2023-03-22 flutter widget更新2

我是小胡胡分胡

在Flutter中，当调用setState方法时，Flutter会重新构建当前组件及其子组件的widget tree（也就是widget树），并将其与上一次构建的widget tree进行比较，从而确定哪些widget需要更新。

在widget tree比较时，Flutter会递归遍历新旧widget tree的节点，对比它们的类型和key属性是否相同。如果两个节点的类型相同且key相同，则Flutter认为它们是同一个widget，不需要更新，否则需要更新。

当一个widget需要更新时，Flutter会创建一个新的实例，并将它与旧的widget实例进行比较，以确定需要更新哪些部分。这个过程称为diff算法，也就是计算出新widget树和旧widget树的不同之处。

一旦确定需要更新哪些部分，Flutter会将这些更新打包成一个更新操作（也称为element），并添加到一个更新队列中。最后，Flutter会异步地执行这个更新队列，从而更新widget tree。

总之，当我们通过setState方法修改一个widget时，Flutter会递归地比较新旧widget tree的节点，并根据差异进行更新，从而实现界面的刷新。

setState -> markNeedsBuild -> scheduleBuildFor ->onBuildScheduled ->

SchedulerBinding.ensureVisualUpdate ->
SchedulerBinding.scheduleFrame(); ->
platformDispatcher.scheduleFrame();

1、setState

```
setState
  void setState(VoidCallback fn) {
    assert(fn != null);
    assert(() {
      if (_debugLifecycleState == _StateLifecycle.defunct)
{
        throw FlutterError.fromParts(<DiagnosticsNode>[
          ErrorSummary('setState() called after dispose():
$this'),
          ErrorDescription(
            'This error happens if you call setState() on a
State object for a widget that '
            'no longer appears in the widget tree (e.g.,
whose parent widget no longer '
            'includes the widget in its build). This error
can occur when code calls '
            'setState() from a timer or an animation
callback.',
          ),
          ErrorHint(
            'The preferred solution is '
            'to cancel the timer or stop listening to the
animation in the dispose() '
            'callback. Another solution is to check the
"mounted" property of this '
            'object before calling setState() to ensure the
object is still in the '
```

```
            'tree.',
          ),
          ErrorHint(
            'This error might indicate a memory leak if
setState() is being called '
            'because another object is retaining a
reference to this State object '
            'after it has been removed from the tree. To
avoid memory leaks, '
            'consider breaking the reference to this object
during dispose().',
          ),
        ]);
      }
      if (_debugLifecycleState == _StateLifecycle.created
&& !mounted) {
        throw FlutterError.fromParts(<DiagnosticsNode>[
          ErrorSummary('setState() called in constructor:
$this'),
          ErrorHint(
            'This happens when you call setState() on a
State object for a widget that '
            "hasn't been inserted into the widget tree yet.
It is not necessary to call "
            'setState() in the constructor, since the state
is already assumed to be dirty '
            'when it is initially created.',
          ),
        ]);
      }
      return true;
    }());
    final Object? result = fn() as dynamic;
    assert(() {
```

```
      if (result is Future) {
        throw FlutterError.fromParts(<DiagnosticsNode>[
          ErrorSummary('setState() callback argument
returned a Future.'),
          ErrorDescription(
            'The setState() method on $this was called with
a closure or method that '
            'returned a Future. Maybe it is marked as
"async".',
          ),
          ErrorHint(
            'Instead of performing asynchronous work inside
a call to setState(), first '
            'execute the work (without updating the widget
state), and then synchronously '
            'update the state inside a call to
setState().',
          ),
        ]);
      }
      // We ignore other types of return values so that you
can do things like:
      //   setState(() => x = 3);
      return true;
    }());
    _element!.markNeedsBuild();
  }
```

2、markNeedsBuild

3、scheduleBuildFor

scheduleBuildFor 方法实现：

```
  void scheduleBuildFor(Element element) {
    assert(element != null);
    assert(element.owner == this);
    assert(() {
      if (debugPrintScheduleBuildForStacks)
        debugPrintStack(label: 'scheduleBuildFor() called
for $element${_dirtyElements.contains(element) ? " (ALREADY
IN LIST)" : ""}');
      if (!element.dirty) {
        throw FlutterError.fromParts(<DiagnosticsNode>[
          ErrorSummary('scheduleBuildFor() called for a
widget that is not marked as dirty.'),
          element.describeElement('The method was called
for the following element'),
          ErrorDescription(
            'This element is not current marked as dirty.
Make sure to set the dirty flag before '
            'calling scheduleBuildFor().',
          ),
          ErrorHint(
            'If you did not attempt to call
scheduleBuildFor() yourself, then this probably '
            'indicates a bug in the widgets framework.
Please report it:\n'
            '  https://github.com/flutter/flutter/issues/
new?template=2_bug.md',
          ),
        ]);
      }
      return true;
    }());
    if (element._inDirtyList) {
      assert(() {
        if (debugPrintScheduleBuildForStacks)
```

```
          debugPrintStack(label:
'BuildOwner.scheduleBuildFor() called;
_dirtyElementsNeedsResorting was
$_dirtyElementsNeedsResorting (now true); dirty list is:
$_dirtyElements');
        if (!_debugIsInBuildScope) {
          throw FlutterError.fromParts(<DiagnosticsNode>[
            ErrorSummary('BuildOwner.scheduleBuildFor()
called inappropriately.'),
            ErrorHint(
              'The BuildOwner.scheduleBuildFor() method
should only be called while the '
              'buildScope() method is actively rebuilding
the widget tree.',
            ),
          ]);
        }
        return true;
      }());
      _dirtyElementsNeedsResorting = true;
      return;
    }
    if (!_scheduledFlushDirtyElements && onBuildScheduled !
= null) {
      _scheduledFlushDirtyElements = true;
      onBuildScheduled!();
    }
    _dirtyElements.add(element);
    element._inDirtyList = true;
    assert(() {
      if (debugPrintScheduleBuildForStacks)
        debugPrint('...dirty list is now:
$_dirtyElements');
      return true;
```

```
    }());
  }
```

主要是是

- onBuildScheduled
- _dirtyElements.add(element);

onBuildScheduled ->

WidgetsBinding._handleBuildScheduled

```
  void _handleBuildScheduled() {
    // If we're in the process of building dirty elements, then changes
    // should not trigger a new frame.
    assert(() {
      if (debugBuildingDirtyElements) {
        throw FlutterError.fromParts(<DiagnosticsNode>[
          ErrorSummary('Build scheduled during frame.'),
          ErrorDescription(
            'While the widget tree was being built, laid out, and painted, '
            'a new frame was scheduled to rebuild the widget tree.',
          ),
          ErrorHint(
            'This might be because setState() was called from a layout or '
            'paint callback. '
            'If a change is needed to the widget tree, it should be applied '
            'as the tree is being built. Scheduling a
```

```
change for the subsequent '
            'frame instead results in an interface that
lags behind by one frame. '
            'If this was done to make your build dependent
on a size measured at '
            'layout time, consider using a LayoutBuilder,
CustomSingleChildLayout, '
            'or CustomMultiChildLayout. If, on the other
hand, the one frame delay '
            'is the desired effect, for example because
this is an '
            'animation, consider scheduling the frame in a
post-frame callback '
            'using SchedulerBinding.addPostFrameCallback or
'
            'using an AnimationController to trigger the
animation.',
          ),
        ]);
      }
      return true;
    }());
    ensureVisualUpdate();
  }
```

4、ensureVisualUpdate

```
  void ensureVisualUpdate() {
```

```
    switch (schedulerPhase) {
      case SchedulerPhase.idle:
      case SchedulerPhase.postFrameCallbacks:
        scheduleFrame();
        return;
      case SchedulerPhase.transientCallbacks:
      case SchedulerPhase.midFrameMicrotasks:
      case SchedulerPhase.persistentCallbacks:
        return;
    }
  }
```

5、scheduleFrame

```
  void scheduleFrame() {
    if (_hasScheduledFrame || !framesEnabled)
      return;
    assert(() {
      if (debugPrintScheduleFrameStacks)
        debugPrintStack(label: 'scheduleFrame() called.
Current phase is $schedulerPhase.');
      return true;
    }());
    ensureFrameCallbacksRegistered();
    platformDispatcher.scheduleFrame();
    _hasScheduledFrame = true;
  }
```

主要是：

- ensureFrameCallbacksRegistered();
- platformDispatcher.scheduleFrame();

6、ensureFrameCallbacksRegistered

```
  void ensureFrameCallbacksRegistered() {
    platformDispatcher.onBeginFrame ??= _handleBeginFrame;
    platformDispatcher.onDrawFrame ??= _handleDrawFrame;
  }
```

7、platformDispatcher.scheduleFrame();

```
  void scheduleFrame() native
'PlatformConfiguration_scheduleFrame';
```

最后调用 native：

```
  ui.PlatformDispatcher get platformDispatcher =>
ui.PlatformDispatcher.instance;
```

platformDispatcher.scheduleFrame();->
platformDispatcher.onDrawFrame ??= _handleDrawFrame;
-> binding.dart void _handleDrawFrame()
-> binding.dart handleDrawFrame();
->

```
      for (final FrameCallback callback in
_persistentCallbacks)
        _invokeFrameCallback(callback,
_currentFrameTimeStamp!);
```

->

```
 @pragma('vm:notify-debugger-on-exception')
  void _invokeFrameCallback(FrameCallback callback,
Duration timeStamp, [ StackTrace? callbackStack ])
```

->RenderBinding _handlePersistentFrameCallback

```
  void _handlePersistentFrameCallback(Duration timeStamp) {
    drawFrame();
    _scheduleMouseTrackerUpdate();
  }
```

-> WidgetBinding void drawFrame()

```
  void drawFrame() {
    assert(!debugBuildingDirtyElements);
    assert(() {
      debugBuildingDirtyElements = true;
      return true;
    }());

    TimingsCallback? firstFrameCallback;
    if (_needToReportFirstFrame) {
      assert(!_firstFrameCompleter.isCompleted);

      firstFrameCallback = (List<FrameTiming> timings) {
        assert(sendFramesToEngine);
        if (!kReleaseMode) {
          // Change the current user tag back to the
default tag. At this point,
          // the user tag should be set to
"AppStartUp" (originally set in the
          // engine), so we need to change it back to the
default tag to mark
          // the end of app start up for CPU profiles.
          developer.UserTag.defaultTag.makeCurrent();
          developer.Timeline.instantSync('Rasterized first
useful frame');
          developer.postEvent('Flutter.FirstFrame',
<String, dynamic>{});
        }
```

```
SchedulerBinding.instance.removeTimingsCallback(firstFrameC
allback!);
        firstFrameCallback = null;
        _firstFrameCompleter.complete();
      };
      // Callback is only invoked when FlutterView.render
is called. When
      // sendFramesToEngine is set to false during the
frame, it will not be
      // called and we need to remove the callback (see
below).

SchedulerBinding.instance.addTimingsCallback(firstFrameCall
back!);
    }

    try {
      if (renderViewElement != null)
        buildOwner!.buildScope(renderViewElement!);
      super.drawFrame();
      buildOwner!.finalizeTree();
    } finally {
      assert(() {
        debugBuildingDirtyElements = false;
        return true;
      }());
    }
    if (!kReleaseMode) {
      if (_needToReportFirstFrame && sendFramesToEngine) {
        developer.Timeline.instantSync('Widgets built first
useful frame');
      }
    }
    _needToReportFirstFrame = false;
```

```
    if (firstFrameCallback != null && !sendFramesToEngine)
{
      // This frame is deferred and not the first frame
sent to the engine that
      // should be reported.
      _needToReportFirstFrame = true;

SchedulerBinding.instance.removeTimingsCallback(firstFrameC
allback!);
    }
  }
```

-> buildOwner!.buildScope(renderViewElement!);

- 最后调用到

  buildOwner!.buildScope(renderViewElement!);

8、buildScope

```
 @pragma('vm:notify-debugger-on-exception')
  void buildScope(Element context, [ VoidCallback?
callback ]) {
    if (callback == null && _dirtyElements.isEmpty)
      return;
    assert(context != null);
    assert(_debugStateLockLevel >= 0);
    assert(!_debugBuilding);
    assert(() {
      if (debugPrintBuildScope)
        debugPrint('buildScope called with context
$context; dirty list is: $_dirtyElements');
      _debugStateLockLevel += 1;
      _debugBuilding = true;
      return true;
    }());
```

```
    if (!kReleaseMode) {
      Map<String, String> debugTimelineArguments =
timelineArgumentsIndicatingLandmarkEvent;
      assert(() {
        if (debugProfileBuildsEnabled) {
          debugTimelineArguments = <String, String>{
            ...debugTimelineArguments,
            'dirty count': '${_dirtyElements.length}',
            'dirty list': '$_dirtyElements',
            'lock level': '$_debugStateLockLevel',
            'scope context': '$context',
          };
        }
        return true;
      }());
      Timeline.startSync(
        'BUILD',
        arguments: debugTimelineArguments
      );
    }
    try {
      _scheduledFlushDirtyElements = true;
      if (callback != null) {
        assert(_debugStateLocked);
        Element? debugPreviousBuildTarget;
        assert(() {

context._debugSetAllowIgnoredCallsToMarkNeedsBuild(true);
          debugPreviousBuildTarget =
_debugCurrentBuildTarget;
          _debugCurrentBuildTarget = context;
          return true;
        }());
        _dirtyElementsNeedsResorting = false;
```

```
        try {
          callback();
        } finally {
          assert(() {

context._debugSetAllowIgnoredCallsToMarkNeedsBuild(false);
            assert(_debugCurrentBuildTarget == context);
            _debugCurrentBuildTarget =
debugPreviousBuildTarget;
            _debugElementWasRebuilt(context);
            return true;
          }());
        }
      }
      _dirtyElements.sort(Element._sort);
      _dirtyElementsNeedsResorting = false;
      int dirtyCount = _dirtyElements.length;
      int index = 0;
      while (index < dirtyCount) {
        final Element element = _dirtyElements[index];
        assert(element != null);
        assert(element._inDirtyList);
        assert(() {
          if (element._lifecycleState ==
_ElementLifecycle.active &&
element._debugIsInScope(context)) {
            throw FlutterError.fromParts(<DiagnosticsNode>[
              ErrorSummary('Tried to build dirty widget in
the wrong build scope.'),
              ErrorDescription(
                'A widget which was marked as dirty and is
still active was scheduled to be built, '
                'but the current build scope unexpectedly
does not contain that widget.',
```

```
            ),
            ErrorHint(
              'Sometimes this is detected when an element
is removed from the widget tree, but the '
              'element somehow did not get marked as
inactive. In that case, it might be caused by '
              'an ancestor element failing to implement
visitChildren correctly, thus preventing '
              'some or all of its descendants from being
correctly deactivated.',
            ),
            DiagnosticsProperty<Element>(
              'The root of the build scope was',
              context,
              style: DiagnosticsTreeStyle.errorProperty,
            ),
            DiagnosticsProperty<Element>(
              'The offending element (which does not
appear to be a descendant of the root of the build scope)
was',
              element,
              style: DiagnosticsTreeStyle.errorProperty,
            ),
          ]);
        }
        return true;
      }());
      final bool isTimelineTracked = !kReleaseMode &&
_isProfileBuildsEnabledFor(element.widget);
      if (isTimelineTracked) {
        Map<String, String> debugTimelineArguments =
timelineArgumentsIndicatingLandmarkEvent;
        assert(() {
          if (kDebugMode) {
```

```
              debugTimelineArguments =
element.widget.toDiagnosticsNode().toTimelineArguments();
            }
            return true;
          }());
          Timeline.startSync(
            '${element.widget.runtimeType}',
            arguments: debugTimelineArguments,
          );
        }
        try {
          element.rebuild();
        } catch (e, stack) {
          _debugReportException(
            ErrorDescription('while rebuilding dirty
elements'),
            e,
            stack,
            informationCollector: () => <DiagnosticsNode>[
              if (kDebugMode && index <
_dirtyElements.length)

DiagnosticsDebugCreator(DebugCreator(element)),
              if (index < _dirtyElements.length)
                element.describeElement('The element being
rebuilt at the time was index $index of $dirtyCount')
              else
                ErrorHint('The element being rebuilt at the
time was index $index of $dirtyCount, but _dirtyElements
only had ${_dirtyElements.length} entries. This suggests
some confusion in the framework internals.'),
            ],
          );
        }
```

```
        if (isTimelineTracked)
          Timeline.finishSync();
        index += 1;
        if (dirtyCount < _dirtyElements.length ||
_dirtyElementsNeedsResorting!) {
          _dirtyElements.sort(Element._sort);
          _dirtyElementsNeedsResorting = false;
          dirtyCount = _dirtyElements.length;
          while (index > 0 && _dirtyElements[index -
1].dirty) {
            // It is possible for previously dirty but
inactive widgets to move right in the list.
            // We therefore have to move the index left in
the list to account for this.
            // We don't know how many could have moved.
However, we do know that the only possible
            // change to the list is that nodes that were
previously to the left of the index have
            // now moved to be to the right of the right-
most cleaned node, and we do know that
            // all the clean nodes were to the left of the
index. So we move the index left
            // until just after the right-most clean node.
            index -= 1;
          }
        }
      }
      assert(() {
        if (_dirtyElements.any((Element element) =>
element._lifecycleState == _ElementLifecycle.active &&
element.dirty)) {
          throw FlutterError.fromParts(<DiagnosticsNode>[
            ErrorSummary('buildScope missed some dirty
elements.'),
```

```
            ErrorHint('This probably indicates that the
dirty list should have been resorted but was not.'),
            Element.describeElements('The list of dirty
elements at the end of the buildScope call was',
_dirtyElements),
          ]);
        }
        return true;
      }());
    } finally {
      for (final Element element in _dirtyElements) {
        assert(element._inDirtyList);
        element._inDirtyList = false;
      }
      _dirtyElements.clear();
      _scheduledFlushDirtyElements = false;
      _dirtyElementsNeedsResorting = null;
      if (!kReleaseMode) {
        Timeline.finishSync();
      }
      assert(_debugBuilding);
      assert(() {
        _debugBuilding = false;
        _debugStateLockLevel -= 1;
        if (debugPrintBuildScope)
          debugPrint('buildScope finished');
        return true;
      }());
    }
    assert(_debugStateLockLevel >= 0);
  }
```

核心方法是调用 **element.rebuild();**

循环遍历 dirty 脏标记的 element 调用 rebuild 方法

```
   while (index < dirtyCount) {
        final Element element = _dirtyElements[index];
      element.rebuild();
}
```

- setState 的那个 StatefullWidget
- rebuild
- performRebuild
- build widget
- updateChild 比较 2 个 widget
- update -》 rebuild
- rebuild 到 child widget 递归遍历子 widget
- inflateWidget 创建新 newWidget.createElement();
- newChild.mount(this, newSlot)
- _firstBuild-》 rebuild
- parentDataWidget.applyParentData(renderObject);
- markNeedsLayout
- owner!.requestVisualUpdate();

```
void requestVisualUpdate() {
    onNeedVisualUpdate?.call();
  }
```

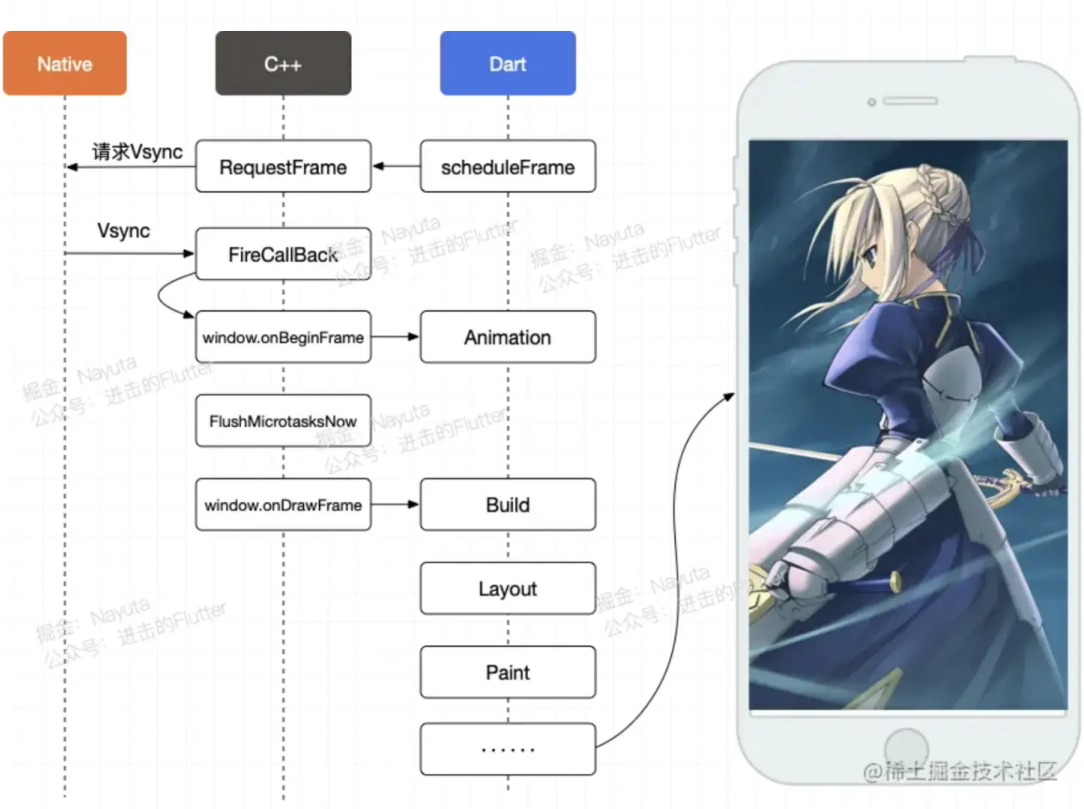
- ensureVisualUpdate

```
void ensureVisualUpdate() {
  switch (schedulerPhase) {
    case SchedulerPhase.idle:
    case SchedulerPhase.postFrameCallbacks:
      scheduleFrame();
      return;
    case SchedulerPhase.transientCallbacks:
    case SchedulerPhase.midFrameMicrotasks:
    case SchedulerPhase.persistentCallbacks:
      return;
  }
}
```

SchedulerBinding.scheduleFrame();

最后又回到了 scheduleFrame

看看别人的分析吧：



Flutter 核心渲染流程分析 [完结篇]： https://juejin.cn/post/6973818961724964901

Flutter渲染机制—UI线程： http://gityuan.com/2019/06/15/flutter_ui_draw/