Flutter的InheritedWidget

wyler

1.概述

场景1:一个界面是由众多组件拼组而成。经常需要将一个组件进行封装，但此时有一个问题，如何让多个组件去共享一些值。一个最直接的方法就是通过构造函数将变量和函数一层层向下传递。如果嵌套的很深，那简直就是噩梦，如果子组件需要更新父组件的数据或者状态，再加个回调?

额。。。这个时候我们可以使用Flutter提供的InheritedWidget。

- 特性:

  InheritedWidget是Flutter中的一个非常重要的功能组件，它能够提供数据在widget树中从上到下进行传递。保证数据在不同子widget中进行共享。

  show code ~~

```
import 'package:flutter/material.dart';
///数据共享
class SecondDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return TestPage(
      child: Scaffold(
        appBar: AppBar(
          title: Text('InheritedWidget Demo'),
```

```
        ),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            WidgetA(),
            WidgetB(),
            WidgetC(),
          ],
        ),
      ),
    );
  }
}
// ignore: must_be_immutable
class InheritedProvide extends InheritedWidget {///数据共享

  InheritedProvide({
    Key key,  key,
    @required Widget child,
    @required this.data,
  }) : super(key: key, child: child);

  final TestPageState data;

  @override
  bool updateShouldNotify(InheritedProvide oldWidget) {
    return true;
  }
```

```
}
class TestPage extends StatefulWidget {
  TestPage({
    Key key,    key,
    this.child,
  }) : super(key: key);

  final Widget child;

  @override
  TestPageState createState() => TestPageState();

  static TestPageState of(BuildContext context) {
    // ignore: deprecated_member_use
    return
(context.inheritFromWidgetOfExactType(InheritedProvide) as
InheritedProvide)?.data;
  }    return (context.getInheritedWidgetOfExactType<InheritedProvide>() as
       InheritedProvide).data;
}

class TestPageState extends State<TestPage> {
  int counter = 0;

  void incrementPageCounter() {
    setState(() {
      counter++;
    });
  }
```

```
  @override
  Widget build(BuildContext context) {
    return InheritedProvide(
      data: this,
      child: widget.child,
    );
  }
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    //父或祖先widget中的InheritedWidget改变(updateShouldNotify
返回true)时会被调用。
    //如果build中没有依赖InheritedWidget，则此回调不会被调用。
    print("Dependencies change");
  }
}
class WidgetA extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context);
    print('----------WidgetA build------');
    return Center(
      child: Text('点击的次数',),
    );
  }
}
```

```
class WidgetB extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context,);

    print('----------WidgetB build------');

    return  new Padding(
      padding: const EdgeInsets.only(left: 10.0, top: 10.0,
right: 10.0),
      child:  new Text('${state.counter}', style:
Theme.of(context).textTheme.display1,),
    );
  }
}
class WidgetC extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context);
    print('----------WidgetC build------');
    return RaisedButton(
      onPressed: () {
        state.incrementPageCounter();
      },
      child: Icon(Icons.add),
    );
  }
```

```
}
```

问题:在上面的 demo 中，存在一个性能问题

```
class WidgetC extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context);
    print('----------WidgetC build------');
    return RaisedButton(
      onPressed: () {
        state.incrementPageCounter();
      },
      child: Icon(Icons.add),
    );
  }
}
```

这个 WidgetC 不应该 build,因为调用了 final TestPageState state = TestPage.of(context);这句代码，也就是说依赖了 Widget 树上面的 InheritedWidget（即 InheritedProvider ）Widget，所以当点击完按钮后，数据发生变化，会通知 TestPage, 而 TestPage 则会重新构建子树，所以 InheritedProvider 将会更新，此时依赖它的子孙 Widget 就会被重新构建。那么如何避免 WidgetC 不 Build 呢?

- 解决方案：

  既然 WidgetC 重新被 build 是因为 WidgetC 和 InheritedWidget 建立了依赖关系，那么我们只要打破或

解除这种依赖关系就可以了。如何打破呢？需要用到inheritFromWidgetOfExactType和ancestorWidgetOfExactType这两个方法。

代码如下：

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

//局部刷新
class ThirdDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return TestPage(
      child: Scaffold(
        appBar: AppBar(
          title: Text('InheritedWidget Demo'),
        ),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            WidgetA(),
            WidgetB(),
            WidgetC(),
          ],
        ),
      ),
    );
  }
```

```
}
// ignore: must_be_immutable
class InheritedProvider extends InheritedWidget {///数据共享

  InheritedProvider({
    Key key,
    @required Widget child,
    @required this.data,
  }) : super(key: key, child: child);

  final TestPageState data;

  @override
  bool updateShouldNotify(InheritedProvider oldWidget) {
    return true;
  }
}
class TestPage extends StatefulWidget {
  TestPage({
    Key key,
    this.child,
  }) : super(key: key);

  final Widget child;

  @override
  TestPageState createState() => TestPageState();
```

```
  static TestPageState of(BuildContext context, {bool
rebuild = true}) {
    if (rebuild) {
      // ignore: deprecated_member_use
      return
(context.inheritFromWidgetOfExactType(InheritedProvider) as
InheritedProvider)?.data;
    }
    // ignore: deprecated_member_use
    return
(context.ancestorWidgetOfExactType(InheritedProvider) as
InheritedProvider)?.data;
  }
}

class TestPageState extends State<TestPage> {
  int counter = 0;

  void incrementPageCounter() {
    setState(() {
      counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return InheritedProvider(
      data: this,
```

return (context.dependOnInheritedWidgetOfExactType<
InheritedProvide>()
as InheritedProvide)
.data;

return (context.findAncestorWidgetOfExactType<InheritedProvide>()
as InheritedProvide)
.data;

```
      child: widget.child,
    );
  }

}
class WidgetA extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context);
    print('----------WidgetA build------');
    return Center(
      child: Text('点击的次数',),
    );
  }
}
class WidgetB extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context,);

    print('----------WidgetB build------');

    return  new Padding(
      padding: const EdgeInsets.only(left: 10.0, top: 10.0,
right: 10.0),
      child:  new Text('${state.counter}', style:
Theme.of(context).textTheme.display1,),
```

```
    );
  }
}
class WidgetC extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final TestPageState state = TestPage.of(context,
rebuild: false);
    print('----------WidgetC build------');
    return RaisedButton(
      onPressed: () {
        state.incrementPageCounter();
      },
      child: Icon(Icons.add),
    );
  }
}
```

1.创建类InheritedProvide，继承于InheritedWidget，新增成员变量：父控件的satae；
2.在类InheritedProvide中，重写updateShouldNotify方法，并返回true，通知子widget更新其相关的依赖；
3.在父组件中新增Widget类型的成员变量child
4.在父组件中新定义of方法，该方法通过类InheritedProvide返回其新增的成员变量state，让子组件通过这个of方法访问state数据；
5.父组件的state类的build方法中返回新建的类InheritedProvide通过构造函数创建的对象，传入父组件新增成员变量child；
6.子组件通过父组件的of方法访问父组件state里面的数据。

再点击会发现“重新build和WidgetC build”这句话不打印了。说明 WidgetC 不 build 了。 还是会全部刷新B和C

- 总结：调用 inheritFromWidgetOfExactType() 和 ancestorWidgetOfExactType() 的区别就是前者会注册依赖关系，而后者不会，所以在调用 inheritFromWidgetOfExactType() 时，InheritedWidget 和依赖它的子孙组件关系便完成了注册，之后当 InheritedWidget 发生变化时，就会更新依赖它的子孙组件，也就是会调这些子孙组件的 build() 方法。而当调用

的是 ancestorWidgetOfExactType() 时，由于没有注册依赖关系，所以之后当 InheritedWidget 发生变化时，就不会更新相应的子孙 Widget。

2. ValueListenableBuilder

InheritedWidget 提供一种在 widget 树中从上到下共享数据的方式，但是也有很多场景数据流向并非从上到下，比如从下到上或者横向等。为了解决这个问题，Flutter 提供了一个 ValueListenableBuilder 组件，它的功能是监听一个数据源，如果数据源发生变化，则会重新执行其 builder。

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

///直接调用SetState
class FourthDemo extends StatefulWidget{
  @override
  State<StatefulWidget> createState() {
    // TODO: implement createState
    return _FourthDemoState();
  }
}

class _FourthDemoState extends State<FourthDemo>{
  int counter = 0;
```

```
  void incrementCounter() {
//    setState(() {
      counterNotifier.value = counter++;
//    });
  }
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    print("Dependencies change");
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('FirstDemo'),),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
          WidgetA(),
          WidgetB(counter),
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
```

```
  }
}
class WidgetA extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    print('----------WidgetA build------');
    return Center(
      child: Text('You have pushed the button this many
times:',),
    );
  }
}
ValueNotifier<int> counterNotifier = ValueNotifier<int>(0);
class WidgetB extends StatelessWidget {
  final int counter;
  WidgetB(this.counter);
  @override
  Widget build(BuildContext context) {
    print('----------WidgetB build------');
    return ValueListenableBuilder<int>(
        valueListenable: counterNotifier,
        builder: (BuildContext context,int counter,Widget
child){
          print('----------ValueListenableBuilder
build------');
          return Text('$counter', style:
Theme.of(context).textTheme.display1,);
```

```
        });
    }
}
```

- 总结

1. 和数据流向无关，可以实现任意流向的数据共享。
2. 实践中，ValueListenableBuilder 的拆分粒度应该尽可能细，可以提高性能。

1. 使用ValueListenableBuilder封装需要重新build的控件，并传入创建的全局的ValueNotifier对象。
2. 创建一个全局的ValueNotifier对象；
3. 改变counterNotifier.value的值，重新build控件。