

## 11 | 组件实战：如何实现瀑布流？

2022-04-20 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 26:40 大小 24.42M



你好，我是蒋宏伟。

现在，国内购物 App 的首页大都采用了双列瀑布流的布局，假如你的产品经理也想实现同样的瀑布流效果，你在网上找了很多的 **React Native** 列表组件，但都满足不了需求，你会怎么办？你会选择改让产品改方案，还是自己再研究研究？

大概是 2019 年的时候，我们的产品也提了同样的需求，使用 **React Native** 实现瀑布流效果。当时我们有个同事试了很多方案，比如双 **List**、多层嵌套 **List**，性能都很差效果不好，后来我们开会的时候提到了这个问题，我也参与了讨论。

当时我提出了一种思路：改 **RecyclerView** 的源码。我说，**RecyclerView** 的布局原理是绝对定位，每个 item 的 **x**、**y** 轴坐标是根据传入的 **height**、**width** 值算出来的，现在它的布局算法是单列的，我们只要把单列布局算法改成双列布局算法，这件事情应该能成。

后来我们团队的另一个大牛把它落地实现了，实现了一个 React Native 瀑布流页面。

在准备写实战案例的时候，我又想起了当初的这个事情。使用瀑布流的业务场景很多，却没有直接能用的开源方案，但它的实现原理其实并不复杂，应该是一个很好的实战案例。于是我就基于 `RecyclerView` 最新的版本，又实现了一版。

我今天就和你讲讲，我是如何通过修改 `RecyclerView` 组件的源码，实现瀑布流效果的。希望你能通过这次实战，把我们以前学的知识和技巧都用起来。也只有通过实战才能**把知识变成能力**，快和我一起动手试试吧。

## 准备开发调试环境

关于 `RecyclerView` 的基础用法，我在《List》一讲中已经介绍，它主要是通过列表数据 `dataProvider` 来驱动列表项的渲染 `rowRenderer`，并且指定为列表项指定了布局方式 `layoutProvider`。

现在，你需要做的是准备开发调试环境。准备开发调试环境永远是第一步，而且现在我们要调试的是放在 `node_modules` 目录下的第三方组件 `RecyclerView`，所以现在我们要准备**第三方依赖包的开发调试环境**，这怎么准备呢？

在 React Native 中，我们是通过 `import` 导入第三方模块 `RecyclerView` 的：

```
1 import {RecyclerView, DataProvider, LayoutProvider} from 'recyclerlistview'
```

 复制代码

这段代码的意思是从 `recyclerlistview` 模块中导入 `RecyclerView` 组件、`DataProvider` 列表数据类、`LayoutProvider` 布局方式类。

那 `recyclerlistview` 模块到底在哪呢？通常情况下，该模块是 `node_modules/recyclerlistview` 目录下的 `index.js` 文件 `export` 导出的模块。不过第三方库，也可以通过 `package.json` 中的 `main` 字段进行配置。`recyclerlistview` 采用的就是这种配置方法：

```
1 // node_modules/recyclerlistview/package.json
```

 复制代码

```
2 {  
3   "name": "recyclerlistview",  
4   "main": "dist/reactnative/index.js",  
5   ...  
6 }
```

你看，在recyclerlistview的package.json文件中，它通过main参数指定了模块路径dist/reactnative/index.js。

但你再看下 recyclerlistview 的目录：

```
1 node_modules/recyclerlistview/  
2 |— dist  
3 |   |— reactnative  
4 |       |— core  
5 |       |— index.js  
6 |— src  
7 |   |— core  
8 |   |— index.ts  
9 |— package.json
```

 复制代码

你会发现，dist目录下放的是编译后的 .js 文件。也就是说，如果我们直接跑项目，只能调试编译后的 .js 文件，不能调试放在 src 目录中的 .ts 源码。

那怎样才能调试 .ts 的源码文件呢？有一招很简单，修改 recyclerlistview 的导出模块的配置：

```
1 // node_modules/recyclerlistview/package.json  
2  
3 - "main": "dist/reactnative/index.js"  
4 + "main": "src/index.ts"
```

 复制代码

你只需把recyclerlistview/package.json的 main参数改为src/index.ts即可，React Native 会在编译时通过 babel 将 .ts、.tsx 文件编译为 .js 文件再执行。

改完之后，你再重新跑一次yarn start，会遇到一个报错：

```

1 error: node_modules/recyclerlistview/src/core/RecyclerListView.tsx:
2
3 `import debounce = require('lodash.debounce')` is not supported by @babel/plugin-
4
5 Please consider using `import debounce from 'lodash.debounce';` alongside Types
6
7 > 21 | import debounce = require('lodash.debounce');
8       | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

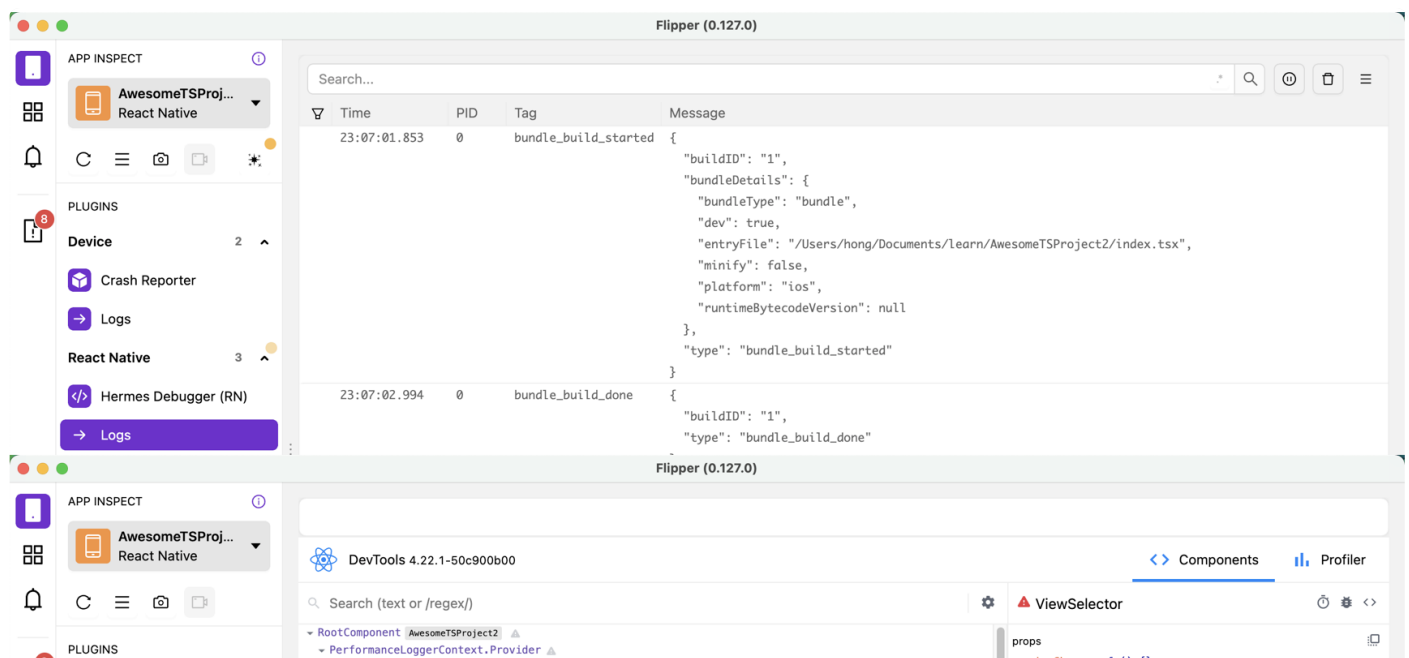
现在我们来分析这个报错信息。你还记得解决具体 **BUG** 的顺序吗？“一推理”、“二分法”、“三问人”。

遇到报错先不用着急去网上搜，先看看红屏的报错提示。第一行报错，说的是 `core/RecyclerListView.tsx` 文件中有报错。第二行，说的是 `@babel/plugin-transform-typescript` 插件不支持 `import = require()` 的导入语法。第三行，直接把建议答案告诉你了，你可以用 `import from` 的语法进行导入。第四行，提示了具体是哪行代码报错了。

你看，这类报错信息都把答案都告诉我们了，我们只要认真读一下就行，我们的“一推理”刚刚开始，就把这个 **BUG** 解决了，都用不着后面的“二分法”和“三问人”。

在你把代码跑起来后，还需要准备一下调试工具 **Flipper** 和 **React Native Tool**。

**Flipper** 调试工具，你只需要下载一下就行，它的功能很强大：



Device2

Crash Reporter

Logs

React Native3

Hermes Debugger (RN)

Logs

React DevTools

AppContainer

RootTagContext.Provider

View ForwardRef

TextAncestorContext.Provider

View key="1" ForwardRef

TextAncestorContext.Provider

App

View ForwardRef

TextAncestorContext.Provider

ViewSelector

TouchableHighlight ForwardRef

TouchableHighlight

View ForwardRef

TextAncestorContext.Provider

Text ForwardRef

TextAncestorContext.Provider

viewChange: f() {}

viewType: 0

new entry: ""

rendered by

App

createLegacyRoot()

react-native-renderer@18.0.0-experimental-568dc3532

source

App.js:74

Flipper (0.127.0)

DevTools 4.22.1-50c900b00

ComponentsProfiler

FlamegraphRanked

1 / 16

Commit information

Priority: Immediate

Committed at: 1.6s

Render duration: 13.7ms

What caused this update?

TouchableHighlight

RootComponent (AwesomeTSProject2)

PerformanceLoggerContext.Provider

AppContainer

RootTagContext.Provider

View (ForwardRef)

TextAncestorContext.Provider

View (ForwardRef) key="1"

TextAncestorContext.Provider

App

View (ForwardRef)

TextAncestorContext.Provider

ViewSelector

RecyclerListView

Touchable...

ScrollView

ScrollView (ForwardRef)

ScrollView

ScrollViewContext.Provider

View (ForwardRef)

TextAncestorContext.Provider

LogBoxState...

LogBoxNotifi...

View (Forwar...

TextAncestor...

View (Forwar...

TextAncestor...

LogBoxLogN...

View (Forwa...

TextAncestor...

LogBoxButton

TouchableW...

View (Forw...

ScrollView

Did not render.

Flipper (0.127.0)

UIApplication

UIWindow addr=0x7f831672ead0

UITransitionView addr=0x7f8317711b0

UIDropShadowView addr=0x7f83176261b0

UIViewController addr=0x7f831762c6e0

RCTRootView addr=0x7f831762dfb0

RCTRootContentView addr=0x7f83177b3a10

RCTView addr=0x7f83177c0d00

RCTView addr=0x7f83177ad500

RCTView addr=0x7f83175bb310

RCTView addr=0x7f8317582df0

RCTScrollView addr=0x7f83167954b0

RCTCustomScrollView addr=0x7f830b01b600

RCTScrollContentView addr=0x7f8316795210

RCTView addr=0x7f8316796f40

RCTView addr=0x7f8317591620

RCTView addr=0x7f83167923b0

RCTView addr=0x7f831d913f30

RCTView addr=0x7f83177ba750

RCTView addr=0x7f831678ae90

RCTView addr=0x7f8317587b40

RCTView addr=0x7f8317587b40

CALayer

backgroundColor: {}

borderColor: {}

borderWidth: 0

cornerRadius: 0

masksToBounds: 0

shadowColor: rgba(0, 0, 0, 1)

shadowOffset: {height: -3, width: 0}

shadowOpacity: 0

shadowRadius: 3

UIView

alpha: 1

backgroundColor: {}

bounds: {origin, size}

center: {x: 192, y: 610.5}

clipsToBounds: false

frame: {origin, size}

layoutMargins: {bottom: 42, left: 8, right: 8, top: 8}

tag: 0

Accessibility

accessibilityHint: ""

accessibilityIdentifier: ""

accessibilityLabel: ""

accessibilityTraits: {UIAccessibilityTraitAdjustable: false, UIAccessibilityTraitImage: false, UIAccessibilityTraitText: false, UIAccessibilityTraitVoice: false}

Flipper (0.127.0)

Search...

Request TimeDomainMethodStatusSizeTime

23:21:39.352images.dog.ceo/breeds/husky/n0211GET20020.0 kB4ms

23:21:39.353images.dog.ceo/breeds/husky/2018GET200443.3 kB4ms

23:21:39.356images.dog.ceo/breeds/husky/MsMikGET20047.6 kB7ms

23:21:39.363images.dog.ceo/breeds/husky/n0211GET20029.4 kB1ms

23:21:39.363images.dog.ceo/breeds/husky/n0211GET20012.5 kB1ms

23:21:39.368images.dog.ceo/breeds/husky/n0211GET20022.4 kB5ms

23:21:39.369images.dog.ceo/breeds/husky/n0211GET20040.4 kB5ms

23:21:39.370images.dog.ceo/breeds/husky/2018GET200359.0 kB5ms

23:21:53.363localhost:8081/statusGET20023 B7ms

23:21:53.372localhost:8081/index.bundleGET20023 B7ms

23:21:53.689dog.ceo/api/breed/husky/imagesGET20011.9 kB1ms

23:21:53.828localhost:8081/symbolicatePOST2002.7 kB19ms

23:21:53.876localhost:8081/assets/node\_module:GET200312 B5ms

23:21:53.921localhost:8081/symbolicatePOST2006.3 kB15ms

23:21:53.947dog.ceo/api/breed/beagle/imagesGET20012.7 kB2ms

23:21:54.213localhost:8081/symbolicatePOST2005.0 kB13ms

23:21:54.216images.dog.ceo/breeds/husky/n0211GET20064.0 kB5ms

23:21:54.216images.dog.ceo/breeds/husky/n0211GET20051.6 kB7ms

23:21:54.216images.dog.ceo/breeds/husky/n0211GET20028.1 kB7ms

23:21:54.217images.dog.ceo/breeds/husky/n0211GET20034.6 kB7ms

23:21:54.222images.dog.ceo/breeds/husky/MsMikGET20047.6 kB6ms

23:21:54.224images.dog.ceo/breeds/husky/n0211GET20029.4 kB4ms

23:21:54.224images.dog.ceo/breeds/husky/n0211GET20022.4 kB5ms

23:21:54.238images.dog.ceo/breeds/husky/n0211GET20040.4 kB6ms

23:21:54.238images.dog.ceo/breeds/husky/n0211GET20012.5 kB7ms

23:21:54.239images.dog.ceo/breeds/husky/n0211GET20038.7 kB7ms

23:21:54.239images.dog.ceo/breeds/husky/n0211GET20029.6 kB9ms

23:21:54.244images.dog.ceo/breeds/husky/2018GET200491.4 kB3177ms

23:21:54.247images.dog.ceo/breeds/husky/n0211GET20020.0 kB3ms

Request

KeyValue

Full URLhttps://dog.ceo/api/breed/beagle/images

Hostdog.ceo

Path/api/breed/beagle/images

Query String

Response Headers

KeyValue

Content-Typeapplication/json

Access-Control-Allow-Headers\*

x-cache-hit14

Alt-Svc

Age0

nel{"success\_fraction":0,"report\_to":"cf-nel","max\_age":604800}

report-to{"endpoints":[{"url":"https://a.nel.cloudflare.com/report/v3?s=cUKXV1E0PH94XzsyldY4515vh%2BrzT%2BdEhgCAz1mBqAltrQm%2FNes7FcS1z8sQxChEa7XQ9ypq6firUT8AJITicEfpetcXnU4A3%2Fo56Gdvvw3%2B%2BqTtGZHdRjBRetzVJAEiB%2BSc"}],"group":"cf-nel","max\_age":604800}



23:21:54.247	images.dog.ceo/breeds/husky/n0211 GET	200	43.7 kB	13ms	Server	cloudflare
23:21:54.250	images.dog.ceo/breeds/husky/n0211 GET	200	54.7 kB	10ms	cf-cache-status	DYNAMIC

这张长截图显示了 Flipper 的功能，从上到下依次是打印日志 Logs、组件树 Components、性能火焰图 Profiler、宿主组件树和布局 Layout、网络请求 Network。它还有一个功能是支持 Hermes 引擎 Debugger，它和浏览器的 Debugger 类似，这里我没有进行截图了。

现在 RecyclerView 源码已经跑起来了，调试环境也已经准备完成，接着下一步就是找到控制布局的关键源码。

## 找到关键源码

要找到关键源码，我们得先从整体上理解源码。很多人读源码没有方法，不知道从哪里入手，甚至有些程序员从来没有读过别人的源码。其实写代码和读代码，就像写文章和读文章的关系一样，没有大量的阅读积累，怎么能写出好的代码呢？

理解 RecyclerView 这类复合组件的源码，我有一个技巧，就是从复合组件 JSX 部分开始切入。

这时我们能直接观察到的是 UI 视图，以 UI 视图为锚点，去理解 JSX 文件就很容易了。在你了解 JSX 之后，再根据状态 state 和属性 props 去推断组件的内部逻辑 f，会容易很多。一个页面，无非也就是由这几个部分组成：

复制代码

```
1 UI/JSX = f(state, props)
```

了解了基本方法，现在我们开始分析 RecyclerView 组件的源码，请你先[在本地打开 `RecyclerView` 的源码文件](#)。下面是我从 RecyclerView 类组件摘出来的 3 个和 JSX 相关的方法：

复制代码

```
1 // node_modules/recyclerview/src/core/RecyclerView.tsx
2 public renderCompat() { // 先调用 render 后调用它
3   return (
4     <ScrollComponent>
5       {this._generateRenderStack()}
6     </ScrollComponent>
7   );
}
```

```

8  }
9
10 private _generateRenderStack(){
11     for(const key in this.state.renderStack){
12         renderedItems.push(this._renderRowUsingMeta(this.state.renderStack[key]))
13     }
14     return renderedItems
15 }
16
17 private _renderRowUsingMeta() {return <ViewRenderer/>}

```

它们分别是：

- `renderCompat` 方法：实际就是类组件的 `render` 方法，它最外层是一个滚动组件 `ScrollComponent`；
- `_generateRenderStack` 方法：循环了状态 `state.renderStack`，生成了若干个 `renderedItems`；
- `_renderRowUsingMeta` 方法：返回的是具体的 `renderedItems`，也就是 `ViewRenderer` 容器元素。

当你把 UI 视图和 JSX 部分联系在一起时你就明白了，`RecyclerView` 渲染出的 UI 页面，是由一个滚动容器和若干个 `ViewRenderer` 容器组成的。

我们还是回到最基础的 UI 公式：

```
1 UI/JSX = f(state, props)
```

 复制代码

现在我们已知 **JSX** 是 `ScrollView + View`，已知 **state** 是用 `for...in` 循环的对象 `renderStack`，还已知 `RecyclerView` 的三个必传 **props**：列表数据 `dataProvider(dp)`、列表项的布局方法 `layoutProvider`、列表项的渲染函数 `rowRenderer`。

这时虽然你还不知道组件内部逻辑 **f** 具体是什么，但是你应该已经把握住了函数的“入口”和“出口”，你知道放进去的入参是什么，能够产出的 UI 又是什么。知道了“入口”“出口”的特点后，再从两端往中间推理，理解组件内部逻辑 **f** 就会变得简单很多。

这时候，你可能会有一些关于内部逻辑f的问题，你或许想问state.renderStack和三个props 是怎么控制 JSX 的？

那我们就要再仔细读一下\_renderRowUsingMeta中的代码了：

 复制代码

```
1 private _renderRowUsingMeta(itemMeta: RenderStackItem): JSX.Element | null {
2   const dataIndex = itemMeta.dataIndex;
3   const data = this.props.dataProvider.getDataForIndex(dataIndex);
4   const type = this.props.layoutProvider.getLayoutTypeForIndex(dataIndex);
5   return (
6     <ViewRenderer
7       data={data}
8       layoutType={type}
9       index={dataIndex}
10      layoutProvider={this.props.layoutProvider}
11      childRenderer={this.props.rowRenderer}
12    />
13  );
14 }
```

在这段源码中，我只标记出了 state、props 和 ViewRenderer 三个部分，即便只有这些代码片段，你可以猜出它的大致逻辑。

状态 state.renderStack[key] 就是 itemMeta，每个 itemMeta 的 dataIndex 是不一样的，通过 dataIndex 从列表数据 dataProvider 和布局方法 layoutProvider 中，选取了对应项的数据 data 和布局类型 type，并将这些值和列表项的渲染函数 rowRenderer 都赋值给了 ViewRenderer的childRenderer属性。

读完这段源码片段，你大概能够补全此段代码的内部逻辑f。ViewRenderer 是一个容器，内部装的是你传给它的渲染函数 rowRenderer 方法，并且按照你指定的数据data、类型type进行渲染。

因为ViewRenderer是你指定的列表项rowRenderer的父容器，父容器的位置决定了你列表项的位置。这时候，你再读一遍\_renderRowUsingMeta的源码：

 复制代码

```
1 private _renderRowUsingMeta(itemMeta: RenderStackItem): JSX.Element | null {
2   const dataSize = this.props.dataProvider.getSize();
```



```

3      const dataIndex = itemMeta.dataIndex;
4      const itemRect = (
5      this._virtualRenderer.getLayoutManager() as LayoutManager
6      ).getLayouts()[dataIndex];
7      return (
8          <ViewRenderer
9              key={key}
10             data={data}
11             x={itemRect.x}
12             y={itemRect.y}
13             layoutType={type}
14             index={dataIndex}
15             layoutProvider={this.props.layoutProvider}
16             onSizeChanged={this._onViewContainerSizeChange}
17             childRenderer={this.props.rowRenderer}
18             height={itemRect.height}
19             width={itemRect.width}
20         />
21     );
22 }

```

在这个源码片段中，这些由LayoutManager类的getLayouts方法生成的 x/y/width/height 属性，就是决定你列表项布局方式的关键源码。

但 getLayouts 到底是什么呢？请你打开 LayoutManager 类的源码：

 复制代码

```

1 // node_modules/recyclerlistview/src/core/layoutmanager/LayoutManager.ts
2 public getLayouts(): Layout[] {
3     return this._layouts;
4 }
5
6 public relayoutFromIndex(): void {
7     let startX = 0;
8     let startY = 0;
9     let maxBound = 0;
10
11     for () {
12         oldLayout = this._layouts[i];
13         if () {
14             itemDim.height = oldLayout.height;
15             itemDim.width = oldLayout.width;
16             maxBound = 0;
17         }
18         while () {
19             startX = 0;
20             startY += maxBound;
21         }
22     }
23 }

```

```
22         maxBound = Math.max(maxBound, itemDim.height);
23         this._layouts.push({ x: startX, y: startY, height: itemDim.height,
24
25     }
26 }
27 }
```

上述的代码片段是 `getLayouts` 方法和设置 `this._layouts` 的 `relayoutFromIndex` 方法。大致扫一眼，你就能明白 `relayoutFromIndex` 方法通过一堆计算，计算出了实现单列布局的 `x/y/height/width` 值，然后把它们作为对象 `push` 到了 `this._layouts`。

而 `ViewRenderer` 根据 `this._layouts` 把你的列表项，渲染到了指定的位置上。因此，我们要想实现双列瀑布流布局，就得理解和修改 `relayoutFromIndex` 方法。

## 修改源码

在“找到关键源码”这一步，我们读源码其实只要有宏观上的理解就行了，但要“修改别人源码”就需要更微观上的理解了。

我说的宏观上理解源码，讲究的是速度，大致理解就行，细节上有点小偏差也不要紧。但微观上理解源码，讲究的是准确，我们要改别人的源码，理解要是不准确，改起来肯定容易出问题。

在提高理解的准确性上，我是这么做的。首先我会使用断点工具，一行一行地执行代码，并对上下文中的变量进行一些“终极拷问”：“变量从哪来”、“变量用到哪里去”、“变量的意义是什么”，再把自己的理解马上备注起来，不然容易忘。

在微观理解上，我们也要找到切入点。比如，在理解 `relayoutFromIndex` 方法时，我找的切入点就是设置列表项的 `x/y`。设置 `x/y` 的核心代码如下：

 复制代码

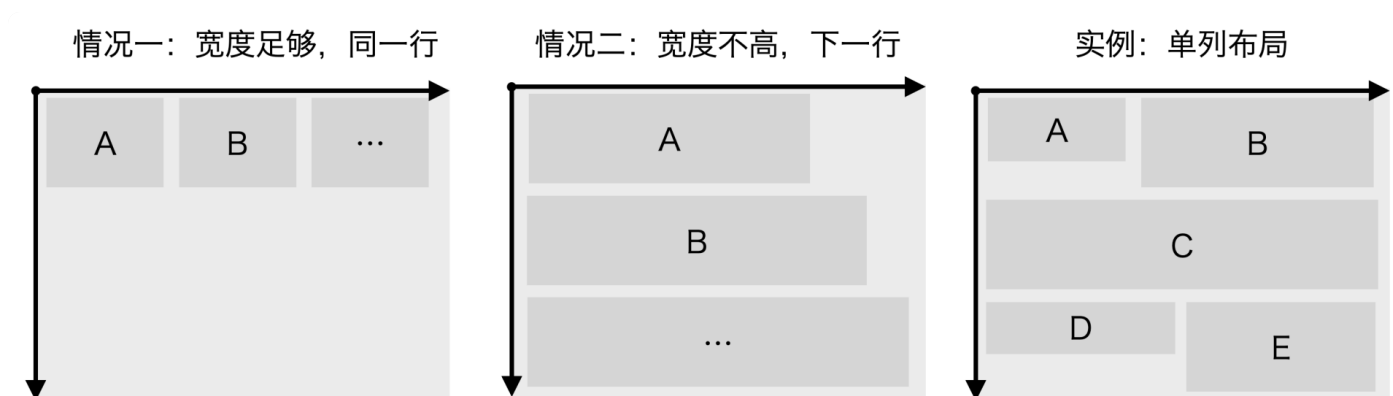
```
1  public relayoutFromIndex(itemCount: number): void {
2      // 新 item x y 坐标
3      let startX = 0;
4      let startY = 0;
5      // 记录当前一行最高元素的高度
6      let maxBound = 0;
7
8      for (let i = 0; i < itemCount; i++) {
```

```

9      // 如果当前多个 item 宽度之和超过屏幕宽度就换行
10
11      while (!this._checkBounds(startX, startY, this._layouts[i])) {
12          // 将实际 x 坐标设置为 0
13          startX = 0;
14          // 将实际 y 坐标设置增加上一行最高 item 的高度
15          startY += maxBound;
16          maxBound = 0;
17      }
18
19      // 设置新的宽高
20      this._layouts.push({ x: startX, y: startY, height: itemDim.height, width:
21
22      // 记录当前一行最高 item 的高度
23      maxBound = Math.max(maxBound, this._layouts[i].height);
24      // 默认情况下: 下一个 item 的初始化的 x 坐标
25      startX += itemDim.width;
26  }
27  }
28  private _checkBounds(
29      itemX: number,
30      itemY: number,
31      itemDim: Dimension,
32  ): boolean {
33      return itemX + itemDim.width <= this._window.width;
34  }

```

虽然我已经把代码精简并写了备注，但理解起来可能还是有点难度，所以我还给你配了单列布局的原理示意图：



单列布局的原理是什么呢？

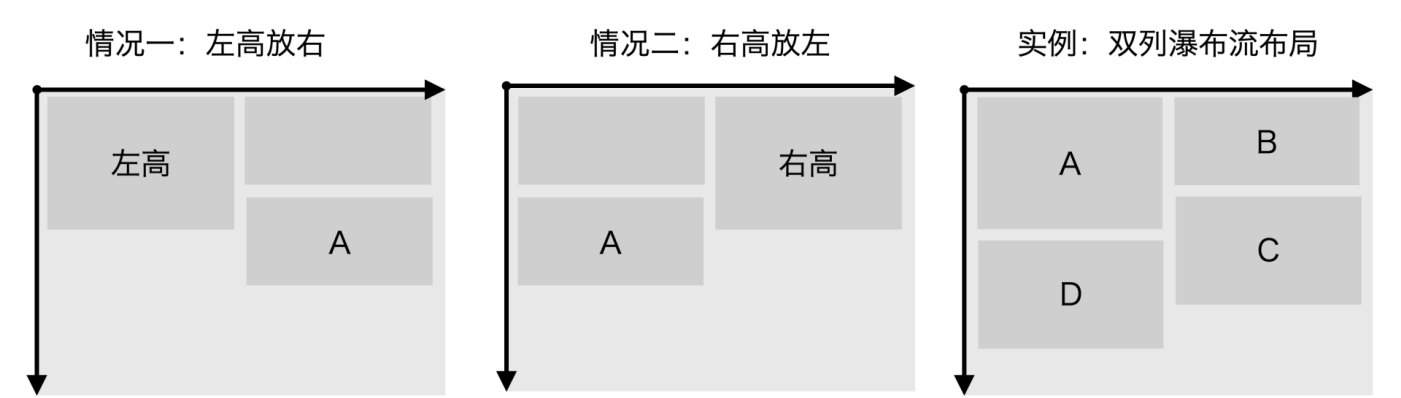
从代码层面看，它对你传入的列表项进行for循环遍历，并通过 `_checkBounds` 方法来判断。如果当前遍历的列表项宽度和当前一行已有列表项的宽度之和，不超过屏幕宽度，也就是 `itemX + itemDim.width <= this._window.width`，那么就跳过 while 循环，直接使

用同一行前几个列表项的宽度之和`startX += itemDim.width`，作为当前列表项的`x(startX)`坐标。也就是情况一：**宽度足够，放到同一行。**

如果`_checkBounds`判断，同一行剩余宽度不够了，那么就进入 `while` 循环，将当前列表项的 `x(startX)` 坐标设置为 0，`y` 坐标设置增加上一行最高 `item` 的高度`maxBound`。也就是情况二：**宽度不够，放到下一行。**

我还在图中给你画了一个单列布局的例子。第一行 **A**、**B** 列表项宽度正好占满整个屏幕宽度，所以列表项 **C** 得再起一行，其 `x` 坐标为 0，其 `y` 坐标为 **A** 的 `y` 坐标和 **B** 的高度 `height` 之和。列表项 **C** 横向独占了一行，所以 **D**、**E** 就只能放到下一行了。整体上看，`RecyclerView` 实现的还是一种单列布局，只不过同一行中可以放置多个列表项。

理解完 `RecyclerView` 的单列布局源码后，接下来就要设计我们自己的双列瀑布流布局了。我给你画了一张瀑布流的示意图：



双列瀑布流布局只有两种情况，第一种情况是如果左边已有列表项的高度之和 `startLeftY` 大于右边已有列表项的高度之和 `startRightY`，那么下一个列表项就要放右边。第二种情况则刚好相反，我们需要把下一个列表项放在左边。简单来说就是，**左高放右、右高放左。**

我同样给你举了一个例子，你可以对照双列瀑布流布局的图片看一下。起始时左右两边一样高，所以先是左 **A**，再是右 **B**。接下来，由于左边比右边高，所以再是右 **C**，最后是左 **D**。如果是单列布局，**C** 应该放在左边，**D** 应该放在右边，这就是双列瀑布流布局和单列布局不同之处。

双列瀑布流实现的核心代码如下：

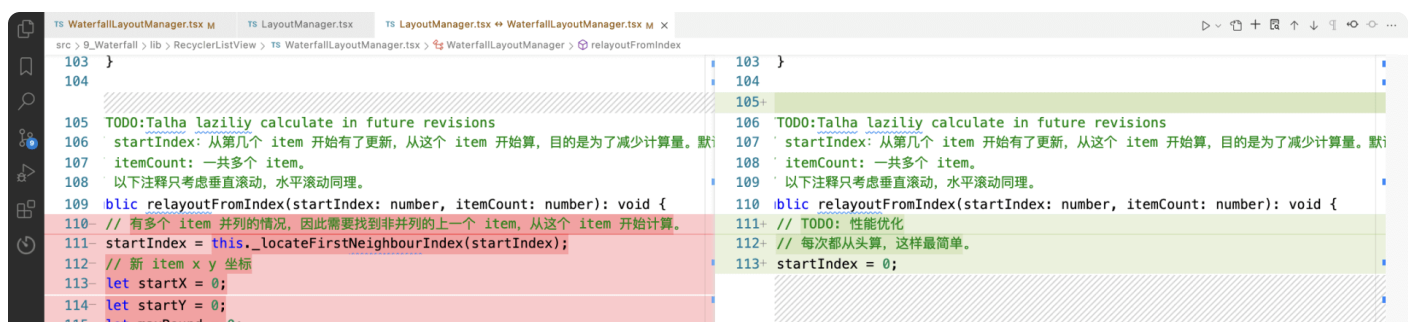
```

1 public relayoutFromIndex(startIndex: number, itemCount: number): void {
2
3     // 假设：每个 item 的宽度为 1/2*window.width 两种情况
4     const halfWindowWidth = this._window.width / 2;
5
6     let startLeftY = 0; // 左边所有 item 的高度之和
7     let startRightY = 0; // 右边所有 item 的高度之和
8
9     let startX = 0; // 新增 item 的 X
10    let startY = 0; // 新增 item 的 Y
11
12    for (let i = startIndex; i < itemCount; i++) {
13        itemDim.height = oldLayout.height;
14        itemDim.width = halfWindowWidth;
15
16        // 保证一行中所有的 item 宽度之和不超过屏幕宽度，超过就换行
17        if (startLeftY > startRightY) {
18            startX = halfWindowWidth;
19            startY = startRightY;
20            startRightY += itemDim.height;
21        } else {
22            startX = 0;
23            startY = startLeftY;
24            startLeftY += itemDim.height;
25        }
26
27        // 如果是 item 是新增的，在添加新的 layout
28        this._layouts.push({x: startX, y: startY, height: itemDim.height, width:
29    }

```

首先，双列瀑布流有一个假设，假设每个列表项的宽度为屏幕的一半。其次，我们还需要记录左边的高度之和startLeftY和右边的高度之和startRightY。在for遍历列表项时，如果左边高 startLeftY > startRightY，那么当前列表项放右边startX = halfWindowWidth，否则当前列表项放左边startX = 0，同时记录最新的左边 / 右边高度之和。最后把当前列表项 push 到 this.\_layouts 中。

将单列布局改为双列瀑布流布局，改动的代码量很少，你可以现在就动手试一试。我也将我改动的代码前后对比图，放在了下面，你可以参考一下：



```

115- let maxBound = 0;
116-
117- const startVal = this._layouts[startIndex];
118-
119- // 初始化新 item x y 坐标 = 上一个 item x y 坐标
120- if (startVal) {
121-   startX = startVal.x;
122-   startY = startVal.y;
123-   // 初始化整体 scrollView 的高度
124-   this._pointDimensionsToRect(startVal);
125- }
126-
127- const oldItemCount = this._layouts.length;
128- // 初始化新 item 的宽高
129- const itemDim = {height: 0, width: 0};
130-
131- let itemRect = null;
132- let oldLayout = null;
133-
134- for (let i = startIndex; i < itemCount; i++) {
135-   // 旧 item 的 layout (x/y/宽/高)
136-   oldLayout = this._layouts[i];
137-   // 调用 LayoutProvider 第一个入参函数
138-   // LayoutProvider( () => return 'type', fn2)
139-   const layoutType = this._layoutProvider.getLayoutTypeForIndex(i);
140-   // 在高度不确定的动态布局情况下, 业务会开启 forceNonDeterministicRendering, 此时
141-   // 第一次取 LayoutProvider 第二个入参函数返回值 (走 else),
142-   // 第二次 ViewRenderer _onViewContainerSizeChange 会调用 layoutManager 重写
143-   if (
144-     oldLayout &&
145-     oldLayout.isOverridden &&
146-     oldLayout.type === layoutType
147-   ) {
148-     itemDim.height = oldLayout.height;
149-     itemDim.width = oldLayout.width;
150-   } else {
151-     // 调用 LayoutProvider 第二个入参函数, 设置 itemDim
152-     // LayoutProvider(fn1,(type, itemDim, index) => { itemDim.width = 100; i
153-     this._layoutProvider.setComputedLayout(layoutType, itemDim, i);
154-   }
155-   // 保证当前 item 最大宽度不超过屏幕宽度
156-   this.setMaxBounds(itemDim);
157-   // 保证一行中所有的 item 宽度之和不超过屏幕宽度, 超过就换行
158-   while (!this._checkBounds(startX, startY, itemDim, this._isHorizontal)) {
159-     if (this._isHorizontal) {
160-       startX += maxBound;
161-       startY = 0;
162-       this._totalWidth += maxBound;
163-     } else {
164-       startX = 0;
165-       startY += maxBound;
166-       this._totalHeight += maxBound;
167-     }
168-     maxBound = 0;
169-   }
170-
171-   // 下一个 item 增加的 y 轴距离 (如果是一行则不增加) = 当前一行最高 item 的高度
172-   // (当前一行会跳过 this._checkBounds && startY += maxBound, 所以当前一行实际没
173-   maxBound = this._isHorizontal
174-     ? Math.max(maxBound, itemDim.width)
175-     : Math.max(maxBound, itemDim.height);
176-
177-   //TODO: Talha creating array upfront will speed this up
178-   // 如果是 item 是新增的, 在添加新的 layout
179-   if (i > oldItemCount - 1) {
180-     this._layouts.push({
181-       x: startX,
182-       y: startY,
183-       height: itemDim.height,
184-       width: itemDim.width,
185-       type: layoutType,
186-     });
187-     // 如果是 item 是已经渲染过一次的, 已经记住原有 layout, 重新赋值
188-   } else {
189-     itemRect = this._layouts[i];
190-     itemRect.x = startX;
191-     itemRect.y = startY;
192-     itemRect.type = layoutType;
193-     itemRect.width = itemDim.width;
194-     itemRect.height = itemDim.height;
195-   }
196-
197-   // 下一个 item 的初始化的 x 坐标
198-   if (this._isHorizontal) {
199-     startY += itemDim.height;
200-   } else {
201-     startX += itemDim.width;
202-   }
203- }
204- // 如果 list 的长度减少了, 也就是商品数量减少了,
205- if (oldItemCount > itemCount) {
206-   this._layouts.splice(itemCount, oldItemCount - itemCount);
207- }
208- // 设置 scrollView 的最终高度
209- this._setFinalDimensions(maxBound);
210-
211-
212-ivate _pointDimensionsToRect(itemRect: Layout): void {
213-  if (this._isHorizontal) {
214-    this._totalWidth = itemRect.x;
215-  } else {
216-
114
115- // 假设: 每个 item 的宽度为 1/2*window.width 两种情况
116- const halfWindowWidth = this._window.width / 2;
117-
118- let startLeftY = 0; // 左边所有 item 的高度之和
119- let startRightY = 0; // 右边所有 item 的高度之和
120-
121- let startX = 0; // 新增 item 的 X
122- let startY = 0; // 新增 item 的 Y
123-
124- // 重新计算 scrollView 的高度
125- this._totalHeight = 0;
126- this._totalWidth = 0;
127-
128- const oldItemCount = this._layouts.length;
129- // 初始化新 item 的宽高
130- const itemDim = {height: 0, width: 0};
131-
132- let itemRect = null;
133- let oldLayout = null;
134-
135- for (let i = startIndex; i < itemCount; i++) {
136-   // 旧 item 的 layout (x/y/宽/高)
137-   oldLayout = this._layouts[i];
138-   // 调用 LayoutProvider 第一个入参函数
139-   // LayoutProvider( () => return 'type', fn2)
140-   const layoutType = this._layoutProvider.getLayoutTypeForIndex(i);
141-   // 在高度不确定的动态布局情况下, 业务会开启 forceNonDeterministicRendering, 此时
142-   // 第一次取 LayoutProvider 第二个入参函数返回值 (走 else),
143-   // 第二次 ViewRenderer _onViewContainerSizeChange 会调用 layoutManager 重写
144-   if (
145-     oldLayout &&
146-     oldLayout.isOverridden &&
147-     oldLayout.type === layoutType
148-   ) {
149-     itemDim.height = oldLayout.height;
150-   } else {
151-     // 调用 LayoutProvider 第二个入参函数, 设置 itemDim
152-     // LayoutProvider(fn1,(type, itemDim, index) => { itemDim.height = 300; i
153-     this._layoutProvider.setComputedLayout(layoutType, itemDim, i);
154-   }
155-   itemDim.width = halfWindowWidth;
156-
157-   // 保证一行中所有的 item 宽度之和不超过屏幕宽度, 超过就换行
158-   if (startLeftY > startRightY) {
159-     startX = halfWindowWidth;
160-     startY = startRightY;
161-     startRightY += itemDim.height;
162-   } else {
163-     startX = 0;
164-     startY = startLeftY;
165-     startLeftY += itemDim.height;
166-   }
167-
168-   // 如果是 item 是新增的, 在添加新的 layout
169-   if (i > oldItemCount - 1) {
170-     this._layouts.push({
171-       x: startX,
172-       y: startY,
173-       height: itemDim.height,
174-       width: itemDim.width,
175-       type: layoutType,
176-     });
177-     // 如果是 item 是已经渲染过一次的, 已经记住原有 layout, 重新赋值
178-   } else {
179-     itemRect = this._layouts[i];
180-     itemRect.x = startX;
181-     itemRect.y = startY;
182-     itemRect.type = layoutType;
183-     itemRect.width = itemDim.width;
184-     itemRect.height = itemDim.height;
185-   }
186-
187-   // 如果 list 的长度减少了, 也就是商品数量减少了,
188-   if (oldItemCount > itemCount) {
189-     this._layouts.splice(itemCount, oldItemCount - itemCount);
190-   }
191-   // 设置 scrollView 的最终高度
192-   this._totalHeight = Math.max(startLeftY, startRightY);
193-   this._totalWidth = this._window.width;
194-
195-
196-ivate _pointDimensionsToRect(itemRect: Layout): void {
197-  if (this._isHorizontal) {
198-    this._totalWidth = itemRect.x;
199-  } else {

```



## 保存修改

现在，我们来到了最后一步保存修改的源码。

在修改完 `node_modules` 中的源码后，如果不进行保存，很有可能就会丢失。并且，有时候我们需要和同事进行合作，同事也需要我们修改后的代码，又或者是在使用上线平台进行打包时，也需要将修改后的 `node_modules` 源码同步给上线平台一份。本地修改 `node_modules` 源码后，不保存、不同步肯定会出线上问题。

怎么把修改好的 `node_modules` 代码保存呢？有三种思路：

第一种**直接复制源码**。但复制源码后续想要升级 `Recyclerview` 的版本会非常困难，每次升级可能面临的是一次重新改造。

第二种**在运行时进行修改**。这种方法对源码的侵入性小，但每次升级前我们还是需要手动检查一下的，不然相关代码逻辑有变化，我们的修改就会受到影响。

我在这次实战中，采用的就是在运行时进行修改的方案。我观察了一下 `Recyclerview` 的代码，它的代码风格是面向对象的编程风格，几乎把所有的内部类都暴露出来了。

但由于它 `LayoutManager` 类的所有属性是私有属性，我没办法通过继承的方式读取到 `LayoutManager` 的私有属性。

因此我复制了 `LayoutManager` 和 `layoutProvider` 类，并将其重写为 `WaterfallLayoutManager` 和 `WaterfallLayoutProvider`。

当你的列表是单列布局时，就应该使用 `layoutProvider` 类，当你的列表是双列瀑布流布局时，就可以使用我创建的 `WaterfallLayoutProvider` 类。

第三种**在编译时修改**。这里利用的是 `patch-package` 即时修复第三方 `npm` 包的能力，它的原理是先对你的修改进行保存，然后在你每次安装 `npm` 包的时候把你原先的修改给注入进去，也就是 `patch package`。它是侵入式的修改方式，步骤如下：

在修改完 `node_modules` 目录下的 `RecyclerListview` 文件后，你直接运行如下命令：

 复制代码

```
1 $ npx patch-package some-package
```

这时你修改的代码就会以 `patch` 文件的形式进行保存，`patch` 文件的示例代码如下：

 复制代码

```
1 diff --git a/node_modules/recyclerlistview/src/core/layoutmanager/LayoutManager
2 index e9454a4..3168330 100644
3
4 --- a/node_modules/recyclerlistview/src/core/layoutmanager/LayoutManager.ts
5 +++ b/node_modules/recyclerlistview/src/core/layoutmanager/LayoutManager.ts
6
7 @@ -95,75 +95,113 @@ export class WrapGridLayoutManager extends LayoutManager {
8     }
9   }
10
11 -   public relayLayoutFromIndex(startIndex: number, itemCount: number): void {
12
13 +   // startIndex: 从第几个 item 开始有了更新，从这个 item 开始算，目的是为了减少计算量。默i
14 +   // itemCount: 一共多少个 item。
15 +   // 以下注释只考虑垂直滚动，水平滚动同理。
16 +   public relayLayoutFromIndex(startIndex: number, itemCount: number): void {
17
18     private _pointDimensionsToRect(itemRect: Layout): void {
19       if (this._isHorizontal) {
```

那如果别人想用你瀑布流版本的 `RecyclerListview` 怎么办呢？首先，你需要修改 `package.json` 文件：

 复制代码

```
1 // package.json
2 "scripts": {
3 +   "postinstall": "patch-package"
4 }
```

然后将修改后的 `package.json` 和前面自动生成的 `patch` 文件用 `Gitlab/GitHub` 保存起来。

这样，你同事下载最新代码，再执行 `npm install` 或 `yarn` 命令后，就会自动触发 `patch-package` 命令。`patch-package` 命令会利用你生成的 `patch` 文件，将官方的 `RecyclerListview` 修改成你的瀑布流版本的 `RecyclerListview`。

一般来说，无论是快速修改第三方组件源码，还是修改 React Native 的 JavaScript 层的源码，我都不建议使用第一种直接复制源码的方式。我会**优先考虑在运行时的修改方法**，通常该方案改动最小、侵入性也最小。**如果运行时方案改不了，我才会考虑有侵入性的编译时的 `patch-package` 方案。**

## 总结

在前面的课程中，我讲的大多是概念性的知识，要消化这些概念性的知识就必须要有练习，所以我在每节课中都给你留了一道实操的练习题，目的就是帮你把知识内化为能力。这一讲中，我准备的实战案例也是为了让你把前几讲中学习到的知识灵活运用起来。

首先你需要提前准备好写代码时会用到调试工具 **Flipper**，并灵活运行“一推理”、“二分法”、“三问人”的思路来解决过程中遇到的问题。

在理解别人的组件代码时，利用 `UI/JSX = f(state, props)` 这个最基本 React/React Native 原理，先找到实现 UI 的 JSX 部分，再找到 `state`、`props`，然后再理解逻辑 `f` 的部分。

在修改别人的逻辑代码时，先通过调试工具来理解各个变量上下文含义，理清楚别人的逻辑后，再根据自己目的进行修改。

最后要意识到，你修改的是别人的源码，你可以通过运行时、编译时两种方案把其保存下来。

## 附加材料

1. [🔗 patch-package](#) 可以帮你保存对第三方模块的问题修复。
2. 本节课的源码，我放在了 [🔗 GitHub](#) 中。


## 作业

1. 请你根据这节课的资料，实现一个三列瀑布流布局。
2. 你觉得阅读源码，有什么意义？

欢迎在评论区写下你的想法。我是蒋宏伟，咱们下节课见。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 10 | Debug：解决 BUG 思路有哪些？

## 精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。