Flutter 之 InheritedWidget


云层之上 LV.3
2022年02月11日 11:21 · 阅读 383

关注

> 我们在进行Flutter开发时，经常会遇到数据传递的问题。我们可能会逐级逐级一层一层传递，由于Widget 树的关系，Widget层级可以做得非常深，在这些层级间传递数据会让效率变得很低。也可能在某一层widget不需要这些数据，但是在下一层会使用的这些数据，造成了不需要数据的widget却也持有数据，显得冗余且不优雅。

如：

kotlin 复制代码

```
return A(
  data:data
  child:B(
    data:data
    child:C(
      data:data
      child:D(
        data:data
      )
    )
  )
);
```

其中可能B和C都没用到data，只是D使用到了data，却也不得不传递data，就显得很呆！

为了解决以上问题，Flutter提供了 `InheritedWidget` 这样一个功能型组件，它提供了一种在 widget 树中从上到下共享数据的方式，即在父widget 中通过 `InheritedWidget` 共享了一个数据，那么在任意子widget都能获取该共享的数据！

例如我平时使用到的 `MediaQuery.of(context)` 、 `Theme.of(context)` ，都是使用InheritedWidget来实现数据共享的。

> 实际上Flutter中使用 `setState()` 和 `InheritedWidget` 足以解决绝大部分的状态管理问题，现有的很多状态管理框架也是基于 `InheritedWidget` 封装的。

首先通过继承 `InheritedWidget` ，将当前计数器点击数据保存在 `CounterInheritedWidget` 中：

- CounterInheritedWidget

```scala
class CounterInheritedWidget extends InheritedWidget {
  /// 构造方法
  const CounterInheritedWidget({Key? key, required this.count,
    required Widget child}):super(key:key, child: child);

  /// 需要共享的数据
  final int count;

  /// 默认的约定：如果状态是希望暴露出的，应当提供一个`of`静态方法来获取其对象，开发者便可直接通过该方
  /// 返回实例对象，方便子树中的widget获取共享数据
  static CounterInheritedWidget? of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<CounterInheritedWidget>();
  }

  /// 是否通知widget树中依赖该共享数据的子widget
  /// 这里当count发生变化时，是否通知子树中所有依赖count的Widget重新build
  /// 这里判断注意：是值改变还是内存地址改变。
  @override
  bool updateShouldNotify(covariant CounterInheritedWidget oldWidget) {
    return count != oldWidget.count;
  }
}
```

我们自定义2个Widget（分别继承 `StatelessWidget` 和 `StatefulWidget` ）来模拟多个子Widget共享数据：

- CounterOneWidget：

```scala
```

```
class CounterOneWidget extends StatelessWidget {
  const CounterOneWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Container(
      width: 100, height: 100,
      color: Colors.redAccent,
      alignment: Alignment.center,
      child: Text(CounterInheritedWidget.of(context)!.count.toString(),style: const Tex
    );
  }
}
```

- CounterTwoWidget：

```scala 复制代码
class CounterTwoWidget extends StatefulWidget {
  const CounterTwoWidget({Key? key}) : super(key: key);
  @override
  State<CounterTwoWidget> createState() => _CounterTwoWidgetState();
}

class _CounterTwoWidgetState extends State<CounterTwoWidget> {
  @override
  Widget build(BuildContext context) {
    return Container(
      width: 100, height: 100,
      color: Colors.greenAccent,
      alignment: Alignment.center,
      child: Text(CounterInheritedWidget.of(context)!.count.toString(),style: const Tex
    );
  }
}
```

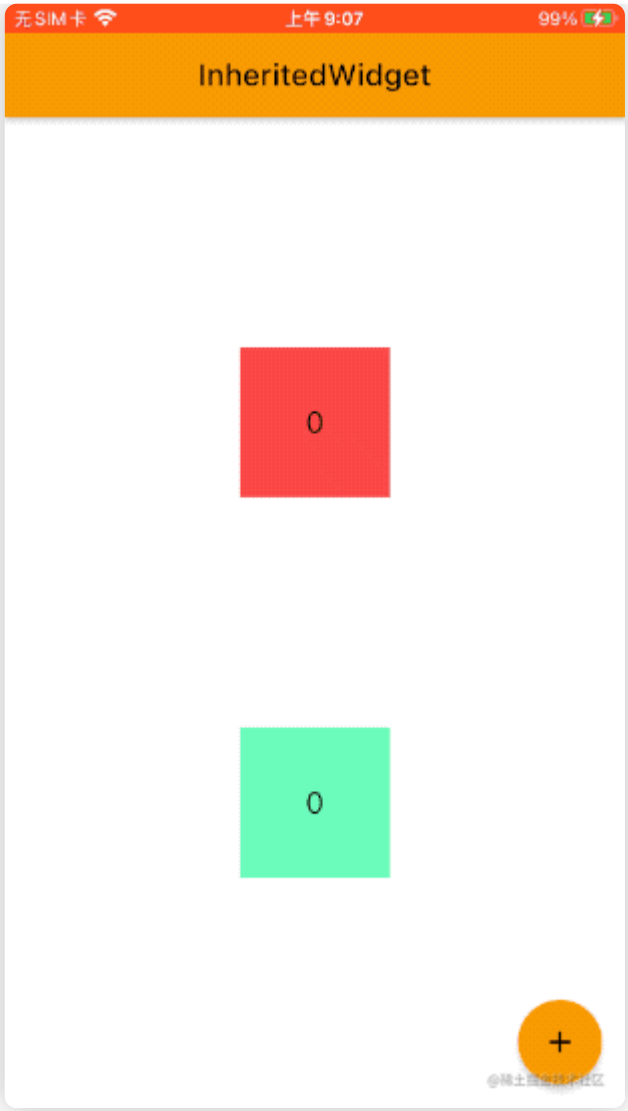
- page页面

```scala 复制代码
class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key}) : super(key: key);
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _count = 0;
  @override
  Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(title: const Text("InheritedWidget"),),
      body: CounterInheritedWidget( /// 父节点使用
        count: _count,
        child: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: const [ /// 所有子节点均可以共享数据
              CounterOneWidget(),
              CounterTwoWidget(),
            ],
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: (){
          if(!mounted) return;
          setState(() {
            _count += 1;
          });
        },
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

- 效果:



日常开发中，我们经常会遇到跨页面共享数据。且在某一个页面更改数据，所有依赖页面的数均会改变，例如Theme - 改变主题功能。

场景：

我们来使用 `InheritedWidget` 来实现这样的场景：两个页面共享数据，且在一个页面更改数据，两个页面均会发生改变。

分析：

- 跨页面： `InheritedWidget` 需要在 `MaterialApp` 的上层节点
- 更新状态： 需要使用 `StatefulWidget` 来 `setState()`

实现：

- Person类：

```dart
/// 我们使用更加贴合日常开发的模型类作为需要共享的数据来类比。
class Person{
  String name;
  int age;
  Person({required this.name, required this.age});
}
```

- PersonInheritedWidget：

```scala
class PersonInheritedWidget extends InheritedWidget {
  /// 构造方法
  const PersonInheritedWidget({Key? key, required this.person,
    required this.updateCallback,required Widget child}): super(key:key, child: child);

  /// 需要共享的数据
  final Person person;
  /// 我们使用回调方法来更新数据
  final Function(Person person) updateCallback;

  /// 定义一个便捷方法，获取对象，方便子树中的widget获取共享数据
  static PersonInheritedWidget? of(BuildContext context) {
      return context.dependOnInheritedWidgetOfExactType<PersonInheritedWidget>();
  }
  /// 提供一个刷新方法
  void updateInfo(Person person){
    updateCallback(person);
  }

  /// 是否通知依赖该树共享数据的子widget
  @override
  bool updateShouldNotify(covariant PersonInheritedWidget oldWidget) {
    return person != oldWidget.person;


  }
```

```
}
```

- PersonStateWidget

scala 复制代码

```scala
class PersonStateWidget extends StatefulWidget {
  const PersonStateWidget({Key? key,required this.person,required this.child}) : super(
  final Person person;
  final Widget child;
  @override
  _PersonStateWidgetState createState() => _PersonStateWidgetState();
}

class _PersonStateWidgetState extends State<PersonStateWidget> {
  late Person _person;

  @override
  void initState() {
    // TODO: implement initState
    _person = widget.person;
    super.initState();
  }
  @override
  Widget build(BuildContext context) {
    return PersonInheritedWidget(person: _person, updateCallback: _updateCallback, child
  }
  /// 使用setState更新数据
  void _updateCallback(Person person){
    if(!mounted) return;
    setState(() {
      _person = person;
    });
  }
}
```

- MyApp

scala 复制代码

```scala
/// 由于跨页面，`InheritedWidget`需要在`MaterialApp`的上层节点
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return PersonStateWidget(
      person: Person(name: "张三",age: 18),
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
```

```
        theme: ThemeData(
          primarySwatch: Colors.orange,
        ),
        home: const FirstPage(),
      ),
    );
  }
}
```

- FirstPage

scala 复制代码

```
class FirstPage extends StatefulWidget {
  const FirstPage({Key? key}) : super(key: key);
  @override
  _FirstPageState createState() => _FirstPageState();
}

class _FirstPageState extends State<FirstPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("FirstPage"),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Text("姓名：${PersonInheritedWidget.of(context)!.person.name}",style: const 
            Text("年龄：${PersonInheritedWidget.of(context)!.person.age}",style: const T
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: (){ /// 跳转下一页
          Navigator.push(context, MaterialPageRoute(builder: (context) {
            return const PersonTwoPage();
          }));
        },
        child: const Icon(Icons.chevron_right_outlined),
      ),
    );
  }
}
```

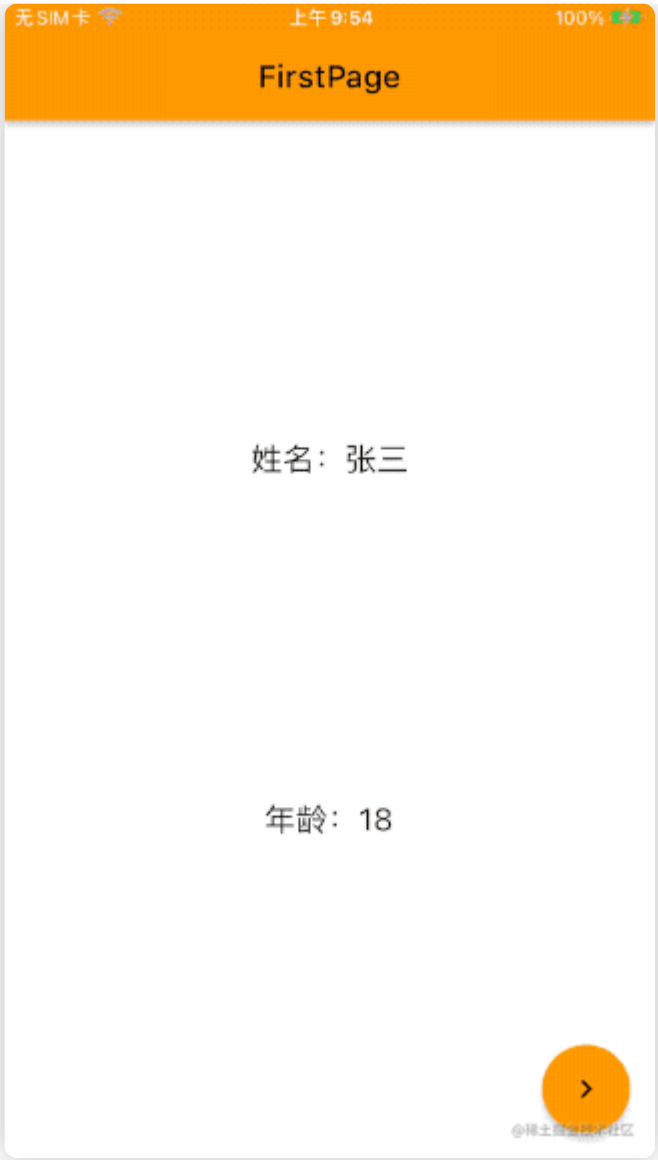
- SecondPage

scala 复制代码

```
class SecondPage extends StatefulWidget {
  const SecondPage({Key? key}) : super(key: key);
  @override
  _SecondPageState createState() => _SecondPageState();
}

class _SecondPageState extends State<SecondPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("SecondPage"),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Text("姓名：${PersonInheritedWidget.of(context)!.person.name}",style: const 
            Text("年龄：${PersonInheritedWidget.of(context)!.person.age}",style: const T
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: (){ /// 点击更改数据
          PersonInheritedWidget.of(context)?.updateCallback(Person(name: "李四",age: 28)
        },
        child: const Icon(Icons.change_circle_outlined),
      ),
    );
  }
}
```

- 效果：



didChangeDependencies

额外提一下这个方法：didChangeDependencies

我们知道在 `StatefulWidget` 的 `State` 对象有一个 `didChangeDependencies` 回调，它会在“依赖”发生变化时被Flutter 框架调用。而这个“依赖”指的就是子 widget 是否使用了父 widget 中 `InheritedWidget` 的数据！如果使用了，则代表子 widget 有依赖。这种机制可以使子组件在所依赖的 `InheritedWidget` 变化时来更新自身！

一般来说子 widget 很少会重写此方法，因为在依赖发送改变后，会调用 `build()` 方法构建组件树。如果我们需要在重构后做一些耗时、昂贵的操作，如网络请求，就可以在此方法中执行，这也是 didChangeDependencies 生命周期存在的意义。

我们能读取共享数据的前提是通过一个自定义的静态方法 `of` 来获取 `InheritedWidget` 对象。`of` 静态方法调用了 `dependOnInheritedWidgetOfExactType()` 方法。其实我们还可以通过另外一个方法 `context.getElementForInheritedWidgetOfExactType().widget` 来获取对象：

```javascript
/// 定义一个便捷方法，获取对象，方便子树中的widget获取共享数据
static PersonInheritedWidget? of(BuildContext context) {
  return context.getElementForInheritedWidgetOfExactType<PersonInheritedWidget>()!.widg
}
```

对比两者实现源码：

- dependOnInheritedWidgetOfExactType()源码：

```dart
@override
T? dependOnInheritedWidgetOfExactType<T extends InheritedWidget>({Object? aspect}) {
  /// 断言，用于监测在调试模式下是否有正在使用的父Widget
  assert(_debugCheckStateIsActiveForAncestorLookup());
  /// 获取到_inheritedWidgets数组数据
 final InheritedElement? ancestor = _inheritedWidgets == null ? null : _inheritedWidget
  if (ancestor != null) {
    /// 返回并调用更新方法
    return dependOnInheritedElement(ancestor, aspect: aspect) as T;
  }
  _hadUnsatisfiedDependencies = true;
  return null;
}
```

- getElementForInheritedWidgetOfExactType()源码：

```java
@override
InheritedElement? getElementForInheritedWidgetOfExactType<T extends InheritedWidget>()
  assert(_debugCheckStateIsActiveForAncestorLookup());
  final InheritedElement? ancestor = _inheritedWidgets == null ? null : _inheritedWidge
  return ancestor;
}
```

对比之后我们最直观的发现就是 `dependOnInheritedWidgetOfExactType()` 源码中，多调用了一个 `dependOnInheritedElement()` 方法，查看其源码：

- dependOnInheritedElement()

```typescript 复制代码

@override
InheritedWidget dependOnInheritedElement(InheritedElement ancestor, { Object? aspect })
 assert(ancestor != null);
 _dependencies ??= HashSet<InheritedElement>();
 _dependencies!.add(ancestor);
 ancestor.updateDependencies(this, aspect);
 return ancestor.widget;
}
```

可以看到 `dependOnInheritedElement()` 方法中主要是注册了依赖关系！

两个方法的区别就是前者会注册依赖关系，而后者不会。所以当前者 `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 `didChangeDependencies()` 方法和 `build()` 方法。而当调用的是 `getElementForInheritedWidgetOfExactType()` 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。

其次在两者源码中，首先在 `_inheritedWidgets` 查找对应的 `InheritedElement`，

其中 `_inheritedWidgets`：

```dart 复制代码

Map<Type, InheritedElement> _inheritedWidgets;
```

查找其赋值的源码：

```ini 复制代码

@override
void _updateInheritance() {
  assert(_lifecycleState == _ElementLifecycle.active);
  final Map<Type, InheritedElement>? incomingWidgets = _parent?._inheritedWidgets;
  if (incomingWidgets != null)
    _inheritedWidgets = HashMap<Type, InheritedElement>.of(incomingWidgets);
  else
    _inheritedWidgets = HashMap<Type, InheritedElement>();
  _inheritedWidgets![widget.runtimeType] = this;
}
```

其中 `_updateInheritance()` 在「**Element**」也调用了：

```ini
void _updateInheritance() {
  assert(_lifecycleState == _ElementLifecycle.active);
  _inheritedWidgets = _parent?._inheritedWidgets;
}
```

上述源码说明，每个「Element」 中都含有 `_inheritedWidgets` 集合。

非 `InheritedElement` 的 「Element」 中 `_inheritedWidgets` 等于父组件的 `_inheritedWidgets`，而 `InheritedElement` 会将自身添加到 `_inheritedWidgets` 中，系统通过此方式将组件和 `InheritedWidgets` 的依赖关系层层向下传递。

再深入，可以看到在「mount」和「activate」 中均调用了 `_updateInheritance` 。说明在当前组件mount和activate阶段，系统调用 `_updateInheritance` 方法将InheritedWidget类型的父组件添加到 `_inheritedWidgets` 集合中。

当依赖子组件通过自定义静态方法 `of` 调用 `ancestor.updateDependencies` 时：

- ancestor.updateDependencies源码：

```typescript
@protected
void updateDependencies(Element dependent, Object aspect) {
  setDependencies(dependent, null);
}

@protected
  void setDependencies(Element dependent, Object value) {
    _dependents[dependent] = value;
  }
```

向 「_dependents」 中添加注册， `InheritedWidget` 组件更新时可以更具此列表通知子组件。

我们知道widget发生变化时会调用 `update` 方法：

- update源码：

```ini
@override
void update(ProxyWidget newWidget) {
  final ProxyWidget oldWidget = widget;
```

```
  assert(widget != null);
  assert(widget != newWidget);
  super.update(newWidget);
  assert(widget == newWidget);
  updated(oldWidget);
  _dirty = true;
  rebuild();
}
```

其中 InheritedElement 重写了 updated 方法：

scss 复制代码

```
@override
void updated(InheritedWidget oldWidget) {
  if (widget.updateShouldNotify(oldWidget))
    super.updated(oldWidget);
}
```

当 updateShouldNotify 返回 true时，执行更新操作。而其父类的 updated方法如下：

scss 复制代码

```
@protected
void updated(covariant ProxyWidget oldWidget) {
  notifyClients(oldWidget);
}
```

在看看 `notifyClients` 中实现了什么：

- notifyClients源码：

dart 复制代码

```
@override
void notifyClients(InheritedWidget oldWidget) {
  assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
  for (final 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);
  }
}
```

一眼就看到遍历依赖的子组件集合： `_dependents` ，并调用 `notifyDependent` 。

- notifyDependent源码：

typescript 复制代码

```
@protected
void notifyDependent(covariant InheritedWidget oldWidget, Element dependent) {
  dependent.didChangeDependencies();
}
```

这里看到了熟悉的方法： `didChangeDependencies()` 。再次说明了依赖的子组件发生变化，重建时候会调用 `didChangeDependencies` 。

总结

- widget在 `mount` 和 `activate` 阶段，系统调用 `_updateInheritance` 方法，将 `InheritedWidget` 类型的父组件添加到 `_inheritedWidgets` 集合中。
- 子组件调用 `dependOnInheritedWidgetOfExactType` 时，从 `_inheritedWidgets` 集合中获取指定的 `InheritedWidget` 类型的父组件，并将当前组件注册到 `InheritedWidget` 类型父组件的 `_dependents` 集合中。
- `InheritedWidget` 组件数据发生变化（ `updateShouldNotify` 方法返回true），重建时， `InheritedWidget` 组件遍历 `_dependents` 集合中所有依赖的子组件，执行子组件的 `didChangeDependencies` 的方法。

分类： 前端 标签： Flutter