Flutter与原生工程的混合开发

原创 拉维 iOS小生活 2021-11-16 10:40

收录于话题
#Flutter
58个 >

实际上，Flutter与原生的混合开发，就分为两大类：

- Flutter工程里面包原生工程，即Flutter项目调用原生的某些功能
- 原生工程里面包含Flutter模块

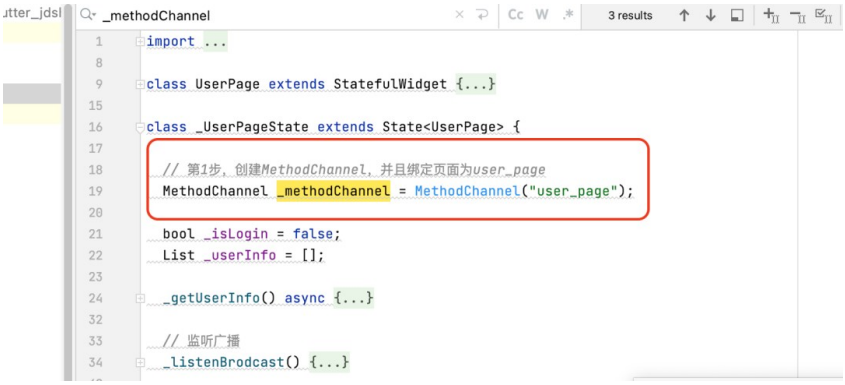
上述这两大类都是可以实现的，技术层面没有任何问题。但是我并不建议在Flutter页面和原生页面之间来回穿插切换，原因如下：

1. Flutter对自己的定位是一个完整的应用程序，这一点从MaterialApp这个Widget的命名上就能看出来，它并不甘心只做某一块功能页面的开发，虽然它给开发者留出了原生加载某一些Flutter页面的API调用。
2. 原生调用Flutter页面会调起FlutterEngine，这是很占用内存的，比较耗性能。
3. 原生调用Flutter会新生成一个FlutterViewController，大概占用80M的内存，使用完毕之后只有很小一部分会被销毁，这就导致了内存泄漏，这一点是Flutter官方已经承认了的。

Flutter项目调用原生的某些功能

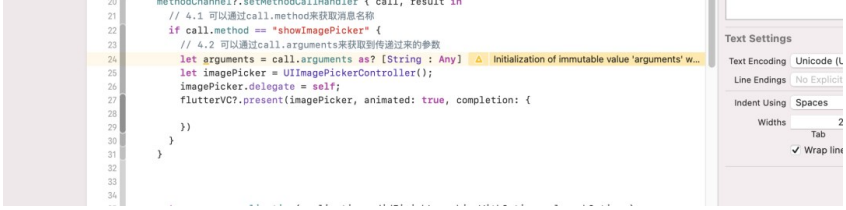
Flutter给原生工程发消息

第1步，在Flutter工程中创建MethodChannel，并且给该channel绑定页面或者功能Id。

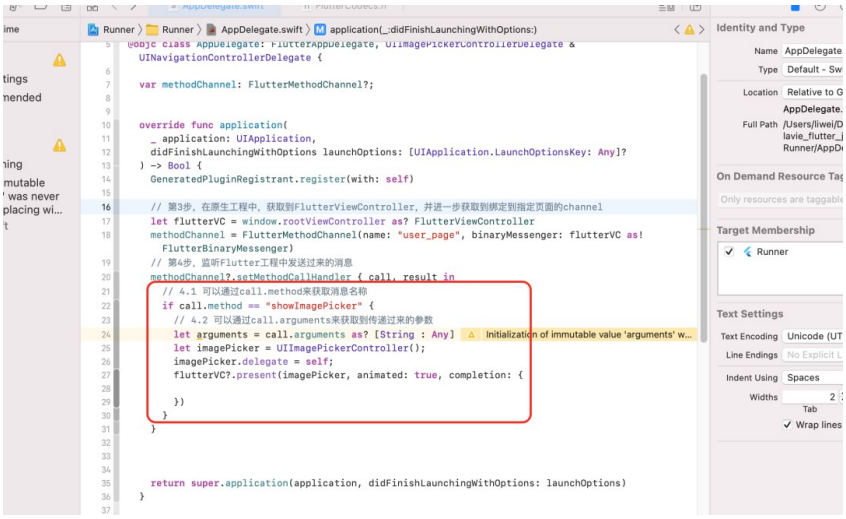


第2步，在Flutter工程中，通过第1步创建的channel给原生发送消息，发送消息的时候必须写明消息名，并且可以携带参数。

第3步，在原生工程中，获取到FlutterViewController，然后进一步获取到绑定到指定页面的channel。

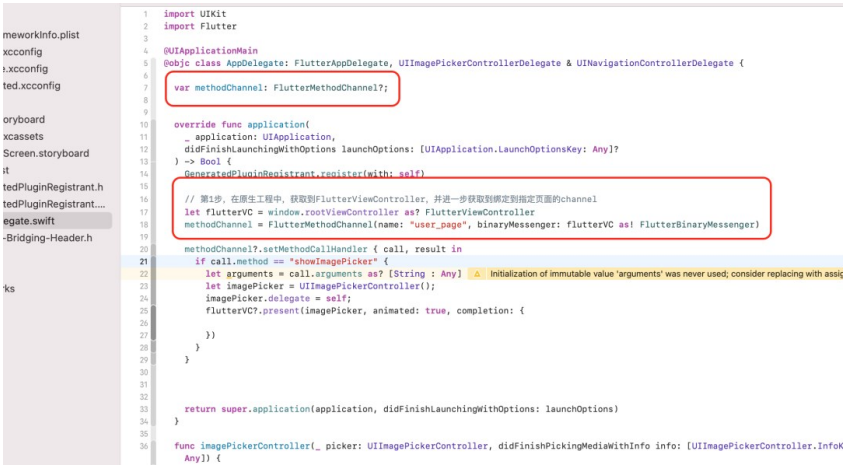


第4步，在原生工程中，监听Flutter中发送过来的消息。



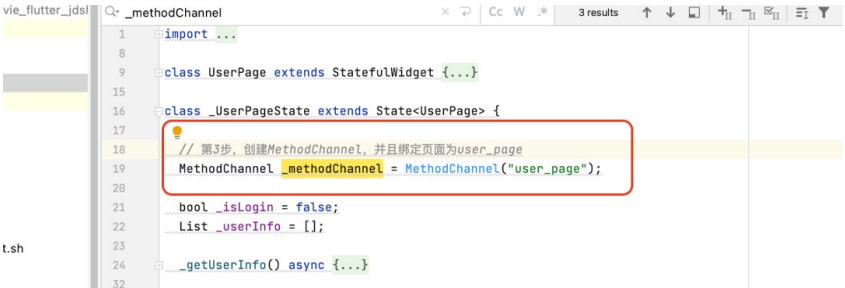
原生给Flutter发送消息

第1步，在原生工程中，获取到FlutterViewController，并进一步获取到绑定到指定页面或者功能模块的channel。



第2步，在原生工程中，通过第1步获取到的channel给Flutter发送消息，其中消息名称必传，而且可以携带arguments参数。

第3步，在Flutter工程中，创建指定页面或者功能的MethodChannel。



```
33    // 监听广播
34    _listenBrodcast() {...}
40
41    @override
42    void initState() {
```

第4步，在Flutter工程中，通过channel来监听原生端发送过来的消息，其中既可以获取到消息名，也可以获取到传递过来的参数。

```
43    super.initState();
44    _getUserInfo();
45    _listenBrodcast();
46
47    // 第4步，监听原生端发送过来的消息
48    _methodChannel.setMethodCallHandler((call) async {
49      if (call.method == "imagePath") {
50        print(call.arguments);
51      }
52    });
53  }
54
55  @override
56  Widget build(BuildContext context) {...}
```

实际上，在Flutter项目中调用原生的某些功能，有很多的第三方插件可以实现，并且这些插件都很好用。比如，如果我们要调用原生的相册或者相机，那么就可以使用image_picker这个第三方插件。实际上，如果是在Flutter项目中调用原生的某些功能，我们也是优先选择使用第三方插件，原因是什么呢？原因就在于，一个Flutter开发工程师可能对于iOS原生和安卓原生都不了解，这样的话，让他直接在原生工程中写原生代码，实际上是比较为难的。

原生工程里面包含Flutter模块

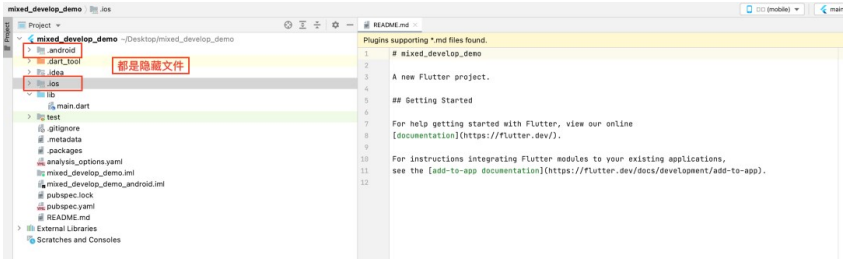
上面讲了在Flutter项目中调用某些原生的功能，实际上，这也是最纯正的Flutter用法。因为Flutter自身的定位就是一个独立的完整的应用程序，无论是从他的Widget命名还是从它的设计（比如有自己独立的渲染引擎）都可以看出来。对于一些小型的或者新起的项目，使用Flutter工程包原生功能的这种方式还是比较合适的。

还有一种方式是，原生工程里面包含Flutter功能模块，这种方式是比较耗性能的，会吃内存并且会导致内存泄漏，所以对于一些小型项目如果采用这种方式的话，会得不偿失。但是对于一些大型项目，如果想要其中一些功能改造成Flutter，或者新的需求使用Flutter去做，此时采用原生工程包含Flutter模块的方式还是比较合适的。

在原生工程中跳转到Flutter页面

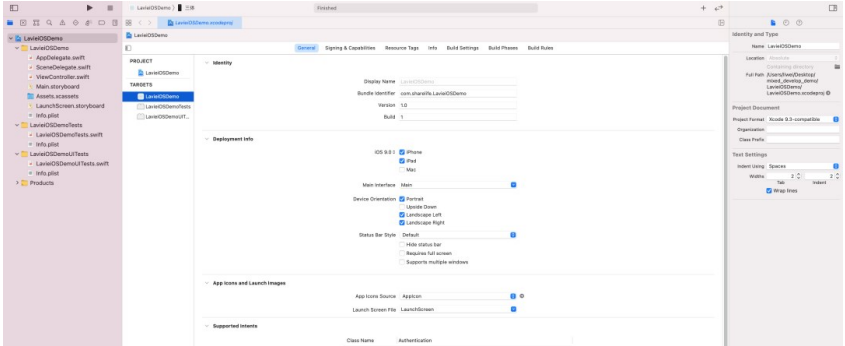
接下来我们就来看一下如何在原生工程中引入Flutter模块。

第1步，创建一个Flutter-Module，创建好之后打开，目录如下：

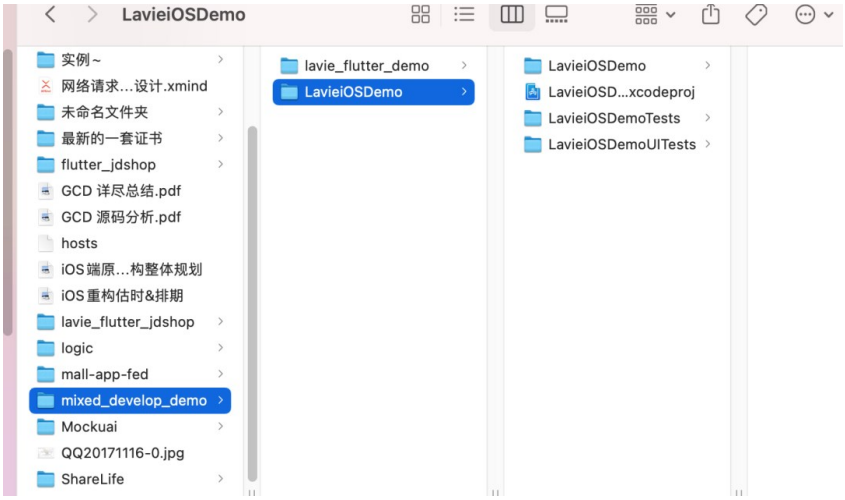


我们发现，module工程里面也是有一个android和一个ios文件夹的，只不过跟Application工程不同的是，module工程的android和ios文件夹名称前面都有一个.，这说明该文件夹是一个隐藏文件夹。那么为什么module工程的android和ios文件夹是隐藏文件夹呢？因为这两个文件夹下面的原生工程完全是作为测试使用的，**方便开发人员在module开发过程中即时测试**，不然的话还得集成到主原生工程才能看到测试效果（这样就比较麻烦了）。

第2步，创建一个纯iOS原生项目

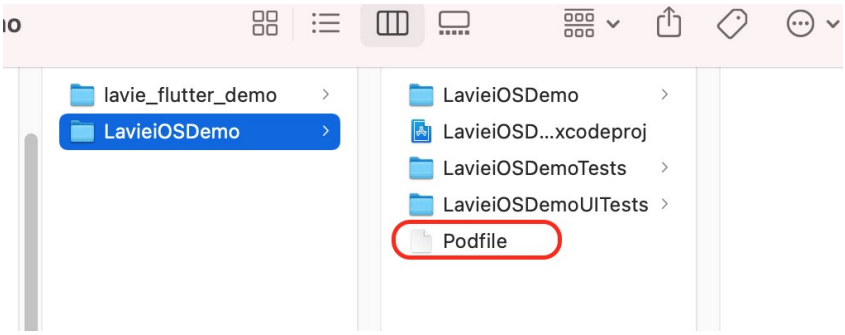


需要注意的是，FlutterModule和iOS原生工程要在同一个目录下

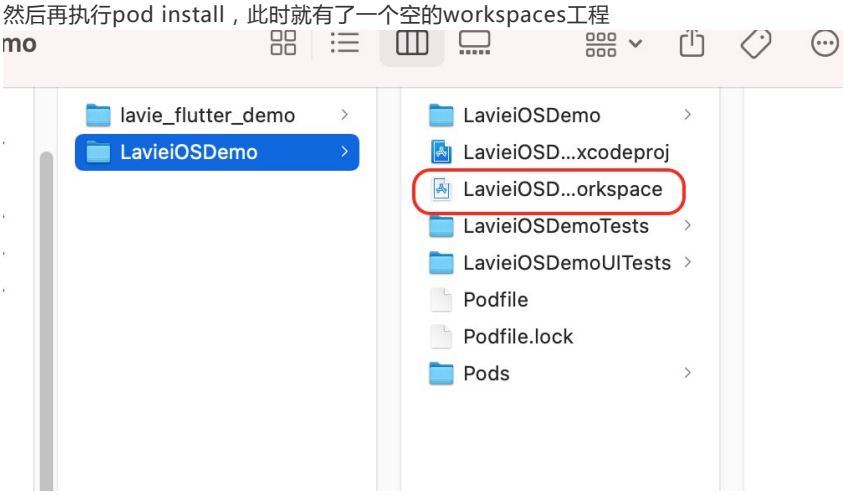


第3步，将FlutterModule与原生工程联系在一起

来到LavieiOSDemo文件夹，终端定位到该文件夹下，然后执行pod init命令，之后该文件夹下就会多了一个Podfile文件



然后再执行pod install，此时就有了一个空的workspaces工程

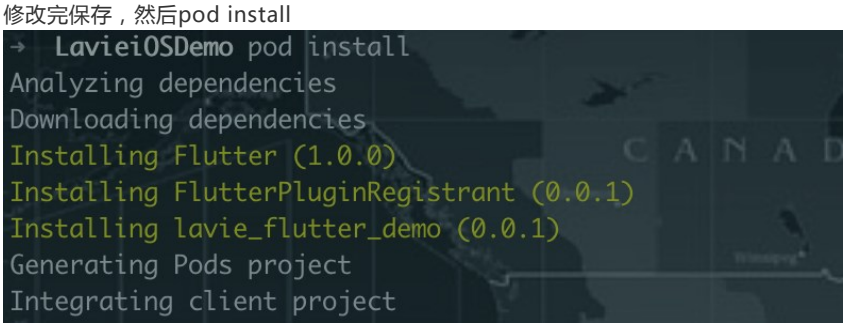


然后打开Podfile，按照如下格式进行修改：

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

flutter_application_path = '../lavie_flutter_demo' # Flutter工程的相对路径
。../表示的是当前目录回退出去，也就是当前目录的上一级目录
load File.join(flutter_application_path, '.ios', 'Flutter', 'podhelper.r
b')

target 'LavieiOSDemo' do
  # Comment the next line if you don't want to use dynamic frameworks
  install_all_flutter_pods(flutter_application_path)
  use_frameworks!

  # Pods for LavieiOSDemo

  target 'LavieiOSDemoTests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'LavieiOSDemoUITests' do
```

```
    # Pods for testing
  end

end
```

修改完保存，然后pod install



这样就将FlutterModule给引入到iOS原生工程里面了。

第4步，在原生工程中展示Flutter页面

```
}

@IBAction func buttonClicked(_ sender: Any) {
  let flutterVC = FlutterViewController()
  self.present(flutterVC, animated: true) {}
}


}
```

这样，就可以在原生工程里面看到Flutter页面的内容啦~~~

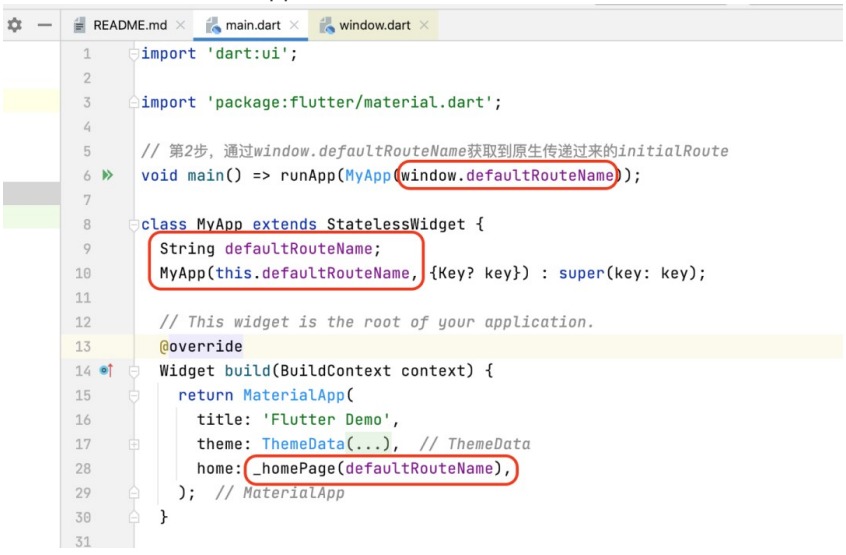
需要注意的是，如果你修改了Flutter页面的内容，但是在原生工程中重新运行之后没有展示出来，那么就Clean一下再重新运行，之后就可以了。

在原生工程中跳转到指定的Flutter页面

在原生工程中是可以指定跳转到Flutter模块的哪一个页面的，步骤如下。

第1步，在原生工程中，初始化FlutterViewController的时候，将initialRoute参数传入。

第2步，在Flutter工程中，最顶层（MaterialApp的外层），获取到原生端传递过来的initialRoute，并传入MaterialApp。



第3步，通过获取到的routeName来决定具体是展示哪一个Flutter页面。

```dart
String defaultRouteName;
MyApp(this.defaultRouteName, {Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(...),  // ThemeData
    home: _homePage(defaultRouteName),
  );  // MaterialApp
}

// 第3步，通过RouteName来决定具体展示哪个页面
Widget _homePage(String routeName) {
  switch (routeName) {
    case "one":
      return MyHomePage(title: '''Lavie's 第一个 Page''');
    default:
      return MyHomePage(title: '''Lavie's 第二个 Page''');
  }
}
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);

  //...
```

以上这种方式，确实是可以实现从原生工程中跳转到指定的Flutter页面，但是它有很大的一个弊端，就是非常吃内存！！！
一个FlutterViewController，它被创建出来之后，就不会被销毁，而且一个FlutterViewController要占80M左右的内存空间，如果是以上述方式在原生工程中点开Flutter页面的话，多点几个Flutter页面，应用程序估计就会内存爆满！！！

因此，我不建议在原生工程中每次跳入Flutter页面的时候，都重新创建FlutterViewController！！！

在原生工程中高性能地跳转到指定的Flutter页面

上面的这种方式，每跳入一个新的Flutter页面就会重新创建FlutterVC，很吃内存，因此我们就想，可否将FlutterVC和FlutterEngine设置成单例，这样全局共用一份，就可以极大地减少内存使用率。

第1步，在原生工程中，创建一个FlutterEngine单例，并且创建完了之后就立马执行该Engine。

```dart
// 第1步，实例化一个共享的Engine，最好是做成单例
lazy var flutterEngine: FlutterEngine = {
  let engine = FlutterEngine(name: "lavie")
  engine.run() // 让这个Engine提前运行起来
  return engine
}()
```

第2步，在原生工程中，通过传入Engine的方式创建一个FlutterViewController单例。

```dart
// 第2步，实例化一个共享的FlutterVC，最好是做成单例
lazy var flutterViewController: FlutterViewController = {
  return FlutterViewController(engine: flutterEngine, nibName: nil, bun
dle: nil)
}()
```

需要注意的是，在第1步和第2步的代码示例中，我并不是创建的单例，在ni自己封装的时候，可以将FlutterVC 和 Engine都封装成单例。

第3步，在原生工程中的需要跳转到Flutter页面的地方，通过MethodChannel进行传参，具体步骤如下:
（1）创建一个`FlutterMethodChannel`，在其构造方法中可以传入channel的名称，以及flutterVC。
我们可以以页面或者功能模块来定义不同的channel的维度。
（2）通过`methodChannel.invokeMethod`来给该通道发送消息以及传递参数
（3）跳转`flutterViewController`
（4）监听Flutter中传递过来的消息，并做对应的响应

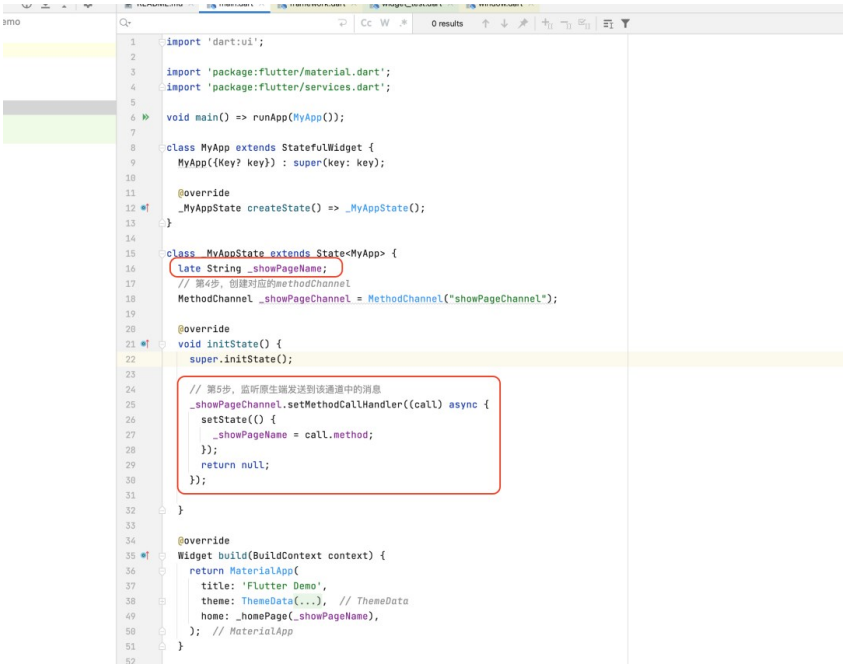
```dart
// 第3步，通过FlutterMethodChannel来指定跳转到哪个页面，并且接收Flutter页面
中传递过来的消息
```

```swift
    // 跳转到指定的Flutter页面
    let methodChannel = FlutterMethodChannel(name: "showPageChannel", bi
naryMessenger: flutterViewController as! FlutterBinaryMessenger)
    methodChannel.invokeMethod("showPageOne", arguments: "这里是参数，可以
是任意类型")
    self.present(flutterViewController, animated: true) {}
    methodChannel.setMethodCallHandler { call, result in
      // 这里面监听Flutter中传递回来的消息
      if call.method == "blablabla" { // 通过消息名称来判断是哪个消息
        print(call.arguments) // 获取到消息的参数
      }
    }
```

第4步，在Flutter工程中，通过channel名称来创建指定的channel。

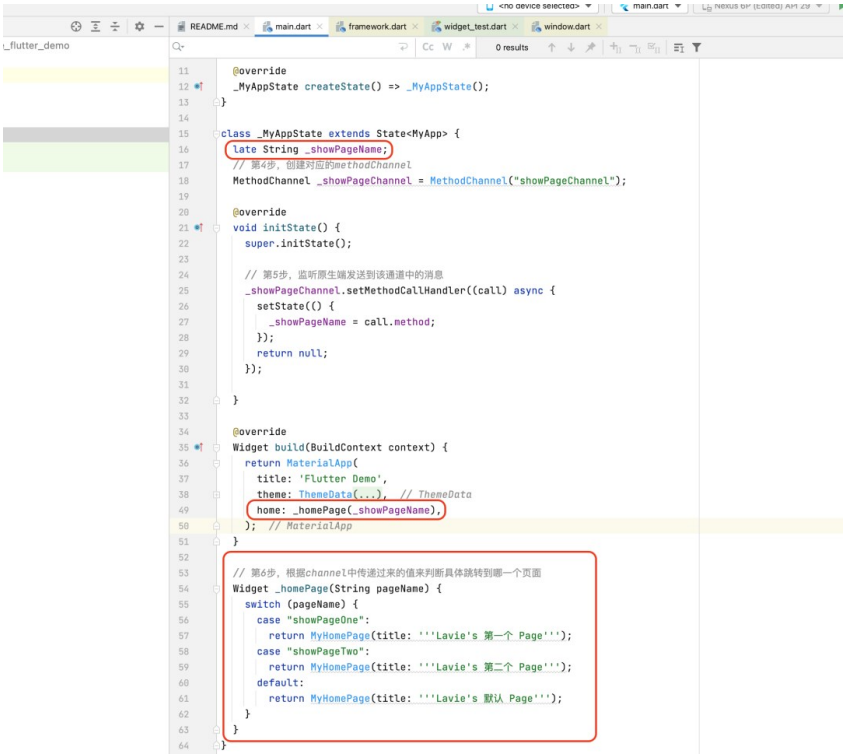
```dart

class _MyAppState extends State<MyApp> {
  late String _showPageName;
  // 第4步，创建对应的methodChannel
  MethodChannel _showPageChannel = MethodChannel("showPageChannel");

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

第5步，在Flutter工程中监听原生端发送到指定通道中的消息。



```dart
import 'dart:ui';

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

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

class MyApp extends StatefulWidget {
  MyApp({Key? key}) : super(key: key);

  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late String _showPageName;
  // 第4步，创建对应的methodChannel
  MethodChannel _showPageChannel = MethodChannel("showPageChannel");

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

    // 第5步，监听原生端发送到该通道中的消息
    _showPageChannel.setMethodCallHandler((call) async {
      setState(() {
        _showPageName = call.method;
      });
      return null;
    });

  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(...),  // ThemeData
      home: _homePage(_showPageName),
    );  // MaterialApp
  }
```

第6步，根据channel中传递过来的值判断具体是跳转到哪个页面。



```dart
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late String _showPageName;
  // 第4步，创建对应的methodChannel
  MethodChannel _showPageChannel = MethodChannel("showPageChannel");

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

    // 第5步，监听原生端发送到该通道中的消息
    _showPageChannel.setMethodCallHandler((call) async {
      setState(() {
        _showPageName = call.method;
      });
      return null;
    });

  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(...),  // ThemeData
      home: _homePage(_showPageName),
    );  // MaterialApp
  }

  // 第6步，根据channel中传递过来的值来判断具体跳转到哪一个页面
  Widget _homePage(String pageName) {
    switch (pageName) {
      case "showPageOne":
        return MyHomePage(title: '''Lavie's 第一个 Page''');
      case "showPageTwo":
        return MyHomePage(title: '''Lavie's 第二个 Page''');
      default:
        return MyHomePage(title: '''Lavie's 默认 Page''');
    }
  }
}
```

第7步，如果Flutter页面也想给原生端发消息，那么可以通过channel的`invokeMethod`方法实现。

```
// 第7步，给原生端发送消息，且可以传递参数
_showPageChannel.invokeMethod("blablabla", _counter);
```

这样的话，原生端就可以接收到Flutter这边传递过来的消息了。

这样一改造，我们再运行，就会发现，应用程序只有在第一次加载展示FlutterViewController的时候内存会暴涨80M左右，后面再进入的时候内存的变化就不会很大了。

我们在真正的开发时，一般不会频繁的在原生页面和Flutter页面之间切换，在原生工程跳转到某个Flutter页面之后，余下的页面最好能形成一个闭环。

Flutter与原生端通信的三种方式

Flutter与原生端的通信，有三种不同类型的channel可以实现，如下：

- FlutterMethodChannel
- FlutterEventChannel
- FlutterBasicMessageChannel

这三种channel分别是什么意思呢？下面一一做介绍。

一、FlutterMethodChannel

这种channel主要是用于**调用方法**的，通过invoke的形式来一次性地调用方法，这种方式是**一次通讯**。这种channel的具体用法上面已经做了详尽的阐述，这里不赘述。

二、FlutterBasicMessageChannel

这种channel用于**传递字符串和半结构化信息**，所谓的半结构化信息指的就是类似于结构体、data等这样的信息。它是**持续通讯**的，收到消息之后可以回复此次消息。比如，原生端将遍历到的文件信息陆续传递给Flutter；再比如，Flutter将从服务端陆续获取到的信息交给原生端加工，原生端处理完毕之后返回给Flutter。

在FlutterModule页面中使用

第1步，通过channel名称来创建一个对应的MessageChannel。

```
@override
_MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late String _showPageName;
  MethodChannel _showPageChannel = MethodChannel("showPageChannel");

  // 第1步，根据channel名称创建一个BasicMessageChannel
  BasicMessageChannel _messageChannel = BasicMessageChannel("lavieMessageChannel", StandardMessageCodec());

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

    _showPageChannel.setMethodCallHandler((call) async {
      setState(() {
        _showPageName = call.method;
      });
      return null;
    });
```

第2步，持续接收原生端发送过来的消息。

```
// 第1步，根据channel名称创建一个BasicMessageChannel
BasicMessageChannel _messageChannel = BasicMessageChannel("lavieMessageChannel", StandardMessageCodec());

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

  _showPageChannel.setMethodCallHandler((call) async {
    setState(() {
      _showPageName = call.method;
    });
    return null;
  });

  // 第2步，持续接收原生端发送过来的信息
  _messageChannel.setMessageHandler((message) async {
    print("收到了来自iOS的messageChannel消息—${message}");
    return null;
  });
}
```

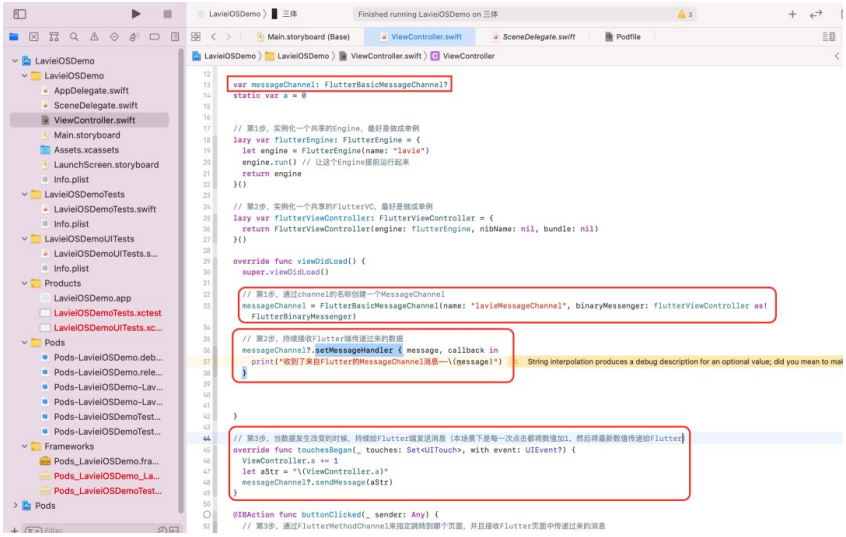
第3步，当数据发生改变的时候，持续给原生端发送消息（本场景下是写入什么文字就立即发送什么内容）

```
// horizontal).
mainAxisAlignment: MainAxisAlignment.center,
children: <Widget>[
  const Text(
```

```
            'Lavie, You have pushed the button this many times:',
          ),  // Text
          Text(
            '$_counter',
            style: Theme.of(...).textTheme.headline4,
          ),  // Text
          TextField(
            onChanged: (value) {
              // 第3步，数据发生改变的时候，持续给原生端发送消息（本场景下是写入什么文字就立即发送什么内容）
              _messageChannel.send(value);
            },
          )  // TextField
        ],  // <Widget>[]
      ),  // Column
    ),  // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.  // FloatingActionButton
  );  // Scaffold
}
```

在原生项目中使用

第1步，通过channel名称来创建一个对应的MessageChannel
第2步，持续接收Flutter端传递过来的数据
第3步，当数据发生改变的时候，持续给Flutter端发送消息（本场景下是每一次点击都将数值+1，然后将最新的数值传递给Flutter端）



三、FlutterEventChannel

这种channel是用于**数据流（stream）的通讯**，它是一种**持续通信**，但是收到消息之后无法回复此次消息。这种channel通常用于原生端向Flutter的通信，比如：手机电量的变化、网络连接的变化、传感器等。

以上这三种类型的channel全部都是双向通信，即Flutter可以向原生端通信，原生端也可以向Flutter通信。

以上。

收录于话题 #Flutter 58

‹ 上一篇
Flutter中的Key详解

下一篇 ›
Flutter中使用event_bus进行事件广播和事件监听