前言

大家可能或多或少知道 SingleChildScrollView 组件在性能上比 ListView 等 ScrollView 一族组件要差。但至于问什么，可能很多事也说不清楚，本章我们将从 SingleChildScrollView 组件源码实现的角度来看一下，它和 ScrollView 一族到底有什么区别，它的弊端在本质上是由什么决定的。

一、认识 SingleChildScrollView

1. 构造方法

如下构造方法中，除了 child 入参外，其余入参我们在前面章节中都有详细介绍，它们在这里的含义和用法也是一样的。可以看出 SingleChildScrollView 组件其实功能非常单一，就是让一个组件可以支持滚动。

```dart
class SingleChildScrollView extends StatelessWidget {
  /// Creates a box in which a single widget can be scrolled.
  const SingleChildScrollView({
    Key? key,
    this.scrollDirection = Axis.vertical,
    this.reverse = false,
    this.padding,
    bool? primary,
    this.physics,
    this.controller,
    this.child,
    this.dragStartBehavior = DragStartBehavior.start,
    this.clipBehavior = Clip.hardEdge,
    this.restorationId,
    this.keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual,
  }) : assert(scrollDirection != null),
```

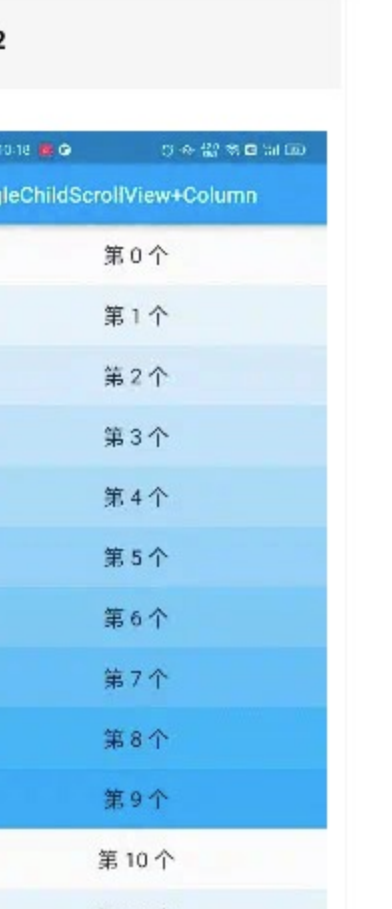
那么问题来了：既然只要为组件套一个 SingleChildScrollView 组件，就可以实现其滚动的功能。那么为什么还有整出 ScrollView 一族呢？

```dart
/// When you have a list of children and do not require cross-axis
/// shrink-wrapping behavior, for example a scrolling list that is always the
/// width of the screen, consider [ListView], which is vastly more efficient
/// that a [SingleChildScrollView] containing a [ListBody] or [Column] with
/// many children.
```

在源码的介绍中提及：如果子组件列表中孩子非常多，而且在交叉轴方向的尺寸是固定的，使用 ListView 要远优于 通过 SingleChildScrollView 来包裹 ListBody 和 Column 。

2. SingleChildScrollView 的使用

我们知道 Column 组件本身是不正常滑动的，它可以将多个子组件竖直排列。但当其内容超过父类约束尺寸后，就会出现如下左图的异常提示。这时，在其外部嵌套 SingleChildScrollView 就可以让 Column 支持滑动。

测试1	测试2
	

```dart
---->[15/01_SingleChildScrollView/main.dart]----
// 测试1
Column(
  children: children,
)

// 测试2
SingleChildScrollView(
  child: Column(
    children: children,
  ),
)
```

3. 初看 SingleChildScrollView 的弊端

上面测试中使用了如下 64 个条目数据，通过日志可以看出：通过这种方式实现滑动，在一开始就会实例化所有组件，并且 执行所有组件的 build 逻辑 ，而且在滑动的过程中也不会去销毁条目。因为 SingleChildScrollView 并没有缓存机制，对应的渲染对象就 不存在 预加载和删除子渲染对象的概念，所有这在大批量数据构建列表时是万万不可取的。

```dart
final List<int> data = List.generate(64, (index) => index);

final List<Widget> children = data.map((index) => ItemBox(index: index)).toList();
```

```
I/flutter ( 4544): ----构建ItemBox-----54--------
I/flutter ( 4544): ----构建ItemBox-----55--------
I/flutter ( 4544): ----构建ItemBox-----56--------
I/flutter ( 4544): ----构建ItemBox-----57--------
I/flutter ( 4544): ----构建ItemBox-----58--------
I/flutter ( 4544): ----构建ItemBox-----59--------
I/flutter ( 4544): ----构建ItemBox-----60--------
I/flutter ( 4544): ----构建ItemBox-----61--------
I/flutter ( 4544): ----构建ItemBox-----62--------
I/flutter ( 4544): ----构建ItemBox-----63--------
```

这样我们对 SingleChildScrollView 组件有了一个简单的认识，下面一起从源码的角度来分析一些其根本的原因。

二、SingleChildScrollView 源码探索

1. SingleChildScrollView 的 build 方法

SingleChildScrollView 作为一个 StatelessWidget 的子类，就注定它本身只是对已有组件的拼合。核心的逻辑处理只有 build 方法。如下源码所示， SingleChildScrollView 是主要基于 Scrollable 和 _SingleChildViewport 组件进行组合的，绝大多数的属性的作用也是构造这两个组件对象。

```dart
@override
Widget build(BuildContext context) {
  final AxisDirection axisDirection = _getDirection(context);
  Widget? contents = child;
  if (padding != null)
    contents = Padding(padding: padding!, child: contents);
  final ScrollController? scrollController = primary
    ? PrimaryScrollController.of(context)
    : controller;
  Widget scrollable = Scrollable(
    dragStartBehavior: dragStartBehavior,
    axisDirection: axisDirection,
    controller: scrollController,
    physics: physics,
    restorationId: restorationId,
    viewportBuilder: (BuildContext context, ViewportOffset offset) {
      return _SingleChildViewport(
        axisDirection: axisDirection,
        offset: offset,
        clipBehavior: clipBehavior,
        child: contents,
      ); // _SingleChildViewport
    },
  ); // Scrollable
```

前面我们知道 Scrollable 组件主要用于处理手势的拖拽事件交互，它会将 ViewportOffset 对象通过 viewportBuilder 回调传递出来，以供视口组件提供偏移量来处理显示的内容。可以看出对于 SingleChildScrollView 滑动体而言，其使用的视口是 _SingleChildViewport 组件。
和 ScrollView 相比，两者只有视口组件不同，使用两者的差异就体现在视口的实现了。也就是说 SingleChildScrollView 和 ScrollView 一族的差异，就等价于 _SingleChildViewport 组件和 Viewport 组件的差异。

2. 认识 _SingleChildViewport 组件

实现 Viewport 继承自 MultiChildRenderObjectWidget ，表示它可以接受多个孩子作为内容。而 _SingleChildViewport 组件继承自 SingleChildRenderObjectWidget ，也就是说它只能有一个孩子。

```dart
@override
RenderViewport createRenderObject(BuildContext context) {
  return RenderViewport(
    axisDirection: axisDirection,
    crossAxisDirection: crossAxisDirection ?? Viewport.get...
    anchor: anchor,
    offset: offset,
    cacheExtent: cacheExtent,
    cacheExtentStyle: cacheExtentStyle,
    clipBehavior: clipBehavior,
  );
}
```

Viewport 组件 createRenderObject 方法

```dart
@override
_RenderSingleChildViewport createRenderObject(BuildContext context) {
  return _RenderSingleChildViewport(
    axisDirection: axisDirection,
    offset: offset,
    clipBehavior: clipBehavior,
  );
}

@override
void updateRenderObject(BuildContext context, _RenderSingleChildViewport ...
```

_SingleChildViewport 组件 createRenderObject 方法

它们二者都作为 RenderObjectWidget 的实现类类，使用需要覆写 createRenderObject 方法，来创建渲染对象。其中 Viewport 组件会创建 RenderViewport 渲染对象， _SingleChildViewport 组件会创建 _RenderSingleChildViewport 渲染对象。

3. 探索 _RenderSingleChildViewport 组件

我们在前面已经认识过 RenderViewport ，其内部会维护一个 双链表 来维护子渲染对象列表。在滑动时，会创建 SliverConstraints 约束传递给子渲染对象进行布局，其中包括缓存区域的相关信息，这样子渲染对象就可以根据缓存区域来确定构建的范围和处理垃圾，从而实现缓存机制。

从上面的表现上来看， _RenderSingleChildViewport 应该没有处理缓存相关的事宜。而且由于 _SingleChildViewport 只有一个孩子，也表示 _RenderSingleChildViewport 只有一个子渲染对象。从类定义上来看，它继承自 RenderBox ，并实现了 RenderAbstractViewport 。

```dart
class _RenderSingleChildViewport
    extends RenderBox
    with RenderObjectWithChildMixin<RenderBox>
    implements RenderAbstractViewport {...}
```

另外，它混入了泛型为 RenderBox 的 RenderObjectWithChildMixin ，这就表明其子渲染对象的类型必须是 RenderBox 的，这也是为什么 Column 这种 RenderBox 类型的组件可以直接作为滑动内容的原因。从这里也可以看出，其实 SingleChildScrollView 组件实现的滑动和 Sliver 组件是没有半毛钱关系的，因为 Sliver 组件作为滑动内容，其对应的渲染对象类型是 RenderSliver ，根本无法用于 SingleChildScrollView 组件中。比如，你要是强行将 SliverList 塞到 SingleChildScrollView 中，就会异常伺候。

```
Performing hot reload...
Syncing files to device PDNT00...
Reloaded 1 of 558 libraries in 1,032ms.

════════ Exception caught by widgets library ════════════════════
The following assertion was thrown building RawGestureDetector-[LabeledGlobalKey<RawGestureDetectorState>#2459d](s...
[vertical drag], behavior: opaque):
A _RenderSingleChildViewport expected a child of type RenderBox but received a child of type RenderSliverList.

RenderObjects expect specific types of children because they coordinate with their children during layout...
```

```dart
SingleChildScrollView(
  child: SliverList(
    delegate: SliverChildListDelegate(children),
  )
)
```

Column 、 Wrapper 、 Row 、 Stack 等组件，本身就不是为了滑动而存在的。它们对应的渲染对象及其子渲染对象的约束都是盒约束 BoxConstraints ，所以其布局本身没有滑动相关的约束信息，自然也就没有滑动中的缓存区域概念。

对于渲染对象而言，最重要的是布局和绘制逻辑，如下 SingleChildScrollView 对应的渲染对象的布局逻辑，主要是执行 Column 对应渲染对象的布局方法，并传入盒约束对象：

```dart
@override
void performLayout() {
  final BoxConstraints constraints = this.constraints;
  if (child == null) {
    size = constraints.smallest;
  } else {
    child!.layout(_getInnerConstraints(constraints), parentUsesSize: true);
    size = constraints.constrain(child!.size);
  }

  offset.applyViewportDimension(_viewportExtent);
  offset.applyContentDimensions(_minScrollExtent, _maxScrollExtent);
}
```

比如上面的的子组件是 Column 组件，其对应渲染对象是 RenderFlex 。在它的布局方法中，也会遍历执行子渲染对象列表，对执行各个孩子的布局方法。这也是为什么 SingleChildScrollView + Column 在一开始就会构建所有条目的根本原因。

```dart
while (child != null) {
  final FlexParentData childParentData = child.parentData! as FlexParentData;
  final int flex = _getFlex(child);
  if (flex > 0) {...} else {
    final BoxConstraints innerConstraints;
    if (crossAxisAlignment == CrossAxisAlignment.stretch) {...} else {
      switch (_direction) {
        case Axis.horizontal:
          innerConstraints = BoxConstraints(maxHeight: constraints.maxHeight);
          break;
        case Axis.vertical:
          innerConstraints = BoxConstraints(maxWidth: constraints.maxWidth);
          break;
      }
    }
    final Size childSize = layoutChild(child, innerConstraints);
    allocatedSize += _getMainSize(childSize);
    crossSize = math.max(crossSize, _getCrossSize(childSize));
  }
  assert(child.parentData == childParentData);
  child = childParentData.nextSibling;
}
```

在 _RenderSingleChildViewport 的绘制方法中，和 RenderViewport 类似，都是通过滑动偏移量来对绘制内容进行偏移，从而实现内容随拖动事件而滑动。

```dart
@override
void paint(PaintingContext context, Offset offset) {
  if (child != null) {
    final Offset paintOffset = _paintOffset;

    void paintContents(PaintingContext context, Offset offset) {
      context.paintChild(child!, offset + paintOffset);
    }

    if (_shouldClipAtPaintOffset(paintOffset) && clipBehavior != Clip.none) {
      _clipRectLayer.layer = context.pushClipRect(
        needsCompositing,
        offset,
        Offset.zero & size,
        paintContents,
        clipBehavior: clipBehavior,
        oldLayer: _clipRectLayer.layer,
      );
    } else {
      _clipRectLayer.layer = null;
      paintContents(context, offset);
    }
  }
}
```

_RenderSingleChildViewport 渲染对象 paint 绘制方法

所以，总的来看 _RenderSingleChildViewport 就是一个中规中矩的盒型渲染对象，布局和绘制都没有什么亮点，只是根据 Scrollable 的滑动偏移量对绘制内容进行偏移，实现滑动效果而已。

三、SingleChildScrollView 绘制边界问题

下面是 _RenderSingleChildViewport 对 offset 属性进行设置的逻辑，可以看出这里会对 _offset 进行监听，执行 _hasScrolled 方法。也就是每次拖动事件都会触发这个方法，而我们知道拖动事件的触发是非常频繁的。

```dart
set offset(ViewportOffset value) {
  assert(value != null);
  if (value == _offset)
    return;
  if (attached)
    _offset.removeListener(_hasScrolled);
  _offset = value;
  if (attached)
    _offset.addListener(_hasScrolled);
  markNeedsLayout();
}
```

_hasScrolled 的主要逻辑就是执行 markNeedsPaint 进行重绘。这样就会有些问题，在之前探索《重绘范围 RepaintBoundary 一文》 中介绍过，重回范围间的任意一个渲染对象被重新绘制，区间内其他的渲染对象都会执行绘制。而滑动又是高频触发的事件，这样就会导致一些频繁的绘制，如果绘制逻辑非常复杂，是非常不好的。

```dart
---->[_RenderSingleChildViewport#_hasScrolled]----
void _hasScrolled() {
  markNeedsPaint();
  markNeedsSemanticsUpdate();
}
```

其实需要测试一些也非常简单， ItemBox 中有文字，我们知道 Text 组件底层对应的渲染对象是 RenderParagraph 。我们在 RenderParagraph 的 paint 方法中打印一条日志测试一下，可以发现通过 SingleChildScrollView 实现的滑动会不断地触发绘制逻辑。（记得测试后要完删掉打印语句）。

当你运行 【02_ListView】 中的 ListView 测试代码时，就会发现并不会随着滑动一直进行文字绘制。这样就表明 ListView 默认会有对 RepaintBoundary 的优化。默认情况下 ListView 的 addRepaintBoundaries 是 true ，在代理类创建条目对象时，如果 addRepaintBoundaries 为 true ，会默认套上 RepaintBoundary 将条目组件对应的渲染对象将进行隔断，这就是为什么 ListView 在滑动时，并不会频繁触发子渲染对象绘制的根本原因。

```dart
@override
Widget? build(BuildContext context, int index) {
  assert(children != null);
  if (index < 0 || index >= children.length)
    return null;
  Widget child = children[index];
  final Key? key = child.key != null? _SaltedValueKey(child.key!) : null;
  assert(
    child != null,
    "The sliver's children must not contain null values, but a null value was found at index $index"
  );
  if (addRepaintBoundaries)
    child = RepaintBoundary(child: child);
```

大家也可以通过将 addRepaintBoundaries 设为 false ，来对比一下区别。对于 SingleChildScrollView 也一样，如果内部的绘制内容非常复杂，我们可以自己为条目组件套上 RepaintBoundary 。

```dart
final List<Widget> children = data
    .map((index) => RepaintBoundary(
      child: ItemBox(index: index ))).toList();
```

到这里大家可能感觉 SingleChildScrollView 组件似乎没有什么优点，什么都比不过 ScrollView 一族。如果在弹出框等场景下产生了越界，想支持滑动，使用 SingleChildScrollView 组件也未尝不可。这样最简单方便，由于内容并不是太多，性能影响也不是很大，在可容忍的范围内，自然越方便越好，但同时要注意是否有复杂的绘制来确定是否需要加 RepaintBoundary 。
一般，内容列表尽可能使用 ScrollView 一族来完成滑动，但千万不要为了方便使用 SingleChildScrollView + Column ，这样是非常不明智的。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论