制各个页面的跳转操作。在 `Flutter` 中可以将 `MaterialApp` 理解为iOS的导航控制器，其包含一个 `navigationBar` 以及导航栈，这和iOS是一样的。

在iOS中除了用来显示的视图外，视图还有对应的 `UIViewController`。在 `Flutter` 中并没有专门用来管理视图并且和 `View` 一对一的类，但从显示的角度来说，有类似的类 `Scaffold`，其包含控制器的 `appBar`，也可以通过 `body` 设置一个 `widget` 当做其视图。

theme

`theme` 是 `Flutter` 提供的界面风格API, `MaterialApp` 提供有 `theme` 属性, 可以在 `MaterialApp` 中设置全局样式, 这样可以统一整个应用的风格。

```
new MaterialApp(  
  title: title,  
  theme: new ThemeData(  
    brightness: Brightness.dark,  
    primaryColor: Colors.lightBlue[800],  
    accentColor: Colors.cyan[600],  
  )  
);
```

如果不想使用系统默认主题, 可以将对应的控件或试图用 `Theme` 包起来, 并将 `Theme` 当做 `Widget` 赋值给其他 `Widget`。

```
new Theme(  
  data: new ThemeData(  
    accentColor: Colors.yellow,  
  ),  
  child: new FloatingActionButton(  
    onPressed: () {},  
    child: new Icon(Icons.add),  
  ),  
);
```

有时 `MaterialApp` 设定的统一风格, 并不能满足某个 `Widget` 的要求, 可能还需要有其他的外观变化, 可以通过 `Theme.of` 传入当前的 `BuildContext`, 来对 `theme` 进行扩展。

`Flutter` 会根据传入的 `context`, 顺着 `Widget` 树查找最近的 `Theme`, 并对 `Theme` 复制一份防止影响原有的 `Theme`, 并对其进行扩展。

```
new Theme(  
  data: Theme.of(context).copyWith(accentColor: Colors.yellow),  
  child: new FloatingActionButton(  
    onPressed: null,  
    child: new Icon(Icons.add),  
  ),  
);
```

网络请求

`Flutter` 中可以通过 `async`、`await` 组合使用, 进行网络请求。`Flutter` 中的网络请求大体有三种:

1. 系统自带的 `HttpClient` 网络请求, 缺点是代码量相对而言比较多, 而且对post请求支持不是很好。
2. 三方库 `http.dart`, 请求简单。

3. 三方库 `dio`，请求简单。

http网络库

`http` 网络库定义在 `http.dart` 中，内部代码定义很全，包括 `HttpStatus`、`HttpHeaders`、`Cookie` 等很多基础信息，有助于我们了解 `http` 请求协议。

因为是三方库，所以需要在 `pubspec.yaml` 中加入下面的引用。

```
http: '>=0.11.3+12'
```

下面是 `http.dart` 的请求示例代码，可以看到请求很简单，真正的请求代码其实就两行。生成一个 `Client` 请求对象，调用 `client` 实例的 `get` 方法(如果是 `post` 则调用 `post` 方法)，并用 `Response` 对象去接收请求结果即可。

通过 `async` 修饰发起请求的方法，表示这是一个异步操作，并在请求代码的前面加入 `await`，修饰这里的代码需要等待数据返回，需要过一段时间后再处理。

请求回来的数据默认是 `json` 字符串，需要对其进行 `decode` 并解析为数据对象才可以使用，这里使用系统自带的 `convert` 库进行解析，并解析为数组。

```

import 'package:http/http.dart' as http;

class RequestDemoState extends State<MyHomePage> {
  List dataList = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  // 发起网络请求
  loadData() async{
    String requestURL = 'https://jsonplaceholder.typicode.com/posts';
    Client client = Client();
    Response response = await client.get(requestURL);

    String jsonString = response.body;
    setState(() {
      // 数据解析
      dataList = json.decode(jsonString);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title)
      ),
      body: ListView.builder(
        itemCount: dataList.length,
        itemBuilder: (BuildContext context, int index) {
          return Text(dataList[index]['title']);
        },
      ),
    );
  }
}

```

在调用 `Client` 进行post数据请求时，需要传入一个字典进去，`Client` 会通过将字典当做post的 `from` 表单。

```
void requestData() async {
  var params = Map<String, String>();
  params["username"] = "lxz";
  params["password"] = "123456";

  var client = http.Client();
  var response = await client.post(url_post, body: params);
  _content = response.body;
}
```

dio网络库

`dio` 库的调用方式和 `http` 库类似，这里不过多介绍。`dio` 库相对于 `http` 库强大的在于，`dio` 库提供了更好的 `Cookie` 管理、文件的上传下载、`formData` 表单等处理。所以，如果对网络库需求比较复杂的话，还是建议使用 `dio`。

```
// 引入外部依赖
dio: ^1.0.9
```

数据解析

convert

系统自带有 `convert` 解析库，在使用时直接 `import` 即可。`convert` 类似于iOS自带的JSON解析类 `NSJSONSerialization`，可以直接将json字符串解析为字典或数组。

```
import 'dart:convert';
// 解析代码
dataList = json.decode(jsonString);
```

但是，我们在项目中使用时，一般都不会直接使用字典取值，这是一种很不好的做法。一般都会将字典或数组转换为模型对象，在项目中使用模型对象。可以定义类似 `Model.dart` 这样的模型类，并在模型类中进行数据解析，对外直接暴露公共变量来让外界获取值。

自动序列化

但如果定义模型类的话，一个是要在代码内部写取值和赋值代码，这些都需要手动完成。另外如果当服务端字段发生改变后，客户端也需要跟着进行改变，所以这种方式并不是很灵活。

可以采用json序列化的三方库 `json_serializable`，此库可以将一个类标示为自动JSON序列化的类，并对类提供JSON和对象相互转换的能力。也可以通过命令行开启一个 `watch`，当类中的变量定义发生改变时，相关代码自动发生改变。

首先引入下面的三个库，其中包括依赖库一个，以及调试库两个。


```
dependencies:
  json_annotation: ^2.0.0

dev_dependencies:
  build_runner: ^1.0.0
  json_serializable: ^2.0.0
```

定义一个模型文件，例如这里叫做 `User.dart` 文件，并在内部定义一个 `User` 的模型类。随后引入 `json_annotation` 的依赖，通过 `@JsonSerializable()` 标示此类需要被 `json_serializable` 进行合成。

定义的 `User` 类包含两部分，实例变量和两个转换函数。在下面定义json转换函数时，需要注意函数命名一定要按照下面格式命名，否则不能正常生成 `user.g.dart` 文件。

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

// 定义合成后的新文件为用户.g.dart
part 'user.g.dart';

@JsonSerializable()

class User {
  String name;
  int age;
  String email;

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

下面就是 `user.dart` 指定生成的 `user.g.dart` 文件，其中包含JSON和对象相互转换的代码。

```
part of 'user.dart';

User _$UserFromJson(Map<String, dynamic> json) {
  return User(
    json['name'] as String, json['age'] as int, json['email'] as String);
}

Map<String, dynamic> _$UserToJson(User instance) => <String, dynamic>{
  'name': instance.name,
  'age': instance.age,
  'email': instance.email
};
```

有的时候服务端返回的参数名和本地的关键字冲突，或者命名不规范，导致本地定义和服务端字段不同的情况。这种情况可以通过 `@JsonKey` 关键字，来修饰json字段匹配新的本地变量。除此之外，也可以做其他修饰，例如变量不能为空等。

```
@JsonKey(name: 'id')  
final int user_id;
```

现在项目中依然是报错的，随后我们在 `flutter` 工程的根目录文件夹下，运行下面命令。

```
flutter packages pub run build_runner watch
```

此命令的好处在于，其会在后台监听模型类的定义，当模型类定义发生改变后，会自动修改本地源码以适配新的定义。以文中 `User` 类为例，当 `User.dart` 文件发生改变后，使用 `Cmd+S` 保存文件，随后VSCode会将自定改变 `user.g.dart` 文件的定义，以适配新的变量定义。

系统文件

主要文件

- iOS文件：iOS工程文件
- Android：Android工程文件
- lib：Flutter的dart代码
- assets：资源文件夹，例如font、image等都可以放在里面
- .gitignore：git忽略文件

packages

这是一个系统文件，`Flutter` 通过 `.packages` 文件来管理一些系统依赖库，例如 `material`、`cupertino`、`widgets`、`animation`、`gesture` 等系统库就在里面，这些主要的系统库由 `.packages` 下的 `flutter` 统一管理，源码都在 `flutter/lib/src` 目录下。除此之外，还有一些其他的系统库或系统资源都在 `.packages` 中。

yaml文件

在 `Flutter` 中通过 `pubspec.yaml` 文件来管理外部引用，包含本地资源文件、字体文件、依赖库等依赖，以及应用的一些配置信息。这些配置在项目中时，需要注意代码对其的问题，否则会导致加载失败。

当修改 `yaml` 文件的依赖信息后，需要执行 `flutter get packages` 命令更新本地文件。但并不需要这么麻烦，可以直接 `Cmd+S` 保存文件，VSCode编译器会自动更新依赖。

```
// 项目配置信息
name: WeChat
description: Tencent WeChat App.
version: 1.0.0+1

// 常规依赖
dependencies:
  flutter:125864
  sdk: flutter
  cupertino_icons: ^0.1.2
  english_words: ^3.1.0

// 开发依赖
dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  uses-material-design: true
// 图片依赖
assets:
  - assets/images/ic_file_transfer.png
  - assets/images/ic_fengchao.png

// 字体依赖
fonts:
  - family: appIconFont
    fonts:
      - asset: assets/fonts/iconfont.ttf
```

Flutter开发

启动函数

和大多数编程语言一样，`dart` 也包含一个 `main` 方法，是 `Flutter` 程序执行的主入口，在 `main` 方法中写的代码就是在程序启动时执行的代码。`main` 方法中会执行 `runApp` 方法，`runApp` 方法类似于 iOS 的 `UIApplicationMain` 方法，`runApp` 函数接收一个 `Widget` 用来做应用程序的显示。

```
void main() {
  runApp()
  // code
}
```

生命周期

在iOS中通过 `AppDelegate` 可以获取应用程序的生命周期回调，在 `Flutter` 中也可以获取到。可以通过向 `Binding` 添加一个 `Observer`，并实现 `didChangeAppLifecycleState` 方法，来监听指定事件的到来。

但是由于 `Flutter` 提供的状态有限，在iOS平台只能监听三种状态，下面是示例代码。

```
class LifecycleDemoState extends State<MyHomePage> with WidgetsBindingObserver {
  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    super.didChangeAppLifecycleState(state);

    switch (state) {
      case AppLifecycleState.inactive:
        print('Application Lifecycle inactive');
        break;
      case AppLifecycleState.paused:
        print('Application Lifecycle paused');
        break;
      case AppLifecycleState.resumed:
        print('Application Lifecycle resumed');
        break;
      default:
        print('Application Lifecycle other');
    }
  }
}
```

矩阵变换

在 `Flutter` 中是支持矩阵变化的，例如 `rotate`、`scale` 等方式。`Flutter` 的矩阵变换由 `Widget` 完成，需要进行矩阵变换的视图，在外面包一层 `Transform Widget` 即可，内部可以设置其变换方式。

```

child: Container(
  child: Transform(
    child: Container(
      child: Text(
        "Lorem ipsum",
        style: TextStyle(color: Colors.orange[300], fontSize: 12.0),
        textAlign: TextAlign.center,
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
      ),
      padding: EdgeInsets.all(16.0),
    ),
    alignment: Alignment.center,
    transform: Matrix4.identity()
      ..rotateZ(15 * 3.1415927 / 180),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
)

```

在 `Transform` 中可以通过 `transform` 指定其矩阵变换方式，通过 `alignment` 指定变换的锚点。

页面导航

在iOS中可以通过 `UINavigationController` 对页面进行管理，控制页面间的push、pop跳转。`Flutter` 中使用 `Navigator` 和 `Routers` 来实现类似 `UINavigationController` 的功能，`Navigator` 负责管理导航栈，包含push、pop的操作，可以把 `UIViewController` 看做一个 `Routers`，`Routers` 被 `Navigator` 管理着。

`Navigator` 的跳转方式分为两种，一种是直接跳转到某个 `Widget` 页面，另一种是为 `MaterialApp` 构建一个 `map`，通过 `key` 来跳转对应的 `Widget` 页面。`map` 的格式是 `key : context` 的形式。

```

void main() {
  runApp(MaterialApp(
    home: MyAppHome(), // becomes the route named '/'
    routes: <String, WidgetBuilder> {
      '/a': (BuildContext context) => MyPage(title: 'page A'),
      '/b': (BuildContext context) => MyPage(title: 'page B'),
      '/c': (BuildContext context) => MyPage(title: 'page C'),
    },
  ));
}

```

跳转时通过 `pushNamed` 指定 `map` 中的 `key`，即可跳转到对应的 `Widget`。如果需要从push出来的页面获取参数，可以通过 `await` 修饰push操作，这样即可在新页面pop的时候将参数返回到当前页面。

```
Navigator.of(context).pushNamed('/b');
```

```
Map coordinates = await Navigator.of(context).pushNamed('/location');  
Navigator.of(context).pop({"lat":43.821757,"long":-79.226392});
```

编码规范

VSCode有很好的语法检查，如果有命名不规范等问题，都会以警告的形式表现出来。

1. 驼峰命名法，方法名、变量名等，都以首字母小写的驼峰命名法。类名也是驼峰命名法，但类名首字母大写。
2. 文件名，文件命名以下划线进行区分，不使用驼峰命名法。
3. `Flutter` 中创建 `Widget` 对象，可以用 `new` 修饰，也可以不用。

```
child: new Container(  
  child: Text(  
    'Hello World',  
    style: TextStyle(color: Colors.orange, fontSize: 15.0)  
  )  
)
```

4. 函数中可以定义可选参数，以及必要参数。

下面是一个函数定义，这里定义了一个必要参数 `url`，以及一个 `Map` 类型的可选参数 `headers`。

```
Future<Response> get(url, {Map<String, String> headers});
```

5. `Dart` 中在函数定义前加下划线，则表示是私有方法或变量。
6. `Dart` 通过 `import` 引入外部引用，除此之外也可以通过下面的语法单独引入文件中的某部分。

```
import "dart:collection" show HashMap, IterableBase;
```

=>调用

在 `Dart` 中经常可以看到 `=>` 的调用方式，这种调用方式类似于一种语法糖，下面是一些常用的调用方式。

当进行函数调用时，可以将普通函数调用转换为 `=>` 的调用方式，例如下面第一个示例。在此基础上，如果调用函数只有一个参数，可以将其改为第二个示例的方式，也就是可以省略调用的括号，直接写参数名。

```
(单一参数) => {函数声明}  
elements.map((element) => {  
    return element.length;  
});
```

```
单一参数 => {函数声明}  
elements.map(element => {  
    return element.length;  
});
```

当只有一个返回值，并且没有逻辑处理时，可以直接省略 `return`，返回数值。

```
(参数1, 参数2, ..., 参数N) => 表达式  
elements.map(element => element.length);
```

当调用的函数中没有参数时，可以直接省略参数，写一对空括号即可。

```
() => {函数实现}
```

小技巧

代码重构

VSCode支持对 `Dart` 语言进行重构，一般作用范围都是在函数内小范围的。

例如在创建 `Widget` 对象的地方，将鼠标焦点放在这里，当前行的最前面会有提示。点击提示后会有下面两个选项：

- `Extract Local Variable`
将当前 `Widget` 及其子 `Widget` 创建的代码，剥离到一个变量中，并在当前位置使用这个变量。
- `Extract Method`
将当前 `Widget` 及其子 `Widget` 创建的代码，封装到一个函数中，并在当前位置调用此函数。

除此之外，将鼠标焦点放在方法的一行，点击最前面的提示，会出现下面两个选项：

- `Convert to expression body`
将当前函数转换为一个表达式。
- `Convert to async function body`
将当前函数转换为一个异步线程中执行的代码。

附加效果

在 `Dart` 中添加任何附加效果，例如动画效果或矩阵转换，除了直接给 `Widget` 子类的属性赋值外，就是在被当前 `Widget` 外面包一层，就可以使当前 `Widget` 拥有对应的效果。


```
// 动画效果
floatingActionButton: FloatingActionButton(
  tooltip: 'Fade',
  child: Icon(Icons.brush),
  onPressed: () {
    controller.forward();
  },
),

// 矩阵转换
Transform(
  child: Container(
    child: Text(
      "Lorem ipsum",
      style: TextStyle(color: Colors.orange[300], fontSize: 12.0),
      textAlign: TextAlign.center,
    )
  ),
  alignment: Alignment.center,
  transform: Matrix4.identity()
    ..rotateZ(15 * 3.1415927 / 180),
),
```

快捷键(VSCode)

- `Cmd + Shift + p`：可以进行快速搜索。需要注意的是，默认是带有一个 `>` 的，这样搜索结果主要是 `dart` 代码。如果想搜索其他配置文件，或者安装插件等操作，需要把 `>` 去掉。
- `Cmd + Shift + o`：可以在某个文件中搜索某个类，但前提是需要提前进入这个文件。例如进入 `framework.dart`，搜索 `StatefulWidget` 类。

注意点

- 使用 `Flutter` 要注意代码缩进，如果缩进有问题可能会影响最后的结果，尤其是在 `.yaml` 中写配置文件的时候。
- 因为 `Flutter` 是开源的，所以遇到问题后可以进入源码中，找解决方案。
- 在代码中要注意标点符号的使用，例如第二个创建 `Stack` 的代码，如果上面是以逗号结尾，则后面的创建会失败，如果上面是以分号结尾则没问题。

```
Widget unreadMsgText = Container(  
  width: Constants.UnreadMsgNotifyDotSize,  
  height: Constants.UnreadMsgNotifyDotSize,  
  child: Text(  
    conversation.unreadMsgCount.toString(),  
    style: TextStyle(  
      color: Color(AppColors.UnreadMsgNotifyTextColor),  
      fontSize: 12.0  
    ),  
  ),  
);  
  
avatarContainer = Stack(  
  overflow: Overflow.visible,  
  children: <Widget>[  
    avatar  
  ],  
);
```

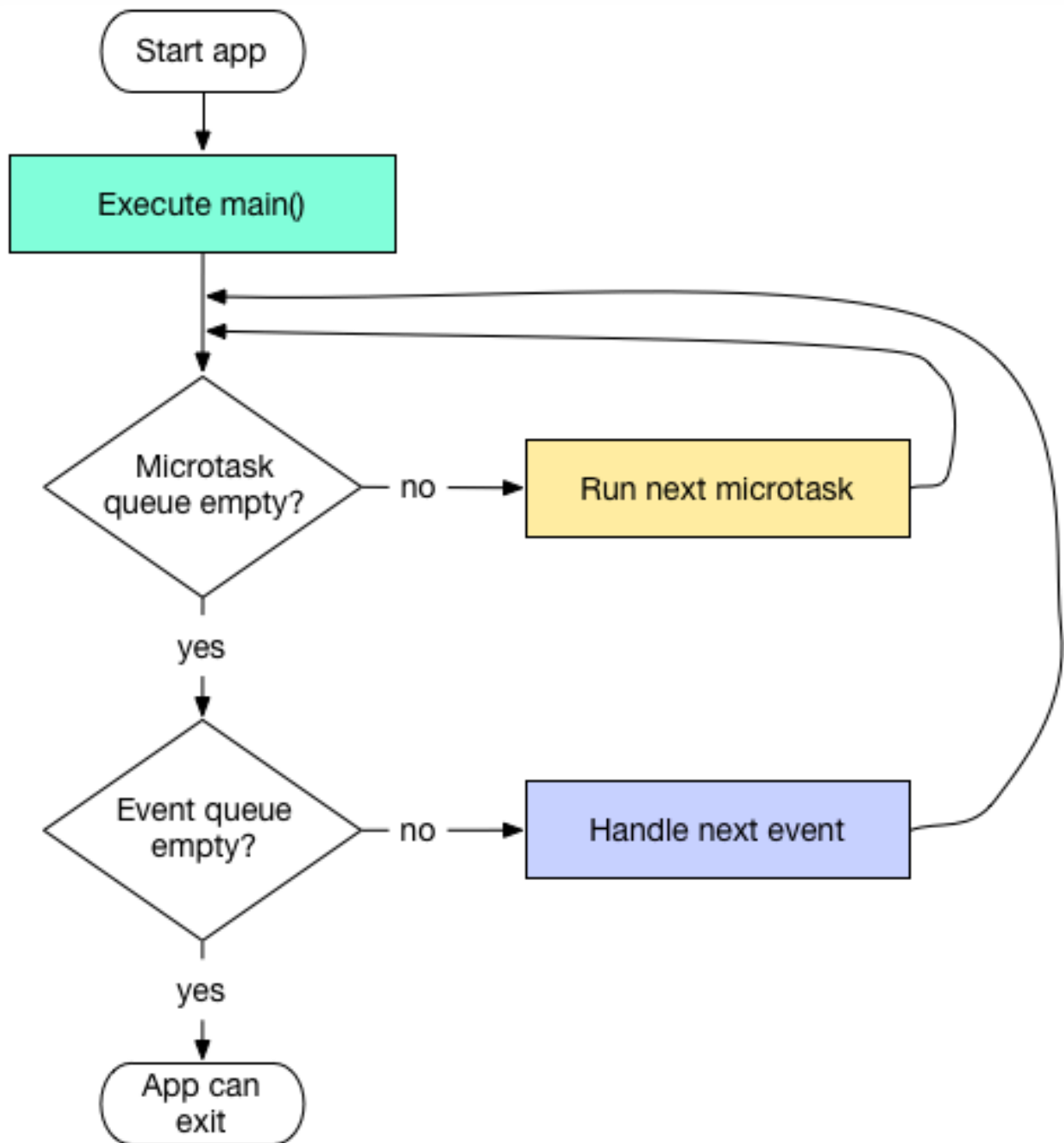
事件队列

`Flutter` 默认是单线程任务处理的，如果不开启新的线程，任务默认在主线程中处理。

和iOS应用很像，在 `Dart` 的线程中也存在事件循环和消息队列的概念，但在 `Dart` 中线程叫做 `isolate`。应用程序启动后，开始执行 `main` 函数并运行 `main isolate`。

每个 `isolate` 包含一个事件循环以及两个事件队列，`event loop` 事件循环，以及 `event queue` 和 `microtask queue` 事件队列，`event` 和 `microtask` 队列有点类似iOS的 `source0` 和 `source1`。

- `event queue`：负责处理I/O事件、绘制事件、手势事件、接收其他 `isolate` 消息等外部事件。
- `microtask queue`：可以自己向 `isolate` 内部添加事件，事件的优先级比 `event queue` 高。



这两个队列也是有优先级的，当 `isolate` 开始执行后，会先处理 `microtask` 的事件，当 `microtask` 队列中没有事件后，才会处理 `event` 队列中的事件，并按照这个顺序反复执行。但需要注意的是，当执行 `microtask` 事件时，会阻塞 `event` 队列的事件执行，这样就会导致渲染、手势响应等 `event` 事件响应延时。为了保证渲染和手势响应，应该尽量将耗时操作放在 `event` 队列中。

async、await

在异步调用中有三个关键词，`async`、`await`、`Future`，其中 `async` 和 `await` 需要一起使用。在 `Dart` 中可以通过 `async` 和 `await` 进行异步操作，`async` 表示开启一个异步操作，也可以返回一个 `Future` 结果。如果没有返回值，则默认返回一个返回值为 `null` 的 `Future`。

`async`、`await` 本质上就是 `Dart` 对异步操作的一个语法糖，可以减少异步调用的嵌套调用，并且由 `async` 修饰后返回一个 `Future`，外界可以以链式调用的方式调用。这个语法是 `JS` 的 `ES7` 标准中推出的，`Dart` 的设计和 `JS` 相同。

下面封装了一个网络请求的异步操作，并且将请求后的 `Response` 类型的 `Future` 返回给外界，外界可以通过 `await` 调用这个请求，并获取返回数据。从代码中可以看到，即便直接返回一个字符串，`Dart` 也会对其进行包装并成为一个 `Future`。

```
Future<Response> dataReqeust() async {
    String requestURL = 'https://jsonplaceholder.typicode.com/posts';
    Client client = Client();
    Future<Response> response = client.get(requestURL);
    return response;
}

Future<String> loadData() async {
    Response response = await dataReqeust();
    return response.body;
}
```

在代码示例中，执行到 `loadData` 方法时，会同步进入方法内部进行执行，当执行到 `await` 时就会停止 `async` 内部的执行，从而继续执行外面的代码。当 `await` 有返回后，会继续从 `await` 的位置继续执行。所以 `await` 的操作，不会影响后面代码的执行。

下面是一个代码示例，通过 `async` 开启一个异步操作，通过 `await` 等待请求或其他操作的执行，并接收返回值。当数据发生改变时，调用 `setState` 方法并更新数据源，`Flutter` 会更新对应的 `Widget` 节点视图。

```
class _SampleAppPageState extends State<SampleAppPage> {
    List widgets = [];

    @override
    void initState() {
        super.initState();
        loadData();
    }

    loadData() async {
        String dataURL = "https://jsonplaceholder.typicode.com/posts";
        http.Response response = await http.get(dataURL);
        setState(() {
            widgets = json.decode(response.body);
        });
    }
}
```

Future

`Future` 就是延时操作的一个封装，可以将异步任务封装为 `Future` 对象。获取到 `Future` 对象后，最简单的方法就是用 `await` 修饰，并等待返回结果继续向下执行。正如上面 `async`、`await` 中讲到的，使用 `await` 修饰时需要配合 `async` 一起使用。

在 `Dart` 中，和时间相关的操作基本都和 `Future` 有关，例如延时操作、异步操作等。下面是一个很简单的延时操作，通过 `Future` 的 `delayed` 方法实现。

```
loadData() {  
  // DateTime.now(), 获取当前时间  
  DateTime now = DateTime.now();  
  print('request begin $now');  
  Future.delayed(Duration(seconds: 1), () {  
    now = DateTime.now();  
    print('request response $now');  
  });  
}
```

`Dart` 还支持对 `Future` 的链式调用，通过追加一个或多个 `then` 方法来实现，这个特性非常实用。例如一个延时操作完成后，会调用 `then` 方法，并且可以传递一个参数给 `then`。调用方式是链式调用，也就代表可以进行很多层的处理。这有点类似于iOS的 `RAC` 框架，链式调用进行信号处理。

```
Future.delayed(Duration(seconds: 1), () {  
  int age = 18;  
  return age;  
}).then((onValue) {  
  onValue++;  
  print('age $onValue');  
});
```

协程

如果想要了解 `async`、`await` 的原理，就要先了解协程的概念，`async`、`await` 本质上就是协程的一种语法糖。协程，也叫作 `coroutine`，是一种比线程更小的单元。如果从单元大小来说，基本可以理解为进程->线程->协程。

任务调度

在弄懂协程之前，首先要明白并发和并行的概念，并发指的是由系统来管理多个IO的切换，并交由CPU去处理。并行指的是多核CPU在同一时间里执行多个任务。

并发的实现由非阻塞操作+事件通知来完成，事件通知也叫做“中断”。操作过程分为两种，一种是CPU对IO进行操作，在操作完成后发起中断告诉IO操作完成。另一种是IO发起中断，告诉CPU可以进行操作。

线程本质上也是依赖于中断来进行调度的，线程还有一种叫做“阻塞式中断”，就是在执行IO操作时将线程阻塞，等待执行完成后再继续执行。但线程的消耗是很大的，并不适合大量并发操作的处理，而通过单线程并发可以进行大量并发操作。当多核CPU出现后，单个线程就无法很好的利用多核CPU的优势了，所以又引入了线程池的概念，通过线程池来管理大量线程。

协程

在程序执行过程中，离开当前的调用位置有两种方式，继续调用其他函数和 `return` 返回离开当前函数。但是执行 `return` 时，当前函数在调用栈中的局部变量、形参等状态则会被销毁。

协程分为无线协程和有线协程，无线协程在离开当前调用位置时，会将当前变量放在堆区，当再次回到当前位置时，还会继续从堆区中获取到变量。所以，一般在执行当前函数时就会将变量直接分配到堆区，而 `async`、`await` 就属于无线协程的一种。有线协程则会将变量继续保存在栈区，在回到指针指向的离开位置时，会继续从栈中取出调用。

async、await原理

以 `async`、`await` 为例，协程在执行时，执行到 `async` 则表示进入一个协程，会同步执行 `async` 的代码块。`async` 的代码块本质上也相当于一个函数，并且有自己的上下文环境。当执行到 `await` 时，则表示有任务需要等待，CPU则去调度执行其他IO，也就是后面的代码或其他协程代码。过一段时间CPU就会轮训一次，看某个协程是否任务已经处理完成，有返回结果可以被继续执行，如果可以被继续执行的话，则会沿着上次离开时指针指向的位置继续执行，也就是 `await` 标志的位置。

由于并没有开启新的线程，只是进行IO中断改变CPU调度，所以网络请求这样的异步操作可以使用 `async`、`await`，但如果是执行大量耗时同步操作的话，应该使用 `isolate` 开辟新的线程去执行。

如果用协程和iOS的 `dispatch_async` 进行对比，可以发现二者是比较相似的。从结构定义来看，协程需要将当前 `await` 的代码块相关的变量进行存储，`dispatch_async` 也可以通过 `block` 来实现临时变量的存储能力。

我之前还在想一个问题，苹果为什么不引入协程的特性呢？后来想了一下，`await` 和 `dispatch_async` 都可以简单理解为异步操作，OC的线程是基于 `RunLoop` 实现的，`Dart` 本质上也是有事件循环的，而且二者都有自己的事件队列，只是队列数量和分类不同。

我觉得当执行到 `await` 时，保存当前的上下文，并将当前位置标记为待处理任务，用一个指针指向当前位置，并将待处理任务放入当前 `isolate` 的队列中。在每个事件循环时都去询问这个任务，如果需要进行处理，就恢复上下文进行任务处理。

Promise

这里想提一下 `JS` 里的 `Promise` 语法，在iOS中会出现很多 `if` 判断或者其他的嵌套调用，而 `Promise` 可以把之前横向的嵌套调用，改成纵向链式调用。如果能把 `Promise` 引入到OC里，可以让代码看起来更简洁，直观。

isolate

`isolate` 是 `Dart` 平台对线程的实现方案，但和普通 `Thread` 不同的是，`isolate` 拥有独立的内存，`isolate` 由线程和独立内存构成。正是由于 `isolate` 线程之间的内存不共享，所以 `isolate` 线程之间并不存在资源抢夺的问题，所以也不需要锁。

通过 `isolate` 可以很好的利用多核CPU，来进行大量耗时任务的处理。`isolate` 线程之间的通信主要通过 `port` 来进行，这个 `port` 消息传递的过程是异步的。通过 `Dart` 源码也可以看出，实例化一个 `isolate` 的过程包括，实例化 `isolate` 结构体、在堆中分配线程内存、配置 `port` 等过程。

`isolate` 看起来其实和进程比较相似，之前请教阿里架构师宗心问题时，宗心也说过“`isolate` 的整体模型我自己的理解其实更像进程，而 `async`、`await` 更像是线程”。如果对比一下 `isolate` 和进程的定义，会发现确实 `isolate` 很像是进程。

代码示例

下面是一个 `isolate` 的例子，例子中新创建了一个 `isolate`，并且绑定了一个方法进行网络请求和数据解析的处理，并通过 `port` 将处理好的数据返回给调用方。

```
loadData() async {
    // 通过spawn新建一个isolate，并绑定静态方法
    ReceivePort receivePort =ReceivePort();
    await Isolate.spawn(dataLoader, receivePort.sendPort);

    // 获取新isolate的监听port
    SendPort sendPort = await receivePort.first;
    // 调用sendReceive自定义方法
    List dataList = await sendReceive(sendPort,
    'https://jsonplaceholder.typicode.com/posts');
    print('dataList $dataList');
}

// isolate的绑定方法
static dataLoader(SendPort sendPort) async{
    // 创建监听port，并将sendPort传给外界用来调用
    ReceivePort receivePort =ReceivePort();
    sendPort.send(receivePort.sendPort);

    // 监听外界调用
    await for (var msg in receivePort) {
        String requestURL =msg[0];
        SendPort callbackPort =msg[1];

        Client client = Client();
        Response response = await client.get(requestURL);
        List dataList = json.decode(response.body);
        // 回调返回值给调用者
        callbackPort.send(dataList);
    }
}

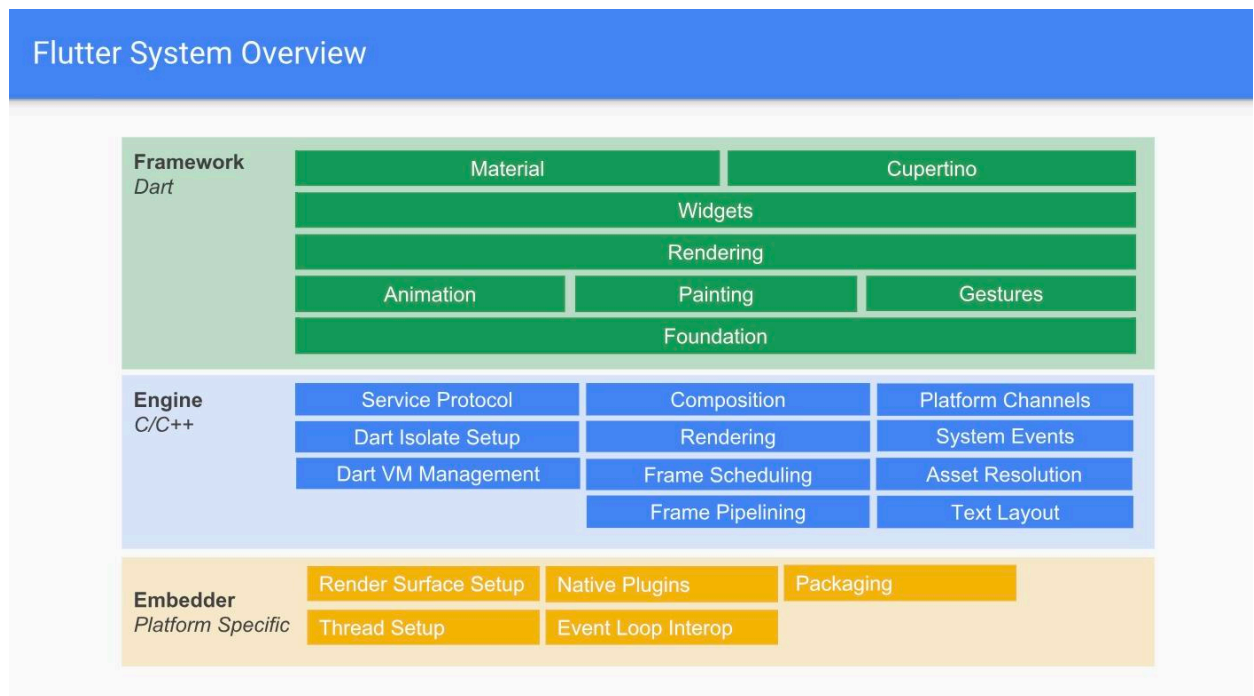
// 创建自己的监听port，并且向新isolate发送消息
Future sendReceive(SendPort sendPort, String url) {
    ReceivePort receivePort =ReceivePort();
    sendPort.send([url, receivePort.sendPort]);
    // 接收到返回值，返回给调用者
    return receivePort.first;
}
```


`isolate` 和iOS中的线程还不太一样，`isolate` 的线程更偏底层。当生成一个 `isolate` 后，其内存是各自独立的，相互之间并不能进行访问。但 `isolate` 提供了基于 `port` 的消息机制，通过建立通信双方的 `sendPort` 和 `receivePort`，进行相互的消息传递，在 `Dart` 中叫做消息传递。

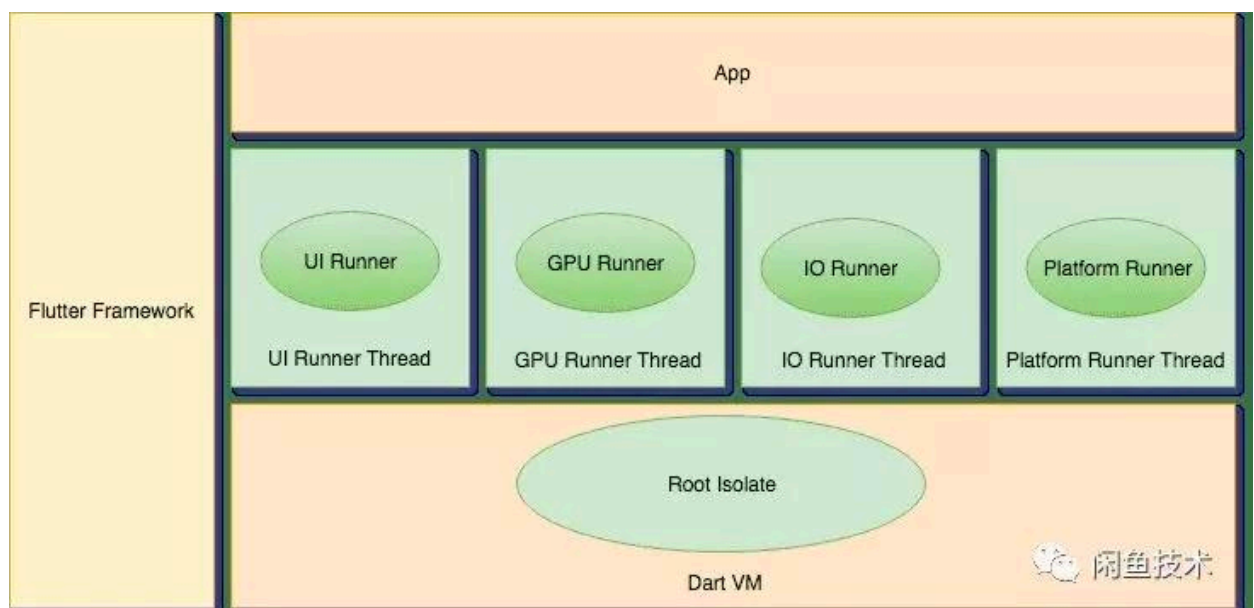
从上面例子中可以看出，在进行 `isolate` 消息传递的过程中，本质上就是进行 `port` 的传递。将 `port` 传递给其他 `isolate`，其他 `isolate` 通过 `port` 拿到 `sendPort`，向调用方发送消息来进行相互的消息传递。

Embedder

正如其名，`Embedder` 是一个嵌入层，将 `Flutter` 嵌入到各个平台上。`Embedder` 负责范围包括原生平台插件、线程管理、事件循环等。



`Embedder` 中存在四个 `Runner`，四个 `Runner` 分别如下。其中每个 `Flutter Engine` 各自对应一个 `UI Runner`、`GPU Runner`、`IO Runner`，但所有 `Engine` 共享一个 `Platform Runner`。



`Runner` 和 `isolate` 并不是一码事，彼此相互独立。以iOS平台为例，`Runner` 的实现就是 `CFRunLoop`，以一个事件循环的方式不断处理任务。并且 `Runner` 不只处理 `Engine` 的任务，还有 `Native Plugin` 带来的原生平台的任务。而 `isolate` 则由 `Dart VM` 进行管理，和原生平台线程并无关系。

Platform Runner

`Platform Runner` 和iOS平台的 `Main Thread` 非常相似，在 `Flutter` 中除耗时操作外，所有任务都应该放在 `Platform` 中，`Flutter` 中的很多API并不是线程安全的，放在其他线程中可能会导致一些 bug。

但例如IO之类的耗时操作，应该放在其他线程中完成，否则会影响 `Platform` 的正常执行，甚至于被 `watchdog` 干掉。但需要注意的是，由于 `Embedder Runner` 的机制，`Platform` 被阻塞后并不会导致页面卡顿。

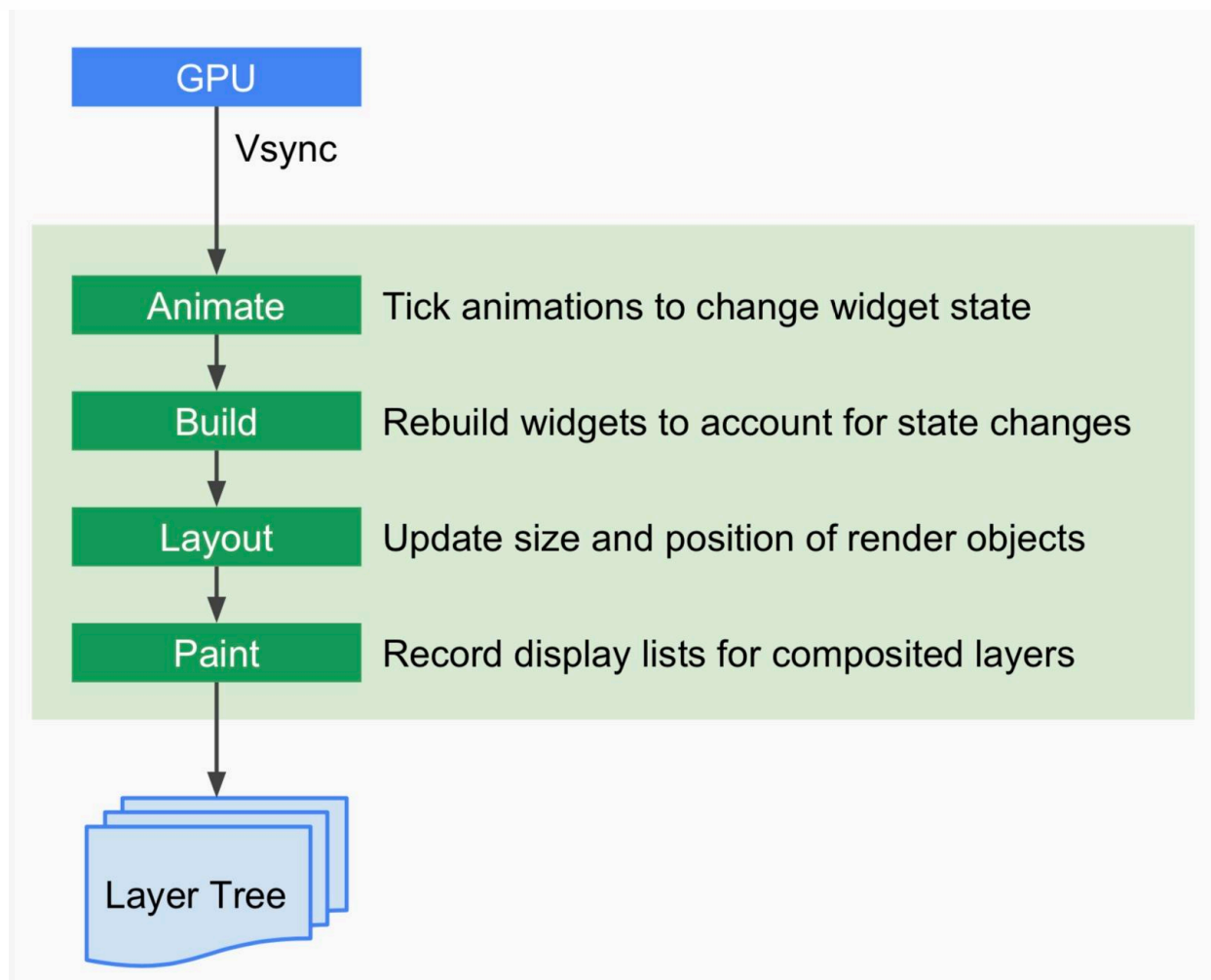
不只是 `Flutter Engine` 的代码在 `Platform` 中执行，`Native Plugin` 的任务也会派发到 `Platform` 中执行。实际上，在原生侧的代码运行在 `Platform Runner` 中，而 `Flutter` 侧的代码运行在 `Root Isolate` 中，如果在 `Platform` 中执行耗时代码，则会卡原生平台的主线程。

UI Runner

`UI Runner` 负责为 `Flutter Engine` 执行 `Root Isolate` 的代码，除此之外，也处理来自 `Native Plugin` 的任务。`Root Isolate` 为了处理自身事件，绑定了很多函数方法。程序启动时，`Flutter Engine` 会为 `Root` 绑定 `UI Runner` 的处理函数，使 `Root Isolate` 具备提交渲染帧的能力。

当 `Root Isolate` 向 `Engine` 提交一次渲染帧时，`Engine` 会等待下次vsync，当下次vsync到来时，由 `Root Isolate` 对 `Widgets` 进行布局操作，并生成页面的显示信息的描述，并将信息交给 `Engine` 去处理。

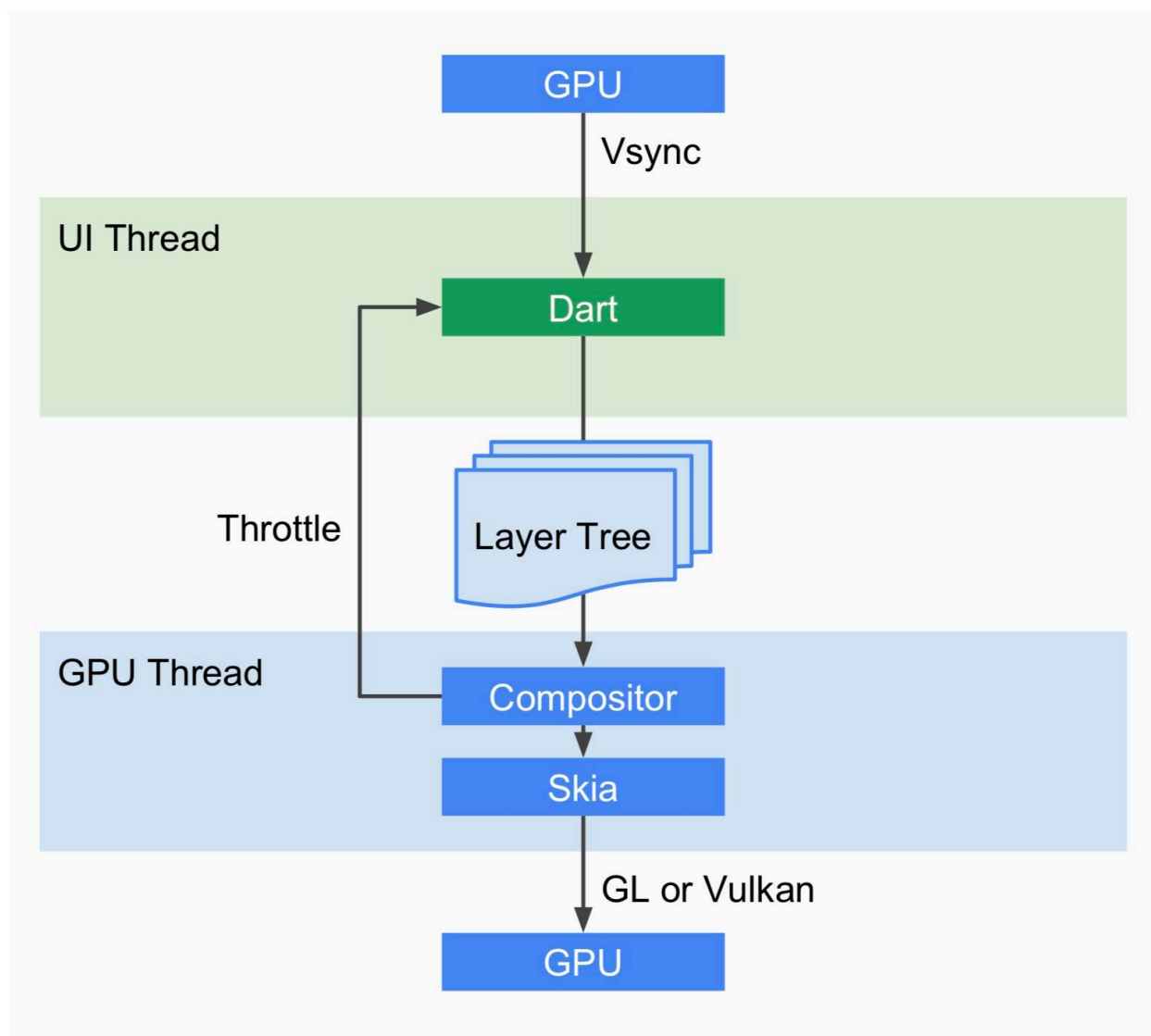
由于对 `widgets` 进行 `layout` 并生成 `layer tree` 是 `UI Runner` 进行的，如果在 `UI Runner` 中进行大量耗时处理，会影响页面的显示，所以应该将耗时操作交给其他 `isolate` 处理，例如来自 `Native Plugin` 的事件。



GPU Runner

`GPU Runner` 并不直接负责渲染操作，其负责GPU相关的管理和调度。当 `layer tree` 信息到来时，`GPU Runner` 将其提交给指定的渲染平台，渲染平台是Skia配置的，不同平台可能有不同的实现。

`GPU Runner` 相对比较独立，除了 `Embedder` 外其他线程均不可向其提交渲染信息。



IO Runner

一些 `GPU Runner` 中比较耗时的操作，就放在 `IO Runner` 中进行处理，例如图片读取、解压、渲染等操作。但是只有 `GPU Runner` 才能对GPU提交渲染信息，为了保证 `IO Runner` 也具备这个能力，所以 `IO Runner` 会引用 `GPU Runner` 的 `context`，这样就具备向GPU提交渲染信息的能力。

iOS接入Flutter

在进行 `iOS` 和 `Flutter` 的混编时，`iOS` 比 `Android` 的接入方式略复杂，但也还好。现在市面上有不少接入 `Flutter` 的方案，但大多数都是千篇一律相互抄的，没什么意义。

进行 `Flutter` 混编之前，有一些必要的文件。

1. `xcode_backend.sh` 文件，在配置 `flutter` 环境的时候由 `Flutter` 工具包提供。
2. `xcconfig` 环境变量文件，在 `Flutter` 工程中自动生成，每个工程都不一样。



xcconfig文件

`xcconfig` 是 `Xcode` 的配置文件，`Flutter` 在里面配置了一些基本信息和路径，接入 `Flutter` 前需要先将 `xcconfig` 接入进来，否则一些路径等信息将会出错或找不到。

Flutter 的 xcconfig 包含三个文件，Debug.xcconfig、Release.xcconfig、Generated.xcconfig，需要将这些文件配置在下面的位置，并且按照不同环境配置不同的文件。

Project -> Info -> Development Target -> Configurations

▼ Configurations

Name	Based on Configuration File
▼ Debug	2 Configurations Set
▼  Runner	Debug ⚙
 Runner	Debug ⚙
▼ Release	2 Configurations Set
▼  Runner	Release ⚙
 Runner	Release ⚙
▼ Profile	2 Configurations Set
▼  Runner	Release ⚙
 Runner	Release ⚙

有些比较大的工程中已经在 Configurations 中设置了 xcconfig 文件，由于每个 Target 的一种环境只能配置一个 xcconfig 文件，所以可以在已有的 xcconfig 文件中 import 引入 Generated.xcconfig 文件，并且不需要区分环境。

脚本文件

xcode_backend.sh 脚本文件用来构建和导出 Flutter 产物，这是 Flutter 开发包为我们默认提供的。需要在工程 Target 的 Build Phases 加入一个 Run Script 文件，并将下面的脚本代码粘贴进去。需要注意的是，不要忘记前面的 /bin/sh 操作，否则会导致权限错误。

```
/bin/sh "$FLUTTER_ROOT/packages/flutter_tools/bin/xcode_backend.sh" build
/bin/sh "$FLUTTER_ROOT/packages/flutter_tools/bin/xcode_backend.sh" embed
```

在 xcode_backend.sh 中有三个参数类型，build、thin、embed，thin 没有太大意义，其他两个则负责构建和导出。

混合开发

随后可以对 Xcode 工程进行编译，这时候肯定会报错的。但是不要慌张，报错后我们在工程主目录下会发现一个名为 Flutter 的文件夹，其中会包含两个 framework，这个文件夹就是 Flutter 的编译产物，我们将这个文件夹整体拖入项目中即可。

这时候就可以在 iOS 工程中添加 Flutter 代码了，下面是详细步骤。

1. 将 AppDelegate 的集成改为 FlutterAppDelegate，并且需要遵循 FlutterAppLifecycleProvider 代理。

```
#import <Flutter/Flutter.h>
#import <UIKit/UIKit.h>

@interface AppDelegate : FlutterAppDelegate <FlutterAppLifecycleProvider>

@end
```

2. 创建一个 `FlutterPluginAppLifeCycleDelegate` 的实例对象，这个对象负责管理 `Flutter` 的生命周期，并从 `Platform` 侧接收 `AppDelegate` 的事件。我直接将其声明为一个属性，在 `AppDelegate` 中的各个方法中，调用其方法进行中转操作。

```
- (BOOL)application:(UIApplication *)application willFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
    [self.lifeCycleDelegate application:application
willFinishLaunchingWithOptions:launchOptions];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application {
    [self.lifeCycleDelegate applicationWillResignActive:application];
}

- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    [self.lifeCycleDelegate application:application openURL:url
sourceApplication:sourceApplication annotation:annotation];
    return YES;
}
```

3. 随后即可加入 `Flutter` 代码，加入的方式也很简单，直接实例化一个 `FlutterViewController` 控制器即可，也不需要传其他参数进去(这里先不考虑多实例的问题)。

```
FlutterViewController *flutterViewController = [[FlutterViewController alloc]
init];
```

`Flutter` 将其看做是一个画布，实例化一个画布上去之后，任何操作其实都是在当前页面完成的。

常见错误

到这个步骤集成操作就已经完成，但是很多人在集成过程中会遇到一些错误，下面是一些常见错误。

1. 路径错误，读取不到 `xcode_backend.sh` 文件等。这是因为环境变量 `FLUTTER_ROOT` 没有获取到，`FLUTTER_ROOT` 配置在 `Generated.xcconfig` 中，可以看一下这个文件是不是配置的有问题。
2. `lipo info *** arm64` 类似这样的错误，一般都是因为 `xcode_backend.sh` 脚本导致的，可以检查一下 `FLUTTER_ROOT` 环境变量是否正确。
3. 下面这种问题一般都是因为权限导致的，可以查看 `Build Phases` 的脚本写的是不是有问题。

```
*** /flutter_tools/bin/xcode_backend.sh: Permission denied
```

混合开发

在进行混编过程中，Flutter 有一个很大的优势，就是如果 Flutter 代码出问题，不会导致原生应用的崩溃。当 Flutter 代码出现崩溃时，会在屏幕上显示错误信息。

在开发过程中经常会涉及到网络请求和持久化的问题，如果混编的话可能会涉及到写两套逻辑。例如网络请求有一些公共参数，或返回数据的统一处理等，如果维护两套逻辑的话会容易出问题。所以建议将网络请求和持久化操作都交给 Platform 处理，Flutter 侧只负责向 Platform 请求并拿来使用即可。

这个过程就涉及到两端数据交互的问题，Flutter 对于混编给出了两套方案，MethodChannel 和 EventChannel。从名字上来看，一个是方法调用，另一个是事件传递。但实际开发过程中，只需要使用 MethodChannel 即可完成所有需求。

Flutter to Native

下面是 Flutter 调用 Native 的代码，在 Native 中通过 FlutterMethodChannel 设置指定的回调代码，并且在接收参数并处理。由 Flutter 通过 MethodChannel 对 Native 发起调用，并传入对应的参数。

代码中在 Flutter 侧构建好数据模型，然后调用 MethodChannel 的 invokeMethod，会触发 Native 的回调。Native 拿到 Flutter 传过来的数据，进行解析并执行播放操作，随后会把播放的状态码回调给 Flutter 侧，交互完成。


```

import 'package:flutter/services.dart';

Future<Null> playVideo() async{
  var methodChannel = MethodChannel('flutterChannelName');
  Map params = {'playID' : '302998298', 'duration' : '2520', 'name' : '三生三世十里桃花'};
  String result;
  result = await methodChannel.invokeMethod('PlayAlbumVideo', params);

  String playID   = params['playID'];
  String duration = params['duration'];
  String name     = params['name'];
  showCupertinoDialog(context: context, builder: (BuildContext context){
    return CupertinoAlertDialog(
      title: Text(result),
      content: Text('name:$name playID:$playID duration:$duration'),
      actions: <Widget>[
        FlatButton(
          child: Text('确定'),
          onPressed: (){
            Navigator.pop(context);
          },
        )
      ],
    );
  });
}

```

```

NSString *channelName = @"flutterChannelName";
FlutterMethodChannel *methodChannel = [FlutterMethodChannel
methodChannelWithName:channelName binaryMessenger:flutterVC];
[methodChannel setMethodCallHandler:^(FlutterMethodCall * _Nonnull call,
FlutterResult _Nonnull result) {
  if ([call.method isEqualToString:@"PlayAlbumVideo"]) {
    NSDictionary *params = call.arguments;

    VideoPlayerModel *model = [[VideoPlayerModel alloc] init];
    model.playID = [params stringForKey:@"playID"];
    model.duration = [params stringForKey:@"duration"];
    model.name = [params stringForKey:@"name"];
    NSString *playStatus = [SVHistoryPlayUtil playVideoWithModel:model

showPlayerVC:self.flutterVC];

    result([NSString stringWithFormat:@"播放状态 %@", playStatus]);
  }
}];

```

Native to Flutter

Native 调用 Flutter 的代码和 Flutter 调用 Native 的基本类似，只是调用和设置回调的角色不同。同样的，Flutter 由于要接收 Native 的消息回调，所以需要注册一个回调，由 Native 发起对 Flutter 的调用并传入参数。

Native 和 Flutter 的相互调用都需要设置一个名字，每一个名字对应一个 MethodChannel 对象，每一个对象可以发起多次调用，不同调用以 invokeMethod 做区分。

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

@override
void initState() {
  super.initState();

  MethodChannel methodChannel = MethodChannel('nativeChannelName');
  methodChannel.setMethodCallHandler(callbackHandler);
}

Future<dynamic> callbackHandler(MethodCall call) {
  if(call.method == 'requestHomeData') {
    String title = call.arguments['title'];
    String content = call.arguments['content'];
    showCupertinoDialog(context: context, builder: (BuildContext context){
      return CupertinoAlertDialog(
        title: Text(title),
        content: Text(content),
        actions: <Widget>[
          FlatButton(
            child: Text('确定'),
            onPressed: (){
              Navigator.pop(context);
            },
          ),
        ],
      );
    });
  }
}
```

```
NSString *channelName = @"nativeChannelName";
FlutterMethodChannel *methodChannel = [FlutterMethodChannel
methodChannelWithName:channelName binaryMessenger:flutterVC];
[RequestManager requestWithURL:url success:^(NSDictionary *result) {
  [methodChannel invokeMethod:@"requestHomeData" arguments:result];
}];
```

调试工具集

在 iOS 和 Android 开发中，各自的编译器都提供了很好的调试工具集，方便进行内存、性能、视图等调试。Flutter 也提供了调试工具和命令，下面基于 VSCode 编译器来讲一下 Flutter 调试，相对而言 Android Studio 提供的调试功能可能会更多一些。

性能调试

VSCode 支持一些简单的命令行调试指令，在程序运行过程中，在 Command Palette 命令行面板中输入 performance，并选择 Toggle Performance Overlay 命令即可。此命令有一个要求就是需要 App 在运行状态。



随后会在界面上出现一个性能面板，这个页面分为两部分，GPU线程和UI线程的帧率。每个部分分为三个横线，代表着不同的卡顿层级。如果是绿色则表示不会影响界面渲染，如果是红色则有可能会影响界面的流畅性。如果出现红色线条，则表示当前执行的代码需要优化。

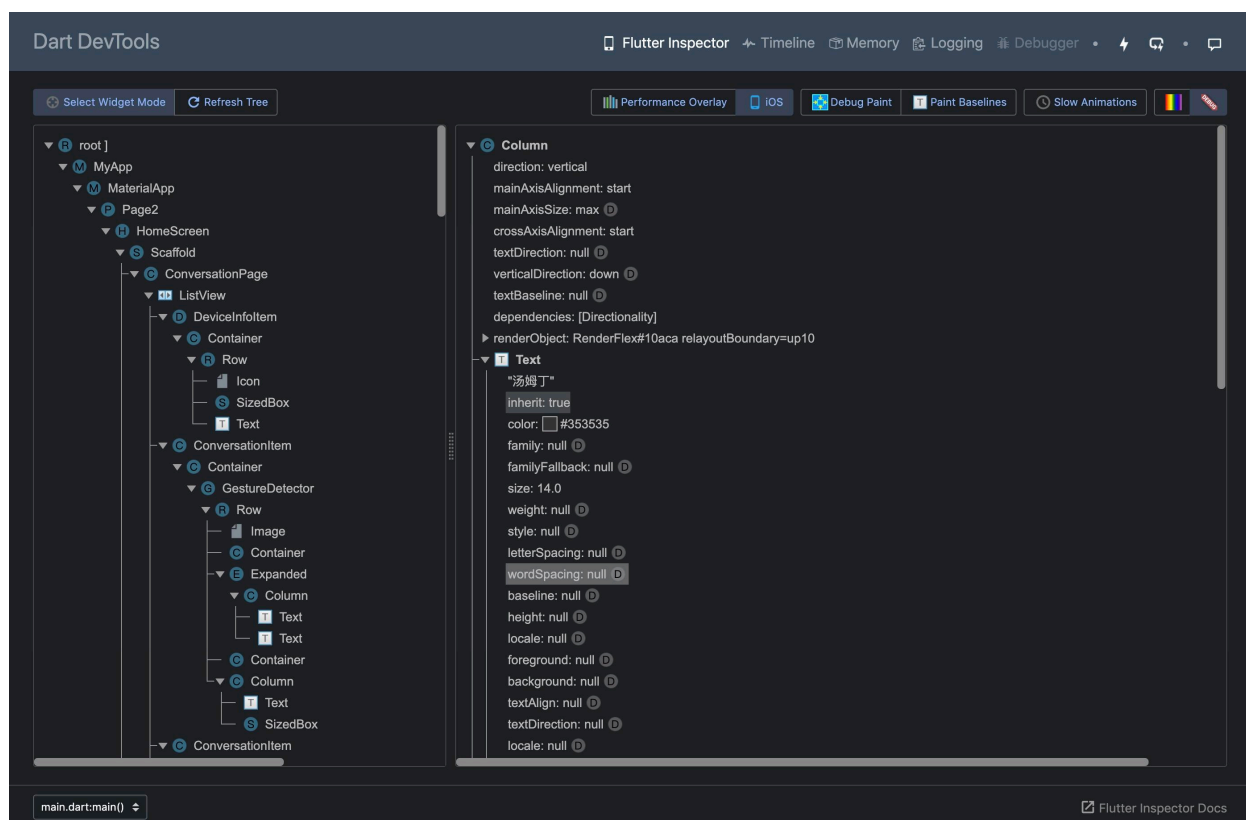
Dart DevTools

VSCode 为 Flutter 提供了一套调试工具集-Dart DevTools，这套工具集功能非常全，包含性能、UI、热更新、热重载、log日志等很多功能。

安装 Dart DevTools 后，在App运行状态下，可以在 VSCode 的右下角启动这个工具，工具会以网页的形式展现，并且可以控制App。

主界面

下面是 Dart DevTools 的主界面，我运行的是一个界面类似于微信的App。从 Inspector 中可以看到页面的视图结构，Android Studio 也有类似的功能。页面整体是一个树形结构，并且选中某一个控件后，会在右侧展示出控件的变量值，例如 frame、color 等，这个功能非常实用。



我运行的设备是 Xcode 模拟器，如果想切换 Android 的 Material Design，点击上面的 iOS 按钮即可直接切换设备。刚才上面说到的查看内存的性能面板，点击 iOS 按钮旁边的 Performance Overlay 即可出现。

Select Widget

如果想知道在 Dart DevTools 中选择的节点，具体对应哪个控件，可以选择 Select Widget Mode 使屏幕上被选中的控件高亮。



Debug Paint

点击 `Debug Paint` 可以让每个控件都高亮，通过这个模式可以看到 `ListView` 的滑动方向，以及每个控件的大小及控件之间的距离。

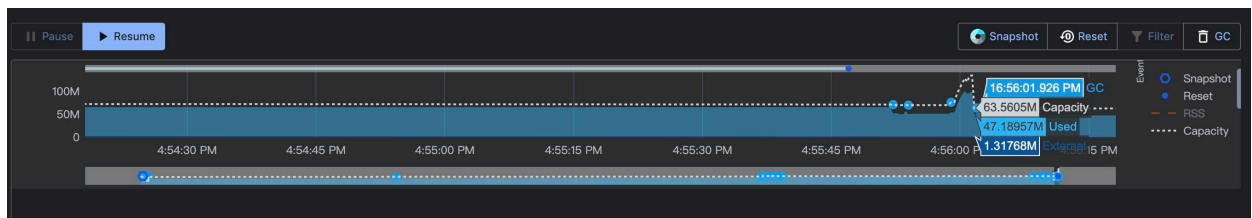




除此之外，还可以选择 `Paint Baseline` 使所有控件的底线高亮，功能和 `Debug Paint` 类似，不做叙述。

Memory

`Dart DevTools` 中提供的内存调试工具更加直观，可以实时显示内存使用情况。在刚开始运行时，我们发现一个内存峰值，把鼠标放上去可以看到具体的内存使用情况。内存会有具体分类，`Used`、`GC` 等。



`Dart DevTools` 的内存工具还是不够完美，`Xcode` 可以选择某段内存，看到这块内存中涉及到主要堆栈调用，并且点击调用栈可以跳转到 `Xcode` 对应的代码中，而 `Dart DevTools` 还不具备这个功能，可能和 `Web` 的展示形式有关系。

内存管理 `Flutter` 使用的是 `GC`，回收速度可能不是很快，`iOS` 中的 `ARC` 则是基于引用计数立即回收的。还有很多其他的功能，这里就不一一详细叙述了，各位同学可以自己探索。

多实例

项目中是通过实例化 `FlutterViewController` 控制器来显示 `Flutter` 界面的，整个 `Flutter` 页面可以理解为一个画布，通过页面不断的变化，改变画布上的东西。所以，在单实例的情况下，`Flutter` 页面中间不能插入原生页面。

这时候如果我们想在多个地方展示 `Flutter` 页面，而这些页面并不是 `Flutter -> Flutter` 的连贯跳转形式，那怎么来实现这个场景呢？`Google` 的建议是创建 `Flutter` 的多实例，并通过传入不同的参数实例化不同的页面。但这样会造成很严重的内存问题，所以并不能这么做。

Router

如果不能真正创建多个实例对象，那就需要通过其他方式来实现多实例。`Flutter` 页面显示其实并不是跟着 `FlutterVC` 走的，而是跟着 `FlutterEngine` 走的。所以在创建一次 `FlutterVC` 之后，就将 `FlutterEngine` 保存下来，在其他位置创建 `FlutterVC` 时直接通过 `FlutterEngine` 的方式创建，并且在创建后进行跳转操作。

在进行页面切换时，通过 `channelMethod` 调用 `Flutter` 侧的路由切换代码，并将切换后的新页面 `FlutterVC` 添加到 `Native` 上。这种实现方式，就是通过 `Flutter` 的 `Router` 的方式实现的，下面将会介绍 `Router` 的两种表现形式，静态路由和动态路由。

静态路由

静态路由是 `MaterialApp` 提供的一个 `API`，`routes` 本质上是一个 `Map` 对象，其组成结构是 `key` 是调用页面的唯一标识符，`value` 就是对应页面的 `Widget`。

在定义静态路由时，可以在创建 `Widget` 时传入参数，例如实例化 `ContactWidget` 时就可以传入对应的参数过去。

```
void main() {
  runApp(
    MaterialApp(
      home: Page2(),
      routes: {
        'page1': (_) => Page1(),
        'page2': (_) => Page2()
      },
    ),
  );
}

class Page1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ContactWidget();
  }
}

class Page2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return HomeScreen();
  }
}
```

进行页面跳转时，通过 `Navigator` 进行调用，每次调用都会重新创建对应的 `Widget`。进行调用时 `pushNamed` 函数会传入一个参数，这个参数就是定义 `Map` 时对应页面的 `key`。

```
Navigator.of(context).pushNamed('page1');
```

动态路由

静态路由的方式并不是很灵活，相对而言动态路由更加灵活。动态路由不需要预先设定 `routes`，直接调用即可。和普通 `push` 不同的是，动态路由在 `push` 时通过 `PageRouteBuilder` 来构建 `push` 对象，在 `Builder` 的构建方法中执行对应的页面跳转操作即可。

结合之前说的 `channelMethod`，就是在 `channelMethod` 对应的 `Callback` 回调中，执行 `Navigator` 的 `push` 函数，接收 `Native` 传递过来的参数并构建对应的 `Widget` 页面，将 `Widget` 返回给 `Builder` 即可完成页面跳转操作。所以说动态路由的方式非常灵活。

无论是通过静态路由还是动态路由的方式创建，都可以通过 `then` 函数接收新页面返回时的返回值。

```
Navigator.of(context).push(PageRouteBuilder(  
  pageBuilder: (BuildContext context, Animation<double> animation,  
  Animation<double> secondaryAnimation) {  
    return ContactWidget('next page value');  
  }  
  transitionsBuilder: (BuildContext context, Animation<double> animation,  
  Animation<double> secondaryAnimation, Widget child) {  
    return FadeTransition(  
      child: child,  
      opacity: animation,  
    );  
  }  
)).then((onValue){  
  print('pop的返回值 $onValue');  
});
```

但动态路由的跳转方式也有一些问题，会导致动画失效。所以需要重写 `Builder` 的 `transitionsBuilder` 函数，来自定义转场动画。

无论是通过静态路由还是动态路由的方式创建，都会存在一些问题。由于每次都是新建 `Widget`，所以在创建时会有黑屏的问题。而且每次创建的话，都会丢失当前页面上次的上下文状态，每次进来都是一个新页面。