Flutter 中的组件绘制完成监听、组件生命周期和APP生命周期

A_si 关注

0.87 2020.09.07 23:04:07 字数 1,542 阅读 3,453

Flutter 的 生命周期

说到 Flutter 的生命周期，其实就是说 StatefulWidget 的生命周期，因为 StatelessWidget 是静态控件。

StatefulWidget，通过借助于 State 对象，处理状态变化，并体现在 UI 上。这些阶段，就涵盖了一个组件从加载到卸载的全过程，即生命周期。

而一个应用的生命周期，包括了页面组件的生命周期和整个 app 的生命周期。我们分别了解下。

State 生命周期

首先我们看下 State 里面的这几个方法：

```
@override
 void initState() {
   super.initState();
   print('MyHomePage==initState');
 }

 @override
 void didChangeDependencies() {
   super.didChangeDependencies();
   print('MyHomePage==didChangeDependencies');
 }

 @override
 void didUpdateWidget(MyHomePage oldWidget) {
   super.didUpdateWidget(oldWidget);
   print('MyHomePage==didUpdateWidget');
 }

 @override
 void reassemble() {
   super.reassemble();
   print('MyHomePage==reassemble');
 }

 @override
 void deactivate() {
   super.deactivate();
   print('MyHomePage==deactivate');
 }

 @override
 void dispose() {
   print('MyHomePage==dispose');
   super.dispose();
 }
   @override
 Widget build(BuildContext context) {
   print('MyHomePage==build');
   。。。
 }

```

接下来我们运行应用，看下打印日志：

```
flutter: MyHomePage==initState
flutter: MyHomePage==didChangeDependencies
flutter: MyHomePage==build
```

然后路由 `Navigator.push` 到新页面：

```
flutter: SecondPage==initState
flutter: SecondPage==didChangeDependencies
flutter: SecondPage==build
```

然后路由 `Navigator.pop` 到回到首页：

```
flutter: SecondPage==deactivate
flutter: SecondPage==dispose
```

State 初始化时会依次执行： 构造方法 -> initState -> didChangeDependencies -> build，随后完成页面渲染。
State 销毁时会依次执行： deactivate -> dispose 随后完成页面释放。

接下来我们看下这几个方法。

构造方法

构造方法是 State 生命周期的起点，Flutter 会通过调用 StatefulWidget.createState() 来创建一个 State。我们可以通过构造方法，来接收父 Widget 传递的初始化 UI 配置数据。这些配置数据，决定了 Widget 最初的呈现效果。

initState

当Widget第一次插入到Widget树时会被调用，对于每一个State对象，Flutter framework只会调用一次该回调，所以，通常在该回调中做一些一次性的操作，如状态初始化、订阅子树的事件通知等。
但是使用 InheritFromWidget 的时候，不能在该回调中调用BuildContext.dependOnInheritedWidgetOfExactType（该方法用于在Widget树上获取离当前widget最近的一个父级InheritFromWidget，原因是在初始化完成后，Widget树中的InheritFromWidget也可能会发生变化，所以正确的做法应该在在build（）方法或didChangeDependencies()中调用它。

didChangeDependencies

didChangeDependencies 则用来专门处理 State 对象依赖关系变化，会在 initState() 调用结束后，被 Flutter 调用。
哪些情况下 State 对象的依赖关系会发生变化呢？比如使用 InheritedWidget 作数据共享的时候，InheritedWidget 的发生了变化，子widget的didChangeDependencies()回调都会被调用。典型的场景是，系统语言 Locale 或应用主题改变时，系统会通知 State 执行 didChangeDependencies 回调方法。

build

它主要是用于构建Widget子树的，会在如下场景被调用：

- 在调用initState()之后。
- 在调用didUpdateWidget()之后。
- 在调用setState()之后。
- 在调用didChangeDependencies()之后。
- 在State对象从树中一个位置移除后（会调用deactivate）又重新插入到树的其它位置之后。

deactivate

deactivate()：当State对象从树中被移除时，会调用此回调。在一些场景下，Flutter framework会将State对象重新插到树中，如包含此State对象的子树在树的一个位置移动到另一个位置时（可以通过GlobalKey来实现）。如果移除后没有重新插入到树中则紧接着会调用dispose()方法。

dispose

dispose()：当State对象从树中被永久移除时调用；通常在此回调中释放资源。

didUpdateWidget

下面我们写个嵌套布局，在 MyHomePage 里嵌套 ChildView，然后在 MyHomePage 里调用 setState：

```
body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           FlatButton(
             onPressed: () {
               setState(() {
                 _count++;
               });
             },
             child: Text('点击父控件'),
           ),
           ChildView(),
         ],
       ),
     ),
```

看下打印信息：

```
flutter: MyHomePage==build
flutter: ChildView==didUpdateWidget
flutter: ChildView==build
```

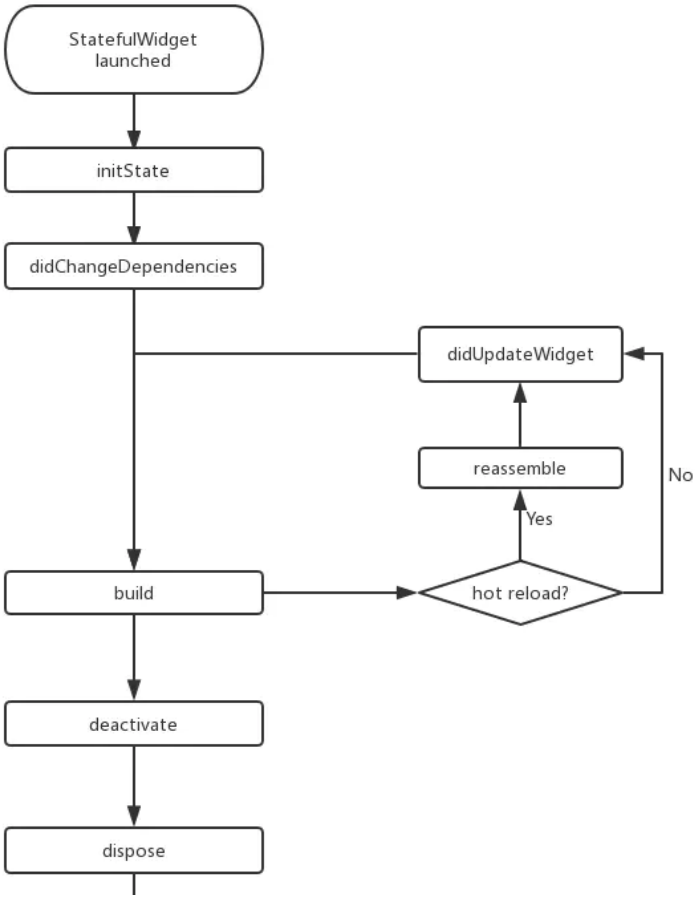
didUpdateWidget：当 Widget 的配置发生变化时，比如，父 Widget 触发重建（即父 Widget 的状态发生变化时），热重载时，系统会调用这个函数。

reassemble

上面我们说热重载的时候，didUpdateWidget会被调用，其实 reassemble 也会被调用，reassemble 是一个 debug 方法，在热重载的时候调用，我们点 IDE 的热重载按钮：

```
Performing hot reload...
Syncing files to device iPhone 11 Pro...
flutter: MyHomePage==reassemble
flutter: ChildView==reassemble
flutter: MyHomePage==didUpdateWidget
flutter: MyHomePage==build
flutter: ChildView==didUpdateWidget
flutter: ChildView==build
Reloaded 1 of 513 libraries in 286ms.
```

会先深度调用 reassemble ，然后再 调用 didUpdateWidget 和 build。



生命周期.jpg

APP 的生命周期

WidgetsBindingObserver

在原生开发中，我们都会获取应用是否在前台的状态，在 Flutter 中同样需要。我们可以利用 WidgetsBindingObserver 实现。

```
abstract class WidgetsBindingObserver {
  //页面pop
  Future didPopRoute() => Future.value(false);

  //页面push
  Future didPushRoute(String route) => Future.value(false);

//系统窗口相关改变回调，如旋转
  void didChangeMetrics() {}

// 文本缩放系数变化
  void didChangeTextScaleFactor() {}

// 系统亮度变化
  void didChangePlatformBrightness() {}

// 本地化语言变化
  void didChangeLocales(List locale) {}

// App生命周期变化
  void didChangeAppLifecycleState(AppLifecycleState state) {}

// 内存警告回调
  void didHaveMemoryPressure() {}

// Accessibility相关特性回调
  void didChangeAccessibilityFeatures() {}
}
```

可以看到，WidgetsBindingObserver 这个类提供的回调函数非常丰富，常见的屏幕旋转、屏幕亮度、语言变化、内存警告都可以通过这个实现进行回调。我们通过给 WidgetsBinding 的单例对象设置监听器，就可以监听对应的回调方法。

下面我们写个demo：

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

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
    super.dispose();
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state) async {
    print("$state");
    if (state == AppLifecycleState.resumed) {
    }
  }
  }
```

从前台切到后台：

```
flutter: AppLifecycleState.inactive
flutter: AppLifecycleState.paused
```

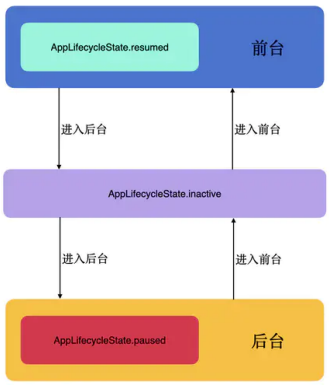
再从后台切刀前台：

```
flutter: AppLifecycleState.inactive
flutter: AppLifecycleState.resumed
```

didChangeAppLifecycleState

didChangeAppLifecycleState 回调函数中，有一个参数类型为 AppLifecycleState 的枚举类，这个枚举类是 Flutter 对 App 生命周期状态的封装。它的常用状态包括 resumed、inactive、paused 这三个。

- resumed：可见的，并能响应用户的输入。
- inactive：处在不活动状态，无法处理用户响应。
- paused：不可见并不能响应用户的输入，但是在后台继续活动中。



前后台.001.jpeg

封装一个工具类：

```
class LifecycleEventHandler extends WidgetsBindingObserver{
  static const String TAG = '==lifecycle_event_handler==';
  final AsyncCallback resumeCallBack;
  final AsyncCallback suspendingCallBack;

  LifecycleEventHandler({
    this.resumeCallBack,
    this.suspendingCallBack,
  });

  @override
  Future<Null> didChangeAppLifecycleState(AppLifecycleState state) async {
    switch (state) {
      case AppLifecycleState.resumed:
        if (resumeCallBack != null) {
          await resumeCallBack();
        }
        break;
      case AppLifecycleState.inactive:
      case AppLifecycleState.paused:
      case AppLifecycleState.detached:
        if (suspendingCallBack != null) {
          await suspendingCallBack();
        }
        break;
    }
  }
}
```

State就不需要再混入WidgetsBindingObserver ，使用起来就简单多了：

```
class _MyHomePageState extends State<MyHomePage>  {

  LifecycleEventHandler _lifecycleEventHandler = LifecycleEventHandler(
      resumeCallBack: () async {}, suspendingCallBack: () async {});

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(_lifecycleEventHandler);
  }
  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(_lifecycleEventHandler);
    super.dispose();
  }
  }
```

另一种用法：
我们可以在入口函数里直接调用，这样就不用侵入widget了：

```

void main() {
  WidgetsFlutterBinding.ensureInitialized();
  WidgetsBinding.instance.addObserver(LifecycleEventHandler(
      resumeCallBack: () async {}, suspendingCallBack: () async {}));

  runApp(MyApp());
}
```

SystemChannels.lifecycle

除了上面的方法，Flutter 还为我们提供了一种方法， SystemChannels.lifecycle，同样可以监听到 APP 的生命周期：

```
class _MyHomePageState extends State<MyHomePage> {
  @override
```

```
  void initState() {
    super.initState();
    SystemChannels.lifecycle.setMessageHandler((msg) {
      switch (msg) {
        case "AppLifecycleState.paused":
          print(msg);
          break;
        case "AppLifecycleState.inactive":
          print(msg);
          break;
        case "AppLifecycleState.resumed":
          print(msg);
          break;
        default:
          break;
      }
    });
  }

  @override
  void dispose() {
    super.dispose();
  }
}
```

前后台切换：

```
Syncing files to device iPhone 11 Pro...
flutter: MyHomePage==build
flutter: AppLifecycleState.inactive
flutter: AppLifecycleState.paused
flutter: AppLifecycleState.inactive
flutter: AppLifecycleState.resumed
```

同样我们可以封装个工具类：

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

class LifecycleEventHandler {
  final AsyncCallback resumeCallBack;
  final AsyncCallback suspendingCallBack;

  LifecycleEventHandler({
    this.resumeCallBack,
    this.suspendingCallBack,
  }) {
    SystemChannels.lifecycle.setMessageHandler((msg) async {
      switch (msg) {
        case "AppLifecycleState.resumed":
          if (resumeCallBack != null) {
            await resumeCallBack();
          }
          break;
        case "AppLifecycleState.paused":
        case "AppLifecycleState.detached":
          if (suspendingCallBack != null) {
            await suspendingCallBack();
          }
          break;
        default:
          break;
      }
    });
  }
}

```

然后使用：

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  void initState() {
    super.initState();
    handleAppLifecycleState(resumeCallBack:()async{
```

```
      print('resumeCallBack');
    },suspendingCallBack:()async{
      print('suspendingCallBack');
    });
  }
  }
```

切换前后台：

```
flutter: MyHomePage==build
flutter: suspendingCallBack
flutter: resumeCallBack
```

为了不入侵 Widget ,同样可以在入口函数处调用：

```
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  handleAppLifecycleState(resumeCallBack: () async {
    print('resumeCallBack');
  }, suspendingCallBack: () async {
    print('suspendingCallBack');
  });
  runApp(MyApp());
}

```

View 绘制完成

在原生中，还有一个常用操作是监听 View 绘制完成，防止空指针。

```

//view重绘时回调
view.getViewTreeObserver().addOnDrawListener(new OnDrawListener() {
    @Override
    public void onDraw() {
    // TODO Auto-generated method stub

    }

```

Flutter 同样给我们有两个回调函数：

1. addPostFrameCallback 只有首次绘制完才回调
2. addPersistentFrameCallback 每次重绘都回调

```
  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
    WidgetsBinding.instance.addPostFrameCallback((_) {
      print("单次Frame绘制回调"); //只回调一次
    });
    WidgetsBinding.instance.addPersistentFrameCallback((_) {
      print("实时Frame绘制回调"); //每帧都回调
    });
  }

```

```
flutter: MyHomePage==build
flutter: 实时Frame绘制回调
flutter: 单次Frame绘制回调
flutter: 实时Frame绘制回调
```

点击 setState：

```
flutter: MyHomePage==build
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
```

```
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
flutter: 实时Frame绘制回调
```

因为是递归回调的，所以会调用多次。

有了这两个函数，我们可以实现原生一样的功能。