flutter 状态管理 InheritedWidget 原理分析

蜗牛安卓_郝郝

1.InheritedWidget是什么?

InheritedWidget是Flutter中非常重要的一个功能型组件，它提供了一种数据在widget树中从上到下传递、共享的方式，比如我们在应用的根widget中通过InheritedWidget共享了一个数据，那么我们便可以在任意子widget中来获取该共享的数据！这个特性在一些需要在widget树中共享数据的场景中非常方便！如Flutter SDK中正是通过InheritedWidget来共享应用主题（Theme）和Locale (当前语言环境)信息的。

InheritedWidget和React中的context功能类似，和逐级传递数据相比，它们能实现组件跨级传递数据。

InheritedWidget的在widget树中数据传递方向是从上到下的，这和通知Notification的传递方向正好相反。

2.源码分析

InheritedWidget

先来看下InheritedWidget的源码：

```
abstract class InheritedWidget extends ProxyWidget
{  const InheritedWidget({ Key key, Widget
child }): super(key: key, child:
child);  @override  InheritedElement createElement()
```

=>InheritedElement(this); @protected bool updateShould Notify(covariant InheritedWidget oldWidget);}

它继承自 ProxyWidget：

abstract class ProxyWidget extends Widget

{ *const ProxyWidget({ Key key, @required this.child }) : super(key: key); final Widget child;}*

可以看出 Widget 内除了实现了 createElement 方法外没有其他操作了，它的实现关键一定就是 InheritedElement 了。

InheritedElement 来看下 InheritedElement 源码

class InheritedElement extends ProxyElement

{ *InheritedElement(InheritedWidget widget) : super(widget); @override InheritedWidget get wi dget => super.widget; // 这个 Set 记录了所有依赖的 Elementfinal Map<Element, Object> _dependents = HashM ap<Element, Object>();*

//该方法会在 Element mount 和 activate 方法中调用，

_inheritedWidgets 为基类 Element 中的成员，用于提高 Widget 查找父节点中的 InheritedWidget 的效率，它使用 HashMap 缓存了该节点的父节点中所有相关的 InheritedElement，因此查找的时间复杂度为 o(1) @override void _updateInheritance()

{final Map<Type, InheritedElement> incomingWidgets = _parent?._inheritedWidgets;if (incomingWidgets !

= null) _inheritedWidgets = HashMap<Type, InheritedElement>.from(incomingWidgets); else _inheritedWidgets = HashMap<Type, InheritedElement>(); _inheritedWidgets[widget.runtimeType] = this; }

//该方法在父类ProxyElement中调用，看名字就知道是通知依赖方该进行更新了，这里首先会调用重写的updateShouldNotify方法是否需要进行更新，然后遍历_dependents列表并调用didChangeDependencies方法，该方法内会调用mardNeedsBuild，于是在下一帧绘制流程中，对应的Widget就会进行rebuild，界面也就进行了更新 @override void notifyClients(InheritedWidget oldWidget)

{ assert(_debugCheckOwnerBuildTargetExists('notifyClients'));for (Element dependent in _dependents.keys)

{ notifyDependent(oldWidget, dependent); } }

其中_updateInheritance方法在基类Element中的实现如下：

```
void _updateInheritance() {
  _inheritedWidgets = _parent?._inheritedWidgets;
}
```

总结来说就是Element在mount的过程中，如果不是InheritedElement，就简单的将缓存指向父节点的缓存，如果是InheritedElement，就创建一个缓存的副本，然后将自身添加到该副本中，这样做会有两个值得注意的点：

InheritedElement的父节点们是无法查找到自己的，即InheritedWidget的数据只能由父节点向子节点传递，反之不能。

如果某节点的父节点有不止一个同一类型的InheritedWidget，调用inheritFromWidgetOfExactType获取到的是离自身最近的该类型的InheritedWidget。

看到这里似乎还有一个问题没有解决，依赖它的Widget是在何时被添加到_dependents这个列表中的呢?

回忆一下从InheritedWidget中取数据的过程，对于InheritedWidget有一个通用的约定就是添加static的of方法，该方法中通过inheritFromWidgetOfExactType找到parent中对应类型的的InheritedWidget的实例并返回，与此同时，也将自己注册到了依赖列表中，该方法的实现位于Element类，实现如下：

```
@overrideT dependOnInheritedWidgetOfExactType
// 这里通过上述mount过程中建立的HashMap缓存找到对应类型的InheritedElementfinal InheritedElement ancestor = _inheritedWidgets == null ? null : _inheritedWidgets[T];if (ancestor != null)
{    assert(ancestor is InheritedElement);return dependOnInheritedElement(ancestor, aspect:
aspect);  }  _hadUnsatisfiedDependencies = true;  return null;}
```

@overrideInheritedWidget dependOnInheritedElement(InheritedElement ancestor, { Object aspect })
{ assert(ancestor != null);
// 这个列表记录了当前*Element*依赖的所有*InheritedElement*，用于在当前*Element deactivate*时，将自己从*InheritedElement*的*_dependents*列表中移除，避免不必要的更新操作 _dependencies ??
= HashSet<InheritedElement>(); _dependencies.add(ancestor); ancestor.updateDependencies(this, aspect);return ancestor.widget;}
3.如何使用 InheritedWidget
1）、创建一个类继承自 Inheritedwidget
class InheritedContext extends InheritedWidget{ final InheritedTestModel inheritedTestModel; InheritedContext({   Key
key,   @required this.inheritedTestModel,   @required Widget child}): super(key: key, child:
child);static InheritedContext *of*(BuildContext context)
{   return context.dependOnInheritedWidgetOfExactType<InheritedContext>(); } @override bool updateShouldNotify(InheritedContext oldWidget)
{   return inheritedTestModel !=
oldWidget.inheritedTestModel; }}
2）、InheritedTestModel类为数据容器（这里定义了一个

List<int>数据源）

```
class InheritedTestModel{ final List _list;  InheritedTestMod
el(this._list);  List getList(){    return _list;  }}
class ArrayListData{  static List _list;static List getListData()
{    _list = new List();    _list.add(1);    _list.add(2);    _list.add(
3);    _list.add(4);return _list;  }}
```

3）、定义一个 Widget 使用 InheritedContext类的数据 InheritedTestModel

```
class ListDemo extends StatefulWidget{  @override  State
createState() {    return new ListDemoState();  }}
class ListDemoState extends State<ListDemo>{List _list;  I
nheritedTestModel _inheritedTestModel;  Timer _timer;  Du
ration oneSec = const Duration(seconds: 1);  @override  vo
id initState() {    _list =
ArrayListData.getListData();    _inheritedTestModel = new I
nheritedTestModel(_list);    _timer = Timer.periodic(oneSec
, (timer) {      _doTimer();    });  }  void _doTimer() {    for(int i
= 0; i < _list.length; i++){      _list[i] = _list[i]
+ 1;    }  setState(()
{    _inheritedTestModel = new InheritedTestModel(_list);  }
);  }Widget _buildBody()
{    return Container(child: ListDemo2(),    );  }  @override
Widget build(BuildContext context)
{    return InheritedContext(inheritedTestModel: _inheritedT
```

estModel, child: Scaffold(appBar: AppBar(title: Text("ListDemo"), actions:
<Widget>[IconButton(icon: Icon(Icons.*add*),)],), body:
_buildBody(),),); } @override void dispose()
{ super.dispose();if (_timer != null)
{ _timer.cancel(); } }}

4）、在ListDemo中通过Timer更新InheritedTestModel中的数据，然后再下一个Widget中获取更新的数据作为展示

```
class ListDemo2 extends StatefulWidget{ @override State createState() { return new ListDemoState2(); }}
class ListDemoState2 extends State<ListDemo2>{InheritedTestModel _inheritedTestModel; Widget _buildListItem(BuildContext context,int index) { return Container(height: 50, width: 100, alignment: Alignment.center, child: Text(_inheritedTestModel.getList()[index].toString()), ); }Widget _buildBody()
{ _inheritedTestModel = InheritedContext.of(context).inheritedTestModel;return Container(child: ListView.builder(itemBuilder: (context, index)=>_buildListItem(context,index),itemCount: _inheritedTestModel.getList().length,), ); } @override Widget build(BuildContext context) { return _buildBody(); }}
```

这样就可以在父 widget 中更新数据，子 View 不需任何操作直接从数据容器 InheritedTestModel 中获取到更新后的新数据

这是一个数据共享的简单的例子，基本的流程，大致就是 A 去更新 B 的数据，A 和 B 有一个共同的父类，实现数据的共享

4.上面说了原理和基本的使用，但是在实际项目当中，我当然不建议这样来使用，Google 已经为我们封装好了功能更加强大的插件 Provider，其内部原理就是基于 InheritedWidget 来实现的，我们理解了基本原理，可以更好的在项目中运用 Provider