React Native 中 ScrollView 性能探究

2016-07-07 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：Race604（@Android笔记）
> 链接：http://www.race604.com/react-native-scrollview-performance/

1 基本使用

ScrollView 是 React Native（后面简称：RN） 中最常见的组件之一。理解 ScrollView 的原理，有利于写出高性能的 RN 应用。

ScrollView 的基本使用也非常简单，如下：

```
<ScrollView>
  <Child1 />
  <Child2 />
  ...
</ScrollView>
```

它和 View 组件一样，可以包含一个或者多个子组件。对子组件的布局可以是垂直或者水平的，通过属性 horizontal=true/false 来控制。甚至还默认支持“下拉”刷新操作。另外还有一个特别赞的特性，超出屏幕的 View 会自动被移除，从而节省资源和提高绘制效率。我们来看如下一个例子：

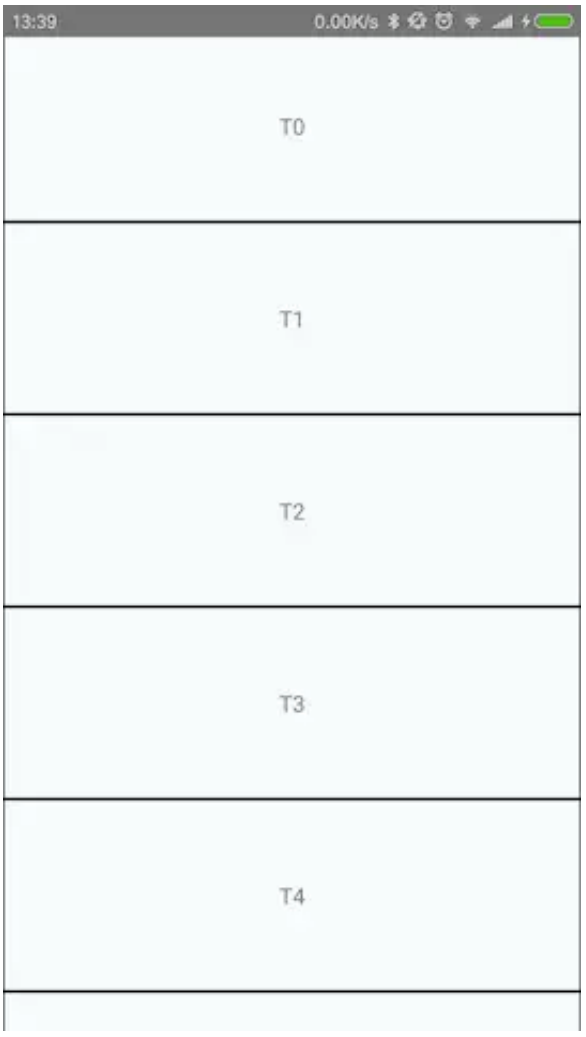
```
class ScrollViewTest extends Component {

  render() {
    let children = [];

    for (var i = 0; i < 20; i++) {
      children.push(
        <View key={"key_" + i} style={styles.child}>
          <Text>{"T" + i}</Text>
        </View>);
    }
```

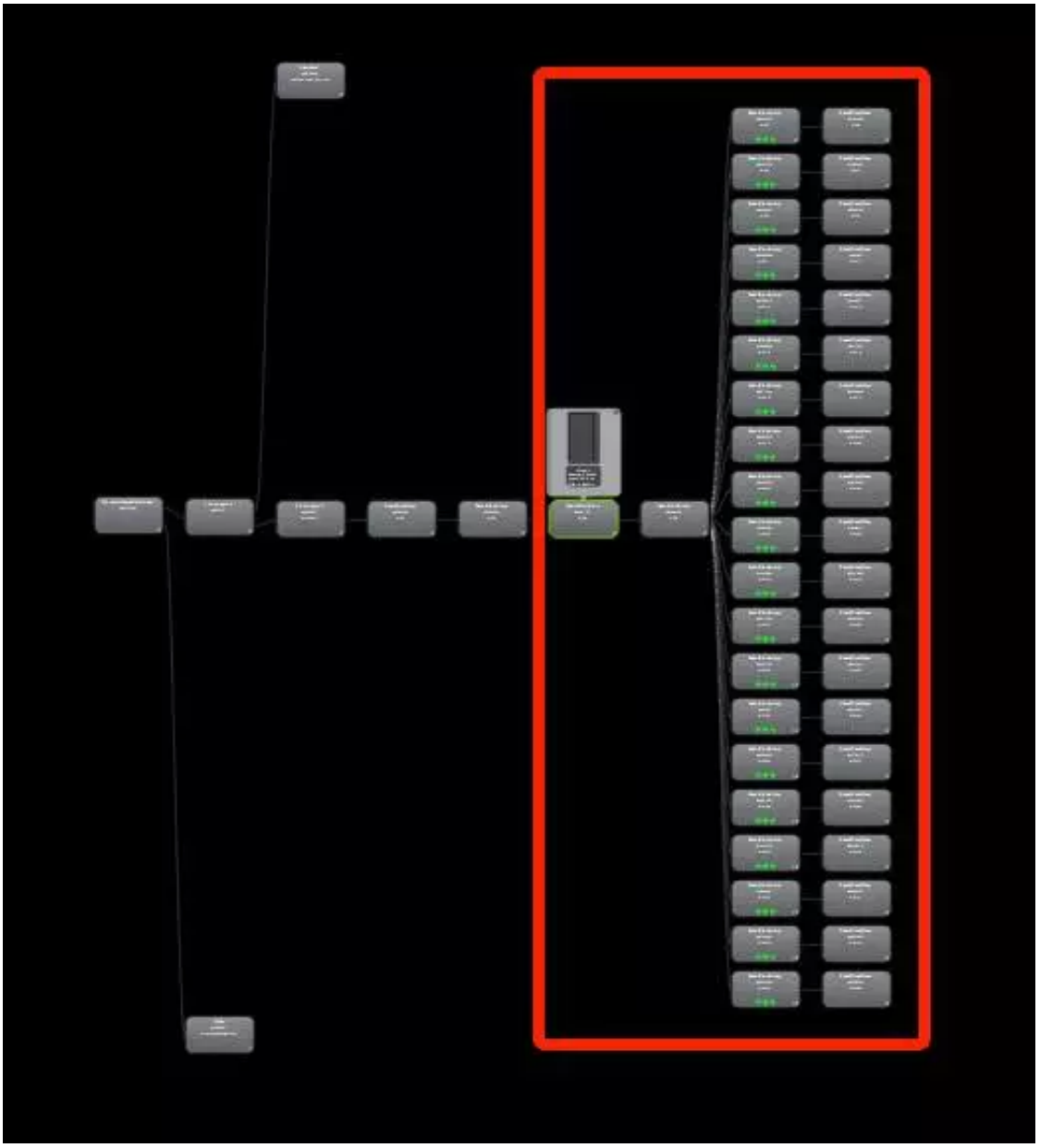
```
    return (
      <ScrollView style={styles.scrollView}>
       {children}
      </ScrollView>
    );
  }
}
```

在 Android 上的效果如下：



如图，我们在 ScrollView 中添加了 20 个子组件，但是我们的屏幕任意时刻最多只能显示 5 个子项目。

下面我们来看实际对应的 Native 控件的情况。RN 中的 ScrollView 对应到 Native 的 RCTScrollView，自动把子组件包含在一个 ViewGroup 中（因为Android 的 ScrollView 只能有一个直接子控件），如下图中的红色框内：



注意到，我们在 JS 中添加了 20 个子组件，但是在 RCTViewGroup 中只有在屏幕上显示的 5 个子控件，在屏幕外的组件，也会自动添加到 View 树中，这与 Native 的 ScrollView 表现一致。

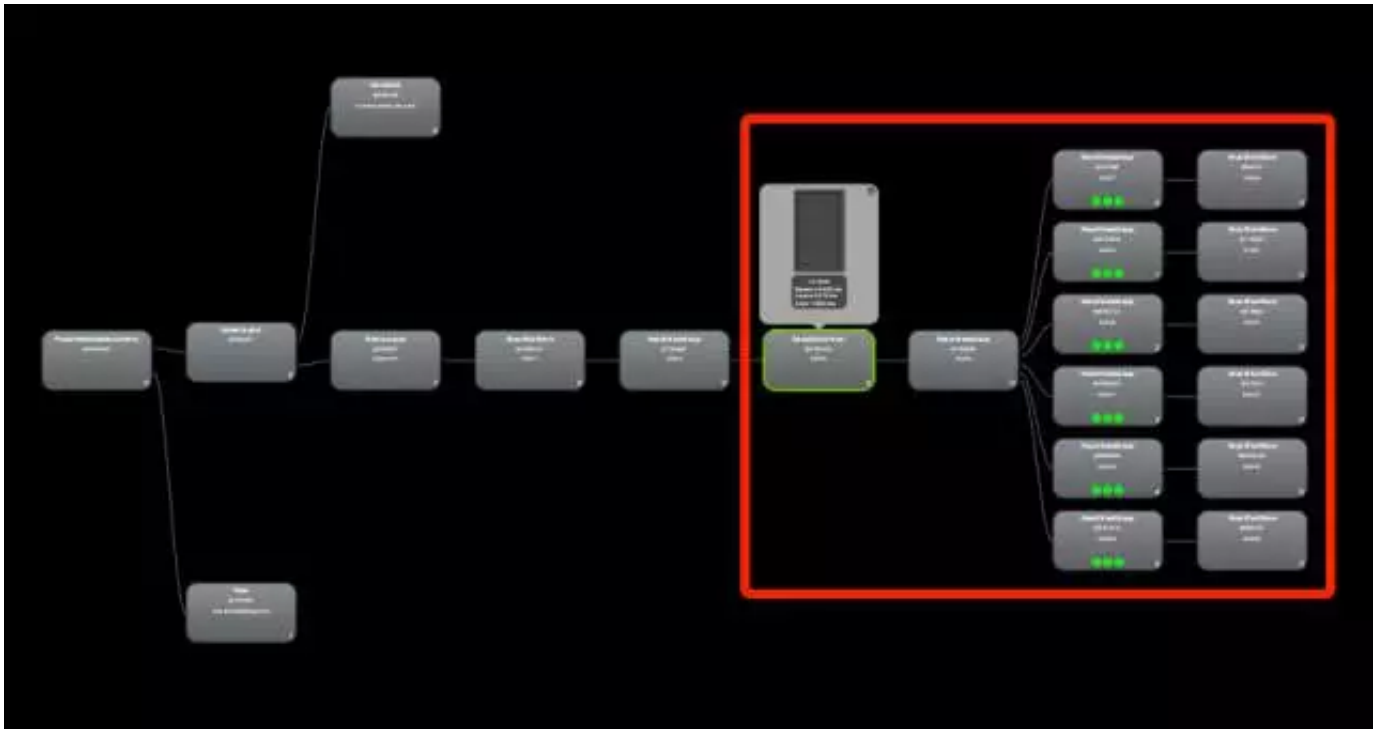
其实，RN 中的 ScrollView 有一个 removeClippedSubviews 属性，表示如果子 View 超出可视区域，是否自动移除，虽然默认是 true。但是也需要子 View 的 overflow: 'hidden'属性配合。所以，给子组件的 style 添加如下属性即可。

```
<View key={"key_" + i} style={styles.child}>
  <Text>{"T" + i}</Text>
</View>;

const styles = StyleSheet.create({
  child: {
```

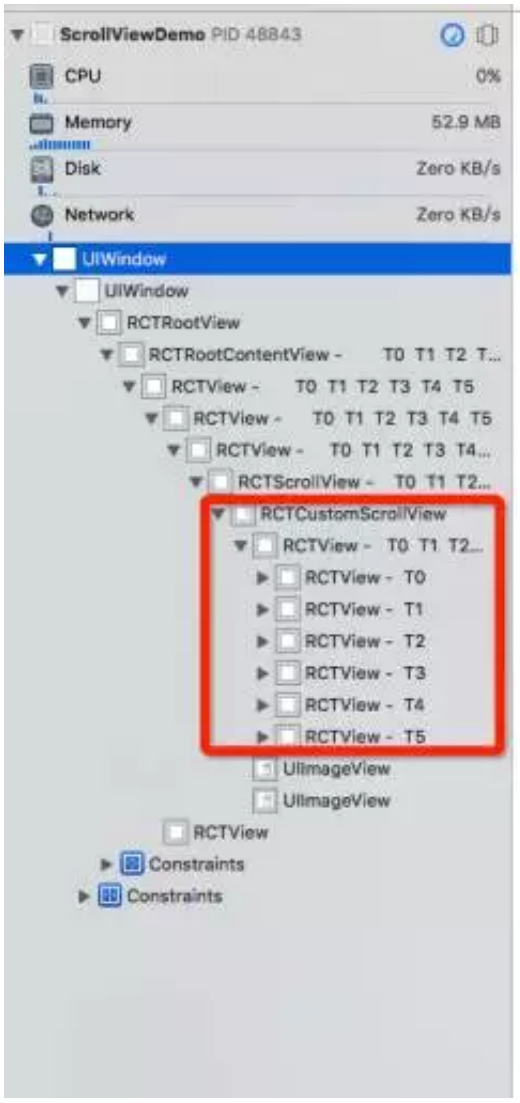
```
    ...
    overflow: 'hidden',
  },
});
```

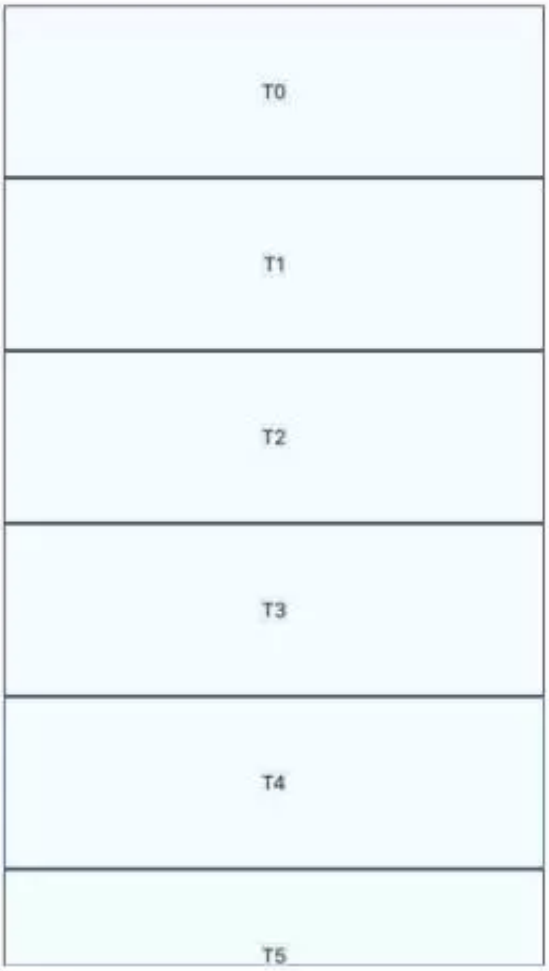
得到的效果是，在使用上完全没有区别，而我们来看一下界面的 Tree View，如下图：



可见，屏幕外的子 View，就被自动从 View 树中移除了。

同时，我们来看一下 iOS 平台上的表现，与 Android 上类似：





这印证了我们前面的结论，RN 自动优化了 Native 平台 ScrollView，在这个层面，我们可以说 RN 比 Native 的性能还要高。

2 性能研究

通过上面的实例，我们可以看到，ScrollView 应该是非常高效的，它使用简单，并且还能按需构建 View 树，高效渲染，有点类似 Native 平台上的 ListView 了，是我心目完美 ScrollView 该有的样子。

但是，之前看到腾讯的 TAT.ronnie 一篇文章 探索 react native 首屏渲染最佳实践，文中提到的优化方法，主要就是针对 ScrollView 的。作者认为，在 ScrollView 中，即使不可见（例如，超出屏幕）的组件还是会绘制的。为了优化 ScrollView 的绘制性能，不可见的组件，应该在 JS 中避免添加到 ScrollView 中。

显然，这与我们前面观察到的结论是矛盾的。但是，作者的通过那样处理，确实优化了显示性能，这是怎么回事呢？为了验证，我们也和文中一样，使用 componentDidMount() 和 componentWillMount() 的时间差衡量显示速度。在 Android 上，测试 ScrollView 的子

组件数量分别为 10，100，1000 的时候，显示的时间，以及 APP 所占用的内存：

子组件数量	加载时间(ms)	占用内存(MB)	绘制时间*(ms)
10	309	19.7	14.666
100	1170	21.9	15.016
1000	9461	26.5	15.025

> * 注，这里的绘制时间，是在 Tree View 中获得的 Draw 时间。

从加载时间看，时间随着子组件的数量线性增加，占用内存也有类似趋势，说明 TAT.ronnie 的改进方法确实是有效的。另外我们也注意到，随着子组件的数量增加，Draw 的时间并没有明显的变化，其实 Measure 和 Layout 时间也没有明显的变化。

说明 ScrollView 虽然有 removeClippedSubviews 属性，也确实在 View Hierarchy 中去掉了不可见的 View。但是组件的加载时间消耗资源还是随着子组件的数量成正比。

3 原因分析

来看一下 RN 中 ScrollView 的相关的源码，主要分析 Android 平台的代码，iOS 类似，就不赘述了。

```
// ScrollView.js
var AndroidScrollView = requireNativeComponent('RCTScrollView', ScrollView, nativeOnlyProps);
var AndroidHorizontalScrollView = requireNativeComponent(
  'AndroidHorizontalScrollView',
  ScrollView,
  nativeOnlyProps
);

var ScrollView = React.createClass({
  render: function() {
    var contentContainer =
      <View
        ...
        removeClippedSubviews={this.props.removeClippedSubviews}
        collapsable={false}>
        {this.props.children}
```

```
      </View>;

    var ScrollViewClass;
    if (Platform.OS === 'ios') {
      ...
    } else if (Platform.OS === 'android') {
      if (this.props.horizontal) {
        ScrollViewClass = AndroidHorizontalScrollView;
      } else {
        ScrollViewClass = AndroidScrollView;
      }
    }

    // 为了简单，忽略有下拉刷新的情况
    return (
      <ScrollViewClass ...>
        {contentContainer}
      </ScrollViewClass>
    );
  }
});
```

JS 部分的代码逻辑很简单。首先把 ScrollView 所有子组件包装在一个 View contentContainer 中，并继承设置了 removeClippedSubviews 属性。根据 ScrollView 是否是水平方向，决定是用 RCTScrollView 或者 AndroidHorizontalScrollView Native 组件来包含 contentContainer。

所以，我们先来看 RCTScrollView 本地组件对应的代码（AndroidHorizontalScrollView 原理也类似）。JS 中的 RCTScrollView 组件由 com.facebook.react.views.scroll.ReactScrollViewManager 提供，具体的 View 的实现是 com.facebook.react.views.scroll.ReactScrollView。

其中 ReactScrollViewManager 是最基础的 ViewManager 的实现，导出了一些属性和事件。ReactScrollView 则继承于 android.widget.ScrollView，并实现了 ReactClippingViewGroup 接口。关于 Scroll 事件相关的代码我们先忽略，我主要关心 View 绘制相关的代码。主要在下面这段代码：

```
@Override
public void updateClippingRect() {
```

```
  if (!mRemoveClippedSubviews) {
    return;
  }
  ...
  View contentView = getChildAt(0);
  if (contentView instanceof ReactClippingViewGroup) {
    ((ReactClippingViewGroup) contentView).updateClippingRect();
  }
}
```

可见，如果不开启 mRemoveClippedSubviews，它就和普通的 ScrollView 一样，否者，它就会调用了它的第一个（也是唯一的一个）子 View 的 updateClippingRect() 方法。从上面的 JS 中我们可以看到，它的第一个子元素应该就是一个 View 组件，对应的 Native 的控件就是 ReactViewGroup。 ReactViewGroup 是 RN for Android 中最基础的控件，它直接继承于 android.view.ViewGroup：

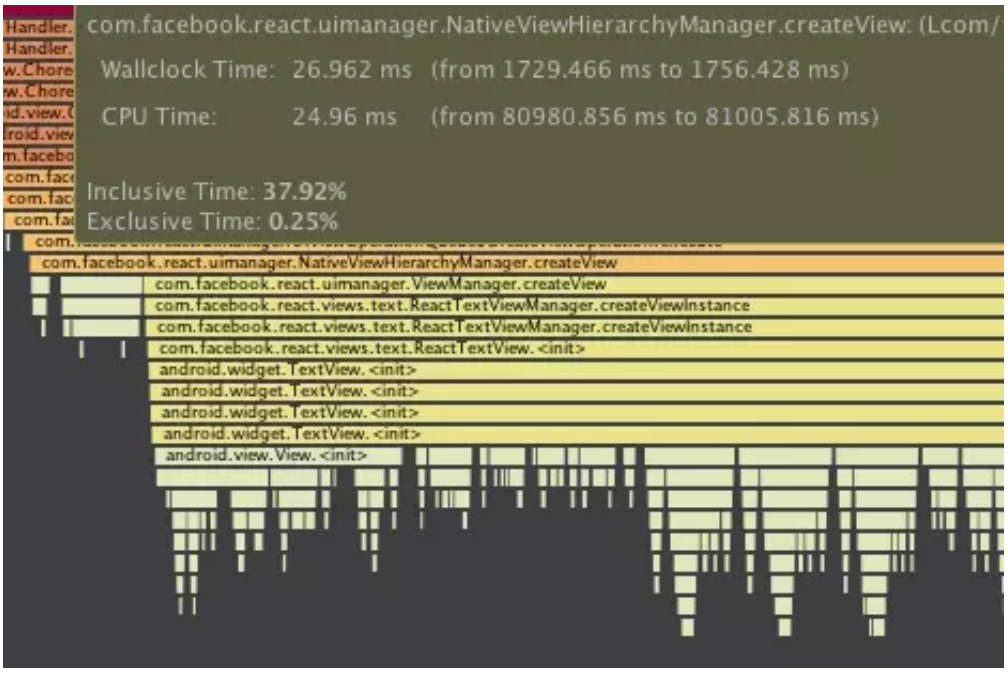
```
public class ReactViewGroup extends ViewGroup implements
    ReactInterceptingViewGroup, ReactClippingViewGroup, ReactPointerEventsView, ReactHitSlopView {
  private boolean mRemoveClippedSubviews = false;
  // 用来保存所有子 View 的数组，包括可见和不可见的
  private @Nullable View[] mAllChildren = null;
  private int mAllChildrenCount;
  // 当前 ReactViewGroup 于父 View 相交矩阵，
  // 也就是它自己在父 View 中可见区域
  private @Nullable Rect mClippingRect;
  ...
}
```

在 ReactViewGroup 中实现 removeClippedSubviews 的功能也非常直接，需要更新界面 Layout 的时候，遍历所有的子 View，看子 View 是否在 mClippingRect 区域内，如果在，就通过 addViewInLayout() 方法添加此 View，否者就通过 removeViewsInLayout() 方法移除它。

到这了，我们就可以解释前面的矛盾了。虽然在 ScrollView 的 View Hierarchy 中，会自动移除不显示的 View，但是实际上还是创建了所有的子 View，所以所占内存和加载时间会线性增加。

关于创建所有子 View，我这里可以多分析一下。我们知道在 Android 中，创建 View 的代价是很大的。特别是在 ScrollView 中，所有的子 View 都是同时创建的。如果 ScrollView

中子 View 的数量很多，这样的代价累加起来，对 APP 造成的延迟和卡顿是相当可观的。例如前面的测试中有 1000 个子组件，加载时间竟然长达 9.5 秒。我们用Method Tracing 看一下创建一个子 View 所花的时间，如下图：



这里只是简单的创建一个 TextView 就消耗了大约 25ms 的时间。当然 Tracing 过程本身会拖慢 APP 运行，但是不影响我们的结论。所以 Android 中列表类的控件，都内部支持对 View 的复用，尽量避免创建 View。

通过前面的分析，我们可以得到的结论是：RN 中的 ScrollView 并不像我们想象的那样高性能。

4 ListView

在这里提到 ListView，是因为 RN 中的 ListView 就是基于 ScrollView 的，但是有一些优化。这里简要介绍一些 ListView 的原理。

ListView 其实是对 ScrollView 的一个封装，对应到 Native 平台，和 ScrollView 的表现一模一样。但是 ListView 在显示列表内容的时候，会根据滑动距离，逐步向 ScrollView 中添加子组件（通过调用 renderRow() 方法）。注意到 ListView 有 initialListSize 属性，表示第一次加载的时候添加多少个子项，默认是 10，还有 pageSize 属性，表示每次需要添加的时候，增加多少个子项，默认是 1。

通过上面的分析我们可以看到，ListView 在第一次加载的时候，不论你的列表有多大，默认最多加载 initialListSize 个子项，所以能保证启动速度，如果还没有充满，或者在向下滑动过程中，再组件添加子项。这样的操作似乎比较合理，但是注意到，整个操作中，会逐渐

向 ListView 中添加子项，新出现的子项，都是通过创建新的 View，而完全没有复用的过程。所以，如果在应用中，ListView 中的子项数量特别多，ListView 往下滑动的过程中，内存会逐渐上涨的。

值得一提的是，ListView 提供了 renderScrollComponent，可以使用其他 Scroll 组件来替换 ScrollView，并且 RecyclerViewBackedScrollView 组件来作为备选。看到这个名字我很欣喜，说明它支持子项的回收复用（Recycler）。首先，看到 iOS 的实现 RecyclerViewBackedScrollView.ios.js，其实它就是 ScrollView，并没有实现所谓的复用，失望了一半。继续看 Android 的实现，它实际上是对应 Native 的 com.facebook.react.views.recyclerview.AndroidRecyclerViewBackedScrollView，它继承与 Android 的 RecyclerView。看到这里，如果使用这种方法，我直观感觉 RN 的 ListView 性能在 Android 上表现应该会比 iOS 好。

我们继续来看它是怎么实现回收复用的，AndroidRecyclerViewBackedScrollView 内部实现了一个 RecyclerView.Adapter，如下：

```
static class ReactListAdapter extends Adapter<ConcreteViewHolder> {

  private final List<View> mViews = new ArrayList<>();

  public void addView(View child, int index) {
    mViews.add(index, child);
    ...
  }

  public void removeViewAt(int index) {
    View child = mViews.get(index);
    if (child != null) {
      mViews.remove(index);
      ...
    }
  }

  @Override
  public ConcreteViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return new ConcreteViewHolder(new RecyclableWrapperViewGroup(parent.getContext()));
  }

  @Override
```

```
  public void onBindViewHolder(ConcreteViewHolder holder, int position) {
    RecyclableWrapperViewGroup vg = (RecyclableWrapperViewGroup) holder.itemView;
    View row = mViews.get(position);
    if (row.getParent() != vg) {
      vg.addView(row, 0);
    }
  }

  @Override
  public void onViewRecycled(ConcreteViewHolder holder) {
    super.onViewRecycled(holder);
    ((RecyclableWrapperViewGroup) holder.itemView).removeAllViews();
  }
}
```

注意到这里有一个 mViews，用来保存所有的子 View，绑定 View 的时候只是简单用一个空的 View（RecyclableWrapperViewGroup）包了一下。这样一来，RecyclerView 完全没有什么起到复用的作用呀！测试一下，确实也是这样，性能问题还是很严重。

这里我们也可以得到一个结论：RN 中的 ListView 也不是我们想象的 ListView 该有的性能。

5 改进方案

通过前面的分析，我们已经知道了 RN 中的 ScrollView 或者 ListView 的性能瓶颈了，同时也有了改进的思路。下面针对各种情况分析：

1. 如果要优化首次加载速度，也就是启动速度：可以参考 TAT.ronnie 的文章中的方法，根据实际情况，最小化 ScrollView 或者 ListView 初始子项数量；
2. 优化内存：因为 ScrollView/ListView 会保存所有子 View 在内存中，因为我们没法删掉子项，但是我们可以尽量减少每个子项所占的内存。例如这个项目 react-native-sglistview，它在子项不可见的时候，就把它退化成一个最基本的 View；
3. 终极解决方案：要真正达到高性能，就需要尽量少的创建 View，要想办法真正重复利用已经创建的子项。目前只有一些想法，待我实现了，再来更新。