1 开篇 状态流入 Flutter 源码系列
2022-01-01 · 阅读 2 分钟
2 ActivityA 页面跳转二级缓存页面
2022-01-01 · 阅读 2 分钟
3 状态流入 Flutter 源码系列解读之 Navigator
2022-01-01 · 阅读 2 分钟
4 前端每个月都在内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
5 前端每个月都在内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
6 前端每个月都在内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
7 前端每年又要面对内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
8 前端每年又要面对内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
9 前端每年又要面对内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟
10 前端每年又要面对内卷，又是内卷的一年
2022-01-01 · 阅读 2 分钟

现在我们来看下 StatefulWidget 和 State 的源码实现一般情况，这部分内容主要是 StatefulWidget 的 createElement 方法，StatefulWidget 的 createState 方法，StatefulWidget 的 initState 方法，State 的 initState 和 build 方法，这几个方法的执行顺序，不是很记忆的可能有一定难度。

在本文的最后，会给出下面这个例子的一般情况下的一张时序图。

```
1. Form initStateWidgetBinding()
2. runApp WidgetBinding.scheduleAttachRootWidget(rootWidget) ...Form initStateWidgetBinding()
3. Form initStateWidgetBinding()
```

一、StatefulWidget 类

1. StatefulWidget 类的定义

下面是 StatefulWidget 的定义部分，分析可以，创建 StatefulElement 实例作为 StatefulWidget 的 createElement，和有状态的创建有关系，这个工作 createElement 和 State 工作有关系。

```
abstract class StatefulWidget extends Widget {
  const StatefulWidget(Key? key) : super(key: key);

  @override
  StatefulElement createElement() => StatefulElement(this);

  @override
  @protected
  State createState();
}
```

从这部分分析出：StatefulWidget 只是用来创建 State 的，不用用 State 写事，那么，问题来了，StatefulWidget 是何时被调用，创建 State 的呢？

2. createElement 流程时序图

这个部分内容，我们可以通过时序图来分析它的流程，我们来看一，我们开始，【NO.1】吧！

```
class main {
  runApp(Form? form) {
    ...
    ...
    class Form extends StatefulWidget {
    const Form(Key? key) : super(key: key);

    ...
    ...
    @override
    StatefulElement createElement() => StatefulElement(this);
    }

    class _FormState extends State<Form> {
    ...
    ...
    @override
    Widget build(BuildContext context) {
      return const Column(
        children: <Widget>[
          width: 100,
          height: 100,
          color: Colors.red,
          child: Text('a.red'),
          ),
        ],
      );
    }
    }
}
```

IDEA/AndroidStudio

地下图的内容主要是，通过 StatefulWidget 实例的时间戳，调用 createElement 方法创建 Element 对象为 _Lifeline 的实例对象，然后使用该 _Lifeline 对象的 _Lifeline 的 _unlinked 属性，然后使用该 _Lifeline 对象的 _Lifeline 的 _unlinked 属性的值 false 和 true 对象，来使用不同的实现模式。

二、State 类生命周期

1. State 类的定义

State 是抽象类，只定义了一个 build 抽象方法，用于构建 Widget 对象，它的用途有可以重写 initState 和 dispose 两个功能方法，用于创建和销毁资源的时机，下面是 State 的定义部分内容

```
abstract class State<T extends StatefulWidget> {
  /// Called when this object is inserted into the tree.
  ///
  /// The framework will call this method exactly once for each
  /// [State] object it creates.
  ///
  /// Overrides should start with a call to the inherited method
  /// of the same name.
  ///
  /// See also:
  ///
  /// * [Object.initState], which has more details about this method.
  ///
  /// * [State.initStateWidgetBinding()] ...
  ///
  /// * [State.initStateWidgetBinding()] ...
  @protected
  @mustCallSuper
  initStateWidgetBinding();

  /// Called when this object is removed from the tree.
  ///
  /// The framework calls this method when this [State] object
  /// is removed from the tree.
  ///
  /// Subclasses usually override this method to release any
  /// resources held by this [State] object.
  ///
  /// Overrides should end with a call to the inherited method
  /// of the same name.
  ///
  /// See also:
  ///
  /// * [Object.dispose], which has more details about this method.
  @protected
  @mustCallSuper
  dispose();

  /// Describes the part of the user interface represented by this widget.
  ///
  /// The framework calls this method in a number of different situations. For
  /// example, after calling [setState], or when the parent of this widget
  /// rebuilds itself.
  ///
  /// Typically implementations return a newly created constellation of
  /// widgets that are configured with information from this widget and
  /// from this [State] object.
  ///
  /// Given these inputs, the returned widgets should
  /// be a complete representation of the user interface
  /// represented by this [State] object.
  ///
  /// See also:
  ///
  /// * [StatefulWidget], which contains the [State] that
  /// * [Widget], which is the base class for all widgets.
  @protected
  @mustCallSuper
  Widget build(BuildContext context);
}
```

IDEA/AndroidStudio

2. 元素类型时 State 的时间表

现在思考一下，State 对象只是创建类型时创建对象的，编辑 State 的整个时间，因为 State 的时间，State 对象，那是何时 State 对象的时间呢？答案是 StatefulElement 元素类型，StatefulElement 元素类型，下面是 StatefulElement 元素类型的定义部分，编辑类型时间表情况 State 的时间表情况。

```
class StatefulElement extends ComponentElement {
  StatefulElement(StatefulWidget widget)
      : _state = widget.createState(),
        _stateBeingCreated = true,
        super(widget) {
    final State<dynamic> state = _state;
    assert(state._debugLifecycleState == _StateLifecycle.created);
    assert(state._debugBuilder == null);
    assert(state._debugChangedDependencies == false);
    assert(state._debugDeactivated == false);
    assert(state._debugDependencies == null);
    assert(state._debugSetupInitialized == false);
    assert(state._debugTicker == null);
    assert(state._debugWidget == null);
    assert(state._debugElement == null);
    assert(state._debugLifecycleState == _StateLifecycle.created);
  }

  State<StatefulWidget> _state;
  bool _stateBeingCreated = false;

  @override
  void _debugAddToParent(ElementBase? newParent) {
    assert(_state._debugElement == null);
    assert(_state._debugLifecycleState == _StateLifecycle.defunct);
    super._debugAddToParent(newParent);
    final bool hadAssertions = false;
    assert(() { hadAssertions = true; return true; }());
    assert(!hadAssertions);
    _state._debugElement = this;
    assert(() { _state._debugLifecycleState = _StateLifecycle.initialized; return true; }());
  }

  @override
  void _debugRemoveFromParent(ElementBase? oldParent) {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState == _StateLifecycle.initialized);
    super._debugRemoveFromParent(oldParent);
    final bool hadAssertions = false;
    assert(() { hadAssertions = true; return true; }());
    assert(!hadAssertions);
    _state._debugElement = null;
    assert(() { _state._debugLifecycleState = _StateLifecycle.defunct; return true; }());
  }

  @override
  void _debugMarkDetached() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkDetached();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    _state._debugLifecycleState = _StateLifecycle.defunct;
  }

  @override
  void _debugMarkMoved(int? oldSlot, int? newSlot) {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkMoved(oldSlot, newSlot);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkNewDependencies() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkNewDependencies();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkWasUsedForDependency() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkWasUsedForDependency();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkDirty() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkDirty();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkNeedsBuild() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkNeedsBuild();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkNeedsNotifyDependents() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkNeedsNotifyDependents();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkNeedsReassemble() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkNeedsReassemble();
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
  }

  @override
  void _debugMarkNeedsUpdate() {
    assert(_state._debugElement == this);
    assert(_state._debugLifecycleState != _StateLifecycle.defunct);
    super._debugMarkNeedsUpdate
```